Verifying C++ Firmware Sequences In UVM Environment

Ashwini Holla

Advanced Micro Devices, Inc.
Markham, Canada

www.amd.com

ABSTRACT

In AMD designs, a significant number of power management features are being implemented in firmware rather than hardware. Firmware sequences are used to define and control hardware behaviour while stepping through different power and performance states. As a result of such interaction with hardware, firmware qualification in VCS is becoming important. This presents a unique verification problem, one of devising a smart approach to verifying C++ firmware sequences in a UVM test environment with RTL design blocks. The standard UVM scoreboard methodology only increased the complexity of verification while not providing the necessary coverage. So a hybrid method of test-based scoreboard combined with transaction collection using the UVM infrastructure was adopted. This paper describes how this method was used to check correctness of firmware programming and sequence while also ensuring proper functioning of hardware. It has enabled effective qualification of hardware behaviour with firmware early in the design process.

Table of Contents

Abbreviations:

ASIC - Application Specific Integrated Circuit
SOC - System on Chip
PM - Power Management
Mst - Master
Slv - Slave
Intf - Interface
Msg - Message
SBI - Side Band Interface

Introduction

Why is firmware becoming so important?

There is an ever increasing demand for better performance in our designs and very tight timelines for ASIC delivery. This requires that we ensure minimal hardware changes from one generation to another. At the same time, there is a need for flexibility and continuous improvements. Implementing all features in hardware cannot guarantee that we can meet these conflicting goals – minimal hardware changes while ensuring significant improvements. Moreover, the current complexity of designs results in a huge verification effort even if the change in hardware is minimal.

This has necessitated architectural changes (this is out of scope of the current discussion) while shifting focus on firmware to implement various features. Several power management features are being handled by firmware in our designs. This has enabled tremendous flexibility and has provided adequate room for improvement, while keeping hardware changes within acceptable limits. Bugs which would have resulted in either cancellation of a feature or even worse hardware re-spins are now fix-able even after the silicon is back in the lab.

It's not all win with firmware

However, this flexibility with firmware comes with some penalty. Firstly, there is increased latency when firmware implements a feature rather than the hardware. Secondly, given the confidence that firmware can be changed even after the chip has taped out, silicon bring-up times are steadily increasing due to inadequate firmware qualification in the pre-silicon stages. These factors have brought firmware verification into spotlight. Moreover, even as our designs are evolving to support plug and play architecture, legacy interfaces continue to exist, and latency critical paths are implemented with custom sideband interfaces. Firmware sequences that exercise such datapaths might expose hardware issues. So, verifying hardware-firmware interaction early on in the developmental process becomes very important.

What can you expect from the paper?

In this paper, I would like to discuss how hardware-firmware interaction was verified with VCS in our designs. The preliminary step was to identify how much of the firmware needs to be verified with VCS given its dynamic nature. VCS simulations are time consuming, and ensuring that only the essential aspects are covered is important to be able to meet project timelines. With growing complexity of firmware, we felt the need to make use of non-VCS platforms that are faster than VCS especially for quality checking of firmware, and there is a brief discussion on

the same. For VCS qualification, traditional UVM scoreboards were experimented with but they only added to complexity of verification. Instead, a test-based scoreboard approach was devised and this has proved to be very effective in firmware qualification. There is a detailed discussion on this revised scoreboard methodology in the paper.

1. Brief Overview of the Architecture

This section provides a brief overview of the design architecture to better understand the verification requirements.

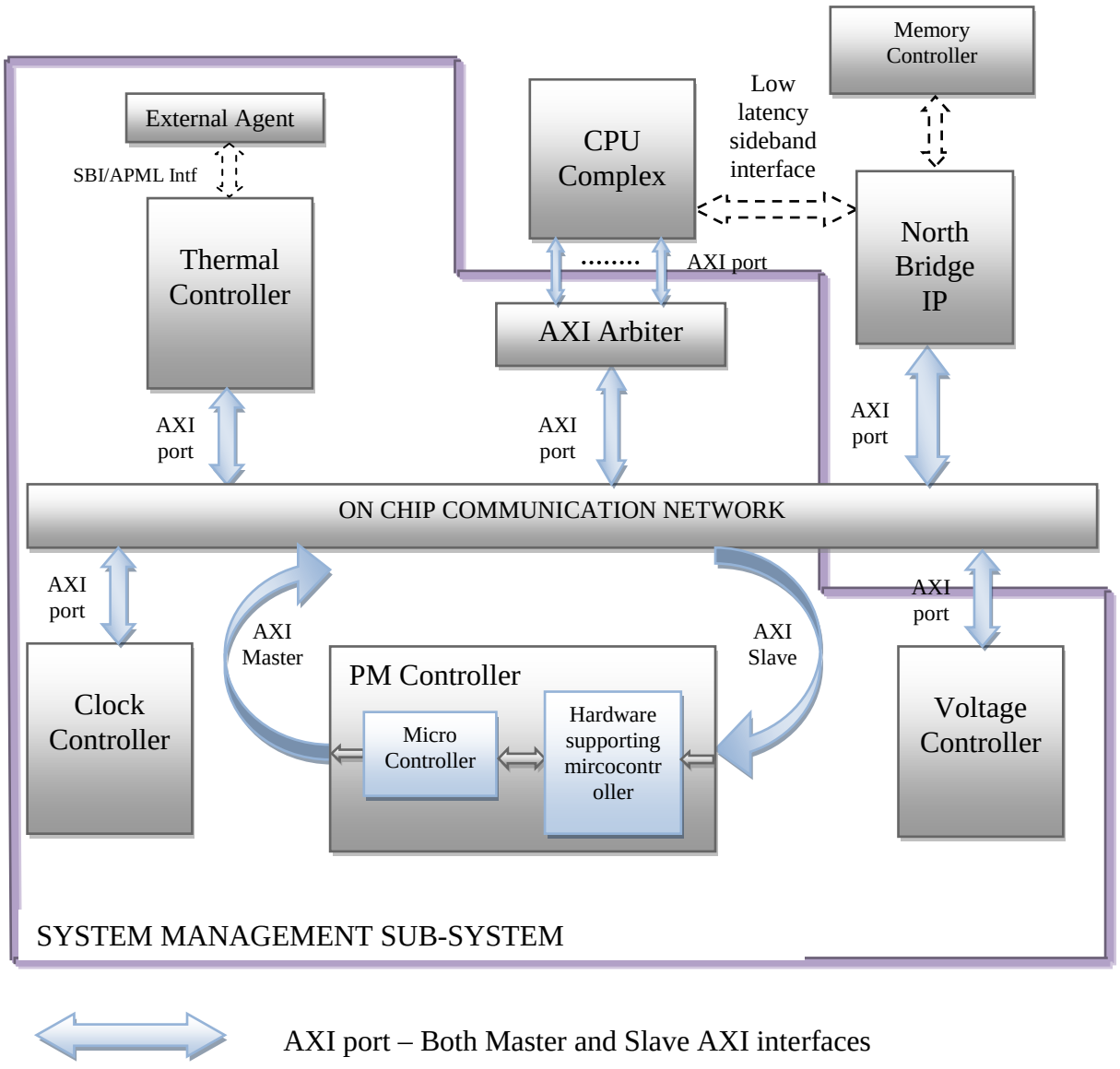

Figure 1. Overview of design architecture.

At AMD, our designs are evolving significantly. A new architecture has been devised that allows plug and play capability to interface various sub-blocks into the system. All sub-blocks communicate through an on-chip communication network using the AXI standard. Custom sideband interfaces for communication between sub-blocks are being deprecated steadily. This approach promotes scalability of the design to cater to varying customer requirements. Moreover, it minimises the verification needs at the sub-system (system consisting of several sub-blocks) and SOC level.

Each sub-block including the on-chip communication network can be verified standalone for functionality. At the sub-system and SOC level, functional testing is minimal and must eventually tend to nil. The primary goal of SOC/sub-system verification in such architecture is to ensure proper connectivity.

Bootup and power management sequences are also required to be verified at the sub-system and SOC level because their action determines proper functioning of the system as a whole. Moreover, as the current architecture is still in progressive development, legacy interfaces continue to exist between sub-blocks. Latency critical communication for power management is implemented through custom sideband interfaces instead of on-chip communication channel. These factors make the verification of power management firmware at the sub-system level a necessity and this is the primary focus of the discussions in this paper.

2. Power Management Firmware

2.1 Overview

Power management firmware is a collection of C++ functions, also referred to as interrupt service routines. These sequences execute on a microcontroller that resides in an IP called PM controller. Detailed description of the hardware around the micro-controller is in Appendix A - *Hardware for Firmware Execution*. This makes firmware an equivalent of a sub-block that is capable of interacting with the rest of the system. This is the primary reason why VCS verification of firmware is necessary.

Firmware sequences for power management control the power states in the design at any given point in time. They define voltage and frequency for all sub-blocks based on inputs like power budget, temperature, activity monitors, etc. There are two types of firmware sequences.

1. Ones that execute periodically:
   The periodic PM sequences ensure that the performance state (determined by voltage and frequency) of all sub-blocks is at an optimal level. This ascertains that current consumption of the chip does not exceed acceptable limits of at any given point in time.

These PM sequences read activity monitors and temperature information from time to time and compute the ideal power state of the chip.

2. Ones that execute on request:
   An external agent like OS or a sub-block can request firmware to execute a PM sequence. Typical examples of OS requests are changing the power state of a CPU core, enabling or disabling threads in CPU core etc. Communication from an external agent through SBI interface for accessing CPU information etc. also falls under this category. Then there are package-level features like C6 and CState Boost which are handled by North Bridge IP (It has low-latency sideband interface with CPU cores to track their power states). When conditions for C6 or CState Boost are satisfied, it issues request to firmware to execute the corresponding sequence.

