Chapter 7: Assembler Directives and Data Definitions

We begin the discussion of IBM Mainframe Assembler Language by making a broad distinction. Every program to be processed on a computer begins in a form that can be seen as a text file; that is, it is a sequence of statements that can be read by humans. The distinction to be made here is between those statements that will be converted into executable code and those that instruct the assembler to perform some specific function.

There are two specific classes of non–executable instructions: declaratives and directives. Declaratives are used mostly to allocate memory in which to store data. Directives are used for a number of functions that direct operation of the assembler. We shall discuss directives first. There are a number of directives available; here are the ones that we shall discuss.

CSECT	Identifies the start or continuation of a control section.
DSECT	Identifies the start or continuation of a dummy control section. A dummy section is used to pass data by reference to a subroutine.
EJECT	Start a new page before continuing the assembler listing. This is used to format a paper printout of the assembler text.
END	End of the assembler module or control section.
EQU	Equate a symbol to a name or number.
LTORG	Begin the literal pool.
PRINT	Sets some options for the assembly listing.
SPACE	Provides for line spacing in the assembler listing.
START	Define the start of the first control section in a program.
TITLE	Provide a title at the top of each page of assembler listing.
USING	Indicates the base registers to use in addressing.

CSECT

By definition, a **control section** (CSECT), is "a block of coding that can be relocated (independent of other coding) without altering the operating logic of the program."*****

Every program to be executed must have at least one control section. If the program has only one control section, as is usually the case for student programs, we may begin it with either a **CSECT** or **START** directive.

When used, a **START** directive "defines the start of the **first control section** in a program". We shall later discuss reasons why a program might need more than one control section. In this case, it is probably best to use only the **CSECT** directive.

DSECT

A **DSECT (Dummy Section)** is used to describe a data area without actually reserving any storage for it. This is used to pass arguments by reference from one program to another. Recall that there are two common ways for a program to pass arguments to another: call by value and call by reference. Students of programming languages will know several more.

In **call by value**, the value of an argument is passed to the receiving program, which can be considered to get only a local copy of the value. If the receiving program changes this value, the change is not reflected in the calling program. Some programming languages do not allow the receiving program to change the value of an argument passed by value; most do.

In **call by reference**, the address or some other identifier of the argument is passed to the receiving program. If the receiving program changes the value of the argument, the changed value is returned to the calling program.

Suppose that PROGA calls PROGB and passes a list of arguments. When a DSECT is used, what is passed is the starting address of a section of memory assigned to those arguments. The receiving program has a DSECT that mimics the structure of the section of memory in the calling program. This structure and the address of the data section in the calling program are used to compute the address (in the calling program) of every argument.

We shall discuss Dummy Sections in more detail later.

END

The **END** statement must be the last statement of an assembler control section.

The form of the statement is quite simple. It is

```
     END  Section_Name
```

So, our first program had the following structure.

```
LAB1    CSECT
     Some program statements
        END LAB1
```

Note that it easily could have been the following.

```
LAB1    START
     Some program statements
        END LAB1
```

EQU

The **EQU** directive is used to equate a name with an expression, symbolic address, or number. Whenever this name is used as a symbol, it is replaced.

We might do something, such as the following, which makes the symbol **R12** to be equal to 12, and replaced by that value when the assembler is run.

```
          R12    EQU  12
```

Consider the following two lines of assembler code, assumed to occur after the above EQU statement in the code. The original form uses "**R12**", and appears as follows.

```
         BALR  R12,0
         USING *,R12
```

The effect of the EQU directive is to change this text into the following before the assembler actually produces any machine language code.

```
         BALR  12,0
         USING *,12
```

Within this context, the label R12 will always reference register 12. We recognize this and use the EQU directive to allow the listing to be "**R12**" to reflect that fact. One advantage of the EQU is that the label R12 will show up in assembler cross–listings of labels used, while the value 12 will not be used. We get to see every place register R12 is used.

