Configurable Arithmetic Arrays with Data-Driven Control

Behrooz Parhami
Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA
E-mail: parhami@ece.ucsb.edu

Abstract

Configurable digital systems offer flexibility that leads to cost reduction, performance tuning, and fault tolerance. While one can use FPGA-based methods to synthesize configurable arithmetic structures, this approach results in limited functionality and/or excessive complexity, given that off-the-shelf FPGAs are endowed with only bit-level computational capabilities. Using serial arithmetic cells can increase the ratio of resources devoted to computation (arithmetic, logic) versus communication (wiring), thereby improving per-chip performance or cost-effectiveness. When combined with pipelining, even the absolute latency can be improved. We argue that a controlled-precision digit-serial additive multiplier constitutes an ideal building block for configurable arithmetic arrays and detail our solutions for control of functionality, selection of precision, and reduction of power dissipation within such an array.

Keywords: Bit-serial arithmetic, digit-pipelined arithmetic, FPGA-like arrays, DSP hardware, on-line algorithm, pipelining, programmable logic, redundant number system.

1. Introduction

A great deal of interest exists in the design of configurable digital systems [Vill98]. Configurability is desirable for:

- Cost reduction via mass production and reduced design time
- Performance tuning via customization or dynamic adaptation
- Fault tolerance via detection and avoidance of bad elements

Most implementations of configurable digital systems rely on off-the-shelf FPGA technology in view of ready availability of such hardware components along with associated design tools [Mang97], [Vill98], [Rath99]. However, the bit-level computational capabilities of FPGAs makes the synthesis of arithmetic-intensive functions quite complex. Even though some of the performance lost in fitting the computation to a general-purpose structure is regained by using clever algorithms or customized word widths [Klot96], [Luo00], [Shir95], [Tiou99], performance does remain far below that of custom hardware.

Case studies indicate that even small, short-word digital filters, for example, use up hundreds of logic blocks in an FPGA [Do98]; such designs utilize neither the full flexible functionality of the logic blocks nor the extensive interconnect resources that link these blocks. Most often, the FPGA resources are configured into regular arrays of full-adders. The flip side of this coin is that systems with "chunky" function units (ALUs, multipliers, etc.) are more rigid and it is hard to decide what functionalities to include in the chunky units [Mang97]. At the extreme, logic blocks might be replaced by "processors"; but then, even more speed is sacrificed to gain the added flexibility.

In this paper, we propose and evaluate particular types of cells that can be used in FPGA-like structures to facilitate the synthesis of arithmetic-intensive computations. Our thesis is that such structures are limited by their I/O pins and intercell connectivity. The cells themselves can be made quite complex, even with present-day technology (transistors are cheap; pins and wires are expensive). Such complex cells need not lead to performance degradation or excessive power dissipation when only very simple functionalities are required, provided that the unused sections of a cell do not form part of the critical signal path and are not clocked in idle mode.

Our scheme involves digit-serial arithmetic in radix 2, using either two's-complement encoding or redundant representation with the digit set {–1, 0, 1}, or in radix 4. Extension to higher radices is possible but not treated here. Digit-serial arithmetic is slower when a single arithmetic operation is considered, but can become competitive, or even offer speed advantages, when:

- Cascaded operations are pipelined with some overlap
- Simplicity of digit-level operation allows faster clocking
- Savings in wiring and pins lead to more compact circuits
- Two or more operations can be merged (e.g., $vw + xy$)

Pipelining and overlap are particularly effective with redundant representation and MSD-first processing, to the extent that significant speedup can result in computations involving cascaded operations [Erce84], [Erce88].

0-7803-6514-3/00/$10.00©2000 IEEE

2. Configurable Additive Multipliers

An additive multiplier, computing $wx + y$, is an important building block for general-purpose and custom hardware aimed at signal processing applications. The multiply-add (also known as multiply-accumulate, or inner-product step) operation performed by such a unit can be used in synthesizing larger multipliers as well as circuits for computations such as FIR filtering, matrix multiplication, and discrete Fourier transform. It is thus natural to consider an additive multiply module (AMM) as the basic cell of a configurable arithmetic array.

A radix-2 bit-serial AMM needs only four data I/O terminals, thus utilizing interconnects quite efficiently. Because we will also consider the use of binray signed-digit (BSD) arithmetic with 2-bit digit encodings and wish to compare the binary/BSD results, we base a second binary design on an additive double-multiply module (ADMM) that yields the result $s = uv + wx + y$. Figure 1 shows the high-level structure of a binary ADMM. Such an ADMM has roughly the same number of I/O connections and storage elements (flip-flops) as a radix-2 AMM based on BSD numbers or a radix-4 AMM.

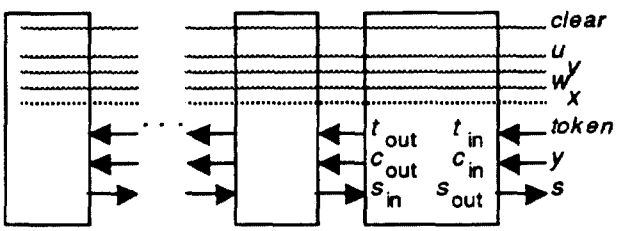

Fig. 1. Structure of an LSB-first bit-serial additive double-multiply module, computing $s = uv + wx + y$.

The principles of operation and detailed design of the ADMM shown in Fig. 1 can be found elsewhere [Hayn96]. Briefly, the five inputs enter bit-serially, LSB first. The four multiplicative inputs, namely u, v, w, and x, are broadcast to all cells. A token is passed from right to left on successive clock ticks, marking the cell that must store the incoming bits of the multiplicative inputs. After the ith clock tick, i bits of these operands are stored in the cells. The accumulated partial product, kept in double-carry-save form (set of three binary numbers) is combined with the four new terms resulting from the incoming bits using a set of (7; 3)-counters, one in each cell.

We designate the preceding module type as ADMM2, where the 2 at the end conveys the number representation radix. An AMM2 module can be similarly defined. In fact, the design of such a module that computes $s = wx + y$ is identical to the bit-serial multiplier in [Ienn94], with the additive input y, instead of 0, connected to c_{in}.

We also consider two other module types that are of comparable pin and storage complexities to ADMM2 when operands have the same number of digits: these are AMM2±, using BSD input in MSD-first order, and AMM4, based on radix-4 input. AMM4 is very similar to ADMM2, with the four new terms added to the residual formed differently. The design of AMM2± is substantially the same as an on-line BSD multiplier ([Parh00], p. 424), with one digit of the additive operand accommodated in each cycle in a way that adds neither gate levels nor complexity to the digit slices of the nonadditive version.

Table I. Main characteristics of our arithmetic cells

Cell type	Input format	First input	Data pins	Gate levels	Latency add; AM
AMM2	Binary	LSB	4	13	1; k
ADMM2	Binary	LSB	6	15	1; k
AMM4	Radix 4	LSD	8	15	1; $k/2$
AMM2±	BSD	MSD	8	14	2; 3

We make the ADMM design of Fig. 1 configurable by dividing it into blocks or cells, each dealing with h-bit operands. A $k \times k$ ADMM will then need a set of $\lceil k/h \rceil$ such blocks in a linear arrangement (Fig. 2). For concreteness, we will take $h = 8$ digits in our discussions, although other values are quite feasible. Ignoring the possibility of overflow, an 8-digit ADMM produces a 16-digit result. Because data is simply a stream of digits, one can use the full 16-digit result in subsequent operations or drop the lower half of the result to keep the word width at 8 digits. The same point applies to wider additive multipliers built of a number of AMM or ADMM cells.

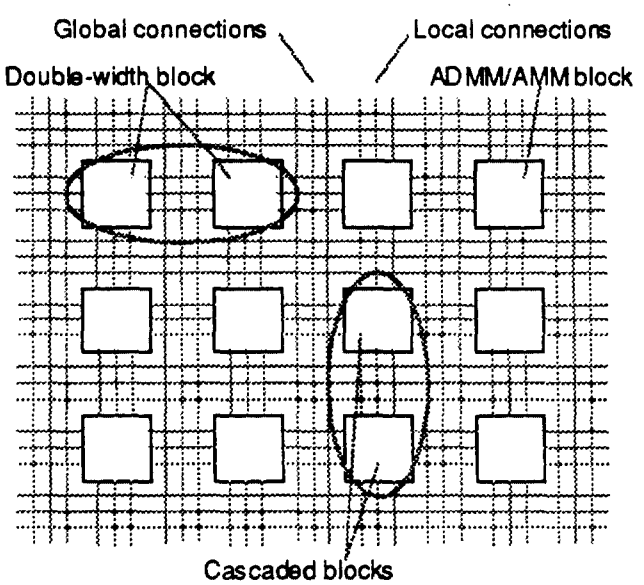

Fig. 2. An FPGA-like array of ADMMs or AMMs.

The same partitioning scheme can be applied to the three AMM designs listed in Table I. With AMM2± cells, mechanisms for conversion to/from standard binary will also be needed. Assuming the use of the (n, p) encoding for BSD digits, whereby digit values –1, 0, and 1 are represented as (1, 0), (0, 0), and (0, 1), respectively, conversion from binary to BSD becomes trivial; a bit b is encoded as $(0, b)$, with the sign bit of a 2's-complement number encoded as $(b, 0)$.

Conversion from BSD to binary, though more complicated, is also straightforward and can be done on the fly [Erce87]. Because some form of I/O or boundary cell must be provided in any case, the conversion hardware, consisting of two registers and controlled transfers between them, does not lead to undue complexity. With reference to Table I, the resulting C2±/2 cells require 3 pins, 4 gate levels, and a latency of k cycles. Merging of C2±/2 and AMM2± cells is quite feasible. The latter cells already contain more than enough storage space for to the two registers needed. Whether or not the resulting uniformity and flexibility outweighs the drawbacks of more complex, and slightly slower cells, remains to be established.

3. Data-Driven Control Scheme

The cells described in Section 2 must be programmable in two distinct but complementary ways. First, the cell function must be dynamically alterable. Possible functions include additive multiplication with prestored coefficient(s) w (and u), AM with variable coefficient(s) as input(s), and simple addition. Simple multiplication is the same as AM with the additive input y set to 0. Note that simple addition cannot be said to be equivalent to AM with one multiplicative input set to 0, because the latencies are different for the two operations (see Table I).

Many arithmetic-intensive computations of interest in signal processing can be formulated as dependence graphs with AM nodes, augmented with simple nodes that basically transfer or redirect (change the flow direction of) the input data [Kwai99]. Before showing how our additive multiply cells can be modified to handle such operations, let us point to the second aspect of programmability to accommodate varying word widths. In practice, word width changes are much less frequent than coefficient value adjustments, so this type of adaptation is best handled at configuration set-up (compile) time.

Figure 3 depicts the computation of $y_k = \sum_{i+j=k} x_i w_j$, subject to $0 \le i, j \le N - 1$. Each of the black nodes in Fig. 3 represents an AM operation. Also shown is the projection of the dependence graph onto a three-node linear array with its two data streams x and y. Practical use this linear array for performing the convolution algorithm, requires that four operations be performed [Guse92]:

- Loading the weighting coefficients into the nodes
- Delivering data to nodes doing the first operation
- Performing inner-product (additive multiply) steps
- Draining data from nodes doing the last operation

Figure 4 shows an augmented dependence graph in which the three auxiliary operations (besides inner-product steps) are also shown. At the left end of Fig. 4, the weighting coefficients are *delivered* through the input links of data stream y, flowing to the shaded nodes; the *store* operation is simply a change in the direction of data flow, from diagonal to horizontal, for the coefficients.

Besides storing the coefficients, we need to deliver data elements to nodes that operate on them first. For example, computing y_0 involves only one inner-product step that occurs within a node at the bottom of dependence graph. The white nodes in the middle of Fig. 4 forward data elements to nodes where the first computation is to occur. Likewise, some of the results (y_3 and y_4) are produced at nodes that have no direct access to the output link of data stream y. Such results must be drained out, via the hatched nodes at the right end of Fig. 4.

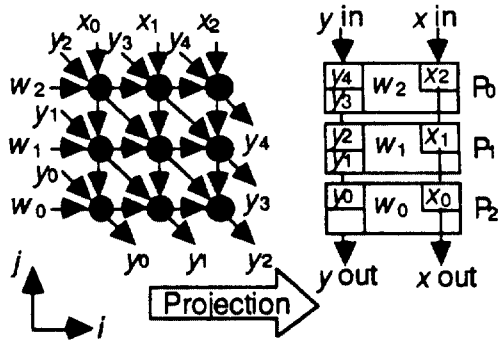

Fig. 3. Dependence graph for convolution algorithm and its projection to yield a three-node linear array.

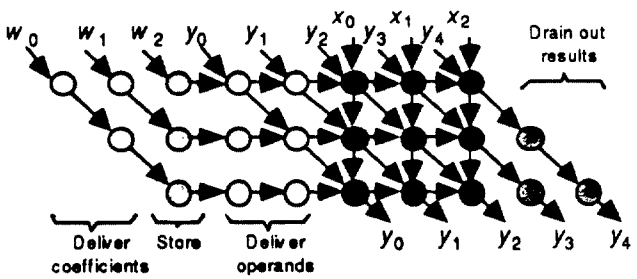

Fig. 4. Dependence graph for convolution algorithm with data loading and result drainage also included.

Note that in Fig. 4, nodes of the same type are vertically aligned, so that a two-bit tag attached to the elements of data stream x suffices to instruct the nodes as to which of the four functions (actually two functions, and one control bit, after merging compatible nodes) they should perform. The foregoing is an example of a data-driven control methodology that we have successfully applied to the design of several application-specific digital systems (see, e.g., [Kwai96], [Kwai97], [Parh99]).

For the arithmetic arrays under discussion here, we take the simple case where the cell only performs the multiply-add operation and use a single control tag bits for each input data digit. This "FD" tag signifies that an input is the first incoming digit of an operand (LSD or MSD, depending on the computation mode). Operand length is stored in the cell control logic at configuration set-up time, so timing of the last digit is automatically deduced.

Cell operation is determined by the coincidence of FD tags. Basically, the FD tag of the additive input y triggers the computation, while the FD tags of other operands specify change of value for those operands. For example, the AMM2± cell will perform the functions shown in Table II depending on the FD tags of w and x. Each time a new value of w or x is provided, it is stored in the cell; so, loading of coefficients does not require additional control. The cell continues to use the previously supplied values, if required, until new values are provided. The increment operation (line 1 of Table II) can be used for delaying y by one AM operation if we set $c = d = 0$.

Table II. Cell function for AMM2± based on FD tags

FD of w	FD of x	Function	Comment
0	0	$y + cd$	Increment y
0	1	$y + cx$	Use saved w
1	0	$y + wd$	Use saved x
1	1	$y + wx$	AM operation

4. Comparisons and Tradeoffs

As a simple example, we consider the implementation of the convolution algorithm (Fig. 3) with $N = 4$, using the cell types of Table I. We assume a word width of 16 bits, leading to $k = 8$ for the radix-4 implementation and $k = 16$ for the three radix-2 options. In all but the ADMM2 case, seven cells will be cascaded vertically (see Fig. 5). In all but the radix-4 case which requires no horizontal cascading, two cells are connected horizontally (Fig. 5). However, where cells are used to delay y, a single cell will suffice, even though the control circuit of that cell acts as if it is performing a 16-bit AM.

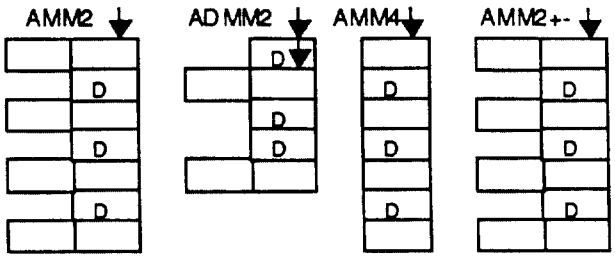

Fig. 5. Cascading structure of the four solutions.

Table III shows key parameters of the implementations. The latency in the last column is determined by multiplying the number of vertically cascaded cells by the AM/ADM latency and adding in the serial I/O time. For AMM2±, the BSD-to-binary conversion time of k cycles is also included. The external I/O pin count includes FD control tag bits. Note that in Fig. 3, equitemporal lines extend diagonally in SW-NE direction; thus, the y values must be delayed in their downward vertical movement. The cell count includes both those used for computation and the ones that merely delay results to achieve proper timing (Fig. 5).

