VERIFICATION TEST PLAN

Fundamentals of Pre-Silicon Validation  
Winter -2024

Project Name: Develop Specification, Implementation and Verification of Asynchronous FIFO

Members:   
SAI SUKITHA PULI

DEEPTHI CHEVURU

MASAAKI ISHII

Date:  
03-05-24

Acknowledgement:

Table of Contents

[2 Introduction: 4](#_Toc157438328)

[2.1 Objective of the verification plan 4](#_Toc157438329)

[2.2 Top Level block diagram 4](#_Toc157438330)

[2.3 Specifications for the design 4](#_Toc157438331)

[3 Verification Requirements 4](#_Toc157438332)

[3.1 Verification Levels 4](#_Toc157438333)

[3.1.1 What hierarchy level are you verifying and why? 4](#_Toc157438334)

[3.1.2 How is the controllability and observability at the level you are verifying? 4](#_Toc157438335)

[3.1.3 Are the interfaces and specifications clearly defined at the level you are verifying. List them. 4](#_Toc157438336)

[4 Required Tools 4](#_Toc157438337)

[4.1 List of required software and hardware toolsets needed. 4](#_Toc157438338)

[4.2 Directory structure of your runs, what computer resources you will be using. 4](#_Toc157438339)

[5 Risks and Dependencies 4](#_Toc157438340)

[5.1 List all the critical threats or any known risks. List contingency and mitigation plans. 4](#_Toc157438341)

[6 Functions to be Verified. 4](#_Toc157438342)

[6.1 Functions from specification and implementation 4](#_Toc157438343)

[6.1.1 List of functions that will be verified. Description of each function 4](#_Toc157438344)

[6.1.2 List of functions that will not be verified. Description of each function and why it will not be verified. 4](#_Toc157438345)

[6.1.3 List of critical functions and non-critical functions for tapeout 4](#_Toc157438346)

[7 Tests and Methods 4](#_Toc157438347)

[7.1.1 Testing methods to be used: Black/White/Gray Box. 4](#_Toc157438348)

[7.1.2 State the PROs and CONs for each and why you selected the method for this DUV. 4](#_Toc157438349)

[7.1.3 Testbench Architecture; Component used (list and describe Drivers, Monitors, scoreboards, checkers etc.) 4](#_Toc157438350)

[7.1.4 Verification Strategy: (Dynamic Simulation, Formal Simulation, Emulation etc.) Describe why you chose the strategy. 4](#_Toc157438351)

[7.1.5 What is your driving methodology? 4](#_Toc157438352)

[7.1.6 What will be your checking methodology? 4](#_Toc157438353)

[7.1.7 Testcase Scenarios (Matrix) 4](#_Toc157438354)

[8 Coverage Requirements 4](#_Toc157438355)

[8.1.2 Assertions 4](#_Toc157438356)

[9 Resources requirements 4](#_Toc157438357)

[9.1 Team members and who is doing what and expertise 4](#_Toc157438358)

[10 Schedule 4](#_Toc157438359)

[10.1 Create a table with plan of completion. You can use the milestones as a guide to fill this 4](#_Toc157438360)

[11 References Uses / Citations/Acknowledgements 4](#_Toc157438361)

# **Introduction:**

## **Objective of the verification plan**

The verification plan for an Asynchronous First-In-First-Out (FIFO) design aims to ensure that the asynchronous FIFO operates correctly and reliably according to its specifications testing the design with both a traditional class-based testbench approach and using a UVM testbench methodology. The key objectives that our verification plan for an Asynchronous FIFO to achieve are:

