Towards Explaining the Fault Sensitivity of Different QDI Pipeline Styles

Patrick Behal, Florian Huemer, Robert Najvirt, Andreas Steininger, and Zaheer Tabassam
Institute for Computer Engineering, TU Wien, Vienna, Austria
{pbehal, fhuemer, rnajvirt, steininger, ztabassam}@ecs.tuwien.ac.at

***Abstract*—Asynchronous circuits, specifically those using a quasi delay-insensitive (QDI) implementation are known for their high resilience against timing uncertainties. However, their event-based operation principle impedes their temporal masking capability, making them more susceptible to fault-induced transitions caused by single event transients. While synchronous circuits obtain high resilience through temporal masking that is established through the sampling of data by flip flops, asynchronous circuits, by design must be flexible about the phases of data validity leaving a larger attack surface for faults. Consequently, previous work has proposed to narrow down the windows in which data changes are accepted, in order to improve the temporal masking in QDI designs.**

In this paper, we analyze the fault sensitivity of asynchronous QDI circuits when subjected to single event transients. We do so by performing extensive fault injection experiments into different buffer styles to identify parameters that are the main contributors to the fault sensitivity of the circuit and compare their resilience.

For that purpose, we use two variants of a multiplier circuit as target circuits. One with the shift and add operations arranged in a linear pipeline, and another one with an internal ring structure that computes the result iteratively, yielding designs with the same logic and buffer implementations, yet very different modes of operation. By varying the buffer styles, we are able to show the difference in robustness as well as the effectiveness of fault mitigation techniques inherent in some buffer styles.

***Index Terms*—asynchronous circuits, SET, fault-tolerance**

I. Introduction

Transient faults have been a threat for the proper operation of microelectronic circuits throughout the past decades. In the regime of shrinking feature size and reduced supply voltage this threat is definitely not losing its relevance. Single event transients caused through particle hits are one often cited example here. While hardening techniques on layout and cell level may reduce the rate of such transients, their occurrence cannot completely be avoided, and hence, systems need to be able to tolerate a certain amount of faults, when employed in safety critical applications. This fault tolerance can be established on different levels, with the circuit level being an important first bastion against fault propagation. For synchronous digital circuits it is well understood that undesired transients can be masked by low-pass filtering (electrical masking), logic functions that temporarily disable certain signal paths (logical masking) and by flip flops ignoring the state of their input during certain time windows (temporal masking). There is a wealth of explicit fault-tolerance techniques like duplication and comparison, coding, TMR and others, that explicitly leverage and extend such masking. The main weakness of synchronous design, however, is its high sensitivity to timing-related faults. With the aggressive reduction of timing margins in the interest of high performance, along with their inability to self-adapt their speed of operation, synchronous circuits are particularly vulnerable by faults that cause a temporal displacement of transitions in any way

Asynchronous circuits, specifically quasi-delay insensitive (QDI) circuits, have a very flexible, self-adaptive timing and hence promise to be very robust in the time domain. At the same time, their self-timed, transition-centric operation principle tends to make them vulnerable to transients. In addition, in the absence of flip flops, their capability of temporal masking needs to be questioned, and also their potential for logical masking appears to suffer from the indication principle which largely prohibits the masking of signal paths. So their vulnerability in the value domain seems to be higher than in the synchronous case – which may be compensated by suitable fault-tolerance techniques on circuit level. Unfortunately, many of the methods from the synchronous world cannot (at least directly) be applied in asynchronous circuits. Still, there is already quite some research on robustness of asynchronous circuits and methods for its enhancement.

Unfortunately, it is hard to combine the existing insights and results into a global picture, as they all have been derived for specific circuits and pipeline types, under specific experimental conditions (or by theoretical analyses) and with specific targets in mind. One vision in our project is to elaborate such a global picture through a large experimental study (complemented by theory) that allows an apples-to-apples comparison of different pipeline styles and fault-tolerance enhancements. Since the masking effects in asynchronous design seem to depend on many operational parameters like the pipeline fill level, or the data being processed, another target is the identification of such factors along with a modeling of their specific influence. On the foundation of this understanding, we can then identify the main vulnerabilities, the most efficient existing enhancement approaches, and finally elaborate further improvements.

In this paper we report about some first important steps in this direction. We present an experimental environment that allows the convenient generation of target circuit descriptions, as well as the fully automated conduction of large gate-level simulation experiments with millions of fault injections, while still providing the ability to precisely reproduce each single

This research was partially supported by the project ENROL (grant I 3485-N31) of the Austrian Science Fund (FWF).

fault injection for closer inspection of interesting cases. Using this tool we perform a detailed comparison of fault effects seen in different QDI pipeline styles, separated for data and control path, based on different target circuit types. We will furthermore leverage the analysis ability of our toolset and investigate an observation that seems counter-intuitive in the first place. While there is still a far way to go towards our vision, the results elaborated so far already allow interesting insights into the specific strengths and weaknesses of different pipeline styles and circuit-level fault-tolerance methods, along with their direct comparison.

