2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

uvm_mem – challenges of using UVM infrastructure in a hierarchical verification

Joachim Geishauser, NXP Deutschand GmbH, Munich, Germany (*joachim.geishauser@nxp.com*)
Aditya Chopra, NXP Semiconductors India Pvt Ltd, Noida, India (*aditya.chopra@nxp.com*)
Stephan Ruettiger, NXP Deutschand GmbH, Munich, Germany (*stephan.ruettiger@nxp.com)*
Luca Rossi, NXP Semiconductors France, Mougins, France *(luca.rossi@nxp.com*)
Sanjay Kakasaniya, NXP USA INC, Austin, USA (*sanjay.kakasaniya@nxp.com*)
L.N. Zhang, NXP Qiangxin (Tianjin) IC Design Co., Ltd, Tianjin, China*(lina.zhang1@nxp.com)*

***Abstract*—Along the line of increasing amount of Software that amount of memories increase. On the verification side most of these memories need to be peeked and poked into. The Universal Verification Methodology (UVM) provides for this purpose a class called `uvm_mem` which can be used for this. Within NXP a framework called "NXP memory driver API" was created that leverages the UVM infrastructure to load e.g. an image into the memories of a SoC. As today's SoC grow in complexity, the verification of those SoCs is moving away from SoC only verification to an hierarchical approach. The hierarchical verification allows to manage the complexity and is also better aligned to the overall development process. This hierarchical verification imposes on the "NXP memory driver API" infrastructure to support this style as well. Here, the UVM infrastructure shows weaknesses for which the paper will show work a rounds. The paper will also show how to use such infrastructure to speed up the bring up of emulating the design under test.**
(Style: Abstract)

Keywords—UVM; memory; verification; TLM2; GP; emulation

INTRODUCTION

The increasing complexity of today's SoC include not only Hardware, but also Software. With this increase the amount of memory for the various cores on the SoC also increase. Shrinking geometries along with safety requirements enforce the implementation of ECC on the memories as well as bit line reordering. As a result, backdoor loading of memories is no straight forward implementation anymore. Beside the need to improve the reuse of the complex memory backdoor loading drivers, the simulation performance becomes a bottleneck. The same effect can be seen on the design side, as the construction of these complex SoCs from simple blocks is difficult to be handled. Thus the design is using subsystems to assemble complex SoCs. Verification along the subsystem approach, does also increase the simulation performance, but imposes additional effort to create these additional testbenches. Therefore, it is important to build the testbenches in a most efficient way to allow as much code to be reused at the next level. Another approach to address the simulation performance is to move the design under verification into an emulation system. In the emulation system the loading of the memories is the first task on the infrastructure side after the design was compiled for the emulator. Here again the best practice is to reuse the existing loading infrastructure on the simulation environment to save setup time on emulation side.

USE CASE SCENARIO

To be able to understand the details, the use cases need to be described. The uses cases are also important to understand the disconnect and the limitations between the current implementation and the one documented in this paper. The use cases contain two cases which show up when looking at the distinct types of subsystems to build

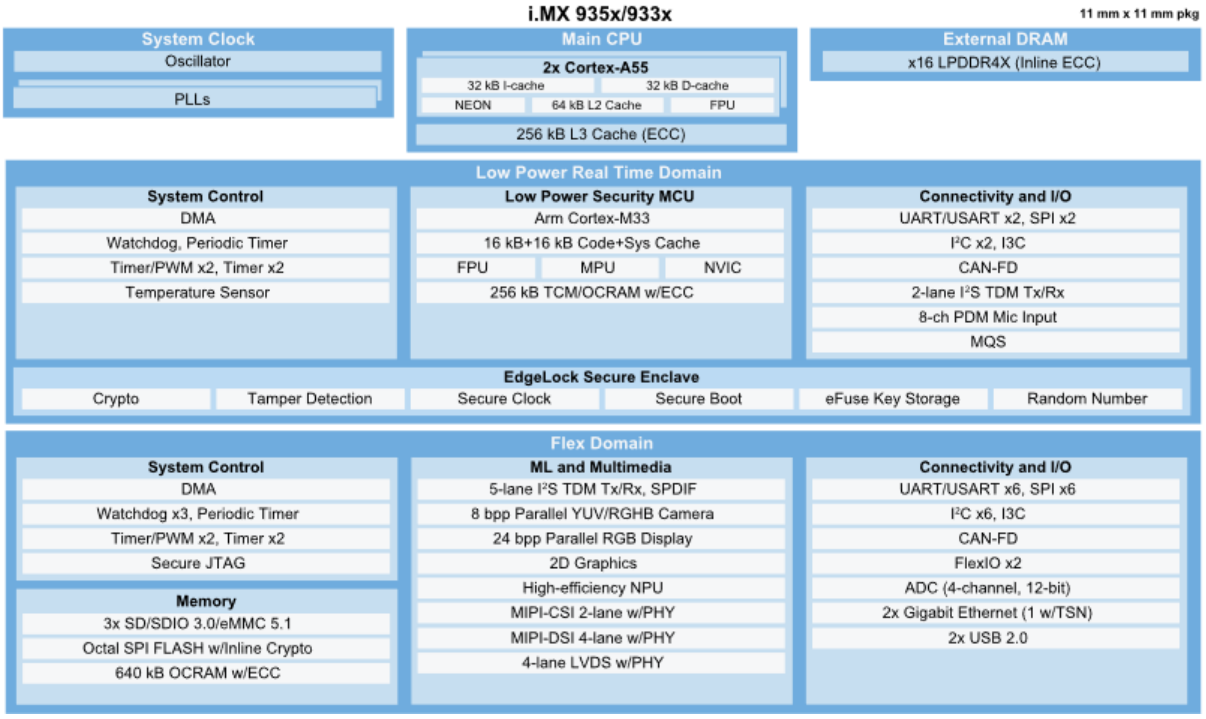

Figure 1. i.MX 93x Block Diagram [1]

SoCs. As an example for the use case the NXP device i.MX93x is used to showcase the cases. Figure 1 shows the block diagram of such device.

