Notes on Economical 16/16 = 16 Unsigned Integer Division on the Freescale CPU08

David T. Ashley
dta@e3ft.com

Version Control Revision: 1.12
Version Control Date: 2007/05/26 18:25:46 (UTC)
RCSfile: cpu08div16by16a.tex,v
LaTeX Compilation Date: May 26, 2007

Abstract

This document contains my notes on attempting to optimize the 16 = 16/16 unsigned integer division subroutine distributed with a Freescale CPU08 compiler. I attempted to replace the standard shift-compare-subtract algorithm with Knuth's algorithm.

In the end, the attempt was mostly a failure. There seemed to be no way to gain the execution time advantages of Knuth's algorithm without increasing FLASH size. (The goal would have been to obtain more favorable execution time *and* FLASH size, rather than a tradeoff.)

1 Introduction and Overview

The fundamental goal of practical computer integer arithmetic is to obtain an exact result using machine instructions that are as fast and compact as possible.

For three of the four fundamental operations (addition, subtraction, and multiplication), it is intuitively obvious to most programmers how to use existing machine instructions to operate on operands that are larger than the machine instructions can accommodate.

The fourth fundamental operation (division), however, is less well understood by a typical programmer than the other three. It is not obvious to most programmers how to use machine division instructions to divide integers larger than the native machine division instructions will accommodate.

In 2007, I noticed that the integer division library functions associated with a particular *C* compiler were of the shift-compare-subtract variety. As the CPU08 has a 16/16=8 native division instruction, I suspected but was not sure that the 16/16=16 division function could be improved.

Note that for a library function used by a *C* compiler, it may not be necessary (although it may be desirable) to calculate the quotient and the remainder at the same time. An algorithm that calculates the quotient but does not calculate the remainder, or vice-versa, may be acceptable for use by compiled code.

Although I haven't examined the *C* standards, I doubt that there is any special requirement for the quotient or remainder calculated when the denominator is zero. The only requirement is probably that the calculation not continue indefinitely (i.e. not an infinite loop).

2 Nomenclature and Notation

I use the nomenclature "16/16=16 division" to denote a division function that accepts a 16-bit unsigned numerator and a 16-bit unsigned denominator; and produces either/both a 16-bit quotient and a 16-bit remainder.

I use n to denote the numerator, d to denote the denominator, q to denote the integer quotient, and r to denote the remainder.

Any of the 16-bit quantities can be subscripted with "H" or "L" to denote the most-significant or least-significant byte. For example,

$$n = 2^8 n_H + n_L. \tag{1}$$

Within any of the 16-bit quantities, bits are subscripted in the traditional fashion. For example,

$$n = \sum_{i=0}^{15} 2^i n_i. \tag{2}$$

Within any of the 8-bit quantities, bits are also subscripted in the traditional fashion. For example,

$$n = 2^8 \sum_{i=0}^{7} 2^i n_{H_i} + \sum_{i=0}^{7} 2^i n_{L_i} = \sum_{i=0}^{7} 2^{i+8} n_{H_i} + \sum_{i=0}^{7} 2^i n_{L_i}. \tag{3}$$

Because only non-negative integers are involved, *floor()* and *div* are used interchangeably, i.e.

$$\left\lfloor \frac{a}{b} \right\rfloor = a \text{ div } b, \quad a, b \geq 0. \tag{4}$$

An identity that is frequently used in this document is

$$\frac{a}{b} = a \text{ div } b + \frac{a \bmod b}{b} \tag{5}$$

$$= \left\lfloor \frac{a}{b} \right\rfloor + \frac{a \bmod b}{b}. \tag{6}$$

3 Analysis of Existing Code

This section presents an analysis of the behavior and characteristics of the existing code.

