Chapter 9: Instruction Formats

One of the major design decisions undertaken by computer architects is the choice of formats for the binary machine language instructions. The basic question revolves around the length of the instruction. Should all instructions have the same length, or should a variety of instruction lengths be allowed. This question is essentially a trade-off between complexity of the control unit and efficient use of memory space.

More bluntly, the big issue is the cost of random access memory for a computer. In the early 1960’s, memory was quite expensive. As an example, consider that a fully equipped NCR mainframe, shipped in 1966, had only 256 KB of memory, which cost $100,000 ($400,000 per megabyte). As late as 1979, memory cost was $75,000 per megabyte. As a result, a small System/360 might ship with only 16 KB to 64 KB installed. Within that context, the design emphasis was on an instruction set that made the most efficient use of memory.

For this reason, the S/360 instruction set provides for instruction lengths of 2 bytes, 4 bytes, and 6 bytes. This resulted in six instruction classes, each with an encoding scheme that allowed the maximum amount of information to be specified in a small number of bytes.

These formats are classified by length in bytes, use of the base registers, and object code format. The five instruction classes of use to the general user are listed below.

<table>
<thead>
<tr>
<th>Format</th>
<th>Length</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>2</td>
<td>Register to register transfers.</td>
</tr>
<tr>
<td>RS</td>
<td>4</td>
<td>Register to storage and register from storage</td>
</tr>
<tr>
<td>RX</td>
<td>4</td>
<td>Register to indexed storage and register from indexed storage</td>
</tr>
<tr>
<td>SI</td>
<td>4</td>
<td>Storage immediate</td>
</tr>
<tr>
<td>SS</td>
<td>6</td>
<td>Storage–to–Storage. These have two variants, each of which we shall discuss soon.</td>
</tr>
</tbody>
</table>

Before we launch on a formal description of these formats, it might be helpful to give some informal comments. We begin by noting that the opcode (machine code representation) of each instruction has a length of exactly one byte. With 8 bits to represent the opcode, this allows for 256 different operations, more if an extra encoding scheme is used.

Consider the register–to–register instructions. Since there are only 16 registers, each register can be fully specified by a 4–bit hexadecimal digit, and one byte will suffice to specify the two registers. Thus, the specification of one operation and two registers would require only 2 bytes. For this reason, the type RR instructions are encoded into two bytes of memory.

Consider now the mechanism used to specify an address. It calls for a base register (encoded in 4 bits) and a 12–bit address offset (encoded in 12 bits), for a total of 16 bits or two bytes. Given that the operation must specify a source or destination register, the sum grows to 20 bits. With the addition of an 8–bit opcode, the total grows to 28 bits, or 3.5 bytes. As fractional bytes cannot be accommodated in memory, the total is increased to four bytes and the instruction expanded to include two registers. The encoding is then 8 bits for the opcode, 8 bits for two source/destination registers, and 16 bits for the address: 4 bytes in all.
Branch Instructions
One of the encodings used to minimize the instruction size is to use the idea of a condition mask to extend two basic branch instructions into fourteen equivalent branch instructions. This device is often called “syntactic sugar” or extended mnemonics. There are two basic branch instructions in the IBM instruction set.

\[
\text{BC } \text{MASK, TARGET} \quad \text{A TYPE RX INSTRUCTION}
\]

\[
\text{BCR } \text{MASK, REGISTER} \quad \text{A TYPE RR INSTRUCTION}
\]

In the Type RX instruction, the target address is computed using the base register and displacement method, with an optional index register: D2(X2,B2). In the Type RR instruction, the target address is found as the contents of the register.

Each of these instruction formats uses a four–bit mask, with bit numbers based on the 2–bit value of the condition code in the PSW, to determine the conditions under which the branch will be taken. The mask should be considered as having bits numbered left to right as 0 – 3.

- Bit 0 is the equal/zero bit.
- Bit 1 is the low/minus bit.
- Bit 2 is the high/plus bit.
- Bit 3 is the overflow bit.

The Standard Combinations
The following table shows the standard conditional branch instructions and their translation to the BC (Branch on Condition). The same table applies to BCR (Branch on Condition, Register), so that there is another complete set of mnemonics for that set.

