# CS 474: Object Oriented Programming Languages and Environments

Fall 2022

*Second Ruby project*

*Due time:* 11:59 pm on Sunday 10/23/2022

Instructor: Ugo Buy
TAs: Huy Truong and Alexis Rodriguez

*Total points: 100*



This project is about building an *Assembly Language Interpreter (ALI)* for a *Simple Assembly Language (SAL)*. SAL has a limited set of instructions, specified in the table below. ALI is a virtual (emulated) machine consisting of the following components: (1) a memory, which stores both the SAL program's source code and the data that the program uses, (2) an *accumulator* register, (3) an additional register and (4) a *Program Counter (PC)*, which keeps track of the next instruction to be executed. Your ALI can execute SAL programs one line at a time (in a sort of debug mode) or all at once until either a halt instruction is encountered or the last line of source code is reached.

The execution of the ALI consists of the following three steps:

1. Prompt the user for a file name in the current directory.
2. Read a SAL program from the file. The program is stored in the memory starting at address 0.
3. Execute a command loop consisting of the following three commands:

   `s` – Execute a single line of code, starting from the instruction at memory address 0; update the PC, the registers and memory according to the instruction; and print the value of the registers, the zero bit, the overflow bit, and only the memory locations that store source code or program data after the line is executed.

   `a` – Execute all the instructions until a halt instruction is encountered or there are no more instructions to be executed. The program's source code and data used by the program are printed.

   `q` – Quit the command loop.

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

1. *Memory.* A 32-bit, word-addressable memory (RAM) for data and source code, holding 256 32-bit words. Words are addressed by their location, starting from location 0 all the way up to location 255. Locations 0 through 127 hold program source code (i.e., a sequence of SAL instructions). Locations 128 through 255 hold program data (i.e., 32-bit 2's-complement integers). Thus, each location may either hold a signed integer in 2's complement notation or a SAL instruction depending on the address of the location.
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. Legal values for this register are unsigned integers from 0 to 127.

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 for 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 an available location in data memory. |
| 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. |
| STR *symbol* | Stores content of accumulator into data memory at address of *symbol.* |
| XCH | Exchanges the content 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.

Here are additional directions for this project.

1. You may assume that SAL programs are entered correctly by users of ALI. (You are not required to perform error checking or correction).

2. Information in data memory and the registers can be displayed in binary, decimal, or hexadecimal format. This means that you do not have to translate decimal numbers into binary or hex format within ALI. However, you should be aware of the range of 2's complement 32-bit integers in order to set the overflow bit appropriately after additions. Recall that this range is from $-2^{31} \rightarrow 2^{31} - 1$.

3. Program source code should be displayed in symbolic (i.e., textual) format.

4. Before starting program execution all memory locations, bits and registers are filled with zeros.

5. 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 kind of instruction. The abstract superclass may define a deferred method *execute()*, and constants *Opcode* and *ArgType* denoting the symbolic opcode (e.g., ADD) and argument type (one of NONE, NUMBER, STRING) of each instruction type. Concrete subclasses could then define the specific details and behavior of each instruction type.

6. Beware of infinite loops in SAL programs. You may want to halt the execution of a SAL program after, say, 1,000 instructions have been executed and ask the user whether to continue execution.

**You must work alone on this project.** Your project should be submitted as a zip archive by clicking on the link provided with this assignment. File I/O in Ruby is discussed, e.g., at the following URL: https://www.tutorialspoint.com/ruby/ruby_input_output.htm. No late submissions will be accepted.