Hardware Debugging Method Based on Signal Transitions and Transactions

Nobuyuki Ohba Kohji Takano

IBM Research, Tokyo Research Laboratory, IBM Japan Ltd.
Yamato city, Kanagawa, Japan 242-8502
Tel: +81-46-215-4547
Fax: +81-46-273-7413
e-mail: {ooba, chano}@jp.ibm.com

Abstract - This paper proposes a hardware design debugging method, Transition and Transaction Tracer (TTT), which probes and records the signals of interest for a long time, hours, days, or even weeks, without a break. It compresses the captured data in real time and stores it in a state transition format in memory. It can be programmed to generate a trigger for a logic analyzer when it detects certain transitions. The visualizer, which shows the captured data in the matrix, timing-chart, and state-transition diagram formats, helps the engineer effectively find bugs.

I. Introduction

System On Chip (SoC) design is widely used to boost the performance, lower the power consumption, and reduce the overall system costs by integrating many resources. In ASIC/SoC design, however, growing design complexity has forced engineers to tackle deeper bugs, and thus the development requires more work in the test and debugging. Hardware prototyping is widely used [1, 2] for accelerating the work. It sure is a powerful tool giving outstanding test speed, which is usually 100 to 100,000 times faster than software simulation. It even allows the engineer to run real firmware, an operating system, and applications [3, 4].

As the size and complexity of ASIC/SoC increases, many more test cases are required to achieve sufficient test coverage in the verification. In addition, long-running tests taking hours or even days are needed to remove bugs from the product. In such testing and debugging, engineers often face difficulties in identifying what causes a bug. This is especially the case if the error is intermittent and not reproducible, as when the test program sometimes does and sometimes does not generate the error.

To trace the behavior of signals in a hardware prototype, designers normally use a logic analyzer and FPGA built-in signal trace tools, such as the Xilinx ChipScope [5] or Altera SignalTap [6]. The logic analyzer is a powerful tool for debugging since it allows the engineer to see what is happening in the target hardware in real time. However, the logic analyzer also has weak points:

- A logic analyzer has a limited amount of memory, so that it records signal behavior for only part of a test run, as shown in .
- It is not always obvious to the engineer as to which trigger conditions will pinpoint the source of the bug.
- Human designers find it difficult to fully understand the large amount of collected data in the timing-chart format.

During hardware debugging, we occasionally came across problems, which are hard to solve by using conventional approaches. Let us show three typical problems:

1) I ran a test program on the prototype board and got an error. Running the test program again, there was no error. I ran it ten more times, but still had no error. Where is the error gone?
2) I connected my new core to Design X from Company Y. I ran a test program and got an error. We only have a minimal data sheet for Design X. According to the specifications, my core should work. Why not?
3) I am using a logic analyzer to trace an error, but I have no idea as to what kind of trigger condition I have to set.

To address these problems, we have been developing a hardware debugging tool named Transition and Transaction Tracer (TTT). Our experiences in hardware development show that the target ASIC/SoC is well verified for the transactions that occur frequently. However, the ASIC/SoC tends to have potential flaws in processing transactions that rarely happen. For this reason, TTT captures the time varying signals as a series of vectors, and records them for a long time, such as for hours, days, or even weeks without a break. TTT constantly monitors the transition counts between states to help the engineer effectively find the problem. When it detects a new or unexpected transition, it calls the attention of the engineer by generating a trigger for the logic analyzer.

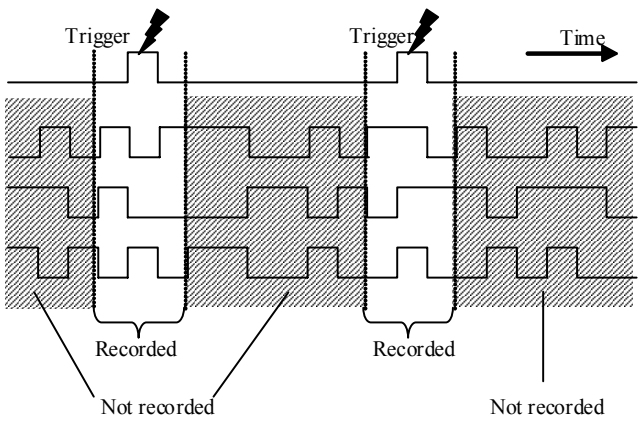

Fig. 1. Signal transitions recorded by a logic analyzer

0-7803-9451-8/06/$20.00 ©2006 IEEE.

