

Application Using Overlapping Transactions to Track Random Dynamic Multithreaded Processes

Alicia Strang

Marvell Semiconductor

<http://www.marvell.com>

astrang@marvell.com

ABSTRACT

Multithreaded processes are widely used in today's system verification environments to test the parallel functions of an ASIC. However, when an error occurs, debugging multithreaded processes can be very difficult and time-consuming. Debugging becomes increasingly unmanageable when many parallel threads are generated randomly and spawned dynamically.

This session will discuss using overlapping transactions to track random dynamic multithreaded processes, and using links to create a transaction relationship between each process and its detailed stage transactions. (The stage where an error occurs is recorded as an error transition.) Overlapping process transactions, introduced in Cadence IUS 5.7, are visualized in the waveform window, and each stage of the process can be viewed by a link.

This technology enables participants to analyze every stage of every process at the transaction level. This not only makes debugging multithreaded processes more manageable, but it also greatly reduces the amount of time spent inspecting log files and waveform signals. As a case study, this session will outline how the overlapping process transaction technology was used during the system verification of a Fibre Channel ASIC product to track a random number of dynamic SCSI commands.

1.0 Introduction

Section 2.0 discusses the reason for a multithreaded approach in system verification. Section 3.0 points out the challenges of using a dynamic random multithreaded test bench in system simulation. Section 4.0 introduces overlapping transactions, a new feature introduced in Cadence IUS5.7, and provides an illustration of how this new feature can be used to solve the problems a multithreaded approach faces. Section 5.0 presents a case study showing how the overlapping transaction feature was used during the system verification of a Marvell Fibre Channel product to track a random number of dynamic SCSI commands in a dynamic random multithreaded test bench.

2.0 Why Write a Multithreaded Testbench?

Multithreaded test benches construct efficient and complex system tests for today's system on chip (SOC) ASIC's. As ASIC's become more complex than ever, their functional verification becomes drastically more challenging. Single threaded testbenches not only are not good enough to cover all the functions of a SOC, but can also be very complicated.

A multithreaded test program can use threads to simplify the complex testbench by dedicating a unique thread to each small, independent task. Each functional task can now be written pretty much as if it was the only task running on the SOC.

A multithreaded testbench can also maximize parallelization. For most multithreaded tasks, where each thread spends its time waiting for a certain condition, maximum parallelization can be achieved. And on a multi-clock domain SOC, the testbench will be able to execute threads truly concurrently by allowing one thread to run on each clock domain.

By using multiple threads to separate the tasks of a test bench, each task will be executed only when the resource is available. These random combinations of small portion of tasks are actually automatically generated random test scenarios, and provide broader test coverage. Due to the fundamental advantages of incorporating multiple threads in writing test benches, multithreaded processes are widely used in today's system verification environments to test the parallel functions of an SOC.

3.0 Problems of Using a Multithreaded Testbench

With the new multithreading capabilities of writing testbenches came new challenges for the verification engineer. While there are many advantages to using multiple threads in writing testbenches, multithreads also introduce the complexity of error debugging and also introduce the possibility of encountering errors in the testbench operation such as starvation, deadlock, and improper synchronization.

Because of the dynamic interleaving manner of the multithreaded testbench, the execution of a small portion of each task are scheduled independently by the simulator. Consequently, each thread might be part of the way through one of the tasks when its execution is interrupted and the other thread is allowed to run. Based on when a SOC design bug is hit, the symptom could be intermittent, which can be very difficult to diagnose and allocate.

In addition to the intermittent manner of errors, in the case of an SOC error in a process of a multiprocessing simulation, not only the specific process is affected. All the threads in the entire simulation could be affected. To find out which thread originates the problem is very challenging too.

4.0 solutions

Every verification engineer who has used multithreaded processes knows how hard it is to manage runtime errors. If a waveform existed which could display all the concurrent tasks running on different threads, and could display the error location when an error occurs, then debugging runtime errors could be visualized.

