**Matmul Name**

# Matmul Digital High Level Design Version 0.1

# Revision Log

| Rev | Change | Description | Reason for change | Done By | Date |
|---|---|---|---|---|---|
| 0.1 | | A verification plam of Matmul design | | Noan Guez<br>Nadav Rozenfeld | 1,Mars,2024 |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

# Table of Content

# 1. LIST OF FIGURES

# 2. LIST OF TABLES



# 3. VERIFICATION PLAN

Our verification strategy primarily hinges on the automated numerical comparison between the Device Under Test (DUT) and a gold standard model we've constructed using Python's NumPy. To facilitate this automated testing, we have developed a Python-based randomizer that generates a file containing control registers and matrices in a specific sequence, ensuring both our test stimuli and the gold standard model can interpret it correctly. To guarantee effective communication between the test stimuli and the DUT via the APB master-slave communication protocol, we have created a checker and coverage tools. The foundational elements for all these testing modules are within a package and an interface, which detail all necessary parameters and signals. Further elaboration on these components will be provided in Section 1.2.

The sole aim of this test bench is to execute the widest range of verification tests possible on the DUT, that we've created in part 1 of the project. To achieve this, it was essential to first ensure that all verification test modules were functioning flawlessly to facilitate the correction of the DUT. In this section, we will detail all the tests that have been conducted on our module and successfully passed.

| test num | test name | functionality being tested | test data set | expected result | special observations |
|---|---|---|---|---|---|
| 1 | writing to the fifo's using apb slave | fifo, apb slave | basic param*<br>basic matrix* | full fifos of right data | none (other than that that nothing worked and we debugged it all ) |
| 2 | calculating a result and writing it to scratchpad | PEs, SP, fifo and inner connections | basic param*<br>basic matrix* | data seeps through in PE and result written in SP | none (other than that that nothing worked and we debugged it all ) |

| | | | | | |
|---|---|---|---|---|---|
| 3 | reading all values from the matmul | reading abilities to all modules (SP, A and B matrix, flags and Control reg) | basic param*<br>basic matrix* | seeing all the values in the stimulus | none (other than that that nothing worked and we debugged it all ) |
| 4 | calculating a matrix using bias from SP | PE bias ability and inner connections with SP | basic param*<br>reading two matrices and a control from file | seeing correct results with the bias | none |
| 5 | calculating a matrix using reload bits | reload ability in fifo and inner connection. | basic param*<br>reading the other matrix not reloaded and control reg | accurate values of calculation | was very complicated to make this work |
| 6 | calculating various matrices automatically (200 matrices) | TESTING EVERYTHING!! | basic param*<br>reading 200 matrices + control reg generated from randomiser | perfect comparison of DUT and golden model | |
| 7 | validating the output using checker assertions | APB master-slave communication and inner verification testing | basic param*<br>simple reading file | non assertion's errors | Couldn't run this test because of servers error (more to it in 3.4) |
| 8 | creating automatic tests (6) on all possible scenarios of the basic parameters | all of what 6 is testing on all the possible parameters: EVERYTHING$^2$ | all possible parameters combinations and generated file for each | perfect comparison of DUT and golden model | |

*the basic param: BW-32 DW-16 AW-32 SPN-4.

*the basic matrix- a 2on2 matrix we wrote at stimulus for basic tests

It should be noted that a significant portion of this project was creating internal tests on the test bench itself to ensure its validity, including the use of checker and coverage tools for the stimulus, as well as the randomizer for both the stimulus and the gold standard model and more. These crucial components, though not detailed in the table above, form the backbone of our testing strategy. In the following sections of this document, we will explore into every aspect of our test bench, discussing the purpose, challenges, and functionality of each file.

## 3.1 Verification Test Objectives

As described earlier, the primary aim and the purpose of our test bench is to thoroughly evaluate our DUT across the broadest spectrum of scenarios, ensuring that our module consistently delivers accurate outputs. Recognizing this as the essence of that part, and understanding that this is as well the longest part to accomplish, we strategically planned to develop the test bench sequentially, allowing us the longest time possible to debug our module and simultaneously refine our verification tools. Hence, our first goal was to establish a reliable APB master within our stimulus verification module, involving the creation of a basic interface, package, and TB top.

Upon achieving this, we progressed simultaneously on five parallel objectives, which we pursued in a semi-independent manner, acknowledging the interdependence inherent within the project:

1. Module Debugging: This, in our view, was the paramount and most time-consuming objective. Ideally, debugging would commence with a flawless test bench in place, but given our time constraints, we chose to start as early as feasible.