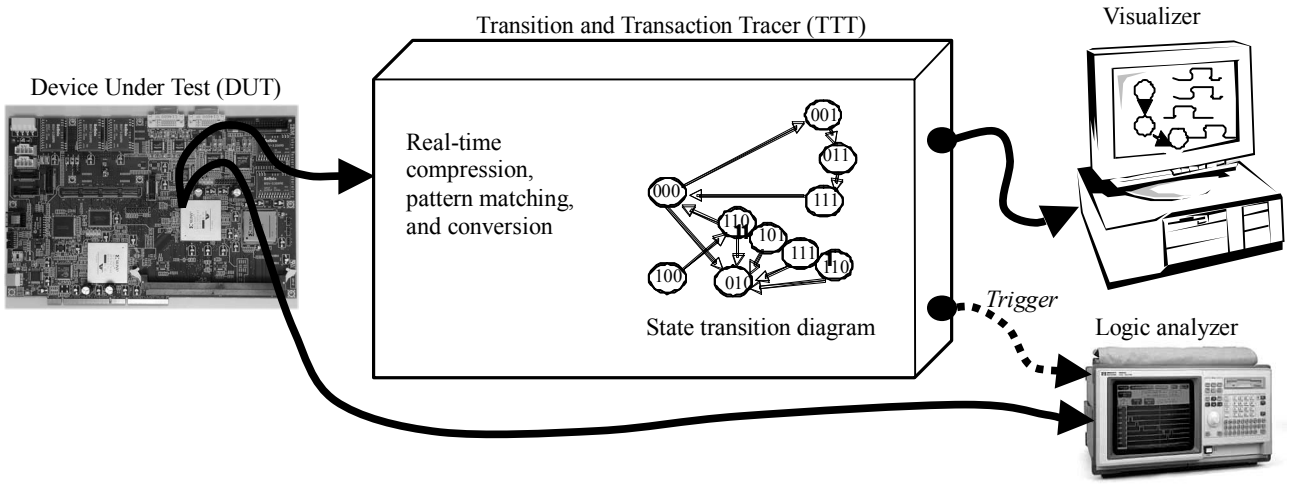

Fig. 1. Debugging System Overview

To help engineers easily perceive what happens in the target hardware, TTT shows the signal behavior not only in the timing-chart format but also in the state transition diagram. To automatically identify and extract a transaction from the signal transition sequence, we developed an idle state detector, which splits two adjacent transactions.

This paper is organized as follows: Section 2 describes the hardware architecture of TTT; Section 3 shows an implementation of TTT using an FPGA prototyping board; Section 4 shows the results obtained by running TTT on a PCI bus; and Section 5 offers concluding remarks and discusses future work.

II. Hardware Architecture of Transition and Transaction Tracer

A. Overview of the debugging system

TTT captures the transitions of the target signals and records them in a state transition form for a long time, for hours, days, or even weeks, without a break. This greatly decreases the possibility of missing the signal behavior associated with an error. In contrast to a conventional logic analyzer, TTT generates a trigger for signal capture when any new transition is detected.

Fig. 1 is the overview of the typical debugging system, which consists of a Device Under Test (DUT), Transition and Transaction Tracer (TTT), a visualizer of TTT, and a logic analyzer. TTT probes the target signals of the DUT, captures the signal transitions, compresses the data in real time, performs matching against the state transitions already stored, and sends the analyzed data to the visualizer. It also has a trigger generator connected to the logic analyzer.

B. Internal Structure of Transition and Transaction Tracer

Fig. 2 shows the internal structure of TTT. It is composed of a transition recorder and transaction tracer. The transition recorder focuses on the transitions between pairs of adjacent states. The transition tracer, on the other hand, has a higher view of completed transactions, where the transactions are delimited by idle states.

The transition recorder consists of a transition memory (U1) and a counter memory (U2). It captures the state transition between adjacent states, which is the vectors at Clock N-1 (previous state) and Clock N (current state). The vector pair is stored in the transition memory. The counter memory records the number of each transition between adjacent states. If the transition has not yet been stored in the transition memory, the transition recorder generates a trigger. If the transition has already been stored, the transition recorder increments the counter that is associated with the captured transition.

Fig. 3 is a user interface example, which shows the state transitions in a two-dimension matrix format. In the figure the number 43,123 (marked with *) is the transition count from state A to state B. The cell is automatically shaded because the transition count exceeds the user specified threshold, 10,000 in this case. In like manner, the number 3 (marked with **) shows the transition count from states D to B. The cell color is reversed because the transition count is lower than the user-specified threshold of 10, helping the user easily spot the rare transitions.

