uvm_objection – challenges of synchronizing embedded code running on cores and using UVM

Yassmina Eliouj, NXP Deutschland GmbH, Munich, Germany (yassmina.eliouj@nxp.com)
Vasundhara Gontia NXP Deutschland GmbH, Munich, Germany (vasundhara.gontia@nxp.com)
Sefa Veske, NXP Deutschland GmbH, Munich, Germany (sefa.veske@nxp.com)
Shripad Nagarkar NXP Deutschland GmbH, Munich, Germany (shripad.nagarkar@nxp.com)
Tobias Thiel, NXP Deutschland GmbH, Munich, Germany (tobias.thiel@nxp.com)
Joachim Geishauser, NXP Deutschland GmbH, Munich, Germany (joachim.geishauser@nxp.com)

***Abstract*—UVM is a commonly used base class library when testbenches are constructed. However, on the SoC, typically there are embedded cores in the design under test which are also used to execute verification code. The completion of the UVM life cycle is done using the UVM objections. On the embedded software side, a similar implicit SoC specific life cycle is running. This SoC life cycle is defined for sure by the various resets, but also by functions like e.g. logic BIST, clock ramp, security – just to name few. Further, as there are typically multiple cores on the SoC and they themselves have their independent life cycles, the embedded cores among themself need to be synchronized. The completion control from the tests running on the embedded cores in junction with the UVM testbench is topic this paper.**
(Style: Abstract)

Keywords— UVM; objections; verification; software driven verification; vertical reuse; SoC

INTRODUCTION

The complexity of a current SoC not only originates from the large number of peripherals, but even more from the growing number of compute cores. This creates new challenges for functional verification, as scenarios for different boot cores, core enable sequences, power and reset scenarios need to be covered. As a result, one single verification sequence can contain multiple software threads executed on different cores, either in parallel, or sequentially.

The UVM objection model is not the optimal fit for such scenarios containing multiple software threads. It was built with one or more linear sequences in mind. A software thread can be started at any time in the functional sequence, it may come to a regular end or terminate early, e.g. through a full or partial reset of the SoC. So, an additional mechanism needs to be implemented that collects and checks the status of all running software threads.

Determining when the simulation sequence is finished is not easy, but essential. If the finish criteria are missed, the simulation will hang as an objection doesn't get dropped. This failure type is unwanted but easily detectable. It can be much worse if the finish criteria are reached too early, as in this case the simulation may end early reporting a run with no errors while the verification objective has not been met.

