Chapter 11 Basic Packed Decimal Instructions

Normally, when data are read into the memory of the computer, they are inputted in character form, also called zoned decimal format, and must be converted to packed form prior to being used in any mathematical operation. The result of the calculation must then be unpacked or converted back to character form prior to any printing operations. There is a basic dichotomy between the two data types: character and numeric. Character data can never be directly used in mathematical operations, but can be printed or displayed on a CRT. Numeric data, on the other hand, can be used directly in mathematical operations, but can never be printed or displayed directly.

Numeric data in the computer is also unique in that it can take one of three forms: packed, binary, or floating point. Perhaps the simplest form to begin to use in mathematical operations is packed decimal and are covered first. At least the packed decimal values are stored as decimal digits so that the results in main storage are more easily recognizable when something goes wrong.

In this chapter we present a description of the following basic packed decimal instructions: Pack, Add Pack, Unpack, Logical Or Immediate, Move Zone, Subtract Pack, and Compare Pack.

Cycle of Packed Decimal Arithmetic

The following cycle describes briefly the sequence of events required for preparing character data for use in packed decimal arithmetic operations.

A. Numbers are read into memory as character fields (C type) using the GET instruction.
B. The character fields are then converted into packed decimal fields (P type).
C. Arithmetic operations are performed on the packed decimal fields.
D. The packed decimal result field is unpacked back into a character field for printing.
E. The sign byte of the unpacked character field (the low order byte) must be fixed up.
F. The character field is then outputted using the PUT instruction.

Steps A and F should be fairly easy to accomplish at this point. Because this sequence is so very basic to understanding the operation of packed decimal arithmetic, it should be committed to memory. If you have to stop and figure out the next step to perform each time you are involved with an arithmetic operation, your productivity is reduced significantly.
High level languages, such as COBOL and PL/1, follow basically the same sequence. It is not necessary, however, that all of the detailed steps are coded in these high-level languages as must be coded in Assembler programs. This is because they are hidden from view as the compiler automatically inserts the proper sequences when it converts the code into machine language.

Let's examine the instructions in the order of the cycle of packed decimal arithmetic.

The PACK Instruction

The function of the PACK instruction is to convert character data in a C-type field and place it into packed P-type field. For example, a character field containing the data X'F1 F2 F3' would be packed into a packed field as X'12 3F'. The format of the PACK instruction is

```
PACK Operand1,Operand2
```

The PACK instruction is a SS (Storage to Storage) instruction where the data held in the second operand (Operand2) is packed into the first field (Operand1). Unless the two fields overlap, the data in Operand2 is unchanged and Operand1 is updated to contain the packed version of Operand2. Therefore, Operand1 must be the field defined as a packed P-type field and Operand2 is the character C-type field.

Both operands can have an implied or explicit length. This differs from all the preceding instructions we've covered thus far. This one can pack a small number into a large field. Basically, the pack instruction recognizes the integrity of both numbers in the packed decimal instruction set.

The maximum length for both operands is 16 bytes. Generally, the data in Operand2 must be valid numeric data F0 through F9 with no blanks or non-numeric characters.

The pack operation executes from right to left. That is, it packs the character in the unit's position of the field then the ten's position, then the hundred's position, etc. It is important to remember that the fields are NEVER checked for valid signs or valid digits. That is your responsibility. The PACK instruction is virtually indestructible. It (almost) never fails! Consider the following examples.

Example 1. PACK PFIELD,CFIELD
A) First, the sign is moved. The sign is the left nibble of the low order byte of the character field.
B) Insert the digits starting at the right going to the left within both fields.
C) If the fields are not of correct size, PACK pads the unfilled digit positions of the packed field with zeros or truncates any digits that exceed the size of the packed field.

**Truncation**

In Example 2, PFIELD, a one byte field, is not large enough to receive all of the digits from CFIELD. Truncation occurs.

```
PACK PFIELD, CFIELD
```

In Example 3, PFIELD is larger than CFIELD and the PACK fills the unused character positions with zeros as needed.

```
PACK PFIELD, CFIELD
```

**Padding**

In Example 3, PFIELD is larger than CFIELD and the PACK fills the unused character positions with zeros as needed.

```
PACK PFIELD, CFIELD
```

**Packing Bad Data**

Suppose that CFIELD is a three-byte field that contains two digits and a space character.

```
PACK PFIELD, CFIELD
```

As you can see, PACK executed correctly. However, as soon as you try to use the packed data in PFIELD, however, you get a data exception - invalid packed data - no sign. Remember, if you pack garbage (non-numeric data), you get compressed or packed garbage!
Valid signs in the sign position are the letter C or F for positive (+) data. The C is preferred. Negative (-) data is indicated by a D in the sign position.

