Computing and Informatics, Vol. 23, 2004, 517–535

A GENERIC DUAL CORE ARCHITECTURE WITH ERROR CONTAINMENT

Thomas Kottke

Robert Bosch GmbH
Robert-Bosch-Strasse 2
717 01 Schwieberdingen, Germany
e-mail: thomas.kottke@de.bosch.com

Andreas Steininger

Technische Universität Wien
Treitlstrasse 1-3
1 04 0 Wien, Austria
e-mail: steininger@ecs.tuwien.ac.at

Manuscript received 9 January 2005

Abstract. The dual core strategy allows to construct a fail-silent processor from two instances (master/checker) of any arbitrary standard processor. Its main drawbacks are its vulnerability with respect to common mode failures and the existence of residual single points of failure. In this paper we propose a generic frame that systematically eliminates these drawbacks. First, we employ temporal redundancy to cope with common mode failures. Unlike similar approaches we can ensure error containment even if – as a result of the temporal redundancy – the comparison by the checker core is delayed. We attain this by introducing a specific delay element for outgoing data. Second, we perform a systematic analysis of potential single points of failure and eliminate these by careful layout, self-checking circuits and similar methods. We finally validate our approach by means of exhaustive fault injection experiments. The results indicate a 100% self-checking coverage for stuck-at faults and complete error containment. Since the proposed framework has been kept generic in the sense that the individual standard processor cores are treated as black boxes, these results are valid independent of the core actually used.

Keywords: Dual core, error detection, self-checking checker, common mode failure, fail-silent processor

1 INTRODUCTION

In future automobiles more and more microprocessor-based control systems for safety-critical applications such as antilocking brake system, electronic stability program or x-by-wire systems (for example distributed notes connected by CAN, TTP/C, TTCAN or FlexRay) will be implemented. To meet the high safety requests of future applications – as demanded in the European standard EN 61508, for instance – in spite of the increasing rate of transient errors [1, 2] that results from shrinking feature size [3] the microprocessors have to be equipped with powerful mechanisms for error detection and error handling. Generally the intention is to make the microprocessor fail silent and to enable a recovery from transient faults. While it is relatively easy to protect memory or communication interfaces by means of coding techniques, e.g., the core is more difficult to protect. One attractive generic solution in this context is a dual core system.

The advantage of a dual core system is that it is quite easily implemented with standard cores. In contrast to other error detection mechanisms for a processor – such as, e.g., code prediction [4, 5, 6, 7] or redundant computation (optionally with modified operands [8, 9]), – no change in the core itself is necessary. At the same time a very high error detection coverage can be attained. However, since the two processors are implemented on the same die, are operating in lock step, have the same power supply, the same clock generator and are working with the same input data, there is a potential for common-mode failures and single points of failure. This constitutes the main drawback of the dual core approach. Several dual-core solutions have already been proposed, but all of them had some disadvantages. For example, the master-checker approach from [10] requires two processor chips and has hence proven too expensive because of the board space and assembly costs for two chips, while in [11] single points of failure still remain, such as, for example, the error signal. In our approach two cores are located on the same die to yield a fast and economically viable solution. By means of a careful analysis we systematically eliminate all potential single points of failure. In addition to a high error detection coverage we are striving for short error detection latency. In particular we want all errors within our dual-core processor to be detected before they propagate to the output and hence pollute other functional units such as memory or peripherals. This error containment is essential for fast recovery.

Since the automotive market is highly competitive the performance of the processor must closely fit to the application requirements. Therefore a wide range of processors with different performance is employed. It is, however, very expensive to repeat the safety verification for every single core used. Our approach here is to perform a safety verification for the proposed generic dual core frame once, and

show that this verification is valid independent of the actual type of core embedded in our frame.

This paper starts with an introduction of the proposed dual core frame along with its implementation in Chapter 2. In Chapter 3 error handling within the proposed concept will be explained. Our approach for ensuring error containment will be presented in Chapter 4. Chapter 5 is concerned with an analysis of single points of failure in the system. Subsequently the setup of the fault injection experiments will be sketched in Chapter 6, while Chapter 7 presents and discusses the experimental results. Finally, chapter 8 concludes the paper.

2 IMPLEMENTATION OF THE DUAL CORE FRAME

2.1 Overview

The proposed framework is based on the assumption of a Harvard architecture for the processor cores used. Knowing that most processors practically used in automotive applications exhibit a Harvard architecture this is a reasonable restriction. The concrete processor we use to test our framework and to demonstrate its functionality is called SPEAR [12, 13]. This processor was selected because it provides all features found with state of the art processors. It is a RISC processor with Harvard architecture, has memory mapped IO and is able to meet the temporal demands of real-time systems. Although today automotive processors normally have 32 bit bus width, we considered the 16 bit architecture of SPEAR satisfactory for our study of the specific implementation problems of dual core systems. While this choice does not change the fault tolerance properties we intended to investigate, the resulting reduction in hardware complexity allowed us to implement the system in a FPGA and furthermore yielded shorter simulation time for the fault injection experiments.

Figure 1 shows the implementation of the dual core system. The components of the frame are transparent while the embedded standard processors are grey shaded and the additional standard components (memories) are grey hatched. Notice that we do not duplicate the expensive memory components but employ non-redundant fail-silent memories as proposed in [14] instead.

