In *Proceedings of the IEEE Computer Society Annual Workshop on VLSI, April 27–28, 2000, Orlando, Florida.*

Fine-Grain Pipelined Asynchronous Adders for High-Speed DSP Applications*

Montek Singh and Steven M. Nowick

Department of Computer Science
Columbia University, New York, NY 10027
{montek,nowick}@cs.columbia.edu

Abstract

A new asynchronous pipeline scheme (called LP_{HC}), and two new pipelined asynchronous adder implementations, are introduced for high-throughput applications such as DSP's for multimedia processing. The pipeline scheme is targeted to dynamic datapaths. A novelty of the approach is that it uses decoupled control for pull-up and pull-down stacks. The adders are pipelined at the gate-level and achieve very high throughput: 930–1023 million additions per second in a 0.6μ CMOS process. These results are expected to scale to several Gigaoperations per second in more modern technologies.

1. Introduction

In this paper, two new asynchronous adder designs are introduced. The designs are pipelined at a very fine granularity to achieve high throughput. The implementations are based on two pipelining schemes: one scheme ($LP_{SR}2/1$) was recently introduced [8], while the second (LP_{HC}) is new.

Asynchronous circuits [9] fit naturally with the current trend towards system-on-a-chip for several reasons. First, asynchronous design avoids issues related to the distribution of a global clock: wasteful clock power, unmanageable clock skew and much design effort. Second, when pipelining is pushed to the granularity of a single gate (*gate-level pipelining*), design of the high-speed clock itself becomes extremely challenging. Finally, the inherent flexibility of asynchronous components, which can interface with varied environments at different rates, makes these circuits attractive for reusable components and IPs. Asynchronous design has recently attracted industry, including the use of fully-asynchronous 80C51 microcontrollers in commercial Philips' pagers (Myna and Fiori), and recent initiatives announced by Motorola,[1] and Sharp, Videonics and Cadence.[2]

The pipelines in this paper are part of a larger class which we call *lookahead pipelines (LP)*. The key strategy is one of *anticipation*: reacting to, or initiating, certain critical events earlier by using a more flexible communication between pipeline stages. In the first protocol, $LP_{SR}2/1$ [8] (recently introduced, but without any adder designs), a stage communicates not only with the next stage, but with *two stages ahead*. The second *high-capacity* protocol, called LP_{HC}, is new. Using a decoupled control of precharge and evaluate, every pipeline stage is able to store a distinct data item. In contrast, in traditional asynchronous dynamic pipelines, alternating stages usually must contain "spacer" tokens. In each case, high throughput is obtained.

A particular focus is on dynamic logic because of its high speed and small area. Dynamic pipelines can also be designed without explicit latches between pipeline stages; with careful control sequencing, the gates themselves can function as implicit latches. In spite of these advantages, however, there is a lack of asynchronous pipeline designs which take full advantage of dynamic logic. This paper attempts to fill this void.

Initial SPICE simulations of our 32-bit ripple-carry adders indicate throughputs of 930 million additions/second for an $LP_{SR}2/1$ implementation, and 1023 million additions/second for an LP_{HC} implementation, in a 0.6μ CMOS process. We anticipate that these results will scale to several Gigaoperations/second in more modern technologies.

The paper is organized as follows. Section 2 provides some initial background on Williams' asynchronous PS0 pipelines, and Section 3 reviews our recent $LP_{SR}2/1$ pipelines. Section 4 then introduces the new LP_{HC} pipelines and Section 5 describes the new pipelined adder designs. Results are presented in Section 6, and Section 7 gives conclusions.

Related Work. Several synchronous pipelines have been proposed for high throughput applications. In *wave pipelining*, or "*maximum rate pipelining*," multiple waves of data are allowed at any time between two latches [12, 3, 5]. However, this approach requires much design 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. Other approaches include *skew-tolerant domino* [2, 1] and *self-resetting circuits* [6, 1]. All of these styles have partial asynchronous behavior (e.g., precharge control or waves of data). They require complex timing constraints which are difficult to verify, lack elasticity and still require high-speed global clock distribution.

A number of asynchronous pipelines have also been proposed (see [8] for more details). Many suffer from significant performance overheads, and few are optimized for dynamic logic. Two of the fastest styles will not function correctly with a dynamic pipeline unless there are explicit latches [4, 13]. In addition, the former has complex timing assumptions which are not explicitly formalized; in fact, an early version was unstable due to timing issues.

*This work was supported by NSF Award No. CCR-97-34803.

[1] See http://motorola.com/SPS/MCORE/press_19oct99.html.

[2] See http://www.digitalproducer.com/pages/videonics_first_to_use_sharp_ele.htm.

2. Background: Williams' PS0 Pipeline

This section gives some brief background on Williams' PS0 dual-rail asynchronous pipeline [10, 11]. This pipeline is significantly different from ours, with a dual-rail datapath (instead of our single-rail) and a more conservative protocol (instead of our more timing-optimized ones). Yet, our design decisions and protocol optimizations can be more easily understood in the context of this approach.

2.1. PS0 Pipeline Structure

Fig. 1 shows Williams' PS0 pipeline. Each pipeline stage is composed of a dual-rail function block and a completion detector. The completion detectors indicate validity or absence of data at the outputs of the associated function block.

