Accessing DesignWare® Sensor and Control IP Subsystem Resources in an OVM/UVM Testbench using a Register Abstraction Layer (RAL)

Stefan Neumann

Intel Corporation
Leixlip, Ireland

www.intel.com

ABSTRACT

The DesignWare® Sensor and Control IP Subsystem features integrated hardware accelerators and peripherals for sensor-specific applications to maximize sensor processing efficiency. This is achieved by tightly coupled internal interfaces directly accessible from the embedded DesignWare® ARC® EM processor. This paper details how we leveraged an OVM/UVM-based Register Abstraction Layer (RAL) to design a constrained random OVM/UVM testbench around the fully configurable DesignWare® Sensor and Control IP Subsystem to simplify the vertical integration and reuse of the verification IP (VIP). We utilize a small assembly program executed on the embedded processor to provide an essential interface between the verification environment and the available processor-internal resources. The provided interface allows the resources to be entirely controlled by the verification environment using RAL or higher-level sequences, avoiding the need for test specific Assembly/C programs to model complex scenarios on cluster or full chip level.

Table of Contents

Table of Figures

1. Introduction

The DesignWare® Sensor and Control IP Subsystem is optimized to process data from digital and analog sensors, offloading the host processor and enabling more power efficient processing of the sensor data. This is achieved by tightly coupled interfaces and resources only directly accessible from the embedded DesignWare® ARC® EM processor. Those resources include but are not limited to integrated peripherals, embedded memories as well as timers and a configurable number of external interrupts, power management and additional hardware accelerators as shown in Figure 1.

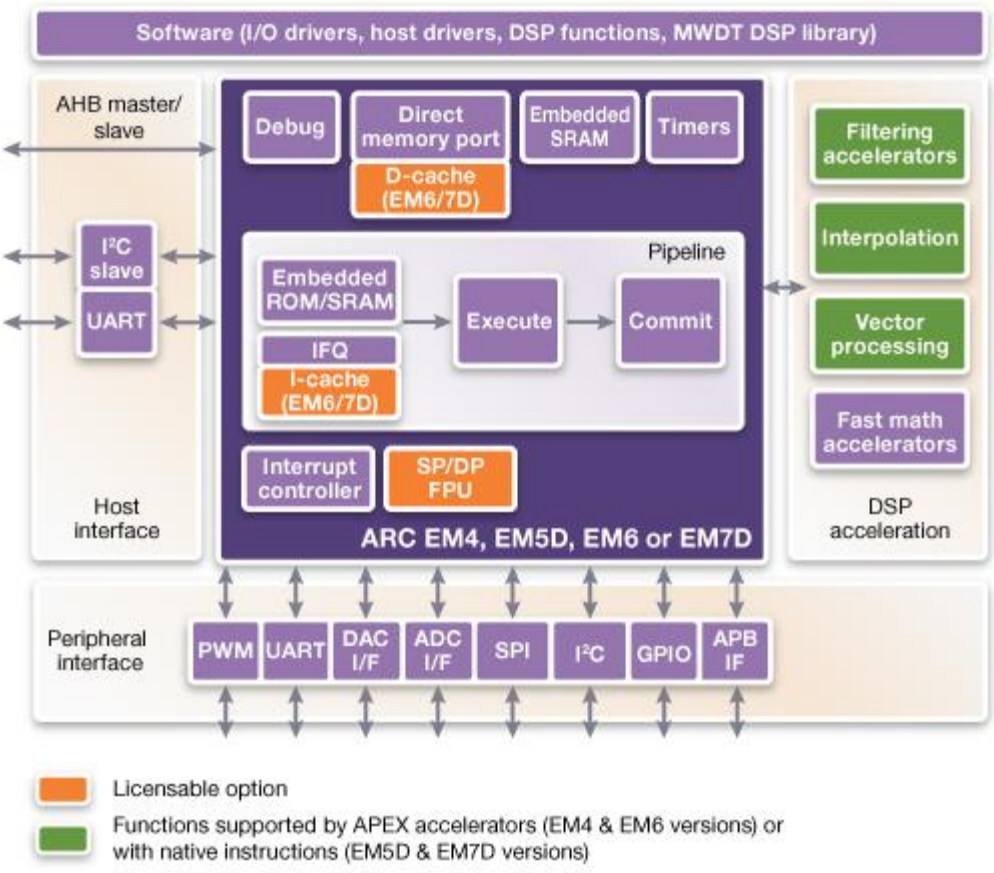

Figure 1 Sensor and Control IP Subsystem Overview [1]

