Interfacing The Am9511 Arithmetic Processing Unit

Marvin L. De Jong
Department of Mathematics-Physics
The School of the Ozarks
Pt. Lookout, MO 65726

Introduction

If you are interested in a hardware solution to the problem of addition, subtraction, multiplication, division, and functions such as sine, cosine, tangent, square root, exponential, logarithm and their inverse functions, then the Am9511 integrated circuit will be of interest to you. The Am9511 Arithmetic Processing Unit is a product of Advanced Micro Devices Inc., 901 Thompson Place, Sunnyvale, CA 94086. It performs signed multiplication, addition, subtraction and division with either 16-bit integers or 32-bit integers, in twos complement form. It also does these operations and evaluates a variety of functions (mentioned above) in a 32-bit floating point form. In the floating point form, the mantissa of the number is represented by 24 bits (equivalent to approximately seven significant decimal digits). The exponent is represented by six bits and a sign bit, giving a range of numbers that can be represented from roughly 10^{-19} to 10^{+19}. The one bit not accounted for so far is the sign of the mantissa. Thus, the Am9511 should satisfy most of the calculating needs of microcomputer users. It is important to point out that the Am9511 is a *binary* device as opposed to a *BCD* device. If you intend to use it like a calculator, then appropriate BCD-to-binary and binary-to-BCD routines will be needed to input and output numbers.

Timing of the various control pins on the Am9511 is one of the most important considerations in constructing an interface between it at the microprocessor. The timing requirements seem to be more relaxed in the most recent specification sheets, but my original specifications were quite complex. Perhaps it would be easy to interface the Am9511 somewhere in the address space, using address lines and control lines to operate it. However, given the complexities of the original timing diagrams, we used an interface adapter (the 6522, although any of the other popular interface adapters such as the 6530 can also be used with our programs). One port is used for data transfers, while several pins of the other port on the interface adapter is used to control the Am9511. These techniques produce an extremely simple interface at the expense of some overhead in software.

Before proceeding to the details of the circuit and the driver programs it should be pointed out that if you are interested in building and using this or some other circuit that uses the Am9511, you will want to get complete specification sheets, a publication called "Algorithm Details for the Am9511 Arithmetic Processing Unit," and a card-type Am9511 reference card. All three of these publications are available from Advanced Micro Devices. The Am9511 itself costs about $200, a number which may cause you to turn to the next article. A few mail order houses such as Advanced Computer Products are beginning to list the chip in their advertisements. Be sure to request all the literature mentioned above because you will need it to know how to use the chip. Space does not permit us to write a complete description of all the features of the chip.

The Am9511 Interface Circuit

The interface circuit is given in Figure 1. It is very simple because the complexity is absorbed in the software that must accompany this circuit. As noted, any 6502 system such as the SUPERKIM, KIM-1, AIM 65, etc., may be used, and any two-port interface adapter can be used. Be sure to include the 0.01 microfarad bypass capacitors, keep the leads between the Am9511 and the microcomputer short, and tie the unused control inputs ($\overline{\text{EACK}}$ and $\overline{\text{SVACK}}$) to logic one as shown in Figure 1. I will not reveal how many hours of grief the failure to follow these standard procedures cost me. Keep it simple, neat, and don't try any shortcuts. Also follow the usual procedures in handling integrated circuits that are susceptible to damage by static discharge. This is not your typical El Cheapo IC: $200 makes it irreplaceable. Avoid any Benjamin Franklin type experiments.

The Driver Subroutines

Listing 1 gives five subroutines that work with the interface circuit in Figure 1 to operate the Am9511. The subroutines are:

1. **RESET** - A subroutine that is used to reset the Am9511 either after power is applied or to clear the Am9511 to a known condition. This subroutine must be called after power-up and before using the Am9511.
2. **WRITE** - This subroutine transfers a byte of data in the accumulator of the 6502 to the stack of the Am9511.
3. **COMMAND** - A subroutine that transfers an eight-bit command word from the accumulator

of the 6502 to the command register of the Am9511.

4. **READ** - Subroutine READ takes one byte of data (part of the answer) from the stack of the Am9511 and returns it to the X - register in the 6502.
5. **STATUS** - This subroutine reads the status register of the Am9511 and transfers its contents to the X - register in the 6502.

The comments in the various subroutines should be studied in connection with the Am9511 specification sheets to understand the functions of the various instructions. We only note here that each of the access subroutines, WRITE, COMMAND, READ, and STATUS, wait for the Am9511 to signal that an operation is complete when its $\overline{\text{PAUSE}}$ pin returns to logic one.

We will describe a few operations with the Am9511 to illustrate how the subroutines work. Refer to the literature mentioned previously for more details on the stack operation. The Am9511 stack may be regarded either as an eight-level, 16-bit wide stack, or as a four-level, 32-bit wide stack. Writing once to the Am9511 places an 8-bit word on the stack. However, since all of the "words" operated on by the Am9511 are either 16 bits or 32 bits wide, you must write at least 16 data bits (two bytes) to fill a 16-bit stack location. You must write four bytes to fill a 32-bit stack location. The *last* level filled (either 16 bits or 32 bits wide) is called TOS (acronym for top of stack). The level filled *previously* is referred to as NOS (next on stack).

An example will clarify the operation of the stack. Suppose we wish to add two 16-bit integers (they must be in twos complement form). Using the WRITE subroutine, we write the least-significant byte of one of the numbers to the Am9511 stack. Call this byte B1. Next we write B2, the most-significant byte of the same integer, to the Am9511. This puts a 16-bit integer onto TOS, the top level of the stack. The other addend, call it A1 and A2 for the least-significant and most-significant bytes respectively, is placed on the TOS by calling subroutine WRITE two more times. Now number B (B1 and B2) is in NOS and A (A1 and A2) is in TOS. The command code for a 16-bit addition, $6C, is now placed in the 6502 accumulator and subroutine COMMAND is called. The Am9511 adds TOS to NOS and puts the result into TOS. The result R, consisting of the most-significant byte R1 and the least-significant byte R2 of the 16-bit answer, is obtained by calling subroutine READ. The first call of READ retrieves the most-significant byte R2, and the second call of READ retrieves the least-significant byte of the result R. The status register can be read to see if the addition produced a carry or an overflow.

Subtraction follows exactly the same pattern. The minuend M is loaded on the stack, followed by the subtrahend S to obtain the difference D where D = M - S. After M and S are loaded on the stack, the subtraction command ($2D for a 32-bit word) will result in the difference D in TOS. Calling subroutine READ (twice for a 16-bit integer, four times for a 32-bit integer) gives the answer in the order from most-significant byte to least-significant byte. In division, the dividend is loaded on the stack followed by the divisor, and the quotient is read after the operation is completed. Some of you will recognize that the Am9511 uses RPN.

A program to illustrate these 16-bit operations is given in Listing 2. Suppose we wish to subtract $32FC from $FF5B. We would load $5B into location $0004, $FF into location $0003, $FC into location $0002, and $32 would be loaded into location $0001. The 16-bit subtraction command for the Am9511, $6D, would be loaded into location $0000. The program in Listing 2 will call the appropriate subroutines and place the answer in locations $00FF (most-significant byte) and $00FE (least-significant byte). This program can be used to test many of the operations of the Am9511, including sine, cosine, etc., by loading a 32-bit number (fixed or floating-point representation) on the stack, and then placing a command on the stack. It is a nice simple test program, but remember that many of the Am9511 functions require that the argument is in floating point form, so to find the square root of four requires that you convert four to a floating-point number. The Am9511 will do this if you either cannot or will not.

A word about execution time may be useful at this point. Instructions take from 16 clock cycles for a 16-bit integer addition to several thousand clock cycles for functions like sine, cosine, etc. We operated our Am9511 at 1MHz, but it can be operated at 2MHz and other versions go as high as 4MHz. Clearly the subroutines in Listing 1 require a significant amount of overhead for the simple integer operations, but become insignificant in terms of time overhead when the complex functions are called. Perhaps some reader will design an interface where instructions like STA DATA, STA COMMAND, LDA DATA, and LDA STATUS can be used instead of the subroutines. The difficulty is in working out the necessary timing requirements for the READ and WRITE operations of the 6502. The Am9511 timing seems to be more closely related to 8080A systems than either 6502 systems or 6800 systems.

Our final illustrative program is one that was designed to generate a sine table consisting of one cycle of a sine wave residing in one page of memory. The amplitude of the sine wave is $7F00, in other words, we found $7F00*Sin[Y*(Pi/128] where Y is a number that varied from $00 to $FF (0 to 255). This result was converted to a 16-bit fixed point format, and the most-significant byte was stored in a table in page $0E, while the least-significant byte was stored in a table in page $0F. Note that the result will be in twos complement form, so at location $0E80 in the table when we are exactly half-way through the sine wave, you will find $00, but at location $0E81 you will find the first negative value of the sine wave and it is $FC, the one in the most-significant bit of the 16-bit result indicating a minus number.

What do you do with a sine wave table? You could read it out to a D/A converter at various rates and play a tune, or you could add a series of sine waves to make a more complex sound. My purpose was to test the AM9511 and in the future I will use the sine wave table as part of a fast-Fourier transform program (I hope). Instead of synthesizing music I would really like to synthesize $20 bills. Let me know if you succeed.

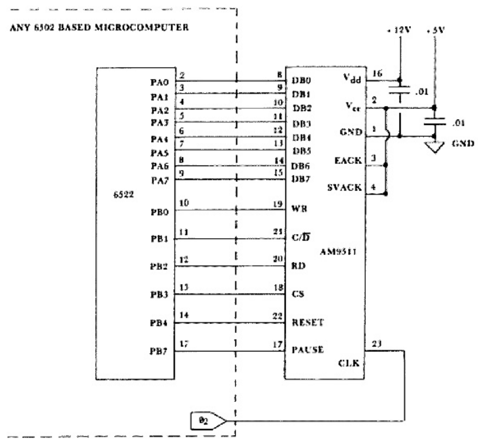

Figure 1.
Interfacing the AM9511 Arithmetic Processing Unit to a 6522 VIA Chip. Other interface adapters that may be used include the 6520, the 6530 and the 6532. No special handshaking pins are used.

