



Visualizing Debugging Using Transaction Explorer In SoC System Verification

Alicia Strang, Robert C. Carden IV
Marvell Semiconductor, Inc. CA

MOVING FORWARD
FASTER®

Overview

- The traditional debug process requires viewing large swaths of waveforms at one time.
- As semiconductor designs continuously grow in complexity, using a waveform viewer to debug a SoC system is like examining a forest from an ant's view: this becomes impractical.
- To move us from an ant's view to an eagle's view of the forest, data flow and bus transactions must be presented in way that will empower an understanding of the big picture of a system without unnecessarily cluttering the debug environment.

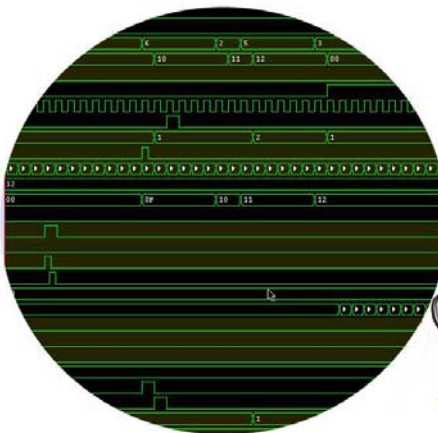


Too Much Or Not Enough Information?



The Traditional Debug Process

Using a waveform viewer to debug a SoC system is like examining a forest from an ant's view: this becomes impractical.



Traditional test benches typically will generate complex text based log file and huge waveform dumps. Waveform viewer implementations, such as Simvision and VirSim adhere to logic analyzer paradigm where information presentation is necessarily primitive, these files are of a raw format that is impractical to be used effectively for large systems.

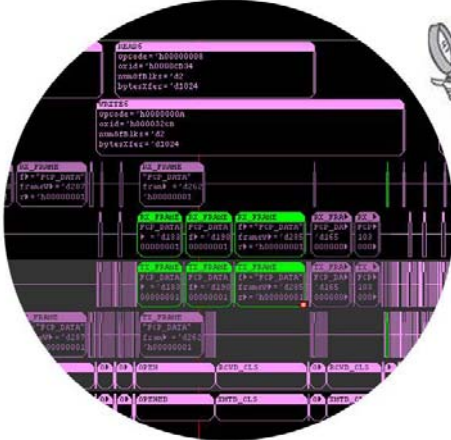


Simply, the sequential log file form and the underlying massively parallel DUT presents cognitive dissonance; it is a distraction from the problem at hand.



Use High level of abstraction

To move us from an ant's view to an eagle's view of the forest, data flow and bus transactions must be presented in way that will empower an understanding of the big picture of a system without unnecessarily cluttering the debug environment.

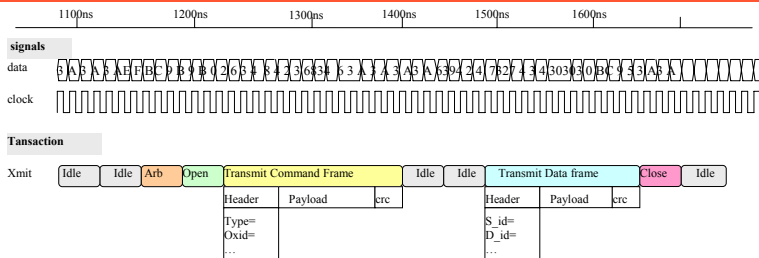


Cadence Transaction Explorer (TXE) enables advanced transaction viewing, navigating and organizing the waveforms.

Use high level of abstraction – transaction based - linked hierarchical transactions instead of signals to track data flow through a SOC, debugging and signal exploration is simplified. The presentation of data at only the relevant layer of abstraction allows the engineer to quickly locate the problem area, find the fundamental problem.



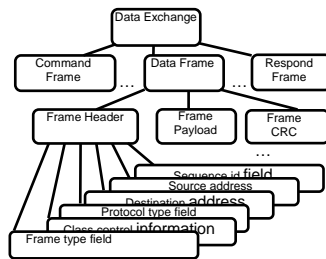
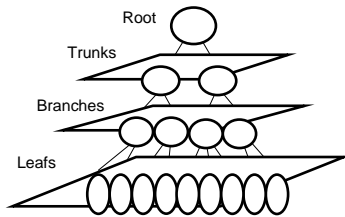
Getting Started to Record Transactions



To reduce the debugging environment clutter, it is practical to collapse data busses into a textual block to encode control signals into human readable text, and displayed bits as status messages. A few blocks of text can be used to replaces huge amounts of low level binary waveforms. The next step in the simulation is to move from the visualizing of absolute waveforms to context based transactions. These transactions can then be used to generate one or more higher level transactions. Thus a hierarchical tree of transactions can be constructed and traversed to view data flows, system configuration, and even inter-chip communications.



