Computer Architecture - Fall 2014
Lab 1

Introduction

The purpose of this lab is to reinforce the concepts of assembly language and assemblers. In this lab assignment, you will write an LC-3b Assembler, whose job will be to translate assembly language source code into the machine language (ISA) of the LC-3b. You will also write a program to solve a problem in the LC-3b Assembly Language.

In Lab Assignments 2 and 3, you will close the loop by completing the design of two types of simulators for the LC-3b, and test your assembler by having the simulators execute the program you wrote and assembled in this lab.

Part I: Write an assembler for the LC-3b Assembly Language

The general format of a line of assembly code, which will be the input to your assembler, is as follows:

```
label opcode operands ; comments
```

The leftmost field on a line will be the label field. Valid labels consist of one to twenty alphanumeric characters (i.e., a letter of the alphabet, or a decimal digit), starting with a letter of the alphabet. A valid label cannot be the same as an opcode or a pseudo-op. A valid label must start with a character other than 'x' and consist solely of alphanumeric characters – a to z, 0 to 9. The label is optional, i.e., an assembly language instruction can leave out the label. **A valid label cannot be IN, OUT, GETC, or PUTS. The entire assembly process, including labels, is case-insensitive.** A label is necessary if the program is to branch to that instruction or if the location contains data that is to be addressed explicitly.
The opcode field can be any one of the following instructions:

```
ADD, AND, BR, HALT, JMP, JSR, JSRR, LDB, LDW, LEA, NOP,
NOT, RET, LSHF, RSHFL, RSHFA, RTI, STB, STW, TRAP, XOR
```

The number of operands depends on the operation being performed. It can consist of register names, labels, or constants (immediates). If a hexadecimal constant is used, it must be prefixed with the 'x' character. Similarly, decimal constants must be prefixed with a '#' character.
Optionally, an instruction can be commented, which is good style if the comment contains

meaningful information. Comments follow the semicolon and are not interpreted by the Assembler. Note that the semicolon prefaces the comment, and a newline ends the comment. Other delimiters are not allowed.

In this lab assignment, the `NOP` instruction translates into the machine language instruction `0x0000`.

Note that you should also implement the `HALT` instruction as `TRAP x25`. Other `TRAP` commands (`GETC`, `IN`, `OUT`, `PUTS`) need not be recognized by your assembler for this assignment.

In addition to LC-3b instructions, an assembly language also contains pseudo-ops, sometimes called macro directives. These are messages from the programmer to the assembler that assist the assembler in performing the translation process. In the case of our LC-3b Assembly Language, we will only require three pseudo-ops to make our lives easier: `.ORIG`, `.END`, and `.FILL`.

An assembly language program will consist of some number of assembly language instructions, delimited by `.ORIG` and `.END`. The pseudo-op `.END` is a message to the assembler designating the end of the assembly language source program. The `.ORIG` pseudo-op provides two functions: it designates the start of the source program, and it specifies the location of the first instruction in the object module to be produced. For example, `.ORIG N` means "the next instruction will be assigned to location N." The pseudo-op `.FILL W` assigns the value W to the corresponding location in the object module. W is regarded as a word (16-bit value) by the `.FILL` pseudo-op.

The task of the assembler is that of line-by-line translation. The input is an assembly language file, and the output is an object (ISA) file (consisting of hexadecimal digits). To make it a little more concrete, here is a sample assembly language program:

```
;This program counts from 10 to 0
       .ORIG x3000
       LEA R0, TEN          ;This instruction will be loaded into memory location x3000
       LDW R1, R0, #0
START  ADD R1, R1, #-1
       BRZ DONE
       BR START
                            ;blank line
DONE   TRAP x25             ;The last executable instruction
TEN    .FILL x000A          ;This is 10 in 2's comp, hexadecimal
```

```
        .END                 ;The pseudo-op, delimiting the source program
```

And its corresponding ISA program:

```
        0x3000
        0xE005
        0x6200
        0x127F
        0x0401
        0x0FFD
        0xF025
        0x000A
```

Note that each line of the output is a four digit hex number, prefixed with "0x", representing the 16-bit machine instruction. The reason that your output should be prefixed with "0x" is because the simulator for Lab Assignment 2 that you will write in C expects the input data to be expressed in hex, and C syntax requires hex data to start with "0x". Also note that BR instruction is assembled as the unconditional branch, BRnzp.

