On Status Flags, and the Emulation of Different Arithmetic Representations using Two's-Complement Hardware

Charles Abzug, Ph.D.
Department of Computer Science
James Madison University
Harrisonburg, VA 22807

Voice Phone: ***540-568-8746,*** **E-mail:** ***CharlesAbzug@ACM.org***
Home Page: ***http://www.cs.jmu.edu/users/abzugcx***

© 1999 Charles Abzug

NOTE: The code samples shown here are all correct in principle, but they are not guaranteed to be free of syntax and logical errors.

The functions of the four status flags are discussed in *Bebop Bytes Back.* The purpose of this tutorial exposition is to review the information presented in the textbook and to re-examine it in view of the needs both of multiple-precision arithmetic and of the different types of arithmetic that we have covered in this course. First, let us consider the basic principle of what is the fundamental significance of each flag, that is, what has that flag been supplied to tell us? All of the flags have been supplied for the purpose of informing the program/programmer about the status either of the last arithmetic or logical operation that was performed by the Arithmetic-Logic Unit (ALU), or sometimes of the last load operation performed. The significance of each flag is generally different. After defining the basic significance of each flag, let us see how the intended purpose of that flag presents a challenge in multiple-precision arithmetic, and how that challenge can be met through thoughtful programming. Finally, we shall consider the changes in the implementation of the status flags under the additional challenge presented by different kinds of integer arithmetic.

Revised 19 Oct, 29 Nov, and 08 Dec 1999

Carry Flag

The **Carry Flag** indicates that there is a carry out from the highest bit of the addition. When the sole purpose of the *ADD* operation is to add two eight-bit two's-complement numbers, then there is absolutely no significance to the presence of a '1' in this flag; it can safely be ignored. The principal use of the Carry Flag in Two's-Complement arithmetic is when multiple-precision arithmetic is being performed, that is, when the augend, the addend, and the sum are each composed of two or more bytes, which must be considered in concatenation to represent together a single number. Under these circumstances this flag has enormous significance. Consider a double-precision integer addition. This means that the augend, the addend, and the sum each consists of two bytes concatenated to form a single 16-bit number. To add such numbers in a digital computer designed for one-byte arithmetic, we must first add the Less Significant Byte of the addend to the Less Significant Byte of the augend to obtain the Less Significant Byte (LSB) of the sum. Then, we must add the two More Significant Bytes for the augend and the addend together. However, that is not enough to obtain the true sum. Consider first the Less Significant Byte in isolation. When we add the lowest addend bit to the lowest augend bit to obtain the lowest sum bit, we then have to take the carry out from that bit-wise addition and carry it in to the next bit of addition. Thus, we progress from bit zero up through bit seven, with each bit's carry-out becoming the carry-in to the next higher bit. All of this is done for us by the ALU; that is, the portion of the ALU that carries out the bit 0 addition produces a carry-out of either '0' or '1', which is then carried in to the bit 1 sum operation, and so on all the way up through bit seven.. Now, however, to effect the addition of two-byte numbers, the carry out from bit seven of the LSB summation must also become the carry in to the summation for bit zero of the MSB. The ALU has no ability by itself to do that for us; we must explicitly instruct it if we want to obtain that result. Consider the following assembly code to effect a two-byte (double-precision) integer addition:

```
                #Program to perform double-precision integer addition:

                .ORG        $4000       #Situate the program at location hex 4000.

A:              .2BYTE                  #Storage for 2 bytes of augend.
B:              .2BYTE                  #Storage for 2 bytes of addend.
C:              .2BYTE                  #Storage for 2 bytes of sum.

                #Perform the addition for the Less Significant Byte (LSB).
                LDA         [A + 1]
                    #Load the LSB of the augend into the accumulator.
                ADD         [B + 1]
                    #Add to it the LSB of the addend.
                STA         [C + 1]
```


© 1999 Charles Abzug

```
        #Store the result in the LSB of the sum.

#Perform the addition for the More Significant Byte (MSB).
LDA     [A]
        #Load the MSB of the augend into the accumulator.
ADDC    [B]
        # Add to it the MSB of the addend.
STA     [C]
        #Store the result in the MSB of the sum.
HALT
        #End of program;  cease to perform any further action.

.END    #Assembler end directive.
```

