8-bit AVR® Microcontroller

Application Note

AVR201: Using the AVR® Hardware Multiplier

Features

- **8- and 16-bit Implementations**
- **Signed and Unsigned Routines**
- **Fractional Signed and Unsigned Multiply**
- **Executable Example Programs**

Introduction

The megaAVR is a series of new devices in the AVR RISC microcontroller family that includes, among other new enhancements, a hardware multiplier. This multiplier is capable of multiplying two 8-bit numbers, giving a 16-bit result using only two clock cycles. The multiplier can handle both signed and unsigned integer and fractional numbers without speed or code size penalty. The first section of this document will give some examples of using the multiplier for 8-bit arithmetic.

To be able to use the multiplier, six new instructions are added to the AVR instruction set. These are:

- MUL, multiplication of unsigned integers
- MULS, multiplication of signed integers
- MULSU, multiplication of a signed integer with an unsigned integer
- FMUL, multiplication of unsigned fractional numbers
- FMULS, multiplication of signed fractional numbers
- FMULSU, multiplication of a signed fractional number and with an unsigned fractional number

The MULSU and FMULSU instructions are included to improve the speed and code density for multiplication of 16-bit operands. The second section will show examples of how to efficiently use the multiplier for 16-bit arithmetic.

The component that makes a dedicated digital signal processor (DSP) specially suitable for signal processing is the multiply-accumulate (MAC) unit. This unit is functionally equivalent to a multiplier directly connected to an arithmetic logic unit (ALU). The megaAVR microcontrollers are designed to give the AVR family the ability to effectively perform the same multiply-accumulate operation. This application note will therefore include examples of implementing the MAC operation.

The multiply-accumulate operation (sometimes referred to as multiply-add operation) has one critical drawback. When adding multiple values to one result variable, even when adding positive and negative values to some extent cancel each other, the risk of the result variable to overrun its limits becomes evident, i.e. if adding 1 to a signed byte variable that contains the value +127, the result will be -128 instead of +128. One solution often used to solve this problem is to introduce fractional numbers, i.e. numbers that are less than 1 and greater than or equal to -1. The final section presents some issues regarding the use of fractional numbers.

Rev. 1631A–02/00

In addition to the new multiplication instruction, a few other additions and improvements are made to the megaAVR processor core. One improvement that is particularly useful is the new instruction MOVW - Copy Register Word, which makes a copy of one register pair into another register pair.

The file "AVR201.asm" contains the application note source code of the 16-bit multiply routines.

A listing of all implementations with key performance specifications is given in Table 1.

Table 1. Performance Summary

8-bit x 8-bit Routines:	**Word (Cycles)**
Unsigned multiply 8 x 8 = 16 bits	1 (2)
Signed multiply 8 x 8 = 16 bits	1 (2)
Fractional signed/unsigned multiply 8 x 8 = 16 bits	1 (2)
Fractional signed multiply-accumulate 8 x 8 += 16 bits	3 (4)
16-bit x 16-bit Routines:	
Signed/unsigned multiply 16 x 16 = 16 bits	6 (9)
Unsigned multiply 16 x 16 = 32 bits	13 (17)
Signed multiply 16 x 16 = 32 bits	15 (19)
Signed multiply-accumulate 16 x 16 += 32 bits	19 (23)
Fractional signed multiply 16 x 16 = 32 bits	16 (20)
Fractional signed multiply-accumulate 16 x 16 += 32 bits	21 (25)

8-bit Multiplication

Doing an 8-bit multiply using the hardware multiplier is simple, as the examples in this section will clearly show. Just load the operands into two registers (or only one for square multiply) and execute one of the multiply instructions. The result will be placed in register pair R0:R1. However, note that only the MUL instruction does not have register usage restrictions. Figure 1 shows the valid (operand) register usage for each of the multiply instructions.