II. Background on Asynchronous Design

The key purpose of a design style is to coordinate the hand-over of data items (further called "tokens") from a source to a sink, or from one pipeline stage to the next. In the synchronous paradigm a rigid clock signal dictates when a data token is captured by the sink and when the source can remove that token and present the next one. By introducing a handshake that allows source and sink to communicate, asynchronous design styles achieve a much higher degree of flexibility with respect to their timing. This is specifically true for delay-insensitive (DI) circuits, where an appropriate data encoding allows to recognize the completeness of a data token (i.e., when computation of all its bits has been completed) by a special function block, a so-called completion detector (CD). This knowledge can be used to have the sink capture a token right at the moment it becomes valid (i.e., complete). Once finished with capturing, the sink acknowledges the successful reception to the source via a dedicated acknowledgment signal (ack), upon which the source issues the next token for computation. The handshake between source and sink establishes a closed loop that allows the timing to automatically adapt to the circumstances (actual gate and signal delays, actual path length in the computation of the current data token, faults affecting the delay). This makes timing assumptions and restrictions irrelevant for the correct operation of a DI circuit (not for performance, though). However, in practice, it must be ensured that the branches of some wire forks (namely those contained in adversarial paths [1]) adhere to certain relative delay constraints, since the class of "pure" DI circuits is very limited, at least with regard to single-output gates. Such circuits are then referred to as QDI.

The handshake between source and sink always follows a certain pattern (protocol), which can be classified as 2-phase or 4-phase. Fig. 1 shows an example of the latter: After the source issued a data token c_n, the sink replies with a rising transition on ack. Now the source issues a so called "spacer" token, which is nothing more than a return to zero on all data wires, which is answered by ack going to zero as well. Then the cycle starts over. In a 2-phase protocol the spacer would be left out and a subsequent data token would be sent instead, with each (up and down) transition of ack acknowledging the reception. However, compared to the 4-phase approach, 2-phase circuits are a lot more complex, which makes it fairly impractical to implement actual processing logic with them. Hence, in this paper we will focus on the 4-phase protocol.

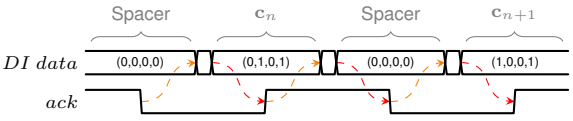


Fig. 1. Sequence of transitions in a 4-phase protocol

Note in Fig. 1 that on the data bus there is an interval in which it carries neither a data nor a spacer token. This is an important feature of the delay-insensitive encoding: In our example (and often in practice as well) the dual-rail encoding is used, with two signal rails ($d.t$ and $d.f$) representing one bit value, which can be *HI* (10), *LO* (01), *null* (00), or *illegal* (11). The data bus in the figure consists of four data rails ($d_1.t$, $d_1.f$, $d_0.t$, $d_0.f$). Hence, the transmitted binary data (d_1, d_0) is given by $(0, 0)$ followed by $(1, 0)$. Note that for a delay-insensitive circuit the sequence in which the transitions on the individual rails occur when going from a *data* token (i.e., one with all dual-rail bits *HI* or *LO*) to a spacer (with all bits *null*) or back must be irrelevant. However, in the phase where one dual-rail bit already changed from *null* to *HI* or *LO* (or vice versa) while the other one did not, the data word is not *complete* and the CD will hence only fire after all bits are in the same phase. This transition phase is essential for the fault sensitivity of a QDI circuit. Its length is determined by delay mismatch along the different signal paths on the individual rails.

III. Related Work

The focus of this work is on effects of transient faults in the value domain, so here we survey models, effects and hardening techniques for dealing with them. In synchronous systems transient faults are efficiently mitigated through masking capabilities that are partly inherent, and, where required, additionally established with fault-tolerance techniques. The latter, however, tend to have large overhead and architecture constraints when ported to asynchronous systems. In [2], e.g., the authors compare the sensitivity of asynchronous logic blocks to transient faults and with that of synchronous ones, and analyze the respective masking effects. In an earlier work, [3] had modified a synchronous concurrent error detection method for asynchronous circuits with two major modifications. First, they synchronized the comparator inputs by handshake signals. Second, a monitor function was used to detect the few cases where the asynchronous comparator fails. A more scalable synchronous concept for obtaining single event transient (SET) tolerance is full or partial replication [4]. In general, some form of redundancy is always required for error detection. [5] elaborates the faults (transient and permanent) and their effects (deadlock, synchronization failure, token generation and consumption) in detail. They propose detection of transient faults using an inverted redundant synchronization channel with an invalid detector (that is provided sufficient time to react). In addition, they handle problematic forks with differing branch

delays through careful layout to enforce the desired relative timing. Three further hardening techniques for transients are presented in [6]: (1) Duplication of computational logic allows them to tolerate single faults with a 2x area overhead and a slight increase in latency. As both logic units share the same input lines, there is no protection against transients on the input. (2) Synchronizing rails of two adjacent data elements (latched only when both show transition). (3) using 1-bit control circuit as synchronizer. A prominent duplication-based approach is to double check (cross coupling) the results of a doubled-up circuit to ensure SET tolerance, as proposed in [7] (more details on this scheme, will be presented below).

Based on duplication topology, a complete radiation hard by design QDI processor (DD1) has been presented in [8]. In [9] the authors use a combination of spatial redundancy and guard gate to harden a controller against radiation. Information redundancy is leveraged in [10] to make a whole processor low power and asynchronous at the same time. In [11] a newly proposed latch topology, based on time redundancy, is compared with duplication, rail synchronization, and basic Muller pipeline, and indeed shows lower fault propagation probability. [12] presents an error detection and fail-stop mechanism by introducing redundant data and acknowledge lines. The proposed technique targets both transient and permanent faults.

The Muller C-element[1], the most important state holding element in asynchronous environments, can by itself mask faults. However, depending on the current circuit state, it can also turn transient glitches at its input into errors, manifested in its internal state. This was elaborated further in [13] who devised a strategy to highlight the fault sensitive parts of a design. For that purpose they define the notion m-sensitivity of an n-input C gate ($m < n$), meaning that the C gate requires a certain number of input state changes to flip its internal state. This strategy allows to investigate the sensitive windows of certain parts of a design.

