HPChecker: An AMBA AHB On-Chip Bus Protocol Checker with Efficient Verification Mechanisms

Liang-Bi CHEN†(a), Student Member, Jiun-Cheng JU†, Chien-Chou WANG†, and Ing-Jer HUANG†(b), Nonmembers

SUMMARY Bus-based system-on-a-chip (SoC) design has become the major integrated methodology for shortening SoC design time. The major challenge is how to verify on-chip bus protocols efficiently. Although traditional simulation-based bus protocol monitors can check whether bus signals obey bus protocol or not, they are still lack of an efficient bus protocols verification environment such as FPGA-level or chip-level. To overcome the shortage, we propose a rule-based synthesizable AMBA AHB on-chip bus protocol checker, which contains 73 related AHB on-chip bus protocol rules to check AHB bus signal behaviors, and two corresponding verification mechanisms: an error reference table (ERT) and a windowed trace buffer, to shorten verification time.

key words: AMBA, debugging, system-on-a-chip (SoC), protocol checker, verification

1. Introduction

In recent years, the demands for more powerful consumer electronics products and the huge capacity of today’s silicon technology have changed system-on-a-chip (SoC) design from the leading-age to mainstream design practice, thus, the complexity of the SoCs designs increases continuously. The typical SoC always contains one or more processors, as well as large amount of memory, bus-based architecture, peripherals, coprocessors, I/O channels, and so on. These functional blocks are called intellectual property (IP) cores [1].

Nowadays, the on-chip bus architecture has become the major integrated methodology for implementing a system-on-a-chip (SoC). The on-chip bus specification provides a standard interface that facilitates IPs integration and easily communicates with each IP in an SoC, as shown in Fig. 1. On the other hand, the designers just integrate their owned IPs with the third party IPs into the SoC to significantly reduce design cycles. However, the main issue is how to efficiently make sure the IP functionality, which works correctly after integrating to dedicated on-chip bus architecture.

When adopting these IP cores provided by IP vendors or designed by ourselves, then the system designer must do verification works. The purpose of verification is to guarantee that the design meets the functional requirements as defined in the functional specification. In general, verification of SoCs takes about 70 to 80 percent [2] of the total development effort for the SoC designs. Hence, the verification design has become one of the main challenges of the SoC development.

For bus protocol verification, the protocol checking [2] verifies that no bus protocol violations or block-to-block interconnect violations occurred during simulation. Therefore, the protocol checking exercises bus-centric test which verifies correct bus operation while applying stimulus via stimulus generators. However, many errors may occur in real-time that traditional monitor-based bus verification approaches cannot find errors in simulation environment. Thus several commercial verification IPs provide bus protocol checker to solve the problem. Although protocol checkers can find errors on real-time environment, however, we still need to have more efficient ways to debug the system, because only violated rules cannot efficiently help designer to rapidly find errors.

As for the problem, we propose a rule-based synthesizable AMBA AHB on-chip bus protocol checker called HPChecker, based on monitor-based methodology. The HPChecker consists of a protocol checker module and two efficient verification mechanisms. The proposed protocol checker has 73 rules that provide master, slave, bus components, and reset issues. Two verification mechanisms (an error reference table and a windowed trace buffer) are proposed to improve debugging ability. The error reference table (ERT) can summarize total errors that have been occurred in simulation. The windowed trace buffer can capture multiple error bus signals, which designer can find protocol
errors according to these waveforms.

The rest of the paper is organized as follows. Section 2 reviews related works. The proposed AMBA AHB bus protocol checker is introduced in Sect. 3. Section 4 describes how to verify the proposed protocol checker. Section 5 demonstrates the experimental results. Our developed 3D graphics acceleration SoC and the LEON2 SoC are adopted as case studies in Sect. 5. Finally, we conclude in Sect. 6.

2. Related Works

We have performed a survey of previous researches, which works on the bus protocol verification and validation. We classify these works based on their focused problems.

