Signed Multi-Byte Multiplication in PSoC® 1
Assembler

AN2038

Author: Dave Van Ess
Associated Project: Yes
Associated Part Family: All PSoC 1 Parts
Software Version: PSoC Designer 5.1 SP1
Associated Application Notes: AN2032, AN2341, AN2384

Application Note Abstract
AN2038 describes how to perform the multiplication of multi-byte signed numbers in PSoC® 1 assembly code. Multiplying numbers is a fundamental mathematical operation. PSoC 1 contains a dedicated multiply/accumulate block that is capable of multiplying two signed single byte operands. If multiple bytes are to be multiplied, some level of CPU interaction is necessary. This Application Note will show that with a little understanding of the interaction between signed and unsigned values, fast signed multi-byte multiplication algorithms can easily be developed.

Introduction
Publication of Application Note AN2032 - Unsigned Multiplication generated a lot of interest with users. However, many requested similar information for signed multi byte multiplication. Please view this application note as a continuation of AN2032. It is recommended that you read AN2032 first.

The interaction between signed and unsigned values is discussed and techniques will be shown that allow the PSoC MCU to quickly multiply multiple-byte signed multiplicands and give a full resolution signed output. Software to multiply two 16-bit signed values will be developed.

This application uses the following PSoC microcontroller resources: MAC (Multiply/Accumulate)

Part Signed, Part Unsigned
The two bytes that make up a 16-bit signed word are a combination of a signed upper byte and an unsigned lower byte. Equation 1 and Equation 2 show how these multiplicands are represented:

\[
lu \ x + = 256^{16} \ y
\]

Equation 1

\[
uy \ y + = 256^{16} \ y
\]

Equation 2

Equation 3 and Equation 4 show that the multiplication of two, 16-bit values can be expressed as four multiplications of two 8-bit values.

\[
(lu \ x)(lu \ y) = 256^{16}(lu \ x + y) = 256^{16}(lu \ x + y)
\]

Equation 3

\[
(uy \ y)(uy \ y) = 256^{16}(uy \ y + y) = 256^{16}(uy \ y + y)
\]

Equation 4

Of the four multiplications:
- One is a signed byte times signed byte operation.
- Two are signed byte times unsigned byte operations.
- One is an unsigned byte times unsigned byte operation.

The first is already in the form the multiplier requires. The third has already been covered in Application Note...
This just leaves the case of a signed byte and unsigned byte multiplication.

The difference between signed and unsigned values is how the most significant bit is interpreted. In the two's complement system, the most significant bit represents the sign of the number. Because of the nature of the two's complement expression of numbers in binary, signed values may be converted to unsigned values using simple logic, as shown in Equation 5.

\[
x_{\text{unsigned}} = x_{\text{signed}} + 256 f(x) \quad f(x) = \begin{cases} 1 & x_{\text{bit}7} = 1 \\ 0 & x_{\text{bit}7} = 0 \end{cases}
\]

Equation 5

Equation 6 shows that the multiplication of a signed and an unsigned byte yields a signed result.

\[
\text{Result}_{\text{signed}} = y_{\text{signed}} x_{\text{unsigned}}
\]

Equation 6

Substituting Equations 6 into Equation 5 produces an expanded Equation 7.

\[
\text{Result}_{\text{signed}} = 256 f(x) y_{\text{signed}} + x_{\text{signed}} y_{\text{signed}}
\]

Equation 7

As stated in Equation 6, multiplication of signed byte and unsigned byte produces a signed 2-byte result. Parsing Equation 7 yields the following results:

- \([x_{\text{signed}} y_{\text{signed}}]\) is the 2-byte output of the signed multiplier.
- \([256f(y)x_{\text{signed}}]\) has the effect of adding \(x_{\text{signed}}\) to the upper byte of the result if \(y_{\text{bit}7}\) is 1.

Equation 8 and Equation 9 put this together to show how the resultant upper byte and lower byte for an unsigned multiply are calculated:

\[
\text{Result}_{\text{signed}} = \text{MUL } _{\text{DH}} + 256 f(x) y_{\text{signed}} \quad \text{Equation 8}
\]

\[
\text{Result}_{\text{unsigned}} = \text{MUL } _{\text{DL}} \quad \text{Equation 9}
\]

For either signed or unsigned operands, the lower byte of the resultant is always the same.

The assembly macro shown in Code 1 calculates the signed MSB of the multiplication of an unsigned value \(x\) and of a signed value \(y\):

**Code 1. GetXuYsMSB Assembly Macro**

```assembly
macro GetXuYsMSB
; @0 @1 & MUL_DH determine value
    mov A,reg[MUL_DH]
    tst [01],80h
    jz .+4 ; add A,[00] ;endif
Endm
```

To simplify matters, the macro shown in Code 2 calculates the signed MSB of the multiplication of a signed value \(x\) and of an unsigned value \(y\):

**Code 2. GetXsYuMSB Assembly Macro**

```assembly
macro GetXsYuMSB
; @0 @1 & MUL_DH determine value
    mov A,reg[MUL_DH]
    tst [01],80h
    jz .+4 ;
    add A,[00] ;endif
Endm
```

These macros are found in “signedmath.inc,” located in the project file associated with this Application Note. These routines are presented as macros to simplify the explanation of the algorithm development. Conversion of these macros to subroutines or to distinct code is left as an exercise for the reader.

