To appear in *Proceedings of The Eighth IEEE International Symposium on Asynchronous Circuits and Systems ("Async2002"), April 8–11, 2002, Manchester, UK.*

An Adaptively-Pipelined Mixed Synchronous-Asynchronous Digital FIR Filter Chip Operating at 1.3 GigaHertz*

Montek Singh,[1] Jose A. Tierno,[2] Alexander Rylyakov,[2] Sergey Rylov,[2] and Steven M. Nowick[3]

[1]Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, NC 27514, USA.
[2]IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA.
[3]Dept. of Computer Science, Columbia University, New York, NY 10027, USA.
E-mail: *montek@cs.unc.edu*, {*tierno,sasha,sergeyr*}*@us.ibm.com*, and *nowick@cs.columbia.edu*

Abstract

A high-throughput low-latency digital finite impulse response *(FIR) filter has been designed for use in* partial-response maximum-likelihood *(PRML) read channels of modern disk drives. The filter is a hybrid synchronous-asynchronous design. The speed critical portion of the filter is designed as a high-performance asynchronous pipeline, sandwiched between synchronous input and output portions, making it possible for the entire filter to be dropped into a clocked environment. A novel feature of the filter is that the degree of pipelining is dynamically variable, depending upon the input data rate. This feature is critical in obtaining a very low filter latency throughout the range of operating frequencies.*

The filter was fabricated in a 0.18μ CMOS process. Resulting chips were fully functional over a wide range of supply voltages, and exhibited throughputs of over 1.3 Giga items/second, and latencies as low as four clock cycles. The internal asynchronous pipeline was estimated to be capable of significantly higher throughputs, around 1.8 Giga items/second. With these performance metrics, the filter has better performance than that reported for existing digital read channel filters.

1. Introduction

This paper presents the design of a high-speed pipelined digital filter chip. The chip is partly asynchronous and partly clocked. The speed-critical portion of the filter is implemented as an asynchronous pipeline, and the remainder, which is clocked, acts as the pipeline's environment.

A novel feature of the filter is that the degree of pipelining is *dynamically variable,* depending upon the rate of arrival of input data. In particular, from the viewpoint of the synchronous input and output environments, the asynchronous pipelined portion, which consists of nine pipeline stages, can naturally provide varying depths of pipelining by varying the number of data items present in the pipeline. At the lowest input rates, the asynchronous pipeline behaves similar to a block of flow-through combinational logic, with a latency of only a single clock cycle. As the input rate is increased, the behavior progressively changes to that of a pipeline that is one, two or more stages deep, with a corresponding latency of one, two or more clock cycles. This behavior is intrinsic to the asynchronous nature of the pipeline; no architectural modifications are needed. The advantage of this "adaptive pipelining" feature is that the chip naturally handles slow synchronous environments with a low latency penalty (in terms of number of clock cycles), yet can still accommodate fast synchronous environments as a highly-pipelined design. This adaptive nature was the main motivation for pursuing a mixed synchronous-asynchronous approach for the design of the filter. In contrast, a comparable fully clocked pipeline would be limited to a fixed pipelining depth, with a latency of nine clock cycles, irrespective of the input rate.

The filter is a real-world design intended for use in disk drive read channels [15]. The magnetic data that is picked up by the read head suffers from intersymbol interference, *i.e.,* adjacent bits of data overlap each other due to dispersion of the read pulses. This interference can be partially removed by passing the input stream through the filter, a process known as equalization. The filter output is then passed through a partial-response maximum-likelihood (PRML) detector [5, 2], which looks at a finite history of inputs to compute the likelihood of the current input being a "1" or a "0." The filter itself belongs to a larger category called finite impulse response (FIR) filters [7].

The design of the filter chip is an interesting case study for several reasons. First, the chip has two distinct timing domains, one clocked and the other asynchronous. Second, the filter pipeline uses a mix of static and dynamic logic function blocks: the asynchronous domain uses dynamic blocks, and the clocked one uses static logic. Also, as a real-world case study, the design exhibits a highly-varied datapath, ranging from 30 to 216 wires in width at varying points in the pipeline. Finally, as mentioned above, the depth of pipelining dynamically adapts to the input data rate.

The pipeline style used for the asynchronous portion of the filter is the high-capacity pipeline (HC) style introduced

*This work was carried out at the IBM T.J. Watson Research Center, NY; it was also supported by NSF ITR Award No. CCR-00-86007, NSF Award No. CCR-97-34803, a gift from Sun Microsystems, and a grant from the NY State Center for Advanced Technology (CAT).

in [12]. This style is for dynamic logic implementations, and provides the benefits of high throughput, low latency, and a 100% storage capacity (*i.e.,* each stage can hold a distinct data item). Several other high-speed asynchronous pipeline styles have been proposed recently [10, 14, 13], but each of these has disadvantages compared to HC. The pipelines of [10, 14] have the drawback of more complex timing constraints, requiring aggressive circuit techniques and much designer effort. HC pipelines, on the other hand, have much simpler implementation and less stringent timing requirements. The pipelines of [13], although comparable to HC in performance and ease of design, have only half the storage capacity.

The filter was fabricated in IBM's 0.18μ CMOS-7SF process. Resulting chips were fully functional over a range of supply voltages, and had throughputs of up to 1.32 Giga items/second. Interestingly, the filter throughput was limited by the synchronous portion of the chip; the asynchronous pipeline was actually capable of around 1.8 Giga items/second throughput. The fastest existing digital read channel filter, by Rylov et al. [9], has a peak throughput of 2.3 Giga items/second, in the same silicon process. However, the filter of [9] is a "half-rate" design, *i.e.,* it consists of two pipelines in parallel, each having a peak throughput of 1.15 Giga items/second. Therefore, the filter chip of this paper is effectively 15% faster than the fastest existing filter reported. However, the main novelty of the new filter is the dynamically variable pipeline depth, and, hence, a variable latency (as measured in clock cycles), which can adapt to varying input data rates.

The remainder of this paper is organized as follows. Section 2 gives background on read channel filters, and on high-capacity pipelines. Section 3 gives an overview of the filter architecture, and then Section 4 presents the detailed implementation. Section 5 discusses the operation of the filter, focusing on the adaptive pipelining feature. Performance analysis is provided in Section 6, and measurements of chip performance are given in Section 7. Finally, Section 8 gives conclusions.

2. Background

This section first provides background on read channel FIR filters, and then reviews the high-capacity (HC) asynchronous pipeline style.

2.1. Read Channel FIR Filters

Read channel filters are used in all magnetic and optical disk drives. The function of a read channel filter is to take the noisy data picked up by the read head, and turn it into a clean stream of "0" and "1" symbols. With ever-increasing data rates available from magnetic and optical media, high-speed read channel filters have become key to the design of modern disk drives. This subsection reviews the theory and implementation of commonly-used digital read channel filters.

2.1.1. Theory

A read channel filter belongs to the category of finite impulse response (FIR) filters [7]. In a digital FIR filter, the output at any given time, $Y(k)$, is a weighted sum of the p most recent inputs, $X(k), X(k-1), X(k-2)\ldots X(k-p+1)$:

$$Y(k) \;=\; \sum_{0\leq i<p-1} w_i \cdot X(k-i) \qquad (1)$$

where $w_0, w_1, w_2 \ldots w_{p-1}$ are the constant weights by which the inputs are weighted. Such a filter, with p weights, is said to be a "p-tap" filter. Each of the terms, $w_i \cdot X(k-i)$, is called a "partial sum."

