Division by 10

R.A. Vowels
Department of Computer Science
Royal Melbourne Institute of Technology Limited
Box 2476V, GPO, Melbourne, Victoria 3001, Australia

Division of a binary integer and a binary floating-point mantissa by 10 can be performed with shifts and adds, yielding a significant improvement in hardware execution time, and in software execution time if no hardware divide instruction is available. Several algorithms are given, appropriate to specific machine word sizes, hardware and hardware instructions available, and depending on whether a remainder is required.

The integer division algorithms presented here contain a new strategy that produces the correct quotient directly, without the need for the supplementary correction required of previously-published algorithms. The algorithms are competitive in time with binary coded decimal (BCD) divide by 10.

Both the integer and floating-point algorithms are an order of magnitude faster than conventional division.

Keywords and phrases: division, integer, floating-point, shift, add, divide by 10, BCD, quotient, remainder, microcode.

CR Categories: B1.5, C.0, D.1.m, F.2.1, G.1.0.

Copyright © 1992, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the ACJ's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received 2 January 1991.
Revised manuscript received August 1991.

1 INTRODUCTION

In this paper are presented new algorithms for dividing binary integer and binary floating-point values by the constant 10. Because the algorithms are concise, significant savings in time are afforded compared to conventional division.

One of the drawbacks of binary integer arithmetic compared to binary coded decimal (BCD) is the lack of an efficient divide by 10 instruction. Division by 10 is needed for scaling and for conversion of binary to BCD and ASCII, as well as for general computation. The writer is convinced that if one of these algorithms were implemented in hardware, it would render binary arithmetic a far more attractive proposition compared to BCD.

The binary integer division algorithms exhibited in this paper employ shifts and adds, and include an elegant new strategy, devised by the author, for producing the correct quotient directly (and the remainder, when required). The new strategy relies on the unusual step of adding a round-off *prior* to shifting, and of maintaining the minimum of fraction bits consistent with producing a correct quotient. Minimising the number of fraction bits is important in minimising the number of high-order bits of intermediate result. (For the idea of shifts and adds, acknowledgement is given to Knuth (1969).)

Various binary integer division algorithms are given in this paper for different machine word sizes, hardware and hardware instructions, and depending on whether a remainder is required. All are characterised by being relatively fast compared to conventional integer division. The floating-point algorithm given in this paper is an order of magnitude faster than a conventional floating-point divide. (For an example of conventional division, refer to the restoring division algorithm in Zaks (1982).)

This paper is organised as follows: it begins with an outline of the origins of division by 10 algorithms. Next, an explanation of the source of truncation error of earlier implementations of integer division and how it is eliminated here, is given. Then, several algorithms for both unsigned and signed integer division by 10 are provided. Remarks on hardware implementation are included, algorithms for obtaining both the quotient and remainder, and for floating-point division by 10, are shown.

2 ORIGINS

It has variously been stated that division of a positive binary integer by 10 can be performed by a series of shifts and adds (Knuth, 1969, 1981; Allison, 1980; Cortesi, 1981). Several algorithms on this topic have been published: those of Allison and Cortesi (ibid), however, are of two stages, the first of which yields an interim quotient, while the second corrects this to produce the true quotient. Knuth gives only the first stage (ibid). A one-stage algorithm is a special case for quotients less than 256 (Ashley, 1981).

Bibliotheek
Centrum voor Wiskunde en Informatica

Knuth and Allison observe that the interim quotient is occasionally low by 1, and in Allison's algorithm the deficiency is remedied by multiplying the interim quotient by 10, subtracting the product from the dividend, and thereby forming an interim remainder. When the interim remainder is more than 9, the interim quotient is corrected by incrementing it by 1 while the interim remainder is corrected by subtracting 10. (An outline of the correction appears in Knuth (1969, p 283).) These additional steps detract from what would otherwise be a compact algorithm. (One implementation for the Z80 and 8080 microprocessors uses 22 instructions for the first stage (including a subroutine) and a further 18 for the second (Cortesi).)

3 THE ERROR

The shifts and adds used by Knuth, Allison and Cortesi are equivalent to multiplying the integer by a *truncated* form of 0.1, hence the error. This error can be eliminated by adding a round-off of unity to the dividend *prior* to the division. Adding a round-off of 1 to the dividend is equivalent to adding a value slightly less than 0.1 to the quotient. Thus a dividend whose least significant digit in the decimal system is 9 will not effectively be rounded up to the next 10. The round-off will swamp the truncation error, provided that sufficient fraction bits are maintained (usually between 4 and 6 bits).