* Functional Correctness:
  + Verify that data is stored and retrieved from the FIFO in the correct order.
  + Confirm that the FIFO behaves as expected under various data patterns and conditions.
* Asynchronous Operation:
  + Validate that the asynchronous nature of the FIFO is correctly implemented.
  + Check that the FIFO handles data transfers correctly in the absence of a common clock signal between reading and writing processes.
* Empty and Full Conditions:
  + Ensure proper handling of FIFO empty and full conditions.
  + Verify that flags or signals indicating whether the FIFO is full or empty are set and reset correctly.
* Data Integrity:
  + Confirm that data integrity is maintained during read and write operations.
  + Check for potential data corruption, glitches, or metastability issues.
* Corner Cases/Edge Conditions:
  + Test the FIFO under extreme conditions, such as maximum and minimum possible idle cycles for reads and writes, to ensure robustness.
  + Verify that the FIFO gracefully handles Corner cases/edge conditions.
* Cross-Clock Domain Operation:
  + Verify correct operation when dealing with two different clock domains.
  + Checking that synchronization and data transfer mechanisms between two clock domains are properly implemented.
* Compliance with Standards:
  + Ensure that the Asynchronous FIFO design complies with relevant industry standards.

The verification plan serves as a roadmap for systematically testing and validating the Asynchronous FIFO, covering a range of scenarios and conditions to measure its correctness and reliability.

## **Top Level block diagram**

A diagram of a computer system

Description automatically generated

## **Specifications for the design**

|  |  |
| --- | --- |
| Producer Specifications | Consumer Specifications |
| Frequency: 500 MHz | Frequency: 225 MHz |
| Duty-cycle of 50% | Duty-cycle of 50% |
| Number of idle cycles between successive writes = 2 | Number of idle cycles between successive reads = 1 |
| Max Write Burst Size: 1024 |  |

* Idle Cycles: the number of idle Cycles provided are the maximum which either producer can send the data with that number of idle Cycles or the maximum which the consumer can read the data.
  + The number of idle cycles between two successive writes is 2 clock cycles, which means that after writing one data, the producer module is waiting for 2 clock cycles to initiate the next write. Thus, it can be understood that for every 3 clock cycles, one data is written.
  + The number of idle cycles between two successive reads is 1 clock cycle, which means that after reading one data, the consumer module is waiting for 1 clock cycle to initiate the next read.  Thus, it can be understood that for every 2 clock cycles, one data is read.
* Burst length: This is the amount of data sent at once by the producer for one write request and the amount of data read by the consumer for one read request. Burst length of 1024 means when there is a write request from the producer to the fifo, the producer sends $1,024 bits (packets) continuously without any break.
* FIFO depth:  The FIFO depth and timing information was calculated using information from reference 6. Since the FIFO is working with two modules of different clock domains, we need to calculate the depth that is required of the FIFO since the producer system is faster at writing to the buffer than the consumer system is at reading data from the buffer. The depth of the FIFO is determined in a way, so that the FIFO can store all the data which cannot be read by the slower consumer module [6].
  + FIFO operating conditions:
    - The number of idle cycles between two successive writes is 2 clock cycles, which means that after writing one data, the producer module is waiting for 2 clock cycles to initiate the next write. Thus, it can be understood that for every 3 clock cycles, one data is written.
    - The number of idle cycles between two successive reads is 1 clock cycle, which means that after reading one data, the consumer module is waiting for 1 clock cycle to initiate the next read. Thus, it can be understood that for every 2 clock cycles, one data is read.
    - Time required to write one data item = 3 \* (1/500MHz) = 6 ns.
    - Time required to write all the data in the burst = 1024 \* 6ns = 6144 ns.
    - Time required to read one data item = 2 \* (1/225MHz) = 8.89 ns.
    - For every 8.89 ns, the consumer module is going to read one data in the burst.
    - In a period of 6144 ns, 1024 numbers of data items can be written.
    - The number of data items that can be read in a period of 6144 ns = (6144ns/8.89ns) = 691.2
    - The remaining number of bytes to be stored in the FIFO = 1024 – 691 = 333.
    - The FIFO must be capable of storing 333 data items, so the minimum depth of the FIFO should be 333.
* FIFO size: for 333 locations, we will need FIFO of 512 locations as memories are available only in powers of 2. Our data width for read and write is 8 bits, so total memory size required for fifo would be 512 \* 8 bits = 512 KB of memory.
* FIFO Memory access: We assume FIFO addresses are encoded in gray code and not binary so that the counters can easily access the next memory location on increment.
* Handshaking Protocols: We can go with a half-interlocked source synchronous protocol for both read and write. The host will send a read or write enable signal along with the number of idle cycles as well to FIFO and wait till the FIFO sends an acknowledgment that read ready or write ready after which data transfer happens. When there is a request to FIFO, it calculates if a write or read transaction is eligible to happen serving the whole burst length. For eg., if the FIFO gets a write enable signal with 0 idle cycles, it checks if it can fit 1024 packets with 2ns frequency, based on the current read activity in case it is in progress. Only if the FIFO can accommodate the burst in the case of no read request after the burst starts, the acknowledgement will be sent to the producer to send the data. Same with read request as well. If there is a write which has just started and the read request is with 0 idle cycles, it will calculate and only after it reaches the required amount of data available in the FIFO queue, it sends an acknowledgement to start the read activity.
* Producer module: This module would receive data from testbench and sends the wr\_en signal to fifo and waits till it receives wr\_rdy signal from fifo to send the data to FIFO. This functionality can also be implemented via testbench and remove the module itself. We can decide on this.
* Consumer module: Same with consumer as well. This module sends rd\_en signal to fifo and starts reading data after it receives rd\_rdy signal. It can either store the data in a table or send it to testbench which would in turn display it. We can also do it via testbench. To be finalized.

