LUND UNIVERSITY

An MPSoCs demonstrator for fault injection and fault handling in an IEEE P1687 environment

Petersen, Kim; Nikolov, Dimitar; Ingelsson, Urban; Carlsson, Gunnar; Larsson, Erik

2012

Link to publication

Citation for published version (APA):
Petersen, K., Nikolov, D., Ingelsson, U., Carlsson, G., & Larsson, E. (2012). *An MPSoCs demonstrator for fault injection and fault handling in an IEEE P1687 environment*. Paper presented at IEEE European Test Symposium (ETS), 2012, Annecy, France.

Total number of authors:
5

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

An MPSoCs demonstrator for fault injection and fault handling in an IEEE P1687 environment

Kim Petersén[1], Dimitar Nikolov[2], Urban Ingelsson[3], Gunnar Carlsson[1] and Erik Larsson[4]
[1]Ericsson AB Stockholm, Sweden
[2]Linköping University Linköping, Sweden
[3]Semcon Linköping, Sweden
[4]Lund University Lund, Sweden

ABSTRACT: As fault handling in multi-processor system-on-chips (MPSoCs) is a major challenge, we have developed an MPSoC demonstrator that enables experimentation on fault injection and fault handling. Our MPSoC demonstrator consists of (1) an MPSoC model with a set of components (devices) each equipped with fault detection features, so called instruments, (2) an Instrument Access Infrastructure (IAI) based on IEEE P1687 that connects the instruments, (3) a Fault Indication and Propagation Infrastructure (FIPI) that propagates fault indications to system-level, (4) a Resource Manager (RM) to schedule jobs based on fault statuses, (5) an Instrument Manager (IM) connecting the IAI and the RM, and (6) a Fault Injection Manager (FIM) that inserts faults. The main goal of the demonstrator is to enable experimentation on different fault handling solutions. The novelty in this particular demonstrator is that it uses the existing test features, i.e. IEEE P1687 infrastructure, to assist fault handling. The demonstrator is implemented and a case study is performed.

I. Introduction

While recent semiconductor technologies enable the design and manufacturing of extremely complex integrated circuits (ICs) that may contain multi-processor system-on-chips (MPSoCs), these MPSoCs are increasingly susceptible to defects. Unfortunately, an increasing number of defects show up during operation. These defects are due to escapes from manufacturing test, aging effects leading to development of faults while the MPSoC is in operation, and environmental impact [1]. The defects lead to faults, which can be classified as soft (transient and intermittent) and hard (permanent). To handle these faults, there is a need of fault handling to automate detection, identification, and recovery from faults occurring in-operation. A key for the fault handling is to correct faults as quickly as possible and keep the system in an operational state. It should be noted that fault detection and correction are best performed as close to the origin of the fault as possible, often in the interior of a system component, while keeping the system in an operational state is best controlled on the system level to have the overall view.

To aid development of fault handling solutions, this paper proposes a demonstrator in which an MPSoC can be simulated and the impact of faults can be studied.

The main parts of the demonstrator are:

- An MPSoC model, which consists of set of components such as CPUs, DSPs, accelerators and memories. Each component is equipped with features called instruments for fault detection.
- An Instrument Access Infrastructure (IAI) to connect the instruments to the master CPU. The IAI follows the proposal for IEEE standard P1687, a standardization of access to on-chip instrumentation.
- A Fault Indication and Propagation Infrastructure (FIPI), which holds fault indications and fault codes, and propagates fault indications to the system-level to shorten the time between the occurrence of a fault and its detection at a system-level.
- A Resource Manager (RM) for collecting fault statuses from instruments, conducting fault handling tasks and scheduling jobs, based on the fault statuses.
- An Instrument Manager (IM), to automate operations given by the RM on the instruments over the IAI. The IM acts as the RMs interface to the instruments.
- A Fault Injection Manager (FIM) which is capable of injecting soft and hard faults into the MPSoC.

