

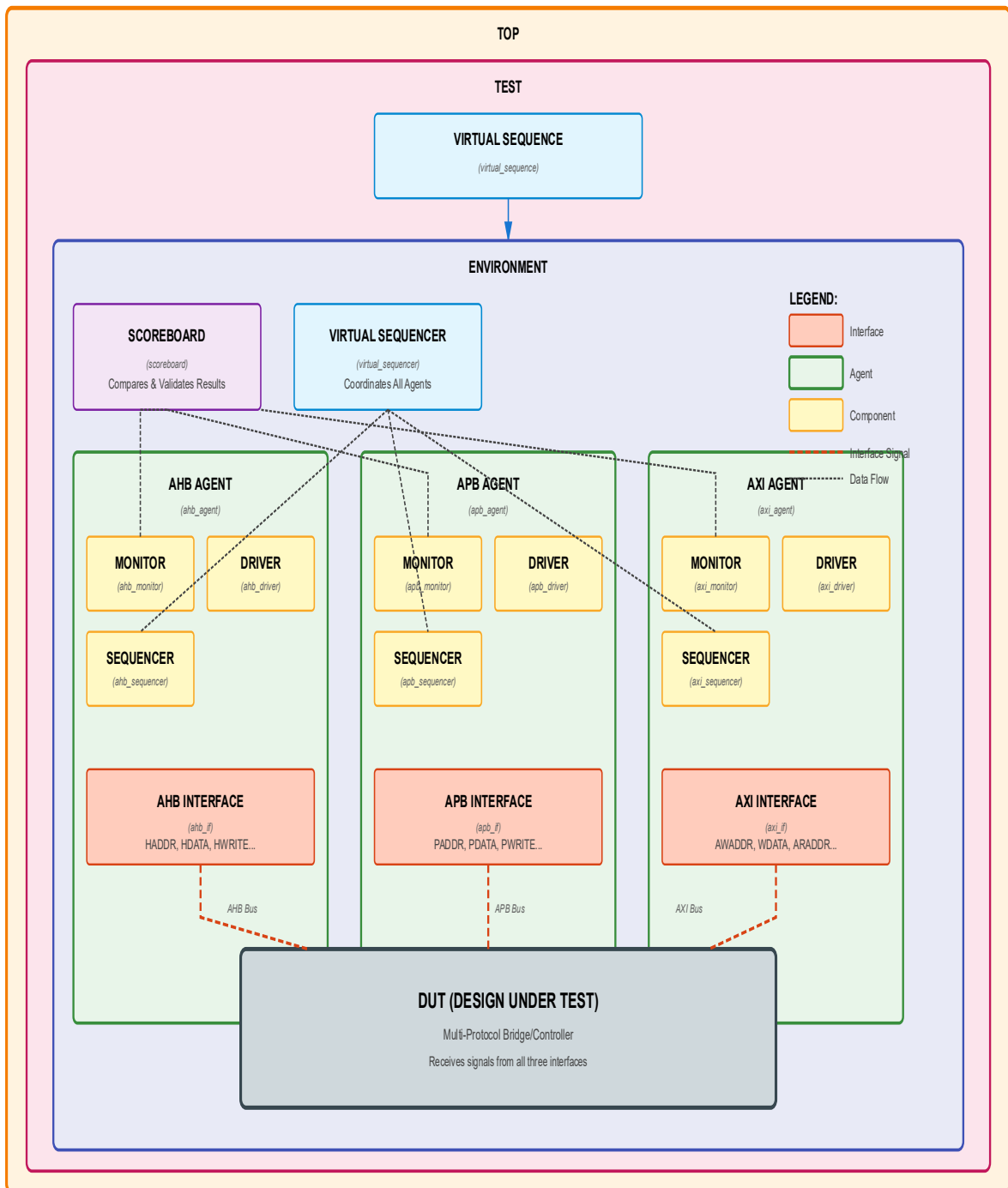
VIRTUAL SEQUENCE AND VIRTUAL SEQUENCER

[madanrajcr](#) | [madan raj c r](#) | madanrajcr@gmail.com

In **UVM-based SoC verification**, a single interface is rarely sufficient. Real SoC designs integrate **multiple protocols** such as **APB, AHB, AXI, PCIe, SPI**, etc. Each interface has its **own agent, sequencer, driver, and monitor**.

