Verification and Test

In the absence of perfect design …

… and proven tools …

… and proven designers …

… testing is vital!

What is "testing"?

Here, divide into:

- verification – have you made what you intended?
- validation – does your design fulfil the requirements?
- testing – does this particular part work?

The first two (should) also apply to software; the last is restricted to 'physical stuff'

- In hardware development the checking is done by **simulation** rather that prototyping.
  - Making a chip is expensive!
  - Want every one to be right first time!

Simulate early, and often

Customers are only interested in products that work. Thus whatever we are trying to build needs to be designed in order to work and then built successfully.

Given that, the next incentive is to achieve this at minimum cost.

Eliminating errors early is a Good Idea because continuing to build something which will not work is a waste of effort. Design and production goes through a number of stages:

Modelling

Early in the lifetime of a product there is likely to be some 'design space exploration' – a feasibility study and an attempt to find a good (the 'best'?) way of solving the problem. This is likely to use some high-level **behavioural model**. This does not represent a particular detailed implementation although it may parallel the way blocks work at some abstract level.

With the increasing prevalence of SoCs this has become common in **Transaction-Level Modelling** (**TLM**) where subsystems interact by passing 'messages'.

Simulation

'Simulation' could apply to many abstraction levels; here we'll stick to **Register Transfer Level** (**RTL**). An RTL description will fix much of the implementation; it will exhibit both the logical functionality and the timing to cycle level, and possibly further.

An RTL simulation may be used to characterise a behavioural model, i.e. get its timing more accurate.

Testing

At the final stage it is important to filter out as many production defects as possible. This can be time-consuming when a system is very complex, simply because there is so much to do. There are techniques such as self-test which can be used to simplify (speed up) the testing process.

Regression Testing

Tests should be developed in parallel with the code, and kept. As a project progresses the test suite should become a powerful aid to keeping things working.

Most designers are used to running a few tests on each new piece of code to increase confidence that it works. Unfortunately it is often 'too much trouble' to keep these in a state where they can be used again. Frequently a piece of test code is *modified* such that it serves some new test *instead* of appending further tests.

Regression tests are tests which (attempt to) ensure that a developing design stays working. Thus, when changes have been made the tests can be run again to demonstrate that something which had previously passed will still do so. As these should be run quite regularly during development it is a good idea to have them fairly automated, such that inconsistencies are detected automatically and reported with a useful error message. That way, after benign editing, all that should be needed is a test run and a check that it completed okay.

Designing tests to do this is not wholly trivial. The tests should accommodate all legal behaviour rather than just the initial design; for example the number of cycles taken to complete an operation may change (hopefully reduce!) during development so a result should be examined when it is indicated valid, not at a preset time determined by the first implementation.

With a little trouble a good set of regression tests can make development and maintenance much ~~easier~~ more reliable.

Check the results … automatically

Note that a test environment is not just for applying inputs; it can compare the outputs with expected values too. Although some tests (and much debugging) are done by human examination of the output this is not a good plan for large test suites, being tedious and error prone.

Leave the big, boring jobs to machines; that's what they're good at.

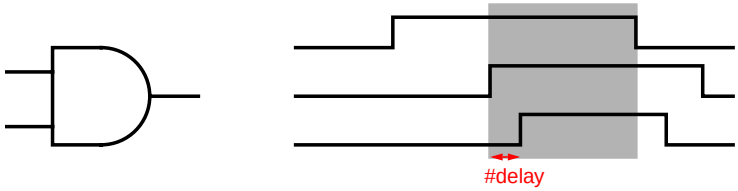
Verification

Want to ‘prove’ the logic or implementation works with as little effort as necessary.

Use different techniques for different purposes.

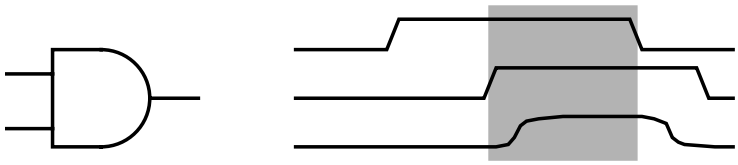
Digital simulation

- verifies logic
- reveals initialisation problems
- no (or approximate) timing
- quick



Analogue simulation

- verifies timing (and logic)
- reveals edge speeds
- slow



Digital simulation

Digital simulation – such as you use in the labs. – provides a fast way to check the logic in a circuit. Digital simulation represents signals as discrete levels {0, 1} and other states (such as X ‘unknown’). Switching between states happens instantaneously, even if a delay is inserted by the gate.

