SX Arithmetic Routines

Application Note 13

July 1999

1.0 Introduction

This application note presents programming techniques for performing commonly found arithmetic operations, such as multi-byte binary addition and subtraction, multi-digit BCD addition and subtraction, multiplication and division.

2.0 Binary Addition and Subtraction

The default configuration of SX device is to ignore the carry flag in addition and subtraction operations even the results of those operations do affect that flag. For multi-byte arithmetic operations, it is often desirable for the result of lower bytes to propagate to higher bytes by means of the carry flag.

To enable the effect of the carry flag, ***carryx*** must be included in the list of device directives which are specified before the instructions, to make the carry flag an input to ADD, and SUB instructions.

The carry flag should be set to zero first before any addition.

The SUB instruction will set the carry flag to zero if there is an underflow. Therefore, it is necessary for us to set it to one before any subtraction is performed.

The following program segment illustrates 32 bit binary addition. The 4-byte operand1 and the 4-byte operand2 are added together. The result is put back into operand2.

Note that operand1 is located at locations 8, 9, a, b, hence 10xx binary and operand2 is at locations c, d, e, f or 11xx binary. Therefore toggling bit 2 of the FSR register effectively enable us to switch back and forth among the two operands. With that in mind, the indirect addressing of SX helps in saving code by just using IND as the register pointed to by FSR.

This routine assumes that the two operands are adjacent to one another and operand1 starts at the 08 location. To relocate the operands to other locations, make sure that they are still adjacent to one another, thus occupying a contiguous 8 bytes, and that operand1 is aligned to x0 or x8. The only change needed in the code will be the ending condition. Note that in the example, we tested bit 4 which will be toggled after the **inc. fsr** instruction if FSR was $f, and therefore pointing to the last byte. To make the routine work with operands located in $10-$17, for example, would need the ending condition be changed from **sb fsr.4** to **sb fsr.3** since the **inc fsr** instruction will change the address of last byte from $17 to $18 (%00011000) and set bit 3. Using this technique, we can save the need to store the count separately in order to keep track of the number of bytes added.

```
                ;32 bit addition
                ;entry = 32 bit operand1 and 32 bit operand2 in binary form
                ;exit = operand2 become operand1 + operand2, carry flag=1 for overflow from MSB
add32            clc                        ; clear carry, prepare for addition
                 mov   fsr,#operand1        ; points to operand 1 first
add_more   clrb   fsr.2                     ; toggle back to operand 1
           mov    W,ind                     ; get contents into the work register
           setb   fsr.2                     ; points to operand 2
           add    ind,W                     ; operand2=operand2+operand1
           inc    fsr                       ; next byte
           sb     fsr.4                     ; done? (fsr=$10?)
           jmp    add_more                  ; not yet
           ret                              ; done, return to calling routine
```

The 32 bit subtraction routine is very similar to addition, except that we set the carry flag first to indicate no underflow. Note that the result is in operand2 and it is operand2-operand1, not the other way around. When the carry flag is 0 on return, it means that the result is negative and is therefore in 2's complement form.

```
              ;32 bit subtraction
              ;entry = 32 bit operand1 and 32 bit operand2 in binary form
              ;exit  = operand2 become operand2-operand1, carry flag=0 for underflow from MSB
sub32         stc                          ; set carry, prepare for subtraction
              mov    fsr,#operand1         ; points to operand 1 first
sub_more      clrb   fsr.2                 ; toggle back to operand 1
              mov    W,ind                 ; get contents into the work register
              setb   fsr.2                 ; points to operand 2
              sub    ind,W                 ; operand2=operand2-operand1
              inc    fsr                   ; next byte
              sb     fsr.4                 ; done? (fsr=$10?)
              jmp    sub_more              ; not yet
              ret                          ; done, return to calling routine
```

3.0 BCD Addition and Subtraction

In applications where calculation result needs to be displayed, BCD or binary coded decimal can be much more easily converted into visual form, as in the case of adding machine or calculator.

The algorithm here for BCD addition is very similar to binary addition except for 1 important difference: decimal adjustment or correction. The need for such operation will be evident as we examine the follow simple addition:

```
85+15 = 9A
```

Obviously the correction result should be 100 in BCD. We can see that by adding 6 to the least significant digit (LSD), in this case, $9A+6=$A0, will correct the LSD. Finally, by adding a $60 to the whole number (equal to adding 6 to the most significant digit, MSD), the entire number is corrected to $00 with a carry of 1, which can be propagated into the next byte.

