UNIVERSITY OF TURKU

Design & Verification of Bus Monitor in Debug and Trace sub-system in Event Socket

MASTER'S THESIS

AUTHOR: RUPESH BASNET
DEPARTMENT OF FUTURE TECHNOLOGIES
UNIVERSITY OF TURKU, TURKU, FINLAND
SUPERVISORS:
DR. TOMI WESTERLUND, UNIVERSITY OF TURKU
KIMMO LAAKSONEN, NOKIA-FI/ESPOO
JARI LAHTINEN, NOKIA-FI/ESPOO

ABSTRACT

University of Turku
Department of Future Technologies
Rupesh Basnet: Design and Verification of bus monitor in Debug and Trace Subsystem in Event Socket
Master's Thesis, 78 pages + 23 pages appendices
Master's Degree in Technology
August 2018
Keywords: Event Socket, bus monitor, event, trace, bus monitor, CoreSight, Design, Verification

This thesis introduces the concept behind the Event Socket (ES) HW and debug and trace architecture in ES, a hardware accelerator targeted for a baseband SoC. The SoC handles the baseband layer 1 processing for multi-RAT (radio access technology), both 4G (LTE) and 5GNR (new radio). The motivation behind ES boils down to the bottleneck that Amdahl's law infers. ES is essentially used for dynamic load balancing among heterogenous set of processing engines such as processors, DSPs, microcontrollers, ASIPS and other hardware accelerators.

The work done for this thesis involves the register transfer level (RTL) implementation of the bus monitor in DTSS architecture and its verification. Bus monitor unit in DTSS is non-trivial. It is responsible for capturing the transaction non-invasively on the interfaces it is connected to and produce a trace input data for ARM CoreSight architecture. Verification of a system design is critical. Pre-silicon verification of an SoC ensures that the design works as per the requirement. The verification in this work is based on UVM. The hardware description language used for the work is VHDL. DTSS architecture in ES has bus monitors to monitor the interfaces along with the standard ARM CoreSight components like System Trace Macrocell and Embedded Trace FIFO.

The requirements include the features such as data capture, extraction, filtering and AXI translation for the bus monitor. These features were verified against the output from a reference model. In addition, register access was also verified. VIP from the scratch was developed for the bus monitor functional verification while for the register access, existing Nokia AXI VIP was used.

The DTSS in the event socket allows non-intrusive trace of the hardware events inside the event socket thereby ensuring the correctness of the SW. In the SoC level, ES debug and trace architecture is instantiated in DTSS sub-system of the entire SoC.

ACKNOWLEDGEMENTS

I would first like to thank Event Socket lead, Kimmo Laaksonen for pointing me the potential topic for my master's thesis. It was only after his pointer that I thought of doing my master's thesis in this topic. Kimmo not only pointed me to the potential thesis topic but also "rescued" my thesis work from the "storm" that swept all the colleagues working in the Event Socket project to Beamer project. I am also grateful to my line manager Jani Lemberg who let me continue my thesis work even during the "beamer storm".

I would also like to thank Jari Lahtinen, Debug and Trace sub-system lead for his support and guidance during the project. Jari has made the technical details in the design simple enough for me to understand and implement. I really appreciate his efforts.

Furthermore, I would like to extend my sincere appreciation and thanks to my thesis supervisor from the University of Turku, Embedded Computing department, Dr. Tomi Westerlund for accepting to be my thesis supervisor and also for the opportunity that he provided me during the study period to carry out the summer internship in the department.

It would not have been possible without the support of my beloved wife, Bhawana Adhikari who stayed home and looked after our newly born amazing little boy, Ryan Basnet. I am grateful that Ryan has come to our lives (May 2017) and since has always been our source of inspiration.

Table of Contents

1. INTRODUCTION **1**
2. THEORETICAL BACKGROUND **4**
2.1 BASEBAND L1 SOC 4
2.2 EVENT SOCKET 6
2.2.1 Event Manager *7*
2.2.2 Event Manager EMA *8*
2.2.3 Buffer Manager *9*
2.2.4 Buffer Manager EMA *11*
2.3 DEBUG AND TRACE SUB-SYSTEM IN EVENT SOCKET 12
2.3.1 Bus Monitor *13*
2.3.1.1 Data Capture Unit 14
2.3.1.2 Extraction Unit 14
2.3.1.3 Filtering Unit 15
2.3.1.4 AXI Writer Unit 16
2.3.1.5 Trace Control Block 16
2.3.2 ARM CoreSight *16*
2.3.2.1 System Trace Macrocell 18
2.3.2.2 Embedded Trace FIFO 19
2.3.2.3Trigger Components 20
2.3.2.4 Timestamp Components 20
2.3.3 NIC-400 Cross-Bar Interconnect *21*
2.3.4 Hardware Interfaces *22*
2.3.4.1 AXI Interface 22
2.3.4.2 Valid-ready handshake Interface 24
2.3.4.3 Hardware Event Observation Interface 24
2.3.5 Software Interfaces *25*
2.3.5.1 Debug Advanced Peripheral Bus (APB) Interface 25
2.3.5.2 Advanced Trace Bus Interface 26
2.4 VERIFICATION ENVIRONMENT 26
2.4.1 Sequence Item *28*
2.4.2 Sequence *28*
2.4.3 Sequencer *28*
2.4.4 Driver *28*
2.4.5 Monitor *28*
2.4.6 Scoreboard *29*
2.4.7 Interface *29*
3. RELATED WORKS **30**
3.1 OPEN EVENT MACHINE 32
3.2 NEXUS 5001 36
3.3 DEBUG SUPPORT ARCHITECTURE 35
4. IMPLEMENTATION **33**
4.1 DESIGN IMPLEMENTATION 33
4.1.1 Register Bank Generation *33*
4.1.2 Data Capture Unit Implementation *40*
4.1.3 Extraction Unit Implementation *41*
4.1.4 Filtering Unit Implementation *42*
4.1.5 AXI Writer Implementation *43*
4.1.6 Trace Control Block Implementation *45*
4.2 VERIFICATION IMPLEMENTATION 46
4.2.1 UVM Environment Implementation *47*
4.2.1.1 Register Access Test *49*
4.2.1.2 Trace Control Block Test *51*
4.2.1.3 Data Capture Unit Test *52*
4.2.1.4 Extraction Unit Test *54*
4.2.1.5 Filtering Unit Test *56*

4.2.1.6 AXI Writer Unit Test *58*
4.2.2 BUILD ENVIRONMENT SETUP 58
5 RESULTS AND DISCUSSION 61
5.1 DIRECTED TEST RESULTS 61
5.1.1 Data Capture Unit Test Result *61*
5.1.2 Extraction Unit Test Result *62*
5.1.3 Filtering Unit Test Result *62*
5.1.4 AXI Master Writer Test Result *63*
5.1.5 Trace Control Block Test Result *64*
5.2 UVM TEST RESULTS 66
5.2.1 Register Access Test *66*
5.2.2 Trace Control Block Test *66*
5.2.3 Data capture unit test *67*
5.2.4 Extraction Unit Test Result *69*
5.2.5 Filtering Unit Test Result *70*
5.2.6 AXI Writer Unit Test *71*
6 CONCLUSION AND FURTHER DEVELOPMENTS 73
6.1 FURTHER DEVELOPMENTS 74
REFERENCES 75
APPENDICES 1
APPENDIX 1: SIMULATION RESULTS FOR THE BUS MONITOR ON THE INTERFACE BETWEEN RX QUEUE AND EM 1
APPENDIX 2: SIMULATION RESULTS FOR THE BUS MONITOR ON APC INTERFACE 5
APPENDIX 3: SIMULATION RESULTS FOR THE BUS MONITOR ON EM CREDIT INTERFACE 9
APPENDIX 4: SIMULATION RESULTS FOR THE BUS MONITOR ON BM CREDIT INTERFACE 13
APPENDIX 5: SIMULATION RESULTS FOR THE BUS MONITOR ON ALLOCATION INTERFACE 16
APPENDIX 6: SIMULATION RESULTS FOR THE BUS MONITOR ON THE COMMAND INTERFACE 19

List of Tables

Table 1: Trace data fields [11] 14
Table 2: Trace Control Register [11] 40
Table 3: Address Mapping[11] 44
Table 4: Data alignment[11] 45

List of Figures

Figure 1: Speedup vs Number of processors 1
Figure 2: High-level view of a baseband L1 SoC 5
Figure 3: Event Socket block diagram 6
Figure 4: Event Manager block diagram 8
Figure 5: RX/TX Queue Entry [2] 8
Figure 6: EM EMA block diagram [8] 9
Figure 7: Buffer Manager block diagram 10
Figure 8: BM EMA block diagram 11
Figure 9: DTSS block diagram [11] 12
Figure 10: Bus Monitor block diagram 13
Figure 11: CoreSight sub-system block diagram [11] 17
Figure 12: STM Inputs and Outputs [13] 18
Figure 13: ETF Configuration [14] 19
Figure 14: Burst Write in AXI 23
Figure 15: Burst Read AXI 23
Figure 16:Valid-ready interface timing diagram 24
Figure 17: APB Write and Read 25
Figure 18: UVM Testbench 27
Figure 19: UVM Phases 30
Figure 20: Data capture Unit I/O interface 40
Figure 21: Sampling and data output state in DCU 41
Figure 22: Extraction unit I/O interface 41
Figure 23: Extraction process timing diagram 42
Figure 24: Filtering Unit I/O interface 42
Figure 25: Filtering process timing diagram 43
Figure 26: AXI writer I/O interface 44
Figure 27: Trace Control Block I/O interface 46
Figure 28: Bus Monitor Testbench Architecture 47
Figure 29: Data Capture Unit test setup 52
Figure 30: Extraction Unit Testbench 55
Figure 31: Filtering Unit Testbench 57
Figure 32: ModularMake [29] 60
Figure 33: Directed test result for data capture unit 61
Figure 34: Directed test result for extraction unit 62
Figure 35: Directed test result for filtering unit 63
Figure 36: Directed test result for axi master writer 63
Figure 37: Directed test result for trace control block 64
Figure 38: Directed test for the bus monitor 65
Figure 39: Register Access uvm test 66
Figure 40: Trace control block uvm test 67
Figure 41: Data capture unit uvm test 67
Figure 42: Extraction unit uvm test 69
Figure 43: Screenshot of the reference queue and output queue for extraction unit 70
Figure 44: Filtering unit uvm test 70
Figure 45: Screenshot of the reference queue and output queue for filtering unit 71

Figure 46: AXI Slave interface....................71
Figure 47: Screenshot of the reference queue and output queue for AXI writer unit 72
Figure 48: DTSS in SoC-level....................73

List of Abbreviations

ES - Event Socket

IP - Intellectual Property

SoC - System on Chip

LTE - Long Term Evolution

NR - New Radio

PCB - Printed Circuit Board

DTSS - Debug and Trace Sub-system

CPU-SS - CPU sub-system

DL/UL - Downlink/Uplink

PE- Processing Engine

L1/L2 - Layer 1 and 2 in 4G/5G protocol stack

EM - Event Manager

EMA - Event Machine Adapter

BM EMA - Buffer Manager EMA

AXI - Advanced Extensible Interface

RX - Receive

TX - Transmit(send)

HW - Hardware

SW - Software

STM - System Trace Macrocell

ETF - Embedded Trace Fifo

CTI- Cross Trigger Interface

DAP - Debug Access Port

TPIU - Trace Port Interface Unit

ETB- Embedded trace buffer

SWO- Serial Wire Output

TMC - Trace Memory Controller

FIFO - First in First Out

CTI - Cross Trigger Interface

CTM - Cross Trigger Matrix

AMBA - Advanced Microcontroller Bus Architecture

AHB - AMBA High Performance Bus

HEOI - Hardware Event Observation Interface

APB - Advanced Peripheral Bus

ATB - Advanced Trace Bus

UVM - Universal Verification Methodology

DPI - Direct Programming nterface

OpenEM - Open Event Machine

ODP - Open Data Plane

API - Application Programming Interface

OS - Operating System

NUMA - Non-uniform Memory Access

JTAG - Joint Test Action Group

IO - Input/Output

EDA - Electronic Design Automation

DUT - Design Under Test

RAL - Register Abstraction Layer

Chapter 1

1. Introduction

The speedup of any system is a relative measure of its performance. Performance is generally based on either the latency or the throughput of the system. Speedup, to certain extent, could be achieved by parallelizing the algorithm and increasing the number of available cores. However, it does not hold true as the number of processing cores keep increasing. In fact, it is such that the serial bottleneck in the algorithm ultimately confines the overall speedup. Figure 1 below conveys the idea well.

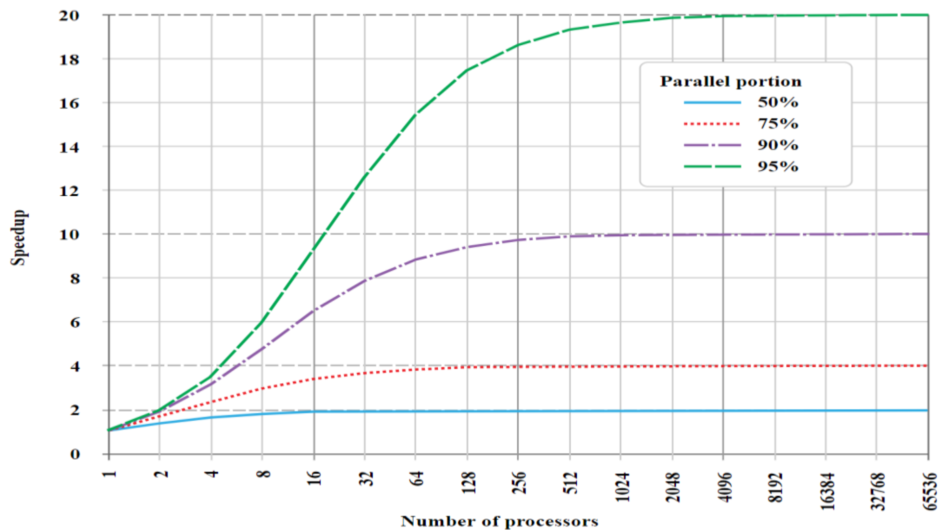


Figure 1: Speedup vs Number of processors [1]

As shown in Figure 1, speedup scales well with the parallelized algorithm as the number of processors increase but ultimately it is the serial nature of the algorithm that is the bottleneck. The idea that the serial nature of the algorithm being critical in determining the overall speedup of the system was put forth in 1967 by computer scientist Gene Amdahl and it is popularly known as Amdahl's law [1].

ES IP [2] tries to address the issue put forth by Amdahl's law by extending the support to offload the event processing and memory management to hardware thereby reducing the software load. ES implements the hardware acceleration logic for event processing which will be instantiated in SoC level. The job of the ES is to dynamically distribute the loads among the PEs such that the speedup tends to be linear with the number of available processing cores. The SoC that uses the ES IP is essentially L1 SoC [4] that handles baseband processing for 4G LTE and 5G NR.

Unlike PCBs, the observability and visibility of communication paths and interfaces in an SoC is internal to the device thereby making it impossible for external instrumentation tools to probe the communication paths. Increasing clock frequency and multi-core approach exacerbate the situation. The lack of monitoring mechanism makes it difficult to track down the bugs and hence increase the development time and consequently loose the consumer confidence as the system becomes less reliable. In [3], it is mentioned that about 77 % of electronic failures in automobiles stem from the software. Thus, a built-in debug and trace infrastructure is a must. Debug and Trace architecture in any design allows the programmer to trace the program execution thereby allowing them to debug their programs.

DTSS in ES makes it possible to keep track of data plane application program execution. DTSS inside ES IP is an architecture that allows the non-intrusive tracing of hardware events in the ES. DTSS in the ES is integrated to the Design and Trace sub-system on the SoC level which is the architecture responsible of debug and trace of the entire SoC. DTSS in ES is realized with ARM CoreSight. From onwards, it is advised to understand that DTSS refers to debug and trace infrastructure used in ES unless explicitly stated otherwise.

This thesis is structured such that chapter 2 presents the theoretical background of the design and a general overview of the UVM based verification environment; Chapter 3 presents the related works; chapter 4 describes the implementation;

chapter 5 presents the results and chapter 6 presents the conclusion and future development. The appendix section contains the simulation result other than the one presented in chapter 5.

Chapter 2

2. Theoretical Background

The debug and trace sub-system architecture in ES is miniscule in comparison to the design complexity of an entire L1 SoC yet it proves its non-triviality because of the fact that it is a fundamental block in allowing programmers to keep track of their program execution. This section tries to lay down the foundation for the readers to ultimately have a high-level picture of the entire baseband processing system and realize the scope of ES in a baseband SoC and in turn the scope of DTSS in ES.

The approach taken in this section of thesis is top to down. This section is structured such that section 2.1 presents a high-level view of a typical baseband L1 SoC, section 2.2 presents the theory behind ES and realize the scope of ES in a baseband L1 SoC and section 2.3 presents the DTSS in ES.

2.1 Baseband L1 SoC

The baseband L1 SoC [4] is responsible for baseband processing. The ES IP is targeted as hardware acceleration module for the Baseband L1 SoC. The transaction (data packet) coming from L2 is translated into events and put into the dedicated queues. ES operates on these events to achieve dynamic load balancing among various PEs. Figure 2 shows the high-level view of a Baseband L1 SoC and potential use case of ES IP in the SoC level.

The assumption made as per the following figure is that the higher layer protocols are handled by the CPU-SS (ARM cores) and Layer 1 is handled in the Baseband L1 SoC which typically has DSP cores for processing. The SoC has a dedicated downlink and uplink processing chain which does the heavy lifting of baseband processing. Albeit the Baseband L1 SoC, to which the ES IP is targeted for, support multi-RAT technology (LTE and 5GNR), the figure only shows a single DL/UL processing chain. The consideration of dedicated processing chains for 4G LTE and 5G NR is left to the reader's discretion.

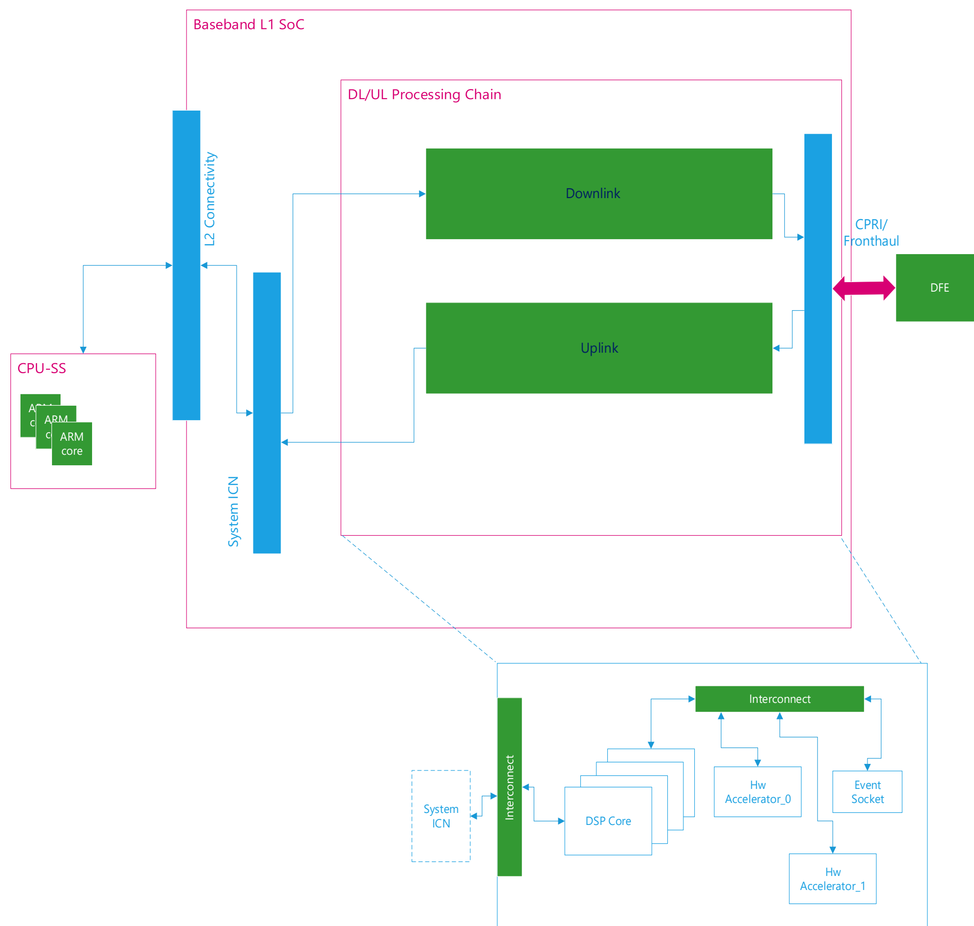


Figure 2: High-level view of a baseband L1 SoC

2.2 Event Socket

Event Socket (ES) [2] is a hardware accelerator intended to accelerate quadrature data [5] (IQ) processing. The primary idea behind ES is to implement the open event machine [6] functionality in hardware such that it does the heavy lifting of event processing in hardware to achieve a better performance. When the software requests a service, it is translated as an event and sent to the event queues. ES then operates on these events to achieve a dynamic load balancing among the processing engines (PEs). No event is tied to a particular PE rather the scheduling is done on the fly based on the availability of the core thereby achieving dynamic load balancing.

ES is an IP used in SoC level used to achieve dynamic load balancing. PEs process the input data and produce some output. These outputs are translated into events prior to sending to other PEs. The software (data plane applications) pushes the events to the dedicated local queues in ES HW. The events do not carry data payload with them but only the pointer to the data payload. The job of the ES is to dynamically distribute these events among various processing engines. Figure 3 shows the top-level view of the ES hardware architecture.

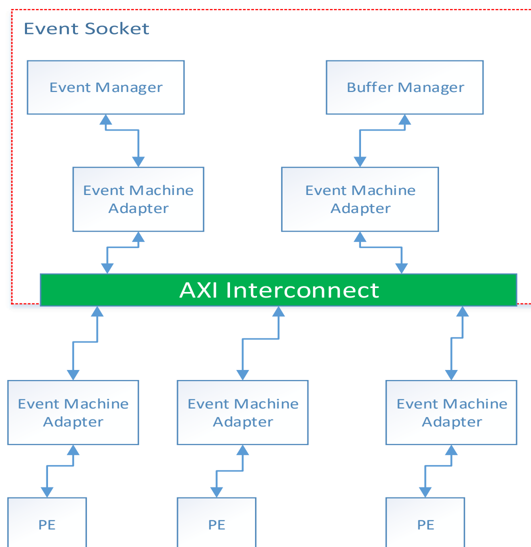


Figure 3: Event Socket block diagram

ES is built around four primary blocks; Event Manager (EM), Event Machine Adapter (EMA), Event Timer (ET) and Buffer Manager. Event Timer module is not shown in Figure 3 as the DTSS in ES does not generate any trace input packet out of the transaction in ET and hence is out of the scope of this work. Each of these building blocks has their own dedicated job in the event socket. The job of the event manager is to distribute events to the processing engines. Buffer Manager is responsible for dynamic memory management while event timer is responsible for generating timer events. EMA implements the adapter logic and makes it possible to connect different processing engines with the EM and Buffer Manager.

ES implements various AXI4 (master/slave) interface via which the communication between the PEs and ES is possible. As shown in Figure 3, there is an AXI interconnect between the PEs and ES which is used to write and read the transaction to and from ES.

2.2.1 Event Manager

Event Manager [7] block is responsible for distributing events to PEs. It performs dynamic load balancing and manages complex queue types. It has classifier, queue manager and scheduler as its primary building sub-blocks. Classifier receives the events from PEs via its local receive queues and pushes them to queue manager. Queue manager looks after queues. Queue manager pushes the event to the designated queues based on the event parameters such as scheduler group and priority values. Scheduler distributes events to local RX queues in PE EMAs. Figure 4 below shows the functional units in event manager.

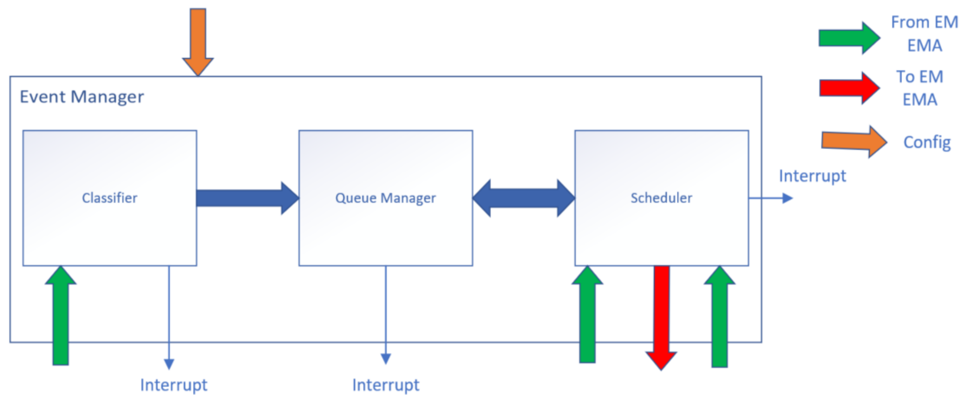


Figure 4: Event Manager block diagram

Event Manager has its own dedicated queues for RX and TX entries in the EMA. Event Manager receives the entries from EMA RX queue and send the TX entries to the EMA TX queue. The data structure model for the RX and TX entries is shown in Figure 5. Only the fields of interest that will be used in DTSS are shown in the figure. Other fields from the data structure is hidden intentionally because of the confidentiality issue.

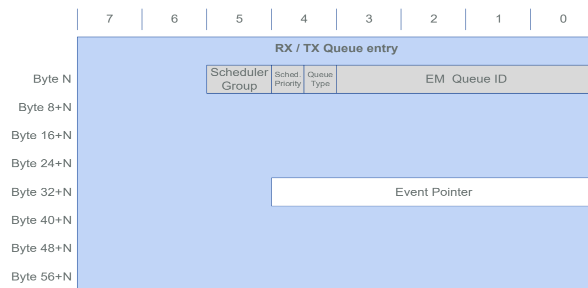


Figure 5: RX/TX Queue Entry [2]

2.2.2 Event Manager EMA

EM EMA [8] is hardware block that connects PEs with the event manager. The primary job of EM EMA is to receive the entries from PEs and queue them up for EM. These entries are then forwarded to dedicated queues in the EM. EM EMA is also responsible for writing the entries back to PEs as output after the EM finishes its operation on the entries. Figure 6 shows the block diagram of EM EMA.

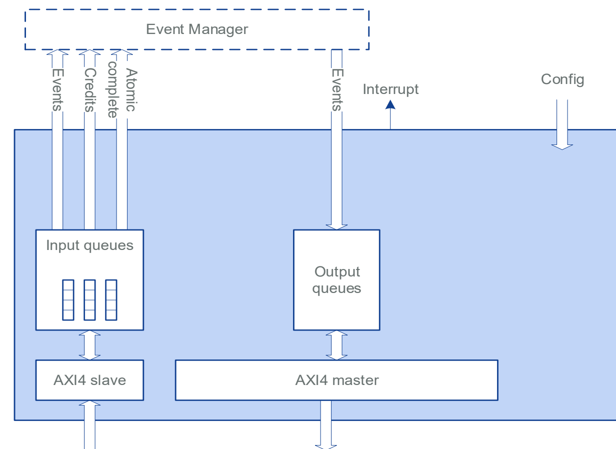


Figure 6: EM EMA block diagram [8]

2.2.3 Buffer Manager

Buffer Manager [9] provides accelerations for memory allocations. The primary job of a buffer manager is to enable the fast allocation of memory such that the required memory is available in time. It provides addresses for fixed size blocks of memory, buffers from memory pool. A memory pool is software configurable entity and refers simply to an unordered list of buffers. The software defines the size of a memory pool, the number of buffers to be accommodated in a pool. The software also defines the size of a buffer; however the buffer size is fixed for a pool.

The primary building blocks of a buffer manager are allocation control, deallocation control, table control, pool table and reference count control. The allocation control block is responsible for handling the credit entries. Credit entries are sent to request a buffer from the buffer manager. It also handles the allocation entries. For each allocation entry, this block also updates the reference count in the reference count control block.

Reference count value infers whether or not a particular buffer in a pool is allocated. If allocated, how many masters is it allocated to. Deallocation control block handles

the command entries. Command entries are sent to free the buffer or to update the reference count value if a PE wants to use an already allocated buffer. Deallocation control updates the reference count value in the reference count control block based on the command entry it received.

The job of the table control is to read the next free buffer id from the pool table and return it to allocation control. Pool table has all the pools configured by SW. The buffer id returned by the pool table is random as pool is an unordered list of buffers. The reference count control block is responsible for updating the reference count value. Figure 7 shows the block diagram of a buffer manager.

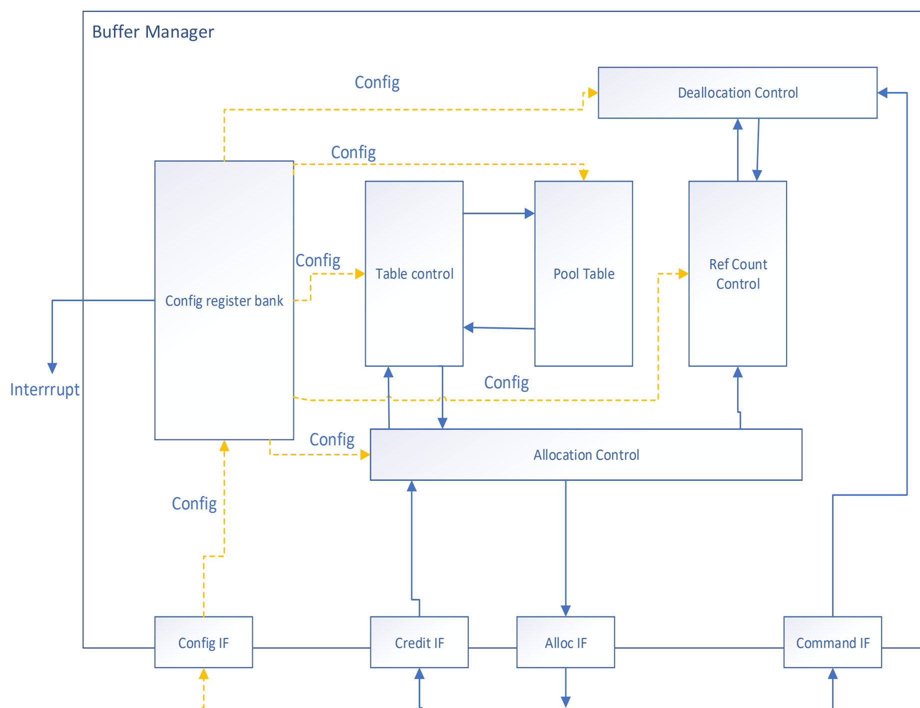


Figure 7: Buffer Manager block diagram

As shown in Figure 7, there is a dedicated interface for credit, command and allocation entries. PEs request buffers writing a credit via credit interface and gets an allocation back via allocation interface. Command entries to update the reference count is sent

via command interface. There are dedicated data structure for credit, allocation and command entries which are not shown because of confidentiality issue.

2.2.4 Buffer Manager EMA

BM EMA [10] is used as an adaptation layer between the PEs and BM such that it allows the communication between PEs and BM. BM EMA receives the entries from the PEs and queue the entries to the dedicated queues. Credit queue and command queue are two input queues which store credit entries and command entries respectively. These entries are then sent to BM for processing. Allocation queue is the output queue. BM EMA receives allocation entries from BM and store them in allocation queue. These entries are then sent to PEs. Figure 8 shows the block diagram of the BM EMA block.

