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 a working test away – it will be needed again!

---

**Verification**

The Reasons for Verification:

To establish that a design or implementation ‘works’ …in all imaginable cases. The ‘first cut’ of a new design will almost certainly have some fairly gross errors. Because they are so bad they cause symptomatic faults very rapidly and so are easy to find. It is fairly normal to start with some crude tests and simply examine a short output trace at this stage. More complex, ‘exceptional’ cases may gradually be added and examined.

Later the design may be modified to fix esoteric faults or to accommodate extra features. Any changes will invalidate all earlier test results so the tests will need to be repeated. This ‘regression testing’ should be planned so that it is as cheap as possible to perform.

It is difficult to devise ‘high-level’ tests which can exercise every ‘low-level’ component thoroughly. [Imagine picking a random, 3-input gate in a Pentium and wanting to write a programme that used all eight input combinations.] Therefore deliberate testing and qualification of each unit in isolation is essential if the whole system is to work under all design circumstances.

As always the process needs to be both as **thorough** and as **cheap** as possible.

---

**Jargon**

Several words are used interchangeably for the test process. It is not possible to give hard-and-fast rules on these, but as a guide:

- “Verification” is the process of determining that what is built is what was specified.
- “Validation” is the process of determining if the final system meets the requirements.
- “Testing” can cover all the processes but is also used specifically for the post-production process of determining if a particular instance of the design is fault-free.

(The first two of these are formalised by the IEEE w.r.t. software.)
**What to look for: Control**

- Start by stimulating simple sequences
- do the interfaces go through the appropriate phases
- are the FSMs behaving as expected? (observe internal state)
- have all the transitions in the state diagram been observed …
- … for all the reasons that might trigger them
- termination
- (test coverage)

- e.g. a processor fed with 'NOP’s
  - resets and starts
  - performs fetch/decode/execute sequence
  - can stall if memory is slow
  - …

---

**‘Families’ of tests**

It is often convenient to divide tests into control and data tests. The former checks the sequencing and timing of operations so that the correct things happen in the desired order. In such tests data values may not be important.

The latter checks the data values as expected.

These are not ‘hard and fast’ divisions. Examples:

- control may include a test for data stability even though the value is not checked
- is the number of cycles performed ‘control’ or ‘data’?

**Thinking up test cases**

Consider an ALU in a pipelined processor design. This performs a given set of operations, typically one per clock cycle although pipeline stalls and flushes may interfere with this. May want to check:

- Normal data flow through unit
- That the output retains data when stalled
- If the output stalled is the input stalled?
  - this may be under local or external control
- Is data validity passed through correctly?
- Is an attempt to stall the output ignored if the stage contains invalid data?
  - this would be an optimization for performance
- Does the unit ADD, SUB, AND, etc.?
  - signed, unsigned values, shifts …
- If an input is unused (e.g. MOV) is it genuinely ignored?
  - a use for ‘don’t care’ values, perhaps

Each test is simple in itself but they can be numerous, even in a ‘simple’ unit! It is likely that this list is not exhaustive.

---

**Example: An interface between blocks**

In the laboratory system the pixel output interface from the drawing engine has the following timing rules:

There are two control signals, {req, ack} with an associated bundle of data. Following an active clock edge:

- if req is not asserted, ack should not be asserted
- if req is asserted, ack may be asserted to indicate the acceptance of data
- ack will be asserted for a single clock cycle for each data item accepted

In the figure:

- The first transfer goes through immediately
- The second transfer has to wait for one cycle
- The third and fourth transfers do not wait and follow each other on consecutive cycles

Notes:

- this interface has been specified to be *quite simple* to use
- small delays have been incorporated into the figure for clarity
- this is not a ‘handshake’ interface as data bursts can go through without separate request transitions

Do the tests cover all these cases?

Is (for example) the data held stable when the second transfer waits?
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 of the ‘data’ are not really considered. 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 … assuming 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

If control signals are interacting correctly the data can be checked. Probably the first thing to establish is that the data is stable when it is expected to be. It is not much use if a pipeline’s control signals stall correctly but the data tries to keep flowing.