The main goal of the demonstrator is to enable experimentation on fault handling solutions at system level, while assuming that fault detection has already been implemented at the component level. The novelty in the demonstrator is that it makes use of existing test features, i.e. test instruments that are connected through dynamically configurable infrastructure that is based on IEEE P1687 standard, to assist fault handling (system monitoring and enable access to faulty components that require some recovery actions).

The paper is organized as follows. A review of the literature is given in Section II. Section III details the main parts of the demonstrator. In Section IV, a case study shows how the demonstrator can be used to perform experiments on fault management, including fault injection and fault handling. Section V concludes the paper.

II. Related Work

This section reviews prior work related to the demonstrator's main parts, as they are listed in Section I, as well as previous works for verifying fault handling solutions.

A. Fault Handling in MPSoCs

Fault handling is that which automates detection, identification and recovery from faults occurring in-operation, with the purpose to maintain correct system operation.

For MPSoCs, several studies aim to improve the reliability by employing fault tolerant techniques and spare parts [2], [3]. A good example of a fault-management solution for MPSoC [3] is an approach in which a job is executed in-synch on two processors and fault detection is achieved by comparing the state of the two processors. A system-level component reacts by re-executing the job as a recovery process. Fault handling solutions typically has a multi-level approach with system-level decision support and at least one level of fault detection and recovery mechanisms. In our demonstrator, the RM provides system-level decision support and the instruments provide fault detection and recovery mechanisms.

B. Instrument Access Infrastructure

To connect the instruments to the system-level decision support in the RM, our demonstrator includes an Instrument Access Infrastructure (IAI) implemented according to IEEE P1687 [4]–[7]. Also known as Internal-JTAG (IJTAG), P1687 aims to standardize access to on-chip instrumentation by connecting a network of instruments from multiple vendors to a chip-level JTAG TAP. Previously available standards have limitations which lead to large overhead when the number of instruments increase beyond a few hundred. Uses of P1687 include access to sensors, Design for Test (DfT), configuration features, debug features, *etc.* Research has investigated the use of P1687 to tune high-speed serial links [8], how to calculate instrument access time [9] and how to optimize instrument access time by design of the instrument access infrastructure [10]. Scalable fault management architecture based on IEEE P1687 is presented in [11].

C. Fault Indication Propagation Infrastructure

Our demonstrator includes a Fault Indication Propagation Infrastructure (FIPI). An analogy to FIPI is the interrupt function in a microprocessor, which triggers for several types of faults with the purpose of notifying the operating system of the fault as soon as possible and trigger corresponding actions. The purpose of FIPI is to indicate occurring faults as soon as possible to the system-level where fault handling takes corresponding actions.

D. Fault Handling with a Resource Manager

In our demonstrator, a resource manager handles both job scheduling and fault handling decisions. This suits many fault handling solutions, such as the approach in [3], which on top of managing faults keeps track of the system 'health' and schedule jobs accordingly. An example is the fault tolerant technique rollback and recovery [12], which involves restarting jobs that are affected by faults. For this purpose the fault handling needs to be able to schedule jobs. In other words, fault handling solutions are incorporated in resource management. Such fault handling solutions can be simulated in our demonstrator, using the resource manager.

In the Razor approach to power and performance scaling [13], an extra latch is put on each flip-flop to detect and recover from delay faults. A delay fault count from each Razor flip-flop is considered while adjusting the supply voltage for minimizing the power consumption while keeping the number of delay faults low. The Razor flip-flops can be seen as fault detection instruments for which a RM keeps the delay fault count and makes the decisions to use other instruments to adjust the supply voltage.

E. Instrument Management

So far, no commercial tool is available for automating operations on instruments over a P1687 IAI, such as is the task of the IM. An ad-hoc solution based on beyond-the-standard use of JTAG exist [14].

F. Fault Injection and Testing Fault Handling

Fault injection is a well studied method for investigating the impact of a fault on system behavior. Fault injection is used to evaluate fault-tolerance methods [15]–[17]. In [15], fault injection-based testing is proposed as a compliment to formal methods for increasing the confidence in fault-tolerant method implementations. They found that fault injection-based testing is essential for verifying fault tolerance, because critical parts of the fault tolerance system are only exercised when faults occur and are handled. Therefore, fault injection is included in our demonstrator.

