A Cellular Multiplier for Programmable Logic

Steven A. Guccione

Mario J. Gonzalez

Computer Engineering Research Center

Department of Electrical and Computer Engineering

The University of Texas at Austin

Austin, TX 78712

February 17, 1994

Abstract

Recently, several experimental systems based on programmable logic have been designed and implemented. At the moment, these systems are programmed using a hardware design methodology. If these systems are to gain widespread acceptance, a more traditional software design environment must be developed. One necessary component of this software environment will be a library of standard macrocells corresponding to commonly used arithmetic and logical operations. In this paper a multiplier designed specifically for programmable logic is developed. This multiplier is cellular, highly pipelined and uses only of local interconnections.

1 Introduction

Recently, several general purpose reconfigurable coprocessors based on programmable logic have been designed and built [AS93] [GHK$^+$90] [Kea89]. These machines have demonstrated very high levels of performance using a very small amount of hardware.

One drawback to existing reconfigurable systems is the lack of high level software. Currently, most systems rely on a hardware design as their underlying programming model. Circuit design tools such as hardware design languages and schematic capture must be used to specify the behavior of the coprocessor.

One proposed alternative to the circuit design approach is a high-level language approach based on the data parallel programming methodology [GG93a]. Data parallel programs developed using this methodology may be translated directly into high-performance pipelined circuits. This technique has been demonstrated on some popular computationally intensive algorithms [GG93a] [GG93b] [GG94].

The implementation of these high-performance circuits depends on the existence of an underlying library of high-performance macrocell-style building blocks. These macrocells should be of a very regular structure suitable for implementation on a programmable logic device. They should also be pipelined and rely only on interconnections between neighboring cells.

One of the largest, and perhaps most important functions used by such a general-purpose system is multiplication. What is desired is a high-performance multiplier that can be mapped efficiently to an array of programmable logic cells.

2 Multiplication

Several popular and well-understood methods for multiplying two binary numbers exist. All are based on successive additions to produce a final product. In the standard representation, all pairs $a_i b_j$ for $0 \leq i, j < n$ are produced and added appropriately. The diagram for this general algorithm is shown in Figure 1.

$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & b_3 & b_2 & b_1 & b_0 \\
\cline{4-8}
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

Figure 1: Multiplication of two 4-bit numbers.

While several types of circuits are popularly used to perform multiplication, the one which appears most nearly suited to a cellular implementation is the array multiplier. Figure 2 shows a general diagram of an array multiplier.

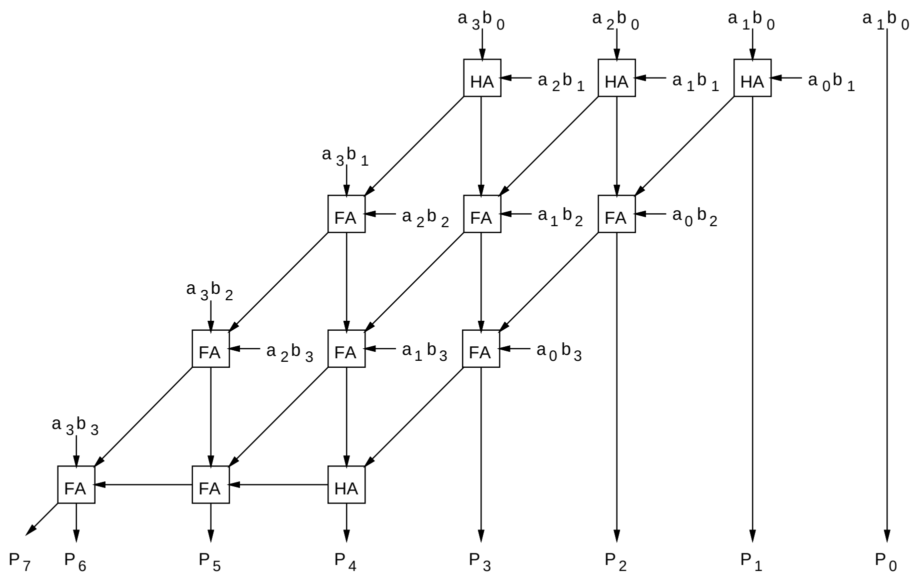

Figure 2: An array multiplier.