# **Verification Requirements**

## **Verification Levels**

### **What hierarchy level are you verifying and why?**

Unit and Core Level

* Unit: Collection of Designer Level blocks that performs specific function
* Core: Collection of Units, can be used across multiple designs

The hierarchy levels that we are verifying are the Unit and Chip levels. We are verifying at these levels so that we can first verify at the unit level that the producer, consumer, FIFO, and control unit (write and read enable) all work. After verifying that all the separate components pass their Unit Level testing we want to conduct Core level testing of our whole design, which connects all the individual units together.

### How is the controllability and observability at the level you are verifying?

Controllability is the measure of ease at which the DVE can create the specific scenarios that are of interests (inputs), which for our design is very high since it is easier to control at a smaller level when compared to a higher level and Unit level testing is one of the lowest levels. Observability is the measure of ease at which the DVE can identify the incorrect behavior of the design (outputs), which for our design is also high for the same reasoning as the controllability of our design since we are verifying at the Unit level first.

### Are the interfaces and specifications clearly defined at the level you are verifying. List them.

The system interfaces with external entities, including:

Input Devices: Devices or systems providing data inputs to the Input Module.  
Output Devices: Devices or systems receiving processed data from the Output Module.

* Producer Module: The producer module in the FIFO system is responsible for managing the input data stream and efficiently writing data into the FIFO buffer. Its primary objective is to ensure a seamless and controlled flow of data from the source to the FIFO, adhering to the specified data transfer protocols.
* Data Input: The producer module accepts incoming data from an external source, which could be a sensor, another module, or an external communication interface.
* Data Formatting: It is equipped to format or preprocess the incoming data, ensuring compatibility with the data format expected by the FIFO.
* Write Data to FIFO: The producer module controls the initiation of write operations to the FIFO. It manages the timing and sequencing of these operations to prevent data collisions and maintain the integrity of the stored data.
* Write Enable Signal: Status signals indicate the readiness of the producer, the occurrence of write operations, or any relevant status updates. The timing and synchronization ensure that the data is correctly captured by the FIFO.
* FIFO Module: The FIFO module within the system is designed to manage the orderly and sequential storage and retrieval of data. It follows the First-In-First-Out principle, ensuring that data written into the FIFO is read out in the same order. The module serves as a crucial buffer for handling data transfer between components that may operate at different rates or in different clock domains.
* Sequential Data Storage: The FIFO module stores incoming data in a sequential manner, preserving the order in which it was written. This ensures that the oldest data is read out first.
* Write and Read Pointers: The module employs write and read pointers to keep track of the current write and read positions within the FIFO buffer. These pointers determine where the next write or read operation will occur. . Our current iteration of the asynchronous FIFO utilizes a binary counter instead of a gray code counter. If time permits going forward we will try to implement a gray code counter.
* FIFO Depth: FIFO depth is 792 as calculated in section 2.3. It means that the fifo can store 792 data chunks at any particular point of time. We will store
