High-Speed Non-Linear Asynchronous Pipelines

Recep O. Ozdag[1]
ozdag@usc.edu

Montek Singh[2]
montek@cs.unc.edu

Peter A. Beerel[1]
pabeerel@usc.edu

Steven M. Nowick[3]
nowick@cs.columbia.edu

[1]Department of Electrical Engineering—Systems Division, USC, Los Angeles, CA 90089
[2]Department of Computer Science, UNC–Chapel Hill, Chapel Hill, NC 27599 (formerly at Columbia University)
[3]Department of Computer Science, Columbia University, New York, NY 10027

Abstract

Many approaches recently proposed for high-speed asynchronous pipelines are applicable only to linear datapaths. However, real systems typically have non-linearities in their datapaths, i.e. stages may have multiple inputs ("joins") or multiple outputs ("forks"). This paper presents several new pipeline templates that extend existing high-speed approaches for linear dynamic logic pipelines, by providing efficient control structures that can accommodate forks and joins. In addition, constructs for conditional computation are also introduced. Timing analysis and SPICE simulations show that the performance overhead of these extensions is fairly low (5% to 20%).

1. Introduction

High-speed asynchronous design is increasingly becoming an attractive alternative to full-custom synchronous design because of its freedom from clock distribution and clock skew problems, and because it naturally provides robust interfaces to slower components (e.g., [1][2]). A number of fast asynchronous fine-grain pipeline templates have been proposed for high-speed design, including IPCMOS [3], GasP [4][2] and pulse-mode circuits [5]. These ultra-high-speed designs have very aggressive timing assumptions that introduce stringent transistor sizing requirements and high demands on post-layout verification.

In recent work, Singh and Nowick have proposed several high-speed dynamic logic pipeline templates [6][7], as well as high-speed static logic pipeline templates [8], that achieve comparable performance with much less stringent timing assumptions. The dynamic pipelines were introduced for linear datapaths (i.e. without forks and joins), although preliminary solutions for handling joins were proposed in [7]. In addition, an initial approach to handling slow or stalled environments for the limited case of linear pipelines was also proposed in [6]. However, the synchronization problems that arise when using arbitrary forks and joins are much more complex and challenging, and the approaches of [6][7] do not address these issues. This paper attempts to fill this void.

The contribution of this paper is a set of five new non-linear pipeline templates that extend the two dynamic logic pipeline styles from Columbia: *lookahead pipelines* (LP) [6] and *high-capacity pipelines* (HC) [7]. Several distinct lookahead pipeline styles were proposed in [6], both single-rail and dual-rail. This paper builds upon one representative each of single-rail ($LP_{SR}2/2$) and dual-rail (LP3/1) lookahead pipelines, and also upon the single-rail high-capacity pipeline (HC). The ideas presented here, however, can be easily adapted to the remaining styles.

The remainder of this paper is organized as follows. Section 2 gives background on single and dual-rail datapaths, and reviews some of the basic linear pipelines of [6] and [7]. Section 3 gives an overview of some of the challenges involved in the design of non-linear pipelines. Sections 4-6 present the new non-linear designs in detail, including their protocols, implementation, and timing analysis. Extensions to handle conditional computation are proposed in Section 7 and, finally, experimental results and conclusions are given in Sections 8 and 9.

2. Background

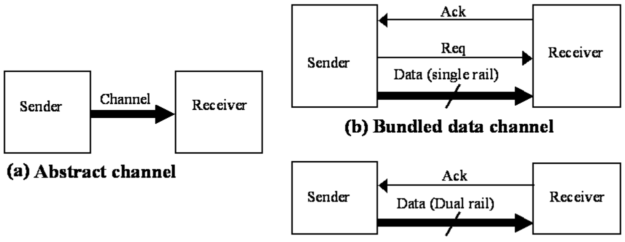
This section first gives background on commonly-used asynchronous data representation schemes. Then, it reviews three asynchronous pipelining styles: (i) $LP_{SR}2/2$, a single-rail lookahead pipeline, (ii) LP3/1, a dual-rail lookahead pipeline, and (iii) HC, the high-capacity pipeline.

2.1 Bundled Data vs. Dual-Rail Encoding

One common paradigm of asynchronous system design is to decompose the system into functional units that communicate data via channels, as shown in Figure 1(a). In these channels, data can be encoded in many ways. In the *single-rail* encoding scheme, one wire per bit is used to transmit data, and an associated request line is used to indicate data validity, as shown in Figure 1(b). The associated channel is called a bundled-data channel [12]. Alternatively, in *dual-rail* encoding, the data is sent using two wires for each bit of information, as shown in Figure 1(c) [10]. Extensions to 1-of-N and M-of-N encoding also exist.

Both single-rail and dual-rail encoding schemes are commonly used, and there are tradeoffs between each. Dual-rail encoding allows for data validity to be indicated by the data itself. Single-rail, in contrast, requires the associated request line that is driven by a matched delay line that must always be longer than the computation. This latter approach requires careful timing analysis but allows the reuse of synchronous single-rail logic.

This work was supported by a large-scale NSF ITR Award No. CCR-00-86036, a gift from Sun Microsystems, Inc., a gift from TRW, Inc., and a grant from New York State Center for Advanced Technology at Columbia University.