All inputs of the two processor cores are checked for correctness, and their outputs are compared. Obviously this is not true for power supply and clock. In order to facilitate the detection of common mode failures induced for example over the power supply or by electromagnetic interference (as shown in [15]) in general, the two cores are operating in time diversity: the master is directly controlling the peripherals such as the memory. The slave is working with a delay of exactly 1.5 clock cycles and does not perform write access to any resources. In order to obtain the delay, the slave must be driven with the inverted clock. Temporal alignment between master and slave is achieved by delaying the master's inputs by 1.5 clock cycles before providing them to the slave. The master's outputs have to be delayed by 1.5 clock cycles before they are compared with the output of the slave. Since

a mismatch between master and slave cannot be detected before the slave's results are available, the master may apply its erroneous output signals to peripherals for 1.5 clock cycles. Given that the typical fault tolerance time of automotive applications is much higher than several clock cycles, this does not constitute a substantial problem, but makes a software recovery after transient faults more difficult. However, by delaying the master's output until the result of the comparison is known, incorrect output signals can as well be completely prevented. In our implementation the price is a 2 clock cycle increase of the memory write access time.

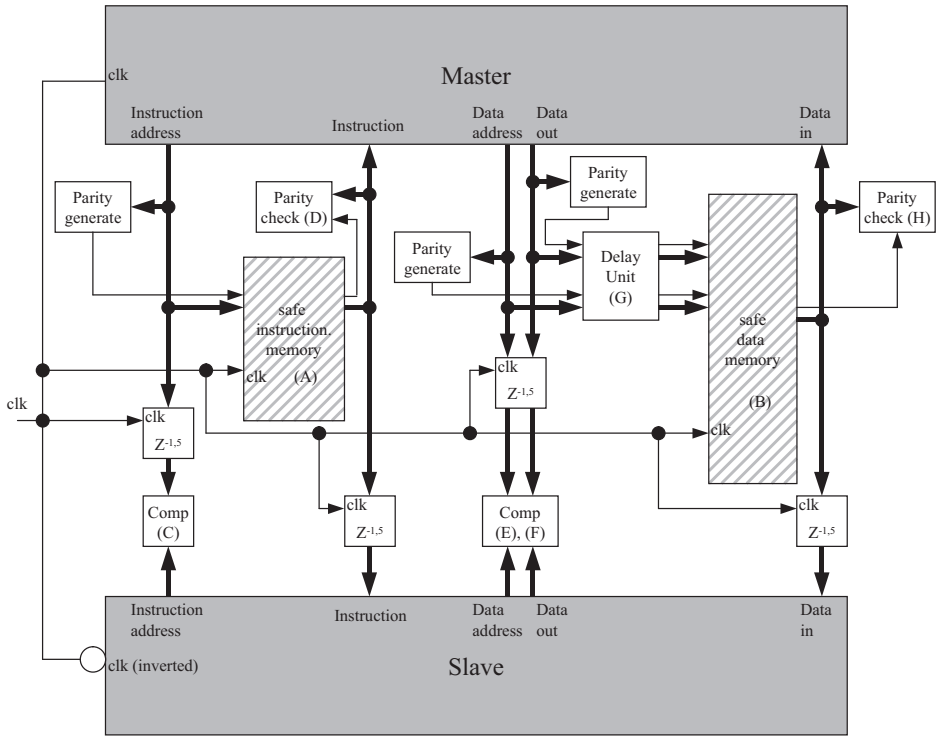


Fig. 1. Dual core with time diversity

2.2 Delay Components

The delay components $z^{-1.5}$ (Fig. 2) are built of a series-connection of two flip-flops each. The first flip-flop uses the same clock as the master, while the second is provided with an inverted clock. This component is used to delay the master's outputs before the comparison with the slave's outputs. These outputs are, for example, address and data for memory and peripherals, and control signals such as "write enable". Furthermore, all status-signals (like "access violation") that shall

be compared have to be delayed as well. Similarly, the slave's input signals (such as the incoming instructions, e.g.) have to be delayed.

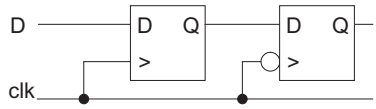


Fig. 2. Delay component

In an FPGA design it is often not possible to invert the clock-input of a flip flop; thus, two clock networks have to be used. In order to achieve a good timing behavior of the design the first flip flop has to be triggered with the clock of the source and the second flip flop with the clock of the receiver. Because all receivers like the comparators or the slave are working with the inverted clock, the first flip flop is triggered with the normal clock and the second with the inverted clock.

2.3 Comparators

The comparison of the output signals of master and slave is done by totally self-checking comparators as proposed in [16, 17]. If the comparison of the output signals is time critical, an approach for the comparator as shown in [18] can be used. All signals on the output buses are compared: data-lines, address-lines and the incoming and outgoing control signals for the external modules. Occasionally it may be desirable to compare some core-internal signals as well. This can reduce the error detection latency and allows an easier fault recovery. Although such an extension is quite straightforward to implement, we do not further consider it, since our initial approach was to treat the used cores as black boxes.

2.4 Bus

Like the comparators the buses for the incoming and outgoing data and instructions are non-redundant components as well, and therefore they require specific protection. We decided to use the parity bit for this purpose. In order to ensure that at least one core is getting correct data – otherwise the bus would constitute a single point of failure –, the buses have to be routed to the slave first and then to the master with the parity checker as shown in Figure 3 a). Should an error be located on the bus beyond the parity checker, it can be assumed that the slave is getting correct data and the comparators will detect the failure. With this layout rule fail-silent behavior can be achieved with respect to the incoming buses. In order to provide an equally efficient protection for the outgoing buses, the parity is first generated for the master's data and then the bus is routed to the delay component before the comparison with the slave's output is performed. The respective circuit diagram is shown in Figure 3 b). With the single fault assumption there is either a failure on

the bus or in the parity generator, which in both cases produces a detectable error (see below).