There are also uses in which symbolic addresses are equated. Consider this example.

```
PRINT   DC   CL133' '
P       EQU  PRINT  Each symbol references the same address
```

One can also use the location counter, denoted by "*****", to set the symbol equal to the current address that is allocated by the assembler. This example sets the symbol **RETURN** to the current address, which is not associated with an instruction.

```
RETURN   EQU   *          BRANCH TO HERE FOR NORMAL RETURN
```

The Location Counter

The location counter is denoted by the asterisk "*****". One might have code such as.

```
      SAVE   DS   CL3
      KEEP   EQU  *+5
```

Suppose the symbol **SAVE** is associated with location X'3012'. It reserves 3 bytes for storage, so the location counter is set to X'3015' after assembling the item.
The symbol **KEEP** is now associated with X'3015' + X'5' = X'301A'

LTORG

The Literal Pool contains a collection of anonymous constant definitions, which are generated by the assembler. The LTORG directive defines the start of a literal pool.

While some textbooks may imply that the LTORG directive is not necessary for use of literals, your instructor's experience is different. It appears that an explicit LTORG directive is required if the program uses literal arguments.

The classic form of the statement is as follows, where the "L" of "LTORG" is to be found in column 10 of the listing.

Generally, this statement should be placed near the end of the listing, as in the next example taken from an actual program.

```
                     240 *    LITERAL POOL
                     241 **********************************
000308               242          LTORG *
000308 00000001      243                =F'1'
000000               244          END   LAB1
```

Here, line 243 shows a literal that is inserted by the assembler. Note that lines 242 and 243 are listed at the same address (hexadecimal 000308). The directive generates no code.

PRINT
This directive controls several options that impact the appearance of the listing.

Two common variants are:

```
    PRINT ON,NOGEN,NODATA   WE USE THIS FOR NOW
    PRINT ON,GEN,NODATA     USE THIS WHEN STUDYING MACROS
```

The first operand is the listing option. It has two values: **ON** or **OFF**.

ON	Print the program listing from this point on. This is the normal setting, especially for student programs that are run only a few times.
OFF	Do not print the listing. This is used for "production programs" that have been thoroughly tested and now are used for results only.

The second operand controls the listing of macros, which are single statements that expand into multiple statements. We shall investigate them later.

The two options for this operand are **NOGEN** and **GEN**.

GEN	Print all the statements that a macro generates.
NOGEN	Suppress the generated code. This is the standard option.

Here is an two samples of the same code, which contains the user–defined macro STKPUSH, used to manage a stack. The first sample is a listing generated with the GEN option.

```
                                    100         STKPUSH HHW,H
000068 4830 C0C6            000CC   101+        LH    R3,STKCOUNT
00006C 8B30 0002            00002   102+        SLA   R3,2
000070 4120 C0CA            000D0   103+        LA    R2,THESTACK
000074 4840 C1CE            001D4   104+        LH    R4,HHW
000078 5043 2000            00000   105+        ST    R4,0(3,2)
00007C 4830 C0C6            000CC   106+        LH    R3,STKCOUNT
000080 4A30 C43A            00440   107+        AH    R3,=H'1'
000084 4030 C0C6            000CC   108+        STH   3,STKCOUNT
                                    109         STKPUSH FFW
000088 4830 C0C6            000CC   110+        LH    R3,STKCOUNT
00008C 8B30 0002            00002   111+        SLA   R3,2
000090 4120 C0CA            000D0   112+        LA    R2,THESTACK
000094 5840 C1CA            001D0   113+        L     R4,FFW
000098 5043 2000            00000   114+        ST    R4,0(3,2)
00009C 4830 C0C6            000CC   115+        LH    R3,STKCOUNT
0000A0 4A30 C43A            00440   116+        AH    R3,=H'1'
0000A4 4030 C0C6            000CC   117+        STH   3,STKCOUNT
```

The second listing is generated with the NOGEN option. Note the suppression of all of the code from the expansion of the two macros.

```
                                    100         STKPUSH HHW,H
                                    109         STKPUSH FFW
```

The NOGEN option is the standard because the student rarely wants to see the code expansion of macros, especially the system macros such as OPEN, GET, and PUT. The GEN option is mostly used in programs that explore the use of user–defined macros.

The above example is taken from a program that defines macros to operate on a stack data structure. It is important in this program to see the code generated by each expansion of the macro; it is for this reason that the GEN option has been selected.

The third operand controls printing of the hexadecimal values of constants.

`DATA`	Print the full hexadecimal value of all constants.
`NODATA`	Print only the leftmost 16 hex digits of the constants.

USING
A typical use would be found in our first lab assignment.

```
         BALR  12,0                   ESTABLISH
         USING *,12                   ADDRESSABILITY
```

The structure of this pair of instructions is entirely logical, though it may appear as quite strange. The first thing to note is that the statement "**`USING *,12`**" is a directive, so that it does not generate binary machine language code, and is not assigned an address. To the extent that the statement can have an address associated, it is the address of the next executable assembly language instruction, which commonly follows it immediately.