Listing 1 Subroutines to drive the AM9511

```
0300 A9 1F     RESET     LDA $1F      Make PB0 - PB4
0302 8D 02 A0            STA PBDD     output pins to con-
                                      trol the AM9511.
0305 A9 0F               LDA $0F      RESET pin to
0307 8D 00 A0            STA PBD      logic zero.
030A A9 1F               LDA $1F      Hold RESET high
030C 8D 00 A0            STA PBD      for at least five
030F EA                  NOP          clock cycles.
0310 EA                  NOP
0311 A9 0F               LDA $0F      Bring RESET pin
0313 8D 00 A0            STA PBD      to logic zero to
                                      run the AM9511.
0316 60                  RTS          Return to the call-
                                      ing program.
              **************
0320 8D 01 A0  WRITE     STA PAD      A contains the
0323 A9 04               LDA $04      byte to be written
0325 8D 00 A0            STA PBD      to the AM9511
                                      (A = accumula-
                                      tor) CS low, C/D
                                      low, WR low.
0328 AD 00 A0  WAIT      LDA PBD      Read PBD to see
                                      if PAUSE pin is at
032B 10 FB               BPL WAIT     logic zero (no data
                                      transfer allowed).
032D A9 FF               LDA $FF      If PAUSE is high,
032F 8D 03 A0            STA PADD     make PAD an
                                      output port to
                                      transfer data to
                                      the AM9511.
0332 EE 00 A0            INC PBD      Bring WR high to
                                      complete data
                                      transfer.
0335 A9 0F               LDA $0F      Next bring CS,
                                      C/D high.
0337 8D 00 A0            STA PBD
033A A9 00               LDA $00      Now make Port A
033C 8D 03 A0            STA PADD     (PAD) an input
                                      port again.
033F 60                  RTS          Return to the
                                      calling program.
              **************
0340 8D 01 A0  COMMAND   STA PAD      A contains the
                                      command for the
                                      AM9511.
0343 A9 06               LDA $06      CS low, C/D
0345 8D 00 A0            STA PBD      high, WR low.
0348 AD 00 A0  LOAF      LDA PBD      Is PAUSE low?
034B 10 FB               BPL LOAF     Yes, then wait
                                      until it goes high.
034D A9 FF               LDA $FF      Make Port A an
034F 8D 03 A0            STA PADD     output port.
0352 EE 00 A0            INC PBD      Bring WR high.
0355 A9 0F               LDA $0F      Bring other con-
0357 8D 00 A0            STA PBD      trol pins high.
035A A9 00               LDA $00      Return Port A to
035C 8D 03 A0            STA PADD     input status.
035F 60                  RTS
0360 A9 01     READ      LDA $01      CS low, C/D low,
0362 8D 00 A0            STA PBD      RD low.
0365 AD 00 A0  LOITER    LDA PBD      Read PBD to see
                                      if PAUSE is low.
0368 10 FB               BPL LOITER   If it is, then wait
036A AE 01
A0                       LDX PAD      until it goes high.
                                      Am9511 output
                                      to X register.
036D A9 0F               LDA $0F      Bring control pins
036F 8D 00 A0            STA PBD      high.
0372 60                  RTS          Return to calling
                                      program with out-
                                      put in X.
              **************
0380 A9 03     STATUS    LDA $03      CS low, C/D
0382 8D 00 A0            STA PBD      high, RD low.
0385 AD 00 A0  DELAY     LDA PBD      Is PAUSE low?
0388 10 FB               BPL DELAY    Yes, then wait
                                      until it goes high.
038A AE 01 A0            LDX PAD      Read status regis-
038D A9 0F               LDA $0F      ter of AM9511
                                      and keep it in the
                                      X register.
038F 8D 00 A0            STA PBD      Bring control pins
                                      high.
0392 60                  RTS          Status is in X
                                      upon return.
```

Listing 2 Program that loads four bytes (32 bits) and a command into the Am9511

```
0400 20 00 03  START  JSR RESET     Reset the AM9511
                                    to start using it.
0403 A2 03            LDX #03       Initialize X to
                                    count four bytes.
0405 B5 01     LOOP   LDA DATA,X    Get byte from the
                                    data table.
0407 20 20 03         JSR WRITE     Write the byte in-
                                    to
                                    the Am9511.
040A CA               DEX           Decrement byte
                                    counter.
040B 10 F8            BPL LOOP      Loop until four
                                    bytes are written.
040D A5 00            LDA CMND      Get command
                                    byte from location
                                    $0000.
040F 20 40 03         JSR           Write command
                      COMMAND       to the AM9511.
0412 20 60 03         JSR READ      Get MSB of 16-
                                    bit answer.
0415 86 FF            STX MSB       Put most-signifi-
                                    cant byte here.
0417 20 60 03         JSR READ      Get LSB of 16-
                                    bit answer.
041A 86 FE            STX LSB       Put least-signifi-
                                    cant byte in
                                    $00FE.
041C 00               BRK           End sample pro-
                                    gram here.
```

Listing 3. Sine table generator.

```
0500 20 00 03  SINE   JSR RESET     Reset the
                                    Am9511.
0503 A9 1A            LDA $1A       Push Pi
0505 20 40 03         JSR           (3.14159...) on
                      COMMAND       TOS by writing
                                    $1A to
                                    Am9511.
0508 A9 80            LDA $80       Load 128 =
050A 20 20 03         JSR WRITE     $0080 on TOS,
050D A9 00            LDA $00       Pi is pushed
050F 20 20 03         JSR WRITE     down to NOS.
0512 A9 1D            LDA $1D       Convert 128 =
0514 20 40 03         JSR           $0080 from
                      COMMAND       fixed point to
                                    to floating
                                    point form.
0517 A9 13            LDA $13       Divide NOS by
0519 20 40 03         JSR           TOS (Pi/128),
                      COMMAND       result onto
                                    TOS.
051C A0 00            LDY $00       Y serves as
                                    counter for 256
                                    points.
051E A9 37     REPEAT LDA $37       Duplicate NOS
0520 20 40 03         JSR           with TOS.
                      COMMAND       Pi/128 is now
                                    in TOS and
                                    NOS.
0523 98               TYA           Duplicate Y in
                                    accumulator.
0524 20 20 03         JSR WRITE     Push down
                                    TOS.
0527 A9 00            LDA $00       stack, Y into
0529 20 20 03         JSR WRITE     TOS.
052C A9 1D            LDA $1D       Change Y into
052E 20 40 03         JSR           floating point
                      COMMAND       form.
0531 A9 12            LDA $12       Multiply to get
0533 20 40 03         JSR           Y*(Pi/128).
                      COMMAND       Result to NOS.
                                    Pop stack up.
0536 A9 02            LDA $02       Take SIN[Y*
0538 20 40 03         JSR           (Pi/128)], result
                      COMMAND       to TOS.
053B A9 00            LDA $00       Push $7F00 on
053D 20 20 03         JSR WRITE     stack.
0540 A9 7F            LDA $7F
0542 20 20 03         JSR WRITE
0545 A9 1D            LDA $1D       Convert $7F00
0547 20 40 03         JSR            = 32512 to
                      COMMAND       floating point
                                    form.
054A A9 12            LDA $12       Find 32512*
054C 20 40 03         JSR           SIN[Y*(Pi/
                      COMMAND       128)], result to
                                    NOS, pop
                                    stack up.
054F A9 1F            LDA $1F       Convert that
0551 20 40 03         JSR           number to
                      COMMAND       fixed point
                                    format.
0554 20 60 03         JSR READ      Get MSB of
0557 8A               TXA           16-bit result in
                                    X register.
0558 99 00 0E         STA MSB,Y     Store it in a
                                    table in page
                                    $0E.
055B 20 60 03         JSR READ      Get LSB of 16-
055E 8A               TXA           bit result.
055F 99 00 0F         STA LSB,Y     Store it in a
                                    table in page
                                    $0F.
0562 C8               INY           Increment Y
                                    counter.
0563 D0 B9            BNE REPEAT    Repeat until
                                    table is filled.
0565 00               BRK           Break to the
                                    monitor.
```

A BCD to Floating-Point Binary Routine

Marvin L. De Jong
Department of Mathematics-Physics
The School of the Ozarks
Pt. Lookout, MO 65726

Introduction

The principal purpose of this article is to provide the reader with a program that converts a BCD number (ASCII representation) with a decimal point and/or an exponent to a floating-point binary number. The floating-point binary number has a mantissa of 32 bits, an exponent byte consisting of a sign bit and seven magnitude bits, and a sign flag (one byte) for the mantissa. Positive and negative numbers whose magnitudes vary from $1.70141183 * 10^{38}$ to $1.46936795 * 10^{-39}$ and zero can be handled by this routine. In subsequent articles I *hope* to provide an output routine and a four-function arithmetic routine. The routine described here could be used in conjunction with the Am9511 Arithmetic Processing Unit [1] to perform a large variety of arithmetic functions.

Floating-Point Notation

Integer arithmetic is relatively simple to do with the 6502. Consult the Bibliography for a number of sources of information on multiple-byte, signed number addition, subtraction, multiplication and division. Scanlon's book, in particular, has some valuable assembly language routines of this sort. However, additional problems arise when the decimal number has a fractional part, such as the "14159" in the number 3.14159. Also, integer arithmetic is not suitable for handling large numbers like $2.3 * 10^{15}$. The solution is to convert decimal numbers to floating-point binary numbers. A binary floating-point number consists of a **mantissa** with an **implied binary point** just to the left of the most-significant non-zero bit and an **exponent** (or characteristic) that contains the information about where the binary point must be moved to represent the number correctly. Readers who are familiar with **scientific notation** will understand this quickly. Scanlon's book has a good section on floating-point notation. We will merely illustrate what a decimal number becomes in floating point binary by referring you to Table 1. The dashed line over a sequence of digits means that they repeat. For examples, 1/3 = $.\overline{33}$ and 1/11 = $.09\overline{090}$ = $.\overline{090}$ while a binary example is 1/1010 = $.0001100\overline{1100}$ = $.000\overline{1100}$.

Table 1. Decimal number to floating-point binary conversions.

NUMBER	BINARY NUMBER	FLOATING POINT NOTATION	MANTISSA	EXPONENT
0	0	0×2^0	0	0
1	1	$.1 \times 2^1$	1	1
2	10	$.1 \times 2^2$	1	10
4	100	$.1 \times 2^3$	1	11
1.5	1.1	$.11 \times 2^1$	11	1
0.75	.11	$.11 \times 2^0$	11	0
0.1	$0.0001100\overline{1100}$	$.\overline{1100} \times 2^{-3}$	$\overline{1100}$	-11
31	11111	$.11111 \times 2^5$	11111	101
32	100000	100000	1	110

A close examination of Table 1 yields some important conclusions. Unless a number is an integer power of two (2^n where n is an integer), the mantissa required to correctly represent the number will require more bits as the numbers increase. Thus, the number 1 can be correctly represented with a one-bit mantissa, but the number 31 requires a five-bit mantissa. A n-bit mantissa can correctly represent a number as large as $2^n - 1$, but no larger. There is another problem associated with numbers like 0.1_{ten} that become *repeating* numbers in binary. It should be clear that no mantissa with a *finite* number of bits can represent 0.1 *exactly.* The fact that computers use a finite number of bits to represent numbers like 0.1 can be illustrated by using BASIC to add 0.1 to a sum and print the answer repeatedly. Starting with a sum of zero, we obtained an answer of 3.6 after 36 times through the loop, but the next answer is 3.69999999 which is clearly incorrect. The error incurred by using a finite number of bits, to represent a number that requires more than that number of bits to correctly represent it, is called *roundoff error.*

How many bits should be used for the mantissa? Clearly it should be an integer number of bytes for ease in programming. Some computers have software packages that use a 24 bit mantissa. The largest number that can be represented by 24 bits is $2^{24} - 1 = 16777215$. This represents about seven decimal digits, giving about six digit accuracy after several calculations. With my salary there is no trouble with six digit accuracy, but many financial calculations require accuracy to the nearest cent, and six digits are frequently not enough. If we choose 32 bits for our mantissa size we get a little more than nine digits (4.3×10^9). This is the mantissa size used in several versions of Microsoft BASIC, and it is the size chosen here. The propagation of round-off errors through the calculations normally gives about eight digit accuracy. It is generally true that the roundoff errors accumulate as the number of calculations to find a specific result increases, but this is a subject beyond the scope of this article.

How big should the **exponent** be? If we choose to represent the binary exponent with one byte then we will have seven bits to represent the exponent (one sign bit and seven magnitude bits). The largest exponent is then +127. If all the bits in the mantissa are ones, then the largest number that can be represented is $(1/2 + 1/4 + 1/8 + 1/16 + ... + 1/2^{32})*2^{127}$, which is approximately $1.70141183*10^{38}$. The smallest exponent is -128. The smallest *positive* number that the mantissa can be is 1/2, thus the smallest positive number that can be represented is 2^{-129} which is approximately $1.46936795*10^{-39}$. Of course, if we chose to use two bytes for the exponent then much larger and smaller exponents could be accommodated, but for most calculations by earth people, a range of 10^{-39} to 10^{38} will do quite nicely. Remember that if you try to enter a number whose absolute value is outside of the range just given (except for zero) you will obtain erroneous results. No overflow or underflow messages are given when entering numbers with this routine.

One more note before turning to the program. The mantissa is said to be *normalized* when it is shifted so that the most-significant bit is one, and the binary point is assumed to be to the left of the most-significant bit. The only exception to this is the number zero which is represented by zeros in both the mantissa and the exponent. Although you are free to assume the binary point is some other place in the mantissa, it is conventional to keep it to the left of the mantissa, as illustrated in Table 1.

