CHAPTER

5

Arithmetic Operations

Chapter Outline

Introduction

Applications of microcontrollers often involve performing mathematical calculations on data in order to alter program flow and modify program actions. A microcontroller is not designed to be a "number cruncher," as is a general-purpose computer. The domain of the microcontroller is that of controlling events as they change (real-time control). A sufficient number of mathematical opcodes must be provided, however, so that calculations associated with the control of simple processes can be done, in real time, as the controlled system operates. When faced with a control problem, the programmer must know whether the 8051 has sufficient capability to expeditiously handle the required data manipulation. If it does not, a higher performance model must be chosen.

The 24 arithmetic opcodes are grouped into the following types:

Mnemonic	**Operation**
INC destination	Increment destination by 1
DEC destination	Decrement destination by 1
ADD/ADDC destination,source	Add source to destination without/with carry (C) flag
SUBB destination,source	Subtract, with carry, source from destination
MUL AB	Multiply the contents of registers A and B

DIV AB	Divide the contents of register A by the contents of register B
DA A	Decimal Adjust the A register

The addressing modes for the destination and source are the same as those discussed in Chapter 3: immediate, register, direct, and indirect.

Flags

A key part of performing arithmetic operations is the ability to store certain results of those operations that affect the way in which the program operates. For example, adding together two one-byte numbers results in a one-byte partial sum, because the 8051 is and eight-bit machine. But it is possible to get a 9-bit result when adding two 8-bit numbers. The ninth bit must be stored also, so the need for a one-bit register, or carry flag in this case, is identified. The program will then have to deal with the ninth bit, perhaps by adding it to a higher order byte in a multiple-byte addition scheme. Similar actions may have to be taken when a larger byte is subtracted from a smaller one. In this case, a borrow is necessary and must be dealt with by the program.

The 8051 has several dedicated latches, or flags, that store results of arithmetic operations. Opcodes covered in Chapter 6 are available to alter program flow based upon the state of the flags. Not all instructions change the flags, but many a programming error has been made by a forgetful programmer who overlooked an instruction that does change a flag.

The 8051 has four arithmetic flags: the carry (C), auxiliary carry (AC), overflow (OV), and parity (P).

Instructions Affecting Flags

The C, AC, and OV flags are arithmetic flags. They are set to 1 or cleared to 0 automatically, depending upon the outcomes of the following instructions. The following instruction set includes *all* instructions that modify the flags and is not confined to arithmetic instructions:

INSTRUCTION MNEMONIC	FLAGS AFFECTED		
ADD	C	AC	OV
ADDC	C	AC	OV
ANL C,direct	C		
CJNE	C		
CLR C	C = 0		
CPL C	$C = \overline{C}$		
DA A	C		
DIV	C = 0		OV
MOV C,direct	C		
MUL	C = 0		OV
ORL C,direct	C		
RLC	C		
RRC	C		
SETB C	C = 1		
SUBB	C	AC	OV

One should remember, however, that the flags are all stored in the PSW. *Any* instruction that can modify a bit or a byte in that register (MOV, SETB, XCH, etc.) changes the flags. This type of change takes conscious effort on the part of the programmer.

A flag may be used for more than one type of result. For example, the C flag indicates a carry out of the lower byte position during addition and indicates a borrow during subtraction. The instruction that *last* affects a flag determines the use of that flag.

The parity flag is affected by every instruction executed. The P flag will be set to a 1 if the number of 1's in the A register is odd and will be set to 0 if the number of 1's is even. All 0's in A yield a 1's count of 0, which is considered to be *even*. Parity check is an elementary error-checking method and is particularly valuable when checking data received via the serial port.

Incrementing and Decrementing

The simplest arithmetic operations involve adding or subtracting a binary 1 and a number. These simple operations become very powerful when coupled with the ability to repeat the operation—that is, to "INCrement" or "DECrement"—until a desired result is reached.[1] Register, Direct, and Indirect addresses may be INCremented or DECremented. No math flags (C, AC, OV) are affected.

The following table lists the increment and decrement mnemonics.

