CHAPTER 24

DISTRIBUTED ARITHMETIC

Rajeevan Amirtharajah
Department of Electrical and Computer Engineering
University of California–Davis

Distributed arithmetic (DA) [1, 2] is a computation algorithm that performs multiplication using precomputed lookup tables (LUTs) instead of logic. It is well suited to implementation on homogeneous field-programmable gate arrays (FPGAs) because of its high utilization of the available LUTs. It may also have advantages for modern heterogeneous FPGAs that contain built-in multipliers because it is area efficient for implementing long digital filters. DA targets the sum-of-products (or vector dot product) operation, and many digital signal processing (DSP) tasks such as filter implementation, matrix multiplication, and frequency transformation can be reduced to one or more sum-of-products computations.

24.1 THEORY

The theory behind DA is based on reorganizing the vector dot product operation around the binary representation of the vector elements [2]. Suppose that X is the vector of input samples and A is a constant vector of filter coefficients, corresponding to the taps of a finite impulse response (FIR) filter. Vectors X and A each consist of M elements X_k and A_k. The dot product y of X and A (corresponding to the convolution of X with the FIR impulse response) can be written as

$$y = \sum_{k=0}^{M-1} A_k X_k \tag{24.1}$$

We can represent each element of the input sample vector X in N-bit 2's complement notation. Then equation 24.1 can be expressed as

$$y = \sum_{k=0}^{M-1} A_k \left[-b_{k(N-1)} 2^{N-1} + \sum_{n=0}^{N-2} b_{kn} 2^n \right] \tag{24.2}$$

where $b_{k(N-1)}$ is the sign bit of the input sample X_k in N-bit 2's complement notation, and b_{kn} is the nth bit of input sample X_k. The possible values of b_{ki}

are either 0 or 1. Equation 24.2 can be further rearranged into equation 24.3 by multiplying out the factors and changing the order of the summation:

$$y = -\sum_{k=0}^{M-1} A_k b_{k(N-1)} 2^{N-1} + \sum_{n=0}^{N-2} \left[\sum_{k=0}^{M-1} A_k b_{kn} \right] 2^n = Z_{sign} + Z_{n1} \tag{24.3}$$

Consider each term in the brackets of the second summation in equation 24.3, labeled Z_{n0} in the following:

$$Z_{n0} = \sum_{k=0}^{M-1} A_k b_{kn} \tag{24.4}$$

where term Z_{n0} has 2^M possible values because b_{kn} is either 1 or 0. Therefore, each summation term $A_k b_{kn}$ can have the value of either A_k or 0. Instead of using a multiplier to compute any of these 2^M possible values whenever necessary, we can precompute them and store them in a LUT with depth 2^M. The contents of the LUT are then addressed directly by the bit-serial input data, $[b_{0n}, b_{1n}, b_{2n}, \ldots b_{Mn}]$, corresponding to the nth bits of each element X_k of input vector X. Multiplication by the factor 2^n in equation 24.3 can be realized by a shifter and the addressed LUT contents shifted and accumulated to form term Z_{n1} in $(N-1)$ cycles.

The sign term Z_{sign} can be handled in the same way with additional circuitry to implement subtraction; it takes one additional clock cycle. The final result y is formed after N cycles. Note that, if the filter length is greater than the bit width of the input data (i.e., $M > N$), DA computes the final result in fewer cycles than an implementation using a single multiply–accumulate functional unit. However, because the size of the LUT grows exponentially in the number of vector elements (2^M), most practical implementations use multiple LUTs and adders to combine partial dot products into the final result.

24.2 DA IMPLEMENTATION

A simple DA implementation is shown in Figure 24.1. It requires a 16-bit shift register for the input vector, a 16-entry LUT, an adder/subtractor, and an accumulator (`Result`) for the output. The `x2` operation is handled purely by wiring. This unit is a direct implementation of the DA algorithm described in the preceding section, and it is capable of computing the dot product of a 4-element vector X and a constant 4-element vector A.