The first type of use case is the type of subsystems that contain a core, or more general expressed a bus master, as well as one or more memories. Figure 2 is an example of such subsystem from the i.MX93x block diagram. This block contains an Arm Cortex-M33 core along with 256 kB TCM RAM with ECC.

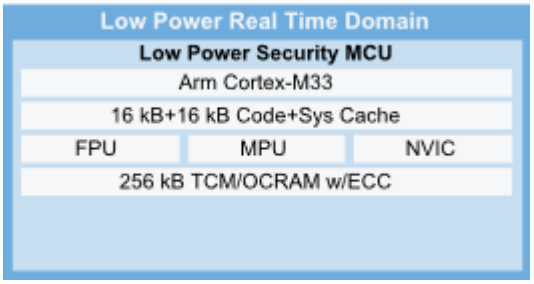

Figure 2. Bus master with memory subsystem

The second type of subsystems is defined to contain one or more memories but no bus master. This case is shown in the figure below. This memory subsystem contains 640kB OCRAM with ECC.

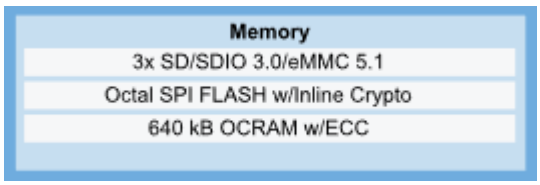

Figure 3. Subsystem with memory

Out of these subsystems, SoCs can be assembled and the described infrastructure allows to create the required infrastructure therefore.

Basics

One of the big advantages of UVM is that the internet can be used to query for questions on how to solve verification problems in UVM. In most areas, this provides plenty of hits, but especially when it comes to the `uvm_mem` usage, the results are rather thin. This observation was another motivation to create this paper. The `uvm_mem` is an extension of the `uvm_object`, therefore it is not a UVM component, and it will not be phased like a component. This means that the `uvm_mem` does not have the standard methods like e.g. build, connect. The closest class in the UVM framework to the `uvm_mem` is the `uvm_reg` class, which is used to model design registers in the testbench. Like the `uvm_reg` the `uvm_mem` support front door and back door access mechanism to provide redundant paths to the register and memory implementation and verify the correctness of the decoding and access path, as well as increased performance after the physical access paths have been verified. In this paper we concentrate on the backdoor function as this is used to load e.g. a compiled executable into the memory. The front door to the memories in this paper is done via C code running on an embedded core or via a special sequence that executes the C code on the workstation and leverages a bus functional model to access the memory via the front door.

There is some reference code which shows a basic example of a standard UVM infrastructure setup for `uvm_mem`. A register block class `root_reg_block` is defined. It represents a design hierarchy which contains an address map `uvm_reg_map` accessible via a specific physical interface and a memory modelized with the memory abstraction base class `uvm_mem` accessible via the address map. First the `m_root_mem_map` address map is created with the `create_map` method of the `uvm_reg_block` base class. The `create_map` method allows to configure the base address of the map which will be used as an offset for all the registers, memories and sub-blocks within the map, the byte width of the bus for which this map is used and the endianness. Then a new instance of `uvm_mem` base class named `m_root_mem` is instantiated. The `size` determines the number of memory locations, *`n_bits`* specifies the number of bits in each memory location and `access` sets the access policy for the memory. The parent block `m_root_mem` for the `uvm_mem` is then configured and the backdoor is created. As last step the newly created memory `m_root_mem` is added to the address map `m_root_mem_map` at a specific offset.

```
class root_reg_block extends uvm_reg_block;
    uvm_reg_map    m_root_mem_map;
    uvm_component m_parent;
    uvm_mem  m_root_mem;
    function new (string name = "root_reg_block",
                  int has_coverage = UVM_NO_COVERAGE);
        super.new (name, build_coverage(UVM_CVR_ADDR_MAP));
    endfunction : new
    function void build ();
        m_root_mem_map = create_map (.name("m_root_mem_map"),
            .base_addr(0),
            .n_bytes(4),
            .endian(UVM_LITTLE_ENDIAN));
```

```
            begin : M_ROOT_MEM_BUILD
                m_root_mem = new (.name({get_name(),".m_root_mem"}),
                    .size((128*1024)/(32/8)),
                    .n_bits(32),
                    .access = "RW");
                m_root_mem.configure (root_reg_block, "");
            end : M_ROOT_MEM_BUILD
            begin : M_ROOT_MEM_MAP
                m_root_mem_map.add_mem (.mem(m_root_mem),
                    .offset('h42000000));
            end : M_ROOT_MEM_MAP
        endfunction : build
endclass : root_reg_block
```