Mnemonic	**Operation**
INC A	Add a one to the A register
INC Rr	Add a one to register Rr
INC add	Add a one to the direct address
INC @ Rp	Add a one to the contents of the address in Rp
INC DPTR	Add a one to the 16-bit DPTR
DEC A	Subtract a one from register A
DEC Rr	Subtract a one from register Rr
DEC add	Subtract a one from the contents of the direct address
DEC @ Rp	Subtract a one from the contents of the address in register Rp

Note that increment and decrement instructions that operate on a port direct address alter the *latch* for that port.

The following table shows examples of increment and decrement arithmetic operations:

Mnemonic	**Operation**
MOV A,#3Ah	A = 3Ah
DEC A	A = 39h
MOV R0,#15h	R0 = 15h
MOV 15h,#12h	Internal RAM address 15h = 12h
INC @R0	Internal RAM address 15h = 13h
DEC 15h	Internal RAM address 15h = 12h
INC R0	R0 = 16h
MOV 16h,A	Internal RAM address 16h = 39h
INC @R0	Internal RAM address 16h = 3Ah
MOV DPTR,#12FFh	DPTR = 12FFh
INC DPTR	DPTR = 1300h
DEC 83h	DPTR = 1200h (SFR 83h is the DPH byte)

[1]This subject will be explored in Chapter 6.

CAUTION

Remember: *No* math flags are affected.

All 8-bit address contents overflow from FFh to 00h.

DPTR is 16 bits; DPTR overflows from FFFFh to 0000h.

The 8-bit address contents underflow from 00h to FFh.

There is no DEC DPTR to match the INC DPTR.

Addition

All addition is done with the A register as the destination of the result. All addressing modes may be used for the source: an immediate number, a register, a direct address, and an indirect address. Some instructions include the carry flag as an additional source of a single bit that is included in the operation at the *least* significant bit position.

The following table lists the addition mnemonics.

Mnemonic	**Operation**
ADD A,#n	Add A and the immediate number n; put the sum in A
ADD A,Rr	Add A and register Rr; put the sum in A
ADD A,add	Add A and the address contents; put the sum in A
ADD A,@Rp	Add A and the contents of the address in Rp; put the sum in A

Note that the C flag is set to 1 if there is a carry out of bit position 7; it is cleared to 0 otherwise. The AC flag is set to 1 if there is a carry out of bit position 3; it is cleared otherwise. The OV flag is set to 1 if there is a carry out of bit position 7, but not bit position 6 or if there is a carry out of bit position 6 but not bit position 7, which may be expressed as the logical operation

$$OV = C7 \text{ XOR } C6$$

Unsigned and Signed Addition

The programmer may decide that the numbers used in the program are to be unsigned numbers—that is, numbers that are 8-bit positive binary numbers ranging from 00h to FFh. Alternatively, the programmer may need to use both positive and negative signed numbers.

Signed numbers use bit 7 as a *sign* bit in the most significant byte (MSB) of the *group* of bytes *chosen* by the programmer to represent the largest number to be needed by the program. Bits 0 to 6 of the MSB, and any other bytes, express the magnitude of the number. Signed numbers use a 1 in bit position 7 of the MSB as a negative sign and a 0 as a positive sign. Further, all negative numbers are not in true form, but are in 2's complement form. When doing signed arithmetic, the programmer must *know* how large the largest number is to be—that is, how many bytes are needed for each number.

In signed form, a single byte number may range in size from 10000000b, which is −128d to 01111111b, which is +127d. The number 00000000b is 000d and has a positive sign, so there are 128d negative numbers and 128d positive numbers. The C and OV flags have been included in the 8051 to enable the programmer to use either numbering scheme.

Adding or subtracting unsigned numbers may generate a carry flag when the sum exceeds FFh or a borrow flag when the minuend is less than the subtrahend. The OV flag is not used for unsigned addition and subtraction. Adding or subtracting signed numbers can lead to carries and borrows in a similar manner, and to overflow conditions due to the actions of the sign bits.

Unsigned Addition

Unsigned numbers make use of the carry flag to detect when the result of an ADD operation is a number larger than FFh. If the carry is set to one after an ADD, then the carry can be added to a higher order byte so that the sum is not lost. For instance,

$$\begin{array}{rl} 95\text{d} = & 01011111\text{b} \\ \underline{189\text{d}} = & \underline{10111101\text{b}} \\ 284\text{d} & 1\ 00011100\text{b} = 284\text{d} \end{array}$$

The C flag is set to 1 to account for the carry out from the sum. The program could add the carry flag to another byte that forms the second byte of a larger number.

