Processing Character Data

We now discuss the definitions and uses of character data in an IBM Mainframe computer. By extension, we shall also be discussing **zoned decimal data**.

Character data and zoned decimal data are stored as **eight–bit bytes**.

These eight–bit bytes are seen by IBM as being organized into two parts.

This division is shown in the following table.

Portion	Zone				Numeric			
Bit	0	1	2	3	4	5	6	7

Note again the bit numbering scheme used by IBM.

Character constants have a few constraints.

1. Their length may be defined from 1 to 256 characters.
   Long character constants should be avoided.

2. They may contain any character. Characters not available in the standard set may be introduced by hexadecimal definitions.

3. The length may be defined either explicitly or implicitly.
   It is usually a good idea not to do both.

The EBCDIC Character Set

Here is the set of important EBCDIC codes.

Character	EBCDIC
blank	40
A – I	C1 – C9
J – R	D1 – D9
S – Z	E2 – E9
0 – 9	F0 – F9

Note that the EBCDIC codes for the digits ‘0’ through ‘9’ are exactly the zoned decimal representation of those digits. (But see below).

The **DS** declarative is used to reserve storage for character data.

The **DC** declarative is used to reserve initialized storage for character data.

We now cover a few topics:

1. Moving Character Data
2. Comparing Character Data
3. Literals and immediate instructions.

Other Character Sets

The original IBM System/360 supported the ASCII character set, but that was almost never used. The ASCII support was deleted from the IBM System/370 and has never reappeared.

Unicode™ is the only other character set given native support by the IBM Z–Series computers. Note that “Unicode” is a registered trademark of Unicode, Inc.

To support Unicode™ character representations, we see commands such as

`CLCLU`	Compare Logical Long Unicode
`MVCLU`	Move Long Unicode
`PKU`	Pack Unicode
`UNPKU`	Unpack Unicode

We shall not investigate the Unicode extensions, but they are there.
See section 7 of the IBM z/Architecture Principles of Operation , SA32–7832–06

Declaring Unicode Characters

All IBM z/System assembly languages use the DC construct for defining a label with a character value.

The later z/System languages use type extensions to define the values.

```
A1      DC  CA‘ASCII’

E1      DC  C‘EBCDIC’

E2      DC  CE‘EBCDIC’

U1      DC  CU‘Unicode’
```

The Unicode standard supported by the z/System is UTF–16, which calls for 16 bit (2 byte) representations for each character.

See also the manual
IBM High Level Assembler for z/OZ & z/VM & z/VSE
Language Reference, Release 6
SC26–4940–06

Zoned Decimal Data

The **zoned decimal format** is a modification of the EBCDIC format.

The zoned decimal format seems to be a modification to facilitate processing decimal strings of variable length.

The length of zoned data may be from 1 to 16 digits, stored in 1 to 16 bytes.

We have the address of the first byte for the decimal data, but need some "tag" to denote the last (rightmost) byte.

The assembler places a "sign zone" for the rightmost byte of the zoned data.

The common standard is **`X'C'`** for non–negative numbers, and **`X'D'`** for negative numbers.

Other than the placing of a hexadecimal digit **`X'C'`** or **`X'D'`** for the zone part of the last digit, the two representations are almost identical.

Consider the negative number **`–12345`**.

As a string of EBCDIC characters it is hexadecimal **`60 F1 F2 F3 F4 F5`**.

In the zoned decimal representation it is hexadecimal **`F1 F2 F3 F4 D5`**.

As packed decimal format it is stored as **`12 34 5D`**. Spaces are only for readability.

The MVC Instruction

The MVC (Move Character) instruction is designed to move character data, but it can be used to move data in any format, one byte at a time.

The instruction may be written as **`MVC DESTINATION,SOURCE`**

The format of the instruction is **`MVC D1(L,B1),D2(B2)`**

An example of the instruction is **`MVC F1,F2`**

Here are a few comments on MVC.

1. It may move from 1 to 256 bytes, determined by the use of an 8–bit number as a length field in the machine language instruction.

   The destination length is first decremented by 1 and then stored in the length byte. This disallows a length of 0, and allows 8 bits to store the value 256.

2. Data beginning in the byte specified by the source operand are moved one byte at a time to the field beginning with the byte in the destination operand.