3.1 Listing

The existing code is reproduced below, with line numbers.

```
001: c_umod:
002:         bsr        c_udiv          ; divide
003: pshh                               ; transfer MSB
004:         pula                       ; in place
005:         sta        c_reg
006:         txa                        ; LSB in place
007:         rts                        ; and return
008: ;
009: c_udiv:
010:         psha                       ; save NL
011:         pshh                       ; DH on the stack
012:         tst        1,sp            ; test zero
013:         bne        full            ; full division
014:         pulh                       ; clean stack
015:         cpx        c_reg           ; compare DL with NH
016:         bls        half            ; half division
017:         lda        c_reg           ; NH
018:         psha                       ; in
019:         pulh                       ; H
020:         pula                       ; NL in A
021:         div                        ; DL in X, divide
```

```
022:          clr         c_reg           ; QH is zero
023: fini:
024:          pshh                        ; move RL
025:          pulx                        ; in X
026:          clrh                        ; RH is zero
027:          rts                         ; and return
028: half:
029:          lda         c_reg           ; NH in A
030:          clrh                        ; 1st divide 8 X 8
031:          div                         ; divide
032:          sta         c_reg           ; QH in place
033:          pula                        ; complete dividend with NL
034:          div                         ; divide
035:          bra         fini            ; complete remainder
036: full:
037:          pshx                        ; save DL
038:          ldx         #8              ; counter
039:          clra                        ; Extention
040:          pshx                        ; save on
041:          psha                        ; the stack
042:          tsx                         ; stack addressed by H:X
043: bcl:
044:          lsl         4,x             ; shift E:NH:NL
045:          rol         c_reg           ; left
046:          rola
047:          sta         0,x             ; store E
048:          cmp         3,x             ; compare DH
049:          blo         next            ; too low, continue
050:          bne         ok              ; ok to subtract
051:          lda         c_reg           ; compare NH
052:          sub         2,x             ; with DL
053:          bhs         ok2             ; ok, complete result
054:          lda         0,x             ; restore value
055:          bra         next            ; and continue
056: ok:
057:          lda         c_reg           ; substract D
058:          sub         2,x             ; from E:NH
059: ok2:
060:          sta         c_reg           ; in place
061:          lda         0,x
062:          sbc         3,x
063:          inc         4,x             ; set result bit
064: next:
065:          dbnz        1,x,bcl         ; count down and loop
066:          sta         0,x             ; store E
067:          pulh                        ; RH in place
```

```
068:        ais       #3            ; clean up stack
069:        ldx       c_reg         ; RL in place
070:        pula                    ; QL in place
071:        clr       c_reg         ; QH is zero
072:        rts                     ; and return
```

3.2 Algorithmic Analysis

The algorithm is divided into two cases:

- Case I: $d_H = 0$ (lines 14-35).
- Case II: $d_H > 0$ (lines 36-72).

3.2.1 Case I

In the case of $d_H = 0$, the division is 16/8:

$$\begin{aligned} \frac{2^8 n_H + n_L}{d_L} &= \frac{2^8 n_H}{d_L} + \frac{n_L}{d_L} && (7) \\ &= 2^8 \left(n_H \text{ div } d_L + \frac{n_H \bmod d_L}{d_L} \right) + \frac{n_L}{d_L} && (8) \\ &= 2^8 (n_H \text{ div } d_L) + \frac{2^8 (n_H \bmod d_L) + n_L}{d_L} && (9) \end{aligned}$$

(9) is an exact expression involving rational numbers. However, we don't want to calculate the left side of (9); rather, we wish to calculate its *floor()*. Applying the *floor()* function to both sides of (9) yields:

$$\left\lfloor \frac{2^8 n_H + n_L}{d_L} \right\rfloor = 2^8 (n_H \text{ div } d_L) + \left\lfloor \frac{2^8 (n_H \bmod d_L) + n_L}{d_L} \right\rfloor . \qquad (10)$$

Note that (10) is in a form that can be readily evaluated using a processor with 16/8 division capability; so long as

$$\frac{2^8 (n_H \bmod d_L) + n_L}{d_L} < 2^8, \qquad (11)$$

a fact that can be easily verified by the reader.

(10) can be readily evaluated by a processor with 16/8 division capability because such a division instruction always provides both quotient and remainder. It is easy to see that (10) can be evaluated with a division, a re-staging of bytes, and a second division.