Overlapping transactions provide the way to generate such a waveform. Overlapping transactions are a new feature introduced in Cadence IUS5.7. With this feature, the simulation records the concurrent tasks at the transaction level, taking advantage of the Simulation Database Interface (SDI2) library. SDI is a C++ interface that lets us record transactions from a simulation into an SST2 database files, which can be viewed in the Simvision waveform viewer.

The transaction-based verification concept is language independent. It has verilog, vhdl, C++, testBuilder, systemC interfaces, and I believe it will have a SystemVerilog interface also.

Verification engineers now have a way to record all concurrent tasks as overlapping transactions. When an error occurs or a SOC design bug is hit on a certain running task, the corresponding transaction can be marked as an error transaction. By viewing the overlapping transactions in Simvision, verification engineers can easily diagnose what caused the runtime errors, and can accurately allocate where and when it happened.

Figure 1 shows how to construct a multithreaded, debugging friendly testbench using overlapping transactions:

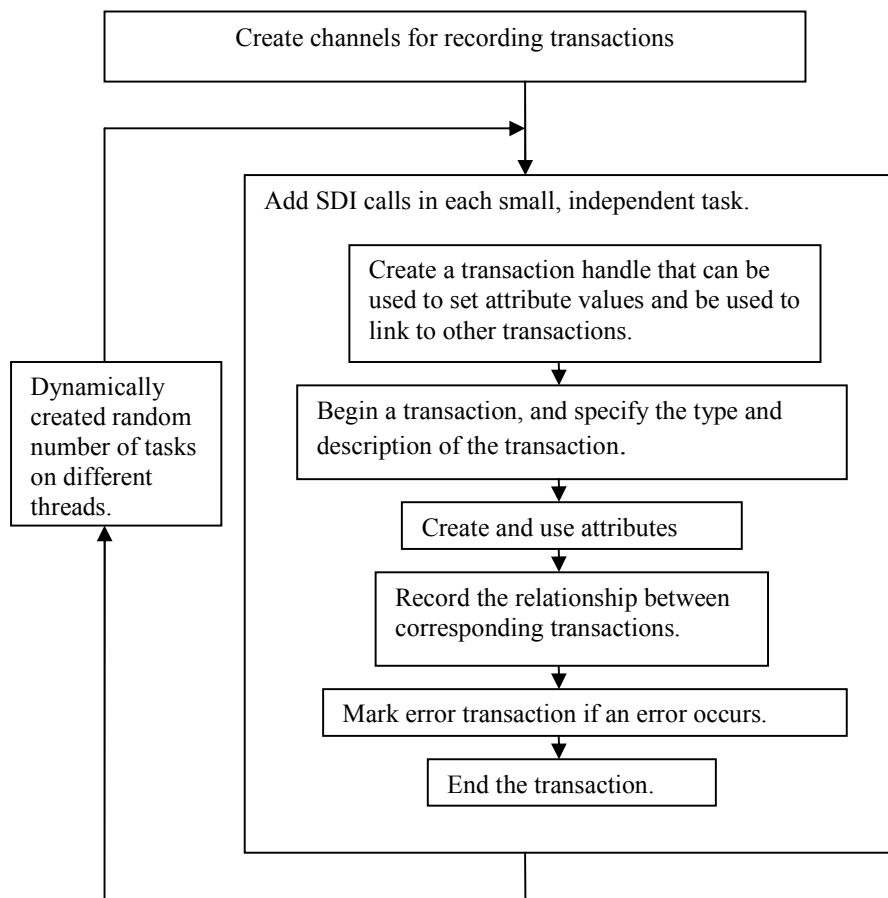


Figure 1 -- Implementation Flow Chart

First channels for recording transactions need to be created. These channels are call streams. Streams are created when the `sdi_create_fiber` system function is encountered in the code and a handle is returned so that it can be used to create transaction on the stream.

