社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2007－SLDM－130（7）
2007／5／11

ハードウエアユニット検証環境の自動生成

バスブリッジ設計での実証

森沢 和道† 岩下 洋哲† 高山浩一郎†

† 株式会社富士通研究所 〒211-8588 神奈川県川崎市中原区上小田中 4-1-1
E-mail: †{rafael,iwashita.hiroak,k.takayama}@jp.fujitsu.com

あらまし SoC の複雑化によって検証に要する工数は増大している。多くの場合、SoC とそれを構成するハードウエアユニットの検証はシミュレーションに頼っている。シミュレーションによる検証作業の内訳を調べてみると大きく 3 つに分かれている: 仕様書の理解、検証環境の構築 (テストベンチ、テストパターン、参照モデル作成を含む)、シミュレーション。検証の工数を削減するためには、この 3 つの作業の工数を減らさなければならない。その内、仕様書の理解及び、シミュレーションの工数削減に対する取り組みは存在するが、技術的な課題は大きい。一方、検証環境構築の工数削減の課題は他のと比較してハードルは低くいが効果は大きいため、後者に着目した。検証環境構築の工数削減に有効な手段は検証環境を構成する部品の再利用及び、検証環境部品の自動生成である。任意のハードウエアユニットの単体検証に対応した検証環境構築は非常に難しい問題である。特にハードウエアユニットが複雑な機能を実現する場合は参照モデルの構築が難しい。逆にデータ加工をしない、主に転送をするユニット (バスブリッジは代表的な例) は多くの場合では参照モデル構築が必要なく、検証環境構築のハードルは低い。本稿は後者の検証環境構築問題に着目し、テストベンチの基本的な部品の自動生成手法を開発した。この手法をバスブリッジのユニット検証環境構築に用いて検証作業の工数削減を実現した。

キーワード テストベンチ、ユニット検証、トランザクタ、プロトコルチェッカ

Automatic Generation of a Verification Environment for Hardware Units

Application to a Bus Bridge Design

Rafael K. MORIZAWA†, Hiroaki IWASHITA†, and Koichiro TAKAYAMA†

† Fujitsu Laboratories, LTD. 4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki 211-8588 Japan
E-mail: †{rafael,iwashita.hiroak,k.takayama}@jp.fujitsu.com

Abstract The verification cost of complex SoCs has been increasing in a fast pace. Many techniques and methodologies have been developed to address this problem. Nevertheless logic simulation is still the most used technique to verify SoCs. In order to decrease the verification cost (time) of a design using logic simulation, we first must analyze the subtasks that compose it: specification analysis, building the verification environment (which includes the testbench, test pattern, and reference model), and simulation. There has been work on methodologies that improve the specification analysis and the simulation cost, but the technical challenges that must be overcome are big. On the other hand, improving the process of building verification environments has a lower technical hurdle (meaning low cost of implementation) but is very effective. We feel that this is not very explored, thus, in this paper we target this problem. Verification component's reuse and its automatic generation are key factors to decrease the cost of building a verification environment. In general it is very difficult to automatically generate components such as reference models for complex hardware units in an SoC. However, hardware units such as bus bridges or DMA controllers, where there is no data computation, do not require complex reference models. This lowers the hurdle for automatic generation of verification environments. In this paper we target the latter type of hardware units and propose a methodology to generate components used in a verification environment. We also present a case study where the proposed methodology has been used to build the verification environment of a bus bridge used in a commercial product.

Key words Testbench, Unit Verification, Transactor, Protocol checker

1. Introduction

The design and implementation of an SoC has been increasing in complexity. However, the verification complexity of an SoC has been increasing at even a higher pace. Various methodologies, techniques, and tools have been developed to address this problem. Despite the proposal of new techniques, by far the most used verification methodology is still logic simulation.

When we focus on the task of verifying an SoC or a hardware unit using simulation, we can divide it into 3 subtasks: understanding the DUT's (Design Under Test) specification, building the verification environment, and the actual simulation. From our experience, about a third of the total verification effort is spent in each of these subtasks. There has been some ongoing work on methodologies to improve the process of analyzing a specification [2] and on improving simulation cost [1]. However, the technical hurdles to overcome are big. On the other hand, the technical hurdles that need to be overcome to improve the process of building verification environments are not as big, meaning lower cost of implementation. We feel that this is not very explored as it should be.

