*Proceedings of 1990 International Conference on Application Specific Array Processors,* Princeton University, New Jersey, September 5-7, 1990, 6 pages

# AN EFFICIENT DIVIDER FOR LARGE OPERANDS - A SUMMARY

Kathleen A. Kramer
Rodney M. F. Goodman
*California Institute of Tech.*
*Pasadena, California*

Anthony J. McAuley
*Bell Communications Research*
*Morristown, New Jersey*

## ABSTRACT

*In this paper a new binary divider is proposed for n bit integer operands which produces the quotient and remainder in $O(n)$ time using $O(n^2)$ area for the parallel implementation and $O(n)$ area with the serial/parallel implementation.*

## 1. INTRODUCTION

Efficient VLSI methods for implementing the four basic arithmetic operations have recently pushed back many system performance limitiations. However, when compared with improvements in performance of addition, subtraction, and multiplication, division has been relatively ignored. There is continued need for re-evaluation of arithmetic architectures, since the efficiency of implementation can be divorced from neither the implementation technology nor the size of operand. As a result of VLSI many new archtitectures are coming into favor, either because of the changing importance of design factors or because it is now possible to design bigger chips.

Existing area efficient division methods, such as the SRT and CORDIC techniques [1] have O(n logn) worst case delay, where n is the number of bits of the operands. Purdy and Purdy did develop an O(n) time division algorithm [2]; it requires approximately 3*n steps to calculate the remainder and n more for the quotient. Our algorithm calculates the remainder and quotient simultaneously in approximately n steps, about one-fourth of those required by the Purdy and Purdy divider.

Because of the large operands they require, public key cryptography and many other digital signal processing applications would be excellent applications for this architecture.

Section 2 defines and explains the problem of division. Section 3 will briefly covers some existing divider schemes. Section 4 explains our algorithm. Section 5 shows how and why the algorithm works. Section 6 is devoted to the parallel and serial parallel systolic implementations of the divider.

## 2. DIVISION

The division operation involves finding the quotient (Q) and remainder (R) of a dividend (X) and divisor (Y). These four parameters satisfy the equation $X = Q * Y + R$
(where R is less than Y and of the same sign as X). For a given 2n bit X and n bit Y, there is a n+1 bit Q and n bit R.

The paper and pencil method of division, using the binary operands X = 010011 and Y = 011 (X = 19 and Y = 3 in decimal numbers) is demonstrated briefly with the following:

$$\begin{array}{rrl}
 & \underline{\quad 0110} & = Q \\
Y = 011 & |010011 & = R^0 \\
 & \underline{-\ 01100} & = 2^2Y \\
 & 111 & = R^1 \\
 & \underline{-\ 00110} & = 2^1Y \\
 & 001 & = R^2 = R
\end{array}$$

Although we get the final remainder after two subtractions ($2^2$Y and $2^1$Y) the partial remainder was compared with $2^3$Y ($2^3Y > R^0$) and $2^1$Y ($2^1Y > R^2$); but since the current partial remainder was less than these multiples of the divisor, they were never subtracted.

Division and multiplication are in many respects dual operations. As a shift and subtract process division superficially resembles the shift and add method of multiplication. Division, however, requires the results of one subtraction to determine the next one. This introduces a sequential ordering in the subtraction of multiples of the divisor from the partial remainders that is not present in the addition of the partial products. Implementations of division in the integer ring (that is, normal arithmatic, with carries) have been slower than the equivalent multiplication operation. A parallel multiplier has O(logn) worst case delay, while the area efficient serial parallel multiplier (SPM) , has O(n) worst case delay[1]. Division on the other hand has no equivalents (with the same performance) to these multiplier architectures.

## 3. EXISTING DIVIDER IMPLEMENTATIONS

Conventional divider implementations [1] use some variant of the paper and pencil method of division: either doing explicit comparison and subtraction (non-restoring), or by using subtraction and compensating addition (restoring division).

Often, using iterative techniques [1] , the faster multiplier has been used to speed up division. However, this is very area inefficient, unless a multiplier is already required. Other methods, notably the SRT and CORDIC algorithms [1], employ a fast adder in their design. However, all these methods have O(nlogn) worst case gate delay (both require n n-bit additions).