When using multiple threads to separate the tasks of a testbench, each task will be wrapped with SDI calls. At the beginning of each task, start a transaction on a stream using the `sdi_transaction` function call. Specify the type of transaction as "Begin_No_Parent" that indicates this is an overlapping transaction.

For each task, arguments and characteristics are recorded as attributes to transactions. Attributes are name-value pairs. `sdi_set_attribute` call will add attributes to transactions.

Each concurrent task is executed only when the resource is available. When the small portion of the task is executing, it will cause other transactions on other streams. The transaction of this task is the predecessor, and the other corresponding transaction is the successor. The `sdi_link_transaction` call is used to create a link relationship between transactions, so that even when a small portion of each task interleaves, which task a result belongs to will still be able to be traced. When the checking function of a task finds unexpected activity, specify the type of transaction as “Error”, mark this is an error transaction.

By performing the above steps in each task, multithreaded processes can be traced and any bug or error can be more easily found.

5.0 Application Example

The Marvell enterprise disk drive controller SOC supports command queuing and the multiplexing feature of the scsi-3 fcp-2 protocol. If initiators and targets execute only one command between them at a time, the system will be very inefficient. During command execution, there are periods of inactivity when the disk drive is seeking, reading, or writing data. By allowing multiple commands to be active at the same time, the interface utilization is greatly improved. The Marvell disk drive controller SOC architecture improved the efficiency of command processing by placing received commands in a command queue, and by allowing order changes in which it executes commands in the task set. Therefore the SCSI commands are executed in a dynamic interleaved manner.

Figure 2 shows an example of a single write-type command. An example of a read-type command is illustrated in Figure 3.

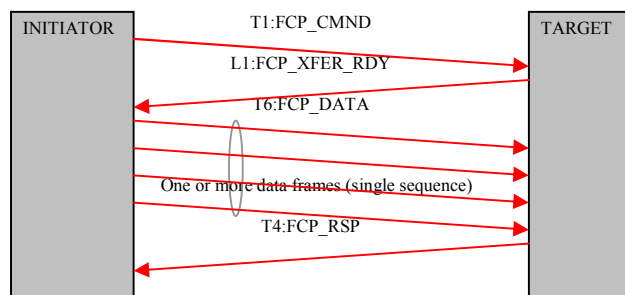


Figure 2 -- Single Command: FCP Write

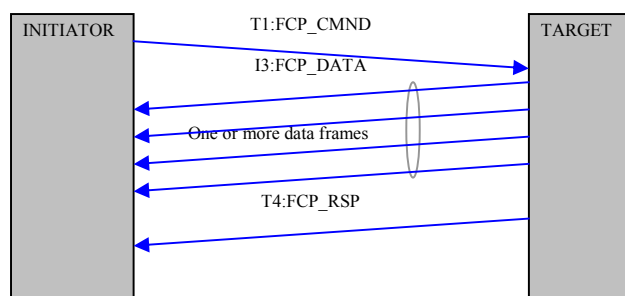


Figure 3 -- Single Command: FCP Read

Figure 4 shows that two commands are executed at the same time. When a random number of commands are executed at the same time, they will not only be tangled together, but also will not be in order. It is impossible to draw a figure with all the random number of concurrent commands.