In the UVM `top_env` the register block class `root_reg_block` is used to create a design hierarchy in which the `m_root_reg_block` contains 2 sub-blocks `m_a_reg_block` and `m_b_reg_block`, derived from the same class and mapped to different memory addresses of the `m_root_reg_block uvm_reg_map`. The register blocks are created using the build construct described before and then added to the memory map of the `m_root_reg_block`. The `m_a_reg_block` is added through the `add_mem` at the offset `32'h4200_0000`. A memory can be added to multiple address map if it is accessible by different interfaces, however the address maps to which is added need to have the same parent block of the memory. Instead the memory map of the block `m_b_reg_block` is set through `add_submap` method as a submap of `m_b_reg_block` at address `32'h0042_0000`. As last step the register blocks are locked to allow the computation of the final memory map

```
class top_env extends uvm_env;
    root_reg_block  m_root_reg_block;
    root_reg_block  m_a_reg_block;
    root_reg_block  m_b_reg_block;
    virtual function void build_phase (uvm_phase phase);
        begin: ROOT_REG_MODEL_BUILD
          m_root_reg_block = new("m_root_reg_block");
          m_root_reg_block.m_parent = this;
           m_root_reg_block.build();
        end : ROOT_REG_MODEL_BUILD
        begin: A_REG_MODEL_BUILD
           m_a_reg_block = new("m_a_reg_block");
          m_a_reg_block.m_parent = this;
```

```
        m_a_reg_block.build();
      end : A_REG_MODEL_BUILD
      begin: B_REG_MODEL_BUILD
        m_b_reg_block = new("m_b_reg_block");
        m_b_reg_block.m_parent = this;
        m_b_reg_block.build();
      end : B_REG_MODEL_BUILD
endfunction : build_phase

function void connect_phase (uvm_phase phase);
    begin : HDL_ROOT
      m_root_reg_block.set_hdl_path_root("testbench.top");
    end : HDL_ROOT
    begin : CONNECT_TOP_A_REG_MODEL
      m_a_reg_block.configure(.parent(m_root_reg_block),
                              .hdl_path("a_wrapper"));
      m_a_reg_block.m_root_mem_map.add_mem(.offset(32'h4200_0000));
    end : CONNECT_TOP_A_REG_MODEL

     begin : CONNECT_TOP_B_REG_MODEL
       m_b_reg_block.configure(.parent(m_root_reg_block),
                               .hdl_path("b_wrapper"));
       m_b_reg_block.m_root_mem_map.add_submap(
                               m_a_reg_block.m_root_mem_map,
                               32'h0042_0000);
     end : CONNECT_TOP_B_REG_MODEL
endfunction : connect_phase

virtual function void end_of_elaboration_phase (uvm_phase phase);
    begin: A_REG_MODEL_LOCK
      m_a_reg_block.lock_model();
    end: A_REG_MODEL_LOCK
    begin: B_REG_MODEL_LOCK
      m_b_reg_block.lock_model();
    end: B_REG_MODEL_LOCK
```

```
        begin: ROOT_REG_MODEL_LOCK
          m_root_reg_block.lock_model();
        end: ROOT_REG_MODEL_LOCK
    endfunction : end_of_elaboration_phase
endclass : top_env
```

Based on the topology and complexity of the design and of the physical interfaces, one or more memories could be added to the address map, and a register block could contain one or more address maps.

Limitiations

As outlined in [2], the UVM memory infrastructure is limited as it has the model in mind that there is one design hierarchy with one viewpoint into the system. This shows up as soon as a memory is added to two maps that belong to different blocks to start with. When you do this, the following code will error out

```
if (mem.get_parent() != get_parent()) begin
   `uvm_error("RegModel",
      {"Memory '",mem.get_full_name(),"' may not be added to address map '",
       get_full_name(),"' : they are not in the same block"})
   return;
end
```

Same if you want to add a map multiple time, an error will be flagged by the code below

```
if (m_parent_map != null) begin
   `uvm_error("RegModel",
       $sformatf("Map \"%s\" already a submap of map \"%s\" at offset 'h%h",
                 get_full_name(), m_parent_map.get_full_name(),
                 m_parent_map.get_submap_offset(this)));
   return;
end
```

or

```
`uvm_error("RegModel",
  {"Submap '",child_map.get_full_name(),"' may not be added to this ",
  "address map, '", get_full_name(),"', as the submap's parent block, '",
  child_blk.get_full_name(),"', is not a child of this map's parent block, '",
  m_parent.get_full_name(),"'"})
```

In the regression we run an UVM warning will also make the testcase fail by the code below

```
`uvm_warning("RegModel",
    {"Memory '",get_full_name(),"' is not contained within map '",map.get_full_name(),"'",
     (caller == "" ? "": {" (called from ",caller,")"})})
```

The above code sections are just a few examples of error and warnings you can get when trying to use the standard UVM code for your use case.

