Explicit Use of Base Registers

In this lecture we consider the explicit use of the general–purpose registers as base registers. This leads to more observations on the structure of the machine language.

Consider the following line of code.

```
        MVC    PRINT+60(2),ASTERS
```

In this and similar examples, the field ASTERS is defined as “**”, two asterisks.

Question 1:	How many characters are moved?
Answer 1:	The explicit length field is set at 2, so two are moved.
Question 2:	To what addresses are they moved?
Answer 2:	The address of the first operand is PRINT + 60. One asterisk is moved to each of addresses PRINT + 60 and PRINT + 61.
Question 3:	What is the real address?
Answer:	Suppose that register 4 is used as a base register, having value X‘8002’ PRINT is at offset 1A, so it has value X‘801C’. Decimal 60 = X‘3C’, so PRINT + 60 is at offset X‘1A’ + X‘3C’ = X‘56’. The target address is X‘8002’ + X‘0056’ = X‘8058’.

Move Example: Second Form

Now consider the two lines of code that follow. What happens here?

```
        LA      8,PRINT+60
        MVC     0(2,8),ASTERS
```

The first instruction loads general–purpose register 8 with the address PRINT+60. Note that it is not the contents of that address that are loaded.

Remember that the LA instruction still uses register 4 as a base address.
We have just seen that 1) PRINT is at offset 1A, so it has address X‘801C’, and
2) PRINT+60 is at offset 56, having address X‘8058’.

General purpose register 8 is loaded with the value X‘8058’. It will serve as the explicit base register for the first operand in the MVC instruction.

Each of the arguments in MVC is really of the form Displacement (Length, Base) and references the address Contents(Base) + Displacement.

The displacement here is 0 (zero); 0(2,8) denotes the address 0 + Contents(8), or
0 + X‘8058’ = X‘8058’.

The 2 in (2,8) is the number of bytes to be moved.

Move Example: Third Form

Now consider the two lines of code that follow. What happens here?

```
        LA     8,PRINT
        MVC    60(2,8),ASTERS
```

The first instruction loads general–purpose register 8 with the address PRINT.
Note that it is not the contents of that address that are loaded.

Remember that the LA instruction still uses register 4 as a base address.
We have just seen that   1)   PRINT is at offset 1A, so it has address X‘801C’.

General purpose register 8 is loaded with the value X‘801C’. It will serve as the explicit base register for the first operand in the MVC instruction.

Each of the arguments in MVC is really of the form Displacement (Length, Base) and references the address Contents(Base) + Displacement.

The displacement here is 60, which in hexadecimal is X‘3C’.
60(2,8) denotes the address X‘3C’ + Contents(8), or X‘3C’ + X‘801C’ = X‘8058’.

The 2 in (2,8) is the number of bytes to be moved.

Explicit Addressing for Packed Instructions

The general form for packed instructions permits lengths for both operands 1 and 2.

There are a few issues to be remembered when using packed decimals.

1) The explicit lengths for the instructions must be in bytes, not decimal digits, and
2) Digits tend to be packed two per byte.

Consider the following two declarations from the data section of a program.

```
KGS      DC  PL3‘12.53’   Stored as 01253C.
POUNDS   DS  PL5
```

The following code sequence moves the value of KGS into POUNDS.

```
LA    6,KGS               Address of KGS
LA    8,POUNDS            Address of POUNDS
ZAP   0(5,8),0(3,6)
```

Tables and Arrays

The book discusses tables, which according to the author “contain a set of related data arranged so that each item can be referenced according to its location in the table”.

According to our author, there are two basic types of tables.

> “A **static table** contains defined data, such as income tax steps…. A **dynamic table** consists of a series of adjacent blank or zero fields defined to store or accumulate related data.”

We would call these structures “arrays”, with the static table corresponding to an array of constant values and the dynamic table corresponding to a plain array.

The main difference in the table structure, other than the older terminology, is that the values stored can be composite values. One might consider these tables as equivalent to an array of records or an array of structs.

In each of tables and arrays, the elements are addressed using an offset from the base address of the table or array.

Let’s now learn some of the older IBM terminology for tables.

Sample Table

Consider the following table, adapted from the textbook.

```
MONTAB    DC  C‘01’,‘JANUARY  ’
          DC  C‘02’,‘FEBRUARY ’
          ...
          DC  C‘09’,‘SEPTEMBER’
          DC  C‘10’,‘OCTOBER  ’
          DC  C‘11’,‘NOVEMBER ’
          DC  C‘12’,‘DECEMBER ’
```

In the terminology of the book, the first string (representing the month number) is called the **table argument**. The month name is called the **table function**.

The table is to be searched using a value that may match one of the table arguments. This value is called the **search argument**.

While one might think of this in terms of a database table, there is no requirement (other than good coding practice) that the table arguments be unique.