If (10) is evaluated as suggested, it needs to be verified whether the remainder of the second division is the same as the remainder of the larger division, i.e.

$$(2^8 n_H + n_L) \bmod d_L =?((2^8 \bmod d_L) + n_L) \bmod d_L. \qquad (12)$$

The question of whether (12) is an equality is the question of whether

$$ka \bmod b = (k(a \bmod b)) \bmod b. \tag{13}$$

In order to prove or disprove (13), decompose a into $ib + j$. Then, since $kib \bmod b = 0$,

$$\begin{aligned} k(ib + j) \bmod b &= kj \bmod b & (14) \\ kj \bmod b &= kj \bmod b. & (15) \end{aligned}$$

Thus, if (10) is evaluated as suggested (with two divisions), the final remainder will be the same as the remainder for the original division. (10) will, in fact, deliver both the quotient and remainder economically.

3.2.2 Case II

The case of $d_H > 0$ (§3.1, lines 36-72) is conventional shift-compare-subtract division. Only eight iterations of the loop are required because with $d_H > 0$, $d \geq 2^8$, and $n/d < 2^8$.

3.3 Timing Analysis

The code of §3.1 is reproduced below, with instruction timing (number of clocks) and FLASH requirements (number of bytes) added as (clocks/bytes). It was determined that *c_reg* resides in zero-page (i.e. *direct*) memory.

```
001: c_umod:
002:         bsr        c_udiv     4/2 ; divide
003: pshh                              ; transfer MSB
004:         pula                  2/1 ; in place
005:         sta        c_reg      3/2
006:         txa                   1/1 ; LSB in place
007:         rts                   4/1 ; and return
008: ;
009: c_udiv:
010:         psha                  2/1 ; save NL
011:         pshh                  2/1 ; DH on the stack
012:         tst        1,sp       4/3 ; test zero
013:         bne        full       3/2 ; full division
014:         pulh                  2/1 ; clean stack
015:         cpx        c_reg      3/2 ; compare DL with NH
016:         bls        half       3/2 ; half division
017:         lda        c_reg      3/2 ; NH
018:         psha                  2/1 ; in
019:         pulh                  2/1 ; H
020:         pula                  2/1 ; NL in A
```

```
021:         div                 7/1 ; DL in X, divide
022:         clr         c_reg   3/2 ; QH is zero
023: fini:
024:         pshh                2/1 ; move RL
025:         pulx                2/1 ; in X
026:         clrh                1/1 ; RH is zero
027:         rts                 4/1 ; and return
028: half:
029:         lda         c_reg   3/2 ; NH in A
030:         clrh                1/1 ; 1st divide 8 X 8
031:         div                 7/1 ; divide
032:         sta         c_reg   3/2 ; QH in place
033:         pula                2/1 ; complete dividend with NL
034:         div                 7/1 ; divide
035:         bra         fini    3/2 ; complete remainder
036: full:
037:         pshx                2/1 ; save DL
038:         ldx         #8      2/2 ; counter
039:         clra                1/1 ; Extention
040:         pshx                2/1 ; save on
041:         psha                2/1 ; the stack
042:         tsx                 2/1 ; stack addressed by H:X
043: bcl:
044:         lsl         4,x     4/2 ; shift E:NH:NL
045:         rol         c_reg   4/2 ; left
046:         rola                1/1
047:         sta         0,x     2/1 ; store E
048:         cmp         3,x     3/2 ; compare DH
049:         blo         next    3/2 ; too low, continue
050:         bne         ok      3/2 ; ok to subtract
051:         lda         c_reg   3/2 ; compare NH
052:         sub         2,x     3/2 ; with DL
053:         bhs         ok2     3/2 ; ok, complete result
054:         lda         0,x     2/1 ; restore value
055:         bra         next    3/2 ; and continue
056: ok:
057:         lda         c_reg   3/2 ; substract D
058:         sub         2,x     3/2 ; from E:NH
059: ok2:
060:         sta         c_reg   3/2 ; in place
061:         lda         0,x     2/1
062:         sbc         3,x     3/2
063:         inc         4,x     3/2 ; set result bit
064: next:
065:         dbnz        1,x,bcl 5/3 ; count down and loop
066:         sta         0,x     2/1 ; store E
```

```
067:        pulh                    2/1 ; RH in place
068:        ais         #3          2/2 ; clean up stack
069:        ldx         c_reg       3/2 ; RL in place
070:        pula                    2/1 ; QL in place
071:        clr         c_reg       3/2 ; QH is zero
072:        rts                     4/1 ; and return
```

There are three distinct timing cases for the *c_udiv* function:

1. $d_H = 0$ and $n_H < d_L$: 47 clocks are required, representing straight flow of the instructions from line 10 through line 27.
2. $d_H = 0$ and $n_H \geq d_L$: 54 clocks are required.
3. $d_H > 0$ and every bit of the quotient is 1, in which case 400 clocks are required. This represents 22 clocks up through line 42, 45 clocks $\times$ 8 in the lines from 43 through 65, and 18 clocks in the lines from 66 through 72.

