Cellular Logic Array for Computation of Squares [1]

M. Shamanna, S. Whitaker and J. Canaris
NASA Space Engineering Research Center
for VLSI System Design
University of Idaho
Moscow, Idaho 83843

Abstract- **A cellular logic array is described for squaring binary numbers. This array offers a significant increase in speed, with a relatively small hardware overhead. This improvement is a result of novel implementation of the formula $(x+y)^2 = x^2 + y^2 + 2xy$. These results can also be incorporated in the existing arrays achieving considerable hardware reduction.**

1 Introduction

The advent of VLSI has spurred a renewed interest in the development of specialized arithmetic circuits. Special arithmetic functions like squares and square-roots are generally implemented in software. However, when a machine is designed for a specific application, wherein squaring is a frequent process, it may prove advantageous in terms of speed to use a hardware implementation. Most of the approaches, reported in literature for squaring and square-rooting, use array multipliers or special purpose arrays which perform a multitude of other operations in addition to squaring. As a result, there are very few arrays which are solely devoted to extraction of squares. However, Dean[1] has reported such a dedicated array which is probably among one of the fastest squaring circuits known, thus far. In addition, Dean's array uses considerably less hardware than other arrays reported so far. Hence Dean's array has been selected as the obvious choice for comparison with the array proposed in this paper. The proposed array, will provide a significant gain in speed, with a very small hardware overhead, as compared to Dean's squarer[1].

2 Algorithm

Dean[1] has not presented a formal algorithm for his implementation. So, the widely used general binary squaring algorithm[3] will be presented first followed by the proposed algorithm for purposes of clarity and easy understanding. The existing algorithm for binary squaring is generally formulated as follows:

$$(1)^2 = (01)_b$$

$$(a_1 1)^2 = (a_1)^2 + (0a_1 01)_b \quad \text{or}$$

[1]This research was supported (or partially supported) by NASA under Space Engineering Research Center Grant NAGW-1406.

$$F_2 = F_1 + (0a_101)_b$$

where $F_1 = (01)_b$ if $a_1 = 1$ and $F_1 = (00)_b$ otherwise. Similarly, we have

$$(a_2a_11)^2 = (a_2a_1)^2 + (00a_2a_101)_b, \text{ or}$$

$$F_3 = F_2 + (00a_2a_101)_b$$

In general if $a_{r+1}= 1$ then,

$$F_{r+1} = F_r + D_r$$

where $F_r=(a_ra_{r-1}\dots a_2a_1)^2$ is the r^{th} square and $D_r = \overbrace{00....0}^{r\,times}a_ra_{r-1}\dots a_101$ is called the r^{th} radicand. It is obvious that $F_{r+1} = F_r$ if $q_{r+1} = 0$. The above iterative formula applies for all $r = 1, 2, \dots, n$. Figures 4 and 5 show the schematic details of a three bit squaring array for the above algorithm[3].

The proposed algorithm makes use of the well known formula $(x + y)^2 = x^2 + y^2 + 2xy$. Consider a three bit number $(a_22^2 + a_12^1 + a_02^0)$. The LSB–1 and LSB of the square of any number will respectively be 0 and LSB of the original number itself. Therefore,

$$(a_22^2 + a_12^1 + a_02^0)^2 = (a_2 + a_1)2^4 + (a_2a_0)2^3 + (a_1a_0 + a_0)2^2 + a_0.$$

The same result can also be achieved by the repeated application of the formula $(x + y)^2 = x^2 + y^2 + 2xy$ where y is the LSB and x is the rest of the binary number.

$$\begin{aligned}(\underbrace{a_22^1}_{x} + \underbrace{a_12^0}_{y})^2 &= \underbrace{(a_22^1)^2}_{x^2} + \underbrace{2(a_2a_12^1)}_{2xy} + \underbrace{a_12^0}_{y^2} \\ &= (a_2)2^2 + (a_2a_1)2^2 + a_12^0 \\ &= (a_2 + a_2a_1)2^2 + a_12^0 \end{aligned} \quad (1)$$

Also,

$$\begin{aligned}(\underbrace{a_22^2 + a_12^1}_{x} + \underbrace{a_02^0}_{y})^2 &= \underbrace{(a_22^2 + a_12^1)^2}_{x^2} + \underbrace{2(a_2a_02^2 + a_1a_02^1)}_{2xy} + \underbrace{a_02^0}_{y^2} \\ &= (a_22^1 + a_12^0)^22^2 + (a_2a_02^3 + a_1a_02^2) + a_02^0 \end{aligned} \quad (2)$$

Equation 1 proves that the LSB–1 bit and the LSB of the final answer is always 0 and the LSB of the original number itself respectively. Since multiplication by 2 implies a left-shift by one bit position the term $(2a_2a_1)$ has been shifted from the 2^1 bit position to 2^2 bit position in Equation 1. This result for a three bit binary number is realized by the array of Figure 1. The algorithm can easily be extended to any n bit number. The novelness of the algorithm lies in the fact that squaring of the number is carried out in steps coupled with the ingenious use of left-shifts in the bit positions.

