A High-Level Debug Environment for Communication-Centric Debug

Kees Goossens[1,2], Bart Vermeulen[1], Ashkan Beyranvand Nejad[3]
[1]NXP Semiconductors Research / SOC Architectures and Infrastructure
5656 AE Eindhoven, The Netherlands, {Bart.Vermeulen,Kees.Goossens}@nxp.com
[2]Computer Engineering, Delft University of Technology, The Netherlands
[3]KTH, Royal Institute of Technology, Stockholm, Sweden

***Abstract*—A large part of a modern SOC's debug complexity resides in the interaction between the main system components. Transaction-level debug moves the abstraction level of the debug process up from the bit and cycle level to the transactions between IP blocks. In this paper we raise the debug abstraction level further, by utilising structural and temporal abstraction techniques, combined with debug data interpretation and logical communication views. The combination of these techniques and views allow us, among others, to single-step and observe the operation of the network on a per-connection basis. As an example, we show how these higher-level abstractions have been implemented in the debug environment for the Æthereal NOC architecture and present a generic debug API, which can be used to visualise an SOC's state at the logical communication level.**

I. Introduction

Modern systems on chip (SOC) are very complex, and, as a result, their prototype silicon may still contain errors. Debugging is the process of finding the cause of erroneous behaviour, which may arise in a hardware or software component of the SOC, or in their combination. However, traditional debug methods and tools often separate hardware debug and software debug. In addition, current debug methods tend to focus on the computation, e.g. the programmable processors and their interaction with memory. But many SOCs contain multiple processors, and a large part of the SOC debug complexity resides in their interactions (e.g. via shared distributed memories), rather than the individual processors. For these reasons, we propose to supplement conventional computation-centric debug with *communication-centric debug* methods [1].

Finding the cause of an error is a refinement process. The state of (part of) the SOC is first examined at a high level of abstraction (e.g. which applications are running, what kind of video frame is being processed). Time advances as the SOC transitions between these high-level states (e.g. go to the next use case, or to the next video frame), or when high-level events occur (e.g. the next I frame). When a suspect state or event is observed, the (debug) user zooms in and examines the SOC state in more detail, at a lower level of abstraction. In the end, this may require examining bits and advancing the SOC by single clock cycles. *Transaction-level* debug [1] moves this debug process up from the bit and cycle level to the transactions between IP blocks. In this paper we raise the abstraction level of the debug process another step.

Our work addresses multi-processor SOCs containing a network on chip (NOC). NOCs present many new debug challenges because they implement split, pipelined, concurrent transactions on connections between IP blocks. We add on-chip debug hardware infrastructure, such as monitors, described in Section III. This infrastructure is controlled by our off-chip Integrated Circuit Debug Environment (InCiDE) [2], which comprises both hardware and software, as described in Section IV.

In this paper we extend prior work by raising the abstraction of the debug process in several new ways:

1) *Structural abstraction*: the user of the debug environment is presented with the logical NOC topology instead of a hierarchy of VHDL modules, or series of flip-flops in scan chains. For example, `get_monitor(router)` retrieves the monitor attached to a router, which can be programmed without knowing the location of its registers in the right scan chain.

2) *Data interpretation*: The value of state bits found in debug scan chains can be interpreted as values in RTL registers of an IP block, by being aware of the logical hardware structure. For example, a FIFO can not only be printed as RTL registers (read and write pointer and a set of data words), but also as an ordered list of valid data in the FIFO. Similarly, the debug infrastructure is used at the level of transactions and start/stop actions are translated to/from appropriate bit values in registers.

3) *Logical communication* view: a NOC is a component that is programmed to implement different use cases (sets of connections) at a given point in time. Our debug environment uses information about the NOC configuration and topology to offer a view in terms of connections that spans multiple IPs (NIs, routers). For example, `get_router(conn)` retrieves the routers used by a connection. This dynamic, logical view extends the (abstracted) static structural views.

