CYPRESS MICROSYSTEMS

# Application Note AN2032

## *Unsigned Multiplication*

**By**: Dave Van Ess
**Associated Project**: *unsignedmath.zip*
**Associated Part Family**: CY8C25xxx, CY8C26xxx

## Summary
A built in MAC (Multiply/Accumulate) assists the PSoC™ MCU with digital signal-processing applications. But what if unsigned multiplication is needed? The MAC uses data in signed, 2's-complement format. This Application Note will show that with a little understanding of signed vs. unsigned values, fast unsigned multiplication algorithms can easily be developed.

## Introduction
Without dedicated hardware, performing multiplication can be a tedious task. Any dedicated hardware multiplier is going to have finite resolution and applications may arise where more resolution is required. Techniques will be shown that allow the PSoC MCU to quickly multiply multiple-byte-operands and give a full resolution output.

This application uses the following PSoC microcontroller resources:

- MAC (Multiply/Accumulate)

## The PSOC Multiplier
Figure 1 shows that four registers are used to access the multiplier.

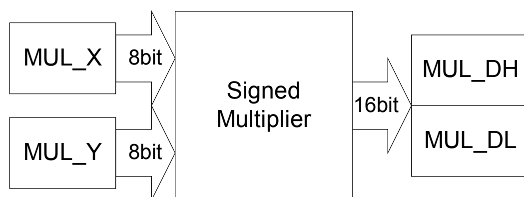


**Figure 1: PSoC Multiplier Block Diagram**

These four registers are all located in register Bank 0, having the following names and addresses:

- MUL_X (Bank 0 E9h)
- MUL_Y (Bank 0 E8h)
- MUL_DH (Bank 0 EAh)
- MUL_DL (Bank 0 EBh)

MUL_X and MUL_Y are "write only" registers where signed, 2's-complement 8-bit values are input to the multiplier. The signed, 2's-complement 16-bit output is deposited into two 8-bit "read only" registers, MUL_DH (upper) and MUL_DL (lower). The program code below is a simple example to multiply the contents of XVal (-1) and YVal (-1) and deposit the answer in "Result."

```
;-------------------------------------------
;  This program places the contents of "XVal"
;  and "Yval" into the multiplier. The answer
;  is then moved to "Result"
;---------------------------------------------
export _main
include "m8c.inc"
export Result
export XVal
export YVal
area bss(RAM)
   Result:    BLK  2   ; 16 bit
   XVal:      BLK  1   ; 8 bit
   YVal:      BLK  1   ; 8 bit
area text(ROM,REL)
_main:
   mov [XVal],-1       ; initialize data
   mov [YVal],-1

   mov A,[XVal]
   mov reg[MUL_X],A
   mov A,[YVal]
   mov reg[MUL_Y],A
   mov A,reg[MUL_DL]
   mov [Result + 1],A
   mov A,reg[MUL_DH]
   mov [Result],A
   loop:               ; done!
      jmp loop
```

**Code 1**

When Code 1 is run on the ICE and halted at the end of the calculation, the contents of RAM will show that "Result" is 1.

## Some Macros Please!

There are going to be a lot of examples where data is pushed into and retrieved from the multiplier. The following macros will eliminate some of the repetitive tedium. All these macros can be found in *unsignedmath.inc*, located in the project file associated with this Application Note.

Macro to fill MUL_X:

```
macro PushMulX
;  reg[MUL_X] = @0
   mov A,[@0]
   mov reg[MUL_X],A
endm
```

Macro to fill MUL_Y:

```
macro PushMulY
;  reg[MUL_Y] = @0
   mov A,[@0]
   mov reg[MUL_Y],A
endm
```

Macro to retrieve the lower byte of the answer:

```
macro GetLSB
   mov A,reg[MUL_DL]
endm
```

Macro to retrieve the upper byte:

```
macro GetSignedMSB
   mov A,reg[MUL_DH]
endm
```

Macro that ties it all together:

```
macro Multiply16s_8s_8s
;  @0 = @1 * @2
;  multiplies two 8-bit, signed values and returns
;  a 16-bit, signed value.
   PushMulX @1
   PushMulY @2
   GetLSB
   mov [@0 + 1],A
   GetSignedMSB
   mov [@0],A
endm
```