[14] presents more than 10 SET mitigation techniques with a focus on reducing the sensitive window, making fault consequences easier to detect and resolve, and overcoming the effects of faults. One of those approaches will also be used in our evaluation, and is discussed in more detail below.

In [15] the authors perform fault injection simulations to identify the fault sensitivity windows in several QDI logic styles. Using a special visualization method they compare the fault tolerance of a plain Weak-Conditioned Half Buffer (WCHB) pipeline, as well as two enhanced two half-buffer designs they propose, with a few notable implementations from [14], [16], and [7].

IV. Experiment Setup

The long-term aim of our experiments is to perform an apples-to-apples comparison of the resilience of different QDI pipeline styles, and to analyze their respective strengths and weaknesses. While the former requires a highly automated setup that allows to conduct a statistically significant number of experiments in reasonable time and with little user interaction, the latter calls for well controlled fault injection with detailed logging of cause and effect. In the following we will outline how we tried to achieve both.

A. Target Circuit

A trade-off must be found between a highly realistic, complex target circuit that, however, requires excessive computational performance for running the anticipated high number of fault injections, and a too simplistic target, that impairs the significance of the results. In addition to this dilemma concerning complexity, there does not seem to be any generally agreed single "representative" function or application either. For our experiments we chose a multiplier as target circuit, as it is an elementary function in many applications, and it comprises an appreciable amount of combinational logic – partly in the shape of adders, which by themselves represent another elementary function. Another useful property of the multiplier is its regularity that allows for a choice between linear and iterative (recursive) implementation, and, for the former, different degree of pipelining. From this spectrum we chose to implement two different variants. The first version is a fully pipelined circuit, where each stage calculates a partial product and adds the result to a sum variable. This is illustrated in Fig. 3. The number of pipeline stages is given by the input bit width (plus one additional output buffer).

Fig. 2 shows the circuit variant, which we refer to as the iterative version. Here the partial products are calculated in a feedback loop. Because the hardware to calculate and add up the partial products is shared, the resulting circuit has less area overhead when compared to the pipelined version. However, this also leads to lower throughput and higher latency.

The target circuits are created using a custom (python-based) tool flow based on Production Rule Sets (PRSs). Because of their high regularity the multiplier circuits are directly generated using a script. Delay-Insensitive Minterm Synthesis (DIMS) [17] is used as logic style and ripple-carry adders are utilized to implement the adders. The script based circuit generation allows for an easy change of buffer styles. The PRS is annotated with static (inertial) delays, which we randomly varied by 10% to model PVT variations in the circuit. Wires in the PRS are considered ideal (i.e., zero delay).

The fault injection is also done based on the resulting PRS, i.e., each rule is a potential injection victim. Note, however, that we don't inject faults on (i) primary inputs and (ii) gates driving primary outputs.

So overall we believe that our multiplier represents a reasonably complex target of which we can, thanks to our tool flow, easily generate numerous variants. Of course the high simulation efforts for the controlled fault injection and detailed tracing limit the attainable bit width to (currently) 8. While we are aware that 32 bit or even larger may be desirable, such values are out of reach here. Still we believe

[1]The Muller C-element, or short C gate, is a fundamental gate in asynchronous logic. Its function is to output the logic level seen at its inputs when these match, and to retain the last valid output state otherwise. It can hence also be viewed as an AND gate with hysteresis.

that our results can give initial insights, especially since we also include a 4-bit variant, whose comparison with the 8-bit variant facilitates a first judgment of the impact of bit width on the fault effects.

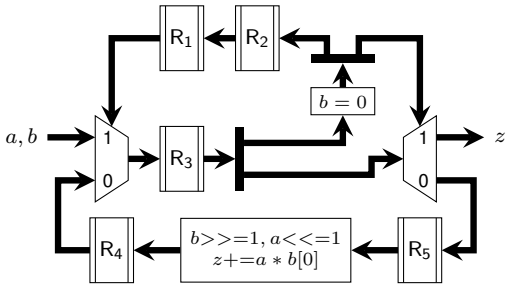


Fig. 2. Iterative multiplier

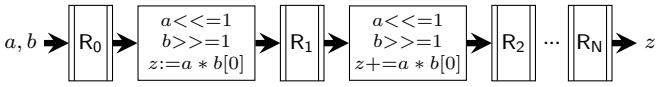


Fig. 3. Pipelined multiplier

B. Considered Buffer Styles

In the context of this paper a buffer is a circuit element, that is able to store either a data or spacer token. It has one dual-rail input and output channel, which also include the respective acknowledgment signals. To generate the acknowledgment for the input channel it must contain a CD. As long as a buffer is well behaved with respect to the 4-phase handshaking protocol, it can seamlessly be swapped out for a different implementation.

We use the WCHB as a point of reference for our experiments, since (i) it is arguably the most simple 4-phase buffer (ii) it is widely used in QDI design and (iii) most other buffer styles we are investigating are basically variations of it. Moreover, it does not not need any special circuit elements apart from the 2-input C gates for its storage elements, which should, however, not be an issue for any asynchronous design.

However, with respect to its fault sensitivity there are a few potential weaknesses associated with this design, which have already been extensively covered by literature [14], [15]. To better understand them, let us have a quick look at its principle of operation. For that purpose we refer to Fig. 4.

