Department of Electrical and Computer Engineering

The University of Texas at Austin

ECE460N, Fall 2024
Lab Assignment 1
Due: Sunday, Sep. 15th, 2024, 11:59 pm

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 have them execute the program you wrote and assembled by the assembler.

Some useful code to get you started can be found HERE.

A reference assembler is also given for you to check the output of your own test cases against the reference version.
Reference LC-3B assembler for linux is available at: assembler.linux
Use the following commands on a linux server (e.g., an LRC server). Make sure each command is copied as a single line with no breaks:

```
    curl -L "https://docs.google.com/uc?export=download&id=1lcJzkMmLFTbWecHH8sk_Lpcch56T4et5" > assembler.linux
    chmod 700 assembler.linux
```

To run : `./assembler.linux example.asm example.obj`

Note: The grader is finicky – please be very careful about the formatting of your files and following all submissions instructions precisely.

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 letter 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(all 8 variations), 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 can assume the input assembly is correct.**

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
```

Part II: Write a program to solve the following problem

THIS PART OF THE LAB HAS BEEN UPDATED, PLEASE CONSIDER THE CHANGED PROBLEM STATEMENT

Write a program in LC-3b assembly language that does the following: Memory location 0x3100 and 0x3101 contain two **8-bit unsigned integers** (We set bit 7 low in the test cases, so you don’t have to worry about LDB sign extending the operands). Write a program that multiplies the two bytes and stores the one byte result in memory location 0x3102. Additionally, if the multiplication results in an overflow of the byte, store a 1 in memory location 0x3103, otherwise store a 0 in memory location 0x3103. **(Yes, overflow checking is part of the program still)**

That is:
Mem[0x3102] = Mem[0x3100] * Mem[0x3101]
Mem[0x3103] = did_overflow_byte(Mem[0x3100] * Mem[0x3101])

Your program should start at location x3000 and be contained in a file called mul_bytes.asm.

You will have no way to determine if your assembly language code works (yet!), but you can use it to determine if your assembler works. Despite this, **we will grade this program for correctness.**

(Note to future 460n TAs: Dr. Patt had us change the problem for part 2 to be multiplying instead of adding. Ask Luke how to change it back if you want)

Requirements

Important note: because we will be evaluating your code in Unix, please be sure your code compiles using gcc with the `-std=c99` flag. This means that you need to write your code in C such that it conforms to the C99 standard.

You can use the following command to compile your code:

```
gcc -std=c99 -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

" mul_bytes.asm ".

Submission instructions are posted here.

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: A FAQ for this semester is below. 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 standard C function `isalnum()` can be used to check if a character is alphanumeric.

6. 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()`.

7. 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];
```

8. 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()`.

9. 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 -std=c99 -o assemble assembler.c
```

10. 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.

11. 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.

12. .FILL can take an address, signed number, or unsigned number.
13. The trap vector for a TRAP instruction should be a hex number.
14. You can assume all locations will fit within the 16-bit address space of LC3-b, i.e., you do not have to check if instruction addresses go beyond xFFFE.

Lab Assignment 1 Submission Instructions

You must use the following naming convention for the files in Lab 1.

- **`mul_bytes.asm`** – The LC-3b assembly language program you wrote.
- **`assembler.c`** – The C source code for your assembler.

You may not submit more than two files for Lab 1.

Notes

- If you worked on the assignment with a partner, *only one* of you needs to submit the files and only one of you may test the lab.
- **Please confirm that your file compiles by running `gcc -std=c99 assembler.c` on any ECE LRC linux machine before submitting your program. You should also test your program on an ECE LRC linux machine.**
- In order to help us assign you the grades, please make sure that you put your names and UTEIDs on the top of the assembler.c file in the **EXACT** following format, as a C comment:

```
/*
    Name 1: Fullname of the first partner
    Name 2: Fullname of the second partner
    UTEID 1: UTEID of the first partner
    UTEID 2: UTEID of the second partner
*/
```

- Example:

```
/*
    Name 1: John Smith
    Name 2: Jane Doe
    UTEID 1: js1234
```

```
    UTEID 2: jd1234
*/
```

- If you worked alone:

```
/*
    Name 1: Jane Doe
    UTEID 1: jd1234
*/
```

Before the deadline, you may resubmit any of the files without penalty. Every time you resubmit a file, the original file is overwritten.

Instructions for submission

Turn in the following files:

`mul_bytes.asm`
`assembler.c`

by following these instructions.