Module: UVM-1
Verification Microarchitecture

UVM Verification Microarchitecture
IBEX Core

➢ **Overview:**

The verification environment incorporates three key interfaces, each grouping related signals into a cohesive unit for modularity and simplicity:

1. **dut_probe_intf:** Monitors the core status and controls fetch enable signals.
2. **instr_mem_intf:** Handles signals associated with the fetch unit.
3. **data_mem_intf:** Manages signals for the Load Store Unit (LSU).

Additionally, the testbench includes a unified memory model that serves both instruction and data requests. Program binaries are loaded into this memory, enabling the core to fetch instructions via the fetch unit and process load/store requests through the LSU.

➢ **Environment Components:**

The verification components are organized within the **`verif/`** directory of the codebase. The testbench architecture is built around three primary UVCs that collaboratively ensure seamless operation:

1. **INSTR_UVC:** Manages instruction-side signals and handles instruction requests via key components like **`instr_driver`**, **`instr_monitor`**, **`instr_agent`**, **`instr_sequence`**, and a **`sequencer`**.
2. **DATA_UVC:** Monitors and drives LSU-side signals, managing data requests. Components include **`data_driver`**, **`data_monitor`**, **`data_agent`**, **`data_sequence`**, and a **`sequencer`**.
3. **ENV_UVC:** Acts as the top-level abstraction, integrating instruction and data agents. It features a **`virtual_sequencer`**, **`sequences`**, and a **`base_test`** for environment configuration.

At the highest level, the **`tb_top`** module configures and connects these components, instantiating the DUT for simulation.

➢ **Folder Structure Diagram:**

The verification environment has the following folder structure:

```
. DIRECTORY STRUCTURE
|-- doc
|   `-- images
|-- rtl
|-- simv.vdb
|   `-- snps
|       `-- coverage
|           `-- db
|               |-- auxiliary
|               |-- design
|               |-- shape
|               `-- testdata
|                   `-- test
|-- tests
|   |-- add
|   |-- addi
|   |-- and
|   |-- andi
|   |-- asm_tests
|   |-- beq
|   |-- bge
|   |-- blt
|   |-- bltu
|   |-- bne
|   |-- jal
|   |-- lbu
|   |-- lh
|   |-- lhu
|   |-- lui
|   |-- lw
|   |-- or
|   |-- ori
|   |-- sb
|   |-- sh
|   |-- sll
|   |-- slli
|   |-- slt
|   |-- slti
|   |-- sltiu
|   |-- sltu
|   |-- sra
|   |-- srai
|   |-- srl
|   |-- srli
|   |-- sub
|   |-- sw
```

```
|   |-- xor
|   `-- xori
`-- verif
    `-- UVCs
        |-- data_UVC
        |-- env_UVC
        `-- instr_UVC

54 directories
```

➢ Test Workflow

High-Level Overview:

The test workflow begins in the **`tb_top`** module by invoking **`run_test("base_test")`**. This triggers the execution of the **`base_test`** class, where key setup and configuration tasks are performed:

1. Interfaces (e.g., instruction and data) are configured using **`uvm_config_db`**, enabling components like monitors and drivers to access and interact with these interfaces.
2. The **`base_test`** handles DUT configuration, loading the program binary into the memory model, and setting up termination conditions (e.g., detecting an **`ecall`** instruction).

Base Test:

1. **Run Phase:**
    - **DUT Setup**: A simulation objection is raised, and the **`fetch_enable`** signal is set to activate instruction fetch requests.
    - **Sequence Execution**:
        - The loaded memory model is passed to the virtual sequence, which is started on the **`virtual_sequencer`**.
    - **Monitoring and Termination**:
        - A fork-join construct runs two threads:
            - One executes the test sequence.
            - Another monitors the DUT for an **`ecall`** (**`instr_rdata_i = 32'h00000073`**).

- Upon detecting an **`ecall`**, the test terminates, and the objection is dropped.

