Pipelined Multipliers for Reconfigurable Hardware

Mitchell J. Myjak and José G. Delgado-Frias
School of Electrical Engineering and Computer Science, Washington State University
Pullman, WA 99164-2752 USA
{mmyjak, jdelgado}@eecs.wsu.edu

Abstract

Reconfigurable devices used in digital signal processing applications must handle large amounts of data in vector form. Most signal processing algorithms use multiplication extensively; thus, the hardware must support this operation to achieve high performance. However, mapping a multiplier on traditional fine-grain devices produces a complex structure whose performance is limited by the routing overhead. In this paper, we present a novel pipelined multiplier structure suitable for medium-grain and coarse-grain reconfigurable cell arrays. We first implement an unsigned n-bit multiplier using m-bit cells. Then, we show how the same structure can work with two's-complement data with small changes to the configuration. The structure requires $\lceil n/m \rceil^2$ cells, but can execute vector operations in a pipelined fashion. We also discuss the benefits of using a hierarchical design for large multipliers.

1. Introduction

Reconfigurable hardware has become an attractive option for implementing digital signal processing (DSP), especially in applications that require both high performance and flexibility. The performance of reconfigurable devices typically falls between custom integrated circuits and digital signal processors, while the flexibility may even surpass both alternatives. In addition, reconfigurable hardware incurs a low development cost and can be adapted if the needs of the application change, even after deployment [1].

DSP algorithms place great demands on the processing power of any hardware implementation, due to the large amount of binary arithmetic involved. For example, the basic operation of a finite impulse-response (FIR) filter contains several multiplications and additions:

$$y[n] = b_0x[n] + b_1x[n-1] + \ldots + b_kx[n-k] \quad (1)$$

As another example, the essential component of the Fast Fourier Transform (FFT) also requires these two operations, although the inputs and outputs are complex numbers in this case:

$$Y_0 = X_0 + X_1,$$
$$Y_1 = (X_0 - X_1) \times W. \quad (2)$$

Most DSP algorithms repeat basic operations such as these for every sample in the data set. To achieve maximum performance, the hardware can pipeline the multiplication and addition operations so that multiple samples can be processed simultaneously. This technique dramatically reduces the execution time of DSP algorithms, but incurs a penalty of additional overhead.

Consider the problem of using reconfigurable hardware to perform binary arithmetic on large data sets. In general, reconfigurable devices contain an array of programmable cells and interconnection structures [2]. Traditional fine-grain architectures such as the field-programmable gate array (FPGA) place little functionality in the cells. Implementing an adder on a fine-grain device presents no major problems, as the cells can easily generate the carry and sum bits required for this operation. However, implementing a multiplier is a significant challenge, due to the routing delays associated with inter-cell communication in fine-grain devices. One way around this problem is to include dedicated multiplication hardware in the architecture [3, 4]. Another approach is to extend the capabilities of each cell to work with *m*-bit data words instead of single bits [5]. In fact, these medium-grain and coarse-grain reconfigurable devices show great promise for DSP applications [6, 7].

With this approach, the problem then becomes the following: given an array of *m*-bit cells, design a structure to multiply two unsigned *n*-bit integers *A* and *B*:

$$Y_{2n-1:0} = (A_{n-1:0} \times B_{n-1:0}). \quad (3)$$

Note that the output *Y* contains 2*n* bits. As an initial approach, consider the familiar carry-save multiplier [8]. Figure 1 illustrates this structure for *n*=20 and *m*=4. The

multiplier contains a rectangular array of cells with $\lceil n/m \rceil$ cells in the horizontal direction and $\lceil n/m \rceil + 1$ cells in the vertical direction. A and B are divided into m-bit portions and broadcast across the columns and rows of the array.