The Program To Float A Number

The program in Listing 1, written in the form of a subroutine, together with the other subroutines given in the listings, will accept numbers represented by ASCII from an input device and convert the numbers into their floating point representation. A typical entry might be +12.3456789E+24 or -.123456789E-30. The plus sign is optional since the computer simply disregards it. Up to 12 significant digits may be entered, although the least-significant three will soon be disregarded, leaving approximately 9 decimal digits (32 binary digits). At the completion of the routine, the floating-point representation will be found in locations $0001, $0002, $0003, $0004 (mantissa), $0005 (exponent) and location $0007 contains the sign of the mantissa. The sign byte is $FF if the number is negative, otherwise it is $00. Note that the accumulator (locations $0001-$0004) has **not** been complemented in the case of a minus number. Forming the twos complement may be done, when required, by the arithmetic routines. If a format compatible with the Am9511 Arithmetic Processing Unit is required, simply drop the least-significant byte of the mantissa ($0004), put the sign (set the bit for a minus, clear it for a plus) in bit seven of the exponent ($0005) and shift the sign of the exponent from bit seven to bit six, making sure to keep the rest of the exponent intact. Table 2 gives a summary of the important memory locations.

Table 2. Memory assignments for the BCD to floating-point binary routine.
$0000 = OVFLO; overflow byte for the accumulator when it is shifted left or multiplied by ten.
$0001 = MSB; most-significant byte of the accumulator.
$0002 = NMSB; next-most-significant byte of the accumulator.
$0003 = NLSB; next-least-significant byte of the accumulator.
$0004 = LSB; least-significant byte of the accumulator.
$0005 = BEXP; contains the binary exponent, bit seven is the sign bit.
$0006 = CHAR; used to store the character input from the keyboard.
$0007 = MFLAG; set to $FF when a minus sign is entered.
$0008 = DPFLAG; decimal point flag, set when decimal point is entered.
$000A = ESIGN; set to $FF when a minus sign is entered for the exponent.
$000B = TEMP; temporary storage location.
$000C = EVAL; value of the decimal exponent entered after the "E."
$0017 = DEXP; current value of the decimal exponent.

After clearing all of the memory locations that will be used by routine, the program in Listing 1 jumps to a subroutine at $0F9B. Most users will not want to call this subroutine, since it merely serves to clear the AIM 65 display. Subroutine INPUT, called next, must be supplied by the user. It must get a BCD digit represented in ASCII code from some input device, store it in CHAR at $0006, *and* return to the calling program with the ASCII character in the 6502's accumulator. The necessary subroutines for the AIM 65 are given in Listing 4. They are given in the "K" disassembly format with no comments since they have previously been described by De Jong[2]. Our subroutines input the number on the keyboard and echo the number on the printer and the display.

The algorithm for the conversion routine was obtained from an article by Hashizume[3]. If you are interested in more details regarding floating-point arithmetic routines, please consult his fine article. A flow chart of the routine in Listing 1 is given in Figure 1. The flow chart and the program comments should be sufficient explanation. Basically it works by converting the number, as it is being entered, to binary and multiplying by ten, in binary of course. Later, if and when the exponent is entered, the number is either multiplied or divided by ten, in binary, to get a normalized mantissa and an exponent representing a power of two rather than a power of ten. Each time a multiplication or division by ten occurs the mantissa is renormalized *and* rounded upward if the most-significant discarded bit is one. Each normalization adjusts the binary exponent. When the decimal exponent finally reaches zero no more multiplications or divisions are necessary since $10^0 = 1$. To maintain 32-bit precision, an extra byte, called OVFLO, is used in the accumulator for all *10 and /10 operations.

REFERENCES

1. De Jong, Marvin L., "Interfacing the Am9511 Arithmetic Processing Unit," **COMPUTE II.** (in press).
2. De Jong, Marvin L., "An AIM 65 Notepad," **MICRO,** No. 16, Sept. 1979, p. 11.
3. Hashizume, Burt, "Floating Point Arithmetic," **BYTE,** V 2, No. 11, Nov. 1977, p. 76.

BIBLIOGRAPHY

1. **Programming and Interfacing the 6502, With Experiments,** Marvin L. De Jong, Howard W. Sams & Co., Indianapolis, 1980.
2. **6502 Assembly Language Programming,** Lance A. Leventhal, Osborne/McGraw-Hill, Berkeley, 1979.
3. **6502 Software Design,** Leo J. Scanlon, Howard W. Sams & Co., Indianapolis, 1980.

Listing 1. ASCII to Floating-Point Binary Conversion Program

```
$0E00 D8        START  CLD          Decimal mode not required
0E01 A2 20             LDX $20      Clear all the memory loca-
0E03 A9 00             LDA $00      tions used for storage by
0E05 95 00      CLEAR  STA MEM,X    this routine by loading
                                    them with zeros.
0E07 CA                DEX
0E08 10 FB             BPL CLEAR
0E0A 20 9B 0F          JSR CLDISP   Clears AIM 65 display.
0E0D 20 30 0F          JSR INPUT    Get ASCII representation of
0E10 C9 2B             CMP $2B      BCD digit. Is it a + sign?
0E12 F0 06             BEQ PLUS     Yes, get another character.
0E14 C9 2D             CMP $2D      Is it a minus sign?
0E16 D0 05             BNE NTMNS
0E18 C6 07             DEC MFLAG    Yes, set minus flag to $FF.
0E1A 20 30 0F   PLUS   JSR INPUT    Get the next character.
0E1D C9 2E      NTMNS  CMP $2E      Is character a decimal
                                    point?
0E1F D0 08             BNE DIGIT    No. Perhaps it is a digit.
                                    Yes, check flag.
0E21 A5 08             LDA DPFLAG   Was the decimal point flag
                                    set?
0E23 D0 2C             BNE NORMIZ   Time to normalize the
                                    mantissa.
0E25 E6 08             INC DPFLAG   Set decimal point flag,
0E27 D0 F1             BNE PLUS     and get the next character.
0E29 C9 30      DIGIT  CMP $30      Is the character a digit?
0E2B 90 24             BCC NORMIZ   No, then normalize the
                                    mantissa.
0E2D C9 3A             CMP $3A      Digits have ASCII repre-
0E2F B0 20             BCS NORMIZ   sentations between $30
                                    and $39.
0E31 20 00 0D          JSR TENX     It was a digit, so multiply
0E34 A5 06             LDA CHAR     the accumulator by ten and
0E36 38                SEC          add the new digit. First
0E37 E9 30             SBC $30      strip the ASCII prefix by
                                    subtracting $30.
0E39 18                CLC          Add the new digit to the
0E3A 65 04             ADC LSB      least- significant byte
                                    of the accumulator.
0E3C 85 04             STA LSB      Next, any "carry" will be
0E3E A2 03             LDX $03      added to the other bytes of
                                    the accumulator.
$0E40 A9 00     ADDIG  LDA $00
0E42 75 00             ADC ACC,X    Add carry here.
0E44 95 00             STA ACC,X    And save result.
0E46 CA                DEX
0E47 10 F7             BPL ADDIG    The new digit has been
                                    added.
0E49 A5 08             LDA DPFLAG   Check the decimal point
                                    flag.
0E4B F0 CD             BEQ PLUS     If not set, get another
                                    character.
0E4D C6 17             DEC DEXP     If set, decrement the
0E4F 30 C9             BMI PLUS     exponent, then get another
                                    character.
0E51 20 30 0D   NORMIZ JSR NORM     Normalize the mantissa.
0E54 84 0B             STY TEMP     Save Y. It contained the
0E56 A9 20             LDA $20      number of "left shifts" in
                                    NORM.
0E58 38                SEC          The binary exponent is 32 -
0E59 E5 0B             SBC TEMP     number of left shifts that
0E5B 85 05             STA BEXP     NORM took to make the
                                    most-significant bit one.
0E5D A5 01             LDA MSB      If the MSB of the accumu-
0E5F F0 5A             BEQ FINISH   lator is zero, then the
0E61 A5 06             LDA CHAR     number is zero, and its all
0E63 C9 45             CMP $45      over. Otherwise, check if
                                    the last character was an
                                    "E".
0E65 D0 52             BNE TENPRW   If not, move to TENPRW.
0E67 20 30 0F          JSR INPUT    If so, get another character.
0E6A C9 2B             CMP $2B      Is it a plus?
0E6C F0 06             BEQ PAST     Yes, then get another
                                    character.
0E6E C9 2D             CMP $2D      Perhaps it was a minus?
0E70 D0 05             BNE NUMP     No, then maybe it was a
                                    number.
0E72 C6 0A             DEC ESIGN    Set exponent sign flag.
0E74 20 30 0F   PAST   JSR INPUT    Get another character.
0E77 C9 30      NUMB   CMP $30      Is it a digit?
0E79 90 3E             BCC TENPRW   No, more to TENPRW.
0E7B C9 3A             CMP $3A
0E7D B0 3A             BCS TENPRW
0E7F 38                SEC          It was a digit, so strip
                                    ASCII prefix.
$0E80 E9 30            SBC $30      ASCII prefix is $30.
0E82 85 0B             STA TEMP     Keep the first digit here.
0E84 20 30 0F          JSR INPUT    Get another character.
0E87 C9 30             CMP $30      Is it a digit?
0E89 90 13             BCC HERE     No. Then finish handling
0E8B C9 3A             CMP $3A      the exponent.
0E8D B0 0F             BCS HERE
0E8F 38                SEC          Yes. Decimal exponent is
0E90 E9 30             SBC $30      new digit plus 10 times the
                                    old digit.
0E92 85 0C             STA EVAL     Strip ASCII prefix
                                    from new digit.
0E94 A5 0B             LDA TEMP     Get the old character and
0E96 0A                ASL A        multiply it by ten. First
                                    times two.
0E97 0A                ASL A        Times two again makes
                                    times four.
0E98 18                CLC
0E99 65 0B             ADC TEMP     Added to itself makes times
                                    five.
0E9B 0A                ASL A        Times two again makes
                                    times ten.
0E9C 85 0B             STA TEMP     Store it.
0E9E 18         HERE   CLC          Add the new digit,
0E9F A5 0B             LDA TEMP
0EA1 65 0C             ADC EVAL     to the exponent.
0EA3 85 0C             STA EVAL     Here is the exponent,
0EA5 A5 0A             LDA ESIGN    except for its sign. Was
                                    it a negative?
0EA7 F0 09             BEQ POSTV    No.
0EA9 A5 0C             LDA EVAL     Yes, then form its twos
0EAB 49 FF             EOR $FF      complement by complemen-
0EAD 38                SEC          tation followed by adding
                                    one.
0EAE 69 00             ADC $00
0EB0 85 0C             STA EVAL     Result into exponent value
                                    location.
0EB2 18         POSTV  CLC          Prepare to add exponents.
0EB3 A5 0C             LDA EVAL     Get "E" exponent.
0EB5 65 17             ADC DEXP     Add exponent from input
                                    and norm.
0EB7 85 17             STA DEXP     All exponent work finished.
$0EB9 A5 17     TENPRW LDA DEXP     Get decimal exponent.
0EBB F0 71             BEQ FINISH   If it is zero, routine is
                                    done
0EBD 10 61             BPL MLTPLY   Ir it is plus, go multiply by
                                    ten.
0EBF A2 03      ONCMOR LDX $03      It's minus. Divide by ten.
0EC1 06 04      BACK   ASL LSB      First shift the accumulator
```

```
0EC3 26 03           ROL NLSB    three bits left.
0EC5 26 02           ROL NMSB
0EC7 26 01           ROL MSB
0EC9 26 00           ROL OVFLO
0ECB C6 05           DEC BEXP    Decrease the binary
0ECD CA              DEX         exponent for each left shift.
0ECE D0 F1           BNE BACK
0ED0 A0 20           LDY $20     Number of trial divisions
0ED2 06 04    AGAIN  ASL LSB     of $0A into the accumu-
0ED4 26 03           ROL NLSB    lator giving a $20 = 32
                                 bit quotient.
0ED6 26 02           ROL NMSB
0ED8 26 01           ROL MSB
0EDA 26 00           ROL OVFLO
0EDC 88              DEY
0EDD F0 0E           BEQ OUT     Get out when number of
0EDF A5 00           LDA OVFLO   trial divisions reaches
                                 $20 = 32.
0EE1 38              SEC         Subtract 10 = $0A from
0EE2 E9 0A           SBC $0A     partial divident in OVFLO.
0EE4 30 EC           BMI AGAIN   If result is minus, zero into
                                 quotient
0EE6 85 00           STA OVFLO   Otherwise store result in
0EE8 E6 04           INC LSB     OVFLO, and set bit to one
                                 in quotient.
0EEA 18              CLC
0EEB 90 E5           BCC AGAIN   Try it again.
0EED A5 00    OUT    LDA OVFLO   Check once more to see if
0EEF C9 0A           CMP $0A     quotient should be rounded
                                 upwards.
0EF1 90 15           BCC AHEAD   No.
0EF3 A2 04           LDX $04     Yes. Add one to quotient.
$0EF5 B5 00   REPET  LDA ACC,X   Get each byte of the accu-
0EF7 69 00           ADC $00     mulator and add the carry
0EF9 95 00           STA ACC,X   from the previous addition.
0EFB CA              DEX
0EFC D0 F7           BNE REPET
0EFE 90 08           BCC AHEAD   What if carry from accumu-
0F00 A5 01           LDA MSB     lator occurred? Get most-
0F02 09 80           ORA $80     significant byte and put a 1
                                 in bit seven.
0F04 85 01           STA MSB     Result into high byte,
0F06 E6 05           INC BEXP    and increment the binary
                                 exponent.
0F08 A5 01    AHEAD  LDA MSB     Because of three-bit shift at
0F0A 30 0A           BMI ARND    start of division, a one-bit
0F0C 06 04           ASL LSB     shift (at most) may be re-
0F0E 26 03           ROL NLSB    quired to normalize the
                                 mantissa now.
0F10 26 02           ROL NMSB
0F12 26 01           ROL MSB
0F14 C6 05           DEC BEXP    If so, also decrement binary
                                 exponent.
0F16 A9 00    ARND   LDA $00     Clear overflow byte.
0F18 85 00           STA OVFLO
0F1A E6 17           INC DEXP    For each divide-by-10,
0F1C D0 A1           BNE ONCMOR  increment the decimal ex-
0F1E F0 0E           BEQ FINISH  ponent until it is zero.
                                 Then its all over.
0F20 A9 00    MLTPLY LDA $00     Clear overflow byte.
0F22 85 00           STA OVFLO
0F24 20 00 0D STLPLS JSR TENX    Jump to multiply-by-ten
                                 subroutine.
0F27 20 30 0D        JSR NORM    Then normalize the
                                 mantissa.
0F2A C6 17           DEC DEXP    For each multiply-by-10,
0F2C D0 F6           BNE STLPLS  decrement the decimal ex-
0F2E 60       FINISH RTS         ponent until it's zero. All
                                 finished now.
```

