H8/300 Family

Application Note
16x16 Multiply

Tom Hampton

INTRODUCTION

The H8/300 CPU core is very powerful considering that it is an 8-bit architecture. Part of its power comes from the flexible instruction set, which allows for many byte and word operations. Part of its power also comes from the ability of the architecture to allow use of the general purpose register as either 16-bit or 8-bit registers as needed. While the instruction set is extremely powerful in its ability to handle bit-wide and byte-wide data, it only has a small number of instructions (other than data transfer) that allow operations on word-wide data.

One of the instructions that is noticeably missing is in the arithmetic area. While the CPU has the capability of doing an 8x8 unsigned multiply, it lacks the capability of doing a 16x16 unsigned multiply. Even though this instruction does not exist, the function can be easily implemented using the instructions currently available.

In this application note, we will examine a routine that provides the user with a 16x16 unsigned multiply function as well as perform it in a very short amount of time. In performing this operation, we will make use of the flexible instruction set as well as the architecture's flexibility to be used as either byte-wide or word-wide registers. Only five general purpose registers are used in this routine, including the two that pass the parameters. No fancy tricks were used, but it was important to pay special attention to the manner in which the multiply instruction operated on the registers.

This instruction requires that the destination operand be contained in the lower half of a 16-bit register even though the entire register will be used to hold the results. For this reason, it was necessary to use other register for working registers and temporary storage.

MULTIPLICATION PROCEDURES

STANDARD PROCEDURE

If one examines the normal procedure of a 16-bit multiply operation (see Figure 1), it would be easy to see the steps that would be taken if the H8/300 CPU had a 16-bit unsigned multiply instruction. The first step would be to multiply the 16-bits of one operand by the lower "digit" of the second operand, thus potentially yielding a 24-bit response. The second step would be to multiply the same 16-bit one operand by the higher "digit" of the second operand, which could yield yet another 24-bit response. The final step would be to add these 24-bit responses together in the proper sequence (with required shifting) to form a 32-bit result. This is the procedure that we are all used to for multiplication.

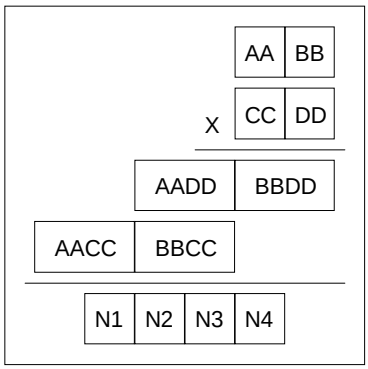

Figure 1: Standard Multiplication Procedure

MODIFIED PROCEDURE

Since the H8/300 CPU does not have a 16-bit multiply instruction, the normal procedure cannot be used. Instead we must modify the procedure to account for the creation of only 16-bit intermediate results (see Figure 2). In this procedure, we must multiply the individual 8-bit "pieces" of the operand to form intermediate results. This requires four steps since we actually have four bytes of operand data that we must multiply together. The results of these pieces of intermediate data must then be added together in the proper sequence (with shifting) to form the final 32-bit result.

SOFTWARE DESCRIPTION

The routine written to perform the 16x16 unsigned multiply function is shown in Listing 1. You may wish to refer to this listing during the following discussions. The routine occupies only 38 bytes of code space while executing in 8.6 µsec. One of the first things to note in this routine is that no registers are saved even though some of the general purpose registers are

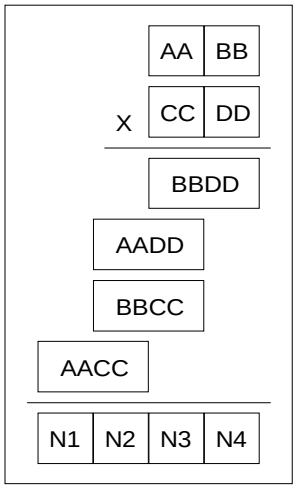

Figure 2: Modified Multiplication Procedure

used for performing the function. If the user decides it is important to save the current state of the working registers, then it is easy to add "push" instructions at the beginning of the routine to save the data, and "pop" instructions at the end of the routine to restore the data. As we discuss this routine, we will examine the modified procedure in detail as well as an example of data used in the execution of the routine.

