Design Verification

Refresher Notes: Week 1

These notes are provided as a brief refresher and guidance for you at each lab session. These are note comprehensive notes; these are not revision notes; these do not contain everything you need to know; these do not tell you exactly how to do the entire coursework.

Introduction

- Kerstin Eder, Ed Nutting, Dave McEwan
- Kerstin is available after lecturers, in her office a lot of the week, or via email (Kerstin.eder@bristol.ac.uk) – email may not be reliable.
- Ed and Dave are available in the lab session or via email: Edward.nutting@bristol.ac.uk

Lectures, Labs & Help sessions

- Lectures in weeks 1 to 12 excluding 8
    - 11:00-11:50 Tuesdays (QB 1.68)
    - 14:00-15:50 Tuesdays (MVB 1.11)
- Labs in weeks 1 to 12
    - 14:00-14:50 Wednesdays (MVB 2.11)
    - May shift if timetable conflict
- Help sessions in weeks 1 to 12
    - Available on demand
    - 17:00-17:50 Thursdays (MVB 1.11 / QB 1.7 in week 4)

Assessment

- 2 assignments (25% each), 1 exam in January (50%)
- 1st assignment due in week 4 or 5
- 2nd assignment due in week 10 or 11 or 12 – TBC
- Obtain "feed forward" before each deadline to boost your mark

Online materials

- Blackboard contains complete copies of slides
- Blackboard assignments also come with additional necessary material
- Verificationacademy.com

How to pass this unit

1. Do the coursework early.
    a. It is set early so you can get it out of the way before other modules set their coursework
    b. Verilog is **hard.** Get stuck into it now while you have 4 lab sessions (i.e. 4 hours of TA help time!) available. **We can't help you if you leave it to the week before the deadline.**
2. Don't spend ages going in circles.
    a. Ask for help from us (Kerstin, Ed and Dave) early on, before you spend ages in a blind alley.

    b. Verilog is **hard.** We understand, we're here to help.
    c. Verilog is not normal programming. It is a hardware description language. It describes circuits – i.e. stuff that strings together like a circuit – circuits happen in parallel. Appreciate this and you'll get ahead.
3. Understand the features of Verilog and how to use them before attempting the coursework.
    a. Play around with the example code given inside ModelSim for a couple of hours.
    b. Understand how tasks and functions work
    c. Understand how loops work in Verilog (they are not sequential loops like in Java)
    d. Understand how the parallelism of Verilog descriptions (aka code) can help you (rather than hinder you)
4. Plan your work before you start
    a. Do the test plan – it will save you a lot of time in the long run
    b. Do the test plan – it will give structure to your code which helps us give you a good mark
    c. Do the test plan – you can get feedback on it straight away which helps us guide you in the right direction
    d. Do the test plan – It will help you think about things you've missed
    e. Do the test plan – If you don't know where to start, you need our help, so please just ask
    f. Do the test plan – Are you getting the vibe yet? 😉
5. "Working state to working state"
    a. Transform your code from the example to what you want slowly
    b. Add a little bit of code, test it in ModelSim, then continue
6. Target the low-hanging fruit
    a. Directed testing may not be great, but it's a lot easier than the other (harder) methods you'll be learning about. Start with some directed tests.
    b. Then do some systematically generated tests.
    c. Then attempt some more thought-out (ie optimised) systematically generated tests
    d. Oh, you got this far? We bet you've missed something. Keep doing those smart tests – random generation will come in assignment 2.

What are we trying to achieve?

1. Understanding of verification processes: How do people actually verify stuff?
2. Understanding of verification terminology: Aka. How you communicate to other engineers who do verification of stuff
3. Practical skills:
    a. Some practical use of standard hardware verification tools
    b. Some practical use of cutting-edge hardware verification tools
4. Understanding and experience of the challenges of verification

What is verification anyway?

1. Verification: Making sure something works right: first time, every time
2. Validation: Checking whether a system fits the environment it's a part of.

Why isn't stuff just correct when we build it?

1. When you create a design, you have some sort of specification for how it should work
    a. Whether you've written that spec down or not doesn't matter

    b. In your head, there is some kind of idea as to how the thing is supposed to work. This is an informal specification
2. When you translate a design from your idea in your head into code, you probably produce something close to what you thought, but not exactly the same
    a. For one thing, programs require a lot more precision than our thoughts do
    b. You probably didn't consider every possible input or environment your thing could be used in. Hey, what happens if I throw it at the sun? 😊
3. There is probably some kind of consequence if your design doesn't match what you thought it should do
    a. Remember that probe they sent to Mars that crashed into the planet because of a units conversion error?
4. So you'd probably like some assurance that what you designed did what you thought it should do
5. And that "assurance" is achieved by verification

Verification is relative in two ways:

1. You can only verify against the specification you think of. If you design a phone but don't think about what happens if it gets wet, then your spec won't say anything about verifying what happens when your phone is dropped in the sea (or a toilet, more likely).
    a. Specifications never account for everything. They account for what we care about and/or are aware might be worth caring about.
2. You cannot (through testing) prove the total absence of mistakes. You can only prove that they are more or less unlikely to exist.
    a. Exhaustive testing still only tells you that no bugs exist in the areas you tested against the specification you created. If someone does something outside the spec (which you may or may not expect to be allowed) then they may uncover a bug.
    b. This is especially problematic in systems which interact with humans, as humans don't stick to specs. E.g. what happens if I put metal in a microwave? Does the software stop the microwave when it senses burning?
        i. Side note: Almost every system interacts, or could interact, with humans, regardless of how embedded it is.

How much do mistakes cost?

0 to infinity. Add more if you don't catch them until it's too late. This gets worse with hardware as opposed to software. Could also cost lives, privacy, liberties, etc.