Look at the padding and truncation that occur in the following series when the packed field is of various sizes. Assume CFIELD is defined as shown and that we execute the pack instruction

PACK PFIELD,CFIELD

CFIELD contains X'F1 F2 F3 F4'

If PFIELD is 1 byte then PFIELD = X'4F'
If PFIELD is 2 bytes then PFIELD = X'23 4F'
If PFIELD is 3 bytes then PFIELD = X'01 23 4F'
If PFIELD is 4 bytes then PFIELD = X'00 01 23 4F'

It is possible for the PACK instruction to pack data into itself. This is generally considered to be a bad practice since it can become very confusing. For example, if we had a field that was defined as

CFIELD DS CL3

and we execute the following PACK instruction

PACK CFIELD,CFIELD

The data would be packed as specified. However, if you were to look at the description of the field, you would reasonably assume that the field contained character data. Very confusing.

As previously indicated, the length of either or both operands may be specified explicitly in the instruction. But it is very infrequent that an explicit length is assigned to Operand1. Consider the following.

PFIELD DC PL2'0' appears as X'00 0C'
CFIELD DC CL3'123' appears as X'F1 F2 F3'
PACK PFIELD(1),CFIELD

After execution, the two fields would contain

CFIELD contains X'F1 F2 F3'
PFIELD contains X'3F 0C'

Notice that PFIELD has two signs. Generally this can mean trouble unless other processing occurs before any arithmetic add operations are attempted. Thus, it is very unusual to find explicit lengths applied to Operand1 fields.

On the other hand, an explicit length Operand2 can be used to advantage in some instructions. Examine the following sequences carefully to see the result of packing data where the data in Operand2 (CFIELD) is packed using an explicit length and relative addressing. In all cases, Operand1 (PFIELD) is defined as a two-byte field. The bracketed characters in CFIELD indicate those which are packed. Notice also that relative addressing can be used in the instruction to indicate the starting position for the packing operation.

PFIELD DS PL2 Note [] represents bytes to be packed

Contents after execution

PACK PFIELD,CFIELD

PACK PFIELD,CFIELD(1)
As you can see, it is possible to access or pick off parts of a larger field. A practical use of this might be in a situation where the date field is stored as character data in a YYMMDD sequence and has been defined as

```
DATE DS CL6
```

Assume that you want to pack the month number into a packed field defined as

```
PMONTH DS PL2
```

The following instruction would accomplish this objective

```
PACK  PMONTH, DATE+2 (2)
```

### Determining the Size of the Required Packed Field

Finally, it is important to know how large a packed field must be in order to pack the data from a given character field. One alternative is to make it "plenty" big. Although this can be done, it causes significant problems when performing multiply and divide operations. The formula below serves as a guide for determining the correct length of the packed field.

**Length of P-field = \( \frac{\text{Length of C-field} + 1}{2} \) - drop fractions**

If there is a fraction, it is dropped. Using this formula, a character field with the following lengths would require the corresponding packed field lengths.

- `CL1` \( \rightarrow \) `PL1`
- `CL2` \( \rightarrow \) `PL2`
- `CL3` \( \rightarrow \) `PL2`
- `CL7` \( \rightarrow \) `PL4`

This formula is one that you want to memorize. When writing assembler code, you may find that you are forever converting character fields into packed fields as you perform calculations. You cannot afford to be slow when making these conversions. It can cost you too much time.

A PACK instruction, along with all of the decimal instructions have lengths for both operands. Since the packed version takes up fewer total bytes, there must be lengths for both operands. The maximum length for both operands is 16 bytes.

Why? All packed decimal machine instructions occupy 6 bytes. The first byte contains the op code that identifies the instruction. The address of each operand is stored in 2 bytes. That leaves only one byte remaining for the lengths. Recall that MVC and CLC instructions are also
SS instructions that occupy 6 bytes. They also have only 1 byte to store the length to move or compare. The length byte stores the number in fixed binary format. So for MVCs, the length byte can range from X'00' through X'FF' or from 0 to 255 bytes in decimal. Since a move cannot have a 0 length, the machine code length is interpreted to be the real length minus one byte. Thus, a length code of X'00' means move 1 byte; a length code of X'FF' means move 256 bytes. With the packed decimal instructions, the single length byte is divided in half, with a nibble for each length. Thus, for all packed decimal instructions either operand's length ranges from X'0' to X'F' or from 1 to 16 in decimal.

