The Transaction-Based Verification Methodology

Cadence Berkeley Labs, Technical Report # CDNL-TR-2000-0825, August 2000.

Dhananjay S. Brahme, Steven Cox, Jim Gallo[1], Mark Glasser, William Grundmann, C. Norris Ip, William Paulsen, John L. Pierce, John Rose, Dean Shea, Karl Whiting

Cadence Design Systems, Inc.

Keywords verification, simulation, test bench, transaction, system-on-chip (SoC).

Abstract

This paper summarizes a transaction-based verification methodology (TBV) that makes functional verification of RTL descriptions using simulation more effective By raising the verification effort to a higher level of abstraction, an engineer can develop and diagnose tests from a system level perspective. This capability enhances the reusability of each component in the test benches. It improves the debugging and coverage analysis process by presenting information in terms of transactions and their relationships, rather than signals and waveforms.

Several designs have been verified using this methodology. It was found that TBV can be mastered by hardware engineers in a short time, and the teams were able to identify and fix design errors quickly.

1. Introduction

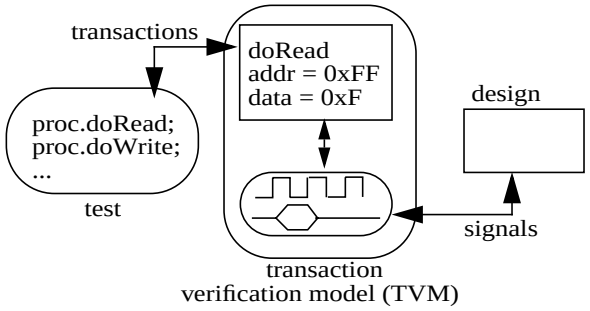
Developing an effective test suite for an HDL design is an arduous process. With exploding design sizes, we need to generate a large amount of high quality stimulus with a minimum of effort. While the test benches should be exercising the design thoroughly, they should also be self-checking to avoid manual confirmation of expected operation. It should be easy to check the activities in a simulation run to identify problems in the design. It should also be easy to analyze the resulting coverage of the design to access the quality of the test suites.

The transaction-based verification methodology (TBV) is designed to make it easy to create and reuse test benches, to debug simulation runs, and to analyze coverages by introducing the concept of transactions to the verification tools. It was introduced in 1998 [1,2,3] and integrated into a verification tool suite in 1999 [4,5]. The corresponding C++ library for transaction-based test bench authoring is released in 2000 as an open-source licensed software [6,7]. A similar work can be found in [8].

1. no longer with Cadence (work done while at Cadence Design Systems, Inc.).

In short, TBV uses the concept of transactions to raise the verification effort to a higher level of abstraction for the purpose of improved productivity. It fundamentally relies on separating a test bench into two layers, as shown in Figure 1. The top layer is the tests, which orchestrate transaction-level activity in the system without regard to the specific detail of signal-level protocols on the design's interfaces. The bottom layer is the transaction verification model (TVM), which provides the mapping between transaction-level requests made by the tests and detailed signal-level protocols on the design's interfaces.

Figure 1: A layered test bench



With the separation of responsibility into these two layers, many new and complex tests can be developed quickly, as the detailed protocols are already captured in the TVM layer. Although this separation of responsibility can be performed manually on any test bench authoring tool, it is much easier to use a test bench authoring tool that explicitly supports transactions, to speed up the learning process, to guide the user to model their test benches in an appropriate style, and to simplify the coding. This strategy allows reuse of individual components in the test bench. Similar discussion in reusability can be found in [9,12].

Furthermore, a better productivity gain can be obtained if the debugging tools and the coverage tools can communicate and work with the test bench authoring tools in terms of transactions. With these tools, a user can

- write new self-checking tests easily in terms of transactions, and pass this information to other tools,

- record and display transactions and establish relationships among transactions, to facilitate the debugging process,
- record and analyze coverage information in terms of transactions and their relationships.

A core part of this methodology has been implemented as a C++ library called TestBuilder [12], available with an open-source licence [10,11]. This library includes a text-based database for transaction recording and analysis. More sophisticated debugging and coverage tools are also available commercially.

