HD2BIST: a hierarchical framework for BIST scheduling, data patterns delivering and diagnosis in SoCs

HD²BIST: a hierarchical framework for BIST scheduling, data patterns delivering and diagnosis in SoCs

Authors: Benso A., Chiusano S., Di Carlo S., Prinetto P., Ricciato F., Spadari M., Zorian Y.,

Published in the Proceedings of the IEEE International Test Conference (ITC), 3-5 Oct. 2000, Atlantic City (NJ), USA.

DOI: 10.1109/TEST.2000.894300

HD2BIST: a Hierarchical Framework for BIST Scheduling, Data patterns delivering and diagnosis in SoCs

Alfredo BENSO, Silvia CHIUSANO, Stefano DI CARLO,
Paolo PRINETTO, Fabio RICCIATO

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso duca degli Abruzzi 24 - I-10129, Torino, Italy
Email: {benso, chiusano, dicarlo, prinetto, ricciato}@polito.it
http://www.testgroup.polito.it

Maurizio SPADARI
LSI Logic
Agrate Brianza, Italy
Email: maurizio@lsil.com

Yervant ZORIAN
LogicVision
San Jose CA, USA
Email: zorian@lvision.com

Abstract[1]

This paper proposes HD²BIST, a complete hierarchical framework for BIST scheduling, data patterns delivering, and diagnosis of a complex system including embedded-cores with different test requirements as Full Scan cores, Partial Scan cores, or BIST-ready cores. The main goal of HD²BIST is to maximize and simplify the reuse of the built-in test architectures, giving the chip designer the highest flexibility in planning the overall SoC test strategy. HD²BIST defines a Test Access Method (TAM) able to provide a direct "virtual" access to each core of the system, and can be conceptually considered as a powerful complement to the P1500 standard, whose main target is to make the test interface of each core independent from the vendor.

[1] This work was partially supported by POLITECNICO DI TORINO under the project GIOVANI RICERCATORI 1999.

1. Introduction

In the last years, the significant growth of digital system applications such as digital communication systems, led to a strong competition in terms of quality and costs. With the increasing IC's complexity, the trend is to integrate in a single chip all the devices and functions of a system. This new philosophy leads to a radical change in the project flow.

The need of reducing time-to-design and development costs have driven to resort to reusability in both hardware and software designs. In system design, the availability of a big variety of pre-designed cores, provides designers the possibility of integrating in the same device, usually called System-On-Chip (SOC), a large number of different functional blocks and memories of various kind, size and access methods. Thus, reusability is nowadays considered as one of the dimensions of the design space, and *Design-for-Reusability* rules are systematically followed in most design centers. Since testing is worldwide recognized as one of the major components of electronic system costs, reusability should not be limited to hardware cores, but

extended to their entire test-related issues, including test structures and strategies [1]. One of the most commonly used techniques to reuse the test strategy is to provided cores with *Built-In-Self-Test* (BIST) structures, which greatly simplify the test execution and its reuse at different stages of the design cycle.

Cores, unlikely components on a circuit board, need to be tested after being embedded. This is due to the fact that components on a board are already manufactured and tested by the vendor. When dealing with embedded cores instead, the vendor can, at most, guarantee the goodness of the core description but is the chip integrator who should test the core after its integration in the SoC. Usually the chip integrator has not a good knowledge of the core structure, and has to rely on the test strategies provided by the core vendor, who, on the other hand, has to define test strategies, patterns, and protocols without knowing neither the target technology nor the end-system where the core will be integrated. This is a critical issue in the embedding process because the application in which the core is integrated may strongly influence the test strategy in terms of fault-coverage, power consumption, and silicon area. The test strategy becomes therefore a key point in the characterization of the core with equal importance on the core functional behavior.

Moreover, physical constraints often limit the possibility of testing concurrently all the embedded cores. This is due to the fact that BIST and test execution in general, typically results in a circuit activation rate higher than the one of normal operation mode. Parameter as power consumption and noise result significantly stressed, and an over-dissipation of power may seriously damage the device. This problem can be solved if a scheduling mechanism of the different test sessions of the cores is implemented and supported in the SoC.

Eventually, the SoC that is today designed integrating different embedded cores might become tomorrow a core to be used as an embedded entity in another SoC. Consequently it is necessary to define test architectures able to support a hierarchical design flow.

In conclusion, in designing a SoC, the chip integrator should:

- Provide full accessibility to all the embedded cores in order to test and diagnose possible faults;
- Implement a flexible test management strategy in order to overcome the limited freedom due to the test structures already implemented in the cores;
- Design the SoC using a vertical paradigm, thinking that the design might be reused in future as an embedded core itself.