4) *Temporal abstraction* is the essence of transaction-based debug, and moves the debug process from the clock cycle level to handshakes for data elements, requests/responses, and transactions on a single connection. Single stepping [3] of one or more connections involves monitoring and controlling handshakes at multiple locations in the NOC, and is used to force a particular trace of transactions. In this paper, we

add the capability to single step on multiple connections, but without forcing a particular order that has to be known up front (which is unrealistic). With our approach, time is abstracted to a series of globally consistent states across multiple connections.

In all cases, multiple connections can be debugged at the same time, at different levels of abstraction, while other connections (and related IPs and applications, etc.) are unaffected. This debug environment sets the first step towards our goal to structurally improve the debug refinement process by offering logical, high-level views and by allowing interactions on structure, data, communication, and time.

We compare our approach with related work in Section II. The new improved on-chip debug hardware infrastructure is introduced in Section III, and the off-chip software infrastructure in Section IV. The structure and implementation of the high-level API for communication-centric debug are described in Section V. Its use is illustrated in Section VI. We conclude in Section VII.

II. Related Work

There are two types of work related to our debug environment: hardware debug tools for SOCs, and software debug tools for distributed and parallel software.

Hardware debug software, such as [4], [5] allows basic access to scan chains. RISE++ [6], In-situ debugging tool [7], and Innerview [8] additionally use abstraction to map signal names to on-chip locations similar to our debug kernel InCiDE. [9] addresses hybrid CPU/FPGA systems, and abstracts from the structural design view of hardware to a source code view of software. However, none of these take a communication-centric approach to debugging, or use a logical view on communication between IPs.

Debug tools for distributed and parallel software include p2d2 [10], MPVisualizer [11], and visualisation methods of [12]. Like us, they also instrument (software), monitor, and act on events. Moreover, they use information on the topology and message passing state of the software processes involved in the distributed computation. But this information is only used after an event has triggered, to retrace a data flow to possible errors. In our case, the topology and configuration information is also used earlier, to decide where to monitor and which events to trigger on.

III. Hardware Infrastructure

We implement our communication-centric debug method on the Æthereal NOC [13]. A NOC consists of routers and network interfaces (NI). The architecture of the NI has been made more modular compared to earlier work. The NI kernel still implements network level functions (essentially, moving data from one NI kernel to another, subject to flow control and quality of service). Compared to prior work, the NI shell has been split in a local bus and new NI shells, as shown in Figure 1. A master is now connected to a local bus that demultiplexes requests to different slaves to different NI shells, and interleaves returning responses in the correct

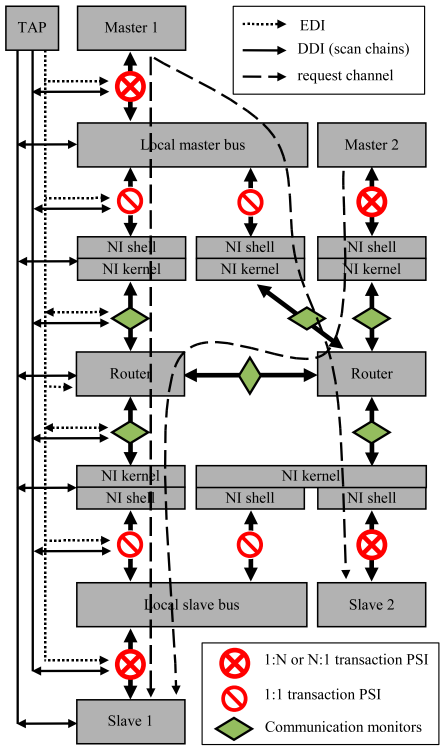

Fig. 1. NOC and Debug Hardware

order. The bus implements a multi-slave (narrowcast) connection [3]. The local bus uses an IP protocol such as AXI [14] or DTL [15]. Each NI shell serialises transaction requests to request messages, and deserialises response messages to transaction responses. Messages are (de)packetised by NI kernels, and transported by routers. The architecture at slaves is similar. A simple *connection* between a master and a slave thus contains two unidirectional *channels*: one for requests, and one for responses. Figure 1 shows the request channels (long dashed lines) of a multi-slave connection from master 1 to both slaves, and a simple connection from master 2 to slave 1. The response channels are the reverse. To offer quality of service, connections are configured at run time by programming memory-mapped registers in the NI kernels and local bus [16]. For debug purposes we add the following hardware blocks to a SOC: monitors that observe and then generate events, an *event distribution interconnect* (EDI), *protocol-specific instrumentation* (PSI) to act on events, and a *debug data interconnect* (DDI) to read out and program these blocks. We only discuss the PSIs in more detail and all others briefly, as they are already discussed in more detail in [1], [3].