The **`BALR 12,0`** is an **incomplete subroutine call**. It loads the address of the next instruction (the one following the **`USING`**, since that is not an instruction) into register 12 in preparation for a **Branch and Link** that is never executed. We shall discuss subroutine calls in a later lecture, focusing then on statements in which the second argument is not 0.

The "**`USING *`**" part of the directive tells the assembler to use register **`12`** as a base register and begin displacements for addressing from the next instruction. The mechanism, base register and offset, is used by IBM in order to save space. It serves to save memory space.

Directives Associated with the Listing
Here is a list of some of the directives used to affect the appearance of the printed listing that usually was a result of the program execution process.

In our class, this listing can be seen in the Output Queue, but is never actually printed on paper. As a result, these directives are mostly curiosities.

EJECT This causes a page to be ejected before it is full. The assembler keeps a count of lines on a page and will automatically eject when a specified count (maybe 66) is reached. One can issue an early page break.

SPACE This tells the assembler to place a number of blank lines between each line of the text in the listing. Values are 1, 2, and 3; with 1 as the default.

`SPACE`		Causes normal spacing of the lines.
`SPACE`	**`1`**	Causes normal spacing of the lines
`SPACE`	**`2`**	Double spacing; one blank line after each line of text
`SPACE`	**`3`**	Triple spacing; 2 blank lines after each line of text.

TITLE This allows any descriptive title to be placed at the top of each listing page. The title is placed between two single quotes.

```
TITLE   'THIS IS A GOOD TITLE'
```

Data Definition

We now discuss the rules for defining data items to be used by an assembler program. There are two important functions to be performed by each data definition statement.

1. To allocate space that will be associated with the label.
2. Optionally to allocate an initial value to be associated with the label.

The two primary data definition statements are the DS directive and the DC directive. The DS directive allocates space for a label; the DC allocates space and assigns an initial value.

The DC and DS Declaratives

These declaratives are statements that assign storage locations.

The DC declarative assigns initialized storage space, possibly used to define constants. This should be read as "Define Initialized Storage"; assembly language does not support the idea of a constant value that cannot be changed by the program.

The DS assigns storage, but does not initialize the space.

The assembler recognizes a number of data types. There are many others. Those of importance to this course are as follows:

A	address	(an unsigned binary number used as an address)
B	binary	(using the binary digits 0 and 1)
C	character	
F	32–bit word	(a binary number, represented by decimal digits)
H	16–bit half–word	
P	packed decimal	
X	hexadecimal number	

DS (Define Storage)

The general format of the DS statement is as follows.

Name	DS	dTLn 'comments'

The name is an optional entry, but required if the program is to refer to the field by name. The standard column positions apply here.

The declarative, DS, comes next in its standard position.

The entry "dTLn" is read as follows.

d	is the optional duplication factor. If not specified, it defaults to 1.
T	is the required type specification. We shall use A, B, C, F, P, or X. note that the data actually stored at the location does not need to be of this type, but it is a good idea to restrict it to that type.
L	is an optional length of the data field in **<u>bytes</u>**.

The next field does not specify a value, but should be viewed as a comment.
This comment field might indirectly specify the length of the field, if the L
directive is not specified.

Examples of the DS Statement
Consider the following examples.

```
C1       DS  CL5    A CHARACTER FIELD OF LENGTH 5 BYTES.
                    THIS HOLDS FIVE CHARACTERS..

C2       DS 1CL5    EXACTLY THE SAME AS THE ABOVE.
                    THE DEFAULT REPETITION FACTOR IS 1.

P1       DS  PL2    A PACKED DECIMAL FIELD OF LENGTH TWO
                    BYTES.  THIS HOLDS THREE DIGITS.

P2       DS 5PL4    FIVE PACKED DECIMAL FIELDS, EACH OF
                    LENGTH 4 BYTES AND HOLDING 7 DIGITS.

F1       DS  F‘23’  ONE 32-BIT FULL WORD.  THE FIELD HAS
                    LENGTH 4 BYTES.  THE ‘23’ IS TREATED
                    AS A COMMENT; NO VALUE IS STORED.

         DS  C      AN ANONYMOUS FIELD FOR ONE CHARACTER.
                    THE LENGTH DEFAULTS TO 1.  THERE IS
                    NO NEED TO NAME EVERY DECLARATION.
```

Input Records and Fields
Data are read into records. You may impose a structure on the record by declaring it with a zero duplication factor. This allows the record to be broken into fields, each with a data item relevant to the record. Note that the sum of the field sizes must be that of the record.

