Cache Documentation

Authors: Nikodem Gazda, Luke Saleh, Brennan Borchert, Quinlan Stewart

Table of Contents

[Table of Contents 1](#_Toc181621844)

[General Implementation 3](#_Toc181621845)

[Cache Operations 3](#_Toc181621846)

[Read Hit 3](#_Toc181621847)

[Read Miss 3](#_Toc181621848)

[Write Hit 3](#_Toc181621849)

[Write Miss 3](#_Toc181621850)

[Results 3](#_Toc181621851)

[Individual Components 3](#_Toc181621852)

[Cache Block 3](#_Toc181621853)

[Functionality 3](#_Toc181621854)

[Results 3](#_Toc181621855)

[Sub-component: Cache Data 3](#_Toc181621856)

[Sub-component: Cache Tags 3](#_Toc181621857)

[Sub-component: Cache Valid Bits 3](#_Toc181621858)

[Sub-component: Cache Hit Logic 3](#_Toc181621859)

[Sub-component: Cache LRU Buffer 3](#_Toc181621860)

[Sub-component: Cache Control 4](#_Toc181621861)

[Ram Block 4](#_Toc181621862)

[Functionality 4](#_Toc181621863)

[Results 4](#_Toc181621864)

[Miscellaneous Logic 4](#_Toc181621865)

[Register 4](#_Toc181621866)

[Delay 4](#_Toc181621867)

# General Implementation

## Cache Operations

### Read Hit

### Read Miss

### Write Hit

### Write Miss

## Results

# Individual Components

## Cache Block

### Functionality

### Results

### Sub-component: Cache Data

#### Functionality

#### Results

### Sub-component: Cache Tags

#### Functionality

#### Results

### Sub-component: Cache Valid Bits

#### Functionality

#### Results

### Sub-component: Cache Hit Logic

#### Functionality

#### Results

### Sub-component: Cache LRU Buffer

#### Functionality

#### Results

### Sub-component: Cache Control

#### Functionality

#### Results

## Ram Block

### Functionality

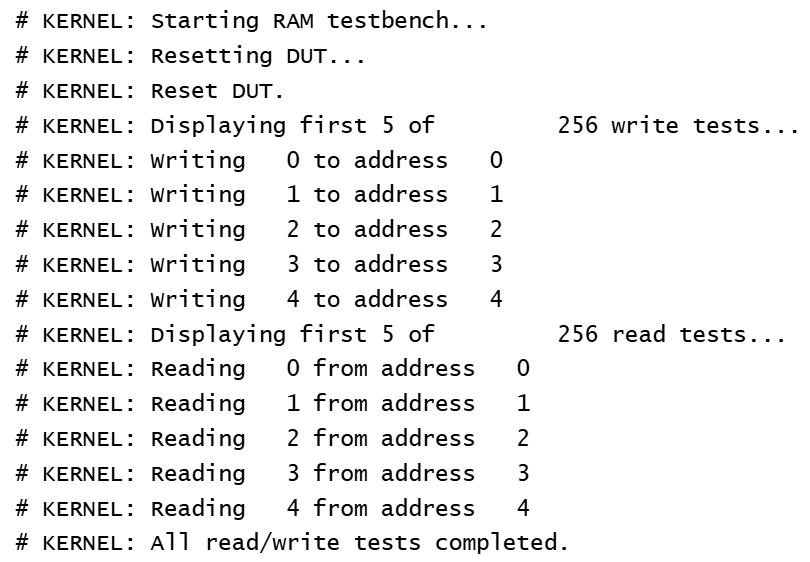
The RAM block is implemented as a DEPTH-long WIDTH-bit array of memory/register locations. The RAM takes in an address, data, and a write enable as inputs, and its only output is a WIDTH-bit data output. Both RAM reads and writes are synchronous, only updating the contents of the ram on a cycle where the write enable is asserted. Reads are only valid a cycle after write enable is not asserted.

### Results

The RAM testbench consists of a set of direct testcases and random testcases.

