FILTER SYNTHESIS USING FINE-GRAIN DATA-FLOW GRAPHS

Waqas Akram, Cirrus Logic Inc., Austin, Texas

Abstract: this project is concerned with finding ways to synthesize hardware-efficient digital filter algorithms given technology and data-rate constraints. The synthesis flow targets embedded systems implemented in application specific integrated circuits (ASICs). The flexibility inherent in such custom implementations provides opportunities for optimization down to the bit-level. This effort attempts to construct a convenient framework for the architectural manipulation of filter algorithms at the bit-level, in order to reduce hardware complexity while meeting fixed data-rate constraints. The ultimate goal here is to take an applicative language description of an algorithm and transform it into the most hardware-efficient descriptive language representation at the bit-level.

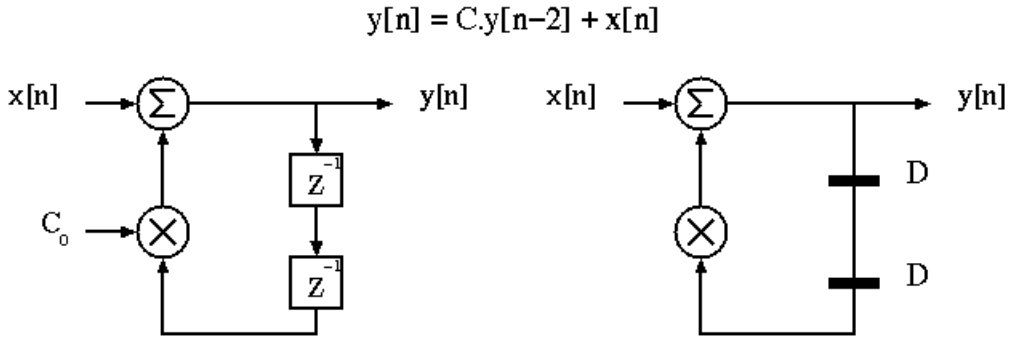
1.0 INTRODUCTION

In order to maintain compatibility with products from a wide range of vendors, most widely deployed systems adhere to openly available standards which normally dictate parameters such as data-rates and packet lengths. The key challenge here is to create a design that meets the specifications of such standards, while providing the most cost-effective solution possible. Rapidly changing standards have created a strong desire for system re-configurability, making programmable Digital Signal Processors (DSPs) an attractive implementation choice. Consequently, most efforts to expose and exploit the parallelism and concurrency in signal processing algorithms have focused on taking advantage of multiple execution units (inter-DSP as well as intra-DSP). These efforts have traditionally been applied at the operation-level, instruction-level, as well as basic execution unit -level.

The motivation for this project stems from the fact that application-specific hardware implementations are still more attractive for low-power/low-memory/low-cost embedded applications (like portable devices), as well as high data-rate communication systems.

The techniques already developed for exploiting parallelism and concurrency at the instruction level can be used equally effectively at the bit-level. The use of dataflow graphs for representing DSP algorithms has found much success with researchers, due to the manipulative convenience and the inherently intuitive ties with the flow of data through such systems. Consequently, most commonly used transformations have been applied on some form of dataflow graph or another. In order to extend these techniques to bit-level manipulation, some modifications need to be made to existing dataflow models of computation.

FIGURE 1. Dataflow representation of a simple recursive filter



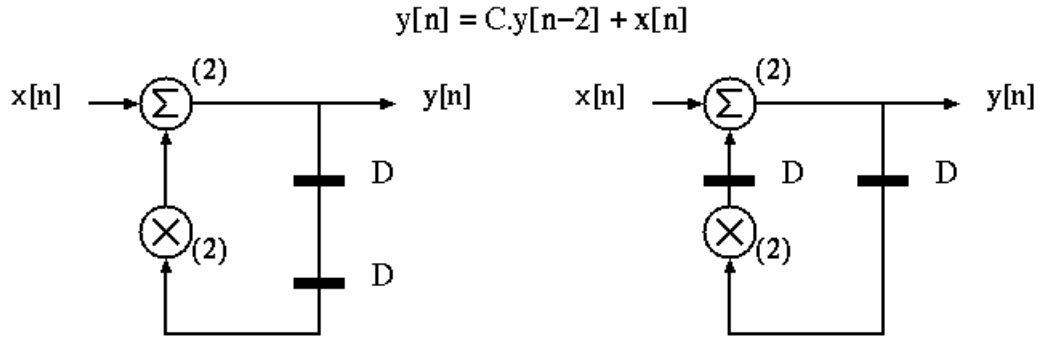
2.0 REVIEW

Digital filters can be modeled as iterative homogeneous synchronous dataflow (SDF) graphs. Certain classes of digital filters, such as recursive and adaptive filters, contain feedback loops which impose a lower bound on the iteration period. An example dataflow graph representation of a simple recursive filter is shown in Figure 1.

