Politecnico Di Torino

Master's Degree in Electronics Engineering

Embedded Systems

Master's Degree Thesis

Deterministic Cache-Based Execution of On-Line Self-Test Routines in Multi-Core Automotive System-On-Chips

Supervisor

Dr. Edgar Ernesto Sánchez Sánchez

Candidate

Tzamn Meléndez Carmona

Co-supervisors

Andrea Floridia

Davide Piumatti

Annachiara Ruospo

December 2021

Contents

Abstract

In this thesis, the author wants to discuss the problematics that arouse while working with the Self-Test procedures that are conducted online in a multicore context. Self-Test procedures have not been considered until now for their use on caches-based systems due to the lack of proof of their deterministic behavior when executing instructions. Execution in cache and deterministic are concepts considered exclusive one to each other. If we add to this conception the execution of those procedures on multicore context, the bus contention produces a strong timing unpredictable system. Having Online Self-Test procedures being executed under those conditions might lead to a variation in the fault coverage or the failure of the software test. This thesis presents a brief description of the behavior of a test procedure in single core and multicore. There is proposed a cache-based strategy for achieving both deterministic behavior and stable fault coverage from the execution of self-test procedures for multi-core systems. The proposed strategy is applied to two representative modules of the microprocessors that are affected by the issues of executing self-test software in a multi-core execution: synchronous imprecise interrupts logic and pipeline hazard detection unit. The experiment results demonstrate that it is possible to attain a stable execution whilst also improving the state-of-the-art approaches for the on-line testing of embedded microprocessors. The level of effectiveness of the methodology was established on all the cores of a multi-core industrial System-on-Chip (SoC) intended for automotive ASIL D applications.

Introduction

Software Test Libraries (STLs) have become one of the preferred solutions for online testing of automotive processors SoCs. The cost of the STLs is lower in comparison with other test solutions due to the fact that they are a totally software based, with a high effectiveness [1], [2], [3], [4], [5], [6], [7]. STLs are composed of tens of self-test procedures, written in source code, that look for the occurrence of permanent faults. In other words, their goal is to identify when some misbehavior occurred in the SoC. This approach was introduced by Satish M. Thatte and Jacob A. Abraham in [8]. In the paper the authors proposed the idea of using a so-called test signature. This approach involves running a test program with some specific patterns as inputs where the results are accumulated in a variable (test signature). Afterwards, the test might be either pass or fail depending on the result of the comparison of the test signature with an expected signature. The expected test signature is obtained when the test is run in context, where there are no failures present (fault free scenario).

This thesis consists of four parts. The first part involves reviewing the concepts that are related to software testing and their applications in SoCs. What follows, an explanation of a the importance of a test signature in a STL and why by means of the test signature, it is a safe way to detect the occurrence of faults [9], [10], [11], [12]. Furthermore, in the first part of the thesis there is explained the Self-test procedures and how they are categorized in two; boot-time and run-time tests [2], [7]. Run-time testing has the advantage of being executed when the microprocessor is in standby. However, the author of this thesis focuses on boot-time test since it requires some special considerations [13].

The author describes the importance of STLs and why they must comply with the typical requirements of the embedded software in the automotive industry. Since STLs run concurrently with operating system or an application program, the resources that are available to the programmer are limited. The other important fact is that nowadays most of the new high performance SoCs that are released in the market are based on multicore architectures. This implies a new challenge for software based self-test (SBST) techniques. SBST must be adapted to these new architectures since

most of the test software that has been developed, has been focusing single core architectures [10] and that is why the importance of having parallel testing is discussed in this paper as well.

For better understanding of the units under test discussed in this work, there is reviewed the general architecture of a typical 5 stages MIPS microprocessor with its forwarding unit, the ARM Cortex-A8 and its instructions buffer, due to its similarities with our case of study. The final important concepts that are developed in detail in the first part of this work is the principle of locality and cache memories since they are an important part of the solution that concern to this thesis.

When STLs are running in multicore architectures some issues appeared due to the complexity of the architecture. In order to have high effectiveness STLs must be executed uninterruptedly, in particular those test procedures targeting permanent faults in Hazard Detection Units, Control Units, Forwarding Units, etc. A not interrupted execution is one of the main problems that appear when there is switched from a single core architecture to multiple cores. The system suffers from a bus contention, this is an effect that occurs at the time when there are multiple requests in a bus. In this case, in the memory bus. When there is a high bus contention while accessing to the instruction memory, the microprocessor core that has a request must wait until other cores will have been attended. The microprocessor solves this conflict stalling one of the cores at the fetching stage until the bus is available. These affections causes that the test procedure deals with a limited timing predictability in a multicore context [14]. This implies two important consequences on the self-test procedures that require a specific sequence of instructions. The first consequence is fault grading. The fault coverage become uncertain, and it fluctuates depending on which processor is excited depending on the system bus activity. This first consequence affects the test code when it is intended to detect a specific fault in a certain location on the microprocessor. The fact that it does not execute where it should be results in the non-detection of the intended faults. The second important consequence is test signature instability. The signature that is generated by the test program cannot identify the intended fault safely. When the test expects to have a set of instructions in a certain order or has to be finished the test in expected time, and it does not, clearly it will exist a mismatch in

the signature. The signature does not have a steady value and cannot be reliable.

The main intent of this thesis is to propose a deterministic methodology for executing in-field self-test routines in a multicore scenario. The solution is based on the use of cache memories. By means of the use of cache memories in a loop structure, it guarantees a stable signature and deterministic fault coverage, in specific for those test procedures that target CPU modules where the time instability affects their fault coverage negatively. The methodology does not require significant modifications to the single core STLs algorithms. The cache-based methodology exposes better response at the time of testing in parallel all the cores of a multicore architecture than test in which each core is treated independently. This methodology is purely software and neither does it require any additional on-chip hardware, nor an external device, that is a good solution for online-testing. The use of cache memories as a solution for testing multicore SoCs has been researched previously. Those previous solutions store the self-test procedures and the test responses by share on-chip memory in the multicore system. Yet the solution targets end-of-manufacturing testing [15]. Meanwhile, the cache-based proposed in this work reduce the test application time and there is no need of off-chip memory accesses. Those other methodologies use an external device to load the test into the cache through an external tester and they can only be applied for the end-of-manufacturing testing and only when the device is in the factory. In the cache-based proposed in this work the test is loaded in cache by the test itself and it can be used in the field. There exists a methodology that uses a cache-aware test scheduler which takes advantage of the memory hierarchy for speeding-up the run-time tests [16]. However, it does not deal with the in-field execution of boot-time procedures. Other work that analyses a parallel execution of boot-time tests is proposed in [13]. The paper proposed some scheduling alternatives considering conflicts that occur when testing in shared resources. Nevertheless, the determinism of the self-test procedures in a multi-core scenario is a statement that has not been covered previously in other works so far.

This document is organized by chapters. The second chapter starts with a brief description of Testing on SOCs. This is where the general architecture of pipelined microprocessors is explored followed by a

description of a pipelined microprocessor control unit and exceptions. The last issues exposed in the Chapter 2 are cache memories and the concept of principle of locality. The third chapter indicates all the problems that arise as soon as STLs are executed in multicore architectures and their consequences in testing reliability. The fourth chapter presents the author's proposed methodology that solves the issues exposed in the third chapter. The chapter 5 focuses on the results of the methodology applied in a real case multicore microprocessor. The methodology is compared with other existing solutions. The last part of the thesis is the author's conclusions.

Chapter 2

Literature Review

In this chapter is reviewed all the literature that is required for the understanding the topics exposed on this work. The chapter begins with a brief description of what is testing on SOCs and the importance of testing in the vehicle industry. Following by an explanation of SBST. The concept of boot time diagnostics and run time diagnostics are introduced in this chapter as well. For a general understanding of the unit under test the author exposes the general functionality of the MIPS microprocessor and its forwarding unit. Another microprocessor that is explored is a double issue ARM microprocessor (Cortex-A8) due to the similarities with the case of study. The last concepts that are reviewed are cache memories and the concept of principle of locality because of their importance on the proposed solution of this thesis.

Testing on System on Chips.

Nowadays the execution of software online is part of our lives in regular basis, more specific in the automotive industry, the use of STLs is present in most of the new vehicles produced today. This kind of software is executed into SoCs integrated in the cars and have become extremely important because they increase the safety of the system. STLs assure the correct functionality of the SoCs a prior or when the vehicle is in use. Assuring a correct functionality of the SoC is crucial due to the number of components that are managed by them. *"The role of testing is to detect whether something went wrong"* [17].

The consequences of skip Testing can carry different problematics in the future, depending on the severity of when and where the misbehavior might occur. As example, supposing there is a failure within the SoC, if the failure happens in a signal that shows whether the stereo is switched on or

off, the consequences of this will not produce a significant effect. Meanwhile, if a failure happens in a signal that activates the brake system, the consequences directly affect the lives of one or some human beings, and the result can be catastrophic. Testing helps to prevents this kind of scenario and gives a certain level of confidence when the device is released from the factory.

Online testing based on STLs, has been adopted for testing the device when it is in use. STLs are basically a piece of code written in assembly language that are designed for being executed looking for the presence of certain faults. [2]. For a better understanding of those procedures and for the proposed methodology on **Chapter 4** is important to briefly explain this concept.

There are different approaches for testing, which are: Software-Based Self-Test (SBST) which is based on STLs and Built-In Self-tests (BIST) [1]. BIST it is not discussed by the author of this thesis. The topic that concerns to this thesis is SBST since the proposed methodology of this thesis is SBST based on STLs. SBST based on STLs can be found in the industry such as [2], [3], [4], [5] and [6], which is an starting point for propose a solution to the problem that is presented later in this paper.

SBST was originally proposed by Satish M. Thatte and Jacob A. Abraham in [8]. What they proposed was to apply a specific set of patterns generated by software instructions. Later to observe the results of those instructions and accumulate what results in a variable. This variable is called test signature. Subsequently the test signature is compared with a beforehand computed value (expected test signature). The result of the comparation is used to determine whether it is a test pass or test fail. The expected test signature is well known since its computation is obtained by means of the execution of the test procedure in a fault free scenario.

Satish M. Thatte and Jacob A. Abraham describe the procedures in 4 steps:

Step 1:	*Initialize the queue **Q** with all the registers so that **Ri** lies ahead of **Rj**, if and only if l(**Ri**) < l(**Rj**); Initialize the set **A** as empty.*
Step 2:	***A** register at the front of **Q**; Update **Q**.*
Step 3:	*REPEAT*
Step 4:	*Repeat Steps 1, 2, and 3 with complementary data.*

Source [8]

If we look forward into the literature, until now there is no other method for the representation of the occurrences of faults within this scenario, if it is not done with a test signature such as the one described in the sources [9], [10], [11] and [12].

