Algorithms Transformation Techniques for Low-Power Wireless VLSI Systems Design

Naresh R. Shanbhag[1]

This paper presents an overview of algorithm transformation techniques and discusses their role in the development of hardware-efficient and low-power VLSI algorithms and architectures for communication systems. Algorithm transformation techniques such as retiming, look-ahead and relaxed pipelining, parallel processing, folding, unfolding, and strength reduction are described. These techniques are applied statically (i.e., during the system design phase) and hence are referred to as *static algorithm transformations* (SATs). SAT techniques alter the structural and functional properties of a given algorithm so as to be able to jointly optimize performance measures in the algorithmic (signal-to-noise ratio [SNR] and bit error rate [BER]) and VLSI (power dissipation, area and throughput) domains. Next, a new class of algorithm transformations referred to as *dynamic algorithm transformations* (DAT) is presented. These transformations exploit the nonstationarity in the input signal environment to determine and assign minimum computational requirements for an algorithm in real time. Both SAT and DAT techniques are poised to play a critical role in the development of low-power wireless VLSI systems given the trend toward increasing digital signal processing in these systems.

KEY WORDS: Low power; VLSI; wireless; architectures.

1. INTRODUCTION

We are witnessing a tremendous growth in the communications arena in general and wireless communications in particular. The latter includes traditional services such as paging, cellular communications, and satellite systems, and the more recent personal communications services (PCS) [1], wireless local area networks (WLANs) [2], etc. In this paper, we will present techniques that jointly optimize algorithm and VLSI performance. These techniques are applicable to general digital signal processing (DSP) or communications applications. We will indicate the relevance of these techniques in the design of low-power wireless transceivers, especially those parts that involve intensive digital signal processing.

[1] Coordinated Science Laboratory/ECE Dept., University of Illinois at Urbana-Champaign, 1308 West Main Street, Urbana, Illinois 61801; email: shanbhag@uivlsi.csl.u iuc.edu.

The processing in wireless transceivers can be partitioned into radio frequency (RF) processing and baseband digital processing. Traditionally, the function of the RF section has been low-noise amplification, channel select filtering, and up/downconversion of the baseband information to/from radio frequencies. RF sections are usually implemented via discrete components due to the high signal frequencies involved. In recent years, the complementary metal oxide semiconductor (CMOS) analog design community [3, 6, 7] has focused its interest on developing CMOS RF front ends. Designing CMOS RF sections is a challenging proposition as CMOS technology is inherently slow as compared to the power-hungry bipolar technology. However, a CMOS RF front end offers the undeniable advantage of large-scale integration with baseband processing and thus approaching the holy grail of a single-chip radio.

In recent years, schemes advocating the placement

1068-9605/98/0400-0147$15.00/0

of analog-to-digital converters (ADCs) at increasingly higher frequencies such as intermediate frequency (IF) sampling schemes [7] have appeared. An extreme example of this trend is the concept of *software radio* [4], where the sampling is done as close to the antenna as possible and the digital section is made programmable so that a class of modulation schemes can be implemented by the user. A good example of power-optimal placement of the analog-digital interface can be found in Ref. [5], where power consumption as a function of data precision, filter length, operating frequency, technology scaling, and the maturity of the fabrication process has been studied. This points to an increase in the complexity of the digital processing section. Given the extensive research being conducted in the area of multimedia wireless, whereby joint source and channel coding is being explored, it can be envisaged that the digital signal processing component of future transceivers will grow even more. The techniques presented in this paper are applicable to wireless transceivers where the predominant processing is done digitally.

Modern-day wireless communication systems are characterized by high bit rates over severely time-varying, band-limited channels. Robust transmission schemes require the implementation of complex signal processing algorithms while mobile wireless applications require low-power dissipation. These conflicting requirements make the design of these systems a challenging proposition. The traditional approach (see Fig. 1A) to realizing a concept in silicon consists of two major steps: algorithm design and VLSI implementation. The major concern in the algorithm design phase consists of meeting the algorithmic performance requirements such as SNR and bit error rate (BER). Constraints from the VLSI domain such as area, power dissipation, and throughput were addressed only after the parameters (and sometimes the structure) of the algorithm were well defined. It is now well recognized that such an approach results in a long transition time from algorithm design to a silicon prototype. Therefore, there has been a strong need to develop a unified design paradigm, whereby the design of signal processing/communications algorithms and VLSI can be addressed in a cohesive manner. Design methodologies and design tools based on such a paradigm will be necessary to realize complex VLSI systems for signal processing and communications.

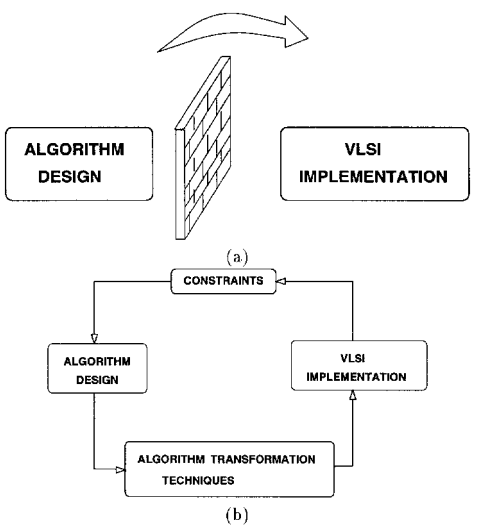


Fig. 1. VLSI systems design: (A) the traditional and (B) the modern approach.

The present design trend (see Fig. 1B) is to incorporate VLSI issues as constraints into the algorithm design phase. In particular, *algorithm transformation* techniques [8] were proposed as an intermediary step in the translation to VLSI hardware. These techniques were originally developed for high-throughput applications. However, they have found applications in low-power design as well [9]. Algorithm transformation techniques modify the algorithm structure and/or performance in order to introduce VLSI-friendly features. These techniques include retiming [10], look-ahead pipelining [11, 12], relaxed look-ahead [13], strength reduction [9, 14], block processing [11, 16], algebraic transformations [17], folding [18, 19] and unfolding [20, 21], which have been employed to design low-power and high-throughput DSP and communications systems.

This paper will review algorithm transformation techniques and indicate their application in the design of low-power VLSI systems for wireless systems. We refer to the existing algorithm transformations mentioned above as *static algorithm transformations* (SAT), because these are applied during the algorithm design phase assuming a worst-case scenario. In contrast, we have recently proposed *dynamic algorithm transformations* (DAT) [22], whereby the algorithm and architecture is dynamically adapted to the input nonstationarities so that the minimum possible functionality is activated at any given time. Power savings via this approach have proved to be substantially higher than applying SAT alone.

In Section 2 we present preliminaries regarding the data-flow graph (DFG) representation for DSP algorithms, power dissipation, and speed in CMOS VLSI

systems. The SAT techniques are described in Section 3, while Section 4 discusses the more recent DAT techniques.

2. PRELIMINARIES

In this section, we will review the DFG representation of DSP algorithms, and the basics of power dissipation and speed in the commonly used CMOS technology.

2.1. The Data-Flow Graph (DFG) Representation

A common representation of DSP algorithms is necessary in order to be able to apply the algorithm transformation techniques in a uniform manner. To do this, we will employ a DFG representation (see Fig. 2) of the algorithm. Figure 2 shows alternative representations of the DFG, whereby the graph on the left is a *weighted directed graph*, while the one on the right has a one-to-one correspondence with the actual hardware. The weighted directed DFG (referred to as a graph-theoretic DFG) is employed whenever the algorithm transformations are based upon traditional graph-theoretic results. Retiming [10] and folding [19] are examples of such transformations. In this paper, we will be employing both representations interchangeably.

The graph-theoretic DFG in Fig. 2 is a weighted directed graph $G = (V, E, d, w)$, where V is the set of graph nodes representing algorithmic operations, E is the set of directed edges/arcs connecting the nodes, w is the set of edge weights (equal to the number of delays on that edge), and d is the set of node delays. For example, the DFG on the left in Fig. 3 represents an FIR filter with a transfer function $H(z^{-1}) = c_0 z^{-1} + c_1 z^{-2}$. The corresponding graph-theoretic DFG is shown on the right in Fig. 3, where

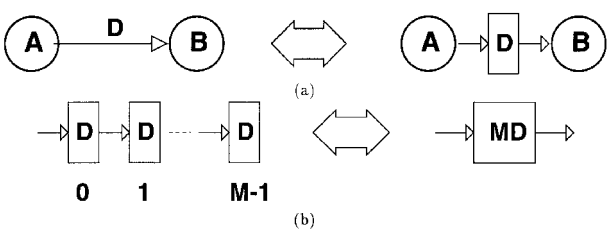


Fig. 2. Data-flow graph representation: (A) alternative DFG representations, and (B) the lumped delay model.

$$V = \{v_h, v_1, v_2, v_3\} \quad (1)$$

$$E = \{e_0, e_1, e_2, e_3, e_4\} \quad (2)$$

$$w(e_0) = w(e_2) = w(e_3) = 0 \quad (3)$$

$$w(e_1) = 1, w(e_4) = 1 \quad (4)$$

$$d(v_h) = 0, d(v_1) = d(v_2) = d(v_3) = 1 \quad (5)$$

Note that a host node v_h with zero computational delay is defined. This node represents the input-output interface between the DSP algorithm and the external world. The delays of the DFG nodes represent the time required to produce one sample of the output. In this paper, we will employ positive real numbers to describe the delay without actually specifying the units. This is acceptable as our interest here is to compare two architectures (the original and the transformed one) designed with the same hardware/software library. The delay itself depends on the precision requirements of the algorithm and the components of the hardware/software library. Hence, without loss of generality, we will assume that a DFG (such as in Fig. 3) represents a finite-precision DSP algorithm.

We now define certain properties of the DFG such as paths, critical path, iteration period, and iteration period bound. A *path* p is a sequence of nodes and arcs denoted as $u \rightarrow v \rightarrow \cdots \rightarrow w$, where u is the *source* node and w is the *destination* node of the path. Therefore, a path $u \rightarrow v$ is an arc with u as the source and v as the destination node. The *iteration period* (*IP*) (or the *sample period*) of the DFG is given by:

$$IP = \max_{\forall p \in \text{aDFG}} d(p) \quad (6)$$

where aDFG is an *acyclic* version of the original DFG obtained from it by removing arcs with nonzero delays. Thus, the iteration period of the DFG in Fig. 3 is 30 time units. The *critical path* of a DFG is a path p such that $d(p) = IP$. The goal of most algorithm transformation techniques is to reduce the delay of the critical path. The

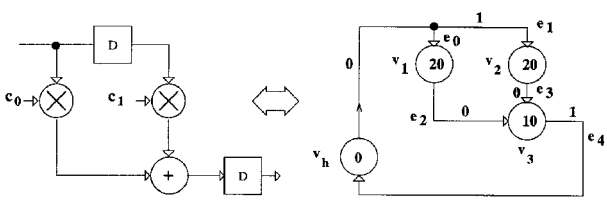


Fig. 3. Example of a DFG representation of an FIR filter.

iteration period bound (*IPB*) [8] for a DFG is defined as follows:

$$IPB = \max_{\forall L} \frac{\sum_{v \in L} d(v)}{\sum_{e \in L} w(e)} \tag{7}$$

where L is a loop in the DFG, where a loop is defined as a path p whose source and destination nodes are identical. Note that IP can be altered via the application of various algorithm transformation techniques. However, the IP will always be greater than or equal to the IPB.

Note that the DFG on the right in Fig. 3 describes the flow of data between the computational nodes of the algorithm. In other words, the number of DFG vertices represent the number of operations that need to be performed in order to produce one output sample. We refer to this DFG as an *algorithmic DFG* (or simply a DFG). In contrast, a DFG that represents the actual hardware blocks and the interconnection between them is referred to as the *hardware DFG* (HDFG). For example, if all the nodes of the DFG on the right in Fig. 3 are mapped to a unique hardware block, then the corresponding HDFG would be identical to that on the left in Fig. 3. The HDFG is obtained by mapping the nodes of an algorithmic DFG onto the hardware elements via the processes of *resource binding* and *scheduling* [37]. For a given DFG, numerous HDFGs exist. The speed of an HDFG is specified in terms of the *clock period* T_{clk}, which may or may not equal the iteration period IP of the corresponding DFG.

2.2. Power Dissipation

We will be considering CMOS technology and in particular we will consider the static circuit design style. Both the technology and the design style are by far the most popular. The static design style is shown on the left in Fig. 4 for a CMOS inverter. The PMOS transistor is ON when the input voltage V_{in} is at $0V$, while the NMOS transistor is ON when $V_{in} = V_{dd}$, the supply voltage. The output node is connected to $V_{dd}(0V)$ when the PMOS (NMOS) transistor is ON, thus accomplishing the inverter functionality. The load capacitance C_L in Fig. 4 plays a critical role in determining the power dissipation and the speed of the logic gate. In any case, the inverter behavior can be abstracted as shown on the right in Fig. 4, where the PMOS and NMOS transistors

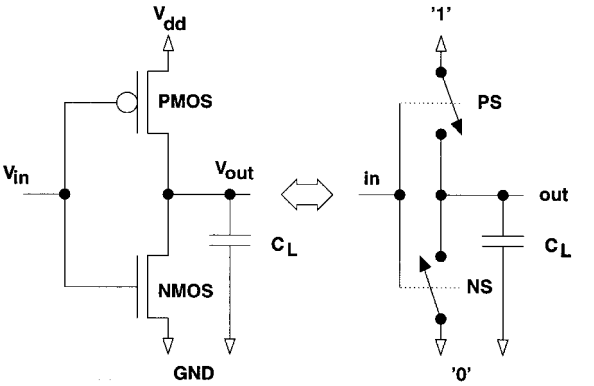


Fig. 4. Power dissipation in CMOS circuits.

are represented as a p-switch (PS) and an n-switch (NS), respectively. Other logic gates can be similarly designed by generalizing the switches to PMOS and NMOS networks.