Listing 2. Multiply by Ten Subroutine.

```
$0D00 18     TENX  CLC          Shift accumulator left.
0D01 A2 04         LDX $04      Accumulator contains
0D03 B5 00   BR1   LDA ACC,X    four bytes so X is set to
                                four.
0D05 2A            ROL A        Shift a byte left.
0D06 95 10         STA ACCB,X   Store it in accumula-
                                tor B.
0D08 CA            DEX
0D09 10 F8         BPL BR1      Back to get another
                                byte.
0D0B A2 04         LDX $04      Now shift accumulator B
0D0D 18            CLC          left once again to get
                                "times four."
0D0E 36 10   BR2   ROL ACCB,X   Shift one byte left.
0D10 CA            DEX
0D11 10 FB         BPL BR2      Back to get another byte.
0D13 A2 04         LDX $04      Add accumulator to
0D15 18            CLC          accumulator B to get
                                A + 4*A = 5*A.
0D16 B5 00   BR3   LDA ACC,X
0D18 75 10         ADC ACCB,X
0D1A 95 00         STA ACC,X    Result into accumulator.
0D1C CA            DEX
0D1D 10 F7         BPL BR3
0D1F A2 04         LDX $04      Finally, shift accumula-
0D21 18            CLC          tor left one bit to get
                                2*5*A = 10*A.
0D22 36 00   BR4   ROL ACC,X
0D24 CA            DEX
0D25 10 FB         BPL BR4      Get another byte.
0D27 60            RTS
```

Listing 3. Normalize the Mantissa Subroutine.

```
$0D 30 18      NORM  CLC
0D 31 A5 00    BR6   LDA OVFLO Any bits set in the over-
0D33 F0 0F           BEQ BR5   flow byte? Yes, then
                               rotate right.
0D35 46 00           LSR OVFLO No, then rotate left.
0D37 66 01           ROR MSB
0D39 66 02           ROR NMSB
0D3B 66 03           ROR NLSB
0D3D 66 04           ROR LSB   For each shift right,
0D3F E6 05           INC BEXP  increment binary
                               exponent.
0D41 B8              CLV       Force a jump back.
0D42 50 Ed           BVC BR6
0D44 90 0D     BR5   BCC BR7   Did the last rotate cause
0D46 A2 04           LDX $04   a carry? Yes, then round
0D48 B5 00     BR8   LDA ACC,X the mantissa upward.
0D4A 69 00           ADC $00   Carry is set so one is
                               added
0D4C 95 00           STA ACC,X
0D4E CA              DEX
0D4F 10 F7           BPL BR8
0D51 30 DE           BMI BR6   Check overflow byte
                               once more.
0D53 A0 00     BR7   LDY $00   Y will count number of
                               left shifts.
0D55 A5 01     BR10  LDA MSB   Does most-significant
0D57 30 0D           BMI BR11  byte have a one in bit
                               seven? Yes, get out.
0D59 18              CLC       No. Then shift the
0D5A A2 04           LDX $04   accumulator left one bit.
0D5C 36 00     BR9   ROL ACC,X
0D5E CA              DEX
0D5F D0 FB           BNE BR9
0D61 C8              INY       Keep track of left shifts.
0D62 C0 20           CPY $20   Not more than $20 = 32
                               bits.
0D64 90 EF           BCC BR10
0D66 60        BR11  RTS       That's it.
```

Listing 4. AIM 65 Input/Output Subroutines.

```
$0F30 20 JSR E93C    $0F60 A2 LDX #13     $0F72 8D STA A44C
0F33 20 JSR F000     0F62 8A TXA          0F75 A2 LDX #01
0F36 85 STA 06       0F63 48 PHA          0F77 BD LDA A438,X
0F38 20 JSR 0F72     0F64 BD LDA A438,X   0F7A CA DEX
0F3B 20 JSR 0F60     0F67 09 ORA #80      0F7B 9D STA A438,X
0F3E A5 LDA 06       0F69 20 JSR EF7B     0F7E E8 INX
0F40 60 RTS          0F6C 68 PLA          0F7F E8 INX
                     0F6D AA TAX          0F80 E0 CPX #15
$0F85 A2 LDX #12     0F6E CA DEX          0F82 90 BCC 0F77
0F87 BD LDA A438,X   0F6F 10 BPL 0F62     0F84 60 RTS
0F8A E8 INX          0F71 60 RTS
0F8B 9D STA A438,X
0F8E CA DEX          $0F9B A2 LDX #13
0F8F CA DEX          0F9D A9 LDA #20
0F90 10 BPL 0F87     0F9F 9D STA A438,X
0F92 A9 LDA #20      0FA2 CA DEX
0F94 8D STA A438     0FA3 10 BPL 0F9F
0F97 20 JST 0F60     0FA5 60 RTS
0F9A 60 RTS
```

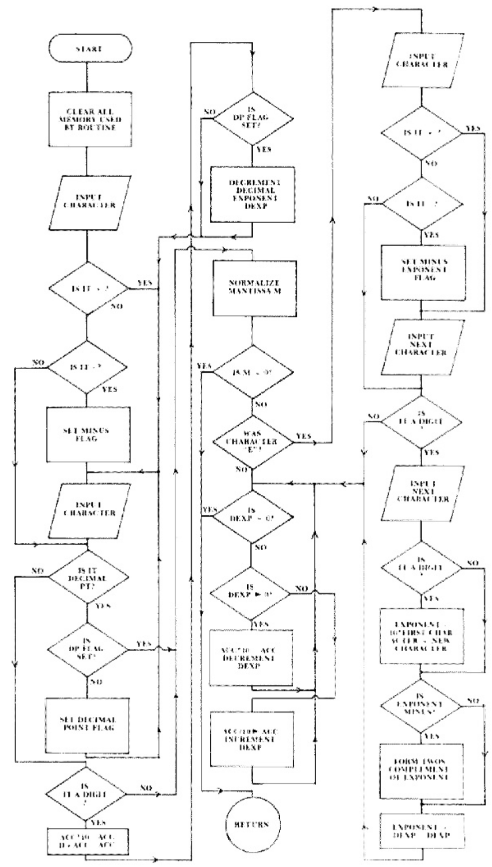

Figure 1. A Flow Chart for the BCD to Floating-Point Binary Routine.

©

A Floating-Point Binary To BCD Routine

Marvin L. DeJong
Department of Mathematics-Physics
The School of the Ozarks
Pt. Lookout, MO 65726

Introduction

A previous issue of **COMPUTE!** carried a BCD to Floating-Point Binary Routine that can be used to convert a series of decimal digits and a decimal exponent to a binary number in a floating-point format. The purpose of such a routine is to enable the user to perform floating-point arithmetic. The program described in this article performs the reverse operation; that is, it converts a floating-point binary number to a decimal number and a decimal exponent. With these two routines and an Am9511 Arithmetic Processing Unit one can do most of the functions found on scientific calculators. I hope to provide a few simple arithmetic routines in the near future. In the meanwhile, you can amuse yourself by converting numbers to floating-point binary numbers and then back to decimal numbers.