According to the literature [2] and [7], Self-test procedures are divided in two main categories:

- ***Boot-time (Boot diagnostics)*** and run-time tests (Run time diagnostics). The first one is executed once, at the time that the system starts or goes online, with the purpose of interfere as less as possible with the real-time execution.
- ***Run time diagnostics*** is executed while the system running in its normal operation. In this one before the test is executed the data or the context should be stored in order not to destroy any important information.

It is important to highlight that some Boot diagnostics programs to have a good accuracy require to execute the instructions in a certain order and being performed uninterrupted. This without withdrawing that; the STLs must fulfil all the requirements included in any typical embedded software, this last statement is important since STLs works concurrently with any other application programs and with an operating system. Therefore, when working in testing libraries, the programmer should be extremely careful since the access to resources such as data memory and namely code are limited, this means that the test code must be efficient in terms of memory.

Technology evolves fast through the time, this implies that as soon as a new microprocessor appears in the market with more complex architectures, SBST must evolve at the same time. Although we are in a time

where SBST has reached important advances in the industry [10], still there is a large field of improvement.

Currently, most of the devices coming out to the market every year are multicore and naturally they are used in the automotive industry, however the Self-Test in those multicore architectures has not developed totally since most of the Test software available is designed to single core architecture, that is why most of the research in SBST are focused now on how to test effective those new architectures. SBST for multicore is not being carried out faster as the single core, due to those new devices bringing new problematics as a result of their design complexity. Those microprocessors came out with dynamic instruction execution, multithreading, GPUs, multicores, etc. [10], concepts that were not included before.

General Architecture of a Pipelined Microprocessor

For Testing any kind of device, it is important to know its functionality and as far as possible the details of components that assemble it, for this reason is taken as a base other analysis available in the literature. In the following section is described in general a single core pipelined microprocessor **MIPS**, it is a load-store word microprocessor with a Harvard memory architecture structure. Branches are present in the microprocessor whenever the result of registers operations are equal to zero and manage integer Arithmetic Logic Unit operations [18], making emphasis of the forwarding unit functionality and the control unit. In addition, a general architecture of dual issue pipelined microprocessor ARM Cortex-A8 is discussed, both analyses are present in [18]. There are emphasized those units because are the ones involved in **Chapter 3** and **Chapter 4**. The final architecture presented in this part of the work is the SPC58EC SOC, which it is a SOC targeted for the automotive industry. This last SOC is a real example and give the reader an idea of the case of study discussed in the **Chapter 5.**

The simplest version of the MIPS is a RISC microprocessor which consists in 5 stages instruction pipelined, Instruction Fetch Unit, Instruction Decode Unit, Execution Unit, Memory Access Unit and Writeback Unit. The pipelined

data path is shown in ***Figure 2.1*** and the summary of what is done in each stage is explained below.

- **Fetch stage:** During this stage the Instruction word (IW) is read from the instruction memory, and the Program Counter (PC) receive the address of the next instruction added by 4.
- **Decode Stage:** This stage is *fixed-field decoding* [18] Here is where the Register File (RF) is located and the IW is decoded, depending on the type of Instruction, one or two register sources are being read. The sing extended unit is contained in this stage, this is needed for the immediate operation, and finally there are two more components: One Adder which performs the immediate Jump and the Conditional Test Unit, where the out of the register file is compared in order to perform a Brach.
- **Execution Stage:** In This implementation it only consists of the Arithmetic Logic Unit (ALU), where are performed all the operations that are required including the computation of the Data Memory address. The other component in this stage is a multiplexer which only selects the second operand of the ALU that might come from a register, or the extension unit of the Decode Stage.
- **Memory Stage:** The Data Memory is located here; this microprocessor is Harvard memory structure which implies that Instruction Memory is separated from Data Memory. The target address is computed in the previous stage and this stage just read or write from it.
- **Write Back Stage:** The last step consists in bypassing the value that is obtained either from the Data Memory or the ALU to the RF, this operation is made by a multiplexer.

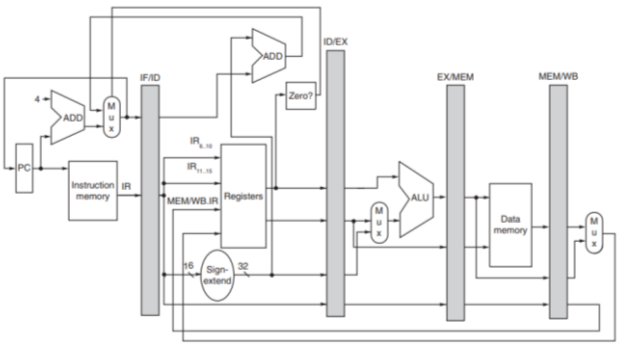

Figure 2.1, Pipelined MIPS Datapath. Image source: [18]

As we can see, it is not the best version of the MIPS because this implementation is not optimal neither in terms of Hardware nor performance, however this gives us a brief idea of the functionality of a current microprocessor used in the automotive industry and for the purpose of this thesis there is no need going deeply into it.

The Forwarding unit consists in a couple of multiplexers added to the data path in the inputs of the ALU, which bypassed other results from other stages, specifically from the Memory stage and the Write Back stage, its implementation is shown in ***Figure 2.2*** The multiplexers are driven by a forwarding logic which its goal is to identify and manage the data dependencies. In order to identify when a data dependency occurs in the steam of instructions, in some cases the forwarding logic is included in a special unit called Hazard Detection Unit. In other cases, the forwarding logic is included within the multiplexer itself, being a modified version of the typical multiplexer.

Data Dependencies are present when an instruction is performed, and its execution depends on previous results that are still being computed at that time. Those are called *Data Hazards* [18]. This implies that the Datapath either reads the wrong data or it must wait until the previous instructions are finished. Stalling the microprocessor directly affects its performance. Certainly, the use of forwarding is a good solution for Data Hazards and is used widely today. This understanding of the multiplexers of the forwarding unit helps to achieve the goal of this thesis.

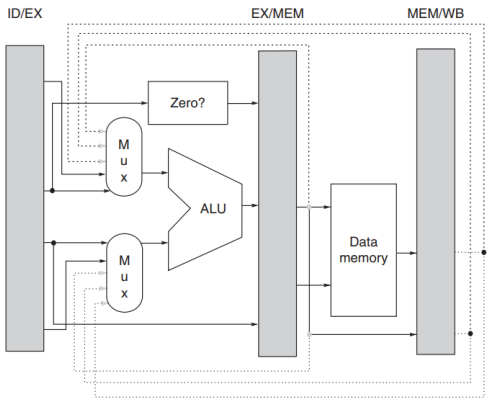

Figure 2.2, Forwarding MIPS Implementation. Image source: [18]

Most of the microprocessors available in the market today are multicore. Which implies that STLs need to do the transition from singlecore to multicore as well. When the change from a single issue to a multiple issue is done, the hardware and the stages of the microprocessor become more complex, the pipeline might require more stages compare with the typical 5 stages proposed on the MIPS. The ARM Cortex-A8 is the perfect example of this last statement, as it has some important characteristics. It is a dual issue microprocessor, it has a static scheduler, and it has dynamic issue detection, concepts that were not included into the basic 5 stages MIPS, furthermore there are 13 stages pipeline. The design of this microprocessor is extremely helpful for the understanding of the behavior of our case due to its similarities.

One of the stages that are worth looking into slightly more is the decode stage. This stage gives us an idea of the functionality of how the instructions are issued in our case of study. In the ***Figure 2.3*** is shown the five-stage instruction decode of the Cortex-A8. It shows how the instruction is decoded from the fetch unit. Up to two instructions can be decoded and then they are stored into the queue, afterwards it goes to the Score board + issue logic where it is decided when the instruction can be issue. It is important for our work to notice that this is the stage where it is decided if the instructions received are branches, or if there are some dependencies that cannot be managed by the forwarding unit.

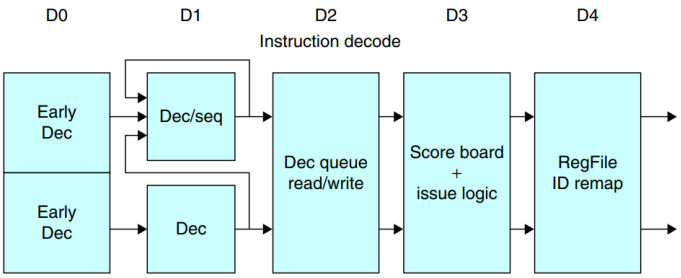

Figure 2.3, The five-stage instruction decode of the Cortex-A8. Image source: [18]

The execution unit of the Cortex-A8 has some important aspects to analize, it consist In 6-stage pipeline, and two instruction can be exacuted at a time, depending of the type of operation. There are two main aspects to highlight important for our analisis, as it is shown in the ***Figure 2.4*** both pipes cannot execute all the instructions, for example the is only one multiplier and is set it in pipeline 0. The other important aspect is that there is comunication among the pipelines, and within the same pipeline, that means that is fully bypassing.