Before the routine is called, the user must place the two 16-bit operands to be multiplied in registers R1 and R2 (see Figure 3). For this discussion, R1 will contain the "multiplier" and R2 will contain the "multiplicand." The result will be returned in these same registers, so if the original operands are to be used later, it is also up to the user to save them elsewhere. In our example, we will use the data H'37DF and H'40FF for the multiply routine.

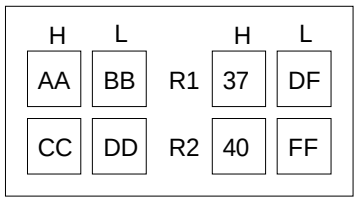

Figure 3:Parameter Passing

The first step we must perform in this routine is to prepare the destination working registers (R3 and R4, only R4 requires clearing) and also to save the multiplicand into a temporary register (R5) because we will need it again later (see Figure 4). This is performed with the following instructions:

```
mov.w     #0,r4
mov.w     r2,r5
```

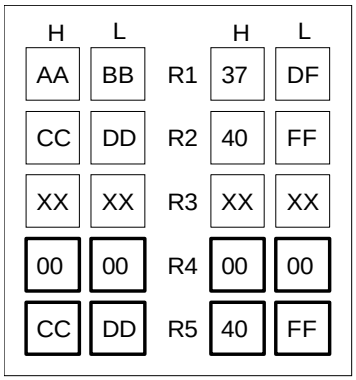

Figure 4: Step 1

In the next step, we perform the multiplication of the two low-bytes of the multiplier and multiplicand (see Figure 5). This result (H'DE21), which exists in R2, is then placed into our destination registers (R3 and R4, only R3 is required at this time). This is performed with the following instructions:

```
mulxu     r1l,r2
mov.w     r2,r3
```

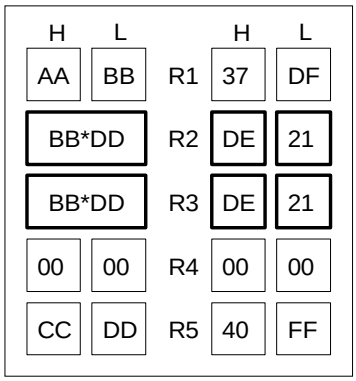

Figure 5: Step 2

Our original multiplicand that used to be in R2 is no longer valid since R2 has been corrupted because of the multiply instruction. In the third step (see Figure 6), we must retrieve the high-byte of the saved multiplicand and place it into the lower half of register R2. Remember that the destination operand must exist in the lower half of the register in order for the multiply instruction to execute. This is performed with the following instruction:

```
mov.w     r5,r2
```

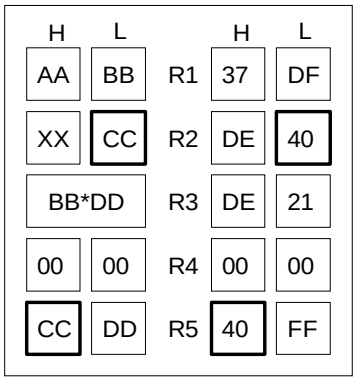

Figure 6: Step 3

The fourth step now multiplies the upper half of the multiplicand by the lower half of the multiplier. This result (H'37C0) is added to the previous result (see Figure 7), but not directly to it since some shifting of the results must be performed. We must add the low byte of this result (R2L) with the high byte of the previous result (R3H). We must then add the high byte of this result with the carry-over from the previous addition and place the result in R4L. This generates a 32-bit result of H'00389E21 (H'37C000 + H'DE21). This is performed with the following instructions:

```
mulxu     r1h,r2
add.b     r2l,r3h
addx      r2h,r4l
```

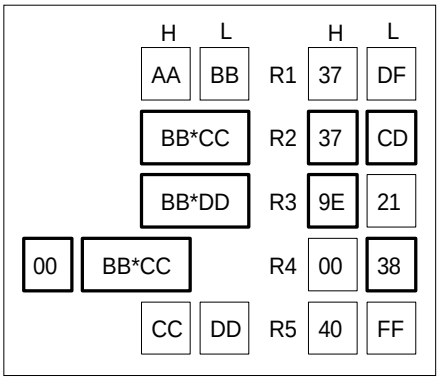

Figure 7: Step 4

We have now performed an 8x16 multiply function, but that is not what we wanted to do, but we are halfway through. Again our original multiplicand that was in R2 is no longer valid, so we must retrieve it from storage (see Figure 8). This is performed with the following instruction:

```
mov.w     r5,r2
```

