Debugging Distributed-Shared-Memory Communication at Multiple Granularities in Networks on Chip

Bart Vermeulen[1], Kees Goossens[1,2], Siddharth Umrani[2]
[1] Research, NXP Semiconductors, The Netherlands, {Bart.Vermeulen,Kees.Goossens}@nxp.com
[2] Computer Engineering, Delft University of Technology, The Netherlands

Abstract

We present a methodology to debug a SOC by concentrating on its communication. Our extended communication model includes a) multiple signal groups per interface protocol at each IP port, b) the handshakes per signal group (e.g. for command), and c) the handshakes within a signal group (e.g. for write and read data elements). As a result, our debug methodology is the first to offer debug control at three communication granularities: individual data elements in a message, messages (i.e. requests or responses), and entire transactions.

Communication to distributed shared memories is supported in networks on chip (NOC) by transparently (de)multiplexing different master-slave channels based on the memory address, also called narrowcast. In this paper, we extend previous work on NOC debug that allowed per-connection debug (i.e. a master without differentiating between its slaves) to also support per-channel (i.e. per master-slave pair) debugging, also for narrowcast connections. This enables essential fine-grained debug control for multi-processor SOCs that use distributed-shared-memory communication.

The debug infrastructure consists of hardware components, and a software API and library. We define the hardware infrastructure and the required changes to a NOC. Our architecture cleanly separates the monitoring and distribution of events from how they are interpreted and used, in terms of hardware and programming. We define a high-level software API for run-time user control. The debug methodology offers run-time programmable breakpoints, stopping, continuing, and single-stepping of distributed-shared memory communication at three granularities, at the cost of 2.5% NOC area increase and no speed penalty.

1. Introduction

With the emergence of complex SOCs comes the unintentional but inevitable slip of some design errors (located in hardware or software) to the product bring-up phase. Finding these errors in a timely and cost-effective manner is increasingly important to ensure that the product can be released to the market on time. Traditionally the task of debugging an embedded system has been made easier through the up-front inclusion of debug support functions in the design, an activity known as Design-for-Debug (DfD). Debug support functions included in SOCs across the industry today [11] fall into two categories: real-time trace and run-stop control. To enable real-time trace, key internal signals are brought out, in real-time, onto chip pins. The ability to observe these signals is a great advantage during debugging.

Run-stop control uses on-chip support to stop the functional operation of the chip when a programmable condition occurs. Traditionally the response to this occurrence is to either have the processors in the system jump to an exception handler, and wait to be contacted by an external debugger tool, or by gating all functional clocks, freezing the complete system state. Afterwards, the external debugger software can switch the system to a debug mode, in which the system state can be examined, and where required, modified, before functional execution is resumed or restarted [12]. Software debugging takes place at the application source code level. Hardware debugging takes place at the IP clock cycle level. A single source code line can take many clock cycles to execute, making the system debug process very time consuming, as there are no intermediate levels on which debugging can take place as well. In addition, debugging at the clock cycle level is known to be very difficult, especially in SOCs with multiple clock domains, and in the presence of non-deterministic behavior and environmental conditions [5, 4].

In this paper, we therefore establish new intermediate debugging levels, address communication based on distributed shared memories for multi-processor SOCs, implement single stepping, and define and implement a high-level user API for run-time debug. For this we focus on the on-chip communication architecture, and extend the concepts and implementations explored in [8] and [18].

Key contributions of this paper are:

- We introduce an extended communication model that includes multiple protocol signal groups, handshaking per group (e.g. for the command group) and within a group (e.g. write and read data elements).
- This allows us to define intermediate levels for effective run-stop debugging of embedded systems, focussing on the on-chip communication, instead of on the on-chip computation. These increasingly coarse levels are: individual data elements (of write and read data), request and response messages, and entire transactions. The first level is new.
- NOCs implement distributed-shared-memory communication by demultiplexing requests from a master to the appropriate slave, and multiplexing the responses, called a narrowcast connection [16]. Prior work [8, 18] allowed debugging of connections with a master and multiple slaves, such as a narrowcast connection, only in a limited manner. This paper defines and implements per-channel, i.e. per master-slave pair, debug support. This significantly increases the flexibility and applicability of the debug methodology, which is required by SOCs with multiple processors that communicate via distributed shared memories.
- We show new details of the event distribution mechanism, including finite state machines (FSM) and the operation of stop event distribution.
- Single stepping, i.e. repeated stopping and continuing, is a key feature of a debug methodology. Although prior work introduced the concept, it did not implement it. In particular, single stepping at any of the three debug levels introduced here, while guaranteeing that no events are missed requires additional hardware support to atomically continue and stop. This is more complex than the separate stop and continue functions, described by earlier work.
- The new features (three debug levels, narrowcast debug, and single-stepping) all require changes to the network interface (NI) shell FSMs. We show an implementation for a particular protocol (DTL [14]), and a general recipe to modify NI Shell FSMs for other protocols.
- While prior work defined the basic steps on how to use a debug infrastructure, this paper gives both more low-level details of the test point registers (TPR), and defines a generic debug interface port and software API to abstract away from the basic, implementation-specific operations to a more generic and user-friendly interface.

In our examples we use a network on chip (NOC), but our concepts and implementation can be applied equally well to bus-based SOC architectures.

The remainder of this paper is organized as follows. Section 2 discusses the interconnection and communication models. In Section 3 we describe a typical session for debugging a system using a communication-centric approach, and derive debug control requirements. Section 4 describes how these can be implemented in a NOC and made accessible via a generic debug interface. Section 5 contains a description of the high-level software API we developed to control the SOC communication at the system level with different granularities of stopping and single stepping. In Section 6 we present the results of experiments we conducted, including silicon area cost, and impact on the maximum functional network frequency. We conclude in Section 7.