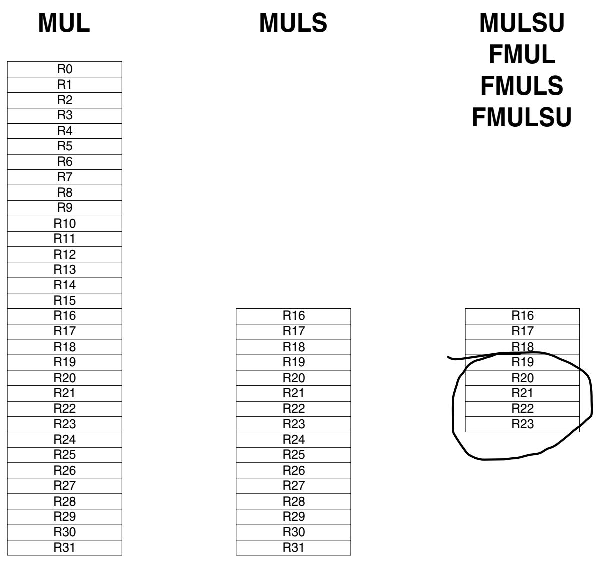
Example 1 – Basic Usage

The first example shows an assembly code that reads the port B input value and multiplies this value with a constant (5) before storing the result in register pair R17:R16.

```
in    r16,PINB     ; Read pin values
ldi   r17,5        ; Load 5 into r17
mul   r16,r17      ; r1:r0 = r17 * r16
movw  r17:r16,r1:r0; Move the result to the r17:r16
                   ; register pair
```

Note the use of the new MOVW instruction. This example is valid for all of the multiply instructions.

Figure 1. Valid Register Usage



Example 2 – Special Cases

This example shows some special cases of the MUL instruction that are valid.

```
lds   r0,variableA; Load r0 with SRAM variable A
lds   r1,variableB; Load r1 with SRAM variable B
mul   r1,r0       ; r1:r0 = variable A * variable B

lds   r0,variableA; Load r0 with SRAM variable A
mul   r0,r0       ; r1:r0 = square(variable A)
```

Even though the operand is put in the result register pair R1:R0, the operation gives the correct result since R1 and R0 are fetched in the first clock cycle and the result is stored back in the second clock cycle.

Example 3 – Multiply-accumulate Operation

The final example of 8-bit multiplication shows a multiply-accumulate operation. The general formula can be written as:

$$c(n) = a(n) \times b + c(n-1)$$

```
; r17:r16 = r18 * r19 + r17:r16

in  r18,PINB ; Get the current pin value on port B
ldi r19,b    ; Load constant b into r19
mulsr19,r18  ; r1:r0 = variable A * variable B
add r16,r0   ; r17:r16 += r1:r0
adc r17,r1
```

Typical applications for the multiply-accumulate operation are FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filters, PID regulators and FFT (Fast Fourier Transform). For these applications the FMULS instruction is particularly useful. The main advantage of using the FMULS instruction instead of the MULS instruction is that the 16-bit result of the FMULS operation always may be approximated to a (well-defined) 8-bit format. This is discussed further in the “Using Fractional Numbers” section.

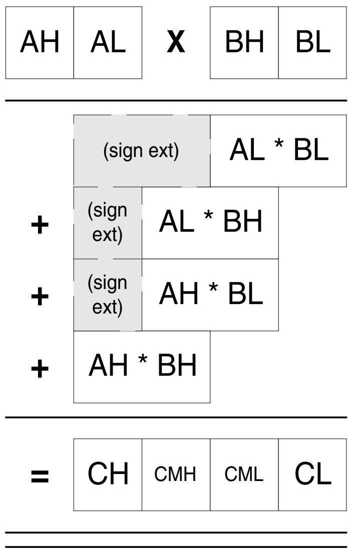
16-bit Multiplication

The new multiply instructions are specifically designed to improve 16-bit multiplication. This section presents solutions for using the hardware multiplier to do multiplication with 16-bit operands.

Figure 2 schematically illustrates the general algorithm for multiplying two 16-bit numbers with a 32-bit result (C = A • B). AH denotes the high byte and AL the low byte of the A operand. CMH denotes the middle high byte and CML the middle low byte of the result C. Equal notations are used for the remaining bytes.

