Project 3 Programming, Linking, and ("long") Jumps

Objectives: Loading from multiple files, linking labels across files, and jumping to addresses further than PC + a 9-bit offset.

1. Open 1.c.1.asm in an editor of your choice. Read over the code and comments to gain information about the intended purpose of the program.
2. Using the LC3 Instruction Set Summary Guide, read over the code and attempt to confirm correctness. It may also help to try to assemble and simulate the execution of the code.

Q.10 Were there any errors in the code? If so, describe them here and describe the correction.

Q.11 What was the final result of the programs computation and where was the result stored?

As you have likely observed, this program makes use of a procedure which begins at location label "neg2pos". Lets assume that we plan to add a few more code snippets (which will occupy about 300 address spaces) to this .asm file, and thus to simulate this occurrence.

3. Insert a .BLKW 300 pseudo-op to block out 300 words of address space just before neg2pos. (This pseudo-op has already been placed in the file … simply uncomment this line.)
4. Reassemble and simulate the code.

You should get a few errors here related to the increase in memory spaces between segments of code (one related to the load instruction and one related to the branch instruction). Read the error message(s).

5. Update the .asm file with your corrections noted above, assemble and simulate to confirm correctness. (See question and hints below.)

Q.12 Describe each error and your fix, in detail. List out the code corrections in your response. What happens and why? Explain your solution. See hints below.

Hint 1: Note a similar "load offset" issue was avoided in lines:

```
          LD    R2 FileStartPtr ; load starting address
```

Hint 2: Branch Offset issue. The target of the branch is too far, but how can we overcome this? Our branching capabilities are limited by a PC offset. However, the JMP command gets the 16-bit target address from a register. You need to

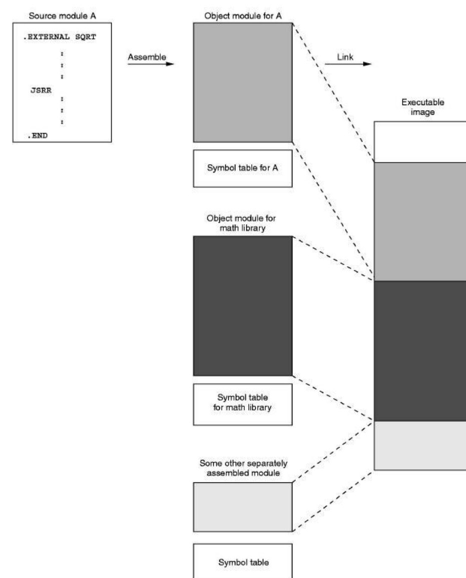
restructure the conditional to account for this change. Try the following design scheme

```
      ; in the preamble, Load a PTR to the subroutine
      ; in the preamble, Load a PTR to just after conditional … or
somewhere reasonable
      …
      ; Generate condition
      ; Branch (conditional)
            ; load appropriate register
            ; JMP to subtask1
      ; subtask2
      ; continue sequential execution

      … far far away …

      ;subtask1
      ;JMP back to continue sequential … or somewhere reasonable
```

(Note: in some instances, it is better to use JSSR and RET, … we will learn more about this in the near future.)

C.2 In your personal svn branch, commit the corrected versions of the 1.c.1.asm file.

Note that in this example, many procedures and data were contained in 1 .asm file. This makes the job of the loading the code to the LC3 memory fairly easy. However, it is often the case that there are many interrelated files that may be compiled and or assembled separately, but whose executions are interdependent. For example, you may have multiple files with many methods that call methods from other files. Without the proper management, this can lead to issues once the binary files are mapped to memory: including data fidelity issues, e.g. newly loaded code may clobber or overwrite some existing code, if not properly managed. Also, there is a concern of interoperability, referencing across multiple files. For example, in .asm file #1 you wish to refer to a procedure (by address pointer) contained in .asm file #2 … but how can we know where file #2 will be loaded in memory since the memory address label(s) are not defined in file #1…?



This issue is generally handled by a link editor or linker. This program will take the resulting assembler outputs such as multiple .obj files from multiple .asm file, and will “link” them to create one final executable image, .exe file, to be loaded by the loader.