Micro-Level Details:

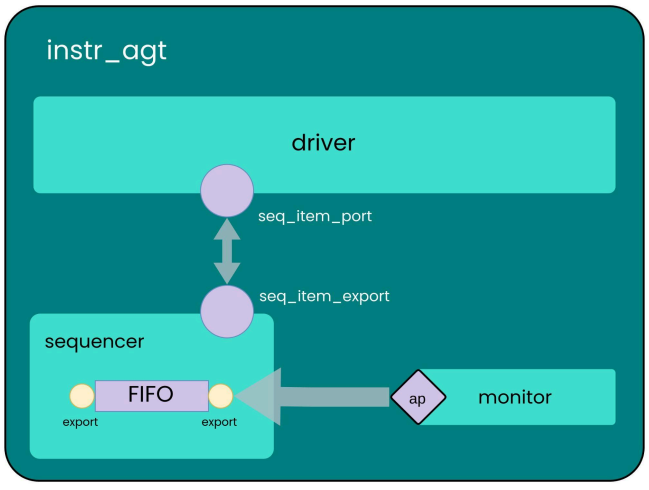
Instruction Agent:

- **Monitor**: Tracks instruction requests from the DUT and sends these transactions to the analysis port.
- **Sequence**: Captures transactions in a sequencer FIFO and generates corresponding responses for the driver to execute.

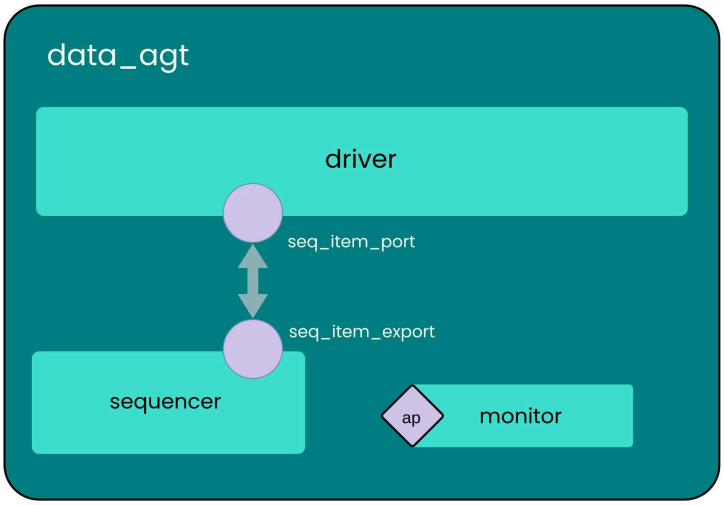
Data Agent:

- **Slave Sequence**:
  - Begins with an empty request item sent to the driver.
  - The driver populates this request with pin-level signal values from the LSU interface and returns it to the sequence.
- **Memory Operations**:
  - For data requests (read or write), the memory model handles the request and generates response items.
  - These response items are sent back to the driver, which drives them onto the interface.

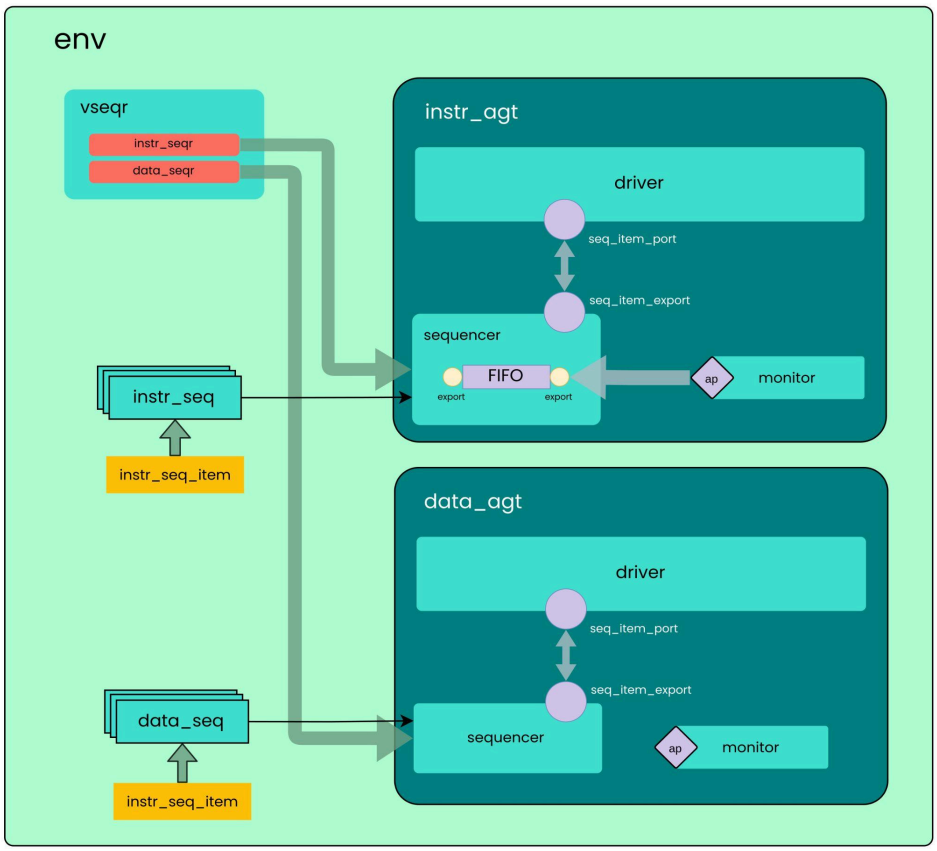
A simplified block-level diagram is provided below to enhance understanding.