Further detail on the two types of firmware sequences is available in Appendix B – *Role of Firmware in Power Management*

2.2 Firmware Datapath

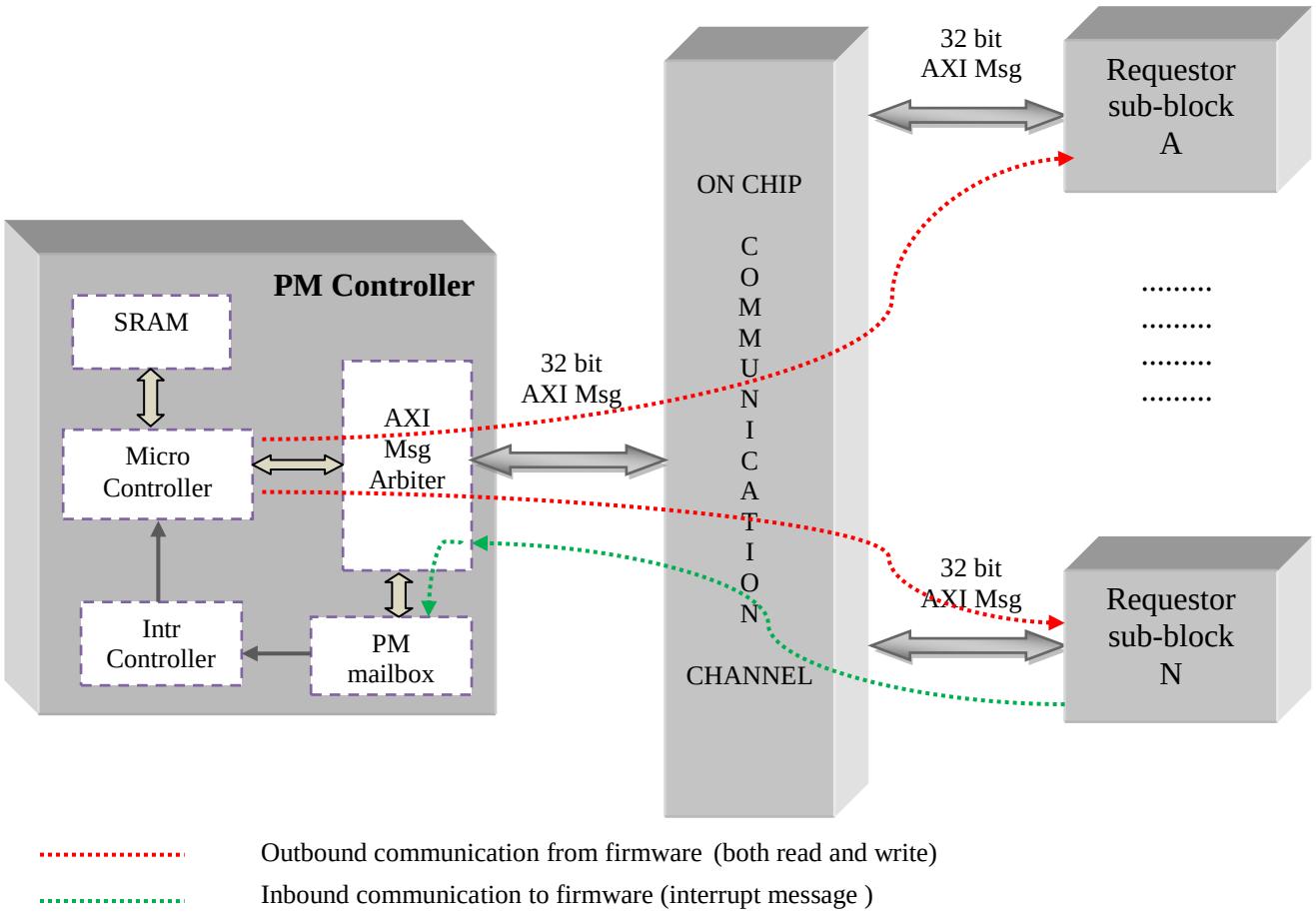

Figure 2. Firmware datapath.

Figure 2 depicts the inbound and outbound communcation of firmware. Communication between all sub-blocks including PM controller is through the on-chip communication channel. The messaging protocol used is the AMBA-AXI standard.

Inbound communication:
A requestor sub-block issues an AXI message to the mailbox in PM controller to interrupt the firmware. The mailbox is addressable through on-chip communication channel. The interrupt controller which is closely situated to the mailbox detects an outstanding message and asserts the interrupt line to the microcontroller. This is the inbound communication to firmware from requestor sub-blocks.

Outbound communication:
The following describes the outbound communication from firmware to the various sub-blocks:
Interrupt target address programming - During system initialisation, firmware programs requestor sub-blocks with target address for interrupt messages. This is the address of the PM mailbox as seen by the on-chip communication channel.
Undertsanding the request - When firmware receives an interrupt message from a requestor sub-block, the service routine may involve reading certain control registers in the requestor sub-block. The data extracted through these reads helps firmware understand the request better.
Control register programming – An interrupt service routine will involve control register programming. Such programming affects voltage, frequency, interlock of interrupt logic in requestor sub-block etc. After control registers are programmed, firmware waits for completion of the corresponding sequence by reading the status register and polling for done status.

2.3 Interrupt Messaging

The interrupt message to firmware is a 32 bit AXI transaction from the requestor sub-block as shown in Figure 3.

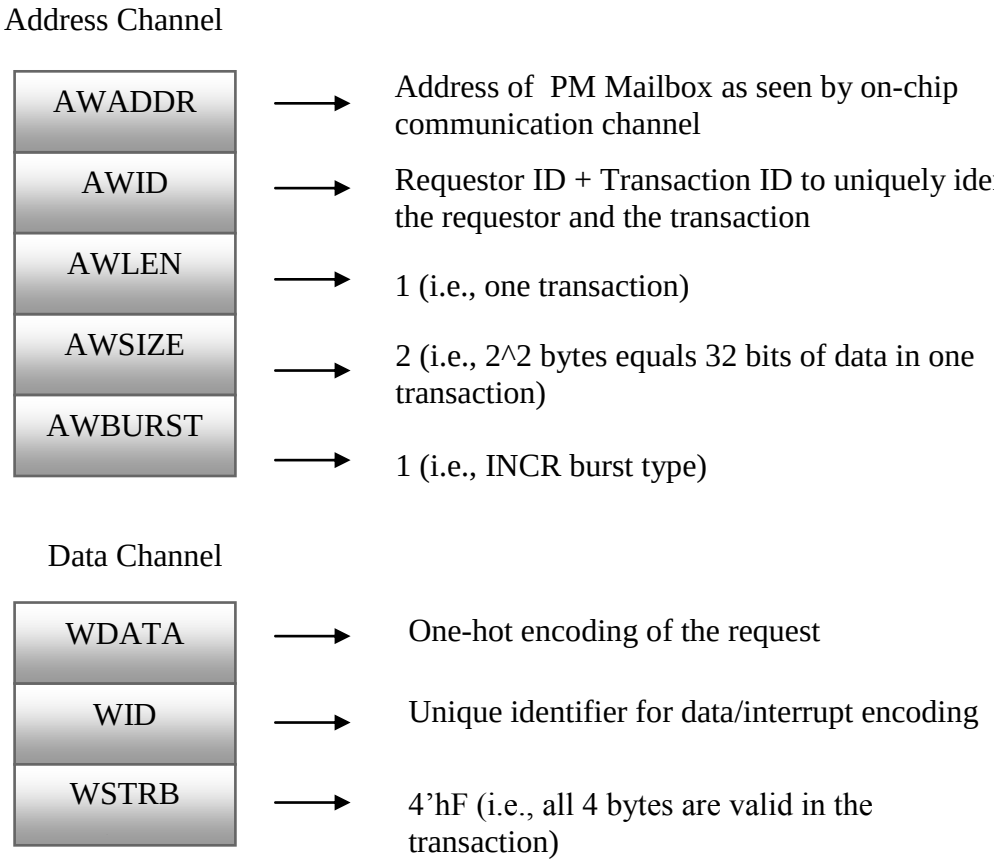

Figure 3. AXI interrupt message scheme.

Here's a description on the type of information available through the AXI interrupt message:

- The requestor is identified through the AXI message ID (AWID in Figure 3). The message ID consists of two parts – transaction ID and requestor ID. The transaction ID uniquely identifies the transactions from the requestor. Requestor ID is the unique identifier for the requesting sub-block on the on-chip communication channel .

- The 32 bit data in the AXI message (WDATA in Figure 3) are used to identify the request. Each interrupt has a unique bit positioning in the 32 bit message. The requestor, while sending this message must properly indicate the one-hot encoding of the interrupt being requested (Figure 4). (Information on the one-hot encoding is exchanged aprior between the firmware developers and RTL designers of the requestor sub-blocks during the process of micro-architecture definition.)

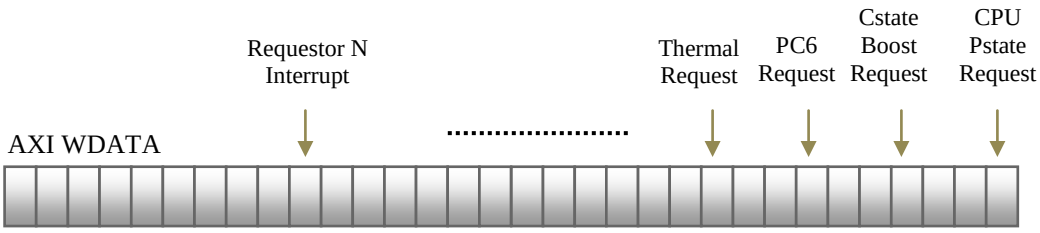

Figure 4. One-hot encoding in interrupt message.

- Firmware decodes the 32-bit AXI message to execute a unique interrupt service routine.

