You know what they way - sometimes small things can make a big difference. Setting your A/C thermostat to a reasonable temperature can help save energy, sending flowers to someone you care about and so on. This is also true in programming. Sometimes even a seemingly redundant single line of code could make a big difference. I'm not talking about the core logic of your code, I'm talking about something so simple and straight-forward that we may tend to forget about its existence. I'm talking about the habit of writing assertions in your code.

Don't get this wrong - I'm not talking about assertions in the context of checking the DUT, I'm talking about adding a safety net to your own verification code so that it would be far easier to catch bugs in the verification environment as early as possible.

Let me give you an example (given in e, but in SystemVerilog you can pretty much do the same so read on):

Let's say you want to write a method that receives a list of packets and sums up the overall number of bytes in all the payloads. Here's the e code for doing that:

{code}

```
sum_all_bytes(packets: list of packet): uint is {
for each in packets {
 result += it.payload.size();
 };
};
```

{/code}

This piece of code is going to get the work done alright, but there might be something missing. We haven't verified our implicit assumption that the list actually contains at least one item. Adding a simple assertion to verify this could save us from trouble later on because we don't know in advance how this method will be used. What we do know for sure is that we wrote this method assuming nobody is ever going to call it with an empty list. So let's add a simple assertion to take care of that:

{code}

```
sum_all_bytes(packets: list of packet): uint is {
assert that packets.size() > 0 else error ("packets size should not be zero");
 for each in packets {
 result += it.payload.size();
 };
};
```

{/code}

Note that the use of error rather than dut_error is not accidental. The error situation here will rarely have anything to do with the DUT, so we don't use the dut_error command.

So the nice thing about protecting your own code with assertions is that at a really small price (usually a single line) you could save yourself a lot of debug time in cases where the root cause resides in the environment. Assertions will normally help verify your assumptions on the parameters of a function and help you nail programming errors the very minute they happen.

Let's take another example. Let's say you want to write a frame generator for some TDM-based interface where each frame is exactly 50 bytes long. You allow random values to be generated for the various fields and then you want to pack everything into a stream of bytes. The generator also feeds a scoreboard.

Your packing method might look like this:

{code}pack_frame(): list of byte is {
 result = pack(packing.high, header, payload);
};{/code}

So far so good, but is there any implicit assumption we're making on any of the elements? To answer this question let's assume that for some reason (programming error, generation corner case and a bit of bad luck, etc.) one frame is generated with an extra byte in its payload (51 bytes instead of 50). It might happen.

Now the packing method doesn't know that, because it simply takes the physical fields and re-formats them as bytes. The BFM in turn will take the stream of bytes and send them to the DUT one by one. No problem so far and that's exactly the problem

The generator has violated a rule (need exactly 50 bytes per frame) causing an imminent mismatch between what the environment thinks and what the DUT actually sees but we won't know it until some scoreboard fails and we spend some time in debug. As you probably guessed, a simple assertion in the packing method can make sure that the packing operation succeeded and the frame length is valid:

{code}pack_frame(): list of byte is {
 result = pack(packing.high, header, payload);
 assert result.size() == 50;
 };{/code}

This will catch the problem as soon as it happens.

I hope you managed to get the hang of it. I honestly believe that if you make it a habit to reconfirm your assumptions with assert, you will end up with a robust environment, and save yourself precious debug time. Plus, the code will be a bit more understandable as you give more hints about your intent. So my recommendation is to be assertive!