   One of the reasons for complexity of the implementation is that the source and destination regions may overlap.

3. The length of the destination field determines the number of bytes moved.

More On MVC

The form is **`MVC D1(L,B1),D2(B2)`**. The object code format is as follows:

Type	Bytes	Form	1	2	3	4	5	6
SS(1)	6	D1(L,B1),D2(B2)	OP	L	B_1 D_1	D_1D_1	B_2 D_2	D_2D_2

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'
```

MVC: Explicit Register Usage

The instruction may be written in the form **`MVC D1(L,B1),D2(B2)`**

Consider the following example: **`MVC 32(5,7),NAME`**.

In this example, suppose that general–purpose register 7 has the value **`X'22400'`**.

We note that the label **`NAME`** represents an address that will be converted to the form **`D2(B2)`**; that is, a displacement from a base register. This base register might be register 7 or any of the ten registers (R3 – R12) available for general use.

We examine the specification of the first argument, which is the destination address. It is of the form **`D1(L,B1)`**.

The length is **`L`** = 5. This indicates that five characters are to be moved.

The displacement is decimal 32, or **`X'20'`**.

The address of the first character in the destination is given by adding this displacement to the contents of the base register: **`X'22400'`** + **`X'20'`** = **`X'22420'`**.

Five characters are moved to the destination. The fifth character is moved to a location that is four bytes displaced from the first character; its address is **`X'22424'`**.

MVC: Explicit Register Usage (Continued)

Consider again the example: **`MVC 32(5,7),NAME`**.

Suppose that the label **`NAME`** corresponds to an address given by offset **`X'250'`** (592 in decimal) from general–purpose register 10 (denoted in object code by **`X'A'`**).

When the instruction is written in the form **`MVC D1(L,B1),D2(B2)`**, we see that it has the form **`MVC 32(5,7),592(10).  ALL NUMBERS ARE DECIMAL.`**

In the object code format, the value stored for the length attribute is one less than the actual length. The length is 5, so the stored value is 4, or **`X'04'`**.

The object code format is **`D2 04 70 20 A2 50`**.

Again, recall the object code format for this instruction.

Op Code		Length		Base	Displacement			Base	Displacement		
D	**2**	**0**	**4**	**7**	**0**	**2**	**0**	**A**	**2**	**5**	**0**

MVC: Example 1

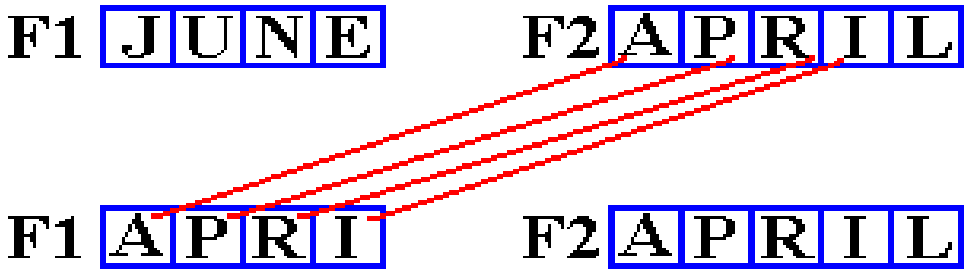
The number of bytes (characters) to move may be explicitly stated.

If the number is not explicitly stated, the number is taken as the length (in bytes or characters) of the destination field.

Consider the following program fragment.

```
MVC  F1,F2
F1   DC  CL4'JUNE'
F2   DC  CL5'APRIL'
```

What happens is shown in the next figure.



The assembler recognizes F1 as a four–byte field from its declaration by the DC statement. This implicitly sets the number of characters to be moved.

The 'L' is not moved, as it is the fifth character in F2. It is at address **F2+4**.

MVC: Example 2

The number of bytes (characters) to move may be explicitly stated.

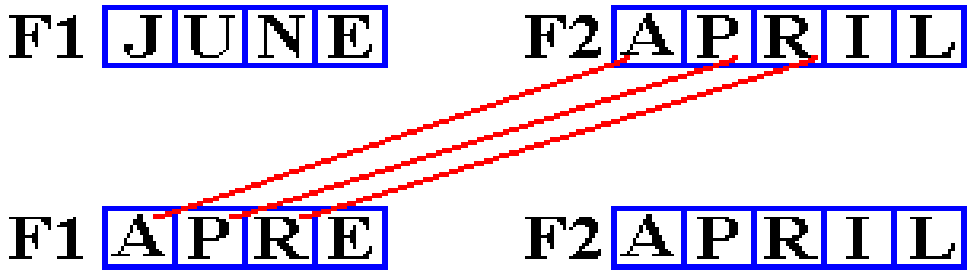
While the explicit length may exceed that of the destination field,
your instructor (but not many textbook authors) considers that bad programming practice.

Consider the following program fragment, in which an explicit length of 3 is set.

Recall the form of the instruction: **`MVC D1(L,B1),D2(B2)`**.

