Self-Timed Dynamically Pipelined Adaptive Signal Processing System: A Case Study of DLMS Equalizer for Read Channel

Sizhong Chen and Tong Zhang, *Member, IEEE*

***Abstract*— Many pipelined adaptive signal processing systems are subject to a trade-off between throughput and signal processing performance incurred by the pipelined adaptation feedback loops. In the conventional synchronous design regime, such throughput/performance trade-off is typically fixed since the pipeline depth is usually determined in the design phase and remains unchanged in the run time. Nevertheless, in many real-life scenarios, the overall system performance can be potentially improved if we can run-time dynamically configure this trade-off. With this motivation, we propose to apply self-timed pipeline, an alternative to synchronous pipeline, to implement the pipelined adaptive signal processing systems, in which the pipeline depth can be dynamically changed to realize run-time configurable throughput/performance trade-offs. Based on a well-known high speed self-timed pipeline style, we developed architecture and circuit level design techniques to implement the self-timed pipelined adaptation feedback loop with configurable pipeline depth. We demonstrate the proposed design approach using a delayed least mean square (DLMS) adaptive equalizer for magnetic recording read channel. The data transfer rate in hard disk varies as the read head moves among tracks with different distance from the center of the disk platter. By adjusting the pipeline depth on-the-fly, the DLMS equalizer can dynamically track the best equalization performance allowed by the varying data transfer rates. Simulation result shows a significant performance improvement compared with its synchronous counterpart.**

***Index Terms*— Self-timed, Pipelining, Adaptive signal processing, DLMS.**

I. Introduction

OVER the last two decades, adaptive signal processing has developed into a self-contained field [1], [2] that finds wide range of real-life applications such as adaptive equalization, noise and echo cancellation, linear predictive coding, and adaptive beam-forming. Adaptive signal processing algorithms are characterized by their recursive operations for realizing algorithmic self-designing/adaptation. To realize high-throughput VLSI implementation of adaptive signal processing algorithms, architecture-level technique *pipelining* is typically used [3]. Pipelined adaptive signal processing systems are essentially subject to a trade-off between system throughput and signal processing performance, *i.e., deeper pipelined adaptation feedback loop can realize higher throughput, but the delayed feedback will incur larger performance degradation*. It should be pointed out that, for other recursive algorithms such as infinite impulse response (IIR) filtering and Viterbi algorithm, direct pipelining may simply ruin their functionality and appropriate algorithm-level modification is required for the use of pipelining. A pipelined adaptive signal processing algorithm implemented using the conventional synchronous pipeline typically has a fixed pipeline depth that is determined in the design phase to accommodate the highest run-time throughput requirement. Although it is possible to on-the-fly configure the pipeline depth of synchronous pipeline by selectively bypassing certain levels of registers, this is very inflexible and cannot realize fine-grain graceful configuration on the throughput/performance trade-offs. For example, consider an 8-stage pipelined recursive adaptation loop in which the registers are almost evenly placed along the loop for maximizing the throughput. If we bypass one level of registers to realize a 7-stage pipeline, the delay of the critical path may double and the throughput will reduce almost by half.

Self-timed pipeline [4], [5] works in a different way from its synchronous counterpart. Without a common and discrete notion of time, self-timed pipeline relies on the handshake between components to perform the synchronization and communication. Each distinct data propagating through a self-timed pipeline is conventionally called a *token*. The pipeline depth of a self-timed pipeline simply equals the number of tokens present in the pipeline at the same time. Hence, we can dynamically configure the pipeline depth by controlling the number of tokens present in the pipeline. This property of self-timed pipeline has been exploited in the design of a mixed synchronous-asynchronous FIR filter that can support variable latency (in terms of clock cycles) [6] and power management of an embedded, single-issue processor [7]. In pipelined adaptive signal processing systems, the pipeline depth of the adaptation feedback loops is the key to tune the inherent trade-off between throughput and signal processing performance. This directly motivates us to apply self-timed pipeline for the implementation of adaptive signal processing systems to realize gracefully configurable throughput/performance trade-off. This can be leveraged to improve the overall system performance in many circumstances. For example, for adaptive signal processing systems with variable data rate, we can dynamically adjust the pipeline depth to the minimum allowable value according to the current data rate to realize the best signal processing performance. Although the basic idea of the above design approach is simple and intuitive, how to implement it in the real systems involves the following three critical design issues:

Manuscript received August, 2004; revised January, 2005.

The authors are with the department of Electrical, Computer and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY.

1) What type of self-timed pipeline structure should be used? Clearly, to justify the practicality of this design approach, the employed self-timed pipeline must be able to support the same (or comparable) throughput as its synchronous counterpart when they have the same pipeline depth. This means that the recursive self-timed pipeline datapath should have the same (or comparable) propagation delay as its synchronous counterpart. This is a very strict requirement since most self-timed pipeline design schemes involve extra delay overhead for realizing self-timed handshake and have the longer latency than their synchronous counterparts, although they can support very fine-grain pipeline to realize high throughput. In this work, we propose to use the well-known Ted William's high-speed self-timed pipeline [4], [8] because of its *zero-delay-overhead* feature (i.e., no extra handshake delay is incurred when data propagate through the pipeline). Hence the zero-delay-overhead pipeline can achieve the same latency performance as its synchronous counterpart.
2) How to realize the self-timed data flow synchronization in the recursive adaptation loop? In self-timed datapath, synchronization of parallel computational threads relies on *forks* and *joins*, where fork refers to a stage with one input channel and multiple output channels and join refers to a stage with multiple input channels and a single output channel. The recursive adaptation loop of adaptive signal processing algorithms contains many forks and joins. However, like many other self-timed pipeline styles, the zero-delay-overhead self-timed pipeline was initially proposed for linear datapath (i.e., without forks and joins). Therefore, it must be appropriately modified to support forks and joins.
3) How to realize run-time addition/removal of tokens in order to change the pipeline depth? In a feedforward-only datapath, the pipeline depth can be readily changed by adjusting the input data rate. However, as we will show later, it is not trivial to change the pipeline depth in recursive adaptation loops. We have to design some special circuit elements that can be placed on the recursive adaptation loop to realize run-time addition/removal of tokens.

In this work, we developed techniques to tackle the above design issues, which provides a complete architecture/circuit-level design solution to realize the run-time configuration of throughput/performance trade-off in pipelined adaptive signal processing systems. Furthermore, we use magnetic recording read channel adaptive equalizer as a test vehicle to demonstrate the effectiveness of the proposed design approach. This is motivated by two factors: (1) PRML (partial response maximum likelihood) equalization has been widely used in magnetic storage systems [9], where LMS (least mean square) adaptive filters are typically used to improve the equalization performance. LMS filter must be pipelined to meet the very high data transfer rate requirement in magnetic storage. (2) Modern magnetic hard disks use a technique called *zoned bit recording* [10] to maximize the storage capacity. It groups the adjacent disk track into zones, and the outside zones have more sectors per track than the inside zones. As a result, when the disk read head moves across different zones of tracks, the data transfer rate will vary. For example, Hitachi Deskstar 120G hard drive [11] has 31 track zones and its data transfer rate varies between 185Mbps and 384Mbps. Therefore, magnetic recording read channel PRML equalizer can leverage the proposed design approach to improve the overall performance by dynamically tracking the best possible equalization performance allowed by different data transfer rates. We designed the recursive adaptation loop of an LMS filter in PRML equalizer at transistor level using our proposed design techniques. Circuit simulation shows that it can support upto 1.1Gbps data rate. Assuming 4 different data transfer rates with the ratio of 4:5:6:7, MATLAB simulations show a significant PRML equalization performance improvement over the one using conventional synchronous pipeline.

