Every computer has RWM for storing variable data. Depending on the computer and the application, the programs may also be stored in RWM, or they may be stored in ROM, PROM, or EPROM.

All of the memories described above are *random-access memories*, because all locations have equally fast access. However, computer jargon has developed so that the acronym "RAM" most commonly refers to read/write memory only. To be correct, use "RWM" when writing about read/write memory, but pronounce it "ram" to keep your tongue intact!

## 5.2 ACCUMULATOR-BASED PROCESSORS

The simplest processor organization has only one or two registers, called *accumulators*, in which arithmetic and logical operations and data transfers take place. The Intel MCS-48 is a single-accumulator processor; the Motorola 6809 is a two-accumulator processor. Processors with more than two registers for arithmetic and logical operations are classified as general-register processors and are discussed in the next section.

### 5.2.1 Organization of a Single-Accumulator Processor

Figure 5–2 shows the internal organization of a single-accumulator processor. This hypothetical processor has only one accumulator and a subset of the other registers and of the instructions of the Motorola 6809 (Chapter 17), so we'll call it the H6809.

The H6809 accesses a memory of up to $2^{16}$ (65,536) bytes, arranged as shown in Figure 5–1(a); addresses are two bytes long. Most instructions manipulate 1-byte quantities; a few process 2-byte quantities. An H6809 instruction occupies one, two, or three bytes in memory. The processor has several registers and functional units, briefly described on the following page:

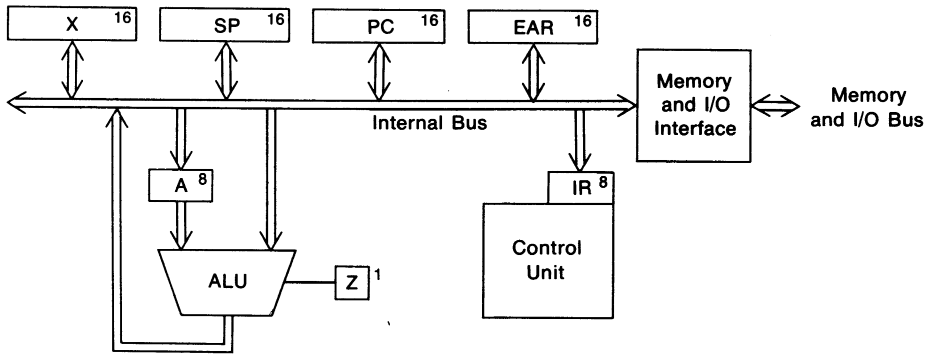

**FIGURE 5–2** Single-accumulator processor organization, the H6809.

- *Instruction Register (IR):* an 8-bit register that holds the first byte of the currently executing instruction.
- *Effective Address Register (EAR):* a 16-bit register that holds an address at which the processor reads or writes memory during the execution of an instruction.
- *Program Counter (PC):* a 16-bit register that holds the memory address of the next instruction to be executed.
- *Accumulator (A):* an 8-bit register containing data to be processed.
- *Index Register (X):* a 16-bit register containing an address or 16-bit data used by a program.
- *Stack Pointer (SP):* a 16-bit register containing the address of the top of a return-address stack in memory.
- *Zero Bit (Z):* a 1-bit register that the processor sets during the execution of each data manipulation instruction, to 1 if the instruction produces a zero result, to 0 if the instruction produces a nonzero result.
- *Arithmetic and Logic Unit (ALU):* combines two 8-bit quantities to produce an 8-bit result.
- *Control Unit:* decodes instructions and controls the other blocks to fetch and execute instructions.
- *Memory and I/O Interface:* reads and writes memory and communicates with I/O devices according to commands from the Control Unit.

Although all of the blocks above are essential to the internal operation of the H6809, only the registers PC, A, X, SP, and Z are explicitly manipulated by instructions and have values that are meaningful after each instruction's execution. Such registers comprise the *processor state*, and may be shown in a *programming model* for the processor, as in Figure 5–3. Only these registers are of concern to a programmer.

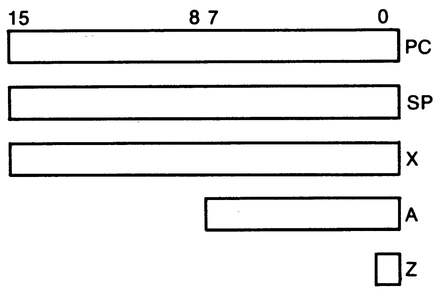

**FIGURE 5–3** Programming model for H6809.

### 5.2.2 Basic Instruction Cycle

The operation of the H6809 (or almost any other computer processor) consists of endless repetition of two steps: read the next instruction from memory (the *fetch cycle*), and perform the actions it requires (the *execution cycle*). This basic instruction cycle may be defined by the Pascal simulation in Table 5–1. The fetch cycle simply reads the first byte of the instruction that PC points to, and increments PC to point to the next byte in memory. The operations performed during the execution cycle depend on the instruction, and may include reading additional instruction bytes and updating PC accordingly. The use of a counter for PC is no accident. As observed by von Neumann,

> It is clear that one must be able to get numbers from any part of the memory at any time. The treatment in the case of orders can, however, be more methodical since one can at least partially arrange the control instructions in a linear sequence. Consequently the control will be so constructed that it will normally proceed from place *n* in the memory to place $(n+1)$ for its next instruction.

### 5.2.3 Machine Instructions

The "machine instructions" of the hypothetical H6809 are one to three bytes long. The first byte of each instruction is called the *opcode*; it uniquely

**TABLE 5–1** Basic instruction cycle of the H6809 processor.

```
PROGRAM H6809 (input,output);
TYPE byte = ARRAY [7::0] OF bit;
  word = ARRAY [15::0] OF bit;
VAR MEM : ARRAY [0..65535] OF byte;
  EAR, PC, X, SP : word;
  IR, A : byte; Z : bit;

PROCEDURE Fetch;
  BEGIN
    IR := MEM[PC];  {Read next instruction.}
    PC := PC + 1;  {Bump PC to next instruction.}
  END;

PROCEDURE Execute;
  BEGIN  {Will be defined later.}  END;

BEGIN
  PC := 0;  {Start at PC=0 on cold start.}
  WHILE true DO
    BEGIN
      Fetch;
      Execute;
    END;
END.
```

specifies the operation to be performed. Additional bytes specify an *operand*, giving an 8-bit data value or a 16-bit memory address to use when executing the instruction.

