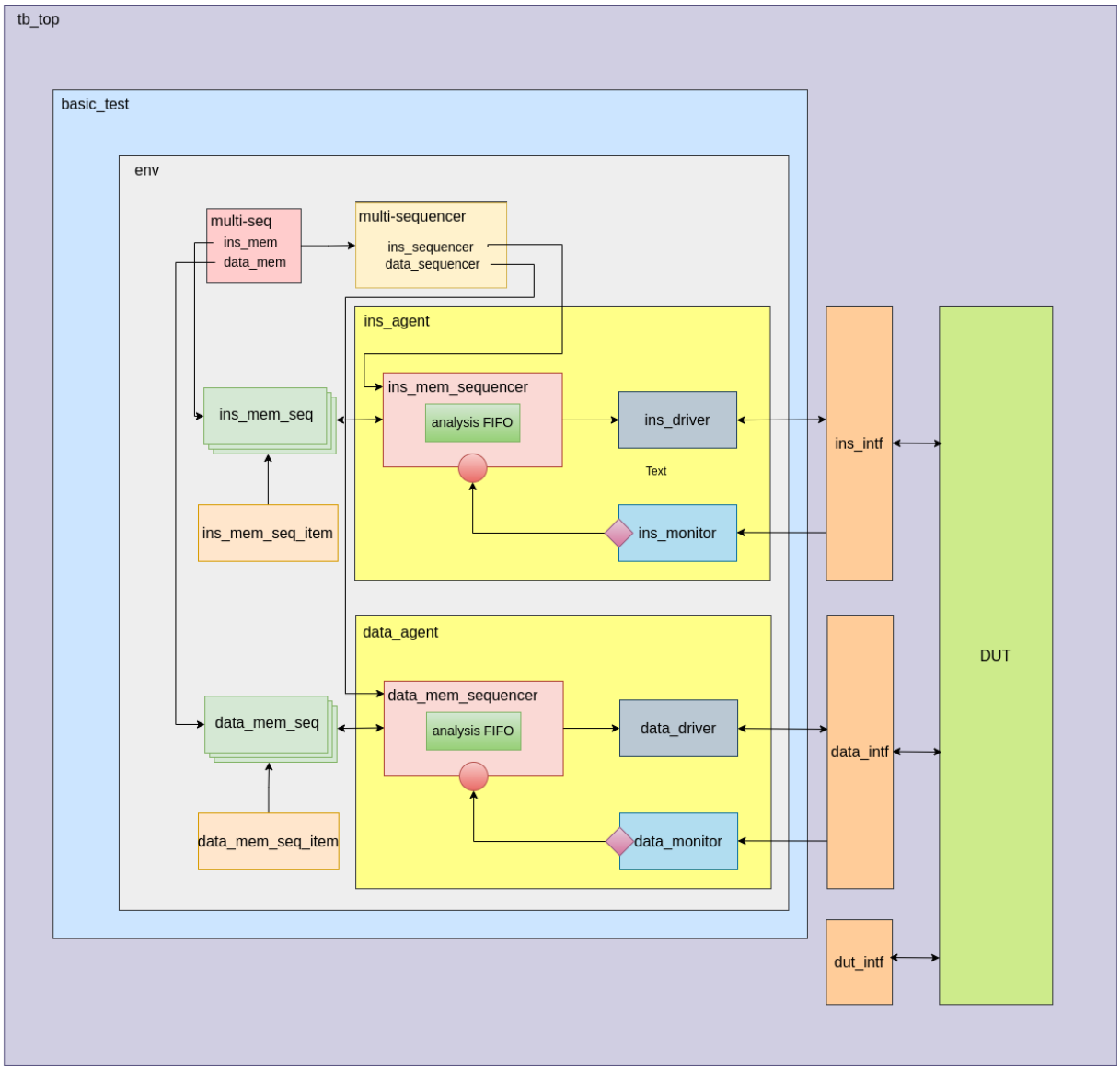
UVM Capstone - RISC-V Ibex Core Verification

Project Overview:

Ibex is an open-source 32-bit RISC-V CPU core developed in SystemVerilog. This CPU core is highly customizable and ideal for use in embedded control systems. The goal of this project is to create a UVM based verification environment of the Ibex core. Instructions are loaded into memory through a binary file and the core simulation is started. The test ends when the ECALL instruction is detected. A trace log is generated which indicates the executed instructions. Also, a result log is generated at the end from which the execution of the verification environment can be tracked.

