

The MSP430 Hardware Multiplier

Function and Applications

Application Report

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Hardware and Registers | 2 |
| 1.1.1 | Operand1 Registers | 3 |
| 1.1.2 | Operand2 Register | 3 |
| 1.1.3 | SumLo Register | 4 |
| 1.1.4 | SumHi Register | 4 |
| 1.1.5 | SumExt Register | 4 |
| 1.2 | Hardware Multiplier Rules | 5 |
| 2 | Multiplication Modes | 6 |
| 2.1 | Unsigned Multiply | 6 |
| 2.2 | Signed Multiply | 6 |
| 2.3 | Multiply-and-Accumulate (MAC) | 7 |
| 2.4 | Multiplication Word Lengths | 7 |
| 3 | Hardware Multiplier Programming | 9 |
| 3.1 | Assembler .MACROS | 9 |
| 3.1.1 | Unsigned Multiplication 16×16-Bits | 9 |
| 3.1.2 | Signed Multiplication 16×16-Bits | 10 |
| 3.1.3 | Unsigned Multiplication 8×8-Bits | 11 |
| 3.1.4 | Signed Multiplication 8×8-Bits | 12 |
| 3.2 | Interrupt Usage | 13 |
| 3.3 | Speed Comparison with Software Multiplication | 14 |
| 3.4 | Software Hints | 14 |
| 3.5 | Speed Increase With Floating Point Package FPP4 | 15 |
| 4 | Software Applications | 17 |
| 4.1 | Multiplication Exceeding 16 Bits | 17 |
| 4.2 | Sensor Characteristics | 21 |
| 4.3 | Table Calculations | 21 |
| 4.4 | Wave Digital Filters | 22 |
| 4.5 | Finite Impulse Response (FIR) Digital Filter | 22 |
| 4.6 | Fast Fourier Transform Algorithm | 24 |
| 5 | Conclusion | 29 |
| 6 | References | 29 |

List of Figures

| | | |
|---|---|----|
| 1 | Hardware Multiplier Block Diagram | 2 |
| 2 | Hardware Multiplier Internal Connections | 3 |
| 3 | Multiplication Exceeding 16 Bits | 17 |
| 4 | 40 × 40-Bit Unsigned Multiplication MPYU40 | 17 |
| 5 | 32 × 32-Bit Signed Multiplication MPYS32 | 19 |
| 6 | Finite Impulse Response Filter | 22 |
| 7 | Finite Impulse Response Filter Storage | 23 |
| 8 | RAM and ROM Allocation for the Fast Fourier Transform Algorithm | 25 |

List of Tables

| | | |
|---|---|----|
| 1 | Results With Unsigned-Multiply Mode | 6 |
| 2 | Results With Signed-Multiply Mode | 7 |
| 3 | Results With Unsigned Multiply-and-Accumulate Mode | 7 |
| 4 | CPU Cycles Needed With Different Multiplication Modes | 14 |
| 5 | CPU Cycles Needed for FPP Multiplication (FLT_MUL) | 15 |

The MSP430 Hardware Multiplier

Lutz Bierl

ABSTRACT

The 16×16-bit hardware multiplier of the MSP430 family is explained in detail. Function, modes and proven application examples are given for this fast and versatile peripheral. Also shown is a comparison of the speed of solutions using this peripheral versus pure software solutions.

1 Introduction

The hardware multiplier allows three different multiply operations (modes):

- Multiplication of unsigned 8-bit and 16-bit operands (MPY)
- Multiplication of signed 8-bit and 16-bit operands (MPYS)
- Multiply-and-accumulate function (MAC) using unsigned 8-bit and 16-bit operands

Any mixture of operand lengths (8 and 16 bits) is possible. Additional operations are possible when supplemental software is used, such as signed multiply-and-accumulate.

Figure 1 shows the hardware modules that comprise the MSP430 multiplier. The accessible registers are explained in the following sections. Figure 1 is not intended to depict the physical reality; it illustrates the hardware multiplier from the programmer's point of view.

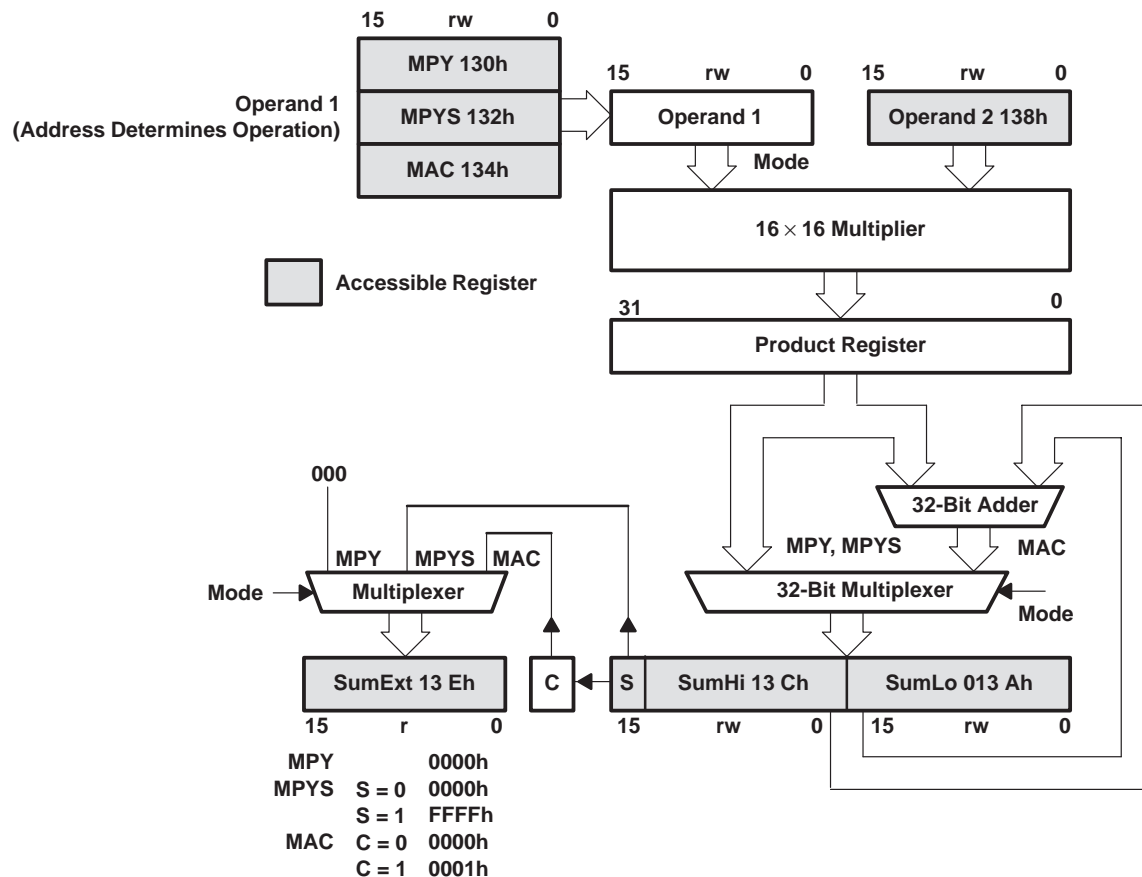


Figure 1. Hardware Multiplier Block Diagram

1.1 Hardware and Registers

The hardware multiplier is not part of the MSP430 CPU—it is a peripheral that does not interfere with the CPU activities. The multiplier uses normal peripheral registers that are loaded and read using CPU instructions. The programmer-accessible registers are explained in this chapter.

The hardware multiplier registers are not affected by POR or PUC events. Read and write operations can be performed in all the registers but the SumExt register. Hardware multiplier definitions are presented in Section 3.