The CMOS inverter in Fig. 4 has many different components of power dissipation. Of these, the dynamic power dissipation [23] P_D is the predominant component in digital CMOS VLSI circuits, accounting for more than 90% of the total power dissipation. This component occurs due to the cyclical charging and discharging of the load capacitance C_L. The average dynamic P_D power dissipation for the inverter is given by

$$P_D = P_{0\to1} C_L V_{dd}^2 f_{clk} \tag{8}$$

where $P_{0\to1}$ is the average probability of a $0 \to 1$ transition at the output, C_L is the load capacitance, V_{dd} is the supply voltage, and f_{clk} is the frequency of operation. Existing power reduction techniques [24] involve reducing one or more of the four quantities $P_{0\to1}$, C_L, V_{dd}, and f_{clk}. For example, complexity-reducing algorithm transformations such as strength reduction (see Section 3.8) reduce C_L by eliminating redundant arithmetic operations. On the other hand, low-voltage technologies reduce V_{dd}. An aggregate of Eq. (8) over all switching nodes in a VLSI chip will provide the total chip power dissipation, a task that is nontrivial and is being actively researched in the area of power estimation [25].

2.3. Speed

The speed of the inverter in Fig. 4 depends upon the rate at which the load capacitance C_L can be charged and discharged. In fact, the delay of this inverter t_{inv} [23] can

be approximated as:

$$t_{inv} = \frac{C_L}{V_{dd}} \left(\frac{L_n}{k_n W_n} + \frac{L_p}{k_p W_p} \right) \tag{9}$$

where L_p and W_p are the length and width of the PMOS transistor, L_n and W_n are the length and width of the NMOS transistor, and k_n and k_p are process parameters not under the designer's control. From Eqs. (8) and (9), it is clear that reducing C_L benefits both speed and power metrics. Unfortunately, a major component of C_L is proportional to the areas ($W_n L_n + W_p L_p$) of the transistors in the following stage (assumed to be identical to the current stage). Usually, the transistor lengths L_n and L_p are kept at the minimum allowable by the technology. Hence, the widths need to be minimized also. However, this will increase the delay t_{inv} as can be seen from Eq. (9).

Equations (8) and (9) bring out the well-known trade-off between power and speed as the supply voltage V_{dd} is scaled down. In recent years, V_{dd} scaling [24] (to reduce power as shown in Eq. (8)) accompanied by throughput-enhancing algorithm transformations (to compensate for loss in speed as indicated by Eq. (9)) such as pipelining [8, 12, 13] and parallel processing [8, 11] have been proposed as an effective low-power technique for DSP applications.

3. ALGORITHM TRANSFORMATION TECHNIQUES

In this section we will describe static algorithm transformation techniques that modify the properties of a given algorithm so as to enable a VLSI implementation that meets the constraints on power, area, and speed. It will be seen that most of these techniques transform the *structural* properties of the algorithm without altering the *functional* properties. An exception to this rule is the relaxed look-ahead technique [13] for pipelining of adaptive digital filters.

3.1. Retiming

Retiming is a transformation by which delays in a DFG are transferred from one arc to another without impacting the output sequence. An example of retiming is shown in Fig. 5, where one delay from the output of node B is transferred to both of its inputs. This transfer of delay is transparent to the host node (not shown) or to the external world. Retiming is typically employed to reduce the iteration period IP, reduce power, and improve scheduling of the DFG.

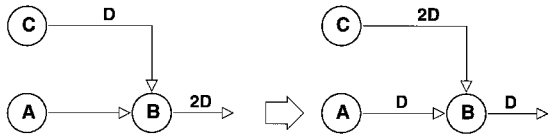


Fig. 5. Retiming.

To see how the iteration period IP of a DFG is reduced, consider the DFG on the left in Fig. 6. Assume that a multiply computation time (or delay) $T_m = 20$ and an add time $T_a = 10$. The original system has an $IP = 50$. Transferring two latches from the output, we obtain the retimed DFG as shown on the right in Fig. 6. This DFG has an $IP = 20$, which is an improvement by more than a factor of 2.

The examples in Figs. 5 and 6 illustrate manual retiming. While this may be possible for simple DFGs, it becomes very hard if the DFG is complex and irregular. Hence, a formal definition of retiming [10] is needed, whereby each node u in the DFG is assigned an integer-valued retiming label $r(u)$. The host node v_h always has $r(v_h) = 0$. It can be shown that [10] a valid retiming exists if the retiming values $r(u)$ for all nodes $u \in V$ can be determined such that for every edge $e \in E$, the expression

$$w_r(e) = w(e) + r(v) - r(u) \tag{10}$$

is nonnegative, where e is an arc with u and v as the source and destination nodes, respectively. In that case, replacing the number of delays on edge $e(w(e))$ by $w_r(e)$ will not alter the functionality of the DFG. For example, consider the retiming of the FIR filter DFG in Fig. 7A, which has an $IP = 30$. The retimed DFG in Fig. 7B can be obtained by assigning the retiming values $r(v_h) = 0$, $r(v_1) = 0$, $r(v_2) = 0$ and $r(v_3) = 1$. This retimed DFG has an $IP = 20$ as can be seen from Fig. 7B.

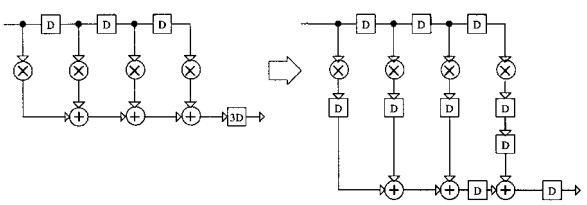


Fig. 6. An ad-hoc retiming example.

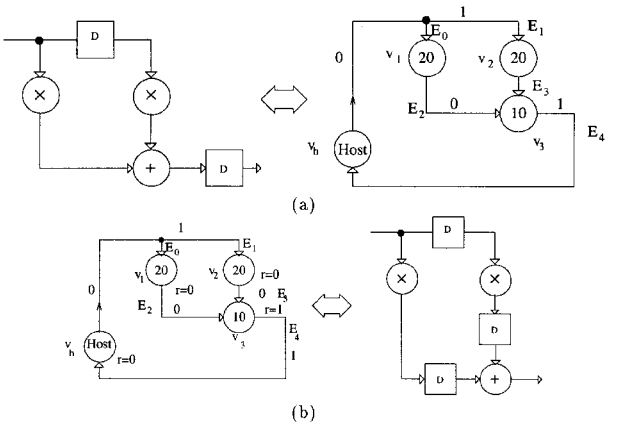


Fig. 7. A systematic retiming example: (A) the original DFG and (B) the retimed DFG.

The examples described so far illustrate the use of retiming for reducing the *IP* of the DFG. As mentioned before, retiming can also reduce the power dissipation by reducing the average transition probability $P_{0\to1}$ in Eq. (8). This can be accomplished by retiming the delays D to equalize the path delays $d(p)$ into a node so that the probability of glitches at the node output is reduced. In recent years, equivalence between retiming and scheduling [26] and retiming and clock skew optimization have been derived [27], thus indicating the versatility of this transformation.

While retiming can transfer existing delays to provide the benefits of reduced *IP* and lower power, it cannot create additional delays. Having additional delays in a DFG provides flexibility from a VLSI implementation perspective. The pipelining technique, described in the next subsection, is able to create additional delays, which can then be exploited by retiming.

3.2. Pipelining

Pipelining is an architectural technique for enhancing the throughput of an algorithm. Conceptually, it involves placing pipelining delays at appropriate arcs of the DFG such that the *IPB* of the DFG is reduced. These delays result in the DFG being a cascade of pipelining stages with each stage operating concurrently. In recent years, pipelined architectures have found application in low-power design as well due to the inherent power-delay trade-off described in Section 2.3. The utility of pipelined algorithms in low-power design is described next.

Let $C_L V_{dd}^2 f$ be the dynamic power dissipation of a serial architecture. Then the power dissipation of an L-level pipelined architecture (with each stage operating at L times the speed of the serial architecture) is given by

$$P_{pipe} = C_L V_{dd}^2 L f \tag{11}$$

where Lf is the L-fold increase in throughput due to pipelining. This increased throughput can be traded off with power by scaling V_{dd} and the operating frequency Lf by a factor of L. This would result in the throughput (see Section 2.3) of the pipelined and the serial architectures to be identical. The resulting power dissipation for the pipelined architecture is given by:

$$P_{pipe} = \frac{C_L V_{dd}^2 f}{L^2} \tag{12}$$

which is L^2 times lower than that of the serial architecture. Note that we have ignored the slight increase in C_L in Eqs. (11) and (12) which would arise due to the introduction of pipelining latches.

Pipelining of nonrecursive DFGs is quite straightforward and can be accomplished via the *feedforward cutset* pipelining technique [28, 29].

Definition 1: A cutset of a DFG is a set of arcs in the DFG (not including the arcs emanating from the host node v_h) that when removed results in two disjoint graphs.

For example, arcs E_2 and E_3 in Fig. 7A constitute a cutset of the DFG. Consider a line drawn through all the elements of a cutset with its two ends extending all the way to infinity. Such a line partitions the DFG into two sub-DFGs.

Definition 2: A feedforward (FF) cutset is that cutset in which the source nodes of all the cutset elements lie in the same sub-DFG and the destination nodes all lie in the other sub-DFG.

With this definition, it can be seen that the cutset (E_2, E_3) is also an FF cutset of the DFG in Fig. 7A.

The FF cutset pipelining technique first identifies an FF cutset of the DFG and then places M delays in each element of the cutset. This is illustrated in case of an FIR filter shown in Fig. 8A, where three FF cutsets are shown via dashed lines. We obtain an $IP = 40$ assuming $T_m = 20$ and $T_a = 10$. By placing $M = 1$ delays at the vertical FF cutsets, we obtain the pipelined architecture on the left in Fig. 8B. This architecture has an $IP = 30$. Similarly, by placing $M = 1$ delays at the horizontal cutset (see Fig. 8A),

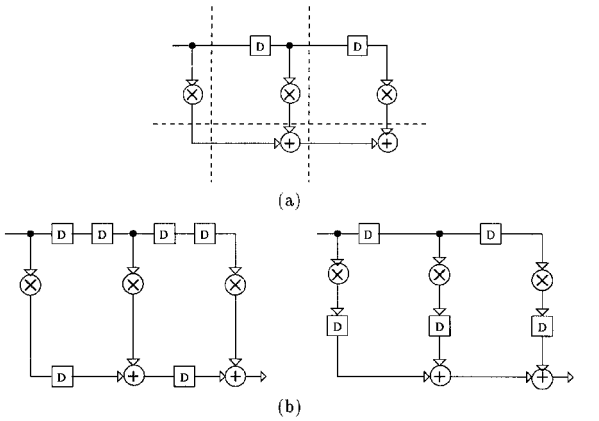


Fig. 8. Feedforward cutset pipelining: (A) identification of feedforward cutsets and (B) pipelining via delay placement at the cutsets.

we obtain the pipelined DFG on the right in Fig. 8B. This DFG has an $IP = 20$, which is half of that of the original DFG in Fig. 8A.

By employing values of $M > 1$, we obtain multiple delays at the cutsets, which can then be retimed. However, introducing M delays at a cutset increases the *sample latency* of the DFG, where sample latency is defined as the delay between the input and the output in terms of the number of sample periods. For most signal processing and communications this increase in latency is usually not critical.

In this subsection, we introduced the FF cutset pipelining technique and discussed its application to DFGs that do not have any feedback loops. This technique is not applicable to DFGs that have feedback loops as would be the case for IIR and adaptive filters. The next two subsections will present pipelining techniques for recursive DFGs.

3.3. Look-Ahead Pipelining

The previous subsection addressed the issue of pipelining nonrecursive structures via retiming and FF cutset pipelining. While retiming requires the presence of delays (so that they can be transferred), FF cutset pipelining provides these delays for nonrecursive structures. However, in case of recursive DFGs, cutset transformation is not applicable as no FF cutsets exist in such DFGs. Deliberately applying this transformation to any cutset (not just FF cutsets) will alter the functionality of the DFG. Hence, pipelining of recursive DFGs is nontrivial and the *look-ahead transformation* technique has been proposed [11] in order to get around this problem.

Consider the first-order recursion in Fig. 9, where computational delays are assumed to be $T_m = 20$, $T_a = 10$, and the equation describing the system is given by

$$x(n) = ax(n-1) + u(n) \tag{13}$$

The computation time of (13) is lower bounded by a single multiply and add time and hence the $IP = 30$ time units. From the definition of the IP bound (see Eq. 7), we find that the $IPB = 30$. Therefore, no implementation can achieve an IP smaller than 30 time units. This is a throughput bottleneck that can be broken by the application of the *look-ahead pipelining* technique. For the simple first-order recursion shown above, an M-step look-ahead pipelining reduces to expressing $w(n)$ as a function of $w(n-M)$ as shown below:

$$x(n) = a^M x(n-M) + \sum_{i=0}^{M-1} a^i u(n-i) \tag{14}$$

This transformation introduces M latches into the recursive loop, which can be retimed [10] to attain M-level pipelining of the multiply and add operations. This implies an M-level speed-up assuming that the composite of the multiply-add computation is pipelined uniformly. Note that this transformation has not altered the input-output behavior. This invariance with respect to the input-output behavior has been achieved at the expense of the *look-ahead overhead* term (the second term in Eq. 14), which is of the order of NM (N is the filter order) and can be expensive for large values of M. A look-ahead pipelined fourth-order IIR filter operating at 85 MHz has been implemented in VLSI [30], demonstrating the practical utility of this technique.

Note that the look-ahead overhead is a nonrecursive structure and hence can be pipelined easily via the techniques of FF cutset pipelining [28] and retiming [10] described in Sections 3.2 and 3.1, respectively. This is shown by the diagram at the top of Fig. 10 where an $M =$

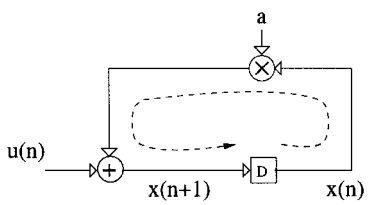


Fig. 9. A first-order fixed-coefficient recursive section.

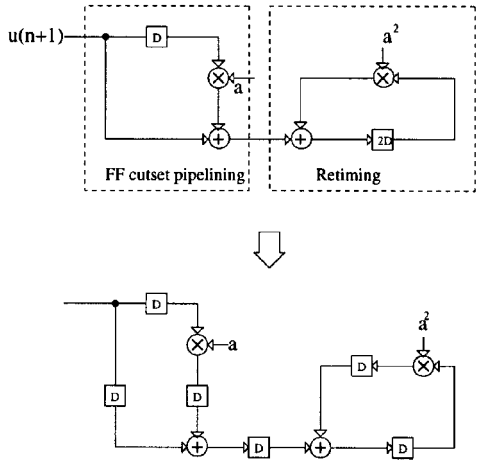


Fig. 10. A look-ahead pipelined first-order recursive section.