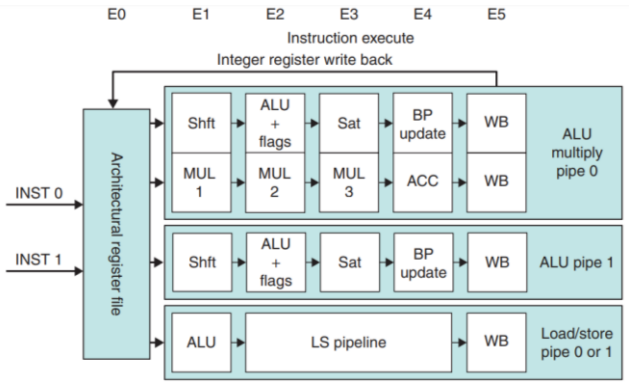

Figure 2.4, The execution pipeline for the Cortex-A8. Image source: [18]

All the previous architectures mentioned in this chapter let the reader know more about the CPUs that are embedded in modern vehicles SOC. What follows, is a brief analysis of a real SOC that is used on the regular basis in the automotive industry. For example, we can choose a SOC from STMicroelectronics, the SPC58EC which is a dual-issue double core SOC, which is part of the line of the General-Purpose Automotive Power Architecture® MCUs from STMicroelectronics [19]. This SOC was designed

to satisfy the ISO 26262 and ASIL-B standards. The architecture is illustrated in the ***Figure 2.5***

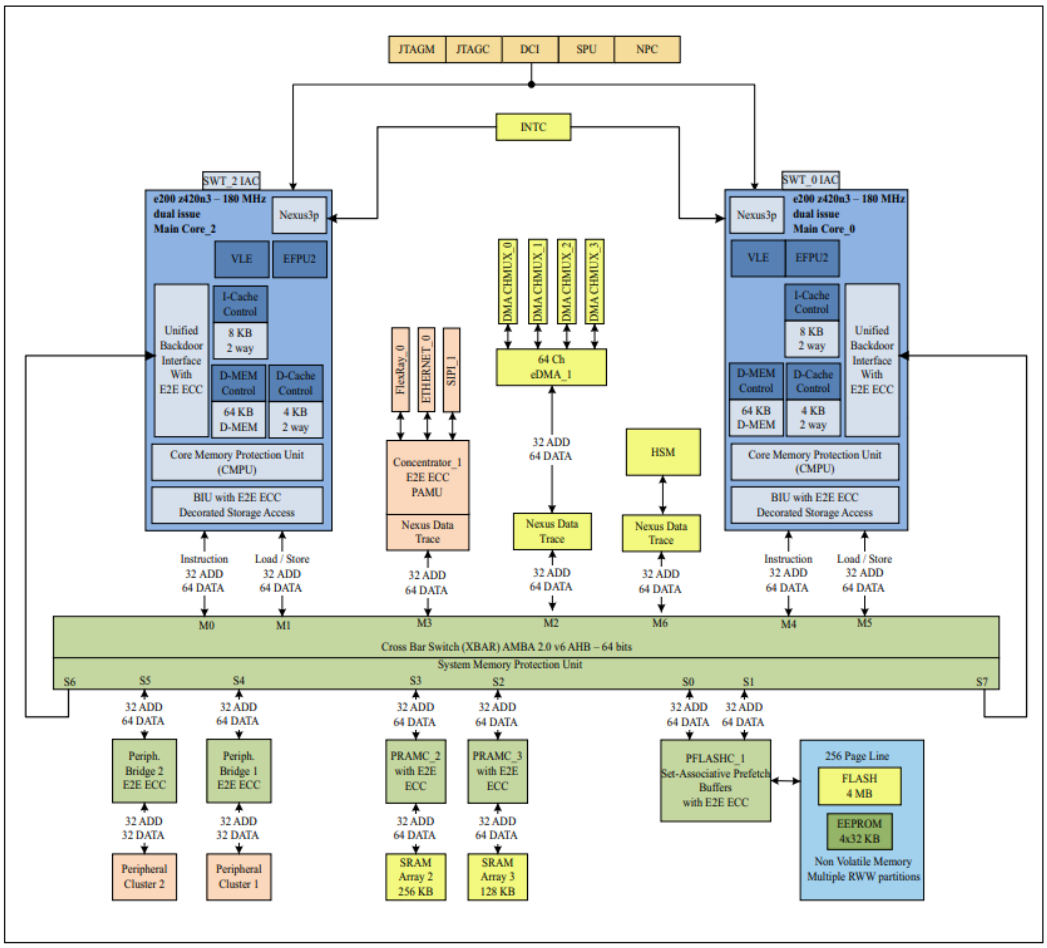

Figure 2.5, SPC58EC SOC used in the Automotive industry [20]

In the ***Figure 2.5*** the reader can observe that the SPC58EC has 2 e200z420n3 cores. The e200z420n3 is a dual issue CPU, which has a similar behaviour as the Cortex-A8 mentioned before. It is important to emphasize the path between the Memory and the cores for fetching an instruction. Every instruction that is executed for each core must be prefetched to the Set-Associative Prefetched-Buffers. Once they are in the buffers the instructions are then assigned to their respective CPU by passed in the crossbar switch. Afterwards, the instructions can be decoded and executed independently in each core. It is important to notice that now the process of assigning the instructions takes some extra clock cycles.

Control Unit and Exceptions

The control unit is the logic that manage and switch all the signal into Datapath depending on which type of that is intended to be executed, this kind of logic take as input the instruction, each instruction has a unique identifier, and the outputs are the signals that activate the components in each stage of the pipelined. It is important to notice that in a pipelined microprocessor such as the MIPS, the signals need to be pipelined as well in order to have the right signal at the corresponding pipelined stage. In the **Appendix 1** shows the instructions and the actions to be perform in each stage for the MIPS.

The control unit is the one in charge to stall the pipelined microprocessor when a hazard is detected, and it cannot be managed by the Datapath. They are 3 types of hazards:

- **Structural hazards:** Those came out when 2 or more instructions get in conflict because they claim the same hardware resource at the same time and the microprocessor is not able to manage it, it is a matter of how the microprocessor is designed, therefore the microprocessor needs to stall.
- **Data hazards:** Those does not depend on the design of the microprocessor, the came out when there is a dependency in the data, that means that when a current instruction claims some hardware resource and the hardware requires a value or values of the instructions that are either in computation at that specific moment or have not been computed yet. As a consequence, the microprocessor needs to stall.
- **Control hazards**: they came out when branch, Jump or any other instructions that modify the Program Counter. The microprocessor stalls here because the instructions that were fetched by the microprocessor before it realizes that a branch need to be performed, they need to stop their execution. [18]

Hazards are not the only reasons why the control unit must stall the pipelined microprocessor, other are *"Exceptions, Interrupts, or faults"* [18]. We should be careful with terminology as those terms are used indistinctively depending on the author.

- **Precise Interrupt:** those kinds of interruptions are present when all the context is saved, and all the states before interruptions are performed and well define.
- **Imprecise interrupt**: those ones according to the literature, [21] they occur when an instruction does not meet the 3 following conditions otherwise it is precise interrupt:
  a) All the instructions before the current pointed instruction by the Program Counter have been correctly executed, and the process state is well modified.
  b) All the instructions after the current pointed instruction by the Program Counter have not been executed and they have not modified the process state.

  c) When an exception condition is present in a program, the interruption is called, afterwards the interrupted instruction is pointed by the saved Program Counter, this instruction either has not been executed yet or has been executed, that depends on how interruptions are managed in the microprocessor and what kind of exception causes the interruption. In any case the instruction that was interrupted was executed or will start its execution.

  When one of those conditions has not met, it complicates the resuming of the Program after the interruption is called. Because the context changes and the process states might has changed as well, that is why they are called imprecise interruptions.

At the same time according with [18], the interrupts are categorized in two other terms, synchronous or asynchronous.

- **Synchronous interruptions:** These ones are called so because they happen every time the program is executed, furthermore, they use the same Data and the same memory allocation.
- **Asynchronous Interruptions:** contrary to Synchronous ones, they do not use the same memory allocation nor Data, typically those ones are not produced by the microprocessor itself or by the main memory, they are caused by an external device.

As a matter of this work the author centralize in Synchronous *imprecise interrupts*. Synchronous imprecise interrupts refer to the concepts of synchronization and imprecise interrupt meet at the same time. This refers to the interruptions that are present in the program that always use the same data and memory allocation but once that those interruptions are executed, the context and the state of process might change which may lead to problems at the time of resuming the normal execution of the program.

Cache Memories and The Principle of Locality

The solution to the problems exposed in **Chapter 3**, are based on the use of cache memories since they take advantage of the Principle *of Locality or Locality of reference*. The principle of Locality is a tendency that happens when a program is running, and it must access to any memory location, either for Instruction or for Data, those memory references tend to be called repeatedly, some authors such as [18], suggest a rapidly statistical number conclusion "*that a program spends 90% of its execution time in only 10% of the code".* This might not be clear if the program is short, but if the program is longer enough it can be visualized clearly.

For example, if we consider a program where the same temporally variable is called several times or a program where the same data is constantly read. Consequently, the same Data memory locations are accessed several times. Let's discuss the other example.

When a function is called in several parts of the code, the same set of instructions are executed each time, which implies same instructions memory localities are accessed continually, this is also very clear in loops; the same code is executed N number of times.

When a program is executed, the same memory information is used several times. It can be said that the memory addresses used by the microprocessor ingroup in small packages and the program gets trapped within them. Of course, over time the memory information used by the microprocessor changes since it performs diverse tasks, which implies that another group of memory addresses must be used. Those last statements help to conclude that in a short period of time the microprocessor uses a

group of memory allocations but over long period another group of memory allocations are used [22]. With this information in mind, it is possible to use this natural tendency in order to improve the efficiency of the microprocessor, when accessing to memory, and here is where the role of cache memories take importance.

Cache memories are a specific kind of storage devices that are part of the computer organization. Those kinds of memories are the fastest memories available within the memory hierarchy organization shown in the ***Figure 2.6***. Cache memories are located between the main memory and the CPU, caches memories have the same content of a small portion of the main memory. These memories are smaller compared with the main ones, but the access time is shorter. The idea of having this organization is based on the principle of the Locality, since the program gets trapped for a short period of time in a certain group of memory allocations, and it changes group overtime. This tendency of principle of locality applies to all levels of the memory hierarchy, cache memories are thought to take advantage of this to improve performance [18]. That means that it is not necessarily to have all the content of the main memory available all the time, since most of it will not be used. What matters is having the right memory content into the right level of the memory. Those diverse memory levels are filled by a cache memory, following the following rule as nearby to the CPU as quickest.

It might not be trivial understand how increasing the number of memories in the system helps to improve performance. Since now the system have lots of memories and is well known that accessing memory has a considerable latency. However, what is needed is having the memory content in the lowest cache level at time that is required. When this happens, it is called a *cache-hit***.** When a *cache-hit* happens in a low level of the memory hierarchy, the accessing time decrease significantly, since the lowest levels of cache memories are fastest. Meanwhile, if the content is not stored in the right level, the access time of the memory hierarchy organization is worse compare with accessing straight to the main memory, this is called a *cache-miss***.** When a *cache-miss* is present the latency is much worse than the original one, since now we must go through all the memories and to the originally time latency, we most add all the time latencies of the all memories, that is why in the system is crucial to have a high level of *cache-hits* and as lees *cache-miss* as possible.

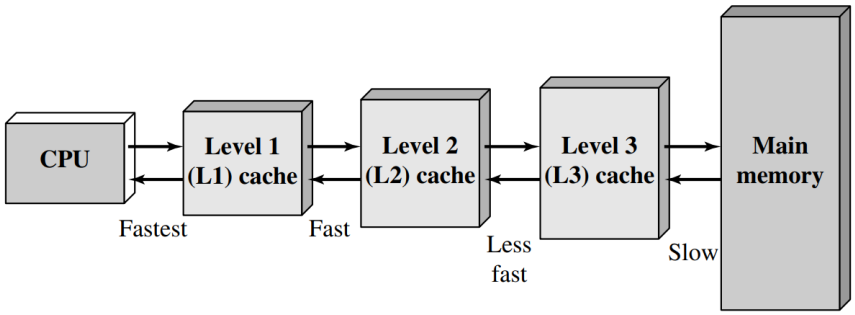

Figure 2.6, Memory hierarchy organization, Image Source: [22]

Data transfer between CPU and the main memory has a considerable latency, which means that is a bottleneck, the use of cache memories helps reducing the latency with the condition of store the right data at the right cache level.