By looking at another example: 19+19=32. After the addition, the digit carry will be set to one, indicating an overflow in the LSD. The result then can be corrected by adding 6 to the LSD, giving us the correct answer of 38.

In general, we will do a correction on LSD of the result if the digit carry is set or the LSD is greater than 9. The same is true for the MSD. It will be corrected, i.e., added with 6, when the carry bit is set or the MSD is greater than 9.

The tricky part now is how to check if the digit is greater than 9. A straight implementation will require masking 1 nibble off at a time and do a subtraction. This will require additional storage if we do not want the operands (and the result) to be changed. The way it is implemented here is a bitwise comparison.

Let us look at a 4 bit number, if bit number 3 is 0, the number must be then%0xxx, and therefore ranges from 0-7, hence less than 9. If that's not the case, then we go on to check bit 2. If it is a one, then we have%11xx, and the number is definitely bigger than 9, since the minimum is already%1100 or 12. If bit 2 is a zero, we proceed to check bit 1. If this bit is a zero, then we have%100x, which means the number is either 8 or 9, and no correction is needed. But if bit 1 is an one, then we have %101x, which is higher than 9 and correction will be needed.

This method of detecting whether the digit is greater than 9 or not, is used twice in the code. Once for LSD and once for MSD. The changes is only the bit number that is being checked on.

One more point worth noting is the carry bit. After the initial binary addition, we have to store the carry bit that is used to propagate the result to higher bytes. The reason for doing this is simple: the decimal correction process of adding 6 to the number will clear the carry bit.

Notice also that the ending condition has been changed to **sb fsr.2** instead of **sb fsr.4.** This is simply because the code happens to point at operand 1 at that time and it just saves us code to check if fsr is pointing to the last byte of operand 1 at location $0b (%1011) or not. The fsr will be $0c (%1100) after the increment operation and therefore setting bit 2.

```
              ;8 BCD digit addition
              ;entry = 8 BCD digit operand1 and 8 BCD digit operand2 in BCD form
              ;exit  = operand2 become operand2+operand1, carry flag=1 for overflow from MSB
              ;      operand1 will be DESTROYED
badd32        clc                             ; clear carry, prepare for addition
              mov    fsr,#operand1            ; points to operand 1 first
badd_more
              mov    W,ind                    ; get contents into the working register
              clr    ind
              setb   fsr.2                    ; points to operand 2
              add    ind,W                    ; operand2=operand2+operand1
              clrb   fsr.2
              rl     ind                      ; store carry bit which will be altered by decimal
                                              ; adjustment (adding 6)
              setb   fsr.2                    ; points back to operand 2
              snb    status.1                 ; digit carry set? if so, need decimal correction
              jmp    dcor

              jnb    ind.3,ck_overflow        ; if 0xxx, check MSD
              jb     ind.2,dcor               ; if 11xx, it's >9, thus need correction
              jnb    ind.1,ck_overflow        ; 100x, number is 8 or 9, no decimal correction

              ; here if 101x, decimal adjust
dcor          clc                             ; clear effect of previous carry
              add    ind,#6                   ; decimal correction by adding 6

              ; finish dealing with least significant digit, proceed to MSD
ck_overflow   clrb   fsr.2                    ; points to operand1
              jb     ind.0,dcor_msd           ; stored carry=1, decimal correct
              ; test if MSD > 9
              setb   fsr.2                    ; points back to operand2
              jnb    ind.7,next_badd          ; if 0xxx, it's <9, add next byte
              jb     ind.6,dcor_msd           ; if 11xx, it's >9, thus need correction
              jnb    ind.5,next_badd          ; if 100x, it's <9

              ;here if 101x, decimal adjust
dcor_msd      clc                             ; clear effect of carry
              setb   fsr.2                    ; make sure that it's pointing at the result
              add    ind,#$60                 ; decimal correct


next_badd     clrb   fsr.2                    ; points to stored carry
              snb    ind.0                    ; skip if not set
              stc                             ; restore stored carry
              inc    fsr                      ; next byte
              sb     fsr.2                    ; done? (fsr=$0c?)
              jmp    badd_more                ; not yet
              ret                             ; done, return to calling routine
```

