Masterarbeit

Design and Implementation of a Self-test Concept for an Industrial Multi-core Microcontroller

Burim ALIU

Institut für Technische Informatik
Technische Universität Graz

Begutachter: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
Betreuer: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
Dipl.-Ing. Roland Mader

Graz, im Mai 2012

Kurzfassung

Seit einigen Jahren werden Mikrocontroller in der Automobilindustrie als Steuergeräte für Bremsen oder Motorsteuerungen genutzt. Heutzutage gibt es viele Multi-Core Mikrocontroller, die aber nicht alle für sicherheitskritische Anwendungen in der Automobilindustrie verwendet werden können.

Immer, wenn einige kritische Aufgaben ausgeführt werden müssen, muss sichergestellt sein, dass der Mikrocontroller ordnungsgemäß funktioniert und das die funktionale Sicherheit im System erhalten bleibt.

Während der Laufzeit können Störungen (z.B. Übergangsfehler, gekoppelte Fehler usw.) in den verschiedenen Teilen des Mikrocontrollers auftreten. Diese Fehler können zu Ausfällen des gesamten Systems führen.

Zur Erkennung und Handhabung solcher Fehler sind Mikrocontroller mit diversen Sicherheitsfunktionalitäten ausgestattet. Diese können in Form von Architekturen eines Lock-Step Modus, asymmetrischen Modus, symmetrischen Modus oder Selbsttests realisiert sein, die entweder in Hardware oder in Software implementiert werden.

In dieser Arbeit wurden verschiedene Multi-Core-Architekturen und verschiedene Selbsttestalgorithmen bewertet.

Das erste Ziel war, zur Verfügung stehende Multi-Core-Mikrocontroller mit Bezug auf die gewählten Kriterien zu bewerten. Ein weiteres Ziel war die Entwicklung und Implementierung eines Online-Selbsttest-Konzepts für einen ausgewählten Multi-Core-Mikrocontroller. Dieses Konzept beinhaltet, software- und hardwarebasierte Selbsttests für bestimmte Teile des ausgewählten Mikrocontrollers wie RAM oder CPU.

Diese Hardware und softwarebasierten Selbsttests werden durch einen externen Watchdog ergänzt, der es ermöglicht, den Programmablauf zu überwachen.

Abstract

During the last years microcontrollers have been used for control devices in the automotive industry like auto-brakes, motor control etc. Nowadays there are many multi-core microcontrollers, but not all of them can be used in the automotive industry for safety critical applications. Whenever some critical tasks are executed, it must be ensured that the microcontroller is working correctly and the system maintains the functional safety.

During runtime, faults can occur in various parts of the microcontroller such as ALU, RAM or peripherals starting from, stuck at faults, transition faults, coupled faults etc. These faults can cause failures of the complete system.

To detect and handle such faults, microcontrollers are equipped with safety features in the form of architectures like lock-step mode, asymmetric mode, symmetric mode or self tests which are implemented either in hardware or in software.

Different multicore architectures and different self testing algorithms were reviewed for the thesis. The first goal was to evaluate available multicore microcontrollers with respect to chosen evaluation criteria. Another goal was to design and implement an online self test concept for a selected multi-core controller, which includes software based self tests as well as hardware built-in self tests for specific parts of the chosen microcontroller like RAM or CPU cores. These hardware and software-based self tests are supplemented by an external watchdog that is used for the program flow monitoring.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....

date (signature)

Acknowledgments

This thesis was performed at the Institute for Technical Informatics at the Technical University of Graz in the scope of the MEPAS[1] project in cooperation with AVL List GmbH[2] and The Virtual Vehicle Competence Center[3].

First of all I want to thank my supervisors Dipl.-Ing. Roland Mader and Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger who have supported me during the time I was working on my thesis.

During my study I was greatly supported by my family: my father, my mother, and my two brothers.

Graz, May 2012 Burim Aliu

[1]Methods and processes for automotive embedded software development, verification and validation
[2]http://www.avl.com
[3]http://www.vif.tugraz.at

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Nowadays, different accidents happen while driving a car or travelling by plane. There were cases in which cars on the highway automatically executed the emergency braking without interaction by the driver, or such cases when, during a radiotherapy, the device killed a patient with an overdose [Mon99].

In our daily life, everyone is accustomed only to use different devices, but many do not realize that the machines have a certain functional area. The slightest change in the system can cause a total system failure.

These changes can be different in the electronic systems. It is sufficient that only a bit flip occurs, to cause that the brakes of a car do not work because this bit flip makes the program flow in the control unit jumps somewhere else.

Such changes or faults which produce unwanted results in a system can have several causes. If a complete control system is considered, faults can be found in different forms: control faults, development faults or mechanical faults because of mechanical utilization.

Faults can be classified as follow [Mon99]:

1. Source: development faults, run-time faults
2. Type: permanent, sporadic, conditional faults
3. Region: value, time, unsolicited actions

Development faults are permanent faults which occur during the development process because of the high complexity, low verifiability or insufficient specification and are found during the system lifetime. Run-time faults occur because of hardware, communication, mechanical failures or overloading.

Permanent faults last until they are treated. Sporadic faults occur spontaneously and are not reproducible until the fault source is found. Conditional faults are caused by temperature, vibration or radiation disorders.

Fault based on the region can occur if the system does not generate the correct output or if it fails to meet the timing requirements. Such failures can occur if the real time task does not fulfil the response time window or the scheduler cannot practically provide the theoretically controllable load.

Once a fault has occurred, it can take a long time until it brings the entire system in an undefined state. Depending on where it occurs, it may have different effects on the system and the environment. When the fault is in the processor core it is very likely that it can produce an accident.

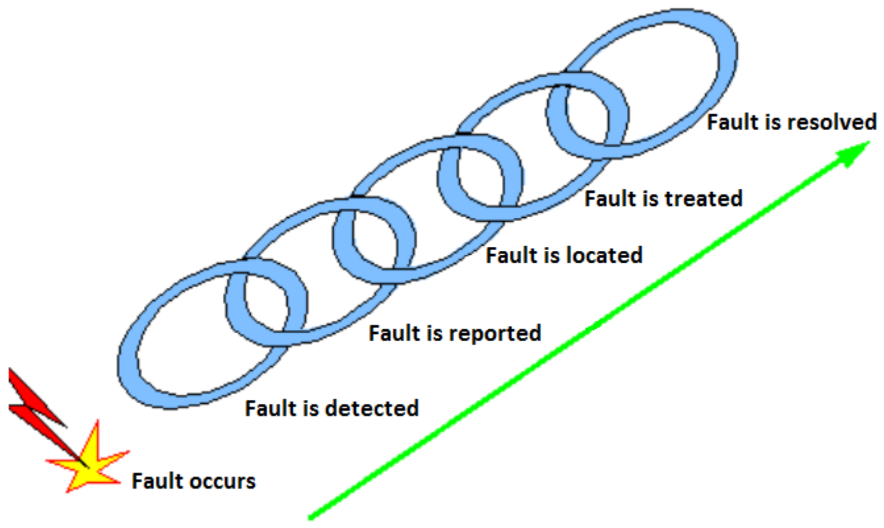


Figure 1.1: *Action chain for fault handling [Mon99]*

A system must always be ready to intercept and handle faults. Fault-tolerant measures must ensure that the system still works correctly if a fault is detected, or the system must switch into a safe state. The complete chain steps of fault handling after a fault occurs, illustrated in Figure 1.1, are: detection, reporting, delimiting, treating, and resolving.

After the faults are detected, other modules must be alerted. Then it must be located to prevent the fault from spreading over the system. Hence it must be determined which module is responsible for the fault and the particular module must be treated by fault tolerant mechanisms. At the end, the fault can be resolved either by fault model or by user intervention. All steps in the action chain can be taken in two ways:

1. Operative - The fault model is always and periodically looking for faults in the system in order to find and correct them.

2. Responsive - If errors occur, they are reported. A module detects the error and reports it to an error handling module.

Such operative or responsive measures can be integrated into the hardware or in form of software-based self-tests (SBST). SBST are testing techniques, executed in the background of the application. However, the integration of such SBST tests needs additional ROM space and execution time.

As a motivation to write this thesis, there was the challenge to see and learn more about, how the functional safety can be achieved. This is not only about how the concepts are implemented but also to explore different algorithms and their mathematical background. The result should be a software concept which could be implemented in such a system in order to maintain the safety integrity.

During the preparation of this thesis, different multi-core microcontrollers for safety-relevant embedded systems which are available on the market were evaluated in terms of safety-related components, fault coverage and performance.

1.2 Objective

Designing an effective online testing strategy requires information about the architecture, RAM, ROM, instruction set, branch prediction etc. This information must be collected and evaluated because in (E/E/PES) systems, failures can arise in different parts of the system, which are caused by hardware, software, electromagnetic or different influences.

For creating such an online-testing strategy, sufficient knowledge about the architecture of the system and different algorithms should be present. Hence, the first task of the thesis was:

1. Study different safety strategies and architectures
2. Study different algorithms, their mathematical structure and fault models
3. Create a list of evaluation criteria based on which the microcontrollers will be evaluated
4. Check which safety features and protection mechanisms that are implemented in different architectures
5. Based on the gained knowledge create a testing strategy
6. Implement the online periodic testing strategy and evaluate the results

After the evaluation task is finished, the online testing strategy which will be implemented to maintain the integrity of the system during runtime, the testing strategy would invoke these functional properties defined in previous tasks. This means that as a part of the strategy, the support of hardware tests will be included.

At the end, the gained knowledge will form an on-line periodic strategy where the tests which are developed must collaborate with each other to run in the background during runtime and the application will perform its normal operation.

1.3 Structure

The second chapter gives a detailed introduction to the field of safety-related concepts, architectures and software-based self-tests. Fail-safe safety strategies are used to achieve a safe state in case of a fault. Therefore, online fault detection is necessary.

Safety architectures are created using different algorithms which are proofs of concept in the field and provide safety using redundant hardware components that minimize the causes of common cause failures (CCF).

Software-based self tests (SBST) are described which are part of a solution to provide safety using the functional and structural information of microcontrollers. Using this information, tests are generated and applied to cover faults which can cause unsafe states.

At the end of the chapter, the state of the art in the field of RAM testing algorithms are described starting from the mathematical background, complexity, and execution time.

The third chapter describes the different architectures which are evaluated and based on the evaluation criteria; a concept is built to cover the faults during runtime in the TMS570 microcontroller. The concept is an online strategy which consists of hardware and software tests.

Chapter four presents the implementation of the concept described in the fourth chapter. This chapter covers the evaluation board and the tool chain which are used for the development, and each test is described regarding the implementation, execution time and fault injection.

Chapter five presents the conclusion of this thesis and prospects for future work.

Chapter 2

Related Work

2.1 Hardware architectures and strategies

2.1.1 Faults and Failures

In the field of safety related systems, microcontrollers are very common devices. The safety integrity of these devices must be maintained during system operation. Appropriate testing strategies are required to detect faults in microcontrollers during runtime to be able to achieve and maintain a safe state.

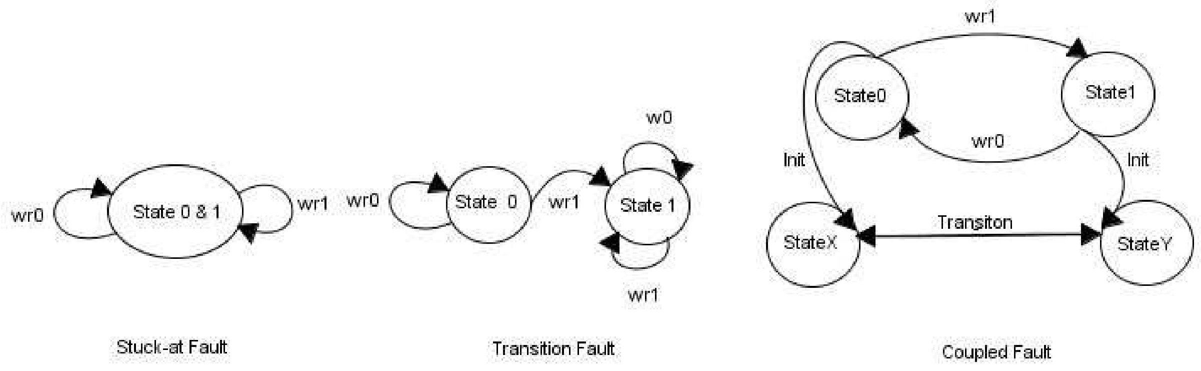


Figure 2.1: *Finite State Machines describing basic faults*

Faults are generally subdivided into:

1. Hardware faults
2. Software faults

Another group of faults which can be found in architectures where cores are physical coupled are the common cause failures (CCF) [Tum09]. Here, a fault as the root cause affects the entire system because of the coupling mechanism. The fault as root cause can be: temperature spreading, using same power supply, or clock source for both cores etc.

Dependent on which part of the microcontroller the faults occur, they are called as follows:

1. CPU Faults
   - Arithmetic faults
   - Logical faults
   - Conditional faults
2. RAM Faults
   - Stuck at faults
   - Transition faults
   - Coupling faults
3. Peripheral Faults

The arithmetical faults cause the ALU to perform calculations incorrectly. This can be multiplication, addition, division etc. Logical and conditional faults are the reason that comparisons and jumps in programs are not performed correctly and can cause an instruction routine jump elsewhere than intended.

But the most important faults which must be avoided in a system are RAM faults. Any single bit fault in a RAM cell can be a root cause for other failures in the system such as previously described CPU faults.

Stuck-at faults(SAF) can occur in two forms: stuck at 0 and 1. As illustrated in the finite state machine, stuck at 0 or 1 means that any attempt to change the state of a RAM cell from 0 to 1 or vice versa fails and the cell holds the original value.

Transition faults(TF) prevent a cell from changing its actual state from 0 to 1 or 1 to 0. The difference between transition faults and stuck at faults are that, transition faults affect just one side of the state transition.

Coupling faults(CF) ensure that any transition in one RAM cell from one state to another causes the changing of their actual state to one or to many other cells.

These three basic faults are the basis for other faults that can occur in RAM memory and are illustrated in the Table 2.1 [WWW06]:

Fault	**Abbreviation**
Address decoder faults	AF
Stuck-open fault	SOF
Data retention fault	DRF
Inversion coupling fault	CFin
Idempotent coupling fault	CFid
State coupling fault	CFst
Read disturb fault	RDF

Table 2.1: *Table with faults that can occur in RAM*

Each fault is illustrated in Table 2.1 and has the following meaning:

- The address decoder takes one or many input bits and constructs a specific address as output. If some specific cells, addresses or blocks of addresses, cannot be accessed, a fault is present.
- If a cell cannot be accessed because of a faulty voltage amplifier which cannot sense the voltage difference of the bit line, it is a stuck open fault.
- If a specific cell cannot hold its value for a specific time due to a leakage current or pull-up resistor problems, it is a data retain fault.
- While a writing operation is performed on the cell ”j”, the cell ”i” changes its value and the fault is called an inversion coupling fault.
- If a cell ”j”, during a transition operation forces any other cell ”i” on the RAM memory to be in a fixed value forever, the fault is called idempotent coupling fault.
- If a coupled cell or line ”i” makes a forced transition to a specific value x only if the coupling cell or line ”j” is in state y, then it is described as a state coupling fault.
- If a cell during reading changes its logical value, it is called read disturb fault.

2.1.2 Fail-Safe System Strategies

Because of the various fault types, different strategies are developed to maintain the integrity in a controller. In order to fulfil all specific constraints in different architectures, adequate strategies are needed.

These strategies must fulfil different criteria, but the most important of them are [SD06]:

- A strategy must satisfy system safety requirements.[1] Fault detection and handling must be provided within the safe system response times.
- The system is transitioned to a safe state within the required safe fault response time.
- Level of independent checking provided
- Performance
- Technology availability
- Development effort

So far, there have been different proven strategies that generally use watchdogs, dual, symmetric and asymmetric architectures.

[1]SIL/ASIL levels

2.1.2.1 Single Controller Strategy

A single controller executes the instructions of an application. At the same time, self-tests that can be hardware circuitry or software self tests, observe the internal state of the architecture.

With this mechanism, it can only achieved to catch the conditions related to the program flow such as endless loops or incorrect flows, but it does not catch wrong calculations in the output. A possible solution for this strategy is to implement protection mechanisms that periodically check the internal state of the ALU or RAM, and to have a separation between the critical parts of the architecture.

The safety of such systems depends on the reliability of the watchdog circuits. If the properties of such a circuit [HA99] are observed, it is possible that in some special cases the watchdog does not detect the failures.

If the reset occurs within a defined period, the watchdog timer reacts. If however, a fault occurs in the microcontroller, a faster reset can be generated, and the watchdog is not able to differentiate between a normal refresh with period T and a period that is smaller than T.

To overcome this limitation, extra logic parts must be included, to make the watchdog reliable [HA99]. Although equipped with extra protection mechanisms, it has the problem that extra self-test routines must be implemented (hardware or software) for the integrity. This strategy is illustrated in Figure 2.2.

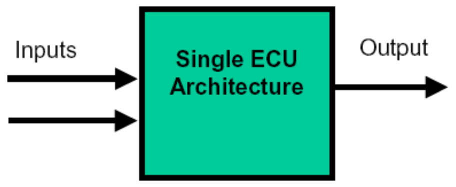


Figure 2.2: *Single controller strategy [SD06]*

2.1.2.2 Symmetric Controller Strategy

The next strategy [SD06] is the symmetric strategy. Here, the instructions are executed in parallel and the results of the outputs are compared. Synchronization plays a very important role in that matter, because every processor has a slightly different clock frequency.

The comparison can be made for every instruction or periodically (at specific points of time). This strategy can cover almost all errors that may occur during calculation in hardware or software.

Advantages of this strategy can be specified as:

- Almost all random hardware faults can be detected
- Complex self-checking techniques can be avoided

Disadvantages or limitations:

- Both processors must be synchronized
- Large size and cost because the architecture has two processors

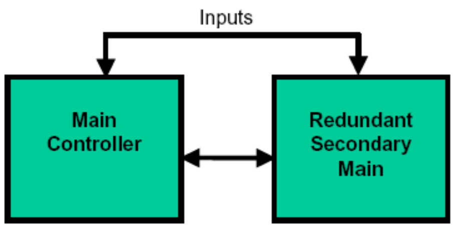


Figure 2.3: *Symmetric controller strategy [SD06]*

2.1.2.3 Dual Core Controller Strategy

This strategy is similar to the single controller strategy. The difference is that this strategy, has two cores that compare the output of each other and can be combined with a watchdog circuitry to avoid common cause failures. Additional software self-checking diagnostics can be used [LCD$^+$05], such as checksums, redundant coding or RAM tests to extend the integrity of the controller.

Failures that occur in the relation with the CPU can be detected immediately, but it depends on the execution time of the self-checking routines, for example ALU Checking, output checking, etc.

Two architectures that are based on this strategy are Dual-Core Lock Step on Page 20. Other possible system architectures can be found in [Has]:

- Heterogeneous $\rightarrow$ runs different OSs and has different types of cores
- Asymmetric Multi-Processing $\rightarrow$ containing two or more cores of the same type and able to run different or same OS
- Symmetric Multi-Processing $\rightarrow$ runs same OS with same core types

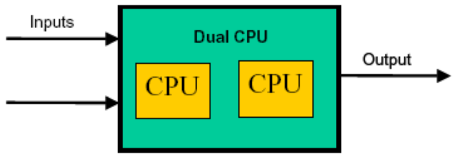


Figure 2.4: *Dual-Core controller strategy [SD06]*

2.1.2.4 Asymmetric Controller Strategy

This strategy as depicted by Figure 2.5 consists of two processors that are connected to each other via a dedicated line. The second controller can be perceived as a watchdog circuit, which monitors the primary processor. It does not execute application-specific codes, but it only checks the main controller.

Typically, the second controller is a standard microcontroller (off-the-shelf), because for using specific ASICs, it needs more design time and has higher costs.

This strategy has a good diagnostic coverage and guarantees controller integrity, because an external processor checks its integrity and not an internal self-checking procedure.

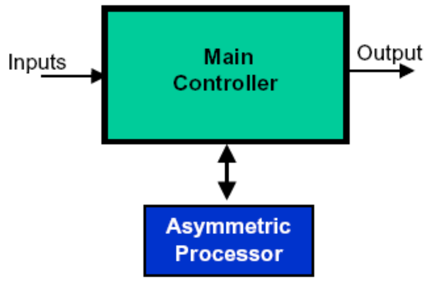


Figure 2.5: *Asymmetric controller strategy [SD06]*

2.1.2.5 Distributed Controller Strategy

In this strategy, two controllers are connected with each other in a network. One controller serves as primary controller and the second checks the functionality of the first. According [SD06] to the way, how the functionality of the second controller is implemented, it can be grouped in two methods:

1. Independent execution of checking procedure
2. Independent checking of the primary processor

With the first method, the primary controller sends the received signal from the secondary controller and the results to the second controller, so the second controller performs the same operations, and if there is a discrepancy, it can shut-down or disable the main controller.

With the second method the second controller periodically sends a seed to the primary controller, performs calculations and sends the result back. This result is compared to a pre-calculated value to see if something in the primary controller is going wrong. The "seed" is used in many self-checking methods to check the execution of the program, and to see if a program has executed all the branches. For more information, refer to [LCD$^+$05].

Depending on factors like network bandwidth, response time and synchronization, two shutdown strategies of the main microcontroller are possible:

1. Dedicated shutdown via a dedicated wire from the second to the primary controller (hard-wired)
2. Shutdown via a local network controller in the primary controller

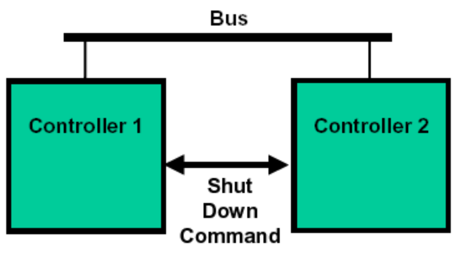


Figure 2.6: *Distributed controller strategy [SD06]*

2.1.3 Multi-core Architectures with redundant structures

Most of today's modern controller architectures use redundancy to give controllers the fault-tolerant property. But only with redundancy, the fault-tolerant property cannot hold because of common cause faults that can occur in single or dual core architectures [Tum09].