2 step look-ahead pipelined architecture is obtained from the architecture in Fig. 9. A horizontal cutset is applied to the FIR section and the two delays are retimed in the IIR section to obtain the architecture at the bottom of Fig. 10. This architecture has a critical path delay of $IP = 20$, while the $IPB = 15$. If the delays are placed such that all pipelining stages have identical delay, then the $IP = IPB = 15$ time units. This is possible to do in an application-specific integrated circuit implementation of the algorithm.

For IIR filters of order greater than unity, there are two types of look-ahead transformations: *clustered* and *scattered look-ahead*. In a serial (or unpipelined) recursive digital filter, the current state $w(n)$ is computed as a function of past states $w(n-1)$, $w(n-2)$, ... , $w(n-N)$, and present and past values of input $u(n)$. In other words,

$$\begin{aligned} x(n) = f_{serial}(&x(n-1), x(n-2), \ldots, \\ &x(n-N), u(n), u(n-1), \ldots, \\ &u(n-P)) \end{aligned} \tag{15}$$

where N is the order of the filter, and $f_{serial}(.)$ is a linear function. On the other hand, an M-step clustered look-ahead pipelined filter can be described as

$$\begin{aligned} x(n) = f_{c,pipe}(&x(n-M), x(n-M-1), \ldots, \\ &x(n-M-N+1), u(n), u(n-1), \ldots, \\ &u(n-M+1)) \end{aligned} \tag{16}$$

where it can be seen that the present state $x(n)$ is computed as a function of a *cluster* of N states that are M sample periods in the past. The hardware overhead due to clustered look-ahead is $O(M)$ as indicated by Eq. (16).

In a scattered look-ahead pipelined filter, the current state is computed as

$$\begin{aligned} x(n) = f_{s,pipe}(&x(n-M), x(n-2M), \ldots, \\ &x(n-NM), u(n), u(n-1), \ldots, \\ &u(n-NM-2)), \end{aligned} \tag{17}$$

where $f_{s,pipe}(.)$ is the scattered look-ahead function. The hardware overhead due to scattered look-ahead is $O(NM)$, which can be reduced via *decomposition* [11] to $O(N \log_2(M))$. Therefore, scattered look-ahead has a higher hardware overhead as compared to clustered look-ahead. However, a significant advantage of scattered look-ahead is that it preserves stability, while clustered look-ahead may not. Closed-form expressions for deriving the clustered and scattered look-ahead filter transfer functions can be found in [11]. Here we illustrate the two look-ahead techniques via an example.

Consider the serial digital filter described by the following equation:

$$x(n) = -0.5x(n-1) + 0.24x(n-2) + u(n) \tag{18}$$

Assuming $T_m = 20$ and $T_a = 10$, the filter in Eq. (18) has a critical path delay of 40 time units. This is also the value of IP and the IPB. Next, the application of clustered look-ahead results in the following equation:

$$\begin{aligned} x(n) = &-0.365x(n-3) + 0.1176x(n-4) + u(n) \\ &- 0.5u(n-1) - 0.49u(n-2), \end{aligned} \tag{19}$$

with the corresponding architecture shown in Fig. 11. Similarly, the scattered look-ahead pipelined architecture is shown in Fig. 12 and is described by the following

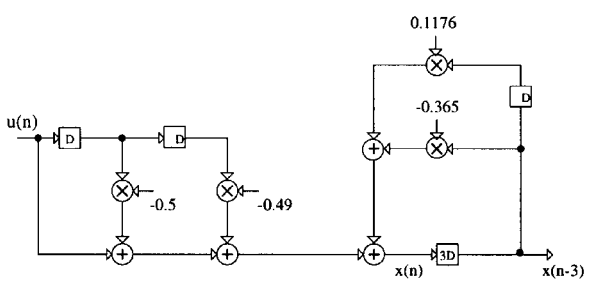


Fig. 11. A clustered look-ahead pipelined section.

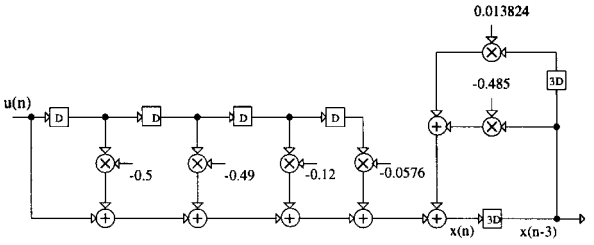


Fig. 12. A scattered look-ahead section.

equation:

$$x(n) = -0.485x(n-3) + 0.013824x(n-6) + u(n) - 0.5u(n-1) - 0.49u(n-2) - 0.12u(n-3) - 0.0576u(n-4) \quad (20)$$

Both architectures in Figs. 11 and 12 have an $IPB = 40/3$, which can be achieved via uniform pipelining.

The look-ahead pipelining technique has proved to be very effective for designing high-throughput low-power fixed coefficient filters. However, application of look-ahead to adaptive filters can result in large hardware overhead. In the next section we present a pipelining technique for adaptive filters, which is in fact based on an approximate form of the look-ahead technique presented in this section.

3.4. Relaxed Look-Ahead Pipelining

For adaptive filtering applications, a direct application of look-ahead techniques (described in the previous subsection) would result in a very high computational complexity. This was the motivation for developing the *relaxed look-ahead* technique [13]. The relaxed look-ahead pipelining technique allows very high sampling rates to be achieved with minimal hardware overhead. This technique involves the following two steps:

1. Application of look-ahead technique [11] to the serial algorithm
2. Approximating the functionality of various blocks in the look-ahead pipelined algorithm such that the impact on the overall convergence behavior is minimal

While Step 1 results in a unique pipelined algorithm, Step 2 permits various approximations (or relaxations) that result in a family of pipelined algorithms. However, due to the relaxations made in Step 2, the convergence behavior of the final pipelined algorithm will be different from that of the serial algorithm. Hence, unlike in all other algorithm transformations, convergence analysis is an integral part of the relaxed look-ahead technique. For the same reason it also represents a true joint optimization of algorithm design and VLSI. We now describe the relaxed look-ahead technique via an example. Consider a first-order time-varying recursive filter as described below:

$$x(n+1) = a(n)x(n) + b(n)u(n) \quad (21)$$

where $u(n)$ is the primary input as indicated in the architectural block diagram of Fig. 13. If $T_m = 20$ and $T_a = 10$, then $IP = IPB = 30$ time units. From Step 1 above, we first apply an M-step look-ahead to Eq. (21) to obtain,

$$x(n+M) = [\Pi_{i=0}^{M-1} a(n+i)]x(n) + \sum_{i=0}^{M-1} [\Pi_{j=1}^{i} a(n-M-j)] \cdot b(n+M-1-i)u(n+M-1-i) \quad (22)$$

It can be seen that the complexity of Eq. (22) is substantially higher than that of the serial architecture in Fig. 14A. This complexity increase is due to the "exact" nature of the look-ahead transformation. For adaptive filtering applications such an "exact" transformation is not needed as it is of more interest to maintain the average convergence behavior. Hence, we may approximate or relax this exactness via the help of various relaxations at the expense of slightly altered convergence behavior. We now describe three types of relaxations: *sum*, *product*, and *delay* relaxations, which have proved to be very useful for pipelining LMS-type adaptive filters.

If we assume that the input $a(n)$ is close to unity and that the product $b(n)u(n)$ does not change substantially over M cycles, then we can obtain the following relaxed

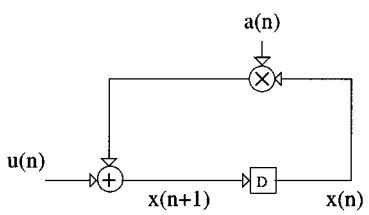


Fig. 13. A first-order time-varying filter.

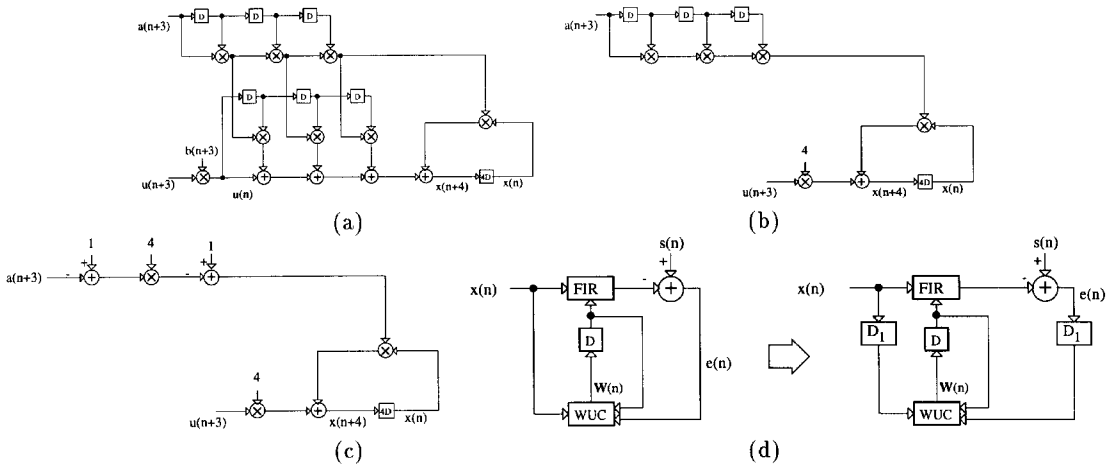


Fig. 14. Relaxed look-ahead pipelining: (A) look-ahead pipelining, (B) sum, (C) product, and (D) delay relaxations.

look-ahead pipelined algorithm:

$$x(n+M) = [\prod_{i=0}^{M-1} a(n+i)]x(n) + Mb(n+M-1)u(n+M-1) \quad (23)$$

The architecture corresponding to Eq. (23) is shown in Fig. 14B and the relaxation employed in obtaining Eq. (23) is called *sum relaxation*. Going a step further, we can approximate the product in the first term Eq. (23) as follows:

$$x(n+M) = [1 - M(1 - a(n+3))]x(n) + Mb(n+M-1)u(n+M-1) \quad (24)$$

which is a valid approximation if $a(n)$ is close to unity and slowly varying over M cycles. This relaxation is referred to as the *product relaxation*. Figure 14C shows a four-step relaxed look-ahead pipelined architecture that was obtained by the application of both sum and product relaxations. Finally, we show the *delay relaxation* in Fig. 14D. The block diagram on the left in Fig. 14D consists of an FIR filter whose coefficient vector $\mathbf{W}(n)$ is being updated by the weight-update block **WUD** every clock cycle as follows:

$$\mathbf{W}(n+1) = \mathbf{W}(n) + f(\mathbf{W}(n), x(n), e(n)) \quad (25)$$

where $f(\mathbf{W}(n), x(n), e(n))$ is a correction term, $x(n)$ is the input sample, and $e(n)$ is the error sample. Delay relaxation involves the modification of the correction term as follows:

$$\mathbf{W}(n+1) = \mathbf{W}(n) + f(\mathbf{W}(n), x(n-D_1), e(n-D_1)) \quad (26)$$

which is applicable only if the value of the $f(\mathbf{W}(n), x(n-D_1), e(n-D_1))$ is close to that of $f(\mathbf{W}(n), x(n), e(n))$. When applied to the least-mean-squared (LMS) [44] algorithm, the delay relaxation results in the "delayed LMS" [45].

In addition to the two relaxations presented above, other relaxations can be defined by approximating the algorithm obtained via application of look-ahead. The application of these relaxations, individually or in different combinations, results in a rich variety of architectures. However, these architectures will have different convergence properties and it is necessary to analyze their convergence behavior. We now apply relaxed look-ahead to the LMS algorithm as shown next.

Consider the serial LMS filter described by the following equations:

$$\mathbf{W}(n) = \mathbf{W}(n-1) + \mu e(n)\mathbf{X}(n);$$
$$e(n) = d(n) - \mathbf{W}^T(n-1)\mathbf{X}(n), \quad (27)$$

where $\mathbf{W}(n)$ is the weight vector, $\mathbf{X}(n)$ is the input vector, $e(n)$ is the adaptation error, μ is the step size, and $d(n)$ is the desired signal. The critical path delay for the serial LMS is given by

$$T_{c,serial} = 2T_m + (N+1)T_a \quad (28)$$

where N is equal to the number of taps in the filter block (or **F**-block) and we have assumed that the **WUD** block adders and the **F** block adders are single precision.

The relaxed look-ahead pipelined LMS architecture (see Refs. [13, 33] for details) is given by

$$\mathbf{W}(n) = \mathbf{W}(n - D_2) + \mu \sum_{i=0}^{LA-1} e(n - D_1 - i)\mathbf{X}(n - D_1 - i);$$

$$e(n) = d(n) - \mathbf{W}^T(n - D_2)\mathbf{X}(n) \tag{29}$$

where D_1 delays are introduced via the delay relaxation and D_2 delays are introduced via the sum relaxation. The D_1 and D_2 delays can be employed to pipeline the hardware operators in an actual implementation. In fact, the strategic location of D_1 and D_2 delays enables pipelining of all the arithmetic operations at a fine-grain level. Relaxed look-ahead pipelined filters have found practical applications in the design of a 100 MHz adaptive differential pulse code modulation (ADPCM) video codec chip [31], 51.84 Mb/s ATM-LAN [32] and broadband access transceiver chip sets.

The application of relaxed look-ahead requires a subsequent convergence analysis of the pipelined filter. For the sake of brevity, we will not describe this analysis in detail. It will suffice to mention that the bounds on the step-size μ become tighter and the adaptation accuracy degrades slightly as the ratio D_1/D_2 increases. The convergence speed is not altered substantially if the value of LA is chosen to be close to D_2. The interested reader is referred to Refs. [13, 33] for further details on the convergence analysis of the relaxed look-ahead pipelined LMS filter. As relaxed look-ahead results in a small hardware overhead, the increased throughput due to pipelining can be employed to meet the speed requirements, reduce power (in combination with power supply scaling [24] described earlier in Section 3.2), and reduce area (in combination with the folding transformation [19] to be described later in Section 3.7).