BCD subtraction is very similar to addition except for a few notes, which are summarized below:

- Carry flag is set first before subtraction which means no borrow;
- Decimal correction is done when:
  - digit carry is 0;
  - least significant digit (LSD) is greater than 9;
  - carry is 0;
  - most significant digit (MSD) is greater than 9;
- when the result is negative, it is not suitable for display, e.g., on 7 segment LEDs. Therefore, an operation which negates the number is performed by 0-result. This will enable us to obtain the magnitude of the number. The no carry condition will keep us reminded of the fact that it is a negative number. This situation is also occurring in a binary subtraction, whereas a no carry condition means the result is in 2's complement form. This is fine since the 2's complement is not used for display and it is useful for further computation.

```
                ;8 BCD digit  subtraction
                ;entry = 8 BCD digit operand1 and 8 BCD digit operand2 in BCD form
                ;exit  = operand2 become operand2-operand1, carry flag=0 for underflow from MSB
                ;                                         carry flag=1 for positive result
                ;      operand1 will be DESTROYED
bsub32          call   bs32                     ; do subtraction
                snc                             ; no carry=underflow?
                jmp    bs_done                  ; carry=1 positive, done
                call   neg_result               ; yes, get the magnitude, 0-result
                call   bs32                     ; keep in mind that this result is a negative
                                                ; number (carry=0)
bs_done         ret

bs32            stc                             ; set carry, prepare for subtraction
                mov    fsr,#operand1            ; points to operand 1 first
bsub_more
                mov    W,ind                    ; get contents into the working register
                clr    ind
                setb   ind.7                    ; set to 1 so that carry=1 after rl instruction
                setb   fsr.2                    ; points to operand 2
                sub    ind,w                    ; operand2=operand2+operand1
                clrb   fsr.2
                rl     ind                      ; store carry bit which will be altered by decimal
                                                ; adjustment (adding 6)
                setb   fsr.2                    ; points back to operand 2
                sb     status.1                 ; digit carry set? if so, need decimal correction
                jmp    dec_cor

                jnb    ind.3,ck_underflow       ; if 0xxx, check MSD
                jb     ind.2,dec_cor            ; if 11xx, it's >9, thus need correction
                jnb    ind.1,ck_underflow       ; 100x, number is 8 or 9, no decimal correction

                ; here if 101x, decimal adjust
dec_corstc                                      ; clear effect of previous carry
                sub    ind,#6                   ; decimal correction by subtracting 6

                ; finish dealing with least significant digit, proceed to MSD
ck_underflow    clrb   fsr.2                    ; points to operand1
                jnb    ind.0,dadj_msd           ; stored carry=0, decimal adjust
                ; test if MSD > 9
                setb   fsr.2                    ; points back to operand2
                jnb    ind.7,next_bsub          ; if 0xxx, it's <9, add next byte
                jb     ind.6,dadj_msd           ; if 11xx, it's >9, thus need correction
                jnb    ind.5,next_bsub          ; if 100x, it's <9
```

```
                ;here if 101x, decimal adjust
dadj_msd        stc                                   ; clear effect of carry
                setb   fsr.2                          ; make sure that it's pointing at the result
                sub    ind,#$60                       ; decimal correct


next_bsub       clrb   fsr.2                          ; points to stored carry
                sb     ind.0                          ; skip if not set
                clc                                   ; restore stored carry
                inc    fsr                            ; next byte
                sb     fsr.2                          ; done? (fsr=$0c?)
                jmp    bsub_more                      ; not yet
                ret                                   ; done, return to calling routine

; move the result to operand1 and change operand2 to 0
                ; the intention is prepare for 0-result or getting the magnitude of a
                ; negative BCD number which is in complement form
neg_result      mov    fsr,#operand2                  ; points to
mov_more        setb   fsr.2                          ; operand2
                mov    W,ind                          ; temp. storage
                clr    ind                            ; clear operand2
                clrb   fsr.2                          ; points to operand1
                mov    ind,W                          ; store result
                inc    fsr                            ; next byte
                sb     fsr.2                          ; done?
                jmp    mov_more                       ; no
                ret                                   ; yes, finish
```

4.0 Binary to BCD Conversion

In many situations, we will find BCD representations very difficult to deal with, especially when anything more than addition and subtraction is needed, due to the need for decimal correction. This problem is alleviated by representing the numbers internally as binary to facilitate computation and convert it to BCD for display or printing purposes. In this section, we will discuss how that is implemented.