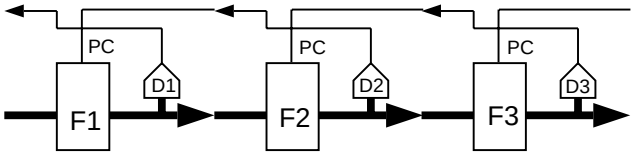

Figure 1. Block diagram of a PS0 pipeline

Each *function block* is implemented using dynamic logic. The precharge/evaluate control input, PC, of each stage is tied to the output of the next stage's completion detector. Since a precharge logic block can hold its data outputs even when its inputs are reset, it also provides the functionality of an implicit latch. Therefore, a PS0 stage has no explicit latch. Fig. 2(a) shows how a dual-rail AND gate, for example, would be implemented in dynamic logic; the dual-rail pair, f_1 and f_0, implements the AND of the dual-rail inputs a_1a_0 and b_1b_0.

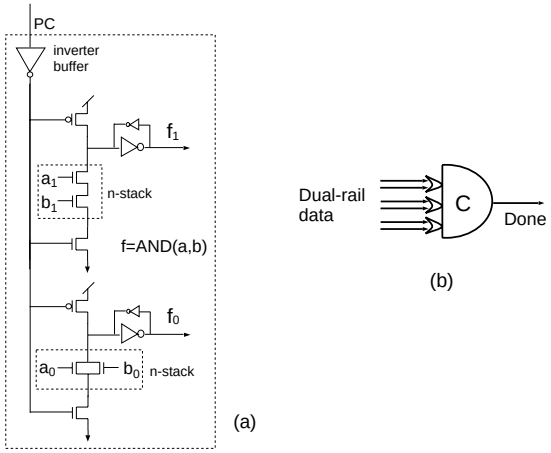

Figure 2. (a) A dual-rail AND gate in precharge logic, and (b) a dual-rail completion detector

The *completion detector* at each stage signals the completion of every computation and precharge. Validity, or non-validity, of data outputs is checked by OR'ing the two rails for each individual bit, and then using a C-element to combine all the results (Fig. 2(b)). A C-element [10] is a basic asynchronous state-holding element. The output of an n-input C-element is high when all inputs are high, is low when all inputs are low, and otherwise holds its previous value. It is typically implemented by a CMOS gate with a series stack in both pull-up and pull-down, and an inverter on the output (with weak feedback inverter atttached to maintain state).

2.2. PS0 Pipeline Protocol

The sequencing of pipeline control is quite simple. Stage N is precharged when stage $N+1$ finishes evaluation. Stage N evaluates when stage $N+1$ finishes reset. (Of course, the actual evaluation will commence only after valid data inputs have also been received from stage $N-1$.) This simple protocol ensures that consecutive data tokens are always separated by reset tokens or spacers.

The complete cycle of events for a pipeline stage is derived by observing how a single data token flows through an initially empty pipeline. The sequence of events from one evaluation by stage 1, to the next is: (i) Stage 1 evaluates, then (ii) stage 2 evaluates, then (iii) stage 2's completion detector detects completion of evaluation, and then (iv) stage 1 precharges. At the same time, after completing step (ii), (iii)' stage 3 evaluates, then (iv)' stage 3's completion detector detects completion of evaluation, and initiates the precharge of stage 2, then (v) stage 2 precharges, and finally, (vi) stage 2's completion detector detects completion of precharge, thereby releasing the precharge of stage 1 and enabling stage 1 to evaluate once again. Thus, there are six events in the complete cycle for a stage, from one evaluation to the next.

2.3. PS0 Pipeline Cycle Time and Latency

The complete cycle for a pipeline stage, traced above, consists of 3 evaluations, 2 completion detections and 1 precharge. The analytical pipeline cycle time, T_{PS0}, therefore is:

$$T_{\text{PS0}} = 3 \cdot t_{Eval} + 2 \cdot t_{CD} + t_{Prech}$$

where, t_{Eval} and t_{Prech} are the evaluation and precharge times for each stage, and t_{CD} is the delay through each completion detector.[3]

The per-stage forward latency, L, is defined as the time it takes the first data token, in an initially empty pipeline, to travel from the output of one stage to the output of the next stage. For PS0, the forward latency is simply the evaluation delay of a stage:

$$L_{\text{PS0}} = t_{Eval}$$

3. High-Throughput Pipelines: $\text{LP}_{\text{SR}}2/1$

This section presents $\text{LP}_{\text{SR}}2/1$, an asynchronous pipeline style which we recently introduced [8]. In this style, unlike Williams', an *early evaluation* protocol is used, in which a pipeline stage receives control information not only from the subsequent stage, but also *from its successor.* As a result, the pipelines have shortened cycle times.

[3] To simplify the presentation, this paper will sometimes assume that all the pipeline stages have the same evaluation delay and the same precharge delay, and that all the completion detectors are equally fast. More realistic HSPICE simulations are presented in the Results section.