As self-timed VLSI system design becomes an increasingly mature technology and holds great promise to tackle many challenges faced by the current semiconductor industry, appropriately exploiting the unique characteristics of self-timed circuits at the signal processing algorithm and architecture levels for improving the overall system performance becomes a new and promising research paradigm. We believe that this work is a valuable step in this direction, and hopefully it will prompt more research efforts to this paradigm. Remainder of this paper is organized as follows. Section II introduces necessary background of self-timed pipeline. Section III presents the self-timed pipelined LMS filter architecture. Key circuit design issues are discussed in Section IV. The circuit design and simulation results of an example LMS filter in PRML equalizer are given in Sections V, and the conclusion is drawn in Section VI.

II. Background

This section briefly describes the zero-delay-overhead self-timed pipeline according to [4] and discusses some basic concepts and properties of self-timed pipeline. For detailed discussion on self-timed design, readers are referred to [5].

Fig. 1(a) shows the structure of a zero-delay-overhead self-timed pipeline, where the function block at each pipeline stage is implemented using dynamic differential cascode voltage switch logic (DCVSL) [12] as illustrated in Fig. 1(b). The data validity information in support of self-timed operation is embedded into the dual-rail signaling of the DCVSL logic: When the dual-rail output F and $\overline{F}$ are both 0, it represents an *invalid* datum; when one of F and $\overline{F}$ switches to 1 during evaluation (EN=1), it represents a *valid* datum (1 or 0). The completion detector (CD) at each stage, as shown in Fig. 1(a), generates 1 when it detects valid data, otherwise generates 0.

The basic idea of zero-delay-overhead self-timed pipeline is to make each DCVSL stage keep *ready-to-evaluate* status so that it can start the evaluation as soon as tokens arrive, hence tokens can propagate through the pipeline without being blocked (or delayed) by handshake. According to the pipeline as shown in Fig. 1(a), the operation of zero-delay-overhead self-timed pipeline can be described as follows: The pipeline is initialized in such a way that each stage generates invalid

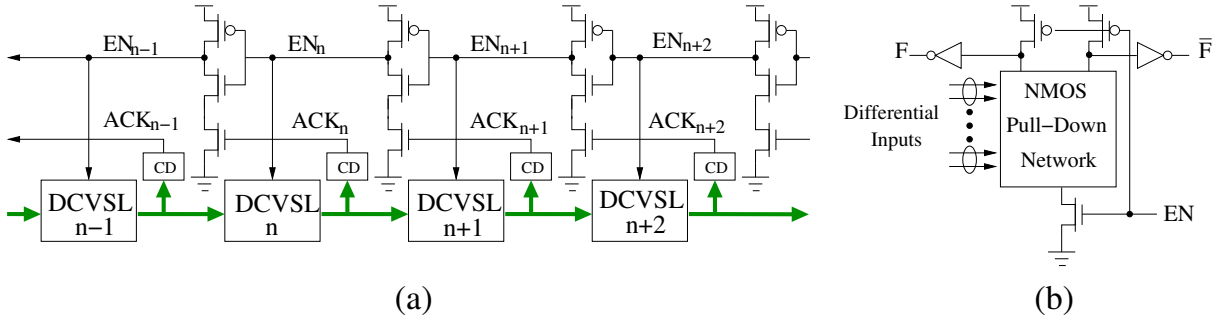


Fig. 1. (a) Zero-delay-overhead self-timed pipeline structure, and (b) DCVSL structure.

output data (i.e., each ACK$_i$ is 0) and is ready to evaluate (i.e., each EN$_i$ is 1). Once valid data enter the pipeline and reach stage n, stage n starts the evaluation; after finishing the evaluation, it outputs valid data to its successor (i.e., stage $n+1$) that will subsequently start the evaluation. The output valid data of stage n will invoke ACK$_n$ switch from 0 to 1. As both EN$_n$ and ACK$_n$ are 1, according to Fig. 1(a), EN$_{n-1}$ will switch from 1 to 0, leading to the precharge of stage $n-1$. In the same manner, after the stage $n+1$ finishes the evaluation and generates valid data, stage $n+2$ will start to evaluate and stage n will be precharged (i.e., EN$_n$ switches from 1 to 0). Clearly, EN$_n$=0 will make EN$_{n-1}$ switch back to 1 so that stage $n-1$ becomes ready to receive and evaluate new valid data. In this way, valid data can propagate through the pipeline datapath. The name *zero-delay-overhead* comes from the fact that the forward propagation latency exactly equals the function block latency without any extra delay incurred by self-timed handshake as in many other self-timed pipeline design styles. Such high speed performance comes at the cost of degraded robustness, i.e., to guarantee the correct functionality, the precharge of a stage *must* be faster than the evaluation of its successor. This assumption is practically reasonable and can be easily satisfied in the real implementations. Finally, we note that the dual-rail dynamic logic DCVSL is self-consistent with such zero-delay-overhead self-timed handshake and can provide a 2x speed performance advantage compared with conventional static CMOS logic. As the cost, dynamic circuits generally suffer from higher power dissipation and less noise immunity.

From the above description, we know that, when data propagate through the pipeline, all the stages alternate among three different statuses (denoted as D, S, and B respectively as illustrated in Fig. 2): (1) *holds a data* (D): the stage receives and processes valid data (EN=1); (2) *holds a spacer* (S): the stage is precharged (EN=0) and generates invalid data output; (3) *holds a bubble* (B): the stage does not receive valid data and keeps invalid data output with EN=1. Notice that a stage holding a data is always followed by a stage holding a spacer. It is a convention that we say the two neighboring stages holding a data and a spacer together hold a *token*. As a token moves forward through the pipeline, what is left is a bubble that is available to be occupied by another token.

