Relative Addressing

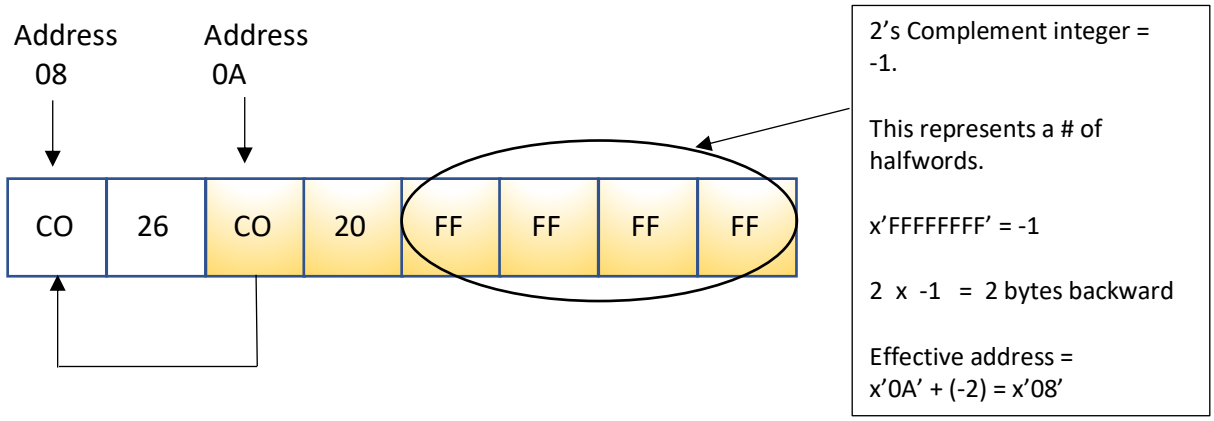
A VisibleZ Lesson

Base/displacement addressing was the original idea for addressing memory on the IBM 360. It was a beautiful solution that worked well for many years through several generations of the machine. It is still used extensively as an addressing technique. As programs grew larger over the years, programmers began to construct programs that needed more and more base registers to address storage. The limitations of base/displacement addressing soon became apparent.

If you work on older assembler programs, it's not uncommon to encounter a program that has maxed out the base registers it is using. Adding a field or even another instruction to the program may generate thousands of addressability errors when a program is reassembled. Working on these types of programs can be tricky.

Relative addressing was introduced in the ESA/390 architecture to help alleviate this problem. Instructions that use relative addresses don't need a base register at all. Instead of measuring everything from a base address forward, relative addresses are two's complement integers that represent forward or backward displacements from the beginning byte of the instruction.

The object code below occurs in the VisibleZ program **larl.obj**, and illustrates the idea. The yellow bytes represent the first **LARL** instruction, Load Address Relative Long, that occurs in the program. We are trying to load the address of the leftmost byte in the diagram, at address x'08'. The four rightmost bytes of the instruction comprise a 2's complement integer that has a value of -1. This number represents the number of halfwords from the beginning byte of the instruction to the address we wish to load. The machine multiplies this value by 2 and adds the result to the address of the first byte of the LARL instruction in order to compute the target address.



Notice that the object code for LARL does not use a base register at all.

Now use VisibleZ to load program **larl.obj** into memory. The program loads registers 2, 3, and 4 with addresses. Can you explain the values that are loaded into registers 3 and 4 by the second and third LARL instructions? You will have to perform some 2's complement arithmetic either by hand or with a calculator. It's tempting to think that using a calculator will be easier, but you might be surprised. Hex calculators treat the values you enter as positive, so a fullword hexadecimal x'FFFFFFFF', which represents -1 in 2's complement, evaluates to + 4294967295 in most hex calculators. What we need is a 2's complement calculator. 2's complement arithmetic is influenced by the number of bits we are using, so you have to indicate the number of bits in each computation.

You can find online 2's complement calculators, but the input they accept is usually in binary, so you may have to convert hex digits to binary before each calculation.

I'll help you with the calculation for register 3 and leave it to you to explain the register 4 calculation. The 2's complement integer inside the second LARL is x'FEDCBA98'. That's negative since the leading digit, F, has a 1 in the first bit. Converting to decimal we get a value of -19088744. This represents a number of halfwords, so we multiply by 2 and get -38177488. We add this to the address of the second LARL which is x'10' or 16 in decimal. This gives us -38177472. In 32 bit 2's complement, this is x'fdb97540'. This is the address that is left in register 3 after execution of the LARL. Obviously, this value is way off our address space and we expect most address calculations to produce a positive value, but the calculation is instructive.

Now execute the third LARL and explain the value in register 4.