High-Speed Dynamic Start-Stop Pipelines

M. Rajesh Babu, CH. Venkata Bhikshapathi, CH. Aishwarya, B. Ashok, K. Sumal & D. Thirupathi Rao

Department of Computer Science and Engineering, Bapatla Engineering College

Abstract – The most recent approaches proposed for high-speed dynamic pipelines are applicable only to linear datapaths. However, real systems are dynamical 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%).

I. INTRODUCTION

High-speed dynamic 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. The 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 [1][2], as well as high-speed static logic pipeline templates [3], that achieve comparable performance with much less stringent timing assumptions.]. In addition, an initial approach to handling slow or stalled environments for the limited case of linear pipelines was also proposed in [1]. However, the synchronization problems that arise when using arbitrary forks and joins are much more complex and challenging, and the approaches of [1][2] do not address these issues. This paper attempts to fill this void and 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 data paths, and reviews some of the basic linear pipelines of [1] and [2]. Section 3 gives an overview of some of the challenges involved in the design of dynamic pipelines.

Sections 4-6 present the new dynamic 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.

II. BACKGROUND

This section first gives background on commonly-used start-stop data representation schemes. Then, it reviews three: Start-stop pipelining styles: (i) LPSR2/2, 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 start-stop system design is to decompose the system into functional units that communicate data via channels, as shown in Fig 2.1(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 Fig 2.1(b). The associated channel is called a bundled-data channel [7]. Alternatively, in *dual-rail* encoding, the data is sent using two wires for each bit of information, as shown in Fig 2.1(c) [5]. 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.

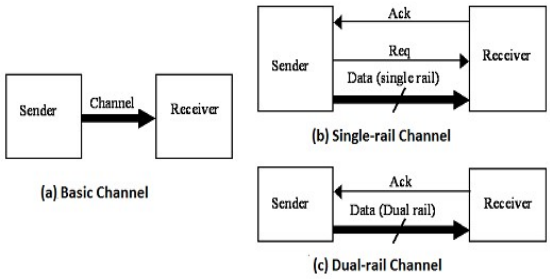

Fig 2.1 : Pipeline Channels

2.2 Lookahead Pipelines (Single-Rail)

Fig 2.2(a) shows the structure of one stage of the LP$_{SR}$2/2 lookahead single-rail pipeline [1]. 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 [1], 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.

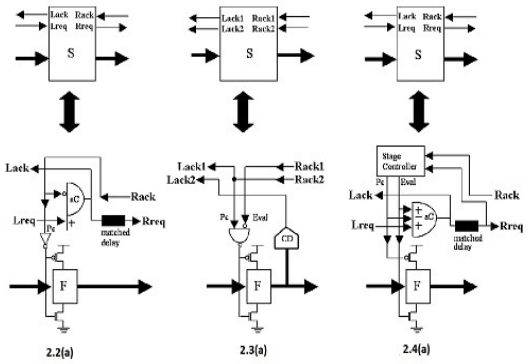

Fig 2.2 (a) LPSR2/2, Fig 2.3(a) LP3/1 and Fig 2.4(a) HC pipelines

2.3 Lookahead Pipelines (Dual-Rail)

Fig 2.3(a) shows the structure of one stage of the dual-rail LP3/1 pipeline [1]. 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 [4][5], 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.

2.4 High-Capacity Pipelines(Single-rail)

Finally, the structure of one stage of the HC pipeline [2] is shown in Fig 2.4(a). 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.

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).

Much like in LP$_{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 [3], 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.

III. CHALLENGES OF HANDLING FORKS AND JOINS

There are two basic challenges involved in designing dynamic pipelines: (i) synchronization of a stage with multiple *destinations* (e.g., for forks), (ii) synchronization of a stage with *multiple sources* (e.g., for joins).

3.1 Slow or Stalled Right Environments in Forks

In many existing linear Start-stop pipelines such as Williams' and Horowitz' classic PS0 pipeline [5], 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.* If the precharge of previous stage is assumed to be fast then 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. Th*us non-persistence is usually not problematic in linear pipelines: all stages can be reasonably assumed to react fast enough to acknowledgment pulses [4][5].

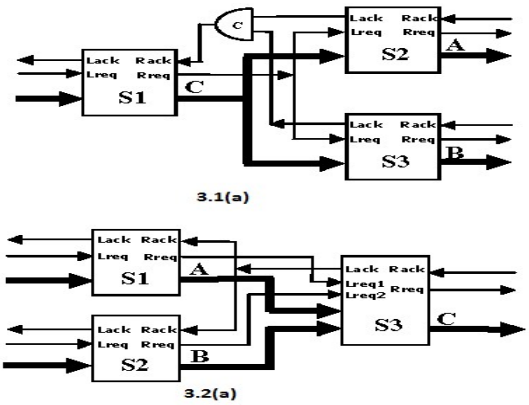

Fig 3.1. (a) Forks and Fig 3.2 (a) joins

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.