Verification of the Sensor and Control IP Subsystem and its components is usually done through a set of confidence tests (CCT) which are supplied with the ARC® EM build environment and are provided for basic verification purposes only. The tests are assembly and C-code based and executed on the embedded processor by invocation of a software debugger which can be attached to the RTL simulator and controls the test execution in the environment. More complex scenarios emerge while integrating the Subsystem IP into a SoC environment with a host processor and requires additional test content written in a higher level programming language. This on the other hand requires extensive knowledge of the ARC® EM processor and software programming infrastructure and increases the learning curve required for integrating the IP.

Another drawback of this approach is the fact that the test execution and stimulus generation is predefined by the compiled test code rather than by the OVM/UVM testbench environment of the rest of the SoC design and is rather static in nature.
For our SoC design we chose to design a constrained random OVM/UVM environment around the DesignWare® Sensor and Control IP Subsystem to simplify the vertical integration and horizontal reuse of the verification IP (VIP) and return the control of the test execution back to the testbench environment. Therefore we leveraged an OVM/UVM-based Register Abstraction Layer (RAL) to provide access to the internal processor restricted resources and start transactions on the ARC® EM processor to interact with other components of the SoC design.
This paper details the testbench architecture for the Sensor and Control IP Subsystem which uses a two layer approach implemented using RAL to control the ARC® EM processor. First we developed an access layer to create an access path to the restricted resources and built a control layer to handle different resource types on top of it.

2. Subsystem Address Space Layout and Resources

Let's take a closer look at the processors view on its resources in Figure 2. The embedded ARC® EM processor is based around a set of general purpose registers building the source and destination for executed instructions. Control and status registers of the core are mapped into a separate auxiliary register space and can be accessed using the Load Register (LR) and Store Register (SR) instructions. In contrast to this external and closely coupled memories are accessed using Load (LD) and Store (ST) instructions.

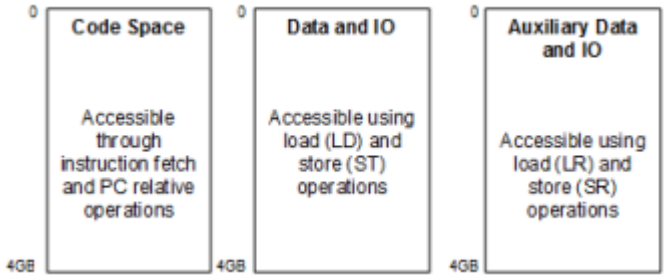

Figure 2 ARC® EM Address Space Layout [2]

Control registers of peripherals and hardware accelerators are also located in the Auxiliary Data and IO space and are not memory mapped to be accessed by the host processor in the SoC design. Accessing the auxiliary space is limited to the ARC® EM processor or an external debugger. However, using the debugger interface has a large impact on simulation performance. To avoid this performance degradation we need to find another access path to the resources.