Signed Addition

Signed numbers may be added two ways: addition of like signed numbers and addition of unlike signed numbers. If unlike signed numbers are added, then it is not possible for the result to be larger than −128d or +127d, and the sign of the result will always be correct. For example,

$$\begin{array}{rl} -001\text{d} = & 11111111\text{b} \\ \underline{+027\text{d}} = & \underline{00011011\text{b}} \\ +026\text{d} & 00011010\text{b} = +026\text{d} \end{array}$$

Here, there is a carry from bit 7 so the carry flag is 1. There is also a carry from bit 6, and the OV flag is 0. For this condition, no action need be taken by the program to correct the sum.

If positive numbers are added, there is the possibility that the sum will exceed +127d, as demonstrated in the following example:

$$\begin{array}{rl} +100\text{d} = & 01100100\text{b} \\ \underline{+050\text{d}} = & \underline{00110010\text{b}} \\ +150\text{d} & 10010110\text{b} = -106\text{d} \end{array}$$

Ignoring the sign of the result, the magnitude is seen to be +22d which would be correct if we had some way of accounting for the +128d, which, unfortunately, is larger than a single byte can hold. There is no carry from bit 7 and the carry flag is 0; there is a carry from bit 6 so the OV flag is 1.

An example of adding two positive numbers that do not exceed the positive limit is:

$$\begin{array}{rl} +045\text{d} = & 00101101\text{b} \\ \underline{+075\text{d}} = & \underline{01001011\text{b}} \\ +120\text{d} & 01111000\text{b} = 120\text{d} \end{array}$$

Note that there are no carries from bits 6 or 7 of the sum; the carry and OV flags are both 0.

The result of adding two negative numbers together for a sum that does not exceed the negative limit is shown in this example:

$$\begin{array}{rl} -030\text{d} = & 11100010\text{b} \\ \underline{-050\text{d}} = & \underline{11001110\text{b}} \\ -080\text{d} & 10110000\text{b} = -080\text{d} \end{array}$$

Here, there is a carry from bit 7 and the carry flag is 1; there is a carry from bit 6 and the OV flag is 0. These are the same flags as the case for adding unlike numbers; no corrections are needed for the sum.

When adding two negative numbers whose sum does exceed −128d, we have

```
 −070d = 10111010b
 −070d = 10111010b
 −140d   01110100b = +116d
```

Or, the magnitude can be interpreted as −12d, which is the remainder after a carry out of −128d. In this example, there is a carry from bit position 7, and no carry from bit position 6, so the carry and the OV flags are set to 1. The magnitude of the sum is correct; the sign bit must be changed to a 1.

From these examples the programming actions needed for the C and OV flags are as follows:

FLAGS		ACTION
C	OV	
0	0	None
0	1	Complement the sign
1	0	None
1	1	Complement the sign

A general rule is that *if the OV flag is set, then complement the sign.* The OV flag also signals that the sum exceeds the largest positive or negative numbers thought to be needed in the program.

Multiple-Byte Signed Arithmetic

The nature of multiple-byte arithmetic for signed and unsigned numbers is distinctly different from single byte arithmetic. Using more than one byte in unsigned arithmetic means that carries or borrows are propagated from low-order to high-order bytes by the simple technique of adding the carry to the next highest byte for addition and subtracting the borrow from the next highest byte for subtraction.

Signed numbers appear to behave like unsigned numbers until the last byte is reached. For a signed number, the seventh bit of the highest byte is the sign; if the sign is negative, then the *entire* number is in 2's complement form.

For example, using a two-byte signed number, we have the following examples:

```
+32767d = 01111111 11111111b = 7FFFh
+00000d = 00000000 00000000b = 0000h
−00001d = 11111111 11111111b = FFFFh
−32768d = 10000000 00000000b = 8000h
```

Note that the lowest byte of the numbers 00000d and −32768d are exactly alike, as are the lowest bytes for +32767d and −00001d.

For multi-byte signed number arithmetic, then, the lower bytes are treated as unsigned numbers. All checks for overflow are done only for the highest order byte that contains the sign. An overflow at the highest order byte is not usually recoverable. The programmer has made a *mistake* and probably has made no provisions for a number larger than planned. Some error acknowledgment procedure, or user notification, should be included in the program if this type of mistake is a possibility.