Monitors (◇ in Figure 1) observe the SOC architecture and generate events when something of interest happens. They can be placed anywhere, but in our example, we use monitors to observe links between routers. They are programmed with a pattern that, when matched with data on the link, triggers an event. The EDI is a simple high-speed broadcast mechanism that propagates events to all PSIs. Events can be generated by monitors, (debuggers on) IPs, and by the user through the DDI. The DDI is implemented by using dedicated debug scan chains that are connected to an IEEE 1149.1 Test Access Port (TAP) [17]. This allows (low-speed) run-time access from off-chip debug hardware and software (as described in Section IV), independently and transparently from the functional operation of the SOC. The state of the monitors and the PSIs are observable and controllable via *test-point registers* (TPR) that are accessible through the DDI. The state of the functional IPs is also accessible through the TAP and the manufacturing test scan chains. However this approach is intrusive, because it requires that the functional clock is stopped first and then switched to the debug clock TCK.

To observe and control the state of transactions at the master or slave we insert PSI hardware blocks. Essentially, a PSI allows us to observe whether a request or response is in progress, and whether there are pending responses (for pipelined transactions). By manipulating the valid/ready handshakes for various signal groups we can stop requests from being accepted or responses from being offered to the master or slave, at the granularity of individual data words (elements) or whole requests/responses (messages). PSIs are inserted between master/slave and its local bus to control the transactions between a master and all its slaves, or a slave and all its masters (⊗ in Figure 1). They are also inserted between the local bus and the NI shells, to control the transactions of a single master-slave pair (⃠ in in Figure 1). PSIs are programmed through the DDI to perform an action, such as starting, stopping, or single stepping, at a certain granularity (element, message, or transaction), when an event is received through the EDI. The clock of IP blocks can also be stopped by a PSI.

Given the above debug hardware infrastructure, we can use the TAP to program monitors with interesting values to be matched, program PSIs to take actions on events arising from monitors or user, observe where events took place, and observe and modify the state of the SOC IPS and the debug infrastructure.

IV. Design Flow and Software Infrastructure

The NOC hardware is generated by an automated design flow [18]. Given a specification of the IP ports and a set of use cases (sets of concurrent applications), the RTL files of an application-specific NOC instance are generated. The specification is updated by the flow with the IP to NI mapping and information on how to configure the NOC at run time for different use cases. In addition, embedded configuration software is generated for each use case. The monitors, EDI, PSIs, DDI, and TAP controller are automatically generated and instantiated at RTL, when the user requests a debuggable design. Commercial tools are subsequently used to synthesise the NOC RTL and insert scan chains. [1], [3]. Scan insertion tools also produce a debug chain database (DCD) file that associates the bits in an RTL register (of the IPs and NOC components) with their position in the debug scan chain. Figure 2 shows the debug set-up. Off-chip debug hardware and software, such as our Integrated Circuit Debug Environment