The data can be sampled for correctness at the time an interacting unit would do so. This requires some expected results for comparison, which could be generated in a HDL test harness or imported from another model. Note that the expected results should have been verified in some way.

Data crossing interfaces should be deducible (if not directly available) in a higher level model. Such a model can be used in various ways to aid testing.

- to generate stimuli for a HDL model
- to generate a file of expected test results
- as part of a cosimulation where it interacts with the HDL simulation

A high-level model may also be able to generate internal variable sequences with little extra effort.

Exhaustive tests

It is possible to test some units exhaustively, i.e. try every possible case. These are usually combinatorial units as each bit of buried state doubles the number of tests required.

Example: 8 * 8 multiplier block

There are $2^{(8*8)} = 65536$ possible input combinations. It is quite feasible to test this and it is probably the simplest thing to do.

Counterexample: 32 * 32 multiplier block

There are $2^{32*32} = 18446744073709551616$ possible input combinations. Checking at one per nanosecond would still require over 500 years to complete.

Exhaustive tests are feasible for control circuits, not for (most) data.

Representative samples

If exhaustive testing is impossible some sample data are needed. There are various ways of generating these:

- hand generated
- algorithmically generated
- random
- biased random
- …

Each method has merits and drawbacks. Example: a set of random inputs to a divider may be unlikely to test the ‘divide by zero’ case. Random inputs could be biased so that (for example) small numbers are more likely if a block which counts leading zeros is under test.

Data timing

Timing checks require some more detail in the model. Logic timings are difficult to estimate at an early simulation stage; however there are some timing checks which it may be worth modelling early.

A good example is reading a memory. On the laboratory board there is external SRAM to provide capacity for the frame buffer. This SRAM has an access time of 55 ns – which is more than one clock period (40 ns). A simple HDL RAM model might supply data as soon as the address was valid. In this case the data could be latched (accidentally) after one cycle and appear to work.

Even more worryingly, the 55 ns is a worst-case time; a real SRAM device may meet the 40 ns deadline … under some conditions. This could lead to many field failures, e.g. when the weather warms up or the battery is below half charge.

A more sophisticated model, which forces data ‘unknown’ until the relevant time has elapsed, would expose this fault because the ‘unknown’s would be latched and propagated.
Test Coverage

- How do you know what you have (haven’t) tested?
- Wouldn’t it be nice if there was CAD assistance?

These days there are test coverage tools which assist with this task.

Test coverage may indicate such things as:

- which lines of a HDL were executed at least once
- which decisions were 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 about

- e.g. missing ‘default’ in a case with all cases specified explicitly
  - except, perhaps the ‘unknown’ case!

Works on the ‘better safe than sorry’ principle!

Another possible visualisation of FSMs in the Cadence ICCR tool is a view of the state diagram. This can reveal not only the states which have (not) been visited but also which transitions have (not) been taken. The latter duplicates some of the ‘block’ coverage in that, if all blocks have been executed all possible transitions must have happened. However it does provide a (useful?) alternative view.

What to look for: Timing

Although it may not be possible practical to predict units’ timing relationships, even on a cycle-by-cycle basis, it may be useful to monitor some sequence properties for verification.

Examples:

- Is the correct number of operations performed?
  - e.g. when drawing is the number of pixels plotted correct?
- Is something happening?
  - Is a timeout detector appropriate?
- ...

Test coverage ‘failures’

There are some things which it cannot reveal:

- Things you forgot to put in
- Scheduling issues
- Some consequences of internal state (see example below)
- Implementation: the nodes which have toggled are bits as specified in the source language; it is likely that a gate-level implementation will introduce other wires not ‘visible’ yet.