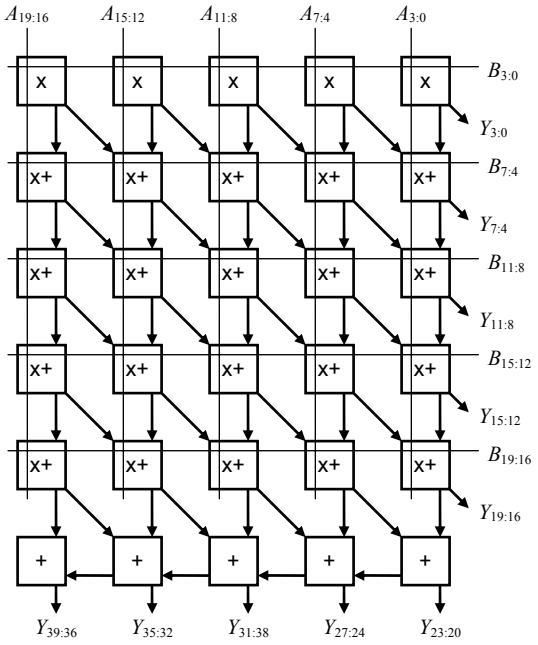

Figure 1. Carry-save multiplier (n=20, m=4)

Unlike a typical carry-save multiplier, each cell works with m-bit inputs instead of 1-bit inputs. Cells on the top row multiply two m-bit portions of A and B, passing the upper and lower portions of result to other cells or the Y output. Cells in the middle rows perform the same multiplication and then add up to two m-bit terms to the result. Finally, cells on the bottom row add up to three m-bit terms together. As a function of n and m, the carry-save multiplier requires $\lceil n/m \rceil^2 + \lceil n/m \rceil$ cells, while the critical path is $2\lceil n/m \rceil$ cells long.

In the remainder of this paper, we describe a novel improvement to the carry-save multiplier that reduces the total number of cells required. As described in Section 2, we modify the interconnection structure so that every cell carries the same workload. The resulting structure can be pipelined easily for high throughput. In Section 3, we demonstrate that the same structure can be used to perform two's-complement multiplication with slight changes to the operations performed by each cell. Section 4 briefly explores a hierarchical architecture in which each cell uses a matrix of smaller elements to implement the necessary operations. Finally, Section 5 gives some concluding remarks.

2. Unsigned Multiplier

The most complex operation performed by any cell in the carry-save multiplier is the m-bit multiply-accumulate (MAC) function

$$y_{2m-1:0} = (a_{m-1:0} \times b_{m-1:0}) + c_{m-1:0} + d_{m-1:0}. \quad (4)$$

Here a and b represent the two m-bit portions of A and B, and c and d denote the two terms added to the result. Figure 2 illustrates the inputs and outputs of a cell that computes the MAC operation.

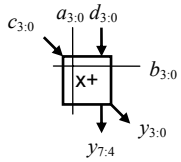

Figure 2. Inputs and outputs of MAC cell (m=4)

As we proposed in [9], one can reduce the size of the multiplier by ensuring that all cells perform the same operation. Such a reduction is possible because reconfigurable devices typically contain an array of identical cells. Observe in Figure 1 that the bottom row of cells do not perform multiplication, and the cells in the left column add one term instead of the usual two. Hence, we can eliminate the bottom row of cells and rearrange the interconnection scheme so that the cells in the left column "take up the slack". Figure 3 depicts the resulting structure for the 20-bit case. This improved design has a size of $\lceil n/m \rceil^2$ cells and a critical path $2\lceil n/m \rceil - 1$ cells long.

Notice that two extra n-bit terms C and D can be incorporated into the top row of cells. This enhancement creates a powerful n-bit MAC unit that evaluates the expression

$$Y_{2n-1:0} = (A_{n-1:0} \times B_{n-1:0}) + C_{n-1:0} + D_{n-1:0}. \quad (5)$$

The primary advantage of using arrays of cells to perform multiplication is that the DSP algorithm can exploit the benefits of pipelining. Suppose that we superpipeline the structure in Figure 3 so that each cell occupies one pipeline stage. The clock cycle time thus includes the time to evaluate the m-bit MAC function as well as the time to transfer the result to adjacent cells. Figure 4 labels each cell in the improved MAC unit with the clock cycle at which it calculates the intermediate result. We can then insert the appropriate number of pipeline registers into the module, as depicted by slashes in the figure.