<table>
<thead>
<tr>
<th>Bit Mask Flags</th>
<th>Condition</th>
<th>Extended instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>Sort</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>No branch</td>
<td>BC 0,XX NOP</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>Bit 3: Overflow</td>
<td>BC 1,XX BO XX</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>Bit 2: High/Plus</td>
<td>BC 2,XX BH XX BP</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>Bit 1: Low/Minus</td>
<td>BC 4,XX BL XX BM</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>1, 2, 3: Not Equal</td>
<td>BC 7,XX BNE XX BNZ</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>Bit 0: Equal/Zero</td>
<td>BC 8,XX BE XX BZ</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>0, 2, 3: Not Low</td>
<td>BC 11,XX BNL XX BNM</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>0, 1, 3: Not high</td>
<td>BC 13,XX BNH XX BNP</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>0, 1, 2, 3: Any</td>
<td>BC 15,XX B XX</td>
</tr>
</tbody>
</table>

Note the two sets of extended mnemonics: one for comparisons and an equivalent set for the results of arithmetic operations.

These equivalent sets are provided to allow the assembler code to read more naturally.
The Idea of a Sort Order
Two data items of a specific data type are said to be “comparable” if they can be subjected to some sort of comparison operator with well defined results. The order used depends on the data type of the operands being compared. Note that it is not a valid operation to attempt comparison of operands of different data types. The basic comparison types are character (using EBCDIC code order), packed decimal, integer, and floating point.

One common operator that can be applied to many operations is that of equality, denoted “=”. The negation of equality is inequality, denoted “≠”. Remember that the assembler language syntax includes none of these algebraic symbols.

We also are interested in other comparisons, implied by what is called a “sort order”. Given two data items of the same type, it is convenient to define three operators.

\[ A > B \] if \( A \) follows \( B \) in the sort order.
\[ A = B \] if \( A \) and \( B \) occupy the same place in the sort order.
\[ A < B \] if \( A \) precedes \( B \) in the sort order.

Remember that each of these operators has an “opposite”.

\[
\begin{align*}
\text{If } A &> B \quad \text{then not } A \leq B. & \quad \text{Assembler pair: } BH \text{ and } BNH \\
\text{If } A &= B \quad \text{then not } A \neq B. & \quad \text{Assembler pair: } BE \text{ and } BNE \\
\text{If } A &< B \quad \text{then not } A \geq B. & \quad \text{Assembler pair: } BL \text{ and } BNL
\end{align*}
\]

Overflow: “Busting the Arithmetic”
Consider the half–word integer arithmetic in the IBM System/360. Integers in this format are 16–bit two’s complement integers with a range of –32,768 to 32,767.

Consider the following addition problem: 24,576 + 24,576.
Now + 24,576 (binary 0110 0000 0000 0000) is well within the range.

\[
\begin{align*}
0110 & \quad 0000 \quad 0000 \quad 0000 \quad 24576 \\
0110 & \quad 0000 \quad 0000 \quad 0000 \quad 24576 \\
1100 & \quad 0000 \quad 0000 \quad 0000 \quad -16384
\end{align*}
\]

What happened? We had a carry into the sign bit. This is “overflow”. The binary representation being used cannot handle the result. On the System/360, such an invalid operation will set the overflow bit.

Note that this sum will work very well in both 32–bit fullword arithmetic, as the true result is well within the range. It would also work in unsigned 16–bit arithmetic, except that the S/360 does not support that mode. Note that 24,576 + 24,576 = 49,152 = 32768 + 16384. This is the origin of the strange result from 16–bit signed arithmetic.

In mathematical terms, we would note that computers do not represent integers, but only a finite subset of the infinite set of integers.

Having discussed the branch instructions, let us now discuss the instruction formats.
The Object Code Format

Here is a table summarizing the formats of the five instruction types. Note that the fifth type has two variants, each of which will be explored in due turn.

<table>
<thead>
<tr>
<th>Format</th>
<th>Length</th>
<th>Explicit operand form</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>2</td>
<td>R1,R2</td>
<td>OP</td>
<td>R1</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS</td>
<td>4</td>
<td>R1,R3,D2(B2)</td>
<td>OP</td>
<td>R1</td>
<td>R3</td>
<td>D2</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>RX</td>
<td>4</td>
<td>R1,D2(X2,B2)</td>
<td>OP</td>
<td>R1</td>
<td>X2</td>
<td>D2</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>4</td>
<td>D1(B1),I2</td>
<td>OP</td>
<td>I2</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>SS(1)</td>
<td>6</td>
<td>D1(L,B1),D2(B2)</td>
<td>OP</td>
<td>L</td>
<td>B1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
</tr>
<tr>
<td>SS(2)</td>
<td>6</td>
<td>D1(L1,B1),D2(L2,B2)</td>
<td>OP</td>
<td>L1</td>
<td>L2</td>
<td>B1</td>
<td>D1</td>
<td>D2</td>
</tr>
</tbody>
</table>

NOTES:  
OP is the 8–bit operation code.
R1 R2 and R1 X2 each denote two 4–bit fields to specify two registers.
The two byte entry of the form B D D D denotes a 4–bit field to specify a base register and three 4–bit fields (12 bits) to denote a 12–bit displacement.
L denotes an 8–bit field for operand length (256 bytes maximum).
L1 and L2 each denote a 4–bit field for an operand length (16 bytes max.).
I denotes an 8–bit (one byte) immediate operand. These are useful.