Other measures are needed to extend this capability, such as runtime tests, hardware isolation like guard rings or memory protection mechanisms. In [BFM+03] different architectures are described that will be explained later on.

2.1.3.1 Lock-Step Dual & Dual Lock-Step Processor Architectures

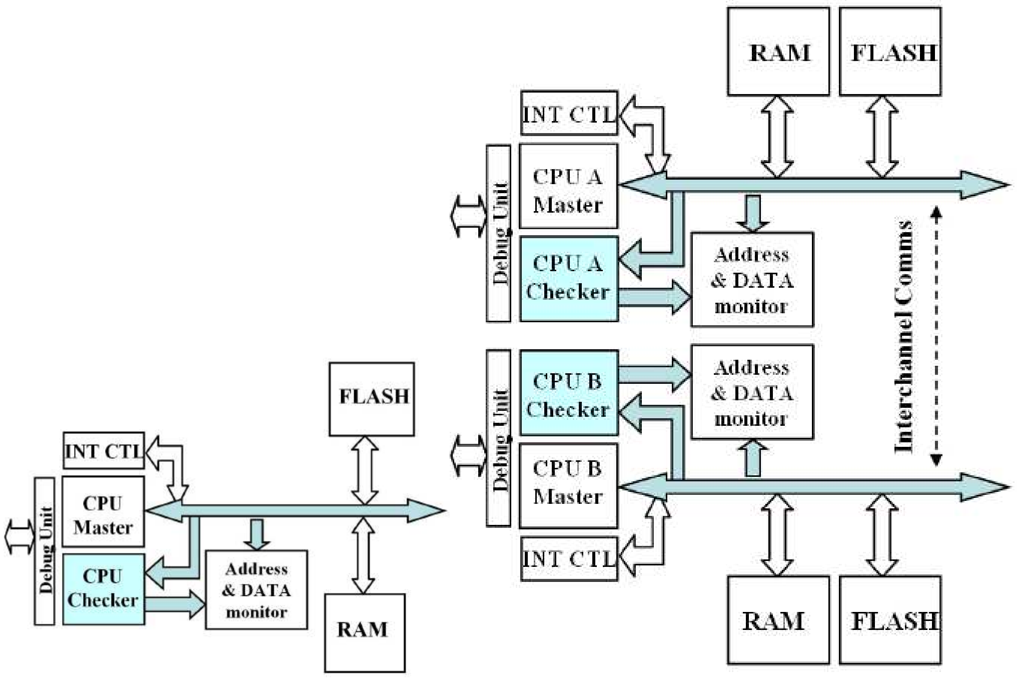
The first architecture is a so-called lock-step. It contains two processors that are connected by a compare unit and are called "master" and "checker".

The master executes the instructions and the checker is responsible to execute the same instructions as the "master". The compare unit as illustrated in Figure 2.7a continuously compares whether the calculations are correct.

Another important property is that the Compare Unit check only whether an error has occurred during calculations, but it does not identify which part of the system causes this error. Normally, to detect such errors and errors that happen because of common cause faults [Tum09] an extra logic is needed implementing Error Correction Codes and parity bits in peripheral devices.

Another modification of the first architecture is illustrated in Figure 2.7b, that includes two lock-step controllers as in Figure 2.7a, that are interconnected to achieve enhanced coverage.

Fault tolerance is guaranteed only for the tasks that are executed in parallel, and yields fault coverage of about 100%. The tasks can be checked internally in each controller by software self tests.



(a) *Lock-Step Dual Architecture* **(b)** *Dual Lock-Step Architecture*

Figure 2.7: *Architectures based on the Lock Step [BFM+03]*

2.1.3.2 Loosely-Synchronized Dual & Triple Modular Processor Architectures

In Loosely-Synchronized architecture two processors are connected to each other but are independent and have their own memory and flash parts. Another strategy is implemented here. Only a set of tasks marked as critical are duplicated, and each controller is responsible for checking the result of the other.

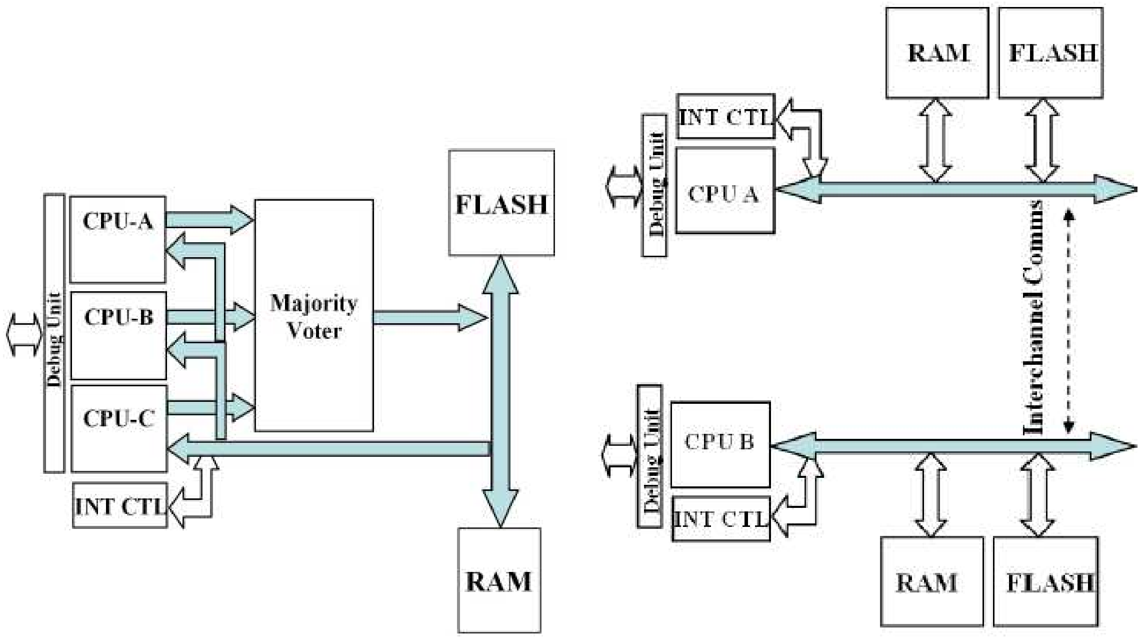
If a mismatch occurs, self-checking techniques can be involved to find the faulty part of the system.

The last architecture is illustrated in Figure 2.8b. Here, three controllers execute the same instructions and send the results to the Majority Voter, which decides, based on the results, whether the calculations are correct.

2.1.3.3 Generic Dual Core Architecture

Based on the previously described architectures, safety-related optimizations were proposed. An example is reconfigurable dual-core [KS06] [BFM+03] as visualized in the Figure 2.9. Kottke and Steininger [KS06] proposed a reconfigurable dual core architecture to handle demands on safety and computing power, efficiently using the dual core. In almost all architectures, the system works in master/checker mode for the tasks that are marked as critical, whereas the non-critical tasks are processed in normal mode.

The special feature of the reconfigurable dual-core system is that it can be switched



(a) *TMR Architecture* **(b)** *Loosely-Synchronized Dual Architecture*

Figure 2.8: *TMR and Loosely-Synchronized Dual Architecture [BFM+ 03]*

between two modes of operation:

1. Safety mode
2. Performance mode

For safety-critical applications, the safety mode operates in lock-step mode. The first core or the master controls the peripherals and the memory. The checker is not connected with the peripherals but it receives the same instructions and the results are compared with the master outputs.

To avoid common cause failures which can detected be in form of electromagnetic or any other external influence, the outputs of the master core are delayed for 1.5 clock cycles.

In safety mode, the same instruction stream is executed, and a kind of synchronization is needed in order to assure that the registers and cache are identical before it is switched from performance to safety mode. This can be done using an operating system. The main challenge is the cache synchronization because it is a non deterministic part. A possible solution is to flush all the cache tables before the mode switch is performed, or use a flag list which identifies which cache lines are valid in the safety mode.

In the performance mode, both cores are working independently as a dual core system, and the delay on the instructions is disabled. But these features can be used only in the safety and performance mode. An extra module is needed such as Memory Management Unit, which allows access to critical regions only in the supervisor mode.

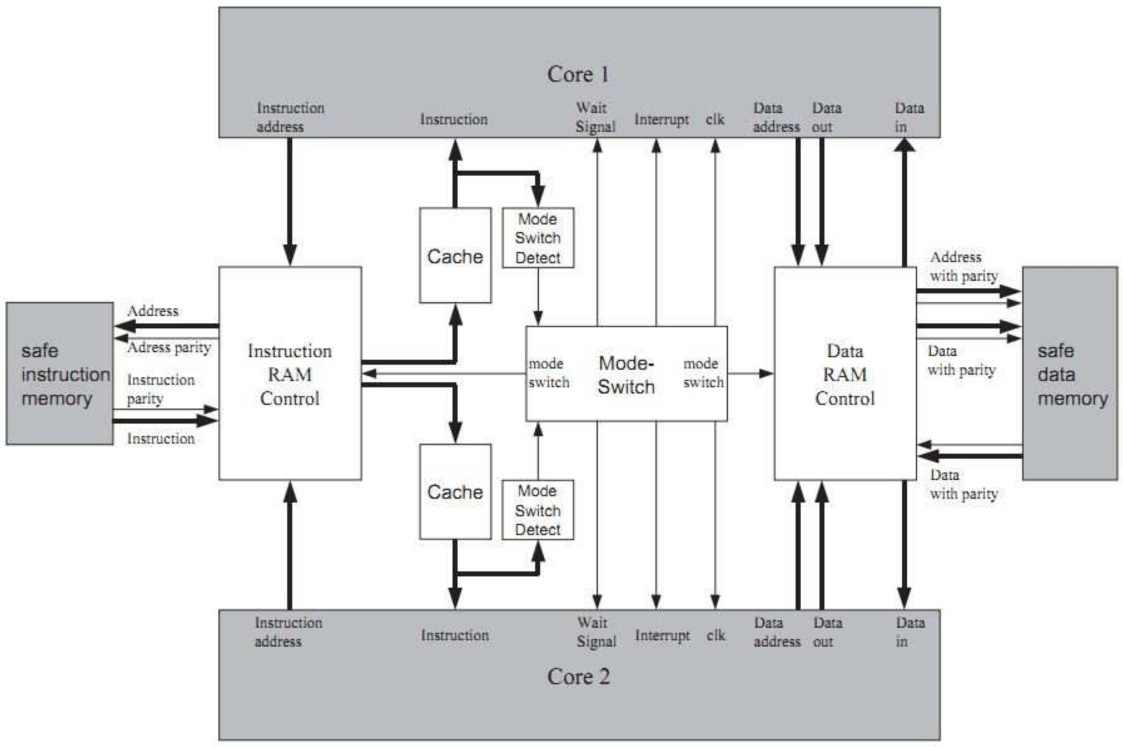


Figure 2.9: *Reconfigurable dual core architecture [KS06]*

The main memory is subdivided into the data and instruction memory, which are equipped with self-test routines to detect faults during during runtime. To prevent access to the same memory region, two special units are used to manage the RAM memory:

1. Instruction RAM control unit
2. Data RAM control unit

In the safety mode the instruction control unit serves exclusively the core which manages the peripherals. In the performance mode, a priority scheme is built to manage the access from both cores. At a specific address, one bit is used as identification for the operating mode.

2.2 Software strategies

2.2.1 Software Based Self tests

As multicore processor architectures becomes more popular, the time which is needed to test the cores scales depends on the number of cores. This prompts a challenge for the industry to consider new testing methods and integrate them into the microprocessor test flow. The purpose of the methods was to target the defective parts per million rate, which is a demand on quality product development. Such testing methods are known as functional self testing (instruction-based self testing) or commonly ”software-based self-testing” [WWC+05].

”The key idea of SBST is to exploit on chip programmable resources to run normal programs that test the processor itself” [WWC+05]. Some SBST tests, exploit the instruction set of the Architecture and performs safety critical checks in the background using the actual clock frequency.

Software-based self testing methods [PGSR10] are subdivided as illustrated in Figure 2.10 into two groups:

1. Functional methods
2. Structural methods

The first group exploits only the functional information about the processor, such as the Instruction Set Architecture during the test generation. Structural methods use the structural information of the architecture to generate the tests. The information can be either gate-level or RTL description.

This categorization is performed based on the type of processor description and not on a specific fault model.

Apart from this categorization in the group of functional tests, there are also tests, that concentrate on specific fault models. In this group are software analysis methods and algorithms like march, galloping, walking, checkerboard and butterfly, which are primarily used to test RAM memories.

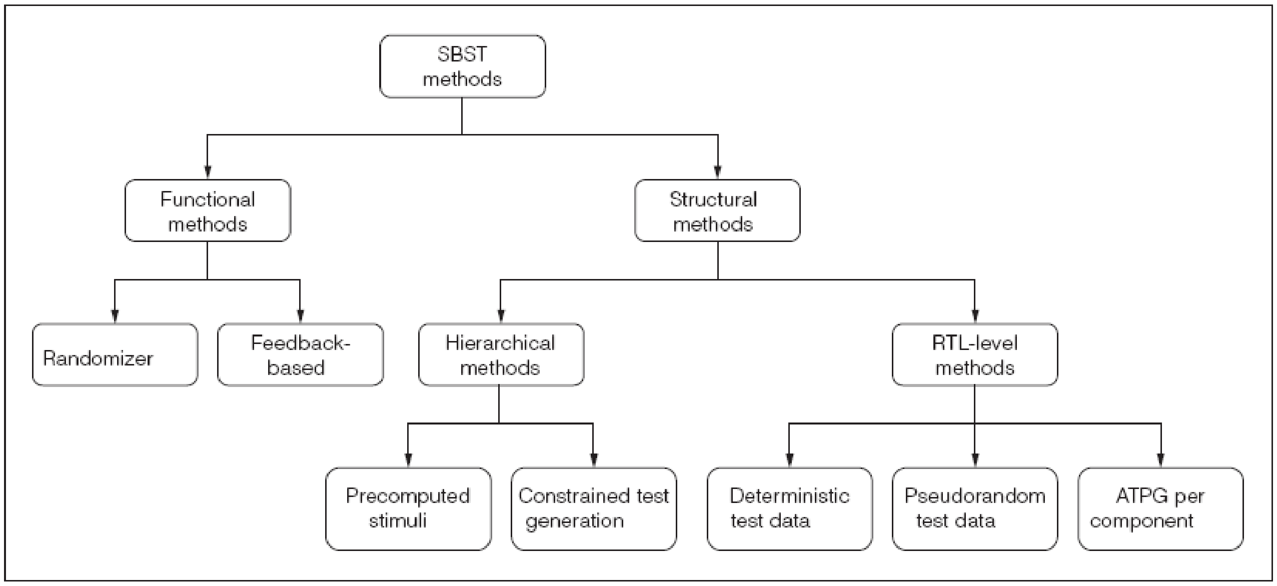


Figure 2.10: *Tree categorization of SBST methods [PGSR10]*

2.2.2 Functional Tests

Functional tests as illustrated in Figure 2.10 based on the logic they use are subdivided into two groups:

1. Randomized - generate tests with specific constraints during runtime
2. Feedback based - evaluate previous results to be considered for the new test generation during development time

Good representatives for the randomized tests are the methods defined in:

- VERTIS [SA98]
- FRITS [PML02]
- Load & GO [BHW06]

VERTIS uses the instruction set from the Programmers' Manual to extract the information about the functionality of the processor. This information is grouped and written in a predefined format that can be used by the generation tool. This file format can be found in [SA98]. The test generated from the tool is pseudo assembler and can be adopted easily for different architectures.

VERTIS can generate data randomly or as specified by the user which are used for different processor instructions. With the generated test sequences different parts of the system are tested from the functional view like: CPU, memory, fetching units etc.

By using this methodology of generating random tests, two processors were tested, and a fault coverage of about 94.04% for single stuck-at faults was achieved, which is a good result compared to the other techniques which have a smaller fault coverage as presented in Tables 2.2 and 2.3.

	HITEC	CRIS	VERTIS
FaultCoverage [%]	81.55	91.30	94.04
CPU time	119 min	180 min	3 min

Table 2.2: *VERTIS compared to the methods HITEC & CRIS for the Viper Processor [SA98]*

	HITEC	CRIS	VERTIS
FaultCoverage [%]	22.28	46.73	90.20
Efficiency [%]	24.89	47.64	90.20
CPU time	50.4 hrs	9.31 hrs	3.1 min

Table 2.3: *VERTIS compared to the methods HITEC & CRIS for the GL85 Processor [SA98]*

FRITS as a tool is used for microprocessors which have an extended instruction set. FRITS tests (kernels) are executed in real time as presented in execution flow Figure 2.11.

They do not produce any cache misses and therefore it is not necessary to control the address and bus cycles.

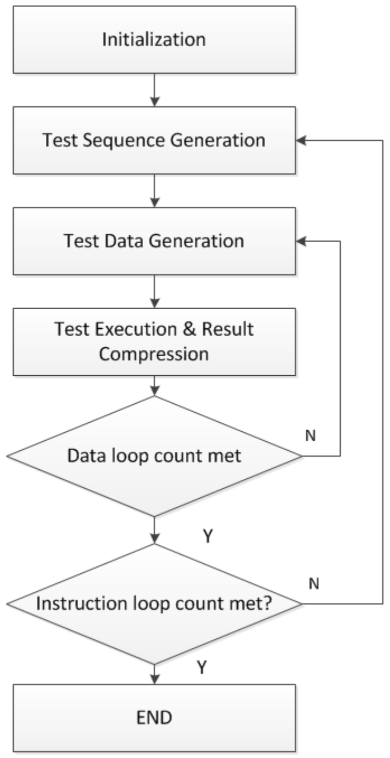


Figure 2.11: *Execution flow of the FRITS tool [PML02]*

These kernels have several constraints to be generated:

- Tests should not produce any bus cycle
- Instruction generation efficiency
- User-controlled instruction generation
- Code size, to be portable in the target on-board cache
- Debug possibility to detect failures

FRITS is used to generate tests for the Intel architectures, x86 ISA and Intel Itanium with a fault coverage of 70% and 85% to 90% for the second architecture.

The method Load & GO tool [BHW06] describes a process to insert the code into the cache and to start the program execution from there. To apply this method several problems must be solved:

- The program execution flow must hit every time in the cache
- Generation of the randomized tests

- To catch the results of the test (passed or failed)

For the proof of concept the UltraSPARC microprocessor was used, which shows that even with all simulations and test optimization this architecture lacks non-determinism of the flow.

The second category of functional tests is feedback-based [RSCS04], with the μGP architecture as illustrated in Figure 2.12.

The method generates a set of testing programs and optimizes them using the feedback information gained from the simulator. Afterwards, they are evaluated with different coverage metrics.

In the first step, the syntax of the target assembly language is encoded in a compact format, which can be used by the code generators. After that, a set of valid assembler programs are generated forming the initial seed (set). The μGP automatically optimizes the test set by tuning and modifying the assembly programs by exploiting the subroutines and software traps in the generated code.

Using this method two processors were tested, i8051 and Leon2[2]. With about 8.4 millions of simulated instructions, μGP generated a set with about 100% coverage evaluating five million instructions.

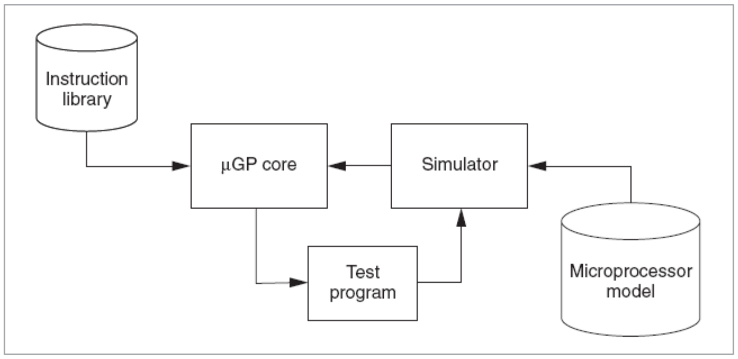


Figure 2.12: *Functional feedback tests realized with μGP architecture [RSCS04]*

2.2.2.1 Software Fail-Safe Techniques

In the group of SBST tests, many software fail-safe techniques [LCD+05] exists, but the most important are:

- Read/Write complementary data
- RAM test with checksums
- Redundant coding
- Program flow monitoring

[2]http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.gaisler.com/products/leon2/leon.html

Working with complementary data it is the first basic technique. Specific data is written in a part of the memory, the one's complement is calculated and stored in a separate part of the system. Checking whether a fault has occurred in the memory is done by the summation of these two values. The result should be zero. This checking can be done periodically or randomly based on the testing strategy. The main limitation of this technique is the RAM size, because in order to store every complementary value, a separate storage is needed.

RAM tests are performed to assure that data can be held and manipulated without errors. RAM tests can be done during system initialization, before the program is running, or during runtime, online periodically.

Checksums can be used to check the integrity of ROM, Flash, or EEPROM. For system values that do not change at runtime, the checksum can be calculated and stored in ROM. But checking the entire ROM needs many cycles, and that requires time. Testing the EEPROM is faster. An approach to reduce testing time is to check only the pieces that are marked as safety-critical. Checksums can find faults like memory errors, bit flips, and other data changes.

Redundant coding is a technique to implement, store, and run the same safety-critical code in different pieces of memory. For the same input, the result should be every time the same. It is a software protection method which needs extra implementation to assure that the code fragments do not have access to the same data and that the program itself runs correctly. The last technique is the program flow monitoring.

