LAB #2 – PART I: Simulator Lab

Using the machine definition and instruction set descriptions given in Beck, design, code, and test program to simulate the execution of the abstract machine.

Object Files

Your input will be a file of records as follows:

1. A Header Record
   - record position 1: H
   - record positions 2-7: a 6 character segment name
   - record positions 8-11: a 4 Hex character value denoting the initial program load address
   - record positions 12-15: a 4 Hex character value denoting the length of the segment
2. Text Records
   - record position 1: T
   - record positions 2-5: a 4 Hex character address at which the information is to be stored
   - record positions 6-9: Initial value of that address, as a 4 Hex character string
3. End Record
   - record position 1: E
   - record positions 2-5: a 4 Hex character address at which execution is to begin

Figure 1 shows an example input file. Like all (correct) input files, it starts with a single header record, followed by a sequence of text records, and finishes with a single end record. Do not assume that initial contents will be given to you for each of the memory addresses in the segment. Thus, the length of the segment given in the header record can be larger than the number of text records in the input file (as in the sample above).

Requirements

Your program should have three major components: one (the \loader") to put the input information into the data structures that represent the memory and registers of the machine, one (the \interpreter") to simulate the operation of the machine as it executes each instruction, and one (the \simulator") to act as the simulator's user interface, displaying the state of the machine as appropriate and controlling execution.

The loader must detect and provide appropriate messages under the following error condition:

- invalid contents (i.e., illegal characters in an input record)
- other errors you identify

The instruction interpreter must be capable of executing each of the machine's instructions as specified. The interpreter should not be capable of stopping execution, unless the HALT instruction is issued. The simulator must halt instruction interpretation when a user-specified time limit has been exceeded. This time limit should be specified in terms of number of instructions executed. Some reasonable default for this time limit should be provided. In addition, you may also select to have some I/O errors be fatal.

```
HSAMPLE30000104
T3000E300
T300256E0
T300454A0
T300614A4
T3008040A
T300A6840
T300C16C4
T300E1261
T301014BF
T30120E04
T30145020
T30161003
T3018F031
T301AF025
T31000001
T31010001
T31020001
T31030001
E3000
```

Figure 1: Example Input File

Your simulator should have at least 3 modes: quiet, trace, and step. In quiet mode, it simulates the execution of the loaded program without interruption (unless the time limit is exceeded, of course).

In trace mode, it should generate a trace of execution including:

1. The state of the machine (memory page and registers) immediately after loading but before execution.
2. Each executed instruction, including the memory locations and registers affected or used.
3. The state of the machine (memory page and registers) after execution.

Step mode should be similar to trace mode, except that the user is prompted before each step is taken. You may add other modes as you feel necessary.

LAB #2 – PART II: Assembler Lab

In this lab write an assembler for the abstract machine.

1. Input

The input to the assembler is a file containing an assembly language program. Each line of the file will have the following format:

Record Position	**Meaning**
1-6	Label, if any, left justified
7-9	Unused
10-14	Operation field
15-17	Unused
18-end of record	Operands and comments (Comments begin with a semicolon (;)
Exception:	A semicolon (;) in the first record position indicates that the entire record is a comment.

Notes:

1. This (somewhat rigid) record structure is intended as a convenience for your parsing of program lines. It allows you to treat each line in the file as an array, with particular ranges of the array containing certain fields.
2. Despite the previous comment, you may allow slightly different syntax for input programs, if you find that easier. For example, you may allow the fields to be separated by tabs rather than spaces. You may also impose an upper limit on the length of input lines.
3. For the purposes of this assignment, you may assume a maximum of 100 symbols, 50 literals, and 200 source records in any given program. However, these constraints should be easy to change

Labels

Labels may be up to 6 alphanumeric characters (e.g., they may not include blanks). The first character of a label must be alphabetic, but may not be an “R" or an “x".

Alphabetic characters can be upper or lowercase and an upper case character should be treated differently from its corresponding lower case character.

Instructions

Give details of architecture of your abstract machine.

You may require instructions to be all uppercase. Prepare the list of instructions supported by your assembler along with its format & machine code.

Operands and Comments

In the “operands and comments" field, you may prohibit the “operand" part from including blanks.

Registers are indicated by their explicit name. Constants are written either in hexadecimal notation (preceded by a lowercase "x") or as decimal integers, which may be positive or negatives (preceded by "#").

An imm5 operand must be in the range #-16…#15, or x0…x1F. An address operand must be in the range #0… .#65535, or x0…xFFFF. Note that only the least significant 9 bits of this value are used in the machine code encoding.

Symbols can be used in place of any operand. Symbols can also be used in place of immediate arguments. In this case, the (absolute) symbol must have a value in the appropriate range. The address operand can contain a literal. A literal is denoted by an "=", followed by a constant. A literal causes the assembler to generate a location for the literal, place the value indicated by the operand in that location, and use the address of that location in the instruction.