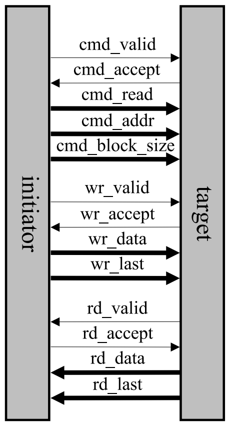

Figure 1. Example signal groups and signals of a DTL port.

2. Interconnection & Communication Models

Communication Model

To enable their re-use, IPs communicate on their ports using standardized transaction-based protocols, such as DTL [14], AXI [1], and OCP [13]. A transaction is initiated by a master port on an IP, and consists of a request message from master to slave. The execution of a request message by the slave can generate an optional response message.

A request message is encoded as two or more signal groups: the command group and the write-data group. A response message is encoded as one or more signal groups, e.g. the read-data group. Successive data words of the write and read data groups are called message (data) elements. Figure 1 shows some of the signal groups of DTL, which we use as a running example in Sections 4.3 (converting transactions to packets) and 6 (experimental results).

A valid/accept handshake is used to transfer an element per signal group. For example, the element of the command group comprises the command (read/write), address, and perhaps some flags. For the command and data groups

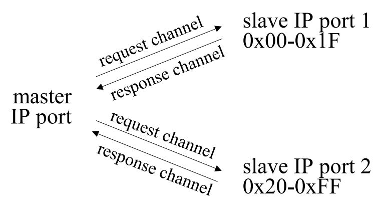

Figure 2. Narrowcast connection, implementing distributed shared memory communication between one master and multiple slaves.

the initiator produces data on the signal group and asserts the group's valid signal. The target then consumes the data and indicates this by asserting the group's accept signal. For the response message the role of initiator and target are reversed.

We distinguish three consistent granularities of communication. Starting with the smallest, these are: elements (coinciding with signal group handshakes), messages (requests or responses, consisting of one or more elements on one or more signal groups), and transactions (consisting of a request message and optional response message). The debug infrastructure introduced in Section 4 allows the debugging of the communication of a SOC to take place at each of these levels, depending on how it is configured.

Our canonical NOC [7] consists of routers and network interfaces (NI). A master communicates with a slave using two uni-directional channels: one for request messages and one for response messages. Most communication protocols implement distributed shared memory, where a master communicates transparently with multiple slaves. In other words, the master uses an address space without knowing how it is distributed over the slaves (on-chip and external memories, peripherals, etc.). A master communicates transparently with multiple slaves using a single narrowcast connection [16], see Figure 2.

NI Architecture

As illustrated in Figure 3, channels are implemented by the NI kernel, and connections are implemented by the NI shells [16]. After serializing the request signal groups ("s" in Figure 3), requests of a single master are demultiplexed to multiple slaves on a single connection in the master NI shell ("d" in Figure 3). Split pipelined requests may be sent to different slaves, and the responses may come back with different delays, hence the master NI shell also interleaves the responses in the correct order. A slave may be used by different masters. Hence the slave NI shell multiplexes requests of different masters and demultiplexes the responses. The NI Shell FSMs implement the (de)serialization, reordering, and handshaking for the particular protocol of the

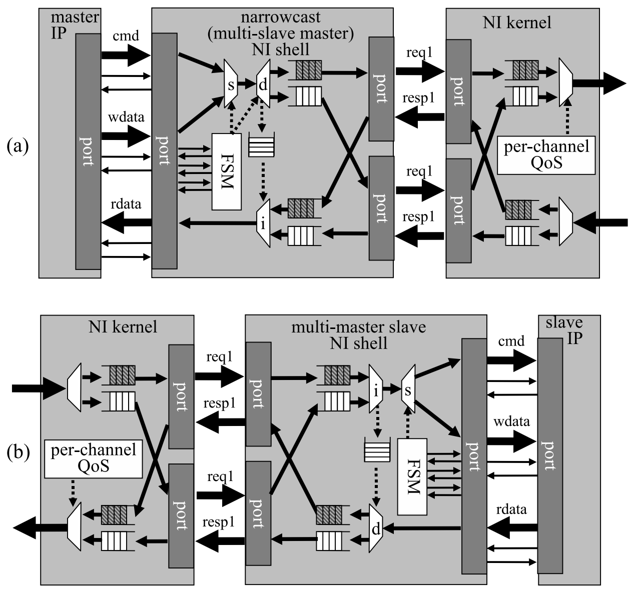

Figure 3. (a) Narrowcast (Multi-Slave) Master and (b) Multi-Master Slave, with their Shells and Kernels.

port. In Section 4.3 we describe these FSMs and how they have been modified to support our debug methodology.

3. Communication Debug

For run-stop type debugging, a debug engineer first has to determine at what point during the functional execution (the so-called breakpoint) the internal state of the embedded system needs to be examined. This decision is typically based on the point in time at which faulty behavior becomes visible on the system outputs, and setting an internal breakpoint prior to that moment. For communication-centric debug, several choices exist for the granularity at which the system execution can be stopped, and subsequently single-stepped, see Figure 4.