USE CASE SCENARIO

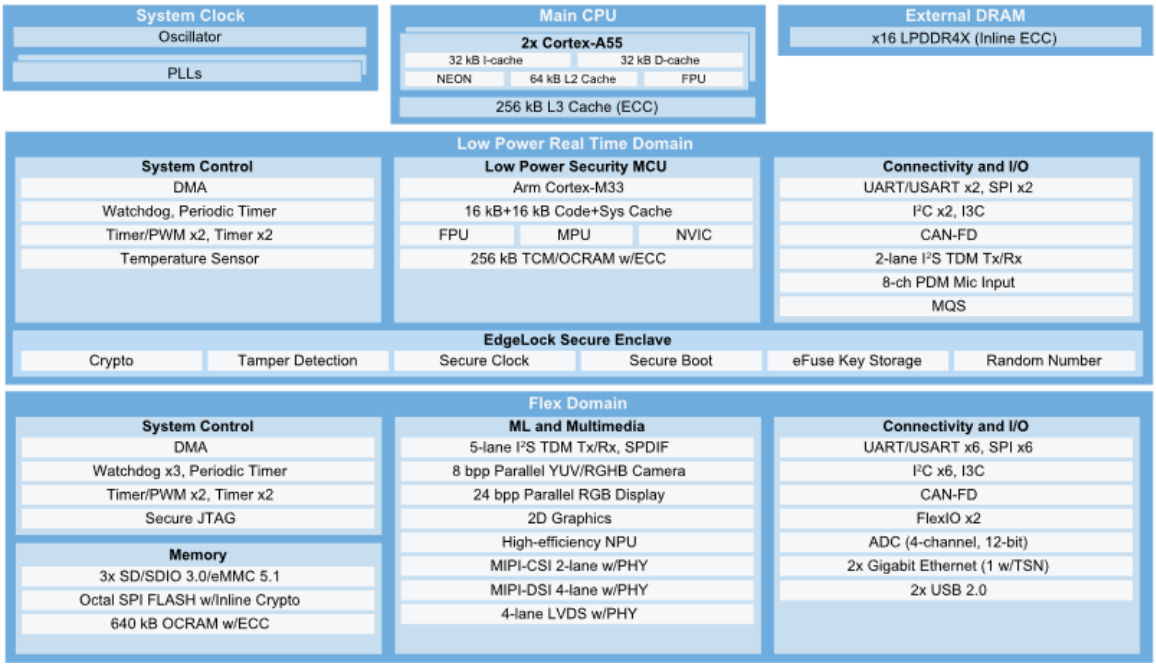

Figure 1. NXP Device Block Diagram

As a use case for this paper a NXP device is used as it contains multiple cores as well as a UVM based testbench.

UVM OBJECTIONS

As the complexity of the integrated circuit designs continues to grow, efficient and accurate verification methodologies become more important to ensure correctness of the designs. One of the key components of the UVM is the objection mechanism which plays a vital role in managing the end-of-test scenario in the verification environment. By raising and dropping objections, components in the testbench can share counter in order to ensure that all tasks have been executed before the simulation terminated. This mechanism prevents abnormal termination of the simulation. The objection mechanism works within UVM component hierarchy, allowing for controlling the simulation end condition. Also, the objection mechanism is an integral part of the UVM which makes it easy reusability and consistency across different testbench. One of the important features of the objection mechanism is traceability by recording the history of raised and dropped objections, which can be helpful in identifying issues while debugging.

Basics of UVM objection

Objection controlling mechanisms:

Methods for raising and dropping objections and for setting the drain time:

```
raise_objection (uvm_object obj = null, string description = "", int count = 1)
```

Raises the number of objections for the source object by count, which defaults to 1. The raise of the objection is propagated up the hierarchy, unless `set_propagate_mode(0)` is used, in which case the propagation is directly to uvm_test_top.

```
drop_objection (uvm_object obj = null, string description = "", int count = 1)
```

Drops the number of objections for source object by count, which defaults to 1. The drop of the objection is propagated up the hierarchy. If the objection count drops to 0 at any component, an optional `drain_time` and that component's `all_dropped()` callback is executed first. If the objection count is still 0.
After this, propagation proceeds to the next level up the hierarchy.

```
set_drain_time (uvm_object obj = null, time drain)
```

Sets the drain time on the given object.

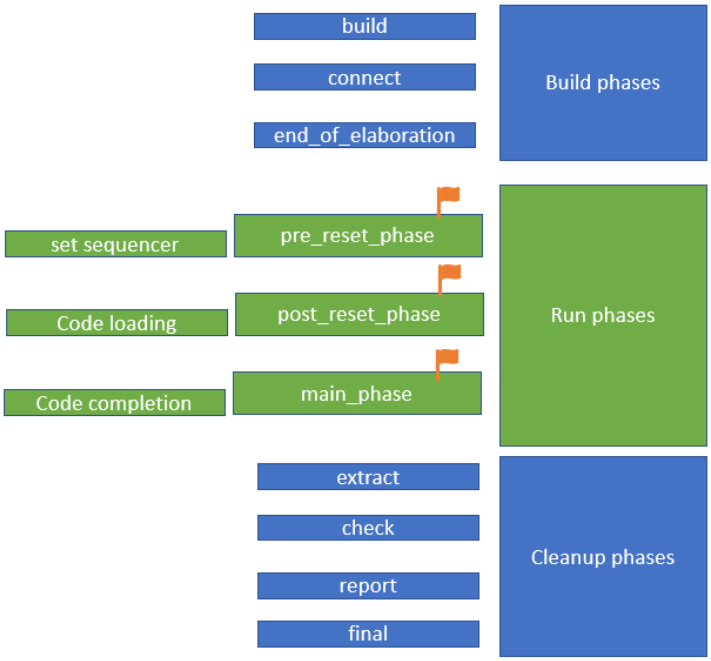

Figure 2. UVM Phases

Objections are flags that keep an existing phase active. Based on `raise_objection()` and `drop_objection()`, the UVM testbench counter will increase or decrease the overall objection count. The object will be drained for the duration of the time frame indicated by the `set_drain_time()` method. This method will synchronize all uvm_components and is mostly used to extend the simulation time after all objections have been dropped.

