Concurrent Error Detection in Self-Timed VLSI *

David A. Rennels and Hyeongil Kim
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024

Abstract

This paper examines architectural techniques for providing concurrent error detection in self-timed VLSI pipelines. Signal pairs from Differential Cascode Voltage Switch Logic are compared with a checker that is composed of a tree of dual-rail (morphic) comparators to detect errors and signal completion. An efficient implementation is shown that compares favorably in speed and area with conventional completion signal generators. A simple pipeline is examined with error checkers at each computation stage and handshaking control circuits that are modified to improve error detection. Its error-detecting properties are discussed, and preliminary error simulation results are presented. Based on these studies we have concluded that self-timed logic offers considerable fault-tolerance potential due to its built-in redundancy that can be effectively exploited for error checking.

Keywords: *self-checking, self-exercising, self-timed, concurrent error detection, VLSI circuits.*

1 Introduction

It is difficult to avoid noticing the similarity of the complementary pair coding used to detect completion in asynchronous Differential Cascode Voltage Switch Logic (DCVSL) and the complementary signalling used in the classic paper describing self-checking logic of Carter et. al. [Cart72]. It has been recognized by many that the redundancy inherent in self-timed logic can be used for testing and error detection. Fault modeling and simulation of DCVS circuits is discussed in [Barz85] and the DCVS circuits provide on-line testability with their complementary outputs [Mont85]. In [Jha89] testability of DCVS EX-OR gates and DCVS parity trees is analyzed. Methods for testing DCVS one-count generators, and adders are given in [Jha90]. In [Taka90] easily testable DCVS multipliers are presented in which all detectable stuck-at, stuck-on and stuck-open faults are detected with a small set of test vectors, and the impact of multiple faults on DCVS circuits is examined [Kano90]. A technique for designing self-checking circuits using DCVS logic was presented in [Kano92].

There has also been a great deal of interest lately in asynchronous logic design in the VLSI community [Mart89, Jaco90]. Self-timed logic is typically more complex than synchronous logic with similar functionality, but it offers potential advantages. These include higher speed, the avoidance of clock-skew in large chips, better layout topology, and the ability to function correctly with slow components.

Our primary motivation for this study is the need for lower power in many fault-tolerant systems. The current way of providing high-coverage concurrent error detection with conventional synchronous processors is to run two in lock-step and compare them. Synchronous processors consume relatively high power, much of which comes from clock distribution, and that power is doubled in a duplex configuration. Self-timed designs are expected to require considerably less power than two synchronous chips. A remarkable 16-bit asynchronous RISC processor has been fabricated by Martin and his students at Caltech that delivers 20 MIPS while only consuming 145 mW [Mart91a]. He points out that, contrary to popular belief, many of the asynchronous circuits in his design are only slightly larger in chip area than the corresponding synchronous circuits [Mart91b].

The designs we are studying consist of a set of self-timed combinational logic blocks made up of DCVSL circuits. The control and sequencing of these logic blocks is done with synchronizing circuits using Muller C-gates [Meng91, Mead80]. Our approach has been to modify existing designs of both the control and data sections to provide concurrent error detection. Although this logic is not self-checking in the formal

*This research was sponsored by the Office of Naval Research under grant (N00014-91-J-1009).

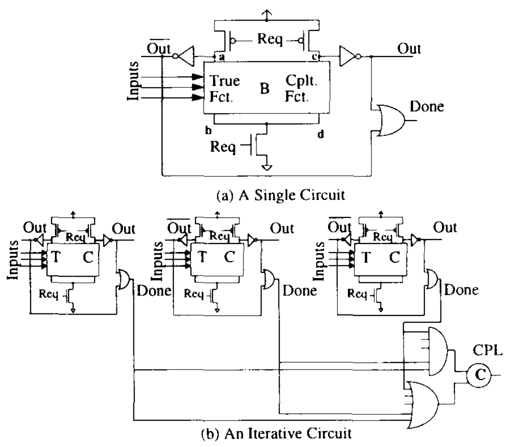

Figure 1: Differential Cascode Voltage Switch Logic (DCVSL)

sense, it may provide, for most purposes, the same capability of detecting faults in the checkers as well as faults in the circuits being checked. We have initiated a study of the effectiveness of error coverage through analysis and simulation. Experimental fault-insertion testing is currently being done on switch-level simulations of a two stage asynchronous circuit, and the preliminary results are highly encouraging. Before continuing, it is necessary to first present some technical background information.

1.1 Differential Cascode Voltage Switch Logic (DCVSL)

An asynchronous design requires redundant encoding that can provide completion information as part of the logic signals. A logic module is held in an initial condition until a completion signal arrives from a previous module indicating that its inputs are ready. Then it is started, and when its outputs indicate completion, other modules may be started in turn. This requires a form of encoding that allows the receiver to verify that the data is ready.