Table 5–2 lists the 29 machine instructions of the H6809. With an 8-bit opcode, we could have defined up to 256 instructions, and in fact the real 6809

**TABLE 5–2** Machine instructions of the H6809.

| Mnem. | Operand | Length (bytes) | Opcode (hex.) | Description |
|---|---|---|---|---|
| NOP | | 1 | 12 | No operation |
| CLRA | | 1 | 4F | Clear A |
| COMA | | 1 | 43 | Ones' complement bits of A |
| NEGA | | 1 | 40 | Negate A (two's complement) |
| LDA | #data | 2 | 86 | Load A with data |
| LDA | @X | 1 | A6 | Load A with MEM[X] |
| LDA | addr | 3 | B6 | Load A with MEM[addr] |
| STA | @X | 1 | A7 | Store A into MEM[X] |
| STA | addr | 3 | B7 | Store A into MEM[addr] |
| ADDA | #data | 2 | 8B | Add data to A |
| ADDA | addr | 3 | BB | Add MEM[addr] to A |
| ANDA | #data | 2 | 84 | Logical AND data to A |
| ANDA | addr | 3 | B4 | Logical AND MEM[addr] to A |
| CMPA | #data | 2 | 81 | Set Z according to A−data |
| CMPA | addr | 3 | B1 | Set Z according to A−MEM[addr] |
| LDX | #addr | 3 | 8E | Load X with addr |
| LDX | addr | 3 | BE | Load X with MEMW[addr] |
| STX | addr | 3 | BF | Store X into MEMW[addr] |
| CMPX | #addr | 3 | 8C | Set Z according to X−addr |
| CMPX | addr | 3 | BC | Set Z according to X−MEMW[addr] |
| ADDX | #addr | 3 | 30 | Add addr to X |
| ADDX | addr | 3 | 31 | Add MEMW[addr] to X |
| LDS | #addr | 3 | 8F | Load SP with addr |
| BNE | offset | 2 | 26 | Branch if result is nonzero (Z=0) |
| BEQ | offset | 2 | 27 | Branch if result is zero (Z=1) |
| BRA | offset | 2 | 20 | Branch unconditionally |
| JMP | addr | 3 | 7E | Jump to addr |
| JSR | addr | 3 | BD | Jump to subroutine at addr |
| RTS | | 1 | 39 | Return from subroutine |

Notes: Mnem. = mnemonic; hex. = hexadecimal; data = 8-bit data value; addr = 16-bit memory address (two bytes); offset = 8-bit signed integer added to PC if branch is taken.

MEM[i] denotes the memory byte stored at address i.

MEMW[i] denotes the memory word beginning at address i, that is, the concatenation of MEM[i] and MEM[i+1].

Opcodes are the same as in the Motorola 6809, except for ADDX, LDS, LDA @X, and STA @X, which are two bytes long in the real 6809.

The ADDX mnemonic and some assembler notations differ from those used in Motorola's 6809 assembly language. See Chapter 17 for details.

defines instructions for 223 8-bit opcodes. Associated with each 8-bit opcode is an alphabetic *mnemonic* that we can use to conveniently name and recognize the instruction. In some cases two or three opcodes have the same mnemonic (e.g., three opcodes have "LDA"); we symbolically distinguish them by placing a special symbol (# or @) before the operand.

It is impossible to distinguish between instructions and data just by looking at the contents of memory. For example, the byte $4F_{16}$ may represent either the opcode CLRA or the number $79_{10}$. Only the processor distinguishes between the two. During the fetch cycle, the processor interprets memory bytes as instructions. During the execution cycle, it interprets them as data. There are no other checks. If an error causes PC to point into a data area, the processor will blindly forge ahead, trying to interpret the data as a sequence of instructions. Likewise, if a program stores data bytes into the memory locations occupied by its own instructions, it will destroy itself.

### 5.2.4 Instruction Groups

Instructions in the first group in Table 5–2 involve the accumulator A. The load and store instructions (LDA, STA) are by far the most commonly used, since data must be placed in the accumulator in order to be manipulated. ADDA adds an 8-bit quantity to A, while ANDA performs the bit-by-bit logical AND of A with another 8-bit value. CMPA compares A with another value without modifying either one. CLRA, COMA, and NEGA clear, complement, and negate A. NOP does nothing; it is used in program debugging to fill holes left by deleted instructions.

The second group contains instructions that manipulate X and SP. In most programs, X is used to hold a 16-bit address or other 16-bit data. The value of X can be loaded, stored, incremented, decremented, or compared with another 16-bit value. SP always points to the top of a push-down stack of return addresses, used by subroutine call and return instructions as explained later. The LDS instruction may be used to initialize SP at the beginning of a program.

Instructions in the last group can conditionally or unconditionally alter the program flow by forcing a new value into PC. The 3-byte instructions JMP and JSR can jump to any address in memory, while the shorter BNE, BEQ, and BRA branch to addresses nearby the current instruction.[2] A branch instruction interprets its `offset` byte as a signed, two's-complement number in the range $-128$ through $+127$. If the branch is taken, this number is added to PC, otherwise control passes to the instruction at the next address. Thus, the branch target address is within $-128$ to $+127$ bytes of the next address.

[2]"Branch" and "Jump" mean the same thing. However, the words are sometimes used to distinguish between absolute jumps and relative branches as explained in Section 8.6.

The instructions may also be grouped according to how they "address" their operands:

- *Inherent addressing.* The identity of the operand is inherent in the opcode itself (`NOP`, `CLRA`, `COMA`, `NEGA`, `RTS`).
- *Immediate addressing.* An 8-bit operand is contained in the byte following the opcode (`LDA #data`, `ADDA #data`, `ANDA #data`, `CMPA #data`), or a 16-bit operand is contained in the two bytes following the opcode (`LDX #addr`, `CMPX #addr`, `ADDX #addr`, `LDS #addr`).
- *Absolute addressing.* The address of the operand is given in the two bytes following the opcode, high-order byte first (`LDA addr`, `STA addr`, `ADDA addr`, `ANDA addr`, `CMPA addr`, `LDX addr`, `STX addr`, `CMPX addr`, `ADDX addr`, `JMP addr`, `JSR addr`).
- *Register indirect addressing.* The address of the operand is contained in the index register `X` (`LDA @X`, `STA @X`).
- *Relative addressing.* The address of the operand is computed as the sum of the `PC` and an 8-bit two's-complement number in the byte following the opcode (`BNE offset`, `BEQ offset`, `BRA offset`).