Pack Execution Errors

How can a PACK instruction error out? About the only common way is addressing exceptions (packing non existent bytes) and protection exception (packing into an area outside your program). Such errors may occur in later chapters when handling tables. Generally, PACK is nearly a failure proof instruction.

Summary of the Pack Instruction

***************************************************************************************************************  
* 1. The format of the operand2 is changed from character format to the packed decimal format and is placed in location of operand1.  
* 2. The fields are processed one byte at a time from the right to the left.  
* 3. Fields are not checked for valid sign or digit combinations.  
* 4. If the operand1 is too long, it is padded with high order zeros.  
* 5. If the operand1 is too small, truncation occurs and any remaining high order digits in the second operand is ignored.  
* 6. The maximum size of either operand is 16 bytes.*  
***************************************************************************************************************
The ADD Packed Decimal Instruction

Once the fields that are to be added have been converted into packed fields using the PACK instruction, mathematical operations can be done. Let's first examine the Add Packed instruction. The format is

\[
\text{AP} \quad \text{Operand1}, \text{Operand2}
\]

The Add Packed instruction is a SS (storage to storage) type instruction where the data in Operand2 is added to the data in Operand1 and the result of the addition is stored in Operand1. It is permissible for the data specified by Operand2 to be a literal (a self-defining constant). When doing this, the literal must be defined as packed data.

In the following instruction, a packed number 1 is added to a field called LINECT.

\[
\text{AP} \quad \text{LINECT}, =\text{P}'1'
\]

In addition, both operands can have an explicit length. It is extremely rare, however, to perform an add operation on only a part of a field. Further, the use of explicit lengths in an AP instruction can be dangerous. If the size of the DS of the field is changed, the explicit lengths must also be changed everywhere else. To forget is to invite numerous data exceptions that are extremely difficult to locate.

Operand1 must be large enough to hold the result of the add pack instruction. If not, truncation occurs and your answer is too small. The assembler does not check to see that the result field is not large enough, and no error message is issued. It is your responsibility to check to make sure the result field is large enough. Keep in mind the two problems that can arise when adding with the Add Pack instruction.

a. If the receiving field is too small to hold the results of an addition, truncation occurs. The maximum size length of both fields is sixteen bytes.

b. If either operand or both operands do not contain valid packed decimal data, a data exception occurs and the computer terminates the program.

The add operation proceeds from right to left, with the units, tens, hundreds, etc. being added in turn. The sign nibble in all fields is automatically converted to the preferred positive (C) or negative (D) sign. Notice the way the sign changes after execution in the following instruction

\[
\text{AP} \quad \text{TOTAL, UNITCOST}
\]

UNITCOST contains X'23 5F'
TOTAL contains X'00 00 0F'
TOTAL after contains X'00 23 5C'

When working with the Add Pack instruction you should remember that the rules used in manual addition apply to the Add Pack instruction.

\[
\text{field operation field = result}
\]
+ + + = +
- + - = -
+ + - = either + or - depending on magnitude of
- + + = the values in each field.

If we code
AP FIELD1, FIELD2
FIELD2 holds X'11 2C'
FIELD1 holds X'00 05 0C'
FIELD1 after X'00 16 2C'
or
FIELD2 before X'01 0D'
FIELD1 before X'00 11 1C'
FIELD1 after X'00 10 1C'

It is possible to perform elementary multiplication by repetitively adding a field to itself. Notice that after executing the following sequence, FIELD1 contains the same value as would occur had FIELD1 been multiplied by 4.
AP FIELD1, FIELD1 doubles the value in FIELD1
AP FIELD1, FIELD1 quadruples the value in FIELD1

Finally, if the length of Operand1 is too small to hold the result, an overflow condition occurs. This does not cause a program error; it merely sets the condition code to 3.