One way to do this in self-timed designs is to use 1-out-of-2 coding. Output signals from the logic modules are sent as a set of two-wire pairs, taking on the values 0,0 before starting, and 0,1 or 1,0 after completion. Such logic can be implemented using differential cascode voltage switch logic (DCVSL). A typical DCVSL circuit is shown in Figure 1.

DCVSL is a differential precharged form of logic. When the *Req* (request) signal is low, the PMOS pull-up transistors precharge points *a* and *c* to *Vdd*. At this time the circuit is in an initial state, and its outputs are 0,0. The circuit block **B** contains two complementary functions. One pulls down, and the other is an open circuit. When the input signals are ready, *Req* is raised, the pull ups are turned off, the NMOS pull down transistor connects points *b* and *d* to ground, and the circuit computes an output value. The side that forms a closed circuit discharges and the side that remains an open circuit stays precharged. The outputs, driven by inverters, go from 0,0 (the setup state) to either 1,0 or 0,1. A completion signal *CPL* from a set of DCVSL modules is shown in Figure 1(b). It is generated from the logical AND of the Done signals. For delay-insensitive implementations the completion signal is augmented by inputting it and the logical OR of the Done signals to a Muller C-gate (represented by a circle enclosing a C in the figure). A C-gate provides a form of hysteresis. Its output only goes to one when both inputs are one, and only goes to zero when both inputs are zero. This guarantees that the completion signal will only rise when the computation is finished, and it will only fall when the circuit is totally reset (see Figure 1(b)).

Fault-Effects on DCVSL Logic Elements It is easy to see the similarities between this logic and the complementary signalling in synchronous self-checking logic. DCVSL signalling redundancy can be used to provide error detection. Consider a single DCVSL circuit Figure 1(a). The circuit block **B** can be designed so that a single transistor fault or error will only affect one of the two sides (true or complement) and thus will only affect one output [Barz85]. The fault will cause a detectable output pair of 0,0 (detected by timeout because the circuit stalls due to no completion) or an illegal output pair of 1,1. Failure to precharge will eventually result in a 1,1 output while failure to pull down through the common pulldown transistor will result in 0,0 outputs. Precharging while evaluating or pulling down while precharging cause degraded signals that will either result in the correct output, 1,1 or 0,0 depending upon transistor sizing.

Fault Effects in Multi-Module DCVSL Circuits Given that faults in single-level DCVSL circuits produce (1,1 or 0,0) outputs, it is easy to show that a multi-level network behaves similarly. When a good circuit receives incorrect signal input pairs from a faulty circuit (i.e. 0,0 or 1,1), it will produce either the correct output or in an output of 0,0 or 1,1 because DCVSL circuits have paths corresponding to each minterm that either pull down one side or the other. A (0,0) input pair will disable some minterms

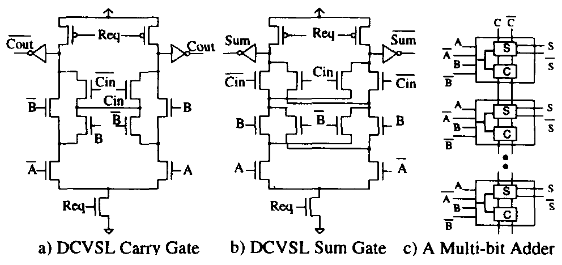

Figure 2: A DCVSL Computational Function Block

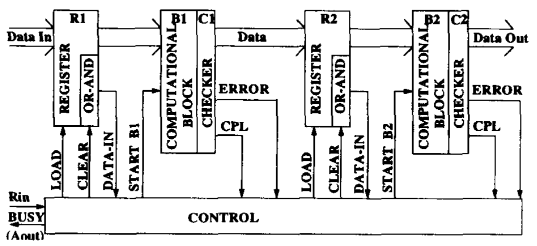

Figure 3: A 2-Stage Asynchronous Circuit (Micropipeline)

and can only prevent a side from being pulled down – producing an error output of 0,0. Similarly a (1,1) on an input pair will activate additional minterms and can only produce an error output of 1,1. These errors will be either masked or propagated through multiple level DCVSL and be detectable at the output.

1.2 A DCVSL Computational Network – An Adder Example

Figure 2 shows the DCVSL circuits used in a ripple-carry adder. The sum and carry circuits are shown in Figure 2(a) and Figure 2(b). The complete adder is shown as a multi-module circuit in Figure 2(c), and a 4-bit adder of this type is used as the function blocks in the circuit simulations to be described later.

2 An Asynchronous System with Concurrent Error Detection

We now show how these DCVSL function blocks are combined into a larger sequential system with concurrent error detection. Figure 3 shows a 2-stage micropipeline.

