Hitachi America, Ltd.

TN-0203

Application Engineering
TechNote

Sam Darwin

32/16 Divide for H8/300 Family

Hitachi's H8/300 family is currently equipped with a 16/8 bit divide command called DIVXU. This command performs unsigned division on values in two specified registers, and produces an 8-bit result and 8-bit remainder which are placed back into a register upon completion.

The code offered in this technote may be used to perform 32/16 division, and it differs in a number of ways from the 'DIVXU' command, besides the fact that it allows calculation with larger numbers. First of all, the numbers to be calculated are held in RAM memory space instead of in registers. Thus, in order to use this routine as a subroutine within a larger program, the programmer must place the desired dividend and divisor within the specificed memory area.

Secondly, this code may be used as signed or unsigned division, as it does not modify any flags. If the user wishes, he may include the necessary compare instructions in order to set certain flags. Moreover, this code does not incorporate the DIVXU command at all, but rather uses a bit by bit method of calculation. There is no simple way to use the DIVXU command repeatedly in order to create a 32/16 divide, although it may initially appear that there would be. The following example illustrates the process used here, and it will be explained further on the next pages.

Example division:

```
                              result
0010 ⌐0110      --->  divisor  ⌐dividend
```

after shifting stage we have:

```
1000 ⌐1100
```

Compare 1100 to 1000. Since 1000 is smaller subtraction occurs.

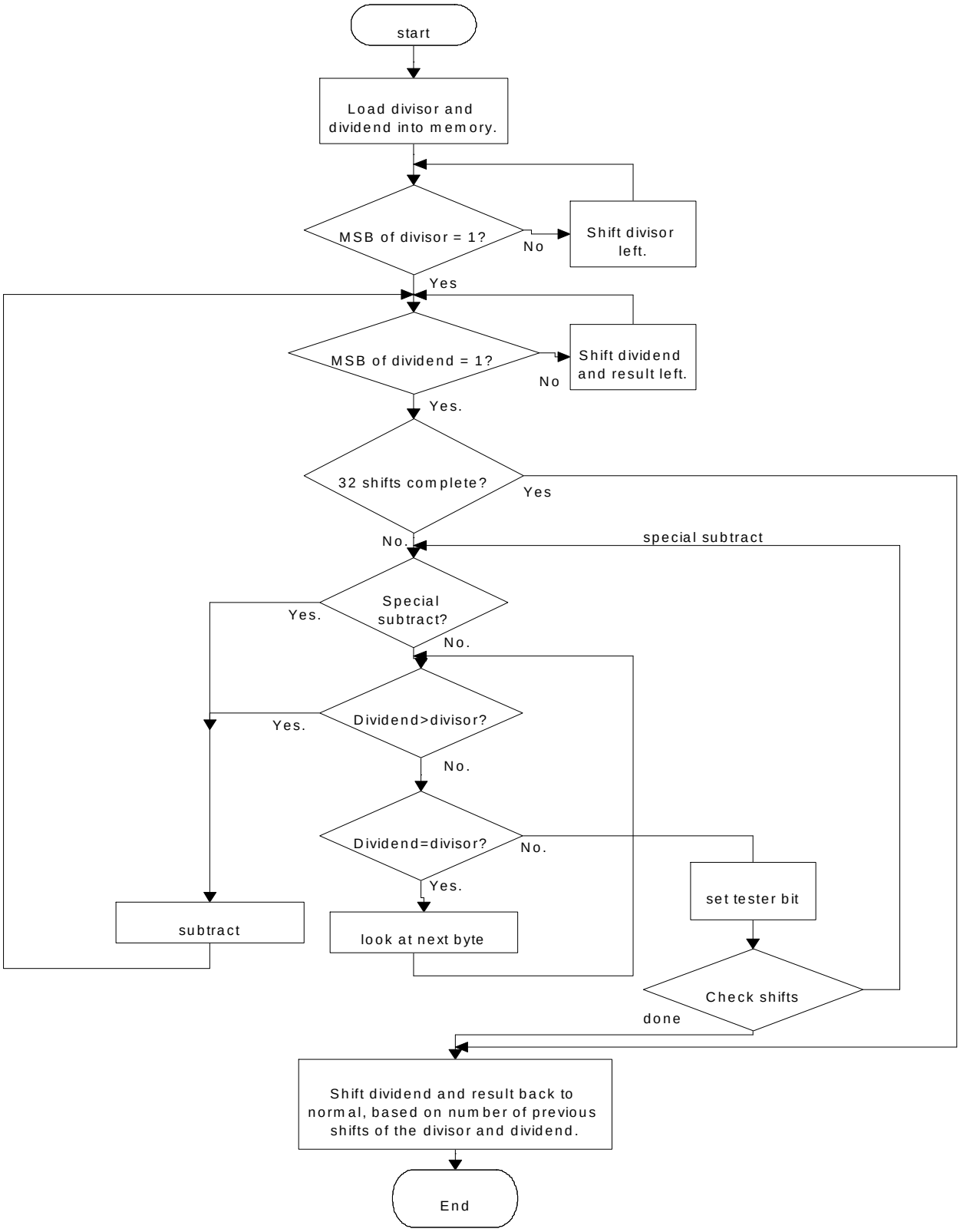
```
         1
1000 ⌐1100
     -1000
      0100
```

