of the same type or one of the other types. Each structure has only one entry point and one exit point. These programming structures may seem restrictive, but using them usually results in algorithms which are easy to follow. Also, as we will show you soon, if you write the algorithm for a program carefully with these standard structures, it is relatively easy to translate the algorithm to the equivalent assembly language instructions.

## Finding the Right Instruction

After you get the structure of a program worked out and written down, the next step is to determine the instruction statements required to do each part of the program. Since the examples in this book are based on the 8086 family of microprocessors, now is a good time to give you an overview of the instructions the 8086 has for you to use. First, however, is a hint about how to approach these instructions.

You do not usually learn a new language by memorizing an entire dictionary of the language. A better way is to learn a few useful words and practice putting these words together in simple sentences. You can then learn more words as you need them to express more complex thoughts. Likewise, you should not try to memorize all the instructions for a microprocessor at once.

For future reference, Chapter 6 contains a dictionary of all the 8086 instructions with detailed descriptions and examples of each. As an introduction, however, the few pages here contain a list of all the 8086 instructions with a short explanation of each. Skim through the list and pick out a dozen or so instructions that seem useful and understandable. As a start, look for move, input, output, logical, and arithmetic instructions. Then look through the list again to see if you can find the instructions that you might use to do the "read temperature sensor value from a port, add +7, and store result in memory" example program.

You can use Chapter 6 as a reference as you write programs. Here we simply list the 8086 instructions in *functional* groups with single-sentence descriptions so that you can see the types of instructions that are available to you. As you read through this section, do not expect to understand all the instructions. When you start writing programs, you will probably use this section to determine the type of instruction and Chapter 6 to get the instruction details as you need them. After you have written a few programs, you will remember most of the basic instruction types and will be able to simply look up an instruction in Chapter 6 to get any additional details you need. Chapter 4 shows you in detail how to use the move, arithmetic, logical, jump, and string instructions. Chapter 5 shows how to use the call instructions and the stack.

## DATA TRANSFER INSTRUCTIONS

*General-purpose byte or word transfer instructions:*

| MNEMONIC | DESCRIPTION |
|---|---|
| MOV | Copy byte or word from specified source to specified destination. |
| PUSH | Copy specified word to top of stack. |
| POP | Copy word from top of stack to specified location. |
| PUSHA | (80186/80188 only) Copy all registers to stack. |
| POPA | (80186/80188 only) Copy words from stack to all registers. |
| XCHG | Exchange bytes or exchange words. |
| XLAT | Translate a byte in AL using a table in memory. |

*Simple input and output port transfer instructions:*

| | |
|---|---|
| IN | Copy a byte or word from specified port to accumulator. |
| OUT | Copy a byte or word from accumulator to specified port. |

*Special address transfer instructions:*

| | |
|---|---|
| LEA | Load effective address of operand into specified register. |
| LDS | Load DS register and other specified register from memory. |
| LES | Load ES register and other specified register from memory. |

*Flag transfer instructions:*

| | |
|---|---|
| LAHF | Load (copy to) AH with the low byte of the flag register. |
| SAHF | Store (copy) AH register to low byte of flag register. |
| PUSHF | Copy flag register to top of stack. |
| POPF | Copy word at top of stack to flag register. |

## ARITHMETIC INSTRUCTIONS

*Addition instructions:*

| | |
|---|---|
| ADD | Add specified byte to byte or specified word to word. |
| ADC | Add byte + byte + carry flag or word + word + carry flag. |
| INC | Increment specified byte or specified word by 1. |
| AAA | ASCII adjust after addition. |
| DAA | Decimal (BCD) adjust after addition. |

*Subtraction instructions:*

| | |
|---|---|
| SUB | Subtract byte from byte or word from word. |
| SBB | Subtract byte and carry flag from byte or word and carry flag from word. |
| DEC | Decrement specified byte or specified word by 1. |

| | |
|---|---|
| NEG | Negate — invert each bit of a specified byte or word and add 1 (form 2's complement). |
| CMP | Compare two specified bytes or two specified words. |
| AAS | ASCII adjust after subtraction. |
| DAS | Decimal (BCD) adjust after subtraction. |

*Multiplication instructions:*

| | |
|---|---|
| MUL | Multiply unsigned byte by byte or unsigned word by word. |
| IMUL | Multiply signed byte by byte or signed word by word. |
| AAM | ASCII adjust after multiplication. |

*Division instructions:*

| | |
|---|---|
| DIV | Divide unsigned word by byte or unsigned double word by word. |
| IDIV | Divide signed word by byte or signed double word by word. |
| AAD | ASCII adjust before division. |
| CBW | Fill upper byte of word with copies of sign bit of lower byte. |
| CWD | Fill upper word of double word with sign bit of lower word. |

## BIT MANIPULATION INSTRUCTIONS

*Logical instructions:*

| | |
|---|---|
| NOT | Invert each bit of a byte or word. |
| AND | AND each bit in a byte or word with the corresponding bit in another byte or word. |
| OR | OR each bit in a byte or word with the corresponding bit in another byte or word. |
| XOR | Exclusive OR each bit in a byte or word with the corresponding bit in another byte or word. |
| TEST | AND operands to update flags, but don't change operands. |