Code 1 program code example has been rewritten to utilize these new macros and is shown below in Code 2:

```
;-------------------------------------------
;  This program places the contents of “XVal”
;  and “Yval” into the multiplier. The answer
;  is then moved to “Result”
;--------------------------------------------
export _main
include "m8c.inc"
include “unsignedmath.inc”

export Result
export XVal
export YVal

area bss(RAM)
   Result:    BLK  2   ; 16 bit
   XVal:      BLK  1   ; 8 bit
   YVal:      BLK  1   ; 8 bit
area text(ROM,REL)
_main:
   mov [XVal],-1       ; initialize data
   mov [YVal],-1

   Multiply16s_8s_8s Result, XVal, YVal
   loop:               ; done!
      jmp loop
```

**Code 2**

It takes 42 CPU cycles to retrieve data, perform the multiply operation, and place the answer in memory. For a CPU speed of 24 MHz this works out to 1.75 usec. Signed, 2’s-complement multiplies can quickly be done. Now, on to unsigned, non-2’s complement math header.

## 2’s C or Not 2’s C

That is the question.
Whether ‘tis nobler as to bind in common
With ones and zeros of lesser potential,
Or to take odds against a sum of minors,
And by opposing end them?

This is just a more Elizabethan way of asking, “*What makes a signed value differ from an unsigned value*?”

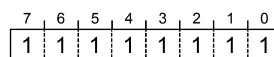


**Figure 2: A Byte All Filled With Ones**

If asked the value of the byte in Figure 2, the unsigned camp would say **255** while the signed camp would say **-1**. Both answers are correct. It depends on how the byte’s contents are interpreted.

The on-board MAC interprets its inputs as signed, 8-bit integers. Performing unsigned multiplication requires some extra work.

Equation (1) defines the relationship between unsigned and signed values:

$$x_{unsigned} = x_{signed} + 256 f(x) \quad f(x) = \begin{cases} 1 & x_{bit7} = 1 \\ 0 & x_{bit7} = 0 \end{cases} \quad (1)$$

Applying Equation (1) to the contents of the byte in Figure 2 shows that it is valid.

$$255_{unsigned} = -1_{signed} + 256 \quad f(-1_{signed}) = 1$$

Multiplying two single, unsigned operands produces an unsigned result:

$$R_{unsigned} = x_{unsigned} y_{unsigned} \quad (2)$$

Using Equation (1) to substitute for the unsigned terms in Equation (2), results in Equation (3):

$$R_{unsigned} = (x_{signed} + 256 f(x))(y_{signed} + 256 f(y)) \quad (3)$$

Expanding Equation (3) produces a more decipherable Equation (4):

$$R_{unsigned} = \begin{array}{c} 65536\, f(x) f(y) + \\ 256\, f(x) y_{signed} + \\ 256\, f(y) x_{signed} + \\ x_{signed}\, y_{signed} \end{array} \quad (4)$$

As stated in Equation (1), multiplication of two unsigned bytes produces an unsigned 2-byte result. Parsing Equation (4) yields the following results:

- [$x_{signed}y_{signed}$] is the 2-byte output of the signed multiplier.
- [256$f$(y)$x_{signed}$] has the effect of adding $x_{signed}$ to the upper byte of the result if $y_{bit7}$ is 1.
- [256$f$(x)$y_{signed}$] has the effect of adding $y_{signed}$ to the upper byte of the result if $x_{bit7}$ is 1.
- [65536$f$(x)$f$(y)] has a value of either zero or it falls outside the range of the result. Either way, it can be ignored.

Equation (5) puts this together to show how the resultant upper byte and lower byte for an unsigned multiply are calculated:

$$\begin{aligned} (R_{unsigned})_H &= MUL_DH + f(x)y_{signed} + f(y)x_{signed} \\ (R_{unsigned})_L &= MUL_DL \end{aligned} \quad (5)$$

**Note**: The lower byte resultant of an unsigned multiply is equal to the lower byte resultant of a signed multiply. **If the results of any multiplication are truncated to have the same resolution as the inputs, there is no difference between signed and unsigned multiplication**.

The segment code below calculates the unsigned MSB byte of an unsigned multiply:

```
;-----------------------------------
;
;  Code to get an unsigned MBS in A
;
;-----------------------------------
mov A,reg[MUL_DH]      ; get the data
tst [XVal],80h
jz skip_y
   add A,[YVal]
skip_y:
tst [YVal],80h
jz skip_x
   add A,[XVal]
skip_x:
```

**Code 3**

The following macro performs the same function:

```
macro GetUnsignedMSB
; @0 @1 & MUL_DH determine value
   mov A,reg[MUL_DH]
   tst [@0],80h
   jz .+4 ;(skip_y)
      add A,[@1]
;  skip_y
   tst [@1],80
   jz .+4 ;(skip_x)
      add A,[@0]
;  skip_x
endm
```