We conclude this section with an example illustrating the speed-up due to relaxed look-ahead pipelining. Consider the $N = 5$ tap serial architecture in Fig. 15A, which has a critical path delay (see Eq. (28)) of $T_{clk} = 200$, where we have assumed $T_m = 40$ and $T_a = 20$. For a speed-up of $M = 40$, the critical path delay of the pipelined system should be 5. This can be achieved with the relaxed look-ahead pipelined filter (see Fig. 15B) with $D_1 = 44$ and $D_2 = 4$ and where each adder is pipelined into four stages while each multiplier is pipelined into eight stages.

While pipelining is an attractive throughput-enhancing technique due to its low hardware overhead, the extent of pipelining can be limited by what is known as an input-output bound. This bound puts a limit on the maximum rate at which the data can be exchanged with an integrated circuit due to large parasitic capacitance on the pin leads. In that case, we may employ parallelization techniques along with pipelining to achieve throughputs that cannot be achieved by either one alone. In the next section, we present parallel architectures for digital filters.

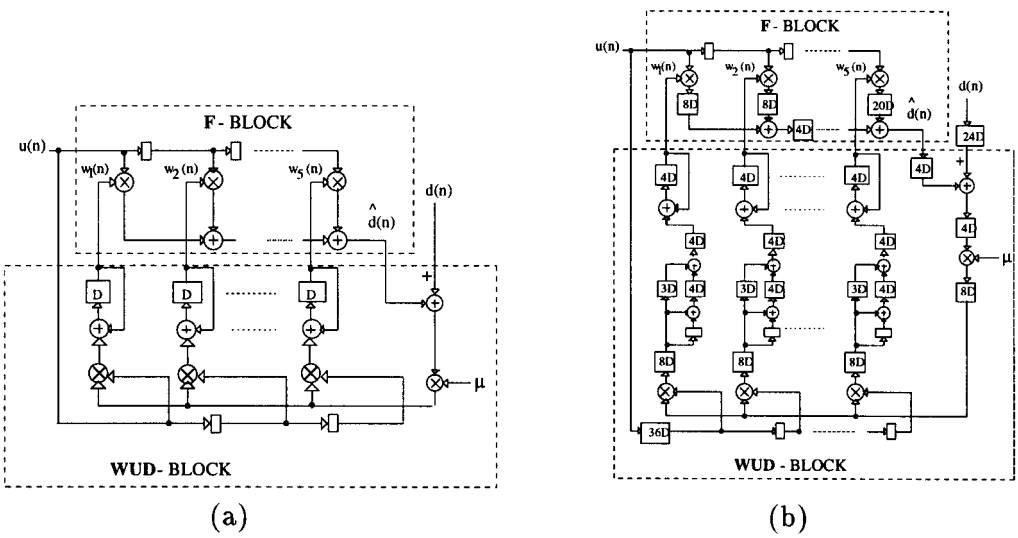


Fig. 15. Example: (A) serial architecture and (B) pipelined architecture with speed-up of 48.

3.5. Block/Parallel Processing

Pipelining achieves high throughput via the placement of pipelining latches so that the cascaded sections of the hardware can operate concurrently. In contrast, parallel/block processing involves replication of hardware in order to process a block of inputs in parallel. Thus, parallel architectures have an area penalty, which would be of concern. However, if the pipelined architecture is up against the input-output bound, then we can employ parallelization techniques to overcome it. Wireless receivers with IF sampling and software radio [4] architectures usually have a high sample rate signal processing front end. Such architectures can benefit from a combination of pipelining and parallel processing. As was the case in pipelined architectures, a power vs. throughput trade-off is also possible for parallel processing architectures, which can be seen in the following discussion.

Let $C_L V_{dd}^2 f$ be the dynamic power dissipation of a serial architecture. Then the power dissipation of an L-level block architecture (which each hardware instance operating at the same speed as the serial architecture) is given by

$$P_{par} = LC_L V_{dd}^2 f \tag{30}$$

where LC_L is the L-fold increase in switching capacitance due to hardware replication. The throughput of this parallel architecture is L times greater than that of the serial architecture and hence it is possible to scale V_{dd} and the operating frequency f by a factor of L so that the overall throughput of the parallel and the serial architectures are the same. This results in the following power dissipation for the parallel architecture:

$$P_{par} = \frac{C_L V_{dd}^2 f}{L^2} \tag{31}$$

which is L^2 times lower than that of the serial architecture. Both Eqs. (30) and (31) do not include the overhead capacitance due to serial-to-parallel converters, parallel-to-serial converters, and the routing overhead.

Systematic techniques for parallelizing serial digital filter architectures have been proposed in Refs. [15, 16, 11]. An L-level parallel architecture (see Fig. 16) has L outputs $y(kL)$, $y(kL+1)$, ... , $y(kL+L-1)$ that need to be computed. As shown in Fig. 16, the delays in a block architecture are L-slow, i.e., each clock tick will result in one block delay (or L sample delays). Deriving an L-level parallel architecture for an FIR filter is quite straightforward as shown in Fig. 17A, where an $L = 2$ level parallel architecture is shown for the following serial algorithm:

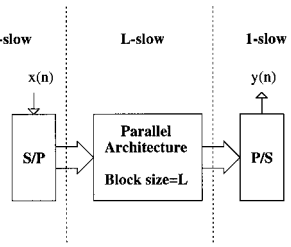


Fig. 16. A general parallel architecture indicating 1-slow and L-slow blocks, which operate at sample rate and $1/L^{th}$ of sample rate, respectively.

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) + a_3 x(n-3) \tag{32}$$

The architecture in Fig. 17A can be obtained by substituting $n = 2k$ (for even output samples) and $n = 2k + 1$ (for odd output samples) in Eq. (32). Note that hardware replication is clearly visible in Fig. 17A.

We now consider parallelizing IIR digital filters, which is nontrivial. This is because an L-level parallel IIR filter of order N (i.e., with N states) requires L outputs to be computed and N states to be updated in every L-slow clock cycle. Given the L-slow restriction on the delays and the fact that each of the state updates requires N past states (and inputs), there exists numerous ways in which the update can be done. A straightforward manner in which an IIR block filter can be realized is to recursively compute all the elements of the present block state vector in terms of the past block state vectors. This results in a *parallel direct form filter*. Consider the following recursive algorithm:

$$y(n) = ax(n) + by(n-1) \tag{33}$$

Parallelizing Eq. (33) by a level $L = 3$ requires the computation of the next block of outputs/states [$y(3k + 5)$, $y(3k + 4)$, $y(3k + 3)$] in terms of the current block [$y(3k + 2)$, $y(3k + 1)$, $y(3k)$]. Equation (33) indicates that the state $y(3k + 5)$ (in the previous block) can be computed from $y(3k)$ (in the current block). This is done by writing a three-step clustered look-ahead form of (33) as shown below:

$$y(n) = ax(n) + abx(n-1) + b^2 ax(n-2) + b^3 y(n-3) \tag{34}$$

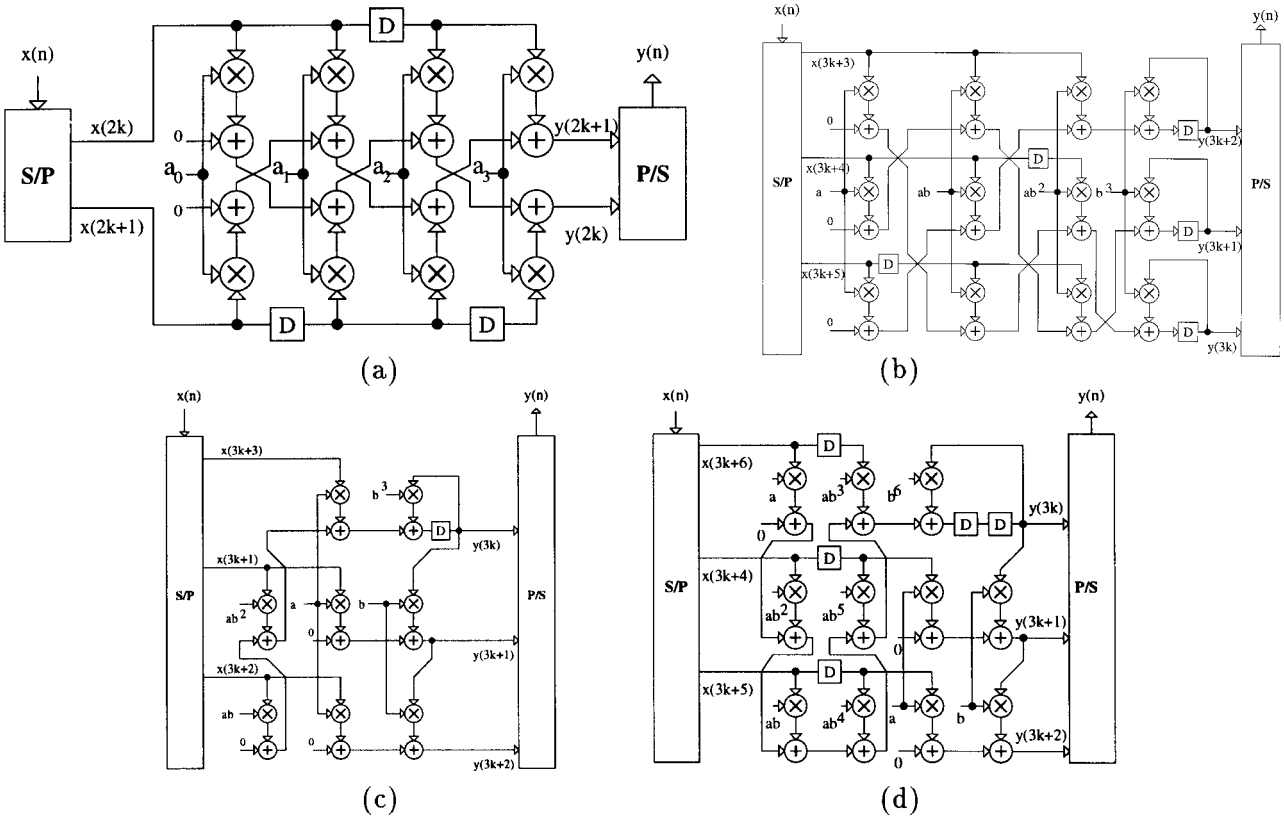


Fig. 17. Parallel architectures: (a) a parallel FIR filter with $L = 2$, $N = 3$, (b) a parallel direct-form block IIR filter with $L = 3$, $N = 1$, (c) an incremental parallel direct form IIR filter with $L = 3$, $N = 1$, and (d) a pipelined incremental parallel direct form IIR filter with $L = 3$, $N = 1$, and $M = 2$.

and then substituting $n = 3k + 5$, $3k + 4$, $3k + 2$ to obtain the architecture in Fig. 17B. Given that the complexity of an M-level pipelined clustered look-ahead filter is $(2N + M)$, it can be shown that the complexity of parallel direct form filters is $O(L^2)$.

This square dependence on the block-size L can be reduced to a linear dependence if only $min(L, N)$ states are computed recursively and the remaining $|L - N|$ states are computed nonrecursively or incrementally from the present states. This gives rise to the *incremental parallel direct form filter* shown in Fig. 17C, where we see that filter state $y(3k + 3)$ (from the next block) is updated via $y(3k)$ (of the current block), while state $y(3k + 1)$ is computed from $y(3k)$ and $y(3k + 2)$ is computed from $y(3k + 1)$, incrementally. Due to the $3 - slow$ delays, the $N = 1$ state is updated recursively while 2 states are updated nonrecursively/incrementally. The complexity of the parallel incremental direct form filter is linear in the block size for $L > N$.

One could combine the ideas of pipelining and block processing to come up with a *pipelined incremental parallel filter*, whereby a speedup of LM can be achieved by choosing a block size of L and a pipelining level of M. In the example being discussed, a structure with $L = 3$ and $M = 2$ can be derived (see Fig. 17D) by updating the state $y(3k + 6)$ in terms of state $y(3k)$. In addition, the states $y(3k + 1)$ and $y(3k + 2)$ are updated incrementally as in Fig. 17C. The complexity of such a filter is known to be linear in the block size L, and logarithmic in the pipelining level M.

We now consider parallel algorithms which have been proposed for adaptive filters [34–36]. The serial adaptive filter is described via Eq. (27), where a coefficient vector $\mathbf{W}(n)$ is updated by first calculating an error value $e(n)$ and then adding a correction term $\mu e(n)\mathbf{X}(n)$ to the current coefficient vector $\mathbf{W}(n - 1)$. The parallel algorithm in Ref. [34] adapts the coefficient vector $\mathbf{W}(n)$ once per block of output samples, i.e., the update is done

once in L samples. It has been shown [34] that if the step size of the parallel filter $\mu_B = L\mu$ (μ is the step size of the serial filter), then the parallel and serial filters have the same adaptation accuracy and convergence speed. In short, the parallel filter in Ref. [34] updates at a rate L times lower than the serial filter, but each update step is L times larger than that of the serial filter.

The parallel algorithm in Ref. [34] is suited for a stationary or a slowly varying (as compared to the block length) environment. In case of a nonstationary environment, there is a need to update the coefficients every sample period. This is achieved by the parallel algorithm in Ref. [36], which was applied to a decision feedback equalizer. The architecture [36] employs L parallel adaptive filters each operating on nonoverlapping blocks of data vectors and employing Eq. (27) to do so. Given the nature of Eq. (27), it is clear that the filters which operate on input blocks that are later in time will have to start with wrong initial conditions. The effects of wrong initial conditions are then corrected once the end of the data block has been reached. Note that this problem is not present in pipelined adaptive filters (described in the previous subsection) as these filters adapt at the sample rate and with the correct initial conditions.

In the next section we present a related algorithm transformation known as unfolding that enables high-throughput processing. While parallel and pipelined architectures reduce the *IP* and *IPB* for dedicated implementations, the unfolding technique is capable of reducing the *IP* down to the *IPB* (without altering the latter) for multiprocessor implementations.

3.6. Unfolding

The origins of the unfolding technique are in the compiler theory where it is also referred to as software pipelining. The unfolding technique [20, 21] was proposed in the 1980s as a method to match the widely differing sample rates in DSP systems. Unfolding is a powerful technique in the context of multiprocessor implementations of DSP algorithms, for generating schedules that have an *IP* equal to the *IPB* of the original DFG. For dedicated implementations, the unfolding technique has been employed [20, 21] to design digit-serial arithmetic architectures from bit-serial ones.

