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
2 Computer Engineering, Delft University of Technology
overview

- transaction-based communication-centric debug

- traditional debug architecture & flow and NOC architecture
  - distributed shared memory (DSM)
  - communication model

- new debug architecture & flow and NOC architecture
  - debug granularity, DCI, TPR, EDI, FSM, TAP, API

- example

- conclusions
debug is…

- **error localisation** when a chip does not work in its intended application
- difficult due to **limited visibility** of the internal behaviour
- debugging first silicon uses >50% of project time
- unpredictable
- **negative impact** on
  - time to market
  - brand image
communication-centric debug

- processor debug is mature
- system debug complexity resides in the interactions between IP blocks
  - multi-processor debug is a challenge

- older interconnects serialized all transactions
  - a unique global communication trace
- latest interconnects allow split, pipelined, concurrent transactions
  - no unique communication trace
communication-centric debug

- traditional processor-centric debug focusses on control of the IP (computation)
- interconnect is the locus of all IP interactions
- we propose to focus debug on the interactions between IPs through control of the interconnect (communication)
transactions

- transaction
- request & response
- valid/accept handshake
  - signal groups
  - data words (elements)
- communication types
  - peer-to-peer streaming
  - distributed shared memory

```
initiator
  cmd_valid
  cmd_accept
  cmd_read
  cmd_addr
  cmd_block_size

slave 0x00-0x1F
  wr_valid
  wr_accept
  wr_data
  wr_last

slave 0x20-0xFF
  rd_valid
  rd_accept
  rd_data
  rd_last

target
```

master ↔ slave

- master
- slave 0x00-0x1F
- slave 0x20-0xFF
communication & debug granularities

- Clock cycle
- Message element (write or read data element)
- Message (request or response)
- Transaction (request and response)
- Channel (request or response between a master and 1 slave)
- Connection (requests and response channels between a master and all its slaves)

Coarser granularity

Finest grain that is based on handshake

Finest grain that is based on transactions required for distributed shared memory
debug flow

Start

Program Breakpoint(s)

Optional Functional Reset

Monitor(s) hit breakpoint?

Distribute event

Quiescent State?

Switch to Debug Mode

Inspect System State

Switch to Functional Mode

New run or continue?

Force Stop?

Finish

2008-04-08 NOCS
conventional master network interface

- **NI shell FSM** implements
  - protocol (de)serialisation (s)
  - distributed address map (d)
  - request/response ordering (i)
  - width conversion (not shown)

- **NI kernel FSM** implements
  - per-channel QoS
  - (de)packetisation
conventional slave network interface

- converse for slave shell
SOC architecture

Master IP port 1

NI 1
FSM

request
valid
accept
request
valid
accept
request
valid
accept
request
valid
accept
Slave IP port 1

Master IP port 2

NI 2
FSM

NI 3
FSM

NI 4
FSM

Slave IP port 2

Router R00
debug architecture: monitors

EDI distributed events from monitors to NI shells (and IP)
EDI node FSM

reset / 0 -> wait

wait

event / 1 -> send

send

event / 0 -> more?

more?

- / 0

idle

- / 0

event / 0

event / 1 -> idle
debug architecture: test point registers (TPR)

debug behaviour is controlled by TPRs
test point registers (TPR)

- **control debug behaviour**
  - link monitors: which conditions to monitor
  - NI shells: how to react to incoming events per channel
- operate on test clock

<table>
<thead>
<tr>
<th>Enable</th>
<th>Condition</th>
<th>Triggered?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$W = \text{width of data (and control) on monitored link.}$

<table>
<thead>
<tr>
<th>Enable</th>
<th>Granularity</th>
<th>Condition</th>
<th>Quiescent?</th>
<th>Continue</th>
<th>IP_stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request channels</td>
<td>Resp. channels</td>
<td>Request channels</td>
<td>Resp. channels</td>
<td>Request channels</td>
<td>Resp. channels</td>
</tr>
</tbody>
</table>

$N = \text{Number of Request channels} = \text{Number of Response channels.}$
NI shell FSM

- stop conditions (s2, s6)
  - original_condition and stop_enable and (stop or stop_condition)
- modified transitions (f2', f6', d7')
  - original_condition and not (stop_enable and (stop or stop_condition))
- continue conditions (c2, c6, c7)
  - original_condition and continue
- protocol serialisation can now be stopped & resumed
- general recipe for different protocols
debug architecture: debug control interconnect

TPRs are controlled by DCI (dedicated asynchronous scan chain)
debug architecture: scan chains, clock control, etc.

down/upload functional state using DDI (scan chains for structural test)
debug architecture: software control API

- The debug architecture is controlled using IEEE1149.1 test access port from a PC running debug software.
- Basically can down/upload system state, on the test clock.
- Separate scan chains for debug control/status and functional state.
  - Can modify debug state independently from functional state, and during functional mode.
- “High-level” functions to get/set debug state:
  - reset
  - set_bp_monitor <condition>
  - set_bp_action <channel> <granularity> <condition>
  - get_mon_status <monitor>
  - get_ni_status <ni>
  - continue: set continue bits in NI TPRs
  - synchronise: down/upload entire SOC state
while the system is running in functional mode
set breakpoint on value 378 in link monitor
make channel between master 1 & slave 2 sensitive to events (A)

**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} {inc i} {continue }`
10. `set_bp.action {top.NI1.ch1} element edi`
11. `continue`
- while polling the monitor
- after a number of transactions (B)
- it triggers and the NI receives a stop event (C)
- NI completes ongoing message & ignores next request (D)
- after checking that there are no transactions in flight
- program NI to single-step mode with message granularity (E)
- and continue (F)
- the NI accepts a single write request (G)
- and continue again (read request, H)

**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
example

- change debug granularity to word (data element) (I)
- and continue 5 times
  - one command and four data handshakes (J, K)
example

- change debug sensitivity to EDI only (i.e. no single stepping) (L)
- communication resumes at full speed after continue pulse (M)
- all this time, the rest of the system could have been in functional mode
conclusions

- **debug scope**
  - per channel (master-slave pair)
  - per connection (master with all its slaves)

- **debug granularity**
  - data words (equivalently: valid/accept handshake)
  - request/response
  - transaction

- all channels can be debugged or not, at any granularity, independently

- required for **distributed-shared memory** debugging

- **debug architecture**
  - **re-uses** existing functional & test infrastructures (e.g. scan chains)
  - simple programmable **building blocks** (monitors, TPRs)
  - **general recipe** to modify functional NI shell FSM for debug
  - very basic software API