3 Comparison

The implementation of the proposed algorithm for a 3 bit and a 4 bit number has been illustrated in Figures 1 and 3 respectively. The proposed array is built of the basic half-adder cell shown in Figure 2. Its function may be defined as follows:
$u = (w + v_{-1}) \oplus (xy)$
$v = (w + v_{-1}) \cdot (xy)$
The symbols + and · stands for the Inclusive-Or and And operations in the above expressions.

The implementation of 3 bit squarer based on Dean's algorithm is also illustrated in the Figures 6 and 7. The basic cell (Figure 7) has two control inputs A and B. The inputs on the lines C and D are added in the cell, S being the sum out and P being the carry out. When both A and B are present, a further digit is added to the sum (and carry), so that the cell then functions as a full-adder[1].

It can be seen that the proposed array has $1 + \sum_{i=3}^{n} i$ whereas Dean's array [1] uses $1 + \sum_{i=1}^{n} i$ cells resulting in a overhead of $(n-2)$ cells. However, the hardware inside the proposed basic cell is much simpler, as it utilizes only half-adders, compared to full-adders in Dean's array. So the increase in the number of cells is offset by the reduction in the complexity of the individual cell. This leads to the authors contention that the hardware overhead which translates into increased chip area is almost negligible. Moreover, the propagation time through the proposed array is only $n\tau$ as compared to $(2n-3)\tau$ which is the delay through Dean's array. The hardware overhead-speed gain relation follows the square law for most specialized arithmetic arrays. Here, an increase in speed has been accomplished with a linear increase in hardware.

The proposed array has a number of unused inputs which can be used to add in an other number so that the array would function as a full squarer (all outputs in 1 state). A specialized array of this sort has a number of applications including the generation of binary logarithms[2] which depends on iterative squaring.

4 Conclusions

A new cellular array for extraction of squares of binary numbers has been presented. An squaring algorithm based on the formula $(x+y)^2$ has been described. The proposed array provides impressive speed gains compared to the existing arrays at the expense of negligible hardware overhead. It is hoped, that the algorithm discussed in this paper will provide fresh insights, to reduce redundant hardware present in most of the existing squaring arrays.

References

[1] K. J. Dean, Cellular Logical Array for Obtaining the Square of a Binary Number, *Electronics Letters*, Vol. 5, Aug. 1969, pp.370–371.

[2] K. J. Dean, A fresh approach to logarithmic computation, *Electron. Engng.*, 41, April 1969, pp.488–490.

[3] K. Hwang, Computer Arithmetic: Principles, Architecture and Design, *John Wiley and Sons*, 1979.

