STUMPEDASM – AN ASSEMBLER FOR STUMPED

You will be writing a simple two-pass assembler in Java for the **STumpEd** CPU. A two-pass assembler process the input file twice. On the first pass it determines the address of the labels and on the second pass it writes out the machine code. The details of this process are described below.

OVERVIEW

The **STumpEd** simulator takes in a simplified text-based object file and simulates the execution of a CPU. The object file must conform to the same file format that Logisim takes as a memory image file, and simply contains a list of numbers in hexadecimal format. There is a one line header which reads “v2.0 raw”, and after that each line will contain a single hexadecimal number that matches the bit width of the memory cell.

The conversion from human readable assembler mnemonics, such as PUSH and POP, to their machine readable op-codes given in the table below is called *assembly*. To write in assembly language means to write code an algorithm using the mnemonic operations of the CPU. An assembler converts this assembly language into machine language by translating each line of mnemonics into the appropriate machine code While there is much resistance to coding in assembly language, humans still overwhelmingly prefer to write code in assembler mnemonics and pseudo ops, called directives rather than writing directly in machine language.

As described below, assembly language is a line-based format. Each instruction is defined by a single line of text. This means we can process the input file on a per line basis. This is much simpler than free format languages where we must treat the program as a stream of tokens.

THE INPUT FORMAT AND STRUCTURE

Each line of an assembler consists of several parts. Notice that for each line, every single part is optional. This doesn’t mean that each part is always optional, for example, you can’t have a JMP instruction that doesn’t have an operand. Consequently, you will have to determine from context, how to process each line.

[label] [instruction mnemonic] [operand] [comment]

Each field in the input line is separated by whitespace. Labels in the input file are always terminated with a colon when they define a label. Thus, you can use this fact to determine whether a given input string at the start of a line is a label. When labels appear in the operand field, they are not terminated by a colon. Thus, a colon is not considered to be a part of the label name, but is used as a way to identify label definitions. Comments are always preceded by a semicolon, consequently, you can always use this fact to determine whether a string is a comment. Anything appearing after a semicolon is a part of a comment, hence, once you see a semicolon then you can consume the rest of the line. Both labels and comments can appear on lines by themselves, or with other fields. Instructions that have no operand, e.g., HALT, do not need anything in their operand field. Instructions can be assembler instructions, or what are known as *pseudo-ops*, or pseudo operations. For this assembler we only define one pseudo operation, DW. DW stands for *define word* and takes a single 16-bit operand that will be used to initialize the memory cell at that location.

You will want to carefully parse your input lines so that you properly define each of the fields.

In a two-pass assembler you will have to process the instructions twice. Consequently, you will find it easier to read the file once, storing each line, or an object that represents that line, in a Java collection. You then only need to iterate over that collection twice to execute the two passes.

THE ASSEMBLY PROCESS

Consider the following simple example. Each instruction is stored on a single line and consists of an instruction mnemonic, an operand, and, in this case a comment. Our goal then is to read this in and convert it to machine language, that is, a list of numbers that are CPU can recognize.

```
        PUSHI 225    ; push 225 onto the stack
        PUSHI 15     ; push 15 onto the stack
        ADD          ; add 225 to 15, push result onto stack
        POP k        ; store result at location k
        HALT         ; halt the simulator
k:      DW 0         ; reserve a word for location k and initialize to 0
```

This series of instructions would assemble to the following opcodes using the table provided below. There are some details here that are important. Note that the assembler knows that the label "k" is at location 005 in memory. How does this happen? In a two-pass assembler the first pass is about finding these labels and identifying their address. Since our CPU has only one word per instruction, this is relatively straightforward. Once we have identified the address, we store the symbol "k" along with its address 005 in a symbol table.

In the second pass of the assembler we convert each assembly instruction into the machine language that we wish to generate. Whenever we encounter a label, we look it up in the symbol table and use its associated address to compute the machine instruction. You can see below that the POP k instruction combines the op-code for POP, 3, with the address for k, 005.

Your assembler should have a -l option, for listing, that causes it to print out a table like you see below to standard err upon completion of the assembly of the input file. This table lists each label and its associated address as well as each assembler instruction along with its assembled machine code instruction. Your output must match the format below and include the address for each instruction. Printing out the full *listing* will help you debug your assembler. You can see at a glance whether you are creating the output that you expect.

```
*** LABEL LIST ***
k       005

*** MACHINE PROGRAM ***
000:10E1        PUSHI 225
001:100F        PUSHI 15
002:F000        ADD
003:3005        POP k
004:0F00        HALT
005:0000        k: 0
```

Your assembler should then write out an object file. This file will be in the Logisim V2.0 raw format shown below. Obviously, you do not include the "snip" lines. Your file must be then readable by your CPU simulator.

```
-------------------------snip-------------------------
v2.0 raw
10E1
100F
F000
3005
0F00
0000
-------------------------snip-------------------------
```

LABELS AND JUMPS

We need to discuss briefly the concept of a jump and how that impacts your assembler.

```
        JMP end         ; jump to the end of the program
        …               ; a bunch of other code
end:                    ; the target label
        HALT
```

Notice that this code does not have any data value associated with the “end” label. Since it does not have any data value, you do not want to reserve an address for this location. Keep this in mind while you are parsing your input file. Make sure that you increment your address count correctly. Only instructions and the DW pseudo-op will cause the counter to increase. Lines with just label definitions, or comments, will not increment the address.

COMMAND LINE INPUTS AND OUTPUT FORMAT

Your assembler will run from the command line using the Java interpreter. Your program must be saved in a single source file called Stasm.java. You may include multiple non-public classes in your file, but, your file must run with Java 11’s single source execution. You would then run it from the command line as follows:

```
        Your command line prompt> java Stasm.java <source file> <object file> [-l]
```

Your assembler will read in the source file name, assemble the source, and write out the object file to the specified name. For simplicity in this project, you must specify the full source and object file names. That is, your assembler should not attempt to make any assumptions about the extensions. If the user does not specify the required parameters then your code should print helpful usage as follows.

```
        Your command line prompt> java Stasm.java
        USAGE:  java Stasm.java <source file> <object file> [-l]
                        -l : print listing to standard error
```

SUBMISSION REQUIREMENTS

In your writeup include a brief discussion of your assembler In addition to writing the assembler in .java, you must use your assembler to assemble the test programs that you included with your simulator. For each assembler program you will want to include your source program as well as your listing (from -l) in your writeup.

You will be uploading a .pdf of your report and a .java file of your simulator.