Here is an example of an inadequate test applied to two different circuits.

```verilog
// OR gate
assign q = a || b;

// Transparent latch
always @ (a, b) if (!a) q = b;

initial // Test
begin
  a = 0; b = 0;
  #100 b = 1;
  #100 a = 1;
  #100 b = 0;
  #100 a = 0;
  #100 $stop;
end
```

In either case all the statements have been executed and all the nodes toggled.

“100% test coverage”: ‘same function’; two different circuits.
How to stress things

There are various approaches to designing 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**
  - 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 (e.g. targeted state tests with random data.)

Black-box testing. Tests 'in ignorance' of the implementation of the unit. The tester knows the interface and the intended function and attempts to find faults purely by applying input sequences.

White-box testing. The tests are designed by looking at the implementation and intentionally stressing parts of its design. This is also known as 'glass-box testing' and by other names.

Both have advantages and disadvantages. A black-box test may fail to identify some special case which has to be 'trapped' in the implementation. On the other hand if the code (or schematic) is available it may influence the test strategy; verifying all the code may suggest that the system works but what about the cases the designer didn’t think of? It’s all too easy to follow an existing plan.

Specific tests

The tester inputs some (probably simple) values to check the expected function of the system. For example a 'black box' test of a division unit could have inputs which are positive, negative and zero. [What is the defined behaviour of a division by zero? What about 0 ÷ 0?]

Algorithmic test cases

Having decided on a particular test is should be fairly easy to repeat it, with some variation, a number of times. Such tests can be programmed quite easily in Verilog. In some cases it may be feasible to test exhaustively; in other cases this would be prohibitive and some sample data may be chosen which can be supplied by a simple algorithm. Try to design this so it is likely to cover different test cases.

Random tests

So called ‘Monte Carlo’ testing applies random input patterns. This is often a cheap way of generating wide test coverage and has an advantage in that it may ‘think of’ cases a human wouldn’t. However it offers no guarantees of complete coverage.

The distribution of the random numbers may also be important:

Adder: testing with linearly distributed random inputs will exercise most functionality quite quickly.

‘CLZ’: linearly distributed random inputs unlikely to find most output patterns.

The ARM instruction ‘CLZ’ (Count Leading Zeros) takes a single 32-bit input and returns a value 0-32 which is the number of contiguous 0 bits starting from the most significant bit. [Note: this is easy to do in hardware but would be tedious if it had to be done in software.]

Assuming a linear distribution of random inputs, half should give the answer ‘0’, a quarter ‘1’, an eightieth ‘2’ etc. For each input iteration there is only a 1 in 2^32 chance that the answer 32 would be obtained – incidentally the only output pattern producing a ‘1’ in bit 5 of the result.

Random patterns generated logarithmically could change this chance to (say) 1 in 33 giving better test coverage with far fewer iterations.

State

If a module has internal state – and many do – this state also forms part of the input. Thus it may be necessary to test the unit with a particular set of inputs in each of its internal states. This can greatly enlarge the test set!

To ensure(?) that the unit is in the correct state may require a deliberate test sequence beforehand. Thus the above categories should not be seen as exclusive: e.g. a particular set of inputs could be used to precede a random input set.

As the design size increases (i.e. more modules are integrated) control of internal states becomes increasingly hard. Thus confidence in each unit of the hierarchy is essential.
Ordering and Scheduling

Consider your 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 …
Is likely to be executed by routine tests.

One day, Bob turns up early. Later you think “Where’s Alice?”

- Test coverage can’t help because it can only do what you asked it to.
- Here a test strategy should strive to test every case in every order.

---

**Ordering**

It is likely that a hardware system has a number of subsystems operating in parallel.