Most of the approaches proposed so far rely on a single controller to perform device level BIST scheduling. [2] and [3] propose a centralized controller, implementing the scheduling of the BIST sessions through the activation of one session at a time. [4] introduces an IEEE 1149.1 based hierarchical test access architecture used to manage the BS logic of complex IC designs containing many embedded cores. In order to reduce the routing cost, [5] proposes a distributed architecture, where intermediate blocks link the units under test to a centralized controller, and are used to control and activate the test of a sub-set of BISTed blocks. In [6], the authors present HDBIST, a hierarchical and distributed approach to support BIST scheduling of BIST-ready embedded cores. Despite the novelty of the approaches, both the modularity and the flexibility of the architectures are still limited, and none of the proposed solutions support BIST as well as scan-based tests of full or partial-scan cores.

This paper proposes a Hierarchical-Distributed-Data BIST architecture (HD^{2}BIST), which supports the integration of embedded cores with different test requirements, as Full Scan cores, Partial Scan cores, or BIST-ready cores. HD^{2}BIST allows adding to the SoC design a high degree of reusability and flexibility in terms of:

- Test structure: the hardware inserted to manage the different test strategies of the embedded cores is customizable on a trade-off among routing, area, and test length.
- Scheduling: the HD2BIST structure allows to apply and/or activate and check the test procedures of each core of the system in any possible order, also resorting to complex scheduling control flow mechanisms as "wait" and conditional operations.
- Test Access Protocol: the approach defines a unified Test Access Method (TAM) to the different cores of the system, independent from their built-in test access protocols.
- Hierarchy: the HD2BIST is completely reusable during different phases of the product life cycle (horizontal reuse), and at different levels of integration (vertical reuse).

The main goal of the HD^{2}BIST architecture is to maximize and simplify the reuse of the built-in test architectures, giving the chip designer the highest flexibility in planning the overall SoC test strategy. HD^{2}BIST defines a Test Access Method (TAM) able to provide a direct "virtual" access to each core of the system, and can be conceptually considered as a powerful complement to the P1500 standard [7] [8] (Figure 1), whose main target is to make the test interface of each core independent from the vendor.

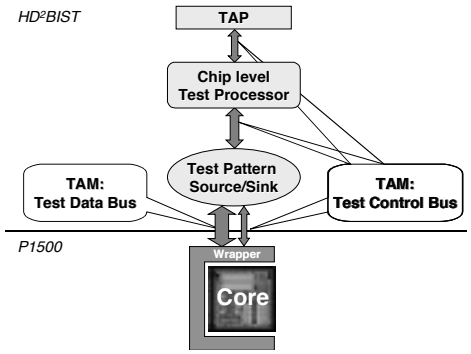

Figure 1 HD²BIST Basic Architecture

The paper is organized as follows: Section 2 presents the main features of the proposed approach, whereas Section 3 details its RT-level implementation. Section 4 presents some interesting experimental results obtained on two industrial case studies. Finally, Section 5 draws some conclusions.

2. Architecture overview

The key idea is to distribute test data to each core through a *Test Bus (TBUS)*. Each core uses the bus to gather the test data inputs and to send out the test data outputs. Each core is connected to the TBUS through ad-hoc interfaces called *Test Blocks* (TB).

The number of devices (i.e., cores) connected to the TBUS is not conceptually limited, and depends only on the project size. Data flowing on the bus are divided into:

- *Static signals*, which change only few times in a test session (usually at the beginning and at the end of the test). They are used to configure the test interfaces (TBs) and to start and stop the test execution of each core.
- *Dynamic signals*, which change many times in the test session and include test data patterns, diagnosis data, scan enable, etc…

Consequently, we logically split the TBUS into two different busses:

- The Test Control Bus (TCB), in charge of transmitting static signals;
- The Test Data Bus (TDB), in charge of transmitting dynamic signals and designed in order to support the typical test approaches used in core's testing like Full scan, Partial Scan, or BIST.

In the reminder of this paper we will mainly focus on the Test Data Bus, since the main functionality's of the Test Control Bus can be found in [5]. The next sections will briefly summarize the most innovative aspects of the HD²BIST architecture in terms of hierarchy, distributed test management, scheduling, and RT-level implementation.

2.1. Hierarchy

One of the most important test requirements highlighted in the introduction is the need to support a hierarchical approach in the definition of the test strategy.

Using the HD²BIST approach, it is possible to partition a complex system in a number of subsystems, each managed by a dedicated TBUS that allows to view each subsystem as a single TB, i.e., a single Unit Under Test (UUT). The obtained structure is a tree of test busses.

The management of a TBUS, as well as the connection between a higher and a lower level bus, are demanded to the so-called *Test Processor* (TP). The TPs allow to support a distributed approach in the execution of the system test. The general idea is that the TP performs all the operations on the lower bus needed to resolve the commands coming from the upper bus. In order to provide accessibility to each core from the highest hierarchical levels, a TP can be programmed to physically connect the upper bus with the lower one. In this way, the operations on a single core can be performed not only from the bus level to which the core belongs, but also from an upper hierarchical level. This approach is particularly useful to test full/partial scan cores, where, properly configuring the TP, the designer can create a direct path from the highest level to the core scan in/out pins and directly apply test patterns.