**Directed Testcases:**

In the directed testcases, we set the address and data input to the incrementing test number, set the write enable to 1, and wait for the rising edge of the clock (ex. Testcase 2 has addr=2, data\_in=2, we=1). We do this for every address in the RAM. The testbench itself has a reference model that mimics the behavior of the RAM in an array. After all addresses have been written, we then read from every address and compare with the data saved in the same addresses of the reference model. This testcase ensures that data that is written to RAM is maintained over longer periods of time, even as other addresses are written to and read from. See the figure below to see the first 5 values written to and read from the RAM.

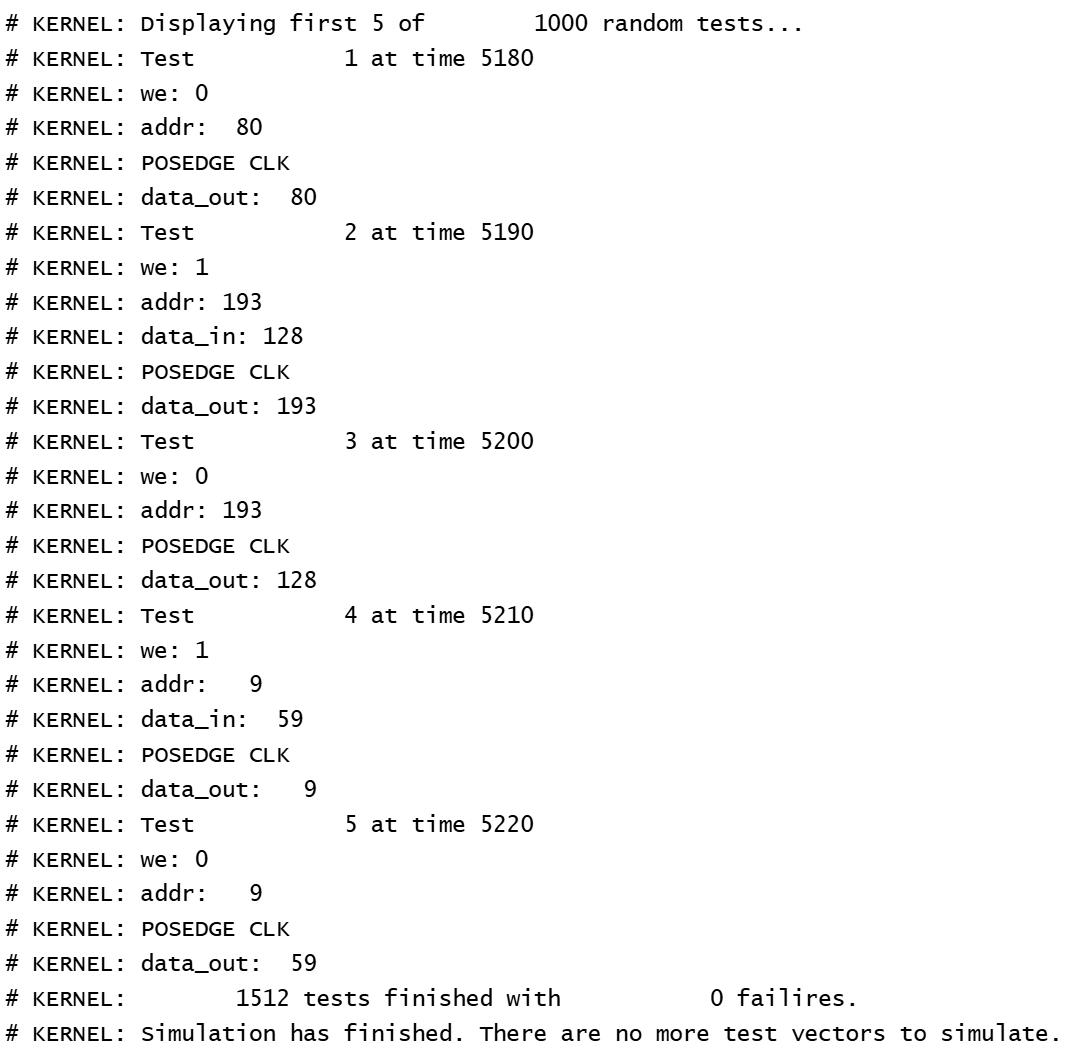


**Figure X:** Directed tests for the RAM.

**Random Testcases:**

We performed 1000 random test cases with the following constrained random verification:

* Data input is all 0s 10% of the time, all 1s 10% of the time, and the rest of the values the other 80% of the time.
* Address input follows the same constraints as the data inputs.
* The write enable is asserted 50% of the time for 50% reads and 50% writes.
* Writes are followed by reads which must use the same address used for the write so that each write to a random address is checked.