Each stage begins with a register made up of two gated latches for each complementary pair of input signals received from DCVSL circuits. The register data

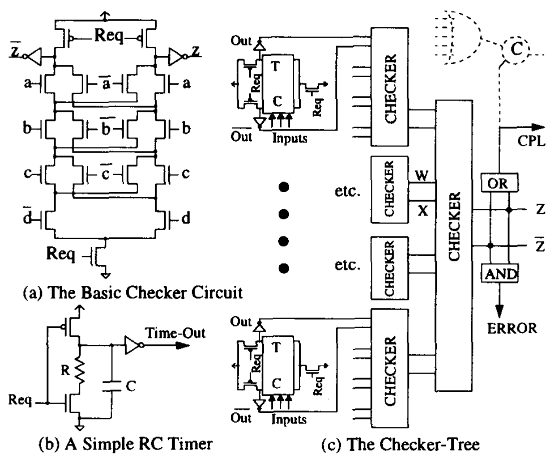

Figure 4: The Checker Circuit

is sent to a DCVSL computational block (a DCVSL 4-bit adder in our simulation studies). A checker is provided at the output of the computational block. The checker circuit is shown in Figure 4. It is logically equivalent to a tree of morphic AND gates used in synchronous self-checking checkers by Carter et. al. The outputs (z,$\bar{z}$) are complementary if all of the input pairs are complementary, and if any input pair is 0,0 or 1,1 the outputs (z,$\bar{z}$) take on that same value. This checker raises a completion signal (CPL = z OR $\bar{z}$) if all the complementary input pairs have at least one one, and raises an error signal (ERROR = z AND $\bar{z}$) if one or more input pairs has value 1,1. (An augmentation of the checker is shown in dotted lines where the logical OR of all the input signals and the intermediate checker tree signals is combined with the CPL output as inputs to a C-gate. This can be used in delay-insensitive applications to make the CPL signal fall only when the internal circuits are precharged and reset. It was not included in our simulated design because the handshaking signal delays provided ample time for resetting the computational block.)

If one of the input circuits fails to complete and generates a 0,0 signal pair, then (z,$\bar{z}$=0,0), a completion signal CPL never occurs, and the checker uses a simple time-out counter (see Figure 4(b)) to signal the error. Note that since a circuit with 0,0 locks up, the setting of the timer is not critical. Since most circuits should complete within less than a microsecond, we assume that the RC time constant would be set on the order of 100 microseconds – way outside the range of circuit variability.

If one of the computational circuits generates a 1,1 error pair the checker also generates a (z,z̃=1,1) output that is detected by (Error = z AND z̃) as shown. (There is sufficient time delay in resetting the circuit and checker that if a 1,1 is loaded into the next-stage latch following the checker, it will also get to the checker outputs (z,z̃=1,1) long enough to be detected. There is also a small possibility of Byzantine behavior where a 1,1, or 0,0 output can be sent to the next stage without being detected by the checker. But it is latched at the next stage, and a latched 0,0 or 1,1 in one of the input variables to the next stage has a very high probability of being detected at that stage.)

The checker is largely self-checking because it is logically a tree of Carter's self-checking morphic-and gates. For any stuck-at signals in the tree, there is a set of "good" inputs that will cause that internal signal pair to take on values 0,0 or 1,1. Any stuck-at-zero in the checker tree will result in a 0,0 output from the checker for some "good inputs," and result in a timeout error. Similarly a stuck-at-one anywhere in the checker tree will generate a 1,1 output for some "good inputs."

We found an interesting problem in using this type of checker to generate a completion signal in an asynchronous environment. Under some conditions a stuck-at-one in the tree can generate premature completion signals. If a member of a complementary pair inside the tree w,x (say w) is stuck at one and the input signals would normally generate w=1 and x=0, the tree can generate a premature completion signal without waiting for the circuits preceding w,x. This premature completion signal may cause a data word to be sent to the next stage of a pipeline before some of the circuits have finished setting up. It is not a problem because this leads to 0,0 input to the input register to the next stage. This will halt the next stage and be detectable by a timeout. (We have also not protected the final gates that generate the CPL and ERROR signals, since their failure is unlikely, and it usually leads to other detectable errors.)

The following is a simplified view of the control sequence (see Figure 3). Each computational block is initiated when its input register has data and the input register to the next stage is empty. Upon computing a result it loads the next stage's input register with its output, which in turn resets its input register and precharges the computational block to be ready to start the next computation cycle.

2.1 Control Circuits

A detailed view of the control circuits are shown in Figure 5. The circuits in Figure 5 can be viewed as multiple stages of logic where each stage consists of a register followed by a block of DCVSL computational logic. The stages can operate concurrently and form a pipeline. We started with an Interconnection and Synchronization Circuit (ISC) by Meng [Meng91] shown in Figure 5(a). The objective was to add whatever was necessary to provide concurrent error detection in the system. In the original design, the registers were implemented with edge-triggered flip flops whose inputs used only the "true" signal from each input pair. The Q and Q̃ flip flop outputs provided the complementary signals for the following DCVSL computational block. This had two problems. First, the one-out-of-two checking code was lost at the registers so a flip-flop error was undetectable. Second, loss of a clock to one or more flip-flops was also undetectable.