Support for transaction-level test bench authoring and the supports for intuitive and automated transaction-level debug and coverage capabilities are central to TBV. In particular, this support is particularly useful for self-checking constrained random tests, where it is important to confirm that the algorithmic test generator is actually generating appropriate and difficult simulation scenarios.

A four-port ethernet packet router is used in this paper to illustrate various concepts in TBV. A system-on-a-chip (SoC) for Voice-over-Packet (VoP) application has been verified using TBV at Cadence Design Systems. Designers from several companies have been involved in the evaluation of the methodology.

2. Traditional functional verification approaches

A typical design process starts with the creation of a functional specification for the design. Once the functional specification has been completed, a test plan is created to specify the functionality that requires testing, at both the block and system levels.

The test plan usually includes coverage requirements as well as specific corner cases that require testing. The team then creates test benches, including both deterministic generators and checkers to verify design functionality. Test bench creation continues until test plan requirements have been satisfied.

The tests can be classified into three categories:

- deterministic tests that are written for a typical scenario or a specific corner case,
- directed random tests with constraints,
- trace-driven tests that utilize traces captured from real-world stimuli and responses.

The correctness of the designs is usually checked by

- executing high-level behavioral models in parallel with the design and comparing the outputs of both,
- executing protocol checkers that monitor the events at the interface and checking the conformance of the events to the protocol specification,
- executing event checkers that monitor the events internal to the design and checking the conformance of the events according to some ad-hoc scripts.

It is tedious and time-consuming to write deterministic tests, random tests, behavioral models, protocol checkers, and event checkers. In many cases, the tests are written to communicate to the design directly at the signal-level interface, requiring a lot of code for underlying protocols and making reuse of tests difficult. It is also difficult to use these tests to drive the system-level behavioral models, leading to ad-hoc use of alternate models, some with cycle-accurate details, some with ad-hoc comparison of system-level events and signal-level events. The protocol checkers and event checkers are typically confined to checking simple relationships among signals, with limited ability to check global relationships.

On the other hand, a system-on-chip design is typically conceived at the transaction level. For example, system architects do not start out thinking about the relationship between the enable pin and the address bus. They start their design by thinking about what kind of data flows through the system and how and where it is stored. The transaction-based verification methodology is a natural extension of the high-level design process, which allows deterministic tests and random tests to be written at the system level, the system-level behavioral models to be used directly, and the protocol and event checkers to be expanded to include global relationships.

3. Raising the level of abstraction to transactions

Raising the level of abstraction from signals to transactions facilitates the creation of tests, the debugging process, and the measurement of functional coverage. While the design works at the signal level with ones and zeros, a transaction-based methodology allows a hardware designer to create tests and work in a more intuitive way. In many ways, it allows the tests to interact with the design in the same way that humans interact with computers using keyboards and mice instead of binary punch cards.

There are three fundamental concepts in a transaction-based verification environment:

- transactions,
- transaction verification models (TVM),
- transaction-based tests.

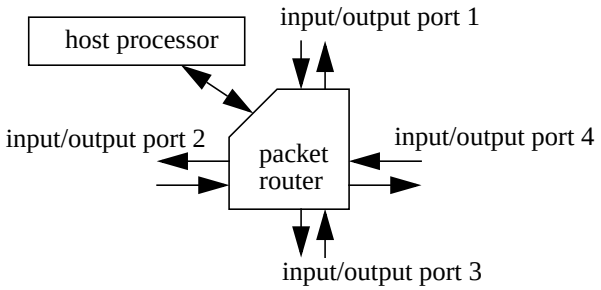
3.1 Transactions

A transaction is a single conceptual transfer of high-level

data or control between the test bench and the *design under verification* (DUV) over an interface. It is defined by its begin time, end time, and attributes. It can be as simple as a memory read or as complex as the transfer of an entire structured data packet through a communication channel. The transaction level is the level at which the design is architected and it is the level at which the design can be verified in a most effective way.