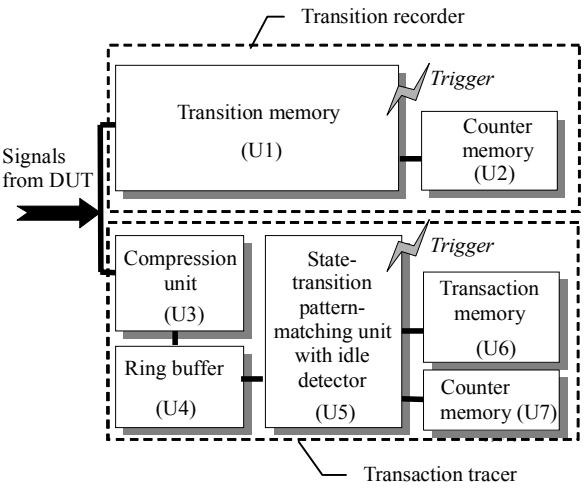

Fig. 2. Internal Structure of Transition and Transaction Tracer

State at Clock N-1 \ State at Clock N	A	B	C	…
A	121	43,123*	0	…
B	0	324	0	…
C	815	0	132	…
D	0	**3****	43	…
…	…	…	…	…

High threshold = 10,000 Low threshold = 10	(defined by the user)

Fig. 3. Visualization example in the matrix format

The transaction tracer captures the signal behavior as a transaction, which is a series of states between idle states. The idle states are defined by the user or by the idle state detector. The transaction tracer consists of a compression unit (U3), a ring buffer (U4), a state-transition pattern-matching unit with an idle state detector (U5), a transaction memory (U6), and a counter memory (U7). The compression unit carries out run length coding to compress the captured data. The state-transition pattern-matching unit searches for the captured transaction pattern in the transaction memory. If it finds the transaction to be new, it stores the data in the transaction memory. The counter memory stores the number of each specific transactions observed. The state transition diagram drawn in the visualizer is constructed by using the information stored in the transaction memory.

Table 1 shows an example of the compression procedure. The transaction is defined as a series of states starting with the transition from the idle state to a non-idle state and ending with the transition from a non-idle state to the idle state. The run-length expression in the table shows how the input states are compressed in the run-length form. This is based on blocks of repeated states and their repetition counts.

The compressed data is stored in the transaction memory. The transaction tracer uses hashing for the search and store operations. Since transactions vary in length, the compressed data are stored in a linked list form.

C. Idle State Detector

In the first version of TTT, the idle state was defined by the user. By experiences of debugging work, we have learned that the idle state can easily be detected in many cases. The bus is not busy all the time; rather it frequently "hovers" over the idle state, in which the signals tend to keep the constant value. From a power saving perspective, it is a good idea to keep the signals unchanged in the idle state, and many hardware designs adhere to this design scheme.

If the input signals are unchanged more than C cycles, the idle state detector registers the state as IDLE, where C is a user defined integer ($C \geq 2$). By definition, one or more states can be registered as IDLE. The transaction tracer uses this state or these states to find a series of signal transitions between the idle states and records them as a transaction.

Table 1: An example of a compression procedure

Sampling time	Input state	Direct expression	Run-length expression
+0	<idle>		
+1	A	A	A
+2	B	AB	AB
+3	C	ABC	ABC
+4	B	ABCB	ABCB
+5	C	ABCBC	A(BC)1
+6	B	ABCBCB	A(BC)1B
+7	C	ABCBCBC	A(BC)2
+8	D	ABCBCBCD	A(BC)2D
+9	<idle>		

III. Hardware Implementation

We implemented the transition and transaction tracer using a custom-made FPGA prototyping board. Fig. 4 is a photograph of the board. It has two Xilinx VirtexII XC2V4000 FPGAs and a card-edge PCI interface, so that it can be installed in a PCI slot.

We made the design in VHDL and synthesized it for the FPGA configuration data using Xilinx ISE version 6.3. The transition memory (U1) in Fig. 2 is a fully-associative memory. The other memories and counters (U2, U6, and U7) use a two-port 18 Kb block SRAM provided in the FPGA [7]. All the counters are 36 bits wide. Table 2 shows the number of FPGA slices and 18 Kb SRAM modules used for the implementation. In the table, the capacity is the maximum number of transitions that can be stored in the transition memory (U1). The transition recorder uses many more slices than the transaction tracer. This is because the transition memory is a fully-associative CAM running as fast as the DUT signals. The CAM is implemented by using the primitive latches and comparators of the FPGA. The depth is the maximum recordable block length of a repetition. In Table 1, for example, the block length is two for (BC).