The preceding examples show the need to add the carry flag to higher order bytes in signed and unsigned addition operations. Opcodes that accomplish this task are similar to the ADD mnemonics: A C is appended to show that the carry bit is added to the sum in bit position 0.

The following table lists the add with carry mnemonics:

Mnemonic	Operation
ADDC A,#n	Add the contents of A, the immediate number n, and the C flag; put the sum in A
ADDC A,add	Add the contents of A, the direct address contents, and the C flag; put the sum in A
ADDC A,Rr	Add the contents of A, register Rr, and the C flag; put the sum in A
ADDC A,@Rp	Add the contents of A, the contents of the indirect address in Rp, and the C flag; put the sum in A

Note that the C, AC, and OV flags behave exactly as they do for the ADD commands.

The following table shows examples of ADD and ADDC multiple-byte signed arithmetic operations:

Mnemonic	Operation
MOV A,#1Ch	A = 1Ch
MOV R5,#0A1h	R5 = A1h
ADD A,R5	A = BDh; C = 0, OV = 0
ADD A,R5	A = 5Eh; C = 1, OV = 1
ADDC A,#10h	A = 6Fh; C = 0, OV = 0
ADDC A,#10h	A = 7Fh; C = 0, OV = 0

CAUTION

ADDC is normally used to add a carry after the LSB addition in a multi-byte process. ADD is normally used for the LSB addition.

Subtraction

Subtraction can be done by taking the 2's complement of the number to be subtracted, the subtrahend, and adding it to another number, the minuend. The 8051, however, has commands to perform direct subtraction of two signed or unsigned numbers. Register A is the destination address for subtraction. All four addressing modes may be used for source addresses. The commands treat the carry flag as a borrow and always subtract the carry flag as part of the operation.

The following table lists the subtract mnemonics.

Mnemonic	Operation
SUBB A,#n	Subtract immediate number n and the C flag from A; put the result in A
SUBB A,add	Subtract the contents of add and the C flag from A; put the result in A
SUBB A,Rr	Subtract Rr and the C flag from A; put the result in A
SUBB A,@Rp	Subtract the contents of the address in Rp and the C flag from A; put the result in A

Note that the C flag is set if a borrow is needed into bit 7 and reset otherwise. The AC flag is set if a borrow is needed into bit 3 and reset otherwise. The OV flag is set if there is a borrow into bit 7 and not bit 6 or if there is a borrow into bit 6 and not bit 7. As in the case for addition, the OV Flag is the XOR of the borrows into bit positions 7 and 6.

Unsigned and Signed Subtraction

Again, depending on what is needed, the programmer may choose to use bytes as signed or unsigned numbers. The carry flag is now thought of as a borrow flag to account for situations when a larger number is subtracted from a smaller number. The OV flag indicates results that must be adjusted whenever two numbers of unlike signs are subtracted and the result exceeds the planned signed magnitudes.

Unsigned Subtraction

Because the C flag is always subtracted from A along with the source byte, it must be set to 0 if the programmer does not want the flag included in the subtraction. If a multi-byte subtraction is done, the C flag is cleared for the first byte and then included in subsequent higher byte operations.

The result will be in true form, with no borrow if the source number is smaller than A, or in 2's complement form, with a borrow if the source is larger than A. These are *not* signed numbers, as all eight bits are used for the magnitude. The range of numbers is from positive 255d (C = 0, A = FFh) to negative 255d (C = 1, A = 01h).

The following example demonstrates subtraction of larger number from a smaller number:

$$\begin{array}{rrcl}
 & 015\text{d} & = & 00001111\text{b} \\
\textit{SUBB} & \underline{\textit{100d}} & = & \underline{\textit{01100100b}} \\
 & -085\text{d} & 1 & 10101011\text{b} = 171\text{d}
\end{array}$$

The C flag is set to 1, and the OV flag is set to 0. The 2's complement of the result is 085d.

The reverse of the example yields the following result:

$$\begin{array}{rcl}
100\text{d} & = & 01100100\text{b} \\
\underline{015\text{d}} & = & \underline{00001111\text{b}} \\
085\text{d} & & 01010101\text{b} = 085\text{d}
\end{array}$$

The C flag is set to 0, and the OV flag is set to 0. The magnitude of the result is in true form.