Note that there is a critical difference between the two halves of this program. Each half consists of a "load" instruction followed by an "add" instruction, and then a "store" instruction. The first three instructions are for the LSB, and the next three are for the MSB. But note that the "add" part of the program is different for the two bytes, in the first case being the simple *ADD* instruction and in the second case being the *ADDC* instruction. The difference between these instructions is in their use of the Carry Flag. The *ADD* instruction ignores the Carry Flag, performing a simple one-byte addition with a carry-in for bit 0 of '0'. The *ADDC* instruction, on the other hand, will perform its addition with a carry-in to bit 0 of ***either*** a '0' or a '1', depending on the value of the Carry Flag. Thus, with the use of this instruction for the addition of the MSB, we have successfully taken the carry-out from bit 7 of the LSB addition and used it to form the carry-in to bit 0 of the MSB addition. If you check Table C.2 on pages C-14 and C-15 of *Bebop Bytes Back*, which shows a summary of all *Beboputer* operations, you will note that neither the *STA* instruction nor the *LDA* instruction affects the value of the Carry Flag. Therefore the interposition of these two instructions between the *ADD* for the LSB and the *ADDC* for the MSB does not affect the status of the Carry Flag, and thus preserves the accuracy of the result. Note that *Bebop Bytes Back* contains a similar, though more complicated, example of code for a two-byte addition on pages G-5 through G-9.

It is a simple matter to extend this example from double-precision addition to triple-precision. This is left as an exercise to the reader.

Zero Flag

© 1999 Charles Abzug

The purpose of the **Zero Flag** is to indicate that the last arithmetic or logic or load operation resulted in a value of zero. Why do we need to know this? There are several possible reasons. Sometimes in writing a program we need to perform some operation repetitively (loop) for a fixed number of times. We can therefore set up some location as a "counter" variable, and each time we execute the loop we decrement the counter. How do we know when we are finished cycling through the loop? When we have reached a "counter" value of zero. Thus, we can use a conditional jump instruction (*JZ*) that examines the value of the Zero Flag to decide when to take us out of the loop. Consider the following piece of code:

```
                #Pseudo-Program to illustrate the use of the Zero Flag in implementing
                # a counted loop:

                .ORG        $4000       #Situate the program at location hex 4000.

COUNTER:        .BYTE       #Storage allocation for the Counter variable.

                LDA         12          #Define initial counter value of decimal 12.
                STA         [COUNTER]   #Initialize the counter.

LOOP:           NOP
                    #Replace this "No-Op" with some useful task sequence to be
                    #executed on every iteration of the loop.

                LDA         [COUNTER]   #Load the value of "Counter" .
                DECA                    #Decrement the value of "Counter".
                STA         [COUNTER]   #Store the new value of "Counter" and free
                                        #up the accumulator for some other usage.
                JZ          [OUT_LOOP]
                                #Exit the loop at its proper end ("Count" = '0').
                JMP         [LOOP]      #No exit;  back to the beginning of the loop.

OUT_LOOP:       #Some new task here to be executed when the loop is done.

                HALT
                    #End of program;  cease to perform any further action.

                .END        #Assembler end directive.
```


© 1999 Charles Abzug

Question: In the code segment above, the counter was initialized to a positive value, then decremented on each iteration through the loop until it reached a value of zero. Could a negative loop counter have been used instead? Consider the following code:

```
                #2nd Pseudo-Program to illustrate the use of the Zero Flag in implementing
                # a counted loop:

                .ORG        $4000       #Situate the program at location hex 4000.

COUNTER:        .BYTE       #Storage allocation for the Counter variable.

                LDA         12          #Define initial counter value of decimal 12.
                XOR         $FF
                            #Take the Ones'-Complement of the original value.
                INCA        #Increment to convert to the Ones'-Complement value to
                            # the Two's-Complement.
                STA         [COUNTER]
                            #Initialize the counter with the decimal value -12.

LOOP:           NOP         #Replace this "No-Op" with some useful task sequence to
                            # be executed on every iteration of the loop..

                LDA         [COUNTER]   #Load the value of "Counter"
                INCA                    #Increment the value of "Counter".
                STA         [COUNTER]   #Store the new value of "Counter" and free
                                        #up the accumulator for some other usage.
                JNZ         [LOOP]
                                    #Back to the beginning of the loop ("Count" != '0').

END-LOOP:       #Some new task here to be executed when the loop is done.

                HALT
                    #End of program;  cease to perform any further action.

                .END        #Assembler end directive.
```


© 1999 Charles Abzug

Two different methods have been shown for counting through the loop iterations, one using a positive-value loop counter which is decremented on each iteration through the loop, and the other using a negative-value loop counter which is incremented on each iteration of the loop. Which method would you prefer to use, and why? The two code segments also illustrate two different methods for checking the loop exit condition and exiting from the loop. There is no correlation between the method used in each case for counting the loop iterations and the method for checking the loop exit condition. Either method for counting loop iterations could have been combined with either method for checking the loop exit condition. Which method for checking loop exit condition is better, and why?