2.2. Scheduling description

The flexibility in the test scheduling has been implemented through a set of *test instructions* or commands issued by the Test Processor to the targeted Test Block. This mechanism can be used to activate the execution of the test of a BIST-ready core, or to set a path that an external ATE needs for applying scan-based test patterns to a full-scan core. The test session of the SoC can be therefore considered as a collection of *test programs*, each of them implemented as a sequence of commands. The order in which the test programs are executed can be chosen by the user, and not necessarily at design time. The available commands are:

1. *Start:* it starts the BIST procedures of a single block (or of a group of blocks) in the system.
2. *Configure*: it is used to configure the TBs of the cores connected to the TBUS. In particular, it is possible to:
   - configure the interconnections between the scan chains in the cores and the Test Data Bus;

- configure the interconnections between each block (core or BIST controller) and the Test Bus;
- set the blocks in one of its possible functional modes (test, transparent,…);
- configure, when available, the P1500 wrapper.

3. *Collect:* it collects the results of the BIST sessions.
4. *Wait:* in order to address possible power budget constraints, it stops the scheduling until the BIST procedure of one or more blocks is completed.
5. *Jump:* depending on the result of the BIST session of one or more blocks, it jumps to a certain label in the test program. This command adds flexibility in the test scheduling, allowing the designer to take decisions on the fly. It is particularly useful to avoid testing additional parts of an already revealed faulty component.
6. *SetEnv/UnsetEnv:* these two instructions are used to set an environment in the hierarchy. In particular, they can be sent in order to address, from the top-level, blocks belonging to different hierarchical levels. These two instructions are usually not parT of a test program, but can be very useful when applied directly from the outside for debugging and diagnosis purposes.

3. RT-level implementation

After introducing the HD^{2}BIST architecture at a logical level, we now detail how the test structures actually appear at the RT level. In particular, we will focus on the bus interface of the cores (i.e., the TBs) and on the implementation of the Test Bus (i.e., the TBUS).

3.1. The Test Blocks

The TB interfaces a core with the Test Bus. It must therefore provide a standard way to access cores implemented with different test strategies. The test blocks belonging to a TBUS can be of four types:

- *BISTed Core TB* (Figure 2.a): the TB interfaces a core that has an embedded BIST controller. It uses the Test Data Bus only in an eventual diagnosis phase and the TCB for receiving the start BIST and to communicate the results;
- *Scan TB* (Figure 2.b): it interfaces a full or partial-scan core;
- *BIST Controller TB* (Figure 2.c): it interfaces a BIST controller whose UUT is posed on the same hierarchical level (see next TB type);
- *UUT TB* (Figure 2.c): it interfaces the UUT related to a certain BIST controller connected on the same test bus (see previous TB type);

The configuration described in the last two points is useful in order to reuse the same BIST controller for more than one UUT, or to move the BIST controller away from its UUT. The idea is to connect the BIST Controller TB and the UUT TB through the TBUS and perform the test.

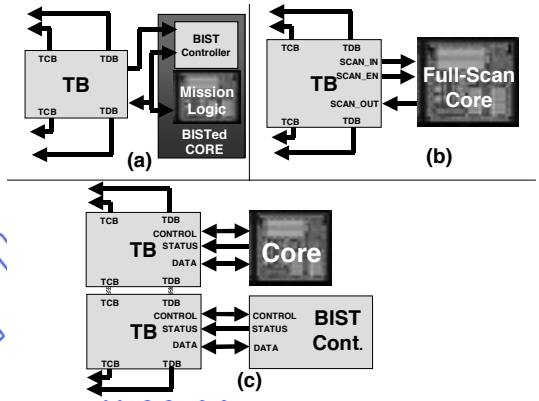

Figure 2 BISTed Core TB

Internally, the TBs can be considered as register files, where the registers can be used to configure the blocks, to set or reset a signal toward the core, to read a signal coming from the core, and to read the status of the block. There are three different classes of registers:

1. *Test Control Registers (TCR):* they are usually written by a TP to perform an operation on the core test logic. For example the TP can write a TCR to start the BIST of a BIST-ready core;
2. *Test Status Registers (TSR):* They are usually connected to a test status signal of the core and read by the TP to monitor the status of the core test;
3. *Test Mode Registers (TMR):* they are usually written by the TP to properly configure the block. Each *mode* (or configuration) implemented in the TMR corresponds to a different interconnection scheme between the Test Data Bus and the core I/Os. It is possible to define as many modes as necessary to cover all the test requirements.

The number of registers usually depends on the number of operations (TCR), modes (TMR) and statuses (TSR) that the core needs to be tested.

3.2. The Test Bus

Each Test Bus connects the TBs belonging to the same hierarchical level. The bus has been implemented as a ring bus, and the hierarchy is realized connecting different rings through ad-hoc TPs. No conceptual constraints are posed to choose a different bus shape.

