Part 1: Data Transfer Group MOV Instructions

Register MOV Instructions:

The mnemonic **MOV** represents a simple data move: between two registers, between a register and a memory location, or between to two memory locations. But here, the term “move” can be deceptive. In the physical world, the term implies an object physically changes location, transferring from one place to another. In programming, however, a MOV instruction simply places a copy of a byte of data into another location. There are two important facts to remember when using the MOV instruction:

- The copied data remains intact within its source location.
- The copied data overwites the data present in the destination register or memory location.

Knowing the proper syntax for writing MOV instructions is also important. Table 9.1, beginning on page 73 of the C8051F330/12/3/4/5 datasheet is an excellent quick reference for verifying assembly language syntax and recalling the function of each instrction. (The entire Data Transfer group of the CIP-51 instruction set is contained on page 73 of the datasheet.)

Assume, for example, in the course of writing an assembly language program, we decide to save the contents of the accumulator in a general purpose register (R0 through R7). We view view Table 9.1 of the datasheet and find two instructions involving data transfer between the accumulator and a general purpose register: **MOV A, Rn** and **MOV Rn, A**. Here, **A** represents the accumulator and **Rn** represents any of the eight general purpose registers within a given bank. (The register banks are discussed in detail in Lesson Seven of this course book.) The operand following the MOV mnemonic represents the destination register. The operand following the comma represents the source register. Assume for this example, we choose to store the accumulator data in R4. We then would write the code as **MOV R4, A**. At a later point in the code, to place the R4 data into the accumulator, we would write **MOV A, R4**.

The basic MOV instructions encompass the 256-bytes of the internal data RAM (256 byte SRAM) as indicated in Figure 4-1. The **MOVX** instruction must be used for moves involving the 512-byte external data memory (designated as XRAM in Figure 4-1). (The use of the MOVX instruction and the function of the data pointer (**DPTR**) in accessing the external memory of the C8051F330 MCU are covered later in this lesson.) Register-to-register moves within the internal data RAM typically involve the accumulator, the B register, and the four banks of general purpose registers. As indicated in Figure 4-1, the accumulator and register B are special function registers (**SFR’s**) contained within the SFR section of the upper 128 bytes of the internal data RAM. The four banks of general purpose registers comprise the lower 32 bytes of the internal data RAM. Each bank consists of eight registers designated as R0 through R7. (Methods for accessings these four banks are covered in Lesson 7.)