The Purdy and Purdy algorithm [2] performs division in O(n) time. It differs from most dividers in that the remainder is calculated first by performing three consecutive tests/subtractions on the most significant bits of a carry-save representation of the partial remainder successively eliminating the most significant bits. After the remainder is finally computed, it is subtracted from the dividend to make a "divisible dividend" the quotient is then generated from least to most significant bit by successively testing the even/oddness of the dividend and subtracting.

## 4. NEW DIVIDER ALGORITHM

The following is the divider algorithm in program form. An n+m bit dividend is divided by an n bit divisor to generate an m+1 bit quotient and an n bit remainder. Sum-carry additions (the carries are not propagated but stored as inputs to the adder during the next add) are performed m times. Three boolean functions (f0, f1, and f2) of the three most significant bits of the divisor, the PRSUM, and the PRCARRY registers determine whether zero, the current multiple of the divisor, or twice the current multiple of the divisor is to be subtracted from the current partial remainder (stored as the sum of the PRSUM and PRCARRY registers).

```
INPUT    DIVIDEND[n + m], DIVISOR[n];
OUTPUT   QUOTIENT[m + 1], REMAINDER[n];

begin
    PRSUM:= DIVIDEND; PRCARRY:=0; SD[-1]:= 2^M*DIVISOR;
    i:= 0;
    while (i <= m - 1)
         do begin
            SD[i] := SD[i - 1]/2;
            if ( f0) then
            begin
                PRSUM, PRCARRY := PRSUM + PRCARRY;
                QC[m - i] := 0; QS[m - i]:=0;
            end;
            else if (f1) then
            begin
                PRSUM, PRCARRY := PRSUM + PRCARRY - SD[i];
                QC[m - i] := 0; QS[m - i]:=1;
            end;
            else
            begin
                PRSUM, PRCARRY:= PRSUM + PRCARRY - 2*SD[i];
                QC[m - i] := 1; QS[m - i]:=0;
            end;
            i:= i + 1;
{f0 is (S = 0 and C = 0) and (S-1 = 0 or C-1 = 0)
 f1 is (((S = 1 or  C = 1) and ( S-1 = 0 or C-1 = 0) and( (S-1 != C-1) or
      (maj. of S-2, C-2, and D-2 = 0)) or (S = 0 and C = 0 and S-1 = 1 and C-1 = 1))
 f2 is (not f0 and not f1)  Where S and C are the msb's of PRSUM and PRCARRY, S-1
and C-1 are the next most significant bits and S-2, C-2, and D-2 are the third most
significant bits of PRSUM, PRCARRY, and the two's complement of the divisor
}
         end;
     REMAINDER := PRSUM + PRCARRY;
     if (REMAINDER > 2*DIVISOR) then
     begin
         REMAINDER := REMAINDER - 2*DIVISOR; QC[0]:= 1; QS[0] := 0;
     end;
     else if (REMAINDER > DIVISOR) then
     begin
         REMAINDER := REMAINDER - DIVISOR; QC[0]:= 0; QS[0] := 1;
     end;
     else
```

```
        begin
            QC[0]:= 0; QS[0] := 0;
        end;
        QUOTIENT := 2*QC + QS; {QC and QS are the integers
                                formed by the bits QC[i], QS[i]}
end;
```

## 5. WHY IT WORKS

To avoid a long comparison with the entire partial remainder (PR); the n bit partial remainder (contained in an n bit SUM and an n bit CARRY register) is not compared with an n bit shifted over version of the divisor ($SD_n$). Instead, an n - 1 bit version ($SD_{n-1}$) is subtracted no times, once, or twice depending on the values of the highest order bits of the carry save representation of the partial remainder; leaving an n-1 bit partial remainder. After iteration i the partial remainder is known to take up a maximum of n - i bits in PRSUM and n - i bits in PRCARRY. In addition, this partial remainder has a total value of less than $2^{n-i}$. The final result of these subtractions, after the divisor itself is subtracted, is contained in a PRSUM and PRCARRY each with possibly as many bits as the divisor. The result may not be the remainder, but is either the remainder, the remainder plus the divisor, or the remainder plus twice the divisor. So PRSUM and PRCARRY must be added together and the divisor must be compared or subtracted twice to compute the remainder. The quotient is in sum-carry form as the number of times each $SD_i$ was subtracted.

