Low-Overhead Testing of Delay Faults in High-Speed Asynchronous Pipelines

Gennette Gill, Ankur Agiwal & Montek Singh
Dept. of Computer Science
UNC Chapel Hill
Chapel Hill, NC 27599, USA
{gillg,ankur,montek}@cs.unc.edu

Feng Shi & Yiorgos Makris
Electrical Engineering Dept.
Yale University
New Haven, CT 06520, USA
{feng.shi,yiorgos.makris}@yale.edu

Abstract

We propose a low-overhead method for delay fault testing in high-speed asynchronous pipelines. The key features of our work are: (i) testing strategies can be administered using low-speed testing equipment; (ii) testing is minimally-intrusive, i.e. *very little testing hardware needs to be added; (iii) testing methods are extended to pipelines with forks and joins, which is an important first step to testing pipelines with arbitrary topologies; (iv) test pattern generation takes into account the likely event that one delay fault causes several bits of data to become corrupted; and (v) test generation can leverage existing stuck-at ATPG tools.*

In describing our testing strategy, we use examples of faults from three very different high-speed pipeline styles: MOUSETRAP, GasP, *and high-capacity (*HC*) pipelines. In addition, we give an in-depth example—including test pattern generation—for both linear and non-linear* MOUSETRAP *pipelines.*

1. Introduction

Most asynchronous high-speed pipeline styles achieve high performance by making timing assumptions, thereby sacrificing some timing robustness. Even if these timing assumptions are verified during design, they may be violated in practice due to delay faults caused by manufacturing variations and defects. Therefore, the ability to test for delay faults in a fabricated chip containing high-speed pipelines is critical to inspire confidence in the use of asynchronous hardware.

Delay faults are quite challenging to test, both in terms of test application and test pattern generation. Many delay faults occur only under certain operating conditions that seem difficult to create using typical low-speed testing equipment. In addition, a single delay fault may cause errors on one or several wires at once, leading to non-deterministic error behavior that is difficult to test.

Several approaches [1, 6, 9] for testing delay faults have been proposed earlier, but they are intrusive, *i.e.,* they require additional test circuitry to be added to the pipeline. The extra circuitry has an area overhead typically proportional to the number of stages in the pipeline. In addition, the added circuitry is typically on the critical path of each stage's cycle, thereby negatively impacting the pipeline's performance.

In this paper, we propose a new approach for delay fault testing of asynchronous pipelines, which overcomes the limitations of existing approaches. Our approach directly addresses the acknowledged challenges to delay fault testing: activating "at-speed" delay faults using low speed testing equipment, and handling non-deterministic error behavior. Moreover, our approach applies not only to linear pipelines, but also to those with forks and joins; this is a first step towards development of testing methods for pipelines with arbitrary topology. Finally, we also introduce a method of test pattern generation that can expose errors on multiple wires caused by a single delay fault. Our pattern generation approach maps delay fault testing to stuck-at-fault testing of a dual circuit, thereby allowing us to leverage existing stuck-at ATPG tools.

A key beneficial feature of our approach is that it is minimally-intrusive, and hence low-overhead, yet it can create the operating conditions necessary to expose delay faults using only low speed testing equipment. In fact, our approach is non-intrusive for linear pipelines; no additional circuitry is required to be added in order to test them. Although some additional circuitry must be added for full fault coverage of pipelines that have forks and joins, the total number of extra gates added is proportional to the number of forks and joins present, not to the total number of stages.

The remainder of the paper is organized as follows. Section 2 discusses previous work on asynchronous pipeline testing, including stuck-at-fault testing for MOUSETRAP. It also provides background on three asynchronous pipeline styles which we will use to illustrate our approach: MOUSETRAP [11], GasP [12], and high-capacity (HC) pipelines [10]. Section 3 proposes a classification of timing constraints into two key categories—forward and reverse—and proposes test strategies for delay faults that occur when these constraints are violated. Section 4 extends our testing strategies to paths containing forks and

joins. Section 5 applies our test strategy to linear MOUSETRAP pipelines, and introduces a test pattern generation method which can test for delay faults that cause errors on an undetermined number of wires. Section 6 addresses testing of MOUSETRAP pipelines containing forks and joins. Finally, Section 7 offers conclusions and directions for future work.

2. Previous Work and Background

This section discusses previous work in the testing of asynchronous pipelines and presents background on three high-speed asynchronous pipeline styles—MOUSETRAP, GasP, and high-capacity (HC)—which are used in the remainder of the paper to illustrate our test approach.

2.1. Previous Work

Several methods have been proposed for testing asynchronous micropipelines. Pagey *et al.* [5] proposed methods for generating test patterns to test stuck-at faults in traditional micropipelines, but do not consider delay faults. A more recent approach to testing micropipelines is by [2], which focuses on test sequence generation for testing both stuck-at and delay faults inside C-elements.

Several authors [1, 6] have proposed full-scan approaches for testing micropipelines. Though these full-scan methods test both stuck-at faults and violations of the bundled delay constraint, they are high-overhead in both area and speed. Roncken *et al.* [7, 8] proposed a more optimal approach that employs partial scan. Their approach targets faults in both the control and datapath, and covers not only stuck-at fault testing, but also bridging faults as well as IDDQ testing.

Recently, van Berkel *et al.* [14] have proposed adding synchronous as well as LSSD modes of operation to asynchronous circuits in order to make them testable. In addition, te Beest *et al.* [13] present an approach that uses only multiplexers, instead of latches, to break feedback loops, and thus leverage combinational test pattern generation without the overhead of scan latches. Both of these approaches were proposed mainly in the context of the Tangram/Haste design flow. Finally, Kondratyev *et al.* [3] have proposed a test approach for NCL circuits.

None of the above approaches, however, addresses the specific test needs of fine-grain high-speed asynchronous pipeline styles.

Very recently, Shi *et al.* [9] have presented an approach specifically targeted to testing high-speed asynchronous pipelines. Testing of both stuck-at and delay faults is addressed. There are two limitations of their approach to delay fault testing: (i) it is intrusive and therefore has area and performance overheads, and (ii) it uses a limited delay fault model that assumes the fault will only affect a single bit.

The approach of this paper overcomes the limitations of [9] and provides a minimally-intrusive fault testing strategy. It also handles a larger class of delay faults, in which a single fault can cause errors on one or more bits simultaneously.

2.2. Background: Asynchronous Pipeline Styles

2.2.1. MOUSETRAP Pipelines. MOUSETRAP [11] is a two-phase pipeline style that uses static logic. Since MOUSETRAP uses transition signaling, every transition on a request wire indicates new data is ready and every transition on an acknowledge wire indicates that old data can be overwritten. Thus, when the request and acknowledge going into a stage are the same, the stage becomes "empty" and when they are different the stage becomes "full". The latches that hold data begin open and close just after new data arrives.