Hindsight

The BCD to floating-point binary routine described previously used a divide-by-ten routine that was part of the main program. With my excellent hindsight I now realize that the divide-by-ten routine should have been written as a *subroutine,* to

Listing 1. A New Divide-by-Ten Routine.

```
$0EBF 20 C5 0E  ONCMOR  JSR DIVTEN   Jump to divide-by-ten subroutine.
 0EC2 B8                CLV          Force a jump around the routine.
 0EC3 50 51             BVC ARND     The new subroutine is inserted
 0EC5 A9 00     DIVTEN  LDA $00      here. Clear accumulator for use
 0EC7 A0 28             LDY $28      as a register. Do $28 = 40 bit
 0EC9 06 00     BRA     ASL OVFLO    divide. OVFLO will be used as
 0ECB 26 04             ROL LSB      "guard" byte.
 0ECD 26 03             ROL NLSB     Roll one bit at a time into the
 0ECF 26 02             ROL NMSB     accumulator which serves to hold
 0ED1 26 01             ROL MSB      the partial dividend.
 0ED3 2A                ROL A        Check to see if A is larger than
 0ED4 C9 0A             CMP $0A      the divisor, $0A = 10.
 0ED6 90 05             BCC BRB      No. Decrease the bit counter.
 0ED8 38                SEC          Yes. Subtract divisor from A.
 0ED9 E9 0A             SBC $0A
 0EDB E6 00             INC OVFLO    Set a bit in the quotient.
 0EDD 88        BRB     DEY          Decrease the bit counter.
 0EDE D0 E9             BNE BRA
 0EE0 C6 05     BRC     DEC BEXP     Division is finished, now normalize.
 0EE2 06 00             ASL OVFLO    For each shift left, decrease the
 0EE4 26 04             ROL LSB      binary exponent.
 0EE6 26 03             ROL NLSB     Rotate the mantissa left until a
 0EE8 26 02             ROL NMSB     one is in the most-significant bit.
 0EEA 26 01             ROL MSB
 0EEC 10 F2             BPL BRC
 0EEE A5 00             LDA OVFLO    If the most-significant bit in the
 0EF0 10 12             BPL BRE      guard byte is one, round up.
 0EF2 38                SEC          Add one.
 0EF3 A2 04             LDX $04      X is byte counter.
 0EF5 B5 00     BRD     LDA ACC,X    Get the LSB.
 0EF7 69 00             ADC $00      Add the carry.
 0EF9 95 00             STA ACC,X    Result into mantissa.
 0EFB CA                DEX
 0EFC D0 F7             BNE BRD      Back to complete addition.
 0EFE 90 04             BCC BRE      No carry from MSB so finish.
 0F00 66 01             ROR MSB      A carry, put in bit seven,
 0F02 E6 05             INC BEXP     and increase the binary exponent.
 0F04 A9 00     BRE     LDA $00      Clear the OVFLO position, then
 0F06 85 00             STA OVFLO    get out.
 0F08 60                RTS
   .                     .
   .                     .           Empty memory locations here.
   .                     .
 0F16 A9 00     ARND    LDA $00      Remainder of BCD-to-floating
   .                     .
   :                     :           point routine is here.
```

Listing 2. Modifications to the BCD-to-Floating-Point Binary Routine.

```
$0E54 18                CLC          Clear carry for addition.
 0E55 A5 05             LDA BEXP     Get binary exponent.
 0E57 69 20             ADC $20      Add $20 = 32 to place binary
 0E59 85 05             STA BEXP     point properly.
 0E5A EA                NOP
 0E5B EA                NOP
$0D53 A0 20     BR7     LDY $20      Y will limit the number of
 0D55 A5 01     BR10    LDA MSB      left shifts to 32.
 0D57 30 0D             BMI BR11     If mantissa has a one in its
 0D59 18                CLC          most-significant bit, get out.
 0D5A A2 04             LDX $04
 0D5C 36 00     BR9     ROL ACC,X    Shift accumulator left one bit.
 0D5E CA                DEX
 0D5F D0 FB             BNE BR9
 0D61 C6 05             DEC BEXP     Decrement binary exponent for each
 0D63 88                DEY          left shift.
 0D64 D0 EF             BNE BR10     No more than $20 = 32 bits shifted.
 0D66 60        BR11    RTS          That's it.
```

be called by *both* the BCD to floating-point binary routine and the binary to decimal routine described here. So my first task was to rewrite the divide-by-ten routine as a subroutine. I also discovered that the divide-by-ten routine described in the previous article did not give sufficient precision. In any case, the divide-by-ten routine was completely revised and appears in Listing 1 in this article. It uses the location $0000, called OVFLO, as a "guard" byte to give the necessary precision. It actually starts at $0EC5, but our listing starts at $0EBF to indiciate a few changes that must be made in the original listing to insert the subroutine.

Some other minor modifications to the program are given in Listing 2. Although the BCD to Floating-Point Binary program will work without these changes, it will work better if you introduce the changes shown in Listing 2. The development of the program described in this article enabled me to find some places to improve the other routine. The modifications are simple and short.

The Conversion Routine

The program to convert a normalized floating-point binary number and its exponent to a BCD number and then output the result is given in Listing 3. A 32-bit binary to BCD conversion subroutine is called by this program and it is found in Listing 5. A flowchart of the entire process is given in Figure 1. The normalized floating-point binary mantissa is operated on by a series of "times ten" or "divide by ten" operations until the binary point is moved from the left of the mantissa to the right of the 32 bit mantissa. In other words, we multiply by ten or divide by ten until the binary exponent is 32. Then the mantissa represents an integer and can be converted to a BCD number using the subroutine in Listing 5. The algorithm for this latter routine is from Peatman's (John B)

Listing 3. A Floating-Point Binary to BCD Routine.

```
$0B00 A5 01     BEGIN    LDA MSB       Test MSB to see if mantissa is zero.
 0B02 D0 0E              BNE BRT       If it is, print a zero and then
 0B04 20 9B 0F           JSR CLDISP    get out. Clear display.
 0B07 A9 30              LDA $30       Get ASCII zero.
 0B09 20 A6 0F           JSR OUTCH     Jump to output subroutine.
 0B0C A9 0D              LDA $0D       Get "carriage return."
 0B0E 20 A6 0F           JSR OUTCH     Output it.
 0B11 60                 RTS           Return to calling routine.
 0B12 A9 00     BRT      LDA $00       Clear OVFLO location.
 0B14 85 00              STA OVFLO
 0B16 A5 05     BRY      LDA BEXP      Is the binary exponent negative?
 0B18 10 0B              BPL BRZ       No.
 0B1A 20 00 0D           JSR TENX      Yes. Multiply by ten until the
 0B1D 20 30 0D           JSR NORM      exponent is not negative.
 0B20 C6 17              DEC DEXP      Decrement decimal exponent.
 0B22 B8                 CLV           Force a jump.
 0B23 50 F1     BVC BRY  Repeat.
 0B25 A5 05     BRZ      LDA BEXP      Compare the binary exponent to
 0B27 C9 20              CMP $20       $20 = 32.
 0B29 F0 48              BEQ BCD       Equal. Convert binary to BCD.
 0B2B 90 08              BCC BRX       Less than.
 0B2D 20 C5 0E           JSR DIVTEN    Greater than. Divide by ten until
 0B30 E6 17              INC DEXP      BEXP is less than 32.
 0B32 B8                 CLV           Force a jump.
 0B33 50 F0              BVC BRZ
 0B35 A9 00              LDA $00       Clear OVFLO
 0B37 85 00              STA OVFLO
 0B39 20 00 0D  BRW      JSR TENX      Multiply by ten.
 0B3C 20 30 0D           JSR NORM      Then normalize.
 0B3F C6 17              DEC DEXP      Decrement decimal exponent.
 0B41 A5 05              LDA BEXP      Test binary exponent.
 0B43 C9 20              CMP $20       Is it 32?
 0B45 F0 2C              BEQ BCD       Yes.
 0B47 90 F0              BCC BRW       It's less than 32 so multiply by 10.
 0B49 20 C5 0E           JSR DIVTEN    It's greater than 32 so divide.
 0B4C E6 17              INC DEXP      Increment decimal exponent.
 0B4E A5 05     BRU      LDA BEXP      Test binary exponent.
 0B50 C9 20              CMP $20       Compare with 32.
 0B52 F0 0F              BEQ BRV       Shift mantissa right until exponent
 0B54 46 01              LSR MSB       is 32.
 0B56 66 02              ROR NMSB
 0B58 66 03              ROR NLSB
 0B5A 66 04              ROR LSB
 0B5C 66 0B              ROR TEMP      Least-significant bit into TEMP.
 0B5E E6 05              INC BEXP      Increment exponent for each shift
 0B60 B8                 CLV           right.
 0B61 50 EB              BVC BRU
 0B63 A5 0B     BRV      LDA TEMP      Test to see if we need to round
 0B65 10 0C              BPL BCD       up. No.
 0B67 38                 SEC           Yes. Add one to mantissa.
 0B68 A2 04              LDX $04
 0B6A B5 00     BRS      LDA ACC,X
 0B6C 69 00              ADC $00
 0B6E 95 00              STA ACC,X
 0B70 CA                 DEX
 0B71 D0 F7              BNE BRS
 0B73 20 67 0D  BCD      JSR CONVD     Jump to 32 bit binary-to-BCD
                                       routine.
 0B76 A0 04     BRM      LDY $04       Rotate BCD accumulator right until
 0B78 A2 04     BRP      LDX $04       non-significant zeros are shifted
 0B7A 18                 CLC           out or DEXP is zero, whichever
 0B7B 76 20     BRQ      ROR BCDN,X    comes first.
 0B7D CA                 DEX
 0B7E 10 FB              BPL BRQ
 0B80 88                 DEY
 0B81 D0 F5              BNE BRP
 0B83 E6 17              INC DEXP      Increment exponent for each shift
 0B85 F0 06              BEQ BRO       right. Get out when DEXP = 0.
```

Microprocessor Based Design (McGraw-Hill).

Of course, each time the binary number is multiplied by ten or divided by ten the decimal exponent is adjusted. Thus, we are left with a BCD number in locations $0020 - $0024 (five locations for ten digits) and a decimal exponent in $0017. The rest of the routine is largely processing required to give a reasonable output format. Since we don't want to print a group of non-significant zeros, the BCD number is rotated right until all the zeros are shifted out *or* the decimal exponent is zero, whichever comes first.

Next the routine starts examining the BCD number from the left and skips any leading zeros. Thus, the first non-zero digit is the first digit printed. Of course, if the number is minus (a non-zero result in location $0007) a minus sign is printed. Next the decimal point is printed, and finally the exponent is printed in the form "E XX." Thus, the format chosen always has the decimal point to the right of the significant digits, 3148159. E-6 for example. If you want scientific notation for non-integer results you can modify the output routine. It's simply a matter of moving the decimal point. The flowchart and the comments should allow you to understand and modify the code.

```
0B87 A5 20              LDA LBCDN     Has a non-zero digit been shifted
0B89 29 0F              AND $0F       into the least-significant place?
0B8B F0 E9              BEQ BRM       No. Shift another digit.
0B8D EA         BRO     NOP           Oops. These NOPs cover an
0B8E EA                 NOP           earlier mistake.
0B8F EA                 NOP
0B90 EA                 NOP
0B91 EA                 NOP
0B92 20 9B 0F           JSR CLDISP    This routine simply clears the
0B95 A5 07              LDA MFLAG     AIM 65 20-character display.
0B97 F0 05              BEQ BRN       If the sign of the number is minus,
0B99 A9 2D              LDA $2D       output a minus sign first.
0B9B 20 A6 0F           JSR OUTCH     ASCII " - " = $2D. Output
                                      character.
0B9E A9 0B      BRN     LDA $0B       Set digit counter to eleven.
0BA0 85 0B              STA TEMP
0BA2 A0 04      BRI     LDY $04       Rotate BCD accumulator left to
0BA4 18         BRH     CLC           output most-significant digits
0BA5 A2 FB              LDX $FB       first. But first bypass zeros.
0BA7 36 25      BRG     ROL BCDN
0BA9 E8                 INX
0BAA D0 FB              BNE BRG
0BAC 26 00              ROL OVFLO     Rotate digit into OVFLO.
0BAE 88                  DEY
0BAF D0 F3              BNE BRH
0BB1 C6 0B              DEC TEMP      Decrement digit counter.
0BB3 A5 00              LDA OVFLO     Is the rotated digit zero?
0BB5 F0 Eb              BEQ BRI       Yes. Rotate again.
0BB7 18         BRX     CLC           Convert digit to ASCII and
0BB8 69 30              ADC $30       output it.
0BBA 20 A6 0F           JSR OUTCH
0BBD A9 00              LDA $00       Clear OVFLO for next digit.
0BBF 85 00              STA OVFLO
0BC1 A0 04              LDY $04       Output the remaining digits.
0BC3 18         BRL     CLC
0BC4 A2         $FB     LDX $FB
0BC6 36 25      BRJ     ROL BCDN,X    Rotate a digit at a time into
0BC8 E8                 INX           OVFLO, then output it. One digit
0BC9 D0 FB              BNE BRJ       is four bits or one nibble.
0BCB 26 00              ROL OVFLO
0BCD 88                 DEY
0BCE D0 F3              BNE BRL
0BD0 A5 00              LDA OVFLO     Get digit.
0BD2 C6 0B              DEC TEMP      Decrement digit counter.
0BD4 D0 E1              BNE BRX
0BD6 A5 17              LDA DEXP      Is the decimal exponent zero?
0BD8 F0 48              BEQ ARND      Yes. No need to output exponent.
0BDA A9 2E              LDA $2E       Get ASCII decimal point.
0BDC 20 A6 0F           JSR OUTCH     Output it.
0BDF A9 45              LDA $45       Get ASCII "E".
0BE1 20 A6 0F           JSR OUTCH
0BE4 A5 17              LDA DEXP      Is the decimal exponent plus?
0BE6 10 0D              BPL THERE     Yes.
0BE8 A9 2D              LDA $2D       No. Output ASCII " - "
0BEA 20 A6 0F           JSR OUTCH
0BED A5 17              LDA DEXP      It's minus, so complement it and
0BEF 49 FF              EOR $FF       add one to form the twos
                                      complement.
0BF1 85 17              STA DEXP
0BF3 E6 17              INC DEXP
0BF5 A9 00      THERE   LDA $00       Clear OVFLO.
0BF7 85 00              STA OVFLO
0BF9 F8                 SED           Convert exponent to BCD.
0BFA A0 08              LDY $08
0BFC 26 17      BR1     ROL DEXP
0BFE A5 00              LDA OVFLO
$0C00 65 00             ADC OVFLO
0C02 85 00              STA OVFLO
0C04 88                 DEY
0C05 D0 F5              BNE BR1
```