There are some differences of cache memories regarding to the main memory, the data in the main memory is stored by Words while in the case of cache, it is stored by Blocks. This is more efficient due to *"spatial locality"* [18]. That means that not only the content of the memory of one single memory allocation will be used but also the neighbor memory allocations will be more likely accessed too, which implies that bringing a set of memory allocations is better than bringing just one.

The ***Figure 2.7*** shows how the data is transferred from main memory to the cache. In each line (row) of the cache there is stored a complete Block; A Block contains a copy of *K* words of the main memory with a Tag; A Tag helps to match, validate, and identify that the block information with respect to the data into the main memory.

Each cache contains fewer rows in comparison with the main memory. The number of lines is the cache memory size, and the cache are considerably smaller in comparison with the main memory. These memories need to have the data that most likely will be accessed soon in order to keep the performance. Due to the fact that the CPU executes different programs, the data transfer must be dynamic. This implies that the content within the cache memory cannot be fixed.

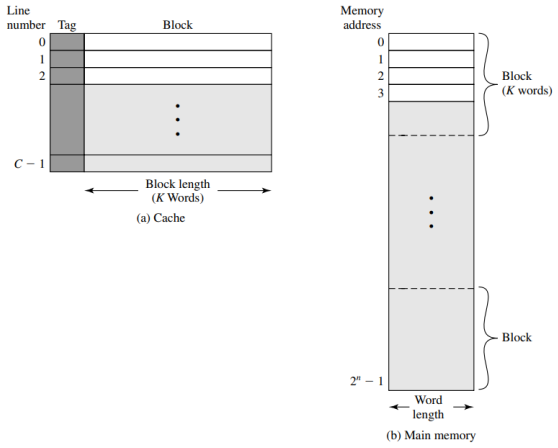

Figure 2.7, Representation of a memory within a cache, Image source: [22]

It is necessary to know when and which data should be stored or replaced. These are split into writing policies and replacement algorithms. Replacing algorithms chose which locations in the cache might be not necessary anymore and can be replaced for new data. As far as this thesis concerned those algorithms are not detailed, there are only exposed writing policies.

When a data is stored in the cache, the content from the cache must be transferred to the main memory. This can be performed in different ways, some of writing policies found in the literature are the following:

- **Write-through:** whenever that a write operation is present, writing is performed in all the memory hierarchy, that assures that the main memory is always valid. This means that every time that something is intended to be written in the cache, the content is stored in the main memory as well, which means that the content is stored at least twice every time; The first one in the cache and the other one in the main memory [22].
- **Write back:** Write in the main memory as less as possible. Whenever a replacement is needed, then update the data in the main memory. In this policy, the content is stored in the cache instead of being written into the main memory, which would create a Data Incoherency because the contents are not the same. However, the concept of a *dirty-bit* or *used-bit* is applied here, it is an extra bit in the cache that is set when incoherency in the data is present. In this way, the

memory allocation that is incoherent can be identified and will be replaced later.

When the microprocessor needs writing the data in cache, it will be performed when a cache-miss occurs. Furthermore, it is needed to write the whole block and it can be done in two different ways:

- **Write allocate***:* Every time that a write-miss occurs, the block is allocated followed by a write hit operation. *Write misses act like read misses*. In other words when a write operation is performed the whole block is brought to the cache [18].
- **No-write allocate:** This one does not perform the write operation when the write miss occurs and only the low memory-level block is modified. In other words, the content is stored in the main memory first and when there is a read operation, there will be produced a cache-miss, and finally, the data is brought in the cache. [18].

In general, write-allocate is better for write-back caches meanwhile no-write-allocate makes more sense for write-through caches.

Chapter 3

Multicore Issues

Once that it has been reviewed concepts related to this work, the author presents in this chapter the main problematics that occurs when STLs procedures run in a multicore context. The chapter centralized in the issues related to booting time diagnostic and the constrains that implicates working with it, such as, memory available, avoid no-deterministic procedures and the possible conflicts with external peripherals. When it is not worked properly with the constraints just mentioned, there is affected negatively the fault grading and the stability of the test signature. The author focuses on that the reader can observe in detail how the stream of instructions is affected by the bus contention in the instant that the instructions are executing the test procedures. Finally, there is discussed the importance in the use of the performance counters and why they should be included in the test procedures when available.

Boot-Time Test Issues In Multicore Context

As mentioned before, SoCs are widely spread in automotive applications and they are easily found all over the vehicles. STLs is the most used solution for testing microprocessors online. Currently, those SoCs are based in multi-core architectures. What follows, the testing procedures must be designed and developed for working with these parallel architectures instead of single-core SoCs. This transition is not trivial. When these STLs are applied in multi-core SoCs, the results are unexpected, specifically in booting-time tests as it is exposed in the case of study presented in ***Chapter 5***. This section exclusively exposes the problems related to creating testing procedures followed by the issues of programming STLs for high-performance multi-core SoCs.

The advantage of Run-time tests is that they can run concurrently with the application generally when the processor is not in use. Nevertheless,

boot-time test carries more complex issues because there are more constraints that the programmer should follow. The awareness of the limited resources available is crucial, and these constrains are exposed below:

- One of the main constraints is the memory usage. The memory storage and RAM available in the SoC is limited, and it cannot be forgotten that in all modern SoCs there is an Operating System included. Hence, STL source code cannot be large, and the use of RAM most be limited since there are not much space reserved for testing [13].
- Boot-time software test must be deterministic and any kind of indeterministic source must be avoided. The *S*equences of instructions can be modified by the microprocessor anytime that the microprocess might require it [13]. At the same time, STLs must execute uninterruptedly. This leaves the programmer with the obligation to reduce as many conflicts with the microprocessor as possible. At the same time, the programmer must be aware of a need of deactivate any unit that might cause the interruption of the boot-time procedure.
- Weather there are some other peripherals or Interrupt controllers avoid conflicts with resources that the STLs might use as well [13].

Timing Issues

An important problematic that appears when applying the STLs to multiple-core SoCs is Timing. However, Timing is uncertain and changes depend on the STLs. Time uncertainty came out as a result of the shared memory bus that is common to all the cores in the microprocessor. When one of the cores tries to access the memory and there are other cores that are performing a memory request it results in a bus conflict and the access memory time increases. This is called *Bus Contention* [14]. When this problematic occurs, the microprocessor stalls at the fetching stage. This happens because even though each core has their own instruction memory, The instructions come from the main memory and then they are assigned to at core. Nevertheless, the fact that the instructions are coming from the main memory and it is shared by all the cores. This is what causes the memory

bus contention. When being aware that the memory access is already the bottleneck in the architectures and adding a bus conflict to this. This effect becomes more significant in multicores SoCs. This bottleneck causes that STLs have a low efficiency.

When the microprocessor performs undesired stalls, the instructions in the pipeline change. This leads to the fact that STLs execute with volatile interruptions making it extremely difficult to determine when the instruction is going to be executed. This directly affects two main aspects in self-test procedures.

- **Fault grading**: Due to the fact that the time in which the instruction is performed changes depending on whether the processor stalls or not. The fault that is intended to be stimulated might be not excited which means that this specific fault remains non-detected.
- **Incorrect signature:** when the timing changes, the signature becomes *unstable*, and the matching of the signatures is not trustable.

Those affections are clearly unsought and as a consequence of this affections, it will be an untrustworthy fault coverage. Since we are talking about STLs that are thought for being used in the automotive industry and eventually in everyday vehicles, the STLs that are used in the SoCs must fulfil modern safety standard requirements such as ISO 26262 for the automotive industry.

For a better understanding, it will be discussed the MIPS microprocessor explained in **Chapter 2**, which consist of a typical 5 stages pipelined microprocessor and the unit under test is the forwarding unit. In the example that is analyzed in the following paragraphs the forwarding by-passing occurs when there is a dependency of the value of two consecutive arithmetic or logic instructions. Indeed, the MIPS is a single-issue microprocessor, however the same logic can be perfectly transpose to multiple-issue microprocessors, with the consideration that instead of using the term issued instructions at the same time, there must be used the term package of instructions issued at the same time with dependencies occurring either within instructions in the package or between packages.

Assuming that a forwarding path between the data at the end of the execution stage to the beginning of the execution stage exists, and it is intended to be tested. The source test code for this specific path is shown in

Figure 3.1. For testing EX to EX forwarding path, there is created a data dependency between the first and the second instruction, aiming that the data produced in the first AND instruction, forwards as an input of the second AND instruction, for then complete its execution one clock cycle after the first instruction was completed.

Clock Cycles				1	2	3	4	5	6
AND	R7	R8	R9	IF	ID	EX	MEM	WB	
AND	R10	R7	R8		IF	ID	EX	MEM	WB

Figure 3.1, EX-EX test source code, single core execution

As it is expected in a single core microprocessor, there is no memory accessed delay and no stalling has been produced. Consequently, this portion of a test code excites correctly the fault intended to test. For a better detection of the fault, we can use the methodology to discovered Performance Faults described in [23], The methodology describes the use of performance monitors, such as Performance Counters if they are available, they improve the fault coverage. The performance counters can store some important information such as Processor cycles, Instruction completed, Processor cycles with 0,1 or 2 instructions issued, Instruction fetched transitions, branch instruction or other important events that occur during the execution of a program. This information released by the performance counters increases the probability of detecting an abnormal behavior within the Hazard Detection Unit.

On the other hand, ***Figure 3.2*** shows exactly the same STL source code, but it is executed in a multiple-issue microprocessor, where the test is executed in parallel by other cores. The higher system bus contention in the microprocessor has the consequence of stall the microprocessor, releasing no time predictability and the fault coverage drops significantly.

Analyzing what happens in deep, the following results were obtained: The first instruction is fetched in the first clock cycle, and it seems that everything was working correctly at that point. Nevertheless, the problem appeared in the next clock cycle when the second instruction should have been fetched but was not. This is the result of the bus contention, which means that accessing to the memory and bringing the next instruction is not possible in this clock cycle. Outcoming with stalling the microprocessor until the next instruction is prepared in the instruction buffer for being fetched. As we can see in the ***Figure 3.2*** the instruction has been fetched in the fifth clock cycle.