(a) Abstract channel

(b) Bundled data channel

(c) Dual–rail channel

Figure 1. Pipeline Channels

2.2 Lookahead Pipelines (Single-Rail)

Figure 2(a) shows the structure of one stage of the $LP_{SR}2/2$[1] lookahead single-rail pipeline [6]. Each stage has a dynamic function block and a control block. The function block alternately evaluates and precharges. The control block generates the bundling signal, *Lack*, to indicate completion of evaluation (or precharge). The bundling signal is generated by an asymmetric C-element [6], and passed through a suitable delay, allowing time for the dynamic function block to complete its evaluation (or precharge). Note that there is one dynamic gate for each individual output rail of the stage, and different dynamic gates inside a function block can sometimes share precharge and evaluate (foot) transistors.

This pipeline style has two important features. First, the completion signal, *done*, is sent to the previous stage as an acknowledgment (Lack) by tapping off from before the matched delay. This “early tap-off” is safe because a dynamic function block typically is immune to a reset of its inputs as soon as the first level of dynamic logic has absorbed the input data. The second feature is that the control signal, Pc, is applied directly to the function block, rather than applying the output of the completion detector. Therefore, the function block can be precharge-released even before the arrival of new input data. This early precharge-release is safe because the dynamic logic block will compute only upon the receipt of actual data. Both of these features eliminate critical delays from the cycle time, resulting in very high throughput.

The analytical cycle time can be expressed using the following components:

t_{Eval} = delay of function block evaluation

t_{ac} = delay of control (asymmetric C-element)

For correct operation, the matched delay t_{delay} must satisfy $t_{delay} \geq t_{Eval} - t_{ac}$. For ideal operation, t_{delay} is no larger than necessary, $t_{delay} = t_{Eval} - t_{ac}$. Note that, in fine-grain pipelines, the latency through the function block is often less than the delay of the asymmetric C-element. In such a scenario, no matched delay is necessary; the asymmetric C-element provides sufficient delay to satisfy the bundling constraint. Using the above notation and assumption, the pipeline’s analytical cycle time is:

$$T_{LPSR2/2} = 2.\ t_{Eval} + 2.\ t_{ac}$$

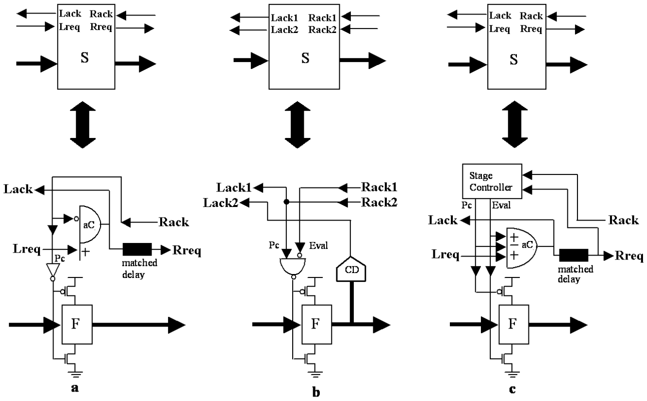


Figure 2. a) LPSR2/2 b) LP3/1 and c) HC pipelines

The per-stage latency of the pipeline is:

$$L_{LPSR2/2} = t_{Eval}$$

Note that both t_{Eval} and t_{ac} consist of two gate delays.

2.3 Lookahead Pipelines (Dual-Rail)

Figure 2(b) shows the structure of one stage of the dual-rail LP3/1[2] pipeline [6]. In this pipeline, there are no matched delays. Instead, each stage has an additional logic unit, called a *completion detector,* to detect the completion of evaluation and precharge of that stage.

Unlike most existing approaches, such as Williams’ and Horowitz’s pipelines [9][10], each stage of the LP3/1 pipeline synchronizes with two subsequent stages, *i.e.,* not only with the next stage, but also its successor. Consequently, each stage has two control inputs. The first input, *Pc*, comes from the completion detector *(CD)* of the next stage, and the second control input, *Eval*, comes from the completion detector two stages ahead.

The benefit of this extra control input is to allow a significantly shorter cycle time. This *Eval* input allows the current stage to evaluate as soon as the subsequent stage has *started* precharging, instead of waiting until the subsequent stage has completed precharging.

The analytical cycle time can be expressed as:

$$T_{LP3/1} = 3.\ t_{Eval} + \ t_{CD} + \ t_{NAND}$$

The per-stage latency of the pipeline is:

$$L_{LP3/1} = t_{Eval}$$

Both t_{Eval} and t_{CD} consist of two gate delays; t_{NAND} is only one gate delay.

2.4 High-Capacity Pipelines (Single-Rail)

Finally, the structure of one stage of the HC pipeline [7] is shown in Figure 2(c). A novel feature of this pipeline style is that it uses *decoupled control* of evaluation and precharge: separate Eval and Pc signals are generated by each stage's control.

[1] The 2/2 label characterizes the operation of the stage of a pipeline: 2 components in the evaluation phase and 2 component delays in the precharge phase, forming a complete cycle.

[2] As with the previous pipeline style, the 3/1 label characterizes the operation of a stage of the pipeline: 3 component delays in the evaluation phase and 1 component delay in the precharge phase, forming a complete cycle.

Precharge occurs when *Pc* is asserted and *Eval* is de-asserted. Evaluation occurs when *Pc* is de-asserted and *Eval* is asserted. When both signals are de-asserted, the gate output is effectively isolated from the gate inputs; this is a new phase, called the *isolate phase* (see below).[3]

Much like in LP$_{\mathbf{SR}}$2/2, an asymmetric C-element, *aC*, is used as a completion detector. The aC element output is fed through a matched delay, which (combined with the completion detector) matches the worst-case path through the function block.

