System Level Testing via TLM 2.0 Debug Transport Interface

# System Level Testing via TLM 2.0 Debug Transport Interface

Authors: Di Carlo S, Hatami N., Prinetto P., Savino A.,

Published in the Proceedings of the IEEE 24th International Symposium on Defect and Fault Tolerance in VLSI Systems (DFTS), 7-9 Oct. 2009, Chicago (IL), USA.



**DOI: 10.1109/DFT.2009.46**

# System Level Testing via TLM 2.0 Debug Transport Interface

Stefano DI CARLO, Nadereh HATAMI, Paolo PRINETTO, Alessandro SAVINO
*Politecnico di Torino*
*Dipartimento di Automatica e Informatica*
*Torino, Italy*
*{stefano.dicarlo, nadereh.hatami, paolo.prinetto, alessandro.savino}@polito.it*

***Abstract***

*With the rapid increase in the complexity of digital circuits, the design abstraction level has to grow to face the new needs of system designers in the early phases of the design process. Along with this evolution, testing and test facilities should be improved in the early stages of the design to provide the architecture with functional test facilities to be later synthesized to hardware and/or software testing infrastructures according to designer's requirements. In the case of hardware, these test infrastructures could be translated, for instance, into online testing facilities at lower levels of abstraction, from which automatic synthesis tools are available. TLM 2.0 is a new OSCI standard widely used to describe systems at high abstraction level. Starting from the increasing use of TLM in hardware design industry, the paper aims at providing a mechanism to fill the gap between the design abstraction level and the level in which testing methodologies are applied. To do the job, the TLM 2.0 "debug transport interface" is used and methods are introduced to synthesize it into known test access methods at RTL.*

## 1. Introduction

With the new advances in technology and the emergence of new human needs, the complexity of digital circuits is increasing every day [16]. Today's system-on-chips (SoC) are built out of a very large number of diverse IP structures and cores. On the other hand, safety and reliability of such systems should be guaranteed by proper testing mechanisms [3]. System level modeling is an answer to the increasing need of dealing with complexity in hardware design industry. It represents a viable solution to facilitate design and debug of complex systems starting at their early design phases. By the new emergence of Transaction Level Modeling (TLM) standards, this methodology is being vastly used by the Electronic System Level (ESL) design industry [4].

The TLM standard, introduced by the Open SystemC Initiative (OSCI) [2], is a transaction-based approach to describe systems at the early stages of the design. It is able to speed up the simulation, and to facilitate debug for huge electronic systems [1, 8, 9]. Two releases of the TLM standard are currently available. TLM 1.0 uses basic channel structures, such as FIFOs, to describe high-level communications among functional units. TLM 2.0 provides additional interoperability by using generic packages for transactions. It also increases the simulation speed and abstraction level by taking advantage of proper interfaces for communications.

While TLM is proving to be a very valuable medium to increase the design flow efficiency by moving the overall design problem to higher abstraction levels, testing solutions and methodologies still remain confined to lower abstraction levels. This may result in completely high-level designed and simulated systems unable to fit specific test end reliability requirements when translated/synthesized into lower abstraction levels (e.g., RTL). The communication-centric view of TLM provides an abstraction that is well-suited to model test infrastructures involving the exchange of significant amounts of data. In addition, performance modeling in TLMs tries to accurately capture the concurrency in a system, which is easily adapted to model the concurrency in testing. [5]

A few publications considered the test problem at TLM level. [4] proposes a plug and test design methodology based on the insertion of testing capabilities at the transaction level by means of testable TLM primitives. While it represents a first tentative of moving test concepts at TLM level, the paper deals with basic FIFO communication channels defined in TLM-1.0 standard, only. [5] shows how TLM can be used to efficiently evaluate Design for Testability decisions in the early design steps, and how to evaluate test scheduling and resource partitioning during test planning. Even if these publications have the benefit of

starting to address the problem of testing at TLM level, the results are still limited and a strong investment to address this problem in a more general way is required.

This paper proposes a design methodology able to consider the test of communication interfaces at high level of abstraction. The introduced method is applicable to both hardware and software communication portions of the design to be later on synthesized into test infrastructures at register transfer level in case of hardware. We focus on TLM 2.0 as a high level modeling environment, and take advantage of the Debug Transaction Interface (DTI), originally introduced to debug the system at TLM 2.0 level.