This multiplier is essentially a hardware implementation of the standard multiplication algorithm in Figure 1. Standard half adder and full adder cells are used to sum the N^2 partial products $a_i b_j$. This circuit is cellular, pipelinable and, at first glance, appears to use only nearest neighbor interconnections.

Unfortunately, the diagram in Figure 2 ignores a crucial part of the circuit. It is tacitly assumed that all of the necessary partial products $a_i b_j$ are available to all of the appropriate cells. In reality, hardware is necessary to perform the ANDing the bits $a_i b_j$ and interconnections must be used to route signals to their appropriate locations. In the design of a cellular multiplier based on a programmable logic array, the production of these partial products cannot be ignored.

3 Partial Product Generation

In the multiplication algorithm shown in Figure 1 all N^2 combinations of the bits representing the input operands A and B are ANDed together. Perhaps the most obvious method of producing these partial products is an array of N^2 AND gates, as shown in Figure 3.

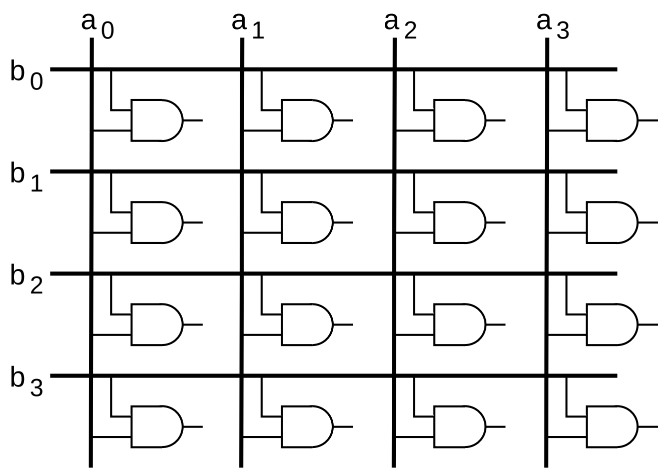

Figure 3: Generation of partial products.

This circuit is cellular and may be implemented using only local interconnections. In this representation, however, the circuit is combinational. All N^2 partial products are generated in parallel.

Another approach is to produce combinations of a_i and b_j systolically. By moving the bits of one operand across the other in the manner of a convolution, between 1 and N partial products are generated per cycle. Figure 4 shows the first two cycles of this operation. This technique has been applied previously to an iterative multiplication method [Swa73].

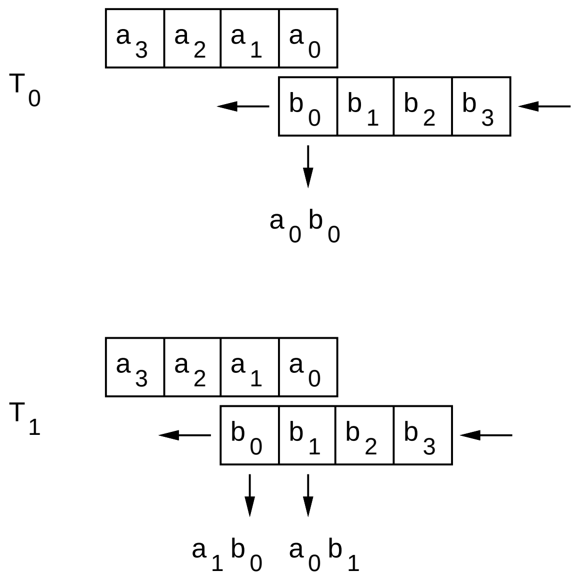

Figure 4: Systolic generation of partial products.