The coarsest granularity that is useful for debugging the on-chip communication is the connection level. This comprises the communication between a single master and all of its slaves. This debug level is sufficient to determine whether the master generates correct read and write transactions and the slaves react in the correct way. This correctness can be determined for example through a correlation with a behavioral simulation of the same system. This however does not explain why (attributes of) the transactions on a particular connection or from a particular slave are incorrect. To determine this, a smaller granularity may be required, for example at the level of individual slaves (i.e. channels), messages (i.e. specific requests or responses on

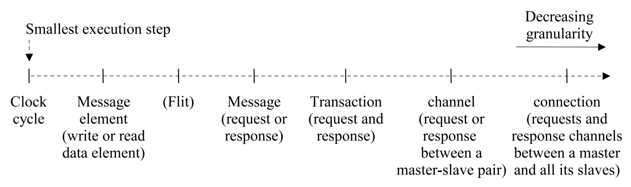
Smallest execution step
Decreasing granularity
Clock cycle
Message element (write or read data element)
(Flit)
Message (request or response)
Transaction (request and response)
channel (request or response between a master-slave pair)
connection (requests and response channels between a master and all its slaves)

Figure 4. Communication Debug Granularity.

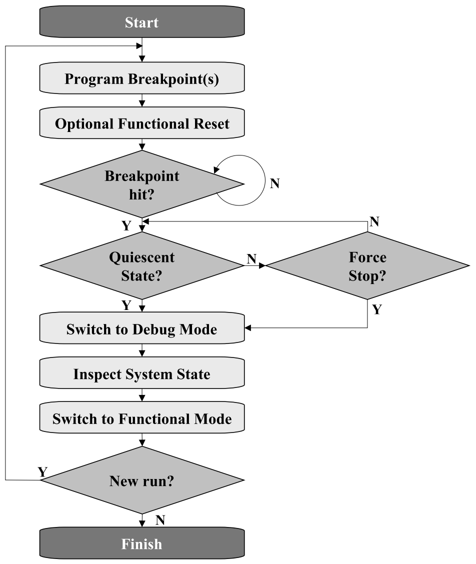

Figure 5. Communication-centric Debug Session.

a channel), or even elements within a message. If the communication infrastructure itself is suspect, debugging might need to take place at the even lower, flit level. Finally, the smallest granularity at which the execution of the system can be controlled, and hence stopped, is the clock-cycle level. Note that when stopping a system at a higher granularity than clock cycles, the stopping may not be immediate, i.e. the system may continue to execute for a certain period of time after the breakpoint was detected, for example to complete an active message or transaction.

Based on these additional levels of granularity, a new flow for a communication-centric debug session can be derived. It is shown in Figure 5. After programming the breakpoint, the engineer can choose to functionally reset the application, to start its execution from a well-defined start-up state, or to let the system continue as is. The external debugger software then continuously checks the execution state of the system, to determine whether the programmed breakpoint has already been hit or not. Once the breakpoint has been hit, the debugger software also has to check whether the system has reached a quiescent state, for example by polling the state of the communication queues in the NI shells. Especially with the transaction granularity, the time between the breakpoint hit and the system communication reaching a quiescent state may take a long time when large transactions are used.

Either the system reaches a quiescent state by itself, or the user has to force the subsequent switch to debug mode. Once in debug mode, the debug engineer has access to the contents of all internal registers and memories, via for example the manufacturing scan-chains [10].

Once the system's state has been inspected, it may be required to restart and/or resume the system's execution to stop at another point (earlier or later) in time for more accurate analysis of the error. To this end, the breakpoint can be reprogrammed, and the execution of the system is either restarted (by resetting the system), or resumed (by reenabling system execution control).

From the description of a communication-centric debug session, we derive the following control requirements to debug the on-chip communication:

- Reset: Functionally reset the system to (re)start the execution from a well-defined, start-up state.
- Internal stop: Stop initiated by an on-chip monitor programmed to recognize and trigger on a condition or sequence on internal signals. When these triggers reach the network interfaces, they may take effect at different levels of granularity (see Figure 4).
- External stop: Stop initiated by the user from the external debugger tool. Due to the latency of the debug channel through which this stop command is communicated, it is often very difficult to precisely control the point at which the system actually stops executing, hence the predominant use of on-chip monitors.
- Continue: Resume functional execution of the system.

The traditional single-step operation also exists for communication debug, but is not explicitly mentioned as a requirement, as a single-step action is the combination of a continue action with a subsequent stop action at a user-specified granularity. The breakpoint programmed can be either an absolute or a relative breakpoint. For single-stepping, a relative breakpoint is used, where the breakpoint is set after the next clock cycle, flit, element, message, or transaction, depending on the user's granularity requirements.

4. On-Chip Debug Infrastructure

In this section we describe the on-chip debug infrastructure that supports a communication-centric debug session as shown in Figure 5 and that meets our communication debug requirements. An overview of this infrastructure is shown in Figure 6. The components specifically added to provide debug support are shown in light gray.

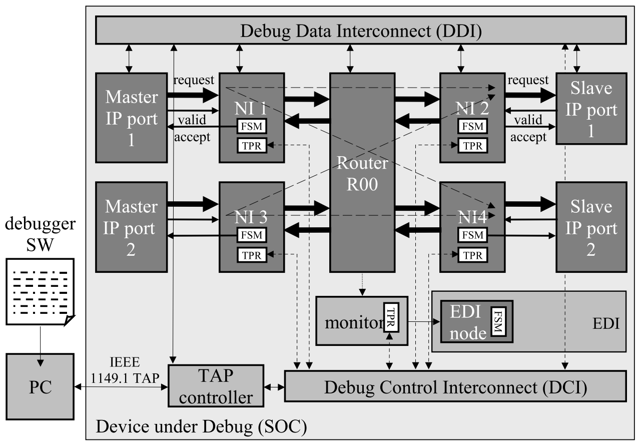

Figure 6. DfD Infrastructure for Communication Debug.