Pseudo Ops

List the Psuedo Ops your assembler should handle.

2. Output

Your assembler should have two primary outputs: an object file (which will subsequently be the input file for the linker/loader you will write in Lab 3), and a listing for the user.

Your object file should provide all of the information needed by a linker & loader to generate the input to the machine simulator defined in Lab 1. It should include a header record, text records, an end record, and other record types as appropriate. That is, your object file should provide the memory contents associated with the instructions and data of the source program, along with information for the loader concerning the relocatability of the object file information and the size of the program's address space. Don't forget to deal with the location at which execution is to begin. For acceptance of this lab, the object file should be written to a file. Remember that this file will be "consumed" by the program you write in the next lab. The listing you output for the user should contain the source program and its assembly, in some suitable format. It does not need to include comments. Remember that this output is for human consumption and should be designed to be as useful as possible to programmers. A recommended format is as follows.

(Address)	Contents In Hex	Contents in Binary	(line #)	Label	Instruction	Operands

Your assembler should print meaningful diagnostics if errors in assembly are encountered. It should be capable of detecting errors involving each of the following conditions: invalid operation, invalid label, invalid operand (symbol, literal, integer, and register), undefined reference, and multiple definition of a symbol.

Pseudocode for Two-Pass Assembler

1. The following pseudocode IS NOT COMPLETE. There are several holes you need to fill in for it to be correct. It is, however, a good overview of the algorithm for the assembler. You should understand it thoroughly.
2. This is not the only way to do things. There are many equally good (and some better!) solutions. But before developing your solutions you need to understand the task at hand.

Data Structures

- Machine Op Table
- Pseudo Op Table
- Location Counter
- Symbol Table
- Literal Table

Pass 1

```
begin
        read in first non-comment line of source
        if (not an .ORIG line)
                return error
        endif
        store segment information
        initialize Location Counter
        read in next line of input
        while (not an .END line)
                if (line is not a comment line)
                        if (there is a label)
                                add symbol to Symbol Table
                                - report an error if it's already there
                                - otherwise set its value (using Location Counter or operand of EQU)
                        endif
                        if (instruction in Machine Op Table or Pseudo Op Table)
                                increment Location Counter appropriately
                        else
                                error (invalid operation)
                        endif
                        if (operand contains a literal)
                                add literal to Literal Table:
                                - avoid duplicate entries
                                - set the name and value
                        endif
                        write line to intermediate file
                endif
                read in next line of input
        end while
        set addresses of literals
        use Location Counter to calculate program length
end
```

Pass 2

```
begin
        output header record to object file
        read in first line from intermediate file
        while (not done with intermediate file)
                if (there are symbols in the operands)
                        replace symbols with their value from Symbol Table
                        - report an error if not present
                endif
                if (there are literals in the operands)
                        replace literal with its address from Literal Table
                endif
                assemble line
                output line (text record) to object file
                output line to listing file
                read in next line from intermediate file
        end while
end
```

LAB #3: Linker/Loader Lab

Part 1 (Modifying Assembler)

Modify the assembler you built in Lab 2 so that it is capable of dealing with external symbols.
Specifically, your assembler should be capable of processing the following two pseudo-ops, in addition to the ones described in Lab 2.

Mnemonic	Meaning
.ENT	[ENTry name] The operand field of this pseudo-op is a list of symbols that must be defined in the current segment, and are permitted to be referenced by other segments.
.EXT	[EXTernal name] The operand field is a list of symbols that may be referenced legitimately by this segment, but are not defined in this segment. The symbols must be defined in some other segment.

For both the ENT and EXT pseudo ops, there may not be a label, and there must be an operand. You may, at your option, require that the list of symbols be bounded by some small number, such as four. Appropriate error messages should be written if there are violations of the syntax rules for the ENT and EXT pseudo-ops. The object file produced by your modified assembler is up to you, but it must contain all the essential information (e.g., text, relocation information, length, start address for execution) as well as appropriate information about external symbols, so the loader can perform any linking needed. Submit separate sheet describing the format of this file thoroughly.

Part 2 (Linking Loader)

Write a (relocating) linking loader for abstract machine. Your system must be capable of handling multiple source files, and linking them together in order to form the executable file that is input to the simulator. If multiple files are linked, they are all required to be relocatable. The format of the input file is the same as the output file produced by modified assembler.

The output of the loader should be an absolute executable file, ready to be passed to the simulator which will interpret the instructions. The format of the output file should be exactly that described in Lab 1 as input.

Part 3 (Integration)

Integrate the assembler (produced in part 1 of Lab 3), the loader (produced in part 2 of Lab 3), and the simulator that you wrote in Lab 1, to produce a system capable of processing a (correct) source program written in the assembly language from translation through linking, loading, and execution.
Also submit the documentation for lab 3 which should include a brief appendix which gives instructions for using the three components in tandem.

Reference for Object file format: Portable Formats Specification, Version 1.1(Tool Interface Standards)