2.1.2. Implementation: Distributed Arithmetic Style

Several implementations of a read channel filter are possible. For example, one could use one or more multiplier units to compute each of the product terms, and then use one or more adders to produce the final result.

A particular approach that is very well-suited for a high performance implementation is the *distributed arithmetic architecture* [8]. This approach does not use multiplier units. Instead, partial sums are precomputed and stored in a lookup table, indexed by the input data values. As a result, each multiplication can be performed quite fast, frequently in a single clock cycle.

Several techniques are used to keep the size of the lookup table manageable. First, the entire multiplication operation is bit-sliced. Second, within each bit slice, the input values are separated into two groups, one containing only the even-indexed values and the other only odd-indexed values, with each group having its own distinct lookup table. Finally, a particular data representation style is used which introduces symmetry into the lookup table, further reducing the amount of storage needed. Each of these techniques is now discussed in detail.

Bit Slicing. Suppose each input value has b bits. The expression of Equation 1 can be evaluated separately for each bit position in the input stream, and then the b individual results can be appropriately aligned and added together, to produce the same result as would be obtained if the computation of Equation 1 were performed directly with the b-bit inputs.

As a further optimization, the result of this entire expression can be precomputed and stored in a lookup table. The p most recent input values for that particular bit position form a p-bit word that is used as the address to access the table. Each lookup table will therefore have 2^p entries. For a 10-tap filter, this corresponds to a table with 1024 entries.

Partitioning. The size of the lookup table can be significantly reduced by partitioning the inputs into even- and odd-indexed groups. That is, starting with the current input, every other input belongs to the "even group," and the remaining inputs belong to the "odd group." The even and odd groups have their own lookup tables. Therefore, for a 10-tap filter, there are two lookup tables, each having a

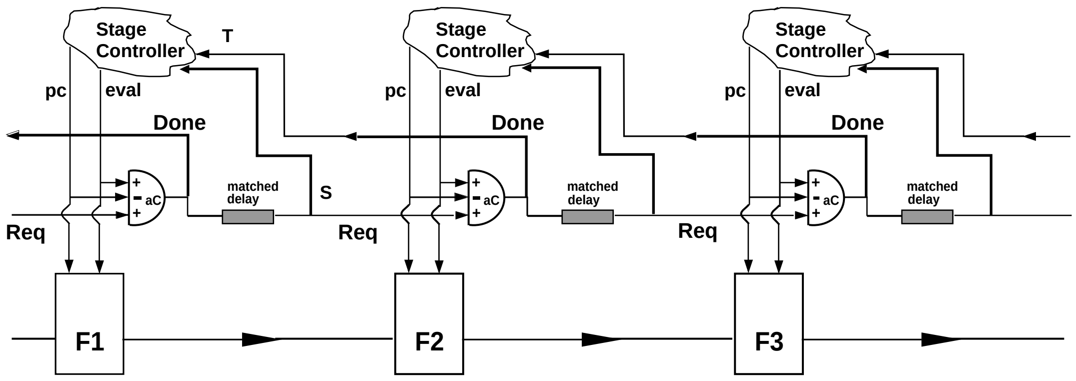


Figure 1. Block diagram of an HC pipeline

5-bit address word and only 32 entries. This represents a dramatic reduction in the size of memory required, from one 1024 entry table, to two 32 entry tables. However, there is a slight tradeoff: twice the number of partial sums are generated, requiring an additional adder stage to combine them.

Exploiting Symmetry. To further reduce the table size in half, a data representation scheme is used that makes the table symmetric. In particular, the *signed-digit offset binary notation* [8] is used, in which the symbols "0" and "1" stand for negative and positive exponents of 2. For example, in this notation, the 4-bit number "1001" stands for the value $8 - 4 - 2 + 1 = 3$. The advantage of this representation is that arithmetic negation is simply achieved by complementing each bit: "0110" stands for the value -3. An interesting feature of the filter equation (Equation 1) is that, if all the inputs are negated, the filter output is also negated. Consequently, when this representation is used, if two address words for the lookup table are bit-wise complements of each other, then the corresponding table entries will also be bit-wise complements of each other. Exploiting this symmetry, half of the table can be discarded.

2.2. High-Capacity Asynchronous Pipelines

This subsection reviews the pipelining approach adapted for the asynchronous portion of the filter chip. A class of pipelines, called *high-capacity* (HC), is used, which are targeted to dynamic logic implementations [12]. They are based on a novel protocol that maximizes the pipeline storage capacity by allowing every dynamic stage to hold a distinct data item. In contrast, in traditional latch-free asynchronous dynamic pipelines (*e.g.,* [18, 13]), alternating stages usually must contain "spacers," or "reset tokens," limiting the pipeline capacity to 50%.

The key idea in the HC approach is one of *decoupled control:* the pull-up and pull-down of the dynamic gates are made separately controllable. Therefore, the precharge and evaluate controls can both be simultaneously de-asserted, allowing the gate to enter a special "isolate phase"—between "evaluation" and "precharge"—in which its output is protected from further input changes. As a result, every pipeline stage can store a distinct data item, providing the capability of supporting 100% storage capacity. In addition, the decoupled control leads to increased overall pipeline concurrency which in turn directly results in a significantly increased throughput.

2.2.1. Structure

Figure 1 shows a simple block diagram of an HC pipeline. Each stage consists of three components: a *function block*, a *completion generator* and a *stage controller.* In steady-state operation, the function block alternately produces data tokens and reset spacers for the next stage, and its completion generator indicates completion of the stage's evaluation or precharge. The third component, the stage controller, generates the decoupled control signals—*pc* and *eval*—which control the function block and the completion generator.

HC pipelines use a single-rail bundled datapath [11, 1]. A control signal, *Req*, indicates arrival of new inputs to a stage. A high value of *Req* indicates the arrival of new data: the previous stage has completed evaluation. A low *Req* indicates the arrival of a spacer: the previous stage has completed precharge. For correct operation, a simple timing constraint must be satisfied: *Req* must arrive after the data inputs to the stage are stable and valid. This requirement is met by inserting a "matched delay" which is greater than or equal to the worst-case delay through the function block.

Function Block. Figure 2 shows one gate of a dynamic function block in a pipeline stage. In general, for a multiple output function block, there will be one such dynamic gate for each output.[1] The *pc* input controls the pull-up network and the *eval* input controls the "foot" of the pull-down network. Precharge occurs when *pc* is asserted low and *eval* is de-asserted low. Evaluation occurs when *eval* is asserted high and *pc* is de-asserted high. In HC pipelines, the two control signals, *pc* and *eval*, are separately generated and are decoupled. Therefore, when both signals are de-asserted, the gate output is effectively isolated from the gate inputs;

[1]For complex logic, where a single dynamic gate would be too large and slow, decomposition into a multi-level monotonic network is used.

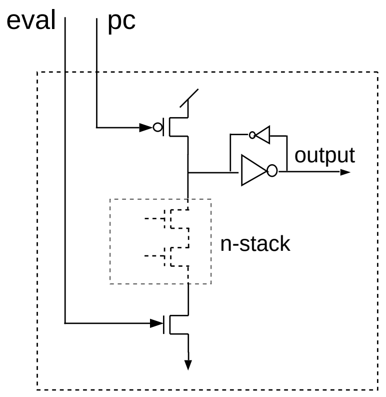


Figure 2. Details of a gate within an HC stage's function block

thus, it enters the "isolate phase." To avoid a short circuit, *pc* and *eval* are never simultaneously asserted.

