The enclosed sheets are Sections 11 through 17 and Appendix A of CGTM No. 17, the revised Table of Contents, and errata for Sections 1 through 10.

<table>
<thead>
<tr>
<th>Page</th>
<th>Line</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-5</td>
<td>2</td>
<td>... odd-numbered ...</td>
</tr>
<tr>
<td>5-2</td>
<td>5</td>
<td>... and the 4-bit ...</td>
</tr>
<tr>
<td>5-2</td>
<td>7b</td>
<td>... register 1110 is ...</td>
</tr>
<tr>
<td>5-3</td>
<td>4</td>
<td>... is 430A7468</td>
</tr>
<tr>
<td>5-3</td>
<td>6</td>
<td>... namely 716 ...</td>
</tr>
<tr>
<td>5-3</td>
<td>7</td>
<td>... means that R7 ...</td>
</tr>
<tr>
<td>5-3</td>
<td>13</td>
<td>... (from R7) ...</td>
</tr>
<tr>
<td>5-3</td>
<td>15</td>
<td>... (from R10) ...</td>
</tr>
<tr>
<td>6-3</td>
<td>3b</td>
<td>...</td>
</tr>
<tr>
<td>9-7</td>
<td>5</td>
<td>... of his program ...</td>
</tr>
<tr>
<td>10-1</td>
<td>12</td>
<td>... than or equal to ...</td>
</tr>
<tr>
<td>10-1</td>
<td>17</td>
<td>... than or equal to ...</td>
</tr>
<tr>
<td>10-1</td>
<td>20</td>
<td>... digits ...</td>
</tr>
<tr>
<td>10-5</td>
<td>4</td>
<td>... 0,4(2,7) ...</td>
</tr>
</tbody>
</table>
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
  11. Instructions (II), Mnemonics and Operands
  12. Establishing and Maintaining Addressability
  13. Constants, Storage Areas, and Literals

V. Basic Instructions
  14. General Register Shifting and Data Transmission
  15. Conditional Branching
  16. Fixed-Point Arithmetic Instructions
  17. Logical Operations and Instructions
<table>
<thead>
<tr>
<th>TABLES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Hexadecimal, Decimal, and Binary Digits</td>
<td>2-2</td>
</tr>
<tr>
<td>II. Multiples of Powers of 16</td>
<td>2-7</td>
</tr>
<tr>
<td>III. EBCDIC Character Representation</td>
<td>10-3</td>
</tr>
<tr>
<td>IV. RR Instructions</td>
<td>11-2</td>
</tr>
<tr>
<td>V. RX Instructions</td>
<td>11-8</td>
</tr>
<tr>
<td>VI. RS, SI Instructions</td>
<td>11-11</td>
</tr>
<tr>
<td>VII. SS Instructions</td>
<td>11-12</td>
</tr>
</tbody>
</table>
Glossary of Abbreviations

CC Condition Code (PSW34-35)
CPU Central Processing Unit
Fn Floating-Point Register n
IA Instruction Address (PSW40-63)
ILC Instruction Length Code (PSW32-33)
IR Instruction Register
LC Assembler Location Counter
MAR Memory Address Register
MDR Memory Data Register
PSW Program Status Word
Rn General-Purpose Register n
RR Register-to-Register
RS Register-to-Storage
RX Register-to-Indexed-Storage
SI Storage-Immediate
SS Storage-to-Storage
HW2 Halfword Operand 2
FW2 Fullword Operand 2
C1 Character (or Byte) Operand 1
11. INSTRUCTIONS (II), MNEMONICS AND OPERANDS

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

Mnemonics provide a short abbreviation for a descriptive word or phrase which designates the action of each operation code. They may range from something as simple as "A" meaning "Add", to "BXLE" meaning "Branch on Index Low or Equal". To simplify the presentation, we will discuss each class of instructions separately, and sometimes give examples of how they are written. A number of abbreviations such as r₁, s₂, I, etc. will be explained as we go along.

RR Instructions

Instructions of RR format are given in Table IV; several things should be noted about the instructions listed there. First, not all of the available digit combinations between 00₁₆ and 3F₁₆ (in the columns labeled "Opcode") are used as actual operation codes. Second, all of the instructions in the second column refer to the floating-point registers, the uses of which will be described in detail later. (The floating-point instructions operate on data in a format which is interpreted differently from the integer representations discussed in Section 6.) Third, two of the instructions (namely SSK and ISK) are not normally available to the programmer and their descriptions will therefore be deferred (they are called privileged operations).
<table>
<thead>
<tr>
<th>Opcode (hex)</th>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Opcode (hex)</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>SPM</td>
<td>Set Program Mask</td>
<td>20</td>
<td>LPDR</td>
<td>Load Positive</td>
</tr>
<tr>
<td>05</td>
<td>BALR</td>
<td>Branch and Link</td>
<td>21</td>
<td>LNDR</td>
<td>Load Negative</td>
</tr>
<tr>
<td>06</td>
<td>BCTR</td>
<td>Branch on Count</td>
<td>22</td>
<td>LTRD</td>
<td>Load and Test</td>
</tr>
<tr>
<td>07</td>
<td>BCR</td>
<td>Branch on Condition</td>
<td>23</td>
<td>LCTR</td>
<td>Load Complement</td>
</tr>
<tr>
<td>08</td>
<td>SSK</td>
<td>Set Storage Key</td>
<td>24</td>
<td>HDR</td>
<td>Halve</td>
</tr>
<tr>
<td>09</td>
<td>ISK</td>
<td>Insert Storage Key</td>
<td>28</td>
<td>LDR</td>
<td>Load</td>
</tr>
<tr>
<td>0A</td>
<td>SVC</td>
<td>Supervisor Call</td>
<td>29</td>
<td>CDR</td>
<td>Compare</td>
</tr>
<tr>
<td>10</td>
<td>LFR</td>
<td>Load Positive</td>
<td>2A</td>
<td>ADR</td>
<td>Add Normalized</td>
</tr>
<tr>
<td>11</td>
<td>LNR</td>
<td>Load Negative</td>
<td>2B</td>
<td>SDR</td>
<td>Subtract Normalized</td>
</tr>
<tr>
<td>12</td>
<td>LTR</td>
<td>Load and Test</td>
<td>2C</td>
<td>MDR</td>
<td>Multiply</td>
</tr>
<tr>
<td>13</td>
<td>LCR</td>
<td>Load Complement</td>
<td>2D</td>
<td>DDR</td>
<td>Divide</td>
</tr>
<tr>
<td>14</td>
<td>NR</td>
<td>Logical AND</td>
<td>2E</td>
<td>AWR</td>
<td>Add Unnormalized</td>
</tr>
<tr>
<td>15</td>
<td>CLR</td>
<td>Compare Logical</td>
<td>2F</td>
<td>SWR</td>
<td>Subtract Unnormalized</td>
</tr>
<tr>
<td>16</td>
<td>øR</td>
<td>Logical øR</td>
<td>30</td>
<td>LPER</td>
<td>Load Positive</td>
</tr>
<tr>
<td>17</td>
<td>XR</td>
<td>Exclusive øR</td>
<td>31</td>
<td>LNER</td>
<td>Load Negative</td>
</tr>
<tr>
<td>18</td>
<td>LR</td>
<td>Load</td>
<td>32</td>
<td>LTER</td>
<td>Load and Test</td>
</tr>
<tr>
<td>19</td>
<td>CR</td>
<td>Compare</td>
<td>33</td>
<td>LCER</td>
<td>Load Complement</td>
</tr>
<tr>
<td>1A</td>
<td>AR</td>
<td>Add</td>
<td>34</td>
<td>HER</td>
<td>Halve</td>
</tr>
<tr>
<td>1B</td>
<td>SR</td>
<td>Subtract</td>
<td>35</td>
<td>CER</td>
<td>Compare</td>
</tr>
<tr>
<td>1C</td>
<td>MR</td>
<td>Multiply</td>
<td>36</td>
<td>AER</td>
<td>Add Normalized</td>
</tr>
<tr>
<td>1D</td>
<td>DR</td>
<td>Divide</td>
<td>3B</td>
<td>SER</td>
<td>Subtract Normalized</td>
</tr>
<tr>
<td>1E</td>
<td>ALR</td>
<td>Add Logical</td>
<td>3C</td>
<td>MER</td>
<td>Multiply</td>
</tr>
<tr>
<td>1F</td>
<td>SLR</td>
<td>Subtract Logical</td>
<td>3D</td>
<td>DER</td>
<td>Divide</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3E</td>
<td>AUR</td>
<td>Add Unnormalized</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3F</td>
<td>SUR</td>
<td>Subtract Unnormalized</td>
</tr>
</tbody>
</table>

**TABLE IV.**

**RR Instructions**

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

\[ r_1, r_2 \]

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

To explain the meaning of the notation "\( r_1, r_2 \)", it is perhaps useful to refer to the example of a machine instruction statement in Fig. 9.5, in
which the operation and operand fields were "LR 7,3". (It was noted in the description of the figure that execution of this instruction would cause the contents of R7 to be replaced by the contents of R3.) In this case, "r1" is "7" and "r2" is "3". In fact, the quantities r1 and r2 must simply be absolute (i.e., non-relocatable) expressions of value less than 16; a more formal definition of the term "expression" will be given shortly.

Thus, we could just as well have written LR X'7',B'11' in this example. For RR instructions, the values of the expressions in the operand field are placed by the Assembler into two adjacent hexadecimal digits, called operand register specification digits, in the second byte of the instruction (which was labeled "Register Specification" in the first diagram of Fig. 4.2), as in the following figure.

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>r1</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 11.1 RR Instruction Showing Register Specification Digits

The subscripts on the quantities "r1" and "r2" are simply a way to distinguish which operand is being referred to; in general we will find that using the terms "first operand", "second operand", etc. in a consistent manner will help in remembering what actions are being performed by each instruction. We would therefore say for most of the RR instructions that the operand r1 specifies the register containing the "first operand". It will become apparent that the word "operand" is used here in two different senses: as part of the operand field entry of some instruction statement, an operand is an expression which will eventually be translated by the Assembler into some part of an instruction; we also call an operand one of the quantities in a register or in memory which at execution time participates in the given operation. The difference is not terribly important but can be confusing, and which is meant will normally be clear from context. Thus the operands (first meaning) in the operand field entry of the instruction LR 7,3 are 7 and 3, whereas at execution time the operands (second meaning) of the
LR instruction will be found in general registers 7 and 3. Using Table IV
to find that the operation code corresponding to the mnemonic LR is \( 18_{16} \),
the two-byte instruction which would be assembled from the statement as
given would be \( 1873 \) in hexadecimal.

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

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

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

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

Each of the latter three will be described later. An expression using a
symbol and a self-defining term is \( \text{GETC}^\text{NST}+X'4A' \) and an expression
using only self-defining terms is \( X'12'+C'.B'1010001'+7 \) which the
reader can verify to have the value \( 19_{16} \).

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

\[ \text{ABS}^{425} \quad \text{EQU} \quad 425 \]
serves to define an absolute symbol with value \( 425_{10} \). (This is the unusual case mentioned in Section 10 where the value of the symbol is not the value of the LC when the symbol was encountered.)

Parentheses in an expression may be used, as in ordinary mathematical use (and as in algebraic procedural languages such as FORTRAN, ALGOL, and PL/1) to indicate groupings. As one might suspect, an expression may not contain two operators in succession; a less familiar restriction is that an expression may not begin with an operator, so that \(-5 + \text{ABS}^{425}\) is invalid, whereas \(0 - 5 + \text{ABS}^{425}\) is correct. (The maximum number of terms allowed and the maximum level of nesting of parentheses in an expression both depend on the size and sophistication of the Assembler; we will simply mention an upper limit of 16 and 5 respectively, corresponding to the OS/360 Assembler.)

**Expressions**

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

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

2. Parenthesized subexpressions are evaluated first, and the resulting value used in computing the value of the rest of the expression. Thus in the expression \((X'100' + 2 * (\text{ABS}^{425} - 420)) + 1\) (where \(\text{ABS}^{425}\) is assumed to have been defined as above), the value of \((\text{ABS}^{425} - 420)\) would be evaluated first.

3. As is the case in procedural languages, multiplications and divisions are done before additions and subtractions. Thus the value of the expression just given would be evaluated as \((X'100' + (2 * (5))) + 1\) and not \(((X'100' + 2) * (5)) + 1\). Note that relocatable terms or subexpressions may not occur in multiply or divide operations.
4. Operations are performed in left-to-right order. Thus $5 \times 2 / 4$ means $(5 \times 2) / 4$, not $5 \times (2 / 4)$.

