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)
Agenda

• Use Cases
• UVM Memories
  • Basics
  • Limitations
• NXP memory driver API
  • Architecture
  • Hierarchical Simulation
  • Emulation
• Conclusion
**Use Case**

- **SoC are build out of subsystems**
  - Every core can have its own map
- **SoC contains many cores**
- **SoC contains multiple memories**
  - Memories are shared

---

<table>
<thead>
<tr>
<th>Subsystems with Core</th>
<th>Subsystems with Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems Clock</td>
<td>PLLs</td>
</tr>
<tr>
<td>Oscillator</td>
<td></td>
</tr>
<tr>
<td>Watches</td>
<td>Timers</td>
</tr>
<tr>
<td>Cortex-M33</td>
<td>Cortex-A55</td>
</tr>
<tr>
<td>32 KB + 16 KB + 272 KB</td>
<td></td>
</tr>
<tr>
<td>NEON</td>
<td>MPB</td>
</tr>
<tr>
<td>94 KB L2 Cache</td>
<td></td>
</tr>
<tr>
<td>FPU</td>
<td></td>
</tr>
<tr>
<td>256 KB L3 Cache (ECC)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>System Control</td>
<td>Low Power Security MCU</td>
</tr>
<tr>
<td>LLC</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>Arm Cortex-M33</td>
</tr>
<tr>
<td>16 KB + 16 KB Code + Cache</td>
<td></td>
</tr>
<tr>
<td>FPU</td>
<td>MPU</td>
</tr>
<tr>
<td>256 KB TCM/OCRAM w/ECC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Connectivity and I/O</td>
<td>EdgeLock Secure Enclave</td>
</tr>
<tr>
<td>UART/SPI x2</td>
<td>Secure Clock</td>
</tr>
<tr>
<td>IIC x2, I2C</td>
<td>Secure I2C</td>
</tr>
<tr>
<td>CAN-FD</td>
<td>eFuse Key Storage</td>
</tr>
<tr>
<td>100 khz Tx/Rx</td>
<td>Random Number</td>
</tr>
<tr>
<td>6-ch PDM Mic Input</td>
<td></td>
</tr>
<tr>
<td>MGS</td>
<td></td>
</tr>
</tbody>
</table>

---

**SoC Description**

- **SoC are build out of subsystems**
- **SoC contains many cores**
  - Every core can have its own map
- **SoC contains multiple memories**
  - Memories are shared
UVM Memories

• 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.

• The closest class in the UVM framework to the `uvm_mem` is the `uvm_reg` class.
UVM Memories: Basics

• Like the `uvm_reg` the `uvm_mem` support front door and back door access

• The `uvm_reg_block` base class can be used to represent a design hierarchy which contains an address map `uvm_reg_map` and a memory modelized with the memory abstraction base class `uvm_mem` accessible via the address map.

• 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.
UVM Memories: Basics (cont.)

• In the UVM top_env multiple instances of class `uvm_reg_block` can be used to create a model representing the design hierarchy
  • For example, a `m_root_reg_block` can contain 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` in the `uvm_reg_map`.
  • Through `add_mem` method, memory can be added to multiple address maps if it is accessible by different interfaces, however the address maps to which it is added need to have the same parent block of the memory.

• The register blocks need to be locked to allow the computation of the final memory map.
UVM Memories: Limitations

• One `uvm_reg_block` supported for a `uvm_reg_map`

```verilog
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
```

• A `uvm_reg_map` can only have one parent map

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

• A `uvm_mem` can only be in one `uvm_reg_block`

```verilog
```

```verilog
`uvm_warning("RegModel",
"Memory ", get_full_name(), "," is not contained within map ",map.get_full_name(), ","
(caller == "", caller, ")")
```
NXP memory driver API: Motivation

- Almost all NXP devices contain an embedded core which needs memory from which SW will be executed.
  - Every testbench need to load memory with content to verify the device.
- Safety, geometry and size requirements cause that the memory implementation becomes not straight forward
  - Loading memory cuts and calculate ECC need to be handled by infrastructure
- Due to SoC complexity, SoC assembly is done hierarchically
  - Hierarchical structure needs to be supported.
- Devices have multiple cores, with different address maps associated.
- Verification spawns the whole space from system to gate level as well as different engines
  - Infrastructure need to support the usage of the same code along the 9 yards.
NXP memory driver API: Architecture

- Architecture is split into frontend and backend.
- Frontend API available for:
  - UVM sequences
  - Verilog tasks
  - C stimulus
- Connection to backend uses TLM2:
  - To support SC and emulation use cases
NXP memory driver API: Frontend View

