# PACKED DECIMAL ARITHMETIC

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 make sure they are printed in the correct positions on any report.

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

**Copying Packed Decimal 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. Many beginners make the mistake of using **MVC** when copying packed decimal values. This can lead to an error 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'
```

We are copying a 2-byte packed field, PKFIELD, to a 3-byte field, AFIELD. Since **MVC** has an $SS_1$ 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 **ZAP** above first 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 involved 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 = X'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
         BZ    WASZERO           BRANCH IF ZERO
```

## Adding Packed Decimal Fields

Next we consider the problem of adding several packed decimal fields. 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   DS    PL3
FIELD2   DS    PL3
FIELD3   DS    PL3
SUM      DS    PL7
```

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 7 field will hold the sum of any 3 packed length 3 fields.

## Subtracting Packed Decimal Fields

The comments above about adding 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 final result. The code below will compute the difference of FIELDA AND FIELDB.

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

The SP instruction is 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 compare two packed decimal fields and branch based on how the two fields compare to each other. In assembly language, packed fields can be compared using the **CP** instruction. This has the effect of setting the condition code. Branch instructions are then coded in order to 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,FIELDA     FIELDB > FIELDA?
        BNH   THERE             BRANCH IF EQUAL OR LOW
        ZAP   BIGGER,FIELDB     CHANGE TO THE LARGER VALUE
THERE   EQU   *
```

It is a beginner’s mistake to compare packed fields with the **CLC** instruction. The compare logical character instruction was not designed to accommodate packed decimal data. The following code illustrates some of the problems that can occur.

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

## 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 planning to multiply two fields, in this case FIELD1 and FIELD2, you must plan for a field which 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, prior to multiplying, 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 WITH THE DIVIDEND
         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 big 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 big 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 above, the following division would produce an error, eventually.

```
         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. A future reference to either of these fields as a packed decimal value will cause an abend.

**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 in order 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         P = X'000123000C'
                 SRP   Q,64-3,5      Q = X'0000010C'
                 ...
         P       DC    PL5'123'      P = X'000000123C'
         Q       DC    PL4'9876      Q = X'0009876C'
```

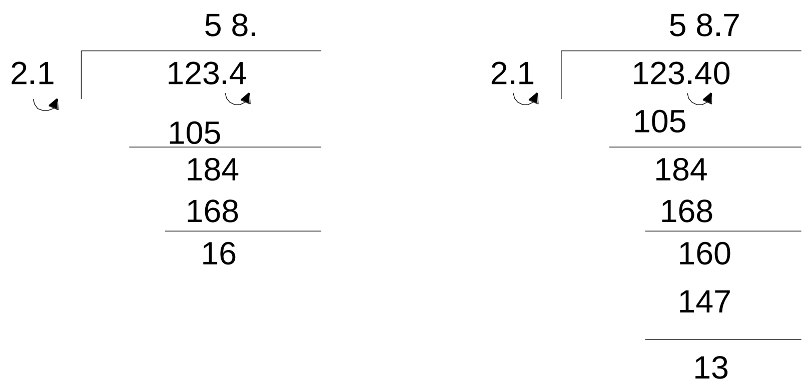
In the first **SRP**, the shift factor of 3 indicates a left shift by 3 digits. Three digits are lost on the left and 3 zero digits are shifted in on the right. This shift is logically equivalent to multiplying by 1000. In the second SRP, the shift factor of 64 - 3 indicates a right shift by 3 digits. 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, 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.

```
          ZAP   PRODUCT,S       INITIALIZE THE MULTIPLICAND
          MP    PRODUCT,T       ...2 DIGITS TO RIGHT OF DECIMAL PT
          SRP   PRODUCT,64-1,0  REMOVE ONE DIGIT WITHOUT ROUNDING
          ...
S         DC    PL3'1234.5'    S = X'12345C'  NO DECIMAL PT
T         DC    PL2'10.0'      T = X'100C'    NO DECIMAL PT
PRODUCT   DS    PL5
```

First the multiplicand is ZAPed into a 5 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'001234500C'). The **SRP** shift out the rightmost digit leaving PRODUCT = X'000123450C'. 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.



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 DIVISOR
        MVC   QUOTOUT,EDWD   EDIT WORD GOES TO OUTPUT AREA
        ED    QUOTOUT,QUOT   PREPARE QUOTIENT FOR PRINTING
        ...
M       DC    PL4'123.4'     M = X'0001234C'  (NO DECIMAL PT)
N       DC    PL2'2.1'       N = X'021C'   (NO DECIMAL PT)
WORK    DS   0CL6            WORK FIELD FOR DIVISION
QUOT    DS    PL4            QUOTIENT
REM     DS    PL2            REMAINDER
QUOTOUT DS    CL9
EDWD    DC    X'402020202021204B20'
```

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 two decimal places to the right of the decimal point. If we simply divide, the quotient would have 1 digit to the right of the decimal point. In order to finish with a quotient that has two digits rounded, we must generate 3 digits to the right of the decimal point before we divide. We can achieve this precision by shifting to the left by 2 digits before dividing. The following code illustrates this idea.

```
ZAP    WORK,X          PREPARE THE DIVIDEND
SRP    WORK,2,0        SHIFT IN 2 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' (NO DECIMAL PT)
        Y        DC      PL2'2.1'         Y = X'021C'   (NO DECIMAL PT)
                 WORK      DS    0CL7       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. In order 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 QUOTIENT BACK TO 2 DIGITS
                 ....
         WORK    DS   0CL10           WORK AREA
         QUOT    DS    PL8            QUOTIENT
         REM     DS    PL2            REMAINDER
```

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