The *pipeline depth* of a self-timed pipeline equals the number of tokens present in the pipeline at the same time, which is a variable determined by how the pipeline is initialized and/or how it is used by the environment. This is in sharp contrast to a synchronous pipeline, where the pipeline depth is fixed as the number of levels of registers along the datapath. We call the feature of variable pipeline depth as **dynamic pipelining**. How to control the pipeline depth essentially differs between feedforward-only datapaths and datapaths with feedback (or recursive datapaths):

- In a feedforward-only datapath such as an FIR filter, the pipeline depth can be easily controlled by the input data rate. A self-timed feedforward-only pipeline behaves like a shallow pipeline with a small pipeline depth for a slow input data rate, yet will accommodate a fast input data rate as a deep pipeline.
- In a recursive datapath, the pipeline depth is determined by the initialization. Once the stages in the recursive datapath are initialized to be tokens and bubbles, the number of tokens and bubbles remains constant and cannot be changed by varying the input/output data rate. Hence, to dynamically change the pipeline depth, the recursive loop must contain some special elements that can be run-time configured to add/remove tokens. Fig. 2(b) shows a recursive self-timed pipeline, in which parallel threads of computation are synchronized by joins and forks.

We note that the throughput of a self-timed recursive loop does not *always* increase with the increase of the pipeline depth [13]. The throughput depends on the ratio of the number of tokens (or pipeline depth) to the number of bubbles. If the ratio is less than a threshold, the datapath is in the *token limited* mode and increasing the pipeline depth will improve the throughput. Otherwise, the datapath is in *bubble limited* mode and increasing pipeline depth will decrease the throughput. Readers are referred to [5], [13] for a detailed discussion.

Finally, it should be pointed out that, besides the pipeline register bypass as we mentioned in Section I, a synchronous technique called wave pipelining [14], [15] can also realize a variable pipeline depth by allowing multiple data waves to flow at any time between two neighboring registers. However, this approach requires extensive design effort to accurately balance the path delays, and is very vulnerable to process, temperature and voltage variations. The latter will get worse since the variation is inevitably becoming more and more serious as CMOS technology continuously scales down.

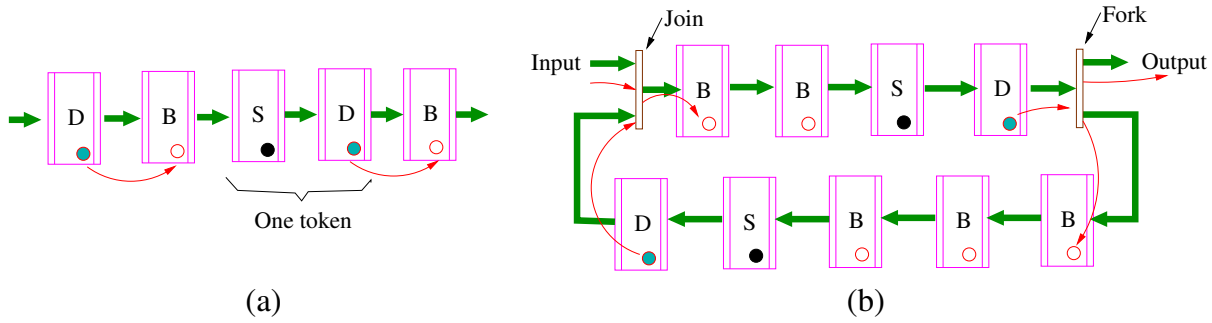


Fig. 2. (a) Abstract view of a feedforward-only zero-delay-overhead pipeline with token-flow snapshot, and (b) abstract view of a recursive zero-delay-overhead pipeline with token-flow snapshot.

III. Architecture of Dynamically Pipelined LMS Filter

In this section, we present the architecture of pipelined LMS adaptive filter that employs the self-timed pipeline to realize dynamic pipeline depth configuration. In the conventional LMS algorithm [2], the filter coefficients are updated as:

$$\begin{cases} \mathbf{w}(n) &= \mathbf{w}(n-1) + \mu \cdot e(n) \cdot \mathbf{u}(n), \\ e(n) &= d(n) - \mathbf{w}^T(n-1) \cdot \mathbf{u}(n), \end{cases} \quad (1)$$

where $\mathbf{w}^T(n) = [w_0(n), w_1(n), \cdots, w_{N-1}(n)]$ is the filter coefficient vector, $\mathbf{u}(n)$ is the input vector, μ is the step size, $d(n)$ is the desired signal, and $e(n)$ is the error.

Since the filter coefficients adaptation loops prevent the standard LMS adaptive filter from being directly pipelined, we have to appropriately modify the algorithm to intentionally create extra delays on the adaptation loops for pipelining. To this end, two methods have been proposed: delayed LMS (DLMS) [16], [17] and relaxed look-ahead pipelined LMS (PIPLMS) [3], [18]. DLMS introduces the same amount of delays in the adaptation formula of all the filter coefficients, while PIPLMS presents a more general framework to insert delays onto the adaptation loop and includes DLMS as a special case. The filtering performance of these adaptation-delayed approaches is inferior to that of the standard LMS algorithm. The performance loss is generally determined by the amount of delays inserted onto the adaptation loop. Although some modified DLMS algorithms [19]–[21] have been proposed to improve the performance, they suffer from significant hardware overhead. For the purpose of simplicity, we consider the standard DLMS in this work. The design scheme presented below can be straightforwardly applied to other pipelined adaptive filters. The DLMS is formulated as:

$$\begin{cases} \mathbf{w}(n) &= \mathbf{w}(n-1) + \mu \cdot e(n-D) \cdot \mathbf{u}(n-D), \\ e(n-D) &= d(n-D) - \mathbf{w}^T(n-D-1) \cdot \mathbf{u}(n-D), \end{cases} \quad (2)$$

where D is the number of extra delays on the adaptation loops for pipelining. As the cost of the increased throughput, the convergence rate is degraded due to the delayed feedback of error signals.

Fig. 3(a) shows a DLMS architecture with synchronous pipeline, where all the registers R's share a common clock. The coefficient adaptation delay D is fixed as the number of the extra levels of registers on the adaptation loops. Retiming [22] can be used to distribute the registers along the adaptation loops to balance the latency between neighboring registers and hence improve the overall throughput. In this scenario, the trade-off between throughput and equalization performance is **fixed** at the design phase.

Fig. 3(b) shows the self-timed DLMS with dynamic pipeline depth control. For simplicity and clarity, we omit the joins and forks. Each computational unit (multiplier or adder) is implemented using zero-delay-overhead self-timed pipeline with fine granularity. The term *fine granularity* means that the self-timed pipeline datapath contains enough number of pipeline stages so that the adaptation loops always work in the token limited mode within the target pipeline depth range. Therefore, the filter throughput monotonically varies with the pipeline depth (or the number of tokens), i.e., we can increase and decrease the throughput by increasing and decreasing the number of tokens in the datapath, respectively. In this scenario, the value of D in (2) will become a variable that is equivalent to the current pipeline depth. To enable the above self-timed DLMS to work in a synchronous environment, we need to use two input self-timed FIFOs (first-in first-out) buffers and one output self-timed FIFO to communicate with external synchronous environment, as shown in Fig. 3(b).