A precise description of each instruction will be given later by means of a Pascal simulation.

## 5.2.5 A Machine Language Program

Table 5–3 shows the values stored in memory for a sequence of instructions and data that forms a program for multiplying 23 by 5. The sequence of addresses, opcodes, and data in the two left-hand columns of the table is called a *machine language program*. This sequence completely specifies the operations to be performed by the computer. We shall explain the operation of the program shortly.

## 5.2.6 Assembly Language

Obviously, the two left-hand columns of Table 5–3 don't mean much to a human reader. Fortunately, the Label, Opcode, and Operand columns specify the machine language program in symbolic form, using mnemonics for opcodes and alphanumeric labels for addresses and data values. The Comments column gives an English explanation of what the program does. These four columns form an *assembly language program* that can be translated into machine language by a program called an *assembler*. The assembler produces, among other things, a listing of the equivalent machine language program as in the two left-hand columns of the table. In addition to machine instructions,

**TABLE 5–3** Memory contents for a sequence of instructions and data.

| Machine Language | | Assembly Language | | | |
|---|---|---|---|---|---|
| Addr (hex) | Contents (hex) | Label (sym) | Opcode (mnem) | Operand (sym) | Comments |
| ... | | | ORG | 2A40H | Multiply MCND by MPY. |
| 2A40 | 4F | START | CLRA | | Set PROD to 0. |
| 2A41 | B7 | | STA | PROD | |
| 2A42 | 2C | | | | |
| 2A43 | 00 | | | | |
| 2A44 | B6 | | LDA | MPY | Set CNT equal to MPY |
| 2A45 | 2C | | | | |
| 2A46 | 02 | | | | |
| 2A47 | B7 | | STA | CNT | and do loop MPY times. |
| 2A48 | 2C | | | | |
| 2A49 | 01 | | | | |
| 2A4A | B6 | LOOP | LDA | CNT | Done if CNT = 0. |
| 2A4B | 2C | | | | |
| 2A4C | 01 | | | | |
| 2A4D | 27 | | BEQ | OUT | |
| 2A4E | 10 | | | | |
| 2A4F | 8B | | ADDA | #-1 | Else decrement CNT. |
| 2A50 | FF | | | | |
| 2A51 | B7 | | STA | CNT | |
| 2A52 | 2C | | | | |
| 2A53 | 01 | | | | |
| 2A54 | B6 | | LDA | PROD | Add MCND to PROD. |
| 2A55 | 2C | | | | |
| 2A56 | 00 | | | | |
| 2A57 | BB | | ADDA | MCND | |
| 2A58 | 2C | | | | |
| 2A59 | 03 | | | | |
| 2A5A | B7 | | STA | PROD | |
| 2A5B | 2C | | | | |
| 2A5C | 00 | | | | |
| 2A5D | 20 | | BRA | LOOP | Repeat the loop again. |
| 2A5E | EB | | | | |
| 2A5F | B6 | OUT | LDA | PROD | Put PROD in A when done. |
| 2A60 | 2C | | | | |
| 2A61 | 00 | | | | |
| 2A62 | 7E | | JMP | 1000H | Return to operating system. |
| 2A63 | 10 | | | | |
| 2A64 | 00 | | | | |
| ... | | | ORG | 2C00H | |
| 2C00 | ?? | PROD | RMB | 1 | Storage for PROD. |
| 2C01 | ?? | CNT | RMB | 1 | Storage for CNT. |
| 2C02 | 05 | MPY | FCB | 5 | Multiplier value. |
| 2C03 | 17 | MCND | FCB | 23 | Mutliplicand value. |
| ... | | | END | START | |

Notes: Addr=Address; hex=hexadecimal; sym=symbolic; mnem=mnemonic.

the assembly language program contains *pseudo-operations* that tell the assembler how to store the machine language program. The four pseudo-operations used in Table 5–3 are described below:

- ORG (Origin). The operand specifies the address at which the next instruction is to be deposited when the program is loaded into memory. Subsequent instructions are deposited in successive memory addresses.
- RMB (Reserve Memory Bytes). The operand specifies a number of memory bytes to be skipped without storing any instructions or data, thereby reserving space to be used by variables in a program.
- FCB (Form Constant Byte). The specified byte value is stored into memory when the program is first loaded into memory, thereby establishing a constant value that may be accessed when the program is run.
- END (End Assembly). This instruction denotes the end of the text to be assembled and gives the address of the first executable instruction of the program.

In all of the assembly language statements, numeric arguments are assumed to be given in decimal notation unless they are followed by an H for hexadecimal. When an identifier appears in the Label field, it is assigned the value of the Address field. For example, the values of START and CNT are 2A40H and 2C01H, respectively. When an identifier appears in the Operand field, the assembler substitutes the value that has been assigned to it. Therefore, the instruction "STA CNT" is equivalent to "STA 2C01H" .

All of the above operations occur at *assembly time*; an identifier such as CNT refers to the memory address of a variable. At *run time*, when the program is executed, values stored in memory will be manipulated. Strictly speaking, we should refer to such a value as MEM[CNT]. However, when we discuss run-time operations, it is customary to use the identifier to refer to the value in memory itself. Thus, the comment "Set CNT equal to MPY" means "MEM[2C01H] := MEM[2C02H]."

The difference between assembly-time and run-time operations is probably the greatest single source of confusion to novice assembly language programmers, and so we'll explain it again. The assembler is a system program that translates lines of text into a sequence of instruction and data bytes that can be stored in the computer's memory; the assembler "goes away" before the machine language program that it produced is executed. As far as the assembler is concerned, an identifier such as PROD stands for the address that was assigned to it (2C00H); the assembler is unconcerned with what may happen to the contents of the memory address 2C00H when the program is run. Using an identifier frees the programmer from keeping track of the exact address at which an instruction or datum is located. Even though the identifier refers to an address, the programmer is usually more interested in the *contents* of the memory address at run time. Therefore the programmer informally uses the *name* of the address (PROD) to refer to its contents (MEM[2C00H]), just to save typing.

Most instructions in the H6809 are more than one byte long. It is therefore convenient to compress the program listing, showing all bytes associated with the same instruction on one line as in Table 5–4.