3. Access Layer Architecture using RAL backdoor path

Our proposed solution for this issue utilizes a small assembly program (test daemon) executed on the embedded processor to provide an essential interface between the verification environment and the available processor-internal resources. The OVM/UVM environment controls the execution of this program by writing to a set of general purpose register in the core using RAL

with the register content being evaluated by the ARC® EM processor itself. As the environment does not have access to the general purpose register over an external bus interface, we use a RAL backdoor mechanism to access them. The registers are therefore directly modified using the HDL paths and SV VPI calls.
The test daemon is designed to be always running and responsive to the test environment. This requires it to handle every possible exception or interrupt condition during execution in a generic way and report those conditions to the environment. This allows the environment to respond to certain conditions and it stays in control of the overall test execution and stimulus generation.

```
.align 32; inf_loop:
bbit0 STATUS_REG, 31, inf_loop ;Command valid == 0? Branch
                               ;back to inf_loop
                               ;Bit 31 in status register
                               ;signals valid cmd issued by TB
bi    [CMD_REG]                ;Indexed branch to aligned task

.align 32; set_aux_reg_cmd:
sr    DATA_REG,   [ADDR_REG]   ;Store data to aux register
mov   STATUS_REG, 0            ;Success, clear status register
b     inf_loop                 ;Branch back to inf_loop

.align 32; get_aux_reg_cmd:
lr    DATA_REG,   [ADDR_REG]   ;Read data from aux register
mov   STATUS_REG, 0            ;Success, clear status register
b     inf_loop                 ;Branch back to inf_loop

.align 32; set_memory_cmd:
st    DATA_REG,   [ADDR_REG]   ;Store data to memory
sync                           ;Wait for outstanding memory to
                               ;catch potential exceptions
mov   STATUS_REG, 0            ;Success, clear status register
b     inf_loop                 ;Branch back to inf_loop

.align 32; get_memory_cmd:
ld    DATA_REG,   [ADDR_REG]   ;Load data from memory
sync                           ;Wait for outstanding memory to
                               ;catch potential exceptions
mov   STATUS_REG, 0            ;Success, clear status register
b     inf_loop                 ;Branch back to inf_loop

.align 32; excp_handler:       ;Recover from exception by
sr    inf_loop,   [ERET]       ;returning to inf_loop
mov   STATUS_REG, 1            ;Flag exception in status
rtie                           ;register and clear bit 31
```

Figure 3 Basic Functionality of the Test Daemon Program

Figure 3 shows the assembly code of the test daemon with some of the most rudimentary tasks necessary. The daemon runs a minimal infinite loop, constantly checking a general purpose register (STATUS_REG) for bit 31 to be set by the testbench via RAL backdoor access. This status bit indicates a valid instruction request being issued by the testbench and requests the processor to start execution of the selected task. This task itself is specified via a different general purpose register (CMD_REG), with command specific information like data and addresses being stored in DATA_REG and ADDR_REG. With the four shown tasks, the test daemon already provides all necessary functionality to access the data and auxiliary data address spaces from Figure 2 via get and set methods.
Exceptions are recognized by the test daemon synchronously once they occur and due to the simplicity of the program they can only be caused by the instructions in the issued tasks. An example for an exception would be an illegal address for an auxiliary register which will be raised once the register is accessed by the LR/SR instruction. A simplified exception handler is also shown in Figure 3 where program execution is recovered by returning to the start of the infinite loop and with the failure, occurred during execution of the task, being flagged in the status register.
The test environment can notice that a task has finished execution by reading back the STATUS_REG and checking the most significant bit being cleared.