A UVM testbench architecture, if using the standard phasing, has `phase.raise_objection()` and `phase.drop_objection()` methods inside the run phase and in between these two methods, we start the sequence on to the sequencer. Hence, the objection mechanism helps the simulator to decide at what point of time it can proceed to extract phase from run phase.

CORE EXECUTION

On the C side, the embedded code is running on a core in four phases shown in the picture below:

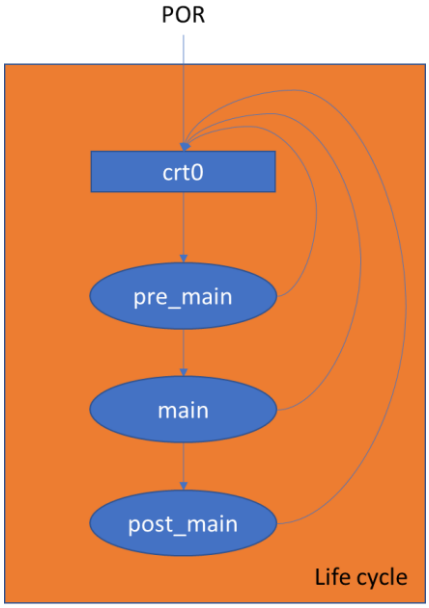

Figure 3. Core life cycle

1. ***CRT ()***
   - Once the core comes out of reset, certain configuration needs to be done handled in assembly code during this phase CRT (C Run Time 0)
   - Specific registers are then initialized inside the CPU, along with setup of stack pointer and vector table base.
   - Caches are initialized, and then error counters.

2. ***Pre Main ()***
   - All activities that are common to that SoC, such as configuring system clocks at maximum frequency (Power up of PLLS) is done by the boot core: Low Power Security MCU (Figure 1). This latter brings up the main CPU, or other application cores in case of a multi-core system, out of reset depending on target scenarios.
   - All interrupts in the SoC are then enabled. A default handler is installed for each interrupt that prints a fatal message in case of generation of an unexpected interrupt, then the test will end in this case.
   - Clocks of all IPS are enabled based on arguments to make a test case reusable for all SoC.
   - In Pre-main, the CPU will use the `RAISE_OBJECTION` call to raise a UVM objection and blocks the core in this phase until the UVM sequence is finished.

3. ***Main ()***
   - This will call the main function in the stimulus.

4. ***Post Main ()***
   - The correspondent `DROP_OBJECTION` is then called in `post_main()`. The master core will wait in this phase until all objections are dropped from the other slave cores using the `GET_OPEN_OBJECTIONS` call.
   - System health checks are done, then the test case will end.

As the SoC does not consist of one core, the execution on the cores can also be captured in a life cycle picture.