**Figure X:** First 5 random testcases for the RAM block.

In this testcase, we can see that a read from address 80 returns 80, which was the value written to that address in the directed testcase. The next operation is a write of 128 to address 193, and we can see in the next operation, which is a read from address 193, that the data out is 128, and so the data was saved correctly. The next 2 testcases exhibit the same behavior for address 9 with the data value 59.

**Assertions:**

* **out\_correct\_check:** On a read/when the write enable is de-asserted, data output is equal to the data found in the reference model using the same address.
* **write\_check:** When the write enable is asserted and followed by a read, and the address is the same for the read as for the write, then we wait a cycle for the read to happen and check that the input for the write was the data output for the read.

## Miscellaneous Logic

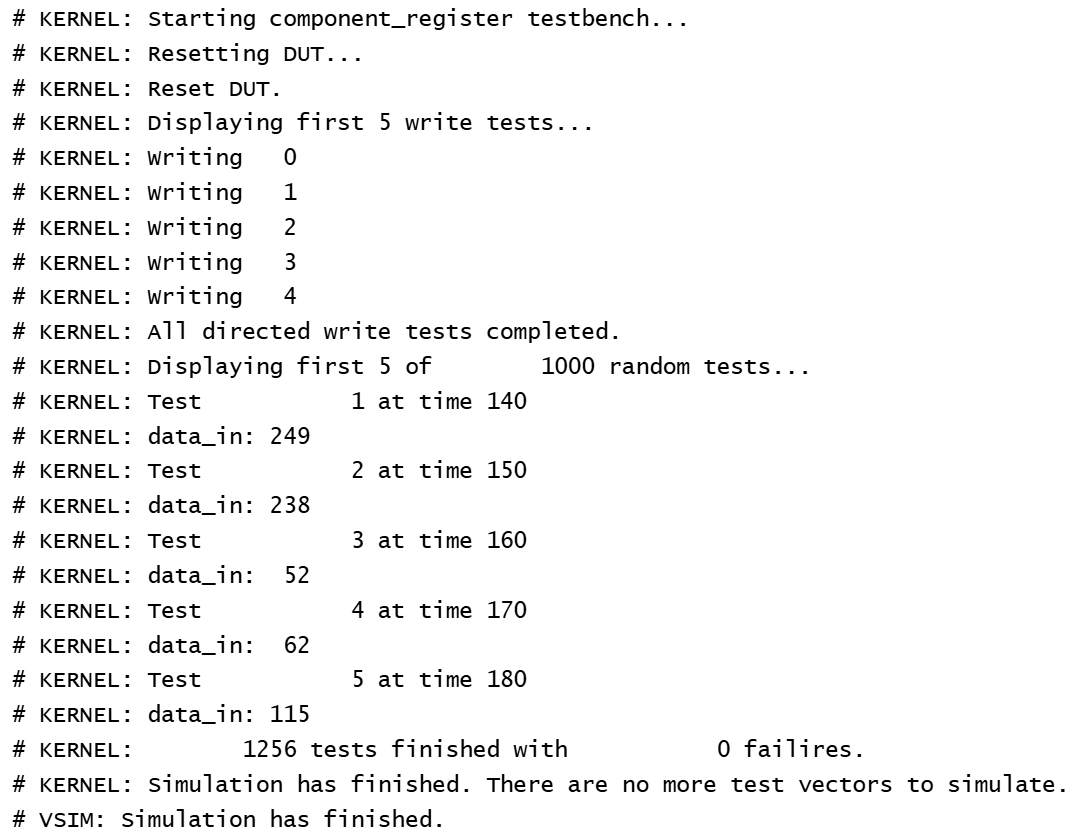
### Register

#### Functionality

The register entity is a standard register with a WIDTH-bit input and output with no enable.

#### Results

To test the register, we first did a set of directed tests where we assigned an 8-bit register each possible value from 0 to 255. See the figure below for the first 5 directed tests. We also did a set of 1000 randomized tests, with the data inputs being constrained to all 0s 10% of the time, all 1s 10% of the time, and all values in-between for the remaining 80% of the time.