```
MVC F1(3),F2
F1   DC CL4'JUNE'
F2   DC CL5'APRIL'
```

What happens is shown in the next figure.



Note that only "APR" is moved. The last character of F1, which is an "E",
is not changed. This last character is at address **`F1+3`**.

MVC: Example 3

We may use relative addressing as well as an explicit length declaration.

Consider the following program fragment.

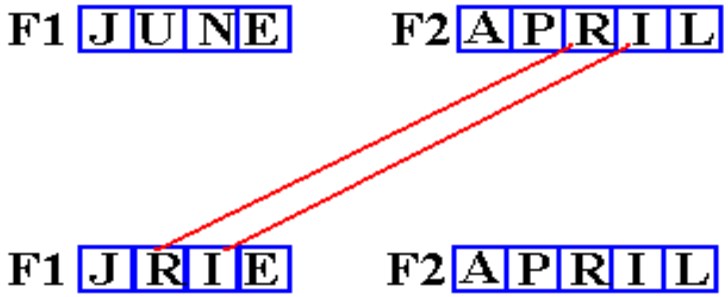
```
MVC  F1+1(2),F2+2
F1   DC  CL4'JUNE'
F2   DC  CL5'APRIL'
```

This calls for moving two characters from address **F2+2** to address **F1+1**.

What two characters are at address **F2+2**? Answer: **"RI"**.

What two characters are at address **F1+1**? Answer: **"UN"**.

What happens is shown in the next figure.



The other two characters in F1, at addresses **F1** and **F1+3**, are not changed.

MVC: Example 4

We now consider the explicit use of base registers.

Recall the form of the instruction: **`MVC D1(L,B1),D2(B2)`**.

In the following three examples, we suppose that **`PRINT`** is a label associated with an output field of length 80 bytes. In reality, it only must be "big enough".

```
FRAG01   MVC PRINT+60(2),=C'**'

FRAG02   LA  R8,PRINT+60       LOAD THE ADDRESS.
         MVC 0(2,8),=C'**'     DEST ADDRESS IS PRINT+60

FRAG03   LA  R8,PRINT          LOAD THE ADDRESS.
         MVC 60(2,8),=C'**'    NOTE OFFSET IS 60
```

Suppose that the address of **`PRINT`** is given by base register 12 and displacement **`X'200'`**. Suppose register 12 contains a value of **`X'1000'`**.

The label **`PRINT`** references address **`X'1200'`**.

The value of **`PRINT+60`** is then **`X'1200'`** + **`X'3C'`** = **`X'123C'`**.

We shall repeat this example and discuss similar examples in a future lecture on accessing arrays and tables.

Character Comparison: CLC

The **CLC (Compare Logical Character)** instruction is one of the two used to compare character fields, one byte at a time, left to right.

Comparison is based on the binary contents (EBCDIC code) contents of the bytes. The sort order is from X'00' through X'FF'.

The instruction may be written as **`CLC Operand1,Operand2`**

The format of the instruction is **`CLC D1(L,B1),D2(B2)`**

An example of the instruction is **`CLC NAME1,NAME2`**

This instruction sets the condition code that is used by the conditional branch instructions. The condition code is set as follows:

If Operand1 is equal Operand2 Condition Code = 0

If Operand1 is lower than Operand2 Condition Code = 1

If Operand1 is higher than Operand2 Condition Code = 2

The operation moves, byte by byte, from left to right and terminates as soon as an unequal comparison is found or one of the operands runs out.

Using the Condition Codes

The character comparison operators, CLC and CLI, set the condition codes.

These codes are used by the branching instructions in their non–numeric form.

Here are the standard comparisons.

BE	Branch Equal	Condition Code = 0
BNE	Branch Not Equal	Condition Code ≠ 0
BL	Branch Low	Condition Code = 1
BNL	Branch Not Low	Condition Code ≠ 1
BH	Branch High	Condition Code = 2
BNH	Branch Not High	Condition Code ≠ 2.

Here are two equivalent examples.