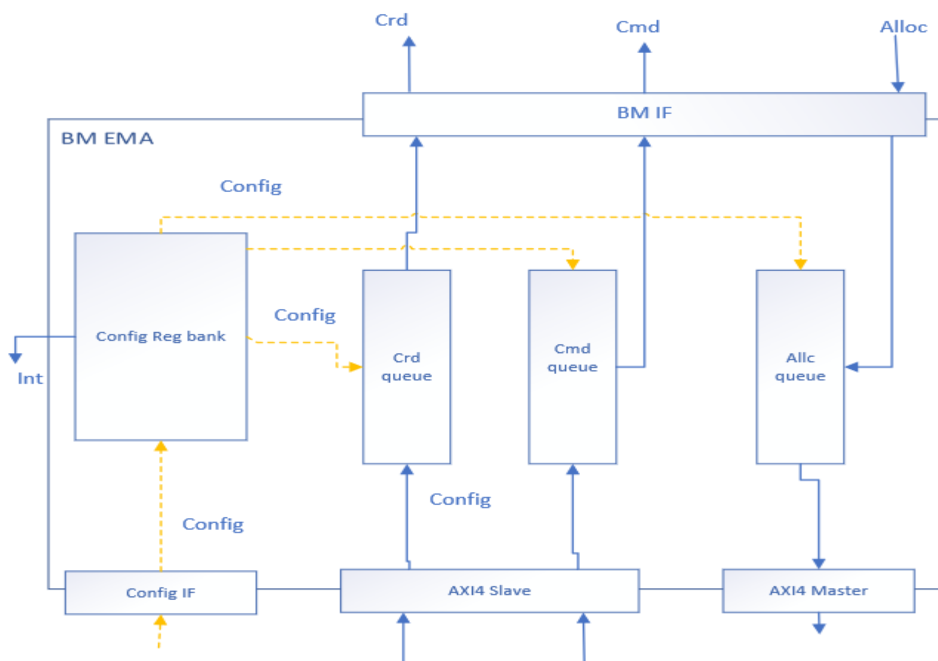


Figure 8: BM EMA block diagram

2.3 Debug and Trace Sub-System in Event Socket

ES Debug and Trace architecture [11] is responsible for generating trace packets. The generated trace packets are based on the configuration setup. These configuration parameters are written by the PEs (software) to the dedicated registers prior to the process. The event socket DTSS provides Debug APB interface for software access. DTSS is built around the fundamental components such as bus monitor, STM-500, ETF and CTI. STM, ETF and CTI are standard ARM CoreSight components while the bus monitor unit is the custom hardware implemented in RTL. Figure 9 shows the block diagram of DTSS in event socket.

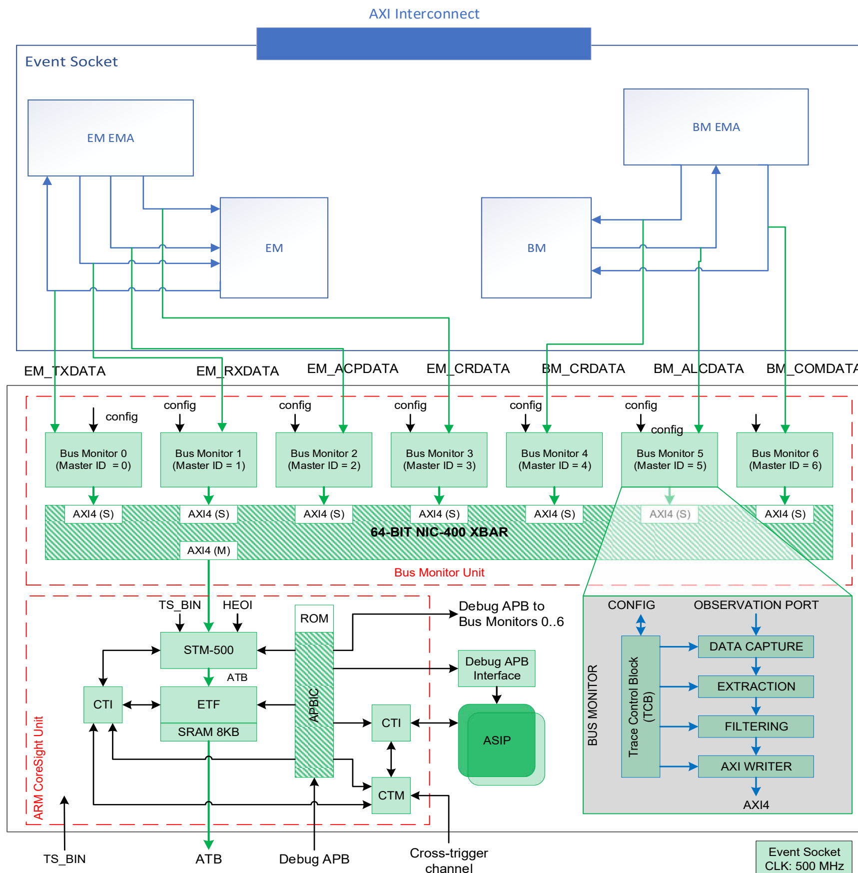


Figure 9: DTSS block diagram [11]

2.3.1 Bus Monitor

Bus monitor [11] is a custom HW that is built around the primary blocks data capture unit, extraction unit, filtering unit, AXI writer and trace control block. Trace control block is responsible for all the configurations related to bus monitor. Figure 10 shows the block diagram of a bus monitor. The bus monitor module does not intrude the bus transaction, meaning it does not change or modify the transaction in any way. It only monitors the traffic in the interface in question. The bus monitor module has primarily three configuration registers; trace control register, filter value register and filter mask register. The purpose of each of these registers is explained in the implementation chapter in section *Register Bank Generation*. The orange arrows in Figure 10 show the configuration flow in the bus monitor while the blue arrows show the data flow.

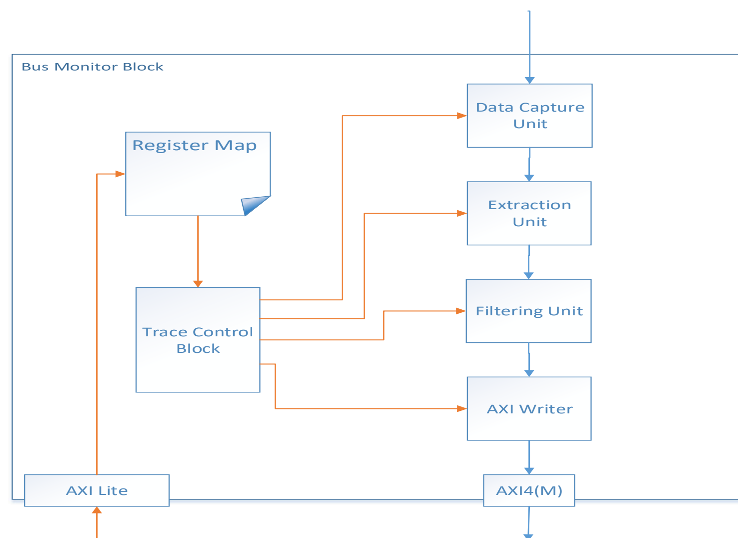


Figure 10: Bus Monitor block diagram

The valid-ready interfaces to be monitored in the scope of this work are shown as green arrows connecting ES to ES DTSS in Figure 9. There are seven interfaces that the bus monitor will be used across to monitor the traffic; *EM_RXDATA*, *EM_TXDATA*, *EM_CRDATA*, *EM_APCDATA*, *BM_CRDATA*, *BM_ALCDATA* and *BM_COMDATA*. These are the interfaces between RX queue in EM EMA and EM, EM and TX queue in EM EMA, Credit queue in EM EMA and EM, Atomic processing complete queue in EM EMA

to EM, Credit queue in BM EMA and BM, BM and allocation queue in BM EMA and command queue in BM EMA and BM respectively.

2.3.1.1 Data Capture Unit

Data Capture unit [11] in the bus monitor is responsible for sampling the data in the interface it is connected to. It has an observation port as an interface towards the monitored interface. The data is captured by the data capture unit when a valid transaction is observed in the interface. The implemented interface in data capture unit is a simple ready-valid handshake interface. The implementation of bus monitor and all other functional units used in bus monitor is described in the implementation section.

2.3.1.2 Extraction Unit

Extraction unit [11] in bus monitor is responsible for extracting the field of interest from the data signal. The field of interest is a configurable parameter which is written to the trace control register by SW. The data fields of interest are listed in the table below. All the data fields can be included or dropped off in the trace packet to be generated based on the configuration written to trace control register.

Table 1: Trace data fields [11]

Trace Block	Trace Entry	Data fields from Entry
Event Manager (EM)	RX Queue entry	• 32-bit EM queue ID • 40-bit event pointer • 8-bit Scheduler Group • 4-bit Scheduler Priority • 4-bit queue type • Time stamp (a relative timestamp is added by STM)
	TX Queue entry	• 32-bit EM queue ID • 40-bit event pointer

		• 16-bit Local Queue ID • Time stamp (a relative timestamp is added by STM)
	Credit entry	• 8-bit number of credits • 16-bit Local Queue ID • Time stamp (a relative timestamp is added by STM)
	Atomic processing complete entry	• 32-bit EM Queue ID • 16-bit Atomic Group ID • Time stamp (a relative timestamp is added by STM)
Buffer Manager (BM)	Allocation Queue entry	• 16-bit Pool ID • 16-bit Buffer ID • 16-bit Local Queue ID • Timestamp (a relative timestamp is added by STM)
	Command Queue entry	• 16-bit Pool ID • 16-bit Buffer ID • 8-bit reference count delta • Timestamp (a relative timestamp is added by STM)
	Credit entry	• 8-bit number of credits • 16-bit Pool ID • 16-bit Local Queue ID • Timestamp (a relative timestamp is added by STM)

2.3.1.3 Filtering Unit

DTSS allows input filtering based on the configuration written to trace control register. The motivation behind having a filtering unit [11] in bus monitor is such that it helps to manage trace data bandwidth. With filtering enabled, trace packet is generated only from a subset of the input entries. The filter value and filter mask value are used to carry out the filtering which is described in section *"Filtering Unit Implementation"*.

2.3.1.4 AXI Writer Unit

As shown in Figure 9, the bus monitor writes to STM-500, standard ARM CoreSight component, via AXI interconnect. The captured data in the bus monitor needs to be translated to AXI transaction such that the input to STM complies with the standard AXI interface. The AXI writer unit is responsible for translating captured data in bus monitor into AXI transaction. The address and data mapping and how it is done is described in the implementation chapter in section *AXI Writer Implementation*.

2.3.1.5 Trace Control Block

Trace control block [11] in bus monitor is an adaptation layer between bus monitor register bank and other functional units. It reads the configuration parameter written by a SW in the configuration register and provides the configuration data to the designated block. The trace control block will have simple interface with dedicated signals towards the other functional unit block and AXI Lite [18] interface towards the register bank. The bus monitor does not capture the data without enabling the data capture unit nor does it filter. These parameters should be first configured by SW by writing an appropriate value to the configuration registers.

2.3.2 ARM CoreSight

ARM CoreSight [12] architecture allows real-time debug and collection of trace information in complex heterogenous multi core environment. It provides a system wide solution for debug and trace meaning that the scope is beyond the cores for example buses. The CoreSight technology offers the system designers flexibility in implementing the debug and trace logic into their design.

There are standard CoreSight components that can be used to implement the debug and trace logic as per designer’s requirement. These components are not a fixed component rather they offer flexibility in configuration and one can configure them to

match their own's need. However, it should be taken into consideration that these CoreSight components support dedicated interfaces for transaction and the system designer should implement the corresponding compatible interfaces to communicate with these CoreSight components to generate the trace packets.

There are primarily five different categories that CoreSight SoCs are classified into. Control and Access components, Sources, Links, Sinks and Timestamp. Control and Access components provide access to other debug components and control of debug behavior. Debug Access Port (DAP) and Embedded Cross Trigger are the examples. Sources are components like System Trace macrocell, Embedded Trace macrocell and Program Trace macrocell that generate the trace data for output.

Links provide connection, triggering and flow of trace data. Replicator and funnel are the examples of links. Sinks are the end points for trace data. Examples include Trace Port Interface Unit (TPIU), Embedded Trace Buffer (ETB) and Serial Wire Output (SWO). Timestamp components generate and transport timestamp across the SoC. Timestamp generator, Timestamp encoder and Timestamp decoder are the examples. The CoreSight sub-system to be used in ES DTSS is shown in Figure 11 and briefly described in the following sections.

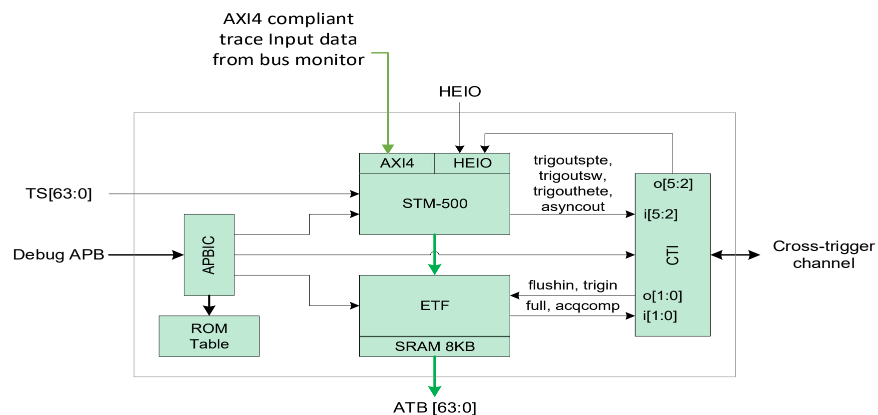


Figure 11: CoreSight sub-system block diagram [11]

2.3.2.1 System Trace Macrocell

System Trace Macrocell (STM) [13] is a standard ARM CoreSight component that allows tracing of system activity from the sources like instrumented software and hardware events. The STM captures the activity it observes in its input interface and generates the trace packet based on the configuration written to it. The version of STM to be used in ES DTSS (STM-500) generates trace stream that complies with MIPI [17] system trace protocol version 2 (STPv2). MIPI stands for Mobile Industry Processor Interface and they develop interface specifications for mobile ecosystem.

Figure 12 below shows the STM as a black box with inputs, configuration and output interfaces.

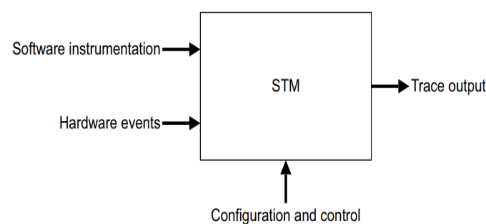


Figure 12: STM Inputs and Outputs [13]

Programming STM requires consideration of two main parts:

Configuration registers:

There are various configuration registers that are accessible both by the software running on the chip and by an external debugger. These registers are used to configure the STM such that it generates the trace packet as per the requirement. These registers occupy 4KB block.

Extended stimulus port registers:

These registers are accessible by the software running on the chip but may not be accessible by external debugger. There are up to 65536 stimulus ports available. Each extended stimulus port occupies 256 consecutive bytes in the memory map and each

of these ports provide multiple locations to allow the software to configure it to generate the required trace stream.

The STM allows multiple software masters to write software instrumentation independently. Each master can use multiple stimulus ports. STM has an ability to timestamp the generated trace packet based on the request and the configuration written to its configuration register.

2.3.2.2 Embedded Trace FIFO

TMC [14] is a standard ARM CoreSight component that enables the capturing of trace information using a debug interface such as 2 pin serial wire debug. TMC has different configuration options; Embedded trace buffer, embedded trace FIFO and embedded trace router. TMC is configured as ETF [14] for the DTSS in ES. ETF enables trace to be stored in a dedicated SRAM. It can either be used as a circular buffer or as a FIFO. Figure 13 shows the ETF configuration.

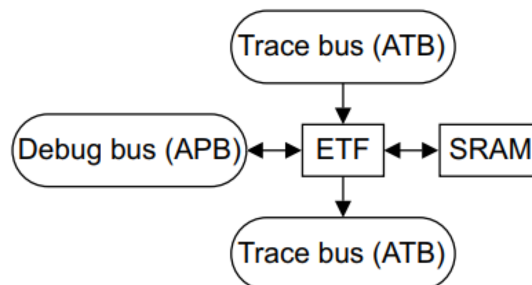


Figure 13: ETF Configuration [14]

TMC can be configured to capture the trace information in one of the three modes: circular buffer mode, hardware FIFO mode and software FIFO mode. As mentioned previously, ETF supports both circular buffer and FIFO implementation. In circular buffer mode, the trace information in the storage overwritten once the buffer is full.

In FIFO mode, the ETF uses its storage as a FIFO and acts as a link between a trace source and a trace sink. No trace is lost or overwritten in this mode.

2.3.2.3Trigger Components

CoreSight SoC has CTI and CTM as trigger components to control the logging of debug information [12]. CTI and CTM form Embedded Cross Trigger (ECT) subsystem that connects many Cross-Trigger Interfaces (CTIs) and Cross Trigger Matrix (CTM) [15] and thereby enable CoreSight sources to interact with each other. The primary job of ECT is to pass debug events from one processor to another for example debug state information so that program execution in processors can be stopped simultaneously if deemed required.

Trigger request is made when a processor wants to send a debug event to another. CTI combines the trigger requests and broadcasts them to all other interfaces in the ECT sub-system as channel events. CTI on receiving a trigger request, maps into a trigger output. This enables the ETM subsystems to interact with each other. CTM controls the distribution of trigger requests. It enables the connection between CTIs and other CTMs where required.

2.3.2.4 Timestamp Components

Timestamp components [12] are used to generate and distribute timestamp values to multiple destinations in a SoC. The scope of narrow timestamp is only within the CoreSight architecture while the wide timestamp is processor generic timestamp and its scope is system wide. Timestamp components in CoreSight architecture is used to distribute narrow timestamp. System wide timestamp distribution is not possible with CoreSight timestamp components.

Timestamp components are timestamp generator, timestamp encoder and timestamp decoder. Narrow timestamp replicator, narrow timestamp synchronous bridge,

narrow timestamp asynchronous bridge and timestamp interpolator are also timestamp components but they are not in the scope of ES DTSS.

Timestamp generator generates a timestamp value that provides a uniform view of time for various blocks in a SoC. It can generate either CoreSight timestamps or processor generic timestamp. Timestamp encoder encodes the 64-bit timestamp value into 7-bit encoded timestamp value. This encoded value is called narrow timestamp. It also encodes and sends the timestamp value to a 2-bit synchronization channel. The timestamp decoder decodes the encoded timestamp value, the data on the narrow timestamp interface and synchronization interface and converts it back to 64-bit value.

2.3.3 NIC-400 Cross-Bar Interconnect

Cross-bar interconnect [16] is used to connect the multiple bus monitors with the ARM CoreSight system. The cross-bar interconnect has 7 AXI4 slave input interface and one AXI4 master interface. The trace input data is written using the AXI master interface in each bus monitor to the AXI slave interface in the interconnect. The cross-bar interconnect then maps the transaction to the AXI master interface output on the interconnect. The STM macrocell is connected to the AXI master interface on the interconnect.

The cross-bar interconnect has 7 slave inputs which is connected to a single master interface. Each transaction coming into any of the slave input is forwarded to the master interface. The cross-bar interconnect is a NIC-400 component. It is a configurable, high performance, optimized AMBA compliant interconnect. It allows to configure the number of slave interface ranging from 1 to 128 and master interface ranging from 1 to 64. The interface protocol supported are AXI3, AXI4 and AHB-lite.

2.3.4 Hardware Interfaces

Hardware interfaces in a design are the medium through which the hardware modules communicate with each other. The hardware interfaces used in ES DTSS is briefly described in the following sections.

2.3.4.1 AXI Interface

AXI is the most common interface used in SoC design [18]. AXI interface implements five different channels; write address channel, write data channel, write response channel, read address channel and read data channel. Read response signals are integrated in read data channel. AXI has different variants AXI3, AXI4, AXI lite etc. AXI4 is used for data transaction while AXI lite is used for configuration setup in this work. The bus monitors in DTSS in ES implements AXI4 master interface that writes to AXI4 slave interface on the AXI interconnect. The interconnect then maps the transaction and forward to the right AXI master interface on the interconnect. On the other side of the interconnect is CoreSight architecture component.

Each channel in the AXI has their own ready-valid pair which should do a handshake prior to the actual communication. The valid ready handshake mechanism in AXI is similar to the valid ready handshake mechanism on the interfaces between EM and EM EMA and BM and BM EMA.

AXI supports burst-based transaction to achieve a better throughput. There are three different variants of burst based transaction in AXI. Fixed burst is used if the transaction is intended to the same address location. In case of memory access, incremental burst is used. The address offset increases based on the size of data access. Wrap burst is not so common and little bit tricky to get around the head. The increment burst and wrap burst is out of the scope of this work. Figure 14 and Figure 15 show the fixed burst based write and read transaction respectively.

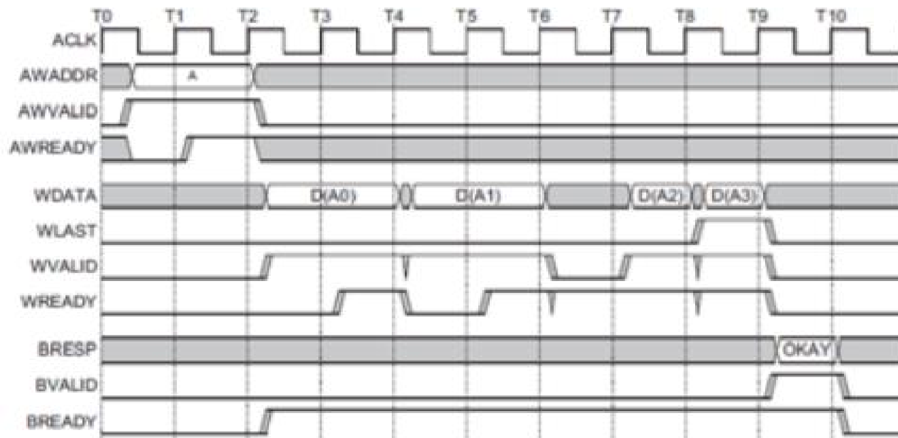


Figure 14: Burst Write in AXI [18]

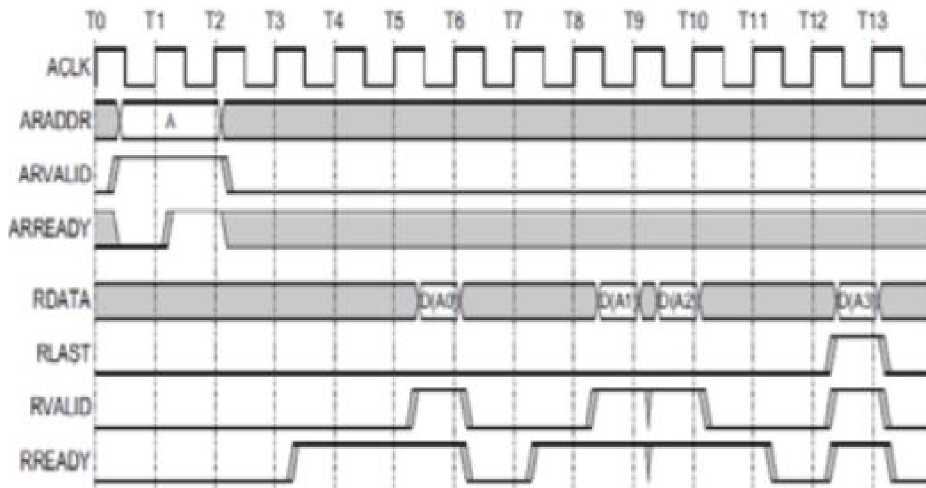


Figure 15: Burst Read AXI [18]

As shown in Figure 14 and Figure 15, the burst transaction follows a handshake process. The communication starts with the handshake process in the address channel. The master drives the destination address via AxADDR signal and asserts the AxVALID signal high. The slave when ready asserts the AxREADY signal high. xDATA is driven to the data signal and xVALID is asserted high for all the valid beats in the burst. The slave receives the data asserting the ready signal high. xLAST signal is asserted high to indicate the last beat of the burst. Response channel communication also starts

with valid ready handshake. The response is driven to the xRESP signal to indicate the status of the transfer. [18]

2.3.4.2 Valid-ready handshake Interface

Valid-ready handshake interface implements three signals; valid, ready and data signal. The handshake occurs prior to actual transfer. The master first asserts the valid when it has the valid data to drive to the interface. The slave asserts the ready signal indicating that it is ready to accept the transfer and then the master drives the data to the data signal. Figure 16 shows the handshake process in valid-ready interface. In Figure 9, all the seven interfaces shown as green arrows are the simple valid ready handshake interface.

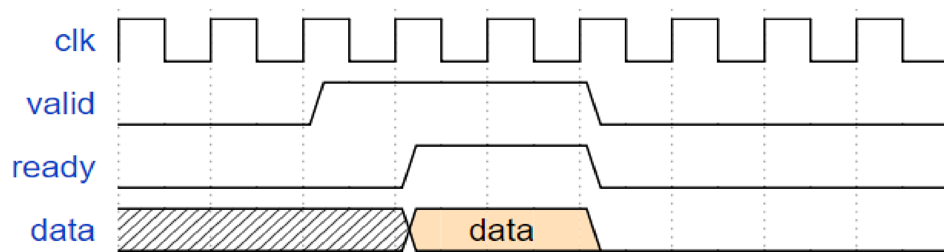


Figure 16:Valid-ready interface timing diagram

2.3.4.3 Hardware Event Observation Interface

HEOI [13] is an interface on STM that enables the monitoring and tracing of hardware events. HEOI provides interface for 32 rising edge hardware events. The input to HEOI could be interrupts, cross-triggers or other signals of interest in the system. When a hardware event is asserted and HEOI signal connected to that particular hardware event is enabled, the event is captured and a trace packet is generated.

2.3.5 Software Interfaces

Software interfaces in a system allow the software access. Typical example of software access in a system is for configuration registers. Each hardware module having configuration registers is generally meant to be configured by the software before startup. This is typically achieved through software interface. The software interfaces used in ES DTSS are following.

2.3.5.1 Debug Advanced Peripheral Bus (APB) Interface

APB [19] comes under AMBA3 protocol family and is optimized for minimal power consumption. It is a low bandwidth protocol and is typically used in the scenarios where timing is not critical for example writing the configuration. Debug APB is used for debugging purpose. In ES DTSS, the software uses the debug APB interface to write the configuration. However, the configuration registers in the bus monitors in DTSS does not support APB access. The register file for the bus monitors is generated using Nokia in-house tool called reg-gen which only supports Node protocol or AXI lite protocol. It is thus a bridge between the Debug APB and AXI lite (or Node) is necessary when the bus monitor is instantiated in DTSS level. Figure 17 shows the write and read transfer in APB bus. The ***PWRITE*** signal when asserted high indicates a write transfer and when asserted low indicates a read transfer.

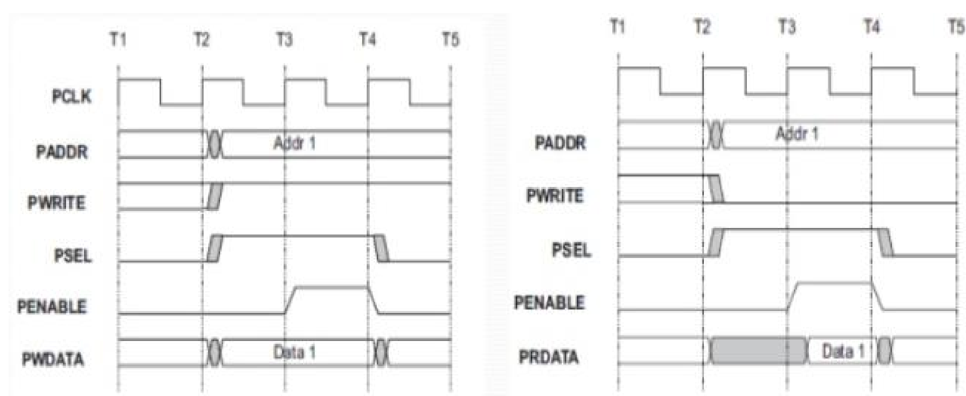


Figure 17: APB Write and Read [19]

2.3.5.2 Advanced Trace Bus Interface

ATB [20] interface in ES DTSS implements a 64-bit wide trace bus through which the trace packet generated by macrocell is written as output. ATB also has hand shaking signals. Valid signal is asserted when a valid trace data is present and the slave asserts the ready signal to indicate that it is ready to accept the trace data. ATB also provides signals to carry out the flushing of a FIFO in CoreSight. Like trace data transaction, flushing also starts with a valid ready handshake. When a flushing request is made via ATB, the macrocell must drain the FIFO.

2.4 Verification Environment

The term verification in this scope refers to the front-end functional verification. It is advised to comprehend accordingly wherever the term is used. The main objective of verification is to make sure that the design works as per the requirements listed in the specification. It is critical in SoC design as it helps in tracking down the bug in time. An SoC coming out of a foundry with a bug is not only a loss of the effort that was put in to develop the chip but also a loss of substantial amount of money spent in the engineering. It is therefore verification in SoC design is of utmost important. In fact, it is widely accepted fact in the silicon industry that verification of a design takes much more effort than the actual implementation of a design itself. It is a known fact that it can never be assured that a design is 100% bug free. However, with the verification, the goal is to ensure that a design is as much bug free as possible.

The state of the art SoC design ecosystem is getting more and more complex as our technology is moving into the era of automation; from automated vehicles to smart homes and to smart cities. This very fact ultimately ushers an absolute realm of Internet of Things (IoT) where everything around us will be connected to each other through a network. This means that there will be billions of devices in the network and

one malicious device could cost quite a bit for the entire network. On the one hand there is increasing complexity in design ecosystem and on the other hand, the state-of-the-art verification technology is lagging in addressing all the verification needs. On top of that short time-to-market requirement for industries exacerbate the situation. The widely accepted industrial standard for verification is Universal Verification Methodology popularly known as UVM [21]. It is based on System Verilog language. UVM provides a framework to achieve coverage driven verification (CDV). CDV includes automatic test generation, self-checking testbenches and coverage metrics such that the time spent verifying a design is significantly reduced. In addition, UVM allows to add randomization and constraints in the test that helps in locating not only anticipated but also unanticipated bugs in a design.

UVM is a layered modular testbench architecture that exploits the re-usable verification components. Typically, an UVM testbench consists of an environment, agent, driver, monitor, sequencer, sequence and sequence item. A test object then encapsulates these components to make a complete testbench. An interface is then required for the testbench to communicate with the design under verification. Figure 18 shows a typical UVM based testbench architecture.

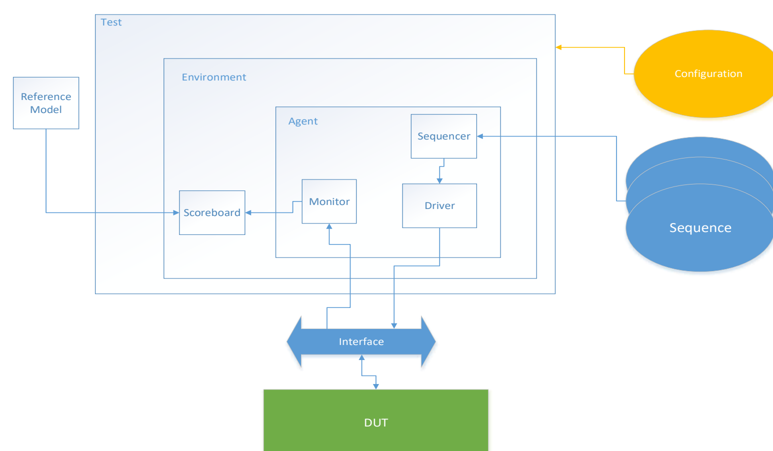


Figure 18: UVM Testbench

2.4.1 Sequence Item

Sequence item [21] is the actual transaction item that is sent as stimuli to the DUT. UVM provides a base class called uvm_sequence_item which is inherited while creating user defined sequence item. Sequence item is an abstraction that abstracts away the signal level information to form a transaction object.