The fault coverage drops as there is unwanted stalls. This yields in that the STL code does not excite the forwarding path correctly that is intended to be tested, even if the test code creates the dependency to stimulate the fault. The register R7 reads the value of the previous instruction when the value is already stored in the Register File. This means that when the second instruction reads the value of R7, it reads it form the Register File instead of the forwarding path that is meant to be tested. Indeed, the consequences are that the signature is not able to catch the fault.

The other important fact is when performance counters are not available, the signature will not display any difference in the result in comparison to the single core scenario. This might errantly be assumed as the test code is working correctly. however, when the test code is fault simulated, the fault coverage drops significantly since the signature is not reliable.

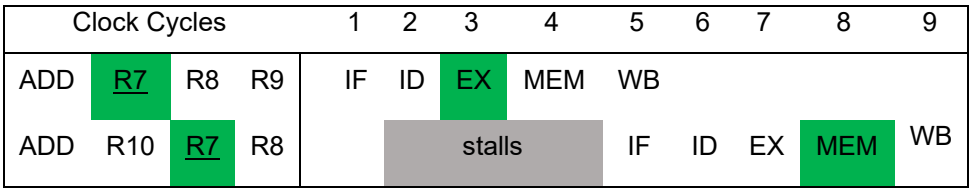

Figure 3.2, EX-EX test source code, multicore execution

The inclusion of Performance Counters into the signature brings a better reliability to the test program as it is demonstrated in the single core

scenario. In the following paragraph, there is discussed the inclusion of this Performance monitors on the signature for the second scenario.

When analyzing the examples of ***Figure 3.1*** and ***Figure 3.2,*** we can see that in the second example the execution time of the STL code is larger compared to the first one. Evidently, that happens because of the fact that there are unsought stalls. If Performance Counters are used, they count the number of stalls performed by the microprocessor; it reports no stalls for the first scenario, and three stalls for the second scenario. Including that value into the test signature, a better reliability is expected.

The reader can notice that when adding the Performance Counter to the test signature causes that the signatures are mismatched. In this specific multicore scenario, the microprocessor has performed three undesired stalls, but it is not certain how many stalls might be performed. It may be zero stalls (when there are no problems) or any other number of stalls (when bus contention or any other issue occurs). There is no way to predict the value of signature when a problematic is present, in other words, the signature becomes unstable. Once that it is known that the signature becomes unstable when undesired stalls are present, it is possible to use this information to catch some permanent faults.

Monitoring the performance of the program using Performance Counters helps to detect permanent faults related to time predictability, and they inform you when there are unexpected stalls. This is a great solution and the awareness of including Performance Counters in multicore SoC is critical, since in multicore SoC there might exist other factors that might alter the continue execution of the program. However, this solution does not assure the continues execution of the STLs in multicore scenario, it only highlights when the execution is interrupted. It is not yet stablished how to assure the correct excitation of the EX to EX forwarding path which the STL source code in the ***Figure 3.2*** is intended for. The solution of this statement is the main focus of this thesis, and it will be discussed in the following Chapter.

Chapter 4

Proposed approach

After having analyzed the main problematics, when traditional STL approaches designed for single issue microprocessors are applied to multiple issue ones. The aim of this thesis is to propose a new methodology for executing in-field self-test routines in a multi-core scenario, avoiding all the problems previously discussed. Bearing in mind all the typical constraint in an automotive multicore SoC. Not forgetting that the system has limited resources in terms of memory and the software execution must be predictable.

Cache Memories as a Solution

As it was discussed in **Chapter 2,** most of the programs and their processes are trapped in a tendence called the principle of locality, which means that when a certain memory location is accessed by the microprocessor, the same memory locations and the neighbor locations, it is highly probable that they will be re-accessed soon. It was referred as well that cache memories take advantage of this concept. Cache memories take data that might be required and stored it in advance. Afterwards, they deliver the information much faster to the microprocessor in comparison with normal memories. The work of the caches is reflected in the performance of the CPU when accessing memory. This is a key clause since one of the main problems that appear when STLs are applied in multicore, is the time delayed when bus contention occurs.

Furthermore, cache memories bring isolation. They are located at the lowest level in the memory hierarchy. The data is read by the microprocessor directly from the caches, avoiding the whole system. Cache memories are present in all modern multicore SoCs. All those statements are the essence of this thesis. There is proposed a new methodology, with use of the cache memories and all their advantages for creating testing libraries

to work in-field, solving the problematics described earlier in multicore scenario for its use in multicore automotive SoCs.

The use of caches as a solution for testing in multicore scenario is not trivial, given that caches are not deterministic. As it was previously discussed, their functionality depends on different aspects such as cache depth size, cache writing and replacing policy, etc. This indeterministic situation must be avoided since it is one of the rules to follow when creating self-test libraries.

The challenges are clear: Create a methodology based on the use of caches to increase the performance of the microprocessor (at the time of accessing memory), the fact that cache brings isolation to the whole system, this implies not entering conflict with the Operating System and avoiding their natural indeterministic behavior.

Caches in SoCs are distributed in levels. The lowest level of cache is reserved to each processor core, as it is exposed in our case of study in the ***Chapter 5***. Consequently, when a program is stored in the lowest cache level, it becomes isolated. Therefore, if we move the STLs to the lowest cache memory, it becomes isolated of the whole system as well. Yet, writing the code directly within the cache, requires a big effort and a perfect knowledge of how it is designed the cache. Moreover, the data in the cache can be modified anytime, because the data that is stored is just temporally data, and STL are typically larger than the memory space available in the lowest level of cache.

The methodology that is proposed to store the self-testing procedure in the lowest cache levels. It takes advantage of the cache memory using them in accordance with its original purpose, along with having the concept of principle of locality in mind. This clause can be explained by recalling briefly how a cache-based systems work. It is known that when the system needs to read information from memory, firstly it will look for it in the cache. If the data is not available in cache, then the microprocessor will determine whether the data should be carried into the cache or not. This depends on whether the data might be needed again (principle of locality). Being aware of the statement might allow the system to believe that the data will be accessed again, which came out with the system keeping the wanted information in the cache and assuring that the data will not be replaced soon.

Therefore, designing a proper code structure assures having the intended information in the cache.

The Loop Solution

One of the structure programs where caches are extremely useful is when loops are present in the code. Due to the fact that loops are parts of the code that is executed repeatedly by the microprocessor, and they have high cache locality. By means of the cache, the code that is within a loop does not have to access to the main memory and load the needed Data every time. It only loads once the Data and then it reads it from the cache, which is faster. Based on this valuable evidence, the solution for the original program comes out. A structure based on a loop results in having the general self-test code into the lowest cache level.

As soon as the test is stored in the lowest cache level, the program will be accessed by the microprocessor with a better performance, avoiding the bus contention. This means that the STL code is executed without stalling making the signature deterministic and stable. A solution based on the use of cache memories assures that the signatures generated by a single core self-test procedure intended for specific modules of the microprocessors, can also be transpose into multi-core execution and come out with a stable execution and with a fault coverage that is deterministic. The proposed methodology does not require modifying the existing solutions significantly and they do not result in introducing penalties from the memory footprint perspective.

The proposed solution starts from a generic boot-time single-core test program that is shown in the ***Figure 4.1.*** It is modified structure for a multi-core microprocessor explained in detail below:

1) The program must be executed twice inside of a loop-based mode. The *body of the loop* that is represented by the blocks (a) and (c) in the ***Figure 4.2*** is the single-core test procedure but intended for testing the SoC multi-core microprocessor, which, in the original single-core methodology is represented by the blocks (b) and (c) in the ***Figure 4.1****,* Although, it might undergo some slight modifications;

the test might be longer since the original code is intended for test one pipe instead in multiple-issued context, there is a need to modify the code for testing the multiple pipes, which in most cases is just replicate the code to stimulate all the pipes, this operation is trivial.

When the self-test code is executed more than one time, it produces strong temporal locality, since all the instructions are called multiple times. In our proposal it is enough to call the instructions two times and consequently the addresses of each instruction are referenced exactly twice.

During the first loop (*Loading Loop*), as result of the Instruction Cache and Data Caches are activated as is shown in the ***Figure 4.2*** in the block (b), all the instructions are moved to the instruction cache and if there is a content from the data memory its' addresses are stored into the Data cache, assuming that the cache-memory is a write allocate (data at the missed-write location is loaded to cache).

If the cache has another write policy, which is our case of study, after each store operation, there is performed a *dummy load operation* in the same address that forces the cache to have a read cache-miss, and when this is triggered, the data is stored into the data-cache. Hence, the second time that the code is executed, the microprocessor will find all the instructions in the instruction-cache and all the data in the data-cache. That means that the store operations will not result in any write-miss.

Another import fact that we must consider is that during the Loading Loop the signatures that are generated are not reliable. Due to the fact that during the Loading Loop the microprocessor is full of not deterministic actions, such as stalls provoked by all the misses in the cache memories. The order in which the instructions are assigned to the pipes depends on the logic of the circuit; one pipe might work more extensively than the others, which produce a totally indeterministic signature.

On the other hand, when the second execution is performed, the *Execution Loop*, all the instructions and data required in the test are loaded in the caches. As a consequence, the microprocessor has access to all the content faster, avoiding stalls that might be provoked due to high system bus contention. Additionally, each test is assigned to its respective pipe, resulting in that the entire test is deterministic, what allows the signatures to be computed without any risk or any influence by the rest of the system.