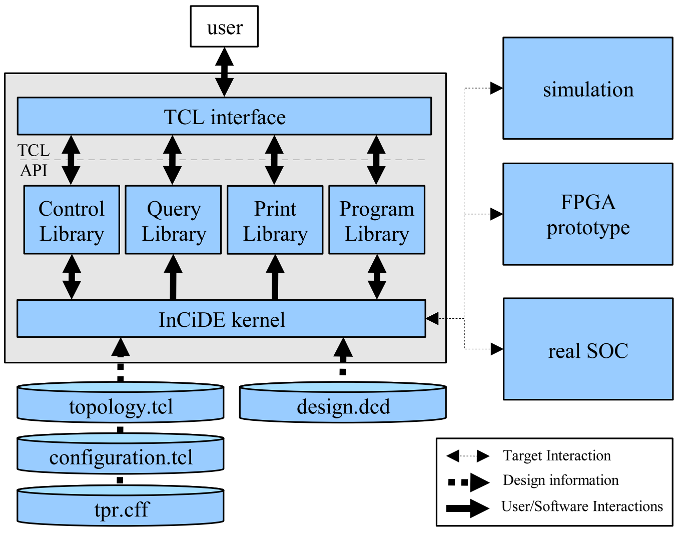

Fig. 2. Debug Software Environment

(InCiDE), connects through the TAP to the on-chip TPRs and functional scan chains. By programming the TAP, reset, and clock controllers, the user can place (parts of) the SOC in functional or debug mode, and inspect or modify the state of IPs (functional registers) or debug components (test point registers). InCiDE uses the DCD file to access functional registers at the right position in the right scan chain. It has a TCL interface to read, modify, and synchronise on-chip scan chains and their copies in the off-chip database. InCiDE can interact with a SOC simulation, an FPGA or real SOC hardware.

Although the NOC design flow and InCiDE allow us to automatically generate the required debug hardware to inspect and modify the network state through scan chains, several steps are still missing in the current flow. We have automated the insertion of monitors and PSIs. Currently, these are either not inserted, inserted everywhere, or at user-defined locations in the NOC topology. Smart algorithms on where to insert these are not integrated yet.

Next, the `topology.tcl` and `configuration.tcl` files are generated TCL versions of the corresponding XML files. The former allows InCiDE to abstract from IP modules to the NOC topology, to select IP modules based on the topology (e.g. NIs attached to a given router), and to translate sets of registers in routers and NIs to higher-level logical views (e.g. displaying the valid data in a FIFO). The latter defines which connections are active for all use cases, and maps the logical view on communication (i.e. connections between IP ports) to the structural view on communication (e.g. through which routers a connection goes).

V. Debug High Level API

Our API builds on the InCiDE kernel, and consists of four parts, as illustrated in Figure 2: a control API to control InCiDE, a query API to inspect the NOC topology and configuration, a print API to display information, and a program

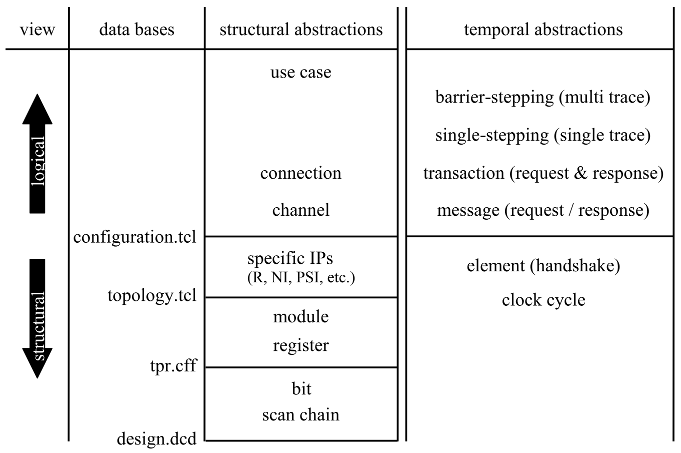

Fig. 3. Debug Abstractions

API to program the debug infrastructure.

These APIs allow a SOC to be debugged at more abstract levels than the traditional bit and clock cycle level. Figure 3 shows how a SOC's state is abstracted from scan chains structures to registers, generic RTL modules, and specific IP functions such as routers, NIs, and PSIs. The DCD, TPR, and topology files are used to perform this abstraction. These files are automatically generated by our NOC tools. One further abstraction step uses the configuration file to abstract from a structural hardware view on communication (with routers, NIs, local busses, etc.) to a logical software-configured view (with paths, credits, quality of service attributes, channels, connections, use cases, etc.).