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

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

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

8. When the expression has been completely evaluated, it is truncated to the value contained in its rightmost $2^4$ bits, which is then considered (as was noted for self-defining terms) to have a positive value, even though the bits dropped off may have all been ones.

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

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

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

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

```
R7     EQU  7
R3     EQU  3
L$AD   LR   R7,R3
```
Note that there is a difference between (1) the notational convenience "R7" (meaning general register 7) introduced in Section 3, (2) the definition of an absolute symbol R7 to have the value 7, and (3) the use of the symbol as an operand in the operand field entry of a machine instruction where the use of register 7 is indicated. The above example is entirely equivalent to the two below.

\[
\begin{align*}
\text{Z\{RCH} & \text{ EQU 3} & \text{R7 EQU 3} \\
\text{ZILCH} & \text{ EQU 7} & \text{R3 EQU 7} \\
\text{L\{OAD LR ZILCH,Z\{RCH} & \text{L\{OAD LR R3,R7}
\end{align*}
\]

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

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

**RX Instructions**

RX instructions are given in Table V. As was the case in Table IV, not all of the available digit combinations are used as actual operation codes; and all of the instructions in the right-hand column again refer to operations on the floating-point registers and will be discussed later. None of the RX instructions is privileged, and the format of the operand field entry is the same for each. It should be kept in mind that RX instructions always refer to memory in some way. Referring to Fig. 11.2, we see that four quantities are to be specified -- the operand register specification digit r1, the index register specification digit x2, the base register specification digit b2, and the displacement d2. (We are again entering on a fairly technical discussion, the details of which need not be assimilated at this point, since many later examples will be given in illustration of the various possibilities.)
Table V.
RX Instructions

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

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

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

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

\[ d_2(x_2, b_2) \]

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

```
\[ \begin{array}{ccc}
43 & 0A & 7468 \\
43 & 00 & 7468 \\
43 & 07 & 0468 \\
\end{array} \]
```

\[ \text{IC} \quad 0,X'468'(10,7) \]

\[ \text{IC} \quad 0,1128(0,7) \]

\[ \text{IC} \quad 0,1128(7,0) \]

Figure 11.3 RX Instruction with Explicit Operands

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

As was mentioned in the discussion of addressing in Section 5, the use of the index register specification digit \(x_2\) when the base register specification digit \(b_2\) was intended can lead to programs which function more slowly, though correctly. By specifying only the base digit when no indexing is intended, the program is both more efficient and more easily understood -- the second of the above examples, where we could have written \(1128(,7)\) also, is therefore preferable to the third.
The utility of the Assembler becomes more apparent when we consider all the forms in which the second operand of an RX instruction may be written; these are given in Fig. 11.4 below.

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

Figure 11.4 Address Specification in RX-Type Instructions

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

In the two cases where an implied address is desired, the quantity $s_2$ may be either an absolute or a relocatable expression of value less than $2^{24}$. This means that we may write instructions such as $L\ 0,\ ANSWER$ and leave it to the Assembler to compute the proper base and displacement; how this is done will be discussed in the next section. For the moment suppose that the Assembler has sufficient information so that the instruction $IC\ 0,\ BYTE$ is translated into $143'001714681$ as in Fig. 11.3. Then if the index register to be used is R10, the instruction $IC\ 0,\ BYTE(10)$ would be translated into $143'0A714681$.

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

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

\[
\begin{align*}
L & \ 0,N \quad \text{LOAD FROM \( N \) INTO \( RO \)} \\
A & \ 0,\phi \text{NE} \quad \text{ADD INTEGER CONSTANT 1} \\
ST & \ 0,N \quad \text{STORE RESULT BACK AT \( N \)}
\end{align*}
\]

where it is assumed that an addressable fullword area named \( \phi \text{NE} \) which contains the integer constant +1 has been defined in the program. We will see later that there are several ways to define such constants.

**RS and SI Instructions**

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

<table>
<thead>
<tr>
<th>Opcode (hex)</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>SSM</td>
<td>Set System Mask</td>
</tr>
<tr>
<td>82</td>
<td>LPSW</td>
<td>Load PSW</td>
</tr>
<tr>
<td>83</td>
<td>Diagnose</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>WRD</td>
<td>Write Direct</td>
</tr>
<tr>
<td>85</td>
<td>RDD</td>
<td>Read Direct</td>
</tr>
<tr>
<td>86</td>
<td>BXH</td>
<td>Branch on Index High</td>
</tr>
<tr>
<td>87</td>
<td>BXLE</td>
<td>Branch on Index Low</td>
</tr>
<tr>
<td>88</td>
<td>SRL</td>
<td>Shift Right SL</td>
</tr>
<tr>
<td>89</td>
<td>SLL</td>
<td>Shift Left SL</td>
</tr>
<tr>
<td>8A</td>
<td>SRA</td>
<td>Shift Right S</td>
</tr>
<tr>
<td>8B</td>
<td>SLA</td>
<td>Shift Left S</td>
</tr>
<tr>
<td>8C</td>
<td>SRDL</td>
<td>Shift Right DL</td>
</tr>
<tr>
<td>8D</td>
<td>SLDL</td>
<td>Shift Left DL</td>
</tr>
<tr>
<td>8E</td>
<td>SRDA</td>
<td>Shift Right D</td>
</tr>
<tr>
<td>8F</td>
<td>SLDA</td>
<td>Shift Left D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode (hex)</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>STIM</td>
<td>Store Multiple</td>
</tr>
<tr>
<td>91</td>
<td>TM</td>
<td>Test Under Mask</td>
</tr>
<tr>
<td>92</td>
<td>MVI</td>
<td>Move</td>
</tr>
<tr>
<td>93</td>
<td>TS</td>
<td>Test and Set</td>
</tr>
<tr>
<td>94</td>
<td>NI</td>
<td>Logical AND</td>
</tr>
<tr>
<td>95</td>
<td>CLI</td>
<td>Compare Logical</td>
</tr>
<tr>
<td>96</td>
<td>( \phi )I</td>
<td>Logical ( \phi )R</td>
</tr>
<tr>
<td>97</td>
<td>XI</td>
<td>Exclusive ( \phi )R</td>
</tr>
<tr>
<td>98</td>
<td>IM</td>
<td>Load Multiple</td>
</tr>
</tbody>
</table>

**TABLE VI.**

**RS and SI Instructions**

(For Shift Instructions, \( S = \) Single, \( L = \) Logical, \( D = \) Double)

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

SS Instructions

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

\begin{table}[h]
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Opcode (hex) & Mnemonic & Instruction & Opcode (hex) & Mnemonic & Instruction \\
\hline
\texttt{D1} & \texttt{MVN} & Move Numeric & \texttt{F1} & \texttt{MV\phi} & Move with Offset \\
\texttt{D2} & \texttt{MVC} & Move & \texttt{F2} & \texttt{PACK} & Pack \\
\texttt{D3} & \texttt{MVZ} & Move Zone & \texttt{F3} & \texttt{UNPK} & Unpack \\
\texttt{D4} & \texttt{NC} & Logical AND & & & \\
\texttt{D5} & \texttt{CLC} & Compare Logical & \texttt{F3} & \texttt{ZAP} & Zero and Add \\
\texttt{D6} & \texttt{C} & Logical \& & \texttt{F9} & \texttt{CP} & Compare \\
\texttt{D7} & \texttt{XC} & Exclusive \& & \texttt{FA} & \texttt{AP} & Add \\
\texttt{DC} & \texttt{TR} & Translate & \texttt{FB} & \texttt{SP} & Subtract \\
\texttt{DD} & \texttt{TRT} & Translate and Test & \texttt{FC} & \texttt{MP} & Multiply \\
\texttt{DE} & \texttt{ED} & Edit & \texttt{FD} & \texttt{DP} & Divide \\
\texttt{DF} & \texttt{EIMK} & Edit and Mark & & & \\
\hline
\end{tabular}
\end{table}

TABLE VII.

SS Instructions

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

To conclude this short presentation of the instruction repertoire of System/360, a summary is given in the figure below of some of the overall characteristics of the instructions as they depend on the first four bits of the operation code. As was illustrated in Section 4, the first two bits

11-12
determine the type and length of the instruction. The second pair of bits determines (depending on the instruction type) the operand length or the general functions performed by the instructions.

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

Figure 11.5 General Instruction Classification

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

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

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

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

<table>
<thead>
<tr>
<th>Location</th>
<th>Name</th>
<th>Operation</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>BALR</td>
<td></td>
<td>6,0</td>
</tr>
<tr>
<td>5002</td>
<td>BEGIN</td>
<td>L</td>
<td>2,N</td>
</tr>
<tr>
<td>5006</td>
<td>A</td>
<td>ADD</td>
<td>2,$NE</td>
</tr>
<tr>
<td>500A</td>
<td>ST</td>
<td>STORE</td>
<td>2,N</td>
</tr>
<tr>
<td>5024</td>
<td>N</td>
<td>DC</td>
<td>F'8'</td>
</tr>
<tr>
<td>5028</td>
<td>$NE</td>
<td>DC</td>
<td>F'1'</td>
</tr>
</tbody>
</table>

--- twenty-two additional bytes of instructions, data, etc. ---

Figure 12.1 A Simple Program Segment
Some explanation of the items in the example may be helpful. The instructions 
L, A, and ST respectively (1) put the contents of a fullword from memory 
into a general register (i.e., Load the register), (2) Add the contents of 
a fullword area in memory to the contents of a register, and (3) replace 
the contents of a fullword area in memory with the contents of a general 
register (i.e., Store the register). The DC statements, which are treated 
in the next section, are meant simply to provide two fullword areas of 
memory with names "N" and "ONE" which contain the fullword integer values 
desired; we have arbitrarily set the contents of the fullword at N to the 
integer 8 even though in an actual program any value might be possible. 
All of these instructions will be covered in detail later.

When the program has begun and after the BALR has been executed, R6 
will contain(xx005002)\textsubscript{16}, where xx stands for two hex digits whose values 
are of no concern at the moment. To determine the proper displacement for 
the L instruction at 5002\textsubscript{16}, we can use the known contents of R6 (since 
the xx digits are ignored in address computations) to compute a displacement 
of 5024\textsubscript{16} - 5002\textsubscript{16} = 022\textsubscript{16}; then the assembled machine instruction (using 
the operation code 58 for the mnemonic L) should be 28206022. Then 
when the instruction is executed, the computation of the effective address 
yields 022 + 005002 = 005024, which is what we want. If we continued in 
this fashion for the rest of the instructions, we would find that the 
following "assembled" quantities in the indicated locations would give the 
desired results.

<table>
<thead>
<tr>
<th>Location</th>
<th>Assembled Contents</th>
<th>Original Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>0560</td>
<td>BALR 6,0</td>
</tr>
<tr>
<td>5002</td>
<td>58206022</td>
<td>BEGIN L 2,N</td>
</tr>
<tr>
<td>5006</td>
<td>5A206026</td>
<td>A 2,ONE</td>
</tr>
<tr>
<td>500A</td>
<td>50206022</td>
<td>ST 2,N</td>
</tr>
<tr>
<td>5024</td>
<td>00000003</td>
<td>N DC F'8'</td>
</tr>
<tr>
<td>5028</td>
<td>00000001</td>
<td>ONE DC F'1'</td>
</tr>
</tbody>
</table>

Figure 12.2 Simple Program Segment with Assembled Contents
So far, so good: we have constructed a sequence of statements which will give a desired result if it is placed in memory at the right place. It is natural to ask at this point what would happen if the program had been put elsewhere by the Supervisor. So, assume that the same program segment begins at $84E8_{16}$, as in the figure below.

<table>
<thead>
<tr>
<th>Location</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$84E8$</td>
<td>BALR 6,0</td>
</tr>
<tr>
<td>$84EA$</td>
<td>BEGIN L 2,N</td>
</tr>
<tr>
<td>$84EE$</td>
<td>A 2,ϕNE</td>
</tr>
<tr>
<td>$84F2$</td>
<td>ST 2,N</td>
</tr>
</tbody>
</table>

--- the same 22 bytes of odds and ends ---

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

Figure 12.3 Same Program Segment, Different Memory Location

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

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

Figure 12.4 Same Program Segment with Assembled Contents

The identical assembled program would be used in each case to perform the desired calculation. It therefore appears that so long as the same fixed relationship is maintained between the various parts of the program segment (namely that there be 22 bytes between the ST instruction and the fullword named N, and that N and ϕNE name areas that fall on fullword boundaries, the segment could be placed anywhere in memory and still execute correctly.
This is because the displacements of the three RX-type instructions were calculated on the assumption that at the time the program is executed there would be an address in R6 (namely the address of the L instruction named BEGIN) which could be used for a base address. Indeed, we could have assumed that the program began at memory location zero (even though an actual program would not be placed there) because the contents of R6 after the BALR would then be xx000002 and the displacements would be calculated exactly as before. In the first example, the actual origin of the program segment was 5000₁₆; we could by chance have assigned that value as a relative origin in the program and had the values of the Assembler's Location Counter correspond identically to the actual locations later assigned by the Supervisor to each instruction. In that case, we would need to inform the Assembler that the quantity to be used as a base is 5002₁₆, and that it would be found in R6 at execution time. Similarly, in the second example, the relative origin would be 84E8₁₆, and the contents of R6 that the Assembler should assume in order to calculate the correct displacements would be 84EA₁₆. If the value of the actual origin is assigned to the relative origin by the programmer, and if the Assembler knows that the contents of R6 at execution time will also be the value of the symbol BEGIN, then the correct displacements will be found. However, in each of the above examples, the computation of the displacements actually depended not on a knowledge of the actual locations of the instructions at execution time, but only on their locations relative to one another and on the value assumed to be available for addressing purposes. Thus, the technique used is to assign a relative origin for the program, and then to give some value relative to that relative origin which may be used for computing displacements; although this seems complicated, we will find it quite simple in practice.