The case study that demonstrates the presented design methodology is an encryption/decryption system using Advanced Encryption Standard (AES) [10] for encoding data. The method is first applied at the high level of abstraction in which hardware/software partitioning is not done yet. Then it is shown that even after partitioning, the method is still applicable to both hardware and software communication interfaces. At last, we have translated the designed system into an RTL description to show the synthesizability of the proposed method from TLM to RTL.

The organization of the paper is as follows: section 2 gives a brief overview of system level design flow, which usually starts from high level description of the system in TLM and ends to the netlist and the software related to the designed hardware. Section 3 includes a short introduction on TLM 2.0 and in particular its Debug Transaction Interface (DTI). The methodology of testing at transaction level is introduced in detail in section 4. A case study is discussed in section 5 to apply the introduced test infrastructures at TLM level to a practical model. Section 6 demonstrates the experimental results obtained from synthesizing the introduced model in case study to RTL whereas section 7 contains the conclusion.

## 2. System Level Design Flow

A short overview of a typical system level design flow is essential in order to understand the motivation behind the need of considering test concepts in the early steps of the design. Figure 1 shows a typical design flow starting from the definition of system requirements.

The requirements are then used to develop the system architecture model which can be later used to extract the TLM model of the design. The TLM model is then partitioned into the hardware and the software portion. From this point, the hardware and software design teams can start working in parallel to develop a system that meets the original requirements.

Each step of the design flow proposed in Figure 1 requires taking decisions in order to make the design more complete. Moving from each level to the next one, a Validation/Verification (V&V) campaign is required to understand whether the current design is compatible with the system requirements, and represents the best and optimized solution. Failing in a V&V step means that at least one of the decisions made in previous levels was not correct. This may lead to the need of several iterations in the design flow (see Figure 2).

Figure 2 clearly highlights that considering test issues late in the design flow (e.g., at the RTL level), may require long backtracks leading to a drastically increasing design cost and complexity. It is therefore mandatory to consider test related decisions starting from the system specifications instead of postponing them to later stages of the design flow.

## 3. TLM 2.0 Standard

Transaction level modeling [1, 8, 9] is a transaction-based modeling approach originally based on high-level programming languages such as C++ and SystemC [13]. It enables system level design and simulation of large hardware/software systems, for which RTL simulation would require an unacceptable amount of time [5]. TLM emphasizes on separating communications from computations within a system. In the TLM notion, computation units are modeled as modules with a set of concurrent processes that calculate and represent their behavior. These modules communicate in the form of transactions through abstract channels [1].

OSCI introduced two standards of TLM based on C++ class libraries written in SystemC. The first standard of TLM (TLM-1.0) uses the concept of channels and interfaces with focus on blocking, non-blocking, unidirectional, and bidirectional transfers [7]. As the abstraction level increases, this standard becomes less applicable and not fast enough for describing models at higher levels [9]. In addition, with the lack of model interoperability in the TLM-1.0 standard, another problem would be how producing IP cores to be used by a common customer.

TLM-2.0 is the new OSCI standard and it consists of core interfaces from TLM-1.0 together with the blocking and non-blocking transport interfaces, the direct memory interface (DMI), the Debug Transport Interface (DTI), the write interface, and the analysis interface [6]. It enables SystemC model interoperability and reuse, and provides an essential framework for architecture analysis, software development, performance analysis, and hardware verification [8]. It also defines a set of interfaces that should be thought of as low-level software programming mechanisms for implementing transaction level models. Each interface is suitable to work with TLM-2.0 designs characterized by different timing details.