### 5.2.7 Operation of a Simple Program

Now we can explain how the program in Table 5–4 works. It multiplies MCND by MPY by intializing PROD to 0 and then adding MCND to it MPY times. The program's execution is traced by Table 5–5, which shows the contents of registers and memory *after* each instruction is executed, and by the steps below.

(1) CLRA sets A to zero.

(2) STA PROD is a "memory reference" instruction. The two bytes following the opcode B7H refer to memory location 2C00H, the address where the contents of A (00H) are to be stored.

**TABLE 5–4** Compressed program listing.

| Addr | Contents | Label | Opcode | Operand | Comments |
|---|---|---|---|---|---|
| ... | | | ORG | 2A40H | Multiply MCND by MPY. |
| 2A40 | 4F | START | CLRA | | Set PROD to 0. |
| 2A41 | B7 2C00 | | STA | PROD | |
| 2A44 | B6 2C02 | | LDA | MPY | Set CNT equal to MPY |
| 2A47 | B7 2C01 | | STA | CNT | and do loop MPY times. |
| 2A4A | B6 2C01 | LOOP | LDA | CNT | Done if CNT = 0. |
| 2A4D | 27 10 | | BEQ | OUT | |
| 2A4F | 8B FF | | ADDA | #-1 | Else decrement CNT. |
| 2A51 | B7 2C01 | | STA | CNT | |
| 2A54 | B6 2C00 | | LDA | PROD | Add MCND to PROD. |
| 2A57 | BB 2C03 | | ADDA | MCND | |
| 2A5A | B7 2C00 | | STA | PROD | |
| 2A5D | 20 EB | | BRA | LOOP | Repeat the loop again. |
| 2A5F | B6 2C00 | OUT | LDA | PROD | Put PROD in A when done. |
| 2A62 | 7E 1000 | | JMP | 1000H | Return to operating system. |
| ... | | | ORG | 2C00H | |
| 2C00 | ?? | PROD | RMB | 1 | Storage for PROD. |
| 2C01 | ?? | CNT | RMB | 1 | Storage for CNT. |
| 2C02 | 05 | MPY | FCB | 5 | Multiplier value. |
| 2C03 | 17 | MCND | FCB | 23 | Multiplicand value. |
| ... | | | END | START | |

**TABLE 5–5** Register and memory contents after executing instructions in multiplication program.

| Step | Instruction | PC | A | Z | MEM[2C00] (PROD) | MEM[2C01] (CNT) |
|---|---|---|---|---|---|---|
| 0 | ... | 2A40 | ?? | ? | ?? | ?? |
| 1 | CLRA | 2A41 | 00 | 1 | ?? | ?? |
| 2 | STA PROD | 2A44 | 00 | 1 | 00 | ?? |
| 3 | LDA MPY | 2A47 | 05 | 0 | 00 | ?? |
| 4 | STA CNT | 2A4A | 05 | 0 | 00 | 05 |
| 5 | LDA CNT | 2A4D | 05 | 0 | 00 | 05 |
| 6 | BEQ OUT | 2A4F | 05 | 0 | 00 | 05 |
| 7 | ADDA #-1 | 2A51 | 04 | 0 | 00 | 05 |
| 8 | STA CNT | 2A54 | 04 | 0 | 00 | 04 |
| 9 | LDA PROD | 2A57 | 00 | 1 | 00 | 04 |
| 10 | ADDA MCND | 2A5A | 17 | 0 | 00 | 04 |
| 11 | STA PROD | 2A5D | 17 | 0 | 17 | 04 |
| 12 | BRA LOOP | 2A4A | 17 | 0 | 17 | 04 |
| 5 | LDA CNT | 2A4D | 04 | 0 | 17 | 04 |
| 6 | BEQ OUT | 2A4F | 04 | 0 | 17 | 04 |
| | ... | | | | | |
| 5 | LDA CNT | 2A4D | 03 | 0 | 2E | 03 |
| 6 | BEQ OUT | 2A4F | 03 | 0 | 2E | 03 |
| | ... | | | | | |
| 5 | LDA CNT | 2A4D | 02 | 0 | 45 | 02 |
| 6 | BEQ OUT | 2A4F | 02 | 0 | 45 | 02 |
| | ... | | | | | |
| 5 | LDA CNT | 2A4D | 01 | 0 | 5C | 01 |
| 6 | BEQ OUT | 2A4F | 01 | 0 | 5C | 01 |
| | ... | | | | | |
| 5 | LDA CNT | 2A4D | 00 | 1 | 73 | 00 |
| 6 | BEQ OUT | 2A5F | 00 | 1 | 73 | 00 |
| 13 | LDA PROD | 2A62 | 73 | 0 | 73 | 00 |
| 14 | JMP 1000H | 1000 | 73 | 0 | 73 | 00 |

(3) LDA MPY is another memory reference instruction. It specifies that A is to be loaded with the contents of memory location 2C02H.

(4) STA CNT stores the multiplier into memory location 2C01H to keep track of how many more times the loop must be executed.

(5) LDA CNT puts the value of location 2C01H into A again. Since data manipulation instructions set Z according to their results, this instruction sets Z to 1 if CNT=0, and to 0 otherwise.

(6) BEQ OUT causes PC to be set to 2A5FH if Z=1, that is, it branches to step 13 if CNT=0. Instead of giving the actual target address (2A5FH), the second byte of the machine instruction contains an offset: if the condi-

tion is true then the next instruction to be executed is $10_{16}$ bytes past the instruction that would normally be next. In other words, the offset is added to the updated PC if the branch is taken. However, notice that the assembly language statement specifies the actual target address; the assembler figures out the proper offset value for the machine instruction and gives an error message if an out-of-range target is specified.

(7) ADDA #-1 subtracts 1 from A (if the program hasn't just branched to OUT). This instruction uses an "immediate" operand. When the control unit decodes opcode 8BH (ADDA#), it reads the next byte from the instruction stream (FFH) and adds it to A.

(8) STA CNT stores A as before.

(9) LDA PROD puts the value of location 2C00H into A.

(10) ADDA MCND adds the value of location 2C03H to A.

(11) STA PROD stores the new value of A back into location 2C00H.

(12) BRA LOOP unconditionally loads the value 2A4AH into PC, returning control to step 5. The target address 2A4AH is obtained by extending the sign of the offset EBH to make a 16-bit two's-complement integer, FFEBH, and adding to the updated PC (2A5FH).

(13) LDA PROD puts the value of location 2C00H into A again.