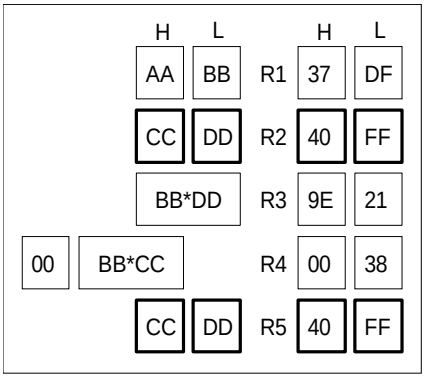

Figure 8: Step 5

The sixth step now multiplies the upper half of the multiplier with the lower half of the multiplicand. The result of this operation (H'36C9) is also added to the previous result, again with some shifting performed (see Figure 9). We must add the low byte of this result (R2L) with the high byte of the previous result (R3H). We must then add the high byte of this result with the carry-over from the previous addition and the current value in R4L, and place the result in R4L. The carry-over from this addition is placed into R4H. This alters our previous 32-bit result to be H'006F6721 (H'00389E21 + H'36C900). This is performed with the following instructions:

```
mulxu     r1h,r2
add.b     r2l,r3h
addx      r2h,r4l
addx      #0,r4h
```

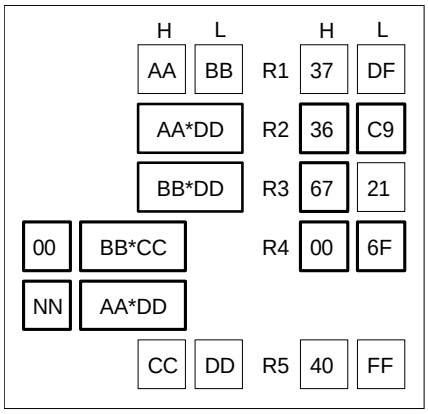

Figure 9: Step 6

Again our original multiplicand that was in R2 is no longer valid, so we must retrieve it from storage (see Figure 10). We must retrieve the high-byte of the saved multiplicand and place it into the lower half of register R2. Remember that the destination operand must exist in the lower half of the register in order for the multiply instruction to execute. This is performed with the following instruction:

```
mov.b     r5h,r2l
```

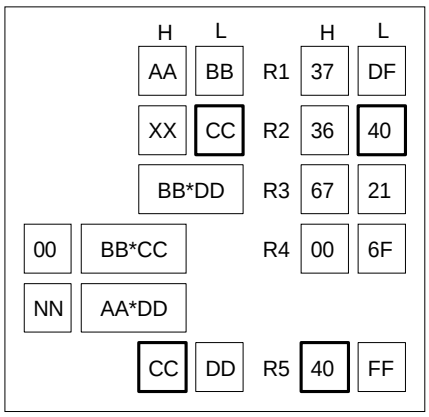

Figure 10 Step 7

In the eigth step (the final one of our multiplcation itself) we multiply the upper byte of the multiplicand by the upper byte of the multiplier (see Figure 11). This result (H'0DC0) is then added to the upper word (R4) of the previous results to provide our final answer, H'0E2F6721 (H'006F6721 + H'0DC00000). At this time we also move the result to registers R1 and R2 for return to the calling program (see Figure 12). This is performed by the following instructions:

```
mulxu     r1h,r2
add.w     r4,r2
mov.w     r3,r1
rts
```

