3.1

3.1.1

Assertion 1 holds, assertion 2 fails within 1 cycle from reset

3.1.2

The assertion itself is wrong based on how a D-flip flop is expected to behave. The input 2 cycles ago is not necessarily the same as the current output. For instance with the input sequence 1 0 0, at the third cycle Q will be 0, while D 2 cycles ago was 1.

3.2

3.2.1

We forgot to add a specific property to check the behavior when applying 0 0 to the inputs.

3.3

3.3.1

It should not be possible (except for right after rst), as it is dependant on the internal "busy" signal, which in turn is dependant on "valid" being asserted. If "done" is asserted without receiving a valid input, it is due to the surrounding design sending the wrong signals.

3.3.2

Yes. Since it checks for the correct result after the correct amount of cycles, any bugs made by i.e. wrongfully resetting internal signals can still be caught. We are also asserting that "done" is only high for exactly 1 cycle, at the right time, through a combination of "done_low" and "verify_computation". All outputs are therefore verified accordiing to the spec, and since we use a formal verification tool, all possible inputs are checked.

3.3.3

combinatorial is much easier to write verification for, and runs in way less cycles (immediate versus "WIDTH" amount of cycles). However, the combinatorial circuit has a long critical path which could slow down the clock rate of a larger system. The sequential also guarantees the output for (near) a full cycle, while the combinatorial is only correct as long as the input vector is held at it's appropriate value.

3.3.4

Total runtime is as follows:

- sequ16: 0 s
- sequ32: 0 s
- sequ64: 0 s
- sequ128: 6 s
- sequ256: 33 s
- comb16: 0 s
- comb32: 0 s
- comb64: 0 s
- comb128: 0 s
- comb256: 0 s

It seems to only count in whole seconds (despite showing 2 digits after the decimal). The combinatorial implementation is too quick to even measure in this scale, but the sequential seems to scale a little more than quadratically with input size (i.e. 2x input = 4x run time). This fits with the fact that you have double the amount of input bits to change, as well as double the amount of cycles that input can vary before the output comes on the other end. Even though input is not read during that time by the RTL, the input *could* theoretically change the output, so they are tested since this is formal verification.

3.4

3.4.1

Assume that we are in IF phase, as that is when we can read what instruction we have gotten

3.4.2

It should be IF phase, to ensure that the program returns to the correct phase to start the next instruction

3.4.3

The jump instruction is executed in the ID phase and therefore takes two cycles before returning to the IF phase.

3.4.4

The same goes for the branch instruction, it will take two cycles to execute it.

3.4.5

The difference between writing `PC == prev_PC + 2` and `PC == prev_PC + 16'd2`, is that if we write `2` instead of `16'd2`, the value of `2` has an **undefined width**, which can lead to unexpected conflicts due to the **implicit casting**.

By using `16'd2`, we force the constant to be a **16-bit value** so we ensure it works properly with the other **16-bit values**.