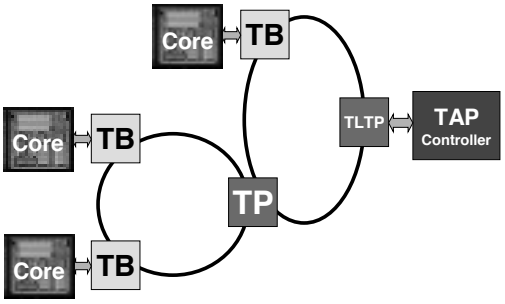

Figure 3: Test Bus structure example

As previously mentioned, the Test Bus is split into Test Control Bus and Test Data Bus.

3.2.1. The Test Control Bus

The Test Control Bus (TCB) connects a certain number of TBs. The TCB sees each TB as a register file that can be read or written. The communication protocol implemented on the TCB is Token-based, i.e., all the information exchanged are packed in a fixed length frame called '*token*'. Using a ring structure, each token is transmitted on a point-to-point connection between two TBs (TP). Each token received by a TB (TP) has to be, if necessary, forwarded to the next block in order to reach its correct destination. When a token comes back to the source, the TB (TP) must remove it from the bus. In this way, each block on the bus can read or write the registers of another block belonging to the same bus. Normally, each ring in the TCB is an isolated transmission domain. A token cannot therefore be transmitted between two different hierarchical levels. However, this operation might be necessary, for example during the diagnosis phase. To solve the problem the *SetEnv* and *UnSetEnv* instructions are used. As the name suggests, the *SetEnv* instruction sets an environment in the hierarchy. The blocks that do not belong to that environment will not react to the next tokens until they receive an *UnSetEnv* instruction. In particular, the *SetEnv* instruction can be sent only to a TP. All the blocks crossed by the *SetEnv* instruction fall in an idle state and they do not react to the following tokens. The destination TP, after receiving a *SetEnv* instruction, starts to forward all the following tokens to the lower ring. With this mechanism, the transmitted tokens are processed only in the lower ring. Continuing to send *SetEnv* instructions, we can reach any desired block in the hierarchy. To restore the normal situation it is only necessary to send a broadcast *UnSetEnv*. The instruction flows along the path and all the blocks crossed by the *UnSetEnv* instruction restarts to recognize and process the tokens.

3.3. The Test Data Bus

The Test Data Bus is completely dedicated to dynamic data. It is therefore used to carry:

- Test data patterns for a full-scan testing of any core from the highest hierarchical level;
- Test data patterns from a BIST controller to a core;
- The scan chains content of core for diagnosis purposes.

Using the TCB, the Test Data Bus can be dynamically configured in order to permit the accessibility to all the blocks in the hierarchy, and to allow a flexible allocation of the bus lines. The designer can choose to have different types of configurations for each block. In general, he can decide how to connect the lines of the data bus with the I/Os of the core. Locally to a block, each line of the TDB can be in two states:

1. *Bypass*: the received data are forwarded to the next block;
2. *Connect*: the received data are forwarded to a core input. On the same line of the TDB, an output of the core is forwarded to the next block.

In other words, if the line is in Connect state, the ring line is opened and an input and an output of the core are connected to the bus. The idea is to use the incoming bus lines part to force the core inputs and the outgoing part to read the core outputs. The TDB can be shared:

- In width: some lines of the TDB are used for testing a core, some others to simultaneously test another core;
- In time: different blocks use the same lines in different moments. For example, it is possible to use some lines to connect a BIST controller with an UUT and then, after finishing the test of the UUT, the same lines to connect the same BIST controller with another UUT;

The transmission domain over the TDB can be local to the bus or expanded to more rings. In particular, in a hierarchical architecture, the Test Processors have to connect the upper TDB with the lower TDB. Properly configuring all the Test Processors and the Test Blocks, the designer can create a data path among the blocks in the hierarchy.

3.3.1. Using the Test Data Bus for full-scan testing

One of the most useful applications of the TDB is the support of full-scan cores testing (Figure 4.a). The idea is to create a path to the core from the top level TAP controller, from where the test patterns are applied. A preliminary configuration phase is needed to set the path. Data patterns can then flow from outside to the scan-in of the core and from the core scan-out to outside. Usually the number of lines allocated for this purpose is equal to the number of the scan chains in the UUT plus an additional line for the scan enable signal. If, due to the allocation policy or the TDB width, this amount of lines is not available, the designer can use a *Scan Chain Router* to

reduce the number of scan chains. The 'in width' sharing of the bus allows the concurrent test of two or more full-scan cores. The external ATE possibly applying the patterns only has to consider the initial and final latency due to the path configured along the hierarchy.