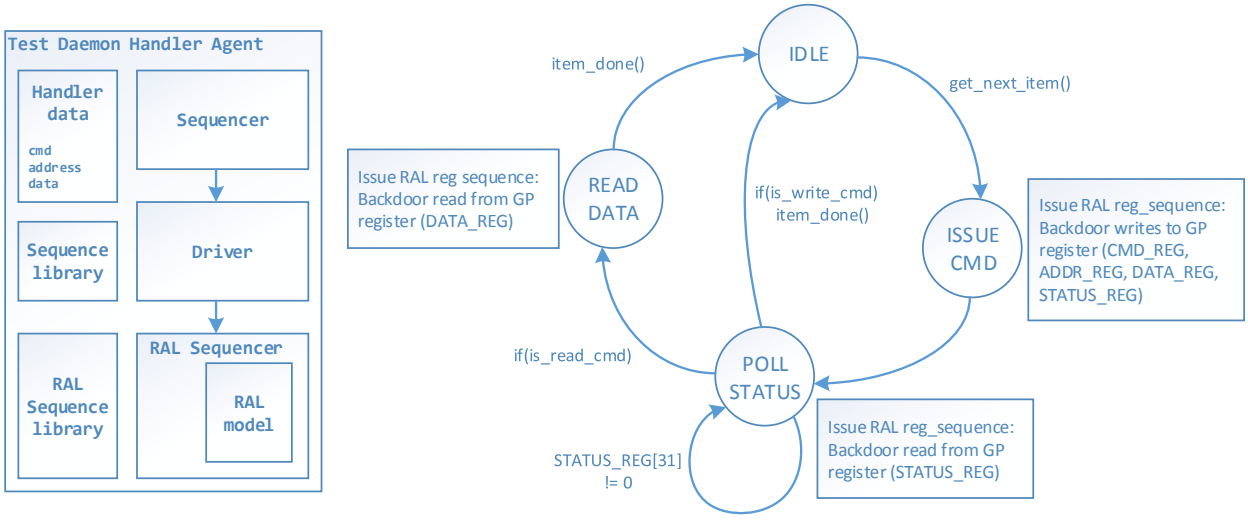

Figure 4 Test Daemon Handler Agent Overview and Driver State Machine

All control of the test daemon executed on the embedded processor is handled by a typical OVM/UVM agent which provides a sequencer and driver component which understands and maintains the state of the program as shown in Figure 4. Conceptually you should think of the driver as being like any other ovm/uvm_driver in any other ovm/uvm_agent. The only difference is that instead of toggling bits on a physical interface, it is executing RAL register sequences to cause a transaction being started by the embedded processor.
To control the test daemon, a small state machine is implemented in the drivers run phase that handles a sequence item holding all information necessary to request execution of a particular task (e. g. read from an auxiliary register). In case a sequence is sending a sequence item to the sequencer, the driver will transition from IDLE to ISSUE_CMD state and will start a RAL

sequence on the RAL sequencer. Once all registers are setup the STATUS_REG will be written, causing the test daemon to transition out of the infinite loop and branch to the instruction to execute, while the driver will transition into the POLL_STATUS state. The STATUS_REG will be cleared once the task has finished and is polled via backdoor path by the driver on every system clock cycle. Once the cleared STATUS_REG has been recognized by the driver it will transition into IDLE state for write commands. In case of a read command, where data needs to be read back from the general purpose registers, it will start another backdoor read sequence in the READ_DATA state before going back to IDLE again. Implementing the access to the test daemon in a state machine avoids conflicting accesses and every task gets executed atomically from the view of the testbench.