In the figure the four 4-bit-wide elements of X are fed into the address decoder in most significant bit (MSB) first order to select the appropriate LUT row contents. The selected content is added with the left-shifted version of the previous `RESULT` value to form the current `RESULT` value. `Ts` is the sign bit timing signal that controls the add/subtract operation; when `Ts` is high, the current LUT content is subtracted from the left-shifted version of the previous result. The final vector dot product is obtained in four cycles. Shifting in the bit vector

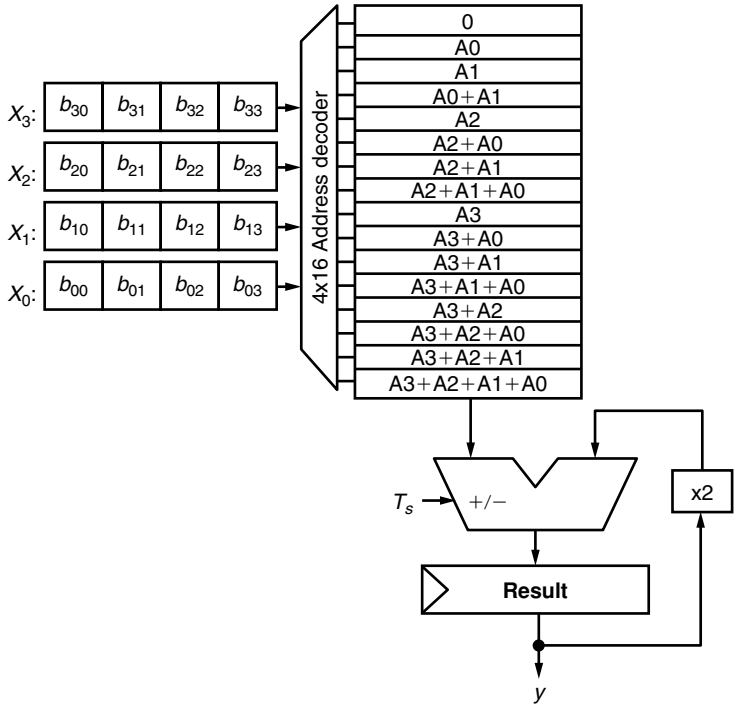

FIGURE 24.1 ■ A simple implementation of distributed arithmetic.

least significant bit (LSB) first also produces the correct final value and has the advantage of eliminating long carry propagations when accumulating the intermediate results.

The only modifications to Figure 24.1 required for this alternative are to reverse the bits of vector X_{in}, the shift register, and replace the left shift by 1 bit and the right shift by 1 bit. Various other modifications to this structure are possible. For example, the input sample shift register can be serial in/serial out or parallel in/serial out depending on the application.

LUT size can be a determining factor in the total hardware cost of a DA implementation. It is possible to modify the structure in Figure 24.1 to reduce the table size by a factor of 2. To achieve this reduction, consider a different representation of the input data samples X_k:

$$X_k = \frac{1}{2}[X_k - (-X_k)] \tag{24.5}$$

The 2's complement representation of the negative of X_k can be expressed as

$$-X_k = -\overline{b}_{k(N-1)}\,2^{N-1} + \sum_{n=0}^{N-2} \overline{b}_{kn}\,2^n + 1 \tag{24.6}$$

where each bit of X_k has been complemented and a 1 has been added to the complemented bits. Plugging equation 24.6 into equation 24.5 yields

$$X_k = \frac{1}{2}\left[-\left(b_{k(N-1)} - \overline{b}_{k(N-1)}\right)2^{N-1} + \sum_{n=0}^{N-2}\left(b_{kn} - \overline{b}_{kn}\right)2^n - 1\right] \tag{24.7}$$

Each difference term $\left(b_{kn} - \overline{b}_{kn}\right)$(for $n = 0$ to $N-1$) in equation 24.7 can take on values of +1 or −1. This alternate representation for X_k is convenient because, in the resulting summation for the dot product, each linear combination of A_k has a corresponding negative linear combination. Only one of these combinations needs to be stored in the LUT, with the negative being applied during operation using the subtractor. Substituting equation 24.7 into equation 24.1 and rearranging terms yields the following new expression for the result of the dot product y:

$$y = \sum_{n=0}^{N-1} Q(b_n) + Q(0) \tag{24.8}$$

where

$$Q(b_n) = \frac{1}{2}\sum_{k=0}^{M-1} A_k\left(b_{kn} - \overline{b}_{kn}\right)2^n, \quad n \neq N-1 \tag{24.9a}$$