RR (Register–to–Register) Format

This is a two-byte instruction of the form OP R1,R2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>OP</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>2</td>
<td>R1,R2</td>
<td>OP</td>
<td>R1</td>
<td>R2</td>
</tr>
</tbody>
</table>

The first byte contains the 8–bit instruction code.
The second byte contains two 4–bit fields, each of which encodes a register number.
This instruction format is used to process data between registers.

Here are some examples.

AR 6,8 1A 68  Adds the contents of register 8 to register 6.
AR 10,11 1A AB  Adds the contents of register 11 to register 10.
AR R6,R8 1A 68  Due to the standard Equate statements we use in our program assignments, R6 stands for 6 and R8 for 8.
RR (Register–to–Register) Format: Branch Instructions
There are two formats used with conditional branching instructions.

The BCR (Branch on Condition Register) instruction uses a modified form of the RR format. The BC (Branch on Condition) uses the RX format.

The BCR instruction is a two–byte instruction of the form OP M1,R2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>2</td>
<td>M1,R2</td>
</tr>
</tbody>
</table>

The first byte contains the 8–bit instruction code, which is X’07’.

The second byte contains two 4–bit fields.

The first 4–bit field encodes a branch condition.

The second 4–bit fields encodes the number of the register containing the target address for the branch instruction.

For example, the instruction BR R8 is the same as BCR 15,R8.

The object code is 07 F8. Branch unconditionally to the address in register 8.

We shall discuss the BC and BCR instructions in more detail at a later lecture.

RS (Register–Storage) Format
This is a four–byte instruction of the form OP R1,R3,D2(B2).

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>4</td>
<td>R1,R3,D2(B2)</td>
</tr>
</tbody>
</table>

The first byte contains the 8–bit instruction code.

The second byte contains two 4–bit fields, each of which encodes a register number.

Note that some RS format instructions use only one register, here R3 is set to 0.

In this instruction format, “0” is taken as no register, rather than register R0.

The third and fourth byte contain a 4–bit register number and 12–bit displacement, used to specify the memory address for the operand in storage.

Recall that each label in the assembly language program references an address, which must be expressed in the form of a base register with displacement.

Any address in the format of base register and displacement will appear in the form.

| B   | D1 | D2 | D3 |

B is the hexadecimal digit representing the base register.

The three hexadecimal digits D1 D2 D3 form the 12–bit displacement.
RS (Register–Storage) Format Examples
These are four–byte instructions of the form OP R1, R3, D2 (B2).

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>4</td>
<td>R1,R3,D2(B2)</td>
<td>OP</td>
<td>R1</td>
<td>R3</td>
<td>B2</td>
</tr>
</tbody>
</table>

1. Load Multiple  Operation code = X'98'.

Suppose the label FW3 (supposedly holding three 32–bit full–words) is at an address specified by offset X'100' from base register R7. Then we have

LM R5, R7, FW3  98 57 71 00

Unpacking the object code, we again find the parts.

The operation code is X'98', which indicates a multiple register load.

The next byte has value X'57', which indicates two registers: R5 and R7. Here it is used to represent a range of three registers: R5, R6, and R7.

The last two bytes contain the address of the label FW3. The two bytes 71 00 indicate

1) that the base address is contained in register R7, and

2) that the displacement from the base address is X'100'.

2. The above example with an explicit base register.

LA R4, FW3  Load the address FW3 into R4
LM R5, R7, 0(4)  The address is displaced 0 from the value in R4, the explicit base register

One might have an instruction of the following form, which is not equivalent to the above.

LM R5, R7, 12(4)  The address is displaced 12 (X'C') from the value in R4

3. Shift Left Logical  Operation code = X'89'

This is also a type RS instruction, though the appearance of a typical use seems to deny this. Consider the following instruction which shifts R6 left by 12 bits.

SLL R6, 12  Again, I assume we have set R6 EQU 6

The deceptive part concerns the value 12, used for the shift count. Where is that stored?

The answer is that it is not stored, but assembled in the form of a displacement of 12 to a base register of 0, indicating that no base register is used.

The above would be assembled as 89 60 00 0C  Decimal 12 is X'C'

Here are three lines from a working program I wrote on 2/23/2009.