Figure 2 shows the research organization for our related works. In our classification, the problem of bus protocol verification and validation can be briefly divided into three main aspects. The first one is formal modeling and verification that they used formal modeling to describe a bus protocol, and then used symbolic technologies to verify the bus’s correctness. The second one is interface synthesis that wishes to automatically generate the interface or called bridge between mismatched protocols. The last one is monitor-based checking. It also can be divided into two different efforts: test case generation and rule-based checker. Our work belongs to monitor-based checking.

In the formal modeling and verification, the most earlier researches are focused on this area, which the bus protocol can be modeled as a finite state transition and the properties are specified as a logical formula like computation tree logic (CTL) [3], then related tools performed a symbolic model checking based on binary decision diagrams (BDDs) [4]–[8] used this methodology to verify PCI bus functionality. Chauhan et al. [4] verified PCI bus protocol in two steps, single PCI bus and two PCI buses with bridge. Campos et al. [5] not only modeled PCI bus, they also added some information into model to help them to take care of performance issue. Goel and Lee used formal modeling and symbolic checking to verify IBM CoreConnect bus functionality. Roychoudhury et al. [9] used formal modeling and symbolic check to verify AMBA AHB bus functionality, and they mentioned the importance of bus pipeline characterization. Because of pipeline characterization, there exist two transfers on the bus at the same time that we must record enough information to keep checking the correctness of bus protocol. Shimizu et al. [10] also used formal modeling and symbolic check to verify Intel Itanium processor bus functionality. Formal modeling and symbolic check is the one of most popular topics in this domain. Formal verification can verify bus protocol correctness, but still it can only be done at higher abstraction level. They cannot be synthesized so as to help integrator to find the bugs at physical level.

In interface synthesis, from hardware electrical level to software message level, it may occur that integrator wants to integrate heterogeneous components into one project. To design an interface that communicates two heterogeneous components has become a challenge. Borriello et al. [11] introduced seven different levels of interface, and their associated concerns and synthesis artifacts. Rajawat et al. [12] represented a survey of some approaches for interface synthesis and compared them in a broad framework. D’silva et al. [13] used formal modeling to describe the interface between mismatched protocols. The formalism contained special primitives for modeling data, control and multiple clocks.

The other area of researches is monitor-based checker that captures the bus signals and checks the correctness. The monitor can capture signals and warn us if there is anything wrong with the signals. As shown in Fig. 2, the monitor-based approach includes rule-based checkers and test case generators. The rule-based checker is that we can specify some rules written with some specification and error signal asserts if the bus behaviors violate these rules. The advantage of rule-based checker is that we can establish rules without entirely understanding all the bus behavior, we just need to know the signal sequence and we can make rules. Another advantage is that rules can be added or deleted easily, because all rules are independent with each other.

Although monitor-based approach is efficient, it only can detect some rules that we established. This is not enough to confirm the compliance of bus protocol. Thus, an automated methodology is proposed. Through some specifications, tools can automated generate required test cases, correctness checker and coverage metrics.

Shimizu et al. [14] proposed an early rule-based monitor to check PCI bus protocol. They proposed a new style of specification to create some constraints, and monitor the bus signals to check errors. They especially noted that each constrain could only constraint output of one agent. Oliveira and Hu [15] proposed a high-level specification style that can generate a hardware monitor. The Interface-monitor-based methodologies are to watch the interface between a block and the rest of the system which are also proposed in [16] and [17]. These rule-based monitors can easily create some rules they want to observe by hardware or some specifications that can automatically generate hardware modules. However, it cannot guarantee the bus compliance if outputs pass all the rules.

![Fig. 2 Classification of related works for bus protocol verification and validation.](image-url)
Shimizu and Dill [18] proposed an effective method that they generate simulation environment, correctness checker, and functional coverage metrics from a signal specification. This method suits for any device under verification (DUV) that can be specified by their specification. Hence, we can apply this method to verify bus protocol.

Lin et al. [19] proposed a FSM model, which can help to extract the necessary properties and verify the data part of a bus transfer. They found the M-paths that indicate the meaningful path to create coverage metrics from FSM model. The advantage of this method is that we can easily find all the meaningful paths to create case metrics, but the disadvantage is that because the FSM model only constructs the legal state, they cannot find the bus protocol errors. Extending their FSM model in [19], Yang et al. [20] proposed a novel branch-and-bound algorithm for interface protocol compliance verification to find all the possible cases.