*Shift instructions:*

| | |
|---|---|
| SHL/SAL | Shift bits of word or byte left, put zero(s) in LSB(s). |
| SHR | Shift bits of word or byte right, put zero(s) in MSB(s). |
| SAR | Shift bits of word or byte right, copy old MSB into new MSB. |

*Rotate instructions:*

| | |
|---|---|
| ROL | Rotate bits of byte or word left, MSB to LSB and to CF. |
| ROR | Rotate bits of byte or word right, LSB to MSB and to CF. |
| RCL | Rotate bits of byte or word left, MSB to CF and CF to LSB. |
| RCR | Rotate bits of byte or word right, LSB to CF and CF to MSB. |

## STRING INSTRUCTIONS

A *string* is a series of bytes or a series of words in sequential memory locations. A string often consists of ASCII character codes. In the list, a "/" is used to separate different mnemonics for the same instruction. Use the mnemonic which most clearly describes the function of the instruction in a specific application. A "B" in a mnemonic is used to specifically indicate that a string of bytes is to be acted upon. A "W" in the mnemonic is used to indicate that a string of words is to be acted upon.

| | |
|---|---|
| REP | An instruction prefix. Repeat following instruction until CX = 0. |
| REPE/REPZ | An instruction prefix. Repeat instruction until CX = 0 or zero flag ZF ≠ 1. |
| REPNE/REPNZ | An instruction prefix. Repeat until CX = 0 or ZF = 1. |
| MOVS/MOVSB/MOVSW | Move byte or word from one string to another. |
| COMPS/COMPSB/COMPSW | Compare two string bytes or two string words. |
| INS/INSB/INSW | (80186/80188) Input string byte or word from port. |
| OUTS/OUTSB/OUTSW | (80186/80188) Output string byte or word to port. |
| SCAS/SCASB/SCASW | Scan a string. Compare a string byte with a byte in AL or a string word with a word in AX. |
| LODS/LODSB/LODSW | Load string byte into AL or string word into AX. |
| STOS/STOSB/STOSW | Store byte from AL or word from AX into string. |

## PROGRAM EXECUTION TRANSFER INSTRUCTIONS

These instructions are used to tell the 8086 to start fetching instructions from some new address, rather than continuing in sequence.

*Unconditional transfer instructions:*

| | |
|---|---|
| CALL | Call a procedure (subprogram), save return address on stack. |
| RET | Return from procedure to calling program. |
| JMP | Go to specified address to get next instruction. |

*Conditional transfer instructions:*

A "/ " is used to separate two mnemonics which represent the same instruction. Use the mnemonic which most clearly describes the decision condition in a specific program. These instructions are often used after a compare instruction. The terms *below* and *above* refer to unsigned binary numbers. *Above* means larger in magnitude. The terms *greater than* or *less than* refer to signed binary numbers. *Greater than* means more positive.

| | |
|---|---|
| JA/JNBE | Jump if above/Jump if not below or equal. |
| JAE/JNB | Jump if above or equal/Jump if not below. |
| JB/JNAE | Jump if below/Jump if not above or equal. |
| JBE/JNA | Jump if below or equal/Jump if not above. |
| JC | Jump if carry flag CF = 1. |
| JE/JZ | Jump if equal/Jump if zero flag ZF = 1. |
| JG/JNLE | Jump if greater/Jump if not less than or equal. |
| JGE/JNL | Jump if greater than or equal/ Jump if not less than. |
| JL/JNGE | Jump if less than/Jump if not greater than or equal. |
| JLE/JNG | Jump if less than or equal/Jump if not greater than. |
| JNC | Jump if no carry (CF = 0). |
| JNE/JNZ | Jump if not equal/Jump if not zero (ZF = 0). |
| JNO | Jump if no overflow (overflow flag OF = 0). |
| JNP/JPO | Jump if not parity/Jump if parity odd (PF = 0). |
| JNS | Jump if not sign (sign flag SF = 0). |
| JO | Jump if overflow flag OF = 1. |
| JP/JPE | Jump if parity/Jump if parity even (PF = 1). |
| JS | Jump if sign (SF = 1). |

*Iteration control instructions:*

These instructions can be used to execute a series of instructions some number of times. Here mnemonics separated by a "/ " represent the same instruction. Use the one that best fits the specific application.

| | |
|---|---|
| LOOP | Loop through a sequence of instructions until CX = 0. |
| LOOPE/LOOPZ | Loop through a sequence of instructions while ZF = 1 and CX ≠ 0. |
| LOOPNE/LOOPNZ | Loop through a sequence of instructions while ZF = 0 and CX ≠ 0. |
| JCXZ | Jump to specified address if CX = 0. |

If you aren't tired of instructions, continue skimming through the rest of the list. Don't worry if the explanation is not clear to you because we will explain these instructions in detail in later chapters.