2. Randomizer Development: To enhance the efficiency, breadth, and quality of the test files for both the stimulus and later the golden model, we decided to develop a Python-based randomizer, moving towards automating our testing processes.

3. Assertion and Coverage: Implementing checker and functional coverage modules was crucial for verifying precise communication between the stimulus and the DUT via the APB buses. This step was vital for diagnosing both fundamental and complex APB protocol issues. sadly eventually this part wasn't as significant as we hoped it to be because of the server crushing. detailed further in Sections 3.4 and 3.5.

4. Golden Model Creation: The cornerstone of our testing capability lies in the golden model's ability to compute expected DUT responses and automatically verify these calculations, with more details provided in Section 3.6.

5. upgrading our DUT: In the previous section of the project (design level) we did’nt implement the APB salve in FSM architecture. After we consulted with the instructor we realized that an FSM implementation will fit better to both AMBA spec and performance.

These objectives shaped the focus of this project phase. Having met these goals, we integrated them into the top TB and proceeded with extensive testing. The following sections will offer a more detailed exploration of each testing module.

## 3.2 Test Bench High Level Diagram and Architecture Example:

Our verification environment is compromised from the following parts:

- Randomizer – randomize the control register, and A, B matrices. It's a python script that generates a data text file (data_file.txt).
- Stimulus – Reads the tests flow generated by the Randomizer and feeds the DUT. The Stimulus sends the data to the DUT by acting as an APB master. It also reads the results from DUT via APB transactions.
- Golden model – The golden model is a python script that reads the same data from the data text. It calculates the results as defined in the Matmul spec. It writes the results into another text file (golden.txt). In addition, there is a component (matmul_golden.sv) that compares the results to the DUT's.
- Interface – system Verilog interface that connects the verifications components with DUT.
- Checker – Assertion indicating on critical cases of the design. Implemented in SVA
- Coverage – functional coverage in System-Verilog.

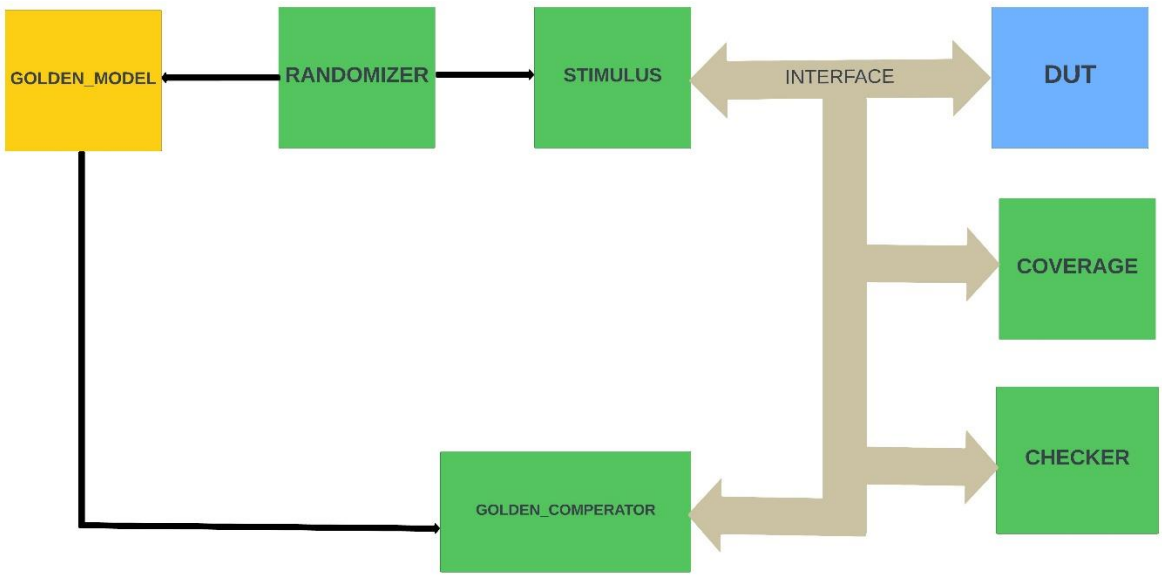


Figure 1: Test Bench High Level Diagram

## 3.3 Test Bench Low Level Architecture and Functionality

STIMULUS

The Stimulus is verification component implemented in system-Verilog that feeds the tests data to the DUT. It reads the data from data.txt file. This text file is generated either automatically by the randomizer or manually by the verificator.