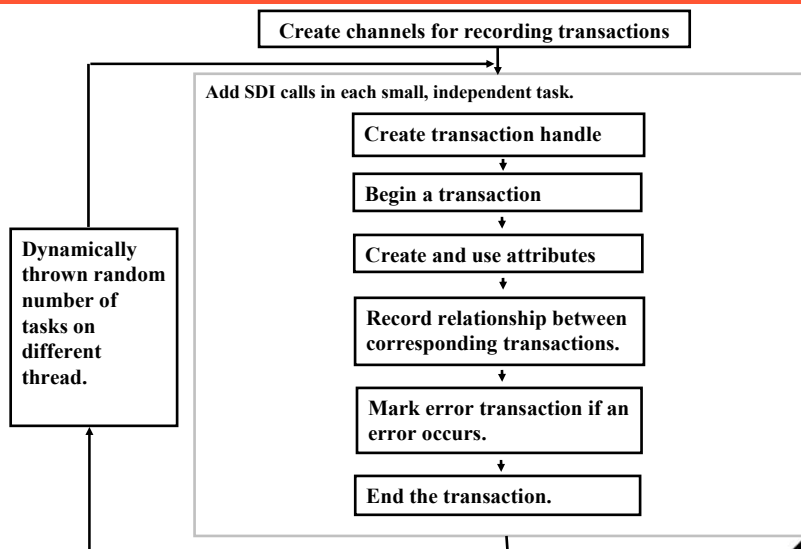
Hierarchical Transactions



- Transactions can be hierarchical
- Transactions can be linked together
- Linked hierarchical transaction methodology involves three dimensions:
 - Transaction hierarchies
 - Time
 - Stages of data flow (format changes)



Transaction Construction Flow Chart



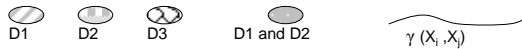
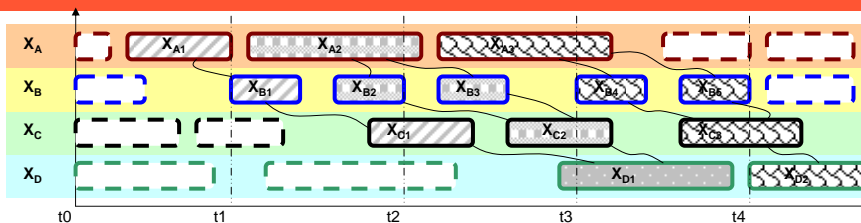
Methodology Implementation

Use high level of abstraction - transaction:

- Record all concurrent tasks as overlapping transactions.
- Mark an error transactions.
- Debugging by viewing the overlapping transactions in waveform.
- Locate error by tracing the transaction links.



Linking Across Functional Units



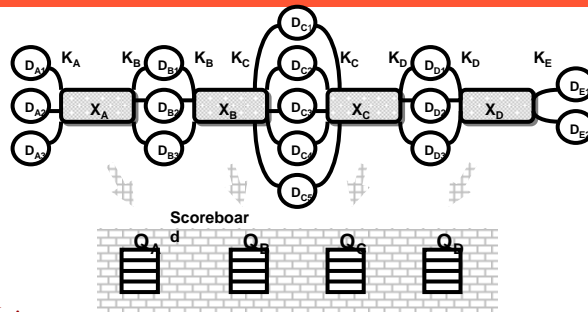
•Each transaction captures the processing done by indicated hardware and the data transformations at each stage

•A unique identifier at each stage tracks each logical entity through all data transformations

•Trace logical entities across hardware functional units, explicitly exposes causality



Process Diagram



Convert K_B to K_C :

Convert data from format B to format C. This process is recorded as Transaction B which has a handle B.

Set $\sigma_{BC} = (H_B, K_C)$ pair :

Set the transaction handle B associated with the identifier of new data format C.

Save σ_{BC} into Q_B :

Save the mapping pair into a queue on the scoreboard.

Search for σ_{AB} in Q_A using K_B as index :

Search the previous mapping queue on the scoreboard, using identifier of the old data format B as index to get the mapping to previous transaction handle A.

if (H_A, K_B) found then invoke $\gamma(X_A, X_B)$:

If a match is found, use the previous transaction handle A and current handle B to set up a causality link, where A is a predecessor and B is a successor.

else set error transaction associated with X_B :

Otherwise, if no match is found, an error is detected. Mark current transaction as an error transaction.