Here is a declaration of an 80–byte input area that will be divided into fields.

```
CARDIN   DS  0CL80   The record has 80 bytes.
NAME     DS    CL30  The first field has the name.
YEAR     DS    CL10  The second field.
DOB      DS    CL8   The third field.
GPA      DS    CL3   The fourth field.
         DS    CL29  The last 29 chars are not used.
```

The idea is that the entire record is read into the 80–byte field CARDIN.
The data can be extracted one field at a time. Here is the COBOL equivalent.

```
01  CARDIN.
    10  NAME      PIC X(30).
    10  YEAR      PIC X(10).
    10  DOB       PIC X(08).
    10  GPA       PIC X(3).
    10  FILLER    PIC X(29).
```

More on the Zero Repetition Factor
Consider the previous example, used to define fields in a card image input.

As in all of this type of programming the input is imagined as an 80–column card.

```
CARDIN   DS 0CL80
NAME     DS CL30
YEAR     DS CL10
DOB      DS CL8
GPA      DS CL3
FILLER   DS CL29
```

Because of the zero repetition factor the line **`CARDIN DS 0CL80`** can be viewed almost as a comment. It declares that a number of statements that follow will actually allocate 80 bytes for 80 characters, and associate data fields with that input.

Here we see that 30 + 10 + 8 + 3 + 29 = 80. In this case, it is the five statements following the **`CARDIN DS 0CL80`** that actually allocate the storage space.

Remember that it is not the number of statements that is important, but the number of bytes allocated. For character data in EBCDIC form the character count is the byte count.

Another Card Definition
Suppose that we have a program that is to read a list of integers, one per card and do some processing with those integers. The same logic will apply to numbers in any format (fixed point, floating point, etc.), but the example is most easily made with positive integer data.

At this point, we cannot process a sign bit; "+" and "–" are out.

The first choice is how many digits are to be used for the positive integer.

The second choice is where to place those digits.

Here we choose to use five digits, placed in the first five columns of the card.
The appropriate declaration follows.

```
RECORDIN DS    0CL80  THE CARD HAS 80 COLUMNS
DIGITS   DS     CL5   FIVE BYTES FOR FIVE DIGITS
FILLER   DS     CL75  THE NEXT 75 COLUMNS ARE IGNORED.
```

Notice that the first statement does not allocate space. The next two statements allocate a total of 80 bytes for 80 characters.

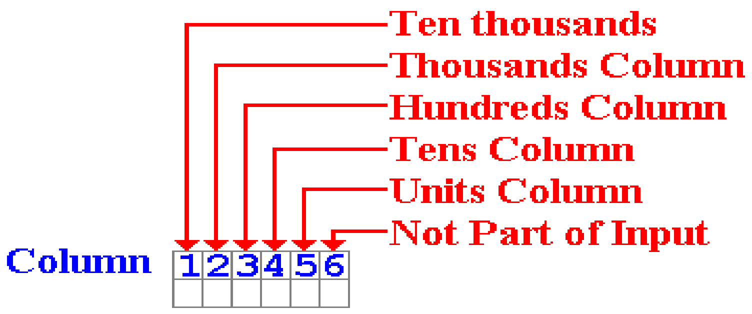
Sample Input Data Record: Reading Positive Integers
Suppose that we wanted to read a list of five–digit numbers, one number per "card".
Each digit is represented as a character, encoded in EBCDIC form.

The appropriate declaration might be written as follows.

```
RECORDIN DS    0CL80  THE CARD HAS 80 COLUMNS
DIGITS   DS     CL5   FIVE BYTES FOR FIVE DIGITS
FILLER   DS     CL75  THE NEXT 75 COLUMNS ARE IGNORED.
```

In this, the first five columns of each input line hold a character to be interpreted as a digit. The other 75 are not used. This input is **<u>not free form</u>**.