A second possible use for the Zero Flag is to provide a way to avoid dividing by zero. When the result of one operation is to be divided into another number, some alternative action must be taken in the event that the intended divisor turns out to have a value of zero. The Zero Flag, in conjunction with the conditional branch operations *JZ* and *JNZ,* provides a convenient means for checking for this circumstance and branching to the alternative code segment.

For the case of both code segments shown above, the determination of whether a zero value was present was easy to do; since only a single byte was used to store the "counter" variable, the **Zero Flag** as set up by the processor was sufficient. How can we set up to check a double-precision integer for the possibility of a zero value? Consider the following code segment:

```
                #Program to perform double-precision integer Two's-Complement
                # addition and properly set a Zero Flag.:

                .ORG        $4000       #Situate the program at location hex 4000.

A:              .2BYTE          #Storage for 2 bytes of augend.
B:              .2BYTE          #Storage for 2 bytes of addend.
C:              .2BYTE          #Storage for 2 bytes of sum.
Z_FLAG:         .BYTE           #Storage for a soft Zero Flag.

                #Perform the addition for the Less Significant Byte (LSB).
                LDA         [A + 1]
                    #Load the LSB of the augend into the accumulator.
                ADD         [B + 1]
                    #Add to it the LSB of the addend.
```


© 1999 Charles Abzug

```
                STA         [C + 1]
                    #Store the result in the LSB of the sum.

                #Perform the addition for the More Significant Byte (MSB).
                LDA         [A]
                    #Load the MSB of the augend into the accumulator.
                ADDC        [B]
                    #Add to it the MSB of the addend, propagating any possible
                    # carry from the .LSB summation.
                STA         [C]
                    #Store the result in the MSB of the sum.

                LDA         0       #Initial value of Zero Flag.
                STA         [Z_FLAG]

                LDA         [C + 1]         # Is LSB of Sum = '0'?
                JNZ         [FINISHED]      # If not, then flag must remain cleared.
                LDA         [C]             # Is MSB of Sum also = '0'?
                JNZ         [FINISHED]      # If not, then flag must remain cleared.

                            #However, if both bytes are zero, then the flag must be set.
                LDA         $02             #Zero Flag assigned a value of '1' in
                                            # the appropriate bit position..
                STA         [Z_FLAG]        #See Figure 8.19 for proper bit position.

FINISHED:       HALT
                    #End of program;  cease to perform any further action.

                .END        #Assembler end directive.
```

Negative Flag

The **Negative Flag** is used to indicate that the value produced by the last arithmetic or logical or load operation is negative. The utility of having such a facility should be fairly obvious. For Two's-Complement arithmetic, the value of the **Negative Flag** is simply equal to the value of the leftmost bit of the number. For multiple-precision integer arithmetic, this will be the leftmost bit of the MSB. When the arithmetic is carried out byte by byte, starting from the Least Significant Byte and going on up to the Most Significant Byte, then the value of the **Negative Flag** resulting from the last operation will be correct for the entire number.

© 1999 Charles Abzug

Overflow Flag

The purpose of the **Overflow Flag** is to indicate that the results of the latest arithmetic operation cannot be stored in the number of bits available to hold it. This illustrates vividly how computer arithmetic is different from the arithmetic that people normally use in their daily lives. For example, when the Unites States of America was founded, the first annual budget for expenditures authorized by Congress was around $7 million. As the years have gone by and the amount of funds expended annually has increased to the present level of over $1 trillion, we have just added a new column onto the left end of the number representation whenever necessary to hold the larger number. In a paper system, this is very easy to do. In a computer system, we are absolutely limited; the hardware allows for only so many bits. Over the years, as hardware has become cheaper to produce, the width of the hardware registers (i.e., the number of bits in the register) has increased. In the earliest microprocessors, four bits was the norm. It didn't take too long to increase this to eight bits. Then came sixteen bits, more recently 32, and the world of microprocessors is seeing already an appreciable movement to 64 bits. Some supercomputers already have 128-bit word lengths. However, there will probably always be some need for performing arithmetic on numbers wider than the current register width, i.e., there will always be a need for multiple-precision arithmetic. We must get around the limitation of processor word width whenever necessary by concatenating several word widths or register widths, in the case of the ***Beboputer*** by concatenating **bytes,** to form a super-number. But that is not automatically done by the computer; it must be programmed. No matter how many bytes we concatenate to form the number, we must always take into account the possibility that the result of a calculation may be a number that is too big to store. Hence, the need for the **Overflow Flag.** How does the computer know how to set the **Overflow Flag?** Here, it is useful to start out with Unsigned-Number arithmetic, and then to see how implementation differs for two's-complement and ones'-complement arithmetic.