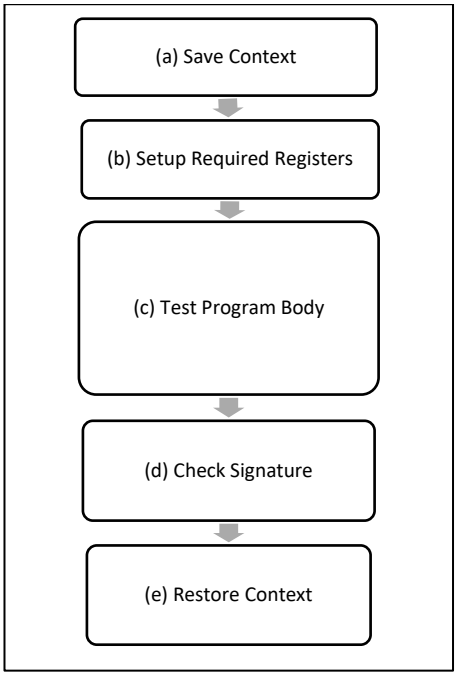

Figure 4.1

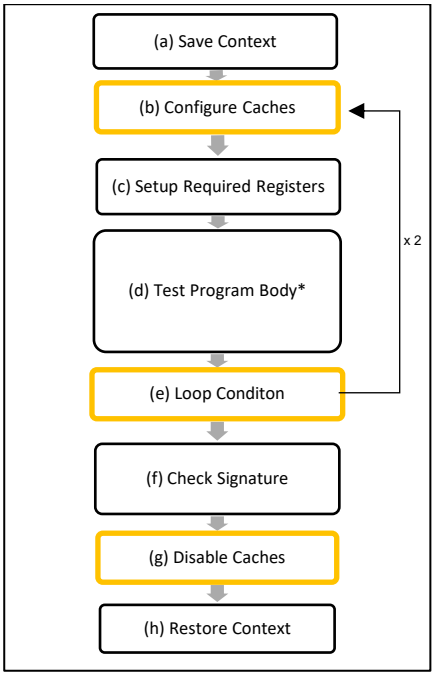

Figure 4.2

2) The *entire* test source code procedure and the data must be stored in the caches at the Loading Loop, as it is explained in the previous section. It brings high spatial locality and stable signatures since unsought stalls and cache-misses are avoided. Yet, there are some other considerations that the proposed methodology must fulfil to prevent the possible alteration of the signature. These considerations are:

2.1 Avoid Conditional Branches in the Execution Loop. When this kind of instructions are present in the code, the order of execution of the test

might change due to the nature of the instruction. it halts the microprocessor.

Clearly, this statement does not apply when the order of execution is intentionally changed by the programmer in order to test specific faults related to alter the execution flow. This do not conditionate the methodology when a loop-based program is intended to be tested, it is needed that all the possible branches are taken and included in the code.

2.2 The test code of the multicore version (***Figure 4.2***) must not be larger than the size of the cache memory. It is important to be aware of the size of the cache. As soon as the program is larger than the size-cache the content of the cache starts to be replaced, depending on the replacing policy, this makes the program being unstable, not deterministic and compromises the fault coverage.

When the program overflows the size of the memory cache, the solution is to divide the large self-test procedure in diverse smaller programs so that they always fit into the cache not to compromise the effectiveness of the test. For example, if the resulting test program is 20KB and the available cache is 8KB the code should be divided in three different test-programs: two of 7KB and one of 6KB.

3) The initialization of the instruction cache and data cache must be performed before the execution of the test procedure. The initialization is done by the invalidation of the content as it is shown in the ***Figure 4.2*** in the block (b) before the part of the code where the test procedure is executed, as it is illustrated in the ***Figure 4.2*** in the blocks (c) and (d).

The proposed methodology which uses the caches, covers all the statements required in order to achieve a deterministic behavior with low resources usage, this is possible thanks to two important facts, which are:

- Caches isolate the processor from the rest of the system, what implies that the execution of the self-test program is performed without being affected by other processor's activity, and there are no external stalls that might affect the efficiency of the test-procedure.
- The program that is stored into the cache does not affect the self-test routine memory footprint.

One important advantage of the proposed approach is that it is a hundred percent software, and it does not require any additional testing device on-chip. Store self-test procedures in cache in a multicore scenario have been studied before in such systems where the memory is common for all the cores as it is presented in [15], however this methodology requires off-chip resources for load the test-codes in the cache memories.

The methodology proposed in this thesis successfully reduced the application time of the test in the SoC, and it can be applied when the vehicles are in use, while in the other methods they are exclusively designed for end-of-manufacturing, they assume that STLs are loaded with an external test device, which clearly is not available when the vehicles are in use.

Another methodology found in the literature is [16]. Where it is possible to take advantage of the memory hierarchy as well, however, the aim is different than the one proposed in this thesis. The one proposed in [16] is called *cache-aware selective testing,* which their aim is to run test procedures faster and does not deal with testing when the vehicle is in use, while the one exposed in this paper does. Moreover, this thesis specifically focuses on boot-time procedures in-field, and it does lead to the uncertainties that occur in multicore SoC architectures by means of the use of cache memories.

The last work that is related to boot-time execution test is covered in [13]. this paper targets the conflicts that arrive when sharing resources and they come out with some scheduling alternatives. Nevertheless, the problem of indeterministic behavior that occurs when self-test procedures are applied in multicore scenario is not covered.

The methodology proposed in this chapter covers all the requirements of a modern automotive multicore SoC using cache memories. It avoids the main problematics that came out when STLs are switched from a single core context to multicore context such as bus contention. Bus contention appears randomly in multicore microprocessors at the time that the microprocessor requires data from memory provoking unwanted stalls. However, if the test procedure is being executed from the cache the microprocessor does not perform additional stalls. Furthermore, when cache memories are activated, they bring isolation from the rest of the system.

Due to the proposed methodology, other issues come out once cache memories are activated in boot-testing. However, the methodology solves them by creating strong spatial locality, in a loop-form program, where the test is executed twice (the loading loop and the execution loop). The first one is a dummy operation, while the second one is the real test. It was shown that other facts should be considered when this methodology is applied. One is always trying to avoid factors that might bring indeterminism to the test. This certainly is an innovation since no other methodology exposed before in the literature covers all those problematics.

Chapter 5

Case Study and Experimental Results

For validate our methodology is necessary apply it to a multi-core microprocessor and confront the results with previous methodologies and see if the results correspond to the expected ones. In this chapter we will discuss and analyze the target multicore device used for carrying out the experiment. Afterwards, the results will be presented with the problematics previously discussed in the **Chapter 3**. Other important aspect that is covered in this chapter is how effective is the methodology in comparison with other possible alternatives.

Case Study

For the experiment, the target device was an industrial triple-core System-On-Chip, manufactured for automotive application, in specific for safety-critical application ranked as ASIL D. which consist in in three dual-issue processor cores. Which from now on they will be referenced as core A, B, and C. Core A and B are twins, each of them is 32 bits processors cores, in other hand core C consist in an extended instruction set version which can perform 64-bits operands.

All the cores have two Tightly Coupled Cache Memories modules, one for private data which a capacity of 4kB and the other for the instructions with capacity of 8kB. The cache modules can be configurable a prior and can supports the two different policies: write allocate and no-write allocate.

Despite of it exists different test-fault models, in this experiment it is only applied the stuck-at fault model, however, that does not exclude the applicability of the proposed methodology to another test-fault model that exist out there.

The analysis starts with the number of stuck-at faults for the core processors, and it suffer a variation from 643,209 in the core C to 473 in the

core B, core A despite being equal from the concept viewpoint to core B, it has a different stuck-at fault list, that happens due to the physical design process. The physical design process changes for each core processor which results in different stuck-at fault lists, hence from the testing point of view they must be treated different.

Recalling one of the main problematics when talking about transposing single-core self-tests to a multiple-core context exposed in **Chapter 3,** the same problematics are shown in the units of the microprocessors who deals with the stalls and the order of the execution. The Interrupt Control Unit and Hazard Detection Unit self-test procedures are the ones that suffers significantly due to these phenomena, it is important to mention that the forwarding mechanism is possible the most affected one, because it is straightly related with the order of execution, the forwarding mechanism is included into the hazard detection unit. Hence in this thesis we centralize the effort exclusively in the faults related to these units, in the microprocessor that is under test in this experiment, the Hazard Detection Unit is composed of The Hazard Detection Control and the Forwarding Logic.

The Hazard Detection Control Unit focus on the detection of the possible problematics due to the dependencies of the issue packets. When it is worked in a multicore situation, it is not talked about isolated instructions, it is referenced as issue packet as it is explained in **Chapter 3.** Depending on possible conflicts of data between different packages of instructions or within the same package of instructions. The Hazard Detection Control Unit solves the conflict by either inserting bubbles in the pipes or driving the forwarding paths.

The Forwarding Logic consists in special multiplexers, which there is a logic within them that helps to manage the forwarding conditions. Additionally, these multiplexers send results produced by the different execution units in distinct stages of the microprocessor to previous stages either in the same pipeline or to the other pipeline.

Testing this kind of units is not something new and there are some methodologies for *"develop a systematic SBST methodology that enhances existing SBST programs to comprehensively test the pipeline logic"* found in [24], and the one that is of our interest for *"produce test programs suitable to*

detect stuck-at faults in computational modules belonging to dual issue processors" [25] as is aimed for multiple issues microprocessors.

We take as an important source [25] due to is similarities with our case of study. In that article is well tested the forwarding mechanism, and it considers the two possible paths existing, *intra-pipeline and inter-pipeline* paths.

intra-pipeline refers to the instruction dependencies of two different consecutive pipelined issue packages. For example, supposing there is a dependency between the instruction 1 and instruction 3 in the execution stage and they are in different pipelines, as is shown in ***Figure 5.1***.

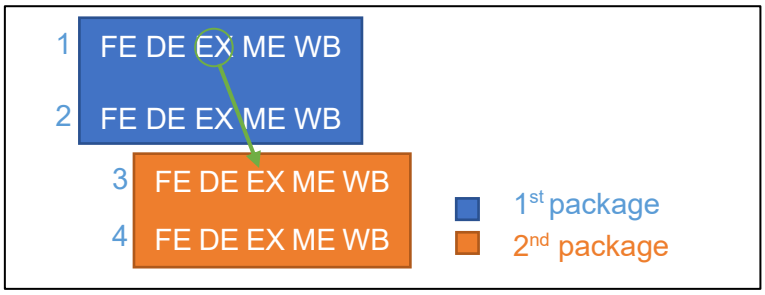

Figure 5.1, intra-pipeline dependency EX-EX execution in a dual issued microprocessor

inter-pipeline refers to the instruction dependencies within the same pipeline in the same issue package. For example, there is a dependency of two consecutive instructions, 1 and 2, and the result of the first one in the execution stage is required in the second one but in the memory stage and both are in the same issue package, as is shown in the ***Figure 5.1***.

Additionally, the method proposed in [25] made use of the performance counters, as it was discussed before in **Chapter 3**, they help for detecting performance faults and realize when the microprocessor produce unwanted stalls, bringing stability at the time of executing the test. The performance monitors in [25] are used for counting the number of times that the microprocessor is interrupted wrongly by the hazard control unit while it is executed of the self-test procedure.

