X + Y is in infix notation
+ X Y is in prefix notation
X Y + is in postfix notation

All arithmetic expressions can be written using any of these representations. However, postfix representation combined with a stack of registers is the most efficient means to evaluate arithmetic expressions. In fact, some electronic calculators (such as Hewlett-Packard) require the user to enter expressions in postfix notation. With a little practice on these calculators, it is possible to rapidly evaluate long expressions containing many nested parentheses without ever stopping to think about how terms are grouped.

Consider the following expression:

$$(X + Y) \times (W - Z) + 2$$

Written in RPN, this becomes:

$$XY + WZ - \times 2+$$

Notice that the need for parentheses to preserve precedence is eliminated when using RPN.

To illustrate the concepts of zero, one, two, and three operands, let's write a simple program to evaluate an arithmetic expression, using each of these formats.

EXAMPLE 5.1 Suppose we wish to evaluate the following expression:

$$Z = (X \times Y) + (W \times U)$$

Typically, when three operands are allowed, at least one operand must be a register, and the first operand is normally the destination. Using three-address instructions, the code to evaluate the expression for *Z* is written as follows:

```
Mult    R1, X, Y
Mult    R2, W, U
Add     Z, R2, R1
```

When using two-address instructions, normally one address specifies a register (two-address instructions seldom allow for both operands to be memory addresses). The other operand could be either a register or a memory address. Using two-address instructions, our code becomes:

```
Load    R1, X
Mult    R1, Y
Load    R2, W
Mult    R2, U
Add     R1, R2
Store   Z, R1
```

Note that it is important to know whether the first operand is the source or the destination. In the above instructions, we assume it is the destination. (This tends to be a point of confusion for those programmers who must switch between Intel assembly language and Motorola assembly language—Intel assembly specifies the first operand as the destination, whereas in Motorola assembly, the first operand is the source.)

Using one-address instructions (as in MARIE), we must assume a register (normally the accumulator) is implied as the destination for the result of the instruction. To evaluate *Z*, our code now becomes:

```
Load    X
Mult    Y
Store   Temp
Load    W
Mult    U
Add     Temp
Store   Z
```

Note that as we reduce the number of operands allowed per instruction, the number of instructions required to execute the desired code increases. This is an example of a typical space/time trade-off in architecture design—shorter instructions but longer programs.

What does this program look like on a stack-based machine with zero-address instructions? Stack-based architectures use no operands for instructions such as `Add`, `Subt`, `Mult`, or `Divide`. We need a stack and two operations on that stack: `Pop` and `Push`. Operations that communicate with the stack must have an address field to specify the operand to be popped or pushed onto the stack (all other operations are zero-address). `Push` places the operand on the top of the stack. `Pop` removes the stack top and places it in the operand. This architecture results in the longest program to evaluate our equation. Assuming arithmetic operations use the two operands on the stack top, pop them, and push the result of the operation, our code is as follows:

```
Push    X
Push    Y
Mult
Push    W
Push    U
Mult
Add
Pop     Z
```

The instruction length is certainly affected by the opcode length and by the number of operands allowed in the instruction. If the opcode length is fixed, decoding is much easier. However, to provide for backward compatibility and flexibility, opcodes can have variable length. Variable length opcodes present the same problems as variable versus constant length instructions. A compromise used by many designers is expanding opcodes.

5.2.5 Expanding Opcodes

Expanding opcodes represent a compromise between the need for a rich set of opcodes and the desire to have short opcodes, and thus short instructions. The idea is to make some opcodes short, but have a means to provide longer ones when needed. When the opcode is short, a lot of bits are left to hold operands (which means we could have two or three operands per instruction). When you don't need any space for operands (for an instruction such as `Halt` or because the machine uses a stack), all the bits can be used for the opcode, which allows for many unique instructions. In between, there are longer opcodes with fewer operands as well as shorter opcodes with more operands.

Consider a machine with 16-bit instructions and 16 registers. Because we now have a register set instead of one simple accumulator (as in MARIE), we need to use 4 bits to specify a unique register. We could encode 16 instructions, each with 3 register operands (which implies any data to be operated on must first be loaded into a register), or use 4 bits for the opcode and 12 bits for a memory address (as in MARIE, assuming a memory of size 4K). Any memory reference requires 12 bits, leaving only 4 bits for other purposes. However, if all data in memory is first loaded into a register in this register set, the instruction can select that particular data element using only 4 bits (assuming 16 registers). These two choices are illustrated in Figure 5.2.

