CS321 – Computer Architecture Project

Two pass assembler for an extended SIMPLE instruction set and Emulator

The aim of this assignment is to write a two pass assembler for an extended SIMPLE instruction set. Then write and test programs in SIMPLE assembly. A final part is to write an emulator for the SIMPLE machine.

Tasks

1. Write a two pass assembler for the assembly language.
   The assembler must,
   - Read assembly language from a text file, assigning label values and instruction opcodes. The format of the assembly language is described later.
   - Diagnose common assembly errors such as unknown instruction, no such label, and duplicate label.
   - Produce an object file of the produced machine code. This file should be a binary file. Code starts at address zero.
   - Produce a listing file. There is a choice of the format of the listing file. It can either be a simple memory dump, or show the bytes produced for each instruction, and that instruction's mnemonic. The formats are shown later. Extra marks are available for the latter type of listing file.
2. Test your assembler with the sample programs listed later.
3. Test your assembler with additional programs and submit evidence of this.
4. Write a bubble sort program in SIMPLE Assembler. The start of this file is provided, you have to fill in the blanks.
5. Write an emulator for the SIMPLE machine. This should have some, but not need all, of the functionality of the `emu` program provided.

Assembly Language

This assembly language is for a machine with four registers,

- Two registers, A & B, arranged as an internal **stack**.
- A program counter, PC
- A stack pointer, SP

These registers are 32 bits in size. Instructions have either no operands or a single operand. The operand is a signed 2's complement value. The encoding uses the bottom 8 bits for opcode and the upper 24 bits for operand.

As with most assembly languages, this is line based (one statement per line). Comments begin with a `;' and anything on the line after the `;' is ignored. Blank lines and lines containing only a comment are permitted (and ignored). White space (` ' and tabs) are permitted at the beginning of a line (and ignored). Label definitions consist of the label name followed by a `:', and an optional statement (there is not necessarily a

space between the `:' and the statement). A label use is just the label name. For branch instructions label use should calculate the branch displacement. For non-branch instructions, the label value should be used directly. A valid label name is an alphanumeric string beginning with a letter. An operand is either a label or a number, the number can be decimal, hex or octal.

The following are all permitted lines

```
; a comment
              ; another comment
label1:       ; a label on its own
ldc 5         ; an instruction
label2: ldc 5 ; a label and an instruction
        adc 5 ; an instruction
label3:ldc label3 ;look no space between label and mnemonic
```

Each statement consists of a mnemonic (instruction name) and an optional operand (number or label).

The Instructions

The instruction semantics do not show the incrementing of the PC to the next instruction. This is implicitly performed by each instruction *before* the actions of the instruction are done.

Mnemonic	Opcode	Operand	Formal Specification	Description
data		value		Reserve a memory location, initialized to the value specified
ldc	0	value	B := A; A := value;	Load accumulator with the value specified
adc	1	value	A := A + value;	Add the value specified to the accumulator
ldl	2	offset	B := A; A := memory[SP + offset];	Load local
stl	3	offset	memory[SP + offset] := A; A := B;	Store local
ldnl	4	offset	A := memory[A + offset];	Load non-local
stnl	5	offset	memory[A + offset] := B;	Store non-local
add	6		A := B + A;	Addition
sub	7		A := B - A;	Subtraction
shl	8		A := B << A;	Shift left
shr	9		A := B >> A;	Shift right
adj	10	value	SP := SP + value;	Adjust SP
a2sp	11		SP := A; A := B	Transfer A to SP;
sp2a	12		B := A; A := SP;	Transfer SP to A

call	13	offset	B := A; A := PC; PC := PC + offset;	Call procedure
return	14		PC := A; A := B;	Return from procedure
brz	15	offset	if A == 0 then PC := PC + offset;	If accumulator is zero, branch to specified offset
brlz	16	offset	if A < 0 then PC := PC + offset;	If accumulator is less than zero, branch to specified offset
br	17	offset	PC := PC + offset;	Branch to specified offset
HALT	18			Stop the emulator. This is not a `real' instruction, but needed to tell your emulator when to finish.
SET		value		Set the label on this line to the specified value (rather than the PC). This is an optional extension, for which additional marks are available.

Listing File Format

The listing file is produced by the assembler and is a human readable file showing what value is stored at each address. The format is an address followed by zero or one 32 bit values (as 8 hex characters).

With the output, you can chose to show the human readable mnemonic and operand, that each instruction corresponds to. You can also show labels, by simply listing the address followed by no data bytes.

Here are some acceptable example outputs,

```
00000000 00000111
00000001 00005AB4
00000002 00006500
00000003 00009D01
```

Or, with 4 locations per line,

```
00000000 00000111 00005AB4 00006500 00009D01
```

When showing labels and mnemonics, the output could be (this is my preferred way of showing things)

```
00000000 00000111 br start
00000001 00005AB4 data 0x5ab4
00000002          start:
00000002 00006500 ldc 0x65
00000003 00009D01 adc 0x9d
```

Example Programs

This is a valid, but nonsense assembly file. Your assembler should not issue any errors (it could issue warnings though).

```
; test1.asm
label:  ; an unused label
        ldc 0
        ldc -5
        ldc +5
loop: br loop ; an infinite loop
br next ;offset should be zero
next:
        ldc loop ; load code address
        ldc var1 ; forward ref
var1: data 0 ; a variable
```