Two simple one-sided timing constraints must be satisfied for the correct operation of the pipeline: *setup time* and *data overrun.*

In order for data to be latched properly in stage N, the data must arrive at least one setup time before the latch closes. Figure 1 (a) illustrates this scenario. The path, $p1$, that closes the latch begins when the request arrives, passes through the bit latch, and finally through the XNOR. The timing constraint is therefore: $t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{\mathrm{Latch}_N} + t_{\mathrm{XNOR}_N\downarrow}$.

Once data enters a stage N, it should be securely captured before new data is produced by the previous stage. If this condition is violated, the data held in stage N will be overwritten by new data. Figure 3 (a) illustrates the two relevant paths. The path, $p1$, that closes the latch in stage N contains only one XNOR gate. The path, $p2$, that produces data in $stage_{n-1}$ contains one XNOR, one latch, and combinational logic. The timing constraint between these two paths is: $t_{\mathrm{XNOR}_{N-1}\uparrow} + t_{\mathrm{Latch}_{N-1}} + t_{\mathrm{logic}_{N-1}} > t_{\mathrm{XNOR}_N\downarrow} + t_{\mathrm{hold}}$

2.2.2. GasP Pipelines. GasP [12] is a four-phase pipeline style that uses static logic. The data latches in a GasP pipeline begin closed, and must open and then close again upon receiving each new data item. The most distinctive feature of GasP is that a single wire, called the state conductor, is used to transmit *both* the request and acknowledge signals between a pair of adjacent stages. A low signal on the state conductor wire indicates that the previous stage is"full" and a high signal indicates that it is "empty".

The controller for GasP is a self resetting NAND, as shown in Figure 1. When both of the inputs to the NAND go high, it goes low. After some delay, this low transition triggers the NAND to reset back to high. Specifically, two possible reset paths, r_1 and r_2, can cause the NAND to go high again. Path r_1 consists of one pull up transistor and one inverter (inv_a). Path r_2 consists of one inverter (inv_b) and one pull down transistor. The time, t_{reset}, to reset the NAND is the *smaller* of the two times: $t_{\mathrm{reset}} = min(t_{\mathrm{pull\_up}} + t_{\mathrm{INVa}\downarrow}, t_{\mathrm{INVb}\uparrow} + t_{\mathrm{pull\_down}})$.

To explain the operation of the GasP pipeline, we first focus on the forward flow of data through the pipeline and then on the reverse flow of empty space. Although GasP has many timing constraints, we only discuss here two that are relevant to our work.

In the forward path, consider the scenario of data and a request arriving to an empty stage, stage N. In order for data to be latched correctly, it must arrive at least one setup time before the latch N closes. Figure 1(b) shows the paths involved in this scenario. The path, $p2$, that closes the latch N consists of one inverter, one NAND, and the self reset path of the NAND. This timing constraint is therefore expressed as: $t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{\mathrm{inva}_N} + t_{\mathrm{NAND}_N} + t_{\mathrm{reset}_N} + t_{\mathrm{INVc_N}\downarrow}$

In the reverse path, data in stage N should be securely captured before new data is produced by stage $N-1$. Figure 3 (b) shows the relevant paths. The path, $p1$, that closes the latch in stage N is the self reset path of the NAND. (For easier interpretation, only one possible branch of this path is shown in the figure.) The path, $p2$, that allows new data to enter consists of one pull down transistor, one NAND, one latch delay, and the delay of any combinational logic in the stage. This leads to the timing constraint: $t_{\mathrm{reset}_N} + t_{\mathrm{INVc_N}\downarrow} + t_{\mathrm{hold}_N} < t_{\mathrm{pull\_up}_n} + t_{\mathrm{NAND}_{n-1}\uparrow} + + t_{\mathrm{INVc_{N-1}}\uparrow} + t_{\mathrm{Latch}_{n-1}} + t_{\mathrm{logic}}$

2.2.3. High-Capacity Pipelines. The high-capacity (HC) pipeline [10] is a four-phase pipeline style that uses dynamic logic. It is *latchless*. Instead, the dynamic logic of each stage has an "isolate phase", in which its output is protected from further input changes. Specifically, during the isolate phase, the logic is neither precharging nor evaluating.

An HC pipeline stage simply cycles through three phases. After it completes its evaluate phase, it 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.

New data arriving to an empty stage (i.e., a stage in the evaluation phase) must arrive one setup time before the stages switches from the evaluation phase to the isolate phase. Figure 1 (c) shows the paths involved. The path, $p1$ that causes the isolate phase begins when a request arrives. The request propagates through the asymmetric c-element, then through the matched delay and to the controller. The internal logic of the controller (an inverter) causes the evaluation phase to end. This constraint is represented as: $t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{\mathrm{aC}_N} + t_{\mathrm{delay}_N} + t_{\mathrm{INV}_N}$

To protect new data from stage $N-1$ from erroneously overwriting data in stage N, stage N must enter the isolate phase one hold time before stage $N-1$ provides new data. Figure 3 (c) shows the paths involved. The path, $p1$, that causes the isolate phase includes the matched delay and the internal controller logic (an inverter). The path, $p2$, that triggers new data includes the controller logic to begin precharge (a NAND), the asymmetric C-element, the matched delay, and the controller logic to begin evaluation (an inverter). This leads to the timing constraint: $t_{\mathrm{delay}_N} + t_{\mathrm{INV}_N} + t_{\mathrm{hold}_N} < t_{\mathrm{NAND}_{N-1}} + t_{\mathrm{aC}_{N-1}} + t_{\mathrm{delay}_{N-1}} + t_{\mathrm{INV}_{N-1}} + t_{\mathrm{Eval}_{N-1}}$

3. Test Approach

This section introduces our test strategy for high-speed asynchronous pipelines. It focuses on testing delay faults, since test methods for stuck-at faults have already been proposed in [9].

The section begins by identifying two key categories of timing constraints that may result in delay faults—*forward* and *reverse* constraints—in Section 3.1, and then proposes test strategies for each. Section 3.2 then gives functional test methods for loading and unloading a pipeline that are most likely to expose these delay faults.

The test approach is generalized in Section 4 to handle forks and joins, which represents a first step toward handling systems with arbitrary topologies. Test pattern generation is deferred till Section 5.

The test strategy is illustrated using examples of timing constraints from three different pipeline styles: MOUSETRAP, GasP, and high-capacity (HC) pipelines.

3.1. Delay Fault Classification

For the purposes of designing fault tests, we have identified two key categories of timing constraints: *forward* and *reverse* constraints.

A *forward* timing constraint is one that requires a downstream event to occur after an upstream event, *i.e.,* in the same direction as the flow of data. More specifically, the pipeline operates correctly only if a particular event in stage N occurs before a particular event in stage $N+1$. Typical examples of forward constraints include *setup time requirements* of many pipelines: the latch in stage $N+1$ must "capture" a data item and be disabled only after the output of stage N has been generated and stabilized. Such forward constraints typically prevent situations where a data item is not correctly transmitted from one stage to next. Violations of forward timing constraints are likely to manifest when a data item is allowed to travel through an uncongested or empty pipeline; such a scenario allows downstream events to occur unimpeded, thereby exposing violations of the forward constraint.