Unlike most existing pipelines, the HC pipeline stage cycles through three distinct phases. After it completes the evaluate phase, it enters the new isolate phase (where both Eval and Pc are de-asserted) and subsequently the precharge phase, after which it re-enters the evaluate phase, completing the cycle. Furthermore, unlike the other pipelines covered in this paper as well as the PS0 style in [8], the HC pipeline has only one explicit synchronization point between stages. Once the subsequent stage has completed its evaluate phase, it enables the current stage to perform its entire next cycle.

The analytical cycle time can be expressed as:

$$T_{HC} = t_{Eval} + t_{Prech} + t_{aC} + t_{NAND3} + t_{INV}$$

The per-stage latency of the pipeline is:

$$L_{HC} = t_{Eval}$$

In this design, t_{Eval}, t_{Prech} and t_{aC} consist of two gate delays each, though t_{NAND} and t_{INV} consist of only one gate delay.

3. Challenges of Handling Forks and Joins

There are two basic challenges involved in designing non-linear pipelines: (i) synchronization of a stage with *multiple destinations* (e.g., for forks), and (ii) synchronization of a stage with *multiple sources* (e.g., for joins). This section discusses these issues in detail, and outlines strategies to address them. Subsequent sections provide our detailed solutions for each of the three pipeline styles considered in this paper.

3.1 Slow or Stalled Right Environments in Forks

In many existing linear asynchronous pipelines—such as Williams' and Horowitz' classic PS0 pipeline [10], as well as lookahead and high-capacity pipelines—certain acknowledgments between stages are essentially timed pulses, i.e., some inter-stage communications are *non-persistent.* In particular, after a stage asserts its acknowledgment, causing a precharge of the previous stage, it assumes that the precharge of that previous stage is quite fast. Therefore, it does not explicitly check for the precharge's completion before de-asserting its acknowledgment signal. This timing assumption is referred to as a *fast precharge assumption,* and is typically easily satisfied. Thus non-persistence is usually not problematic in linear pipelines: all stages can be reasonably assumed to react fast enough to acknowledgment pulses [9][10].

However, when a datapath has a fork, non-persistence can be a challenge. In this case, multiple acknowledgment signals are received by the forking stage. These signals are therefore pulses, which may be *non-overlapping.* Therefore, acknowledgments may not be correctly merged using a simple C-element.

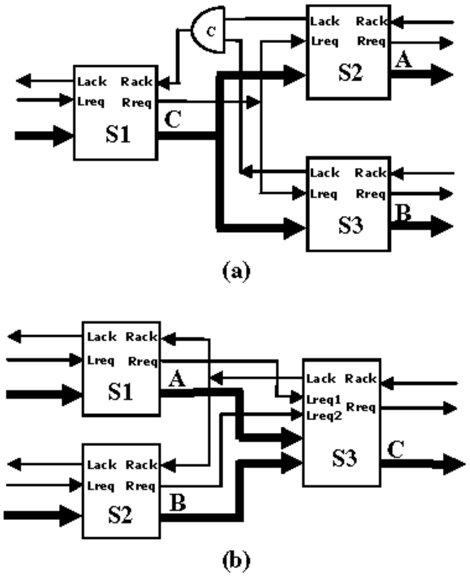


Figure 3. a) Fork and b) join

As an example, Figure 3(a) shows an abstract two-way fork for a bundled datapath, where the forking stage S1 drives stages S2 and S3. For correct operation, S1 must receive acknowledgments from both S2 and S3. However, stages S2 and S3, and the subsequent stages of each, may be operating largely independent of each other. Suppose stage S3 is arbitrarily delayed (or stalled), thus delaying the acknowledgment for S1 from S3. Meanwhile, an early non-persistent acknowledgment is received by S1 from S2, which is not delayed. As a result, the two acknowledgments received by S1 *may have no overlap,* and, if combined using a C-element, may not generate the precharge signal for S1 at all!

This problem is referred to as the *slow* or *stalled right environment (SRE) problem*. In this paper, two general solutions are proposed to address this issue. The first solution is to *condition* the acknowledgments received from the stages immediately to the right of the fork to make these acknowledgments persistent. In this case, later stages in the non-delayed fork branch of the pipeline (i.e., S2's successors above) have no further constraints on their behavior. Thus, this solution requires only local changes, i.e., to the immediate forked stages.

The second solution requires more global modifications. In particular, the basic control circuit of *every subsequent* pipeline stage is modified so as to make *all acknowledgments persistent.* As a result, the fast precharge constraint is eliminated, allowing for simpler strategies to combine acknowledgments, which are now persistent. In this case, later stages in a non-delayed fork branch (i.e. S2's successors above) *are* further constrained in their behavior. (They can only go through a precharge on the new data item, but not enter the subsequent precharge-release phase.)

Interestingly, the SRE problem can also be formulated as a relative-timing constraint [13]: the request from the forking stage de-asserts prior to the de-assertion of the acknowledgement from either of the immediate forked stages (i.e., S2 and S3), thereby preserving the four-phase handshaking protocol on the channels in-between.

[3] To avoid short circuit, Pc and Eval are never simultaneously asserted.

3.2 Slow or Stalled Left Environments in Joins

The second challenge is one of synchronization with multiple input channels in joins, as shown in Figure 3(b).