Program flow monitoring (PFM) is a technique to include a specific seed/key values program flow. With the help of these two values, it can be checked whether the program has executed all steps. These values can be inserted between function calls, or can be integrated into the program structure. Based on how the PFM values are included, the PFM can be implemented in the following ways:

1. Application-independent
2. Application-dependent
3. Time-dependent

The application-independent method updates the values between each function call. A disadvantage of this method is that the value can be updated without execution of the function. The advantage is that the PFM code can be reused without the modification across the application.

The application-dependent PFM is tightly integrated into the program execution. The update of the value is performed within the function, and assures that all functions are called.

The time-dependent PFM helps to verify that specific functions are called within a required timing window. This is accomplished by updating the values at specific times, during program execution.

2.2.2.2 March Tests

March tests [BBC+08], [Goo93] form a group of tests which exist in different variations that are simple and have linear complexity O(n).

The basic operation in all variations is called "march element". Every march element contains basic operations which are performed in a cell and then proceeds to the next cell:

- Writing 0 into the cell - w0
- Reading expected 0 from the cell - r0
- Writing 1 into the cell - w1
- Reading expected 1 from the cell - r1

After these operations are performed in a cell, the next cell is selected with increasing ($\Uparrow$) or decreasing($\Downarrow$) address order. The symbol ($\Updownarrow$) is used if the addressing order is not important.

Very common variations of March Algorithm are:

1. MATS+ described with the formula 2.1
2. March C- described with the formula 2.2
3. March B described with the formula 2.3
4. March G described with the formula 2.4

with the following mathematical representation:

$$\Updownarrow (w0) \Uparrow (r0, w1) \Downarrow (r1, w0) \tag{2.1}$$

$$\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0) \tag{2.2}$$

$$\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0) \tag{2.3}$$

$$\begin{aligned}\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0);\\ (DELAY; \Updownarrow (r0, w1, r1); DELAY; \Updownarrow (r1, w0, r0))\end{aligned} \tag{2.4}$$

In order to detect the AF 2.1.1 faults, fulfilling the following conditions as illustrated in Table 2.4 is needed, which includes at least two march elements and starts with a different addressing order.

The MATS+ test finds all address decoder faults(AF) 2.1.1 because of the mathematical structure and it fulfils the condition in the Table 2.4. Stuck at faults (SAF) are detected because in the Equation 2.1, the values 0 and 1 are read from a cell. The number of operations to complete this test is 5n[3].

[3]"n" is the number of cells in a memory

Condition	March element
1	$\Uparrow (rx, ...w\overline{x})$
2	$\Downarrow (rx, ...w\overline{x})$

Table 2.4: *Two conditions for a test to detect the AF Faults*

The MarchC- test is an extension of the MarchC, which has linear complexity O(n) and requires 10n operations to complete the test. Three groups of faults are targeted with this test: AF because it satisfies the conditions in Table 2.4, SAF because in the equation 2.2, it reads 1 and 0 from the cells and all CFin and CFSsts faults are detected due to the successively read and write operations.

The MarchB test which requires 17n operations, targets and detects the following faults: all AFs, SAFs, TFs, CFins and linked CFids, described in Section 2.1.1.

Condition	March element
1	$..., rx, ...r\overline{x}$
2	$..., r\overline{x}, ...rx$

Table 2.5: *Two conditions for a test to detect the COF Faults*

A COF Fault is present if a specific RAM cell cannot be accessed. To ensure that every cell is accessible, it must be guaranteed that 0 and 1 is read from the cell as illustrated in Table 2.5. To extend a march test to allow the detection of DRF faults as well, it must contain delays between writing a value "x" and reading it back.

The MarchG test is an extension of the MarchB test which includes these two extensions and specifically detects COF and DRF faults. To complete the test it requires 23n + 2Del[4] operations.

2.2.2.3 Abraham Test

Another functional test for RAM memory is the Abraham Test. In order to apply the test, there need not be structural information about the processor and covers specifically stuck at faults and coupled faults. It not only covers these faults but it covers the faults that occur in different parts of the RAM:

1. Memory Cell Array,
2. Read/Write Logic and
3. Decoder Logic

[4]Delayed operations

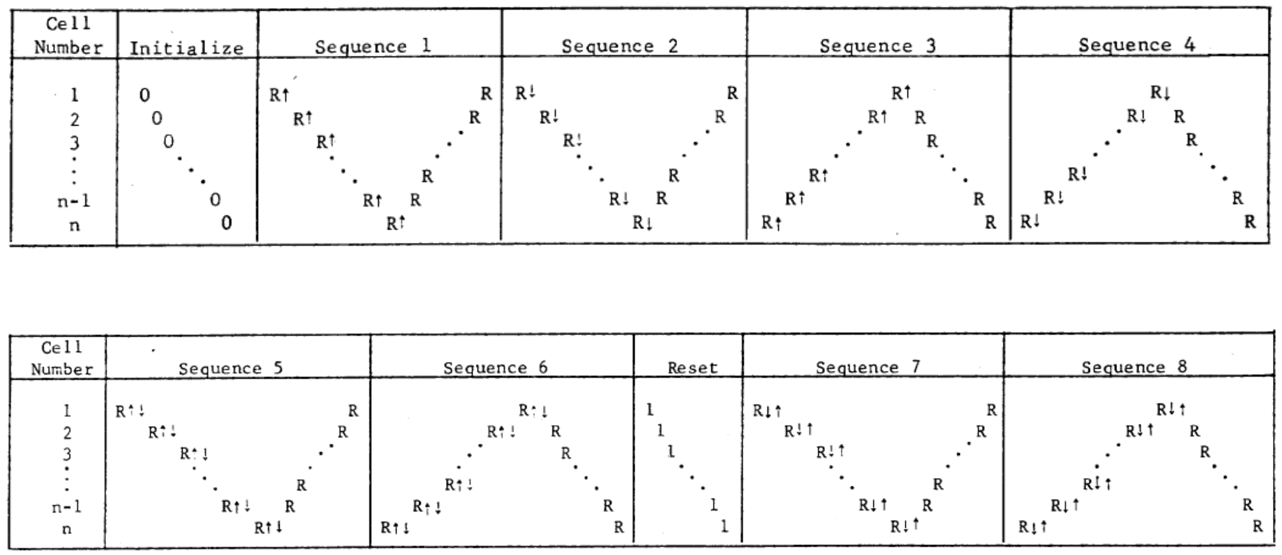


Figure 2.13: *Abraham Test [NTA78]*

The test consists of three conditions which are necessary to be fulfilled:

1. Every memory cell must make these forced transitions:
   - A 0-1 Transition
   - A 1-0 Transition

   and are read after each transition, before any other operation is made.

2. For each pair of cells (i, j) cell "i" must be read before the cell "j" makes a forced transition, and before the cells "i" and "j" make any next forced transition for the following states "i" and forced transitions "j":
   - Cell "i" in state 0, cell "j" makes a 0-1 transition
   - Cell "i" in state 1, cell "j" makes a 0-1 transition
   - Cell "i" in state 0, cell "j" makes a 1-0 transition
   - Cell "i" in state 1, cell "j" makes a 1-0 transition

3. If the cell "j" makes a transition of y in $\overline{y}$ after the cell "i" has made a transition from x to $\overline{x}$ and before the cell k in state z is read, for every triplet i,j,k, the algorithm must fulfil the following conditions with $[x, y, z \in 0, 1]$:
   - Cell k is read in state z, after the cell i makes a transition from x to $\overline{x}$ and before the cell j makes a transition from y to $\overline{y}$.
   - Cell k is read in state z, after the cell j makes a transition from y to $\overline{y}$ and before the cell i makes a transition from x to $\overline{x}$.

The test is illustrated in Figure 2.13 and the symbols inside have the following meaning:

- $\downarrow$ - Forced transition from 1 to 0
- $\uparrow$ - Forced transition from 0 to 1
- R - Read RAM cell

By these steps, it is ensured that all coupled faults are detected. The sequences 1,2,5 and 7 go through the RAM with increasing memory address, and the sequences are 3,4,6,8 with decreasing address.

In the worst case the algorithm performs 30n operations in which "n" is the number of RAM cells, which means that the algorithm has a linear complexity O(n).

2.2.2.4 Galloping and Walking patterns

Next two [WWW06] other common algorithms which are used for RAM testing are described:

1. Galloping (ping-pong) pattern (GALPAT)
2. Walking pattern (WALPAT)

Algorithm 1 GALPAT Algorithm

```
$Initialize : RamCells[n] = 0, step = 1$
**while** $step <= 2$ **do**
  **for** $i = 0 \to n-1$ **do**
    $\overline{RamCell[i]}$
    **for** $j = 0, j! = i \to n-1$ **do**
      $VerifyCell[i]$
      $VerifyCell[j]$
      $j \Leftarrow j+1$
    **end for**
    $\overline{RamCell[i]}$
    $i \Leftarrow i+1$
  **end for**
  $Initialize : RamCells[n] = 1$
  $step = step + 1$
**end while**
```

The Galloping pattern algorithm has quadratic runtime complexity of $O(4n^2)$ and detects faults like SAF, AF, TF and CFs but it is not recommended for large RAM memories because of its quadratic runtime complexity. The complete algorithm is illustrated in the Algorithm 1.

The algorithm begins with initializing the complete RAM cells to 0 and complements the first "i" cell. In the next step, it performs a read of the current cell and all other cells (VerifyCell[i], VerifyCell[j]). Then it complements the "i" cell again and increments "i" until it reaches n. Afterwards, it performs the same step but now initializing all cells to 1.

Algorithm 2 WALPAT Algorithm

```
$Initialize : RamCells[n] = 0, step = 1$
**while** $step <= 2$ **do**
  **for** $i = 0 \to n - 1$ **do**
    $\overline{RamCell[i]}$
    **for** $j = 0, j! = i \to n - 1$ **do**
      $VerifyCell[i]$
      $j \Leftarrow j + 1$
    **end for**
    $\overline{VerifyCell[j]}$
    $\overline{RamCell[i]}$
    $i \Leftarrow i + 1$
  **end for**
  $Initialize : RamCells[n] = 1$
  $step = step + 1$
**end while**
```

The other alternative is the WALPAT, which is a modification of the GALPAT with complexity of $O(2n^2)$. The only difference is that after complementing the first "i" cell it reads the basic cell after all other cells are read.

Algorithm 3 Checkerboard Algorithm

```
**for** $i = 0 \to N - 1$ **do**
  $RamCell[i] = 0$
  $RamCell[i + 1] = 1$
  $i \Leftarrow i + 2$
**end for**
**for** $i = 1 \to N - 1$ **do**
  $ReadCell[i]$
**end for**
**for** $i = 0 \to N - 1$ **do**
  $RamCell[i] = 1$
  $RamCell[i + 1] = 0$
  $i \Leftarrow i + 2$
**end for**
**for** $i = 1 \to N - 1$ **do**
  $ReadCell[i]$
**end for**
```

2.2.2.5 Checkerboard

The checkerboard [WWW06] test places the bit patterns like the structure of the checkerboard game. It has O(n) complexity with 4n operations. The test begins with writing the first pattern to even or odd cells and all neighboring cells are set to a different value. After that, a read back is performed. The same procedure is applied with a different pattern.

It is mainly used just to activate different faults, and a delay is included between read and write operations. It targets but does not detect all AF, SAF, TF and CF faults.

2.2.2.6 Butterfly Algorithm

Algorithm 4 Butterfly Algorithm

```
$"maxdist < 0.5 * col/rowlength"$
$Initialize : RamCells[n] = 0, step = 1$
while $step <= 2$ do
  for $i = 0 \to n - 1$ do
    $\overline{RamCell[i]}$
    $dist = 1, j = i$
    while $dist <= maxdist$ do
      $Read[i][j - 1]$
      $Read[i - 1][j]$
      $Read[i][j + 1]$
      $Read[i + 1][j]$
      $Read[i][j]$
      $dist* = 2$
    end while
    $i \Leftarrow i + 1$
  end for
  $\overline{RamCell[i]}$
  $Initialize : RamCells[n] = 1$
  $step = 2$
end while
```

This test [WWW06] is a modified version of GALPAT, with the purpose to find only AFs and SAFs. The time complexity is $\mathbf{5n} * \log \mathbf{n}$. In the first step, all cells are initialized at zero. As illustrated in the Algorithm 4 the read operations are performed in butterfly form. These operations are performed twice with the initialization of the cells at one.

2.2.3 Structural Tests

Basically all structural tests [PGK+01] use RTL description or gate-level information about the controller. Based on the test tree 2.10 and how this information is used, approaches are subdivided into:

- Hierarchical tests
- RTL-level tests

Hierarchical tests concentrate on specific controller modules of a processor core. Test vectors are generated for every module and converted into instruction sequences that can be applied to the controller. The "pre-computed test sets" and "constrained test generation" are part of this group.

"Pre-computed test sets" denotes a group of methods that generates stimuli sets for every module of the controller under consideration, and then exploits the functional mechanism to give the module the precomputed stimuli and to propagate the results to observable locations.

A representative is a method [GVA06] which generates test sequences with the main focus on functional faults that are difficult to detect. It involves controllability and observability properties with Boolean differences and a bounded model checker.

To simulate faults difficult to detect, pseudo random instructions are generated with 36579 instructions for the OpenRISC 1200 processor[5] and using an available commercial tool. The fault coverage is evaluated which has a result of about 68% for stuck-at faults.

The list of undetected faults from the first step, is the basis for next step. A selection process is performed to group the instructions based on the modules, and then, the commercial ATPG tool generates for each module new test sequences. The observability properties are guaranteed with the expression of propagation requirements as Boolean differences.

The model checker evaluates the sequences to see which one propagates the effects of the observed fault to the primary output, and finally only these sequences remain. The complete process on the OpenRISC 1200 processor increases the fault coverage by about 14% to overall 82% for stuck-at faults.

Constrained test generation describes the modules on the processor with different abstraction levels. The module under consideration is described at RTL level, whereas the other modules of the system are described at a higher level. With the detailed description of only the module under consideration, the ATPG tool has a lower circuit complexity than the original one.

Part of this group is the method proposed by Chen [CRRD03], which divides a given processor up into different levels of modules, generates templates sets, and, using the controllability and observability properties, selects the most suitable templates. A constrained test generation is performed with selected templates and, for specific module, and at the end, the module test patterns are translated into instruction test sequences. Using this method for a RISC processor, a fault coverage of 95% was achieved for common faults like stuck-at faults, but it can be applied to other fault models like bridging faults and transistor-level faults.

SBST methods that use structural information (RTL) during the process of tests generation are part of the RTL structural test branch as illustrated in Figure 2.10. Based on the information which is used to generate test vectors, they are subdivided into:

- Deterministic algorithms
- ATPG Algorithms
- Pseudorandom methods

Deterministic algorithms use information about a specific function that a controller performs.

Kranitis [KPGX05] proposed a component-based divide and conquer approach that uses information about the ISA and RTL description. In the first step, components with

[5]http://www.opencores.org/openrisc,or1200

their operations are identified, next, they are classified and prioritized, and, finally, deterministic routines are generated. This approach was used for two processors, and the following results were observed; Plasma achieved fault coverage of 95% and MIPS R3000 achieved 95% total fault coverage.

Another method that is part of this group was proposed by Paschalis [PGH$^+$06]. It efficiently tests the processor data path without changes in the structure of the processor and can be applied to any word length and different internal architectures.

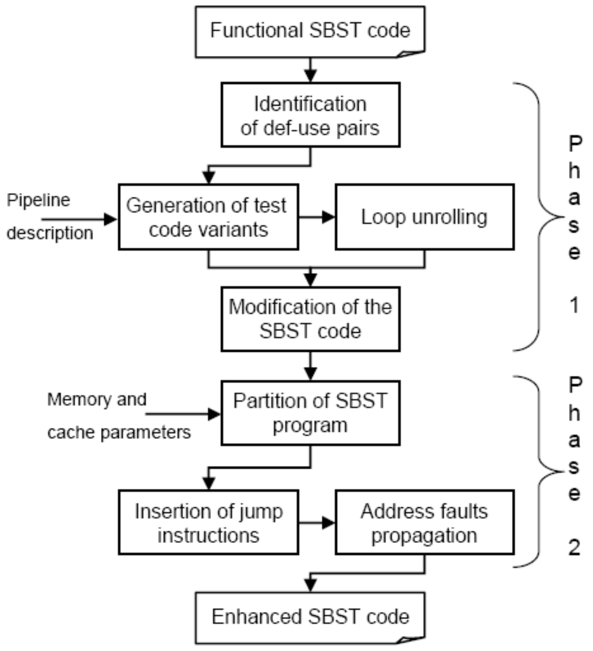

Functional SBST code
Identification of def-use pairs
Pipeline description
Generation of test code variants
Loop unrolling
Modification of the SBST code
Phase 1
Memory and cache parameters
Partition of SBST program
Insertion of jump instructions
Address faults propagation
Phase 2
Enhanced SBST code

Figure 2.14: *SBST methodology for pipelined processors [PGH$^+$06]*

The methodology, illustrated in Figure 2.14, is divided up into two phases: Phase 1 & 2. It takes existing SBST programs and different processor pipeline parameters to make a list of programs that achieve high fault coverage. Phase 1 deals with identifying def-use[6] pairs, code variants generation and optimization including removing dependencies within a loop with "loop unrolling".

The output is a modified SBST code, which is used in the second phase. Here, the code is partitioned with respect to virtual memory, size of test program, and memory parameters. Jump instructions are included in the end of every module to guarantee the propagation of address related faults.

The methodology was simulated with two pipeline processors: miniMIPS[7] and OpenRISC 1200[8]. From an average fault tolerance of 82.81%, an improvement of 12.34 % was achieved, increasing the overall stuck-at fault coverage to 93.03%.

[6]def-use analysis is used for optimization in compilers; a variable's value is "defined" when an assignment is made to it and is "used" when it appears on the right side of an assignment

[7]http://opencores.org/project,minimips

[8]http://opencores.org/project,or1k

The method proposed by Chen [CWLG07] which generates test stimuli using the gate level of the processor description as entry point, is part of the ATPG Algorithms group .

The proposed method consists of multiple levels of abstraction, which are input to the test development based on the information about the processor architecture, register transfer-level and gate-level. To apply this method, the processor core must be subdivided into different parts: ISA registers, IP Units, control, steering logic, pipeline registers, and hazard related logic.

After the classification, test routines are generated for every module. In the ISA Registers, the development is focused on the structural faults of D Flip Flops. For the identified IPs, control & steering logic, and pipeline related logic, an ATPG tool generates the test patterns focused on the ISA specification, RTL descriptions, and pipeline architecture of the target processor.

The methodology was tested in a processor core with an ARMv4 instruction set which achieves a fault coverage of 93.74 % for stuck-at faults related to pipeline registers, ALU, Decoder, memory access unit etc.

Pseudo-random methods generate data together with testing instructions and let the processor perform the evaluation. Such a method is presented by a group of researchers [KLC$^+$02]. It generates tests under the constraints of the ISA to avoid unwanted test patterns. It is targeting structural faults like stuck-at and delay faults.

The method consists of two steps: test preparation and self-testing. Using a special software program, the tests which are generated are deliverable test patterns. The results are stored in the memory, and a selection is made using the constraints of the instruction set of the processor.

A component level fault simulation is performed to evaluate the test patterns. After evaluating the tests, evaluated they are tested on-chip.

Using this method, a test was performed on Parwan and DLX processors. For the Parwan processor it took 5.3 instructions on average for a test vector with a fault coverage of 99.8%, while in the DLX processors 5.9 instructions with fault coverage of 96.3% for stuck-at and delay faults.

2.2.4 On-line Periodic Tests

Online periodic tests are performed during the normal operation of a processor. These tests reside in the RAM or Flash and are called by the operating system as normal programs.

Gizopoulos [PG04] proposed a SBST methodology to classify the processor components and characteristics of SBST test programs to be suitable for online periodic tests. The methodology consists of three phases.

In the first phase, information is extracted from the ISA and the low register transfer level. The component operations are identified with specific input and outputs that perform different operations, which includes multi-cycle data-paths or pipeline register.

In the second phase, different processor components are selected based on the same properties and component prioritization to generate test patterns, which will be transformed to a test routine.

Based on how the different components are visible for the programmer, they can be characterized as follows:

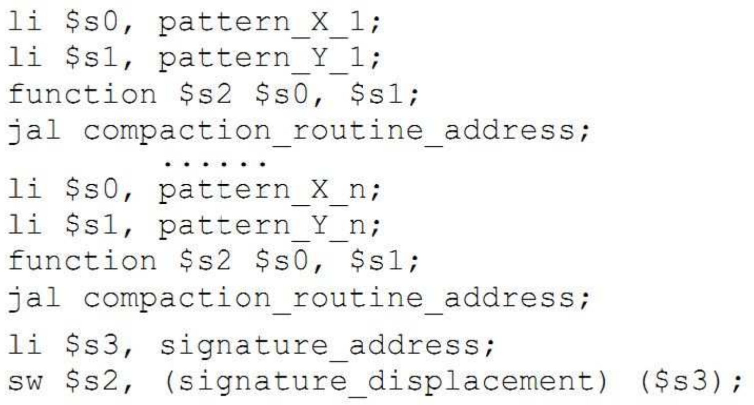
1. Visible components
2. Data-visible components
3. Address-visible components
4. Mixed (data & address) visible components
5. Partially visible components
6. Hidden components

Visible components are parts whose inputs and outputs are accessible from the assembler language programmers. Data-visible components are components like ALU, multipliers, dividers data registers etc. which serve as storage for input data test patterns. The output data can be stored at register file, data memory, or both of them.

Address-visible components are components whose inputs and the outputs, receive addresses of the memory system. These appear in the instruction fetch unit or data-memory controller.