The algorithm is basic for all multiplication. All of the partial 16-bit results are shifted and added together. The sign extension is necessary for signed numbers only, but note that the carry propagation must still be done for unsigned numbers.

Figure 2. 16-bit Multiplication, General Algorithm



16-bit x 16-bit = 16-bit Operation

This operation is valid for both unsigned and signed numbers, even though only the unsigned multiply instruction (MUL) is needed. This is illustrated in Figure 3. A mathematical explanation is given:

When A and B are positive numbers, or at least one of them is zero, the algorithm is clearly correct, provided that the product C = A • B is less than 2^{16} if the product is to be used as an unsigned number, or less than 2^{15} if the product is to be used as a signed number.

When both factors are negative, the two’s complement notation is used; A = 2^{16} - |A| and B = 2^{16} - |B|:

$$C = A \bullet B = (2^{16} - |A|) \bullet (2^{16} - |B|) = |A \bullet B| + 2^{32} - 2^{16} \bullet (|A| + |B|)$$

Here we are only concerned with the 16 LSBs; the last part of this sum will be discarded and we will get the (correct) result C = |A • B|.

Figure 3. 16-bit Multiplication, 16-bit Result

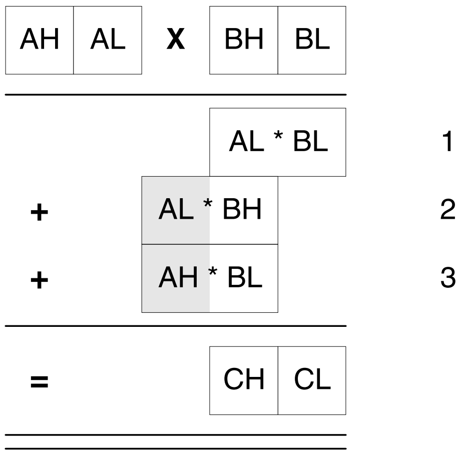
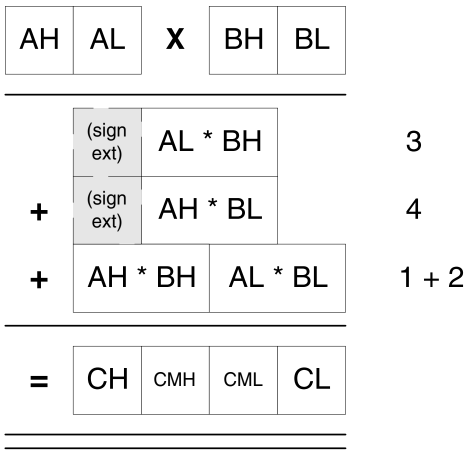


Figure 4. 16-bit Multiplication, 32-bit Result



When one factor is negative and one factor is positive, for example, A is negative and B is positive:

C = A • B = $(2^{16}$ - |A|) • |B| = $(2^{16}$ • |B|) - |A • B| = $(2^{16}$ - |A • B|) + 2^{16} • (|B| - 1)

The MSBs will be discarded and the correct two's complement notation result will be C = 2^{16} - |A • B|.

The product must be in the range $0 \leq C \leq 2^{16} - 1$ if unsigned numbers are used, and in the range $-2^{15} \leq C \leq 2^{15} - 1$ if signed numbers are used.

When doing integer multiplication in C language, this is how it is done. The algorithm can be expanded to do 32-bit multiplication with 32-bit result.