**Summary of the Add Pack Instruction**

* 1. The data in Operand2 is added to the data in Operand1 and the result is placed in Operand1.
* 2. Data in both operands must be in packed form.
* 3. The result that is generated is in packed form.
* 4. Addition executes algebraically right to left with Operand1 determining the size of the result.
* 5. If the resultant value is larger than can be held in Operand1, truncations occurs.
* 6. All signs and digits are checked for validity.
* 7. A positive result will have a "C" in the sign.
The UNPACK Operation

After a calculation is finished, the result field must be converted back into its character form. This is accomplished by the UNPACK instruction. The format is

```
<----------------
UNPK OPERAND1,OPERAND2
```

The unpack (UNPK) instruction is also a SS (storage to storage) instruction in which the contents of the packed field (Operand2) are unpacked into the character field (Operand1). Both fields have a maximum length of sixteen bytes.

You must make certain that Operand1 is large enough to hold the result or truncation occurs. Because of this, you should be extremely careful if you unpack a field into itself. Suppose you code the following instruction

```
UNPK CFIELD,PFIELD
```

The Unpack operates as follows
a. Puts in the sign—the sign nibble of the P-field is placed into the left nibble of the low order byte of the character field.
b. Fill out the rest of the character field's left nibbles by placing an F in the left nibble of each byte.
c. Insert the digits in the right nibble of each byte going from right to left.
d. Pad or truncate on the left of the character field as needed.

Both operands in the UNPK operation can use explicit lengths. It is more common to specify a length with the character field identified in Operand1. The following example can be useful when an output field has not been given a unique name in the description of the output record.

```
UNPK LINE+32(3),PQTY
```

This instruction says to unpack the data in the packed field PQTY into LINE beginning at the 33rd byte for a length of three bytes. In a later chapter, we show some uses for relative addressing
and explicit lengths used on Operand2, the P-field.

Examine the following examples of padding and truncation with the UNPK instruction.

<table>
<thead>
<tr>
<th>UNPK</th>
<th>Operand1</th>
<th>P-field</th>
<th>after unpacking</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTOTL,PTOTL</td>
<td>PTOTL</td>
<td>12 3D</td>
<td>F1 F2 D3</td>
</tr>
<tr>
<td>OQTY,PTOTQTY</td>
<td>PTOTQTY</td>
<td>12 3D</td>
<td>F0 F0 F1 F2 D3</td>
</tr>
<tr>
<td>GCOST,PCOST</td>
<td>PCOST</td>
<td>12 34 5C</td>
<td>F3 F4 C5</td>
</tr>
</tbody>
</table>

**Determining the Size of the C-Field for a Given P-Field**

The rule for determining the correct size of a character field from a given length of a packed field is

\[
\text{Length of C-field} = (2 \times \text{Length of P-field}) - 1
\]

Therefore, we find that to unpack all digits, we need

- PL1 ----> CL1
- PL2 ----> CL3
- PL3 ----> CL5
- PL5 ----> CL9

Notice that the length of the character field is always an odd number.

**Summary of the Unpack Instruction**

***************************************************************************
* 1. The data in Operand2 is changed from packed form to character form and the result is placed in Operand1. *
* 2. The data in the fields are processed right to left. *
* 3. The packed field's sign goes in the C-field's low order byte's left nibble. *
* 4. A hex "F" nibble is placed in the remaining left nibbles of the C-field. *
** 5. The digits of the P-field are inserted from right to left in the C-field. 
** 6. If Operand1 is too long, it is padded with F0's.
** 7. If Operand1 is too small, any remaining high order digits of the P-field are not unpacked and truncation occurs.

*******************************************************

Fixing up the Sign Byte

If we were to print the character field OTOTL after the following unpacking operation

```
UNPK OTOTL, TOTL
```

where TOTL contains X'12 3D', the OTOTL field would contain

```
OTOTL X'F1 F2 D3'
```

And the printed output would be 12L, since a X'3D' is an L in EBCDIC.

If the result of the unpack operation had generated an output with a zero digit such as OTOTL X'F1 F2 C0', there would be an invalid EBCDIC character in the rightmost byte and the output would print the characters 1 and 2 followed by a blank in the units position. Unprintable characters print as blanks. Of course, this is unacceptable. The sign nibble must be changed so that it reflects the true digit character. There are two techniques that can be used to solve this problem and correct the sign byte. They are the Or Logical Immediate instruction and the Move Zone.