G. MPSoC Demonstrators of Fault Handling Solutions

One previous study [18] presents a simulation-framework for experimenting with reliability solutions targeting soft faults. The framework includes fault injection capability and a reliability manager. The key idea in [18] is to analyze a task graph at MPSoC design time and add redundancy and self-checking schemes as needed to increase reliability, followed by fault injection-supported verification in a hardware-software co-simulation. In contrast to our demonstrator, the framework in [18] only targets soft faults, does not support experimentation on fault handling schemes and does not include instruments, an IAI, a FIPI nor any other similar concepts.

H. Motivation for our MPSoC Demonstrator

As seen from above, reliability enhancement through fault handling is important for MPSoCs [2], [3]. Indeed, new implementations of fault tolerance techniques must be verified and fault injection is appropriate for studying fault tolerance techniques in action [15], [16]. The same is true for fault handling solutions. Fault injection is best done in a simulation environment, and therefore the study in [18] created a framework for simulating an MPSoC with fault injection to experiment on methods to handle soft faults. To allow experiments on a larger set of fault handling solutions, including such that feature instruments for fault detection and correction (facilitated by an instrument access infrastructure) and that feature fault identification (facilitated by a fault indication propagation infrastructure) and handling of both soft and hard faults, we present our demonstrator. Furthermore, our demonstrator is the first to feature an instrument access interface based on P1687, which is capable of extending to hundreds of instruments.

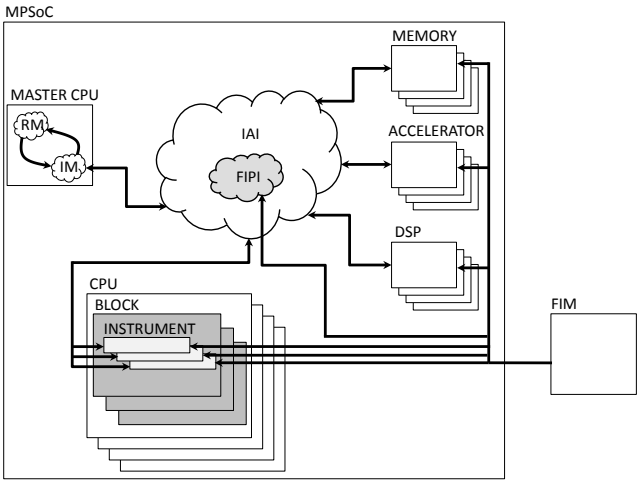

Figure 1: Graphical presentation of the demonstrator

III. Demonstrator

The main parts of the MPSoC demonstrator (see Figure 1), as listed in Section I, are detailed below.

A. MPSoC

The MPSoC consists of a number of components such as CPUs, DSPs, accelerators, memories, *etc.*. For the CPUs, we assume that one of the processors is the master CPU and the rest are work-horse CPUs. The master CPU is used by the Resource Manager and the Instrument Manager. To emulate execution of jobs we use the work-horse CPUs. Each of the work-horse CPUs contains a number of blocks (units), *e.g.* ALUs, multiplication units, floating-point units, *etc.*. Each block contains operating registers, status registers and some test logic circuitry. An operating register is used by the CPU to enable execution of jobs, such as a Program Counter, Stack Pointer, general-purpose register, *etc.*. A status register reports the state of a given CPU. For example, the content of the status register can be a fault indication code. Test logic circuitry is a test mechanism that enables testing a block of a CPU during operation. In this paper, we use the term instrument, whenever we address an operating register, status register or test logic circuitry.

B. Instrument Access Infrastructure

The Instrument Access Infrastructure (IAI) enables access to the instruments in the MPSoC from the master CPU. While an IAI can be implemented using various techniques, some requirements should be considered. In particular, access to instruments might occur more frequently at some points in time and less frequent at other times. It becomes important to have an infrastructure that can be dynamically configured to optimize access time to instruments.