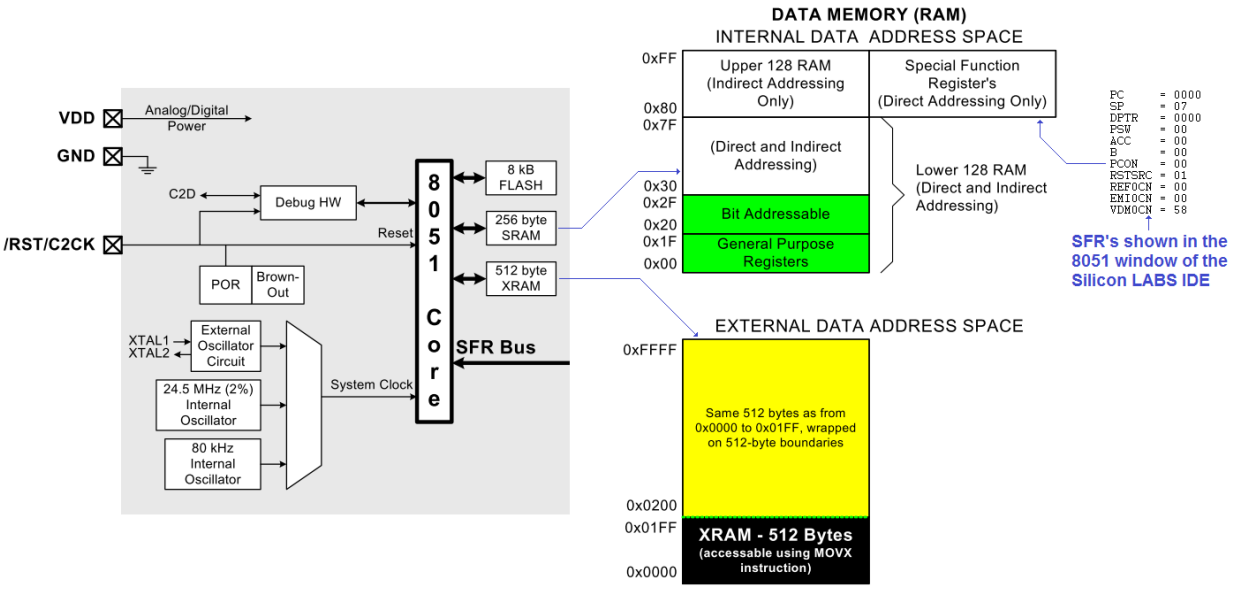

Figure 4-1

Although not shown in Table 9.1 of the C8051F330/12/3/4/5 datasheet, data transfers are possible between the accumulator and register B, using either the **MOV A, B** or **MOV B, A** instruction. Also (again not shown in Table 9.1), **MOV B, Rn** and **MOV Rn, B** instructions are also possible. However, a datat tranfer cannot be made directly between two general purpose registers. For example, when using the Silicon Labs IDE, attempting to enter **MOV R5, R4** as a line of code will result in an INVALID REGISTER error message during invocation of the assembler.

Example 4-1:

Figure 4-2 contains a brief code segment illustrating register-to-register data transfer. The first instruction, **CLR A**, which is found in the Logical Operations Group of Table 9.1, clears the contents of the accumulator to 0x00. There is no CLR B instruction in the CIP-51 instruction set, so a **MOV B, A** command is used to place 0x00 in register B. The **MOV R3, A** and **MOV R4, B** instructions place 0x00 in general purpose registers R3 and R4.

The **MOV A, #3BH** and **MOV A, #3AH** instructions in the fifth and seventh steps of the code segment are examples of **immediate addressing**. (Immediate addressing and data constants were introduced in Lesson Three.) Note the syntax for specifying immediate data after the comma: Immediate data is always preceded by the numeric symbol #. Hexadecimal numbers are followed by the suffix **H**. During the sixth step of the example, data contained by the accumulator is copied to register B. At that point, both the accumulator and register B hold 0x3B. In step 7, the contents of the accumulator is overwritten with immediate data byte 0x3A. In steps 8 and 9, the A and B data bytes are copied to registers R3 and R4.

```
Main_1:
CLR A
MOV B, A
MOV R3, A
MOV R4, B
MOV A, #3BH
MOV B, A
MOV A, #3AH
MOV R3, A
MOV R4, B
JMP Main_1
```

Figure 4-2

At this juncture, you are advised to procede to Part 4 of this Lesson and perform **Programming Exercise 4-1**.

Direct MOV Instructions:

Direct addressing can be used to transfer data between a register and data RAM location. It can also be used to transfer data between two RAM locations. Here, the term "direct" indicates the address of a RAM location is directly identified as an operand within a line of code. For example, **MOV R2, 40H** causes the data byte at internal data RAM location 40H to be copied to general purpose register R2. The H suffix is necessary to specify the memory address is hexadecimal. Maintaining a hexadecimal format is advisable in direct addressing because internal RAM locations, as seen in the Silicon Labs IDE RAM window, are displayed in a **hex dump** format. As shown in Figure 4-3, a hex dump is a display of the contents of an area of data memory in a row and column matrix.

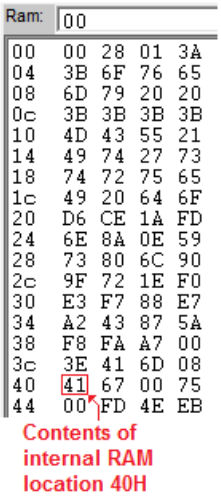

Figure 4-3

Assume the programmer enters a line of code as **MOV R2, 40**, inadvertently omitting the H suffix from the memory location, the code would execute without error. However, by not detecting the H suffix for the address operand, the assembler reads the operand 40 as a decimal value. Thus, in creating the machine code, the assembler replaces 40 with 28, the hexadecimal equivalent of decimal 40. This is explained further in the Example 4-2 and proven in Programming Exercise 4-2.

Example 4-2:

Figure 4-4 contains a brief code segment illustrating data transfer between SFR registers and internal data RAM locations using direct addressing. Figure 4-4a contains the assembly language segment, while Figure 4-4b shows the resultant machine code as seen in the IDE Disassembly Window.

