CHAPTER 8: PACKING IT IN

In which we learn to add, subtract, multiply, and divide decimal data.

Creating Packed Data

Packed decimal data fields are created by specifying a "P" for the type in a DC or DS declarative. In this format, when assembled, each byte in the field contains 2 decimal digits except the right-most byte in the field which contains a decimal digit followed by a sign. The valid positive signs are the hexadecimal characters A, C, E, and F. The valid negative signs are the hexadecimal characters B and D. The "preferred" signs, those which are generated by the machine or chosen by the assembler, are positive "C" and negative "D". It is standard practice to use "C" and "D" but be aware that the other signs are recognized as well. Consider the two fields defined below,

```
AMOUNT1   DC     PL3'+20'
AMOUNT2   DC     PL2'-34'
```

In the example above, the assembler generates x'00020C' for AMOUNT1 and x'034D' for AMOUNT2.

A packed decimal field can be created using either of the following formats,

```
name      DC     dPLn'constant'
```

or

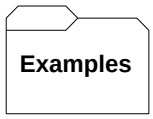
```
name      DS     dPLn
```

- 'name' is an optional field name
- 'd' is a duplication factor that creates consecutive copies of the field (default = 1)
- 'P' represents the packed decimal data type
- 'L' denotes an optional length (default = 1 byte)
- 'n' is the number of bytes in the field
- 'constant' is the initial value of the field in decimal format. Optionally, a decimal point and a + or - sign can be coded but the decimal point is not represented internally when assembled.

It is important to note that if a name for a field is specified, it will represent the address of the first byte of the field. Additionally, the name of the field will be associated with a length attribute which equals the number of bytes in the field. If the "Ln" construction is omitted in a DC, the length will be just large enough to accommodate the digits specified in the constant (along with a padded 0 on the left if the number of digits in the constant is even). Packed fields always contain an odd number of decimal digits because a sign is always stored. In a DS declarative, "Ln" must be coded. The length of a packed field defined using a DC or DS is limited to a maximum of 16 bytes. This limits the size of decimal fields to 31 digits.

During assembly, fields that were defined consecutively in the source with DC's or DS's are assigned consecutive storage locations in the program according to the value of the location counter which is maintained by the assembler. For example, in the fields below, if PKD1 is associated with address x'1000', then PKD2 is located at x'1008' and PKD3 is located at x'1011'.

```
LOCATION

1000    PKD1      DS    PL8
1008    PKD2      DC    PL9'20'
1011    PKD3      DS    PL3
```



Some Typical DS's and DC's:

```
Q          DS    PL8          AN 8-BYTE FIELD RESERVED FOR PACKED DATA
R          DS    PL16         MAX SIZE OF PACKED FIELD (31 DIGITS + SIGN)
S          DS    PL1          MIN SIZE OF PACKED FIELD (1 DIGIT + SIGN)
T          DS   3PL4          THREE CONSECUTIVE PACKED FIELDS,
                                T REFERS TO FIRST FIELD, OTHERS UNNAMED
U          DC    P'123'       U = X'123C' FIELD IS ASSUMED POSITIVE
V          DC    P'-123'      V = X'123D'
W          DC    P'12'        W = X'012C' 0 PADDED ON LEFT TO FILL A BYTE
X          DC    PL4'12'      X = X'0000012C' CONSTANT IS SHORTER THAN
                                THE FIELD LENGTH - 0 PADDING ON LEFT
Y          DC    PL2'12345'   Y = X'345C' CONSTANT TOO BIG, TRUNCATION ON
                                THE LEFT
Z          DC    P'123.45'    Z = X'12345C' DECIMAL PT TREATED AS COMMENT
```

Tips

1. Be sure to use DC instead of DS when providing a constant. Consider the example below,

   ```
   AMOUNT DS PL4'1234'
   ```

   The assembler will not complain about you providing a constant on a DS statement. It will, however, ignore the constant and treat it as a comment. The field remains uninitialized.

   Be sure to make any packed fields you define large enough to hold any possible values that might be developed in them. You cannot err by overestimating the size of a packed field. On the other hand, it is disastrous to underestimate the size of a packed field - this will result in truncation of digits in arithmetic operations (decimal overflow exception).

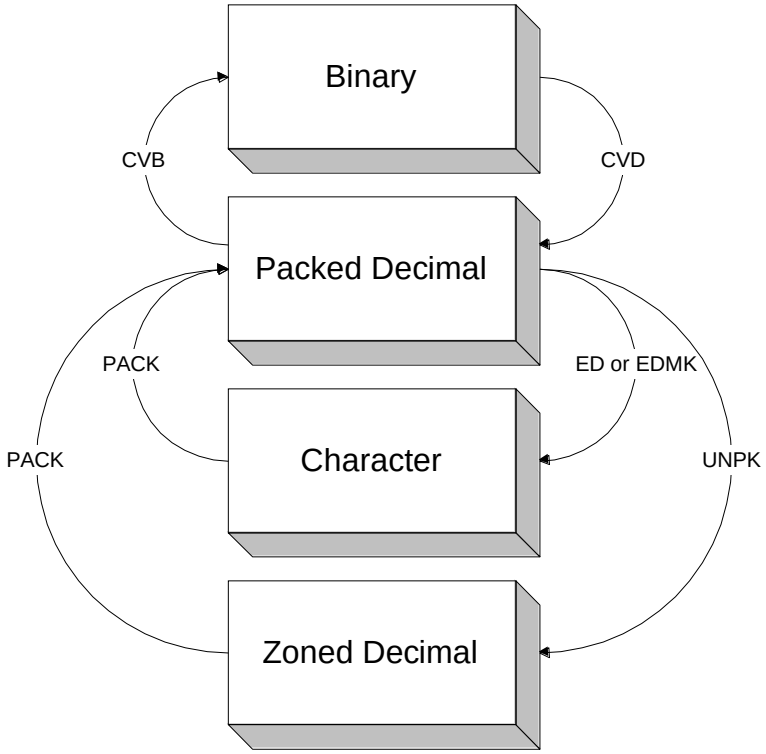
2. The specification of "P" as the type of data does not ensure that the data is packed. In fact, many kinds of data might be stored in a packed field.

3. It's a good idea to provide explicit lengths for packed fields with constants, rather than letting the assembler decide.

   ```
   AMOUNT     DC    PL4'82.57'
   ```

Types of Data and Their Relationships

The diagram below lists four types of data and the instructions used when converting from one type to another. Packed decimal data is the subject of this chapter and I want to examine how to convert between packed decimal and zoned decimal, and between packed decimal and character data. In the next chapter we will consider conversions between binary and packed decimal.



Converting from Zoned and Character to Packed

Zoned decimal data and character data that contains only digits have closely related data formats. For example, consider the definitions below.

```
CFIELD   DC    C'12345'    CFIELD = X'F1F2F3F4F5'
ZFIELD   DC    Z'12345'    ZFIELD = X'F1F2F3F4C5'
```

In fact, the only difference above occurs in the zone portion of the rightmost byte. The zoned field contains a "C" while the character field contains an "F". In a zoned format both "C" and "F" represent a positive sign (+) and so CFIELD AND ZFIELD are equivalent as zoned decimal fields.

As a result, the **PACK** instruction can be used to convert both fields to a packed format:

```
CFIELD   DC    C'12345'    CFIELD = X'F1F2F3F4F5'
ZFIELD   DC    Z'12345'    ZFIELD = X'F1F2F3F4C5'
CPACKED  DS    PL3
ZPACKED  DS    PL4
DBLWD    DS    D
         ...
         PACK  CPACKED,CFIELD      CPACKED = X'12345F'
         PACK  ZPACKED,ZFIELD      ZPACKED = X'0012345C'
         PACK  DBLWD,CFIELD        DBLWD = X'000000000012345F'
```

In converting the character field to packed decimal in the first example above, we decided to use a 3-byte field to hold the result. This decision is somewhat arbitrary. Since the character field contained 5 digits, it takes at least 3 bytes to hold the result in a packed format (two packed digits per byte plus the sign). A smaller choice would have caused truncation of digits on the left. A larger field could have been chosen which would have resulted in the field being padded with zeros on the left. This in fact, happened when we packed the zoned field into a 4-byte field and again when we packed CFIELD into a doubleword. If our goal is to convert a packed field to binary, we will need to first put the field into a "staging area" that is a doubleword. This is discussed in the next topic.

Converting from Packed Decimal to Character

Preparing a packed field for printing involves the creation of an appropriate edit word and the use of the **ED** or **EDMK** instruction. This is an error-prone process for many beginning assembler programmers and many headaches can be avoided by paying attention to a few details:

1) Start with the packed field and count the number of packed digits it contains.

2) Create an edit word for the packed field. Count the number of X'20's and X'21's that appear in the edit word. The count in step 2 **must equal** the count in step 1. If you make a mistake with this, the results are highly unpredictable!

3) Count the number of **bytes** in the edit word. Count every byte including the fill character. Now create an output field that is exactly the size of the edit word. Again, a mistake here is fatal.

4) At execution time, move the edit word to the output field and edit the data.

For example, assume we want to edit the packed field below into a dollars and cents format.

```
PKFIELD  DC    PL4'543210'   PKFIELD = X'0543210C'
```

Examining PKFIELD we see that it contains 7 packed digits (2 digits x 4 bytes minus 1 for the sign). This means that the edit word we create must contain seven X'20's or X'21's. The following edit word will work. Edit words need a fill character (X'40') at the beginning to pad leading zeroes, and we added a X'4B' for the decimal point.

```
EDWD DC X'4020206B2020214B2020'
```

Counting the bytes in EDWD we see it contains 10. Next, we create an output field capable of containing the edit word.

```
PKOUT  DS  CL10
```

Now we are ready to edit the data. We move the edit word to the output area and edit on top of it.

```
         MVC   PKOUT,EDWD
         ED    PKOUT,PKFIELD
```

The hexadecimal contents of PKOUT after executing the code above is X'4040F5F46BF3F24BF1F0' and would appear in a printed report as "5,432.10".

Converting from Packed Decimal to Zoned Decimal

The conversion to zoned decimal from packed decimal is a simple and straightforward use of the **UNPK** instruction. Be sure to provide a field big enough to hold the result. Consider the following conversion.

```
         UNPK ZFIELD,PKFIELD
         ...
PKFIELD  DC   P'112233'    PKFIELD = X'0112233C'
ZFIELD   DS   ZL7
```

After execution, ZFIELD contains X'F0F1F1F2F2F3C3'.

Programming With Packed Data

Packed decimal is a convenient format for doing many arithmetic calculations in assembly language for several reasons:

1) All computations occur in integer arithmetic (no decimals, 5/2 = 2, etc.),

2) Packed decimal fields are easy to read in a storage dump,

3) Computations occur in base 10.

The main disadvantage to packed decimal arithmetic is that decimal points are not stored internally. This means a programmer must keep up with decimals, and shift and round to achieve the desired accuracy and precision. As a result, all computations are done with integers. Precision is achieved by shifting left or right before and after many computations.