The following macro multiplies unsigned bytes and stores the resultant answer in memory:

```
macro Multiply16u_8u_8u
;  @0 = @1 * @2
;  multiplies two 8-bit, unsigned values and returns
;  a 16-bit, unsigned value.
   PushMulX @1
   PushMulY @2
   GetLSB
   mov [@0 + 1],A
   GetUnsignedMSB @1, @2
   mov [@0],A
endm
```

Code 4 below is similar to Code 3 only this time the unsigned resultant is stored:

```
;-----------------------------------------
;  This program places the contents of "XVal"
;  and "Yval" into the multiplier. The unsigned
;  answer is then moved to "Result"
;-------------------------------------------
export _main
include "m8c.inc"
include "unsignedmath.inc"
export Result
export XVal
export YVal
area bss(RAM)
   Result:    BLK  2   ; 16 bit
   XVal:      BLK  1   ; 8 bit
   YVal:      BLK  1   ; 8 bit
area text(ROM,REL)
_main:
   mov [XVal],ffh      ; initialize data
   mov [YVal],ffh

   Multiply16u_8u_8u Result, XVal, YVal
   loop:               ; done!
      jmp loop
```

**Code 4**

When this program is run on the ICE and halted at the end of calculation, the contents of RAM will show that the "Result" is **255*255 or FE01h**. The time it takes to perform the retrieval of data, the actual multiply, and placement of the unsigned answer in memory is 80 CPU cycles. For a CPU speed of 24 MHz this works out to 3.33 usec.

## Multiple Byte Multiplication

If two unsigned, 2-byte operands are multiplied together the result is an unsigned, 4-byte answer. Figure 3 shows that this multiplication is just the combination of four smaller 8-bit, unsigned multiplications.

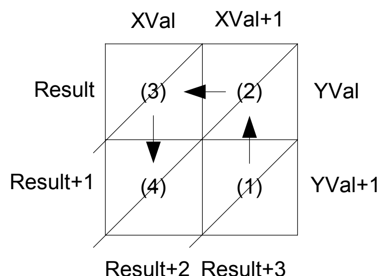


**Figure 3: Napier Matrix for 16-Bit Multiply**

The following four-step algorithm illustrates four, smaller 8-bit unsigned multiplications:

1. Unsigned multiply of XVal+1 and YVal+1 with the lower byte resultant stored in Result+2 and the upper byte stored inResult+3.
2. Move up and multiply XVal+1 and YVal and place the upper resultant byte in Result+1, and add the lower resultant byte with Result+2. By moving only one space horizontally each time, only one multiplicand needs to be reloaded, cutting the overhead of loading the multiplier nearly in half.
3. Move left, multiply XVal and YVal, store the upper resultant byte in Result, and add the lower resultant byte with Result+1.
4. Move down, multiply XVal and YVAL+1, add the upper resultant byte to Result+1, and then add the lower resultant byte to Result+2.

The following macro implements this algorithm:

```
macro Multiply32u_16u_16u
;  result = XVal * YVal
;  @0 = @1 * @1
;  16bit by 16 bit unsigned multiply
;  with 32 bit unsigned result
;
;  (1)
   Multiply16u_8u_8u (@0 + 2),(@1 +1),(@2 +1)
;  (2)
   PushMulY  @2
   GetUnsignedMSB  (@1 +1),@2
   mov [@0 +1],A
   GetLSB
   add [@0 + 2],A
   adc [@0 + 1],0
;  (3)
   PushMulX @1
   GetUnsignedMSB @1,@2
   mov [@0],A
   GetLSB
   add [@0 +1 ],A
   adc [@0],0
;  (4)
   PushMulY (@2 + 1)
   GetUnsignedMSB, @1,(@2 + 1)
   push A
   GetLSB
   add [@0 + 2],A
   pop A
   adc [@0 +1],A
   adc [@0],0
 endm
```

Code 5 is similar to Code 4 only this time, 16-bit, unsigned values are multiplied and the stored unsigned resultant is 32 bits:

```
;------------------------------------------
;  This program places the contents of "XVal"
;  and "Yval" into the multiplier. The unsigned
;  answer is then moved to "Result"
;--------------------------------------------
export _main
include "m8c.inc"
include "unsignedmath.inc"

export Result
export XVal
export YVal

area bss(RAM)
   Result:    BLK  4   ; 32 bit
   XVal:      BLK  2   ; 16 bit
   YVal:      BLK  2   ; 16 bit
area text(ROM,REL)
_main:
   mov [XVal],-1       ; initialize data
   mov [XVal+1],-1
   mov [YVal],-1

   Multiply32u_16u_16u Result, XVal, YVal
   loop:               ; done!
      jmp loop
```

**Code 5**

When this program is run on the ICE and halted at the end of the calculation, the contents of RAM will show that the "Result" is **FFFE0001h**. The time it takes to perform the retrieval of data, the actual multiply, and placement of the unsigned answer in memory is 333 CPU cycles. For a CPU speed of 24 MHz this works out to 13.8 usec.