16-bit x 16-bit = 32-bit Operation

Example 4 – Basic Usage 16-bit x 16-bit = 32-bit Integer Multiply

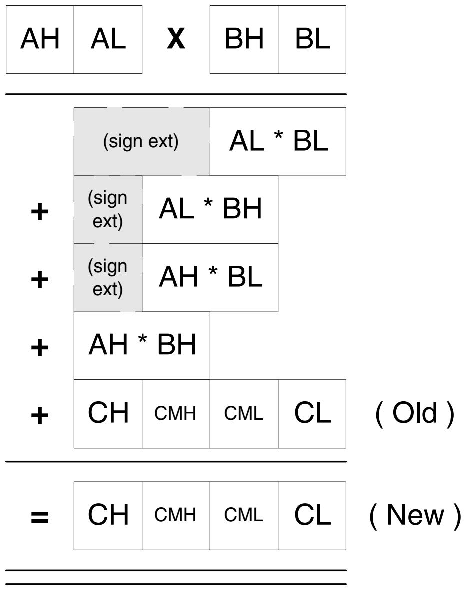
Below is an example of how to call the 16 x 16 = 32 multiply subroutine. This is also illustrated in Figure 4.

```
ldi R23,HIGH(672)
ldi R22,LOW(672) ; Load the number 672 into r23:r22
ldi R21,HIGH(1844)
ldi R20,LOW(1844); Load the number 1844 into r21:r20
callmul16x16_32  ; Call 16bits x 16bits = 32bits
                 ; multiply routine
```

The 32-bit result of the unsigned multiplication of 672 and 1844 will now be in the registers R19:R18:R17:R16. If "muls16x16_32" is called instead of "mul16x16_32", a signed multiplication will be executed. If "mul16x16_16" is called, the result will only be 16 bits long and will be stored in the register pair R17:R16. In this example, the 16-bit result will not be correct.

16-bit Multiply-accumulate Operation

Figure 5. 16-bit Multiplication, 32-bit Accumulated Result



Using Fractional Numbers

Unsigned 8-bit fractional numbers use a format where numbers in the range [0, 2> are allowed. Bits 6 - 0 represent the fraction and bit 7 represents the integer part (0 or 1), i.e. a 1.7 format. The FMUL instruction performs the same operation as the MUL instruction, except that the result is left-shifted 1 bit so that the high byte of the 2-byte result will have the same 1.7 format as the operands (instead of a 2.6 format). Note that if the product is equal to or higher than 2, the result will not be correct.

To fully understand the format of the fractional numbers, a comparison with the integer number format is useful: Table 2 illustrates the two 8-bit unsigned numbers formats. Signed fractional numbers, like signed integers, use the familiar two's complement format. Numbers in the range [-1, 1> may be represented using this format.

If the byte "1011 0010" is interpreted as an unsigned integer, it will be interpreted as 128 + 32 + 16 + 2 = 178. On the other hand, if it is interpreted as an unsigned fractional number, it will be interpreted as 1 + 0.25 + 0.125 + 0.015625 = 1.390625. If the byte is assumed to be a signed number, it will be interpreted as 178 - 256 = -122 (integer) or as 1.390625 - 2 = -0.609375 (fractional number).

Table 2. Comparison of Integer and Fractional Formats

Bit Number	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Unsigned integer bit significance	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Unsigned fractional number bit significance	$2^0 = 1$	$2^{-1} = 0.5$	$2^{-2} = 0.25$	$2^{-3} = 0.125$	$2^{-4} = 0.0625$	$2^{-5} = 0.3125$	$2^{-6} = 0.015625$	$2^{-7} = 0.0078125$

Using the FMUL, FMULS and FMULSU instructions should not be more complex than the MUL, MULS and MULSU instructions. However, one potential problem is to assign fractional variables right values in a simple way. The fraction 0.75 (= 0.5 + 0.25) will, for example, be "0110 0000" if 8 bits are used.

To convert a positive fractional number in the range [0, 2> (for example 1.8125) to the format used in the AVR, the following algorithm, illustrated by an example, should be used:

Is there a "1" in the number?
Yes, 1.8125 is higher than or equal to 1.
  Byte is now "1xxx xxxx"
Is there a "0.5" in the rest?
  0.8125 / 0.5 = 1.625
Yes, 1.625 is higher than or equal to 1.
Byte is now "11xx xxxx"
Is there a "0.25" in the rest?
  0.625 / 0.5 = 1.25
Yes, 1.25 is higher than or equal to 1.
  Byte is now "111x xxxx"
Is there a "0.125" in the rest?
  0.25 / 0.5 = 0.5
No, 0.5 is lower than 1.
  Byte is now "1110 xxxx"
Is there a "0.0625" in the rest?
  0.5 / 0.5 = 1
Yes, 1 is higher than or equal to 1.
  Byte is now "1110 1xxx"

Since we do not have a rest, the remaining three bits will be zero, and the final result is "1110 1000", which is 1 + 0.5 + 0.25 + 0.0625 = 1.8125.