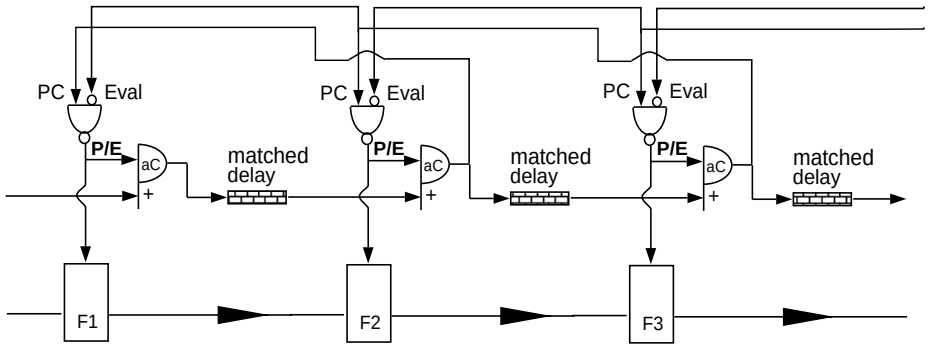

Figure 3. Block diagram of an $LP_{SR}2/1$ pipeline

The pipeline name indicates that 2 of 3 events in the stage's cycle are in the evaluate phase, and 1 event falls in the precharge phase; "SR" indicates single-rail. (More correctly, these designs are not always single-rail, since two rails are sometimes needed in dynamic logic to generate unate functions. However, in all cases, unlike Williams' design, no dual-rail completion detectors are used: only matched delays.)

3.1. Pipeline Structure

Fig. 3 shows the block diagram of an $LP_{SR}2/1$ pipeline. Like PS0, each stage has a function block and a control block. The function block alternately evaluates and precharges. However, unlike PS0, no dual-rail completion detector is used. Instead, we use a common *bundled data* scheme: the control block generates a *bundling signal* to indicate completion of evaluation (or precharge) [9]. The signal passes through a *matched delay* timed to equal or exceed the worst-case latency of the dynamic function block (evaluate and precharge). The resulting *Done* output is the stage's completion signal, indicating a valid input to the next stage. Note that for the special case of gate-level pipelines, the matched delay is usually unnecessary: the stage delay is often matched by the control block.

The main novelty is that an $LP_{SR}2/1$ stage, unlike a PS0 stage, now has two control inputs. The stage receives an input not only from the subsequent stage (PC), but also from its successor (EVAL). This second input enables early evaluation; it is the key to achieving a shorter cycle time.

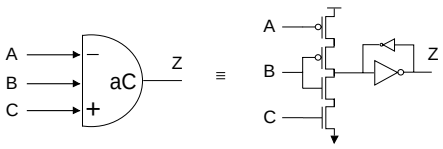

Figure 4. Asymmetric C-element notation

An *asymmetric C-element, aC* [10], is used to implement the completion detector. Fig. 4 shows an example. The notation indicates the *set* (+) and *reset* (-) enabling conditions. The remaining unmarked inputs are conditions for both set and reset. In the given example, $B = C = 1$ is the set condition, $A = B = 0$ is the reset condition, and all other input states result in hold. Arbitrary set and reset conditions can be specified, as long as they are disjoint.

In the pipeline of Fig. 3, the *aC* asserts the bundling signal (high) when two conditions hold: (i) the stage is in the evaluate phase, and (ii) valid input has arrived (signaled through matched delay input). The bundling signal is de-asserted low when only one condition holds: the stage is in its precharge phase.

3.2. Pipeline Protocol and Performance

We now present the synchronization conditions between stages. As in PS0, *stage N precharges when $N+1$ completes evaluation*. However, unlike PS0, stage N will start its next evaluation early. The idea is to allow stage N to evaluate as soon as stage $N+1$ has *started* precharging, instead of waiting for it to *complete* precharging. This idea can be used because a dynamic logic stage entering the precharge state is insensitive to changes on its inputs.[4] Similarly, stage $N+1$ starts to precharge once $N+2$ has finished evaluating. Therefore, our new early evaluation condition is: *stage N evaluates when $N+2$ completes evaluation.*

The pipeline structure directly implements these conditions. The evaluate phase is enabled when EVAL is asserted high (from $N+2$), or PC is de-asserted low (from $N+1$), or both. The former condition occurs when stage $N+2$ completes its evaluation: this is the "early evaluate" signal. The latter condition is identical to the conservative evaluation signal of PS0; its role will be explained below.

The evaluate phase ends, and the precharge phase begins, when two conditions hold: (i) stage $N+1$ has completed evaluation (as in PS0: PC is asserted high), and (ii) stage $N+2$ has completed precharging (EVAL is de-asserted low). The NAND gate in Fig. 3 combines these two conditions.

Interestingly, during LP2/1's evaluate phase, the early EVAL signal from stage $N+2$ may be *non-persistent:* it may be de-asserted low even before stage N has had a chance to evaluate its new data! The result could be a malfunction due to glitchy control. However, one-sided timing constraints are imposed below to ensure a correct evaluate phase: PC=low will always arrive in time to *takeover* control

[4] In general, this property is true of "fully-controlled" or "footed" dynamic logic, which we use.

of the evaluate phase, which will then be maintained until stage N has completed evaluating its inputs (as in PS0).