After the stimulus reads an A matrix, a B matrix, and a control register from the text file it starts sending the data (in the same order) to the DUT. The transactions are according to both APB and MATMUL specifications (I.E stimulus acts as APB master, sends to relevant addresses etc.).

After it finished sending the control register, the stimulus waits to the 'busy' (DUT's output) to rise and fall, meaning that DUT started the calculations and finished them. After 'busy' falls, the stimulus reads the results from the DUT via APB read transactions (put the address of the scratchpad). The results go from the stimulus to the golden.sv component for comparation with golden design results.

* A Note: if the control register contains a reload demand, the stimulus will challenge the DUT by not sending him a new A / B matrix before the control.


Figure 2: Stimulus concept Diagram

GOLDEN MODEL

We implemented a golden model by python. It reads the same data as the DUT and calculates the results according to Matmul spec. The static parameters are given in the beginning of the script as constants. The golden reference manages his own Scratchpad, exactly like the DUT.

In addition to the golden model script, there is also an sv component – matmul_golden.sv. It reads the results from the outputs generated by the golden script and compares them to the results from the DUT. It

compares the two results matrices (one from golden, the other from DUT) by counting the number of elements that are equal. Only if this number is N * M the test is considered a success.

The verification module matmul_golden.sv waits for a signal “intf.read_results_dut” which indicates that the Stimulus has finished reading a result matrix from the DUT. Then he reads the next matrix from the reference mode (from golden.txt). Then he performs the elements comparations and finds if the matrixes are equal.

It counts the amount of success test and manages an indicator vector in the size of the test amount (configured in the package). Each bit k in the indicator vector is set only if the test k passed successfully.

* A Note : both randomizer and golden model are implemented in the same script “golden.py”. For running the golden model only, (without generating new randomized data) the verificator should run the command - “golden.py 0”. The verificator can also randomize new data before the golden is activated by running the command – “golden.py 1 <x>” when x is the number of tests (each test is A, B + control register).

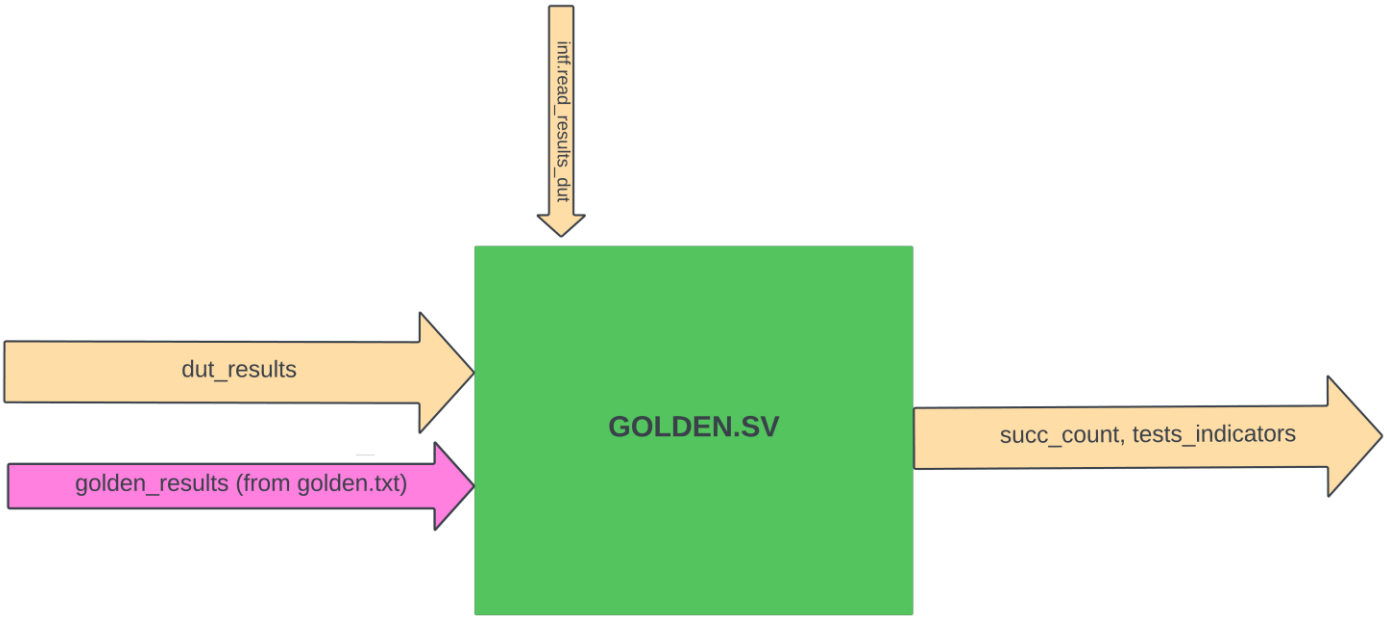


Figure 3: Golden concept Diagram

Interface

The Interface is a system Verilog component that connects the verifications components with DUT. It simulates an APB bus connecting the master (Stimulus) to the slave (DUT). It also transforms relevant data from the DUT (read by Stimulus) to other verification components such golden comparator, SVA and Coverage.