Transaction Usage Coding Example

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

//create a stream
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");
    ...
}
```



The Coding Example - continue

```
//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--;
}
```

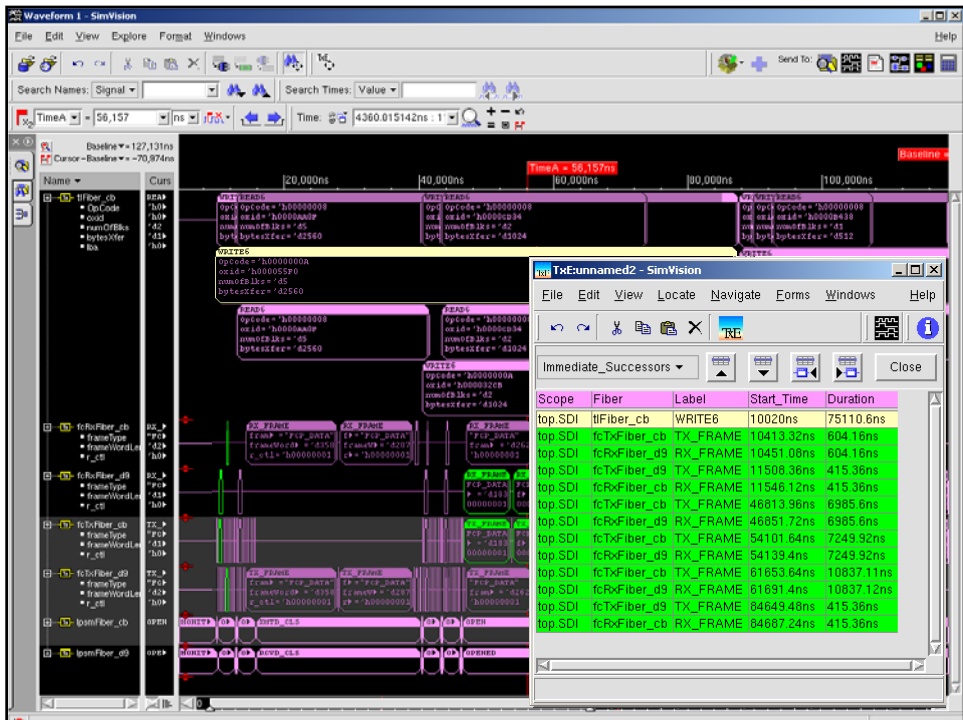
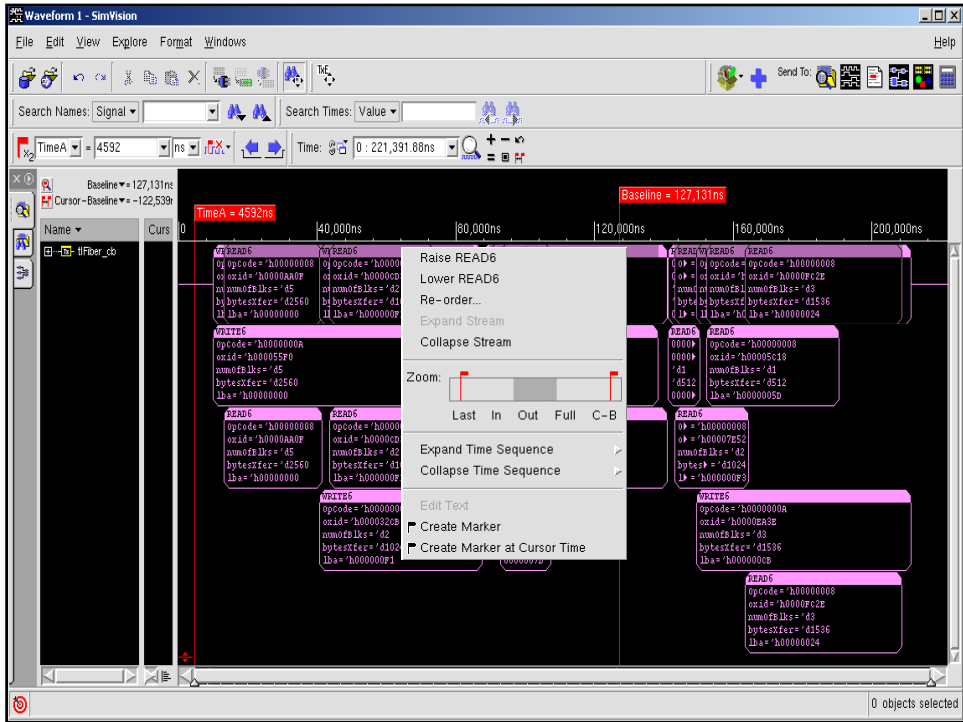


Transaction (Event) Sequences

The screenshot displays a simulation environment with two main windows. The left window shows a timing diagram with a signal trace for 'Fiber' and 'ModelCmd'. The right window shows a code editor with the following Verilog code:

```
2 init {
3   set next_state idle
4   set count 0
5 }
6 apply {
7   always {
8     set state $next_state
9   }
10  fiber top SBI fwFiber {
11    trans_type op48 {
12      switch -glob == $state {
13        "idle" {
14          set next_state atatal
15          set state {}
16        }
17      }
18    }
19  }
20  fiber top SBI fwFiber {
21    trans_type INT1 asserted {
22      switch -glob == $state {
23        "atatal" {
24          incr count
25          set next_state idle
26          set state {}
27        }
28      }
29    }
30  }
31 }
32 table {
33   row op48_intl $count
34 }
```

The code editor also shows a 'Result' window with the output: 'op48_intl : 1'. The timing diagram shows a signal trace for 'Fiber' and 'ModelCmd' with a time scale from 0 to 600,000ns. The signal trace shows a series of pulses and transitions, indicating the execution of the code in the code editor.



Transaction-Based Functional Coverage

The screenshot displays a functional coverage tool interface with several windows:

- Test Results Table:** A table showing test results with columns for test ID, test name, count, goal, and status.

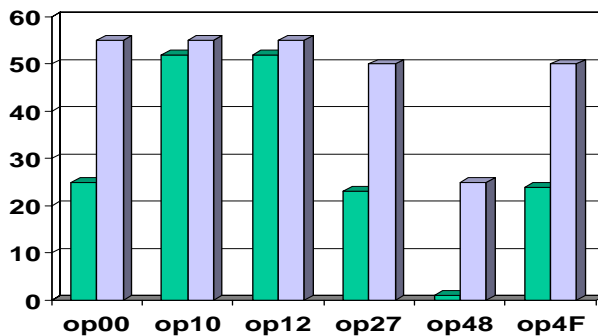
test_id	test_name	count	goal	status
test3_1	SELECT w/o ATN	1	1	OK
test3_10	SELECT w/ATH AND 8 MESSAGE	0	1	FAIL
test3_11	Reset/act Command	0	1	FAIL
test3_12	Bus-Initiated Reset/act3 Command w/o CDB Transaction	4	1	OK
test3_14	Bus-Initiated Selwath1 Command w/CDB Transaction	3	1	OK
test3_15	Bus-Initiated Selwath2 Command w/CDB Transaction (Carry)	1	1	OK
test3_16	Bus-Initiated Selwath2 Command w/o CDB Transaction, ID, TTD	4	1	OK
test3_17	Bus-Initiated Selwath2 Command w/o CDB Transaction, ID, DTAG	2	1	OK
test3_18	Bus-Initiated Sel w/o ath Command w/o CDB Transaction	4	1	OK
test3_19	Bus-Initiated Selwath3 Command w/o CDB Transaction / One ID	1	1	OK
test3_2	SELECT w/ATH AND STOP	0	1	FAIL
test3_20	Bus-Initiated Selwath1 Command w/CDB Transaction w/STOP On X	0	1	FAIL
test3_21	Bus-Initiated Selwath1 Test w/CDB Transaction w/Glitch Filter	1	1	OK
test3_22	Bus-Initiated Selwath2 Test w/CDB Transaction w/Glitch Filter	4	1	OK
test3_23	Arbitrate Without ID	0	1	FAIL
- Hardware Component Lists:** Several windows show lists of hardware components and their coverage counts, such as:
 - Component: `wt_BC_memCfg2`, Count: 2
 - Component: `wt_MP_modeSelLow`, Count: 1
 - Component: `wt_SC_Command`, Count: 24
 - Component: `wt_SC_ErrorInterruptStatus`, Count: 177
 - Component: `wt_SC_FF_ODatdBt`, Count: 48
 - Component: `wt_SC_FF_ODatdBt`, Count: 28
 - Component: `wt_SC_FF_ODatdBt`, Count: 22
 - Component: `wt_SC_SelectionConfiguration1`, Count: 24