A complete cycle of events, from one evaluation of stage F_1 to the next, can be simulated in Fig. 3. Assume no matched delays are required for the gate-level pipeline, *i.e.* the *aC* completion detector already matches the stage's delay. The simulation consists of three events: (i) stage F_1 evaluates, (ii) stage F_2 evaluates and, through the NAND, asserts the precharge control of stage F_1, then (iii) stage F_1 precharges. Concurrently, after completing step (ii), we also have (iii)' stage F_3 evaluates and, through the NAND, asserts an "early evaluate" of stage F_1, thus terminating precharge and initiating the next evaluate phase (in the next step). Thus, the cycle time is:

$$T_{\mathrm{LP_{SR}2/1}} = 2 \cdot t_{Eval} + t_{aC} + t_{\mathrm{NANDB}}$$

where t_{Eval} is the evaluation delay of a stage (gate), and t_{aC} and t_{NANDB} are the delays of the *aC* element and NAND gate, respectively. The latency of a stage is: $L_{\mathrm{LP_{SR}2/1}} = t_{Eval}$.

3.3. Special Issues

The pipeline uses two optimizations that take advantage of the innate property of dynamic logic. The first is aimed at reducing the cycle time; the second is aimed at decreasing latency.

The first optimization is to produce an *early done* signal: to tap the *Done* signal to the previous stage from before the matched delay, instead of after the matched delay (see Fig. 3). For footed dynamic logic, it is safe to indicate completion of precharge as soon as the precharge cycle begins: during precharge, the stage is effectively isolated from changes at its inputs. Similarly, completion of evaluation can be indicated soon after the stage begins to evaluate on valid inputs; at this point, the outputs are effectively isolated from a reset of the inputs.

The second optimization is to allow an *early precharge-release*. In dynamic logic, a function block can be precharge-released before new valid inputs arrive. Once inputs finally arrive, the block (and corresponding *aC* completion element) will start evaluating. Thus, in our design, precharge release of the function block is decoupled from the arrival of the inputs.

3.4. Timing Constraints

One-sided timing constraints must be satisfied for correct operation. We found, in practice, that these timing constraints are easily satisfied. Below is a brief summary; for more details, see [8].

Precharge Width. $\mathrm{LP_{SR}2/1}$ pipelines have a shorter precharge phase than PS0 pipelines since the start of the evaluation phase is advanced. A minimum precharge width must be enforced. Considering the earlier simulation, the precharge width of stage N is bounded by the completion detector delay in stage $N+2$, which terminates the precharge with the next "early evaluate" signal. In practice, the precharge phase benefits from the extra inverter delay which the EVAL=high signal must go through at the inputs of the NAND gate.

$$t_{Prech_N} \leq t_{aC} + t_{\mathrm{INV}}$$

In our experience, this constraint is easily satisfied.

Safe Takeover. For correct operation in the evaluation phase, the "takeover" signal from stage $N+1$ (PC=low) must arrive at the inputs of the NAND gate before the non-persistent EVAL from stage $N+2$ is de-asserted low. This requirement ensures a glitch-free evaluation phase when early evaluation is used.

An equation can be derived assuming that stage $N+2$ has just completed evaluation, which starts the early evaluation of state N (EVAL=1). Stage $N+1$ will produce the desired takeover signal, PC=low, from a path through its NAND gate and *aC* element. Concurrently, EVAL will be de-asserted low once $N+2$ precharges, *i.e.* from a path through $N+3$'s *aC* element, and $N+2$'s NAND gate and *aC* element, and finally, an inverter to N's NAND gate. Thus, to maintain uninterrupted evaluation, the takeover should arrive at least a setup time, t_{setup}, before EVAL is de-asserted:

$$t_{\mathrm{NANDB}} + t_{aC} + t_{\mathrm{setup}} \leq t_{aC} + t_{\mathrm{NANDB}} + t_{aC} + t_{\mathrm{INV}}$$

Assuming all stages are similar, this constraint becomes:

$$t_{\mathrm{setup}} \leq t_{aC} + t_{\mathrm{INV}}$$

This constraint is also easily satisfied. Hold time and other constraints are discussed in [8].

4. High-Capacity Pipelines: $\mathrm{LP_{HC}}$

This section presents our second pipeline design, $\mathrm{LP_{HC}}$, which is targeted toward *high storage capacity*. This new pipeline is capable of storing as many distinct data tokens as the number of stages. In contrast, conventional latch-free dynamic pipelines, such as PS0 and $\mathrm{LP_{SR}}$2/1, only have half the storage capacity: for correct operation, a "reset spacer" must separate adjacent data tokens.

There has been another recent asynchronous pipeline approach which uses novel latch structures to improve the storage capacity [7]. However, the throughput of this scheme is worse than that of Williams' PS0, though it is still an improvement over a more conservative PC0 scheme [10].

4.1. Overview

Our new $\mathrm{LP_{HC}}$ pipeline combines both asynchronous and self-resetting features. A novelty is that it *decouples* the control of pull-up and pull-down. A dynamic gate is now controlled by two separate inputs, *pc* and *eval*.[5] Using these signals, unlike traditional approaches, a stage is driven through three distinct phases in sequence: *evaluate*, *isolate* and *precharge*. In the isolate phase, a stage holds its outputs stable irrespective of any changes at its inputs. As a result, adjacent pipeline stages are capable of storing distinct data items.

The remainder of this section presents the structure, protocol, implementation, and performance and timing analysis of $\mathrm{LP_{HC}}$ pipelines.

[5] Note that the *pc* and *eval* signals of this section are different from the PC and EVAL signals of $\mathrm{LP_{SR}}$2/1 pipelines. Whereas PC and EVAL were first combined using a NAND gate and then used inside the dynamic gate, *pc* and *eval* are here directly used as two separate gate inputs.