One way of developing a dynamically configurable infrastructure is to follow the IEEE standard proposal P1687 [19], which aims to standardize access to on-chip instruments. A key feature in P1687 is that the infrastructure can be hierarchically organized. In [9] it was suggested that a multi-level hierarchy can be used to prioritize instruments that are more frequently accessed than others.

While several options for the design of the IAI can be envisioned, from the above we can see that the IAI should be dynamically configurable and preferably implement some sort of multi-level hierarchy.

C. Fault Indication and Propagation Infrastructure

The Fault Indication and Propagation Infrastructure (FIPI) is an important part in the demonstrator. Faults that are detected in the instruments, inside blocks which in turn are inside components, might threaten the operation of the complete MPSoC, unless proper action is taken. The time between the manifestation of the fault and the corresponding action is critical, not just for correct operation, but also for performance. Therefore, the FIPI implements an interrupt-like function that propagate fault indications, in instruments that are capable of fault detection, to a system-level fault indication instrument that holds a global fault indication. In other words, the fault indication at the fault detection instrument identifies the fault origin, but this information takes a certain time to access. In contrast, the system level fault indication instrument just indicates that a fault is present somewhere in the MPSoC, but can be quickly accessed, much like an interrupt. To facilitate identification of the fault origin, the FIPI implements a traceable path back to the instrument (or instruments) that first raised the alarm.

D. Resource Manager

The purpose of the Resource Manager (RM) is to obtain correct system operation and maximize the system's throughput. The RM is implemented as software that runs on the master CPU, and it performs these two main tasks:

- Schedules and assigns (distributes) jobs to available components (CPUs, DSPs, accelerators, *etc.*), and
- Provides fault handling, *i.e.* keeps track of defective components, with the goal of increasing both system availability and reliability.

For scheduling, the RM assumes given is a list of jobs and the number of components from each component type in the MPSoC. Each job in the job list is associated with an execution time and a criticality (priority) level. The goal of the RM is to assign jobs to the components, such that the total time required for all the jobs to be completed while utilizing the available components in the MPSoC is minimized. Due to the fact that faults may occur during the execution of jobs and that some components may become defective during system operation, the RM needs to provide some fault handling to obtain correct system operation.

For fault handling, close system observation is required from the RM to get information regarding the status of the MPSoC, *e.g.* answers are required to questions such as: Are there any errors in the CPUs on which jobs are being executed? Are there any defective parts in the MPSoC? There are multiple alternatives on how the fault handling

is to be carried out. The demonstrator provides a number of features to facilitate fault handling.

To be able to detect malfunctioning of any of the components, it is required that the RM monitors the operation and constantly collects fault statuses by accessing the different instruments. The collected data is analyzed and appropriate action is carried out by the RM. An appropriate action can be to employ a fault-tolerant technique to a job or to initiate a functional test or a Built-In Self-Test. Based on the collected data, the RM can perform fault marking of components through a System Health Map (SHM) that keeps track on defective components. The RM inspects the SHM when scheduling jobs to identify fault-free components on which the jobs can be executed.

E. Instrument Manager

The main objective of the Instrument Manager (IM) is to carry out the communication between the RM and the instruments. In that sense, the IM behaves as a mediator between the RM and the instruments.

As discussed in the previous section, the RM has only the global view of the MPSoC in terms of types and number of components. In contrast to the RM, the IM contains structural information of the system, meaning that the IM is aware of the different instruments, and how they are connected through the IAI. Given the structural information, the IM is able to translate the requests from the RM into physical addresses of the instruments.

A typical scenario for the IM would be as follows. The RM sends a request to the IM through a specific high-level command. Whenever the IM receives a high-level command, the IM translates the high-level command into a low-level command (bit stream) that is sent into the hardware through the IAI. Once the low-level command is performed and a low-level response in form of a bit stream is received, the IM translates the response to some high-level response (*e.g.* response from read or write command) that is sent to the RM.