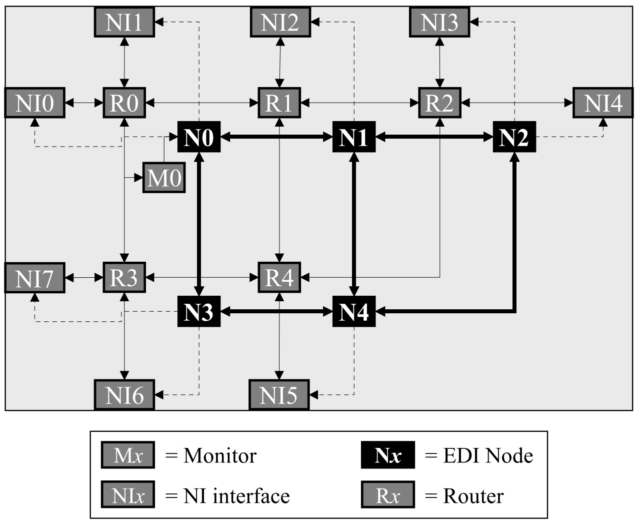

Figure 8. Example Event Distribution Interconnect.

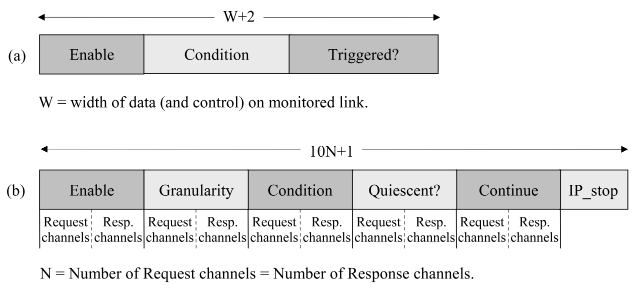

Figure 7. (a) Monitor TPR, and (b) NI Shell TPR.

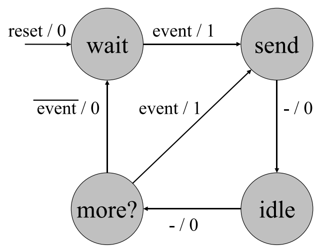

Figure 9. FSM of the Event Distribution Interconnect Node.

4.1. On-chip Monitors

Monitors may be added to a system to observe the progress of the computation in the master and slaves, and/or the communication in the communication architecture. Communication monitors observe the data on the interfaces at the boundaries of the network [17], and/or on internal links [2, 3], routers, NIs, etc. Under which conditions a monitor generates a trigger can be programmed via the Debug Control Interconnect (DCI). In our case, the DCI consists of a daisy-chain of, among others, Monitor Test Point Registers (TPR) (see Figure 6). A Monitor TPR contains a breakpoint condition, and its enable and triggered flags (refer to Figure 7(a)). The TPR chain is accessible from an IEEE 1149.1 Test Access Port (TAP) using a special debug instruction (see Subsection 4.5). This access mechanism is identical to the DCBs in [19]. Once the monitor detects the programmed breakpoint condition on the link or interface it observes, it asserts its output for as long as the breakpoint condition remains true.

4.2. Event Distribution Interconnect

The output signals of the monitors are connected to the Event Distribution Interconnect (EDI). The basic component of the EDI is the EDI node. The EDI node is parametrizable in the number of neighboring nodes. The EDI follows the topology of the communication architecture (for an example with one monitor, refer to Figure 8).

The FSM diagram of an EDI node is shown in Figure 9. Upon a functional reset, this FSM enters the `wait` state, in which it waits for an incoming event signal from its nearby monitor or other EDI nodes. When an event is detected, the FSM transitions to the `send` state, while it broadcasts the event to all its neighboring EDI nodes. In the next clock cycle, the FSM transitions to the `idle` state where it deactives its outgoing event signal, and ignores any incoming returning event signals from its neighbors. This state is key to the attenuation of the event signals in the EDI, as it ensures that eventually the entire EDI will be free of event signals again. In the next clock cycle, the FSM transitions to the `more?` state, where it checks whether the event input

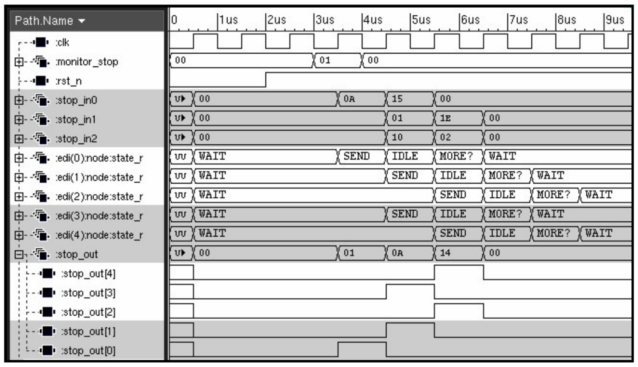

Figure 10. Event Distribution Example.

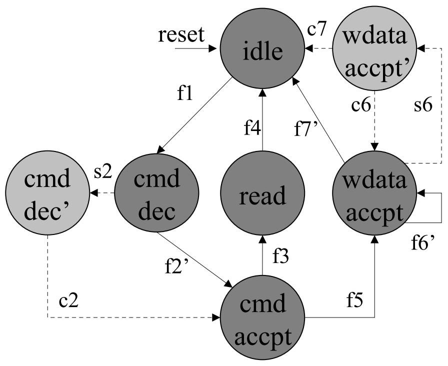

Figure 11. Modified Network Interface FSM for a Narrowcast Master.

signal is still asserted. If so, it will transition back to the `send` state, while broadcasting the event to all its neighboring EDI nodes. If the event input signal is deasserted, the FSM transitions to the initial `wait` state, where it again resumes to wait for an incoming event signal.