A *reverse* timing constraint, on the other hand, is one that requires two events to be ordered such that the upstream event must occur after the downstream event, *i.e.*, counter to the flow of data. More specifically, the pipeline operates correctly only if a particular event in stage N occurs before a particular event in stage $N-1$. Typical examples of reverse contraints are *hold time requirements* of many pipelines: once a data item is received in stage N, the latch in stage N must be disabled before new data is generated by stage $N-1$. Such reverse constraints typically prevent current data from being erroneously overwritten by new data. Therefore, violations of reverse timing

Pipeline Style	Timing Constraint
MOUSETRAP	$t_{\text{data}_{N-1}} + t_{\text{setup}_N} < t_{\text{req}_{N-1}} + t_{\text{Latch}_N} + t_{\text{XNOR}_N\downarrow}$
GasP	$t_{\text{data}_{N-1}} + t_{\text{setup}_N} < t_{\text{req}_{N-1}} + t_{\text{inva}_N} + t_{\text{NAND}_N} + t_{\text{reset}_N} + t_{\text{INVc}_N\downarrow}$
HC	$t_{\text{data}_{N-1}} + t_{\text{setup}_N} < t_{\text{req}_{N-1}} + t_{\text{aC}_N} + t_{\text{delay}_N} + t_{\text{INV}_N}$

Table 1: Forward Constraint examples

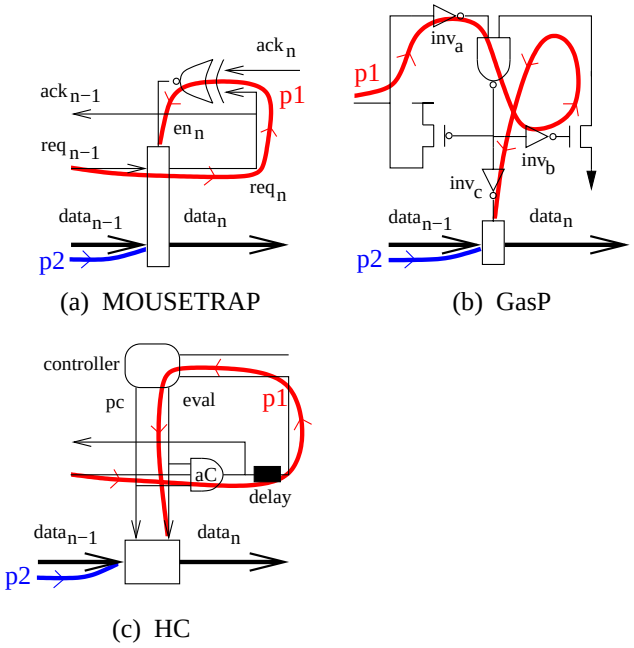

Figure 1: Examples of forward timing constraints.

constraints are likely to be exposed when one data item follows closely behind another.

In some pipeline styles, timing constraints exist that do not fit into either of these categories. For example, GasP has a timing constraint that relates two events within a single pipeline stage. We have not yet devised a standard method for testing for violations of these types of timing constraints.

3.2. Test Strategy for Linear Pipelines

3.2.1. Forward Delay Faults.

Examples. Many pipeline styles have forward delay constraints, most commonly in the form of the bundled data constraint. Table 1 gives the forward timing constraints that we identified for MOUSETRAP, GasP, and HC pipelines in Section 2.2. For clarity, Figure 1 represents these constraints graphically.

The constraints necessary for proper operation are actually less strict than the bundled data constraint since there is some delay between the time the request arrives and when the data is actually needed. In particular, strictly speaking, the bundled data constraint requires that $t_{\text{data}_{N-1}} < t_{\text{req}_{N-1}}$. However, the setup time constraints shown in the table and the figure are somewhat more relaxed: in both MOUSETRAP and GasP, the data must arrive at stage N one setup time before the latch closes; in HC pipelines, the data must arrive one setup time before the evaluation phase ends (*i.e.*, isolate phase begins). Thus, the constraints identified are more properly referred to as setup time constraints.

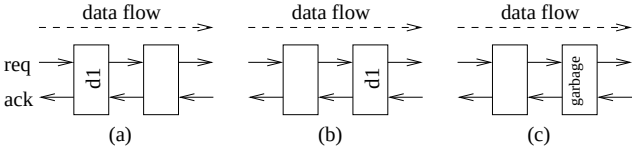

Figure 2: a) Beginning state b) Correct end behavior c) Behavior with forward delay fault

Violations of these timing constraints actually manifest themselves as pipeline malfunction only when there is a bubble in stage N, *i.e.*, stage N is ready to process the new data from stage $N-1$. This is so because the above setup time constraints were derived assuming the worst-case scenario that stage N is ready to capture the data arriving from the previous stage. In each of the three pipeline styles, if stage N is not ready to receive new data, then more time is available for that data to arrive from stage $N-1$ and stabilize at stage N's inputs before it must be captured. In particular, in MOUSETRAP and GasP, if stage N is not empty, its latch remains disabled. Similarly, in HC pipelines, if stage N is not empty it will not commence its next precharge-evaluate cycle.

Testing Approach. The observation that stage N must be empty for a setup time violation to manifest itself helps us construct a test for these forward delay faults. We first present a scenario that exposes these faults, and then present a functional test strategy to efficiently test an entire pipeline.

Figure 2 illustrates a scenario in which setup time violations are likely to be exposed. Assume the scenario begins with the snapshot of the pipeline shown in Figure 2(a): stage N is empty and stage $N-1$ has data item $d1$. If the pipeline operates correctly, the data item $d1$ flows to the right, as shown in snapshot (b). However, if the setup time constraint between the two stages is violated, the data item $d1$ will not be properly captured (*i.e.*, latched for MOUSETRAP and GasP, or evaluated for HC) by stage N. In the case of MOUSETRAP and GasP, the data item $d1$ will be corrupted by any stale data that was previously in stage N, since some or all of the bits of $d1$ could not be properly latched. For HC pipelines, the result in stage N could be all zeros (*i.e.*, the result of the previous precharge), or partial evaluation of some of the bits before evaluation was prematurely interrupted.

In order to test the entire pipeline for setup time violations, the above scenario must be created at each stage in the pipeline. Once again, the recent test approach of [9] accomplishes this goal by added circuitry to the pipeline for greater controllability, but the intrusiveness of that approach

Table 2: Examples of reverse timing constraints: Analytical expressions.

Pipeline Style	Timing constraint
MOUSETRAP	$t_{\text{XNOR}_N\downarrow} + t_{\text{hold}_N} < t_{\text{XNOR}_{N-1}\uparrow} + t_{\text{Latch}_{N-1}} + t_{\text{logic}_{N-1}}$
GasP	$t_{\text{reset}_N} + t_{\text{INVc}_N\downarrow} + t_{\text{hold}_N} < t_{\text{pull\_up}_N} + t_{\text{NAND}_{N-1}} + t_{\text{INVc}_{N-1}\uparrow} + t_{\text{Latch}_{N-1}} + t_{\text{logic}_{N-1}}$
HC	$t_{\text{delay}_N} + t_{\text{INV}_N} + t_{\text{hold}_N} < t_{\text{NAND}_{N-1}} + t_{\text{aC}_{N-1}} + t_{\text{delay}_{N-1}} + t_{\text{INV}_{N-1}} + t_{\text{Eval}_{N-1}}$

causes loss of performance during actual operation.