OR Logical Immediate

The OR Logical Immediate is a Boolean binary logic instruction that changes the sign byte of the unpacked character field into the true character representation of that field. Boolean Or logic says that if either or both bits are true or on or 1, then the result is true or a 1. Consider these four possible cases

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>0</td>
<td>OR</td>
</tr>
<tr>
<td>1</td>
<td>OR</td>
<td>0</td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
</tbody>
</table>

```
1 1 1 0
```

The OR Logical Immediate instruction should OR the 8 bits of the byte containing the sign with one byte of immediate data. Example: the calculation result field OTOTAL contains X'F1 F2 F3 F4 C5'. This means that the sign byte contains a C5 and we want it to be an F5 so that
it can be printed. In order to do the OR, we need the bit configuration that represents the X'C5' and X'F5'. This is given in the Code Translation Table. Look at your Yellow Card and find the X'C5'. The binary representation is in the last column. You should find the following bit patterns

\[
\begin{align*}
X'C5' & \quad X'F5' \\
B'1100 0101' & \quad B'1111 0101'
\end{align*}
\]

Thus, we need to leave the right nibble alone and convert all bits of the left nibble into 1's. We could do the following

\[
\begin{align*}
1100 0101 & \quad \text{the X'C5'} \\
\text{OR } 1111 0000 & \quad \text{the OR mask value} \\
\hline
1111 0101 & \quad \text{which is F5}
\end{align*}
\]

This string, 1111 0000 (in binary), guarantees that the left nibble ends up an F while the right nibble remains untouched.

If we had a negative field, say DQUANT which contains X'F1 F2 D3', we would OR as follows

\[
\begin{align*}
1101 0011 & \quad \text{the X'D3'} \\
\text{OR } 1111 0000 & \quad \text{the OR mask} \\
\hline
1111 0011 & \quad \text{or X'F3'}
\end{align*}
\]

Thus, if we adopt, as a standard, the 1111 0000 as the value to OR, no matter what the sign is, the result should yield a printable number. That is, if there is a digit in the right nibble.

Notice that when the sign byte is fixed up, we lose the sign. Only when we edit the data (see chapter 13) can we salvage the sign as we unpack the digits.

Now that we know how the Boolean OR works, we will now see how this technique is implemented through the OI instruction.

<--------------------

\[
\begin{align*}
\text{OI 1 byte storage, 1 byte mask of immediate data with which to OR}
\end{align*}
\]

The one byte mask is ORed with one byte of storage and the result of the OR is in the one byte of storage. The mask can be shown as B'11100000' but this is too large and inconvenient. It is much easier to convert this value into an X'F0'.

The only problem encountered with the OI instruction must point to the sign byte. Relative addressing must be used. Therefore, in order to fix up the character field, we must code

\[
\begin{align*}
\text{OCOUNT DS CL4 containing X'F1 F2 F3 C4' to be fixed up} \\
\text{OCCOUNT+0 +1 +2 +3} \\
\text{OI OCCOUNT+3,X'F0'}
\end{align*}
\]

There is one common mistake made with the OI instruction, counting properly to get to
the sign byte. Suppose we have a character field as a part of an output line

... current contents of these fields

\[
\begin{align*}
\text{DS} & \quad \text{CL3} & 40 & 40 & 40 \\
\text{CFIELD} & \quad \text{CL5} & F1 & F2 & F3 & F4 & C5 \\
\text{DS} & \quad \text{CL10} & 40 & 40 & 40 & 40 & \ldots \\
\end{align*}
\]

We may assume that the line has been blanked out. Then we code

```
UNPK CFIELD, PFIELD
OI CFIELD+5, X'F0'
```

What happens? We fix up the first byte of the 10-byte filler. Thus, in storage we get

\[
\text{X'40 40 40 F1 F2 F3 F4 C5 F0 40 40'}
\]

which prints as 1234E0. Notice that the relative addressing is off by one byte, leaving the sign unfixed and it then fixed the sign on a filler byte converting the blank X'40' into X'F0'. The correct OI instruction should be coded

```
OI CFIELD+4, X'F0'
```

Remember that you start with zero as the first character position and count over to the end of the field. That is, it is an offset that's used. Or, you can take the defined length of the field and subtract one to get the correct relative address.

There are other uses for the OI instruction, but these are covered in later chapters.