A four-port ethernet packet router, as shown in Figure 2, illustrates the concepts in this paper. The job of an ethernet packet router is to accept packets from the four input ports, determine the output port to which each packet should be sent, and send each packet to the corresponding output port. A host processor is connected to the router for configuration and performance monitoring.

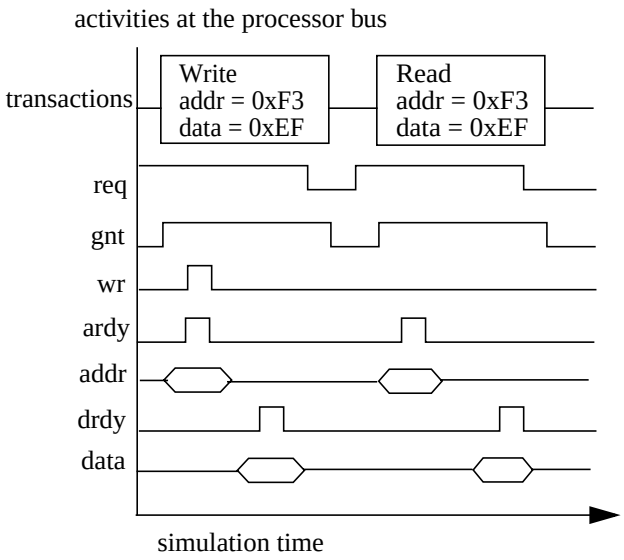
Figure 2: A four-port ethernet router



The host processor communicates to the router via read and write transactions. For example, a write transaction begins when the processor has successfully arbitrated for the bus and ends when the related actions are finished. It has two attributes, the data and the address to which the data is to be written, as shown in the top left transaction in Figure 3. A read transaction and a simplified version of the underlying activities at the signal level are shown in the bottom of the diagram.

In this simplified interface, there are seven signals and a clock signal for a non-pipelined bus. The processor asserts the signal *req* to initiate a new transaction. The transaction can proceed when *gnt* is asserted by the DUV. The signal *wr* is used to determine whether it is a read or a write transaction. When the transaction begins, the signal *ardy* is asserted and the processor puts the address in the address bus. The validity of the data at the data bus is indicated by the signal *drdy*, which can be asserted by the processor or by the design, depending on whether the transaction is a read or a write. By abstracting all this information into transactions, it becomes easier to write tests, debug a simulation run, and analyze the coverage.

Figure 3: Transactions vs. signals



3.2 Transaction Verification Models

A TVM serves as an abstraction layer between the test and the design. It is constructed as a collection of tasks, each of which executes a particular kind of transaction. A TVM connects to the DUV at a design interface. An interface is a collection of pins that move data to and from the design according to the requirements of the design's interface protocol. Since most designs have multiple interfaces, they also have multiple TVMs to drive stimulus and check results. TVMs are sometimes referred to by a variety of names, such as bus functional models (BFM) and transactors.

Given a set of attributes from a test, a task can initiate a transaction by arbitrating at the protocol level, pass on each attribute to the DUV at the appropriate time, determine other attributes according to the responses of the DUV, and terminate the transaction when the related actions are completed.

For example, a packet router is written in Verilog with signals *clk*, *req*, *ardy*, *addr*, *drdy*, and *data* declared as part of the input and output ports:

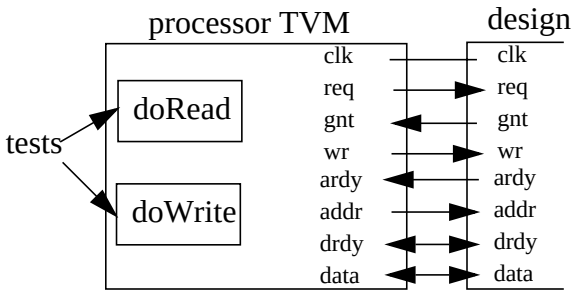
```
module design ( ... );
  ...
  input clk;
  input req;
  output gnt;
  input wr;
  input ardy;
  input [63:0] addr;
  inout drdy;
  inout [63:0] data;
  ...
endmodule
```

