"Every program has at least one bug and can be shortened by at least one instruction—from which, by induction, one can deduce that every program can be reduced to one instruction which doesn't work."

—Anonymous

CHAPTER 5

A Closer Look at Instruction Set Architectures

5.1 INTRODUCTION

We saw in Chapter 4 that machine instructions consist of opcodes and operands. The opcodes specify the operations to be executed; the operands specify register or memory locations of data. Why, when we have languages such as C++, Java, and Ada available, should we be concerned with machine instructions? When programming in a high-level language, we frequently have little awareness of the topics discussed in Chapter 4 (or in this chapter) because high-level languages hide the details of the architecture from the programmer. Employers frequently prefer to hire people with assembly language backgrounds not because they need an assembly language programmer, but because they need someone who can understand computer architecture to write more efficient and more effective programs.

In this chapter, we expand on the topics presented in the last chapter, the objective being to provide you with a more detailed look at machine instruction sets. We look at different instruction types and operand types, and how instructions access data in memory. You will see that the variations in instruction sets are integral in distinguishing different computer architectures. Understanding how instruction sets are designed and how they function can help you understand the more intricate details of the architecture of the machine itself.

5.2 INSTRUCTION FORMATS

We know that a machine instruction has an opcode and zero or more operands. In Chapter 4 we saw that MARIE had an instruction length of 16 bits and could have, at most, 1 operand. Encoding an instruction set can be done in a variety of ways.

Architectures are differentiated from one another by the number of bits allowe per instruction (16, 32, and 64 are the most common), by the number of operand allowed per instruction, and by the types of instructions and data each can proces: More specifically, instruction sets are differentiated by the following features:

- Operand storage in the CPU (data can be stored in a stack structure or in registers
- Number of explicit operands per instruction (zero, one, two, and three bein the most common)
- Operand location (instructions can be classified as register-to-register, register to-memory, or memory-to-memory, which simply refer to the combinations o operands allowed per instruction)
- Operations (including not only types of operations but also which instruction can access memory and which cannot)
- Type and size of operands (operands can be addresses, numbers, or ever characters)

5.2.1 Design Decisions for Instruction Sets

When a computer architecture is in the design phase, the instruction set format must be determined before many other decisions can be made. Selecting this format is often quite difficult because the instruction set must match the architecture, and the architecture, if well designed, could last for decades. Decisions made during the design phase have long-lasting ramifications.

Instruction set architectures are measured by several different factors, including: (1) the amount of space a program requires; (2) the complexity of the instruction set, in terms of the amount of decoding necessary to execute an instruction, and the complexity of the tasks performed by the instructions; (3) the length of the instructions; and (4) the total number of instructions. Things to consider when designing an instruction set include:

- Short instructions are typically better because they take up less space in memory and can be fetched quickly. However, this limits the number of instructions, because there must be enough bits in the instruction to specify the number of instructions we need. Shorter instructions also have tighter limits on the size and number of operands.
- Instructions of a fixed length are easier to decode but waste space.
- Memory organization affects instruction format. If memory has, for example, 16- or 32-bit words and is not byte-addressable, it is difficult to access a single character. For this reason, even machines that have 16-, 32-, or 64-bit words are often byte-addressable, meaning every byte has a unique address even though words are longer than 1 byte.
- A fixed length instruction does not necessarily imply a fixed number of operands. We could design an ISA with fixed overall instruction length, but allow the number of bits in the operand field to vary as necessary. (This is called an **expanding opcode** and is covered in more detail in Section 5.2.5.)

- There are many different types of addressing modes. In Chapter 4, MARIE used two addressing modes: direct and indirect; however, we see in this chapter that a large variety of addressing modes exist.
- If words consist of multiple bytes, in what order should these bytes be stored on a byte-addressable machine? Should the least significant byte be stored at the highest or lowest byte address? This little versus big endian debate is discussed in the following section.
- How many registers should the architecture contain and how should these registers be organized? How should operands be stored in the CPU?

The little versus big endian debate, expanding opcodes, and CPU register organization are examined further in the following sections. In the process of discussing these topics, we also touch on the other design issues listed.

5.2.2 Little versus Big Endian

The term **endian** refers to a computer architecture's "byte order," or the way the computer stores the bytes of a multiple-byte data element. Virtually all computer architectures today are byte-addressable and must, therefore, have a standard for storing information requiring more than a single byte. Some machines store a two-byte integer, for example, with the least significant byte first (at the lower address) followed by the most significant byte. Therefore, a byte at a lower address has lower significance. These machines are called **little endian** machines. Other machines store this same two-byte integer with its most significant byte first, followed by its least significant byte. These are called **big endian** machines because they store the most significant bytes at the lower addresses. Most UNIX machines are big endian, whereas most PCs are little endian machines. Most newer RISC architectures are also big endian.

