POLITECNICO DI TORINO

Master's Degree in Micro- and Nanotechnologies for Integrated Systems

Master's Degree Thesis

Design and Integration of a Debug Unit for Heterogeneous System-on-Chip Architectures

Supervisors

Prof. Luca CARLONI

Dr. Paolo MANTOVANI

Prof. Luciano LAVAGNO

Candidate

Gabriele TOMBESI

October 2020

Summary

The purpose of this dissertation is to provide a flexible, platform-based pre- and post-silicon verification methodology, leveraging Design for Testability (DFT) to address the increasing complexity of modern heterogeneous System-on-Chips (SoCs). The proposed approach is applied for the first time in the context of an ongoing project at Columbia University, which aims at the development of a companion agile flow for SoC Application-specific-integrated-circuit (ASIC) implementations targeting advanced technology nodes.

With the end of Dennard scaling, the increasing demand for power-efficient systems faced by the semiconductor industry has promoted the transition from homogeneous to heterogeneous architectures, which couple general-purpose processors with a growing number of specialized hardware accelerators. Academia is putting a significant effort in investigating architectures for several domain-specific accelerators and providing innovative solutions addressing the integration challenges in heterogeneous systems. In most of the cases, the instruction set architecture (ISA) of the RISC-V[1] project is leveraged for the processor cores of the proposed platforms. Among the existing implementations, the Embedded Scalable Platform (ESP) project developed at Columbia University plays a distinctive role. As opposed to other open-source RISC-V platforms, ESP promotes a system-centric view, based on the coupling of a modular socket-based architecture with a companion system-level methodology [2]. The goal of this approach is to relieve the designer from the burden of the intellectual property (IP) integration process, by decoupling the IP design from the rest of the system. At the same time, ESP promotes a shift of the engineering effort by moving the abstraction level from register-transfer level (RTL) specifications to a system-level viewpoint leveraging high-level-synthesis

[1]RISC-V is an instruction set architecture (ISA) based on reduced instruction set computer (RISC) principles, developed at the University of California, Berkeley, in 2010. Unlike many other ISA, it is an open source standard which does not require royalties to be used. Both Academia and Industry have announced RISC-V hardware in the past decade, with over 65 RISC-V cores available today [1]

(HLS) flows [3].

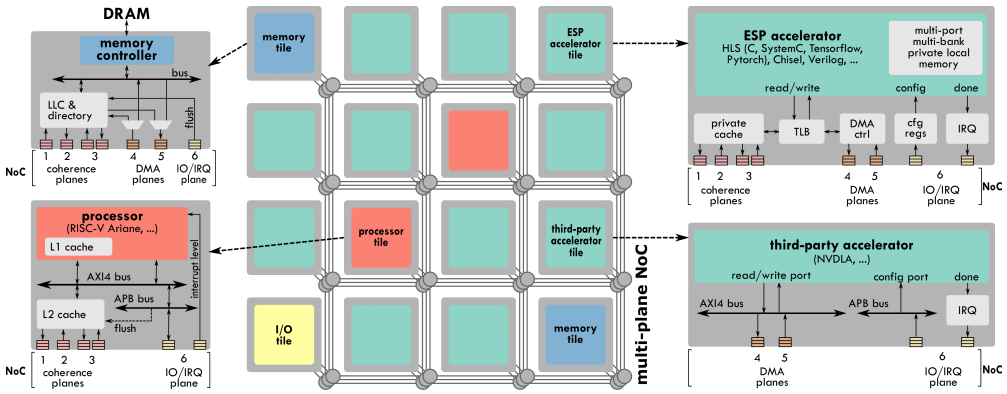


Figure 1: Overview of ESP tile-based architecture [4].

Several ESP instances capable of booting Linux have already been prototyped on different classes of FPGAs in the past decade [5]. Nevertheless, in the context of chip manufacturing, pre-silicon verification and design for testability assume a fundamental role. In this perspective, this dissertation addresses some of the challenges related to the testing, debugging and verification of a heterogeneous platform modules. Specifically, I developed a modular Debug Unit for the pre- and post-silicon unit test of IP blocks integrated in each ESP tile. The approach adopted in this work adapts well to the platform-based architecture of ESP; in fact, my Debug Unit implements a new platform service that enables the independent test of every tile, decoupled from the system interconnect. Throughout the dissertation, I give a comprehensive overview of the steps involved in the design and integration of the Debug Unit in the system. Figure 2 shows a high-level view of the tile's new internal arrangement after the Debug Unit integration, in normal and test operating mode.

The work presented is organized in five chapters. Chapter 1 is an introduction to ESP. The different design flows supported by the system-level companion methodology are presented. Amongst those, a more accurate description is dedicated to the design flows leveraging commercial high-level-synthesis (HLS) tools. The tile-based architecture of the platform, summarized in Figure 1, is then discussed showing the services offered by each type of socket for the transparent integration of in-house and external IPs, together with the latency-insensitive communication infrastructure connecting different tiles. Chapter 2 analyses the state-of-the-art testing strategies adopted so far in post-manufacturing SoC structural verification: Scan Chain insertion and Built-in self-test (BIST) . In addition, it introduces the methodology supported in this dissertation, which leverages DFT for functional tests of single SoC modules. Chapter 3 derives the test interface specifications,

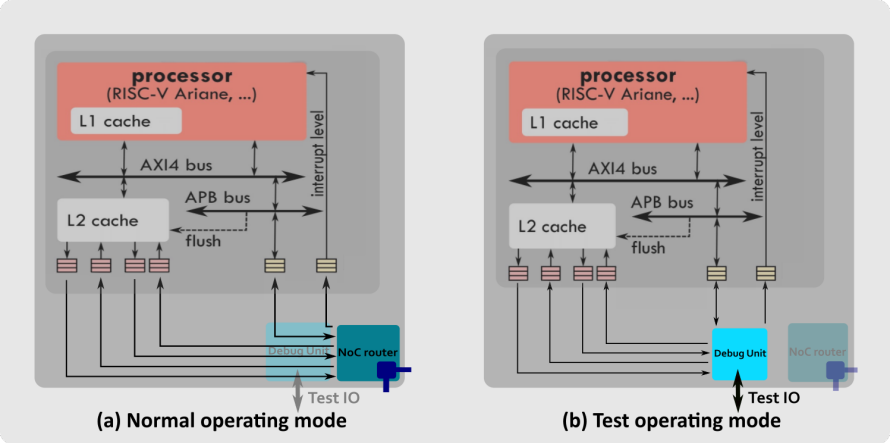


Figure 2: Proposed approach for a tile functional test.

taking the backend flow constraints largely into account. The entire stack of the test flow is described and the FPGA-based setup used for testing is illustrated. Finally, a particular focus on the types of test application targeted is provided. Chapter 4 describes the design and integration of a Debug Unit which applies the verification methodology proposed inside ESP. A high level view of the implemented test unit integrated in the ESP tile is showed in Figure 2. After describing two alternative RTL implementations, the test interface performances are evaluated with RTL simulations for both of them, highlighting the improvements delivered by the second design version. The final chapter draws the main conclusions related to the test methodology proposed in this dissertation, which confirm the thesis that DFT can be leveraged to provide a general testing approach, not limited to ESP, but generally applicable to address the increasing complexity of heterogeneous SoC designs. In addition, it introduces a BIST-oriented approach to improve the Debug Unit test performances, which consists in integrating a private memory inside the tile under test in order to reduce the latency of flits transfers during the test.

Acknowledgements

I would first like to express my sincere gratitude to my Scientific Supervisor, Prof. Luca P. Carloni, whose expertise was invaluable in formulating the research questions and methodology.
In addition, I would like to thank my Scientific Co-supervisor, Dr. Paolo Mantovani, for his consistent support and valuable guidance throughout my research activity. His knowledge and insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.
Besides my supervisors, I also wish to thank my Thesis Advisor, Prof. Luciano Lavagno, for his professionality and continuous availability during all this period.
Finally, I must express my very profound gratitude to my parents and my brother, for providing me with unfailing support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them.

Table of Contents

List of Tables IX

List of Figures X

Acronyms XIII

1 Introduction to ESP 1
- 1.1 Raise of Heterogeneous Computing . . . 1
- 1.2 ESP Methodology . . . 4
- 1.3 ESP Architecture . . . 7
  - 1.3.1 NoC . . . 8
  - 1.3.2 Tiles . . . 13

2 SoC Test trends 18
- 2.1 Functional Test . . . 19
- 2.2 Structural Test . . . 20
  - 2.2.1 Design for testability . . . 21
- 2.3 Conclusions . . . 24

3 ESP Test approach 26
- 3.1 Backend flow constraints . . . 26
  - 3.1.1 NoC bypass . . . 26
  - 3.1.2 Ethernet . . . 27
  - 3.1.3 Conclusions and additional remarks . . . 31
- 3.2 Test interface specifications . . . 32
  - 3.2.1 Insertion point and Debug Unit interface . . . 32
  - 3.2.2 Pin count constraints and pin sharing . . . 37
- 3.3 Test flow . . . 41
- 3.4 Test programs . . . 44

4 Debug Unit Design 48
4.1 First version - single register . . . 49
4.1.1 FSM . . . 49
4.1.2 Datapath . . . 51
4.1.3 Simulation . . . 53
4.1.4 Conclusions . . . 59
4.2 Second version - multiple registers . . . 60
4.2.1 FSM . . . 61
4.2.2 Datapath . . . 63
4.2.3 Simulation . . . 65
4.2.4 Conclusions . . . 69

5 Conclusions 71
5.1 Future improvements . . . 71

A End of Dennard Scaling 74

B AMBA-AXI protocol 79

C CPU tile operation 82

Bibliography 85

List of Tables

3.1 I/O pins partition for a 16-tiles ESP ASIC instance. 41
3.2 NoC planes purposes from the CPU tile perspective. 42

4.1 Simulation trace for the *load* and *store* instructions. 54

List of Figures

1 Overview of ESP tile-based architecture [4]. iii
2 Proposed approach for a tile functional test. iv

1.1 Technology scaling trends [7]. 2
1.2 HLS-based design space exploration for a single SoC component [2]. 6
1.3 Pareto migration from compositional to system-level DSE [3]. . . . 6
1.4 Design and integration flows supported by ESP methodology [2]. . . 7
1.5 ESP modular architecture [5]. 8
1.6 ESP NoC interface [2]. 10
1.7 LID protocol and Shell encapsulation in ESP. 13
1.8 Tightly-coupled accelerator model [3]. 15
1.9 Co-processor model [3]. 15
1.10 Loosely-coupled accelerator model [3]. 15
1.11 Overview of accelerator, processor and memory sockets, together with the related platform services [4]. 17

2.1 Rule of ten for ICs test costs [19]. 19
2.2 General Test approach [18]. 19
2.3 Relevant examples of chip defects [20]. 21
2.4 Scan chain implementation. 23
2.5 BIST implementation [19]. 23

3.1 (a)Typical Debug communication used in LEON system based on SPARC Architecture [25]. (b) DSU default arrangement in ESP. . . 29
3.2 (a) Debug Unit placement in the preexisting Design. (b) Tile's operating modes. 33
3.3 Debug Unit interface. 35
3.4 Pin sharing approach for ESP Debug Unit. 39
3.5 General Chip-FPGA test plan. 40
3.6 Input flits format. 44
3.7 Test flow. 45
3.8 Startup.S : default test application stored in the bootrom. 46

3.9 main.c: program printing "Hello from ESP!" to the UART peripheral. 46

4.1 FSM of Design 1. 50
4.2 Datapath of Design 1. 52
4.3 Execution of *store* instruction in test operating mode. 55
4.4 Execution of *load* instruction in test operating mode. 55
4.5 **Phase 1**: The test logic is performing correctly but the FIFO queue connected to NoC plane 1 is gradually filled up. 57
4.6 **Phase 2**: The test logic is still performing correctly, but the FIFO queue gets full and the AXI write destination switches to channel 5. A flit is written in this queue. 58
4.7 **Phase 3**: The flit from channel 5 is selected following a fixed priority implemented by the FSM, causing the failure of the next read and check. 59
4.8 FSM revision. 62
4.9 Datapath revision. 64
4.10 Test interface activation sequence. 66
4.11 Test: initial injection phase across multiple planes and detailed view of the first serial injection on NoC plane 3. 67
4.12 Detailed view of the first serial injection on NoC plane 5. 67
4.13 READ & CHECK sequence on NoC plane 5. 68
4.14 READ & CHECK sequence on NoC plane 1. 68

5.1 Third operating mode for Built-in unit test. 73

A.1 Technology scaling effect on performance [33]. 74
A.2 (a)Scaling of a traditional CMOS technology. (b) Technology scaling rules [34]. 75
A.3 (a)Power trends upon aggressive gate scaling [36].(b) MOSFET transfer characteristics in subthreshold region [35]. 77
A.4 Fin FET technology A.4. 78
A.5 FD-SOI techonology A.5. 78

B.1 Classical AMBA-based SoC [38]. 79
B.2 (a)AXI Read transaction (b) AXI Write transaction. 80
B.3 AXI handshake examples [41]. 81

C.1 Detailed internal structure of the CPU hardware socket. 84

Acronyms

SoC
System-on-Chip

ESP
Embedded Scalable Platform

HLS
High-level synthesis

DSE
Design Space Exploration

RTL
Register-transfer level

VHDL
Very High Speed Integrated Circuit Hardware Description Language

CPU
Central Processing Unit

OS
Operating System

LID
Latency-insensitive Design

NoC
Network on Chip

IP

Intellectual Property

DMA

Direct memory access

LCA

Loosely coupled accelerator

DSU

Debug Support Unit

EDCL

Ethernet Debug Communication Link

JTAG

Joint Test Action Group

UART

Universal Asynchronous Receiver-Transmitter

DUT

device under test

DFT

Design for Testability

TDI

Test Data in

TDO

Test Data out

TMS

Test Mode Select

TCLK

Test Clock

Chapter 1

Introduction to ESP

This chapter presents the open source Embedded Scalable Platform (ESP), which is used to apply and evaluate the verification methodology proposed in this dissertation. Section 1.1 reports a general overview of heterogeneous computing as a new alternative to face the crisis of technology scaling affecting semiconductor industry in the past decades. Among the ongoing open source heterogeneous platforms, ESP is introduced with its distinguishing features. In Section 1.2, the system-level companion methodology of the platform is described, and the different design flows supported are presented. Amongst those, a more accurate description is dedicated to the design flows leveraging commercial high-level-synthesis (HLS) tools. In addition, the advantages of full-system simulation promoted by the ESP virtual platform are stressed and supported by examples of system-level design-space-exploration (DSE). Finally, section 1.3 offers an overview of the tile-based architecture of the platform. A particular focus is dedicated to the description of the platform services offered by each type of socket, as well as to the presentation of the latency-insensitive communication infrastructure connecting different tiles.

1.1 Raise of Heterogeneous Computing

The end of Dennard scaling, discussed in detail in Appendix A, marked the beginning of a revolution in the way of looking at computing platforms. Even if the device geometry-scaling has confirmed Moore's law predictions thanks to a skyrocketing progress of lithography and fabrication processes, the power dissipation and operating frequency of the system have stopped scaling as expected in the past decade. As a consequence, an increasing gap between the system energy-efficiency and integration capacity has consolidated over the years [6], as showed in Figure 1.1.

To face this challenge, there have been several studies about non-classical CMOS

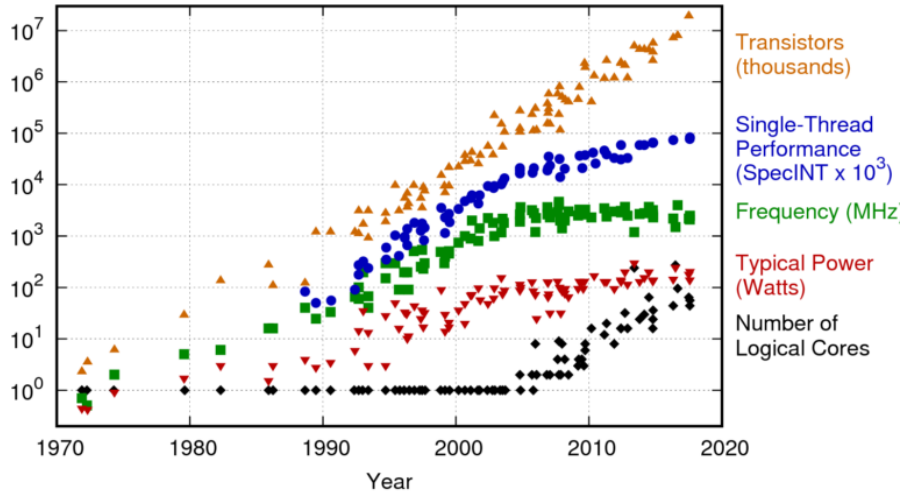


Figure 1.1: Technology scaling trends [7].

devices, based on innovative technology boosters and conduction mechanisms which deliver higher performances. Nevertheless, in order to cope with the increasing gap while respecting the market road-map, an immediate change of paradigm was necessary at architectural level. In this perspective, the advent of multi-core architectures from 2005, introduced the possibility to exploit higher parallelism by running different threads on multiple cores. The base idea laying behind this approach is that the only way to prevent damages to the circuit, due to excessive power dissipation, is to use just a part of it at a time. In particular, depending on the type of application workload currently run, just a part of the logic is activated to execute while the rest, commonly referred to as *dark silicon*, is turned off. In addition, by reducing the operation frequency of each core, a subset of the cores can run parallel workloads within the system power budget [3]. Of course a tight hardware-software collaboration is crucial to exploit the increased integration density with this approach: it is key that the software application is developed with the same degree of parallelism, so that multiple execution threads can be used.
With the wide range of domain-specific applications developed in recent years, however, the increased hardware parallelism offered by multi-core processing could not keep up with the required computational effort. A growing demand for performances focused on specific types of applications has gradually started to dominate the market at this stage. Aggressive hardware specialization was the only remaining path for the architecture community to accomplish such requirements. This is how customized hardware accelerators, capable of performing a restricted set of tasks in an optimized way [8], started to be integrated in embedded system together with microprocessors. Even if the cost of development and production

of customized hardware was extremely difficult to support at the beginning, it got soon overwhelmed by the benefits coming with it. In particular, customized computational units delivered a significant performance increase when coupled with general-purpose microprocessors. This is mainly because each accelerator comes not only with its own optimized way of processing the target data-workload, but also with a customized interconnect for data movement [6]. In addition, the advent of dark silicon makes the impact on the overall power envelope negligible since each specialized unit can be switched off when not used during the program execution. To sum up, different ways to handle the crisis of Dennard scaling have been proposed in the past decades. Amongst those, the integration of several components, differing in customization level and functionality, into heterogeneous computing platforms, has emerged as the optimal approach to satisfy the market demands.

In this perspective, several projects from the open source hardware community (OSHC) have provided significant contribution to the progress of heterogeneous computing platforms. Amongst those, it is worth mentioning the following initiatives:

- **Parallel Ultra-Low Power (PULP)** computing platform, a joint effort of ETH Zurich and University of Bologna. PULP is a multi-core platform addressing the demands of Internet Of Things (IoT) applications. It is characterized by a parallel ultra-low-power programmable architecture which allows to meet the computational requirements of these applications, remaining within the power envelope of a few mW [9].

- **Rocket Chip**, developed at UC Berkeley, is an open-source System-on-Chip (SoC) design generator. It is implemented in Chisel and combines general-purpose processor cores using the open source RISC-V instruction-set-architecture (ISA) with custom accelerators, integrated mainly in the form of instruction set extensions or coprocessors. [10].

- **ESP**: an open-source SoC platform combining a modular and scalable tile-based architecture with a flexible system-level design methodology. The development of this research project at Columbia University was driven by the shift of paradigm faced by the information technology industry, which led to the spread of heterogeneous computing across a large set of domains, from edge applications in embedded systems to data centers. On one side, the heterogeneous architecture of ESP couples general purpose processor with specialized hardware accelerators in the attempt of striking a balance between regularity and specialization. On the other side, the companion methodology raises the level of abstraction from Register-transfer-level (RTL) to higher level descriptions, and takes care of hardware/software integration by enabling several design flows for in-house accelerators [2].

1.2 ESP Methodology

The ESP methodology is mainly driven by the necessity to continue sustaining the progress of the semiconductor industry while facing the growing complexity of SoC design [5]. It lays its foundation on a high degree of flexibility, necessary to accomodate different design flow and CAD tools. As of today, ESP already supports several design flows, making it possible for designers working at different abstraction levels to contribute to the platform IP library. In particular, different design flows are supported by ESP to obtain an accelerator implementation:

1. The traditional and cycle accurate **RTL flow** can be used starting from standard hardware description languages such as Verilog, VHDL or Chisel.

2. Commercial high-level-synthesis tools combined with in-house ESP automation tools can be used to create accelerators starting from loosely-time or un-timed behavioral description of the system in C-like languages (mainly C++ and System C). The most relevant HLS flows supported by ESP include: C/C++ with **Xilinx Vivado HLS** and **Mentor Catapult HLS**, SystemC with **Cadence Stratus HLS** [2] .

3. The open source **hls4ml** flow can be deployed to obtain synthesizable accelerators from domain-specific libraries for deep learning like Keras, Tensorflow and Pytorch.

Amongst those supported, the HLS-based flow is worth of a more accurate analysis, since it represents a key element of the ESP vision. In particular, the HLS flow shifts the engineering effort from low-level and time-consuming RTL descriptions to system-level specifications written with high-level languages. With the increasing complexity of modern SoCs, several features of the RTL flow lead to suboptimal results. In fact, each point in the multi-objective design space differs significantly from its neighbours in terms of micro-architecture, meaning that a significant portion of the RTL code needs to be changed to target the optimal design, with a high probability of incurring in bugs while moving across the different layers of abstraction [3]. In contrast, HLS can be leveraged to automatically generate many RTL implementations from a single system-level synthesizable specification, promoting the exploration of a vast design space with many micro-architectural alternatives [2]. Such a variety of RTL implementations can be obtained by using a rich set of configuration knobs normally offered by state-of-the-art HLS tools. Some of the most common HLS directives are *function inlining*, *function sharing*, *loop pipelining*. In addition, timing constraints on the clock can be imposed, thereby impacting the number of states included in the RTL code (differently from synthesis, where a stronger timing constraints impacts on the technology to choose [3]). Given a high-level description, different combinations of these directives passed

to the HLS engine produce alternative RTL implementations, corresponding to different points in the multi-objective design space, as showed in Figure 1.2. It is important to remark that all these implementations are not equivalent from an RTL viewpoint, since they do not produce exactly the same outputs, clock by clock, for a certain sequence of input signals, but they are still valid RTL implementations of the original SLD specification [5]. To be more precise, they all belong to a latency-equivalent class and any of them can be integrated within an ESP instance because its tile socket implements a latency-insensitive protocol [5]. To clarify this aspects, the concept of latency-insensitivity design will be furtherly discussed in the dedicated Section 1.3.1. After selecting the option that reflects the best performance trade-off, a logic synthesis step is necessary to generate the scheduling and resources for the hardware implementation: the HLS tool uses the estimation of the performances for each primitive operation and generates the resources according to the initial timing constraint, using a scheduling algorithm which is typically the List-Scheduling (LS) algorithm [11]. ESP actively contributes to the process of obtaining a suitable IP implementation by providing accelerator templates to simplify the design stage, as well as fully-working accelerator skeleton obtained from a set of parameters passed by the designer (namely name ID, desired HLS flow, configuration registers, bitwidth of data tokens and size of batches to execute without CPU interruption [2]).The proposed HSL-based flow and the associated application-level design space exploration is crucial for an IP designer seeking to integrate the accelerator in a pre-existing system with certain features and requirements. Nevertheless, when it comes to building an SoC from scratch, the SoC architect needs to take into account a bigger set of parameters derived from the overall operation of the system and not limited to a single computational unit. In this case the target of the designer is not only the specific accelerator implementation, but the optimal mix of tiles, including the number of instances per accelerator to run a certain application. Unfortunately, this can not be targeted by simply combining the results obtained from several single-component, application-level DSE (also referred to as compositional DSE) [3].
This is where a second key element of ESP vision comes in: ESP offers a FPGA-based rapid prototyping platform that both hardware and software engineers can use to emulate and validate a component as part of the whole system. The full-system simulation is run on top of the software stack that will be deployed in the final system, including the Operating System (OS) [5]. The main factors taken into account in a system-level DSE are the contention of shared-resources, the overhead of the communication infrastructure involved in p2p communication channels, as well as a set of non-deterministic effect of the operating system and DRAM access [3]. It is worth noticing that, when all these factors are taken into account by the platform simulator, a significant migration of the Pareto points

curve [1] for a single component can take place. In certain cases, implementations belonging to the Pareto curve in the ideal case turn out to be non Pareto optimal at system level, as visible in the example highlighted in Figure 1.3 reporting the ideal and system-level DSE for an accelerator targeting the WAMI app [2]. To conclude, on one side, full-system simulation offers the possibility to target a certain combination of tiles for a specific application hiding the complexity of heterogeneous integration given by the interaction of all system components. On the other side, it encourages hardware/software co-design responding to one of the main premises in the ESP approach: the specific target application workload must drive the software-programming and hardware-design efforts throughout all stages of the SoC realization [5] .

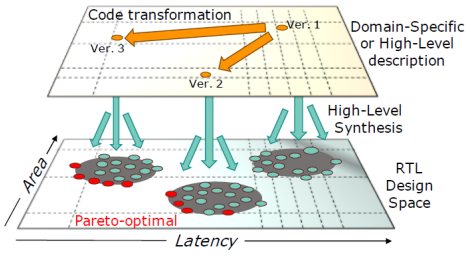


Figure 1.2: HLS-based design space exploration for a single SoC component [2].

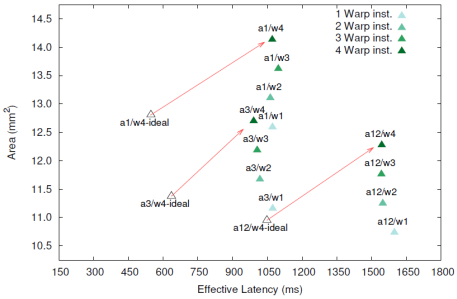


Figure 1.3: Pareto migration from compositional to system-level DSE [3].

The ESP methodology is summed up in Figure 1.4. On the left, the accelerator design flow for the creation of an IP library is showed. On the right side, the SoC flow based on the in-house virtual platform shows how to automate the integration of heterogeneous components into a complete SoC.

[1] In the Design Space, a point is a Pareto point if there is no other point of the design space with all the objectives inferior or equal and at least an inferior objective. The Pareto points curve is the interpolation of all the Pareto points for a certain design evaluation [11].