In this topic we discuss commonly used packed decimal computations through examples. Later we will consider arithmetic operations with decimal points.

Copying Packed Fields

When copying a packed decimal field, be sure to use the Zero and Add Packed instruction, **ZAP**. By using **ZAP**, you are assured that the target field will be properly initialized. It is a beginner's error to use **MVC** for copying packed decimal values. This can lead to a problem which is illustrated in the following example.

```
          MVC   AFIELD,PKFIELD    AFIELD = X'038CFF'
          ZAP   AFIELD,PKFIELD    AFIELD = X'00038C'
          ...
AFIELD    DS    PL3
PKFIELD   DC    PL2'38'           PKFIELD = X'038C'
          DC    X'FF'
```

In the first **MVC** instruction, we are copying a 2-byte packed field, PKFIELD, to a 3-byte field, AFIELD. Since **MVC** has an SS1 format, the length of AFIELD is used to determine that 3 bytes will be "moved" by this instruction. The effect of this instruction is to copy the 2 bytes in PKFIELD and the byte which follows PKFIELD as well. As a result, AFIELD does not contain a packed value. On the other hand, the second instruction, **ZAP**, initializes AFIELD with a packed decimal zero just before adding the packed value of PKFIELD. This produces the correct packed decimal value X'00038C'. This type of error with the **MVC** instruction occurs each time the fields have different sizes.

Care must be taken even when using a **ZAP** to copy a packed field. If the target field is too small to hold the result, high order truncation of digits can occur, causing an overflow. Consider the following example involving AFIELD defined above.

```
ZAP  AFIELD,=P'123456789'  AFIELD = '56789C'
```

After executing the instruction above, the high-order digits of the packed decimal literal have been truncated. This may or may not cause the program to abend, depending on the decimal-overflow mask. (See **SPM**.) In the case where the program continues execution, the programmer is not immediately aware that an error has occurred.

One side effect of executing a ZAP is that the condition code is set to indicate how the target field compares to zero. The condition code can be tested with the branch on condition instruction using the extended mnemonics. Here is an example,

```
          ZAP   FIELD1,FIELD1     SET THE CONDITION CODE
          JZ    WASZERO           JUMP IF ZERO
```

Adding Packed Decimal Fields

Next, we consider the problem of adding several packed decimal fields to compute a total. In doing this, we must estimate the size of the sum and define a packed decimal work field that will contain it. The first field that will participate in the sum can be **ZAP**ed into the work field. All other fields that contribute to the sum will be

added using the **AP** instruction. The following example computes the sum of 3 packed decimal fields.

```
            ZAP   SUM,FIELD1
            AP    SUM,FIELD2
            AP    SUM,FIELD3
            ...
FIELD1      DC    PL3
FIELD2      DC    PL3
FIELD3      DC    PL3
SUM         DS    PL4
```

Notice that SUM was uninitialized but was zeroed out by the ZAP operation prior to the addition of FIELD1. The size of SUM is somewhat arbitrary and could vary based on our knowledge of the data. In the code above we have avoided the cardinal error of choosing a field that is too small to hold the result - a packed length 4 field will hold the sum of any 3 packed length 3 fields.

Subtracting Packed Decimal Fields

The comments above about adding several packed fields also apply when subtracting them. Use the **SP** instruction to perform the subtraction. The main error to avoid is not providing a field large enough to hold the result. The code below will compute the difference of FIELDA AND FIELDB.

```
            ZAP   RESULT,FIELDA
            SP    RESULT,FIELDB
...
FIELDA    DS    PL3
FIELDB    DS    PL3
RESULT    DS    PL4
```

The SP instruction is also useful for zeroing out a packed decimal field. Subtracting a field from itself will accomplish this result.

```
            SP DIFFER,DIFFER      DIFFER = 0
```

Comparing Packed Decimal Fields

It is often necessary to write conditional logic based on how two packed decimal fields compare. Packed decimal fields can be compared using the **CP** instruction. This instruction has the effect of setting the condition code based on an arithmetic comparison of the two fields. The compare operation is followed by one or more branch instructions that test the condition code and branch accordingly. Consider the following example which leaves the "larger" of two packed fields in a field called "BIGGER". First FIELDA is copied to BIGGER, then the fields are compared. A branch instruction, **BNH**, tests the condition code and a branch occurs to the label "THERE" if the condition code is "**N**ot **H**igh". In other words, a branch occurs if FIELDB is equal or less than FIELDA. On the other hand, if FIELDB is larger, the branch is not taken, and execution continues with the **ZAP** which copies FIELDB over the previous value in BIGGER.

```
          ZAP  BIGGER,FIELDA  ASSUME FIELDA >= FIELDB
          CP   FIELDB,FIELDB  FIELDB > FIELDA?
          BNH  THERE          BRANCH IF EQUAL OR LOW
          ZAP  BIGGER,FIELDB  CHANGE TO THE LARGER VALUE
THERE     DS   0H
```

It is a beginner's mistake to compare packed fields with the **CLC** instruction. Consider that X'5C' and X'5A' are equal as packed data but different when compared logically with CLC. The compare logical character instruction was not designed to accommodate packed decimal data. Comparing packed fields arithmetically requires that the lengths of both operands must be considered, as well as their signs. CLC determines the number of bytes to compare from the length associated with operand 1. The following code illustrates some of the problems that can occur.

```
          CP    AFIELD,BFIELD    CONDITION CODE = EQUAL
          CLC   AFIELD,BFIELD    CONDITION CODE = HIG
          CLC   SHORTNO,LONGNO   CONDITION CODE = HIGH
          ...
AFIELD    DC    X'12345C'        AFIELD = +12345
BFIELD    DC    X'12345A'        BFIELD = +12345
SHORTNO   DC    X'123C'
LONGNO    DC    X'0000123C'
```

Using **CP** in the first line, the fields are properly compared as equal packed decimal fields. (Remember that C and A are valid plus signs for packed decimal data.) The first **CLC** instruction sets the condition code to "high" when it compares the third bytes of AFIELD and BFIELD. As character data, X'5C' is higher than X'5A'. The second **CLC** illustrates another problem with using **CLC**. In this case, the condition code is set to high when comparing the first bytes as character data. In fact, the fields are equal when treated as packed decimal fields of different lengths.

Multiplying Packed Decimal Fields

The **MP** mnemonic is used for multiplying packed decimal fields. This instruction contains two operands which are multiplied; the product is copied to the first operand, destroying the original contents. The following code illustrates how to multiply two fields together.

```
          ZAP   PRODUCT,FIELD1
          MP    PRODUCT,FIELD2
          ...
FIELD1    DS    PL5
FIELD2    DS    PL3
PRODUCT   DS    PL8
```

When multiplying two fields, in this case FIELD1 and FIELD2, you must plan for a field size that is large enough to hold the product. The rule of thumb is that the length of the product field should be at least as large as the size of the multiplier length plus the multiplicand length. In the example above we compute the product length to be 5 + 3 = 8. The first step is to copy the multiplicand to the work field with the **ZAP** instruction. The operation is then completed by executing the **MP** instruction.

While Operand 1 (containing the multiplicand) can be as large as 16 bytes, Operand 2 (containing the multiplier) is limited to a maximum of 8 bytes.

The **MP** instruction will cause your program to abend if there are not enough leading 0's in the multiplicand prior to multiplication. The rule is that there must be at least as many *bytes* of leading 0's in the multiplicand as there are *bytes* in the multiplier. Consider the following example,

```
          MP    PRODUCT,FIELDB      ABEND!
          ...
PRODUCT   DC    X'00001234567C'
FIELDB    DC    X'00887C'
```

The multiply instruction above causes an interrupt and the program abends because the multiplicand contains only 2 bytes of leading 0's, while the multiplier is 3 bytes in length.

Dividing Packed Decimal Fields

Use the **DP** mnemonic for the division of packed decimal fields. Initially, operand 1 is initialized with the dividend and the divisor occupies operand 2. After the divide operation, Operand 1 contains the quotient, followed immediately by the remainder. Here is an example division which computes X / Y.

```
          ZAP   WORK,X   INITIALIZE THE DIVIDENT
          DP    WORK,Y   Y IS THE DIVISOR
          ...
WORK      DS   0CL8      GROUP FIELD
QUOT      DS    PL5      QUOTIENT OF X / Y
REM       DS    PL3      REMAINDER OF X / Y
X         DS    PL5
Y         DS    PL3
```

You must plan the size of each work area when dividing. In the example above, we are dividing a packed length 5 field by a packed length 3 field. The work area in which the division will occur must be large enough to contain a quotient and a remainder. How large could the quotient become? Since we are performing integer arithmetic, the quotient could be the same size as the dividend (consider division by 1). How large could the remainder become? The largest remainder is always one less than the divisor, but the field size of the remainder might be just as large as the divisor. Because of these considerations, the work area size should be at least as large as the size of the dividend plus the size of the divisor. In the code above, we made WORK eight bytes since the dividend was 5 bytes and the divisor was 3.

The first step was to **ZAP** the work area with the dividend, and then divide by Y. Suppose X initially contains X'000012356C' and Y contains X'00100C'. After the division, WORK will contain X'000000123C00056C'. Notice that WORK is no longer packed but contains two packed fields. It is a common error to reference the work area as a packed field after the division. This is a mistake which causes the program to abend.

Using the definition of WORK below, the following division would produce an error, if QUOT or REM were referenced as packed decimal fields.

```
WORK     DS   0CL8   GROUP FIELD
QUOT     DS    PL5   QUOTIENT OF X / Y
REM      DS    PL3   REMAINDER OF X / Y
         ...
         ZAP   WORK,=P'123456'
         DP    WORK,=PL2'100'
```

The problem arises because **the remainder's size is completely determined by the divisor's size**. Since we divided by a 2-byte field, the remainder will occupy 2 bytes of WORK and the quotient fills the other 6 bytes. After the division, WORK contains X'00000001234C056C', but the field definitions of QUOT and REM do not match these results.

Shifting Packed Decimal Fields

Since decimal points are not stored internally for packed decimal fields, and since packed decimal arithmetic is integer arithmetic, it is necessary for an assembler programmer to shift fields left and right to obtain the precision required for most calculations. This is accomplished with the shift and round pack instruction which has mnemonic **SRP**. (Some shifts can be completed using the **MVO** instruction, but **SRP** is easier to use and offers more flexibility.) Using **SRP**, a packed decimal field can be shifted left or right while leaving the sign digit fixed. For right shifts, digits are lost on the right and 0's fill in for digits which are shifted out on the left. For left shifts, leading 0's are lost on the left and 0's fill in for digits shifted out on the right.

