Aseembly Language

1 Assembly Language

We generally partition mechanical languages into 2 classes:

- **high-level**
  High-level languages tend to be **ISA-independent**. ***e.g.*** C, C++, Java, Python
- **low-level**
  Assembly languages are low-level languages, and they are very much **ISA-dependent**. It is usually the case that each ISA has only one assembly language.

2 An Assembly Language Program

2.1 Instructions

An instrution in assembly language consists of 4 parts, and two of it (`Label` and `Comment`) are optional.
`Label Opcode Operands ; Comment`

2.1.1 Opcodes

The **opcode** is a *symbolic name* for the opcode of the correspounding LC-3 instruction so we can memorize the instruction easier. ***e.g.*** `ADD`, `AND`, or `LDR` rather than 0001, 0101 or 0110.

2.1.2 Operands

The number of operands depends on the operation being performed. ***e.g.*** `ADD` requires 3 operands.

A **literal value** must contain a symbol identifying the representation base of the number. We use *# for decimal*, *x for hexadecimal*, and *b for binary*. (must)

Sometimes we use labels as operands so that we don't need to remember the explicit 16-bit addresses. Details will be covered in the next part.

2.1.3 Labels

Labels are symbolic names used to *identify memory locations* that are referred explicitly in the program. In LC-3, a label consists of from 1 to 20 alphanumeric characters starting with a letter of the alphabet.(reserved words excluded)

There are 2 reasons for explicitly referring to a memory location:

- The location is the target of a branch instruction. ***e.g.*** `BRnzp LOOP`
- The location contains a value that is loaded or stored. ***e.g.*** `LD R1, TEMP`

2.1.4 Comments

Comments are messages intened only for human consumption. They have *no effect* on the translation process and indeed are not acted on by the LC-3 assembler.

They are identified by *semicolons*. A semicolon signifies the rest of the line is a commment and is to be ignored by the assembler. ***e.g.*** `LD R0, ASCII ; Load the ASCII template` The message "Load the ASCII template" is a comment.

2.2 Pseudo-Ops (Assembler Directives)

Pseudo-op is also called **assembler directive**, and it dose not refer to an operation that will be performed by the program during execution.

2.2.1 .ORIG

`.OGIR` tells the assembler where in memory to place the LC-3 program. (to specify the start address) We normally write `.ORIG x3000`, which means our program will start at the address `x3000`.

2.2.2 .FILL

`.FILL` tells the assembler to set aside the next location in the program and initiallize it with the value of the operand. The value can be either a number or a label. ***e.g.*** `x3006: .FILL x0030` then `x0030` will be stored in the location `x3006`.

2.2.3 .BLKW

`.BLKW` tells the assembler to set aside some number of sequential memory locations. (**i.e.** a **BL**oc**K** of **W**ords) ***e.g.*** `x3007: .BLKW 1` then the location `x3007` will be set aside then we can store or write content to that position.

2.2.4 .STRINGZ

`.STRINGZ` tells the assembler to initiallize a sequence of $n+1$ memory locations. The argument is asquence of n characters inside double quotation marks. The first n words of memroy are initiallize with the zero-extended ASCII codes of the correspounding characters in the string. The final word is 0. (`\0`)

2.2.5 .END

`.END` tells the assembler it has reached the end of the program. Contents after `.END` will not be processed by the assembler.

Note that `.END` does not stop the program during execution. In fact, `.END` does not even exist at the time of execution.

3 The Assembly Process

It's the job of the LC-3 assembler to perform the translation from the LC-3 assembly language into a machine language program.

We use the command `assemble` and it requires the filename of your assembly language program as an argument, and it produces the file outfile, which is in the ISA of LC-3. `assemble soutiona1.asm outfile`

The assembly process is done in **two complete passes** (from beginning to `.END`) through the entire assembly language program.

3.1 The First Pass: Creating the Symbol Table

The **symbol table** is simply a correspondence of *symbolic names* with *their 16-bit memory addresses*. In the **first pass** we identify each label with the memory address of its assigned entry.

e.g.

Symbol	Address
TEST	x3004
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

3.2 The Second Pass: Generating the Machine Language Program

The **second pass** consists of going through the assembly language line by line, with the help of the symbol table. At each line, the assembly language instruction is translated into an LC-3 machine language instruction.

The only part of the `LD` instruction left to do is the PCoffset. So it's necessary that the address of the source is no more than +256 or -255 memory locations from the `LD` instruction. Otherwise, assembly error.

4 Beyond the Assembly of a Single Assembly Language Program

Actually, this part will probably not be in the final exam. If interested, you can turn Chapter 7: Linking of CSAPP (**i.e.** Computer Systems: A Programmer's Perspective), or refer to my note.

When a computer begins execution of a program, the entity being executed is called a **executable image**. The executable image is created from modules often created independently by several different programmers (also different object files).

we write `PTR .FILL STARTofFILE` in the program but there is no such a label `STARTofFILE` in our program while the label is in another module by different programmer. We can use `.EXTERNAL STARTofFILE`, then at link time when all modules are combined, the linker will find the symbol table entry.