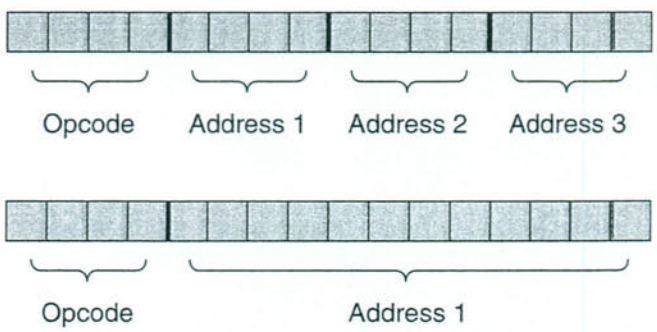

FIGURE 5.2 Two Possibilities for a 16-Bit Instruction Format

Suppose we wish to encode the following instructions:

- 15 instructions with 3 addresses
- 14 instructions with 2 addresses
- 31 instructions with 1 address
- 16 instructions with 0 addresses

Can we encode this instruction set in 16 bits? The answer is yes, as long as we use expanding opcodes. The encoding is as follows:

```
0000 R1   R2   R3
...                          } 15 3-address codes
1110 R1   R2   R3

1111 0000 R1   R2
...                          } 14 2-address codes
1111 1101 R1   R2

1111 1110 0000 R1
...                          } 31 1-address codes
1111 1111 1110 R1

1111 1111 1111 0000
...                          } 16 0-address codes
1111 1111 1111 1111
```

This expanding opcode scheme makes the decoding more complex. Instead of simply looking at a bit pattern and deciding which instruction it is, we need to decode the instruction something like this:

```
if (leftmost four bits != 1111 ) {
   Execute appropriate three-address instruction}
else if (leftmost seven bits != 1111 111 ) {
   Execute appropriate two-address instruction}
else if (leftmost twelve bits != 1111 1111 1111 ) {
   Execute appropriate one-address instruction }
else {
   Execute appropriate zero-address instruction
}
```

At each stage, one spare code is used to indicate that we should now look at more bits. This is another example of the types of trade-offs hardware designers continually face: Here, we trade opcode space for operand space.

5.3 INSTRUCTION TYPES

Most computer instructions operate on data; however, there are some that do not. Computer manufacturers regularly group instructions into the following categories: data movement, arithmetic, Boolean, bit manipulation (shift and rotate), I/O, transfer of control, and special purpose.

We discuss each of these categories in the following sections.

5.3.1 Data Movement

Data movement instructions are the most frequently used instructions. Data is moved from memory into registers, from registers to registers, and from registers to memory, and many machines provide different instructions depending on the source and destination. For example, there may be a `MOVER` instruction that always requires two register operands, whereas a `MOVE` instruction allows one register and one memory operand. Some architectures, such as RISC, limit the instructions that can move data to and from memory in an attempt to speed up execution. Many machines have variations of load, store, and move instructions to handle data of different sizes. For example, there may be a `LOADB` instruction for dealing with bytes and a `LOADW` instruction for handling words. Data movement instructions include `MOVE`, `LOAD`, `STORE`, `PUSH`, `POP`, `EXCHANGE`, and multiple variations on each of these.

5.3.2 Arithmetic Operations

Arithmetic operations include those instructions that use integers and floating point numbers. Many instruction sets provide different arithmetic instructions for various data sizes. As with the data movement instructions, there are sometimes different instructions for providing various combinations of register and memory accesses in different addressing modes. Instructions may exist for arithmetic operations on both signed and unsigned numbers, as well as for operands in different bases. Many times, operands are implied in arithmetic instructions. For example, a multiply instruction may assume the multiplicand is resident in a specific register so it need not be explicitly given in the instruction. This class of instructions also affects the flag register, setting the zero, carry, and overflow bits (to name only a few). Arithmetic instructions include `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `INCREMENT`, `DECREMENT`, and `NEGATE` (to change the sign).

5.3.3 Boolean Logic Instructions

Boolean logic instructions perform Boolean operations, much in the same way that arithmetic operations work. These instructions allow bits to be set, cleared, and complemented. Logic operations are commonly used to control I/O devices. As with arithmetic operations, logic instructions affect the flag register, including the carry and overflow bits. There are typically instructions for performing AND, NOT, OR, XOR, TEST, and COMPARE.