There are many different algorithms for binary to BCD conversions. We will only consider one of the easiest to implement, that is, shifting the binary number to the left and let the most significant bit be shifted into a BCD result. The result is then continuously decimally corrected to give a right answer.

In the following code segment, we have implemented a 32 bit binary number to 10 digit BCD conversion routine. With the RL instruction of the SX, the shift operation of both numbers together is a breeze.

Decimal correction is done here differently than before. Instead of checking the carry and digit carry, we check the BCD value before a shift and adjust it properly. This will save us both code and time. This was not possible before in our addition and subtraction routines since we were not doing shift operations.

To see how this is done, let's look at some examples:

Current value	Binary	Shifted value in binary	Shifted value in hex	What the shifted value should be in BCD
0	0000	0000	0	0
1	0001	0010	2	2
2	0010	0100	4	4
3	0011	0110	6	6
4	0100	1000	8	8
5	0101	1010	A	10
6	0110	1100	C	12
7	0111	1110	E	14
8	1000	1 0000	10	16
9	1001	1 0010	12	18

From the table, we can see that whenever the current value is 4 or less, then it is okay. For all digits of 5 and above, decimal correction is needed. This can be done by adding 6 to the shifted value or by adding 3 to the current value. If we add 3 to all current values and check if they are greater than 7, all number satisfying this condition will need decimal correction and we will just keep that added number, otherwise we fall back to the original number.

This decimal correction process applies also to the most significant digit, except we use $30 instead of 3.

```
                    ; 32 bit binary to BCD conversion
                    ; entry: 32 bit binary number in $10-13
                    ; exit: 10 digit BCD number in $14-18
                    ; algorithm= shift the bits of binary number into the BCD number and
;                          decimal correct on the way
bindec              mov    count,#32
                    mov    fsr,#bcd_number     ; points to the BCD result

clr_bcd             clr    ind                 ; clear BCD number
                    snb    fsr.3               ; reached $18?
                    jmp    shift_both          ; yes, begin algorithm
                    inc    fsr                 ; no, continue on next byte
                    jmp    clr_bcd             ; loop to clear
```

```
shift_both          mov    fsr,#bin_number     ; points to the binary number input
                    clc                        ; clear carry, prepare for shifting

shift_loop          rl     ind                 ; shift the number left
                    snb    fsr.3               ; reached $18? (finish shifting both
; numbers)
                    jmp    check_adj           ; yes, check if end of everything
                    inc    fsr                 ; no, next byte
                    jmp    shift_loop          ; not yet

check_adj           decsz  count               ; end of 32 bit operation?
                    jmp    bcd_adj             ; no, do bcd adj
                    ret

bcd_adj             mov    fsr,#bcd_number     ; points to first byte of the BCD result

bcd_adj_loop        call   digit_adj           ; decimal adjust
                    snb    fsr.3               ; reached last byte?
                    jmp    shift_both          ; yes, go to shift both number left again
                    inc    fsr                 ; no, next byte
                    jmp    bcd_adj_loop        ; looping for decimal adjust

digit_adj           ; consider LSD first
                    mov    W,#3         ; 3 will become 6 on next shift
                    add    W,ind        ; which is the decimal correct factor to be added
                    mov    temp,W
                    snb    temp.3       ; > 7? if bit 3 not set, then must be <=7, no adj.
                    mov    ind,W        ; yes, decimal adjust needed, so store it

                    ; now for the MSD
                    mov    W,#$30              ; 3 for MSD is $30
                    add    W,ind               ; add for testing
                    mov    temp,W
                    snb    temp.7              ; > 7?
                    mov    ind,W               ; yes, store it

                    ret
```

5.0 BCD to Binary Conversion

Input from keyboards can be easily rendered into BCD form. To let the CPU process the number effectively, however, binary representation is more desirable.

In this section we will discuss how the BCD to binary conversion process is implemented. It is basically a reversal of the binary to BCD conversion process: we shift the BCD number to the right and let the least significant bit be shifted into a binary result. The original BCD number is then continuously decimally corrected to maintain the BCD format.

In the following code segment, we have implemented a 10 digit BCD number to 32 bit binary number conversion routine. With the RR instruction of the SX, the shift operation of both numbers together can be very efficiently implemented.