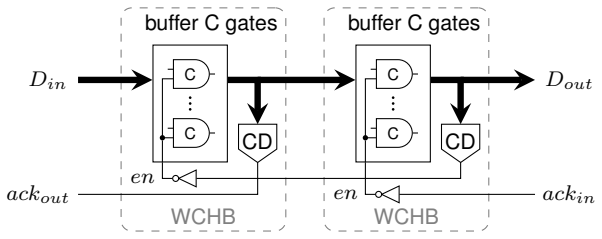


Fig. 4. WCHB pipeline with 2 stages

Assume that in the beginning, the input data rails (D_{in}) and the input acknowledgment (ack_{in}) are zero, i.e., the circuit is in the null phase. Due to the inverter on the lower input of the buffer C gates, the C gates are armed for rising transitions on the data rails. After some input data arrives, some of the C gates will be switched to one (i.e., exactly one for each dual-rail pair). The CD detects this condition and eventually generates the output acknowledgment ack_{out}. At the same time the input data transitions also travel through the whole pipeline, setting the respective C gates in each stage, until the data appears at the output of the pipeline. When the data propagated through the CD in the next stage the acknowledgment to the first stage will be deasserted, arming the C gates for falling input transitions, i.e., the spacer.

Now, the issue with this circuit lies with the intervals in which the C gates are armed for rising transitions. From the very beginning on, all C gates in a buffer are armed for rising transitions, although the input data may arrive with a large input skew. Moreover, depending on feedback delay through the succeeding buffer and CD, there can be quite a long window where the buffer is armed for rising input transitions, although the input data is already complete. During these time windows the buffer is vulnerable to faulty input transitions and accumulates them in its internal state.

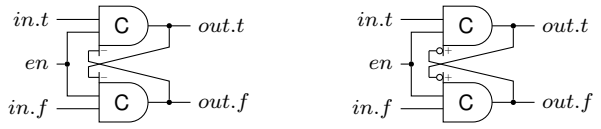
To mitigate these shortcomings various modifications and improvements have been proposed. Most WCHB modifications aim at shortening the time window in which the buffer stores input transitions, in one way or another.

The first two modifications we want to discuss here are, referred to as the *deadlocking* and *interlocking* WCHB, initially proposed in [15]. Fig. 5 shows the respective storage elements (for a single dual-rail bit) used in these buffers. The rest of the buffer is the same as for the classic WCHB. Cross-coupled asymmetric C gates[2], whose outputs are fed back to the asymmetric input of the respective other gate are used to achieve a certain circuit behavior in the presence of faults. If both C gates of a single dual-rail bit get set simultaneously, the feedback inhibits the deadlocking WCHB from entering the null phase again, effectively forcing the circuit into a deadlock. This state can be considered a fail-stop state as it prevents the circuit from processing possibly faulty data.

The interlocking WCHB only allows the first transition at its input to propagate to its output, thus prohibiting passing on of the invalid dual-rail state (11). However, the fact that the buffer always relays and captures the first transition creates the potential of forwarding a wrong value (50% when assuming random faults).

To shorten the sensitive window of the WCHB, it is also possible to use two CDs for every buffer, one for the input channel and another one for the output channel (see Fig. 6a). The C gates of the buffer are then only armed when there is actually data at the input. Consequently we refer to this buffer

[2]In an asymmetric C gate inputs labeled with "+" are considered for up-transitions of the output only. A "-" indicates that the input is relevant for the down transition only.

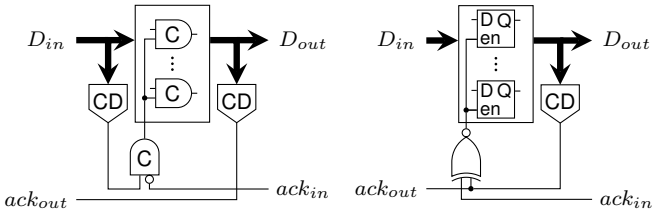


(a) Deadlocking WCHB (b) Interlocking WCHB

Fig. 5. WCHB storage element modifications proposed in [15]

as the Dual CD WCHB. This idea is presented in more detail in [14], where it is referred to as normally closed latch.

Another approach to avoid the lasting consequence of a faulty transition is to use different storage elements altogether (see Fig. 6b). For that purpose a D latch based Mousetrap-style pipeline structure as proposed in [16] can be used. Here a simple buffer control circuit, consisting of just a single XOR gate, is responsible to enable and disable the buffer's D latches. When the D latches are transparent, input glitches caused by SETs can freely propagate through the latch to the output. Unless the latch is closed in exactly this time instance, the latch will not store the faulty value. Similar to the previous approach the buffer is closed as soon as the CD detects the data phase. Although strictly speaking this circuit is not QDI because it introduces a small timing constraint, we still want to include it in our survey since it should show quite a different behavior in the analysis and will refer to it as MTDLatchHB (Mousetrap-style D Latch half buffer). Note, however that this buffer has not been proposed to improve fault-tolerance.



(a) Dual CD WCHB (b) Mousetrap-style buffer

Fig. 6. Alternative buffer styles

Finally, we also want to mention a completely different approach proposed by Jang and Martin in [7]. This is not really a design or buffer style on its own, rather a technique for hardening existing QDI designs. Here the original design is duplicated and both copies are interlocked with C gates that essentially vote on every intermediate signal of the circuit. The authors formally show that this scheme is able to tolerate (single) faults on any internal signal, which means that in our analysis it should not show any erroneous behavior. However, it is easy to see that this approach entails more than double the area overhead of the original design. Moreover, the additional logic for synchronization of the two replicas also results in a slightly slower circuit. Following the terminology used in [7] we refer to the WCHB using the described technique as a doubled-up double-checking (DD) WCHB.