## Truncated 16 Bit, A Valid Subset

As stated earlier, if the results of any multiplication are truncated to have the same resolution as the inputs, there is no difference between signed and unsigned multiplication. Many times the result of a 16-bit multiply is truncated to just 16 bits. In this case, unsigned calculations yield valid results. Figure 4 shows a subset of the matrix in Figure 3.

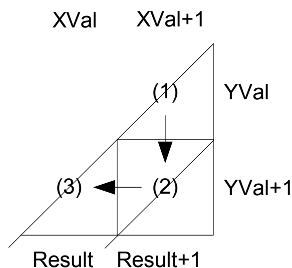


**Figure 4: Matrix for 16-Bit Multiply, Truncated Result**

Because only the lower two bytes of the result is needed, less than half of the full calculation is required. Again, it does not matter if the inputs are signed or unsigned.

This macro implements the Napier Matrix in Figure 4:

```
macro Multiply16_16_16
;  result = XVal * YVal
;  @0 = @1 * @1
;  16 bit by 16-bit multiply
;  with a 16-bit result

;  (1)x1y0  r1
   PushMulY  (@2 +0)
   PushMulX  (@1 +1)
   GetLSB
   mov [@0],A
;  (2)x1y1 r01
   PushMulY (@2 +1)
   GetLSB
   mov [@0 +1],A
   GetUnsignedMSB  (@1 +1), (@2 +1)
   add [@0 +0],A
;  (3)x0y1 r1
   PushMulX (@1 +0)
   GetLSB
   add [@0 +0],A
endm
```

Code 6 is similar to Code 5 only this time 16-bit values are multiplied and the stored resultant is 16 bits:

```
;-------------------------------------------
;  This program places the contents of “XVal”
;  and “Yval” into the multiplier. The unsigned
;  answer is then moved to “Result”
;---------------------------------------------
export _main
include "m8c.inc"
include “unsignedmath.inc”
export Result
export XVal
export YVal
area bss(RAM)
   Result:    BLK  2   ; 32 bit
   XVal:      BLK  2   ; 16 bit
   YVal:      BLK  2   ; 16 bit
area text(ROM,REL)
_main:
   mov [XVal],-1       ; initialize data
   mov [XVal+1],-1
   mov [YVal],-1

   Multiply16_16_16 Result, XVal, YVal
   loop:               ; done!
      jmp loop
```

**Code 6**

When Code 6 is run on the ICE and halted at the end of the calculation, the contents of RAM will show that the “Result” is 1. The time it takes to perform the retrieval of data, the actual multiply, and placement of the unsigned answer in memory is 115 CPU cycles. For a CPU speed of 24 MHz this works out to 4.8 usec.

## 24-Bit Operands, 48-Bit Result

No, this is not just showing off. There is, indeed, a valid reason to implement a 24-bit, unsigned multiplier. Floating-point numbers store their mantissa as a 24-bit, unsigned value. A 24-bit multiplier would go most of the way to making a floating-point multiplication routine. All that is left to deal with is the sign, exponent adjustment, zero, infinity, NAN, etc. Figure 5 shows the Napier Matrix for a 24-bit, unsigned multiply with a 48-bit, unsigned result.