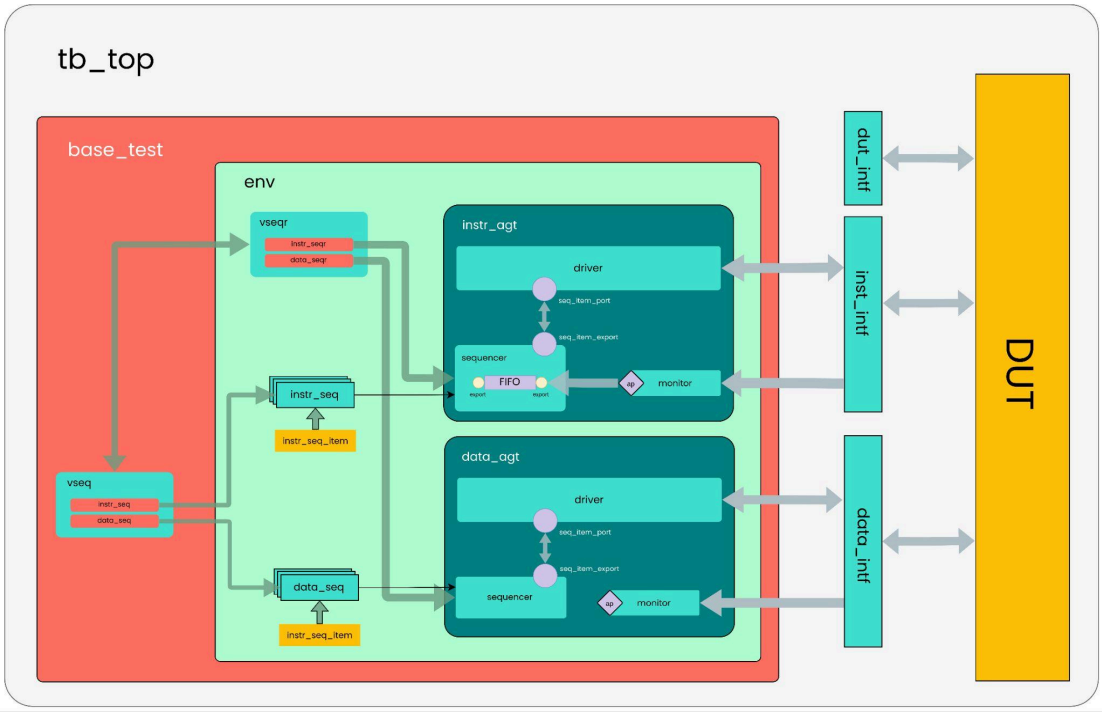
instr_agent



data_agent



Environment


tb_top
base_test
env
vseqr
instr_seqr
data_seqr
instr_agt
driver
seq_item_port
seq_item_export
sequencer
FIFO
export
export
ap
monitor
instr_seq
instr_seq_item
data_agt
driver
seq_item_port
seq_item_export
sequencer
ap
monitor
data_seq
instr_seq_item
vseq
instr_seq
data_seq
dut_intf
inst_intf
data_intf
DUT

tb_top