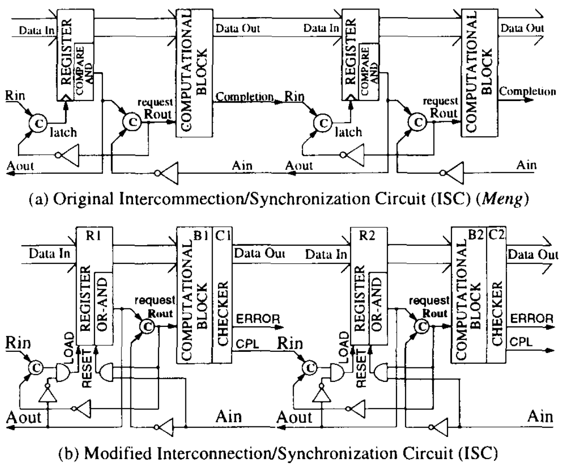

Figure 5: The Handshaking Control Circuits

The modified ISC circuit in Figure 5(b) uses essentially the same handshaking conventions as Meng. However, to improve the testability and fault tolerance of the control and synchronization circuit we modified it in the following fashion. First, each flip flop in the register was replaced by two gated latches, one for each signal of a DCVSL output pair, and thus it maintains the 1-out-of-2 coding for error detection. Second, all latches are reset after their contents have been used (to 0,0 pairs). This was done to detect stuck-at-zero clock faults that might leave old but properly coded complementary signals in the register. The dual gated latches are simpler than the single positive edge triggered flip-flops used in the original design.

The main synchronization signals are:

- **Rin** - a data-ready by the checker, signalling completion of a computational block.

- **Aout** - a register-loaded signal, indicates that every signal pair in a register has at least one "one". It is the logical AND of the OR of each latch pair of signals.
- **Rout** (or Request) - A signal releasing the pull ups and causing computation in the DCVSL computational logic. This signal is interlocked by the C-gate so that the logic cannot be started until the input register is loaded and the following register is free. Similarly it cannot be released to reset the computational block until data has been loaded into the following register and its input register is reset.
- **Ain** - the same as Aout, it is the busy signal from the next register.

The transition graph of signals in Figure 5 is shown in Figure 6. which shows a full-handshake between function blocks as explained in [Meng91]. This is the synchronizing function performed by the control circuits. Both the positive and negative values of the control signals are shown by the superscript + and −. Arrows show signal conditions that must be true before the following transition is allowed to proceed. A careful examination of the graph shows that this provides the appropriate interlocking so that a module on the right has to complete before the module on the left is allowed to take the next computational step. The $Rout^-$ to $Rout^+$ step then provides the reset to pull up the DCVSL functional block before the next computation is started.

It was pointed out by reviewers that the design that we simulated is not delay-insensitive, but depends upon the delay of the handshaking circuits being long enough to guarantee that registers and computational blocks will be fully reset before new computations are started. This works well in the circuits here because the pull-up (reset) process is much faster than the several gate delays that transpire between the reset of a circuit and the introduction of new data. To be fully delay insensitive and more robust, C-gates should be included in the checker and register completion signal circuits as described above, and the feedback signals from Rout can be taken instead from the CPL signal of the checker.

2.2 Stuck-at Faults

The interlocking nature of the feedback control signals causes the circuit to "hang up" and stop if one of the signals *Ain*, *Aout*, *Rin*, *Completion*, .. sticks at a one or zero value (see Figure 6). A time out counter is employed to detect the stopped condition.

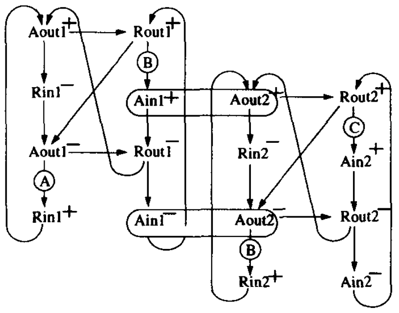

Figure 6: Transition Graph for a Full Handshake

In nearly all cases, stuck-at values in a register or computational block will cause a detectable value of 0,0 or 1,1 to appear at the checker. This occurs because the dual-rail DCVSL logic block circuits pass on an uncoded (0,0 or 1,1) output when an uncoded input (0,0 or 1,1) occurs. When input signals occur that would normally cause a stuck circuit to go to the other value, its complementary circuit takes on the same value, generating an uncoded signal that passes through the Computational Block to the checker.

