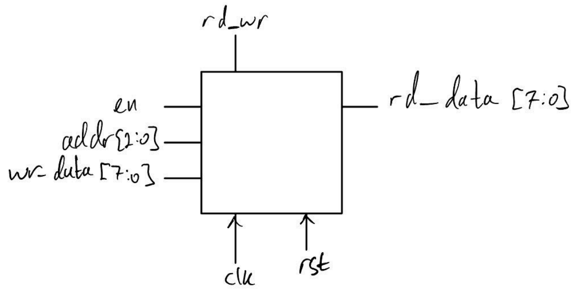
Memory Verification Plan:

Our Design (DUT) is a Memory Model:
The Memory model is capable of storing 8bits of data per address location Reset values of each address memory location is ‘hFF.



Input Signals:
clk - clock signal
rst - reset signal
addr[2:0] - Address signal on which the address is specified
enable - indicated that a write/read operation can be performed
rd_wr - write operation 0, read operation 1
write_data[7:0] - signal for write data
Output Signals:
rd_data[7:0] - signal for read data

We will use this signals direction in our Interface.

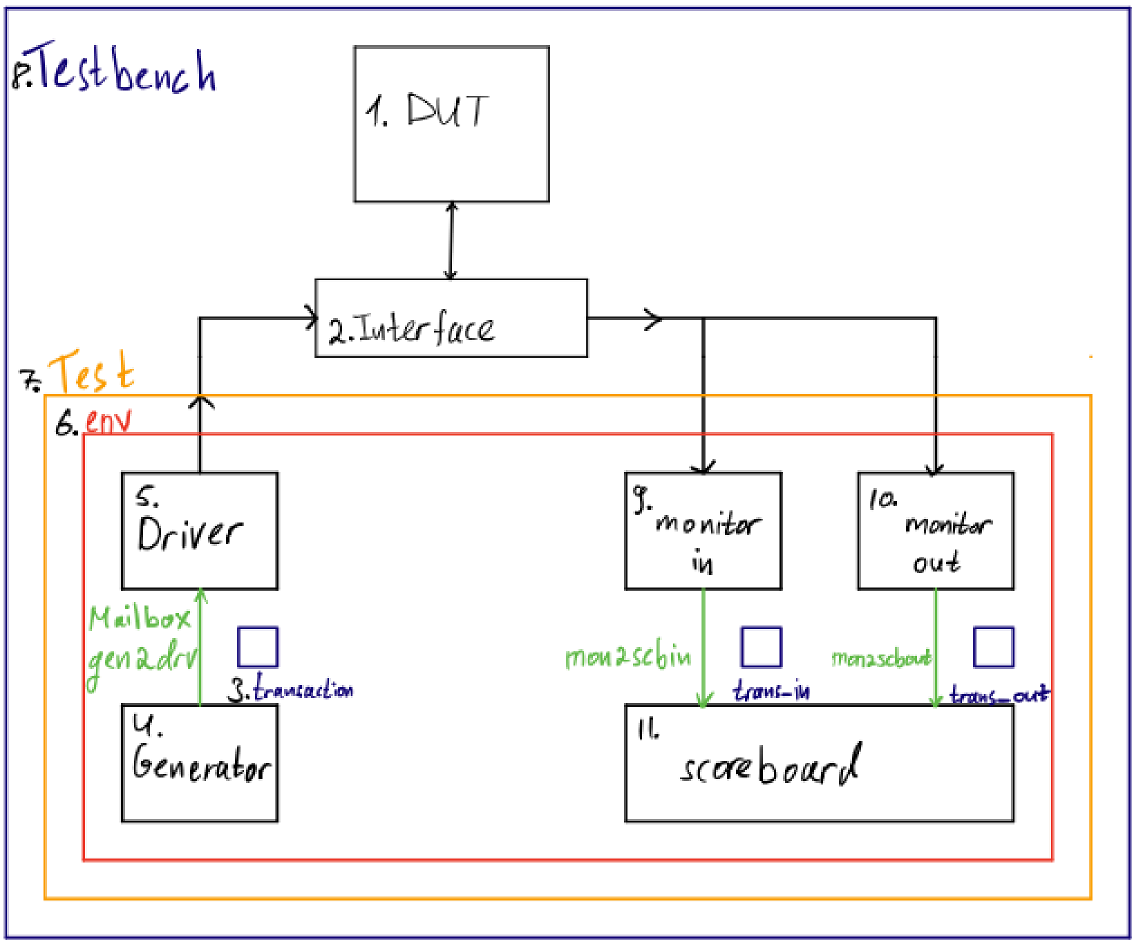
Operations:

Write - address, rd_wr = 0, enable and wr_data should be driven at the same clock cycle.
Read - address rd_wr = 1, enable should be driven on the same clock cycle, design will respond with the rd_data in the next clock cycle.

Verification Plan:

1. Write and Read to a particular memory location - Perform write to any memory location, read from the same memory location, read data should be same as written data.
2. Write and Read to all memory location - Perform write and read to all the memory locations (as address is 3bit width the possible address are 3‘b000, 3’b001, 3’b010 and 3’b011, 3‘b100, 3’b101, 3’b110 and 3’b111).
3. Default memory value check - Check default memory values. (before writing any locations, do read operation. You should get default values as ‘hFF).
4. Reset in Middle of Write/Read Operation - Assert reset in between write/read operations and check for default values. (after writing to few locations assert the reset and perform read operation, you should get default memory location value ‘hFF).