NXP MEMORY DRIVER API

Architecture

The NXP memory driver API leverages the UVM memory infrastructure by providing a defined Application Programming Interface (API) to the stimulus as well as defining how a memory driver needs to be implemented. The interface to the stimulus is called frontend API in this paper. The memory drivers are implemented in the

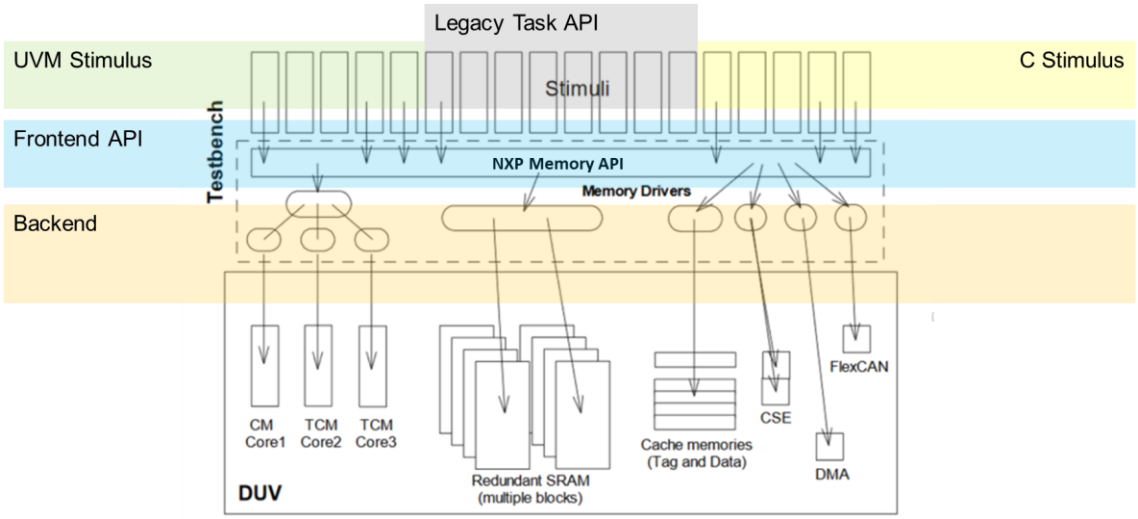

Figure 4. Memory Infrastructure Overview

backend infrastructure of NXP memory driver API. Figure 4 gives an overview of the architecture. The figure shows the division into a frontend and backend part of the infrastructure. Along with the division, there are two APIs defined. The two parts are then connected via a Transaction Level Model 2 (TLM2) generic payload (GP). The TLM2 GP connection between the frontend and the backend was chosen to support connection into system level models as well as a potential hook to emulation systems. Additional information that need to be passed is carried via a GP extension.

The front-end class structure with its APIs is shown in Figure 5. The figure shows two parts. The first API is provided with the `mem_api_base` class. This API is used for UVM sequences. On the right side of the picture, an extended `uvm_mem` infrastructure is shown that captures the connection between the `uvm_mem` and the extended `uvm_reg_backdoor`. Further the TLM2 connection is shown in form of using the `m_nb_initiator_socket` as being part of the GP sequencer.

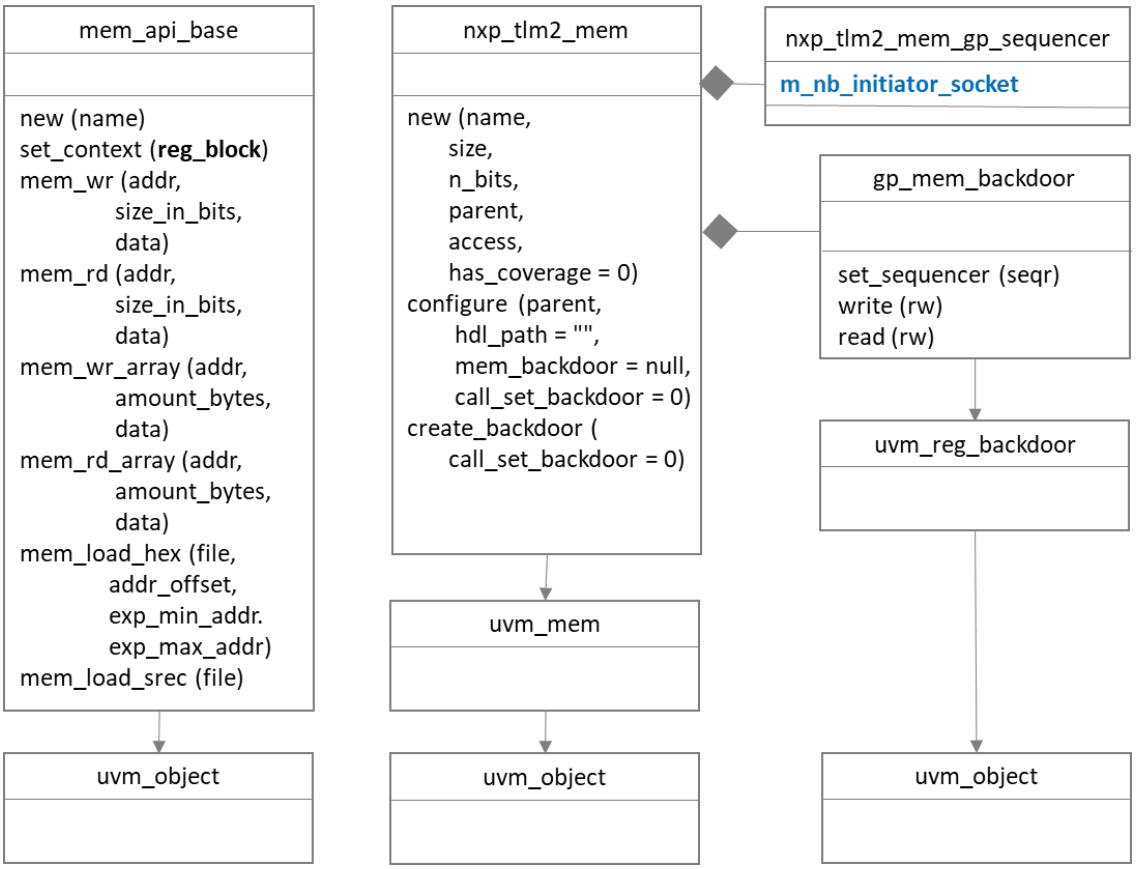

Figure 5. Frontend UML

The connection between the `mem_api_base` and the nxp_tlm2_mem is that the nxp_tlm2_mem is registered to the `uvm_map` as defined by UVM. Along with the UVM definition, the `uvm_map` is registered with an `uvm_reg_block`. This `uvm_reg_block` is passed to the `mem_api_base`. The `mem_spi_base` class is then using the UVM methods to determine which `uvm_mem` to use for a given read or write transaction provided via an API call.