The mixed visible components use a mixed type of the inputs and outputs of the visible and data visible components such as the adder used with the relative addressing. Partially visible components are the components which generate control signals and are implemented as finite state machines. One such component is the processor control unit which affects the visible components. They have a medium testability.

The hidden components are architecture components which are included for performance and are not visible to assembly programming language. Such components are pipeline control units, branch prediction mechanisms etc.



```
li $s0, pattern_X_1;
li $s1, pattern_Y_1;
function $s2 $s0, $s1;
jal compaction_routine_address;
            ......
li $s0, pattern_X_n;
li $s1, pattern_Y_n;
function $s2 $s0, $s1;
jal compaction_routine_address;
li $s3, signature_address;
sw $s2, (signature_displacement) ($s3);
```

Figure 2.15: *Test routine generated according to the methodology [PG04]*

The development of the test routines is performed in the third phase based on the different TPG strategies. Such a test routine is illustrated in Figure 2.15.

A 32 bit RISC processor[9] called Plasma was used to demonstrate the proposed methodology, which achieves a fault coverage of 95.6 % for stuck at faults.

For the project SafetyLoan [TP07] several SBST tests were developed for an ARM7 microcontroller. To test the RAM memory, the walking pattern algorithm is used. It is optimized in a manner that read/write operations are performed in block segments of 32 bytes, by a single command using multiple load and store. This reduces the number of operations to be performed in the RAM memory.

For the data integrity, cyclic redundancy checks with 16 bit are performed, while for register testing, the galloping pattern is used. The galloping pattern is described in Section 2.2.2.4.

The main problem with the register testing is that during the test runs, the values are stored in the memory. A possible solution to the problem is to swap the content between the registers while the tests are running.

Testing the arithmetic logic unit is done by subdividing the instruction set into command classes, which use the opcode as classifying parameter. The entire RAM was tested within 0.95 s, while ALU & register tests had a total execution time of 4.26 ms with a high diagnostic coverage.

2.2.5 Specialized Tests for Multi-cores

The research of last the years is going towards multi-core architectures for better performance and exploiting parallelism for instruction execution. These advances bring new challenges for SBST techniques which must be adopted for multi-core architectures and can be defined as [PGSR10]:

1. Use of SBST techniques that are proven for single core architectures to test all cores individually
2. Speed up self testing routines with core and thread parallelism
3. Testing of interoperability logic at cores and threads levels

In two recent papers based on these challenges, Apostolakis attempts to reduce application time [APG^{+}09] and tries to exploit the core and thread parallelism [AGPP09].

The proposed approach [APG^{+}09] is used to transfer the SBST test from uni-core to symmetric multiprocessor architectures (SMP). The complete algorithm is very complex and reduces time overheads caused by data cache and bus. The test consists of different steps which are performed to start the test. In the first step, data and test code is loaded directly in the cache. Secondly, every CPU connected to the matrix executes this code at its actual clock speed. In the third step, the results are uploaded to a low-cost tester and are checked from an external device. The complete structure is illustrated in Figure 2.16.

The main objective of the method was to improve the SBST test routines for the OpenSPARC T1 Processor, which has eight cores.

[9]MIPS architecture

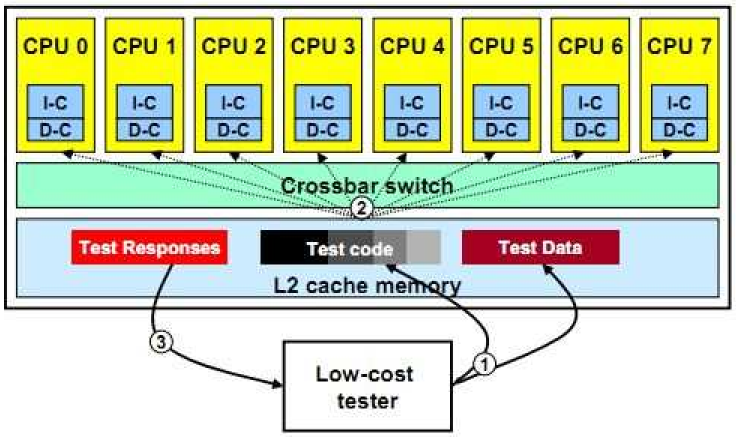


Figure 2.16: *Functional tests with multicore architectures [APG+ 09]*

After a long analysis and simulation the following two equations were derived for maximum speed-up:

$$S_{max} = \frac{idle\ \ time}{run\ \ time + 1}, or \tag{2.5}$$

$$S_{max} = \frac{memory\ \ latency + pipeline\ \ latency}{run\ \ time + 1}$$

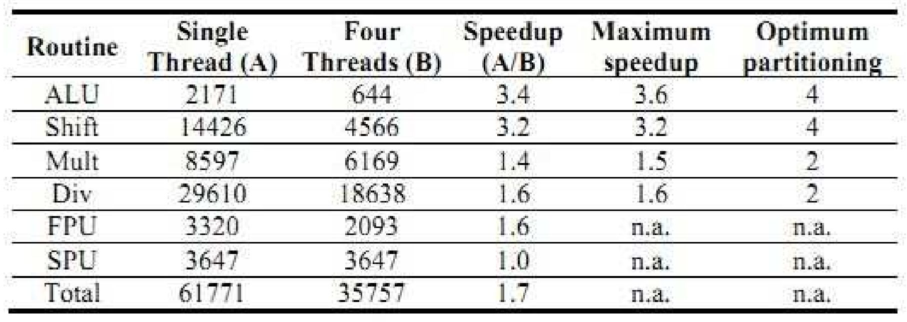
$$S_{max} = \frac{100 - long\ \ latency\ \ instruction\ \ time}{long\ \ latency\ \ instruction\ \ time} + 1 \tag{2.6}$$

Equation 2.6 is for ALU and shift test routines while the Equation 2.6 is for Mult and Div operations because they spend much time on long latency instructions. With the help of these two equations, two rules are formulated that help to transform single thread self-tests into a multi-threaded version:

1. Do not use more threads that execute the same self test routine concurrently than the optimum calculated by expressions 2.6 and 2.6.
2. The number of self routines that have high L1 miss rate must be reduced, since the memory waiting intervals will not be overlapped

The methodology can be described with the following steps:

1. Analyze the performance of the test routines
2. Calculate the total execution time of the single threaded version and optimum total execution time of multithreaded version
3. If the execution time of the single-threaded routine is longer, it must be split into shorter routines, and group these routines in a set



Routine	Single Thread (A)	Four Threads (B)	Speedup (A/B)	Maximum speedup	Optimum partitioning
ALU	2171	644	3.4	3.6	4
Shift	14426	4566	3.2	3.2	4
Mult	8597	6169	1.4	1.5	2
Div	29610	18638	1.6	1.6	2
FPU	3320	2093	1.6	n.a.	n.a.
SPU	3647	3647	1.0	n.a.	n.a.
Total	61771	35757	1.7	n.a.	n.a.

Figure 2.17: *Performance results of multi-threaded over single-threaded [APG+09]*

4. Select the longest routine from the set of self routines, assigns to it the smallest execution time, which is the sum of all time single threaded routines assigned to the thread until now
   - If this calculation contradicts one of two rules, give the routine the next shortest execution time
   - Remove this routine from the set
5. If the set is not empty, go to step 3, or exit.

Using different set-ups this method achieves a speed-up of 3.3 of the execution time, compared to single-threaded and straightforward multi-threaded methodologies. The complete results are illustrated in Table 2.17

The TLP[10] method [AGPP09] splits the self test routines into shorter ones. Then it assigns the new routines to hardware threads of the core, thus increasing parallel execution and reducing the idle intervals of the core. Idle intervals can occur due to cache misses or other latency's.

The complete strategy consists of three phases:

1. Allocation of the test code in the shared cache
2. Allocation of the test responses in the shared cache
3. Efficient SBST scheduling between the cores

To evaluate the methodology different benchmarks with dual-core, quad-core and octal-core were used. The benchmarks are based on the OpenRISC 1200 processor, which is used in numerous research activities for SBST tests, and in this case, the fault evaluation results for the three benchmarks are about 90 % total stuck-at fault coverage.

[10]Thread-level parallelism

Chapter 3

Analysis and Design

3.1 Evaluated Architectures

During the first steps of the thesis, different multi-core architectures were evaluated. Obviously, there are many different multi-core microcontrollers which are intended to be used in safety-related systems and many of them are designed to fulfil the IEC 61508 SIL-3 certification criteria. But not all of them are suitable to be used in this area because they lack of safety features and are vulnerable to common cause failures.

According to the safety features, properties and the architectures, three microcontrollers were chosen to be evaluated:

1. Texas Instruments TMS570
2. Infineon TC1766 (Audo-NextGeneration)
3. Toshiba/Yogitech fRMethodology (Fault Robust Methodology)

In the Sections 2.1.2 and 2.1.3 different controller strategies are discussed, and a short description about possible implementations is given. Now, architectures are examined from the industrial perspective to see how these concepts are used in practice. In the following, the three architectures are described.

3.1.1 TMS570 - Texas Instruments

Referring to Figure 3.1, the controller has two cores connected in Lock - Step Mode[1]. It uses two *ARM*® Cortex-R4F cores, with floating point unit to meet the SIL-3/ASIL-D level according to the IEC-61508 and ISO-26262.

Target applications for the microcontroller are:

- Electronic Power Steering
- Active Steering
- Integrated Chassis Management

[1]Described in Page 21 with Figure 2.7a

- Braking and Stability Control
- Dynamic Damping and Driver Assistance

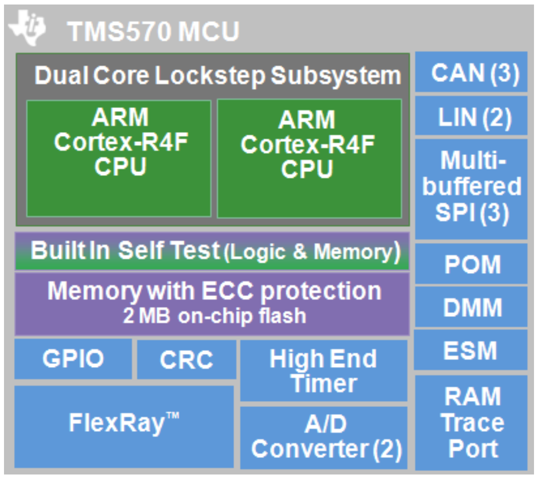


Figure 3.1: *The TMS570 microcontroller from Texas Instruments*

The main safety features on the TI Controller are:

1. Guard Ring between the two cores
2. Cycle delays on both CPU's
3. On-line hardware checker of the outputs of the CPU's
4. Memory Protection Unit protects up to eight regions in the memory space
5. ECC calculation for every RAM read/write access
6. Separate address spaces for flash & RAM to store ECC data
7. Parity check for all communication peripherals
8. Safety tests for cores, memory and bus

The guard ring (physical separation) isolates the two cores from each other. The cycle delays as illustrated in Figure 3.2 and the 180 ° rotation of one core, isolates the influence from one core to the other and increases the robustness against common cause failures.

The memory protection unit includes different levels of permissions which can be used to protect the eight regions to be read, written or executed either by a user or the privileged mode. This assures that a user program can not modify registers which are protected and can be used only by the operating system or by using supervisor calls.

All communication peripherals and the address bus are protected by parity check during runtime. Additionally, safety tests can be applied at the start-up to the peripherals, memory, and the cores to check the integrity of these components.

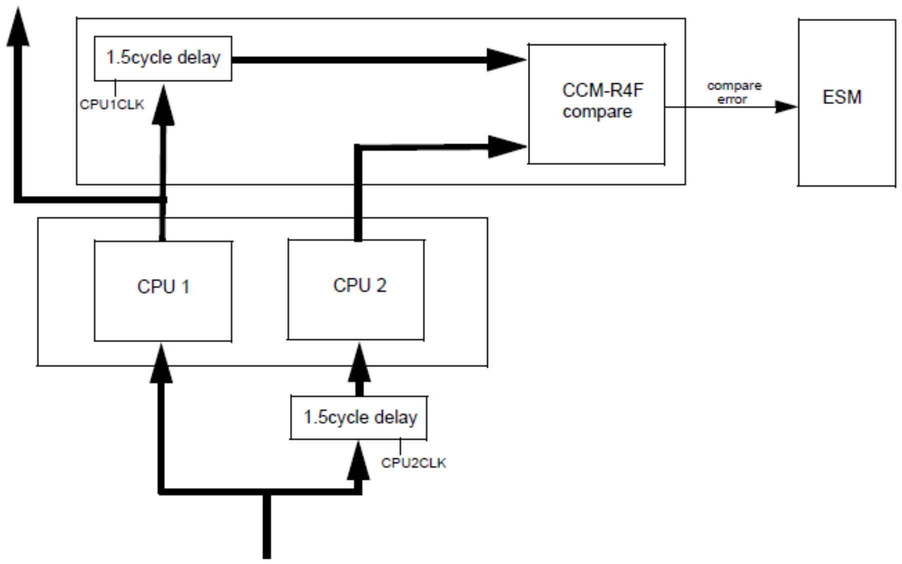


Figure 3.2: *Comparing CPU instruction results [TMS]*

A physical separation of data regions exists in the architecture. For the SRAM section, a SRAM-ECC section exists, in which the ECC of the SRAM contents is stored. The same principle is applied to the Flash section, but here a mirrored physical section of Flash & Flash-ECC exists, which assures better integrity of the data. For protection against mutual influences, a physical, unused reserved space is included between the address spaces of the SRAM, ECC, Flash and the mirrored address spaces.

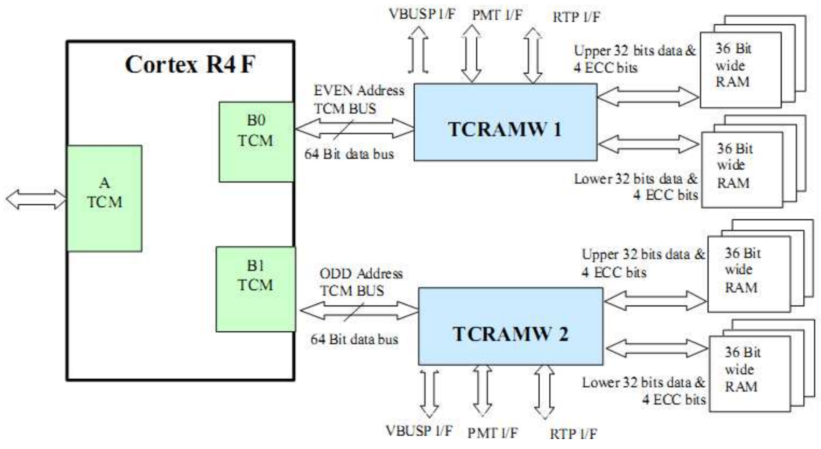


Figure 3.3: *Memory read/write access with ECC calculation [TMS]*

In Figure 3.3, the connection between CPU and the RAM banks is presented. Inside the Cortex-R4F core for every read/write access on the RAM, the ECC data is calculated and stored. The core has two other interfaces which are used to access even or odd memory addresses.

3.1.2 TC1766 - Infineon Technologies

The architecture [BSE07] from Infineon Technologies has two hardware-decoupled 32 bit cores on the same silicon die, the first is from the family of TriCore[2] and the second is a PCP(Peripheral Control Processor) core with the following characteristics:

- Connected to the TriCore with a bandwidth about 1,6 GB/sec
- Compared to the TriCore, it uses other register file, pipeline, and physical memories
- Software is created using a different compiler

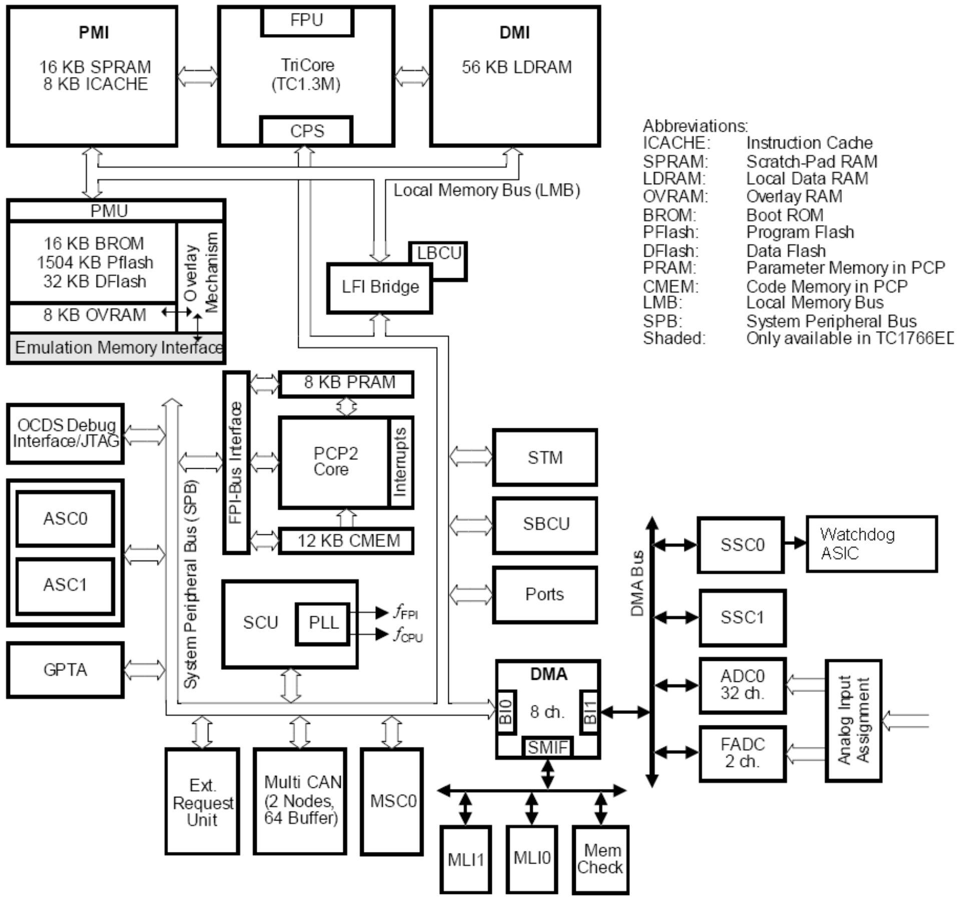


Figure 3.4: *The TC1766 microcontroller from Infineon Technologies [BSE07]*

The architecture is illustrated in Figure 3.4. With the monolithic design, it is not robust against common cause failures because it uses the same clock source and power

[2]http://www.infineon.com

supply for both cores. As a protection against those failures, an ASIC watchdog monitors the clock and power supply, and the cores use separate RAM memories for local variables.

Major components are:

1. Computational Unit - TriCore
2. Monitoring Unit - PCP Core
3. Safety unit for common cause failures - Watchdog ASIC

In the computational unit, the TriCore runs the application, answers the requests of the monitoring unit, and checks the operation of safety-critical parts. The TriCore runs the tests during normal operation, start-up and shut-down and is equipped with a floating point unit, dual multiply accumulator, and a DSP instruction set.

It includes different hardware safety features such as parity protection of SRAMs, ECC protection of flash, memory protection tables, internal backup oscillator and a DMA controller with memory protection.

The PCP core is a RISC controller and is based on the asymmetric strategy that is presented in Section 2.1.2.4. It runs the checking application to monitor the TriCore main controller.

It sends periodical requests to the TriCore, which can intentionally be correct or incorrect to ensure that the TriCore calculation unit works correctly. Intentionally, incorrect means that the responses to the test vectors are checked for equality and inequality with true and false data. Based on the safety strategy, if too many incorrect results are returned, the PCP core can shut-down the system to enter a safe state.

Monitoring the requests can occur in two ways:

- Question/Answer method
- Using shared structures

For the first method, the PCP generates different opcode tests which cover all registers, different instruction types[3] and the stack. The PCP transmits every generated test to the computational unit and compares the result to the pre-calculated value stored in the local RAM.

For the second method, the results are written in the shared structure. The PCP monitors that the test are performed based on different parameters, like test counter increments etc.

Furthermore, the ASIC watchdog monitors the power supply and clock speed to avoid common cause failures.

3.1.3 fRMethodology - Yogitech & Toshiba

The FMEA methodology described in [BCM07], is a new approach that is used to design a controller based on the IEC 61508 safety standard. The methodology subdivides the SoC into "critical zones", which are elementary failure points, and then extracts the information about RTL Logic, which is used to build the SRS[4] document about the system.

[3]Memory, arithmetic, logic and control flow operations

[4]Safety Requirements Specification

The information is extracted automatically from the controller [MKS10] by a tool, in the next step, the information is grouped in a database and then failure rates of the "critical zones" are calculated.