Our strategy, once again, is to create this error-exposing situation functionally, without the need for intrusive circuitry. We start with an empty pipeline. Next, we feed one data item to the pipeline. As the data item travels forward, it creates the condition shown in 2 for each stage. Specifically, the setup time constraint between stages 1 and 2 is tested first, between 2 and 3 next, and so on all the way to the rightmost stages in the pipeline.

A setup time violation results in corruption of data, and is easily checked functionally if the test data patterns are carefully chosen so as to propagate this fault to the output end of the pipeline. This topic is covered in detail in Section 5.

3.2.2. Reverse Delay Faults.

Examples. Examples of reverse constraints can be found in many pipeline styles. Table 2 gives the reverse timing constraints that we identified for MOUSETRAP, GasP, and HC pipelines in Section 2.2. For clarity, these constraints are graphically shown in Figure 3.

Each of the constraints shown in Table 2 and Figure 3 are hold time constraints. In particular, the constraint for MOUSETRAP states that, once a data item enters stage N, that stage's latch must be disabled at least a hold time (t_{hold_N}) before the previous stage can perturb the data at stage N's inputs. The constraint for GasP similarly ensures that stage N's latch is disabled at least a hold time before new data arrives from stage N. Finally, the constraint for HC ensures that stage N, upon evaluating a data item, enters its *isolate phase* (during which the dynamic logic in that stage is neither precharging nor evaluating) before stage $N-1$ goes through a complete new cycle of precharge and evaluation to generate a new data item.

Violations of these timing constraints actually manifest themselves as pipeline malfunction only when a new data item is readily available for processing by stage $N-1$. This is so because the above hold time constraints were derived assuming the worst-case scenario of a steady stream of input data. For instance, in MOUSETRAP, a violation of the above timing constraint causes an actual pipeline malfunction only if there is actually a new data value or request (or both) on the input to stage $N-1$. Similarly in GasP and HC,

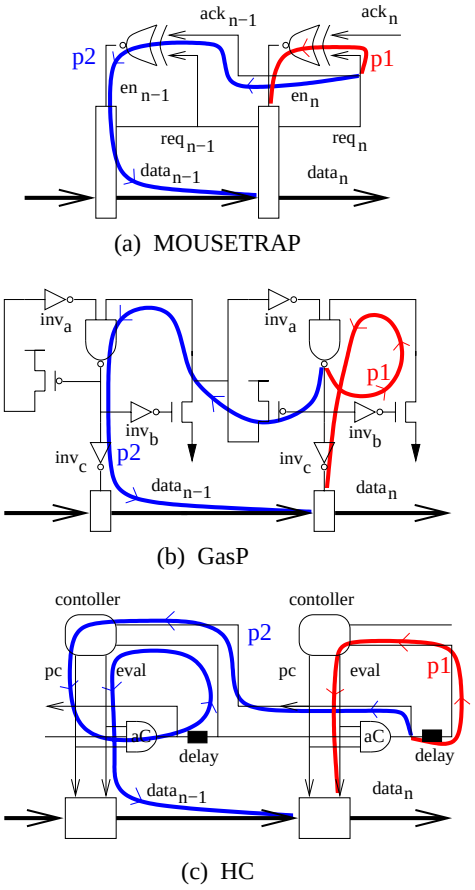

Figure 3: Examples of reverse timing constraints

a timing violation causes an error only if there is new data available on the input side of the datapath of stage $N-1$.

Testing Approach. The observation that a new data item must be waiting at the input side of stage $N-1$ in order for a hold time violation to manifest itself helps us construct a test for these reverse delay faults. We first present a scenario that exposes these faults, and then present a functional test strategy to efficiently test an entire pipeline.

Figure 4 illustrates a scenario in which hold time violations are likely to be exposed. Assume the scenario begins with the snapshot of the pipeline shown in Figure 4(a): there are two data items, $d1$ and $d2$ in stages $N-1$ and $N-2$, respectively, and there is a bubble in stage N. If the pipeline operates correctly, the two data items remain distinct as they flow to the right, taking the pipeline through the snapshots (b) and (c). However, if the hold time constraint between the two rightmost stages is violated, the data item $d1$ will be overwritten by the item $d2$, as shown in snapshot (d).[1]

In order to test the entire pipeline for hold time violations, the above scenario must be created at each stage in

1 For simplicity of presentation, we assume that when a hold time violation occurs, a data item is completely overwritten by the next data item. Strictly speaking, it is possible that only some bits of a data item are overwritten. Section 5 addresses this more general scenario.

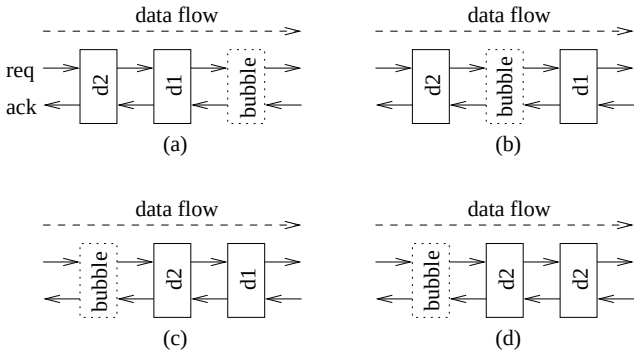

Figure 4: a) Beginning stage b) bubble propagates backwards c) Correct end behavior d) Behavior with reverse delay fault

the pipeline. The recent test approach proposed in [9] accomplishes this goal by adding circuitry to the pipeline to provide adequate controllability. However, that approach is intrusive and creates overheads to the normal functioning of the pipeline, thereby resulting in a loss of performance. Moreover, the intrusiveness of that approach also modifies the timing constraint itself, thus perturbing the very constraint it is trying to verify.

Our strategy instead is to create this error-exposing situation functionally, without the need for intrusive circuitry. We start with an empty pipeline. Next, we fill the entire pipeline with data, while withholding acknowledgments to the rightmost stage. We then add one bubble to the right end of the pipeline by sending a single acknowledgement. As this bubble propagates leftward through the pipeline, it sequentially creates the situation shown in Figure 4 for each stage. Specifically, the hold time constraint between stages $N-1$ and N is tested first, followed by the constraint between stages $N-2$ and $N-1$, and so on all the way to the leftmost stages in the pipeline.

A hold time violation results in loss or corruption of data, and is easily checked functionally if the test data patterns are carefully chosen so as to expose, and not mask, hold time violations, and propagate them to the output end of the pipeline. This topic is covered in detail in Section 5.