In many assemblers there are pseudo-ops to help handle this issue by declaring address labels to be global (using .GLOBAL) and referring to address labels outside of a current .asm file using pseudo-op .EXTERNAL. See below example with label L2. During assembly, no errors are created for "undefined" labels which are declared to be .EXTERNAL. It is assumed these labels are defined in some external file.

```
.ORIG x0200

.EXTERNAL  L2

;;------- Preamble ----
    LD  R4 Data_ptr
    LD  R7 main_ptr
    JMP R7
Data_ptr: .FILL Data
main_ptr: .FILL main

main:
    BRnp L1
    LDR R7 R4 #0
    JSRR R7
L1: ADD R1 R1 R1

   .BLKW 300

Data: .FILL L2
END_ADDRESS:
.END
```

```
;;------------
;;-- g.asm
;;-- a function
;;------------
.ORIG x0000
.GLOBAL L2
L2: ADD R2 R2 R2
    JMP R7
END_ADDRESS:
.END
```

During the assembly of each .asm file, a relocation record is created for each EXTERNAL label used in that file. The appropriate address is then determined at link time, using .sym file data and information used when the single executable image. Thus a cohesive executable file (with all appropriate addresses replacing all labels) can be loaded.

With this in mind lets break up our previous example into two separate .asm files and attempt to link them correctly … NOTE: we will need to do this manually (we can't use pseudo-ops like .EXTERNAL and .GLOBAL) since PennSim does not have a linker!!!

1. Create two new files 1.c.2.asm and 1.c.3.asm such that 1.c.2.asm contains all of the code from 1.c.1.asm except the neg2pos procedure. Place the neg2pos procedure into 1.c.3.asm. Thus you have effectively split the original .asm file into two different .asm files.
2. Also, feel free to remove the .BLKW directive as we no longer need to simulate the need for a long jump.

Note we will need to play the role of the linker. Let's assume that we plan to load 1.c.2.obj into memory starting at x3000, and we plan to load the 1.c.3.obj into the memory just after it.

We have two link/load issues:

I. We have a label "neg2pos" in 1.c.3.asm, which is the address of a procedure; however, this label is not defined in 1.c.2.asm.
II. We plan to load 1.c.2.obj into memory starting at x3000, and we plan to load the 1.c.3.obj into the memory just after it; however, we do not know exactly what address to load 1.c.3.obj.

Lets begin to address issue I. You likely have a ptr to the procedure defined, e.g.

```
neg2posPtr .FILL neg2pos
```

However this will fail, since neg2pos is not defined here. We do not know the target address of this label at this time, so for now,

3. define this pointer to point to some arbitrary address so we can assemble the file without error

```
neg2posPtr .FILL x0000
```

Now, we should be able to assemble and load 1.c.2.obj. As a result we can determine the ending address, and thus, the beginning address to load 1.c.3.obj. (thus remedying issue II) To make this more clear,

4. Add the following label (as a placeholder) at the end of 1.c.2.asm

```
ENDADDRESS .END
```

5. Assemble 1.c.2.asm, load the .obj file into PennSim and open the .sym file using an editor.

Q.13 Using PennSim confirm starting load address of 1.c.2.obj. What is it? Using PennSim and the .sym file, what is the value of label ENDADDRESS?

We can now remedy issue II. We should load 1.c.3.obj at the address value of ENDADDRESS.

6. Remove line

```
ENDADDRESS .END
```

   from 1.c.2.asm (as this is no longer needed)

7. Change the .ORIG statement in 1.c.3.asm such that the file is loaded to the appropriate address.
8. Assemble and load both files and confirm that 1.c.3.obj is loaded to the correct location.

Q.14 What starting memory location is 1.c.3.obj loaded at? Is this correct? If not, adjust and explain.

Now we have enough information to rectify issue I. Observe that both files are now consecutively loaded, so address label neg2pos is known.

Q.15 What is the address value of label neg2pos?

9. Open 1.c.3.sym and find the value of neg2pos and change the value in 1.c.2.asm accordingly:
```
neg2posPtr .FILL ????
```

10. Re-assemble and load both files and confirm that you are, in fact, an expert linker and that the program now executes correctly.

C.3 In your personal svn branch, commit the corrected versions of 1.c.2.asm and 1.c.3.asm.