Nightingale and Goodenough [21] proposed a compliance test environment that contained a rule-based protocol checker, a rule-based coverage monitor and user defined test sequences to test AMBA AHB components. Their protocol checker checks bus signals cycle by cycle to ensure DUV obeys the rules they defined in protocol checker. Coverage monitor has a list of test cases that must be verified. The advantage of this method is we can establish test environment quickly, but it is very dependent on designer’s knowledge. It only can verify the rules they defined.

Bhaskar et al. [22] proposed a framework that includes symbolic validation and test case generation. They used specific high-level description of bus protocol and then their tool can automated generate variable verification aids. They generated OVA (OpenVera Assertions)-based protocol checker, constrained random test benches and coverage monitor. The advantage of this framework is that they provide variable verification aids to help debugging and verification. But their verification framework still runs at high level abstraction that cannot run at physical level. Besides, traditional protocol checkers would assert error signals or provide a printout error message for each error. However, this way is very inefficient for early debugging in long simulation period, because some violations would occur repeatedly, even occur huge times, if we want to print error messages every time when one rule violation is detected, then the error massage may have huge lines, but it just violates five rules. Therefore, to overcome this problem, we propose an error reference table which can solve this situation, we just indicate which IP violated which rule, this will help the SoC integrator for debugging easily in the early debugging period.

3. AMBA AHB Bus Protocol Checker

This section introduces the proposed AHB bus protocol checker (called HPChecker). The HPchecker is rule-based AHB protocol checker and it can capture all the AHB signals as the inputs for checking bus protocols. The checking rules are defined in [25], [27].

The block diagram of the HPChecker is shown in Fig. 3. The HPChecker consists of a protocol checker, a configuration register, an error reference table, and a windowed trace buffer. These functional blocks are described subsequently.

3.1 Protocol Checker

The protocol checker is the major module that performs the bus protocol checking. There are 73 rules to check the AMBA AHB protocol. We classify these rules into four categories based on the property of the IP as follows.

- **Master related**: 31 rules.
- **Slave related**: 16 rules.
- **Bus component related**: 15 rules, relating to arbiter and decoder.
- **System reset related**: 11 rules.

For each category, we also classify these rules into 16 sub-modules based on the rule property, as shown in Fig. 4.
The purpose of the classification is that the designer can modify the source code easily and fast.

3.2 Configuration Register

The configuration register can allow the designer to set some parameters, including mask, protocol checker enable, and max waiting cycle. The designer can base on the requirement of the system to mask some unnecessary rule checked. The enable bit can enable or disable the HPChecker. For example, if we integrate one master IP into pre-verified AHB bus platform, then we just need to check master-related rules and disable all other rules. If we modified an arbiter, we only need to check arbiter related rules. The max waiting cycle is set by the designer to check the HREADY max waiting cycle.

The rule configuration register will mask the ERROR outputs as shown in Fig. 3. If we disable some rules, the corresponding ERROR bits will be masked. Another applicable debugging method is that if we want to focus on a target IP, then we can enable corresponding rules and disable irrelevant rules, so we can narrow down the problem region.

3.3 Error Reference Table (ERT)

In general, traditional protocol checkers would assert error signals or provide a printout error message for each error, but this way is very inefficient for early debugging in long simulation period. In early debugging period, there may have many errors, especially when test bench or real case application has very long execution time. Such large amount of log messages or error signals waveform could help designer to debug limitedly, because that will be difficult to read or analyze in long execution time. Moreover, we are not concerned about the timing information when error occurred in most case. Because some violations would occur repeatedly, even occur huge times, if we want to print error messages every time when one rule violation is detected, then the error message may have huge lines, but it just violates five rules. Thus, we provide an error reference table (ERT) that can summarize what errors have been occurred. The ERT can provide the information that the IP violates bus protocol. It can help the designer for debugging easily in the early debugging period. The ERT is formed by a matrix. The input of the ERT is from the protocol checker, including the master and slave ID, error detected bit, and the ERROR signals.

