Verification & Debugging

COMP32211

Implementing System on Chip

Verification and Debugging

There are lots of potential sources of errors in a design.

This session ignores most of these – and concentrates on **functional test**.

DESIGN THE TESTS!!

- Does it reset?
  - Is the initial state achieved?
  - Are there embarrassing 'unknown's in the circuit?
- Does it do the most straightforward operations?
- Have you tested every statement?
- Can you break it?

There are techniques to help with testing

- Tools — but they will not find everything for you
- What didn't you think of?
- Don't throw working tests away — they will be needed again. (Regression tests.)

What to look for: control

- Start with simple sequences
- Do the interfaces go through the expected protocol steps
- Are the FSMs behaving as expected? (Observe internal state.)
- Have all the transitions in the state diagram been observed?...
- … for all the reasons which might trigger them?
- Termination
- Test coverage

E.g. a new processor implementation: first feed instruction bus with 'NOP's

- Does it reset and start fetching?
- Does it perform fetch-decode-execute sequence (or pipeline phases)?
- If the 'instruction memory' (NOP feeder) is slow (wait states) does it stall
- …

What to look for: data

Just because a design runs through appropriate control sequences this doesn't mean that the output data is correct.

In an RTL abstraction the details (values) of the 'data' are not always considered.

But it is important to check that the data are also correct.

- Data values can be compared with independently generated test results
- Some data faults become readily apparent
  - Out of range values
  - The wrong number of operations (cycles) are performed
  - 'silly' graphics drawing (etc.) … if the values can be viewed
- Self-testing may be possible
  - A processor can be tested by running some software
  - It will probably crash if there is an error in (e.g.) instruction decode

Test coverage

- How do you know what you have (haven't) tested?

Wouldn't it be nice if there was CAD assistance?

There are test coverage tools which assist with this task.

Test coverage may indicate such as:

- Which lines of a HDL were executed (at least once)
- Which decisions were (not) taken during a simulation run
- Which wires have adopted both possible digital states
- Which states of an FSM have been visited
  - Which transitions have been traversed

May even tell you things you didn't care bout

- e.g. flagging having missed an unreachable 'default' in a case statement

Works on the 'better safe than sorry' principle!

How to s-t-r-e-s-s things

There are various approaches to designing verification tests. Here are some suggestions:

- Individual, deliberately targeted tests
  - Manually contrived so that (at least) one case of every (known) circumstance is used
  - Effective. Expensive.
- Algorithmically generated
  - e.g. all possible cases of inputs, regular input patterns …
  - Relatively easy to do
  - Requires *some* design effort
- Random patterns ('Monte Carlo')
  - Typically gives high (but incomplete) coverage of *common* cases
  - Probably misses specific input combinations
  - Cheap to produce

In general a combination of the above may be appropriate.

Ordering and Scheduling

Consider two friends you go out with: Alice and Bob.

Alice is always early, Bob is usually late.

```
if (Bob is here) then set off
```

Seems to work … maybe passes routine tests …

One day, Bob turns up early! Later on you suddenly think 'Where's Alice?'

- Test coverage can't help because it can only check the things you asked to do.
- A test strategy should strive to test every case in every conceivable order.

Techniques for finding faults

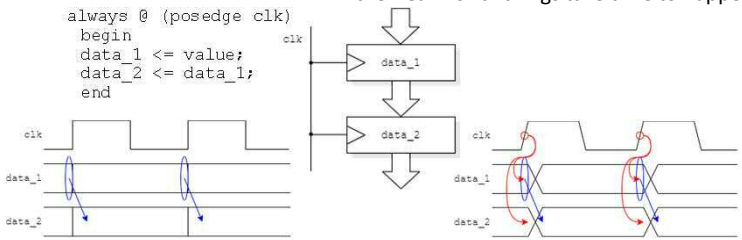
- Readability of traces
  - If no real-time necessity, choose a 'simple' clock frequency
  - Add some (testbench) delays to signal changes for visibility
  - Add some signal(s) to indicate errors, test phase etc. to testbench
    - Waveform viewers allow searching for signal values & changes
- Export diagnostic information
  - `$time`-stamp printed error reports
  - Print status occasionally (e.g. "`Phase 2 succeeded`")
  - Maybe `$stop` after an error occurs …
  - … or stop reporting after N occurrences of the same thing (reduces clutter)
- **Test the tests**
  - Induce (temporary!) failures in DUT to make sure tests find them!

‘Cosmetic’ delays

In simulation there is no need for time to pass for causality.

In the ‘real world’ things take time to happen.

```
always @ (posedge clk)
 begin
 data_1 <= value;
 data_2 <= data_1;
 end
```



The behavioural model above will work because:

- Non-blocking assignments happen simultaneously (regardless of written order)
- … but this can be confusing to read from a trace

Using tasks

- Reminder: ‘task’ is Verilog for method/procedure/subroutine/…
- Designed for repeated invocation
- Ideal for (e.g.) sending test data across a bus
- Here’s an example task which executes a communication handshake

```
task strobe;
  begin
  req <= 1;
  while (ack == 0) @ (posedge clk);  // Wait for acknowledgement
  req <= 0;
  while (busy == 1) @ (posedge clk); // Wait for operation to complete
  repeat (10) @ (posedge clk);       // Pause before next command
  end
endtask
```

This can be invoked with a single keyword which saves a lot of typing/clutter.

P.S. this is not particularly good code: don’t just copy – improve upon it.

Reviewing

- People (*even engineers*) are fallible
- psychologically you tend to see what you expect to see

Reviewing is a process by which others try to find faults in your work.

Useful at all stages of development

- Design review
  - Have all eventualities been though of?
  - Is it more complicated than it needs to be?
- Code review
  - Does this code do what it was meant to?
- Test review
  - Are there cases going untested

… etc.

Regression testing

Test process

- Identify something which needs testing
- Devise a test for it
  - Can *extend* a previous test but don’t simply change one
- Apply test
  - Iterate until design passes

Then

- **Keep the test**
- Continue development
- Rerun the test (periodically)
- Iterate until the design is (believed) correct and complete

That way the final design still meets all the test criteria.