**PROJECT – Part 1 – due Monday, March 22 at 11:55 pm  
Virtual Machine, Machine Language, Assembly Language**

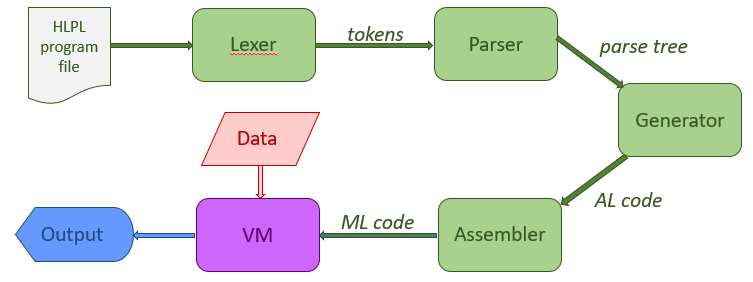
* **Introduction**

In this first part of the project, you will design a machine language similar to the one we studied in Unit 3 and implement its interpreter in C. Yes, it sounds a bit strange, a lower-level language interpreted to run on a virtual machine (VM) written in a higher-level language (HLPL), but why not? The point is for you to have the experience of designing a language and writing a simple interpreter for it. Then you will write a more sophisticated translator from a higher-level language to this VM.

In addition to designing the numeric ML, you will also design a symbolic version of it, like an assembly language (AL). You will use the symbolic version as an intermediate step in the translation from the higher-level to the lower-level numeric language. It will be useful for you to examine and debug the code generated in translating from the higher-level language to code that runs on the VM.

**Don’t forget to apply good design principles in designing both ML and AL!**

The overall translation process, once you are done with the entire project, will look as in the figure:



* **The Virtual Machine, Its Language and the Interpreter**

**Architecture**

The machine you are designing the machine language for is a bit different from the one in Unit 3. In the first place, it has a larger memory (20000 words), but each word is still 10 digits long plus a sign.

The machine also uses some registers. For now, you are only going to be concerned with two registers (other registers will be introduced later):

* The IP register: it stores the address of the next instruction.
* The ACC register: this is the **Accumulator**, a temporary storage place for data.

**Instruction Set**

In this machine, the instructions use only two operands. The third operand, if needed, is assumed to be in the ACC register. This gives us some choices in how we design the instruction set. For example, an instruction that adds two numbers could use one of the two following alternative sets of assumptions:

* The instruction contains both operands. The result is stored in the Accumulator and, from there, transferred to main memory by another instruction.
* The instruction contains one operand and a destination. The other operand is in the accumulator. Executing the instructions uses the operand in the accumulator and the one in the instruction (in a given order—not important for addition but important for subtraction and division, for example) and stores the result in the destination specified in the instruction.

The instruction set should also allow for storing literal values in the instruction itself, not just memory addresses. You need, of course, to have a way of distinguishing between when the operand represents a memory address and when it represents a value and, in the latter case, whether it is positive or negative. Obviously, the literal values that you can store in an instruction are limited to less than what can fit in a whole word.

Regarding the operators that should be in this language, you can get inspired by the ones used by the pseudo code of Unit 3, but you can also modify that set and include new ones. Just make sure you include the ones you might need now and leave a few free for operations you may need to add later.

After you take all these decisions, make sure that your design clearly specifies the syntax and semantics of the instructions and provide a table of operators (opcodes) indicating the assumptions they make about operands and results.

**ML Interpreter**

The ML interpreter should be written in C. It should follow the read-decode-execute cycle presented in slide set (3e), though the decoding process of this instruction set will be a little different.

It should simulate the data and code memory structures described above in the architecture section using C arrays, where each element of the array corresponds to a memory address.

**Machine Language Program Organization**

You will probably want to use a ML program to debug the interpreter. If you write a ML program to a file, please use the same organization as for the Unit 3 language. I.e., the program in a file has: a first part that is used to initialize data memory, a separator, the second part that contains code, and if you are going to have data in the file following the program, another separator followed by the data itself.

* **The Assembly Language (AL) and Assembler**

The symbolic assembly language that precedes the ML in the translation process should have a very close mapping to the ML. You should be able to convert from Al to ML easily. In order to do this, the AL code should use a fixed format, separating the opcode from the operands by white space and using fixed width columns. E.g.:

012345678901234567890123456789

OPCODE OPERAND1 OPERAND2

The example above shows that the opcode always starts in column 0, the first operand in column 10, the second operand in column 20. Unlike the ML, where the opcode and operands contain a fixed number of digits, the symbolic names of the opcodes and operands can vary in length, as long as they are no longer than 9 characters (in order to leave at least one space of separation before the next element in the instruction).

In designing your AL, you have more freedom than with the machine language. For example, some of your instructions may not need to have operands or have both operands. When you convert those instructions to ML, you will, however, need to make the necessary adjustments.

The assembler, which converts from AL to ML, should also be written in C. You will need to build a label table for symbolic labels, and a symbol table for symbolic variable names.

**Assembly Language Program Organization**

The AL program file can have a less strict organization than a ML program file, but it will help you to keep the structure of the files somewhat aligned. For example, you could choose a “à la COBOL” organization, where you signal with a reserved word the start of the DATA section and use another reserved word to signal the conclusion of that and the start of the CODE section.

* **Submission and Grading**

The entire project counts for 15% of your final grade. This delivery counts for 3% of your final grade, or 1/5of the project grade.

The contents of the submission are specified in the table below.

**The team captain should submit to Jenzabar, on or before the due date, a zip or rar file.**

|  |  |
| --- | --- |
| **CONTENTS** | **POINTS** |
| **Cover document** | **10** |
| Team members | 1 |
| Team captain for this delivery | 1 |
| Paragraph explaining who worked on what for this delivery | 1 |
| Paragraph reflecting on team dynamics (successes, challenged, issues, etc.) for this delivery | 7 |
| **Assembler** | **20** |
| A design document for the AL, describing and justifying your choices | 5 |
| Code for the assembler | 10 |
| Documentation for your assembler | 5 |
| **ML Interpreter** | **20** |
| A design document for the ML, describing and justifying your choices | 5 |
| Code for the ML interpreter | 10 |
| Documentation for your ML interpreter | 5 |
| **Sample programs with expected output** | **10** |
| The simple code can be 10 lines or less | 2 |
| The medium complexity code should include one loop and one conditional | 3 |
| The high complexity code should include nested loops and nested conditionals. | 5 |
| **TOTAL** | **60** |

Code documentation requirements: It is assumed that your code will contain sufficient internal documentation to be easily understandable. The external documentation should provide a brief overall overview of the structure of the program (how code is structured in files and major data structures) and how to compile it and run it. Any assumption about the location of input and output files should also be stated here.