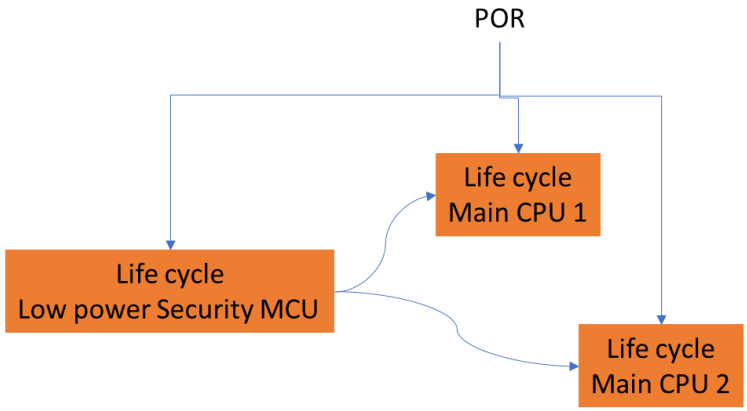

Figure 4. SoC Cores life cycle

USE CASE

Vertical Verification Reuse

Taking the UVM objection basics, the solution must address the following use case. Today's SoC has become so complicated that the integration of all building blocks cannot be done at the SoC level. As can be seen in Figure 1, todays SoC consists of multiple functional blocks. These blocks are used to start integration verification. Some of the blocks consist of very complex IPs which are also part of the use case the objections can be used for. The use case that shall be addressed by the objection setup described is the vertical verification reuse. If the case is considered that a complex IP is used in a subsystem, then the complex IP comes with a UVM environment that instantiates the solution shown in the next section and calls the API functions defined in the section CAPI to control the end of test process. The integration verification pattern provided by the complex IP is then used in the subsystem along with the subsystem specific configuration and control code for this complex IP to verify the integration of the IP. In this simple case, the reuse of the complex IP UVM environment along with the reuse objection mechanism functions as a self-containing unit. On top of the integration verification, it is often required to verify that multiple blocks within a subsystem work together as specified. For this verification, the integration verification pattern of e.g. the complex IP and additional pattern written patterns which are specific for the subsystem need to run concurrently. For this case, the UVM objection mechanism determines the overall end of test state of UVM sequences and the C-code running on the embedded cores. Figure 4. SoC Cores life cycle depicts the use case to run C-code on multiple CPUs, not shown in the picture are UVM sequences that run on the UVM side with contribute to the use of the UVM objection concept to the overall end of test definition.

Health Check

On top of the vertical reuse, a function at the end of the testcases to check that there are no unwanted side effects created needs to be performed. An example of this is for example the content of safety registers that could have been triggered by the test unintentionally through the creation of a single fault ECC error. To perform this check, it is important that all testcases running on embedded cores are done. The function GET_OPEN_OBJECTIONS allows to determine in the `post_main` stage of an embedded C testcase that cores that can create unwanted side effects are done with their testcases. The code example shows the use of this API.

```
void post_main ()
{

  uint32_t open_objections = 2;

  while (open_objections > 1)
  {
     GET_OPEN_OBJECTIONS(&open_objections);
  }
```

```
    do_health_check();

    DROP_OBJECTION("MAIN_CPU");
  }
```

The variable `open_objections` is initialized to 2 to not leave the while loop at the first loop. The while loop is terminated, once the return of the `GET_OPEN_OBJECTS` is 1, which means that only the MAIN_CPU has raised an objection.

SOLUTION

Architecture

The UVM implementation provides a UVM environment to be placed into the testbench. The goal is to support also the hierarchical reuse use case. This means the objection handling can be added to a subsystem testbench and this subsystem testbench can then be reused in e.g. the SoC testbench by instantiating the subsystem testbench.

The Figure 2 shows the relation between the UVM phases on the right vs the core phases on the right denoted with the headline CAPI.