The instruction has three operands: Operand 1 is the field that will be shifted, Operand 2 is the shift factor, and Operand 3 is a rounding factor for right shifts. The shift factor is a 6-bit 2's complement integer that we will represent as a decimal integer between 1 and 31 for left shifts, and as 64 - n for right shifts of n digits. Operand 3, the rounding factor, is an integer from 0 to 9 that is added to the leftmost digit which is shifted out during a right shift. Any carry is propagated through the rest of Operand 1. Consider the following example.

```
         SRP   P,3,5       BEFORE: P = X'000000123C'
                           AFTER:  P = X'000123000C'

         SRP   Q,64-3,5    BEFORE: Q = X'0009876C'
                           AFTER:  Q = X'0000010C'

P        DC    PL5'123             P = X'000000123C'
Q        DC    PL4'9876'           Q = X'0009876C'
```

In the first **SRP**, the shift factor of 3 indicates a left shift by 3 digits. Three zero digits are lost on the left and 3 zero digits are shifted in on the right. This shift is logically equivalent to multiplying by 1000. In the second **SRP**, the shift factor of 64 - 3 indicates a right shift by 3 digits. The 8, 7, and 6 are shifted off. Before shifting off the 8 which is the leftmost digit, the rounding factor of 5 is added to the contents of P. This addition causes a carry and creates the number 103 which is shifted one final digit right, leaving a value of 10 in P.

Shifting is commonly used when working with integers that contain decimal points. Consider the problem of multiplying S and T and leaving a product that contains 1 digit to the right of the decimal point. Remember that the machine does not store decimal points internally for packed decimal fields. Here's an example:

```
          ZAP   PRODUCT,S         INITIALIZE MULTIPLICAND
          MP    PRODUCT,T         2 DIGITS TO RIGHT OF DEC PT
          SRP   PRODUCT,64-1,0    REMOVE ONE DIGIT, NO ROUNDING
          ...
S         DC    PL3'1234.5'       S = X'12345C' NO DEC PT STORED
T         DC    PL3'10.0'         T = X'100C'   NO DEC PT STORED
PRODUCT   DS    PL6
```

First the multiplicand is ZAPed into a 6-byte field called `PRODUCT` which is large enough to hold the product of S and T. The multiplication leaves `PRODUCT` with 2 digits to the right of the decimal point (`PRODUCT = X'00001234500C'`). The **SRP** shifts out the rightmost digit leaving `PRODUCT = X'00000123450C'`. This result could be edited using **ED** or **EDMK** and the decimal point could be inserted for printed output.

Arithmetic on packed decimal fields that "contain" decimal points requires some careful thought on the programmer's part. Consider dividing 123.4 by 2.1 using integer arithmetic. Assume that after the division we would like the quotient to contain 1 decimal digit to the right of the decimal point.

We illustrate two possible divisions below.

```
      58.                 58.7
    -----               -----
2.1 123.4           2.1 123.40
    105                 105
    ---                 ---
    184                 184
    168                 168
    ---                 ---
     16                 160
                        147
                        ---
                         13
```

Keep in mind that decimal points are not stored internally. The first division illustrates dividing 2.1 into 123.4. Since we are working with integers, this is equivalent to dividing 21 into 1234. The result is 58 and contains no decimal point. This will not give us the precision we demand in the quotient.

In the second division, by shifting the dividend to the left by one digit (bringing

in a 0 on the right), we are effectively dividing 21 into 12340, and producing a quotient of 587 which could be edited to 58.7 for printing.

The code below illustrates how the above division might appear in assembly language.

```
          ZAP   WORK,M         M IS THE DIVIDEND
          SRP   WORK,1,0       M NEEDS MORE PRECISION
          DP    WORK,N         N IS THE DIVISION
          MVC   QUOTOUT,EDWD   COPY EDWD TO OUPUT AREA
          ED    QOUTOUT,QUOT   PREPARE QUOT FOR PRINTING
          ...
M         DC    PL4'123.4      M = X'0001234C'
N         DC    PL2'2.1'       N = X'021C'
WORK      DS   0PL6            WORK FIELD FOR DIVISION
QUOT      DS    PL4            QUOTIENT
REM       DS    PL2            REMAINDER
EDWD      DC    X'402020202021204B2060'
          ...
QUOTOUT   DS    CL10
```

The work area field was designed as 6 bytes since the dividend was 4 bytes and the divisor was 2 bytes. The dividend was moved to the work area and then shifted left for more precision. After the division, QUOT has the answer we would like to print. An edit word is created which matches the 4-byte packed field QUOT and moved to an output area. QUOT is then edited into the output area. (See **ED** for details on editing.)

Next, we consider the problem of generating an answer that is **rounded** to a specified precision. Suppose we are going to divide 1234.56 by 2.1, and we would like to compute the quotient rounded to 2 decimal places to the right of the decimal point. If we simply divide, the quotient will have 1 digit to the right of the decimal point. To finish with a quotient that has 2 digits rounded, we must generate three digits to the right of the decimal point before we divide. We can achieve this precision by shifting to the left by two digits before dividing. The following code illustrates this idea.

```
          ZAP   WORK,X         PREPARE THE DIVIDEND
          SRP   WORK,2,0       SHIFT IN TWO 0'S ON THE RIGHT
          DP    WORK,Y         Y IS THE DIVISOR
          SRP   QUOT,64-1,5    SHIFT RIGHT BY 1 AND ROUND
          ...
X         DC    PL4'1234.56'   X = X'0123456C'
Y         DC    PL2'2.1'       Y = X'021C'
WORK      DS   0PL7            WORK AREA FOR DIVISION
QUOT      DS    PL5            QUOTIENT
REM       DS    PL2            REMAINDER
```

Why was WORK created as a 7-byte field when X contained 4 bytes and Y contained 2 bytes? The reason is that after moving X to WORK, we shifted it to the left by 2 digits, effectively making it a 5-byte field. Making WORK a 7-byte field insures we have enough room in the work area for the division. Since we divided by a 2-byte field, the remainder has 2 bytes, and the rest of the work area is the quotient.

As a final example of handling decimal points, consider the problem of computing M times N and dividing the result by P. We would like the final answer to

have 2 decimals to the right of the decimal point, rounded. The declarations of M, N, and P are listed below with the comments indicating the precision in each field.

```
M         DS    PL4       99999.99
N         DS    PL3        9999.9
P         DS    PL2          99.9
```

If we simply multiply M and N, the product will have 3 decimal places to the right of the decimal point. Dividing by P would reduce the number to 2. To finish with an answer that has 2 decimals rounded, we need 3 digits before shifting. That means the product must be shifted to the left by 1 digit before the division. The following code could be used.

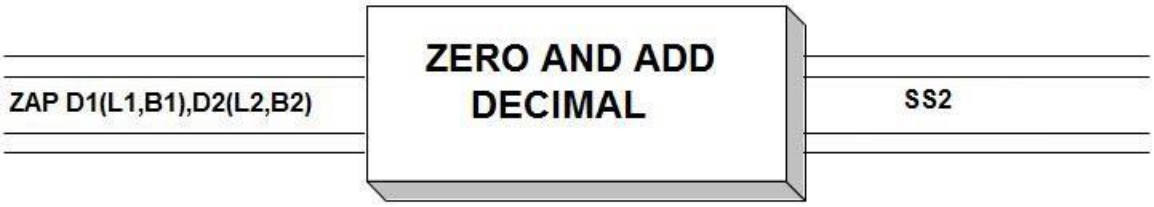
```
            ZAP   WORK,M        M IS THE MULTIPLICAND
            MP    WORK,N        COMPUTE THE PRODUCT
            SRP   WORK,1,0      SHIFT IN A ZERO ON THE RIGHT
            DP    WORK,P        QUOTIENT WILL HAVE 3 DECIMAL PLACES
            SRP   QUOT,64-1,5   ROUND QUOT BACK TO 2 DIGITS
            ...
WORK        DS   0CL10
QUOT        DS    PL8
REM         DS    PL2
```

Again, shifting the product left means the work area needs to be adjusted by one byte.

A Basic Instruction Set for Packed Decimal Data Processing

The following operations represent a working set of packed decimal instructions:

ZAP – Zero and Add Decimal
AP – Add Decimal
SP – Subtract Decimal
MP – Multiply Decimal
DP – Divide Decimal
SRP – Shift and Round Decimal
CP – Compare Decimal
PACK – Pack Decimal
UNPK – Unpack Decimal
ED – Edit
TP – Test Decimal



Op Code	L_1L_2	B_1D_1	D_1D_1	B_2D_2	D_2D_2

ZAP is an SS_2 instruction which is used to copy packed decimal fields between storage locations in memory. The copying preserves the arithmetic quality of the sending field by first zeroing the target field, and then adding the sending field to it. The initial contents of Operand 1 don't affect the operation. Operand 2 must contain a packed field before the operation occurs, otherwise an S0C7 abend will occur. The operands are limited to a maximum length of 16 bytes and may have different sizes since this is a SS_2 instruction.

A decimal overflow (condition code = 3) can occur if a significant digit is lost from the source. A decimal overflow may or may not cause a program interruption (abend). This depends on the setting of a bit in the PSW (See SPM). Otherwise, the condition code is set to indicate whether the result was zero (condition code = 0), negative (condition code = 1), or positive (condition code = 2). You can test these conditions with BZ or BNZ, BM or BNM, and BP or BNP.

Consider the following AP example.

```
         ZAP    APK,BPK
         BP     APOSITIV     BRANCH IF APK IS POSITIVE
         ...
APOSITIV DS    0H
```

Assume the variables are initially in the following states,

```
APK      DC    PL4'34'  =   X'0000034C'
BPK      DC    PL3'22   =   X'00022C'
```

After the ZAP instruction has executed, the variables have the following values.

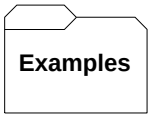
```
APK = X'0000022C'
BPK = X'00022C'
```

The contents of APK were zeroed in a packed decimal format, and the contents of BPK were added and the result stored in APK. The contents of BPK were unaffected by the ZAP operation. After the branch, execution would resume at APOSITIV since the condition code was set to positive.

On the other hand, consider the following example,

```
         ZAP   APK,APK
         BP    THERE
         ...
APK      DC   PL2'999'   = X'999D'
```

After the ZAP instruction has executed, the condition code is set to 1, indicating low or negative. APK is unchanged. The branch to THERE is not taken.



Some unrelated ZAP's:

```
A        DC    P'12345'     = X'12345C'
B        DC    P'-32'       = X'032D'
C        DC    Z'11'        = X'F1C1'

RESULTS:
          ZAP  A,=P'20'     A = X'00020C'   C.C. = HIGH/POSITIVE
          ZAP  B,=P'-20'    B = X'0020D'    C.C. = LOW/MINUS
          ZAP  B,=P'12345'  B = X'345C'     C.C. = OVERFLOW
          ZAP  A,A          A = X'12345C'   C.C. = HIGH/POSITIVE
          ZAP  B,B          B = X'032D'     C.C. = LOW/MINUS
          ZAP  B,=X'0D'     B = X'000C'     C.C. = ZERO(SIGN CHANGE)
          ZAP  A,C          DATA EXCEPTION – C NOT PACKED
```

Tips

1. Use ZAP to copy packed data. Don't use MVC to transfer packed decimal fields between storage locations.
2. ZAP a field to itself if you want to set the condition code based on the contents of the field.