C. Tooling

To obtain the necessary statistical coverage of the experiment space in our desired comparison of pipeline styles, and their dependence on certain parameters, we executed over 100M simulations on a total of 120 different target variations. To make this feasible we extensively rely on automation of parameterization, target generation, simulation and result extraction. Fig. 7 shows an overview of this process.

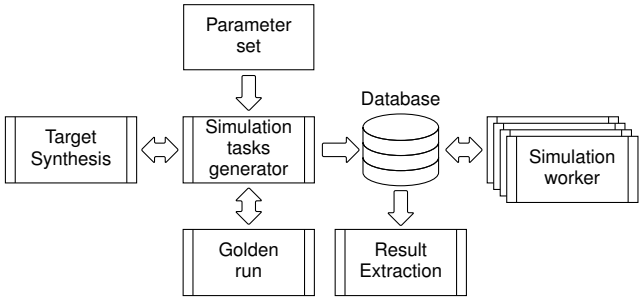


Fig. 7. Simulation setup

Everything is built around a central SQL database, that stores (i) the simulation tasks that need to executed and (ii) the results of those simulations. In a first step a parameter set has to be defined which is then issued to the simulation task generator, which generates the appropriate circuit (using our custom Python based PRS framework) as well as an accompanying testbench, and configures all the required simulation parameters.

Some of those parameters must be configured on a per-target and per-experiment basis. Consider for example the sink and source delays, which have to be adapted to bring the target in a given load scenario. Since those values depend on the actual target-circuit timing, they are determined using some preliminary simulations during circuit generation. Another example is the amount of simulations to be executed to get an adequate coverage with randomized injection time and targets. Furthermore, the injection window (confining the allowed injection time) is dependent on the target timing and needs to be adjusted on a per-target basis. During this process also the golden (i.e., fault-free) simulation run is performed.

After all the necessary data for an experiment (circuit, testbench, simulation parameters) has been generated the simulation task is divided into several reasonable-sized work packages that are added to the database. Those work packages can then be processed by multiple simulation workers (on multiple physical machines) in parallel. Simulation workers can be added and removed dynamically from our setup, which helps with restricting and balancing the computational load. Every active simulation worker periodically checks the database for open work packages. If one is present, it is claimed and the associated simulations are performed. To save space only results which deviate from the golden run are saved in the database.

To run the simulation we use a network of 10 physical machines (3.5 GHz 7$^{\text{th}}$ generation Intel i5 processor, 16 GB

RAM) each running 4 workers in parallel (one worker per core). The combined runtime of all simulations across all machines is approximately 1200 hours. The actual simulator used is QuestaSim (version 10.6c).

After all simulations are complete the final results can be extracted from the database using SQL queries. The information stored in the database also allows for each individual simulation to be rerun, such that unexpected behavior or interesting effects can be investigated.

V. Evaluation Results and Discussion

A. Effect classes

During a simulation run, we inject faults into all internal signals of the target circuit that are visible on the PRS level, i.e., we consider the PR atomic and do not resolve its internal implementation. The test bench monitors the primary outputs of the circuit only and records all deviations from the golden run. This choice was made to include the fault masking capability of the buffer styles in the results. The more masking the circuit provides, the fewer effects will propagate to and be observable at the primary outputs, thus reducing the effective set of signals sensitive to faults. We introduce the following classification of observed deviations from the golden run:

- *Timing Deviation:* A transition happened earlier or later than expected. The circuit being DI, this is not a fault, but rather an observation.
- *Value Fault:* A wrong data value was delivered to the output, but the circuit correctly adhered to the protocol.
- *Code Fault:* An invalid DI code word was observed at the output (i.e., both rails of a dual-rail bit high).
- *Glitch:* A signal changed its value twice during a protocol phase. This includes protocol violations (e.g., acknowledgment before data completion)
- *Deadlock:* The circuit reached a state where no further transitions were possible.
- *Token count error:* The number of tokens at the output channel did not match the golden run.

The test bench includes monitors to capture the occurrence of events falling into any of these classes. Note, that the effects of a single simulation run can fall into multiple categories, e.g., the circuit may produce a coding fault and then deadlock.

B. Parameters for comparison

Since the tooling we built to perform the experiments allowed us quick and automated changes to the target circuit with a seamless adaptation of the fault injection settings, one of our goals was to study what effects well controlled changes have on the resilience of a circuit and identify important design parameters. For this purpose we systematically varied the following design parameters during our analysis:

- Buffer style, see Section IV-B
- Implementation (pipelined vs. iterative), see Section IV-A
- Data width (4 bit, 8 bit)
- Operations per stage (pipelined version)
- Pipeline load factor

The *operations per stage* (OPS) parameter of the pipelined implementation dictates, how many stages of computational logic are placed between two pipeline buffers. The fully pipelined multiplier computing one partial product in each pipeline stage has 1 OPS. When set to two, every other buffer is removed with its input and output signals wired together, leaving logic for two partial product computations in each pipeline stage.

Both, OPS and data width settings allowed us to change the ratio of logic related gates to gates used to implement the buffers. While the size of the used adders increases with rising bit width, so does the width of data words stored in the buffers along with the CDs. By using the OPS parameter we were able to vary the amount of logic between pipeline stages keeping the implementation and width of the buffers unchanged.