Decimal correction is done again differently here since we are shifting right instead of shifting left.

To derive the algorithm, let's look at the following table:

Current value	Binary	Shifted value in binary	Shifted value in hex	What the shifted value should be in BCD
0	0000	0000	0	0
2	0010	0001	1	1
4	0100	0010	2	2
6	0110	0011	3	3
8	1000	0100	4	4
10	10000	1000	8	5
12	10010	1001	9	6
14	10100	1010	A	7
16	10110	1011	B	8
18	11000	1100	C	9

As we can see, whenever the shifted value has a 1 on bit 3, the result should be subtracted with 3 to make it correct. And this is the algorithm that we have adopted in the following code: shift right both numbers and decimally adjust the BCD number along the way. Note that for the most significant digit in each BCD number, we subtract $30 instead of 3 to account for its position.

```
             ; 10 digit BCD to 32 bit binary conversion
             ; entry: 10 digit BCD number in $14-18
             ; exit: 32 bit binary number in $10-13
             ; algorithm= shift the bits of BCD number into the binary number and decimal
             ;          correct on the way
decbin       mov    count,#32          ; 32 bit number
             mov    fsr,#bin_number; points to the binary result

clr_bin      clr    ind                ; clear binary number
             inc    fsr                ; no, continue on next byte
             snb    fsr.2              ; reached $13? (then fsr will be $14 here)
             jmp    shift_b            ; yes, begin algorithm
             jmp    clr_bin            ; loop to clear

shift_b      mov    fsr,#bcd_number+4  ; points to the last BCD number
             clc                       ; clear carry, prepare for shifting right

shft_loop    rr     ind                ; shift the number right
             dec    fsr                ; reached $10? (finish shifting both numbers)
             sb     fsr.4              ; then fsr will be $0f
```

```
                jmp    chk_adj             ; yes, check if end of everything
                jmp    shft_loop           ; not yet

chk_adj         decsz  count               ; end of 32 bit operation?
                jmp    bd_adj              ; no, do bcd adj
                ret

bd_adj          mov    fsr,#bcd_number     ; points to first byte of the BCD result

bd_adj_loop     call   dgt_adj             ; decimal adjust
                snb    fsr.3               ; reached last byte?
                jmp    shift_b             ; yes, go to shift both number right again
                inc    fsr                 ; no, next byte
                jmp    bd_adj_loop         ; looping for decimal adjust

                ; prepare for next shift right
                ; 0000  --> 00000 -->0
                ; 0010 --> 0001      2 -->1
                ; 0100 --> 0010      4 -->2
                ; 0110 --> 0011      6 -->3
                ; 1000 --> 0100      8 -->4
                ; 1 0000 --> 1000   10-->8 !! it should be 5, so -3
                ; 1 0010 --> 1001   12-->9 !! it should be 6, so -3
                ; in general when the highest bit in a nibble is 1, it should be subtracted with 3
dgt_adj         ; consider LSD first
                sb     ind.3               ; check highest bit in LSD, =1?
                jmp    ck_msd              ; no, check MSD
                stc                        ; prepare for subtraction, no borrow
                sub    ind,#3              ; yes, adjust

                ; now for the MSD
ck_msd          sb     ind.7               ; highest bit in MSD, =1?
                ret                        ; no

                ; yes, do correction
                stc                        ; no borrow
                sub    ind,#$30            ; this is a 2 word instruction, and cannot be skipped
                ret
```

6.0 Multiplication

The need for multiplication permeates through the use of microcontrollers. Here we will consider both 8 bit by 8 bit and 16 bit by 16 bit multiplications. As we can see, the basic algorithms are all the same regardless of the number of bits involved.

Let's first discuss how the multiplier, multiplicand, and the result are generally organized.

Multiplicand

Upper product	Multiplier (lower product)

The lower part of the result are initially occupied by the multiplier and the upper part is cleared to zero.

To summarize, the following steps are needed to do a multiplication by software:

- Initialize multiplier, multiplicand from calling program;
- clear the upper product to zero;
- shift right the whole product to the right;
- if carry is 1, i.e., the lsb of the multiplier is one, then add the multiplicand to the upper product;
- repeat step 3 and 4 until all bits of the multiplier has been shifted out

This algorithm is amazingly elegant as we can see in the next program segment.