Based on this statistical information two important factors[5] are computed:

1. Safe Failure Fraction - SFF and
2. Diagnostic Coverage - DC

which are required by the IEC61508 to define the rates for the Safe Failure Fraction and Diagnostic Coverage.

$$\mathbf{SFF} = \frac{\sum \lambda_S + \sum \lambda_{DD}}{\sum \lambda_S + \sum \lambda_D} \tag{3.1}$$

$$\mathbf{DC} = \frac{\sum \lambda_{DD}}{\sum \lambda_D} \tag{3.2}$$

The elements in the equations represent the following failure rates:

- λ_S - rate of safe failures
- λ_D - rate of dangerous failures
- λ_{DD} - rate of dangerous failures that are detected

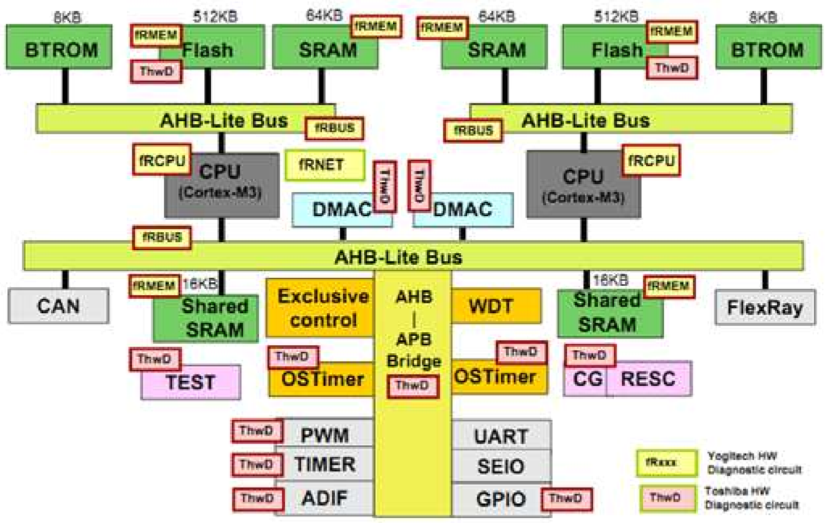


Figure 3.5: *The fRM architecture [BCM07]*

[5]http://www.ntnu.no/ross/slides/chapt10.pdf

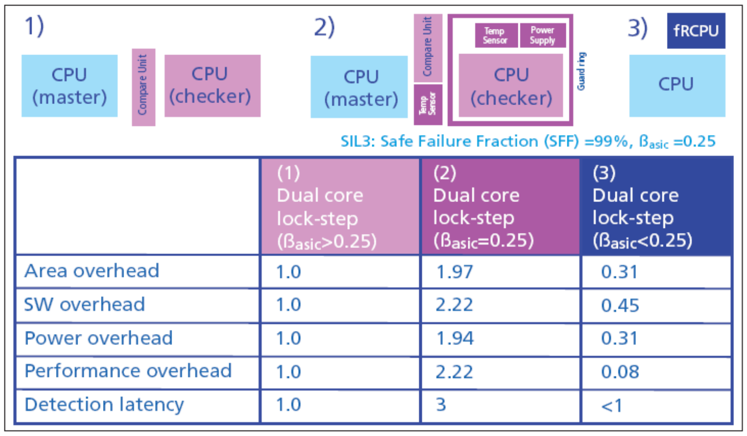
According to the equations 3.1 and 3.2, the so-called faultRobust IPs(fRIPs)[FM07] and Toshiba diagnostic circuits (ThwD) are built to support fault tolerance and fault detection. There are different IPs, dependent on the "critical region" they supervise.

In Figure 3.5, an architecture is presented that is built with the faultRobust Methodology. The safety functional units are:

1. fRCPU
2. fRMEM
3. fRBUS
4. ThwD
5. fRNET

The fRCPU is a fault supervisor which supervises the CPU core with its complete functionality. It is called the "optimized tightly coupled fault supervisor" because it has the same instruction control flow as the CPU but it is optimized to supervise the critical sections identified by the FMEA methodology, which can lead to failures. The fRCPU has a dedicated interface, which is used to monitor the CPU to increase the fault coverage and detection latency.

The fRMEM is a configurable IP, which uses EDC and ECC[6] to implement safety functions. They can either use parity bits or Hamming codes with different distances. Because the fRMEM is configurable, it can be optimized for different parameters.



	(1) Dual core lock-step (ßasic>0.25)	(2) Dual core lock-step (ßasic=0.25)	(3) Dual core lock-step (ßasic<0.25)
Area overhead	1.0	1.97	0.31
SW overhead	1.0	2.22	0.45
Power overhead	1.0	1.94	0.31
Performance overhead	1.0	2.22	0.08
Detection latency	1.0	3	<1

Figure 3.6: *Advantages of fRM Methodology over traditional architectures [Mar07]*

The fRBUS is a supervisor for the master and slave buses, which checks the address decoding, data transport in the buses, and includes ECC capabilities, and it is capable of detecting the loss of power supply.

In cooperation with Toshiba, the mechanism called "ThWD"[7] is included for peripheral supervision. Generally, these mechanisms are used to test the system timers mainly

[6]Error Detection/Correction Codes
[7]TOSHIBA hW diagnostic

focusing on: counter failures, register failures, clock failure, capture failures, and timer comparator failures. To facilitate testing, a complete redundant counter structure is implemented, and with the help of parity schemas the peripheral registers are tested to detect register failures.

The fRNET is the connection unit between all IP units and the ThwD logic described before. All error signals which are generated from different fRIPs are passed on to fRNET, which then fires a global error signal. It is equipped with self-test logic to test the signal path and to avoid latent faults.

	TMS570	**TC-1766**	**TSB/TC MCU**
Run time tests in Hardware	STC/LBIST for CPU's Compare Module for CPU's PBIST for memories ECC in memories & buses	Sequential Tests in start-up/shut-down (Flash Checksum, SRAM Tests) Periodic tests in peripherals	Self-check circuitry on fRIPs
Computing power	160 MHz with 250 DMIPS	80 MHz with 130 DMIPS	Dependent on the processor in which fRM is implemented
Guard Ring	"Guard ring per CPU, Minimum distance 100m"	No	Present
CCF Common Cause Failures - Robustness	Second CPU mirrored and rotated Cycle delayed lock-step Duplicated clock tree CPU's	Watchdog ASIC to monitor power supply	Intellectual Properties (faultRobust IPs or fRIPs) are architecturally and functionally diverse with respect to sub-block that they supervise
Availability	On Market	On Market	2011
Architecture	Lock-Step Mode on single chip	Asymmetric mode on single chip	fRMethodology Optimized Tightly Coupled (OTC) Dual Core
Number of cores	2 Cores	2 CPU's (TriCore and PCP2 core)	2 Cores
Cache	No	8 KByte instruction cache	No Information
Branch prediction	No	No	No Information

Table 3.1: *Evaluation criteria for the Microcontrollers-Part 1*

In comparison [FM07] to the different architectures according to Figure 3.6 [MB07], the faultRobust approach provides great savings in terms of area, power and performance compared to normal dual core lock-step architectures, such as described in Section 2.7a.

3.1.4 Evaluation Criteria

After the selection of the three microcontrollers, a list of evaluation criteria was formulated as illustrated in Tables 3.1 and 3.2, which cover safety-relevant features.

The properties of each controller were assessed with respect to the evaluation criteria. The main focus of the evaluation is on the safety features.

As illustrated in Table 3.1, each microcontroller has a different architecture and attempts to achieve safety in different ways. The TMS570 includes hardware safety tests for the main components: CPU, RAM, Flash, and Peripherals. Each of these features is integrated into a special hardware module.

As illustrated in Figure 3.5, the fRCPU supervises only the critical paths of the main CPU. The TC1766 has only sequence tests which can be executed at the start-up or shut-down.

The safety hardware STC/CPU test can be subdivided into different interval sets and executed at run-time dependent on the application requirements. RAM memory and flash are protected with ECC, which are calculated and evaluated on each read/write access to catch runtime failures. Additionally, all microcontrollers have built-in self tests covering all memories included in the chip.

Comparing the performance criteria, the TMS570 has an advantage of over 100 DMPIS comparing to the TC1766, but the TC MCU has no information about the speed and it is dependent on the architecture in which the fRM Architecture will be implemented in the future.

For the common cause failures, the fRM methodology has a more sophisticated protection concept rather than the TMS570. A disadvantage is that the TC-MCU is actually not on the market, and the TMS570 includes a guard ring between the cores with duplicated clock tree.

The TMS570 microcontroller implements the architecture described in Figure 2.7a, and the two others the architecture depicted in Figure 2.1.2.4. The advantages and disadvantages are described in the corresponding sections.

With respect to the criteria defining the development kits and prototyping board, again, the TMS570 is supported in a better way because it includes different tools, IDEs for development, flash, compilation etc., and has two types of prototyping boards on the market, which can be used for prototyping implementation. The only advantage of the TC1766 is the support for Mathlab & Simulink, which extends the support for simulation of control systems.

The Instruction cache is included in the TC1766 microcontroller. The presence of the cache has advantages for the instructions which are executed but it is non deterministic for testing purposes.

Continuing with the communication interfaces, the first two microcontrollers are similar, considering the number of interface types. The TMS570, however, has additional powerful features for the interfaces including parity and bus protection.

The Memory protection and online checker are present on three microcontrollers, it differs only how these features are implemented and how many regions they protect.

	TMS570	**TC-1766**	**TSB/TC MCU**
Development environments	CCS[8] IDE v4.1 HalCoGen[9] TargetLink JTAG Emulator nowFlash nowECC	Mathlab & Simulink TriCore VX-Toolset MemTool(Infineon)	No Information
Instruction set	ARM, Thumb, Thumb2 v7	PCP2 - RISC instruction set	No information
Prototyping board	TMS570 Microcontroller Development Kit & Microcontroller USB Kit	Starter Kit TC1762 & TC1766	No information
Memory protection	MPU protects up-to 8 regions	Memory Protection Tables for 4 data regions & two code regions DMA controller with memory protection	The fRMEM supervisor ECC based with measure extensions MCE block acts as MPU
Safety features	ECC on Flash & SRAM CRC using DMA protects static data in memory	Parity on all SRAMs ECC on Flash CRC32 peripheral for critical sections on RAM	fRCPU, fRMEM fRBUS, ThwD fRNET
Peripheral Interfaces	Three Multi-buffered SPIs Three CAN Controllers Dual Channel FlexRay Controller Two UARTs (SCI) with LIN Interface	Asynchronous, Synchronous Serial & High-Speed Synchronous Serial Interface Micro Link Serial Bus Micro Second Bus MultiCAN module	No information
Peripheral Safety	Parity protection of peripherals memories Bus protection	Internal bus tests and peripheral test	ThWD safety mechanism Parity on registers
Pipeline	8 stage pipeline	4 stage pipeline	No information
Online Checker	Hardware Compare Module of CPU outputs	PCP Monitor Unit ASIC watchdog	Hardware fRCPU comparing instructions at run-time

Table 3.2: *Evaluation criteria for the Microcontrollers-Part 2*

In the TMS570 microcontroller, eight memory regions can be protected from changes during runtime from changes which can be intended by a user or other influences, The TMS570 has an advantage of four regions, compared to the TC1766, which protects only four data regions and two code regions.

The TMS570 has a better pipeline with eight stages, and the TC1766 has only four of them.

The online comparison modules in the TM570 and TSB/TC-MCU are implemented in hardware and compares every instruction executed in the main core. In the TMS570 every instruction is duplicated and executed on both cores, and the outputs are transmitted to the compare module which compares the results for equality.

Based on these criteria, the TMS570 was chosen as the most suitable for the implementation of safety-related measures, because it has a SIL-3 certification, better performance, good development environment, and numerous safety features.

In the Section 3.1.1, the architecture of the TMS570 was described, but in the next section, the safety features from the technical view will be described, and more details will be listed, which will help to incorporate them in the safety strategy.

3.2 CPU, RAM and Peripheral tests in the TMS570 microcontroller

According to the Technical Reference Manual [TMS] the main safety features of the TMS570 are integrated into the following modules implemented in hardware:

1. STC/LBIST (Self Test Controller / Logical Built in Self-Test)
2. CCM-R4F (Core Compare Module)
3. PBIST (Programmable Built-In Self-Test)
4. ESM (Error Signalling Module)

These are the primary relevant modules for the strategy which will be implemented.

3.2.1 CPU Tests

The CPU Self-Test Controller (STC) is an integrated on-chip BIST support to test the CPU cores with high diagnostic coverage. It is possible to run different levels of the tests, while during the execution, the CPU cores are isolated and all bus transactions signals are in idle mode.

Furthermore, all pending interrupts are served and processed after the tests are finished. The test can be run at start-up without needing to perform a backup of the registers. During runtime before the test is started, a backup of the register is required. The register contents need to be restored after the test.

With a diagnostic coverage of over 90% and 32 intervals as illustrated in Table 3.3, the complete test execution requires between 0.926 msec and 1.11 msec depending on the hardware and STC clock.

Intervals	Test Coverage [%]	Test Cycle
0	0	0
1	57,14	1555
2	65,82	3108
.	.	.
.	.	.
15	86,19	23297
16	86,56	24850
17	86,97	26403
.	.	.
.	.	.
30	90,46	46592
31	90,64	48145
32	90,84	49698

Table 3.3: *STC levels and coverage [TMS]*

3.2.2 Core Compare Module Tests

The core compare module not only is responsible to compare the results of the output instructions, it also contains the following built in self tests which can be used during runtime:

- self-test
- error forcing test
- self-test error forcing mode

The start of the test must be issued by the application. During the "self test", the module can generate compare match and compare mismatch patterns. Applying compare match patterns (four patters 0x0, 0x1, 0xA, 0x5) at the outputs, equal results are expected. If a fault is detected, an ESM error is generated, and the test is terminated, otherwise the termination must be performed by the application.

The compare mismatch patterns are applied in the same manner with the difference that the patterns for the second core are inverted, and different result is expected as in the first core. The test takes 3615 test cycles.

Applying the error forcing test only one pattern is applied to force that an error is introduced in path between CCMR4F and ESM. If after the test is applied, and no error is asserted on the ESM module, a hardware fault is present.

The tests that are described check only the CCMR4F module, and the normal application can use the CPU's in the background, but the outputs of the CPU's are not compared with each other.

3.2.3 RAM and Peripheral Tests

The PBIST (Programmable Built-In Self Test) architecture is a built-in testing engine controlled by a small CPU. It offers the possibility to test all types of memories with pre-implemented algorithms in the microcontroller.

Ram Group	Module	Memory type
3	DCAN1	Single Port
4	DCAN2	Single Port
5	DCAN3	Single Port
6	ESRAM	Single Port
7	MIBSPI	Single Port
8	VIM	Single Port
9	MibADC	Two Port
10	DMA	Two Port
11	NHET	Two Port
12	HET TU	Two Port
13	RTP	Two Port
14	Flexray	Single Port

Table 3.4: *RAM groups in TMS570 [TMS]*

In Table 3.4, all RAM memories with types of the microcontroller are presented. For each type of memory, in Table 3.5 the algorithms are shown.

Memory type	Module	Valid RAM Groups
Single Port	march13n-red	3,4,5,6,7,8,14
Two Port	march13n-red	9,10,11,12,13
Single Port	down1A-red	3,4,5,6,7,8,14
Two Port	down1A-red	9,10,11,12,13
Single Port	mapcolumn	3,4,5,6,7,8,14
Two Port	mapcolumn	9,10,11,12,13
Single Port	precharge	3,4,5,6,7,8,14
Two Port	precharge	9,10,11,12,13
Single Port	dtxn2	3,4,5,6,7,8,14
Two Port	dtxn2	9,10,11,12,13

Table 3.5: *Algorithm to RAM mapping in TMS570 [TMS]*

- March13N - As described in Section 2.2.2.2, March13N provides very high diagnostic coverage. The fault types detected by this algorithm are manifold starting from, decoder, stuck-at, coupled, parametric faults, and ending with logic faults. With 13N operations where N is number of RAM Cells, it has linear complexity O(n).

- Map Column - is commonly used to find bit lines sensitivities in memory. It consists of writing on each line of memory 1 and 0's and repeatedly reading and writing back.
- Pre-Charge - used to address the pre-charging of the SRAM cells because the analog portion is frequency-sensitive.
- DOWN1a - addresses the data bits and address bits by switching their content with a forced transition.
- DTXN2a - mainly used to test decoding logic in RAM memories.

The complete content of the affected memory part are lost after the tests are applied. Therefore, all these tests are carried out at the start-up. The complete tests needs 23.23 msec at HCLK = 80 MHz and ROMCLK = 100 MHz and 43.63 msec with HCLK = 100 MHz and ROMCLK = 40 MHz to be completely executed.

3.2.4 Error Signaling Module - ESM

The ESM Module is used to control the error signals that come from different sources of the microcontroller and can be transmitted to a dedicated external pin.

The main properties of the ESM module are :

- 96 interrupt/error channels for various error sources
- Dedicated pin $\overline{ERROR}$ to signal external sources

The first thirty two error sources are maskable during normal runtime, and the rest are non-maskable.

3.3 Functional Safety Requirements

- At start-up, the RAM should be initialized with ECC.
- At start-up,all RAM memories must be tested with the PBIST logic, which ensures that the RAM and Peripheral memories are in a safe state.
- The RAM must be initialized to a known state.
- It must be ensured that all errors are reported correctly and that the system does not enter an unsafe state.
- During runtime the RAM must be tested completely, and it must be guaranteed that the normal application software is not halted in its normal operation.
- It must be ensured that after disabling the Interrupts, they are enabled after the test is finished.
- During runtime the execution time of the RAM test must not exceed the critical time for which the Interrupts are disabled.

- At start-up, the two cores must be tested with 32 levels of test patterns with the STC controller
- At runtime, the STC tests can be run with the complete test level or the tests can be divided from 1 to 32 levels and can be run periodically.

3.4 Design Concept

After describing the microcontrollers and listing the safety requirements, which a safety strategy must fulfil, a design concept needs to be found. This concept consists of:

1. Hardware tests
2. Software tests

Using the hardware tests is advisable, because the functionality which is offered in hardware is reliable, faster, and less error prone than tests that are implemented in software. Software tests will be implemented because the implemented functionality does not offer an online periodic testing schema of the SRAM at the microcontroller.

The hardware and software test as illustrated in Tables 3.6 and 3.7 are subdivided into tests which can be executed at start-up and tests which can be continuously executed continuously during runtime.

Algorithms:	Start- Up	Runtime
March13N	X	
Map Column	X	
STC / LBIST	X	X
DTXN2a	X	
DOWN1a	X	
Pre-Charge	X	
Watchdog Test		X

Table 3.6: *Hardware tests in the TMS570 microcontroller*

Algorithms:	Start-Up	Runtime
March-Tests		X

Table 3.7: *Software tests in the TMS570 microcontroller*

FreeRTOS[10] is available to be used in connection with the TMS570. The basic thread of execution in the FreeRTOS [Bar10] is called a task, which has a dedicated stack and priority that runs forever, and the scheduler is responsible to give each of them CPU time for execution. The advantage of using such a kernel over a common "finite state machine"

[10]http://www.freertos.org/

are synchronizing issues. The synchronization of the application and the tests, during the execution are done by the FreeRTOS scheduler. The task with the higher priority if in ready state is scheduled to run.

In Section 2.2.2.2 different RAM testing algorithms were introduced and described. Deciding upon an appropriate algorithm based on the criteria which are merged within Tables 3.8 and 3.9.

The smallest number of operations and the linear complexity are such criteria. Sorting according to complexity, three types of algorithms are present: linear, logarithmic, and quadratic.

In the group with linear complexity are the algorithms, Checkerboard, March B and Abraham with the Checkerboard as the best representative. They has smaller number of operations by factor 4X and 7X compared to the other two.

Including the second criteria in the evaluation, namely the covered faults, the "March B" covers a larger set of faults than the Checkerboard test or the Abraham test. Constructing the tuple (Number of Operations, Covered faults) of criteria the MARCH-B proves to be as the best representative for the implementation.

Algorithms	Number of Operations	Covered Faults
Checkerboard	$4n$	detects not all AF, SAF, TF and CFs
March B	**17n**	all AFs, SAFs, TFs, CFins and linked CFids
Abraham Test	**30n**	TFs, CFs
Butterfly	$\mathbf{5n * \log n}$	AFs and SAFs
GALPAT	$\mathbf{4n^2 + 4n}$	AFs, TFs, CFs, and SAFs
WALPAT	$\mathbf{2n^2 + 8n}$	SAF, AF, TF and CFs

Table 3.8: *RAM test Faults coverage and number of operations*

The algorithms with quadratic complexity are not suitable to be implemented because the number of operations are very high and they do not cover the kind of faults that are covered by the group of linear complex algorithms.

Complexity / **Size**	**n**	$\mathbf{n * \log n}$	$n^{3/2}$	n^2
1K	0.0001s	0.001s	0.0033 s	0.105 s
4K	0.0004s	0.0048s	0.0262 s	1.7 s
16K	0.0016s	0.0224s	0.21 s	27 s
64K	0.0064s	0.1s	1.678 s	7.17 m
256K	0.0256s	0.46s	13.4 s	1.9 h
1MB	0.102s	2.04s	1.83 m	1.27 d

Table 3.9: *Test Time as a Function of Memory Size [WWW06]*

According to these criteria, the MARCH B test is implemented. Another March test is implemented in the hardware with 13N operations, which will be run at the start-up. Concerning the structure, they are the same, they differ only in the number of operations. The implemented march test is redundant and can be run in parallel with the application.

3.4.1 Modules

The complete concept is built on three modules as illustrated in Figure 3.7.

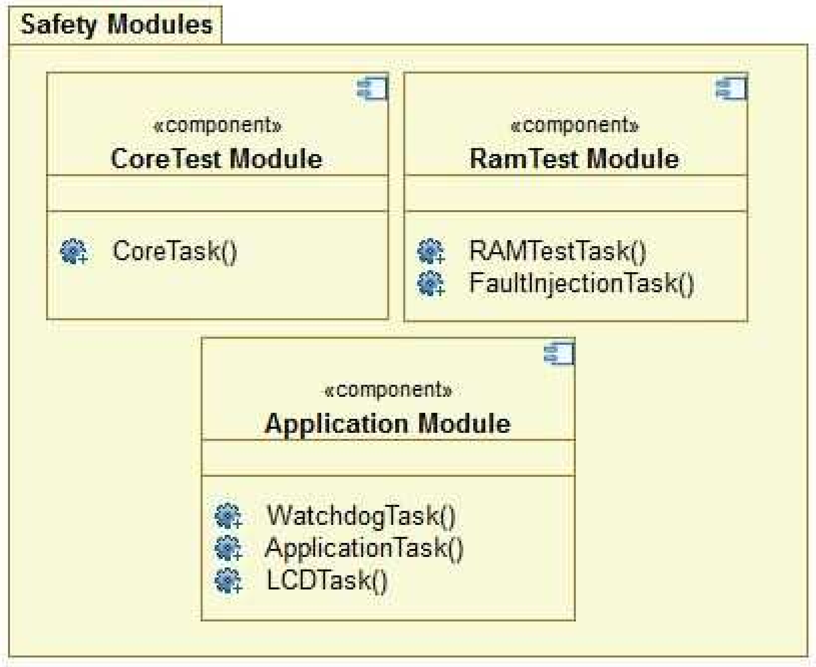


Figure 3.7: *Modules in the concept*

At the start-up, the PBIST algorithms of the TMS570 are carried out, because the content of the memory is completely lost during testing. The watchdog, RAM and STC tests are applied inside the tasks, periodically online.

