Data Definition, Movement, and Comparison

Topics covered include:

1. Data Definition: the DS and DC statements.
2. The use of literals as statements to define constants.
3. Data movement commands
4. Data comparison commands
5. Data conversion commands

Each of the last three topics will be covered again in a future lecture.
It never hurts to mention some things twice.

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 **bytes**.

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.

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

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

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.

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

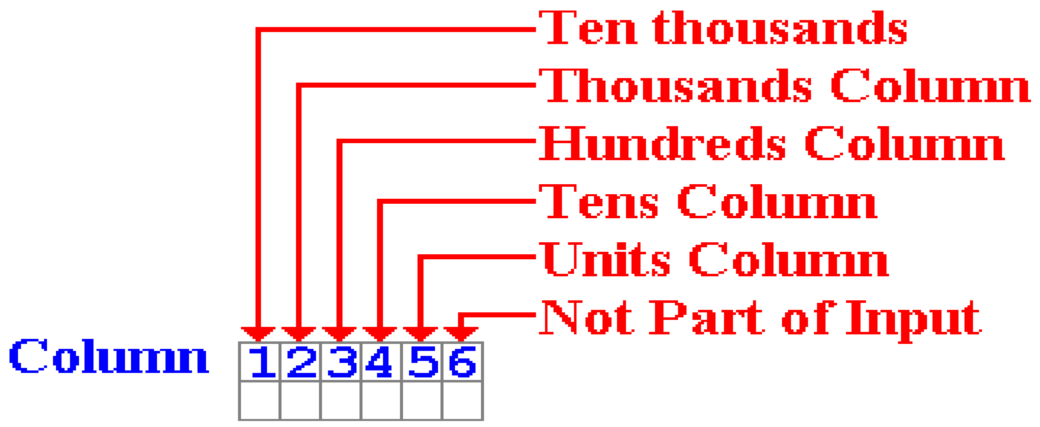
Suppose that we wanted to read a list of five–digit numbers, one number per “card”. Each digit is represented as a character, encoded in EBCDIC form.

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 **not free form**.

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 **<u>not proper</u>** 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.

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 **<u>bytes</u>**.

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 = 32
                       EIGHT BYTES OF STORAGE ARE ALLOCATED,
                       WITH CONTENTS 00 00 00 20 00 00 00 20.
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).

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.

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.

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.

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
Packed	DC P’123’	2	123C
Hexadecimal	DC X’123’	2	0123

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 standard bit numbering.

Bit No.	9	8	7	6	5	4	3	2	1	0
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’262`
`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)

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.

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.

Data Movement

The MVC instruction is designed to move character data.

As it moves a number of bytes from a source to a destination, it can
be used to move data of any type.

We shall discuss the effect of this instruction at a later time, focusing especially
on moving characters between fields of unequal length.

Packed Data

The ZAP (Zero Add Packed) instruction can be used to transfer packed decimal data.

Fullword (and Address) Data

Most commonly, moves of this type involve a general–purpose register as either
the source, the destination, or both.

```
L  6, F1           Load a register from a fullword

L  8, =A(TABLE)    Load an address
LR 8, 3            Register to register copy
ST 8, F2           Place the value into a fullword
```

Data Comparison and Branching

One important tool in programming is the ability to compare the contents of two fields and branch conditionally based on the result of these comparisons.

The comparison strategy calls for a two–step process:

1) Do an operation, such as a comparison or arithmetic operation, that sets a standard condition.
2) Then invoke a conditional branch based on one of the standard conditions.

The four standard conditions recognized by the System/360 architecture are:

00	Equal or zero
01	Low or minus
10	High or plus
11	Arithmetic overflow

These numbers are reflected in a two–bit field in the System/360 PSW, the Program Status Word.

The contents of the PSW are set by specific comparison instructions, to be discussed later.

Branch Instructions

The assembler provides two sets of conditional branch statements, one for general comparisons and another (exactly equivalent) for arithmetic comparisons.

General Conditional Branch Instructions

BE	Branch equal	BNE	Branch not equal
BL	Branch low	BNL	Branch not low
BH	Branch high	BNH	Branch not high

Arithmetic Comparison Branch Instructions

BZ	Branch zero	BNZ	Branch not zero
BM	Branch minus	BNM	Branch not minus
BP	Branch plus	BNP	Branch not plus
BO	Branch overflow	BNO	Branch not overflow

Unconditional Branch

B	Branch

Comparing Character Data

The CLC instruction is used to compare data encoded as EBCDIC, or equivalently in Zoned Data format.

Comparison is made on a character by character basis, left to right, following the sort order corresponding to the EBCDIC code.

The example in the book (page 95) is slightly misleading due to its focus on business processing of numeric account codes represented as zoned data.

```
         CLC    NEW, PREV
         BH     B20HIGH    NEW sorts higher than PREV
         BE     B21EQUAL   The two are equal
*        HERE IF NEW SORTS LESS
```

The problem here is that the comparison is treating the numbers as zoned data.

If the number of digits in the two numbers is the same, then the comparison will be the same as for an arithmetic comparison.

Numeric Comparisons

The comparison discussed above treats the fields as unsigned sequences of characters.

The next two make algebraic comparisons; negative always is less than positive.

Packed Data

The CP instruction is used to compare packed data.

Binary Data

There are three binary comparison instructions.

C	compare a register to a 32–bit fullword,
CH	compare a register (as a halfword) to a 16–bit halfword
CR	compare two registers.

Data Conversions

Recalling that numeric data has the same representation as either character data or zoned data, we have four conversion instructions.

Character to Packed Format

Use the PACK instruction, as in PACK PACKFLD, ZONEFLD

Packed to Binary Format

Use the CVB instruction, as in CVB 6, DBLWORD
This is an RX type instruction; the destination must be a register.

Binary to Packed Format

Use the CVD instruction, as in CVD 8, DBLWORD
This is also an RX type instruction; the source must be a register.

Packed to Character (Zoned) Format

Use the UNPK instruction, as in UNPK ZONEFLD, PACKFLD