Computer Systems

G. Bell, D. Siewiorek, and S.H. Fuller, Editors

# A Fast Division Technique for Constant Divisors

Ehud Artzy, James A. Hinds, and Harry J. Saal
State University of New York at Buffalo

**A fast algorithm for division by constant divisors is presented. The method has proved very useful implemented as microcode on a binary machine, and can be adapted directly into hardware. The mathematical foundations of the algorithm are presented as well as some performance measures.**

**Key Words and Phrases: constant divisors, division algorithms, bit addressable memory, microprogram**

**CR Categories: 4.13, 4.49, 6.32**

## 1. Introduction

We are concerned with the generation of fast algorithms for division by specified integers. The question arose from system design considerations for addressing a bit-addressable memory on the Burroughs B1700. The first part of this paper provides the practical motivation for the use of these algorithms. We next present the algorithm itself, accompanied by its mathematical foundation.



Authors' addresses: E. Artzy, Everyman's University, 35 Mishmeret St., Afeka, Israel; J. A. Hinds, Computer Science Department, University of Utah, Salt Lake City, UT 84112; H. J. Saal, IBM Israel Scientific Center, Computer Science Building, Technion City, Haifa, Israel.

## 2. Motivation

The Burroughs B1700 imposes no hardware constraints (or advantages) on the choice of container size (byte, word, etc.) from 1 to 24 bits in width. (See [1] for some consequences in memory utilization.) A natural convenience of integral powers of 2 is the simplicity of using shifts to convert from one set of units to another. The interpreters we have written for the B1700 use a variety of other widths (e.g. 18, 34, etc.), and we must multiply or divide by small integers such as these.

Multiplication by a particular integer using shifts, adds, and subtracts is fairly straightforward. For example, multiplication by 17 is done by a four-bit shift and add; multiplication by 15 is done by a four-bit shift followed by a subtract. These algorithms are presumably optimal for the numbers in question if we restrict ourselves to shifts, adds, and subtracts.

We have frequently found it to be appropriate on the B1700 to use bit addresses rather than unit addresses. We give up maximal addressing capability in a fixed width field, but this has not been a restriction on our designs. There are several places where a unit address is needed, and this requires a fast division method. We now present our technique for rapidly dividing by a given constant. This method easily detects any nonzero remainder, but not its value. In our applications, since we use exact multiples of the unit width, any nonzero remainder would be an error analogous to the System 360 specification exception (see [2]).

## 3. The Algorithm

In this section we present our algorithm in detail, but first we describe the general method and how it works. Division of the dividend $R$ by an arbitrary given integer $d$ is accomplished in two steps. The first simply tests for a possible nonzero remainder by inspecting the low-order bits of $R$, corresponding to any powers of two in the factorization of $d$, and shifts out the zeros. Consequently. the second step is restricted to the case where the divisor is odd.

The inverse of every odd integer (greater than one) is a fraction whose binary representation is of the form $.s_1s_2 \ldots s_n \; s_1s_2 \ldots s_n \; s_1s_2 \ldots$ etc. The bits $s_1s_2 \ldots s_n$ are the representation of the integer $s$ where $ds = 2^n - 1$. Temporarily restrict the possible values of $R$ to the integer multiples of $d$ which are less than $2^n$. Then, the $n$ lower-order bits of the product $(2d)s = 2^{n+1} - 2 \equiv 2^n - 2 \pmod{2^n}$, and in general, for the range we are

Table I. Minimum Values for $s$, $n$, and $w(s)$.