How do we make mistakes cost less?

1. Verify stuff from the start.
2. Make verification faster (more productive).
    a. Parallelism: Add more resources (people, time, money, compute power, etc)
    b. Abstraction: Use a higher level language (eventually you end up using plain maths)
    c. Automation: Make a computer do it for us 24/7/365, not generally possible due to NP completeness or problem size explosion
3. Make verification more comprehensive.

What kinds of verification?

1. Functional

2. Behavioural
3. Timing
4. Integration / Bus-level
5. System
6. Manufacture / Fabrication test

Formal verification: Checks every possibility. Founded on mathematical reasoning that provides strong guarantees.

Non-formal verification: Do a bunch of tests, measure your coverage to see if you covered enough of the design. If you did, whoop: you've gained the confidence you want. If you didn't: keep verifying.

What results might a test give?

1. Type 1: False positive
    a. The design didn't contain a bug
    b. You found what appeared to be a bug but actually the test or spec was wrong
2. Type 2: False negative
    a. The design did contain a bug
    b. You failed to find a bug and it ended up slipping through into the final design.
        i. You didn't have enough coverage
        ii. Or your spec was wrong so failed to spot the mistake
        iii. Or your tests were wrong
    c. But overall, there was a bug in the design itself, and it was missed in testing.
3. Type 3: True positive
    a. There was a bug in the design and you found it. Whoop, go fix it.
4. Type 4: True negative
    a. There wasn't a bug in the design, you tested for one, and you correctly identified the absence of a bug. Great, but this is what we were hoping for so just carry on searching for bugs.

What am I learning?

You're going to be learning about a wide range of aspects of verification in your lectures.

For your coursework, you will practice a small part of verification: functional and timing verification at the unit level. (A unit is a module of hardware, a bit like a 'class' or 'module' in software)

How do I do the coursework?

A few notes on your coursework:

1. Assignment 1 is black-box testing – so you just drive the inputs with values and check whether the outputs are correct or not, and whether they appear at the right time or not.
2. You are testing at unit level
3. The specification you're given may or may not be complete
4. Your test plan is vital. How many input scenarios you test matters, and no amount of coding is going to make you think of more. Sitting down and writing out the test plan will force you to think systematically and thoroughly about all possible inputs to the design (whether they are valid or otherwise).
5. There are no bonus marks for working 30hr a week on this coursework just because you got stuck and didn't ask for help.

6. There are no extra marks for working a stupid number of hours on this coursework. If you can't do it in the time intended, please ask for help – it's not supposed to take *that* long.
7. Don't try to test everything – you won't have time to run exhaustive tests (e.g. (2^32)^2 possible inputs just for addition is more than you can test in a lifetime – think of a better way to test the meaningful combinations – the ones that give you most information about what could be going wrong.)
8. Driving in Verilog = Assigning an input signal (aka wire aka variable) a value
9. Checking in Verilog = Comparing the value of an output signal (aka wire aka variable) to its intended value
10. Your testbench is simulated as though it too is hardware – you cannot write your testbench code like normal code. It is not "executed" imperatively. It is simulated as hardware. It has the same design and timing constraints as hardware. You will need to consider these things.

Watch this video and attend the lab sessions for how to access the software:
https://youtu.be/NQGvJ47RsSM

We will cover how to use the software and learning Verilog in future sessions.

What steps should you take to tackle the coursework as whole? Here are some suggested ones:

1. We will do our best to help you learn Verilog
2. You will write a small generic framework in Verilog for testing the design you are given
3. You will then write a test plan
4. You will then write tests in Verilog as set out in your test plan
5. You will then evaluate tests which fail (i.e. detect incorrect output) and write a 1 page report to tell us what you've found
7. We will assess how good your testbench is.
    a. We will be running your testbench against a different model to the one you receive.
    b. Our model will match the same specification and code interface but will contain different bugs. This allows us to evaluate how comprehensive your testbench is (i.e. your practical coverage).

What IDE do I use?

Sorry, Verilog sucks, it doesn't have a good IDE.

But you could try:

- VSCode + the Verilog extension
- Emacs + Verilog-mode
- Notepad++ with Verilog language

Should I use Verilog or SystemVerilog?

Use SystemVerilog. It's a little trickier to learn but MUCH more powerful – you will find it a lot easier to write loops, test modules, etc in SystemVerilog than in Verilog.

All Verilog code is also SystemVerilog code, but not the other way around.

SystemVerilog uses .sv extension.

Remember to configure ModelSim to use SystemVerilog not Verilog – ask a TA for help.

How should I structure my code?

Download the example code. This initially has no structure. Start by practicing writing Verilog functions and tasks in this one file.

For example:

1. Extend the existing code by adding 3 more directed tests
2. Refactor the existing code by moving the constant, hard-coded values into functions (which just return the same constant values)
3. Refactor the existing code by moving each directed test into a separate task
4. Extend the existing code by allowing each value function to take parameters to determine the value generated
5. Extend the existing code by allowing each task to take parameters to determine the values it tests (by passing those input values to the value functions)
6. Extend the existing code by putting each test task in a loop that generates different inputs for each expanded iteration of the loop.
7. You have just achieved some basic systematically generated directed tests. You are 1/3 of the way to your testbench.

You should now consider parallelism in your testbench. At the moment, you start a test, then wait for the result, then check the answer. This is not a good overall strategy for more complex tests. In the next lab session, we will discuss how to make a better, more general, parallel framework.

How transferable is this stuff? Can I apply it to software?

Yes, testing applies to software. It also relies on the same principles and is designed in very similar ways. For example, the tests you do in your coursework are analogous to (and designed in the same way as) unit tests in software.