4 POSITIVE INTEGER DIVISION BY 10

The division of a positive binary integer k by 10 can be achieved by using one of Algorithm 1(a), (b) or (c) according to word size. These algorithms have been devised by the author of this paper. (For the idea of shifts and adds, performed in an order that minimises the number of shifts, acknowledgement is given to Knuth (1969) and Allison (1981).)

(a) *8-bit integer* ($0 <= k <= 128$):

```
j = k + 1          round off
m = j + j/2        shift of 1
n = m + m/16       shift of 4
q = n/16           shift of 4 to discard the fraction
```

(b) *16-bit integer* ($0 <= k <= 32,768$):

```
j = k + 1          round off
m = j + 2j         shift of 1
n = m + m/16       shift of 4
p = n + n/256      shift of 8
q = p/32           shift of 5 to discard the fraction
```

(c) *32-bit integer* ($0 <= k <= 2,147,483,648$):

```
j = 2(k + 1)       round off and double
m = j + 2j         shift of 1
n = m + m/16       shift of 4
p = n + n/256      shift of 8
r = p + p/65536    shift of 16
q = r/64           shift of 6 to discard the fraction
```

Algorithm 1: Unsigned integer divide by 10.

In the first algorithm, the multiplying factor is $\frac{51}{512}$; in the second, the factor is $\frac{13017}{131072}$, while in the third, the factor is $\frac{858993459}{8589934592}$.

In each of these algorithms, all divisions and multiplications are performed by logical shifts. Any overflow condition or flag is ignored, and for 16- and 32-bit words, an extension word is required. Any bits shifted out of the right-hand end of the least significant word are ignored. If the processor is endowed with arithmetic facilities on 8-bit and 16-bit words, the only shifts required are of 1 and of either 4, 5 or 6 bits.

As to the 16-bit algorithm, if the use of an extension word is inconvenient, re-order the steps thus: $j = k/2 + 1$; $m = j + 2j$; $n = m + m/16$; $p = n + n/256$; $q = p/16$. A similar economy may be effected with the 32-bit version, as follows: $j = k/2 + 1$; $m = j + 2j$; $n = m + m/16$; $p = n + n/256$; $s = p + p/65536$; $q = s/16$. (In both these variations, the integer k is truncated to an even value before applying a round-off of 2, thereby retaining an effective round-off of 3/16 which would otherwise have been truncated to 1/8.)

If a word holds positive integers only, encoded in all bit positions (e.g., a byte can hold integers in the range 0 through 255), the carry must be maintained (possibly in an extension word). An extra add is required as the penultimate step, in order to sum the extension word and the low-order word (which dispenses with a shift). In the specific case of an 8-bit byte, however, a more efficient approach is available, namely: begin with a right shift, and finish up with a shift of 3 places instead of 4, as shown in Algorithm 2. (Apart from the round-off, the algorithm is, by co-incidence, attributable to Knuth (1969).)

```
j = k/2 + 1        halve and round off
m = j + j/2        shift of 1
n = m + m/16       shift of 4
q = n/8            shift of 3
```

Algorithm 2: Unsigned integer divide by 10.

For a 16-bit integer k ($0 <= k <= 65535$), use the algorithm $j = k/2 + 1$; $m = j + j/2$; $n = m + m/16$; $p = n + n/256$; $q = n/8$, and retain and sum one bit shifted out in all but the first right shift steps. On many systems, this bit is accessible via the carry bit. If the bit is not accessible, an alternative is to add in the bit prior to the shift, as follows: $j = k/2 + 1$; $t = j \& 1$; $m = j + j/2$; $u = m/8 + t$; $n = m + u/2$; $t = u \& 1$; $v = n/128 + t$; $p = n + v/2$; $q = p/8$. (The symbol & signifies a *logical and* operation.)