The signal-level activities corresponding to a read transaction may be generated by the following task (in a pseudo code format).

```
begin read task
  get address from the test;
  wait for 10 units of time;
  assert signal req;
  wait until signal gnt
     is asserted by the DUV;
  record the fact that a read
     transaction begins here;
  deassert signal wr to indicate a read
  assert ardy and set signal addr
     to the right address
  wait until signal drdy
     is asserted by the DUV;
  copy the value at signal data
     to a local variable;
  deassert signal req at the next
     positive edge of signal clk;
  wait for the signal gnt to be deasserted
     by the DUV.
  record the address as one attribute
      of the transaction;
  record the data as one attribute
      of the transaction;
  record the fact that the read
     transaction has ended
end task
```

Similarly, a write task can be used to generate the signal-level activities corresponding to a write transaction. The corresponding TVM will be a static structure in the test bench, with two tasks communicating to the DUV through the signals *clk*, *req*, *gnt*, *wr*, *ardy*, *addr*, *drdy*, and *data*, as shown in Figure 4.

Figure 4: A processor TVM



It is important to record the transactions with appropriate attributes in the simulation database. This facilitates the process of debugging and coverage analysis, as discussed in Section 5 and Section 6. For example, the first stage of the debugging process is to analyze the relationship of the transactions generated from the simulation run, without looking at the signal-level activities. When the area of concern has been narrowed down to a few related transactions, the second stage of the debugging process can expand those transactions into their corresponding signal-level activities to pin-point the bug in the DUV.

3.3 Transaction-based Tests

A test is a program that generates organized (direct or random) sequences of task invocations for each of the TVMs in the system. These TVMs, in turn, generate transactions into the DUV. A TVM is typically written once for a design interface, whereas new tests are constantly added to the test suite to exercise the TVMs and the design in different ways to cover different corner cases. A task can be invoked in a blocking fashion (run) or non-blocking fashion (spawn). Spawning a task dynamically creates a new thread of execution and supports task re-entrancy. Multiple copies of the same task can be spawned without regard to whether the previous invocation has finished or not. This allows specification of concurrent events to support pipelined protocols and other complex test scenarios.

In the ethernet router, the implementation of the read task indicates that only one transaction can be outstanding at a time. This can be achieved by waiting on a semaphore or a mutex in a TVM or by invoking the read and write tasks in a blocking fashion only.

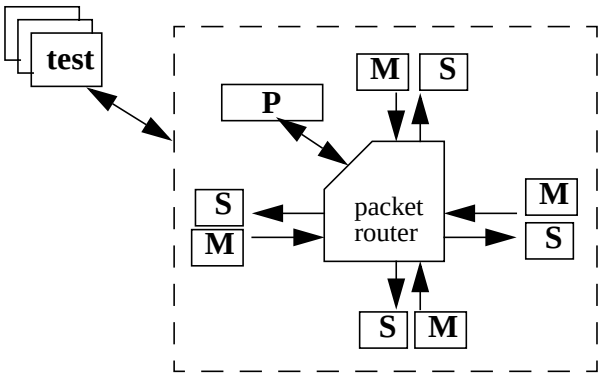
A simple test can be created in the following form

```
loop 1000 times
    select a task to invoke (read or write);
    select an address;
    if the write task is selected,
       select a value for the data;
    invoke the selected task with the address
       (and the data if applicable);
end loop
```

Similar tests can be written quickly without referring to the signals at the DUV interface, reusing the same TVMs in every test. Furthermore, in subsequent projects of similar designs, the tests can be reused. When a design has a different protocol than a previous design, the verification team can replace the TVMs with a set of new ones and reuse the high-level tests.

The overall transaction-based verification methodology is summarized in Figure 1. The communication between the tests and the TVMs is performed by task invocations and the communication between the TVMs and the DUV is performed by signal value changes. Figure 5 shows the corresponding picture for the 4-port ethernet packet router example with 7 TVMs and a set of tests. The block labeled P is the processor TVM described before; the blocks labeled M are master TVMs that generate packets to the router; and the blocks labeled S are slave TVMs that check the packets from the router. The tests can directly invoke the tasks in any of these TVMs. Typically, when a test invokes a task in a master TVM to generate a packet, it also invokes a task in a slave TVM to monitor the packet that should be coming from one of the output ports.