Completion Generator. An asymmetric C-element, *aC* [3], is used as a completion generator. The *aC*'s output, *Done*, is set when the stage has *begun to evaluate, i.e.,* when two conditions occur: the stage has entered its evaluate phase (*eval* is high), and the previous stage has supplied valid data input (completion signal *Req* of previous stage is high). *Done* is reset simply when the stage is enabled to precharge (*pc* asserted low).

The *aC* element output is fed through the matched delay, which (when combined with the completion generator) matches the worst-case path through the function block. Note that, for extremely fine-grain or "gate-level" pipelines, the matched delay is often *unnecessary:* the *aC* delay itself often already matches the function block delay, so no additional matched delay is required.

Finally, the completion signal in turn is fed to three components: (i) the previous stage's controller, indicating the current stage's state, (ii) the current stage's controller (through the matched delay), and (iii) the next stage's completion generator (also through the matched delay).

Stage Controller. The stage controller produces the control signals for the function block and the completion generator. It receives two inputs—the delayed *Done* of the current stage, S, and the *Done* of the next stage, T—and produces the two decoupled control signals, *pc* and *eval*. Details of the stage controller's implementation will be discussed shortly, after presenting the desired protocol.

2.2.2. Protocol

An HC pipeline stage simply cycles through three phases, as shown in Figure 3. After it completes its evaluate phase, it then enters its isolate phase and subsequently its precharge phase. As soon as precharge is complete, it re-enters the evaluate phase again, completing the cycle.

The novelty of the approach is seen in the protocol which governs the interaction between stages. Unlike nearly all other pipeline approaches, HC has *only one explicit synchronization point* between stages. Once a stage $N + 1$ has completed its evaluate phase, it enables the previous stage N to perform its *entire next cycle:* precharge, isolate, and evaluate new data item. In contrast, the dynamic pipelines of [18, 13] use two explicit synchronization points between adjacent stages: one to enable the start of evaluation, and another to enable the start of precharge. As usual, there is also one implicit synchronization point: the dependence of stage's $N + 1$'s evaluation on its predecessor N's evaluation. That is, a stage cannot produce new data until it has received valid inputs from its predecessor. Both of the synchronization points are shown by the causality arcs in Figure 3.

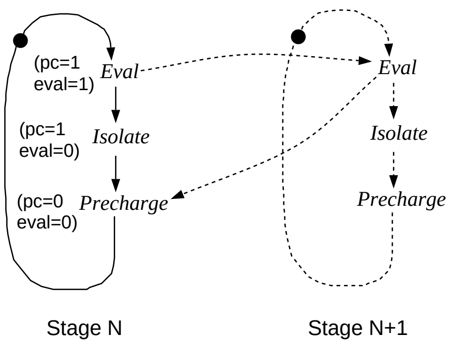


Figure 3. HC pipelines: Operation of, and synchronization between, adjacent stages

The introduction of the isolate phase is the key to the new protocol. Once a stage finishes evaluation, it immediately isolates itself from its inputs by a self-resetting operation *regardless* of whether this stage is allowed to enter its precharge phase. As a result, the previous stage can not only precharge, but even safely evaluate the next data token, since the current stage will remain isolated.

There are two benefits of this protocol: (a) higher throughput, since a stage N can evaluate the next data item even before stage $N + 1$ has begun to precharge; and (b) higher capacity for the same reason, since adjacent pipeline stages are now capable of simultaneously holding distinct data tokens, without requiring separation by spacers.

2.2.3. Stage Controller Implementation

Figure 4 shows a complete implementation of the stage controller. The implementation is very simple, with the two outputs—*pc* and *eval*—and an internal state variable, *ok2pc*, each implemented using a single gate. Figure 5 shows one complete pipeline stage along with its stage controller.

Note that the generation of the *ok2pc* signal is designed to be off of the critical path. While in Figure 4, *ok2pc* appears to add an extra gate delay to the control path to *pc*, this is not the case: the protocol allows *ok2pc* to be set in "background mode," so that *ok2pc* is typically set before T gets asserted. As a result, the critical path to *pc* is only one gate delay: from input T through the 3-input NAND gate, NAND3, to the output *pc*.

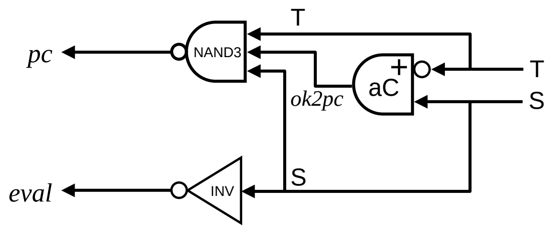


Figure 4. Stage Controller Implementation

2.2.4. Analytical Cycle Time and Latency

A complete cycle of events for stage N can be traced in Figure 1. From one evaluation by N to the next, the cycle consists of three operations: (i) stage N evaluates, (ii) stage $N+1$ evaluates, which in turn enables stage N's controller to assert the precharge input (*pc*=low) of N, (iii) stage N precharges, the completion of which, passing through stage N's controller, enables N to evaluate once again (*eval* asserted high).

Let the evaluation and precharge times for a stage be denoted by t_{Eval} and t_{Prech}, and the delay through the completion generator by t_{aC}. Then, the delay of step (i) is t_{Eval}, the delay of step (ii) is $t_{\mathrm{aC}} + t_{\mathrm{NAND3}}$, and the delay of step (iii) is $t_{\mathrm{Prech}} + t_{\mathrm{INV}}$. Here, t_{NAND3} and t_{INV} are the delays through the NAND3 and the inverter, respectively, of Figure 4. Thus, the analytical pipeline cycle time is:

$$T = t_{\mathrm{Eval}} + t_{\mathrm{Prech}} + t_{\mathrm{aC}} + t_{\mathrm{NAND3}} + t_{\mathrm{INV}} \qquad (2)$$

The forward latency through a stage, L_f, is simply the evaluation delay of the stage:

$$L_f = t_{\mathrm{Eval}} \qquad (3)$$

2.2.5. Timing Constraints

HC pipelines require several one-sided timing constraints for correct operation.

State Variable. The state variable *ok2pc* gets set once the current stage has evaluated, and the next stage has precharged (ST=10). Subsequently, T goes high as a result of evaluation by the next stage. For correct operation, *ok2pc* must complete its rising transition before T goes high:

$$t_{ok2pc\uparrow} \leq t_{\mathrm{aC}} + t_{\mathrm{INV}} \qquad (4)$$

In practice, this constraint is easily satisfied.