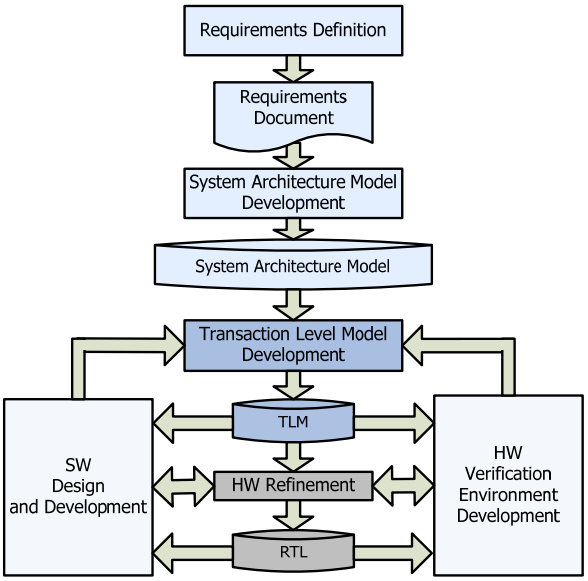


**Figure 1.TLM-Based design flow [12]**

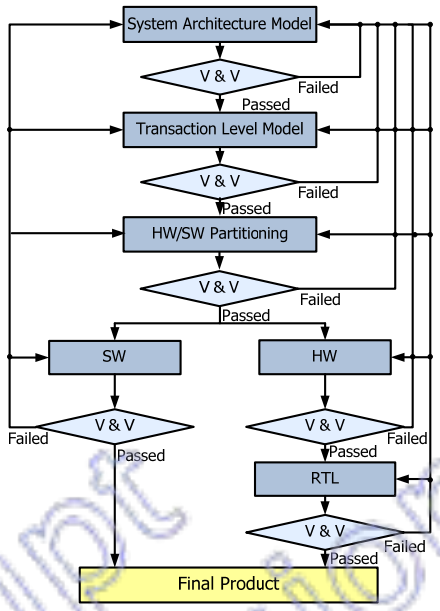


**Figure 2.System level design flowchart**

The transport interfaces are the primary interfaces used to transport transactions between initiators, targets and interconnect components [6]. The Direct Memory Interface is able to bypass the normal path of multiple transport interface calls from initiator through interconnect components to target to offer a large potential increase in simulation speed for memory access between initiator and target. The Debug Transport Interface provides a means to read and write to storage in a target, over the same forward path from initiator to target as is used by the transport interface, but without any of the delays, waits, event notifications or side effects associated with a regular transaction. Because this interface follows the same path as the transport interface, the implementation of the debug transport interface can perform the same address translation as for regular transactions. [6]

The Debug Transport Interface was originally designed to provide the designer with a debugging tool at the high level of abstraction. The idea proposed in this paper is to take advantage of this interface not as a debugging facility, but as a test structure that can be efficiently translated/synthesized into known RTL level test facilities. In other words, we use this interface as a facility providing the designer with the ability of modeling the required test infrastructure structure at high level of abstraction.

## 4. Testing at transaction level using the Debug Transport Interface

This section aims at discussing how TLM-2.0 specific features can be exploited to model test infrastructures, and to consider test related problems at the very early stages of the design flow, when the system is not yet partitioned into hardware and software parts. This early decision making may prevent the costs related to design backtracks that may arise when test related decisions are directly performed in the late steps of the design flow (see Section 2). The challenge here is to allow the designer to easily model testing capabilities in the design at the very early stages of the design process, regardless of the fact that specific parts of the systems will be then mapped into hardware or software components. Design for Testability (DfT) at high level may provide a big picture of the design, and facilitate high-level design, partitioning, and synthesis.

It is clear that at this very high level of abstraction, typical structural-based DfT approaches targeting structural fault models cannot be applied. Actually, the concept of hardware components still has to be defined at this level. Hence, the only possible and reasonable alternative to introduce test concepts at Transaction level is resorting to pure functional testing [4]. Moreover, besides providing a specific implementation of a test architecture, the idea here is to provide the designer with a very flexible high level facility that can be exploited to describe different types of test approaches.

Test and testability requirements include those for computational modules as well as those for the communication mechanisms. While computational modules implemented in hardware are mapped into IP cores or even more complex circuits that usually already include testability features, test methods for the communication mechanisms still remain a critical point.

This paper therefore focuses on modeling at TLM level test infrastructures for communication mechanisms. In particular, we will exploit the *Debug Transport Interface* (DTI) introduced by TLM-2.0 to model test methods for communication interfaces connecting initiators and targets together via sockets. As DTI uses the same path as the transport interface while skipping all the delays related to the normal path of communication,

it can be used as an additional communication channel for testing. The *transport_dbg* function related to the DTI can be used to model the test method to be applied at lower level for the test.

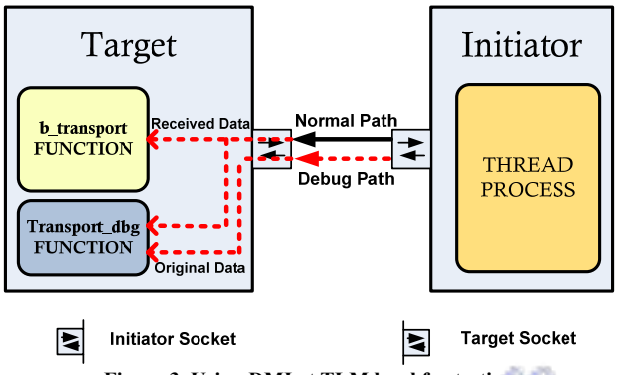


**Figure 3. Using DMI at TLM level for testing**

Figure 3 shows the main architecture of the proposed high level test schema. The normal path, which can be a blocking or a non-blocking transport interface, transmits data from the initiator to the target through sockets while the DTI is exploited to send test related information. The transport_dbg function uses both the received data and the test data to check whether the channel is working properly, and therefore to implement the desired test algorithm for the channel.