5 SIGNED DIVISION BY 10

If signed values are to be divided by 10, *arithmetic* shifts must be used throughout, wherever division is indicated. The following algorithms are required for signed binary integer division in 8, 16 and 32-bit words respectively:

(a) *8-bit integer* ($—128 <= k <= 127$):

```
j = k + 1
if k < 0 then j = j — 2
k = k + k/2
```

```
if k < 0 then k = k — 1
m = j + j/16
q = m/16
if q < 0 then q = q + 1
```

(b) *16-bit integer* (—32768 <= *k* <= 32767):

```
j = k + 1
if k < 0 then j = j — 2
j = j + 2j
if j < 0 then j = j — 1
m = j + j/16
n = m + m/256
q = n/32
if q < 0 then q = q + 1
```

(c) *32-bit integer* (2,147,483,648 <= *k* <= 2,147,483,647):

```
j = k + 1
if k < 0 then j = 2(k/2) to drop the least significant bit.
j = j + 2j
m = j + j/16
n = m + m/256
p = n + n/65536
q = p + p/4,294,967,296
q = q/32
if q < 0 then q = q + 1
```

Algorithm 3: Signed integer divide by 10.

For the 8-bit, 16-bit and 32-bit versions of Algorithm 3 respectively, an intermediate precision of 9, 18 and 34 bits must be maintained.

Each of the conditionals (e.g., if q < 0 then q = q + 1) can be implemented by an arithmetic shift and a subtract (e.g., q = q — q/256 for an 8-bit word), or by a move word, a left shift into the carry bit, followed by an add with carry. A conditional in the microcode is not required.

Two further possible alternatives for signed division are given here for 32-bit words:

(a) if k < 0 then k = k + 1; j = k + 2147483648 + 3;
m = j + 2j;
n = m + m/16; p = n + n/256; q = p + p/65536;
r = q + q/4,294,967,296; s = r/64;
t = s — 214748365;
if t < 0 then t = t — 1.
In this instance, each conditional can be implemented by a shift and an add or subtract.

(b) j = k; if k < 0 then j = —j; {as for division of a positive integer} if k < 0 then q = —q.
In this instance, the conditionals are unavoidable.

6 HARDWARE DIVISION

Using one of the preceding algorithms, it should be possible to produce an efficient binary integer divide-by-10 in hardware, requiring fewer than half the cycles of a conventional hardware binary integer divider. In a conventional divider, a subtract, test and a shift are performed for all the bits of the divisor register, or 48 steps for a 16-bit divisor. On the other hand, for the same 16-bit dividend, dividing a positive integer by 10 (employing shifts and adds) requires only nine steps.

The algorithm requires only $m+2\log_2\frac{b}{2}$ additions and shifts, and an aggregate of approximately $b+2$ shift positions (where $m = 2$ or 3 depending on the initial setup, and b = number of bits in a word).

An example microcode program for a hardware unsigned integer divide by 10, derived from Algorithm 1(b), is shown below. It is assumed that the processor already has a hardware multiplier/divider and thus has a double precision facility associated with the adder. Specifically, it is assumed that both the divisor and dividend are 16 bits, and that there are at least 17 bits of intermediate storage in the scratch register *temp.* (*acc* is an accumulator).

```
increment acc
temp ← acc
temp ← left shift temp 1 place
temp ← temp + acc
acc  ← right shift temp 4 places
temp ← temp + acc
acc  ← right shift temp 8 places
temp ← temp + acc
acc  ← right shift temp 5 places
```

A hardware divide instruction designed on these principles would be competitive with its BCD counterpart, and will reduce the need to perform arithmetic using decimal facilities. This technique can be used to enhance existing decimal facilities. (For example, the decimal adjust instruction AAM on the 80386 divides by 10 a one-byte positive binary integer of value less than 82, to produce a 4-bit quotient and 4-bit remainder. The time taken for this operation (17 clock cycles) is at least as much as (sometimes more than) that for binary integer division of a one-byte integer (14-17 clock cycles), using the DIV instruction (Pappas, 1988).)

7 REMAINDER

The remainder can be obtained either by multiplying the quotient by 10 (by shifts and adds) and subtracting from the dividend, or by preserving more bits of the fraction formed with the quotient, and then recovering that fraction and multiplying by 10 (by shifts and adds).

In the two versions of Algorithm 4, the latter method is used to produce both the remainder *r* and quotient *q* for a positive dividend *k*:

(a) *8-bit word*	(b) *32-bit word*	
		Begin forming the quotient:
j = k + 1	j = 2(k + 1)	
m = j + 2j	m = j + 2j	
n = m + m/16	n = m + m/16	

```
p = n + n/256    p = n + n/256
                 s = p + p/65536
                                    Begin forming the
                                    remainder:
r = p & 31       r = s & 63         extract the fraction
r = 4r + r       r = 4r + r         multiply by 5
r = r/16         r = r/32           final shift for the
                                    remainder
q = p/32         q = s/64           complete the
                                    quotient
```

Algorithm 4: Unsigned integer division with remainder.

