ELEMENTARY PROGRAM EXAMPLES
(R. Horvath, Introduction to Microprocessors, Chapter 7)

1 MULTIPLE PRECISION ARITHMETIC

Motorola processors store multi-precision numbers (addresses as well as data) sequentially in memory with the highest ordered byte at the lowest address.

Programs, which add or subtract such multi-precision numbers must start with the least significant parts (LSPs) and work toward the other end. The basic approach is to combine the LSPs with an ADD/SUB instruction to generate the LSP of the result and the carry/borrow out of this part. This single bit is automatically stored in the C-bit (or X-bit) in the condition code register. The more significant parts are then combined together with the carry/borrow from the preceding combination by using the add with carry (or extend) or subtract with carry (or extend) instructions.

1.1 Double-Precision Addition With the MC6809

The program shown in Figure 1 will add two double-precision numbers and store the double-precision sum in memory. The program defines the locations and the values of the two operands and the location for the sum. Lines 1 through 5 ask the assembler to insert the values of the two double-precision operands into memory starting at location 8000 (hex) and to label them as shown. Lines 6 and 7 ask the assembler to set aside two labeled locations into which the program will store the result. Notice that these "data" locations are separated from the program area by the two ORG statements. Line 8 tells the assembler to start the subsequent coding (from line 9 on) in location 1000. Line 9 actually contains the first MC6809 instruction in the program; lines 1-8 contain assembler directives.

The instruction in line 9 loads the lower byte of the first operand into A. Notice that it uses the extended addressing mode with a symbolic operand, which will be evaluated as 8001H. Lines 10 and 11 compute the lower half of the sum and store it. Line 12 loads the higher half of the first operand into A. Notice that lines 11 and 12 must preserve the value in the carry bit in the CCR unchanged in order that the addition process may be chained between the two halves of the operands. The programming aid in Appendix A shows that this is indeed the case. The carry bit is not affected by load or store operations. This is true in all processors which use the carry to support multi-precision arithmetic. Line 13 completes the chaining of the addition between the halves of the operands, and line 14 stores the result.
1. ORG $8000
2. HI1 FCB 5A FIRST OPERAND IS IN 8000 AND 8001
3. LO1 FCB 96 SECOND OPERAND IS IN 8002 AND 8003
4. HI2 FCB 93 SUM WILL BE STORED IN 8004 AND 8005
5. LO2 PCB 8B
6. HISUM FCB 0
7. LOSUM FCB 0
8. ORG $1000
9. DPSUM LDA LO1 START OF PROGRAM
10. ADDA LO2 ADD LOW BYTES
11. STA LOSUM STORE LOW HALF OF SUM
12. LDA HI1
13. ADCA HI2 ADD HIGH BYTES WITH CARRY FROM LOW
14. STA HISUM STORE HIGH HALF OF SUM
15. RTS HALT THE PROGRAM

Try to assemble this program and to execute (or simulate its execution) with whatever system you may have available for running MC6809 programs. Execute the program one instruction at a time and observe the changes which occur in the pertinent registers and memory locations.

2 Moving Tables in Memory, Simple Loops

Each of the programs shown here will move a table of data from its original location in memory to a new location in memory. The starting address of the original table must be stored in the locations labeled ORGTAB in the program. The starting address of the new table must be stored in locations labeled FINTAB in the program. The first location in the original table must contain the length of the table (not counting itself). This is a common way to indicate the size of a table stored in memory.

The programs also demonstrate a commonly used method of implementing loops.

7.2.1 The MC6809 Program to Move a Table

The MC6809 assembly language program moves a table one byte at a time. The number of bytes in the table is specified in the first location of the table. The program picks up each entry in register A, using X for a pointer, and then stores it, using Y for a pointer. It uses register B as a counter to count the bytes as they are moved.

In lines 1 and 2 the starting address vectors are identified for the assembler. An address vector is a memory location, which will be used to store an address that will be used in a program. Including the addresses of the tables by pre-storing them into memory and then referring to the memory location, rather than embedding the addresses in the program, makes the program more flexible.
The first two MC6809 instructions, in lines 4 and 5, establish X and Y as the address pointers and load them with the appropriate starting addresses. Note that each of these instructions uses the extended addressing mode with a double byte of data found in memory, starting at the address named ORGTAB or FINTAB. Thus, before the program can be executed, the starting addresses of the tables must be pre-stored into these locations.

Line 6 loads the counter, B, with the first entry from the original table. This is the value of the number of bytes to be moved. Line 7 moves this value into the new table. Notice that both of these indexed instructions take advantage of the post-inc option to change the values in the pointers after using them. After they are executed, X and Y will contain the addresses of the second entries in the tables.