## 6. ARCHITECTURES

Figures 1a and 1b show the two building blocks used in the divider implementation For a serial-parallel implementation, the cells contain latches to store the results from each iteration around the while loop ( see program in section 4). For a parallel implementation, the latches are optional, and can be used to improve throughput. Figure 1a is the standard bit level adder cell [3] with a small change in select logic. A multiplexor is used in order to select from the three possible inputs (0, -1, or -2 times the divisor). This additional logic does not increase the worst case delay. Figure 1b is the cell which looks at three bits of the operands to decide on the subtraction multiples of the divisor (setting QC and QS). This cell is significantly more complex than that of figure 1a, and is approximately 30% slower.

Figure 2 shows an O(mn) array of cells, a parallel implementation of the algorithm which calculates the main loop of the program by distributing the iterations on a row by row basis. Latches can be used to buffer the results allowing the array to be pipelined to accept new operands each clock cycle.

Figure 3 shows a linear chain of O(m+n) cells which can be used for a serial-parallel implementation. It calculates the main looop of the program by doing one iteration each clock cycle and storing the intermediate results in latches.

## 7. CONCLUSION/ FUTURE WORK

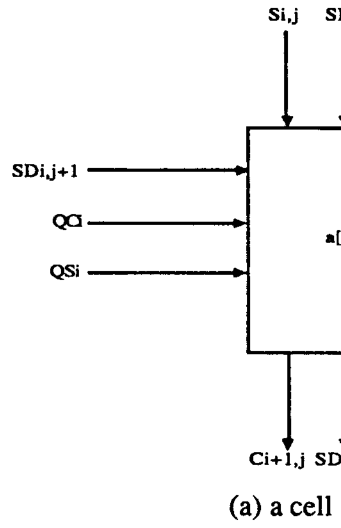
This paper describes a fast algorithm together with parallel and serial-parallel implementations that offer better asymptotic throughput than existing algorithms. The serial-parallel implementation appears particularly attractive; dividing in approximately n steps with n + m cells, one-fourth the steps and cells used for the Purdy and Purdy divider (the test conditions slightly reduce our clock rate, so the true speed up is around three times). More work is needed to replace the global control lines (QC and QS) with a systolic pipeline. Also, an asynchronous logic implementation appears to offer significant benefits for speeding up the division algorithm.

## 8. REFERENCES

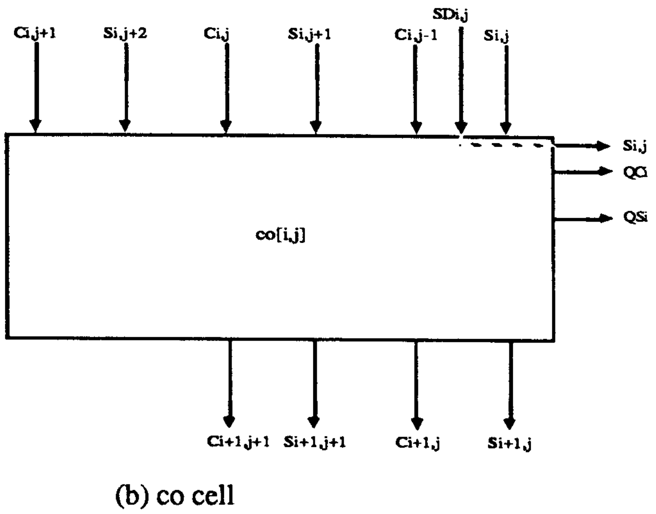
[1] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design.* New York: Wiley, 1979.

[2] C. N. Purdy and G. B. Purdy, "Integer Division in Linear Time with Bounded Fan-In," *IEEE Trans. on Computers*, vol. C-36, pp. 640 - 644, May 1987.

[3] H. T. Kung, "Why Systolic Architectures?" *IEEE Computer*, vol.15, pp. 37-46, Jan. 1982.