When this program is loaded into the simulator, the instruction 0xE005 will be loaded into the memory location specified by the first line of the program, which is x3000. As instructions consist of two bytes, the second instruction,0x6200, will be loaded into memory location x3002. Thus, memory locations x3000 to x300C will contain the program.

We have included below another example of an assembly language program, and the result of the assembly process. In this case, the .ORIG pseudo-op tells the assembler to place the program at memory address #4096.

```
        .ORIG #4096
A       LEA R1, Y
        LDW R1, R1, #0
        LDW R1, R1, #0
        ADD R1, R1, R1
        ADD R1, R1, x-10     ;x-10 is the negative of x10
        BRN A
        HALT
Y       .FILL #263
        .FILL #13
        .FILL #6
        .END
```

would be assembled into the following:

```
0x1000
0xE206
0x6240
0x6240
0x1241
0x1270
0x09FA
0xF025
0x0107
0x000D
0x0006
```

Important note: even though this program will assemble correctly, it may not do anything useful.

The Assembly Process

Your assembler should make two passes of the input file. In the first pass, all the labels should be bound to specific memory addresses. You create a symbol table to contain those bindings. Whenever a new instruction label is encountered in the input file, it is assigned to the current memory address.

The second pass performs the translation from assembly language to machine language, one line at a time. It is during this pass that the output file should be generated.

You should write your program to take two command-line arguments. The first argument is the name of a file that contains a program written in LC-3b assembly language, which will be the input to your program. The second argument is the name of the file to which your program will write its output. In other words, this is the name of the file which will contain the LC-3b machine code corresponding to the input assembly language file. For example, we should be able to run your assembler with the following command-line input:

```
assemble <source.asm> <output.obj>
```

where **`assemble`** is the name of the executable file corresponding to your compiled and linked program; **`source.asm`** is the input assembly language file, and **`output.obj`**; is the output file that will contain the assembled code.

You will need to include some basic error checking within your assembler to handle improperly constructed assembly language programs. Your assembler must detect three types of errors and must return three different error codes. The errors to be detected are *undefined labels* (error code 1), *invalid opcodes* (error code 2), and *invalid constants* (error code 3). An invalid constant is a constant that is too large to be assembled into an LC-3b instruction. If the `.ORIG` pseudo-op contains an address that is greater than an address that can be represented in the 16-bit address space, your program should return error code 3. Also, if the `.ORIG` statement specifies an address that is not word-aligned, your program should return error code 3. Your program must return the error codes via the `exit(n)` function, where n denotes the error code number. If the assembly language program does not contain any errors, you must exit with error code 0. Exiting with the correct codes is very important since they will be used in the grading process. On Linux, you can determine the exit code of your assembler by executing **`echo $?`** right after running the assembler.

This error checking is the bare minimum that we expect. You can return error code 4 for any other errors you find. Just be sure that the errors don't fall within the first three categories specified above.

Examples of error codes

Error code 1: undefined label

A label is used by an instruction but the label is not in the symbol table, e.g.

- ```
  .ORIG x3000
      LEA R0, DATA1     ; DATA1 is not defined in the assembly code
      .END
  ```
- ```
  .ORIG x3000
      JSR ADD    ; JSR is parsed as an opcode, and ADD is an
                 ; illegal label. While illegal labels
                 ; should return error code 4, we accept
                 ; error code 1 for this case, too.
      .END
  ```

Error code 2: invalid opcode

An invalid opcode is one that is not defined in the LC-3b ISA, e.g.

- ```
  .ORIG x1000
  MUL R0, R1, R2
  .END
  ```
- ```
  .ORIG x1000
  ABC
  .END
  ```

Error code 3: invalid constant

An invalid constant is a constant that is too large to be assembled into an LC-3b instruction. An odd constant that follows `.ORIG` is also an invalid constant.

- ```
  .ORIG x1000
  ADD R0, R1, #20      ; error
  .END
  ```
- ```
  .ORIG x1001          ; error
  ADD R0, R1, #1
  ```

```
.END
```