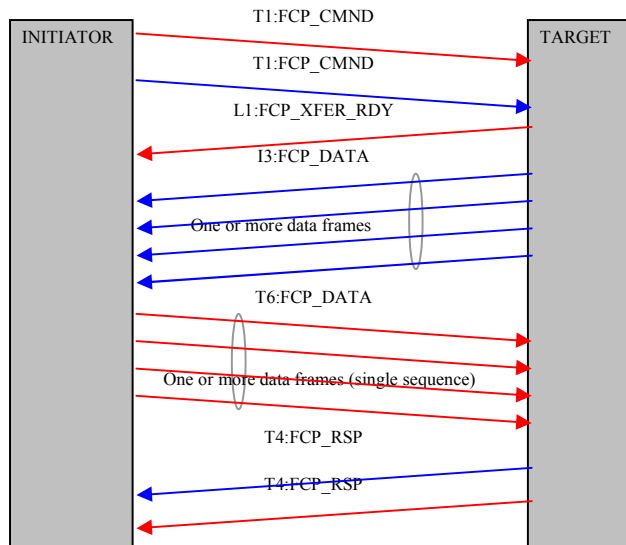


Figure 4 -- Two Commands Queuing and Multiplexing

Verifying multiple command queuing and multiplexing at the same time is a great challenge. A traditional way to write testbenches is to manually control each frame communication between initiator and target one by one. This is not only very tedious, but also very hard to test all possible cases. The multithreaded approach will generate a random number of dynamic SCSI commands, and execute these commands in a parallel interleaving manner.

The example in the following pseudo code shows how to use SDI calls to wrap the parallel commands so they can be recorded as overlapping transactions.

The pseudo code:

```

=====
//fiber and transaction handle declaration
integer fiberHandle;
integer xactHandle;

//create a fiber
fiberHandle=$sdi_create_fiber(fiberName);

//+++++
//task of command execution
//+++++
task execute_command (command cmdT)
{
    string cmdName = cmdT.getName();

//begin parallel transaction
    xactHandle = $sdi_trans("Begin_No_Parent", FiberHandle, cmdName, cmdName,"");

//set transaction attributes
    $sdi_set_attribute( xactHandle, "OpCode",    cmdT.getOpCode(),    "h");
    $sdi_set_attribute( xactHandle, "oxid",     cmdT.getOxid(),           "h");
    $sdi_set_attribute( xactHandle, "numOfBlks", cmdT.getNumOfBlks(),   "d");
    $sdi_set_attribute( xactHandle, "bytesXfer", cmdT.getBytesXfer(),   "d");
    $sdi_set_attribute( xactHandle, "lba",     cmdT.getLBA(),           "h");
}

```

```

...
//command execution code goes here

...
frameR.print_header_info();
frameR.set_header_attribute(fcRxXactH);

//link transactions
$sdi_link_trans(scsiCmdH[frameR.getOxid()],fcRxXactHandle);
...

//mark errors
if(mismatch)
    xactHandle = $sdi_trans("Error", FiberHandle);
...

//end transaction
$sdi_end_transaction(XactHandle);
}

//+++++
//random commands are executed at the same time
//+++++
cmdNumber = random();
while(cmdNumber)
{
    fork execute_command (nextCmd); join none
    suspend_thread();
    cmdNumber--;
}

//=====

```

When running a simulation, the simulator records concurrent tasks at the transaction level and stores them in the SST2 Database. The resulting transactions can be viewed in the Simvision window, along with the lower-level signal waveforms.

The dynamic commands are shown in the Simvision waveform as overlapping transactions in Figure 5. They are collapsed together.

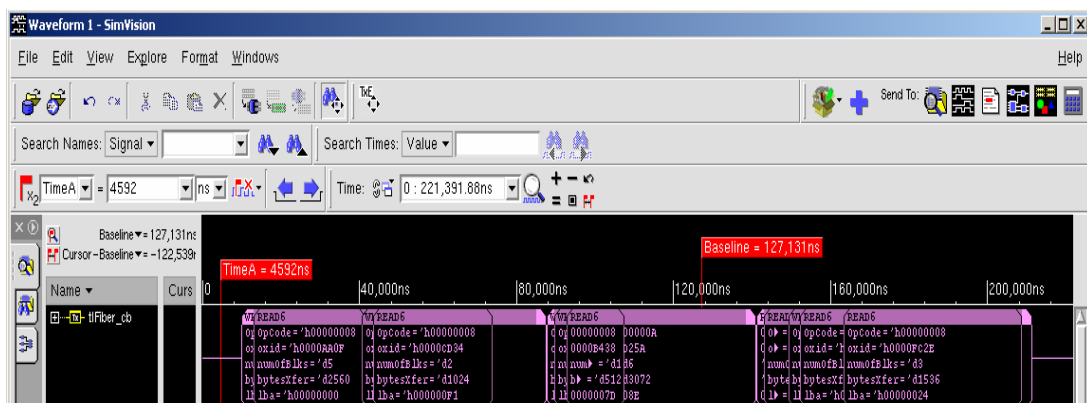


Figure 5 -- Multithreaded Process Displayed as Overlapping Transactions in Simvision Waveform

It is difficult to see what is really going on, but it is easy to tell that there are concurrent tasks running on different threads at the same time.