Several testing approaches for communication channels [17, 18, 19] can be efficiently modeled by resorting to the schema proposed in Figure 3. Besides the test patterns exchanged to actually perform the test, which are strictly related to the low level implementation, the main point to highlight here is that by carefully choosing the way the DTI is used, we can model high level test facilities that can be then easily translated into RTL and real implementations in the late steps of the design. Depending on the way the transport_dbg function is invoked we can distinguish among two main test approaches:

Concurrent test: in a concurrent test approach, each time a transaction is issued, the transport_dbg function should be used to check whether the transaction was correctly performed or not. In this case, the DTI can be efficiently exploited to transp redundancy information ranging from error-detection/correction codes (e.g., parity bits or SEC/DED codes) to a complete duplication of the transmitted data flow;

Off-line test: in an off-line test approach, a set of transactions is issued on the channel to transmit a set of data (test patterns) designed to detect specific faults on the target implementation. In this situation the system needs to enter a so-called test mode in which its normal behavior is interrupted and test activities are then performed. In this context, the DTI may assume several roles. As storing test data for comparison into the target may represent a high cost, the DTI may be exploited to send copies of the data sent on the normal path. In this case, as the operation does not require to be performed concurrently with the normal behavior of the system, a channel with reduced bandwidth can be exploited to perform this job.

Given these two main templates for using the DTI, this interface can be then exploited in a very flexible way to describe specific test algorithms at high-level and more importantly to evaluate their impact on the system.

In the following section we will exploit the application of this test modeling approach to a case study. The main ide s to provide examples of how test schema can be actually modeled, and how the high level model can be translated during the design flow down to the final RTL implementation.

## 5. Case Study

To show a real application of the idea introduced in Section 4, an encryption/decryption sub-system based on the Advance Encryption Standard (AES) [14] is considered here and implemented at three levels of abstraction.

### *High level of abstraction before HW/SW partitioning:*

The highest abstraction level (Figure 4) includes the behavioral description of the system prior to HW/SW partitioning. In this case, the system contains two processing elements (cores), described in TLM-2.0 loosely timed style, for encryption and decryption respectively, and an I/O core. The I/O core sends data to the AES core to encrypt information, or sends a cipher text to the decryption core to decrypt the received data. In this architecture, the DTI is used to build a test infrastructure for the communication between both the AES core and the I/O core, and the Decryption core and the I/O core.

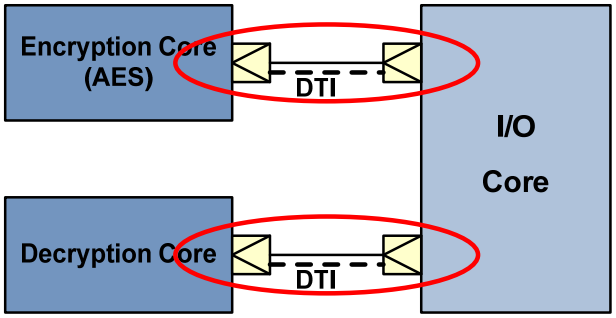


**Figure 4. System architecture at high level of abstraction**

At this level, we can decide whether to apply online or offline test infrastructure to our high level system architecture.

We consider the two different scenarios proposed in Section 4:

*A. Offline testing:* In this scenario, the system has two operation modes. In "test" mode, the initiator sends a stream of bytes in parallel to the target. In the meanwhile, it starts transferring the same data in serial, byte by byte, to the target through the DTI. At the end of the transfer, the transport_dbg function compares the original and received data to see if the channel is working properly. After the proper functioning of the communication channel has been checked by the DTI, the system can enter its "normal" mode. Figure 5 shows the implementation of the debug function in this case. As it can be seen, the debug function receives the serial data and store it in a temporary variable until all the data is got. At this stage, the comparison is performed to guarantee the correctness of the communication path.

```
1  unsigned int AES::transport_dbg (tlm::tlm_generic_payload& trans){
2    char temp = (char) trans.get_data_ptr();
3    if (temp != '\0')      {
4            orig_data[counter] = temp;
5            counter ++;
6    }
7    else {
8            orig_data[counter] = temp;
9            bool flag = true;
10           for (int i=0; i<=counter; i++)
11           if (_INPUT[i] != orig_data[i])
12           flag = false;
13           if (flag)
14                   cout <<"Test Succeed"<< endl;
15           else
16                   cout <<"Test Failed"<< endl;
17           counter = 0;
18           return flag;
19   }
20 }
```

