Surfing: A Robust Form of Wave Pipelining Using Self-Timed Circuit Techniques

Brian D. Winters and Mark R. Greenstreet
Department of Computer Science
University of British Columbia
{*bwinters,mrg*}*@cs.ubc.ca*

Abstract

This paper presents "surfing," a novel variation of wave pipelining. In previous wave pipelined designs, timing uncertainty grows monotonically as data propagates through gates and other logic elements. Our designs propagate a timing pulse along with the data values, and our logic elements have delays that decrease in the presence of the pulse. This produces a "surfing" effect wherein events are bound in close proximity to the timing pulse. This produces a robust variant of wave-pipelining where timing dispersion is bounded regardless of the length of the pipeline. We demonstrate our approach with the design of a simple proof-of-concept chip.

1 Introduction

This paper presents a novel variation of wave pipelining called "surfing." In surfing rings and pipelines, a timing pulse is propagated along the pipeline, and logic elements are modified so as to have reduced propagation delays in the presence of this pulse. We show that when a few, simple conditions are satisfied, events in the data path will propagate in bounded temporal proximity to the timing pulse. This prevents timing uncertainties from accumulating in the data path, and we can propagate multiple, separate waves around a computing ring for an arbitrarily long time without interference. Alternatively, we can implement deep pipelines that support a large number of concurrent waves in flight.

Our approach actively *decreases* timing uncertainty while increasing performance. In deep submicron processes, statistical variations between transistors can result in large variations in the delays of geometrically identical circuits. This unpredictability impedes the use of high-speed circuit techniques that depend on controlled timing behavior in deep-submicron processes. To the best of our knowledge, our surfing technique is the first method presented that reduces timing uncertainty on a timescale finer than a pipeline stage.

The main contributions of our paper are:

- We show how modulating the delay of logic gates can create "event attractors" (Section 3). These attractors propagate faster than the non-surfing delay of logic elements, resulting in negative overhead. Furthermore, these attractors ensure timing uncertainties remain bounded, even in rings or arbitrarily long pipelines. This removes the need for latches and their associated overhead.

- We present a simple surfing ring (Section 4). We report Spice simulation results based on a model extracted from a layout of the ring cells (Section 7). Due to surfing, propagation delays of the XOR gates in the ring are about 3% faster than the corresponding, non-surfing, self-resetting domino implementations. Spice simulations indicate that the surfing ring is fast and robust in the presence of parameter variation and power supply noise.

- We describe a CMOS logic family that implements surfing (Sections 5 and 6). These circuits are a simple variation of existing, self-resetting domino designs [CC+91].

- We describe several pipeline configurations where surfing can be utilized (Section 8), including: surfing rings, surfing pipelines and rings used as components in larger synchronous designs, and simple networks of communicating, surfing rings.

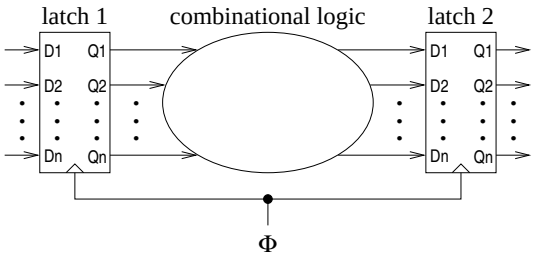

Figure 1: Synchronous Design

2 Wave Pipelining

Figure 1 shows a traditional synchronous design. Let the period of the clock, Φ, be P, and let $\delta_{\min}$ and $\delta_{\max}$ be the minimum and maximum delay from the inputs to the outputs of the combinational logic. For simplicity, we ignore latch set-up and hold times, latch propagation delays, and clock skew, noting that the qualitative observations that we make continue to hold in more detailed models. Classical synchronous design is based on the observation that if $\delta_{\max} < P$, then the values output by latch 1 on clock event n will propagate through the logic and be stable at the D inputs of latch 2 prior to clock event $n+1$. The minimum clock period is determined by the slowest path through the combinational logic. In general, reducing the clock period increases performance.

With careful control of the delays in the combinational logic, wave pipelined designs [BC$^+$98] can achieve clock periods less than $\delta_{\max}$. The key idea in wave pipelining is that multiple "waves" of computation can propagate through the combinational logic at the same time. The delays of the logic elements must ensure that waves these waves cannot overtake one another. Figure 2 illustrates such a design.

For each gate i, let $\delta_{\text{in},i,\min}$ denote the minimum delay from a clock event to the change of an *input* of gate i as a consequence of that clock event. Likewise, let $\delta_{\text{out},i,\max}$ denote the maximum delay from a clock event to the change the *output* of gate i. Let P be the clock period, and let

$$P_{\min} \quad = \quad \max_i \delta_{\text{out},i,\max} - \delta_{\text{in},i,\min} \tag{1}$$

If $P > P_{\min}$, then the output of each gate will settle for the current wave before the next wave affects any of the gate's inputs. $P_{\min}$ grows with the *uncertainty* in event times. If this uncertainty is sufficiently small, then the combinational logic can operate with multiple, distinct waves of concurrent computation.

We will first describe operation with two waves. Assume that $P_{\min} \leq \delta_{\max}/2 < \delta_{\min}$. Then, for any clock period P with $\delta_{\max}/2 \leq P < \delta_{\min}$, data output by latch1 on clock event n propagate through the logic and be available at the D inputs of latch 2 for clock event $n+2$. With two waves in flight, the combinational logic block operates with twice the throughput but the same latency as the classical synchronous design.

Figure 2 illustrates this operation, showing three waves shortly after a clock event $n+2$. The leftmost wave corresponds to data that propagated through latch 1 with clock event $n+2$. The middle wave shows data that propagated through latch 1 with clock event $n+1$. The one clock-cycle head start of wave $n+1$ ensures that it will arrive at latch 2 at the end of the current clock cycle without being overtaken by wave $n+2$. Finally, on clock event $n+2$, latch 2 acquired the results from the combinational logic for data that propagated through latch 1 with clock event n; this is the rightmost wave in the figure.

More generally, if $P_{\min} \leq \delta_{\max}/k < \delta_{\min}/(k-1)$ holds for some positive integer k, then the circuit can operate at a clock period P satisfying

$$\delta_{\max}/k \quad \leq \quad P \quad < \quad \delta_{\min}/(k-1) \tag{2}$$

This allows that circuit to operate with k times the throughput of classical, synchronous designs.

