Direct extensions such as the ISA bus are fairly easy to implement and serve well in applications where I/O response time does not unduly restrict microprocessor throughput. As computers have gotten faster, the throughput of microprocessors has rapidly outstripped the response times of all but the fastest I/O devices. In comparison to a modern microprocessor, a hard-disk controller is rather slow, with response times measured in microseconds rather than nanoseconds. Additionally, as bus signals become faster, the permissible length of interconnecting wires decreases, limiting their expandability. These and other characteristics motivate the decoupling of the microprocessor's local bus from the computer's I/O bus.

An I/O bus can be decoupled from the microprocessor bus by inserting an intermediate bus controller between them that serves as an interface, or translator, between the two buses. Once the buses are separated, activity on one bus does not necessarily obstruct activity on the other. If the microprocessor wants to write a block of data to a slow device, it can rapidly transfer that data to the bus controller and then continue with other operations at full speed while the controller slowly transfers the data to the I/O device. This mechanism is called a *posted-write,* because the bus controller allows the microprocessor to complete, or *post*, its write before the write actually completes. Separate buses also open up the possibility of multiple microprocessors or logic elements performing I/O operations without conflicting with the central microprocessor. In a multimaster system, a specialized DMA controller can transfer data between two peripherals such as disk controllers while the microprocessor goes about its normal business.

The *Peripheral Component Interconnect* (PCI) bus is the industry-standard follow-on to the ISA bus, and it implements such advanced features as posted-writes, multiple-masters, and multiple bus segments. Each PCI bus segment is separated from the others via a PCI bridge chip. Only traffic that must travel between buses crosses a bridge, thereby reducing congestion on individual PCI bus segments. One segment can be involved in a data transfer between two devices without affecting a simultaneous transfer between two other devices on a different segment. These performance-enhancing features do not come for free, however. Their cost is manifested by the need for dedicated PCI control logic in bridge chips and in the I/O devices themselves. It is generally simpler to implement an I/O device that is directly mapped into the microprocessor's memory space, but the overall performance of the computer may suffer under demanding applications.

3.9 ASSEMBLY LANGUAGE AND ADDRESSING MODES

With the hardware ready, a computer requires software to make it more than an inactive collection of components. Microprocessors fetch instructions from program memory, each consisting of an opcode and, optionally, additional operands following the opcode. These opcodes are binary data that are easy for the microprocessor to decode, but they are not very readable by a person. To enable a programmer to more easily write software, an instruction representation called *assembly language* was developed. Assembly language is a low-level language that directly represents each binary opcode with a human-readable text mnemonic. For example, the mnemonic for an unconditional branch-to-subroutine instruction could be `BSR`. In contrast, a high-level language such as C++ or Java contains more complex logical expressions that may be automatically converted by a compiler to dozens of microprocessor instructions. Assembly language programs are assembled, rather than compiled, into opcodes by directly translating each mnemonic into its binary equivalent.

Assembly language also makes programming easier by enabling the usage of text labels in place of hard-coded addresses. A subroutine can be named `FOO`, and when `BSR FOO` is encountered by the assembler, a suitable branch target address will be automatically calculated in place of the label `FOO`. Each type of assembler requires a slightly different format and syntax, but there are general assembly language conventions that enable a programmer to quickly adapt to specific implementations

once the basics are understood. An assembly language program listing usually has three columns of text followed by an optional comment column as shown in Fig. 3.14. The first column is for labels that are placeholders for addresses to be resolved by the assembler. Instruction mnemonics are located in the second column. The third column is for instruction operands.

This listing uses the Motorola 6800 family's assembly language format. Though developed in the 1970s, 68xx microprocessors are still used today in embedded applications such as automobiles and industrial automation. The first line of this listing is not an instruction, but an assembler *directive* that tells the assembler to locate the program at memory location $100. When assembled, the listing is converted into a memory dump that lists a range of memory addresses and their corresponding contents—opcodes and operands. Assembler directives are often indicated with a period prefix.

The program in Fig. 3.14 is very simple: it counts to 30 ($1E) and then sends the "Z" character out the serial port. It continues in an infinite loop by returning to the start of the program when the serial port routine has completed its task. The subroutine to handle the serial port is not shown and is referenced with the `SEND_CHAR` label. The program begins by clearing accumulator A (the 6800 has two accumulators: ACCA and ACCB). It then enters an incrementing loop where the accumulator is incremented and then compared against the terminal count value, $1E. The # prefix tells the assembler to use the literal value $1E for the comparison. Other alternatives are possible and will soon be discussed. If ACCA is unequal to $1E, the microprocessor goes back to increment ACCA. If equal, the accumulator is loaded with the ASCII character to be transmitted, also a literal operand. The assumption here is that the `SEND_CHAR` subroutine transmits whatever is in ACCA. When the subroutine finishes, the program starts over with the branch-always instruction.

Each of the instructions in the preceding program contains at least one operand. `CLRA` and `INCA` have only one operand: ACCA. `CMPA` and `LDAA` each have two operands: ACCA and associated data. Complex microprocessors may reference three or more operands in a single instruction. Some instructions can reference different types of operands according to the requirements of the program being implemented. Both `CMPA` and `LDAA` reference literal operands in this example, but a programmer cannot always specify a predetermined literal data value directly in the instruction sequence.

Operands can be referenced in a variety of manners, called *addressing modes,* depending on the type of instruction and the type of operand. Some types of instructions inherently use only one addressing mode, and some types have multiple modes. The manners of referencing operands can be categorized into six basic addressing modes: *implied*, *immediate*, *direct*, *relative*, *indirect*, and *indexed*. To fully understand how a microprocessor works, and to efficiently utilize an instruction set, it is necessary to explore the various mechanisms used to reference data.

- *Implied addressing* specifies the operand of an instruction as an inherent property of that instruction. For example, `CLRA` implies the accumulator by definition. No additional addressing information following the opcode is needed.

```
                .ORIG       $100

BEGIN           CLRA
INC_LOOP        INCA
                CMPA        #$1E          ; compare ACCA = $1E
                BNE         INC_LOOP      ; if not equal, go back
                LDAA        #'Z'          ; else, load ASCII 'Z'
                BSR         SEND_CHAR     ; send ACCA to serial port
                BRA         BEGIN         ; start over again
```

FIGURE 3.14 Typical assembly language listing.