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

```
USING   s,r1
```

where s is a relocatable or absolute expression (usually just a symbol will be used) whose magnitude is less than 2²⁴, and r₁ is an absolute
expression of value less than 16 which specifies the register to be used as a base. (As usual, there is more to using USING than has been stated here, but we will use this simplified explanation for the time being.) Thus, the statement USING BEGIN, 6 would inform the Assembler that register 6 may be assumed (for purposes of computing displacements) to be a base register which will contain the value of the symbol BEGIN. We could rewrite the sample program segment to include the USING statement as in the figure below.

```
BALR 6,0
USING BEGIN, 6
BEGIN L 2,N
A 2,ONE
ST 2,N

N DC F'8'
ONE DC F'1'
```

Figure 12.5 Program Segment with USING Instruction

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

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

\[
\text{BEGIN } L \ 6,N \ \text{LOAD CONTENTS OF N INTO R2}
\]

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

<table>
<thead>
<tr>
<th>Location</th>
<th>Assembled Contents</th>
<th>Statement</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0560</td>
<td>BALR 6,0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>USING BEGIN,6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>58606022</td>
<td>BEGIN L 6,N</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5A206026</td>
<td>A 2,NE</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>50206022</td>
<td>ST 2,N</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>00000008</td>
<td>N DC F'8'</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>00000001</td>
<td>NE DC F'1'</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12.6 Sample Program Segment with Erroneous Statement

It is apparent that this program will assemble correctly, as did the one in Figure 12.5, since all quantities are properly specified. However, at execution time, things go rapidly awry. Suppose again that the actual location assigned by the Supervisor to the BALR is 5000_16, so that when the L instruction is executed, R6 contains xx005002_16. Now, the L instruction transmits a fullword from the memory location at the effective address given by the second operand into the register specified by the first operand, which in this case is R6. When the effective address of N is being calculated, R6 will contain the correct base address; but when the execution of the L instruction is complete, the contents of R6 will have become 00000008_16, and not xx005002. When the next instruction is executed, the effective address calculated is 26_16 + 8_16 = 00002E_16 and not 5028_16, which is where the desired operand is to be found. In this case, the generated effective address is not divisible by 4, so that it refers to the incorrect byte of the required fullword operand; hence a specification exception occurs, and remedial action can be initiated immediately. This does not by any means imply that at any time we have the misfortune to destroy the contents of a
base register that the CPU will be able to detect the error. Indeed, if the contents of the fullword at \( N \) had been the integer 2 instead of 8, then the effective address would have been computed to be \( 2 + 2^6 = 2^8_{16} \), which is a perfectly acceptable address for a fullword. The subsequent instructions would thus have gone their way, adding the contents of the fullword at memory location \( 2^8_{16} \) to \( R2 \), and storing the result at location \( 2^4_{16} \), which is obviously not what is intended. It is partly a matter of chance as to how much further damage such a program error can cause when the program is executed; indeed, when the CPU finally (if it ever) detects an error, all evidence pointing to the offending instruction may have been lost (\( R6 \) may have been changed several times!), making error tracing difficult. Thus the programmer must take care to insure the integrity of the contents of registers being used for base registers, since the Assembler makes no checks for instructions performing operations on registers designated in \texttt{USING} instructions as base registers. This warning should not be taken lightly; the errors caused by mishandling base registers are among the most destructive of program continuity and the most difficult to find.

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

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

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

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

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

1. the value of the constant (in this case $8_{10}$);
2. the type of internal representation to be used for the given value (in this case two's complement integer);
3. the length of the constant (in this case four bytes); and
4. the alignment in memory of the constant (in this case on a fullword boundary).

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

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

When the relative origin is specified by the programmer at the start of his program, the Assembler checks whether the value given is exactly divisible by eight; if not, it is "rounded up" to the next larger multiple of eight, which is then used as the relative origin of his program. Thus the Assembler insures that the program begins with the most restrictive possible boundary alignment. Then if a constant is defined which must fall on some particular kind of word boundary, the Assembler need insure only that its Location Counter be divisible by the proper power of two (that is, by 2, 4, or 8) at the location of the leftmost byte of the constant. The Linkage Editor and Resident Supervisor must then respect this assumed alignment for the beginning of the program; this ensures that data and instructions will fall on the proper boundaries when the program is finally loaded into memory for execution. We will of course assume that this is exactly what happens in the rest of our discussion; some of the implications of this method of handling programs will be treated in later discussions which give more details of the processes of linkage editing and loading.
We must now investigate what it is that the Assembler actually does to ensure that its Location Counter is indeed divisible by the desired quantity. Suppose in some program that after a sequence of instructions has been processed the value of the LC is $12E_{16}$, so that if another machine instruction were assembled at this point it would begin on a halfword boundary between two fullword boundaries (recall that instruction addresses need only be divisible by 2). Suppose also that the next statement is not a machine instruction statement but is `DC F'8'` instead. To assemble the four bytes representing the constant (namely $00000008_{16}$) beginning at $12E_{16}$ would be incorrect, since an instruction which referred to the constant might require that its memory address be on a fullword boundary. To avoid such an erroneous situation, the Assembler will automatically skip enough bytes to obtain the desired boundary alignment. Thus in this simple example the LC would be increased to $130_{16}$ before the fullword constant is assembled into the program, and the LC would have a value of $134_{16}$ after the constant is processed rather than the value of $132_{16}$ which would be the case if no automatic alignment had been performed. An automatic alignment is not performed in the following circumstances:

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

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

```
Ln
```

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

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

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

\[
\text{DC F}'8' \quad \text{and} \quad \text{DC FL}'8'\]

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

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

\[
\begin{align*}
\text{IMPLIED DC F}'8' \\
\text{EXPLICIT DC FL}'8' \\
\end{align*}
\]

Figure 13.1 Implied and Explicit Length Specifications

Because no boundary alignment is performed in the latter case it is clear that the value of the symbol EXPLICIT will be \(12E_{16}\). In the former case, however, two bytes must be skipped by the Assembler to achieve the required boundary alignment implied by type F. Since we will want to be able to refer to the constant by using the symbol IMPLIED, it is also clear that it should have the value given to the location of the leftmost byte of the constant, namely \(130_{16}\). Thus if a symbol is to be defined, it is given its value after bytes are skipped to achieve boundary alignment. In fact, a general rule may be stated: the Assembler will never automatically assign the value of a symbol to the location of skipped bytes. (The programmer can find ways to do so if he is so inclined.) This includes the case where a byte must be skipped to ensure that an instruction begins on a halfword boundary. When bytes are skipped to achieve alignment of a following constant or instruction, the Assembler will insert zeros into the bytes skipped.
The three useful constant types C, X, and B differ from F and H in that no default values are assumed for either length or alignment. For example, the five bytes required to store the constant generated by the statement

DC C'12345' will be placed by the Assembler at the next available address given by the current value of the LC. If a particular boundary alignment is desired, extra steps must be taken which will be described later in this section. The method of writing such constants is, as might be guessed, the same as for writing character, hexadecimal, and binary self-defining terms, except that the limitations on length and value are different. In the case of self-defining terms, the value of the term was restricted to being less than $2^{24}$, whereas much longer constants can be defined with the DC instruction. Thus one can define constants in statements such as in Figure 13.2 below.

```
TITLE    DC C'THIS IS A LONG CHARACTER CONSTANT'
DIGITS   DC X'8462AFCB975310'
```

Figure 13.2 Examples of Character and Hexadecimal Constants

In the discussion of data converted according to types F and H it was reasonable that the resulting binary numbers should be placed with the least significant digit at the right-hand end of the desired storage area, and that the sign bit should be extended to the left. In all the examples given, the constants were small enough to fit safely in the allotted space. The problem may arise as to what should be done if (1) the constant is too small to occupy fully the number of bits allocated for it by the length specification (whether an explicit length modifier or the default length is used), or if (2) the constant is too large to fit in the allotted space. Some examples of such cases are given in Figure 13.3, along with the constants actually stored by the Assembler. The rules used to determine the final values of the constants are given below.
For all of the constants on the left, some part of the true value must be truncated to make it fit into the allotted space, since a length is specified in each case. For all the constant types we are discussing except C, excess information is dropped at the left end of the constant, and the rightmost portion is what is eventually assembled; for character constants the excess is trimmed off the right end, as may be verified in the example above. Note that the special rules concerning the apostrophe and ampersand in character self-defining terms also apply to character constants.

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

<table>
<thead>
<tr>
<th>Constant too Large</th>
<th>Assembled Value</th>
<th>Constant too Small</th>
<th>Assembled Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>H'65537'</td>
<td>0001_{16}</td>
<td>H'2'</td>
<td>0002_{16}</td>
</tr>
<tr>
<td>FL1'-300'</td>
<td>D4_{16}</td>
<td>FL1'-6'</td>
<td>FA_{16}</td>
</tr>
<tr>
<td>CL3'SMITH'</td>
<td>E2D4C9_{16}</td>
<td>CL3'S'</td>
<td>E24040_{16}</td>
</tr>
<tr>
<td>XL2'56789'</td>
<td>6789_{16}</td>
<td>X'56789'</td>
<td>056789_{16}</td>
</tr>
<tr>
<td>BL1'100100100'</td>
<td>00100100_{2}</td>
<td>B'101'</td>
<td>000001001_{2}</td>
</tr>
</tbody>
</table>

Figure 13.3 Examples of Truncated and Padded Constants
In the last two examples in Figure 13.3 some padding with zeros was required at the left end of the constants to fill out the partially-specified byte.

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

The A-type constant is similar to F-type constants in that a length of four bytes and a fullword boundary alignment are implied; thus A(10) and F'10' are equivalent operands, as are AL4(10) and FL4'10' . A major difference lies in the ability to specify constants such as A(X'12E') and A(C'1') (which are the same as F'302' and F'64' respectively), in which the use of such expressions may greatly simplify the programming task. In particular one may define constants using operands such as A(ABS425) where the symbol ABS425 may have been defined in an EQU statement (as in Section 11) to have some particular value. Though the utility of such constructs is not apparent now, we will see through later examples that clarity and simplicity can be gained through their use.

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

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

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

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

Figure 13.4 Summary of Rules for Certain DC Operands

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

STORAGE DS 100F

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

\[ \text{AREA1 DS 80C} \quad \text{and} \quad \text{AREA2 DS C180} \]

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

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

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

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

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

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

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

\[
\begin{align*}
=\text{F}'1' & \quad =\text{C}'\text{LIT}\text{ER}\text{AL}' & \quad =\text{BL2}'111101' \\
=\text{H}'1' & \quad =\text{CL7}'\text{BLANK}' & \quad =\text{X}'7654321' \\
=\text{A}(1) & \quad =\text{F}'1,2,3,4' & \quad =\text{AL3}(5,\text{X}'D7'/'\text{C}'')
\end{align*}
\]

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

1. a literal is a term which may not be combined with other terms
   (thus IC 0,=F'1'+3 is illegal);
2. an instruction may not store or modify a literal (thus ST 7,=F'1'
   is illegal);
3. a literal may not be specified in an address constant (about which
   more later) (so that A(=F'1') is illegal);
4. multiple operands may not be specified, but multiple values may;
5. the duplication factor may not be zero;
6. the alignment of the data described in the literal is that implied
   by the constant type (so that L 2,=X'2B' will probably cause a
   specification exception).
To illustrate the use of a literal in a program segment, we could rewrite the example in Figure 12.1 in the form given in Figure 13.5 below.

```
BALR 6,0
USING BEIN,6
BEIN L 2,N
A 2,=F'1'
ST 2,N

N DC F'8'
```

Figure 13.5 Sample Program Using a Literal

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

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

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

A notational convenience will be introduced here: because we will often have need to use the phrase "general purpose register r1" where r1 indicates the value supplied for an operand in the operand field entry of a machine instruction statement, we will use the abbreviation "Rr1" instead. Thus if r1 has the value 5, the register being referred to is R5.