*Interrupt instructions:*

| | |
|---|---|
| INT | Interrupt program execution, call service procedure. |
| INTO | Interrupt program execution if OF = 1. |
| IRET | Return from interrupt service procedure to main program. |

*High-level language interface instructions:*

| | |
|---|---|
| ENTER | (80186/80188 only) Enter procedure. |
| LEAVE | (80186/80188 only) Leave procedure. |
| BOUND | (80186/80188 only) Check if effective address within specified array bounds. |

## PROCESSOR CONTROL INSTRUCTIONS

*Flag set/clear instructions:*

| | |
|---|---|
| STC | Set carry flag CF to 1. |
| CLC | Clear carry flag CF to 0. |
| CMC | Complement the state of the carry flag CF. |
| STD | Set direction flag DF to 1 (decrement string pointers). |
| CLD | Clear direction flag DF to 0. |
| STI | Set interrupt enable flag to 1 (enable INTR input). |
| CLI | Clear interrupt enable flag to 0 (disable INTR input). |

*External hardware synchronization instructions:*

| | |
|---|---|
| HLT | Halt (do nothing) until interrupt or reset. |
| WAIT | Wait (do nothing) until signal on the TEST pin is low. |
| ESC | Escape to external coprocessor such as 8087 or 8089. |

| | |
|---|---|
| LOCK | An instruction prefix. Prevents another processor from taking the bus while the adjacent instruction executes. |

*No operation instruction:*

| | |
|---|---|
| NOP | No action except fetch and decode. |

Now that you have skimmed through an overview of the 8086 instruction set, let's see whether you found the instructions needed to implement the "read sensor, add +7, and store result in memory" example program. The IN instruction can be used to read the temperature value from an A/D converter connected to a port. The ADD instruction can be used to add the correction factor of +7 to the value read in. Finally, the MOV instruction can be used to copy the result of the addtion to a memory location. A major point here is that breaking down the programming problem into a sequence of steps makes it easy to find the instruction or small group of instructions that will perform each step. The next section shows you how to write the actual program using the 8086 instructions.

## Writing a Program

### INITIALIZATION INSTRUCTIONS

After finding the instructions you need to do the main part of your program, there are a few additional instructions that you need to determine before you actually write your program. The purpose of these additional instructions is to *initialize* various parts of the system, such as segment registers, flags, and programmable port devices. Segment registers, for example, must be loaded with the upper 16 bits of the address in memory where you want the segment to begin. For our "read temperature sensor, add +7, and store result in memory" example program, the only part we need to initialize is the data segment register. The data segment register must be initialized so that we can copy the result of the addition to a location in memory. If, for example, we want to store data in memory starting at address 00100H, then we want the data segment register to contain the upper 16 bits of this address, 0010H. The 8086 does not have an instruction to move a number directly into a segment register. Therefore, we move the desired number into one of the 16-bit general-purpose registers, then copy it to the desired segment register. Two MOV instructions will do this.

If you are using the stack in your program, then you must include instructions to load the stack segment register and an instruction to load the stack pointer register with the offset of the top of the stack. Most microcomputer systems contain several programmable peripheral devices, such as ports, timers, and controllers. You must include instructions which send control words to these devices to tell them the function you want them to perform. Also, you usually want to include instructions which set or clear the control flags, such as the interrupt enable flag and the direction flag.

The best way to approach the initialization task is to make a checklist of all the registers, programmable devices, and flags in the system you are working on. Then you can mark the ones you need for a specific program and determine the instructions needed to initialize each part. An initialization list for an 8086-based system, such as the SDK-86 prototyping board, might look like the following.

INITIALIZATION LIST

Data segment register DS

Stack segment register SS

Extra segment register ES

Stack pointer register SP

8255 programmable parallel port

8259A priority interrupt controller

8254 programmable counter

8251A programmable serial port

Initialize data variables

Set interrupt enable flag

As you can see, the list can become quite lengthy even though we have not included all the devices a system might commonly have. Note that initializing the code segment register CS is absent from this list. The code segment register is loaded with the correct starting value by the system command you use to run the program. Now let's see how you put all these parts together to make a program.

### A STANDARD PROGRAM FORMAT

In this section we show you how to format your programs if you are going to construct the machine codes for each instruction by *hand*. A later section of this chapter will show you the additional parts you need to add to the program if you are going to use a computer program called an *assembler* to produce the binary codes for the instructions.

To help you write your programs in the correct format, *assembly language coding sheets* such as that shown in Figure 3-4 are available. The ADDRESS column is used for the address or the offset of a code byte or data byte. The actual code bytes or data bytes are put in the DATA/CODE column. A *label* is a name which represents an address referred to in a jump or call instruction; labels are put in the LABELS column. A label is followed by a colon (:) if it is used by a jump or call instruction in the same code segment. The MNEM column contains the opcode mnemonics for the instructions. The OPERAND(S) column contains the registers, memory locations, or data acted upon by the instructions. A COMMENTS column gives you space to describe the function of the instruction for future reference.

Figure 3-4, p. 46, shows how instructions for the "read temperature, add +7, store result in memory" program can be written in sequence on a coding sheet. We will discuss here the operation of these instructions