A systolic circuit for performing this operation consists of a downward movement of one input operand and a diagonal movement of the second input operand across the first. At junctures, the partial products are generated by ANDing the two values. All cell outputs are latched. Figure 5 shows the systolic circuit used to produce the partial products. Note the resemblance to the original combinational circuit in Figure 3.

Note that the partial products produced across each row correspond to one

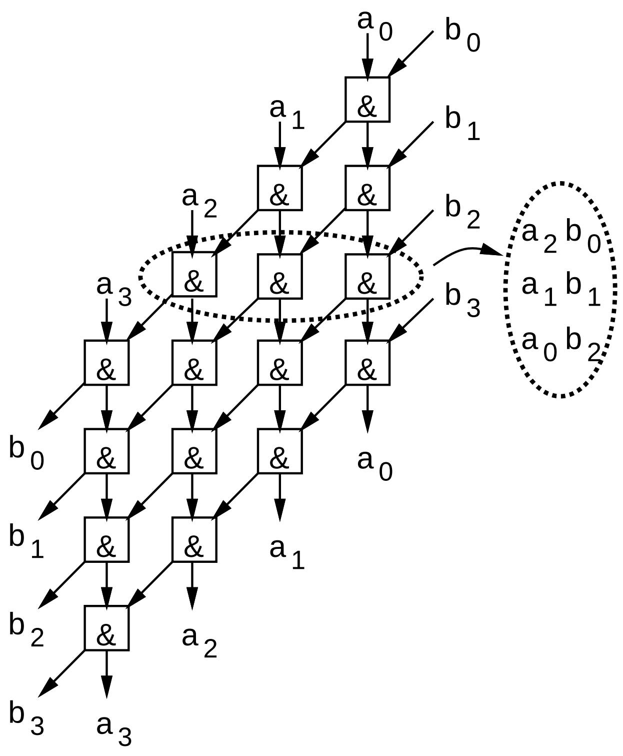

Figure 5: A circuit to systolically produce partial products.

column of partial products from Figure 1. These partial products must be summed, with the carries being propagated to the next column. Unfortunately, adding the partial products in this manner presents a problem. As the A and B operands proceed downward in the array, the sums must be performed horizontally across the row. The downward propagation of data cannot continue until all partial products across a row are summed. A more desirable situation would be to have the partial products generated in columns rather than rows. This would permit data to flow in a downward direction through the array without having to wait for the horizontal propagation of the sums.

Fortunately, there is a simple transformation that permits the partial products to be produced in the desired columns. Figure 6 shows the systolic circuit.