There are several key benefits of our approach. First, unlike [9], our approach is non-intrusive, with no overhead to steady-state performance. Second, our approach can test hold time faults for an entire pipeline in a single sweep, *i.e.,* by propagating a single bubble backward through the pipeline. In contrast, the approach of [9] must test for hold time violations at each pipeline stage individually, thereby requiring significantly greater test effort. Finally, much like the approach of [9], our approach only requires low-speed ATE, yet provides at-speed testing for delay faults.

4. Test Strategy for Non-Linear Pipelines: Handling Forks and Joins

Thus far, we have only discussed testing strategies for faults in straight pipeline paths. Here, we introduce extensions to the test strategy to handle pipelines with forks and joins, which is a first step toward handling systems with arbitrary topologies.

4.1. Forward Delay Faults

Challenges. Testing forward delay faults offers some challenges. Specifically, setup time violations between the join element and all of its immediate predecessors are difficult to test. If data from one branch arrives at the join much after data from the other, only the later-arriving data will expose setup time violations. The earlier-arriving data will have had sufficient time to stabilize before the join reacts, thereby masking any setup time violations between that branch and the join stage.

Testing Approach. To ensure testing of setup time faults between a join stage and each of the branches that feed into it, we need to be able to control which branch provides a data item first. Our strategy is to modify the pipeline by inserting extra circuitry immediately before each join, which allows the testing environment to directly provide data items to the input side of the join, as shown in Figure 5.[2] Specifically, we add multiplexors to both the control and the data input of each join stage, which allows data to be fed into the join externally during testing. They key idea is to provide data externally for each input interface of a join stage, *except* for the input interface that is to be tested for setup time violations.

Testing takes place as follows. Assume we are testing the interface between the join stage and the upper branch of the pipeline shown in Figure 5(a). First we initialize the pipeline to be empty. Then we set the multiplexors in the circuitry of Figure 5(b) on the lower branch of the join so that data is read from the testing environment and sufficient time is allowed for this data to be read and stabilized. At this point, the setup time constraint between the lower branch and the join stage is trivially satisfied because the join will not latch this data until data also arrives from the upper branch; therefore, a potential setup time violation between the upper branch and the join stage is now exposed and testable. The test proceeds by sending a single data item into the pipeline from its left interface, which flows rightward through the pipeline, effectively exposing setup time faults throughout the upper branch, through the join and all the way through the right end of the pipeline. Finally, we read the output from the pipeline and examine it for errors. The test is then repeated for the lower branch that feeds into the join.

4.2. Reverse Delay Faults

Challenges. Testing reverse delays presents two challenges: (i) "unbalanced" branches, *i.e.,* forked paths whose

2 In order to conserve the number of input pins used by the test environment, externally data is typically supplied bit-serially, and converted to bit-parallel on chip before being supplied to the circuit under test.

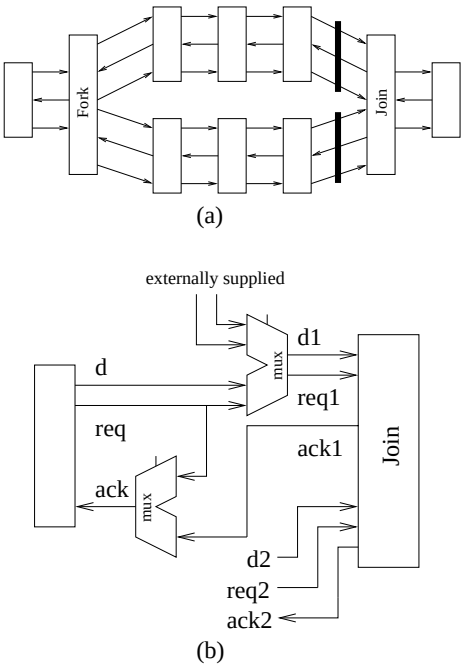

Figure 5: a) Extra circuitry for forward delay. b) Generic implementation

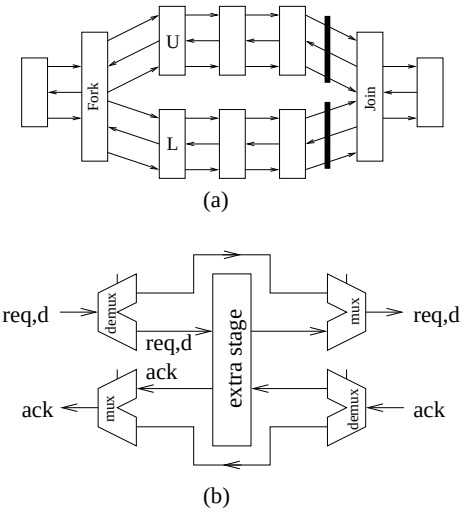

Figure 6: a) Extra circuitry for reverse delay. b) Generic implementation

branches have unequal number of stages, are not fully testable using only functional methods; and (ii) fork stages themselves are difficult to test robustly for hold time violations. These challenges are briefly discussed before our testing approach is introduced.

Unbalanced branches: It is challenging to functionally test pipelines with forked paths that reconverge if the branches have unequal number of stages, *i.e.,* if the branched paths are not "slack matched" [4]. In particular, our reverse delay fault testing approach, presented above for linear pipelines, relies on filling the entire pipeline with data items, and then propagating a bubble backward through it. If unbalanced reconvergent forks exist in the pipeline, then not all branches can be completely filled with data, and therefore our functional test strategy cannot be directly applied to the longer of the branches.

In more detail, assume that a fork creates two branches, $B1$ and $B2$, which reconverge. Assume branch $B1$ has $N1$ stages and branch $B2$ has $N2$ stages. Then, if $N1 > N2$ then the first $N1 - N2$ stages of branch $B1$ cannot be made to hold data while our test is applied, rendering them functionally untestable using our method. However, it is important to note that the shorter of the two branches, $B2$, is still fully testable for reverse delay faults, a fact that will be exploited by our test approach.

Fork stages: If all fork branches are balanced, then our test approach for linear pipelines can be easily adapted to test for hold time violations throughout the non-linear pipeline, *except* to test for faults between the fork stage and its immediate successors. This difficulty is because of the unpredictability of the order of arrival of bubbles at the fork stage from each of its successors. In particular, if one branch of a 2-way fork propagates a bubble towards stage N faster than the other branch, then only the later-arriving bubble can actually expose a delay fault upon reaching the fork stage. The earlier arriving bubble will not be able to expose a hold time violation because the fork stage, which is waiting for the other bubble to arrive as well, cannot immediately generate a new data item. Therefore, to fully test for hold time violations, greater controllability of pipeline operation is required in order to control the order of arrival of bubbles at a fork stage from different fork branches.

Testing Approach. We now present our test strategy, which addresses both of the above challenges.

Unbalanced branches: Our approach avoids the problem due to unbalanced branches by requiring that all reconvergent paths are balanced (*i.e.*, slack matched). This is easily done by either inserting buffer stages in the shorter branch, or by pipelining the longer branch more coarsely using fewer stages. Fortunately, slack matching is already considered good design practice for performance reasons [4], so requiring it for ease of delay fault testability is not too onerous a requirement.