The unfolding technique [20, 21] accepts a DFG and generates another DFG by unfolding or exposing J iterations of the original DFG, where J is the *unfolding factor*. The unfolding algorithm has the following steps [21]:

1. For each node u in the original DFG, create J instances labeled as $u_0, u_1, \ldots, u_{J-1}$.
2. For each arc $u \rightarrow v$ in the original DFG with no delay (or zero weight), create arcs $u_q \rightarrow v_q$ for $q = 0, \ldots, J-1$.
3. For each arc $u \rightarrow v$ in the original DFG with i delays (or a weight of i), do step **3.1** if $i < j$; otherwise do step **3.2.**
   - **3.1.** Draw arcs $u_{J-i+q} \rightarrow v_q$ with one delay for $q = 0, \ldots, i-1$. Draw $u_{q-i} \rightarrow v_q$ with no delays for $q = i, \ldots, J-1$.
   - **3.2.** Draw arcs $u_{\lceil (i-q)/J \rceil J-i+q} \rightarrow v_q$ with $\lceil (i-q)/J \rceil$ delays for $q = 0, \ldots J-1$.

Step 1 creates J instances of each node, while Steps 2 and 3 specify the method to connect these nodes. For example, the DFG in Fig. 18A has an $IPB = (t_A + t_B)/3$ and it produces one sample per *IPB* assuming uniform pipelining. A $J = 2$ unfolded DFG (J is referred to as the *unfolding factor*) in Fig. 18B produces two samples in $IPB = (2t_A + 2t_B)/3$ time units. Therefore, unfolding does not increase the throughput for dedicated applications. However, the unfolded architecture in Fig. 18B exposes *interiteration* precedence, which reduces the *IP* of multiprocessor schedules. In fact, it can be shown that [8] unfolding by a factor J_{opt}, where J_{opt} is the least common multiple (LCM) of all the loop delays, results in multiprocessor schedules with $IP = IPB$. This can be easily checked for the example in Fig. 18A, where $J_{opt} = 3$. Unfolding this DFG with a factor of 3 will result in three decoupled loops all with one delay. DFGs with single delay loops are also called *perfect-rate* DFG because these can be scheduled with an $IP = IPB$. In this section, we will focus upon another application of unfolding, which is to systematically generate digit-serial arithmetic.

Consider an HDFG where the nodes represent bit-parallel operations executed by ripple-carry hardware operators, whereby one sample of word length B bits is

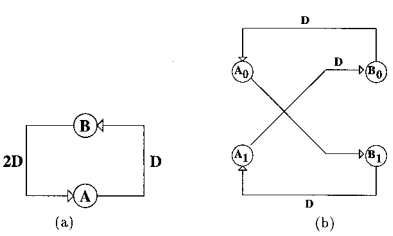


Fig. 18. Unfolding transformation: (A) the original DFG and (B) a 2-unfolded DFG.

processed in one clock cycle. Therefore, the clock period T_{clk} is a function of the number of bits being processed B and the time it takes to process 1 bit t_b plus some design margin plus overhead t_0 as indicated below:

$$T_{clk} = t_0 + Bt_b \tag{35}$$

From Eq. (35), it is clear that there is no fundamental reason why one cannot process $J < B$ bits per clock cycle with a clock period of

$$T_{clk} = t_0 + Jt_b \tag{36}$$

and take B/J (assuming J is a multiple of B) clock cycles to compute one sample of the output. In a similar fashion, the area of such an operator is given by

$$A = a_0 + Ja_b \tag{37}$$

where a_0 is the overhead term and a_b is the area consumed by 1 bit. Typical values of $t_0 = 8t_b$ and $a_0 = 4a_b$ have been observed [20] in practice. From Eqs. (36) and (37) it is clear that reducing J results in a reduction in area A and an increase in the clock rate. This type of computation is referred to as *digit-serial* computation with a digit size of J bits. When $J = 1$ and $J = B$, we obtain the well-known *bit-serial* and *bit-parallel* computations, respectively. From Eq. (36), the time taken to compute 1 B-bit output sample (i.e., the sample period T_s) is given by

$$T_s = \frac{B}{J}(t_0 + Jt_b) \tag{38}$$

Equation (38) indicates that the throughput (or sample rate) increases linearly with J for small values of J and from Eq. (37) we find that the area increases linearly with J. Hence, it can be shown [20] that the area-delay product AT_s is minimized when $J = \sqrt{a_0 t_0 / a_b t_b}$, which is approximately 5 to 6 bits for the typical values of a_0 and t_0 mentioned above. In any case, Eq. (38) implies that any digit-serial architecture will necessarily have a lower achievable sample rate than a bit-parallel architecture.

An example of unfolding to generate a $J = 2$ digit-serial architecture from a bit-serial architecture is shown in Fig. 19. If the precision requirements of the algorithm are more than 6 to 8 bits (which is typical) and one wishes to operate with an AT_s optimal digit size without losing throughput, then one can unfold the DFG itself. Say

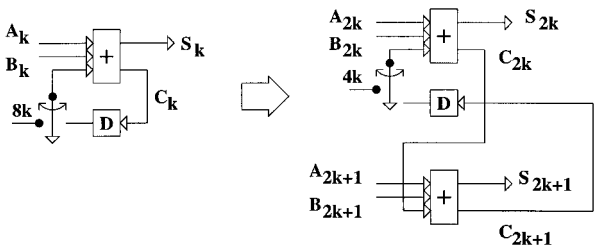


Fig. 19. Unfolding of a bit-serial adder.

$B = 12$ bits and the digit is equal to $J = 2$. Then we can gain the lost throughput due to digit serial processing by unfolding the DFG by a factor of 2 (as done in Fig. 18) so that *two digits* of *two consecutive outputs* are produced in *one clock cycle*. This means that two complete output samples will be generated at the end of two clock cycles or one complete output sample in one clock cycle.

Note that the folding technique to be described in the next subsection is in fact related to the unfolding technique and can be thought of as being the inverse of each other. There is one difference between the two: while unfolding results in a unique unfolded architecture, folding is a one-to-many mapping.

3.7. Folding

In systems where the sample period T_s is much larger than the computational delay of the hardware units, it is possible to map multiple algorithmic DFG nodes onto one HDFG node. In such cases, there exist many such mappings and one requires a systematic technique to synthesize the HDFG. This process of synthesizing an HDFG from an algorithmic DFG is known as *high-level synthesis* [37], and a comprehensive body of knowledge in this area has been developed since the early 1980s. In particular, systematic techniques for mapping regular algorithmic DFGs to *systolic array* architectures [38–40] have been developed. Commercial CAD tools are in the process of incorporating some of these techniques in recognition of the fact that design complexity is exploding and system-level design exploration (via high-level synthesis tools) has become more or less mandatory. The main reason for employing folding is to reduce area. However, the power dissipation of the resulting folded architecture depends on the manner in which the algorithmic operations are actually folded. This is because the average transition probability $P_{0\to1}$ in the folded architecture depends upon the input signal statistics and the folding scheme.

The examples in Fig. 20 illustrate the principle behind folding. In Fig. 20A, two identical operations (A) in the algorithmic DFG are computed with different inputs ((a, b) and (c, d)). It is, therefore, possible to map these two operations onto one hardware unit as shown on the right in Fig. 20A, assuming that the speed of the hardware unit permits this. Another example of folding is shown in Fig. 20B, which is applicable only to filtering operations done on independent data streams $x_1(n)$ and $x_2(n)$. In that case, the HDFG on the right in Fig. 20B implements the identical hardware but with all delays scaled up by a factor of 2. This creates additional delays that can be retimed so that the HDFG on the right in Fig. 20B can meet the throughput requirements. Such a scenario exists in the digital front end of a software radio located at a base station [4], where multiple receivers are located and identical processing is executed on independent data streams. It can be seen from Fig. 20 that folding always entails an interconnection overhead due to the presence of multiplexers and counters/control units. For DSP applications, this overhead is very small as the original DFG itself is very regular.

From the example above it can be seen that the folding transformation is related to high-level synthesis and hence we will briefly describe the key components of high-level synthesis. Any high-level synthesis algorithm consists of two major interdependent steps: *resource allocation/binding* and *scheduling* [41]. Resource allocation algorithms determine which of the algorithmic DFG nodes need to be mapped to a node in the HDFG. Scheduling algorithms determine the time step in which a particular algorithmic DFG node needs to be assigned to an HDFG. For example, in Fig. 20B, it was determined that all DFG nodes will be mapped to one HDFG node and that operations corresponding to $x_1(n)$ will be executed in time step 0 and those corresponding to $x_2(n)$ will be executed in time step 1. Clearly, these two steps are interrelated because if the resource allocation step assigns fewer hardware resources, then the scheduling algorithm will necessarily result in a longer schedule.

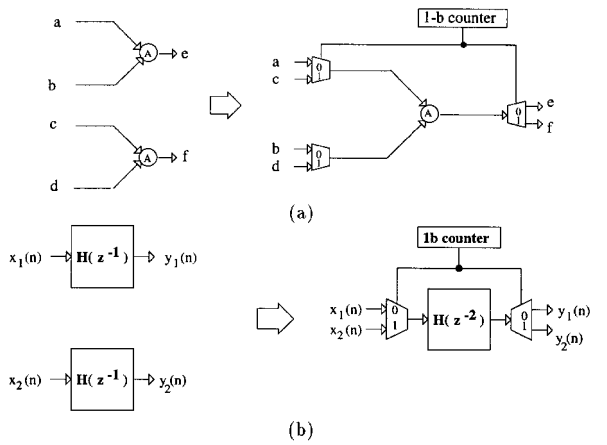


Fig. 20. Folding of independent data streams: (A) a general example and (B) a filter.

DSP algorithms are nonterminating programs (i.e., the input is a never-ending stream of data) with an IP equal to the sample period. Let K be the number of algorithmic operations mapped onto one hardware unit. In that case, we divide the sample period T_s into K time steps. The process of high-level synthesis for DSP algorithms involves assigning specific DFG operations to each time step (scheduling) and to a specific hardware unit (resource allocation). Once scheduling and resource allocation have been accomplished what remains is to synthesize the interconnection network between the hardware units and the control circuits as shown in Fig. 20. The *folding transformation* [19] accomplishes this in a systematic manner. The folding technique requires the specification of *folding sets*, which is a set $\mathcal{S}$ of ordered pairs (H_i, U_i), where H_i is the hardware unit and U_i is the time unit to which the algorithmic DFG node i has been mapped. The folding set is then the result of resource allocation and scheduling. Clearly, the cardinality of set $\mathcal{S}$ equals the number of nodes in the algorithmic DFG. For example, in Fig. 20B, the folding set is given by $\{(H_0, 0), (H_0, 1)\}$, where we have denoted the hardware unit by H_0 and the algorithmic operation by the inputs.

Consider algorithmic nodes u and v shown at the top of Fig. 21. Assume these to be distinct and that these need to be mapped to an HDFG that may have pipelined hardware modules. This is indicated by the dotted lines

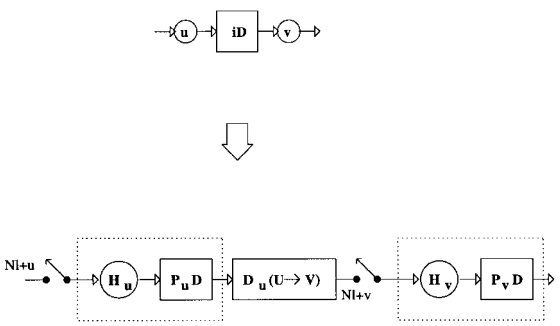


Fig. 21. The folding equation.

in the bottom part of Fig. 21, where $\mathbf{H}_u$ is the hardware unit and P_u is the number of delays employed to pipeline it. A similar description applies to $\mathbf{H}_v$. The pipelining delays P_u and P_v in Fig. 21 are not dependent on the algorithmic DFG but instead are a property of the library from which we expect to construct the hardware. Hence, the block diagram in the bottom of Fig. 21 operates as follows: In the l^{th} iteration of the algorithmic DFG and in time partition t_u (or the $Kl + t_u$ clock cycle), the hardware module $\mathbf{H}_u$ accepts the input of node u. The result of the computation of $\mathbf{H}_u$ appears after a delay of P_u clock cycles in cycle number $Kl + t_u + P_u$. This result is needed by $\mathbf{H}_v$ to compute the $K(l + i)^{th}$ iteration of the DFG in time partition t_v or clock cycle number $K(l + i) + t_v$. Clearly, the delays P_u and the time partition t_u should be such that the result computed by $\mathbf{H}_u$ is indeed available at the correct time partition and after i sample delays. This can be achieved by assigning the following value to the *folded arc delay* $D_F(u \rightarrow v)$,

$$\begin{aligned} D_F(u \rightarrow v) &= K(l + i) + t_v - (Kl + t_u + P_u) \\ &= Ki - P_u + t_v - t_u \end{aligned} \tag{39}$$

While t_u and t_v are obtained via scheduling, and P_u and P_v are library dependent, the designer can determine $D_F(u \rightarrow v)$ from Eq. (39) to satisfy this constraint. Note that it is entirely possible for Eq. (39) to result in a negative value for $D_F(u \rightarrow v)$, especially if the pipelining level P_u is high. However, this is not a problem because the timing relationship is not altered if qK delays are added to the right-hand side of Eq. (39). This is equivalent to adding an additional q sample delays to the algorithmic DFG shown at the top in Fig. 21. However, if this arc is present inside a loop, then one needs to remove qK folded delays from some other arc in the loop. This and another restriction that applies to arcs on parallel paths are sufficient to guarantee that all the folded arc delays are nonnegative and the folded architecture is a correct implementation of the algorithmic DFG.

Consider the example of folding a 4-tap FIR filter shown at the top in Fig. 22, where the operations enclosed within the dashed lines are mapped to different processors. The algorithmic DFG has four multiplications while the HDFG has two multipliers indicating that the folding factor $K = 2$ for this example. The HDFG is shown in the bottom of Fig. 22, where we see the unpipelined ($P_m = 0$ and $P_a = 0$) hardware multipliers and adders. In time partition 0, multiplication with coefficients a and b is executed while multiplication with c and d is executed in time partition 1. The overheads for this folding set are the four multiplexers, the 1 bit counter, the additional latch, and the switch. Note that the assumption here is that the unpipelined multipliers can operate at least twice the sample rate of the input. This depends on the technology and the arithmetic style.