5.3.4 Bit Manipulation Instructions

Bit manipulation instructions are used for setting and resetting individual bits (or sometimes groups of bits) within a given data word. These include both arithmetic and logical SHIFT instructions and ROTATE instructions, each to the left and to the right. Logical shift instructions simply shift bits to either the left or the right by a specified number of bits, shifting in zeros on the opposite end. For example, if we have an 8-bit register containing the value 11110000, and we perform a logical shift left by one bit, the result is 11100000. If our register contains 11110000 and we perform a logical shift right by one bit, the result is 01111000.

Arithmetic shift instructions, commonly used to multiply or divide by 2, treat data as signed two's complement numbers, and do not shift the leftmost bit, since this represents the sign of the number. On a right arithmetic shift, the sign bit is replicated into the bit position(s) to its right: if the number is positive, the leftmost bits are filled by zeros; if the number is negative, the leftmost bits are filled by ones. A right arithmetic shift is equivalent to division by two. For example, if our value is 00001110 (+14) and we perform an arithmetic shift right by one bit, the result is 00000111 (+7). If the value is negative, such as 11111110 (−2), the result is 11111111 (-1). On a left arithmetic shift, bits are shifted left, zeros are shifted in, but the sign bit does not participate in the shifting. An arithmetic shift left is equivalent to multiplication by 2. For example, if our register contains 00000011 (+3), and we perform an arithmetic shift left, one bit, the result is 00000110 (+6). If the register contains a negative number such as 11111111 (−1), performing an arithmetic shift left by one bit yields 11111110 (−2). If the last bit shifted out (excluding the sign bit) does not match the sign, overflow or underflow occurs. For example, if the number is 10111111 (−65) and we do an arithmetic shift left by one bit, the result is 11111110 (−2), but the bit that was "shifted out" is a zero and does not match the sign; hence we have overflow.

Rotate instructions are simply shift instructions that shift in the bits that are shifted out—a circular shift basically. For example, on a rotate left one bit, the leftmost bit is shifted out and rotated around to become the rightmost bit. If the value 00001111 is rotated left by one bit, we have 00011110. If 00001111 is

rotated right by one bit, we have 10000111. With rotate, we do not worry about the sign bit.

In addition to shifts and rotates, some computer architectures have instructions for clearing specific bits, setting specific bits, and toggling specific bits.

5.3.5 Input/Output Instructions

I/O instructions vary greatly from architecture to architecture. The input (or read) instruction transfers data from a device or port to either memory or a specific register. The output (or write) instruction transfers data from a register or memory to a specific port or device. There may be separate I/O instructions for numeric data and character data. Generally, character and string data use some type of block I/O instruction, automating the input of a string. The basic schemes for handling I/O are programmed I/O, interrupt-driven I/O, and DMA devices. These are covered in more detail in Chapter 7.

5.3.6 Instructions for Transfer of Control

Control instructions are used to alter the normal sequence of program execution. These instructions include branches, skips, procedure calls, returns, and program termination. Branching can be unconditional (such as jump) or conditional (such as jump on condition). Skip instructions (which can also be conditional or unconditional) are basically branch instructions with implied addresses. Because no operand is required, skip instructions often use bits of the address field to specify different situations (recall the `Skipcond` instruction used by MARIE). Some languages include looping instructions that automatically combine conditional and unconditional jumps.

Procedure calls are special branch instructions that automatically save the return address. Different machines use different methods to save this address. Some store the address at a specific location in memory, others store it in a register, while still others push the return address on a stack.

5.3.7 Special Purpose Instructions

Special purpose instructions include those used for string processing, high-level language support, protection, flag control, word/byte conversions, cache management, register access, address calculation, no-ops, and any other instructions that don't fit into the previous categories. Most architectures provide instructions for string processing, including string manipulation and searching. No-op instructions, which take up space and time but reference no data and basically do nothing, are often used as placeholders for insertion of useful instructions at a later time, or in pipelines (see Section 5.5).

5.3.8 Instruction Set Orthogonality

Regardless of whether an architecture is hard-coded or microprogrammed, it is important that the architecture have a complete instruction set. However, designers must be careful not to add redundant instructions, as each instruction trans-