//Code Snippet for ISR mapping

```
switch (axi_wdata){ //axi_wdata is the interrupt message from requestor
                    //captured in PM mailbox
  case 0x1:         cpu_pstate_isr();
  case 0x8:         cstate_boost_isr();
  case 0x40:        pc6_isr();
  case 0x200:       thermal_isr();
  case 0x80_0000:   requestor_N_isr();
  default:          call_error_handler();   //custom function that sets
                                            //an error flag in a
                                            //register
  }
send_ack_to_requestor();
```

The interrupt service routine (one of the functions – cpu_pstate_isr through requestor_N_isr) performs the necessary actions corresponding to the request. Once done, it sends an Ack/Done to the requestor via the on-chip communication channel to indicate completion.

3. Verification of Firmware

3.1 Identifying Requirements

The previous sections have provided an overview of firmware sequences for power management. This section is a discussion on the verification requirements of firmware sequences. As mentioned in Section 3.1 (Power Management Firmware - Overview) there are two types of sequences – ones that execute periodically and ones that execute on request. VCS verification is focussed on the latter type of sequences since they involve most of the hardware interactions of firmware. In fact, they are a super-set of the periodic firmware sequences in that they provide coverage for all types of hardware programming that the firmware would do.

Verification requirements may be broadly classified into two categories:

1. Verifying firmware datapath – both inbound and outbound (refer to Figure 2. Firmware datapath)
2. Verifying correctness of firmware sequences

Verifying Firmware Datapath:

The primary goal of datapath verification is to ensure that interrupts issued by requestor sub-blocks are received and recognised correctly by the PM controller, resulting in execution of proper firmware sequence. That end-to-end firmware communication (i.e., between firmware and sub-blocks via on-chip communication channel) goes through seamlessly is covered. This includes qualification for firmware reads and writes to various sub-blocks that the interrupt service routine involves. Datapath tests also provide coverage for dependencies between interrupts or PM sequences or even the sub-blocks. As a matter of fact, they have helped expose deadlock scenarios in firmware.

Verifying correctness of firmware sequence:

The primary goal here is to ensure if the firmware sequence undertakes the necessary action in response to an interrupt. A PM sequence involves executing specific control register programming in a certain order, violation of which might jeopardize the electrical behaviour of the design. Ensuring that the necessary steps are executed (which are predefined in the firmware micro architecture specification) is the intent of the scoreboard methodology employed here.

Firmware is an interesting verification component in that it involves both hardware and software sequences. Both are interdependent and essential for each other's proper functioning.

Verification Platforms:

There are two platforms for verifying firmware:

1. VCS:
   Qualification of hardware interactions of firmware are best covered with most of the design blocks as RTL, if not all. This requires the use of VCS platform to simulate realistic design scenarios. With VCS, we can run cycle accurate simulations with a PLL generated clock and estimate the time taken to execute a firmware routine. Assertions can be enabled to track protocol violations or incorrect design behaviour during the execution of firmware sequence.

2. Instruction Set Simulator:
   Instruction set simulator is a software simulation model for the micro-controller and is provided by the vendor (the micro-controller executing the PM firmware is a 3rd party IP). There is no notion of the outside world and all entities that firmware communicates with are modelled in C++. The platform is much faster than VCS, however it does not enable cycle accurate simulation. Since there is no notion of RTL design, there is no scope for enabling assertions. However it is very suitable for fine grained verification of firmware sequences to ensure correctness of computations, ordering of sequences etc.

A satisfactory qualification of firmware requires both types of verification – datapath and quality. Datapath verification has higher precedence because of hardware interactions and missed hardware bugs can prove to be very expensive. Bugs missed due to inadequate verification for quality can still be fixed during post silicon bring-up. However, given the enormous amount of firmware implementation, combined with increased difficulty in identifying bugs in post silicon platforms, there is greater emphasis on verification for quality using pre-silicon platforms.

3.2 Testbench Architecture

Verification at system management sub-system level:

In order to understand the scope of firmware verification, let's refer back the system architecture diagram in *Section 2 : Brief Overview of the Architecture*. The PM controller, like any other sub-block is verified standalone for functionality. However in this IP level verification, firmware is not yet in picture. Firmware development begins in the system management sub-system, wherein multiple sub-blocks like voltage, clock and thermal controllers come together. There is a notion of the entire system at the system management sub-system. This makes the sub-system testbench a good starting point for VCS verification of firmware.

However interaction of firmware is not limited to sub-blocks in system management sub-system. It extends to the CPUs and North Bridge IP which are visible as RTL only in the SOC testbench. Given the standardisation of communication between various entities in the design, the CPUs and North Bridge IP are easily modelled using AXI BFMs in the system management testbench. Coverage for sideband communication between CPUs and North Bridge IP, during execution of firmware sequences, is covered in SOC level simulations.

Adequate coverage for firmware datapath and quality is available in the system management testbench. Moreover simulations are much faster in this testbench compared to SOC testbench making it the most preferred platform for firmware verification. The sections to follow describe in detail the infrastructure required for verification in this sub-system.

3.2.1 Plugging in AXI UVCs

As all sub-blocks in the entire design communicate mostly through the AXI standard via the on-chip communication channel, BFM development is greatly simplified. The verification component (also referred to as UVC) for an AXI interface is pre-developed by the methodology team in the company. This UVC is used in sub-system level verification to model the behaviour of CPU cores, North Bridge IP as AXI agents.

Discussed below are the key aspects of the AXI UVC:

- An AXI UVC can be configured to be **Master or Slave**. When configured as Master, the UVC can be used to issue outbound transactions from the AXI interface to which it is plugged. When configured as a Slave, the UVC is primarily a responder to incoming requests.

- The AXI UVC configured as Slave has built-in memory. This memory can be preloaded with address/data pairs from the test. Now read transaction from the firmware or any other entity to the pre-loaded address returns the corresponding pre-loaded data. This is very useful to model the actual behaviour of the external sub-block. AXI write transactions targeted to the Slave UVC will also be recorded in the local memory (i.e., the data written to a particular address will be available for read-back).

- A sub-block can have both Master and Slave AXI interfaces. The UVC at the respective interface must be configured appropriately.

Figure 5 shows the various design points wherein the AXI UVC is plugged in for firmware verification at the system management sub-system level.

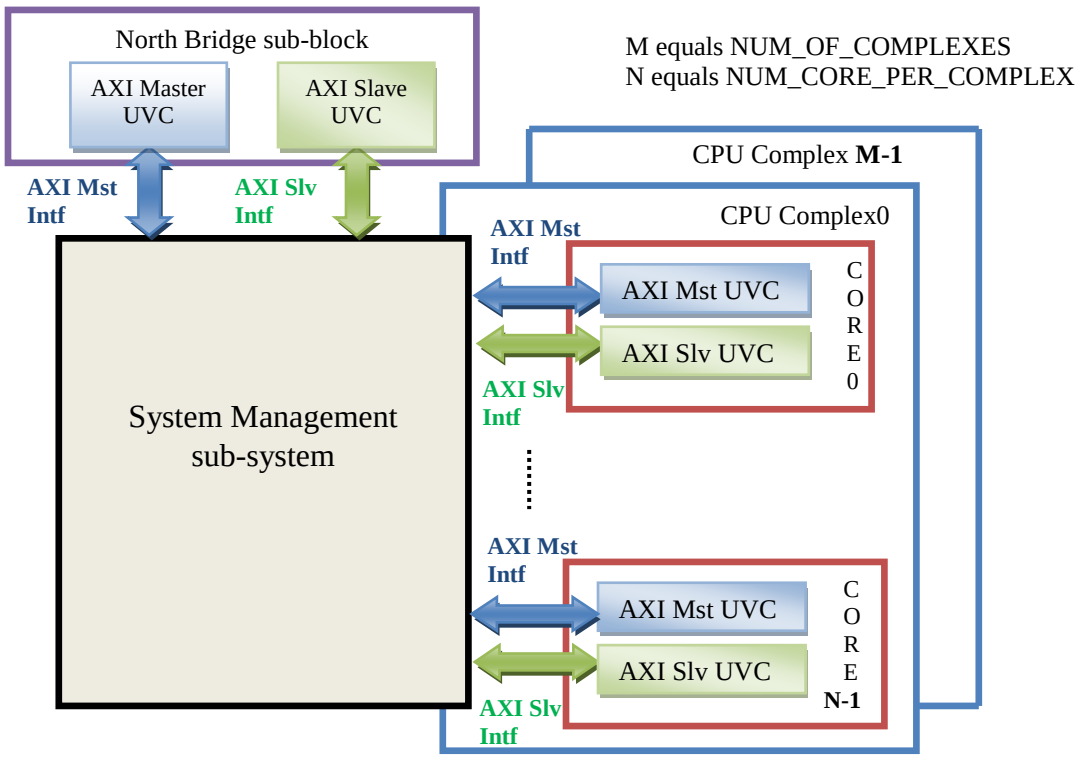

Figure 5. Plugging in AXI UVCs.

3.2.2 Transaction Collection

Need for transaction collectors:

Firmware sequence is primarily a set of control register programming through the on-chip communication channel. Correctness and completeness of the sequence are the key goals of verification. To achieve this, we need to track AXI transactions issued by firmware to interesting targets or end points.

Tracking transactions is also necessary in order to setup the status/response registers in the testbench (i.e. in UVCs for sub-blocks like CPU complex, North Bridge). These status registers are polled by firmware after programming the corresponding control registers, in order to track completion of the control sequence. Let's say firmware programs a control register to change clocks. It would, after this programming, poll a status register to wait for clock change to be done. The transaction collector would detect such control register programming and populate the corresponding status register (using the inbuilt memory with the Slv UVC, refer to *Section 4.2.1 Plugging in AXI UVCs*). Thus sub-blocks modelled using the AXI UVC will respond to firmware just as the corresponding RTL would have responded.