Based on the above declaration, one would characterize the first five columns as follows:



Sample Code and Data Structure

Consider the code fragment below, containing operations not yet defined. This uses the above declarations, specifically the first five columns as digits.

```
         PACK PACKIN,DIGITS     CONVERT TO PACKED DECIMAL
         AP   PACKSUM,PACKIN    ADD TO THE SUM.
```

Given the definition of **DIGITS**, the **PACK** instruction expects the input to be right justified in the first five columns. The input below will be read as "23".

```
   23      Note the three spaces before the digits.
           2 is in the tens column and 3 is in the units.
```

The following input is **not proper** for this declaration.

```
  37        "3" in column 3, "7" in column 4, column 5 blank.
```

The **PACK** instruction will process the first five columns, and result in a number that does not have the correct format. The **AP** (an addition instruction) will fail because its input does not have the correct input, and the program will terminate abnormally.

DC (Define Constant)

Remember that this declarative **defines initialized storage**, not a constant in the sense of a modern programming language. For example, one might declare an output area for a 132–column line printer as follows:

```
PRINT    DS   0CL133
CONTROL  DC   C' '         The single control character
LINEOUT  DC   CL132' '     Clear out the area for data output
```

One definitely wants to avoid random junk in the printer control area.

The data to be output should be moved to the **LINEOUT** field.

NOTE: One might have to clear out the output area after each line is printed.

Character constants are left justified. Consider the following.

```
B1      DC CL4'XYZ'   THREE CHARACTERS IN 4 BYTES
```

What is stored? **ANSWER E7 E8 E9 40 (hexadecimal)**

The above is the EBCDIC for the string **"XYZ "**

DC: Define Initialized Storage

Consider the following assembly language code and associated data declarations.

Again, this uses assembly language operators that have yet to be defined.

```
         L   R5,=F'2234'  PUT THE VALUE 2234 INTO REGISTER
                          R4.  IT IS A 32-BIT INTEGER.
         ST  R5,FW01      STORE INTO THIS LOCATION.
         More code here.
FW01     DC  F'0'         A 32-BIT FULLWORD WITH VALUE
                          INITIALIZED TO ZERO.
```

There is no problem with changing the value of this "constant", called **FW01**.

The **DC** declaration just sets aside storage space (here four bytes for a fullword) and assigns it an initial value. The program is free to change that value.

There is nothing in this assembler language that corresponds to a constant, as the term is used in a higher level language such as Java, C++, or Pascal. The only way to insure that a value initialized with a DC directive is treated as a real constant is not to write code to change it.

The assembler enforces very little, and certainly not the idea of a constant. Nevertheless, the idea of using a **DIS** directive might be too strange for some students.

More On DC (Define Constant)

The general format of the DC statement is as follows.

Name	DC	dTLn 'constant'

The name is an optional entry, but required if the program is to refer to the field by name. The standard column positions apply here.

The declarative, DC, comes next in its standard position.

The entry "dTLn" is read as follows.

d	is the optional duplication factor. If not specified, it defaults to 1.
T	is the required type specification. We shall use A, B, C, F, P, or X. Note that the data actually stored at the location does not need to be of this type, but it is a good idea to restrict it to that type.
L	is an optional length of the data field in **bytes**.

The 'constant' entry is required and is used to specify a value.
If the length attribute is omitted, the length is specified implicitly by this entry.

The Duplication Factor
The duplication factor is used in both the DC and DS statements.
Here are a few examples.

```
P1       DS 3PL4     Three 4–byte fields for packed decimal
C1       DS 1CL5     ONE 5-CHARACTER FIELD.
C2       DS CL5      ANOTHER 5-CHARACTER FIELD.
H1       DS 4H       FOUR HALF WORDS (8 BYTES)
F2       DS 5F       FIVE FULL WORDS (20 BYTES)
F3       DC 2F‘32’   TWO FULL WORDS, EACH HAS VALUE 32
                     EIGHT BYTES OF STORAGE ARE ALLOCATED,
                     WITH CONTENTS 00 00 00 20 00 00 00 20.
```

In the above example, recall that decimal 32 is hexadecimal 0x20. As a 32–bit fullword, this number is stored in four bytes represented by the eight hexadecimal digits 0x0000 0020.

```
TPL      DC 3C‘1234’ THREE COPIES OF THE CONSTANT ‘1234’
                     TWELVE BYTES OF STORAGE ARE ALLOCATED:
                     F1 F2 F3 F4 F1 F2 F3 F4 F1 F2 F3 F4
```

NOTE: The address associated with each label is the leftmost byte allocated.
For TPL, this would be the byte containing the **F1** (the first one).