Timing uncertainties are the Achilles' heel of wave pipelined design. To minimize delay uncertainties, typical wave-pipelined designs arrange logic blocks into levels as shown in Figure 3. The uncertainty at the output of a latch is determined by skew and jitter of the clock and variations of the propagation delay of the latch. This uncertainty grows monotonically as each level adds its uncertainty to the accumulated uncertainty of the previous levels (assuming all paths are sensitizable). This accumulation of uncertainty presents a fundamental limit to wave pipelining. In particular,

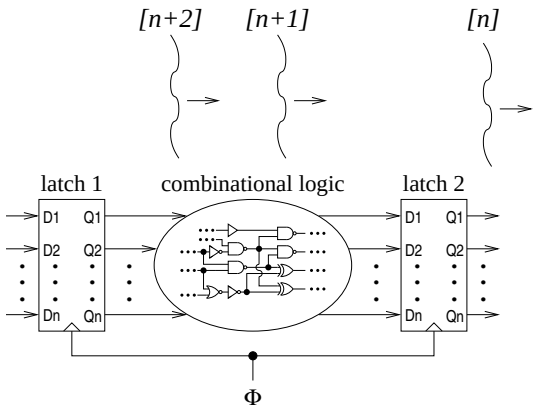

Figure 2: Wave Pipelining with Two Waves

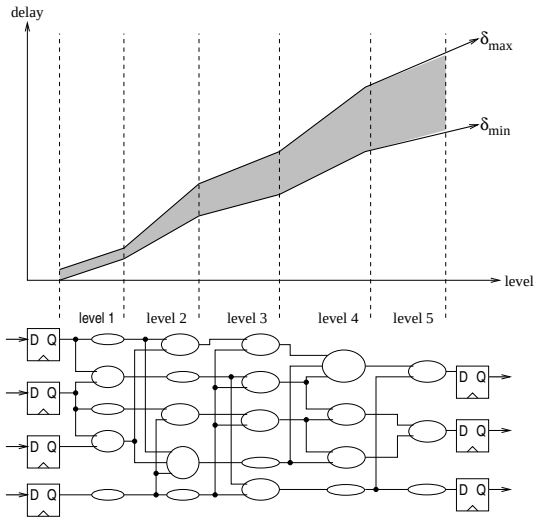

Figure 3: Timing Uncertainty

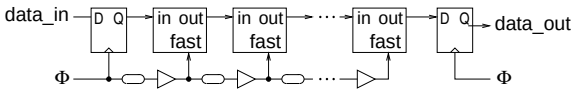

Figure 4: A Surfing Pipeline

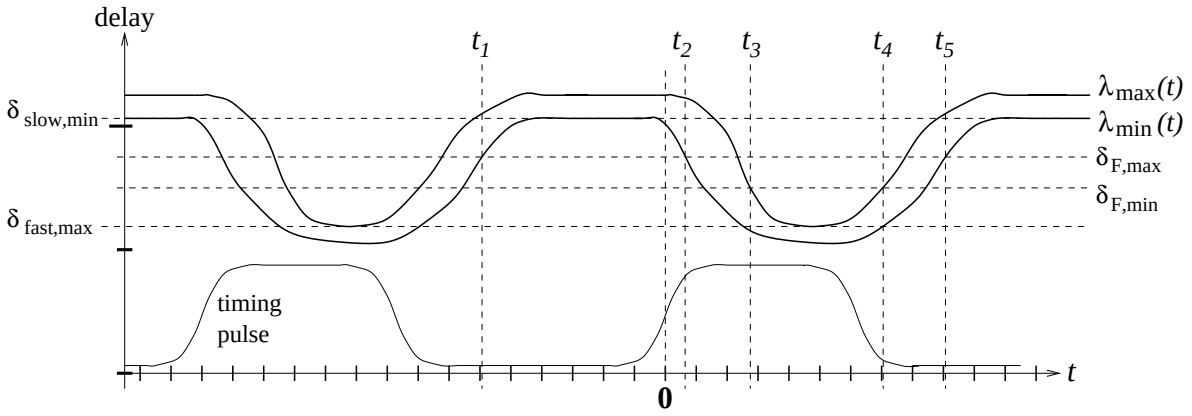

Figure 5: Timing for Surfing

Equation 2 implies that $k < \delta_{max}/(\delta_{max} - \delta_{min})$ and

$$P \quad > \quad \delta_{max} - \delta_{min} \tag{3}$$

Thus, uncertainty in the timing leads directly to a limit on the operating frequency. Furthermore, long datapaths that would benefit the most from latchless operation are those with the greatest timing uncertainty. In practice, most wave pipelined designs have only supported two to four waves. To achieve this, most wave pipelined designs employ logic restructuring and extra delay padding to minimize delay variation. This gives these designs greater latency than their classical equivalents, and the throughput is improved by a factor much smaller than the degree of wave pipelining. We now introduce "surfing" and show how it can tightly bound delay uncertainty.

3 Surfing

Consider again the synchronous circuit depicted in Figure 1. Due to timing uncertainties, the signals at the inputs of latch 2 may settle at different times. For proper operation, the latch must be triggered after the last data input has settled. Viewed from a slightly different perspective, latches bound timing uncertainty by slowing down events that propagate too fast. Recognizing this property of latches, Dooply and Yun [DY99] refer to latches as "roadblocks" when deriving timing constraints for self-resetting domino circuits.

Instead of *slowing down* the fast signals, we propose to *speed up* the slow ones. Thus, our pipelined circuits have lower latency than their unclocked combinational equivalents. This is the basis for our claim of negative overhead for our self-timed pipelines. This section describes how selective acceleration of slow paths provides a mechanism for increasing performance and bounding timing uncertainty.

Figure 4 depicts a simple surfing pipeline. Each logic block in the pipeline has a special input labeled fast. Asserting fast decreases the delay of the block. In section 5 we describe a surfing logic family based on self-resetting CMOS logic. For these circuits, active transitions are always low-to-high, and asserting fast shifts outputs slightly toward V_{dd} to reduce the time of any eventual upward transition. Surfing can be effected using other circuit-level phenomena as well. For example, body-bias modulation can be used to briefly reduce the delay of a gate at a cost of increased leakage currents.