3.3.2. Using the Test Data Bus to connect a BIST controller to a core

The TDB can be used to connect a BIST controller to its UUTs (Figure 4.b). After the usual data bus configuration phase, the patterns generated by the BIST controller can flow on the TDB towards the UUT. As usual, dynamic signals are transmitted on the TDB, whereas the static ones are exchanged on the TCB. Due to the configuration features of the data bus, a single BIST controller can be used to test different UUT. Supposing to have two cores of the same type on the same ring, instead of having two separate and identical BIST controllers, it is possible to put on the ring a single BIST controller and to connect it first to one core and then to the other one. The idea is to conceptually divide the BIST controller in the Test Patterns Generator (TPG) part and in the Output Data Evaluator (ODE) part. Supposing that the two parts have two enable signals, the two corresponding TBs can pilot the generation of the test patterns, the evaluation of the results, and the UUT scan enable to synchronize the flow of bits.

In addition, since the BIST controller was originally designed to be in direct contact with the UUT, the interface between the two blocks may have more signals than the ones available on the TDB. In this case the TBs are able to compact the signals, transmit them on the Test Bus, and de-compress them before applying them to the UUT. During the compression/decompression operations, the BIST controller and the UUT scan-in are disabled through the enable signals. The compression/ decompression mechanism obviously does not allow at-speed testing and may introduce a considerable test time overhead.

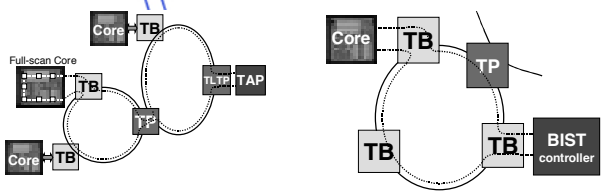

Figure 4: Using of TDB to test different types of cores

3.3.3. Using the Test Data Bus for diagnosis of BISTed blocks

At the end of test session, the BISTed cores often release not only a boolean result of the test, but in case of failure they allow to unload the scan chains content for diagnosis purposes. In this case, the TDB can be used, similarly to the full-scan cores, to extract the chain contents.

3.3.4. Using the Test Data Bus to test glue logic

The chip integrator often adds some user-defined logic (glue logic) to connect the embedded cores. This logic is usually not localized and can't be wrapped or considered as a core. This logic is usually tested using a full-scan approach and, since it is not localized, the use of a TDB to deliver test patterns seems to be ineffective. Nevertheless, due to the hierarchical design approach, the glue logic can be split in different parts each referring to a certain bus domain. From each master TP, some scan chains go through the glue logic referred to that bus. This scan chains can be considered as pseudo TDB lines and connected to the upper TDB. This mechanism allows testing the glue logic almost as a standard scan-chain of a full-scan core (Figure 5).

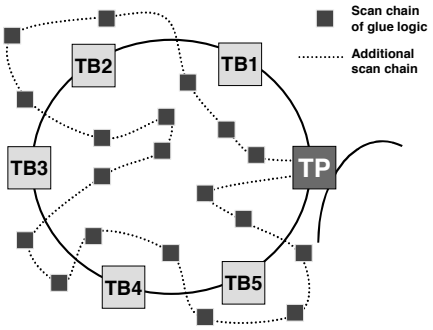

Figure 5: Pseudo TDB lines for testing the glue logic

3.4. Test Access Port Interface

Interfacing the H^{2}DBIST with a Boundary Scan TAP interface is necessary to control the TLTP using a standard interface and to simplify the use of an external ATE to test full or partial scan cores. The idea is to control the Test Bus of the top-level ring from the TAP controller. Directly controlling the Test Bus, we conceptually substitute the TLTP with the external ATE. The TLTP contains a Command Register connected to Test Data In (TDI) and Test Data Out (TDO) pins of the TAP, which can be used to set it in one of the following functional modes:

1. Scheduling Mode: this instruction starts the execution of one of the test programs implemented in the TLTP. In this mode, the Test Bus is not accessible from outside. At the end of the test, the results are loaded into the Command Register and can be scanned out from the TDO pin.
2. Full Mode: the TCB is controlled using the TDI and TDO pins. The instruction register is directly used to load and unload the tokens to be sent or received

on the TCB. The TDB is controlled using the scan-in and scan-out lines.

3. Data Mode: only the TDB is accessible from the outside using the TDI and TDO pin. The TCB cannot be accessed. Before using this mode, all the TBs must be configured using the Full Mode.

4. Experimental results

We implemented and synthesized the HD²BIST architecture (using Synopsys™ and the G11 LSI Logic™ library) on a complex test case with two hierarchical levels. The test case has been chosen to demonstrate the effectiveness of the proposed approach in terms of *reusability* of the test structures in a hierarchical architecture, *test patterns delivering* for full-scan cores, and *scheduling* of the BIST-ready blocks. The following section analyze the example in details, providing experimental results in terms of area overhead.

4.1. DacTOPplus Architecture

The goal of the presented test case, named *DacTOPplus*, is to prove the flexibility of the HD²BIST architecture in a design embedding full-scan modules and BIST-ready blocks. DacTOPplus is composed of four identical macro-cores (*DacTOP*) and two BISTed RAMs (8192x8) (Figure 6).