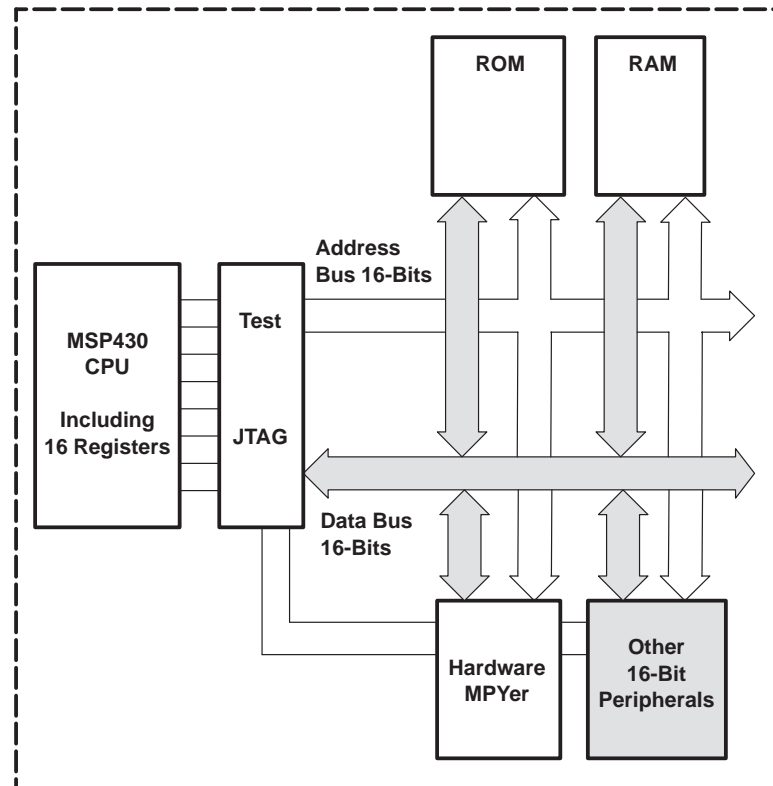


Figure 2. Hardware Multiplier Internal Connections

1.1.1 Operand1 Registers

The operational mode of the MSP430 hardware multiplier is determined by the address where Operand1 is written:

- **Address 130h:** execute unsigned multiplication (MPY)
- **Address 132h:** execute signed multiplication (MPYS)
- **Address 134h:** execute unsigned multiply-and-accumulate (MAC)

The address of Operand1 alone determines the operation to be performed by the multiplier (after modification of Operand2). No operation is started by modification of Operand1 alone.

EXAMPLE: a multiply unsigned (MPY) operation is defined and started. The two operands reside in R14 and R15.

```
MOV    R15,&130h ; Define MPY operation
MOV    R14,&138h ; Start MPY with operand 2
...    ; Product in SumHi|SumLo
```

1.1.2 Operand2 Register

The Operand2 Register (at address 138h) is common to all three multiplier modes. Modification of this register (normally with a MOV instruction) starts the selected multiplication of the two operands contained in Operand1 and Operand2 registers. The result is written immediately into the three hardware registers SumExt, SumHi, and SumLo. The result can be accessed with the next instruction, unless indirect addressing modes are used for source addressing.

1.1.3 *SumLo Register*

This 16-bit register contains the lower 16 bits of the calculated product or sum. All instructions may be used to access or modify the *SumLo* register. The high byte cannot be accessed with byte instructions.

1.1.4 *SumHi Register*

The contents of this 16-bit register, which depend on the previously executed operation, are as follows:

- **MPY:** the most-significant word of the calculated product.
- **MPYS:** the most-significant word, including the sign of the calculated product. Twos complement notation is used for the product.
- **MAC:** the most significant word of the calculated sum.

All instructions may be used with the *SumHi* register. The high byte cannot be accessed using byte instructions.

1.1.5 *SumExt Register*

The sum extension register *SumExt* allows calculations with results exceeding the 32-bit range. This read-only register holds the most significant part of the result (bits 32 and higher). The content of *SumExt* is different for each multiplication mode:

- **MPY:** *SumExt* always contains zero, with no carry possible. The largest possible result is: 0FFFFh x 0FFFFh = 0FFFE0001h.
- **MPYS:** *SumExt* contains the extended sign of the 32-bit result (bit 31). If the result of the multiplication is negative (MSB=1) *SumExt* contains 0FFFFh. If the result is positive (MSB = 0) *SumExt* contains 0000h.
- **MAC:** *SumExt* contains the carry of the accumulate operation. *SumExt* contains 0001 if a carry occurred during the accumulation of the new product to the old one, and zero otherwise.

Register *SumExt* simplifies multiple word operations—straightforward additions are performed without conditional jumps, saving time and ROM space.

EXAMPLE: the new product of a MPYS operation (operands in R14 and R15) is added to a signed 64-bit result located in RAM words RESULT through RESULT+6:

```
MOV    R15,&MPYS           ; First operand
MOV    R14,&OP2             ; Start MPYS with operand 2
ADD     SumLo,RESULT        ; Lower 16 bits of result
ADDC    SumHi,RESULT+2      ; Upper 16 bits
ADDC    SumExt,RESULT+4     ; Result bits 32 to 47
ADDC    SumExt,RESULT+6     ; Result bits 48 to 63
```

NOTE: Using the MACROs defined in the *Assembler .MACROS* section instead of the method shown above is strongly recommended. The resulting code is much more descriptive when using MACROs containing known abbreviations such as MPYU, MPYS, and MACU.

With the software shown above, no checks and conditional jumps are necessary. The result automatically contains the signed-accumulated sum.

1.2 Hardware Multiplier Rules

The hardware multiplier is essentially a word module. The hardware registers can be addressed in word mode or in byte mode, but the byte mode can only address the lower bytes. The upper byte cannot be addressed.

The operand registers of the hardware multiplier (addresses 0130h, 0132h, 0134h and 0138h) behave like the CPU's working registers R0 to R15 when modified in byte mode: the upper byte is cleared in this case. This allows for any combination of 8-bit and 16-bit multiplications. See the examples in section 2.4.

The floating point package (FPP) version 4 uses the hardware multiplier when variable HW_MPY is set to one:

```
HW_MPY      .equ 1
```

See the *Metering Application Report* for details.

If the result of a hardware multiplier operation is addressed using indirect mode, or indirect-autoincrement mode, then an NOP instruction is necessary after multiplication to allow time to complete the multiplication. See the examples in section 3.1.

2 Multiplication Modes

The three different multiplication modes available are explained in the following sections.

2.1 Unsigned Multiply

The two operands written to operand registers 1 and 2 are treated as unsigned numbers with:

00000h being the smallest number
 0FFFFh being the largest number

The maximum possible result is obtained with input operands 0FFFFh and 0FFFFh:

$$0FFFFh \times 0FFFFh = 0FFFE0001h$$

No carry is possible, the *SumExt* register always contains zero. Table 1 gives the products for some selected operands.

Table 1. Results With Unsigned-Multiply Mode

| Operands | SumExt | SumHi | SumLo |
|-------------|--------|-------|-------|
| 0000 × 0000 | 0000 | 0000 | 0000 |
| 0001 × 0001 | 0000 | 0000 | 0001 |
| 7FFF × 7FFF | 0000 | 3FFF | 0001 |
| FFFF × FFFF | 0000 | FFFE | 0001 |
| 7FFF × FFFF | 0000 | 7FFE | 8001 |
| 8000 × 7FFF | 0000 | 3FFF | 8000 |
| 8000 × FFFF | 0000 | 7FFF | 8000 |
| 8000 × 8000 | 0000 | 4000 | 0000 |

2.2 Signed Multiply

The two operands written to operand registers 1 and 2 are treated as signed twos complement numbers with:

08000h being the most negative number (–32768)
 07FFFh being the most positive number (+32767)

The *SumExt* register contains the extended sign of the calculated result:

SumExt = 00000h: the result is positive
SumExt = 0FFFFh: the result is negative

Table 2 gives the signed-multiply products for some selected operands.

Table 2. Results With Signed-Multiply Mode

| Operands | SumExt | SumHi | SumLo |
|-------------|--------|-------|-------|
| 0000 × 0000 | 0000 | 0000 | 0000 |
| 0001 × 0001 | 0000 | 0000 | 0001 |
| 7FFF × 7FFF | 0000 | 3FFF | 0001 |
| FFFF × FFFF | 0000 | 0000 | 0001 |
| 7FFF × FFFF | FFFF | FFFF | 8001 |
| 8000 × 7FFF | FFFF | C000 | 8000 |
| 8000 × FFFF | 0000 | 0000 | 8000 |
| 8000 × 8000 | 0000 | 4000 | 0000 |

2.3 Multiply-and-Accumulate (MAC)

The two operands written to operand registers 1 and 2 are treated as unsigned numbers (0h to 0FFFFh). The maximum possible result is obtained with input operands 0FFFFh and 0FFFFh:

$$0FFFFh \times 0FFFFh = 0FFFE0001h$$

This result is added to the previous content of the two sum registers (*SumLo* and *SumHi*). If a carry occurs during this operation, the *SumExt* register contains 1 and is cleared otherwise.