The delay and buffer chain in Figure 4 generates the fast signal for each logic block. Let $\delta_{slow,min}$ be the minimum delay of a logic block when fast is not asserted, and let $\delta_{fast,max}$ be the maximum delay of a logic block when fast is

asserted. Let $\delta_{F,min}$ and $\delta_{F,max}$ denote the minimum and maximum delay between fast signals for consecutive stages of the pipeline. To ensure proper operation, we require:

$$\delta_{fast,max} < \delta_{F,min} < \delta_{F,max} < \delta_{slow,min} \tag{4}$$

When the constraints of Equation 4 are satisfied, events in the chain of logic blocks are attracted to the leading edge of the pulses of the fast signals. To see this, consider what happens if the outputs of a logic block change before fast is asserted for that block. In this case, the propagation delay for the next block will be at least $\delta_{slow,min}$ which is greater than $\delta_{F,max}$. Therefore, the timing pulse will catch up (or partially catch up) with the logic events. Conversely, if the output of a logic block changes after fast is asserted, then the propagation delay for the next block will be at most $\delta_{fast,max}$ which is less than $\delta_{F,min}$. In this case, the data events propagate faster than the timing pulse and eventually catch up.

As a metaphor, we view the propagation of data events as a swimmer in the ocean, and propagation of the timing events as a wave. Unassisted, the swimmer cannot swim as fast as the wave. However, there is a region on the leading edge of the wave where the swimmer is accelerated by the wave to travel at the same rate as the wave. Accordingly, we refer to this mechanism as "surfing."

To examine surfing in more detail, we need to consider the continuous variation of the propagation delay of the logic block under the influence of the timing pulse. We will say that an input event to a logic block is an enabling event if it is the last input required to enable a transition on at least one output of the block. Let $\lambda_{min}(t)$ be the minimum delay from an enabling input event to the corresponding output event if the input event occurs t time units after the arrival of the timing pulse. Likewise, let $\lambda_{max}(t)$ be the maximum delay from an enabling input event to the corresponding output event if the input event occurs t time units after the arrival of the timing pulse. Figure 5 shows $\lambda_{min}(t)$ and $\lambda_{max}(t)$ for a prototypical surfing logic block. We have also drawn the timing pulse in this figure to illustrate the relationship between the timing pulse and the varying delay of the logic block.

Figure 5 illustrates the timing properties of a surfing pipeline in greater detail. The bottom trace depicts the timing pulse (i.e. the fast signal) at a particular stage of the pipeline. The upper pair of solid curves show the maximum and minimum delays of the logic block for inputs that change at the time indicated on the horizontal axis – when the fast signal is high, delays are decreased compared with the delays when fast is low. The horizontal dashed lines show the quantities that appear in Equation 4. The tick marks on the axes indicate that the plot is drawn with much greater time resolution for the vertical axis than the horizontal one.

Equation 4, used the quantities $\delta_{slow,min}$, and $\delta_{fast,max}$. These are related to $\lambda_{min}(t)$ and $\lambda_{max}(t)$ by the relations:

$$\begin{aligned} \delta_{slow,min} &= \max_t \lambda_{min}(t) \\ \delta_{fast,max} &= \min_t \lambda_{max}(t) \end{aligned} \tag{5}$$

Now, define t_1, t_2, t_3, t_4, and t_5 as indicated below:

t_1: The time at which $\lambda_{min}(t)$ crosses above $\delta_{F,max}$ in response to the falling edge of the previous timing pulse.

t_2: The time at which $\lambda_{min}(t)$ crosses below $\delta_{F,max}$ in response to the rising edge of the current timing pulse.

t_3: The time at which $\lambda_{max}(t)$ crosses below $\delta_{F,min}$ in response to the rising edge of the current timing pulse.

t_4: The time at which $\lambda_{max}(t)$ crosses above $\delta_{F,min}$ in response to the falling edge of the current timing pulse.

t_5: The time at which $\lambda_{min}(t)$ crosses above $\delta_{F,max}$ in response to the falling edge of the current timing pulse.

In Figure 5, dashed vertical lines depict these times.

The key properties of surfing are:

- If the enabling input events for one stage arrive in the interval $[t_2, t_3]$ at one stage, then all input events will be in interval $[t_2, t_3]$ at all subsequent stages.
- If the enabling input events for one stage arrive in the interval (t_1, t_4) at one stage, then the input events at the next stage will be in a smaller interval contained in (t_1, t_4). The sequence of such intervals for successive stages converges to $[t_2, t_3]$.

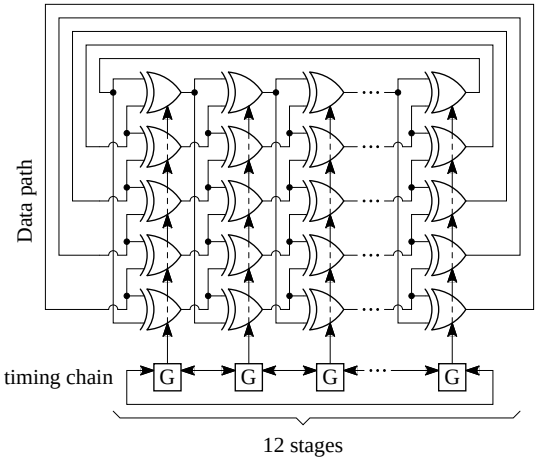

Figure 6: A 5-bit LFSR

In other words, the interval (t_1, t_4) is the "capture interval" for surfing. The interval $[t_2, t_3]$ is the steady state event uncertainty; we call this the "surfing interval." We omit the proofs of these properties due to space limitations, noting that they are straightforward. Events that arrive in the interval $[t_4, t_5]$ could surf with the current timing pulse, or they could "fall off" and slip to the next pulse. Events in this interval are timing violations that could give rise to metastable behaviors [CM73] and related malfunctions. In practice, the steady state interval and the violation interval are both much smaller than the capture interval – this gives rise to the robustness of surfing.

Note that surfing gates are *faster* when the timing pulse is asserted. With this negative overhead, performance is improved by implementing every gate on the critical timing paths as a surfing gate. By using surfing on every gate, timing uncertainty is minimized. Typically, such extreme pipelining is unacceptable for traditional, latched designs because of the latency overhead of the latches. In contrast with latched designs, surfing simultaneously lowers latency and bounds timing uncertainty.

4 Example

To demonstrate surfing, we are implementing a pipelined computation of the recurrence:

$$Q_{i+1}(u) \quad = \quad (Q_i(u)(u \oplus 1)) \bmod (u^5 - 1) \tag{6}$$

where $\oplus$ denotes addition modulo-2. This recurrence has a cyclic solution of length 15. By counting the number of times a wave propagates around the ring, we can verify the correctness of the computation.

