**Introduction to Assembly Language**

The "programs" we spend most of our professional lives writing cannot really be called "computer programs" - they are more properly described as formalized algorithms, and are written in a "higher level language" that is designed precisely to effectively and unambiguously capture algorithms in different fields of endeavor (business, science, networking, web design, etc.)

These "source code" files have to then be translated ("compiled" or "interpreted") into machine language - i.e. the binary patterns (the instructions) that cause a microprocessor to perform the required operations on other binary patterns stored in memory (the data). Every microprocessor has a unique set of such instructions, and hence its own machine language.

Of course, it would be a nightmare if the only way we could program our microprocessors was to write the code directly in binary! So we have an intermediate form, called assembly language, which is just a human readable version of machine language - e.g. in the LC-3, the mnemonic ADD instead of the code 0001.

Before we get into the programming, though, we have to outline some basic ideas about what a microprocessor can actually do (or rather, the microprocessor plus system memory).

Let's start with the most basic description of a computer: an electronic device that takes in information, stores it, and manipulates that information to create new information.

1. System Memory ("Random Access Memory" or RAM) stores BOTH the information (as binary - the data) AND the list of steps for manipulating that information (also as binary - the instructions).   
You can think of memory as a huge wall of slots, each slot having an address. You can read the binary value stored in any given address, and you can write a new binary value to any address.

2. The microprocessor has circuits for storing "words" of data (these may be of different widths in different microprocessors - 8 bits, 16 bits, 32 bits, 64 bits ...); it has circuits for moving this data around; and it has circuits for manipulating this data - e.g. performing arithmetic or logic operations on data. The circuits for storing words of data are called registers, and the collection of circuits that do the operations is called the Arithmetic and Logic Unit - the ALU.

3. A "program" is a list of instructions, stored in memory, to be executed in sequence. You "launch" a program by pointing to the first instruction; that does its job, then the next instruction is fetched, etc. Some instructions - as we'll see below - perform arithmetic or logic manipulations; some read or write data from/to memory; and some alter the sequence in which the following instructions are fetched.

So now we're ready for an overview of assembly languages: even though every family of microprocessors has its own unique assembly language, most of them follow the same three basic categories.

**OPERATIONS:**

These are the instructions that cause the microprocessor to pass data to the ALU for arithmetic and boolean logic operations.

In the LC3, there are only three such instructions:  
**ADD** (add 2 two's complement integers), **AND**, **NOT** (bitwise boolean operations).

The "bit-width" of the ALU *(i.e. the size of the data words it can operate on)* is called the **word size** of the microprocessor. The LC-3 has a word size of 16 bits; your laptop's microprocessor probably has a word size of 64 bits.

More powerful microprocessors, of course, also have distinct instructions for subtraction, multiplication, and division of integers; plus similar operations for floating point numbers.

**DATA MOVEMENT:**

These are the instructions that move data between RAM and the microprocessor's on-board registers.  
There are usually only a small number of these (e.g. 8 in the LC-3), but they are fast and directly connected to the ALU, and are the same bit-width as the ALU; whereas RAM is usually vast, and often stores data as single bytes, no matter the word size of the microprocessor.  
e.g. in a typical Intel system, a single 32-bit integer would have to be stored across 4 one-byte RAM addresses, whereas on-board the microprocessor it would be stored in a single 32-bit register (or in one half of a 64-bit register).

The LC-3 design is a bit different from most production systems; it stores data as single 16-bit words both in RAM and in the registers on-board the microprocessor.

As you would expect, there are two sets of data movement instructions: Load instructions copy data from RAM to registers, and Store instructions copy data back from registers to RAM. In the LC-3 there are three of each (you will learn how to use these three "memory addressing modes" correctly over labs 2 and 3):

Load: **LD** ("Load direct"); **LDR** ("Load Relative"); **LDI** ("Load Indirect")

Store: **ST** ("Store Direct"); **STR** ("Store Relative"); **STI** ("Store Indirect");

There is also a "helper" instruction **LEA** ("Load Effective Address") that does not actually copy any data, but sets things up, usually for LDR or STR

**CONTROL:**

As you already know from your higher level programming, it is the control structures (branching and looping, function calls) that make our programs able to do interesting things - and it is the same for assembly language.