F. Fault Injection Manager

To be able to simulate the system in presence of faults, there is a need to have a mechanism to inject faults. The Fault Injection Manager (FIM) is developed to enable testing of different fault handling solutions.

The FIM is able to inject soft (transient) and hard (permanent) faults in any instrument, given a list of faults. The list of faults specifies for each fault which instrument to inject into, the type of fault (soft or hard) and the time at which the fault should manifest.

In Section III-C, we discussed that fault indication is incorporated within the instruments. We assume that the contents of these instruments are results of some underlying error detection mechanism. However, the error detection mechanisms are not always perfect, *i.e.* an error detection mechanism may identify a presence of fault even when no such fault actually exists in the system. Therefore, another useful feature of the FIM is the ability to inject "false" faults, *i.e.* faults that have not occurred in reality.

IV. Case Study

In this section, we use our demonstrator in a case study demonstrating: 1) fault injection, 2) fault detection, 3) fault identification, 4) fault marking and 5) recovery. For that purpose, we implemented the demonstrator and simulated a scenario where a permanent fault occurs in a CPU register. We illustrate all needed steps taken by the RM to detect the fault, identify the fault location, fault mark the defective CPU and recover the operation.

Due to that some of the parts of the demonstrator were implemented in VHDL (MPSoC, IAI, FIPI and FIM) and other parts in software (RM and IM), we used commercial hardware-software co-simulation tools.

Below we first describe the implementation of the demonstrator, and then we show how the demonstrator carries out the before mentioned scenario.

A. Demonstrator implementation

The MPSoC is implemented in VHDL and it consists of one master CPU, 10 identical work-horse CPUs and two DSPs (the DSPs are due to space limitation not detailed). The structure of a work-horse CPU is depicted in Figure 3. A work-horse CPU consists of two blocks: ALU and CTRL (control). The ALU block contains two instruments, *i.e.* a scan chain and a register file. The CTRL block contains a Program Counter (PC) to emulate execution of jobs. The DSPs are not detailed further as the case study focuses on a fault in a CPU.

The IAI is implemented to follow the IEEE P1687 standard. The reason behind this is to enable flexible access to instruments. A key component that enables flexible access in IEEE P1687, is the Segment Insertion Bits (SIBs) which make the network (scan-path) dynamically configurable. We have designed the IAI such that the instruments, including the IAI specific instruments (status registers), which we address as EIF (Error Indication Flag), are connected in a mutli-level hierarchy (tree-like structure) (see Figure 2 and Figure 3). The advantage with the design is that we easily can access different instruments, *e.g.* the system-level EIF can be accessed by an overhead of accessing only two SIBs (see Figure 2).

By programming (opening) the SIBs, it is possible to access any of the IAI levels (System- level, Component-Type-Level, Component-Level, Intra-Component-Level and Instrument-Level). As the IAI is implemented in a tree-like structure, opening a given SIB that belongs to a given hierarchical level allows access to isolated set of SIBs (subtree) that belong to the lower hierarchical levels. For example if we in Figure 2 open the leftmost SIB in the Component-Type-Level, we allow access to the SIBs that belong to the lower levels (that belong to the CPUs).

The EIFs contain fault indication information. At Instrument-Level, there is one EIF bit per instrument, at Intra-Component-Level there is one EIF bit per component block, at Component-Level there is one EIF bit per component, at Component-Type-Level there is one EIF