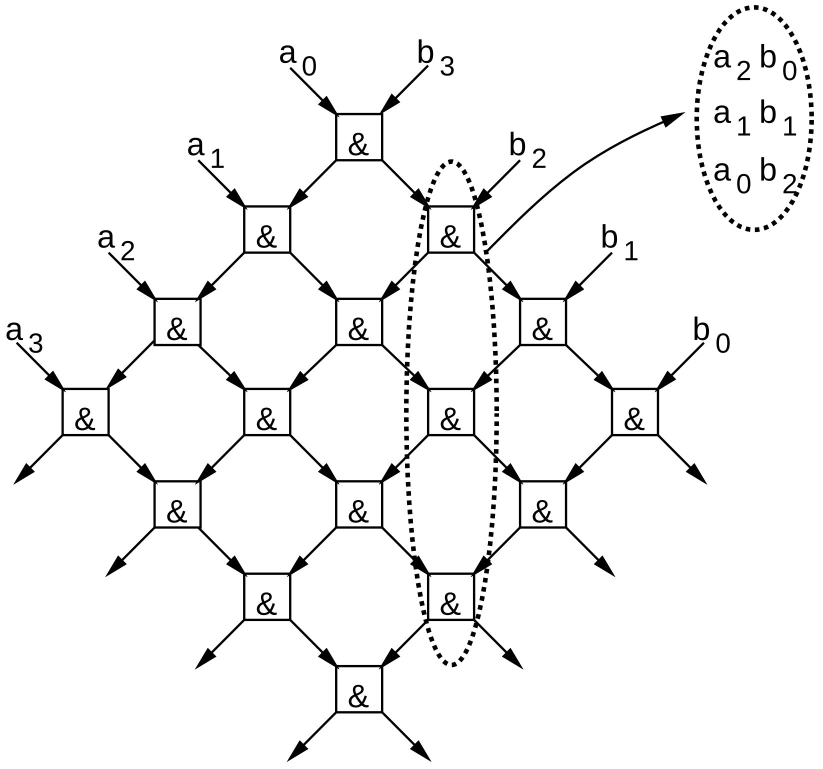

Figure 6: The transformed systolic circuit.

Another desirable effect of this transformation is the re-ordering of the bits of the input operands. In the original systolic circuit, the A value was input with its bits ordered from most to least significant, while the B value was input with its bits in the opposite order. The transformed circuit takes both A and B values in most to least significant bit order.

4 Constructing the Multiplier

Using the hexagonal array in Figure 6, partial products are generated such that their columns can be added to produce the final product. From here it is a simple matter augment this array to produce a full multiplier.

Figure 7 shows this augmented circuit. The cell at the top of each column takes a_i and b_j as inputs and produces a partial product. This partial product is passed to the next cell in the column, while the inputs are passed unchanged along the diagonals.

As with the cells at the top of the columns, the other cells take inputs a_i and b_j and produce a partial product $a_i b_j$. Again, the inputs are passed unchanged along the diagonals. In these cells, however, adder logic is used to produce the sums of these partial products.

Figure 8 shows the logic implementation of the three types of cells used in the multiplier array. When a single sum input is available, a half adder circuit is used to produce a sum and carry output. The carry output is passed diagonally downward in parallel with the b_j output to the next column. When two sum inputs are available, a full adder circuit is used. As with the half adder the sum output is passed downward and the carry is sent downward to the next column.

It should be pointed out that the circuits in Figure 8 are logical representations. In the most general case, a programmable array would implement these cells as 4-input, 4-output look-up tables. A less general, but simpler cell could be based on the ANDing full adder. Setting the S_{in} and C_{in} inputs to zero

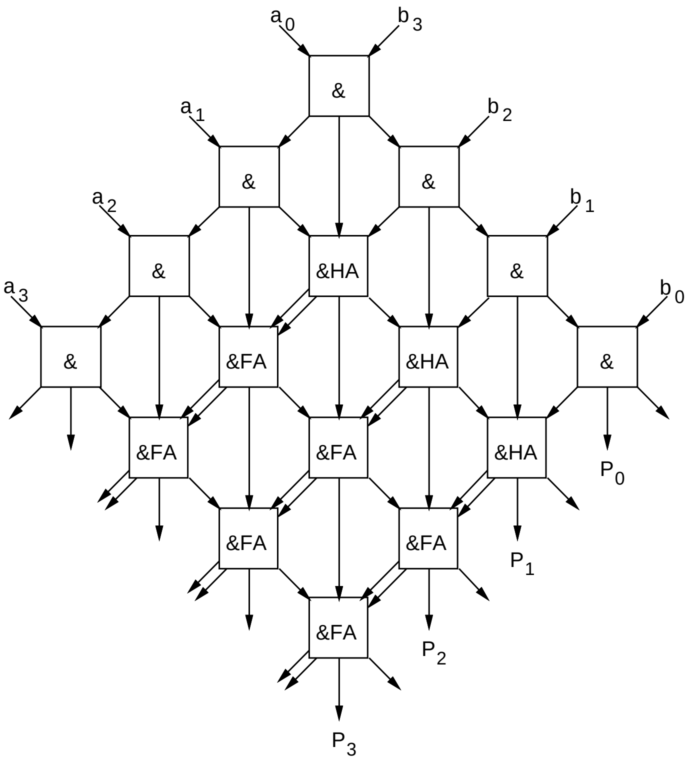

Figure 7: The multiplier (least significant bits).

would produce the functionality of the ANDing (&) cell, while setting only the C_{in} input to zero would produce the functionality of the ANDing half adder (&HA) cell.