Figure 5 shows the concept of the ERT, which the rows are corresponding to every error. The columns are corresponding to every ID of master or slave. The ERT supports 16 masters and 16 slaves, thus the number of column is 16. The ERT has 31 master-related rules, 16 slave-related rules, 11 reset-related rules, and 15 bus components-related rules. Each master-related and slave-related error needs one row due to that every 16 master/slave may occur this error. Bus-related and reset-related errors consume one row for each other. Thus ERT total has 16 columns and 49 rows. For every rule, the ERT defines the location. Designer can read these locations to find out the violated IP. For example, if designer read the row 1 of the ERT, he can know the Master 3 violates rule 1.

3.4 Windowed Trace Buffer

Traditional trace mechanisms are based on the forward tracing, which designer set one or more trigger points, the trace will begin when bus signals match trigger value. However, sometimes designers need to understand the signals before some errors occurred. Thus, the backward tracing mechanism is needed to capture the history signals especially in program and protocol error debugging.

In general backward tracing, designer use a circular trace buffer to store the trace data until first error occurred, which the trace depth depends on trace buffer size as shown in Fig. 6. Thus, if we have multiple errors to debug, then we need to do multiple backward tracings to debug all errors, which may consume quite a lot of debugging times. In case of protocol check debugging, we only need recent few cycles of trace data to find where the error comes from. Hence, we propose windowed trace buffer to address this problem as illustrated in Fig. 7. The main idea of windowed trace buffer is that we record recent few cycles trace data while every error occurred. The trace buffer not only records first error trace data, but also records other error trace data as much as we can.

Figure 7 shows windowed trace buffer architecture. Trace data buffer stores bus signals for debugging and stores current ERROR signals to indicate what errors occurred. For our observation, we require to capture 194-bits bus sig-
nals and three cycles of trace data for debugging. All of the required signals are summarized in Table 1. As shown in Fig. 7, 194-bits bus signals pass through a three-stage shift registers to capture recently three cycle bus signals. The trace buffer write controller (TBWC) receives Error Flag, when Error Flag is asserted, then the TBWC asserts DataWrite for three cycles that we can store latest 3 cycle bus signals and ErrorWrite for one cycle that we can store what errors occurred. In conclusion, we need the total 655 \((194 \times 3 + 73 = 655)\) bits per error occurred. If we use 1 KByte memory as trace buffer, we can store up to 12 error occurrences, which can significantly reduce the debugging time than traditional backward tracing mechanism.

4. Verification Strategy for the HPChecker

We use three methods to verify our HPChecker as follows. 1) RTL trigger to generate bus signals; 2) Pre-compiled cycle-level simulation patterns; and 3) Synopsys VIP. The first two verification strategies are buggy verification plans. We construct the wrong condition case by case. If the rule case is constructed, the protocol checker module will observe these signals. While the protocol checking module detect that it is a protocol violation then it will assert a corresponded error flag to tell the reference table and history memory.

- **RTL trigger to generate bus signals**: Behavioral buggy masters and slaves connect to the bus and then the HPChecker will observe these signals.
- **Re-compiled cycle-level simulation patterns**: We use software to generate the corresponding condition. The software generates a ROM file and the verification module base on the content of the file periodically delivers these signals to the bus. Then the HPChecker will determine whether these signals violate the protocol or not.
- **Synopsys VIP**: The software is available from the Synopsys. It can also verify the IP protocol generate the correct behavior of IP. We use the generated correct behavior to correct our HPChecker. This step is used to avoid misunderstanding on AMBA AHB bus protocols or some unconsidered conditions.

We run the simulation on the ARM EASY (Example AMBA System) environment [23] as shown in Fig. 8. For each rule, the constructed case will add to the bus as a master or slave IP and then run the simulation.

5. Experimental Results

The proposed HPChecker has been successfully integrated into two SoC environments as case studies. One is the tile-based 3D graphics acceleration SoC and the other is the LEON2 SoC. The HPChecker detected some bus protocol violations and ineffective transfers during the progress of the 3D graphics SoC design. No protocol violations are detected when we ran a test-bench in LEON2 SoC environment. This section demonstrates the violation condition and caused reason that run in FPGA and RTL simulation environment in details as follows.