The result of the subtraction becomes the new dividend

```
         1
1000 ⌐0100
```

Shift, and repeat

```
        11
1000 ⌐1000
     -1000
         0
```



Division Flowchart

HITACHI
Hitachi America, Ltd. • San Francisco Center • 2000 Sierra Point Parkway • Brisbane, CA 94005-1819 • (415) 589-8300

First of all, the divisor is shifted until there is a one in the highest order bit position. Long-division is accompished through a series of subtractions that begin with the highest order digits, and progress towards the lower digits. Thus, the dividend is also shifted until there is a one in the highest order digit. In all cases, zeros will be shifted on the right side.

Subtraction, and 'Special Subtraction'

Before the subtracting stage, we must determine if a subtraction is possible. Is the dividend larger than the divisor? We can determine this by comparing the most significant 8 bits of the dividend and divisor. If the dividend is larger, a one belongs in the highest order digit of the result, and the subtraction occurs. If the two 8-bit numbers are equal, then the comparison must continue by comparing the next 8 bits of the two numbers. If the divisor is larger, we know that the subtraction should not take place and that a zero should be placed in the proper digit of the result. We also know that the next digit of the result should be a one, and that a subtraction will be allowed. Is this certain? It must be, due to the fact that both the dividend and the divisor have a one in their highest order bit, and if the divisor is larger than the dividend in the current comparison, it will not be after a shifting of one bit to the right. This is extremely important to understand, as it is a common situation in division. It shall be referred to here as 'special subtraction', and can be explained again in slightly different terms: if the divisor is larger than the dividend then a subtraction would result in a negative value. Let's use the example of 1100 - 1111. We don't want this to happen and so no subtraction occurs. The program puts a zero in the result instead of a one, and then shifts the dividend just like normal. The shift yields 11000, so that a subtract is now appropriate. Of course, the highest order "1" has been shifted out of the picture, but we can remember that it should still be there by setting a flag which has been named the "tester bit" for this program. When the subtraction occurs the carry bit will be set, but this really is of no concern. As you can see, this subtract will always be allowed, since a 10000 is bigger than even a 1111.

The dividend is shifted in every cycle and the result is created bit by bit depending upon whether or not the subtraction took place. The comparison is made, and if it is allowed then the divisor is subtracted from the dividend and the cooresponding result bit is set. The shifting and subtracting continues until the dividend is smaller than the divisor, and what is left in the dividend is aptly called "the remainder". How do we know when this has finally happened? The program knows that the division is completed when the divisor is larger than the dividend, and finds this out through the comparision that is done in every cycle. The two numbers are compared, and if the divisor is larger than the dividend then the number of shifts are also examined. If these two factors in conjunction indicate that the divisor is truly large than the remaining dividend, the division is complete.

