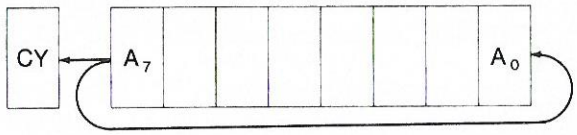
RLC



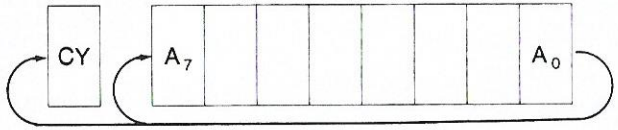
$(A_{n+1}) \leftarrow (A_n)$
$(A_0) \leftarrow (A_7)$
$(CY) \leftarrow (A_7)$

Flags: CY

When the RLC instruction is executed, each bit of the accumulator is shifted left, $(A_{n+1}) \leftarrow (A_n)$, and the most significant bit, A_7, is copied into the least significant bit, A_0, and into the carry, CY.

The corresponding rotate instruction in the opposite direction is the *rotate accumulator right*, RRC, instruction.

RRC

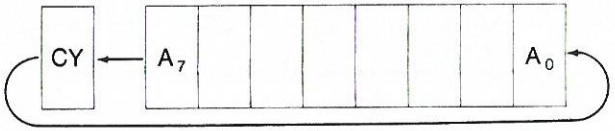


$(A_{n-1}) \leftarrow (A_n)$
$(A_7) \leftarrow (A_0)$
$(CY) \leftarrow (A_0)$

Flags: CY

Two other rotate instructions treat the accumulator and carry together, as if they constitute a 9-bit register. The *rotate accumulator left through carry*, RAL, instruction has the following form:

RAL

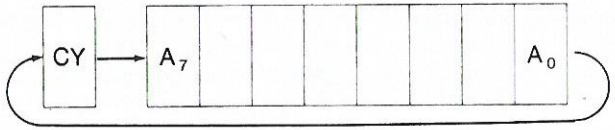


$(A_{n+1}) \leftarrow (A_n)$
$(CY) \leftarrow (A_7)$
$(A_0) \leftarrow (CY)$

Flags: CY

A 9-bit rotation in the opposite direction is implemented with the *rotate accumulator right through carry* instruction.

RAR



$(A_{n-1}) \leftarrow (A_n)$
$(CY) \leftarrow (A_0)$
$(A_7) \leftarrow (CY)$

Flags: CY

Note the apparent transposition of mnemonics and instruction descriptions for the rotate instructions.

RAL*: *Rotate accumulator Left through Carry
RLC*: *Rotate Accumulator Left

Consider a situation where input port 7 inputs sequentially generated BCD digits, as the least significant 4 bits of a byte. After being input, the BCD digits are packed, two digits to a byte, and stored in memory. The following program segment carries out part of this task: it inputs and packs two digits, leaving the result in the accumulator.

```
IN PORT7    ;input most significant of the two BCD digits
ANI 0FH     ;clear most significant 4 bits of byte input
RLC         ;shift BCD digit four places to the left
RLC
RLC
RLC
MOV B, A    ;store shifted digit in B
IN PORT7    ;input next digit
ANI 0FH     ;clear most significant 4 bits of byte input
ORA B
```

4.4 FLOWCHARTING

Once the functions to be implemented in software have been defined, the designer selects or develops appropriate algorithms for their implementation. An ***algorithm*** is a computational or logical method of producing a desired result. The development and representation of an algorithm are facilitated by a ***flowchart***, a graphic method of representing the order in which operations are carried out and the decisions that determine that order.

The flowchart is the software counterpart of the block diagram used in hardware design. In hardware design, the least detailed block diagram indicates the major hardware subsystems required to implement the overall system function and the information transfer among these subsystems. In software design, the least detailed flowchart indicates the major software subsystems required to implement the overall system function and their order of execution.

Just as the hardware designer must know the function and characteristics of MSI circuits in detail, the software designer must know the functions and characteristics of available instructions in similar detail. For an instruction, its function is represented by its register transfer expressions and a statement of the flags affected by its execution. The characteristics of an instruction are the number of bytes, machine cycles, and states associated with it.

In flowcharts a rectangle represents an operation or process, and a diamond represents a decision (see Fig. 4.4-1). An oval represents the beginning and/or the end of the instruction sequence. Brief statements indicating the operations or decisions associated with each symbol appear inside it. The symbols are interconnected by directed line segments that indicate program flow, just as directed line segments in a hardware block diagram indicate information flow. In an overall