LC-3 error "Instruction references label that cannot be represented in a 9-bit signed PC offset"

The problem is that the PC-Relative memory addressing mode (used in LD, LDI, LEA, ST, STI, BR) uses 9 bits to generate the two's complement offset, and so can only refer to labels that are less than 256 lines before or after the instruction (JSR also uses PC-Relative addressing mode, but it uses 11 bits, so a subroutine label can be up to 1024 lines before or after the SR call).

Sometimes just re-arranging the code can bring all labels back into range, but when that fails the only solution is to break the program up into multiple files (i.e. subroutines), and assemble them into separate object (.obj) files.

Normally, you would then build a single executable by linking these object files and having the linker resolve the various external references; unfortunately, the LC-3 does not come with a linker, so we have to do it manually. (Actually, I think this is a good thing, since you finally get to understand what a linker does!).

An example:

MAIN:

```
                .ORIG     x3000
; some code
                LD        R5, SUB1_AD      ;load the starting address of subroutine 1 into R5.
                JSRR      R5               ;now we can jump to the subroutine no matter where
                                           ;it is located in memory.
; some more code
                HALT

; data block
SUB1_AD         .FILL x4000                ;we have to manually supply subroutine1's address.
                                           ;normally the linker would resolve this for us.
                .END
```

SUB1:

```
                .ORIG     x4000             ;could be anywhere in memory.
;subroutine code
                RET
; data block if needed

                .END
```

MAIN and SUB1 are assembled separately, producing main.obj and sub1.obj

In the LC-3 simulator, load sub1.obj - this loads the code into memory starting at x4000, then loads the PC with the value x4000 (which is NOT what we actually want!).

Then load main.obj, which loads the code into memory starting at x3000, and loads the PC with the value we actually want, x3000

Then you can run the program exactly as though a linker had created the executable image.

Of course, if the subroutine needs to access the data block created with main (which is very likely), you have to supply the starting address of that data block to the subroutine as an "argument", i.e. in a register.

Suppose main's data block holds an array:

```
ARR1            .BLKW #100
```

Then in main, before calling the subroutine, we would load that address into a register:

```
                    LEA         R6, ARR1
```

The subroutine could then access the data in array1 using LDR and STR with R6 as the base register.

It is also possible to use these same techniques to create a separate data block - i.e. a file whose sole purpose is to set up storage space for data.