[2] The WAMI app is an accelerated version of the Wide-Area Motion Imagery (WAMI) application [12]: an image processing application used for aerial surveillance [13]

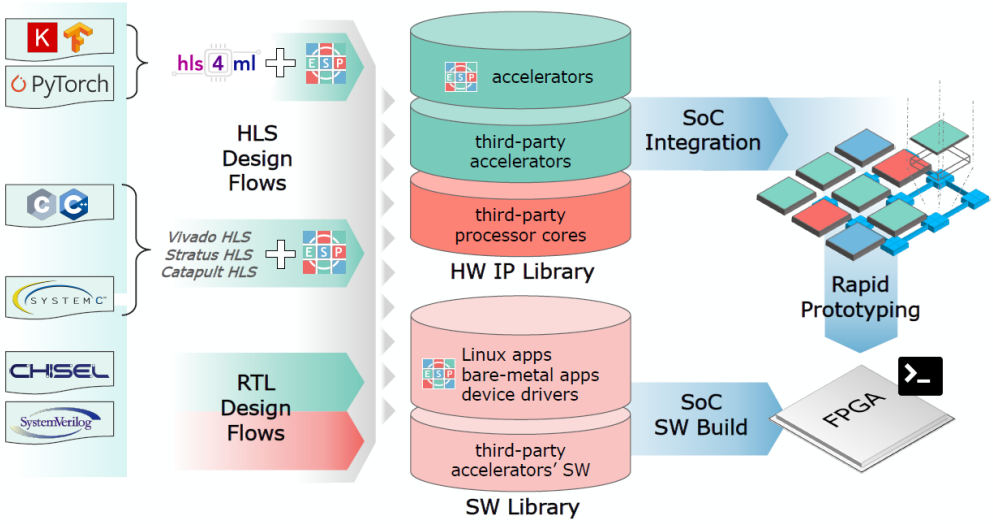


Figure 1.4: Design and integration flows supported by ESP methodology [2].

1.3 ESP Architecture

The ESP design methodology is effective because it was developed together with the companion ESP architecture. The platform is characterized by a modular, tile-based multi-core architecture that aims at striking the right balance between heterogeneity and regularity, implementing a distributed system that is inherently scalable [12]. An example of an ESP instance consisting in a 16-tile SoC organized in a 6 x 6 matrix is showed in Figure 1.5. Each tile can contain a processor, an accelerator, a memory link to off-chip memory, or some I/O peripherals necessary for the platform services. A more detailed description of the internal components of each of them is presented later in this paragraph.

The design environment comes with a Graphical User Interface (GUI) that guides the designer to the interactive floor-planning of the SoC with a push button capability for rapid prototyping of ESP on FPGA. The target combination of tiles can be determined through the aforementioned design space exploration and it is thus strictly driven by the application workload. The key architectural element of ESP tiles is the modular socket interface. A socket interface is used for each tile to interface with the Network-on-chip (NoC) and it performs a set of platform services following the paradigm of latency-insensitive-design[14] discussed in paragraph 1.3.1. As a result, the design of the communication infrastructure of the system is decoupled from the design of the internal component, making the integration of heterogeneous IPs transparent to the IP designer and to the SoC architect. This also applies to the cases where the third-party IP, which needs to be integrated in

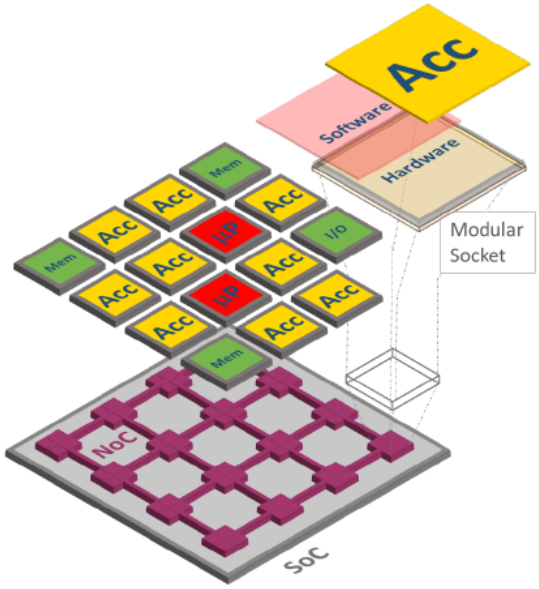


Figure 1.5: ESP modular architecture [5].

the platform, already implements some of the services offered by the ESP socket. Thanks to the modularity of the socket, it is indeed possible to choose the set of services to be instantiated at design time, or enable them with reconfigurability option at run time, so to easily get a simplified version with less services [2]. As of today, ESP enables the transparent integration of newly developed accelerators with third-party intellectual property blocks speaking the Advanced Extensible Interface (AXI) protocol, including the ARIANE 64-bit RISC-V processor core and the NVIDIA Nvdla accelerator.

1.3.1 NoC

Different tiles are connected with a packet-switched, multi-plane NoC characterized by A MESI directory-based protocol [3] and a 2D mesh topology as briefly summarized

[3]Cache memories are adopted to improve the performances of systems implementing the shared memory paradigm. This is done by temporary storing frequently accessed data in the higher levels of the memory hierarchy, i.e. those closer to the processors. The main drawback is that data accesses targeting addresses which are residing in more than one location may incurs in correctness problems. Cache coherence protocol are used to maintain a unified view of the memory hierarchy. Amongst those, directory-based protocols rely on a point to point communication for coherence requests, resulting in lower bandwidth requirements with respect to broadcast protocols [15]. In particular, this is made possible by using a directory which keeps track of the shared

in Figure 1.6. The NoC only interacts with the socket and it can distribute messages across multiple physical planes to maximize the accelerators performance while preventing protocol deadlock. In particular, it provides support for system level coherency on top of three dedicated planes, avoiding any possibility to create a deadlock condition. Two other planes of the NoC are used to handle Direct Memory Access (DMA) requests and response between the accelerator tiles and memory tiles, while an additional plane supports IO/IRQ channels. Those are used for various purposes and in particular to program accelerators. A more deatailed description of the type of messages travelling on each plane of the NoC is here reported:

1. **Coherence planes** :
   (a) Plane 1: Coherence requests from CPU to directory.
   (b) Plane 2: Coherence forwarded messages to CPU.
   (c) Plane 3: Coherence response messages from and to CPU (bidirectional).
2. **DMA planes** :
   (a) Plane 4: DMA response to accelerators and Coherent DMA requests from accelerators (bidirectional).
   (b) Plane 6: DMA requests from accelerators and Coherent DMA response to accelerators (bidirectional).
3. **IO/IRQ plane**
   (a) Plane 5: Remote Advanced Peripheral Bus (APB) requests from processor and APB response to processor for memory-mapped registers configuration. Interrupt requests to processor and remote acknowledge from processor for interrupt handling. Advanced High-performance (AHB) requests to processor Debug Support Unit (DSU) and AHB responses from processor DSU for communication with the Debug Ethernet interface. Other types of transactions speaking the AHB protocol for communication with other peripherals of the IO tile, such as UART [4] and a digital visual interface (DVI) to enable video output(bidirectional).

The router architecture provides look-ahead dimensional routing. In other words, the route computation is removed from the critical path [15] and it is performed

cache lines, and the corresponding current sharers.

[4]The universal asynchronous receiver-transmitter (UART) is an hardware device, usually part of an integrated circuit, used for serial communications over a serial port [16].

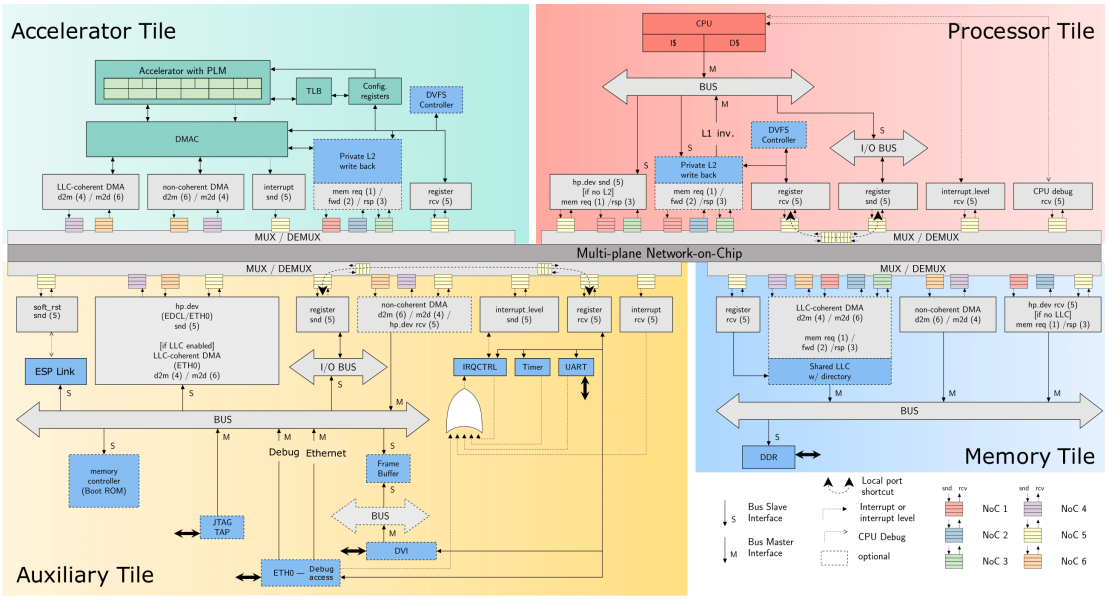


Figure 1.6: ESP NoC interface [2].

in parallel with arbitration [4], so that every hop takes just one clock cycle. All the design choices for the communication infrastructure of ESP are driven by the transaction level modeling (TLM) abstraction: a high-level approach which decouples the details of communication among modules from the details of the implementation of the functional units [17]. As a matter of fact, the NoC acts as a transparent communication layer so that both the processor and accelerators work as if any other component of the SoC was connected to the ports of the local bus in the socket. In the specific case of ESP, this is only made possible by the use of a couple of proxy components instantiated in the tiles for each type of inter-module transaction, showed in gray in figure 1.6. Their task is to translate CPU and accelerators requests into packets suitable for travelling on the NoC infrastructure, as well as turning the messages coming from the NoC back to their original shape when they reach the destination. Each proxy has an associated buffering queue to respect the rules of the latency-insensitive design. This methodology plays a fundamental role in driving some of the most important decisions in the design of the NoC, as well as the socket interface. As a consequence, it deserves a description of its main features that is given in the following subparagraph.

Latency-insensitive Design

The synchronous paradigms applied at RTL defines a digital system as a collection of interacting modules, each one including combinational logic defining the module

functionality and registers storing the state and output variables. At each clock cycle, each module uses the current value of the inputs and the state variables to update the state and output variables. The synchronous hypothesis states that the computation phase (within the functional modules) and the communication phase (transferring the computed values across modules) occur in sequence without any overlap between them [14]. The crisis of this paradigm started with what is normally referred to as the "Wire Problem". As already mentioned, with the progress of semiconductor industry, the adoption of nanometers technology led to the integration of a growing number of transistors on the same chip. Throughout this trend, however, the global wires connecting different modules could not shrink in the same way as the local wires and logic gates did. As a direct consequence, most of the paths connecting different modules started to become critical paths due to their consistent resistive/capacitive delays. The great amount of exceptions to the synchronous paradigm coming with the technology scaling needed long stages of CAD flows for being fixed, without the guarantee of an optimal result. In this context, latency-insensitive design (LID) emerged as a candidate for a new methodology which relaxes the timing constraints imposed by the synchronous paradigm from the early design stages, when a correct estimation of the global wire connections is not available yet [14]. To be more specific, this approach states that if the modules composing an embedded system implementing the LID methodology are functionally correct, then the system can work independently of the inter-module channel latencies. Two key elements are necessary to reach this goal:

1. A **protocol** responsible for making the communication on inter-module channels independent from latencies.

2. A **shell** to implement an interface between the module and the latency-insensitive channels, flexible and adaptable to any instances of a component picked from a design space exploration set.

In the LID theory, two signals are said to be latency-equivalent it they present the same ordered sequence of valid data items, regardless of the exact timing of each event on each signal. In addition, if a system functionality is only dependent on the order of events of each signal, then it is said to be a patient system. These definitions are made more rigorous by defining two different types of events possibly taking place on such signals at each clock cycle: (1) an informative event, corresponding to a valid data item. (2) a stalling event, consisting in a packet that does not contain any valid information. In this way, the previous definition of latency-equivalent signals can be rephrased as follows: two signals are latency equivalent if they contain the same sequence of informative events, regardless of the stalling events interleaving at any point in such sequence [14]. The implementation

of the necessary logic for making a system patient is defined as *shell encapsulation* and it mainly consists in a wrapper that instantiates a buffering stage and some synchronization logics to interface the module core with the latency-insensitive communication channel. A condition to apply this logic without affecting the core functionality is that the core is stallable, i.e. it can be forced to wait for a new valid input for any number of clock cycles without losing its internal state. Once this key requirement is satisfied, the shell must be able to perform the following operations [14]:

- When a valid data item is retrieved at the input port of the core, the shell forwards it to the core that can update its state and emits a new output at the next clock cycle.

- Instead, when an invalid data reaches one of the input ports, the core is stalled until a new valid data item reaches that input port. At the same time, the valid data items reaching other ports of the core need to be temporary stored so to be used in the proper order when the core will be unstalled. This is the origin of the need of a buffering stage in the shell logic.

Amongst several implementations of the protocol and shell-encapsulation able to perform these operations, a particular focus on the one implemented in ESP is here proposed and summarized in Figure 1.7. The latency-insensitive protocol used for the communication channels, i.e. the different planes of the NoC, make use of two signals, in addition to the one conveying the data items: a void bit, that is used to distinguish a valid data item from a stalling one; an additional stop bit is also present to provide a very useful service commonly referred to as *backpressure*. For practical reasons, the buffering stage previously described has a limited storage capability. In other words, after a few clock cycles that a core has been stalled due to an invalid data on one of the input ports, the queues on the other ports will get full. In this scenario, it is necessary to prevent the uplink shells connecting to each of those ports from sending other valid data items that would overwrite the content of the queues. In other words, in certain cases, the downlink shell needs to exert backpressure on the uplink shell to keep the protocol correctly working. The easiest way to implement this capability is through the additional stop bit previously mentioned. When such signal is asserted, the corresponding core stops sending out valid token even if the input signals are valid, until the donwlink shell deasserts the stop bit. To sum up, a sequence of stalling events on an output channel is either due to a lack of alignment of valid token across the input ports or to backpressure from the downlink shell.

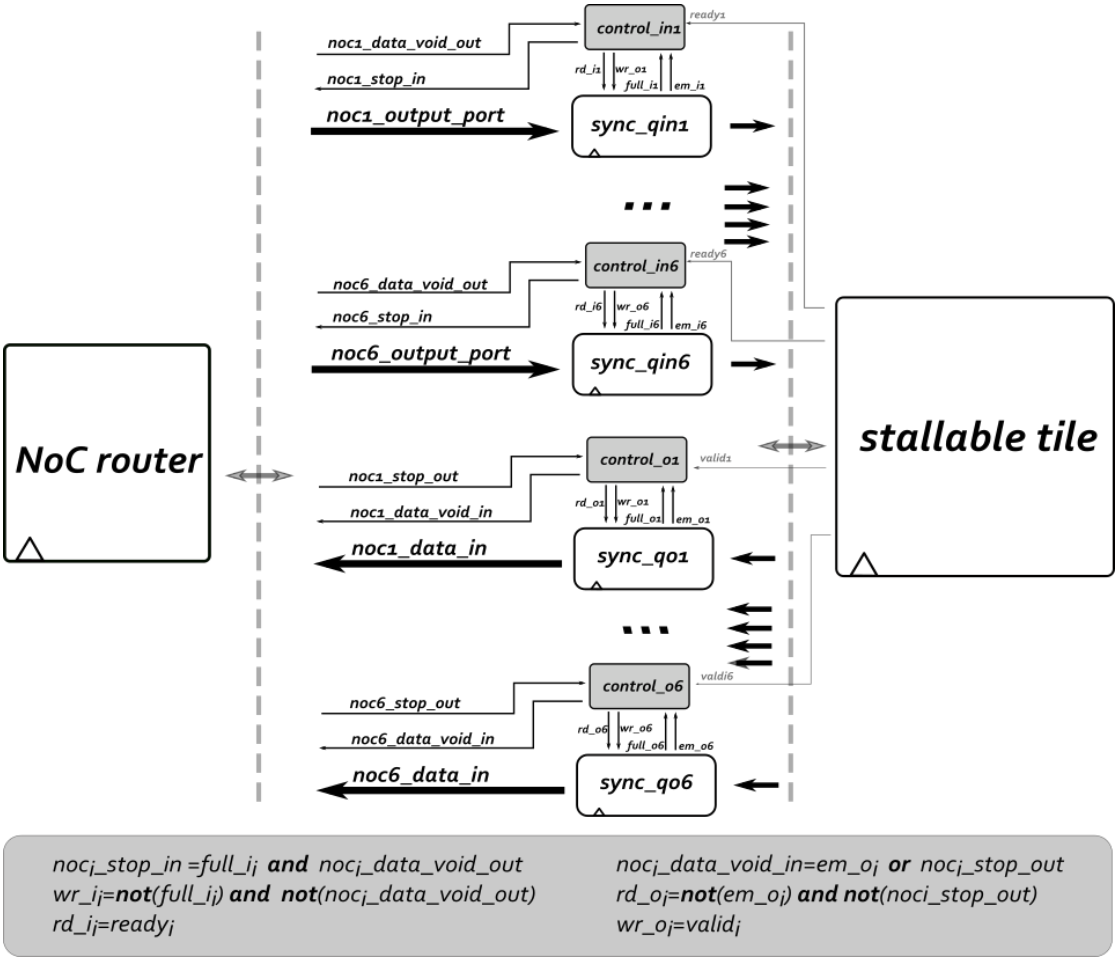


Figure 1.7: LID protocol and Shell encapsulation in ESP.

1.3.2 Tiles

The different types of socket are discussed in details in the following.

Accelerator tile

As opposed to many other open source RISC-V platforms, the ESP accelerators have their own dedicated tile and are thus given the same architectural relevance as the processor. These types of accelerators are commonly referred to as *loosely-coupled-accelerators*. Before proceeding with the description of the ESP accelerator socket, it is worth giving a brief overview of the existing accelerators classification. The main property driving the classification is the accelerator-processor coupling. This is a distinguishing feature that directly determines a set of important behavioral

trends for the whole SoC. Amongst those, the local storage transparency to the processor, the type of access to external memory and the granularity at which the data set of the target application is handled by the accelerator.

1. In processor-centric designs, the accelerators are **tightly-coupled functional unit** integrated in the processor pipeline as showed in Figure 1.8 . They are generally activated through a portion of control logic shared with the processor pipeline, after decoding the activation instruction obtained from a dedicated ISA extension. In addition, their location imposes several constraints on the internal design. A major limitation is given by the low available area, which imposes a limited amount of storage elements implemented as registers and prevents from the use of SRAM banks due to their limiting physical constraints.

2. With an increasing complexity of the task to perform, the accelerator design necessitates a relaxation of the coupling with the processor. A very common solution, adopted in several open source platforms, is given by accelerators integrated as **co-processors**. In this case, the specialized hardware is integrated as a separate entity still sharing many processor's resources and tightly connected to it through a dedicated interface. The interface is suitable for both instruction or data transfer as showed in Figure 1.9, depending on the specific implementation and the co-processor activity. When the co-processor activity is too high and causes resources contention, a direct interface to the bus system can be used leveraging either a DMA mechanism or the cache coherency of the system itself, as proposed with the Rocket Chip Generator [10].

3. An even more flexible solution is provided by **loosely coupled accelerators** as shown in Figure 1.10. Instantiated separately from the processor, they are not limited by any ISA or dedicated interface with the processor and can thus deal with more complex workload thanks to a higher degree of parallelism

Even if the implementations sticking to the first two models have demonstrated to gain significant performance improvements, when it comes to executing more complex tasks, the loosely-coupled model makes it possible for any external customized hardware to be completely decoupled from the processor and automatically integrated in the socket. This is the reason why ESP naturally leverages loosely coupled accelerators, also referred to as LCAs. In addition, the absence of particular constraints makes the accelerator tile of ESP suitable for hosting customized private local-memories implemented with multi-bank and multi-port structure, able to satisfy higher bandwidth requirements . This system-centric view, as opposed to a processor-centric view, distinguishes ESP from many other open source RISC-V

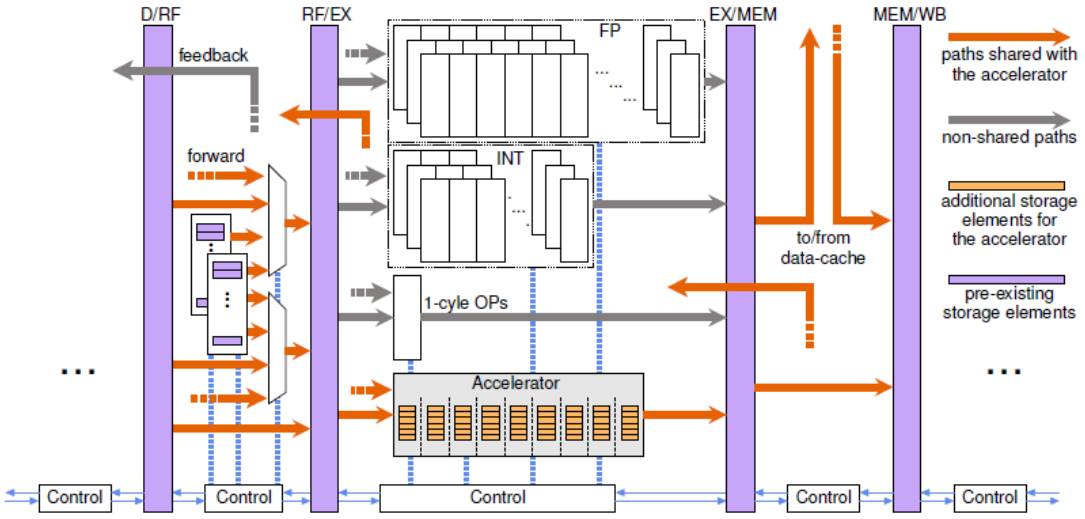


Figure 1.8: Tightly-coupled accelerator model [3].

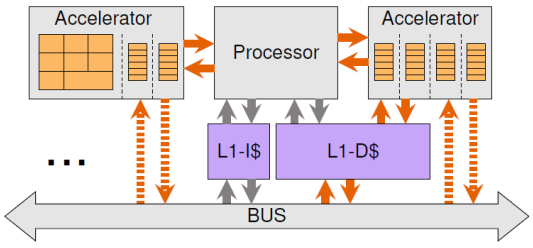


Figure 1.9: Co-processor model [3].

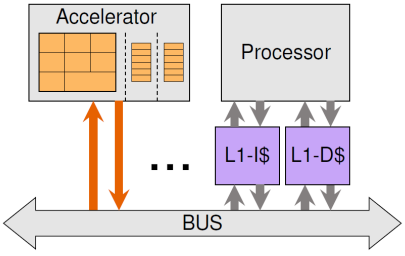


Figure 1.10: Loosely-coupled accelerator model [3].

platforms. Furthermore, the adoption of LCAs is a clear example of how ESP promotes IP reuse across different SoCs, making of flexibility and scalability two key concepts of SoC design.

The ESP accelerator socket provides several platform services, among which:

1. A run time selection of coherency models for the accelerator. The socket currently supports four types of coherence protocols:
   (a) **Fully coherent model**, using the existing memory hierarchy and an additional private L2 cache instantiated in the accelerator socket.
   (b) **Coherent-DMA model**, without a private cache but coherent with all the other caches of the system.

   (c) **LLC coherent model**, without a private cache and not coherent with the rest of the system.

   (d) **Non-coherent DMA model**, completely bypassing the cache hierarchy.

2. A DMA controller handling input read and output write requests with a latency-insensitive protocol.

3. A set of configuration registers used to activate different services without the processor intervention and an interrupt controller.

Processor tile

Each processor tile contains a core which can be selected among those available in the RTL processor library (the currently available processors are the RISC-V 64-bits Ariane processor from ETH Zurich and the SPARC 32-bit LEON3 processor from Gaisler, both coming with a private L1 cache and capable of booting Linux). The processor tile replicates the same type of seamless encapsulation of external IP thanks to the platform services delivered by the socket interface. The internal communication is handled by an AHB bus in LEON3 and an AXI bus in Ariane, while the proxies work as adapters to communicate with the memory and the interrupt controller in the IO tile, as well as to handle memory-mapped registers. The DMA engine is replaced by a unified write-back L2 cache while the configurations registers remain approximately the same as those discussed in the accelerator tile. In addition, three software layers complete the processor socket: an ESP Linux module, which allows the operating system to recognize all accelerators in the SoC, some ESP Linux device drivers to configure them, and an ESP user-level library which simplifies accelerator programming [5].

Memory tile

Memory tiles work as the access points to off-chip memory [4]: each memory tile contains one channel to the external primary memory. The number of memory tiles can be configured at design time and this typically varies from 1 to 4 depending on the size of NoC. Once this is configured, all the necessary hardware logic to support the partitioning of the addressable memory space is automatically generated, again transparently. Each memory tile contains one partition of the last-level cache (LLC) and of the corresponding directory, in order to support the MESI coherence protocol. Their size is also configurable, similarly to the L2. Additional logic implements the coherency protocol and all of the mechanism necessary to support coherent DMA transfer for the accelerators. An overview of the ESP sockets addressed so far is visible in Figure 1.11.

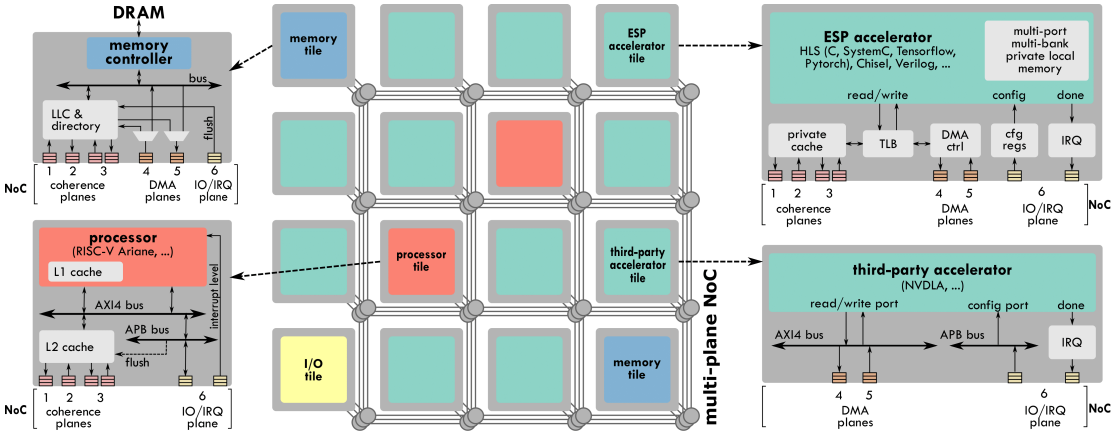


Figure 1.11: Overview of accelerator, processor and memory sockets, together with the related platform services [4].

IO tile

The IO tile contains the peripherals that guarantee off-chip communication, as well as a set of fundamental platform services. In this tile, the interrupt controller is instantiated and interfaced to the rest of the system with an interrupt proxy that sends and receives requests from the processors through the NoC plane 5. The Ethernet link is used for both logging in the ESP instance via Secure Shell (SSH) and accessing memory-mapped registers. A frame buffer is used for the video output from the DVI, while the UART provides a console interface and the timer enables periodic tasks [12].