References [4], [5] propose methodologies for functional verification of hardware units in an SoC and how to build a verification environment. These methodologies basically share the same characteristics by recommending that verification environment components be built at a higher level of abstraction (namely transaction level [7]) than the signal level abstraction of DUTs to ease the implementation and reusability of the components. These components can be divided into three categories: an architecture or framework used to drive/collect stimuli to/from the DUT, the stimuli set, and the reference model [4], [5].

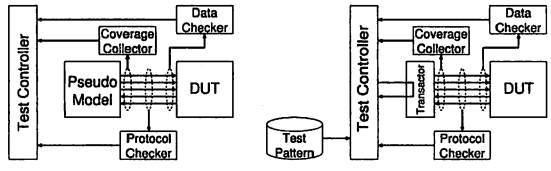
Although the methodologies recommend best practices and the authors (tool vendors) provide libraries and tools to be used in the verification of hardware units, there are a few points that are not properly addressed in these methodologies. These points are how to build an adequate stimuli set, how to build a reference model, and how to build a *transactors* between transaction and signal levels. It is up to the user of these methodologies to implement each of these key points.

There is some ongoing work on the generation of stimuli sets from functional specifications [3]. The reference models used in verification to decide whether the response of the DUT is correct or not, can be derived from the same reference model used for designing the DUT. Many methods that automate the creation of verification components related to signal-level interface protocols have been proposed. Regular expressions have often been used as basis of formal specification languages for signal-level interface protocols. Protocol checkers can be generated automatically from them [8], [11]. Once a protocol checker is generated as an FSM, *constraint synthesis* technique [9] can be used to generate a pseudo model. It is also reported that transactors have been successfully generated from regular-expression-based specifications [10]–[12].

In this paper we focus on the automatic generation of verification environments for hardware units. Our target is the generation of the components used to interface transaction and signal levels, which is left to the user by the tool vendor's methodologies. The main contribution of this paper is that we have filled this gap by providing tools that implement the techniques that allow the automatic generation of transactors and pseudo models.

In section 2. we briefly explain our unit verification environment. Then in section 3. we describe our method for generating the transactors, pseudo models, and protocol checkers. In section 4. we present a case study in which we actually used the developed tools to build the verification environment of a bus bridge in a commercial SoC product.

2. Unit Verification Environment



(a) Random simulation (b) Transaction-based simulation

Fig. 1 Unit verification environments.

For unit verification, we need to build the environment that provides stimuli for the design under test (DUT) and checks its response. In this paper the DUT is a hardware unit such as bus bridges or DMA controllers where no data computation is done by the DUT. Examples of unit verification environments are illustrated in Fig. 1.

The first example (a) is an environment for random simulation. The *pseudo model* selects DUT inputs every cycle from the patterns that do not violate the interface protocol at random. The *protocol checker* monitors all interface signals, finds interface protocol errors of the DUT, and possibly finds errors of the pseudo model. The *data checker* checks the correctness of the data computed by the DUT.

The second example (b) is an environment for transaction-based simulation. The user of the environment controls test scenarios and calls an API of the transactor for each event at transaction-level. The *transactor* translates transaction-level communication into signal-based communication and returns a DUT's response to the driver through the API.

In both cases, the simulation effort is evaluated based on coverage analysis. Several types of coverage may be defined in the verification of a DUT. *Code coverage* is the percentage of a DUT's source code activated in a simulation run, which includes various subclasses, e.g. *block coverage* and *expression coverage*. *Functional coverage* is the percentage of a DUT's functions checked by the test scenarios. Another coverage, called *transaction coverage* in

this paper, can be measured using the protocol checker. The transaction coverage is the percentage of the signal-level instances of the interface protocol covered in a simulation run. It can be measured as the protocol checker's internal state transitions covered in a simulation run and is collected by the *coverage collector*.

3. Generating a Verification Environment

Our target is the automatic generation of transactors, data checkers, coverage collectors, protocol checkers, and pseudo models. The goal of the proposed method is to build a framework in which designers specify the interface protocol definition of hardware units, this specification is shared among the design and verification teams, and it is also used to generate various verification components.