The TLM2 initiator socket is made available using the method `create_backdoor()` on top of the basic UVM memory setup code.

Similar to the frontend class structure, Figure 6 shows the backend UML class diagram. The connection to the frontend is shown in the figure as `m_nb_target_socket`. The figure shows two variants of memory driver that can be used. One is `mem_api_driver` which is typically used for design internal memories. This memory driver uses a behavioral memory model which is provided typically from the memory compiler. The `nxp_mem_gp_driver` is the second option to create a memory driver. This second variant provides already a sparse memory array and is typically used for testbench memories and just needs a front-end access API state machine, like e.g. an AHB interface.

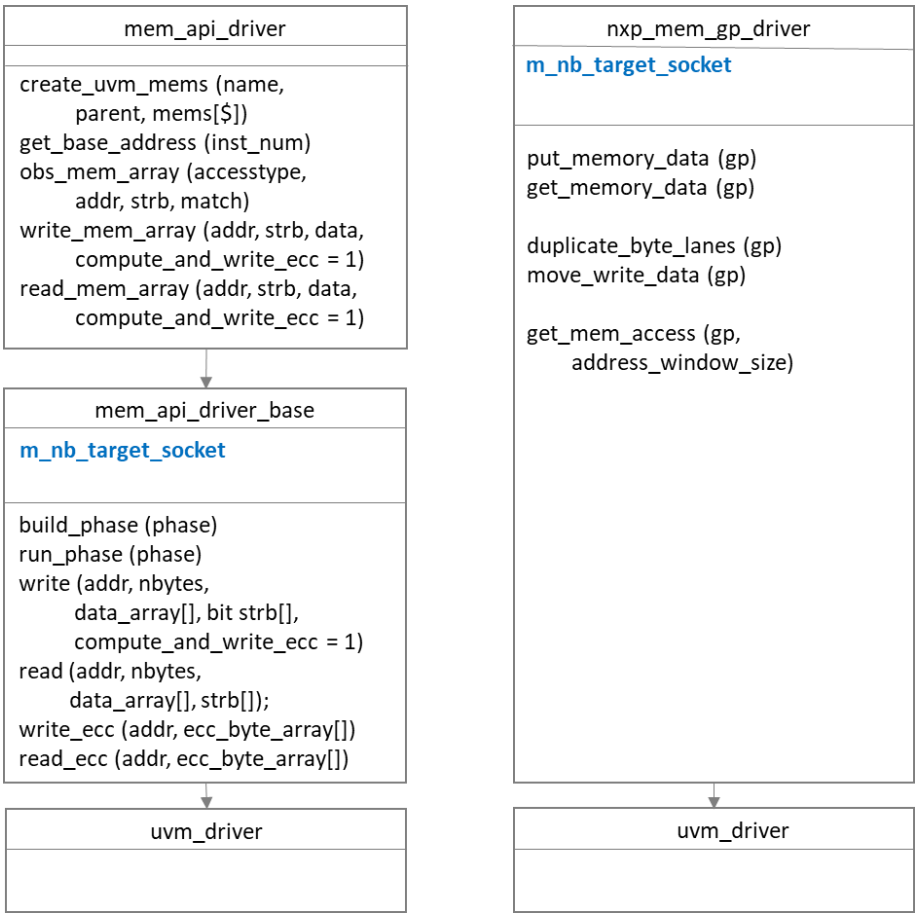

Figure 6. Backend UML

As mentioned earlier, the frontend is connected to the backend infrastructure using a TLM2 socket. This connection is done in the environment class in the connect phase as shown in the code below

```
function void my_env::connect_phase (uvm_phase phase);
    m_a_reg_block.m_a_drv_mem.m_mem_sequencer.m_nb_initiator_socket.connect
         (m_a_mem_drv.m_nb_target_socket);
```

In addition to the connect function, the convenience function `set_backdoor()` as shown in Figure 5, needs to be called in the register model build method to allow the connection to the backend classes via TLM2. This method is made available in the `nxp_tlm2_mem` which is an extension of the `uvm_mem`.

```
begin : M_B_DRV_MEM_BUILD
    m_b_drv_mem = new (.name({get_name(),".m_b_drv_mem"}),
        .size((256*1024)/(32/8)),
        .n_bits(32),
        .parent(m_parent));
    m_b_drv_mem.configure (root_reg_block, "");
    m_b_drv_mem.set_backdoor (m_b_drv_mem.create_backdoor());
end : M_B_DRV_MEM_BUILD
```

To overcome the issues outlined in the Limitiations section for our use cases, the classes shown in **Error! Reference source not found.** have been added.