The amount of hardware resources for implementing the transition and transaction tracer depends on the capacity and depth, both of which are related to the complexity of the target signal behavior, but not to the usage duration.

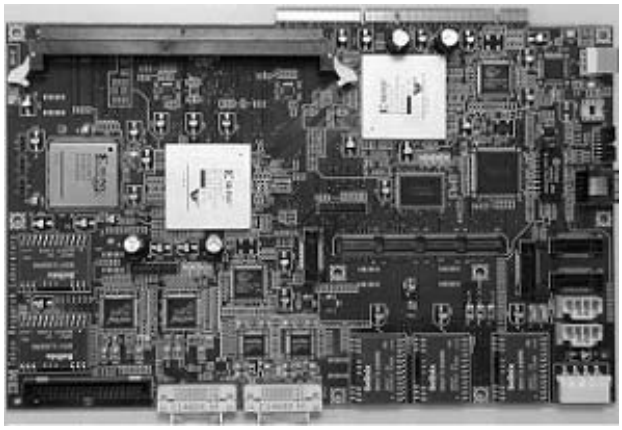

Fig. 4. FPGA prototyping board

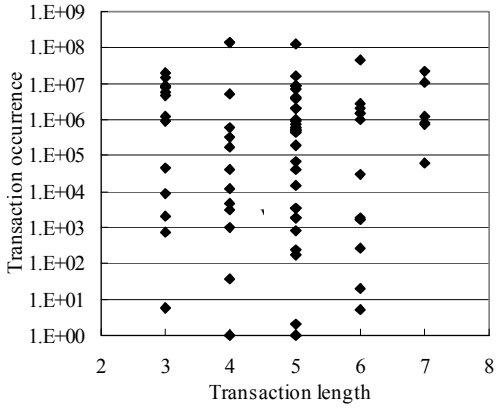

Fig. 5. Occurrence of PCI transactions

Table 2: Hardware resources

(1) Transition recorder		
Capacity	Slices	SRAM modules
64	2,720	2
128	3,065	2
256	4,118	2
512	12,871	2
1,024	26,705	4

(2) Transaction tracer excluding transaction memory		
Depth	Slices	SRAM modules
2	84	0
3	137	0
4	207	0
5	288	0
6	382	0
7	492	0
8	621	0
9	767	0
10	928	0

(3) Transaction memory		
Capacity	Slices	SRAM modules
512	315	2
1,024	315	3
2,048	315	5
4,096	315	9
8,192	315	17
16,384	315	33

(4) Idle state detector		
C	Slices	SRAM modules
31	674	3

IV. Evaluation

To see the effectiveness of the proposed method, we captured the behavior of a 32-bit 33 MHz PCI bus. The capacity of the transition recorder we chose for this application was 64. The depth and capacity of the transaction tracer were 4 and 1,024, respectively. By using the speed optimized option in the synthesizer, the Virtex-II XC2V4000-6 FPGA can run at up to 100 MHz.

In addition to the FPGA board, a network card and an IDE control card were resident in the PCI bus. Ten PCI control signals, FRAME#, IRDY#, TRDY#, DEVSEL#, STOP#, GNT#, C/BE#[3,2,1,0], were monitored.

Firstly, the bus state was manually defined as IDLE if FRAME#, IRDY#, and TRDY# were all inactive. All of the PCI transactions were monitored for about an hour. During the test run, 78 types of transactions were captured. Fig. 5 shows the occurrence counts of the recorded transactions. The transaction length is defined as the number of the states between the idle states. The figure shows that the occurrence count for the most frequently captured transaction was more than 10^8, and a transaction that happened only once was also recorded.

Without any background knowledge, it is very hard for conventional methods to capture such a rare transaction. The method proposed in this paper records all of the transactions and display them in the state transition diagram, which helps the engineer clearly understand what is happening in the target hardware.

Secondly, we engaged the idle state detector. Value 31 was assigned to C, and therefore the states that stay unchanged for more than 30 cycles were automatically defined as IDLE. After two hour test run, two states shown in Table 3 were registered as IDLE.