A problem can arise if an “eager” function block is used for the implementation of stage S3, i.e., S3 may produce outputs after consuming only one (not both) of its data inputs (see [9]). For example, suppose S3 contains a dual-rail OR function that evaluates eagerly (i.e., as soon as one high input bit arrives). Then, after evaluation, it will send an acknowledgment to *both* S1 and S2, even though S1 may not have produced data. As a result, if input stage S1 is particularly slow or stalled, it may receive an acknowledgment from S3 too soon. This behavior can treat the output of the slow stage as a new unwanted data token, and thus corrupt the synchronization between the stages!

This problem is referred to as the *stalled left environment (SLE) problem*. Note that the SLE problem does not arise in single-rail pipelines: a stage can verify that all of its senders have produced valid data by examining the associated bundling signals.

There are two solutions to this problem. One solution is to simply use “non-eager” function blocks; that is, every function block explicitly checks for the validity of all of its dual-rail inputs, before producing a valid output. Such function blocks are sometimes referred to in literature as *weakly-indicating* or *weak-conditioned* logic blocks [10][11][12][15][16]. However, the term “weak-conditioned” is often used in a somewhat more restrictive sense than “non-eager”: weak-conditioned logic blocks not only explicitly check for the validity of all inputs before producing an output, but they also explicitly check for the *reset* of all inputs before resetting the output. Therefore, non-eager blocks are sometimes referred to as “semi-weak-conditioned.”

The second solution is to allow eager function blocks, but still ensure that the generation of the acknowledge signal occurs only after data from all of the input stages has been received. This latter solution requires modification to the control, and is discussed in more detail in the sections that follow.

The SLE problem can also be formalized as a relative-timing constraint: the join stage must generate an acknowledgement signal only after all input channels to the join stage have valid data, thereby preserving the four-phase handshaking protocol on all input channels.

4. Lookahead Pipelines (Single-Rail)

Handling joins in single-rail lookahead pipelines is straightforward, and was initially proposed in [7]. The join stage receives multiple request inputs (Lreq’s), all of which are merged together in the asymmetric C-element (aC) that generates the completion signal. In particular, each additional request is accommodated by adding an extra series transistor in the pull-down stack of the aC element. The aC will only acknowledge the input sources after all of the Lreq’s are asserted and the stage evaluates.

To handle forks, on the other hand, a C-element must be added to the forking stage to combine the acknowledgments from its immediate successors. In addition, other stages of the pipeline must also be modified to overcome the SRE problem of Section 3.1. As discussed earlier, the problem is that the acknowledge signal from an immediate successor to a forking stage is *non-persistent*; it may be de-asserted before its predecessor forking stage has completed its precharge. This section gives two distinct solutions for correctly handling such forks in LP$_{SR}$2/2.

4.1 Solution 1 for LP$_{SR}$2/2 Forks

The first solution is to modify only the immediate successor stages (say S2 and S3) of a forking stage (S1), in order to make their acknowledges persistent. In particular, in each such immediate successor stage, the *Lack* acknowledgment signal is made persistent by effectively *latching* it, and the stage’s next evaluation is delayed until its predecessor has completed its precharge. For LP$_{SR}$2/2, this solution is shown in Figure 4: the *Lack* generation logic is made persistent and the control of the foot transistor is also modified.

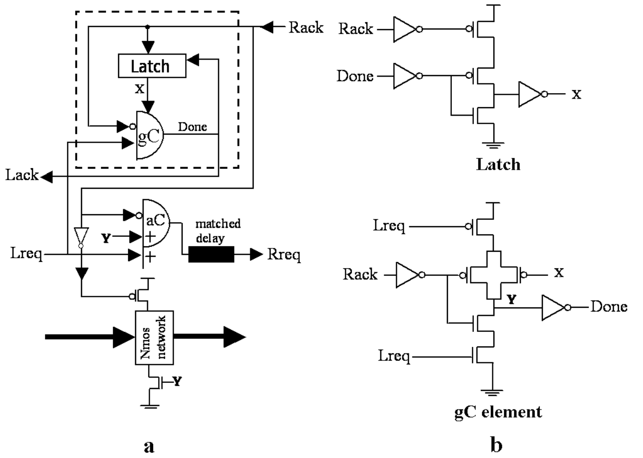


Figure 4. a) Modified first stage after the fork. b) Detailed implementation of individual gates

The new control circuit operates as follows. Assume a forked stage S2 has just evaluated and the acknowledgment signal *Lack* has just been asserted. Eventually, the right environment will assert *Rack,* causing the output of the dynamic latch, *X*, to be asserted (*X=0, i.e., active low*), effectively latching the non-persistent acknowledgment signal. Note that the *X* output is held low even when *Rack* is de-asserted. In particular, *X* is de-asserted (*X=1*) only after *Done* goes low, in turn caused by *Lreq* going low, which indicates that the input forking input stage has precharged. Effectively, the foot transistor now prevents any re-evaluation until *after* X goes low, thus delaying re-evaluation until all inputs (including any slow input) are guaranteed to have precharged.

These modifications ensure that even late acknowledgments from another stage S3, immediately after the fork, are guaranteed to be properly received by forking stage S1, while still ensuring that S3 satisfies the fast precharge constraint. As a result, the SRE problem is solved. Interestingly, the interaction of S3 with the remainder of the pipeline to its right remains unchanged: the stages to the right of S3 are unmodified, and thus allowed to generate non-persistent acknowledgments.

The only new timing assumption introduced by this template, compared to LP$_{SR}$2/2, is that the *Rack* pulse width must be long enough to properly latch it. This pulse width assumption, however, is less restrictive than the original timing assumption that remains:

the pulse width must be longer than the stage's precharge time.