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

\[ L \quad r1,\text{d}(x2,b2) \]

causes the fullword second operand to replace the contents of Rr1. The original contents of Rr1 are lost, and the contents of the fullword area in memory remain unchanged. As a reminder, the term "operand" was used here to mean the data referred to at execution time by the effective address, which was computed from components of the instruction determined during assembly from the second operand in the operand field entry of the instruction statement. As mentioned before, which meaning of the word "operand" is intended will usually be clear from context.
For example, to set the contents of R9 to zero we could write

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

and to set it to the maximum negative number,

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

would suffice.

The inverse operation ST is written explicitly as

\[ ST \ r_1,d_2(x_2,b_2) \]

and causes the contents of Rr1 to replace the contents of the fullword area of memory at the effective address of the second operand. The contents of the register are unchanged, and the original contents of the fullword area of memory are lost. For example, to duplicate at B the contents of the fullword at A, we could write

\[ L \ 0,A \ ST \ 0,B \]

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

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

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

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

\[
\begin{align*}
&L \ 1,A \quad ST \ 1,B \\
&L \ 2,A+4 \quad ST \ 2,B+4 \\
&L \ 3,A+8 \quad ST \ 3,B+8
\end{align*}
\]
If the number of registers is large, however, this can be cumbersome and slow, and it is more convenient in many cases to use the LM (Load Multiple) and STM (Store Multiple) instructions. Each of these is an RS-type instruction for which three operands must be specified in the operand field entry, as follows:

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

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

As usual, \(r_1\) and \(r_3\) must be positive absolute expressions of value 15 or less, and the base and displacement may be given explicitly or left for the Assembler to compute from the value of a symbol or other relocatable expression. The meanings of the register specification digits in the STM instruction are as follows: beginning with \(R_{r_1}\), transmit the registers in order of increasing number to the successive fullwords in memory which start at the effective address of the second operand, until \(R_{r_3}\) has been transmitted. If \(r_3\) is equal to \(r_1\), only one register is transmitted. If \(r_3\) is less than \(r_1\) then \(R_{r_1}\) through \(R_{15}\) will be transmitted, followed by \(R_0\) through \(R_{r_3}\); thus \(R_0\) may be considered to follow after \(R_{15}\), so that the general registers "wrap around" from the highest to lowest numbered. The LM instruction follows the same rules except that the registers are loaded in sequence from successive fullwords in memory.

For example, \(\text{LM 2,6,=5F'0'}\) would cause the contents of \(R_2, R_3, R_4, R_5,\) and \(R_6\) to be set to zero. Similarly, \(\text{STM 0,15,SAVE}\) would cause the contents of all sixteen registers to be stored beginning at SAVE, which could be defined in a statement such as \(\text{SAVE DS 16F}\) which ensures that the proper boundary alignment will be specified for the second operand address. If we assume that \(R_1\) contains the address of a list of
four fullword constants, we could load them into R7 through R10 by executing the statement \( \text{LM} \ 7,10,0(1) \) and if we assume that R13 contains the address of a register save area, then \( \text{STM} \ 14,12,12(13) \) would store \( R_{14}, R_{15}, R_{0}, \ldots R_{12} \) in successive fullwords, beginning with the fourth fullword of the area. These last two examples illustrate certain conventions commonly used in communicating with subroutines, which will be treated in detail later. As a final example, suppose we wish to exchange the contents of \( R_{0} \) through \( R_{7} \), as a block, with the contents of \( R_{8} \) through \( R_{15} \). We could then write

\[
\begin{align*}
\text{STM} & \ 0,15,\text{SAVE} \\
\text{LM} & \ 8,7,\text{SAVE} & \text{or} & \text{STM} & \ 8,7,\text{SAVE} \\
\text{---} & \text{ or } \text{---} & \text{---} & \text{---} \\
\text{SAVE} & \ DS \ 16F & \text{SAVE} & \ DS \ 16F
\end{align*}
\]

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

The transmission of halfword data between memory and registers is somewhat more complicated, because a halfword requires only half of a general register. The relevant instructions, \text{LH} (Load Halfword) and \text{STH} (Store Halfword) are similar to \text{L} and \text{ST}; both are RX instructions, and the operand field entry is written the same way. \text{STH} is the simpler of the two: the rightmost 16 bits (the right half) of \( R_{r1} \) replaces the halfword at the effective address of the second operand, and \( R_{r1} \) remains unchanged. If the contents of the register represent an integer too large to be correctly represented as a 16-bit two's complement integer, some significance is lost; no indication is made that the halfword in memory may not have the desired value. (An example illustrating this will be given shortly.) Conversely, when data is being transmitted from memory to a register by the \text{LH} instruction, it is reasonable to assume that the programmer wants to perform some arithmetic operations on the value transmitted, so that the data should occupy the entire
register with the least significant bit at the right-hand end. To give a correct representation in the 32-bit register, the sign bit of the 16-bit halfword operand must therefore be extended to the left to occupy the left half of the general register. One may visualize this process as taking place in two steps. The halfword operand is brought from memory and placed in the Memory Data Register (MDR), which is an internal register used for communicating between the CPU and memory. The leftmost bit of the halfword is duplicated to the left by 16 positions, providing a 32-bit representation of the original 16-bit two's complement operand. The resulting 32 bits are then transmitted to the designated general register. Though none of the models of System/360 use the MDR in precisely this fashion, we will find that the descriptions of many instructions can be simplified considerably by supposing it to take an active part in the handling of data passing between memory and the CPU. Note that there is also an instruction with mnemonic MDR; we will indicate which is meant if there is a possibility of confusing the two. Thus the statements \( \text{LH } 0,=H'1' \) and \( \text{LH } 0,=H'-1' \) would cause the contents of RO to be set to \( 00000000_{16} \) and \( FFFFFFFF_{16} \) respectively. As long as the value of the halfword operand \( X \) involved satisfies \(-2^{15} \leq X < 2^{15}\) it can be correctly represented in 16 bits and will therefore be correctly transmitted by LH and STH instructions. If this is not the case, situations such as those illustrated in the next two examples can arise.

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

\[
\begin{align*}
\text{L} & \quad 0,B \quad \text{C(RO)}=X'00010001' \\
\text{STH} & \quad 0,A \quad \text{C(A)}=X'0001' \\
\text{LH} & \quad 1,A \quad \text{C(R1)}=X'00000001' \\
\text{---} & \\
\text{A} & \quad \text{DS} \quad \text{H} \\
\text{B} & \quad \text{DC} \quad \text{F'65537'}
\end{align*}
\]

Figure 14.2 Loss of Significant Digits when Using STH

14-5
The contents of RO and Rl are different because the quantity in RO being stored by the second instruction is too large. A more awkward result is illustrated in Figure 14.3.

\[
\begin{array}{ll}
\text{L} & 0,=F'65535' \\
\text{STH} & 0,A \\
\text{LH} & 1,A \\
\hline
\text{A} & \text{DS} & H
\end{array}
\]

C(RO)=X'0000FFFF'
C(RO)=X'FFFF'
C(RL)=X'FFFFFFFF'

Figure 14.3 Loss of Significant Digits when Using STH

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

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

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

\[
\begin{array}{ll}
\text{IC} & 0,X \\
\text{STC} & 0,Y+1 \\
\text{IC} & 0,X+1 \\
\text{STC} & 0,Y \\
\hline
\text{X} & \text{DC} & \text{C'AB'} \\
\text{Y} & \text{DS} & \text{CL2 BECOMES C'BA'}
\end{array}
\]
Occasionally when memory space is at a premium it is convenient to use a single byte to contain a small integer constant; its value may be placed in a register using the following instruction sequence.

\[
\begin{align*}
\text{L} & 1,=F'0' \\
\text{IC} & 1,\text{LITLC\&N} \\
\text{LITLC\&N} & \text{CLEAR REGISTER} \\
\text{DC} & \text{FL1'53'}
\end{align*}
\]

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

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

Figure 14.4 Action of Certain General Register Instructions

The meanings of the CC settings are given below.

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

Figure 14.5 Condition Code Settings
As can be seen from Figure 14.4, the actions of LR and LTR are identical except that LTR also sets the CC. It is not uncommon to test the contents of a register by writing an instruction such as LTR 4,4 which has no effect other than to set the CC, which may then be tested by a BC or BCR instruction, which will be discussed later. For the other three instructions, the arithmetic operations are those implied by a 32-bit two's complement representation; thus overflow can occur during execution of LCR or LPR only if C(Rr2) is the maximum negative number, \(-2^{31}\), and no overflow can occur during execution of LNR because all representable positive values have a corresponding two's complement representation of their negative values.

The following short instruction sequence illustrates possible uses of the instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Rr1</th>
<th>Rr2</th>
<th>Rr3</th>
<th>Immediate</th>
<th>CC</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM</td>
<td>2,3</td>
<td>F'1,0'</td>
<td></td>
<td>C(R2)=1, C(R3)=0, CC NOT SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LR</td>
<td>7,3</td>
<td></td>
<td></td>
<td>C(R7)=0, CC NOT SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTR</td>
<td>2,2</td>
<td></td>
<td></td>
<td>C(R2)=1, CC=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LNR</td>
<td>1,7</td>
<td></td>
<td></td>
<td>C(R1)=0, CC=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCR</td>
<td>4,2</td>
<td></td>
<td></td>
<td>C(R4)=-1, CC=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LPR</td>
<td>0,4</td>
<td></td>
<td></td>
<td>C(R0)=+1, CC=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LNR</td>
<td>5,2</td>
<td></td>
<td></td>
<td>C(R5)=-1, CC=1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14.6 Example of Use of Certain RR Instructions

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

The shifting instructions to be described next are more interesting, since they allow the programmer to manipulate data in more varied ways than the instructions described up to now. All of the eight shift instructions are RS-type; they differ from LM and STM in the important respect that the R3 register specification digit (see Figure 14.1) is ignored when the
instructions are executed, and thus the operand field entry for shift instructions is written

\[ r_1, d_2(b_2) \]

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

The basic operation in shifting is the unit shift, in which each bit moves to the right or left by one binary digit position; the vacated bit position on the left or right end is handled differently for logical and arithmetic shift instructions. For the logical shifts, the vacated bit position is always set to zero, and any bits shifted off the opposite end are lost and ignored; for arithmetic shifts this is true only at the right end. Thus, if the contents of R8 are \( 8765_{16} \) and the instruction SLL 8,1 is executed, the result in R8 will be \( 0ECA8642_{16} \). Note that we could have written SLL 8,1(0) also, because the explicit use of 0 as a base register specification digit causes no base register to be used in the calculation of an effective address. Again supposing R8 to contain \( 8765_{16} \) and R3 to contain \( 82F3A2B5_{16} \), execution of the instruction SRL 8,16(3) would cause the contents of R8 to be shifted right \( \ldots B5_{16} + 10_{16} = 05_{16} \) (modulo 40_{16}) bit positions, leaving \( 043B2A19_{16} \) as the result.