Signed Subtraction

As is the case for addition, two combinations of unsigned numbers are possible when subtracting: subtracting numbers of like and unlike signs. When numbers of like sign are subtracted, it is impossible for the result to exceed the positive or negative magnitude limits of +127d or −128d, so the magnitude and sign of the result do not need to be adjusted, as shown in the following example:

$$\begin{array}{rrcll}
 & +100\text{d} & = & 01100100\text{b} & \text{(Carry flag = 0 before SUBB)} \\
\text{SUBB} & \underline{+126\text{d}} & = & \underline{01111110\text{b}} & \\
 & -026\text{d} & 1 & \textit{11100110b} = -026\text{d} &
\end{array}$$

There is a borrow into bit positions 7 and 6; the carry flag is set to 1, and the OV flag is cleared.

The following example demonstrates using two negative numbers:

$$\begin{array}{rrcll} & -061\text{d} & = & 11000011\text{b} & \text{(Carry flag = 0 before SUBB)} \\ \text{SUBB} & \underline{-116\text{d}} & = & \underline{10001100\text{b}} & \\ & +055\text{d} & & 00110111\text{b} = +55\text{d} & \end{array}$$

There are no borrows into bit positions 6 or 7, so the OV and carry flags are cleared to zero.

An overflow is possible when subtracting numbers of opposite sign because the situation becomes one of adding numbers of like signs, as can be demonstrated in the following example:

$$\begin{array}{rrcll} & -099\text{d} & = & 10011101\text{b} & \text{(Carry flag = 0 before SUBB)} \\ \text{SUBB} & \underline{+100\text{d}} & = & \underline{01100100\text{b}} & \\ & -199\text{d} & & 00111001\text{b} = +057\text{d} & \end{array}$$

Here, there is a borrow into bit position 6 but not into bit position 7; the OV flag is set to 1, and the carry flag is cleared to 0. Because the OV flag is set to 1, the result must be adjusted. In this case, the magnitude can be interpreted as the 2's complement of 71d, the remainder after a carry out of 128d from 199d. The magnitude is correct, and the sign needs to be corrected to a 1.

The following example shows a positive overflow:

$$\begin{array}{rrcll} & +087\text{d} & = & 01010111\text{b} & \text{(Carry flag = 0 before SUBB)} \\ \text{SUBB} & \underline{-052\text{d}} & = & \underline{11001100\text{b}} & \\ & +139\text{d} & & 10001011\text{b} = -117\text{d} & \end{array}$$

There is a borrow from bit position 7, and no borrow from bit position 6; the OV flag and the carry flag are both set to 1. Again the answer must be adjusted because the OV flag is set to one. The magnitude can be interpreted as a +011d, the remainder from a carry out of 128d. The sign must be changed to a binary 0 and the OV condition dealt with.

The general rule is that *if the OV flag is set to 1, then complement the sign bit.* The OV flag also signals that the result is greater than −128d or +127d.

Again, it must be emphasized: When an overflow occurs in a program, an error has been made in the estimation of the largest number needed to successfully operate the program. Theoretically, the program could resize every number used, but this extreme procedure would tend to hinder the performance of the microcontroller.

Note that for all the examples in this section, it is *assumed* that the carry flag = 0 before the SUBB. The carry flag must be 0 before any SUBB operation that depends upon C = 0 is done.

The following table lists examples of SUBB multiple-byte signed arithmetic operations:

Mnemonic	**Operation**
MOV 0D0h,#00h	Carry flag = 0
MOV A,#3Ah	A = 3Ah
MOV 45h,#13h	Address 45h = 13h
SUBB A,45h	A = 27h; C = 0, OV = 0
SUBB A,45h	A = 14h; C = 0, OV = 0
SUBB A,#80h	A = 94h; C = 1, OV = 1
SUBB A,#22h	A = 71h; C = 0, OV = 0
SUBB A,#0FFh	A = 72h; C = 1, OV = 0

CAUTION

Remember to set the carry flag to zero if it is not to be included as part of the subtraction operation.

Multiplication and Division

The 8051 has the capability to perform 8-bit integer multiplication and division using the A and B registers. Register B is used solely for these operations and has no other use except as a location in the SFR space of RAM that could be used to hold data. The A register holds one byte of data before a multiply or divide operation, and one of the result bytes after a multiply or divide operation.