There are no keys to this table.

Sample Table Again

Here again is the sample table, showing only six of its twelve entries.

```
MONTAB    DC  C‘01’,‘JANUARY  ’
          DC  C‘02’,‘FEBRUARY ’
          ...
          DC  C‘09’,‘SEPTEMBER’
          DC  C‘10’,‘OCTOBER  ’
          DC  C‘11’,‘NOVEMBER ’
          DC  C‘12’,‘DECEMBER ’
```

Note that the following table is exactly equivalent.

```
MONTAB    DC  C‘01JANUARY  ’
          DC  C‘02FEBRUARY ’
          ...
          DC  C‘09SEPTEMBER’
          DC  C‘10OCTOBER  ’
          DC  C‘11NOVEMBER ’
          DC  C‘12DECEMBER ’
```

Note also that every table entry has exactly the same length (11 characters).
This is required by the table search algorithm.

Searching the Table by “Key Value”

Here is a fragment of code, written in a better style, that searches the above table.
The search argument, MONIN, is defined as two digits.

```
         USING *,4,5
         LA    8,MONTAB     Address of table in reg 8
         L     9,=H’12’     Number of entries in table
C10LOOP  CLC   MONIN,0(8)  Compare to table argument
         BE    C30EQUAL     Have a match
         BL    C20NOTEQ     Table is ordered; no hit.
         AH    8,=H‘11’     Move to next row
         BCT   9,C10LOOP    Decrement counter, branch if > 0
C20NOTEQ Do something
C30EQUAL Do something else.
```

Note that the search argument is being compared to the two characters at the addresses MONTAB, MONTAB + 11, MONTAB + 22, …, etc.

Note that this is a counted loop. It will search no more than twelve table entries.

Terminating the Loop

The textbook suggests the “high values” method of terminating the loop.

This is **extremely bad coding practice** and should be avoided.

Consider the table of packed data in Figure 11–8 on page 284 of the textbook.

The high value here is the five digit string ‘99999’.

Thus, one has **hard coded** an upper limit of 99,998 for the discounts. Orders of a million units cannot be discounted more than orders of 500 units.

Suppose that this table listed applicable discounts by account number.

The largest valid account number would be ‘99998’.

Suppose that an unsuspecting account manager assigned account number ‘99999’? This has happened. **[Ridiculous true story goes here.]**

Ordered and Unordered Tables

The textbook’s discussion of table searching implicitly assumes that the table arguments are sorted in increasing order by table entry.

Here are the book’s rules for comparing the search argument to each table argument.

1. The search argument is equal. The table function has been found and can be used.
2. The search argument is high. Continue the search unless this is the last entry in the table.
3. The search argument is low. Stop the search and take action appropriate to the type of table. If it is a table with steps, return with the step number.

For unordered tables, the basic comparisons are restricted to Equal or Not Equal.

As we know, ordered tables can be searched using **binary search**. This very efficient search technique is discussed in the textbook.

Linked Lists

It is possible to use the table structure to implement a linked list.

Every entry in the table contains an additional field specifying the offset in the table of the next item in the list.

The textbook's example (on page 292) is as good as any.

Offset	Part No.	Price	Next Offset
0000	0103	12.50	0036
0012	1720	08.95	0024
0024	1827	03.75	0000
0036	0120	13.80	0048
0048	0205	25.00	0012

Note that the order on the linked list is by part number.

The textbook illustrates the creation of a linked list. We may consider this example later; by developing an insert and delete method, as well as a more modern create method.

Direct Table Addressing

Consider a table in which the table arguments are sequential and consecutive.
One good example would be the table of months, already discussed.

```
MONTAB    DC  C‘01’,‘JANUARY  ’
          DC  C‘02’,‘FEBRUARY ’
          ...
          DC  C‘09’,‘SEPTEMBER’
          DC  C‘10’,‘OCTOBER  ’
          DC  C‘11’,‘NOVEMBER ’
          DC  C‘12’,‘DECEMBER ’
```

The table argument is implied by the position in the table. This could be written as:

```
MONTAB    DC  C‘JANUARY  ’
          DC  C‘FEBRUARY ’
          ...
          DC  C‘SEPTEMBER’
          DC  C‘OCTOBER  ’
          DC  C‘NOVEMBER ’
          DC  C‘DECEMBER ’
```

Entry K in the table is at offset (K – 1)•9

Calculation of the Direct Address

Let us define the following terms.

A(F)	is the address of the required function (table entry).
A(T)	is the address of the table.
SA	is the numeric value of the search argument with range 1 through table_length.
L	is the length (in bytes) of each function (table entry). All table entries have the same length.

The equation of interest is

$$A(F) = A(T) + (SA - 1) \bullet L$$

This is just the access formula for describing a singly dimensioned array in memory.