Although all product terms have been produced and used in a calculation, there is still some work to be done. Only the lower N bits of the product P have been produced. The remaining sum and carry outputs must be manipulated further to produce the N high-order bits of the product. This final portion of the calculation counts for only a small portion of the hardware, but can represent

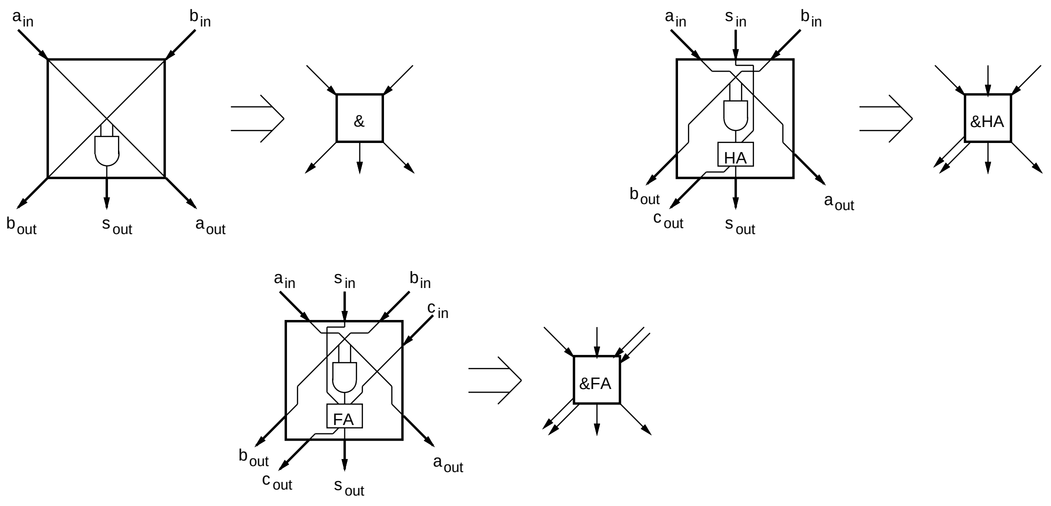

Figure 8: The multiplier cells.

up to half of the circuit delay. This is because of the data dependencies necessary to produce these outputs.

Many traditional high-performance multipliers opt to use a special fast adder to produce these final product bits. While this is an option, the approach here will use standard adders in the established cellular framework.

Figure 9 shows the multiplier with the "tail" of adder cells cascading from the lower left corner of the array. The cells connecting this tail with the rectangular portion of the array are used exclusively for routing. These routing cells also serve to provide appropriate delays to the signals in this portion of the array.

