Functional units in Verilog

In digital logic there are often multiple ways of implementing the same module by building circuits with different area-time trade-offs. There are also multiple different ways of expressing the same thing using Verilog language features, and different ways of testing the same module functionality.

This folder contains implementations of three different functional units:

- An iterative multiplier.
- A pipelined multiplier.
- A single-port register file.

Each functional unit also has at least one test-bench.

Testing the code

All the functional units and test-benches are correct (or at least, I don't know of any errors).

1. Apply each test-bench to each implementation, and check they are correct.
2. You might like to write a script to test each group of implementations with one command.
3. You might like to write a script to test all functional units with one command.

Understanding the code

Look at each implementation, and check you understand it's functionality.

- Look at the module interfaces. Ignoring the implementation, what are the semantics? How would you document them?
- Compare the different language features being used - which do you prefer?
- Sketch the circuit that the Verilog describes, paying attention to flip-flops versus combinatorial logic.
- Consider any area-time implications. Is one implementation bigger than another? Does one take more cycles?
- Try to get a sense of the critical path. How do you think the placement of combinatorial logic relative to flip-flops works out?

Understanding the test-benches

There are four test-benches, which use a combination of styles

- Can you see any potential fragility in the test-bench?

- Is it possible for the test-bench to look like it succeeded when it failed?
- Are there edge cases that could be considerer?
- How do the different approaches to generating "random" inputs differ?

Breaking and fixing

Having seen working code, one of the most useful exercises is to break it, then get other people to fix it. This will expose you to the types of cryptic error messages that come out of the compiler, and what that might mean. It is suggested that you pair up, and each introduce subtle errors into each implementation. You then swap code snapshots and try to fix what was broken.

Suggested types of errors include:

- Syntactic errors: delete brackets, add brackets, add an extra `begin`,...
- Language errors: use an input as an output, use the wrong assignment type, turn something into a `wire`, ...
- Semantic errors: modify a constant, swap two statements around, invert a condition, ...
- Break the test-bench, but leave the circuit correct.

You should try to be as devious and subtle as you can, to force people to deploy different debugging techniques.