Checker

The checker which is implemented in System Verilog Assertions (SVA) responsible for watching that the DUT functions as expected and indicate if a forbidden behavior is happening. by raising assertions. We used the assertion approach to verify that the APB slave is behaving according to AMBA spec.

## 3.4 Functional Coverage

Unfortunately, those sections of functional coverage and checker weren't fundamental to this project because we couldn't work with them because the servers were down and we haven’t had access to the Questasim app(aren’t back to this point). but as this is an important part of any verification test bench those are the coverages and later the assertions we've implemented.

| Function | Event | Cover Point | Bins | Scenario |
|---|---|---|---|---|
| Reset Signal Transitions | Positive edge of pclk | rst_n | reset_asserted, reset_deasserted | Standard |
| Transaction Type | Positive edge of pclk | pwrite | write, read | Standard |
| Address Specific Bins | Positive edge of pclk | paddr[4:0] | A_mat, B_mat, scratchpad, control_reg | Standard |
| Error Handling | Positive edge of pclk | pslverr, pwrite | no_error, error, cross of error with read/write | Standard/Extreme |
| Strobe Size | Positive edge of pclk | pstrb | size1, size2, size4 | Standard |
| Covering bias | Positive edge of pclk | Pwdata(control reg) | Bias, no bias | Not implemented |
| Covering n,m,k | Positive edge of pclk | Pwdata(control reg) | K=2,3,4<br>M=2,3,4<br>N=2,3.4 | Not implemented |

Table 1: *Test Plan Functional Coverage*

## 3.5 Test Bench Functional Checkers

As explained in 3.4, this is our checker which we developed before the server crush but haven't had the opportunity to run it.

| Condition | Event | Expected Result | Scenario |
|---|---|---|---|
| Transaction starts with psel high and penable low (setup) then changes to penable high (access) | On the positive edge of pclk when psel is high | Transaction must only start when penable is low (setup) only then penable rises (access) | Standard |
| Transaction completes correctly | On the positive edge of pclk when pready is high | penable must be deasserted after transaction completion (from access to idle or back to setup if sel is stable) | Standard |
| Write data is defined during a write transaction | On the positive edge of pclk during a write transaction | Write data (pwdata) must not be 'x' (undefined) | Standard |
| Read data is defined during a read transaction | On the positive edge of pclk following a read transaction | Read data (prdata) must not be 'x' (undefined) after pready is asserted | Standard |
| address is defined during a write transaction | On the positive edge of pclk following a read transaction | address (paddr) must not be 'x' (undefined) after pready is asserted | Standard |
| buses are stable when penable is high | On the positive edge of pclk when penable is high | psel, paddr, pwrite, pwdata, pstrb must remain stable | extreme |
| pready eventually asserted after penable in maximum of two cycles | On the positive edge of pclk after penable is high | pready must be asserted eventually, indicating the slave is ready to complete the transaction | Standard |
| pslverr asserted under valid conditions | On the positive edge of pclk when pslverr is asserted | pslverr can only be asserted when both psel and penable are high | Standard |

Table 2: *Test Plan FunctionalCheckers*

### 3.6 Golden Model

Here we will elaborate the .txt files structures. All values are in decimal representation.

data_file.txt

The file data_file.txt can be generated either by randomizer, or manually by user ( see *Note in section 3.3).

Each test in data_file.txt starts with “#control” label. Then there is a number in a separate line which is the control value. Then comes label ‘A’ and A matrix, followed by ‘B’ and another matrix. Each test ends with the label “END”.

The matrices  values in the same line are separated by ‘ ‘ (space).

golden.txt

This is the results file from the golden model. It is built in a similar way as the data_file.txt.

# 4. APPENDIX

## 4.1 Terminology

**DUT** - Device Under Test

**SP** - ScratchPad

**LSB** - Least Significant Bit

**TBR** - To Be Reviewed

**TBD** - To Be Defined

**IF** - Inteface

**BW** - Bus_Width parameter

**DW** - Data_Width parameter

**AW** - Address_Width parameter

**SPN** - size of SP parameter