Precharge Width. For correct operation, an adequate precharge width must be enforced, *i.e.*, once precharge is asserted for a stage, it should not be de-asserted before that stage's precharge is complete. Suppose T just went high for stage 1. At this point, stage 1's NAND3 is triggered, thereby starting the precharge of stage 1 (in Figure 1). Concurrently, T will be reset after a path through stage 2's matched delay, stage 3's *aC* element, stage 2's NAND3 and *aC*, thereby de-asserting the output of stage 1's NAND3. Therefore, for correct precharge, the following must hold:

$$t_{\mathrm{NAND3}} + t_{\mathrm{Prech}_1} \leq t_{\mathrm{delay}_2} + t_{\mathrm{aC}_3} + t_{\mathrm{NAND3}} + t_{\mathrm{aC}_2} + t_{\mathrm{NAND3}} \qquad (5)$$

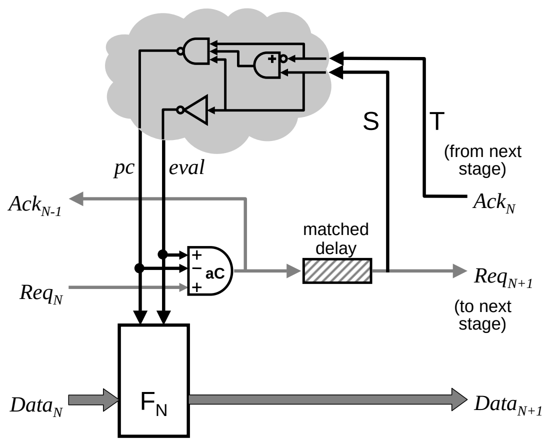


Figure 5. An HC pipeline stage: the function block along with its stage controller

For a general stage N, the constraint can be written as:

$$t_{\mathrm{Prech}_N} \leq t_{\mathrm{Prech}_{N+1}} + t_{\mathrm{aC}_{N+2}} + t_{\mathrm{NAND3}} \qquad (6)$$

Note that this timing constraint also gets exported to the left environment, requiring it to precharge reasonably fast. In practice, this constraint is easily satisfied as well.

3. Overview of Filter Architecture

Figure 6 shows the top-level architecture of the digital filter. The filter is a 10-tap 6-bit FIR filter using the distributed arithmetic architecture [8]. The figure gives a detailed view of one bit slice; as indicated, there are actually six such bit slices, stacked on top of each other. Data inputs enter from the left, and are processed by the filter as they flow to the right. The filter can be divided into three portions, from left to right. The leftmost portion is clocked, from the input side to the domino latches. The middle portion, from the XOR gates to the end of the carry lookahead adder, is asynchronous. Finally, the rightmost portion, consisting of an output latch, is again clocked.

The architecture of the filter is best understood by following the flow of data from left to right. As the stream of data enters the filter, it first passes through a shift register, which stores the most recent input values that are needed to compute the filter output. In particular, for a p-tap filter, for each bit, there is a p-place shift register that stores the most recent history for that bit. These stored input values are then multiplied by their respective filter weights. The multiplication is accomplished very efficiently by fetching precomputed results from a lookup table. In the figure, the lookup table is composed of two banks of registers containing the precomputed results—called even and odd partial sums—and two output multiplexors. The entire multiplication process is bit-sliced, with one slice for each bit of the input data. The result of the multiplications is a set of partial sums which are fed to the asynchronous portion of the filter pipeline for addition. The asynchronous portion

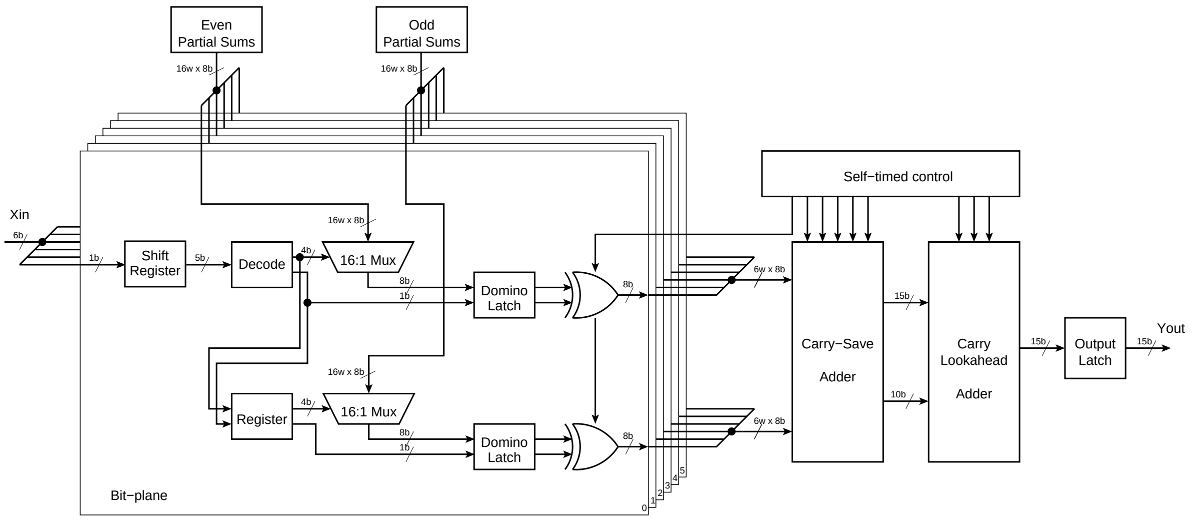


Figure 6. Top-level filter architecture

is a nine-stage pipeline that adds all of the partial sums together, and produces the result. Finally, this result is latched by a clocked latch and output to the right environment.

4. Filter Implementation

The FIR filter implementation is now considered in more detail. The synchronous and asynchronous portions of the chip are discussed separately, followed by a discussion of the interface between the two domains.

4.1. The Synchronous Portion

The synchronous portion of the filter consists of two parts, one at the input side of the filter, and the other at the output side.

4.1.1. The Synchronous Input Portion

This part receives the input to the filter. The input stream consists of data values which are six bits wide. A 10-slot shift register at the input side of the filter stores the 10 most recent data values. These stored input values are needed to compute the current filter output, which is a weighted sum of these values.

The multiplication of inputs by their respective filter weights is accomplished very efficiently by precomputing all possible products and storing them into a lookup table. The entire multiplication is bit-sliced, with one slice for each of the six bits in the input data. Therefore, within each bit slice, there are 10 input bits which together form a 10-bit address for accessing the lookup table.

The size of the lookup table is reduced by employing two techniques, as discussed in Section 2.1. First, partitioning is used: the 10-bit address is divided into two 5-bit addresses, one composed of only the even-index bits, and the other composed of the odd-index bits. Each of these two addresses has a distinct lookup table associated with it, as shown in Figure 6. To understand the filter operation with a partitioned lookup table, consider a simulation of partial sum lookup. The 10-bit pattern (after passing through the decoder unit) is used to generate separate groups of even- and odd-indexed bits. In particular, only the five even bits are used; they are forked to the even multiplexor as its select bits, and also to a clocked register where, after one clock cycle delay, they become the odd-index select bits to the bottom multiplexor, for the next clock cycle. Appropriate entries in the even and odd lookup tables are then selected and sent to the domino latches.

Finally, a second optimization is used: a *signed-digit offset binary notation* [8] is used to represent table entries and addresses, which enables the separation of the sign-bit from each address, further shortening the addresses to 4-bit words (see Section 2.1). As a result, the table size is dramatically reduced: two tables with only 16 ($= 2^4$) entries each are needed, as opposed to one table with 1024 ($= 2^{10}$) entries.

The lookup tables are implemented using registers and multiplexors, as shown in Figure 6. Each table has 16 registers, each of which can store an 8-bit entry, per bit slice. Each of the tables has a 16:1 multiplexor at its output, controlled by the 4-bit address word.[2] The odd-index address word is generated from the even-index address word by delaying it by one clock cycle.

The result of the multiplication is a set of products, called partial sums, that is sent to the asynchronous pipeline for addition, through the synchronous-asynchronous interface.

4.1.2. The Synchronous Output Portion

The right synchronous portion simply consists of a master-slave latch that receives the final result from the asynchronous pipeline and makes it available as the filter output.

[2]For faster decoding, the 4-bit address word was actually encoded using 9 wires: 8 wires represented the one-hot code [16] for three address bits, and the ninth wire represented the remaining fourth address bit.