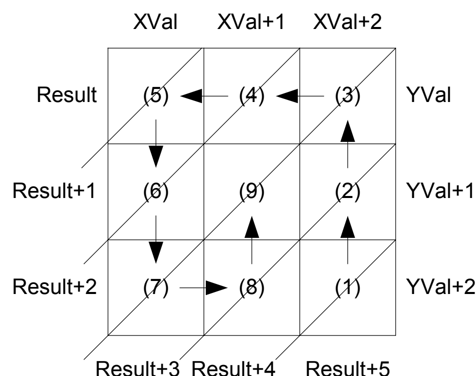


**Figure 5: Napier Matrix for 24-Bit Multiply**

Following is macro code example for 24-bit, unsigned multiply with a 48-bit, unsigned result:

```
macro Multiply48u_24u_24u
;  result = Xval * Yval
;    @0 = @1 * @2
;   24 bit by 24-bit, unsigned multiply
;   with a 48-bit, unsigned result
;

;  (1) x2 y2 r45
   Multiply16u_8u_8u (@0 + 4),(@1 +2),(@2 +2)
;  (2) x2 y1 r34
   PushMulY  (@2 +1)
   GetUnsignedMSB  (@1 +2), (@2 +1)
   mov [@0 +3],A
   GetLSB
   add [@0 + 4],A
   adc [@0 + 3],0
;  (3) x2y0 r23
   PushMulY  (@2 +0)
   GetUnsignedMSB  (@1 +2), (@2 +0)
   mov [@0 +2],A
   GetLSB
   add [@0 + 3],A
   adc [@0 + 2],0
;  (4) x1y0 r12
   PushMulX  (@1 +1)
   GetUnsignedMSB  (@1 +1), (@2 +0)
   mov [@0 +1],A
   GetLSB
   add [@0 + 2],A
   adc [@0 + 1],0
;  (5) x0y0 r01
   PushMulX  (@1 +0)
   GetUnsignedMSB  (@1 +0), (@2 +0)
   mov [@0 +0],A
   GetLSB
   add [@0 + 1],A
   adc [@0 + 0],0
;  (6) x0y1 r012
   PushMulY  (@2 +1)
   GetUnsignedMSB  (@1 +0), (@2 +1)
   push A
   GetLSB
   add [@0 + 2],A
   pop A
   adc [@0 + 1],A
   adc [@0 + 0],0
;  (7) x0y2 r0123
   PushMulY  (@2 +2)
   GetUnsignedMSB  (@1 +0), (@2 +2)
   push A
   GetLSB
   add [@0 + 3],A
   pop A
   adc [@0 + 2],A
   adc [@0 + 1],0
   adc [@0 + 0],0
;  (8) x1y2 r01234
   PushMulX  (@1 +1)
   GetUnsignedMSB  (@1 +1), (@2 +2)
   push A
   GetLSB
   add [@0 + 4],A
   pop A
   adc [@0 + 3],A
   mov A,0
   adc [@0 + 2],A
   adc [@0 + 1],A
   adc [@0 + 0],A
;  (9) x1y1 r0123
   PushMulY  (@2 +1)
   GetUnsignedMSB  (@1 +1), (@2 +1)
   push A
   GetLSB
   add [@0 + 3],A
   pop A
   adc [@0 + 2],A
   adc [@0 + 1],0
   adc [@0 + 0],0
endm
```

The time this macro takes to fetch the contents of RAM, perform a 24-bit unsigned multiply, and return the 48-bit unsigned result back to RAM is 802 CPU cycles. For a CPU speed of 24 MHz this works out to 33.4 usec. It seems reasonable that the rest of the overhead needed to perform a floating-point multiply could be done in 15 usec. This would make for a floating-point multiply that takes less than 50 usec.

## Conclusion

A signed multiplier does not limit a user to only signed math. An understanding of the differences between signed and unsigned values makes fast, multi-byte multiplication possible. Table 1 gives a summary of the different multiplication routines discussed in this Application Note.

Ideas discussed herein will allow the user to develop their own multiplication routines customized for their particular data processing requirements.

**Table 1: Multiplication Routine Summary**

| Operands | Result | CPU Cycles | Size in Bytes |
|---|---|---|---|
| 8 Bit | 8 Bit | 31 | 12 |
| 8-Bit Signed | 16-Bit Signed | 42 | 16 |
| 8-Bit Unsigned | 16-Bit Unsigned | 80 | 28 |
| 16 Bit | 16 Bit | 115 | 46 |
| 16-Bit Unsigned | 32-Bit Unsigned | 333 | 89 |
| 24-Bit Unsigned | 48-Bit Unsigned | 802 | 279 |

Cypress MicroSystems, Inc.
22027 17th Avenue S.E. Suite 201
Bothell, WA 98021
Phone: 877.751.6100
Fax: 425.939.0999
http://www.cypressmicro.com/ http://www.cypress.com/aboutus/sales_locations.cfm support@cypressmicro.com
Copyright © 2002 Cypress MicroSystems, Inc. All rights reserved.
PSoC™ (Programmable System on Chip) is a trademark of Cypress MicroSystems, Inc.
All other trademarks or registered trademarks referenced herein are property of the respective corporations.
The information contained herein is subject to change without notice.