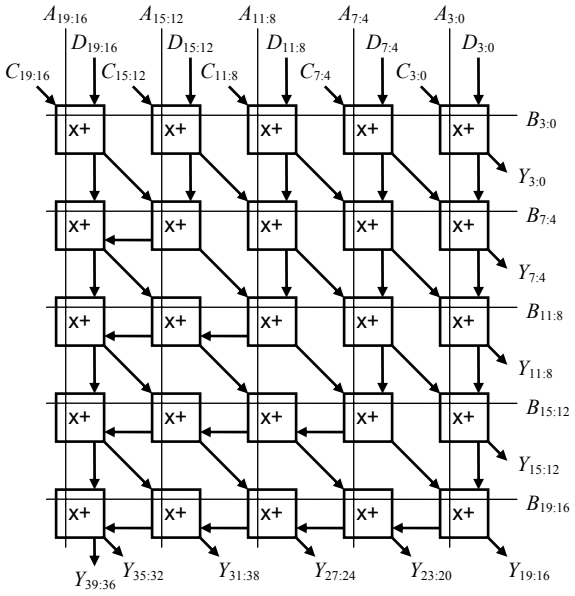

Figure 3. Improved MAC unit (*n*=20, *m*=4)

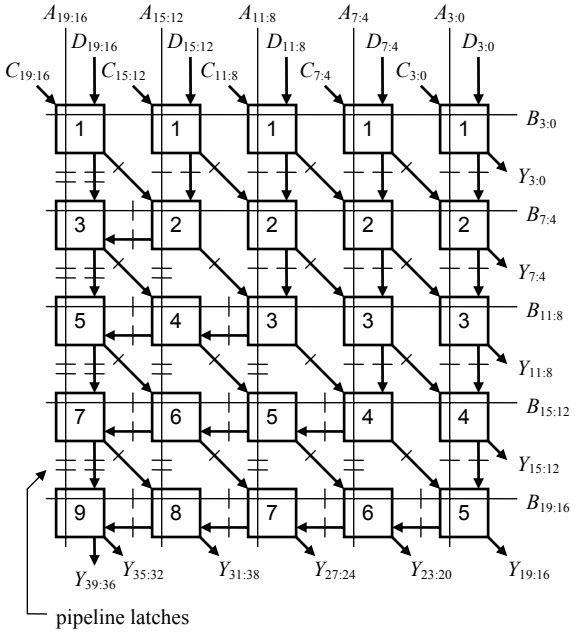

Figure 4. Superpipelined MAC unit

The latency of this superpipelined design equals the length of the critical path, or $2\lceil n/m \rceil - 1$ cycles. However, the structure can initiate one operation per clock cycle, making the resulting throughput very high. Notice that the MAC unit generates the *Y* output in a staggered fashion: the least significant *m* bits in the first cycle, the next *m* bits in the second cycle, and so forth. The *B* input should also arrive in this staggered fashion.

The pipelining scheme in Figure 4 has one disadvantage: the *m*-bit portions of the *B* input must be broadcast across several columns of cells during the same clock cycle. Depending on the architecture of the reconfigurable device, such an operation may not be feasible. One way to eliminate the broadcast is to pipeline all the internal lines of the MAC unit. As shown in Figure 5 for the 20-bit case, this change increases the total latency of the module to $3\lceil n/m \rceil - 2$. This additional delay should be negligible for most DSP algorithms, since the multiplier still initiates one operation per clock cycle. For example, to multiply two 20-bit vectors of 1000 elements, the former design requires 1008 cycles, whereas the latter design requires 1012 cycles. If the clock cycle time can be reduced due to the absence of broadcast operations, the MAC unit with pipelined internal lines actually achieves higher performance.

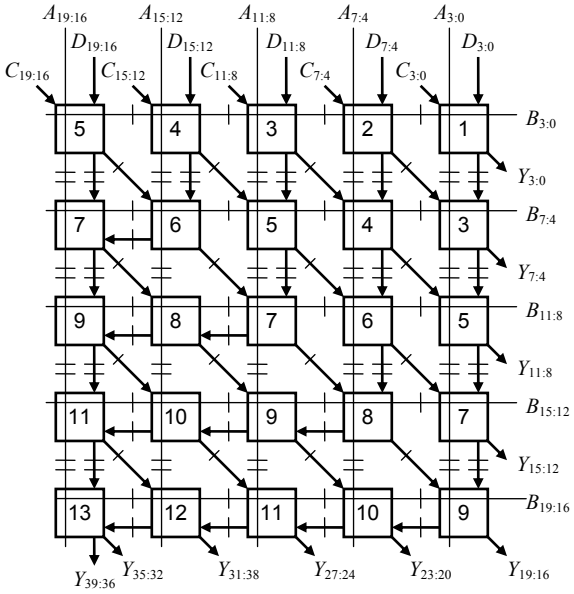

Figure 5. Superpipelined MAC unit with pipelined internal lines

3. Two's-Complement Multiplier

As a rule, DSP algorithms work with both positive and negative numbers, so it is reasonable to expect that applications may require two's-complement multiplication. In fact, the same multiplier structure can be used to perform this operation, except that some cells evaluate slightly different functions. Before we discuss these modifications, recall from (4) that each cell in the unsigned MAC unit evaluates the *m*-bit MAC function

$$y_{2m-1:0} = (a_{m-1:0} \times b_{m-1:0}) + c_{m-1:0} + d_{m-1:0}. \quad (4)$$

Figure 6 illustrates how these m-bit terms are defined for various cells in the design. For consistency, the c input to the cell always appears to the left of the d input.

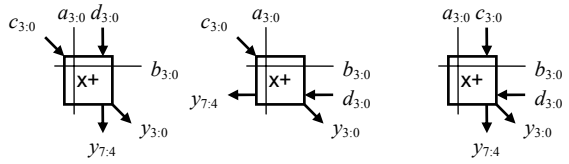

Figure 6. Cells in improved MAC unit

Now consider a two's-complement MAC unit that handles n-bit inputs in m-bit portions. From the properties of two's-complement numbers, the most significant m-bit portion has two's-complement format, but the remaining m-bit portions have unsigned format. Hence, if we modify the unsigned MAC unit to perform two's-complement arithmetic, many of the cells will still operate on unsigned inputs. Figure 7 depicts the two's-complement multiplier for n=20 and m=4. Solid lines denote unsigned data; dashed lines denote two's-complement data.

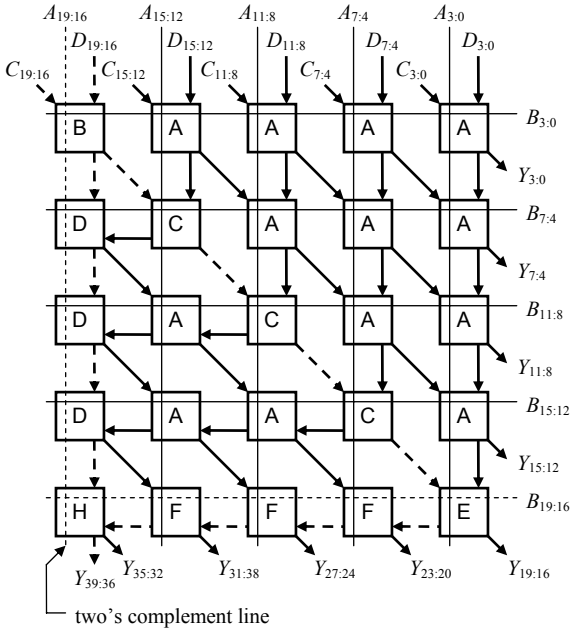

Figure 7. Two's-complement MAC unit (m=20, n=4)

Observe that some of the cells generate two's-complement outputs, whereas other cells do not. In fact, the two's-complement MAC unit contains seven types of cells, labeled A through H in the figure (G is missing for technical reasons). The A cells simply evaluate the unsigned MAC function in (4). However, the B cell must multiply the two's-complement portion of A with an unsigned portion of B. The cell also adds two's-complement portions of C and D to the result.

In order to represent the entire range of valid outputs, the B cell must generate a $2m$-bit output y whose upper m bits and lower m bits are both two's-complement numbers. This data format is unusual, but is the best choice for representing the result. One can think of the cell as generating two m-bit outputs satisfying the expression

$$2^m y_{2m-1:m} + y_{m-1:0} = (a_{m-1:0} \times b_{m-1:0}) + c_{m-1:0} + d_{m-1:0}. \qquad (6)$$

where $a_{m-1:0}$, $c_{m-1:0}$, $d_{m-1:0}$, $y_{2m-1:m}$, and $y_{m-1:0}$ are in two's-complement format. Table 1 lists several example 4-bit calculations for the B cell. Recall that a 4-bit two's-complement numbers ranges from –8 to 7, whereas a 4-bit unsigned number ranges from 0 to 15.

Table 1: Example calculations of the B cell

$a_{3:0}$	$b_{3:0}$	$c_{3:0}$	$d_{3:0}$	$y_{7:4}$	$y_{3:0}$	$y_{7:0} = 16y_{7:4} + y_{3:0}$
5	5	5	–5	2	–7	25
5	10	5	5	4	–4	60
–5	5	5	–5	–2	7	–25
–5	10	–5	–5	–4	4	–60
7	15	7	7	7	7	119
–8	15	–8	–8	–8	–8	–136

A similar analysis can be performed for the remaining cells used in the multiplier. For example, the C cells generate an unsigned output $y_{2m-1:m}$ and a two's-complement output $y_{m-1:0}$. With the data formats shown in Figure 7, the n-bit multiplier can generate a two's-complement output Y without additional hardware. Table 2 lists the input and output format of each type of cell (including the G cell used later). A "+" sign denotes unsigned format, and a "–" sign denotes two's-complement format.

Table 2: Data format requirements for each cell

Type	$a_{m-1:0}$	$b_{m-1:0}$	$c_{m-1:0}$	$d_{m-1:0}$	$y_{2m-1:m}$	$y_{m-1:0}$
A	+	+	+	+	+	+
B	–	+	–	–	–	–
C	+	+	+	–	+	–
D	–	+	–	+	–	+
E	+	–	–	+	–	+
F	+	–	+	–	–	+
G	+	+	–	+	+	–
H	–	–	–	–	–	+

4. Hierarchical Multiplier

The last two sections have demonstrated that n-bit MAC units in general require seven types of cells. Each cell performs the MAC function on m-bit inputs, but different cells use different data formats. A natural question is how each cell can implement the required m-bit operations. For m=1, a simple combinational circuit suffices, but for larger m, the most practical solution may involve some kind of arithmetic unit.

For reconfigurable devices, consider the following alternative: to implement the m-bit operations required by each cell, use an $m \times m$ array of 1-bit cells. In other words, the proposed architecture contains a two-level hierarchy of cells and "elements", where cells work with m-bit words and elements work with single bits. The next question is how the $m \times m$ array of elements can implement all the functionality required by m-bit cells. For type A cells, the solution is simple: use the unsigned multiplier structure presented in Section 2. As shown in Figure 8 for m=4, each of the elements works with data in unsigned form. Hence, one can classify the elements as type A as well.