To realize the correct filtering according to (2), we must initialize the self-timed DLMS in such a way that all the coefficient adaptation loops have the same number of tokens (i.e., the same number of pipeline stages are initialized to hold valid data in all the loops). Moreover, as shown in Fig. 3(b), the buffer element TR, which contains two pipeline stages and each stage contains a simple DCVSL inverter, must be initialized to hold a token, i.e., one stage holds valid data and another one holds a spacer. The other buffer elements that are labelled as SR in the buffer chain as shown in Fig. 3(b) can be initialized to either token or bubble subject to the requirement that the buffer chain must have the same number of tokens as that of the coefficient adaptation loops.

As we pointed out in the above, the number of tokens (or pipeline depth) in a recursive loop cannot be changed by simply changing the input/output data rate. To support dynamic pipeline depth control, we propose to modify one pipeline stage on the recursive loop in such a way that this pipeline stage can act as a *pipeline depth modifier* (PDM). Configured by external request signals, PDM can add/remove tokens or simply operate as a normal pipeline stage. The design of PDM will be discussed in Section IV-B. Clearly,

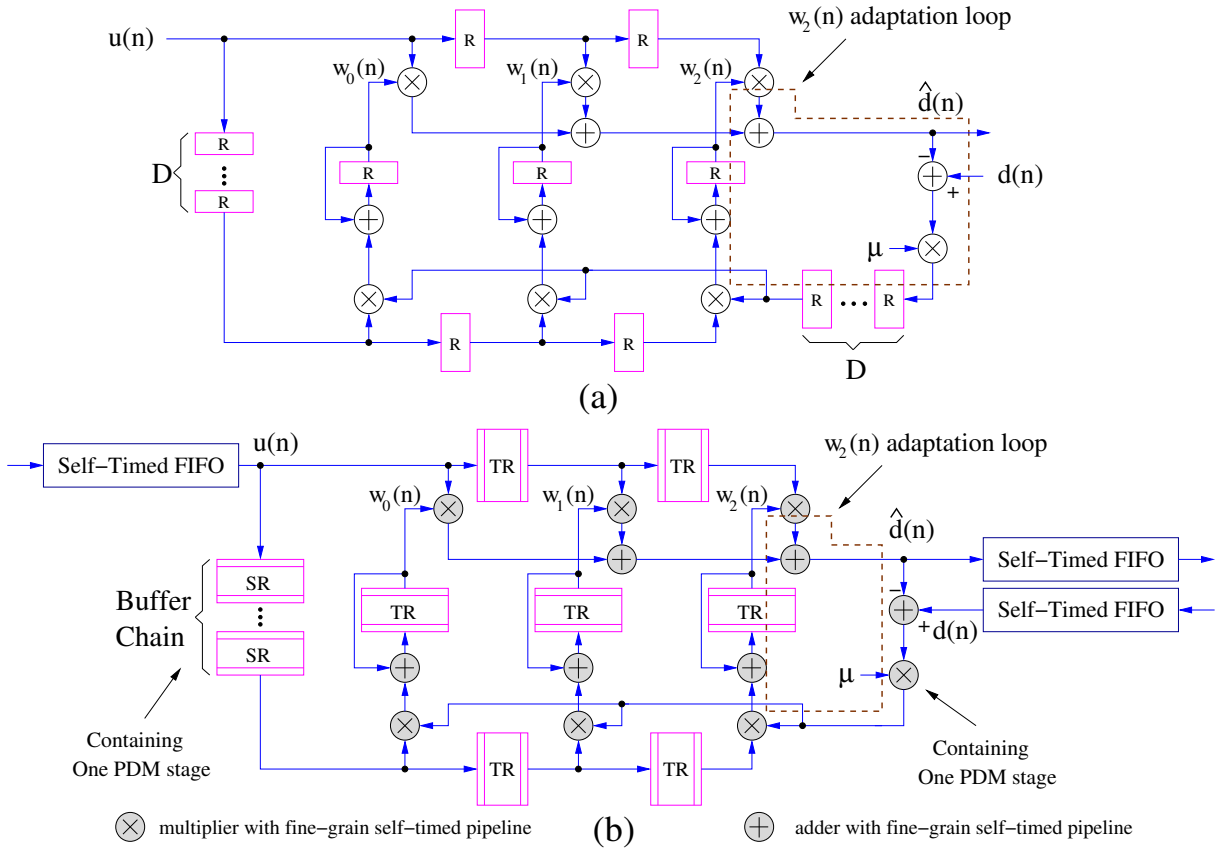


Fig. 3. Synchronous and self-timed DLMS filter architectures.

it is desirable to put such PDM on the common part of all the coefficient adaptation loops in DLMS so that we can control the number of tokens on all the loops simultaneously using only one PDM. Moreover, when we add/remove tokens to/from the loops, the same amount of tokens should be correspondingly added or removed in the SR buffer chain in order to guarantee the correct functionality according to (2). Hence, as illustrated in Fig. 3(b), totally two pipeline stages should be modified as PDM.

We expect that such self-timed pipelined adaptive filters can be leveraged to improve the overall system performance in many scenarios, including (a) When the input data rate changes, we can correspondingly change the pipeline depth to a minimum allowable value to obtain the best filtering performance. (b) Using the self-timed FIFOs as elastic buffers, we can configure the throughput vs. performance trade-off when the adaptive filter works in different modes with different filtering performance requirements (such as training and tracking modes). (c) By examining the magnitude of the error signal and using the self-timed FIFOs as elastic buffers, we can dynamically adjust the throughput vs. performance trade-off to achieve better overall system performance, i.e., if the magnitude of the error signal becomes larger, we can decrease the pipeline depth under the constraint that the FIFOs will not overflow, and after the magnitude of the error signal drop back, we can correspondingly increase the pipeline depth.

IV. Design of Key Circuit Elements

This section presents the circuit level design techniques to tackle the latter two design issues posed in Section I, in support of the implementation of dynamically pipelined adaptive signal processing systems. We present the techniques to design forks and joins for zero-delay-overhead self-timed pipeline in Section IV-A, and present a design solution to implement the pipeline depth modifier in Section IV-B.

A. Design of Fork and Join

In self-timed datapath, synchronization of parallel computational threads relies on forks and joins, where fork refers to a stage with one input channel and multiple output channels and join refers to a stage with multiple input channels and a single output channel. The original zero-delay-overhead pipeline is only suitable for the datapath without forks and joins. However, implementation of coefficient adaptation feedback loop in DLMS and other adaptive filters involves many forks and joins for the synchronization of computations. In this work, we develop the design of fork and join that support zero-delay-overhead pipeline, as described in the following.

