Lab #2 Writing an Assembler — Due Date: 28 February 2021

How to Write a Two-Pass Assembler

All of you are familiar with writing programs in one or more assembly languages. But you may not have really considered how an assembler works when it translates an assembly language program into machine code. These are the functions that the assembler must perform when translating:

- Replace symbolic addresses with numeric addresses
- Replace symbolic operation codes with machine opcodes
- Reserve storage for instructions and data
- Translate constants into machine representation.

The assignment of numeric address can be performed without foreknowledge of what actual locations will be occupied by the assembled program. It is necessary only to generate addresses relative to the start of the code or data segment. Thus, we assume that we have separate data and code segments and that our assembler normally generates address starting a 0.

To illustrate, consider the following assembly code which is equivalent to the following pseudocode:

```
IF flag THEN answer := alpha + 2 * gamma / (C3P0 - R2D2) ENDIF
PRINT answer
```

```
 1    Section  .data
 2          flag:       word
 3          answer:     word
 4          alpha:      word
 5          gamma:      word
 6          C3PO:       word
 7          R2D2:       word
 8    Section     .code
 9          RVALUE      flag
10          GOFALSE     L0
11          LVALUE      answer
12          RVALUE      alpha
13          PUSH        2
14          RVALUE      gamma
15          MPY
16          RVALUE      C3P0
17          RVALUE      R2D2
18          SUB
19          DIV
20          ADD
21          STO
22          LABEL       L0
23          RVALUE      answer
24          PRINT
25          HALT
```

In translating line 9 of our example program, the resulting machine instruction would be assigned to address 0 of our code segment and occupy 4 bytes, if all machine instructions are 4 bytes (32 bits) long. Hence the instruction corresponding to line 10 would be assigned address 4. Similarly, the variable with the label **`flag`** would be at address 0 of our data segment, and occupy 4 bytes, if words are 32 bit entities.

Implementation Issues:

The assembler uses a counter to keep track of machine-language addresses. Because these addresses will ultimately specify locations in main stores, the counter is called the **location counter**. (“Address counter” would be a more accurate term.) Before assembling each section, the location counter is initialized to zero. After each source lien has been examined on the first pass, the location counter is incremented by the length of the machine-language code that will ultimately be generated to correspond to that source line. When a label definition is found, the assembler places both the label and its address (as determined by the value of the location counter) in a **symbol table**. Creation of the symbol table requires one pass over the source text. During a second pass, the assembler uses the address collected in the symbol table to perform the translation. As each symbolic address is encountered in the second pass, the corresponding numeric address is substituted for it in the object code.

Two types of logical errors can occur due to improper use of symbols. If a symbol appears in the operand field of some instruction, but nowhere in a label field, it is **undefined**. If a symbol appears in the ***label*** fields of more than one instruction, it is **multiply-defined**. The former error will be detected on the second pass, whereas the latter will be found on the first pass.

For an assembler, the symbol table structure is usually rather simple. It simply contains the symbolic labels, the appropriate address, and some simply type information (data or code, # data bits, etc.)

Assignment:

Your task is write an assembler for a hypothetical stack machine operating on 32 bit integers with the following instructions:

PUSH	*v*	-- push *v* (an integer constant) on the stack
RVALUE	*l*	-- push the contents of variable *l*
LVALUE	*l*	-- push the address of the variable *l*
POP		-- throw away the top value on the stack
STO		-- the rvalue on top of the stack is place in the lvalue below it and both are popped
COPY		-- push a copy of the top value on the stack
ADD		-- pop the top two values off the stack, add them, and push the result
SUB		-- pop the top two values off the stack, subtract them, and push the result
MPY		-- pop the top two values off the stack, multiply them, and push the result
DIV		-- pop the top two values off the stack, divide them, and push the result
MOD		-- pop the top two values off the stack, compute the modulus, and push the result
NEG		-- pop the top value off the stack, negate it, and push the result
NOT		-- pop the top value off the stack, invert all the bits, and push the result
OR		-- pop the top two values off the stack, compute the logical OR, and push the result
AND		-- pop the top two values off the stack, compute the logical AND, and push the result
EQ		-- pop the top two values off the stack, compare them, and push a 1 if they are equal, and a 0 if they are not
NE		-- pop the top two values off the stack, compare them, and push a 1 if they are not equal, and a 0 if they are equal
GT		-- pop the top two values off the stack, compare them, and push a 1 if the first operand is greater than the second, and a 0 if it is not
GE		-- pop the top two values off the stack, compare them, and push a 1 if the first operand is greater than or equal to the second, and a 0 if it is not
LT		-- pop the top two values off the stack, compare them, and push a 1 if the first operand is less than the second, and a 0 if it is not
LE		-- pop the top two values off the stack, compare them, and push a 1 if the first operand is less than or equal to the second, and a 0 if it is not
LABEL	*n*	-- serves as the target of jumps to *n*; has no other effect
GOTO	*n*	-- the next instruction is taken from statement with label *n*
GOFALSE	*n*	-- pop the top value; jump if it is zero
GOTRUE	*n*	-- pop the top value; jump if it is nonzero
PRINT		-- pop the top value off the stack and display it as a base 10 integer
READ		-- read a base 10 integer from the keyboard and push its value on the stack
GOSUB	*l*	-- push the current value of the program counter on the call stack and transfer control to the statement with label *l*
RET		-- pop the top value off the call stack and store it in the program counter
HALT		-- stop execution

For two operand instructions, the operand on top of the stack is the second operand, and the one immediately below it is the first operand

The numeric opcodes are as follows:

HALT	**0**	**DIV**	**10**	**LT**	**20**
PUSH	**1**	**MOD**	**11**	**LE**	**21**
RVALUE	**2**	**NEG**	**12**	**LABEL**	**22**
LVALUE	**3**	**NOT**	**13**	**GOTO**	**23**
POP	**4**	**OR**	**14**	**GOFALSE**	**24**
STO	**5**	**AND**	**15**	**GOTRUE**	**25**
COPY	**6**	**EQ**	**16**	**PRINT**	**26**
ADD	**7**	**NE**	**17**	**READ**	**27**
SUB	**8**	**GT**	**18**	**GOSUB**	**28**
MPY	**9**	**GE**	**19**	**RET**	**29**

Write an assembler with the following specifications:

All instructions for this machine are 32 bits (4 bytes) long, with the following format: Bits 32-21 are ignored (in practice, filled with zeros,) bits 20-16 hold the opcode, and bits 15-0 hold the operand. (If there is no operand, those bits are filled with zeroes, but otherwise ignored.) Your assembler should produce a binary file which is a set of machine instructions in Big-Endian format. We will use the Harvard memory model, with two separate 256KB (65,536 32-bit words) for instructions and data, and you will have two location counters (one for code, and one for data) rather than one. This also means that, any label with a numeric value greater than 65535 would be an error, although a highly unlikely one to encounter in a debugged assembler. To keep things simple, we will assume that the memory is word-addressable rather than byte-addressable. Thus, the symbol table would conceptually look like:

Lexeme	Type	Address
flag	**Int**	**0**
answer	**Int**	**1**
alpha	**Int**	**2**
gamma	**Int**	**3**
C3P0	**Int**	**4**
R2D2	**Int**	**5**
L0	**Code**	**13**

Note that implication of having word addressability means that the location counter will be incremented by 1 rather than by 4 at each step.

Assume labels are sequences of alphanumeric characters only, starting with an alphabetic character. The maximum length of a label can be up to you, either fixed or unlimited, but your implementation must allow labels to be at least 12 characters in length. Since the only data type the assembler recognizes is a 32-bit integer, **word** is the only data declaration directive your assembler will need to be able to handle.

You instructor will provide you with a virtual machine for this abstract stack machine. (In other words, he will do the hard part of actually executing the binary code.)

You may use Java, C, C++ , or Python 3 to implement your assembler, but it must generate the appropriate binary code in Big Endian format. If you use Java, you may use any platform you wish. If you use any other language, you must either provide a makefile or complete compilation instructions for generating an executable under Linux