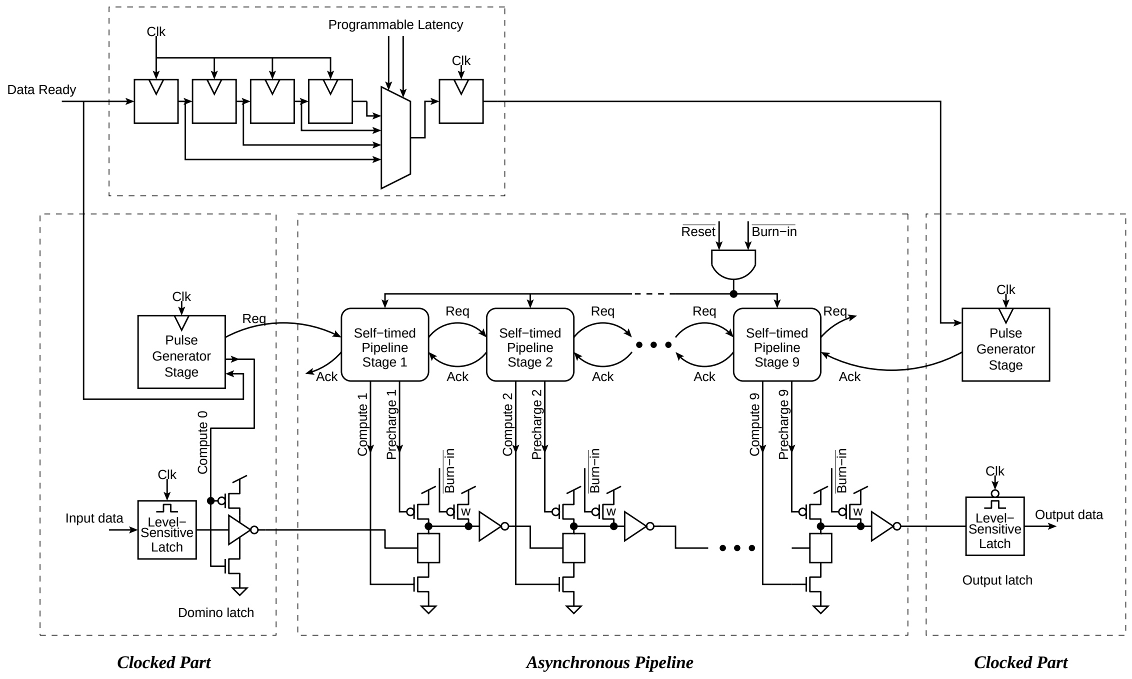


Figure 7. Asynchronous datapath and control with clocked interfaces

4.2. The Asynchronous Portion

The asynchronous portion of the filter consists of a pipeline that lies between the synchronous input and output portions, as shown in detail in Figure 7. The function of this asynchronous pipeline is to take the partial sums generated by the synchronous input portion, add them up to produce the final filter result, and send it to the synchronous output portion. The pipeline was designed using the high-capacity pipeline style [12].

The asynchronous datapath uses dynamic logic, and consists of nine stages. The first stage is a layer of XOR gates that restores the correct sign to the partial sums. The next five stages correspond to five layers of carry-save adders [4]. The last three stages implement a carry-lookahead adder [4]. Since both true and complement values of the data bits are needed to compute the XOR and addition functions, the entire datapath was implemented in dual-rail. The datapath is quite wide at the input to the first stage: 216 wires $(= (8 \text{ data bits} + 1 \text{ sign bit}) \cdot 2 \text{ (even and odd)} \cdot 6 \text{ (bit slices)} \cdot 2 \text{ (wires/bit)})$. The output of the last stage is a 15-bit result represented using 30 wires.

Interestingly, since the filter has a very fine-grain datapath, no explicit matched delays are required. The delay of each function block is matched by the completion generator's *aC* element itself, through appropriate device sizing.

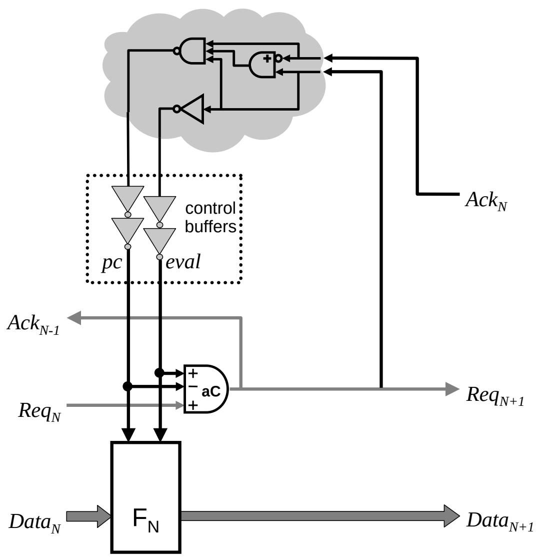
The self-timed control of a high-capacity pipeline, shown in Figure 5, needs a slight modification to handle the wide datapath of the filter. In particular, buffers must be inserted in order to amplify the control signals which are broadcast to the entire width of the datapath.

Two different versions of the control were designed, one more robust and the other faster, as shown in Figure 8. The two versions differ in the placement of the amplifying buffers. In the first version, Figure 8(a), the buffers amplify the control signals—*pc* and *eval*—for *both* the datapath as well as the completion generator. This version is very robust to variations in buffer delays because the completion signals are delayed by the same amount as the datapath. However, the buffers are on the critical path, thus increasing the pipeline cycle time. In the second version, Figure 8(b), the completion generators use control signals that are tapped off from before the buffers. As a result, the buffer delays are taken off of the critical path, resulting in a shorter cycle time. However, each stage's function block now lags behind its completion generator by an amount equal to the buffer delay. Consequently, for the pipeline to function correctly, *all* the stages throughout the pipeline are required to have comparable buffer delays.

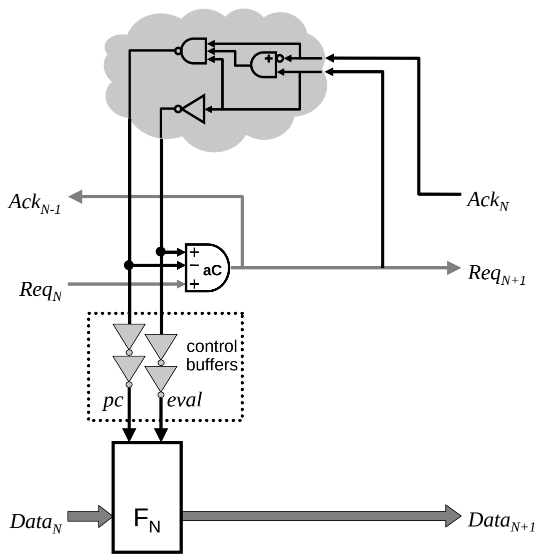
4.3. The Synchronous-Asynchronous Interface

The interface between the asynchronous and the synchronous portions of the chip must mediate certain differences in data representation and control sequencing. In particular, the asynchronous datapath uses dual-rail dynamic logic, whereas the synchronous portions of the chip use single-rail static logic. Moreover, the asynchronous pipeline communicates by means of local handshakes (using *req*'s and *ack*'s) at each end, whereas the synchronous portion uses global clocking.

Figure 7 shows the interfaces at either end of the asyn-



(a) safer but slower



(b) less robust but faster

Figure 8. Pipeline stage controls: two versions

chronous pipeline. At each interface, special latches are used to perform data conversion (from single-rail to dual-rail, and vice versa), and pulse generators are used to mimic the handshaking protocol use by the asynchronous pipeline.