2.4.2 Sequence

Sequence [21] starts the sequence item. A user defined sequence is created extending uvm_sequence base class provided by UVM. It is the uvm_sequence that is responsible in generating the transaction item.

2.4.3 Sequencer

Sequencer [21] is responsible for forwarding the sequence item to the driver. The sequencer also serves as the arbiter to control the flow of transaction item from various sequences. UVM provides the uvm_sequencer base class to create a user defined sequencer.

2.4.4 Driver

Driver [21] is the component responsible for driving the test stimuli to the DUT via an interface. The driver maps the transaction objects into signal level activities based on the interface protocol. UVM has uvm_driver base class which is extended to create a user defined driver.

2.4.5 Monitor

Monitor [21] is a passive uvm component that is responsible for monitoring the transaction in an interface. It does not whatsoever invade the transaction in the interface but only monitors. The protocol specific transaction is extracted by the

monitor and then it maps it to a corresponding transaction item. UVM provides uvm_monitor base class which is extended to create a user defined monitor.

2.4.6 Scoreboard

Scoreboard [21] is responsible for evaluating the input and output transaction. The scoreboard checks whether the result is correct or not. The mechanism of how a scoreboard compares the input and output is user specific. The scoreboard is typically fed in the reference input and output through some externals sources and compare them with the actual input and output in the interface or it could read the input output from a reference model directly exploiting the DPI provided by UVM. Scoreboard is created by extending the uvm_scoreboard base class.

2.4.7 Interface

Interface [21] is created using a keyword *Interface.* Interface enables the communication between the UVM testbench and the DUT. A virtual instance of interface is created in the UVM testbench to allow the communication. The virtual interface instance is required because a DUT is a static module and the UVM testbench is dynamic and transient. UVM provides the concept of virtual interface that enables the communication between a dynamic testbench and a static module.

In addition to these UVM objects and components, UVM also provides a factory mechanism with which a component or an object could be registered to a factory and could be accessed from anywhere in a testbench. The factory mechanism comes handy if an instance or type of a component or an object need to be overridden.

UVM testbench is a phase based testbench architecture. There are defined phases that each component goes through to set up the test environment and execute the test. These phases are executed in order when a test is carried out. Figure 19 shows the phases that a test undergoes through.

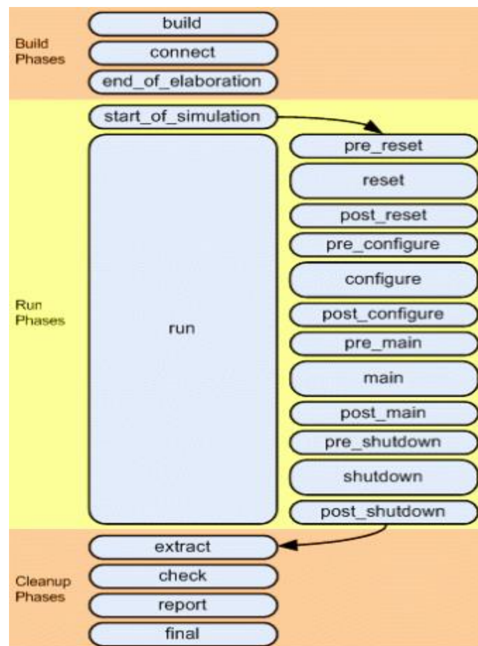


Figure 19: UVM Phases

Primarily there are three phases as shown in the figure above. When the simulation starts, build phase is the first phase to execute. During this phase all the components used in the testbench is constructed, configured and connected. This phase executes in top-down fashion. After the completion of end of elaboration phase in the build phase, run phase is executed. During this phase, stimuli are generated and executed. Run phase is a concurrent process. Finally, the cleanup phase is executed to extract the information from the scoreboard and coverage monitors.

Chapter 3

3. Related Works

The demand for a better performance of a computing engine is indispensable for the evolution of technology. Consequently, deployment of multi-core processing system is ubiquitous. However, the sheer deployment of multi-core processing system does not yield performance boost. There are multitude aspects to take care of; the capability of an algorithm to exploit available resources to the fullest is the fundamental one. As Amdahl's stated in his law, the serial nature of the algorithm ultimately becomes the bottleneck for the cent percent utilization of computing resources. However, a significant boost can be achieved provided that the serial nature of the algorithm is accelerated in SW/HW.

No matter how efficient the algorithm is and how refined the HW to run that algorithm on is, only the correctness of the outcome guarantees the usefulness. Debug and trace infrastructure allows the opportunity to guarantee the correctness of the outcome.
It is also seen that many embedded processors have used scan-chains also for the system level HW/SW debugging. However, it does not scale well with the complexity of cutting-edge SoCs [22]. It is seen that most of the vendors follow either of the two major specifications: ARM CoreSight and IEEE-ISTO 5001 (popularly known as Nexus 5001) [24]. ARM CoreSight is the target implementation of this work and hence Nexus 5001 based implementation is discussed here.

This section is structured such that section 3.1 describes the related implementation of ES functionalities and section 3.2 and section 3.3 describe the related debug and trace infrastructure implementation. Albeit the scope of this work is limited to the

implementation of debug and trace infrastructure, related works carried out in the ES level is presented in section 3.1 to lay the foundation for the better understanding of ES concept.

3.1 Open Event Machine

OpenEM [23] is a lightweight, run-to-completion, event processing environment targeted for multicore SoC for the data plane applications. It is optimized for multicore system and tailored to yield high performance. It is essentially data plane processing concept based on asynchronous queues and event scheduling. OpenEM applications are built from execution objects, events and queues. The execution object is an entity that gets called at each event receive. The scheduler picks an event with the highest priority from a queue and calls the receive function of the execution objects for event processing. Run-to-completion is the phenomenon that restricts the preemption of the events processing. Only after an event returns, the scheduler picks another event.

OpenEM is used as software acceleration in a multi-core processing environment mainly for the event scheduling. The related work in the ODP event processing acceleration can be found in [23]. OpenEM provides services to manage events, event queues and execution objects. All the available cores share the objects but, yet multi-core safety is guaranteed.

OpenEM implements the following fundamental entities for computation:

- Events
- Execution Objects
- Event Queues
- Process
- Dispatcher
- Scheduler

Events carry data to processes. When a process has to communicate with other processes, it translates the information to be sent into an event and send it to the other processes. However, an event can also have no data payload but a pointer to the payload. When an event has the pointer to the payload, the OpenEM also provides services to access these payloads. Each event belongs to its own dedicated event pool. All events in an event pool, in the beginning, are queued in a free queue and waits for the allocation. OpenEM provides services for allocation and freeing of events.

Execution objects embeds the algorithm on how to process the received events. The receive function executes the algorithm. The user implements the receive function and register it with the execution object. Creating and deletion of execution objects services are provided by OpenEM.

Event queues relates the queued events with the execution objects, alternately saying connects the data with the algorithm. Each event queue is tied to one execution object and all the events from that event queue is processed by this execution object. The service for event queue creation and deletion and sending events to the event queues are provided by OpenEM.

Process is a higher abstraction level entity that has its own event pools, event queues and execution objects. When a process is initialized, the event pools are provided and the event queues and execution objects are created at run time. A process in an OpenEM is a unique entity and has its own device ID and a process ID. It maintains its identity because of the following rules:

- No core sharing between processes
- A process has access to the native event pools tied to it but not to the ones in foreign processes.
- A process has access to the native execution objects tied to it but not to the ones in foreign processes.
- Sending of events to foreign event queues of foreign process is allowed.

Dispatcher triggers the scheduling of events. Each process has one dispatcher. When a dispatcher is called, it looks for the event to be scheduled. If no event is available, the dispatcher returns immediately with a negative response. If an event is available, the dispatcher calls the receive function from the execution object. The assurance of no deadlock scenario is achieved by implementing the receive function such that it never waits on a condition dependent on the execution of another receive function.

Scheduler is responsible for scheduling of event queues based on the algorithm implemented. Each process has one scheduler that describes how scheduling of events from event queues is carried out. Scheduler is triggered by the dispatcher running in the same process. Scheduling may be based on priority, atomicity and order. Based on priority, the event queue with a higher priority is scheduled first. However, OpenEM is run-to-completion machine and hence no preemption of events is allowed in scheduling. Atomicity ensures that an event queue is exclusively processed by a single core/process at a time. This very feature excludes the need for semaphores and mutex. Based on the order, the oldest event (that has spent most time waiting) gets scheduled first.

Nokia's OpenEM implementation on Intel's platform can be found in [6]. Benchmarking results presented in [23] by Texas Instruments is depicted in Figure 20. The result is based on three benchmarking parameters: execution time, input data amount and output data amount. The simulation is run on KeyStone Architecure [25] with eight processing cores.

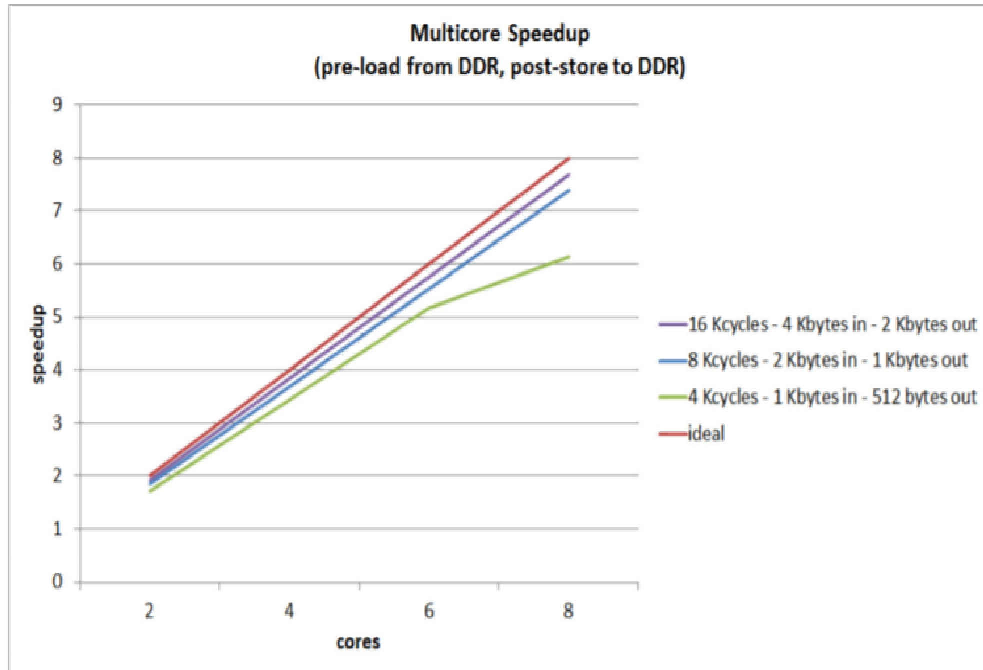


Figure 20: Multicore Speedup with OpenEM [23]

As shown in Figure 20, approximately 96% increase in speedup is achieved. The speedup is clearly affected by the execution time, input data amount and output data amount. The assumption for this simulation is that there are no memory access stalls in the sequential execution while in case of parallel execution pre-load and post-store is considered while realizing OpenEM overhead, NUMA overhead and memory stalls.

An OpenEM process is able to operate in heterogeneous environment. OpenEM process can run on bare metal and also on top of an OS. The motivation behind having it run on an OS is access to more services such as memory management, file systems, device drivers etc. Having it run on bare metal suffers it from the services like memory management while having it run on an OS incurs the overhead due to OS. It is thus the hardware acceleration of ODP event processing and memory management with ES pose the better performance. ES HW not only addresses event processing aspect but also the memory management. The target of the ES HW is to allow ODP APIs and OpenEM APIs to run on top of it.

3.2 Nexus 5001

Debug initiative based on IEEE ISTO 5001 [24] debug specification is the Nexus 5001. It allows the embedded processor vendors to have a unifying standard to implement in their debug and trace infrastructure as such it provides consistent set of auxiliary pins as the access interface. In addition, it provides message-based transfer protocols and standard development features to facilitate debug implementations. Figure 21 shows a typical debug and trace blocks in Nexus based infrastructure.

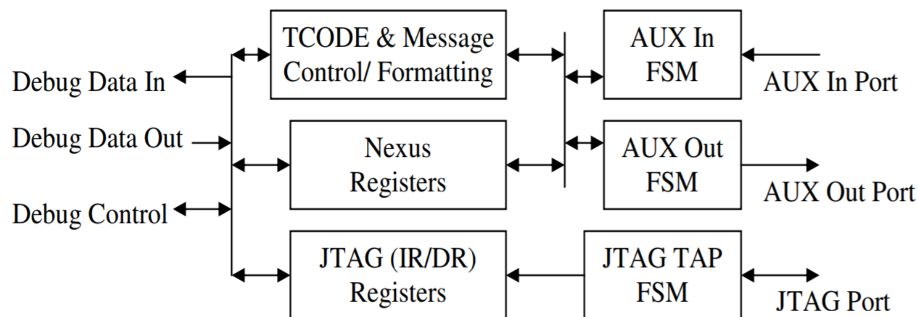


Figure 21: Debug and Trace blocks in Nexus based infrastructure [24]

The Nexus standard defines the nexus-based debug support interface. The Nexus IO signal is defined by leveraging IEEE 1149.1 standard [26] widely accepted as a test and debug pin interface. The standard defines the signal IO as an extensible auxiliary port (AUX) that can either be used as JTAG port or as a stand-alone development port. The primary purpose of AUX out port is to provide higher trace throughput. JTAG port on the interface can also be used in Nexus-specific ways; for an instance, to embed the nexus trace output into JTAG messages. [24]

The message format is also defined by the standard. The message consist of a 6-bit transfer code and each value of transfer code corresponds to a different number of packets as defined in the standard. The standard also defines several dedicated registers which facilitate the integration of debug support to different cores. For an instance, different Device Identification (DID) register for each core to identify the control and debug operations associated with it. [24]

The standard defines four implementation classes such that the designs can select the important features as per their need. Class 1 provides features similar to standard JTAG implementations. Complex debugging features with real-time monitoring is provided by class 2. Class 3 provides data tracing services and includes the ability to read and write memory and I/O during run-time and class 4 allows the features like remapping the memory and I/O ports. [24]

The related work carried out based on Nexus standard is in [27]. R. Stence presents real time calibration and debug techniques of embedded processors with the Nexus 5001 interface in this work.

3.3 Debug Support Architecture

Debug support architecture is defined in [28] where the proposition is the modular breakdown of the architecture such that the architecture comprises of three complementing parts: an extended JTAG module as the interface between the SoC and debug host computer, modules that connect debug interface to processor and the processor specific on-chip debug support modules. Figure 22 shows the system level integration of debug support architecture. JTAG module is the first complementing part from the modular breakdown, IO client is the second part and OCDS module is the third part. Processor 0 is connected to a specific IO client as it has no FPI bus connection while processor 1 and processor 2 has FPI bus connection and thus connected to the JTAG module via IO client 1.

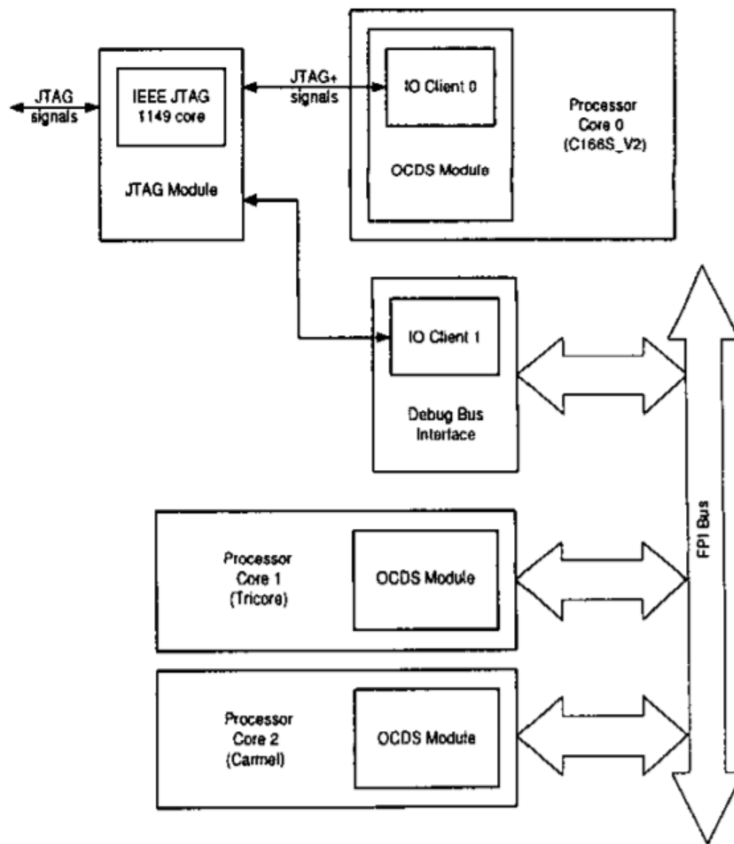


Figure 22: On-chip debug support architecture [28]

Chapter 4

4. Implementation

This chapter describes the design implementation and testbench setup for verification. The implementation phase of this thesis work can be split into Design Implementation, Verification Implementation and Build Environment setup. The verification of the top-level module (DTSS) is carried out in the Event Socket top level. It is thus outside the scope of this work.

4.1 Design Implementation

Design implementation involves register bank generation, bus monitor sub blocks implementation and sub blocks instantiations on the bus monitor top-level.

4.1.1 Register Bank Generation

The RTL implementation of the bus monitor started off with the generation of the register bank. Nokia has in-house EDA tool for register generation. The bus monitor has three primary configuration registers: Trace Control Register, Filter Value Register and Filter Mask Register. Filter value register and filter mask register are used only when filtering is enabled in the bus monitor. Filter value register simply holds the value to be filtered and filter mask register holds the value such that all the fields of interest do not get masked. For example, if the filter mask value is FF, only the least significant byte of the input data is not masked and if this value is equal to the value in the filter value register, the filtered data is generated.

The trace control register is depicted in Table 2.

Table 2: Trace Control Register [11]

Byte	3								2								1								0							
Bits	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Name	Stimulus Port								Reserved	Reserved	Master ID						Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Select 2	Select 1	Select 0	Timestamp	Filter Enable	Trace Enable

4.1.2 Data Capture Unit Implementation

After the register bank was in place, the data capture unit was implemented. The data capture unit is responsible for capturing the data in the interfaces. The data capture unit has observation port implemented as “monitor” that samples the data on the interface only when both the valid and ready signal are high.

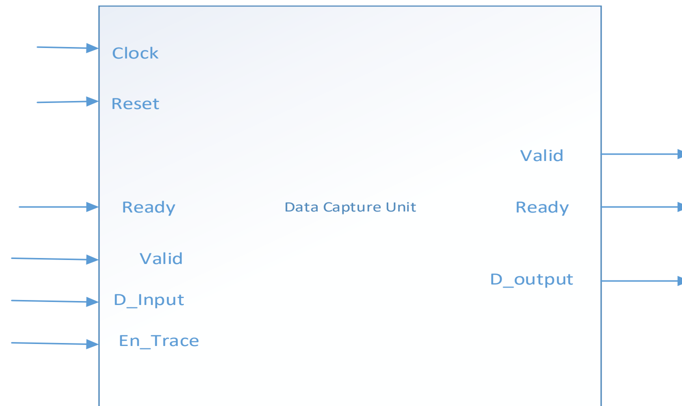


Figure 23: Data capture Unit I/O interface

Figure 23 shows the input and output interface of the data capture unit. The signal *En_Trace* carries the trace enable information from the trace control register via trace control block to the data capture unit. The timing diagram in Figure 24 shows when data is sampled and output is written.

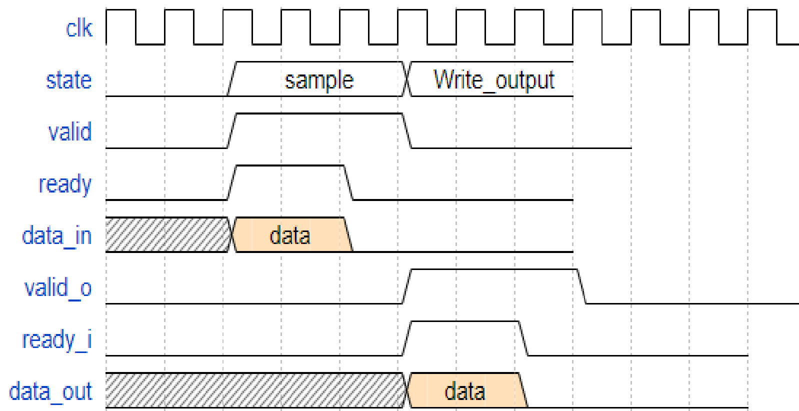


Figure 24: Sampling and data output state in DCU

4.1.3 Extraction Unit Implementation

The extraction unit is responsible for extracting the data field from the data input based on the configuration. The instrumentation software configures the trace control register's "select" fields and the extraction unit performs the extraction task based on this vector. Figure 25 shows the input and output interface implemented in the extraction unit. The signal *sel_field* comes from the trace control register.

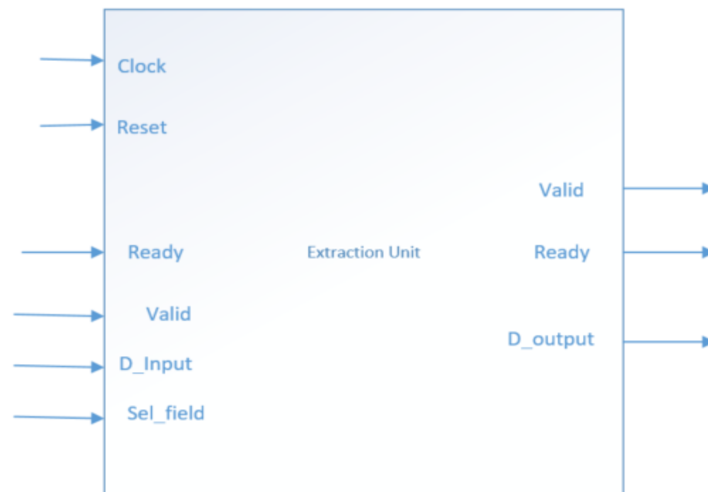


Figure 25: Extraction unit I/O interface

The extraction unit collects the complete data structure and based on the select vector, performs the extraction. Figure 26 shows the timing diagram of the extraction process.

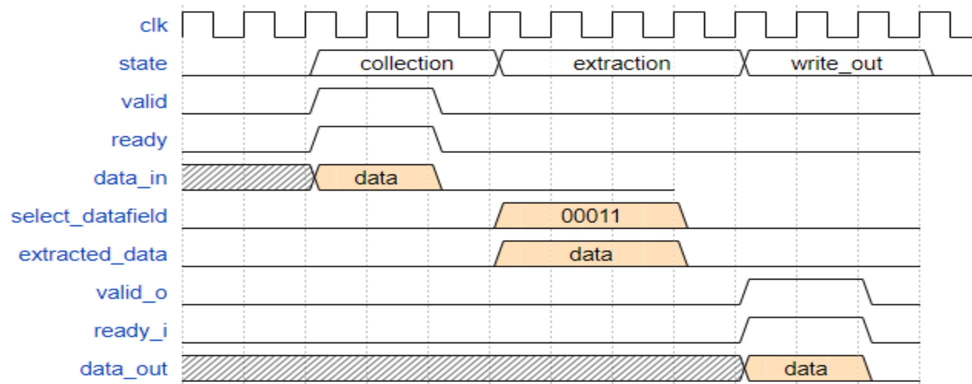


Figure 26: Extraction process timing diagram

4.1.4 Filtering Unit Implementation

Filtering unit performs filtering of the trace input data if filtering is enabled. Filter enable information is read from the configuration register and the decision on whether or not to perform the filtering is made. If filtering is not enabled, the filtering unit simply propagates the input data to the AXI writer unit without filtering. The implemented input and output interface in the filtering unit is depicted in Figure 27.

Figure 27: Filtering Unit I/O interface

The three signals *en_filter, fil_value* and *fil_mask* come from the dedicated configuration registers in the register bank. *En_filter* tells whether filtering is enabled or not. Filter value and filter mask pattern are used for filtering. The mask value determines which bits from the input data fields are compared with the filter value. A filtered trace input data is generated only on a compare match. Figure 28 shows the timing diagram of the filtering process in filtering unit.

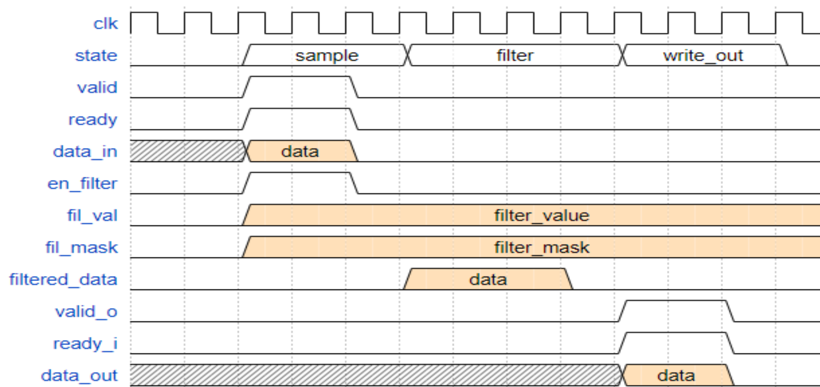


Figure 28: Filtering process timing diagram

4.1.5 AXI Writer Implementation

AXI writer module is responsible for translating the bus monitor transaction into AXI transaction. The transaction in the bus monitor is a simple valid-ready transaction, meaning the transaction object in the bus monitor encapsulates the valid-ready signal pair and a data signal. The job of the AXI writer is to convert the valid-ready transaction object into AXI transaction object. The motivation behind this translation is the CoreSight components used in DTSS. The STM macrocell in the CoreSight architecture is the entry point for the trace input data. It has a slave AXI interface through which the input trace data is written. It is thus the transaction from the bus monitor is translated into AXI protocol compatible transaction by the AXI writer unit. The implemented input and output interface in AXI writer module is shown in Figure 29.

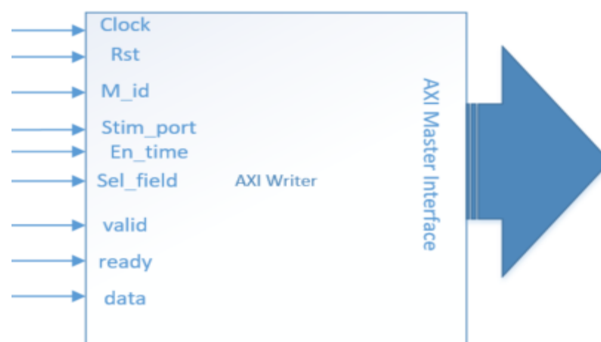


Figure 29: AXI writer I/O interface

The AXI writer module uses write burst if the trace input data cannot be accommodated in a single 64-bit word. For an example, on the interface between Event Manager and RX-queue in EMA, if the *select datafield* vector is “11111”, the trace input data will have 32-bit ODP queue ID, 40-bit Event pointer, 4-bit scheduler group, 8-bit scheduler priority and 4-bit queue type. In this case, the AXI writer has to do the write burst of length 2.

The AXI translation primarily involves address mapping and data mapping. STM has memory-mapped stimulus port where the trace input data is written. The 32-bit address signal is mapped as shown in the table below:

Table 3: Address Mapping[11]

Address bits				
31:30	29:24	23:20	19:8	7:0
0x00	<master ID>	0x0000	<stimulus port>	<packet type>

Table 3 shows that bit 7 downto 0 indicates the packet type. CoreSight defines various packet types. However, only G_D (guaranteed, data-access) and G_DTS (guaranteed, data-access with timestamp) are considered in the scope of DTSS in the ES. The selection on whether G_D or G_DTS is considered is based on the timestamp field in the trace control register. If timestamp enable field in the trace control register is set,

then G_DTS is selected. STM has extended stimulus port where not only trace input data can be written but also can be augmented with metadata. This augmentation of data enables the STM to know which kind of trace to generate. Bit 7 downto 0 in the address signal provides the augmentation for the trace input data. Address offset for G_D is 0x18 and that for G_DTS is 0x10. Bit 19 downto 8 indicate the stimulus port address and bit 29 downto 24 indicate the master that is writing the trace input.

STM uses the write strobe signal to determine the size of the transfer and to locate the valid data on the data bus. Strobe signal has one bit for every byte of data. For 64 bits data, strobe signal has 8 bits. The following table shows how data is aligned on the data bus and the corresponding strobe vector for the data.

Table 4: Data alignment[11]

<table>
<tr><th rowspan="2">Beat</th><th rowspan="2">WRSTRB</th><th colspan="8">Byte</th></tr>
<tr><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>11111111</td><td colspan="4">Event Pointer LSB</td><td colspan="4">ODP Queue ID</td></tr>
<tr><td>2</td><td>00001111</td><td colspan="4"></td><td>QT</td><td>SP</td><td>SG</td><td>EP MSB</td></tr>
</table>

Table 4 shows the data alignment for EM-RX queue interface. Beat refers to the data word transferred in a burst. The AXI writer does burst write of length 2. The first beat has 32-bit ODP queue id and 32 LSB event pointer for which the write strobe vector is *FF.* The second beat has 8 MSB from event pointer, 8-bit scheduler group, 8-bit scheduler priority (padded 4-bit MSB) and 8-bit queue type (padded 4-bit MSB) for which the strobe vector is *F.*

4.1.6 Trace Control Block Implementation

The trace control block is responsible for distributing the configuration and control information to the various sub-modules in the bus monitor. The configuration and control information for the bus monitor is written to the dedicated registers in the register bank by the instrumentation software. The trace control block reads this information from the register bank and distributes them to the dedicated sub-blocks

in the bus monitor. The implemented input and output interface in the trace control block is shown in Figure 30.

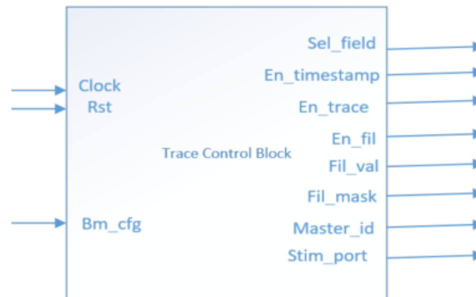


Figure 30: Trace Control Block I/O interface

The trace control block takes the input from the register bank via *Bm_cfg* signal. *Bm_cfg* is an abstracted signal that abstracts the control and configuration information from the register bank.

4.2 Verification Implementation

After the RTL implementation of all the sub blocks in bus monitor, they were instantiated in the bus monitor top-level. Figure 10 shows the structure of the bus monitor after having all the sub-blocks instantiated. Once the design implementation was done, verification implementation was carried out to verify the design. For the scope of this work, bus monitor was verified in the module level. However, in the Event Socket project, all the verification is carried out in the ES top-level. Figure 31 shows the testbench architecture implemented for the verification of the bus monitor.