Temporal abstraction is shown on the right side of Figure 3. This first allows multiple clock cycles to be abstracted to one or more data element handshakes. Only structural information on the valid and accept signals used by the communication protocol are needed for this. The steps to messages on channels and to transactions on connections move the abstraction level to the logical communication level.

The next two temporal abstraction levels are more complex as they involve the synchronised stepping of multiple communication channels. We define a basic single step for a communication channel to mean that the PSIs involved have to at least leave their stopped state and process one communication request. The command `sstep(`S`,`L`)` performs S single steps in succession for *all* PSIs in the list L. For multiple channels, all stopped PSIs of the channels involved will need to process one communication request.

Note that single stepping forces a unique transaction order that must be known in advance to accurately represent the original use case. Otherwise there can be unwanted dependencies between the channels that are single-stepped, which potentially can lead to a deadlock.

Consider for example the three channels in Figure 1. To single step these three channels in parallel requires that Master 1 performs one transaction to each of its slaves. Suppose that master 1 repeatedly sends a transaction to either slave 1 or slave 2. Normally, we don't know the order (e.g. slave 1,1,2,1,2 etc.). Waiting on a response from both slaves in the wrong order may cause the single step command to wait indefinitely after the first response from slave 1.[1] Waiting on a single slave, but guessing the order incorrectly (i.e. starting with slave 2) will produce the same result.

For this reason we introduce the barrier stepping command `bstep(`S`,`L`,`N`)`, where *at least* N out of the PSIs in list L must perform a single step. Barrier stepping is equal to single stepping when N is equal to the number of PSIs in L. Returning to our example, $N = 1$ enables stepping on master 1 without deadlock, even when the transaction order to slave 1 and 2 is unknown in advance.

A. API Implementation

Our debug API is implemented on top of the TCL interface provided by the InCiDE kernel, by adding a number of TCL functions and data bases. To support the refinement-based debug process, the user can mix the original and new abstract functions as required.

Control API: This API is closest to the basic InCiDE functionality, and serves to control the (simulated, prototyped, or real) SOC through the TAP, reset, and clock controllers, and scan chains. The `reset` command resets the TAP controller, and `nop` tells it to idle for a specific number of TCK cycles. InCiDE maintains a snapshot of the SOC state data to speed up user access to the data. These databases can be saved and reloaded for off-line debugging, using `save_state` and `load_state`. As shown in Figure 1, there are three sets of scan chains: for functional IPs, monitors, and PSIs. `read_tpr` and `read_sc` scan out and copy the SOC's debug (monitor and PSI TPRs) and functional scan chains to InCiDE's snapshot, respectively. `write_tpr` and `write_sc` do the reverse, to update the SOC scan chains, after local modifications by InCiDE. `stop` sends a stop pulse from the TAP controller to all PSIs via the EDI. Note that only those that have been programmed to be sensitive to events will react.

Query API: The DCD file tells InCiDE the mapping of RTL registers to their location in a scan chain. Registers have a hierarchical name, to reflect the structure of the IP modules. However, to debug a NOC-based SOC the NOC topology and the mapping of IPs to NIs is essential. We extended the NOC design flow to store this information in the topology file. The `get_{router,ni,ip,monitor}` commands enable the user to examine the SOC topology. A tick ('✓') for function `get_x(`*`list of y's`*`)` in Table I returns the list of all objects of type x attached to the objects in list y. An 'X' means the function is not possible, and a dash ('-') that it has not been implemented yet. For example, `get_monitor(get_router({ni1}))` returns the monitors attached to the router attached to NI `ni1`. Using the argument `all` returns all xs. `get_tpr` accepts master and slave IP names of connections and returns TPRs related to their PSI modules.

[1]For simplicity, we assume here that the master only issues non-split transactions, i.e. at most one outstanding transaction, e.g. AHB. Otherwise it may take more steps to deadlock.