Chapter 2

SoC Test trends

This chapter offers a brief overview of the state-of-the-art testing strategies adopted so far in post-manufacturing verification. In particular, both functional and structural tests are evaluated as possible candidates to address the increasing complexity of heterogeneous SoC testing, respectively in Sections 2.1 and 2.2 . With regard to the structural test, the main practices of Design For Testability (DFT) are reported and illustrated. Finally, the test approach adopted in this dissertation is introduced in Section 2.3.

The advance of nanometers technology turned into a dramatic decrease of the feature size of gates and wires integrated in complex integrated circuits (ICs), as well as in the increase of operating frequencies and clock speed accurately described by the Moore's law. The reduction of features size comes with the main drawback of increased probability of defects, incurring in the manufacturing steps, leading to a faulty chip [18]. A fraction of ICs is defective after the tape-out and a good test approach is needed to minimize the number of faulty chips delivered to the client. It is crucial to test ICs at different stages of the manufacturing flow. As a matter of fact, there is a general agreement with the *rule of ten*, stating that the cost of discovering a faulty chip increases by an order of magnitude at each successive level of integration, from die/package to board and system level as showed in Figure 2.1.

With the scaling of heterogeneous SoC complexity, semicondutor companies are looking for innovative and efficient ways to accelerate the time-to-market of complex heterogeneous systems while keeping their quality and costs competitive. Two main types of test have been adopted so far for digital systems and both of them follow a general approach: a set of test stimuli is applied to the inputs of the Circuit Under Test (CUT) and the output responses are analyzed, as showed in Figure 2.2. At this point, if the output responses match those expected for all the input stimuli, the circuit passes the test and is considered as fault-free. On the contrary, any failure in matching the expected outputs is sufficient to define the

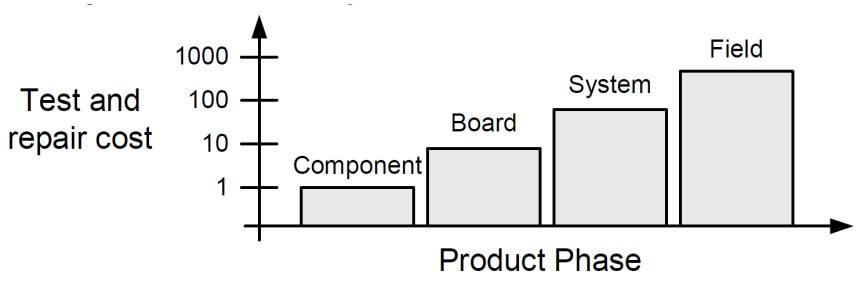


Figure 2.1: Rule of ten for ICs test costs [19].

circuit as faulty. The following two sections offer a brief overview of the tests and an evaluation of their performances when applied to complex SoC architectures [18].

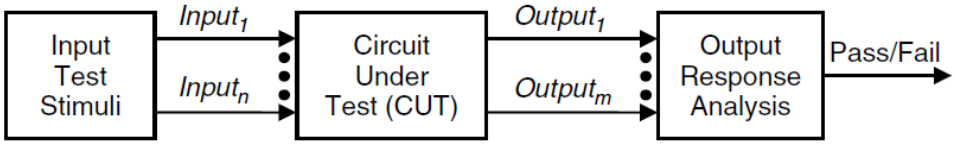


Figure 2.2: General Test approach [18].

2.1 Functional Test

A functional test provides verification that a design will perform its designated function. A set of test vectors is applied to the primary inputs of the device under test (DUT) and the response is checked for functional correctness. The aim of this procedure is to detect any mismatch between the implementation of the design and the original project specifications. The main drawback of this approach is given by the fact that it can become extremely time consuming under certain conditions. Once again, this is related to the fact that new generations of semiconductor process technology make it possible for designers to integrate more and more circuitry in smaller dies. When the system is tested, this translates in a larger number of test patterns drawn from larger vector sets. In other words, increased functional capability is the main responsible for a longer test development time. As a clear example, it is sufficient to consider a single but more complex gate to test, such as a 10-inputs AND gate. In this case, it is straightforward to conclude that a conventional functional test would involve a number of test vectors equal to all the possible input patterns, i.e. 2^{10}. This is because, if we strictly stick to the definition of functional test provided at the beginning of the section, the correct

functionality of the gate can only be proven by verifying that the gate itself is a 10-inputs AND gate. In order to do that, all the possible input patterns need to be applied to the gate inputs to compare the results with the expected ones [19].

As a consequence, conventional functional test techniques have gradually turned out to be unsuitable for verifying the complete functionality of heterogenous SoCs. Extending this concept to much more complicated system is thus clearly impossible for practical reasons and poses a challenge to the applicability of functional test to SoC modules.

2.2 Structural Test

Structural test aims at detecting physical and manufacturing defects responsible for any possible discrepancy between the implemented design and the manufactured chip. Even though this looks like a much more modest task with respect to the one targeted with functional test, the emergence of SoC architectures brought several challenges in finding a suitable test approach to do that. The effort in facing these challenges led to the development of a whole new set of test methodologies and strategies, briefly summarized later in this section. The development of structural testing techniques was dominated by the necessity to propose a model able to bridge the gap between the huge variety of possible physical defects affecting a chip and the impact of such defects at higher abstraction level. Indeed, with the increasing complexity of heterogeneous SoCs and the evolution of the processes involved in the manufacturing steps, the list of possible flaws incurring during fabrication has got significantly longer. Amongst those, it is worth mentioning processing defects (missing contact windows, parasitic transistors, oxide breakdowns, over/under etching, defective photoresist), material defects (cracks, particle contaminations, crystal imperfections, ion migration), time-dependent failures ,dielectric breakdown, electromigration) as well as packaging failures (contact degradation, leaks) [19]. Figure 2.3 displays just a subset of the possible process and material-related defects which can affect a gate integrated in the chip.
It was thus necessary to find a way to address the biggest possible subset of real fabrication defects with a fault model, technology-independent and capable of raising the abstraction level from the physical cause to the functional effect on the circuit operation. The most widely accepted model in this field is provided at RTL of description with the *single stack-at fault model*, which consists in representing single defective lines as permanently tied to the logic value 0 or 1. Unlike functional tests, structural tests shift the attention from the implemented function, which is assumed to be correct, to the circuit structure, i.e. the gates, the interconnects and the corresponding netlist. This type of test is strictly dependent on the fault models mentioned before: an algorithm is derived from the fault models so that

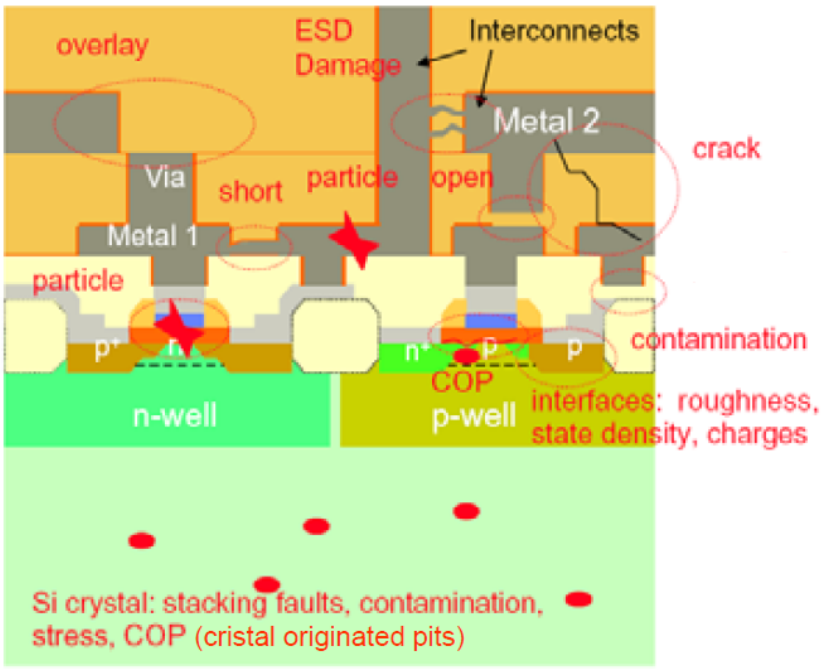


Figure 2.3: Relevant examples of chip defects [20].

a dedicated testing software can derive a minimal set of test patterns to force transitions on every net possibly affected by such faults. This step is commonly referred to as *Automatic Test Pattern Generation* (ATPG).
Nonetheless, a major issue to overcome when it comes to testing complex SoCs though, is the fact that most of the gates are deeply embedded and extremely difficult to reach and be tested from the primary inputs of the circuits. In other words, the circuit testability, given by the combination of controllability and observability of each fault possibly affecting the circuit nets, is intrinsically very low. The enhancement of testability can be achieved by adopting appropriate logic and architectural synthesis techniques that are discussed in the following subsection.

2.2.1 Design for testability

Design for testability (DFT) emerged as a new testing paradigm aiming at improving the circuit testability by including in the system dedicated hardware to perform the test. This is done while keeping the test development economically viable and the execution time and performance overhead low enough in order to guarantee a competitive product.

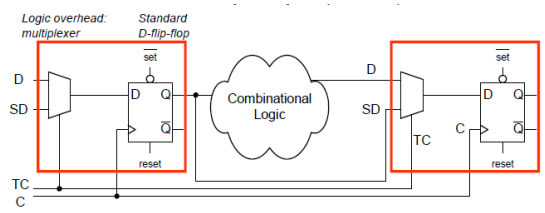
Some of the most used structural DFT methods include:

- **Scan-chain insertion**: a well known DFT practice capable of turning a sequential test in a purely combinational one. In the proposed approach, the Flip-Flops (FFs) of a design are turned into Scan Flip-Flops (SFFs) as showed in Figure 2.4a so that they can be connected together into a unique, large, distributed shift register. In this way, when the test mode is activated, a test vector can be shifted in the scan chain as showed in Figure 2.4b. Once the input stimula are applied to the combinational logic, the test results can be shifted to be compared with the expected ones. The efficiency of this approach led a group of electronic manufacturers to promote an industry standard, commonly known as IEEE Std 1149.1 Standard Test Access Port and Boundary-Scan Architecture. The architecture of the IEEE Std 1149.1 is showed in Figure 2.4c [21]. It consists of a set of registers used for instruction decoding, data storage, DFT bypass and other optional services, and a boundary-scan register composed of several boundary-scan cells. A total of four pins is used for the test I/O:

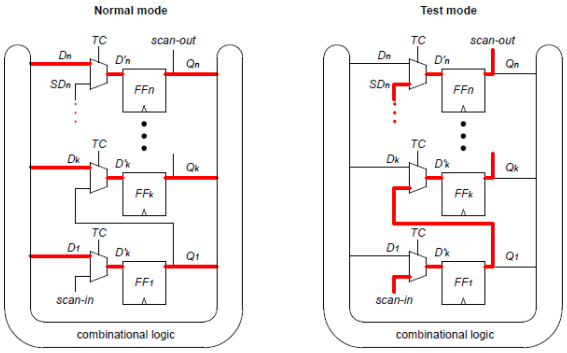
  1. A Test Data In (**TDI**) and a Test Data Out (**TDO**) pin for shifting and extracting the test vectors.
  2. A Test Mode Select (**TMS**) pin to drive the TAP controller state.
  3. A Test Clock (**TCLK**) pin for driving the test logic.

- **Built-In Self-Test (BIST)**: it consists in an elaboration of the idea of DFT which aims at improving the chip testability by substituting the expensive Automatic Test Equipment (ATE) with embedded hardware for test vectors generation and response analysis. The high-level implementation details are showed in Figure 2.5. In this way, the generation of test patterns can be performed on-chip and the controllability of the internal nets significantly improves. At the same time, the on-chip assessment of the test response improves the overall observability [19]. Originally started from simple circuitry surrounding memory blocks, the idea was extended to the logic with very satisfying results in terms of test coverage and costs. The main drawback coming with it is that the preexisting hardware is extensively modified [22].

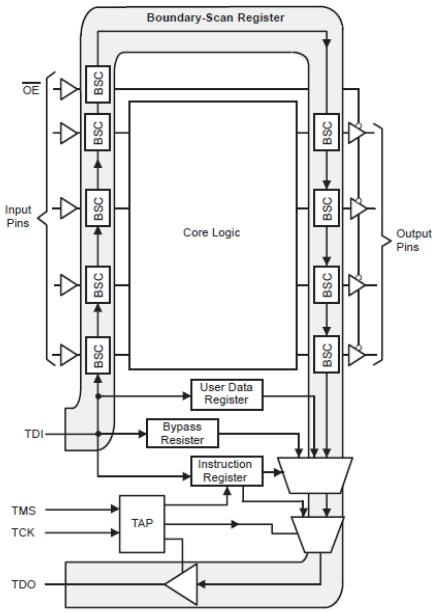
While the progress of structural DFT has given a major contribution in the development of SoC testing, many critical aspects are limiting their actual applicability in complex architectures. First of all, the fault model driving the test pattern generation is not complete since stuck-at fault model is not able to represent all the possible flaws in a system and can only target a subset of the real world defects. In particular, tiny bridges or necks in the interconnect, irregularities in vias and unpredicted coupling between signal lines create high resistive/capacitive delays which are not taken into account in fault models, but can still cause system failure.



(a) Scan flip-flops [19].



(b) Scan chain operating modes [19].



(c) Boundary-Scan Architecture [21].

Figure 2.4: Scan chain implementation.

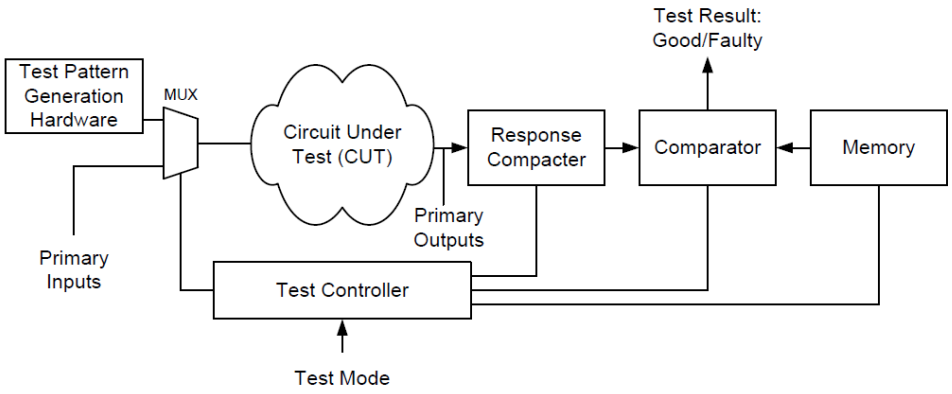


Figure 2.5: BIST implementation [19].

[23]. In addition, heterogeneous SoCs are often characterized by an extensive reuse of third-party IP. As for the case of ESP, different components coming from the open-source hardware community or from commercial vendors may come with their own test methodology. In particular an IP provider has most certainly already inserted DFT in the design, either with a scan or by implementing a BIST function.

The SoC test designer will thus need to employ a variety of different test strategies that need to be stitched together in a coherent system-level test methodology [24]. In the case of several cores provided with internal scan chains, for example, a significant engineering effort might be required to connect the scan chains to the external environment without changing the core interfaces. While this is a challenging task in normal conditions, it gets even harder when a third-party IP block is encrypted to protect the supplier's proprietary data and the netlist is not available, since the block can not be included in an overall chip synthesis strategy.

The latter point has been one of the main problems driving the evolution of DFT for SoC in the past decades. A viable solution which is often implemented consists in making use of IPs with the same test methodology, i.e. coming from the same ASICs vendor. While this approach represents an easy option to overcome the limitations mentioned, it strongly limits the SoC architect possibility to make use of heterogeneous third-party components coming from different sources.A more widely applicable solution consists in adopting a standard interface capable of adapting each core to a general SoC test methodology [23], independently of the core specific implementation. In other words, the test access to an internal scan-chain structure is implemented through a standard test wrapper that can be instantiated around any core, providing transparency during normal operating mode and isolation of the core in test mode [23]. The arrangement of the wrappers, the test access mechanisms and the possibility to run BIST in parallel in a specific SoC instance can then be decided by the SoC architect, according to the package pin and power consumption constraints. All these DFT strategies require a significant engineering effort to reach a good test coverage, and may end up to be more time-consuming than the design stage of the SoC itself. As a consequence, it is crucial that the synthesis of circuits with scan features is supported by an appropriate enhancement of the synthesis tools used [11].

2.3 Conclusions

To sum up, possible DFT solutions have been proposed so far to address the complexity of heterogeneous SoC testing by replacing time-consuming functional tests with a structural approach derived from fault models. The most important trends include Scan Chain Insertion and Built-in-Self-test (BIST). The former is a well known approach to improve the circuit testability, which undergoes a significant drop in test coverage when the system integrates several third-party IPs. The latter is a very popular approach integrating test pattern generators and response analyzers in the logic, with the drawback of extensive hardware modifications. My thesis is that, in parallel with the effort to build an efficient and flexible SoC structural test, taking advantage of the CAD tools to automate

the test features integration, the same amount of attention should be placed on alternative and faster approaches to verify the correct operation of SoC modules. In particular, this dissertation proposes a new flexible methodology to leverage DFT for single-core functional test. As already mentioned, a functional test focuses on verifying the function of a system rather than the faults affecting it. The approach proposed for each tile in ESP consists in forcing the module to execute a specific program where the test results can be checked instruction by instruction. This is done by making use of a Debug Unit embedded in the logic surrounding the tile, with I/O interface decoupled from the standard communication channels, i.e. the NoC. It is worth stressing that the proposed procedure works as both a pre- and post-silicon test. Indeed, rather than a simple test procedure, this approach aims at providing a flexible verification methodology which can be applied at any stage of the design flow and extended to other SoCs platforms. The following chapters will outline the working principles of such test interface, with a particular focus on how it implements the upload and injection of the test program instructions, as well as the check of the results obtained from the module under test.

Chapter 3

ESP Test approach

In this chapter, some of the main aspects driving the implementation of the testing logic for ESP are presented and discussed. In particular, the tasks of some fundamentals components critical for the system operation are discussed in more details in Section 3.1. Section 3.2 derives the high level specifications for a test interface capable of injecting program instructions in the tile under test and checking the response bypassing the NoC. Then, relevant considerations on the integration of the testing logic in the pre-existing hardware environment are proposed. Finally, the test flow and the test set-up are illustrated in detail in section 3.3, while the main types of applications used for testing the tiles are presented in Section 3.4.

3.1 Backend flow constraints

This section deals with some critical constraints imposed by the ASIC backend flow on the test methodology. To be more specific, the purpose of the discussion is to make the test strategy suitable for addressing a post-fabrication decrease of reliance on the functionality of some parts of the system.

3.1.1 NoC bypass

A few additional remarks concerning the importance of the communication infrastructure of the chip and its role in the operation of the system are functional to the testing discussion that will be presented in the following pages. The NoC used for ESP and briefly described in the previous sections, is composed of 6 channels connecting all the tiles of the SoC through multi-port look-ahead routers. Each channel plays a different role and delivers different services for ESP functionality, as reported in Section 1.3.1. In other words, each type of information exchange

among different tiles of the SoC is strongly relying on the correct operation of the NoC infrastructure. These include the whole set of fundamentals procedures for the system boot. Hence, it is of primary importance for the system operation, as for many others system-centric SoC, that the NoC is working correctly at each stage of the design flow, and that it is not affected by any fabrication defect after the tape-out. Given the many physical constraints imposed by the backend flow of the chip, however, the NoC turns out to have a significant footprint, since it needs to reach the tiles located at opposite corners of the SoC. As a direct consequence, the chances of having at least a defect in one of the many wires of the channels spreading around the chip increase accordingly, leading to potential threats for the whole system operations. Even considering the high reliability of the technology processes used for the ASIC implementation of ESP, the minimum risk of a defective NoC comes with a huge damage in terms of system functionality in case any fault. This is worth of being taken into account from the early design stages of the test logic. In particular, when a Test Access Mechanism (TAM) is proposed for the functional test strategy, it should be decoupled from the NoC so that it can perform correctly in case a communication channel of the NoC is not working. To conclude, a first backend-related directive for the testing logic is a NoC-independent access to the tile that needs to be tested.

3.1.2 Ethernet

Other components of the SoC playing a crucial role in the system boot should be taken into account. This is crucial to avoid a huge waste of engineering effort in designing a complex SoC with high quality platform services and transparent IP integration, that cannot be start the system boot due to fabrication defects. As previously mentioned, the NoC plays a key communication role since the early stages of the system boot, because the I/O tile can only communicate with the modules located in the rest of the SoC by using one dedicated channel of the NoC, namely Channel 5. Nevertheless, it is the I/O tile responsibility to implement all the main I/O channels and peripherals responsible for transmitting the relevant off-chip information, included the ones necessary for correctly starting the system, as well as supporting a suitable debug interface. In particular, there is a key communication link fundamental for both debugging and booting the system: the Ethernet Debug Communication link (EDCL). In the following, a brief description of this component and its purpose is presented to better understand how it affects the system functionality.

Ethernet Debug Communication link (EDCL)

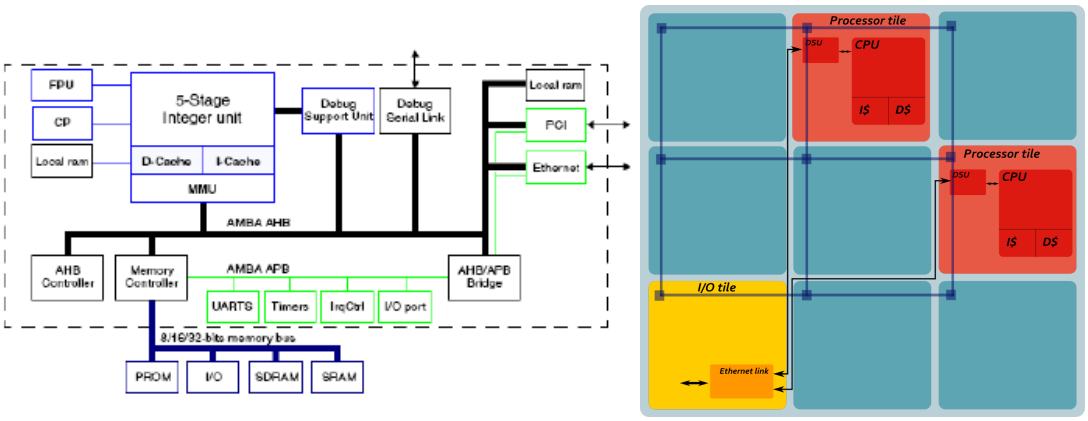
In first place, the Ethernet connection plays a crucial role for remote software debugging of applications running on embedded hardware. This type of debugging differs from what is normally referred to as *native debugging* [1] because it consists in running a debugger software in a machine that connects to the target hardware through a communication link [25]. Several types of connections are normally used, ranging from the slower JTAG connection to the faster EDCL (generally 2 or 3 orders of magnitude faster than JTAG). The communication between the host machine running the debugger software and the target hardware follows a predefined protocol and it is based on a few routines that allow the running application to be stopped from the debugger. In other words, a significant part of the debugger software runs on the external machine while just a few routines necessary for gathering information are directly handled by dedicated hardware on the SoC. In a typical computing system based on an open source library, the core comes with a Debug Support Unit (DSU) slave on the AHB bus, and thus accessible from any possible AHB master at any time. In the specific case of ESP, the messages between the processor DSU and the I/O tile travels on NoC plane 5, as reported in Section 1.3.1. In order to be able to control the debugging remotely, the DSU is indeed accessed from the external host through one of the communication links previously cited. A schematic of a typical LEON-3 system based on SPARC V8 architecture is showed in Figure 3.1a.
When it comes to a system-centric SoC as ESP, the configuration previously illustrated must adapt to the modular architecture which characterizes the platform. In other words, the DSU of the core and the communication link to the external host machine are located in different tiles. While the DSU resides in the CPU tile, the link is located inside the I/O tile, which is responsible for handling the most relevant peripherals and the off-chip communication, as illustrated in Figure 3.1. Having said that, a secondary task performed by the Ethernet link is more relevant to our discussion. In particular, the Ethernet is also used for the transfer of a set of crucial information for the SoC system boot. To be more specific, three additional fundamental tasks are carried out by the Ethernet in normal operating mode:

1. Load of the program to execute inside the bootrom [2].

2. Release the reset to boot the system.

[1] This expression is used to define the standard debugging approach, where the debugging software is developed in the same environment where it is used to debug [25].

[2] The Bootrom is a memory chip containing a few instructions which have to be executed by the processor to complete a successful boot.



(a) **(b)**

Figure 3.1: (a)Typical Debug communication used in LEON system based on SPARC Architecture [25]. (b) DSU default arrangement in ESP.

3. Configuration of the memory mapped registers instantiated in the chip, amongst which some crucial tile parameters.

All these tasks can be automatically executed via the Ethernet by sending out to each tile interface packets containing the corresponding relevant information. The parameters/registers configuration procedure, however, raise a further issue related to the synthesis flow. A multi-tile instance of ESP can include tiles that are functionally equivalent. This is done to provide higher parallelism for the applications running on top of the platform. Before approaching the project of developing a backend flow for chip design, the IDs of each tile, necessary for the tiles to be recognizable from the NoC routers, where hard-coded so to be constant at design time (generic ports at RTL). This clearly represents an issue now for the researchers involved in the physical flow, since several instances of the same core/accelerator, are actually differing for the tile IDs and thus necessitate separated synthesis. In order to prevent the burden of different synthesis for the same tiles, the following approach was adopted at RTL: all the generic ports for the tile IDs were turned into simple ports in all the entities of the different tiles, so that they could be configurable at run time. At this point, the next big challenge consists in finding a smart way to configure these registers before the system boot. The most natural way to deal with this issue is to rely on the same method previously adopted for configuring all the other types of memory-mapped registers: use the Debug Unit to configure the registers. Furthermore, the configuration of all the other tile parameters could be furtherly automated by making each of them strictly dependent on the corresponding tile ID. In such a way, once the tile ID is configured using this procedure, all the relevant settings are transparently generated. However, a critical

issue incurs in the initialization sequence that the tile IO launches at time zero before the system boot. Some of the secondary parameters automatically derived from the tile ID contain relevant information for the packets routing around the SoC. In particular, the coordinates of the tile are generally used from the network router to calculate the next hop for the flit being currently transferred. Since the tile coordinates are not known at time zero, the initialization sequence generated from the tile IO must be tailored to send out packets proceeding with a predetermined order. For example, a smart way to handle this particular condition, only occurring during the system boot, may consist in proceeding in the configuration by adjacent tiles. In other words, the routers only need to route the packet to the adjacent tile. In this way, respecting a predetermined order, all the configuration flits can be gradually spread out to the whole chip and reach each tile, configuring the tile ID and the corresponding dependent parameters. Another drawback coming with the transformation of the generic constants into configurable signals is the following: one of the parameters which are made configurable by setting the tile ID is the bus index. This parameter contains the information on the bus ports which is accessed by the specific tile. If no specific modification is provided, the following scenarios can be noticed: if the bus index is a constant, all the ports which are not connected to any peripheral are automatically erased during logic synthesis. On the other hand, if the bus index is a register, the logic synthesis will automatically infer a huge multiplexer with a significant area overhead, which must be manually avoided.