Figure 5: A test bench for a 4-port ethernet packet router



Furthermore, with the advent of system-level design tools, transaction-based tests can be developed in a system-level design tool for simulating system-level description, then be reused in the RTL simulation by adding appropriate TVMs to translate the system-level events (transactions) to the signal-level events.

4. Test bench authoring

From our experience, an effective test bench usually contains concurrency, encapsulation and abstraction, self-checking, automatic test stimulus generation, and reusable components.

A test bench authoring tool for transaction-based verification uses the concept of transactions to help the user create effective test benches with these five features. There are four aspects:

- *partitioning the responsibility among TVMs and tests:* This partitioning speeds up the development process, enhances reuse, and allows automated recording of transactions.
- *specifying cause and effect relationships among transactions:* an engineer can write complex self-checking tests that are easy to understand.
- *specifying complex concurrency using inter-transaction synchronization:* an engineer can verify systems with complex out-of-order or pipelined characteristics.
- *specifying localized constraints in the attributes of a transaction:* an engineer can write constrained random tests that are easy to understand.

Furthermore, we have found that a transaction-based test bench provides a level of good abstraction for advanced techniques such as test stimulus transformation [13]. Test stimulus transformation is a technique that modifies the stimuli from a test bench to improve simulation coverage. We are currently implementing a prototype for transforming stimuli in a transaction-based test bench.

We have selected C++ as our test bench authoring language and created a C++ library called *TestBuilder.* TestBuilder facilitates the creation of transaction-based test benches. The main innovations in the design of this library are discussed in [12].

4.1 Partition of Responsibility

As shown in Figure 1, a transaction-based test bench is partitioned into two layers: tests and TVMs. The concept of a transaction is captured by a task within a TVM. A task translates this high-level concept into a series of signals at the design interface.

These concepts provide a convenient mechanism to divide the responsibility of test bench development among several verification engineers. Engineers with knowledge about the interface protocol only can develop TVMs; other engineers can develop tests from functional requirements without knowledge of the actual interface protocol. Protocol and internal event checkers for the basic protocol operations can be captured in a TVM. System-level event checkers can be captured in a system-level test.

With the responsibilities partitioned, new and complex test scenarios can be developed quickly. By encapsulating the protocol in a reusable module, the total size of the test bench shrinks, as measured by lines of code, and the value per line of the test code is enhanced.

Furthermore, this partitioning facilitates test bench re-use in two dimensions: TVMs can be reused for different designs that have similar interfaces but different functions; tests can be re-used for different designs that have similar functions but different interfaces.

4.2 Cause and effect testing

Once a test bench has been raised to the transaction level, as indicated in the previous section, we can easily add rigorous self-checking capabilities into the test bench.

Usually, users know the expected responses associated with specific stimulus sequences. For example, if we send a packet to a port of an ethernet switch, we expect that the packet will come out of a different port at some unspecified future time. However, because it is frequently difficult to manage the temporal disparity between stimulus on one interface and response on another, users frequently revert to less powerful mechanisms, such as logging results to a file for subsequent processing to determine correctness.

By invoking tasks in TVMs in a separate thread of execution (called spawning), it is a simple matter to perform what is known as cause and effect testing. Specifically, cause and effect testing relies on a so-called active slave TVM: while another TVM is generating stimulus to the design for a particular transaction, the active slave TVM calculates the expected output from the same transaction and monitors the actual output to report errors.

With active slave TVMs, the following straightforward algorithm can be used in a test to receive the benefit of knowing exactly when something bad occurs during simulation, without having to waste simulation cycles or wait until simulation results have been post-processed or visually analyzed.

```
loop until satisfied
   Assign (or randomly select) values
      for a task invocation;
   Spawn a task to generate stimulus
      to the design;
   Calculate expected transaction
      from the selected values.
   Spawn a task to check the outputs
      of the design;
end loop
```

Such an algorithm is particularly valuable when the expected transaction can be calculated by calling a reference model of the design being tested. In such a case, algorithms like this can be used even in directed ran- dom testing.