(14) JMP 1000H unconditionally loads the value 1000H into PC when the program is done. We've assumed that this is the starting address of an operating system program that controls the machine when the user's program has finished executing.

### 5.2.8 Pascal Simulation of Instruction Execution

Before discussing the rest of the H6809 instruction set, we present an extended-Pascal simulation that precisely defines the operation of each instruction. Table 5–6 is the Execute procedure that goes with the simulation given in Table 5–1. We've taken some liberty with Pascal by allowing "#" and "@" to appear in identifiers. The body of the procedure is a CASE statement, one case for each valid opcode. You should study Table 5–6 to verify that each of the instructions that we've introduced so far does what you think it does, and also to get a preview of the instructions yet to come.

Notice that the standard fetch cycle in Table 5–1 leaves the PC pointing to the byte after the opcode, regardless of whether this byte is the next opcode or part of the current instruction. The execution cycle reads one or two additional bytes and bumps PC past them only for opcodes that require them, so that at the end of the execution cycle PC always points to the first byte of the next instruction.

### 5.2.9 Indirect Addressing

The program in Table 5–4 manipulated only simple variables and constants. More complicated data structures such as arrays, stacks, and queues are used in almost all programs. Consider the problem of initializing the five components of an array Q[0..4] to zero, using only the instructions that we've introduced so far. An assembly language solution is shown in Table 5–7. Note that the operand expressions "Q+1," "Q+2," and so on are evaluated at assembly time. Also, the second ORG statement affects the next line to be assembled, but does not affect the Address column on its own line.

The choice of a 5-component array above was very judicious — the corresponding program for a 100-component array would not fit on one page. A problem with the direct addressing mode used by most H6809 instructions is that addresses must be known at assembly time, resulting in programs like Table 5–7. *Indirect addressing* avoids this problem by computing addresses at run time. The H6809 has one indirect addressing mode, in which the address of the operand is taken from the X register when the instruction is executed. The accumulator may be loaded from or stored into memory using this mode. Thus we can write a loop to initialize an array, in which the X register contains a new address in the array on each iteration of the loop. Before showing such a loop, we must introduce some instructions for manipulating the X register itself.

**TABLE 5–6** Instruction execution procedure for the H6809.

```
PROCEDURE Execute;
  CONST {Mnemonic-opcode correspondences}
    NOP=12H; CLRA=4FH; COMA=43H; NEGA=40H; LDA#=86H; LDA@X=0A6H;
    LDA=0B6H; STA@X=0A7H; STA=0B7H; ADDA#=8BH; ADDA=0BBH;
    ANDA#=84H; ANDA=0B4H; CMPA#=81H; CMPA=0B1H; LDX#=8EH;
    LDX=0BEH; STX=0BFH; CMPX#=8CH; CMPX=0BCH; ADDX#=30H; ADDX=31H;
    LDS#=8FH; BNE=26H; BEQ=27H; BRA=20H; JMP=7EH; JSR=0BDH; RTS=39H;
  VAR opnum : integer;

  FUNCTION NextByte : byte;  {Get next byte in instruction stream.}
    BEGIN NextByte := MEM[PC]; PC := PC + 1 END;

  PROCEDURE FetchAddr;  {Get 2-byte address from instruction stream.}
    BEGIN EAR[15::8] := NextByte; EAR[7::0] := NextByte END;

  PROCEDURE TestByte (b : byte);  {Set Z according to byte value.}
    BEGIN IF b=0 THEN Z:=1 ELSE Z:=0 END;

  PROCEDURE TestWord (w : word);  {Set Z according to word value.}
    BEGIN IF w=0 THEN Z:=1 ELSE Z:=0 END;

  PROCEDURE Branch;
    BEGIN  {Sign-extend offset in EAR[7::0] and add to PC.}
      IF EAR[7]=0 THEN EAR[15::8] := 0 ELSE EAR[15::8] := 0FFH;
      PC := PC + EAR;
    END;
```

**TABLE 5–6** (continued)

```
BEGIN {Statement part of Execute}
  opnum := IR;
  CASE opnum OF
    NOP:    ;
    CLRA:   BEGIN A := 0; TestByte(A) END;
    COMA:   BEGIN A := Bcom(A); TestByte(A) END;
    NEGA:   BEGIN A := -A; TestByte(A) END;
    LDA#:   BEGIN A := NextByte; TestByte(A) END;
    LDA@X:  BEGIN A := MEM[X]; TestByte(A) END;
    LDA:    BEGIN FetchAddr; A := MEM[EAR]; TestByte(A) END;
    STA:    BEGIN FetchAddr; MEM[EAR] := A; TestByte(A) END;
    STA@X:  BEGIN MEM[X] := A; TestByte(A) END;
    ADDA#:  BEGIN A := A + Nextbyte; TestByte(A) END;
    ADDA:   BEGIN FetchAddr; A := A + MEM[EAR]; TestByte(A) END;
    ANDA#:  BEGIN A := Band(A,Nextbyte); TestByte(A) END;
    ANDA:   BEGIN FetchAddr; A := Band(A,MEM[EAR]); TestByte(A) END;
    CMPA#:  BEGIN TestByte(A-Nextbyte) END;
    CMPA:   BEGIN FetchAddr; TestByte(A-MEM[EAR]) END;
    LDX#:   BEGIN FetchAddr; X := EAR; TestWord(X) END;
    LDX:    BEGIN FetchAddr; X[15::8] := MEM[EAR];
              X[7::0] := MEM[EAR+1]; TestWord(X) END;
    STX:    BEGIN FetchAddr; MEM[EAR] := X[15::8];
              MEM[EAR+1] := X[7::0]; TestWord(X) END;
    CMPX#:  BEGIN FetchAddr; TestWord(X-EAR) END;
    CMPX:   BEGIN FetchAddr; TestWord(X-(MEM[EAR]|MEM[EAR+1])) END;
    ADDX#:  BEGIN FetchAddr; X := X + EAR; TestWord(X) END;
    ADDX:   BEGIN FetchAddr; X := X + (MEM[EAR]|MEM[EAR+1]);
              TestWord(X) END; {"|" denotes concatenation of bytes}
    LDS#:   BEGIN FetchAddr; SP := EAR; TestWord(SP) END;
    BNE:    BEGIN EAR[7::0] := NextByte; IF Z=0 THEN Branch END;
    BEQ:    BEGIN EAR[7::0] := NextByte; IF Z=1 THEN Branch END;
    BRA:    BEGIN EAR[7::0] := NextByte; Branch END;
    JMP:    BEGIN FetchAddr; PC := EAR END;
    JSR:    BEGIN
              FetchAddr; SP := SP - 2;  {Reserve 2 bytes on stack.}
              MEM[SP] := PC[15::8]; MEM[SP+1] := PC[7::0]; {Save PC}
              PC := EAR;  {...and jump to subroutine.}         END;
    RTS:    BEGIN  {Restore PC from top of stack.}
              PC[15::8] := MEM[SP]; PC[7::0] := MEM[SP+1];
              SP := SP + 2;  {Pop stack.}              END;
  END; {End CASE}
END; {End Execute}
```