This sort of simulation is sometimes called ‘discrete event simulation’ for this reason. A big (speed) advantage is that if nothing is switching no work needs to be done.

One useful side-effect of digital simulation is that it can have ‘unknown’ states which can reveal potential problems with flip-flops not being initialised.

Examples of digital circuit simulation include:

RTL (behavioural) simulation

The type of simulation you will (should?) be familiar with from the labs. All the logical wires are modelled and the gate-level function can be checked. There is no inherent timing model; approximate timings can be added but the accuracy depends on human calibration.

Such simulation can be applied to HDL source code or synthesised/hand-drawn schematics.

High-level modelling

Digital models which connect functional ‘black boxes’ allow architectural exploration and confirmation of basic algorithms without the need to do detailed RTL design. These are faster to create than RTL models and, because they omit implementation details, will run an order of magnitude faster. However the lack of detail means that even if tools to compile them to hardware exist the result may be inefficient.

Analogue simulation

The physical world is not digital – at least at the scales where VLSI is concerned. The circuits do not have only discrete states and changes are not instantaneous. If a more accurate model of a circuit is required – for example to measure timing characteristics – then a more detailed model must be used. These model the voltages on each wire (including inside gates) continuously using the transistor characteristics, wire capacitances, temperature etc.

Necessarily these models are slower than digital simulation.

Because each node has an analogue voltage there are no ‘unknown’ states. A simulator will initialise the system starting with some assumption (e.g. all nodes are at 0 V) and iterating until some stable state appears. This means that the initial state may not be the hardware starting state *and you may not notice*!

Transistor-/gate-level simulation

Simulators are available that give reasonable analogue approximations to the behaviour of the real circuits by modelling the voltages in short steps. This gives a piece-wise linear approximation to the analogue curves (i.e. the curves are approximated by a number of lines). Timing accuracy can be within 10% of reality which is adequate for most purposes.

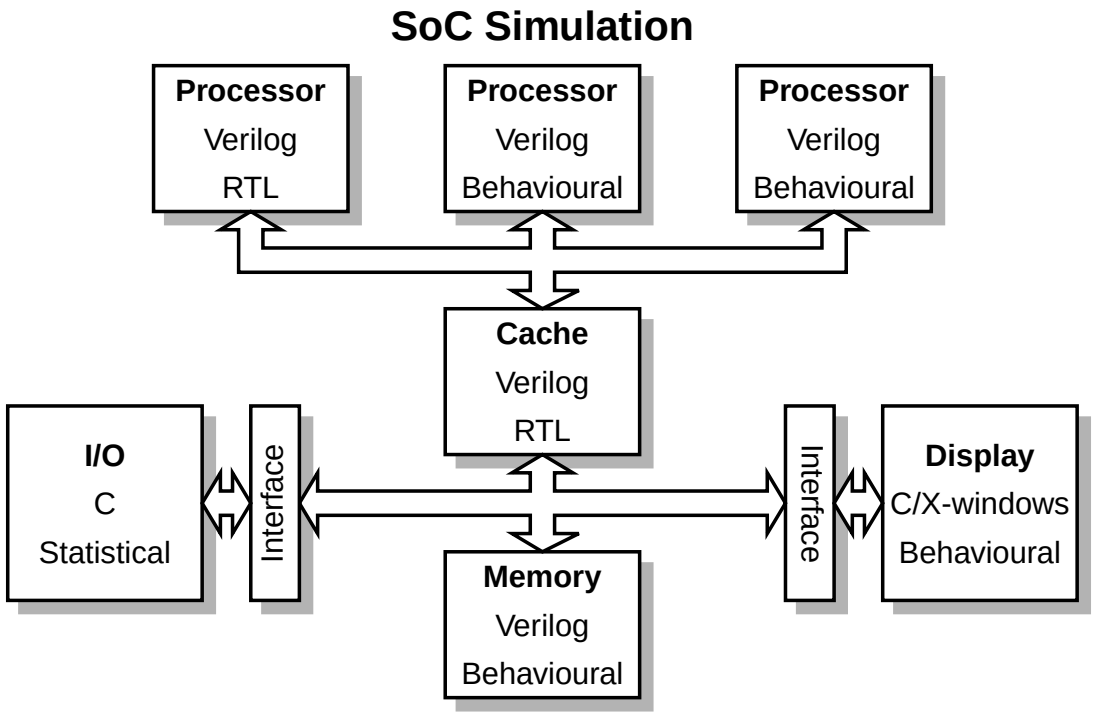
To do this requires more calculation (and more memory on the simulator) and is thus maybe an order of magnitude (or more) slower than a digital RTL simulation.