- Opcode Coverage Table:** A table showing opcode coverage:

opcode	count
op00	25
op10	52
op12	52
op27	22
op48	1
op4F	24
- Hardware Register List:** A list of hardware registers and their coverage counts, such as:
 - Component: `BIG_ENDIAN`, Count: 1
 - Component: `COL_3`, Count: 0
 - Component: `COL_9`, Count: 0
 - Component: `DDR`, Count: 0
 - Component: `DDR_RAM_Bcol`, Count: 0
 - Component: `DDR_RAM_Bcol_32_Bit`, Count: 0
 - Component: `DDR_RAM_Bcol`, Count: 0
 - Component: `DISK_MODE`, Count: 0
 - Component: `DMA_MODE`, Count: 1
 - Component: `INT_1`, Count: 1
 - Component: `INT_1`, Count: 1
 - Component: `LITTLE_ENDIAN`, Count: 0
 - Component: `MODE_0`, Count: 0
 - Component: `MODE_2`, Count: 0
 - Component: `MODE_3`, Count: 0
 - Component: `MODE_4`, Count: 0
 - Component: `MODE_7`, Count: 0
 - Component: `MP_16_Bit`, Count: 1
 - Component: `MP_68000_16`, Count: 1
 - Component: `MP_68000_8`, Count: 0
 - Component: `MP_68000_8`, Count: 0
 - Component: `MP_68000_8LE`, Count: 0
 - Component: `MP_68030_16`, Count: 0
 - Component: `MP_68166_16`, Count: 0
 - Component: `MP_68166_8`, Count: 0
 - Component: `MP_68188_8`, Count: 0
 - Component: `MP_68188_8_DTACK`, Count: 0
 - Component: `MP_68188_8`, Count: 0
 - Component: `MP_M69000_16`, Count: 0
 - Component: `MP_S47021_16`, Count: 0
 - Component: `MP_S47032_16`, Count: 0
 - Component: `MP_TMS320C5_16`, Count: 0
 - Component: `MULT_DGS`, Count: 1
 - Component: `NCH_SHIFT_ACCR`, Count: 1
 - Component: `SDRAM`, Count: 1
 - Component: `SDRAM_Scol_2ram`, Count: 1
 - Component: `SHIFT_ACCR`, Count: 0
 - Component: `SINGLE_DGS`, Count: 0
 - Component: `SIZE_0`, Count: 1
 - Component: `SIZE_1`, Count: 0

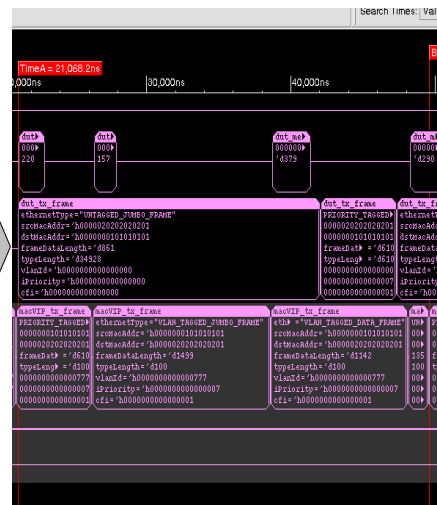
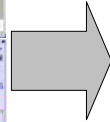
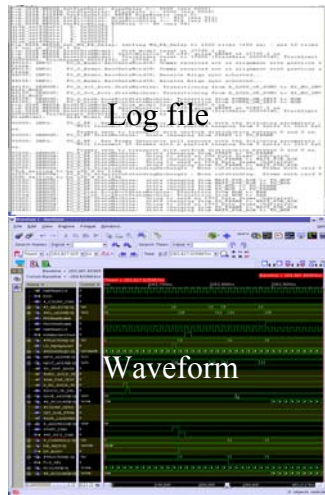
Exporting Coverage Data to Excel

Use TxE to export result to Microsoft Excel:

- Write the result table to comma separated values (csv) file
- Import the csv file to Excel



From Ant's View To Eagle's View



Conclusion

- Higher Levels of Abstraction take the first step in the revolutionary process.
- Error Tracing Across Functional Units
- Going Down Hierarchy for More Detail
- Abstract level display making viewing hardware function in big picture possible. This opens a new world to the EDA company to create performance analysis tool function, which can help architects in the early design stage.



What's next?

We suggest:

- **Enhanced graphical representations**
- **Error flag propagation through multiple levels**
- **Advanced waveform and transaction browsing via multiple windows and layouts.**
- **Transaction color coding based on data and customer specifications.**