000014 5840 C302 00308 47  L  R4, =F'1'
000018 8940 0001 00001 48  SLL  R4, 1
00001C 8940 0002 00002 49  SLL  R4, 2
RX (Register–Indexed Storage) Format

This is a four-byte instruction of the form OP R1,D2(X2,B2).

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>4</td>
<td>R1,D2(X2,B2)</td>
<td>OP</td>
<td>R1</td>
<td>X2</td>
<td>B2</td>
</tr>
</tbody>
</table>

The first byte contains the 8-bit instruction code.

The second byte contains two 4-bit fields, each of which encodes a register number.

In order to illustrate this, consider the following data layout.

```assembly
FW1 DC F'31'
DC F'100' Note that this full word is not labeled
```

Suppose that FW1 is at an address defined as offset X'123' from register 12. As hexadecimal C is equal to decimal 12, the address would be specified as C1 23.

The next full word might have an address specified as C1 27, but we shall show another way to do the same thing. The code we shall consider is

```assembly
L R4,FW1 Load register 4 from the full word at FW1.
The operation code is X'58'.
AL R4,FW1+4 Add the value at the next full word address.
The operation code is X'5E'.
```

The load instruction, remembering that the address of FW1 is specified as C1 23, the base register is R12, the displacement is X'123', and there is no index register; so we have

```
58 40 C1 23
```

The next instruction is similar, except for its operation code, which is 5E 40 C1 27. In each of the examples above, the 4-bit value X2 = 0. When a 0 is found in the index position, that indicates that indexed addressing is not used. Register 0 cannot be used as either a base register or an index register.

RX Format (Using an Index Register)

Here we shall suppose that we want register 7 to be an index register. Consider the three line sequence of instructions, in which R7 is given the value 4 to index from address FW1.

```assembly
L R7,=F'4' Register 7 gets the value 4.
L R4,FW1 Operation code is X'58'.
AL R4,FW1(R7) Operation code is X'5E'.
```

The object code for the last two instructions is now.

```
58 40 C1 23 This address is at displacement 123 from the base address, which is in R12.
Note X2 = 0, indicating no indexing.
5E 47 C1 23 R7 contains the value 4.
The address is at displacement 123 + 4 or 127 from the base address, in R12.
```
More on “Index Register 0”
Consider the instruction

\[ L \ R4,FW1 \] Operation code is X’58’.

The object code for this instruction is of the form

\[ 58 \ 40 \ C1 \ 23 \]

The second byte of the instruction has the destination register set as 4 (either decimal or hexadecimal; you choose), and the “index register” set to 0.

The intent of the instruction is that indexed addressing not be invoked.

There are two common ways to handle this addressing procedure.

1. The solution chosen by IBM is that the 0 indicates “do not index”, and that the value of register R0 is not used or changed.

2. Another common solution, tried as early as the CDC–6600, is to specify that register R0 stores the constant 0; R0 \( \equiv 0 \).

The 0 in the index register position would then indicate “index by R0”, that is to add 0 to the base–displacement address; in other words, no indexing.

Each method has its advantages. The second simplifies design of the control unit.

**RX (Register–Indexed Storage): Explicit Base Register**

This is a four–byte instruction of the form \[ OP \ R1,D2(X2,B2) \].

The second byte contains two 4–bit fields, each of which encodes a register number. The first hexadecimal digit, denoted R\(_1\), identifies the register to be used as either the source or destination for the data. The second hexadecimal digit, denoted X\(_2\), identifies the register to be used as the index. If the value is 0, indexed addressing is not used.

The third and fourth bytes contain a standard address in base/displacement format. We now consider the source code formats that use an explicit base register.

As an examples of this type, we consider the two following instructions:

\[ L \ \text{Load Fullword} \quad \text{Opcode is X’58’} \]
\[ A \ \text{Add Fullword} \quad \text{Opcode is X’5A’} \]

We consider a number of examples based on the following data declarations. Note that the data are defined in consecutive fullwords in memory, so that fixed offset addressing can be employed. Each fullword has a length of four bytes.

\[ \text{DAT1} \ \text{DC F’1111’} \]
\[ \text{DAT2} \ \text{DC F’2222’ \ AT ADDRESS (DAT1 + 4)} \]
\[ \text{DAT3} \ \text{DC F’3333’ \ AT ADDRESS (DAT2 + 4) OR (DAT1 + 8)} \]

A standard code block might appear as follows.

\[ L \ R5,\text{DAT1} \]
\[ A \ R5,\text{DAT2} \]
\[ A \ R5,\text{DAT3} \quad \text{NOW HAVE THE SUM.} \]
One variant of this code might be the following. See page 92 of R_17.

\[
\begin{align*}
&\text{LA R3, DAT1} & \text{GET ADDRESS INTO R3} \\
&\text{L R5, 0 (}, 3) & \text{LOAD DAT1 INTO R5} \\
&\text{A R5, 4 (}, 3) & \text{ADD DAT2, AT ADDRESS DAT1+4.} \\
&\text{A R5, 8 (}, 3) & \text{ADD DAT3, AT ADDRESS DAT1+8.}
\end{align*}
\]

Note the leading comma in the construct \((, 3)\), which is of the form (Index, Base). This indicates that no index register is being used, but that R3 is being used as a base register. It is synonymous with \((0, 3)\), which might be a preferable usage.