These two terms, little and big endian, are from the book *Gulliver's Travels*. You may remember the story in which the Lilliputians (the tiny people) were divided into two camps: those who ate their eggs by opening the "big" end (big endians) and those who ate their eggs by opening the "little" end (little endians). CPU manufacturers are also divided into two factions. For example, Intel has always done things the "little endian" way, whereas Motorola has always done things the "big endian" way. (It is also worth noting that some CPUs can handle both little and big endian.)

For example, consider an integer requiring 4 bytes:

On a little endian machine, this is arranged in memory as follows:

```
Base Address + 0 = Byte0
Base Address + 1 = Byte1
Base Address + 2 = Byte2
Base Address + 3 = Byte3
```

On a big endian machine, this long integer would then be stored as:

```
Base Address + 0 = Byte3
Base Address + 1 = Byte2
Base Address + 2 = Byte1
Base Address + 3 = Byte0
```

Let's assume that on a byte-addressable machine, the 32-bit hex value 12345678 is stored at address 0. Each digit requires a nibble, so one byte holds two digits. This hex value is stored in memory as shown in Figure 5.1, where the shaded cells represent the actual contents of memory.

There are advantages and disadvantages to each method, although one method is not necessarily better than the other. Big endian is more natural to most people and thus makes it easier to read hex dumps. By having the high-order byte come first, you can always test whether the number is positive or negative by looking at the byte at offset zero. (Compare this to little endian where you must know how long the number is and then must skip over bytes to find the one containing the sign information.) Big endian machines store integers and strings in the same order and are faster in certain string operations. Most bitmapped graphics are mapped with a "most significant bit on the left" scheme, which means working with graphical elements larger than one byte can be handled by the architecture itself. This is a performance limitation for little endian computers because they must continually reverse the byte order when working with large graphical objects. When decoding compressed data encoded with such schemes as Huffman and LZW (discussed in Chapter 7), the actual codeword can be used as an index into a lookup table if it is stored in big endian (this is also true for encoding).

However, big endian also has disadvantages. Conversion from a 32-bit integer address to a 16-bit integer address requires a big endian machine to perform addition. High-precision arithmetic on little endian machines is faster and easier. Most architectures using the big endian scheme do not allow words to be written on non-word address boundaries (for example, if a word is 2 or 4 bytes, it must always begin on an even-numbered byte address). This wastes space. Little endian architectures, such as Intel, allow odd address reads and writes, which makes programming on these machines much easier. If a programmer writes an instruction to read a value of the wrong word size, on a big endian machine it is always read as an incorrect value; on a little endian machine, it can sometimes result in the correct data being read. (Note that Intel finally has added an instruction to reverse the byte order within registers.)

Address ⟶	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

FIGURE 5.1 The Hex Value 12345678 Stored in Both Big and Little Endian Format

Computer networks are big endian, which means that when little endian computers are going to pass integers over the network (network device addresses, for example), they need to convert them to network byte order. Likewise, when they receive integer values over the network, they need to convert them back to their own native representation.

Although you may not be familiar with this little versus big endian debate, it is important to many current software applications. Any program that writes data to or reads data from a file must be aware of the byte ordering on the particular machine. For example, the Windows BMP graphics format was developed on a little endian machine, so to view BMPs on a big endian machine, the application used to view them must first reverse the byte order. Software designers of popular software are well aware of these byte-ordering issues. For example, Adobe Photoshop uses big endian, GIF is little endian, JPEG is big endian, MacPaint is big endian, PC Paintbrush is little endian, RTF by Microsoft is little endian, and Sun raster files are big endian. Some applications support both formats: Microsoft WAV and AVI files, TIF files, and XWD (X windows Dump) support both, typically by encoding an identifier into the file.

5.2.3 Internal Storage in the CPU: Stacks versus Registers

Once byte ordering in memory is determined, the hardware designer must make some decisions on how the CPU should store data. This is the most basic means to differentiate ISAs. There are three choices:

1. A stack architecture
2. An accumulator architecture
3. A general-purpose register (GPR) architecture

Stack architectures use a stack to execute instructions, and the operands are (implicitly) found on top of the stack. Even though stack-based machines have good code density and a simple model for evaluation of expressions, a stack cannot be accessed randomly, which makes it difficult to generate efficient code. In addition, the stack becomes a bottleneck during execution. **Accumulator architectures** such as MARIE, with one operand implicitly in the accumulator minimize the internal complexity of the machine and allow for very short instructions. But because the accumulator is only temporary storage, memory traffic is very high. **General-purpose register architectures**, which use sets of general-purpose registers, are the most widely accepted models for machine architectures today. These register sets are faster than memory, easy for compilers to deal with, and can be used very effectively and efficiently. In addition, hardware prices have decreased significantly, making it possible to add a large number of registers at a minimal cost. If memory access is fast, a stack-based design may be a good idea; if memory is slow, it is often better to use registers. These are the reasons why most computers over the past 10 years have been general-register based. However, because all operands must be named, using registers results in longer instructions, causing longer fetch and decode times. (A very important goal for ISA designers is short instructions.) Designers choosing an

ISA must decide which will work best in a particular environment and examine the tradeoffs carefully.