Table III. Convolution performed using various cells

Cell type	Number of cells	C2±/2 cells	External I/O pins	Latency (clocks)
AMM2	11	–	5	128
ADMM2	7	–	5	80
AMM4	7	–	8	64
AMM2±	11	2	8	53

Examination of Table III reveals some interesting tradeoffs. Note that the clock rate is assumed to be the same for all four designs. This is not unreasonable, given the fairly small differences in gate levels within cells, per Table I, and the fact that the differences turn even less significant once latching overhead and safety margin for clock skew and gate delay variations are factored in. Cell complexities, however, are not the same and must be considered in evaluating the cost-effectiveness of these solutions.

It is worth mentioning that both k and N are quite small in our example; AMM2± will show its advantage for larger k and/or N. Despite its simpler cells, AMM2 appears to be the least competitive solution. The middle two solutions, namely those based on ADMM2 and AMM4, are quite close and must be compared in greater detail before one can be chosen over the other. Compared to solutions based on standard FPGAs, the number of cells has been reduced by some two orders of magnitude and much less wiring is required. The number of clock cycles is greater here, but the clock can be considerably faster. These latter aspects will be quantified in our future work.

5. Conclusion

Configurable arrays, of the types proposed in this paper, can be used for cost-effective implementation of arithmetic-intensive applications. The down side of configurability is the potential for reduced performance and greater power dissipation, stemming from programmable elements inserted along the critical path and greater circuit complexity. Power dissipation can be mitigated by selective disabling of cells or cell functionalities; this is easily accomplished, given our data-driven control scheme. Similarly, fault detection and tolerance capabilities can be readily introduced [Kwai97]. Other avenues for future research include detailed comparison of our approach with alternative methods, such as [Owen87], [Fisk88], [Hube90], for configurable arithmetic systems. Figure 6 relates our approach to other methods of arithmetic synthesis, including special-purpose processors and dynamically programmable gate arrays (DPGAs).

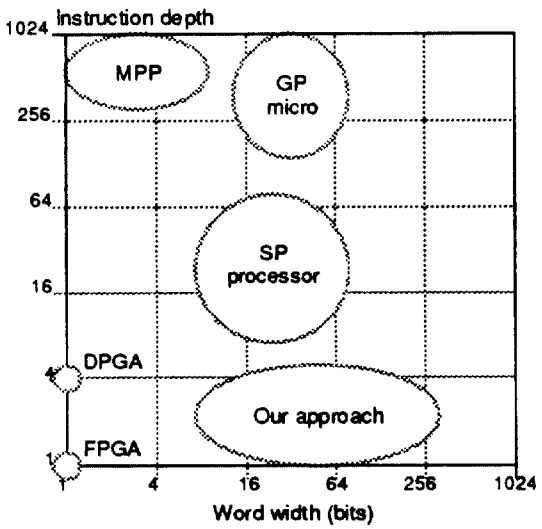

Fig. 6. The hardware design space for implementing arithmetic-intensive applications (after [DeHo00]).

References

[DeHo00] DeHon, A., "The Density Advantage of Configurable Computing," *IEEE Computer*, Vol. 33, pp. 41-49, Apr. 2000.

[Do98] Do, T.-T., H. Kropp, C. Reuter, and P. Pirsch "A Flexible Implementation of High-Performance FIR Filters on Xilinx FPGAs," in [Hart98], pp. 441-445.

[Erce84] Ercegovac, M.D., "On-Line Arithmetic: An Overview," *Real Time Signal Processing VII*, SPIE Proc. 495, 1984, pp. 86-93.

[Erce87] Ercegovac, M.D. and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations," *IEEE Trans. Computers*, Vol. 36, pp. 895-897, July 1987.

[Erce88] Ercegovac, M.D. and T. Lang, "On-Line Arithmetic: A Design Methodology and Applications," *VLSI Signal Processing III* (Proc. IEEE Workshop), 1988, pp. 252-263.

[Fisk88] Fiske, S. and W.J. Dally, "The Reconfigurable Arithmetic Processor," *Proc. Int'l Symp. Computer Architecture*, 1988, pp. 30-36.