4.2 Solution 2 for $\mathrm{LP_{SR}}$2/2 Forks

The second solution is to modify *each stage* on all paths beyond the forking stage, so that they do not de-assert their acknowledgments until after all input stages are guaranteed to have precharged. This solution can be implemented using the modified $\mathrm{LP_{SR}}$2/2 template shown in Figure 5 in which the asymmetric C-element is converted to a symmetric C-element. As suggested earlier, this modification removes the fast precharge constraint, implicitly solving the SRE problem.

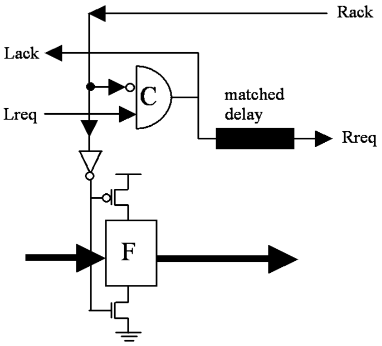


Figure 5. An $\mathrm{LP_{SR}}$2/2 stage with a symmetric C-element

4.3 Pipeline Cycle Time

For the first solution, the cycle time expressions do not change if the additional acknowledgment signals simply increase stack height and do not add additional gates. For multi-way forks and joins, however, the cycle time will increase by the additional C-elements needed to combine them. For the second solution, the cycle time becomes:

$$T_{LPSR2/2} = max\ (2.\ t_{Eval} + 2.\ t_{gc,}\ t_{Eval} + t_{prech} + 2.\ t_{gc})$$

5. Lookahead Pipelines (Dual-Rail)

This section extends a dual-rail lookahead pipeline, LP3/1, to handle forks and joins. Since both the stalled left environment (SLE) and the stalled right environment (SRE) problems of Section 3 can arise in dual-rail pipelines, detailed solutions are presented for both forks and joins.

5.1 Joins

Unlike $\mathrm{LP_{SR}}$2/2, the LP3/1 pipeline has no explicit request line and thus may not function correctly unless it is modified to handle the SLE problem in joins. Our proposed solution allows the use of eager function blocks; however it still ensures that no acknowledgment is generated from a stage until after all its input stages have evaluated.

In particular, our solution is to add *explicit* request signals to each input channel of a join stage, and feed them into the join stage's completion detector, as illustrated in Figure 6. The join's completion detector now delays asserting its acknowledgment until not only the function block is done computing, but also until after all of its input stages have completed evaluation.

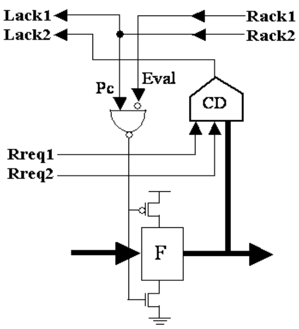


Figure 6. The LP3/1 pipeline with a modified CD to handle joins

Note that the additional request signals are taken from the outputs of the preceding stages' completion detectors. While this modification does not affect the latency of the pipeline, the analytical cycle time changes to:

$$T_{LP3/1} = 2.\ t_{Eval} + 2.\ t_{CD} + \ t_{NAND}$$

5.2 Forks

To handle forks, as in the single-rail lookahead pipeline, $\mathrm{LP_{SR}}$2/2, a C-element is added to the forking stage to combine the multiple acknowledgments it receives from the fork branches. In addition, there are two solutions for the slow or stalled right environments. These solutions are similar in essence to the solutions for the single-rail case, but adapted to dual-rail.

The implementation of solution 1 is very similar to $\mathrm{LP_{SR}}$2/2 and involves modifying the forking stage and the first stages after the fork to make *Lack1* persistent and not generate nor use *Lack2* signals. First, the completion detector (CD) of the first stages after the fork are modified such that the acknowledgment signal is de-asserted only after the forking stage has precharged, as shown in Figure 7. Second, the re-evaluation of the function block of this stage is delayed until after the forking stage has precharged using a decoupled foot transistor controlled by the Y signal. Finally, the generation of *Lack2* is removed from this stage (notice that *Lack2* does not appear in Figure 7). Thus, the PC signal of the forking stage is controlled directly by Lack1 signals, thereby eliminated the need for a NAND gate in the forking stage. Consequently, in this forking structure, the feedback is limited to one stage ahead, rather, than the original two stages ahead.

The second solution is to add an explicit request line to all LP3/1 channels and delay de-assertion of the acknowledgment (*Lack1* in this case) until after all immediate predecessors have precharged, as shown in Figure 8. The request line is generated via a C-element that combines the incoming request line(s) and the output of the completion detection. The output of this C-element becomes the new *Lack1*. Because the C-element de-asserts its acknowledgment only after *Lreq* is de-asserted, the fast precharge constraint is removed, solving the SRE problem.