As implemented for 8 bit by 8 bit multiplication, this routine requires only 2 bytes of RAM provided the multiplicand is pre-loaded into the W, working register.

```
             ; 8 bit x 8 bit multiplication (RAM efficient, 2 bytes only)
             ; entry: multiplicand in W, multiplier at 09
             ; exit : product at $0a,09
                                          ; cycles
mul88        mov    upper_prdt,W          ; 1    store W
             mov    count,#9              ; 2    set number of times to shift
             mov    W,upper_prdt          ; 1    restore W (multiplicand)
             clr    upper_prdt            ; 1    clear upper product
             clc                          ; 1    clear carry
                                          ; the following are executed [count] times
m88loop      rr     upper_prdt            ; 1    rotate right the whole product
             rr     multiplier            ; 1    check lsb
             snc                          ; 1    skip addition if no carry
             add    upper_prdt,W          ; 1    add multiplicand to upper product
no_add       decsz  count                 ; 1/2  loop 9 times to get proper product
             jmp    m88loop               ; 3    jmp to rotate the next half of product

             ret                          ; 3    done...
                                   ; one time instructions = 1+2+1+1+1+3= 9 cycles
                                   ; repetitive ones= (1+1+1+1+1+3)9-3+2=71
                                   ; total worst case cycles=80 cycles
```

A faster implementation can be obtained if we unroll the loop and repeat the code using a macro:

```
             ; fast 8 bit x 8 bit multiplication (RAM efficient, 2 bytes only)
             ; entry: multiplicand in W, multiplier at 09
             ; exit : product at $0a,09

             ; macro to rotate product right and add
rra          MACRO
             rr     upper_prdt            ; 1    rotate right the whole product
             rr     multiplier            ; 1    check lsb
             snc                          ; 1    skip addition if no carry
             add    upper_prdt,W          ; 1    add multiplicand to upper product
             ENDM
                                          ; cycles
fmul88       clr    upper_prdt            ; 1    clear upper product
             clc                          ; 1    clear carry
                                          ; the following are executed [count] times
```

```
            rra                         ; call the macro 9 times
            rra
            rra
            rra
            rra
            rra
            rra
            rra
            rra

            ret                         ; 3    done...
                                        ; one time instructions = 1+1+3= 5 cycles
                                        ; repetitive ones= (1+1+1+1+1)9=36
                                        ; total worst case cycles=41 cycles
```

We have saved almost half of the time by using macros and eliminating the loop control. Notice that in both algorithms, 9 shifts are needed to obtain a correct result. The last shift is used to align the result properly.

The same algorithm has been implemented for 16 bit by 16 bit multiplication, which is included as follows:

```
            ; 16 bit x 16 bit multiplication
            ; entry: multiplicand in $09,08, multiplier at $0b,$0a
            ; exit : 32 bit product at $0d,$0c,$b,$a
                                        ; cycles
mul1616
            mov    count,#17            ; 2    set number of times to shift
            clr    upper_prdt           ; 1    clear upper product
            clr    upper_prdt+1         ; 1    higher byte of the 16 bit upper product
            clc                         ; 1    clear carry
                                        ; the following are executed [count] times
m1616loop   rr     upper_prdt+1         ; 1    rotate right the whole product
            rr     upper_prdt           ; 1    lower byte of the 16 bit upper product
            rr     mr16+1               ; 1    high byte of the multiplier
            rr     mr16                 ; 1    check lsb
            sc                          ; 1    skip addition if no carry
            jmp    no_add               ; 3     no addition since lsb=0
            clc                         ; 1    clear carry
            add    upper_prdt,md16      ; 1    add multiplicand to upper product
            add    upper_prdt+1,md16+1 ; 1    add the next 16 bit of multiplicand
no_add      decsz  count                ; 1/2  loop [count] times to get proper product
            jmp    m1616loop            ; 3    jmp to rotate the next half of product

            ret                         ; 3    done...
                                        ; one time instructions = 8 cycles
                                        ; repetitive ones= 15*16+11+2=253
                                        ; total worst case cycles=261 cycles
```

Note that the only difference is the number of bits that we shift, and more bytes to add and rotate. Other than that, it is basically the same as a 8 x 8 multiplication. A fast version is also available but it is too lengthy to list here. Please see the program file for details. A saving of 26% is achieved here by unrolling the loop and reduced the cycles to 193.