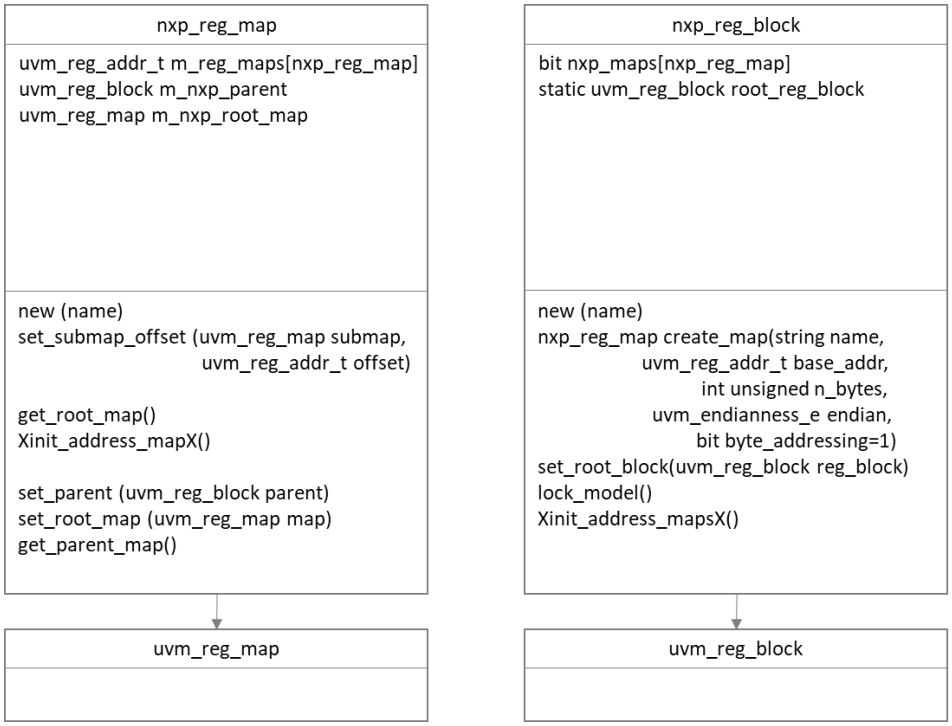

Figure 7. Extended UVM classes

This means that instead of creating an e.g. `uvm_reg_map`, an `nxp_reg_map` needs to be used. These classes provide on top of modified UVM functions new functions. Main objective of the updates are to overcome the single parent concept of the standard UVM implementation. The new functions are divided into functions that need to be called in the user code, and other functions that are used internally. One of the new user code functions is the `set_root_block()` method. As outlined in the Basics sections, there is a method in reg_block called `build()` that creates the reg_map. In this method the `set_root_block()` method needs to be called first. The bold lines show the additional lines that need to be added on top of the basic UVM code

```
function void my_nxp_reg_block::build ();
    set_root_block (this);
    m_top_b_mem_map = create_map (.name("m_top_b_mem_map"),
        .base_addr(0),
        .n_bytes(4),
        .endian(UVM_LITTLE_ENDIAN));
```

The second new user function is the `nxp_reg_block::Xinit_addess_mapsX()`, this function needs to be called to complete the address map calculation. This function is called in the `end_of_elaboration_phase()` function in the UVM environment.

```
function void my_env::end_of_elaboration_phase (uvm_phase phase);
    begin: A_REG_MODEL_LOCK
        m_a_reg_block.lock_model();
        m_a_reg_block.Xinit_address_mapsX();
    end: A_REG_MODEL_LOCK
```

The new internal functions are `nxp_reg_map::set_parent()`, `nxp_reg_map::set_root_map()` and `nxp_reg_map::get_parent_map()`. Methods that have updated functions are `nxp_reg_map::Xinit_address_mapX()` and `nxp_reg_map::set_submap_offset()`.

The function `set_submap_offset()` adds the submap to the `m_reg_maps` array and calls then the base `set_submap_offset()` function. This is done to overcome the checks in the UVM base code that block the use case we have by using the UVM recommended function `add_submap()`. This means further that instead of calling the method `add_submap()` in our code, we use `set_submap_offset()` method instead to allow adding a submap to one or more maps. An example is shown below

```
m_b_reg_block.m_top_b_mem_map.set_submap_offset(
  m_a_reg_block.m_top_a_mem_map, 32'h0A00_0000);
```

The method `Xinit_address_mapX()` calls the base `Xinit_address_mapX()` function and sets the parent of the submaps captured in the `m_reg_maps` array to the component that passed via the `set_parent()` method.

Hierarchical Simulation

As detailed in the introduction, the approach taken to assemble complex SoCs, is to create subsystems first. These subsystems contain cores as well as memories. Other subsystems contain only memory, others only cores. For subsystems with only a core, a memory is required in the testbench. For subsystems with only a memory, a core in the testbench is required. When the subsystems get assembled a memory of a memory only subsystem, will become visible to a subsystem with a core. The same memory may become visible to a core in another subsystem at another address. A memory that was placed inside the testbench of a subsystem will be replaced by a real memory on the SoC. All these cases need to be taken care of by the infrastructure and the implementation of them will be outlined in the following sections.