The *pipeline load factor* is a metric that specifies whether the circuit is operated in a more bubble or token limited way. While having the nature of a measurement rather than a design parameter, it can be varied by changes of the average response time of the input and output channels in the test bench. Note however, that our simulation setup automation which determines the test bench speed to reach a certain pipeline load factor averages the pipeline load measurement over all buffers while 100 tokens pass through them, while the actual fault injection simulation is significantly shorter and the measured pipeline load factor of that shorter simulation time can differ from the desired setting. It is the averaged ratio of the time the individual buffers of a circuit spend waiting for the next data or null phase and of the time waiting for the acknowledgment. A well balanced pipeline should have load factor of 1 when operated at maximum speed; delaying the acknowledgments on the output channel will make the circuit bubble-limited thus increasing the load factor.

Furthermore, in the analysis, we differentiated between injection victims being *control and data signals*. Control signals, like the acknowledgment and latch enable signals, are responsible for value and spacer token migration through the circuit, which is not to be confused with the full control part of the iterative multiplier implementation, i.e., the upper portion of the circuit depicted in Fig. 2.

Of course, our tool conveniently allows adapting the relevant fault and delay parameters to a given technology and a given physically grounded fault model. For our more general study here, we performed preliminary experiments to assess the impact of the *width of the injected fault pulse* (relative to the circuit delays) on the observed effect classes. It turned out that pulses shorter than the gate delays were filtered by the inertial delay model we used for the gates (corresponding to electrical masking), while arbitrarily increasing the pulse width did not bring any new insight. Thus we only used a fixed width of 1.5 ns for the injected pulses that was slightly above the range of randomization we used for most of the gate delays.

The results will be presented with plots showing, for each effect class, the number of injections that provoked the respective observable effect, divided by the total number of injections into the considered set of signals (when differentiating between

control and data signals). In Fig. 8, we present results for all combinations of 4 and 8 bit data width, control and data signals as injection victims, with the 6 considered buffer styles. Each subplot shows the configured pipeline load factor on the x axis and the fault injection sensitivity on the y axis.

C. Effects of parameter changes

Fig. 8 clearly shows that the *pipeline load factor* is an important parameter with a significant influence on the resilience of the studied circuits. The control signals of all buffer styles have lower fault sensitivity in the token-limited operation, i.e., with a low pipeline load factor. Also for data signals, most buffers show better resilience in token-limited operation. The exception here is the MTDLatchHB buffer that has its data latches transparent while waiting for data and thus naturally provides less temporal fault masking in token-limited operation, which also increases the chance that glitches are propagated to the output.

Comparing the results when the *data width* is increased, it can be noticed that the sensitivity of the control signals decreases with the higher data width. This is due to the CDs signals being part of the control signal group. The higher data width requires larger CDs trees which are less sensitive to faults than for example acknowledgments and buffer enable signals. The increased number of CDs wires in the control signal group while keeping the number of other control signals mostly unchanged causes a relative decrease of overall control signal sensitivity. For the data signals, we observe an increase of value faults across all the buffer styles, as data width increases – which corresponds with intuition.

Fig. 9 shows the results for the 8 bit pipelined multiplier in 1 and 2 OPS configuration as well as the iterative multiplier, allowing the three designs to be compared. The pipeline load factor of the iterative multiplier is not practically controllable by varying handshake delays at the interface to the circuit because of its self-timed operation while computing all partial products in a loop. Thus, for the iterative implementation, the pipeline load factor was only measured, rather than controlled, and found to be 1.25 on average for the different buffer styles, ranging between 0.98 and 1.36. Given that we have seen the pipeline load factor having a significant influence on the results, for a fair comparison, when plotting the results for the pipelined multiplier, we only used data with the pipeline load factor fixed at 1.2, the closest value simulated to that of the iterative multiplier. Fig. 9a can thus be considered as a vertical cut through the 8 bit values from Fig. 8 at load factor 1.2, slightly right of the center, both for data signals (upwards of 0 on the y axis in the bar plot) and control signals (downwards from 0 in the bar graph).

We can see from Fig. 9 that increasing the *operations per stage* has no remarkable impact on the fault sensitivity of the circuit. The most significant difference for data signal is with the Interlocking buffer, however, in this case, the measured pipeline load factor during the fault injection simulation was 0.78 for the 1 OPS simulation while it was 1.28 for the 2 OPS simulation. Taking the effect of the pipeline load factor from Fig. 8 into consideration, the deteriorated performance of the Interlocking buffer can be explained by the imperfection of our automated test bench setup yielding a discrepancy between the targeted pipeline load factor of 1.2 and the actual pipeline load factor during fault injection.

For the iterative implementation we observe a better resilience. One notable difference lies in the reduced coding faults for the Deadlocking buffer, because the iterative nature of the circuit prevents the coding faults detected in the internal loop to be propagated to the output when a deadlock occurs.

D. Buffer style comparison

Compared to the WCHB, the masking provided by the Interlocking buffer successfully reduced the number of faults that can propagate through the buffer. While being seemingly worse, the deadlocking buffer in fact performs as expected and, by not allowing a spacer into a buffer once a coding fault has been detected, it turns coding faults into coding, deadlock and tokencount events logged by the test bench.

The DualCD buffer performs similar to the WCHB albeit showing less sensitivity for control signals. As discussed when looking at the increase of the data width, also here the added CDs increase the relative number of CD signals in the group of control signals thus making the overall result relatively better.

It can also be seen that the DD WCHB buffer (and logic) style successfully blocks the SET injections into the circuit from causing an observable effect other than a timing deviation from the golden run. The quality of the results for the DD WCHB style along with its very large size compared to the other styles is also the reason we did not simulate all circuit variations with this style, it is rather included as a sanity check for our tool and simulation flow. The DualCDWCHB performs very similar to the WCHB buffer, while the MTDLatchHB buffer is more resilient albeit propagating more glitches and being more sensitive on the control signals.