When a **fork** holds a token, some of its successors can evaluate immediately while some others may stall for certain reasons (e.g., being blocked by their own successors). Hence a fork should hold the token until it receives ACK signals from all of its successors. But if we simply AND all the ACK signals together, this may result in a potential malfunction:

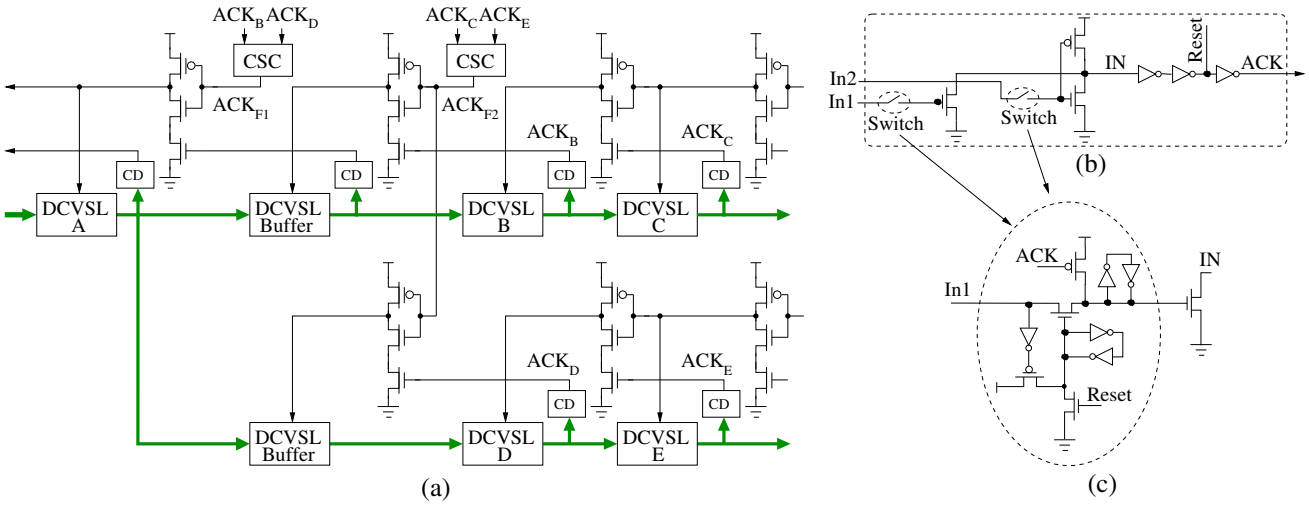


Fig. 4. (a) Fork design, (b) 2-input clock strobe circuit (CSC), and (c) strobe circuit switch.

consider the scenario that, before the fork receives the ACK signals from all the successors, one of its successors has passed the token to its own successor and is ready to receive a new token. This successor will interpret the token still being held by the fork as a new token and then process this token for the second time, which will lead to a malfunction. It is referred to as slow or stalled right environment (SRE) problem in [23]. Two solutions are proposed to address the fork design issue in [23]. The first solution is to condition the acknowledgment signals received from the stages next to the fork stage and make them persistent. The second solution, which requires more significant modifications, is to modify the basic control circuit of every subsequent pipeline stage and make all the acknowledgment signals persistent. In this paper, we propose a fork design scheme as illustrated in Fig. 4 (a), in which we adopt the *clock strobe circuit* as shown in Fig. 4(b) and (c) from IPCMOS developed at IBM [24]. The function of CSC is to combine several acknowledge ACK pulses (each ACK is a 0→1→0 pulse) into one pulse, somewhat similar to the functionality of a C-element that combines signal levels (0 or 1) not pulses. For detailed discussion of CSC, readers are referred to [24]. As shown in Fig. 4(a), identical buffers (few cascaded DCVSL inverters) are placed on the interface between fork and each output channel. Controlled by the same signal ACK_{F2} that is generated by a CSC with the input of ACK_C and ACK_E, the buffers on all the channels start to hold bubble (i.e., be available to receive new token) at the same time.

The operation can be briefly described as follows: when a token propagates through a fork, the valid data will be copied to all the buffers simultaneously and then propagate on all the channels independently. Notice that each ACK pulse generated by the completion detector indicates that a token has propagated through its associated DCVSL function block. With the help of CSC that combines several independent ACK pulses into one pulse, the fork will hold spacers (i.e., precharged and keeps EN=0) until the valid data move away on all the channels.

According to its definition, a **join** must wait until all the inputs become valid before generating valid output data. However, if we use the conventional DCVSL circuit to implement the join, this cannot be always guaranteed, i.e., the join stage may generate valid output data even though the data from one or more input channels are not valid yet. For example, consider the DCVSL 2-input AND/NAND gate in Fig. 5(a) that is derived from the binary decision diagram (BDD) [25] of the Boolean logic function. Suppose the input {A, $\overline{\text{A}}$} and {B, $\overline{\text{B}}$} are from two independent channels, i.e., this AND/NAND gate acts as a join. When the input {B, $\overline{\text{B}}$} is not valid and {A, $\overline{\text{A}}$} is valid and equals to {0, 1}, this gate will generate valid data output. This violates the above requirement on join.

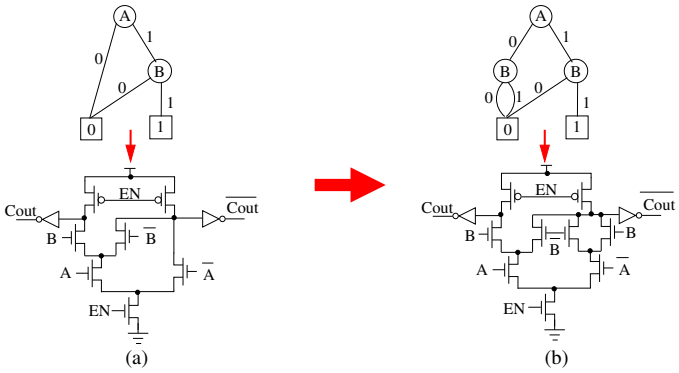


Fig. 5. Modified DCVSL circuit for joins.

To solve this problem, authors of [23] proposed to add explicit request signals to each input channel of the join, and thus required additional control circuit. Our approach is to modify the DCVSL design of the join in such a way that it cannot finish the evaluation until the data from all input channels become valid. To this end, we propose to apply *redundant binary decision diagram* (RBDD) to design the pull down network of the DCVSL function block of a join. The only difference between RBDD and BDD is that, in RBDD, each decision path must go through the decision of all the input data, as illustrated in Fig. 5(b). It is clear that the resulted

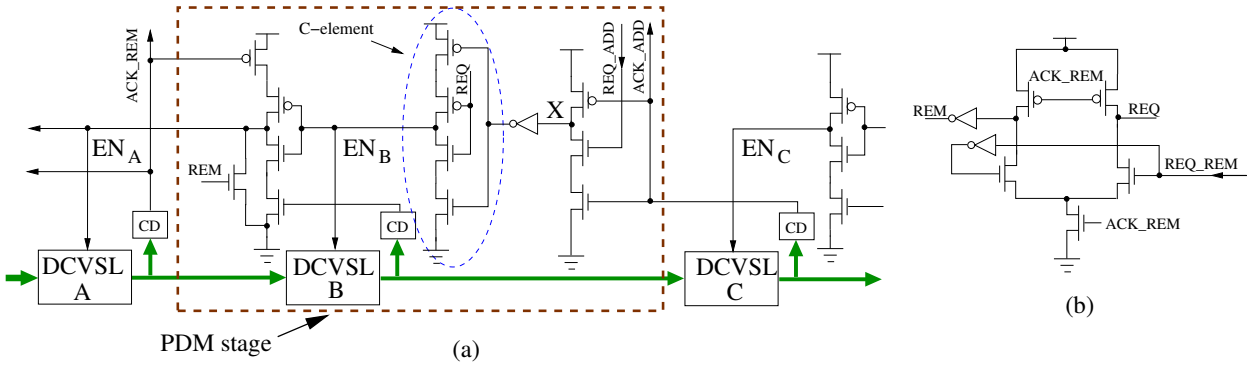


Fig. 6. Pipeline depth modifier structure.

circuit will be more complex than the original one. As a return, such circuit will keep the output as invalid (i.e., equals to {0, 0}) until all the input data have become valid (or differential). Through such simple circuit modification, we can easily realize the join in the self-timed pipeline.