The real challenge arises when **stimulus must be coordinated across these interfaces**. This is where **Virtual Sequencers and Virtual Sequences** become essential.

UVM TESTBENCH ARCHITECTURE - BLOCK DIAGRAM



[madanrajcr](#) | [madan raj c r](#) | madanrajcr@gmail.com

VIRTUAL SEQUENCE AND VIRTUAL SEQUENCER

 madanrajcr |  madan raj c r |  madanrajcr@gmail.com

A virtual sequencer is required because SoC-level verification is **no longer limited to driving a single interface independently**; instead, it focuses on coordinating behavior across multiple protocols. In a basic UVM environment, **a sequencer controls stimulus for only one driver**, which is sufficient when interfaces operate in isolation. However, real SoCs involve dependencies such as configuring registers over APB before triggering AXI transactions or running AHB configuration in parallel with AXI traffic. Since a normal sequencer has no visibility beyond its own interface, it cannot manage such cross-protocol ordering or synchronization. Embedding this logic inside a protocol-specific sequencer would violate abstraction and reduce reusability. A virtual sequencer addresses this by acting as a centralized control entity that does not connect to any driver but instead holds handles to multiple real sequencers. Virtual sequences running on it coordinate the start, ordering, and parallel execution of protocol-specific sequences, enabling clean, system-level scenario modeling that closely mirrors real SoC behavior.

Important Technical Points:

- A virtual sequencer is not associated with any physical interface or driver; it exists purely for control and coordination.
- It resides at the environment level and maintains handles to all protocol-specific sequencers.
- It enables cross-interface dependencies, ordering constraints, and parallel execution of sequences.
- Virtual sequences run on the virtual sequencer and launch child sequences on real sequencers.
- It preserves protocol independence by keeping coordination logic out of individual agents.
- It improves test readability, scalability, and reuse for complex SoC verification scenarios.
- It allows verification to shift from interface-level testing to true system-level validation

Example:

```
class apb_seqr extends uvm_sequencer #(apb_seq_item);  
  
    `uvm_component_utils(apb_seqr)  
  
    function new(string name, uvm_component parent);  
        super.new(name, parent);  
    endfunction  
  
endclass
```

VIRTUAL SEQUENCE AND VIRTUAL SEQUENCER

 [madanrajcr](#) |  [madan raj c r](#) |  madanrajcr@gmail.com

```
class ahb_seqr extends uvm_sequencer #(ahb_seq_item);  
  `uvm_component_utils(ahb_seqr)  
  function new(string name, uvm_component parent);  
    super.new(name, parent);  
  endfunction  
endclass
```

```
class axi_seqr extends uvm_sequencer #(axi_seq_item);  
  `uvm_component_utils(axi_seqr)  
  function new(string name, uvm_component parent);  
    super.new(name, parent);  
  endfunction  
endclass
```

Virtual_sequencer

```
class virtual_sequencer extends uvm_sequencer;  
  `uvm_component_utils(virtual_sequencer)  
  
  apb_seqr m_apb_seqr;  
  ahb_seqr m_ahb_seqr;  
  axi_seqr m_axi_seqr;  
  
  function new(string name, uvm_component parent);  
    super.new(name, parent);  
  endfunction  
endclass
```

 [madanrajcr](#) |  [madan raj c r](#) |  madanrajcr@gmail.com

VIRTUAL SEQUENCE AND VIRTUAL SEQUENCER

 madanrajcr |  madan raj c r |  madanrajcr@gmail.com

A virtual sequence is used in UVM to describe **system-level test scenarios** where stimulus **must be coordinated across multiple interfaces** rather than generated for a single driver. In a typical environment, a normal sequence creates sequence items and sends them to one sequencer, which then drives a specific interface. However, at the SoC level, verification scenarios often involve multiple agents that must work together in a specific order or in parallel. A virtual sequence addresses this need by acting as a high-level control flow that orchestrates other sequences instead of directly generating transactions.