$$Q(b_{N-1}) = -\frac{1}{2}\sum_{k=0}^{M-1} A_k\left(b_{k(N-1)} - \overline{b}_{k(N-1)}\right)2^{N-1}, \quad n = N-1 \tag{24.9b}$$

$$Q(0) = -\frac{1}{2}\sum_{k=0}^{M-1} A_k \tag{24.9c}$$

Note that the expressions for $Q(b_n)$ and $Q(b_{N-1})$ have 2^{M-1} possible magnitudes, with signs determined by the input bits, and that the computation of y requires an additional register to hold the constant term $Q(0)$. This leads to the reduced DA memory implementation shown in Figure 24.2, where the exclusive-or (XOR) gates are required to recode the addresses to access the appropriate LUT row and to control the timing of the sign bit into the adder/subtractor. The XOR gates, the initial condition register for $Q(0)$, and a 2-input multiplexer are the only additional hardware required to reduce the memory size by a factor of 2.

The implementations in both Figures 24.1 and 24.2 require N clock cycles to compute the final result, although additional cycles may be needed to match the throughput of the DA unit to other functional units in the system for a particular application. In Section 24.3 we will discuss mapping these basic structures onto FPGA fabrics. We will address the issue of performance improvement (by reducing the number of required clock cycles and increasing the clock frequency) in Section 24.4.

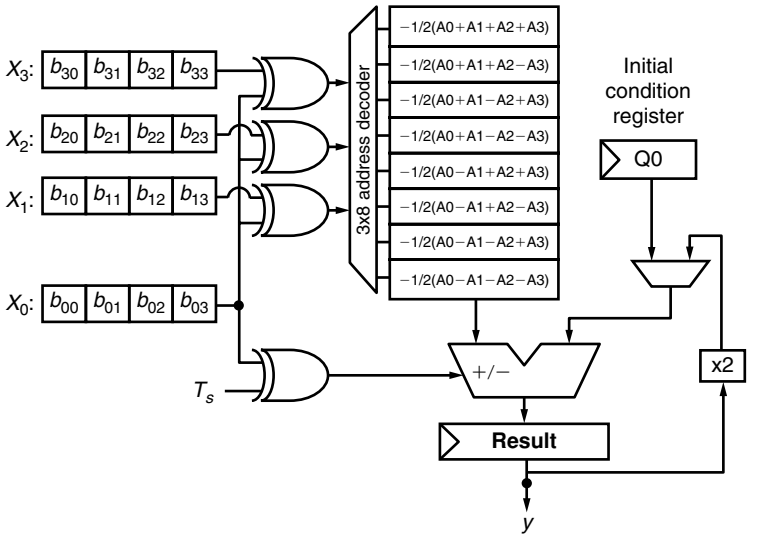

FIGURE 24.2 ■ Reduced DA memory implementation.

24.3 MAPPING DA ONTO FPGAs

Consider mapping a 16-tap FIR filter ($M = 16$) operating on 16-bit data ($N = 16$) onto an FPGA fabric based on 4-input LUTs. As discussed earlier, DA's primary drawback is that the size of the LUTs grows exponentially in the number of filter coefficients (or filter taps). If we want to use 16-bit data to represent the precomputed values, we need $16 \times 2^{16} = 1$ Mbit of memory. To limit this growth, long filters can be partitioned into several smaller DA units whose outputs are then combined using a tree of 2-input adders, as shown in Figure 24.3. This partitions the 16 filter taps A_0 to A_{15} among four DA units, each of which incorporates N 1-bit-wide 4-input LUTs.

The partitioning is chosen to correspond to the LUT size of the individual logic elements or CLBs. If the filter taps are symmetric (which they often are for typical signal-processing applications), the memory size can be reduced by a further factor of 2 by summing the appropriate elements of the input vector X_k using serial addition and using the bits of the resulting sum to address the LUTs. In addition to the serial adder hardware, this memory reduction comes at the expense of an additional clock cycle of latency before the final result is valid.