| $d$ | $s$ | $ds = 2^n \pm 1$ | $w(s)$ |
|---|---|---|---|
| 3 | 1 | $2^1+1$ | 1 |
| 5 | 1 | $2^2+1$ | 1 |
| 7 | 1 | $2^3-1$ | 1 |
| 9 | 1 | $2^3+1$ | 1 |
| 11 | 3 | $2^5+1$ | 2 |
| 13 | 5 | $2^6+1$ | 2 |
| 15 | 1 | $2^4-1$ | 1 |
| 17 | 1 | $2^4+1$ | 1 |
| 19 | 27 | $2^9+1$ | 3 |
| 21 | 3 | $2^6-1$ | 2 |
| 23 | 89 | $2^{11}-1$ | 4 |
| 25 | 41 | $2^{10}+1$ | 3 |
| 27 | 19 | $2^9+1$ | 3 |
| 29 | 565 | $2^{14}+1$ | 5 |
| 31 | 1 | $2^5-1$ | 1 |
| 33 | 1 | $2^5+1$ | 1 |
| 35 | 117 | $2^{12}-1$ | 4 |
| 37 | 7085 | $2^{18}+1$ | 6 |
| 39 | 105 | $2^{12}-1$ | 4 |
| 41 | 25 | $2^{10}+1$ | 3 |
| 43 | 3 | $2^7+1$ | 2 |
| 45 | 91 | $2^{12}-1$ | 4 |
| 47 | 178481 | $2^{23}-1$ | 8 |
| 49 | 42799 | $2^{21}-1$ | 7 |

temporarily considering, when $R = kd$, then $Rs \equiv 2^n - k \pmod{2^n}$. Consequently, in order to determine $k$, the quotient we desire, we multiply the dividend $R$ by $s$, a constant determined from $d$ (see Table I), and then take the ($n$-bit) two's complement.

What happens if the number $R$ is not an exact multiple of $d$? It is proved in the Appendix: (a) that only the numbers which are exact multiples of $d$ are mapped onto $2^n - 1, 2^n - 2, \ldots, 2^n - k$, and (b) that numbers in the same remainder class are mapped next to each other by the multiplication process.

This result allows us to check for a nonzero remainder with one comparison. One can find both quotient and remainder of any dividend by including at most $d$ comparisons and subtractions after multiplication. (Note that as $d$ grows large, the number of comparisons will become prohibitive).

Handling dividends larger than $2^n$ is also straightforward. We need to multiply the dividend by several periods of $s$, which is easily accomplished by first multiplying by $s$, then by $2^n + 1$, $2^{2n} + 1$, $2^{4n} + 1$, etc. We double the maximum dividend width in each step, at the cost of only a multibit shift and add.

With this general description as background, we present our algorithm in detail. Let us introduce some additional notation.

For an $l$-bit register $R$ and a positive constant $d$, let $m$, $n$, and $s$ be any positive integers such that $ds = 2^m(2^n \pm 1)$ and let $q_{\max} = \lfloor (2^l - 1)/d \rfloor$ and $k_l = \max\{\lceil (\log_2 (l/n)) - 1\rceil, 0\}$. (The existence of such integers is guaranteed by Lemma 2 of the Appendix.) For this algorithm one auxiliary register, say $T$, is used. In addition, one counter, say $J$, is needed.[1] We use $R$ and $T$ to denote the contents of registers $R$ and $T$ respectively. To simplify the solution we assume that all bits shifted over word boundaries are lost and the overflow bit after arithmetic operations is ignored.

## The Algorithm

We choose to represent the algorithm as a macroprogram in order to emphasize the method of constructing a specialized "division by $d$" algorithm once the values of $d$ and the register width $l$ have been selected. The variable $SIGN$ has the value '+' when according to Table I $ds$ is of the form $2^m(2^n + 1)$ and '−' otherwise.

```
BEGINMACRO
IF m ≠ 0 THEN
          OUTPUT('if the m least significant bits of R are not zero
            then return("R is not divisible by d")
            else R ← R shifted right by m bits;');
OUTPUT('R ← R times S;');
IF n ≤ l THEN
  DO OUTPUT('T←R shifted left by n bits;');
  IF SIGN = '+'
    THEN OUTPUT('R←T−R;')
    ELSE OUTPUT('R←T+R;');
  OUTPUT('for J = 1 to k_l
                do
                T ← R shifted left by n·2^J bits;
                R ← R + T
                end;
            R←2^l−R;')
  END
  ELSE IF SIGN = '−' THEN OUTPUT ('R←2^l−R;');
OUTPUT('if R≤q_max
  then return("R is the quotient desired")
  else return("R is not divisible by d");')
ENDMACRO
```

We note that to perform the indicated multiplication of $R$ times $S$ a variety of techniques can be used. In particular, if we restrict ourselves to binary shifts, adds, and subtracts, the arithmetic weight $w(s)$ is an upper bound on the number of shift and add or shift and subtract steps required. $w(s)$ is defined as

$$\min\left[\sum_{i=0}^{n} |a_i|\right] - 1, \text{ where } a_i = \{0, \pm 1\}$$

$$\text{such that } s = \sum_{i=0}^{n} a_i 2^i.$$

(Values of $w(s)$ are listed in Table I.) Without these restrictions even faster means of multiplication may be found, for example, by using table lookup methods on groups of bits; however, we shall not elaborate further.