As an example, Fig 3.1(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.

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.

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 Fig 3.2(a). 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 [4]). 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 [5][6][7][10][11]. However, the term "weak-conditioned" is often used in a somewhat more restrictive sense than "non-eager".

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.

IV. LOOKAHEAD PIPELINES (SINGLE-RAIL)

Handling joins in single-rail lookahead pipelines is straightforward, and was initially proposed in [2]. 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. 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.

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 Fig 4.1 the *Lack* generation logic is made persistent and the control of the foot transistor is also modified.

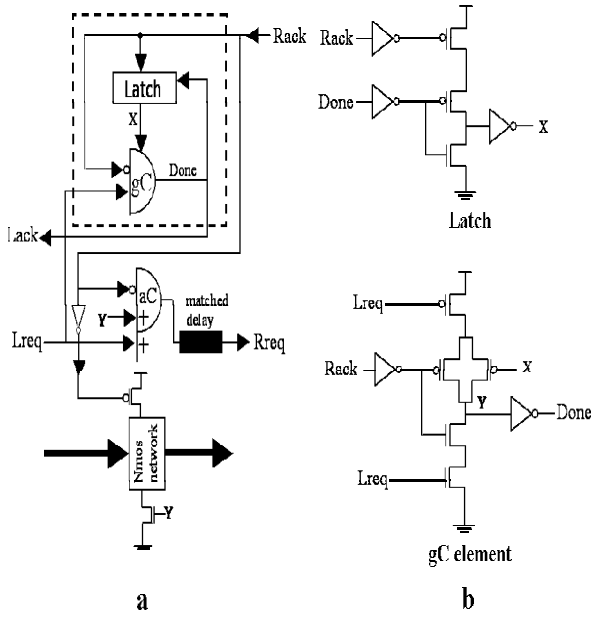

Fig 4.1. a) Modified first stage after the fork. b) Detailed implementation of individual gates

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. . 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.

4.2 Solution 2 for 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 LP**SR**2/2 template shown in Fig 4.2 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.

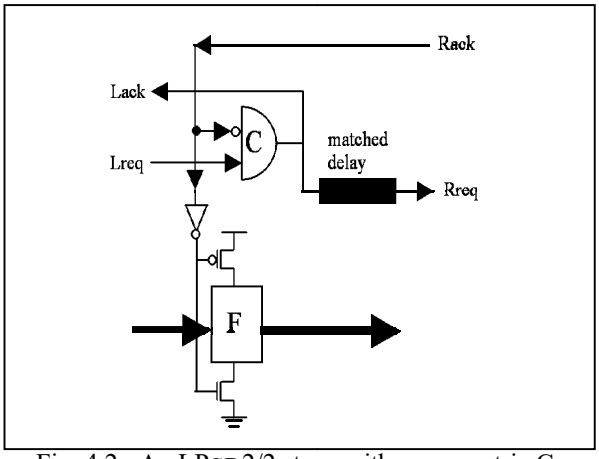

Fig. 4.2 : An LP$_{SR}$2/2 stage with a symmetric C-element

V. 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

The 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 Fig 5.1. 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.

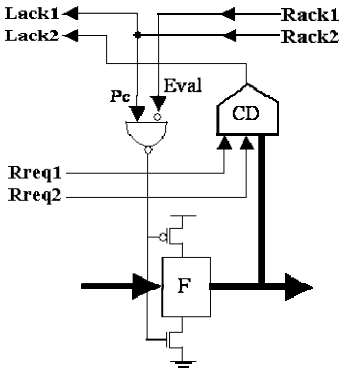

Fig. 5.1 : The LP3/1 pipeline with a modified CD to handle joins

5.2 Forks

To handle forks, as in the single-rail lookahead pipeline, 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 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 Fig 5.2.1. 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.

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 Fig 5.2.2. 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.