The debug flow also updates the configuration file that specifies all use cases, the connections and channels in each use case, their attributes (path, credits, address map, etc.), and to which NI they are mapped. With this information, it is possible to obtain e.g. all channels (`get_ch all`), all channels departing from a NI (`get_ch ni`), or all slaves a master uses in a particular use case (`get_ip conn [get_conn ch [get_ch ni [get_ni ip master]]]`). Please note that some arguments have been omitted for brevity.

These functions dramatically improve the effectiveness of communication-centric debugging.

	argument type					
command	router	NI	IP	monitor	conn.	ch.
get_router	all	✓	X	-	✓	✓
get_ni	✓	all	✓	-	✓	✓
get_ip	X	✓	all	-	✓	✓
get_monitor	✓	-	-	all	X	X
get_tpr	X	X	✓	X	X	X
get_conn	✓	✓	✓	X	all	✓
get_ch	✓	✓	✓	X	✓	all

TABLE I
QUERY API COMMANDS

Print API: After selecting the object of interest (router, NI, TPR) using the query API, the traditional way of displaying their state would be a list of values in their RTL registers. By using structural information from all known IP blocks, i.e. routers, NI, monitors, PSIs, we can interpret the values in registers. (Note that the structure and hence interpretation of these components is heavily parametrised.) `print_tpr_psi` and `print_tpr_mon` pretty-print the TPRs of PSIs and monitors, respectively. `print_router` and `print_ni` are more sophisticated because they interpret the read and write pointers of their FIFOs, to show only the used locations of each FIFO. We illustrate these functions in Section VI. This API can be easily extended to include other IPs.

Programming API: The query and print APIs only query and display NOC state. The programming API deals with managing the debug activity. First, `set_mon_bp` enables the user to set break-points by programming monitors to be inactive, or to be active and trigger on particular (masked) data. Using `set_psi_action` the user specifies to which events PSIs react and how. They can be passive (do not act on anything), act unconditionally (e.g. stop, continue, or single step now), or act when events arrive from the event distribution interconnect (EDI). All of these can be at the element (handshake of individual data word) or message (request/response) granularity.

Second, the programming API implements single stepping and barrier stepping, to raise the temporal abstraction level. `continue(`L`)` continues all stopped transactions in list L, for one or an infinite number of element or message handshakes. Single stepping `sstep(`S,L`)` is equal to `bstep(`S,L,L`.length)`, as discussed above, and performs S single steps in succession, for all stopped transactions (PSIs)

```
1   wait until all $l$ in $L$ stopped
2   continue all stopped $l$ in $L$
3   wait until $N$ elements of $L$ left stop state and stopped again
4   go to 2 if # of passed steps $< S$
5   wait until all $l$ in $L$ stopped
```

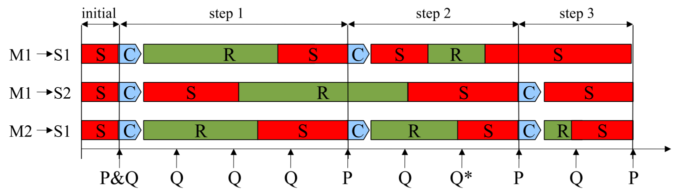

Fig. 4. Example of barrier stepping

in the list L. Barrier stepping `bstep(`S,L,N`)` implements that at least N of the PSIs in L perform a step. Table 4 shows the pseudo code for barrier stepping, and an example. In the design of Figure 1, `bstep(3,{ch1,ch2,ch3},2)` specifies stepping three times with at least two channels out of three proceeding. The crucial part of this algorithm in each step is where we check that at least two channels left their stop state and have stopped again. Note that waiting involves querying the TPRs of the PSIs until they are in the right state, specified by the quiescent and stopped state bits. Table 4 shows when the TPRs of the PSIs are programmed ('P') and when they are queried ('Q'). 'S' indicates that a channel has stopped, C that it has been continued (i.e. is allowed to run), 'R' that it has resumed and is running. Note that in step 1, the second channel is slow to resume running, and has not stopped when moving to step 2. This is ok because channel 1 and 3 stopped. In step 2 channel 2 does not need to be continued. Moreover, at querying moment 'Q*' two channels have stopped, and the decision to continue to the next step is taken. The last channel stopped between the last query and the reprogramming; due to sampling this was not seen. This is not a problem because it was not continued, and counts as an event in the next step.