For the example EDI shown in Figure 8, the concerted operation is shown in Figure 10. Monitor M0 asserts its output (`monitor_stop[0]`), thereby signalling an event to EDI Node N0. EDI Node N0 transitions to the `send` state while asserting its output signal, `stop_out[0]`. In the next clock cycle, its neighboring EDI nodes, N1 and N3, take similar action, to signal the remaining EDI nodes, N2, and N4 via `stop_out[1]` and `stop_out[3]` respectively. Consequently all nodes go through the state sequence `wait` → `send` → `idle` → `more?` → `wait`. Afterwards, the complete EDI is in the same state as it was before the event came in from the monitor, but in between all network interfaces have been informed of this event, through the assertion of the `stop_out[i]` signals.

It takes the EDI a single clock cycle to propagate the pulses generated by a monitor through a EDI node. Given a communication architecture that communicates data at the granularity of flits (3 cycles for the Æthereal NOC), this ensures that any monitor event always reaches the borders of the network ahead of the data itself. This is a key debug feature we exploit, as it allows this data to be kept within the borders of the communication architecture for an (potentially) infinite amount of time. The actual processing of this data by the receiving IP can then be analyzed in the necessary detail required to find an error cause, by subsequently single-stepping the delivery operation for this data at the required debug granularity.

4.3. Network Interface Debug Operation

We illustrate the functional states and transitions of the NI shell FSMs, and then describe how they are modified for debug. Figure 11 shows the FSM of the narrowcast NI shell for a DTL master port as shown in Figure 3(a). Other NI shell FSMs are similar. Please refer to Figure 1 for the relevant signals and groups of a DTL port, which we use in our implementation. The states of the FSM serialize and handshake the DTL signal groups in the correct order (`cmd dec` and `cmd accpt` for the command, then `read` for read data or `wdata accpt` for write data). The `cmd dec` state decodes the address group to select the channel corresponding to the right slave, which is the defining feature of the narrowcast connection that implements distributed-shared-memory communication.

For the communication to be stopped when a breakpoint is detected, this FSM needs to be adapted. The states that are responsible for handshaking are duplicated in so-called shadow states. These are the lighter gray states in the state diagram, with an apostrophe appended to the name of the original state. Shadow states differ from their original counter-part. First, when in a shadow state, the FSM deactivates the NI shell's handshake signals, causing communication between the master and NI shell ports to (eventually) stop. Second, to take an FSM out of a shadow state, a signal from the external debugger software is required.

In the particular FSM of Figure 11, the stop transitions `s2` and `s6` are equal to the original `f2` and `f6`, but include checking that the channel should be stopped, and that an unconditional stop or stop condition occurs: `(stop_enable[i] = logic-1) AND ((stop = logic-1) OR (stop_condition[i] = logic-1))`, where `i` is the channel identifier. `f2'` and `f6'` are modified from `f2` and `f6` respectively by including the negated stop condition. The continue transitions `c2`, `c6`, and `c7` are equal to the original `f2`, `f6`, and `f7` ANDed with the `continue[i] = logic-1` signal, respectively. A general recipe for other protocol FSMs can be easily derived from this example.

The `stop_enable`, `stop_condition`, and `continue` signals come from the NI shells TPRs, described in the following section. They control how the NI shell hardware reacts to incoming events on the stop signal. TAP controller instructions set and read the TPRs, as described in Section 4.5. The user uses a higher-level debug API, defined in Section 5, built on top of the TAP controller instructions.

4.4. Network Interface Debug Control

The debug signals required to control the state progression of the NI shell FSMs originate from an NI-shell TPR (see Figure 6). All NI-shell TPRs are included in the TPR daisy-chain described earlier in Section 4.1. The NI TPR is a data register that provides the user with all required debug control over the interconnect interactions. By programming the various NI TPRs the user can achieve transaction, message, and/or element debugging per channel. As shown in Figure 7(b), the NI TPR consists of 5 fields: `stop_enable`, `stop_granularity`, `stop_condition`, `continue`, and `ip_stop`. All but the last field are present per channel.

1) Stop Enable: This field indicates whether the communication on a particular channel is stopped on an internal event or not. A `logic-0` means that the communication on this channel does not stop when an event signal is received from the EDI. Also a possible software stop is ignored (see the description of the `stop_condition` field below). A `logic-1` stops the channel on the conditions, specified by the `stop_granularity` and `stop_condition` fields.

2) Stop Granularity: This field controls the granularity at which the communication on a certain channel is stopped. A `logic-0` and `logic-1` allow ongoing messages and elements, respectively, to complete before stopping. The latter will stop the channel faster.

3) Stop Condition: Provided that stopping has been enabled (i.e. `stop_enable` set to `logic-1`) for the channel, and the appropriate stop granularity has been set, this bit determines under which condition this channel will stop. A `logic-0` means that the channel will stop only after a pulse from the EDI has been received. A `logic-1` means the channel will stop unconditionally before the next element, at the granularity specified by the `stop_granularity` bit. This channel will stop irrespective of whether a stop pulse arrived from the EDI or not.