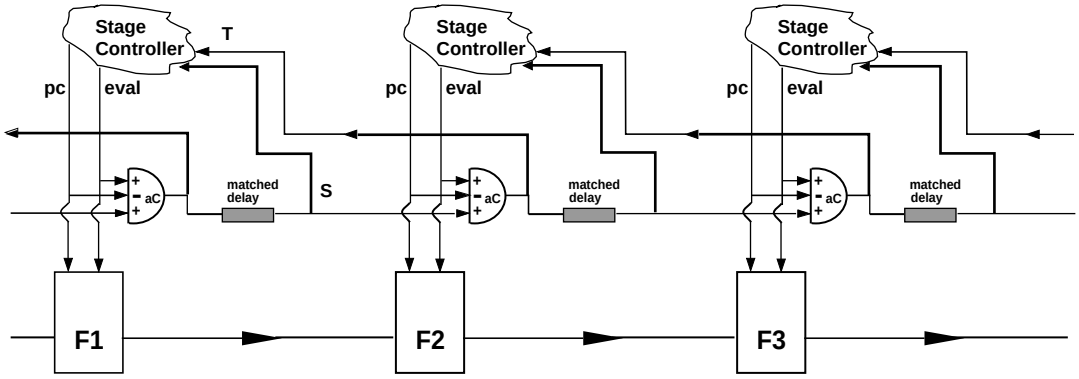

Figure 5. Block diagram of an LP_{HC} pipeline

4.2. Structure

Fig. 5 shows a block diagram of a LP_{HC} pipeline. Each stage consists of three components: a *function block*, a *completion detector* and a *stage controller.* Much like $\text{LP}_{\text{SR}}2/1$, the function block alternately produces data tokens and reset spacers for the next stage, and the completion detector indicates completion of the stage's evaluation or precharge. The third component, the stage controller, generates the decoupled signals, *pc* and *eval*, which control the function block and completion detector.

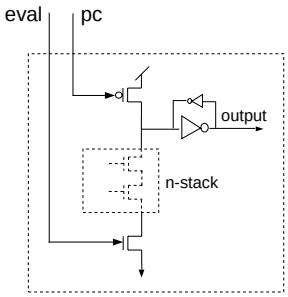

Figure 6. Details of a stage function block

Fig. 6 shows one gate of a *function block* in a pipeline stage. The *pc* input controls the pull-up stack and the *eval* input controls the "foot" of the pull-down stack. 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. When both signals are de-asserted, the gate output is effectively isolated from the gate inputs; this is the "isolate phase." To avoid a short circuit, *pc* and *eval* are never simultaneously asserted.

As in our earlier design, an asymmetric C-element, *aC*, is used as a *completion detector*. The detector's output is set when the stage has begun to evaluate, *i.e.,* when two conditions occur: the stage is in evaluate phase (*eval* is high), and the previous stage has supplied valid input (completion detector output of previous stage is high). Its output is reset when the stage is enabled to precharge (*pc* asserted low). Thus, precharge will immediately reset the completion detector's output, while evaluate will only set the detector's output if valid data inputs have also arrived.

The *aC* element output is again fed through a matched delay, which (combined with the completion detector) matches the worst-case path through the function block. As indicated earlier, for a gate-level pipeline, the matched delay is often unnecessary: the *aC* delay already matches the function block delay.

The resulting completion signal T (of stage F_2 in the figure), in turn, is fed to three components: (i) the previous stage's controller, indicating current stage's state, (ii) the current stage's controller (through the matched delay), and (iii) the next stage (through the matched delay).

The stage controller will be discussed shortly, after presenting the desired protocol.

4.3. Protocol

A pipeline stage simply cycles through three phases, as shown in Fig. 7. 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 PS0 and $\text{LP}_{\text{SR}}2/1$ pipelines, there is now *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, Williams' PS0 uses two explicit synchronization points between adjacent stages: for start of evaluation and for start of precharge. Likewise, our $\text{LP}_{\text{SR}}2/1$ design uses two explicit synchronization points, but signaled from two distinct stages: from $N+1$ (to start precharge) and from $N+2$ (to start evaluation).

As usual, there is one additional implicit synchronization point: the dependence of stage's $N+1$'s evaluation on its predecessor N's evaluation. A stage cannot produce new data until it has received valid inputs from its predecessor. Both of the synchronization points are shown by causality arcs in Fig. 7.

The introduction of the isolate phase is the key to the protocol. Once a stage finishes evaluation, it immediately isolates itself from its inputs by a self-resetting operation—regardless of whether this stage will soon be allowed to en-