This result conforms to the PCI specification [8]. FRAME#, IRDY#, TRDY#, DEVSEL#, and STOP# are all sustained tri-state[1] signals, and therefore they must be in H state for the idle cycle. C/BE#[3,2,1,0], on the other hand, are tri-state signals and can be left H or L after a transaction is completed.

To help the engineer understand the target behavior, we developed a visualizer, which runs on Windows and Linux. It shows the signal transitions in three formats: 1) transition matrix, 2) timing chart, and 3) state transition diagram. The transition matrix, as illustrated in Fig. 3, is a two dimensional matrix, which shows the transition counts for adjacent states. Fig. 6 is a screen shot of the visualizer, which shows a state transition diagram and a timing chart. While TTT is monitoring the signals, the state transition diagram is generated on the fly. Newly recorded transitions are colored red to be easily identified.

To increase the bus bandwidth, address pipelining is used in several bus protocols, such as IBM CoreConnect and ARM AMBA. By definition, the address phase begins before the previous data phase is completed. In such a case, a traced transaction path delimited by an IDLE sate will contain two or more transactions. Although the state transition diagram becomes bigger, it still gives important information on how two or more transactions are overlapped. To make it easier for the user to distinguish address and data phases, the visualizer can color the states in accordance with the specified signal status.

Table 3. Idle states

Signal	IDLE1	IDLE2
FRAME#	H	H
IRDY#	H	H
TRDY#	H	H
DEVSEL#	H	H
STOP#	H	H
GNT#	H	H
C/BE#[3]	H	L
C/BE#[2]	H	L
C/BE#[1]	H	L
C/BE#[0]	H	L

(H: high voltage, L: low voltage)

[1] Sustained Tri-State is an active low tri-state signal owned and driven by one and only one agent at a time. The agent that drives a sustained tri-state pin low must drive it high for at least one clock before letting it float.