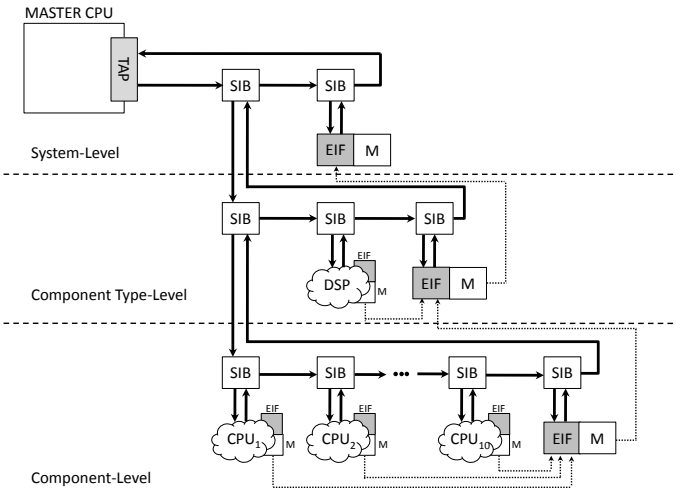

Figure 2: Tree-like IAI and FIPI

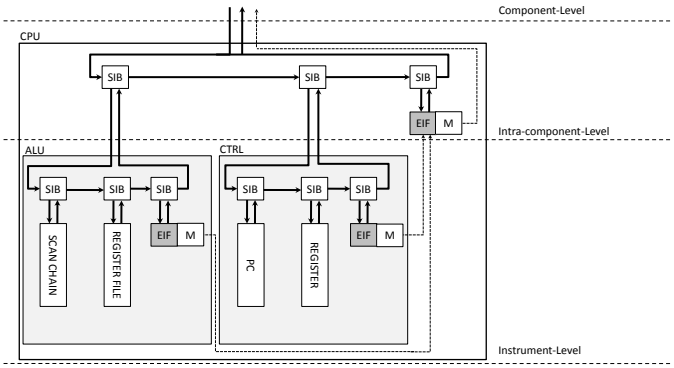

Figure 3: Details on CPU with IAI and FIPI

bit for each component type, and at System-Level there is one EIF bit.

The FIPI, implemented in VHDL, is a combinational logic that enables the contents of an EIF at one level in the IAI to be propagated to the EIF of the upper level. In this way fault effects are immediately reported from a lower level to the system-level (root) EIF. From each EIF, the FIPI propagates a single bit to an EIF in the upper level, by performing an OR operation on all the bits in the lower level EIF. For example, if a fault occurs in a register in the ALU, the corresponding bit in the Instrument-Level EIF of the ALU is set. An OR operation is performed on the Instrument-Level EIF bits, and a single bit (corresponding to the ALU block) is set in the Intra-Level-Component EIF of the CPU. The fault indication is further propagated to the System-Level EIF. To enable fault marking, the FIPI associates a mask register (M) for each EIF. The mask register is of the same size as the EIF. Setting the mask register at an EIF blocks fault propagation upwards.

The RM, which in this case study is straight forward, constantly polls the System-Level EIF to check if there are any faults in the system. If a fault is detected, the RM traces the fault and identifies the CPU where the fault has occurred by stepping through the hierarchy. Note that to find the root cause of a fault, several iterations are needed.

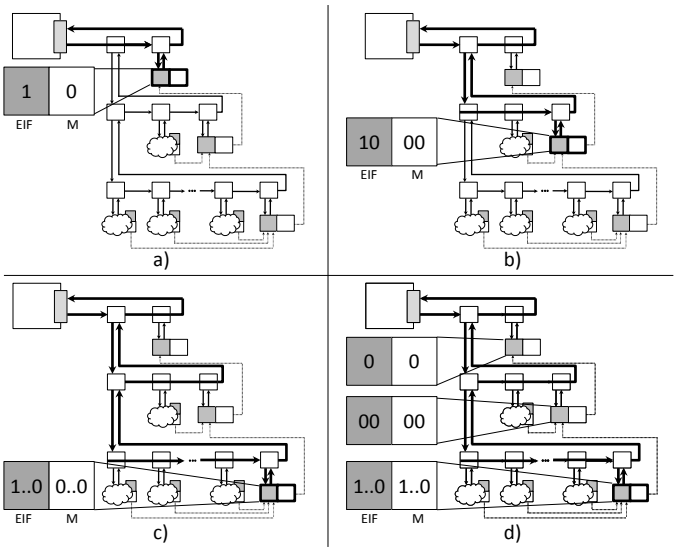

Figure 4: Fault identification through reconfiguration of IAI (IEEE P1687 network)

In each iteration, new commands are given to the IM and the RM gets more knowledge on the fault location. Once the fault is identified, the RM first performs fault marking, *i.e.* updates the SHM, and issues a command to the IM to write in the corresponding mask register, and after that it forces re-execution of the job on a different CPU.