Ethernet bypass

The use of the Ethernet for the parameters configuration is viable and guarantees that the whole physical flow is run just once for each type of accelerator or CPU. Nevertheless, an alternative option to rely on in case of any additional issues affecting the Ethernet operation must be provided. Each of the tasks normally performed using the Ethernet interface basically consists in a set of instructions to be written in the corresponding tile before the system boot. In normal conditions, this information originates from the I/O tile, that received the information from the outside through the Ethernet link, and reaches the the tile interface through the channel 5 of the NoC. If the link is not working, an other type of access to the target tiles is necessary. A valuable strategy consists in using the test pins of the Debug Unit implemented to inject the configuration packets in the tile. To be more specific, before the system bring up, the test mode is selected so that the configuration packets can be injected in the tiles through the dedicated testing logic. Once the packets have been injected and the corresponding task has been terminated, the normal mode can be restored. At this point, the reset is released and the system waits for the first request coming from the CPU tile.

The purpose of all of this is to have an alternative way to boot the system in the worst case scenario of an Ethernet link not properly working. In this way, even if the SSH log cannot be enabled and the other tasks undergo a significant increase of latency when performed through the debug unit, the system functionality is preserved.

3.1.3 Conclusions and additional remarks

To sum up, it is necessary to develop a Debug Unit to be integrated in each tile of the SoC so that the post tape-out scenarios illustrated in this section can be faced preserving the system operation. In particular:

1. If the NoC is not properly working, the testing logic allows to perform all the preliminary operations (registers configuration, reset release, etc..) as well as to send the other instructions of the target application.

2. If the Ethernet link is not working, the following approach is used: the test mode is enabled before the system boot and the memory-mapped registers containing the tiles parameters are configured by injecting the configuration flits in each tile through the Debug Unit. At this point, after injecting the instruction that releases the reset, the test mode can be disabled so that the tile interface gets reconnected to the NoC planes and the system can run in normal mode as usual.

In order to guarantee the correct switch from test-operating mode to normal mode, a modification in the design turns out to be fundamental for the following reason. During a CPU tile test, there might be a critical passage weakening the robustness of the test in case of an Ethernet link not working. More specifically, when all the operations that the Debug Unit is performing in place of the Ethernet link are completed, one final instruction is sent to the CPU through the Debug Unit in order to correctly boot the system: the instruction for releasing the reset. Once this instruction reaches the CPU and is executed, the CPU is free to start issuing requests. In our case, the first expected request is a read request for the first address of the bootrom. In order to guarantee that the request reaches the correct destination (i.e. the first address of the bootrom located in the IO tile), it is necessary to make sure that such request can be forwarded on the NoC of the SoC. Nevertheless, if the switch of the logic from test operating mode to normal operating mode is not fast enough, the first CPU request may reach the interface before the switch has actually completed, and get consequently lost, thus blocking the execution of the rest of the program. A smart solution to avoid a situation like this has been proposed as follows: an internal counter can be instantiated in such a way that the Finite State Machine (FSM) regulating the switch between

the different states of resets takes into account the need of a certain delay during the switch from one state to the other.

3.2 Test interface specifications

This paragraph proposes high-level specifications for the Debug Unit, addressing the challenges derived from the constraints of the back-end flow discussed in Section 3.1.

The conclusions derived in the previous paragraphs makes it clear that an implementation of a ESP prototype chip requires a test unit implemented as additional platform service, capable of substituting the NoC and the Ethernet link. In this way, in the worst case scenario of one of those components not correctly working, the test interface will have the additional capability of manually boot the system by injecting the configuration flits one by one or verify the functionality of all the tiles of the chip by simply emulating the NoC behavior.

As mentioned in Section 2.3, the best approach to replace the NoC in test mode is to include an additional customized unit directly inside each tile of the SoC, with its own dedicated pins to communicate with the external environment. The Debug unit must be able to perform the following tasks:

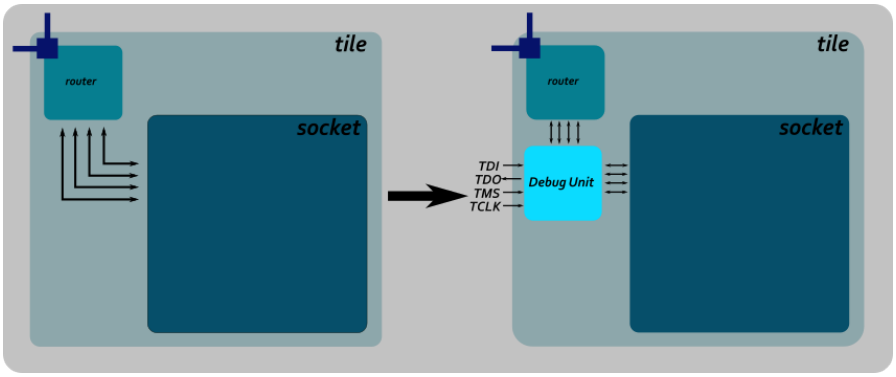
1. Accept instructions injected serially through one of the dedicated I/O pin.
2. Verify the type of instruction and act accordingly on the tile interface to execute the test.
3. Use another pin to extract the result of the test, i.e. the messages reaching the test interface as a result of the flits previously injected.

Before proceeding with the illustration of the test specification , it is worth mentioning that the instructions injected from the test access point will be directly taken from a trace obtained in a simulation of the SoC running in normal operating mode. This will guarantee that the testing logic will act exactly as the NoC, directing the packets to the tiles with the same order as the NoC does, and expecting the same response from the corresponding tiles. The steps previously cited will be illustrated and discussed in details in Section 3.3, where the proposed test flow is presented, but are here mentioned to give an overall idea of the principle laying behind the Debug Unit operation.

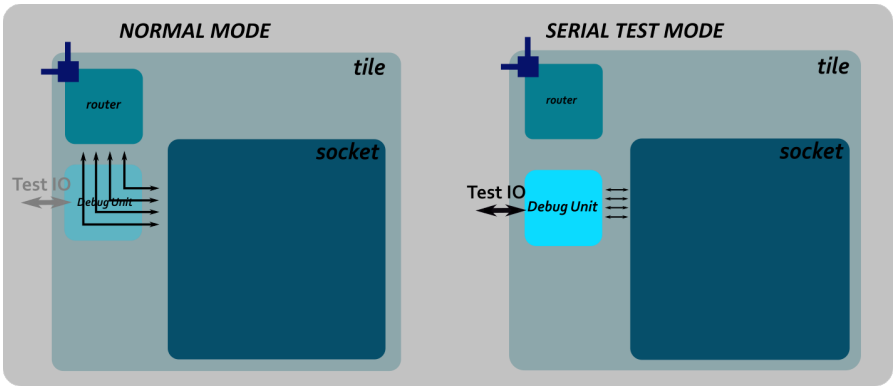
3.2.1 Insertion point and Debug Unit interface

A key requirement of the integration process is that the additional testing logic does not affect the whole system while working in normal operating mode. In order

to do that, the ports connecting the tile interface with the NoC planes should be multiplexed in such a way to switch the connections according to the current value of the signal used for setting the operating mode. Hence, the Debug unit must be placed at the same RTL where the standard connection of the internal hardware socket with the router ports takes place. With respect to the previous arrangement in the modules hierarchy, the direct connection between the hardware socket of the tile and the router is now interrupted by the Debug unit laying in between, as showed in the upper part of Figure 3.2. The internal logic works in such a way that the original connection is restored in normal operation, while the NoC gets completely disconnected in test mode, as summarised in the lower section of Figure 3.2.



(a)



(b)

Figure 3.2: (a) Debug Unit placement in the preexisting Design. (b) Tile's operating modes.

Listing 3.1 and Figure 3.3 give more detailed information on the proposed test interface by providing the code of the VHDL entity implemented for the Debug Unit and the schematic view, respectively. As discussed in Section 1.3.1, the NoC

supports a latency-insensitive communication protocol, which makes use of two additional single-bit signals to handle stalling events and backpressure, respectively a void-bit and a stop-bit. Those signals need to be passed to the testing logic in the same way as the one simply carrying data/instruction does. Let us discuss in detail why this is crucial for each of them:

1. **nocX_stop_in**: stop bit conveying backpressure from the tile to the NoC. In normal operation, the value imposed from the tile (i.e. testX_stop_in in Figure 3.3) is simply assigned to it. In test mode, on the contrary, it should be set to one as soon as a valid message coming from the NoC on that port is traced. This is a convenient way to prevent that any message gets lost travelling on the NoC while the tile is in test mode. For example, when the test logic is used to replace the Ethernet link, the standard connection should be restored after the system boot. In this case, if some messages were flying on the NoC during the test, a hold on the input ports of the tile under test is necessary during testing not to loose them.

2. **nocX_data_void_out**: void bit validating the messages in the direction NoC/Tile. This must be passed to the Debug Unit so that it is directly forwarded to the tile in normal mode. On the other hand, when the test mode is activated, it is not used by the testing logic.

3. **nocX_stop_out**: stop bit conveying backpressure from the NoC to the tile under test. This must be passed to the Debug Unit so that it is directly forwarded to the tile in normal mode. On the other hand, when the test mode is activated, it is not used by the testing logic.

4. **nocX_data_void_in**: void bit validating the messages in the direction Tile/NoC. In normal mode, the value imposed by the tile (i.e. testX_data_void_in) is assigned to it. On the other hand, when the test mode is activated, it is automatically set to 1 by the testing logic. Once again, the reason can be traced back to one of the guidelines driving the design, namely the necessity to completely decouple the operating modes. In fact, in this case, setting the void bit to 1 is necessary in order to avoid that any wrong flits is forwarded from the testing logic to the NoC during the test.

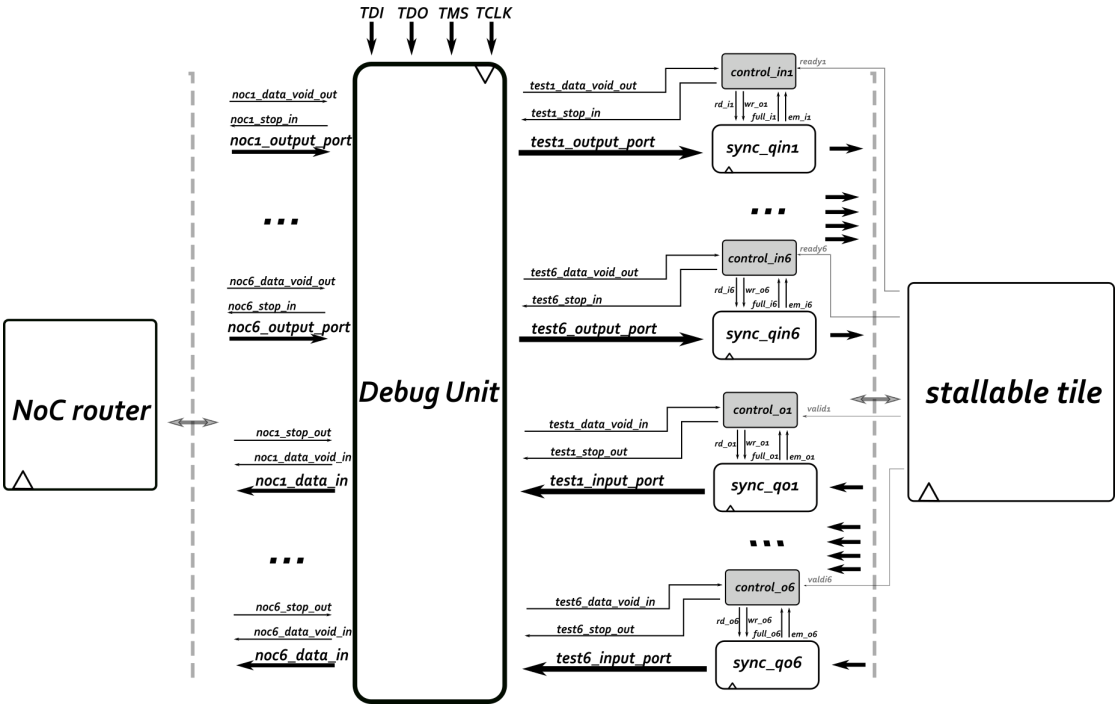


Figure 3.3: Debug Unit interface.

Listing 3.1: Debug Unit entity.

```
entity jtag_test is
  generic (
    test_if_en : integer range 0 to 1 := 0);
  port (
    rst     : in std_ulogic;
    refclk  : in std_ulogic;

    tdi                  : in  std_ulogic;
    tdo                  : out std_ulogic;
    tms                  : in  std_ulogic;
    tclk                 : in  std_ulogic;
    -- NoC out to Test/TileQ in
    noc1_output_port     : in  noc_flit_type;
    noc1_data_void_out   : in  std_ulogic;
    noc1_stop_in         : out std_ulogic;
    noc2_output_port     : in  noc_flit_type;
    noc2_data_void_out   : in  std_ulogic;
    noc2_stop_in         : out std_ulogic;
    noc3_output_port     : in  noc_flit_type;
    noc3_data_void_out   : in  std_ulogic;
    noc3_stop_in         : out std_ulogic;
    noc4_output_port     : in  noc_flit_type;
```

```
23      noc4_data_void_out   : in  std_ulogic;
24      noc4_stop_in         : out std_ulogic;
25      noc5_output_port     : in  misc_noc_flit_type;
26      noc5_data_void_out   : in  std_ulogic;
27      noc5_stop_in         : out std_ulogic;
28      noc6_output_port     : in  noc_flit_type;
29      noc6_data_void_out   : in  std_ulogic;
30      noc6_stop_in         : out std_ulogic;
31      -- Test/NoC out to TileQ in
32      test1_output_port    : out noc_flit_type;
33      test1_data_void_out  : out std_ulogic;
34      test1_stop_in        : in  std_ulogic;
35      test2_output_port    : out noc_flit_type;
36      test2_data_void_out  : out std_ulogic;
37      test2_stop_in        : in  std_ulogic;
38      test3_output_port    : out noc_flit_type;
39      test3_data_void_out  : out std_ulogic;
40      test3_stop_in        : in  std_ulogic;
41      test4_output_port    : out noc_flit_type;
42      test4_data_void_out  : out std_ulogic;
43      test4_stop_in        : in  std_ulogic;
44      test5_output_port    : out misc_noc_flit_type;
45      test5_data_void_out  : out std_ulogic;
46      test5_stop_in        : in  std_ulogic;
47      test6_output_port    : out noc_flit_type;
48      test6_data_void_out  : out std_ulogic;
49      test6_stop_in        : in  std_ulogic;
50      -- TileQ out to Test/NoC in
51      test1_input_port     : in  noc_flit_type;
52      test1_data_void_in   : in  std_ulogic;
53      test1_stop_out       : out std_ulogic;
54      test2_input_port     : in  noc_flit_type;
55      test2_data_void_in   : in  std_ulogic;
56      test2_stop_out       : out std_ulogic;
57      test3_input_port     : in  noc_flit_type;
58      test3_data_void_in   : in  std_ulogic;
59      test3_stop_out       : out std_ulogic;
60      test4_input_port     : in  noc_flit_type;
61      test4_data_void_in   : in  std_ulogic;
62      test4_stop_out       : out std_ulogic;
63      test5_input_port     : in  misc_noc_flit_type;
64      test5_data_void_in   : in  std_ulogic;
65      test5_stop_out       : out std_ulogic;
66      test6_input_port     : in  noc_flit_type;
67      test6_data_void_in   : in  std_ulogic;
68      test6_stop_out       : out std_ulogic;
69      -- Test/TileQ out to NoC in
70      noc1_input_port      : out noc_flit_type;
71      noc1_data_void_in    : out std_ulogic;
```

```
72      noc1_stop_out         : in  std_ulogic;
73      noc2_input_port       : out noc_flit_type;
74      noc2_data_void_in     : out std_ulogic;
75      noc2_stop_out         : in  std_ulogic;
76      noc3_input_port       : out noc_flit_type;
77      noc3_data_void_in     : out std_ulogic;
78      noc3_stop_out         : in  std_ulogic;
79      noc4_input_port       : out noc_flit_type;
80      noc4_data_void_in     : out std_ulogic;
81      noc4_stop_out         : in  std_ulogic;
82      noc5_input_port       : out misc_noc_flit_type;
83      noc5_data_void_in     : out std_ulogic;
84      noc5_stop_out         : in  std_ulogic;
85      noc6_input_port       : out noc_flit_type;
86      noc6_data_void_in     : out std_ulogic;
87      noc6_stop_out         : in  std_ulogic);
88
89  end;
```

3.2.2 Pin count constraints and pin sharing

The considerations drawn above provide sufficient material to define the required amount of pins dedicated to the debug unit off-chip communication, as well as the exact location where such unit should be instantiated in the hierarchy of files contained in the RTL description of the SoC. In particular, for each tile:

- A **TDI** pin for injecting test flits in the tile.
- A **TDO** pin for extracting test response from the tile.
- A **TMS** pin for selecting the operating mode of the system.
- A **TCLK** pin for distributing the test clock across the logic. This is because, given the functional type of test targeted, all the operations involved in it can be performed at a much lower clock frequency with respect to the one characterizing the system operation.

However, in order to reduce the number of pins dedicated to testing and meet the upper limit on the total pin counts imposed by the backend constraints, a strategy of pin sharing is necessary. The original SoC test plane imposes that the Debug Unit is instantiated in each tile of ESP, so that any of them can be tested in a ASIC implementation. With the a 36 tiles-instance of ESP, for example, 144 pins would indeed be required for the SoC testing (4 for each tile). Sharing the TCLK and TMS pins among all the tiles is an immediate and feasible way to decrease the resource usage dedicated to testing. In this way, just the TDI and TDO pins would

be replicated for all the tiles, while a single pair of pins for TMS and TCLK would be used for the whole chip. This would bring the total number of test pins from 144 to 74. The reason why the TCLK pin can be shared is intrinsically related to the type of testing executed. As mentioned, the clock frequency of a functional test is order of magnitudes lower than the one of the system operation. As a consequence, timing closure can be easily reached on a wider area and the same clock can thus be distributed on the whole chip footprint. Let us now consider the consequences of using a single TMS for all the tiles. In this case, when a specific tile is selected for being tested, the global TMS value is set to 1 from outside. As a consequence, in all the tiles of the SoC, the tile interface connection switch from the NoC ports to the Debug Unit ports. Nevertheless, it can be proved that the tile targeted will be the only one to be activated and successively involved in the test. The main reason is that the TDI pins are distributed one for each tile, i.e. the information for the test will be only passed to the pin accessing the tile of interest. In addition, different tiles have different tile IDs and thus different configuration flits. In other words, no other computational unit of the chip would be able to respond to the initial configuration phase and the test would stop immediately. An example of the pin sharing technique proposed applied to a 9-tiles instance of ESP is illustrated in figure 3.4.

Note that this section is providing a general approach for post-silicon unit test of tile-based SoC. Specific choices can then be made to modify the proposed methodology and adapt the pin usage strategy to the needs of a specific SoC implementation. In order to do that with a specific ASIC target, more information on the test infrastructure controlling the Debug Unit are necessary, together with a general overview of the chip I/O partition.

Figure 3.5 shows a detailed overview of a possible I/O plan for a 16-tiles ASIC instance of ESP. Each connection, labeled with the corresponding FPGA mezzanine card (FMC) [3], is listed here in detail:

- 4 FPGA-based memory links (FMC1 and FMC2) are used to connect each memory tile of the chip to the off-chip DDR4 [4]. More specifically, each memory tile is connected via memory link to the FPGA, which in turns interfaces with 4 DDR modules (FMC6, FMC7, FMC8 and FMC9) and an Ethernet interface to store external data in memory (FMC 10). For each memory link, a 64-bit bidirectional bus converting LLC and DMA request into memory accesses

[3]FPGA Mezzanine Card (FMC) is a standard that defines I/O modules with connection to ant device with I/O capability, FPGAs included [26].

[4]Double Data Rate 4 (DDR4) is a type of synchronous dynamic random-access memory characterized by a high bandwidth interface largely used for modern SoCs memory system [27]

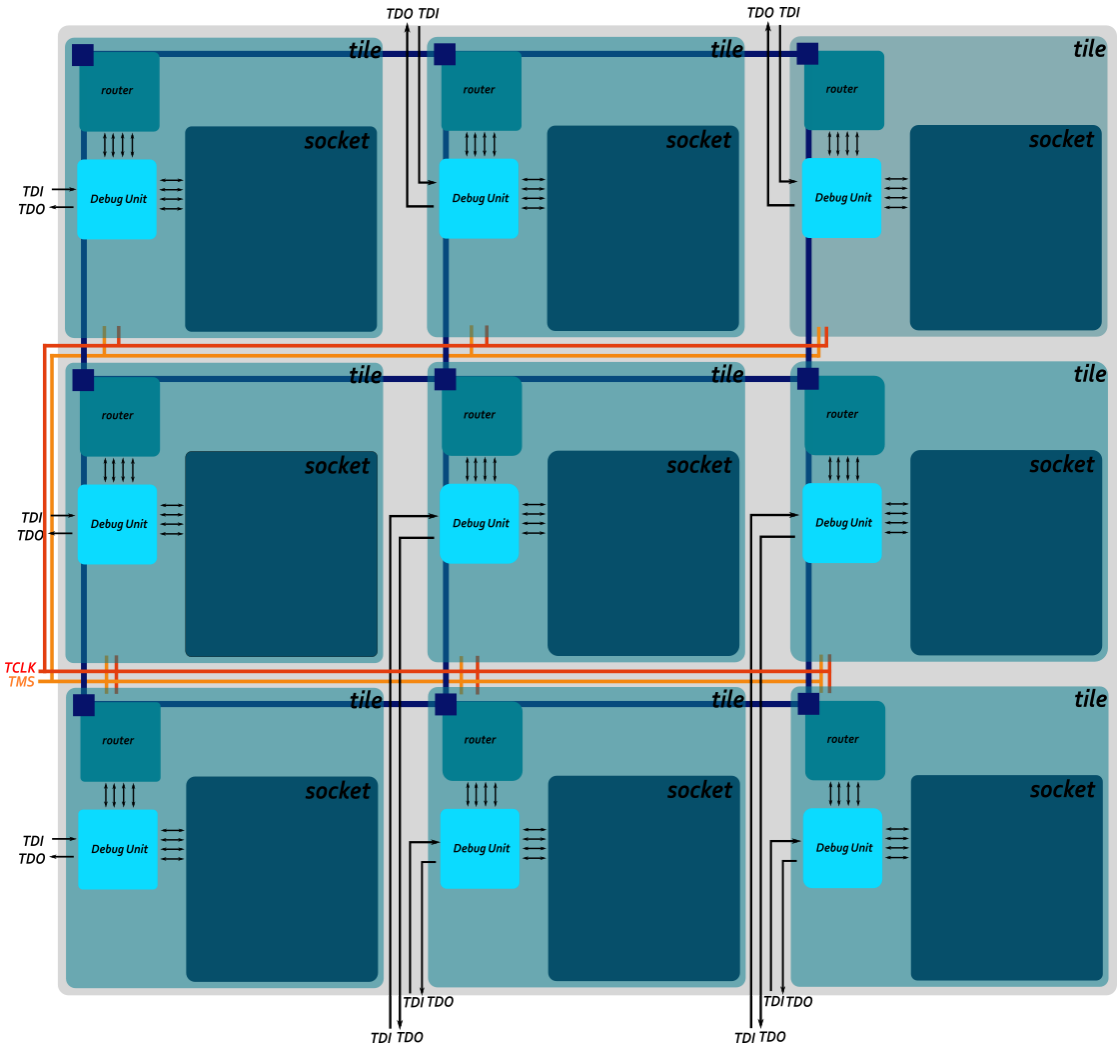


Figure 3.4: Pin sharing approach for ESP Debug Unit.

is required. The FPGA link is source-synchronous [5] and characterized by a credit-based flow control [6].

[5]This means that the FPGA takes care of generating the clock for synchronising the data to be sent to the chip, and the chip sends back the data together with the same clock.

[6]Credit-based flow control works with the receiver sending out credits to the sender in order to indicate the availability of buffering elements and the sender waiting for credits to transmit the data [28]

- Ethernet link (FMC 5) with a single PHY [7] and a double MAC layer [8]. The first one is the ESP debug link and the second one is the Ethernet peripheral.
- UART link via USB.
- JTAG link for the Debug Unit control (FMC4).
- Reset and Clock pins (FMC 3). The clock is internally generated by Digital Controlled Oscillators (DCOs), but a back up external clock is used for some tiles in case those will fail. In addition, a monitor pad to measure some clock frequencies is used.

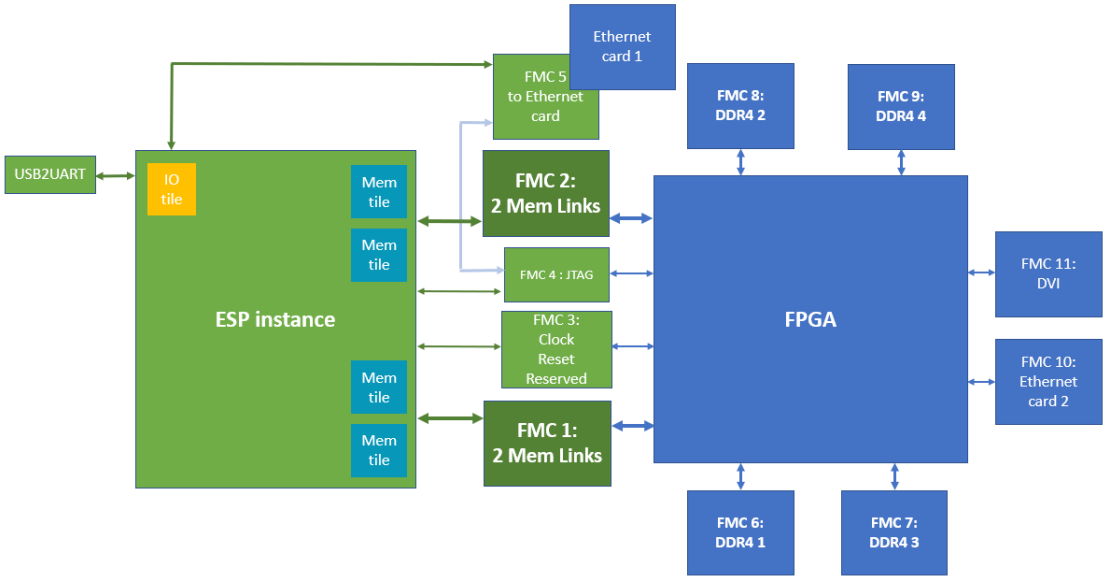


Figure 3.5: General Chip-FPGA test plan.

Each of the communication links mentioned involves a certain amount of pins. Given the shortage of IO resources, it may be necessary to limit the Debug Unit use to a meaningful subset of the SoC tile to reduce the corresponding test pins. Table 3.1 contains a detailed summary of a possible pin partition strategy for a 16-tiles ESP instance. Adopting the sharing approach illustrated above, the total amount of pins used to test a selected subset of 11 tiles via Debug Unit is 24.

[7] PHY is the physical layer implementing the communication functions in a network interface controller [29].

[8] The medium access control (MAC) is the layer that controls the hardware responsible for interaction with the wired, optical or wireless transmission medium, which propagates the signals in a communication link [30]

IO function	Pin
FPGA Link	280
IO tile peripherals	24
JTAG interface	24
Clock	12
Total	**340**

Table 3.1: I/O pins partition for a 16-tiles ESP ASIC instance.