SumExt = 00000h: no carry occurred during the accumulation

SumExt = 00001h: a carry occurred during the accumulation

The results of Table 3 assume that *SumHi* and *SumLo* contain the accumulated content C000,0000 before the execution of each example. See Table 1 for the results of an unsigned multiplication without accumulation.

Table 3. Results With Unsigned Multiply-and-Accumulate Mode

| Operands | SumExt | SumHi | SumLo |
|-------------|--------|-------|-------|
| 0000 × 0000 | 0000 | C000 | 0000 |
| 0001 × 0001 | 0000 | C000 | 0001 |
| 7FFF × 7FFF | 0000 | FFFF | 0001 |
| FFFF × FFFF | 0001 | BFFE | 0001 |
| 7FFF × FFFF | 0001 | 3FFE | 8001 |
| 8000 × 7FFF | 0000 | FFFF | 8000 |
| 8000 × FFFF | 0001 | 3FFF | 8000 |
| 8000 × 8000 | 0001 | 0000 | 0000 |

2.4 Multiplication Word Lengths

The MSP430 hardware multiplier allows all possible combinations of 8-bit and 16-bit operands. The examples given in chapter 3 for 8-bit and 16-bit operands may be adapted to mixed length operands.

Notice that input registers Operand1 and Operand2 behave like CPU registers, where the high-register byte is cleared if the register is modified by a byte instruction. This simplifies the use of 8-bit operands. The following are examples of 8-bit operand use for all three modes of the hardware multiplier.

```
; Use the 8-bit operand in R5 for an unsigned multiply.
;  MOV.B  R5,&MPY      ; The high byte is cleared
;
; Use an 8-bit operand for a signed multiply.
  MOV.B  R5,&MPYS      ; The high byte is cleared
  SXT    &MPYS         ; Extend sign to high byte
;
; Use an 8-bit operand for a multiply-and-accumulate.
;  MOV.B  R5,&MAC       ; The high byte is cleared
```

Operand2 is loaded in a similar fashion. This allows all four possible combinations of input operands:

16×16 8×16 16×8 8×8

The MACROS that can be modified are discussed in the following section.

3 Hardware Multiplier Programming

The hardware multiplier registers are initially defined in accordance with the *MSP430 Family Architecture Guide and Module Library* to prevent misunderstandings.

```
; MSP430 Hardware Multiplier Definitions
;
MPY    .equ    130h    ; Multiply unsigned
MPYS   .equ    132h    ; Multiply signed
MAC     .equ    134     ; Multiply-and-Accumulate
OP2     .equ    138h    ; Operand 2 Register
SumLo   .equ    013Ah   ; Result Register LSBs 15..0
SumHi   .equ    013Ch   ; Result Register MSBs 32..16
SumExt  .equ    013Eh   ; Sum Extension Register 47..33
```

3.1 Assembler .MACROS

Due to the MACRO construct of the multiply instructions (normally two MOV instructions form a multiplication sequence), all seven addressing modes are possible for source and destination. If the register-indirect or register-indirect with autoincrement addressing modes are used to address the result, an NOP is necessary after the .MACRO call due to the fast access time of these addressing modes. This allows sufficient time to complete the multiplication.

Examples for each .MACRO definition are given. The number of execution cycles required depends on the addressing modes used with multiplier and multiplicand.

The given MACROS can be easily converted to subroutines. An example follows for unsigned multiplication:

```
; Subroutine Definition for the unsigned multiplication
; 16x16 bits. The two operands are contained in R4 and R5
;
MPYU_16    MPYU16 R4,R5      ; Unsigned MPY 16x16
           RET               ; Result in SumHi|SumLo
;
```

3.1.1 Unsigned Multiplication 16×16-Bits

```
; Macro Definition for the unsigned multiplication 16x16 bits
;
MPYU16 .MACRO arg1,arg2      ; Unsigned MPY 16x16
      MOV    arg1,&0130h
      MOV    arg2,&0138h
      .ENDM                  ; Result in SumHi|SumLo
;
; Multiply the contents of the two registers R4 and R5
;
      MPYU16 R4,R5           ; MPY R4 and R5 unsigned
      MOV    SumLo,R6        ; LSBs of result to R6
      MOV    SumHi,R7        ; MSBs of result to R7
      ...                   ; Continue
;
```

```
; Multiply the contents located in a table, R6 points to
; The result is addressed in indirect mode: a NOP is necessary
; to allow the completion of the multiplication
;
    MOV    #SumLo,R5      ; Pointer to LSBs of result
    MPYU16 @R6+,@R6      ; MPYU the table contents
    NOP                    ; Allow completion of MPYU16
    MOV    @R5+,R7        ; Fetch LSBs of result
    MOV    @R5,R8         ; Fetch MSBs of result
    ...                  ; Continue
;
; Macro Definition for the unsigned multiplication and
; accumulation 16x16 bits.
;
MACU16 .MACRO arg1,arg2    ; Unsigned MAC 16x16
    MOV    arg1,&0134h    ; Carry in SumExt
    MOV    arg2,&0138h
    .ENDM                ; Result in SumExt | SumHi | SumLo
;
; Multiply-and-accumulate the contents of registers R5 and R6
; to the previous content (IROP1 x IROP2L) of the Sum registers
;
    MPYU16 IROP1,IROP2L   ; Initialize Sum registers
    MACU16 R5,R6          ; Add (R5 x R6) to result
    ADD    &SumExt,RAM    ; Add carry to RAM extension
    ...                  ; Continue
```

3.1.2 Signed Multiplication 16×16-Bits

The following software examples perform signed 16×16-bit multiplication (MPYS16), or 16×16-bit signed multiplication-and-accumulation (MACS16).

The *SumExt* register contains the extended sign of the result in *SumHi* and *SumLo*: 0000h (positive result), or 0FFFFh (negative result).

```
; Macro Definition for the signed multiplication 16x16 bits
;
MPYS16 .MACRO arg1,arg2    ; Signed MPY 16x16 bits
    MOV    arg1,&0132h
    MOV    arg2,&0138h
    .ENDM                ; Result in SumExt|SumHi|SumLo
;
; Multiply the contents of two registers R4 and R5
;
    MPYS16 R4,R5          ; MPY signed R4 and R5
    MOV    &SumLo,R6      ; LSBs of result to R6
    MOV    &SumHi,R7      ; MSBs of result to R7
    MOV    &SumExt,R8     ; Sign of result to R8
    ...                  ; Continue
;
; Multiply the contents located in a table, R6 points to
; The result is addressed in indirect mode: a NOP is necessary
```

```

; to allow the completion of the multiplication
;
    MOV    #SumLo,R5          ; Pointer to LSBs of result
    MPYS16 @R6+,@R6          ; MPY signed table contents
    NOP                      ; Allow completion of MPYS16
    MOV    @R5+,R7            ; LSBs of result to R7
    MOV    @R5+,R8            ; MSBs of result to R8
    MOV    @R5,R9             ; Sign of result to R9
    ...                      ; Continue
;
; Macro Definition for the signed multiplication-and-
; accumulation 16x16 bits. The accumulation is made in the
; RAM: MACHi, MACmid and MAClo. If more than 48 bits are used
; for the accumulation, the SumExt register is added to all
; further extensions (RAM or registers) here shown for only
; one extension (48 bits).
;
MACS16 .MACRO arg1,arg2      ; Signed MAC 16x16 bits
    MOV    arg1,&0132h        ; Signed MPY is used
    MOV    arg2,&0138h
    ADD    &SumLo,MAClo       ; Add LSBs to result
    ADDC   &SumHi,MACmid      ; Add MSBs to result
    ADDC   &SumExt,MACHi      ; Add SumExt to MSBs
    .ENDM
;
; Multiply and accumulate signed the contents of two tables
;
    MACS16 2(R6),@R5+         ; MAC for the table contents
    ....                     ; Accumulation is yet made

```

3.1.3 Unsigned Multiplication 8×8-Bits

When byte instructions are used to load the hardware multiplier registers, the high byte of these registers is cleared like a CPU register. This feature is used with unsigned 8×8-bit multiplication.

```

; Macro Definition for the unsigned multiplication 8x8 bits
;
MPYU8 .MACRO    arg1,arg2    ; Unsigned MPY 8x8
    MOV.B      arg1,&0130h    ; 00xx to 0130h
    MOV.B      arg2,&0138h    ; 00yy to 0138h
    .ENDM                      ; Result in SumLo. SumHi = 0
;
; Multiply the contents of the low bytes of two registers
;
    MPYU8      R12,R15        ; MPY low bytes of R12 and R15
    MOV        &SumLo,R6       ; 16 bit result to R6
    ...                      ; SumExt = SumHi = 0
;
; Macro Definition for the unsigned multiplication-and-
; accumulation 8x8 bits