(a) a cell



(b) co cell

**Figure 1 (a) and (b)**

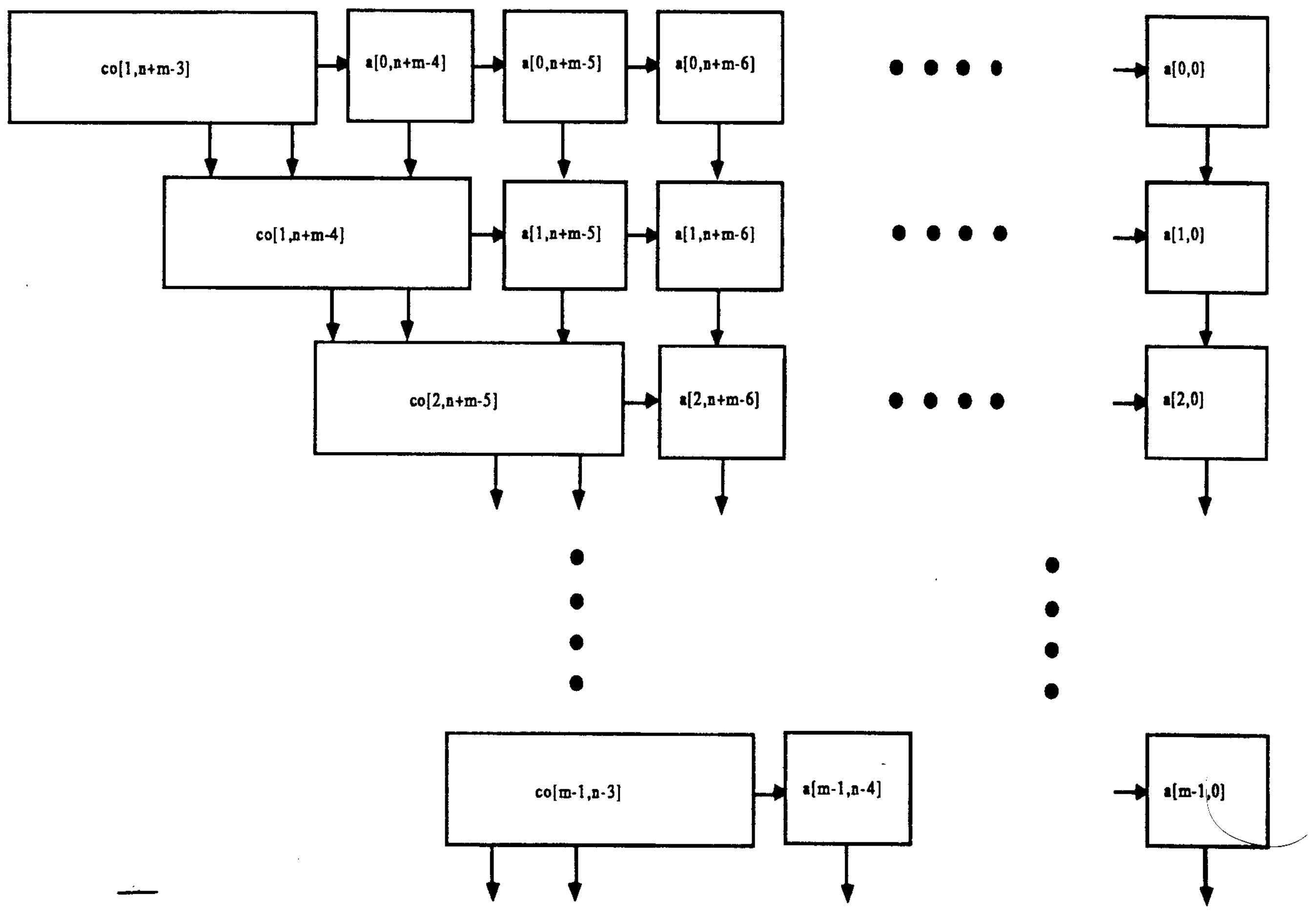


**Figure 2: Parallel Implementation**

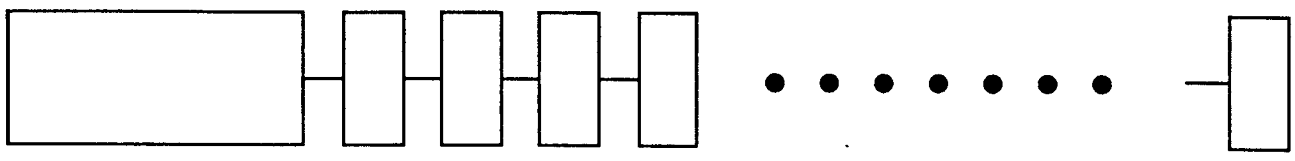


**Figure 3: Serial/Parallel Implementation**