Standard method of transaction collection:

Let us consider the standard approach for transaction analysis/collection of AXI writes targeted to the AXI Slave interfaces shown in *Figure 5. Plugging in AXI UVCs*. Shown below is the code for the same.

```
`define NUM_OF_COMPLEXES 2
`define NUM_CORE_PER_COMPLEX 4
`define NUM_AXI_SLV_INTF  1
`define NORTH_BRIDGE  0
```

//Step 1:

//Define macros for collecting AXI write transactions to each core AXI SLV interface

```
`uvm_analysis_imp_decl(_complex_0_core_0)
`uvm_analysis_imp_decl(_complex_0_core_1)
….
….

`uvm_analysis_imp_decl(_complex_1_core_2)
`uvm_analysis_imp_decl(_complex_1_core_3)
```

//Define macro for collecting AXI write transactions at north-bridge sub-block AXI SLV //interface

```
`uvm_analysis_imp_decl(_north_bridge_sub_block)
```

//top level env component

```
class firmware_top_env extends base_env;
```

//Step 2:

//Declare the AXI UVCs for core and north bridge interfaces

```
axi_mst_uvc   complex_0_core_mst_uvc[`NUM_CORE_PER_COMPLEX],
              complex_1_core_mst_uvc[`NUM_CORE_PER_COMPLEX];
axi_slv_uvc   complex_0_core_slv_uvc[`NUM_CORE_PER_COMPLEX],
              complex_1_core_slv_uvc[`NUM_CORE_PER_COMPLEX];
axi_mst_uvc   north_bridge_mst_uvc;
axi_slv_uvc   north_bridge_slv_uvc;
```

//Configure the UVCs (i.e., specify the AXI interface, its type – Master or Slave, etc.)
//These details are not discussed since it is out of scope of the discussion in the paper

//Step 3:

//Declare a queue to collect all AXI write transactions targeted to ***core AXI SLV interface****.*
//There are `NUM_OF_COMPLEXES CPU complexes and each complex has

//`NUM_CORE_PER_COMPLEX cores. Hence the two dimensional array where each array
//element is a queue.

//axi_write_req_txn as the name suggests, is an AXI transaction of type Write. For more
//details on an AXI transaction structure, refer to AMBA-AXI standard

```
axi_write_req_txn
core_prefill_queue[`NUM_OF_COMPLEXES][`NUM_CORE_PER_COMPLEX][$];
```

//Declare an array of queues to collect all AXI write transactions targeted to various AXI
//SLV interfaces(eg: North Bridge IP etc.)

```
axi_write_req_txn axi_prefill_queue[`NUM_AXI_SLV_INTF][$];
```

//Step 4:

//Declare instances of analysis imports – note that each one has a ***unique instance name***
//Syntax: uvm_analysis_imp_class_name #(transaction type, parent) <name>
//Note that the name has "export" embedded in it. This is just to indicate that this is the
//point of termination for the detection of the write transactions. Whatever is captured here
//is exported to another (or rather used by another) component

```
uvm_analysis_imp_complex_0_core_0 #(axi_write_req_txn, firmware_top_env)
slv_export_complex_0_core_0;
….
….
uvm_analysis_imp_complex_1_core_0 #(axi_write_req_txn, firmware_top_env)
slv_export_complex_1_core_3;

uvm_analysis_imp_north_bridge_sub_block #(axi_write_req_txn,
firmware_top_env) slv_export_north_bridge_sub_block;
```

//Step 5:

//Create analysis imports – note that each one has a ***unique name***

```
function new (string name, uvm_component parent);
  uvm_analysis_imp_complex_0_core_0 = new ("slv_export_complex_0_core_0",
  this);
  ….
  ….
  uvm_analysis_imp_complex_1_core_3 = new ("slv_export_complex_1_core_3",
  this);
  uvm_analysis_imp_north_bridge_sub_block = new
  ("slv_export_north_bridge_sub_block", this);
endfunction : new
```

//Step 6:

//Associated with each analysis port is a ***unique write method****.*
//This write method is invoked every time a write transaction is detected at the interface.
//The write method maybe custom defined to perform any specific action as shown below.

```
virtual function void write_complex_0_core_0 (input axi_write_req_txn
transaction);
  core_prefill_queue[0][0].push_back (transaction);
endfunction: write_complex_0_core_0

…
…

virtual function void write_complex_1_core_3 (input axi_write_req_txn
transaction);
  core_prefill_queue[1][3].push_back (transaction);
endfunction: write_complex_1_core_3

virtual function void write_north_bridge_sub_block (input
axi_write_req_txn transaction);
  axi_prefill_queue[`NORTH_BRIDGE].push_back (transaction);
endfunction: write_north_bridge_sub_block
```

//Step 7:

//Create the AXI UVCs for the core and north-bridge interfaces

```
virtual function void build();
  super.build();
  //Build the AXI Mst/Slv UVCs here
  for (int i = 0; i < ` NUM_OF_COMPLEXES; i++) begin
    for (int j = 0; j < ` NUM_CORE_PER_COMPLEX; j++) begin
      case (i)
      0: begin
        complex_0_core_mst_uvc[j] = new(<expected parameters>);
        complex_0_core_slv_uvc[j] = new(<expected parameters>);
      end
      1: begin
        complex_1_core_slv_uvc[j] = new(<expected parameters>);
        complex_1_core_mst_uvc[j] = new(<expected parameters>);
      end
      endcase
    end
  end

  north_bridge_mst_uvc = new(<expected parameters>);
  north_bridge_slv_uvc = new(<expected parameters>);
endfunction : build
```

//Step 8:

//The analysis imports declared in function "new" in Step 5 are connected to the analysis //ports at the respective Slv interfaces. An analysis port for each AXI interface is built into //the AXI UVC for the interface. What we declare in Step 5 is used for a specialised function //and must connect to the inbuilt analysis port of the interface UVC.

```
    //The analysis port detects inbound write transactions and implements the corresponding
    //write function. The write function in our case results in the respective queue being
    //populated every time there is a write transaction at the interface.
    virtual function void connect();
      super.connect();

      for (int i = 0; i < ` NUM_OF_COMPLEXES; i++) begin
        for (int j = 0; j < ` NUM_CORE_PER_COMPLEX; j++) begin
          case (i)
          0: begin
            case (j)
            0: begin
                 complex_0_core_slv_uvc[j].analysis_port.connect
                 (slv_export_complex_0_core_0);
               end
            ….
            ….

            3: begin
                 complex_0_core_slv_uvc[j].analysis_port.connect
                 (slv_export_complex_0_core_3);
            end
            endcase
          end
          1: begin
            case (j)
            0: begin
                 complex_1_core_slv_uvc[j].analysis_port.connect
                 (slv_export_complex_1_core_0);
               end
            ….
            ….

            3: begin
                 complex_1_core_slv_uvc[j].analysis_port.connect
                 (slv_export_complex_1_core_3);
            end
            endcase
          end
          endcase
        end
      end
      north_bridge_slv_uvc. analysis_port.connect
                                          slv_export_north_bridge_sub_block);
    endfunction : connect
endclass : firmware_top_env
```

- From the above code snippet, it very clear that using the standard approach to declare and create analysis imports results in a lot of manual work for interfaces that are repetitive.

- For each interface, an analysis import with unique name needs to be created. The write function to populate the prefill queue is defined for every analysis port, although each one performs the action. There isn't a way to loop through the creation of analysis imports and automate the write functions.

- While the generation of code for repetitive interfaces may be automated through "dpl", the generated UVM env component will be large and difficult to read.

Modified method of transaction collection:

It is desirable to have an array of analysis imports for repetitive interfaces with identical behaviour. The standard method of using the macro **does not allow for declaring an array of ports** since these **macros are elaborated at compile time** whereas use of **arrays involves run time computations**. This unique problem of transaction analysis for repetitive interfaces is solved with the use of UVM subscribers.

```
`define NUM_OF_COMPLEXES 2
`define NUM_CORE_PER_COMPLEX 4
`define NUM_AXI_SLV_INTF  1
`define NORTH_BRIDGE  0
```

Shown below is the implementation with uvm_subscriber class.

//Step 1:
//Declare a uvm_suscriber class for transaction analysis.
//The class structure is very similar to the one created with uvm_analysis_imp_<name> macro
//For further details refer to uvm_tlm_macros.svh in UVM source code

// subscriber class for repetitive core interface

```
class core_axi_slv_export #(type T = axi_write_req_txn, type parent_ptr =
firmware_top_env) extends uvm_sunscriber;
  parent_ptr ptr;
  bit [`NUM_CORE_PER_COMPLEX-1 : 0] core_id;
  bit [`NUM_OF_COMPLEXES-1 : 0] complex_id;

  virtual function new (string name = "core_axi_slv_export", uvm_component
  parent);
    super.new(name, parent);
    $cast (ptr, parent);
  endfunction
```

```
  virtual function void write (axi_write_req_txn transaction);
    ptr.core_prefill_queue[complex_id][core_id].push_back(transaction);
    //Note the access of the core_prefill_queue array variable in the parent component
  endfunction
endclass : core_axi_slv_export
```

```
// subscriber class for any AXI Slv interface that is not repetitive
class axi_slv_export #(type T = axi_write_req_txn, type parent_ptr =
firmware_top_env) extends uvm_sunscriber;
  parent_ptr ptr;
  int sub_block_id;

  virtual function new (string name = "axi_slv_export", uvm_component
  parent);
    super.new(name, parent);
    $cast (ptr, parent);
  endfunction

  virtual function void write (axi_write_req_txn transaction);
    ptr.axi_prefill_queue[sub_block_id].push_back(transaction);
    //Note the access of the core_prefill_queue array variable in the parent component
  endfunction
endclass : axi_slv_export
```

```
//Top level component
class firmware_top_env extends base_env;
```

//Step 2:

```
    //Declare the AXI UVCs for core and north bridge interfaces
    axi_mst_uvc   complex_0_core_mst_uvc[`NUM_CORE_PER_COMPLEX],
                  complex_1_core_mst_uvc[`NUM_CORE_PER_COMPLEX];
    axi_slv_uvc   complex_0_core_slv_uvc[`NUM_CORE_PER_COMPLEX],
                  complex_1_core_slv_uvc[`NUM_CORE_PER_COMPLEX];
    axi_mst_uvc   north_bridge_mst_uvc;
    axi_slv_uvc   north_bridge_slv_uvc;

    //Configure the UVCs  by specifying the AXI interface, its type – Master or Slave, etc.
    //These details are not discussed since it is out of scope of the discussion in the paper
```