The problem of finding the maximum sampling rate of recursive algorithms is considered in [1] by introducing the concept of a loop iteration bound for such systems. The iteration bound is tied with a particular dataflow graph description. The iteration bound of a loop is defined as the total computation time for the loop, divided by the number of delay elements in that loop. The iteration bound of a dataflow graph is the highest loop bound present in that graph.

Retiming [2] is a technique for reducing the critical path delay of a dataflow graph. The critical path in a dataflow graph has the longest computation time among all paths that contain zero delays. Bottlenecks are eliminated by redistributing the critical path delay across other paths, thereby reducing the sample period, enabling a higher data-rate. Retiming is performed by relocating delay elements around the graph without changing the input/output functionality of the system. Both the number of delays in the graph and the iteration bound remain unchanged after retiming. An example of retiming a simple recursive filter is shown in Figure 2. The numbers in brackets next to the nodes represent computation delays associated with those nodes. The initial dataflow graph has an iteration bound of 2, and a critical path of 4 time units. After retiming, the critical path has attained the iteration bound of 2.

FIGURE 2. Retiming Example



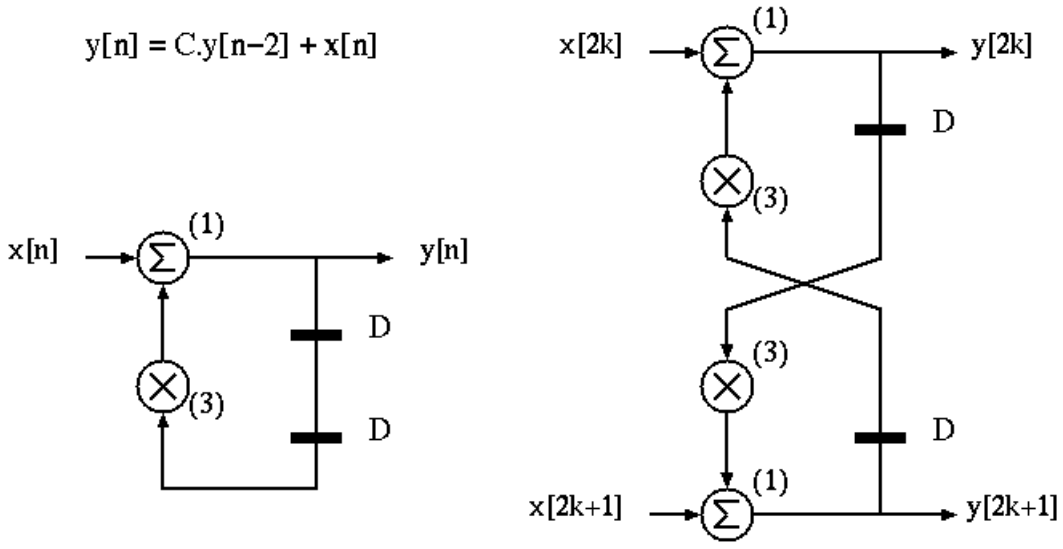
Under certain conditions, retiming can also be used to reduce hardware complexity and power consumption. This can be done by moving delays from multiple paths to single paths (for example, moving a delay element from each input path of an adder to its output path). At the word-level, this represents a lower delay element cost, but at the bit-level, the cost at the output may be greater depending on the bit-width of the binary result word.

Various ways in which dataflow graphs can be transformed into more parallel architectures are described in [3], including unfolding and folding. Individual iterations of an algorithm can be represented as linear dependency graphs, and retiming can be used to reduce the critical path. However, any parallelism present across iterations cannot be exploited, unless we unfold the recursion (either partially or completely). Unfolding is used to represent multiple iterations of a loop as a single iteration, while preserving the number of delays as well as precedence constraints in the dataflow graph. Zero-delay paths in the DFG obey intra-iteration precedence constraints and non-zero delay paths obey inter-iteration precedence constraints.

