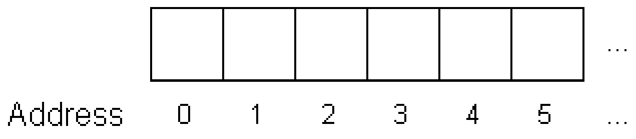
Base Displacement Addressing

A Visible/Z Lesson

The Idea:

Internal memory on an IBM mainframe is organized as a sequential collection of bytes. The bytes are numbered starting with 0 as pictured below,

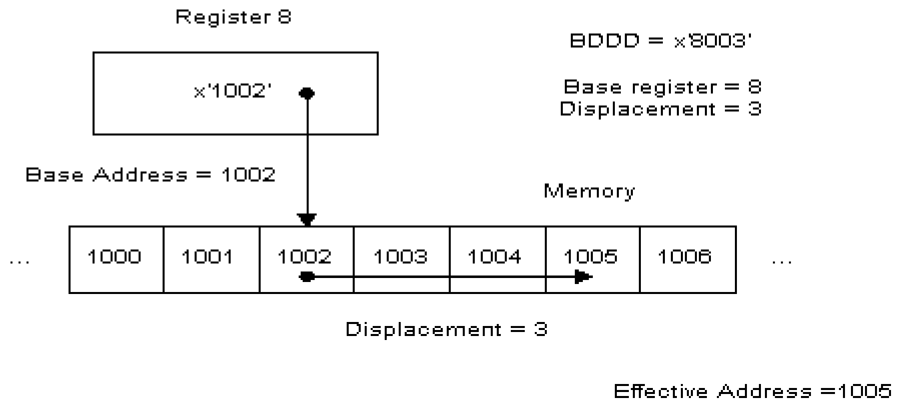


The IBM System/z architecture supports several addressing modes. The mode we focus on for beginning assembly language allows each address to be expressed as a collection of 31 consecutive bits. The smallest address would be represented by 31 consecutive 0's which denotes address 0. The largest address would be represented by 31 consecutive 1's whose value is $2^{31} - 1$ = 2,147,483,647. If we work with addresses in this form (let's call these direct addresses), each address occupies a fullword, or four bytes.

Why do we need an address in the first place? Consider the following instruction,

```
MVC   COSTOUT,COSTIN
```

In order for the machine to move the contents of COSTIN to COSTOUT, it must know the locations of these two fields. The computer identifies a field by the address of the first byte in the field. When an instruction is assembled, the assembler converts any symbols like "COSTOUT" to their corresponding addresses. The assembler does not, however, generate direct addresses, but produces an address in **base** / **displacement** format. Addresses in this form consist of 4 hexadecimal digits or two bytes, BDDD, where B represents a base register and DDD represents a displacement. The smallest displacement is x'000' = 0 and the largest displacement is x'FFF' = 4095. Ultimately, the base / displacement address must be converted to a direct address. So how does this occur? The diagram below indicates how this process occurs starting with the base/displacement address x'8003'.



Notice that the effective address, which is the direct address equivalent to the base/displacement address, is computed by adding the contents of the base register x'1002' and the specified displacement of x'003'. This produces an effective address of x'1005'.

Effective address = Contents(Base Register) + Displacement

There are two advantages of using base/displacement addresses instead of direct addresses in the object code that the assembler produces:

1) Every address is shorter. Instead of being 4 bytes long, the addresses are only 2 bytes, so all our programs are shorter.

2) The base/displacement addresses are correct no matter where the program is loaded in memory. Each symbol is represented by a displacement from a fixed point inside the program. If the program is relocated in memory, the displacement to a given variable does not change. The base register remains fixed as well. The main thing that changes when we relocate a program is the contents of the base register. This can be handled when the program runs. As a result, the base/displacement address is correct. On the other hand, if we had used direct addresses, every symbol would have a new address if the program were relocated.

Trying It Out:

1) Load the program **basedisplacement.obj** from the \Codes directory and single step through each instruction. Here is the original program with the object code listed on the left and the equivalent assembler instruction on the right:

```
0d c0                      BASR  R12,R0

                           USING *,R12

d2 03 c0 08 c0 0c          MVC   X,Y

07 fc                      BCR   15,R12

00 00 00 00          X     DS    F

ff ff ff ff          Y     DS    F
```

- The BASR loads the address of the next instruction (the address of the MVC since USING is an assembler directive and generates no code) into register 12. What address gets loaded? (Answer: the address of MVC.) The address that is loaded becomes the base address for the rest of the program, and R12 is established as the base register. The USING directive describes our wish to use register 12 as a base register, and associates R1 with the base address denoted by *. (Remember that * refers to the current value of the location counter in the assembler.)
- The MVC will move four bytes (x'03' + 1). The target address is x'c008' – an eight byte displacement off the contents of register 12. The source address is x'c00c' – a twelve byte displacement off the contents of register 12. Currently the target location contains x'0000' and the source contains x'ffffffff'.
- The BCR branches on condition 15 = x'1111'. In this case we branch under all conditions (equal, low, high, overflow) to the address in register 12. This returns execution to MVC.

1) Load and run **basedisplacement1.obj**. Notice that the program executes the same MVC instruction (d2 03 c0 10 c0 14) two times. How is it possible that the same instruction moves different fields during these executions? Answer: The program executes a

second BASR which changes the contents of base register 12, so the same object code references different areas of storage.

3) Modify the program so that it swaps the contents of X and Y. This will require another fullword and some more MVCs. Let X initially contain x'aaaaaaaa', let Y contain x'bbbbbbbb', and put x'00000000' in the third location. Write the object code that implements the swap. Step through it in VisibleZ. Check each MVC to make sure the target address is red and the source address is green. You can create your own VisibleZ object code programs as simple text files. Each byte is two characters (representing hex digits), separated by a space. You can put each statement on a separate line for readability or not. The program above would look like this:

```
0d c0

d2 03 c0 08 c0 0c

07 fc

00 00 00 00

ff ff ff ff
```

Save the file in the \Codes directory and run it. I save mine with a .obj extension.

Hint: Before you can completely write the code, you have to figure out the location of each of the variables. The assembler does this too! It creates a location counter and assigns a location to each instruction and each variable in the program.

When you are done, you should be able to cycle through the program several times watching the a's and b's swap places.

Answer:

Many answers are possible, but here's mine:

0d c0

d2 03 c0 20 c0 18

d2 03 c0 18 c0 1c

d2 03 c0 1c c0 20

07 fc

00 00 00 00

aa aa aa aa

bb bb bb bb