The tasks inside the modules illustrated in Figure 3.8 exist in the context of the OS Kernel, and the scheduler is responsible to provide each of them with execution time.

These tasks are created to cover the following functionality:

- Core Test - It was decided to run the CPU test periodically because during that test the interrupts are not served, the core test is adaptable to configure the number of intervals, and the test is executed under 1 ms.
- RAM Test - The pseudo-code of the selected RAM test is illustrated in the Algorithm 5, and it describes all operations which must be performed to run the test.
- Watchdog - The TMS570 Prototyping board has an external windowed watchdog that can be used for program flow monitoring of the application and it is integrated

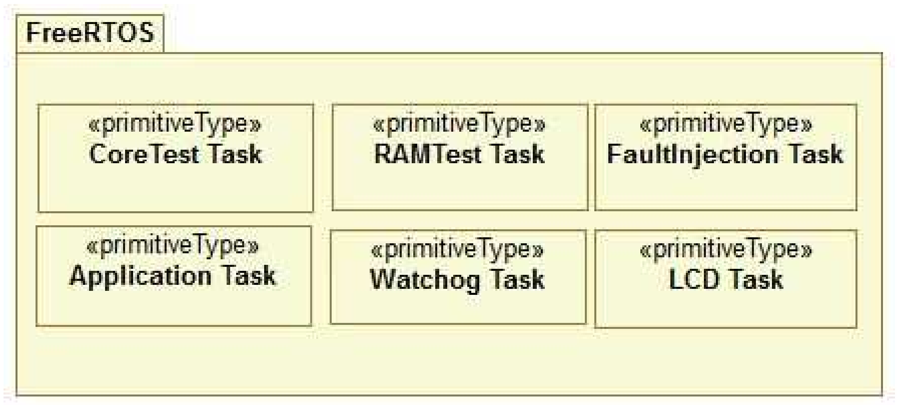

FreeRTOS
«primitiveType» CoreTest Task
«primitiveType» RAMTest Task
«primitiveType» FaultInjection Task
«primitiveType» Application Task
«primitiveType» Watchog Task
«primitiveType» LCD Task

Figure 3.8: *Tasks executed in the FreeRTOS kernel*

into the concept. This watchdog starts counting after the first rising edge on the WDI interface as illustrated in Figure 3.9, and after a defined time, this interface must be set to low (falling edge). After the switch SW4 is turned on, then the watchdog must be reset during the defined time interval. The time interval cannot be changed.

- Application - As part of the application, the status of the test is presented on the LCD Display. An application ("blinky") is implemented which uses the numerous LEDs on the board. The external watchdog monitors the program flow. The blinky application is a led show consisting of the various LEDs which are present on the prototyping board.
- Fault injection - For verification purposes, these tasks will inject faults into the RAM.

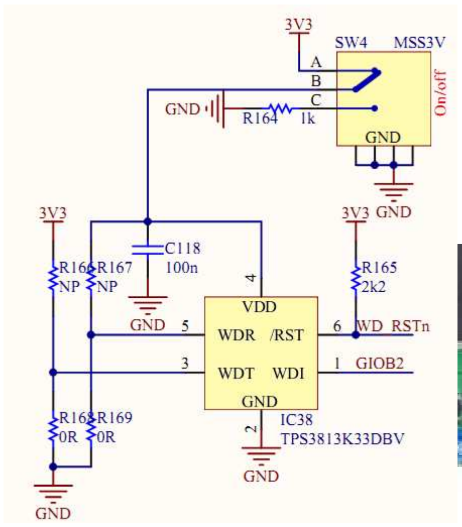
The Figures 3.9 and 4.10 illustrates how the watchdog is integrated into the prototyping board and how the watchdog needs to be reset within the appropriate timing window.

3.4.2 Properties of the modules

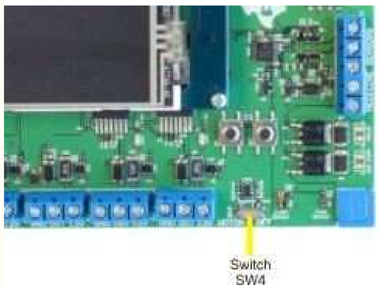
The tests that are active at run-time are included in separate tasks. During the implementation phase, the following parameters need a special treatment:

1. Task Priorities
2. Execution time
3. Context Switch
4. Interrupts

Task priority is important because the scheduler chooses the ready task to run based on the priority. Obviously, the tests will have lower priority than the application, but the



(a) *Circuit diagram*



(b) *On/off button*

Figure 3.9: *Watchdog circuit used for program flow monitoring*

execution time of each test should not stop the application in its correct operation. This can be seen as a compromise between priority and execution time.

The FreeRTOS Kernel offers nine levels of priority with the lowest priority represented as 1 and the highest represented with the number 9. The application implemented in the concept has the priority 4, and all other tests 3. This ensures that the tests are running if the application is in idle mode. These priorities assigned to the tasks present a risk if the application is busy forever, because it takes the CPU, and the other tasks never get into the running state. This must be taken into account during the implementation.

When each task is ready to run, a context switch is performed, which means the complete context of the actual task (program counter, registers, stack pointer) are stored on the task's stack on the RAM memory.

Hence, these two conditions must be avoided/disabled at the run-time of the tests. During the startup of the test the interrupts are disabled to avoid test interruptions, the actual code region will be marked as critical to avoid a context switch, and another task is scheduled to run. For the RAM test implementation, some possibility to perform fault injection testing must be implemented to verify the concept.

While the tasks with the tests are running, a context switch is not allowed. A context switch can occur on two conditions:

- An Interrupt happens during that time
- One task with higher priority becomes ready

The fault coverage is guaranteed due to the March Test structure, which is discussed in Section 2.2.2.2, but a fault injection strategy must be implemented.

Two cases are at disposal:

1. Change the RAM content in the cells during runtime
2. Use ECC functionality of the microcontroller to corrupt the RAM

Changing the content of RAM during a running test would, however, not meet the requirements listed in the previous chapter, which means context switching and interrupts are disabled and all tasks are suspended until the test is finished with the actual sector.

Algorithm 5 March B testing the memory with N cells

```
for $i = 0 \to n - 1$ do
    $RamCell[i] = 0$
end for
for $i = 0 \to n - 1$ do
    $CheckRamCell[i] == 0$
    $RamCell[i] = 1$
    $CheckRamCell[i] == 1$
    $RamCell[i] = 0$
    $RamCell[i] = 1$
end for
for $i = 0 \to n - 1$ do
    $CheckRamCell[i] == 1$
    $RamCell[i] = 0$
    $RamCell[i] = 1$
end for
for $i = 0 \to n - 1$ do
    $CheckRamCell[i] == 1$
    $RamCell[i] = 0$
    $RamCell[i] = 1$
    $RamCell[i] = 0$
end for
for $i = 0 \to n - 1$ do
    $CheckRamCell[i] == 0$
    $RamCell[i] = 1$
    $RamCell[i] = 0$
end for
```

The Cortex R4F core always internally calculates an ECC for all RAM read/write accesses. It can detect single bit errors and correct them. Double and more bit errors are detected but cannot be corrected. It is also possible to enable/disable the reporting of detected errors, but it is not possible to disable the detection and correction.

In the implemented strategy, the reporting is enabled. If it fails during the program flow the redundant RAM test will find the faults in the memory.

Chapter 4

Implementation and Results

4.1 TMS570 Evaluation Board

The MCBTMS570 [KEI] prototyping board by KEIL is presented in Figure 4.1. It contains an ARM9[1] processor with two Cortex R4F Cores in Lockstep mode. As presented in Block Diagram 4.2, it has many peripherals and an analog input part.

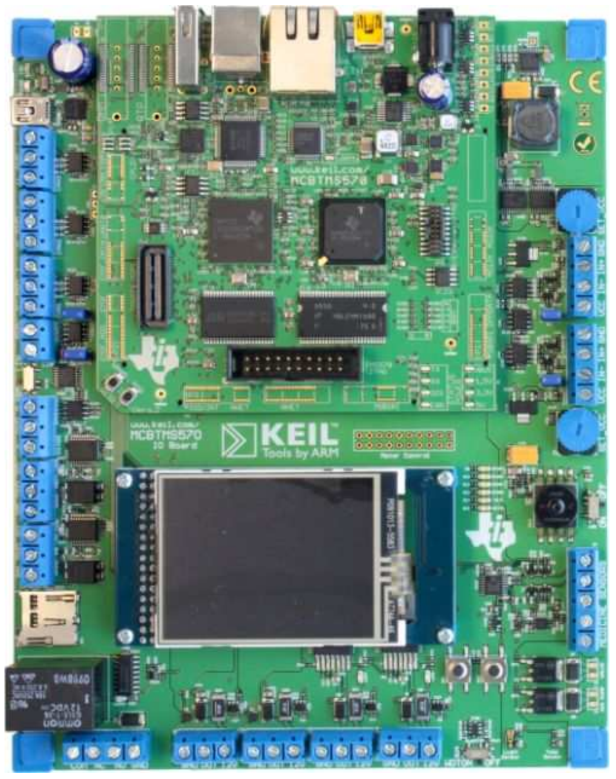


Figure 4.1: *MCBTMS570 Development Kit [KEI]*

[1]Advanced RISC Machines

The Prototyping consists of two parts:

1. The MCBTMS570 CPU board
2. The MCBTMS570 I/O board for the peripherals and the analog circuits

The CPU board includes the main logic of the prototyping board and the following components:

- A TMS570LS20216 microcontroller
- A 16 MHz Oscillator
- The ETM Interface (MIPI) connector which provides instruction-level trace debugging support
- Three USB Standard ports: USB 1.1, 2.0 and mini-B connector
- An 100/10M Ethernet Port A standard (RJ45) connector connected to an on-board Ethernet transceiver
- Two JTAG interfaces: USB/Ethernet-JTAG connector and a standard JTAG 20-pin connector
- Two push buttons for: The RESET WARM (S1) to wake the microcontroller from sleep mode and the RESET POR (S2) button used as power-on reset circuit

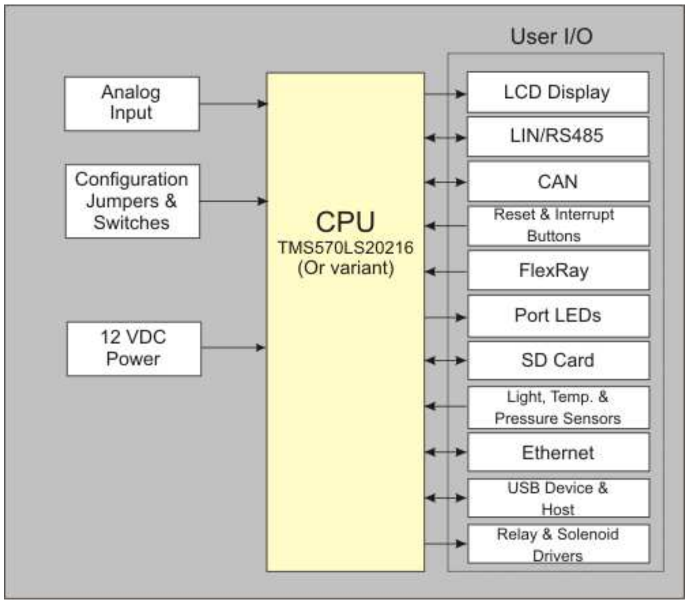
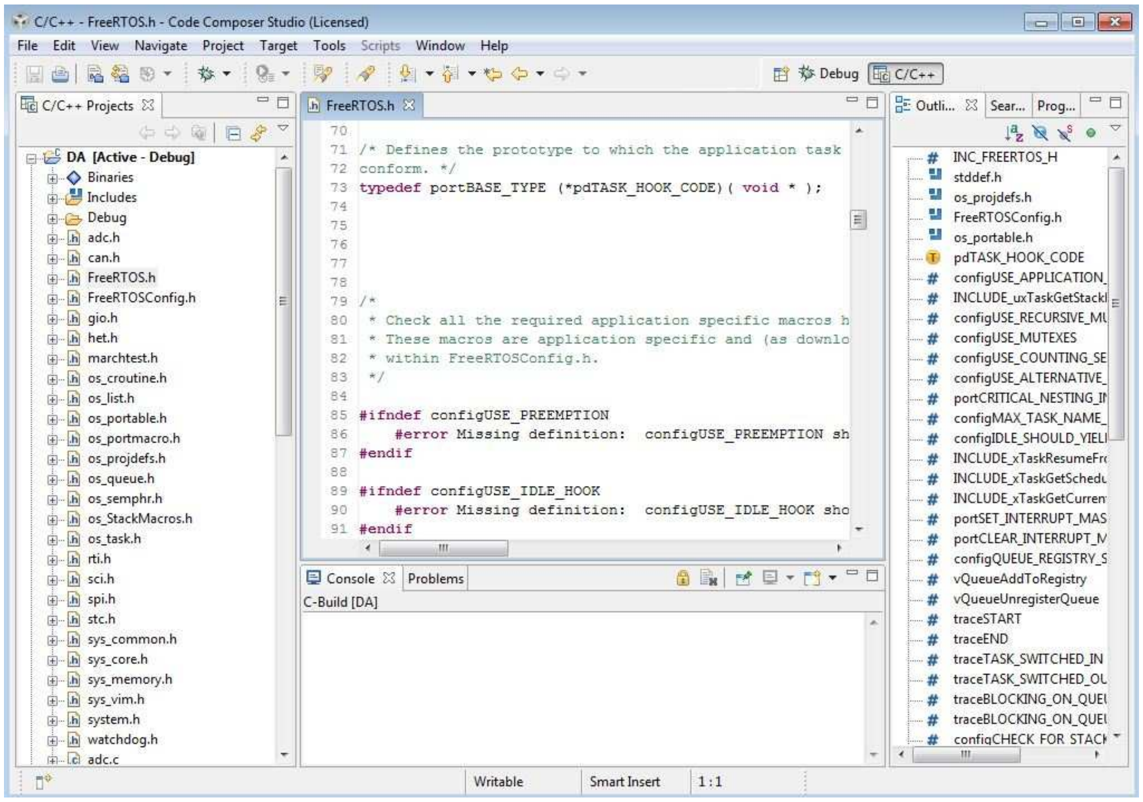


Figure 4.2: *Block diagram of the MCBTMS570 prototyping board [KEI]*

The second part or the I/O board contains the peripherals, transceivers for the communication ports, and other components as listed below:

- Three CAN Ports with transceivers
- One LIN port and one RS485[2] port with the corresponding transceivers
- Two connectors for FlexRay network communication
- A 240x320 TFT Touch Screen/LCD Display
- Temperature and light sensor realized with on-board ADC converters
- Amplified speaker for audio output
- MicroSD card connector for SD cards
- Two push buttons: SW1 and SW2, which can be used for the application
- Four jumpers and three switches for: LIN/RS485 switching, on/off watchdog timer and on/off pressure sensor

Internally the prototyping board has an XDS100v2 emulator. It is possible to flash the board without an external JTAG emulator and it can be used with an USB connector.



```

/* Defines the prototype to which the application task
conform. */
typedef portBASE_TYPE (*pdTASK_HOOK_CODE)( void * );





/*
 * Check all the required application specific macros h
 * These macros are application specific and (as downlo
 * within FreeRTOSConfig.h.
 */

#ifndef configUSE_PREEMPTION
    #error Missing definition:  configUSE_PREEMPTION sh
#endif

#ifndef configUSE_IDLE_HOOK
    #error Missing definition:  configUSE_IDLE_HOOK sho
#endif
```

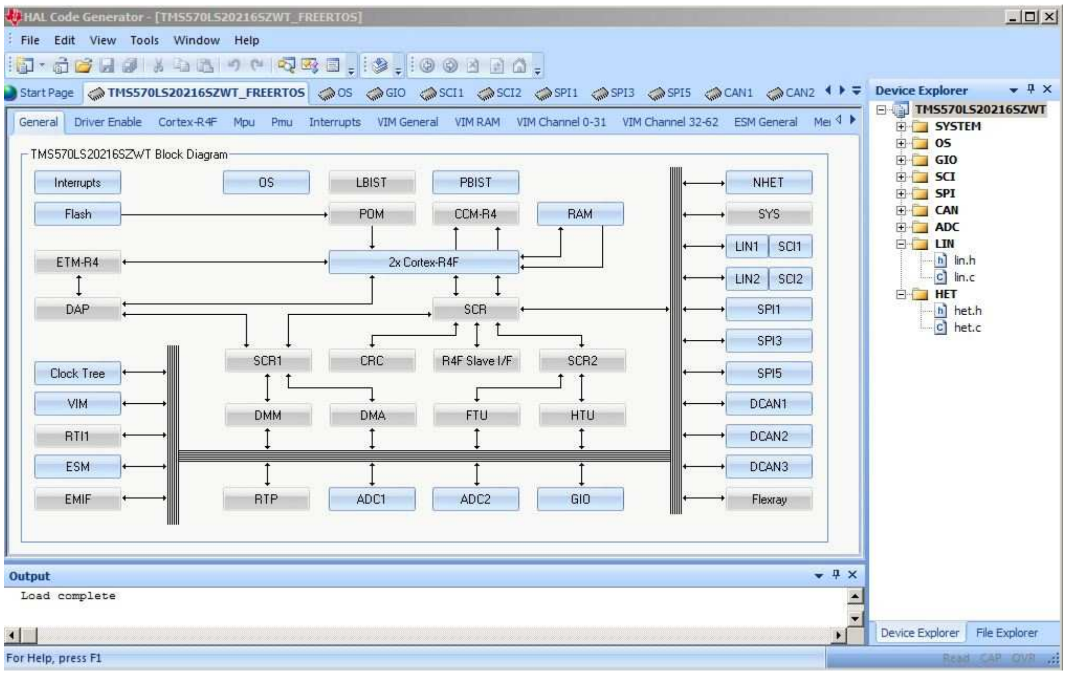
Figure 4.3: *Code Composer Studio IDE*

[2]EIA-485 is a specification of local networks and communications links

4.2 Workflow and Tool Chain

The TMS570 microcontroller is supported by a large group of tools which can be used during the development of applications. Starting from IDEs and flashing tools which are essential for every developer, the Starter Kit includes more supporting software as listed below:

1. Code Composer Studio IDE
2. HALCoGen Peripheral Drivers Generation Tool
3. Flash programming tool integrated into Code Composer Studio
4. nowFlash
5. nowECC
6. HET GUI/Simulator/Assembler including Synapticad WaveViewer
7. FMzPLL & FPLL Calculators



```
Load complete
```

Figure 4.4: *HalCoGen tool from Texas Instruments*

Code Composer Studio is an Eclipse-based framework which includes tools from TI, debugger, compiler, building environment, simulator, and the possibility to flash directly from the IDE. This feature and the debugging features are very helpful at the beginning of the work with the prototyping board.

Texas Instruments has developed a code generator and configuration tool(HalCoGen) which helps to configure the microcontroller, starting from the PLL, interrupt sources, operating system, all peripherals, safety features, all clock trees etc.

The HalCoGen allows to generate code that can be compiled and deployed on the prototyping board. The compiled binary can be deployed with the nowFlash tool or directly from the CCStudio. It is recommended to generate the ECC of the complete binary and to flash it manually as presented in Figure 4.5 using the nowFlash tool.

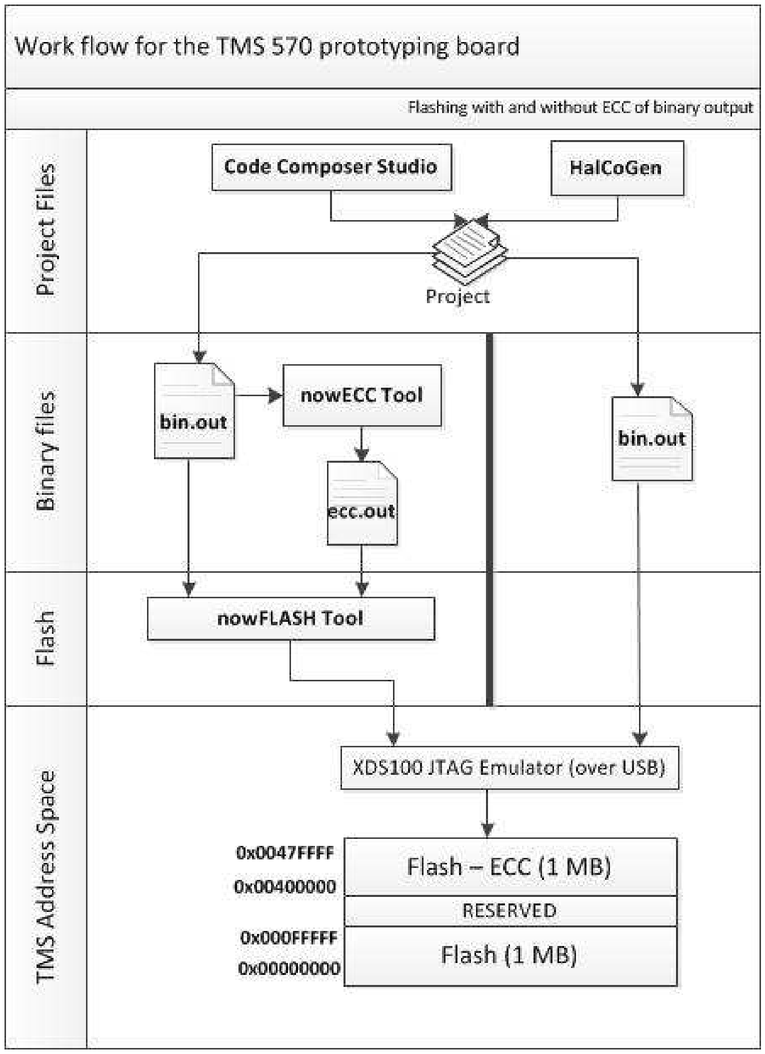


Figure 4.5: *Workflow to deploy binarys on the TMS570*

The TMS570 architecture has separate address spaces for the ECC part and the program/data part. As safety feature it includes mirror images in separate address spaces for ECC part and program part. More information about the ARM software flow can be found in [ARM11] on page 16.