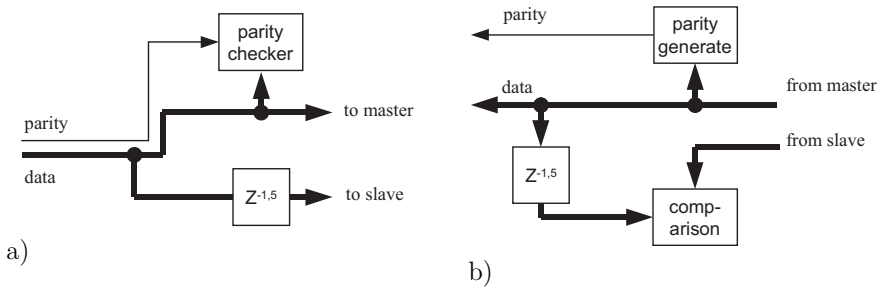


Fig. 3. Routing of the buses a) for incoming and b) for outgoing data

2.5 Parity Generator

Parity generation is performed for the master's data, so the parity can be transmitted to the destination simultaneously with the data, i.e. without delay. The data path is protected by the comparison; any bus fault can be detected by a comparator mismatch. In case of a comparator match the proposed layout ensures that the parity has been generated over correct data. Should the parity generator fail or the parity signal be corrupted, a parity check at the destination will detect this. For the single fault assumption we need not anticipate both these scenarios, hence the parity generator does not require any protection.

2.6 Parity Checker

To ensure complete protection of the master's input bus the parity check is performed directly at the master's inputs, i.e. after the fork of the bus to the delay component that feeds the slave. Since there is no way of detecting a stuck-at-inactive behavior of the parity checker, this unit requires a totally self-checking implementation. For the same reason the output signal of the parity checker is coded as an alternating dual-rail signal (see below).

2.7 Interrupts and Reset

To protect single-bit input signals to the dual core system, such as interrupts and the reset, e.g., dual-rail coding is employed. After having passed through an input synchronization stage, the non-inverted rail of such a signal is routed to the master, while the inverted rail goes to the slave. Care should be taken to perform the re-inversion of the slave rail only after the delay unit. Otherwise electromagnetic interferences could disturb both rails in the same way, which constitutes a common

mode failure. If, however, the interrupt or reset signal for the slave can be delayed at the source already, the inversion is not necessary. Due to these provisions interferences at these signals would either result in the two cores executing the interrupt or the reset at different points in time (relative to their respective program flow) or only one executing an interrupt or a reset at all. Both these types of diverging control flow can be easily detected by the comparators.

2.8 Error Signal

All internal error detection mechanisms have to signal the error state by an alternating dual rail signal. That means that a correct state of the system is indicated either by "01" or "10" and an incorrect state is indicated by "00" or "11". This allows an evaluation by a dual-rail comparator. The signal has to alternate with every check performed in order to detect stuck-at faults before accumulation occurs. In order to include the output pins in the protection the signal must be routed to the outside of the dual core system in a dual-rail manner.

3 ERROR HANDLING

In order to further support recovery the dual core framework provides an error status register (for each core separately). It is accessible as memory mapped I/O device. The following error sources are mapped to individual bits in this register (see also the respective letters in Figure 1):

(A) *IRAM-Error* indicates a parity error in the instruction memory. Parity is checked internally in the safe instruction memory by totally self checking decoders as outlined in [17]. Parity checks are performed on data, data address and the control signals.

(B) *DRAM-Error* indicates an internal parity error in the safe data memory (analogous to (A)).

(C) *Instruction-Address-Error* detected by a comparator in case the two processor cores do not agree on the instruction address.

(D) *Instruction-Error* detected by a parity check of the instruction word at the dual core input. As already mentioned this additional parity check is intended to protect the bus.

(E) *Data-Address-Error* detected by a comparator in case the two processor cores do not agree on the data address.

(F) *Data-Out-Error* detected by a comparator in case the two processor cores do not agree on the data to be written.

(G) *Delay-Unit-Error-1 and Delay-Unit-Error-2* detected by internal error detection mechanism in the delay-unit. These error signals will be further explained in Chapter 4.2.

(H) *Data-In-Error* detected by a parity check of the data word at the dual core input (analogous to (D)).

If an error is encountered in the master, the corresponding error signal is activated and the bit in the error status register is set. 1.5 clock cycles later the slave will also set its error flag in the same way. All error signals are dual rail encoded. They are combined to a global error signal called "Error Dual Core" by a totally self-checking comparator. This global error signal is routed to an external watchdog and to an interrupt line of the dual core system (see Figure 4).