As CMOS technology has scaled and the complexity of individual CLBs has increased with succeeding FPGA generations, the hardware cost of implementing our example filter has shrunk dramatically. Based on an early implementation of an 8-tap, 8-bit filter using DA on a Xilinx 3042 FPGA [3], our example would consume approximately 120 CLBs, including control logic, even using the

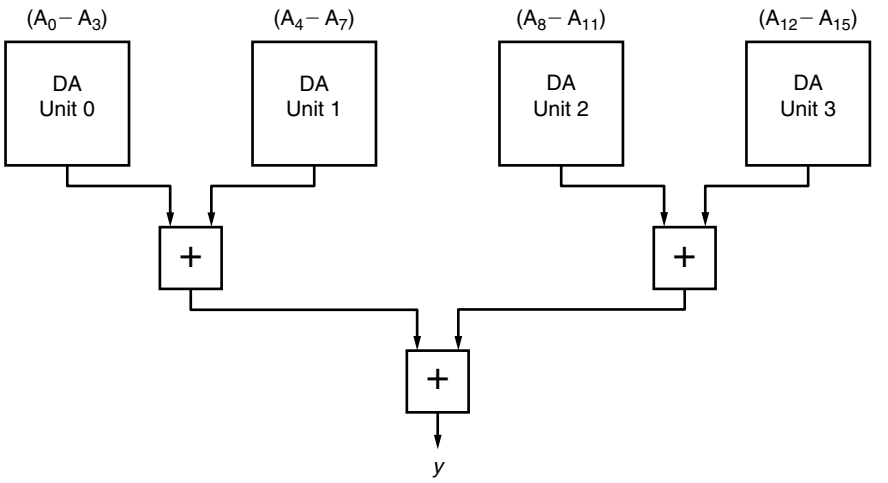

FIGURE 24.3 ■ A 16-tap FIR filter mapped onto multiple DA units.

symmetry of the filter coefficients to reduce the memory requirements. This would consume roughly the entire FPGA chip. Resource usage would be dominated by the input shift registers (60 CLBs) since this older FPGA architecture only allowed the local CLB flip-flops to be used in a shift configuration.

In contrast, a recent FPGA architecture encompasses four logic "slices" in each CLB, where two slices each roughly correspond to an entire CLB in the older architecture [6]. Because LUTs in Xilinx Spartan-3E FPGAs can be configured as 16×1 shift registers, the number of CLB resources to implement the data memory for DA is drastically reduced. Each logic slice also contains carry propagation logic for efficient implementation of adder circuits, which can be used to increase the speed of DA computation, as will be shown later. Implementing the example filter on a Spartan-3E FPGA requires approximately 113 slices, corresponding to 29 CLBs. This is under 12 percent of the total number of slices available in the smallest member of the 3S100E FPGA family.

Further enhancements to the architecture building blocks may allow for more efficient DA implementation in the future. For example, the potential of heterogeneous or coarse-grained FPGAs to support DA more efficiently by incorporating small adders and accumulators directly in the CLB is currently being explored [7].

24.4 IMPROVING DA PERFORMANCE

Two approaches can be taken to improve DA performance on an FPGA platform. First, the design can be modified to reduce the number of cycles required to compute the final result. Second, the cycle time can be decreased by reducing

the number of logic stages in the critical path of the computation. Examples of both approaches will be discussed in this section.

A simple approach to speeding up DA computation is to recognize that multiple bits from each input vector element X_k can be used to address multiple LUTs in each clock cycle (because addition is associative, we can perform the sum in equation 24.3 using any combination of partial sums that is convenient). This leads to an architecture like the one shown in Figure 24.4, which uses 2 bits of the input data vector elements at a time. The LUTs are identical because they contain the same linear combinations of filter coefficients A_k. The LUT outputs must be scaled by the correct exponent of 2 to maintain the significance of the bits added to the accumulated result (the `x2` unit in Figure 24.4). Only two cycles are required to compute the result y for this implementation, instead of four cycles for the implementation in Figure 24.2. For longer bit-width input data, this idea can be extended to using more bits at a time.

The modification just described provides the benefit of a linear decrease in the number of clock cycles at the expense of a linear increase in LUT memory size. In addition, the number of inputs and the bit width of the adder/subtractor must increase. Mapping this approach onto an FPGA involves a trade-off between the routing resources consumed and the speed of the computation, as the input data bit vectors must be divided into subwords and distributed to multiple CLBs. In addition, multiple LUT outputs must be accumulated at a single destination to form the result, which consumes further routing.