```
CLC     X, Y            CLC     X, Y
BL      J20LOEQ         BNH     J20LOEQ
BE      J20LOEQ
```

CLC: An Example

Consider the following code fragment. Note that the comparison value is given as the seven EBCDIC characters **'0200000'**.

Presumably, this would be converted into seven Packed Decimal digits and held to represent the fixed point number 2000.00, presumably $2,000.00.

```
C20       CLC SALPR,=C'0200000'      COMPARE TO 2,000.00
          BNH C30                    NOT ABOVE 2,000.00
          BL  C40                    LESS THAN 2,000.00
*         EQUAL TO 2,000.00
```

Again, this is presented as representing Packed Decimal data, which it probably does represent. The comparison, however, is an EBCDIC character comparison.

Here is another example, built around the first one. It represents an important special case that we shall consider when discussing Packed Decimal format.

```
C20       CLC  SALPR,=C'       '     IS THE FIELD BLANK?
          BNE  NOTBLNK
          MVC  SALPR,=C'0000000'     CONVERT BLANKS TO 0'S
NOTBLANK  PACK SALNUM,SALPR
```

MVI and CLI

These two operations are similar to their more general “cousins”, except that the second operand is a one–byte immediate constant.

The immediate constant may be of any of the following formats:

- B binary
- C character
- X hexadecimal

The format of these instructions are:

```
MVI Operand1,ImmediateOperand
CLI Operand1,ImmediateOperand
```

Examples of these instructions are:

```
MVI CONTROL,C'$'   Character '$'
CLI CODE,C'5'      Character '5'
```

Character Literals vs. Immediate Operands

The main characteristics of an immediate operation is that the operand, called the "immediate operand" is contained within the instruction.

The main characteristic of a literal operand is that it is stored separately from the operand, in a literal pool generated by the assembler.

Here are two equivalent instructions to set the currency sign.

Use of a literal: **`MVC DOLLAR,=C'$'`**

Use of immediate operand **`MVI DOLLAR,C'$'`**

Note the **`"="`** in front of the literal. It is not present in the immediate operand.

NOTE: The two instructions have the same affect in the program, however, they are different.

The opcode for MVC is **`X'D2'`**

The opcode for MVI is **`X'92'`**

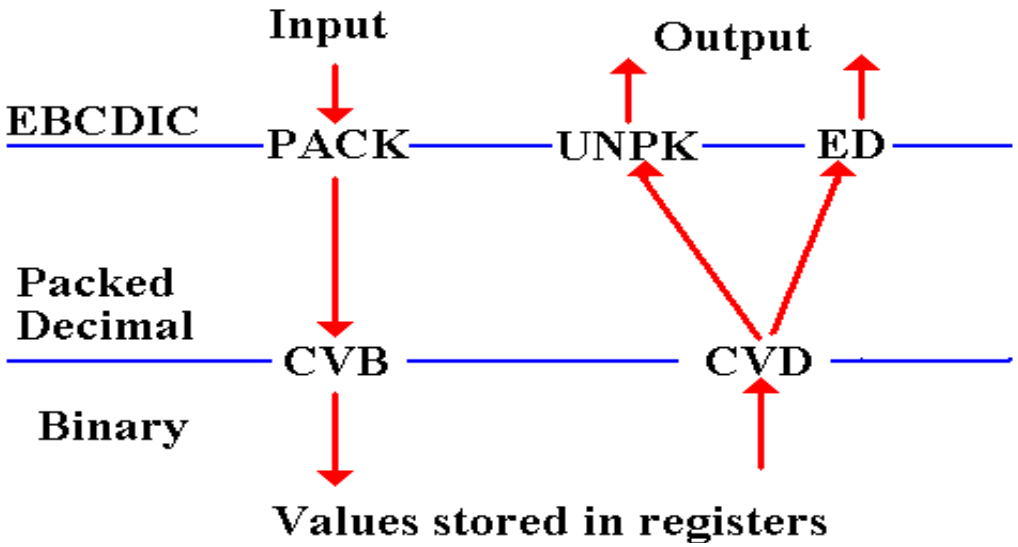
Look Ahead: Processing Packed Decimal and Integer Data

The standard way of processing any data that has print representation is to assume that the input will be in fixed pre–defined columns, as will the output.

All data are input and output as character data, in the EBCDIC format.

If the data are assumed to represent packed decimal values, the code must convert from EBCDIC to packed decimal format.

If the data are assumed to represent integer data, they are commonly converted twice: first to packed decimal format and then to integer format.



Outline of Code for Processing Free–Form Integer Input

As an exercise, we shall write and discuss code for the direct conversion of digital data, represented in EBCDIC code, to integer data in two's–complement form.

Here is the strategy for that conversion.

1. Scan left to right, looking for a non–blank character.
2. Is this a minus sign? Is this a digit?

Suppose that the label D has been defined as **`DS CL1`**, holding one character.
Here are some of the tests we run on the input.