- **3D graphics accelerator SoC environment (FPGA environment).**
- **LEON2 SoC environment (RTL environment).**

5.1 Case Study 1: 3D Graphics Acceleration SoC

Figure 9 shows the architecture of our developed 3D graphics acceleration SoC [24]. The 3D graphics acceleration SoC has three main modules: 1) geometry module (GM); 2) tile divider (TD) module; and 3) rendering module (RM). All of three modules do the 3D graphics computation independently and use handshake to communicate with each other. The HPChecker plays a role of protocol checker in the environment. The HPChecker has detected two protocol violations.
violations and two performance warnings during the execution on this environment. These violations and warnings are described subsequently.

5.1.1 Case 1

In the first case, the GM caused a violation. According to the AMBA AHB specification [25], the HBUSREQ signal is sent out by the master to request access to the bus. If a master asserts the HBUSREQ signal, then it indicates that it wants to perform a transfer. Thus it must assert the HBUSREQ signal until it has granted to access the bus.

As shown in Fig. 10, a master asserts the HBUSREQ signal during T0 to T2 and is granted at T2 and T3. Therefore, it performs a transfer at T3 and T4 (a transfer begins after the master is granted). It is successful transfer. However, in the dotted part of the Fig. 10, it is a wrong condition. The GM module de-asserts the HBUSREQ signal at T5 without being granted. It just completes two transfers (HGRANT# is 1 at T2 and T3 in dotted part) and the remainder request part (HBUSREQ# is 1 at T3, T4, and T5 in the dotted part) is not finished. The behavior is ambiguous because we cannot make sure that the remainder ungranted part whether is truly necessary for the master or not.

5.1.2 Case 2

In the second case, the RM caused a violation. For the defined length burst, the master can de-assert the HBUSREQ signal once the master has been granted the bus for the first transfer.

As in Fig. 11 HBUSREQ shown, de-asserted after T3. This can be done because the arbiter is able to count the transfers in the burst and keep the master granted until the burst is completed. The RM needs to perform an INCR 16 transfer. It can only perform the first beat transfer successfully and the other fifteen beats transfer is failed when runs in FPGA. The HPChecker finds out the RM that cannot complete the burst transfer. The reason why the RM cannot complete the transfer that the RM de-asserts the HBUSREQ is that it is too early to complete the transfer. The RM de-asserts the HBUSREQ signal immediately when it is granted, like Fig. 11 HBUSREQ# signal. Then the arbiter will do the arbitration to decide transfer. The arbiter decides which master can use the bus. The RM could only grant it for a single transfer. This is why the RM can only succeed for the first transfer and the other burst transfers are failed. For the modification, the HBUSREQ signal is de-asserted after it has been granted the bus for the first transfer, the INCR16 burst transfer can work correctly in the FPGA design flow.

5.1.3 Case 3

The efficiency of transferability is also an important issue in an SoC design. The HPChecker provides two rules to detect ineffective transfers of the AHB on-chip bus. The first rule is to detect the ineffective transfer before the effective transfer.

For example, as shown in Fig. 12, a master performs a transfer. The effective transfer is that the master starts NON-SEQ transfer after being granted the bus. The ineffective transfer is that the master performs additional IDLE transfer before NONSEQ, like the HTRANS# at T4 in the Fig. 12.
The reason of ineffective transfer may be that the designer is not familiar with the timing of the AMBA AHB and runs the dummy transfer.

The second rule in the HPChecker detected the ineffective transfer is that the HBUSREQ is de-asserted too late. The AMBA AHB arbiter samples HBUSREQ signal at the rising edge and then based on the arbitration policy to decide which master can access the bus. As shown in Fig. 13, the HBUSREQ# is de-asserted after the NONSEQ transfer. So the arbiter will grant the master for additional two cycle transfers. If the master just wants to perform a single transfer, then it will waste other unwanted transfer. If this master has the highest priority, then it will waste two IDLE transfers for each transfer.