For a simple example of the use of the single-register logical shift instructions, suppose we have a large table of data, where each entry is six bytes long and is aligned on a halfword boundary. Suppose also that the first three bytes contain character information of some sort, and the remaining three bytes are to contain a 24-bit two's complement integer value associated with the characters. We want to load and store the integer value into and from R5, where it will be used for some purpose in the program. Now it is clear that L and ST cannot be used, since it is not possible to obtain the proper alignment of the operand in memory; similarly, LH and STH handle only two of the three bytes. A simple solution is to pack the integer value so that its rightmost eight bits occupy the first byte, and the
leftmost 16 bits occupy the second and third bytes. Suppose \(R_5\) contains \(FFFFA620B_{16}\), and \(R_{12}\) contains the address of the first byte of the particular 6-byte data entry under consideration. Then the sequence of instructions below can be used to pack the number into memory. (The letters \(XXYYZZ\) are meant to represent the hex digits of the three characters in the data entry.)

\[
\begin{align*}
\text{STC} & \quad 5,3(0,12) \quad \text{C(\text{DATA ENTRY}) = XXYYZZOB-----} \\
\text{SRL} & \quad 5,8 \quad \text{C(R5) = 00FFFFA62} \\
\text{STH} & \quad 5,4(0,12) \quad \text{C(\text{DATA ENTRY}) = XXYYZZOBFA62}
\end{align*}
\]

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

\[
\begin{align*}
\text{LH} & \quad 5,4(0,12) \quad \text{C(R5)=FFFFFA62} \\
\text{SLL} & \quad 5,8 \quad \text{C(R5)=FFFF6200} \\
\text{IC} & \quad 5,3(0,12) \quad \text{C(R5)=FFFF620B}
\end{align*}
\]

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

We could also have arranged the data so that the three-byte integer value occupied the first three bytes of the data entry, and the characters occupied the last three bytes. The integer value would then be stored in memory with its bits in the proper arithmetic sequence; the instructions needed to load the value into \(R_5\) would be as follows, assuming that the data entry contained \(FA620BXXYYZZ\).

\[
\begin{align*}
\text{LH} & \quad 5,0(0,12) \quad \text{C(R5)=FFFFFA62} \\
\text{SL} & \quad 5,8 \quad \text{C(R5)=FFFF6200} \\
\text{IC} & \quad 5,2(0,12) \quad \text{C(R5)=FFFF620B}
\end{align*}
\]

It is apparent that the particular arrangement of the data in memory may depend on the programmer's inclinations, as well as on considerations of ease of programming or speed of execution.
The double-length logical shift instructions SLDL (Shift Left Double Logical) and SRDL (Shift Right Double Logical) work in exactly the same way as SLL and SRL except that a pair of registers is shifted. The register specified by the first operand (Rr) must be an even-numbered register; otherwise a specification exception will occur. The next higher numbered register is the low-order half of the double-length register pair, with bits shifted out the right end of Rr entering the left end of Rr+1, and vice versa. (This is one of the reasons for showing the general registers in pairs in Figure 3.7.)

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

\[
\begin{align*}
\text{LH} & \ 2,A & \text{LH} & \ 2,A & \text{L} & \ 2,A & \text{LH} & \ 2,A \\
\text{SRDL} & \ 2,16 & \text{SRDL} & \ 2,16 & \text{STH} & \ 2,A & \text{LH} & \ 3,A+2 \\
\text{LH} & \ 2,A+2 & \text{LH} & \ 2,A+2 & \text{SRL} & \ 2,16 & \text{STH} & \ 2,A+2 \\
\text{SLDL} & \ 2,16 & \text{SRDL} & \ 2,16 & \text{STH} & \ 2,A+2 & \text{STH} & \ 3,A \\
\text{ST} & \ 2,A & \text{ST} & \ 3,A \\
\end{align*}
\]

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

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

Figure 14.7 Four Integers Packed in a Fullword

A sequence of instructions which unpacks the integers and places them in the fullwords labeled FIRST, SECOND, THIRD, and FOURTH, follows; assume that R9 contains the address of the data word. The comment statements give the binary contents of RO and R1: the bits of the integers are labeled A, B, C, and D; X represents a bit whose value is unknown, and 0 is a 0 bit. The "." is simply to indicate the boundary between RO and R1.
Another code sequence to do the same task is:

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

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

The arithmetic shift instructions are almost identical to the logical shift instructions, with the differences being in the setting of the CC and the treatment of the sign bit. The instructions are SLA (Shift Left...
Arithmetic), SRA (Shift Right Arithmetic), SLDA (Shift Left Double Arithmetic),
and SRDA (Shift Right Double Arithmetic). On right shifts, the sign bit is
duplicated in the vacated sign position after each unit shift; thus the
arithmetic integrity of the shifted operand is maintained. To illustrate
the difference between logical and arithmetic shifts, suppose a right shift
of two places is performed on a register containing \texttt{FFFFF8}:

\begin{verbatim}
L 0,=F'-8'
SRL 0,2
L 0,=F'-8'
SRA 0,2
\end{verbatim}

After the logical shift, \(C(RO)=3FFFFF2\), and after the arithmetic shift,
\(C(RO)=FFFFF2\). For positive operands, the SRL and SRA instructions will
leave identical results in the register shifted; SRA will set the CC but
SRL will not. The instruction SRDA is similar to SRA except that an even-odd
register pair is shifted.

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

\begin{verbatim}
L 0,=F'-8'
SRL 0,2
SLA 0,4
C(RO)=FFFFF8, CC UNCHANGED
C(RO)=3FFFFF2, CC UNCHANGED
C(RO)=7FFFFF20, CC SET TO 3, OVERFLOW
\end{verbatim}

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

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

Figure 14.8 CC Settings after Arithmetic Shifts
A CC value of 3 is not possible after the SRA and SRDA instructions. Note that because the result tested for CC settings for SLDA and SRDA is a double-length operand, these instructions provide a simple means for testing whether both registers contain zero: both SRDA 0,0 and SLDA 0,0 will set the CC to zero if RO and R1 contain zero.

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

\[
\begin{align*}
L & \quad 3,=F'5' & C(R3) = 00000005 \\
SRA & \quad 3,1 & C(R3) = 00000002 \\
L & \quad 3,=F'-5' & C(R3) = FFFFFFFF = -5 \\
SRA & \quad 3,1 & C(R3) = FFFFFFFD = -3
\end{align*}
\]

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

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

\[
\begin{align*}
SRA & \quad 9,4 & SRL & \quad 9,4 \\
SLA & \quad 9,4 & SLL & \quad 9,4
\end{align*}
\]

The logical shifts can be used because whatever bit is shifted out of the sign position by the SRL instruction is put back by SLL. If a CC setting is desired to indicate the status of the result, then the first code sequence must be used; if not, the second is preferable because it will operate slightly faster, because the CPU need not bother with duplicating the sign bit nor checking for overflow.
To conclude our discussion of shifting, we will re-examine the problem of unpacking the data contained in the fullword pictured in Figure 14.7, on the supposition that the four integers are in signed two's complement representation rather than the unsigned logical representation assumed before. The following code segment stores the four signed integers as required.

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

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

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

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

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

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

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

If the branching condition is met, the branch address must be determined. For the BC instruction, the branch address is the same as the effective address computed as usual from the base, index, and displacement fields of the instruction; for the BCR instruction, the branch address is given by the rightmost 24 bits of the general register specified by the r2 digit of the instruction unless r2 is zero, in which case no branch ever occurs. To complete the execution of the branch instruction, the IA portion of the PSW is replaced by the branch address. The next instruction to be fetched will therefore come from the location specified by the branch address. Branch instructions are also called "jump" and "transfer" instructions, in the sense that a jump is made, or control is transferred, to the branch address.
Whether the branch condition is met or not is determined by examining the bits of the register specification digit \( r_1 \). Because this digit does not refer to \( Rr_1 \), but is treated simply as a bit pattern (called a mask), we will rewrite the operand field entries as \( m_1, d_2(x_2, b_2) \) and \( m_1, r_2 \) for the RX and RR cases respectively. Thus we can write \( BC \ 7,4(8,2) \) and \( BCR \ 9,4 \) in which the mask fields are \( 0111_2 \) and \( 1001_2 \) respectively. At execution time, a match is made between the 1 bits of the mask and the value of the CC, as indicated in Figure 15.1.

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

Figure 15.1 Mask Bits and Corresponding CC Values

If the CC has a value which matches a 1 bit in the mask field, the branching condition is met; if the CC has a value which matches a 0 bit in the mask, the branching condition is not met, and no branch occurs. Thus in the examples given above, the BC instruction would branch unless the CC had value 0, and the BCR would branch if the CC had value 0 or 3. Further examples are given below.
1) Branch to X if C(R12)=0.
   \[ \text{LTR 12,12 or SRA 12,0} \]
   \[ \text{BC 8,X or BC 8,X} \]

2) Branch to X if C(R0)\#0.
   \[ \text{LTR 0,0 or SLA 0,0} \]
   \[ \text{BC 6,X or BC 7,X} \]

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

3) Multiply C(R5) by 4 and branch to X if the result does not overflow.
   \[ \text{SIA 5,2 \quad \text{BC 14,X}} \]

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

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

5) Place -C(R2) in R8 and branch to X if the result is negative.
   \[ \text{LCR 8,2 \quad \text{BC 5,X}} \]

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

6) A positive nonzero fullword integer at N is to be shifted right as many places as necessary to insure that its rightmost bit is nonzero.
   a) Shift left into R4 until R5 has been vacated:
      \[ \text{L 5,N \quad \text{GET INTEGER}} \]
      \[ \text{L 4,'0' \quad \text{CLEAR R4}} \]
      \[ \text{SHIFT SLDL 4,1 \quad \text{SHIFT LEFT}} \]
      \[ \text{LTR 5,5 \quad \text{TEST R5}} \]
      \[ \text{BC 7,SHIFT \quad \text{BRANCH IF NOT ZERO}} \]
      \[ \text{ST 4,N \quad \text{STORE RESULT}} \]
b) Shift right, testing "lost" bits:

<table>
<thead>
<tr>
<th>L</th>
<th>4,N</th>
<th>GET INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRDL</td>
<td>4,1</td>
<td>SHIFT RIGHT</td>
</tr>
<tr>
<td>LTR</td>
<td>5,5</td>
<td>TEST SIGN NOT F5</td>
</tr>
<tr>
<td>BC</td>
<td>10,SHIFT</td>
<td>BRANCH IF NOT -</td>
</tr>
<tr>
<td>SLDL</td>
<td>4,1</td>
<td>MOVE BIT BACK</td>
</tr>
<tr>
<td>ST</td>
<td>4,N</td>
<td>STORE RESULT</td>
</tr>
</tbody>
</table>

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

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

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

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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Alignment Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{CN\textsuperscript{OP} 0,4}</td>
<td>Beginning of a fullword</td>
</tr>
<tr>
<td>\texttt{CN\textsuperscript{OP} 2,4}</td>
<td>Middle of a fullword</td>
</tr>
<tr>
<td>\texttt{CN\textsuperscript{OP} 0,8}</td>
<td>Beginning of a doubleword</td>
</tr>
<tr>
<td>\texttt{CN\textsuperscript{OP} 2,8}</td>
<td>Second halfword of a doubleword</td>
</tr>
<tr>
<td>\texttt{CN\textsuperscript{OP} 4,8}</td>
<td>Middle of a doubleword</td>
</tr>
<tr>
<td>\texttt{CN\textsuperscript{OP} 6,8}</td>
<td>Fourth halfword of a doubleword</td>
</tr>
</tbody>
</table>

Figure 15.2 \texttt{CN\textsuperscript{OP}} Alignments

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

\texttt{CN\textsuperscript{OP} 2,4} \hspace{1cm} \texttt{ALIGN TO MIDDLE OF WORD}

\texttt{BALR 14,15} \hspace{1cm} \texttt{TWO-BYTE INSTRUCTION}

\texttt{DC A(ANYTHING)} \hspace{1cm} \texttt{NO INTERVENING BYTES}

Note that we could not write

\texttt{DS OH}
\texttt{BALR 14,15}
\texttt{DC A(ANYTHING)}

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

```
BALR 14,15
DS OF
DC A(ANYTHING)
```
since the BALR could again fall on a fullword boundary, leaving two bytes between it and the constant which would be skipped by the Assembler; the contents of the skipped bytes at execution time may be arbitrary, since the Supervisor does not clear the area into which a program is about to be loaded.

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

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

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

Figure 16.1 Fixed-Point Arithmetic Instructions
There are several instructions missing from the table which one might expect to find: there are no logical halfword instructions, there is no "Divide Halfword", and there are no instructions for performing multiplication and division with logical operands. It is possible, however, to compute logical products and quotients using available instructions.