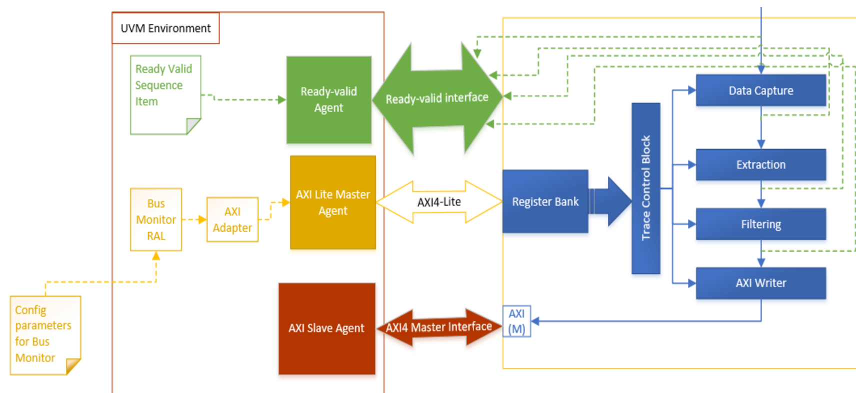


Figure 31: Bus Monitor Testbench Architecture

4.2.1 UVM Environment Implementation

The box with red border and labeled UVM Environment in Figure 31 represents the UVM environment. AXI lite master agent, AXI slave agent and AXI adapter are the verification component instantiated from in-house AXI VIP.

Ready-valid VIP was written from the scratch for this task so that the transactions on the ready-valid interfaces could be verified. The register abstraction layer, RAL was generated using the RAL generator tool called ralgen. The ralgen tool was provided the bus monitor xml file. The bus monitor xml was generated using reg_gen tool during the design implementation phase. Ralgen takes the xml file as input and produces ralf file which abstracts the information of all the registers in the design. This ralf file was used to generate the RAL model.

The arrow with label AXI-Lite in Figure 31 is the interface via which configuration and control registers are written. The green ready-valid interface drives the input data signals to the DUT. The AXI Master interface connected to AXI slave agent drives the AXI transaction towards the testbench. The AXI slave agent in the testbench monitors the received AXI transaction.

All the blocks shown in Figure 31 were instantiated in the UVM environment. Corresponding object creation was done in the build phase. The RAL model, the sequencer and the adapter were connected in the connect phase. The final phase implemented a uvm_report_server to extract the information and print the status of the test. The code snippet below shows the implementation of the UVM environment.

```
//Component declaration
 axi_master_agent                                   u_axi_lite_agt;
  axi_slave_agent                                  u_axi_slave_agt;
  rv_agent                                          u_rv_agt;
  ral_block_default_slv_mmap_bus_monitor           u_bus_mon_ral;
  axi_adapter                                      u_axi_adapter;

 //object creation
 function void bus_mon_top_env::build_phase(uvm_phase phase);
                super.build_phase(phase);

 u_axi_lite_agt=axi_master_agent::type_id::create("u_axi_lite_agt", this);
                u_axi_slave_agt=axi_slave_agent::type_id::create("u_axi_slave_agt",this);
                 u_rv_agt=rv_agent::type_id::create("u_rv_agt",this);
                u_axi_adapter=axi_adapter::type_id::create("u_axi_adapter",this);
                u_axi_adapter.fix_tr_size=1;
                u_axi_adapter.fixed_tr_size=4;

                u_bus_mon_ral=ral_block_default_slv_mmap_bus_monitor::type_id::creat
                e("u_bus_mon_ral");
                 u_bus_mon_ral.configure(null,"");
                u_bus_mon_ral.build();
                 u_bus_mon_ral.lock_model();
endfunction: build_phase

//Connect phase
function void bus_mon_top_env::connect_phase(uvm_phase phase);
                super.connect_phase(phase);
                 u_bus_mon_ral.default_map.set_sequencer(u_axi_lite_agt.sqr,
                u_axi_adapter);
                u_bus_mon_ral.default_map.set_auto_predict(1);
                u_bus_mon_ral.reset();
endfunction: connect_phase

//final phase
function void bus_mon_top_env::final_phase(uvm_phase phase);
```

```
        uvm_report_server  ReportServer;
        string msg;
         super.final_phase(phase);
        ReportServer = uvm_report_server::get_server();

        msg = $sformatf("Simulation Ended: Errors = %0d; Warnings = %0d",
        ReportServer.get_severity_count(UVM_ERROR),
        ReportServer.get_severity_count(UVM_WARNING));

        if(ReportServer.get_severity_count(UVM_FATAL) +
        ReportServer.get_severity_count(UVM_ERROR) > 0)
                        msg = "*** FAILED ***";
        else    if(ReportServer.get_severity_count(UVM_WARNING) > 0)
                        msg = "*** PASSED WITH WARNINGS ***";
        else
                        msg = "*** PASSED ***";
        msg = {msg, "\n", $sformatf("Simulation Ended: Errors = %0d; Warning =
        %0d",     ReportServer.get_severity_count(UVM_ERROR),
        ReportServer.get_severity_count(UVM_WARNING))};
        `uvm_info("TEST_STATUS", msg, UVM_LOW)
endfunction: final_phase
```

After the testbench setup was completed, different test cases were developed. The requirement specification lists primarily five features; trace control, data-capture, extraction, filtering and AXI writer. Thus, for the scope of this work, five test cases were developed. In addition to this, register access test was also run to verify the register access in the design.

4.2.1.1 Register Access Test

After the testbench and the DUT were in place, the first test implemented was register access test to verify if the registers in the design were accessible. The intention of this test was to verify if the registers at a particular address offset are accessible by the software. In addition, the test also verifies that OKAY access [18] to addresses not on the address map is an error.

The register access test used RAL model and AXI master read and write sequences to carry out the test. The RAL model would give all the registers in the DUT and their

offset addresses. The AXI master write sequence was used to write to the registers in the design and the corresponding write response was read. Similarly, AXI master read sequence was used to read from the registers and corresponding read response was read. The write and read response on the AXI bus indicates the status of the transaction which is defined in AXI protocol specification.

Enumerated access mode was defined for the read and write transactions. The enumerated access modes: ***empty_addr***, ***ob_addr*** and ***valid_addr*** were used. Prior to any transaction driving, the modes were randomized. The code snippet from the test below shows the main part of the implementation.

```
//Enumerated access mode
typedef enum {empty_addr,ob_addr,valid_addr} mode;
 mode addr_mode=valid_addr;

//get registers from RAL
bus_mon_env.u_bus_mon_ral.get_registers(regs, UVM_HIER);
//get register offset address
foreach(regs[k])begin
reg_addr_offset[k]=regs[k].get_offset(bus_mon_env.u_bus_mon_ral.default_map);
end
//Randomize the access mode
randomize(addr_mode) with {addr_mode inside {[empty_addr:valid_addr]};};
//AXI master write sequence to write
wr_seq.write(seq_item.addr,seq_item.data[i],'hF,bresp,bus_mon_env.u_axi_lite_agt.
sqr,1);
//Write response check
 if(bresp==0 || bresp==1) begin
        `uvm_error("EMPTY ADDRESS WRITE ERROR", $sformatf("Address %0d is
        empty but access is successful",seq_item.addr))
 end
 else begin
        `uvm_info("SLVERR/DECERR", $sformatf("Access attempt to empty address
        %0d", seq_item.addr), UVM_LOW)
 End
//AXI Master read sequence to read
rd_seq.read(seq_item.addr, data, rresp, bus_mon_env.u_axi_lite_agt.sqr, 0, 0, 1);
//Read response check
if(rresp==0 || rresp==1) begin
```

```
        `uvm_error("EMPTY ADDRESS READ ERROR", $sformatf("Address %0d is
        empty but access is successful",seq_item.addr))
end
else begin
        `uvm_info("SLVERR/DECERR", $sformatf("Access attempt to empty address
        %0d", seq_item.addr), UVM_LOW)
end
```

4.2.1.2 Trace Control Block Test

After the register access was verified, the trace control block test was carried out. The primary purpose of this test was to verify if the trace control block is distributing the control and configuration parameters in the registers to the corresponding design blocks. The trace control block gets the control and configuration data from the register bank and distribute to the dedicated blocks in the design. For example, *enable_filter* field read from the register bank was distributed to the filtering unit.

The test implementation was done using the AXI VIP. Inputs to the design were driven using the AXI VIP. The output from the trace control block is a mix of *std_logic* and *std_logic_vector.* No VIP was developed to monitor these outputs but simply the waveforms of these output signals were manually monitored. This was possible as there were only 3 primary registers in the design.

The code snippet below shows the main_phase implementation:

```
task bus_mon_top_tcb_test::main_phase(uvm_phase phase);
 uvm_reg                regs[$];
 uvm_reg_addr_t         reg_addr_offset[$];
 logic [1:0]            w_resp;

 phase.raise_objection(this);
 `uvm_info("TCB TEST", "Starting Trace Control Block Test", UVM_LOW)
 bus_mon_env.u_bus_mon_ral.get_registers(regs, UVM_HIER);
 foreach(regs[k])        begin
        reg_addr_offset[k]=regs[k].get_offset(bus_mon_env.u_bus_mon_ral.defau
        lt_map);
 end
```

```
  foreach (reg_addr_offset[i]) begin
  assert(randomize(seq_item))
  else `uvm_warning("Randomization failed", "Failed to randomize seq item");

  wr_seq.write(reg_addr_offset[i], seq_item.data[0], 'hF, w_resp,
  bus_mon_env.u_axi_lite_agt.sqr,4);
  #5ns;
 end
 `uvm_info("TCB TEST", "Ending Trace Control Block Test", UVM_LOW)
 phase.drop_objection(this);
endtask: main_phase
```

4.2.1.3 Data Capture Unit Test

After the trace control block test was done, data capture test was implemented. The register access test verified that the configuration and control parameters were written to the dedicated registers. Only the trace control register in the register bank was written with the suitable values for data capture unit test as this is the only register responsible for the configuration and control of data capture unit in the design. The test setup for the data capture unit test is shown in Figure 32.

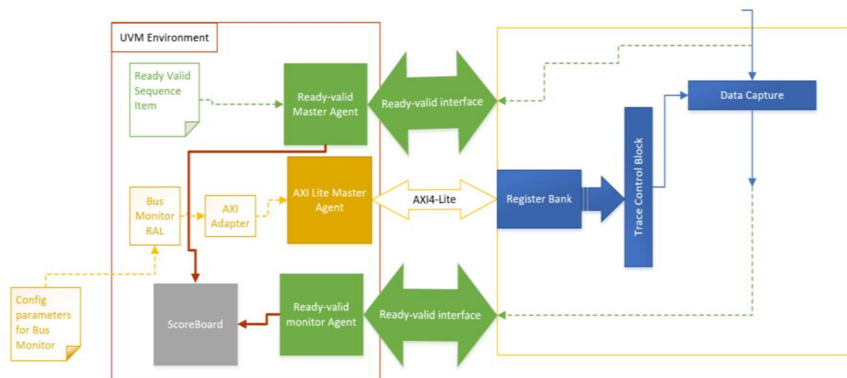


Figure 32: Data Capture Unit test setup

Figure 32 shows one ready-valid master agent and one ready-valid monitor agent. The master agent drives the ready-valid transaction to the DUT while the monitor agent

only monitors the output from the data capture unit. A scoreboard was implemented to compare the input and output to and from the data capture unit. The code snippet below shows the main_phase implementation of the data capture unit test.

```
task bus_mon_top_dcu_test::main_phase(uvm_phase phase);
 phase.raise_objection(this);
  `uvm_info("DCU TEST", "Starting Data Capture Unit Test", UVM_LOW)
   #5ns;
   for(int i=0; i < `NUM_OF_TRANSACTION; i++) begin
     assert(randomize(data))  else  `uvm_info("randomization  failed",$sformatf("Data
rondomization failed in %0d iteration",i), UVM_LOW)
     rv_seq.write(1'b1,1'b1,data, bus_mon_env.u_rv_master_agt.r_sequencer);
     #5ns;
   end
   #5us; //Grace for EOS
  `uvm_info("DCU TEST", "Ending Data Capture Unit Test", UVM_LOW)
 phase.drop_objection(this);
endtask: main_phase
```

The scoreboard implementation primarily has two tlm_analysis_fifo where the input and output data are collected. Data retrieved from these FIFOs were put into the uvm_queue prior to comparison. The code snippet below shows the implementation.

```
task bus_mon_scoreboard:: run_phase(uvm_phase phase);
  rv_seq_item   rv_input_data;
  rv_seq_item   rv_output_data;
  rv_input_data=new("rv_input_data");
  rv_output_data=new("rv_output_data");
  super.run_phase(phase);
  get_items(rv_input_data, rv_output_data, in_q, out_q);
endtask : run_phase

task  bus_mon_scoreboard::get_items(rv_seq_item  item1,  rv_seq_item  item2,  ref
uvm_queue #(logic [135:0]) q1, ref uvm_queue #(logic [135:0]) q2);
  for(int i=0; i < tr_count; i++) begin
  //forever begin
   fork
     begin
       mas_fifo.get(item1);
       q1.push_back(item1.data_in);
     end
```

```
    begin
      mon_fifo.get(item2);
      q2.push_back(item2.data_in);
    end
   join
  end
endtask:get_items

function void bus_mon_scoreboard:: extract_phase(uvm_phase phase);
  rv_seq_item rv_item;
  rv_item=new("rv_item");
  super.extract_phase(phase);
  compare_items(in_q, out_q);
endfunction : extract_phase

function void bus_mon_scoreboard:: compare_items(uvm_queue #(logic [135:0])
ref_item, uvm_queue #(logic [135:0]) comp_item);
  logic [135:0] item1, item2;
  int refq_size, compq_size;
  refq_size=ref_item.size();
  compq_size=comp_item.size();
  if(refq_size !== compq_size) begin
   `uvm_warning("Queue Size Mismatch", $sformatf("Ref queue size is: %0d, Comp
queue size is: %0d", refq_size, compq_size))
  end else begin
   for (int i=0; i < refq_size; i++) begin
    item1=ref_item.get(i);
    item2=comp_item.get(i);
    if(item1==item2) num_of_match++;
    else num_of_mismatch++;
   end
   `uvm_info("Comparison Result", $sformatf("Matches: %0d \t Mismatches:
%0d",num_of_match, num_of_mismatch), UVM_LOW)
  end
endfunction : compare_items
```

4.2.1.4 Extraction Unit Test

The purpose of extraction unit test was to verify the extraction process in the bus monitor. The bus monitor should extract the data field based of the ***select field*** value in the trace control register. Extraction unit test was implemented such that the test class would start the sequencer to drive the sequence items to the DUT and

scoreboard was implemented in the environment to evaluate the output. Figure 33 shows the extraction unit test implementation.

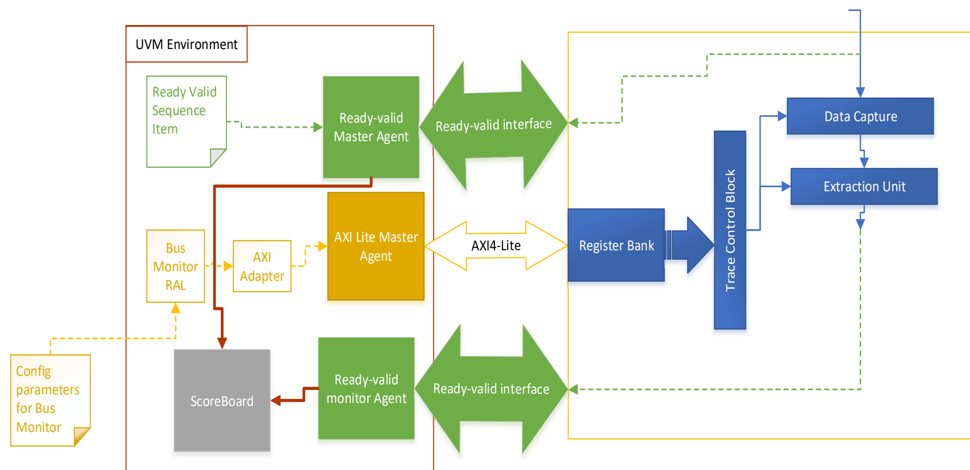


Figure 33: Extraction Unit Testbench

The configuration setup for the extraction unit test was same as in the data capture unit test. The ready-valid interface was connected to the ready-valid interface in between the extraction unit and filtering unit. A dedicated FIFO was implemented in the scoreboard to collect the extraction unit output. The reference model for this work was not in the scope and hence the extraction logic was implemented in the scoreboard itself to produce the extracted data. This extracted data is the reference against which the extraction unit output is compared to.

The extraction for the reference output was implemented with a method which takes a 5-bit vector *sel_field* to determine which field to extract, a two-dimensional unpacked array which stores a complete data structure in the interface in question and a uvm queue which stores the extracted data for the reference. The code snippet below shows the implementation of the extraction logic in system Verilog.

```
function void bus_mon_scoreboard:: extraction_logic(bit[4:0] sel_field, logic [3:0]
[`DATA_WIDTH-1:0] in_data, ref uvm_queue #(logic[87:0]) ref_q);
 logic [87:0] extracted_data;
 case(`DATA_WIDTH)
  136:begin
```

```
    if(sel_field[0]==1'b1) extracted_data[31:0]=in_data[0][31:0];
    if(sel_field[1]==1'b1) extracted_data[71:32]=in_data[2][39:0];
    if(sel_field[2]==1'b1) extracted_data[87:72]=in_data[0][15:0];
    ref_q.push_back(extracted_data);
  end
  ……..
endfunction;
```

A comparison method was implemented to compare the reference data and the output data and hence the extraction logic functional correctness was verified.

4.2.1.5 Filtering Unit Test

Filtering unit performs filtering if enabled. With filtering enabled, only the data field of interest could be filtered and accordingly the trace input can be provided based on the filtered data. The filtering unit takes the control and configuration information from the trace control block and performs the filtering. If filtering is not enabled, the unit simply propagates the input data to the output interface.

The purpose of the filtering unit test was to verify the filtering logic implemented in the bus monitor. The testbench setup for the filtering unit test is shown in the Figure 34. Similar to the extraction unit test, the reference to which the filtering unit output was compared against, was computed in the scoreboard and collected in a queue. The input to this computation in the scoreboard was extracted from input interface to the filtering unit. The output from the filtering unit was also collected in a queue in the scoreboard and a comparison method was implemented to compare the output.

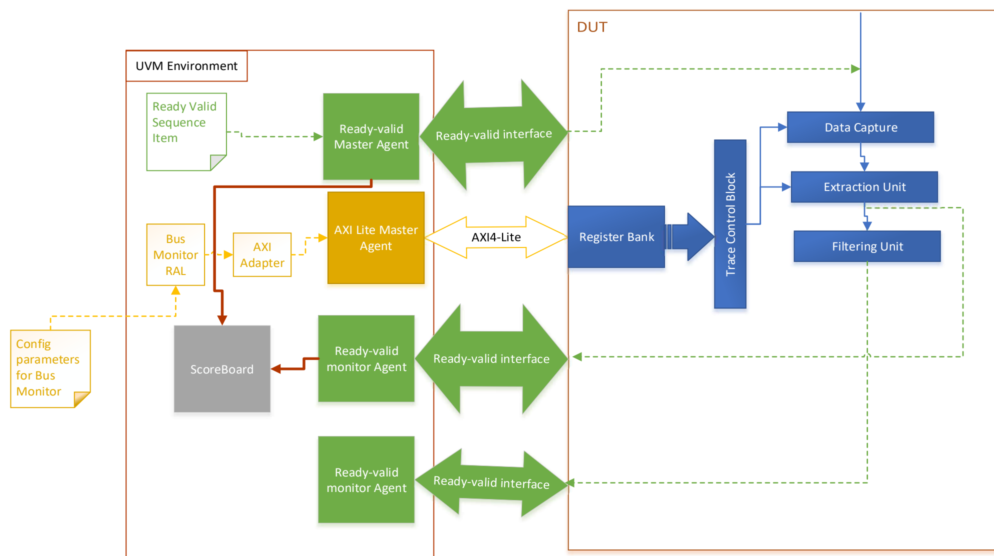


Figure 34: Filtering Unit Testbench

DUT configuration setup for this test was carried out in the *configure_phase*. The value written to the trace control register was the same as in previous tests. However, the filtering unit also requires the filter value register and filter mask register setup to carry out the filtering task. The code snippet below shows how it was implemented in the *configure_phase.*

```
task bus_mon_top_fu_test::configure_phase(uvm_phase phase);
 super.configure_phase(phase);
 phase.raise_objection(phase);
  fork
        bus_mon_env.u_bus_mon_ral.TRACE_CONTROL.write(status,
        'h0101_00FF,    UVM_FRONTDOOR);
        bus_mon_env.u_bus_mon_ral.FILTER_VALUE_0.write(status,'h0000_17A6,
        UVM_FRONTDOOR);
         bus_mon_env.u_bus_mon_ral.FILTER_MASK_0.write(status,'h0000_FFFF,
        UVM_FRONTDOOR);
   join
 phase.drop_objection(phase);
endtask: configure_phase
```

The code snippet above shows that the filter value is 17A6 and filter mask is FFFF. This means that the field of interest is 16-bit LSB from the extracted data. If that data is equal to 17A6, the filtering unit outputs the filtered data to be 17A6 and hence trace packet will be generated only for this data.

4.2.1.6 AXI Writer Unit Test

The bus monitor writes 64-bit AXI transaction item to the stimulus port in STM. If the data to be written is wider than 64 bits, burst transaction is used. The purpose of AXI Writer Unit test is to verify if the AXI writer module translated the valid-ready transaction item into correct AXI transaction. No reference logic was implemented to compare the translated AXI transaction. However, the AXI transaction were observed manually on the interface and also received in the scoreboard for manual evaluation. The AXI writer unit test setup was similar as in the case of filtering unit test. However, the filtering is disabled for this test.

AXI Slave agent was added to the UVM environment to monitor the AXI transaction on the AXI slave interface. As the agent was used as slave, it was responsible to assert the ready signals to receive the transaction and also to produce the response for the AXI master. The pictorial representation of AXI writer test setup is in Figure 31.

4.2.2 Build Environment Setup

All the design compilation and verification run in Nokia is typically launched in the grid system. Nokia provides various tools for the design and testbench compilation and simulation. Synopsis VCS was used for this project. VCS tool can be loaded to the Nokia's grid system by simply running the command *module load vcsmx/<version>*. The bus monitor project was created using the tool called DVT. DVT is the eclipse-based IDE. However, the real compilation and simulation was done using VCS based on Nokia's **ModularMake** [29] approach.

ModularMake is a makefile based framework for RTL simulation based verification in Nokia. Modularity comes from the aspect that there are three distinct Makefiles, each targeted for dedicated jobs. The Makefile system in ModularMake has three primary Makefiles: Makefile, Makefile.mk and Makefile_proj.mk. The top-level wrapper is called simply Makefile that routes the make commands to correct build directories. A build-root directory is created inside each project where the make command is run. Running the make command in the build-root creates build/img_directory. An img_directory is simply a directory which has the image, the combination of specific DUT and Testbench configuration.

The second Makefile called Makefile.mk is copied as Makefile into each image build directory and maintained to be up to date. There is only one image for this project. This Makefile also includes project specific Makefile_proj.mk. The third makefile called Makefile_proj.mk is a project specific makefile and it defines the project specific configuration, dut compilation rule, testbench compilation rule and simulation. Figure 35 shows the block diagram of ModularMake process. The project/image specific makefile called Makefile_proj.mk was setup for this work such that it in turn calls one makefile in the front end (fe) directory for DUT compilation and other makefiles in the verification (verif) directory. The verif directory has dedicated makefiles for VIP compilation, one in rv_vip directory and one in axi_vip directory. This approach enhances the modularity and allows parallelism during compilation.

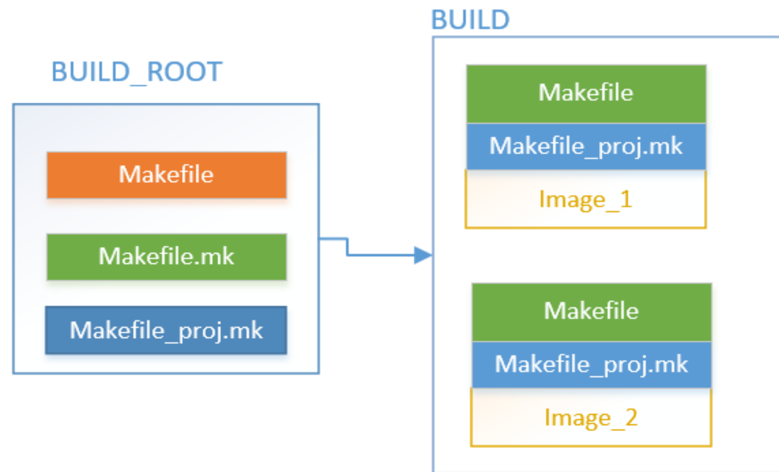


Figure 35: ModularMake [29]

Chapter 5

5 Results and Discussion

This section describes the obtained result from the VHDL testbench (directed test) and UVM testbench result. Section 5.1 includes the result obtained from the directed test and 5.2 includes the result from UVM testbench.

5.1 Directed Test Results

The following simulation results are based on the standalone directed test for each sub-module of the bus monitor. These simulations were run to see if the sub-modules function the way they should to a minimum extent and if the implemented interfaces were working. The simulations were run on Questasim.

5.1.1 Data Capture Unit Test Result

Data capture unit test was run to verify the data capturing feature of the bus monitor. Figure 36 shows the simulation result for the data capture block. As shown in the figure, the block has two pipeline stages: capture and write_out. Data is captured in the capture stage and output is written in write_out stage. The signal *data_input_i* is the data input to the module and *data_output_o* is the data output.

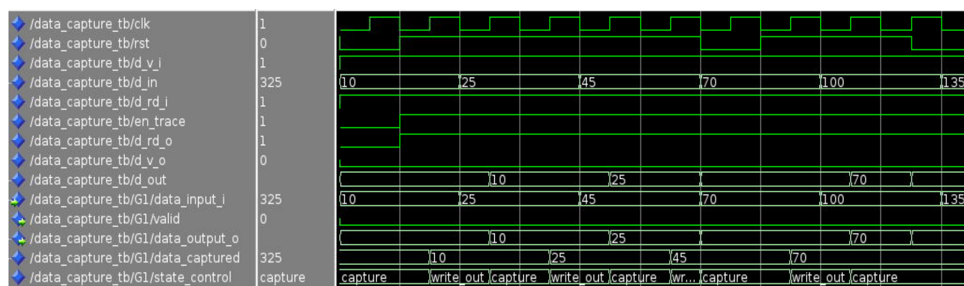


Figure 36: Directed test result for data capture unit

5.1.2 Extraction Unit Test Result

Based on the values in the select0-5 field in the trace control register, the extraction unit extracts the field of interest from the input data. Figure 37 shows the directed test result for extraction unit. The signal *sel_datafield* in the figure represents the value in the select0-5 field in the trace control register. The extraction unit has three pipeline stages: *data_collection*, *extraction* and *write_out*. In *data_collection* stage, the unit samples the input data and collect in an array. Only when the complete data structure is received, extraction is carried out and finally output is written in *write_out* stage. The waveform in Figure 37 shows that *sel_datafield* is “100” and hence as per the specification of bus monitor, only 8 bit scheduler group information is extracted which is a hex value CE as indicated by signal *d_out*.

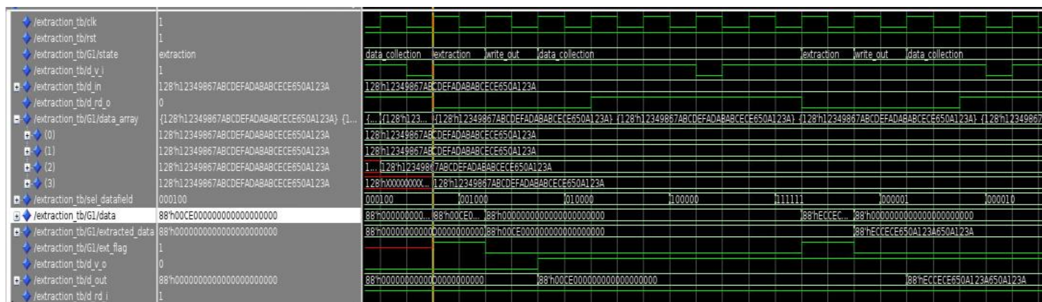


Figure 37: Directed test result for extraction unit

5.1.3 Filtering Unit Test Result

Filtering unit test was run to verify the filtering logic of the bus monitor. Figure 38 shows the waveform of the directed test for filtering unit. Filtering is performed only if the filtering unit is enabled in trace control register. Moreover, the filtering unit uses the filter value from the filter value register and filter mask value from filter mask register to carry out the filtering task. The signal *en_filter* in the figure indicates whether or not the filtering is enabled. Signal *fil_val* and *fil_mask* indicates the value in the filter value register and filter mask register.

In Figure 38, signals *entry_pre_mask* and *entry_post_mask* represent the data prior to masking and post masking respectively. If the value in *entry_post_mask* signal is equal to the filter value, the filtered data is written to the output. The signal *data_out* represents the filtered output data.

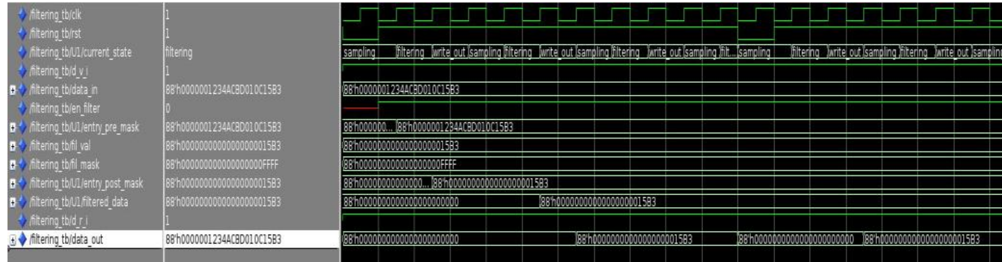


Figure 38: Directed test result for filtering unit

5.1.4 AXI Master Writer Test Result

The bus monitor captures the transaction in the interface it is connected to and produce an AXI output that complies with the trace input for the ARM CoreSight. The AXI translation is carried out by the AXI Master Writer module. Figure 39 shows the output produced by the AXI writer module.

The signals shown in the figure with prefix M are the AXI write channel signals. *M_AWADDR_O* represents the write address and *M_WDATA_O* represents the data to be transferred over AXI interface. *M_WSTRB_O* represents the strobe signal to indicate the valid byte lane. *M_WLAST_O* indicate that the beat being transferred is the last beat. Note: Beat represents the amount of data transferred in a single transfer in a burst transaction.

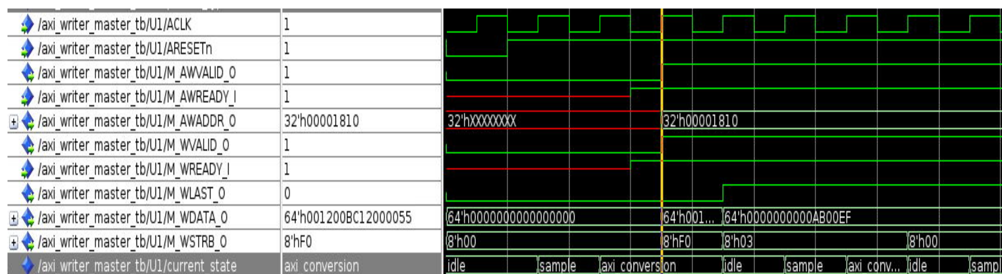


Figure 39: Directed test result for axi master writer

5.1.5 Trace Control Block Test Result

Trace control block reads the control and configuration information from the register bank and distributes the information to other sub blocks of the bus monitor. Figure 40 shows the directed test simulation result for the trace control block. The abstracted signal *t_cfg* represents the control and configuration information read from the register bank and the other signals (except *clk* and *rest*) represent the output from the trace control block. For an example, *select_datafield* indicates the select0-5 field in the trace control register.