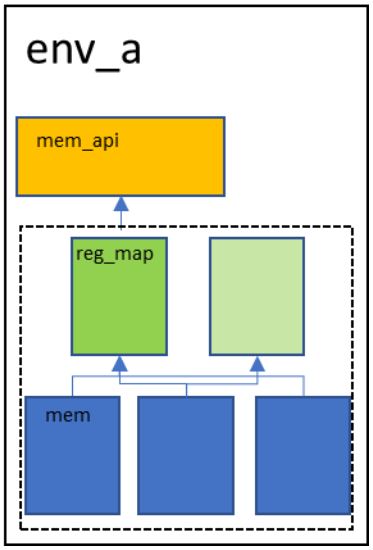

Figure 8. Core with memory testbench setup

A testbench for a subsystem with a core and a memory like shown in Figure 2 is shown in Figure 8. To simplify the pictures, shortened names for the different classes are used. The figure shows the UVM recommended structure to place all the infrastructure inside an extension of the `uvm_env` class. Inside this environment, the `mem_api_base` (yellow box) is instantiated as the viewpoint of the core into the memory map. The actual memory map is passed to the class in form of the reg_block (dotted line box) which got the memory information passed in form of the reg_map (green boxes). The reg_map has then the references to all UVM memories (blue boxes). The connection to the actual memory driver from the UVM memories was described in the Architecture section. Figure 8 also shows that there can be multiple reg_maps to support with multiple viewpoints into the memory map, e.g. to support multiple cores in a subsystem.

The second case is the subsystem that does not have a core. The testbench setup for this scenario is shown in Figure 9. In contrast to the system with the core, the NXP memory driver API class is missing. Only one or more

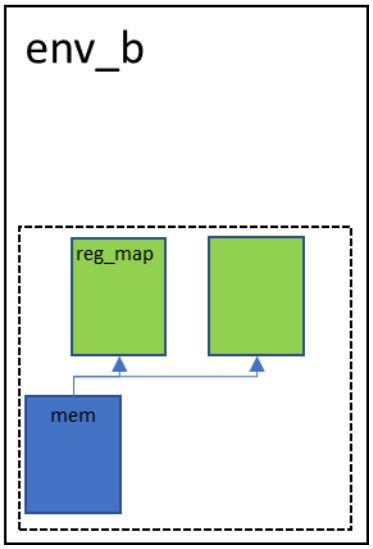

Figure 9. Memory only testbench setup

reg_maps and the corresponding UVM memories are captured in the environment.

At the SoC, these subsystems can then be combined in the SoC environment by instantiating the corresponding subsystem environment in the SoC environment and adding additional memory map definitions into the SoC environment.

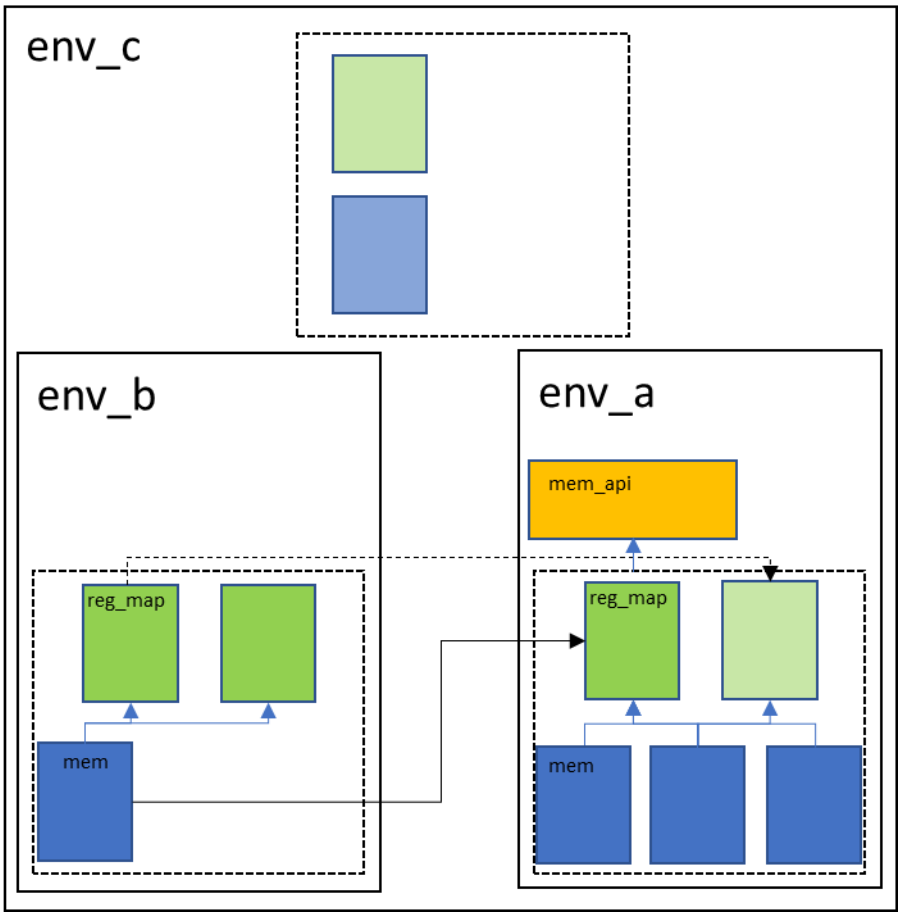

Figure 10. SoC memory assembly

Figure 10 depicts the assembly of an SoC out of the `env_a` and `enb_b` that have been described earlier. The memories of the `env_b` can be mad visible in `env_a` for the core in this subsystem. The figure shows two approaches to register the memories of `env_b` in the address map of the core of `env_a`.