```

```
;
MACU8  .MACRO    arg1,arg2      ; Unsigned MAC 8x8
      MOV.B     arg1,&0134h    ; 00xx
      MOV.B     arg2,&0138h    ; 00yy
      .ENDM                      ; Result in SumExt|SumHi|SumLo

;
; Multiply-and-accumulate the low bytes of R14 and a table
;
      MACU8      R14,@R5+      ; CALL the MACU8 macro (R5+1)
```

3.1.4 Signed Multiplication 8×8 Bits

When byte instructions are used to load the hardware multiplier registers, the high bytes of their registers are cleared like a CPU register. Therefore only the sign extension is required.

```
; Macro Definition for the signed multiplication 8x8 bits
;
MPYS8  .MACRO    arg1,arg2      ; Signed MPY 8x8
      MOV.B     arg1,&0132h    ; 00xx
      SXT       &0132h        ; Extend sign: 00xx or FFxx
      MOV.B     arg2,&0138h    ; 00yy
      SXT       &0138h        ; Extend sign: 00yy or FFyy
      .ENDM                      ; Result in SumExt|SumHi|SumLo

;
; Multiply signed the low bytes of R5 and location EDE
;
      MPYS8      R5,EDE        ; CALL the MPYS8 macro
      MOV        &SumLo,R6     ; Fetch result (16 bits)
      MOV        &SumHi,R7     ; Sign: 0000 or FFFF

;
; Macro Definition for the signed multiplication and
; accumulation 8x8 bits. The accumulation is made in the
; locations MACHi, MACmid and MAClo (registers or RAM)
; If more than 48 bits are used for the accumulation, the
; SumExt register is added to all further RAM extensions
;
MACS8  .MACRO    arg1,arg2      ; Signed MAC 8x8 bits
      MOV.B     arg1,&0132h    ; MPYS is used
      SXT       &0132h        ; Extend sign: 00xx or FFxx
      MOV.B     arg2,&0138h    ; 00yy
      SXT       &0138h        ; Extend sign
      ADD       &SumLo,MAClo   ; Accumulate LSBs 16 bits
      ADDC      &SumHi,MACmid   ; Accumulate MIDs
      ADDC      &SumExt,MACHi   ; Add SumExt to MSBs
      .ENDM                      ;
```



```

;
; Multiply-and-accumulate signed the contents of two byte
; tables
;
MACS8 2(R6),@R5+      ; CALL the MACS8 macro (R5+1)
....                  ; Accumulation is yet made

```

3.2 Interrupt Usage

No special rules apply when the hardware multiplier is used in the foreground only (interrupt handlers).

While not common in real-time programming, the hardware multiplier can be used in the foreground and the background, or in nested interrupt handlers. Three rules must be observed in these particular cases:

- The loading of registers Operand1 (MPY, MPYS and MAC) and Operand2 must not be separated by an interrupt while using the multiplier. The input information for Operand1 can not be restored because three different input registers are possible. See the example below.
- Registers Operand1 and Operand2 can not be reread by the background software because they may be overwritten by the interrupt handler.
- The information in Operand1 can not be used in more than one multiplication and therefore it must be rewritten. The Operand2 register must be rewritten to start the next multiplication. In normal operation, the float-point package FPP4 speeds up the calculation by using Operand1 twice.

```

; Background: multiplication is used together with interrupt
; The interrupt latency time is increased by 9 cycles.
; The NOP is necessary: one additional instruction may
; be executed after the DINT instruction
;
DINT                      ; Ensure non-interrupted -
NOP                      ; load of the MPYer registers
MPYU16 R4,R6              ; (R4) x (R6) -> Sum
EINT                      ; Allow interrupts again
...                      ; Continue with result
; The interrupt handler must save and restore the Sum registers
;
INTRPT_H  PUSH    &SumLo      ; Save the SumLo register
          PUSH    &SumHi      ; Save the SumHi register
          PUSH    &SumExt     ; Save the SumExt register
          MPYU16 #X,C1        ; Call unsigned MPY: X x C1
          ....               ; Continue with MPYer result
          POP     &SumExt     ; Restore SumExt register
          POP     &SumHi      ; SumHi register
          POP     &SumLo      ; SumLo register

```

```
RETI ; Return to background
```

3.3 Speed Comparison with Software Multiplication

Table 4 shows the speed increase for the four different 16×16-bit multiplication modes. The software loop cycles include the subroutine call (CALL #MULxx), the multiplication subroutine itself, and the RET instruction. Only CPU registers are used for the multiplication. See the *Metering Application Report* for details on the four multiplication subroutines.

The cycles given for the hardware multiplier include the loading of the multiplier operands Operand1 and Operand2 from CPU registers and—in the case of the signed MAC operation—the accumulation of the 48-bit result into three CPU registers (see section 3.1.2).

Table 4. CPU Cycles Needed With Different Multiplication Modes

| OPERATION | SOFTWARE LOOP | HARDWARE MPYer | SPEED INCREASE |
|-----------------------|---------------|----------------|----------------|
| Unsigned Multiply MPY | 139...171 | 8 | 17.4...21.4 |
| Unsigned MAC | 137...169 | 8 | 17.1...21.1 |
| Signed Multiply MPYS | 145...179 | 8 | 18.1...22.4 |
| Signed MAC | 143...177 | 17 | 8.4...10.4 |

3.4 Software Hints

Operand1 can be used in consecutive multiplications without having to move it again into the Operand1 register. The first example shows two unsigned multiplications with the content of address TONI. This method saves four bytes and six CPU cycles over the normal procedure.

```
; Multiply TONI x R6 and TONI x R5. Results to diff. locations
;
MPYU16    TONI,R6          ; TONI x R6 -> SumHi|SumLo
MOV       &SumLo,R7        ; Result to R8|R7
MOV       &SumHi,R8
MOV       R5,&0138h         ; TONI still in &0130h
MOV       &SumLo,RESULT     ; TONI x R5 -> SumHi|SumLo
MOV       &SumHi,RESULT+2   ; Result to RESULT+2|RESULT
```

The second example shows three multiply-and-accumulate operations with the same Operand1. The three *Operand2s* cannot be simply added and multiplied once because their sum may exceed the 16-bit range. This method saves eight ROM bytes and twelve CPU cycles over the normal procedure.

```
; Multiply-and-accumulate TONI x R6, TONI x R5 and TONI x EDE
; The accumulated result is moved to RESULT..RESULT+4
;
... ; Initialize SumXxx registers
MACU16   TONI,R6          ; TONI x R6 + SumHi|SumLo
ADD      &SumExt,RESULT+4 ; Add carry to extension
MOV      R5,&0138h         ; Add TONI x R5 to SumXxx
ADD      &SumExt,RESULT+4 ; Add carry to extension
```

```

MOV      EDE,&0138h      ; Add TONI x EDE to SumXxx
MOV      &SumLo,RESULT    ; TONI x (R5+R6+EDE) in SumXxx
MOV      &SumHi,RESULT+2; Result to RESULT..RESULT+4
ADD      &SumExt,RESULT+4

```

3.5 Speed Increase With Floating Point Package FPP4

The hardware multiplier increases the speed of the floating-point multiplication only. The variables X and Y are used in the speed evaluation shown. They are defined as follows:

```

        .if DOUBLE=0      ; 32-bit format
X .float  3.1416          ; 3.1416
Y .float  3.1416*100      ; 314.16
        .else             ; 48-bit format
X .double 3.1416          ; 3.1416
Y .double 3.1416*100      ; 314.16
        .endif

```

The execution cycles shown include the addressing of one operand and the subroutine CALL:

```

MOV      #X,RPRES        ; Address 1st operand
MOV      #Y,RPARG        ; Address 2nd operand
CALL     #FLT_MUL         ; Call the MPY subroutine
....
        ; Product X x Y on TOS

```

Table 5 shows the number of cycles required by the multiplication.

Table 5. CPU Cycles Required for FPP Multiplication (FLT_MUL)

| OPERATION | .FLOAT | .DOUBLE | COMMENT |
|---|--------|---------|---------------------|
| Multiplication $X \times Y$ | 395 | 692 | Software Loop |
| Multiplication $X \times Y$ | 153 | 213 | Hardware MPYer used |
| Speed Increase | 2.58 | 3.25 | SW-cycles/HW-cycles |

Substituting divisions with multiplications whenever possible to take advantage of the hardware multiplier's speed is recommended. The simplest case is when dividing by a constant as the next example illustrates.