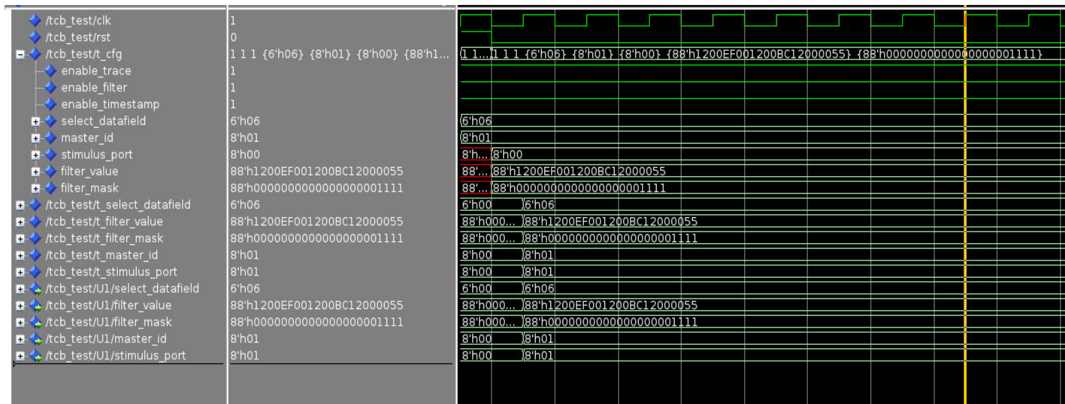
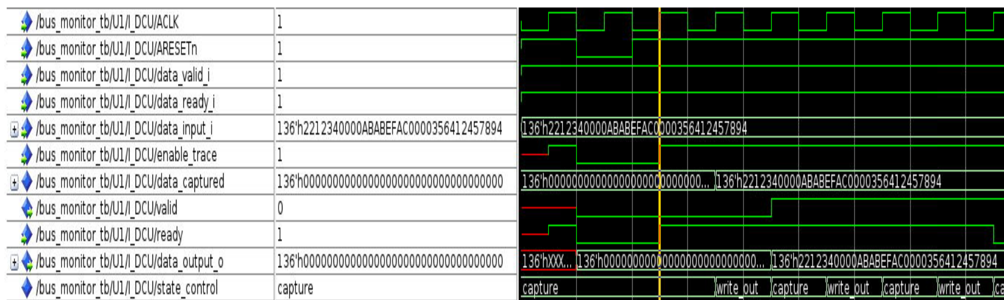


Figure 40: Directed test result for trace control block

After running the directed test for each of the sub modules of the bus monitor, all these sub modules were instantiated in the bus monitor top module and again a directed test was run. Figure 41 shows the directed test simulation result for the bus monitor top level.



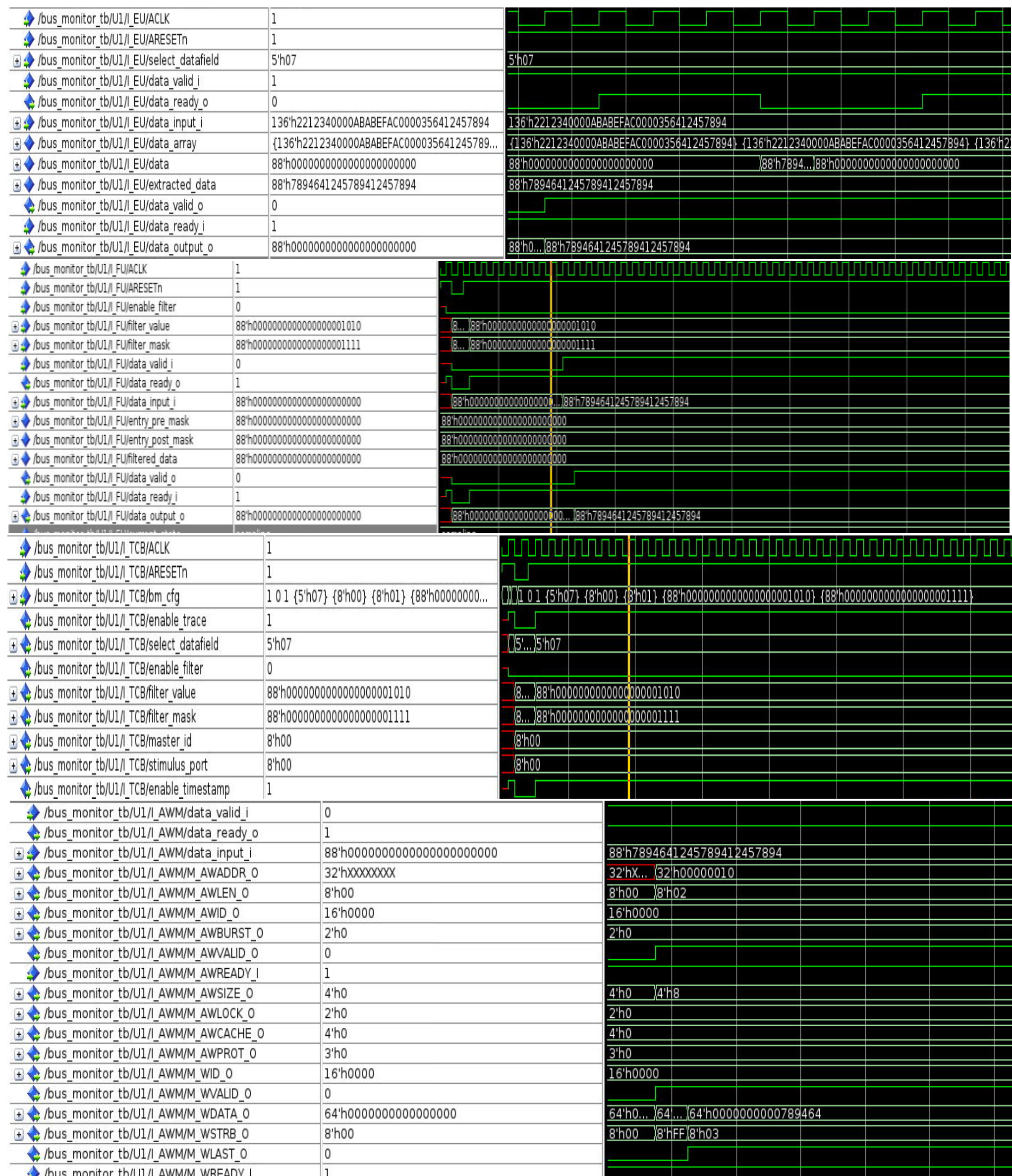


Figure 41: Directed test for the bus monitor

5.2 UVM Test Results

UVM methodology-based verification not only provides the automation in the test stimuli generation but also allows to constrain and randomize the stimuli such that not only the anticipated bugs are tracked but also the unanticipated ones. This section discusses the simulation result achieved after running the UVM based test. The simulation was run on DVT eclipse. For the scope of this work, the primary features requirement outlined by the specification were verified which includes: register access, trace control block features, data capture, extraction, filtering and AXI translation.

5.2.1 Register Access Test

Register access test was run to verify if the registers in the design is accessible by the software or not. The data to be written to these registers were randomized prior to writing to the registers. The write response was checked. The response is OK if it is 2b00. A subsequent read was performed after the registers were written. Figure 42 shows the values written to the registers in the design.

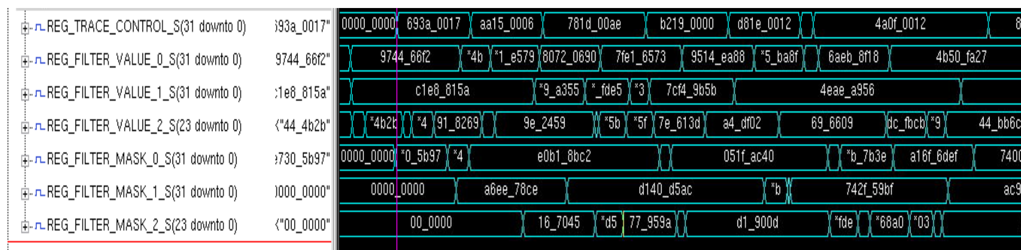


Figure 42: Register Access uvm test

5.2.2 Trace Control Block Test

Figure 43 shows the uvm based test result for the trace control block. The abstracted signal *BM_CFG* reads the register information from the register bank and writes the output on the corresponding output signals.

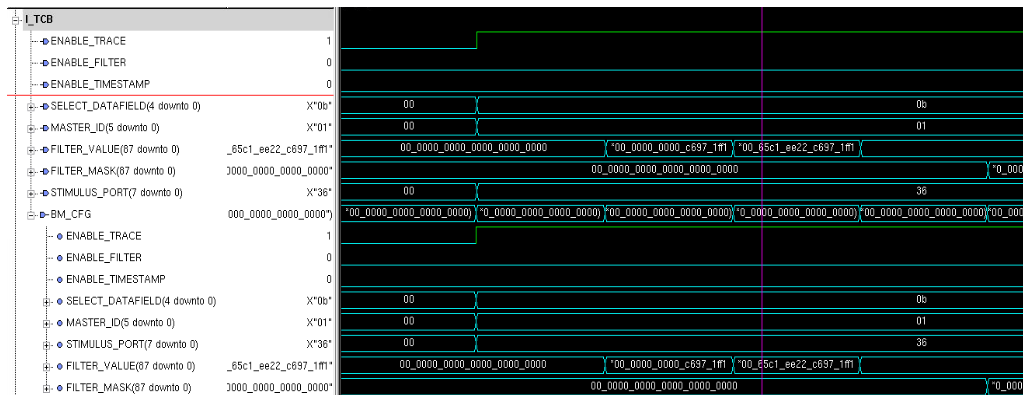


Figure 43: Trace control block uvm test

5.2.3 Data capture unit test

Figure 44 shows the waveform of the input and output interfaces connected to the data capture unit. *rv_vif* is the input interface and *rv_dcu_out_if* is the output interface.

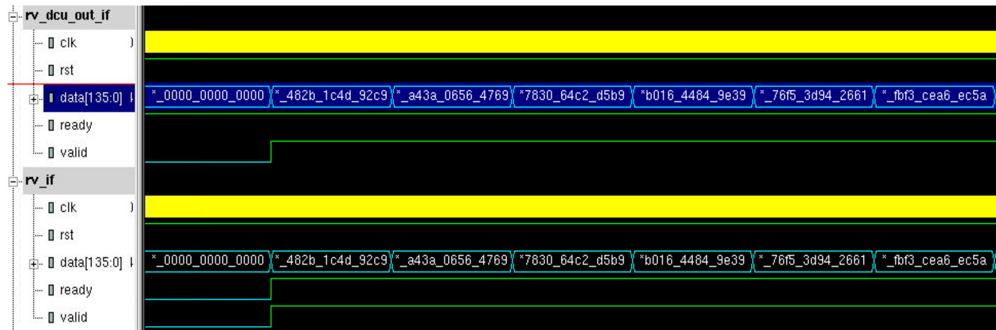
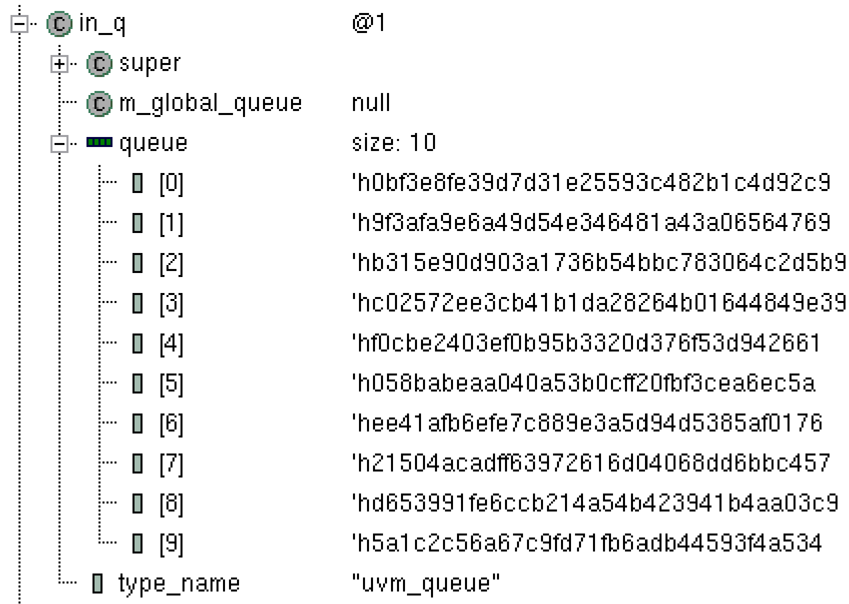


Figure 44: Data capture unit uvm test

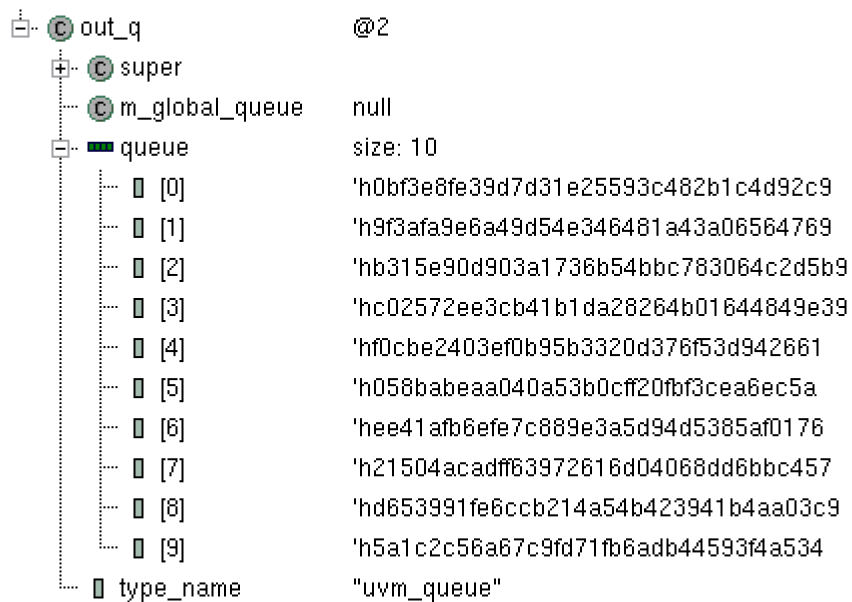
A scoreboard was implemented to compare the output from the data capture unit and its input. There were two agents; one connected to *rv_dcu_out_if* and the other connected to *rv_if.* Transactions on each of these interfaces were collected in the scoreboard and compared.

The data collected in the reference queue is in the screenshot below. The reference against which the data capture unit output was compared against is *in_q*. The number

of transaction set for this test was 10 and hence the queue size. The data in in_q was extracted from the input interface.



The output data collected from the Data Capture Unit output interface is below.



5.2.4 Extraction Unit Test Result

Figure 45 shows the waveform view of the input and output from the extraction unit.

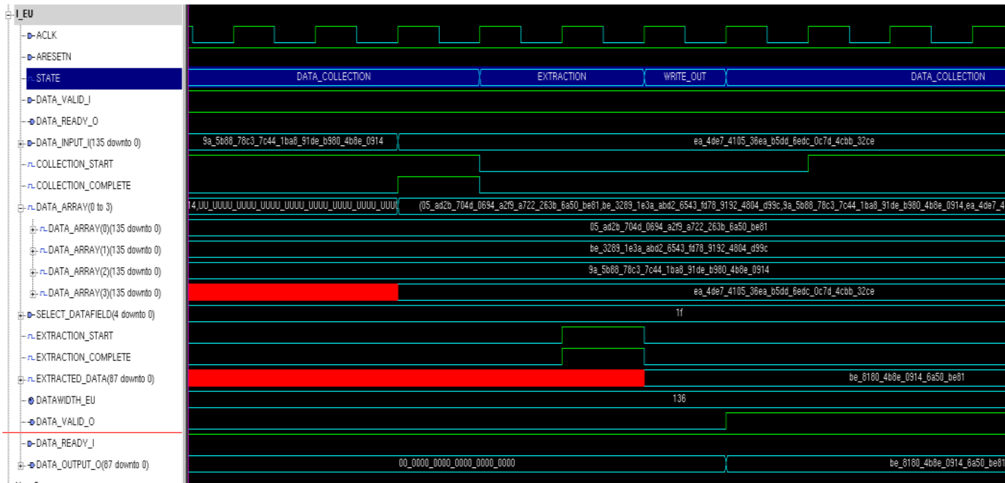


Figure 45: Extraction unit uvm test

The reference data against which the extraction unit output was compared was generated in the scoreboard. A simple system Verilog method was written for this. The output from the extraction unit was collected in a FIFO in the scoreboard.

The following screenshot shows the data collected in the reference queue and the output queue for the extraction unit test. The number of transaction was set to 20. The test was run for the bus width 136 which is the EM2TX interface between the event manager and the TX queue in the EMA. The complete data structure transfer requires four transactions and hence 20 transactions make up 5 complete data structure transfer on the interface in question. The items collected in the *ref_q_eu* in Figure 46 shows the item generated by the reference logic implemented in System Verilog while the *out_q_eu* shows the item collected from the Extraction unit output interface.

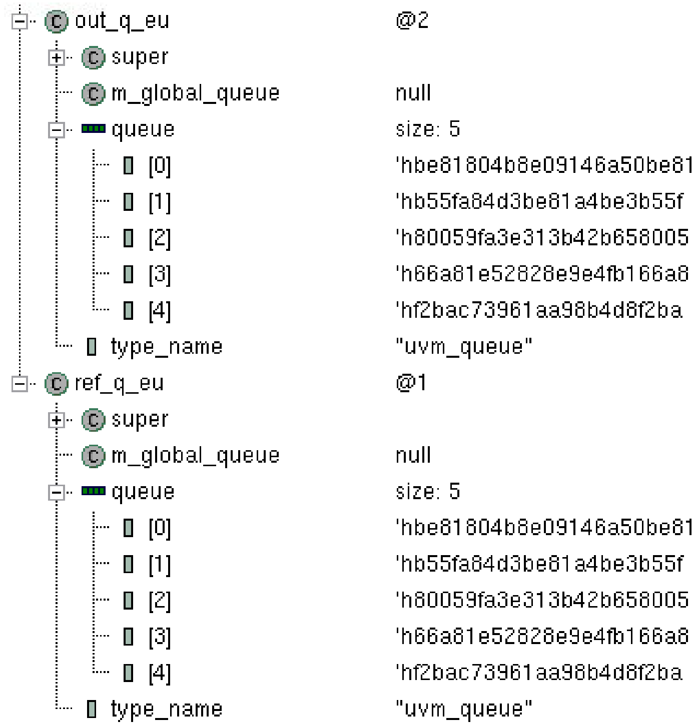


Figure 46: Screenshot of the reference queue and output queue for extraction unit

5.2.5 Filtering Unit Test Result

Figure 47 shows the waveform view of the input and output from the filtering unit.

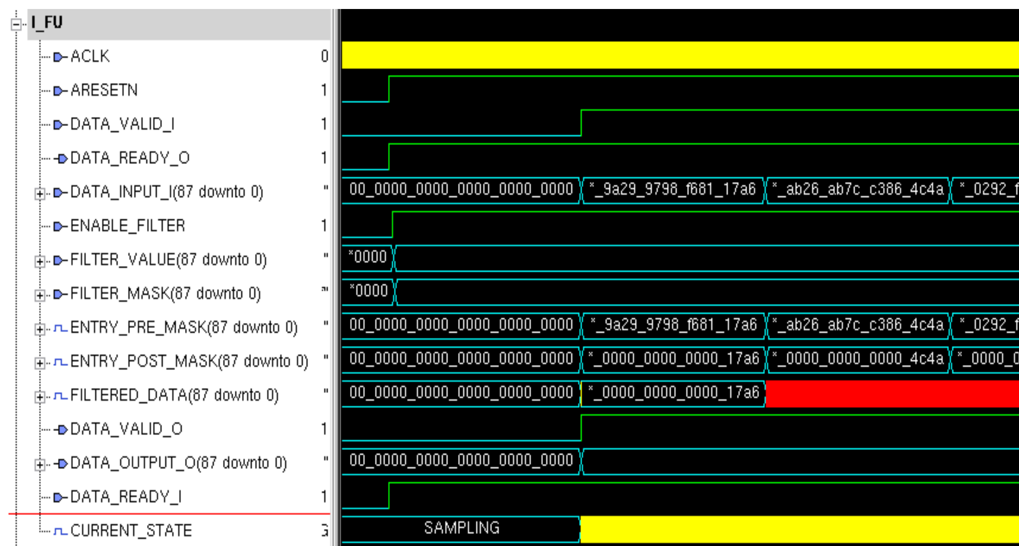


Figure 47: Filtering unit uvm test

The output from the filtering unit was collected in a FIFO in the scoreboard. In addition, a function was implemented in the scoreboard to carry out the filtering which would produce the reference data against which the output collected in the FIFO was compared. The *ref_q_eu and out_q_eu* in Figure 48 show the reference item and the output collected in the FIFO respectively.

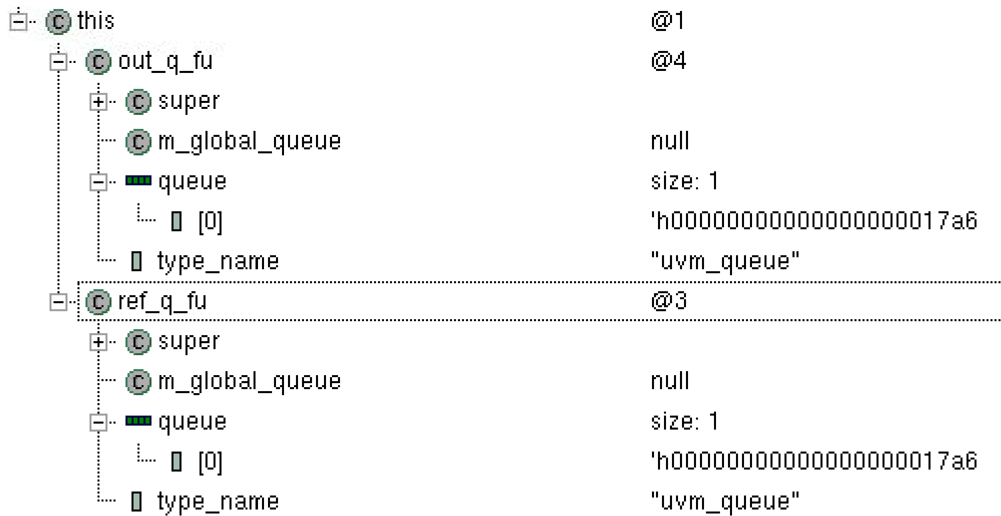


Figure 48: Screenshot of the reference queue and output queue for filtering unit

5.2.6 AXI Writer Unit Test

Figure 49 shows the transactions on the AXI slave interface. The transactions item on the interface were manually observed and verified for this particular test.

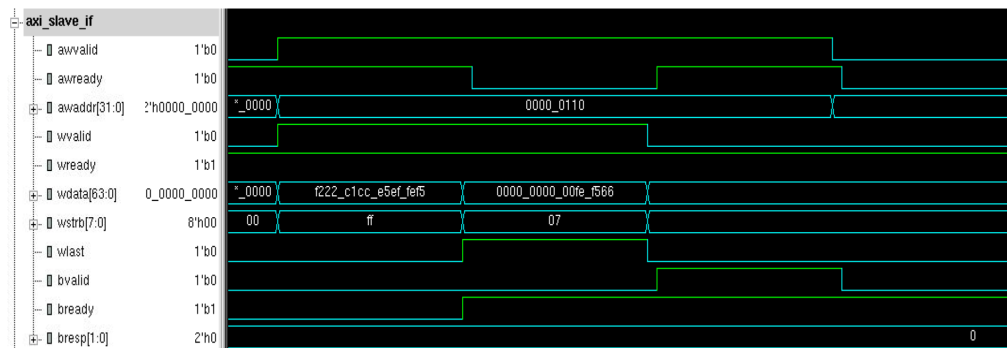


Figure 49: AXI Slave interface

The AXI items were also collected in a FIFO in the scoreboard. Figure 50 shows the item collected in the FIFO. For 20 sequence items, the generated corresponding AXI transaction is 5 for the reason explained earlier. Only the data items are collected in the queue shown in the figure as address is the same for the fixed burst which is 0000_0110 in this case.

Figure 50: Screenshot of the reference queue and output queue for AXI writer unit

Chapter 6

6 Conclusion and Further Developments

Debug and trace feature in any HW design is non-trivial as the SW intended to run on the HW may not function the way it is supposed to. Tracing the data allows to have a clear picture of what is working and what is not during the execution which is the most effective way to fix the SW. The bus monitor in the DTSS non-invasively captures the transaction on the interfaces and generates the trace input data to the CoreSight architecture. The CoreSight architecture then produces the trace output packet depending on the configuration written to the CoreSight.

Figure 51 shows the eventual trace data output in SoC level that a software programmer can see. DTSS programming is done via JTAG interface and trace data output is observed via PCIe (peripheral Component Interconnect express).

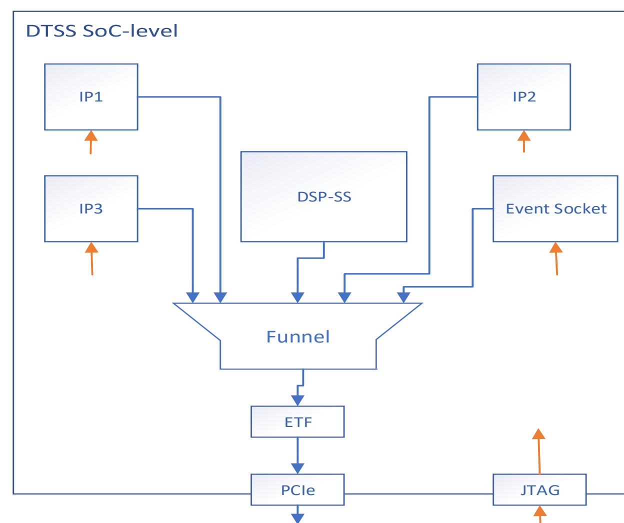


Figure 51: DTSS in SoC-level

The primary features listed in the feature requirement of the bus monitor; data capture, extraction, filtering, AXI translation and glue logic in trace control block were implemented and verified. The scope of this work involves the RTL implementation of the bus monitor and its verification. The verification of the bus monitor was carried out for all the interfaces shown in green arrows in Figure 9. However, the one discussed in the results and discussion section is for the EM_TXDATA interface which has a 136-bit wide data bus and a complete transfer of the data structure on this interface requires four transactions. The simulation results for the other interfaces are listed in the appendix section.

6.1 Further Developments

This thesis work is mainly concerned with the design implementation and verification of the bus monitor IP used in DTSS in ES. It is thus the instrumentation (art of measuring) of performance of the SW application running in the ES is outside the scope of this work. However, the performance instrumentation can be carried as further task to prove the legitimacy of the ES based HW acceleration.

The integration of the bus monitor with the standard ARM CoreSight components (depicted as CoreSight sub-system in Figure 11) to make a complete DTSS in ES will be carried out as further development. The goal of the DTSS in ES is to have a functioning debug and trace sub-system design which can be integrated to the Debug and Trace Sub-System in SoC level. Integrating the ES DTSS in Debug and Trace Sub-System in SoC level will also be carried out as further development, a glance of which is depicted in Figure 51.

References

1. Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. Retrieved from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4785615.