This method of forming the remainder requires fewer steps than that involving multiplying the quotient by 10, used by Allison (1980) and Cortesi (1981).

8 FLOATING-POINT

As binary mantissas are almost universally stored in signed-magnitude form, an adaption of the algorithm for positive integer division will suffice. Division by 10 can be efficiently implemented in hardware; the shifts and adds can be performed readily enough on the mantissa (as for integer), and the final shift of 4 can be accomplished merely by decrementing the exponent by 4 (binary exponent) or by 1 (hexadecimal exponent) and by checking for exponent underflow. (Intermediate checks for underflow are not needed.) The final result may require normalising to take into account a carry.

Division of the mantissa of a 64-bit or 32-bit floating-point value *f* by 10 is depicted in Algorithm 5:

```
a = f + f/2
b = a + a/16
c = b + b/256
d = c + c/65536
e = d + d/4,294,967,296
e = e + roundoff (see below for notes)
e = e/16
```

Algorithm 5: Floating-point divide by 10.

When *f* resides in a 32-bit word, low-order extend it with zeros, and use 64-bit precision for intermediate working, so as to maximise accuracy. For the same reason, it is important that the mantissa be initially in its normalised form.

The round-off to be applied depends on the precision required. This round-off is primarily to force the correct result when the value is exactly divisible by 10. For a 32-bit dividend and with 64-bit intermediate working as suggested above, *roundoff* is a 1-bit added into the least significant end of the double word. (The lower word is then discarded, obviating the need to test for zero mantissa.) For a 64-bit dividend, as little as four additional bits at the right-hand end of the word are required. On a machine like the IBM System /370 which provides a guard digit in the 15th hexadecimal digit position, *roundoff* is 2, added to the guard-digit position. (The guard digit is then discarded, or is set to zero.) For a processor like the 80387 which uses 80-bit words for intermediate working (64 bits for the mantissa), *roundoff* is 1, added as an integer at the least-significant end of the word. In this case, however, the round-off can be added only when the mantissa is non-zero. (Note, however, that this increment, which is superfluous for values that are not multiples of 10, may be discarded by converting to long real or one of the other external forms.)

Considering the relatively long time for floating-point division (94 clocks for real division on an 80387 arithmetic co-processor (Leventhal, 1988)), the reduction in time afforded by a divide-by-10 instruction should be even more significant compared to integer division. Floating-point division by 10 would take 11 steps; conventional floating-point division would take 224 steps, not counting post-normalisation in either case. Thus, an improvement by an order of magnitude is possible.

Incidentally, a divide by 10 instruction cannot be considered a luxury: the 80387 arithmetic co-processor is provided with instructions to generate special constants (e.g., pi) and to perform certain trigonometric functions. If there is space for these, there is more than enough space for the few steps of a divide by 10 instruction.

9 VERIFICATION

The integer division algorithms have been simulated on a CDC 760 using PL/I and on a CDC Cyber 932 using FORTRAN, while the floating-point version has been simulated on a System /370 emulator on the latter machine. Algorithm 1(b) for division of a 16-bit integer was run on a Z80.

In simulating integer division, each integer in the entire range of integers was divided by 10 using each of the integer algorithms given here, and the result compared with that obtained by regular binary integer division. The results were identical. In the case of 32-bit words, only sample ranges were investigated (because of the impracticality of testing all 2^{32} combinations), but always including values in the vicinity of the largest possible integer.

In the case of floating-point, the simulation was performed with sample values using double precision integer arithmetic on the mantissa (using right shifts and adds) and then comparing these results with those produced by ordinary single and double precision floating-point division. The results obtained differed by a maximum of one bit in the least significant position.

10 TIMINGS

Algorithm 1(b) for dividing a binary integer by 10 was written in Z80 machine language and run on a Z80, along with a conventional restoring division program (Zaks, 1982). For timing purposes, the former program was executed 8×65535 times, and the latter program was executed 65535 times. The theoretical times and the observed times were within 1.4%. The theoretical time for traditional restoring division is 4.84 times that of the division by 10 algorithm here. It is noted that in the latter routine, the time for performing the two 4-bit right shifts represents 50% of the run time (owing to the lack of an instruction to shift a register right by 4 bits).