EXAMPLE: division of the last result, sitting on top of the stack, by the constant 2.7182818 is substituted with a multiplication by the constant $1/2.7182818$. This reduces calculation time by a factor of $405/153 = 2.65$. First let's look at the original sequence:

```

DOUBLE .equ 0      ; Use the .FLOAT format
HW_MPY .equ 1      ; Use the HW-MPYer
...
MOV     #FLTe,RPARG ; Address constant e
CALL    #FLT_DIV    ; TOS/e: Division 405 cycl.
....
        ; Quotient on TOS
FLTe    .float 2.7182818 ; Constant e

```

Now lets look at the same calculation performed by the hardware multiplier and substituting division with multiplication:

```
HW_MPY      .equ      1              ; Use the HW-MPYer
...
      MOV      #FLT_ei,RPARG ; Address constant 1/e
      CALL     #FLT_MUL      ; TOS x 1/e. MPY 153 cycles
      ....      ; Result on TOS
FLT_ei      .float     0.3678794     ; Constant 1/e
```

If the .DOUBLE version (48 bits) of FPP4 is used, the division's execution time is decreased by a factor of $756/213 = 3.55$.

4 Software Applications

Typical application examples using the hardware multiplier are given in this section. The comments indicate the location of the decimal (or equivalent hexadecimal) point:

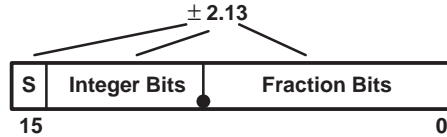


Figure 3. Multiplication Exceeding 16 Bits

4.1 Multiplication Exceeding 16 Bits

The first software example is the unsigned multiplication of two 40-bit numbers. The most significant bytes contain 0, with 48 bits of the result used afterwards, and the lower 32 bits of the product unused. The first operand is contained in register ARG1_xxx, and the second operand in register ARG2_xxx. The result is placed into RESULT_xxx (CPU registers or RAM). The multiply routine is taken out of the FPP4 package.

The execution time for CPU registers is 94 cycles.

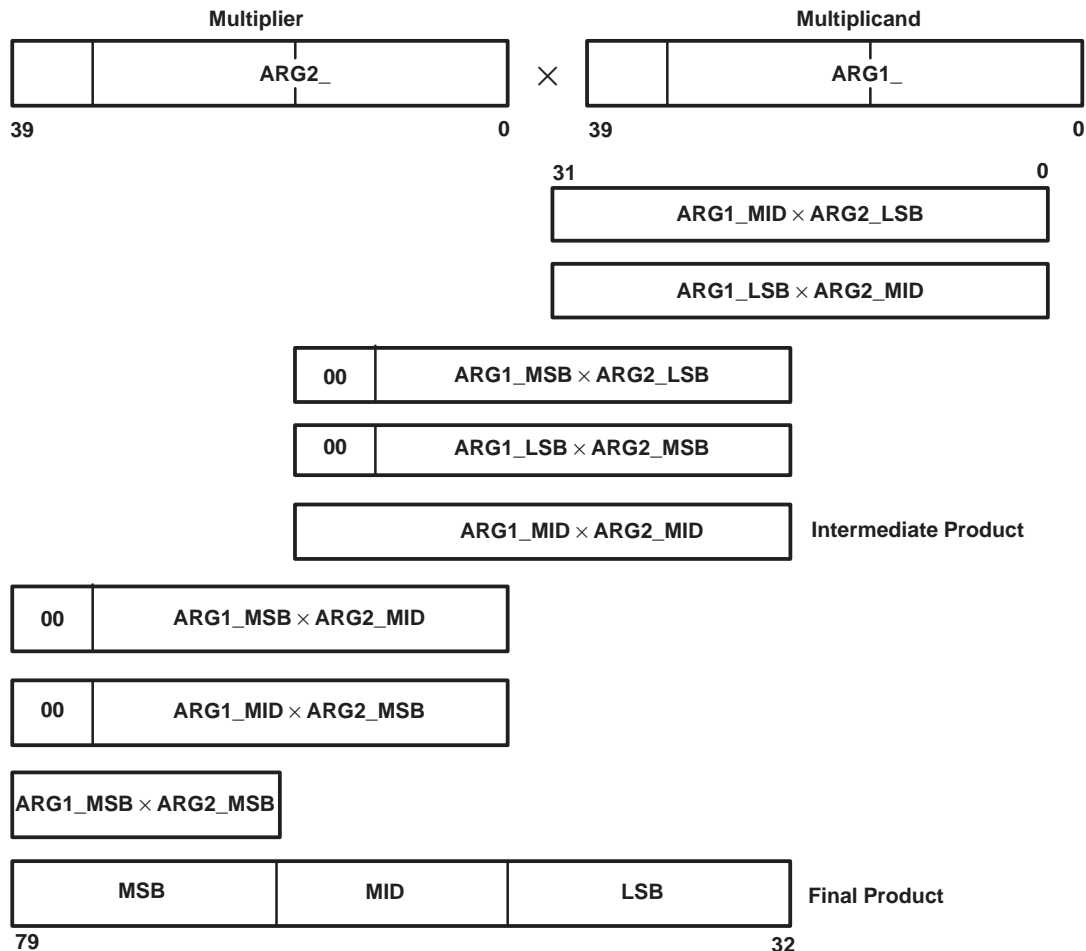


Figure 4. 40×40-Bit Unsigned Multiplication MPYU40

```
; Register Definitions for the 40 x 40 unsigned MPY and MAC
;
ARG1_MSB      .equ   R5      ; Argument 1 (Multiplicand)
ARG1_MID      .equ   R6
ARG1_LSB      .equ   R7
ARG2_MSB      .equ   R8      ; Argument 2 (Multiplier)
ARG2_MID      .equ   R9
ARG2_LSB      .equ   R10
RESULT_MSB    .equ   R11     ; Result (Product)
RESULT_MID    .equ   R12
RESULT_LSB    .equ   R13
;
MPYU40 CLR     RESULT_MSB    ; Clear Result
        CLR     RESULT_MID
        CLR     RESULT_LSB
;
MACU40 MPYU16 ARG2_LSB,ARG1_MID ; Bits 16 to 47
        MACU16 ARG1_LSB,ARG2_MID
        ADD     &SumHi,RESULT_LSB
        ADDC    &SumExt,RESULT_MID
MPYU16 ARG1_MSB,ARG2_LSB ; Bits 32 to 63
        MACU16 ARG1_LSB,ARG2_MSB
        MACU16 ARG1_MID,ARG2_MID
        ADD     &SumLo,RESULT_LSB
        ADDC    &SumHi,RESULT_MID
        ADDC    &SumExt,RESULT_MSB
MPYU16 ARG1_MSB,ARG2_MID ; Bits 48 to 79
        MACU16 ARG2_MSB,ARG1_MID
        ADD     &SumLo,RESULT_MID
        ADDC    &SumHi,RESULT_MSB
MPYU16 ARG1_MSB,ARG2_MSB ; Bits 64 to 79
        ADD     &SumLo,RESULT_MSB
RET                                     ; 48 MSBs in result
```

The second software example shows all four possible multiplication routines for two 32-bit numbers. The full 64-bit result may be used afterwards. The signed 16×16-bit hardware multiplication MPYS cannot be used, since it is designed for the special case of 16×16 bits. Therefore, unsigned multiplication MPY is used with a correction on the final sum at the start of the subroutine.

| | | |
|---------------------------------|------------------------|--------------|
| Execution times (without CALL): | MACU32 58 cycles | unsigned MAC |
| | MPYU32 64 cycles | unsigned MPY |
| | MACS32 64 to 68 cycles | signed MAC |
| | MPYS32 68 to 72 cycles | signed MPY |