This field gives the user the flexibility to either wait for a stop pulse from the EDI (i.e. for an absolute breakpoint or an external stop command), or program a channel to be stopped unconditionally (for a pre-programmed or forced user stop). There are two reasons for providing this field. First, in case of (infinitely) long transactions or errors on the interface, the user can stop the NOC by programming this field without having to wait for a transaction to complete. Second, a single-step consists of a continue atomically followed by an implicit unconditional stop. This field enables this implicit stop.

4) Continue: The stop combined with the continue gives the user the power to observe the functional behavior of the SOC in a controlled fashion during debug. The continue field is interpreted differently from the other fields. Writing `logic-1` in the continue TPR causes an active-high signal to be fed to the NI shell. Upon continuing communication, the shell resets this signal's value automatically through special reset logic. Setting this bit to `logic-1` is thus interpreted as a single continue pulse for the channel.

A continue with the appropriate stop condition therefore ensures an atomic continue and stop, to ensure that exactly one handshake takes place. This accuracy cannot be guaranteed by separate continue and stop commands because they involve user interaction, TPR programming, stop event distribution, etc. all which take time, during which an IP may execute multiple handshakes.

5) IP Stop: Every NI shell TPR also has a single `ip_stop` bit which enables the NI shell to forward an event to the connected IP cores. This is used for a functional stop request for the IP cores, enabling the stopping all the components (the interconnect and the IPs) of a SOC close to each other in time. Otherwise, only stopping the interconnect without the IP cores means that the computational state of the IP cores might still advance as they continue internal, non-communication-related operations. This complicates debug as the states of different parts of the system retrieved later on may be difficult to correlate to one another.

A `logic-0` means not to signal the connected IP cores to stop. Setting the value to `logic-0` is also used to signal a continue action when the IP cores were previously stopped using this method. A `logic-1` signals a stop to the connected IP cores, when a trigger event comes in via the EDI.

4.5. Extra TAP Controller Instructions

The entire on-chip debug infrastructure is controlled and programmable through an IEEE 1149.1 Test Access Port (TAP). A TAP is often already included in a chip design to allow board-level manufacturing test. To support communication-centric debug, the controller associated with the TAP has been extended with a number of user-defined instructions:

- `DBG_RESET`: issue a functional reset of the chip.
- `PROGRAM_TPR`: program the monitor and NI TPRs. The former determine the breakpoint condition(s). The latter control the resulting debug control actions.
- `QUERY_TPR`: query the status of the breakpoint (triggered or not) and the channels (whether there are still on-going transactions) in the NI shells.
- `JTAG_STOP`: send a trigger pulse to the EDI from the TAP.

- PROGRAM_TCB: switch the system between functional and debug modes.
- DBG_SCAN: scan out the complete state of the system via the scan chains in debug mode.

These instructions implement safe reading and writing of TPRs (which can be non-trivial due to the difference in debug and functional clock domains). They hide SOC-dependent implementation details of the TPRs in scan chains, etc. These generic instructions are however still fairly low level for an end user because (s)he would have to know the exact TPR layouts of Figure 7 and their positions in the TPR chain. It is for this reason that we defined a higher-level software API, which is described in the following section.

5. Off-chip Debugger Software API

We extended the TCL interface of our hardware debugger [15] to control the debug functionality in a user-friendly manner. The following API functions are implemented:

• reset : Issues a functional reset of the system by using the DBG_RESET instruction.

• set_bp <monitor> [<condition>]: Sets up the <condition> in the monitor's TPR. When the optional <condition> field is left out, the breakpoint setting is cleared. This call uses the PROGRAM_TPR instruction to program the appropriate monitor TPR bits via the TAP. Here, and below, <monitor>s and <channel>s are specified using their full, hierarchical design names.

• set_bp_action <channel> [<granularity> <condition>]: Sets up a breakpoint action on the channel. The <granularity> is one of transaction, message, or element. The <condition> is edi or always. When the optional <granularity> and <condition> fields are left out, the breakpoint action setting is cleared. This call uses the PROGRAM_TPR instruction to program the appropriate NI shell TPR bits via the TAP.

• get_mon_status [list of <monitor>s]: Returns an ASCII string, indicating whether the specified monitors have triggered (logic-1) or not (logic-0).

• get_ni_status <ni>: Returns an ASCII string indicating whether the channels in the specified NI are idle (logic-1) or not (logic-0).

• continue [list of <channel>s]: Causes the communication on the channels to continue. If the optional field is left out, all channels are continued. This call uses the PROGRAM_TPR instruction to set the continue bits in the appropriate NI TPRs to logic-1 via the TAP.

• synchronize: Retrieve the complete state of the system by first switching the system to the debug mode, using the PROGRAM_TCB instruction, and subsequently scanning out the manufacturing test scan chains, using the DBG_SCAN instruction. Then the system is switched back to functional mode, using the PROGRAM_TCB instruction. The complete state is stored in an internal database for subsequent query by the user.

6. Experimental Results

6.1. Example Use Case

In this subsection we show how the debug infrastructure and the software API work together on an example. Our automated design flow [6] generated the system shown before in Figure 6. This includes the RTL VHDL of the NOC, the clock and reset controllers, test bench and traffic generators, embedded C code to program the NOC, and scripts for gate-level synthesis, scan-chain insertion, etc. Each master has a connection to both slaves. In Figure 12 we show signal traces of the gate-level implementation of the NOC with scan chains. We boot the system until it is running in functional mode (omitted from the trace). The system is debugged first at the message level, and then element level. This is accomplished by Script 1, which uses the software API defined above to control the on-chip debug support.