Op Code	L_1L_2	B_1D_1	D_1D_1	B_2D_2	D_2D_2

AP is an SS_2 instruction which is used to add packed decimal fields. Operand 1 is a field in storage which should contain a packed decimal number. The resulting sum develops in this field. The contents of Operand 2, another packed decimal field in storage, is added to the contents of Operand 1 to produce the sum which is stored in Operand 1. The operands are limited to a maximum length of 16 bytes and may have different sizes since this is a SS_2 instruction.

A decimal overflow (condition code = 3) can occur if the generated sum loses a significant digit when it is placed in the target field. A decimal overflow may or may not cause a program interruption (abend). This depends on the setting of a bit in the PSW (See SPM). Otherwise, the condition code is set to indicate whether the result was zero (condition code = 0), negative (condition code = 1), or positive (condition code = 2). You can test these conditions with BZ or BNZ, BM or BNM, and BP or BNP.

Consider the following AP example.

```
         AP    APK,BPK
         BP    APOSITIV   BRANCH IF POSITIVE
         ...
APOSITIV DS    0H
```

Assume the variables are initially in the following states,

```
APK      DC    PL4'34  = X'0000034C'
BPK      DC    PL3'22' = X'00022C'
```

After the AP instruction has executed, the variables have the following values.

```
APK = X'0000056C'
BPK = X'00022C'
```

The contents of BPK and APK were added, and the result stored in APK. BPK was unaffected by the add operation. After the branch, execution would resume at APOSITIV since the condition code was set to positive.

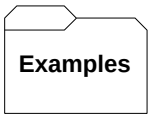
On the other hand, consider the following example,

```
          AP    APK,BPK
          ...
APK       DC    PL2'999'    =  X'999C'
BPK       DC    PL2'3'      =  X'003C'
```

After the AP instruction has executed, the variables have the following values.

```
APK = X'002C'
BPK = X'003C'
```

Notice that an overflow occurred in APK with the loss of a significant digit since the APK field was too short to hold the resulting sum.



Some unrelated AP's:

```
A         DC    P'12345'  = X'12345C'
B         DC    P'-32'    = X'032D'
C         DC    Z'11'     = X'F1C1'

RESULTS:

          AP    A,=P'20'     A = X'12365C'   C.C. = HIGH/POSITIVE
          AP    B,=P'20'     B = X'012D'     C.C. = LOW/NEGATIVE
          AP    B,=P'999'    B = X'967C'     C.C. = HIGH/POSITIVE
          AP    A,=P'-100'   A = X'12245C'   C.C. = HIGH/POSITIVE
          AP    A,B          A = X'12313C'   C.C. = HIGH/POSITIVE
          AP    B,B          B = X'064D'     C.C. = LOW/NEGATIVE
          AP    B,=P'32'     B = X'000C'     C.C. = EQUAL/ZERO
          AP    B,A          B = X'313C'     C.C. = OVERFLOW
          AP    A,C                          S0C7 EXCEPTION - C NOT PACKED
```

Tips

1. Become familiar with your data. The best way to prevent overflows is to plan the size of your fields based on the data at hand. There is a rule of thumb that you can follow for additions: If you are adding two packed fields with m and n bytes, then the sum might be as large as max(m, n) + 1 bytes. You may need to construct a work area to handle the maximum values. For instance,

```
FIELDA   DS    PL4
FIELDB   DS    PL3
WORK     DS    PL5
```

In planning to add FIELDA and FIELDB, we construct a work field called "WORK". The following code completes the task.

```
         ZAP    WORK,FIELDA
         AP     WORK,FIELDB
```

Op Code	L_1L_2	B_1D_1	D_1D_1	B_2D_2	D_2D_2

SP is a SS_2 instruction which is used to subtract packed decimal fields. Operand 1 is a field in storage which should contain a packed decimal number. The resulting sum develops in this field. The contents of Operand 2, another packed decimal field in storage, is subtracted from the contents of Operand 1 to produce the result which is stored in Operand 1. The operands are limited to a maximum length of 16 bytes and may have different sizes since this is a SS_2 instruction.

A decimal overflow (condition code = 3) will occur if the generated result loses a significant digit when it is placed in the target field. A decimal overflow may or may not cause a program interruption (abend). This depends on the setting of a bit in the PSW (See **SPM**). Otherwise, the condition code is set to indicate whether the result was zero (condition code = 0), negative (condition code = 1), or positive (condition code = 2). You can test these conditions with **BZ** or **BNZ**, **BM** or **BNM**, and **BP** or **BNP**.

Consider the following **SP** example.

```
          SP    APK,BPK
          BM    ANEGATIV
          ...
ANEGATIV DS   0H
```

Assume the variables are initially in the following states,

```
APK       DC    PL4'34'   = X'0000034C'
BPK       DC    PL3'22'   = X'00022C'
```

After the SP instruction has executed, the variables have the following values.

```
APK = X'0000012C'
PK  = X'00022C'
```

The contents of BPK were subtracted from APK and the result stored in APK. BPK was unaffected by the subtract operation. The branch is not taken since the condition code is positive.

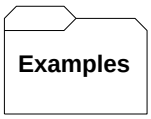
On the other hand, consider the following example,

```
            SP    AP,BPK
            ...
APK         DC    PL2'999'   =X'999C'
BPK         DC    PL2'-5'    =X'005D'
```

After the SP instruction has executed, the variables have the following values.

```
APK = X'004C'
BPK = X'005D'
```

Notice that an overflow occurred in APK with the loss of a significant digit. APK was too small to hold the difference that developed as a result of the **SP**.



Some unrelated SP's:

```
A           DC    PL3'12345'      =X'12345C'
B           DC    PL2'-32'        =X'032D'
C           DC    Z'11'           =X'F1C1'
            ...
            SP    A,=P'20'        A = X'12325C' C.C. = HIGH/POSITIVE
            SP    B,=P'20'        B = X'052D'   C.C. = LOW/NEGATIVE
            SP    B,=P'-40'       B = X'008C'   C.C. = HIGH/POSITIVE
            SP    A,=P'1'         A = X'12344C' C.C. = HIGH/POSITIVE
            SP    A,B             A = X'12377C' C.C. = HIGH/POSITIVE
            SP    B,B             B = X'000C'   C.C. = EQUAL/ZERO
            SP    B,A             B = X'377D'   C.C. = OVERFLOW
            SP    A,C             S0C7 DATA EXCEPTION - C NOT PACKED
```

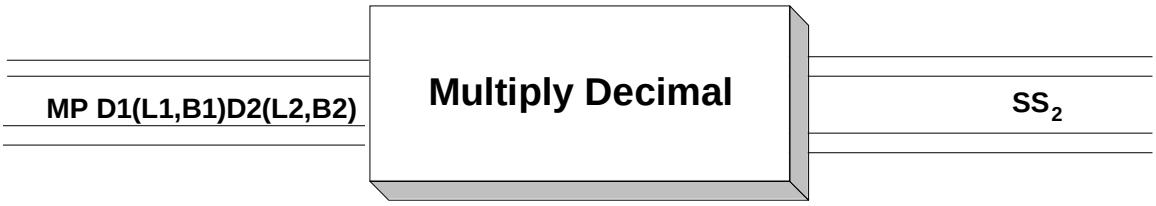
Tips

1. Become familiar with your data. The best way to prevent overflows is to plan the size of your fields based on the data at hand. There is a rule of thumb that you can follow for subtractions: If you are subtracting two packed fields with m and n bytes, then the difference might be as large as max(m, n) + 1 bytes. You may need to construct a work area to handle the maximum values. For instance,

```
FIELDA    DS    PL3
FIELDB    DS    PL5
WORK      DS    PL6
```

In planning to subtract FIELDB from FIELDA, we construct a work field called "WORK" with max(3,5) + 1 = 6 bytes. The following code completes the task.

```
            ZAP   WORK,FIELDA
            SP    WORK,FIELDB
```



Op Code	L_1L_2	B_1D_1	D_1D_1	B_2D_2	D_2D_2

MP is a SS_2 instruction which is used to multiply packed decimal fields. Operand 1 is a field in storage which initially contains a packed decimal number representing the multiplicand. The product develops in this field, destroying the multiplicand. The contents of Operand 2, another packed decimal field in storage, represents the multiplier. This field is unchanged by the multiplication (unless it also participates as operand 1). The condition code is not set by this instruction.

There are three rules that must be followed concerning the lengths of the fields that participate in a multiply packed instruction.

1) The multiplier is limited to a maximum of 8 bytes.

2) The multiplicand field can be as large as 16 bytes.

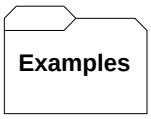
3) If the length of operand 2 is "L2", then the multiplicand must contain at least L2 bytes of leading zeroes before the instruction is executed.

For example,

```
APK      DC    PL4'12345'    =X'0012345C'
BPK      DC    PL2'100'      =X'100C'
         ...
         MP    APK,BPK
```

The **MP** above causes an "0C6" abend because APK contains only 1 byte of leading zeroes and BPK is 2 bytes long. To multiply APK and BPK above, we must code a work area that is large enough to support two bytes of leading zeroes. Adding the lengths of both fields we see that a field of length 6 is large enough to satisfy this requirement.

```
WORK     DS    PL6
         ...
         ZAP   WORK,APK
         MP    WORK,BPK
```

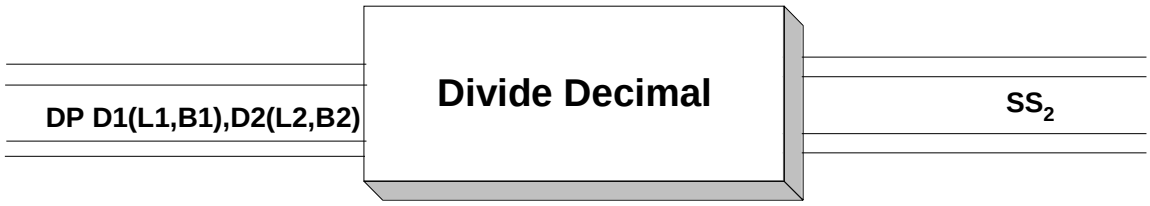


Some unrelated MP's:

```
A        DC    P'25645'       =X'25645C'     3 BYTES
B        DC    P'11'          =X'011C'       2 BYTES
C        DC    P'-4'          =X'4D'         1 BYTE
D        DC    P'9876543'     =X'9876542C'   4 BYTES
WORK     DS    PL6
         ...
         ZAP   WORK,A         WORK = X'00000025645C' 3 BYTES LEADING 0S
         MP    WORK,=PL2'20'  WORK = X'00000512900C'

         ZAP   WORK,B         WORK = X'00000000011C' 4 BYTES LEADING 0S
         MP    WORK,B         WORK = X'00000000121C'

         ZAP   WORK,A         WORK = X'00000025645C' 3 BYTES LEADING 0S
         MP    WORK,B         WORK = X'00000282095C'

         ZAP   WORK,A         WORK = X'00000025645C' 3 BYTES LEADING 0S
         MP    WORK,D         ABEND 0C6 NOT ENOUGH LEADING 0S
```

Tips

1. When creating a work field for a multiplication, the rule of thumb is to make the work area at least as large as the sum of the fields that contain the multiplicand and the multiplier. There is no harm in overestimating the size of the work area, but an area that is too small will cause an abend.