- Each DacTOP macro is composed of four sub-modules:
- One transmission macro-cell NDS_TX;
- One receiving macro-cell NDS_RX.
- Two identical NDS macro-cells;

The NDS_RX, NDS_TX macro-cells are full-scan modules with seven scan chains each, whereas the two NDS modules have been considered as glue-logic, and all their flip-flops are connected through a single scan chain. Table 1 reports the area occupied by the test case.

CORE	Equivalent Gates
NDS	99,801
NDS_RX	102,688
NDS_TX	102,802
DacTOP	430,356
BISTed RAM	163,694
DacTOPplus	2,048,814

Table 1: DacTOPplus dimensions in Synopsys equivalent gates

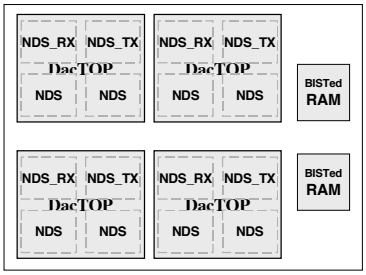

Figure 6: DacTOPplus scheme

In the following, we will focus first on the test structure implemented in the DacTOP macros, and then on the complete test case including the four DacTOP macros as well as the two BISTed memory modules.

4.2. DacTOP Test Structure

The test structure implemented in each DacTOP macro is composed of a single HD²BIST chain controlled by a Test Processor (TP). The NDS_TX and the NDS_RX macros, packaged by a P1500-like wrapper, are controlled by two Test Blocks (TB), whereas the NDS modules are treated as glue logic, and therefore directly controlled (or tested) by the TP. The HD²BIST structure inserted in each DacTOP macro is therefore composed of (Figure 7):

- One Test Bus split into:
  - One Test Control Bus 1-bit wide.
  - One Test Data Bus. Since each module has seven scan chains and the Test Data Bus has to transmit the Scan Enable and the Reset signals driven by the ATPG patterns, we decided to drive all the scan-chain in parallel and we therefore sized the Test Data Bus width to 10.
- Two Test Blocks (TBs). Each Test Block has:
  - One Test Mode Register which can be set in two functional modes:
    - Bypass mode, where the Test Data Bus is in bypass mode;
    - Connect mode, where the Test Data Bus is connected to the scan chains and the scan patterns can be delivered to the module.
  - One Test Control Register used to send commands to the wrappers. In particular, it is connected to the signals of the wrapper used to put the core in Normal or Test mode, in internal or external test, and to allow the shifting of the test results.
- One Test Processor (TP) with three test programs PROG[1-3], used to connect the NDS_RX, NDS_TX, and the two NDS macros respectively, to the *Test Data Bus*. Each program sets a different target block in

Connect mode and the others in Bypass mode. The Test Processor has:

- One Test Mode Register with three functional modes:
  - Bypass mode where the upper Test Data Bus controlled by the TP is in bypass mode (see Section 3.3);
  - ExtCon mode, where the lower Test Data Bus is connected to the TLTP input signals;
  - Glue mode, where the TP creates a direct path from the outside to the scan-chain connecting the glue logic;
- One *Test Control Register* used to start the three different test programs PROG[1-3].
- One *Test Status Register* set to "1" when the commands written on the Test Commands Register is completed and the scan patterns can be delivered to the target block. The register is set to "0" when the program is running.

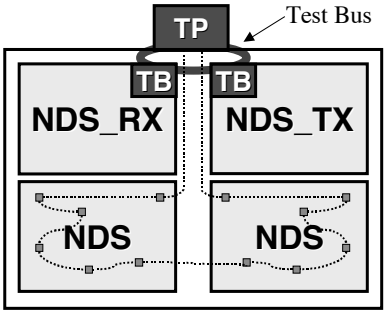

Figure 7 : DacTOP HD²BIST scheeme

Table 2 and Table 3 report the area obtained synthesizing the DacTOP test case and the HD²BIST architecture using the G11 LSI Logic library.

CORE	Equivalent Gates
Glue Logic	199,602
Wrapped NDS_RX	112,049
Wrapped NDS_TX	110,976
DacTOP with wrappers	447,891

Table 2: DacTOP area with wrapped modules

CORE	Equivalent Gates
TB of NDS_RX	3,695
TB of NDS_TX	3,701
TP	6,145
HD²BISTed DacTOP	461,434

Table 3: HD²BISTed DacTOP area occupation

The area overhead of the HD²BIST structure w.r.t. the original DacTOP area is the 7.03%, whereas the overhead w.r.t. the DacTOP area including the wrappers is 2.97%. We included the wrapped version of the DacTOP Plus since we consider the wrappers a test requirement independent from the HD²BIST structure.