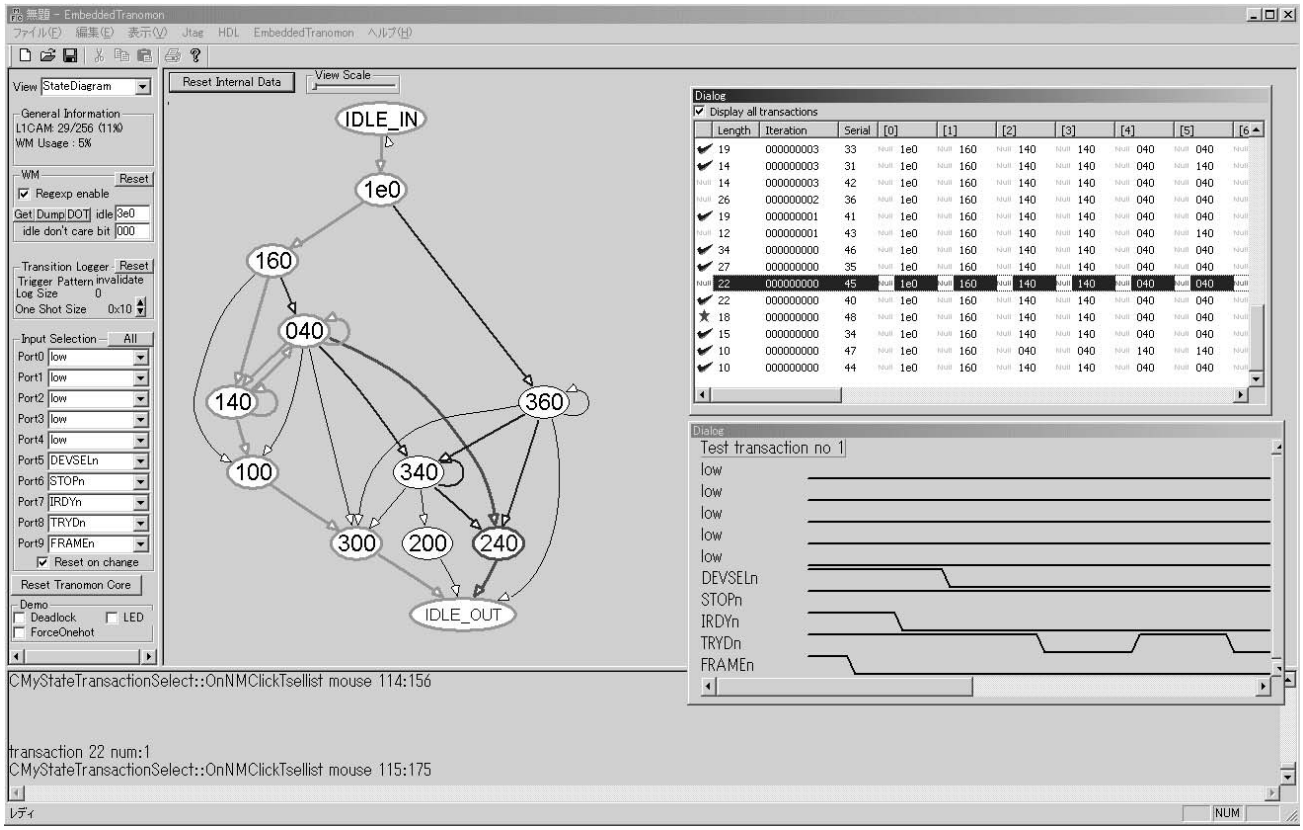

Fig. 6. Screen shot of the visualizer

V. Concluding Remarks

This paper proposes a new hardware debugging method, a transition and transaction tracer, which focuses on the signal transitions and transactions to accelerate ASIC/SoC debugging. We implemented it using a prototyping FPGA board and evaluated the hardware resources and operating speed. A state transition diagram for PCI transactions was shown as an example.

To review, here are the problems listed in the introduction and how we tackled them:

1) I ran a test program on the prototype board and got an error. Running the test program again, there was no error. I ran it ten more times, but still had no error. Where is the error gone?

A1) With TTT, I probed the crucial signals and recorded all the transitions in each test run. The state transition diagram from the test that caused an error has a transition that does not appear in the error-free state transition diagrams. It turned out that the transition occurred only when three devices simultaneously requested a bus. TTT provided a hint to solve the problem.

2) I connected my new core to Design X from Company Y. I ran a test program and got an error. We only have a simple data sheet for Design X. According to the specifications, my core should work. Why not?

A2) TTT showed that the error occurred in a specific situation that is not described in the data sheet of Design X. My core was never expected to encounter this behavior. I contacted an engineer from Company Y and we sorted out the problem.

3) I am using a logic analyzer to trace an error, but I have no idea as to what kind of trigger condition I have to set.

A3) I configured TTT to trigger the logic analyzer to record data for each new transition. By analyzing the captured data in the logic analyzer, I found that the error was associated with a very rare transition event, which appeared roughly once an hour.

Design and verification engineers often delude themselves that a certain input or transition is impossible. Looking at the state transition diagram created with TTT, we sometimes discover the reality is different from what we had believed. We have been using TTT for practical ASIC development and succeeded in accelerating our debugging work.

We are now developing a new function that incorporates TTT with an assertion-based verification method [9]. The assertion specifies state transitions that must not happen (illegal transitions) and those that must happen (transitions that must be tested) in the test run. We define these transitions in the TTT prior to the test run. TTT gives warning to the engineer if it detects an illegal transition. It also records the number of captured states and transitions in each test run for measuring the test coverage.

We are also studying how TTT gives more useful information to the user in pipelined and split-transaction buses.

References

[1] N. Ohba and K. Takano, "An SoC Design Methodology Using FPGAs and Embedded Microprocessors," *Proceedings of Design Automation Conference*, pp.747-752, 2004.

[2] J. O. Hamblen, "Rapid Prototyping Using Field-Programmable Logic Devices," *IEEE Micro*, pp.29-37, 2000.

[3] T. Matsumura, N. Yamanaka, R. Yamaguchi and K Ishikawa, "Real-time Emulation Method for ATM Switching Systems in Broadband ISDN," *Proceedings of IEEE International Workshop on Rapid System Prototyping*, pp.19-21, 1996.

[4] M. Courtoy, "Rapid System Prototyping for Real-Time Design Verification," *Proceedings of Ninth International Workshop on Rapid System Prototyping*, pp.108-112, 1998.

[5] Xilinx Inc., "Chipscope Pro Software and Cores User Guide," October, 2004.

[6] Altera Inc., "Design Debugging Using the SignalTap II Embedded Logic Analyzer," December, 2004.

[7] Xilinx Inc., "Virtex-II Platform FPGA Handbook," December, 2001.

[8] PCI Special Interest Group, "PCI Local Bus Specification – Revision 2.2," December, 1998.

[9] Accellera, "Property Specification Language – Reference Manual," http://www.eda.org/ieee-1850/, version 1.1, 2004.