B. Design of Pipeline Depth Modifier

As we mentioned earlier, the control of the pipeline depth along the recursive datapath is realized by modifying one pipeline stage so that this stage can act as a pipeline depth modifier (PDM) as shown in Fig. 6, where Fig. 6(a) shows the structure of PDM and Fig. 6(b) shows the circuit that generates two internal signals REM and REQ for PDM. The design of PDM involves trading the zero-delay-overhead property for augmented functionality of adding/removing tokens. This is realized by inserting a well-known C-element[1] [26] in the original zero-delay-overhead pipeline, as illustrated in Fig. 6(a), to *destroy* the zero-delay-overhead property of one pipeline stage and, consequently, provide the functionality of adding/removing tokens. The operation of PDM is described as follows. Configured by external host through four handshake signals including REQ_ADD, ACK_ADD, REQ_REM, and ACK_REM, The PDM can operate in three different modes:

1) Normal mode: By default, both REQ_ADD and REQ_REM are 1 and PDM works as a normal pipeline stage that, however, is no longer zero-delay-overhead;
2) Token-removal mode: To remove a token from the recursive datapath (i.e., reduce the pipeline depth by 1), the host switches REQ_REM to 0 right after ACK_REM switches from 1 to 0, and then switches REQ_REM back to 1 right after the next 1→0 switch of ACK_REM;
3) Token-addition mode: To add a token into the recursive datapath (i.e., increase the pipeline depth by 1), the host switches REQ_ADD to 0 right after ACK_ADD switches from 1 to 0, and then switches REQ_ADD back to 1 right after the next 1→0 switch of ACK_ADD.

In the normal mode where both REQ_ADD and REQ_REM are 1, the circuit in Fig. 6 directly reduces to the one as shown in Fig. 7. Because of the extra C-element in PDM, when the PDM stage holds a bubble (i.e., no valid data present), EN_B remains 0 so that the DCVSL function block in the PDM stage keeps being precharged. This is contrast to a zero-delay-overhead pipeline stage: when it holds a bubble, its EN signal is 1 and hence its DCVSL function block is ready to evaluate. When valid data reach the PDM stage, the data will be blocked until EN_B switches from 0 to 1 to enable the data to propagate through. Notice that EN_B will be 1 only when ACK_REM is 1 (i.e., valid data are detected at the front of PDM stage) and ACK_ADD is 0 (i.e., no valid data are being held by the successor of PDM stage). Clearly, the PDM stage is no longer zero-delay-overhead. As a return for the extra delay overhead incurred by the C-element, we can readily realize the token-removal operation as described below.

The basic idea of realizing token-removal is that, when a token reaches the PDM stage (i.e., the token is being held by the stage A according to Fig. 6), we keep the PDM stage being precharged (EN_B keeps 0) and, meanwhile, mimic the normal handshake protocol so that the stage A is precharged first (EN_A switches 0) to remove the token, and then begins to hold a bubble (EN_A switches back to 1) and hence is ready to receive a new token. In this way, a token simply disappears at the interface between the PDM stage and its preceding stage. We can describe the token-removal as the following two-step process:

1) *Preparation step*. When the host needs to remove one token, it switches REQ_REM from 1 to 0 after it detects 1→0 switch of ACK_REM. Notice that 1→0 switch of ACK_REM indicates a token just propagate through the pipeline in the normal mode. According to Fig. 6(b), the 1→0 switch of REQ_REM will affect the circuit behavior when the next token arrives (i.e., ACK_REM switches from 0 to 1).
2) *Removal step*. When the second token propagates through the stage A and reaches the PDM stage, the ACK_REM will switch from 0 to 1. Because REQ_REM is 0 now, according to the circuit that generates the signals REM and REQ as shown in Fig. 6(b), both REM and REQ will be 1. REM=1 will discharge the signal EN_A and therefor pre-charge the stage A; REQ=1 will keep the EN_B as 0 and the PDM stage being pre-charged. In this way, the token is removed from the pipeline. The additional

[1]The C-element is a state-holding element, and, with two inputs X and Y and one output Z, its function can be described as: if X is equal to Y, then Z=X, else Z holds previous value. C-element is extensively used in many self-timed pipeline styles to realize handshake.

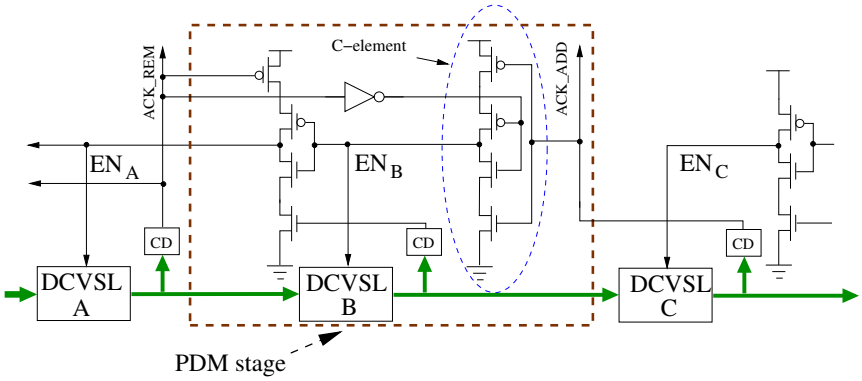


Fig. 7. Equivalent pipeline depth modifier in normal mode (both REQ_ADD and REQ_REM are 1).

PMOS transistor in stage A controlled by ACK_REM is used to prevent the short circuit current when REM=1 and EN_B=0. After the precharge of stage A completes, both ACK_REM and REM will switch to 0. Because both EN_B and ACK_REM are 0 now, EN_A will switch from 0 to 1 so that stage A will hold a bubble (i.e., it is ready to receive a new token). Meanwhile, after the host detects ACK_REM switches from 1 to 0, the host will switch the REQ_REM back from 0 to 1 so that the PDM stage will work in the normal mode when succeeding tokens come.