Op Code	L_1L_2	B_1D_1	D_1D_1	B_2D_2	D_2D_2

DP is a SS_2 instruction which is used to divide two packed decimal fields and produce a quotient and a remainder. Since packed decimal fields are integers, the division that occurrs is an integer division. If you are interested in producing a quotient with decimals, you may need to use **SRP** to shift the quotient before the division, and **ED** to provide a decimal point in the output. (See the topic on "**Decimal Precision**".)

Operand 1 is a field containing a packed decimal number which is the dividend. Both the quotient and the remainder develop in this field. Operand 2 is a packed decimal field containing the divisor. The maximum length of the dividend is 16 bytes, and the maximum length of the divisor is 8 bytes. Keep in mind the following rules when thinking about the sizes of the generated quotient and remainder,

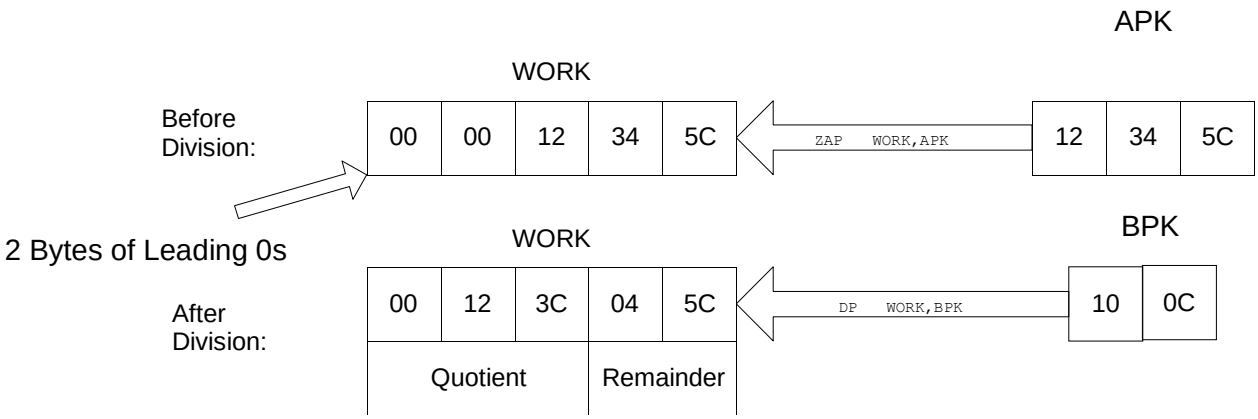
1) The remainder size is equal to the size of the divisor.

2) The quotient occupies the portion of operand 1 that is not occupied by the remainder.

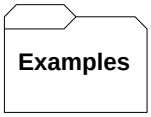
The rules make practical sense when you think about the division process. Suppose you divide an m- byte divisor into an n-byte dividend. Without knowing the specific values, we can see that in an integer division, the remainder might be as large as m bytes, and the quotient might be as large as n-bytes. This observation leads to the following rule of thumb for creating work fields in which to perform the calculation,

If the dividend is an n-byte field, and the divisor is an m-byte field, make the work field for the computation at least n+m bytes.

Following this rule of thumb will ensure that the dividend contains enough bytes of leading zeroes before the computation. Specifically, the number of bytes of leading zeroes in the dividend must be at least large as the number of bytes in the divisor. Here is a sample computation where we want to divide APK by BPK. Since APK is 3 bytes and BPK is 2 bytes, we create a work area called "WORK" which is 5 bytes. This will provide the required number of bytes of leading 0's for our computation.


APK
WORK
Before
Division:
00
00
12
34
5C
ZAP WORK,APK
12
34
5C
2 Bytes of Leading 0s
BPK
WORK
After
Division:
00
12
3C
04
5C
DP WORK,BPK
10
0C
Quotient
Remainder

DP does not set the condition code.



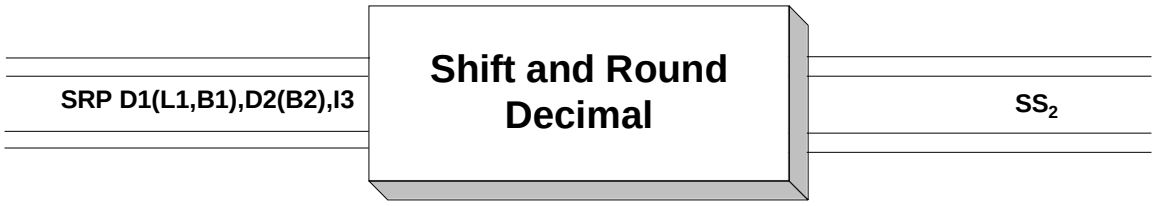
Some unrelated DP's:

```
LPK      DC    PL4'100   =X'0000100C'
MPK      DC    PL3'6'    =X'00006C'
NPK      DC    PL2'6'    =X'006C'
OZONE    DC    Z'11'     =X'F1C1'

WORK1    DS   0CL7
QUOT1    DS    PL4
REM1     DS    PL3

WORK2    DS   0CL7
QUOT2    DS    PL5
REM2     DS    PL2
         ...
         ZAP   WORK1,LPK     WORK1 = X'0000000000100C'
         DP    WORK1,MPK     WORK1 = X'0000016C00004C'
                             QUOT1 = X'0000016C'
                             REM1  = X'00004C'

         ZAP   WORK2,LPK     WORK2 = X'0000000000100C'
         DP    WORK2,NPK     WORK2 = X'000000016C004C
                             QUOT2 = X'000000016C'
                             REM2  = X'004C'

         ZAP   WORK1,LPK     WORK1 = X'0000000000100C'
         DP    WORK1,OZONE   ABEND - OZONE NOT PACKED

         ZAP   WORK1,=P'12345678' WORK1 =X'0000012345678C'
         DP    WORK1,MPK     ABEND - NOT ENOUGH LEADING ZEROES
                             REQUIRES 3 BYTES
```

Tips

1. Don't divide by zero. An attempt to divide by zero causes an "0CB" abend. Protect your divisions by testing the divisor before you divide.

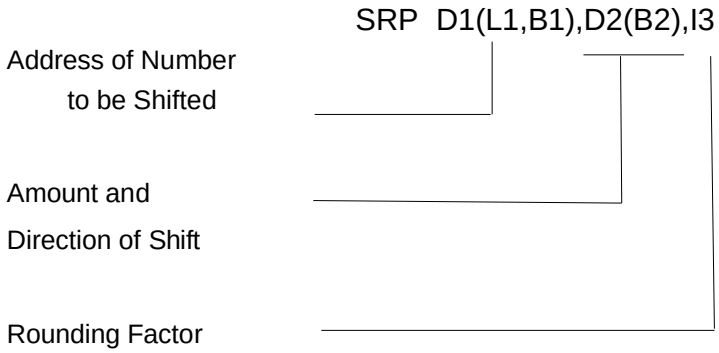
```
            ZAP   BPK,BPK       IS DIVISOR 0?
            BZ    ZERODIV       BRANCH IF ZERO
            ZAP   WORK,APK      OTHERWISE...
            DP    WORK,BPK      WE CAN DIVIDE
            ...
ZERODIV     DS   0H
            (CODE TO HANDLE A ZERO DIVISOR)
```



Op Code	L_1I_3	B_1D_1	D_1D_1	B_2D_2	D_2D_2

SRP is a SS_1 instruction designed to shift the digits in a packed decimal field either left or right while leaving the sign of the field fixed in its position. For right shifts, the last digit that is shifted off can be rounded using a rounding factor. As digits are shifted off during a left shift, 0's are shifted in on the right. On the other hand, as digits are shifted off during a right shift, 0's are shifted in on the left.

Consider the explicit format for this instruction.



As we can see from the diagram, operand 1 specifies the field to be shifted and its length, while the third operand is a rounding factor. This factor is added to the last digit which is shifted off during right shifts.

Typically, you would code the third operand as 5 to round up on a digit of 5 or greater. Coding 0 as the third operand indicates no rounding. A rounding factor must always be coded, even when shifting left.

The second operand above receives special treatment. The notation D2(B2) is converted to an effective address and the last 6 bits of this address is treated as a two's complement number that indicates the number of digits to be shifted and in which direction. A positive number indicates a left shift, and a negative number indicates a right shift. Because of the explicit notation, there are two standard ways to code the second operand.

1) Displacement Only -

```
SRP  XPK,3,0
```

In this case the shift factor, 3, is converted to 6-bit binary (B'000011') and the result is interpreted as 6-bit 2's complement. In this format B'000011' is positive 3 - a left shift by 3.

On the other hand, consider the following example,

```
SRP XPK,62,5
```

The 62 is converted to its binary form (B'111110') and interpreted as a 6-bit 2's complement integer. We see that B'111110' = -2 in base 10. This means the previous shift is a 2-digit right shift.

There is a simple way to deal with the shift factor on right shifts. Simply express the shift factor as 64 - n, where n is the number of digits you wish to shift to the right. For example, the previous example could be expressed as SRP XPK,64-2,5 . The 64 - n expression produces the correct decimal number which will be interpreted as a negative 2's complement integer.

2) Base Register Only –

```
SRP XPK,0(R8),5
```

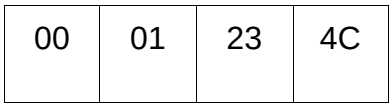
In this format the base register (R8) is initialized with a number representing the shift factor. Because 0 was coded for the displacement, the effective address will be computed to be the contents of the specified register. The number of digits to be shifted and the direction is solely determined by the contents of the register at the time the instruction is executed. This allows us to dynamically change the shift factor. Consider the code below.

```
La    R6,10           SHIFT FACTOR = 10 DIGITS LEFT
SRP   XPK,0(R6),5     DYNAMIC SHIFT
```

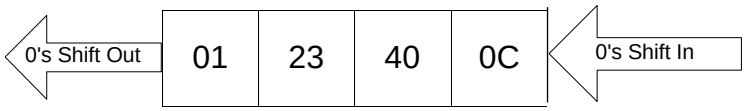
The number 10 (indicating a 10-digit left shift) is loaded into R6. The effective address will be 10.