Multiplication and division treat the numbers in registers A and B as unsigned. The programmer must devise ways to handle signed numbers.

Multiplication

Multiplication operations use registers A and B as both source and destination addresses for the operation. The unsigned number in register A is multiplied by the unsigned number in register B, as indicated in the following table:

Mnemonic	Operation
MUL AB	Multiply A by B; put the low-order byte of the product in A, put the high-order byte in B

The OV flag will be set if A×B > FFh. Setting the OV flag does *not* mean that an error has occurred. Rather, it signals that the number is larger than eight bits, and the programmer needs to inspect register B for the high-order byte of the multiplication operation. The carry flag is always cleared to 0.

The largest possible product is FE01h when both A and B contain FFh. Register A contains 01h and register B contains FEh after multiplication of FFh by FFh. The OV flag is set to 1 to signal that register B contains the high-order byte of the product; the carry flag is 0.

The following table gives examples of MUL multiple-byte arithmetic operations:

Mnemonic	Operation
MOV A,#7Bh	A = 7Bh
MOV 0F0h,#02h	B = 02h
MUL AB	A = 00h and B = F6h; OV Flag = 0
MOV A,#0FEh	A = FEh
MUL AB	A = 14h and B = F4h; OV Flag = 1

CAUTION

Note there is no comma between A and B in the MUL mnemonic.

Division

Division operations use registers A and B as both source and destination addresses for the operation. The unsigned number in register A is divided by the unsigned number in register B, as indicated in the following table:

Mnemonic	**Operation**
DIV AB	Divide A by B; put the integer part of quotient in register A and the integer part of the remainder in B

The OV flag is cleared to 0 unless B holds 00h before the DIV. Then the OV flag is set to 1 to show division by 0. The contents of A and B, when division by 0 is attempted, are undefined. The carry flag is always reset.

Division always results in integer quotients and remainders, as shown in the following example:

$$\frac{A = 213d}{B = 017d} = \begin{matrix}12 \text{ (quotient) and } 9 \text{ (remainder)} \\ 213\ [(12 \times 17) + 9]\end{matrix}$$

When done in hex:

$$\frac{A = 0D5h}{B = 011h} = C \text{ (quotient) and } 9 \text{ (remainder)}$$

The following table lists examples of DIV multiple-byte arithmetic operations:

Mnemonic	**Operation**
MOV A,#0FFh	A = FFh (255d)
MOV 0F0h,#2Ch	B = 2C (44d)
DIV AB	A = 05h and B = 23h [255d = (5 × 44) + 35]
DIV AB	A = 00h and B = 05h [05d = (0 × 35) + 5]
DIV AB	A = 00h and B = 00h [00d = (0 × 5) + 0]
DIV AB	A = ?? and B = ??; OV flag is set to one

CAUTION

The original contents of B (the divisor) are lost.

Note there is no comma between A and B in the DIV mnemonic.

Decimal Arithmetic

Most 8051 applications involve adding intelligence to machines where the hexadecimal numbering system works naturally. There are instances, however, when the application involves interacting with humans, who insist on using the decimal number system. In such cases, it may be more convenient for the programmer to use the decimal number system to represent all numbers in the program.

Four bits are required to represent the decimal numbers from 0 to 9 (0000 to 1001) and the numbers are often called *Binary coded decimal* (BCD) numbers. Two of these BCD numbers can then be packed into a single byte of data.

The 8051 does all arithmetic operations in pure binary. When BCD numbers are being used the result will often be a non-BCD number, as shown in the following example:

$$\begin{array}{rl} 49BCD = & 01001001b \\ \underline{+38BCD} = & \underline{00111000b} \\ 87BCD & 10000001b = 81BCD \end{array}$$

Note that to adjust the answer, an 06d needs to be added to the result.

The opcode that adjusts the result of BCD addition is the decimal adjust A for addition (DA A) command, as shown in the following table:

Mnemonic	Operation
DA A	Adjust the sum of two packed BCD numbers found in A register; leave the adjusted number in A.

The C flag is set to 1 if the adjusted number exceeds 99BCD and set to 0 otherwise. The DA A instruction makes use of the AC flag and the binary sums of the individual binary nibbles to adjust the answer to BCD. The AC flag has no other use to the programmer and no instructions—other than a MOV or a direct bit operation to the PSW—affect the AC flag.