3.3 Test flow

This section aims at providing an overview of all the steps involved in the execution of a test with the proposed Debug Unit.

As with most of the functional test commonly executed, the functionality of each unit under test is verified as follows: a certain stimulus is provided at the input ports of the units under test, and the resulting output is compared with the expected result. When it comes to a more complex system as the one implemented by a tile of ESP, this concept remains untouched. In addition, since the tile shell respects the latency-insensitive paradigm, it is not required to obtain the same exact output sequence cycle by cycle. On the contrary, for the test to be successful, it is sufficient to obtain the expected results in the given order. The most effective way to verify that is by storing both the flits to inject and the expected output results in a set of dedicated registers. Those registers should contain a 64 bit-instruction and some additional bits for opcodes and info on the destination plane (further information on these bits will be given in the following). For a given tile, in different moments of the test, such registers can contain the input test flits to direct towards a certain input port of the tile, as well as the expected output result to compare with the test output sent from the tile output ports. Let us keep in mind that the amount of registers instantiated in the testing logic plays a crucial role in the final test performance. Considering the CPU tile, a brief reminder from Section 1.3.1 on the types of messages handled by the different NoC planes is reported in table 3.2. For what concerns the incoming messages, plane 2 handles incoming flits supporting the system coherency protocols while plane 3 receive memory response.Plane 4 and plane 6 respectively receive coherent and non-coherent DMA requests from accelerators. Plane 5 takes care of incoming interrupts requests as well as all the information from IO peripherals (UART, Debug link, DVSI). This include configuration flits for writing the registers, as well as all the instruction of the application currently tested, stored in the bootrom inside the IO tile. For what concerns the outcoming messages, plane 1 forwards CPU requests to memory while plane 3 send coherency protocol messages. Plane 4 and plane 6 respectively

NoC plane	Input packets	Output packets
1	/	CPU requests to memory
2	Coherency support	/
3	Memory response	Coherence support
4	Coherent DMA requests	Non-coherent DMA response
5	IRQ requests / I/O info	IRQ acknowledge/ I/O info
6	Non-coherent DMA requests	Coherent DMA response

Table 3.2: NoC planes purposes from the CPU tile perspective.

send respectively non-coherent and coherent DMA response to accelerators. Plane 5 takes care of conveying IRQ acknowledge and other types of messages to I/O peripherals. Having said that, two possible scenarios are possible when looking at the content of the test registers during a CPU test:

1. The instruction is a flit that should be injected in the CPU. This type of flit can either be a configuration flit, normally used at the beginning of the system boot and at run time to configure accelerators registers. Alternatively, it can be an accelerator request, or a memory response. Each of this type of messages has its own dedicated plane for being fetched around the SoC. In this scenario, the message should be written to the tile.

2. Alternatively, the instruction stored in the register can be an expected output from the CPU, which is not supposed to be sent to the CPU. Such instruction is the next expected request that the CPU is supposed to issue, as a result of the previous operations performed.

Since the main objective of the test is to completely replace the NoC, two possible approaches can be adopted, but just one of them is feasible and can be put into practice.

In the first approach, the instructions to be injected from the test interface are manually written by the test engineer, either at assembly level or machine code. These instructions can mainly include simple arithmetic operations to be performed on values stored in the CPU registers, or load/store instructions involving the communication with memory. With the testing logic discussed so far, this would translate in the following operations: an instruction is written to the CPU from the NoC plane 5, which is the one from where the instructions of the program to test, normally stored in the bootrom, are transferred in normal operation mode.After that, depending on the type of instruction injected, the CPU request issued on a certain plane is checked in order to verify that it is the same as the expected one. For example, let us consider the case where we want to test the CPU capability to issue a read request toward the memory. A load instruction is fed into the

port of the CPU tile normally connected to the NoC plane 5, in order to emulate an instruction of the program coming from the bootrom. Being the instruction a load, the CPU will issue a read request to a certain memory address to read such location and load its content in one of the the internal registers. The request would travel through NoC plane 1 in normal mode. If the test mode is activated, however, the content of such request becomes the tile's output that needs to be checked. Unfortunately, performing this check requires the designer to manually obtain the expected CPU requests for every instruction sent to the CPU. This method clearly shows a lack of automatization which makes it unsuitable for testing the functionality of a CPU tile talking to 6 different channels of the NoC, even if simple test applications are used.

The second approach is based on the same sequence of steps, but takes advantage of the possibility to extract all the information necessary for the check from a simulation of the system running in normal mode. In particular, a simulation trace can be generated in such a way to obtain a collection of all the transactions executing at the tile interface on the NoC planes while the application is running. As opposed to the previous scenario, the test engineer is thus already in possession of all the expected requests which the CPU is expected to issue at any time of the test execution. In the previous example, right after the packet containing the load instruction has been injected and written to the CPU from plane 5, the testbench proceeds by injecting the following instruction of the trace, which is, in this case, the expected read request that the CPU will issue on plane 1 to access the memory location indicated by the instruction.

The latter method can be reproduced for every type of program: the simulation trace collected in normal operating mode contains all the necessary information to perform the test. In such a way, the testing logic offers the possibility to test even more complex applications, for which manually obtaining the expected results would be unrealistic. This is the reason why the second approach was the one chosen to perform the test from the early stages of the design. An overview of the flow involved in the second approach is illustrated in figure 3.7. The first step illustrated in the upper part of the figure consists in setting up a simulation in normal operating mode, i.e. TMS=0. Before running the simulation, depending on the CAD tool used for it, a list containing a specific set of signals is created. All the signals at the tile interface, are included in the list, together with their dedicated void and stop bit. In the proposed schematic, those signals are the one included in the list cut inside the tile under test, i.e. those on the left-bottom corner of the SoC instance. In this way, all the information necessary for for emulating the NoC behavior are collected in a readable and reusable *.lst* file. Once the simulation of the target application is completed, the simulation trace is passed to an editing script that turns it into a *.txt* format readable from a testbench. In particular, for each time-stamp in the simulation trace, the presence of a valid signal (i.e. void

bit to 0) among those added to the trace is checked, and, if present, the 64-bits instruction is written on the stimulus file, together with some additional bits to encode the type of instruction as well as the plane address. In particular, the format of each line of a stimulus file is showed in figure 3.6. The first 64-bits wide slot stores the instruction directly retrieved from the simulation list. This is followed by a 6-bits address where the source/destination plane involved is indicated in one-hot encoding. A void bit validates the incoming flits and an opcode indicates whether we are dealing with an instruction to write to the tile or to expect from it. Finally, a dirty bit is constantly set to 1 and used to inform the testing logic on the status of the flits shift-in process. A total of 73 bits is thus required to store all the relevant data/control information of a test instruction. In this way, the stimulus files contain all the necessary information to conduct the test: the instruction flits that must be written to different input ports of the CPU tiles, and the expected CPU requests.

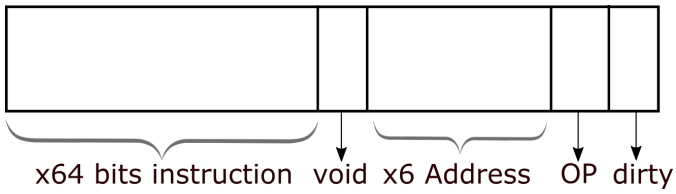


Figure 3.6: Input flits format.

As showed in the lower part of figure 3.7, the second step consists in the actual test of the tile: the TMS is set to 1 in order to disconnect the NoC. All the access points to the tile's internal hardware socket are managed by the test interface and the test can proceed.

3.4 Test programs

In this section, the different types of programs used for testing the SoC tiles are presented.

When an ESP test is performed, the simulation targets cross-compile a default C application for the target processor. Users can edit the application at will, as long as the baremetal cross-compiler can generate the target binary. As an output, the compilation produces memory files for the simulation to be stored in the SoC bootrom and target binaries if the user wants to perform FPGA emulation . After that, the simulator starts either in the terminal or with the GUI depending on the target chosen [31].The default program for testing a CPU is a baremetal application printing the string "Hello from ESP!" to the UART peripheral of the SoC. In particular, the sequence of operations to be performed in this test, in addition to

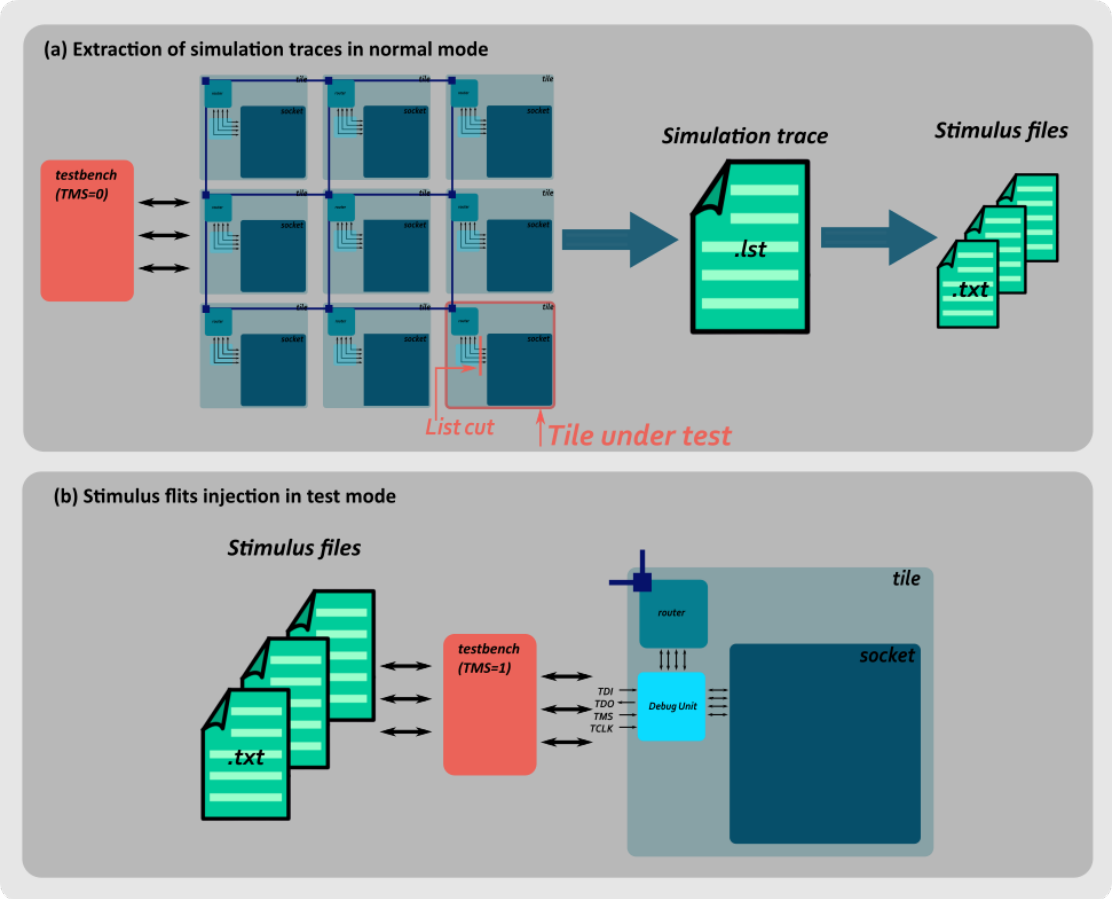


Figure 3.7: Test flow.

the load of the device tree bulb [9], is the following:

1. Set of the stack pointer.
2. Configuration of the registers.
3. Release of the reset.
4. Initialization of the UART peripheral.
5. Print of the message "Hello from ESP" from the UART.

[9]The device tree bulb (DTB), also referred to as a flat device tree, device tree binary, or simply device tree, consists in a database containing the information on the hardware components on a given board and represents the default mechanism to pass low-level hardware information from the bootloader to the kernel [32]

The Assembly-level code used for performing such operations is showed in the program *Startup.s*, in the left column of Figure 3.8. Note that the management of the UART peripheral is performed in a second program called from the main, written in a mix of C and light Assembly, as showed in figure 3.9.

```
#include <smp.h>
#define DRAM_BASE 0x80000000

  .section .text.init
  .option norvc
  .globl _prog_start
_prog_start:
  smp_pause(s1, s2)
  li sp, 0x9ff00000
  call main
  smp_resume(s1, s2)

  csrr a0, mhartid
  la a1, _dtb
  li s1, DRAM_BASE
  jr s1

  .section .dtb
  .globl _dtb
  .align 4, 0
_dtb:
  .incbin "ariane.dtb"
```

Figure 3.8: Startup.S : default test application stored in the bootrom.

```
#include "uart.h"

int main()
{
        init_uart();

        // jump to the address
      __asm__ volatile(
              "li s0, 0x80000000;"
              "la a1, _dtb;"
              "jr s0");

        while (1)
        {
                // do nothing
        }
}
void handle_trap(void)
{

}
```

Figure 3.9: main.c: program printing "Hello from ESP!" to the UART peripheral.

Even though the application does not require the execution of any complex operation, it certainly involves the cross compilation of several functions written in high level languages (in this case, in C language). As a consequence, it is much harder for the user to keep track of the instruction currently tested while using the test interface to inject them flit by flit. While this is not an issue in the case of a correct operation of the system, it gets critical when the origin of a failed check from the test interface has to be found. This is the reason why I decided to build a simpler test to target as a first attempt to verify the tile functionality. In case of success of the simpler application, as will be discussed in the simulations section, the more complex test will be executed.

The customized test which I propose is performed in baremetal, i.e without the support of an operating system, and consists in a few instructions performing simple arithmetical operations on the internal CPU registers, as well as load/and store request issues. The test proposed is showed in Assembly in Listing 3.2. In

particular it consists in:

1. Loading immediate values into the temporary registers t0 and t1.
2. Storing the content of their sum into the register a1.
3. Store the content of the a1 register at the memory address 0x80000004.
4. Load the value back to the t0 register.
5. Increment the value stored in the register a0 by 5 units. and store it back. to the memory at the address 0x80000008.

Listing 3.2: Default test application stored in the bootrom.

```
#include "uart.h"
# start sequence of the bootloader
#
#
#include <smp.h>
#define DRAM_BASE 0x80000000

  .section .text.init
  .option norvc
  .globl _prog_start
_prog_start:
  li sp, 0x9ff00000
  li t0,3
  li t1,4
  add a1,t0,t1
  li a5, DRAM_BASE
  sw a1,4(a5)
  lw t0,4(a5)
  addi a0,a0,5
  sw a0,8(a5)
```

Chapter 4

Debug Unit Design

In this chapter, a comprehensive description of the steps involved in the RTL implementation of the Debug Unit is proposed. The test interface performances are evaluated for both the design versions implemented in Sections 4.1 and 4.2. For each of the two designs, a detailed description of the FSM controlling the the Debug Unit operation and an illustration of the datapath is provided. Then, the simulations results are analysed in detail. The functional features differing between the versions are highlighted, with a particular focus on the limitations of the first one and on the performance improvement delivered by the second one. Finally, the main conclusions related to the test methodology are outlined.As illustrated in the previous sections, different types of tile make different use of the NoC planes. A clear example is given by the fact that the processor tile does not make use of plane 4 and 6 for handling direct memory access, while the accelerator and memory tiles do. Even if implemented test unit is flexible for different types of tile, a detailed description of the internal logic will require practical examples showing the specific type of messages travelling through the interface ports. For the sake of simplicity, the processor tile will be taken as a reference tile for showing such examples, keeping in mind that the same explanation can be equally applied to other types of tiles.

Before proceeding, let us restate what are the constraints that need to be respected by the testing logic of the Debug Unit.
First of all, the logic must be able to temporary store instructions serially injected from the test FPGA through the dedicated serial TDI interface. Afterwords, it must verify the type of instruction stored and act accordingly on the interface of the tile under test in order to emulate the NoC behavior. In addition, the logic must be synchronized with a clock with its own TCLK pin. Hence, when the system operates in test mode, two different clock domains run on the same tile. The first domain covers all the logic performing the test at a lower frequency, while

the second clock domain covers the rest of the tile. As a direct consequence, it is necessary to instantiate dual-clock FIFO queues for each NoC plane in order to synchronize the instruction to inject or extract from the tile with the destination clock domain. Finally, a strict requirement to satisfy is that the logic must not affect the system when it runs in normal mode. In other words, the ports connecting the tile to the NoC must me multiplexed in order to enable a switch according to the selected operating mode. When the testing mode is enabled, i.e. TMS is set to 1, the tile interface is directly communicating with the testing logic and the NoC is disconnected from the tile. The stop bit directed to the NoC is set to 1 to stop messages coming from the NoC. At the same time, the void bit directed to the NoC is automatically set to 1 in order to prevent the transmission of wrong flits to the NoC. On the other hand, in normal mode, the standard connection is restored and the system can normally run with the tile directly communicating with the NoC.

These requirements worked as guidelines throughout all the stages of the design, driving the modifications applied to improve the test performance and flexibility with respect to different types of application test.

4.1 First version - single register

At this point, let us illustrate the first implemented design.
In the first version, a single SIPO-register is used for handling operations on the 6 planes. Hence, the register's content varies in message type and destination plane according to the specific stage of the test.

4.1.1 FSM

The FSM implementation is showed in Figure 4.1 and discussed in detail below.
At the beginning of the test, the FSM is in the IDLE state, waiting for the user to activate the test mode when necessary. Once the test mode is activated by setting the TMS value to 1, the first flit of the stimulus file is read from the testbench and serially injected throughout the TDI pin during the INJECT state. After that, it is temporary stored in a serial-in-parallel-out (SIPO) register (from now on,referred to as "SIPO-register") waiting for the testing logic to verify the instruction type and proceed accordingly. At this point the void bit is read by the logic. If the void bit is set to 1, it indicates that the flit is invalid because it is the last one in the stimulus file. In other words, the test is terminated and the FSM shifts to the state DONE. On the contrary, if the void bit is 0, the opcode determining the type of instruction is read by the logic: if the instruction is an expected CPU request, (OP=0), the FSM shifts to the state WAIT. Conversely, if the instruction is a flit

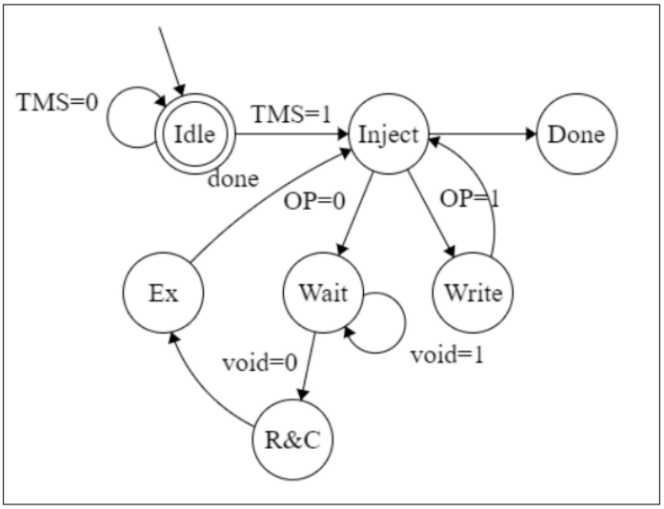


Figure 4.1: FSM of Design 1.

to be written to the CPU (OP=1), the system shifts to the WRITE state. Let us have a closer look at those scenarios:

- In the former case, the system waits for a message coming from one of the output ports of the tile. In particular, a valid message reaching the tile interface can be detected by looking at the value of the associated void bit. The latency insensitive design principle regulating the NoC operation imposes that the true value of the void bit can only be observed by removing the stop on the same signal. At RTL level, this is to say that the valid bit has a combinatorial dependence on the stop bit given from the external, as indicated in the signals description of Figure 1.7. Hence, the void bit and the corresponding valid signal can only be observed if the external environment is not applying backpressure on that port, i.e. if the input stop bit is set to 0. In normal operation, the NoC router takes care of removing the input stop bit to accept incoming flits. The test interface substitutes the NoC in test mode in carrying out this task. The flit stored in the SIPO-register also contains the plane address. As a consequence, the testing logic can act accordingly by removing the input stop bit on the addressed port. For example, if the next expected request issued by the CPU is a write-request travelling on NoC plane 1, such information is encoded in the flits stored in the SIPO-register. The test logic can thus reset the stop signal (noc1_stop_out in this case) on NoC plane 1 in order to retrieve the incoming flit on that channel. A selective removal of the stop bt is crucial in order to avoid loosing others flits reaching the tile interface in the same moment. By keeping all the others stops, the content of the FIFO queues remains stored until the respective queue is actually read.The

output valid signal is temporary stored in a parallel-in-serial-out register (from now on referred to as "PISO-register") and the system shifts to the READ AND CHECK state. At this point, the content of the SIPO-register (i.e. expected CPU request) and the one of the PISO-register (i.e. actual CPU request) can be compared. In case of a match, the TDO value is set high for a cycle to inform the testbench of the successful instruction test. After that, the FSM shifts to the EXTRACT state (Ex), where the content of the PISO-register is serially shifted out through the TDO pin and written to an output trace. Once the extraction is completed, the system goes back to the INJECT state, waiting for the next injection.

- On the other hand, if the injected instruction is a message to inject toward one of the input ports of the tile, it is only necessary to direct it to the specific port, while setting the corresponding void bit to 0 in order for the message to be accepted by the core as a valid flit. Once again, the specific port to access is indicated by the address encoded in the stimulus file flits, and thus stored in the SIPO-register. Once the write operation has been performed, the system goes back to the INJECT state waiting for the next instruction to be injected from the stimulus file.

4.1.2 Datapath

The Datapath of the first design version is showed in Figure 4.2 and commented below.
As mentioned, the registers storing the instructions play a crucial role in the Debug Unit operation. In fact, most of the control and status signals of the logic depend on such registers. This is because the control unit operation is mainly driven by the Address and Opcode information encoded in the stimulus files rows.
Let us first look at the logic of the path going from the test interface to the tile. A demultiplexer is used for directing the content of the SIPO-register to the correct port when the instruction that it stores is a flit to write. Depending on the address stored in the SIPO-register, the write-flit is directed to the appropriate dual-clock FIFO queue connected to the buffering stage.
A dual clock FIFO queue, showed in green, is instantiated for each NoC plane to synchronize the signals with the tile's clock frequency domain. Being the synchronization of the flits across two different clock domains the only purpose here, its depth is set to 2, which is the minimum accepted by the control logic regulating the read and write operations.
Successively, a 2-to-1 multiplexer is used for each NoC plane to switch between different operating modes. As previously mentioned, the selector is a signal directly obtained from the synchronized value of the input pin TMS. In normal mode (TMS=0), the standard connection of the tile ports with the NoC channels is

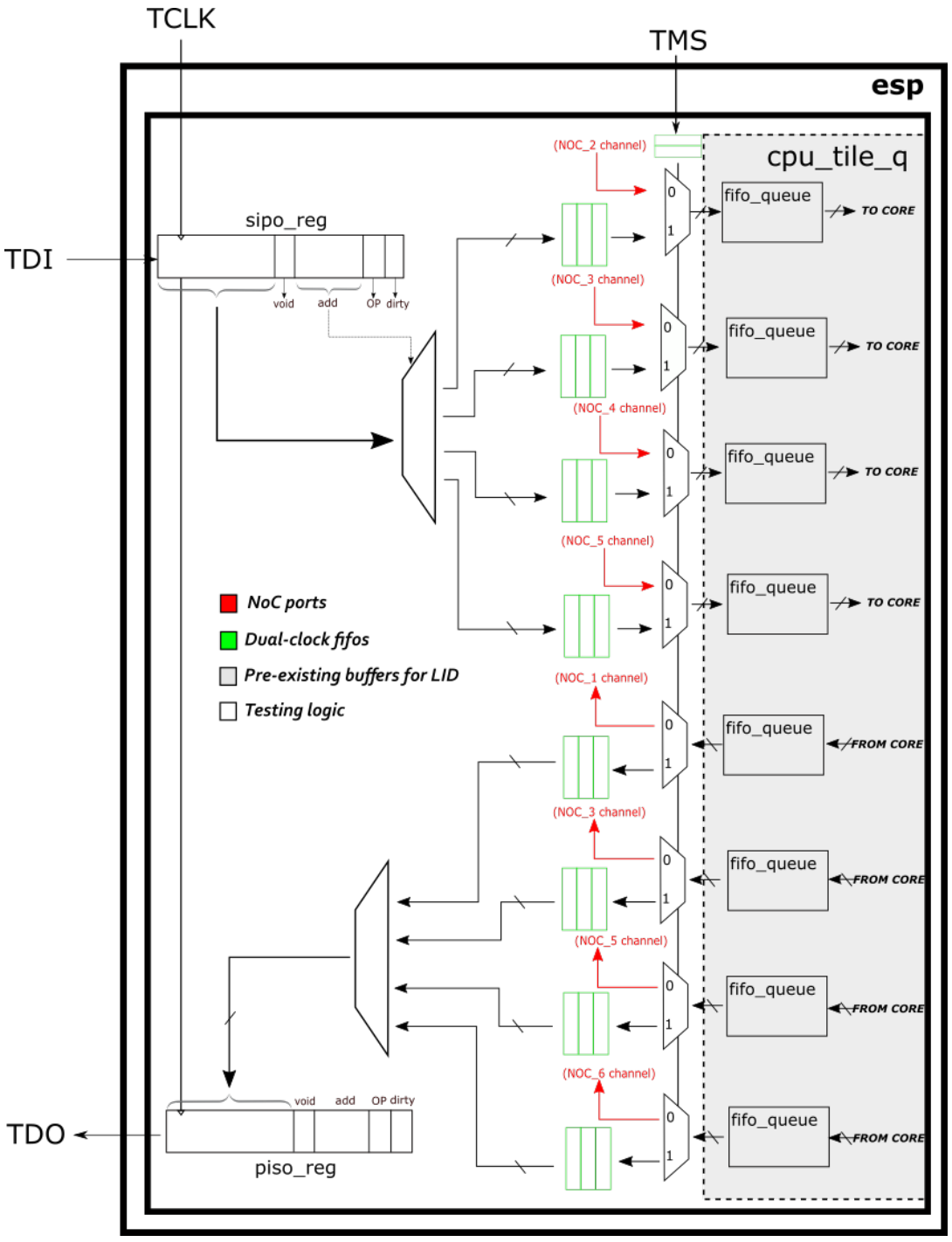


Figure 4.2: Datapath of Design 1.

restored. In test mode (TMS=1), the tile ports are connected to the output ports of the dual clock FIFO queues previously mentioned, so to enable the communication with the test interface.
Once the flits are injected through the multiplexers, the standard path to reach the CPU core is followed starting from the FIFO queues instantiated in the *cpu_tile_q* component. This works as an intermediate buffering stage functional to the latency-insensitive paradigm governing the system.

For what concerns the opposite direction, from the tile to the test interface, the same type of logic is used in a specular way. The 2-to-1 multiplexers are replaced by 2-to-1 demultiplexers but the connection's purpose remains untouched. In test mode, the tile output coming from the *cpu_tile_q* component is directed to the test interface, while the output ports are disconnected from the NoC, with the corresponding void bit of the channel set to 1 to prevent any wrong message from being transferred to the NoC. Conversely, in normal mode, the standard connection with the NoC is restored. The dual clock FIFO queues are now read and written in a reversed order: the input port is now on the tile side to accept incoming CPU requests, which are then sent to the test interface through the output port on the opposite side. Finally, a multiplexer is used to decide from which NoC plane to extract the next expected request, and stored into the PISO-register.