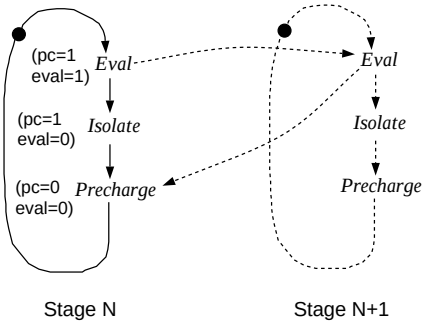

Figure 7. Sequence of phases in a stage cycle, and interaction between stages

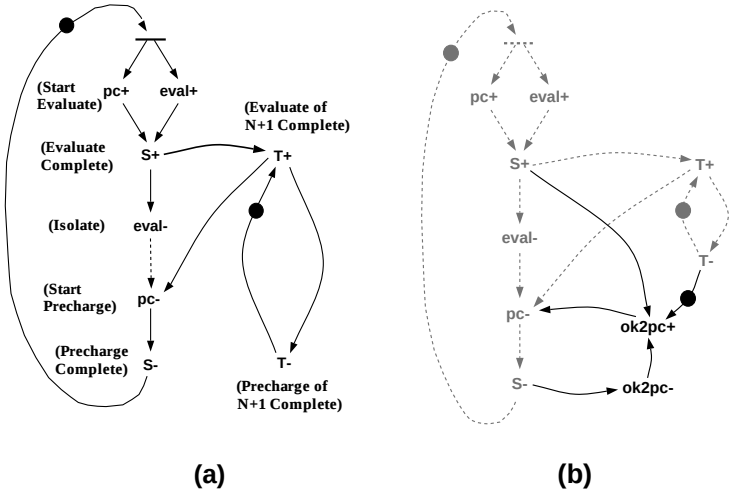

Figure 8. Petri-net specification of pipeline stage controller: (a) before, and (b) after state variable insertion. S **and** T **are controller inputs,** *pc* **and** *eval* **are outputs, and** *ok2pc* **is a state variable.**

ter its precharge phase. Subsequently, its predecessor 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 $N+1$ can evaluate the next data item even before N 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.

4.4. Stage Controller

A formal specification of the stage controller is given in Fig. 8(a). This specification can be easily deduced from the sequence of phases in a stage cycle (Fig. 7). The controller of stage N has two inputs, S and T, which are the "done" outputs of stage N and stage $N+1$ respectively (see Fig. 5), and it has two outputs, *pc* and *eval*, which drive stage N.

The specification of Fig. 8(a) is actually too concurrent to be directly synthesizable. Intuitively, the enabling condition for precharging stage N is ambiguous: N has evaluated a data item and is entering the isolate phase (S high), and $N+1$ has then evaluated the same item (T high). The problem is that, if $N+1$ is blocked or slow, it may continue to maintain its high T output, while stage N processes an entire new data input (precharge then evaluate). In this case, the signals S and T again are both high, but now N and $N+1$ have distinct tokens: since $N+1$ has not absorbed the new data, stage N must not be precharged.[6]

This problem is easily solved by adding a state variable, *ok2pc*, implemented by an asymmetric-C element (see Fig. 8(b)). This variable effectively records whether stage $N+1$ has absorbed a data item: it is reset immediately after stage N precharges (S low), and is only set again once $N+1$ has undergone a subsequent precharge (T low).

[6]More formally, the specification does not satisfy the complete state coding (CSC) property.

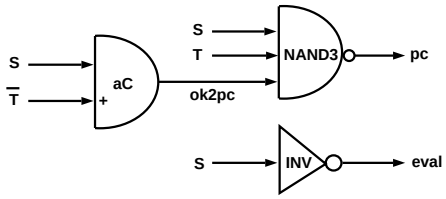

Figure 9. Stage Controller Implementation

Fig. 9 shows an implementation of the controller of Fig. 8(b). The implementation is very simple, with the three signals—*pc*, *eval* and *ok2pc*—each implemented using a single gate. The controllers directly implement the conditions described above and in the previous subsection.

4.5. Pipeline Cycle Time and Latency

A complete cycle of events for stage N can be traced in Fig. 5. 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).

Assume that no extra matched delays are required for the gate-level pipeline, *i.e.* that the completion detector and other delays already match the gate's evaluate and precharge. Then, in the notation introduced earlier, the delay of step (i) is t_{Eval}, the delay of step (ii) is $t_{aC} + t_{\text{NAND3}}$, and the delay of step (iii) is $t_{Prech} + t_{\text{INV}}$. Here, t_{NAND3} and t_{INV} are the delays through the NAND3 and the inverter, respectively, of Fig. 9. Thus, the pipeline cycle time is:

$$T_{\text{LP}_{\text{HC}}} = t_{Eval} + t_{Prech} + t_{aC} + t_{\text{NAND3}} + t_{\text{INV}}$$

A stage's latency is simply the evaluation delay of the stage: $L_{\text{LP}_{\text{HC}}} = t_{Eval}$.

4.6. Timing Constraints

State Variable. LP_{HC} pipelines require a one-sided timing constraint for correct operation. The signal *ok2pc* goes high 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:

$$t_{ok2pc\uparrow} < t_{Eval} + t_{\text{INV}}$$

In practice, this contraint was very easily satisfied.

Precharge Width. As in $\text{LP}_{\text{SR}}2/1$, an adequate precharge width must be enforced. In this design, the constraint is partly enforced by the bundling constraint: the *aC* element and the (optional) matched delay, together, must have greater delay than the worst-case precharge time of the function block. Hence, the S input to the NAND3 in Fig. 9 will be maintained appropriately.