Script 1 Example Debug Script

```
1: set_bp top.R00.M 378
2: set_bp_action {top.NI1.ch1} edi
3: while {[get_mon_status top.R00.M] eq "0"} {}
4: while {[get_ni_status NI1 ] ne "1111"} {}
5: set_bp_action {top.NI1.ch1} always
6: continue {top.NI1.ch1}
7: continue {top.NI1.ch1}
8: set_bp_action{top.NI1.ch1} element always
9: for {set i 0} {$i<5} {incr i} { continue }
10: set_bp_action {top.NI1.ch1} element edi
11: continue
```

Line 1 sets a breakpoint at the monitor attached to router R00, to match the value 378 on the output links of the router. Line 2 specifies that the channel top.NI1.ch1 between Master 1 and Slave 2 is sensitive to events generated by the monitors and user (via the TAP) (edi). Channel ch0to Slave 1 continues to operate. On reception of an event from the EDI, the NI finishes the ongoing message (message). These two commands are executed by the off-chip debugger software, which uses the TAP and DCI (i.e. the TPR chain) to load the appropriate values (Section 4.4) in the monitor and NI TPRs. This is for example shown by the transition of stop_enable, labelled A, in Figure 12.

Line 3 polls the monitor TPR to see if it triggered. After a number of transactions (box labelled B), the monitor triggers, which is shown by the transition on signal ni_stop_in labelled C. NI1 completes the ongoing message on the channel between Master 1 and Slave 2. It then stops, i.e. does not accept the messages for Slave 2 offered by the master (command valid is high, see label D). In line 4 the TPR of NI1 is checked. First that there are no packets in transit on channel ch1 containing (parts of) messages (cf. live_tx_wr_r_1 and live_tx_rd_r_1). Second that all credits have arrived in the producer's NI [9].

Line 5 changes the sensitivity of channel ch1 to single-step mode (always), i.e. only a single message is accepted before stopping again. This is visible at label E, where the stop condition changes. Line 6 continues operation of channel ch1 (label F). Immediately, the write request message that was waiting (label D) is accepted, sent to Slave 2, and executed. Immediately after, Master 1 offers a read transaction, but this command is not accepted. All this is shown in box G. When line 7 is executed, the waiting read request is accepted, executed, and the corresponding response data consumed (see the box labelled H). The read data dtl_rd_data transitions from "xxx" to a defined value, but this is hard to read in the signal trace due to the timescale used. We then change the stop granularity of channel ch1 to the element level, line 8, label I), followed by 5 continue commands (line 9). In the boxes labelled J and K, five elements are accepted: the command dtl_cmd_accept and 4 data elements on dtl_wr_accept. Finally, line 10 (label L) makes channel ch1 sensitive to the EDI only, i.e. no single stepping. Label M shows how the system continues at full speed after a continue pulse.

All debug commands are given from the debug clock domain. The system operates on the functional clock, and parts of the system that are not debugged operate normally. For example, although not shown for lack of space, throughout the example the other master can continue to send transactions to both slaves. Figure 12 has been obtained with a simulator. To debug an FPGA or real silicon, the synchronize call has to be used to download the state of the chip to the debugger, or vice versa. This means that the entire system, including master 2, has to transition from functional mode to debug mode. In general, clock cycle synchronization leaves the system in a potentially inconsistent state due to clock-domain crossings that do not utilize valid/accept handshakes. Proper continuation can then only be achieved be executing again from a system reset. In our case, however, the state of the NOC can be synchronized safely and independently from the clocks used by the IP cores because all interfaces do use valid/accept handshakes. At the start of the synchronize call, *all* channels have to be stopped, e.g. at the element level which is quickest, and they have to be re-enabled after the synchronization.

6.2. Required Silicon Area

For the example described above, the amount of silicon area needed to implement the proposed debug infrastructure is very low: a 2.5% increase of the NOC area, and no decrease in speed when synthesized at 250 MHz. The increase in area is almost entirely due to the TPRs; the area for the monitors and EDI nodes are neglible. Regarding timing, the NI shell FSM complexity is increased marginally, but this is not in the critical path. The EDI runs at NOC speed, and the DDI and DCI speeds are determined by the scan chains and boundary scan logic inserted for manufacturing test.

7. Conclusion

We presented a debug methodology to debug a SOC by concentrating on its communication. We applied it to a NOC because they represent the most complex interconnects. Our extended communication model includes handshakes for each of the multiple signal groups per IP port, and multiple handshakes per signal group (e.g. for read and write data elements). It also addresses narrowcast communication based on distributed shared memories, where a master transparently sends read and write transactions to multiple slaves in its address space. As a result, debug control is offered at three granularities: data elements, messages, and transactions. Orthogonally, it is offered per channel (master-slave pair), also within narrowcast connections. Different channels can be simultaneously debugged at different granularities.

We prove our concepts with an RTL implementation that is automatically generated by our NOC design flow. We show how to extend NI shell FSMs for general communication protocols with shadow states to suspend the valid/accept handshakes on the port interfaces. The monitoring and distribution of events is cleanly separated from how they are interpreted (the debug granularity per channel), in terms of hardware and programming. The software infrastructure has a clearly defined hardware interface (the TPRs and IEEE 1149.1 TAP with additional, generic debug instructions), and an intuitive high-level software API that uses it. The infrastructure offers powerful run-time programmable breakpoints, stopping, continuing, and single stepping at three granularities. In particular, single stepping is a non-trivial extension to atomically continue and stop, to guarantee that no event escapes detection.

Our debug infrastructure consists of hardware components (monitors and event distribution interconnect), and a software API and library. The hardware infrastructure is modular, requires very few changes to the NOC, and scales linearly with the size of the NOC in terms of area. The area cost is only 2.5% compared to the NoC and without speed penalty.