4.1.3 Simulation

In the following paragraph, a critical analysis of the testing logic simulation results is presented. For both types of tests mentioned in Section 3.4, an evaluation of the test interface performance is provided. In particular, for what concerns the second type of test, a particular emphasis is placed on the critical features of the testing logic that led me to revise the design for a more efficient testing procedure.

First of all, the simpler test application, amongst those mentioned in Section 3.4, was executed to verify the functionality of the CPU tile. In this case, after completing the debug of the code, the test interface succeeded in perfectly emulating the NoC. A straightforward method to verify graphically this fact with the simulation waveform window of Modelsim is to check that the TDO value is set high during the READ AND CHECK state. This indicates that the content of the corresponding registers is the same.
In order to give a complete view of the test-interface behavior during the execution of simple operations, the most relevant waveforms observed during simulation are reported below for the following two instructions extracted from the test program of Section 3.4:

1. **sw a1,4(a5)**: after receiving the packet containing this instruction, labeled with the letter A in Figure 4.3, the CPU is supposed to issue a write requests

Time	test5_output_port	test1_input_port	test3_input_port
1	A	–	–
2	–	B	–
3	C	–	–
4	–	D	–
5	–	–	E

Table 4.1: Simulation trace for the *load* and *store* instructions.

directed to memory through channel 1, labeled as B.

2. **lw t0,4(a5)**: after receiving the packet containing this instruction, labeled with C in Figure 4.4, the CPU is supposed to issue a read requests directed to memory through channel 1, labeled with D. After that, the content of the memory location addressed, labeled with E, reaches the tile through channel 3.

The content of the stimulus file obtained from the execution of these two instructions in normal mode is summarized, merging the packets into a single flit for simplicity, in Table 4.1.

It was more convenient, for practical reasons, to reproduce what was observed form the simulation in a dedicated waveform drawer instead of directly reporting sanpshots of Modelsim simulation window. The main reason is that each of the two tested instructions corresponds to three flits stored in the stimulus file (header, body and tail, as imposed by the NoC routing protocol). The representation provided in Figures 4.3 and 4.4 simplifies the simulation results by merging three flits content into one, as if they were contained in a single flit. The waveforms here reported, however, are strictly reflecting the results of the simulation. Several attempts performed with programs of similar complexity confirmed that the proposed implementation of the test interface is suitable for supporting baremetal tests belonging to this category.

Even if the purpose of the test interface is to prove the correct functionality of the tile by testing simple operations, it was worth, at this point of the work, performing the default baremetal test illustrated in Section 3.4. The increase of the test complexity is mainly due to the routines involved in the cross-compilation of programs written in high level languages. These leads to several burst transactions on the internal buses and on the NoC channels, i.e. long sequences of memory accesses to adjacent locations. While this is not a problem in normal mode, it is considered to be one of the main reasons for the failure of the test with the second

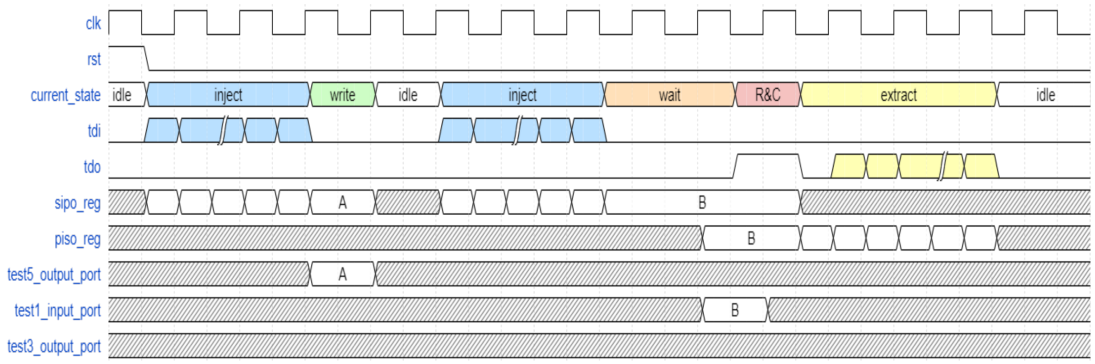


Figure 4.3: Execution of *store* instruction in test operating mode.

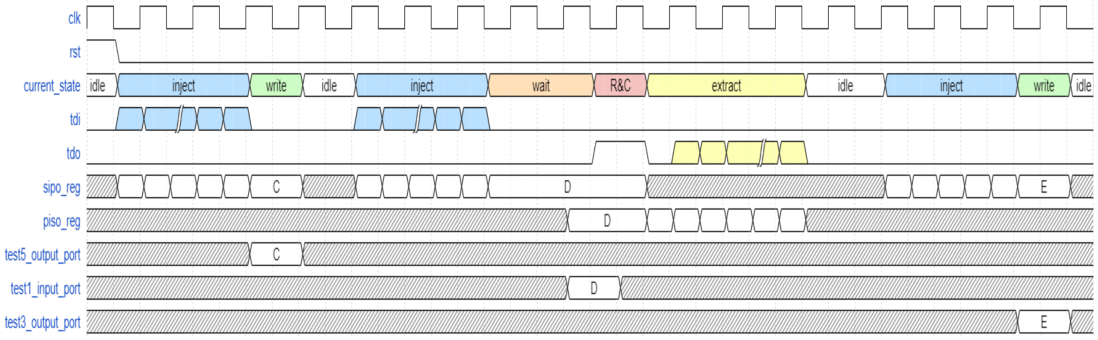


Figure 4.4: Execution of *load* instruction in test operating mode.

application. As a matter of fact, when the test mode was enabled, the simulation started correctly, performing the expected sequence of injection, read and checks and extractions. At a certain point of the test, however, it can be noticed that the test gets stuck when an expected write request from NoC Plane 1 does not pass the read and check. In other words, the expected value was stored in the SIPO-register did not match the flit forwarded by the CPU tile and stored in the PISO-register. Having a closer look at the content of the latter one, it was possible to determine that it was not coming from the expected channel, i.e. Channel 1, but from Channel 5 instead.

Backpressure effect

In order to explain such a behavior, further investigations on the internal components of the CPU tile helped to detect the origin of the test failure. To explain that, a closer look at the internal organization of the tile is provided in the following. In addition, a graphical illustration of the example discussed below is provided in

Figures 4.5,4.6 and 4.7.

The internal communication between different components of the CPU tile is handled through a bus system following the AMBA-AXI protocol, whose working principle is illustrated in more detail in Appendix B. The messages coming from the Ariane core are redirected toward the dedicated plane of the NoC based on their type, which is encoded in the header flit. Before reaching the tile interface, however, an intermediate buffering stage to temporary store the incoming flits is necessary, so that the messages can be sent to the router while respecting the NoC backpressure mechanism. As mentioned in the previous sections, such buffering elements are instantiated in a dedicated VHDL entity, named *cpu_tile_q*. More details on the internal operation of the processor tile are reported in a schematic view in Appendix C. Having said that, when the router is not ready to receive any other flits from one of these buffers, it applies backpressure on the tile to block any additional incoming flit. More specifically, it sets the stop bit (nocX_stop_out in Figure 3.3) high in order to block the output ports of the FIFO queues in the direction tile/NoC, so that no further message can be forwarded from the tile. Generally speaking, for not intensive workloads of this type, the NoC congestion is limited and so is the backpressure applied from the NoC on the tile interface. As a consequence, the stop bit is not kept high for more than a few consecutive clock cycles. Hence, the FIFO queues of the buffering stage, in the direction tile/NoC, tend to have available cells at any time without running out of free storage. This is the main reason why the size chosen for the queues is limited to a few cells.

When the test mode is activated, however, the backpressure normally imposed from the router state, i.e. from the NoC, is directly handled from the test interface as explained in the previous paragraphs. In this case, the stop bit directed to the tile is supposed to remain high during all the states of the FSM illustrated in Section 4.1.1, expect for the WAIT state. In this state, the content of one the FIFO queues is retrieved, but not before removing the stop on the corresponding queue output, i.e. setting the relative stop bit low. After that, the flit stored in the FIFO queue is loaded to the PISO-register to be compared with the expected CPU request. During the rest of the states involved in an instruction test, the stop bit remains high. As a consequence, the tile interface is exposed to a much higher level of backpressure than in normal mode, and the probability of congestion in the FIFO queues increases accordingly. This is because, for each FIFO queue, the output port is stopped from the external hold imposed by the test interface during the long injection and extraction phases (each one taking 73 test clock cycles, equal to the number of bits stored in the registers). At the same time the FIFO input ports on the tile side are connected to the AXI interface, and will keep receiving messages from the core until filling up. At this point, the AXI bus can not write on it until a new cell of the FIFO is available, which will only occur at the next successful read and check, when a flit will be extracted from the FIFO

leaving an empty available cell. While this should not be a problem if a unique communication channel was to be used for the NoC, it turns out to be critical when a multi-plane NoC is used. There are several cases, where a long sequence of CPU requests on a certain NoC plane, is followed by a CPU request on another NoC plane. This translates, at the buffering stage, in a flit directed from the AXI interface to another FIFO queue.
In order to better understand this process, an example is illustrated in Figures 4.5, 4.6 and 4.7.In the specific case illustrated, in normal mode, a long sequence of CPU requests to the plane 1 is transferred to the corresponding FIFO queue and then to the dedicated NoC channel. Thanks to the low workload characterizing the target test application, the overall SoC activity is very low. As a consequence, the transmission can happen without the intervention of any significant backpressure action from the NoC. The whole set of packets directed to the plane 1 is thus transferred within a few clock cycles and the AXI interface can terminate the sequence. Conversely, in test mode, the transmission of the first sequence temporary stops when the corresponding FIFO queue gets full due to the high level of backpressure imposed from the test interface. Figure 4.5 shows how the FIFO queue is gradually filled during the first phase of the test: during the injection of the first flit from the testbench stimulus file, the actual CPU request is already in place and ready for being accessed from the testing logic and stored in the PISO-register. Meanwhile, other messages directed to NoC 1 begin to stack in the available buffering elements. This is the first key difference with respect to the behavior in normal execution, where a significantly lower backpressure imposed from the NoC guarantees that the queues are always not congested.

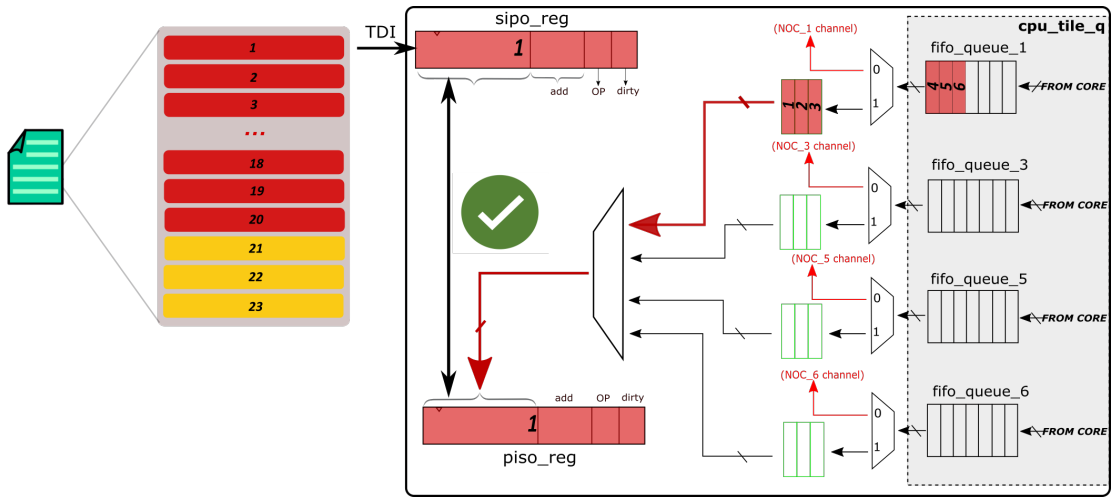


Figure 4.5: Phase 1: The test logic is performing correctly but the FIFO queue connected to NoC plane 1 is gradually filled up.

After a few test-instruction cycles, the FIFO queue gets full as showed in Figure 4.6. At this point, since the AXI bus in unable to write to the input port of that queue, it stalls waiting for the next available cell. Let us now consider that another flit sequence, independent from the one currently handled, is ready to be transferred to its destination NoC channel. In this case, the AXI interface may prioritize this transaction and initialize it. In figure, this can be observed by looking at the FIFO queue connecting to NoC plane 5, which receives the first flit from the CPU.

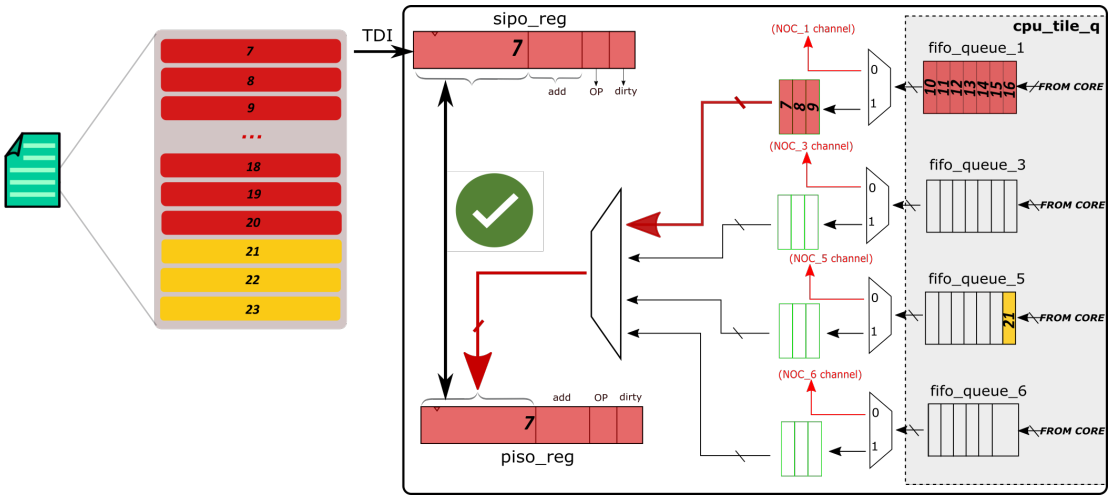


Figure 4.6: Phase 2: The test logic is still performing correctly, but the FIFO queue gets full and the AXI write destination switches to channel 5. A flit is written in this queue.

At the next WAIT state, the test interface finds two FIFO queues with a valid content, i.e. with the void bit set low, as showed in Figure 4.7. The choice of the flit to select for the next read and check is based on a fixed priority implemented by the finite state FSM. This is a weak feature of this test design version, which is intrinsically related to the fact that a single couple of registers is being used to handle operations that can happen simultaneously on 6 different channels. Depending on the priority, one of the two valid flits is forwarded to the testing logic and stored in the PISO-register. In this specific case, the most recently written flit, labeled as flit n21, has priority on the one coming from channel 1, even if it was the last one sent from the CPU, and is forwarded to the PISO-register. Unfortunately, the order of the expected CPU requests stored in the stimulus file is the one imposed by the environment features in normal operating mode. As already mentioned, the lower backpressure level in normal mode makes it possible for the flits sequence directed to NoC plane 1 to be transferred directly in normal mode. The next expected requests written in the stimulus file (flit n8) is thus still coming from NoC plane 1. In test mode, after successfully checking the content of

flit n7 as showed in 4.6, the test interface injects the following expected requests from the stimulus file, i.e. flit n8, and stores it in the SIPO-register. As clearly visible, the expected CPU request (n8) is different from the actual one coming from the tile (n21). This leads to the failure of the comparison operation in the read and check state and, in turns, to the test failure.

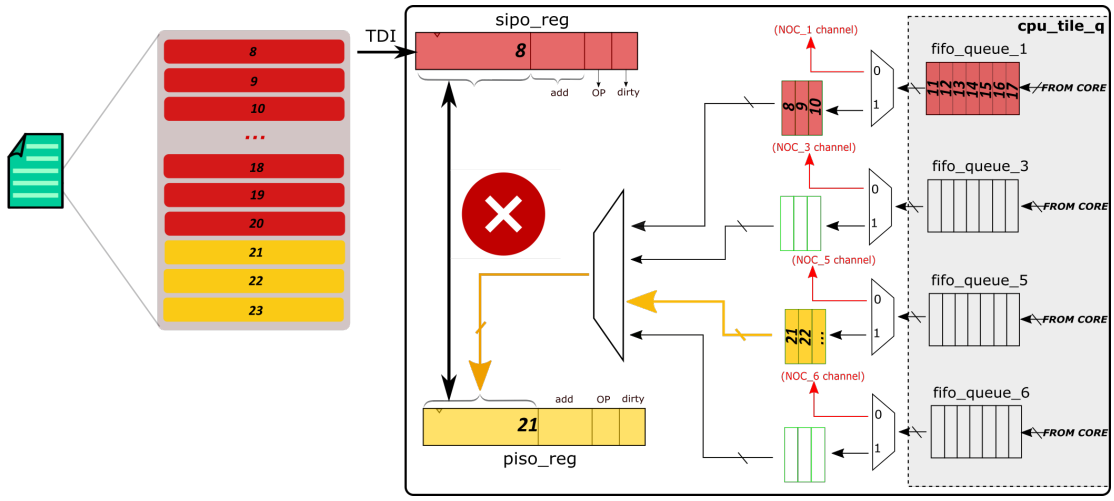


Figure 4.7: Phase 3: The flit from channel 5 is selected following a fixed priority implemented by the FSM, causing the failure of the next read and check.

4.1.4 Conclusions

The simulation of the first implemented design leads to the following conclusions:

- The test interface is suitable for executing test applications composed of a few instructions manually specified at assembly level. In this case, the correct functionality of the tile in performing such instructions can be fully verified.
- When it comes to more complex tests as the one mentioned, the testing logic does not perform correctly because the messages extracted from the test interface can have different order with respect to the simulation stimulus files.
- The adoption of a single register to handle different communication channels comes with the drawback of a significant variation of the backpressure level exerted on the tile interface from the external environment.
- The different level of backpressure raises a variation in the order characterizing the flits sequences reaching the tile interface from the internal AXI bus. This discrepancy, combined with the need to fix a selection priority when more

than one channel is currently storing valid flits coming from the tile in its buffering components, can lead to the failure of a read and check, and in turns, to the test failure.

A smart way to cope with the problems illustrated consists in modifying the test interface so that it can handle different NoC planes independently. This is the starting point for the revision of the design leading the the second version, discussed in detail in the following section.

4.2 Second version - multiple registers

As mentioned, the revision for the final design is driven by the necessity to create additional states for handling each NoC plane independently from the others. This clearly enable a more accurate emulation of the NoC capability to manage several independent communication channels at the same time. The idea laying behind this approach is to minimize the backpressure exerted on the tile from the test interface and get as much closer as possible to the level characterizing the normal operating mode. In this way, it is possible to avoid the issues concerning the order of the requests coming to the interface in situations similar to the one illustrated in the previous section.

In order to accomplish the individual NoC plane management, a duplication of the implemented logic is necessary. As opposed to the previous design, where a single register is used to store flits exchanged across multiple NoC planes, the current version implements dedicated logic for each NoC plane in order to decouple the relative operations.
A first crucial point to consider before diving into the modification of the test control unit and datapath, is the change required for the input stimulus used to transmit the simulation lists to the testbench in test mode. In the previous version, a single stimulus file containing the expected order of transactions across all the NoC planes was used. With the new implemented version, however, each channel is handled independently from the others. As a consequence, a different stimulus file is needed for each NoC plane. Each of those files contains the expected order of transactions happening on a certain plane. In practical terms, this is done by simply modifying the script used for editing the simulation lists into a format readable from the testbench. Compared to the previous flow, six *.txt* files are now obtained, one for each NoC plane.

4.2.1 FSM

The modifications affecting the control unit are discussed below and summarized in Figure 4.8. In addition, the RTL code section reporting the state which manage a single NoC Plane is reported in Listing 4.1.
First of all, once the test mode is activated, the FSM shifts to the state HANDLE1. This state is dedicated to the management of the Plane 1. Once in this state, the first operation to perform is to check the content of the first SIPO-register, i.e. a register entirely dedicated to NoC Plane 1. Then, three possible scenarios taking place during the execution of a test are the following:

a. The register is still empty because the test has just started and no flit has been injected yet. In this case, the FSM shifts to the state INJECT 1. The instruction in the first line of the first stimulus file (*stim1.txt*) is injected through the TDI pin and stored in the SIPO-register. At this point, the FSM goes back to the state HANDLE1.

b. The register contains a flit that needs to be sent to the CPU (OP=1). In this case, a further check is necessary to verify whether the FIFO queue directing the flits from NoC plane 1 to the CPU is full (FF=1) or not. In the first case, no further operation can be done on this plane until the FIFO queue has a new available cell to store the flit. Hence this plane is no longer manageable for the moment and the FSM shifts to the state HANDLE2 for channel 2. On the contrary, if the FIFO queue can be written, the flit is written to its input port and the FSM shifts to the state INJECT1 to update the register content with the next flit from the trace. Once the injection of the next instruction is completed, the FSM goes back to the state HANDLE1.

c. The register contains a flit that is an expected request issued from the CPU, to be compared with the one actually sent out from the CPU (OP=0). In this case, a further check is necessary to verify whether the FIFO queue directing the flits from the CPU to NoC plane 1 is empty or not. In the former case (FE=1), there is not any request issued by the CPU on that plane. Once again, this means that no further operation can be done on that plane, i.e. the plane is not manageable and the FSM shifts to the state HANDLE2. In the latter case (FE=0), the test interface sets to 0 the stop bit controlling the output of the FIFO queue in order to extract the CPU requests and store it in the PISO-register. The FSM then shifts to the state READ & CHECK (R&C), the comparison between the content of the two registers is performed and the TDO is driven accordingly with the same rule established in Section 4.1.1. Finally, the system goes back to the INJECT1 state to update the content of the register with the next instruction from the corresponding trace.

The previous description applies in the same way to the other plans. When the plane 6 is reached and it is not manageable, the FSM shifts to the IDLE state. From here, the cycle restarts from the first plane and the test is completed when all the stimulus files have been read, with the FSM remaining in the IDLE state. To be more precise, an additional state is crucial for the system operation, but not reported in Figure 4.8 for simplicity. Unlike the previous design, the testbench now interacts with six different files, each one containing the information of a NoC plane. Then, the testing logic must include an additional feature to communicate to the testbench the plane of interest before any injection. This can be simply done by adding a state, named REQUEST_INSTR, which precedes any INJECTX state in the FSM displayed in Figure 4.8 and performs the following operations:

- Extract the content of an additional register (named *plane_reg* in the datapath in Figure 4.9), that is used to keep track of the plane currently handled by the testing logic. Such register is thus updated every time that the FSM shifts from one plane to another, i.e. every time there is a state transition of the type HANDLEX => HANDLEY.
- Based on the address stored in plane_reg, it redirects the FSM to the proper INJECTION state so that the flit passed from the TDI pin is stored in the correct SIPO-register.

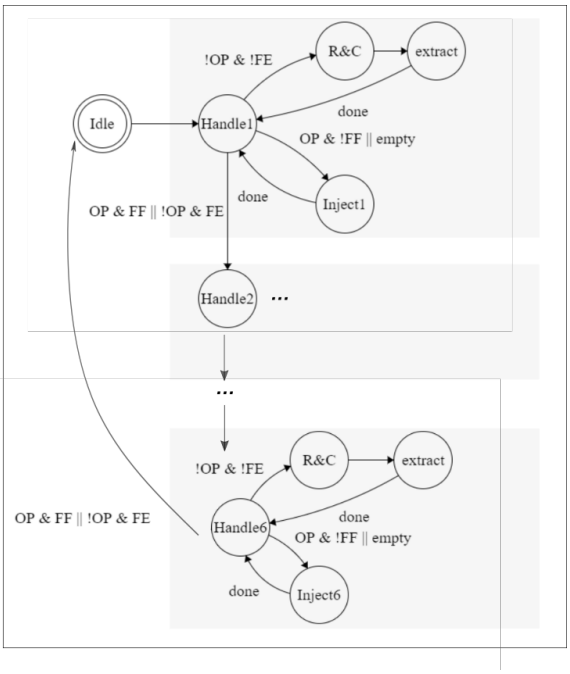


Figure 4.8: FSM revision.

Listing 4.1: Individual plane management.

```
...

when handle2 =>
          if SIPO_done_i(2) = '1' then
            if op_i(2) = '0' then            -- instr is an exp req.
              if test2_cpu_data_void_in = '0' then  -- check queue
                v.compare := "010000";
                v.state   := read_and_check;
              else                                --plane not manageable
                v.state := handle3;
              end if;
            else                            -- instr to write to CPU
              if fwd_wr_full_o(2) = '0' then -- check queue
                v.compare := "010000";

                we_in(2)       <= '1';
                v.PISO_load0   := '1';
                v.PISO_clear0  := '0';
                v.state        := request_instr;
              else                             --plane not manageable
                v.state := handle3;
              end if;
            end if;
          else                               -- register still empty

            v.state       := request_instr;
            v.compare     := "010000";
            v.PISO_load0  := '1';
          end if;
```

4.2.2 Datapath

In order to implement the functionality illustrated in the previous section, it was necessary to duplicate the existing logic for each NoC plane. The result obtained with this modifications is showed in Figure 4.9

Looking at the upper part of the figure, which shows the logic instantiated in the direction test interface/tile, one register for each NoC plane is now instantiated, as opposed to the previous design where just one register was used. The demultiplexer, previously used to redirect the content of single SIPO-register to the correct NoC plane, has now changed its purpose: it is now used for directing the TDI serial input information to the correct SIPO-register. As mentioned, the additional register plane_reg, visible on the bottom-left corner of the image, is used to store the value of the current plane being handled from the NoC. Before any injection, its content is serially shifted out through the TDO pin to inform the testbench on the specific

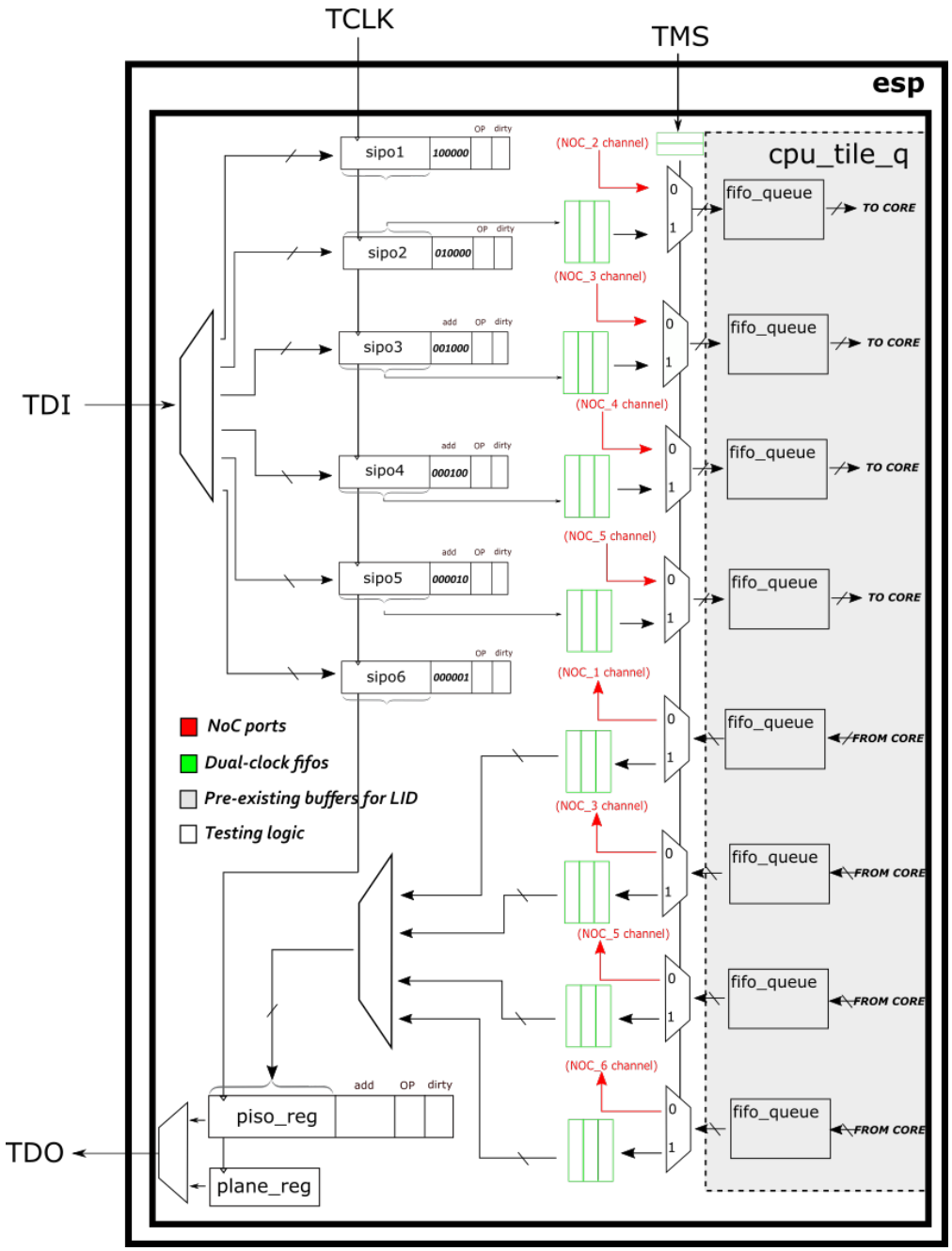


Figure 4.9: Datapath revision.

stimulus file to access in order to retrieve the next flit to inject in the logic. At this point, it is also used by the demultiplexer to direct the input flits injected from the TDI pin to the proper SIPO-register. On the other hand, the rest of the

logic instantiated on the path going from the CPU to the test interface remains untouched.

