

Lab 7 Workshop: Debugging AES with Testbenches

Why This Workshop

The testbench that comes with Lab 7, `testbench_aes_core`, tells you exactly one thing: whether 128 bits of cyphertext are right after ten rounds of encryption. When it fails you get a message like this, and nothing else:

```
Error: cyphertext = 52dd86f39c93b91e8f8daafe20776c9b,  
      expected 3925841d02dc09fdbc118597196a0b32
```

Every one of those bytes is wrong, and nothing in that line says whether the problem is in SubBytes, ShiftRows, MixColumns, the key expansion, or the FSM sequencing them. But FIPS-197 gives you much more than the final answer. Appendix B lists **every intermediate value**: the state at the start of each round, after each step, and every round key. The Lab 7 hints describe how to use that:

When debugging, find the first place you can tell a signal is wrong. Add all the relevant inputs that influence that signal. If one of them is wrong, recursively work backward. When the inputs are good and the output is bad, you've isolated the bug.

Today you will practice that loop on small pieces of AES, so that it is a habit by the time you're debugging your whole `aes_core`.

Learning Objectives

By the end of this workshop you will be able to...

- Convert between the 4×4 state grids in FIPS-197 and the 128-bit packing used by `aes_core`, first by hand and then with modules you can reuse in the rest of Lab 7.
- Write a self-checking testbench for one AES step using test vectors from the specification.
- Choose test vectors that make a bug easy to find, and recognize when a passing test proves very little.
- Localize a bug systematically: read the shape of the error, find the first wrong internal signal, and work backward from there.

Setup

- **Work in pairs.** One laptop per pair is enough. Swap who is typing at each exercise.
- **Open FIPS-197 to Appendix B** (page 34 of the standard, page 42 of the [PDF](#)). You will be reading test vectors out of it all class.
- **Simulator.** Each exercise below has a link that opens its files in a fresh [Jacquard FPGA Studio](#) project (sign in with your [g.hmc.edu](#) account). If you would rather use QuestaSim, download the [workshop folder](#) and add the files for each exercise to a new project. The testbenches work in both.

i Keep your work

Jacquard stores files in your browser. If you want to keep what you write today, use **Download .zip** or push it to your Lab 7 repository before you leave.

Background: One State, Two Layouts

FIPS-197 draws the state as a 4×4 grid of bytes. Your HDL stores it as one 128-bit bus. Every test vector you take from the standard has to be converted from one to the other, and getting that conversion wrong is one of the most common Lab 7 bugs.

The packing used throughout the starter code is in the comment above `aes_core`:

[127:120]	[95:88]	[63:56]	[31:24]	S0,0	S0,1	S0,2	S0,3
[119:112]	[87:80]	[55:48]	[23:16]	S1,0	S1,1	S1,2	S1,3
[111:104]	[79:72]	[47:40]	[15:8]	S2,0	S2,1	S2,2	S2,3
[103:96]	[71:64]	[39:32]	[7:0]	S3,0	S3,1	S3,2	S3,3

Each 32-bit word is a column of the state, not a row. To turn a grid into a 128-bit value, read *down* each column, left to right.

For example, the **input** grid at the top of Appendix B is

```
32 88 31 e0
43 5a 31 37
f6 30 98 07
a8 8d a2 34
```

Reading down the columns gives `128'h3243f6a8_885a308d_313198a2_e0370734`. That is exactly the **plaintext** in `testbench_aes_core`, which is a good way to check that you've got the direction right.

When a test fails, the workshop testbenches print the state back out as a grid, so you can compare it directly against the figures in the standard.

Exercise 1: The State Array, by Hand and in Hardware

About 20 minutes.

Converting by hand once is how you learn the layout. Converting by hand every time is how bugs get in. So you'll do it on paper first, then write two small modules that do it for you, and test them.

Part A: On paper

1. The Appendix B **Cipher Key** is 2b7e151628aed2a6abf7158809cf4f3c. Write it as a 4×4 state grid.

```
-- -- -- --  
-- -- -- --  
-- -- -- --  
-- -- -- --
```

2. Find the **Start of Round** grid for Round 1 in Appendix B. Pack it into a 128-bit value: 128'h_____.
3. Which bits of the 128-bit value hold $S[2][1]$? Which byte of the state is in bits [39:32]? Now write the general rule: the byte in row r and column c is in bits [_____ : _____].
4. In SystemVerilog, you can hold the state as a 4×4 array of bytes:

```
logic [0:3][0:3][7:0] s; // s[r][c] is the byte in row r, column c
```

The ranges are [0:3], not [3:0], on purpose. With [0:3], $s[0][0]$ is the leftmost byte, so a value written row by row, {8'h32, 8'h88, 8'h31, 8'he0, 8'h43, ...}, fills row 0 first and reads just like the grids in the standard.