4. Control Layer Architecture using RAL frontdoor path

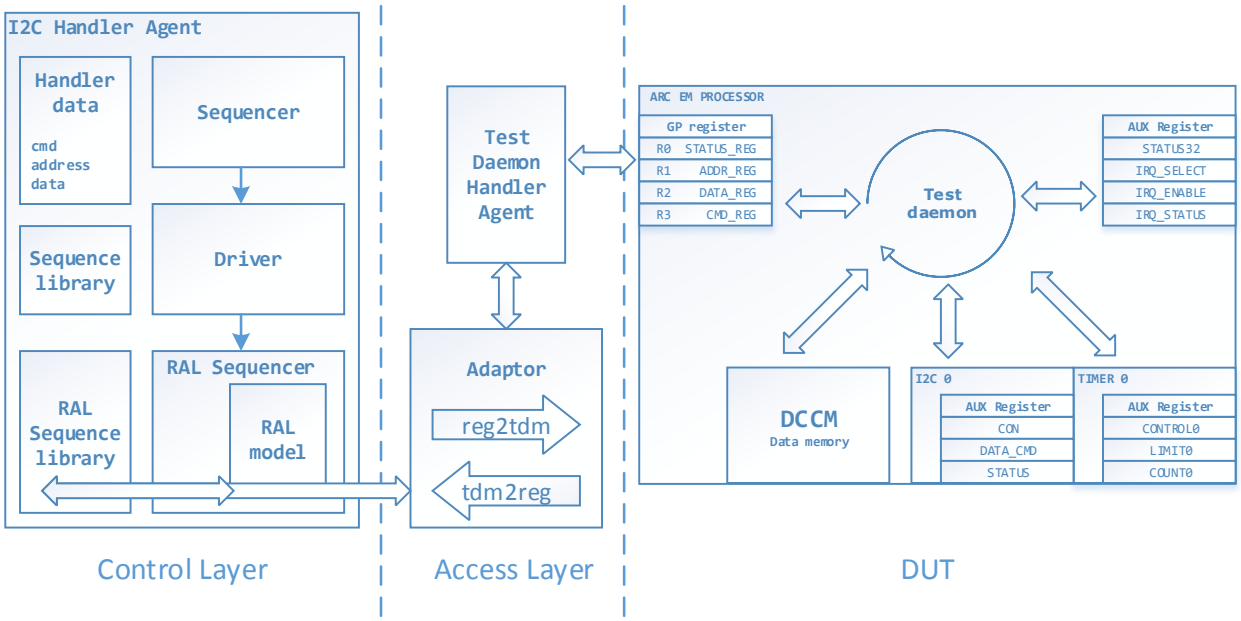

Figure 5 Layered RAL Access

The control layer of our testbench architecture contains various handler agent components for all available resources present in the system that need to be controlled by the testbench. As an example Figure 5 shows an I2C handler agent and its interaction with the access layer to control the DUT. The handler agents are also built on top of RAL operating on a local register model. This register model is configured with the instance specific base address of the register map and the individual register offsets.

Accesses to the registers use the frontdoor access path. This requires a register model adaptor which translates the RAL register access to a specific bus transaction. In our case we want the adaptor to translate into a test daemon task being called, so we use the access layer by starting test daemon handler agent sequences in the adaptor conversion methods. The reason we cannot use a RAL backdoor access in the control layer, like we did in the access layer, is that backdoor read and writes do not account for side effects caused by actual bus transactions. In case of the accesses to the general purpose registers in the access layer, we can cope with this situation as we have full control over these registers at all stages and know how they are used by the test

daemon. So we do not care about any side effects here. For the register model update we use the auto prediction mode of the RAL where the register model gets updated directly when the read and write methods get called.

5. Handling Asynchronous Interrupts in the Environment

Another event we need to consider are asynchronous interrupts, interrupting the regular program execution of the test daemon. As the test environment is the only initiator of transactions and there is no communication path back to the environment in case there is no test daemon task being called, it is difficult to inform the testbench about asynchronous events. Luckily all interrupt sources for the Sensor and Control IP Subsystem are routed through a pin level interface we can observe using a monitor component.
The ARC® EM processor supports level and edge triggered interrupt sources, with the level sensitive source interrupting the program execution as long as the interrupt has not been cleared in the handler. To stop a level sensitive interrupt from constantly jumping into its handler task, we need to turn the interrupt off temporarily. This is necessary to keep the test daemon responsive and allow the test environment to call test daemon tasks to handle the interrupt. It remains the responsibility of the testbench environment to enable the interrupt again after it has been handled appropriately.
The reason we cannot keep the interrupts disabled in the first place and rely on monitoring the external interface are the power management capabilities of the ARC® EM processor. The core supports different power states which can also be controlled via the test daemon and its handler agent and put the processor into different sleep modes. Interrupts are required to exit from these states again and require them to be enabled.
Figure 6 shows the very short generic interrupt handler task which is called for every interrupt source in the test daemon as well as interrupt configuration tasks to allow interrupts to be globally disabled and enabled from the testbench environment.

```
.align 32; seti_cmd:
seti   DATA_REG                ;Global interrupt enable
mov    STATUS_REG, 0           ;Success, clear status register
b      inf_loop                ;Branch back to inf_loop

.align 32; clri_cmd:
clri   0                       ;Global interrupt disable
mov    STATUS_REG, 0           ;Success, clear status register
b      inf_loop                ;Branch back to inf_loop

.align 32; irq_handler:
lr     r8,          [ICAUSE]     ;Get active interrupt
sr     r8,          [IRQ_SELECT] ;Store to IRQ_SELECT to switch
                                 ;banked IRQ config register
sr     ZERO_REG,    [IRQ_ENABLE] ;Disable interrupt
rtie                             ;Return from interrupt
```

