- *Immediate addressing* places an operand's value literally into the instruction sequence. `LDAA #'Z'` has its primary operand immediately available following the opcode. An immediate operand is indicated with the # prefix in some assembly languages. Eight-bit microprocessors with eight-bit instruction words cannot fit an immediate value into the instruction word itself and, therefore, require that an extra byte following the opcode be used to specify the immediate value. More powerful 32-bit microprocessors can often fit a 16-bit or 24-bit immediate value within the instruction word. This saves an additional memory fetch to obtain the operand.

- *Direct addressing* places the address of an operand directly into the instruction sequence. Instead of specifying `LDAA #'Z'`, the programmer could specify `LDAA $1234`. This version of the instruction would tell the microprocessor to read memory location $1234 and load the resulting value into the accumulator. The operand is directly available by looking into the memory address specified just following the instruction. Direct addressing is useful when there is a need to read a fixed memory location. Usage of the direct addressing mode has a slightly different impact on various microprocessors. A typical 8-bit microprocessor has a 16-bit address space, meaning that two bytes following the opcode are necessary to represent a direct address. The 8-bit microprocessor will have to perform two additional 8-bit fetch operations to load the direct address. A typical 32-bit microprocessor has a 32-bit address space, meaning that 4 bytes following the opcode are necessary. If the 32-bit microprocessor has a 32-bit data bus, only one additional 32-bit fetch operation is required to load the direct address.

- *Relative addressing* places an operand's relative address into the instruction sequence. A relative address is expressed as a signed offset relative to the current value of the PC. Relative addressing is often used by branch instructions, because the target of a branch is usually within a short distance of the PC, or current instruction. For example, `BNE INC_LOOP` results in a branch-if-not-equal backward by two instructions. The assembler automatically resolves the addresses and calculates a relative offset to be placed following the `BNE` opcode. This relative operation is performed by adding the offset to the PC. The new PC value is then used to resume the instruction fetch and execution process. Relative addressing can utilize both positive and negative deltas that are applied to the PC. A microprocessor's instruction format constrains the relative range that can be specified in this addressing mode. For example, most 8-bit microprocessors provide only an 8-bit signed field for relative branches, indicating a range of +127/–128 bytes. The relative delta value is stored into its own byte just after the opcode. Many 32-bit microprocessors allow a 16-bit delta field and are able to fit this value into the 32-bit instruction word, enabling the entire instruction to be fetched in a single memory read. Limiting the range of a relative operation is generally not an excessive constraint because of software's *locality* property. *Locality* in this context means that the set of instructions involved in performing a specific task are generally relatively close together in memory. The locality property covers the great majority of branch instructions. For those few branches that have their targets outside of the allowed relative range, it is necessary to perform a short relative branch to a long jump instruction that specifies a direct address. This reduces the efficiency of the microprocessor by having to perform two branches when only one is ideally desired, but the overall efficiency of saving extra memory accesses for the majority of short branches is worth the trade-off.

- *Indirect addressing* specifies an operand's direct address as a value contained in another register. The other register becomes a pointer to the desired data. For example, a microprocessor with two accumulators can load *ACCA* with the value that is at the address in ACCB. `LDAA (ACCB)` would tell the microprocessor to put the value of accumulator B onto the address bus, perform a read, and put the returned value into accumulator A. Indirect addressing allows writing software routines that operate on data at different addresses. If a programmer wants to read or write an arbi-

trary entry in a data table, the software can load the address of that entry into a microprocessor register and then perform an indirect access using that register as a pointer. Some microprocessors place constraints on which registers can be used as references for indirect addressing. In the case of a 6800 microprocessor, `LDAA (ACCB)` is not actually a supported operation but serves as a syntactical example for purposes of discussion.

- *Indexed addressing* is a close relative (no pun intended) of indirect addressing, because it also refers to an address contained in another register. However, indexed addressing also specifies an offset, or index, to be added to that register base value to generate the final operand address: base + offset = final address. Some microprocessors allow general accumulator registers to be used as base-address registers, but others, such as the 6800, provide special *index registers* for this purpose. In many 8-bit microprocessors, a full 16-bit address cannot be obtained from an 8-bit accumulator serving as the base address. Therefore, one or more separate index registers are present for the purpose of indexed addressing. In contrast, many 32-bit microprocessors are able to specify a full 32-bit address with any general-purpose register and place no limitations on which register serves as the index register. Indexed addressing builds upon the capabilities of indirect addressing by enabling multiple address offsets to be referenced from the same base address. `LDAA (X+$20)` would tell the microprocessor to add $20 to the index register, X, and use the resulting address to fetch data to be loaded into ACCA. One simple example of using indexed addressing is a subroutine to add a set of four numbers located at an arbitrary location in memory. Before calling the subroutine, the main program can set an index register to point to the table of numbers. Within the subroutine, four individual addition instructions use the indexed addressing mode to add the locations X+0, X+1, X+2, and X+3. When so written, the subroutine is flexible enough to be used for any such set of numbers. Because of the similarity of indexed and indirect addressing, some microprocessors merge them into a single mode and obtain indirect addressing by performing indexed addressing with an index value of zero.

The six conceptual addressing modes discussed above represent the various logical mechanisms that a microprocessor can employ to access data. It is important to realize that each individual microprocessor applies these addressing modes differently. Some combine multiple modes into a single mode (e.g., indexed and indirect), and some will create multiple submodes out of a single mode. The exact variation depends on the specifics of an individual microprocessor's architecture.

With the various addressing modes modifying the specific opcode and operands that are presented to the microprocessor, the benefits of using assembly language over direct binary values can be observed. The programmer does not have to worry about calculating branch target addresses or resolving different addressing modes. Each mnemonic can map to several unique opcodes, depending on the addressing mode used. For example, the `LDAA` instruction in Fig. 3.14 could easily have used *extended* addressing by specifying a full 16-bit address at which the ASCII transmit-value is located. Extended addressing is the 6800's mechanism for specifying a 16-bit direct address. (The 6800's direct addressing involves only an eight-bit address.) In either case, the assembler would determine the correct opcode to represent `LDAA` and insert the correct binary values into the memory dump. Additionally, because labels are resolved each time the program is assembled, small changes to the program can be made that add or remove instructions and labels, and the assembler will automatically adjust the resulting addresses accordingly.

Programming in assembly language is different from using a high-level language, because one must think in smaller steps and have direct knowledge about the microprocessor's operation and architecture. Assembly language is processor-specific instead of generic, as with a high-level language. Therefore, assembly language programming is usually restricted to special cases such as boot code or routines in which absolute efficiency and performance are demanded. A human programmer will usually be able to write more efficient assembly language than a high-level language compiler