The reset signal sets all register pairs to 0,0 to enable detection of faults caused by the inability to clock one or more sets of latches. If it sticks at zero, the register will not be cleared, making it impossible to load new data into the register, so the system will halt. If it sticks at one, the register will be permanently reset to 0,0 pairs, *Aout* will never go high, and the circuit will stop.

If the *load* signal sticks at zero, the registers will be permanently reset and *Aout* will not be generated, halting the circuit. A stuck at one *load* signal will cause the register not to be held constant while the computational block is working. The C-gate preceding a register normally prevents the register from being reloaded while the outputs of the circuit that sent it inputs is being to reset to 0,0. The stuck at one *load* will allow the register to change while the following computational block is using its data. The results, though difficult to predict, are likely to produce a detectable coding error in the following stage.

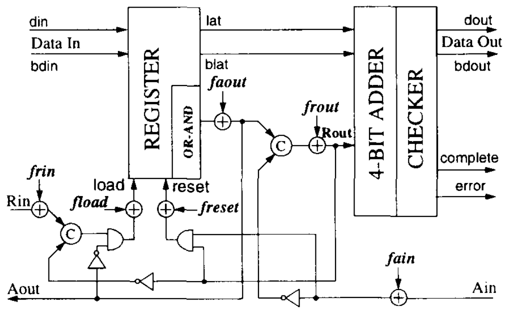

Figure 7: Experimental Fault Insertion Points

3 Practical Experience with Simulated Error Insertions

There is no easy way to analyze the effect of errors in an asynchronous circuit. To help in analyzing the response of the circuit to transient errors we simulated the 2-stage micropipeline circuit by injecting randomly generated faults at points (with labels beginning with f) shown in Figure 7.

The circuit was divided into two sections of control logic and data logic. Transient error insertion points in the control logic are shown as exclusive-or gates in the figure. Two types of errors were injected into the data path section: i) data latched in the register, and ii) data generated by the computational block (a 4-bit DCVSL adder). The simulation has been done using Lsim, which is Mentor Graphics Corp.'s mixed mode simulator. The simulation circuit is built in the netlist and in the modeling M language. In order to make randomly generated transient errors and data patterns for the circuit, an input deck generation program was written in C.

The simulation program operates in one nanosecond time steps, and normally each calculation cycle takes about 100 time steps. Individual simulation runs introduce randomly generated 4-bit numbers into the two stage circuit, and operate for 5000 steps. Approximately 20 errors are inserted in each run, where the timing and duration of the transient errors are determined from the random numbers and are in the range of 10 – 90 time steps. We assume that there is only one transient error at a time. Inputs to the adder circuit are randomly varied, and we can easily determine the expected time sequence of the output variables. The effects of errors on the values or the timing of the outputs can then be analyzed.

As currently implemented, outputs are a series of waveforms for the simulated period. To partially reduce the volume of data to analyze, the output is inhibited during the times no errors are inserted, and values of registers and function block outputs are displayed in a hexadecimal form at steps when they are not changing. The output is converted to a text file that can be automatically analyzed to find cases where an error caused a time-out halt. This is detected if the circuit is idle for 200 time steps. The current set of simulations have required about 75 hours on a SPARC IPC.

Most of the rest of the analysis has been done manually, therefore the number of fault-insertions is relatively small. We are currently working on a more automated way of analyzing data, and we hope to do more extensive testing in the future. However, the results are highly encouraging, with no undetected errors in the faults simulated so far.

Since we are trying to determine the fault detection coverage of this circuit, a successful test occurs if: i) a fault produces a correct output though it may delay the circuit less than the time-out count, or ii) if a fault produces a bad output that is detected. An unsuccessful test occurs if an undetected error is found. The following are the results of the simulations so far. These results are divided into several categories:

- *no effect*
- *tolerated-delayed* – no errors, the circuit halted a short time until the fault was removed.
- *tolerated* – an error occurred in an internal variable but it was not used by the following stage (e.g., data already taken).
- *error detected* – explicitly detected by a timeout or error signal from the checker.

3.1 Error Simulation Results

Transient errors were simulated in the data section as described below:

Simulation of "0" transient errors in the data path. There are 32 data bits in the 2-stage simulated circuit. Each stage has 4-bits of latched data, 4-bits of complement latched data, 4-bits of output data (from the computational block) and 4-bit complement output data. In the simulation one of 32 data bits is randomly selected and it is temporarily stuck at 0 at random times with random duration less than 90 time steps. There was no undetected error in the simulation and effects of the transient errors are classified as:

a. no effect : 515 cases
b. tolerated (delayed) : 158 cases

c. tolerated : 102 cases
d. timeout error detected : 22 cases
Total = 797

Simulation of logic "1" transient errors in the data path. This simulation setup is almost the same as the transient error 0, but this time the selected data bit is temporarily stuck at 1. There was no undetected errors in the simulation and the transient errors are classified as:

a. no effect: 285
b. tolerated (ignored): 285
c. tolerated: 136
d. checker detected error: 254
Total = 960