Figure 6 Generic Interrupt Handler and Global Configuration Tasks in Test Daemon

Now to ultimately handle the interrupt we instantiate an interrupt handler agent in our environment, containing a pin level monitor component and a virtual sequencer, which holds pointers to all other handler agents in the control layer that need to handle interrupts.

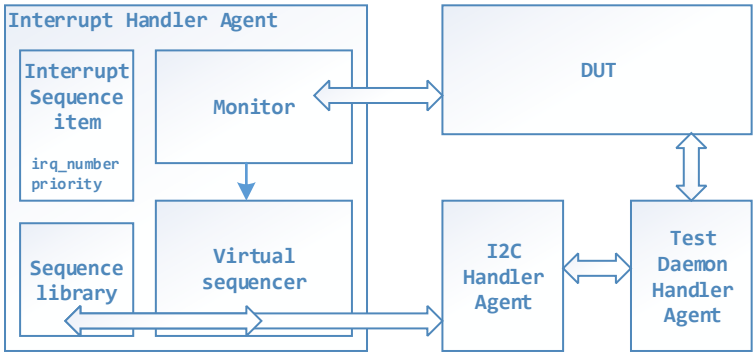

Figure 7 Interrupt Handler Agent

Figure 7 shows an overview of the interrupt handler agent and its connections to the rest of the design. Depending on the raised interrupt that gets monitored on the pin interface, the monitor will start interrupt handler sequences on the virtual sequencer which will then be routed to the corresponding handler agent sequencer. Again the resource specific interrupt handler sequences operate on the RAL register model and the frontdoor register accesses are translated into test daemon tasks using the access layer.

```
function void env::connect_phase(uvm_phase phase)
  ...
  if(intr_handler_agent_cfg.is_active == OVM_ACTIVE)
  begin
    intr_handler_agent_cfg.set_intr_seq(
      I2C_0_ERR,                          //Interrupt src type
      "i2c_handler_agent_err_intr_seq", //Sequence type name
      i2c_handler_agent[0].sequencer);  //Pointer to sequencer
    intr_handler_agent_cfg.set_intr_seq(
      I2C_1_ERR,
      "i2c_handler_agent_err_intr_seq",
      i2c_handler_agent[1].sequencer);
    intr_handler_agent_cfg.set_intr_seq(
      I2C_0_RX_AVAIL,
      "i2c_handler_agent_rx_avail_intr_seq",
      i2c_handler_agent[0].sequencer);
    intr_handler_agent_cfg.set_intr_seq(
      I2C_1_RX_AVAIL,
      "i2c_handler_agent_rx_avail_intr_seq",
      i2c_handler_agent[1].sequencer);
  ...
  end
endfunction : connect_phase
```

Figure 8 Interrupt Handler Sequence Registration

The interrupt handler sequences get registered with the interrupt handler agent in the test environment during the connect phase and are specified per interrupt source connected to the pin interface. Note in Figure 8 that the same interrupt types (e.g. I2C Error Interrupt) for multiple instantiations of the same peripheral type have a common interrupt handler sequence registered to the interrupt agent. In case the interrupts will be raised, this sequence will be executed on the individual handler agent sequencers and operate on the local register model.
As the ARC® EM processor supports different interrupt priority configurations as well as configurable priority thresholds and interrupt triggers, the design of the interrupt monitor is not straight forwards. Depending on the validation requirements it may be necessary to model an accurate interrupt behavior supporting interrupt priorities and preemption.
Our design tries to cope with that by making use of priority based arbitration within OVM/UVM. The handler agent sequencers in the control layer as well as the sequencer in the test daemon handler agent are configured to use the SEQ_ARB_STRICT_FIFO arbitration scheme. This allows interrupt sequences to be executed with their corresponding priority which gets passed to the test daemon handler agent sequencer via the register model adaptor. As a result, register accesses from a low priority interrupt can be preempted by higher priority interrupts by prioritization within the test daemon handler agent sequencer.