For solution 1, compared to the original LP3/1 template, the cycle time is slightly increased to:

$$T_{LP3/1} = 2.\ t_{Eval} + \ 3.t_{CD} + \ t_{Prech}$$

For solution 2, the cycle time increases to:

$$T_{LP3/1} = \ t_{Eval} + \ 3.t_{CD} + \ t_{NAND}$$

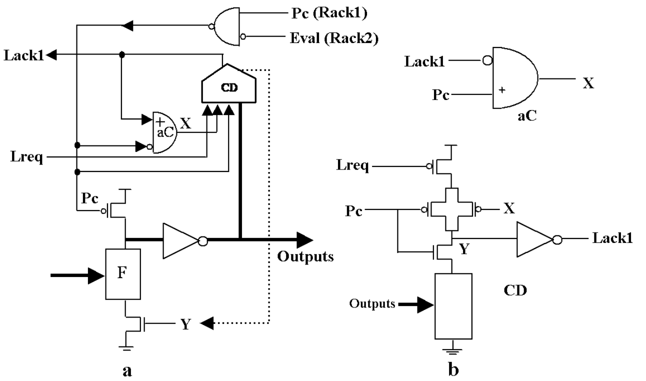


Figure 7. a) Modified first stage after the fork. b) Detailed implementation of the additional gates

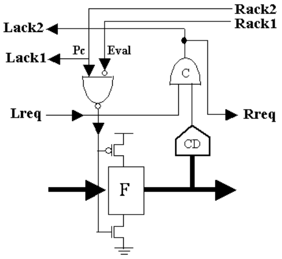


Figure 8. The LP3/1 stage with a C-element

6. High-Capacity Pipelines (Single-Rail)

Next, the basic linear high-capacity style is generalized to include forks and joins. Since the high-capacity style uses single-rail encoding, it already has a request line associated with the data, and thus handling slow or stalled left environments is not an issue. However, because the acknowledgment signals in the high-capacity pipelines are non-persistent, much like in lookahead pipelines, the problem of handling slow or stalled right environments needs to be addressed.

In this section, the basic high-capacity style is first extended to handle arbitrarily slow environments, and then generalized to accommodate forks and joins.

6.1 Handling Arbitrary Environments

Figure 9 shows a simple modification to the original stage controller, which allows the high-capacity pipeline stage to interface with arbitrarily slow left and right environments.

In the new pipeline, the acknowledgment from a high-capacity stage is made persistent by replacing the *NAND3* gate in the control by a state holding generalized C-element (gC), as shown in Figure 9(b). In particular, the gC-element behaves as follows. The acknowledgment signal *Rack* now only triggers the assertion of the precharge control signal, *Pc*. The precharge signal then stays asserted (i.e. *persistent*) until it is de-asserted by the input request signal *Rreq* going low. In addition to this change to the *NAND3* gate, the inverter is replaced by a *NOR2* gate with an additional input. This *NOR2* gate conditions the stage's *Eval* signal, to ensure that the subsequent evaluation phase is delayed until the stale input data has been reset.

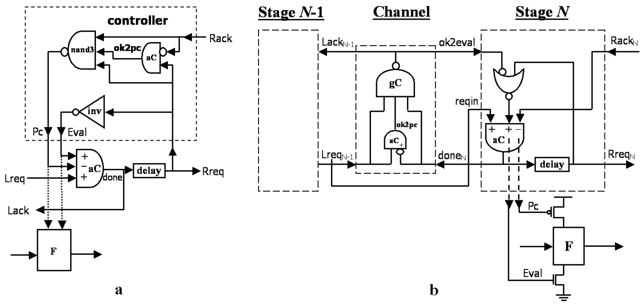


Figure 9. a) Original and b) New HC stage

In the new version of the HC pipeline stage, the state variable, *ok2pc,* is pulled out of the stage controller, and instead placed into the channel between stages *N-1* and *N.*

This new placement of the state variable is justified as follows. The function of the state variable is to keep track of whether stages *N-1* and *N* are computing the same token, or distinct (consecutive) tokens; precharge of *N-1* is inhibited if the tokens are different. If there are two stages, say S1 and S2, supplying data for stage S3, two separate state variables are used, one to keep track of whether stages S1 and S3 have the same token, and the second to keep track of whether stages S2 and S3 have the same token. Similarly, if stage S3 had two successors, S4 and S5, we propose to have two distinct state variables, one each for the pair (S3, S4) and the pair (S3, S5).[4]

To summarize, it is logical to have the *aC* element, which implements the state variable *ok2pc*, pulled out of the stage controller and placed in-between stages *N-1* and *N* (i.e., moved into the channel). In addition, the *gC* element is also moved into the channel to avoid extra wiring.

6.2 Handling Forks and Joins

Using the above generalizations to handle slow environments, an HC stage can now be extended to handle forks and joins. Figure 10 shows the implementation of a template for stage, *N*, for the case where stage *N* is both a fork as well as join. The multiple *reqin's*, *ok2eval's* and *ack's* are handled by simple modifications to the linear pipeline of Figure 9(b).

Multiple *reqin's*: Each additional *reqin* is handled by adding a single series transistor to the aC element that makes up the completion generator, much like in $LP_{SR}2/2$ (Section 4). Hence, *done* is generated only after all input streams have been received.

Multiple *ok2eval's*: Each additional *ok2eval* is handled by adding it as an extra input to the NOR gate that produces the *eval* signal. Consequently, the stage is enabled to evaluate (*eval* asserted) only after all of the *ok2eval* signals are asserted, i.e. after all of the senders have precharged.

Multiple *ack's*: Multiple *ack's* are handled by *OR*'ing them together. Since the *ack's* are all asserted low, the *OR* gate output goes low only when all the *ack's* are asserted, thus ensuring that precharge occurs only after the stage's data outputs have been absorbed by all of the receivers. The *OR* gate is actually implemented as a *NAND* with bubbles (inverters) on the *ack*

[4] Note that, unlike HC, the LP styles do not need state variables in their channels because their operation is relatively less concurrent.

inputs. This *NAND* has an additional input—the stage's completion signal—whose purpose is to ensure that, once precharge is complete, *Pc* is quickly cut off. Otherwise, *Pc* may get de-asserted slightly after *Eval* is asserted, causing momentary short-circuit between supply and ground inside the dynamic gates.