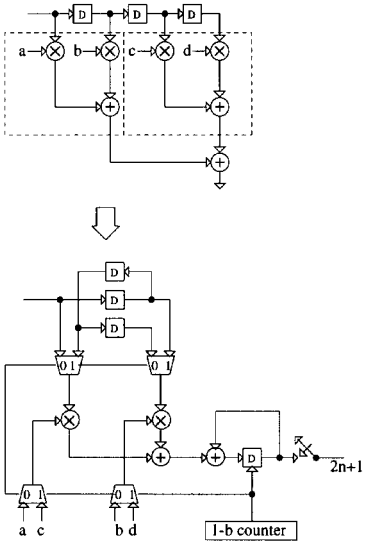


Fig. 22. Example of folding.

There is an interesting relationship between unfolding and folding. A DFG folded by a factor of K and then unfolded by a factor of J with $K = J$ will result in a final DFG that is a retimed and pipelined version of the original DFG. The pipelining effect arises from the fact that we are free to add multiples of K delays to Eq. (39). The retiming effect is due to the restrictions mentioned in the previous paragraph, which requires us to add/remove qK delays from arcs in a loop. Employing Eq. (39), it has been shown that retiming is a special case of scheduling [26]. These results can be extended to multiple implementation styles (different digit sizes) and multiple clocks, which the interested reader can find in Ref. [19].

3.8. Algebraic Transformations

Algebraic transformations [17] seek to exploit the fact that digital filtering is equivalent to polynomial multiplication in order to create inherent concurrencies in the DFG. In this subsection we will describe some of these transformations.

Two simple algebraic transformations are *asso-*

ciativity and *distributivity* , which can be employed to remove DFG nodes from recursive loops. For example, consider the first-order recursion,

$$y(n) = x(n) + ay(n-1) \tag{40}$$

Application of a two-step look-ahead [11] results in the following steps:

$$y(n) = x(n) + a[x(n-1) + ay(n-2)] \tag{41}$$
$$= x(n) + ax(n-1) + a[ay(n-2)] \tag{42}$$
$$= x(n) + ax(n-1) + (a^2)y(n-2) \tag{43}$$

where Eq. (42) is obtained via distributivity and Eq. (43) is obtained via associativity. Note that Eq. (43) can be obtained directly from the look-ahead (see Section 3.3) pipelining technique. Common subexpression elimination (CSE) is another algebraic transformation technique that can be employed to reduce the amount of hardware required to implement multiple-output function. As the name implies, the application of CSE involves identifying expressions that are common to different outputs and eliminating all instances except one. Common subexpression replication (CSR) brings about the opposite transformation as compared to CSE. The desired result in applying CSR is to reduce the critical path length of the DFG.

In this section we will present in detail an algebraic transformation technique referred to as *strength reduction* [9, 14], which has proved to be quite useful in many signal processing applications. Consider the problem of computing the product of two complex numbers $(a + jb)$ and $(c + jd)$ as shown below:

$$(a + jb)(c + jd) = (ac - bd) + j(ad + bc) \tag{44}$$

From Eq. (44) a direct-mapped architectural implementation would require a total of four real multiplications and two real additions to compute the complex product. However, it is possible to reduce this complexity via strength reduction [9, 14]. Application of strength reduction involves reformulating Eq. (44) as follows:

$$\begin{aligned}(a-b)d + a(c-d) &= ac - bd \\ (a-b)d + b(c+d) &= ad + bc\end{aligned} \tag{45}$$

As can be seen from Eq. (45), the number of real multiplications is three and the number of additions is five. Therefore, this form of strength reduction reduces the number of multipliers by one at the expense of three additional adders. Typically, multiplications are more expensive than additions and hence we achieve an overall savings in hardware.

The output of the **F** filtering block in an LMS algorithm (see Eq. (27)) can be written as

$$y(n) = \mathbf{W}^T(n-1)\mathbf{X}(n) \tag{46}$$

Clearly, if the input $\mathbf{X}(n)$ and the filter $\mathbf{W}(n)$ are complex quantities, then we can apply the strength reduction transformation (45) to the polynomial multiplication in Eq. (46) to obtain a low-power architecture. Such an architecture would be useful in communication systems employing two-dimensional modulation schemes such as quadrature amplitude modulation (QAM) and carrierless amplitude/phase (CAP) modulation [46]. These schemes employ a two-dimensional signal constallation, which can be represented as a complex signal. If a complex filter is to be implemented, then we can represent its output as a complex polynomial product. Furthermore, if the transformation in Eq. (45) is employed, then we would need only three real filters (instead of four as in Eq. (44)). Each real filter requires N multiplications and $N-1$ additions. Therefore, the application of the proposed transformation in Eq. (45) would then save a substantial amount of hardware.

Let the filter input be a complex signal $\tilde{\mathbf{X}}(n)$ defined as

$$\tilde{\mathbf{X}}(n) = \mathbf{X}_r(n) + j\mathbf{X}_i(n) \tag{47}$$

where $\mathbf{X}_r(n)$ and $\mathbf{X}_i(n)$ are the real and imaginary parts, respectively. Furthermore, if the filter is also complex, i.e., $\tilde{\mathbf{W}}(n) = \mathbf{C}(n) + j\mathbf{D}(n)$, then its output $\tilde{y}(n)$ can be obtained as follows:

$$\begin{aligned}\tilde{y}(n) &= \tilde{\mathbf{W}}^H(n-1)\tilde{\mathbf{X}}(n) \\ &= [\mathbf{C}^T(n-1) - j\mathbf{D}^T(n-1)][\mathbf{X}_r(n) + j\mathbf{X}_i(n)] \\ &= [\mathbf{C}^T(n-1)\mathbf{X}_r(n) + \mathbf{D}^T(n-1)\mathbf{X}_i(n)] \\ &\quad + j[\mathbf{C}^T(n-1)\mathbf{X}_i(n) - \mathbf{D}^T(n-1)\mathbf{X}_r(n)] \\ &= y_r(n) + jy_i(n)\end{aligned} \tag{48}$$

where $\tilde{\mathbf{W}}^H$ represents the Hermitian (transpose and complex conjugate) of the matrix $\tilde{\mathbf{W}}$. A direct implementation of Eq. (48) results in the traditional cross-coupled structure shown in Fig. 23A. This structure requires four FIR filters and two output adders, which amounts to $4N-2$ adders and $4N$ multipliers. If the channel impairments include severe ISI and/or multipath, which is the

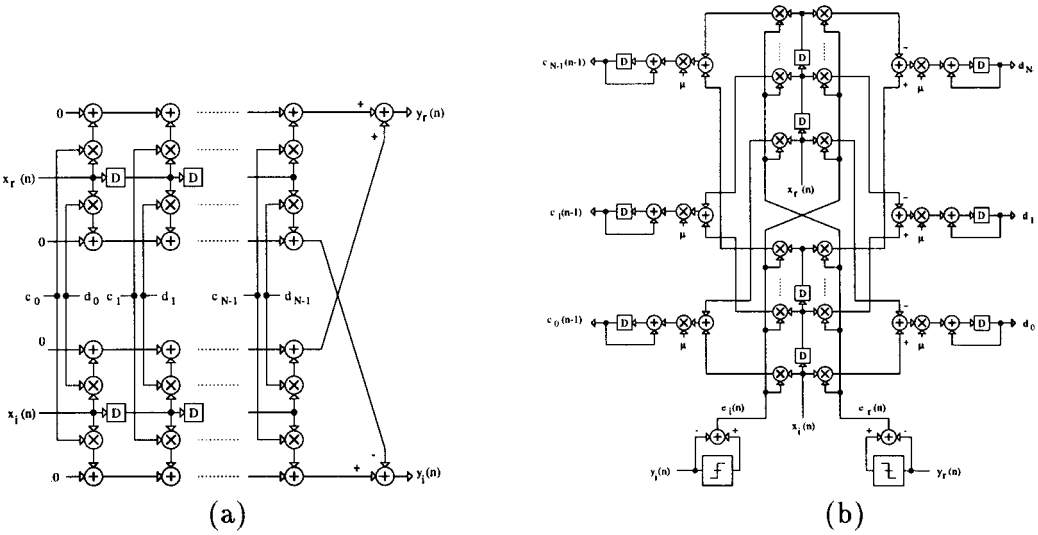


Fig. 23. The cross-coupled equalizer structure: (A) the **F** block and (B) the **WUD** block.

case in mobile wireless, then the number of taps necessary can be quite large, resulting in a high-complexity and high-power dissipation.

In the adaptive case, a weight-update block (or **WUD** block) would be needed to automatically compute the coefficients of the filter. This can be done by implementing a complex version of Eq. (27) as follows:

$$\tilde{\mathbf{W}}(n) = \tilde{\mathbf{W}}(n-1) + \mu\tilde{e}^{*}(n)\tilde{\mathbf{X}}(n) \tag{49}$$

where $\tilde{e}(n) = e_r(n) + je_i(n)$, $e_r(n) = Q[\,y_r(n)] - y_r(n)$, $e_i(n) = Q[\,y_i(n)] - y_i(n)$, $Q[.]$ is the output of the slicer, and $\tilde{e}^{*}$ represents the complex conjugate of $\tilde{e}$. Next, we substitute these definitions of $\tilde{\mathbf{W}}(n)$, $\tilde{e}(n)$ and $\tilde{\mathbf{X}}(n)$ into Eq. (49) to obtain the following two real update equations:

$$\mathbf{C}(n) = \mathbf{C}(n-1) + \mu[e_r(n)\mathbf{X}_r(n) + e_i(n)\mathbf{X}_i(n)] \tag{50}$$

$$\mathbf{D}(n) = \mathbf{D}(n-1) + \mu[e_r(n)\mathbf{X}_i(n) - e_i(n)\mathbf{X}_r(n)] \tag{51}$$

The **WUD**-block architecture for computing Eqs. (50)–(51) is shown in Fig. 23B. It is clear that the hardware requirements are $4N + 2$ adders and $4N$ multipliers for an N-tap two-dimensional filter.

Observing Eqs. (48)–(49) it is clear that strength reduction transformation (Eq. (45)) can be applied to the two complex multiplications present in them. We will see that this application of the transformation at the algorithmic level is much more effective in reducing power as opposed to an architectural-level application. Applying the proposed transformation to Eq. (48) first, we obtain

$$\begin{aligned}\tilde{y}(n) &= \tilde{\mathbf{W}}^{H}(n-1)\tilde{\mathbf{X}}(n) = \tilde{\mathbf{X}}^{T}(n)\tilde{\mathbf{W}}^{*}(n-1)\\ &= [\mathbf{X}_r^{T}(n) + j\mathbf{X}_i^{T}(n)][\mathbf{C}(n-1) - j\mathbf{D}(n-1)]\\ &= [\,y_1(n) + y_3(n)] + j[\,y_2(n) + y_3(n)]\end{aligned} \tag{52}$$

where

$$\begin{aligned}y_1(n) &= [\mathbf{C}^{T}(n-1) + \mathbf{D}^{T}(n-1)]\mathbf{X}_r(n)\\ &= \mathbf{C}_1^{T}(n-1)\mathbf{X}_r(n)\end{aligned} \tag{53}$$

$$\begin{aligned}y_2(n) &= [\mathbf{C}^{T}(n-1) - \mathbf{D}^{T}(n-1)]\mathbf{X}_i(n)\\ &= \mathbf{D}_1^{T}(n-1)\mathbf{X}_i(n)\end{aligned} \tag{54}$$

$$\begin{aligned}y_3(n) &= -\mathbf{D}^{T}(n-1)[\mathbf{X}_r(n) - \mathbf{X}_i(n)]\\ &= -\mathbf{D}^{T}(n-1)\mathbf{X}_1(n)\end{aligned} \tag{55}$$

where $\mathbf{X}_1(n) = \mathbf{X}_r(n) - \mathbf{X}_i(n)$, $\mathbf{C}_1(n) = \mathbf{C}(n) + \mathbf{D}(n)$, and $\mathbf{D}_1(n) = \mathbf{C}(n) - \mathbf{D}(n)$. The proposed architecture (see Fig. 24A) requires three filters and two output adders. This corresponds to $4N$ adders and $3N$ multipliers, which is approximately a 25% reduction in the hardware as compared with the traditional structure (see Fig. 23A). It, therefore, represents an attractive alternative from a VLSI perspective.

We now consider the adaptive version and specifically analyze the **WUD** block. From Eqs. (53)–(55) and Fig. 24A, it seems that an efficient architecture may result if $\mathbf{C}_1(n-1) = [\mathbf{C}(n-1) + \mathbf{D}(n-1)]$ and $\mathbf{D}_1(n-1) = [\mathbf{C}(n-1) - \mathbf{D}(n-1)]$ are adapted instead of $\mathbf{C}(n-1)$

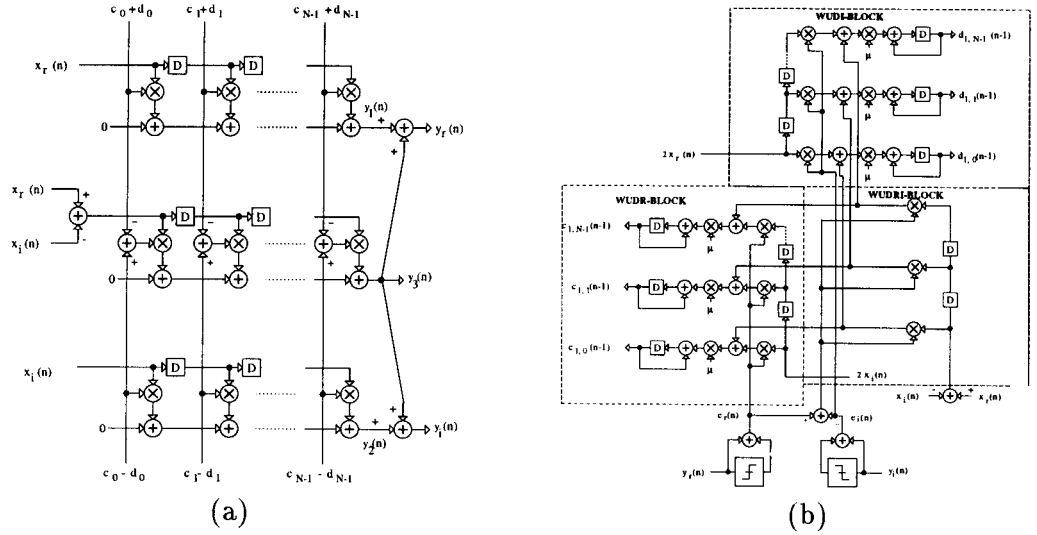


Fig. 24. The strength-reduced equalizer structure: (A) the **F** block and (B) the **WUD** block.

and $\mathbf{D}(n-1)$. In order to see if this is the case, we will derive the update equation for $\mathbf{C}_1(n-1)$ and $\mathbf{D}_1(n-1)$ next.