At the input end of the pipeline, a special master-slave latch, referred to here as a *static-dynamic latch*, is used. The master portion is a standard transparent D-latch with single-rail inputs and complementary dual-rail outputs. The D-latch is controlled by the clock, *Clk*. The slave portion consists of a pair of dynamic buffers, one for each rail of the data, controlled by a pulse generator. The pulse generator emits a high-going pulse every time the clock goes low. As a result, on every downward clock transition, a new data item is launched into the asynchronous pipeline, along with the associated request, *Req*. The request is also forked off to a programmable synchronous delay line, shown in Figure 7, for use at the output end of the pipeline. The acknowledgment from the first stage of the pipeline, *Ack*, is simply ignored.

At the output end of the pipeline, a master-slave pair of synchronous D-latches is used to receive the computed result. Only the true value of the dual-rail output data is used; the complements are simply ignored. In addition, another pulse generator is used to produce the acknowledgment, *Ack*, for the last stage of the pipeline. The request from the pipeline is ignored. Instead, arrival of new valid data at the output of the pipeline is inferred from a delayed version of the *Req* associated with that data item. In particular, the input *Req* to the first stage of the pipeline is simply delayed by an integer number of clock cycles, and used in place of the output *Req* at the right end of the pipeline. It must be ensured, however, that the latency through the delay line is greater than the latency through the asynchronous pipeline; this constraint is easily met by making the delay line have programmable latency. An advantage of this approach is that the output of the asynchronous pipeline is re-synchronized to the clock without any issues of metastability.

5. Filter Operation

5.1. Performance Goals: Discussion

The filter is designed to work over a wide range of clock frequencies, because the input data rate to a read channel can vary greatly. In fact, as the disk read head moves from the innermost track to the outermost track, the data rate can vary by a factor of as much as 1:5. A separate analog circuit ("clock recovery" unit) is used to generate the clock for the digital filter, whose frequency and phase are synchronized with the input data stream.

While high throughput is an important requirement, an additional key design goal is also to have as low a latency as possible. The filter, along with the clock recovery unit, is part of a closed feedback loop that monitors the clock frequency and phase, and corrects any misalignment of clock with respect to input data. In order to ensure that the clock closely tracks the input data stream, this feedback loop must have a fast response time. Consequently, the filter, which is a critical part of the loop, must have a very low latency. This goal of low latency is achieved in the new FIR filter by a novel feature: adaptive pipelining.

5.2. Adaptive Pipelining: Operation

Adaptive pipelining is a characteristic of certain mixed synchronous-asynchronous systems, where the degree of pipelining is dynamically varied, depending upon the rate of arrival of input data. In particular, in the asynchronous portion, at low input rates, the data items will be widely separated, while at higher rates, they will be spaced closer together. As a result, from the viewpoint of the composite synchronous-asynchronous system, at slower clock rates, the *latency* (measured in terms of clock cycles) from the clocked input side to the clocked output side can be very few clock cycles. In contrast, at higher clock rates, the latency (again, measured in terms of clock cycles) can be much higher. Thus, while the asynchronous pipeline has a fixed number of stages (nine) and has roughly a fixed overall latency *in nanoseconds,* the *effective latency* as seen by the clocked output interface (now measured in clock cycles) can be highly varied.

In particular, at the lowest input rates, the asynchronous pipeline behaves similar to a block of flow-through combinational logic, with a latency of at most a single clock cycle. On every clock cycle, one data item is introduced into the asynchronous pipeline by the synchronous input portion of the filter. In the next clock cycle, that data item is removed from the output of the asynchronous pipeline by the synchronous output portion.

As the input rate is increased, the behavior of the asynchronous portion progressively changes to that of a pipeline that is one, two or more stages deep. At these clock rates, the latency through the asynchronous datapath is longer than one clock period, and, therefore, multiple data items will be present in the datapath at any given time. Accordingly, the programmable delay line, which helps interface the right end of the asynchronous pipeline with the synchronous portion of the chip, is set to one, two or more clock period delays.

In conclusion, from the viewpoint of the synchronous portion of the filter, the latency of the asynchronous portion (measured in terms of clock cycles) is dynamically variable, and this feature is taken advantage of to reduce the overall filter latency. This variable-latency behavior is intrinsic to the asynchronous nature of the pipeline, and cannot be easily achieved in fully synchronous implementations. As an example, consider a fully synchronous version of our nine-stage pipeline. This synchronous implementation will have a fixed latency of nine clock cycles (for a single-phase clock), or 4.5 cycles (for a two-phase clock). Unfortunately, the result can be a serious penalty: at low clock rates, these latencies can be inordinately large, thus degrading the performance of the clock recovery loop. In contrast, our asynchronous implementation has roughly a constant latency as measured in nanoseconds, thus enabling a fast response time at all clock frequencies.

5.3. Adaptive Pipelining: Comparison to Synchronous Approaches

There is one synchronous approach, however, that can provide an adaptively-pipelined operation similar to that of the asynchronous implementation: *wave pipelines* [19, 6]. In this approach, multiple waves of data are allowed at any time between two latches, thus allowing a variable number of data items. However, this approach requires much designer effort, from the architectural level down to the layout level, for accurate balancing of path delays (including data-dependent delays), and remains vulnerable to process, temperature and voltage variations. In contrast, the asynchronous implementation is significantly more robust, using instead a handshake protocol to maintain the integrity of data.

6. Performance Analysis

This subsection presents a theoretical analysis of the performance of the filter. Equations relating the maximum filter operating frequency to the number of data items in the pipeline are derived.

The filter performance is determined by two metrics: the maximum allowable throughput of the synchronous portion of the filter, and the maximum allowable throughput of the asynchronous pipelined portion of the filter. The observed performance will be limited by the lower of the two metrics. For a given voltage supply, the maximum throughput of the synchronous portion is fairly fixed. Thus, to simplify discussion, the synchronous portion and the synchronous-asynchronous interfaces are initially ignored from the analysis; their impact on throughput is discussed at the end of this section. The throughput of the asynchronous pipeline, however, can vary greatly, as explained below.

The throughput of the asynchronous pipeline is a function of the number of data items present in the pipeline. When the number of data items is small, the throughput is low, and the pipeline is said to be "data limited." On the other hand, when nearly every stage of the pipeline is filled with data items, the throughput is once again limited because empty stages, or "holes," are needed to allow data items to flow through the pipeline; in this scenario, the pipeline is said to be congested, or "hole limited."

Each of the two scenarios, data limited as well as hole limited, is analyzed in detail to derive the expressions for the throughput of the asynchronous pipeline.

Data Limited Operation. Suppose there is only one data item in the asynchronous pipeline at any given time. On every clock cycle, this data item is removed by the synchronous portion on the right side, and, simultaneously, a new data item is introduced into the pipeline by the synchronous left side. Clearly, for correct operation, the clock period, T, must be longer than or equal to the time it takes one data item to flow through the pipeline from left to right:

$$T \;\geq\; 9 \cdot L_f \tag{7}$$

where L_f is the forward latency through one stage of the nine-stage pipeline.