Conceptually, a virtual sequence runs on a **virtual sequencer**, which has no driver attached but holds references to multiple real sequencers. The virtual sequence uses these references to start protocol-specific sequences, such as an APB configuration sequence, followed by AXI data traffic, or even multiple sequences running concurrently on different interfaces. Because the virtual sequence does not create or send sequence items itself, it remains independent of any particular protocol and focuses only on the timing, ordering, and synchronization of stimulus.

This separation of responsibilities is critical in complex SoC verification. Individual sequences remain reusable and protocol-specific, while the virtual sequence captures real system behavior by coordinating how and when these sequences execute. As a result, tests become cleaner, easier to maintain, and more representative of actual hardware use cases, such as boot flows, DMA operations, or multi-bus interactions.

Important Technical Points

- A virtual sequence is a high-level sequence that controls other sequences rather than generating sequence items.
- It always runs on a virtual sequencer, not on a protocol-specific sequencer.
- It starts child sequences on real sequencers using their handles.
- It is used to enforce ordering, dependency, and parallelism across multiple interfaces.
- It does not interact with any driver directly and remains protocol-independent.
- Virtual sequences are commonly used in SoC-level verification where multiple agents must be synchronized.
- They help model real-world scenarios such as configuration-before-traffic, parallel bus activity, and complex initialization flows.
- This approach improves reuse, scalability, and clarity of UVM tests by separating system-level control from interface-level stimulus generation.

 madanrajcr |  madan raj c r |  madanrajcr@gmail.com

VIRTUAL SEQUENCE AND VIRTUAL SEQUENCER

 [madanrajcr](#) |  [madan raj c r](#) |  madanrajcr@gmail.com

Example:

```
class apb_seq extends uvm_sequence #(apb_seq_item);
```

```
  `uvm_object_utils(apb_seq)
```

```
  function new(string name = "apb_seq");
```

```
    super.new(name);
```

```
  endfunction
```

```
  task body();
```

```
    apb_seq_item item;
```

```
    item = apb_seq_item::type_id::create("item");
```

```
    `uvm_do(item)
```

```
  endtask
```

```
endclass
```

```
class ahb_seq extends uvm_sequence #(ahb_seq_item);
```

```
  `uvm_object_utils(ahb_seq)
```

```
  function new(string name = "ahb_seq");
```

```
    super.new(name);
```

```
  endfunction
```

```
  task body();
```

```
    ahb_seq_item item;
```

```
    item = ahb_seq_item::type_id::create("item");
```

```
    `uvm_do(item)
```

```
  endtask
```

```
endclass
```

 [madanrajcr](#) |  [madan raj c r](#) |  madanrajcr@gmail.com

VIRTUAL SEQUENCE AND VIRTUAL SEQUENCER

 [madanrajcr](#) |  [madan raj c r](#) |  madanrajcr@gmail.com

```
class axi_seq extends uvm_sequence #(axi_seq_item);  
  `uvm_object_utils(axi_seq)  
  function new(string name = "axi_seq");  
    super.new(name);  
  endfunction  
  
  task body();  
    axi_seq_item item;  
    item = axi_seq_item::type_id::create("item");  
    `uvm_do(item)  
  endtask  
endclass
```

Virtual sequence:

```
class virtual_sequence extends uvm_sequence;  
  `uvm_object_utils(virtual_sequence)  
  `uvm_declare_p_sequencer(virtual_sequencer)  
  
  function new(string name = "virtual_sequence");  
    super.new(name);  
  endfunction  
  
  task body();  
    apb_seq apb_s;  
    ahb_seq ahb_s;  
    axi_seq axi_s;  
  
    apb_s = apb_seq::type_id::create("apb_s");  
    ahb_s = ahb_seq::type_id::create("ahb_s");
```