It is important to remember that the DA A instruction assumes the added numbers were in BCD *before* the addition was done. Adding hexadecimal numbers and then using DA A will *not* convert the sum to BCD.

The DA A opcode only works when used with ADD or ADDC opcodes and does not give correct adjustments for SUBB, MUL or DIV operations. The programmer might best consider the ADD or ADDC and DA A as a single instruction and use the pair automatically when doing BCD addition in the 8051.

The following table gives examples of BCD multiple-byte arithmetic operations:

Mnemonic	Operation
MOV A,#42h	A = 42BCD
ADD A,#13h	A = 55h; C = 0
DA A	A = 55h; C = 0
ADD A,#17h	A = 6Ch; C = 0
DA A	A = 72BCD; C = 0
ADDC A,#34h	A = A6h; C = 0
DA A	A = 06BCD; C = 1
ADDC A,#11h	A = 18BCD; C = 0
DA A	A = 18BCD; C = 0

CAUTION

All numbers used must be in BCD form before addition.

Only ADD and ADDC are adjusted to BCD by DA A.

Example Programs

The challenge of the programs presented in this section is writing them using only opcodes that have been covered to this point in the book. Experienced programmers may long for some of the opcodes to be covered in Chapter 6, but as we shall see, programs can be written without them.

EXAMPLE PROBLEM 5.1

Add the unsigned numbers found in internal RAM locations 25h, 26h, and 27h together and put the result in RAM locations 30h (MSB) and 31h (LSB).

▪ **Thoughts on the Problem** The largest number possible is FFh + FFh = 01FEh + FFh = 02FDh, so that two bytes will hold the largest possible number. The MSB will be set to 0 and any carry bit added to it for each byte addition.

To solve this problem, use an ADD instruction for each addition and an ADDC to the MSB for each carry which might be generated. The first ADD will adjust any carry flag which exists before the program starts.

The complete program is shown in the following table:

Mnemonic	Operation
MOV 31h,#00h	Clear the MSB of the result to 0
MOV A,25h	Get the first byte to be added from location 25h
ADD A,26h	Add the second byte found in RAM location 26h
MOV R0,A	Save the sum of the first two bytes in R0
MOV A,#00h	Clear A to 00
ADDC A,31h	Add the carry to the MSB; carry = 0 after this operation
MOV 31h,A	Store MSB
MOV A,R0	Get partial sum back
ADD A,27h	Form final LSB sum
MOV 39h,A	Store LSB
MOV A,#00h	Clear A for MSB addition
ADDC A,31h	Form final MSB
MOV 31h,A	Store final MSB

COMMENT

Notice how awkward it becomes to have to use the A register for all operations. Jump instructions, which will be covered in Chapter 6, require less use of A.

EXAMPLE PROBLEM 5.2

Repeat problem 5.1 using BCD numbers.

■ **Thoughts on the Problem** The numbers in the RAM locations *must* be in BCD before the problem begins. The largest number possible is 99d + 99d = 198d + 99d = 297d, so that up to two carries can be added to the MSB.

The solution to this problem is identical to that for unsigned numbers, except a DA A must be added after each ADD instruction. If more bytes were added so that the MSB could exceed 09d, then a DA A would also be necessary after the ADDC opcodes.

The complete program is shown in the following table:

Mnemonic	Operation
MOV 31h,#00h	Clear the MSB of the result to 0
MOV A,25h	Get the first byte to be added from location 25h
ADD A,26h	Add the second byte found in RAM location 26h
DA A	Adjust the answer to BCD form
MOV R0,A	Save the sum of the first two bytes in R0
MOV A,#00h	Clear A to 00
ADDC A,31h	Add the carry to the MSB; carry = 0 after this operation
MOV 31h,A	Store MSB
MOV A,R0	Get partial sum back
ADD A,27h	Form final LSB sum
DA A	Adjust the final sum to BCD
MOV 30h,A	Store LSB
MOV A,#00h	Clear A for MSB addition
ADDC A,31h	Form final MSB
MOV 31h,A	Store final MSB

COMMENT

When using BCD numbers, DA A can best be thought of as an integral part of the ADD instructions.

EXAMPLE PROBLEM 5.3

Multiply the unsigned number in register R3 by the unsigned number on port 2 and put the result in external RAM locations 10h (MSB) and 11h (LSB).