### 16-Bit Signed Multiplication

If two signed 2-byte operands are multiplied together the result is a signed, 4-byte answer. Figure 2 shows that this multiplication can be expressed as a combination of four smaller 8-bit, unsigned multiplications. For more information on Napier Matrices, see AN2032.

**Figure 2. Napier Matrix for 16-Bit Signed Multiply**

The following four-step algorithm illustrates four, smaller 8-bit unsigned multiplications. Each step's number corresponds to a position in the Napier Matrix, as labeled in Figure 2.

1. Multiply \(X_{\text{Val}}\) and signed byte \(Y_{\text{Val}}\). Place the lower byte of the result in \(\text{Result}_{+1u}\) and the upper byte in \(\text{Results}\).
2. Move down the matrix and multiply \(X_{\text{Val}}\) and \(Y_{\text{Val}+1u}\). Place the lower resultant byte in \(\text{Result}_{+2u}\), and add the upper resultant byte with \(\text{Result}_{+1u}\). Because the upper result is signed but the storage is unsigned, the sign must be extended in to \(\text{Results}\).
3. Move right and multiply \(X_{\text{Val}+1u}\) and \(Y_{\text{Val}+1u}\). Place the lower resultant byte in \(\text{Result}_{+3u}\) and add the upper resultant byte to \(\text{Result}_{+2u}\). As in step 2, a negative result must be sign-extended in to \(\text{Result}_{+1u}\).
4. Move up the matrix and multiply \(X_{\text{Val}+1u}\) and \(Y_{\text{Val}}\). Place the lower byte to \(\text{Result}_{+2u}\) and the upper byte into \(\text{Result}_{+1u}\).
The macro shown in Code 3 implements this algorithm. Macros internal to it can be found in "signedmath.inc" and "unsignedmath.inc," located in the project file associated with this Application Note.

Code 3. Multiply32s_16s_16s Assembly Macro

macro Multiply32s_16s_16s
    ; result = XVal * YVal
    ; @0 = @1 * @1
    ; 16 bit by 16 bit signed multiply
    ; with 32 bit signed result
    ;
    ; (1)
    Multiply16s_8s_8s (@0), (@1), (@2)
    ; (2)
    PushMulY (@2 + 1)
    GetXsYuMSB (@1), (@2 + 1)
    cmp A, 128
    jc . + 4 ; pass on carry
    dec [@0]
    add [@0], A
    GetLSB
    mov [@0 + 2], A
    ; (3)
    PushMulX (@1 + 1)
    GetUnsignedMSB (@1 + 1), (@2 + 1)
    add [@0 + 2], A
    adc [@0 + 1], 0
    GetLSB
    mov [@0 + 3], A
    ; (4)
    PushMulY (@2)
    GetXuYsMSB, (@1 + 1), (@2)
    push A
    cmp A, 128
    jc . + 4
    dec [@0]
    GetLSB
    add [@0 + 2], A
    pop A
    adc [@0 + 1], A
    adc [@0], 0
endm

Code 4 is a program to exercise this macro. This code is also shown in the associated example project. Two 16-bit signed operands (XVal and YVal) are multiplied and a 32-bit resultant (Result) is calculated.

Code 4. Signed Multiplication Test Assembly Code

;------------------------------------------
; Program to test the "signedmath.inc"
macros
;------------------------------------------
export Result
export XVal
export YVal
area bss(RAM)
    Result:    BLK  4 ; 32 bit signed
    XVal:      BLK  2 ; 16 bit signed operand
    YVal:      BLK  2 ; 16 bit signed operand
area text(ROM,REL)
    _main:
        ; initialize X and Y values
        mov [XVal+0], ffh ; MSB of XVal
        mov [XVal+1], fdh ; LSB of XVal, XVal = -3
        mov [YVal+0], ffh ; MSB of YVal
        mov [YVal+1], feh ; LSB of YVal, YVal = -2
    _loop:
        Multiply32s_16s_16s Result, XVal, YVal
        nop ; good place to halt and view data
        jmp _loop

When this program is run on a debugger and halted at the end of calculation, the contents of RAM will show that the “Result” is -1 or 1. The time it takes to perform the retrieval of data, the actual multiply, and place the signed answer in memory is 293 CPU cycles. For a CPU speed of 24 MHz this works out to 12.2 usec.

Summary

A signed multiplier does not limit a user to single byte-signed math. An understanding of the differences between signed and unsigned values makes fast, multi-byte multiplication possible. A two-byte example (Multiply32s_16s_16s) has been shown. This operation takes 293 CPU cycles to execute, and requires 99 bytes of Flash storage for code.

The ideas discussed herein will allow the user to develop their own multiplication routines customized for their particular data-processing requirements.
In March of 2007, Cypress recataloged all of its Application Notes using a new documentation number and revision code. This new documentation number and revision code (001-xxxx, beginning with rev. **), located in the footer of the document, will be used in all subsequent revisions.

PSOC is a registered trademark of Cypress Semiconductor Corp. "Programmable System-on-Chip," PSOC Designer, and PSOC Express are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

© Cypress Semiconductor Corporation, 2002-2011. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress’ product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.