## 4. Discussion

After this work was completed, a Correspondence by Jacobsohn [3] appeared, in which he also considered algorithms for division by fixed integers. Jacobsohn

[1] In practice, the iterative loop is unrolled and no counting is employed.

Fig. 1. B1700 microcode for division by 18.

```
MOVE DIVIDEND TO Y          * uses X and Y registers
IF LSBY TRUE THEN GO TO      TO NON-ZERO-REMAINDER
SHIFT Y RIGHT BY 1 BIT-     * division by 2
MOVE Y TO X
SHIFT X LEFT BY 3 BITS
MOVE DIFF TO Y              * times 7 (DIFF is X-Y)
MOVE Y TO X
SHIFT X LEFT BY 6 BITS
MOVE SUM TO Y               * times 2**6+1 (SUM is X+Y)
MOVE Y TO X
SHIFT X LEFT BY 12 BITS
MOVE SUM TO Y               * times 2**12+1
LIT 0 TO X
MOVE DIFF TO X              * complement to get result
LIT 932068 TO Y
IF X≥Y THEN GO TO NON-ZERO-REMAINDER
MOVE X TO QUOTIENT          * with no remainder
```

Fig. 2. B1700 microcode for conventional integer division algorithm producing quotient and remainder.

```
        MOVE DIVIDEND TO X      * uses registers X,Y,FA,FL and T
        MOVE DIVISOR TO FA
        MOVE 24 TO FL
        NORMALIZE               * remove leading zeroes
        MOVE X TO Y
        CLEAR X
.LOOP   SHIFT XY LEFT BY 1 BIT
        MOVE Y TO T             * save low-order dividend and
        MOVE FA TO Y            * quotient bits
        IF X≥Y THEN BEGIN       * trial subtraction
                  MOVE DIFF     * subtract divisor
                    TO X
                  SET T(23)     * set quotient bit on
                  END
        MOVE T TO Y             * restore dividend/quotient
        COUNT FL DOWN BY 1      * for shifting, reduce loop count
        IF FL≠0 THEN GO TO      * test for completion
          -LOOP
        IF X≠0 THEN GO TO NON-  * remainder is in X
          ZERO-REMAINDER
        MOVE Y TO QUOTIENT      * quotient in Y
```

presents a combinational algorithm for division using multiplication by a fractional inverse, followed by "suitable rounding" so that the integer part of the result is the true quotient. The remainder itself is found by remultiplying the fractional result by the divisor, whereas we require at most $d$ compare and subtracts. Thus we have demonstrated an algorithm and proof that *both* quotient and remainder can be completely determined from the low-order ("fractional") bits, without requiring a double width product. Jacobsohn's approach is superior when the quotient is required and there is no expectation that the remainder will be zero.

Jacobsohn did not observe that multiple periods of the inverse can be handled using the shift-and-add technique shown here. Thus his method takes time linear in the ratio of the register width to the period of the inverse, while we take time proportional to the logarithm of this ratio. (This aspect of the method can be incorporated in Jacobsohn's scheme if desired). We also incorporate recognition of those cases where there is a factor of the inverse in the form $2^n + 1$. This simplifies the initial multiplication step and may, in specific cases, provide sufficient result bits directly, without requiring complementation.

The algorithm presented here has proved quite useful in the practical implementation of interpreters with a variety of unit widths. The method presented can readily be adapted to a hardware implementation along the lines of Jacobsohn's. Its speed results from the fact that, once past step (ii), we double the resulting precision each iteration, at the cost of a single shift and add.