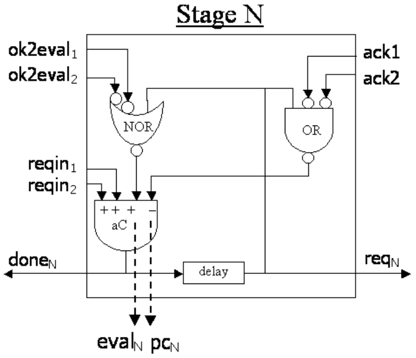


Figure 10. A 2-way join 2-way fork HC stage

6.3 Pipeline Cycle Time

If only joins are present, the cycle time is only slightly increased. Compared with the cycle time obtained in [7], the new cycle time equation has a NOR delay instead of an inverter delay, and a gC delay instead of a NAND3 delay:

$$T_{HC} = t_{Eval} + t_{Prech} + t_{aC} + t_{gC} + t_{NOR}$$

If forks are also present, then the cycle time increases by the delay of the OR gate which is needed to combine the multiple acknowledgments:

$$T_{HC} = t_{Eval} + t_{Prech} + t_{aC} + t_{gC} + t_{NOR} + t_{OR}$$

In these expressions, t_{Eval}, t_{Prech}, t_{aC} and t_{OR} consist of two gate delays each; t_{NOR} and t_{gC} consists of only one gate delay each.

7. Conditionals

There are other complex pipeline structures that allow conditional reading and writing of data. Such structures can also be adapted for use as memory cells. This section briefly discusses the implementation of two such constructs for the LP$_{SR}$2/2 style. Similar circuits can be derived for the other pipeline styles.

As the first example, Figure 11(a) shows a conditional read structure, where the stage reads data from one of several input channels, or, in general, from a subset of several input channels. The decision as to which channels are read from is determined by a bit pattern supplied by a special "select channel." Only those channels that are read from are acknowledged. Similarly, Figure 11(b) shows a conditional write, where the stage reads from an input channel, and writes to one of several output channels. The choice of which channel to write to is once again determined by the word supplied by the select channel. The stage that writes the data receives an acknowledgment only from the output channel where the data is written. Note that the C-elements are only symmetric for the *Rack* input and asymmetric for all others.

The second example is a one-bit memory implemented using the LP$_{SR}$2/2 style, as shown in Figure 12. A and C represent the input and output channels. Channel B provides internal storage. S is an input control channel that selects the write or read operation. When S0 is high, the memory stores the value at the input channel A to the internal storage B; both A and S channels are acknowledged. When S1 is high, the memory is read, and the result is made available on the output channel C. At the same time, the S input channel is acknowledged.

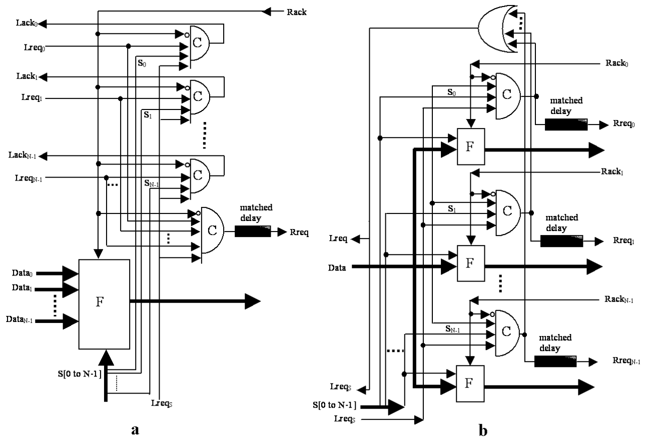


Figure 11. a) Conditional read and b) write.

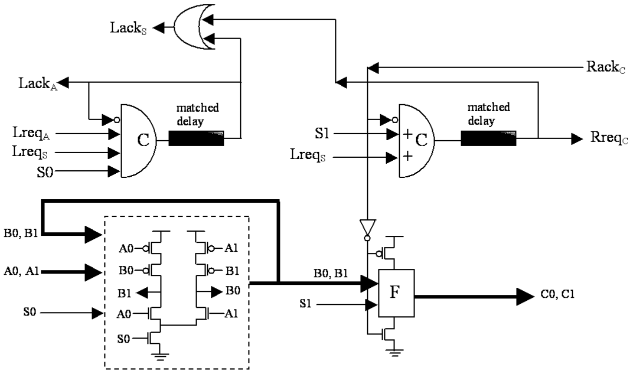


Figure 12. A one-bit LPSR2/2 memory

The detailed implementation of the storage element is shown in the dotted box (similar to [11]). Assuming that there is some data value stored initially, one of the dual-rail bits of B is high and the other is low. When an input A is applied and S0 is asserted, if the value of input A is different from the stored value B, then first both rails of B are lowered (i.e. old memory content is erased), and then one of the two B rails is asserted high, thereby storing the new data. On the other hand, if the value to be written into the memory and the value already stored are identical, then no further action is taken. The C-element, which generates the acknowledgment of the input channel *LackA*, is reset using its own output, since it doesn't receive an acknowledgment from any other channel. The output of this C-element is passed through a delay whose latency matches to the delay of writing the internal node B, to allow sufficient time for the data value to be stored in the memory.

8. Simulation Results

HSPICE simulations were performed to quantify the overhead of accommodating non-linear datapaths compared with linear datapaths. In particular, simulations were performed for the fork and join structures for each of the three pipeline styles considered: LP$_{SR}$2/2, LP3/1, and HC.