In the support package, three other GUI based tools are included. Two PLL GUI calculators for the internal PLL and for the FlexRay PLL can be used to generate values for the internal PLL registers. The High End Timer Simulator can be used to simulate and configure timer functionality like waveforms generators (PWM), memory write triggers, cycle counts etc.

4.3 Start-up Phase

To run the software tests periodically as defined in the concept in Section 3.4, the FreeRTOS kernel is used, which is an open source and can be provided by the HalCoGen tool, which can be employed to configure tick rate, priorities, task modes, scheduler properties and stack/heap size. The OS is responsible for switching between the tests and therefore it is defined as pre-emptive to let the tasks with higher priority run while the others are suspended.

In FreeRTOS Tasks [Bar10] are defined with the following function:

```
portBASE_TYPE xTaskCreate
   ( pdTASK_CODE pvTaskCode,             // Pointer to the Task function
     const signed portCHAR * const pcName,// Name of the Task
     unsigned portSHORT usStackDepth,     // Size of the Task stack
     void *pvParameters,                  // Possible to pass parameters
     unsigned portBASE_TYPE uxPriority,   // Priority of the Task (1-9)
     xTaskHandle *pxCreatedTask           // Handle of the created Task
   );
```

The functions are implemented as endless loops with the possibility to suspend a specific task for a defined time.

```
void pvTaskCode( void *pvParameters ){

    portTickType xLastWakeTime = xTaskGetTickCount();
    for(;;) {

        vTaskDelayUntil( &xLastWakeTime, (PERIOD_MS / portTICK_RATE_MS ));
    }
}
```

Before the system with the defined safety-related tasks are put into operation, at start-up, hardware implemented self tests and additional safety measures are carried out for accuracy. This is necessary to ensure that the device operates normally. At start-up, the sequence of safety tests is as the following:

1. CCM-R4 checks for faults in self test mode by a matching pattern as well as by a mismatching pattern.
2. Cortex-R4F core checks for faults using STC/LBIST running all the test patterns.

3. MPU is configured appropriately to protect memory regions.
4. The complete SRAM is initialized to a defined state with ECC enabled using the auto initialization feature of the board.
5. Hardware PBIST engine executes multiple RAM testing algorithms with 99% fault coverage.
6. Redundant address decoding logic tests are performed on the CPU core.

The complete start-up sequence is presented as UML diagram in Figure 4.6. After the reset interrupt is generated the core registers [RIT] must be initialized with appropriate values. Hence, the reset source is scrutinized whether it is generated by the core test. In this case, it will branch to the previous program flow (main routine or core test task) to continue the operation. Otherwise, the normal start-up procedure continues.

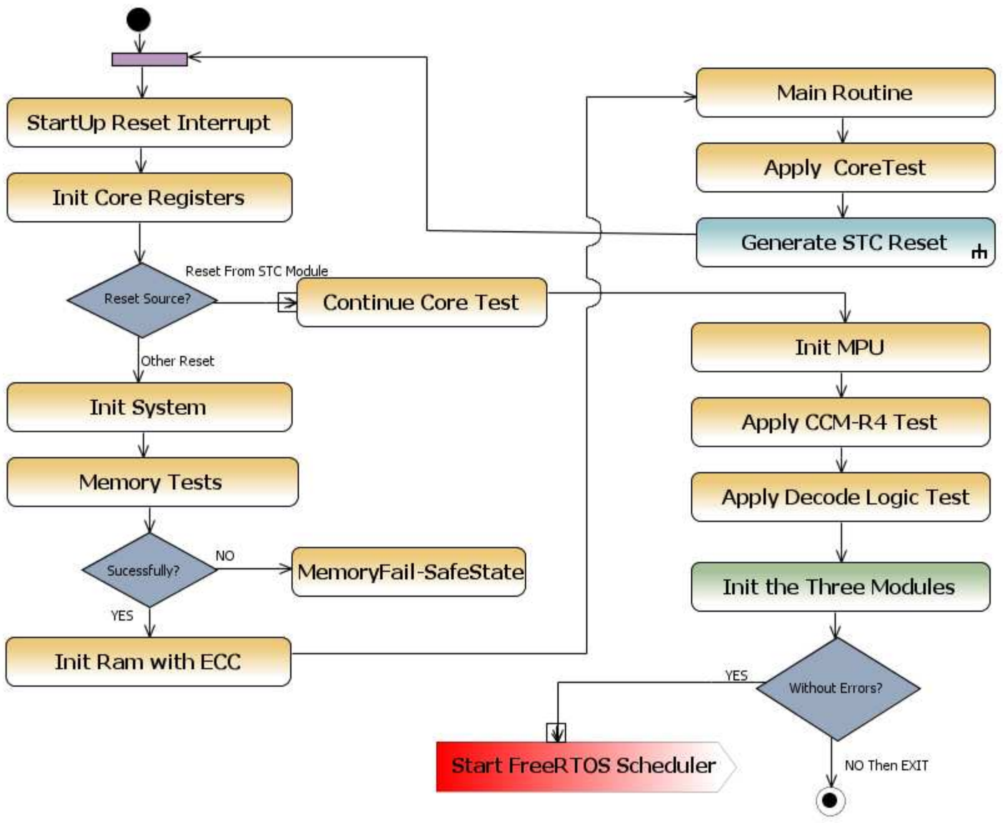


Figure 4.6: *The safety start-up sequence*

After initializing the system clock sources, flash wait cycles etc., the memory tests are applied to the RAM and the peripherals. If the test fails, it branches to a safe state, and the start-up cannot continue because that could lead to undefined error states.

If the test was executed without errors, the memories are initialized with the auto initialization feature. Branching at the main routine, the first step is to apply the core test with the maximum number of the intervals. After generating the reset, the status bit (successfully, failed) is stored.

Then the MPU is initialized and, the core compare module test and the decode logic test are applied. The status bits of these tests are stored too. After initializing the modules as described in Section 3.4.1, all status bits are evaluated.

If one the modules was not successfully initialized or one of the tests has failed, then the procedure exists without starting the application. Otherwise the flow continues and starts the scheduler of the FreeRTOS which continues to execute the tasks defined in the modules.

4.4 Periodic Operation

4.4.1 RAM test

In Chapter 2.2 different RAM testing algorithms are presented and explained based on the running complexity and fault coverage.

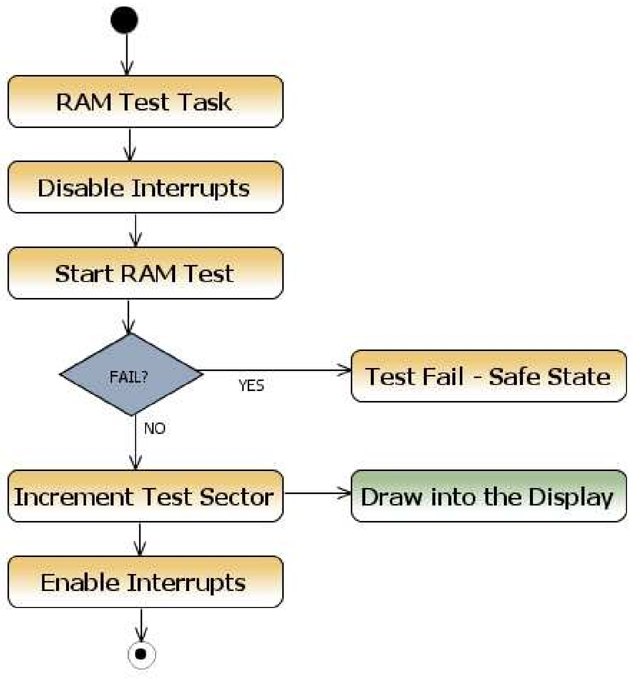


Figure 4.7: *Activity diagram of the RAM Test*

March testing algorithms with March-B are suitable to be implemented, because it has linear complexity O(n) and covers stuck at faults, address decoder faults, transition faults, inversion coupling faults, and linked idempotent coupling faults. The advantage over the March-G, is that it has very small execution time and does not contain delays like March-G.

The RAM test is defined to run periodically as a FreeRTOS Task:

```
xTaskCreate( RAMTest,
 (signed portCHAR *)"Task Ram-Test",
     500,
 (void*)ramTestTxt,
 3,
 &xRamTestHandle);
```

The task is periodically executed by the RTOS Kernel and calls the function in assembler to test the actual sector number of the RAM. Some parts of the test implemented in the assembler are presented in Listing 5.1.

In Figure 4.7, an UML activity diagram is presented, which describes the complete sequences of the OS Task needed to run the ram test. In the first step after the task switches into the running state, all interrupts are disabled.

The test presented in Listing 5.1 is executed, and if it fails, the complete system goes into a safe state. After branching to safe state, the tasks are switched to stop state, and the other resources are disabled. This state would be responsible to switch off actuators or sensors.

Otherwise, the sector address is incremented, interrupts are enabled, and the task changes to sleep state until the scheduler chooses the task next time and the scheduler passes the CPU on to the next ready task.

This test is very efficient but it is destructive for the RAM content, because after the test is applied, the complete content is deleted.

As presented in Figure 4.8, the base address of the RAM is 0x08000000 with a size of 160 kB and an end address at 0x08027FFF. During runtime, the application flow must not be changed and the content of the RAM must be restored after the test is applied.

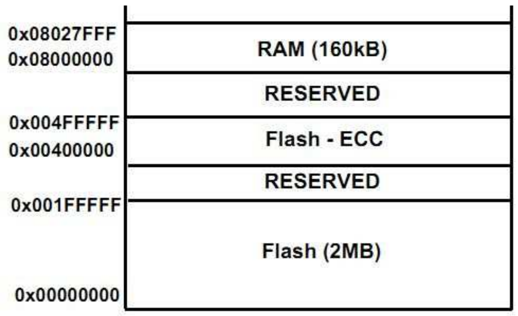


Figure 4.8: *The ESRAM in TMS570 [TMS]*

Hence, a backup strategy must be developed to store the content of the RAM. It is possible to subdivide the RAM memory into active and passive regions, but there are several problems to this solution. First, it is complex to manage, and so it is difficult to fulfil the timing requirements. Second, the local variables in RAM, which are used to run the test, can be overwritten.

Because of these problems, the following requirements must be fulfilled:

1. The RAM cannot be used as backup storage
2. The test must run completely from FLASH without using local variables in RAM
3. The backup strategy must be fast

There are two other components which can be used for backup: Flash and Core registers. Using the Flash as backup storage has advantages and disadvantages. A great advantage is the storage capacity which could be used to backup a huge amount of data from the RAM and offers the possibility to verify the integrity of the data with the help of error correction codes. As a disadvantage, writing/reading to/from Flash is very slow, which does not meet the previous requirements and it is not appropriate to be used for backup. So it is required to backup RAM contents in the CPU registers.

According to the ARM Architecture Reference Manual [ARM] for the Cortex R4F processors, it has 32 general purpose register and, 32 bit width and a subset of registers are accessible from different modes as presented in Figure 4.9.

These modes are used to grant privileges code parts during execution. The ARM architecture supports the following processor modes:

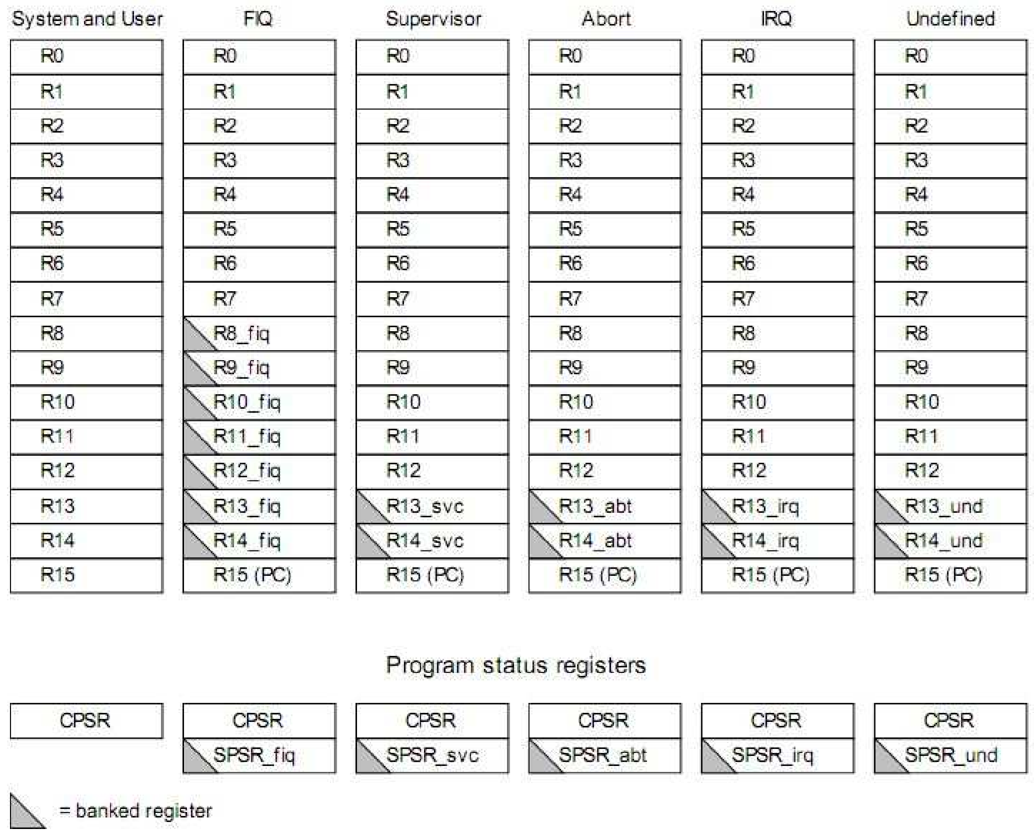


Figure 4.9: *General purpose registers of the ARM Architecture [ARM]*

1. System and User
2. FIQ
3. Supervisor
4. Abort
5. IRQ
6. Undefined

All applications normally run in the system/user mode, and the OS uses the supervisor mode. The two interrupt modes are used to serve the interrupt requests, but the difference between them is that FIQs are of higher priority than the FIQ interrupts. The Abort mode is used for data abort instructions and the Undefined mode is entered if there is an exception[3].

Listing 4.1: *March test implemented in Assembler*

```
_esramTest_:
                  ldr r2, zeropattern
                  ldr r3, onepattern
; Store the content of the RAM in the register
                  LDMIA    R0!, {R5-R12}
; Load back the start adress because LDMIA increments it
; by [numBytes] loaded
                  mov  r1, r0
                  sub  r0, r0, #32

firstLoop:
                  str r2, [r0]
                  add r0, r0, #0x4
                  cmp r0, r1
                  bne firstLoop
                  sub  r0, r0, #32

secondLoop:
                  ldr r4, [r0]
                  cmp r4, r2
                  bne exit
                         .
                         .
                         .
onepattern:   .word 0xFFFFFFFF
zeropattern:  .word 0x00000000
```

[3]If the application is trying to run unknown instructions

As presented in Figure 4.9, the user mode, in which 15 registers and the Program Counter(PC) are visible is very important. The Stack Pointer(SP) and the Link Register (LR) are mapped on the register R13 and R14.

Thus, the registers R0-R12 can be used as backup storage, and to access these registers, the test will be implemented in Assembler and not in C language, because accessing these registers from the assembler is very easy and other assembler instructions for multiple (data at RAM) load and store instructions will be used to move the content from and to RAM. The Cortex R4F Architecture implemented in the TMS570 microcontroller uses two instruction sets, ARM and Thumb-v2. During the implementation of the test in the Assembler, only ARM mode compatible instructions are used, with the advantage that the test can be used in other ARM architectures and because the 32 bit width registers are used as storage with 4 byte access instructions to load/store the content of the RAM memory.

4.4.2 Watchdog test

In the implementation, the watchdog is used to monitor the program flow. It runs in conjunction with the FreeRTOS Kernel and the application as designed in Section 3.4. A watchdog reset is carried out periodically within the window size of 200 ms. The testing results of the current tested sector are displayed on the LCD display.

The application task has the priority of 4. Only the watchdog task has higher priority because in the prototyping board it is not possible to switch it off.

In the running state, the application switches all LEDs present on the board and resets the watchdog inside the window.

```
 xTaskCreate( Application,
              (signed  portCHAR *)"Task Application",
              250,
              (void*)appText,
              4,
              &xAppHandle );
```

After the application task is finished, the next task in ready state is selected by the OS to run. Because the watchdog has the highest priority, it will switch to the watchdog task and continues to reset the watchdog.

Inside, the lower windows have to be careful not to reset the watchdog, as this would reset the complete board.

The timing diagram as illustrated in Figure 4.10 contains two windows:

- Lower window frame with max 10 ms width
- Window frame with min 200 ms

```
xTaskCreate( Watchdog,
          (signed portCHAR *)"Watchdog Task",
          250,
          (void*)textWatchdog,
          8,
          &xWatchdogHandle );
```

The window watchdog can be used as a monitoring circuit for the supply voltage and as window watchdog which can be employed to monitor the program flow of the application. The main problem with the PFM is how to perform the check if all steps are performed and that the specific operations are performed within a defined time.

Consequently, if critical operations are to be performed, they have to be enclosed in the watchdog monitoring because delays in response of a critical operation can lead to unexpected results, and so the program flow is monitored.

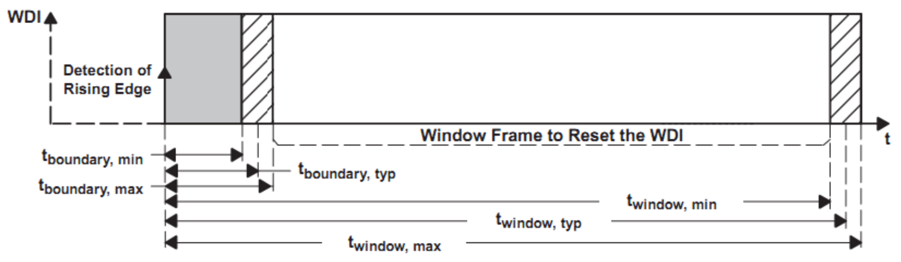


Figure 4.10: *Timing diagram of the watchdog [PSC]*

4.4.3 CPU test

During the start-up, the test is carried out with the maximal number of intervals. At run-time as described in Section 3.4, the core test should run periodically into a task with maximal 16 intervals, because the test must stay under 0.5 ms. The following points must be considered, in order to successfully run the core test from the Self Test Controller (STC) module:

- Backup application registers
- Run the test in intervals
- Restore the application registers

These are the basic requirements to run the test periodically. From practical point of view, it is not clearly specified which registers should backup and where. To integrate the test into the concept with the OS kernel, the registers are stored into the RAM, because the access times to write and read from RAM are negligible. For the following registers a backup is performed:

- All general purpose R0-R15 and CPSR register
- All coprocessor including the banked registers for different operating modes as presented in Figure 4.9
- Memory protection registers and optionally, all FPU registers

To backup the registers in different modes, the M[3:0] bits of the CPSR register as in Figure 4.11 are changed with appropriate values to switch between user, FIQ, IRQ, Supervisor, and Undef modes as illustrated in Figure 4.9.

The task that executes the core test is created with the following parameters:

```
xTaskCreate( CoreTest,
             (signed portCHAR *)"Core Task",
             250,
             (void*)textCoreTest,
             2,
             &xCoreTestHandle);
```

The test is subdivided into intervals and is executed in the background of the application. The core test has the lowest priority compared to all other tasks implemented in the concept. The reason for such a decision is that the test must run only if all other tasks are in idle mode to prevent that one of the tasks becomes ready and cannot switch to running state.

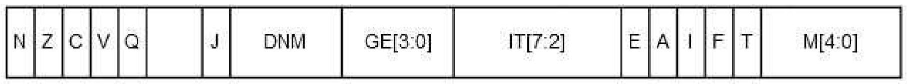


Figure 4.11: *Format of the CPSR Register [CPS]*

After each switch, a backup is performed for the banked registers in that mode. After the core test is finished the same procedure is applied but in reverse order to restore the content of the registers. For the other bits in the CPSR, refer to the link [CPS].

The complete test sequences are presented in Figure 4.12. After the core test is started, the next step is to configure the coretest. Then, a backup of all registers is performed, and the values are stored in the RAM memory. After the test is finished, it generates a reset which is captured by the start-up activity as described in Figure 4.5. Then, the program flow continues with restoring all registers values which are previously stored in the RAM.

The MPU is reinitialized, because it is a static configuration which does not change at run-time and it is overhead to perform a backup of MPU registers and restore them afterwards. The configuration values stay the same.

4.5 Fault Injection, Test Cases and Timing Results

The push buttons as presented in Figure 4.13 can be used to inject faults into the RAM. The buttons can be accessed by configuring the corresponding GPIO's. Externally, they can be always pushed, and internally, they generate a random address and corrupt the content of the RAM-ECC at that address. A task is created to generate the random address and read the GPIOs.