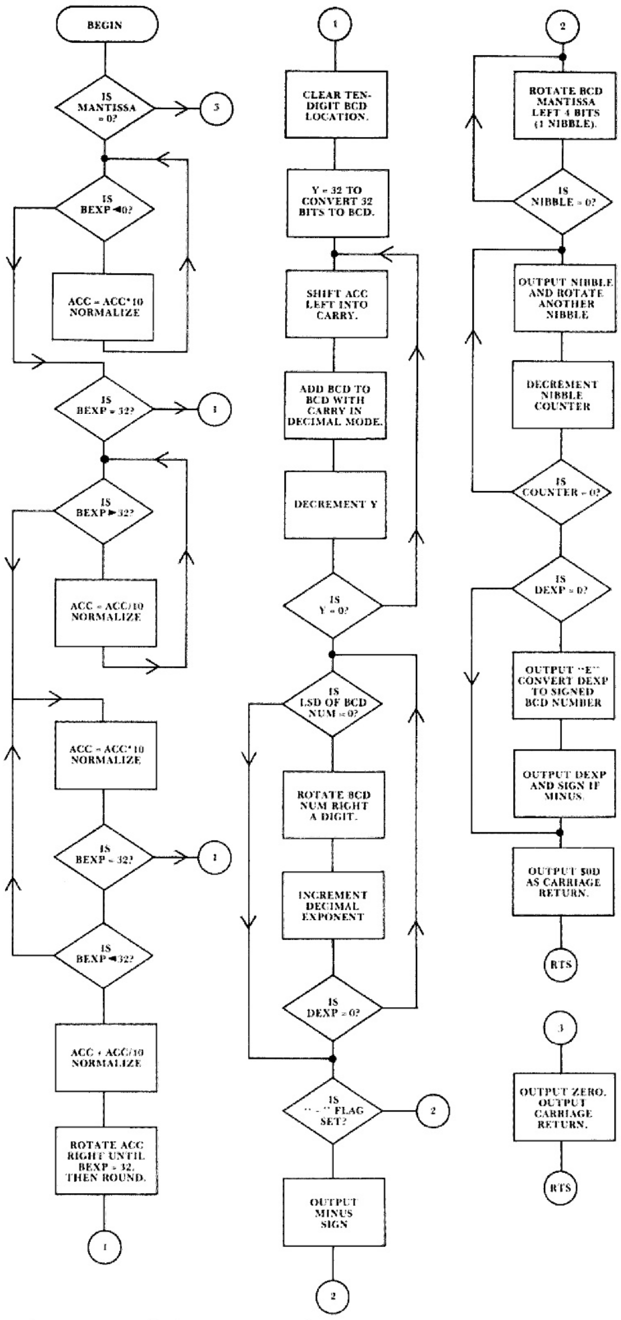

Figure 1. Flowchart of the Floating-Point Binary to BCD Routine.

```
0C07 D8                CLD
0C08 18                CLC
0C09 A5 00             LDA OVFLO    Get BCD exponent.
0C0B 29 F0             AND $F0      Mask low-order nibble (digit).
0C0D F0 09             BEQ BR2
0C0F 6A                ROR A        Rotate nibble to the right.
0C10 6A                ROR A
0C11 6A                ROR A
0C12 6A                ROR A
0C13 69 30             ADC $30      Convert to ASCII.
0C15 20 A6 0F          JSR OUTCH    Output the most-significant digit.
0C18 A5 00     BR2     LDA OVFLO    Get the least-significant digit.
0C1A 29 0F             AND $0F      Mask the high nibble.
0C1C 18                CLC
0C1D 69 30             ADC $30      Convert to ASCII.
0C1F 20 A6 0F          JSR OUTCH
0C22 A9 0D     ARND    LDA $0D      Get an ASCII carriage return.
0C24 20 A6 0F          JSR OUTCH
0C27 60                RTS          All finished.
```

Listing 4. Subroutine OUTCH For the AIM 65.

```
$0FA6 20 00 F0  OUTCH  JSR PRINT    AIM 65 monitor subroutine.
 0FA9 20 72 0F         JSR MODIFY   See previous article in COMPUTE!
 0FAC 20 60 0F         JSR DISPLAY  See previous article in COMPUTE!
 0FAF 60 RTS           RTS
```

Listing 5. A 32 Bit Binary-to-BCD Subroutine.

```
$0D67 A2 05     CONVD  LDX $05       Clear BCD accumulator.
 0D69 A9 00            LDA $00
 0D6B 95 20     BRM    STA BCDA,X    Zeros into BCD accumulator.
 0D6D CA               DEX
 0D6E 10 FB            BPL BRM
 0D70 F8               SED           Decimal mode for add.
 0D71 A0 20            LDY $20       Y has number of bits to be
 0D73 06 04     BRN    ASL LSB       converted. Rotate binary number
 0D75 26 03            ROL NLSB      into carry.
 0D77 26 02            ROL NMSB
 0D79 26 01             ROL MSB
 D7B A2 FB             LDX $FB       X will control a five byte
 0D7D B5 25     BRO    LDA BCDA,X    addition. Get least-significant
 0D7F 75 25            ADC BCDA,X    byte of the BCD accumulator,
 0D81 95 25            STA BCDA,X    add it to itself, then store.
 0D83 E8               INX           Repeat until all five bytes have
 0D84 D0 F7            BNE BRO       been added.
 0D86 88               DEY           Get another bit from the binary
 0D87 D0 EA            BNE BRN       number.
 0D89 D8               CLD           Back to binary mode.
 0D8A 60               RTS           And back to the program.
```

©

A Floating Point Addition And Subtraction Routine

Marvin L. De Jong
The School of the Ozarks
Pt. Lookout, MO

I. Introduction

In previous articles in **COMPUTE!** we have described:

1) A program to convert a decimal number from the keyboard into a floating-point binary number,

Except for a few JSR and JMP instructions, the routine is relocatable. It would not be difficult to put all of these routines in PROM.

2) A program to convert a floating-point binary number to a decimal number and output the number.

3) A program to multiply two signed floating-point binary numbers,

4) A program to divide two signed floating-point binary numbers.

In this article we give a program that adds or subtracts two signed floating-point binary numbers. The programs complete a four-function package.

II. The Subtraction And Addition Routines

As before, three accumulators are used. The contents of accumulator A (ACCA in the program) are subtracted from the number in accumulator B (ACCB), and the result is stored in the result (RES) accumulator. Finally, the answer is moved back to a modified accumulator A that can be used by the output (floating-point binary to BCD routine) program. In the case of the addition program, the numbers in the two accumulators, A and B, are added rather than subtracted.

Accumulator A occupies locations $0000 through $0003 with a guard byte at $0004. The byte at $0000 is the most-significant byte. Accumulator B occupies locations $0020 through $0023 with a guard byte at $0024. The result accumulator is at $0010 to $0014. When the calculation is finished the answer is moved to the accumulator used by the floating-point binary to BCD routine to output the answer. Our accumulator architecture is identical in the four arithmetic function programs.

Here is the algorithm. It makes use of the fact that subtraction can be accomplished by changing the sign of the subtrahend and then adding. From algebra we know

$$a-b=a+(-b).$$

1. Entry point for subtraction. To subtract, complement the sign byte (ACCS) of A, then add.
2. Entry point for addition. Rotate smaller number right until exponents are the same (ACCX = BCCX).
3. Are the signs the same? Yes, go to 4. No, go to 8.
4. Sign of result = sign of addends.
5. Add the numbers.
6. If there is a carry, rotate right one place and increment exponent.
7. Go to round routine (part of multiplication listing).
8. Form the twos complement of the negative number.
9. Add the numbers.
10. If carry results, then the answer is +. Go to 7.
11. If no carry results, then the answer is –. Form the twos complement of the result. Go to 7.

These add and subtract routines use the same round instructions that the multiplication routine used, starting at DETOUR ($0C7D), and those instructions are not repeated here. Thus, you will find a JMP DETOUR instruction near the end of the routine. Except for a few JSR and JMP instructions, the routine is relocatable. It would not be difficult to put all of these routines in PROM. A driver program to test the routines is given in Listing 2.

Listing 2. An Input/Output/Add (or Subtract) Calling Program.

$0050 20 00 0E	JSR INPUT	Call the BCD to Floating-Point Binary Routine.
$0053 30 B0 0F	JSR SUB1	Call the subroutine to modify the accumulator.
$0056 20 C0 0F	JSR SUB2	Transfer ACCA to ACCB.
$0059 20 00 0E	JSR INPUT	Get the second number.
$005C 20 B0 0F	JSR SUB1	Fix the accumulator again.
$005F 20 00 09*	JSR SUB	Subtract the second number from the first.
$0062 20 00 0B	JSR OUTPUT	Output the result using the Floating-Point Binary to BCD Routine.
$0065 00	BRK	