The IM receives commands from the RM. When the IM receives a command from the RM, the IM first configures the SIBs in the IAI such that required instruments are accessed. Once the setup is done, the actual instrument or instruments are accessed.

The FIM inject faults that are specified in a given file, *i.e.* FaultList. The FaultList contains one line for each fault that is to be injected. For each fault given is: a time stamp that represents the time at which the fault is to be injected, a unique address of the instrument where the fault should occur (fault location), the fault effect of the inserted fault (e.g. Sa0, Sa1), the fault type soft or permanent.

B. Demonstrator in operation

In this section, we detail the needed steps required to detect, identify, fault mark and recover from a permanent fault that occurs in the register file, in the ALU of CPU1 after some time in operation. The steps are as follows:

1) Injection: To inject the fault, we generate a FaultList file that contains a single fault (see Figure 5). The FIM reads the FaultList and injects the fault by the following scheme: when the simulation time reaches 44ns, bit 1 (this bit corresponds to the register file) in the Instrument-level EIF of ALU block in CPU_1 is set.The FIPI propagates the fault upwards in the IAI hierarchy such that corresponding bits of the EIF at Intra-Component-level, Component-level, Component-Type-Level and System-Level are set.

2) Detection: During operation, the RM constantly checks if a fault has occurred by sending commands to the IM for checking the EIF bit at System-Level (polling). At detection (polling), for each poll the IM configures

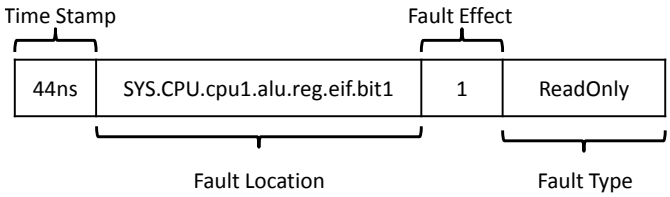

Figure 5: Fault description

the SIBs such that the System-Level EIF is accessed and checked (see Figure 4(a)). If the System-Level EIF is set, a fault in the MPSoC is detected. Once the RM detects that there is a fault, the RM tries to identify the defective component.

3) Identification: Once the RM detects that there is a fault somewhere in the system, the RM sends a command to the IM to check which *type* of component is defective. This is done by checking the Component-Type-Level EIF. The IM configures the SIBs such that the Component-Type-Level EIF is accessed, and reports, in this case, that there is a fault in one of the CPUs (the bit corresponding to the SIB which is connected to the CPUs is set, see Figure 4(b)). Since at this point the RM is aware that there is a faulty CPU, the RM then gives a command to the IM to check which CPU is defective, *i.e.* checks the Component-Level EIF of the CPUs. The IM re-configures the SIBs such that the Component-Level EIF of the CPUs is accessed (see Figure 4(c)), and reports CPU_1 as faulty.

4) Fault Marking: Once the fault is identified, the RM updates the SHM such that CPU_1 is marked as defective and then issues a command to the IM to write in the mask register associated to the Component-Level EIF of the CPUs (observe the contents of the mask register in Figure 4(d)) such that no further faults are propagated from CPU_1 over the FIPI.

5) Recovery: The recovery process, consists of moving the job that was running on CPU_1 to another CPU. The RM checks the SHM to find an idle fault-free CPU. If successful, RM issues a re-execution of the job. After this step, the RM continues polling the System-Level EIF.

V. Conclusion

In this paper we present an MPSoC demonstrator useful to explore in-operation fault injection and fault handling strategies. A key feature of the demonstrator is the combination of the fault propagation architecture and the IEEE P1687 architecture, which gives enables minimized time between fault detection at a component and the fault action at the system-level and allows flexible access to any part of the system. Through a case study, we have with the demonstrator shown the steps from fault injection, fault detection, fault identification, fault marking, and recovery.