Line 8 aborts the program if the table is empty (signified by 0 in the first byte). Notice how the use of labels in assembly language eliminates the need to calculate the offset in this branch instruction. Lines 9 through 12 constitute the loop in which the program moves the table one byte at a time. A byte is picked up in line 9 and deposited in line 10 (both with post-inc). It is then counted in line 11, and in line 12 the count is tested to see whether it has reached 0 yet. This is the most common way to loop through a segment of code a predetermined nonzero number of times—that is, by initializing a counter to the number of times the loop is to be traversed, traversing the loop once, decrementing the count, and then branching back to the start of the loop if the count is not yet 0.

Before this program can be run the contents of all memory locations referenced in it must be defined. The locations $0000 through $0003, which are to contain the starting addresses for the tables as well as the tables themselves, must contain known values if the program is to have a predictable effect.
4 BIT PICKING AND BIT PACKING

The program segments discussed in this section illustrate some of the instructions that can be used to move bits around within a word or to alter selected bits. They make use of the logic operations to clear bits (AND with 0), set bits (OR with 1), or complement bits (EOR with 1). They also illustrate the use of the shift instructions and several conditional branches.

Two of the program segments convert hex digits represented by their ASCII codes into unencoded hex. The ASCII codes for the digits 0 through 9 are 30 through 39, those for the digits A through F are 41 through 46. The segments assume that the characters represented are from this set of 0 through F. A simple series of tests could be included in the program to verify this.

The segments are not complete, many of the labels used are not defined, nor has any attempt been made to make the program segments relocatable. Try to expand these examples by including labels and data and then assembling and running them. Then try to rewrite them to make them relocatable.

4.1 MC6809 Examples of Bit Manipulations

The program segment 1 repacks two hex digits represented by their ASCII codes into a single byte containing the two hex digits. The characters are in locations defined elsewhere as MSCH (most significant character) and LSCH (least significant character). The unencoded hex result is written into the location TEMP, which has been defined elsewhere.

1. START  LDA  MSCH  GET MORE SIGNIFICANT DIGIT
2. CMPA  #$40  IS IT IN THE SET A-F?
3. BHI  LTTR  YES. GO TO LTTR
4. NMBR  ANDA  #$00001111  NO. STRIP OFF LEADING HALF
5. ALIGN  LSLA  MOVE INTO UPPER NIBBLE POSITION
6.  LSLA
7.  LSLA
8.  LSLA
9.  BRA  GTLS  GO TO GET LESS SIG DIGIT
10. LTTR  ANDA  #$00001111  STRIP OFF LEADING HALF
11. ADDA  #9  CONVERT 1-6 INTO A-F
12. BRA  ALIGN
13. GTLS  LDB  LSCH  GET LESS SIGNIFICANT DIGIT
14. CMPB  #$40  IS IT IN THE SET A-F?
15. BHI  LTTR2  YES. GO TO LTTR2
16. NMBR2  ANDB  #$00001111  NO. STRIP OFF THE LEADING HALF
17. BRA  COMB  AND COMBINE
18. LTTR2  ANDB  #$00001111  STRIP OFF LEADING HALF
19. ADDB  #9  CONVERT 1-6 INTO A-F
20. COMB  STB  TEMP  COMBINE NIBBLES
21. ORA  TEMP
The "stripping" operation in lines 4, 10, 16, and 18 is called masking. Bits 4 through 7 are said to be masked out. Note that the mask (operand pattern) is entered in binary in order to show the reader exactly which bits are affected. This is good practice. An operand should be shown in the number system which most clearly illustrates its specific use in that situation.

The segments illustrate some other bit manipulations which are possible. The effects are described in the comments. The bit numbers referred to are the standard ones: bits 0 through 7, right to left. The first four operations obtain a Boolean result bit-by-bit in register A. The other two operations illustrate the use of the BIT (bit test) instruction. This instruction does not alter the value in the designated accumulator. It updates the N- and Z-bits in the CCR to reflect what the result would have been if the operand had been ANDed with the content of that accumulator.

```
* 1. START   LDA   PATTERN   Pick Up Initial Pattern
2. TFR      A,B    Copy Into B For Later Use
3. OP1      ANDA  #00101101   Clear Bits 1,4,6,7 Only
             SET BITS
4. OP2      TFR    B,A     Get Pattern Again
             Complement Bits
5. ORA      ^00101101   Set Bits 0,2,3,5 Only
             Complement Bits
6. OP3      TFR    B,A     Get Pattern Again
7. COMA     B,A     Complement All Bits
8. OP4      TFR    B,A     Get Pattern Again
9. EORA     #00101101   Complement Bits 0,2,3,5 Only
             TEST A SINGLE BIT AND BRANCH
10. OP5     TFR     B,A     Get Pattern Again
11. BITA    #00000100   Set Z-Bit Of CCR If Bit 2 Is 0
12. *       ELSE Clear Z. Don't Change A
13. BEQ      THERE    Go To There If Bit 2 Is 0
             TEST SEVERAL BITS AND BRANCH
14. OP6     TFR     B,A     Get Pattern Again
15. BITA    #00101101   Set Z If Bits 0,2,3,5 Are All 0
             ELSE Clear Z. Don't Change A
16. BNE      THERE2   Go To There2 If Any Bit 0,2,3,5 Is Set
```
5 CONVERSION BETWEEN DECIMAL AND BINARY (HEX)