2. Nokia. (2018). Event Socket. Retrieved from https://nokia.sharepoint.com/:w:/r/sites/EventSocketGeneral/_layouts/15/Doc.aspx?sourcedoc=%7B517D89DF-FC79-41D3-A846-56A2C4655D11%7D&file=Event%20Socket%20HW%20Architecture%20Specification.docx&action=default&mobileredirect=true

3. A.T.B Hopkins, K.D. McDonald-Maer. Debug support strategy for systems-on-chips with multiple processor cores. Retrieved from https://ieeexplore.ieee.org/document/1566578

4. S. Zoran, H. Klaus, K. Milos, E. Victor and S. Ignacio.2011. MAC and baseband processors for RF-MIMO WLAN. Retrieved from https://www.researchgate.net/publication/228453031/download

5. National Instruments. What is I/Q Data? Retrieved from http://www.ni.com/tutorial/4805/en/

6. Nokia. (2013). Open Event Machine. *Sourceforge*. Retrieved from http://download2.nust.na/pub4/sourceforge/e/project/ev/eventmachine/Documents/EM_introduction_1_0.pdf

7. Nokia. (2018). *Event Manager Design Specification.* Retrieved from https://nokia.sharepoint.com/:w:/r/sites/EventSocket20/_layouts/15/Doc.aspx?sourcedoc=%7B1C17482B-0AE9-4BDA-AF15-BBADE2729FAE%7D&action=edit&source=https%3A%2F%2Fnokia%2Esharepoint%2Ecom%2Fsites%2FEventSocket20%2FSitePages%2FHome%2Easpx%3FRootFolder%3D%25

8. Nokia. (2018). *Event Manager EMA Design Specification.* Retrieved from https://nokia.sharepoint.com/:w:/r/sites/EventSocket20/_layouts/15/Doc.aspx?sourcedoc=%7B0B022752-5603-4F0B-89E7-4674F331F5F8%7D&action=edit&source=https%3A%2F%2Fnokia%2Esharepoint%2Ecom%2Fsites%2FEventSocket20%2FSitePages%2FHome%2Easpx%3FRootFolder%3D%25

9. Nokia. (2018, June). *Buffer Manager Design Specification.* Retrieved from https://nokia.sharepoint.com/:w:/r/sites/EventSocket20/_layouts/15/Doc.aspx?sourcedoc=%7B3CA485D2-0C6A-4194-A6D5-CC6DD1D1BE36%7D&file=Buffer%20Manager%20Design%20Specification.docx&action=default&mobileredirect=true

10. Nokia. (2018). *Buffer Manager EMA Design Specification.* Retrieved from https://nokia.sharepoint.com/:w:/r/sites/EventSocket20/_layouts/15/Doc.aspx?sourcedoc=%7B33B72397-11D5-4027-9E9A-B7E0F938A4DB%7D&file=Buffer%20Manager%20EMA%20Design%20Specification.docx&action=default&mobileredirect=true

11. Nokia. (2018). *Debug and Trace Sub System Architecture in Event Socket*. Retrieved from https://nokia.sharepoint.com/:w:/r/sites/EventSocket20/_layouts/15/Doc.aspx?sourcedoc=%7B638DAB30-B181-4469-B530-1C5DA4DD9484%7D&file=EM20_IP_Debug_and_Trace.docx&action=default&mobileredirect=true

12. *ARM CoreSight SoC-400-r3p2.* (2015). Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0480g/DDI0480G_coresight_soc_trm.pdf

13. ARM. (2014). *ARM CoreSight STM-500 System Trace Macrocell*. Retrieved May 25, 2018, from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0528b/DDI0528B_coresight_system_trace_macrocell_r0p1_trm.pdf

14. ARM. (2010). *CoreSight Trace Memory Controller.* Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0461b/DDI0461B_tmc_r0p1_trm.pdf

15. ARM. (2017, December). *Cross Trigger Interface*. Retrieved June 1, 2018, from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344b/DDI0344.pdf

16. ARM. (2013). *ARM Corelink NIC-400 Network Interconnect.* Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0475b/DDI0475B_corelink_nic400_network_interconnect_r0p1_trm.pdf

17. MIPI Alliance. (2018). Retrieved from https://www.mipi.org/

18. ARM. (2011). *AMBA AXI and ACE Protocol Specification*. Retrieved June 1, 2018, from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html

19. ARM. (2004). *AMBA 3, APB Protocol Specification*. Retrieved June 1, 2018, from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html

20. ARM. (2008). *AMBA 3 ATB Protocol Specification.* Retrieved from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html

21. Mentor. *UVM Cookbook.* Retrieved from https://verificationacademy.com/cookbook

22. B. Vermeulen, T. Waayers, S. Bakker. IEEE 1149.1-compliant access architecture for multiple core debug on digital system chips. Retrieved from https://ieeexplore.ieee.org/document/1041745?ALU=LU1043072

23. F. Moerman. Open event machine: A multi-core run-time designed for performance. Retrieved from https://ieeexplore.ieee.org/document/6924355/authors#authors

24. N. Stollon, R. Collins. Nexus Based Multi Core Debug. Retrieved from http://nexus5001.org/wp-content/uploads/2015/02/DesignCon_2006_Nexus_FS2_Freescale.pdf

25. B. Schwaller, B. Ramesh, A.D. George. Investigating TI KeyStone II and quad-core ARM Cortex-A53 architectures for on-board space processing. Retrieved from https://ieeexplore.ieee.org/document/8091094

26. IEEE. IEEE 1149.1 Standard. Retrived from https://standards.ieee.org/standard/1149_1-2013.html

27. R. Stence. Real Time Calibration and Debug Techniques of Embedded Processors with the Nexus 5001Interface. Retrieved from https://www.researchgate.net/publication/296664139_Real_Time_Calibration_and_Debug_Techniques_of_Embedded_Processors_with_the_Nexus_5001_Interface

28. K.D Maier. On-chip debug support for embedded systems-on-chip. Retrieved from https://ieeexplore.ieee.org/document/1206375/authors#authors

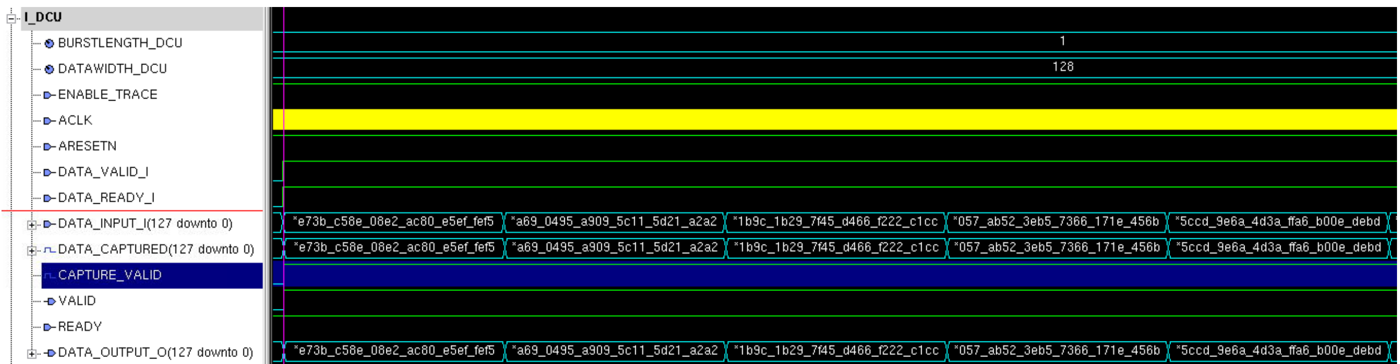
29. Nokia. *Modular Make and VMT System*. Retrieved from https://nokia.sharepoint.com/sites/soc-dftp/verification/SitePages/Modular%20Make%20and%20VMT%20System.aspx

Appendices

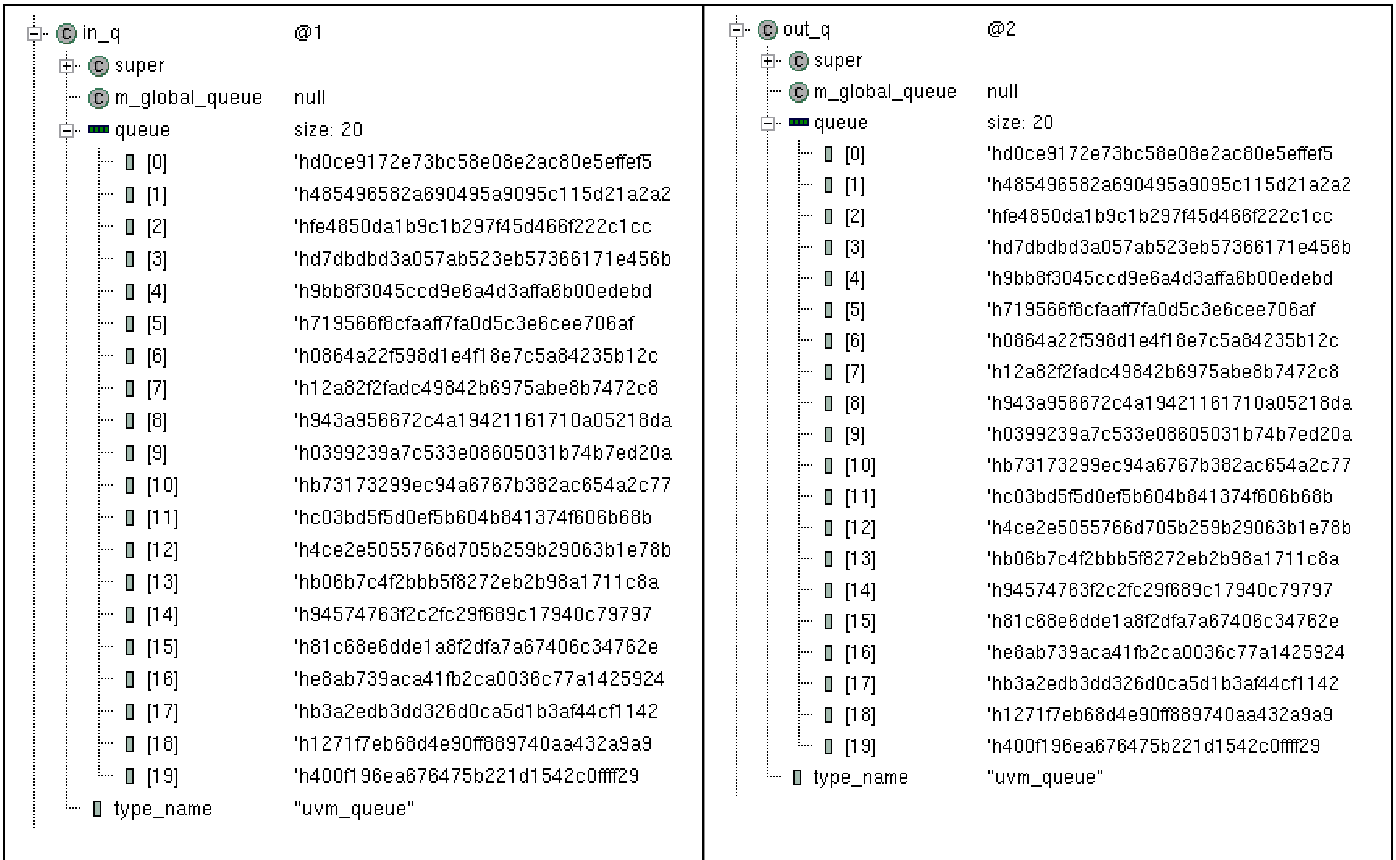
There are seven interfaces in ES to which a bus monitor is connected to non-invasively extract the transaction on the interfaces and generate a trace input data for the ARM CoreSight. The simulation results for six other interfaces (except the one mentioned in result and discussion section for bus width 136) are listed here in the appendix section.

Appendix 1: Simulation results for the bus monitor on the interface between RX queue and EM

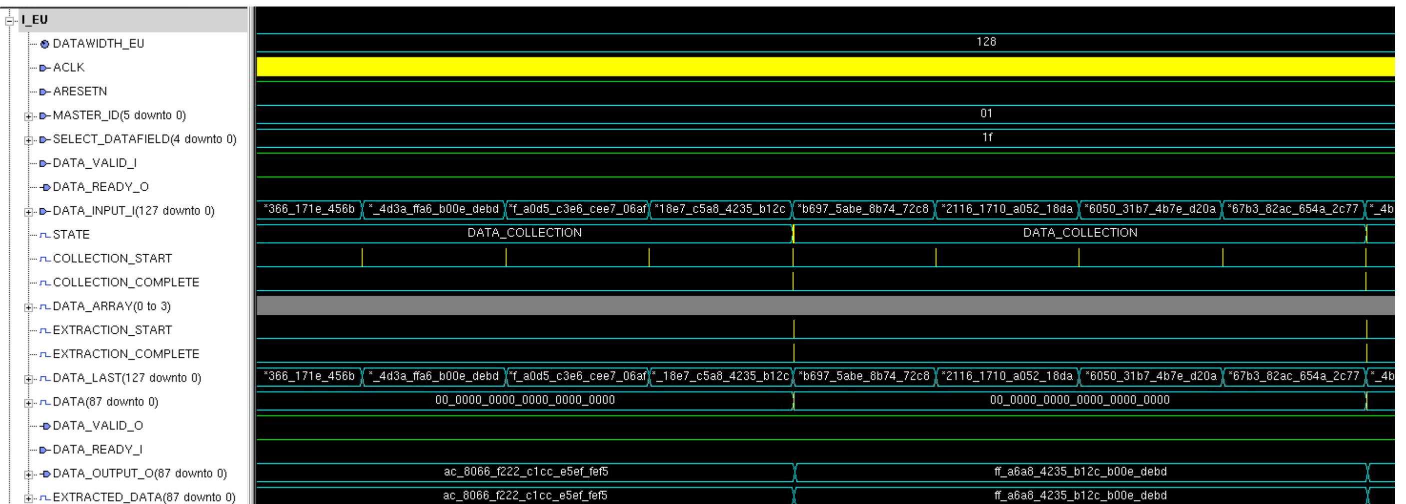
This section includes the simulation result for the bus monitor on the interface between RX queue and EM with bus width of 128 bit. The following screenshot shows the waveform view of the **data capture unit interface**.



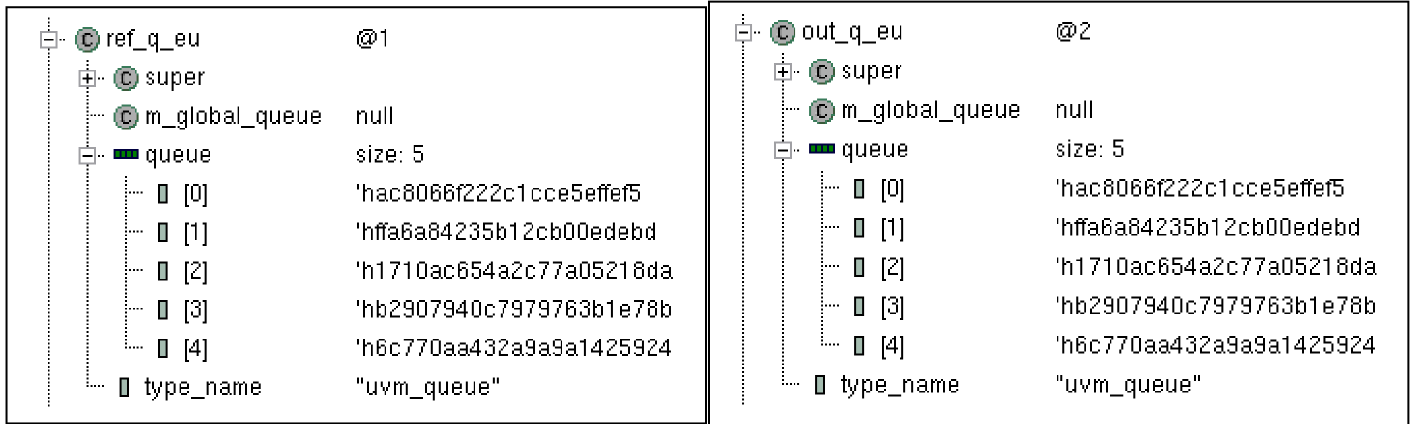
The following screenshot shows the data collected in the reference queue (in_q) and output queue(out_q) in the UVM Scoreboard for the **data capture unit**.



Extraction unit interface waveform view.

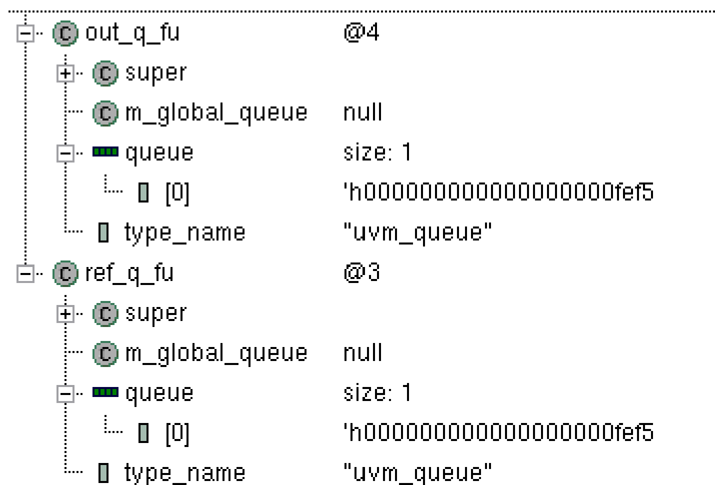


Extraction unit reference item and output collected in a queue for comparison in the scoreboard.

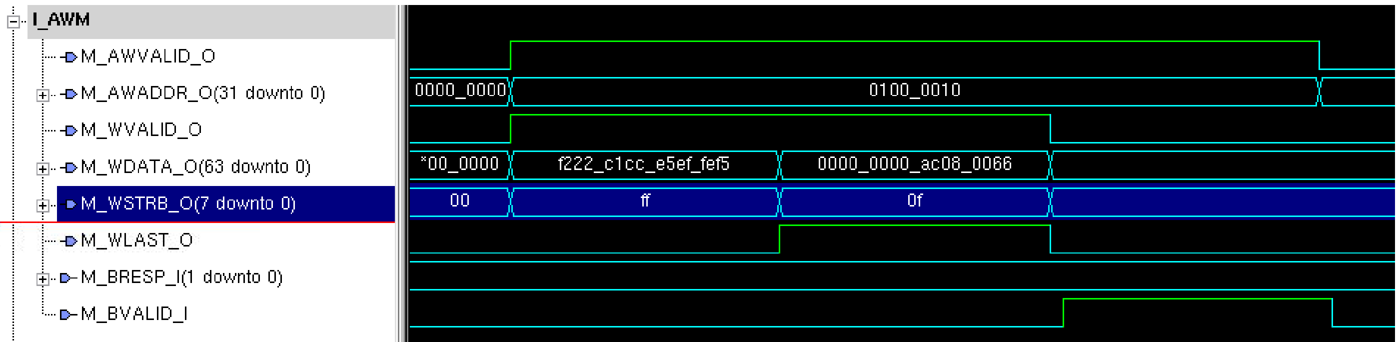


Filtering unit interface waveform is below.

The following screenshot shows the item collected in reference queue and the output queue in the scoreboard for the **filtering unit**. The filter value in filter value register is FEF5 and hence only the data field with value equal to FEF5 is filtered out.



AXI writer module interface output is below.

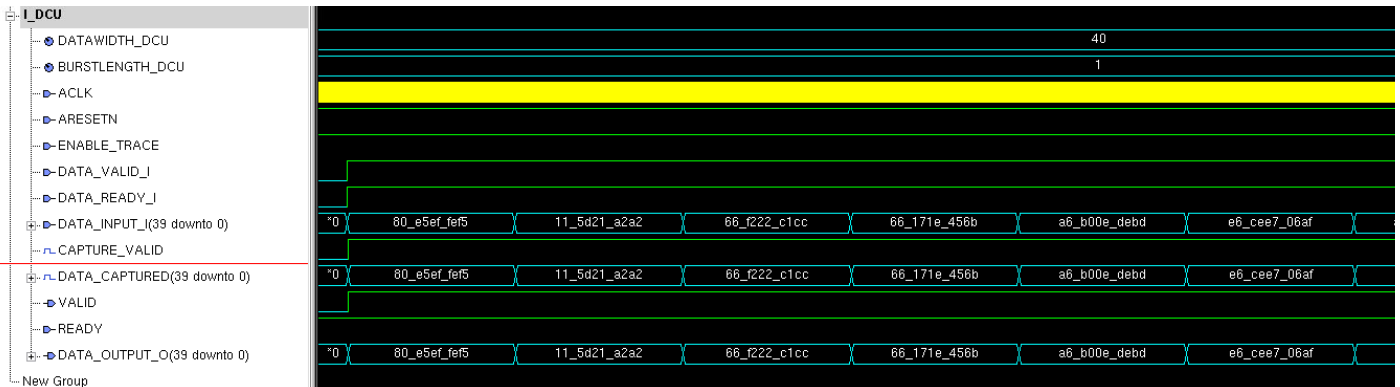


AXI output collected in a uvm queue is below.

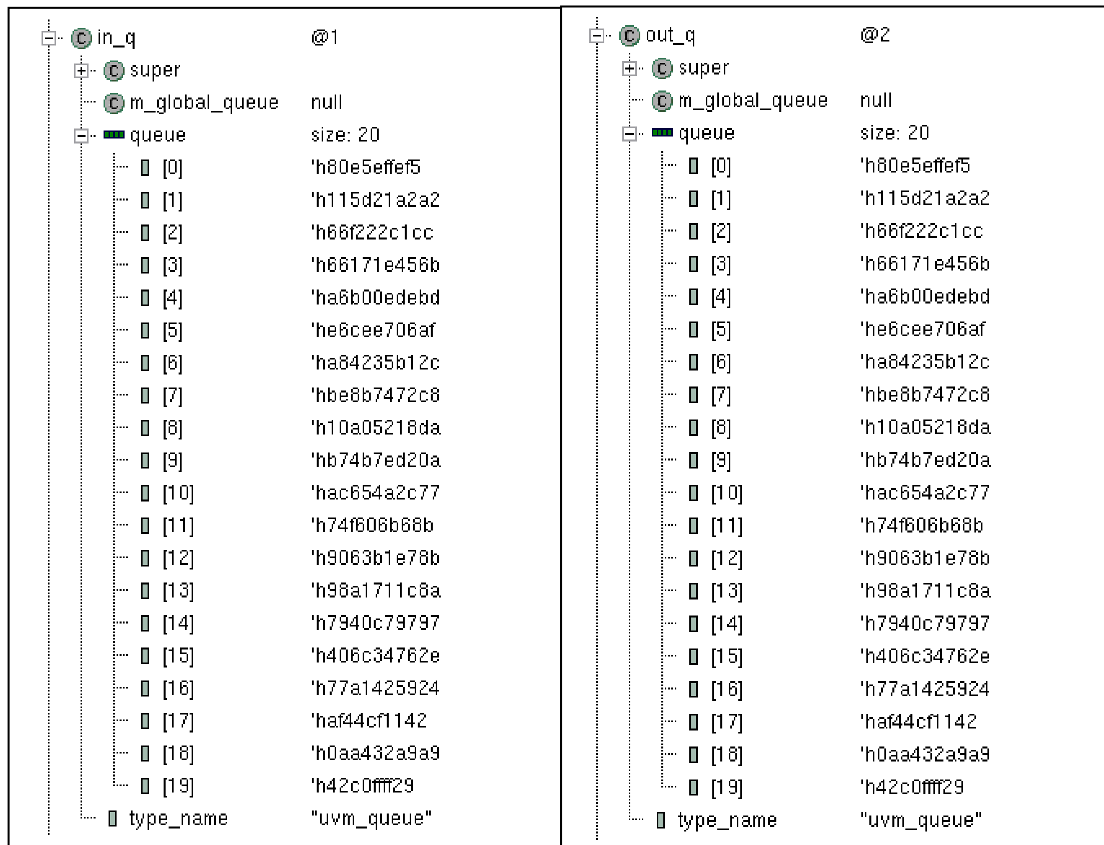
Item	Value
axi_data_q	size: 5
[00]	size: 2
[0]	'hf222c1cce5effef5
[1]	'h00000000ac080066
[01]	size: 2
[0]	'h4235b12cb00edebd
[1]	'h00000000ff0a06a8
[02]	size: 2
[0]	'h654a2c77a05218da
[1]	'h00000000170100ac
[03]	size: 2
[0]	'h40c7979763b1e78b
[1]	'h00000000b2090079
[04]	size: 2
[0]	'ha432a9a9a1425924
[1]	'h000000006c07070a

Appendix 2: Simulation results for the bus monitor on APC interface

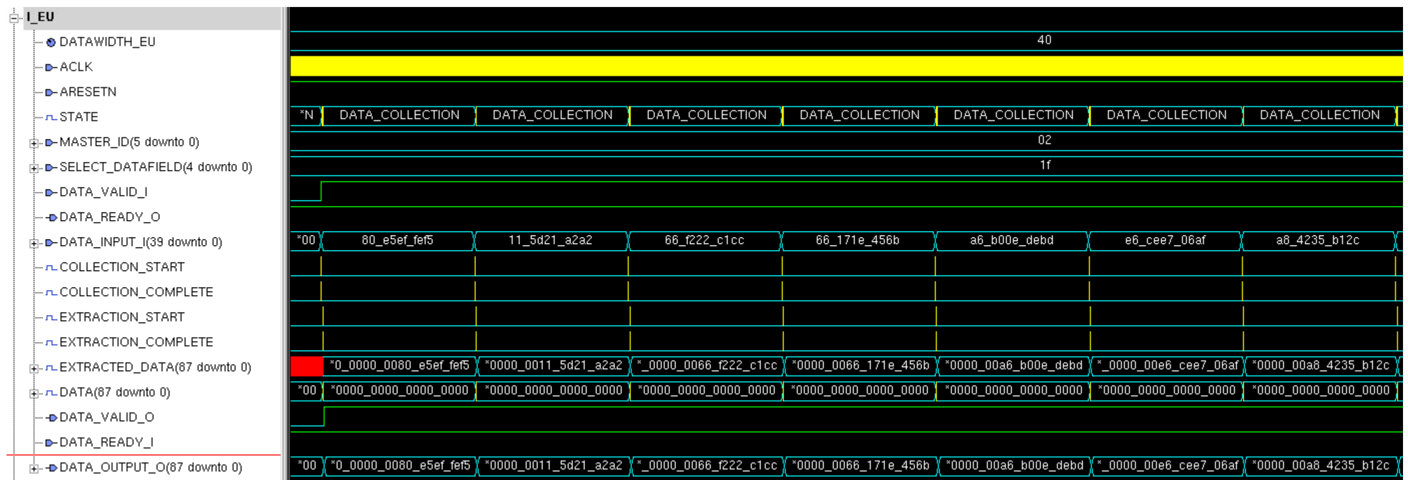
This section includes the simulation result for the bus monitor on the atomic processing complete interface and bus width is 40 bits. The following screenshot shows the waveform view of the **data capture unit interface**.



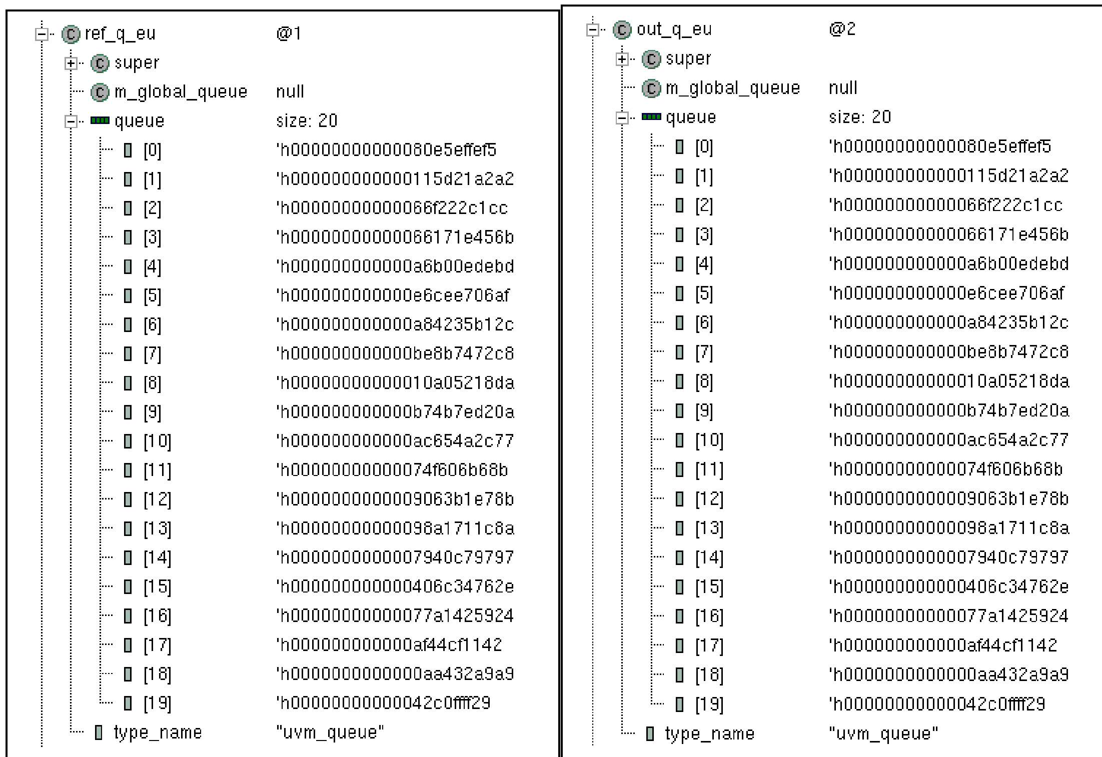
The following screenshot shows the data collected in the reference queue (in_q) and output queue(out_q) in the UVM Scoreboard for the **data capture unit**.



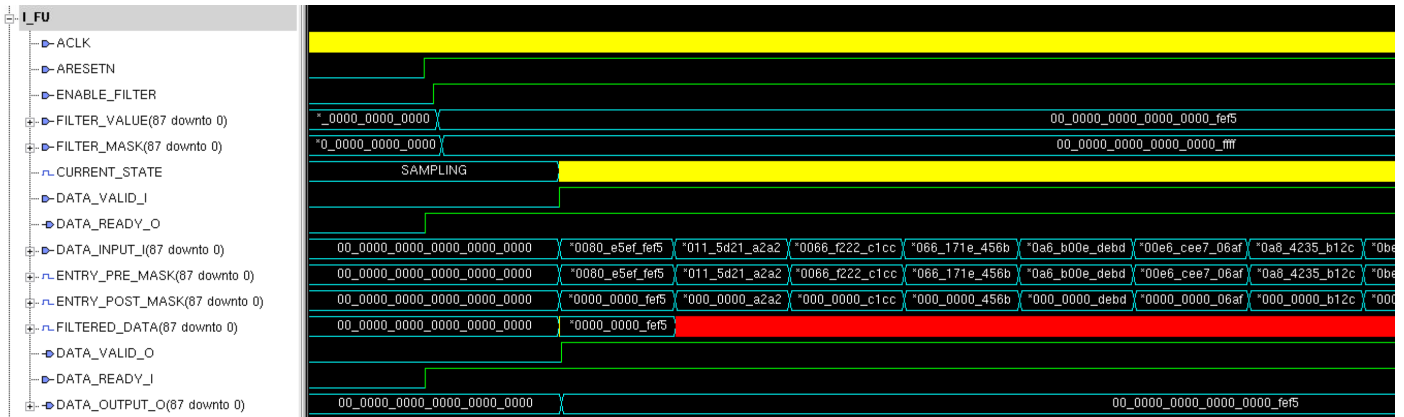
Extraction unit interface waveform is shown in the screenshot below:



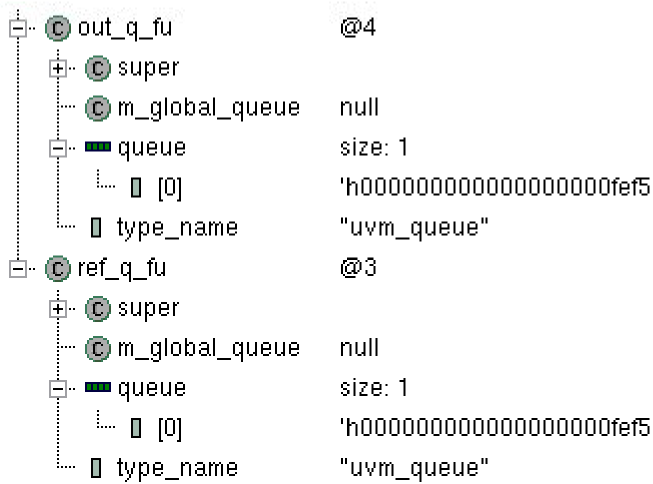
Items collected in the reference queue and output queue for the extraction unit is below:



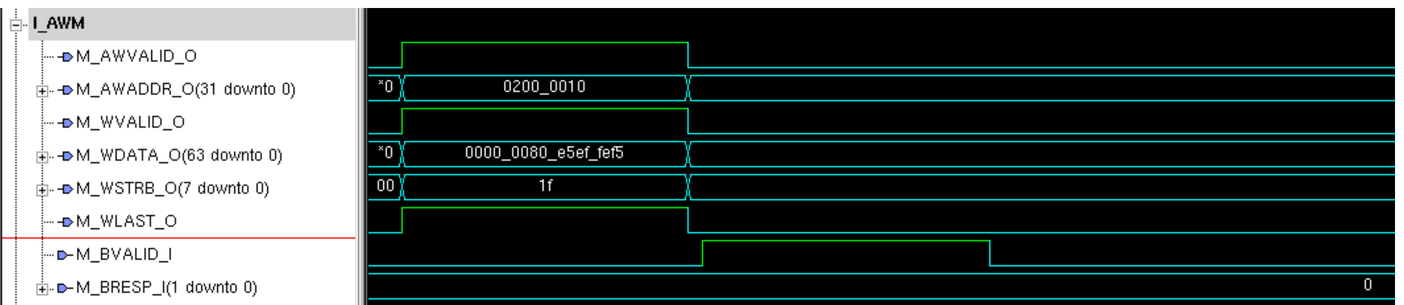
Filtering unit interface waveform view is in the screenshot below.



