Research

Before starting programming or discussing details of my topic, it is important to do research into how it will work and what the existing environment for things like this is.

# Existing similar programs

To start, this overall project is of a single toolkit with the steps of compilation and assembly as well as execution built in. Toolkits like that do exist and are common; the Java Development Kit is exactly this, containing a Java compiler as well as a Virtual Machine to run the bytecode generated. In fact, I may use the JDK as a reference guide for certain aspects of my program. However, my program also will have a focus on the educational value it presents, including both explaining its running and simply ease of use. I cannot find any existing program which does this.

The individual code execution stage is perhaps most similar to the Little Man Computer. This is a basic way of viewing a simple program running in the Von-Neumann architecture, watching the “memory boxes” and registers as they go. My interpreter will be similar to this, and I will try to keep it as simple to view. The differences will be that my interpreter will have much more memory, will have a larger instruction set, and will have a few more registers. I will base it approximately on a real-life CPU, to make it more realistic than the LMC.

The closest thing I could find to an educational compiler is the Oberon-0 compiler (at github.com/rcorcs/OberonC). This is a compiler for the Oberon 1.0 programming language, created in 1986 as a member of the ALGOL language family. The language is very obscure, and it does not actually perform the educational walk-through I want my compiler to. It seems that no actual equivalent of what I am trying to make exists already, so there is no real alternative to my program.

# Compilation process

The overall toolkit can be viewed like this:

Hand-written assembly

The compiler will take in a high-level language and output a custom form of assembly code. The assembler then takes this assembly code, or assembly code that a user has written manually, and outputs a form of bytecode based on the x86 architecture and Windows’s EXE format. The interpreter then takes this bytecode and runs it, simulating the actions of a CPU.

### Compiler

The compiler is the most complicated of these programs. Explain compilation process in some level of detail.

## Assembler

The assembler is much simpler. It splits the assembly code into individual lines, and performs some very basic syntax analysis. The sequence of characters before the first space will be interpreted as the instruction, with the arguments coming after. The assembly language is based on the x86 Intel syntax, which is reasonably simple but will require some contextual knowledge by the assembler, such as finding whether an argument refers to a memory location, numeric literal or register.

The assembler will output a custom form of bytecode, vaguely based on the x86 style, which can be executed by the interpreter.

## Interpreter

The interpreter will take the machine code generated by the assembler and run it, simulating the actions of the CPU. I will describe the exact specifications of this CPU, along with the bytecode and its assembly language, in the Decomposing the Problem (3.06) section. A graphical interface, similar to that of the LMC, will be shown on screen, and the interpreter will run through each command in the bytecode. For each one, it will display the assembly command (e.g. “mov r1, r2”) and an English version of it (“move the content of Register 1 to Register 2”) at the top of the screen, and will animate the relevant numbers moving around the buses and changing the positions where values are stored. There will also be a “memory preview”, which will show a 16\*16 grid of memory around the last accessed value, or the user can select what memory addresses to look at. Overall, this should be a very thorough way of viewing how the program is run.

# CPU to model

It will be very roughly based on the Intel 80386 processor’s execution model and layout, as well as its x86 instruction set, as described by the document “Intel386TM DX Microprocessor”. I have chosen this as it works with the x86 architecture/instruction set, extremely common in modern computers and so accurate for study, but also because this was Intel’s first (and thus simplest) 32-bit processor. This strikes a balance between real-life accuracy (which is where the LMC lacks) and simplicity.

# Languages to use

The assembler and compiler are likely the most complicated parts of this program, and I will therefore program them in Python. Python is an easy-to-write language with lots of tools out of the box to assist in efficiently working through a large task, and I am not altogether concerned about the speed at which it runs – these are merely compilers/assemblers, so they do not handle the execution stage. Python’s StringIO objects, as well as concepts like generators, the iterator protocol and dictionaries, and easy-to-use regular expression support, will all make this a very useful language to program these tools in.

In writing the interpreter however, I want speed to be available. While the animations mean that it will not run at a significant speed, I would as a personal addition like it to be possible for the interpreter to run as an actual virtual machine at a reasonable speed. The speed of execution will not be close to that of compiled languages, but if well written could potentially rival a larger, directly interpreted language like Python. Even if this feature is disabled and educational mode is the main aim, I want fast execution to be a possibility. Therefore, I will program this in C, partly because it is a very fast-running, compiled language, and also because it is the language my program will eventually compile, so it seems like a satisfying symmetry. The only difficulty may be GUI programming, which is possible in C but is very difficult. To tackle this, I may write the GUI in Python and use its ctypes module to link to the C code – the availability of this module is another reason for choosing C as the language (rather than C++ for example).