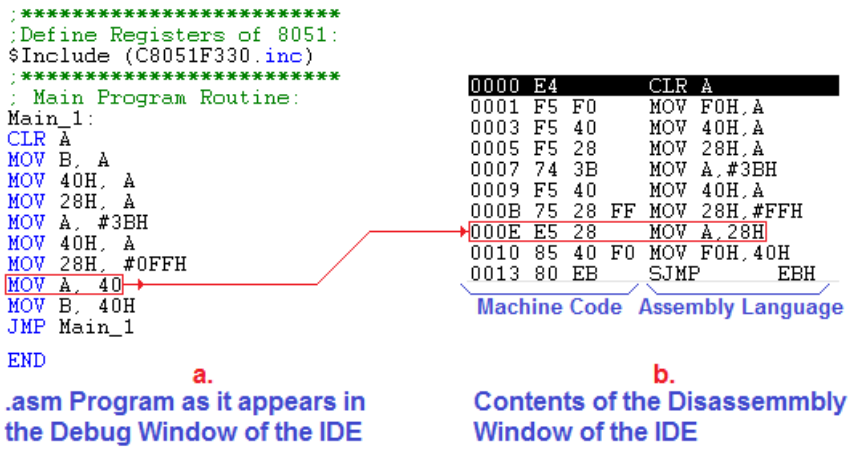

Figure 4-4

The first four lines of code in Figure 4-4 simply clear the accumulator, register B, and the two memory addresses accessed during the program. This allows the program to continously loop and show the changing data during performance of Programming Exercise 4-2. The fifth line of code places immediate data 0x3B into the accumulator and the following line of code places it in memory location 40H, as indicated by the contents of the IDE RAM window Figure 4-5. The following line of code uses both immediate and direct addressing. Here, as seen in Figure 4-5, immediate data byte 0xFF is placed directly into memory location 28H.

Line eight of the code uses direct addressing to move the contents of a memory location into the accumulator. However, because the address operand is written as 40 rather than 40H, the assembler assumes the number to be decimal and automatically converts it to hexadecimal. Thus it specifies the source address of the data as 28H. This is indicated by the connecting red rectangles in Figure 4-4. Thus, after execution of line eight, the accumulator should contain 0xFF, the immediate data previously moved into memory location 28H. The following **MOV B, 40H** instruction places 0x3B, previously moved to RAM location 40H, into register B. The following JMP Main_1 instruction allows the program to loop continuously, until halted by the clicking the IDE RESET button.

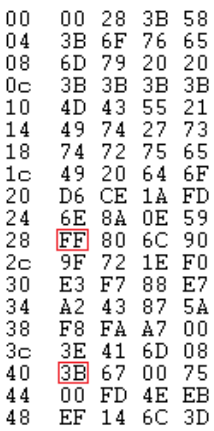

Figure 4-5

At this juncture, you are advised to procede to Part 4 of this Lesson and perform **Programming Exercise 4-2**.

Indirect MOV Instructions:

As was illustrated in Example 4-2, direct memory moves are accomplished by directly specifying the RAM location after after the MOV instruction. With **indirect addressing**, general purpose register R0 or R1 within a given bank is used as an 8-bit memory pointer. **MOV A, @Ri** is an example of indirect addressing. This command could be interpreted as, “Move into the accumulator the data pointed to indirectly by the contents of the given register.” Thus, the data held by that register represents the source address of the data to be placed into the accumulator.

Consider the following code snippet:

```
MOV R1, #30H

MOV A, @R1
```

The first instructon places immediate data 0x30H into general purpose register R1, allowing memory location 30H to be indirectly accessed in the next line of code. The following **MOV A, @R1** instruction moves the contents of internal RAM location 30H into the accumulator.

MOV @Ri, A is another example of indirect addressing. This command could be interpreted as, “Move into the the RAM location pointed to to by the given register, the contents of the accumulator.”

Consider the following code snippet:

MOV R0, #35H

MOV @R0, A

The first instruction places immediate data 0x35H into general purpose register R0, allowing memory location 35H to become the destination for data held by the accumulator. The following **MOV @R0, A** instruction moves the contents of the accumulator to internal data RAM location 35H.

Let’s check our understanding:

1) A student enters the following code into the Debug window of the IDE. When he attempts to assemble the code he sees error messages.

```
;*****************************
;Define Registers of 8051:
$Include (C8051F330.inc)
;*****************************
; Main Program Routine:
Main_1:
CLR A
MOV B, A
MOV R3, #35H
Inc_Loop:
INC B
MOV @R3, B
JMP Inc_Loop

END
```

The reason for the error messages is:

a) The increment (INC) instruction can only be used with the accumulator.

b) Register R3 cannot be used as a memory pointer during indirect addressing.

c) A MOVX instruction should have been used in line 5 of the code.

d) None of the above.

2) In the above program example, line 3 of the code should place the contents of memory location 35H into register R3.

True/False

Example 4-3:

Figure 4-6 illustrates the execution of the MOV @Ri, A instruction. Here, the first instruction, **CLR A**, clears the accumulator to 0x00. With R0 being used as the memory pointer, the next instruction specifies address 35H as the destination address for the incrementing accumulator data. During the Inc_Mem_Loop, the accumulator data is continuously incremented and moved to memory location 35H via the **MOV @R0, A** instruction.