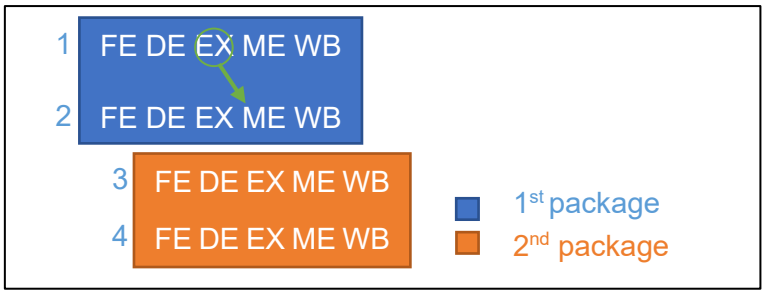

Figure 5.2, inter-pipeline dependency EX-ME execution in a dual issued microprocessor

For the analysis of the Interrupt Control, the interrupts presented here are synchronous *imprecise interrupts* defined in **Chapter 2,** this interruptus are present when the memory or the CPU requires to stop the execution due to one instruction (i.e., synchronously), or exception. The CPU then attends some other work with more priority. *imprecise interrupts* have the characteristic of does not save properly the context before interrupting the CPU, that means that either the program counter, registers or memory states have not been backup before the stalling. This happens do to the fact that we are running in a multiple issue pipelined microprocessor, and it depends on the number of instructions that are running at the same time in the pipelines, this kind of interruptions are hard to recognize it, and when the microprocessor realize about them some other instructions are executed outside that the instruction that is interrupted. Unlike the *precise interrupts* where everything is well defined.

To know exactly how many instructions have been performed by the microprocessor after an *imprecise interrupt* occurred is extremely hard, for the reason that once an instruction of one of the pipelines requires to be interrupted, it will interrupt the process of that instruction. Still all the other instructions that are running at the same time in the other pipelines might have completed their execution when they should not do it. When the imprecise interrupt is recognized after the other instructions that should have not completed their execution, but they have done, the instructions in the pipelines must be reassigned either to the same pipeline or another pipelines, for then been executed again with the correct context. those making the system unpredictive because the number of instructions running into the pipeline change ever time that one of these interrupts are present.

The test codes that aim those kinds of *imprecise interruptions* are unstable because they depend on the other instructions activity running in parallel in the other pipelines. This produce an unstable signature because there is not consistency in the execution time. For managing this kind of faults, we follow the method proposed in [26]. What they do is identify critical points in the test where there exist some resource conflicts or it exists large instruction dependencies and the interruptions occurs in those points (the methodology achieves higher fault coverage than other methodologies). The resultant faults obtained in the units using this technique are reported in ***Figure 5.3*** and ***Figure 5.4*** in the column N of Faults.

As it is explained in **Chapter 4** the last step was enabled both cache memories available, since the microprocessor that is used for this experiment can manage both write policies. It was chosen write-allocate for both test programs. In this way all the write cache misses during the execution loop are avoid it. That implies that it is no necessary add more load operations. It is important to mention that as the size of both test programs are smaller in size compared with the available caches size, both tests fit into the cache memories. This is convenient for the proposed methodology, because it is not need it to split the test code.

Uncertainties in multi-core SoCs

The experiments started applying the multi-core test libraries developed in the multicore microprocessor and analyzing their behavior in a multi-core context. Due to Core A and B are twins, it was developed only two test libraries, one for core A and B and one for core C. It was decided to split the experiment in two analyses, the first one targeting the forwarding mechanism and separately a second analysis where imprecise interrupts and hazard detection unit are treated.

In this initial step, the STLs were only applied to the forwarding mechanism, following the methodology proposed in [13]. It was applied the same software structure and the STLs were executed in parallel on the physical microcontroller. Afterwards with the help of an external debugger, it was tracked the executed instructions of the STL programs, and it was

counted the number of clock cycles stalled because of the access memory concerns that might appeared in each core.

In ***Figure 5.3*** is observed all the results collected by the debugger and it is clearly that when is moved from a single core (only one switched on) to a multiple core (more than one core is active), it can be noticed that as the number of cores are activated as the number of stalls increase. This might be expected, if more cores are working, more times the microprocessor must stalls. However, this is not a linear behavior. As it can be observed in ***Figure 5.3***, the number of stalls that happened, when one core is activated in comparison when 2 cores are active, are significantly higher. That means that it is suffering of bus contention. Recalling what was discussed in **Chapter 3,** this is happening because all the instructions are coming from the main memory and this one is common for all the cores.

Number of Active Cores	IF STALLS (Clock Cycles)	MEM stalls (Clock Cycles)
1	200,679	117,965
2	717,538	305,801
3	1,878,336	663,386

Figure 5.3, quantity of stalls as the number of active cores is present

The instruction fetch unit (IF) is the one that suffers more stalls due to this phenomenon. This is explained because as soon as other cores are started to switch on, the bus contention rises. Therefore, the Instruction Fetch Unit of the core that requires a certain instruction is forced to stall and waits until the other instructions that have been requested have been issued for then finally been fetched.

Finally, it is important to highlight that the results of the second and third row shown in ***Figure 5.3*** represents mean values collected by the debugger after several executions, the real number of stalls depends on the initial state of the SoC in each execution, which makes it unpredictable.

Uncertain Fault Coverage

After the execution of the previous experiments, it is confirmed that when self-test procedures are executed in a multi-core microprocessor, they become highly unpredictable, because they are dependent of the context of execution, the whole system affects the effectiveness of the STLs since they should be executed without being interrupted. Nonetheless, it is not clear yet, how much those interruptions affect the self-test libraries when they are on execution. Due to this fact the experiments exposed below aim to see the effects of the microprocessor stall into the STLs. In fact, the following experiments focus on the fault coverage within the Hazard Detection Unit, which is the one that suffers more the presents of stalls when they are on execution. The experiments were realized with the help of a commercial fault simulator, and with the netlist of the post-layout gate-level of the SoC.

The experiment started with the data of ***Figure 5.3*** exposed in the previous section, it was shown clearly that the interruption of the pipelined due to memory bus contention produce that the Performance Counters (PCs) become unstable when they are used in a multicore context. And as it was explained before in **Chapter 3** they are a good mechanism for the reliability of the self-test-signature.

One solution for avoiding the unstably introduced by the PCs is pull them out of the self-test procedure being aware that the fault-coverage (FC) will drop significantly, however as we can observe in the results showed in the ***Figure 5.4***, the fault coverage does not reach a proper value for guaranteeing a deterministic fault coverage of the forwarding logic.

Core	Number of faults	Min - Max FC [%] no caches no PCs	FC [%] with caches no PCs
A	53,298	64.14 - 75.19	79.61
B	57,506	63.61 - 79.59	82.08
C	113,212	56.24 - 66.48	68.79

Figure 5.4, FC previous solutions vs cache-based solution.

The results of ***Figure 5.4*** were obtained following the procedure exposed in [25], which propose self-test techniques for dual issue microprocessor, however it had to be adapted to the needs exposed before. The logic simulations were fault simulated first removing the Performance Counters from the Self-test procedures, and then they were executed varying the following conditions.

a) Two and three active cores
b) Use of the high, middle, and low Flash memory locations
c) Word, double-word, and double double-word code alignment

Despite of changing all those parameters the signature remains steady, because the resulting signature it is not affected by those changes. Afterwards it was summarized all the results, for a better understanding it is exposed only the Minimum (Min) and Maximum (Max) values from the fault simulation for a clear comparison with the solution proposed in this thesis.

The results of this experiment show that the resultants fault coverage is extremely unpredictable since the results are not consistent. If we observe the core B in the third column from ***Figure 5.4*** the variation reaches almost 16% even if the signature does not change along each of the fault simulations.

In this fault coverage experiment the result of the fault simulations depends mainly on the order of execution, since the self-test is aimed for excite the forwarding paths when the issue packets enter in the pipeline not continually, the aimed paths are not excited and as consequences the fault coverage drops, in the other hand when the packets enter as they were aimed for, the forwarding paths are excited, therefore the fault coverage

increase. This explains the fluctuation into the fault coverage, it is needed to ensure that each packet of instructions is executed in consecutive clock cycles and without stalling for having a steady fault coverage.

As soon as we applied the proposed approach of the **Chapter 4** as it is shown in the last column of the ***Figure 5.4*** with the use of the caches the fault coverage is always higher with respect the Max value of the previous solution, in the best case the improvement is around 4% better, with the significant difference that there are no variations, when the conditions a), b) and c) are present.

Continuing the analysis of the results of the Core C, last row illustrated in the ***Figure 5.4,*** the fault coverage it is lower with respect Core A and Core B, we should recall that the core C is the extended version, which is able to manage 64-bits operation, this implies that the multiplexers in charge of the inter-pipeline and intra-pipeline forwarding mechanism are 64-bits width as well. Although the Microprocessor can manage 64-bit the General-Purpose Registers (GPRs) are still 32-bits wide, which means that when the self-test procedure is done the signature is stored in a 32-bits, this carries some fault simulations effects to be masked. This last statement is not targeted in this thesis, since the main argument is to improve the methodology for the application of the STLs, not for the improvement of the algorithms of the forwarding mechanism.

Unstable Signature

After having been performed the experiments related to the Forwarding Unit, the other units that remained unexamined is the Interrupt Control Unit (ICU) and the Hazard Detection Control Unit (HDCU), which are the ones related with the *imprecise interrupts*. The HDCU was tested with the same algorithm proposed by [25] for the Hazard Unit in a Dual issued Microprocessor where Performance Counters are used. For the ICU the algorithm that it is used is the one previously mentioned by [26]. Both algorithms are applied to all the cores (**Core A**, **B** and **C**). and the results are presented in the ***Figure 5.5.***

The ***Figure 5.5*** shows the results of the fault simulations of ICU and the HDCU in two different contexts. The fourth column shows the fault coverage achieved when the STLs are applied in a single-core scenario, only one core is active at a time, and the proposed cache approach is not applied. In this single-core scenario the test programs achieved stability in the test signature. In this case there is no Performance monitors added to the signatures, and the expected test signature stores only the results of the instructions. As it is expected the execution of the STLs in Single core are stable meaning that the fault coverage under these conditions is stable.

Core	Module	Number of faults	FC Single-Core no caches [%]	FC Multicore with caches [%]
A	ICU	14,230	46.57	51.36
	HDCU	16,096	62.53	70.37
B	ICU	13,149	46.39	50.97
	HDCU	15,783	63.84	70.12
C	ICU	13,888	54.94	60.91
	HDCU	19,931	65.66	68.09

Figure 5.5, ICU and HDCU Fault simulation results, Single-Core vs Multicore.

The last column of the ***Figure 5.5*** shows Fault simulation results that are achieved when the STLs are executed under the proposed cache-based methodology. The methodology archives stable execution and stability in the test signature when all the cores are active and all the cores are running concurrently their test procedure. Other important fact is that when the STLs are applied without activate the cache memories, the test procedure inevitably failed in any configuration. Therefore, the proposed approach is crucial in order to reach stability in the test signatures for then compute the fault coverage.