To convert a negative fractional number, first add 2 to the number and then use the same algorithm as already shown.

16-bit fractional numbers use a format similar to that of 8-bit fractional numbers; the high 8 bits have the same format as the 8-bit format. The low 8 bits are only an increase of accuracy of the 8-bit format; while the 8-bit format has an accuracy of $\pm 2^{-8}$, the16-bit format has an accuracy of $\pm 2^{-16}$. Then again, the 32-bit fractional numbers are an increase of accuracy to the 16-bit fractional numbers. Note the important difference between integers and fractional numbers when extra byte(s) are used to store the number: while the accuracy of the numbers is increased when fractional numbers are used, the range of numbers that may be represented is extended when integers are used.

As mentioned earlier, using signed fractional numbers in the range [-1, 1> has one main advantage to integers: when multiplying two numbers in the range [-1, 1>, the result will be in the range [-1, 1], and an approximation (the highest byte(s)) of the result may be stored in the same number of bytes as the factors, with one exception: when both factors are -1, the product should be 1, but since the number 1 cannot be represented using this number format, the FMULS instruction will instead place the number -1 in R1:R0. The user should therefore assure that at least one of the operands is not -1 when using the FMULS instruction. The 16-bit x 16-bit fractional multiply also has this restriction.

Example 5 – Basic Usage 8-bit x 8-bit = 16-bit Signed Fractional Multiply

This example shows an assembly code that reads the port B input value and multiplies this value with a fractional constant (-0.625) before storing the result in register pair R17:R16.

```
in    r16,PINB     ; Read pin values
ldi   r17,$B0      ; Load -0.625 into r17
fmuls r16,r17      ; r1:r0 = r17 * r16
movw  r17:r16,r1:r0; Move the result to the r17:r16
                    ; register pair
```

Note that the usage of the FMULS (and FMUL) instructions is very similar to the usage of the MULS and MUL instructions.

Example 6 – Multiply-accumulate Operation

The example below uses data from the ADC. The ADC should be configured so that the format of the ADC result is compatible with the fractional two's complement format. For the ATmega83/163, this means that the ADLAR bit in the ADMUX I/O register is set and a differential channel is used. (The ADC result is normalized to one.)

```
ldi r23,$62   ; Load highbyte of
                    ; fraction 0.771484375
ldi r22,$C0   ; Load lowbyte of
                    ; fraction 0.771484375
in  r20,ADCL  ; Get lowbyte of ADC conversion
in  r21,ADCH  ; Get highbyte of ADC conversion
callfmac16x16_32;Call routine for signed fractional
                    ; multiply accumulate
```

The registers R19:R18:R17:R16 will be incremented with the result of the multiplication of 0.771484375 with the ADC conversion result. In this example, the ADC result is treated as a signed fraction number. We could also treat it as a signed integer and call it "mac16x16_32" instead of "fmac16x16_32". In this case, the 0.771484375 should be replaced with an integer.

Implementations

Function

mul16x16_16

Description

Multiply of two 16-bit numbers with a 16-bit result.

Usage

R17:R16 = R23:R22 • R21:R20

Statistics

Cycles: 9 + ret

Words: 6 + ret

Register usage: R0, R1 and R16 to R23 (8 registers)

Note: Full orthogonality, i.e. any register pair can be used as long as the result and the two operands do not share register pairs. The routine is non-destructive to the operands.

```
mul16x16_16:
   mul   r22, r20    ; al * bl
   movw  r17:r16, r1:r0
   mul   r23, r20    ; ah * bl
   add   r17, r0
   mul   r21, r22    ; bh * al
   add   r17, r0
   ret
```

Function

mul16x16_32

Description

Unsigned multiply of two 16-bit numbers with a 32-bit result.