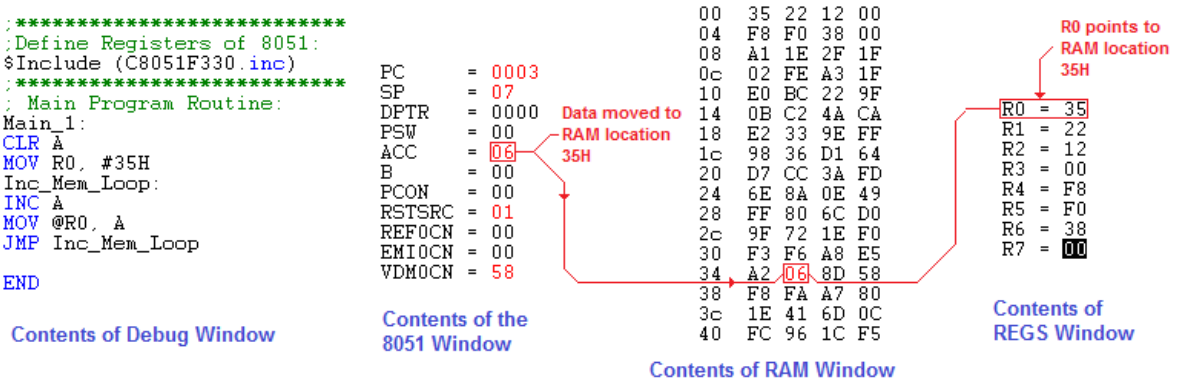

Figure 4-6

Figure 4-7 illustrates the execution of the MOV A, @Ri instruction. Here, direct addressing is used to move immediate data 0xFF to RAM location 35H. Next, with R0 again being used as the memory pointer, 35H is specified as the source location for moving data to the accumulator and register B. During the Dec_Mem_Loop, the data at RAM location 35H is continuously decremented by the DEC 35H instruction and moved to the accumulator and Register B via the **MOV A, @R0** and **MOV B, @R0** instructions.

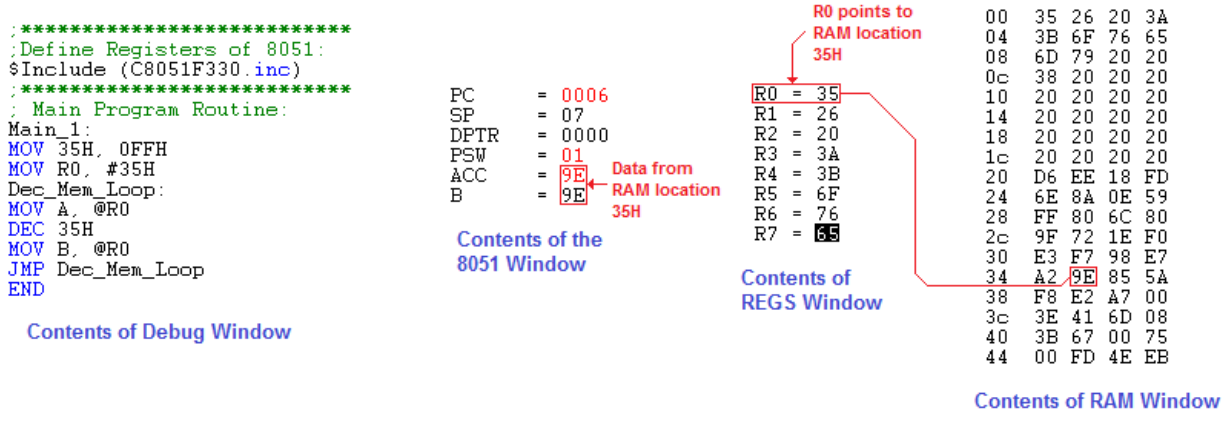

Figure 4-7

Indirect addressing can be used to move immediate data into a memory location. Consider the code snippet below, which utilizes the **MOV @Ri. #data** instruction:

```
MOV R1, #53H
MOV @R1, #4BH
```

The first instructon places immediate data 0x53H into general purpose register R1, allowing that register to point to RAM location 53H. The following **MOV @R1, #4BH** instruction places immediate data byte 0x4B into memory location 53H.

General-purpose registers R0 and R1 can be used to move data from one memory location to another using a combination of indirect and direct addressing.

Consider the code snippet below:

```
MOV R1, #53H
MOV R0, #4AH
MOV @R1, #4BH
MOV @R0, 53H
```

Here, in the first two lines of code, immediate data is placed in registers R1 and R0, allowing R1 to point to RAM location 53H and R0 to point to address 4AH. In the third line of code, immediate data 0x4B is moved into RAM location 53H. In the fourth line of code, the contents of memory location 53H, which is now 0x4B, is placed into memory location 4AH, which is pointed to by register R0.

At this juncture, you are advised to procede to Part 4 of this Lesson and perform **Programming Exercise 4-3**.