The first option is expressed with the solid arrow in the figure. In this case, the memory of `env_b` is registered in the map file of `env_a`. The code that would reside inside the `env_c` looks like

```
function void env_c::connect_phase (uvm_phase phase);
    …
    m_env_a.m_reg_block.m_mem_map.add_mem(
        m_env_b.m_reg_block.m_mem, 32'h20400000);
```

When this option is used, the memory will be placed at `32'h20400000` in the map of env_a.

The second option is expressed with the dotted arrow in the figure. In this case, the map of `env_b` is registered in the map file of `env_a`. The code that would reside inside the `env_c` looks like

```
function void env_c::connect_phase (uvm_phase phase);
    …
    m_env_a.m_reg_block.m_mem_map.set_submap_offset (
        m_env_b.m_reg_block.m_map, 32'h0);
```

Using the map of `env_b` to get the memory added to the memory map of `env_a`, the memory will be placed into the memory map as defined in the map of `env_b` with the offset `32'h0`.

Emulation

To increase efficiency and reduce development time, it is important to reuse memory loading infrastructure from simulation testbench to load memories on emulation too. Emulation memory models may be architected differently than simulation model and may implement different bit reordering scheme and memory repair bits. It is therefore important to make NXP memory driver API memory drivers parameterized for all these features to allow easy portability to emulation.

It is also more efficient to load memory content as a single hex file on each memory cut one time rather than dynamically accessing and updating memory arrays in each memory cut multiple times as done in simulation.

The NXP memory driver API memory drivers implement options to model design memories for memory cut internally. All dynamic accesses to these memory cuts are made to these internal memories when executing for emulation. Contents of each memory cut are then written out to a memory cut specific hex file at start of UVM's 'main_phase'. These hex files can then be used for loading memory cuts on emulation mode.

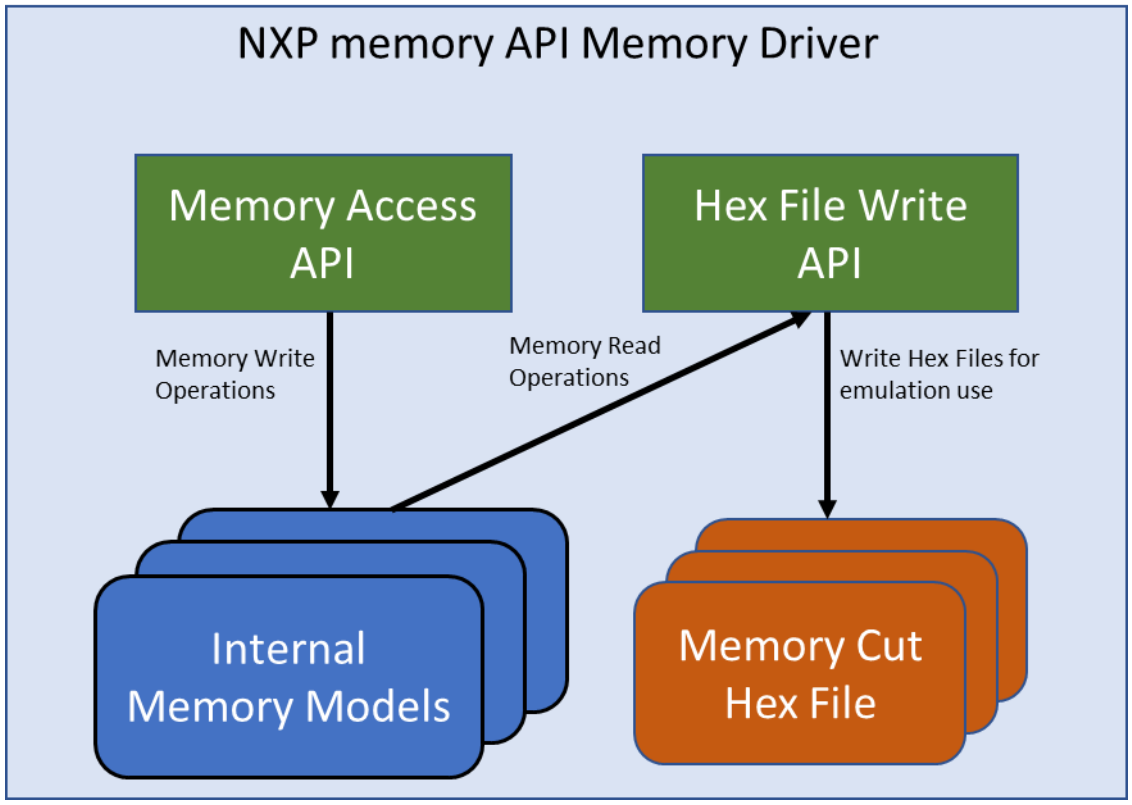

Figure 11. NXP memory driver API for emulation

To increase efficiency and reduce development time, it is important to reuse memory loading infrastructure from simulation

CONCLUSION

Due to the limitations of the UVM register model several new classes had to be developed. As there are several vital class variables that are declared as local, the variable and the function had to be duplicated in the extended classes. Some of the functions that need to be called from the user code have been overloaded to cover the new functionality, however, it may have been better to rename them to make the overloaded behavior visible to the user.

Still, it was possible to cover the required use cases and the memory setup of subsystems can be reused at the next level with little effort. This makes the assembly of SoC testbench much faster with the additional benefit that the components that are reused have been tested at the subsystem.

REFERENCES

[1] https://www.nxp.com/assets/block-diagram/en/i.MX93.pdf

[2] Uwe Simm, “Common UVM Register Model Issues and Pitfalls”, DVClub online Feb 2019

ACKNOWLEDGMENT