Following a derivation similar to that presented by White [2], we can analyze this trade-off quantitatively. Suppose that we are implementing an M-tap filter

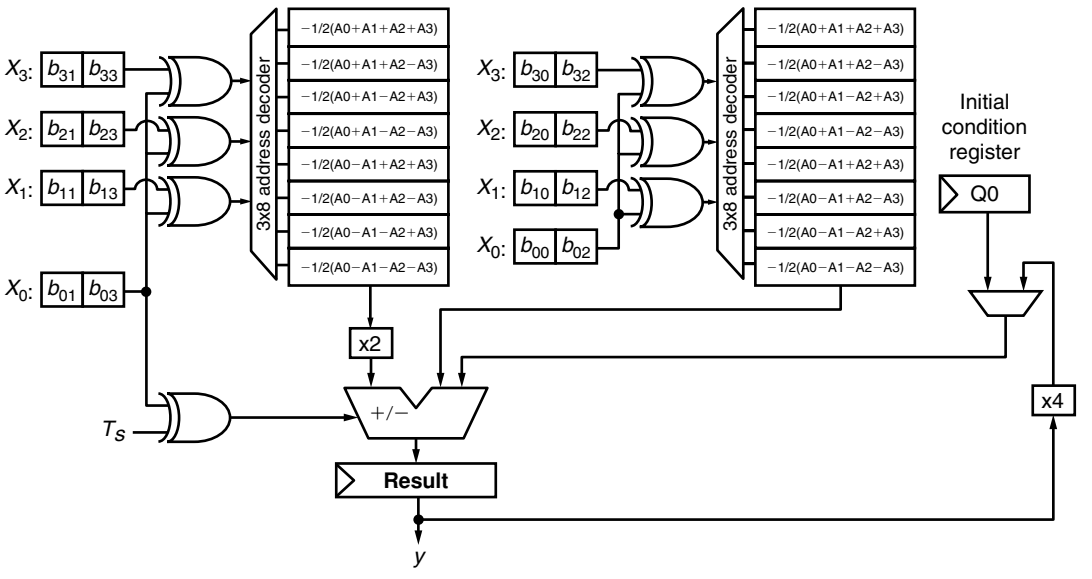

FIGURE 24.4 ■ Two-bit-at-a-time reduced memory DA implementation.

using an N-bit number representation and that the computation is proceeding L bits at a time. Further suppose that the LUT data is W bits wide. Computing the result requires that, in each cycle, MN bits are shifted in and WL bits are read out, and N/L clock cycles must pass. The number of wires N_W is therefore

$$N_W = \frac{MN}{N/L} + WL = (M + W)L \tag{24.10a}$$

If we define the relative importance of minimizing routing resources to minimizing latency as the ratio r, then

$$r = \frac{N/L}{N_W} = \frac{N}{(M + W)L^2} \tag{24.10b}$$

and we can find the L that satisfies our design criterion of relative importance r:

$$L = \left\lceil \sqrt{\frac{N}{r(M + W)}} \right\rceil \tag{24.10c}$$

Now suppose that an application demands low latency and that routing resources are not too tightly constrained; then, for $r = 2$, 32-bit input data ($N = 32$), a 4-tap FIR filter ($M = 4$), and 4-bit LUT data ($W = 4$); this yields $L = 2$. The desired DA implementation takes the input data 2 bits at a time to address the LUTs, completing a dot product computation in 16 cycles.

In addition to exploiting parallelism to speed up the DA computation, it is possible to employ various levels of pipelining. As we saw in Figure 24.1, the critical path involves decoding the address presented by the data shift registers, accessing the row from the LUT, and propagating the carry through the adder/subtractor while meeting the setup time constraints for the accumulator. If the implementation spans multiple CLBs, there is a potentially significant interconnect delay in this critical path in addition to the combinational logic delay. An obvious way to pipeline the simple implementation is to make the LUT synchronous and latch the outputs before they are fed to the adder/subtractor.