SPICE (Simulation Program with Integrated Circuit Emphasis)

SPICE and similar tools model the differential equations which characterise each component of the system. A SPICE model should be a very close representation of what is built. SPICE will be an order of magnitude slower than a transistor-level simulation and even more compute-hungry.

SPICE is used to characterise small circuits (such as gates) for calibrating transistor-level models. For example, all the components in a standard cell library will have been characterised with a SPICE(-like) tool by the silicon foundry.

It is also an essential tool when building analogue circuits.



- Trade off simulation accuracy with speed
  - Memory requirement (of simulator) may be a big factor in speed of simulation

Miscellany

Hardware emulators

Programmable devices are now so large that they can be used for simulation purposes. It is possible to synthesize large logic subsystems and load them into an FPGA. Given a 'box' with a lot of big FPGAs it may be possible to partition and load a whole chip description. This is sometimes referred to as '**hardware emulation**'.

Such an emulation may provide access to signals which will be 'buried' in the real chip but which may be useful for debug purposes; however this is not as flexible as a software simulation which can easily monitor many nodes.

Some properties of hardware emulation are compared below:

Software simulation	Hardware emulation
Slow; a bit faster on 'cluster' machines.	Considerably faster; fast enough to boot operating systems etc.
Extensively controllable; can set breakpoints etc.	Basically just runs!
Offers good observability of all signals	Limited observability; may need logic analysers.
Can show 'unknown' (X) nodes.	No concept of 'unknown'.
Approximate or no timing model.	Timing model will not be applicable to ASIC.
Reasonably affordable.	Significantly expensive.

Some of the issues in controllability and observability are addressed in some hardware emulators. It is still clear that they only have one big advantage over software simulation and that is speed. However the ability to exercise orders-of-magnitude more cycles is a very compelling argument, especially as it saves development time and increases confidence that the final chip will work.

Monte Carlo simulation

A set of test patterns ('**test vectors**') is unlikely to test any substantive design *exhaustively.* It is therefore necessary to choose representative sample tests to find any errors. One method of doing this is for the designer to tailor cases to try and exercise all the control paths (at least). This is usually effective in covering behaviour, although humans are fallible and some cases can be missed.

So called 'Monte Carlo' simulation is a complementary technique which simply uses random inputs. It has a couple of advantages in that it is very cheap to design ('give me 100 random input patterns') and it may produce cases that a human would miss. It's chief disadvantage is in test coverage (would 100 random integer divisions test the divide-by-zero case? Unlikely!).

Reviewing

Reviews are performed by (honestly) exposing designs to other designers who will try and identify potential problems. This may sound 'unscientific' (it probably is!) but it does work. Human brains tend to 'lock' onto ideas and it sometimes takes an independent viewpoint to spot 'obvious' shortcomings.

Frequently, simply the act of explaining a design to others will highlight potential problems.

Unfortunately this may be the only feasible way of checking some very complex interactions as setting up simulations of 'all cases' would be impractical. This is more likely to occur with software than hardware where the complexity (in terms of the possible number of states a system could reach) may be huge.

Reviewing gives no formal proof or such, however it does find faults.

Formal methods

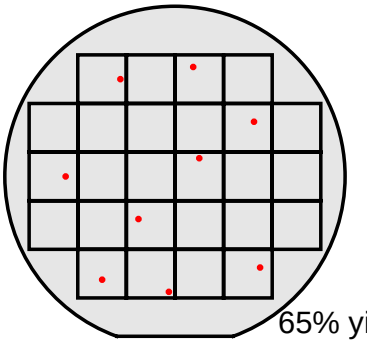
There is much interest in applying formal proofs to hardware development and *knowing* that something works (or even that two descriptions were identical) would be very valuable. To date the scale of the systems involved has outstripped the capacity of automated tools.

Development progresses; hopefully more tools and methods will emerge.

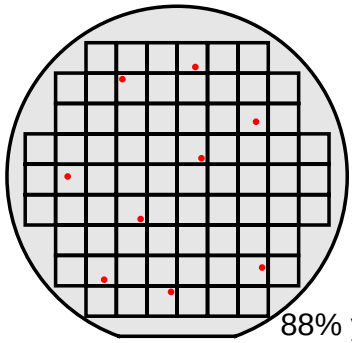
Testing

- VLSI chips are made on silicon wafers.
- Despite taking great care, defects occur: thus some chips will not work.
- The proportion of working chips in a batch is known as the "**yield**".
  - Yields vary but on a mature process may expect (say) 80%+
  - Yield will be affected by chip area: bigger chips are more likely to be faulty

Not to scale



65% yield



88% yield

Testing determines which chips work and which don't.