All that is left to do after this is to finish shifting the result and the remainder, so that they are back in their original positions. All of the shifting that has been done throughout the calculating has left the correct results in both the remainder and the result, but the decimal point is out of whack, so to speak. It is necessary to shift the numbers back, as many times as they were initially shifted forward. Another way of accomplishing this is to continue to shift them forward until a total of thirty-two shifts have occurred (and wrapping the highest order bits around, so they are not "shoved off the edge").

Some additional shifting needs to be done on the result to account for the shifting of the divisor at the beginning. The divisor was shifted in order to make the calculations possible, but it also affects how large the divisor appeared to be. Is it 10 or 100, for instance? This can only be determined by the number of shifts it experienced at the beginning, and the result must be duly compensated by an equal amount of shifting at the end of the computations.

Consider for a moment the most basic of division algorithms: the divisor is subtracted repeatedly from the dividend, and the number of times this is allowed will be the final result of the calculation. Such an option is very simple to understand, but will take a tremendous amount of time for the processor to implement. Thus, it becomes necessary to create an algorithm with both subtracting and shifting to create the result one digit at a time. The next question now becomes, how should this be done to reduce the calculation time to a minimum. The divisor, the dividend, or both may be shift, values must be compared, etc. In other words, there are quite a number of different ways to implement this algorithm. The one presented here works and is efficient, but should not be taken for the only answer.

HITACHI
Hitachi America, Ltd. • San Francisco Center • 2000 Sierra Point Parkway • Brisbane, CA 94005-1819 • (415) 589-8300