Remember first the primal significance of the **Overflow Flag:** that the number resulting from the operation is too big to be stored in the number of bits available. In the case of Unsigned Numbers, the criterion is very simple. If we had eight-bit unsigned numbers, then we could store numbers ranging from zero up through +255 (= 2^8 - 1). Whenever the sum or other ALU operation resulted in a carry-out from bit 7 (the most significant bit), that would mean that the resultant number is too big to store in eight bits, and therefore there is an overflow. Thus, for Unsigned-Number arithmetic, a carry-out from the most significant bit is equivalent to an overflow.

For Ones'- and Two's-Complement arithmetic, the situation is a little more complex. Here, it is useful to consider two different criteria for overflow. The two criteria produce identical results, but one criterion is easier for humans to understand while the other is a little simpler for computer hardware to implement. The easier criterion for humans to comprehend (or to implement in software) is that an overflow will occur whenever two numbers of like sign are added together, and the result is of opposite

© 1999 Charles Abzug

sign. Thus, for example, using eight-bit *Unsigned-Number* arithmetic, the sum of 98 and 103 would work out as follows:

Carry -out	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**Carry-in**		**Decimal Value**
	0	1	1	0	0	0	1	0		Augend	+98
	0	1	1	0	0	1	1	1		Addend	+103
0	1	1	0	0	1	0	0	1	0	Sum	+201

There is no carry-out from the Most Significant Bit, and therefore there is no overflow. What that means is that the correct sum of the augend and addend will fit into width of the designated storage space. We recognize that this is true, in that the correct sum of decimal 201 (98 + 103) does, indeed, fit into an 8-bit register when the contents are to be interpreted as an Unsigned Number.

Taking the identical register contents but considering them to represent Two's-Complement numbers, however, results in a different analysis, as follows:

Carry -out	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**Carry-in**		**Decimal Value**
	0	1	1	0	0	0	1	0		Augend	+98
	0	1	1	0	0	1	1	1		Addend	+103
0	1	1	0	0	1	0	0	1	0	Sum	***-55***

There is still no carry-out from the Most Significant Bit, but yet now there is an overflow. The reason for this is that while the bits are identical to what they were before, but now their interpretation as a number follows different rules. The overflow is a reflection of the fact that two positive numbers have been added, but the sum, which is now interpreted in accordance with the rules of Two's-Complement representation, is a negative number. This is because the true sum of decimal 98 and 103, which is 201, is not representable in an 8-bit Two's-Complement number system, where the largest positive number that can be represented is +127. Similarly, if we add two negative numbers and obtain a positive result, then this, too, implies an overflow, as in the following example:

Carry -out	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**Carry-in**		**Decimal Value**
	1	0	0	1	1	1	1	0		Augend	-98
	1	0	0	1	1	0	0	1		Addend	-103
1	0	0	1	1	0	1	1	1	0	Sum	***+55***

In this case there happens to be a carry-out from the Most Significant Bit, but clearly this is **not** the reason for the overflow; remember that in the previous example there was an overflow but not a carry-out.

For Ones'-Complement arithmetic, the criterion for overflow is identical to that for Two's-Complement: addition of two numbers of like sign results in a sum of the opposite sign.

© 1999 Charles Abzug

For technical reasons related to the complexity of the logic circuitry, computer hardware uses a different, but equivalent criterion for determining whether or not the overflow bit should be set for Ones'- or Two's-Complement arithmetic.