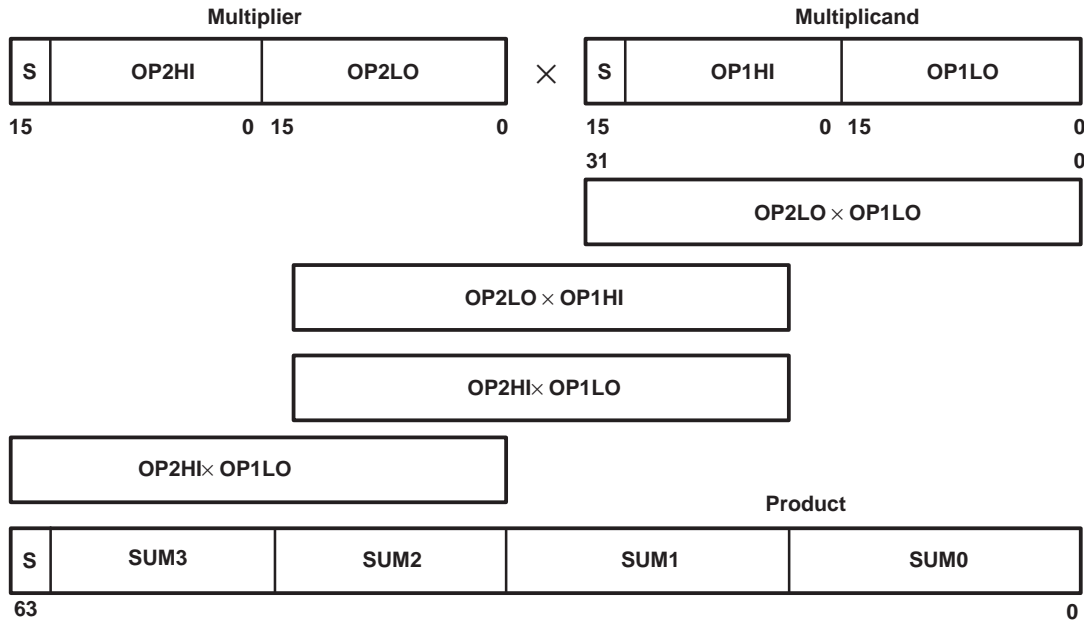


Figure 5. 32×32-Bit Signed Multiplication MPYS32

All four possible 32×32-bit multiplication modes and MAC functions are shown below. The specified operands and result registers may be working registers (as defined) or RAM locations.

```

SUM3 .equ R15 ; Result: sign and MSBs
SUM2 .equ R14 ; (registers or RAM locations)
SUM1 .equ R13
SUM0 .equ R12 ; LSBs
OP1HI .equ R11 ; 1st operand: sign and MSBs
OP1LO .equ R10 ; LSBs
OP2HI .equ R9 ; 2nd operand: sign and MSBs
OP2LO .equ R8 ; LSBs
;
; The unsigned 32 x 32 bit multiplication
;
MPYU32 CLR SUM3 ; Clear the result registers
        CLR SUM2 ; 64 cycles
        CLR SUM1
        CLR SUM0
        JMP MS321 ; Proceed at common part
;
; The signed 32 x 32 bit multiplication
;
MPYS32 CLR SUM3 ; Clear the result registers
        CLR SUM2 ; 68 to 72 cycles
        CLR SUM1
        CLR SUM0

```

```

;
; The signed 32-bit "Multiply-and-Accumulate" subroutine
; The final result is corrected. 64 to 68 cycles
;
MACS32 TST    OP1HI          ; Operand1 negative?
        JGE    MS320          ; No
        SUB    OP2LO,SUM2     ; Yes, correct final sum
        SUBC   OP2HI,SUM3
MS320 TST    OP2HI          ; Operand2 negative?
        JGE    MS321          ; No
        SUB    OP1LO,SUM2     ; Yes, correct final sum
        SUBC   OP1HI,SUM3
;
; The unsigned 32-bit "Multiply-and-Accumulate" subroutine
;
MACU32 .equ   $              ; 58 cycles
;
; Main part for all multiplication subroutines
;
MS321 MPYU16 OP1LO,OP2LO     ; LSBs x LSBs
        ADD    &SumLo,Sum0    ; Add product to result
        ADDC   &SumHi,Sum1
;
        ADC    Sum2           ; Necessary only for MACx32
        ADC    Sum3           ;
;
        MPYU16 OP1LO,OP2HI    ; LSBs x MSBs
        MACU16 OP2LO,OP1HI    ; LSBs x MSBs
        ADD    &SumLo,Sum1    ; Add accumulated products
        ADDC   &SumHi,Sum2    ; to result
;
        ADDC   &SumExt,Sum3    ; Necessary only for MACx32
;
        MPYU16 OP1HI,OP2HI    ; MSBs x MSBs
        ADD    &SumLo,Sum2    ; Add product to final result
        ADDC   &SumHi,Sum3
        RET

```


4.2 Sensor Characteristics

Many applications use digital values delivered by analog-to-digital converters, I/O-ports, or calculation results that may require correction or conditioning. This is normally accomplished through the use of polynomials. For example a cubic polynomial to calculate the corrected output value y from an input value x is:

$$y = a_3x^3 + a_2x^2 + a_1x + a_0$$

The following subroutine illustrates a common solution using the hardware multiplier. To attain maximum speed, the coefficients a_3 to a_0 have decreasing number of bits after the decimal (or equivalent hexadecimal) point. If this tolerance is not acceptable, then shifts and stores between multiplications may be necessary. The input x remains in Operand1 (MPYS 0132h) and is used in all three multiplications.

Example: the output of an ADC is corrected using a cubic polynomial. All values are normalized to be less than 1.0 to achieve maximum resolution. The coefficients a_n used for correction are:

$$a_3: +0.01 \quad a_2: -0.25 \quad a_1: -0.5 \quad a_0: +0.999$$

The Horner scheme is used for the computation:

$$y = (((a_3x) + a_2)x + a_1)x + a_0$$

The numbers $\pm b.a$ in the comments indicate the bits before and after the decimal point of the used numbers.

Execution time (without CALL): 45 cycles

```

;
; Polynomial Calculation for y = a3x^3 + a2x^2 + a1x + a0
; Result in SumHi register
;
POLYNOM    MPYS16    X,A3          ; +0.15 x +0.15 (+1.14)
           ADD       A2,&SumHi     ; ±1.14 + ±1.14 ≥ ±1.14
           MOV       &SumHi,&OP2   ; ±1.14 x ±0.15 (±2.13)
           ADD       A1,&SumHi     ; ±2.13 + ±2.13 ≥ ±2.13
           MOV       &SumHi,&OP2   ; ±2.13 x ±0.15 (±3.12)
           ADD       A0,&SumHi     ; ±3.12 + ±3.12 ≥ ±3.12
           RET                          ; SumHi: ±3.12
;
; Table of coefficients
;
A3    word    +100*08000h/10000    ; +0.01    (±0.15)
A2    .word   -2500*04000h/10000   ; -0.25    (±1.14)
A1    .word   -5000*02000h/10000   ; -0.5     (±2.13)
A0    .word   +9999*01000h/10000   ; +0.9999  (±3.12)

```

4.3 Table Calculations

The .MACRO instructions used on the different multiplication options (8 or 16-bit, signed and unsigned, multiply, and multiply-and-accumulate) allow the use of all seven addressing modes of the MSP430 architecture on source and destination. Therefore the MPY instructions are perfectly adapted to table processing where indirect addressing can be used on both operands of a multiply instruction. An example of table calculation is given in Section 4.5.

4.4 Wave Digital Filters

The main advantage of wave digital filters is that no multiplication is required with fixed coefficients. The filter algorithm uses an optimized shift-and-add sequence. This optimization is not possible when adaptive filter algorithms are used, unless the coefficients are changed. In this case a hardware multiplier has big advantages as the calculation time is independent of the coefficients used.

4.5 Finite Impulse Response (FIR) Digital Filter

The formula for a simple FIR filter is:

$$y_n = a_0 x_n + a_1 x_{n-1} + a_2 x_{n-2} \dots + a_k x_{n-k}$$

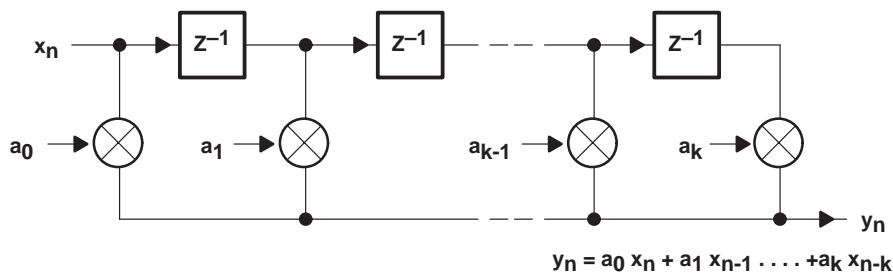


Figure 6. Finite Impulse Response Filter

The example below shows an algorithm that uses the last ADC result as the input to a seventh-order finite impulse response (FIR) filter. The coefficients a_n are stored in ROM (fixed coefficients) or in RAM (adaptable coefficients). The filter may be easily changed to a higher order as follows:

- Change the value k to the desired order.
- Allocate $(k+1)$ RAM words for the input samples x_n starting at label X .
- Enlarge the table with the a_n coefficients to $(k+1)$ coefficients.