DIVIDE.S

```
divid0  .equ    h'02    ;dividend - 32 bits
divid1  .equ    h'00
divid2  .equ    h'00
divid3  .equ    h'00
divis0  .equ    h'02    ;divisor - 16 bits
divis1  .equ    h'00
remain0 .equ    h'FF00  ;memory location of remainder
remain1 .equ    h'FF01
remain2 .equ    h'FF02
remain3 .equ    h'FF03
vis0    .equ    h'FF10  ;memory location of divisor
vis1    .equ    h'FF11
result0 .equ    h'FF20  ;memory location of result
result1 .equ    h'FF21
result2 .equ    h'FF22
result3 .equ    h'FF23

        .section div,text, locate= 00

        .ORG h'00
        .data.w   start     ;jump to start of program

        .ORG    h'02A
start:  mov.w   #h'ff80,R7         ;initialize stackpointer, to this?
        mov.b   #h'00,R5l       ;used as test bit and result bit later
        mov.b   #h'00,R1h       ;for later use
        mov.b   #h'10,R4l       ;for counting divisor shifts
        mov.b   #h'00,R4h       ;counting dividend, result shifts

        mov.b   #divid0,R1l       ;put data into dividend
        mov.b   R1l,@remain0
        mov.b   #divid1,R1l
        mov.b   R1l,@remain1
        mov.b   #divid2,R1l
        mov.b   R1l,@remain2
        mov.b   #divid3,R1l
        mov.b   R1l,@remain3

        mov.b   #divis0,R1l      ;put data into divisor
        mov.b   R1l,@vis0
        mov.b   #divis1,R1l
        mov.b   R1l,@vis1

        mov.b   #h'00,R1l       ;clear result area
        mov.b   R1l,@result0
        mov.b   R1l,@result1
        mov.b   R1l,@result2
        mov.b   R1l,@result3

dov:    btst    #h'7,@vis1      ;test divisor,
        bne     a               ;shift it left
        mov.b   @vis0,R2l
        shal.b  R2l
        mov.b   R2l,@vis0
        mov.b   @vis1,R2l
        rotxl   R2l
        mov.b   R2l,@vis1
        inc.b   R4l             ;count shifts
        jmp     @dov
```

```
a:      btst    #h'7,@remain3   ;test dividend
        bne     b

shif:   mov.b   @remain0,R2l    ;shift dividend left
        shal.b  R2l
        mov.b   R2l,@remain0
        mov.b   @remain1,R2l
        rotxl   R2l
        mov.b   R2l,@remain1
        mov.b   @remain2,R2l
        rotxl   R2l
        mov.b   R2l,@remain2
        mov.b   @remain3,R2l
        rotxl   R2l
        mov.b   R2l,@remain3

        bld     #h'01,R5l       ;load carry
        mov.b   @result0,R2l    ;shift result left
        rotxl.b R2l
        mov.b   R2l,@result0
        bclr    #h'01,r5l       ;clear bit afterwards
        mov.b   @result1,R2l
        rotxl   R2l
        mov.b   R2l,@result1
        mov.b   @result2,R2l
        rotxl   R2l
        mov.b   R2l,@result2
        mov.b   @result3,R2l
        rotxl   R2l
        mov.b   R2l,@result3

        inc.b   R4h             ;count shifts
        cmp.b   #h'20,R4h
        beq     done            ;if more than 32 shifts
        btst    #h'0,R5l        ;test bit for special case
        bne     b
        jmp     @a

b:      cmp.b   R4l,R4h
        bls     fd
        jmp     @done

fd:     btst    #h'0,R5l        ;special case
        bne     subtr

comp:   mov.b   @remain3,R2h    ;put remainder into R2h
        mov.b   @vis1,R2l       ;put divisor into R2l
        cmp.b   R2l,R2h         ;compare values
        bls     comp2           ;if divisor>=remainder, branch

subtr:  bclr    #h'0,R5l        ;clear special bit
        bset    #h'1,R5l        ;set result bit

sub:    mov.b   @remain3,R3h    ;do subtract
        mov.b   @remain2,R3l
        mov.b   @vis1,R2h
        mov.b   @vis0,R2l
        sub.w   R2,R3
        mov.b   R3h,@remain3
        mov.b   R3l,@remain2
        jmp     @a
```

```
comp2:  cmp.b   R2l,R2h         ;are they equal?
        bne     comp3
        mov.b   @remain2,R2h    ;check next lower byte
        mov.b   @vis0,R2l
        cmp.b   R2l,R2h
        bls     comp4
        jmp     @subtr          ;subtract, since it is allowed

comp4:  cmp.b   R2l,R2h         ;are they equal?
        bne     comp3
        jmp     @subtr          ;in the case of equality, do the subtraction

comp3:  cmp.b   R4h,R4l
        bls     done              ;
        bset    #h'0,R5l
        jmp     @shif

done:   cmp.b   #h'20,R4h       ;
        beq     fini
        add.b   #h'1,R4h        ;increment
                                ;shift result by 1
        bld     #h'01,R5l       ;load carry
        mov.b   @result0,R2l
        shal.b  R2l
        mov.b   R2l,@result0
        bclr    #h'01,r5l       ;clear bit afterwards
        mov.b   @result1,R2l
        rotxl   R2l
        mov.b   R2l,@result1
        mov.b   @result2,R2l
        rotxl   R2l
        mov.b   R2l,@result2
        mov.b   @result3,R2l
        rotxl   R2l
        mov.b   R2l,@result3
                                ;shift remainder
        bld     #h'07,@remain3
        mov.b   @remain0,R2l
        rotxl   R2l
        mov.b   R2l,@remain0
        mov.b   @remain1,R2l
        rotxl   R2l
        mov.b   R2l,@remain1
        mov.b   @remain2,R2l
        rotxl   R2l
        mov.b   R2l,@remain2
        mov.b   @remain3,R2l
        rotxl   R2l
        mov.b   R2l,@remain3
        jmp     @done

fini:   bld     #h'07,@result3
        mov.b   @result0,R2l
        rotxl   R2l
        mov.b   R2l,@result0
        bclr    #h'01,r5l       ;clear bit afterwards
        mov.b   @result1,R2l
        rotxl   R2l
        mov.b   R2l,@result1
        mov.b   @result2,R2l
        rotxl   R2l
```

```
        mov.b   R2l,@result2
        mov.b   @result3,R2l
        rotxl   R2l
        mov.b   R2l,@result3
        cmp.b   #h'00,R4l
        beq     over
        dec.b   R4l
        jmp     @fini

over:   nop
        jmp     @over
```


HITACHI
Hitachi America, Ltd. • San Francisco Center • 2000 Sierra Point Parkway • Brisbane, CA 94005-1819 • (415) 589-8300