Let's consider the operation of **adding** a token. Recall that in the normal operation, after one stage, say stage i, receives valid data from its predecessor, say stage $i-1$, and generates valid output data, stage $i-1$ will be immediately precharged and hence hold a spacer. The basic idea of adding a token is to prevent the precharge operation so that the predecessor still holds the valid data that will be interpreted as a new token. As shown in Fig. 6, after valid data propagate through stage B and reach stage C, the internal node X is 1. When stage C generates valid output data, since the request signal REQ_ADD has been switched to 0, the node X will keep 1, i.e., stage B will not be precharged and hence still hold the valid output data. As the valid data held by stage C move forward, stage C will be precharged that leads to the 1→0 switch of ACK_ADD. This acknowledges the host that the command of adding a token has been processed and host must switch the REQ_ADD back to 1 immediately. After stage C is enabled to receive new valid data (i.e., stage C begins to hold a bubble), it will interpret the valid data still being held by stage B as a new valid data. Thus the same valid data will propagate through stage C for the second time leading to one more token in the datapath.

In the above design, the pipeline depth is changed by directly duplicating or removing a token in the datapath. Clearly this is not applicable to some applications such as microprocessor where the interdependence of data and instructions should be maintained all the time, e.g., an instruction can not be simply duplicated or removed from a pipeline. In the context of signal processing, this approach is valid since we just need to preserve the functionality of the algorithm instead of the one-to-one mapping of the input-output behavior.

V. A Case Study of DLMS Equalizer for Magnetic Recording Read Channel

Partial response maximum likelihood (PRML) equalization techniques such as EPRIV PRML are being widely applied to magnetic recording read channels in order to increase the data transfer rate and storage density [27], [28]. An EPRIV PRML read channel mainly contains two key blocks: adaptive filter and Viterbi detector. The adaptive filter typically uses LMS algorithm. As we mentioned in Section I, because of the use of zoned bit recording, the data transfer rate in modern hard disks varies as the disk read head moves across different zones of tracks on the disk. Moreover, the very high data transfer rate demands LMS adaptive filter to be pipelined. Hence, the design of pipelined LMS adaptive filter for EPRIV PRML read channels provides an appropriate real-life application of the proposed design approach.

Using the above proposed design techniques, we designed a coefficient adaptation loop in a 9-tap DLMS adaptive filter that is used in EPRIV PRML read channels. Because of the lack of mature automation design tool for self-timed pipeline, the design is conducted manually at transistor level with IBM 7HP 0.18μm BiCMOS technology in the CADENCE design environment. The finite word-length configuration of this design is as follows: the input sampled data is 6-bit; the filter tap coefficients are 10-bit; the feedback error signal is 10-bit; the step size is 9-bit; all the other intermediate results (outputs of multipliers and adders) are 16-bit. The two's complement multiplication is realized using Baugh-Wooley multiplier that consists of a carry-save adder array and one additional carry ripple adder to merge the partial results. The output of the 9 filter taps is summed together by a carry-save adder array followed by a carry ripple adder. In order to guarantee the datapath can always work in token-limited mode, the numbers of pipeline stages of all the multipliers and the carry-save adder array range from 6 to 8. Moreover, in order to reduce the hardware complexity overhead, we use *partial* completion detection in the design of multipliers and adder array: Because of the wide bit-width in the computations, implementing completion detectors for all the bits in the same pipeline stage will result in very high complexity overhead. Leveraging the regular and well-balanced datapaths in both multipliers and adder array, we only detect a small portion of

all the bits on the same pipeline. Furthermore, because of the fine pipeline granularity, the delay of a completion detector and the followed handshake circuit is comparable to or even larger than the delay one pipeline stage, which further helps to justify the use of partial completion detection.

For the purpose of comparison, we designed a DCVSL based synchronous counterpart of the same coefficient adaptation loop. The pipeline depth is 7, where each multiplier and the carry-save adder array are pipelined with 2 stages, and the CADENCE simulation result shows that the estimated delay of critical path is 1ns leading to a throughput of 1 giga samples per second. The supply voltage is set to be 1.8 V. In the self-timed implementation, with the equivalent pipelining depth (token number), the maximum throughput is about 1.1 giga samples per second. This suggests that under the same pipeline depth, self-timed design can achieve slightly higher maximum throughput than the synchronous one, which can be explained intuitively as follows: The synchronous and self-timed datapaths have the same latency denoted as τ. In the context of synchronous design, with the pipeline depth of D, the pipeline registers cannot be perfectly evenly distributed along the synchronous pipeline leading to the critical path longer than τ/D; on the other hand, the fine granularity of self-timed pipeline can help the tokens be evenly distributed along the pipeline datapath, so that the critical path of the self-timed pipeline can closely approach τ/D. In order to realize fine grain self-timed pipeline, compared with its synchronous counterpart, the self-timed pipeline (implemented using partial completion detection as described above) consumes more transistors (19% more transistors in this case).

BER (bit error rate) performance of the considered EPRIV PRML read channel is simulated using MATLAB. Assume the data rate uniformly changes among 4 values corresponding to the throughput of DLMS filters with pipeline depth of 4, 5, 6, and 7, respectively. The synchronous DLMS filter keeps the fixed pipeline depth of 7 no matter what the current data rate is, nevertheless, its self-timed counterpart can dynamically change the pipeline depth by changing the number of tokens and correspondingly changing the step size μ to realize the best possible equalization performance. The read-back signal from the disk head is modeled by convolving random data sequence with the Lorentzian pulse and adding white Gaussian noise. Output of the adaptive DLMS filter is feed to a 8-state Viterbi decoder for sequence detection. In the simulation, we assume the length of the training sequence is 120. Fig. 8 shows the BER vs. SNR simulation results of both synchronous implementation with fixed pipelining depth of 7 and self-timed implementation with variable pipelining depth. This clearly shows the great potential on improving the overall system performance using the proposed self-timed pipeline with dynamic pipelining depth control.

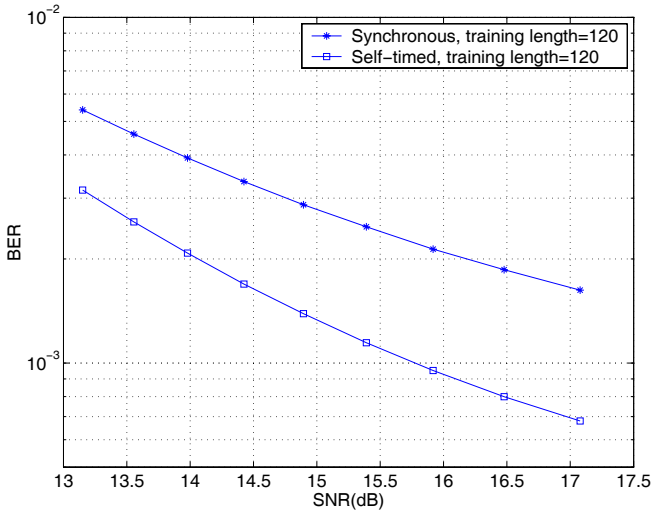


Fig. 8. Simulated performance of EPRIV PRML read channel using synchronous and self-timed DLMS equalizers with the training length of 120.