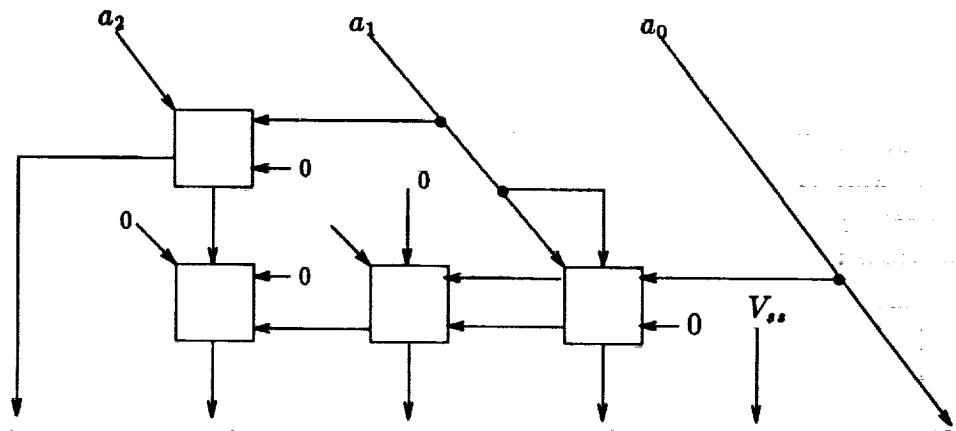

Figure 1: Proposed squaring array for three bit numbers

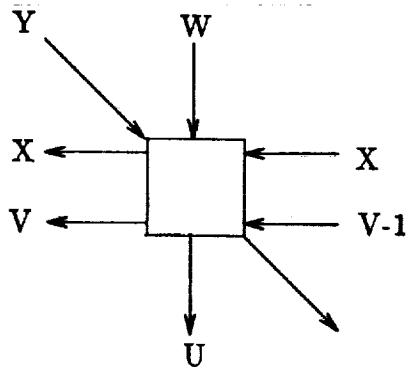

Figure 2: Basic cell used in the proposed squaring array

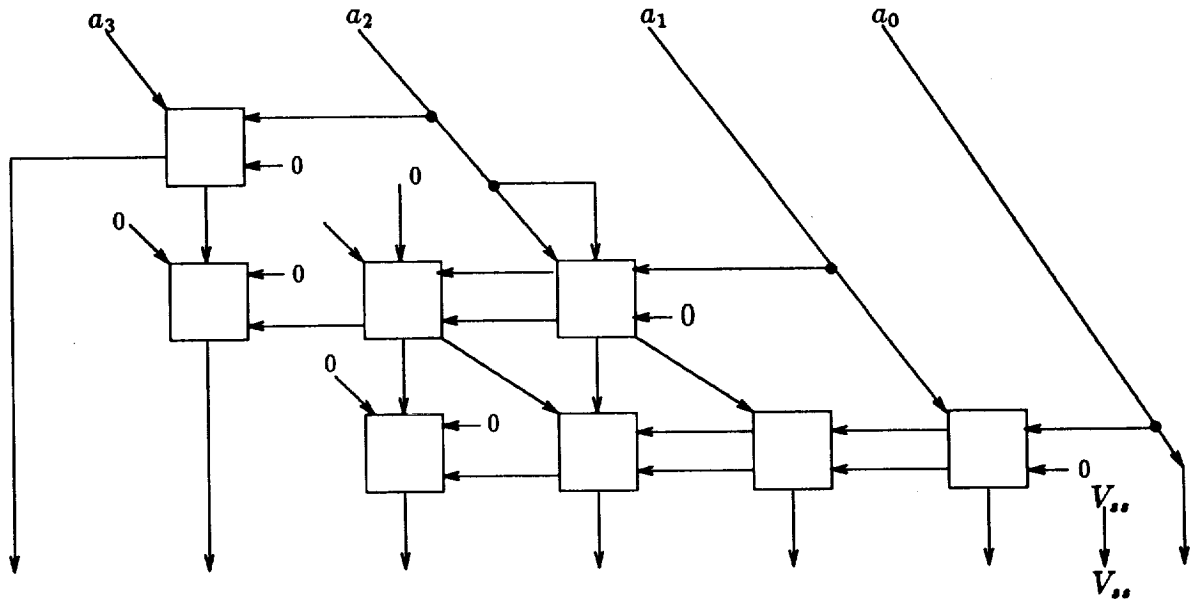

Figure 3: Proposed squaring array for four bit numbers

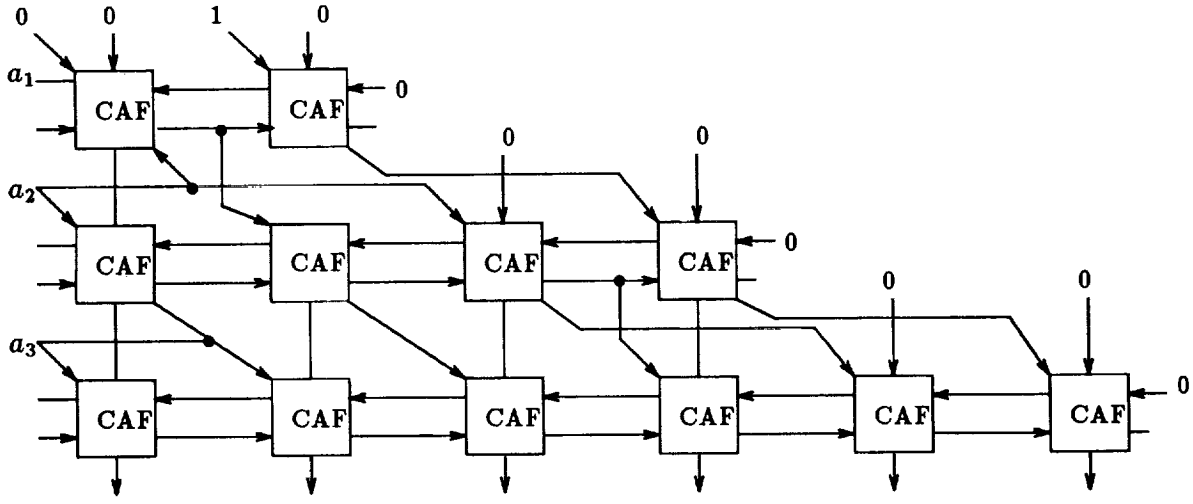

