CS 474: Object Oriented Programming Languages and Environments
Spring 2018

Second C++ project

Due time: **7:00 pm on Saturday 5/5/2018**

Instructor: Ugo Buy
TA: Parneet Kaur and Siddhesh Chavan

Total points: 120
Full credit: 100 points (20 points extra credit)

This project is about building an *Assembly Language Interpreter (ALI)* for a *Simple Assembly Language (SAL)* in C++11. Fortunately for you SAL has the same limited set of instructions, described below, as Project 2 in this semester. SAL programs are executed on a virtual (emulated) machine consisting of a memory, which stores program code and program data, an *accumulator* register, an additional register and a *Program Counter (PC)*, which keeps track of the instruction being currently executed. For the C++ version ALI does not include a graphical user interface. Instead ALI reads a SAL program from a file and executes the program either one line at a time (in debug mode) or all the way to completion (normal run mode). ALI interacts with a user through the following command line:

1. `r` — This command reads a SAL program from a file named *input.sal*. The file should be in the same directory as your executable ALI file. You may assume that the input program is syntactally correct. The program is stored in internal ALI memory and displayed on the user's console.

2. `d` — Execute the program in debug mode. This command causes one line of code to be executed. The PC, registers, bits and the state of memory are updated as a result of the execution. The value of the registers, the PC, the bits and the content of memory after the instruction is executed are diplayed on the user's screen.

3. `r` — Run the program to completion. This command causes the execution of program starting from the current PC instruction until the halt instruction is found or the program reaches its last instruction. The content of the registers, PC, bits and memory are displayed on the user's console.

4. `s` — Save the program state. This command causes the content of memory, the bits and the registers to be saved to a file named *output.txt*.

5. `q` — Quits the ALI.

The computer hardware uses 32-bit words and consists of the following components:

1. *Memory.* A 32-bit, word-addressable memory (RAM) for data, holding 256 words. Words are addressed by their location, starting from location 0 all the way up to location 255. Each location may either hold a signed integer in 2's complement notation or a SAL instruction.

2. *Accumulator.* A 32-bit register. It is also known as *Register A* or *A* for short.

3. *Additional register.* A 32-bit register also known as *Register B* or *B* for short.

4. *Program counter (PC).* An 8-bit program counter (PC). The PC holds the address (number in program memory) of the next instruction to be executed. Before the program starts execution, the PC holds the value 0. It is subsequently updated as each instruction is executed.

5. A *zero-result bit.* This bit is set if the last ADD instruction produced a zero result. This bit is cleared if the last ADD instruction produced a result different from zero. The initial value is zero. The bit is changed only after ADD instructions are executed.

6. An *overflow bit.* This bit is set whenever an ADD instruction produces an overflow (i.e., a result that cannot be stored in 2's complement notation with 32 bits). It is cleared if the ADD instruction did not produce an overflow. The initial value is zero.

The registers are used to hold data in arithmetic operations (i.e., additions). The program counter holds the index value (starting at 0) of the next instruction to be executed. SAL has the instruction set shown in Table 1.

DEC *symbol*	Declares a symbolic variable consisting of a single letter (e.g., X). The variable is stored at the memory location of this instruction.
LDA *symbol*	Loads byte at data memory address of *symbol* into the accumulator.
LDB *symbol*	Loads byte at data memory address *symbol* into B.
LDI *value*	Loads the integer *value* into the accumulator register. The value could be negative but must be in the range of 32-bit 2's complement numbers.
ST *symbol*	Stores content of accumulator into data memory at address of *symbol.*
XCH	Exchanges the content of registers A and B.
JMP *number*	Transfers control to instruction at address *number* in program memory.
JZS *number*	Transfers control to instruction at address *number* if the zero-result bit is set.
JVS *number*	Transfers control to instruction at address *number* if the overflow bit is set.
ADD	Adds the content of registers A and B. The sum is stored in A. The overflow and zero-result bits are set or cleared as needed.
HLT	Terminates program execution.

Table 1: Instruction set of SAL.

You may assume that SAL programs are entered correctly by users of ALI. (You are not required to perform error diagnosis or correction). Information in data memory and the registers should be displayed in decimal format. Program source code should be displayed in symbolic (i.e., textual) format. Finally, you must use inheritance in your implementation. One good place for inheritance is to define an abstract superclass for SAL instructions with concrete subclasses for each particular instruction. Beware of infinite loops in SAL.

Extra credit. If you implement correctly the *Command* pattern from the Gang-of-Four book, you will receive a 20% increase on your project grade, up to 20 extra points.

You must work alone on this project. Save all your code in a collection of header and code files and submit a zip archive with a (short) readme file containing instructions on how to use your ALI. The archive should also contain a sample *input.sal* file. Submit the archive by clicking on the link provided with this assignment. Your code should compile under the GNU C++11 or the CLANG C++11 compiler. No late submissions will be accepted.