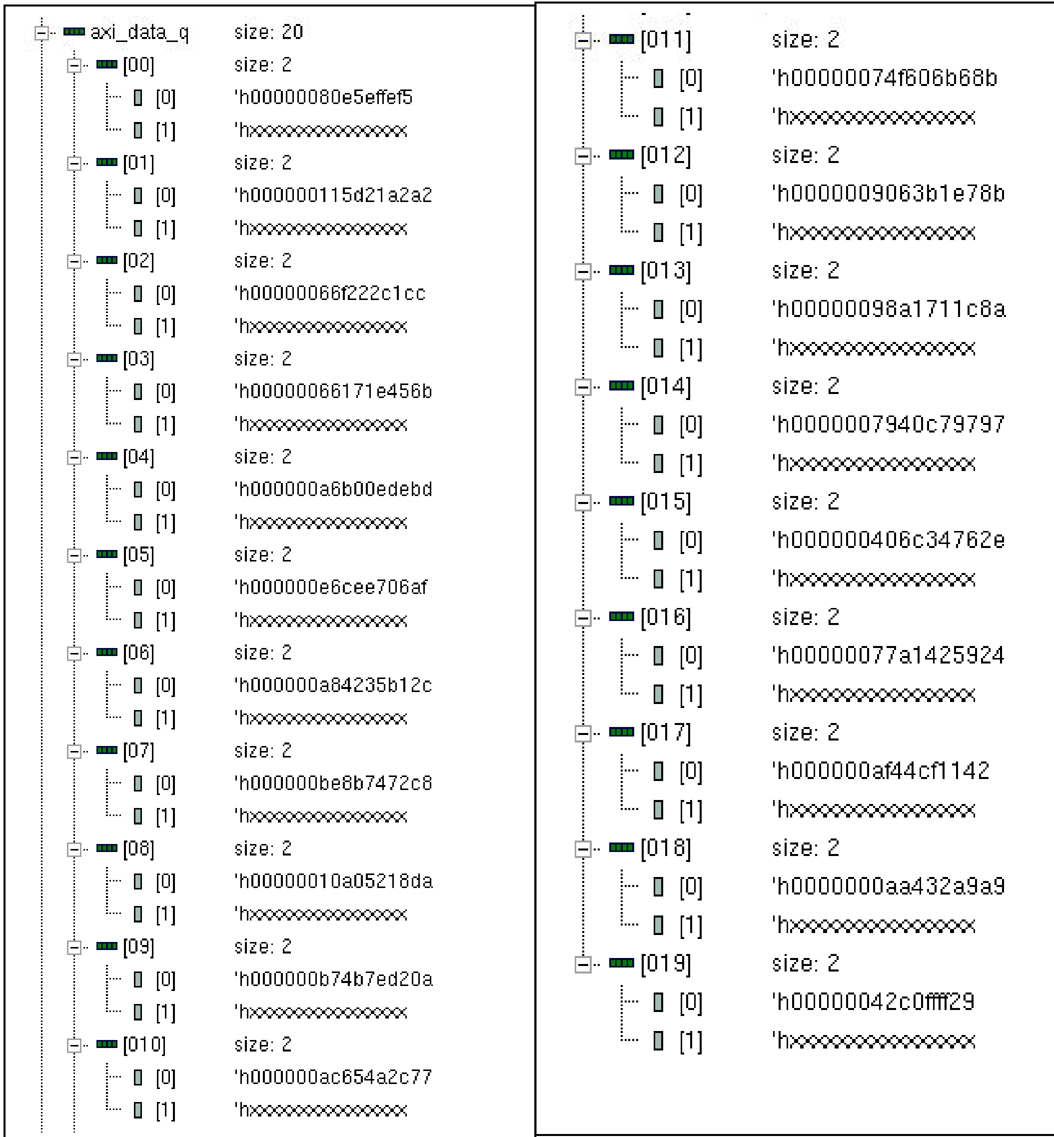
Filtering unit items collected in the reference queue and output queue in the scoreboard is in the screenshot below.



AXI writer module interface waveform view shown in the screenshot below.

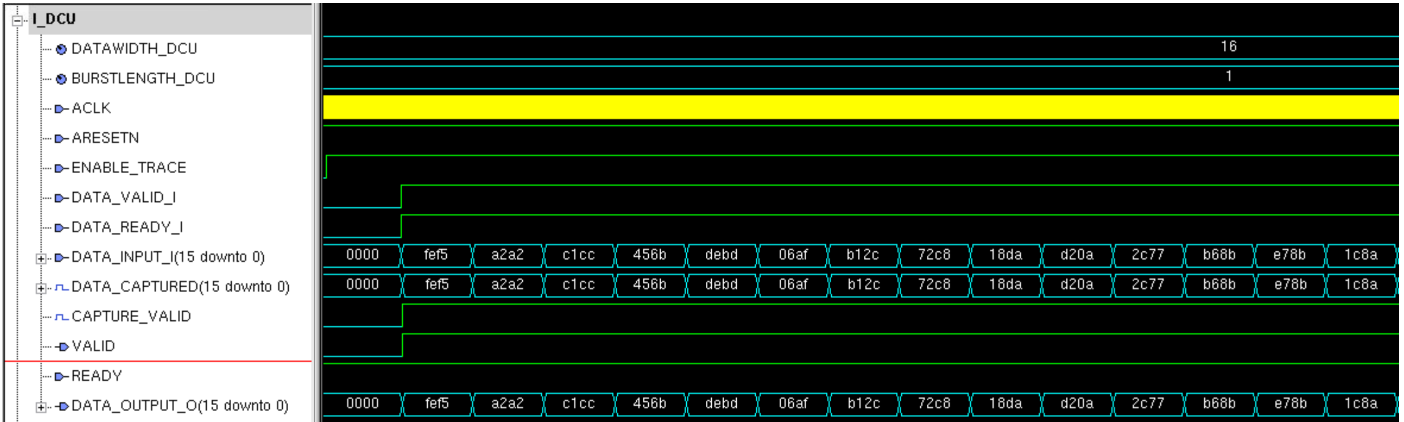


AXI items collected in a queue in the scoreboard is below.

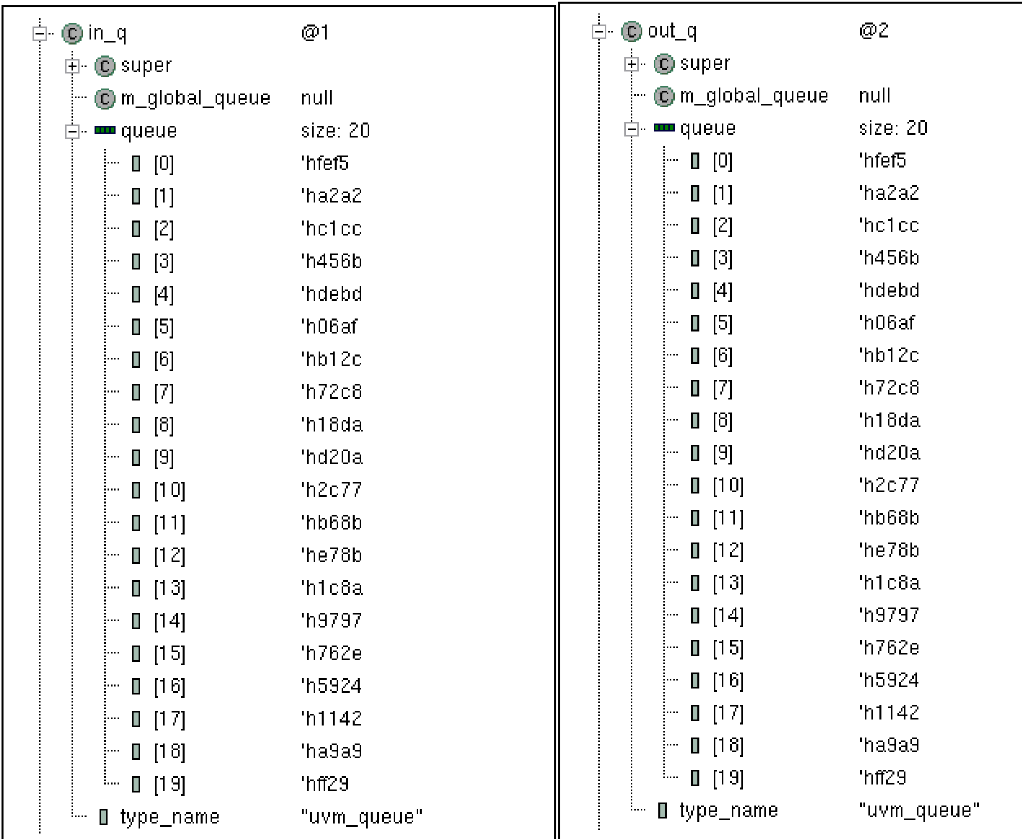

axi_data_q size: 20
[00] size: 2
[0] 'h00000080e5effef5
[1] 'hxxxxxxxxxxxxxxxx
[01] size: 2
[0] 'h000000115d21a2a2
[1] 'hxxxxxxxxxxxxxxxx
[02] size: 2
[0] 'h00000066f222c1cc
[1] 'hxxxxxxxxxxxxxxxx
[03] size: 2
[0] 'h00000066171e456b
[1] 'hxxxxxxxxxxxxxxxx
[04] size: 2
[0] 'h000000a6b00edebd
[1] 'hxxxxxxxxxxxxxxxx
[05] size: 2
[0] 'h000000e6cee706af
[1] 'hxxxxxxxxxxxxxxxx
[06] size: 2
[0] 'h000000a84235b12c
[1] 'hxxxxxxxxxxxxxxxx
[07] size: 2
[0] 'h000000be8b7472c8
[1] 'hxxxxxxxxxxxxxxxx
[08] size: 2
[0] 'h00000010a05218da
[1] 'hxxxxxxxxxxxxxxxx
[09] size: 2
[0] 'h000000b74b7ed20a
[1] 'hxxxxxxxxxxxxxxxx
[010] size: 2
[0] 'h000000ac654a2c77
[1] 'hxxxxxxxxxxxxxxxx
[011] size: 2
[0] 'h00000074f606b68b
[1] 'hxxxxxxxxxxxxxxxx
[012] size: 2
[0] 'h0000009063b1e78b
[1] 'hxxxxxxxxxxxxxxxx
[013] size: 2
[0] 'h00000098a1711c8a
[1] 'hxxxxxxxxxxxxxxxx
[014] size: 2
[0] 'h0000007940c79797
[1] 'hxxxxxxxxxxxxxxxx
[015] size: 2
[0] 'h000000406c34762e
[1] 'hxxxxxxxxxxxxxxxx
[016] size: 2
[0] 'h00000077a1425924
[1] 'hxxxxxxxxxxxxxxxx
[017] size: 2
[0] 'h000000af44cf1142
[1] 'hxxxxxxxxxxxxxxxx
[018] size: 2
[0] 'h0000000aa432a9a9
[1] 'hxxxxxxxxxxxxxxxx
[019] size: 2
[0] 'h00000042c0ffff29
[1] 'hxxxxxxxxxxxxxxxx

Appendix 3: Simulation results for the bus monitor on EM credit interface

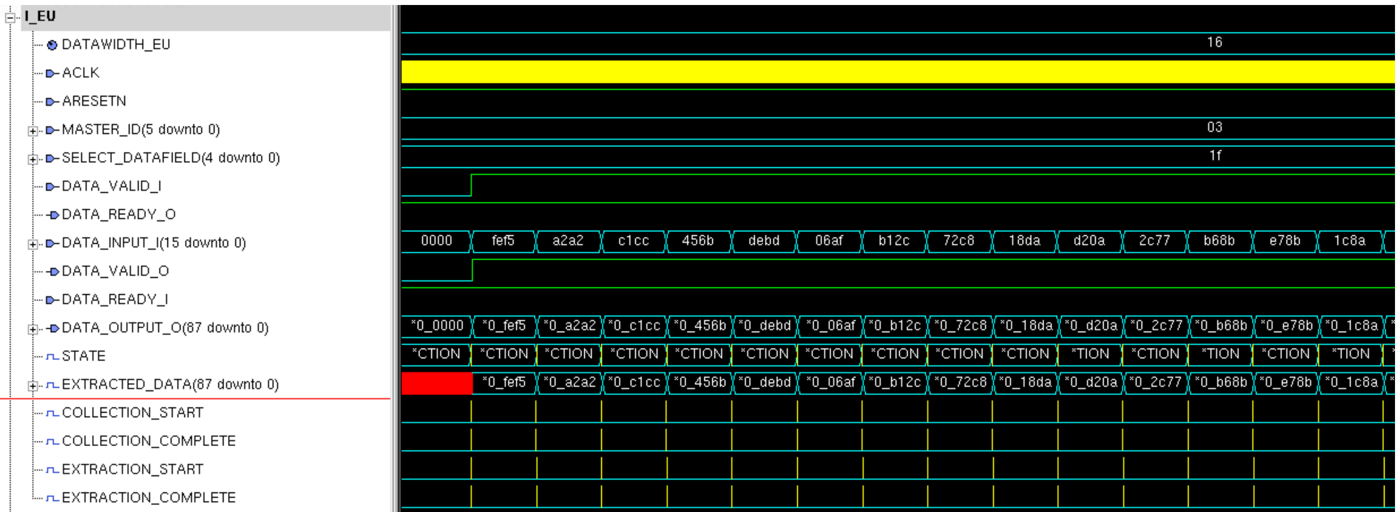
This section includes the simulation result for the bus monitor on the EM credit interface and bus width is 16 bits. The following screenshot shows the waveform view of the **data capture unit interface**.



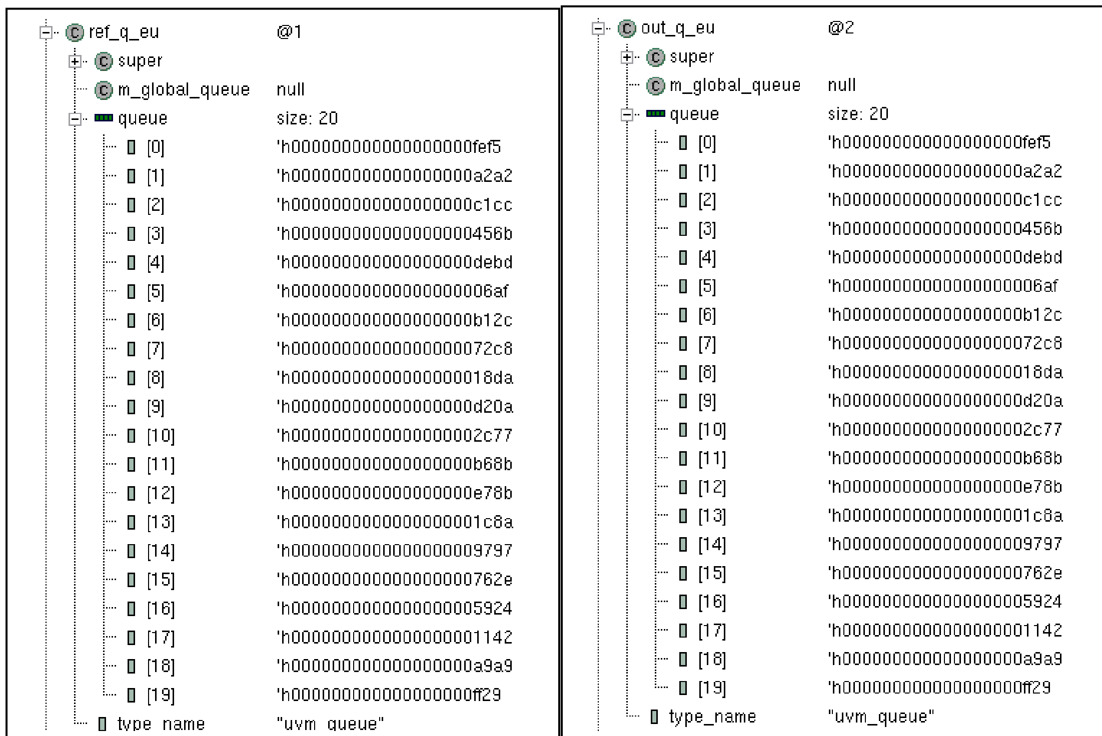
Items collected in the reference queue and output queue for the data capture unit is below.



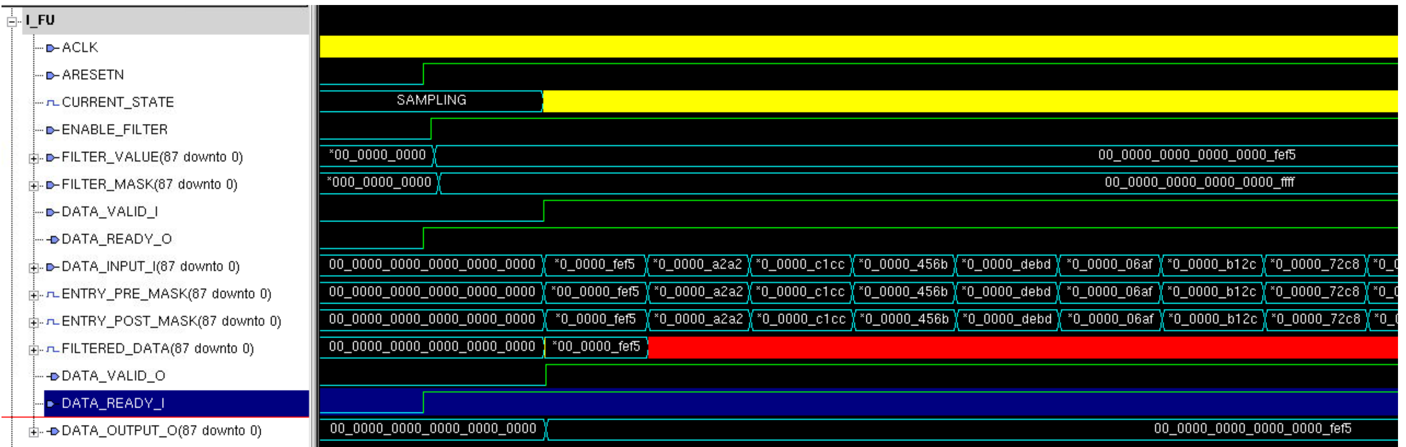
Extraction unit interface waveform view is in the screenshot below.



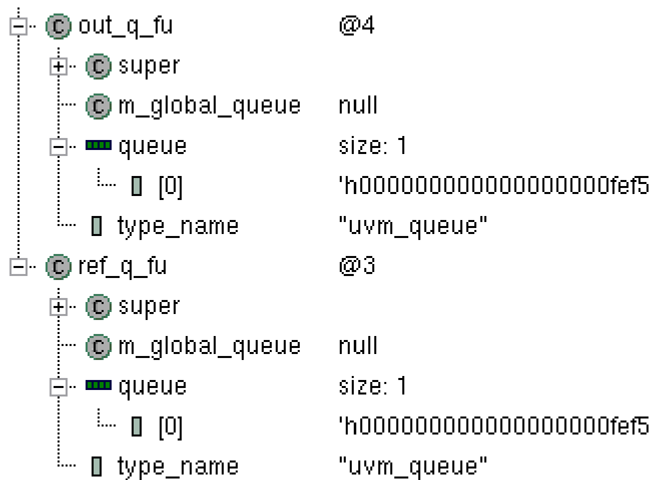
Reference items and output items collected in a queue in the scoreboard for comparison for the extraction unit is shown below.



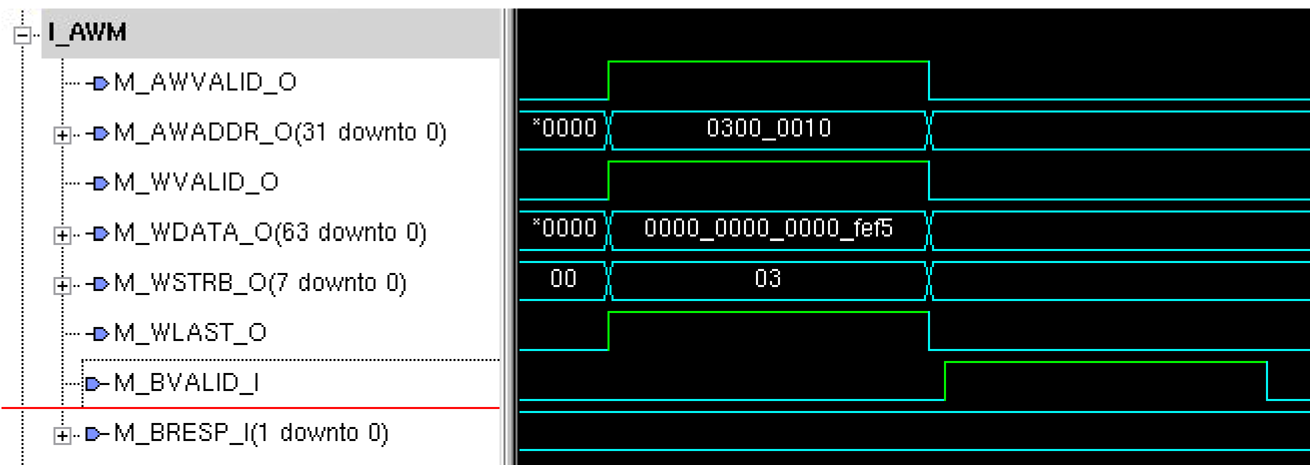
Filtering unit interface waveform view is shown below.



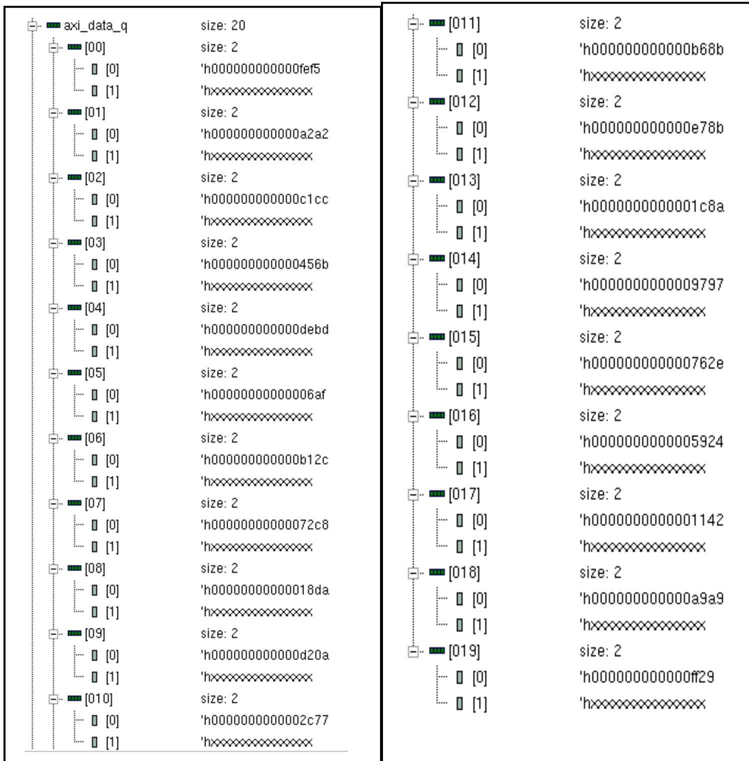
Filtering unit reference item and output item collected in a queue in the scoreboard is below.



AXI Writer module interface waveform is below.



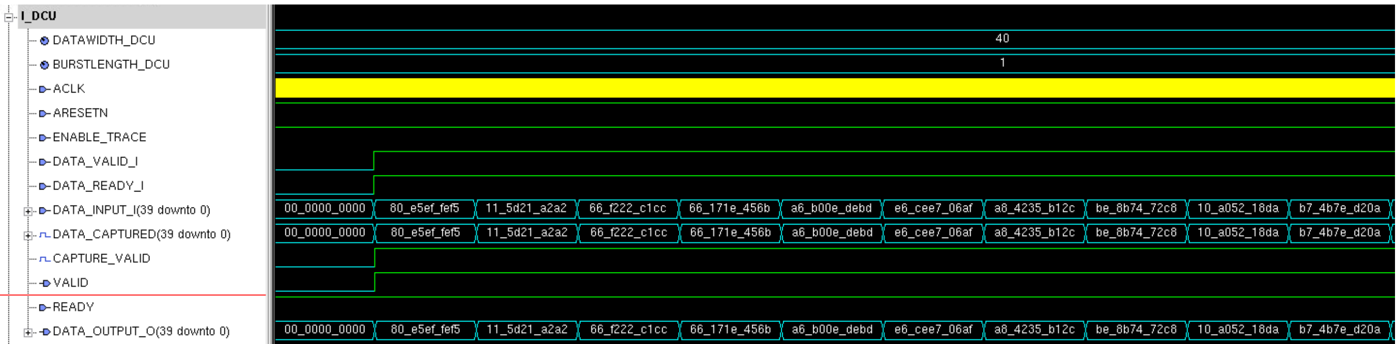
AXI items collected in a queue is below.



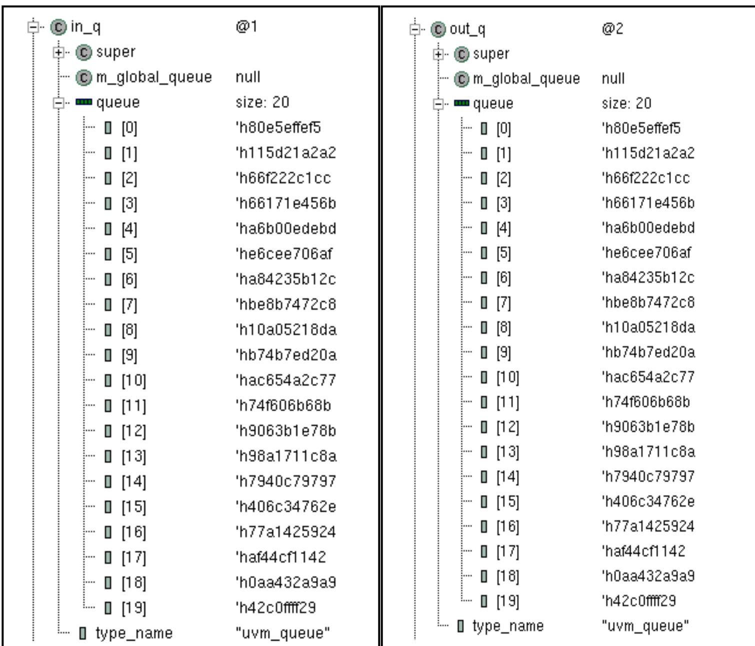
Appendix 4: Simulation results for the bus monitor on BM credit interface

This section includes the simulation results for the bus monitor connected on the BM credit interface which has a data bus width of 40 bits.

The following screenshot shows the data capture unit interface waveform view.



Reference item and output item for data capture unit collected in a queue in the scoreboard for comparison is below.



Extraction unit interface waveform is below.

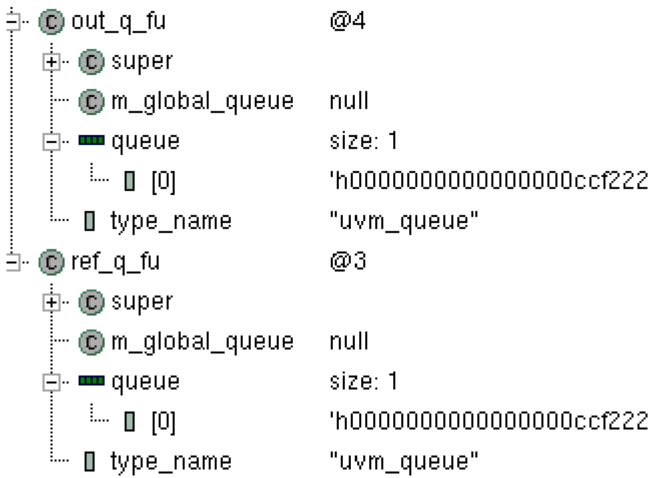
Items collected in the reference queue and the output queue for the comparison is shown below.

ref_q_eu	@1
super	
m_global_queue	null
queue	size: 20
[0]	'h0000000000000080fef5e5ef
[1]	'h00000000000011a2a25d21
[2]	'h00000000000066c1ccf222
[3]	'h00000000000066456b171e
[4]	'h000000000000a6debdb00e
[5]	'h000000000000e606afcee7
[6]	'h000000000000a8b12c4235
[7]	'h000000000000be72c88b74
[8]	'h0000000000001018daa052
[9]	'h000000000000b7d20a4b7e
[10]	'h000000000000ac2c77654a
[11]	'h00000000000074b68bf606
[12]	'h00000000000090e78b63b1
[13]	'h000000000000981c8aa171
[14]	'h00000000000079979740c7
[15]	'h00000000000040762e6c34
[16]	'h000000000000775924a142
[17]	'h000000000000af114244cf
[18]	'h0000000000000aa9a9a432
[19]	'h00000000000042ff29c0ff
type_name	"uvm_queue"

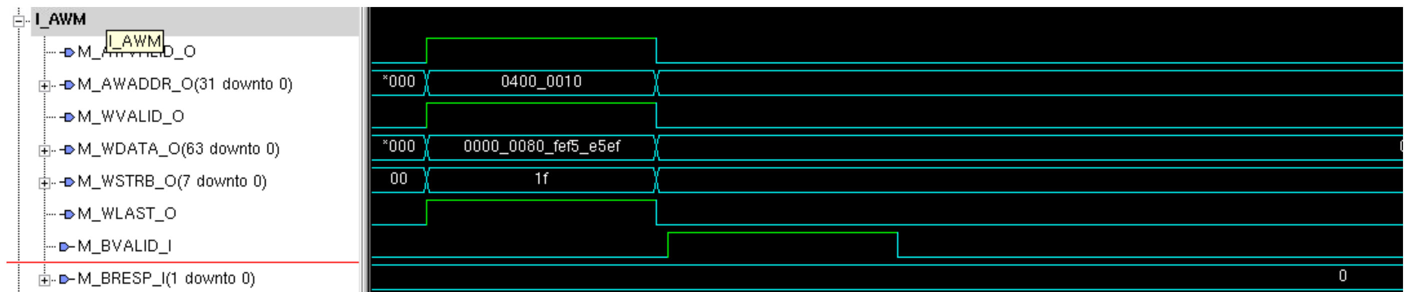
out_q_eu	@2
super	
m_global_queue	null
queue	size: 20
[0]	'h0000000000000080fef5e5ef
[1]	'h00000000000011a2a25d21
[2]	'h00000000000066c1ccf222
[3]	'h00000000000066456b171e
[4]	'h000000000000a6debdb00e
[5]	'h000000000000e606afcee7
[6]	'h000000000000a8b12c4235
[7]	'h000000000000be72c88b74
[8]	'h0000000000001018daa052
[9]	'h000000000000b7d20a4b7e
[10]	'h000000000000ac2c77654a
[11]	'h00000000000074b68bf606
[12]	'h00000000000090e78b63b1
[13]	'h000000000000981c8aa171
[14]	'h00000000000079979740c7
[15]	'h00000000000040762e6c34
[16]	'h000000000000775924a142
[17]	'h000000000000af114244cf
[18]	'h0000000000000aa9a9a432
[19]	'h00000000000042ff29c0ff
type_name	"uvm_queue"

Filtering unit interface waveform view is below.

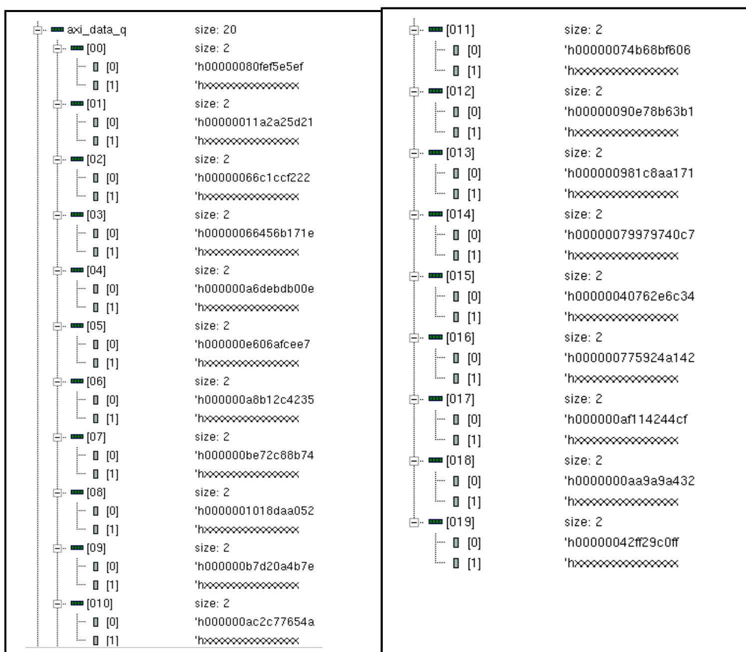
Filtering unit reference item and output item collected in a queue in the scoreboard is shown below.



AXI writer module interface waveform view is shown below.

