**Instruction Set Architecture (ISA)** is the set of information needed to interact with a microprocessor, without the details of the microprocessor. It’s sort of like a buffer for programmers to interact with a microprocessor without having to know all the details of how it works. The **registers** of the ISA are the memory-accessible areas set aside for programmers to work with, as well. But to better understand Instruction Set Architecture, it’s first important to discuss how programming languages work.

Programming languages are usually divided into three different categories. The first is **high-level languages**, which hide the details of the computer and operating system from the programmer. High-level languages are said to be **platform-independent**, or, the same programs can be run on different computers with different microprocessors. Languages such as C++, Java, and FORTRAN are examples of high-level languages.

**Assembly languages** are at a lower level of abstraction: programs written in assembly languages are unique to the microprocessor they were written on. Manufacturers of microprocessors usually make sure their new processors are **backward compatible** with previous versions of microprocessors for this purpose. Unlike high-level languages, assembly language can directly manipulate the data stored in a microprocessor and its internal components. Assembly languages are not platform-independent.

**Machine languages** are the lowest level of programming languages. These languages contain the binary values necessary to perform certain operations on a microprocessor. Machine language is not written directly. Rather, high-level languages and assembly languages are converted to machine language, which is then executed by the microprocessor. Machine languages, like assembly languages, are platform-specific. Usually, the machine language for a particular microprocessor is backward compatible with the previous versions of the microprocessor, as well.

**Compilers** take programs written in a high-level language or assembly language, and convert the program into machine code. High-level languages are **compiled**, and assembly language is **assembled**. Compilers work by first checking to make sure that every line in the program is valid (syntax error checks). Once it passes this step, the **source code**, or original program, is compiled into an **object code** file. Object code is merely the machine language equivalent of the source code. In some cases, a program might use the object code of other programs. For this, the process is next sent through a **linker**, combining the object code with any other required object code. This combined code is stored as an **executable file**. It’s the code in this file that the computer runs. A loader copies the executable file into memory, where it is then ran by the microprocessor in question. While the compiling of a high-level language may be complicated (there may be multiple ways to convert certain statements into machine language, which a compiler would have to choose from), assembly language compiles relatively easy, since it is direct machine instructions. This is the process as it relates to PCs and other computers. For less complicated machines, such as a microwave, less is needed.

Most software for PCs is written in a high-level language, due to their popularity, ease of use, and support. While assembly language is sometimes used in conjunction with high-level languages for some task, this path has its perils as well. While assembly language written code is smaller and faster than its high-level language counterpart, by combining high-level language programs with assembly language programs, the high-level language loses its ability to be supported cross-platform.

Assembly language instructions can be grouped together, depending on what they do. These categories or subclasses may differ from one author to another.

Data Transfer Instructions are the most common operations a microprocessor can perform. They don’t actually change the information; just copy it from one place to another. Types of data transfer instructions might include “load data from memory into the microprocessor”, “store data from the microprocessor into memory”, “move data within the microprocessor”, “input data to the microprocessor” or “output data from the microprocessor”. In addition, some assembly language instruction sets may contain specialized data transfer instructions.

Data Operation Instructions actually do modify their data values. They typically perform some operation using or two data values, or **operands**, to store the result. **Arithmetic instructions** make up a large part of the data operation instructions. These are your typical add/subtract/multiply/divide operations. A special class of arithmetic instructions is floating point instructions, which, obviously, operate on floating point numbers versus integers. Logical instructions perform basic logical operations on data. Logical instructions, like the logic gates from early, work in a bit-wise manner.

Program control instructions handle the flow management of an assembly language program. While it’s a mantra of high level programming languages to avoid a GOTO statement at any cost by using loops like FOREACH or DO…WHILE, assembly languages don’t have this option. Program control instructions allow a program to jump to another part of the program. A microprocessor can accept interrupts as well, which cause the processor to stop what its doing and begin executing something else. An assembly language routine may contain specific instructions for handling an interrupt. These are known as **software interrupts**. There are also interrupts that are triggered outside of the microprocessor, known as **hardware interrupts**. When an instruction does something invalid or performs invalid operations, such as dividing by zero, an **exception**, or **trap** is triggered. One final type of program control instruction is the halt instruction. This causes a microprocessor to stop executing instructions, such as at the end of a program.

Various data types available allow programs to work with a wide variety of data. Numeric data can often be expressed as an integer. Unsigned integers can have any value between 0 and 2n – 1. An n-bit signed integer can have any value between -2n-1 and 2n-1-1, inclusive.

Some numeric data must be expressed as floating point values if their value cannot be expressed without fractions. Some microprocessors may have special registers and instructions exclusively for floating point data.

The boolean values TRUE and FALSE are used often enough to warrant their own data type, **boolean**, and assembly language instructions. Typically TRUE is represented by a non-zero value and FALSE by a 0. Boolean instructions only generate one result, versus logical instructions, which generate one result per bit of the operand.