* Clock Domain Synchronization: To handle scenarios where the FIFO interacts with components in different clock domains, the module incorporates proper synchronizers. These synchronizers mitigate risks associated with clock domain crossings.
* Control Signals: The FIFO module utilizes control signals, such as write enable (WE) and read enable (RE), to coordinate data transfer operations. These signals facilitate the initiation of write and read operations.
* Full and Empty Flags: The module maintains flags indicating whether the FIFO is full or empty. These flags assist in flow control and prevent potential overflow or underflow situations.
* Control Unit: The FIFO Control Unit serves as the central coordinating entity for the FIFO module, managing the orderly flow of data into and out of the buffer. It is designed to ensure synchronized and error-free communication between the producer and consumer modules while maintaining the integrity of the stored data.
* FIFO Consumer Module: The FIFO Consumer Module is designed to efficiently retrieve and process data from the FIFO buffer while maintaining synchronization with the FIFO Control Unit and the overall system. Its primary function is to ensure the orderly consumption of data based on the First-In-First-Out principle.
* Read Operations: The consumer module initiates read operations to retrieve data from the FIFO buffer. It adheres to the sequential order of data storage, reading the oldest data first.
* Data Processing: Processes the retrieved data according to the intended functionality. This may involve data analysis, transformation, or further transmission to downstream components.

### **FIFO Memory Buffer**

A white background with black text

Description automatically generated

This is the memory buffer that is accessed by both the write and read clock domains. The FIFO memory buffer has a set data size (memory data word width) and address size (number of memory address bits). The inputs to the FIFO memory buffer is the data to be written (wdata), the address to be written (waddr), the address to be read (raddr), the clock enable signal from the producer (wclken), the write buffer is full signal (wfull), and the clock signal from the producer (wclk). The output of the FIFO memory buffer is the data to be read (rdata). The output rdata is assigned the value from raddr and at the positive edge of the producer clock, the waddr is assigned the value from wdata [1].

## **A pink and white diagram Description automatically generated with medium confidenceSynthesizer Module (Read-domain to Write-domain Synchronizer)**

Synchronizer module that is used to synchronize the read pointer into the write clock domain through a pair of registers that are clocked by the write-clock. The synchronized read pointer (rptr) will be used by the wptr\_full module to generate the FIFO full condition This module only contains flip-flops that are synchronized to the write clock. No other logic is included in this module [1].

## **Synthesizer Module (Write-domain to Read-domain Synchronizer)**

There is another synchronizer module that is used to synchronize the write pointer into the read clock domain through a pair of registers that are clocked by read-clock. The synchronized write pointer will be used by the rptr\_empty module to generate the FIFO empty condition. This module contains flip-flops that are synchronized with the read clock. No other logic is used in this module [1]. The write pointer uses Gray code counter logic discussed in Section 3.1. The same 2 flip-flop synchronizer is used for the Write-domain to Read-domain Synchronizer as the Read-domain to Write-domain Synchronizer used in section 4.3, Figure 6, except that the input write/read will be reversed. For example, rptr = wptr, wclk = rclk, wq2\_rptr = rq2\_wptr, etc.