[Guse92] Gusev, M. and D.J. Evans, "VLSI Processor Array IPS Cells," *Parallel Computing*, Vol. 18, pp. 997-1007, 1992.

[Hart98] Hartenstein, R.W. and A. Keevallik, *Field-Programmable Logic and Applications: From FPGAs to Computing Paradigm* (Proc. 8th Int'l Workshop), Springer, 1998.

[Hayn96] Haynal, S. and B. Parhami, "Arithmetic Structures for Inner-Product and Other Computations Based on a Latency-Free Bit-Serial Multiplier Design," *Proc. 30th Asilomar Conf. Signals, Systems, and Computers*, Nov. 1996, pp. 197-201.

[Hube90] Huber, M., J. Teich, and L. Thiele, "Design of Configurable Processor Arrays," *Proc. Int'l Symp. Circuits and Systems*, Vol. 2, pp. 970-973, May 1990.

[Klot96] Klotchkov, I.V. and S. Pederson, "A Codesign Case Study: Implementing Arithmetic Functions in FPGAs," *Conf. Engineering of Computer-Based Systems*, Jerusalem, Mar. 1996, pp. 389-395.

[Kwai96] Kwai, D.-M. and B. Parhami, "FFT Computation with Linear Processor Arrays Using a Data-Driven Control Scheme," *J. VLSI Signal Processing*, Vol. 13, pp. 57-66, 1996.

[Kwai97] Kwai, D.-M. and B. Parhami, "An On-Line Fault Diagnosis Scheme for Linear Processor Arrays," *Microprocessors and Microsystems*, Vol. 20, No. 7, pp. 423-428, 1997 Mar. 17.

[Kwai99] Kwai, D.M. and B. Parhami, "High-Performance Array Processing with Fully Pipelined Data Streams and Control Paths," *Proc. 11th Int'l Conf. Parallel and Distributed Computing and Systems*, Nov. 1999, pp. 609-612.

[Luo00] Luo, Z. and M. Martonosi, "Accelerating Pipelined Integer and Floating-Point Accumulations in Configurable Hardware with Delayed Addition Techniques," *IEEE Trans. Computers*, Vol. 49, No. 3, pp. 208-218, Mar. 2000.

[Mang97] Mangione-Smith, W.H. et al, "Seeking Solutions in Configurable Computing," *IEEE Computer*, Vol. 30, pp. 38-43, Dec. 1997.

[Mill98] Miller, N.L. and S.F. Quigley, "A Novel Field Programmable Gate Array Architecture for High Speed Arithmetic Processing," in [Hart98], pp. 386-390.

[Owen87] Owens, R.M. and M.J. Irwin, "The Arithmetic Cube," *IEEE Trans. Computers*, Vol. 36, No. 11, pp. 1342-1348, Nov. 1987.

[Parh90] Parhami, B., "Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations," *IEEE Trans. Computers*, Vol. 39, pp. 89-98, Jan. 1990.

[Parh99] Parhami, B. and D.-M. Kwai, "Data-Driven Control Scheme for Linear Arrays: Application to a Stable Insertion Sorter," *IEEE Trans. Parallel and Distributed Systems*, Vol. 10, pp. 23-28, Jan. 1999.

[Parh00] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford, New York, 2000.

[Rath99] Ratha, N.K. and A.K. Jain, "Computer Vision Algorithms on Reconfigurable Logic Arrays," *IEEE Trans. Parallel and Distributed Systems*, Vol. 10, No. 1, pp. 29-43, Jan. 1999.

[Shir95] Shirazi, N., A. Walters, and P. Athanas, "Quantitative Analysis of Floating-Point Arithmetic on FPGA Based Custom Computing Machines," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, Apr. 1995.

[Tiou99] Tiountchik, A. and E. Trichina, "FPGA Implementation of Modular Exponentiation," *Proc. IPPS/SPDP Workshop on Reconfigurable Architectures*, Apr. 1999, pp. 712-715.

[Vill98] Villasenor, J. and B. Hutchings, "The Flexibility of Configurable Computing," *IEEE Signal Processing Magazine*, Vol. 15, No. 5, pp. 67-84, Sep. 1998.