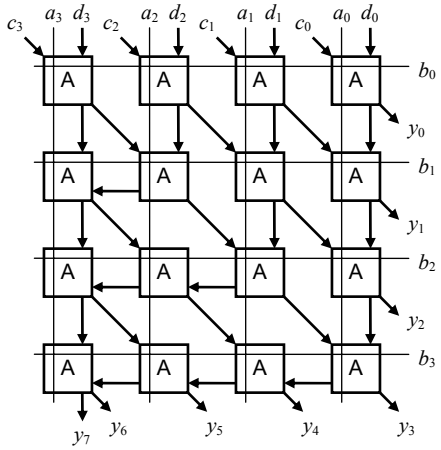

Figure 8. Type A cell (m=4)

Each element computes the 1-bit MAC function

$$\psi_{1:0} = (\alpha \wedge \beta) + \gamma + \delta, \tag{7}$$

where α, β, γ, and δ denote the inputs to the element, and ψ signifies the 2-bit output. Note that multiplication reduces to the logical AND operation, denoted by $\wedge$, in the 1-bit case. Each bit of the output ψ can be expressed in terms of the combinational logic functions

$$\begin{aligned} \psi_1 &= \mathrm{MAJ}(\alpha \wedge \beta, \gamma, \delta) \\ \psi_0 &= \mathrm{XOR}(\alpha \wedge \beta, \gamma, \delta), \end{aligned} \tag{8}$$

where

$$\begin{aligned} \mathrm{MAJ}(P, Q, R) &= (P \wedge Q) \vee (P \wedge R) \vee (Q \wedge R) \\ \mathrm{XOR}(P, Q, R) &= P \oplus Q \oplus R. \end{aligned} \tag{9}$$

As discussed in the last section, two's-complement MAC units require additional types of cells. Type B cells, for example, assume that a, c, and d have two's-complement format, and that b has unsigned format. Using an $m \times m$ array of elements to implement a type B cell produces the result in Figure 9.

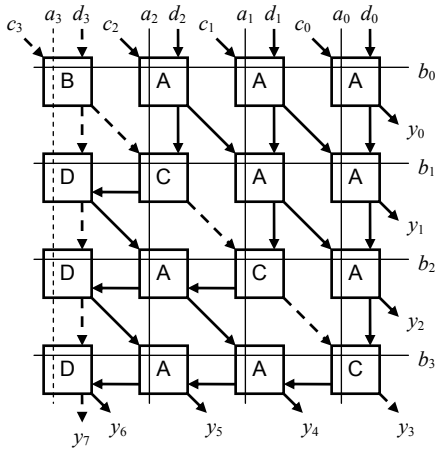

Figure 9. Type B cell (m=4)

Knowing the data format for each input to the cell, one can determine the format of every internal line using the information in Table 2. The procedure closely parallels the analysis for the two's-complement multiplier in Figure 7, except that the signal names are Greek symbols instead of lowercase letters. The implementation of the type B cell requires elements of types A, B, and C. Note that both the upper and lower portions of the y output have two's-complement format, as shown in Figure 7.

Continuing on, cells of types C and D have straightforward implementations (Figures 10-11). Type E cells require five types of elements, including type G (Figure 12). Type F cells are similar (Figure 13). Finally, type H cells have the same formatting assignments as the two's-complement multiplier (Figure 14). This property holds because all the inputs and outputs of a type H cell have two's-complement format.

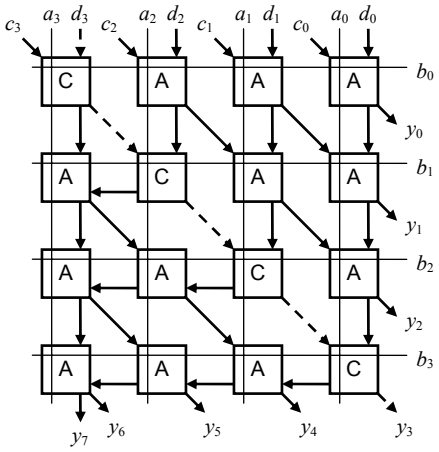

Figure 10. Type C cell (*m*=4)

