Initial Design Log

Tuesday, June 18, 2024 6:58 PM

Goals for the semester *(required for initial design log)*

1. Learn UVM and its application in testbench development.
2. Develop and implement a UART testbench.
4. Improve debugging and problem-solving skills.
5. Create necessary documentation for the design and verification process.

Timeline *(required for initial design log)*

Week 1:
- Introduction to UVM and initial setup of the development environment.
- Start preliminary research on UART.

Week 2:
- Continue studying UVM and its testbench architecture.
- Gather resources and documentation for UART.

Week 3:
- Deepen understanding of UART protocol and components.
- Set up the ASIC environment and create the test diagram.
- Plan and discuss necessary agents for the testbench.

Week 4:
- Develop the testplan for the UART testbench.
- Analyze the register map and spec sheet.

Week 5:
- Finalize the testplan and begin coding the UART testbench.
- Implement the basic structure and components.

Week 6:
- Continue building the testbench and add necessary modules.
- Set up and integrate simulation tools (QuestaSim and ThinLinc).

Week1

Tuesday, June 18, 2024 7:01 PM

Week 1:
Estimated Time Spent: 3.5 Hrs

1) Read and took notes on Chapter 1 of SV for Verification book.
    - Easiest bugs to detect: block level. Next best: at boundaries of blocks.
    - To simulate a single design block, you need to create tests that generate stimuli from all the surrounding blocks
    - Error injection and handling can be the most challenging part of verification
    - General Steps for a testbench:
        - Generate stimulus
        - Apply stimulus to the DUT
        - Capture the response
        - Check for correctness
        - Measure progress against the overall verification goals
    - Directed Testing: When there's enough time and people. Making directed and specific tests.
    - Better Methodolody:
        - Constrained-random stimulus
        - Functional coverage
        - Layered testbench using transactors
        - Common testbench for all tests
        - Test-specific code kept separate from testbench
    - Takes longer to make initial progress. But great for long term.
    - Steps:
        - Run basic constrained random tests with bunch of different seeds for a long time.
        - Analyze functional coverage report, find the holes: Minimal Code modification-> directed test cases for the hole.
    - When verifying DUT: should randomize entire env config: length of sim., no. of devices, how they are configured, etc.
    - Subtle bugs are often revealed when intermittent delays are introduced.
    - Unique Seed: time of day + processor name + process ID.
    - Functional Coverage: need to measure what has been verified in order to check off items in your verification plan.
    - Feedback from functional coverage is imp.
    - Testbench Components: Bus Functional Models (BFM) are testbench components which to the DUT look like real components.
    - Layered Testbench: Don’t try to write a single routine that can randomly generate all types of stimulus, both legal and illegal, plus inject errors with a multilayer protocol.
        - Signal Layer: Contains the DUT and the signals connected to the testbench.
        - Command Layer: Driver (runs single commands: sent to DUT inputs). Monitor(takes signal transitions from DUT output and grps them int commands).
            Assertions(look for indv signals as well as changes across entire command).
        - Functional Layer: Agent (transactor: receives higher-level transactions such as DMA read or write and breaks them into individual
            commands.), Scoreboard(predicts the results of the transaction), Checker (compares the commands from the monitor with those
            in the scoreboard.)
        - Scenario Layer: An operation like download music in MP3 player is a scenario. The scenario layer orchestrates all of the steps involved in an operation with constrained-random values.
        - Test and Functional Coverage Layer: Functional coverage measures the progress of all tests in fulfilling the verification plan requirements.
        - When estimating the effort to test a design, don’t count the number of gates; count the number of designers.
        - If your DUT has several protocol layers, each should get its own layer in the testbench environment.

- Creating a driver:
    - May inject errors or add delays
    - Breaks down the received cmd into indv signals such as bus req. and handshakes.
    - Gen. term: Transactor which at its core is a loop.
- Sim. Env. Phases:
    - Build: Gen. Config (randomize config of DUT and surr. env), Build env (allocate and conn. testbench components based on config).
      Reset the DUT, Configure the DUT (load the DUT cmd registers based on step 1.)
    - Run: Start env(run BFMs and stimulus generators), Run test (layer by layer, use time-out checkers).
    - Wrap-Up: Sweep, Report (after DUT is idle, sweep testbench for lost data).

Future Plan:

1) Continue with onboarding.
2) Write some practice code and learn.