Here is another variant of the above code.

\[
\begin{align*}
&\text{LA R3, DAT1} & \text{GET ADDRESS INTO R3} \\
&\text{LA R8, 4} & \text{VALUE 4 INTO REGISTER 8} \\
&\text{LA R9, 8} & \text{VALUE 8 INTO REGISTER 9} \\
&\text{L R5, 0 (0, 3)} & \text{LOAD DAT1 INTO R5} \\
&\text{A R5, 0 (8, 3)} & \text{ADD DAT2, AT ADDRESS DAT1+4.} \\
&\text{A R5, 4 (8, 3)} & \text{ADD DAT3, AT ADDRESS DAT1+4+4.}
\end{align*}
\]

Here is yet another variant of the above code.

\[
\begin{align*}
&\text{LA R3, DAT1} & \text{GET ADDRESS INTO R3} \\
&\text{LA R8, 4} & \text{VALUE 4 INTO REGISTER 8} \\
&\text{L R5, 0 (0, 3)} & \text{LOAD DAT1 INTO R5} \\
&\text{A R5, 0 (8, 3)} & \text{ADD DAT2, AT ADDRESS DAT1+4.} \\
&\text{A R5, 4 (8, 3)} & \text{ADD DAT3, AT ADDRESS DAT1+4+4.}
\end{align*}
\]

The last line uses a displacement (the integer 4) from an indexed address (R8 is the index register) formed with an explicit base register (R3). It is a rather strange construct.

**RX Format (Branch on Condition)**

The **BC** (Branch on Condition) is a 4-byte instruction of the form

\[\text{OP M1, D2 (X2, B2)}\].

Its operation code is \texttt{X'47'}. Once again, the format is as follows.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>4</td>
<td>R1,D2(X2,B2)</td>
<td>47</td>
<td>M1 X2</td>
<td>B2 D2</td>
<td>D2D2</td>
</tr>
</tbody>
</table>

The first byte contains the 8-bit instruction code, which is \texttt{X'47'}. The second byte contains two 4-bit fields.

The first four bits contain the mask for the branch condition codes.

The second four bits contain the number of the index register used in computing the address of the jump target.

The next two bytes contain the 4-bit number of the base register and the 12-bit displacement used to form the unindexed address of the branch target.

Suppose that address \texttt{TARGET} is formed by offset \texttt{X'666'} using base register 8. No index is used and the instruction is \texttt{BNE TARGET}, equivalent to \texttt{BC 7, TARGET}, as the condition mask for “Not Equal” is the 4-bit number \texttt{0111}, or decimal 7.

The object code for this is \texttt{47 70 86 66}.
SI (Storage Immediate) Format
This is a four-byte instruction of the form \( \text{OP D1(B1),I2} \).

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td>4</td>
<td>D1(B1), I2</td>
<td>OP</td>
<td>I2</td>
<td>B1 D1</td>
<td>D1 D1</td>
</tr>
</tbody>
</table>

The first byte contains the 8–bit instruction code.

The second byte contains the 8–bit value of the second operand, which is treated as an immediate operand. The instruction contains the value of the operand, not its address.

The first operand is an address, specified in standard base register and displacement form.

Two instances of the instruction are:

- **MVI** Move Immediate
- **CLI** Compare Immediate

Suppose that the label ASTER is associated with an address that is specified using register R3 as a base register, with X’6C4’ as offset.

The operation code for MVI is X’92’ and the EBCDIC for ‘*’ is X’5C’.

**MVI ASTER, C ‘*’** is assembled as 92 5C 36 64.

The Storage–to–Storage Instructions
There are two formats for the SS (Storage–to–Storage) instructions. Each of the formats requires six bytes for the instruction object code. The two types of the SS are as follows:

1. **The Character Instructions**
   These are of the form \( \text{OP D1 (L,B1), D2 (B2)} \), which provide a length for only operand 1. The length is specified as an 8–bit byte.