Performing Arithmetic in a Representation Not Built into the CPU

Modern processors typically have the built-in ability to perform arithmetic in several different representation schemes. For example, both Unsigned Numbers and Two's-Complement are frequently found co-existing in the same processor. What this means is that the hardware will follow the appropriate form of arithmetic, ***depending on which op code*** is specified. What happens if it should be necessary to carry out arithmetic in accordance with still another representational scheme, one that the processor is not built to handle? No problem: just emulate the necessary scheme in software. For example, we have already examined above a code segment that sets an artificial Zero Flag in software for double-precision Two's-Complement arithmetic, since the processor only handles in native mode single-precision Two's-Complement arithmetic. Now, let us take a look at how we would examine the same arithmetic operation (i.e., arithmetic carried on the same bit-sequences of 16-bit Augend and Addend operands) to set the Zero Flag appropriately for double-precision Ones'-Complement arithmetic:

> **NOTE** that from a purely superficial perspective this might appear to be a pointless exercise. If the processor does not perform a particular kind of arithmetic, then why do we need to generate software that would make the processor appear to carry out that missing hardware function? Well, the reality is that this **kind** of problem is very much akin to what happens in modern processors when a Reduced Instruction Set Computer (RISC) is used. The RISC is characterized by a smaller instruction set than is present in an older type of traditional, or Complex Instruction Set Computer (CISC). What happens when it is required to carry out an instruction that had been provided in the older CISC processor, but the newer RISC processor does not have that instruction available? The answer is very simple: the instruction is carried out by means of software emulation. Thus, writing a software program to emulate the function of an instruction missing from the hardware is actually a common requirement.

```
#Program to perform double-precision integer Ones'-Complement
# addition, using Two's-Complement hardware, and properly set
# a Zero Flag in software:
```


© 1999 Charles Abzug

```
                .ORG        $4000       #Situate the program at location hex 4000.

A:              .2BYTE                  #Storage for 2 bytes of augend.
B:              .2BYTE                  #Storage for 2 bytes of addend.
C:              .2BYTE                  #Storage for 2 bytes of sum.
Z_FLAG:         .BYTE                   #Storage for a soft Zero Flag.

                #Perform the addition for the Less Significant Byte (LSB).
                LDA         [A + 1]
                    #Load the LSB of the augend into the accumulator.
                ADD         [B + 1]
                    #Add to it the LSB of the addend.
                STA         [C + 1]
                    #Store the result in the LSB of the sum.

                #Perform the addition for the More Significant Byte (MSB).
                LDA         [A]
                    #Load the MSB of the augend into the accumulator.
                ADDC        [B]
                    # Add to it the MSB of the addend.
                STA         [C]
                    #Store the result in the MSB of the sum.


                #Check for the need to perform an "end-around carry", and execute it if
                # necessary:

                JNC         [Z_CHECK]   #No need for "end-around carry";
                                        # proceed to check for Zero.
                LDA         [C + 1]     #Perform "end-around carry" on LSB.
                INCA
                STA         [C + 1]

                LDA         [C]         #Propagate carry-out bit resulting from
                                        # the "end-around carry" executed on
                                        # the LSB to the MSB.
                ADDC        $00
                STA         [C]
```


© 1999 Charles Abzug

```
                #Arithmetic is finished;  set the correct value for the Ones'-Complement
                # soft Zero Flag.

Z_CHECK:    LDA     0           #Initialize the value of the soft Zero Flag.
            STA     [Z_FLAG]    #Clear the soft Zero Flag.

                #First, check for Negative Zero;  if present, then set the soft Zero Flag.

NEG_ZERO:   LDA     [C + 1]     # Is LSB of Sum = "-0"?
            XOR     $FF
            JNZ     [POS_ZERO]
            LDA     [C]         # Is MSB of Sum also = "-0"?
            XOR     $FF
            JNZ     [POS_ZERO]
            LDA     $02         #Assign to the Zero Flag a value of '1'
                                # in the appropriate bit position.
            STA     [Z_FLAG]    #See Figure 8.19 for proper bit position.
            JMP     [FINISHED]  #Negative zero was detected; nothing
                                # more needs to be done; jump
                                # unconditionally to end of program.


                #Negative zero not found; check for Positive Zero, and if present, set flag.

POS_ZERO:   LDA     [C + 1]     # Is LSB of Sum = '0'?
            JNZ     [FINISHED]  #No?  Then the possibility of setting the
                                # Zero Flag has conclusively been ruled out.
            LDA     [C]         # Is MSB of Sum also = '0'?
            JNZ     [FINISHED]  #No?  Then the possibility of setting the
                                # Zero Flag has conclusively been ruled out.
            LDA     $02         #Zero Flag assigned value of '1' in the
                                # appropriate bit position.
            STA     [Z_FLAG]    #See Figure 8.19 for proper bit position.

FINISHED:   HALT
                #End of program;  cease to perform any further action.

            .END    #Assembler end directive.
```