Using cause and effect testing allows the test author to focus on what transactions are expected, as opposed to when they should occur. The intent of the test is captured so that the test not only automatically checks for accurate responses, but is self-documenting and hence much more maintainable.

4.3 Complex concurrency

Running a task in a separate thread of execution relieves the difficulty of predicting exactly when response transactions are expected to occur in simulation. However, many practical designs allow responses at the output interface to be reordered arbitrarily. Two typical scenarios are

- *pipelined cache-coherent protocols:* in these protocols, it is common for data tenures to be re-ordered with respect to the ordering of address tenures.
- *multiple master, single target situations:* many systems (switches, multiprocessors, DMA controllers, etc.) use multiple independent TVMs generating stimulus to different input interfaces. In these scenarios, the order in which the multiple responses on the output interface occur is typically indeterminant and is usually different than the order in which the expected outputs were specified in the test.

Both situations can be handled with a clever use of queues to maintain outstanding transactions, for example, by using a queuing mechanism called smart queues. In order to use a smart queue, a user would specify how to extract a key from a transaction. With this information, a smart queue can maintain a FIFO ordering among transactions with the same key and allow arbitrary reordering among transactions with different keys.

In other scenarios, a rich set of synchronization mechanisms from the concurrent program community, such as semaphores, barriers, and mutexes, can be used.

The ability to verify order-indeterminant systems makes it easier to develop self-checking test benches with complex concurrency. Reactive stimuli can be generated regardless of uncertainties in transaction ordering. The result-checking mechanisms are strengthened with enforcement of custom ordering rules.

4.4 Constrained random testing

The ability to handle cause and effect testing and complex concurrency lays the foundation for another valuable feature in a test bench: constrained random test generation. For example, a transaction-based test bench allows us to easily randomize:

- the TVMs from which traffic is initiated,
- the specific tasks (or task sequences) that are invoked on those TVMs,
- the specific arguments used for invoking the tasks, and
- the synchronization among concurrent tasks.

Besides being able to randomize these selections, it is important that the random selections obey realistic constraints. Constraints can be used to ensure that the random selections obey system-level requirements, such as not writing to ROM space, or they can be used to focus testing in certain areas, such as focusing address selection at cache-line boundaries. A transaction-based test bench provides a set of intuitive locations at which we can specify the constraints: in the registry for the set of existing TVMs and tasks, in the arguments used for invoking the tasks, and in the synchronization within a TVM.