The program segments shown here illustrate methods of converting a single-byte decimal integer (BCD) into a binary integer and vice versa. In the case of conversion from BCD to binary, the tens and units digits are first isolated and then converted by multiplying the tens digit by 10 and adding the units digit to the product.

The conversion from binary to BCD is complicated by the possibility of overflow beyond a single-byte BCD representation. This will occur whenever the hex equivalent of the binary number exceeds 63 (equivalent to 99 in decimal). The routines shown will return FF as the decimal value when overflow occurs. They could easily be modified to return a correct double-precision BCD result when the hex value exceeds 63.

5.1 MC6809 BCD/Binary Conversion Routines

The program segment in Figure 16 converts a single-byte BCD integer assumed to be in a location whose address has been defined elsewhere as DECNUM into binary, and stores the result into a location whose address has been defined as BINNUM.

1. START LDA DECNUM Pick Up Number
2. TENS ANDA #%11110000 Mask Out Units, Get Tens Digit
3. ALIGN LSRA Move Into Lower Position
4. LSRA
5. LSRA
6. LSRA Tens Digit Is Now In A
7. CNVT LDB #10 Weight Of Ten's Position
8. MUL Convert To Binary
9. UNITS STB TEMP Save Converted Tens Digit
10. UNITS LDA DECNUM Pick Up Number Again
11. ANDA #%00001111 Mask Out Tens, Get Units Digit
12. ADDA TEMP Add In Tens Value
13. STA BINNUM Save Result

FIGURE 16 MC6809 to binary program segment.

The routine uses a location named TEMP (defined elsewhere) for temporary storage. The value in DECNUM is assumed to be a valid BCD integer. A simple routine could be devised to check this assumption.

The tens digit is first isolated and shifted into the units position in lines 2 through 6. It is then converted to binary by multiplying it (in binary) by 10 in line 8. The converted value appears in register B (the lower half of the product register D), and is then stored in location TEMP. The units value (which is already in binary—actually BCD) is isolated in line 11 and added to the 10s value in lines 10 through 12.
The program segment of Figure 17 encodes the binary integer found in location BINNUM into BCD and stores the result into location DECNUM. If the decimal value exceeds two digits, overflow is indicated by storing FF in location DECNUM. It, too, uses one temporary memory location named TEMP.

1. START LDA BINUM Pick Up Number
2. TEST CMPA #99 Greater Than 99 (Decimal)?
3. BHI TOOBIG Yes, Indicate Overflow
4. OKAY CLRB No, Start Conversion
5. TENS SUBA #10 Take Out A Ten
6. BLO ALIGN No More Tens, Align The Tens
7. INCB Count The Ten
8. BRA TENS
9. ALIGN LSLB Move Tens Digit Into Position
10. LSLB
11. LSLB
12. LSLB
13. STB TEMP Save Bed Tens Digit
14. UNITS ADDA #10 Restore Units Value
15. ORA TEMP Combine In The Tens Digit
16. STA DECNUM
17. BRA FINIS
18. TOOBIG LDA #$FF Overflow Code
19. STA DECNUM
20. FINIS

The test and branch for overflow take place in lines 2 and 3. If overflow occurs (number greater than 99 decimal) the program branches to TOOBIG (line 18), where the value of FF hex is stored in DECNUM.

The conversion starts in line 4 by clearing register B, which is used to count the number of 10s in the integer. During the loop in lines 5 through 8, successive values of 10 are subtracted out of the original integer and counted. When the subtraction is found to be unsuccessful by the test in line 6, the loop is terminated. At that point, register B contains the number of 10s, which were successfully subtracted out, and register A contains the leftover value minus 10.

In lines 9 through 13 the 10s digit in register B is shifted left to occupy its proper position for the final BCD result and then saved in TEMP. The units value is restored in line 14 and combined with the 10s digit in line 15. The final result is stored away in line 16. The branch in line 17 is to go around the overflow case (TOOBIG) in lines 18 and 19.