Figure 1 illustrates microcode written for a B1700 [4] to implement division by 18 of a 24-bit dividend. Each line represents one microinstruction, and the sequence takes approximately 2.8 microseconds on a B1726 (independent of quotient). By way of contrast, the fastest general purpose division routine of which we are aware (see Figure 2), takes about 42 microseconds with 18 used as the divisor for "random" quotients. (The actual time varies for quotients with differing numbers of leading zeroes, and the number of divisor subtractions performed.)

## Appendix

Let $N$ denote the set of natural numbers and $N^+$ denote the set $N - \{0\}$. If $x$ is a positive rational number, then $\lfloor x \rfloor$ denotes the greatest natural number smaller than or equal to $x$ and $\lceil x \rceil$ denotes the smallest natural number greater than or equal to $x$. If $a$, $b$, $m$ are in $N$ then $a \pmod m$ denotes the smallest integer $b$ such that $a - b$ is divisible by $m$. (We also write $b = a \pmod m$.)

For $l$ in $N$ let $X_l = \{0, 1, \ldots, 2^l - 1\}$.

For $d$ in $N^+$ let $\sigma_d$ be a function from $X_l$ into $N \times N$ defined by $\sigma_d(x) = (q, r)$ if, and only if, $x = qd + r$ with $r < d$.

Let $f$ be a function from $X_l$ onto $X_l$ and let $Y_l$ be the array $< y_0, y_1, \ldots, y_{2^l-1} >$ defined by $y_i = \sigma_d(f^{-1}(i))$.

LEMMA 1. *If $f$ is a function such that $f(x) = q$ whenever $x = dq$, then for $j \in \{0, 1, \ldots, \lfloor (2^l - 1)/d \rfloor\}$, $y_j = (j, 0)$.*

PROOF. By definition $y_j = \sigma_d(f^{-1}(j))$ and (since $0 \le j \le (2^l - 1)/d)$ $j = f(dj)$; hence $y_j = \sigma_d(dj) = (j, 0)$. □

The following lemma gives us an alternative way for representing $d$.

LEMMA 2. *For every $x$ in $N$ there exist three positive integers $m$, $n$, and $s$ such that $xs = 2^m(2^n \pm 1)$.*

PROOF. This lemma is easily proved by the Euler-Fermat Theorem (see, for example, [5], Theorem 22). □

Let $d$ be in $N$. Let $m$, $n$, and $s$ be any positive integers such that $ds = 2^m(2^n \pm 1)$. Note that the existence of such integers is guaranteed by Lemma 2. (In Table I we list minimum values of $s$ and $n$ for odd values of $d$ smaller than 50.)

To avoid a cumbersome notation, we assume, for the rest of this section, that $d$, $m$, $n$, and $s$ are fixed (but arbitrary) integers satisfying the above conditions.

For $l$ in $N$, let $k_l = \max\{\lceil(\log_2(l/n)) - 1\rceil, 0\}$ and let

$$T_l = (s(2^n \mp 1)/2^m) \prod_{i=1}^{k_l} (2^{2^i n} + 1).$$

LEMMA 3. *For $l$ in $N$, let $f_l$ be a function from $X_l$ into $X_l$ defined by $f_l(x) = -T_l x(\text{mod } 2^l)$ (where $-a(\text{mod } 2^l)$ is interpreted as $2^l - a(\text{mod } 2^l)$). (i) $f_l$ satisfies the conditions of Lemma 1. (ii) If $d$ is an odd number, then $f_l$ is a permutation on $X_l$.*

PROOF:

(i) For $l$ in $N$, we have

$$\begin{aligned} -T_l x(\text{mod } 2^l) &= -T_l\, dq(\text{mod } 2^l) \\ &= -\left(\prod_{i=1}^{k_l} (2^{2^i n} + 1)\right) \\ &\quad \cdot (s(2^n \mp 1)/2^m)\, dq(\text{mod } 2^l) \end{aligned}$$

by taking

$$\begin{aligned} d &= 2^m(2^n \pm 1)/s \\ &= -\prod_{i=1}^{k_l} (2^{2^i n} + 1)(2^{2n} - 1)q(\text{mod } 2^l) \\ &= -(2^{2^{k_l+1} n} - 1)q(\text{mod } 2^l), \end{aligned}$$

but $2^{k_l+1}n > l$ and hence

$$-T_l x(\text{mod } 2^l) = q.$$

(ii) If $d$ is an odd number then $m = 0$ and $T_l$ is an odd number. $T_l$ is relatively prime to $2^l$ and the result follows. □