We use timing-diagram-based formal protocol definition in tabular form, called *protocol definition table*. It is easy-to-read as well as easy-to-write. Conventional formal protocol definition is based on text-based mathematical description, which is fully understood only by the person who wrote it and some other skilled verification engineers. It is very important that a formal protocol definition is shared by all members of the design and verification teams because it drastically reduces misunderstanding among the team. Since misunderstanding is a very common cause of design flaws, a formal protocol is really effective in reducing the total cost of design and verification. We intentionally limited the expressive power of protocol definition table in order to keep it easy-to-read for all people. From our experience, interface protocols used in a SoC can be classified into two groups: one is a group of simple protocols and another one is a group of standard high-performance bus or I/O protocols. Our target is the former and the latter would be covered by general reuse and distribution methodologies of verification IPs.

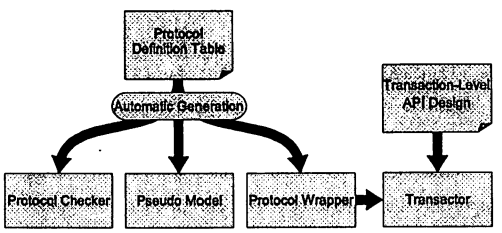


Fig. 2 Verification component generation flow.

The verification component generation flow is shown in Fig. 2. Protocol checker, pseudo model, and protocol wrapper are generated automatically from a protocol definition table. The protocol wrapper is a sub-component of a transactor. Additional information of transaction-level API, which is a manual design, is needed to complete the transactor. Some human decisions are needed to design the API, e.g. controllability of wait cycles, handling of burst-transfer, and choice of blocking/nonblocking interfaces.

The protocol wrapper is also used as a sub-component of a data checker or a coverage collector (Fig. 3 and 4). Likewise the transactor, we need additional information from the verification specification in order to complete the design of the data checker and the coverage collector.

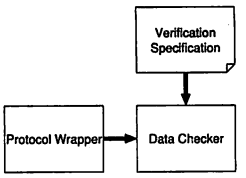


Fig. 3 Generating a data checker.

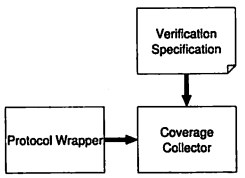


Fig. 4 Generating a coverage collector.

Protocol wrapper generation automates protocol-dependent and error-prone part of transactors, data checkers, and coverage collectors by leaving its customizable part for verification engineers. This enables a reduction in the time required to build a verification environment.

3.1 Protocol Definition Table

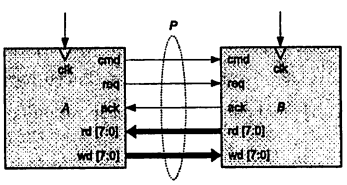


Fig. 5 Interface protocol *P* between units *A* and *B*.

The format of the protocol definition table is originally based on a subset of CWL [11], [12]. Briefly speaking, it is a regular expression with data equality. The protocol definition table for protocol *P* in Fig. 5 is shown in Fig. 6. A protocol definition table is basically a table composed of two blocks ("Port Definition" and "Sequence Definition") as in Fig. 6 (a). The first block can also be viewed as a timing diagram as in Fig. 6 (b). Our tool can switch those two views according to user operation.

In the first block, there are six rows corresponding to six signals defined in this protocol (`clk`, `cmd`, `req`, `ack`, `rd[7:0]`, and `wd[7:0]`). Each of the five columns on the right of the signal names expresses the combination of the signal values (patterns) at a certain cycle. Special value "`@`" means the rising edge of the clock signal and "`?`" means that the signal value is not restricted, i.e. don't-care, at that cycle. For each column, the name of the pattern is defined at somewhere in the second block. The patterns

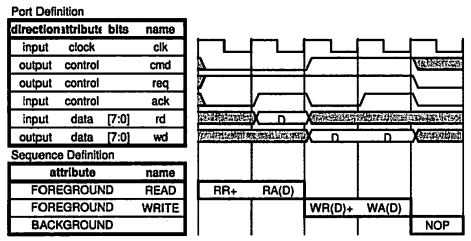
Port Definition

direction	attribute	bits	name					
input	clock		clk	@	@	@	@	@
output	control		cmd	0	0	1	1	?
output	control		req	1	1	1	1	0
input	control		ack	0	1	0	1	0
input	data	[7:0]	rd	?	D	?	?	?
output	data	[7:0]	wd	?	?	D	D	?