Verification Testbench Architecture:



The detailed explanation is as follows:

Memory model:
The testbench creates a memory model instance, which is responsible for loading the compiled assembly test program into memory at the start of each test. This memory model functions as a single memory unit for both instructions and data, handling all memory requests from the memory interface agents.

Interfaces:

- **dut_intf:** fetch signal is sent to DUT on the basis of which core execution starts.
- **ins_intf:** monitors the instruction memory requests and related signals.
- **data_intf:** monitors the instruction memory requests and related signals.

Instruction Agent (ins_agent):
This agent controls the IF signals and requests.

- **ins_monitor:** gets the instruction request and address signals from the ins_intf and writes them to the analysis port which is connected to the TLM analysis FIFO export of ins_sequencer.
- **ins_sequencer:** TLM analysis FIFO export is created which is connected to the analysis port of ins_monitor. This ensures that all the instruction requests are executed in order.
- **ins_driver:** gets the next item from the ins_sequencer and drives the instruction signals onto the ins_intf accordingly. It checks if an instruction request is active, and if so, it grants permission, sets the control signals and transfers instruction data, followed by clock advancements. After processing, it signals completion to the sequencer.

These 3 classes are instantiated into the agent and FIFO export and analysis ports are connected.

Data Agent (data_agent):
This agent controls the LSU signals and requests.

- **data_monitor:** gets the data request and address signals from the data_intf and writes them to the analysis port which is connected to the TLM analysis FIFO export of data_sequencer.
- **data_sequencer:** TLM analysis FIFO export is created which is connected to the analysis port of data_monitor. This ensures that all the data requests are executed in order.
- **data_driver:** gets the next item from the data_sequencer and drives the data signals onto the data_intf accordingly. It checks if data request is active, and if

so, it grants permission, sets the control signals and transfers data, followed by clock advancements. After processing, it signals completion to the sequencer.

These 3 classes are instantiated into the agent and FIFO export and analysis ports are connected.

Multi Sequencer (multi_sequencer):
This sequencer contains the handles of both ins_sequencer and data_sequencer.

Instruction Memory Sequence (ins_mem_seq):
Memory model and ins_mem_seq_item are instantiated. A virtual p_sequencer of ins_mem_sequencer is declared which is used to get the ins_item from the FIFO export of ins_mem_sequencer. After getting request and address signals, it reads the value at the appropriate address and stores it in the designated signal. The control signals can also be set in sequence, but it is more appropriate to set them in the respective driver.

Data Memory Sequence (data_mem_seq):
Memory model and data_mem_seq_item are instantiated. A virtual p_sequencer of data_mem_sequencer is declared which is used to get the data_item from the FIFO export of data_mem_sequencer. Then the sequence is created accordingly. After getting request and address signals, it reads the value at the appropriate address and stores it in the designated signal. The control signals can also be set in sequence, but it is more appropriate to set them in the respective driver.

Multi Sequence (multi_seq):
This sequence contains the handles of both ins_mem_sequence and data_mem_sequence. Memory model is instantiated and is connected to the memory model instances created in ins_mem_sequence and data_mem_sequence. A virtual p_sequencer of multi_sequencer is declared. Then by using p_sequencer, both ins_mem_sequencer and data_mem_sequencer are started in parallel using fork-join.

Environment (env):
ins_agent, data_agent and multi_sequencer are instantiated in the env. ins_sequencer and data_sequencer handles in the multi_sequencer are connected to the respective agents' sequencers.

Base Test (base_test):
Memory model, env and multi_seq are instantiated. Also virtual handles of clk_rst_if, dut_intf, ins_intf and data_intf are created. These are get from the top module using config_db. Topology report is also printed to ensure the correctness of the hierarchy.

In the run phase, after raising the objection, test instructions are loaded from the binary file into the memory model. After that, reset is applied and the memory model is connected to the one instantiated in multi_seq. After that fetch enable signal in the dut_intf is set to start the core execution. This is the **starting condition** for our test.

After that the multi_sequencer is started in parallel with the **ending condition** for our test which is the detection of ECALL instruction. These processes are run in parallel using fork-join_any. After that the objection is dropped and test stops.

Top Module:
All the relevant files are included. The paths for the included files are not static and can be changed. Static interfaces clk_rst_if, dut_intf, ins_intf, data_intf are created and set using config_db. Also the clock is set_active() and the design module **ibex_top_tracing()** is instantiated and relevant signals are connected. Finally, base_test is set to run. Waveform can also be dumped for future reference.