We will show now that for $d$ odd our function $f_l$ introduces a certain order on $y_l$. Note that we have already shown in Lemma 1 that, for $j$ smaller than or equal to $(2^l - 1)/d$, $y_j = (j, 0)$.

First we need to describe the function $f_l^{-1}$.

LEMMA 4. *For $l$ in $N$, thc function $f_l^{-1}$ is given by $f_l^{-1}(z) = zd(\text{mod } 2^l) = x$.*

PROOF. Let $l$ be in $N$. We want to show that if $f_l(x) = z$ then $dz(\text{mod } 2^l) = x$. But $z = f(x) = -T_l x(\text{mod } 2^l)$; hence

$$\begin{aligned} {}_l^{-1}(z) &= f_l^{-1}(-T_l x(\text{mod } 2^l)) \\ &= -dT_l x(\text{mod } 2^l) \\ &= ((-dT_l)(\text{mod } 2^l)x(\text{mod } 2^l))(\text{mod } 2^l) \\ &= 1 \cdot x(\text{mod } 2^l) \\ &= x. \end{aligned}$$ □

The next result describes the ordering on $Y_l$ imposed by $T_l$.

LEMMA 5. *Let $l$ be in $N$. For $k$ in $\{0, \ldots, 2^l - 1\}$, if $y_k = (q, r)$, then*

$$y_k + 1 = \begin{cases} (q + 1, r) & \text{if } (q + 1)d + r < 2^l, \\ (0, r') & \text{otherwise,} \end{cases}$$

*where $r'$ is given by $((q + 1)\, d + r)(\text{mod } 2^l)$.*

PROOF. Let $l$ be in $N$ and let $k$ be in $\{0, \ldots, 2^l - 1\}$. By definition

$$y_{k+1} = \sigma_d(f_l^{-1}(k + 1));$$

hence (by Lemma 4)

$$y_{k+1} = \sigma_d((k + 1)\, d(\text{mod } 2^l)),$$

and so

$$y_{k+1} = \sigma_d((kd(\text{mod } 2^l) + d)(\text{mod } 2^l)).$$

Now $y_k = (q, r) = \sigma_d(f_l^{-1}(k))$, and by Lemma 4 $y_k = (q, r) = \sigma_d(kd(\text{mod } 2^l))$, and by definition of $y_k$, $kd(\text{mod } 2^l) = qd + r$; thus

$$\begin{aligned} y_{k+1} &= \sigma_d((qd + r + d)(\text{mod } 2^l)) \\ &= \begin{cases} (q + 1)d + r & \text{if } (q + 1)d + r < 2^l, \\ (0, r') & \text{otherwise.} \end{cases} \end{aligned}$$ □

COROLLARY. *If the elements of $Y_l$ are ordered in the following way:*

$$Y_l = \langle (0, 0), (1, 0) \ldots (0, r_1), (1, r_1) \ldots (0, r_2), (1, r_2) \ldots (0, r_{d-1}), (1, r_{d-1}) \ldots \rangle,$$

*then $r_i = ir_1$ (mod $d$).*

Received August 1973; revised May 1975

**References**

1. Wilner, W.T. Burroughs B1700 memory utilization. Proc. AFIPS 1972 FJCC, Vol. 41, pp. 579–586, AFIPS Press, Montvale, N.J., 1972.
2. *System 360 Principles of Operation*. GA22-6821, IBM Corp., 1970.
3. Jacobsohn, D.H. A combinatoric division algorithm for fixed integer divisors. *IEEE Trans. Comput. C-22*, 6 (June 1973), 608–610.
4. *B1700 Systems Reference Manual*. 1057155, Burroughs Corp., 1972.
5. Hunter, J. *Number Theory*. Oliver and Boyd, London, 1964.