Sequence Definition

attribute	name					
FOREGROUND	READ	RR+	RA(D)			
FOREGROUND	WRITE			WR(D)+	WA(D)	
BACKGROUND						NOP

(a) Table view



(b) Diagram view

Fig. 6 Protocol definition table of *P*.

are named RR, RA(D), WR(D), WA(D), and NOP respectively. The sign "+" appended to RR and WR(D) means that those patterns can be repeated once or more.

In the second block, there are three rows corresponding to two types of transactions defined in this protocol (READ and WRITE) and the state when no transaction is performed. It means that transaction READ consists of one or more cycles of pattern RR followed by a single cycle of pattern RA(D), transaction WRITE consists of one or more cycles of pattern WR(D) followed by a single cycle of pattern WA(D), and pattern NOP appears when no transaction is active.

The same parameter used in a transaction definition means that the corresponding value must be the same throughout each transaction. Parameter D in the WRITE definition represents that the value of signal wd[7:0] must be stable during a WRITE transaction.

4. Case Study

We have actually used the verification environment generation flow described in this paper in the verification of a bus bridge. The specification of the bus bridge is shown below.

- The master side of the bridge interfaces with an AHB [13] bus and a proprietary bus.
- The slave side of the bridge interfaces with a memory interface controller.
- The bridge has an arbiter with a built-in priority to arbitrate the access of the masters to the memory interface controller.
- It also has a temporary data buffer.

The testbench used in the verification of this bus bridge is shown in Fig.7. The testbench consists of the following blocks.

- The bus bridge (DUT).

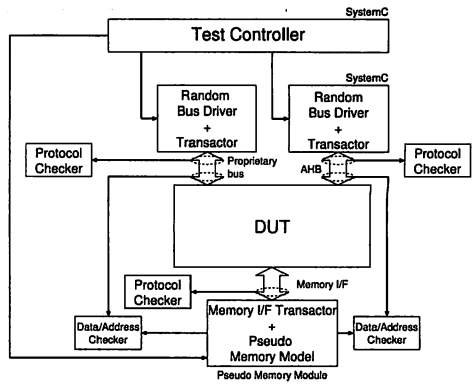


Fig. 7 Bus bridge testbench.

- The test controller.
- Protocol checkers for each protocol handled by the DUT.
  - AHB checker: used existing verification IP.
  - Proprietary bus checker: generated from the protocol definition table.
  - Memory I/F checker: generated from the protocol definition table.
- Bus master models.
  - AHB model: the transactor is an existing verification IP, and the driver is hand-built (SystemC).
  - Proprietary bus model: the transactor is generated from the protocol definition table, and the driver is hand-built (Verilog).
- Pseudo memory module: models the memory I/F controller. The transactor was generated from the protocol definition table. and the pseudo memory model was hand-built (Verilog)
- Data/address checkers.
  - Data/address checker between the AHB driver and the pseudo memory module: hand-built (Verilog).
  - Data/address checker between the proprietary bus driver and the pseudo memory module: hand-built (Verilog).

Table 1 shows the code size of the testbench components that were automatically generated. Table 2 shows the code size of the components that have to be hand-built in order to complete the design of the bus driver and memory model. Table 3 shows the size of the various protocol definition tables we defined in order to build the testbench. The size is shown in terms of the table's columns × rows.

The DUT is a design that is going to be used with different memory interfaces, all sharing the same interface protocol. Different memory interfaces, even though sharing the same protocol, may have different transaction behaviors. Thus the bus bridge was designed to support any behavior within the defined protocol state space.

The standard method to verify this type of design is to bring (or build) the models of the various memory interface controllers that

Table 1 Verilog code size of automatically generated components

Block name	size (lines)
Proprietary bus protocol checker	9168
Proprietary bus transactor	9398
Memory I/F protocol checker	22154
Memory I/F transactor	34122

Table 2 Verilog code size of hand-built components

Block name	size (lines)
Proprietary bus driver	147
Memory model	414

Table 3 Protocol Definition Table Size

Protocol definition	size (columns × rows)
Proprietary bus protocol checker	18 × 15
Proprietary bus transactor	23 × 23
Memory I/F protocol checker	64 × 27
Memory I/F transactor	75 × 40

the DUT may interface with, and verify the DUT with each of them. This is a time consuming task since we must do a simulation run for for each memory interface controller model.