Here are two unrelated SRP's that indicate what happens at the byte level. Assume XPACK is initialized as illustrated below for both SRP's

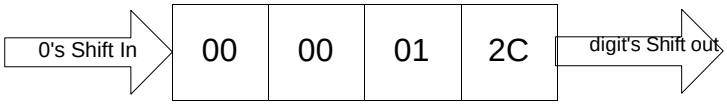
XPACK (Before)

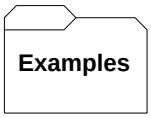


XPACK
After SRP XPK,2,5



XPACK
After SRP XPK,64-2,5





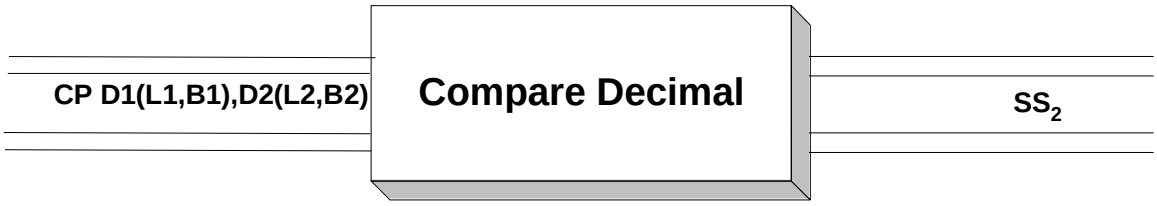
Some unrelated SRP's:

```
APK      DC    PL4'126'          APK = X'0000126C'
BPK      DC    PL4'1122334'      BPK = X'1122334C'
         ...
         SRP   APK,3,5        APK = X'0126000C' LEFT SHIFT 3 DIGITS
         SRP   APK,3,0        APK = X'0126000C' SHIFT AMT NOT A FACTOR
                                                FOR LEFT SHIFTS
         SRP   APK,64-1,5     APK = X'0000013C' RIGHT SHIFT WITH ROUND
         SRP   BPK,1,0        OVERFLOW OCCURS ON LOSING SIG DIG ON LEFT
         SRP   APK,-2,5       ASSEMBLY ERROR – NEG DISPLACEMENT INVALID
```

Tips

1. Keep in mind that you cannot left-shift off a significant (non-zero) digit.

2. Consider using the 64-n notation when coding right shifts. It is easy to read and provides excellent documentation.



Op Code	L_1L_2	B_1D_1	D_1D_1	B_2D_2	D_2D_2

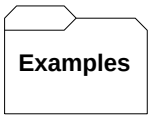
CP is a SS_2 instruction which is used to compare packed decimal fields. This instruction sets the condition code to "equal" (condition code = 0), "low" (condition code = 1) or "high" (condition code = 2) based on a comparison of the two fields as decimal numbers and indicates how the first operand compares to the second operand. The fields may be of equal or different sizes. The only restrictions on the field lengths are that they must be a maximum of 16 bytes in length (the typical restriction for SS_2 fields). Consider the fields and instructions below.

```
APK       DC     P'123'     =X'123C'
BPK       DC     PL3'100    =X'00100C'
CHX       DC     X'123F'    =X'123F'
          ...
          CP     APK,BPK
```

```
APK       DC    P'123'      = X'123C'
BPK       DC    PL3'100'    = X'00100C
CHX       DC    X'123F'
          ...
          CP    APK,BPK     C.C.= HIGH
          CP    APK,CHX     C.C.= EQUAL
          CLC   APK,CHX     C.C.= LOW
```

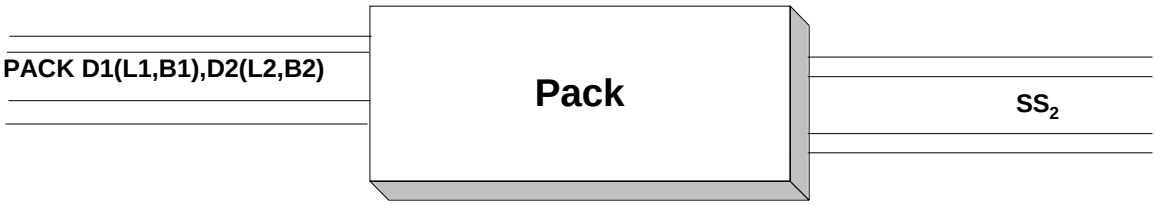
In the first **CP**, the fields are compared arithmetically, and it is found that +123 is greater than +100. In the second compare, the condition code is set to "equal" since +123 is equal to +123 (x'F' is a valid plus sign). The third compare is a logical compare rather than an arithmetic compare. Since x'3C' is lower than x'3F' in the EBCDIC collating sequence, the condition code is set to "low".

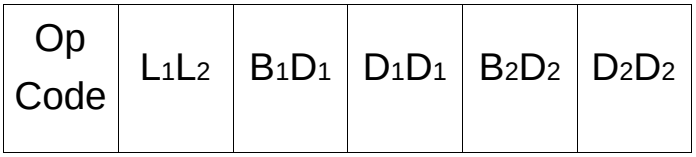
After setting the condition code with a **CP**, the condition code can be tested with a branch instruction. The typical branch instructions you might use are BE or BNE, BL or BNL, and BH or BNH.



Some unrelated CP's:

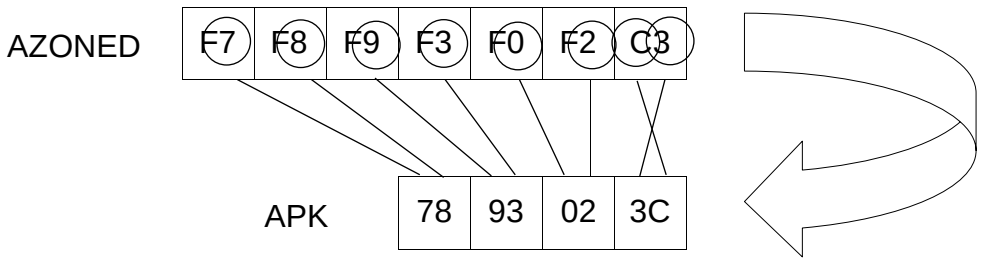
```
QPK      DC    P'12345'          =X'12345C'
RPK      DC    P'-32'            =X'032D'
SZONED   DC    Z'11'             =X'F1C1'
         ...
         CP    QPK,=P'20'        C.C.=HIGH
         CP    RPK,=P'20'        C.C. = LOW
         CP    SZONED,=P'999'    ABEND - SZONED IS NOT PACKED
         CP    QPK,QPK           C.C. = EQUAL
         CP    QPK,RPK           C.C. = HIGH
         CP    RPK,QPK           C.C. = LOW
         CP    QPK,=X'324A'      C.C. = HIGH - LITERAL IS PACKED DATA
         CP    QPK+2(1),RPK      BAD IDEA, BUT IT DOES WORK
                                 C.C. = HIGH,
```





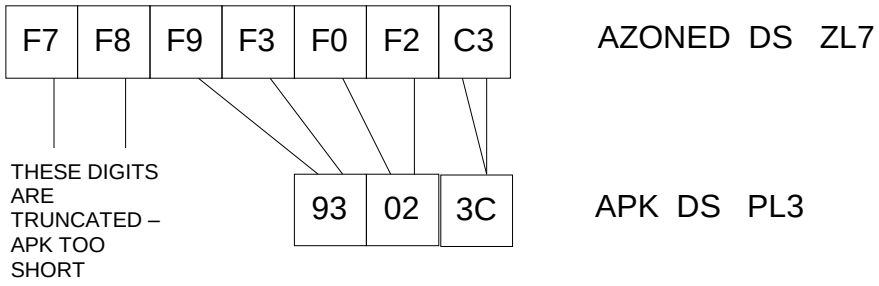
PACK is a SS_2 instruction which is designed to convert data from character or zoned decimal format to packed decimal format. The operation proceeds by transferring the contents of operand 2 to operand 1. Bytes in operands 1 and 2 are referenced from right to left within the fields. The rightmost byte of operand 2 is referenced first, and the zone and numeric parts of the byte are reversed and placed in the rightmost byte of operand 1. Then the numeric parts of the next two bytes of operand 2 are placed in the next byte of operand 1. This process continues by "packing" the numeric parts of operand 2 into operand 1, always moving from right to left, taking two bytes, and packing into one byte. The example below illustrates this idea. The first field, AZONED, contains zoned decimal data, while the second field, APK, shows the results of executing "PACK APK,AZONED". We assume the following definitions,

```
AZONED   DC    ZL7'7893023'   =X'F7F8F9F3F0F2C3'
APK      DS    PL4
         ...
         PACK  APK,AZONED
```



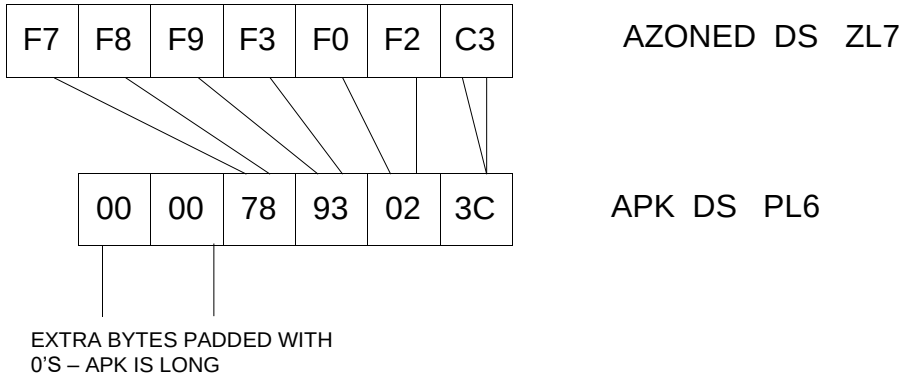
Since **PACK** is SS_2, each operand contributes a 4-bit length which is stored in the second byte of the object code as pictured at the top of this page. The maximum length in the object code is B'1111' = 15. Since the assembler always decrements lengths by 1, packed fields are limited to a maximum of 16 bytes. It is also important to note that the lengths of both fields are used to execute this instruction. For instance, if APK was defined as PL3, the high-order digits in the packed field (those on the left) are truncated.

```
PACK APK,AZONED
```



On the other hand, if there are too few bytes in operand 2, zeros will be padded on the left in operand 1 as illustrated below. Assume APK was defined as PL6.

```
PACK APK,AZONED
```



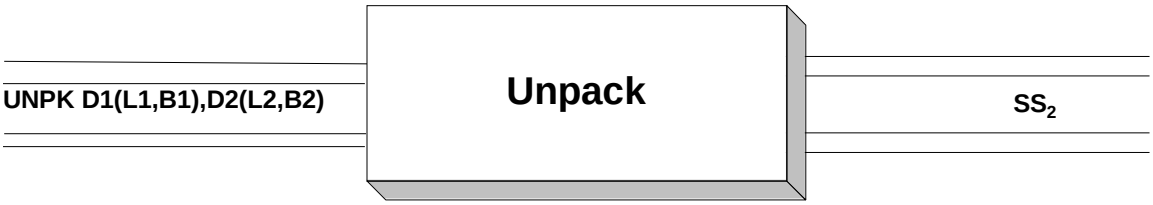
Examples

Some unrelated PACK's:

```
A         DC     C'12345'        =X'F1F2F3F4F5'
B         DC     Z'1234'         =X'F1F2F3C4'
C         DC     C'ABC'          =X'C1C2C3'
D         DC     X'12ABCDEF'     =X'12ABCDEF'
P         DS     PL3
Q         DS     PL2
R         DS     PL4
                 ...
          PACK   P,A              P = X'12345F'    SIGN UNCHANGED
          PACK   Q,A              Q = X'345F'      LEFT TRUNCATION
          PACK   R,A              R = X'0012345F'  PAD ZEROES
          PACK   P,B              P = X'01234C'
          PACK   Q,B              Q = X'234C'      LEFT TRUNCATION
          PACK   R,B              R = X'0001234C'  PAD ZEROES
          PACK   P,C              P = X'00123C'    PACK WILL PROCESS
                                                   ANY DATA WITHOUT ABEND
          PACK   P,D              P = X'02BDFE'    IT WILL PACK ANYTHING!
```

Tips

1. **PACK** was designed to work on zoned decimal data, but it works blindly with any kind of data. That does not mean that you will compute correct results when it is applied to fields that aren't zoned decimal. It simply means that the program will not abend because the data wasn't zoned. Packed decimal problems show up when arithmetic instructions are executed on fields that aren't packed.