In this and the next example, the EBCDIC code for the character string “1234” is “**F1 F2 F3 F4**”. The spaces are added to improve readability.

More on the Duplication Factor
Consider again the declaration of the three character strings.

```
TPL      DC 3C‘1234’ THREE COPIES OF THE CONSTANT ‘1234’
```

Here is the explicit allocation of the twelve bytes, for the twelve characters. Each byte contains the EBCDIC code for the digit represented as two hexadecimal digits.

Address	Contents
TPL	F1
TPL+1	F2
TPL+2	F3
TPL+3	F4
TPL+4	F1
TPL+5	F2
TPL+6	F3
TPL+7	F4
TPL+8	F1
TPL+9	F2
TPL+10	F3
TPL+11	F4

Some Trick Questions
Consider the following two statements.

To what value is each label initialized?

```
DATE1    DS   CL8‘DD/MM/YY’
DATE2    DC   CL8‘MM/DD/YY’
```

ANSWER:

1. **`DATE1`** is not initialized to anything. The string **`‘DD/MM/YY’`** is treated as a comment. I view this as a trick, and would prefer the assembler not to allow this.
2. **`DATE2`** is initialized as expected.

What about the following?

```
DATE3    DC   CL8 ‘MM/DD/YY’  Note the space after CL8.
```

This is an error, likely to cause a problem in the assembly process. The space after the “**`CL8`**” initiates a comment, so the label is not initialized.

As an aside, this problem will be seen any time a space is used inappropriately. The standard for the assembler is to treat the line as having four distinct fields: an optional label followed by a number of spaces, then an instruction or directive followed by some spaces, then an operand. The operand is terminated by a space and anything after that is a comment.

Hexadecimal Constants
Two hexadecimal digits can represent any of the 256 possible values that can appear in a byte or represented as an EBCDIC character. This is handy for specifying character strings that are otherwise hard to code in the assembly language.

Consider the following example.

```
      ENDLINE  DC   X’0D25’   Carriage return / line feed.
```

WARNING
Do not confuse hexadecimal constants with other formats. For example:

Format	Constant	Bytes	Hex Representation
Character	DC C’ABCD’	4	C1C2C3C4
Hexadecimal	DC X’ABCD’	2	ABCD
Character	DC C’1234’	4	F1F2F3F4
Hexadecimal	DC X’1234’	2	1234
Packed	DC P’123’	2	123C
Hexadecimal	DC X’123’	2	0123

In the last example, the three hexadecimal digits would require one and a half bytes to store. However, memory cannot be allocated in half–bytes, so the allocation is two bytes.

More Confusion: All Storage Can Be Seen as Hexadecimal
The trouble is that all data types are stored as binary strings, which can easily be represented as hexadecimal digits.

Consider the positive binary number 263. Now 263 = 256 + 4 + 2 + 1, so we have its binary representation as 01 0000 0110. The table below uses the IBM bit numbering.

Bit No.	0	1	2	3	4	5	6	7	8	9
Value	0	1	0	0	0	0	0	1	1	1
Hex	1		0				7			

Suppose that we have two data declarations

```
BINCON   DC       H‘263’
PACKCON  DC       P‘263’
```

What would be stored. Recall that each form takes two bytes, or four hexadecimal digits for storage. Recall also that a memory map shows the contents in hexadecimal.

Here is what we would see:

Hexadecimal	**Assembly Code**
`0107`	`BINCON    DC   H‘263’`
`263C`	`PACKCON   DC   P‘263’`

An Important Difference from High–Level Languages
Note that each of BINCON and PACKCON can represent perfectly a perfectly good binary number. Each can be viewed as a half–word.

The value in a location depends on the instruction used to process the bits in that location.

Consider the following code fragments, each of which defines RESULT as necessary. In each case, RESULT occupies two bytes: either a half–word or 3 packed digits.

Case 1:Packed decimal arithmetic

```
ZAP  RESULT, PACKCON      COPY PACKCON TO RESULT
AP   RESULT, PACKCON      RESULT NOW HOLDS 526C
```

As 263 + 263 = 526, the value in RESULT is now 526 (represented as 526C).