References

[1] Y. Xiang, T. Chantem, R. P. Dick, X. Hu, and L. Shang, "System-level reliability modeling for MPSoCs," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, Oct. 2010, pp. 297 –306.

[2] N. Hébert, G. Almeida, P. Benoit, G. Sassatelli, and L. Torres, "A cost-effective solution to increase system reliability and maintain global performance under unreliable silicon in MPSoC," in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, Dec. 2010, pp. 346-351.

[3] H.-M. Pham, S. Pillement, and D. Demigny, "Evaluation of fault-mitigation schemes for fault-tolerant dynamic MPSoC," in *Field Programmable Logic and Applications, 2010 International Conference on*, 31 2010-Sept. 2 2010, pp. 159-162.

[4] J. Rearick, B. Eklow, K. Posse, A. Crouch, and B. Bennetts, "IJTAG (internal JTAG): a step toward a DfT standard," in *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, Nov. 2005, pp. 8 pp. -815.

[5] K. Posse, A. Crouch, J. Rearick, B. Eklow, M. Laisne, B. Bennetts, J. Doege, M. Ricchetti, and J.-F. Cote, "IEEE P1687: Toward standardized access of embedded instrumentation," in *Test Conference, 2006. IEEE International*, Oct. 2006, pp. 1-8.

[6] A. Crouch, "IJTAG: The path to organized instrument connectivity," in *Test Conference, 2007. IEEE International*, Oct. 2007, pp. 1 –10.

[7] J. Doege and A. Crouch, "The advantages of limiting P1687 to a restricted subset," in *Test Conference, 2008. IEEE International*, Oct. 2008, pp. 1 –8.

[8] J. Rearick and A. Volz, "A case study of using IEEE P1687 (IJTAG) for high-speed serial I/O characterization and testing," in *Test Conference, 2006. ITC '06. IEEE International*, Oct. 2006, pp. 1 –8.

[9] F. Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson, "Test time analysis for IEEE P1687," in *Test Symposium (ATS), 2010 19th IEEE Asian*, Dec. 2010, pp. 455 –460.

[10] ——, "Design automation for IEEE P1687," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1 –6.

[11] A. Jutman, S. Devadze, and J. Aleksejev, "Invited paper: System-wide fault management based on IEEE P1687 IJTAG," *Reconfigurable Communication-centric System-on-Chip (ReCoSoC), 2011 6th International Workshop on*, June 2011.

[12] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatryk, "Fingerprinting: bounding soft-error-detection latency and bandwidth," *Micro, IEEE*, vol. 24, no. 6, pp. 22 –29, Nov.2004.

[13] D. Blaauw, S. Kalaiselvan, K. Lai, W.-H. Ma, S. Pant, C. Tokunaga, S. Das, and D. Bull, "Razor II: In situ error detection and correction for PVT and SER tolerance," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, Feb. 2008, pp. 400 –622.

[14] M. Majeed, D. Ahlström, U. Ingelsson, G. Carlsson, and E. Larsson, "Efficient embedding of deterministic test data," in *Test Symposium , 2010 19th IEEE Asian*, Dec. 2010, pp. 159 –162.

[15] D. Blough and T. Torii, "Fault-injection-based testing of fault-tolerant algorithms in message-passing parallel computers," in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, June 1997, pp. 258 –267.

[16] A. Ademaj, P. Grillinger, P. Herout, and J. Hlavicka, "Fault tolerance evaluation using two software based fault injection methods," in *On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International*, 2002, pp. 21 – 25.

[17] O. Ballan, U. Rossi, A. Wantens, J.M. Daveau, S. Nappi, and P. Roche, "Verification of soft error detection mechanism through fault injection on hardware emulated platform," in *Dependable System and Network Worskhops (DSN-W), 2010 International conference on*, June 2010.

[18] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto, "A framework for reliability assessment and enhancement in multi-processor systems-on-chip," in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on*, Sept. 2007, pp. 132 –142.

[19] "IEEE P1687/D1.25 Draft Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device"