Similarly, if there are n data items in the asynchronous pipeline, then the latency of n clock cycles must be at least equal to the forward latency through the entire pipeline:

$$nT \;\geq\; 9 \cdot L_f \tag{8}$$

Hole Limited Operation. Suppose all of the nine stages of the asynchronous pipeline are holding distinct data items. At the next rising clock edge, the synchronous portion on the right side will consume the data item at the output of the pipeline, effectively injecting a hole at that end. This hole will percolate through the pipeline, from right to left, and arrive at the first stage of the pipeline after nine "hole latencies;" at this point, the pipeline is ready to accept a new data item. A hole latency, also called *reverse latency*, is denoted by L_r, and is equal to the time it takes for a hole to move from one stage to its immediately preceding stage. More formally, the reverse latency is defined as the time from the completion of precharge in a stage (arrival of a hole in that stage), to the completion of the subsequent precharge in the previous stage (movement of hole into the previous stage).[3]

For correct operation, the hole must arrive at the input end of the pipeline before the synchronous portion on the left side de-asserts the new data item (the domino latches precharge). Clearly, the de-assertion of the new data item occurs exactly a half clock cycle after the hole is injected at the right side of the pipeline. Therefore,

$$\frac{1}{2} \cdot T \;\geq\; 9 \cdot L_r \tag{9}$$

A similar equation can be derived for the case when there are less than nine data items in the pipeline. If n is the number of items in the pipeline, then the pipeline has $9 - n$ holes in it. Each of these $9 - n$ holes can be filled with new input data before a new hole injected into the right end of the pipeline is required to reach the left end of the pipeline. Therefore, Equation 9 can be generalized for the case of n data items:

$$(9 - n) \cdot T + \frac{1}{2} \cdot T \;\geq\; 9 \cdot L_r \tag{10}$$

Overall Upper-Bound on Filter Throughput. Equations 8 and 10 provide upper-bounds on the operating frequency of the filter, F, as a function of the number of data items in the asynchronous pipeline, n:

$$F \;=\; 1/T \;\leq\; \text{Min}\left(\frac{n}{9 \cdot L_f}, \frac{9\frac{1}{2} - n}{9 \cdot L_r}\right) \tag{11}$$

Figure 9 shows a plot of the maximum filter frequency versus the number of data items in the asynchronous pipeline. The rising portion of the curve represents the data limited region, where throughput rises linearly with the number of data items. The falling portion, similarly, represents the hole limited region, where throughput drops linearly with a decrease in the number of holes. The figure also shows a horizontal line, which corresponds to the longest local cycle time within the entire system [18]. In general, this horizontal line may either represent the maximum

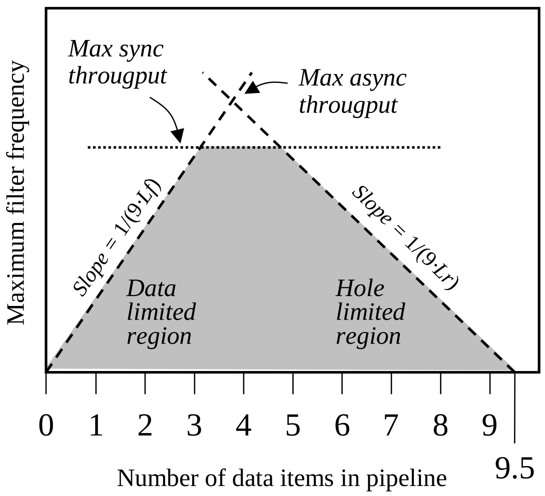


Figure 9. The upper bounds on the maximum filter frequency: shaded area represents operating region

speed of the slowest stage in the asynchronous pipeline, or the maximum operating rate of the filter's synchronous portion or the synchronous-asynchronous interfaces. The overall filter operation will always be constrained to lie under the canopy formed by the three curves.

In our particular filter design, the latencies through all of the asynchronous pipeline stages were fairly uniform. As a result, the local cycle times of all of the asynchronous stages were nearly the same. In this case, the maximum throughput potential of the asynchronous pipeline is given by the intersection of the rising and falling curves in Figure 9 [18]. The horizontal line, however, represents the maximum operating rate that can be sustained by the synchronous portion and the synchronous-asynchronous interfaces. This rate limits the overall filter throughput to a level lower than the maximum asynchronous throughput.

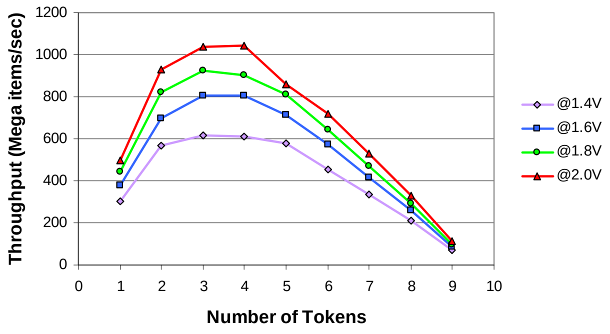
7. Experimental Results

Layout and Fabrication. The chip was laid out and fabricated using the IBM 0.18μ CMOS-7SF process with copper interconnect and 1.8V nominal voltage supply. Figure 11 shows the chip micrograph. The filter core occupies an area of 1.3x0.35mm^2.

The layout of the chip was part standard-cell and part full-custom. The entire synchronous portion was design using standard cells from the IBM ASIC SA-27E cell library. In the asynchronous portion, the datapath was implemented with full-custom dynamic gates. The asynchronous control used a mixture of standard cells (for basic gates) and full-custom cells (for C- and asymmetric C-elements). The interface between the synchronous and the asynchronous portions was designed mostly using standard cells.

Placement and routing were automated using the Silicon Ensemble tool. To simplify the task, the filter was divided

[3]From the analysis of Section 2.2, the reverse latency is easily calculated as $L_r = t_{\text{INV}} + t_{\text{aC}} + t_{\text{NAND3}} + t_{\text{Prech}}$.



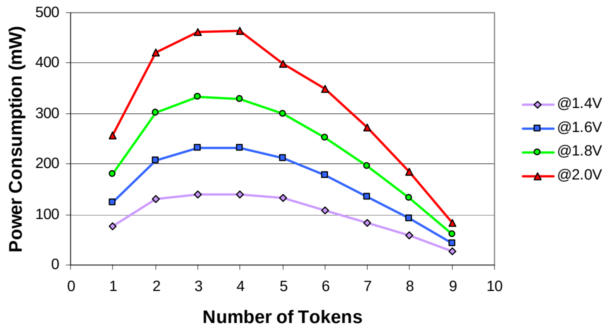


Figure 10. Filter throughput and power consumption as a function of the number of tokens

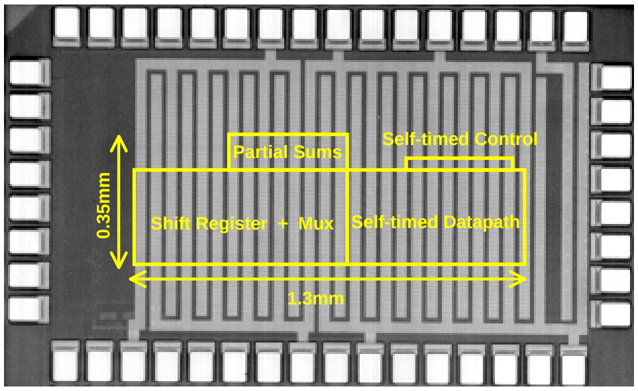


Figure 11. Micrograph of the fabricated chip

into eight parts: the self-timed control block, the XOR block, the carry-save adder, the carry-lookahead adder, the synchronous input and output portions, and the two synchronous-asynchronous interfaces. Next, each part was placed and routed individually using the automated tool. Finally, the tool was used for top-level place and route, to assemble all of the parts together. No resizing of gates was performed after place and route.