The simulations were performed on pre-layout schematic designs, using a 0.25 TSMC process with a 2.5V power supply at 25^{o}C. The purpose of these simulations was only to do a relative comparison of the performance of linear and non-linear pipeline templates. Hence, no attempt was made to fine-tune the transistor sizing to achieve optimum performance. In particular, all transistors were sized in order to roughly achieve a gate delay equal to a small inverter (Wnmos=0.8um, Wpmos=2um, and L=0.24um) driving a same-sized inverter. For the purposes of this comparison, wire delay also has been ignored.

The results of simulation are summarized in Table 1. The cycle times (in ns) are given for each of three styles, first for a linear pipeline, then for a pipeline with a fork, and finally for a pipeline with a join. The columns labeled "Sol1" give results for those designs that are derived using the first solution strategy, *i.e.*, by making only local changes to the stages immediately next to the fork or join point (see Section 3.1). Similarly, the columns labeled "Sol2" give results for designs that use the second solution strategy, where all of the pipeline stages must use modified completion detectors. Note that while the joins add only ~5% to the cycle time, the forks increase the cycle time by ~20% because of the additional C-element needed. Note also that the cost of the more robust solution 2 compared to solution 1 is generally less than 5%.

	LP$_{SR}$2/2		**LP3/1**		**HC**
	Sol1	*Sol2*	*Sol1*	*Sol2*	*Sol2*
Linear	0.99	1.06	1.20	1.28	0.93
Fork	1.23	1.29	1.41	1.45	1.20
Join	1.05	1.10	1.27	1.34	1.01

Table 1. Cycle time (ns) of original linear pipelines vs. proposed non-linear pipelines.

9. Conclusions

This paper has introduced new high-speed asynchronous circuit templates for non-linear dynamic pipelines, including forks, joins, and more complex configurations in which channels are conditionally read and/or written. Two sets of templates arise from adapting the LP$_{SR}$2/2 and LP3/1 pipelines and one set of templates arises from adapting the HC pipelines.

Timing analysis and HSPICE simulation results demonstrate that forks and joins can be implemented with a ~5%–20% performance overhead over linear pipelines. All pipeline configurations have timing margins of at least two gate delays, making them a good compromise between speed and ease of design. One possible area of future work is to formalize the specification and design of these templates using relative-timing based synthesis [13]. Moreover, a more detailed comparison with other high-speed non-linear pipeline approaches such as IPCMOS [3], GasP [4], and pulse-mode [5][14] would be interesting.

References

[1] K.S. Stevens, S. Rotem, R. Ginosar, P. A. Beerel, C.J. Myers, K.Y. Yun, R. Kol, C. Dike, M. Roncken, "An asynchronous instruction length decoder," in IEEE JSSC, Volume: 36 Issue: 2, Feb. 2001 pp. 217–228.

[2] W.S. Coates, J.K. Lexau, I.W. Jones, S.M. Fairbanks, and I.E. Sutherland. "FLEETzero: an asynchronous switching experiment," in Proc. of ASYNC, 2001, pp. 173–182.

[3] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. "Asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz," in ISSCC 2000, pp. 292–293.

[4] I. Sutherland, and S. Fairbanks. "GasP: a minimal FIFO control," in Proc. of ASYNC, 2001, pp. 46–53.

[5] Mika Nystrom. Asynchronous Pulse Logic. Ph.D. thesis, Caltech, 2001.

[6] M. Singh, and S.M. Nowick. "High-throughput asynchronous pipelines for fine-grain dynamic datapaths," in Proc. of Intl. Symp. on Adv. Res. in Asynchronous Circ. and Syst. (ASYNC), 2000, pp. 198–209.

[7] M. Singh, and S.M. Nowick. "Fine-grain pipelined asynchronous adders for high-speed DSP applications" in Proc. of IEEE Computer Society Annual Workshop on VLSI, Orlando, FL, April 2000, pp. 111–118.

[8] M. Singh, and S.M. Nowick. "MOUSETRAP: Ultra-High-Speed Transition-Signaling Asynchronous Pipelines" in Proc. of Intl. Conf. on Computer Design (ICCD), Austin, TX, September 2001.

[9] T.E. Williams, and M.A. Horowitz. "A Zero-overhead self-timed 160ns 54b CMOS divider," in ISSCC Digest of Technical Papers, 1991, pp. 98-296.

[10] Ted Eugene Williams. Self-Timed Rings and their Application to Division. Ph.D. thesis, Stanford University, May 1991.

[11] Andrew Matthew Lines. Pipelined Asynchronous Circuits. M.Sc. thesis, California Institute of Technology, June 1995, revised 1998.

[12] Charles L. Seitz. "System Timing," in Carver A. Mead and Lynn A. Conway, editors, Introduction to VLSI Systems, chapter 7. Addison-Wesley, 1980.

[13] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers." In IEICE Transactions on Information and Systems, Volume: E80-D, No: 3, March 1997.

[14] Luis A. Plana and Stephen H. Unger. "Pulse-Mode Macromodular Systems," in Proc. of Intl. Conference on Computer Design (ICCD), pp. 348-353, October 1998.

[15] Christian D. Nielsen. "Evaluation of Function Blocks for Asynchronous Design," in Proc. of EURODAC, pp. 454-459, 1994.

[16] V.I. Varshavsky (ed.). Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems. Kluwer Academic Publishers, Dordrecht, The Netherlands, January 1990.