//Step 3:

```
    //Declare a multi-dimensional array of queues to collect all AXI write transactions at each
    //core Slv interface
    axi_write_req_txn
    core_prefill_queue[`NUM_OF_COMPLEXES][`NUM_CORE_PER_COMPLEX][$];
```

```
//Declare an array queues to collect all AXI write transactions targeted to various other
//AXI SLV interfaces
axi_write_req_txn axi_prefill_queue[`NUM_AXI_SLV_INTF][$];
```

//Step 4:

```
//Declare an array of analysis subscribers for repetitive core interfaces.
//Should the number of cores or complex change, this setup is very easily scalable.
core_axi_slv_export
core_slv_export[`NUM_OF_COMPLEXES][`NUM_CORE_PER_COMPLEX];
//Declare analysis subscribers for non-core interface like North Bridge AXIS lv interface
axi_slv_export slv_export[`NUM_AXI_SLV_INTF];
```

//Step 5:

```
//Create instances of analysis imports. Note that the name for very instance is uniquely
//specified in a simple manner
function new (string name, uvm_component parent);
  for (int i = 0; i < ` NUM_OF_COMPLEXES; i++) begin
    for (int j = 0; j < ` NUM_CORE_PER_COMPLEX; j++) begin
      core_slv_export[i][j] = new($psprintf
      ("axi_slv_export_complex_%0d_core_%0d", i, j), this);
      core_slv_export[i][j].core_id = i;
      core_slv_export[i][j].complex_id = j;
    end
  end

  slv_export[`NORTH_BRIDGE] = new ("north_bridge_axi_slv_export", this);
  slv_export[`NORTH_BRIDGE].sub_block_id = `NORTH_BRIDGE;
endfunction
```

//Step 6:

```
//Create the AXI UVCs for the core and north-bridge interfaces
//This section remains the same as in standard method.

virtual function void build();
  super.build();
  //Build the AXI Mst/Slv UVCs here
  for (int i = 0; i < ` NUM_OF_COMPLEXES; i++) begin
    for (int j = 0; j < ` NUM_CORE_PER_COMPLEX; j++) begin
      case (i)
      0: begin
        complex_0_core_mst_uvc[j] = new(<expected parameters>);
        complex_0_core_slv_uvc[j] = new(<expected parameters>);
      end
      1: begin
        complex_1_core_slv_uvc[j] = new(<expected parameters>);
        complex_1_core_mst_uvc[j] = new(<expected parameters>);
```

```
        end
        endcase
      end
    end

    north_bridge_mst_uvc = new(<expected parameters>);
    north_bridge_slv_uvc = new(<expected parameters>);
  endfunction : build
```

//Step 7:

//Hook up the subscribers to the source

```
  virtual function void connect();
    super.connect();
    for (int i = 0; i < ` NUM_OF_COMPLEXES; i++) begin
      for (int j = 0; j < ` NUM_CORE_PER_COMPLEX; j++) begin
        case (i)
        0: begin
             complex_0_core_slv_uvc[j].analysis_port.connect
                                                 (core_slv_export[i][j]);
        end
        1: begin
             complex_1_core_slv_uvc[j].analysis_port.connect
                                                 (core_slv_export[i][j]);
        end
        endcase
      end
    end
    north_bridge_slv_uvc.analysis_port.connect(slv_export[`NORTH_BRIDGE]);
  endfunction : connect
endclass : firmware_top_env
```

- As seen in the above piece of code, transaction collection at similar and repetitive interfaces is greatly simplified with the use of uvm subscribers.

- It removes the need to define the same implementation over and over again for transaction collection, thus overcoming the limitation of the standard approach.

- This approach is very scalable and easy to use across various derivatives of the same project.

3.3 Devising a Firmware Test

Section 4.2 Testbench Architecture discussed the components namely AXI models (UVCs) and transaction collectors, required for firmware verification at the system management sub-system level. In this section, there is a detailed discussion on devising a test for firmware verification with the use of these testbench components.

The firmware tests are developed using a hierarchical approach. The functions common to all tests are implemented and invoked in a base test and, power management feature specific verification is done through separate tests. These feature specific tests are derived from the base test. Thus common functions are re-used from the base test while feature specific scenarios are invoked in the respective derived tests.

3.3.1 Firmware Base Test

The firmware base test consists of the following common functions:

1. Initialisation of AXI models by populating key register-data pairs in the Slv UVCs (refer to *Section 4.2.1 Plugging in AXI UVCs* for more details).
2. Wait for completion of micro-controller initialisation, to ensure readiness to process interrupt messages.
3. Activating transaction collectors/analysers at interesting interfaces.

The UVM code for the base firmware test would look like the following:

```
//Define macros used in the test
`define CORE_PSTATE 1      //used for one-hot interrupt encoding
`define MIN_PSTATE_VAL 7

class firmware_base_test extends base_test;
  //base_test is the test derived from uvm_test for the system-management sub-block testbench.
  //If no base_test exists then firmware_base_test would be derived from "uvm_test"

  //Declare scoreboard variables
  //Discussed in Section 4.4 Scoreboard Methodology

  //Create an instance of the firmware env class
  firmware_top_env   m_firmware_top_env;

  //"new" function to create the object of the env instance etc.

  //The common functions and the actual test sequence are executed in the main_phase
```

```
virtual task main_phase (uvm_phase phase);
  phase.raise_objection (this);
  common_functions();    //wait for micro-controller initialisation,  activating transaction
                         //collectors
  test_sequence();
  phase.drop_objection (this);
endtask : main_phase

virtual task common_functions;
  wait_for_ucode_init();
  core_txn_collector();
  north_bridge_txn_collector();
endtask : common_functions

virtual task test_sequence;
  //This task is used to create interesting scenarios in the feature specific tests
  //Note that the task is declared "virtual" so that it can be redefined in the derived test
endtask : test_sequence

task core_txn_collector;
//The transaction collector loop waits for a transaction to appear at the interesting interfaces.
//When a transaction is detected (the subscriber would populate a queue declared in the
//firmware_top_env class – refer to Section 4.2.2 Transaction Collection) one of the following
//actions maybe taken:
//--- Populate scoreboard variable (refer to Section 4.4 Scoreboard Methodology)
//--- Activate a responder  – it populates a status register which firmware would poll (refer to
//Section 4.2.1 Plugging in AXI UVCs)

//The transaction detector is generic in that it detects transactions to the interface from any
//source. However in the firmware tests, no other entity is issuing any traffic to theses
//interfaces. Therefore, it can be safely assumed that all transactions appearing at the
//interface are issued by firmware.

  for (int i = 0; i < `NUM_OF_COMPLEXES; i++) begin
      for (int j = 0; j < `NUM_CORE_PER_COMPLEX; j++) begin
        //Variables used within the parallel threads must be declared automatic
        //This creates a local copy of the variable for each thread
        automatic int core = j;
        automatic int complex = i;
        automatic axi_write_req_txn core_req_txn;
        fork
          while (1) begin
            wait (core_prefill_queue[complex][core].size() != 0);
            core_req_txn = m_firmware_top_env.core_prefill_queue
                                    [complex][core].pop_front();
```

```
            case (core_req_txn.addr)
            //The case statement checks if the transaction received at the interface have one
            //of the following interesting addresses
            addr_1 : begin
                       //When a match to an interesting address is detected; take one of
                       // the following actions:
                       //1.        Populate scoreboard variables
                       //2.        Display addr/data programmed
                       //3.        Activate responders – for eg: populate status register
                       //          when control register is programmed by firmware
                     end

                     ….
                     ….
                     ….

            addr_N : begin
                       ….
                       ….
                       ….
                     end
            default : begin
                        `uvm_warning (get_type_name(), $psprintf
                                ("Transaction to unrecognised address"));
                      end
            endcase
          end
        join none
      end
    end
  endtask : core_txn_collector

  task north_bridge_txn_collector;
    //Similar scheme as the core_txn_collector
    //Wait for subscriber to populate a queue, process the transaction received
  endtask : north_bridge_txn_collector
endclass : firmware_base_test
```

3.3.2 Feature Specific Test

A feature specific test creates the necessary stimulus for triggering a particular PM firmware sequence. The structure of such tests can be summarised as follows:

1. Setup the BFMs/models of those sub-blocks involved in the specific feature to respond with intended data
2. Configure the interrupt message (*refer to Section 3.3 Interrupt Messaging*)
3. Send out the interrupt message from the Master AXI interface of the requestor sub-block
4. Wait for indication that interrupt is serviced by firmware – this is generally written to a specific status register in the requestor (which is tracked using the transaction detector).
5. Check for completeness of the firmware sequence (*discussed in Section 4.4. Scoreboard Methodology*)