Computers must also deal with character data. Characters are stored as binary values encoded using **ASCII**, **EBCDIC**, **UNICODE**, or some other character encoding standard. Special instruction sets exist for dealing with character data, as characters and character strings may be concatenated, replaced, have a substring replaced, and other unique functions.

When working with data, a microprocessor must specify the memory address it needs to access. An assembly language instruction may use one of several **addressing modes** to generate this address. There are several modes available for doing this.

In direct mode, the instruction includes a memory address; the CPU accesses that location in memory. Indirect mode starts off like direct mode, but then performs a second memory access. The address specified is not the address of the operand; it is the address of a memory location that contains the address of the operand. This is used by compilers and operating systems with relocatable code and data. Register modes work the same as the others, except they do not specify a memory address. Instead, they specify a register. Like previously, there are register direct modes and register indirect modes which work accordingly the same as their namesakes. In the immediate mode, the operand specified is not an address; it is the actual data to be used. In implicit mode, an operand is not explicitly specified; the instruction implicitly specifies the operand because it always applies to a specific register. This isn’t usually used for load instructions. In relative mode, the operand supplied is an offset, not the actual address. It is added to the contents of the CPU’s program counter register to generate the required address. The program counter contains the address of the current instructions being executed, so the same relative instruction will produce different addresses at different locations in the program. Finally, there are index modes and base address modes. In index mode, which works like relative mode, except the address supplied the instructions is added to the contents of an index register instead of the program counter. Base address mode works exactly the same as index mode, except that the index register is replaced by a base address register. In theory, index mode instructions supply the base address and the index register supplies the offset to that base address.

Assembly language instructions converted to their binary counterpart are called **instruction code**. It is organized in a specific fashion. Different groups of bits represent different parts of the instruction. The operations to be performed are stored in the **opcode**, while other groups select the operands of the operation. Each instruction can have only one instruction code format. For example, in the operation A = B + C, there is one operation being performed, two source operands, B and C, and one destination operand, A. If a microprocessor can perform 16 different operations, addition being one of them, then it needs four bits to specify one of the operations (because 24 = 16). Here, we assume the bit pattern 1010 specifies addition. Also, assume that there are only four possible operands for this operation, A, B, C and D. The microprocessor needs two bits to specify each operand. For this example, 00 represents A, 01 represents B, 10 represents C, and 11 represents D. Processors may be designed to work with instructions that specify 3, 2, 1 or 0 operands.

With 3 operand instructions, a microprocessor instruction code must have bit fields for the opcode and three operands. It might require four bits for the opcode, and two bits for each of the three operands, for a total of ten bits. The program to perform A = B + C only requires a single instruction in this case. In 2 operand instructions, the first operand is both the destination and one of the source operands. A single 2 operand instruction cannot perform the operation A = B + C, but it can set A = A + B. A s a result, it may require several instructions to perform an operation. A 1 operand instruction format becomes more difficult. In 1 operand instructions, an accumulator register is always the destination and one of the source registers. This is implicit and not specified as part of the instruction code. Finally, with zero-operand instructions, all operands are drawn from a stack. The operands are pushed onto the stack, then pops its operands off the top of the stack, performs its operation, and pushes its result back onto the stack. To store its results, a zero-operand computer pops the result off the stack into the appropriate destination.

The design of an instruction set architecture is arguably the most important step in the design of a microprocessor. A good way to start is to ask the question “What should the instructions et architecture and its processor be able to do?” General purpose computing requires a fairly large instruction set to perform a wide variety of tasks, while a specialized processor must know the tasks needed to perform in advance in order to plan accordingly, and only these tasks are needed. This is the issue of **completeness** of the instruction set architecture; that is, does the instruction set have all of the instructions a program needs to perform its required task.

At the other end of the argument, instruction **orthogonality** comes into play. Instructions are orthogonal if they do not overlap, or perform the same function. Good instructions minimize the overlapping between instructions. Eliminating redundant instructions improve the orthogonality of instruction sets, while reducing the cost and complexity of the CPU. When looking at/designing ISAs, some issues to think about might be:

1. **Does this processor have to be backward compatible with other microprocessors?**

This is very important for general purpose processors, like the one in your home PC. If these types of processors were not backward compatible, software would have to be constantly rewritten for each new processor release.

1. **What types and sizes of data will the microprocessor deal with?**

The types of data used in the ISA will dictate what type of data instructions to include in the ISA. If a processor is only going to handle math-related functions, instructions to handle character data becomes less important, or unnecessary altogether.

1. **Are interrupts needed?**

Interrupts aren’t necessarily important to most tasks, but a few can be enhanced by their inclusion. Think back to the Dining Philosophers Problem for an example where interrupts might become useful, usually in situations with limited but frequently accessed resources.

1. **Are conditional instructions needed?**

Microprocessors generally include **flags**, or 1-bit registers to store values of various conditions that may require conditional jumps.

While it’s tempting to go into detail about the various examples in this chapter dealing with ISAs for various processors, there are entire courses on these subjects for both the Intel style processors and OS/390 assembly languages and instruction sets.