The polynomial recurrence from Equation 6 is easily implemented using five surfing XOR gates per stage. Each of the Q_i is a fourth degree polynomial for which each coefficient is either zero or one. Multiplication by u modulo-$(u^5 - 1)$ corresponds to a circular shift of the coefficients. Exclusive-OR gates implement addition modulo-2. Figure 6 shows our implementation of this recurrence.

Our pipelined LFSR has twelve stages arranged in a ring. In addition to the XOR gates shown in Figure 6, we also include special "loader" and "unloader" cells that allow us to initialize and observer respectively the values propagating in the ring. We can initialize the ring to operate with either one or two waves in flight. By using a ring, we demonstrate how surfing allows pipelined compuation to proceed through an arbitrarily large number of steps without any latches or other storage elements. By operating the ring with two waves, we show how surfing maintains the separation of waves for arbitrarily long paths. In particular, the timing uncertainty for events in each wave remains smaller than the separation of the waves no matter how long the waves are allowed to propagate. This shows the key difference between our design and traditional wave-pipelining. In wave-pipelined designs, such waves would rapidly spread out and interfere with each other. Surfing overcomes this limitation.

The remainder of this paper describes the components of this design. Section 5 describes our modification to self-resetting domino logic to implement gates with the fast input required for surfing. Section 6 describes the self-timed pipeline that propagates the timing pulse.

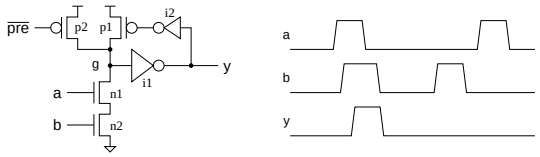

Figure 7: Self-Resetting Domino AND Gate

5 Surfing Circuits

For our proof-of-concept design, we modified self-resetting domino CMOS logic circuits to include an input to modulate the speed of the gate. Self-resetting domino circuits use pulses to represent boolean values. These circuits are very fast, but they are also sensitive to the arrival times of input pulses. As we describe below, pulse logic is particularly amenable for surfing because we can easily anticipate the direction of signal transitions. Conversely, surfing is well-suited for self-resetting domino, because the attraction of switching events to the leading edge of the fast signal provides the required pulse alignment.

5.1 Self-Resetting Domino

Self-resetting domino circuits [CC+91] are a variation of domino circuits [KLL82] where the precharge control signal for each gate is derived from the gate's output. As an example, Figure 7 shows a self-resetting domino, two-input AND gate. Transistors *p1* and *p2* are the precharge transistors. After precharge, node g is high. If the a and b inputs both go high, then node g is pulled low and output y goes high. If either a or b remains low, then the y output remains low as well. Asserted values in self-resetting domino are represented by pulses. After the output y goes high, inverter *i2* drives its output low, enabling transistor *p1* to precharge node g high which returns output y to a low value. Between input pulses, the precharge control signal, $\overline{\text{pre}}$ is low, and node g is held high by transistor *p2*. This maintains the level of g when the gate is operated at low frequencies and improves noise immunity.

Self-resetting domino circuits offer performance advantages because the only P-channel devices on the forward path are those for the output inverters of the gates. The switching networks that implement logic functions are constructed entirely of N-channel transistors with their higher carrier mobilities. The self-resetting operation allows the gate to precharge immediately after the completion of evaluation, minimizing the cycle time.

Input pulses to a multi-input self-resetting gate must have sufficient overlap to allow the N-channel network to fully discharge the precharged node (node g in Figure 7). Furthermore, input pulses must be short enough to avoid fights during the self-resetting precharge. These considerations place two-sided timing constraints on the operation of self-resetting domino circuits. Typically, blocks of self-resetting domino gates are arranged in levels, similar to those shown in Figure 3. This makes wave pipelining a natural technique for use in conjunction with self-resetting domino designs [CC+91]. As with other wave-pipelined designs, accumulated timing uncertainty limits the depth of logic in self-resetting domino designs. Surfing addresses this limitation by bounding timing uncertainties.

5.2 Preswitching for Self-Resetting Domino

To achieve surfing, we designed gates where asserting the fast input causes the output of the gate to shift slightly in the direction of making a transition. For self-resetting domino logic, active transitions are always in the low-to-high direction. Thus, we shift the low output of a gate slightly upwards in response to assertion of the fast input. We call this shift *preswitching*.

As an example of our approach, Figure 8 shows a self-resetting domino AND gate with a fast input implemented as described above. When the fast input is low, transistor p2 is conducting and functions as a keeper for the internal node g. To minimize the capacitance on node g, we implement p2 with a minimum width device. Accordingly, if inputs a and b are both high while fast is low, transistors n1 and n2 can overpower p2 and trigger an output pulse. In this situation, the hindering current sourced by p2 slightly delays the transition, an effect that increases the delay in the non-accelerated regime, therefore increasing the timing margins for surfing.

When fast is high, transistor p2 is turned off, and transistor n3 is conducting. If node g is high, then n3 pulls up against the N-channel pull-down of inverter i1. This raises the voltage of node y slightly above ground and decreases

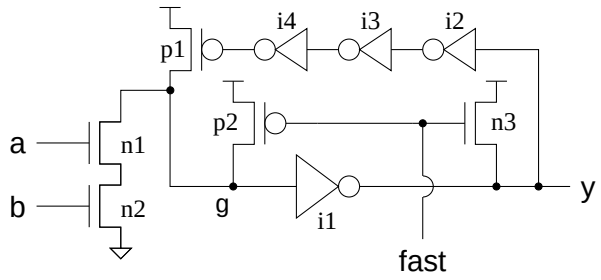

Figure 8: A Surfing AND Gate

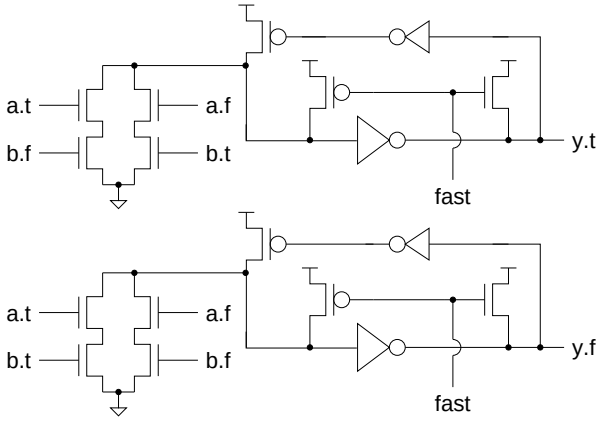

Figure 9: A Dual-Rail Surfing XOR Gate

the delay for a subsequent rising transition of y if node g later goes low. Otherwise, if node g is low, then inverter i1 is already pulling node y high. If node y is in transition, then the extra current from n3 simply accelerates the transition. Thus, rising transitions of y are faster when the fast signal is high than the rising transitions of an otherwise equivalent, non-surfing gate.