Execution time: 28 CPU cycles are necessary per filter tap.

The example does not show an actual filter. A linear phase-response requires the following a_n coefficients:

$$a_n = a_{k-n}$$

which means: $a_0 = a_k$, $a_1 = a_{k-1}$, and so on.

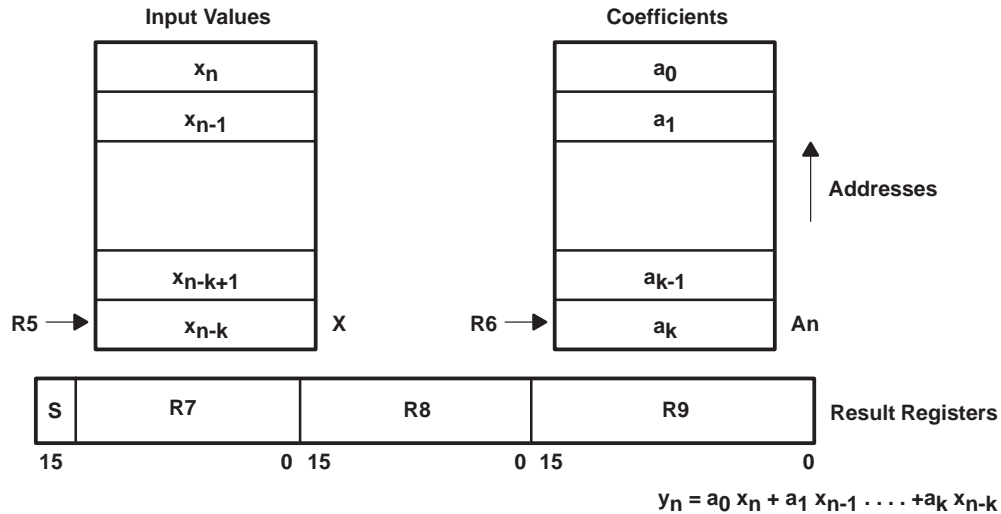


Figure 7. Finite Impulse Response Filter Storage

```

; The special "Multiply-and-Accumulate" .MACRO accumulates the
; products X x An in the registers R7|R8|R9.
; Execution time: 19 cycles for the example below with the
; indirect addressing mode used for both operands.
;
MACS16 .MACRO arg1,arg2      ; Signed MAC 16x16
    MOV    arg1,&0132h      ; Signed MPY is used
    MOV    arg2,&0138h      ; Start MPYS
    ADD    &SumLo,R9        ; Add LSBs to result
    ADDC   &SumHi,R8        ; Add MSBs to result
    ADDC   &SumExt,R7       ; Add SumExt to result
    .ENDM                    ; Result in R7|R8|R9

; Definitions:
; - Value k defines the order of the FIR-filter
; - OFFSET is used to get signed values (E000h..1FFFh) out of
;   the unsigned 14-bit ADC results (0...3FFFh)
; - X defines the address for the oldest input sample x(n-k)
;   in a sample buffer with (k+1) words length
;
k          .equ    7          ; (k + 1)samples are used -
OFFSET     .equ    02000h     ; to get signed ADC values
X          .equ    0200h      ; x(n-k) sample address
;

; With the Timer_A interrupt the calculation is made
;
TIMA_INT   PUSH    R5          ; Save R5 and R6
           PUSH    R6
           MOV     #X,R5       ; Address xn buffer (oldest x)

```

```

MOV    #An,R6           ; Address an constants (ak)
MOV    &ADAT,2*k(R5)     ; New ADC sample to xn
SUB    #OFFSET, 2*k(R5) ; Create signed value for xn
CLR    R7               ; Clear result reg. (MSBs)
CLR    R8
CLR    R9
TA00   MACS16 @R5+,@R6+   ; ak * xn-k added to R7|R8|R9
MOV    @R5,-2(R5)        ; xn-k+1 -> xn-k
CMP    #X+2+(2*k),R5     ; Through? (R5 points outside)
JNE    TA00              ; No, once more
POP    R6                ; Restore R5 and R6
POP    R5
BIS    #CS,&ACTL         ; Start next ADC conversion
RETI                     ; Result: ±17.30 (3 words)
;
; The constants An are fixed in ROM. Format: ±0.15
; (1 bit sign, 15 bits fraction)
; Range: -0.99996 to +0.99996
;
An      .word +9999*8000h/10000 ; ak      +0.9999
        .word -9999*8000h/10000 ; ak-1    -0.9999
        ...                    ; ak-2 to a2
        .word +5000*8000h/10000 ; a1      +0.5
        .word -5000*8000h/10000 ; a0      -0.5

```

4.6 Fast Fourier Transform Algorithm

The RAM buffer pointed by pQR is transformed and overwritten with the result of the fast Fourier transformation (FFT). The formula used for each block consisting of real and imaginary numbers is:

$$\begin{aligned}
 \text{PRi}' &= (\text{PRi} + (\text{QRi} \times \text{WRi} + \text{Qli} \times \text{Wli}))/2 && \text{real part of Pi} \\
 \text{Pli}' &= (\text{Pli} + (\text{Qli} \times \text{WRi} - \text{QRi} \times \text{Wli}))/2 && \text{imaginary part of Pi} \\
 \text{QRi}' &= (\text{PRi} - (\text{QRi} \times \text{WRi} + \text{Qli} \times \text{Wli}))/2 && \text{real part of Qi} \\
 \text{Qli}' &= (\text{Pli} - (\text{Qli} \times \text{WRi} - \text{QRi} \times \text{Wli}))/2 && \text{imaginary part of Qi}
 \end{aligned}$$

Where:

| | |
|----------|---|
| WRi | $\cos(i \times 2\pi/N) = \cos(\omega \times i)$ |
| Wli | $\sin(i \times 2\pi/N) = \sin(\omega \times i)$ |
| ω | $2\pi f$ |
| i | Index number |
| PRi | Real part of PRi before FFT |
| PRi' | Real part of PRi after FFT |

Figure 7 shows the allocation of the three tables in MSP430's RAM and ROM.

Execution time: the buffer with eight complex numbers each for the P and Q parts requires 717 cycles (without CALL) for the transformation (185 μ s at 4 MHz).

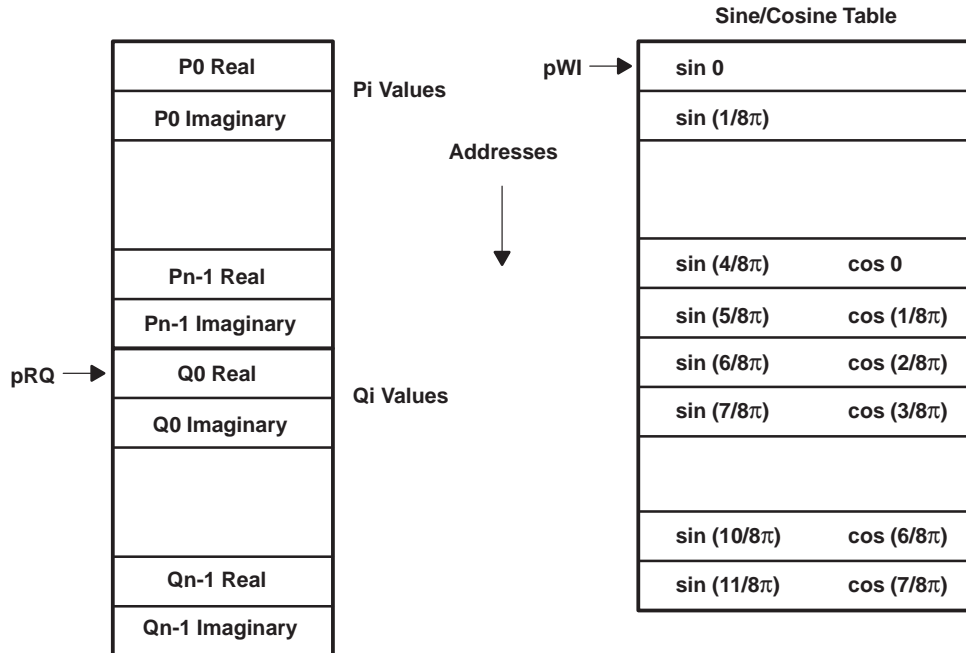


Figure 8. RAM and ROM Allocation for the Fast Fourier Transform Algorithm

```

; Algorithm: 'FFT' optimized butterfly radix 2 for MSP430x33x
;
; Originally developed by M.Christ/TID for TMS320C80
;
; Input data: PR0,PI0,PR1,PI1,.....,QRn-1,QIn-1 (16 bit words)
;
;   Algorithm:
;
;   PR' = (PR+(QR*WR+QI*WI))/2   WR=cos(wt)
;   PI' = (PI+(QI*WR-QR*WI))/2   WI=sin(wt)
;
;   QR' = (PR-(QR*WR+QI*WI))/2
;   QI' = (PI-(QI*WR-QR*WI))/2
;
;   Procedure:
;
;   real = (QR*WR+QI*WI)/2
;   imag = (QI*WR-QR*WI)/2
;
;   PR' = PR/2 + real
;   QR' = PR/2 - real
;
;   PI' = PI/2 + imag
;   QI' = PI/2 - imag