Usage

R19:R18:R17:R16 = R23:R22 • R21:R20

Statistics

Cycles: 17 + ret

Words: 13 + ret

Register usage: R0 to R2 and R16 to R23 (11 registers)

Note: Full orthogonality, i.e. any register pair can be used as long as the 32-bit result and the two operands do not share register pairs. The routine is non-destructive to the operands.

```
mul16x16_32:
   clr   r2
   mul   r23, r21    ; ah * bh
   movw  r19:r18, r1:r0
   mul   r22, r20    ; al * bl
   movw  r17:r16, r1:r0
   mul   r23, r20    ; ah * bl
   add   r17, r0
   adc   r18, r1
   adc   r19, r2
   mul   r21, r22    ; bh * al
   add   r17, r0
   adc   r18, r1
   adc   r19, r2
   ret
```

Function

muls16x16_32

Description

Signed multiply of two 16-bit numbers with a 32-bit result.

Usage

R19:R18:R17:R16 = R23:R22 • R21:R20

Statistics

Cycles: 19 + ret

Words: 15 + ret

Register usage: R0 to R2 and R16 to R23 (11 registers)

Note: The routine is non-destructive to the operands.

```
muls16x16_32:
   clr   r2
   muls  r23, r21; (signed)ah * (signed)bh
   movw  r19:r18, r1:r0
   mul   r22, r20; al * bl
   movw  r17:r16, r1:r0
   mulsu r23, r20; (signed)ah * bl
   sbc   r19, r2 ; Sign extend
   add   r17, r0
   adc   r18, r1
   adc   r19, r2
   mulsu r21, r22; (signed)bh * al
   sbc   r19, r2 ; Sign Extend
   add   r17, r0
   adc   r18, r1
   adc   r19, r2
   ret
```

Function

mac16x16_32

Description

Signed multiply-accumulate of two 16-bit numbers with a 32-bit result.

Usage

R19:R18:R17:R16 += R23:R22 • R21:R20

Statistics

Cycles: 23 + ret

Words: 19 + ret

Register usage: R0 to R2 and R16 to R23 (11 registers)

```
mac16x16_32:      ; Register Usage Optimized
   clr   r2

   muls  r23, r21 ; (signed)ah * (signed)bh
   add   r18, r0
   adc   r19, r1

   mul   r22, r20 ; al * bl
   add   r16, r0
   adc   r17, r1
   adc   r18, r2
   adc   r19, r2

   mulsu r23, r20 ; (signed)ah * bl
   sbc   r19, r2
   add   r17, r0
   adc   r18, r1
   adc   r19, r2

   mulsu r21, r22 ; (signed)bh * al
   sbc   r19, r2  ; Sign extend
   add   r17, r0
   adc   r18, r1
   adc   r19, r2

   ret

mac16x16_32_method_B:      ; uses two temporary
registers (r4,r5), Speed / Size Optimized
               ; but reduces cycles/words by 1
   clr   r2

   muls  r23, r21 ; (signed)ah * (signed)bh
   movw  r5:r4,r1:r0

   mul   r22, r20 ; al * bl

   add   r16, r0
   adc   r17, r1
   adc   r18, r4
   adc   r19, r5

   mulsu r23, r20 ; (signed)ah * bl
   sbc   r19, r2  ; Sign extend
   add   r17, r0
   adc   r18, r1
   adc   r19, r2

   mulsu r21, r22 ; (signed)bh * al
   sbc   r19, r2  ; Sign extend
   add   r17, r0
   adc   r18, r1
   adc   r19, r2

   ret
```

Function

fmuls16x16_32

Description

Signed fractional multiply of two 16-bit numbers with a 32-bit result.

Usage

R19:R18:R17:R16 = (R23:R22 • R21:R20) << 1

Statistics

Cycles: 20 + ret

Words: 16 + ret

Register usage: R0 to R2 and R16 to R23 (11 registers)

Note: The routine is non-destructive to the operands.

```
fmuls16x16_32:
   clr   r2
   fmuls r23, r21   ; ( (signed)ah * (signed)bh ) << 1
   movw  r19:r18, r1:r0
   fmul  r22, r20   ; ( al * bl ) << 1
   adc   r18, r2
   movw  r17:r16, r1:r0
   fmulsu   r23, r20; ( (signed)ah * bl ) << 1
   sbc   r19, r2    ; Sign extend
   add   r17, r0
   adc   r18, r1
   adc   r19, r2
   fmulsu   r21, r22; ( (signed)bh * al ) << 1
   sbc   r19, r2    ; Sign extend
   add   r17, r0
   adc   r18, r1
   adc   r19, r2
   ret
```

Function

fmac16x16_32

Description

Signed fractional multiply-accumulate of two 16-bit numbers with a 32-bit result.