In practice, we draw transistor n3 with a similar shape factor to the N-channel pull-down of inverter i1. This design exploits the fact that N-channel devices make poor pull-ups. Because the fight pits an N-channel pull-up against an N-channel pull-down, our design also enjoys excellent device matching. Traditional, "five-corner" Spice simulations show robust operation over the full range of device parameters for the 0.18μ process that we are using (see Section 7).

The sizing of transistor n3 presents some interesting trade-offs. Increasing the width of this transistor pulls node y higher while waiting for node g to fall and further decreases the gate delay. By making the dip in the timing curve deeper, widening transistor n3 *increases* the robustness of the design to timing variations. On the other hand, pulling node y higher moves y closer to the switching threshold of the next gate. Thus, widening transistor n3 *decreases* the voltage noise margin of the design. This concern is exacerbated by our use of dynamic logic. If the inputs to the next gate are pulled higher than the threshold voltage for N-channel devices, then charge is drained from node g of that gate, even if that gate should not be enabled to switch. If this leakage persists long enough, node g will drop below the switching threshold of inverter i1, and the gate will produce a spurious output pulse.

Our simulations indicate that we obtain a fast and robust design when the width of transistor n3 is approximately 80% of the width of the pull-down in inverter i1. In our previous work using a 0.35μ CMOS process, we made the width of transistor n3 equal to the width of the pull-down in inverter i1 [WG02]. In our current 0.18μ technology, the N-channel pull-up is stronger, relative to its corresponding N-channel pull-down, than in 0.35μ, requiring a reduction in the size of the pull-up. We are currently exploring other circuit variations to better understand the trade-off between timing robustness and noise immunity.

In our example presented in Section 4, the logic consists primarily of XOR gates. The loader and unloader stages are multiplexors that interface the ring with serial shift registers. These multiplexors are implemented with circuits

```
do forever
    [1]  wait until stage[i-1] is full and
             stage[i] is empty
    [2]  transfer data from stage[i-1] to stage[i]
    [3]  set the status of stage[i] to full
    [4]  set the status of stage[i-1] to empty
od
```

Figure 10: A Simple Handshaking Protocol

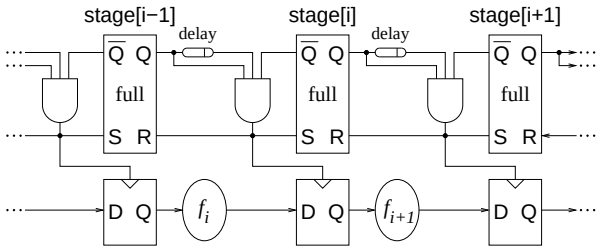

Figure 11: An asP* Pipeline

that closely resemble the XOR gate; so, we focus on the XOR implementation here.

Because domino logic is non-inverting, we use a dual-rail encoding: each signal, x, is produced in *true* and *false* versions with a pulse on the x.t wire indicating a value of true and a pulse on the x.f wire indicating a value of false. Figure 9 shows our dual-rail surfing domino XOR gate.

6 Self-Timed Control

In synchronous designs, state transitions are regulated by clock events. For example, on each rising edge of the clock, each latch copies the value on its input to its output to effect the state transition, and between clock events, combinational logic computes the next state. In contrast, self-timed designs control state transitions by using handshaking signals.

As an example, Figure 10 shows a simple handshaking convention for a self-timed pipeline, and Figure 11 shows one implementation of this protocol. Each time the condition for data transfer is satisfied (step *[1]*), steps 2–4 are performed. If these steps are performed sequentially, with the completion of each step enabling the start of the next, then the implementation is said to be *speed-independent*: the pipeline operates correctly regardless of the delays of the data transfer, set-full, or set-empty operations. Such designs are described, for example, in [Udd84, SS93, Mar96].

Alternatively, steps 2–4 can be performed as a single, parallel action. This is the approach taken in the implementation shown in Figure 11 based on the asP* protocol from [MJ+97]. Each stage has an R/S latch that is set to indicate that the stage is full and reset to indicate that the stage is empty. When stage i is empty and stage $i-1$ is full, the AND gate between the latches for the two stages can set its output high. This triggers the D flip-flop for stage i, acquiring the result from applying function f_i to the data from stage $i-1$. In parallel, the status of stage i is set to full, and stage $i-1$ is set to empty. The delay element on the middle input of the AND gate has a delay that is at least as large as the delay of the circuitry that implements f_i. This ensures that the result of the pipe stage has settled before triggering the D flip-flop.

There are many other possible approaches to asynchronous design. Overviews of various approaches are provided in [BS95, Hau95, DN97, Mye01].

6.1 GasP Backwards Control

In order for surfing to occur, our timing signals must propagate at about the same rate as or a little faster than our fastest domino element. Self-resetting domino logic is very fast. The forward latency is less than two inverter delays,

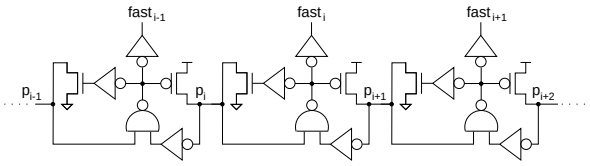

Figure 12: GasP Backwards Control

and when preswitching is being used the forward latency is only slightly greater than one inverter delay. We avoided using a simple inverter chain because pulses can be lost. A very fast self-timed chain is required. A self-timed style that we have found to be well suited to our purposes is GasP [SF01].

Figure 12 shows our control circuit. Operation is similar to the asP* protocol described above. Like the original GasP, adjacent stages communicate over a single wire. A high value on wire p_i indicates that stage i is full, and a low value indicates that the stage is empty. When wire p_{i-1} is high (i.e. stage $i-1$ is full) and wire p_i is low (i.e. stage i is empty), then both inputs of the NAND gate at stage i will go high, the output of the NAND gate will go low, causing wire p_{i-1} to go low (indicating empty) and wire p_i to go high (indicating full). In the original GasP designs, the low output of the NAND gate enabled the stage's latch to effect a data transfer. In our design, we invert the NAND gate's output to provide the fast signal for surfing.