© 1999 Charles Abzug

Now, let's add some code to get the other status flags as well:

```
                #Program to perform double-precision integer Ones'-Complement
                # addition and properly set all status flags:

                .ORG        $4000       #Situate the program at location hex 4000.

A:              .2BYTE          #Storage for 2 bytes of augend.
B:              .2BYTE          #Storage for 2 bytes of addend.
C:              .2BYTE          #Storage for 2 bytes of sum.
Z_FLAG:         .BYTE           #Storage for a soft Zero Flag.
AUG_SIGN:       .BYTE           #Storage for a soft flag to indicate augend sign.
ADD_SIGN:       .BYTE           #Storage for a soft flag to indicate addend sign.
SUM_SIGN:       .BYTE           #Storage for a soft flag to indicate sum sign.
V_FLAG:         .BYTE           #Storage for a soft oVerflow Flag.
N_FLAG:         .BYTE           #Storage for a soft Negative Flag.
C_FLAG:         .BYTE           #Storage for a soft Carry Flag.
STAT_REG:       .BYTE           #Storage for a soft "Status Register".

                #Perform the addition for the Less Significant Byte (LSB).
                LDA         [A + 1]
                    #Load the LSB of the augend into the accumulator.
                ADD         [B + 1]
                    #Add to it the LSB of the addend.
                STA         [C + 1]
                    #Store the result in the LSB of the sum.


                #Perform the addition for the More Significant Byte (MSB).
                LDA         [A]
                    #Load the MSB of the augend into the accumulator.
                ADDC        [B]
                    # Add to it the MSB of the addend.
                STA         [C]
                    #Store the result in the MSB of the sum.
```


© 1999 Charles Abzug

```
                #Check for Carry-Out; and execute "end-around carry" if necessary:

C_CHECK:    LDA     0       #Initial value of soft Carry Flag.
            STA     [C_FLAG]    #Clear the soft Carry Flag.
            JNC     [N_CHECK]   #Leave it cleared, if appropriate, and
                                #  also skip "end-around carry".


            LDA     [C + 1]     #Perform "end-around carry" on LSB.
            INCA
            STA     [C + 1]

            LDA     [C]         #Propagate carry bit from LSB to MSB.
            ADDC    $00
            STA     [C]


                #Arithmetic is finished;  set correct values for soft flags.


                                #Check again for carry bit after "end-around
                                #  carry".
            JNC     [N_CHECK]   #Leave the soft carry flag cleared, if
                                #  appropriate.
            LDA     $01         #Carry Flag assigned a value of '1' in the
                                #  appropriate bit position.
            STA     [C_FLAG]    #See Figure 8.19 for proper bit position.


                #Check for Negative result, and assign correct value to soft N_FLAG:

N_CHECK:    LDA     0           #Initial value of soft Negative Flag.
            STA     [N_FLAG]    #Clear the soft Negative Flag.
            LDA     [C]         #Examine the MSB of the sum.
            ROLC                #Copy the sign bit of the sum into
                                #  the hardware Carry Flag.
            JNC     [Z_CHECK]   #Leave the soft Negative Flag cleared,
                                #  if appropriate.
```


© 1999 Charles Abzug

```
                LDA     $04     #Negative Flag assigned a value of '1' in
                                # the appropriate bit position.
                STA     [N_FLAG] #See Figure 8.19 for proper bit position.


                #Set correct value for Ones'-Complement # Z_FLAG.

Z_CHECK:        LDA     0       #Initialize the value of the soft Zero Flag.
                STA     [Z_FLAG] #Clear the soft Zero Flag.

                #First, check for Negative Zero;  if present, set flag.

NEG_ZERO:       LDA     [C + 1] # Is LSB of Sum = "-0"?
                XOR     $FF
                JNZ     [POS_ZERO]
                LDA     [C]     # Is MSB of Sum also = "-0"?
                XOR     $FF
                JNZ     [POS_ZERO]
                LDA     $02     #Zero Flag assigned value of '1' in the
                                # appropriate bit position.
                STA     [Z_FLAG] #See Figure 8.19 for proper bit position.
                JMP     [V_CHECK] #Negative zero was detected; nothing
                                # more needs to be done; jump to end
                                # of program.


                #Negative zero not found; check for Positive Zero, and if present, set flag.

POS_ZERO:       LDA     [C + 1] # Is LSB of Sum = '0'?
                JNZ     [V_CHECK]
                LDA     [C]     # Is MSB of Sum also = '0'?
                JNZ     [V_CHECK]
                LDA     $02     #Zero Flag assigned value of '1' in the
                                # appropriate bit position.
                STA     [Z_FLAG] #See Figure 8.19 for proper bit position.


                #Check for oVerflow and assign correct value to soft flag:
```