Control Section Errors. This was a simulation of transient errors in selected control signals. Duration of each transient error was selected randomly between 10 to 30 time steps. There were no undetected errors.

a. no effect: 58
b. tolerated: 592
c. timeout error detected: 60
d. checker detected error: 29
Total = 739

3.2 Observations on Error Effects

After examining in detail the traces of logic signals in the error simulations, we have made the following observations about the effects of the fault-categories so far modeled. There were many cases where the inserted transient held a signal in its correct value, so there was no error. Those cases are uninteresting. So we will look at the cases where a real error occurred.

Transient Errors Making Data Bits 0

Errors in Registers/Latches

- If a "zero" transient error occurs in data being latched to a register, the OR-AND circuit delays issuing a latch completion signal (Aout) until the transient error disappears.
- If the error occurs after the latch completion signal (Aout) is generated, i.e., the following computational block starts, but there is one 00 pair in its input data, and it times out.
- If a latched data bit is affected after the following computational block generates a correct output and a completion signal, then the computational block produced a correct output even though one of input data bits later changed to 0, and the error has no effect.

Computational Block Errors

- If one of the output bits of a computational block is pulled to zero before the completion signal is generated, then the generation of the completion signal is delayed until the transient error disappears. This error is tolerated and causes only small time delay in the circuit.
- If a transient error occurs after a correct output has been generated by the computational block but before the data is latched in the register of next stage, then the latch completion signal (Aout) of the next stage is not generated until the transient error disappears, and correct output is generated and latched into the register. If the transient error exists for long enough time to activate the timeout circuit, then a timeout signal is generated. So we can say this error is tolerated or detected.
- If the error happens after output data is latched in the register of the next stage and before the computational block is

  initialized (evaluation signal is high), then the error causes no effect to the circuit. This error is also tolerated by the circuit.

As is explained above, it appears that all the transient errors that make a bit in the data path circuits go to 0 can be tolerated or detected by the timeout circuit. Note that in the simulations, all "zero" transients in the data path either slow the circuit or stop it and are detected by the timeout counter. There were no (1,1) pairs generated so there were no errors detected by the checker tree. The opposite occurs with "one" transients as is seen below.

Transient Errors Making Data Bits One

Errors in Register Latches

- If a data bit in the register is flipped to "one" during initialization, it cannot cause Aout to be asserted since the other register pairs are zeros. Therefore the error will be overwritten when the register is loaded and there will be no effect.
- When a data bit in the register is hit by the error after initialization of the register, the affect

will vary depending on incoming data and the time the error occurs. This error will have no effect (if the bit of incoming data is 1) or it can cause the affect of a 1,1 input pair to the computational block, leading to an incorrect output (1,1) from the computational block and (1,1) from the checker which signals an error.

Errors in the Computational Blocks

- If the computational block is hit by the transient error during precharge state, the affected bit is restored to 0 by the precharge, and the error is tolerated.
- If the data bit of the computational block is affected by the error during evaluation, then the output of the computational block will take on (1,1) and be detected by the checker.
- If the data bit of the computational block is hit before the precharge state and after output data is latched in the following stage, then the checker generates an error signal even though the error was not passed on to the next stage.

As it is shown above, all the logic "1" transient errors inserted so far have been tolerated – either causing no error (other than short delays) or being detected by a checker. Notice that these caused no timeout errors, as would be expected since they generate (1,1) signal pairs.

Transient Errors in the Control Signals

We have not yet done an exhaustive analysis of the effects of errors on all of the control signals, but we have looked at the ones on which errors were inserted. They are briefly (and informally) described below:

1. *Transient errors in the Rin signal* are tolerated due to the characteristics of the C-gate and the AND gate generating the load signal. If there is a transient error in the Rin signal when the request signal (Rout signal) is high, then the output of the C-gate cannot change. If it occurs when Rout signal is low, this means that the next stage is awaiting data. The output of the C-gate goes high and the load signal is generated prematurely. Here, the register will wait until coded data arrives before generating an Aout signal and starting the next circuit. Thus the error is masked. If the error happens when the Aout signal is high, the transient error in Rin signal is masked by the AND gate. If Rin goes to zero prematurely, the C gate does not allow the latch signal to drop until the register is loaded (when Aout is asserted). If Rin is either held to 0 or 1 for a long period of time, the circuit simply stops computing until the transient goes away.

   We are finding many cases where the the circuit simply stops when an error occurs. If the error goes away, the computations continue without error. If it lasts too long, a time-out error is generated.