Goals

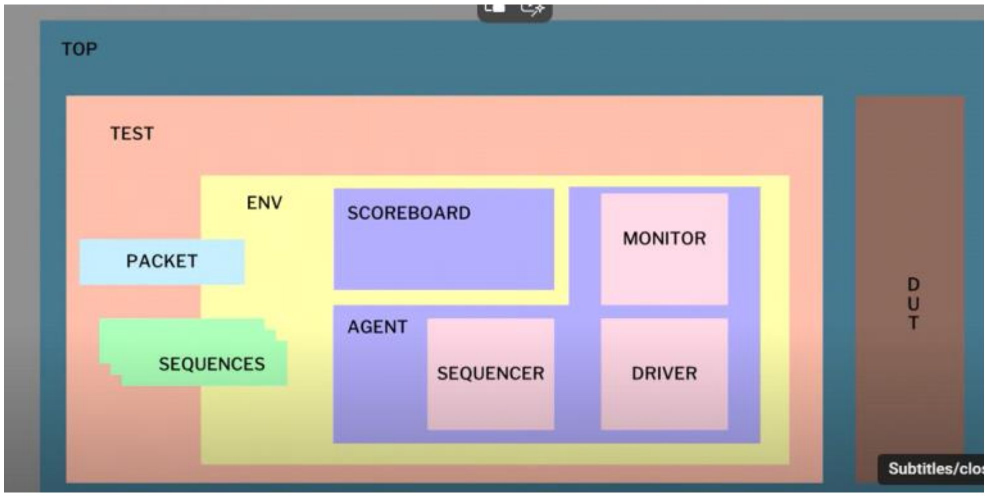
Wednesday, June 19, 2024 5:24 PM

Goals for the semester *(required for initial design log)*

1. Learn UVM and its application in testbench development.
2. Develop and implement a UART testbench.
4. Improve debugging and problem-solving skills.
5. Create necessary documentation for the design and verification process.

Week2

Monday, June 24, 2024 7:04 PM



-Went through Intro to SOCET presentations of all teams to get an overview of the technical terms involved in each.

UVM is a set of rules (what, how and when) to help to verify the design.

Components: Top, Test, Env, Agent, Sequencer, Driver, Monitor (exist throughout the sim.)

Objects (Sequences, Packets/Seq_Item) (Created and Deleted during runtime)

TOP: Contains the Test and the DUT

Test: Env, Packet and Sequence

Env: Agent, Scoreboard

Agent: Seqr, Driver, Monitor

Within UVM Test:

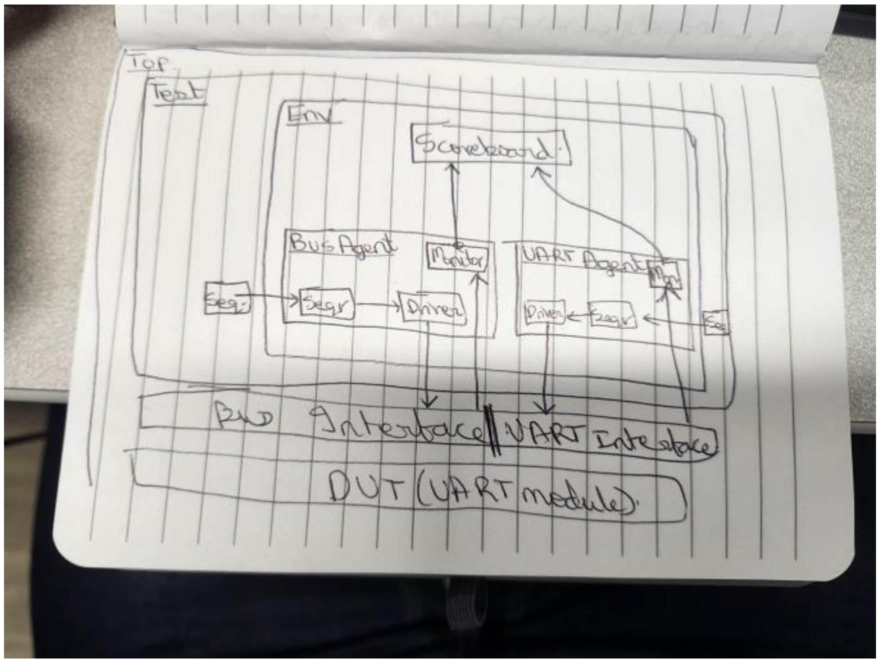
Packet (Sequence_Items)-> Sequences-> Sequencer-> Driver-> DUT (thru interface)-> Monitor checks inp/output-> Scoreboard checks if design passed or not.

-If all the sequences pass: design is error free.

Week3

Thursday, August 1, 2024 9:14 PM

Week 3:
Total Hours: 3.5



- Continued research into UVM testbench coding:
  - Focused on understanding the Universal Verification Methodology (UVM) and how to implement it in a testbench. This involved studying various online resources, articles, and relevant documentation to grasp the fundamental concepts.

- Began research on Vito-UART module:
  - Understanding UART:
    - Investigated the Universal Asynchronous Receiver-Transmitter (UART) protocol, which is critical for serial communication. This included learning about baud rate, parity bits, stop bits, and data bits.
    - Explored how UART functions within a system, including signal transmission, reception, and error handling.
  - Components of the testbench:
    - Studied the key components required for a UART testbench, such as drivers, monitors, and scoreboard. Focused on understanding their roles and how they interact within the testbench.