The operations of the add and subtract instructions are straightforward and are summarized in Figure 16.2 below. Remember that the logical add and subtract produce the same result as the arithmetic add and subtract instructions except that the CC is set differently. For the halfword operations, we may assume (as in the discussion of LH in Section 14) that the second operand is brought from memory to the MDR, extended to a fullword, and then used for the indicated operation. The notation "FW2" means the fullword operand at the effective memory address in the RX instructions, and "HW2" means the same for halfword operands.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
<th>CC Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR</td>
<td>C(Rr1) - C(Rr1)+C(Rr2)</td>
<td>0: Result is zero</td>
</tr>
<tr>
<td>SR</td>
<td>C(Rr1) - C(Rr1)-C(Rr2)</td>
<td>1: Result is &lt; 0</td>
</tr>
<tr>
<td>A</td>
<td>C(Rr1) - C(Rr1)+C(FW2)</td>
<td>2: Result is &gt; 0</td>
</tr>
<tr>
<td>S</td>
<td>C(Rr1) - C(Rr1)-C(FW2)</td>
<td>3: Overflow</td>
</tr>
<tr>
<td>AH</td>
<td>C(Rr1) - C(Rr1)+C(HW2)</td>
<td>4:</td>
</tr>
<tr>
<td>SH</td>
<td>C(Rr1) - C(Rr1)-C(HW2)</td>
<td>5:</td>
</tr>
<tr>
<td>ALR</td>
<td>C(Rr1) - C(Rr1)+C(Rr2)</td>
<td>0: Zero result, no carry</td>
</tr>
<tr>
<td>SLR</td>
<td>C(Rr1) - C(Rr1)-C(Rr2)</td>
<td>1: Nonzero result, no carry</td>
</tr>
<tr>
<td>AL</td>
<td>C(Rr1) - C(Rr1)+C(FW2)</td>
<td>2: Zero result, carry</td>
</tr>
<tr>
<td>SL</td>
<td>C(Rr1) - C(Rr1)-C(FW2)</td>
<td>3: Nonzero result, carry</td>
</tr>
</tbody>
</table>

Figure 16.2 Fixed-Point Add and Subtract Instructions

The CC settings in the rightmost column apply to all the instructions in the same part of the table. It is useful to note several aspects of the CC settings for the logical instructions, which depend on whether a carry occurs out of the leftmost position of Rr1, and whether the result is zero. By referring to the examples in Section 7, we can see that

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

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

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

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

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

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

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

To give another simple example of the use of some of these instructions, suppose we wish to compute \( NEWST\check{OCK} \) from the formula

\[
NEWST\check{OCK} = OLDST\check{OCK} + RECEIPTS - SALES
\]
where all quantities are fullword integers small enough to guarantee that no overflows occur. Both sets of statements below compute the desired result.

\[
\begin{align*}
&L 2, OLDSTOCK \\
&AL 2, RECEIPTS \\
&S 2, SALES \\
&SL 2, SALES \\
&ST 2, NEWSTOCK \\
&ST 2, NEWSTOCK
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Operations</th>
<th>CC Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>0: Operand 1 = Operand 2</td>
</tr>
<tr>
<td>C</td>
<td>1: Operand 1 &lt; Operand 2</td>
</tr>
<tr>
<td>CH</td>
<td>2: Operand 1 &gt; Operand 2</td>
</tr>
<tr>
<td>CLR</td>
<td>3: Operand 1 ≠ Operand 2</td>
</tr>
</tbody>
</table>

Figure 16.3 CC Settings for Compare Instructions

The CC cannot be set to 3 as a result of a compare instruction. It can be seen for the CR, C, and CH instructions that the CC setting is the same as would result from performing SR, S, and SH instructions with the same operands, assuming that no overflow occurs. In fact, this is how the comparison is done by the CPU -- a subtraction is performed internally and the CC is set to reflect the sign and the magnitude of the difference, which would have been placed back in Rr1 for the subtract instructions. Further analysis of the original operands is required in the CPU if the internal result overflows. The logical comparisons do not give the same results as arithmetic comparisons, since numbers in the logical representation are always considered to be positive. The following instruction sequence may help to illustrate the differences.
The last of the statements in the above example is a programming error that occasionally occurs; note that the Assembler gives no indication of the conflicting data types implied by the instruction and the operand.

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

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

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

```
SR    7,7
SRDL  6,3
LTR   7,7
BC    8,A
A     6,'F'1'
A     SLL 6,3
```

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

```
A     7,'F'7'
SRL  7,3
SLL  6,3
```

where in this case the presence of any 1 bit in the three rightmost bit positions of the original number cause a carry into the 23 bit position (that is, bit 28 of R6); the result is the same as before except for the final CC setting.

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

```
L 0, =F'-1'
LR 1, 0
S 0, ARG
S 1, ARG+4
AL 1, =F'1'
BC 12, NC
A 0, =F'1'
NC STM 0,1, ARG
```

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

```
LM 0,1, ARG
LCR 0, 0
LCR 1, 1
BC 8, X
S 0, =F'1'
X STM 0,1, ARG
```

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

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

```
LM  0,1,A
AL  1,8+4
BC  12,NC
A   0,=F'1'
NC  0,3
STM 0,1,C
C   DS  D
B   DC  FLH'22233444555'
A   DC  FLH'9877666555'
```

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

```
LM  0,1,A
SL  1,3+4
BC  3,CA
S   0,=F'1'
CAR S 0,3
STM 0,1,C
C   DS  N
B   DC  FLH'123456787654321'
A   DC  FLH'234567+987654321'
```

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

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

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

\[
\begin{align*}
MR & 2,7 \quad C(R2,R3) = C(R3) \times C(R7) \\
MR & 0,1 \quad C(R0,R1) = C(R1) \times C(R1) \\
MR & 8,8 \quad C(R8,R9) = C(R8) \times C(R9) \\
M & 4,X \quad C(R4,R5) = C(R5) \times C(X) \\
M & 12,=F'932' \quad C(R12,R13) = C(R13) \times 932 \\
LR & 4,4 \quad \text{MOVE MULTIPLICAND TO R5} \\
MR & 4,4 \quad C(R4,R5) = C(R5) \times C(R4)
\end{align*}
\]

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

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

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

we would find that the fullword stored at \( \text{PRODUCT} \) was zero and that \( C(R0) = 1 \); and if we executed the instruction sequence (note that 32768 = \( 2^{15} \))
we would find that \( C(\text{PRODUCT}) = -2^{31} \). There are thus two situations the programmer should be aware of: first, that the size of the product may be such that it overflows the low-order register, and second, that whether or not the high-order register contains significant bits, the leftmost bit of the low-order register is \textit{not} a sign bit, but contains an arithmetically significant digit.

The \textit{MH} instruction produces a single-length result, which is the low-order 32 bits of the product of \( C(R_1) \) and the halfword second operand. Because only a fullword result is retained, \( r_1 \) need not be even, and a specification exception will occur only if the effective address of the halfword operand is odd. Because fewer shifts and adds are needed during multiplication, some small economies may be achieved by the use of \textit{MH}, particularly on the smaller models of System/360. Thus, \textit{MH 5,=H'100'} is a simple way to multiply the contents of \( R_5 \) by 100. If \( X \) and \( Y \) are both halfword operands, their product may be found by writing

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

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

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

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

As an example of the use of a multiply instruction, suppose we want to calculate \( A = B + G \times D \), where all quantities are fullword integers, and it is assumed that all results are small enough so that no overflows occur.
Note that we have used the letters A, B, G, and D to denote both the names of fullword areas of memory and the names of the contents of these areas; this usage is typical of procedural languages, where little distinction is made between the name associated with an area of memory, the contents of the area, and the value associated with the contents. We will explore such considerations further after more data representations have been discussed.

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

\[
\sum_{k=1}^{n} k^3.
\]

A slightly different version of the same program which counts from \( n \) down to 1 follows.
Division is always performed using a double-length dividend and remainder. As was the case for the fullword multiply instructions, the r1 digit must be even, and specifies the register pair containing the dividend; the CC is unaffected. As indicated in Section 8, the quotient replaces the low-order half of the dividend in the odd-numbered register, and the remainder replaces the high-order part of the dividend in the even-numbered register; if a valid quotient cannot be computed, a fixed-point divide exception occurs.

For example, to divide the double-length number in (R8,R9) by the number in R13, we can write DR 8,13 and to divide the same number by 10 we could write D 8,=F'10'. To illustrate the use of a divide instruction, suppose we want to compute the product of C(A) and C(B), and force the result to the next largest multiple of 29 if it is not already a multiple. We will assume that the product is small enough that a fixed-point divide exception will not occur when dividing by 29, and that the final result is contained in a single fullword.

\[
\begin{align*}
L & \quad 3,A \\
M & \quad 2,3 \\
D & \quad 2,=F'29' \\
LDR & \quad 2,2 \\
BC & \quad 8,MPY \\
A & \quad 3,=F'1' \\
MPY & \quad 2,=F'29' \\
ST & \quad 3,RESULT
\end{align*}
\]

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

\[
\begin{align*}
L & \quad 7,N \\
SR & \quad 6,6 \\
D & \quad 6,=F'10' \\
C & \quad 6,=F'5' \\
BC & \quad 4,OKAY \\
A & \quad 7,=F'1' \\
OKAY & \quad 7,2
\end{align*}
\]

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

16-12
First, the initial value of the dividend would not have a correctly extended sign bit for negative arguments; second, because the sign of the remainder is always the same as the sign of the original dividend, the compare instruction would always (when \( C(N) \) is negative) cause the following branch instruction to transfer control to OKAY independent of the magnitude of the remainder. To obtain a correctly represented dividend it is simplest to use the SRDA instruction, as shown.

| L 1,=F'1' | SET UP ROUNding BIT |
| L 6,N | \( C(R6) = C(N) \) |
| SRDA 6,32 | \( C(R6,R7) = 64\text{-BIT DIVIDEND} \) |
| BC 11;DIV | JUMP IF NON-NEGATIVE DIVIDEND |
| LCR 1,1 | OTHERWISE SET ROUNDOFF TO \(-1\) |
| DIV D 6,=F'10' | DIVIDE BY 10 |
| LPR 6,6 | ABSOLUTE VALUE OF REMAINDER |
| C 6,=F'5' | COMPARE TO 5 |
| BC 4,OKAY | BRANCH IF SMALLER THAN 5 |
| AR 7,1 | ADD CORRECTLY-SIGNED ROUNDOFF |
| OKAY ST 7,0 | STORE ROUNDED QUOTIENT |

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

\[
|C(Rr_1)| < \frac{1}{2} \text{ second operand}
\]

is satisfied, the quotient can be computed correctly.
17. LOGICAL OPERATIONS AND INSTRUCTIONS

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

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

```
L O,A
A 0,B
BC 14,ST
LA 0,0
ST ST 0,ANSWER
```
Because the LA instruction computes an effective address, it also provides a simple way to increment the contents of a register by a small positive amount. For example, LA 4,17(0,4) will increase the contents of R4 by 17, if the original contents of R4 are between -17 and 2^{24}-18. This restriction is of course due to the fact that the high-order byte of the register into which the result is placed will be set to zero; thus the use of LA for incrementing registers is usually limited to cases where the quantity being incremented is an address or reasonably small integer. For example, suppose we want to perform the shifting operation described in example 6 of Section 15, where it was required that the fullword at N be shifted right enough places so that its rightmost bit is a 1 bit; we will also require that the halfword at COUNT contain the number of positions shifted.

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

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

A third use of the LA instruction, and possibly the most important, is in generating addresses for actual operands in memory. For example, we may require the address of some operand to be in a given register during the execution of a segment of code. Suppose we want to add three integers, and branch after all additions are completed to NOERR if no overflow occurs, and to ERR1 if one or more overflows occur. Let the integers be stored in successive fullwords beginning at Q.
It should be noted that the instruction with a mask digit of 15 could also be written BC 1,ERR1 without affecting the operation of the code, since the instruction is reached only if the branching condition for the immediately preceding instruction is not met; by specifying a mask of 15 it is clear that the branch must always be taken. There is one important assumption underlying the use of the two LA instructions: the instructions named NOERR and ERR1 must be addressable, since the LA instruction will simply perform the address computation specified by the base and displacement assigned by the Assembler. As mentioned earlier, we are assuming that all symbols (and expressions such as Q+8) are addressable and that the appropriate base register information has been established elsewhere in the program.

It is occasionally easy to forget that the symbols used in LA instructions must be addressable, since no reference is being made to any memory location -- only an address is being generated, and no checks for the validity of that address are made.