However, there is one additional constraint on precharge width: the T input to the same NAND3 must not be de-asserted. Suppose T just went high. At this point, stage F_1's NAND3 starts the precharge of F_1 (in Fig. 5). Concurrently, T will only be reset after a path through F_3's *aC* element, F_2's NAND3 and *aC* and F_1's NAND3:

$$t_{\text{NAND3}} + t_{Prech_N} \leq t_{aC} + t_{\text{NAND3}} + t_{aC} + t_{\text{NAND3}}$$

Assuming all stages are similar, this constraint becomes:

$$t_{Prech_N} \leq t_{aC} + t_{aC} + t_{\text{NAND3}}$$

This final constraint is also easily satisfied.

Isolate Phase. The inverter in Fig. 9 is used to isolate after evaluate. The bundling constraint already ensures that the isolate phase does not start too early.

5. Example: Pipelined Ripple-Carry Adder

As a case study, two versions of a gate-level pipelined adder were implemented using $\text{LP}_{\text{SR}}2/1$ and LP_{HC} styles. A 32-bit ripple-carry adder was selected, since its design is simple and yet amenable to very fine-grain pipelining. Both adders are suitable for high-throughput applications such as DSP's for multimedia processing.

5.1. Adder Datapath

Each stage of a ripple carry adder consists of a full-adder, which has three data inputs (operands A and B, and carry-in C_{in}) and two outputs (carry-out C_{out}, and *Sum*). The logic equations are:

$$\begin{aligned} Sum &= A \oplus B \oplus C_{in} \\ C_{out} &= AB + AC_{in} + BC_{in} \end{aligned}$$

Since exclusive-or needs both true and complemented values of its operands, our implementation uses two rails each to represent A, B, C_{in} and C_{out}:

$$\begin{aligned} Sum &= (a_1b_0 + a_0b_1)c_{in_0} + (a_1b_1 + a_0b_0)c_{in_1} \\ c_{out_1} &= a_1b_1 + (a_1 + b_1)c_{in_1} \\ c_{out_0} &= a_0b_0 + (a_0 + b_0)c_{in_0} \end{aligned}$$

In both designs, each of the three outputs, *Sum*, c_{out_1} and c_{out_0}, was implemented using a single dynamic gate.

5.2. Adder Control

Interestingly, unlike our pipeline structures, the pipelined adder is a *non-linear structure.* A stage now merges three distinct input streams: the two data operands and the carry-in. Therefore, our pipeline structures must be extended to handle multiple sources. In particular, since each full-adder stage represents a synchronization point of multiple input streams, it must have the capability to handle multiple bundled inputs (*i.e.*, "request" signals).

To somewhat simplify our design, it was assumed that the inputs A and B belong to one shared data stream with a common bundling signal req_{ab}. The C_{in} input along with req_c forms the other stream. Thus, only two input streams are assumed: data operands and carry-in. In practice, this is a reasonable assumption in many applications where operands come from the same source. If this assumption does not hold, our approach can be extended to handle three independent threads.

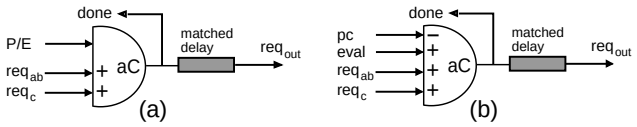

Figure 10. The modified completion detectors for (a) $\text{LP}_{\text{SR}}2/1$ adder, and (b) LP_{HC} adder

Fig. 10 shows the modified completion detectors. The detectors now synchronize on both the data inputs (req_{ab}) and the carry-in (req_c). Each additional request signal is accommodated by adding one transistor to the pull-down stack of the asymmetric C-element of the completion block. The resulting *done* output signal is forked to three destinations: as "acknowledgments" to the stage that sent the carry-in and to the stage that sent the operands, and also as a "request" to the next stage.

Finally, a shift-register is attached to each adder stage to accumulate the stream of sum bits coming out of that stage, as is done in Williams' self-timed divider [10]. Once all the sum bits for an addition operation are available, they can be read off in parallel, one bit from each shift-register. The shift-registers can themselves be built as asynchronous pipelines according to either of the pipeline schemes of this paper.

6. Results

This section presents the results of HSPICE simulations of our two new pipelined adders: $\text{LP}_{\text{SR}}2/1$ and LP_{HC}.

6.1. Experimental Setup

The 32-bit ripple carry adders were simulated in HSPICE using a 0.6μm HP CMOS process with operating conditions of 3.3V power supply and $300\,^{\circ}$K. Special care was taken to optimize the transistor sizing for high-throughput. The precharge PMOS transistors in each dynamic gate had a W/L ratio of $18\lambda/2\lambda$. The NMOS transistors in the evaluation stack were so sized that the effective width of the

Adder Design	t_{Eval} (ns)	t_{Prech} (ns)	t_{aC} (ns)	t_{NANDB} (ns)	t_{NAND3} (ns)	t_{INV} (ns)	Cycle Time, T Analytical Formula	Cycle Time, T (ns)	Throughput 10^6 items per sec.
$\text{LP}_{\text{SR}}2/1$	0.28	0.24	0.28	0.22			$2 \cdot t_{Eval} + t_{aC} + t_{\text{NANDB}}$	1.07	930
LP_{HC}	0.26	0.23	0.26		0.12	0.11	$t_{Eval} + t_{Prech} + t_{aC} + t_{\text{NAND3}} + t_{\text{INV}}$	0.98	1023

Table 1. The performance of the $\text{LP}_{\text{SR}}2/1$ and LP_{HC} adders.

n-stack was $1/3$ that of the *p*-stack. Furthermore, for each of the designs, it was ensured that the timing constraints of Sections 3 and 4 were comfortably met.