© 1999 Charles Abzug

```
V_CHECK:  LDA     0       #Initialize the value of the soft oVerflow Flag.
          STA     [V_FLAG]  #Clear the soft oVerflow Flag.

          LDA     $00       #Initialize sign flags for augend, for addend,
                            # and for sum to positive.
          STA     [AUG_SIGN]
          STA     [ADD_SIGN]
          STA     [SUM_SIGN]


          LDA     [A]       #Check the value of the augend sign.
          ROLC
          JNC     [ADD_CHK] #No carry means that augend was, indeed,
                            # positive.  Skip to code to check sign of
                            # addend.
          LDA     $FF
          STA     [AUG_SIGN] #Reset augend sign flag to negative.

ADD_CHK:  LDA     [B]       #Check the value of the addend sign.
          ROLC
          JNC     [SUM_CHK] #No carry means that addend was, indeed,
                            # positive.  Skip to code to check sign of
                            # sum.
          LDA     $FF
          STA     [ADD_SIGN] #Reset addend sign flag to negative.


SUM_CHK:  LDA     [C]       #Check the value of the sum sign.
          ROLC
          JNC     [CHK_V]   #No carry means that sum was, indeed,
                            # positive.  Skip to code that examines all
                            # three signs (augend, addend, and sum),
                            # to determine whether or not an overflow
                            # condition exists.
          LDA     $FF
          STA     [SUM_SIGN] #Reset sum sign to negative.


          #  All three signs have been examined and recorded.  Now, check
```


© 1999 Charles Abzug

```
                #  for existence of overflow and set the soft oVerflow Flag if appropriate.

CHK_V:          LDA     [AUG_SIGN]
                CMPA    [ADD_SIGN] # Do both augend and addend have the
                                   # same sign?
                JNZ     [FINISHED] #Augend and addend have different sign;
                                   # overflow is impossible.  Skip remaining
                                   # overflow-check code.
                CMPA    [SUM_SIGN] #Augend and addend have same sign;  does
                                   # the sum also have the same sign?
                JZ      [FINISHED] #Sum has same sign as both operands;
                                   # overflow is not present.  Skip remaining
                                   # overflow check code
                LDA     $08        #Assign the oVerflow Flag a value of '1' in
                                   # the appropriate bit position.
                STA     [V_FLAG]   #See Figure 8.19 for proper bit position.


                #All the individual soft flags have been assigned appropriate values.
                #Now, put them all together.

FINISHED:       BLDSP   $EFFF      #Initialize the Stack Pointer to the highest
                                   # addressable value in memory.  This leaves
                                   # as much space as possible for program and
                                   # stack both to grow, without interfering
                                   # with each other.
                LDA     [V_FLAG]
                OR      [C_FLAG]
                OR      [N_FLAG]
                OR      [Z_FLAG]
                STA     [STAT_REG] #Preserve values of flags in a software
                                   # "status register" for later use.

                PUSHA   #Alternative:  set the value in the actual hardware Status
                        # Register to suit the Ones'-Complement arithmetic result.
                        # First, copy the contents of the accumulator onto the stack.

                POPSR   #Next, pop the top of the stack into the Status Register.


                HALT #End of program;  cease to perform any further action.
```


© 1999 Charles Abzug

```
.END            #Assembler end directive.
```

Extending the Methodology to Any Level of Precision and to Any Form of Integer Arithmetic

We have now seen how easy it is to extend the use of the processor in software so that its in-built one-byte Two's-Complement ALU can be made to produce correct results both for double-precision integers and for Ones'-Complement arithmetic. It is a perfectly straightforward matter to extend this concept further to any desired multiple of precision, and to other forms of integer arithmetic, such as Unsigned-Number and Saturation Arithmetic, as well.

© 1999 Charles Abzug