VI. Conclusion

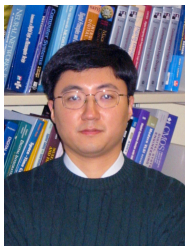
In this paper, for the first time, we propose to exploit the dynamic pipelining property of self-timed pipeline to realize reconfigurable throughput/performance trade-off in pipelined adaptive signal processing systems. PRML read channel equalizer is considered in this work as a test vehicle. For practical implementation, we propose to use a zero-delay-overhead self-timed pipeline style that supports very high speed operation. We develop techniques to enable the application of zero-delay-overhead self-timed pipeline in this context and realize run-time pipeline depth control. Simulations under variable data rate scenarios demonstrate a significant performance gain. It is our hope that this work will motivate the real-life adaptive signal processing system designers to re-think their design from a self-timed perspective integrally at the algorithm, architecture, and circuit levels for potential system performance improvement.

References

[1] B. Widrow and S. D. Stearns, "Adaptive Signal Processing," *Prentice Hall*, 1985.
[2] S. Haykin, "Adaptive filter theory," *Prentice Hall*, 1996.
[3] N. R. Shanbhag and K. K. Parhi, "Pipelined Adaptive Digital Filters," *Kluwer*, 1994.
[4] T. Williams, "Self-Timed Pipelines (Chapter 9 in Design of High-Performance Microprocessor Circuits edited by A. Chandrakasan et al.)," *John Wiley & Sons*, 2000.
[5] J. Sparso and S. Furber, "Principles of Asynchronous Circuit Design: A Systems Perspective," *Kluwer Academic Publishers*, 2002.
[6] M. Singh, J. A. Tierno, A. Rylyakov, S. Rylov, and S. M. Nowick, "An adaptively-pipelined mixed synchronous-asynchronous digital FIR filter chip operating at 1.3 gigahertz," in *Proc. Eighth International Symposium on Asynchronous Circuits and Systems*, April 2002, pp. 84–95.
[7] A. Efthymiou and J. D. Garside, "Adaptive pipeline depth control for processor power-management," in *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Sept. 2002, pp. 454–457.
[8] T. E. Williams and M. A. Horowitz, "A zero-overhead self-timed 160-ns 54-b CMOS divider," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 11, pp. 1651–1661, Nov. 1991.
[9] T. D. Howell, W. L. Abbott, and K. D. Fisher, "Advanced read channels for magnetic disk drives," *IEEE Transactions on Magnetics*, vol. 30, no. 6, pp. 3807–3812, Nov. 1994.
[10] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *Computer*, vol. 27, no. 3, pp. 17–28, March 1994.
[11] Hitachi Global Storage Technologies, "Deskstar 120GXP OEM Specification v4.1," *http://www.hitachigst.com/tech/techlib.nsf/products/Deskstar_120GXP*, 2003.

[12] K. M. Chu and D. L. Pulfrey, "A comparison of CMOS circuit techniques: differential cascode voltage switch logic versus conventional logic," *IEEE Journal of Solid-State Circuits*, vol. 22, no. 4, pp. 528–532, Aug. 1987.
[13] T. E. Williams, "Performance of iterative computation in self-timed rings," *Journal of VLSI Signal Processing*, vol. 7, no. 1-2, pp. 17–31, Feb. 1994.
[14] D. C. Wong, G. De Micheli, and M. Flynn, "Designing high-performance digital circuits using wave pipelining: algorithms and practical experiences," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 1, pp. 25–46, Jan. 1993.
[15] W. P. Burleson, M. Ciesielski, F. Klass, and W. Liu, "Wave-pipelining: a tutorial and research survey," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 3, pp. 464–474, Sept. 1998.
[16] P. Kabal, "The stability of adaptive minimum mean square error equalizers using delayed adjustment," *IEEE Transactions on Communications*, vol. 31, no. 3, pp. 430–432, March 1983.
[17] G. Long, F. Ling, and J. G. Proakis, "The LMS algorithm with delayed coefficient adaptation," *IEEE Transactions on Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing]*, vol. 37, no. 9, pp. 1397–1405, Sept. 1989.
[18] 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 II: Analog and Digital Signal Processing*, vol. 40, no. 12, pp. 753–766, Dec. 1993.
[19] R. D. Poltmann, "Conversion of the delayed LMS algorithm into the LMS algorithm," *IEEE Signal Processing Letters*, vol. 2, no. 12, pp. 223, Dec. 1995.
[20] S. C. Douglas, Q. Zhu, and K. F. Smith, "A pipelined LMS adaptive FIR filter architecture without adaptation delay," *IEEE Transactions on Signal Processing*, vol. 46, no. 3, pp. 775–779, March 1998.
[21] K. Matsubara, K. Nishikawa, and H. Kiya, "Pipelined LMS adaptive filter using a new look-ahead transformation," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 46, no. 1, pp. 51–55, Jan. 1999.
[22] K. K. Parhi, "VLSI Digital Signal Processing Systems: Design and Implementation," *John Wiley & Sons*, 1999.
[23] R. O. Ozdag, M. Singh, P. A. Beerel, and S. M. Nowick, "High-speed non-linear asynchronous pipelines," in *Proc. 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, pp. 1000–1007, March 2002.
[24] S. Schuster and P. Cook, "Low-power synchronous-to-asynchronous-to-synchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 4, pp. 622–630, April 2003.
[25] G. De Micheli, "Synthesis and Optimization of Digital Circuits," *McGraw-Hill*, 1994.
[26] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, June 1989.
[27] R. D. Cideciyan, F. Dolivo, R. Hermann, W. Hirt, and W. Schott, "A PRML system for digital magnetic recording," *IEEE Journal on Selected Areas in Communications*, vol. 10, no. 1, pp. 38–56, Jan. 1992.
[28] A. A. Abidi, "Integrated Circuits In Magnetic Disk Drives," in *Proc. of the 20th European Solid-State Circuits Conference*, pp. 48–57, Sept. 1994.

Sizhong Chen received his B.S. and M.S. degrees in electrical engineering from Fudan university, China, in 1997 and 2000, respectively. He is pursuing his Ph.D. degree in electrical, computer and systems engineering department at Rensselaer Polytechnic Institute. His research interests include VLSI architectures for communication systems and high performance/low power circuits design. Currently he is working on MIMO system algorithm design and VLSI implementation.



Tong Zhang (S'98, M'02) received his B.S. and M.S. degrees in electrical engineering from the Xian Jiaotong University, Xian, China, in 1995 and 1998, respectively. He earned Ph.D. in electrical engineering at the University of Minnesota in 2002. Currently he is an assistant professor in electrical, computer and systems engineering department at Rensselaer Polytechnic Institute. His current research interests include design of VLSI architectures and circuits for digital signal processing and communication systems, with the emphasis on error-correcting coding, MIMO signal processing, and asynchronous VLSI signal processing.