VI. Conclusion

We have performed extensive simulation based fault injection experiments into asynchronous pipelines with different pipeline styles to allow for a direct comparison of their resilience to transient faults. As our target circuit we have selected a multiplier circuit, since its complexity, while not being trivial, still allows understanding and tracing all its operation details and simulation with reasonable computational efforts. Our sophisticated fully automated setup allows performing many millions of fault injections that are still well controlled and reproducible in all detail for later analysis of interesting cases. By varying several parameters like data width, pipeline structure, or buffer style we were able to directly observe the differences in the effects that the same types of injected faults cause under these different conditions. This allowed a direct comparison of the sensitivity of different buffer styles, along with a first analysis of which types of effect are dominant.

Our setup will, with some minor refinements, allow us numerous further investigations like taking the effect analysis to the level of a single buffer, or quantifying the degree of

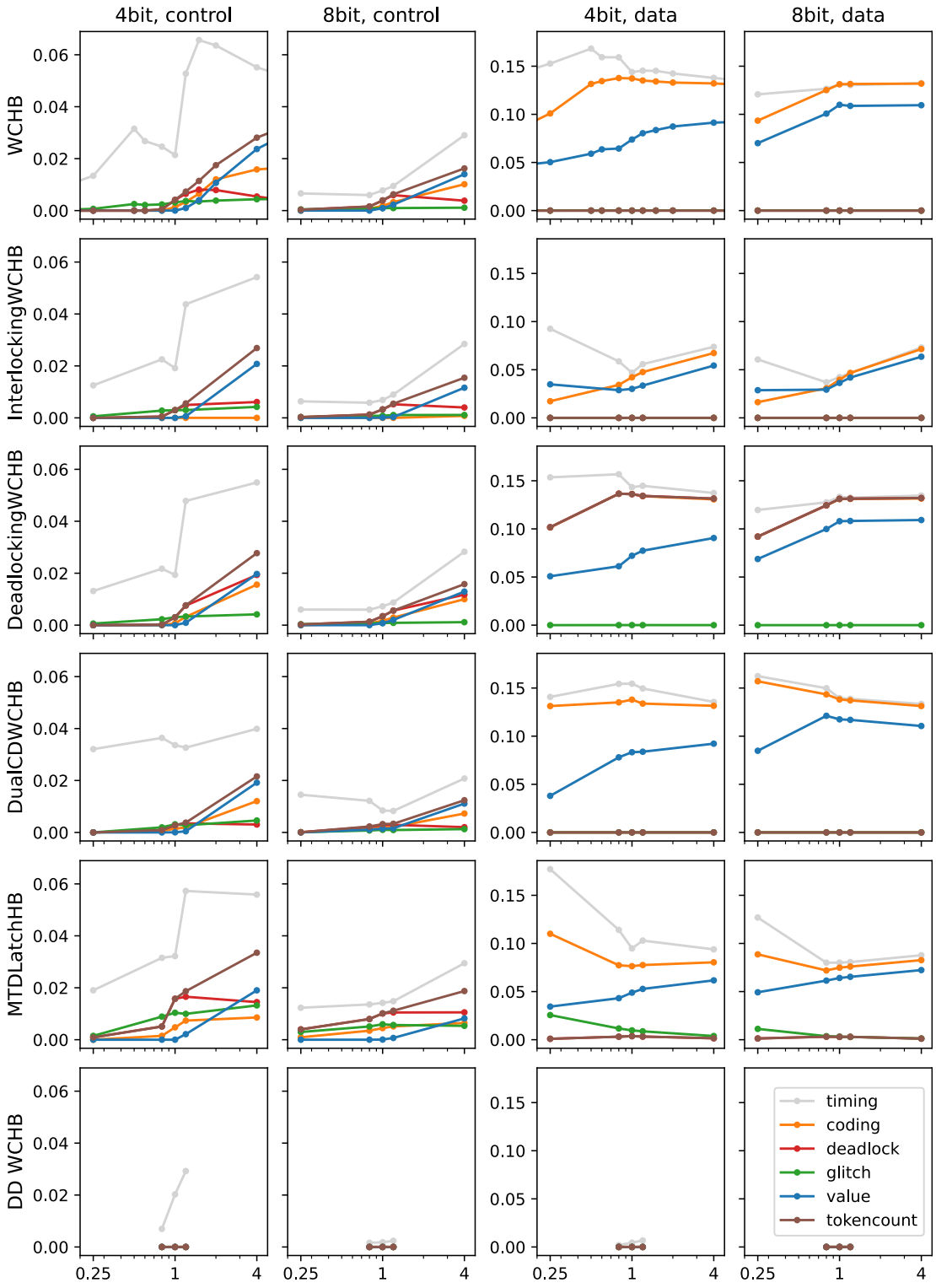
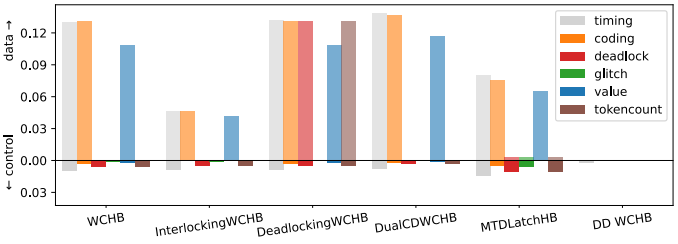
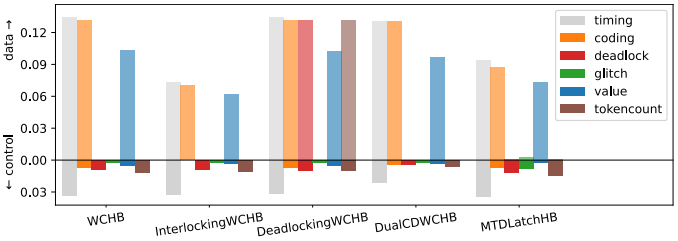