When a chip comes off the production line it may or may not work. The choice of what/when/how much to test is an economic question. Most equipment used in chip manufacture is very expensive to buy and run (think 'five figures' *per hour*) and time on the tester is no exception.

- Test patterns should cover as much as possible as quickly as possible

Note: whilst testing logic functions may be quite straightforward(?!) parametric tests – checking the speed or measuring power supply currents at each stage – requires equipment which as at least as fast as the **Device Under Test** (**DUT**).

Yield and area

The number of defects is on a wafer 'small' and they are scattered randomly. Thus, roughly, expect at most one defect/chip. This implies the number of defects dictates the number of non-working chips.

The total number of chips is (roughly) the wafer area divided by the chip area: smaller chips means more chips. The wafer cost is the same regardless of the number of devices. Thus smaller are both cheaper initially and more likely to give higher yields.

False positives and false negatives

There are four possible outcomes from a test process:

Chip	Test	Outcome
Works	Passes	Correct
Works	Fails	False negative
Broken	Passes	False positive
Broken	Fails	Correct

A false negative means a good chip may be put into the bin: costly!
A false positive means a broken chip may be put into service: more costly?
A test should get the **right answer** as often as possible!

This means maximising **test coverage**.

Memory tests

Memory may occupy a significant proportion of an SoC. RAM is relatively easy to test because of its regular structure. It is easy to ensure that each bit can store both a '0' and a '1' and quite easy to look for bridging faults by varying patterns, knowing that it has a regular array structure.

There are some 'don't' when devising a RAM test though:

- Don't write a pattern and then read it back immediately.
  Even if disconnected wires will retain values for some time.
  Perform a set of writes then a set of reads.
- Don't use the same pattern in every location.
  Check that each location can hold something different so you know that you can talk to more than one word.
- Don't use a too-obvious pattern, such as the address.
  You could just be reading the address back again.
  All the bits in the word should vary.

Better to devise some reproducible data sequence – such as a hash function – which changes all the bits in a fashion not-obviously-related to the addresses.

Production test

Test equipment expensive - especially if looking for timing on I/O pins etc.

- Probe test
  - Test the unpackaged chip to avoid wasting packaging costs
  - Requires time and jig
- Package test
  - Easier to perform
  - Discards more expensive
  - Necessary: chip may be damaged during packaging

Occasionally the whole wafer processing may go wrong.

- Foundry will add test sites to wafer (spaced across surface)
- Parametric tests made at different stages of manufacture
- Wafer abandoned if things go badly wrong

Types of faults

Faults can be just about anything! It's impossible to predict everything that might fail. However there are classes of faults which serve as an illustration:

Logical faults

Stuck-at faults

A stuck-at fault is one where a signal wire always has the same (digital) value, either '0' or '1'. These can be relatively easy to find because a pattern can rely on a wire being in each state and, if it isn't, some observable effect occurs.

Bridging faults

Two signals are joined, thus if both should be the same value they appear okay but if they should differ then one will be wrong. These can be hard to find: in principle there are combinations of states and inputs which can cause each pair of wires to be different (unless they are redundant!) but the number of possible patterns grows factorially with the number of signals so it is practically impossible. In principle it is possible to reduce the problem by only considering signals which are, at some point, on adjacent wires but this involves detailed knowledge of the physical layout. Often it is simply assumed

Open circuits

A broken wire will lead to one or more signal inputs 'floating'. An unknown input will *probably* reveal itself as a wrong state but it can change at random.

A floating input *may* be detected because it leaves a transistor partially 'on' by **Iddq testing**. Basically this involves measuring the supply current in different (static) states and checking if it's above the expected norm.

Parametric faults

Physical devices are subject to manufacturing variation so the 'same' transistor on every chip will be slightly different. This means that the timings as simulated are 'best guess' values and, in practice, there will be variation.

If made as expected then variation is expected and allowed for. However it is possible for a device to be 'weak' so that it just passes logical testing but may fail later – say at a higher temperature

Test Coverage

Test coverage refers to the proportion of the design which is tested.

The requirements differ at different stages of the design and production.