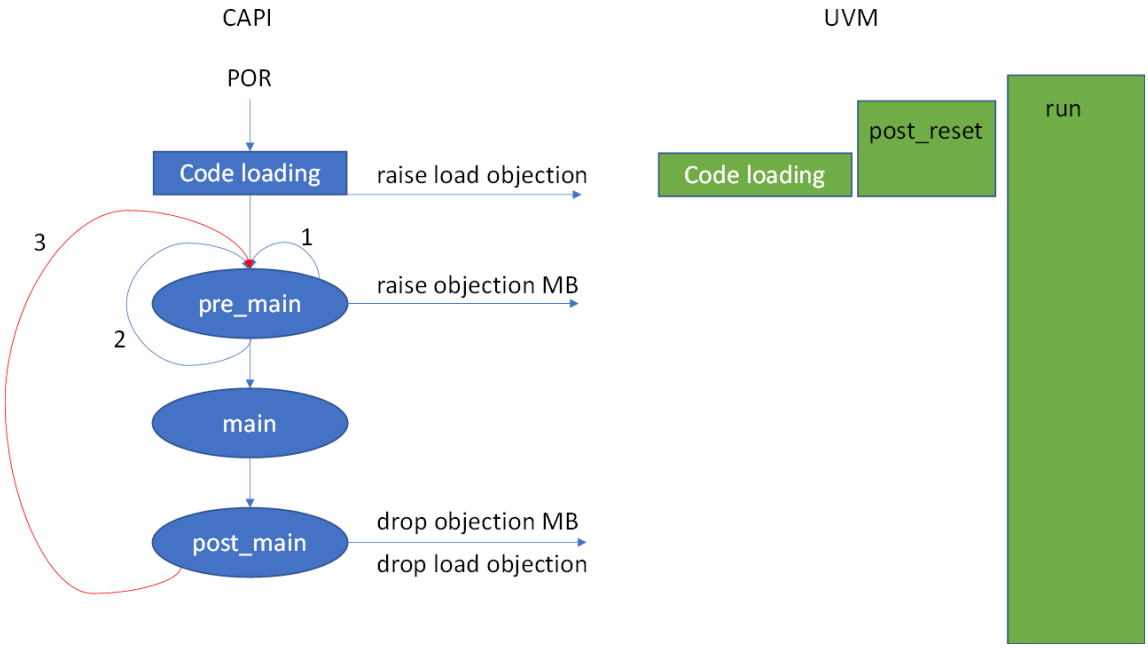

Figure 5. . CAPI execution during UVM run phase

Figure 5 shows the different scenarios. On the left side, the CAPI execution is shown with the different functions used on the CAPI side. On the right, the UVM phases are shown.

Cases:

1. Reset happens before the raise objection mailbox function gets called on the CAPI side. Here the objection raised through calling the function `objection_env::set_memory_loaded()` keeps the simulation running.
2. A reset happens after the raise objection mailbox was set. The objection is held and the `objection_env::reset_callback()` is called.

3. If a reset happens after the drop objection mailbox call, no objection is active until the raise objection mailbox call. This time needs to be bridged by an external objection, e.g. the global UVM test objection.

UVM Side

The UVM implementation can be represented by the following class diagram:

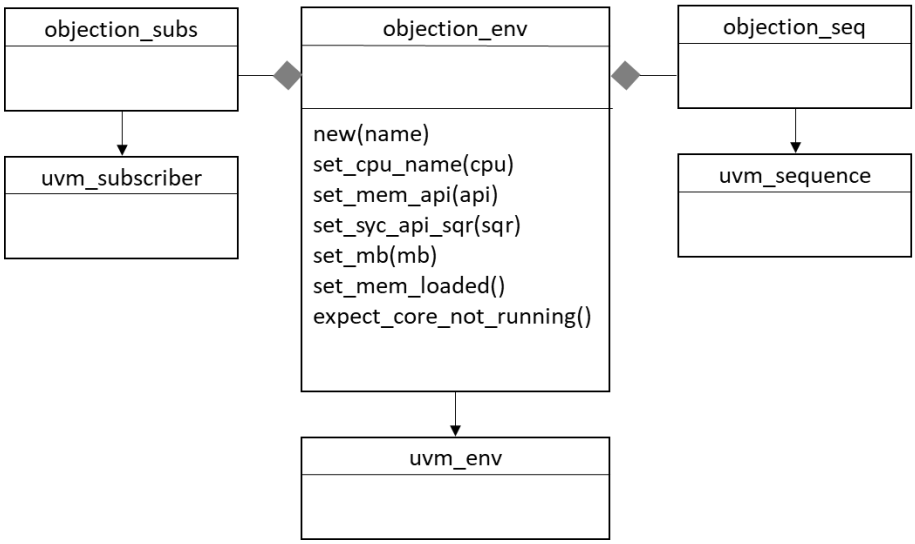

Figure 6. UVM Class Diagram

UVM has two sides of termination process where one of them is raise/drop objection and the other is memories loading. A configuration object should provide the CPU name that needs to be identified in the hex file loader. In the example below the CPU name is hard coded to simplify the code. With the function call `set_mem_api`, the NXP memory driver API is passed to the objection environment. The NXP memory driver API hereby allows the UVM objection environment to fetch strings, such as the CPU name string, from the embedded memories. Details on the NXP memory driver API can be found in [1]. The function `set_sync_api_sqr` is called to pass the UVM virtual sequencer to the objection environment which enables the communication via the UVM code and the C code running on the embedded core. From this communication infrastructure, a mailbox function is used. The mailbox number to be used is set by the function `set_mb`. On the UVM side, the following code needs to be added:

```
class nxp_device_env extends uvm_env;
    // …

    /* Core Objection Handling */
    objection_env m_objection_env;

    virtual function void build_phase(uvm_phase phase);
        // …
        begin: CORE_OBJECTION_BUILD
            m_objection_env = objection_env:: type_id::create ("m_objection_env", this);
            m_objection_env.set_mem_api(main_cpu_api);
            m_objection_env.set_sync_api_sqr(main_cpu_sync_api);
            m_objection_env.set_mb(main_cpu_mb);
            m_objection_env.set_cpu_name("MAIN_CPU");
        end: CORE_OBJECTION_BUILD
        // ...
```

```
    endfunction : build_phase

    virtual task post_reset_phase(uvm_phase phase);
        // …
        if (m_memory_load_seq.m_core_running)
           m_objection_env.set_memory_loaded();
    endfunction : post_reset_phase
endclass : nxp_device_env
```

The UVM code in the `post_reset_phase` detects if an image for the core has been loaded via the variable `m_core_running` in the `m_memory_load_seq` sequence. Only if the core is actually running, as there was also code loaded, the method `set_memory_loaded` will be called and the objection `mechanism` is activated. If there is a raise or drop of an objection signaled to this core without the `set_memory_loaded` function being invoked, will cause an error message. There is also an error raised, if the function `expect_core_not_running` is called and a raise or drop of objection is flagged by the embedded core.

For every core an objection environment needs to be instantiated. As the e.g. imx93_env UVM environment is instantiated in the next level up testbench, the objection handling for this core is also available there.

CAPI

The embedded core will use the following C API calls to raise and drop objections on the UVM side to control completion of the UVM life cycle.

It implements the API `RAISE_OBJECTION` and `DROP_OBJECTION`. The implementation consists of two parts, the CAPI side and the UVM side. On top of this, the function `GET_OPEN_OBJECTIONS` is called by the master core C side that returns the number of open objections at the end of simulation.

RAISE_OBJECTION

The macro, `RAISE_OBJECTION` raises an objection on the UVM side and blocks the testbench to finish simulation.

An example on how this API can be used in the embedded code in the `pre_main()` is shown below:

```
void pre_main ()
{
  RAISE_OBJECTION("MAIN_CPU");
}
```

Here, MAIN_CPU is the string of the core an objection shall be raised for.

DROP_OBJECTION

The macro, `DROP_OBJECTION` drops an objection on the UVM side and allows the testbench to finish simulation.

An example on how this API can be used in the embedded code in the `post_main()` is shown below:

```
void post_main ()
{
  DROP_OBJECTION("MAIN_CPU");
}
```

Here, MAIN_CPU is the string of the core an objection shall be dropped for.

GET_OPEN_OBJECTIONS

The macro, `GET_OPEN_OBJECTIONS` returns the number of open objections raised by this infrastructure in the simulation. Once the master core sees that all objections are dropped by other running CPUs, it will do the required health checks and drops its objection on the UVM side.

An example on how this API can be used in the embedded code in the `post_main()` is shown below:

```
void post_main ()
{
  uint32_t open_objectons;
  ...
  GET_OPEN_OBJECTIONS(&open_objections);
  ...
  DROP_OBJECTION("MAIN_CPU");
}
```

Here, MAIN_CPU is the string of the master core an objection shall be dropped for.

CONCLUSION

The methodology described here was developed over the course of several years and has by now been successfully applied in several projects, both on SoC level verification, subsystem verification, and IP level verification. It provides an integrated solution that links the UVM and C components of the verification environment. Using standard UVM features in combination with an API on the C side allows application in various setups, with both simulated and virtual cores of different make and model. Today we apply it as a reusable building block that significantly simplifies the creation of even the most complex test scenarios. A building block we wouldn't want to miss.

REFERENCES

[1] uvm_mem – challenges of using UVM infrastructure in a hierarchical verification, DVCon 2022 Europe