- Some cases these will be strictly synchronised
  - example: processing pipeline
  - this is quite easy to test

- Some cases have simple timing relationships
  - example: simple router
  - needs a couple of test cases

- Some have complex behaviours
  - example: FIFO buffer
  - requires numerous tests

---

**Example: FIFO Test Set**

To take the last example, let’s look at the possible different operating conditions of the FIFO. On any particular cycle the following may happen:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>State</th>
<th>Input action</th>
<th>Output action</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Don’t care</td>
<td>None</td>
<td>None</td>
<td>+0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Empty</td>
<td>Input</td>
<td>None</td>
<td>+1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Part full</td>
<td>Input</td>
<td>None</td>
<td>+1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Full</td>
<td>Wait</td>
<td>None</td>
<td>+0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Empty</td>
<td>None</td>
<td>Wait</td>
<td>-0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Part full</td>
<td>None</td>
<td>Output</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Full</td>
<td>None</td>
<td>Output</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Empty</td>
<td>Input</td>
<td>Output</td>
<td>+0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Part full</td>
<td>Input</td>
<td>Output</td>
<td>+0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Full</td>
<td>Input</td>
<td>Output</td>
<td>+0</td>
</tr>
</tbody>
</table>

A complete test should exercise all these cases.

This specification includes the ability to ‘forward’ incoming data in the same cycle if the FIFO is empty and to accept an input to a full FIFO on the same cycle as data is removed. These features (italicised in the table) minimise latency at the cost of increased complexity in the FIFO design.

This is not the only possible behaviour of a FIFO. A simpler FIFO would have a slightly different behaviour. It may be the case that different behaviours would be acceptable within the system and so the FIFO tests could be relaxed accordingly.
Techniques for finding faults

- Improve readability of trace
  - Choose a ‘simple’ clock frequency (e.g. ‘1 MHz’) if no real-time considerations
  - Add some artificial delays to signal changes
  - Have test harness signal raised in trace on error
    - Most waveform viewers allow searches for particular signals changing
  - Keep ‘phase’ variable in trace – easy to see what was being tested

- Export diagnostic information
  - Print $time with error reports
  - Print statements as simulation advances
  - Maybe $stop; after an error occurs …
  - … or stop reporting certain errors after N occurrences of the same thing

- Test the tests
  - Induce failures to make sure the tests find them!

Debugging

Every engineer has ideas on ‘the best’ way to find faults. We are not here to tell you how to do it. Here are some tips and ideas on some things which might be useful.

The two readily available methods for diagnosing simulations are using a waveform view of selected signals and printing out diagnostic values.

The first is most useful when first debugging a unit. Errors are expected and the desired diagnostic information is unclear. It gives a more interactive debugging route. The second method is to check and print diagnostics into a log file. This is most useful when tests are run repeatedly (such as checking to see that a fault has not been introduced) and automation is more important than interactivity.

Waveform trace

In small designs all the signals can be traced. In large designs and long simulations keeping all the traces may not be feasible because it slows down the run and can fill the filestore quite quickly.

- Choose some key nodes which will give plenty of information.
  - These are often near the top level of the hierarchy although the control state of various FSMs can be very useful.
  - As far as possible, keep signal names the same throughout the hierarchy.
  - Name (major) instances to simplify browsing.

These are points for the design rather than the test phase but can prove big time savers when diagnosing problems. Having a few dozen different buses all called “data” on the display is not conducive to clarity!

- Mark points of failure within the run.

The simplest method is to stop just after a fault is found. Don’t stop immediately – it’s useful to have a bit more trace to see the faulty values. This makes the fault easy to find.

Alternatively, a signal in the test harness can be switched or pulsed to flag up the position of an error.

Most waveform viewers will allow searches for transitions on selected signals. By creating a test variable which indicates a fault condition it is easy to locate a point of failure in a large trace.

A related technique is to use a test state variable to indicate which phase of a long test is being executed.

```
$display("Resetting");
test_state = 0;
...
$display("Phase 1");
test_state = test_state + 1;
...
```

This is useful as a test file grows to cover a large number of different tests.

- Keep a test state variable to show phases of a long test.

A variable in the test harness which changes according to which test is current is useful for navigating through long traces. This is particularly useful when revisiting a waveform trace after a ‘new’ problem has been found.

Log file

Commands such as $display and $monitor allow progress through a test to be checked.

- A few judicious printed strings help follow progress through a test.
- Including the time can help if returning to a trace to diagnose a fault.
- Printing ‘okay’ at the end (if appropriate!) is reassuring.