Fork stages: If full testability is required for hold time violations between a fork stage and all of its immediate successors, then we need to modify the circuit such that we can control the order of arrival of bubbles at that fork stage. Our strategy is to add a small amount of circuitry to the pipeline, as shown in Figure 6. Specifically, on each of the branches, we add an extra pipeline stage, which is externally configurable to act as either a regular pipeline stage (*i.e.,* capable of storing one data item or bubble) or as pass-though logic with no storage capability.

Essentially, we are configuring one branch to have N stages and configuring the other branch to have $N + 1$ stages. Thus, we have deliberately made the branches *unbalanced* during testing, a technique that allows the shorter

branch to be fully testable. This procedure is then repeated to test the other branch.

For example, assume we are testing the for delay faults between the fork stage and the stage U in Figure 6. First we initialize the pipeline to be empty. Then, we set the external control signals such that the extra circuitry in the upper branch behaves as pass-through logic, but the extra circuitry on the lower branch behaves as a regular pipeline stage. Thus, during this test, the upper branch is shorter than the lower one.

Next, we load a test sequence of eight data elements through the left interface of the pipeline, without providing acknowledgments at the right interface, and wait for a sufficient time for the pipeline to stabilize. At this point the upper branch is completely filled, whereas the lower branch has one bubble in stage L.

Then, we remove one data item from the right end of the pipeline, thereby creating a bubble that propagates leftward through the pipeline. Since there is already a bubble in stage L, the fork will be ready to react quickly when the bubble reaches stage U, thereby exposing any reverse delay faults between stage U and the fork stage.

Errors are easily detected by reading out all of the data from the pipeline and examine it for functional correctness. This testing procedure is then repeated to similarly test for reverse delay faults between stage L and the fork stage.

5. Test Example I: Linear MOUSETRAP

This section applies the testing strategies for linear pipelines given in Section 3 to MOUSETRAP. For each of the possible delay faults in a MOUSETRAP pipeline, either our forward or reverse fault testing strategy is applied. A key contribution is a test pattern generation approach which can help expose these delay faults.

MOUSETRAP exhibits two distinct timing constraints that must be satisfied for correct operation, one forward (setup time) and one reverse (hold time), as was discussed earlier in Sections 2.2. A violation of the first constraint can lead to data corruption because the data fails to arrive a setup time before the latch is disabled. A violation of the second constraint can lead to *two* distinct failure scenarios: "control overrun" (*req*$_N$ is overwritten by *req*$_{N+1}$) and "data overrun" (*data*$_N$ is overwritten by *data*$_{N+1}$). Test methods and pattern generation for testing these delay faults for linear MOUSETRAP are now presented.

5.1. Forward Delay Faults

Setup Time Fault. Violations of the setup time requirement are tested using our test approach introduced for testing forward delay faults in Section 3.2.1. Specifically, we initialize the pipeline to be empty, then feed test data to the pipeline from the input side, and read the data from the output side of the pipeline to determine if a delay fault exists. Determining test data patterns to expose such faults, however, is not trivial.

Recent work [9] has introduced a method for test pattern generation to expose this type of fault. Their method separately tests for setup time violations for each data bit for each pipeline stage. Specifically, it suggests sending one test pattern to initialize that bit to 0, and then sending a second test pattern that checks for a stuck-at-0 fault on that bit. If there is a setup time violation at that bit location, then the bit value will remain at 0 and behave as if it were stuck at 0; otherwise, the bit is correctly set to 1 by the second test pattern. Thus, their work nicely leverages stuck-at ATPG tools to help test for setup time violations.

There is, however, a significant limitation in the test pattern approach of [9]: it relies on the assumption that the setup time fault will affect a single bit in the stage being tested. In practice, however, if one of the lines coming out of a stage is corrupted due to a setup time violation, the other lines coming out of it are likely to be corrupted as well. The combination of multiple errors in one stage could easily lead to the fault's not propagating to the end of the pipeline.

Our approach overcomes this limitation of [9] and can correctly test for setup time faults that may affect multiple bits simultaneously. In particular, our test pattern generation strategy exposes a setup time violation for a particular bit (say, bit k) in a given stage (say, stage N) regardless of whether or not other bits are also affected by that fault. The key idea is to generate pairs of test data items (say, I_1 and I_2) which satisfy the following criteria: (i) they generate values (say, D_1 and D_2) at stage N which differ in only the k-th bit, and (ii) they produce different outputs at the right end of the pipeline.

Test approach. Our test method consists of two steps: sending the first test data item (I_1) through an empty pipeline and removing it from the other end, followed by sending the second test data item (I_2). If criteria (i) is met, a setup time violation can *only* cause the bit k to change, since all other bits are the same between I_1 and I_2 If criteria (ii) is met, then a fault for bit k in stage N is propagated to the output, and is therefore functionally testable. This procedure is repeated for all bits in all pipeline stages.

ATPG setup. Our approach to generating the test pattern sequence (I_1, I_2) leverages existing ATPG tools for stuck-at fault testing. Specifically, we map the problem of finding the test patterns I_1 and I_2 that will expose a setup fault at bit k in stage N in the pipeline into an equivalent problem of testing for a stuck-at-0 fault at the output of the dual circuit shown in Figure 7. The block labeled "Hamming-1$_k$" produces 1 if its inputs differ in only the k-th bit; otherwise it generates 0.[3] Therefore, any input pattern $I_1 I_2$ that tests for output stuck-at-0 fault also satisfies the conditions required for the sequence (I_1, I_2) to be a test pattern for exposing a setup time violation in bit k in stage N. In par-

3 The block tests if its inputs have a Hamming distance of 1, and the only different bit is bit k.

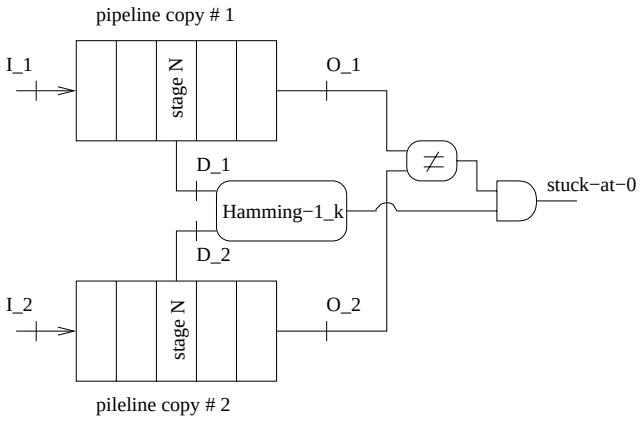

Figure 7: ATPG for testing setup time violations.

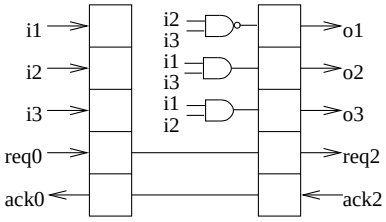

Figure 8: A two-stage pipeline with one level of combinational logic.

ticular, the inequality tester ensures that the two input patterns will generate different pipeline outputs, and the Hamming distance checker ensures that the intermediate values in stage N differ in only bit k. Finally, a test for stuck-at-0 at the output of the AND gate effectively ensures that the test patterns I_1 and I_2 that are generated satisfy the conjunction of these conditions.