5.1.4 Case 4

The fourth case is that the RM cannot complete a fixed length burst transfer. As shown in Fig. 14, these two figures show the result that run in FPGA verification flow. The Fig. 14 (a) is correct result. The RM can perform an INCR8. However, in the Fig. 14 (b), the HPChecker detects that the RM does not complete all beats of an INCR8 burst transfer, so the executed result is not correct. According to the AMBA AHB specification, for a fixed length burst, the master can de-assert the HBUSREQ signal since the master has been granted the bus for the first transfer. This can be done because the arbiter is able to count the transfers in the burst and keep the master granted until the burst completion. The RM de-asserts the HBUSREQ signal as soon as granted. The behavior causes RM can only perform in an INCR8 burst. After the RM de-asserts the HBUSREQ signal since the RM has been granted the bus for the first transfer, it can work correctly, like Fig. 14 (a) shown.

5.2 Case Study 2: LEON2 SoC

The LEON2 [26] SoC model implements a 32-bit processor conforming to the IEEE 1754 architecture. It is designed for embedded applications. New version can easily be added using the AMBA AHB/APB on-chip buses. The VHDL model is fully synthesizable with most synthesis tools and can be implemented on both FPGAs and ASICs. Simulation can be done with all VHDL-87 compliant simulation. The block diagram of the LEON2 is shown in Fig. 15.

The HPChecker is added to the LEON2 environment and executes the AHB bus protocol checking. We use the TB_FULL, which is provided by LEON2 as a test-bench. The TB_FULL is combined memory and functional tests, suitable to generate test vectors for manufacturing. It spends 297,001 cycles on running TB_FULL test-bench. There are no bus protocol violations detected in the LEON2.

5.3 Comparison and Synthesis Results

Finally, we also compare Synopsys VIP [27] and LEON3 AMBAMON [28] with HPChecker, as shown in Table 2. The Synopsys VIP and the LEON3 AMBAMON are unsynthesizable and only output error messages when errors.
occurred. The HPChecker is a synthesizable hardware module and we provide ERT and windowed trace buffer to achieve efficient debugging. The GUI analysis software is also developed called HPChecker analyzer. Figure 16 shows the execution screen of the HPChecker analyzer, HPChecker analyzer is a GUI interface software that can analyze the ERT, then display the violation information in details. Figure 16 (a) shows the total violation rules and indicates which masters or slaves are violated the rules. Figure 16 (b) shows detailed violations. Table 3 shows HPChecker synthesis results, the total gate count is 43,432 gates in TSMC 0.18 µm TSMC 1P6M process.

6. Conclusion

We have proposed a synthesizable AMBA AHB protocol checker, named HPChecker. The HPChecker contains 73 related bus protocol rules. By using these rules, the HPChecker can observe the AHB bus signals and check whether these transactions obey the protocol or not. If these transactions violate the AHB protocol, then the HPChecker will record the violating activity. The designer can use software to read these register contents. The HPChecker is a rule-based bus protocol checker. We have classified these rules according to the categories of violations. The purpose of the classification is that we can modify the source code of the HPChecker easily and fast. The designer just needs to extend the width of some output signals and modify the memory allocation space of the reference table. This design makes the HPChecker more flexible. Finally, the HPChecker successfully identifies protocol violations in the FPGA prototyping. The HPChecker has also been integrated into a 3D graphics acceleration SoC, and the gate counts of the HPChecker are 43,432 gates and its speed is 203 MHz in 0.18 µm TSMC 1P6M process.

Acknowledgements

This work was supported by National Science Council (NSC) of Taiwan, R.O.C. under contrasts NSC-97-2220-E-110-002 and NSC-97-2221-E-110-076-MY3. The authors would like to thank Temento Systems Inc., Grenoble, France, and Global Unichip Inc., Shinchu, Taiwan, R.O.C. for their generous financial and technical supports, and thank Ms. Jinrui Huang from the University of California at Irvine, CA, U.S.A. for proofreading the draft.