The UVM code for a test that issues an interrupt message for pstate change of a core is shown below. The low-level functions for the various steps in the test are in the Appendix.

```
class core_pstate_interrupt_test extends firmware_base_test;
  //Variable declaration
  axi_write_req_txn core_txn; //This is the interrupt message write
                              //transaction from the requesting core
  int complex_id, core_id, pstate_id;
  …
  …

  //Test scenario for triggering the firmware sequence for pstate change
  virtual task test_sequence();
    //Step 1:
    //Randomize the complex id, core id and pstate id.
    //This ensures coverage for all core interfaces
    complex_id = $urandom_range (0, `NUM_OF_COMPLEXES-1);
    core_id = $urandom_range (0, `NUM_CORE_PER_COMPLEX-1);
    pstate_id = $urandom_range (0, `MIN_PSTATE_VAL);

    //Step 2:
    //Setup core registers by using the AXI Slave UVC memory.
    //These are the registers that firmware would be interested in to understand the request
    //For example, register which has the value of OS requested pstate-id for the core
    //Such register information will be found in core sub-block microarchitecture specification
    //and is implementation specific

    setup_core_pstate_reg(complex_id, core_id, pstate_id);

    //Construct AXI message with a one-hot encoding which indicates the nature of the request
    //An AXI transaction is characterised by target address and data to be programmed
    //amongst other parameters (which is discussed in more detail in Section 3.3 Interrupt
    //Messaging)
```

```
    //Here the target address is the mailbox in the PM controller, refer to "inbound
    //communication" in Section 3.2. Firmware Datapath
    core_txn.addr = `PM_mailbox_addr_on_network;
    core_txn.data = '0;
    core_txn.data[`CORE_PSTATE] = 1;

    //Step 3:
    //Issue transaction from core AXI Master Interface
    //axi_write is a custom function that exercises the Master UVC to send an AXI transaction
    //Inputs to the function are - id of the CPU complex, id of the requesting core within its
    //complex, target address of the transaction and the data.
    //The complex id and core id information are used by the function to identify the correct
    //Master UVC using which the AXI transaction is sent out to the target.
    axi_write (complex_id, core_id, core_txn.addr, core_txn.data);

    //Step 4:
    //Wait for firmware to indicate completion of the requested sequence
    //There is a timeout check to ensure that the "Done" indication is received within a
    //reasonable duration
    fork
      begin
        //This is a blocking task which returns only on detecting a "Done" indication from
        //firmware.
        wait_for_intr_done(complex_id, core_id);
      end
      begin
        p1 = process::self();
        # timeout;
        $uvm_error (get_type_name(), $psprintf ("Core Pstate Intr Done
        timeout"));
      end
    join_any
    p1.kill();   //This is necessary to ensure that there isn't a false error when exit from the
                 //fork-join_any loop is because of completion of wait_for_intr_done()

    //Step 5:
    //Check for correctness and completeness of firmware sequence
    //More details on the below will be described in Section 4.4 – Scoreboard Methodology
    check_firmware_scbd();

  endtask : test_sequence

endclass: core_pstate_interrupt_test
```

- Thus, for each such firmware interrupt type from a sub-block, a unique test can be devised.

- Scope for randomisation in such tests is mainly in setting up registers in the sub-block verification component. These registers are read by firmware to better understand the request. Randomising such inputs help stress the firmware sequence.

- These feature-specific tests provide end-to-end coverage in that there is verification of whether the interrupt from a sub-block reaches the firmware, whether the interrupt is recognised and serviced correctly by firmware.

3.4 Scoreboard Methodology

This section discusses the test-based scoreboard methodology. It is a unique method that helps ensure correctness of firmware sequence.

3.4.1 Need for Scoreboard

- The various steps in the test, starting with issuing the interrupt message, up until ensuring completion of the sequence, provide adequate coverage for firmware datapath (both inbound and outbound).

- To ensure if firmware executed the necessary steps was a tricky problem in VCS. The standard UVM scoreboard methodology was experimented with, but it only added to the complexity of verification. It was realised that the standard method is suitable in the case of protocol checking at an interface, whereas for sequence check, it only creates more overhead.

- Sequence check requires tracking transactions, verifying if interesting targets were programmed with the right values. In the architecture, a lot of the control sequences are executed by programming specific control registers. So ensuring that this control register programming does happen, and that it is accurate, is critical to firmware verification.

3.4.2 Test-Based Scoreboard

A test-based scoreboard methodology for ensuring correctness of firmware sequence involves the following steps:

Step 1: Identify interesting targets/programming from the micro-architecture specification of the firmware sequence for an interrupt.

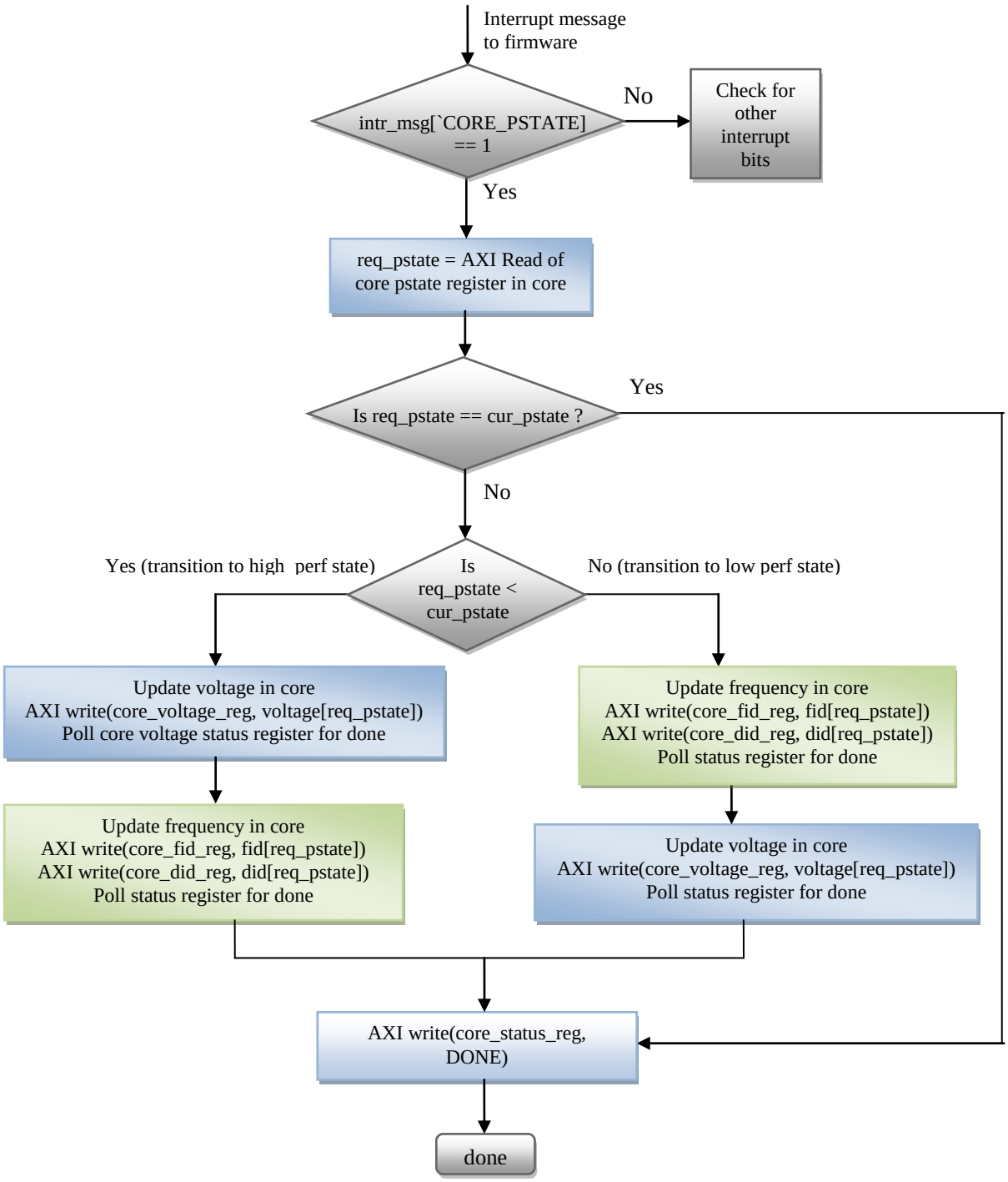

Figure 6. Core pstate flowchart.

Let us consider the example of OS pstate interrupt sequence in firmware. For the sake of simplicity, only some of the key steps required to highlight the effectiveness of test-based scoreboard are mentioned in the flow chart shown in **Error! Reference source not found.**.

Step 2: In the firmware test, program an enumerated structure with interesting targets.

Based on the understanding of the firmware sequence from Figure 6, the enumerated scoreboard structure for the sequence looks like the following:

```
typedef enum {
  WRITE_PSTATE_VOLTAGE,
  WRITE_PSTATE_FID,
  WRITE_PSTATE_DID,
  WRITE_INTR_STATUS
} core_pstate_event;
```

Step 3: Create hash variable with enumerated scoreboard structure as the key.

```
typedef  enum {FALSE=0, TRUE=1} bool;
bool core_pstate_scbd [core_pstate_event];          //Actual
bool core_pstate_scbd_expected [core_pstate_event]; //Expected
core_pstate_event pstate_event;
```

core_pstate_scbd, core_pstate_scbd _expected are hash variables. The hash keys are of type core_pstate_event while the corresponding values are boolean variables.

Shown below is the expansion of the hash:
core_pstate_scbd [WRITE_PSTATE_VOLTAGE] = TRUE/FALSE
core_pstate_scbd [WRITE_PSTATE_FID] = TRUE/FALSE
core_pstate_scbd [WRITE_PSTATE_DID] = TRUE/FALSE
core_pstate_scbd [WRITE_INTR_STATUS] = TRUE/FALSE

Step 4: Program the expected scoreboard variable in the test before issuing the interrupt Message.

For example, let's consider that the test is issuing an interrupt message with pstate id different from the current pstate id of the core. The expected scoreboard enum variable would be as follows:

```
pc6_entry_scbd_expected = '{
              WRITE_PSTATE_VOLTAGE:   TRUE,
              WRITE_PSTATE_FID:       TRUE,
              WRITE_PSTATE_DID:       TRUE,
```

```
            WRITE_INTR_STATUS:      TRUE
          }
```

Step 5: Reset the actual scoreboard hash variable to default values.

```
if (core_pstate_scbd.first(pstate_event))  begin
  do begin
    core_pstate_scbd[pstate_event] = FALSE;
  end while (core_pstate_scbd.next(pstate_event));
end
```

Step 6: In the transaction collector, mark the interesting target when it is hit.