Because s is packed, it is also 128 bits wide, so $s = a;$ compiles. Write down the grid it would give for the Appendix B input. Why is it wrong?

Part B: In hardware

[Open Exercise 1 in Jacquard](#)

File	What's in it
input_to_state.sv	An empty <code>input_to_state</code> module: 128 bits in, state array out.
state_to_output.sv	An empty <code>state_to_output</code> module: the reverse.
input_to_state_tb.sv	A self-checking testbench, with two expected grids for you to fill in.
state_to_output_tb.sv	A self-checking testbench, with one expected value for you to fill in.

1. **Run `input_to_state_tb` before you write anything.** Every test fails. A testbench should fail loudly when it's incomplete instead of quietly passing.
2. **Fill in the two TODO grids in `input_to_state_tb.sv`,** row by row, the same way Test 1 is written. One is the cipher key from Part A. In the other, each byte's value is its own position in the 128-bit input.
3. **Write `input_to_state`** and run until the testbench prints PASSED. If your got grid is the expected grid transposed, look back at Part A, question 4.
4. **Write `state_to_output`,** fill in the TODO in `state_to_output_tb.sv` with your answer to Part A, question 2, and run `state_to_output_tb` until it prints PASSED. Its last test sends a value through both of your modules and checks that it comes back unchanged.

You now have two tested modules that move between the 128-bit packing and the state array. You can use them in Exercise 2 and in Lab 7 whenever rows and columns are the easier way to think.

Exercise 2: ShiftRows, From the Spec to a Passing Test

About 15 minutes. [Open Exercise 2 in Jacquard](#)

File	What's in it
<code>shiftrows.sv</code>	An empty <code>shiftrows</code> module for you to fill in. Its ports are 128 bits, like everything inside <code>aes_core</code> .
<code>shiftrows_tb.sv</code>	A self-checking testbench, with three test vectors for you to fill in.
<code>input_to_state.sv</code> , <code>state_to_output.sv</code>	Empty, and only needed if you use the state array. Paste in your modules from Exercise 1.

How you write `shiftrows` is up to you. You can move bytes around in the 128-bit vector directly. Or you can turn `a` into a state array with `input_to_state`, shift each row, and turn the result back with `state_to_output`. The testbench only sees the 128-bit ports, so it checks both the same way.

1. **Run the testbench before you write anything.** Read the transcript. Every test fails, because the test vectors are still `x`. A testbench should fail loudly when it's incomplete instead of quietly passing.
2. **Read Section 5.1.2 of FIPS-197 (ShiftRows).** In your own words: which bytes move, and by how much?
3. **Predict the “numbered bytes” test by hand.** Its input puts each byte's position in the byte itself, so $S[r][c] = 4c + r$. Draw the 4×4 grid before and after ShiftRows, then pack the “after” grid into a 128-bit hex value and fill it in as the expected output.
4. **Fill in the two Appendix B tests:** the “After SubBytes” grid as the input and the “After ShiftRows” grid as the expected output, for Round 1 and for Round 2.
5. **Write `shiftrows`,** whichever way you chose, and run until the testbench prints PASSED.

When a test fails, the testbench prints what you got next to what was expected, as grids, with a `*` on every wrong byte:

got	expected
00 05* 0a* 0f*	00 04 08 0c
01* 06* 0b* 0c*	05 09 0d 01
02* 07* 08* 0d*	0a 0e 02 06
03* 04* 09* 0e*	0f 03 07 0b

💡 Read the shape of the error

Before you open the code, look at *which* bytes are wrong:

- **One row wrong:** the shift for that row is wrong.
- **The got grid is the expected grid transposed** (as above): rows and columns have been swapped somewhere, probably by treating each 32-bit word as a row. If you used the state array, re-run your Exercise 1 testbenches on the modules you pasted in.
- **Wrong bytes contain values from elsewhere in the input:** bytes landed in the wrong place. The numbered-bytes test tells you exactly which byte went where.

Check yourself. If your `shiftrows` were broken, which of your three tests would tell you the most about *how* it was broken? Why?

Exercise 3: Find the Bug in MixColumns

About 20 minutes.

The `mixcolumns` module in the Lab 7 starter code is correct. Each of these three copies has had **one** bug planted in it, and each bug is different. Your instructor will assign your pair a letter.

Version	Link
A	Open Exercise 3A in Jacquard
B	Open Exercise 3B in Jacquard
C	Open Exercise 3C in Jacquard

❗ Don't diff against the starter code

You could find the bug in seconds by comparing against `aes_starter.sv`. Don't: there won't be a correct version to compare against when the bug is in your own design. The goal is to practice the method.