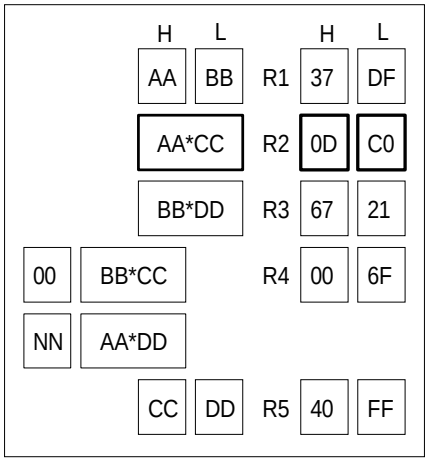

Figure 11: Step 8a

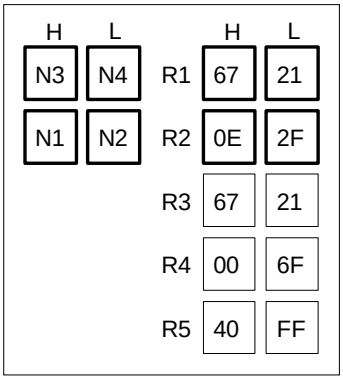

Figure 12: Step 8b

Listing 1: 16x16 Multiply Routine

```
*** H8/300 ASSEMBLER               VER 1.1 ***   05/01/91 12:20:11                                                PAGE     1
PROGRAM NAME =

    1                                       ;H8/300 CPU 16x16 Multiply Routine
    2
    3                                       ;This routine uses strictly registers to maintain all
    4                                       ;storage facilities and for calculation.
    5
    6                                       ;Register Usage
    7                                       ;Entry:
    8                                       ;          R1 = Multiplier
    9                                       ;          R2 = Multiplicand
   10                                       ;       R3,R4 = Temporary Result
   11                                       ;          R5 = Temporary Storage
   12                                       ;Exit:
   13                                       ;          R1 = Result, LSW
   14                                       ;          R2 = Result, MSW
   15
   16                                       ;Pictorial Description:
   17                                       ;               R2H     R2L
   18                                       ;               R1H     R1L
   19                                       ;               _____
   20                                       ;                   R2L*R1L
   21                                       ;               R2H*R1L
   22                                       ;           R2L*R1H
   23                                       ;       R2H*R1H
   24                                       ;       _________
   25                                       ;       - - R E S U L T - -
   26
   27 P        C 0000                       mult16:
   28 P        C 0000 79040000              step1:  mov.w   #0,r4           ;clear result register
   29 P        C 0004 0D25                         mov.w   r2,r5           ;save multiplicand
   30
   31 P        C 0006 5092                  step2:  mulxu   r1l,r2          ;multiplier(L) x multiplicand(L)
   32 P        C 0008 0D23                         mov.w   r2,r3           ;1.     R3  <- R2L*R1L
   33
   34 P        C 000A 0C5A                  step3:  mov.b   r5h,r2l         ;retrieve multiplicand(H)
   35
   36 P        C 000C 5092                  step4:  mulxu   r1l,r2          ;multiplier(L) x multiplicand(H)
   37 P        C 000E 08A3                         add.b   r2l,r3h         ;2.     R3H <- R3H + (R2H*R1L)L
   38 P        C 0010 0E2C                         addx    r2h,r4l         ;       R4L <- (R2H*R1L)H + CY
   39
   40 P        C 0012 0D52                  step5:  mov.w   r5,r2           ;retrieve multiplicand
   41
   42 P        C 0014 5012                  step6:  mulxu   r1h,r2          ;multiplier(H) x multiplicand(L)
   43 P        C 0016 08A3                         add.b   r2l,r3h         ;3.     R3H <- R3H + (R2L*R1H)L
   44 P        C 0018 0E2C                         addx    r2h,r4l         ;       R4L <- R4L + (R2L*R1H)H + CY
   45 P        C 001A 9400                         addx    #0,r4h          ;       R4H <- CY
   46
   47 P        C 001C 0C5A                  step7:  mov.b   r5h,r2l         ;retrieve multiplicand(H)
   48
   49 P        C 001E 5012                  step8:  mulxu   r1h,r2
   50 P        C 0020 0942                         add.w   r4,r2           ;4.     R2  <- R4 + (R2H*R1H)
   51 P        C 0022 0D31                         mov.w   r3,r1           ;setup return results
   52
   53 P        C 0024 5470                         rts                     ;return
   54
   55                                              .end
  *****TOTAL ERRORS       0
  *****TOTAL WARNINGS     0
```