At this stage it is likely that you want to know that an error has occurred but little else. The sort of thing to look for may be an expected and an obtained value, and printing both in the case of a mismatch may be useful. A printed heading will indicate which test failed.

- Use experience gained in debugging to set up appropriate tests.

Finally …

As in software, conditional compilation (‘ifdef) may be used to embed debugging information in synthesizable blocks yet exclude it from synthesis.
‘Cosmetic’ delays

In a simulation there is no need for time to pass for causality.
In the ‘real world’ things take finite times to happen.

always @ (posedge clk)
begin
    data1 <= value;
    data2 <= data1;
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

Two timing effects are shown on the right-hand side of the slide:

❏ The finite edge speed of the transitions
❏ The propagation delay of the circuits

Both of these are governed by electrical effects in the final circuit implementa-
tion which are unknown at this time. Furthermore, in a digital simulation, states
switch instantaneously so there can be no direct representation of edge speed.

However it is possible to represent delays within the circuit artificially. This can
improve legibility of a waveform file by delaying register changes until after the
clock edge.

At its simplest:

always @ (posedge clk)
begin
   #2
   data1 <= value;
   data2 <= data1;
end

This can make the values on the buses at the clock edge clearer. A delay which
is a ‘small fraction’ of the clock cycle probably gives the clearest result.

Delays are not synthesizable; they will be ignored when the circuit is built.

Delays can also be used in combinatorial blocks if an estimate of the perform-
ance of an assembly is available.

Another use for delays is in providing a more realistic simulation environment.
This has been done in the laboratory when the framestore RAM is modelled.
There is a delay in providing read data based on the manufacturer’s (worst case)
figures and the RAM output is undefined until that time. This time is greater
than a clock period. A naive model could provide data as soon as it was
requested; this could then be used in the next clock cycle and everything would,
apparently, work. However if this is attempted with the delay in place an unde-
fined value will be propagated which should be detected further on in the simu-
lation.

Unlike a uninitialised flip-flop, this is a situation where the test code can actively
use unknown values to help find problems.

always @ (data_valid)
begin
   #1
   if (data_valid)
      fs_rdata <= {out_reg3, out_reg2, out_reg1, out_reg0};
   else
      fs_rdata <= 32’hxxxx;
end

[From RAM model used in phase 3 lab.]
Using tasks

- Reminder: a ‘task’ is a method/procedure/subroutine
  - designed for repeated invocation
  - ideal for (e.g.) sending test data

Here is an example task which executes a communication handshake:

```verilog
task strobe;
begin
  req = 1;
  while (ack == 0) #`PERIOD; // Wait for acknowledgement
  req = 0;
  while (busy == 1) #`PERIOD; // Wait for operation to complete
  #(10 * `PERIOD); // Pause before subsequent command
end
endtask
```

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

```verilog
strobe(100, 200, 300, 400, 'h34);
strobe(333, 222, 111, 420, 'RED);
...
```