   Examples:
   - **MVC** Move Characters
   - **CLC** Compare Characters

2. **The Packed Decimal Instructions**
   These are of the form \( \text{OP D1 (L1, B1), D2 (L2, B2)} \), which provide a length for each of the two operands. Each length is specified as a 4–bit hexadecimal digit.

   Examples:
   - **ZAP** Zero and Add Packed (Move Packed)
   - **AP** Add Packed
   - **CP** Compare Packed

Storage–to–Storage: Length Fields
Consider the two formats used to store a length in bytes. These are a four–bit hexadecimal digit and an eight–bit byte. Four bits will store an unsigned integer in the range 0 through 15. Eight bits will store an unsigned integer in the range 0 through 255. However, a length of 0 bytes is not reasonable for an operand. For this reason, the value stored is the one less than the length of the operand.
By examination of all instruction formats, we can show that only the SS (Storage–to–Storage) format instructions require length codes.

Storage–to–Storage: Character Instructions

These are of the form OP D1 (L, B1), D2 (B2), which provide a length for only operand 1. The length is specified as an 8–bit byte.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS(1)</td>
<td>6</td>
<td>D1(L,B1),D2(B2)</td>
<td>OP</td>
<td>L</td>
<td>B1</td>
<td>D1</td>
<td>D2</td>
<td>D3</td>
</tr>
</tbody>
</table>

The first byte contains the operation code, say X ‘D2’ for MVC or X ‘D5’ for CLC. The second byte contains a value storing one less than the length of the first operand, which is the destination for any move. Bytes 3 and 4 specify the address of the first operand, using the standard base register and displacement format. Bytes 5 and 6 specify the address of the second operand, using the standard base register and displacement format.

It is quite common for both operands to use the same base register.

Example of Character Instructions

Consider the example assembly language statement, which moves the string of characters at label CONAME to the location associated with the label TITLE.

MVC TITLE, CONAME

Suppose that:

1. There are fourteen bytes associated with TITLE, say that it was declared as TITLE DS CL14. Decimal 14 is hexadecimal E.

2. The label TITLE is referenced by displacement X ‘40A’ from the value stored in register R3, used as a base register.

3. The label CONAME is referenced by displacement X ‘42C’ from the value stored in register R3, used as a base register.

Given that the operation code for MVC is X ‘D2’, the instruction assembles as

D2 0D 34 0A 34 2C  Length is 14 or X ‘0E’; L – 1 is X ‘0D’
Explicit Base Addressing for Character Instructions
We now discuss a number of ways in which the operand addresses for character instructions may be presented in the source code. One should note that each of these source code representations will give rise to object code that appears almost identical. These examples are taken from Peter Abel [R_02, pages 271–273].

Assume that general–purpose register 4 is being used as the base register, as assigned at the beginning of the CSECT. Assume also that the following statements hold.

1. General purpose register 4 contains the value X’8002’.
2. The label PRINT represents an address represented in base/offset form as 401A; that is it is at offset X’01A’ from the value stored in the base register, which is R4. The address then is X’8002’ + X’01A’ = X’801C’.
3. Given that the decimal number 60 is represented in hexadecimal as X’3C’, the address PRINT+60 must then be at offset X’01A’ + X’3C’ = X’56’ from the address in the base register. X’A’ + X’C’, in decimal, is 10 + 12 = 16 + 6.
   Note that this gives the address of PRINT+60 as X’8002’ + X’056’ = X’8058’, which is the same as X’801C’ + X’03C’. The sum X’C’ + X’C’, in decimal, is represented as 12 + 12 = 24 = 16 + 8.
4. The label ASTERS is associated with an offset of X’09F’ from the value in the base register; thus it is located at address X’80A1’. This label references a storage of two asterisks. As a decimal value, the offset is 159.
5. That only two characters are to be moved by the MVC instruction examples to be discussed. Since the length of the move destination is greater than 2, and since the length of the destination is the default for the number of characters to be moved, this implies that the number of characters to be moved must be stated explicitly.

The first example to be considered has the simplest appearance. It is as follows:

\[
\text{MVC PRINT+60 (2), ASTERS}
\]

The operands here are of the form Destination (Length), Source.
The destination is the address PRINT+60. The length (number of characters to move) is 2. This will be encoded in the length byte as X’01’, as the length byte stores one less than the length. The source is the address ASTERS.

As the MVC instruction is encoded with opcode X’D2’, the object code here is as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS(1)</td>
<td>6</td>
<td>D1(L,B1),D2(B2)</td>
<td>OP</td>
<td>L</td>
<td>B1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>D2</td>
<td>01</td>
<td>40</td>
<td>56</td>
<td>40</td>
<td>9F</td>
</tr>
</tbody>
</table>

The next few examples are given to remind the reader of other ways to encode what is essentially the same instruction.
These examples are based on the true nature of the source code for a MVC instruction, which is \texttt{MVC D1(L,B1),D2(B2)}. In this format, we have the following.