## **Read Pointer, Empty Generation Logic, and Almost Empty Logic**

A pink rectangle with text

Description automatically generated  
This module is completely synchronous to the read clock domain and contains the FIFO read pointer and empty flag logic [1]. The read operation is controlled by the read pointer which points to the location that the next data is read from. This module uses the dual n-bit Gray code counter discussed in *Section 3.1* to generate the read pointer and read address values. When the FIFO read operation takes place and the rptr signal is incremented, the FIFO pointers are no longer equal t o each other so the full signal is de-asserted. Because the two empty and full signal changes occur on two different clock domains, the 2 flip-flop synchronizer described in *Section 4.2, Figure. 6,* is needed to remove the metastability that could be generated [7]. The FIFO design in this paper assumes that the empty flag will be generated in the read-clock domain to ensure that the empty flag is detected immediately when the FIFO buffer is empty, that is, the instant that the read pointer catches up to the write pointer (including the pointer MSBs) [1]. The empty signal is asserted when the synchronized read and write pointer are equal. To do the empty comparison, pointers that are one bit larger than needed for the FIFO memory buffer are used, and if the n+1 bit of the read and write pointers are equal to each other, that means that the pointers have wrapped around the FIFO buffer the same amount of times and a comparison of the rest of the pointer’s bits are done to see if they are equal. If the bits are all equal the FIFO is assumed to be empty [1].  The almost empty signal is asserted when the read pointer is incremented multiple times without new writes.

## **Write Pointer, Full Generation Logic, and Almost Full Logic**

A white rectangular object with pink text

Description automatically generated with medium confidence  
This module is completely synchronous to the write clock domain and contains the FIFO write pointer and full flag logic [1]. The write operation is controlled by the write pointer which points to the memory location where the next data is written to. This module uses the dual n-bit Gray code counter discussed in *Section 3.1* to generate the write pointer and write address values. When the FIFO write operation takes place and the wptr signal is incremented, the FIFO pointers are no longer equal to each other so the empty signal is de-asserted. Because the two empty and full signal changes occur on two different clock domains, the 2 flip-flop synchronizer described in *Section 4.2, Figure. 6,* is needed to remove the metastability that could be generated [7]. The FIFO design in this paper assumes that the full flag will be generated in the write-clock domain to insure that the full flag is detected immediately when the FIFO buffer is full, that is, the instant that the write pointer catches up to the read pointer (except for different pointer MSBs) [1]. The full signal is asserted by running a comparison between the write and read pointers after they have been synchronized. To ensure that there are no issues, a comparison is made after synchronizing the read pointer into the write clock domain. After that, three comparisons are made. The first is if the write pointer and the read pointer’s most significant bit (MSB) are not equal. Second, comparing the write pointer and read pointer’s 2nd MSB are not equal. Finally, checking if all the other write pointer and read pointer bits are equal [1]. The almost full signal will be asserted when the write pointer has been incremented multiple times in a row.

# Required Tools

## List of required software and hardware toolsets needed.

* All the programming will be done in SystemVerilog on QuestaSim using Portland State University’s remote computer lab
* The processor on the provided remote computers is an Intel(R) Xeon(R) CPU E3-1241 v3 @ 3.50 GHz 3.50 GHz with 16 GB of installed RAM.
* The operating system running is Windows 10 Enterprise version 22H2 and is a 64-bit operating system (x64-based processor)
* Questa Sim is a multi-language environment by Siemens (previously developed by Mentor Graphics) for simulation of hardware description languages such as VHDL, Verilog and SystemVerilog.
* The version of Questa Sim that is on the remote computers provided by Portland State University is Questa Sim-64 2023.3. Revision 2023.07 Date: Jul 17 2023.
* Included Libraries on Questa Sim:

**A screenshot of a computer program

Description automatically generated**

## Directory structure of your runs, what computer resources you will be using.

* The directory structure for simulation includes SystemVerilog source code files, UVM testbench components, simulation results, coverage reports, and final documentation.
* As a part of computer resources, the processor used on the provided remote computers is an Intel(R) Xeon(R) CPU E3-1241 v3 @ 3.50 GHz 3.50 GHz with 16 GB of installed RAM.
* The operating system running is Windows 10 Enterprise version 22H2 and is a 64-bit operating system (x64-based processor)
* The version of Questa Sim that is on the remote computers provided by Portland State University is Questa Sim-64 2023.3. Revision 2023.07 Date: Jul 17 2023.