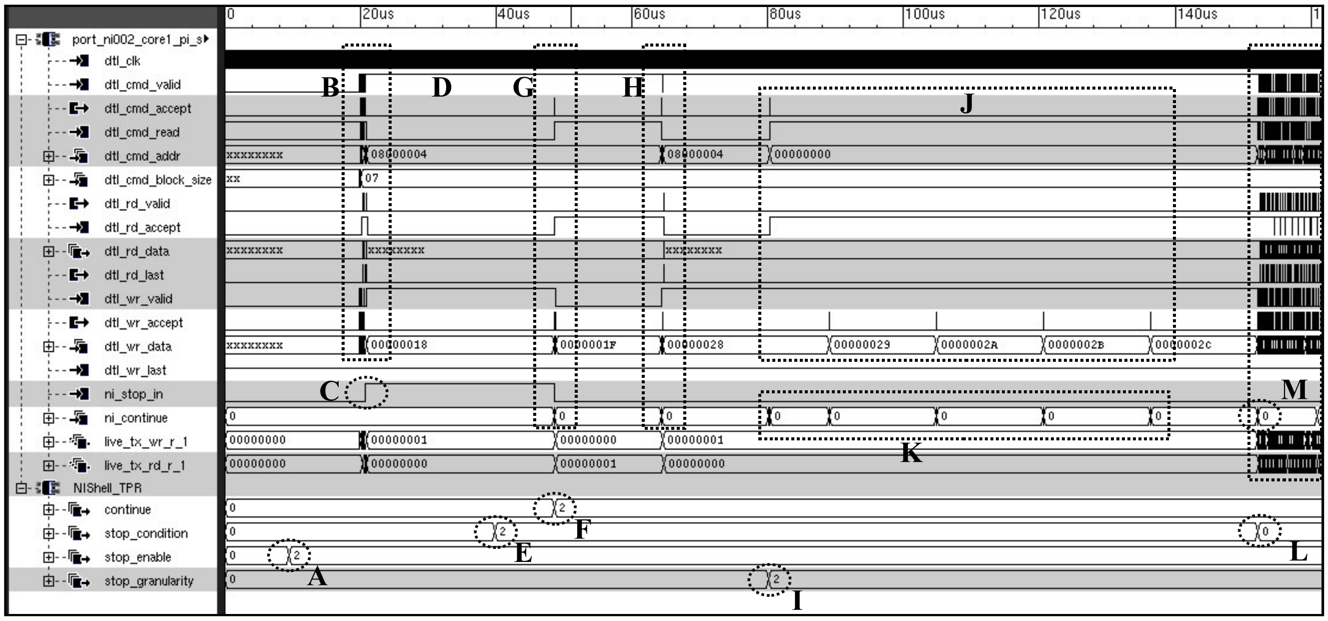

Figure 12. Traces of our Example Debug Session.

References

[1] ARM. *AMBA AXI Protocol Specification*, June 2003.

[2] C. Ciordaş, T. Basten, A. Rădulescu, K. Goossens, and J. van Meerbergen. An event-based monitoring service for networks on chip. *ACM Transactions on Design Automation of Electronic Systems*, Oct 2005.

[3] C. Ciordaş, K. Goossens, T. Basten, A. Rădulescu, and A. Boon. Transaction monitoring in networks on chip: The on-chip run-time perspective. In *Proc. Symposium on Industrial Embedded Systems (IES)*, Oct 2006.

[4] P. Dahlgren, P. Dickinson, and I. Parulkar. Latch Divergency in Microprocessor Failure Analysis. In *Proc. IEEE Int'l Test Conference*, Sep/Oct 2003.

[5] S. K. Goel and B. Vermeulen. Hierarchical data invalidation analysis for scan-based debug on multiple-clock system chips. In *Proc. IEEE Int'l Test Conference (ITC)*, Oct 2002.

[6] S. González Pestana, E. Rijpkema, A. Rădulescu, K. Goossens, and O. P. Gangwal. Cost-performance trade-offs in networks on chip: A simulation-based approach. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Feb 2004.

[7] K. Goossens, J. Dielissen, and A. Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, Sept/Oct 2005.

[8] 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)*, May 2007.

[9] 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)*, May 2007.

[10] K. Holdbrook, S. Joshi, S. Mitra, J. Petolino, R. Raman, and M. Wong. microSPARC: A case study of scan-based debug. In *Proc. IEEE Int'l Test Conference (ITC)*, 1994.

[11] A. Hopkins and K. McDonald-Maier. Debug support for complex systems on-chip: A review. *IEE Proc. Computers and Digital Techniques*, July 2006.

[12] R. Leatherman and N. Stollon. An embedded debugging architecture for SoCs. *IEEE Potentials*, Feb-Mar 2005.

[13] OCP International Partnership. Open core protocol specification, 2001.

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

[15] G. Rootselaar and B. Vermeulen. Silicon Debug: Scan Chains Alone Are Not Enough. In *Proc. IEEE Int'l Test Conference (ITC)*, Sept. 1999.

[16] A. Rădulescu, J. Dielissen, S. González Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, Jan 2005.

[17] S. Tang and Q. Xu. A multi-core debug platform for NoC-based systems. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2007.

[18] B. Vermeulen, K. Goossens, R. van Steeden, and M. Bennebroek. Communication-centric SOC debug using transactions. In *Proc. European Test Symposium (ETS)*, May 2007.

[19] B. Vermeulen, T. Waayers, and S. Goel. Core-based Scan Architecture for Silicon Debug. In *Proc. IEEE Int'l Test Conference (ITC)*, Oct 2002.