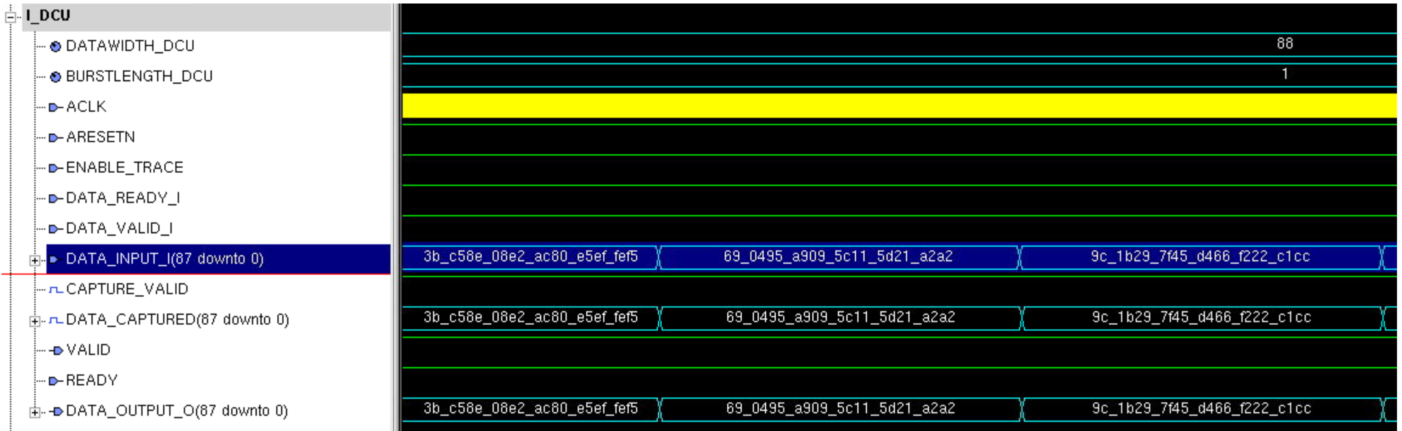


AXI items collected in a queue in the scoreboard is below.

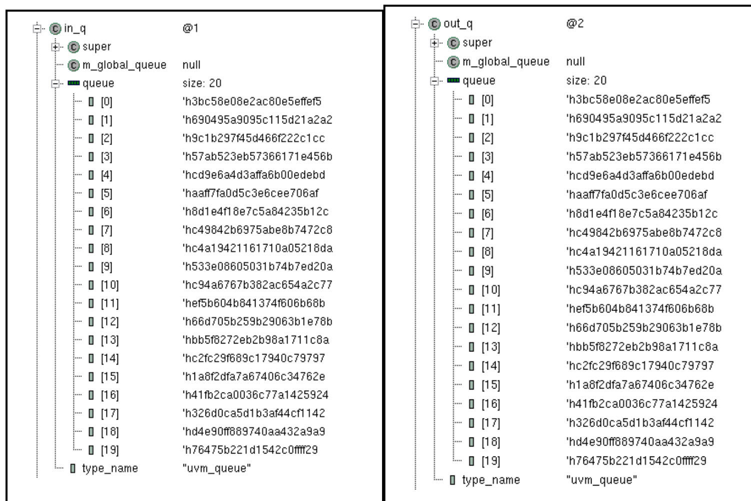


Appendix 5: Simulation results for the bus monitor on allocation interface

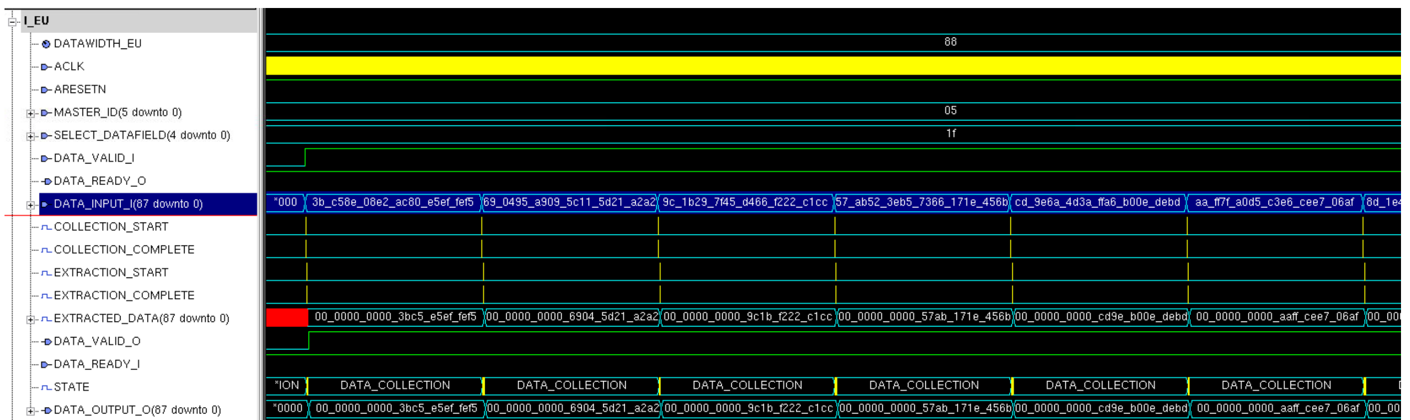
This section includes the simulation result for the bus monitor connected on the allocation interface between BM and BM EMA. The data bus width in this interface is 88 bits. The following screenshot shows the waveform view of the data capture unit interface.



Reference items and output items collected in a queue in the scoreboard for comparison is below.



Below is the waveform view of the extraction unit interface.

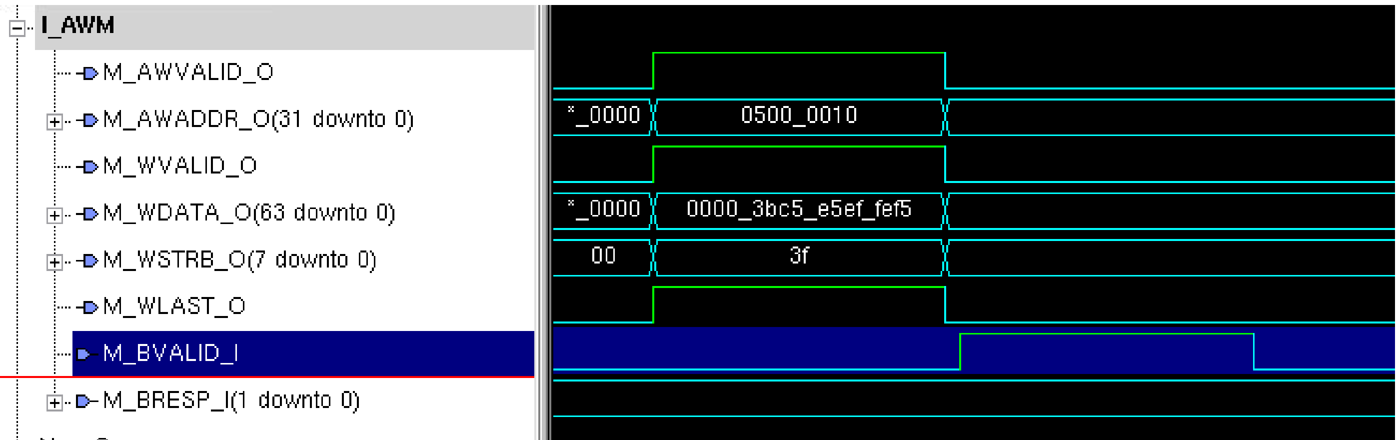


Filtering unit interface waveform is shown below.

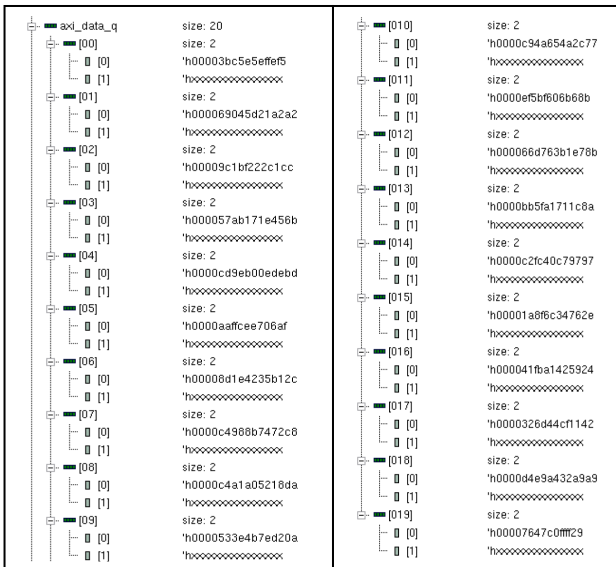
Below is the screenshot view of the filtered item collected in reference queue and output queue.

out_q_fu	@6
super	
m_global_queue	null
queue	size: 1
[0]	'h0000000000000005d21a2a2
type_name	"uvm_queue"
ref_q_fu	@5
super	
m_global_queue	null
queue	size: 1
[0]	'h0000000000000005d21a2a2
type_name	"uvm_queue"

AXI writer module interface waveform view is shown below.

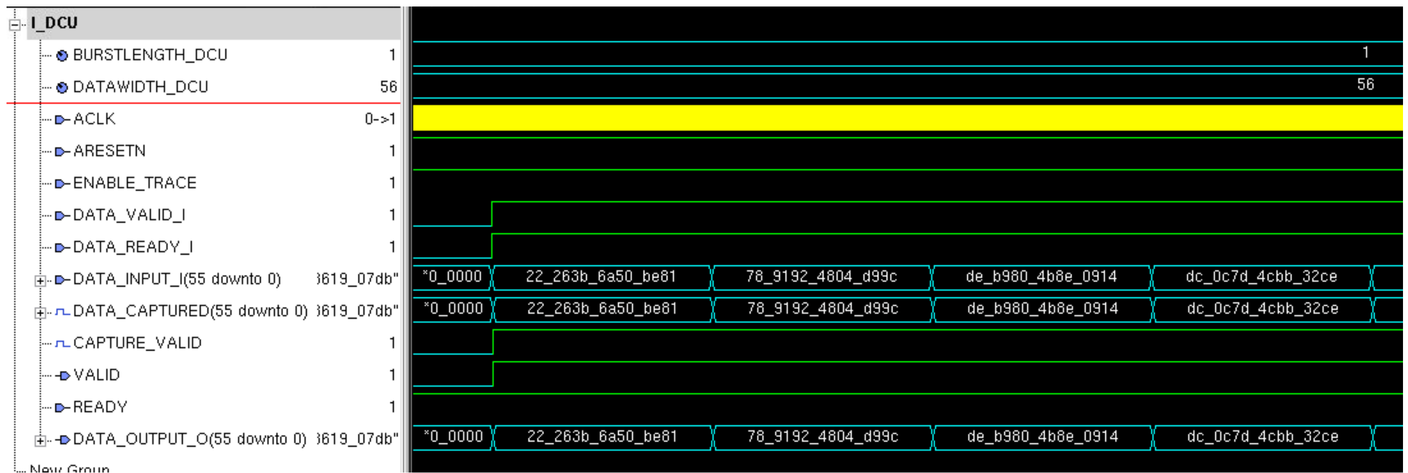


AXI output collected in a queue in the scoreboard is below.

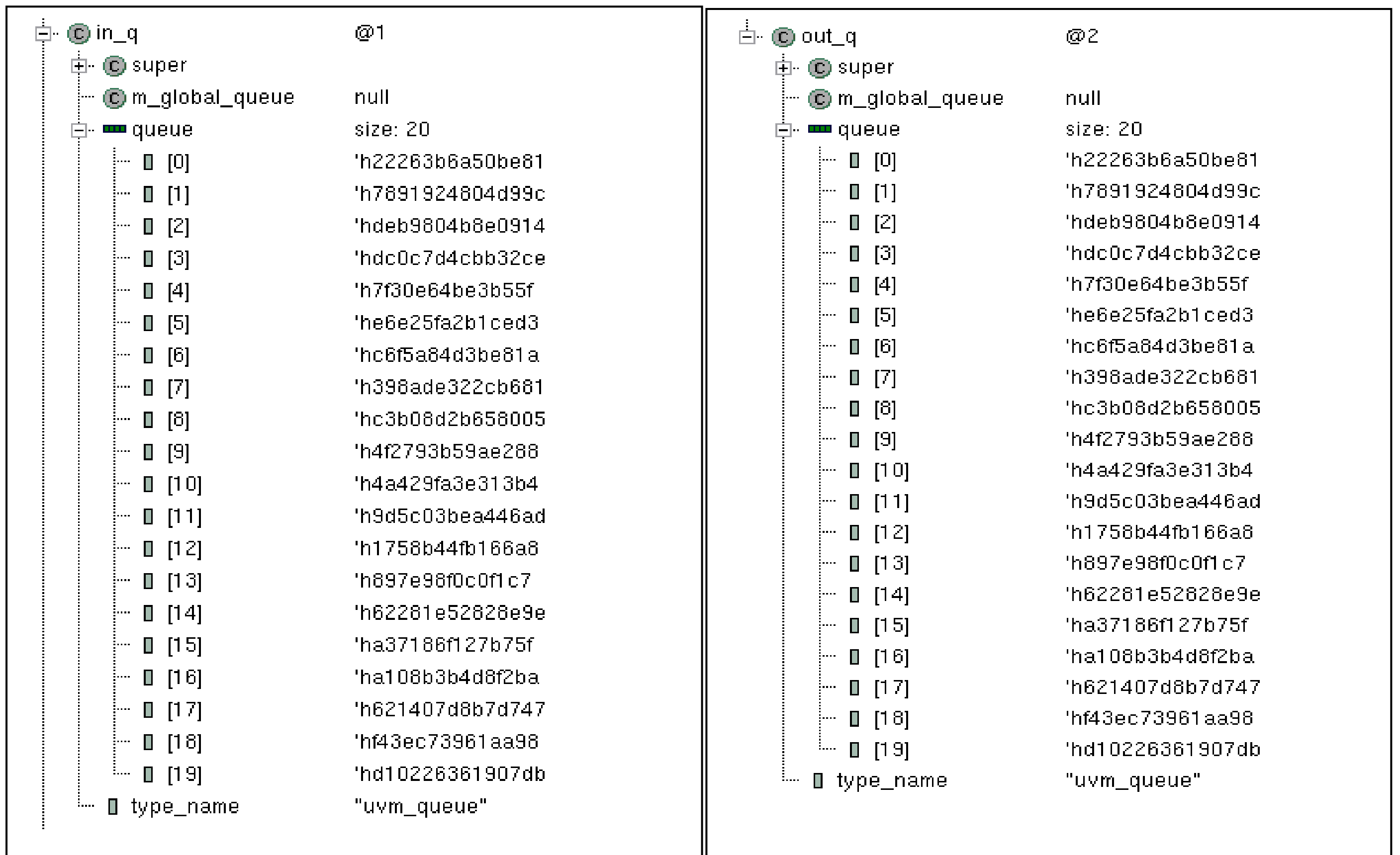


Appendix 6: Simulation results for the bus monitor on the command interface

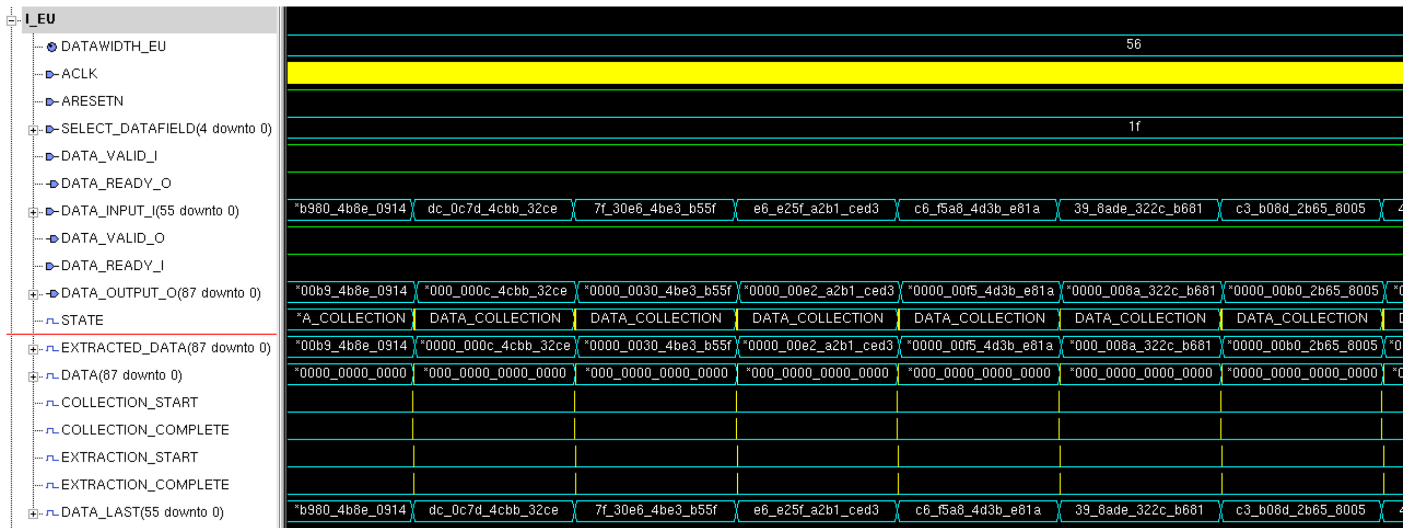
This section includes the simulation result for the bus monitor with bus width of 56 bit. The following screenshot shows the waveform view of the **data capture unit interface**.



The following screenshot shows the data collected in the reference queue (in_q) and output queue(out_q) in the UVM Scoreboard for the **data capture unit**.



The following screenshot shows the waveform view of the **extraction unit interface**.

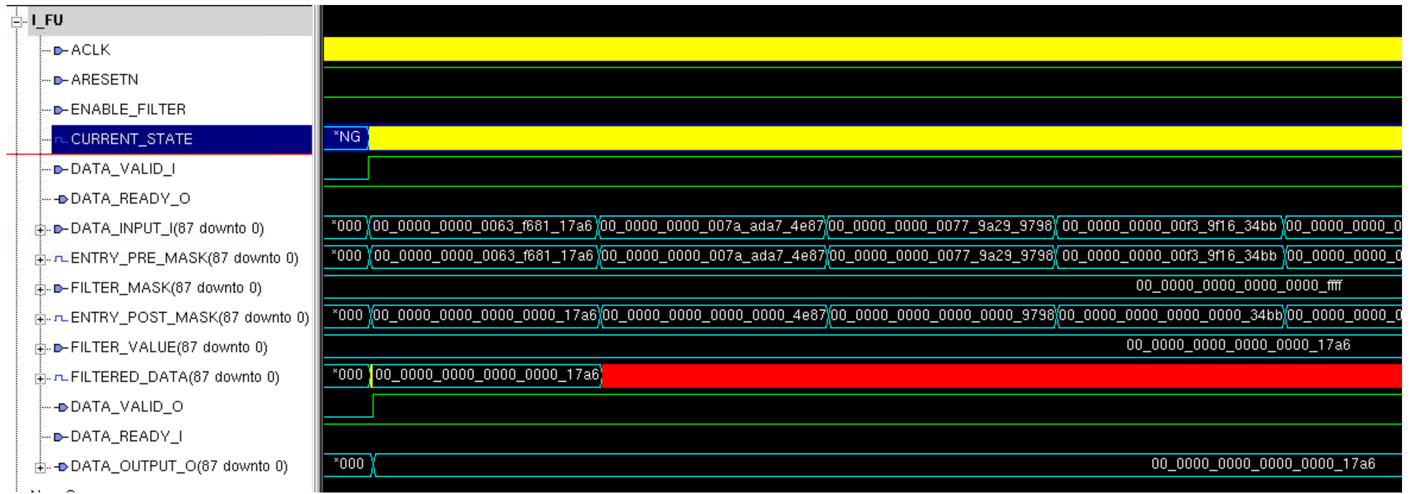


The screenshot below shows the data collected in the reference queue (ref_q_eu) and output queue (out_q_eu) for **extraction unit**.

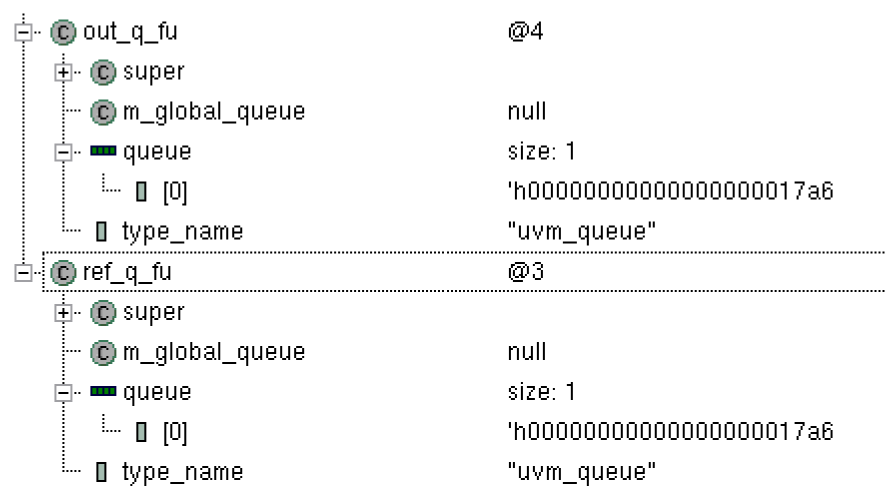
ref_q_eu	@1
super	
m_global_queue	null
queue	size: 20
[0]	'h000000000000266a50be81
[1]	'h00000000000914804d99c
[2]	'h000000000000b94b8e0914
[3]	'h0000000000000c4cbb32ce
[4]	'h000000000000304be3b55f
[5]	'h000000000000e2a2b1ced3
[6]	'h000000000000f54d3be81a
[7]	'h0000000000008a322cb681
[8]	'h000000000000b02b658005
[9]	'h00000000000027b59ae288
[10]	'h0000000000042a3e313b4
[11]	'h000000000005cbea446ad
[12]	'h0000000000000584fb166a8
[13]	'h000000000007ef0c0f1c7
[14]	'h0000000000002852828e9e
[15]	'h00000000000071f127b75f
[16]	'h00000000000008b4d8f2ba
[17]	'h000000000000014d8b7d747
[18]	'h0000000000003e3961aa98
[19]	'h0000000000002361907db
type_name	"uvm_queue"

out_q_eu	@2
super	
m_global_queue	null
queue	size: 20
[0]	'h000000000000266a50be81
[1]	'h00000000000914804d99c
[2]	'h000000000000b94b8e0914
[3]	'h0000000000000c4cbb32ce
[4]	'h000000000000304be3b55f
[5]	'h000000000000e2a2b1ced3
[6]	'h000000000000f54d3be81a
[7]	'h0000000000008a322cb681
[8]	'h000000000000b02b658005
[9]	'h00000000000027b59ae288
[10]	'h0000000000042a3e313b4
[11]	'h000000000005cbea446ad
[12]	'h0000000000000584fb166a8
[13]	'h000000000007ef0c0f1c7
[14]	'h0000000000002852828e9e
[15]	'h00000000000071f127b75f
[16]	'h00000000000008b4d8f2ba
[17]	'h000000000000014d8b7d747
[18]	'h0000000000003e3961aa98
[19]	'h0000000000002361907db
type_name	"uvm_queue"

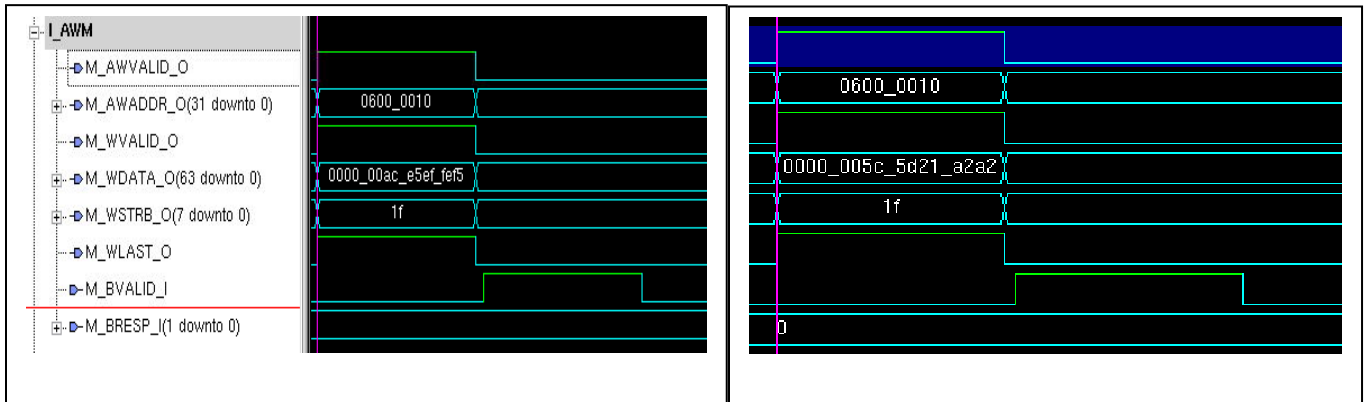
The following screenshot shows the waveform view of the **filtering unit interface**.



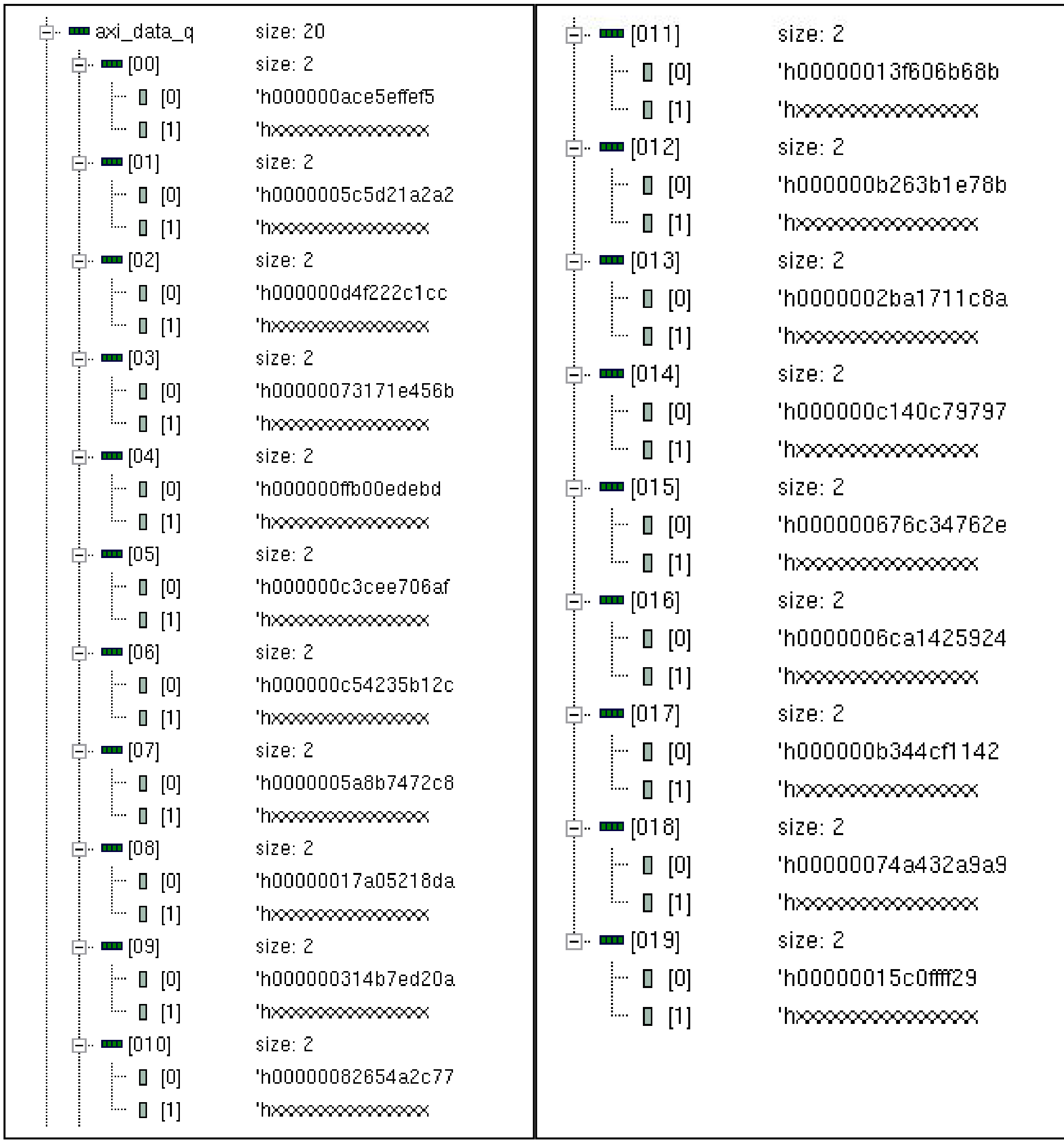
The following screenshot shows the item collected in reference queue and the output queue in the scoreboard for the **filtering unit**. The filter value in filter value register is 17a6 and hence only the data field with value equal to 17a6 is filtered out.



The following screenshot shows the waveform view of the AXI writer module where two transactions are shown.



The following screenshot shows the AXI items collected in a queue in uvm scoreboard. The address signal value in this case is as shown in the screenshot above is 0600_0010.



axi_data_q size: 20
[00] size: 2
[0] 'h000000ace5effef5
[1] 'hxxxxxxxxxxxxxxxx
[01] size: 2
[0] 'h0000005c5d21a2a2
[1] 'hxxxxxxxxxxxxxxxx
[02] size: 2
[0] 'h000000d4f222c1cc
[1] 'hxxxxxxxxxxxxxxxx
[03] size: 2
[0] 'h00000073171e456b
[1] 'hxxxxxxxxxxxxxxxx
[04] size: 2
[0] 'h000000ffb00edebd
[1] 'hxxxxxxxxxxxxxxxx
[05] size: 2
[0] 'h000000c3cee706af
[1] 'hxxxxxxxxxxxxxxxx
[06] size: 2
[0] 'h000000c54235b12c
[1] 'hxxxxxxxxxxxxxxxx
[07] size: 2
[0] 'h0000005a8b7472c8
[1] 'hxxxxxxxxxxxxxxxx
[08] size: 2
[0] 'h00000017a05218da
[1] 'hxxxxxxxxxxxxxxxx
[09] size: 2
[0] 'h000000314b7ed20a
[1] 'hxxxxxxxxxxxxxxxx
[010] size: 2
[0] 'h00000082654a2c77
[1] 'hxxxxxxxxxxxxxxxx
[011] size: 2
[0] 'h00000013f606b68b
[1] 'hxxxxxxxxxxxxxxxx
[012] size: 2
[0] 'h000000b263b1e78b
[1] 'hxxxxxxxxxxxxxxxx
[013] size: 2
[0] 'h0000002ba1711c8a
[1] 'hxxxxxxxxxxxxxxxx
[014] size: 2
[0] 'h000000c140c79797
[1] 'hxxxxxxxxxxxxxxxx
[015] size: 2
[0] 'h000000676c34762e
[1] 'hxxxxxxxxxxxxxxxx
[016] size: 2
[0] 'h0000006ca1425924
[1] 'hxxxxxxxxxxxxxxxx
[017] size: 2
[0] 'h000000b344cf1142
[1] 'hxxxxxxxxxxxxxxxx
[018] size: 2
[0] 'h00000074a432a9a9
[1] 'hxxxxxxxxxxxxxxxx
[019] size: 2
[0] 'h00000015c0ffff29
[1] 'hxxxxxxxxxxxxxxxx