Example. Figure 8 shows a two-stage pipeline with one level of combinational logic. To check setup time faults for the bit *o1*, our ATPG method yields a test sequence consisting of the following two data items: $i_1 i_2 i_3 = 011$ followed by $i_1 i_2 i_3 = 010$. Table 3 shows the correct and faulty behaviors.

5.2. Reverse Delay Faults

The reverse delay fault in MOUSETRAP occurs due to a violation of the hold time requirement. This faults can manifest itself in two flavors: *control overrun* and *data overrun.*

Control Overrun. Control overrun occurs when the hold time constraint for MOUSETRAP (see Table 2) is violated for the incoming *req* signal. That is, once a stage receives a data item along with a request, it fails to disable its latch before a new request overwrites the current request. Since this is a reverse delay fault, we use the testing strategy outlined in 3.2.2. Specifically, we fill the pipeline with data and then insert one bubble at the right end of the pipeline, which exposes faults as it propagates backwards.

The error behavior for control overrun is that two requests, and therefore two pieces of data, are lost. This is

	S.No.	1	2
Test Pattern	req0	1	0
	ack2	0	1
	(i1,i2,i3)	(0,1,1)	(0,1,0)
Correct Behavior	ack0	1	0
	req2	1	0
	(o1,o2,o3)	(0,0,0)	(1,0,0)
Faulty Behavior	ack0	1	0
	req2	1	0
	(o1,o2,o3)	(0,0,0)	(0,0,0)
	Comment	send input for o1=0	send input for o1=1

Table 3: Test patterns for checking setup time fault for line $o1$ in Figure 8.

because MOUSETRAP uses transition signaling. For example, Figure 9 indicates how six requests will be interpreted as only four requests in the presence of a fault.

We can test for this fault quite easily in a linear pipeline. First we fill the pipeline with N data items, where N is the length of the pipeline, while withholding acknowledgments from the right end of the pipeline. Then we present the $N+1$-th data item to the pipeline, although at this point it will not be accepted by the pipeline. Now, every stage in the pipeline has new data available to its left. Next, we remove one item from the right end of the pipeline, effectively introducing one bubble which flows leftward to expose any hold time faults. When this bubble reaches the leftmost pipeline stage, the $N+1$-th data item is accepted by the pipeline. Thus, if there is no fault, the pipeline will once again contain N data items at this point. If there were any hold time violations, then two items will have been lost for each violation, and therefore only $N-2$ or fewer items would remain in the pipeline. The correct and faulty behaviors are easily distinguished: the correct pipeline is full and will not accept any new data, but the faulty pipeline will accept two or more data items.

As an example, Figure 9 shows the behavior of a 3-stage linear pipeline that has a control overrun fault in $stage_2$. First we supply four data items along with their requests; then we remove one data item to cause error behavior. Specifically, *req2* falls while *en2* is still asserted, thereby allowing an incorrect request through. This error is exposed when two subsequent requests are acknowledged by the pipeline. Table 4 shows the test pattern we used in exposing this error. In general, the test sequence for a control overrun fault on a linear N-stage pipeline contain $N+3$ test patterns.

Data Overrun. Unlike control overrun, data overrun does not result in the loss of requests. Instead, data becomes corrupted. Specifically, this hold time violation causes data item D_N to be overwritten partially or entirely with data item D_{N+1}. In MOUSETRAP, data overrun occurs when the hold time constraint is violated only

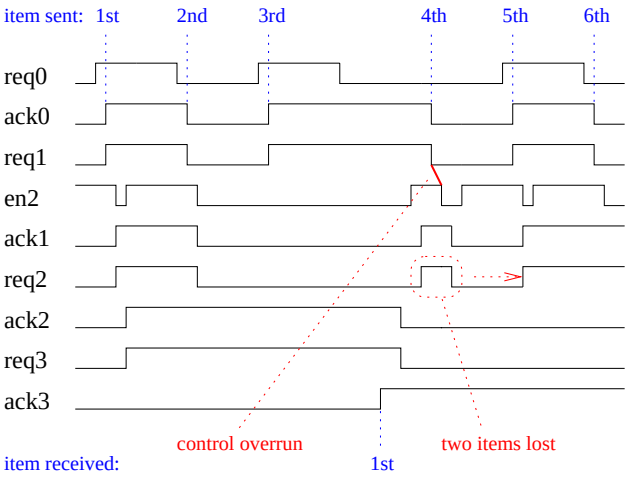

Figure 9: Timing diagram showing control overrun

S.No.	Test Pattern (req0,ack3)	Correct Behavior (ack0,req3)	Faulty Behavior (ack0,req3)	Comment
1	(1,0)	(1,1)	(1,1)	apply 1st req
2	(0,0)	(0,1)	(0,1)	apply 2nd req
3	(1,0)	(1,1)	(1,1)	apply 3rd req
4	(0,0)	(1,1)	(1,1)	apply 4th req
5	(0,1)	(0,0)	(0,0)	add one bubble
6	(1,1)	(0,0)	(1,0)	apply 5th req, observe ack1

Table 4: Test pattern for control overrun

for the latches that capture the data bits and not for the tiny latch that captures the associated request; this scenario is quite possible because data can arrive from the previous stage much earlier than its associated request.

The test for data overrun is more complex than that for control overrun because test patterns must be generated carefully to expose data corruption, as opposed to simply detecting loss of data items. As with setup time faults, if a data overrun fault affects one data bit of a stage, it is likely to affect other bits of the same stage as well. Therefore, testing sequences must be generated for data overrun faults in a manner similar to those generated for setup time faults in Section 5.1.

Interestingly, the ATPG approach of Figure 7 also applies to generating tests for the reverse delay fault. Suppose we are testing stage N for a data overrun fault. In order to begin testing, we must fill the pipeline up to and including stage N with some data; these values do not affect the testing of stage N. The next two data items fed into the pipeline are the same patterns generated by the method of Figure 7: I_1 followed by I_2. Then, one item is removed from the right end of the pipeline, and a bubble propagates backward exposing the delay fault for bit k in stage N. If the pipeline operates correctly, stage N will correctly latch the value D_1; otherwise it will latch the value D_2. The correct and faulty scenario are therefore easily distinguished at the output of the pipeline.

6. Test Example II: Non-Linear MOUSETRAP

This section describes how to test the delay faults in MOUSETRAP in the presence of forks and joins. Setup time faults and data overrun faults use the same test pattern generation methods as their linear counterparts described in Section 5. For control overrun faults, however, we must use a different test pattern than the one given for linear pipelines.