**Figure 5. Implementation of the transport_dbg function in AES core**

The data is carried to the transport_dbg function via generic payload objects. The shared variable _INPUT is a global variable containing the received data through the normal communication channel i.e., blocking transport interface.

```
1   …
2   if (!mode){ // we are in test mode
3           trans.set_data_ptr(data);
4           enc_port->b_transport(trans, delay);

5           //serial transfer
6           for (int i = 0; i < 14; i ++) {
7                   dbg_trans.set_data_ptr(data[i]));
8                   trans.set_write();
9                   dbg_trans.set_write();
10                  enc_port->transport_dbg(dbg_trans);
11          }

12          if (trans.is_response_ok()){
13                  …
14          }
15  }
16  else//we are in normal operation mode
17  {
```

```
18          //System Normal Operaion
19          …
20  }
```

**Figure 6. Calling DTI in test mode to test communication**

In addition to the implementation of the transport_dbg function, the method of calling this function is important to mimic the behavior of a certain RTL test infrastructure at high level. Figure 6 shows the calling of transport_dbg function in the implementation of the first scenario.

The variable "mode" specified in line 2 is a Boolean variable showing the operation mode of the system. If "mode" is set to false, the system works in test mode, and starts using test data, whereas if set to true, the system resumes its normal operation mode.

*B. Online testing:* In this scenario, testing the communication will not affect the normal operation of the system. A possible schema for on-line testing may exploit the use of detection codes such as for instance parity bits to be transferred together with the data through a different communication channel rather than the one used to transfer the data itself. This can be implemented at high level by using the DTI for parity bit transaction. The implementation of the transport_dbg function in this case is shown in figure 7.

```
1   unsignedint AES::transport_dbg (tlm::tlm_generic_payload& trans){
2           bool parity[] = new bool[data_length];
3           bool flag = true;
4           for (int i = 0; i < data_length; i++)
5                   parity[i] = calculate_parity ((char) _INPUT[i]);

6           unsignedchar* cparity;
7           cparity = (unsignedchar*)parity;
8           unsignedchar* rparity;
9           rparity = trans.get_data_ptr();
10          for (int i=0; i< data_length; i++)
11                  if (rpatiry[i] != cparity[i])
12                          flag = false;
13          if (flag)
14                  cout <<"Test Succeed"<< endl;
15          else
16                  cout <<"Test Failed"<< endl;
17          return fla
18  }
```

**Figure 7. Implementation of the transport_dbg function in AES core**

It should be mentioned that "calculate_parity" specified in line 5 is a function which calculates the even parity of a given character.

Besides from the transport_dbg function, the function call in this scenario is different from what we have in the offline testing scenario and can be used as an indicator of the selected testing infrastructure. This is shown in figure 8.

As shown in line 8, the functionality of the communication channel is tested during the normal operation of the system and after each data transaction to guarantee reliability of the communication channel after each transaction. The operation is considered as completed only if the correct functionality of the communication channel is certified by the DTI.

These two scenarios shows that the implementation of the transport_dbg function and also its call sequence can specify the testing infrastructure chosen for testing at RTL. With respect to this fact, we can have different variations of the transport_dbg function implementation and call to refer to different test infrastructures at RTL.

```
1   trans.set_write();
2   trans.set_data_ptr(data);
3   enc_port->b_transport(trans, delay);
4   for (int i = 0; i < data_length; i ++)
5           parity[i] = calculateParity((char)data[i]);
6   dbg_trans.set_data_ptr(reinterpret_cast<unsignedchar*> (parity));
7   dbg_trans.set_write();
8   if (enc_port->transport_dbg(dbg_trans))
9           if (trans.is_response_ok())
10          {
11                  //Normal System Functionality
12                  …
13          }
```

 **Figure 8. Calling DTI in online testing scenario**

### *High level of abstraction after HW/SW partitioning:*

The next level of abstraction contains the system behavior after hardware/software partitioning. To show that the proposed method can be used in both hardware and software, we partition the system into a hardware AES core and a software algorithm for decrypting the cipher text to be executed on a microprocessor. Figure 9 shows the architecture of the system at this level.