Adding Eq. (50) to Eq. (51), we obtain the update equation for $\mathbf{C}_1(n-1)$ as follows:

$$\mathbf{C}_1(n) = \mathbf{C}_1(n-1) + \mu[e_r(n)(\mathbf{X}_r(n) + \mathbf{X}_i(n)) - e_i(n)(\mathbf{X}_r(n) - \mathbf{X}_i(n))] \tag{56}$$

In a similar fashion, subtracting Eq. (51) from Eq. (50) provides us with the corresponding equation for updating $\mathbf{D}_1(n-1)]$ as follows:

$$\mathbf{D}_1(n) = \mathbf{D}_1(n-1) + \mu[e_r(n)(\mathbf{X}_r(n) - \mathbf{X}_i(n)) + e_i(n)(\mathbf{X}_r(n) + \mathbf{X}_i(n))] \tag{57}$$

It is now easy to show that Eqs. (56) and (57) can be written in the following complex form:

$$\tilde{\mathbf{W}}_1(n) = \tilde{\mathbf{W}}_1(n-1) + \mu\tilde{e}(n)[(\mathbf{X}_r(n) + \mathbf{X}_i(n)) + j(\mathbf{X}_r(n) - \mathbf{X}_i(n))] \tag{58}$$

where $\tilde{\mathbf{W}}_1(n) = \mathbf{C}_1(n) + j\mathbf{D}_1(n)$. We can now apply the strength reduction transformation to the complex product in Eq. (58) to obtain a low-power **WUD** architecture. Doing so results in the following set of equations, which describe the strength-reduced **WUD** block:

$$\tilde{\mathbf{W}}_1(n) = \tilde{\mathbf{W}}_1(n-1) + \mu[\mathbf{e}\mathbf{X}_1(n) + \mathbf{e}\mathbf{X}_3(n) + j(\mathbf{e}\mathbf{X}_2(n) + \mathbf{e}\mathbf{X}_3(n))] \tag{59}$$

where

$$\mathbf{e}\mathbf{X}_1(n) = 2e_r(n)\mathbf{X}_i(n) \tag{60}$$

$$\mathbf{e}\mathbf{X}_2(n) = 2e_i(n)\mathbf{X}_r(n) \tag{61}$$

$$\mathbf{e}\mathbf{X}_3(n) = [e_r(n) - e_i(n)][\mathbf{X}_r(n) - \mathbf{X}_i(n)] = e_1(n)\mathbf{X}_1(n) \tag{62}$$

where $e_1(n) = e_r(n) - e_i(n)$ and $\mathbf{X}_1(n) = \mathbf{X}_r(n) - \mathbf{X}_i(n)$. The architecture corresponding to Eq. (59) and Eqs. (60)–(62) is shown in Fig. 24B. It can be seen that this **WUD** architecture requires only $3N$ multipliers and $4N + 3$ adders. Thus, the number of multipliers is reduced by one fourth at the expense of an additional adder as compared to the traditional **WUD** architecture (see Fig. 23B).

Combining the architecture for the **F** block in Fig. 24A and that for the **WUD** block in Fig. 24B, we obtain the proposed strength-reduced low-power adaptive filter architecture in Fig. 25. A complete description of the low-power adaptive filter architecture is given by Eqs. (52–55) and (59–62). In Fig. 25, we show the overall block diagram of the adaptive filter, where **FR** block and **WUDR** block compute Eqs. (53) and (60), respectively. Similarly, **FI** block and **WUDI** block compute

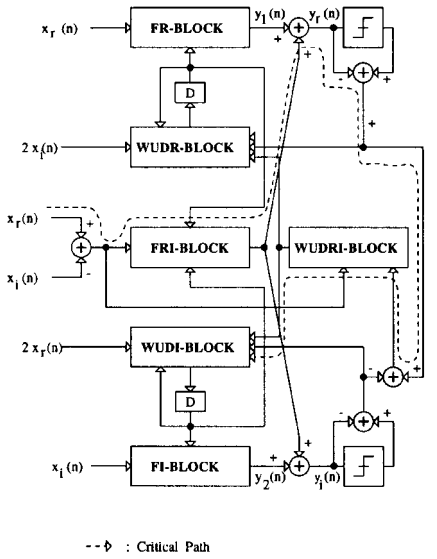


Fig. 25. The strength-reduced equalizer block diagram.

Eqs. (54) and (61), respectively. Furthermore, the **FRI** and **WUDRI** blocks compute Eqs. (55) and (62). Note that in Fig. 24A, we have separated the slicer and the error computation adders from the **WUDR** and **WUDI** blocks. This is done only to depict the error feedback loop clearly.

The performance of the strength-reduced low-power adaptive filter architecture has been studied [14] in a 51.84 Mb/s over 100 meters of unshielded twisted-pair (UTP3) ATM-LAN [46] employing a CAP-QAM modulation scheme. Finite-precision analysis of this structure has indicated that the **F** block in the strength-reduced structure requires at most 1 additional bit, while the **WUD** block requires 1 less bit than the traditional cross-coupled structure. This clearly indicates that the strength-reduced structure [14] should be the architecture of choice when implementing complex filters. Many wireless receivers employ two-dimensional modulation schemes, which require complex filtering in the baseband. In such cases, the proposed strength-reduced adaptive filter will have a direct application.

4. DYNAMIC ALGORITHM TRANSFORMATIONS

We refer to the algorithm transformations described in Section 3 as *static algorithm transformations* (SAT), because these are applied during the algorithm design phase assuming a worst-case scenario and their implementation is time invariant. Most real-life signal environments are nonstationary and hence significant power savings can be expected if the algorithm and architecture can be dynamically tailored to the input. This gives rise to the general concept of data-driven signal processing [42], where the algorithm workload [43] and the voltage supply are varied in real time to optimize the power dissipation.

In this section, we present *dynamic algorithm transformations* (DAT) [22] as another approach to data-driven signal processing, whereby the theoretical power-optimum signal processing architecture is first determined and then practical methods to realize this optimum are developed. Since adaptive filters [44] are inherently data-driven filters, it is quite natural to develop DAT techniques for these filters. We calculate the power-optimum adaptive filter configuration and then propose the DAT-based structure shown in Fig. 26 to approach this optimum. The system in Fig. 26 consists of two major blocks: the signal processing algorithm (**SPA**) block and the signal monitoring algorithm (**SMA**) block. The **SPA** block implements the main signal processing function, which would vary over time. The **SMA** block decides the instant and the extent of change to the **SPA** block so as to optimize a circuit performance measure such as power dissipation while maintaining the algorithm performance such as the mean-squared error.

Simulation results have been shown [22] that illustrate the performance of the proposed DAT-based filter when employed as a near-end cross-talk (NEXT) canceller in 155.52 Mb/s ATM-LAN [46] over category 3 wiring. These results indicate that the power savings for a NEXT canceller range from 21% to 62% as the cable length varies from 70 meters to 100 meters. For mobile wireless systems, the channel variation is substantial due to fading effects [48] and hence DAT-based receiver structures would be beneficial and quite challenging to design.

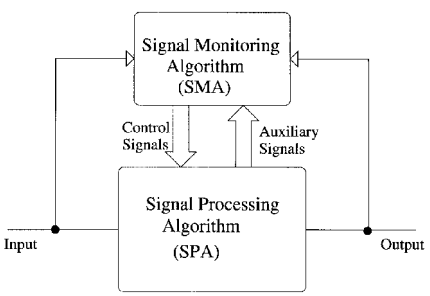


Fig. 26. Dynamic algorithm transformation (DAT): a general framework.

4.1. Hardware Models

In computing the optimum configuration, we will assume that in an N-tap adaptive filter, any filter tap can be powered up or down (and not just the trailing/leading taps [43, 49]). This feature can be algorithmically characterized by defining control signals, $\alpha_i \in \{0, 1\}$, $i = 1, \ldots N$, for each of the filter taps. Here, $\alpha_i = 0$ implies that the tap has been powered down and $\alpha_i = 1$ implies that it is not powered down. The power dissipation P_D for this adaptive filter can be obtained from Eq. (8) as follows:

$$P_D = \left(\sum_{i=1}^{N} \alpha_i \left(\sum_{j=1}^{M} P_{ij} C_j \right) + P_{oh} C_{oh} \right) V_{dd}^2 f_s \quad (63)$$

where M is the number of the hardware units in each tap, C_j is the average switching capacitance for j^{th} hardware unit in any tap, and C_{oh} is the overhead capacitance not considered in C_j's. Also, P_{ij} is the average probability $P_{0\to1}$ at the output of the j^{th} unit in the i^{th} tap, and P_{oh} is the average probability $P_{0\to1}$ for the overhead capacitance C_{oh}.

In order to simplify the problem and to come up with practical **SMA** strategies, we will assume throughout this paper that the input signal $x(n)$ is uncorrelated. We will see later that the **SMA** strategies resulting from this assumption are simple enough to be implemented and also result in substantial power savings in the general case where $x(n)$ is colored.

It can be shown that the power dissipation of a $B_x \times B_c$ bit multiplier, which multiplies a B_x bit uncorrelated input $x(n)$ with a B_c bit coefficient w_k, is given by

$$P_m = B_x \lceil log_2(|w_k|) \rceil C_b V_{dd}^2 f_s \quad (64)$$

where C_b is the switching capacitance of a primitive block of the array. Note that the term $B_x \lceil log_2(|w_k|) \rceil$ represents the number of primitive blocks in the multiplier that are needed to perform the multiplication.

4.2. Algorithm Performance

The mean square error (MSE) can be formulated for the given set of α_i's. The output error of the LMS adaptive filter (see Eq. (27)) can be written as

$$e(n) = d(n) - \sum_{i=1}^{N} \alpha_i w_i x(n - i + 1) \quad (65)$$

where w_i and $x_n - i + 1$) is the coefficient and input signal for i^{th} tap. For an uncorrelated/white input $x(n)$, it can be shown that [44] the minimum MSE (J_{min}) is given by

$$J_{min} = \sigma_d^2 - \sum_{i=1}^{N} \alpha_i |w_i|^2 r(0) \quad (66)$$

where σ_d^2 and $r(0)$ is the energy in the desired signal $d(n)$ and input signal $x(n)$, respectively.

4.3. Joint Optimization

From Eq. (66), we note that powering down taps with small values of w_k results in a small increase in J_{min}, which is desirable. However, from Eq. (64), we also see that a tap with a small value of w_k consumes lesser power as well and hence powering down such a tap will not provide substantial power savings. Clearly, the power-optimum configuration will be the one that powers down those taps which result in maximal power savings and at the same time result in a J_{min}, which is less than a desired, value J_o. This is formally stated as

$$\min_{\alpha_i,\, i \in \{1, \cdots, N\}} \sum_{i=0}^{N-1} \alpha_i \lceil log_2(|w_i|) \rceil$$

$$s.t. \sum_{i=0}^{N-1} \alpha_i |w_i|^2 r(0) > \sigma_d^2 - J_o \quad (67)$$

where $\alpha_i \in \{0, 1\}$ and J_o is the desired value of MSE dictated by the application. Note that Eq. (67) assumes that the multipliers in Eq. (27) are powered down after the adaptive filter has converged, an assumption that is usually true in practice. The optimization problem in Eq. (67) can be solved via standard mixed integer linear programming (ILP) approaches. In the next section we will present practical **SMA** strategies that approach the solution of Eq. (67).

If input statistics are ignored, then the objective function in Eq. (67) reduces to $\sum \alpha_i$. Minimization of $\sum \alpha_i$ is equivalent to powering down the maximum number of taps in the filter subject to the constraint in Eq. (67).

4.4. SMA Strategy 1

In this subsection we will present an algorithm for dynamically controlling the α_k's while maintaining $J(n) < J_o$ and reaching the $min\, P_D$ solution.

SMA Strategy 1

Step 1. Start with $\alpha_k = 1$, $\forall k$.

Step 2. Allow the adaptive filter to converge to the optimum solution. Check J_{min}, which is the converged value of the MSE, $E[e^2(n)]$.

Step 3. If $J_{min} < J_o$, go to Step **3.1;** otherwise, go to Step **3.2.**

3.1. Determine j such that $|w_{o,j}| = min\{|w_{o,k}|, \forall k : \alpha_k = 1\}$. Assign $\alpha_j = 0$ and go to Step **3.1.**

3.2. Determine j such that $|w_{o,j}| = max\{|w_{o,k}|, \forall k : \alpha_k = 0\}$. Assign $\alpha_j = 1$ and go to Step **3.2.**

Therefore, **SMA Strategy 1** *approaches* the power-optimum configuration (obtained as a solution to Eq. (67)) by assigning $\alpha_k = 0$ starting with coefficients with the lowest magnitude until the $J_{min} < J_o$. Equivalently, it minimizes $\sum \alpha_i$, thus *achieving* the solution of Eq. (67) if the input signal statistics are not accounted for. We should mention here that in the power-optimum configuration some of the internal taps may also be powered down, leading to nonuniformly spaced samples.

Other more sophisticated strategies are also possible, which result in a more complex **SMA** block but with increased power savings in the **SPA** block. Thus, there is a fundamental trade-off that can be explored between **SMA** and **SPA** block complexities such that the overall power dissipation is minimized.

4.5. Implementation of DAT-Based Adaptive Filter

We present architectural level implementation of the DAT-based adaptive filter derived in the last section. Figure 27 shows the **SPA** block of the DAT-based adaptive filter, where each tap is enclosed in a dashed box and is composed of two multiply-adds. The control signals, α_k's are employed to force a static value of 0 into one of the inputs of the filtering (**F** block) multipliers in the k^{th} tap if $\alpha_k = 0$. The signals β_k's in the weight-update (**WUD**) block equal zero if either the filter has converged or the tap is powered down ($\alpha_k = 0$). For array multipliers, if one of the inputs to the multipliers is zero, then the switching power consumption of the multiplier is close to zero. Thus, for $\alpha_k = 0$, the **F**-block multiplier in tap k is powered down. Similarly for $\beta_k = 0$, the **WUD**-block multiplier in tap k is powered down and the two inputs to the lower adder are constant. Therefore, the switching activity for this adder will also be zero. If needed, the latch in the weight update block can be powered down by disabling the clock.