Verification Block Diagram Architecture



1. Design - First, we create a design that is a module in SV that does what we defined.

2. Interface - Defines all the signals that come in and go out. (Connectivity, Direction)

3. Transaction - A class in SV that contains all the data required for a specific operation.

4. Generator - It is a class, and its role is to generate transactions and pass them to the driver using the mailbox. It has no clock and operates at a high level. To generate a transaction, the generator creates a new one to make space in memory and then assigns random values to those locations. After that, we want to place the transactions in the mailbox. The mailbox exists in the general environment of the TB, so the generator has a pointer to the mailbox so that it recognizes the mailbox.
   In total, the generator needs to know:
   a.Transaction.
   b.Pointer.

5. Driver - Communicates with the generator (therefore, it also needs a pointer to the mailbox) and translates the transactions into signals. The driver is connected to the interface, which pushes the values to the design. In order to push the transaction to the interface, it needs the pointer to the interface (the pointer comes from the top). In

a complex design, there can be several interfaces. The driver is the only one that has a clock (the monitor also has one) and is the only one that operates at the design level, at the signal level. The rest of the environment operates at the transaction level (this is easier because you can do copy, compare, and display in UVM, which is built into the generator).

6. Environment - A class that includes the generator, driver, and mailbox. The env defines the mailbox and creates a new one, so now the pointer to the mailbox is not null. It creates a new instance of the generator class and gives it the pointer to the mailbox. Similarly, when it creates a new driver, it also provides the pointer to the mailbox.The driver also needs to receive the pointer to the interface, but the interface is not yet present because it resides at the top. So at this stage, we still don't know how to pass the pointer to the driver because the environment doesn't recognize the interface at all (the pointer will come from the top).The generator does a put, and the driver does a get, where the get is blocking. Therefore, as long as the mailbox is empty, the driver is stuck in get and waiting. As soon as the generator places the transaction, the driver's get is released, and immediately the transaction passes to the driver, then through the pointer of the interface (which we will receive from the top), it pushes to the interface. The design is connected to the interface, and the signals reach the design; it already outputs the results to the monitor and scoreboard.

7. Test - It is also a class; it is the only one that can be a program, but it is usually a class. Its role is to generate a specific test. In our environment, there can be many tests; for example, a test that only performs a reset or a read/write test from a memory array. There can also be three different tests, each of which will include the environment of the environment. In summary, the role of the test is to execute.

8. Top - A module that contains the entire verification environment, which includes the test, interface, and design. Once the top recognizes the interface, it can pass the pointer to the driver (it can also do this directly—since it is at a higher level, it can pass it to whoever it wants). The interface pointer, called a virtual interface (a reserved name), passes from the top to the test, from the test to the environment, and from the environment to the driver. When the driver has the pointer, as soon as it pushes the transaction, it will reach that specific interface. As already mentioned, each design has several interfaces, and each interface has its own driver.

9. Monitor In - Located on the interface, it listens to the signals that enter the interface and passes them to the scoreboard. It has its own mailbox and only transfers the fields related to inputs.

10. Monitor Out - Located on the interface, it transfers the signals that are the outputs of our design after receiving the inputs and returning the output.

11. Scoreboard - We build the reference model for our design within it. It receives the transaction from the monitor in, passes it to the reference model, and compares the output with the output of the monitor out.

Remarks in this design:

Transaction - In this transaction, I defined two constraints: one for enable and one for reset, because most of the time I want enable to be 1 and reset to be 0.

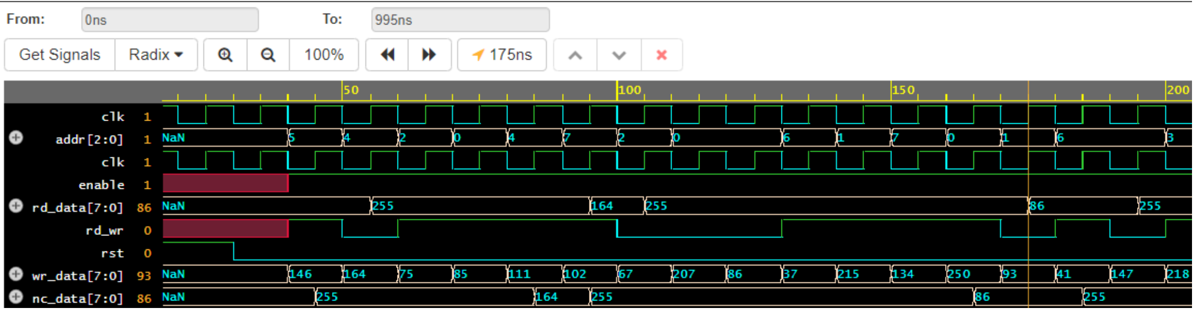
```
//in order to control the distribution of rst\enablbe, we use this method
constraint rst_con    {rst      dist {0 := 90 ,1 := 10};}
constraint enable_con {enable    dist {1 := 90 ,0 := 10};}
```

Driver - the driver pushes the signal on the negative edge - my definition.

Monitor in - The monitor_in samples the inputs from the scoreboard at the positive edge and only if 'enable' is high.

Monitor out - monitor_out samples the output of the interface after both the negative edge and the positive edge; this delay is created to ensure we capture the correct transaction each time we compare the reference model with the design.

Wave Diagram


From: 0ns
To: 995ns
Get Signals
Radix
100%
175ns
50
100
150
200
clk 1
addr[2:0] 1 NaN 5 4 2 0 4 7 2 0 6 1 7 0 1 6 3
clk 1
enable 1
rd_data[7:0] 86 NaN 255 164 255 86 255
rd_wr 0
rst 0
wr_data[7:0] 93 NaN 146 164 75 85 111 102 67 207 86 37 215 134 250 93 41 147 218
nc_data[7:0] 86 NaN 255 164 255 86 255

url For EDAPLAYGROUND with the written code:
https://www.edaplayground.com/x/TR2s