To expand the overlapping transactions, right click on them to display a pop-up menu as shown in **Error! Reference source not found.** Choose Expand Stream. The overlapping transactions are drawn below the in additional rows in parallel. They are not overlapped any more so the detailed attributes of each task can be seen.

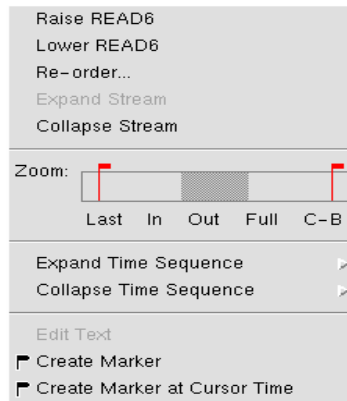


Figure 6 -- Simvision Pop-Up Menu to Expand the Overlapping Transactions

Figure 7 shows concurrent tasks running on different threads in different rows. Each task's name, and what the task does can be seen clearly and simultaneously.

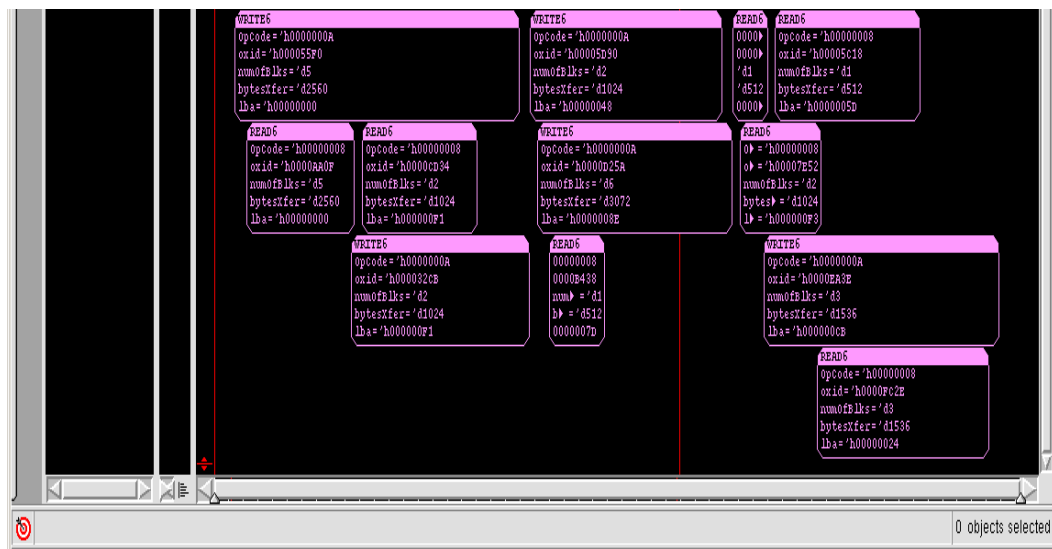


Figure 7 -- Expanding the Overlapping Transactions

Using Transaction Explorer, the linked transactions can be traced. Figure 8 shows the table that displays all the linked transactions which are generated by using the wrapper shown in the above pseudo code.