6.1. Setup Time Fault

Since the setup time fault is a forward delay fault, we use the testing strategy introduced for in Section 4.1. Specifically, for full fault coverage we add logic that allows the join stages to be tested. Test patterns for forked paths can be generated in the same way as for straight paths, as described in Section 5.1

6.2. Control Overrun

Since control overrun is based on a reverse timing constraint, testing forked paths faces the challenges given in Section 4.2. Specifically, we must ensure that forked paths are balanced and add logic to allow testing of the fork stages.

The input patterns for testing control overrun faults for straight pipelines given in Section 5.2 will not work in the presence of forks. Recall the the linear pipeline test relied on being able to detect if the pipeline has lost two (or more) data items after being filled. However, if the fault occurs on only one branch of a fork, the branch *without* the fault will not lose any data items, and will thereby prevent the loss of data in the other branch from being externally observable. In particular, the correctly functioning branch will not allow more data items from being accepted from the environment even though the faulty branch contains bubbles.

For better fault coverage in pipelines with forks, we propose a different strategy. Instead of determining if the pipeline has lost any data items due to delay faults by trying to feed it more items, we determine any losses by reading out the entire contents of the pipeline and counting the number of items read. If fewer than N items emerge from an originally filled pipeline, then some data has been lost due to control overrun.

Figure 10 shows an example of a forked pipeline to be tested. Table 5 shows a test pattern that we use to test for control overrun faults. In the correct scenario, all N of the acknowledges (*ack9*) sent to the right interface of the pipeline lead to further requests coming out of the pipeline (*req9*). In the faulty scenario, only $N-2$ of the acknowledges lead to new requests being generated.

6.3. Data Overrun

Since data overrun is based on a reverse timing constraint, we use the test setup given in 4.2. Specifically, we must ensure that forked paths are balanced and add logic to allow testing of the fork stages. Test patterns for forked paths can be generated in the same way as for straight paths, as described in Section 5.2.

S.No.	Test Pattern			Correct Behavior			Faulty Behavior			Comment
	req0	ack9	d0	ack0	req9	d9	ack0	req9	d9	
1	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	1	(1,1,1,1)	send 1st item, recv 1st item
2	0	0	(0,0,0,0)	0	1	(1,1,1,1)	1	1	(1,1,1,1)	send 2nd item
3	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	1	(1,1,1,1)	send 3rd item
4	0	0	(0,0,0,0)	0	1	(1,1,1,1)	1	1	(1,1,1,1)	send 4th item
5	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	1	(1,1,1,1)	send 5th item
6	0	0	(0,0,0,0)	0	1	(1,1,1,1)	1	1	(1,1,1,1)	send 6th item
7	1	0	(1,1,1,1)	0	1	(1,1,1,1)	1	1	(1,1,1,1)	send 7th item
8	1	1	(1,1,1,1)	1	0	(0,0,0,0)	1	0	(0,0,0,0)	recv 2nd item
9	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	1	(1,1,1,1)	recv 3rd item
10	1	1	(1,1,1,1)	1	0	(0,0,0,0)	1	0	(0,0,0,0)	recv 4th item
11	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	**0**	(1,1,1,1)	recv 5th item
12	1	1	(1,1,1,1)	1	0	(0,0,0,0)	empty: req9 same			recv 6th item
13	1	0	(1,1,1,1)	1	1	(1,1,1,1)	empty: req9 same			recv 7th item

Table 5: Test pattern for control overrun on a forked path

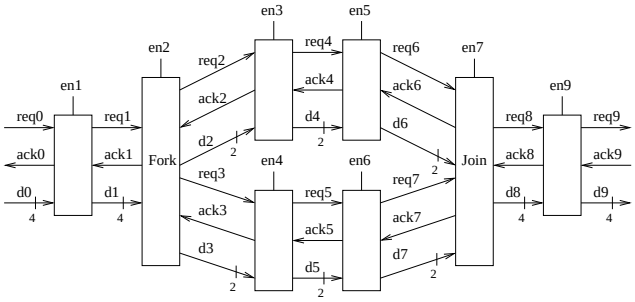

Figure 10: A non-linear MOUSETRAP pipeline

7. Conclusions and Future Work

This paper presented a strategy for testing delay faults in high-speed asynchronous pipelines, including pipelines with forks and joins. We showed that our testing strategies are applicable by giving examples from three different pipeline styles: GasP, MOUSETRAP, and HC. Our testing strategy is non-intrusive for linear pipelines and minimally-intrusive for pipelines containing forks and joins. In addition, all of our tests can be conducted using low-speed testing equipment.

We also demonstrated one specific application of our strategy. Using MOUSETRAP as an example, we showed how our testing strategies work for straight paths and also paths with forks and joins. We also showed how to generate test patterns for MOUSETRAP that are robust to the non-deterministic nature of errors caused by delay faults. This method leverages past ATPG tools.

There are several natural extensions to our work. Our method applies to forward and reverse constraints; the next step is to identify other kinds of timing constraints and develop the corresponding test strategies. In testing pipelines with forks and joins, this work took one step towards testing pipelines with arbitrary topologies. We still need to incorporate testing of more complex pipeline stages, such as data dependent conditional forks. In addition, we plan to apply the testing strategies presented here to circuits with some non-trivial combinational logic.

References

[1] A. Khoche and E. Brunvand. Testing micropipelines. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 239–246, Nov. 1994.

[2] M. King and K. Saluja. Testing micropipelined asynchronous circuits. In *Proc. Int. Test Conf.*, 2004.

[3] A. Kondratyev, A. Streich, and L. Sorensen. Testing of asynchronous designs by inappropriate means: Synchronous approach. In *Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2002.

[4] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Pénzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, Sept. 1997.

[5] S. Pagey, G. Venkatesh, and S. Sherlekar. Issues in fault modeling and testing of micropipelines. In *Proc. of the Asian Test Symp.*, Hiroshima, Japan, Nov. 1992.

[6] O. A. Petlin and S. B. Furber. Scan testing of micropipelines. In *Proc. IEEE VLSI Test Symp.*, pages 296–301, May 1995.

[7] M. Roncken, E. Aarts, and W. Verhaegh. Optimal scan for pipelined testing: An asynchronous foundation. In *Proc. Int. Test Conf.*, pages 215–224, Oct. 1996.

[8] M. Roncken and E. Bruls. Test quality of asynchronous circuits: A defect-oriented evaluation. In *Proc. Int. Test Conf.*, pages 205–214, Oct. 1996.

[9] F. Shi, Y. Makris, S. M. Nowick, and M. Singh. Test generation for ultra-high-speed asynchronous pipelines. In *Proc. Int. Test Conf.*, Nov. 2005.

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

[11] M. Singh and S. M. Nowick. MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines. In *Proc. Int. Conf. Computer Design (ICCD)*, pages 9–17, Nov. 2001.

[12] I. Sutherland and S. Fairbanks. GasP: A minimal FIFO control. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, Mar. 2001.

[13] F. te Beest and A. Peeters. A multiplexor based test method for self-timed circuits. In *Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2005.

[14] K. van Berkel, A. Peeters, and F. de Beest. Adding synchronous and LSSD modes to asynchronous circuits. In *Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2002.