```

```

N      .equ    16      ; 16 point complex FFT
N2     .equ    N*2     ; Byte count (QR - PR)
pQR    .equ    R5      ; Pointer to QRi
pWI    .equ    R6      ; Pointer to sine table tabsin
real   .equ    R7      ; Storage  QR x WR + QI x WI
imag   .equ    R8      ; Storage  QI x WR + QR x WI
TEMP   .equ    R9      ; Temporary storage
TEMP1  .equ    R10     ;
;
; The subroutine FFT is called after the loading of the
; pointer to QR0.
;
; Call:  MOV    #QR,pQR  ; Pointer to QR0 of block (RAM)
;        CALL  #FFT     ; Call the FFT subroutine
;        ...           ; Input table contains results
;
; Definition of the input table located in the RAM
;
        .bss    PR,2,0200h  ; PR0  Preal
        .bss    PI,2        ; PI0  Pimaginary
        .bss    PRi,N2-4    ; PR1, PI1...PRn-1, PIn-1
        .bss    QR,2        ; QR0  Qreal
        .bss    QI,2        ; QI0  Qimaginary
        .bss    QRi,N2-4    ; QR1, QI1 ...QRn-1, QIn-1
;
; Start of the FFT subroutine. pQR contains address of QR0
;
FFT     MOV     #tabsin,pWI  ; Pointer to sin 0
;
; Execution of the 4 multiplications. The halfed result is
; calculated without additional shifts due to the format 2.14
; Calculation of the real part: real = (QR x WR + QI x WI)/2
;
FFTLOP  MPYS16 @pQR+,tabcos-tabsin(pWI);
        MOV     &SumHi,real  ; Store (QR x WR)/2 (2.14)
        MPYS16 @pQR,@pWI    ; (QI x WI)/2    (2.14)
        ADD     &SumHi,real  ; Store real part
;
; Calculation of the imaginary part:
; imag = (QI x WR - QR x WI)/2
;
        MPYS16 @pQR+,tabcos-tabsin(pWI);
        MOV     &SumHi,imag  ; Store (QI x WR)/2 (2.14)

```

```

        MPYS16 -4(pQR),@pWI+      ; (QR x WI)/2      (2.14)
        SUB    &SumHi,imag        ; Store imaginary part
;
; Calculation of PR', PI', QR', QI'. pQR points to QRi+1
; Calculation of PR': PR' = (PR + (QR x WR + QI x WI))/2
;
        MOV    -N2-4(pQR),TEMP    ; PRi to TEMP
        RRA    TEMP                ; PRi/2
        MOV    TEMP,TEMP1         ; Copy PRi/2
        ADD    real,TEMP1         ; PRi/2 + (QRxWR + QIxWI)/2
        MOV    TEMP1,-N2-4(pQR)   ; to PR'   (1.15)
;
; Calculation of QR': QR' = (PR - (QR x WR + QI x WI))/2
;
        SUB    real,TEMP          ; PR/2 - (QRxWR + QIxWI)/2
        MOV    TEMP,-4(pQR)       ; to QR'   (1.15)
;
; Calculation of PI': PI' = (PI + (QI x WR - QR x WI))/2
;
        MOV    -N2-2(pQR),TEMP    ; PI
        RRA    TEMP                ; PI/2
        MOV    TEMP,TEMP1         ; Copy PI/2
        ADD    imag,TEMP1         ; PI/2 + (QIxWR - QRxWI)/2
        MOV    TEMP1,-N2-2(pQR)   ; to PI'   (1.15)
;
; Calculation of QI': QI' = (PI - (QI x WR - QR x WI))/2
;
        SUB    imag,TEMP          ; PI/2 - (QI*WR+QR*WI)/2
        MOV    TEMP,-2(pQR)       ; to QI'   (1.15)
;
; To next input data. Check if FFT is finished
;
        CMP    #tabsin0,pWI       ; Through? (pWI = tabsin0)
        JLO    FFTLOP             ; No
        RET                     ; Yes, return
;
; Sine and cosine table. Format: s.fraction (1.15)
;
tabsin    .word +0000*8000h/10000  ; sin 0.0 = 0.00000
          .word +3827*8000h/10000  ; sin  $\pi/8$  = 0.38268
          .word +7071*8000h/10000  ; sin  $2\pi/8$  = 0.70711
          .word +9239*8000h/10000  ; sin  $3\pi/8$  = 0.92388
tabcos    .word 10000*8000h/10000-1 ; sin  $4\pi/8$  cos 0.0

```

```
.word +9239*8000h/10000 ; sin 5 $\pi$ /8 cos  $\pi$ /8
.word +7071*8000h/10000 ; sin 6 $\pi$ /8 cos 2 $\pi$ /8
.word +3827*8000h/10000 ; sin 7 $\pi$ /8 cos 3 $\pi$ /8
tabsin0 .word +0000*8000h/10000 ; cos 4 $\pi$ /8
.word -3827*8000h/10000 ; cos 5 $\pi$ /8
.word -7071*8000h/10000 ; cos 6 $\pi$ /8
.word -9239*8000h/10000 ; cos 7 $\pi$ /8
;
; An example is given for the FFT:
; The following table contains 32 values that are the data
; for the FFT
; 16 point complex FFT radix 2 DIT
;
DataSt .word 014abh,02e90h,0f6d4h,005d3h ; PR0,PI0..PI1
.word 004b2h,0fecdh,0f78ch,0fcb2h ; PR2,PI2..PI3
.word 0093ch,004f0h,0ffb5h,0017ch ; PR4,PI4..PI5
.word 0fbcbh,002a5h,0f3a3h,0fb38h ; PR6,PI6..PI7
.word 01854h,02a29h,0ffb9h,0f9beh ; QR0,QI0..QI1
.word 0fa49h,00907h,00a10h,0f99bh ; QR2,QI2..QI3
.word 0030ch,0fdadh,0fa2ah,002e3h ; QR4,QI4..QI5
.word 0fddbh,0029bh,0fdf9h,00225h ; QR6,QI6..QI7
;
; The following 32 values are output by the FFT
;
Result .word 0167fh,02c5ch,0fa16h,00013h ; PR'0..PI'1
.word 00384h,0049ch,0fabeh,0f879h ; PR'2..PI'3
.word 00374h,000f2h,0024dh,002e2h ; PR'4..PI'5
.word 0ffa4h,00128h,0fb2ah,0fd01h ; PR'6..PI'7
.word 0fe2bh,00233h,0fcbdh,005bfh ; QR'0..QI'1
.word 0012dh,0fa30h,0fccdh,00438h ; QR'2..QI'3
.word 005c7h,003fdh,0fd67h,0fe99h ; QR'4..QI'5
.word 0fc47h,0017ch,0f879h,0fe36h ; QR'6..QI'7
```


5 Conclusion

As shown by the examples, the hardware multiplier offers its biggest advantages when used with signed and unsigned 16-bit operands, as well as in other applications using 8-bit operands, 32-bit operands, and floating point numbers. The speed increase is significant compared to a pure software solution.

6 References

1. *MSP430 Family Architecture Guide and Module Library*, 1996, Literature #SLAUE10B
2. *MSP430 Metering Application Report*, 1997, Literature #SLAAE10B
3. *Data Sheet MSP430C336, MSP430P337*, 1997, Literature #SLASE10A
4. *A Simple Approach to Digital Signal Processing*, 1995, Literature #SPRDE01A