2. *Transient Errors on the Load Signal* During error-free operation, this signal should be generated when correct morphic input data is available at the input port of the register and the following computational block is reset. At this time, the register has been previously reset to all zero pairs. We find that if it is raised prematurely (while the following computational block is reset and waiting for data) the circuit simply waits until the correctly coded data arrives because Aout will not be asserted.

   If an error causes load to be raised while the following computational block is evaluating, a there is a high probability that a detectable coding error will be created in the computational block. (A changing register value will be input to the evaluating DCVSL logic and cause both sides of some of the circuits to be pulled down, generating a 1,1 output.) If the load signal goes low because of a transient error before correct data is latched to the register, then the generation of Aout signal is delayed until load signal goes back to high and correct data is latched.

3. *Data Register Reset* The data register of the circuit is reset after the use of the data in the register and before the new data is latched. If a faulty reset signal is applied after it has been reset and before new data arrives, then there is no effect on the circuit. If the register has data arriving but the completion signal has not been set and the reset signal goes high because of a transient error, then the circuit will wait for the reset transient to go away and for the data to arrive. If it takes too long, a timeout will occur. If a reset occurs after the computational block has started, the computational block can not generate a correct output. In that case a timeout error is detected. Thus we can say that the transient error on the reset signal is tolerated or detected by the circuit. If the reset signal fails to go to "one" due to a transient, the Aout signal fails to be reset and this is also detected because the circuit times out.

4. *Transients in Aout* The Aout signal of the current stage is the Ain signal of the previous stage. A logic "one" transient error on the Aout signal can cause an early start of a following computational block by prematurely raising its Rout signal when the computational block has already evaluated previous data and precharged the functional circuit. In this case the computational block does not generate morphic outputs until correct input data comes from the register. Thus the circuit waits until correct data arrives or times out.

   If there is a "one" transient error on the Aout signal when the Rout signal of a previous stage is high, and the output of the functional block of the previous stage has not generated yet, then the register of the previous stage is reset forcing the computational block inputs to zero, and eventually a timeout error occurs.

5. *Transients on Rout* The Rout signal starts evaluation of the computational block. If a "one" transient error affects the Rout signal when the Rout signal should be low, then the error on the Rout signal does not have any effect on the circuit (the input register is inputting 0,0 so the circuit will remain precharged). If the Rout signal is high when a "zero" transient error hits the signal, then the effect of the error depends on the timing of the Rout signal. If the Rout signal is just beginning to go to one, then some computational block outputs will be 0,0. No completion will be generated and the circuit waits for the transient to go away. But if the Rout signal is hit by a transient error at the end of evaluation, then evaluation starts again and never finishes. In this case timeout error is detected.

At least in the cases of the transient errors in the control signals that we have studied so far, they are tolerated – producing delays or error signals.

4 Conclusions

Having done experimental fault insertions on both the JPL-STAR and Fault-Tolerant Building Block Computers [Renn72], the author certainly understands that inserting a few thousand errors does not adequately determine coverage for any system. But the fact that we have found no undetected errors so far tends to indicate that the basic design approach is sound. We will not be surprised to find (as occurred in the STAR machine) a few signals that are not adequately covered and have to modify the design to improve their coverage.

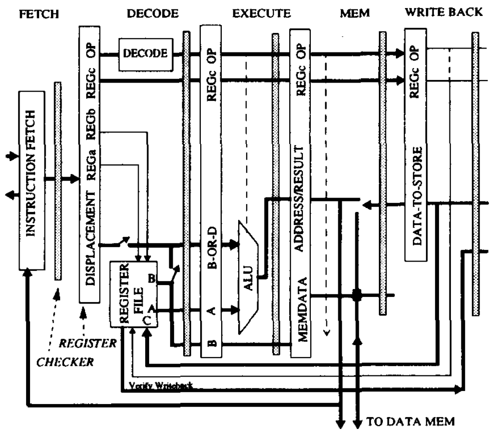

Figure 8: A Conceptual Error Detecting Asynchronous Processor Design

This study indicates that self-timed design techniques can be adapted to fault-tolerant systems, and that they offer considerable potential in the implementation of modules that have concurrent error detection. There are many interesting problems still unexplored. First, more extensive simulation experiments and analysis are needed to prove the effectiveness of these design techniques. Another interesting problem is to compare the power consumption in Watts per MIP of duplicated synchronous processors vs. self-checking self-timed designs. Another intriguing question is the possibility of implementing error recovery in the form of microrollback in micropipelines. By latching old values at each stage, it may be possible to restart and correct computations when an error has occurred. But these interesting problems must be left for subsequent investigations.

We close by suggesting a conceptual design approach for using this type of fault tolerant pipeline in a modern processor similar to Hennessy and Patterson's DLX design as shown in Figure 8 [Henn90]. Here we have a five-stage pipeline. The pipeline registers carry op-codes (in progressively decoded forms), register fields and the displacement as far as they are needed. At each stage, the input registers contain the control information to start the associated self-timed computational block (register reads in Decode, ALU in Execute, the data cache in MEM, and the register write back in Writeback). Control information and outputs to be sent to the next stage are checked and sent to the output register (i.e., the input register of the next stage). An independent set of four buses connect the computational components with two wires for each signal pair. Three state selection is shown in simplified form as switches.