An alternative approach is to use carry save addition to reduce the carry propagation chain in the critical path [8]. The key modification to Figure 24.1 is to use a different structure for the adder/subtractor and to perform the computation in LSB first order. Instead of using a carry propagate adder to accumulate the entire result in one clock cycle, the adder/subtractor is pipelined at the bit level and the sum and carry outputs are stored in flip-flops at each cycle. Each full adder takes one input bit from the LUT output and one from the sum output of the next most significant full adder, automatically accounting for the `x2` scaling required in Figure 24.1. Assuming that the accumulator is wider than N bits, after N clock cycles the least significant N bits of the final result are stored in the LSBs of the accumulator while the remaining MSBs require one more carry propagating addition to produce the final result. This operation adds one extra clock cycle to the latency of the DA computation.

Most modern FPGA fabrics have dedicated paths for high-speed carry propagation. Given that most DA designs require accumulators with not too many more than N bits, the final carry propagation is typically not the critical path for the entire computation. The throughput is determined by the speed of the carry save addition in the accumulator.

Although using carry save addition at the single-bit level results in the greatest speed improvement, it is also the most resource intensive in terms of logic slices and CLBs. A speed versus area trade-off can be achieved by partitioning the adder/subtractor into multiple subcircuits, each of which propagates a carry across p bits ($p = 1$ in the example just described). Speedup factors of at least 1.5 have been observed over the traditional design shown in Figure 24.1 [8].

24.5 AN APPLICATION OF DA ON AN FPGA

In addition to FIR filters, a common DA application on FPGAs is acceleration of frequency transformations such as the discrete cosine transform (DCT), which is a critical component of the MPEG video compression and JPEG image compression standards. The two-dimensional DCT can be implemented as two one-dimensional DCTs and a matrix transposition. Each DCT can be implemented as a matrix–vector multiplication, which is easy to implement on an FPGA using DA because it can be decomposed into a set of vector dot products.

In one example, using DA instead of multiply–accumulate for the DCT resulted in a factor of 2.4 reduction in area for the FPGA implementation (on a Xilinx XC6200 FPGA) [9]. Using DA and pipelining of the routing to improve the algorithm performance, this implementation was fast enough to process VGA resolution images (640 $\times$ 480 pixels) at 25 frames per second—approximately four times faster than a full software implementation running on a microprocessor. The entire two-dimensional DCT consumed a 64 $\times$ 78 array of logic blocks on the chip (about 30 percent of the total FPGA area) and the DA portions of the DCT consumed 3648 logic blocks, or about 70 percent of the two-dimensional DCT total. The average utilization of each logic block for the DA components was 61 percent. This high level of utilization was a result of careful floorplanning in addition to DA's inherent suitability to FPGA implementation.

References

[1] Xilinx, Inc. *The Role of Distributed Arithmetic in FPGA-based Signal Processing*, Xilinx, Inc. (*http://www.xilinx.com/appnotes/theory1.pdf*), January 2006.

[2] S. A. White. Applications of distributed arithmetic to digital signal processing: A tutorial review. IEEE *ASSP Magazine* 6(3), July 1989.

[3] L. Mintzer. FIR filters with field-programmable gate arrays. *Journal of VLSI Signal Processing* 6, 1993.

[4] G. Roslin. A guide to using field-programmable gate arrays (FPGAs) for application-specific digital signal processing performance. Xilinx white paper, 1995.

[5] W. Wolf. *FPGA-based System Design* (Modern Semiconductor Design Series), Prentice-Hall, 2004.
[6] Xilinx, Inc. *Spartan-3E FPGA Family: Complete Data Sheet*, DS312 (v2.0) (*http://www.xilinx.com*), November 2005.
[7] B. Calhoun, F. Honore, A. Chandrakasan. A leakage reduction methodology for distributed MTCMOS. *IEEE Journal of Solid-State Circuits* 39(5), May 2004.
[8] R. Grover, W. Shang, Q. Li. A faster distributed arithmetic architecture for FPGAs. *Proceedings of the 10th ACM International Symposium on Field-Programmable Gate Arrays*, February 2002.
[9] R. Woods, D. Trainor, J.-P. Heron. Applying an XC6200 to real-time image processing. *IEEE Design & Test of Computers* 15(1), January/March 1998.