# Risks and Dependencies

## List all the critical threats or any known risks. List contingency and mitigation plans.

Critical Threats: Metastability which is the condition where a signal’s value is unknown and is between logic values of 0 and 1 due to a timing violation. In metastable states, the circuit may be unable to settle into a stable '0' or '1' logic level within the time required for proper circuit operation. As a result, the circuit can act in unpredictable ways, and may lead to a system failure, sometimes referred to as a "glitch" [8].

Mitigation plan: Use Gray code pointers to handle and synchronize the FIFO pointers and flags of different clock domains.

# Functions to be Verified.

## Functions from specification and implementation

### List of functions that will be verified. Description of each function

* Verify FIFO
  + Write arbitrary values in succession and verify we can read the same values back in order
* Only reading
* Only writing
* Reading and writing at the same time
* Write full scenarios
* Read empty scenarios
* Checking reset behavior
  + Check reset of write only
  + Check reset of read only
  + Check both write and read simultaneously

### List of functions that will not be verified. Description of each function and why it will not be verified.

At the current time of writing this verification plan there are no functions that we can think of not to verify.

### List of critical functions and non-critical functions for tapeout

* Critical Functions
  + FIFO is able to read and write at the same time
  + Reset and initialization of the FIFO will trigger the FIFO empty and de-assert the FIFO full signal
  + Write entry to the FIFO will de-assert the FIFO empty signal
  + Read entry from the FIFO will de-assert the FIFO full signal
* Non-critical functions
  + FIFO almost empty signal
  + FIFO almost full signal
    - Will think about implementing those signals if we can get the critical functions to work correctly

# Tests and Methods

### Testing methods to be used: Black/White/Gray Box.

White box testing will be used since we have the knowledge of the internal data structures, physical logic flow, and architecture of the design and source code.

### State the PROs and CONs for each and why you selected the method for this DUV.

* Black box: Black box testing is a type of testing in which the functionality is not known. The testing is done without the internal knowledge of the products. It is also called Functional testing. Black-box testing focuses on external attributes and behavior. This type of testing looks at an application’s expected behavior from the user’s point of view [10].
* White box: White-box testing or glass-box testing is a testing technique that uses the knowledge of internal data structures, physical logic flow, and architecture at the level of source code. This testing works by looking at testing from the developer’s point of view. This testing is also known as glass box testing, clear box testing, structural testing, or non-functional testing [10].
* Gray box: Gray Box Testing is a combination of the Black Box Testing technique and the White Box Testing technique in software testing. The gray-box testing involves inputs and outputs of a program for the testing purpose but test design is tested by using the information about the code. Gray-box testing is well suited for web application testing because it factors in a high-level design environment and the interoperability conditions [10].

### Testbench Architecture; Component used (list and describe Drivers, Monitors, scoreboards, checkers etc.)

### A diagram of a design process Description automatically generated

Conventional testbench that will check the basic functioning of the RTL for milestone one.  
Conventional testbench that will have sections for a driver (stimulus generator) and a monitor (checker and coverage monitor) for milestone two. Will need to have drivers test case generators, monitors, scoreboards, and checkers.   
Testbench will use SVA and covergroups and bins for further milestones and the final submission.

UVM test architecture will be the final form.

### Verification Strategy: (Dynamic Simulation, Formal Simulation, Emulation etc.) Describe why you chose the strategy.

Functional simulations to verify that individual unit modules/blocks in the design are working correctly.