3.4 FLASH/RAM Consumption Analysis

From §3.3, 93 bytes of FLASH are used. Only one byte of RAM is used (*c_reg*, probably shared with other functions as well).

4 Potential Optimizations

4.1 Potential Case I Optimizations

This section corresponds to *Case I* of §3.2.1.

The most obvious observation about the code (§3.1) is that division instructions are very inexpensive on the CPU08—7 clock cycles, or about 2 typical instructions. Branching based on $n_H \geq d_L$ (§3.1, lines 15-16) may cost more in the test, the branch, and in other data transfer overhead than is saved. It may make sense to apply the full formula in (10) in all cases where $d_H = 0$.

When $d_H = 0$, and if one assumes normal distribution of data, the expected value of execution time is about $(47 + 54)/2 = 50.5$ clocks.[1]

The code below combines two of the three timing cases into one by ignoring the relationship between n_H and d_L.

```
;Condition at function entry:
;N_H in 1,SP
;N_L in A
;D_H in H
;D_L in X
;
```

[1] If the data is assumed normally distributed, n_H has about a 0.5 probability of being at least as large as d_L.

```
;Condition at function exit:
;Q_H in c_reg
;Q_L in A
;R_H in H
;R_L in X
;
c_udiv:
   psha            2/1 ; save NL
   pshh            2/1 ; DH on the stack
   tst   1,sp      4/3 ; test zero
   bne   full      3/2 ; full division
;
;From here on we're committed to the division with
;arbitary numerator, and denominator <= 255.
;  N_H at 3,sp
;  N_L at 2,sp
;  D_H at 1,sp
;
   clrh            1/1
   lda   3,sp      4/3 ; H:A now contains N_H
   div             7/1 ; divide
   sta   c_reg     3/2 ; QH in place
   lda   2,sp      4/3 ; complete dividend with NL
   div             7/1 ; divide.  Q_L in A, R_L in H
   pshh            2/1 ; move RL
   pulx            2/1 ; in X
   clrh            1/1 ; RH is zero
   ais   #3        2/2 ; clean stack
   rts             4/1 ; and return
```

Although the code does raise the minimum execution time from 47 to 48 clocks:

- It lowers the expected value of the $d_H = 0$ execution time from 50.5 to 48 clocks.
- It saves approximately 15 bytes of FLASH.

This optimization is recommended.

4.2 Potential Case II Optimizations

This section corresponds to *Case II* of §3.2.2.

I sent an e-mail to an engineer at the compiler manufacturer indicating that:

- I believed *Case I* could be optimized as indicated earlier in the document.
- I believed *Case II* could be optimized by applying Knuth's algorithm.

The reply I received from the compiler manufacturer was that:

- There was agreeement about *Case I*.
- *Case II* may be a little faster using Knuth's algorithm, but would definitely be larger (code used to evaluate the application of Knuth's algorithm was also provided).

In the test code provided by the compiler manufacturer, the approach used was to obtain a trial quotient and then to subtract the divisor from the remainder up to twice to adjust the quotient up to 2 counts downward (Knuth's algorithm).

I did try a different approach (to iterate on the quotient and to reconstruct $q \times d$ with decreasing q). This approach promised to be slightly more compact because $q \times d$ reconstruction was reutilized. However, it worked out to occupy about 153 bytes rather than 103 for the shift-compare-subtract algorithm (timing was not examined). (This test code is version-controlled in the same directory as this LaTeX document.)

At this point I agree with the compiler manufacturer that there is a tradeoff between size and speed (it seems nearly impossible to get both).

In any reimplementation of this algorithm, will probably need to choose between size and speed. I believe there is some possibility to reduce the reimplementation from 153 bytes, but not down to 103 bytes.