As shown in the second part of Table 5–2, X may be loaded with an immediate value (LDX #addr), in which case the immediate value is two bytes long and is contained in the second and third bytes of the instruction. If X is loaded from a memory address (LDX addr), the instruction specifies the address of the first byte of a word. The high-order byte of X is loaded with

**TABLE 5–7** Initializing an array the hard way.

| Addr | Contents | Label | Opcode | Operand | Comments |
|---|---|---|---|---|---|
| | | | ORG | 3000H | |
| 3000 | 4F | INIT | CLRA | | Set components of Q to zero. |
| 3001 | B7 3100 | | STA | Q | First component. |
| 3004 | B7 3101 | | STA | Q+1 | Second component. |
| 3007 | B7 3102 | | STA | Q+2 | Third component. |
| 300A | B7 3103 | | STA | Q+3 | Fourth component. |
| 300D | B7 3104 | | STA | Q+4 | Fifth component. |
| 3010 | 7E 1000 | | JMP | 1000H | Return to operating system. |
| 3013 | | | ORG | 3100H | |
| 3100 | ?? | Q | RMB | 5 | Reserve 5 bytes for array. |
| 3105 | | | END | INIT | |

MEM[addr], and the low-order byte of X is loaded with MEM[addr+1]. The X register may also be stored into memory, compared with an immediate or memory operand, or have an immediate or memory operand added to it.

The program in Table 5–8 solves the array initialization problem using indirect addressing. As shown in Figure 5–4, it initializes X to point to the first component of Q, and then executes a loop that clears successive components of Q, incrementing X once per iteration of the loop. The CMPX instruction compares the contents of X with an immediate value, the address just past the last array component; it sets Z to 1 if they're equal. Because X is used to access array components with different indices, it is called an *index register*.

Not only does the program in Table 5–8 occupy fewer bytes than the one in Table 5–7, but it also stays the same length for an array of any size. The program is easily modified to work on a different length array by changing the occurrences of the length "5" to the desired length.

**TABLE 5–8** Initializing an array using indirect addressing.

| Addr | Contents | Label | Opcode | Operand | Comments |
|---|---|---|---|---|---|
| | | | ORG | 3000H | |
| 3000 | 4F | INIT | CLRA | | Set components of Q to zero. |
| 3001 | 8E 3100 | | LDX | #Q | Address of first component. |
| 3004 | A7 | ILOOP | STA | @X | Set MEM[X] to zero. |
| 3005 | 30 0001 | | ADDX | #1 | Point to next component. |
| 3008 | 8C 3105 | | CMPX | #Q+5 | Past last component? |
| 300B | 26 F7 | | BNE | ILOOP | If not, go do some more. |
| 300D | 7E 1000 | | JMP | 1000H | Return to operating system. |
| 3010 | | | ORG | 3100H | |
| 3100 | ?? | Q | RMB | 5 | Reserve 5 bytes for array. |
| 3105 | | | END | INIT | |

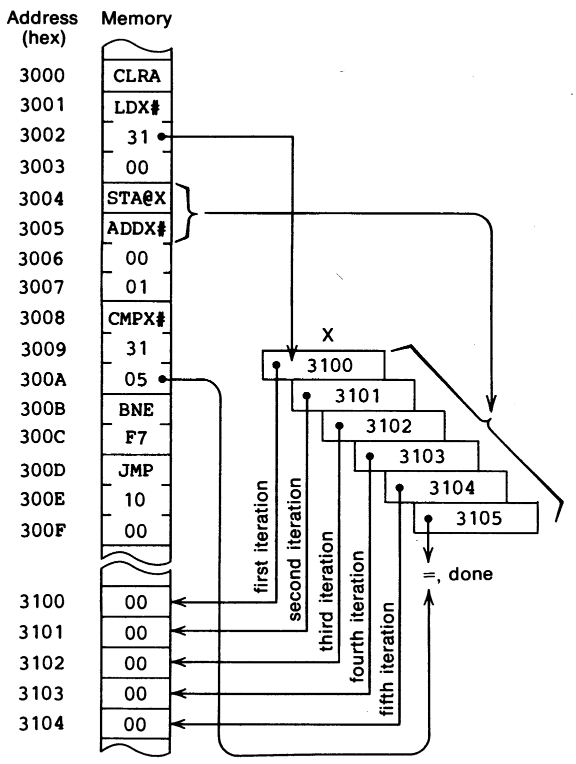

**FIGURE 5–4** Effects of indirect addressing.

Many variations on indirect addressing are found in the real 6809 and most other computers. These addressing modes are explored in Chapter 7.

### 5.2.10 Subroutines

A *subroutine* is the machine language equivalent of a procedure or function in Pascal: a sequence of instructions, defined and stored only once, that may be invoked (or *called*) from many places. In order to write subroutines in machine language, we need instructions to save the current value of the PC each time the subroutine is called, and restore it when the subroutine is finished.