Op Code	L_1L_2	B_1D_1	D_1D_1	B_2D_2	D_2D_2

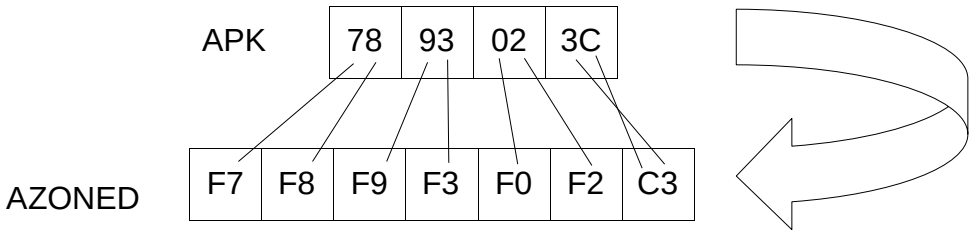
UNPK is a SS2 instruction which is designed to convert data from a packed format to a zoned decimal format. The operation proceeds by transferring the contents of operand 2 to operand 1. Bytes in operands 1 and 2 are referenced from right to left within the fields. The rightmost byte of operand 2 is referenced first, and the zone and numeric parts of the byte are reversed and placed in the rightmost byte of operand 1. The next byte (moving right to left) in operand 1 is processed by taking it numeric part, prefixing it with bits 1111, and storing it in the next byte (moving right to left) of operand 1. Similarly, the zone portion of the byte is prefixed with bits 1111 and stored in the "next" byte of operand 1. This process of splitting the zone and numeric portions of a byte and prefixing them with bits 1111 continues with all subsequent bytes in operand 2. The process is terminated (with possible truncation) if operand 1 becomes filled. Keep in mind that the lengths of both operands are provided in the instruction since it has an SS2 format. If all the bytes in operand 2 are processed and there are bytes remaining in operand 1, the bytes are filled with X'F0's. If operand 1 is filled before processing all the bytes in operand 2, high-order truncation of the value in operand 1 will occur.

The example below illustrates this idea. The first field, APK, contains packed decimal data, while the second field, AZONED, shows the results of executing the code below.

```
UNPK AZONED,APK
```

We assume the following definitions,

```
APK      DC    PL4'7893023'    =X'7893023C'
AZONED   DS    ZL7
```

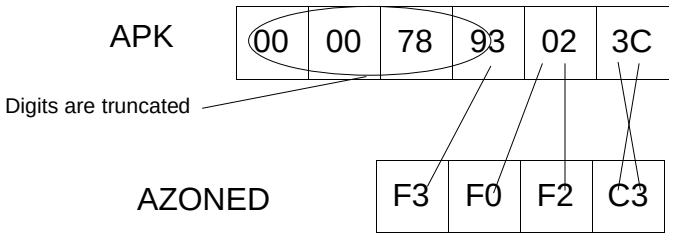


Since **UNPK** is SS2, each operand contributes a 4-bit length which is stored in the second byte of the object code as pictured at the top of this page. The maximum length in the object code is B'1111' =15. Since the assembler always decrements lengths by 1, packed fields are limited to a maximum of 16 bytes. It is also important to note that the lengths of both fields are used to execute this instruction. For instance, if AZONED were defined as ZL3, the high-order digits in the packed field (those on the left) are truncated in the result. Consider execution of the following instruction.

```
UNPK AZONED,APK
```

We assume the following definitions,

```
APK      DC    PL6'7893023'
AZONED   DS    ZL4
```



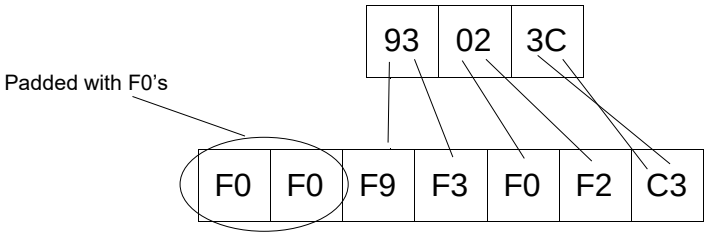
The rightmost bytes of APK are processed and placed in AZONED. When this field is full, the remaining bytes in APK are truncated.

On the other hand, if there are too few bytes in operand 2, zeros will be padded on the left in operand 1 as illustrated below. Assume the following declarations,

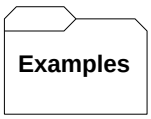
```
APK      DC P'93023'
AZONED   DS ZL6
```

Suppose we execute the following instruction.

```
UNPK AZONED,APK
```



All the bytes of APK are processed and fill up five of the bytes in AZONED. Since AZONED was defined as a seven-byte field, the leftmost 2 bytes are padded with X'F0's.



Some Unrelated UNPK's:

```
A       DC      P'12345'        =X'12345C'
B       DC      PL3'-3'         =X'00003D'
N       DS      ZL7
P       DS      ZL5
Q       DS      ZL3
R       DS      ZL2
        ...
        UNPK    N,A             P = X'F0F0F1F2F3F4C5' ZERO PADDING
        UNPK    P,A             P = X'F1F2F3F4C5'
        UNPK    Q,A             Q = X'F3F4C5' LEFT TRUNCATION
        UNPK    R,B             N = X'F0D3'
```

Tips

1. **UNPK** will operate on any kind of data. That does not mean that you will like the results. It simply means that the program will not abend because of the data in a field you are unpacking.

2. Use **ED** or **EDMK** to convert data to a printable format. **UNPK** leaves packed data in a format that is not acceptable for printing. For example, consider unpacking XPK into XZONE.

```
XPK     DC      P'15'           XPK = X'015C'
XZONE   DS      ZL3
```

After executing "`UNPK XZONE,XPK`", `XZONE` contains `X'F0F1C5'`. When printed the field would appear as `01E`, since `X'C5'` is equivalent to a character E.

Op Code	LL_1	B_1D_1	D_1D_1	B_2D_2	D_2D_2

Packed decimal fields can be converted to a character format using the **ED** instruction. Additionally, editing symbols and features like commas, decimal points, and leading zero suppression can be included in the character version of the packed decimal field that is being edited. The first step in editing a packed decimal field is to create an "edit word" which is a pattern of what the character output of the edit process should look like. Typically, the edit word is moved to a field in the output buffer, which is being built, prior to printing. Then the packed decimal field is "edited" into the output field, destroying the copy of the edit word.

First, we consider how to construct an appropriate edit word for a given packed decimal field. This can be accomplished by defining a string of hexadecimal bytes that represents the edit word. Each byte in the edit word corresponds to a byte in the edited character representation. In creating the edit word there are a collection of standard symbols which are used to describe each byte: X'40' This symbol, which represents a space, is usually coded as the first byte of the edit word where it acts as a "fill character". The fill character is used to replace leading zeroes which are not "significant".

X'20' Called a "digit selector", this byte represents a position in which a significant digit from the packed field should be placed.

X'21' This hexadecimal byte represents a digit selector and a significance starter. Significance starts when the first non-zero digit is selected.

Alternatively, we can force significance to start by coding a single x'21' in the edit word. In this case, significance starts in the byte **following** the x'21'.

Significance is important because every significant digit is printed, even if it is a leading zero.

X'6B' This is the EBCDIC code for a comma.

X'4B' This is the EBCDIC code for a decimal point.

X'60' This is the EBCDIC code for a minus sign. This symbol is sometimes coded on the end of an edit word when editing signed fields. The x'60' byte will be replaced with the fill character if the number being edited is positive. If the number is in fact negative, the x'60' will not be "filled", and the negative sign will appear in the edited

output.

X'C4C2' The EBCDIC version of "DB" (Debit). This functions like the x'60'. Coding these symbols at the end of an edit word causes "DB" to appear in the output if the field being edited is negative, otherewise the "DB" is "filled".

X'C3D9' The EBCDIC version of "CR" (Credit). This functions like the Debit symbol above. When the number being edited is negative, the "CR" symbol will appear in the edited output, otherwise it will be "filled".

X'5C' The EBCDIC symbol for an asterisk. This character is sometimes used as a fill character when editing dollar amounts on checks.

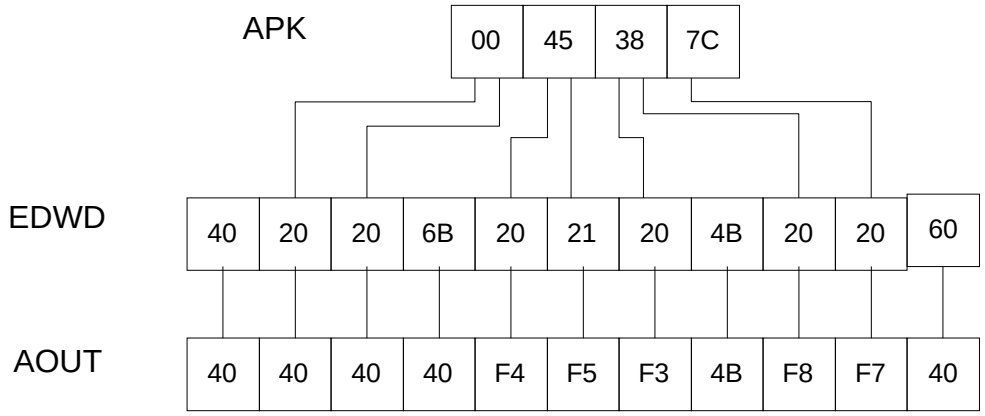
We now consider a sample edit word and the output it would create for several packed fields.

```
EDWD     DC     X'4020206B2021204B202060'
AOUT     DS     CL11
APK      DC     PL4'45387'
```

Assume we execute the instructions below,

```
         MVC    AOUT,EDWD
         ED     AOUT,APK
```

First, the edit word is moved to a field in the output buffer. Then the packed field is "edited" into the output field. The results are illustrated in the diagram below.