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

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

```
\[
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
1 & 1 & 1
\end{array}
\]

\text{AND}

\[
\begin{array}{ccc}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
1 & 1 & 1
\end{array}
\]

\text{OR}

\[
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 1 & 1
\end{array}
\]

\text{EXCLUSIVE OR}
```

Figure 17.1 Logical Functions in System/360
In the first case, the result bit is 1 only if the first AND the second operand bits are 1; in the second case the result bit is 1 if either the first OR the second operand bits (or both) is 1; and in the last case, the result bit is 1 if either the first OR second operand bits is 1, EXCLUSIVE of the case where both are 1. Henceforth we will abbreviate EXCLUSIVE OR by XOR. For the instructions listed in Figure 17.2, the operands are fullwords; however, the result of the operation is obtained by matching the corresponding bits of each word, with no interactions between neighboring bits. A few examples will help to clarify this. As before, "FW₂" means the fullword second operand specified by the effective address.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Type</th>
<th>Action</th>
<th>CC Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>NR</td>
<td>RR</td>
<td>C(R₁) ← C(R₁) ∧ C(R₂)</td>
<td>0: all result bits are zero</td>
</tr>
<tr>
<td>N</td>
<td>RX</td>
<td>C(R₁) ← C(R₁) ∧ C(FW₂)</td>
<td></td>
</tr>
<tr>
<td>φR</td>
<td>RR</td>
<td>C(R₁) ← C(R₁) v C(R₂)</td>
<td>1: result bits are not all zero</td>
</tr>
<tr>
<td>φ</td>
<td>RX</td>
<td>C(R₁) ← C(R₁) v C(FW₂)</td>
<td></td>
</tr>
<tr>
<td>XR</td>
<td>RR</td>
<td>C(R₁) ← C(R₁) ⊕ C(R₂)</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>RX</td>
<td>C(R₁) ← C(R₁) ⊕ C(FW₂)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 17.2 Logical Instructions

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

NR 4,9
01214521₁₆

φR 4,9
EDA96567₁₆

XR 4,9
EC8A2046₁₆

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

<table>
<thead>
<tr>
<th>3</th>
<th>0011</th>
<th>3</th>
<th>0011</th>
<th>3</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>^9</td>
<td>A1001</td>
<td>∨ 9</td>
<td>v1001</td>
<td>⊕ 9</td>
<td>⊕1001</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>B</td>
<td>1011</td>
<td>A</td>
<td>1010</td>
</tr>
</tbody>
</table>

Figure 17.3 Examples of Logical Operations
One important use of the N and NR instructions is for "masking" operations in which it is desired to isolate or extract portions of a word. For example, suppose we wanted only the third of the four positive integers packed in the data word illustrated in Figure 14.7. This could be done by shifting as follows:

\[
\begin{align*}
L & \quad 0,\text{DATAWORD} & \quad \text{GET INTEGERS} \\
SRL & \quad 0,6 & \quad \text{DROP OFF FOURTH ONE} \\
SRDL & \quad 1,13 & \quad \text{MOVE THIRD INTO R1} \\
SRL & \quad 1,19 & \quad \text{POSITION FOR STORING} \\
ST & \quad 1,\text{THIRD}
\end{align*}
\]

or as follows:

\[
\begin{align*}
L & \quad 0,\text{DATAWORD} & \quad \text{DROP OFF FIRST AND SECOND INTEGERS} \\
SLL & \quad 0,13 & \quad \text{DROP OFF FOURTH, POSITION FOR STORING} \\
SRL & \quad 0,19 & \quad \text{DROP OFF FIRST AND SECOND INTEGERS} \\
ST & \quad 0,\text{THIRD}
\end{align*}
\]

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

\[
\begin{align*}
L & \quad 1,\text{DATAWORD} & \quad \text{AAAAAAABBBBCCDCCCCDCCCCC} \\
N & \quad 1,\text{MASK} & \quad \text{000000000000000000000000000} \\
SRL & \quad 1,6 & \quad \text{000000000000000000000000000} \\
ST & \quad 1,\text{THIRD} & \quad \text{000000000000000000000000000} \\
\text{DS} & \quad \text{OF} & \quad \text{STORE DESIRED INTEGER} \\
\text{MASK} & \quad \text{DC} & \quad X'0007F' \quad \text{ALIGN TO FULLWORD BOUNDARY}
\end{align*}
\]

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

To illustrate the use of a logical OR instruction, suppose we want to store a new value for the third integer into the proper part of the data word.
We can do this by shifting the various pieces into place:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L 0,DATAWORD</td>
<td>GET INTEGERS</td>
</tr>
<tr>
<td>SRDL 0,6</td>
<td>MOVE FOURTH INTO R1</td>
</tr>
<tr>
<td>L 0,NEWTHIRD</td>
<td>GET NEW VALUE OF THIRD INTEGER</td>
</tr>
<tr>
<td>SRDL 0,13</td>
<td>MOVE IT IN WITH FOURTH</td>
</tr>
<tr>
<td>L 0,DATAWORD</td>
<td>GET INTEGERS AGAIN</td>
</tr>
<tr>
<td>SRL 0,19</td>
<td>DROP OFF THIRD AND FOURTH</td>
</tr>
<tr>
<td>SRDL 0,13</td>
<td>MOVE FULL WORD INTO FOURTH</td>
</tr>
<tr>
<td>ST 1,DATAWORD</td>
<td>STORE NEW DATAWORD</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L 0,DATAWORD</td>
<td>GET INTEGERS</td>
</tr>
<tr>
<td>N 0,MASKA</td>
<td>CLEAR SPACE FOR THIRD</td>
</tr>
<tr>
<td>L 1,NEWTHIRD</td>
<td>GET NEW VALUE OF THIRD INTEGER</td>
</tr>
<tr>
<td>SLL 1,6</td>
<td>SHIFT INTO PROPER POSITION</td>
</tr>
<tr>
<td>OR 0,1</td>
<td>'OR' INTO PLACE</td>
</tr>
<tr>
<td>ST 0,DATAWORD</td>
<td>STORE NEW DATAWORD</td>
</tr>
</tbody>
</table>

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

The X and XR instructions are used mainly for inverting the value of a bit or a group of bits: it can be seen from Figure 17.1 that the result of XORing a 0 bit to any other bit is to leave it undisturbed, and the result of XORing a 1 bit is to invert it from 1 to 0 or vice versa. Thus, for example, we can form the one's complement of the number in R7 by subtracting it from a word of all 1 bits, or by executing X 7,=F'-1' which does the same thing. We can rewrite the example above to use an X instruction (though in a somewhat roundabout way) as follows:
As another example of the use of the XOR function, suppose we again want to force the integer in R9 to be the next larger multiple of 8 if it is not already a multiple of 8; consider the following code sequences.

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

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

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

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

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

| SR 8,8 | SET SHIFT COUNT TO ZERO |
| SHIFT SLA 6,1 | SHIFT LEFT ONE BIT POSITION |
| BC 1,FINISH | IF OVERFLOW, JUMP |
| LA 8,1(0,8) | INCREMENT SHIFT COUNT |
| BC 15,SHIFT | TRY AGAIN |