Such a processor design (which we have not attempted to refine) would be very complex - especially when issues like forwarding, register write-back, pipeline stalling, and exception handling are taken into account. But it appears to fit the type of fault-tolerant design model we have described.

Most of the techniques described in this paper have been around for a very long time, and we claim no new theoretical results. What we have tried to do is to combine these techniques to create and analyze an example that demonstrates the potential of using DCVSL self-timed designs to provide concurrent error detection in low-power fault-tolerant systems - and which hopefully will help lay the groundwork for using it. We believe it to be an area of great potential.

References

[Barz85] Z. Barzilai, V. S. Iyengar, B. K. Rosen, and G. M. Silberman. Accurate Fault Modeling and Efficient Simulation of Differential CVS Circuits. In *International Test Conference*, pages 722–729, Philadelphia, PA, Nov 1985.

[Cart68] W. C. Carter and P. R. Schneider. Design of Dynamically Checked Computers. In *Proc. IFIP Congress 68*, pages 878–883, Edinburgh, Scotland, Aug 1968.

[Cart72] W. C. Carter, A. B. Wadia, and D. C. Jessep Jr. Computer Error Control by Testable Morphic Boolean Functions - A Way of Removing Hardcore. In *1972 Int. Symp. Fault-Tolerant Computing*, pages 154–159, Newton, Massachusetts, June 1972.

[Henn90] John Hennessy and David Patterson. Computer Architecture A Quantitative Approach. Morgan Kaufman, San Mateo, CA, 1990.

[Jaco90] Gordon M. Jacobs and Robert W. Broderson. A Fully Asynchronous Digital Signal Processor Using Self-timed Circuits. *IEEE Journal of Solid-State Circuits*, 25(6):1526–1537, Dec 1990.

[Jha89] Niraj K. Jha. Fault Detection in CVS Parity Trees: Application to SSC CVS Parity and Two-Rail Checkers. In *Proc. 19th Int. Symp. Fault-Tolerant Computing*, pages 407–414, Chicago, IL, June 1989.

[Jha90] Niraj K. Jha. Testing of Differential Cascode Voltage Switch One-Count Generators. *IEEE Journal of Solid-State Circuits*, 25(1):246–253, Feb 1990.

[Kano92] N. Kanopoulos, Dimitris Pantzartzis, and Frederick R. Bartram. Design of Self-Checking Circuits Using DCVS Logic: A Case Study. *IEEE Transactions on Computers*, 41(7):891–896, July 1992.

[Kano90] N. Kanopoulos and N. Vasanthavada. Testing of Differntial Cascode Voltage Switch (DCVS) Circuits. *IEEE Journal of Solid-State Circuits*, 25(3):806–813, June 1990.

[Mart89] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The Design of an Asynchronous Microprocessor. Technical Report Caltech-CS-TR-89-2, CSD, Caltech, 1989.

[Mart91a] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. Technical Report Caltech-CS-TR-93-28, CSD, Caltech 1991.

[Mart91b] Alain J. Martin. Asynchronous Datapaths and the Design of an Asynchronous Adder. Technical Report Caltech-CS-TR-91-08.

[Mead80] Carver Mead and Lynn Conway. Introduction to VLSI Systems. Addison-Wesley, Reading MA, 1980.

[Meng91] Teresa H. Meng. *Synchronization Design for Digital Systems.* Kluwer Academic Publishers, 1991.

[Mont85] R. K. Montoye. Testing Scheme for Differntial Cascode Voltage Switch Circuits. *IBM Technical Disclosure Bulletin*, 27(10B):6148–6152, Mar 1985.

[Renn72] A. Avizienis and D. Rennels. Fault-Tolerance Experiments with the JPL-STAR Computer. In *Dig. of the 6th Annual IEEE Computer Society Int. Conf. (COMPCON)*, San Francisco, 1972, pp. 321-324.

[Renn91] David A. Rennels and Hyeongil Kim. VLSI Implementation of A Self-Checking Self-Exercising Memory System. In *Proc. 21st Int. Symp. Fault-Tolerant Computing*, pages 170–177, Montreal, Canada, June 1991.

[Sedm80] Richard M. Sedmak and Harris L. Liebergot. Fault Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration. *IEEE Transactions on Computers*, 29(6):492–500, June 1980.

[Taka90] Andres R. Takach and Niraj K. Jha. Easily Testable DCVS Multiplier. In *IEEE International Symposium on Circuits and Systems*, pages 2732–2735, New Orleans, LA., June 1990.