This final portion of the multiplier uses one standard half adder and $(N-2)$

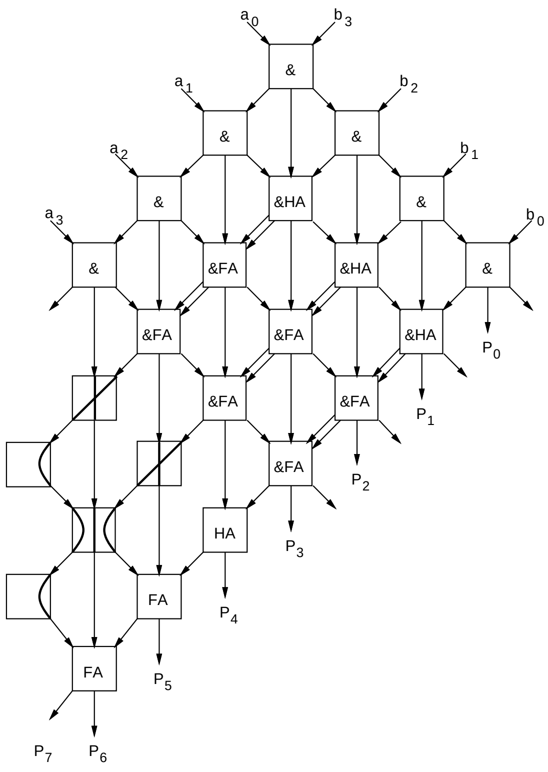

Figure 9: The multiplier (bits 0–$2N$).

standard full adders, for an increased pipeline depth of $(N-1)$. There are also approximately $N^2/4$ cells used for routing.

5 Synchronization

So far, the issue of synchronization has been ignored. Simple synchronous operation was possible for the earlier systolic ANDing circuit. When the downward propagation of sums is included, however, a more complicated scheme becomes necessary.

In the final multiplier circuit, cells have four inputs from the previous two rows of the array. The combination of these signals from rows $(N-1)$ and $(N-2)$ are the source of the timing difficulties. In this arrangement, the downward sums propagate at a rate of approximately twice that of the other signals.

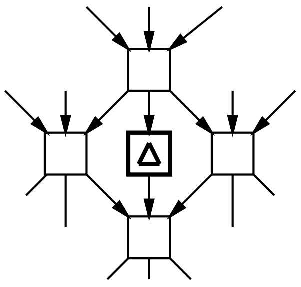

Figure 10: Synchronization by delay insertion.

One solution, as shown in Figure 10, is the addition of a delay stage between successive downward cells. This restores synchronous operation by insuring that all signals into a cell at row N arrive from cells in row $(N-1)$. While this permits a simple single clock to drive all cells in the array, the number of cells in the array is doubled. This may be acceptable, however, since these delay cells will be much simpler that the other cells in the array.

A second solution, as shown in Figure 11, involves switching to a two-phased clocking strategy. Here, cells on "even" rows are clocked, making data available to rows $(N+1)$ on the diagonals and $(N+2)$ downward. Next, "odd" rows are

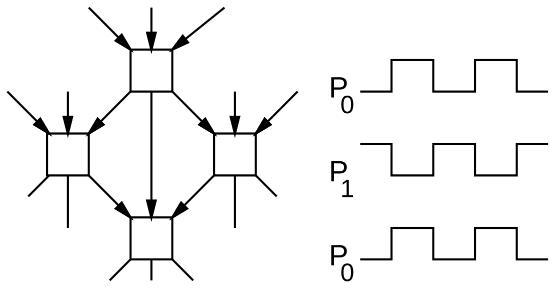

Figure 11: Synchronization via two-phased clocking.

clocked, using previously latched data from rows $(N-1)$ on the diagonals and $(N+2)$ upwards. This keeps the downward flow of data synchronized with the diagonals.

For circuits using latched outputs, this scheme may be implemented using a two-phase non-overlapping clock. In the first phase, ϕ_1, is used to clock “even” rows, and the second phase, ϕ_2, is used to clock the remaining “odd” rows.

For outputs using edge-triggered devices, devices which latch on the rising edge may be used for “even” rows. Devices which latch on the falling edge may be used for the “odd” rows. A similar effect can be achieved by inverting the clock signal on alternate rows.

While the edge-triggered solution uses a simpler clocking strategy, the edge-triggered flip-flops used to construct the pipeline are somewhat more complex than the latches necessary for the two-phased clocking strategy.

6 Performance Characteristics

Both of these synchronization strategies influence the way that performance characteristics are measured. In the two phased scheme, data moves through the array at two rows per clock cycle. While an array may have a pipeline depth of N, the pipeline latency will be only $N/2$ clock cycles. It should be noted, however, that the frequency of the one-phased clock will be approximately twice that of the two-phased clock. Both clocking strategies provide the same throughput.