Figure 4: A three bit squaring array using the general algorithm

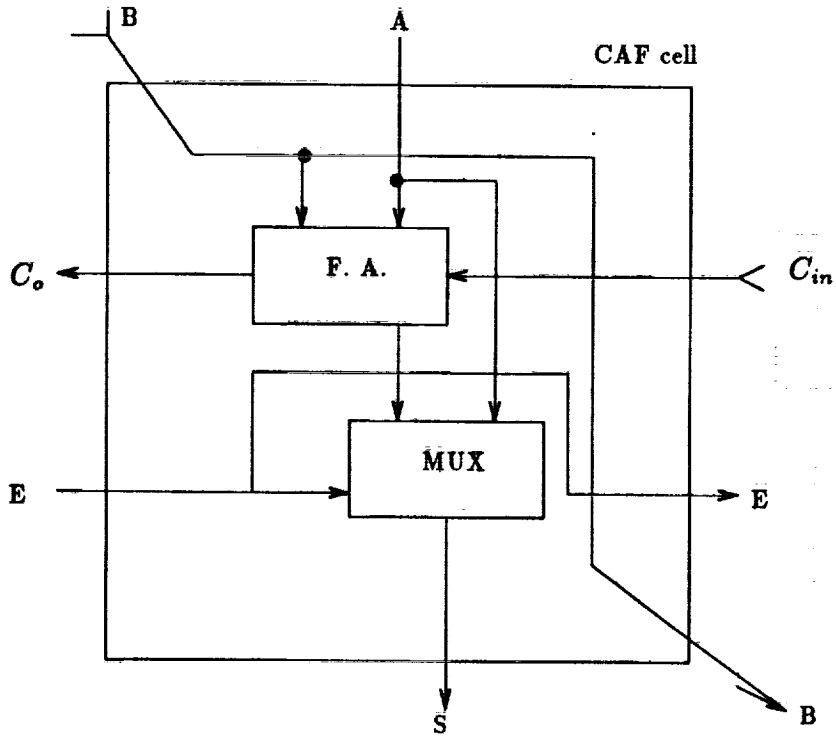

Figure 5: Basic cell used in the general three bit squaring array

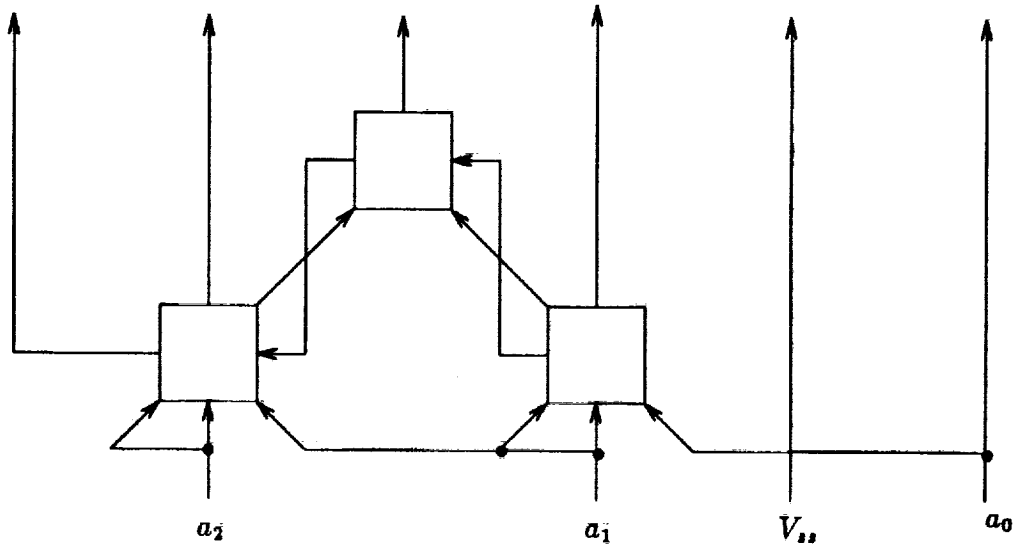

Figure 6: Dean's array for three bit numbers

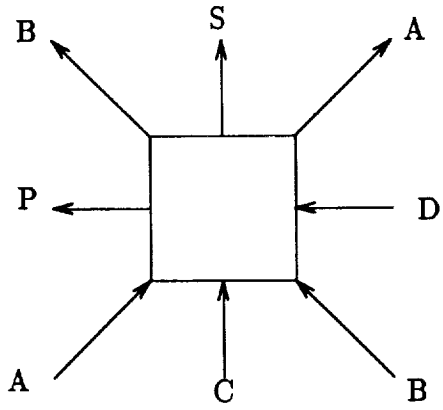

Figure 7: Basic cell used in Dean's array