6. Results

With the use of an OVM/UVM RAL implementation we were able to design a fully configurable testbench environment for the DesignWare® Sensor and Control IP Subsystem. All relevant resources can be controlled by the testbench and are configured based on constrained random configuration objects which define the behavior of the resources and the execution of transactions during run phase.
The small test daemon executed on the embedded processor allows the test environment to gain access to the otherwise restricted auxiliary register and memory address space and other core functionality like interrupt and sleep state control via predefined tasks.
During implementation the test daemon went through several iterations, primarily to increase performance for executing tasks on the embedded processor. Initially the access layer was not using a RAL backdoor mechanism to modify the internal general purpose registers at all. Instead, commands, data and address information were written to external memories. This however turned out to be too slow in certain scenarios as several loads to memory where required by the embedded processor to decode the task. The current implementation is highly optimized in terms of speed and response time, with the decoding performed by the test daemon handler agent, and only requires a single branch to be executed by the test daemon to jump to the selected task.

Let’s discuss the performance, scalability, maintainability and reusability of the testbench architecture in relation to the configuration options available for the Sensor and Control IP Subsystem.

- ✓ Performance - With the described architecture we are able to create a pure OVM/UVM environment based on RAL without the need to attach any external debugger to the simulator to control the embedded core. This comes with an increase in simulation performance by reducing the time to invoke and execute commands on the processor. An access using an external serial debugger interface would require up to a hundred cycles at

a slower clock speed, where executing the commands directly on the core and calling them from the testbench can be done at full system clock speed and requires around 6 to 10 cycles on average.

- ✓ Scalability - The testbench architecture outlined in this paper had a direct impact on the system use case coverage achieved in our SoC regression tests. While all unit level sequences which operate on the RAL register model could be reused on SoC level, also RAL level sequences for other components on the SoC could be reused and run on top of the access layer. This allowed shared external resources to be either controlled by the Sensor and Control IP Subsystem or the host processor in the test environment without any modifications to the handler sequences.

- ✓ Reusability - RAL provides abstraction from the physical address as well as abstraction from the physical interface. This allows all handler agents in the control layer and its sequences to operate on the RAL register model which only holds the base address of the register map and offsets for individual registers. The Sensor and Control IP Subsystem has a configurable number of peripheral components, which may vary between projects. The testbench architecture allows multiple handler agents to be instantiated in parallel and requires only the instance specific pointer to the register model to be set correctly. This simplifies the design and configuration of the VIP and the test environment.

- ✓ Maintainability - In terms of maintainability we are not dependent on any static assembly or C-code pre-compiled and executed on the embedded processor to model complex scenarios on unit or SoC level. Instead the testbench allows us to control all resources of the subsystem using the register abstraction layer on top of the access layer.

7. Conclusions

The testbench architecture outlined in this paper were used in a real project. In fact it was only during the execution of the project that we started to develop the two layer approach to control the Sensor and Control IP Subsystem and its resources. It became increasingly important and necessary once the unit level environment was brought up into the SoC environment were more advanced test case scenarios were required, like power state transitioning and external interrupt handling in a dynamic environment. Leveraging an OVM/UVM RAL implementation together with a small always running test daemon program provides an alternative way to interact with the system other than using a debugger port, with a lot of advantages in terms of performance and controllability of the IP.

8. References

[1] Synopsys, “DesignWare Sensor and Control IP Subsystem”, http://www.synopsys.com

[2] Synopsys, “DesignWare ARCv2 ISA Programmer's Reference Manual”

[3] Synopsys, “DesignWare Sensor IP Subsystem I/O Databook”