The depth of the pipeline is easily calculated. The main array has a depth of $(N + (N - 1))$. The additional sums in the "tail" increases the pipeline depth by an additional $(N - 1)$. This results in a total pipeline depth of $3(N - 1) + 1$.

The array may be extended to a rectangle, with the extra cells being used as delay elements. These delay elements provide the appropriate skew to the inputs and output. Operands can be written and products read out in parallel in a single cycle.

The total size of the rectangular array can be easily calculated. Given an array depth of approximately $3(N - 1)$, and a width of $2N + 1$, the size of the array is given by $3N^2 - 9N + 4$, or less than $3N^2$. Note that this value is the product of the height and the width, divided by two. The division by two is necessary because of the hexagonal array. There are $(2N + 1)$ cells in every *two* rows, not each row.

Figure 12 shows the cell usage of the multiplier. Note that approximately

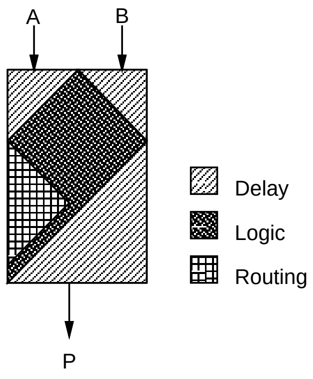

Figure 12: The multiplier cell usage.

half of the cells are used as delays to de-skew the data. Approximately N^2 cells are used for the actual multiplier array. The remaining cells are used strictly to route data.

7 Conclusions

Computation using programmable logic arrays promises to have the flexibility of a software programmable system with the performance of a custom hardware solution. To fully realize the potential of these systems, powerful high-level functions must be constructed. In this paper, a circuit for one of the more important computational functions, multiplication, has been developed. This multiplier is based on a hexagonal array of programmable cells with four inputs and four outputs. Only local nearest neighbor interconnections are used. This results in an regularly structured, high-speed, pipelined multiplier circuit.

Perhaps more significantly, the use of a hexagonal array with local interconnections has been shown to be a suitable architecture for this type of calculation. This is in contrast to the rectangular arrays with dedicated routing that are currently popular.

References

[AS93] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[GG93a] Steven A. Guccione and Mario J. Gonzalez. A data-parallel programming model for reconfigurable architectures. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 79–87, 1993.

[GG93b] Steven A. Guccione and Mario J. Gonzalez. A neural network implementation using reconfigurable architectures. In *Third International Workshop on Field Programmable Logic and Applications*, 1993.

[GG94] Steven A. Guccione and Mario J. Gonzalez. Fractal generation on a reconfigurable architecture (submitted for publication). In *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[GHK$^+$90] Maya Gokhale, William Holmes, Andrew Kosper, Dick Kunze, Dan Lopresti, Sara Lucas, Ronald Minnich, and Peter Olsen. SPLASH:

A reconfigurable linear logic array. In *International Conference on Parallel Processing*, pages I–526–I–532, 1990.

[Kea89] T. A. Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation.* PhD thesis, University of Edinburgh, Department of Computer Science, January 1989.

[MT90] Gin-Kou Ma and Fred J. Taylor. Multiplier policies for digital signal processing. *IEEE ASSP Magazine*, 7(1):6–20, January 1990.

[NOI93] Chetana Nagendra, Robert Michael Owens, and Mary Jane Irwin. Digit systolic algorithms for fine-grain architectures. In *International Conference on Application Specific Array Processors*, pages 466–477, 1993.

[Swa73] Earl E. Swartzlander, Jr. The quasi-serial multiplier. *IEEE Transactions on Computers*, C–22(4):317–321, April 1973.

[Swa90] Earl E. Swartzlander, Jr., editor. *Computer Arithmetic*, volume 1. IEEE Computer Society Press, Los Alamitos, California, 1990.