*Change to 20 06 09 for addition.

```
SOURCE FILE: SUBADD
0C7D:          DETOUR  EQU  $0C7D
0027:          BCCS    EQU  $0027
0005:          ACCX    EQU  $0005
0007:          ACCS    EQU  $0007
0020:          ACCB    EQU  $0020
0025:          BCCX    EQU  $0025
0010:          RES     EQU  $0010
0000:          ACCA    EQU  $0000
----- NEXT OBJECT FILE NAME IS SUBADD.OBJ0
0900:                  ORG  $0900
0900:A5 07     SUB     LDA  ACCS       ;ENTRY POINT FOR SUBTRACTION
0902:49 FF             EOR  #$FF
0904:85 07             STA  ACCS
0906:A5 05     ADD     LDA  ACCX       ;ENTRY POINT FOR ADDITION
0908:C5 25             CMP  BCCX       ;COMPARE EXPONENTS
090A:F0 54             BEQ  OPRAT
090C:30 2A             BMI  ADJA
090E:A2 FB             LDX  #$FB
0910:A0 05             LDY  #05        ;CHECK FOR ZERO MANTISSA
0912:B5 25     BR1     LDA  ACCB+5,X
0914:D0 06             BNE  ROTB
0916:88                DEY
0917:F0 10             BEQ  ZEROB
0919:E8                INX
091A:D0 F6             BNE  BR1
091C:A2 FB     ROTB    LDX  #$FB       ;ROTATE MANTISSA RIGHT
091E:18                CLC              ;AND INCREMENT EXPONENT
091F:76 25     BR2     ROR  ACCB+5,X
0921:E8                INX
0922:D0 FB             BNE  BR2
0924:E6 25             INC  BCCX
0926:18                CLC
0927:90 DD             BCC  ADD
0929:A0 08     ZEROB   LDY  #08
092B:A0 08             LDY  #08        ;MY MISTAKE.  WHO NEEDS TWO LDY'S?
092D:A2 FB     UP      LDX  #$FB       ;MIGHT CATCH A COPYRIGHT VIOLATOR?
092F:76 05     HERE    ROR  ACCA+5,X
0931:E8                INX
0932:D0 FB             BNE  HERE
0934:88                DEY
0935:D0 F6             BNE  UP
0937:60                RTS
0938:A2 FB     ADJA    LDX  #$FB       ;CHECK FOR ZERO MANTISSA AGAIN
093A:A0 05             LDY  #05
093C:B5 05     BR3     LDA  ACCA+5,X
093E:D0 06             BNE  ROTA
0940:88                DEY
0941:F0 0F             BEQ  ZEROA
0943:E8                INX
0944:D0 F6             BNE  BR3
0946:A2 FB     ROTA    LDX  #$FB       ;ROTATE MANTISSA RIGHT
0948:18                CLC              ;AND INCREMENT EXPONENT
0949:76 05     BR4     ROR  ACCA+5,X
094B:E8                INX
094C:D0 FB             BNE  BR4
094E:E6 05             INC  ACCX
0950:90 B4             BCC  ADD
0952:A5 25     ZEROA   LDA  BCCX       ;ADDEND IS ZERO
0954:85 05             STA  ACCX
0956:A2 03             LDX  #03
0958:B5 20     BACK    LDA  ACCB,X
095A:95 01             STA  ACCA+1,X
```

```
095A:95 01              STA  ACCA+1,X
095C:CA                 DEX
095D:10 F9              BPL  BACK
095F:60                 RTS
0960:A5 07      OPRAT   LDA  ACCS       ;CHECK THE SIGNS OF THE ADDENDS
0962:C5 27              CMP  BCCS
0964:D0 19              BNE  OPPOS
0966:20 DC 09           JSR  ADDNUM     ;ADD NUMBERS OF LIKE SIGN
0969:90 11              BCC  BR8
096B:A5 05              LDA  ACCX
096D:69 00              ADC  #00
096F:85 05              STA  ACCX
0971:50 01              BVC  BR6
0973:00                 BRK
0974:A2 FB      BR6     LDX  #$FB
0976:38                 SEC
0977:76 15      BR7     ROR  RES+5,X
0979:E8                 INX
097A:D0 FB              BNE  BR7
097C:4C 7D 0C   BR8     JMP  DETOUR
097F:A5 07      OPPOS   LDA  ACCS       ;COMPLEMENT THE NEGATIVE NUMBER
0981:F0 40              BEQ  CMPB       ;THEN ADD
0983:A2 04              LDX  #04
0985:B5 00      BR9     LDA  ACCA,X
0987:49 FF              EOR  #$FF
0989:95 00              STA  ACCA,X
098B:CA                 DEX
098C:10 F7              BPL  BR9
098E:A0 04              LDY  #04
0990:38                 SEC
0991:B5 00      BR10    LDA  ACCA,X
0993:69 00              ADC  #00
0995:95 00              STA  ACCA,X
0997:CA                 DEX
0998:10 F7              BPL  BR10
099A:20 DC 09   FORTH   JSR  ADDNUM
099D:90 06              BCC  BR11
099F:A9 00              LDA  #00
09A1:85 07              STA  ACCS
09A3:F0 1B              BEQ  BR14
09A5:A9 FF      BR11    LDA  #$FF
09A7:85 07              STA  ACCS
09A9:A2 04              LDX  #$04
09AB:B5 10      BR12    LDA  RES,X
09AD:49 FF              EOR  #$FF
09AF:95 10              STA  RES,X
09B1:CA                 DEX
09B2:10 F7              BPL  BR12
09B4:A2 04              LDX  #04
09B6:38                 SEC
09B7:B5 10      BR13    LDA  RES,X
09B9:69 00              ADC  #00
09BB:95 10              STA  RES,X
09BD:CA                 DEX
09BE:10 F7              BPL  BR13
09C0:4C 7D 0C   BR14    JMP  DETOUR     ;GO TO ROUNDING ROUTINE
09C3:A2 04      CMPB    LDX  #04
09C5:B5 20      BR16    LDA  ACCB,X
09C7:49 FF              EOR  #$FF
09C9:95 20              STA  ACCB,X
09CB:CA                 DEX
09CC:10 F7              BPL  BR16
```

```
09CE:A2 04              LDX   #04
09D0:38                 SEC
09D1:B5 20        BR15  LDA   ACCB,X
09D3:69 00              ADC   #00
09D5:95 20              STA   ACCB,X
09D7:CA                 DEX
09D8:10 F7              BPL   BR15
09DA:30 BE              BMI   FORTH
09DC:A2 04        ADDNUM LDX  #04        ;SUBROUTINE THAT DOES THE ADDITION
09DE:18                 CLC
09DF:B5 00        KCAB  LDA   ACCA,X
09E1:75 20              ADC   ACCB,X
09E3:95 10              STA   RES,X
09E5:CA                 DEX
09E6:10 F7              BPL   KCAB
09E8:60                 RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

```
  00 ACCA        20 ACCB        07 ACCS         05 ACCX
0906 ADD       09DC ADDNUM    0938 ADJA       0958 BACK
  27 BCCS        25 BCCX      0991 BR10       09A5 BR11
09AB BR12      09C0 BR14      0912 BR1        09B7 BR13
09D1 BR15      09C5 BR16      091F BR2        093C BR3
0949 BR4       0974 BR6       0977 BR7        097C BR8
0985 BR9       09C3 CMPB      0C7D DETOUR     099A FORTH
092F HERE      09DF KCAB      097F OPPOS      0960 OPRAT
  10 RES       0946 ROTA      091C ROTB      ?0900 SUB
092D UP        0952 ZEROA     0929 ZEROB
```

©

A Floating-Point Division Routine

Marvin L. De Jong
Department of Mathematics-Physics
The School of the Ozarks
Pt. Lookout, MO

I. Introduction

In three previous articles in **COMPUTE!** we described:

1) a program that converts a decimal number (with a sign and an exponent) to a floating-point binary number (**COMPUTE!** #9)
2) a program that converts a floating-point binary number to a decimal number (**COMPUTE!** #11)
3) a program that multiplies two signed binary floating-point numbers (**COMPUTE!** #12).

In this article we describe a program that divides two floating-point binary numbers. Most of the programming described in this series has been relocatable allowing the user to move the programs or to put them in EPROMs with relative ease. Furthermore, the routines that were used to input and output the numbers can usually be found in a monitor, so that most of the code should be easily adapted to anyone's machine.

II. The Division Routine

Just as the multiplication routine does, the division routine uses three accumulators. The contents of accumulator A (ACCA) is divided *into* the contents of accumulator B (ACCB), and the quotient is stored temporarily in the result accumulator (RES) before the answer is moved back to the accumulator used by the output (floating-point binary to BCD routine) program.

Accumulator A occupies locations with addresses $0000 through $0003 with the most-significant byte in location $0000. The mantissa of the divisor is located in accumulator A. Location $0004 is used as a guard byte, permitting a 34-bit division before rounding the final answer to 32 bits. Thirty-two bits gives an answer that is accurate to approximately nine decimal digits. Accumulator B occupies locations with addresses $0020 through $0023 with a guard byte at location $0024. Accumulator B contains the dividend mantissa. The exponent and sign locations are the same as for the multiplication routine described earlier. The quotient is moved into RES at locations $0010 to $0014 as it is being calculated. When the calculation is finished, the quotient is moved to the accumulator that is used by the floating-point binary to BCD routine to output the answer. The accumulator architecture is exactly the same as for the multiplication routine described in the previous article.

The division algorithm is almost identical to the one you used in elementary school to do long division. Try one of these problems in decimal and then in binary if you want to understand the algorithm. Basically, it proceeds as follows:

1. Set COUNT = 34 = $22 to do a 34 bit division.
2. Calculate DIVIDEN – DIVISOR. If the carry flag is set then the DIVIDEND is greater than the DIVISOR, go to (3). Otherwise go to (4).
3. Replace the DIVIDEND with DIVIDEND – DIVISOR.
4. Shift the CARRY left into the LSB of the QUOTIENT.
5. Shift the new DIVIDEND left. (This is analogous to "bringing down" the next digit.)
6. Decrement COUNT. If COUNT is not zero, go to (2), otherwise go to (7).
7. Normalize and round the quotient.

As in the case of multiplication, the sign of the result is found by forming an exclusive-or with the signs of the divisor and the dividend. Recall from algebra that the exponent of the quotient is found by subtracting the exponent of the divisor from that of the dividend. If the exponent exceeds 127 or is less than -128, the program executes a BRK instruction. It is left to your imagination what you want your BRK routine to do for underflow or overflow. In my case the program simply jumps to the monitor. If the divisor is zero, the program also executes a BRK instruction. If the dividend is zero, the entire division routine is bypassed and the correct answer of zero is placed in the accumulator.

One final important point needs to be made. This division routine uses the same normalize and round instructions that the multiplication routine used. These instructions started at DETOUR ($0C7D) in the previous article and are not repeated here. Thus, you will find a JSR DETOUR instruction just before the routine ends.

In listing 2 you will find a short program to test the division routine. It also makes use of the subroutines published in the previous article in this series. In fact, it differs only in that it jumps to the division subroutine rather than the multiplication subroutine. It duplicates almost exactly Listing 5 in "A Floating Point Multiplication Routine," and you may wish to refer to that article for details.

Listing 1. The Floating-Point Division Routine.

```
$0000 = ACCA; Most-significant byte of the mantissa in accumulator A.
$0005 = ACCX; Exponent for accumulator A.
$0007 = ACCS; Sign byte for accumulator A.
$0010 = RES;  Most-significant byte of the quotient accumulator.
$0020 = ACCB; Most-significant byte of accumulator B, the dividend.
$0025 = BCCX; Exponent of the dividend.
$0027 = BCCS; Sign of the dividend.
$0A70 A5 00   START    LDA ACCA         Is the divisor zero?
 0A72 D0 01            BNE BR1          No.
 0A74 00               BRK              Yes.
 0A75 A5 20   BR1      LDA ACCB         Is the dividend zero?
 0A77 D0 05            BNE BR2          No.
 0A79 A9 00            LDA #00          Yes. Make the answer zero.
 0A7B 85 01            STA ACCA+1
 0A7D 60               RTS              Then return.
 0A7E A5 07   BR2      LDA ACCS         Calculate the sign of the quotient.
 0A80 45 27            EOR BCCS
 0A80 45 07            STA ACCS         Return sign to answer location.
 0A84 38               SEC              Now calculate the exponent.
 0A85 A5 25            LDA BCCX
 0A87 E5 05            SBC ACCX         Subtract exponents when dividing.
 0A89 50 01            BVC BR3          Overflow or underflow?
 0A8B 00               BRK              Yes. Go to BRK routine.
 0A8C 85 05   BR3      STA ACCX         No. Put result into answer location.
 0A8E 18               CLC
 0A8F A2 FC            LDX #$FC         Both the mantissa of the divisor and
 0A91 76 04   BR4      ROR ACCA+4,X     the mantissa of the dividend will now
 0A93 E8               INX              be shifted one bit to the right. It
 0A94 D0 FB            BNE BR4          just makes the division routine easier
 0A96 18               CLC              to write.
 0A97 A2 FC            LDX #$FC
 0A99 76 24   BR5      ROR ACCB+4,X
 0A9B E8               INX
 0A9C D0 FB            BNE BR5          So far so good. Next we will clear
 0A9E A9 00            LDA #00          the locations to store the answer.
 0AA0 A2 04            LDX #04
 0AA2 95 10   LOOP     STA RES,X
 0AA4 CA               DEX
 0AA5 10 FB            BPL LOOP         Answer locations cleared.
 0AA7 A0 22            LDY #$22         Bit count = $22 = 34. Start division.
 0AA9 38      CIRCLE   SEC
 0AAA A2 04            LDX #04          Start by comparing divisor to dividend.
 0AAC B5 20   BR6      LDA ACCB,X       Is the dividend greater than divisor?
 0AAE F5 00            SBC ACCA,X
 0AB0 CA               DEX
 0AB1 10 F9            BPL BR6
 0AB3 90 0B            BCC BR8          No. Then put a zero in the quotient.
 0AB5 A2 04            LDX #04          Yes. Subtract divisor from dividend
 0AB7 B5 20   BR7      LDA ACCB,X       and use the result as the new
 0AB0 F5 00            SBC ACCA,X       dividend. The carry flag will be
 0ABB 95 20            STA ACCB,X       set after this operation, and it
 0ABD CA               DEX              will be moved into the quotient.
 0ABE 10 F7            BPL BR7
 0AC0 A2 04   BR8      LDX #04          Here is where the carry flag gets
 0AC2 36 10   BR9      ROL RES,X        put into the quotient.
 0AC4 CA               DEX
 0AC5 10 FB            BPL BR9
 0AC7 A2 04            LDX #04          Now rotate the new dividend left.
 0AC0 18               CLC
 0ACA 36 20   BR10     ROL ACCB,X
 0ACC CA      DEX
 0ACD 10 FB            BPL BR10         Mission accomplished.
 0ACF 88               DEY              So decrement the bit counter.
 0AD0 D0 D7            BNE CIRCLE       Then branch back if it's not zero.
 0AD2 A0 00            LDY #00          Actually, you don't need this instruction.
 0AD4 A5 10   BR11     LDA RES          Here we normalize the mantissa and
 0AD6 30 0B            BMI BR13         adjust the exponent for all the shifting
 0AD8 18               CLC              done earlier.
 0AD9 A2 04            LDX #04
```