The Move Zone Instruction

An alternate technique for changing the sign position character into a hex 'F' is to use the Move Zone instruction. All that happens is that a hex 'F' is moved from the left nibble of a valid numeric character field into the sign byte position of the unpacked field, thus replacing the bad sign nibble. The format for the Move Zone instruction is

```
<-----------------
MVZ Operand1, Operand2
```

The MOVE ZONE instruction copies left nibbles from the data in Operand2 into the left nibbles of Operand1 moving one left nibble at a time until it is finished. The length of the move is controlled by Operand1. The length of Operand1 can be either implicit or explicit. The MOVE ZONE can move a maximum of 256 left nibbles. Basically, the MVZ instruction follows the same rules of the MVC (Move Characters) instruction. Because of this, there are numerous ways of creating errors. The most common is the error of specifying the wrong length.

Look at the following example. Notice in field CFIELD that there are numerous left nibbles that contain the hex 'F'. In order to fix up CFIELD, we could move the hex 'F' from any of them.

```
CFIELD DS CL5 contains X' F1 F2 F3 F4 C5'
```

We could code any of the following
Chapter 11 Basic Packed Decimal Instructions

MVZ CFIELD+4 (1), CFIELD
MVZ CFIELD+4 (1), CFIELD+1
MVZ CFIELD+4 (1), CFIELD+2
MVZ CFIELD+4 (1), CFIELD+3

We must still point to the correct byte +4 in this case but MUST restrict the length to 1 byte. Otherwise it would alter the next four left nibbles beyond the sign since the length of CFIELD is 5.

What happens if we have two fields OQUANT and PQUANT that are defined as one byte fields.

OQUANT DS CL1
PQUANT DS PL1

and code

UNPK OQUANT, PQUANT

Notice that there is no left nibble containing a hex 'F' that can be moved. What must be done is to make one by using a literal for Operand2.

MVZ OQUANT, =C'1'

The C'1' creates a one byte field containing F1.

Given the choice of the two methods for fixing the sign, the OI is preferred because it has fewer rules. This means that there is less chance for error. Note, there are two other instructions that are similar to the MVZ. They are the MVN (Move Numerics), which is a right nibble mover, and the MVO (Move with Offset) in which bytes alternate. These instructions are used for rounding. However, the SRP Shift and Round Packed is much more convenient. These are discussed in more detail in chapter 12.

Coding Examples

These examples illustrate the relevant code that use the Pack, Add Pack, Unpack and Or Immediate instructions.

Example 1. Count the number of records in a file and print out the total.

GET ...
LOOP EQU *
AP TOTL, =P'1'
GET ...
B LOOP
ENDJOB EQU *
UNPK CTOTL, TOTL
OI CTOTL+2, X'F0'
PUT PRINT,...

... TOTL DC PL2 '0'
CTOTL DS CL3
Note that TOTL must have a starting value because it is being used as a counter, therefore it is defined using a DC with an initial value of zero (0).

Example 2. Develop the total cost for a file of items.

```
GET ... LOOP EQU *
PACK PCOST, COST
AP TOTCOST, PCOST
GET ... B LOOP
ENDJOB EQU *
UNPK CTOTL, TOTCOST
MVZ CTOTL+10(1), CTOTL
PUT ... RECORD DS 0CL100
... COST DS CL7
... TOTCOST DC PL6'0'
PCOST DS PL4
CTOTL DS CL11
```

The Subtract Packed Instruction

The Subtract Packed instruction is similar in operation to the Add Pack instruction. It is a SS (storage to storage) type instruction where the data in Operand2 is subtracted from the data in Operand1 and the result of the subtraction is placed in Operand1. Subtraction is algebraic taking into account the sign of both operands. If the length of Operand1 is too short to contain all significant digits of the difference, an overflow condition occurs and the condition code is set to 3. If Operand2 is shorter than operand1, normal subtraction occurs. The format of the subtract packed instruction is

```
<----------------
SP Operand1, Operand2
```

The data in both operands must be packed and can have a maximum length of 16 bytes. They can have either implicit or explicit lengths, although it is extremely rare to see explicit lengths used. The normal rules for subtraction are followed. Assume the following instruction and the three sets of values for each field before and after execution

```
SP ONHAND, AMTSOLD

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONHAND before</td>
<td>X'10 0C'</td>
<td>X'05 0C'</td>
</tr>
<tr>
<td>AMTSOLD before</td>
<td>X'05 0C'</td>
<td>X'05 0D'</td>
</tr>
<tr>
<td>ONHAND after</td>
<td>X'05 0C'</td>
<td>X'10 0C'</td>
</tr>
</tbody>
</table>
```
The Subtract Packed instruction provides a quick way to reset a field to zero. In order to do this, the field containing valid packed data is subtracted from itself.

```
SP COUNT, COUNT
```

This instruction is quite useful for resetting control fields, totals, and counters to zero. It cannot be used to initialize a field to zero since it must contain valid packed data for SP to work.