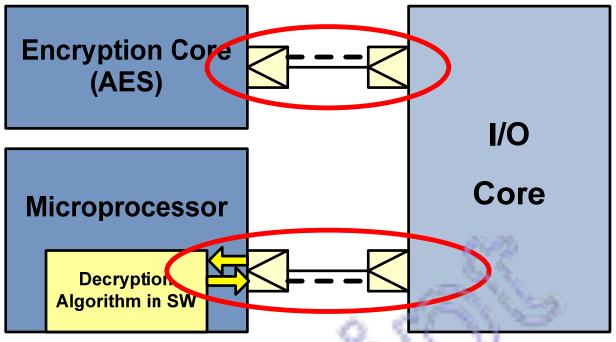


**Figure 9. System architecture after partitioning**

As shown in figure 9, in this case, the path between the I/O core and the AES core is equal to the situation proposed Figure 4, but the path between the I/O core, and the decryption core is split into two distinct paths: a hardware connection from I/O to the microprocessor and a software connection from inside the microprocessor to the specific software process.

The transport_dbg functions implemented in figure 5 and figure 7 can still be used to test the path between the I/O core and the AES core for equipping the design with offline and online test infrastructure, respectively. They can also be used to test the connection between the I/O core and the microprocessor just like the I/O core and the Decryption core. The only difference is that another path is added from microprocessor and the decryption algorithm which can be tested using software testing methods. In other words, in case of software, the connection between the initiator and the microprocessor can be tested in the same way as the initiator and the IP core.

## 6. Experimental Results

To proof the concepts introduced in this paper, we have to show that the high-level test infrastructure proposed in Figure 4 can be easily mapped to one of the widely used test infrastructures for communication channels at RTL level in the presence of an automatic synthesis tool. Since at the moment a complete and automatic synthesis flow from TLM 2.0 down to RTL is still not available, and the translation is mainly performed manually, we focus our attention on the offline test model proposed in Section 5, only. The same results can be then easily extended to the remaining solutions or even to new test solutions not included in this paper.

The different cores proposed in Figure 9 have been mapped to already available IP cores. The microprocessor core has been mapped to the LEON3 microprocessor [20], used as target computation unit for running the AES decryption algorithm, while the encryption core has been mapped to the AES encryption IP core introduced in [15] also including a Built In Self Test (BIST) facility. The communication channels have been mapped to the AMBA bus [21], which is a bus structure compatible with the LEON3 microprocessor, and a custom I/O core has been described to validate the system, and to play the role of the master in the implemented system.

Figure 10 summarizes the architecture of the resulting system. It is important to highlight that at this level the communication channels introduced in Figure 9, i.e., the channel between the I/O core and the Encryption core, and the channel between the I/O core and the microprocessor, are here mapped to a single physical interconnection element, the AMBA bus. The test infrastructure modelled at high level should therefore be mapped to RTL structures able to test the functionality of the AMBA bus.

Describing the test infrastructure at RTL level means translating the DTI, and the transport_dbg function to a specific test architecture. In the offline scenario (see Section 5), according to the high level implementation of the transport_dbg function, the test method should transfer patterns from the I/O core to Encryption core, or to the microprocessor core both on the normal interface, and on the DTI (using a serial transmission). The received patterns should be then compared to detect defects. This leads to the modified architecture of our RTL implementation proposed in Figure 11.

An additional interconnection component has been included to model the DTI. In this case widely used Test Access Mechanisms as the one proposed in [22], as well as custom interconnection schema can be exploited.

In our case we designed a simple serial link allowing the I/O core to send information between the I/O core and both the Encryption Core, and the LEON3 Microprocessor.