References

[1] M. Kreeting and P. Bricaud, Reuse methodology manual: For system-on-a-chip designs, Kluwer Academic Publishers, New York, 2002.
[2] P. Rashinkar, P. Paterson, and L. Singh, System-on-a-chip verification methodology and technologies, Kluwer Academic Publishers, New York, 2001.
[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” ACM Trans. Programming Languages and Systems (TOPLAS), pp.244–263, 1986.
[4] P. Chauhan, E.M. Clark, Y. Lu, and D. Wang, “Verifying IP-core based system-on-chip designs,” Proc. Twelfth Annual IEEE International ASIC/SOC Conference, pp.27–31, Sept. 1999.
[5] S. Campos, E. Clarke, W. Marrero, and M. Minea, “Verifying the performance of PCI local bus using symbolic techniques,” IEEE International Conference in Computer Design, Oct. 1995.
[6] F. Aloul and K. Sakallah, “Efficient verification of the PCI local bus using boolean satisfiability,” Proc. International Workshop on Logic Synthesis (IWLS), 2000.
[7] A. Mokkedem, R. Hosabetta, M. Jones, and G. Gopalakrishnan, “Formalization and analysis of a solution to the PCI 2.1 bus transaction ordering problem,” Formal Methods in System Design, vol.16, 2000.
[8] L. Ivanov and R. Nunna, “Specification and formal verification of interconnect bus protocols,” Proc. 43rd IEEE Midwest Symposium on Circuits and Systems, Aug. 2000.
[9] A. Roychoudhury, T. Mitra, and S.R. Karri, “Using formal techniques to debug the AMBA system-on-chip bus protocol,” Proc. IEEE/ACM Design, Automation and Test in Europe Conference and Exhibition (DATE’03), 2003.
[10] K. Shimizu, D.L. Dill, and C.-T. Chou, “A specification methodology by a collection of compact properties as applied to the Intel Itanium processor bus protocol,” Proc. 11th IFIP WG 10.5 Adv Research Working Conference on Correct Hardware Design and Verification Methods, pp.340–354. LNCS 2144, Springer, 2001.
[11] G. Borriello, L. Lavagno, and R.B. Ortega, “Interface synthesis: A vertical slice from digital logic to software components,” Proc. International Conference on Computer-Aided Design, Nov. 1998.
[12] A. Rajawat, M. Balakrishnan, and A. Kumar, “Interface synthesis: Issues and approaches,” Proc. 13th International Conference on VLSI Design, Jan. 2000.
[13] V. D’silva, S. Ramesh, and A. Sowmya, “Synchronous protocol automata: A framework for modeling and verification of SoC commu-
Liang-Bi Chen received the B.S. and M.S. degrees in Electronic Engineering from National Kaohsiung University of Applied Sciences, Kaohsiung, Taiwan, R.O.C., in 2001 and 2003, respectively, and is currently working toward the Ph.D. degree in Computer Science and Engineering at National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C. From August 2008 to September 2008, he had an internship of Department of Computer Science at National University of Singapore, Singapore. He was a visiting researcher in Department of Computer Science at University of California, Irvine, CA, U.S.A., between September 2008 and August 2009. His research interests include power/performance analysis for embedded mobile applications and devices, power-aware embedded systems design, low-power systems design, SoC debugging, and system-level design space exploration. He is a student member of the ACM, IEEE, and Taiwan IC design Society.

Jiun-Cheng Ju received the B.S. degree in Electrical Engineering from National Chiao-Tung University, Hsinchu, Taiwan, R.O.C. in 2003, and is currently working toward the M.S. degree in Department of Computer Science at National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C. His research interests include AMBA-based on-chip bus verification, debugging, testing, and performance analysis.

Chien-Chou Wang received the B.S. and M.S. degrees in Computer Science and Engineering from National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C. in 2004 and 2008, respectively. He is currently a Senior Engineer at Sitronix Technology Ltd., Taipei, Taiwan, R.O.C. His research interests include AMBA-based on-chip bus verification, debugging, testing, and performance analysis.

Ing-Jer Huang received the B.S. degree in Electrical Engineering from National Taiwan University, Taiwan, R.O.C., in 1986, and the M.S. and Ph.D. degrees in Computer Engineering from the University of Southern California, U.S.A., in 1989 and 1994, respectively. He is currently a professor in the Department of Computer Science and Engineering at National Sun Yat-Sen University, Taiwan, R.O.C. His research interests include computer architecture, SoC design, design automation, system software, embedded systems, and hardware/software co-design/verification. He has actively involved in academic, educational and industrial activities. He has extensive collaboration with several IC related industries. He is a member of the IEEE and ACM.