4.2.3 Simulation

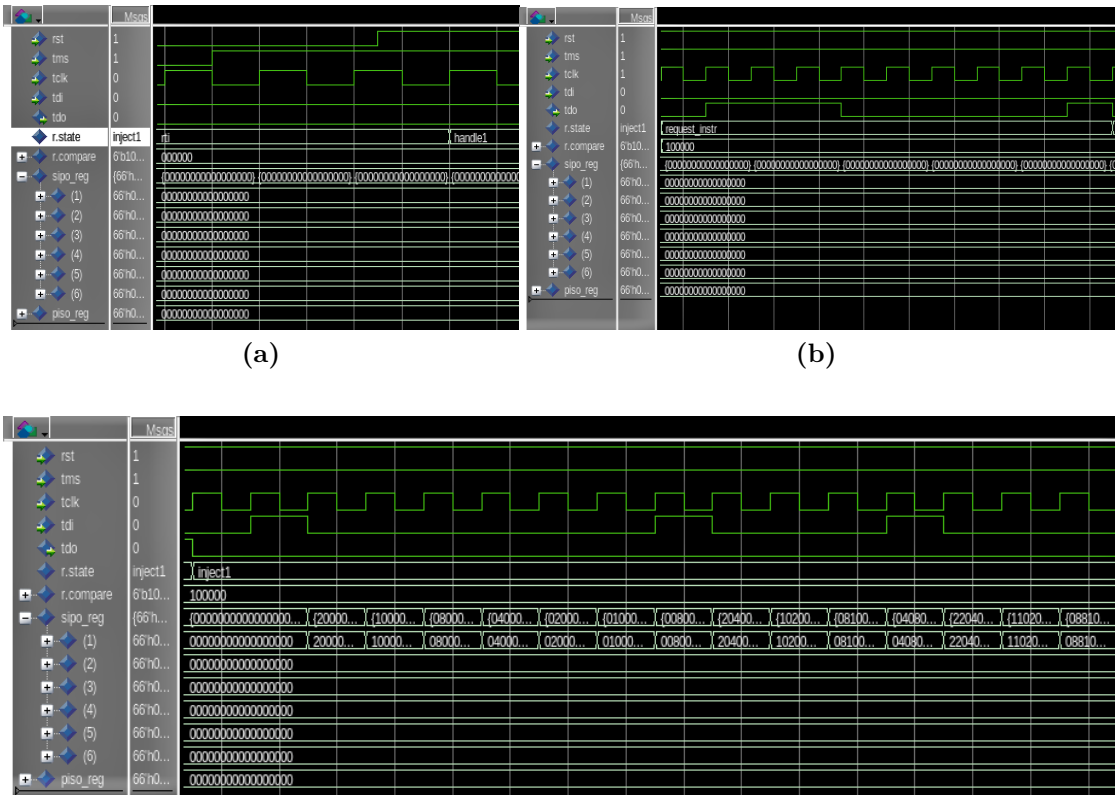
The simulation framework illustrated in the previous section for the first design is adopted in the same way here.

As expected, the Debug Unit completes the test successfully with simple programs composed of a few instructions manually written in Assembly. This design, however, is characterized by a decrease of performance in terms of test speed, due to the independent, plane-by-plane management of the interface. In fact, differently from the previous design, when a new CPU requests reaches one of the ports at the tile interface, it might take several transitions across HANDLE_X states to reach the plane of interest. Nevertheless the speed performance overhead is negligible and not relevant in a functional test of this type. In the same way, it is convenient to specify that the area overhead coming with the logic duplication does not have any impact on the test performance, since the unit footprint remains extremely small and thus not affected by the constraints of the physical design flow.

The waveforms in Figure 4.10 show the test interface activation phases during the execution of the default test application illustrated in Section 3.4. In this specific case, the TMS bit value is set high before the reset is released, as showed in Figure 4.10a. Hence, the tile interface is disconnected from the NoC and connected to the test interface from the beginning of the simulation, in order to verify the execution of all the stages of the system boot, together with the target application program. Nevertheless, as mentioned in the previous sections, the logic is built in such a way to be used even in the middle of a normal execution.

As soon as the test is activated, the interface shifts to the state HANDLE1. At this point, since the SIPO-register of plane 1 is still empty, the FSM shifts to the state REQUEST_INSTR to request the first flit of the stimulus file dedicated to plane 1. This is done, as showed in figure 4.10b, by shifting out the content of the plane_reg register through TDO, in this case equal to "11000001". The tag composed of the first two bits ("11") is used as an opcode in order for the plane vector to be distinguished from a flit by the testbench, while the remaining bits ("000001") indicate the plane address 1-hot encoding (i.e. plane 1). Once the extraction is completed, the FSM shifts to the INJECT1 state and the testbench starts to inject the flit. As visible in Figure 4.10c, the content of the first SIPO-register changes accordingly

A similar procedure is repeated for all the planes, since none of them has a valid content at the beginning of the test. However, when Plane 3 is reached, a different behavior is noticed and a set of consecutive injection phases takes place, as visible in Figure 4.11. As a reminder, NoC plane 3 is generally used for



(a) **(b)**

(c)

Figure 4.10: Test interface activation sequence.

both sending and receiving information from the tile. Nonetheless, in this type of test, it turns out to be a mono-directional plane only used to send memory response to the CPU tile. As a consequence, the flits stored in the corresponding stimulus file are write-flits that must be written to the tile, and no CPU requests are present. When the first write-flit is injected in the SIPO-register 3, it is then directly forwarded to the buffering logic, passing for the asynchronous dual-clock FIFO queue for the synchronization, and another flit from the stimulus file is requested by the testing logic. This is repeated as long as there is enough storage in the dedicated buffers. When those queue get full, the backpressure signals in the direction tile/test-interface is activated and the FSM shifts to the HANDLE4 state, since no further operation can be carried out on plane 3. The same process happens for plane 5, as visible in Figure 4.12.

Figure **??** shows the transactions involved in a successful READ AND CHECK operation, performed on plane 5 and 1, respectively. In the first case, the FSM gets to the state HANDLE5 after having verified that none of the previous planes

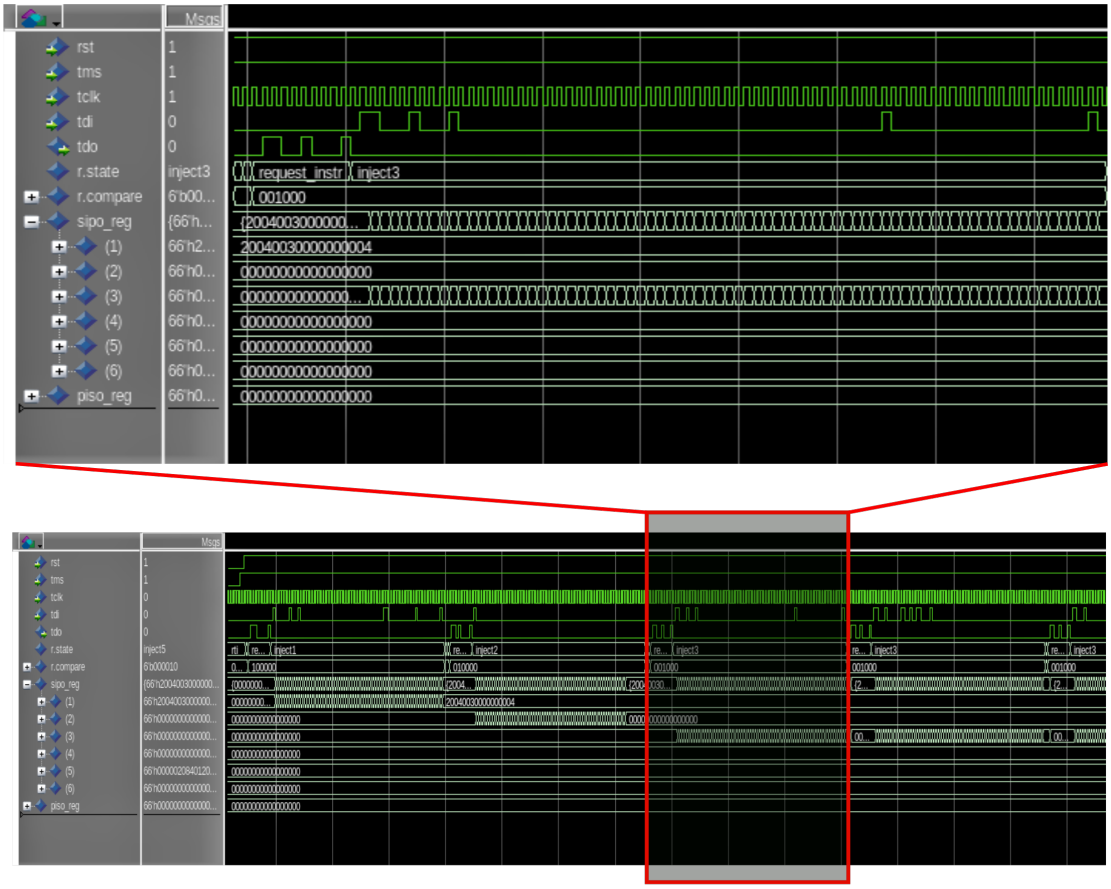


Figure 4.11: Test: initial injection phase across multiple planes and detailed view of the first serial injection on NoC plane 3.

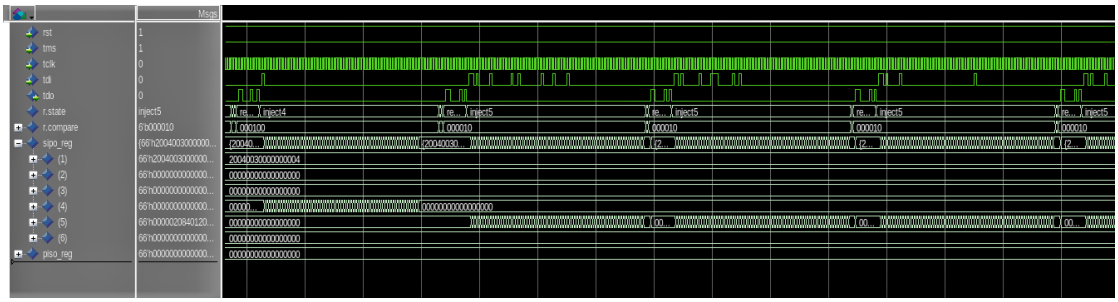


Figure 4.12: Detailed view of the first serial injection on NoC plane 5.

are manageable. In fact, for each of those planes, the corresponding input buffers have been previously checked to see if they contain enough storage to write an

other flit or, in alternative, if the output buffers contain flits coming from the tile to be checked. Since none of these scenarios were found to be consistent, the FSM keeps changing state reaching NoC plane 5. At this point, the content of the register is checked: the flit is an expected CPU request. Then, the content of the corresponding output FIFO queue is checked and a flit is retrieved in it and stored in the PISO-register to be compared with the expected value. At this point the FSM shifts to the state READ AND CHECK. As visible from the waveform in figure 4.13, the content of the SIPO-register and the one of the PISO-register match (in this particular case, the 66-bits flit content is expressed in hexadecimal notation and equal to "00000000200421502") and the TDO is raised accordingly to communicate the test success to the testbench, that is now ready to extract the content of the PISO-register. A similar condition takes place in the second case reported, with the difference that the incoming CPU request is now travelling on channel 1. The value "0000000009FEFFFF8" is checked with success since the content of the corresponding registers match, as visible from the figure 4.14.

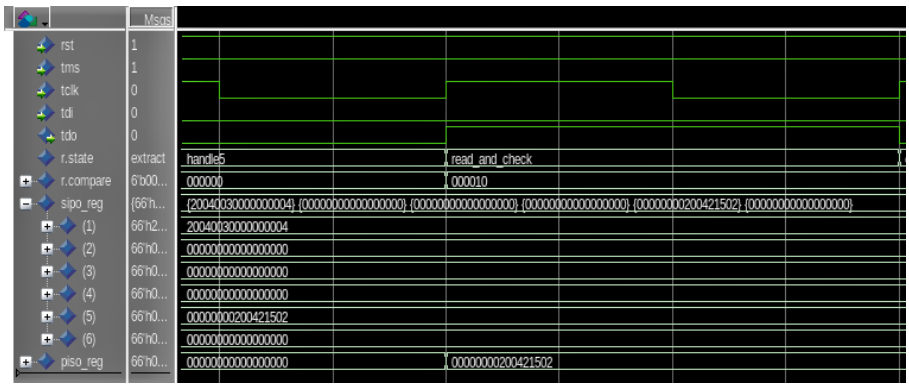


Figure 4.13: READ & CHECK sequence on NoC plane 5.

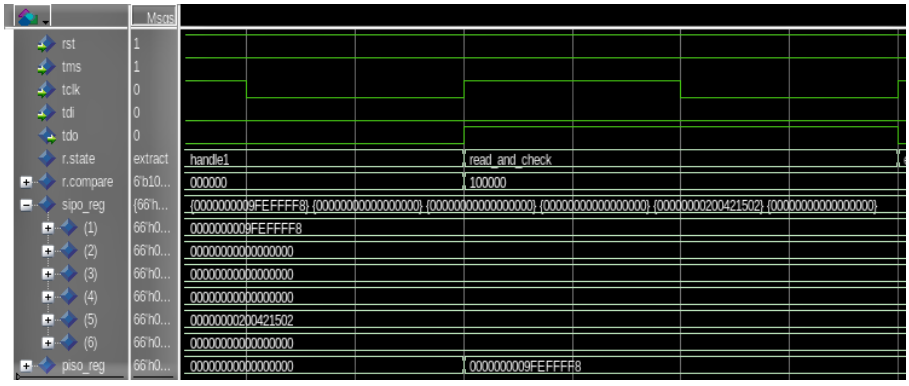


Figure 4.14: READ & CHECK sequence on NoC plane 1.

Overall, the test simulations proceeds for a much higher number of test instructions, going far beyond the point of test failure detected in the first design. This is a clear indication that the approach adopted for decoupling the planes management goes in the right direction. Despite a performance increase, however, the simulation of the default program application does not complete successfully due to a *RC* failure.
A significant amount of time has been spent to find out possible reasons for this behavior, but a unique response was not reached. The most reasonable hypothesis is that the design under study shares the same intrinsic problem of the first version, but in a less pervasive way. In other words, the use of multiple registers is simply not sufficient to correctly reproduce the system backpressure existing in normal operating mode. Even if the NoC planes are now handled independently, the FIFO queues conveying data from the tile to the test interface remain blocked during the time-consuming injection and extraction phases. In addition, this whole set of operations is performed by a logic synchronous with the test clock, which is characterized by a much lower frequency than the reference clock of the system. As a consequence, the data congestion in the queues increases accordingly, and so does the probability of having full queues blocking the access from the AXI interface. This type of mechanism may change, at a certain point during the execution of burst transactions, the order of of collected requests with respect to the expected one.

4.2.4 Conclusions

The simulations clearly show the performance improvement delivered by the revised design. In particular, the following aspects can be outlined:

1. The final version of Debug Unit can be used for the execution of simple programs as that exemplified in Section 3.4, with a minimal performance overhead not relevant for the purposes of a functional test.

2. The adoption of dedicated logic for decoupling the operations on different NoC planes helps reducing the test interface backpressure, thus getting closer to the conditions of normal operation. This allows us to overcome some of the critical sections of the simulation trace leading to *RC* failure in the previous design.

3. The modifications proposed are not sufficient to extinguish the backpressure gap between different operating modes and the test is thus not able to complete. The intrinsic concurrency of the multi-plane NoC, combined with the service queues in the tile that decouple message types from each other, introduces a partial non determinism in the simulation. Specifically, the order in which

different message types are interleaved may change based on the speed of the interface. The latter changes dramatically when switching from the 64-bit parallel NoC plane to the serial test interface.

Chapter 5

Conclusions

This dissertation has proposed a new flexible approach to leverage the thesis that design for testability (DFT) in single-core functional verification of heterogeneous SoCs. The methodology proposed is based on the operation of a Debug Unit that can be leveraged to perform a unit test at every stage of the design, from RTL verification down to post-fabrication testing of physical implementations targeting advanced technology nodes.The development of the verification flow and the design implementation have been conducted following some major drivers. On one side, a great effort was put in adapting the test integration process to the platform-based approach of ESP, making the test-unit a new platform service offered by each tile hardware socket. On the other side, the constraints imposed by the physical design flow determined the type of test access mechanism and the I/O porting strategy.

The experimental results that were obtained from running simulations at RTL show that the Debug Unit is capable of running tests on the ESP processor tile. The flexibility of the test methodology makes it possible to replicate the results for the other tiles, and at different stages of the design, supporting DFT can be used to provide a general verification methodology. In fact, while the effectiveness of the proposed methodology has been demonstrated on the Embedded Scalable Platform (ESP) developed at Columbia University, it is generally applicable to address the increasing complexity of heterogeneous SoC designs.

5.1 Future improvements

In order to improve the performances delivered by the test interface with more complex test applications, a strategy to decrease further the backpressure gap between normal operating mode and testing mode is necessary. The main factors

feeding this gap are the long extraction and injection phases interleaving between two consecutive accesses to the FIFO queues regulating the tile interface. A possible strategy to overcome these limitations consists in executing the test in a different way from the one proposed in section 3.3.

More specifically, a different approach for the flits transfer mechanism is proposed in order to avoid the time-consuming serial injection and extraction of the instructions into the Debug Unit logic. A possible way to do that consists in implementing a local on-chip memory to instantiate inside the tile, where the entire stimulus files would be stored. In such a way, the Debug Unit could retrieve the next flit to store in the corresponding register by simply accessing adjacent locations of the memory. At the same time, the memory should also include enough free space to store the simulation result flits and avoid the extraction cycles. Amongst the DFT trends highlighted in Section 2.2, the proposed solution clearly embraces BIST methodology. In particular, two possible memory implementations could be adopted:

- Using a Read-only memory (**ROM**), with a fixed test application stored in it.
- Using a Random Access Memory (**RAM**). In this case, the test application can be modified by loading a new program through the serial TDI interface. After that, the test can be run.

The adoption of this alternative approach may deliver a significant performance improvement of the test interface during the execution of burst transactions which were found to be critical in the studies conducted.
On the other hand, it comes with some critical drawbacks. First of all, despite a minimal storage required for a simple test program, instantiating private memory blocks in each tile leads to a significant area overhead. Hence, it is necessary to check that this implementation choices are compatible with the backend flow constraints. In addition, the use of a serial test interface to load a single instruction at a time comes with a potential advantage which has not been stressed so far. In fact, in case of a mismatch or deadlock condition during the test execution, the simulation trace could be interactively modified with a dedicated application which controls the FPGA used for testing. The content of the trace could therefore be adjusted to prevent false mismatches of instructions that are misplaced when running in test mode. In contrast, this is not possible when the Debug Unit retrieves the test flits from a local memory, because its content can not be interactively modified during the test execution.

To conclude, a practical and flexible solution to study in the future could include both proposed types of test. A third operating mode could be added to the existing ones (i.e. normal mode and serial test mode) and used to access the content of a

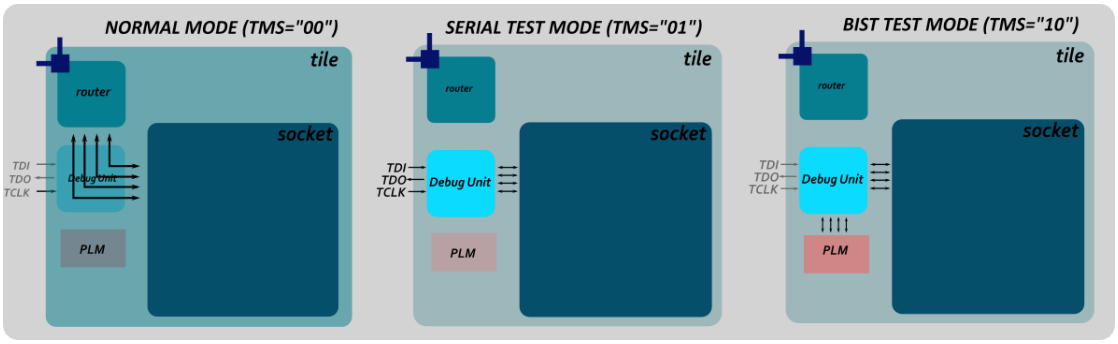


Figure 5.1: Third operating mode for Built-in unit test.

private local memory instantiated in the tile. A schematic view of the new proposed arrangement is showed in Figure 5.1.

Appendix A

End of Dennard Scaling

As commonly accepted, Moore's law has driven the semiconductor industry progress by predicting a grow in the number of transistor fitting in the unit of area of a factor of 2 every eighteen months. In first place, this was made possible by the technological progress which allowed for a gradual and constant decrease of the transistors' feature size. Together with the density increase, this made it possible to have a similar progress in the operating speed of the devices, and thus the final operating frequency of the system. Figure A.1 reports the experimental data summarising the most relevant effects of technology scaling on the IC performaces.

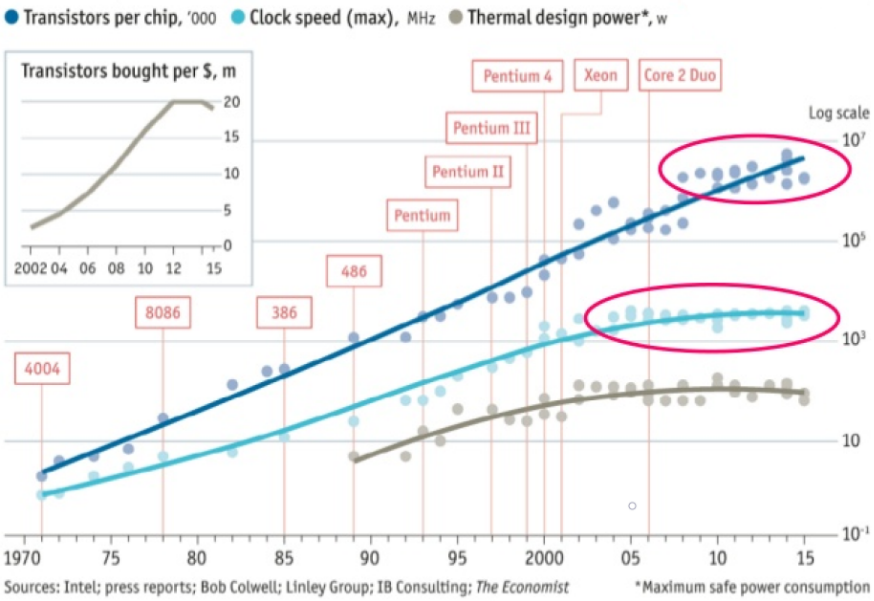
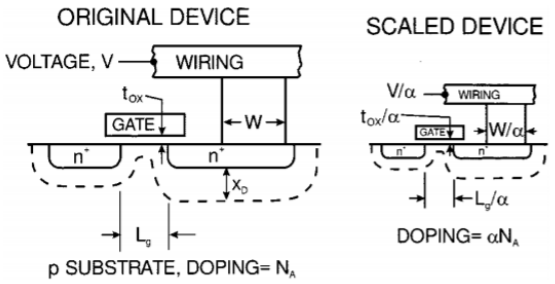


Figure A.1: Technology scaling effect on performance [33].

In order to better understand the origin of this trends, together with the reason for their recent breakdown, it is necessary to have a closer look at the physical phenomena regulating the operation of the transistor. A guide for MOSFET scale-down is given by Dennard observation, which start from the equation of dynamic power consumption for a CMOS logic gate showed below, to make predictions on the effects of scaling on the a CMOS-based system performance.

$$P_{dyn} = \alpha * C * F * V_{dd}^2 \tag{A.1}$$

If a generic parameter α is used for scaling down a standard geometry such as the one in figure, the consequent scaling relations are summarized in table B.2b.



(a)

Physical parameter	Constant-Electric Field Scaling Factor	Generalized Scaling Factor	Generalized Selective Scaling Factor
Channel length, Insulator thickness	$1/\alpha$	$1/\alpha$	$1/\alpha_d$
Wiring width, channel width	$1/\alpha$	$1/\alpha$	$1/\alpha_w$
Electric field in device	1	ε	ε
Voltage	$1/\alpha$	ε/α	ε/α_d
On-current per device	$1/\alpha$	ε/α	ε/α_w
Doping	α	$\varepsilon\alpha$	$\varepsilon\alpha_d$
Area	$1/\alpha^2$	$1/\alpha^2$	$1/\alpha_w^2$
Capacitance	$1/\alpha$	$1/\alpha$	$1/\alpha_w$
Gate delay	$1/\alpha$	$1/\alpha$	$1/\alpha_d$
Power dissipation	$1/\alpha^2$	ε^2/α^2	$\varepsilon^2/\alpha_w\alpha_d$
Power density	1	ε^2	$\varepsilon^2\alpha_w/\alpha_d$

(b)

Figure A.2: (a)Scaling of a traditional CMOS technology. (b) Technology scaling rules [34].