VI. EXAMPLE USE CASE

For the hardware example in Figure 1, a netlist description of the design and the required API databases are automatically generated by the NOC design flow. We then run our debugger software with its extended API in order to perform interactive debugging using a simulated target. The following demonstrates the use of the API to control the NOC during debug.

Lines 1 and 2 reset the TAP controller and provide enough time time to functionally program the NOC. Lines 3 to 10 program break-points inside the monitors on the connection between Master1 and Slave2, as well as program debug actions for the request channel of the TPR belonging to the PSI of Master1. This PSI is programmed to stop communication at the element level after receiving an event via the EDI. Lines 11 to 13 shows the effect of continuing that stopped transaction on the PSI's status.

As a real use-case example demonstration of the barrier stopping shown in Figure 4, Lines 14 to 17 performed stepping over all stopped transactions on the request channels at the master sides of all connections (i.e. Master1 & Master2). The printed INFO lines show our stepping algorithm at work.

Lines 18 to 20 read back the NOC state and print the content of the router and NI queues corresponding to the connection between Master1 and Slave2.

```
1  api:> reset
2  api:> nop 1000
3  api:> set my_conn [get_conn ip {{{Master1 *} {Slave2 *}}}]
4  api:> set my_tpr [get_tpr {Master1 *} {Slave2 *} M req]
5  api:> set my_mon [get_monitor [get_router conn $my_conn]]
6  api:> set_psi_action $my_tpr -gran e -cond edi
7  api:> set_mon_bp $my_mon {-w 0 -fw 2 -value 3648}
8  api:> write_tpr
9  api:> read_tpr
10 api:> print_tpr_psi $my_tpr
 ---------------------------------------------------------
|                {core1 pi} -> {core4 pt}                 |
|---------------------------------------------------------|
|Ch. Type | St.En. | St. Gran. | St. Cond. | state | Left |
|---------|--------|-----------|-----------|-------|------|
|   Req   |  Yes   |  Element  |    EDI    |  Yes  | Yes  |
|   Resp  |  No    |  Message  |    EDI    |  No   |  No  |
 ---------------------------------------------------------
11 api:> continue $my_tpr
12 api:> read_tpr
13 api:> print_tpr_psi $my_tpr
 ---------------------------------------------------------
|                {core1 pi} -> {core4 pt}                 |
|---------------------------------------------------------|
|Ch. Type | St.En. | St. Gran. | St. Cond. | state | Left |
|---------|--------|-----------|-----------|-------|------|
|   Req   |  Yes   |  Element  |    EDI    |  No   | Yes  |
|   Resp  |  No    |  Message  |    EDI    |  No   |  No  |
 ---------------------------------------------------------
14 api:> set my_tpr_all [get_tpr * * M req]
15 api:> set_psi_action $my_tpr_all -gran e -cond edi
16 api:> stop
17 api:> step $my_tpr_all -n 3 -some 2
- INFO: Checking if all Elements are stopped.....
- INFO: All Elements are stopped.
- INFO: Stepping starts.
- INFO: step 1 finished.
- INFO: step 2 finished.
- INFO: step 3 finished.
- INFO: All Elements are stopped.
18 api:> read_sc
19 api:> print_router [get_router conn $my_conn]
 -------------------------------------------
|            BE queue of R00_p1             |
|-------------------------------------------|
| Q.Nr |                DATA                |
|------|------------------------------------|
|  18  | 1000000000000000000000000100100011 |
|  19  | 1100000000000000000000000100100100 |
 -------------------------------------------
- INFO: No valid data in GT queue of R00_p1.
20 api:> print_ni [get_ni conn $my_conn]
 -----------------------------------------
|          INPUT queue of NI000_p2        |
|-----------------------------------------|
| Q.Nr |               DATA               |
|------|----------------------------------|
|  21  | 00001000000000000000000000000100 |
|  22  | 00000000000000000000000100001000 |
|  23  | 00000000000000000000000100001001 |
|  24  | 00000000000000000000000100001010 |
|  25  | 00000000000000000000000100001011 |
 -----------------------------------------
- INFO: No valid data in OUTPUT queue of NI000_p2.
```

VII. Conclusion

We presented techniques to raise the debug abstraction level above the transaction level presented in prior work. Structural and temporal abstraction techniques were combining with debug data interpretation and logical communication views to visualise an SOC's state at the logical communication level. In addition, we presented a generic debug API, and control debug operations at the functional communication level. Results were presented on the implementation of these features in the debug environment for the Æthereal NOC architecture.