In this case study, we built a memory interface transactor based on the memory interface protocol definition table. By integrating the transactor and pseudo memory model into the pseudo memory module, we obtain a memory model that responds to requests from the DUT with transactions that cover the entire protocol state space.

The approach taken to verify this design is simulating the DUT into which transactions are randomly issued by the bus master drivers. The approach consists of:

(1) Each bus master driver generates random transactions and issues them at random.

(2) Compare the address and transaction type issued by the bus master drivers with the address and transaction type that the DUT issues to the memory interface.

(3) Compare the data transferred from/to the bus masters with the data stored in the memory model.

The verification goal is to achieve 100% block, expression, and transaction coverage.

In order to evaluate the effectiveness of the verification using the pseudo model described above, we evaluated the following 2 cases:

Case A The pseudo memory module models a specific type of memory interface block.

Case B The pseudo memory module models transactions that cover the entire memory I/F protocol state space.

In each case we measured 3 different coverage metrics: block, expression, and transaction. The code coverage is shown in Table 4. In this table we show in different entries the block/expression coverage of the hardware block that implements the memory I/F and the other hardware blocks. We can see that both block and expression coverage rates of the memory I/F hardware block are lower in case A compared to case B. Because the only difference in cases A and B is the pseudo memory module, it is natural that block and expression coverage of the other hardware blocks do not differ.

Table 4 Code Coverage

Coverage type	Case A (%)	Case B (%)
Block coverage (memory I/F)	85.7	99.3
Block coverage (other blocks)	99.4	99.4
Expression coverage (memory I/F)	82.0	96.3
Expression coverage (other blocks)	99.2	99.2

Table 5 Transaction Coverage

Coverage type	Case A (%)	Case B (%)
Transaction coverage	64.0	95.7

Table 5 shows the transaction coverage of the memory I/F protocol. We can easily see that the transaction coverage of case A is much lower than of case B. This means that the DUT's memory I/F hardware in case B is more exercised than in case A.

We analyzed the uncovered blocks and expression of both cases A and B. In case B all the uncovered blocks and expressions are false warnings. So if we remove them, we achieve a 100% block and expression coverage. In case A, even if we remove all the false warnings, the block/expression coverage is still very far from 100%.

We also analyzed the uncovered transactions of both cases A and B. In case B, all the uncovered transactions cannot occur because the DUT do not support them. So, if we remove these transactions from the transaction coverage count we achieve 100% transaction coverage. In case A, even if we remove the same transactions we removed in case B from the coverage count, the transaction coverage is 67%, much lower than 100%.

The result of this case study shows that one random transaction generator, which is used in pseudo models and that is built from a protocol specification table, can be used to thoroughly verify a DUT. We do not need to build or bring various pseudo models to achieve a high quality verification (high, 100% or close to 100% block, expression, and transaction coverage)

If this methodology did not exist we would have built all the components (protocol checkers and transactors) by hand, increasing the time required to build and verify the verification environment. In this case study we took about 4 weeks to build the verification environment shown in Fig. 7. There are 9 components (besides the DUT) in the verification environment, 7 were new components and 2 were existing verification IPs. Among the newly generated components, the test controller and both data/address checkers were hand-built. The other 4 components were totally or partially gener-

ated automatically (as it can be seen in Table 2, the hand-built part is very small compared to the automatically generated part). Most of the 4 weeks were spent building and verifying the test controller, the most complex hand-built component, and the data/address checkers.

From our experience, hand building all the 7 new components would have taken at least twice as long. Also, the quality of these components would not be the same compared to the components automatically generated. One might argue that 4 weeks is not a short period of time. However, the author was not completely familiarized with the tools that automatically generate the verification environment components. The protocol checker generator has been in use for some time, but the pseudo model and protocol wrapper generators were first deployed in the case study's verification project (the tools were not mature).

We believe that with experience and mature tools, 3 weeks would be enough. If the DUT complexity and its interfacing protocols were less complex, 1 week would be enough to build the verification environment (we still would need to build the test controller and data/address checkers, which would take most of this 1 week work).