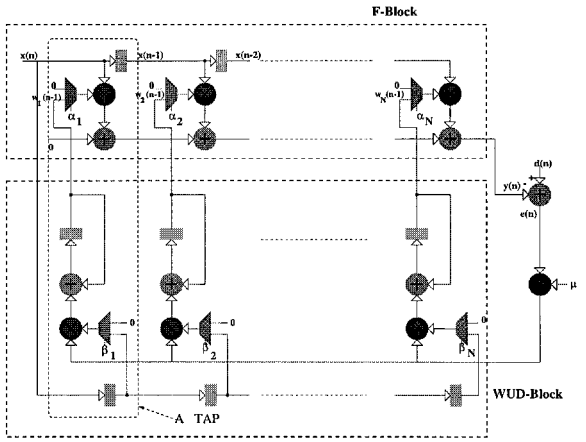


Fig. 27. The SPA architecture.

An abrupt power-down of a tap will cause the MSE to increase suddenly. This can cause a burst of errors in a practical application. It can be prevented if the taps are powered down gradually such as by lowering the initial coefficient $w_{o,k}$ to $w_{o,k}/2$ and then to $w_{o,k}/4$ for a few samples before powering down the tap completely. Another modification to reduce undesirable glitching is to employ a window rather than a single value in Step 3. This implies that any value of $J_{min} \in [J_o - \delta, J_o]$ $(\delta > 0)$ is considered acceptable. If $J_{min} < J_o - \delta$, then coefficients are powered down, and if $J_{min} > J_o$, then a tap is powered up.

Efforts are currently underway to implement DAT-based receivers for very high-speed digital subscriber loops and ATM-LAN applications. Wireless channels are particularly good candidates for DAT-based schemes due to the inherent variabilities in the medium.

5. CONCLUSIONS

In this paper we have presented various algorithm transformation techniques that can be employed to design low-power and high-speed algorithms for DSP and communications systems. These techniques are applicable to the digital processing section of wireless systems also. These transformations should be viewed as a bridge between the domains of algorithm and VLSI design. Transformations such as retiming [10], look-ahead pipelining [11], folding [19], unfolding [20, 21], and strength reduction [14] preserve the input-output behavior of the algorithm. However, transformations such as relaxed look-ahead [13] and dynamic algorithm [22] modify the algorithm performance to

obtain much superior power and speed advantages in the VLSI domain. Application of algorithm transformations requires a new breed of system designers who are conversant with both algorithmic and VLSI implementation considerations so that joint optimization between these two domains can be done.

While numerous algorithm transformations described in this paper can be applied individually, a systematic methodology that enables a coherent application of these transformations does not exist. Investigating this methodology is an important open problem given the increasing complexity of systems being realized on silicon. Development of such a methodology requires that an integrated view of DSP, communications, and VLSI be formed. This is the focus of the VLSI Information Processing Systems (VIPS) group at the University of Illinois at Urbana-Champaign. Our ongoing work includes (a) the development of an information-theoretic framework for VLSI, which will unveil the missing design thread that links various levels of the design hierarchy; (b) development of novel algorithm transformation techniques and application of them to design VLSI systems for DSP and communications; and (c) development of CAD tools that incorporate the results from (a) and (b) so that a designer can architect a complex VLSI system in a systematic manner.

In summary, the design of complex low-power and high-speed VLSI systems requires a joint optimization of algorithmic and VLSI parameters. Algorithm transformation techniques presented in this paper are an avenue by which this joint optimization can be achieved.

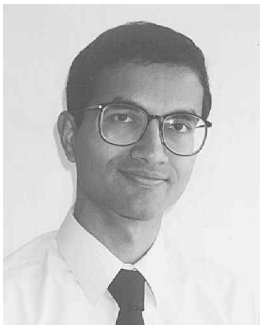
ACKNOWLEDGMENTS

The author would like to acknowledge the efforts of Manish Goel and Raj Hegde in manuscript preparation. Financial support for this work was provided by the National Science Foundation CAREER award MIP-9623737.

REFERENCES

1. T. S. Rappaport, B. D. Woerner, and J. H. Reed, *Wireless Personal Communications: The Evolution of Personal Communications Systems*, Kluwer, Boston, 1996.
2. ETSI/RES, *HIPERLAN*, Services and Facilities, Sophia-Antipolis, France, Dec. 1992.
3. A. Abidi et al., The future of CMOS wireless transceivers, *ISSCC'97*, San Francisco, pp. 118–119.
4. Special issue on software radios, *IEEE Communications Magazine*, May 1995.
5. M. D. Hahm, E. G. Friedman, and E. L. Titlebaum, A comparison of analog and digital circuit implementations of low power matched filters for use in portable wireless communications, *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, Vol. 44, No. 6, pp. 498–506, June 1997.
6. D. K. Shaeffer and T. H. Lee, A 1.5V, 1.5 GHz CMOS low noise amplifier, *IEEE Journal of Solid-State Circuits*, Vol. 32, No. 5, May 1997.
7. A. Rofougaran et al., A 1 GHz CMOS RF front-end IC for a direct-conversion wireless receiver, *IEEE Journal of Solid-State Circuits*, Vol. 31, July 1996, pp. 880–889.
8. K. K. Parhi, Algorithm transformation techniques for concurrent processors, *Proceedings of the IEEE*, Vol. 77, Dec. 1989, pp. 1879–1895.
9. A. Chandrakasan et al., Minimizing power using transformations, *IEEE Transactions on Computer-Aided Design*, Vol. 14, No. 1, Jan. 1995, pp. 12–31.
10. C. Leiserson and J. Saxe, Optimizing synchronous systems, *Journal of VLSI and Computer Systems*, Vol. 1, 1983, pp. 41–67.
11. K. K. Parhi and D. G. Messerschmitt, Pipeline interleaving and parallelism in recursive digital filters—Parts I, II, *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 37, No. 7, July 1989, pp. 1099–1134.
12. H. H. Loomis and B. Sinha, High speed recursive digital filter realization, *Circuits, Systems, Signal Processing*, Vol. 3, No. 3, 1984, pp. 267–294.
13. N. R. Shanbhag and K. K. Parhi, *Pipelined Adaptive Digital Filters*, Kluwer Academic Publishers, Boston, 1994.
14. N. R. Shanbhag and M. Goel, Low-power adaptive filter architectures and their application to 51.84 Mb/s ATM-LAN, *IEEE Transactions on Signal Processing*, Vol. 45, No. 5, May 1997, pp. 1276–1290.
15. W. Sung and S. K. Mitra, Efficient multiprocessor implementation of recursive digital filters, *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, Tokyo, Apr. 1986, pp. 257–260.
16. C. W. Wu and P. R. Cappello, Application specific CAD of VLSI second-order sections, *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 36, May 1988, pp. 813–825.
17. M. Potkonjak and J. Rabaey, Fast implementation of recursive programs using transformations, *Proceedings of ICASSP*, San Francisco, March 1992, pp. V-569–572.
18. H. V. Jagdish et al., Array architectures for iterative algorithms, *Proceedings of the IEEE*, Vol. 75, No. 9, Sept. 1987, pp. 1304–1321.
19. K. K. Parhi et al., Synthesis of control circuits in folded pipelined DSP architectures, *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 1, Jan. 1992, pp. 29–43.
20. R. Hartley and P. Corbett, Digit-serial processing techniques, *IEEE Transactions on Circuits and Systems*, Vol. 37, No. 6, 1990, pp. 707–719.
21. K. K. Parhi, A systematic approach for the design of digit-serial signal processing architectures, *IEEE Transactions on Circuits and Systems*, Vol. 38, No. 4, April 1991, pp. 358–375.
22. M. Goel and N. R. Shanbhag, Dynamic algorithm transformations (DAT) for low-power adaptive signal processing, *Proceedings of the International Symposium on Low-Power Electronic Design*, Monterey, California, Aug. 1997.
23. N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Second Edition, Addison Wesley, 1994.
24. A. Chandrakasan and R. W. Brodersen, Minimizing power consumption in digital CMOS circuits, *Proceedings of the IEEE*, Vol. 83, No. 4, April 1995, pp. 498–523.
25. F. N. Najm, A survey of power estimation techniques in VLSI circuits, *IEEE Transactions on VLSI Systems*, Dec. 1994, pp. 446–455.

26. T. Denk and K. K. Parhi, A unified framework for characterizing retiming and scheduling solutions, *Proceedings of ISCAS'96*, vol. 4, Atlanta, Georgia, May 1996, pp. 568–571.
27. S. S. Sapatnekar and R. B. Deokar, A fresh look at retiming via clock skew optimization, *Proceedings of the ACM/IEEE Design Automation Conference*, 1995, pp. 310–315.
28. S.-Y. Kung, On supercomputing with systolic/wavefront array processors, *Proceedings of the IEEE*, Vol. 72, July 1984, pp. 867–884.
29. S.-Y. Kung, *VLSI Array Processors*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
30. M. Hatamian and K. K. Parhi, An 85 MHz 4th order programmable IIR digital filter chip, *IEEE Journal of Solid-State Circuits*, Feb. 1992, pp. 175–183.
31. N. R. Shanbhag and K. K. Parhi, VLSI implementation of a 100 MHz pipelined ADPCM codec chip, *VLSI Signal Processing VI*, IEEE Press, Oct. 1993 (*Proceedings of the Sixth IEEE VLSI Signal Processing Workshop*, Veldhoven, The Netherlands), pp. 114–122.
32. N. R. Shanbhag and G.-H. Im, VLSI systems design of 51.84 Mb/s transceivers for ATM-LAN and broadband access, *IEEE Transactions on Signal Processing*, Vol. 46, May 1998, pp. 1403–1416.
33. N. R. Shanbhag and K. K. Parhi, Relaxed look-ahead pipelined LMS adaptive filters and their application to ADPCM coder, *IEEE Transactions on Circuits and Systems*, Vol. 40, Dec. 1993, pp. 753–766.
34. G. A. Clark, S. K. Mitra, and S. R. Parker, Block implementation of adaptive digital filters, *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 29, June 1981, pp. 744–752.
35. T. Meng and D. G. Messerschmitt, Arbitrarily high sampling rate adaptive filters, *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 35, April 1987, pp. 455–470.
36. A. Gatherer and T. H.-Y. Meng, High sampling rate adaptive decision feedback equalizer, *IEEE Transactions on Signal Processing*, Vol. 41, Feb. 1993, pp. 1000–1005.
37. D. Gajski et al., *High-level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.
38. H. T. Kung, Why systolic architectures? *IEEE Computer*, Vol. 15, No. 1, Jan. 1982.
39. D. I. Moldovan and J. A. B. Fortes, Partitioning and mapping of algorithms into fixed sized systolic arrays, *IEEE Transactions on Computers*, Vol. C-35, Jan. 1986, pp. 1–12.
40. P. Dewilde, E. Deprettere, and R. Nouta, Parallel and pipelined implementation of signal processing algorithms, in *VLSI and Modern Signal Processing*, Prentice-Hall, 1985.
41. M. C. MacFarland, A. C. Parker, and R. Camposano, The high-level synthesis of digital systems, *Proceedings of the IEEE*, Vol. 78, 1990, pp. 301–318.
42. A. Chandrakasan, Data driven signal processing: An approach for energy efficient computing, *Proceedings of International Symposium on Low Power Electronics and Design*, Monterey, California, August 1996.
43. J. T. Ludwig et al., *Low-power Digital Filtering Using Approximate Processing*, Vol. 31, No. 3, March 1996, pp. 395–400.
44. S. Haykin, *Adaptive Filter Theory*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.
45. G. Long, F. Ling, and J. G. Proakis, The LMS algorithm with delayed coefficient adaptation, *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 37, No. 9, Sept. 1989, pp. 1397–1405.
46. G. H. Im and J. J. Werner, Bandwidth-efficient digital transmission up to 155 Mb/s over unshielded twisted-pair wiring, *IEEE Journal of Selected Areas of Communication*, Vol. 13, No. 9, Dec. 1995, pp. 1643–1655.
47. G. H. Im et al., 51.84 Mb/s 16-CAP ATM LAN Standard, *IEEE Journal of Selected Areas of Communication*, Vol. 13, No. 4, May 1995, pp. 620–632.
48. K. Pahlavan, Channel measurements for wideband digital communication over fading channels, Ph.D. thesis, Worcester Polytechnic Institute, Worcester, Massachusetts, June 1979.
49. C. J. Nicol et al., A low power 128-tap digital adaptive equalizer for broadband modems, *Proceedings of IEEE International Solid-State Circuits Conference*, Feb. 1997, pp. 94–95.
50. P. Landman and J. M. Rabaey, Architectural power analysis: the dual bit type method, *IEEE Transactions on VLSI Systems*, Vol. 3, June 1995, pp. 173–187.



Naresh R. Shanbhag received the B. Tech. degree from the Indian Institute of Technology, New Delhi, India, in 1988, and the Ph.D. degree from the University of Minnesota in 1993, all in electrical engineering. From July 1993 to August 1995, he worked at AT&T Bell Laboratories at Murray Hill in the Wide-Area Networks Group, where he was responsible for development of VLSI algorithms, architectures, and implementations for high-speed data communications applications. In particular, he was the lead chip architect for AT&T's 51.84 Mb/s transceiver chips over twisted-pair wiring for asynchronous transfer mode (ATM)-LAN and broadband access chip sets. In August 1995, he joined the Coordinated Science Laboratory and the Electrical and Computer Engineering Department at the University of Illinois at Urbana-Champaign as an assistant professor. His research interests (see URL http://uivlsi.csl.uiuc.edu /~shanbhag) are in the area of VLSI architectures and algorithms for signal processing and communications. This includes the design of high-speed and/or low-power algorithms for speech and video processing, adaptive filtering, and high-bit-rate digital communications systems. In addition, he is interested in efficient VLSI implementation methodologies for these applications. Dr. Shanbhag received the 1994 Darlington best paper award from the IEEE Circuits and Systems Society, the National Science Foundation CAREER Award in 1996, and is Director of the VLSI Information Processing Systems (VIPS) Group at the University of Illinois at Urbana-Champaign. Since July 1997, he has been appointed as a Distinguished Lecturer for IEEE Circuits and Systems Society and as an Associate Editor for *IEEE Transactions on Circuits and Systems: Part II*. He is the co-author of the research monograph *Pipelined Adaptive Digital Filters* published by Kluwer Academic Publishers in 1994.