■ **Thoughts on the Problem** The MUL instruction uses the A and B registers; the problem consists of MOVes to A and B followed by MOVes to the external RAM. The complete program is shown in the following table:

Mnemonic	Operation
MOV A,0A0h	Move the port 2 pin data to A
MOV 0F0h,R3	Move the data in R3 to the B register
MUL AB	Multiply the data; A has the low order result byte
MOV R0,#11h	Set R0 to point to external RAM location 11h
MOV @R0,A	Store the LSB in external RAM
DEC R0	Decrement R0 to point to 10h
MOV A,0F0h	Move B to A
MOV @R0,A	Store the MSB in external RAM

COMMENT

Again we see the bottleneck created by having to use the A register for all external data transfers.

More advanced programs which do signed math operations and multi-byte multiplication and division will have to wait for the development of Jump instructions in Chapter 6.

Summary

The 8051 can perform all four arithmetic operations: addition, subtraction, multiplication, and division. Signed and unsigned numbers may be used in addition and subtraction; an OV flag is provided to signal programmer errors in estimating signed number magnitudes needed and to adjust signed number results. Multiplication and division use unsigned numbers. BCD arithmetic may be done using the DA A and ADD or ADDC instructions.

The following table lists the arithmetic mnemonics:

Mnemonic	Operation
ADD A, source	Add the source byte to A; put the result in A and adjust the C and OV flags
ADDC A, source	Add the source byte and the carry to A; put the result in A and adjust the C and OV flags
DA A	Adjust the binary result of adding two BCD numbers in the A register to BCD and adjust the carry flag
DEC source	Subtract a 1 from the source; roll from 00h to FFh
DIV AB	Divide the byte in A by the byte in B; put the quotient in A and the remainder in B; set the OV flag to 1 if B = 00h before the division

INC source	Add a 1 to the source; roll from FFh or FFFFh to 00h or 0000h
MUL AB	Multiply the bytes in A and B; put the high-order byte of the result in B, the low-order byte in A; set the OV flag to 1 if the result is > FFh
SUBB A, source	Subtract the source byte and the carry from A; put the result in A and adjust the C and OV flags

Problems

Write programs that perform the tasks listed using only opcodes that have been discussed in this and previous chapters. Use comments on each line of code and try to use as few lines as possible. All numbers may be considered to be unsigned numbers.

1. Add the bytes in RAM locations 34h and 35h; put the result in register R5 (LSB) and R6 (MSB).

2. Add the bytes in registers R3 and R4; put the result in RAM location 4Ah (LSB) and 4Bh (MSB).

3. Add the number 84h to RAM locations 17h and 18h.

4. Add the byte in external RAM location 02CDh to internal RAM location 19h; put the result into external RAM location 00C0h (LSB) and 00C1h (MSB).

5–8. Repeat Problems 1–4, assuming the numbers are in BCD format.

9. Subtract the contents of R2 from the number F3h; put the result in external RAM location 028Bh.

10. Subtract the contents of R1 from R0; put the result in R7.

11. Subtract the contents of RAM location 13h from RAM location 2Bh; put the result in RAM location 3Ch.

12. Subtract the contents of TH0 from TH1; put the result in TL0.

13. Increment the contents of RAM location 13h, 14h, and 15h using indirect addressing only.

14. Increment TL1 by 10h.

15. Increment external RAM locations 0100h and 0200h.

16. Add a 1 to every external RAM address from 00h to 06h.

17. Add a 1 to every external RAM address from 0100h to 0106h.

18. Decrement TL0, TH0, TL1, and TH1.

19. Decrement external RAM locations 0123h and 01BDh.

20. Decrement external RAM locations 45h and 46h.

21. Multiply the data in RAM location 22h by the data in RAM location 15h; put the result in RAM locations 19h (low byte), and 1Ah (high byte).

22. Square the contents of R5; put the result in R0 (high byte), and R1 (low byte).

23. Divide the data in RAM location 3Eh by the number 12h; put the quotient in R4 and the remainder in R5.

24. Divide the number in RAM location 15h by the data in RAM location 16h; put the result in external RAM location 7Ch.

25. Divide the data in RAM location 13h by the data in RAM location 14h, then restore the original data in 13h by multiplying the answer by the data in 14h.