More about tasks

- When a task is invoked (“enabled” in Verilog jargon) control is passed to it until it completes; then it returns.
- Tasks may contain delays so enabling a task may cause a delay to the enabling process.
- Arguments may be of type input, output or inout.
  - inputs may be expressions.
  - outputs and inouts must be assignable (i.e. like the LHS of an assignment).
- Any number of arguments may be passed.
- Tasks can be nested.
  - Tasks can call functions
  - Functions can call functions
  - Functions cannot call tasks
- Tasks are static by default
  - If a task is to be recursive it must be made dynamic by declaring it as “automatic”:
    ```verilog
task automatic my_task;
```

System tasks

Commands starting with ‘$’ (such as $display) are system tasks. These are built in to allow access to facilities such as files. A selection of these has been introduced already.

It is possible to add user-defined system tasks for interaction with other parts of the external environment. We will return to this later, including using some in the lab.
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 various stages of development:

- Design review
  - have all eventualities been thought 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?

Reviewing can seem like an informal process. Unlike, for example, a compiler’s syntax checking it does not guarantee to find every fault (of a certain class) that may be present. Nevertheless it can be a very valuable process. Being a human process the ‘terms of reference’ are not strictly limited and all sorts of things can turn up.

The basic principle is to get other engineers to look at a design (code etc.) and for them to try and spot things the designer hadn’t thought of.

Suggested review process:

- The designer(s) invite a small group of colleagues to perform the review. Ideally reviewers should have some background knowledge of the project but not be ‘too close’ to the topic of the review.
- Designers prepare copies of any background documentation (such as code listings and diagrams).
- Hold a candid meeting where everyone is prepared to raise issues without embarrassment. Some mistakes may be revealed; some ‘dumb’ questions will be asked; that is okay.
- Each designer ‘walks through’ their product in detail. For example, for code this will usually involve looking at each statement individually. Each item should be explained so that everyone is clear about its purpose.
- Reviewers try to think of cases that are not covered and may cause a malfunction. E.g. “What if this value is negative?”, “Did you miss out a ‘default’ here?”, etc.
- Often, simply the act of trying to present a system to others causes the designer to be the first to spot problem cases.
- Take notes, so the problems will be remembered. Annotating code listings is typically convenient.
- Thank the reviewers.
- Revise the design according to the suggestions.
- Re-review, concentrating on the altered areas.

Classic fault in C: the sort of thing that it’s difficult to spot

```c
if (x = 1) …
```

- Legal syntax
- May do what you expected …
  - … by chance …
  - … unless you make a point of testing the case when it shouldn’t happen
Regression tests

Test process
- identify something which needs testing
- devise a test
- test
- iterate until the design passes

Then
- keep the test
- continue with development
- rerun the test
- iterate until the design is complete

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

Regression tests

“Primum non nocere”
[“First do no harm.”]

Regression tests are tests which ensure something hasn’t been broken by later changes. These changes can occur for various reasons, such as:
- a ‘new’ fault was found later in development
- an upgraded version of the system is being produced

An example of the latter case: imagine you are the Intel Corporation. Most of your business is founded on processors which are backwards compatible with earlier generations so it’s vital that new devices can still run old code. You (presumably) therefore keep a suite of old software – contrived to test every aspect you have thought of – to test compatibility of your latest device.

Testing can be boring, especially if you expect everything to pass. This should normally be the case following a minor edit to an apparently unrelated block. If such retesting relies on human attention then it probably isn’t going to be very reliable.

Regression tests should be automated, as much as possible. It is easy to see how this can be done in many cases. For example a processor can be given code which is self-checking; run it and it says ‘okay’ (or not). It’s a bit harder in some systems, but an appropriate test harness can do a similar job. As machines do not get bored then this is a good solution and all you have to do is run it.

Of course these tests must be thorough and, occasionally, may highlight a problem. It is therefore useful if they contain some diagnostic information as well. It would be intimidating to know that there was a fault somewhere in your design but not know where!

The best time to put these tests in place is when the appropriate system is first developed. That is the time when you’re concentrating on what could go wrong and, probably, discovering bizarre behaviour. Having thought of a test, do it tidily and add it to the regression test suite before moving on. It will save time later.

“Quis custodiet ipsos custodes?” Juvenal
[“Who will guard the guards themselves?”]

Are your tests actually working? Sometimes you may get a ‘false positive’ result where they flag up a fault that does not really exist. This may be a genuine mistake; it may be over-zealous checking of values which don’t really matter. Whatever the cause it should draw attention and be fixed.

More worrying is the ‘false negative’ where a test misses a genuine problem. This could be something you didn’t think of – you can’t do much about that, you need independent reviewers to help – but it may be something you thought you tested for but there is a bug in the test suite.

Tests are normally augmented but not changed. Thus the tests should also be debugged when they are produced. To exercise tests there is a need to introduce deliberate faults into (copies of) the system and demonstrate that the tests pick these up satisfactorily.