Scope	Fiber	Label	Start_Time	Duration
top.SDI	tlFiber_cb	WRITE6	10020ns	75110.6ns
top.SDI	fcTxFiber_cb	TX_FRAME	10413.32ns	604.16ns
top.SDI	fcRxFiber_d9	RX_FRAME	10451.08ns	604.16ns
top.SDI	fcTxFiber_d9	TX_FRAME	11508.36ns	415.36ns
top.SDI	fcRxFiber_cb	RX_FRAME	11546.12ns	415.36ns
top.SDI	fcTxFiber_cb	TX_FRAME	46813.96ns	6985.6ns
top.SDI	fcRxFiber_d9	RX_FRAME	46851.72ns	6985.6ns
top.SDI	fcTxFiber_cb	TX_FRAME	54101.64ns	7249.92ns
top.SDI	fcRxFiber_d9	RX_FRAME	54139.4ns	7249.92ns
top.SDI	fcTxFiber_cb	TX_FRAME	61653.64ns	10837.11ns
top.SDI	fcRxFiber_d9	RX_FRAME	61691.4ns	10837.12ns
top.SDI	fcTxFiber_d9	TX_FRAME	84649.48ns	415.36ns
top.SDI	fcRxFiber_cb	RX_FRAME	84687.24ns	415.36ns

Figure 8 -- Table of the Related Transactions

When the checker in the task detects an error occurred in the simulation, the error transaction shows up in the waveform as a sequence time marker. Figure 9 shows a received frame that has an error.

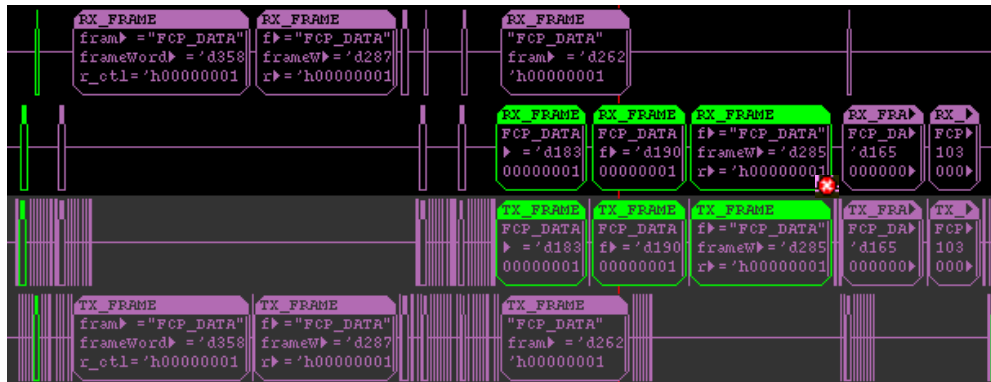


Figure 9 -- Received a Frame that has an Error

Starting from the transaction with an error, and using the related transaction links to trace up and down to find which parallel task this error originated in, finding the source of errors becomes straight forward. This greatly reduces the amount of time spent inspecting log files and waveform signals.

By displaying related transaction links, the order changes in which commands execute can be seen. The link highlights show how the SCSI commands are executed in a dynamic interleaving manner.