Error code 4: other error

These errors which do not belong to any of the above categories.
Examples:

- ```
  .ORIG x1000
  ADD R0, R1    ; wrong number of operands
  .END
  ```
- ```
  .ORIG x1000
  .FILL         ; missing operand
  .END
  ```
- ```
  .ORIG x1000
  ADD R1, #2, R3       ; unexpected operand
  .END
  ```
- ```
  .ORIG x1000
  ADD R9, R0, #1       ; R9 is an invalid register number
  .END
  ```
- ```
  .ORIG x1000
  ADD R1, R0, 1 ; 1 is an invalid operand (neither a register nor an immediate)
  .END
  ```

If a label and an instruction that uses it are too far apart and the offset cannot be specified properly in the machine code, you should produce error code 4.

Your assembler should accept an “empty” program, i.e. one with just a valid `.ORIG` and a `.END`. E.g. the following assembly program would be assembled to only one line containing the starting address (`0x3000`).

```
.ORIG x3000
.END
```

Note: your assembler needs to recognize only labels as operands for `LEA`, `BR`, and `JSR` instructions. For example, if the following line is in an input assembly language program, your assembler can exit with error code 4:

```
LEA R1, x100
```

Part II: Write a program to solve the following problem

Write an LC-3b assembly language program that converts a string assembled on an LC-3 assembler to a string formatted for an LC-3b.

As you may recall, the LC-3 is a 16 bit addressable machine. The convention used for storing strings in an LC-3 is to store one zero-extended ASCII character per memory location. For
```

example if I wanted to store the string “Hello!” in memory starting at location x5000 on an LC-3, memory would look as follows:

x5000	x0048	
x5001	x0065	
x5002	x006c	
x5003	x006c	
x5004	x006f	
x5005	x0021	
x5006	x0000	(null terminating string)

The LC-3b is byte addressable, so if I were to byte-by-byte copy the characters from the LC-3’s memory to an LC-3b, the contents of memory would look as follows:

x5000	x48	
x5001	x00	
x5002	x65	
x5003	x00	
x5004	x6c	
x5005	x00	
x5006	x6c	
x5007	x00	
x5008	x6f	
x5009	x00	
x500a	x21	
x500b	x00	
x500c	x00	(first half of null terminator)
x500d	x00	(second half of null terminator)

Something to think about: Why is the value x48 stored in location x3000 (as opposed to x00 being stored in x3000 and x48 being stored in x3001)?

This format, however, differs from what is expected on an LC-3b and as a result this string would be unreadable to already written LC-3b programs.

Your job: Write an LC-3b assembly language program that converts a null-terminated character string assembled for an LC-3 to a null-terminated character string assembled for an LC-3b. Each element in the character string is a zero-extended byte value containing the ASCII code of a character (or the null termination). Your output will be the same string, but without the zero extensions. For example if your program had been run on the above example, the output would be:

x5000	x48	
x5001	x65	
x5002	x6c	
x5003	x6c	
x5004	x6f	
x5005	x21	
x5006	x00	(only one byte of null termination)

Your assembly language program must begin at memory location `x3000`. You may assume that before your program is loaded into memory and run, address `x4000` contains the starting address of the first element in the LC-3 formatted null-terminated character string, and address `x4002` contains the starting address of where to store the LC-3b formatted null-terminated character string.

You will have no way of determining if your assembly language code works (yet!), but you can use it to determine if your assembler works! Despite this, **Part II will still be graded for correctness.**

~~Hint: Be sure not to overwrite any existing elements.~~
Update: For simplicity, you may assume that the source string and destination string do not overlap at all. (09/04/14)

Requirements

Important note: because we will be evaluating your code in Unix, please be sure your code compiles using gcc with the `-ansi` flag. This means that you need to write your code in C such that it conforms to the ANSI C standard.

You can use the following command to compile your code:

```
gcc -ansi -o assemble assembler.c
```

You should also make sure that your code runs correctly on one of the ECE linux machines.

To complete Lab Assignment 1, you will need to turn in the following:

1. A C file called "assembler.c" containing an adequately documented listing of your LC-3b Assembler.

2. A source listing (LC-3b Assembly Language) of the program described above called "tolc3bformat.asm".

Submission instructions will be posted soon!

Things to watch for:

Be sure that your assembler can handle comments on any line, including lines that contain pseudo-ops and lines that contain only comments. Be careful with comments that follow a `HALT`, `NOP` or `RET` instructions – these instructions take no operand.

Your assembler should allow hexadecimal and decimal constants after both ISA instructions, like `ADD`, and pseudo-ops, like `.FILL`.

The whole assembly process is case insensitive. That is, the labels, opcodes, operands, and pseudo-ops can be in upper case, lower case, or both, and are still interpreted the same. The parser function given in the useful code page converts every line into lower case before parsing it.

You can assume that there will be at most 255 labels in an assembly program. You can also assume that the number of characters on a line will not exceed 255.

Your assembler needs to support all 8 variations of `BR`:

```
BRn LABEL          BRz LABEL
BRp LABEL          BRnz LABEL
BRnp LABEL         BRzp LABEL
BR LABEL           BRnzp LABEL
```

Lab Assignment 1 Clarifications

NOTE: FAQ's for this semester will be posted here. Please check back regularly.

1. Constants can be expressed in hex or in decimal. Hex constants consist of an 'x' or 'X' followed by one or more hex digits. Decimal constants consist of a '#' followed by one or more decimal digits. Negative constants are identified by a minus sign immediately after the 'x' or '#'. For example, `#-10` is the negative of decimal 10 (i.e., -10), and `x-10` is the negative of `x10` (i.e. -16).

2. Since the sign is explicitly specified, the rest of the constant is treated as an unsigned number. For example, `x-FF` is equivalent to -255. The 'x' tells us the number is in hex, the '-' tells us it is a negative number, and "FF" is treated as an unsigned hex number (i.e., 255). Putting it all together gives us -255.

3. Your assembler does not have to check for multiple `.ORIG` pseudo-ops.

4. Since the `.END` pseudo-op is used to designate the end of the assembly language file, your assembler does not need to process anything that comes after the `.END`.

5. The trap vector for a `TRAP` instruction and the shift amount for `SHF` instructions must be non-negative values. If they are not, you should return error code 3.

6. The same label should not appear in the symbol table more than once. During pass 1 of the assembly process, you should check to make sure a label is not already in the symbol table before adding it to the symbol table. If the label is already in the symbol table, you should return error code 4.

7. An invalid label (i.e., one that contains non-alphanumeric characters, or one that starts with the letter 'x' or a number) is another example of error code 4.

8. The standard C function [`isalnum()`] can be used to check if a character is alphanumeric.

9. After you have gone through the input file for pass 1 of the assembler and your file pointer is at the end of the file, there are two ways you can get the file pointer back to the beginning. You can either close and reopen the file or you can use the standard C I/O function [`rewind()`].

10. The following definitions can be used to create your symbol table:

```
#define MAX_LABEL_LEN 20
#define MAX_SYMBOLS 255
typedef struct {
       int address;
       char label[MAX_LABEL_LEN + 1];     /* Question for the reader: Why do we
need to add 1? */
} TableEntry;
TableEntry symbolTable[MAX_SYMBOLS];
```

11. To check if two strings are the same, you can use the standard C string function `strcmp()`. To copy one string to another, you can use the standard C string function `strcpy()`.

12. If you decide to use any of the math functions in math.h, you also have to link the math library by using the command:

```
gcc -lm -ansi -o assemble assembler.c
```

13. When your assembler finds an error in the input assembly language program, it is not required that you print out an error message to the screen. If you choose to do this to make debugging easier, that is fine. What is required is that you exit with the appropriate error code. This is what we will be checking for when we grade your program; we will ignore anything that is printed to the screen.

14. An assembly program which starts with comments before `.ORIG` is valid and your assembler should ignore them. You can assume that there will be no label in front of `.ORIG` and `.END` in the same line.

15. Your assembler needs to be able to assemble programs which begin at any point in the LC-3b's 16-bit address space. While user programs start from x3000 and continue until xFDFF, the assembler could be used to assemble system code as well. The assembler doesn't have enough information when it is assembling the program to determine how the program will be used. In future labs, we will develop what happens if a user tries to access a protected region of memory.

16. .FILL can take an address, signed number, or unsigned number.

17. The trap vector for a TRAP instruction should be a hex number.

18. If a single line in the program contains multiple errors, feel free to exit with the

appropriate error code for any of the errors that line the program contains.