The diagram indicates the results of the edit process: The fill character (x'40') is unaffected and is left in its position. The first decimal digit, 0, is "selected" by the first x'20', and since leading 0's are not significant, the x'20' is replaced by the fill character. The second digit, 0, is also selected, and it too, is filled with a x'40'. Since significance has not started, the x'6B' is filled with x'40'. The first nonzero digit, 4, is selected and this signals that significance has started. (Any non-zero digit which is selected turns on the significance indicator.) Each digit after the 4 will appear in the edited result. The "4" is replaced with its character equivalent - x'F4'. The "5" is selected and its x'20' is replaced with x'F5'. The "3" is selected and is represented as x'F3'. The x'4B', a decimal point, remains unaffected. The "8" is selected and is represented as x'F8'. The "7" is

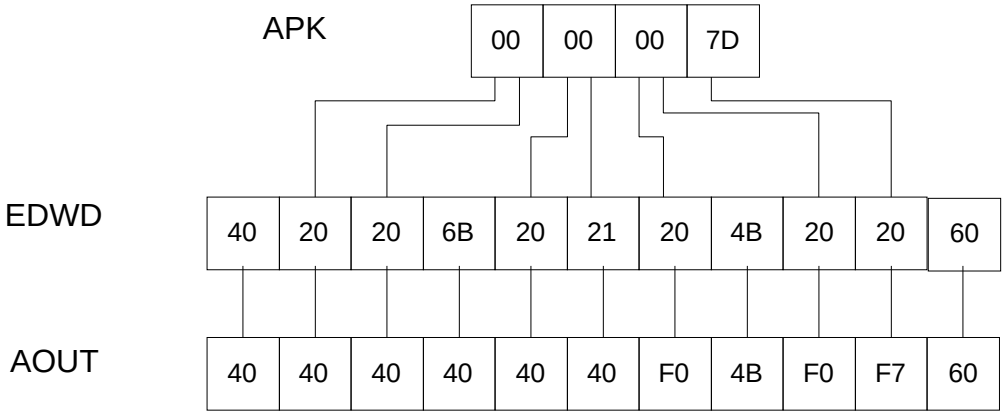
selected and is represented as x'F7'. Since the number being edited is positive, the x'60' is filled with x'40'. The result would print as 453.87.

Consider a second edit which uses the same edit word as in the previous example, but with a different value for APK.

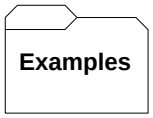
```
EDWD     DC    X'4020206B2021204B202060'
AOUT     DS    CL11
APK      DC    PL4'-7'
```

Again, we execute the same sequence of instructions.

```
         MVC   AOUT,EDWD
         ED    AOUT,APK
```



As in every edit, the x'40' fill character is unaffected by the edit process. The first and second digits, both 0, are selected, and since they are leading 0's and significance has not started, they are filled with x'40'. The x'6B' is also filled with x'40' since significance has not started. The next two digits, both 0, are selected and filled. Since the x'21' selected a leading 0, the significance indicator is turned on - significance starts with the **next** digit. This means that all other digits will appear in a character representation, even if they are leading 0's. All other editing symbols will be printed as well. The fifth digit, 0, is selected and represented as x'F0'. The x'4B' is preserved. The next two digits, 0 and 7, are selected and represented as x'F0' and x'F7'. Finally, since the APK contains a negative number, the x'60' is preserved. The result would print as 0.07-.



Some Unrelated ED's:

```
APK      DC    PL2'123'      =X'123C'
AOUT     DS    CL4
AEDWD    DC    X'40202020'
         ...
         MVC   AOUT,AEDWD
         ED    AOUT,APK      AOUT = X'40F1F2F3' =C' 123'


BPK      DC    PL2'0'        =X'000C'
BOUT     DS    CL4
BEDWD    DC    X'40202020'
         ...
         MVC   BOUT,BEDWD
         ED    BOUT,BPK      BOUT = X'40404040'  =C'    '


CPK      DC    PL2'0'
COUT     DS    CL4
CEDWD    DC    X'40202120'
         ...
         MVC   COUT,CEDWD
         ED    COUT,CPK      COUT = X'404040F0'  =C'   0'


DPK      DC    PL2'0'        =X'000C'
DOUT     DS    CL4
DEDWD    DC    X'5C202120'   ASTERISK IS USED AS A FILL CHARACTER
         ...
         MVC   DOUT,DEDWD
         ED    DOUT,DPK      DOUT = X'5C5C5CF0'  =C'***0'


EPK      DC    PL2'-30'      X'030D' NEGATIVE INTEGER
EOUT     DS    CL4
EEDWD    DC    X'40202120'
         ...
         MVC   EOUT,EEDWD
         ED    EOUT,EPK      EOUT = X'4040F3F0'  =C'  30'


FPK      DC    PL2'=30'      X'030D' NEGATIVE INTEGER
FOUT     DS    CL5           OUTPUT FIELD SIZE MATCHES FEDWD LENGTH
FEDWD    DC    X'4020212060'
         ...
         MVC   FOUT,FEDWD
         ED    FOUT,FPK      FOUT = X'4040F3F060' =C'  30-'
```

Tips

1) There are two common errors that beginners make when using **ED**:

- The number of x'20's and x'21's does not match the number of decimal digits in the field being edited. This is a critical error. If the packed field has length "n", the number of x'20's and x'21's is 2n - 1. For example, if you are editing a packed field of length 6, the edit word must contain exactly 11 x'20's and x'21's. A bad edit word will produce unpredictable output.

- The output field size does not match the edit word size. For example, suppose you coded the following,

```
AEDWD     DC    X'402020202120'
AOUT      DS    CL5
```

When the edit word is moved to AOUT, the last byte of the edit word is not moved since the edit word is 6 bytes and the target field is 5 bytes. The effect is that we are using an incorrect edit word, even though the definition of the edit word was correct.

2) When editing, start with the packed field and design an edit word that matches it. Then define the output field to match the edit word. For example, if we start with a packed field of length 3 (5 decimal digits), we could design x'402021204B2020' as an appropriate edit word (5 x'20's and x'21's). Since the edit word is 7 bytes long, we would design a 7-byte output field to hold the edit word.

```
XPK       DS      PL3
XEDWD     DC      X'402021204B2020'
XOUT      DS      CL7
          ...
          MVC     XOUT,XEDWD
          ED      XOUT,XPK
```

Trying It Out in VisibleZ:

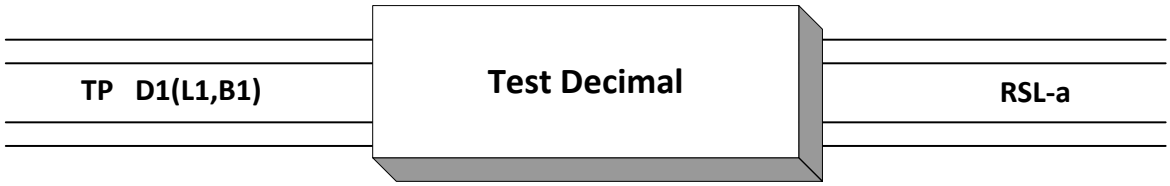
1) Load the program **ed.obj** from the \Codes directory and single step through each instruction until you are about to execute the ED instruction.

2) What are the contents of the source field? What do they represent?

3) What are the contents of the target field? What do they represent?

4) What is the length associated with the ED instruction?

5) After stepping through the ED instruction, what will the target field contain in a character representation?



EB	L_1////	B_1D_1	D_1D_1	////////	C0

TP is an RSL-a instruction that is used to test a field to see if it is in a packed decimal format. A field of length 16 or less is in a packed decimal format if each byte contains two hex digits in the range 0-9, except the rightmost byte which contains a hex digit (0-9) followed by valid sign (hex values A, B, C, D, E, or F). The hex signs A, C, E, and F are positive, while hex signs B and D are negative.

This instruction is a little unusual in that it has a single operand, which denotes the field that is being tested. As a result, some of the space in the object code is unused as shown in the object code diagram above. TP also has a two-byte op-code – EBC0. TP examines the target field and sets the condition code to indicate the results:

Condition Code:

1. 0- All digit codes and the sign valid
2. 1- Sign invalid
3. 2- At least one digit code invalid
4. 3- Sign invalid and at least one digit code invalid

As you can see, in those cases where the target field is not packed, the instruction also provides information about what is wrong with the tested field.

Consider the following TP example.

```
          TP   APK
          BNZ  NOTPACKED
          ...
NOTPACKED EQU  *
```

Assume that APK is in the following state when tested.

```
APK       DC  XL4'0000123C'
```

After executing the TP instruction above, the condition code is set to 0, since the field is packed. As a result, the branch is not taken. On the other hand, if APK had contained X'00000223', then TP would have set the condition code to 1 (bad sign), and the branch would have been taken.

Occasionally, date information is stored in a non-native format called "packed no-sign". For instance, the month, day, and year can be stored as four bytes |mm|dd|yy|yy| where the numeric portion of the rightmost byte is used to store a digit instead of a sign. If the same information were stored in a packed decimal field, it might look like |mm|dd|yy|yy|0C|which requires an extra, wasted byte. The problem with packed no-sign fields is that being a non-native format, there aren't

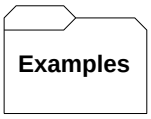
any machine instructions specifically designed to support it. Programmers have resorted to using TRT tables to recognize when a field contains valid packed no-sign data. But TP provides some support for this data format if we are clever. By moving a packed no-sign field adjacent to a one byte packed zero (X'0C') and then using TP to test the entire area, only two conditions codes are possible: 1) 0 – all the packed no-sign digits are valid, and 2) 2 – one of the packed no-sign digits was invalid.

Here's an example. Assume X is supposed to be a packed no-sign field with 4 bytes. We define the following fields,

```
WORK     DS   0XL5
XWORK    DS    XL4
         DC    X'0C'
```

We can then test the field like this,

```
         MVC   XWORK,X        BUILD THE WORK AREA
         TP    WORK           IS XWORK PACKED NO-SIGN?
         BNE   BADFIELD       BRANCH IF BAD
```



Some unrelated TP's:

```
A        DC       X'12345C'   GOOD DIGITS, GOOD SIGN
B        DC       X'123456'   GOOD DIGITS, BAD SIGN
C        DC       X'1A345C'   BAD DIGIT, GOOD SIGN
D        DC       X'1A3456'   BAD DIGIT, BAD SIGN
         ...
         TP A     C.C. = 0 - PACKED FIELD
         TP B     C.C. = 1 - BADSIGN
         TP C     C.C. = 2 - BADDIGIT
         TP D     C.C. = 3 - BADDIGIT,BADSIGN
```

Tips

1. Avoid S0C7s by using TP to test fields which are required to be packed.
2. TP can be used to test packed no-sign fields instead of using the more complicated TRT technique.