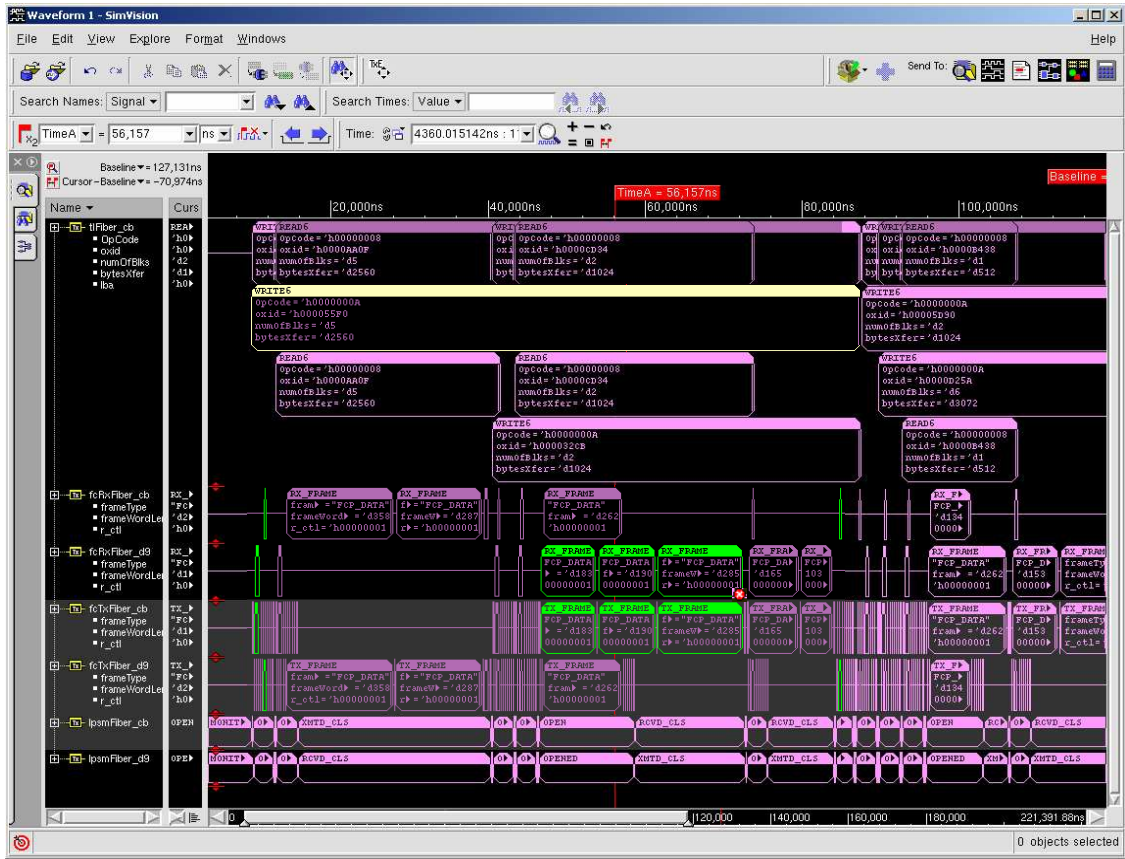


Figure 10 -- Tracing Parallel SCSI Command Tasks Running on Multiple Threads

Figure 10 shows parallel SCSI command tasks running on multiple threads. On the first WRITE6 command thread, the command frame goes out first, and soon after the transfer ready frame is received. Then the SOC interleaves between commands. Instead of a write data frame for this WRITE6 command, the READ6 command on the second thread is executed. Until time 45000ns the data frames of this WRITE6 are sent out, then the SOC interleaves with another command again. Finally at time 85000ns the good response frame of this WRITE6 is sent out and the command is finished.

On the third data frame the target received, an error is detected. From the highlighted link, it originated from the first WRITE6 command thread.

6.0 Conclusion and Recommendations

The overlapping transaction technology allows verification engineers to analyze every stage of a multithreaded process at the transaction level. This visualizes the activities on all threads, making error debugging straight forward. It greatly reduces the amount of time spent inspecting log files and waveform signals. Used together with the transaction links, the dynamic random multithreaded tests can be monitored and followed.

7.0 Acknowledgements

I would like to thank Wayne Datwyler of Marvell Corporation for his editorial assistance and helpful comments.

8.0 References

[1] Cadence Design System, Inc, "Transaction Recording: What's New" Product Version 5.7, February 2006

[2] Cadence Design System, Inc, "Simvision: What's New" Product Version 5.7, February 2006

[3] Cadence Design System, Inc, "Transaction Recording SDI2 Reference" Product Version 5.7, February 2006

[4] Cadence Design System, Inc, "NC-SC CVE Library Reference" Product Version 5.7, February 2006

[5] Cadence Design System, Inc, "Transaction Recording SDI/HDL Reference" Product Version 5.7, February 2006