7.0 Division

Finally, we are going to tackle the most difficult arithmetic problem: that of division. If the reader can recall how he or she was taught how to do division by long hand, then we are very close to understanding the algorithm.

In division by long hand, we examine the dividend digit by digit, and see if it is bigger than the divisor. If it is, then we subtract the divisor or the multiples of it from the dividend and write down that multiple as a digit in our quotient. This process is repeated until all digits of the dividend are exhausted.

This exact process is being implemented in the following code segment with one difference with our long hand division: we are dealing with binary numbers here. So we modify the algorithm as follows:

- initialize the result and remainder register;
- shift the dividend bit by bit into the remainder register (use as a placeholder here);
- do a trial subtraction of the partial dividend in the remainder register and the divisor;
- if the partial dividend is bigger than the divisor, then we subtract the divisor from it and record a 1 bit for the quotient
- shift the quotient to left so that we can calculate the next bit, and repeat step 2 thru 4 till all bits of the dividend is exhausted.

```
             ; 16 bit by 16 bit division (b/a)
             ; entry: 16 bit b, 16 bit a
             ; exit : result in b, remainder in remainder
                                         ; cycles
div1616      mov    count,#16            ; 2    no. of time to shift
             mov    d,b                  ; 2    move b to make space
             mov    d+1,b+1              ; 2    for result
             clr    b                    ; 1    clear the result fields
             clr    b+1                  ; 1    one more byte
             clr    rlo                  ; 1    clear remainder low byte
             clr    rhi                  ; 1    clear remainder high byte
                                         ; subtotal=10
divloop      clc                         ; 1    clear carry before shift
             rl     d                    ; 1    check the dividend
             rl     d+1                  ; 1    bit by bit
             rl     rlo                  ; 1    put it in the remainder for
             rl     rhi                  ; 1    trial subtraction
                                         ; subtotal=5
             stc                         ; 1    prepare for subtraction, no borrow
             mov    W,a+1                ; 1    do trial subtraction
             mov    W,rhi-W              ; 1    from MSB first
             sz                          ; 1/2  if two MSB equal, need to check LSB
             jmp    chk_carry            ; 3    not equal, check which one is bigger
                                         ;
             ; if we are here, then z=1, so c must be 1 too, since there is no
             ; underflow, so we save a stc instruction

             mov    W,a                  ; 1    equal MSB, check LSB
             mov    W,rlo-W              ; 1    which one is bigger?
                                         ; subtotal=7
chk_carry    sc                          ; 1/2  partial dividend >a?
             jmp    shft_quot            ; 3    no, partial dividend < a, set a 0 into quotient

             ; if we are here, then c must be 1, again, we save another stc instruction

                                         ; yes, part. dividend > a, subtract a from it
             sub    rlo,a                ; 2    store part. dividend-a into a
             sub    rhi,a+1              ; 2    2 bytes
             stc                         ; 1    shift a 1 into quotient
                                         ; subtotal=7 worst case
shft_quot    rl     b                    ; 1    store into result
             rl     b+1                  ; 1    16 bit result, thus 2 rotates
```

```
        decsz  count              ; 1/2
        jmp    divloop            ; 3
                                  ; subtotal=6, 4 on last count
        ret                       ; 3
                                  ; one time instructions=13
                                  ; repetitive ones=(19+6)*15+19+4=398
                                  ; total=411
```

The fast version of this division algorithm is implemented by unrolling the loop and repeat all the instructions inside it. It consumes 336 cycles and therefore saves 18% of time

8.0 Conclusions

The SX instructions, namely, ADD (add), ADDB (add bit), SUB (subtract), SUBB (subtract bit), CLC (clear carry), STC (set carry), RL (rotate left 1 bit), RR (rotate right 1 bit), are very useful in implementing arithmetic routines. With careful planning and smart algorithm design, all normal arithmetic functions can be accomplished.

To test the example programs, remember to set the equate options mentioned in the first sentence of the program listing properly (for example, to use BCD routines, set **bcd_test equ 1** and reset all other options to 0). This will enable you to include only the code you need in a program.

9.0 Modifications and further options

There are plenty of literature on computer arithmetic and the implementations included in this application note is not the only way of doing it. It only serves as an example for the readers and help them to bring their product to the market faster by using existing routines.

Lit#: SXL-AN13-01