In the configuration given by Sutherland and Fairbanks [SF01], data propagated in the opposite direction as in our design. Thus, their implementation has four inverter delays in the forward direction and two inverter delays in the backward direction. In the self-timed designs for which they created GasP, the extra forward latency matched the delay of their data paths with latches, and the smaller backward latency provided a small cycle time. Our designs have no latching overhead, and applying preswitching to every gate in the critical path improves both performance and timing margins. GasP pipelines closely resemble self-resetting domino designs: the two, series connected n-channel transistors of the NAND gate correspond to the pull-down network of a self-resetting domino XOR or multiplexor, and the p-channel pull-up to drive wire p_i corresponds to the pull-up transistor in the output inverter of a self-resetting domino gate. The NAND gate is enough simpler than the self-resetting domino gates to allow the timing pulse to propagate slightly faster than self-resetting domino without preswitching. Thus, we found the shorter, backward latency of GasP ideal for propagating our timing pulses.

A noteworthy feature of this design is the role of handshaking in the GasP timing chain. On the one hand, this handshaking is essential to maintain the separation of the timing pulses for two waves as described in the previous paragraph. Otherwise, one pulse would eventually overtake the other, causing the waves to collide and the ring to fail. On the other hand, surfing requires that these pulses propagate around the timing chain at the rate corresponding to the forward delay of the GasP stages; in other words, neither timing pulse should ever encounter a substantial delay waiting for an acknowledgement from the next stage. A closer examination of the handshaking reveals that when one pulse starts to approach the other, then the two inputs of the NAND gate for the latter pulse will change at nearly the same time. As described by Ebergen, Fairbanks, and Sutherland [EFS98], the switching time for the NAND gate is slightly higher when the input events are closely spaced than when they are further apart. This effect maintains the separation of the pulses while allowing both to propagate at a rate very close to the forward delay of a GasP stage (see Winstanley, Garivier, and Greenstreet [WGG02]).

7 Design and Simulation Results

We used the theory of Logical Effort [SSH99] as a starting point for optimizing our gates and matching the delays of the data path to those of the GasP backwards control. As described below, we validated our design by performing five-corner Spice simulations.

We chose to simplify our analysis of the propagation speeds of our GasP timing chain and our XOR gates, because the LFSR is homogeneous. We measured the delays of an XOR with fast at zero volts (no preswitching) and with fast at supply voltage (full preswitching). We sized the transistors in the GasP timing chain to achieve a forward delay in the timing chain that is between the slow and fast delays for the surfing XOR gates. The results of our Spice measurements are listed in Table 1.

We used the layout editor Magic [OH$^+$84] to create a physical layout for the cells of our design. We extracted

NMOS	PMOS	no fast	fast	Ave. GasP
typical	typical	74.0 ps	58.1 ps	71.6 ps
fast	fast	59.1 ps	46.6 ps	56.7 ps
fast	slow	71.7 ps	53.6 ps	70.1 ps
slow	fast	77.2 ps	62.9 ps	73.9 ps
slow	slow	93.8 ps	79.6 ps	90.7 ps

Table 1: XOR Delays at Process Corners

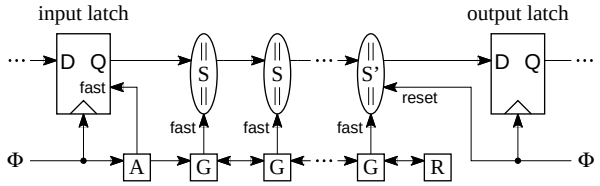

Figure 13: Synchronized Surfing Pipeline

capacitances for our Spice simulations from the layout. We did not create a layout of the entire torus, instead using Spice to connect the cells. We were unable to extract wire resistances; thus, they were omitted from the model. To simulate power supply noise, we placed a 3 nH inductor in series with the power supply.

Our tests were performed in Spice using parameters from a 0.18μ, 1.8 V process. All timing measurements are taken at the fifty percent point. Under typical process parameters, the average measured FO4 delay [HYS98] is 89.5 ps.

We tested both the speed and robustness of the design. We observed correct LFSR operation at all five process corners. Under typical process parameters the average interstage GasP delay is 71.6 ps. Stages fully reset in less than six interstage delays, allowing two waves in flight in our twelve stage pipeline. Our effective issue rate is over 2.3 GHz. At the fast/fast corner, our average interstage delay is 56.7 ps, for an effective issue rate of over 2.9 GHz.

Due to the homogeneity of the ring, inductive power supply noise is minimal during operation. This is an example of an oft-noted property of self-timed designs: because there is no global clock initiating switching activity, the current spikes associated with clock events are avoided. This reduces the problems of inductive power supply noise in self-timed designs. Using a self-timed timing chain, our ring enjoys the same advantage.

8 Configurations

The previous sections described the design and operation of a surfing ring. Such rings demonstrate the stability of surfing: waves can propagate for an arbitrarily large number of orbits of the ring without losing coherence. In this section, we describe alternative configurations for surfing circuits, showing how they can be applied in various situations, and describing the design considerations that arise.

8.1 Pipelines

A straightforward application of surfing is within a larger synchronous design. As shown in Figure 13, a surfing pipeline can be placed between two synchronous latches. In the figure, stages of the surfing data path are labeled "S;" stages of the GasP timing path are labeled "G;" the clock-to-GasP adaptor is labeled "A;" and the end-of-pipeline terminator is labeled "R." With each clock event, data from the "input latch" enters the surfing data path. Simultaneously, the clock event triggers the GasP pipeline through the adaptor "A," which produces the timing pulse that propagates down the GasP chain to keep the data wave coherent. The final surfing stage, S', is reset by the next rising edge of the clock; in other words, it is not self-resetting. This ensures that data output by the pipeline remains stable until it is acquired by the output latch.