In the H6809, the JSR and RTS instructions provide for subroutine calls and returns in conjunction with the stack pointer register SP. Any program that uses subroutines is required to reserve a small area of memory for a push-down stack for return addresses. At the beginning of such a program SP must be initialized to point at this area using the LDS #addr instruction. As shown in Figure 5–5, SP points to the top item in the stack, or just past the stack area if the stack is empty. SP is decremented once before storing each

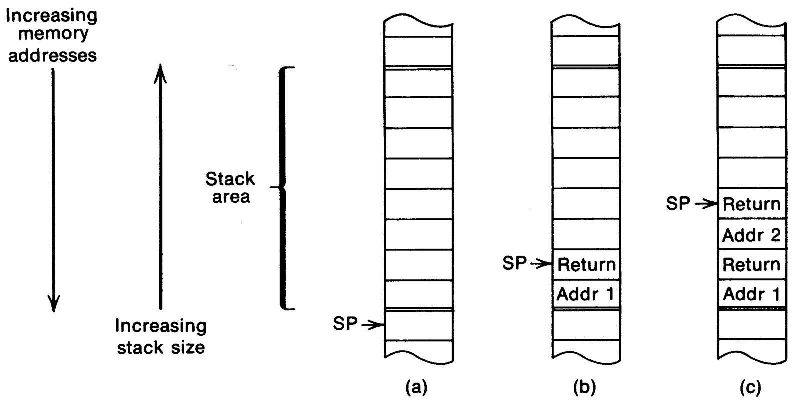

**FIGURE 5–5** H6809 return-address stack: (a) empty; (b) after one subroutine call; (c) after second (nested) subroutine call.

byte on the stack, and incremented once after popping each byte. A return address occupies two bytes.

Table 5–9 outlines a Pascal program with two nested subroutines and the corresponding assembly language statements. The `JSR addr` instruction saves

**TABLE 5–9** Program with two nested subroutines.

| Addr | Contents | Label | Opcode | Operand | Comments |
|---|---|---|---|---|---|
| | | | ORG | 3000H | PROGRAM Subrs (input,output); |
| 3000 | ... | SUBR2 | ... | | PROCEDURE P2; |
| ... | ... | | ... | | BEGIN |
| ... | ... | | ... | | ... |
| 3055 | 39 | | RTS | | END; |
| 3056 | ... | SUBR1 | ... | | PROCEDURE P1; |
| ... | ... | | ... | | BEGIN |
| ... | ... | | ... | | ... |
| 30A2 | BD 3000 | | JSR | SUBR2 | P2; {Call P2} |
| 30A5 | ... | RET2 | ... | | ... |
| ... | ... | | ... | | ... |
| 30C2 | 39 | | RTS | | END; |
| 30C5 | 8F 3FF8 | MAIN | LDS | #STK+8 | BEGIN {Main program} |
| ... | ... | | ... | | ... |
| 312F | BD 3056 | | JSR | SUBR1 | P1; {Call P1} |
| 3132 | ... | RET1 | ... | | ... |
| ... | ... | | ... | | ... |
| 31A7 | 7E 1000 | | JMP | 1000H | END. |
| ... | | | ORG | 3FF0H | |
| 3FF0 | ... | STK | RMB | 8 | |
| ... | | | END | MAIN | |

the address of the next instruction by pushing it onto the stack and then jumps to the instruction at location addr, the first instruction of the subroutine. At the end of the subroutine, RTS pops an address from the stack into PC, effecting a return to the original program sequence.

A stack is the most appropriate data structure for saving return addresses, because it can store more than one return address when subroutines are nested. The number of levels of nesting is limited only by the size of the memory area reserved by the programmer for the stack.

A detailed example program using subroutines is given in Table 5–10. Before describing it, we should point out a few additional assembly language pseudo-operations and features that it uses:

- FCW (Form Constant Word). The specified 16-bit value is stored into two successive memory bytes, high-order byte first. Both FCW and FCB can take multiple operands, separated by commas.
- RMW (Reserve Memory Words). If the value of the operand is *n*, then 2*n* memory bytes are skipped without storing any instructions or data in them.
- EQU (Equate). The identifier in the Label field is assigned the value in the Operand field, instead of the value in the Address field. This makes the identifier a synonym for a constant value for the duration of the assembly process.
- * (Comment Lines). Any line beginning with an asterisk is completely ignored by the assembler.
- * (Program Location Counter). When used in an expression, the symbol "*" denotes the current address at which assembly is taking place.

The program in Table 5–10 contains a main program and two subroutines. The main program initializes SP to point to a 7-word stack. A stack of 2 words would have been sufficient for this program, but it is a good practice to provide "headroom" in case programming errors, modifications, or interrupts (Chapter 11) increase the space required.

The main program loads a 16-bit word into X and calls a subroutine WCNT1S that counts the number of "1" bits in X. WCNT1S splits X into two bytes and calls a subroutine BCNT1S to count 1s in each byte. The power of subroutines is evidenced by the fact that BCNT1S can be called more than once and with a different byte to be converted each time.

Figure 5–6 shows the state of the stack after each instruction that affects it. When WCNT1S returns to the main program, the stack is again empty and A contains the 1s count. The main program terminates by jumping to the operating system.

The individual subroutines in Table 5–10 are worth discussing. On entry, WCNT1S expects the input word to be in X. This is the first case we've seen

**TABLE 5–10** Program that uses subroutines to count the number of "1" bits in a word.

```
Addr Contents Label  Opcode Operand  Comments

                    ORG    2000H
2000 8F 201A MAIN   LDS    #STKE    Initialize SP.
2003 BE 201A        LDX    TWORD    Get test word.
2006 BD 201C        JSR    WCNT1S   Count number of 1s in it.
2009 7E 1000        JMP    SYSRET   Return to operating system.
200C         SYSRET EQU    1000H    Operating system address.
200C ??      STK    RMW    7        Space for 7 return addresses.
201A         STKE   EQU    *        Initialization address for SP.
201A 5B29    TWORD  FCW    5B29H    Test-word to count 1s.
201C         *
201C         *      Count the number of '1' bits in a word.
201C         *      Enter with word in X, exit with count in A.
201C BF 2032 WCNT1S STX    CWORD    Save input word.
201F B6 2032        LDA    CWORD    Get high-order byte.
2022 BD 2035        JSR    BCNT1S   Count 1s.
2025 B7 2034        STA    W1CNT    Save '1' count.
2028 B6 2033        LDA    CWORD+1  Get low-order byte.
202B BD 2035        JSR    BCNT1S   Count 1s.
202E BB 2034        ADDA   W1CNT    Add high-order count.
2031 39             RTS             Done, return.
2032 ??      CWORD  RMW    1        Save word being counted.
2034 ??      W1CNT  RMB    1        Save number of 1s.
2035         *
2035         *      Count number of '1' bits in a byte.
2035         *      Enter with byte in A, exit with count in A.
2035 B7 2061 BCNT1S STA    CBYTE    Save input byte.
2038 4F             CLRA            Initialize '1' count.
2039 B7 2062        STA    B1CNT
203C 8E 2059        LDX    #MASKS   Point to 1-bit masks.
203F A6      BLOOP  LDA    @X       Get next bit mask.
2040 B4 2061        ANDA   CBYTE    Is there a '1' there?
2043 27 08          BEQ    BNO1     Skip if not.
2045 B6 2062        LDA    B1CNT    Otherwise increment
2048 8B 01          ADDA   #1         '1' count.
204A B7 2062        STA    B1CNT
204D 30 0001 BNO1   ADDX   #1       Point to next mask.
2050 8C 2061        CMPX   #MASKE   Past last mask?
2053 26 EA          BNE    BLOOP    Continue if not.
2055 B6 2062        LDA    B1CNT    Put total count in A.
2058 39             RTS             Return.
2059         *      Define 1-bit masks to test bits of byte.
2059 80402010MASKS  FCB    80H,40H,20H,10H,8H,4H,2H,1H
205D 08040201
2061         MASKE  EQU    *        Address just after table.
2061 ??      CBYTE  RMB    1        Save byte being counted.
2062 ??      B1CNT  RMB    1        Save '1' count.
2063                END    MAIN
```