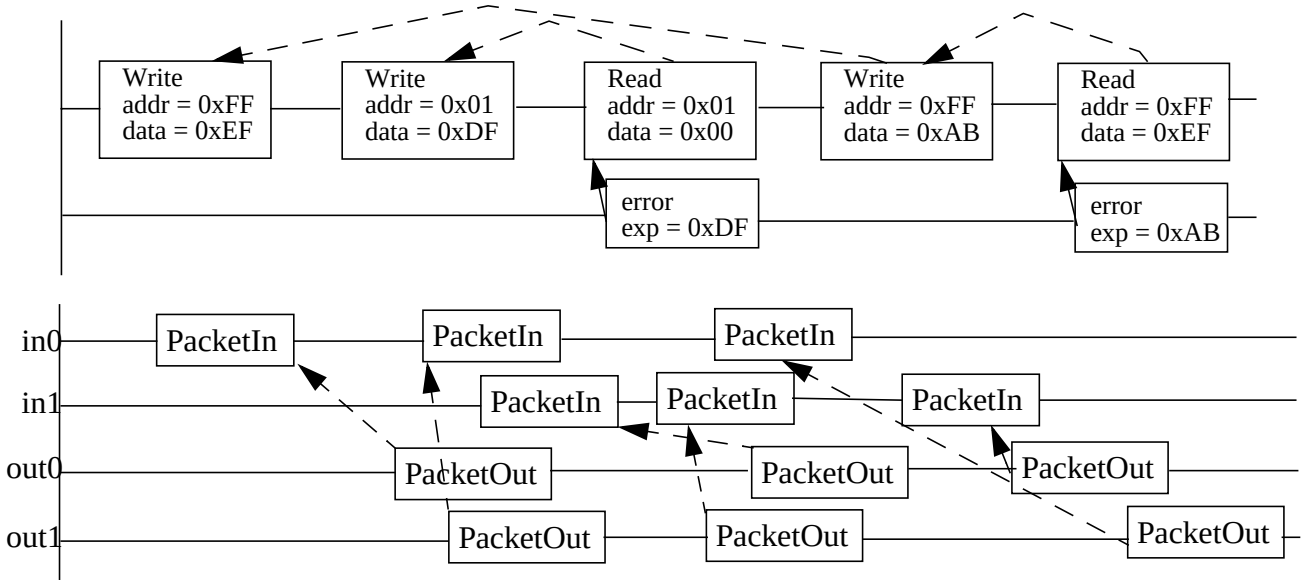
5. Efficient simulation debug

The four aspects of test bench authoring, discussed in the previous section, can be utilized to significantly improve the process of debugging a simulation run. Productivity gains can be achieved by presenting information in terms of transactions:

- transaction viewing: the abstract information about a transaction is displayed, rather than a sequence of transitions on certain signals.
- cause and effect: the relationships among transactions are displayed.
- error transactions: an error detected during simulation is recorded as an error transaction; the original causes can be found easily by checking the cause and effect relationships.
- concurrency: out-of-order / pipelined transactions are displayed in simple blocks with appropriate relative positions.

When users have access to automatically recorded transaction-level information, debugging is much easier. Figure 6

Figure 6: Transactions, transaction linkages, and error transactions



shows how simulation events can be presented in term of transactions.

The top two lines of display correspond to the activities in the processor bus. Five transactions have been exercised, with three for the address 0xFF and two for the address 0x01. A transaction-based test bench can link the transactions with the same address, and also detect problems in a simulation run by generating error transactions. Because a test bench authoring tool that understands transactions can automatically maintain relationships between the transactions, it is a simple matter to select the "Show Related Transactions" function of a transaction display tool to automatically cross-reference to the related write transactions (the stimulus or "cause"). Once the offending transactions have been identified, the relevant signal-level activities can be examined to determine whether the simulation error is a result of a bug in the test, the TVMs, or the design.

The bottom four lines show the usefulness of the concept of transactions when the design is capable of handling complex concurrency. By linking the packet coming out from a router to the original packet, we can easily match their attributes and determine the correctness of the design. If a packet has been corrupted, it is easy to identify the activities that happen between the time the packet entered and the time the packet came out. Similarly, these kinds of diagrams are very useful for pipelined protocols.

6. Intuitive functional coverage

Functional verification must measure functional coverage, identify holes in functional coverage, and determine with confidence when function verification is complete before the design is committed to silicon. Unfortunately, traditional approaches and their limitations cause many design teams to rely on code coverage, rate of new bugs found, or the project schedule to determine when functional verification completed.

With the concept of transactions, the process of checking coverage and writing new tests can be simplified by asking coverage questions in terms of transactions:

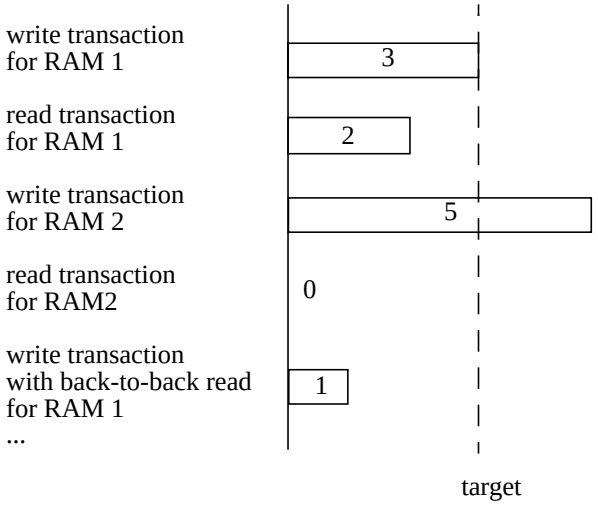
- enumeration of simulated transaction types,
- analysis of simulated transaction sequences,
- analysis of simulated overlapping transitions.