A surfing pipeline as shown in Figure 13 provides a safer alternative to wave-pipelining [BC+98]. Surfing ensures that consecutive waves cannot overtake one another while improving the overall performance of the circuit. For example, many microprocessors use wave-pipelined L1 caches that support two waves: the cache access time is two

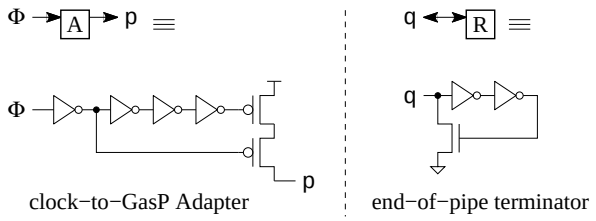

Figure 14: Pipeline Components

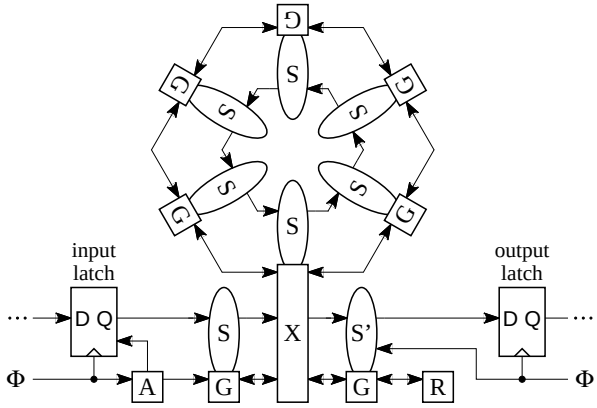

Figure 15: A Surfing Ring In a Synchronous Pipeline

clock periods, and a new access can be initiated every clock cycle. Such designs are amenable to surfing where the surfing stages could correspond to the row decoder, local-bit-line repeaters, the column selector, the tag comparator, etc.

As another example, we previously described a surfing multiplier design [WG02]. While that design is very simple, it illustrates how complex data-path functions can be implemented using surfing pipelines to improve performance. For example, surfing could be used to decrease the latency of floating point function units in a typical microprocessor.

Taking a closer look at the design from Figure 13, the backwards GasP stage, "G," is defined in Figure 12, and Figure 14 shows implementations of the clock-to-GasP adaptor, "A," and the end-of-pipeline terminator, "R." In response to a rising edge of the clock, Φ, the clock to edge adaptor, A, pulls up on shared line p. This triggers the GasP stage connected to A, which in turn returns p to a low level. Conversely, in response to a rising edge of signal q from the previous GasP stage, the end-of-pipeline terminator, R, pulls down on shared line q after a short delay, returning the shared line to its quiescent value. The input latch has self-resetting outputs to produce pulses as required by the surfing datapath. For datapaths that use dual-rail encodings as described in section 5, the latch can be designed to convert single rail inputs to dual-rail outputs. Noting that the output of the latch should change as adaptor A pulls up on the shared line, a surfing implementation can be used for the latch to improve performance and control timing from the beginning of the pipeline.

As shown in Figure 13, the last stage of the surfing pipeline provides persistent outputs to satisfy the timing requirements of the output latch. Thus, the designer must determine safe bounds for the range of possible arrival times of the data with respect to the clock. In many designs (such as the L1 cache example above) this uncertainty will be less that a clock period, in which case a simple latch is sufficient. Delays of nearly two clock periods can be accommodated by the skew tolerant latch presented by Chakraborty and Greenstreet [CG02]. To tolerate even greater variations, pipelines that mix surfing with traditional, asynchronous handshaking can be used as described in section 8.3 below.

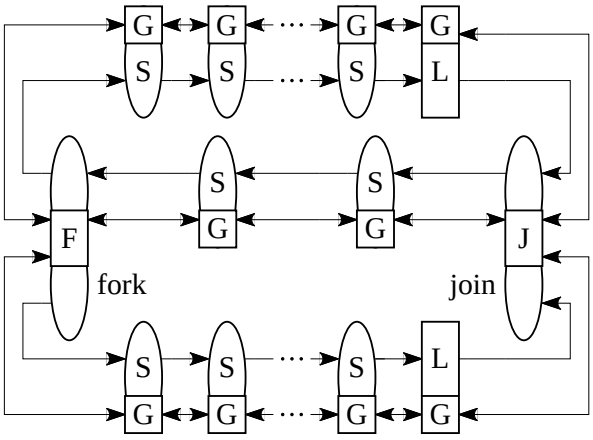

Figure 16: Coupled Rings

8.2 Rings within Pipelines

When a surfing datapath is used to implement an iterative computation, then a surfing ring may be employed within a synchronous pipeline as shown in Figure 15. This is the surfing equivalent of Williams' classical self-timed ring for SRT division [WH91]. In this configuration, the cell labeled "X" provides the interface between the pipeline and the ring. In a simple implementation, the ring is initially empty and waits to receive a wave from the left side of the pipeline. Upon receiving a wave, cell X propagates it into the ring. The iteration itself is controlled by the value of data in the wave. For example, these data may include a field that is used as an iteration counter; when the counter reaches a predetermined value, X propagates the result into the right side of the pipeline. As with the simple pipeline, the last stage of the pipeline provides persistent outputs for the output latch.

When using this configuration, drafting effects [WGG02] must be taken into account. When two or more events propagate through a self-timed pipeline in rapid succession, the outputs of each pipeline stage may not fully transition to the power or ground rails before the next event arrives. This reduces the stage-to-stage propagation delay for the subsequent stages. Thus, later events tend to "draft" earlier ones. Pre-switching can be used to control drafting [WGG02]. We expect that such techniques will be helpful when implementing rings such as the one shown in Figure 15.

8.3 Coupled Rings

So far, we have described simple linear pipelines and rings. While such structures can be used locally in a design to improve performance, large designs typically consist of many interacting pipelined pathways. A stage may receive inputs from or transfer its output to multiple stages, Figure 16 shows a simple instance of such a configuration where a "fork" cell sends its output to two separate pipelines, and the "join" cell combines the outputs of these two pipelines into a single stream. The fork operation presents little difficulty for a surfing design: the output of the forking stage is sent to both receiving pipelines, and a standard GasP fork [SF01] can be used in the timing path.

The join operation introduces a new complication. In general, it is impractical to match precisely the delays of the two pipelines that feed the join; thus, a datum from one pipeline may arrive substantially before the corresponding datum from the other pipeline. Surfing, however, does not provide a mechanism for holding a value. Instead, we use traditional, handshaking pipestages at the end of each surfing pipeline entering a join. In Figure 16, these handshaking stages are labeled "L" to indicate the presence of a true latch. This latch can be, for example, a GasP latch [SF01], a mousetrap latch [ST$^+$02], or any other design that supports the cycle time of the surfing pipeline. Note that the number of handshaking stages only needs to be sufficient to account for timing mismatches between the pipelines entering the join. With careful design, such mismatches should be kept small, and only one or two such latching stages should be needed at the end of any surfing pipeline.