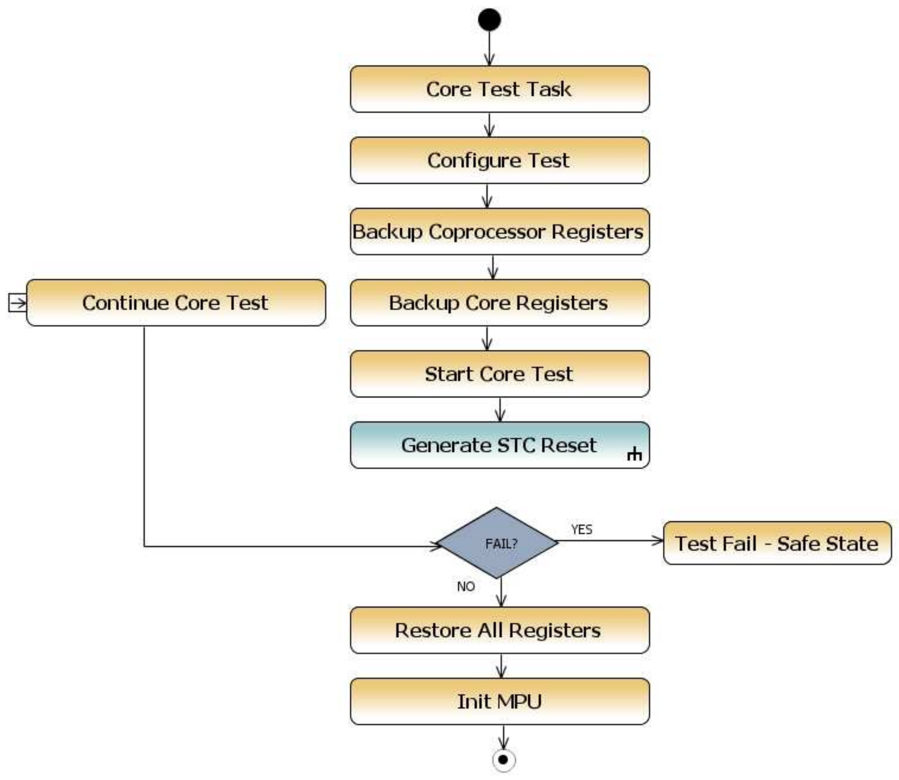


Figure 4.12: *UML activity diagram of the core test*

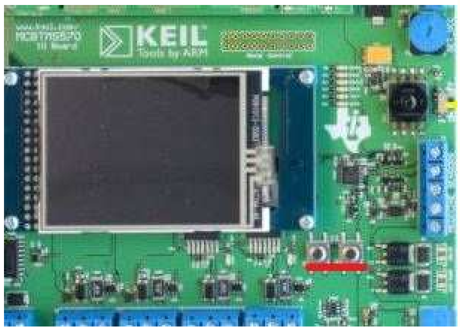
```
xTaskCreate( FaultInjection,
             (signed portCHAR *)"Task Fault-Injection",
             500,
             (void*)ramFaultTxt,
             3,
             &xFaultInjectionHandle );
```

In the Cortex R4F core, the ECC is calculated for every read/write access in the RAM address space. The functionality of the ECC detection reporting is encapsulated in the two TCRAM Wrappers. The TCRAM wrapper serves as an interface support for error detection/correction and decoding.

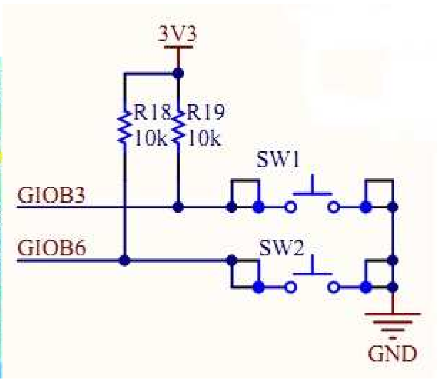
After one of the push buttons is pressed in the Wrapper, the reporting of the detection faults in enabled as presented in Figure 4.15. In the display is written, in which sector the error is introduced.

Reaching the corrupted data sector, a data fault is generated and as impact, all tasks running are stopped, and, optionally dependent on safety requirements all peripherals are stopped. After this fault is caused, in the prototyping board, the two error leds are set to high as in Figure 4.15.

The activities of the RAM fault injection are presented in Figure 4.14. Every time the task takes the CPU, it checks if the buttons are pressed; if yes, it randomly generates a



(a) *Push buttons*



(b) *Circuit diagram*

Figure 4.13: *Push buttons used for the Fault Injection*

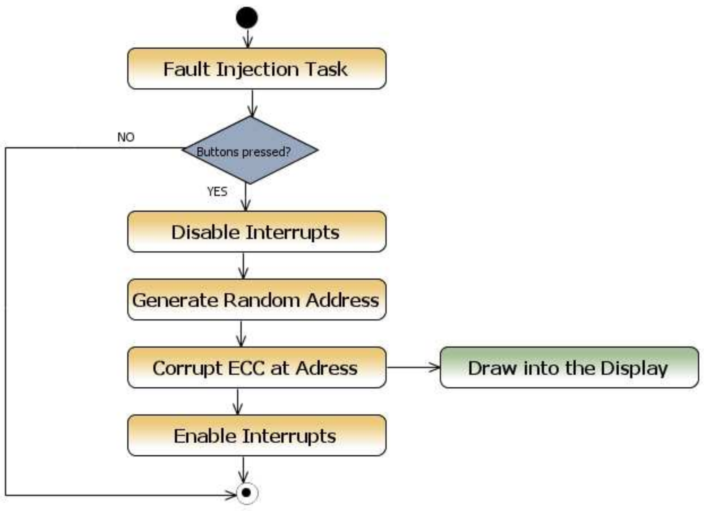


Figure 4.14: *Fault injection activity diagram*

RAM address, corrupts the ECC sum, and changes to sleep state.

The following test cases for the Watchdog test are performed:

- Increasing the time between operations
- Performing operations that take more than 200 ms

In Table 4.1, the timing results of the implemented online test strategy are presented:

The test cases for the watchdog are performed to see whether the evaluation board needs to be reset. This is used if the application waits more than 200 ms for an answer of a critical function.

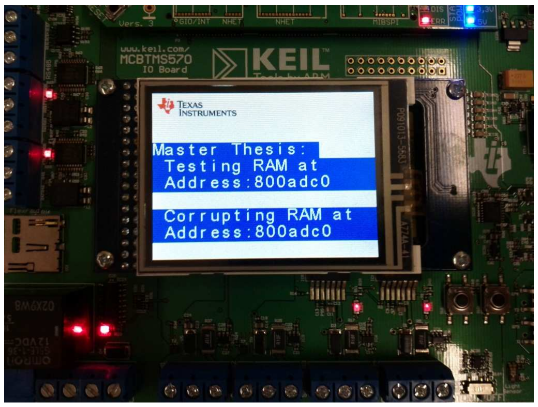


Figure 4.15: *LCD displaying the address of the corrupted RAM region*

	RAM test	Core test	Watchdog
Complete test	30 ms[a] [b]	0,927 ms	100ms
Testing a sector-interval	5,86 µs	0.029 ms	–

[a]Without counting the OS Context switch time
[b]At 160 MHz system clock

Table 4.1: *Timing properties of the tests*

Test cases for the STC test which can be performed to verify the functionality of the module are:

- Set time out counter preload register to a low value
- Initializing the core registers at start-up with wrong values

With the first test case, it is assured that the test is performed within a defined number of CPU cycles. If the test is not finished within the defined number of cycles, a timeout will be generated and it guarantees that the application does not have to wait forever if the test hangs up. The second test is based on the property of the cores, which should be initialized with a defined procedure as described in [RIT] to avoid that at start-up a core compare error is generated.

Chapter 5

Conclusions and Outlook

5.1 Conclusions

At the beginning of this thesis safety-related architectures and self-testing techniques were introduced. The beginning was at the hardware level which includes safety strategies and architectures with a description of the ideas behind such concepts and also the requirements these systems must meet to maintain the functional safety.

The form of hardware strategies is dependent on the fault model. A detailed description of such strategies is given, which can be in a form of single and multi-distributed controllers. From a practical point of view, these strategies are concepts which form the base for architectures and are described in detail in the second chapter.

In order to assure the safety in a microcontroller, algorithms are needed to be implemented, which can be verified either using the mathematical structure or simulation. In the thesis, different algorithms are presented, which can be used in a system to find different types of faults like coupled, stuck-at, transition faults etc. which can be either implemented in hardware as part of an architecture or in software as software-based self test. However, the main focus was on functional tests.

In order to evaluate such architectures on how they fulfil safety criteria, a list of evaluation criteria was created. Based on the safety features, performance, run time tests, developing environment etc, the TMS570 microcontroller was chosen as the appropriate device for further implementations.

The idea was to use safety features of the microcontroller, and different algorithms to be mixed in an online periodic testing strategy which assures safety in critical applications. It was shown in the thesis, which are the main safety features of the chosen microcontroller and which are implemented, the online RAM test and watchdog test running during normal system operation.

An operating system kernel was used to run all the tests, and it was shown how the tests can be integrated into a safety strategy, presenting the advantages and disadvantages of using such a system. As for the performance, the RAM test, which is implemented completely in assembler, can be used by any ARM microcontroller with or without an operating system.

A microcontroller which has not only off-line safety features (tests that can be performed only at startup) but online features as well, which tests the microcontroller's functional parts like RAM, CPU, flash, would extend the possibility for critical applications that could be implemented in the automotive industry or power plants, medical life support equipment etc.

5.2 Outlook

The TMS570 microcontroller offers a variety of tests and safety features which can be used during the development. As presented in the design chapter of the concept, it has some drawbacks because it does not have tests for every part of the system which can be performed at runtime. All RAM tests can be performed only at start-up, and the RAM needs to be extended either with hardware or with software tests for such purposes.

Such an integration would be a powerful feature for all applications in that field. Not only by extending such tests, but also by creating a better OS support for the safety features like the core test, as maintaining and backup the different registers in software to perform the test periodically is error prone because of timing requirements.

There are new microcontrollers with safety features which have more computing cores and adequately address the requirements of safety-critical applications. They will be used in the automotive industry in the near future.

Appendix A

Definitions

A.1 Abbreviations

ATPG	Automatic Test Pattern Generation
AF	Address Decoder Faults
ALU	Arithmetical Logical Unit
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
ASIL	Automotive Safety Integrity Level
BIST	Built in Self Tests
CAN	Controller Area Network
CCF	Common Cause Failures
CPU	Central Processing Unit
CPSR	Current Program Status Register
CF	Coupling Faults
CFin	Inversion Coupling Fault
CFid	Idempotent Coupling Fault
CFst	State Coupling Fault
CU	Comparing Unit
DC	Diagnostic Coverage
DRF	Data Retention Fault
DSP	Digital Signal Processor
EEPROM	Electrically Erasable Programmable Read-Only Memory
ECC	Error Correction Codes
ESM	Error Signaling Module
ETM	Embedded Trace Macrocell
FMzPLL	Frequency modulated phase lock loop
FPLL	Flexray phase lock loop
FSM	Finite State Machine
FPU	Floating Point Unit
FMEA	Failure Modes and Effects Analysis
GALPAT	Galloping Pattern
GUI	Graphical user interface

HALCoGen	Hardware Abstraction Code Generator Layer
HET	High-End Timer
ISA	Instruction Set Architecture
IDE	Integrated Development Environment
ITI	Institute for Technical Informatics
JTAG	Joint Test Action Group Interface
LBIST	Logical Built-in Self Tests
LIN	Local Interconnect Network
MIPI	Mobile Industry Processor Interface
MEPAS	Methods and processes for automotive embedded software development, verification and validation
MPU	Memory Protection Unit
MCU	Microcontroller Unit
NowECC	Error Correction Code Generator
PWM	Pulse Width Modulation
PFM	Program Flow Monitoring
PCP	Peripheral Control Processor
PBIST	Programmable Built-In Self Test
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
RTOS	Real Time Operating System
RTL	Register Transfer Level
RJ45	Registered Jack
SRS	Safety Requirements Specification
SIL	Safety Integrity Level
SAF	Stuck-at Faults
SFF	Safe Failure Fraction
SMP	Symmetric multiprocessor architectures
SOF	Stuck-open fault
STC	Self Test Controller
SFF	Safe Failure Fraction
SBST	Software Based Self Tests
TCRAM	Tightly Coupled Random Access Memory
TI	Texas Instruments
TLP	Thread Level Parallelism
TF	Transition Faults
ThWD	TTOSHIBA hW diagnostic
TFT-LCD	Thin Film Transistor Liquid Crystal Display
WALPAT	Walking Pattern

A.2 Used Symbols

λ_S	Rate of Safe Failures
λ_D	Rate of Dangerous Failures
λ_D	Rate of Dangerous Failures that are Detected
$\Downarrow$	Decreasing Addressing Order
$\Uparrow$	Increasing Addressing Order
$(w0)$	Write Zero to the Specific Ram Cell
$(w1)$	Write One to the Specific Ram Cell
$(r0)$	Read Zero to the Specific Ram Cell
$(r1)$	Read One to the Specific Ram Cell
$\downarrow$	Forced Transition from 1 to 0
$\uparrow$	Forced Transition from 0 to 1

Bibliography

[AGPP09] Andreas Apostolakis, Dimitris Gizopoulos, Mihalis Psarakis, and Antonis Paschalis. Software-Based Self-Testing of Symmetric Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 58:1682 – 1694, 2009.

[APG+09] Andreas Apostolakis, Mihalis Psarakis, Dimitris Gizopoulos, Antonis Paschalis, and Ishwar Parulkar. Exploiting Thread-Level Parallelism in Functional Self-Testing of CMT Processors. *European Test Symposium*, pages 33 – 38, 2009.

[ARM] ARM architecture reference manual.

[ARM11] *ARM Optimizing C/C++ Compiler v4.9*, August 2011.

[Bar10] Richard Barry. *Using the FreeRTOS Real Time Kernel*. 2010.

[BBC+08] Alfredo Benso, Alberto Bosio, Stefano Di Carlo, Giorgio Di Natale, and Paolo Prinetto. March Test Generation Revealed. *IEEE Transactions on Computers*, VOL. 57, NO. 12, 2008.

[BCM07] Gabriele Boschi, Federico Colucci, and Riccardo Mariani. Using an innovative SoC-level FMEA methodology to design in compliance with IEC61508. *Design, Automation & Test in Europe Conference & Exhibition*, 2007.

[BFM+03] Massimo Baleani, Alberto Ferrari, Leonardo Mangeruca, Allberto Luigi Sangiovanni Vincentelli, Maurizio Peri, and Saverio Pezzini. Falut-Tolerant Platforms for Automotive Safety-Critical Applications. *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 170 – 177, 2003.

[BHW06] Ismet Bayraktaroglu, Jim Hunt, and Daniel Watkins. Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues. *Test Conference, ITC '06. IEEE International*, pages 1 – 7, 2006.

[BSE07] Simon Brewerton, Rolf Schneider, and Denis Eberhard. Implementation of a Basic Single-Microcontoller Monitoring Concept for Safety Critical Systems on a Dual-Core Microcontroller. In *SAE International-Safety Critical Systems, Paper Number: 2007-01-1486*, 2007.

[CPS] *Cortex-R4 and Cortex-R4F Technical Reference Manual http://www.arm.com/products/processors/cortex-r/index.php.*

[CRRD03] Li Chen, Srivaths Ravi, Anand Raghunathan, and Sujit Dey. A Scalable Software-Based Self-Test Methodology for Programmable Processors. In *Proc. 40th Design Automation Conference*, pages 548–553, 2003.

[CWLG07] Chung-Ho Chen, Chih-Kai Wei, Tai-Hua Lu, and Hsun-Wei Gao. Software Based Self-Testing with Multiplie-Level Abstractions for Soft Processor Cores. *IEEE Very Large Scale Integration Systems (VLSI)*, 15:505–517, 2007.

[FM07] Peter Fuhrmann and Riccardo Mariani. Comparing fail-safe microcontroller architectures in light of IEC 61508. In *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, 2007.

[Goo93] Ad J. Van De Goor. Using March Tests to Test SRAMs. *IEEE Design and Test of Computers*, vol. 10 no. 1:8–14, 1993.

[GVA06] Sankar Gurumurthy, Shobha Vasudevan, and Jacob A. Abraham. Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. In *IEEE International Test Conference*, pages 1 – 9, 2006.

[HA99] Mohamed S. Hefny and Hassanein H. Ammer. Design of an improved watchdog circuit for microcontroller-based systems. *ICM'99. Eleventh International Conference on Microelectronics*, pages 165 – 168, 1999.

[Has] Atsushi Hasegawa. R&D strategy: Using various types of multi-core architectures to overcome limitations of single-core microcomputers. Technical report, RENESAS.

[KEI] http://www.keil.com/support/man/docs/mcbtms570/.

[KLC$^+$02] Angela Krstic, Wei-Cheng Lai, Kwang-Ting Cheng, Li Chen, and Sujit Dey. Embedded Software Based Self Test for Programmable Core Based Designs. *IEEE Design & Test*, 19:18–27, 2002.

[KPGX05] Nektarios Kranitis, Antonis Paschalis, Dimitris Gizopoulos, and George Xenoulis. Software-based self-testing of embedded processors. *IEEE Transactions on Computers*, 54 Issue: 4:461 – 475, 2005.

[KS06] Thomas Kottke and Andreas Steininger. A Reconfigurable Generic Dual-Core Architecture. *International Conference on Dependable Systems and Networks*, pages 45 – 54, 2006.

[LCD$^+$05] Eldon G. Leaphart, Barbara J. Czerny, Joseph G. D'Ambrosio, Christopher L. Denlinger, and Deron Littlejohn. Survey of Software Failsafe Techniques for Safety-Critical Automotive Applications. *SAE 2005 World Congress & Exhibition, April 2005, Detroit, MI, USA, Session: Safety-Critical Systems (Part 1 & 2)*, 2005.

[Mar07] Riccardo Mariani. Applying IEC 61508 to Integrated Circuits. Technical report, 2007.

[MB07] Riccardo Mariani and Gabriele Boschi. A systematic approach for Failure Modes and Effects Analysis of System-On-Chips. *13th IEEE International On-Line Testing Symposium*, pages 187 – 188, 2007.

[MKS10] Riccardo Mariani, Thomas Kuschel, and Hiroshi Shigehara. A flexible microcontroller architecture for fail-safe and fail-operational systems. Technical report, 2010.

[Mon99] Sergio Montenegro. *Sichere und fehlertolerante Steuerungen - Entwicklung sicherheitsrelevanter Systeme*. Carl Hanser Verlag München Wien, 1999.

[NTA78] R. Nair, Satish M. Thatte, and Jacob A. Abraham. Efficient Algorithms for Testing Semiconductor Random-Access Memories. *IEEE Transactions on Computers*, pages 572 – 576, 1978.

[PG04] Antonis Paschalis and Dimitris Gizopoulos. Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2004.

[PGH$^+$06] Mihalis Psarakis, Dimitris Gizopoulos, Miltiadis Hatzimihail, Antonis Paschalis, Anand Raghunathan, and Srivaths Ravi. Systematic Software-Based Self-Test for Pipelined Processors. 2006.

[PGK$^+$01] Antonis M. Paschalis, Dimitris Gizopoulos, Nektarios Kranitis, Mihalis Psarakis, and Yervant Zorian Zorian. Deterministic Software-Based Self-Testing of Embedded Processor Cores. In *Design, Automation and Test in Europe. Conference and Exhibition*, pages 92 – 96, 2001.

[PGSR10] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. Microprocessor Software-Based Self-Testing. *IEEE Design and Test of Computers*, vol. 27, no. 3:pages 4–19, 2010.

[PML02] Praveen Parvathala, Kaila Maneparambil, and William Lindsay. FRITS - A microprocessor functional BIST method. *ITC '02 Proceedings of the 2002 IEEE International Test Conference*, pages 590 – 598, 2002.

[PSC] *Processor Supervisory Circuits with Window-Watchdog "http://www.ti.com/lit/ds/symlink/tps3813l30.pdf"*.

[RIT] *Recommended Initializations for TMS570 Microcontrollers - http://www.ti.com/lit/an/spna119/spna119.pdf*.

[RSCS04] Matteo Sonza Reorda, Giovanni Squillero, Fulvio Corno, and Ernesto Sanchez. Automatic Test Program Generation: A Case Study. *Design & Test of Computers, IEEE*, 21 Issue: 2:102 – 109, 2004.

[SA98] Jian Shen and Jacob A. Abraham. Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation. *Test Conference Proceedings, International*, pages 990 – 999, 1998.

[SD06] Padma Sundaram and Joseph G. D'Ambrosio. Controller Integrity in Automotive Failsafe System Architectures. In *Safety Critical Systems, Special Publications Paper Collections, SAE International, Product Code: SP-2029*, 2006.

[TMS] *TMS570 Technical Reference Manual.*

[TP07] Thomas Tamandl and Peter Preininger. Online Self Tests for Microcontrollers in Safety Related Systems. In *5th IEEE International Conference on Industrial Informatics*, pages 137 – 142, 2007.

[Tum09] Peter Tummeltshammer. *Analysis of Common Cause Faults in Dual Core Architectures.* PhD thesis, Technische Universität Wien - Fakultät für Informatik, 2009.

[WWC$^+$05] Charles H. P. Wen, Li-C. Wang, Kwang-Ting Cheng, Kai Yang, Wei-Ting Liu, and Ji-Jan Chen. On a software-based self-test methodology and its application. In *VLSI Test Symposium, 2005. Proceedings. 23rd IEEE*, pages 107 – 113, 2005.

[WWW06] Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. *VLSI Test Principles and Architectures - Design for Testability.* Morgan Kaufmann; 1 edition, 2006.