This example contains many errors. Your assembler should spot them all (it need not copy the error message exactly, but should issue something appropriate).

```
; test2.asm
; Test error handling
label:
label: ; duplicate label definition
br nonesuch ; no such label
ldc 08ge ; not a number
ldc ; missing operand
add 5 ; unexpected operand
ldc 5, 6; extra on end of line
0def: ; bogus label name
fibble; bogus mnemonic
0def ; bogus mnemonic
```

If you implement the SET pseudo instruction, this program should assemble

```
; test3.asm
; Test SET
val: SET 75
ldc     val
adc     val2
val2: SET 66
```

Here's a real file, the `ldc result` is *not* a mistake.

```
        ldc 0x1000
        a2sp
        adj -1
        ldc result
        stl 0
        ldc count
        ldnl 0
        call main
        adj 1
        HALT
;
main:   adj -3
        stl 1
        stl 2
        ldc 0           ; zero accumulator
        stl 0
loop:   adj -1
        ldl 3
```

```
        stl 0
        ldl 1
        call triangle
        adj 1
        ldl 3
        stnl 0
        ldl 3
        adc 1
        stl 3
        ldl 0
        adc 1
        stl 0
        ldl 0           ; reload it
        ldl 2
        sub
        brlz loop
        ldl 1           ; get return address
        adj 3
        return
;
triangle:adj -3
        stl 1
        stl 2
        ldc 1
        shl
        ldl 3
        sub
        brlz skip
        ldl 3
        ldl 2
        sub
        stl 2
skip:   ldl 2
        brz one
        ldl 3
        adc -1
        stl 0
        adj -1
        ldl 1
        stl 0
        ldl 3
        adc -1
        call triangle
        ldl 1
        stl 0
        stl 1
        ldl 3
        call triangle
        adj 1
        ldl 0
        add
        ldl 1
        adj 3
        return
one:    ldc 1
        ldl 1
        adj 3
        return
;
count:  data 10
result: data 0
```

Notes

About Files
The relevant files are:

1) Input: a file in text mode containing the assembly code (you are given some examples of input files)

2) Output: a listing file in text mode showing the assembled program (should be similar to the memcpy.l provided)

3) Output: a log file in text mode showing the state of the process (here is where errors, warnings and success/failure should be recorded)

4) Output: a file in binary mode of the assembled program (just the machine code in hexadecimal). Check `fopen` for how to open a file in binary mode.

Two Passes
: A two pass assembler naturally needs to scan the source file twice. You can and should use a single routine to do both passes, provided that on the first pass it outputs no code, and doesn't fail on undefined labels, and that on the second pass it outputs code and does fail on an undefined label. This is a better solution than having two different routines for each pass (because of code reuse).

One Read
: Alternatively you could read the program into an internal form, and then process that twice. This uses less file IO, at the cost of using more memory. If you do this, the internal form should be more advanced than simply storing each line of the file as a string. It is a more complicated solution.

Reading a Line
: The functions `gets` and `fscanf` should be handled with care. `gets` suffers from buffer overrun -- if the lines are longer than the programmer expected, it will merrily trash subsequent locations. `fscanf` can also suffer the same problem with reading strings. However, it has a worse flaw, in that it does not recover well from errors. If the line does not contain what `fscanf`'s format string expects, it is very hard to resynchronize. You should use something like `fgets` to read a buffer and then `sscanf` or `strtok`, for a robust program.

  Also `getchar` and its ilk return an `int`, not a `char`. It needs to do this because it needs to return all the possible `char` values, plus another value to mean 'end of file'. So storing the return value into a `char` variable is **wrong**.

Tables
: If you define your instructions in a suitable data structure, you can define all the instructions in a table. This is better than hard coding particular instruction semantics throughout the program.

Number Format
: Numbers in the source file are written as an optional sign, a digit and trailing digits and letters. Not all these will be *valid* numbers. You can use `strtol` to convert a string to a number, and tell you how far it got into the string. It will also

figure out whether the number is decimal, hex (begins with `0x'), or octal (begins with `0').

Printing Label Values

One of the example listing output shows an instruction printed as `br start`. Firstly, although not mentioned, it would be permissible to show this as `br 75` (if the branch offset is 75), but that wouldn't be as good. As the assembler has replaced label names with their values, how is this achieved? When listing an instruction, the assembler could check if its operand is a PC offset. If so, it could calculate where the branch would go to, and then look in its table of labels to find a label at that address. If it finds one, then use it, otherwise just print the raw offset. Notice, that the assembly source might contain `br 7`. You do not need to check that this actually lands up at an instruction -- if it doesn't it's the assembly language programmer's fault.

ldc result

The `ldc result` instruction in the example program has confused some of you. `ldc` loads a constant, but here I am using it with a label -- why? What I wanted to get hold of was the address of the `result` data location, so I could access it as an array. So, I needed its address, hence the `ldc`. All the assembler does with the operand is lookup the label's value and put that in the instructions operand. *It doesn't understand what an instruction does.* You'll see I access it with `ldnl` instructions later on.

Warnings

What's the difference between errors and warnings? An error is something which is incorrect and prevents completion of the task. A warning is something which is not strictly incorrect (i.e. is allowed), but is strange or dubious. It does not prevent completion of the task. An example is an unused label. It is not incorrect to declare a label and not use it, however that is a strange thing to do, and might be hiding some more important problem. For instance perhaps the programmer misspelt the label when trying to use it, and accidently used a different label. This is one reason why variable and function names should be very different to each other. Don't go calling your variables `var1', `var2', `var3' etc!