9 Conclusions and Future Work

We have presented surfing pipelines and described their implementation using a simple variant of self-resetting domino. These pipelines achieve negative overhead: the latency of the pipeline is less than delay of an purely combinational logic implementation. Furthermore, the event attractors created by surfing support arbitrarily high degrees of wave-pipelining without latches or other road-blocks.

In our surfing pipeline, the delays of logic elements are modulated by timing pulses that propagate along with events in the pipeline's data path. We use self-timed, GasP pipelines to propagate these pulses. The use of a self-timed design was motivated by the high speed of GasP that is well matched to the propagation delays of surfing logic elements. By using self-timed handshaking, GasP ensures that pulses are not lost in the timing chain due to timing imbalances, while avoiding the need for elaborate pulse-shaping circuitry.

To demonstrate this approach, we have designed a small LFSR ring. Spice simulations indicate that the pipeline can operate with an effective issue rate of over 2.3 GHz with typical process parameters and 2.9 GHz at the fast/fast corner. The latency of the logic is reduced by 3% compared with the corresponding, purely combinational design. This shows that surfing does indeed achieve negative overhead as promised.

We have examined robustness issues and the design appears to be tolerant of process parameter variation and power supply noise. Our next step is to fabricate a larger version of the LFSR to experimentally verify the simulation results and perform further tests.

As mentioned in Section 5, our approach to surfing introduces a trade-off between timing margins and noise immunity. Clearly much more extensive analysis and testing must be done to examine the noise sensitivity of surfing domino logic. Furthermore, we are exploring variations of the basic surfing gate design presented in Section 5 to determine if designs that are even faster and/or more robust are feasible.

The design of the LFSR was simplified because its critical paths consist of chains of identical gates. We expect that surfing can be employed profitably in other structures as well. For such designs to be practical, we need to find practical design methodologies that will ensure sufficient matching of forward delay of the control chain to the propagation delay of the data path. Logical effort [SSH99] is an obvious place to start. Determining a consistent effort model for preswitched gates and developing the rest of a design methodology are key areas for future work.

Testing is another major issue that we have yet to address. For example, scan testing relies on stopping the device under test while loading or unloading the scan registers. While stopping can be relatively straightforward with latch based designs, surfing seems much less amenable to stopping: once a wave is launched, it traverses the entire pipeline. We are currently exploring ways to test surfing pipelines while data is in flight.

References

[BC+98] Wayne P. Burleson, Maciej Ciesielski, et al. Wave-pipelining: A tutorial and research survey. *IEEE Trans. on VLSI Systems*, 6(3):464–474, September 1998.

[BS95] Janusz A. Brzozowski and Carl-Johan HṠeger. *Asynchronous Circuits*. Springer, 1995.

[CC+91] Terry I. Chappell, Barbara A. Chappell, et al. A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture. *IEEE J. of Solid-State Circuits*, 26(11):1577–1585, November 1991.

[CG02] Ajanta Chakraborty and Mark R. Greenstreet. A minimalist source-synchronous interface. In *Proceedings of the 15th IEEE ASIC/SOC Conference*, September 2002. to appear.

[CM73] T.J. Chaney and C.E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. on Computers*, C-22(4):421–422, April 1973.

[DN97] A. Davis and S.M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Computer Science Department, University of Utah, September 1997.

[DY99] Ayoob E. Dooply and Kenneth Y. Yun. Optimal clocking and enhanced testability for high-performance self-resetting domino pipelines. In *Proceedings of the Twentieth Anniversary Conference on Advanced Research in VLSI*, pages 220–214, March 1999.

[EFS98] Jo C. Ebergen, Scott Fairbanks, and Ivan E. Sutherland. Predicting performance of micropipelines using Charlie Diagrams. In *Proc. 4th Intl. Symp. on Adv. Research in Asynchronous Circuits and Systems*, pages 238–246, April 1998.

[Hau95] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[HYS98] Mark Horowitz, Chih-Kong Ken Yang, and Stefanos Sidiropoulos. High-speed electrical signaling: Overview and limitations. *IEEE Micro*, 18(1):12–24, Jan./Feb. 1998.

[KLL82] R.H. Krambeck, C.M. Lee, and H.S. Law. High-speed compact circuits with CMOS. *IEEE J. of Solid-State Circuits*, SC-17:614–619, June 1982.

[Mar96] Alain J. Martin. A program transformation approach to asynchronous vlsi design. In Manfred Broy, editor, *Deductive Program Design*, NATO ASI. Springer, 1996.

[MJ+97] Charles E. Molnar, Ian W. Jones, et al. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 279–289. IEEE Computer Society Press, April 1997.

[Mye01] Chris J. Myers. *Asynchronous Circuit Design.* Wiley, 2001.

[OH+84] John K. Ousterhout, Gordon T. Hamachi, et al. Magic: A VLSI layout system. In *Proceedings of the 21th ACM/IEEE DAC*, pages 152–159, Albequerque, NM, June 1984.

[SF01] Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *Proc. 7th Intl. Symp. on Adv. Research in Asynchronous Circuits and Systems*, pages 46–53, April 2001.

[SS93] Jens Sparsø and Jørgen Staunstrup. Delay-insensitive multi-ring structures. *INTEGRATION*, 15(3):313–340, October 1993.

[SSH99] Ivan Sutherland, Bob Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers, Inc., January 1999.

[ST+02] Montek Singh, José A. Tierno, et al. An adaptively-pipelined mixed synchronous-asynchronous digital fir filter chip operating at 1.3 gigahertz. In *Proc. 8th Intl. Symp. on Adv. Research in Asynchronous Circuits and Systems*, pages 77–88, Manchester, UK, April 2002.

[Udd84] Jan T. Udding. *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Eindhoven University of Technology, 1984.

[WG02] Brian D. Winters and Mark R. Greenstreet. A negative-overhead, self-timed pipeline. In *Proc. 8th Intl. Symp. on Adv. Research in Asynchronous Circuits and Systems*, pages 32–41, April 2002.

[WGG02] Anthony J. Winstanley, Aurelien Garivier, and Mark R. Greenstreet. An event spacing experiment. In *Proc. 8th Intl. Symp. on Adv. Research in Asynchronous Circuits and Systems*, pages 42–51, Manchester, UK, April 2002.

[WH91] Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE J. of Solid-State Circuits*, 26(11):1651–1661, November 1991.