Two versions of the filter were designed, differing in the pipeline control circuits used. One version used the control circuit of Figure 8(a) which has fewer timing assumptions, at the cost of some throughput. The second used the circuit of Figure 8(b) which is faster, but has stronger timing assumptions. Both versions were fabricated side-by-side on the same chip. This section gives performance measurements only for the latter version. The performance of the conservative version was around 20% lower, as expected.

Testing. A level-sensitive scan design (LSSD) [17] approach was used to test the filter chip at low speed. A scannable shift register was built onto the chip to provide input data to the filter. An output multiplexor was placed on the chip to select one of the high-speed outputs of the filter, for observation on an oscilloscope. For testing the asynchronous datapath, an additional input (labeled “burn-in” in Figure 7) was used to convert the dynamic datapath into a pseudo-NMOS combinational circuit: both precharge and evaluate controls were de-asserted, and a weak pull-up was asserted. The chip was initialized, loaded with test data, and tested for correctness at a low clock speed. Subsequently, the clock speed was gradually increased until a failure was detected in the output data.

Measured Performance. Figure 10 shows plots of the measured maximum throughput, and the corresponding power dissipation. The graphs show the variation in throughput and power as the number of data items in the asynchronous portion of the filter pipeline is varied. The figure shows plots for a few representative voltages, although the chips were fully functional from around 1V to over 2.1V.

The performance measurements demonstrate the benefits of adaptive pipelining. At the lowest filter frequencies, the asynchronous portion appears externally as a block of flow-through combinational logic, with a single clock cycle latency. As the frequency is increased, the latency of the programmable delay line is increased to two, three or four clock cycles, increasing the depth of pipelining provided by the asynchronous portion.

In order to confirm the hypothesis of Section 6, the filter was also operated with more than four data items in the asynchronous datapath. Under normal circumstances, however, this mode of operation will not be used since it provides poorer latency for the same throughput as for four or fewer tokens. The observed performance exactly matches the behavior predicted by our theoretical model. As the number of tokens is increased from one, the pipeline throughput increases. However, beyond four tokens, the maximum throughput decreases because the pipeline becomes congested. Between two and four tokens, the performance levels off: in this region, the filter throughput is limited by the speed of the synchronous portions of the chip which cannot operate as fast as the native throughput of the asynchronous pipeline.

The best observed performance for the filter chip was around 1.1 Giga items/second, with three or four tokens and 2.1V power supply. The asynchronous pipeline, however, is capable of somewhat higher performance. The native throughput of the asynchronous portion is estimated by

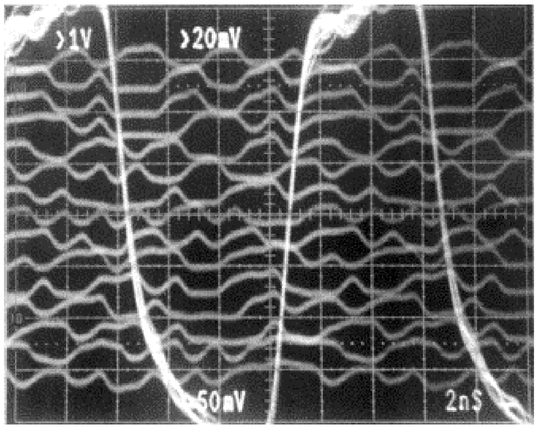


Figure 12. Oscilloscope waveforms showing filter operation at 1.32 Giga items/second

extrapolating the left and right ends of the curves of Figure 10(a), and noting their intersection (See Section 6). Using this technique, the maximum asynchronous throughput is estimated to be 1.5 Giga items/second at 2.1V. Several chip samples were thus tested. The fastest sample had an overall filter throughput of 1.32 Giga items/second at 2.1V, with the asynchronous portion estimated to be capable of throughputs up to 1.8 Giga items/second. Figure 12 shows the filter outputs as seen on an oscilloscope, along with a "sync" signal at 1/16th of the clock frequency.

8. Conclusions

This paper presented the design of an experimental digital FIR filter for use in the read channels of modern high-performance disk drives. The filter design was an interesting case study in hybrid synchronous-asynchronous design. The speed-critical portion of the filter was implemented as an asynchronous pipeline, obtaining a high throughput, yet very low latency. The synchronous portion formed a wrapper around the asynchronous pipeline, making it possible for the filter to be used in a clocked environment.

The recent high-capacity pipeline style [12] was used for the asynchronous pipeline portion of the filter chip. This style uses easy-to-satisfy one-sided timing constraints to achieve high throughput. Compared with other asynchronous techniques [10, 14], this approach required significantly less designer effort, as evidenced by the fact that there was little need for any post-layout gate resizing, even though placement and routing were totally automated. Further, the design exhibits a highly-varied datapath, ranging from 30 to 216 wires in width at varying points in the pipeline, thus demonstrating the scalability of the approach. Measured performance of fabricated chips easily met or exceeded design specifications.

References

[1] Al Davis and Steven M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.

[2] G. Forney. Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference. *IEEE Transactions on Information Theory*, 18:363–378, May 1972.

[3] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[4] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. Wiley, 1979.

[5] H. Kobayashi and D. Tang. Application of partial-response channel coding to magnetic recording systems. *IBM Journal of Research and Development*, 14:368–375, July 1970.

[6] A. Mukherjee, R. Sudhakar, M. Marek-Sadowska, and S.I. Long. Wave steering in YADDs: a novel non-iterative synthesis and layout technique. In *Proc. ACM/IEEE Design Automation Conference*, 1999.

[7] A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1989.

[8] D.J. Pearson, S.K. Reynolds, A.C. Megdanis, S. Gowda, K.R. Wrenner, M. Immediato, R.L. Galbraith, and H.J. Shin. Digital FIR filters for high speed PRML disk read channels. *IEEE Journal of Solid-State Circuits*, 30(12):1517–1523, December 1995.

[9] S. Rylov, A. Rylyakov, J. Tierno, M. Immediato, M. Beakes, M. Kapur, P. Ampadu, and D. Pearson. A 2.3 GSample/s 10-tap digital FIR filter for magnetic recording read channels. In *International Solid State Circuits Conference*, pages 190–191, February 2001.

[10] Stanley Schuster, William Reohr, Peter Cook, David Heidel, Michael Immediato, and Keith Jenkins. Asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz. In *International Solid State Circuits Conference*, February 2000.

[11] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[12] Montek Singh and Steven M. Nowick. Fine-grain pipelined asynchronous adders for high-speed DSP applications. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, pages 111–118. IEEE Computer Society Press, April 2000.

[13] Montek Singh and Steven M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 198–209. IEEE Computer Society Press, April 2000.

[14] Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, March 2001.

[15] Jose Tierno, Alexander Rylyakov, Sergey Rylov, Montek Singh, Paul Ampadu, Steven Nowick, Michael Immediato, and Sudhir Gowda. A 1.3 GSample/s 10-tap full-rate variable-latency self-timed fir filter with clocked interfaces. In *International Solid State Circuits Conference*, San Francisco, CA, February 2002.

[16] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[17] T. W. Williams and K. P. Parker. Design for testability–a survey. *Proceedings of the IEEE*, 31(1):18–22, January 1983.

[18] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.

[19] D.C. Wong, G. De Micheli, and M. Flynn. Designing high-performance digital circuits using wave-pipelining. *IEEE Transactions on Computer-Aided Design*, 12(1):24–46, January 1993.