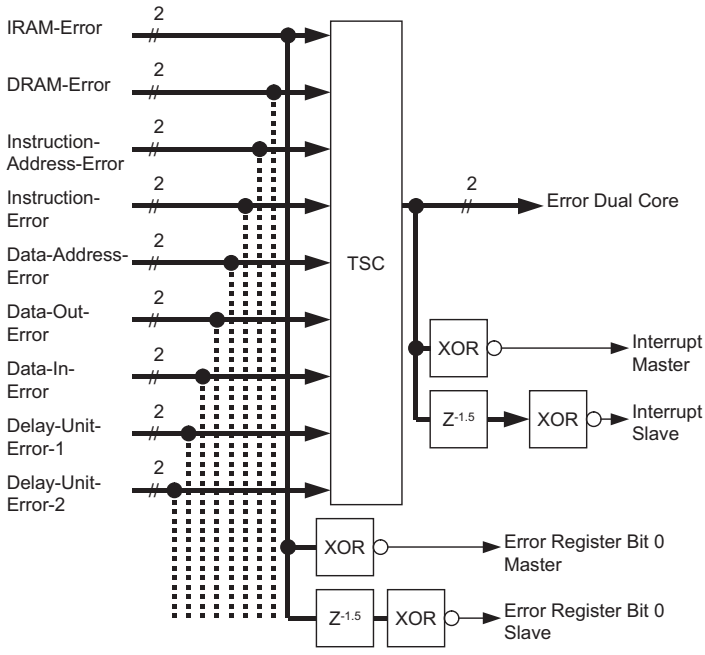


Fig. 4. Error handling

In response to this interrupt the dual core starts the interrupt service routine. At this point the error status register provides extremely important information on the source of the error and thus allows fast, problem specific recovery. Each processor core can acknowledge the interrupt by clearing its error status register. As already mentioned all error signals are dual rail encoded and the comparator is totally self-checking to keep this unit single-fault tolerant as well. Potential bit-flips in the error status register are detected by a diverging program flow of the master and slave. In addition, a watchdog is activated with the global error signal. The dual core has to toggle this watchdog in the course of the interrupt service routine in order to acknowledge the proper recognition of the interrupt.

4 ENSURING ERROR CONTAINMENT

By delaying the slave's operation by 1.5 clock cycles we have introduced the temporal diversity required to minimize the potential for common mode failures. The main drawback of this solution, however, is that the check of the master's data cannot be performed until the slave has provided its results as a reference. During this 1.5 clock cycle latency period the master may write erroneous data to memory, peripheral devices or actuators. Alternatively a faulty master may as well write to an erroneous address or perform a write access instead of a read access. Obviously these scenarios may lead to a failure of the complete system. Even worse, without a clear indication of which external data has actually been adulterated by the faulty master, recovery becomes very difficult.

To solve this problem we have inserted a delay unit to all outputs from the master to the data memory or external memory mapped I/O-components. The penalty is a 1.5 clock cycle delay of write accesses to the singular (i.e. non-duplicated) external components. Since the interface to these external components is bi-directional, the design of this delay unit (shown as a box called "Delay Unit" in Figure 1) is much more complicated than that of the single-directional delay units used for comparison. The related problems will be discussed in the next section.

The purpose of the delay unit is to defer any write access from the master to an external component (memory, I/O module etc.) until the correctness of the respective data, address and control information has been verified by means of comparison with the checker. Since the master may also read data from external components, care must be taken not to delay a read access as well, since this would unnecessarily degrade performance. The implementation of the delay unit is shown in Figure 5.

4.1 Operation of the Delay Unit

The delay unit consists of two branches, a read branch (lower input path of MUX2 in Figure 5 including MUX1) and a write branch (upper input path of MUX2). In the write branch the data are delayed by 2 clock cycles. This delay is longer than the required minimum of 1.5 clock cycles and allows the memory to be clocked with the same clock polarity as the master, which is mandatory for a fast read access. For the sake of consistency we have to delay the corresponding address and control signals in the same way as the data, of course.

In the read branch we bypass the delay by means of MUX2, using the Read/Write signal to control the multiplexor appropriately. We must take care to use the delayed version of the Read/Write signal here, because otherwise the write access would be initiated (a) without the desired delay and (b) 2 cycles before the other related signals are available.

It is easy to imagine that the unequal treatment of read and write leads to problems when switching between read and write access: If we perform a read immediately after a write, the delayed write and the immediate read would have to be performed in parallel. So obviously we must keep a minimum distance of