All three versions of Algorithm 1 for integer division were written in FORTRAN and run on a CDC Cyber 850, along with the corresponding traditional restoring division, also in FORTRAN, and the execution times compared. In the case of the 8-bit version, traditional division took 9.2 times that of the divide by 10; in the case of 16-bit division, traditional division took 14.2 times; for 32-bit division, traditional division took 20.0 times. Thus, for integer software division, the divide by 10 algorithms are at least nine times faster than conventional division.

Favourable comparisons between BCD divide by 10 instructions and the binary integer divide by 10 algorithms can be drawn to illustrate that a hardware divide by 10 instruction would be comparable in performance to its BCD equivalent.

On the Z80, a comparable BCD divide by 10 takes 31% of the time for the 16-bit binary integer divide in Algorithm 1(b). The decimal counterpart uses the specific-purpose Rotate Right Decimal instruction RRD to perform right shifts of 4 bits.

On the System /370 model 158, the decimal divide by 10 instruction SRP would take 108% of the time for software binary integer divide by 10 (based on 32-bit words and 6-byte BCD values) (IBM, 1978). (Not only is software divide by 10 marginally faster than BCD, it is also marginally faster than the binary integer divide instructions D and DR.)

On a CDC Cyber 850, software binary integer divide by 10 took 23% of the time for BCD division by 10.

For the 80386, software binary integer divide by 10 would take 50% more than hardware binary integer divide (Pappas). (BCD divide by 10 is not provided on this system, but the comparison is still useful.)

An improvement in speed of between 5 to 10 times could be anticipated in a hardware implementation of divide by 10. Therefore, a hardware binary divide by 10 would, at the very least, be comparable in time to its decimal counterpart.

11 CONCLUSION

This paper sets out algorithms for division by 10 using shifts and adds, and, in the case of integers, including a new strategy incorporating rounding off and maintenance of sufficient fraction bits, in order to produce the quotient directly. Algorithms were presented for both unsigned and signed integer, as well as binary floating-point divisions. Simulation has confirmed that the algorithms are sound. The paper proves that a software version of the integer division algorithm would be at least four times as fast as a software integer divide. A hardware divide by 10 instruction is shown to offer a similar saving. A binary integer divide by 10 instruction in hardware will be competitive with BCD divide by 10. Floating-point division by 10 is an order of magnitude faster than conventional floating-point division.

12 ACKNOWLEDGEMENTS

The author is grateful to colleague Dimitri Hintze for reading drafts of the paper and for his suggestions, and to the referees for their suggestions, all of which have resulted in a much better presentation.

13 REFERENCES

ALLISON, D.R. (1980): Converting Binary to Decimal, *Dr Dobb's J, No 42,* People's Computer Company, Menlo Park, California, February, Vol 5, No 2, pp 32-33.

ASHLEY, A. (1981): Dr Dobb's Clinic, *Dr Dobb's J. No 60,* People's Computer Company, Menlo Park, California, October, Vol 6, No 10, pp 7, 36-37.

CORTESI, D.E. (1981): Dr Dobb's Clinic, *Dr Dobb's J. No 60,* People's Computer Company, Menlo Park, California, October, Vol 6, No 10, pp 7, 36-41, 43.

INTERNATIONAL BUSINESS MACHINES (1978): *IBM System /370 Model 158 Functional Characteristics.*

KNUTH, D.E. (1969): "The Art of Computer Programming", *Seminumerical Algorithms,* Addison-Wesley, Reading, Massachusetts, Vol 2, Ex 9, pp 289-290.

KNUTH, D.E. (1981): "The Art of Computer Programming", *Seminumerical Algorithms,* 2nd Edition, Addison-Wesley, Reading, Massachusetts, Vol. 2, Ex 9, p 311.

LEVENTHAL, L. (1988): *Lance Leventhal's 80386 Programmers' Guide,* Bantam, NY, New York.

PAPPAS, C.H. and MURRAY, W.H. (1988): *80386 Microprocessor Handbook,* Osborne McGraw-Hill, Berkeley, California.

ZAKS, R. (1982): *Programming the Z80,* 3rd rev. Ed., Sybex, Alameda, California, p 140.

BIOGRAPHICAL NOTE

Robin Vowels is Senior Lecturer in the Department of Computer Science at the Royal Melbourne Institute of Technology Limited, where he has lectured in compiler design and computer architecture subjects. He holds a BE from Sydney University. His research interests include compiler and assembler design, and software tools. In addition to implementing PL/I compilers for the CDC Cyber 72 and 760 systems, and assemblers for several systems, he has authored two textbooks for undergraduate courses in computer programming.