The first column of the table shows what happens in what is commonly referred to as *constant field scaling* (condition obtained by coupling the scaling of voltages with an equal increase of doping intensity [34]). Combining the results in the table with the Equation A.1, it is clear that a constant field scaling results in a circuit speedup of factor α and a density increase of factor α^2. In figure A.1, however, it can be noticed that the average power consumption has increased as well, even if with lower intensity. This means that the industry took advantage of performance scaling even beyond constant power-scaling [3].

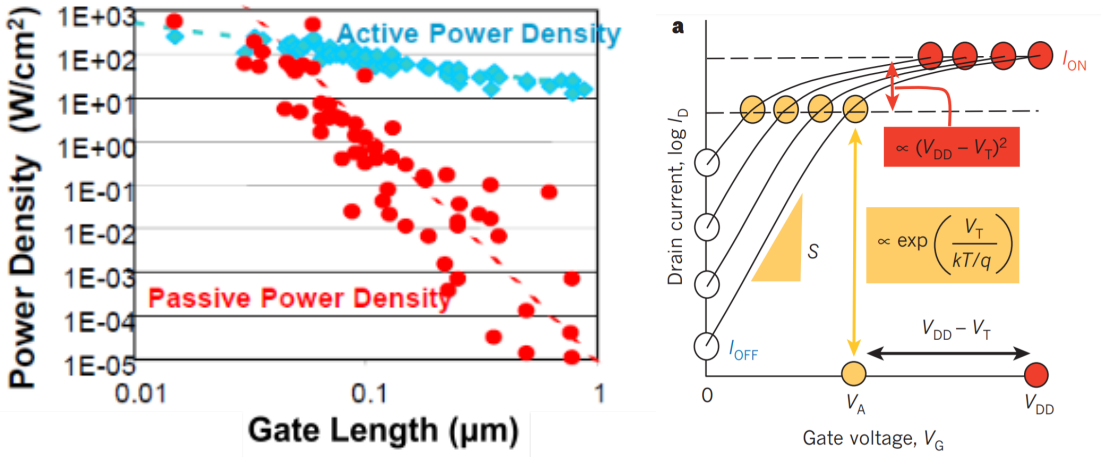
In the past decade, however, the aggressive scaling of geometrical features was not accompanied by an equal progress in frequency and power scaling as expected from Dennard Model. This was mainly because at this scale, certain collateral phenomena that were previously causing negligible effects on the final figures of the system, come in with a significant impact on the final performance. A clear example is showed in Figure A.3a: the static power consumption has always been present but became non-negligible upon aggressive gate dimensions scaling. A first major constraint on the voltage scaling comes from the fixed subthreshold slope (S) of the metal–oxide–semiconductor FET (MOSFET) characteristic. In a MOSFET, the current switching process involves the temperature-dependent injection of electrons over an energy barrier [35]. In other words, the initial current flowing in the device channel, for low values of applied gate voltages, is a purely diffusive process. This turns into an exponential dependence of the current intensity on the applied gate voltage, as showed in the following equation A.2:

$$I_d = q\frac{W}{L}(\frac{n_i}{N_A})^2\frac{(\frac{k_B}{q})^2}{E_s}\mu_{eff}e^(\frac{qV_{GS}}{mk_bT})A \qquad m = 1 + \frac{C_D}{C_{OX}} \tag{A.2}$$

This type of phenomenon sets a fundamental limit to the steepness of the transition slope from OFF to ON state. From the previous equation, the voltage sweep required to increase the current value by an order of magnitude in the subthreshold region can be obtained as showed in equation A.3:

$$S = \frac{dV_g}{d\psi_s}\frac{d\psi_s}{d(log_{10}I_d)} \simeq (1 + \frac{C_d}{C_{ox}})ln10\frac{kT}{q} \rightarrow \frac{kT}{q}ln10 \simeq 60mVdecade^{-1} \mid T = 300K \tag{A.3}$$

where C_d and C_{ox} are the Depletion and the Oxide capacitances, and ψ_s is the surface potential in the channel.



(a) **(b)**

Figure A.3: (a)Power trends upon aggressive gate scaling [36].(b) MOSFET transfer characteristics in subthreshold region [35].

A key factor ignored by the Dennard model is the diffusive nature of the current in the subtreshold region. As showed by Equation A.3, the slope in this region is directly dependent on the physical unit regulating the injection of electrons during diffusion: the temperature.Figure A.3b highlights the direct impact of the subthreshold behavior on the MOSFET performance.

As mentioned, the geometry scaling has allowed for a gradual reduction of the supply voltage V_{DD}, which in turns, guarantees an improvement of the device performance.Nevertheless, it is crucial that the supply voltage reduction is accompanied by the threshold voltage scaling, in order to keep the overdrive factor $V_{DD} - V_T$ and the ON current high. Taking into account the incompressible value of the subtreshold slope, however, a tenfold increase of the OFF current is impossible to avoid as showed in the Figure A.3b. The only way to obtain a lower, sub-thermal value for S is to design a device capable of exploiting a different physical mechanism to perform the electrons injection at this stage of the conduction.

Along with the subtreshold slope and the associated increase of leakage power, several post-Dennardian nano-meter scaling challenges started to affect IC performances. Amongst those, short-channel effects, drain-induced barrier lowering and velocity saturation are worth being mentioned.

The semiconductor industry has reacted to the inevitable failure of Dennard scaling by proposing a shift from the classical bulk planar transistors to non-classical CMOS device architectures. In this perspective, the most important technologies developed to overcome the aforementioned limitations are Fin FET annd FD SOI, showed respectively in figure A.4 and A.5. Both of them share the need to

improve the channel electrostatic to have a higher control on the depletion zone and avoid additional substreshold swing deterioration. In order to do that, both these solutions were based on different ways to make the conductive channels thinner and better controllable from the gate.

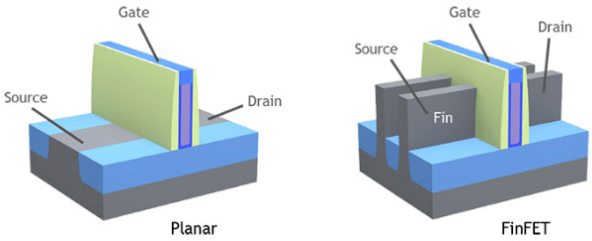


Figure A.4: Fin FET technology A.4.

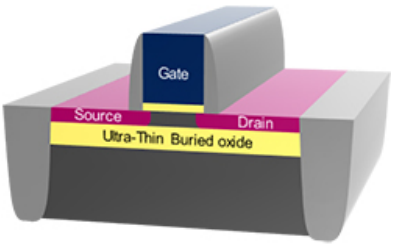


Figure A.5: FD-SOI techonology A.5.

Despite recent efforts in pushing the envelope beyond the limits of thermionic injection with additional technology boosters (negative capacitance transistors) or alternative conductive mechanisms (Tunnel field effect transistors) focused on delivered performance, semiconductor producers encountered increasing difficulties in further reducing the supply voltage. To conclude, the present trends indicate that the scaling of the conventional MOSFETs is fast approaching the end of its useful life time [37].

Appendix B

AMBA-AXI protocol

The Advanced extensible Interface (AXI) is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) specifications. A typical AMBA-based SoC consists of a high-performance system bus (AHB) and a peripheral bus (APB), connected via a dedicated bridge as showed in figure B.1.

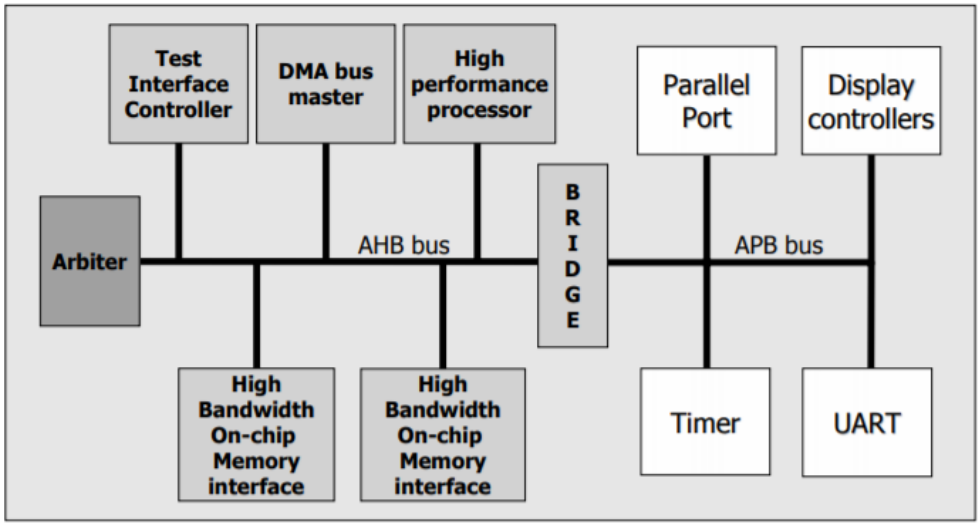


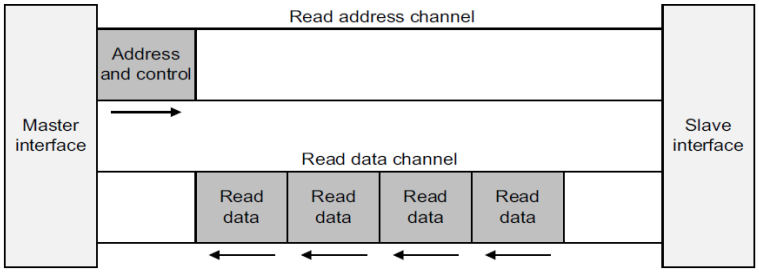
Figure B.1: Classical AMBA-based SoC [38].

The APB is normally used for connecting low-bandwidth peripherals. It doesn't support pipeline operation and burst-data transfers.

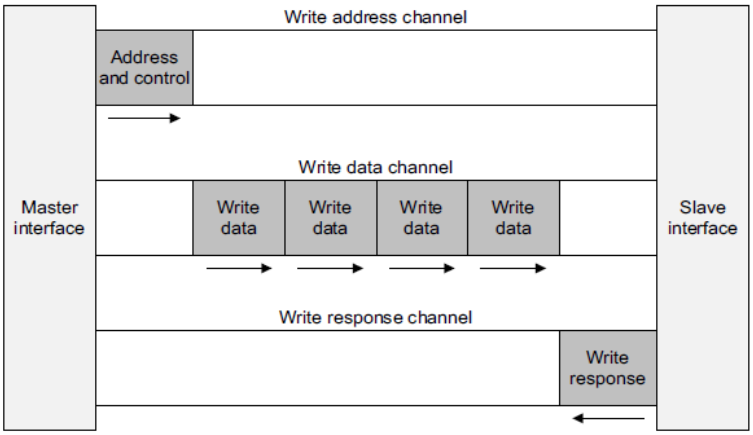
In the AHB protocol, the bus master initiates read or write operations by providing address and control information and the bus slave responds within a given address-space range [39].This protocol supports multiple bus masters and slaves, as showed in the example in Figure B.1. In addition, it enables pipelined

address-phase operation and split transfer and several types of burst accesses, useful to improve the delivered performance during cache line fill [40].

Despite the improvements offered by multi-layer AHB and AHB-lite, extensions of the standard AHB, the architecture implementing the protocol has turned out to be unsuitable for facing the challenges of modern SoCs. The main reason is that AHB is transfer-oriented. Being a shared bus protocol, a single flit can be written to or read from the slave in each transaction. In addition, when a slave is not in the condition of responding to a master request, the master will remain stalled.



(a)



(b)

Figure B.2: (a)AXI Read transaction (b) AXI Write transaction.

AXI protocol, on the contrary, is strongly transaction-oriented. It consists in a high-performance, multi-master and multi-slave communication interface and it is one of the protocol mostly used for on-chip communication. First of all, it has separate address and data phases. Separate channels are provided for write and read transactions, as showed in Figures B.3a and B.3b. In addition, each transaction is composed of address, data and response transfers, each of them on separated channels. Similarly to AHB, it supports burst-based operations.

The great advantage coming with this architecture is the support for out-of-order transaction completion thanks to the specific ID tag carried by each data-item involved in a transfer [40].

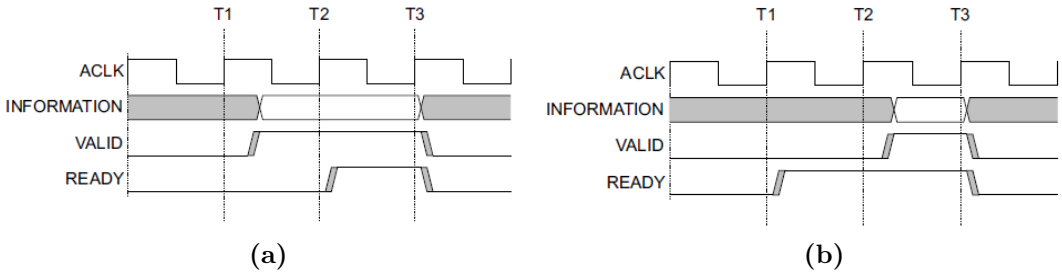
The handshake mechanism proposed by AXI is based on the two signals VALID and READY. In a transaction, the master sets the VALID signals high to indicate that the address/data information is available. At the same time, the destination slave raises the READY value to indicate that it is in the condition of accepting the incoming information. The transaction can take place when both the VALID and READY signal are high [41].

Based on the protocol specifications, certain simple rules are imposed, as reflected in the examples of Figure B.3.

1. A master source doesn't have to wait for a high READY value to assert the VALID signal.

2. In contrast, a destination slave can wait for the VALID signal to be high to assert the READY signal.

3. Finally, in order for the handshake to complete successfully, the READY and VALID signal must remain contemporary at high value for at least a clock cycle.



(a) **(b)**

Figure B.3: AXI handshake examples [41].

Appendix C

CPU tile operation

In this appendix, a more detailed insight of the internal operations of the CPU socket are presented, based on the schematic view showed in figure C.1. To simplify the representation and focus on the tile functionality, the optional and configurable L2 cache is not discussed, neither reported in the schematic representation.
As visible from the schematic, the CPU tile instantiated at the top level of the RTL hierarchy, together with the system interconnect implemented with the multi-plane NoC described in Paragraph 1.3.1. For each NoC plane, given a direction (indicated in figure with the letters n,s,e,w) defining the source/destination of the transaction with respect to the tile location, two buses are dedicated to transfer data from/toward such direction. Those are labeled in Figure 1.7 as nocX_data_in_Y / nocX_data_out_Y, where X stands for the NoC plane and Y indicates the direction. For the sake of simplicity, the buses are grouped in batch of 4 signals according the plane of interest. A peculiar feature of the implementation showed is that the router is not instantiated at the same level as the interconnect wires, but inside the tile's component *sync_noc_set*. The reason for this design choice is strictly driven by the necessity to facilitate the synthesis step and does not involve any particular advantage in terms of functionality. A single-plane router instance is used for each NoC plane, but the whole component is equivalent to a multi-plane NoC router. Each router instance has 4 couples of I/O ports, one for each direction, and an additional port to communicate with the tile. In this case, however, a couple of dual-clock asynchronous fifo queues are instantiated to handle flit synchronization across the two different clock domains of the NoC and the Tile. On the tile side, such queues are directly connected to the multiplexers/demultiplexers of the test interface showed in green in the schematic. The operation of the test interface is the one described in the dissertation, regulated from the four I/O pins (namely TDI,TDO,TMS,TCLK). At the same level, the component *cpu_tile_q* implements the buffering stage often mentioned in the explanations of the LID theory. In figure C.1, just a few fifo queues are showed to simplify the representation, but a total

amount of 21 queues is actually utilized in this component.

On the processor side, the access to the FIFO queues is managed by the 64-bits AXI interface implemented in the component *cpu_axi2noc*. In the schematic proposed, just two of the many paths provided to communicate with the NoC are illustrated in order to keep the representation simple enough for the reader understanding. In particular, juts the paths issuing memory requests through NoC Plane 1 and the path delivering memory response through NoC 3 are highlighted. The AXI controller plays a double role: it takes care of extracting/injecting flits from/to the queues and at the same time provides respectively the read and write enable signals to do that with the appropriate priority. For what concerns the data exchange, the AXI just acts as an intermediate component forwarding requests/response using the dedicated *somi* and *mosi* vectors showed in picture to interface with the core. On the other side, the controller plays a more active role in determining the order of operations to perform. This task is performed by the FSM showed in picture, implementing the AXI protocol.

The core comes with separate ports to handle communications with different components of the SoC. Amongst those, the most relevant one are reported in figure. In particular, the ports *romi/romo* are used to communicated with the chip bootroom. As a consequence, they are connected to the fifo queues of NoC Plane 5. The same applies for the ports *clinti/clinto* , which enable the communication between the processor and the interrupt controller in the IO tile (CLINT stands for Core-local Interrupt Controller). The ports *drami/dramo* are used for communicating with memory, and are thus the one involved in the example paths reported in figure. In addition messages directed to memory-mapped I/O registers are forwarded by the socket to the NoC Plane 5 through the APB adapter [2].

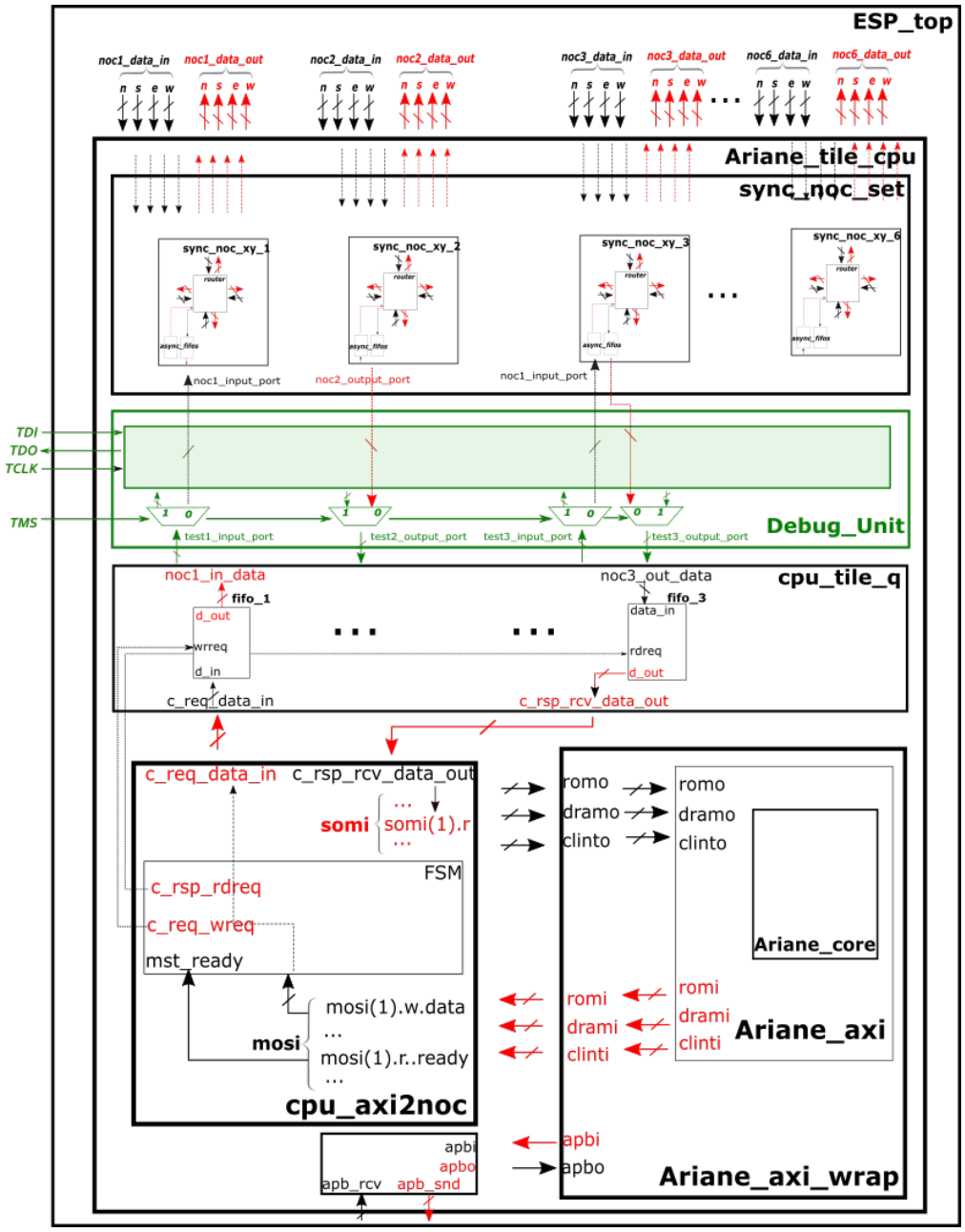


Figure C.1: Detailed internal structure of the CPU hardware socket.

Bibliography

[1] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. «The RISC-V Instruction Set Manual-Volume I:Base User-Level ISA». In: *Electrical Engineering and Computer Science (EECS)* (2011) (cit. on p. ii).

[2] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. «Agile SoC Development with Open ESP». In: *IEEE/ACM International Conference On Computer Aided Design (ICCAD)* (2020) (cit. on pp. ii, 3–8, 10, 83).

[3] Paolo Mantovani. «Scalable System-on-Chip Design». PhD thesis. Columbia University, 2017 (cit. on pp. iii, 2, 4–6, 15, 76).

[4] Davide Giri, Paolo Mantovani, and Luca P. Carloni. «NoC-Based Support of Heterogeneous Cache-Coherence Models for Accelerators». In: *IEEE/ACM International Symposium on Networks-on-Chip (NOCS)* (2018) (cit. on pp. iii, 10, 16, 17).

[5] Luca P. Carloni. «The Case for Embedded Scalable Platforms». In: *Design Automation Conference (DAC)* (2016) (cit. on pp. iii, 4–6, 8, 16).

[6] S. Borkar and A. A. Chien. «The future of microprocessors.» In: *Communication of the ACM* (May 2011) (cit. on pp. 1, 3).

[7] Anand Haridass (IBM Cognitive Systems). «Heterogeneous Computing The Future of Systems.» In: *NITK-IBM Computer Systems Research Group (NCSRG)* (2017) (cit. on p. 2).

[8] John Hennessy. *The End of Moore's Law Faster General Purpose Computing, and a Road Forward*. Stanford University. 2019 (cit. on p. 2).

[9] `https://www.pulp-platform.org/projectinfo.html` (cit. on p. 3).

[10] K. Asanovic' et al. «The rocket chip generator». In: *SemanticScholar* (2016) (cit. on pp. 3, 14).

[11] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994 (cit. on pp. 5, 6, 24).

[12] Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. «High-Level Synthesis of Accelerators in Embedded Scalable Platforms». In: *Asia and South Pacific Design Automation Conference (ASPDAC)* (2016) (cit. on pp. 6, 7, 17).

[13] R. Porter, A. M. Fraser, and D. Hush. «Wide-area motion imagery». In: *IEEE Signal Processing Magazine* (2010) (cit. on p. 6).

[14] Luca P. Carloni. «From Latency-Insensitive Design to Communication-Based System-Level Design». In: *The Proceedings of the IEEE,Vol. 103, No. 11* (2015) (cit. on pp. 7, 11, 12).

[15] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. *On-Chip Networks*. Morgan Claypool, 2017 (cit. on pp. 8, 9).

[16] `https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter` (cit. on p. 9).

[17] `http://www.asic-world.com/systemc/tlm1.html` (cit. on p. 10).

[18] Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. *Vlsi Test Principles and Architectures Design for Testability*. Morgan Kaufmann, 2006 (cit. on pp. 18, 19).

[19] Schmid Alexandre. *Test of VLSI systems*. École polytechnique fédérale de Lausanne (EPFL) (cit. on pp. 19, 20, 22, 23).

[20] Yield Enhancement Working Group. In: *International Technical Rescue Symposium*. Tokyo, 2007 (cit. on p. 21).

[21] Semiconductor Group Texas Instruments. «IEEE Std 1149.1 (JTAG) Testability Primer». In: (1997) (cit. on pp. 22, 23).

[22] Matteo Sonza Reorda. «New Methods for efficient functional testinf of SoCs». Politecnico di Torino (cit. on p. 22).

[23] `https://www.eetimes.com/soc-testing-becomes-a-challenge/` (cit. on pp. 23, 24).

[24] `https://www.evaluationengineering.com/home/article/13001959/dft-strategies-for-soc-designs` (cit. on p. 24).

[25] Marko Isomäki. «Processor Debugging Through Ethernet». MA thesis. Chalmers University of Technology, 2004 (cit. on pp. 28, 29).

[26] `https://en.wikipedia.org/wiki/FPGA_Mezzanine_Card` (cit. on p. 38).

[27] `https://en.wikipedia.org/wiki/DDR4_SDRAM` (cit. on p. 38).

[28] J. a. Heat M. El-Taha. «Queueing Network Models of Credit-Based Flow Control». In: *Department of Mathematics and Statistics,University of Southern Maine and Sun Microsystems* (2005) (cit. on p. 39).

[29] https://en.wikipedia.org/wiki/PHY (cit. on p. 40).

[30] https://en.wikipedia.org/wiki/Medium_access_control (cit. on p. 40).

[31] https://www.esp.cs.columbia.edu/docs/singlecore/singlecore-guide/ (cit. on p. 44).

[32] https://www.informit.com/articles/article.aspx?p=1647051&seqNum=5 (cit. on p. 45).

[33] https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law/ (cit. on p. 74).

[34] D.J Frank, R.H Dennard, E Nowak, P.M Solomon, Y Taur, and Hon-Sum Philip Wong. «Device Scaling Limits of Si MOSFETs and Their Application Dependencies». In: *Proceedings of the IEEE, March 2001, Vol.89(3), pp.259-288* (March 2001) (cit. on pp. 75, 76).

[35] AM Ionescu and H Riel. «Tunnel field-effect transistors as energy-efficient electronic switches». In: *Nature 479 (7373), 329-337* (2011) (cit. on pp. 76, 77).

[36] B. Meyerson (IBM). In: *Semico Conf.* (January 2004) (cit. on p. 77).

[37] http://userweb.eng.gla.ac.uk/fikru.adamu-lema/Chapter_02.pdf (cit. on p. 78).

[38] Massimo Bocchi. «Architetture di bus per Architetture di bus per System-On-Chip». University of Bologna - Corso di Architettura dei Sistemi Integrati. 2002/2003 (cit. on p. 79).

[39] http://utenti.dieei.unict.it/users/gascia/COURSES/sist_emb_14_15/download/SE08_buses_amba.pdf (cit. on p. 79).

[40] https://www.doulos.com/knowhow/arm-embedded/migrating-from-ahb-to-axi-based-soc-designs/ (cit. on pp. 80, 81).

[41] ARM. «AMBA® AXI and ACE Protocol Specification». 2011 (cit. on p. 81).