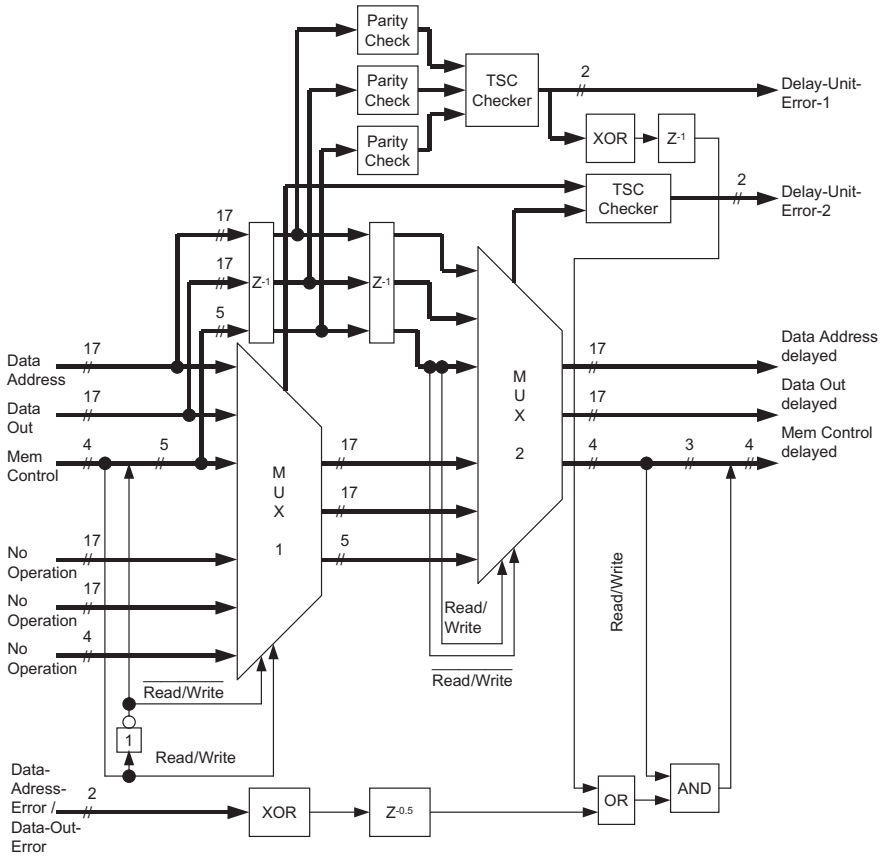


Fig. 5. Delay unit

2 clock cycles between a write and a succeeding read. If we perform a write immediately after a read we get a temporal gap between the end of the read and the start of the delayed write. During this 2 cycle gap MUX2 still activates the read path, but the non-delayed data, address and control information in this path already belong to the write access. To prevent this information from getting to the external bus, we introduce MUX1 and provide uncritical constants ("no operation" in Figure 5) to the lower input of MUX2 during this gap until MUX2 eventually switches to the upper path and performs the actual write access. In principle it would be sufficient to force the signal Read/Write to "read" during this period (which would allow to save MUX1). However, in our generic solution we wanted to ensure for every type of device that might be connected, that only regular accesses are performed in the fault free case. Without MUX1 some state machine within the external component might be upset by the mutilated access,

an access violation be triggered or an unnecessary cache line replacement be performed.

4.2 Protection of the Delay Unit

The Delay Unit is a singular component and hence requires specific protection. In order to protect the interfaces to other components, the signal groups Data Address, Data Out and Mem Control are secured each by a single parity. The parity of these signal groups is routed through the delay unit along with the signals. Since the signals within each group remain mutually independent in the delay unit, the single parity provides sufficient protection against single faults.

Considering the special role of the Read/write signal for controlling the multiplexors we have to pay special attention to its protection. This signal is used alone as well, and not only in context with the whole control signal group; therefore parity protection is not sufficient. Consequently we secure the Read/write signal by a dual rail code right at its entry to the delay unit. This not only prevents a single error from turning a read into a write, it further allows us to implement the multiplexors in a fail-safe fashion as shown in Figure 6.

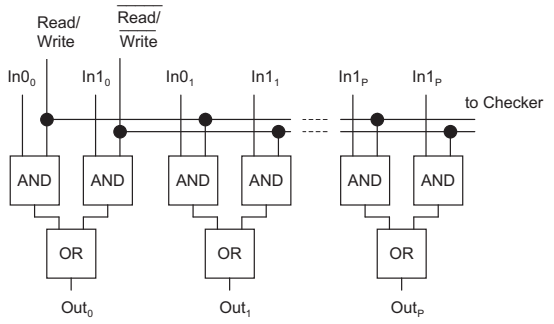


Fig. 6. Internal structure of a multiplexor

In the layout care must be taken to route the dual rail Read/Write signal to the multiplexor first and from there to the self checking comparator as shown in Figure 6. Under this provision an error affecting one rail of the Read/Write signal will be detected by the self checking checker, while a single error in the multiplexor circuit will affect one single output bit and hence be detected by the parity check at the output. Note that a parity check of the signal groups is not necessary in the read path, since even if a write is erroneously turned into a read this will have no effect to the memory contents. Still we have implemented both multiplexors MUX1 and MUX2 as shown in Figure 6 to facilitate detection of an incorrectly switched data path as described above. Note that otherwise this error could not be detected by parity checking, since the data from the wrong path do have correct parity.

To complete our error containment concept, we have to secure the interface to the external component. This can easily be attained by a single parity for each of the signal groups. In addition, the external component should deny a write access, if the delay unit has encountered an internal error. For this purpose a global error signal is available in the delay unit. Most standard components, however, will not be able to make use of this signal and it is up to the dual core to make appropriate use of this indication. We have already given the details on generation and use of the global error signal in Section 3.

Still we can attain write protection in case of an error for standard components like for example the fail safe memory described in [14] as well by simply turning the write into a read (due to timing issues this is only possible for errors indicated by the dual core and not for error sources inside the delay unit). Figure 5 (bottom) illustrates how this is done: The error indication from the master is compared with the slave before it is available as "DRAM Error" with a delay of 1.5 clock cycles. By adding another 0,5 cycles delay we can attain a temporal alignment of this error signal with the corresponding data word in the delay unit. Using an AND gate finally allows us to mask the Read/Write signal and thus inhibit a write access. This signal change will furthermore cause a parity error for the control signal group. The XOR gate shown in Figure 5 turns the dual rail signal into a single rail one.

5 RESIDUAL SINGLE POINTS OF FAILURE

Common-mode failures [19] are failures that are affecting by a single reason more than one part of the system – usually redundant components – in the same way, giving rise to an erroneous output that cannot be detected by comparison. As already mentioned we have minimized the potential for common mode failures by operating the processor cores in time diversity.

Single points of failure are locations in the system for which no redundancy is available to detect or mask a failure. Some of these locations have already been discussed in the previous chapter, like for example all incoming and outgoing signals and buses, hence we do not further consider them here. The remaining components that constitute potential single points of failure are the clock-signal and the power supply.