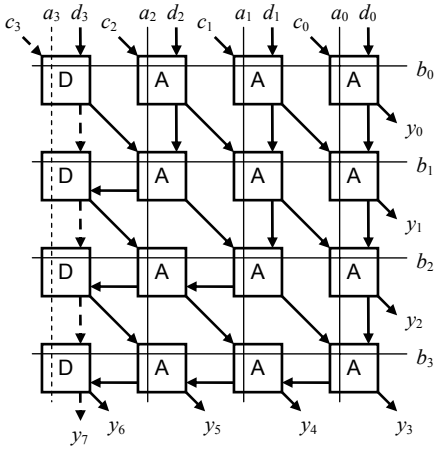

Figure 11. Type D cell (*m*=4)

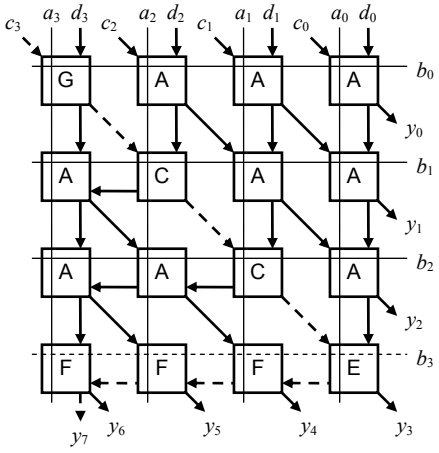

Figure 12. Type E cell (*m*=4)

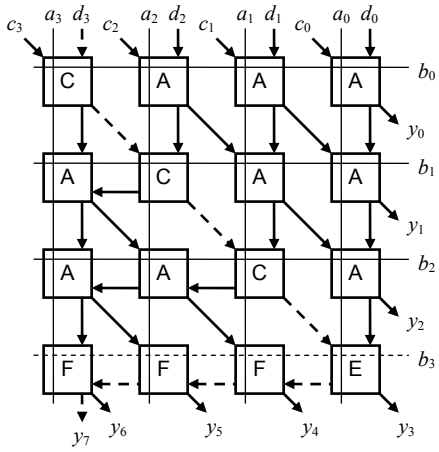

Figure 13. Type F cell (*m*=4)

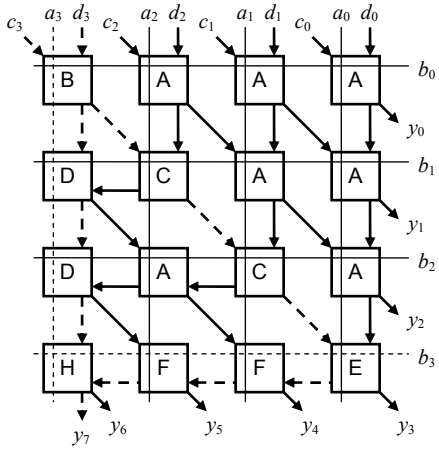

Figure 14. Type H cell (*m*=4)

Now consider the MAC function computed by type B elements. From Table 2, the α, γ, δ, ψ_1, and ψ_0 signals of type B elements all have two's-complement format. For each of these signals, logic 0 denotes 0 and logic 1 denotes –1. Hence, type B elements compute the expression

$$-2\psi_1 - \psi_0 = (-\alpha \times \beta) - \gamma - \delta, \tag{10}$$

which simplifies to

$$2\psi_1 + \psi_0 = (\alpha \wedge \beta) + \gamma + \delta. \tag{11}$$

Since (11) and (7) are equivalent, elements of types A and B implement the same combinational logic expressions.

Performing a similar analysis on the remaining types of cells reveals that only four distinct types of elements are required. In fact, each element implements the same

expression for ψ_0; the only difference is the expression used to compute ψ_1. Table 3 lists the functions corresponding to each type of element. (Note that $\neg$ denotes the logical complement.) A reconfigurable architecture could exploit these similarities to implement all necessary operations efficiently.