1. **Run the testbench.** The first two tests come from a published table of MixColumns examples, and they pass. Before you go on, write down how a *broken* `mixcolumns` could pass both of them. (Hint: in $\text{GF}(2^8)$, what is $a \cdot a$?)
2. **Fill in the two Appendix B tests** ("After ShiftRows" $\rightarrow$ "After MixColumns", Rounds 1 and 2) and run again. Now it fails.
3. **Read the shape of the error.** Are the wrong bytes scattered, one whole row, or one whole column? Write down a hypothesis about where the bug is *before* you open `mixcolumns.sv`.
4. **Find the first wrong signal.**

- Pick one wrong byte. Its column `c` tells you which `mixcolumn` instance computed it (`mc0` through `mc3`).
- In the waveform viewer, open that instance and add `a0–a3`, `tmp`, `t0–t3`, and `y0–y3`. Set the radix to hex.
- First check that `a0–a3` match the column of the input you expect that instance to be working on.
- Each output byte is $y_i = a_i \oplus tmp \oplus t_i$. Work out by hand what each of those three should be. Which one is wrong?
- Keep working backward into whatever produced the wrong signal (for a `ti`, that’s a `galoismult` and its input). Use the Galois field arithmetic from Lecture 13: multiplying by `x` is a left shift, followed by XOR with `0x1B` if the bit shifted out was a 1.
- Stop when you reach a signal whose inputs are all correct but whose output is wrong. The bug is in the line that computes it.

5. **Fix the bug and re-run.** All four tests should pass.

6. **Write down the chain of evidence**, in three sentences: the symptom, the first wrong signal, and the cause. Then trade stories with a pair that had a different letter.

If you finish early, try another letter. Or design a single test vector, simple enough to check by hand, that would catch all three bugs.

Stretch: Trace Your Own `aes_core`

If you have started your `aes_core`, this testbench applies the same method to the whole design. Download [aes_core_trace_tb.sv](#) and add it to your Lab 7 project (**Upload** in Jacquard, or add it to your QuestaSim project).

The tracer watches one register inside your design. Every time that register changes, it looks the new value up in the complete Appendix B table and tells you which step of which round it matches. It assumes nothing about your timing, so it works whether you register the state once per round or after every step.

1. Edit the `TODO` near the top so that `state_probe` points at the register that holds your state, using its hierarchical name (for example `dut.state`). The file won’t compile until you do.
2. Optionally, point `key_probe` at your round key register and set `TRACE_KEY` to 1.
3. Select `aes_core_trace_tb` as the testbench and run.

Here is what a design with a bug in its Galois multiply looks like:

```
cycle   1  state = 3243f6a8885a308d313198a2e0370734  matches input
cycle   9  state = 193de3bea0f4e22b9ac68d2ae9f84808  matches round 1 start
cycle  15  state = a49a7ff46899332b6d5bec43026c504f  NO MATCH
           ^ first divergence: the last state value that matched was round 1 start
```

The bug is somewhere between “round 1 start” and the next value that should have matched. So take round 1’s intermediate values from Appendix B and test each step module on them, just as you did in Exercises 2 and 3.

The tracer also catches two problems that make `testbench_aes_core` hang without saying anything:

- **done is x.** Your FSM isn't put into a known state at the start. Check what `load` does to your state register.
- **done never rises.** The tracer gives up after 200 cycles instead of running forever.

Design for observability

If there isn't a single register you can point `state_probe` at, take that as a hint about your datapath. A design whose state lives in one clearly named register is much easier to debug, both here and on the logic analyzer.

The tracer is a debugging tool. It does not replace `testbench_aes_core` and the full SPI testbench, which are what the Lab 7 [specifications](#) require to pass.

Take It to Lab 7

- **Work in whichever layout makes each step easiest to get right.** Rows and columns are explicit in the state array, and your tested `input_to_state` and `state_to_output` take you there and back.
- **Test each step module on its own against Appendix B before you connect them.** A failing `aes_core` has many possible causes. A failing `shiftrows` has only a few.
- **Choose test vectors that make a bug easy to find.** Numbered bytes show you where every byte went. Inputs where a bug cancels itself out (like a column of identical bytes in `MixColumns`) prove very little.
- **Read the shape of the error before reading code.** Scattered bytes, a whole row, or a whole column each point somewhere different.
- **Find the first wrong signal and work backward.** Stop when the inputs are right and the output is wrong.
- **Write down your hypothesis before changing code,** change one thing at a time, and re-run *all* of your tests after every fix.
- **Remember the sbbox latency.** Run `aes_sbox_tb` from the starter code and mark on the waveform when `y` becomes valid relative to `a`. Your FSM has to wait for that cycle.