5.1 Clock

Hypothesized failures of the clock signal are a totally missing clock and a partially missing clock (i.e. for some components). Failures in the clock generator resulting in pulse omissions are not considered, because they pertain to external components. Transients on the clock line are common mode failures and should hence be covered by the time diversity. As shown in [20] a missing clock signal is difficult to discover by an internal mechanism, hence an external reference, e.g. a watchdog, is necessary. Consequently one of the two cores can be assigned the responsibility to trigger the

watchdog. This, however, does not provide complete coverage of all partial clock failures unless the following layout rule is obeyed:

If the watchdog is triggered by the master and the clock is disrupted only for the delay components and the slave (Figure 7 a), the comparators will always display a correct signal, even when the master produces erroneous outputs, because the data frozen in the delay component and the data frozen in the slave will probably be the same. As a consequence both the watchdog and the comparator will fail to detect this error. Therefore it is necessary to route the clock first to the slave and then to the master that will trigger the watchdog (Figure 7 b). If now a total clock stall happens or a clock stall happens in the clock path to the master (Figure 7 b), pos. 1 and 2), the watchdog will detect this error. The comparator in turn will recognize the failure of a clock stall directly before the delay component (Figure 7 b) pos. 3) or a clock stall before the slave (Figure 7 b) pos. 4). So all partial clock failures are covered. In addition it must be ensured that the watchdog is timed from a source independent of the clock to be checked.

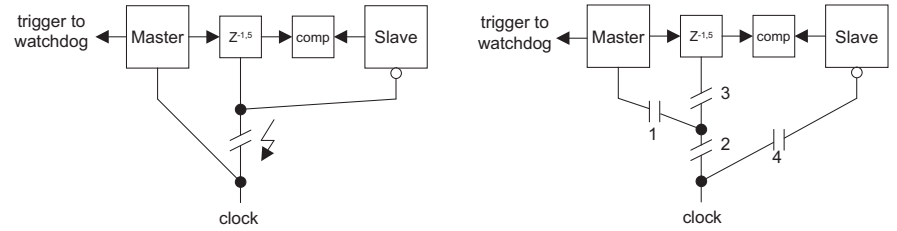


Fig. 7. Clock tree: a) a disadvantageous and b) a advantageous implementation

5.2 Power Supply

Disruption in the power supply will either result in a total stop of the system or – in case of spikes or transient outages – in a temporary disruption of the operation of one or more components of the dual core. Again the time diversity ensures that the disruption causes different effects in the two cores. Moreover, such types of failures tend to result in a program flow disruption such that the processor will very likely not trigger the watchdog correctly. If only one component is disrupted, the comparators will detect the failure. Low supply voltage of the comparators also will be detected, because the dual-rail signals will not be able to display a proper high logic level.

6 FAULT INJECTION EXPERIMENTS

In order to validate our theoretical analysis we have carried out a series of fault injection experiments on the proposed dual core frame. A dual core processor consisting

of two SPEAR cores and the fault tolerant frame incorporating the delay unit for the output has been implemented in VHDL. As shown in Figure 8 two instances of this improved dual core architecture have been used in the test environment, one for the device under test and another for the golden device that operates in lock-step with the device under test and serves as a fault-free reference.

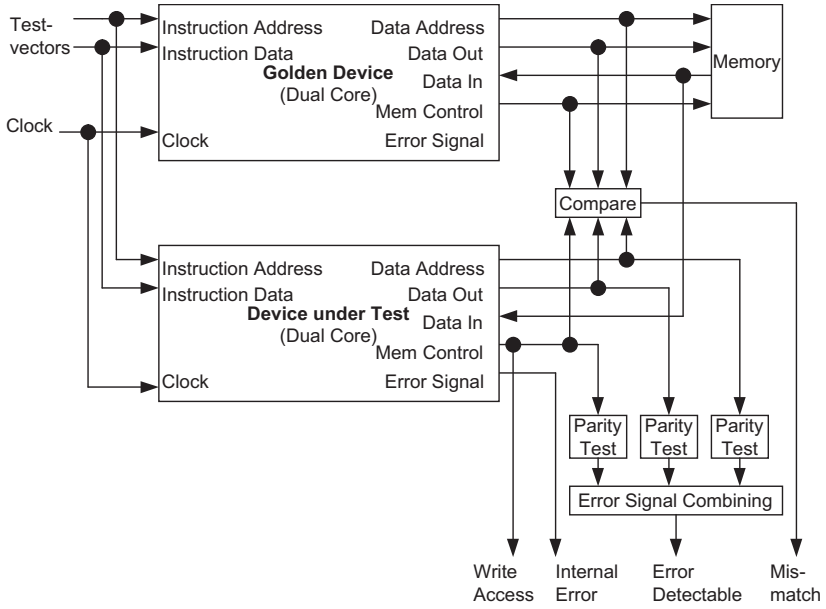


Fig. 8. Test environment

During the experiments we need to observe two important phenomena: (1) has the injected fault become effective and (2) has the resulting error been properly detected in time, i.e. before polluting external devices. To address question (1) we compare all signals issued from our device under test to (virtual) external components like the data memory or memory mapped peripherals – such as data address, data out and memory control – with the respective reference signals from the golden device. In case of a comparator mismatch (indicated by the signal "Mismatch") we can conclude that the device under test is attempting an erroneous access. By observing the Read/Write control signal ("Write Access" in Figure 8) we can further judge whether this access is a malicious write or a harmless read – or has at least been turned into a read by our protection scheme. With respect to question (2) we can check whether an error has been detected by a mechanism inside the dual core framework (signal "Internal Error") or is detectable by an external device that employs a parity test on the buses (signal "Error Detectable"). By monitoring all these signals on a cycle-by-cycle base we can assess the temporal relations, in particular we can find out whether error detection occurred before an erroneous write.