Table 3: Reduction of element types

Type	ψ_1	ψ_0	Same as
A	MAJ($\alpha \wedge \beta$, γ, δ)	XOR($\alpha \wedge \beta$, γ, δ)	A
B	MAJ($\alpha \wedge \beta$, γ, δ)	XOR($\alpha \wedge \beta$, γ, δ)	A
C	MAJ($\alpha \wedge \beta$, γ, $\neg\delta$)	XOR($\alpha \wedge \beta$, γ, δ)	C
D	MAJ($\alpha \wedge \beta$, γ, $\neg\delta$)	XOR($\alpha \wedge \beta$, γ, δ)	C
E	MAJ($\alpha \wedge \beta$, γ, $\neg\delta$)	XOR($\alpha \wedge \beta$, γ, δ)	C
F	MAJ($\alpha \wedge \beta$, $\neg\gamma$, δ)	XOR($\alpha \wedge \beta$, γ, δ)	F
G	MAJ($\alpha \wedge \beta$, $\neg\gamma$, δ)	XOR($\alpha \wedge \beta$, γ, δ)	F
H	$\neg$MAJ($\alpha \wedge \beta$, $\neg\gamma$, $\neg\delta$)	XOR($\alpha \wedge \beta$, γ, δ)	H

5. Concluding Remarks

In this paper, we have presented a novel scheme for performing *n*-bit multiply-accumulate (MAC) operations using a reconfigurable array of *m*-bit cells. Each cell computes an *m*-bit MAC function with two additive terms. The structure can be superpipelined into *m*-bit units for extremely high throughput, as required in signal processing applications. With suitable changes to the configuration of each cell, the structure can handle unsigned or two's-complement inputs. To implement the functionality required by each cell, we propose to use an $m \times m$ matrix of reconfigurable 1-bit elements. Only four types of elements are required to construct multipliers of any size.

As a final note, we have used the concepts presented in this paper to create a two-level reconfigurable architecture for digital signal processing applications [9]. The architecture contains an array of reconfigurable 4-bit cells, each of which consists of a 4×4 matrix of elements. Each element, in turn, uses a 4-input, 2-bit lookup table to evaluate arithmetic or logic functions. Cells can connect to neighboring cells in any direction. However, the matrix of elements can only assume two structures, one of which is the structure of the MAC unit. Having the capability to compute the MAC operation means that cells can perform the arithmetic functions necessary for digital signal processing.

6. Acknowledgment

M. Myjak is supported by the U.S. Department of Homeland Security Graduate Fellowship.

7. References

[1] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: a survey", in Y. Hu (ed.), *Programmable digital signal processors*, Marcel Dekker Inc., 2001.

[2] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, Jun 2002, pp. 171-210.

[3] K. Rajagopalan and P. Sutton, "A flexible multiplication unit for an FPGA logic block," in *Proc. 2001 IEEE International Symposium on Circuits and Systems*, 2001, pp. 546-549.

[4] S. Haynes and P. Cheung, "Configurable multiplier blocks for embedding in FPGAs," *Electronics Letters*, vol. 34, iss. 7, Apr 1998, pp. 638-639.

[5] R. Hartenstein, "Coarse grain reconfigurable architectures," in *Proc. 6th Asia South Pacific Design Automation Conference*, Yokohama, Japan, 2001, pp. 564-570.

[6] J. Smit et al, "Low cost and fast turnaround: reconfigurable graph-based execution units," in *Proc. 7th BELSIGN Workshop*, Enschede, Netherlands, 1998.

[7] P. Heysters and G. Smit, "Mapping of DSP algorithms on the MONTIUM architecture," in *Proc. International Parallel and Distributed Processing Symposium*, Apr 2003, pp. 180-185.

[8] J. Rabaey et al, *Digital Integrated Circuits: A Design Perspective*, 2nd ed., Upper Saddle River, NJ: Pearson Education, Inc., 2003, pp. 591-592.

[9] M. Myjak and J. Delgado-Frias, "A two-level reconfigurable architecture for digital signal processing," in *Proc. 2003 International Conference on VLSI*, Las Vegas, NV, Jun 2003, pp. 21-27.