1. The destination address is given by displacement \(D1\) from the address stored in the base register indicated by \(B1\).
2. The number of characters to move is denoted by \(L\).
3. The source address is given by displacement \(D2\) from the address stored in the base register indicated by \(B2\).

The second example uses an explicit base and displacement representation of the destination address, with general–purpose register 8 serving as the explicit base register.

\[
\text{LA R8,PRINT+60 GET ADDRESS PRINT+60 INTO R8 MVC 0(2,8),ASTERS MOVE THE CHARACTERS}
\]

Note the structure in the destination part of the source code, which is \(0(2,8)\).

\[
\text{Displacement} \uparrow \text{Base} \quad \text{Length}
\]

The displacement is 0 from the address \(X\ '8058'\,\), which is stored in R8. The object code is:

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS(1)</td>
<td>6</td>
<td>D1(L,B1),D2(B2)</td>
<td>OP</td>
<td>L</td>
<td>B1</td>
<td>D1</td>
<td>D1</td>
<td>B2</td>
</tr>
</tbody>
</table>

The instruction could have been written as \texttt{MVC 0(2,8),159(4)}, as the label ASTERS is found at offset 159 (decimal) from the address in register 4.

The third example uses an explicit base and displacement representation of the destination address, with general–purpose register 8 serving as the explicit base register.

\[
\text{LA R8,PRINT GET ADDRESS PRINT INTO R8 MVC 60(2,8),ASTERS SPECIFY A DISPLACEMENT}
\]

Note the structure in the destination part of the source code, which is \(60(2,8)\).

\[
\text{Displacement} \uparrow \text{Base} \quad \text{Length}
\]

The displacement is 60 from the address \(X\ '801C'\,\), stored in R8. The object code is:

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS(1)</td>
<td>6</td>
<td>D1(L,B1),D2(B2)</td>
<td>OP</td>
<td>L</td>
<td>B1</td>
<td>D1</td>
<td>D1</td>
<td>B2</td>
</tr>
</tbody>
</table>

The instruction could have been written as \texttt{MVC 60(2,8),159(4)}, as the label ASTERS is found at offset 159 (decimal) from the address in register 4.
Storage–to–Storage: Packed Decimal Instructions

These are of the form OP D1 (L1, B1), D2 (L2, B2), which provide a 4–bit number representing the length for each of the two operands.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS(2)</td>
<td>6</td>
<td>D1(L1,B1),D2(L2,B2)</td>
<td>OP</td>
<td>L1, L2</td>
<td>B1, D1</td>
<td>D1,D1</td>
<td>B2, D2</td>
<td>D2,D2</td>
</tr>
</tbody>
</table>

The first byte contains the operation code, say X'FA' for AP or X'F9' for CP.

The second byte contains two values, each a 4–bit binary number (one hex digit).

- L1: A value that is one less than the length of the first operand.
- L2: A value that is one less than the length of the second operand.

Bytes 3 and 4 specify the address of the first operand, using the standard base register and displacement format. Bytes 5 and 6 specify the address of the second operand, using the standard base register and displacement format. IBM will frequently call these decimal instructions. Here are two lines from the standard reference card, officially called FORM GX20–1850.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>Decimal Add</td>
<td>CP</td>
<td>Compare Decimal</td>
<td></td>
</tr>
</tbody>
</table>

Example of Packed Decimal Instructions

Consider the assembly language statement below, which adds AMOUNT to TOTAL.

**AP TOTAL, AMOUNT**

Assume:

1. TOTAL is 4 bytes long, so it can hold at most 7 digits.
2. AMOUNT is 3 bytes long, so it can hold at most 5 digits.
3. The label TOTAL is at an address specified by a displacement of X'50A' from the value in register R3, used as a base register.
4. The label AMOUNT is at an address specified by a displacement of X'52C' from the value in register R3, used as a base register.

The object code looks like this: **FA 32 35 0A 35 2C**

Consider **FA 32 35 0A 35 2C**. The operation code X'FA' is that for the Add Packed (Add Decimal) instruction, which is a type SS(2). The above format applies.

The field **32** is of the form L1 L2.

- The first value is X'3', or 3 decimal. The first operand is 4 bytes long.
- The second value is X'2', or 2 decimal. The second operand is 3 bytes long.