The test environment has been synthesized to a net list in EDIF format (electronic design interchange format). Fault injectors were inserted into this net list at each logic gate input of the device under test. The fault injector is capable of injecting stuck-at-one and stuck-at-zero faults. The faults were activated while running a test workload on the dual core system. In order to facilitate a more detailed analysis, we have kept track of whether a fault has been inserted into the master core, the slave core or the frame, and added this information to the respective entry in the observation record. The stuck-at fault model is considered sufficient because the concurrent error detection mechanisms are triggered with an extremely short latency, so that the faults were detected very fast. Under this prerequisite the observed coverage results can be directly projected to the bit flip or transient fault model that are anticipated to dominate during field operation.

In order to assess the gain of the proposed delay unit with respect to error containment we have carried out the experiments once with and once without delay unit.

7 EXPERIMENTAL RESULTS

The results for the dual core without the delay unit are shown in Table 1.

component	master	slave	frame	overall
detected without effect	545	51 324	3 296	55 165
detected before effect	3 226	0	328	3 554
detected during read access	307	0	22	329
detected during write access	0	0	0	0
detected after read access	**31 686**	**0**	**46**	**31 732**
detected after write access	**15 587**	**0**	**32**	**15 619**
not detected without effect	3 755	3 748	760	8 263
not detected with effect	0	0	0	0
	55 106	55 072	4 484	114 662

Table 1. Results of the fault injection for the dual core system without delay unit

Altogether **by inserting a stuck-at-one and a stuck-at-zero at each gate input** 114 662 faults **could be** injected into the dual core system. 106 399 of these faults could be activated by the workload in the sense that either the signal "Internal Error", "Error Detectable" or "Mismatch" became activated. This is about 92,8 percent of all injected faults. 4 484 faults were injected into the frame containing the safety critical single components such as the parity checker and the buses. 760 of these could not be activated, because we could not simulate external components such as memory with an address space of 16 bit. Although this is not directly proven by a measurement result, we can reasonably assume that these 760 faults would also be detected, because the error detection mechanisms for the involved logic gates are the same as those for the others. Faults in the master mostly became effective. The

proportion of errors in the master that were detected before their effect propagated to the output depends on the workload. Hence these numbers would look different for a different workload. As expected faults in the slave never had an effect, since the slave's output is used for comparison and error detection only and not visible for the external components.

An important result was the confirmation that all activated faults were detected within latency of at most 1.5 clock cycle after they became effective. However, 15 619 of the activated faults could be detected only *after* having had an effect by a write access. This significant share of 14 % late detections is problematic, since external components may have already been polluted by a faulty master and as a result recovery becomes difficult.

This observation justifies the proposed insertion of the Delay Unit. Table 2 summarizes the results of the fault injection experiments we have performed on the dual core with Delay Unit.

component	master	slave	frame	overall
detected without effect	204	51 170	3 517	54 891
detected before effect	19 047	98	734	19 879
detected during read access	559	0	601	1 160
detected during write access	0	0	320	320
detected after read access	**31 455**	**0**	**87**	**31 542**
detected after write access	**0**	**0**	**0**	**0**
not detected without effect	4 269	4 276	1 073	9 618
not detected with effect	0	0	0	0
	55 534	55 544	6 332	117 410

Table 2. Results of the fault injection for the dual core system with delay unit

Here a total of 117 410 faults were injected of which 107 792 (about 92 %) were activated by the workload. The frame containing the safety critical single components such as the parity checker, the buses and the output delay unit was now subjected to 6 332 faults. 1 073 of these could not be activated due to the same reasons as above. Again we can reasonably assume that these 1 073 faults would also be detected, because the error detection mechanisms for the involved logic gates are the same as those for the others. An interesting observation in this fault injection experiment is, that 98 faults in the slave are are causing a mismatch between the device under test and the golden device. This stands in contrast to the results in Table 1 where the faults injected into the slave never had an effect. The reason for this unexpected behavior is caused by our protection mechanism that turns a write into a read in case of an error. Injecting a fault into the slave may trigger this mechanism 1.5 clock cycles after the master has started its write access, which is just sufficient to inhibit the write before the 2 cycle delay is over. Eventually this causes a mismatch of the read/write signal between device under test and golden device.

The distinction between read and write accesses enabled by tracing the signal "Write Access" allows some important observations: All faults that manifested as a write access (including reads that turned into a write as a consequence of the fault) are either detected before or during the access to the external component, which facilitates a well targeted recovery. The relevant detection mechanisms are "Internal Error" and "Error Detectable" (i.e. parity on the external buses). All erroneous read accesses are detected as well, but mostly after the access to the external component. This is, however, no problem, because no external memory contents will be adulterated by a read (but for the rare cases of consuming read in special peripheral components). So although error detection cannot in general be guaranteed before the erroneous access to an external device, all potentially malicious types of access are properly handled and error containment is effectively enforced. In addition all types of error observed throughout the experiment regardless of their severity have been detected within a latency of at most 2 clock cycles.

8 CONCLUSION

We have proposed a generic frame for a dual core system that can be used to implement a fault-tolerant processor system with virtually all modern processor cores. Additionally we have proposed an extension that allows to ensure fault containment and hence facilitates fast, fine-grained recovery even if the processor cores are operated in time diversity.