The results in the ***Figure 5.5*** showed that the cache-based approach achieves higher fault coverage in both units under test (ICU and HDCU) in all the cores (all cores are active at a time) in comparison with the single core scenario (one core is active at a time). Rather than be more effective when

STLs run in a single core scenario, the system suffers important latency due to the memory subsystem. The system takes eight clock cycles when fetching an issue packet from the Flash memory. Due to this latency the effectiveness of the STLs drops. When the STLs do not execute continuously, they do not excite all the possible paths that the test is designed for neither they do not trigger all the imprecise interruptions. All those clauses are reflected in the lower fault coverage achieved in a single-core scenario.

Meanwhile, in the second scenario there is no latency when fetching an issue packet from the Instruction Cache memory, it takes one clock cycle rather than eight. Another point that worth to mention is that the HDCU with the proposed approach achieves similar fault coverage in the three cores, around 70 percent. Nonetheless, in the ICU of the core C the fault coverage achieved around 10 percent better effectivity compared with core A and B. This difference basically depends on the different implementations of the ICU depending on the core. The ICU in core C exposes some software-accessible registers that are used to identify the type of possible interrupts that might appear. While this operation in the cores A and B are mapped using the same bits. This statements clearly affect the obtained fault coverage.

Comparisons with Other Solutions

The last analysis presented in this work is compare the proposed solution with the other solution that is used nowadays for testing multicore microprocessors, this alternative solution uses the microprocessor's *Tightly-Coupled Memories* (TCMs). Store STLs in the TCMs is proposed by [27], They also can be found in the literature as *scratchpad memories* proposed by [28]. TCM based solution are used for executing programs in real-time. What they proposed is copy the program when the system is booting and then execute it from the instruction TCM. A TCM is a reserved space of the SRAM memory but without the concept of Data coherency that exist in a Caches memory because it is in fact a real copy of the program. This implies that it must be reserved a permanently space for the TCM in memory for testing purposes.

TCM-based approach has similar advantages alike the cache-based one proposed in this work. However, it must be noticed the amount of SRAM memory space that is needed to reserve, it is the actual size of the whole test library. Additionally, the fact that the reserved space must be fixed during all the time that the microprocessor is in use, it is clearly a weak point. The impact of this clauses affects the test library portability and flexibility. Unlike the cache based one no extra memory space must be reserved. The test programs are designed for: fit into the cache, activate the caches memories, load and run the tests, and deactivate the caches memories. Furthermore, as soon as the test it is executed, the caches become available for the microprocessor. Which means that the instructions of the test can be replaced with other instructions or with another test program.

SOLUTION	Overall Memory Overhead [bytes]	Execution Time [Clock cycles]	Increment Execution Time
TCM-BASED	2,874	16,463	0 %
CACHE-BASED	0	18,043	9.59 %

Figure 5.6, TCM vs Cache methods, Execution Time and Memory Storage.

In the ***Figure 5.6*** it is shown the two approaches, TCM-based and Cache-based. It is reported the execution time and the Overall Memory Overhead of both strategies. Both solutions achieve similar results when STLs target imprecise interruptions and Hazard Detection Units, Since the results are similar those are not reported here. The complexity of the two methodologies is reduce of additional some instructions that are added to the test, this means they do not have substantial differences in terms of flash memory storage. At the same time the fault coverage remains the same in both strategies.

As it can be noticed by the reader the difference that the ***Figure 5.6*** shows is that the TCM-based, requires copying the test program to the

SRAM, and then being executed. However, in comparison with a cache-based strategy there is not a significant difference in terms of clock cycles, it is around 1,500 clock cycles more. The cache-based one it is larger since as it was exposed in the methodology exposed in the *Error! Reference source n ot found.* the test should be executed twice in order to load all the test in cache. If it is considered that this difference might be negligible when the STL is executed at-speed (8.25μs when the considered SoC operates at its maximum frequency of 180 MHz). The other significant difference is the amount of extra memory space that is required for the TCM methodology. The cache-based approach does not increase the overall memory footprint of the self-test procedure.

Conclusion

The use of a cache-based approach for testing Multicore Architectures achieves deterministic execution and solves the problematics that come out when self-test procedures work in a multicore scenario. The methodology provides a deterministic fault coverage, a stable signature, and a steady time predictability. The solution does not need neither an additional in-built hardware nor external device. This is an advantage because the solution is convenient for online testing. The carried-out experiments support the proposed methodology because of its applicability to any self-test procedure. The memory space required for this methodology does not increase and does not need any additional SRAM space reserved for testing, as in case of Tightly-Coupled-Memories (TCM) based methodology. The only concern is that the execution time is longer in comparison to TCM-based ones. However, this difference is not significant since the cache-based methodology only requires slightly more clock cycles.

Appendixes

Appendix 1

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM. ALUOutput} else {PC+4});		
ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);		
	ALU instruction	**Load or store instruction**	**Branch instruction**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

Bibliography

[1] F. Reimann, M. Glaß , j. Teich, A. Cook, L. Rodríguez Gómez, D. Ull, H.-J. Wunderlich, U. Abelein and P. Engelke, “Advanced diagnosis: Sbst and bist integration in automotive e/e architectures,” in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014.

[2] Library, ARM Software Test, [Online]. Available: https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries. [Accessed 2019].

[3] Infineon Software Test Library, [Online]. Available: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/. [Accessed 2019].

[4] Cypress Software Test Library, [Online]. Available: https://www.cypress.com/file/249196/download. [Accessed 2019].

[5] Renesas Software Test Library, [Online]. Available: https://www.renesas.com/eu/en/products/synergy/software/add-ons.html#read. [Accessed 2019].

[6] Microchip Software Test Library, [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf. [Accessed 2019].

[7] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez and A. Sansonetti, “Development flow for on-line core self-test of automotive microcontrollers,” *IEEE Transactions on Computers,* vol. 65, no. 3, p. 744–754, March 2016.

[8] Thatte and Abraham, “Test generation for microprocessors,” *IEEE Transactions on Computers,* Vols. C-29, no. 6, p. 429–441, June 1980.

[9] A. Paschalis, D. Gizopoulos , N. Kranitis, M. Psarakis and Y. Zorian , “Deterministic software-based self-testing of embedded processor

cores," in *IEEE Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, Munich, March 2001.

[10] M. Psarakis, D. Gizopoulos, E. Sanchez and M. S. Reorda, "Microprocessor Software-Based Self-Testing," *IEEE Design & Test of Computers,* vol. 27, no. 3, pp. 4 - 19, May 2010.

[11] N. Kranitis, M. Andreas, T. George, P. Antonis and D. Gizopoulos, "Hybrid-SBST Methodology for Efficient Testing of Processor Cores," *IEEE Design & Test of Computers,* vol. 25, no. 1, pp. 64-75, 08 January 2008.

[12] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 20, no. 3, pp. 369-380, March 2001.

[13] A. Floridia, D. Piumatti, A. Ruospo, E. Sanchez, S. D. Luca and R. Martorana, "A Decentralized Scheduler for On-line Self-test Routines in Multi-core Automotive System-on-Chips," in *2019 IEEE International Test Conference (ITC)*, Washington, 2019.

[14] M. Lv, W. Yi, N. Guan and G. Yu, "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software," in *2010 31st IEEE Real-Time Systems Symposium*, San Diego, CA, USA, 2010.

[15] A. Apostolakis, D. Gizopoulos, M. Psarakis and A. Paschalis, "Software-Based Self-Testing of Symmetric Shared-Memory Multiprocessors," *IEEE Transactions on Computers,* vol. 58, no. 12, pp. 1682-1694, December 2009.

[16] M. A. Skitsas, C. A. Nicopoulos and M. K. Michael, "DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective-Testing in Multi-/Many-Core Microprocessors," *IEEE Transactions on Computers,* vol. 65, no. 5, pp. 1453 - 1466, 1 May 2016.

[17] M. Bushnell and V. Agrawal, Essentials of Electronic Testing for Digital, Memory and Mixed-signal VLSI Circuits, Kluwer Academic Publisher, 2000.

[18] J. L. Hennessy and D. A. Patterson, Computer Architecture A Quantitative Approach, 5th ed., Waltham, MA, USA: Morgan Kaufmann, 2012.

[19] STMicroelectronics, "st.com," STMicroelectronics. [Online]. [Accessed 2021].

[20] STMicroelectronics, "st.com," May 2021. [Online]. Available: https://www.st.com/resource/en/datasheet/spc584c70e3.pdf. [Accessed November 2021].

[21] J. Smith and A. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Transactions on Computers ,* vol. 37, no. 5, pp. 562 - 573, 1988.

[22] W. Stallings, COMPUTER ORGANIZATION AND ARCHITECTURE DESIGNING FOR PERFORMANCE, 8th ed., Upper Saddle River,, New Jersey: Pearson Prentice Hall, 2010.

[23] T.-Y. Hsieh, M. A. Breuer, M. Annavaram, S. K. Gupta and K.-J. Lee, “Tolerance of performance degrading faults for effective yield improvement,” in *2009 International Test Conference*, Austin, TX, USA, 2009.

[24] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan and S. Ravi, "Systematic software-based self-test for pipelined processors," in *2006 43rd ACM/IEEE Design Automation Conference*, San Francisco, CA, USA, 2006.

[25] P. Bernardi, R. Cantoro, S. D. Luca, E. Sanchez, A. Sansonetti and G. Squillero, "Software-Based Self-Test Techniques for Dual-Issue Embedded Processors," *IEEE Transactions on Emerging Topics in Computing,* vol. 8, no. 2, pp. 464 - 477, October 2017.

[26] P. Singh, D. L. Landis and V. Narayanan, "Test Generation for Precise Interrupts on Out-of-Order Microprocessors," in *2009 10th International*

Workshop on Microprocessor Test and Verification, Austin, TX, USA, 2009.

[27] J. Ax, G. Sievers, J. Daberkow, M. Flasskamp, M. Vohrmann, T. Jungeblut, W. Kelly, M. Porrmann and M. Porrmann, "CoreVA-MPSoC: A Many-Core Architecture with Tightly Coupled Shared and Local Data Memories," *IEEE Transactions on Parallel and Distributed Systems,* vol. 29, no. 5, pp. 1030 - 1043, 2018.

[28] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan and P. Marwedel, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, Estes Park, CO, USA, 2002.

[29] M. Psarakis, D. Gizopoulos, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan and S. Ravi, "Systematic Software-Based Self-Test," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 16, no. 11, pp. 1441 - 1453, 2008.