4.3. DacTOPplus Test Structure

The test structure inserted in the DacTOPplus test case is composed of one HD²BIST chain at the top level, and one HD²BIST chain for each DacTOP module. No modifications are necessary to use the test architecture implemented in each DacTOP macro in order to reuse it at the top-level. The top level chain is built of the following blocks (Figure 8):

- One *Test Bus* split into:
  - One Test Control Bus one line wide.
  - One Test Data Bus ten lines wide (each DacTOP module needs ten lines, whereas the BISTed RAMs do not need any data line).
- One *Test Block* for each RAM including:
  - One Test Control Register used to send the START_BIST command to the RAM BIST controllers;
  - One Test Status Register, two bit wide, used to read the BIST_END and the BIST_OK signals from the RAM BIST controllers.
- Four *Test Processors*, one for each DacTOP macro, as described in the previous section.
- One *Top Level Test Processor* with a TAP interface. In the *Test Processor* we implemented thirteen different test programs, which can be executed in any desired order:
  - PROG[1]: it starts the BIST of the two RAMs, waits for the BIST to end, and reads the test results;
  - PROG[2-4]: they start respectively PROG[1-3] of the first DacTOP and they wait for their end. They then connect the *Scan In* of the TAP interface with the first *DacTOP* in order to scan out the test results;
  - PROG[5-7]: the same as PROG[2-4] but with the second DacTOP module;
  - PROG[8-10]: the same as PROG[2-4] but with the third DacTOP module;
  - PROG[11-13]: the same as PROG[2-4] but with the fourth DacTOP module;

The Test Processor has:

- One *Control Register* used to start the thirteen programs. It is loaded through the TDI of the TAP interface with the number corresponding to the specific program to be executed. The end of the program is notified by TPRE signal of the TAP interface;
- One *Test Status Register* which contains, at the end of PROG1, the BIST_OK flag of the first RAM and the BIST_OK flag of the second one. At the end of PROG1 the *Test Status Register* can be read trough the TDO of the TAP interface.

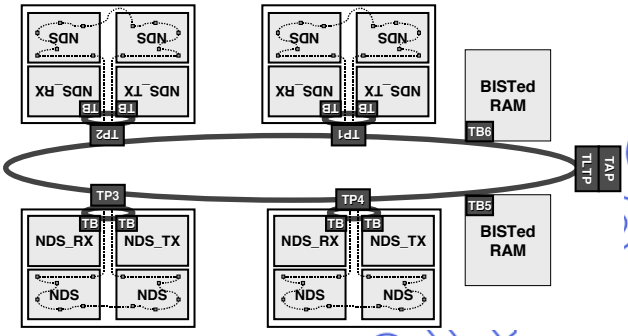

Figure 8: DacTOPplus with HD²BIST

Table 4 reports the area obtained synthesizing the DacTOPplus using the G11 LSI Logic library.

CORE	Equivalent Gates
HD²BISTed DacTOP	461,434
TB of RAM	2,956
TP_TAP	5,958
HD²BISTed DacTOPplus	2,184,997

Table 4 : Area result of DacTOPplus

The area overhead of the HD²BIST structure w.r.t. the original DacTOPplus area is 6.61%, whereas the overhead w.r.t. the DacTOPplus area including the wrappers is 3.06%.

4.4. Running a test program

To show how it is possible to actually exploit the HD²BIST architecture to run the system test, we detail two different test programs that target the BISTed RAMs and DacTOP NDS_RX module. Each program can be launched loading the corresponding code in the Instruction register of the TAP interface.

Table 5 presents PROG[1], which activates the test of the two BISTed RAMs. The program activates the BIST procedures of the two BISTed RAMs by writing a proper value in the Test Control Registers of their Test Blocks. After activating the test procedures, the TLTP starts polling the two Test Blocks waiting for the end of the BIST procedure. Test programs allows a very flexible implementation of any test scheduling: PROG[1] executes the BIST of the two memories in parallel, but, by simply exchanging instruction #2 and #3, it is possible to execute a serial test of the two memories.

Test primitive
`1. Start TB5, RAMBIST` `/* The TLTP sends a WRITE token to TB5 setting its Test Control Register to '1'. This action activates the BIST controller of the first BISTed RAM and the BIST procedure begins*/`
`2. Start TB6, RAMBIST` `/* The same action is performed on TB6 in order to start the BIST procedure of the second BISTed RAM */`
`3. Wait TB5, RAMBIST` `/* The TLTP starts polling TB5 reading its Test Status Register waiting for the end of the BIST procedure. When the test ends, the test program execution jumps to the next instruction */`
`4. Wait TB6,RAMBIST` `/* As for the previous RAM, the TLTP starts polling TB6 waiting for the end of the BIST procedure of the second RAM. When the test ends, the test program execution jumps to the next instruction */`
`5. END` `/* The TLTP sets one of its output signals (TPRE) to '1' in order to inform that the test program is finished and the test results can be scanned out from the TDO pin of the TAP interface */`