**Figure X:** Testbench results for the register component.

**Assertions:**

The assertions for this testbench are just checking that on each clock, the data output from the register is the data that was on the input line in the previous cycle.

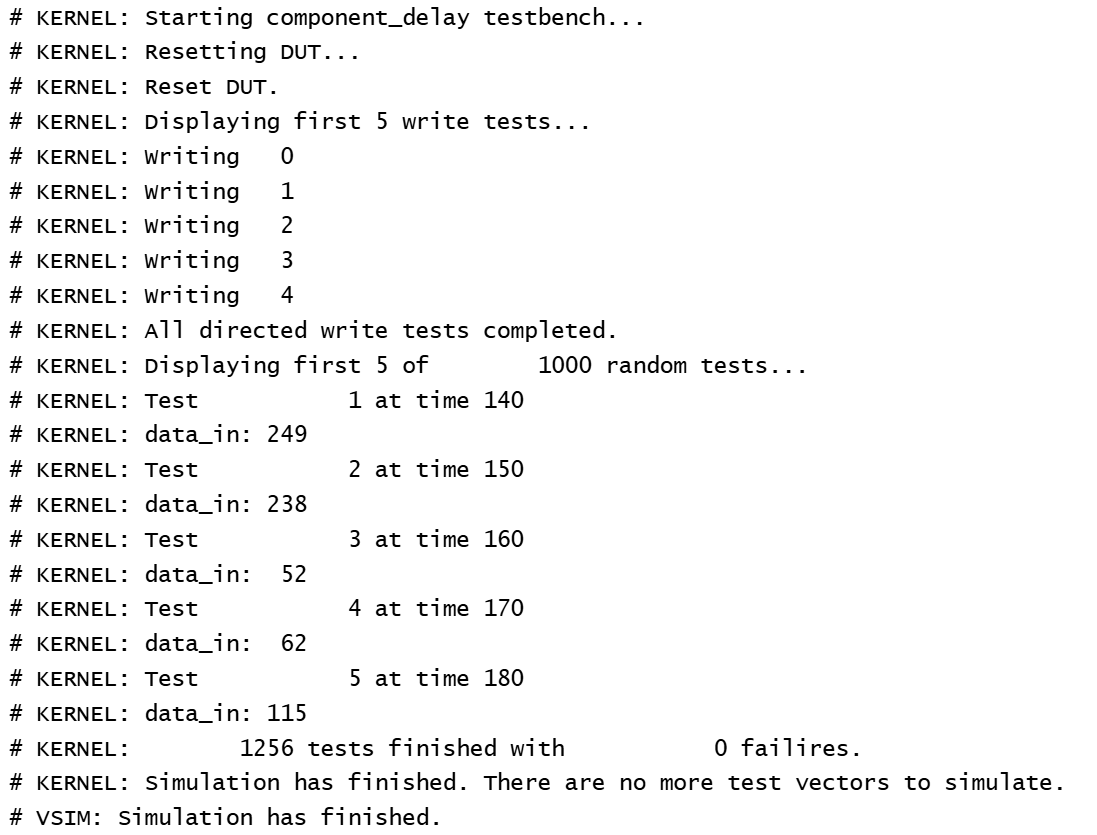
### Delay

#### Functionality

The delay entity is a CYCLE-long array of WIDTH-bit register components.

#### Results

The delay testbench is similar to the register testbench since the delay component is just a string of registers. We did the same directed tests from 0 to 255 and the same set of 1000 randomized tests, with the data inputs being constrained the same way.



**Figure X:** Testbench results for the delay component.

**Assertions:**

The assertions for this testbench are similar to the register testbench’s checking that on each clock, the data output from the register is the data that was on the input line in the previous number of cycles specified by the length of the delay, which is 4 cycles in this testbench.

# Conclusion

This was a fun and challenging project. During this project, we learned that it’s much easier to implement a design by first very clearly specifying the functionality and ONLY then implementing it in code. This is how it’s done in the industry, but we underplayed the complexity of the design and thought we could jump straight into the implementation. Since we realized we would have to go back and define the exact functionality, the design got confusing and we only started making progress once everything was set in stone.

This project was both a good practice of our understanding of caches/the algorithms that help them work and of hardware design/verification.