```
0ADB 36  10        BR12     ROL RES,X
0ADD CA                     DEX
0ADE 10  FB                 BPL BR12
0AE0 C8                     INY          Increment shift counter.
0AE1 D0  F1                 BNE BR11     Branch back until mantissa is normalized.
0AE3 84  0B        BR13     STY TEMP     Calculate the exponent adjustment.
0AE5 A9  07                 LDA #07
0AE7 38                     SEC
0AE8 E5  0B                 SBC TEMP
0AEA 18                     CLC
0AEB 65  05                 ADC ACCX

0AED 50  01                 BVC BR14     Overflow or Underflow?
0AEF 00                     BRK          Yes.
0AF0 85  05        BR14     STA ACCX     Final result into exponent.
0AF2 20  7D 0C              JSR DETOUR   Round and final normalization in
0AF5 60                     RTS          multiplication routine.
```

Listing 2. An Input/Output/Divide Calling Program.

```
$0050 20 00 0E   AGAIN   JSR INPUT    Call the BCD to Floating-Point Binary Routine.
 0053 20 B0 0F           JSR SUB1     Call the subroutine to modify the accumulator.
 0C56 20 C0 0F           JSR SUB2     Transfer ACCA to ACCB.
 0059 20 00 0E           JSR INPUT    Get the second number (divisor).
 005C 20 B0 0F           JSR SUB1     Fix the accumulator again.
 005F 20 70 0A           JSR DIVIDE   Divide the first number by the second.
 0062 20 00 0B           JSR OUTPUT   Convert the result to BCD and output it.
 0065 4C 50 00           JMP AGAIN    Try another pair of numbers.
```

A General Purpose BCD-To-Binary Routine

Marvin L. De Jong
Department of Mathematics-Physics
The School of the Ozarks
Pt. Lookout, MO

A number of routines have been published [1,2,3] that will convert either a two-digit number or a four-digit number in BCD code to a binary number, and Butterfield[4] has published a routine to handle a six-digit BCD number. The routine described here can be easily modified to handle any number of BCD digits. It is a 6502 assembly language interpretation of an algorithm found in Peatman's[5] book. The BCD-to-binary routine assumes its importance from the fact that human beings usually input numbers to a computer in a decimal representation. A number of scientific instruments have BCD outputs that may be interfaced to a microcomputer, requiring some kind of conversion routine before the data from such a device can be processed. Finally, if you want to interface some of the calculator chips to a microprocessor in order to do more complex arithmetic, you will very likely need a BCD-to-binary routine somewhere in your software. A 6502 assembly language routine to go the other way (binary-to-BCD) can be found as a subroutine in reference six at the end of this article.

The BCD-to-binary routine is based on a familiar technique for converting a base-ten number to a base-two number. The decimal number is successively divided by two, and the remainders are noted as either a one or a zero. Each division gives the next more significant binary digit or bit. Example 1 illustrates the process.

Example 1. Convert 59_{ten} to a binary number.

Solution: Successively divide 59_{ten} by two, with the divisions beginning from the right and proceeding to the left.

0	1	3	7	14	29
2) 1	2) 3	2) 7	2) 14	2) 29	2) 59
0	2	6	14	28	58
1	1	1	0	1	1

$59_{ten} = 111011_{two}$

Referring to Example 1 it can be seen that the algorithm requires that the BCD number be successively divided by two and the remainders are saved to become the binary number. The first division remainder is the least significant bit, while the remainder from the last division is the most significant bit. If in Example 1 we wanted to convert 59_{ten} to an eight-bit binary number, namely 0011 1011, we would simply perform two more divisions than shown, providing the two leading zeros in the eight-bit representation.

If you are mildly familiar with BCD numbers you will recall that each digit requires four bits (or one nibble). So an eight-digit decimal number requires four memory locations. Conversely, four memory locations can represent a decimal number as large as 99999999, which is more easily expressed as 10^8-1. Question: How many bits are needed to represent a given number of decimal digits? Let N be the largest number of decimal digits that we need for our particular application, so the largest decimal number is (10^N-1). Let n be the number of binary digits (bits) needed to represent the same number. By analogy, the largest binary number that can be represented by n bits is (2^n-1). Since we wish to represent the same number, we may equate (10^N-1) and (2^n-1) and then solve for n. Thus, with some mathematical magic, the answer to the question posed above is

$$N = N/\log 2 = N/0.30103$$

where a base ten logarithm is implied.

If $N = 8$ then $n = 26.6$ which becomes $n = 27$ when rounded upward (fractional numbers of bits are not allowed as answers for this problem). Twenty-seven bits can be handled quite nicely by four bytes, *but please do not create* your own theorem that the number of memory locations needed to represent a number in binary is equal to the number of memory locations to represent the same number in binary-coded decimal (BCD). Use the equation, and be sure to allocate enough memory to handle the number in either binary or BCD representations. Note that, in the program described by Listing 1, we assume an eight-digit decimal number is being converted to a binary number that will also be stored in four memory locations. The program is easily modified to handle situations where the number of memory locations needed for the BCD number is *different* than the number of memory locations needed for the binary number. Using the immortal words of many authors, "we leave this problem for the student."

So we know how many memory locations to assign to represent the number, and we have a simple algorithm (divide by two and store the remainder) to perform the conversion. Enter some corollary to Murphy's Laws: "nothing is as simple as it seems." Dividing by two is neat and easy for a binary number: successive shifts to the right (LSR or ROR) give successive divisions by two. Dividing by two is considerably more complex for a BCD number. Fortunately, Peatman[5] has pointed out a few tricks that accomplish division-by-two for a BCD number.

The eight bit "weights" in a byte of memory that represent a binary number are 1, 2, 4, 8, 16, 32, 64, and 128, proceeding from the right-most bit to the left-most bit. Clearly, shifting the number to the right divides each bit weight by two. That is why an LSR or an ROR instruction may be used to divide a binary number by two. However, if the same memory location represents a BCD number, then the bit weights are 1, 2, 4, 8, 10, 20, 40, 80. consequently, a shift-right or a rotate-right instruction results in division-by-two only for bits zero, one, two, three, five, six, and seven. Shifting bit four (with a weight of ten) to the right changes its weight to eight. Eight is three more than five, the number you usually get when you divide ten by two. So, the trick to dividing a BCD number by two is to shift right or rotate right as usual, but if a one is shifted from bit four to bit three, then you must subtract three from the shifted-right result to get the correct answer. That's it folks. I wish I could say it was my idea, but I found it in Peatman's[5] book.

```
BCDNUM   = $0000;   Base address of the BCD number to be converted to binary. The
                    most-significant digit of an N digit BCD number is in the high-
                    order nibble of BCDNUM.
BINUM    = $0010;   Base address of the binary number whose most-significant byte
                    will be in BINUM.
BYTE     = $FC;     Twos complement of the number of bytes needed to hold
                    the BCD number; in this program four bytes ($0000 - $0003)
                    are used.

$ 0D00  D8        START    CLD               Clear decimal mode.
  0D01  A9  00             LDA #00           Clear locations that will
  0D03  A2  FC             LDX #BYTE         hold the binary number.
  0D05  95  14    BACK     STA BINUM+4,X
  0D07  E8                 INX
  0D08  D0  FB             BNE BACK          Locations have been
  0D0A  38                 SEC               cleared.
  0D0B  A2  FC    THERE    LDX #BYTE         Rotate the binary number
  0D0D  76  14    RETURN   ROR BINUM+4,X     right, moving the remainder
  0D0F  E8                 INX               from the BCD division into
  0D10  D0  FB             BNE RETURN        the binary number.
  0D12  B0  2B             BCS OUT           If the carry is set, the conver-
  0D14  A2  FC             LDX #BYTE         sion is complete.
  0D16  76  04    AGAIN    ROR BCDNUM+4,X    Start the division-by-two by
  0D18  E8                 INX               shifting BCD number right.
  0D19  D0  FB             BNE AGAIN         Remainder will be in carry
  0D1B  08                 PHP               flag so save it on the stack.
  0D1C  A2  FC             LDX #BYTE         Test bit three of each byte to
  0D1E  38                 SEC               see if a one was shifted in.
  0D1F  B5  04    LAKE     LDA BCDNUM+4,X
  0D21  29  08             AND #08           If so, subtract three.
  0D23  F0  06             BEQ FORWD         If not, no correction needed,
  0D25  B5  04             LDA BCDNUM+4,X    so test bit seven of each byte
  0D27  E9  03             SBC #03           to see if a one was shifted in.
  0D29  95  04             STA BCDNUM+4,X
  0D2B  B5  04    FORWD    LDA BCDNUM+4,X    Here bit seven is checked.
  0D2D  29  80             AND #$80
  0D2F  F0  06             BEQ ARND          No correction.
  0D31  B5  04             LDA BCDNUM+4,X    Correction: subtract 30.
  0D33  E9  30             SBC #$30
  0D35  95  04             STA BCDNUM+4,X
  0D37  E8        ARND     INX
  0D38  D0  E5             BNE LAKE          Repeat for all N bytes.
  0D3A  28                 PLP               Get the carry back because it
  0D3B  B0  CE             BCC THERE         held the remainder.
  0D3D  90  CC             BCC THERE         Go back and put it in the
  0D3F  60        OUT      RTS               binary number. Then finish.
```

If the BCD number is to be represented by several bytes, an added complication occurs. Bit seven in the least-significant byte has a weight of 80. Bit zero in the next most significant byte has a weight of 100. Clearly, shifting a one from bit zero of this byte to bit seven of the least-significant byte does not result in a division-by-two because 100/2 is not 80. However, if we subtract 30 after the shift we do get the correct answer. When performing a divide-by-two operation on a multi-byte BCD number, each byte in the number must be tested to see if a one was shifted into either bit three or bit seven, and then the appropriate remedies must be applied if the tests are positive. In short, if a one is shifted into the most-significant bit position of any of the N nibbles used to represent the N digits in BCD, then the nibble must be corrected by subtracting three.

One other point remains to be made. From Example 1 it is clear that we are interested in the remainder after division-by-two. When dividing by two, the remainder is either zero (even dividend) or one (odd dividend). The remainder will be found in the carry flag after a shift-right operation.