Dynamic simulation running directed tests, random tests, and coverage analysis to verify that the individual units work correctly in a block.

No emulation is required in our current design.

### What is your driving methodology?

#### List the test generation methods (Directed test, constrained random)

* Here, we are planning to test the DUV using both testing methods
* Directed tests
* Constrained Random tests

### What will be your checking methodology?

#### From specification, from implementation, from context, from architecture etc

From Specification:

* Verify the implemented design meets the specifications outlined in the HLDS document.
* Develop test scenarios based on HLDS and use assertions to check if design aligns with documented requirements.

From Implementation:

* Focusing on correctness of design implementation, checking the code represents the intended functionality.
* Using dynamic simulation with testbenches that include various stimuli to test the design and assertions to monitor its behavior during simulation.

From Context:

* Ensuring that the design functions correctly within the environment it is intended for.
* Considering its interactions with other components and verifying its behavior in a system-level simulation.

From Architecture:

* Verifying that the design is according to specific architectural constraints.
* Considering aspects such as data flow, control flow and resource utilization to ensure the design aligns with the intended architecture.

### Testcase Scenarios (Matrix)

#### Basic Tests

|  |  |
| --- | --- |
| Test Name / Number | Test Description/ Features |
| 1.1.1 | Check basic write operation |
| 1.1.2 | Check basic read operation |
| 1.1.3 | Reset during FIFO operation |
| 1.1.4 |  |

#### Complex Tests

|  |  |
| --- | --- |
| Test Name / Number | Test Description/ Features |
| 1.2.1 | Concurrent events (R+W)  Conditions: fifo\_full/fifo\_empty/always\_full/always empty etc. |
| 1.2.2 | Simultaneous write and read operations |
| 1.2.3 | Write happens and the buffer is empty |
| 1.2.4 | Read happens and the buffer is empty |
| 1.2.5 | Write happens and the buffer is full |
| 1.2.6 | Read happens and the buffer is full |
| Test Name / Number | Test Description/ Features |

#### Regression Tests (Must pass every time)

|  |  |
| --- | --- |
| Test Name / Number | Test Description/Features |
| 1.3.1 | Tests that should always pass |
| 1.3.2 | Need to come up with more |

#### Any special or corner cases testcases

|  |  |
| --- | --- |
| Test Name / Number | Test Description |
| 1.4.1 | Special Case testing tests and conditions |
| 1.4.2 | Bug injection and testing scenario |

# Coverage Requirements

#### Describe Code and Functional Coverage goals for the DUV

* Code coverage goal is 100%
  + Code and structural (Implicit coverage): executed by tests during simulation
    - metrics that are automatically derived and extracted from RTL
  + line, statement, toggle, branch, etc.
    - Line coverage: lines of our source code have been executed during simulation. Need to make sure every line is covered.
      * does not check if the code is “correct” just checking to see if we executed every line of code written
    - Statement coverage: measures the % of code statements executed during simulation
    - Block coverage: Identifies if a block of code has been exercised. Set of statements between conditional or procedural statements
      * Measures if the blocks are exercised during simulation for the given tests, not their correctness
    - Branch/decision coverage: Measures the percent of control structures exercised (if, case, repeat, for, etc.)
      * Does not check if the branch reason is correct, just checking to see if all the decision points are taken
    - Expression coverage: Measures the percent of condition expression that are executed both true or false
    - Focused expression coverage: Often used by DO-1788 safety critical software certification standard (We probably don't need)
    - FSM code coverage: Measures the percent of FSM states and transitions that have been exercised (We probably don’t need)
    - Toggle coverage: Measures the percent of signal transition observed during simulation (We probably don’t need)
* Functional (Explicit coverage): measures the features and functionality executed by the test during simulation. Identifies the functionality/features to be checked.
  + Metrics that are manually defined and then implemented by the engineer
  + covergroups/coverpoints/assertions.
    - Cover group modeling
      * Grouping of interface signals
      * Grouping of registers
    - Cover property modeling
      * sequences and events relationships
        + handshaking sequence between control signals
    - Assertion coverage
      * measures assertion properties
* The environment has exercised all types or commands and transactions
* The stimulus has a varying range of data types
* The environment has driven varying concurrent stimulus
* Bug injection and capture has been verified

#### Formulate conditions of how you will achieve the goals. Explain the Covergroups and Coverpoints and your selection of bins.