There can be cases where the iteration bound cannot be attained without unfolding, such as when a node in the loop needs more computation time than the iteration bound itself. Assuming the node cannot be broken down into smaller components, retiming will not help. Figure 3 shows such an example. In the initial dataflow graph, the multiplier takes 3 time units to execute and the iteration bound is 2. Retiming cannot reduce the critical path below 3, which limits the throughput. If we unfold the dataflow graph by a factor of 2, as shown on the right in Figure 2, we can attain the iteration bound of 2, even though the critical path remains at 4 time units. Such, and other, techniques of optimum unfolding are discussed in [4], [5], [6]. Folding is another technique for dataflow graph transforma-

tion, and is the reverse of unfolding, and represents the re-use of DSP hardware by time-division multiplexing. Such transformations require careful control circuitry design [7].

FIGURE 3. Unfolding Example (ITERATION BOUND = 2)



3.0 OBJECTIVE

In the design of application-specific hardware, the trade-off between critical path reduction and register minimization is better answered at the bit-level, instead of the word level. A direct-form FIR filter is shown in Figure 4, where the input samples and filter coefficients are both 8-bits wide. The critical path is 5, and 16 bit delay elements are needed.

FIGURE 4. Direct form FIR Filter

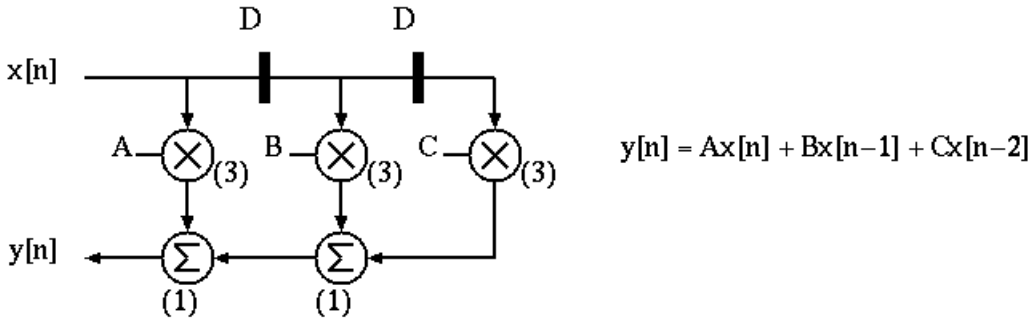


Figure 5 shows the retimed (or transposed) FIR filter with a critical path of 4. This graph requires 32 bit delay elements, due to the extra precision required at the outputs of the computational units. Note that word-level retiming would predict equal resource requirements for both of these architectures.

FIGURE 5. Transposed-form (retimed) FIR filter

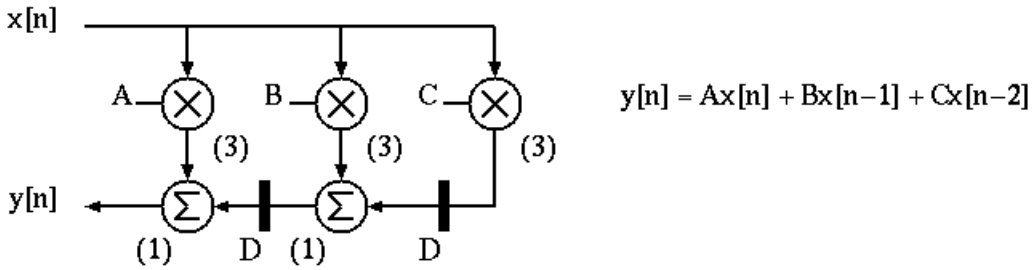
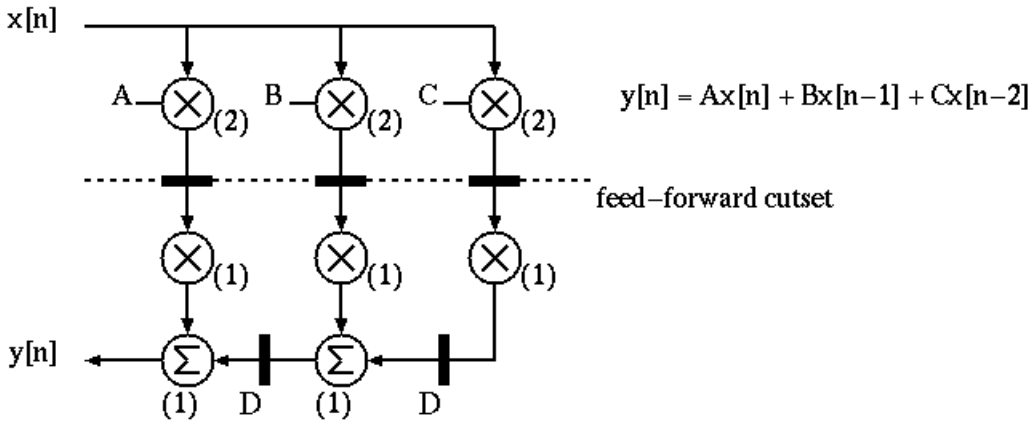


Figure 6 shows an example of fine-grain cutset retiming, where an extra delay element has been introduced in order to pipeline the multipliers. Note that the critical path is now 2 time units. However, apart from the 3 new word delay elements, there is no information on how costly this transformation is in terms of hardware complexity. Some trade-offs in pipeline granularity at the bit-level are considered in [8].