References

[1] K. Goossens, B. Vermeulen, R. van Steeden, and M. Bennebroek, "Transaction-based communication-centric debug," in *Proc. Int'l Symposium on Networks on Chip (NOCS)*. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 95–106.

[2] G. Rootselaar and B. Vermeulen, "Silicon Debug: Scan Chains Alone Are Not Enough," in *Proceedings IEEE International Test Conference (ITC)*, Atlantic City, NJ, USA, Sep. 1999, pp. 892–902.

[3] B. Vermeulen, K. Goossens, and S. Umrani, "Debugging distributed-shared-memory communication at multiple granularities in networks on chip," in *Proc. Int'l Symposium on Networks on Chip (NOCS)*, Apr. 2008, pp. 3–12.

[4] L. Jianhua, Z. Ming, B. Jinian, and X. Hongxi, "A debug sub-system for embedded-system co-verification," 2001, pp. 777–780. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=982678

[5] E. Moerman, S. Bocq, and J. Verfaillie, "Debug architecture for system on chip taking full advantage of the test access port," in *ETW '03: Proceedings of the 8th IEEE European Test Workshop*. Washington, DC, USA: IEEE Computer Society, 2003, p. 155.

[6] S. Vinoski, "Rise++: A symbolic environment for scan-based testing," *IEEE Design and Test of Computers*, vol. 10, no. 2, pp. 46–54, 1993.

[7] K. A. Tomko and A. Tiwari, "Hardware/software co-debugging for reconfigurable computing," in *HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 59.

[8] S. Z. Hanono and S. Z. Hanono, "Innerview hardware debugger: A logic analysis tool for the virtual wires emulation system," in *Master's thesis, Massachusetts Institute of Technology*, 1995.

[9] B. Roesler, E. Nelson, "Debug methods for hybrid CPU/FPGA systems," in *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*. IEEE Computer Society, 2002, pp. 243–250.

[10] D. Cheng and R. Hood, "A portable debugger for parallel and distributed programs," in *In Proc. of Supercomputing'94*, 1994, pp. 723–732.

[11] A. P. Claudio, M. B. Carmo, and J. D. Cunha, "Monitoring and debugging message passing applications with MPVisualizer," *PDP*, vol. 00, p. 376, 2000.

[12] E. Kraemer and J. T. Stasko, "The visualization of parallel systems: an overview," *J. Parallel Distrib. Comput.*, vol. 18, no. 2, pp. 105–117, 1993.

[13] K. Goossens, J. Dielissen, and A. Rădulescu, "The Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 414–421, Sept-Oct 2005.

[14] *AMBA AXI Protocol Specification*, ARM, Jun. 2003.

[15] *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, Philips Semiconductors, Jul. 2002.

[16] A. Hansson and K. Goossens, "Trade-offs in the configuration of a network on chip for multiple use-cases," in *Proc. Int'l Symposium on Networks on Chip (NOCS)*. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 233–242.

[17] B. Vermeulen, T. Waayers, and S. Goel, "Core-based Scan Architecture for Silicon Debug," in *Proceedings IEEE International Test Conference (ITC)*, Baltimore, MD, USA, Oct. 2002, pp. 638–647.

[18] K. Goossens, J. Dielissen, O. P. Gangwal, S. González Pestana, A. Rădulescu, and E. Rijpkema, "A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification," in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*. Washington, DC, USA: IEEE Computer Society, Mar. 2005, pp. 1182–1187.