VIRTUAL SEQUENCE AND VIRTUAL SEQUENCER

 madanrajcr |  madan raj c r |  madanrajcr@gmail.com

```
axi_s = axi_seq::type_id::create("axi_s");

apb_s.start(p_sequencer.m_apb_seqr);
ahb_s.start(p_sequencer.m_ahb_seqr);
axi_s.start(p_sequencer.m_axi_seqr);

endtask

endclass
```

m_sequencer and **p_sequencer**

In UVM, the difference between **m_sequencer** and **p_sequencer** becomes especially critical when you start writing **virtual sequences**, because this is where system-level coordination requires clean and type-safe access to multiple sequencers. Every sequence in UVM automatically gets a handle called **m_sequencer**, which points to the sequencer on which the sequence is currently running. However, this handle is intentionally generic and untyped, because UVM allows any sequence to run on any compatible sequencer at runtime. As a result, **m_sequencer** is of type `uvm_sequencer_base`, which means it does not expose any protocol-specific or user-defined members directly.

When you want to access custom fields or sub-sequencer handles inside a sequencer—such as APB, AXI, or AHB sequencer handles stored inside a virtual sequencer—you cannot do so directly through **m_sequencer**. You must first cast it to the correct sequencer type, and this casting must be done carefully to avoid runtime errors. This makes code verbose and error-prone, especially in complex SoC environments where virtual sequences frequently interact with many sequencers.

To solve this, UVM provides **p_sequencer**, which is a strongly typed handle created using the macro `uvm_declare_p_sequencer`. When this macro is used inside a sequence, UVM automatically defines **p_sequencer** with the exact type of the sequencer the sequence is intended to run on, such as a `virtual_sequencer`. This allows the sequence to directly and safely access all user-defined members of the sequencer without casting. In virtual sequences, this is extremely important because the sequence must frequently access handles to multiple real sequencers stored inside the virtual sequencer.

In practice, virtual sequences almost always rely on **p_sequencer**, because they need clean, readable, and type-safe access to sub-sequencers like `apb_seqr`, `axi_seqr`, or `ahb_seqr`. While **m_sequencer** is always present and useful for generic or reusable sequences, **p_sequencer** is the preferred mechanism for virtual sequences where correctness, clarity, and maintainability are critical.


 madanrajcr |  madan raj c r |  madanrajcr@gmail.com

VIRTUAL SEQUENCE AND VIRTUAL SEQUENCER

 madanrajcr |  madan raj c r |  madanrajcr@gmail.com

Important Technical Points:

- **m_sequencer** is automatically available in every sequence and is of type `uvm_sequencer_base`.
- It is generic and untyped, so it cannot directly access user-defined sequencer members.
- Accessing specific sequencer fields through **m_sequencer** requires explicit casting.
- Incorrect casting can lead to runtime errors and harder-to-maintain code.
- **p_sequencer** is a strongly typed handle created using `uvm_declare_p_sequencer(...)`.
- It provides direct, type-safe access to sequencer-specific members.
- **p_sequencer** is most commonly used in virtual sequences running on virtual sequencers.
- Virtual sequences rely on **p_sequencer** to access multiple real sequencers cleanly.
- Using **p_sequencer** improves readability, safety, and robustness of SoC-level UVM tests.

-
-  **Follow me on LinkedIn for high-quality technical content!**
If you're into **Verilog, SystemVerilog, UVM, and hardware design/verification**, I share useful tutorials, tips, and insights that can help you grow your skills and career.

-
-  **Connect with me:**
 <https://www.linkedin.com/in/madan-raj-c-r-b6029721b/>

Don't forget to **follow** for updates and technical posts — see you there! 🙌

 madanrajcr |  madan raj c r |  madanrajcr@gmail.com