6.2. Simulation Results

Table 1 summarizes the simulation results. For each design, the table lists the overall cycle time as well as its breakdown into components: stage evaluation time (t_{Eval}), stage precharge time (t_{Prech}), the delay though the completion block (t_{aC}), as well as the delays through the control gates (t_{NANDB}, t_{NAND3} and t_{INV}). Finally, the table lists the throughput of each adder in million operations per second.

The throughputs of the adders are quite good: 930 and 1023 million operations per second. Interestingly, the throughput using our new high-capacity structure (LP_{HC}) is *10% faster* than using our earlier high-throughput structure ($\text{LP}_{\text{SR}}2/1$). This improvement can be attributed to two factors. First, after cancelling similar terms from the analytical cycle times of LP_{HC} and $\text{LP}_{\text{SR}}2/1$, the new scheme is left with the relatively smaller t_{Prech} term, compared with the somewhat larger t_{Eval} term that remains in the latter. Second, LP_{HC} has slightly smaller t_{Eval} and t_{Prech} values than in $\text{LP}_{\text{SR}}2/1$; this difference occurs because $\text{LP}_{\text{SR}}2/1$ has heavier loading on the bundling output, which must be forked off to two previous stages instead of one.

We do not yet have a direct comparison with recent synchronous approaches, since we have not implemented, sized and simulated comparable designs. However, it is worth noting that one of the best new synchronous fine-grain pipeline styles has significantly lower performance. In [5], a *wave-steering approach* is proposed to build synchronous gate-level pipelines from decision diagrams. The minimum reported clocking period for correct functionality after optimized device sizing was 1.4 ns in 0.5μ. Our LP_{HC} adder is *42% faster* in 0.6μ technology.

Additional expected benefits of our asynchronous pipelined adders over synchronous versions, especially for system-on-a-chip, are that they: (a) accommodate varied input and output rates (useful as a reusable component, for interfacing with different IPs); (b) only consume (non-trivial) power in the active portions of the adder (no clock gating required); and (c) avoid problems of high-speed clock distribution issues. The latter benefit should be especially useful when re-implementing our designs in more modern processes of $<0.2\mu$, where clock rates of several Gigahertz would be required.

7. Conclusions

This paper has introduced a new asynchronous gate-level pipelining scheme, and two new pipelined adder implementations, for high-throughput applications. The novelty of our approach is two-fold: (i) *decoupled control* for the pull-up and pull-down of dynamic gates, and (ii) only a *single explicit synchronization point* between adjacent pipeline stages. Simulation of our pipelined adder designs shows that the increased concurrency results in extremely high throughput. In addition, the simulations demonstrate that the one-sided timing constraints for our pipelines are very easily satisfied in practice. We expect our control latencies to be competitive with the overheads of wave-pipelined approaches [12], and yet our pipelines are more robust and require much less design effort. In more modern technologies, we expect our pipelines to deliver a throughput of several Gigaoperations/second.

Acknowledgments. The authors gratefully acknowledge Naeem Abbasi and Prof. Ken Shepard for their help with simulations. We also benefited from helpful discussions with Michael Theobald and Tiberiu Chelcea.

References

[1] A. Dooply and K. Yun. Optimal clocking and enhanced testability for high-performance self-resetting domino pipelines. In *ARVLSI'99*.

[2] D. Harris and M. Horowitz. Skew-tolerant domino circuits. *IEEE JSSC*, 32(11):1702–1711, Nov. 1997.

[3] W. Liu, C. T. Gray, D. Fan, W. J. Farlow, T. A. Hughes, and R. K. Cavin. A 250-MHz wave pipelined adder in 2-μm CMOS. *IEEE JSSC*, 29(9):1117–1128, Sept. 1994.

[4] C. Molnar, I. Jones, W. Coates, J. Lexau, S. Fairbanks, and I. Sutherland. Two FIFO ring performance experiments. *Proceedings of the IEEE*, 87(2):297–307, Feb. 1999.

[5] A. Mukherjee, R. Sudhakar, M. Marke-Sadowska, and S. Long. Wave steering in YADDs: a novel non-iterative synthesis and layout technique. In *Proc. DAC*, 1999.

[6] V. Natayanan, B. Chappell, and B. Fleischer. Static timing analysis for self resetting circuits. In *Proc. ICCAD*, 1996.

[7] M. Renaudin, B. Hassan, and A. Guyot. New asynchronous pipeline scheme: Application to the design of a self-timed ring divider. *IEEE JSSC*, 31(7):1001–1013, July 1996.

[8] M. Singh and S. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *Proc. Intl. Symp. Adv. Res. Async. Circ. Syst. (ASYNC)*, 2000.

[9] C. van Berkel, M. Josephs, and S. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, Feb. 1999.

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

[11] T. Williams and M. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE JSSC*, 26(11):1651–1661, Nov. 1991.

[12] D. Wong, G. De Micheli, and M. Flynn. Designing high-performance digital circuits using wave-pipelining. *IEEE TCAD*, 12(1):24–46, Jan. 1993.

[13] K. Yun, P. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proc. Intl. Symp. Adv. Res. Async. Circ. Syst. (ASYNC)*, 1996.