Usage

R19:R18:R17:R16 += (R23:R22 • R21:R20) << 1

Statistics

Cycles: 25 + ret

Words: 21 + ret

Register usage: R0 to R2 and R16 to R23 (11 registers)

```
fmac16x16_32:        ; Register usage optimized
   clr   r2

   fmuls r23, r21    ; ( (signed)ah * (signed)bh )
<< 1
   add   r18, r0
   adc   r19, r1

   fmul  r22, r20    ; ( al * bl ) << 1
   adc   r18, r2
   adc   r19, r2
   add   r16, r0
   adc   r17, r1
   adc   r18, r2
   adc   r19, r2

   fmulsu   r23, r20 ; ( (signed)ah * bl ) << 1
   sbc   r19, r2
   add   r17, r0
   adc   r18, r1
   adc   r19, r2

   fmulsu   r21, r22 ; ( (signed)bh * al ) << 1
   sbc   r19, r2
   add   r17, r0
   adc   r18, r1
   adc   r19, r2

   ret

fmac16x16_32_method_B:     ; uses two temporary
registers (r4,r5), speed / Size optimized
               ; but reduces cycles/words by 2
   clr   r2

   fmuls r23, r21    ; ( (signed)ah * (signed)bh )
<< 1
   movw  r5:r4,r1:r0
   fmul  r22, r20    ; ( al * bl ) << 1
   adc   r4, r2

   add   r16, r0
   adc   r17, r1
   adc   r18, r4
   adc   r19, r5
```

```
fmulsu   r23, r20 ; ( (signed)ah * bl ) << 1
sbc   r19, r2
add   r17, r0
adc   r18, r1
adc   r19, r2
fmulsu   r21, r22 ; ( (signed)bh * al ) << 1
sbc   r19, r2
add   r17, r0
adc   r18, r1
adc   r19, r2

ret
```

Comment on Implementations

All 16-bit x 16-bit = 32-bit functions implemented here start by clearing the R2 register, which is just used as a "dummy" register with the "add with carry" (ADC) and "subtract with carry" (SBC) operations. These operations do not alter the contents of the R2 register. If the R2 register is not used elsewhere in the code, it is not necessary to clear the R2 register each time these functions are called, but only once prior to the first call to one of the functions.

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL (408) 441-0311
FAX (408) 487-2600

Europe
Atmel U.K., Ltd.
Coliseum Business Centre
Riverside Way
Camberley, Surrey GU15 3YL
England
TEL (44) 1276-686-677
FAX (44) 1276-686-697

Asia
Atmel Asia, Ltd.
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
Atmel Japan K.K.
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Atmel Colorado Springs
1150 E. Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL (719) 576-3300
FAX (719) 540-1759

Atmel Rousset
Zone Industrielle
13106 Rousset Cedex
France
TEL (33) 4-4253-6000
FAX (33) 4-4253-6001

Fax-on-Demand
North America:
1-(800) 292-8635
International:
1-(408) 441-0732

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

BBS
1-(408) 436-4309

© Atmel Corporation 2000.
Atmel Corporation makes no warranty for the use of its products, other than those expressly contained in the Company's standard warranty which is detailed in Atmel's Terms and Conditions located on the Company's web site. The Company assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information contained herein. No licenses to patents or other intellectual property of Atmel are granted by the Company in connection with the sale of Atmel products, expressly or by implication. Atmel's products are not authorized for use as critical components in life support devices or systems.

Marks bearing ® and/or ™ are registered trademarks and trademarks of Atmel Corporation.

Terms and product names in this document may be trademarks of others.

Printed on recycled paper.