The transaction collector has a case statement for each address that firmware is expected to program. When control registers involved in specific sequences are written to, it is marked in the corresponding scoreboard hash variable as shown below.

```
    for (int i = 0; i < ` NUM_OF_COMPLEXES; i++) begin
        for (int j = 0; j < ` NUM_CORE_PER_COMPLEX; j++) begin
          //Variables used within the parallel threads must be declared automatic
          //This creates a local copy of the variable for each thread
          automatic int core = j;
          automatic int complex = i;
          automatic axi_write_req_txn core_req_txn;
          fork
            while (1) begin
              wait (core_prefill_queue[complex][core].size() != 0);
              core_req_txn = m_firmware_top_env.core_prefill_queue
              [complex][core] .pop_front();

              case (core_req_txn.addr)
                addr_for_VOLTAGE_CHG : begin
                    core_pstate_scbd[WRITE_PSTATE_VOLTAGE] = TRUE;
                end
                ….
                ….

                addr_for_INTR_STATUS: begin
                    core_pstate_scbd[WRITE_INTR_STATUS] = TRUE;
                end

              endcase
            end
          join_none
        end
    end
```

Step 7: At the end of the test, compare the actual and expected scoreboards.

```
if (core_pstate_scbd.first(pstate_event))  begin
  if (core_pstate_scbd.num() != core_pstate_scbd_expected.num()) begin
    `uvm_error (get_type_name(), $psprintf ("ACTUAL and EXPECTED SCBD size
mismatch!"));
  end

  do begin
    if (core_pstate_scbd[pstate_event] !=
                                    core_pstate_scbd_expected[pstate_event])
      begin
        `uvm_error (get_type_name(), $psprintf ("Mismatch !!! Event -
        %0s\tActual - %0s\tExpected - %0s\n", pstate_event.name,
        core_pstate_scbd[pstate_event].name,
        core_pstate_scbd_expected[pstate_event].name));
    end
  end while (core_pstate_scbd.next(pstate_event));
end
else begin
  `uvm_error (get_type_name(), $psprintf ("Pstate SCBD is empty!"));
end
```

What happens if a scoreboard mismatch is found?

When the scoreboard check fails indicating that a target hasn't been hit, the following procedure is adopted.

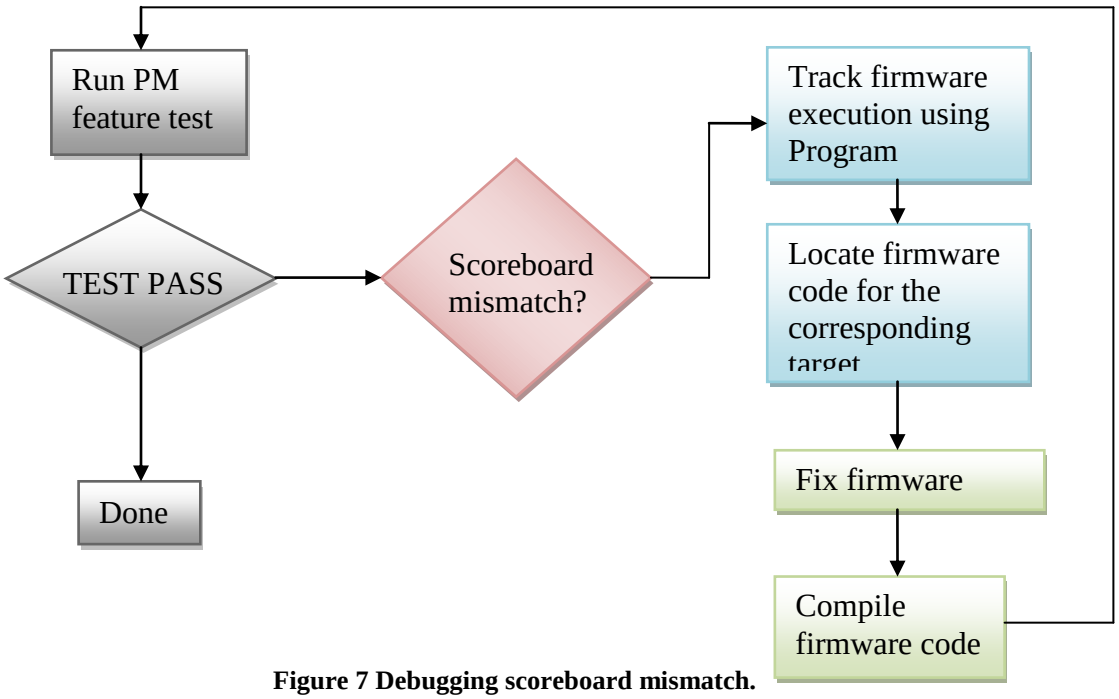

Figure 7 Debugging scoreboard mismatch.

1. Locate the section of the ISR which programs the target register.
2. Try and analyse why the target might not have been hit
3. If the problem is not intuitive, then track the firmware execution on the program counter.
4. Traverse the firmware execution using the program counter and locate the problem point.

Summary:

The approach discussed here provides tracking for essential firmware programming. However, ensuring complete accuracy of the programming sequence requires further developments to the scoreboard structure. In firmware sequences, ordering of certain steps might be important.

For eg. while increasing the performance state of the core, it's voltage must be increased before increasing the frequency. To ensure that firmware indeed obeys this sequence, we need tracking for the order of execution. And for this, the hash variable must be made two-dimensional. This will allow us to associate a tag/sequence number with each programming step which may then be verified at the end of the test to confirm that firmware did execute the sequence in expected order.

4. Conclusion

This paper has discussed the various aspects of verifying C++ firmware sequences in a UVM testbench. Typically firmware qualification has been reserved for post-silicon validation given its flexibility and dynamic nature. However, with increasing use of firmware to implement critical features in the design, pre-silicon validation is becoming important.

Firmware verification is similar to RTL verification in many ways. The distinguishing factor between the two is the check for correctness. Test-based scoreboard methodology has proved very effective in identifying firmware programming issues early in the design process. As discussed in Section 3.1, mathematically intense computations and checking correctness of ordering in the sequence, may be covered in standalone firmware verification using the Instruction Set Simulator as this is setup is much faster. Therefore, the coverage obtained with the above scoreboard methodology is considered adequate for VCS testing.

A number of firmware bugs were uncovered with VCS verification. They included programming issues, firmware infrastructure related issues and so on. It was clear that deferring the identification of these bugs to post-silicon stage would have tremendously increased the bring-up time, while also increasing the risk of finding hardware bugs late in the process. There is growing recognition for the value added by VCS verification and more time and resources are being invested on further fine-grained verification of firmware. The end goal of this verification

effort is to be able to qualify firmware as effectively as hardware before tape out. Although realistically 100% firmware coverage is not possible pre-silicon given the dependency on post silicon characterisation and so on, it certainly helps to have production firmware qualified to a great extent to be release the chip to the consumer as early as possible.

5. References

[1] https://verificationacademy.com/cookbook/uvm
[2] http://www.arm.com/products/system-ip/amba-specifications.php

6. Acknowledgements

I would like to thank my team management at AMD – Aleksander Cejkov, Drago Ignatovic and Ljubisa Bajic for giving me the opportunity to work on exciting projects which has resulted in this paper. Many thanks to all my team members who have contributed directly and indirectly in my work at AMD. I appreciate the support and guidance provided by my SNUG reviewer Paul Lungu, Synopsys reviewer Ravi Gehani and all those involved in one way or the other in helping me write this paper.

7. Appendix

A. Hardware for Firmware Execution

In this section, there is an overview of the hardware associated with power management firmware.

In Figure 7, there is a "PM controller" with a micro controller. The micro controller is a licenced third party IP and firmware code executes on this. PM controller has the necessary infrastructure to support the microcontroller and the key elements are described briefly here:

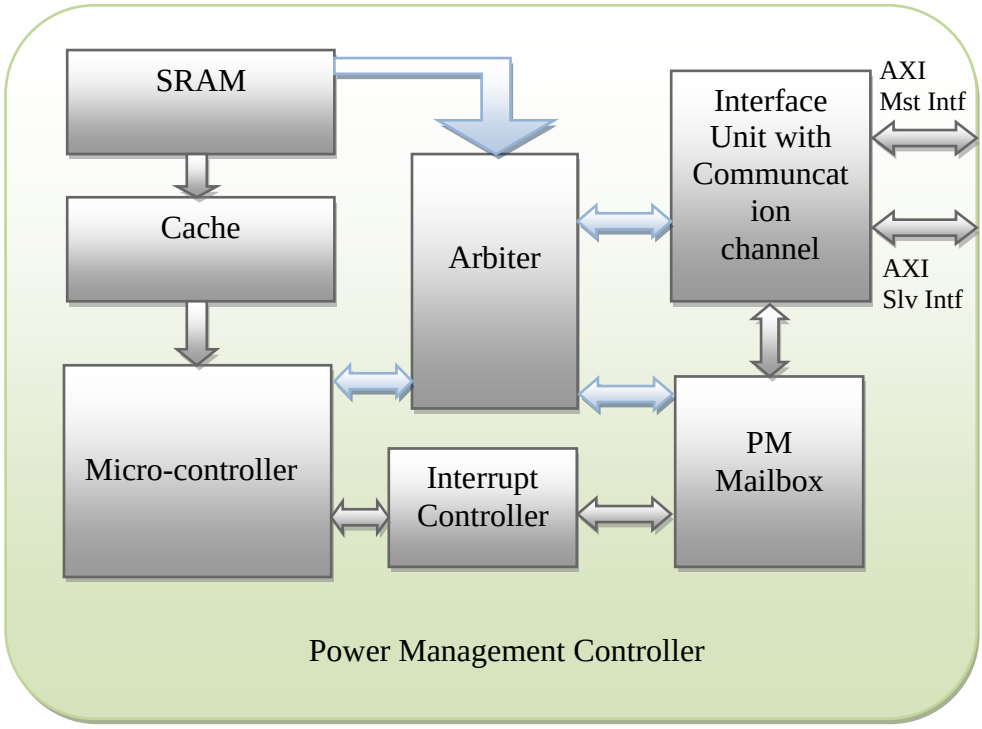

Figure 8. Power management controller diagram.

1. SRAM – 256K bytes
   The firmware micro-code is 512 Kb in size. The entire firmware hex image is loaded to DRAM from the flash memory on start-up during the boot sequence. Half the hex image

is housed in the on-chip SRAM in PM controller. The other half is paged from DRAM as and when required.