The two–byte field **35 0A** indicates that register 3 is used as the base register for the first operand, which is at displacement X'50A'. The two–byte field **35 2C** indicates that register 3 is used as the base register for the second operand, which is at displacement X'52C'. It is quite common for both operands to use the same base register.
Explicit Base Addressing for Packed Decimal Instructions

We now discuss a number of ways in which the operand addresses for character instructions may be presented in the source code. One should note that each of these source code representations will give rise to object code that appears almost identical. These examples are taken from Peter Abel [R_02, pages 273 & 274].

Consider the following source code, taken from Abel. This is based on a conversion of a weight expressed in kilograms to its equivalent in pounds; assuming 1kg. = 2.2 lb. Physics students will please ignore the fact that the kilogram measures mass and not weight.

```
ZAP  POUNDS, KGS
MP   POUNDS, FACTOR
SRP  POUNDS, 63, 5
```

KGS DC PL3 '12.53' LENGTH 3 BYTES
FACTOR DC PL2 '2.2' LENGTH 2 BYTES, AT ADDRESSS KGS+3
POUNDS DS PL5 LENGTH 5 BYTES, AT ADDRESS KGS+5

The value produced is 12.53•2.2 = 27.566, which is rounded to 27.57.

The instructions we want to examine in some detail are the **MP** and **ZAP**, each of which is a type SS instruction with source code format  **OP D1(L1,B1),D2(L2,B2)**. Each of the two operands in these instructions has a length specifier.

In the first example of the use of explicit base registers, we assign a base register to represent the address of each of the arguments. The above code becomes the following:

```
LA R6, KGS
LA R7, FACTOR
LA R8, POUNDS
ZAP 0(5,8), 0(3,6)
MP  0(5,8), 0(2,7)
SRP 0(5,8), 63, 5
```

Each of the arguments in the MP and ZAP have the following form:

```
0(5,8)   0(3,6)   0(2,7)
Offset   Offset   Offset
Base     Base     Base
Length   Length   Length
```

Recall the definitions of the three labels, seen just above. We analyze the instructions.

**ZAP 0(5,8), 0(3,6)**  Destination is at offset 0 from the address stored in R8. The destination has length 5 bytes.

Source is at offset 0 from the address stored in R6. The source has length 3 bytes.

**MP  0(5,8), 0(2,7)**  Destination is at offset 0 from the address stored in R8. The destination has length 5 bytes.

Source is at offset 0 from the address stored in R7. The source has length 2 bytes.
But recall the order in which the labels are declared. The implicit assumption that the labels are in consecutive memory locations will here be made explicit.

KGS     DC   PL3 ‘12.53’   LENGTH 3 BYTES  
FACTOR  DC   PL2 ‘2.2’     LENGTH 2 BYTES, AT ADDRESSS KGS+3  
POUNDS  DS   PL5         LENGTH 5 BYTES, AT ADDRESS KGS+5

In this version of the code, we use the label KGS as the base address and reference all other addresses by displacement from that one. Here is the code.

LA R6, KGS ADDRESS OF LABEL KGS
ZA 5(5,6), 0(3,6) 
MP 5(5,6), 3(2,6) 
SRP 5(5,6), 63, 5

Each of the arguments in the MP and ZAP have the following form:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Base</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>5(5,6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0(3,6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3(2,6)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recall the definitions of the three labels, seen just above. We analyze the instructions.

ZAP 5(5,6), 0(3,6) Destination is at offset 5 from the address stored in R6. The destination has length 5 bytes.  
Source is at offset 0 from the address stored in R6. The source has length 3 bytes.

MP 5(5,6), 3(2,6) Destination is at offset 5 from the address stored in R6. The destination has length 5 bytes.  
Source is at offset 3 from the address stored in R6. The source has length 2 bytes.

In other words, the base/displacement 6000 refers to a displacement of 0 from the address stored in register 6, which is being used as an explicit base register for this operation. As the address in R6 is that of KGS, this value represents the address KGS. This is the object code address generated in response to the source code fragment 0(3,6).

The base/displacement 6003 refers to a displacement of 3 from the address stored in register 6, which is being used as an explicit base register for this operation. As the address in R6 is that of KGS, this value represents the address KGS+3, which is the address FACTOR. This is the object code address generated in response to the source code fragment 3(2,6).

The base/displacement 6005 refers to a displacement of 5 from the address stored in register 6, which is being used as an explicit base register for this operation. As the address in R6 is that of KGS, this value represents the address KGS+5, which is the address POUNDS. This is the object code address generated in response to the source code fragment 5(5,6).

It is worth notice, even at this point, that the use of a single register as the base from which to reference a block of data declarations is quite suggestive of what is done with a DSECT, also called a “Dummy Section”.

Copyright © 2009 by Edward L. Bosworth, Ph.D.