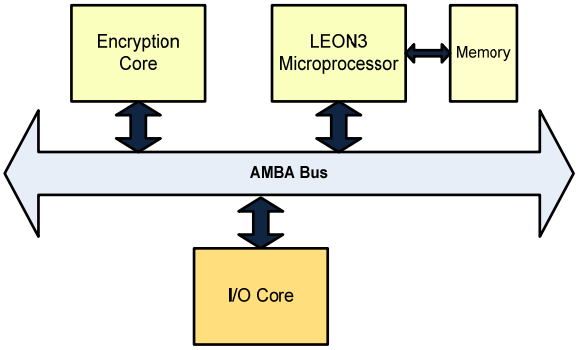


**Figure 10. System architecture at RTL**

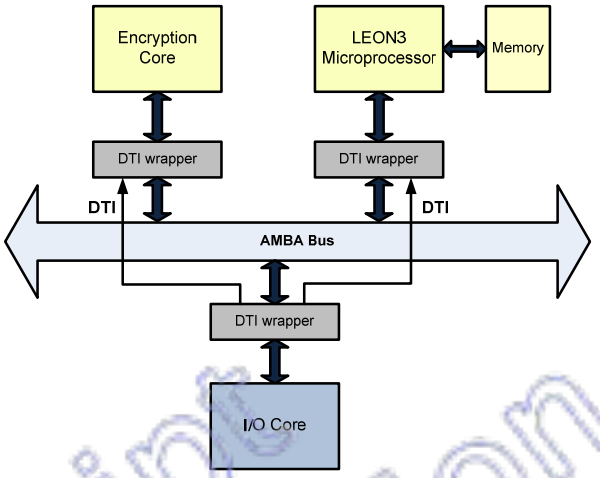


**Figure 11. Modified System architecture at RTL**

Finally the transport_dbg function has been implemented as a small wrapper providing additional functionalities to the Encryption and microprocessor cores. While in the case of the Encryption core this wrapper is in charge of receiving the serial copy of the patterns on the DTI and comparing them with the patterns transmitted on the AMBA bus, in case of the LEON the checking phase can be demanded to the microprocessor itself, thus reducing the complexity of the wrapper.

The architecture shown in Figure 11 presents a quite evident one-to-one correspondence between the high level model and the RTL implementation of the design. At RTL, the main design activity has been devoted to identify available infrastructures that better integrate into the target system, and to design small test modules to directly translate the transport_dbg function activities into a hardware description. Nevertheless, a key element is still missing, a set of test patterns to use during the test phase of the communication channels. Test patterns are strictly connected to physical faults of the target technology, and can be defined only after the definition of the target hardware. In our implementation, we exploited the methodology for generating test patterns proposed in [19]. The paper proposes a test pattern generation algorithm for communication buses able to provide, given the number N of lines composing the bus, $2\lceil \text{Log2}(N) \rceil$ patterns detecting shorts, opens and delay fault models. The patterns are generated using a very simple algorithm:

1. Each line of the bus is labelled with an increasing number expressed as a $\lceil \text{Log2}(N) \rceil$ bit binary number;
2. The binary numbers used to label each interconnection can be used to form a matrix of bits, where each row is associated with an interconnection line and contains the label of the line. The test patterns are obtained by reading the matrix by column (see figure 12.a).
3. In order to include delay faults in the test, each test pattern obtained during step 2 should be followed by a complemented test pattern (see figure 12.b).

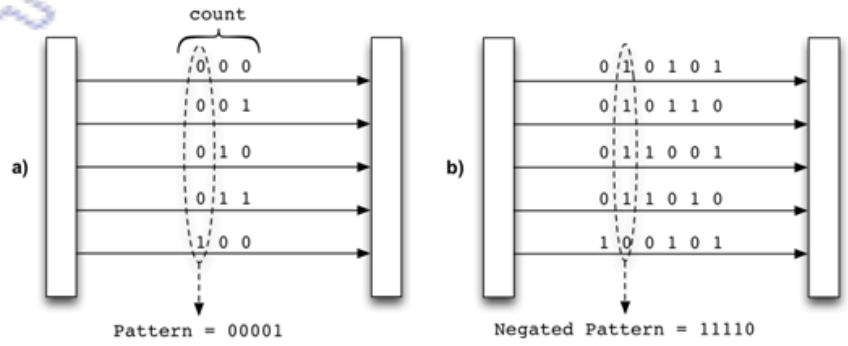


**Figure 12. Test Pattern Generation**

[19] demonstrates that the number of generated patterns is optimum. In our case, we also take advantage of the possibility of offline generation of patterns, without adding any extra hardware or software algorithm.

In our specific implementation, data are transmitted on the AMBA bus as 32bit words. According to [19] we generated a set of 10 test patterns. The same set of patterns has been used during the test phase of both the communication between the encryption core and the I/O core, and between the microprocessor core and the I/O core.

A similar straightforward translation could be easily applied also for the on-line test scenario proposed in section 5.