2. Cache
   For faster execution of the micro-code, caches are implemented with the micro controller. This minimizes processor accesses to SRAM and paging with DRAM during micro code fetch.

3. Power Management Mailbox
   Firmware can execute specific sequences as per requests from certain sub-blocks. More details on these sequences are discussed in Section 2.3 Interrupt Messaging. The sub-blocks request sequence execution by issuing interrupt messages over the on-chip communication channel. These messages are issued to the power management mailbox in the PM controller.

   There are three mailboxes implemented in SRAM to service interrupts of different priorities. Depending on the nature of sequence being requested, an interrupt can be of priority P0, P1 or P2 in the increasing order of priority. The requesting sub-block sends the interrupt message to the corresponding mailbox based on the priority of the interrupt it is requesting.

   By implementing a mailbox to record the incoming interrupts to the firmware, it is ensured that no interrupt is lost if the micro controller is busy in execution while receiving the interrupt.

4. Interrupt Controller
   This module interfaces with the mailbox to interrupt the micro controller. The interrupt controller is hardwired to the micro controller. Whenever the micro controller is available and a pending interrupt is detected in the mailbox, the interrupt controller directs the interrupt from the mailbox to the micro controller. It is possible to have multiple pending interrupts in the mailboxes at any given point in time. The order in which the interrupts are relayed to the micro controller depends on the priority of the interrupts detected.

5. Interface Units to the on-chip communication channel
   The Interface Units use the AXI standard for communication with the on-chip communication channel. It receives inbound messages from the channel and pushes them onto the appropriate mailbox. It routes the outbound messages from the micro controller onto the channel.

6. Crossbar for arbitration between SRAM and Interface Units

Micro controller reads data from SRAM and this is primarily micro code. It also reads data from the mailbox whenever a pending interrupt is flagged by the interrupt controller. Between the two, an arbitration is required for proper flow of data to the micro controller and this is performed by the crossbar.

B. Role of Firmware in Power Management

This section discusses the role of firmware in defining various power states in the chip.
Power state of a design is primarily decided by the voltage and frequency parameters. Transition to different power states involves adjusting the voltage and frequency of a design block appropriately (i.e., with the right values and in the right sequence). The sequence is important to ensure proper electrical behaviour of the circuit, so a voltage increase must precede a frequency increase while a frequency decrease must precede a voltage decrease. Such sequences are best handled in firmware rather than hardware as there is better control over the entities involved in the change. Moreover, there is easier scope for change and improvements all through the design process.

Various types of power management algorithms are executed in firmware. Some of them execute on request, while some of them run periodically. At any given point in time, it is in the control of firmware to ensure that the system is in an optimal power state.

Discussed below are some of the key power management algorithms.

Examples of algorithms that execute periodically:

1. Dynamic voltage and frequency scaling
   This algorithm runs at regular intervals. It analyses the work loads of various entities and computes the most optimal voltage and frequency and initiates change to the desired level as required.

2. Power budgeting
   The major design components namely the graphics and CPU sub-blocks are the primary power consumers in the chip. Most of the power mangement involves proper power budgeting between these entities and this is crucial to ensure optimal performance of the chip. Power, which is a factor of voltage and frequency is allocated in credits between graphics and CPUs. If one entity is found to be handling a lower workload than the peak workload, then some of its credits may be transferred to the other, thus ensuring optimal distribution of resources. This algorithm performs a dynamic computation of the power

budgeting and initiates voltage and frequency changes through the scaling algorithm discussed in point 1.

3. Temperature monitoring
   There are algorithms that monitor temperature periodically and adjust the power levels appropriately. The temperature information is extracted from a temperature sensor module that translates the analog value to digital temperature value suitable for use by firmware. The primary goal of this algorithm is to ensure that the chip never exceeds the maximum permissible temperature at any given point in time. Since temperature is a function of external factors in addition to current voltage and frequency, constant monitoring is necessary to ensure that TDP limits are never exceeded.

Examples of algorithms that execute on request:

1. OS Pstate change
   The operating system can request the firmware to transition a CPU core to different performance level. The performance level, also referred to as Pstates are per the ACPI standard. Through this mechanism, OS can request high performance for a core that is likely to handle significant load or it can request a lower performance for a core that has finished processing a critical task.

2. Package C6
   When all cores are in sleep state, it is beneficial to transition the eco-system around the core to OFF state. This sequence is quite involved in that it must quiesce its boundaries, notify any other sub-block that might be involved in decision making, in addition to saving the current state of the system. Even when requested to exit from this state, the sequence involves several critical steps to restore the CPU eco-system back to what it was prior to package C6. Having the firmware control all these transitions is very beneficial since the complexity is moved entirely to a software sequence. Although latency might be worse than a purely hardware implementation, it's a trade-off between flexibility versus performance.

3. Cstate Boost
   When a significant number of CPU cores enter sleep state, it is beneficial to have the rest of them run at higher perfromance state. The Cstate Boost algorithm handles this sequence. It involves configuration in the controller logic that notifies the firmware of a potential to transition the CPU eco-system to a boost performance state. This configuration ensures that when a sleeping core wakes up, firmware is notified immediately so that performance state that existed prior to the boost state is restored back with minimum latency.

4. Thermal Control
   There is a thermal sensor block that notifies firmware of any events that might require the ASIC to either turn off or transition to a very low performance state. This can happen if the maximum temperature on the chip exceeds a maximum threshold. This block could also have the interface to external agents that might want to communicate with the cores through the SBI protocol. When requests are issued by external agent for puposes of reading CPUID information from the cores, an interrupt is issued to the firmware. Firmware then performs the needful interactions with the cores and gets back with the required data to the thermal module which is then communicated back to the external agent through SBI.

5. Pstate change for North Bridge IP/Memory Controller
   A lot of power mangement functionality was implemented in North Bridge IP in the previous generations. It communicated directly with the core and memory controller through dedicated sideband interfaces. While a lot of this communication has been standardized through the on-chip bus, there is still some portion of this side band interface. This is primarily for ensuring low latency communication with the cores and memory controller. The North Bridge IP and the memory controller can also be put into difference performance states depending on their workload. Transitioning the blocks to a different power state involves explicitly quiescing some of the internal state machines and any other module that communicates directly with them. This is in addition to applying a suitable clock frequency and voltage to the blocks. Given the complexity in the sequence, it is best handled in firmware.

C. Low-Level Functions in Feature-Specific Firmware Tests

```
task wait_for_ucode_init();
  //Read STATUS register which is set by firmware when it has completed all the
  //initialisation steps.
  //sub-blocks are configured with test ports for the benefit of verification and such test
  //ports maybe used to poll for the STATUS register
  //If such a capability is not available; we may probe the internal signal corresponding to
  //the status to check for completion of init sequence.

  bit [31:0] status;
  do begin
    read_register_thr_test_port (`FIRMWARE_STATUS_REG, status);
    #2us;
  while (status != `DONE);
endtask : wait_for_ucode_init
```

```
task setup_core_pstate_reg(input int complex_id, core_id, pstate_id);
  //The AXI UVC configured as Slv has in in-built prefill function. This function loads the
  //UVC memory with the address, data pair specified with the function

  //`CORE_PSTATE_REG is the address seen by the UVC when firmware tries to read the
  //pstate id value from the core.  This address is different from the address of the same
  //register on the on-chip communication channel since the individual core does not have
  //the notion of the communication channel.

  //This read to this register is issued in the Interrupt Service Routine which is invoked
  //when the corresponding interrupt message is received. For more details on how
  //information on the requestor is extracted by firmware, refer to Section 3.3 Interrupt
  //Messaging. The UVC responds with the value prefilled by the test

  case (complex_id)
      0: begin
           complex_0_core_slv_uvc[core_id].prefill
                                        (`CORE_PSTATE_REG, pstate_id);
      end
      1: begin
          complex_1_core_slv_uvc[core_id].prefill
                                        (`CORE_PSTATE_REG, pstate_id);
      end
  endcase
endtask : setup_core_pstate_reg


task axi_write (input int complex_id, core_id, bit [31:0] addr, data);
  //The UVC configured as Master has a built-in write task. When this task is invoked,
  //it issues the transaction on the corresponding AXI Master interface.
  //This transaction is routed through the on-chip communication channel to the target
  //address.

  case (complex_id)
  0: begin
       complex_0_core_mst_uvc[core_id].write(addr, data);
     end
  1: begin
       complex_1_core_mst_uvc[core_id].write(addr, data);
     end
  endcase
endtask : axi_write


task wait_for_intr_done(input int complex_id, core_id);
```

```
  //One completion of the interrupt service routine, firmware is expected to write to the
  //interrupt status register in the core. The "done" status is a one-hot encoding of the
  //interrupt requested.
  //Here the mem_read function which is inbuilt with the AXI Slv UVC is leveraged
  //`CORE_INTR_STATUS_REG is the status register address seen by the UVC

  bit [31:0] status;
  do begin
    case (complex_id)
    0: begin
         complex_0_core_slv_uvc[core_id].mem_read
                                  (`CORE_INTR_STATUS_REG, status);
       end

    1: begin
         complex_1_core_slv_uvc[core_id].mem_read
                                  (`CORE_INTR_STATUS_REG, status);
       end
    endcase
  end while (status != `CORE_PSTATE);
  //Note that the one-hot intr encoding used to construct the interrupt message is what is
  //expected on the status register
endtask : wait_for_intr_done();


task check_firmware_scbd();
  if (core_pstate_scbd.first(pstate_event))  begin
    do begin
      if (core_pstate_scbd[pstate_event] !=
                                       core_pstate_scbd_expected[pstate_e
                                    vent])
        begin
          `uvm_error (get_type_name(), $psprintf ("Mismatch !!! Event -
          %0s\tActual - %0s\tExpected - %0s\n", pstate_event.name,
          core_pstate_scbd[pstate_event].name,
          core_pstate_scbd_expected[pstate_event].name));
      end
    end while (core_pstate_scbd.next(pstate_event));
  end
endtask : check_firmware_scbd
```