```
CLI D,C' '  Is this a blank character?

CLI D, C'-'  Is it a minus sign?

CLI D, C'0'   Compare to the digit zero.

CLI D, C'9'   Compare to the digit nine.
```

To process as a digit, we require **`'0' ≤ D ≤ '9'`**.

More on Comparison

Here, I give two incomplete and purposely ambiguous definitions.
Each declaration is missing the data type, so neither can assemble.

```
X1      DC   '123C'

X2      DC   '123D'
```

First compare these with the CLC (Compare Logical Character) instruction.

Here `'1'` = `'1'`, `'2'` = `'2'`, `'3'` = `'3'`, and `'C'` < `'D'`, so X1 < X2.

Now compare these with the CP (Compare Packed Decimal) instruction.

Here X1 is interpreted as the positive number 123,
and X2 is interpreted as the negative number –123. So X1 > X2.

We shall see later that comparison of randomly chosen packed decimal fields
is not fully determined. This is due to the decimal point not being stored.

Support for Encryption

The modern IBM z/Architecture has direct assembly language support for secret key encryption using either the DEA (Data Encryption Algorithm) or the newer AES (Advanced Encryption Standard).

The two instructions are

`KM` Cipher Message

`KMC` Cipher Message with Chaining

Each of these supports standard DES, triple–DES, and the three variants of AES.

The options for DES are invoked with function codes (predefined constants) **`DEA`** and **`TDEA-192`**.

The options for AES are invoked with function codes **`AES-128`**, **`AES-192`**, or **`AES-256`**.

Source: IBM z/Architecture Principles of Operation
SA32–7832–06

Slightly More on the DES

There are two terms used almost as synonyms

DES the Data Encryption Standard

DEA the Data Encryption Algorithm

Each refers to the standard and algorithm described by IBM in the 1970's.

The DES was selected in 1976 by the U.S. National Bureau of Standards as the official standard for encrypting data that were unclassified but sensitive.

Data that were considered candidates for DES included bank records, orders for money transfer via the Internet, personnel records, etc.

Data that were not considered candidates for DES include anything classified by the U.S. Department of Defense (SECRET, Top Secret, etc.).

From a cryptologic viewpoint, DES is secure. However, it has been recently compromised by the huge amount of computing power.

In January 1999, a distributed computation attack on a DES message was able to retrieve the encryption key in 22 hours and 15 minutes.

Why Triple DES

Since the only known attack on DES involves brute force search, it can be made acceptably secure by lengthening the key.

The original DES uses a 56–bit key.

Triple DES uses three such keys, effectively giving a key length of 168 bits. Given current computer technology, a 168–bit key is secure against brute force.

The process for triple DES appears strange at first.

1. Encrypt with standard DES, using key K1.
2. Decrypt with standard DES, using key K2.
3. Encrypt with standard DES, using key K3.

The reason for this is that it allows a site using triple–DES to communicate with a site using standard DES.

The solution is to demand that K2 = K1, so that the input to stage 3 is the original message in plain text form.

There are some theoretical reasons not to use double–DES, but it is this practical commercial reason that causes triple–DES to be favored.