• Address information is taken from uvm_reg_block
• uvm_mem is used as mirror
• TLM2 socket is used to connect to backend
  • create_backdoor eases the setup

```verilog
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
```
NXP memory driver API: Backend View

• Two types of memory driver's base classes are provided
  • `mem_api_driver` is used for design internal memories
  • `nxp_mem_gp_driver` is used as a base for testbench memories

• Connection to the frontend is done via TLM2

```c
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);
```

```c
mem_api_driver
  create_uvm_mems (name, parent, mems[])
  get_base_address (inst_num)
  obs_mem_array (access_type, addr, strb, match)
  write_mem_array (addr, strb, data, compute_and_write_ecc = 1)
  read_mem_array (addr, strb, data, compute_and_write_ecc = 1)

m_nb_target_socket

nxp_mem_gp_driver
  put_memory_data (gp)
  get_memory_data (gp)
  duplicate_byte_lanes (gp)
  move_write_data (gp)
  get_mem_access (gp, address_window_size)

mem_api_driver_base
  m_nb_target_socket

m_nb_target_socket

uvm_driver
```
NXP memory driver API: Building Blocks

If subsystem contains a CPU:

- mem_api_base
- <project>_config
- m_mem_api_cpu_name
- stim_manager_base
- uvm_object
- uvm_env
- uvm_sequencer
- uvm_reg_block
- uvm_reg_map
- drv_mem
- nxp_reg_block
- nxp_reg_map
- nxp_tlm2_mem
- uvm_mem
NXP memory driver API: Overcome Limitations

- Multiple submap use case

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

- Map use in multiple `uvm_reg_blocks`

```cpp
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));
```

- To calculate the address map

```cpp
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
```
Hierarchical Simulation: CPU Example

- Environment with CPU provides Memory API
- `uvm_reg_map` provides view into subsystem, multiple `uvm_reg_map` captures different views
- Memory driver are provided with the subsystem environment

begin : M33_SYS_TCM_MEM_MAP
m_env_a_mem_map.add_mem (.mem(m_sys_tcm_mem),
.offset('h20000000));
end : M33_SYS_TCM_MEM_MAP

begin: MEMORY_API_KEEP_BUILD
m_memory_api = new ("m_memory_api");
m_memory_api_sqr = new ("m_memory_api_sqr",
this);
m_memory_api_sqr.put_cpu_memory_api(
m_memory_api,
m_env_a_config.m_mem_api_cpu_name,
this);
end: MEMORY_API_KEEP_BUILD
Hierarchical Simulation: Memory Example

- `uvm_reg_map` provides view into subsystem, multiple `uvm_reg_map` captures different views
- Memory driver are provided with the subsystem environment

```verilog
begin : M_OCRAM_DRV_MEM_MAP
m_env_b_mem_map.add_mem (.mem(m_ocram_drv_mem), .offset('h20480000));
end : M_OCRAM_DRV_MEM_MAP

begin : M_OCRAM_DRV_ELAB
    string hdl_path[$];
    string cut_path[2];
    string base_path;
    m_env_b_reg_block.get_full_hdl_path(hdl_path);
    base_path = {hdl_path[0],".mtr_nocmix");
    cut_path = {'"ocram_mem0.mem","ocram_mem1.mem"};
    for (int ii=0; ii < 2; ii++)
        cut_path[ii] = {base_path,".",cut_path[ii]};
    m_ocram_drv.set_mem_scope(cut_path);
end : M_OCRAM_DRV_ELAB
```
Hierarchical Simulation: Assembly View

Setup supports reuse of

• Same driver in multiple maps
• One map as multiple submaps
• Address info from subsystem to SoC
NXP memory driver API: Emulation

• Infrastructure Reuse from Simulation to Emulation
  ▪ Increases efficiency and reduces development time

• Parameterized memory driver API Memory Drivers
  ▪ Easy portability to emulation memory models
  ▪ Supports different bit reordering and memory repair schemes in emulation models

• Models Internal Memories
  ▪ Internal memory models for each DUT memory cut
  ▪ Dynamic memory write operations to internal memories
  ▪ API to write hex files for each DUT memory cut executed after all memory write operations
  ▪ Dumped hex files loaded on corresponding DUT memory cuts

![Diagram showing NXP memory API Memory Driver, Memory Access API, Hex File Write API, Internal Memory Models, Memory Cut Hex File, and write operations.]
Conclusion

• Presented infrastructure addresses all requirements of the use case
• UVM limitations got addressed via extended UVM classes
  • Overwritten UVM functions should have been renamed to make change visible

• Hierarchical assembly code captures level depended properties only
  • Properties can be tested in a divide and conquer approach
  • Assembly of testbench is accelerated
Questions?
Thank You for Attending this Presentation