It would be very tedious and difficult to write most of these queries using specific signals at the interface of the design.

Figure 7 shows how a coverage analysis tool might use the concept of transactions to communicate the coverage information to the user. The histogram shows how many transactions of a certain type are observed at the interface of a design, without referring to the actual signals at the interface. As shown in the figure, there are three write transactions to the address space of RAM1, in which one has a corresponding read immediately afterward. A zero count indicates a hole in the high-level test suite.

The last coverage measurement in the diagram corresponds to a quantitative measurement of transaction sequences that have been exercised. In general, an arbitrarily complex sequence of transactions can be specified and a coverage tool can report how many times such a sequence (or portions of it) has been exercised. A similar work without the concept of transactions can be found in [14].

Figure 7: Basic transaction coverage



7. Case study and conclusion

An early version of the TestBuilder C++ library has been used to support the transaction-based verification methodology (TBV) for a system-on-chip design for a voice-over-packet application. We have been using a four-port ethernet packet router as a benchmark internally to determine the necessary features to support TBV and we have worked with the designers from several companies to test and fine-tune the implementation.

Our overall experience has confirmed that TBV significantly improves the productivity of the verification teams. With a short training in the new tools (C++, which is our test bench authoring language, and the TestBuilder C++ library), the verification teams can easily create new TVMs, tasks, deterministic tests and directed random tests, both for the system verification and IP verification. Design errors were identified quickly, with good feedback on the simulation coverage using transaction coverage.

Acknowledgment

The authors would like to thank the transaction-based verification engineering team for the design and realization of the concept presented in this paper.

References

1. Richard Goering, "DAI introduces Test-Generation Tool," *EE Times*, November 17, 1998.
2. Ann Steffora, "DAI Enters Transaction-Based Verification Market," *Electronic News*, November 17, 1998.
3. Steve Forde, Steve Bishop, and Ramnath S. Velu. "Streamlining HDL Code Coverage Analysis," *Integrated Systems Design*, December, 1998.
4. Michael Santarini, "Cadence Moves Toward Intelligent Testbench," *EE Times*, June 17, 1999.
5. Richard Goering, "Intelligent TestBenches Gaining Ground," *EE Times*, August 10, 1999.
6. Michael Santarini, "Cadence Offers Testbench Class Library via Open-Source License," *EE Times*, August 28, 2000.
7. Gale Morrison, "Cadence Open-Sources T-Builder," *Electronic News*, August 28, 2000.
8. Dennis Abts and Mike Roberts, "Verifying Large-Scale Multiprocessors Using an Abstract Verification Environment." *Design Automation Conference*, 1999.
9. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, pp, 233-243, Addison-Wesley, 1995.
10. *TestBuilder Reference Manual*, open-source document, September 2000. http://www.testbuilder.net
11. *TestBuilder User Guide*, open-source document, September 2000. http://www.testbuilder.net
12. Dhananjay S. Brahme, Steven Cox, João Geada, Mark Glasser, William Grundmann, C. Norris Ip, William Paulsen, John L. Pierce, John Rose, Dean Shea, Karl Whiting. "Creating a C++ Library for Test Bench Authoring Methodology," technical report # CDNL-TR-2000-0820, *Cadence Berkeley Labs*, August 2000.
13. C. Norris Ip, "Simulation Coverage Enhancement using Test Stimulus Transformation," to appear in *International Conference on Computer-Aided Design*, November 2000.
14. Brent E. Nelson, Robert B. Jones, Desmond A. Kirkpatrick. "Simulation Event Pattern Checking with PROTO," *International Conference on Simulation and Hardware Description languages (SHDL)*, 1994.