Case 2: Two’s–Complement Integer Arithmetic

```
LH  R3, PACKCON     LOAD HEXADECIMAL 263C
AH  R3, PACKCON     263C + 263C = 4C78
STH R3, RESULT      RESULT NOW HOLDS 4C78.
```

In hexadecimal: C + C = 18 and 6 + 6 = C
(12 + 12 = 24 = 16 + 8 and 6 + 6 = 12)

Note that the second case represents a mismatch of the operator type and data type.

Literals

The assembler provides for the use of **literals** as shortcuts to the DC declarative. This allows the use of a constant operand in an assembly language instruction.

The **immediate operand**, another option, will be discussed later.

The assembler processes the literal statement by:

1. Creating an object code constant,
2. Assigning the value of the literal to this constant,
3. Assigning an address to this constant for use by the assembler, and
4. Placing this object in the **literal pool**, an area of storage that is managed by the assembler.

The constants created by the literal are placed at a location in the program indicated by the **LTORG** directive.

Your instructor's experience is that the assembler cannot process any use of a literal argument if the **LTORG** directive has not been used.

Setting Up the Literal Pool

The easiest way to do this is to "uncomment" the **LTORG** declaration in the sample program and change the comment. The sample program has the following. Note the "*" in column 1 of the line containing the declaration.

```
****************************************************************
*
*    LITERAL POOL - THIS PROGRAM DOES NOT USE LITERALS.
*
****************************************************************
*         LTORG *
```

Here is the changed declaration, with a new comment that is more appropriate.

```
****************************************************************
*
*    LITERAL POOL – THE ASSEMBLER PLACES LITERALS HERE.
*
****************************************************************
          LTORG *
```

Note that the only real change is to remove the "*" from line 1 of the last statement.

More on Literals
A literal begins with an equal (=) sign, followed by a character for the type of constant to be used and then the constant value between single quotes.

Here are two examples that do the same thing.

Use of a DC:	`HEADING   DC‘INVENTORY’`
	`MVC PRINT,HEADING`

Use of a literal	`MVC PRINT,=C‘INVENTORY’`

Here is another example of a literal, here used to load a constant into a register.
We begin with the instruction itself: `L R4,=F'1'  Set R4 equal to 1.`

```
000014 5840 C302  00308    47          L     R4,=F'1'
```

Here is the structure of the literal pool, after the assembler has inserted the constant 1.

```
                          240 *    LITERAL POOL
                          241 ***************************
000308                    242          LTORG *
000308 00000001           243                =F'1'
000000                    244          END   LAB1
```

Organization of the Literal Pool
The literal pool is organized into sections in order to provide for efficient use of memory.

Consider the following plausible placement, which is not used.

```
H1        DC    X’CDEF’

C1        DC    CL3’123’ At address H1 + 4

F1        DC    F’1728’       At address H1 + 7
```

If the assembler must place F1 at an address that is a multiple of four, it would have to add a useless single byte as a “pad”.

The rules for allocation of the literal pool minimize the fragmentation of memory.
The literals are organized in four sections, which appear in the following order.

1. All literals whose length is a multiple of 8.
2. All literals whose length is a multiple of 4.
3. All literals whose length is a multiple of 2.
4. All other literals.

The implicit statement is the literal pool begins on an address that is a multiple of 8.

Declarations: High Level vs. Assembler
It is important to re–emphasize the difference between data declarations in a high level language and data declarations in assembler language.

High–Level Languages
Suppose a declaration such as the following in Java.

```
int a, b ;     // Declares each as a 32-bit integer
```

The compiler does a number of things.

1) It creates two four–byte (32 bit) storage areas and associates one of the areas with each of the labels. The labels are now called "variables".
2) It often will clear each of these storage areas to 0.
3) It will interpret each arithmetic operator on these variables as an integer operation. Thus "a + b" is an invocation to the integer addition function.

Assembler

```
A    DS H    Set aside two bytes for label A
B    DS H    and two bytes for label B.
```

It is the actual assembly operation that determines the interpretation of the data associated with each label. Note that neither is actually initialized to a value.

Note again that neither directive associates a type with the space allocated; it just allocates space appropriate for the type. In fact, standard practice in writing assembler code often calls for use of a directive to set aside storage to be used with a different type.

As an example, consider the double precision floating point type, designated by a D, which is represented by 64 bits, or 8 bytes. A typical declaration might be of the form

```
XX  DS  D    Set aside 8 bytes for the label X.
```

After label **XX** has been so declared, it may be used to store any eight–byte unit of data, even a double precision floating point number.