We also found out that the protocol definition table is an excellent means to share information. In this case study the verification team actually used the protocol definition table to deepen the understanding of the proprietary protocol. In a conventional approach, the protocol's properties would be listed like PSL properties and a review meeting, that takes a few hours, would be held with the design team to review these properties.

In a PSL description, one would have to specify properties for each signal, its dependency (timing) relationships with respect to other signals using temporal relations. Checking whether every property is correct and whether the assertions correctly define and cover 100% the protocol is a difficult task. Because the protocol definition table defines a protocol like a timing diagram, covering every signal involved in the protocol, a close comparison with timing diagrams is enough to certify whether, for example, a protocol checker will cover a proprietary protocol.

For this reason, using the protocol definition table the verification team was able to certify the proprietary bus protocol by exchanging a few e-mails (about two to three e-mails exchanges were enough).

5. Conclusion

We proposed a framework in which designers specify the interface protocol definition of hardware units, this specification is shared among the design and verification teams, and it is also used to generate various verification components.

We introduced the protocol definition table, which is a timing-diagram-based formal protocol definition in tabular form. The protocol definition table is not only an input to the automatic generation method, but is also a document for all members of the design and verification teams. We intentionally limited the expressive power of protocol definition table in order to keep it easy-to-read for all people.

Protocol checker, pseudo model, and protocol wrapper can be generated automatically from a protocol definition table. Transactors, as well as coverage collectors and data checkers, can be easily created using the protocol wrapper and additional information from either the verification specification or the transaction-level API design specification.

In a case study, we have found that the methodology proposed in this paper is very flexible and can provide sufficient help for designers and verification engineers by allowing the share of information and decreasing the time required to build the verification environment.

References

[1] Minoru Shoji, Fumiyasu Hirose, Shintaro Shimogori, Satoshi Kowatari, Hiroshi Nagai, "Acceleration of Behavioral Simulation on Simulation Specific Machines," in *Proc. European conference on Design and Test*, pp. 373–377, 1997.

[2] Qiang Zhu, Ryosuke Oishi, Takashi Hasegawa, Tsuneo Nakata, "System-On-Chip Validation using UML and CWL," in *Proc. International Conference on Hardware/Software Codesign and System Synthesis*, pp. 92–97, 2004.

[3] Ansuman Banerjee, Bhaskar Pal, Sayantan Das Abhijeet Kumar and Pallab Dasgupta, "Test Generation Games from Formal Specifications," in *Proc. Design Automation Conference*, pp. 827–832, 2006.

[4] Mark Glasser, Adam Rose, Tom Fitzpatrick, Dave Rich, Harry Foster, "The Verification Cookbook—Advanced Verification Methodology SystemC and SystemVerilog Version 2.0", Mentor Graphics, 2006.

[5] Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale, "Verification Methodology Manual for SystemVerilog", Springer, 2005.

[6] Cadence, "Incisive Plan-To-Closure Methodology", 2006. http://www.cadence.com/products/functional_ver/incisive_plan_to_closure_methodology.aspx

[7] OSCI, "SystemC TLM 2.0", 2007. http://www.systemc.org/web/sitedocs/TLM_2_0.html

[8] M. T. Oliveira and A. J. Hu, "High-Level Specification and Automatic Generation of IP Interface Monitors," in *Proc. Design Automation Conference*, pp. 129–134, 2002.

[9] J. Yuan, K. Albin, A. Aziz, and C. Pixley, "Constraint Synthesis for Environment Modeling in Functional Verification," in *Proc. Design Automation Conference*, pp. 296–299, 2003.

[10] F. Balarin and R. Passerone, "Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation," in *Proc. Design, Automation, and Test in Europe Conference*, pp. 1013–1018 , 2006.

[11] K. Ara and K. Suzuki, "A Proposal for Transaction-Level Verification with Component Wrapper Language," in *Proc. Design, Automation, and Test in Europe Conference*, pp. 82–87, 2003.

[12] K. Ara and K. Suzuki, "Fine-Grained Transaction-Level Verification: Using a Variable Transactor for Improved Coverage at the Signal Level," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and System s*, vol. 24, no. 8, pp. 1234–1240, August 2005.

[13] ARM, "AMBA Specification (Rev 2.0)," May 1999. http://www.arm.com/products/solutions/AMBAHomePage.html