Tinker: A Tiny Processor That Can—Stage 1

Introduction

In this stage, you will implement the assembler. It will take a file written in assembly language (see the instruction manual) and produces an output file containing binary code.

The Assembler

The assembler reads files with the .tk extension. The mnemonics of the instructions are all detailed in the instruction manual. The syntax of the assembler is as follows:

- A line that starts with a tab character contains one instruction or a 64-bit data item.
- A line that starts with a semi colon contains comments. The assembler should ignore these comments.
- A line that starts with a colon character contains a name of an address. It could be a name to be used as a data store, or as a target for a jump.
- A line that starts with the ".code" directive indicates that all subsequent lines starting with a tab should be interpreted as containing instructions.
- A line that starts with the ".data" directive indicates that all subsequent lines starting with a tab should be interpreted as containing data items.
- There should be at least one ".code" directive in the file.

Example

```
.code
        add r0, r1, r3
        sub r0, r3, r4
        ld r5, :L1
        br r5
; the above two instructions are for illustration only
.data
:D0
        32
        3
.code
:L1
; the following instruction is actually a macro, it loads r6 with the address specified by :N0
; please observe that the content of memory at the address :D0 is not loaded into r6, just
; the address
        ld r6, :D0
        mov r7, (r6)(0)
        out r7
        halt
; This program should print 32
; end of the program
```

The assembler must guard against malformed instructions or syntax errors.

Goals

The purpose of this exercise is to

1. Gives you a substantial experience in programming in C
2. Tie up all the concepts that were discussed in class about assembly language and machine instructions.
3. Improve your skill in programming in general.

Project Stages

You should structure your code such that you go through two passes on the input file (the assembly code). In the first pass, you parse the input and identify the labels. Most important here is to expand the macros into assembly language. This first pass produces an intermediate file that consists of assembly language instructions with no macros (these should have been expanded already). You compute the addresses of the various labels using this intermediate file. Then, in a second pass, the assembler generates the machine instructions and stores them in a file (with ".tko" extension).

Example

```
.code
        clr r0
; the above line is a macro, it sets the value of register r0 to 0!
        ld r5, :L1
        br r5
; the above two instructions are for illustration only
.code
:L1
        halt
; the above line is also a macro!
```

The assembler should convert the .tk file to an intermediate form containing the following:

```
.code
        xor r0, r0, r0
        ld r5, 0x100c
        br r5
        trap 0x0
```

Discussion: Why 0x100c instead of L1:? Why xor r0, r0, r0 instead of clr r0? Why trap 0x0 instead of halt?
Then, from this intermediate form, simply go through the instructions and generate the code.

Software Engineering and Logistics

The project may appear intimidating but once you understand the overall picture, it is a very mechanical implementation effort. Tenets of software engineering such as modularity in design, defensive programming, regression testing, and a project plan will put some order and reduce stress. It is recommended that you put a project plan with deadlines so that you can scope the effort well. Adhoc and brute force work will not likely yield satisfactory result and may cause undue stress and anxiety. Starting early is *imperative.*

Happy coding.