- Setup of ASIC fabrication environment:
  - Successfully set up the development environment using VSCode and ThinLinc for remote access and collaboration. This setup included configuring necessary plugins and extensions for hardware description languages (HDL) and simulation tools.

- Test diagram creation:
  - Created a detailed test diagram for the UART testbench, outlining the architecture and flow. The diagram included the arrangement of various agents (driver, monitor, scoreboard) and their connections.

- Discussion on agents:

- Participated in discussions regarding the necessary agents for the UART testbench. Gained a deeper understanding of how to implement and connect these agents for effective communication and verification.

Week4

Thursday, August 1, 2024 9:14 PM

Week 4:
Total Hours: 3.5

- Testplan development:
  - Initiated the development of a comprehensive testplan for the UART testbench. This involved defining the scope, objectives, and test cases necessary for thorough verification.

- Receipt and study of register map and specification sheet:
  - Acquired the register map and specification sheet for the UART module. Conducted a thorough analysis to understand the design requirements and constraints, which informed the development of the testplan.

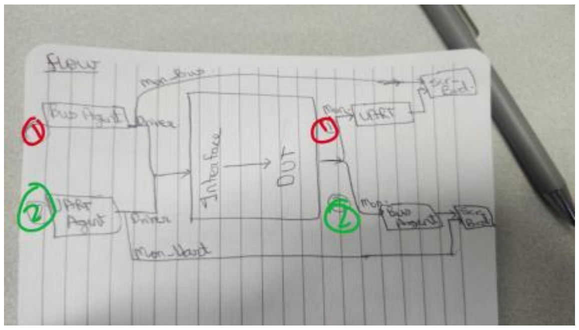
- Research on existing test plans:
  - Reviewed existing test plans for similar UART modules to identify standard practices and common test scenarios. This research helped in formulating a robust set of test cases for the UART testbench.

- Case development for UART testbench:
  - Developed various test cases to cover different scenarios, including normal operation, edge cases, and potential fault conditions. Ensured that all aspects of the UART's functionality were considered.

Week5

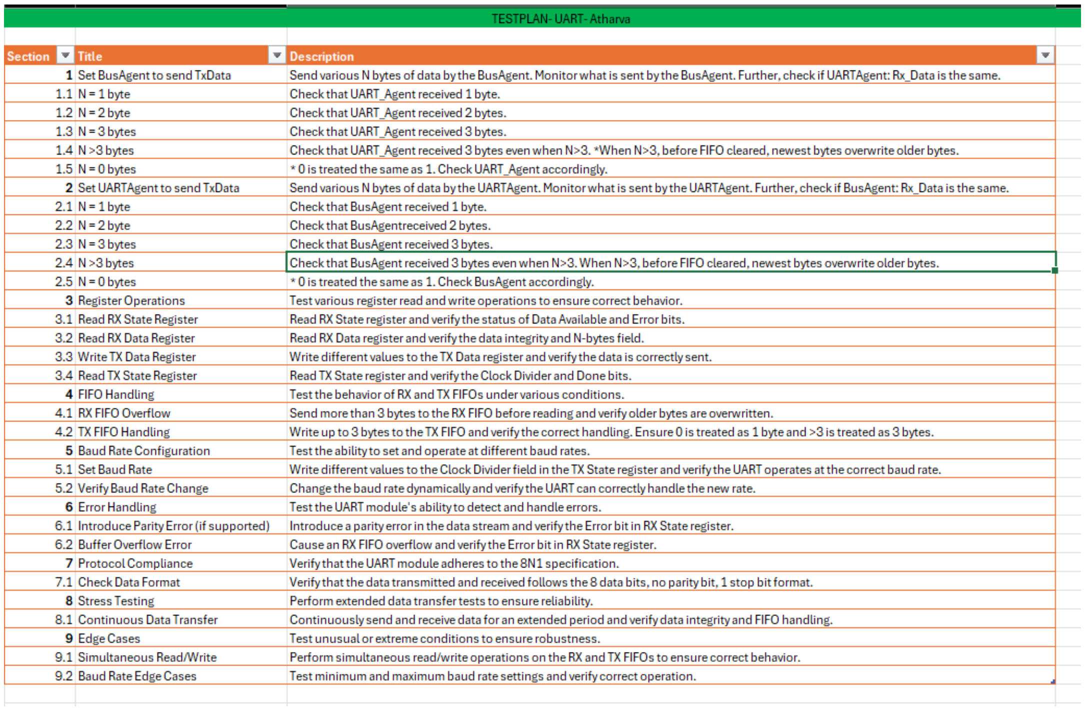
Thursday, August 1, 2024 9:14 PM

Week 5:
Total Hours: 3



- General flow creation for testbench:
  - Designed a general flowchart for the testbench, outlining the sequence of operations and interactions between different components. This visualization helped in understanding the overall structure and flow.