You will learn to reproduce the functionality of if-else branches, while, do-while, and for loops using just two very basic control instructions in the LC-3:

**BR** (Conditional Branch, which will transfer control to a specified instruction depending on the values of the three Condition Codes - see below); and **JMP** (unconditional branch, or Jump to the specified instruction unconditionally)

And you will learn to construct a rather primitive version of functions called subroutines using two forms of a Jump to Subroutine instruction: **JSR** and **JSRR**.  
We'll leave the details till about lab 5.

Then there is an instruction which invokes BIOS ("Basic Input Output System") routines that are written and provided by the manufacturer: **TRAP**

Finally, there is one instruction that we won't be using in this course, but we will mention in about week 9 - **RTI** ("Return from Interrupt")

If you've been counting, you'll see that the LC-3 has a total of just 15 instructions - and yet it could in principle perform any calculation that an iCore 7 could do - though it might take a billion times longer :)  
In fact, you could design a working microprocessor with as few as 7 or 8 instructions  
Having 15 instructions means, of course, that we have to allocate 4 bits for the "opcode" - the part of the instruction that identifies it; the remaining 12 bits are used to convey other parameters needed to implement the instruction - e.g. the register(s) involved, the memory address involved, and so on.

====================================================================================

**Some other details of the LC-3 Instruction Set Architecture (ISA):**

* 16-bit word size (i.e. the ALU can perform 16-bit two's complement arithmetic and 16-bit binary "bitwise" operations. This means that also the registers and buses are 16 bits wide
* 16-bit memory addressing, meaning that it takes a 16-bit word to address a location in memory. So the "address space" *(i.e. the total number of memory "slots" in RAM)* is 2^16 = 64k slots.
* 16-bit addressability i.e. each "slot" in memory holds 16 bits (2 bytes), giving the LC-3 a grand total memory capacity of 64k address space \* 2 bytes per slot = 128 kbytes
* 8 x 16-bit General Purpose Registers, (GPRs) named R0 through R7
* Three Memory Addressing Modes (Direct, Relative & Indirect)
* Three Condition Codes, N, Z, P (see below)
* More later in the course :)

====================================================================================

**So what are condition codes?**

Just the single most important tool for writing useful programs :)

**Every time a value is written to one of the GPRs (General Purpose Registers), three circuits detect whether that value is negative (N), zero (Z), or positive (P) and store a 1 in the corresponding "condition code" (a 1-bit register).**

This allows us to then use the conditional branch instruction BR to take a different path through our code depending on whether the last modified register met a specific test.

For instance, if we want to repeat a block of code 16 times, we would store the value 16 in, say, register 1 (R1); at the end of the loop body, decrement R1 by 1; then, if R1 is still positive (the P code == 1), branch back to the start of the loop.  
As soon as R1 becomes 0, the P condition code will no longer be 1, so don't branch, just continue *(i.e. "fall out of the loop")*.

In pseudo-code, it would look like this:

R1 <= #16 ; write the value decimal 16 to R1

Loop:

; loop body goes here

R1 <= (R1 - 1) ; decrement R1 (NZP codes set)

If (P == 1) Branch to Loop ; otherwise continue to next instruction

In LC-3 assembler, this last instruction, the conditional branch, would be written

BRp Loop ; we read this as "Branch on positive to Loop"

The condition that causes the branch to be taken can be any combination of n, z, p  
 - e.g. BRnz means transfer control to the specified instruction if the last modified register was  
negative or zero *(i.e. not positive)*.   
You can even specify BRnzp *(meaning, ALWAYS take the branch)*.