## 7. Summary and Conclusions

In this paper, we propose a method to define different test infrastructures at high level of abstraction, i.e., TLM, which could be synthesized to proper test facilities at RTL level. The test infrastructure has been defined at high level using the debug transport interface which is originally defined by OSCI TLM 2.0 standard for debugging at transaction level and is not synthesizable. We introduced another usage for DTI in order to introduce the test infrastructure at TLM. This new usage is synthesizable and could be translated to RTL testing facilities in the existence of a proper synthesis tool. We tested our introduced method for two known different test infrastructures: online and offline testing and showed that it works even after partitioning of the system into hardware and software. In other words, the proposed method can be applied to the system at very high level of abstraction before the partitioning of the system and synthesized to the lower levels of abstraction with other parts of the system in each synthesis step. We test our method on TLM 2.0 platform with an encryption/decryption system using AES algorithm and show that the introduced method works for both the system in pure hardware and also in the case that the decryption algorithm is a piece of software executed on a LEON3 microprocessor. The RTL implementations justify the validity of the proposed method and guarantee its synthesizability to lower level of abstraction i.e., RTL.

## 8. References


[1] F. Ghenassia, Ed., Transaction-Level Modeling with SystemC –TLM Concepts and Applications for Embedded Systems. Springer, 2005.

[2] Open SystemC Initiative (OSCI) TLM Working Group, "Transaction level modeling standard 2 (OSCI TLM 2)," June 2008, www.systemc.org.

[3] J. A. Carballo and S. R. Nassif, "Impact of design-manufacturing interface on SoC design methodologies," IEEE Design & Test of Computers, vol. 21, no. 3, pp. 183–191, 2004.

[4] H. Alemzadeh, S. D. arlo, F. Refan, P.Prinetto, Z.Navabi, "Plug & Test at system level via testable TLM primitives," in Proc. IEEE International Test Conference (ITC), 2008, pp. 1-10

[5] M. A. Kochte, C. G. Zoellin, M. E. Imhof, R. Salimi Khaligh, M. Radetzki, H. J. Wunderlich, S. Di Carlo, P. Prinetto, "Test Exploration and Validation Using Transaction Level Models, " Design, Automation and Test in Europe (DATE'09), Nice, France, April 20-24, 2009.

[6] Open SystemC Initiative (OSCI) TLM Working Group, "Transaction level modeling standard 2 (OSCI TLM 2)," June 2008, www.systemc.org.

[7] A. Rose, S. Swan, J. Pierce, J. M. Fernandez, "Transaction Level Modeling in SystemC," TLM 1.0 white paper.

[8] A. Donlin, "Transaction level modeling: flows and use models," in Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2004, pp. 75–80.

[9] L. Cai and D. Gajski, "Transaction level modeling: an overview," in Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2003, pp. 19–24.

[10] Joan Daemen, Vincent Rijmen, *The Design of Rinjael, AES - The Advanced Encryption Standard*, Springer, ISBN 3-540-42580- 241.

[11] D. C. Black, J. Donovan, B. Bunton, A. Keist, *SystemC from the ground up*, Kluwer Academic Publishers, 2004.

[12] W. Tibboel, V. Reyes, M. Klompstra, D. Alders, "System-Level Design Flow Based on a Functional Reference for HW and SW," in Proc. IEEE Design Automation Conference (DAC), 2007.

[13] IEEE Standard SystemC Language Reference Manual, 2006.

[14] Data Encryption Standard, Federal Information Processing Standard (FIPS), Publication 46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., January 1977.

[15] G. Di Natale, M. Doulcier, M. L. Flottes, B. Rouzeyre
Self-Test Techniques for Crypto-Devices, IEEE Transaction on VLSI Systems, 2009.

[16] "International Technology Roadmap for Semiconductor (ITRS)"

[17] W. Kautz, "Testing of faults in wiring interconnection", IEEE Transaction on Computers, vol. 23, 1974, pp. 358–363.

[18] Goel and McMahon. "Electronic Chip-In-Place Test," in Proc. IEEE Design Automation, 1982, pp. 482-488.

[19] A. Jutman, "At-speed on-chip diagnosis of board-level interconnect faults," in Proc. IEEE European Test Symposium (ETS), 2004, pp. 2-7

[20] J.Gaisler, GRLIB IP Library User's Manual, Datasheet. http://gaisler.com/products/grlib/grip.pdf

[21] AMBA™ Specification, http://www.gaisler.com/doc/amba.pdf