**The Compare Packed Instruction**

The Compare Packed instruction is used to determine whether the data held in one packed decimal field is greater than, less than, or equal to the data in another packed decimal field. The format is

```
<----------------
CP Operand1, Operand2
```

The contents of the field stored at the location designated by Operand2 are compared to the data stored at the location designated by Operand1. Operand1 controls the compare operation and a condition code is set depending on the algebraic relationship between the two fields. All fields are checked for validity. When the comparison is made, the shorter field is automatically padded with zeros so that an equal number of digits are compared.

- If Operand1 is > Operand2, the high condition code is set.
- If Operand1 is < Operand2, the low condition code is set.
- If Operand1 is = Operand2, the equal condition code is set.

In order for the compare to operate correctly, both operands must contain valid packed decimal data. In addition, Operand2 can be a self-defining packed decimal field. The maximum length for either operand is 16 bytes with the length being either implicit or explicit. Usually, however, explicit lengths are not coded. For example, it is possible to handle the page overflow condition as follows

```
LINECT DC PL2 '99'
...
CP LINECT, =P'50'
BL OK
BAL 5, HEADS
OK EQU *
...
```

Look at the following instruction and see the condition code that results from the Compare Packed operation

```
CP FIELDA, FIELDB
```

where the following data is in FIELDA and FIELDB

<table>
<thead>
<tr>
<th>FIELD A</th>
<th>FIELD B</th>
<th>CONDITION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 3C</td>
<td>00 00 00</td>
<td>12 3C</td>
<td>=</td>
</tr>
<tr>
<td>0C</td>
<td>12 3C</td>
<td>LOW</td>
<td></td>
</tr>
<tr>
<td>54 3D</td>
<td>00 1C</td>
<td>LOW</td>
<td></td>
</tr>
</tbody>
</table>
The following example illustrates the use of the various instructions covered in this chapter. It shows the coding (with some details omitted) that could be used for controlling page overflow and printing headings by counting lines.

```
START 0
GET ...
LOOP EQU *
    CP LINECT,=P'50'
    BL OK
    BAL 5,HEADRTN
OK EQU *
    ...
    PUT PRINT,LINE
    AP LINECT,=P'1'
    GET ...
    B LOOP
HEADRTN EQU *
    AP PAGENUM,=P'1'
    UNPK PAGECTR,PAGENUM
    OI PAGECTR+2,X'F0'
    PUT PRINT,HEAD
    PUT PRINT,COLHD
    MVI LINE,X'40'
    MVC LINE+1(132),LINE
    PUT PRINT,LINE
    SP LINECT,LINECT
    BR 5
    ...
LINECT DC PL2'99'
PAGENUM DC PL2'0'
LINE DS CL133
HEAD DS 0CL133
    DC CL100'1 report title here PAGE: '
PAGECTR DS CL3
    DC CL30' '
COLHD DC CL133'0 column headings here '
PRINT DTFPR CTLCHR=ASA, ...
```
## Packed Decimal Exercises

Complete the following exercise filling in the resulting values after execution of each instruction in the order shown. Use the most recent value in each variable.

| PACK P1,C1 | C1 = F1 F1 F1 | P1 = |
| PACK P2,C2 | C2 = F2 F2 D2 | P2 = |
| PACK P3,C3 | C3 = F3 F3 F3 | P3 = |
| AP P4,P1 | P4 = 00 36 50 | P4 after = |
| AP P5,P2 | P5 = 33 50 | P5 after = |
| AP P6,P3 | P6 = 01 45 10 | P6 after = |
| SP P5,P6 | P5 after = |
| AP P4,P4 | P4 after = |
| SP P6,P6 | P6 after = |
| AP P6,P82 | P6 after = |
| UNPK C4,P4 | C4 after = |
| UNPK C5,P5 | C5 after = |
| UNPK C6,P6 | C6 after = |
| DI C4+4,X F0 | C4 after = |
| MVZ C5+2(1),C5 | C5 after = |
| MVZ C6+4(1),C4+1 | C6 after = |