- Completion of testplan and coding initiation:
  - Finalized the testplan, detailing all the necessary test scenarios and expected outcomes. Began coding the testbench for the UART module, implementing the defined test cases.



TESTPLAN- UART- Atharva

Section	Title	Description
1	Set BusAgent to send TxData	Send various N bytes of data by the BusAgent. Monitor what is sent by the BusAgent. Further, check if UARTAgent: Rx_Data is the same.
1.1	N = 1 byte	Check that UART_Agent received 1 byte.
1.2	N = 2 byte	Check that UART_Agent received 2 bytes.
1.3	N = 3 bytes	Check that UART_Agent received 3 bytes.
1.4	N >3 bytes	Check that UART_Agent received 3 bytes even when N>3. *When N>3, before FIFO cleared, newest bytes overwrite older bytes.
1.5	N = 0 bytes	* 0 is treated the same as 1. Check UART_Agent accordingly.
2	Set UARTAgent to send TxData	Send various N bytes of data by the UARTAgent. Monitor what is sent by the UARTAgent. Further, check if BusAgent: Rx_Data is the same.
2.1	N = 1 byte	Check that BusAgent received 1 byte.
2.2	N = 2 byte	Check that BusAgentreceived 2 bytes.
2.3	N = 3 bytes	Check that BusAgent received 3 bytes.
2.4	N >3 bytes	Check that BusAgent received 3 bytes even when N>3. When N>3, before FIFO cleared, newest bytes overwrite older bytes.
2.5	N = 0 bytes	* 0 is treated the same as 1. Check BusAgent accordingly.
3	Register Operations	Test various register read and write operations to ensure correct behavior.
3.1	Read RX State Register	Read RX State register and verify the status of Data Available and Error bits.
3.2	Read RX Data Register	Read RX Data register and verify the data integrity and N-bytes field.
3.3	Write TX Data Register	Write different values to the TX Data register and verify the data is correctly sent.
3.4	Read TX State Register	Read TX State register and verify the Clock Divider and Done bits.
4	FIFO Handling	Test the behavior of RX and TX FIFOs under various conditions.
4.1	RX FIFO Overflow	Send more than 3 bytes to the RX FIFO before reading and verify older bytes are overwritten.
4.2	TX FIFO Handling	Write up to 3 bytes to the TX FIFO and verify the correct handling. Ensure 0 is treated as 1 byte and >3 is treated as 3 bytes.
5	Baud Rate Configuration	Test the ability to set and operate at different baud rates.
5.1	Set Baud Rate	Write different values to the Clock Divider field in the TX State register and verify the UART operates at the correct baud rate.
5.2	Verify Baud Rate Change	Change the baud rate dynamically and verify the UART can correctly handle the new rate.
6	Error Handling	Test the UART module's ability to detect and handle errors.
6.1	Introduce Parity Error (if supported)	Introduce a parity error in the data stream and verify the Error bit in RX State register.
6.2	Buffer Overflow Error	Cause an RX FIFO overflow and verify the Error bit in RX State register.
7	Protocol Compliance	Verify that the UART module adheres to the 8N1 specification.
7.1	Check Data Format	Verify that the data transmitted and received follows the 8 data bits, no parity bit, 1 stop bit format.
8	Stress Testing	Perform extended data transfer tests to ensure reliability.
8.1	Continuous Data Transfer	Continuously send and receive data for an extended period and verify data integrity and FIFO handling.
9	Edge Cases	Test unusual or extreme conditions to ensure robustness.
9.1	Simultaneous Read/Write	Perform simultaneous read/write operations on the RX and TX FIFOs to ensure correct behavior.
9.2	Baud Rate Edge Cases	Test minimum and maximum baud rate settings and verify correct operation.

- Discussion and debugging:
  - Engaged in discussions to identify and resolve bugs encountered during the initial coding phase. Focused on debugging and refining the code to ensure accurate and efficient verification.

Week6

Thursday, August 1, 2024 9:15 PM

Week 6:
Total Hours: 2

- Continued testbench development:
  - Continued adding and refining various components in the testbench, including drivers, monitors, and the scoreboard. Focused on ensuring proper integration and functionality.

- QuestaSim and ThinLinc integration:
  - Successfully connected the QuestaSim simulation tool with ThinLinc for remote simulation and testing. This setup enabled seamless verification and debugging processes.

- Limited work due to emergency:
  - Had an unexpected emergency, which limited the amount of work completed during the week. Despite this, significant progress was made in setting up the simulation environment and adding components to the testbench.