With a careful implementation mainly based on the self checking principle and two rail coding complete coverage of single faults could be obtained even for the singular function units of the dual core framework. Our analysis has shown that in this implementation all single points of failure and all common-mode failures – even at the input, output or error signals – can be eliminated, if several layout rules are obeyed and if the operation of the slave is delayed by 1.5 clock cycles. Comprehensive experimental results have confirmed this theoretical analysis: No coverage violation has been observed.

REFERENCES

[1] Baumann, R.: The Impact of Technology Scaling on Soft Error Rate Performance and Limits to the Efficacy of Error Correction. Electron Devices Meeting, 2002. IEDM '02. Digest. International, 8–11 Dec. 2002, pp. 329–332.

[2] Georgakos, G.: Radiation Induced Soft Error Rate for SoC Designs, Infineon Customer Information, Vers. 2.1, Febr. 2003.

[3] Allan, A.—Edenfeld, D.—Joyner, W. H.— Kahng,A. B.— Rodgers, M.— Zorian, Y.: 2001 Technology Roadmap for Semiconductors. Computer, Vol. 35, 2002, No. 1, pp. 42–53.

[4] Boehl, E.—Lindenkreuz, T.—Stephan, R.: The Fail-Stop Controller AE11. Proceedings of the International Test Conference, 1997, pp. 567–577.

[5] Elliott, I. D.—Sayers, I. L.: Implementation of 32-Bit RISC Processor Incorporating Hardware Concurrent Error Detection and Correction. IEE Proceedings of Computers and Digital Techniques, Vol. 137, 1990, No. 1, pp. 88–102.

[6] Russell, R.—Elliott, I. D.: Design of Highly Reliable VLSI Processors Incorporating Concurrent Error Detection/Correction. Euro ASIC, 1991, pp. 316–321.

[7] Pflanz, M.—Vierhaus, H. T.: Online Check and Recovery Techniques for Dependable Embedded Processors. Micro, IEEE, Vol. 21, 2001, No. 5, pp. 24–40.

[8] Patel, J. H.—Fung, L. Y.: Concurrent Error Detection in ALUs by Recomputing with Shifted Operands. IEEE Transaction on Computer, Vol. C32, 1982, No. 7, pp. 589–595.

[9] Li, J.—Swartzlander, E. E.: Concurrent Error Detection in ALUs by Recomputing with Rotated Operands. Defect and Fault Tolerance in VLSI Systems, Proceedings of the IEEE International Workshop on, 1992, pp. 109–116.

[10] MC88100 32-Bit RISC Microprocessor Technical Summary, Document MC88100/D, Motorola Inc. 1990.

[11] Lenoski, D. E.: A Highly Integrated, Fault-Tolerant Minicomputer: the NonStop CLX. Compcon Spring 88, Thirty-Third IEEE Computer Society International Conference, Digest of Papers, 29 Feb.–3 March 1988, pp. 514–519.

[12] Delvai, M.: SPEAR Handbook. Technical Report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[13] Delvai, M.—Huber, W.—Puschner, P.—Steininger, A.: Processor Support for Temporal Predictability – The SPEAR Design Example. Proceedings of the 15th Euromicro Conference on Real-Time Systems, July 2003, pp. 169–176.

[14] Kottke, T.—Steininger, A.: A Fail Silent Memory for Automotive Applications. IEEE European Test Symposium, Infomal Digest of Papers, 2004, pp. 253–258.

[15] Kanekawa, N.—Meguro, T.—Isono, K.—Shima,Y.—Miyazaki, N.—Yamaguchi, S.: Fault Detection and Recovery Coverage Improvement by Clock Synchronized Duplicated Systems with Optional Time Diversity. IEEE Proceedings of FTCS, Vol. 28, 1998, pp. 196–200.

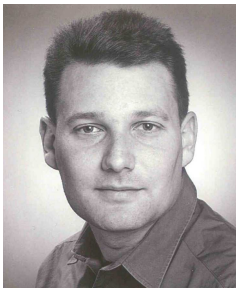
[16] Manthani, S. R.—Reddy, S. M.: On CMOS Totally Self-Checking Circuits. Proceedings of the International Test Conference, 1984, pp. 866–877.

[17] Abramovici, M.—Breuer, M. A.—Friedman, A. D.: Digital Systems Testing & Testable Design. Wiley-IEEE Press, 1994.

[18] Kundu, S.—Sogomonyan, E. S.—Goessel, M.—Tarnick, S.: Self-Checking Comparator with One Periodic Output. IEEE Transactions on Computers, Vol. 45, 1996, No. 3, pp. 379–380.

[19] Laprie, J. C.: Dependability: Basic Concepts and Terminology. Dependable Computing and Fault-Tolerant Systems, Vol. 5, Springer Verlag, 1992.

[20] Usas, A. M.: A Totally Self-Checking Checker Design for the Detection of Errors in Periodic Signals. IEEE Transactions on Computers, Vol. C-24, 1975, No. 5, pp. 483–489.



Thomas Kottke received the diploma in electrical engineering from the University of Stuttgart in 1999. During his employment at Rohde & Schwarz he was concerned with developing universal radio communication testers. In 2002 he joined the Bosch corporate research and development department for chassis systems. His working area is fault tolerant computing with a focus on safety critical automotive applications.

Andreas Steininger received his diploma in electrical engineering in 1988 and the Ph.D. in computer engineering in 1994, both from the Vienna University of Technology, where he is currently Associate Professor at the Institute for Computer Engineering. His research interests include design and evaluation of fault-tolerant computer architectures, testing, real-time systems and asynchronous processors.