The specific bins have yet to be determined for milestone one, but will follow the general format below:

A diagram of a company

Description automatically generated

### Assertions

#### Describe the assertions that you are planning to use and how it will help you improve the overall coverage and functional aspects of the design.

* The write pointer should never wrap around and overwrite valid data.
* The read pointer should never wrap around and read stale data.
* The number of elements in the FIFO should never exceed the depth of the FIFO.
* When the FIFO is full, the write operation should not be allowed.
* When the FIFO is empty, the read operation should not be allowed.
* When reset is asserted the FIFO will be cleared of all content and the write and read pointer will start at the first location.

# Resources requirements

## Team members and who is doing what and expertise.

|  |  |
| --- | --- |
| [Deepthi Chevuru](mailto:chevuru@pdx.edu) | **Design Code**, Verification Plan, HLDS, and testbench |
| [Sai Sukitha Puli](mailto:puli2@pdx.edu) | Design Code, Verification Plan, HLDS, and **testbench** |
| [Masaaki Ishii](mailto:masaaki.ishii@intel.com) | Design Code, **Verification Plan, HLDS**, and testbench |

Bold is the expertise

# Schedule

## Create a table with a plan of completion. You can use milestones as a guide to fill this.

|  |  |
| --- | --- |
| Milestone | Due Date |
| Form Group | 01-18-24 |
| Finalize Design Implementation of Project (HLDS, Design Code, and Verification Plan) | 02-03-24 |
| Milestone 2 (TBD what is needed) | 02-10-24 |
| Milestone 3 (TBD what is needed) | TBD |
| Final Presentation (Final Presentation Slides) | 03-07-24 or 03-12-24 or 03-14-24. Need to sign up for a slot once they open up |
| Final Project Submission (Academic Paper, Presentation, HLDS document, Verification Plan, and HDL Code) | 03-07-24 (Can submit everything on 03-05-24 for 10% extra credit) |

# References Uses / Citations/Acknowledgements.

 [1] Clifford E. Cummings, “Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs,” SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers, March 2001, Section MC1, 3rd paper.

[2] Clifford E. Cummings and Peter Alfke, “Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons,” SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers, March 2002, Section TB2, 3rd paper

[3] “FIFO (Computing and Electronics).” Wikipedia, Wikimedia Foundation, 2 Sept. 2023, en.wikipedia.org/wiki/FIFO\_(computing\_and\_electronics).

[4] Vlsiverify. (2022, December 26). Asynchronous FIFO. VLSI Verify.https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/

[5] https://research.ijcaonline.org/volume86/number11/pxc3893347.pdf

[6] https://hardwaregeeksblog.files.wordpress.com/2016/12/fifodepthcalculationmadeeasy2. pdf

[7] Ramesh, G. (n.d.). Asynchronous FIFO Design with Gray code Pointer for High Speed AMBA AHB Compliant Memory controller. IOSR Journal of VLSI and Signal Processing (IOSR-JVSP), 1(3), 32–37. <https://doi.org/www.iosrjournals.org>

[8]  Chaney, Thomas J. "My Work on All Things Metastable OR Me and My Glitch" (PDF). Archived from the original (PDF) on 2015-12-08. Retrieved 2015-11-05.

[9] <https://research.ijcaonline.org/volume86/number11/pxc3893347.pdf>

[10] https://www.geeksforgeeks.org/difference-between-black-box-vs-white-vs-grey-box-testing/