| FINIS SRA 6,1 | REPOSITION |
| X 6,DIGIT | RESTORE THE LOST BIT |
| ST 8,NORM | STORE SHIFT COUNT |

| NORM DS F |
| DIGIT DC X'40000000' |

17-7
In this case we shift left until the overflow indicates that a bit different from the sign bit has been shifted out of bit position 1. The right shift moves everything back, but instead of restoring the lost bit, extends the sign bit into the second bit position of R6 from which the most significant bit was just lost. Since the sign is known to be the opposite of the lost bit, the X operation inverts the second bit to give the desired result.
<table>
<thead>
<tr>
<th>C</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00*</td>
<td>C</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>01*</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>02*</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
</tr>
<tr>
<td>03*</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
</tr>
<tr>
<td>04*</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
</tr>
<tr>
<td>05*</td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
</tr>
<tr>
<td>06*</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
<td>109</td>
<td>110</td>
</tr>
<tr>
<td>07*</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
</tr>
<tr>
<td>08*</td>
<td>128</td>
<td>129</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td>135</td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
<td>140</td>
<td>141</td>
<td>142</td>
</tr>
<tr>
<td>09*</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
<td>148</td>
<td>150</td>
<td>151</td>
<td>152</td>
<td>153</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
<td>159</td>
</tr>
<tr>
<td>0A*</td>
<td>160</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
<td>165</td>
<td>166</td>
<td>167</td>
<td>168</td>
<td>169</td>
<td>170</td>
<td>171</td>
<td>172</td>
<td>173</td>
<td>174</td>
</tr>
<tr>
<td>0B*</td>
<td>176</td>
<td>177</td>
<td>178</td>
<td>179</td>
<td>180</td>
<td>181</td>
<td>182</td>
<td>183</td>
<td>184</td>
<td>185</td>
<td>186</td>
<td>187</td>
<td>188</td>
<td>189</td>
<td>190</td>
</tr>
<tr>
<td>0C*</td>
<td>192</td>
<td>193</td>
<td>194</td>
<td>195</td>
<td>196</td>
<td>197</td>
<td>198</td>
<td>199</td>
<td>200</td>
<td>201</td>
<td>202</td>
<td>203</td>
<td>204</td>
<td>205</td>
<td>206</td>
</tr>
<tr>
<td>0D*</td>
<td>208</td>
<td>209</td>
<td>210</td>
<td>211</td>
<td>212</td>
<td>213</td>
<td>214</td>
<td>215</td>
<td>216</td>
<td>217</td>
<td>218</td>
<td>219</td>
<td>220</td>
<td>221</td>
<td>222</td>
</tr>
<tr>
<td>0E*</td>
<td>224</td>
<td>225</td>
<td>226</td>
<td>227</td>
<td>228</td>
<td>229</td>
<td>230</td>
<td>231</td>
<td>232</td>
<td>233</td>
<td>234</td>
<td>235</td>
<td>236</td>
<td>237</td>
<td>238</td>
</tr>
<tr>
<td>0F*</td>
<td>240</td>
<td>241</td>
<td>242</td>
<td>243</td>
<td>244</td>
<td>245</td>
<td>246</td>
<td>247</td>
<td>248</td>
<td>249</td>
<td>250</td>
<td>251</td>
<td>252</td>
<td>253</td>
<td>254</td>
</tr>
<tr>
<td>10*</td>
<td>256</td>
<td>257</td>
<td>258</td>
<td>259</td>
<td>260</td>
<td>261</td>
<td>262</td>
<td>263</td>
<td>264</td>
<td>265</td>
<td>266</td>
<td>267</td>
<td>268</td>
<td>269</td>
<td>270</td>
</tr>
<tr>
<td>11*</td>
<td>272</td>
<td>273</td>
<td>274</td>
<td>275</td>
<td>276</td>
<td>277</td>
<td>278</td>
<td>279</td>
<td>280</td>
<td>281</td>
<td>282</td>
<td>283</td>
<td>284</td>
<td>285</td>
<td>286</td>
</tr>
<tr>
<td>12*</td>
<td>288</td>
<td>289</td>
<td>290</td>
<td>291</td>
<td>292</td>
<td>293</td>
<td>294</td>
<td>295</td>
<td>296</td>
<td>297</td>
<td>298</td>
<td>299</td>
<td>300</td>
<td>301</td>
<td>302</td>
</tr>
<tr>
<td>13*</td>
<td>304</td>
<td>305</td>
<td>306</td>
<td>307</td>
<td>308</td>
<td>309</td>
<td>310</td>
<td>311</td>
<td>312</td>
<td>313</td>
<td>314</td>
<td>315</td>
<td>316</td>
<td>317</td>
<td>318</td>
</tr>
<tr>
<td>14*</td>
<td>320</td>
<td>321</td>
<td>322</td>
<td>323</td>
<td>324</td>
<td>325</td>
<td>326</td>
<td>327</td>
<td>328</td>
<td>329</td>
<td>330</td>
<td>331</td>
<td>332</td>
<td>333</td>
<td>334</td>
</tr>
<tr>
<td>15*</td>
<td>336</td>
<td>337</td>
<td>338</td>
<td>339</td>
<td>340</td>
<td>341</td>
<td>342</td>
<td>343</td>
<td>344</td>
<td>345</td>
<td>346</td>
<td>347</td>
<td>348</td>
<td>349</td>
<td>350</td>
</tr>
<tr>
<td>16*</td>
<td>352</td>
<td>353</td>
<td>354</td>
<td>355</td>
<td>356</td>
<td>357</td>
<td>358</td>
<td>359</td>
<td>360</td>
<td>361</td>
<td>362</td>
<td>363</td>
<td>364</td>
<td>365</td>
<td>366</td>
</tr>
<tr>
<td>17*</td>
<td>368</td>
<td>369</td>
<td>370</td>
<td>371</td>
<td>372</td>
<td>373</td>
<td>374</td>
<td>375</td>
<td>376</td>
<td>377</td>
<td>378</td>
<td>379</td>
<td>380</td>
<td>381</td>
<td>382</td>
</tr>
<tr>
<td>18*</td>
<td>384</td>
<td>385</td>
<td>386</td>
<td>387</td>
<td>388</td>
<td>389</td>
<td>390</td>
<td>391</td>
<td>392</td>
<td>393</td>
<td>394</td>
<td>395</td>
<td>396</td>
<td>397</td>
<td>398</td>
</tr>
<tr>
<td>19*</td>
<td>400</td>
<td>401</td>
<td>402</td>
<td>403</td>
<td>404</td>
<td>405</td>
<td>406</td>
<td>407</td>
<td>408</td>
<td>409</td>
<td>410</td>
<td>411</td>
<td>412</td>
<td>413</td>
<td>414</td>
</tr>
<tr>
<td>1A*</td>
<td>416</td>
<td>417</td>
<td>418</td>
<td>419</td>
<td>420</td>
<td>421</td>
<td>422</td>
<td>423</td>
<td>424</td>
<td>425</td>
<td>426</td>
<td>427</td>
<td>428</td>
<td>429</td>
<td>430</td>
</tr>
<tr>
<td>1B*</td>
<td>432</td>
<td>433</td>
<td>434</td>
<td>435</td>
<td>436</td>
<td>437</td>
<td>438</td>
<td>439</td>
<td>440</td>
<td>441</td>
<td>442</td>
<td>443</td>
<td>444</td>
<td>445</td>
<td>446</td>
</tr>
<tr>
<td>1C*</td>
<td>448</td>
<td>449</td>
<td>450</td>
<td>451</td>
<td>452</td>
<td>453</td>
<td>454</td>
<td>455</td>
<td>456</td>
<td>457</td>
<td>458</td>
<td>459</td>
<td>460</td>
<td>461</td>
<td>462</td>
</tr>
<tr>
<td>1D*</td>
<td>464</td>
<td>465</td>
<td>466</td>
<td>467</td>
<td>468</td>
<td>469</td>
<td>470</td>
<td>471</td>
<td>472</td>
<td>473</td>
<td>474</td>
<td>475</td>
<td>476</td>
<td>477</td>
<td>478</td>
</tr>
<tr>
<td>1E*</td>
<td>480</td>
<td>481</td>
<td>482</td>
<td>483</td>
<td>484</td>
<td>485</td>
<td>486</td>
<td>487</td>
<td>488</td>
<td>489</td>
<td>490</td>
<td>491</td>
<td>492</td>
<td>493</td>
<td>494</td>
</tr>
<tr>
<td>1F*</td>
<td>496</td>
<td>497</td>
<td>498</td>
<td>499</td>
<td>500</td>
<td>501</td>
<td>502</td>
<td>503</td>
<td>504</td>
<td>505</td>
<td>506</td>
<td>507</td>
<td>508</td>
<td>509</td>
<td>510</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>E0</td>
<td>3584</td>
<td>3585</td>
<td>3586</td>
<td>3587</td>
<td>3588</td>
<td>3589</td>
<td>3590</td>
<td>3591</td>
<td>3592</td>
<td>3593</td>
<td>3594</td>
<td>3595</td>
<td>3596</td>
<td>3597</td>
<td>3598</td>
</tr>
<tr>
<td>E1</td>
<td>3600</td>
<td>3601</td>
<td>3602</td>
<td>3603</td>
<td>3604</td>
<td>3605</td>
<td>3606</td>
<td>3607</td>
<td>3608</td>
<td>3609</td>
<td>3610</td>
<td>3611</td>
<td>3612</td>
<td>3613</td>
<td>3614</td>
</tr>
<tr>
<td>E2</td>
<td>3616</td>
<td>3617</td>
<td>3618</td>
<td>3619</td>
<td>3620</td>
<td>3621</td>
<td>3622</td>
<td>3623</td>
<td>3624</td>
<td>3625</td>
<td>3626</td>
<td>3627</td>
<td>3628</td>
<td>3629</td>
<td>3630</td>
</tr>
<tr>
<td>E3</td>
<td>3632</td>
<td>3633</td>
<td>3634</td>
<td>3635</td>
<td>3636</td>
<td>3637</td>
<td>3638</td>
<td>3639</td>
<td>3640</td>
<td>3641</td>
<td>3642</td>
<td>3643</td>
<td>3644</td>
<td>3645</td>
<td>3646</td>
</tr>
<tr>
<td>E4</td>
<td>3648</td>
<td>3649</td>
<td>3650</td>
<td>3651</td>
<td>3652</td>
<td>3653</td>
<td>3654</td>
<td>3655</td>
<td>3656</td>
<td>3657</td>
<td>3658</td>
<td>3659</td>
<td>3660</td>
<td>3661</td>
<td>3662</td>
</tr>
<tr>
<td>E5</td>
<td>3664</td>
<td>3665</td>
<td>3666</td>
<td>3667</td>
<td>3668</td>
<td>3669</td>
<td>3670</td>
<td>3671</td>
<td>3672</td>
<td>3673</td>
<td>3674</td>
<td>3675</td>
<td>3676</td>
<td>3677</td>
<td>3678</td>
</tr>
<tr>
<td>E6</td>
<td>3680</td>
<td>3681</td>
<td>3682</td>
<td>3683</td>
<td>3684</td>
<td>3685</td>
<td>3686</td>
<td>3687</td>
<td>3688</td>
<td>3689</td>
<td>3690</td>
<td>3691</td>
<td>3692</td>
<td>3693</td>
<td>3694</td>
</tr>
<tr>
<td>E7</td>
<td>3696</td>
<td>3697</td>
<td>3698</td>
<td>3699</td>
<td>3700</td>
<td>3701</td>
<td>3702</td>
<td>3703</td>
<td>3704</td>
<td>3705</td>
<td>3706</td>
<td>3707</td>
<td>3708</td>
<td>3709</td>
<td>3710</td>
</tr>
<tr>
<td>E8</td>
<td>3712</td>
<td>3713</td>
<td>3714</td>
<td>3715</td>
<td>3716</td>
<td>3717</td>
<td>3718</td>
<td>3719</td>
<td>3720</td>
<td>3721</td>
<td>3722</td>
<td>3723</td>
<td>3724</td>
<td>3725</td>
<td>3726</td>
</tr>
<tr>
<td>E9</td>
<td>3728</td>
<td>3729</td>
<td>3730</td>
<td>3731</td>
<td>3732</td>
<td>3733</td>
<td>3734</td>
<td>3735</td>
<td>3736</td>
<td>3737</td>
<td>3738</td>
<td>3739</td>
<td>3740</td>
<td>3741</td>
<td>3742</td>
</tr>
<tr>
<td>EA</td>
<td>3744</td>
<td>3745</td>
<td>3746</td>
<td>3747</td>
<td>3748</td>
<td>3749</td>
<td>3750</td>
<td>3751</td>
<td>3752</td>
<td>3753</td>
<td>3754</td>
<td>3755</td>
<td>3756</td>
<td>3757</td>
<td>3758</td>
</tr>
<tr>
<td>EB</td>
<td>3760</td>
<td>3761</td>
<td>3762</td>
<td>3763</td>
<td>3764</td>
<td>3765</td>
<td>3766</td>
<td>3767</td>
<td>3768</td>
<td>3769</td>
<td>3770</td>
<td>3771</td>
<td>3772</td>
<td>3773</td>
<td>3774</td>
</tr>
<tr>
<td>EC</td>
<td>3776</td>
<td>3777</td>
<td>3778</td>
<td>3779</td>
<td>3780</td>
<td>3781</td>
<td>3782</td>
<td>3783</td>
<td>3784</td>
<td>3785</td>
<td>3786</td>
<td>3787</td>
<td>3788</td>
<td>3789</td>
<td>3790</td>
</tr>
<tr>
<td>ED</td>
<td>3796</td>
<td>3797</td>
<td>3798</td>
<td>3799</td>
<td>3800</td>
<td>3801</td>
<td>3802</td>
<td>3803</td>
<td>3804</td>
<td>3805</td>
<td>3806</td>
<td>3807</td>
<td>3808</td>
<td>3809</td>
<td>3810</td>
</tr>
<tr>
<td>EE</td>
<td>3818</td>
<td>3819</td>
<td>3820</td>
<td>3821</td>
<td>3822</td>
<td>3823</td>
<td>3824</td>
<td>3825</td>
<td>3826</td>
<td>3827</td>
<td>3828</td>
<td>3829</td>
<td>3830</td>
<td>3831</td>
<td>3832</td>
</tr>
<tr>
<td>EF</td>
<td>3842</td>
<td>3843</td>
<td>3844</td>
<td>3845</td>
<td>3846</td>
<td>3847</td>
<td>3848</td>
<td>3849</td>
<td>3850</td>
<td>3851</td>
<td>3852</td>
<td>3853</td>
<td>3854</td>
<td>3855</td>
<td>3856</td>
</tr>
<tr>
<td>F0</td>
<td>3860</td>
<td>3861</td>
<td>3862</td>
<td>3863</td>
<td>3864</td>
<td>3865</td>
<td>3866</td>
<td>3867</td>
<td>3868</td>
<td>3869</td>
<td>3870</td>
<td>3871</td>
<td>3872</td>
<td>3873</td>
<td>3874</td>
</tr>
<tr>
<td>F1</td>
<td>3882</td>
<td>3883</td>
<td>3884</td>
<td>3885</td>
<td>3886</td>
<td>3887</td>
<td>3888</td>
<td>3889</td>
<td>3890</td>
<td>3891</td>
<td>3892</td>
<td>3893</td>
<td>3894</td>
<td>3895</td>
<td>3896</td>
</tr>
<tr>
<td>F2</td>
<td>3904</td>
<td>3905</td>
<td>3906</td>
<td>3907</td>
<td>3908</td>
<td>3909</td>
<td>3910</td>
<td>3911</td>
<td>3912</td>
<td>3913</td>
<td>3914</td>
<td>3915</td>
<td>3916</td>
<td>3917</td>
<td>3918</td>
</tr>
<tr>
<td>F3</td>
<td>3920</td>
<td>3921</td>
<td>3922</td>
<td>3923</td>
<td>3924</td>
<td>3925</td>
<td>3926</td>
<td>3927</td>
<td>3928</td>
<td>3929</td>
<td>3930</td>
<td>3931</td>
<td>3932</td>
<td>3933</td>
<td>3934</td>
</tr>
<tr>
<td>F4</td>
<td>3936</td>
<td>3937</td>
<td>3938</td>
<td>3939</td>
<td>3940</td>
<td>3941</td>
<td>3942</td>
<td>3943</td>
<td>3944</td>
<td>3945</td>
<td>3946</td>
<td>3947</td>
<td>3948</td>
<td>3949</td>
<td>3950</td>
</tr>
<tr>
<td>F5</td>
<td>3952</td>
<td>3953</td>
<td>3954</td>
<td>3955</td>
<td>3956</td>
<td>3957</td>
<td>3958</td>
<td>3959</td>
<td>3960</td>
<td>3961</td>
<td>3962</td>
<td>3963</td>
<td>3964</td>
<td>3965</td>
<td>3966</td>
</tr>
<tr>
<td>F6</td>
<td>3972</td>
<td>3973</td>
<td>3974</td>
<td>3975</td>
<td>3976</td>
<td>3977</td>
<td>3978</td>
<td>3979</td>
<td>3980</td>
<td>3981</td>
<td>3982</td>
<td>3983</td>
<td>3984</td>
<td>3985</td>
<td>3986</td>
</tr>
<tr>
<td>F7</td>
<td>3988</td>
<td>3989</td>
<td>3990</td>
<td>3991</td>
<td>3992</td>
<td>3993</td>
<td>3994</td>
<td>3995</td>
<td>3996</td>
<td>3997</td>
<td>3998</td>
<td>3999</td>
<td>4000</td>
<td>4001</td>
<td>4002</td>
</tr>
<tr>
<td>F8</td>
<td>4004</td>
<td>4005</td>
<td>4006</td>
<td>4007</td>
<td>4008</td>
<td>4009</td>
<td>4010</td>
<td>4011</td>
<td>4012</td>
<td>4013</td>
<td>4014</td>
<td>4015</td>
<td>4016</td>
<td>4017</td>
<td>4018</td>
</tr>
<tr>
<td>F9</td>
<td>4032</td>
<td>4033</td>
<td>4034</td>
<td>4035</td>
<td>4036</td>
<td>4037</td>
<td>4038</td>
<td>4039</td>
<td>4040</td>
<td>4041</td>
<td>4042</td>
<td>4043</td>
<td>4044</td>
<td>4045</td>
<td>4046</td>
</tr>
<tr>
<td>FA</td>
<td>4048</td>
<td>4049</td>
<td>4050</td>
<td>4051</td>
<td>4052</td>
<td>4053</td>
<td>4054</td>
<td>4055</td>
<td>4056</td>
<td>4057</td>
<td>4058</td>
<td>4059</td>
<td>4060</td>
<td>4061</td>
<td>4062</td>
</tr>
<tr>
<td>FB</td>
<td>4064</td>
<td>4065</td>
<td>4066</td>
<td>4067</td>
<td>4068</td>
<td>4069</td>
<td>4070</td>
<td>4071</td>
<td>4072</td>
<td>4073</td>
<td>4074</td>
<td>4075</td>
<td>4076</td>
<td>4077</td>
<td>4078</td>
</tr>
<tr>
<td>FC</td>
<td>4080</td>
<td>4081</td>
<td>4082</td>
<td>4083</td>
<td>4084</td>
<td>4085</td>
<td>4086</td>
<td>4087</td>
<td>4088</td>
<td>4089</td>
<td>4090</td>
<td>4091</td>
<td>4092</td>
<td>4093</td>
<td>4094</td>
</tr>
</tbody>
</table>