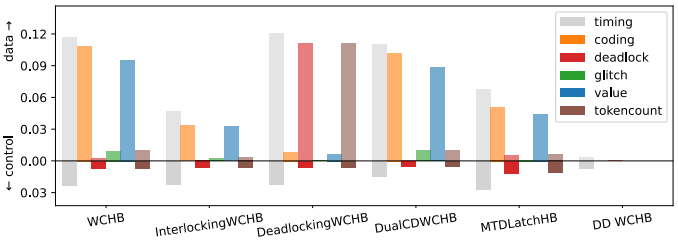
Fig. 8. Number of observed effects relative to the total number of injections over pipeline load factor



(a) Pipelined, 1 operation per stage



(b) Pipelined, 2 operations per stage



(c) Iterative

Fig. 9. Number of observed effects relative to the total number of injections for 8 bit multiplier designs

masking between pipeline stages. The vision is to understand the masking effects well enough to be able to make quantitative predictions for a given parameter set, which would be a valuable foundation for optimizations of the circuits' resilience.

In addition, once we have gained a better understanding of the impact of certain parameter choices, we can reduce the parameter space and use the available computational power to extend our list of target circuits towards more complex ones.

Abbreviations and Acronyms

- **CD** completion detector
- **DI** delay-insensitive
- **OPS** operations per stage
- **PRS** Production Rule Set
- **QDI** quasi delay-insensitive
- **SET** single event transient
- **WCHB** Weak-Conditioned Half Buffer

References

[1] R. Manohar and Y. Moses, "Asynchronous signalling processes," in *25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2019, pp. 68–75.

[2] R. P. Bastos, Y. Monnet, G. Sicard, F. Kastensmidt, M. Renaudin, and R. Reis, "Comparing transient-fault effects on synchronous and on asynchronous circuits," in *15th IEEE International On-Line Testing Symposium*, June 2009, pp. 29–34.

[3] T. Verdel and Y. Makris, "Duplication-based concurrent error detection in asynchronous circuits: shortcomings and remedies," in *Proceedings of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2002, pp. 345–353.

[4] F. A. Kuentzer and M. Krstic, "Soft Error Detection and Correction Architecture for Asynchronous Bundled Data Designs," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–12, 2020.

[5] C. LaFrieda and R. Manohar, "Fault detection and isolation techniques for quasi delay-insensitive circuits," in *International Conference on Dependable Systems and Networks, 2004*, June 2004, pp. 41–50.

[6] Y. Monnet, M. Renaudin, and R. Leveugle, "Hardening techniques against transient faults for asynchronous circuits," in *11th IEEE International On-Line Testing Symposium*, July 2005, pp. 129–134.

[7] W. Jang and A. J. Martin, "SEU-tolerant QDI circuits [quasi delay-insensitive asynchronous circuits]," in *11th IEEE International Symposium on Asynchronous Circuits and Systems*, March 2005, pp. 156–165.

[8] S. Keller, A. J. Martin, and C. Moore, "DD1: A QDI, Radiation-Hard-by-Design, Near-Threshold 18uW/MIPS Microcontroller in 40nm Bulk CMOS," in *21st IEEE International Symposium on Asynchronous Circuits and Systems*, May 2015, pp. 37–44.

[9] F. A. Kuentzer, M. Herrera, O. Schrape, P. A. Beerel, and M. Krstic, "Radiation Hardened Click Controllers for Soft Error Resilient Asynchronous Architectures," in *26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2020, pp. 78–85.

[10] M. Marshall and G. Russell, "A Low Power Information Redundant Concurrent Error Detecting Asynchronous Processor," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, Aug 2007, pp. 649–656.

[11] K. T. Gardiner, A. Yakovlev, and A. Bystrov, "A C-element Latch Scheme with Increased Transient Fault Tolerance for Asynchronous Circuits," in *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, July 2007, pp. 223–230.

[12] S. Peng and R. Manohar, "Efficient failure detection in pipelined asynchronous circuits," in *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*, 2005, pp. 484–493.

[13] Y. Monnet, M. Renaudin, and R. Leveugle, "Asynchronous circuits sensitivity to fault injection," in *10th IEEE International On-Line Testing Symposium*, July 2004, pp. 121–126.

[14] W. J. Bainbridge and S. J. Salisbury, "Glitch Sensitivity and Defense of Quasi Delay-Insensitive Network-on-Chip Links," in *15th IEEE Symposium on Asynchronous Circuits and Systems*, May 2009, pp. 35–44.

[15] F. Huemer, R. Najvirt, and A. Steininger, "Identification and confinement of fault sensitivity windows in qdi logic," in *2020 Austrochip Workshop on Microelectronics (Austrochip)*, Oct 2020, pp. 29–36.

[16] P. McGee, M. Agyekum, M. Mohamed, and S. Nowick, "A Level-Encoded Transition Signaling Protocol for High-Throughput Asynchronous Global Communication," in *14th IEEE International Symposium on Asynchronous Circuits and Systems*, 2008, pp. 116–127.

[17] J. Sparsø and J. Staunstrup, "Delay-insensitive multi-ring structures," *Integration, the VLSI Journal*, vol. 15, no. 3, pp. 313 – 340, 1993, special Issue on asynchronous systems.