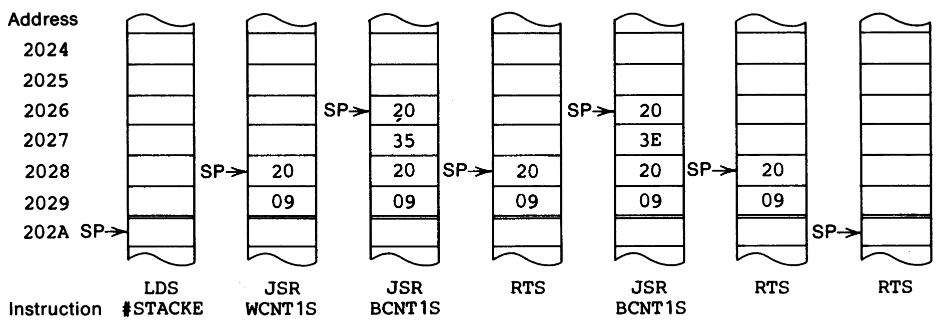

**FIGURE 5–6** Stack contents after the execution of instructions in Table 5–10.

where X is used as a convenient place to hold arbitrary 16-bit data, not an address. Using STX CWORD, the subroutine saves X in two memory bytes that are individually read by LDA CWORD and LDA CWORD+1 and passed to BCNT1S.

When BCNT1S is entered, it expects the input byte to be in A and saves it in CBYTE. The subroutine then checks each bit position for a "1" and maintains a count in B1CNT accordingly. It uses a table of eight "mask bytes," each having a "1" in a different bit position. X contains the address of a mask byte, and CBYTE is combined with each mask byte using the ANDA @X instruction. At each iteration of the loop, ANDA produces a nonzero result if and only if the tested bit of CBYTE is 1.

Like procedures and functions in Pascal, subroutines are the key to the structure of assembly language programs. A typical program is divided into many "modules," each of which is a subroutine with inputs, outputs, and local data. Subroutines will be discussed in detail in Chapter 9.

## 5.3 GENERAL-REGISTER PROCESSORS

A general-register processor (or machine) has a set of *general registers* in which arithmetic and logical operations and data transfers take place. The registers are "general" because they are all treated identically (or almost identically). Any register may contain data to be used by an instruction, or may be used to specify an address. Typical general-register processors have 8 or 16 registers of 16 to 36 bits each. The PDP-11, Z8000, and 68000 are the best examples of general-register processors in this book.

### 5.3.1 Organization of a General-Register Processor

The internal organization of a hypothetical general-register processor is shown in Figure 5–7. This processor contains a subset of the registers and features of the Zilog Z8000, so we'll call it the H8000. Comparing the H8000

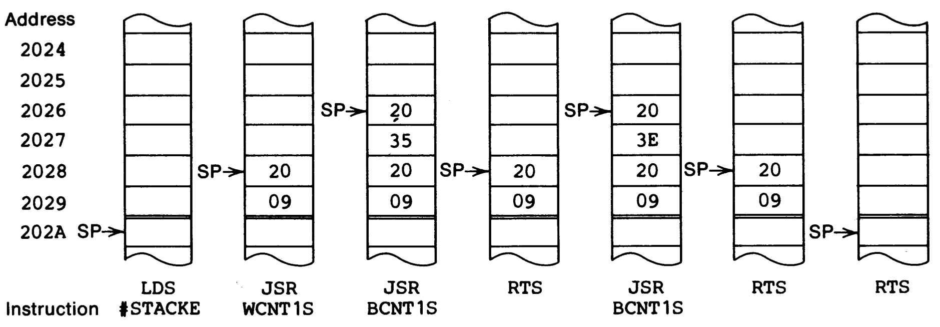

**FIGURE 5–6** Stack contents after the execution of instructions in Table 5–10.

where X is used as a convenient place to hold arbitrary 16-bit data, not an address. Using STX CWORD, the subroutine saves X in two memory bytes that are individually read by LDA CWORD and LDA CWORD+1 and passed to BCNT1S.

When BCNT1S is entered, it expects the input byte to be in A and saves it in CBYTE. The subroutine then checks each bit position for a "1" and maintains a count in B1CNT accordingly. It uses a table of eight "mask bytes," each having a "1" in a different bit position. X contains the address of a mask byte, and CBYTE is combined with each mask byte using the ANDA @X instruction. At each iteration of the loop, ANDA produces a nonzero result if and only if the tested bit of CBYTE is 1.

Like procedures and functions in Pascal, subroutines are the key to the structure of assembly language programs. A typical program is divided into many "modules," each of which is a subroutine with inputs, outputs, and local data. Subroutines will be discussed in detail in Chapter 9.

## 5.3 GENERAL-REGISTER PROCESSORS

A general-register processor (or machine) has a set of *general registers* in which arithmetic and logical operations and data transfers take place. The registers are "general" because they are all treated identically (or almost identically). Any register may contain data to be used by an instruction, or may be used to specify an address. Typical general-register processors have 8 or 16 registers of 16 to 36 bits each. The PDP-11, Z8000, and 68000 are the best examples of general-register processors in this book.

### 5.3.1 Organization of a General-Register Processor

The internal organization of a hypothetical general-register processor is shown in Figure 5–7. This processor contains a subset of the registers and features of the Zilog Z8000, so we'll call it the H8000. Comparing the H8000