The general-purpose architecture can be broken into three classifications, depending on where the operands are located. **Memory-memory** architectures may have two or three operands in memory, allowing an instruction to perform an operation without requiring any operand to be in a register. **Register-memory** architectures require a mix, where at least one operand is in a register and one is in memory. **Load-store** architectures require data to be moved into registers before any operations on those data are performed. Intel and Motorola are examples of register-memory architectures; Digital Equipment's VAX architecture allows memory-memory operations; and SPARC, MIPS, Alpha, and the PowerPC are all load-store machines.

Given that most architectures today are GPR-based, we now examine two major instruction set characteristics that divide general-purpose register architectures. Those two characteristics are the number of operands and how the operands are addressed. In Section 5.2.4 we look at the instruction length and number of operands an instruction can have. (Two or three operands are the most common for GPR architectures, and we compare these to zero and one operand architectures.) We then investigate instruction types. Finally, in Section 5.4 we investigate the various addressing modes available.

5.2.4 Number of Operands and Instruction Length

The traditional method for describing a computer architecture is to specify the maximum number of operands, or addresses, contained in each instruction. This has a direct impact on the length of the instruction itself. MARIE uses a fixed-length instruction with a 4-bit opcode and a 12-bit operand. Instructions on current architectures can be formatted in two ways:

- **Fixed length**—Wastes space but is fast and results in better performance when instruction-level pipelining is used, as we see in Section 5.5.
- **Variable length**—More complex to decode but saves storage space.

Typically, the real-life compromise involves using two to three instruction lengths, which provides bit patterns that are easily distinguishable and simple to decode. The instruction length must also be compared to the word length on the machine. If the instruction length is exactly equal to the word length, the instructions align perfectly when stored in main memory. Instructions always need to be word aligned for addressing reasons. Therefore, instructions that are half, quarter, double, or triple the actual word size can waste space. Variable length instructions are clearly not the same size and need to be word aligned, resulting in loss of space as well.

The most common instruction formats include zero, one, two, or three operands. We saw in Chapter 4 that some instructions for MARIE have no operands, whereas others have one operand. Arithmetic and logic operations typically have two operands, but can be executed with one operand (as we saw in MARIE), if the accumulator is implicit. We can extend this idea to three operands

if we consider the final destination as a third operand. We can also use a stack that allows us to have zero operand instructions. The following are some common instruction formats:

- **OPCODE only** (zero addresses)
- **OPCODE + 1 Address** (usually a memory address)
- **OPCODE + 2 Addresses** (usually registers, or one register and one memory address)
- **OPCODE + 3 Addresses** (usually registers, or combinations of registers and memory)

All architectures have a limit on the maximum number of operands allowed per instruction. For example, in MARIE, the maximum was one, although some instructions had no operands (`Halt` and `Skipcond`). We mentioned that zero-, one-, two-, and three-operand instructions are the most common. One-, two-, and even three-operand instructions are reasonably easy to understand; an entire ISA built on zero-operand instructions can, at first, be somewhat confusing.

Machine instructions that have no operands must use a stack (the last-in, first-out data structure, introduced in Chapter 4 and described in detail in Appendix A, where all insertions and deletions are made from the top) to perform those operations that logically require one or two operands (such as an `Add`). Instead of using general purpose registers, a stack-based architecture stores the operands on the top of the stack, making the top element accessible to the CPU. (Note that one of the most important data structures in machine architectures is the stack. Not only does this structure provide an efficient means of storing intermediate data values during complex calculations, but it also provides an efficient method for passing parameters during procedure calls as well as a means to save local block structure and define the scope of variables and subroutines.)

In architectures based on stacks, most instructions consist of opcodes only; however, there are special instructions (those that add elements to and remove elements from the stack) that have just one operand. Stack architectures need a push instruction and a pop instruction, each of which is allowed one operand. `Push X` places the data value found at memory location *X* onto the stack; `Pop X` removes the top element in the stack and stores it at location *X*. Only certain instructions are allowed to access memory; all others must use the stack for any operands required during execution.

For operations requiring two operands, the top two elements of the stack are used. For example, if we execute an `Add` instruction, the CPU adds the top two elements of the stack, popping them both and then pushing the sum onto the top of the stack. For noncommutative operations such as subtraction, the top stack element is subtracted from the next-to-the-top element, both are popped, and the result is pushed onto the top of the stack.

This stack organization is very effective for evaluating long arithmetic expressions written in **reverse Polish notation** (**RPN**). This representation places the operator after the operands in what is known as **postfix notation** (as compared to **infix notation**, which places the operator between operands, and **prefix** [illegible] which places the operator before the operands). For example: