Final Project Cache Documentation

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

Table of Contents

[2 General Implementation 1](#_Toc183795757)

[2.1 Cache Operations 2](#_Toc183795758)

[2.1.1 Read Hit 2](#_Toc183795759)

[2.1.2 Read Miss 2](#_Toc183795760)

[2.1.3 Write Hit 3](#_Toc183795761)

[2.1.4 Write Miss 3](#_Toc183795762)

[2.2 Simulation Results 3](#_Toc183795763)

[3 Individual Components 5](#_Toc183795764)

[3.1 Cache Block 5](#_Toc183795765)

[3.1.1 Sub-component: Cache Data 5](#_Toc183795766)

[3.1.2 Sub-component: Cache Tags 8](#_Toc183795767)

[3.1.3 Sub-component: Cache Valid Bits 10](#_Toc183795768)

[3.1.4 Sub-component: Cache Hit Logic 12](#_Toc183795769)

[3.1.5 Sub-component: Cache LRU Buffer 15](#_Toc183795770)

[3.1.6 Sub-component: Cache Control 21](#_Toc183795771)

[3.2 Ram Block 30](#_Toc183795772)

[3.2.1 Functionality 30](#_Toc183795773)

[3.2.2 Simulation Results 30](#_Toc183795774)

[3.3 Miscellaneous Logic 31](#_Toc183795775)

[3.3.1 Register 31](#_Toc183795776)

[3.3.2 Delay 32](#_Toc183795777)

[4 Conclusion 33](#_Toc183795778)

# General Implementation

This cache/RAM has been implemented in SystemVerilog and features a RAM (of parameterized size) and an N-set associative cache, configurable at compile time, with the replacement strategy being last recently used, demonstrating a complete functional model. This design is tested with directed and randomized testbenches for each individual block and then the entire design as a whole to verify its functionality.

The Cache interfaces with both a user/uP and a RAM, where the user can only toggle the signals on the user/uP side. Below are the user-side I/O:

* Inputs:
  + Write Enable
  + Read Enable
  + Address
  + Data Input
* Outputs:
  + Operation-in-progress (for debugging purposes)
  + Done
  + Data output

And the RAM-side I/O:

* Outputs:
  + Data output to RAM
  + Address output to RAM
  + Write Enable output to RAM
* Input:
  + Data input from RAM

## Cache Operations

There are four operations the cache is capable of. Below is a brief description of each, the steps in the cache block that happen to achieve these operations, and the total number of cycles each operation requires. The first 4 steps are the same for each operation:

1. Begin operation if either read enable or write enable are high. Register the inputs (read enable, write enable, address, and data input). This step takes 1 cycle.
2. Once the inputs are registered, the Cache Tag and Cache Valid blocks output the values of each bank they have stored for the given index specified by the input address. This step is combinational and doesn’t add any cycles to the operation.
3. The Cache Hit block receives the target tag specified by the input address, the set of tags from the Cache Tag block, and the Cache Valid bits. It uses this information to determine whether this operation is a hit or miss. The Cache Hit block also outputs the relevant bank for this operation, which is used throughout the cache, but most notably to read from/update the Cache Data block. This step is combinational and doesn’t add any cycles to the operation.

Since the inputs only get registered at the start of an operation, and the hit/miss status and relevant bank information are combinational and based off those inputs, these combinational signals also only update once per operation.

### Read Hit

This operation happens when a read is requested and the item is currently in the cache with the following additional steps:

1. The data is output from the Cache Data block using the information obtained from the previous steps. Once valid, the data output is registered and output on the same cycle as when the done signal goes high. This step adds one more cycle to the operation.

Taking into account the registering of the inputs, the registering of the outputs, and the other combinational logic, this operation takes a total of 2 cycles to complete.

### Read Miss

This operation happens when a read is requested and the item is NOT currently in the cache and must update the cache with the value from RAM, with the following steps:

1. When a read miss is detected, we must receive the data from the RAM. The address line for the RAM is always tied to the registered address input for the cache, so we must wait 1 cycle for the data to arrive from the RAM. This steps adds 1 cycle to the operation’s total.
2. Once the data arrives on the cache’s “data\_from\_RAM” line, internal write enable logic in the Cache Control block signals that the Cache Data, Cache Tag, and Cache Valid blocks must be written to using the data from the RAM, which takes some additional MUX-ing. Reference the sections on the [cache control block](#_Sub-component:_Cache_Control) for more information. On the next cycle, the data will be written to these blocks.
3. This step is performed concurrently with the previous step. Since the process is marked as a read miss, when the data from the RAM arrives, it’s immediately muxed for the input of the data output register, and the data shows up on the next cycle when the done signal goes high. This step, performed concurrently with the previous step, add 1 total cycle to the operation.

Considering the registering of the inputs, the delay for the RAM to fetch the data, and the registering of the outputs, this operation takes a total of 3 cycles.

### Write Hit

This operation happens when a write is requested and the item is currently in the cache with the following steps:

1. After steps 1-3, the cache recognizes it’s performing a write hit operation. At this point, the inputs are registered and the write enable is immediately propagated to the Cache Data, Cache Tag, and Cache Valid blocks. The RAM also gets the write enable immediately propagated to it. This write will add one additional cycle to this operation
2. The outputs for write operations are 0s, which will update on the same cycle as once the write has completed to all cache and RAM blocks.

Considering the registering of the inputs and the write to all cache and RAM blocks, this operation takes a total of 2 cycles.

### Write Miss

This operation happens when a write is requested and the item is NOT currently in the cache and must update the cache as well as the RAM, with the following steps:

1. Since the cache and RAM are being written to regardless of whether there is a hit or miss, these operations actually both take the same amount of steps, the only difference being an artificial delay being added here so the miss operation has a penalty.

Considering the delays in the Write Hit operation and the additional artificial delay for the Write Miss, this operation takes a total of 3 cycles.

## Simulation Results

The Top-Level testbench consists of a set of direct testcases and random testcases. All simulations for this design are conducted using the following EDA playground:

Please note that the EDA playground environment is currently configured for the final, integrated testbench. If you’d wish to run the testbenches for the other blocks in the design for yourself, please contact us so we can reconfigure the environment with the appropriate include statements, however, the code, explanation, and runs for each testbench have been included in the submission.

*Directed Testcase 1:*

In the first directed testcase, we test for Write Miss and Read Miss operations by writing to every RAM address first and then reading from every address afterwards. Since each address access is separated by the reads and writes of all the other addresses, each operation will be a miss.

This is accomplished by incrementing the address and data input according to the test number being run (in test 0, address 0 has data 0; in test 1, address 1 has data 1, etc…). Once we set the inputs, we print them, wait for a rising edge, and print the outputs. We do this for 4 cycles so that after we see the done signal, we can see that the done signal goes low and all other registered inputs are cleared.

After the first cycle, we set the write enable back to 0 so that another operation doesn’t start in the middle of the current one. We could’ve made the cache so that it ignores any operation requests until the current operation has finished but felt this would create a confusing testbench.

This testcase ensures that data that is written is maintained over longer periods of time, even as other indices and banks are written to and read from. See the figure below to see the first 5 values written to and read from the cache.

**Figure X:** Directed testcase 1 for the cache

*Directed Testcase 2:*

In the second directed testcase, we test for Write Hit and Read Hit operations by writing to every RAM address twice and reading from the same address immediately afterwards. This is then repeated for each address. Since each address access is sequential, the second write and first read for each operation will be a hit.

This is accomplished by incrementing the address and data input according to the test number being run (address 0 has data 0, address 1 has data 1, etc…) for the first write. The second write increments the data input by 1 and the read reads from that address. This testcase has the same methodology of setting, displaying, and waiting for signals. This testcase moves onto the next operation after the done is detected, however, since additional no-operation cycles were already shown in the previous testcase.

This testcase ensures that data is written and read in the expected delay time. See the figure below to see the first 5 values written to and read from the cache.

**Figure X:** Directed testcase 2 for the cache.

*Directed Testcase 3:*

This testcase is just for fun. As with the first directed testcase, here, we write one character of a message to each address of the RAM. Once each character is written, we read back the data from each address, store it in a string in the testbench, and display the returned message to see if we get the same message back.

See the figure below to see both sent and received messages.

**Figure X:** Directed testcase 3 for the cache.

As you can see in the figure above, we are able to successfully write the first 256 characters of the general implementation of this documentation to the cache and successfully read it back exactly the same. This was a cool one to get working.

**Random Testcases:**

We performed 1000 random test cases with the following constrained random verification. There were less inputs that needed constrainment in the final testbench—for more complex constraints, see the other sub-components in this documentation:

* Both the data and address inputs are all 0s 10% of the time, all 1s 10% of the time, and the rest of the values the other 80% of the time.
* The write enable and read enable cannot both be high or low (XOR) at the same time. We don’t want them to be high at the same time since that’s not a valid operation. For them both to be low at the same time is valid, but we’re already testing that a lot in the random tests.

In the random testcases, we wait until we receive a done signal before performing the next operation. We use several reference models to:

* Store the byte corresponding to the input address on a write operation. Any read outputs from that address are then expected to match the values we have stored in the reference model.
* Track the tags currently being stored in the Cache Tag block so that we can know whether an operation will be a hit or miss and which banks are the least recently used.
* Track the last active operation and its relevant address.

**Figure X:** First 5 random testcases for the cache.

In these testcases, we can see that

**Assertions:**

* **out\_correct\_check:** This assertion checks that when the done signal goes high on a read, the data output is equal to the contents of the reference model.
* **out\_write\_check:** This assertion checks that the when the done signal goes high on a write, the data output is equal to 0.
* **out\_stable\_check:** This assertion ensures that outputs remain unchanged until a done signal goes high.
* **done\_pulse\_check:** This assertion checks that the done signal is only ever high for one cycle at a time, since each operation takes at least 2 cycles.
* **miss\_delay\_check:** This assertion ensures that operations we expect to be misses have a delay of 3 cycles.
* **hit\_delay\_check:** This assertion ensures that operations we expect to be hits have a delay of 2 cycles.
* **re\_we\_pulse\_checl:** This assertion just makes sure that we’re running the tests correctly and that we don’t have any read/write requests one cycle right after another.

Please see the code implementation for more detail. The assertions are at the bottom of each testbench.

# Individual Components

## Cache Block

### Sub-component: Cache Data

#### Functionality

The Cache Data block is the physical storage for data within the cache. The data is organized into multiple banks (configurable at compile time) and the input index is used as the cache address, whose size is the number of banks specified in the cache multiplied by the number of index addresses.

The inputs for this block are the write enable, the desired way to access, the cache index, the data input, and the only output is the data output. Reads from this block are combinational/instant while the writes sequential and take 1 cycle to update.

The memory locations in the Cache Data block must be indexed with both the desired way and the index. For example, if you would like to read from bank 2 in address 0x3, you must specify 2 as the desired way and 0x2 as the desired index.

#### Simulation Results

The Cache Data testbench consists of a set of direct testcases and random testcases.

**Directed Testcases:**

In the directed testcases, we increment the data value of each bank of each index, set the write enable to 1, and wait for the rising edge of the clock (ex. bank 1 of index 2 would have a value of index\*number of banks + current bank = 9). We do this for every index and bank in the Cache Data block. The testbench itself has a reference model that mimics the behavior of the Cache Data block in an array. After all addresses have been written, we then read from every index/bank pair and compare with the data saved in the same addresses of the reference model. This testcase ensures that data that is written to the Cache Data block is maintained over longer periods of time, even as other indices and banks are written to and read from. See the figure below to see the first 5 values written to and read from the Cache Data block.

A white screen with black text

Description automatically generated

**Figure X:** Directed tests for the Cache Data block.

**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.
* Index input follows the same constraints as the data inputs.
* Way input is kept random.
* 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 index and way used for the write so that each write to a random address is checked.

A screenshot of a computer program

Description automatically generated

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

In this testcase, we can see that a read from bank 1 of index 1 returns 5, which is the value that was written to that address in the directed testcase. The next operation is a write to bank 2 of index 1, and we can see the read from this index/bank pair in the next cycle return that value of 253. The remainder of the random testcases passed with no failures.

**Assertions:**

* **out\_correct\_check:** This assertion checks the data output is always equal to the contents of the reference model. This is an immediate assertion since reads from the cache data block are combinational/instant.
* **write\_check:** When the write enable is de-asserted, the write enable on the previous cycle was asserted, the index on this cycle matches the index from the previous cycle, and the way on this cycle matches the way from the previous cycle, then within the same cycle we expect that the data output is equal to the input from the previous cycle.

### Sub-component: Cache Tags

#### Functionality

The Cache Tags block is similar to the Cache Data block in that it acts as the physical storage for the tags in the cache, organized into banks and indices in the same way, with reads being combinational/instant and writes being synchronous as well.

The inputs for this block are the write enable, the desired way to access, the cache index, the target tag input, and instead of outputting the tag from just one bank, each bank from each index is output. This parallel output exists so it’s easier for the Cache Hit Logic block to determine whether there’s been a hit/miss. Reads from this block are combinational/instant while the writes sequential and take 1 cycle to update.

#### Simulation Results

The Cache Tag testbench consists of a set of direct testcases and random testcases and is very similar to the Cache Data testbench.

**Directed Testcases:**

The directed testcases for the Cache Tag block are the same, except that the printed outputs are each bank of the index. The following paragraph can be skipped if you’ve already read about the Cache Data testcase.

In the directed testcases, we increment the data value of each bank of each index, set the write enable to 1, and wait for the rising edge of the clock (ex. bank 1 of index 2 would have a value of index\*number of banks + current bank = 9). We do this for every index and bank in the Cache Tag block. The testbench itself has a reference model that mimics the behavior of the Cache Tag block in an array. After all addresses have been written, we then read from every index/bank pair and compare with the data saved in the same addresses of the reference model. This testcase ensures that data that is written to the Cache Tag block is maintained over longer periods of time, even as other indices and banks are written to and read from. See the figure below to see the first couple values written to and read from the Cache Tag block.

A screenshot of a computer program

Description automatically generated

**Figure X:** Directed tests for the Cache Tag block.

**Random Testcases:**

We performed 1000 random test cases with the following constrained random verification. If you’ve read the CRV for Cache Data, you can assume the same restrictions:

* Tag 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.
* Index input follows the same constraints as the data inputs.
* Way input is kept random.
* 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 index and way used for the write so that each write to a random address is checked.

A screenshot of a computer program

Description automatically generated

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

In this testcase, we can see a write of tag 118 to bank 1 of index 1. In the next testcase, we read from index 1 and see all the values stayed the same from the directed testcase except for bank 1, which has changed to the input tag of 118 from the previous write. The same happens in the next random write/read pair with a value of 61 in bank 2 of index 0. The remainder of the random testcases passed with no failures.

**Assertions:**

* **These assertions are the same as the ones from Data Cache, adjusted for the tags and the full index outputs.**
* **out\_correct\_check:** This assertion checks the tag output is always equal to the contents of the reference model. This is an immediate assertion since reads from the cache tag block are combinational/instant.
* **write\_check:** When the write enable is de-asserted, the write enable on the previous cycle was asserted, the index on this cycle matches the index from the previous cycle, and the way on this cycle matches the way from the previous cycle, then within the same cycle we expect that the tag output of the specified way is equal to the input from the previous cycle.

### Sub-component: Cache Valid Bits

#### Functionality

The Cache Valid Bits block is similar to the Data and Tag blocks in that reads are combinational while writes take one cycle, however, the data stored in this block is 1 bit for each bank for each index. When the write enable is asserted, a 1 is written to the specified bank/index each time, since a first time write to each location would cause that address to become valid, and replacing any address would cause it to become valid for the new address. The valid bits are only cleared when reset is asserted.

The inputs for this block are the write enable, the desired way to access, the cache index, and the valid bit from each bank of the specified index is the output. This parallel output exists so it’s easier for the Cache Hit Logic block to determine whether there’s been a hit/miss.

#### Simulation Results

The Cache Valid testbench consists of a set of direct testcases and random testcases and is very similar to the Cache Tag testbench.

**Directed Testcases:**

The directed testcases for the Cache Valid block are the same except that each bank output has a width of only 1.

In the directed testcases, set the write enable to 1 and wait for the rising edge of the clock for every index and bank in the Cache Valid block. The testbench itself has a reference model that mimics the behavior of the Cache Valid block in an array. After all addresses have been written, we then read from every index/bank pair and compare with the data saved in the same addresses of the reference model. This testcase ensures that data that is written to the Cache Valid block is maintained over longer periods of time, even as other indices and banks are written to and read from. See the figure below to see the first couple values written to and read from the Cache Valid block.

A screenshot of a computer program

Description automatically generated

**Figure X:** Directed tests for the Cache Valid block.

**Random Testcases:**

We performed 1000 random test cases with the following constrained random verification. These slightly differ from the Cache Tag in that the reset signal is randomized as well.

* Index 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.
* Way input is kept random.
* The write enable is asserted 50% of the time for 50% reads and 50% writes.
* The reset signal is asserted 5% of the time so that the block’s contents are cleared occasionally.
* Writes are followed by reads which must use the same index and way used for the write so that each write to a random address is checked.

A screenshot of a computer program

Description automatically generated

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

These first 5 testcases aren’t very telling since only a 1 can be written in each entry, but it does let us know that 1 is retained in the block for each index referenced. At points in the testbench, reset is asserted as well, clearing the contents of the cache and allowing for more variability.

**Assertions:**

* **out\_correct\_check:** This assertion checks the valid bit output is always equal to the contents of the reference model on all falling edges

### Sub-component: Cache Hit Logic

#### Functionality

The Cache Hit Logic block handles the logic for determining whether the incoming address is in the cache, whether it be for a read or write. This block is completely combinational.

The inputs are the target way, the full output of all the tags from the Cache Tag block, the full output of all the valid bits from the Cache Valid block, and the output of the LRU buffer in case of a miss.

The output is a bit labeled hit; when asserted, the Cache Hit Logic block is reporting a hit, and when not asserted, the logic block is reporting a miss. When there is a hit, the block also outputs the bank that corresponds to the matched tag/valid bit. When there is a miss, the block outputs the bank the LRU reports as the least recently used bank for the current index.

#### Simulation Results

The Cache Hit Logic testbench consists of a set of direct testcases and random testcases.

**Directed Testcases:**

There are 6 unique directed testcases for the Cache Hit Logic block, each repeated 5 times in our testing:

**Test 1:** all tags are 0, 1, 2, 3, target tag is 0, valid bits are 0, lru\_way is 0

This is a test case where all inputs are 0 which should return a **miss**, since all the valid bits are 0, and the output **chosen way should be 0**, since the lru\_way is 0.

**Test 2:** all tags are 0, 1, 2, 3, target tag is 0, valid bits are 0, lru way is 1

This is the same test as before with the only change being in the lru\_way from 0🡪1. This should return a **miss** as well and the output **chosen way should be 1** to reflect that the output chosen way reflects the LRU way on a miss.

**Test 3:** all tags are 0, 1, 2, 3, targt tag is 0, valid bits are 1, lru way is 2

This test should return a **hit**, since all the valid bits are asserted and there exists a tag that matches the target tag, and the **output chosen way should be 0**, since that’s the location of the tag that matches. Note that the output chosen way should not reflect the LRU way.

**Test 4:** all tags are 0, 1, 2, 3, targt tag is 1, valid bits are 1, lru way is 3

This test is the same as the previous with the only changes being in the target\_tag: 0🡪1, and the lru\_way: 2🡪3. This test should return a **hit** and the **output chosen way should be 1**. The output chosen way should not reflect the LRU way in this test case either.

**Test 5:** all inputs are 0s

This testcase is similar to Test 1 except that all input tags are 0. This case should only ever be true in the beginning of operation when no cache entries have been written to yet, and it should return a **miss** and the **output chosen way should be 0**, reflecting the LRU way.

**Test 6:** all inputs are 1s

This scenario should never happen in the actual use of the cache since the tags should never be the same in the banks of one index. Either way, we should expect a return of a **hit** and the output **chosen way should be 3**, since the Cache Hit Logic block checks each tag incrementally.

All expected outputs match the true outputs, as seen below:

A screenshot of a computer program

Description automatically generated

Figure X: Directed tests for Cache Hit Logic block.

**Random Testcases:**

We performed 100 random testcases (timed out with any more, limitation of EDA playground) with the following constrained random verification:

* **target\_tag\_ranges, lru\_way\_ranges, tags\_ranges:** make sure target tag, lru way, and tags are ‘0 10% of the time, ‘1 10% of the time, and the rest of the range 80% of the time. Since the tag input is an array, each tag is assigned independently of the others.
* **tags\_never\_same:** This constraint prevents any of the tags from being equal to the others. In the operation of the cache, the tag inputs should never be the same unless in the first couple reads/writes where the valid bits are low, since a read/write to an address with the same tag would only alter that address. This edge case is covered in the multiple iterations of the directed testcases, therefore, we prevent this behavior here.
* **target\_tag\_match:** We want to test an even split of hits and misses, therefore, we need to ensure that one of the tags matches the target tag 50% of the time. To do this, we must randomize three integers before any of the DUT signals; one to get the 50/50 chance of deciding to force a match between one of the tags and the target tag, the second to randomly choose which tag to force, and the third to get a 50% chance of that valid bit being asserted so we have an even number of hits and misses when the tags match. These random variables are solved before the DUT’s signals.

A screenshot of a computer code

Description automatically generated A screenshot of a computer program

Description automatically generated

**Figure X:** Randomized tests for Cache Hit Logic block.

As you can see in the randomized tests, in Test 1, no tag matches the target tag, therefore the output is a miss and the chosen way matches the lru\_way. In test 2, bank 3 matches the target tag, and the corresponding valid bit is high, meaning the output is a hit and the chosen way is bank 3. Test 3 has the same behavior as test 1. Test 4 shows a tag match with bank 0, but the corresponding valid bit is 0, resulting in a miss with the chosen way reflecting the lru\_way. Test 5 follows the same behavior as with tests 1 and 3.

The bottom output reports the number of tests, failures, and tag matches. This is meant to show that the CRV is constraining the inputs how we expect it to, with about a 50% chance of tags matching the target tag.

**Assertions:**

* **Miss\_check:** If a miss is reported, the chosen way must match the lru way.
* **Hit\_check:** If a hit is reported, the tag corresponding to the chosen way must match the target tag.
* **Tags\_match:** If any of the tags match the target tag, there should be a hit reported. This is kinda a duplicate of the hit\_check but I just wanted to make sure.
* **No\_tags\_match:** When no tags match the target tag, we expect a miss.

### Sub-component: Cache LRU Buffer

The Cache LRU Buffer stores the pointers to the least recently used banks for each index. The reads from the buffer are combinational and the buffer updates are synchronous, taking 1 cycle to update. The buffer considers any read or write to the cache as an update to itself.

The inputs for the LRU buffer are the read enable, the write enable, the chosen way that’s output from the cache hit logic, and the index input into the cache. The only output is the replacement way that’s used as an input into the hit logic block whenever a miss is detected.

#### Functionality

The LRU Buffer has a slot for each bank for each index which points to the least recently used bank. The leftmost slot in the buffer always represents the least recently used bank/way, last\_used[0][index], and the rightmost slot always represents the most recently used bank/way, last\_used[NUMBER\_OF\_BANKS][index]. When an update to the buffer is issued, the last slot in the buffer is assigned the chosen way on the input, indicating it is now the most recently used, and the rest of the slots in the buffer update accordingly. Below is a figure demonstrating this behavior:

A blackboard with white text

Description automatically generated

**Figure X:** How the LRU Buffer works.

In this diagram, Bank 0 stays in the least recently used position. The chosen way moves into the slot of the most recently used position, and the rest of the banks move up the chain.

The LRU Buffer has a combinational portion and a synchronous portion:

In the combinational portion, we aim to find the slot in the buffer that corresponds to the chosen way input into the buffer. Each slot in the buffer for the current index is compared to the input way and the matching slot is assigned to a signal called current\_way\_addr, representing the slot we want to use to update the buffer. This can be seen in 1. and 2. of Figure X.

In the synchronous portion, if the buffer is to be updated on a read/write, we loop through each slot starting at the current\_way\_addr and ending one slot before the last slot (most recently used slot). Then, the last slot representing the most recently used bank is assigned the value in the slot of the current chosen way. This can be seen in 3. and 4. of Figure X.  
Since the least recently used bank is always held in the leftmost slot, it can be treated as asynchronous/combinational and always be output.

#### Simulation Results

The Cache LRU Buffer testbench consists of a set of direct testcases and random testcases. We use a reference model and probes into the design to verify the LRU buffer. The reference model is an array of integers with as many elements as there are in the buffer for each index. Each element serves as a counter, and on each buffer update, the corresponding slot is update with the simulation time. Whenever we want to check for the correct output, we can check the least recently used bank in the DUT matches the timer with the simulation time earliest in the simulation.

This method was not used in the design since timers can overflow and having a timer for each bank of each index would not be an efficient use of resources in the design.

**Sequential Directed Testcases:**

In addition to the directed testcases below, we also added sequential testcases which loop through each bank of each index and write. This effectively rotates the LRU buffer, as you can see below.

In the following figure, we are first shown the state of the buffer when the directed tests start. Then, we are shown the signal values (both inputs and outputs) as well as the state of the LRU buffer after the clock edge had passed for the first sequential test. This then repeats for each sequential test.

In the first sequential test, we specify a write to way 0, which is currently in the least recently used bank slot, and so at time 70 it gets rotated to the end of the LRU buffer and the rest of the slots are filled in order. The next test specifies way 1, which is now again in the least recently used bank slot, so on the next cycle it also gets put in the last slot and the rest of the banks move up one. This continues for each bank of each index.

**A screenshot of a computer program

Description automatically generated**

**Figure X:** Sequential directed tests for Cache LRU Buffer.

**Directed Testcases:**

There are 2 unique directed testcases for the Cache LRU Buffer, each repeated individually 10 times in our testing:

**Test 1:** all inputs are low (read/write enable, way, index)

Repetitions of this testcase should continually output 0 as the replacement way/bank.

**Test 2:** all inputs are high

Repetitions of this testcase should continually output 0 as the replacement way/bank as well since bank 3 (2b’11 in binary) will continually be the most recently used bank.

All expected outputs match the true outputs, as seen below:

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

Figure X: Directed tests for Cache LRU Buffer.

**Random Testcases:**

We performed 1000 total randomized testcases with different constraints on their signals:

Test 1 (100 repetitions): Neither read nor write enable are ever asserted to test that the LRU buffer should only ever update when the enables are asserted. The way and index inputs are randomized according to the global constraints listed below.

Test 2 (100 repetitions): Read enable is asserted every cycle, write enable is never asserted. The way and index inputs are randomized as in test 1.

Test 3 (100 repetitions): Same as test 2 but replace read enable with write enable.

Test 4 (700 repetitions): All inputs are randomized with the constraint that the or of read enable and write enable is high only 50% of the time to keep the frequency of updates and non-updates even.

**Global constraints:**

* **way\_ranges** and **index\_ranges:** make sure way and index are ‘0 10% of the time, ‘1 10% of the time, and the rest of the range 80% of the time.

A screenshot of a computer program

Description automatically generated

A close-up of black text

Description automatically generated

**Figure X:** Randomized tests for Cache LRU Buffer.

As you can see in the randomized tests, in Test 1, both RE and WE are 0, so the LRU buffer for index 3 stays 0, 1, 2, 3. In Test 2, way 1 for index 1 is chosen and WE is high, therefore way 1 is put at the end of the LRU buffer and the remaining banks move up one slot. Tests 3 and 4 see the same behavior but for index 2 and with different combinations of RE and WE. In Test 4 repetiton 2, the previous buffer state for index 1 was 0, 2, 3, 1, and this test calls for way 2 of index 1 to be used, so way 2 gets sent to the end of the buffer and the rest of the banks update accordingly for a buffer state of: 0, 3, 1, 2.

The bottom output reports the number of tests, failures, and randomization failures. This is meant to show that the CRV is constraining the inputs how we expect it to and the test is passing with 0 errors.

**Assertions:**

* **LRU\_check:** This assertion makes sure that the least recently used slot points to the same bank as the timer mentioned above for all indices concurrently.
* **No\_update:** This assertion checks the LRU buffer for each index does not update when read enable and write enable are not asserted

A screen shot of a computer program

Description automatically generated

**Figure X:** Assertions for LRU Buffer

### Sub-component: Cache Control

The Cache Control Block handles all miscellaneous logic and timing required for the different sub-components of the cache to work together correctly. Specifically, this block handles:

* Internal write enable logic
* Done signal logic
* Which data source is muxed for the Cache Data block (RAM or uP)
* Which data gets output from the cache to the uP

#### Functionality

**Write Enable Logic:**

The write enable logic is synchronous and nested within its own block. It takes the write enable, read enable, and hit signals as inputs, and outputs the write enable for the Cache Data block. Below is a figure which shows the logic:

A diagram of a flowchart

Description automatically generated

Figure X: Logic diagram for Write Enable.

This logic makes sure that when we want to write to the Cache Data block we can do so, but also so we only do so once in every operation.

Regardless of whether there is a hit or miss, when the uP sends a write enable, we want it to get immediately sent to the Cache Data block. However, a hit operation takes a cycle and the inputs stay the same when the done signal is high, so we want to make sure the cache\_we is not high after that cycle delay, which is why we mux it with the output of a register which stores the write enable’s previous value. A miss operation takes one extra cycle, so we add another register and OR their outputs to make sure write enable is high for only one cycle.

When the uP sends a read enable and it’s a miss, we must wait a cycle for the data to be retrieved from the RAM, and then we write that data to the Cache Data block. Since we only want to write once, we check that if the previous output from that register was high, we don’t write again by muxing a 0 in.

We don’t write to the cache on a read hit.

**Done Signal Logic:**

There are 4 operations the cache is capable of: Read Hit, Read Miss, Write Hit, and Write Miss, as explained in the [General Implementation](#_Cache_Operations) section. These each have a latency of 1, 2, 1, and 2 cycles, respectively (in the context of the cache control block. In the context of the cache as a whole, add 1 to each of those since the inputs to the cache control block are registered outside of it). The Done Logic reflects these cycle delays according to the logic below:

A diagram of a diagram

Description automatically generated

**Figure X:** Logic diagram for Done Logic.

Since the inputs to the cache control block are registered, they stay the same for the duration of the operation. That means that in a normal done signal delay chain, another 1 would be propagated on each cycle of the operation, and we only want done to pulse for one cycle when the operation has completed. Therefore, each chain has some added logic which checks if any of the registers already has a 1 inside. If so, the input into the chain is muxed to be 0 instead of the input.

*EDIT: Actually, a bug found in the top\_level testbench found that sometimes when a miss operation is underway, the hit signal will update once the data values update and the done signal will start propagating for the hit operation, so this code was updated so that if and of the delay chains have a 1, all will receive a 0 on the next cycle.*

There also exists a signal called pre\_done which is used in the Data Output Logic before. This signal is just the done signal but a cycle before.

There also exists an output op\_in\_progress which is asserted while there is an operation in progress. This signal is high as long as any of the registers in the chains, including the input to the chains, are high.

**Cache Data Block Logic:**

This logic is just a mux that switches between the data input from the uP (when WE high) or from the RAM (when WE low) as the data input to the Cache Data block.

**Data Output Logic:**

This logic determines what data is put on the data output port of the cache and it is registered. Independent of the operation currently active, once the done signal goes high, the output updates and stays updated until done goes high again for the next operation. For read hits, the output reflects the data being output from the Cache Data block (using the chosen way). For read misses, the data comes from the RAM. For writes, the data is ‘0.

#### Simulation Results

The Cache Control block testbench consists of a set of 2000 hybrid directed/random testcases with the following constraints and directions:

**Directed Portion:**

The reason we say this testcase is a hybrid directed/random testcase is because some inputs are always explicitly assigned while others are always randomized.

Each testcase has two parts: the read and the write.

Reads are characterized by the read enable being high, write enable being low, and hit being randomized.

When hit is detected as high, this is then followed by a cycle where the read enable stays high and other inputs don’t change (other than the data\_from\_RAM because most inputs from the uP are registered outside the Cache Control block, and RAM inputs don’t need to be, especially because that would cause extra latency), and then it’s followed by one more cycle where read enable is 0 and the other inputs (except for data\_from\_RAM) stay the same.

When miss is detected, this is then followed by 2 cycles where read enable is low and all other inputs are the same except for data\_from\_RAM.

Writes have the same behaving testcases, flipping the behavior of the write enable and read enable.

**Constraints:**

* **Data\_in\_ranges, data\_from\_cache\_ranges, data\_from\_RAM\_ranges:** These inputs have the following distribution independent of each other: ‘0 10% of the time, ‘1 10% of the time, the rest of the range 80% of the time.
* **hit\_we\_re:** Hit signal will be high 50% of the time when we || re == 1. Hit can be either low or high even when neither write nor read enable are high since the Cache Hit block is fully combinational.
* **Reminder:** Read and write enable are controlled directly, so they are not randomized. Also, the chosen banks are left to be completely random since they are one-hot encoded.

A screenshot of a computer program

Description automatically generated

**Figure X:** Read hit randomized test for Cache Control block.

In this testcase we observe a read hit. Each block headed by “Time” is one cycle, and the outputs update in the following cycle. There are four things to check for: cache\_we logic, done logic, cache\_data\_in logic, and data\_out logic.

Since this is not a write enable and since this is not a miss, the write enable to the Cache Data block is not asserted for the duration of the test case.

Since this is a read hit, we expect the done signal to take 1 cycle before going high, and we in fact see that in the second block. You can also see the op\_in\_progress output is high at the beginning of the transaction and goes low after done goes low. Note that done pulses only for one cycle.

Since the write enable is low, cache\_data\_in is the data\_from\_RAM, even thought the cache\_we is not asserted, which is the desired behavior. You can see this value change as data\_from\_RAM changes every cycle.

Data\_out doesn’t change until done is asserted and doesn’t change afterwards, either. This is a read hit, so we expect the data to come from data\_from\_cache[chosen\_way], and so it does.

A screenshot of a computer program

Description automatically generated

**Figure X:** Write miss randomized test for Cache Control block.

In this testcase we observe a write miss.

Since this is a write operation, cache\_we is high immediately. Then, it is low for the rest of the operation since we only want to write to cache once.

Since this is a miss, the done signal takes two cycles (third/last block at the bottom) before pulsing for one cycle and op\_in\_progress is high for the entire time.

Since this is a write operation, cache\_data\_in immediately reflects data\_in, which happens to be 0.

Since this is a write operation, the data output is 0 once done is asserted. Note that before changing, data\_out retains the signal from the previous testcase: 254.

A screenshot of a computer program

Description automatically generated

**Figure X:** Read miss randomized test for Cache Control block.

In this testcase we observe a read miss.

Since this is a read miss, the cache should write the data from the RAM once it takes the RAM a cycle to load the data. Then, the cache\_we should go low since we only want to write to the Cache Data block once.

Since this is a read miss, the done signal is asserted 2 cycles after the start of the operation and op\_in\_progress is asserted for the duration of the operation.

Since this is not a write, cache\_data\_in will reflect data\_in, 146 in this case.

Since this is a read miss, data\_out should be the data\_from\_RAM, and it is 146 as well, which is good since we want the same value output as the one written to cache.

A screenshot of a computer program

Description automatically generated

**Figure X:** Write hit randomized test for Cache Control block.

In this testcase we observe a write hit.

As expected with a write, cache\_we is high at the beginning and during no other cycle.

As expected with a hit, the done signal is high after the first cycle and the op\_in\_progress is high for the duration of the operation, and then low afterwards.

As expected with a write, cache\_data\_in is data\_in.

Since this is a write hit, the data output is 0 when done is asserted but retains the value from the previous test, 146, up until that point.

You can also see at the bottom that all 2000 tests finished with 0 failures, checked by the 13 assertions described below:

**Assertions:**

*Write enable logc:*

* **we\_logic\_high\_check:** If the cache\_we signal is high, it must be for the following reasons:
  + There was a read miss 1 cycle prior, OR
  + There is a write hit in the current cycle AND NOT 1 cycle prior, OR
  + There is a write miss in the current cycle AND NOT 1 cycle prior AND NOT 2 cycles prior either.
* **We\_logic\_low\_check:** If cache\_we is low, none of the conditions from the assertion above must be true.

*Done logc:*

* **Done\_re\_hit:** If there is a read hit in the current cycle and this is the first cycle of the operation (op\_in\_progress was low the cycle before), then in the next cycle done must be asserted.
* **Done\_we\_hit:** If there is a write hit in the current cycle and this is the first cycle of the operation (op\_in\_progress was low the cycle before), then in the next cycle done must be asserted.
* **Done\_re\_miss:** If there is a read miss in the current cycle and this is the first cycle of the operation (op\_in\_progress was low the cycle before), then in the next 2 cycles done must be asserted.
* **Done\_we\_miss:** If there is a write miss in the current cycle and this is the first cycle of the operation (op\_in\_progress was low the cycle before), then in the next 2 cycles done must be asserted.
* **Done\_low:** If done is low in the current cycle that means all of the following sequences MUST be false:
  + There was a read hit in the cycle prior and it was the first cycle of the operation.
  + There was a write hit in the cycle prior and it was the first cycle of the operation.
  + There was a read miss in the 2 cycles prior and it was the first cycle of the operation.
  + There was a write miss in the 2 cycles prior and it was the first cycle of the operation.

*Cache Data logc:*

* **Cache\_data\_in\_check:** This assertion is immediate/combinational. If the write enable is high, cache\_data\_in must equal data\_in. Otherwise, cache\_data\_in must equal data\_from\_RAM.

*Data out logc:*

* **Data\_out\_re\_hit:** If done is currently asserted and there was a read hit in the previous cycle, data\_out is data\_from\_cache[chosen\_way] from the prior cycle.
* **Data\_out\_we\_miss:** If done is currently asserted and there was a write hit in the previous cycle, data out is 0.
* **Data\_out\_re\_miss:** If done is currently asserted and there was a read hit in the previous 2 cycles, data\_out is data\_from\_RAM from the previous cycle.
* **Data\_out\_we\_miss:** If done is currently asserted and there was a write miss in the previous 2 cycles, data out is 0.
* **data\_out\_unchanged**: Data\_out has not changed since the previous cycle unless done is now asserted.

## Ram Block

### Functionality

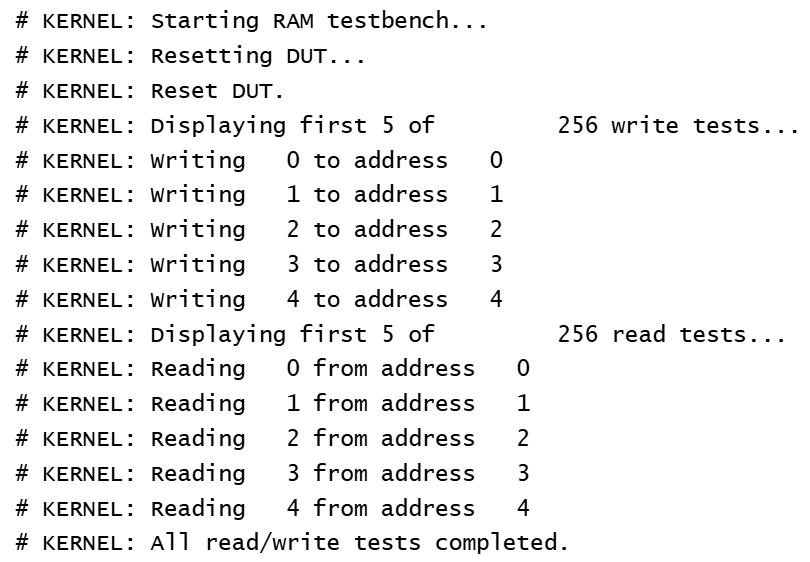
The RAM block is implemented as a DEPTH-long WIDTH-bit array of memory/register locations. The RAM takes in an address, data, and a write enable as inputs, and its only output is a WIDTH-bit data output. Both RAM reads and writes are synchronous, only updating the contents of the ram on a cycle where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

### Simulation 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.

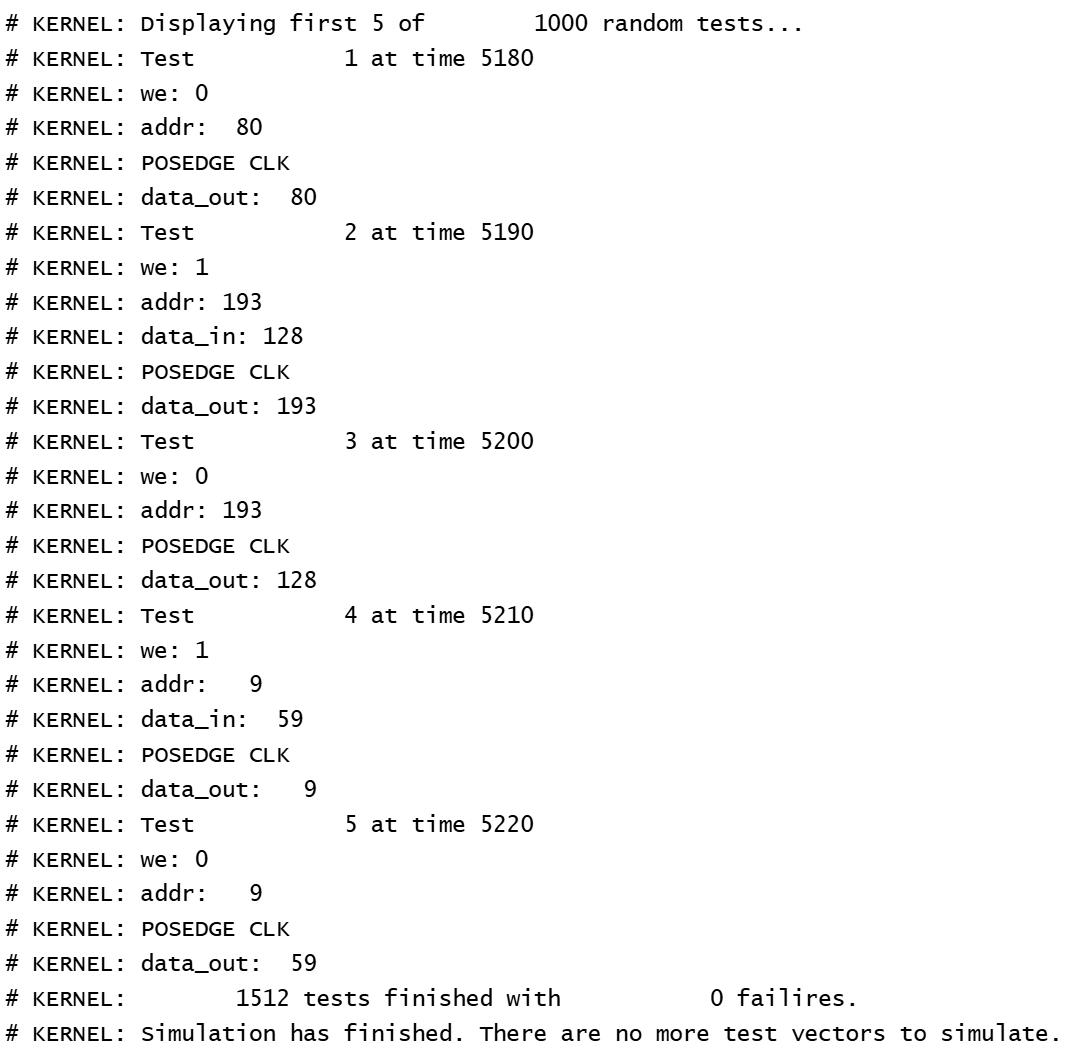


**Figure X:** Directed tests for the RAM.

**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

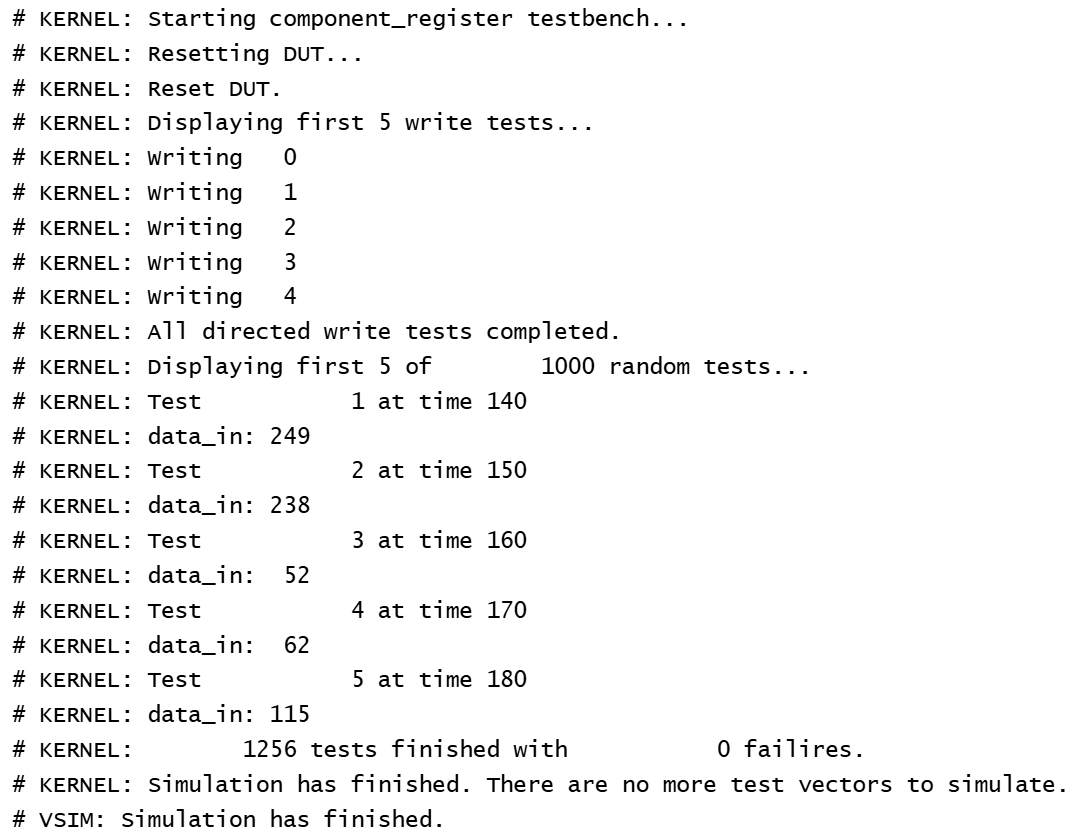
### Register

#### Functionality

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

#### Simulation 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.

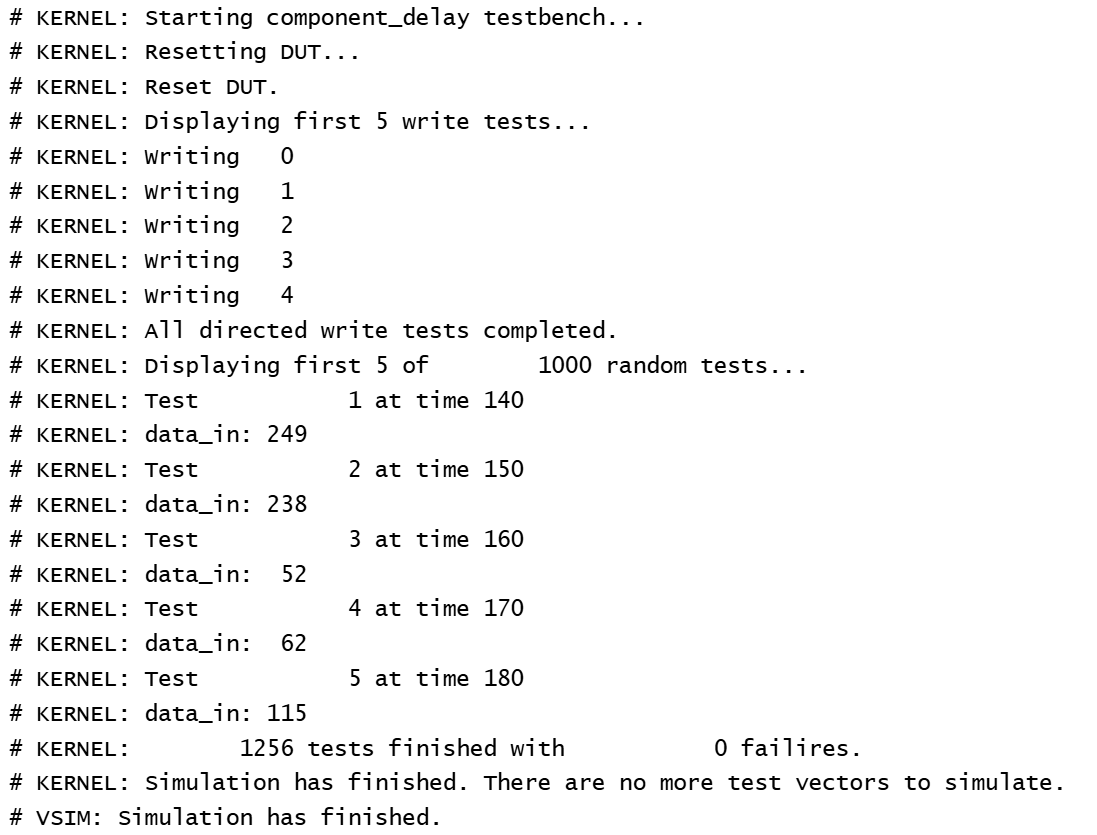
### Delay

#### Functionality

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

#### Simulation 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 an extremely fun and challenging 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. Because of this, we had to go back and change the design many, many times in the middle of verification. The design got confusing and we only started making progress once everything was set in stone.

Future work would include reconstructing the system to allow for pipelining. This would not only improve the efficiency/throughput but also likely simplify the logic so that operations don’t overlap. Future work would also include reconstructing the design as an FSM rather than a set of blocks of logic.

This project was both a good practice of our understanding of caches/the algorithms that help them work and of hardware design/verification. It feels like we used every single SystemVerilog concept under the sun out of pure necessity (not just to look pretty), considering we wrote ~3k lines of testbench code, ~1k lines of design code, and ~4k lines of code total.