Table 5 : Test program PROG[1] of the TLTP

The second example program is PROG[2] (Table 6), which creates a path on the Test Data Bus that directly connects the top-level TLTP to the NDS_RX module of the first DacTOP. After properly configuring the top-level chain, the Test Data Bus of the first DacTOP module is connected to the top-level Test Data Bus in order to allow the TLTP in order to start the execution of PROG[1] in the first DacTOP (Table 7). PROG[1] creates a path from the scan chains of NDS_RX to the Test Data Bus lines. As soon as PROG[1] is completed, the TLTP can assert the TPRE output signals to '1' in order to inform that the configuration phase is finished, a path has been set from the top-level to the addressed block in the hierarchy, and the test patterns can be applied to the TDI signal of the TAP interface.

Test primitive
1. CONF All, BypassMode /* First of all, all the TP and TB of the top-level chain are set in Bypass mode using a broadcast Conf token */
2. CONF TP1, ConnectMode /* The TP of the first DacTOP macro is set in Connect mode, in order to connect its Test Data Bus to the top level Test Data Bus */
3. Start TP1, PROG[1] /* The TLTP writes the Test Control Register of TP1 in order to start the execution of PROG[1] of the DacTOP macro (See Table 7) */
4. Wait TP1, PROG[1] /* At this point, the TLTP waits for the program to end by polling the Test Status Register of TP1 */
5. END /* The TLTP sets the TPRE output signals to '1' in order to inform that the configuration phase is finished, a path has been set from the top-level to the addressed block in the hierarchy, and the test patterns can be applied to the TDI signal of the TAP interface */

Table 6 : Test program PROG[2] of the TLTP

Test primitive
1. CONF All, BypassMode /* All the TBs belonging to the first DacTOP macro are configured in Bypass mode */
2. CONF TB1_1, ConnectMode /* The first TB of the chain is set in Connect mode, in order to connect the scan chains of NDS_RX to the Test Data Bus lines */
3. START All, Isolated /* The wrappers of the NDS_TX and NDS_RX macros are set in Isolated mode. This action is necessary to avoid the outputs of the module under test (NDS_RX) to apply forbidden patterns to the boundaries of the other modules */
4. START TB1_1, InternalTest /* The wrapper of the module under test is set in Internal mode */
5. END /* TP1 sets its Test Status Register to '1' in order to notify the termination of PROG[1] */

Table 7 : Test program PROG[1] of the DacTOP macro

In a similar way it is possible to create the test program to test glue logic and the NDS_TX module.

5. Conclusions

This paper proposed HD²BIST, a complete hierarchical framework to support the definition of the scheduling strategies and data patterns delivering mechanisms of the embedded cores of a complex system. The main goal of the HD²BIST architecture is to maximize and simplify the reuse of the built-in test architectures giving the chip designer the highest flexibility in planning the overall SoC test strategy. HD²BIST defines a TAM able to provide a direct "virtual" access to each core of the system, and can be conceptually considered on a higher level w.r.t. the P1500 standard, whose main target is to make the test interface of each core independent from the vendor. We presented a complex case study where we demonstrated the effectiveness of the approach in terms of complexity and area overhead.

6. References

[1] Varma, P. Bhaita : A Structured Test Re-Use Methodology for Core-Based System Chips. Proceedings IEEE International Test Conference, pp. 294-302. IEEE Computer Society Press, 1998.

[2] F. Beenker, R. Dekker, R. Stans, Implementing MACRO Test in Silicon Compiler Design, IEEE Design & Test of Computers, April 1990, pp. 41-51

[3] O.F. Haberl, T. Kropf, HIST, A Methodology for the Automatic Insertion of a Hierarchical Self test, Proc. IEEE International Test Conference (ITC'92), 1992, pp. 732-741

[4] L. Whetsel, An IEEE 1149.1 Based Test Access Architecture For ICs With Embedded Cores, Proc. IEEE International Test Conference (ITC'97), November 1997, Washington (D.C.), pp. 69-78

[5] Y. Zorian, A distributed BIST Control Scheme for complex VLSI devices, Proc. 11th IEEE VLSI Test Symposium (VTS'93), April 1993, pp. 4-9

[6] A. Benso, S. Cataldo, S. Chiusano, P. Prinetto, Y. Zorian, HD-BIST: a Hierarchical Framework for BIST Scheduling and Diagnosis in SoCs, Proc. IEEE International Test Conference (ITC'99), Atlantic City (NJ), September 1999, pp. 993-1000

[7] Adham : P1500-CTL: Towards a Standard Core Test Language. Proceedings IEEE VLSI Test Symposium, IEEE Computer Society Press., Apr. 1999

[8] Zorian Y.: Preliminary Outline of IEEE P1500 Scalable Architecture for Testing Embedded Cores. Proceedings IEEE VLSI Test Symposium, IEEE Computer Society Press, Apr. 1999