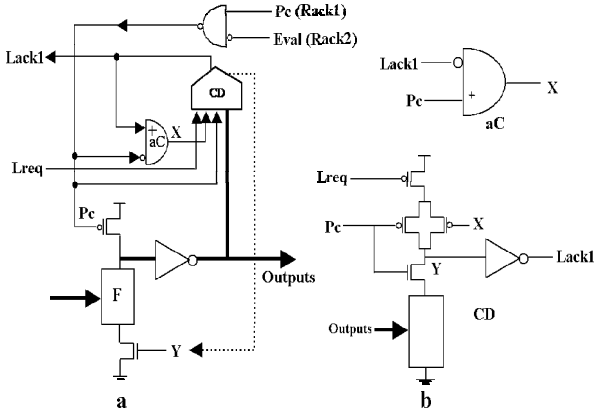

Fig 5.2.1.a) Modified first stage after the fork. b) Detailed implementation of the additional gates

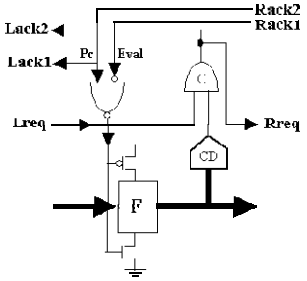

Fig 5.2.2 : The LP3/1 stage with a C-element

VI. HIGH-CAPACITY PIPELINES (SINGLE-RAIL)

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

Fig 6.1 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 Fig 6.1(b).

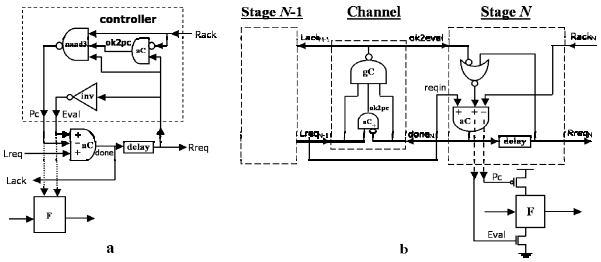

Fig 6.1. a) Original and b) New HC stage

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.

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. Fig 6.2 shows the implementation of a template for stage, *N*, for the case where stage *N* is both a fork as well as join.

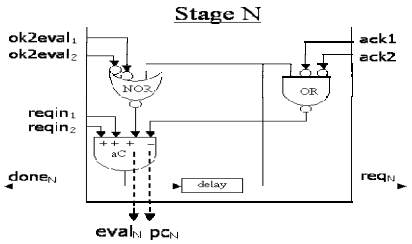

Fig 6.2 : A 2-way join 2-way fork HC stage

VII. CONDITIONALS

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, Fig 7.1(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, Fig 7.1(b) shows a conditional write, where the stage reads from an input channel, and writes to one of several output channels.

The second example is a one-bit memory implemented using the LP**SR**2/2 style, as shown in Fig 7.2. 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.

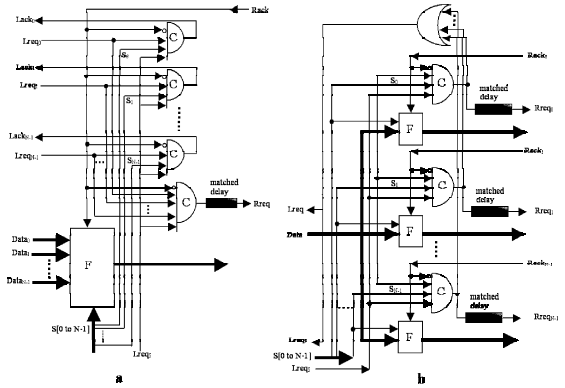

Fig 7.1. a) Conditional read and b) write

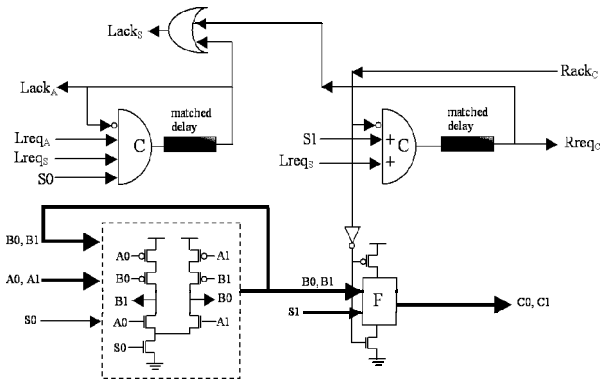

Fig 7.2 : A one-bit LPSR2/2 memory

VIII. SIMULATION RESULTS

HSPICE simulations were performed to quantify the overhead of accommodating dynamic datapaths compared with linear datapaths. 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. 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.

	LPSR2/2		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 dynamic pipelines.

IX. CONCLUSIONS

This paper has introduced new high-speed Start-stop circuit templates for dynamic 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 [8].

X. REFERENCES

[1] 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.

[2] 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

[3] 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

[4] 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.

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

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

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

[8] 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.

[9] 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.

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

[11] 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.

◈◈◈