FIGURE 6. Fine-grain cutset retiming



A slightly enhanced model of computation needs to be developed: fine-grain data-flow might behave like homogeneous SDF, but each token needs to possess a “bit-width”

attribute (which would translate directly into a factor for actual delay element computation). It would be erroneous to model this as non-homogeneous SDF using “bit-width” number of tokens on each edge, since all “bit-width” tokens exist on the edge as a single unit. Also, a cost function needs to be maintained for each node.

FIGURE 7. Filter synthesizer architecture

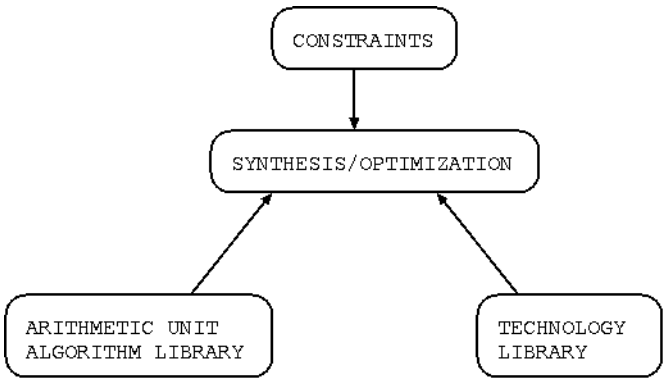


Figure 7 shows a possible implementation of this system. The arithmetic unit algorithm library will contain multiple fine-grain data-flow graph representations of commonly used computational units, including multipliers, adders and multiply-accumulate units (future implementations might include butterfly and add-compare-select units). The technology library will include cost functions for each component (delay elements and fine-grain graph nodes). Alternatively, these could be embedded in the arithmetic unit library. The actual synthesis would consist of the following:

- provide a set of data-rate constraints, and a coarse-grain dataflow graph of the filter
- the synthesis/optimization unit creates a fine-grain representation of this dataflow graph
- perform retiming/unfolding/folding transformations in order to meet the data-rate, and then to reduce hardware complexity,
- convert the resulting optimized fine-grain dataflow graph into a descriptive language representation.

There are three components to such an endeavor: (1) creating efficient data structures for representing and transforming fine-grain dataflow graphs, (2) designing an efficient transformation algorithm for convergence to an optimum solution, given constraints, and (3) test the system on a representative set of filter algorithms.

4.0 REFERENCES

[1] M. Renfors and Y. Neuvo, “The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints,” IEEE Transactions on Circuits and Systems, Vol. CAS-28, No. 3, pp. 196-202, March 1981.

[2] C. E. Leiserson, F. Rose, and J. Saxe, “Optimizing Synchronous Circuitry for Retiming,” 3rd Caltech Conference on VLSI, pp. 87-116, March 1983.

[3] K.K. Parhi, “Algorithm Transformations for Concurrent Processors,” Proceedings of the IEEE, Vol. 77, No. 12, pp. 1879-1895, December 1989.

[4] K.K. Parhi and D.G. Messerschmitt, “Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding,” IEEE Transactions on Computers, vol. 40, no. 2, pp. 178-195, February 1991.

[5] L.-F. Jeng and L.-G Chen, “rate-optimal DSP synthesis by pipeline and minimum unfolding,” IEEE Transactions on VLSI Systems, vol. 2, no. 1, pp. 81-88, March 1994.

[6] L.-F. Chao and E. Sha, “Retiming and unfolding data-flow graphs,” Proceedings of 1992 International Conference on Parallel Processing, part II, pp. 33-40, August 1992.

[7] K. K. Parhi, C.-Y. Wang, and A. P. Brown, “Synthesis of Control Circuits in Folded Pipelined DSP Architectures,” IEEE Journal of Solid-State Circuits, Vol. 27, No. 1, pp. 29-43, January 1992.

[8] C. Nagendra, R. M. Owens, and M. J. Irwin, “Design Trade-offs in High-Speed Multipliers and FIR Filters,” 9th International Conference on VLSI Design, January 1996.