Design-time

- At RTL the designer should endeavour to test all the source code
  - Execute every line of code
  - Take every alternative route (if … else …, case …)
  - Not necessary to try 'every' data pattern (a '+' is going to add all its bits)

Circuit level

- The circuit is synthesized from the description.
- There is not a one-to-one correspondence of wires and HDL variables …

Production

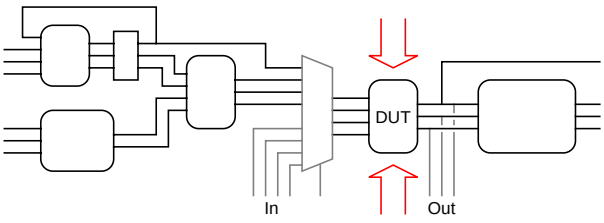
- The *design* is believed to work but any component could be broken
- Need to ensure every wire can change logical state
- Checking for 'bridges' (etc.) is very hard – layout dependent

Fortunately CAD tools are available to assist with much of this.

Design for Test (DfT)

There are two key words in designing a system so it can be tested easily:

- **Controllability**
  If a circuit is deeply 'buried' it can be difficult to get its inputs into the required states. Making it controllable means that it is easy to inject test patterns to the locality.
  This may be as simple as embedding a multiplexer in front of the block with a switch to take patterns from an easily programmed source.



- **Observability**
  The outputs of the Device Under Test also need to be visible. It is sometimes easier to bring them to a test port than to try and deduce their states by looking at subsequent circuits.

Inserting extra logic/loads this way will have an adverse effect on performance but it can often be worthwhile.

Example: A peripheral register may only *need* to output values. *Allowing* a processor to read it back adds some logic but means that the register function (if not the subsequent peripheral) is easily observable.

Example: frequently there is a 'spare' input (or two) left on a chip which switches its pads to different functions allowing them to be test I/O. These are not usually documented for the user: the test control inputs will simply be labelled as (e.g.) 'must-be-zero' (for normal use).

Test port

A system may be based around a multi-master bus. It is possible to include an external bus master which then allows test equipment access to memories etc. in the chip. This can be used for writing and reading locations to verify their existence and behaviour.

BIST

Built-In Self-Test is a means of reducing external complexity by having a chip test itself. This may require the addition of some extra subcircuits.

One method to do this involves applying patterns to logic and checking their results. A simple (but area-expensive) method would be to use a ROM to generate inputs and another ROM to check outputs against.

A smaller checker can be built by applying pseudorandom inputs and accumulating the outputs via a CRC. This can be done with linear feedback shift registers (LFSRs); the cost is quite low but the test coverage may not be as 'planned'.

Software

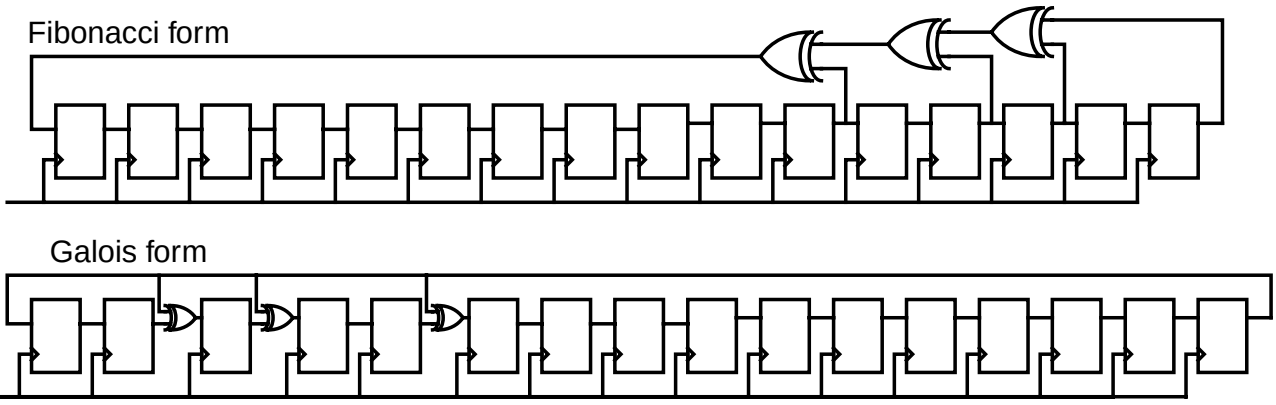
Many (most?) SoCs will now contain one or more processors. These are well suited to running tests on anything they have (easy) access to, such as memories and peripheral registers. They can also self-test their own operation at the same time.

If there is enough ROM space then the test software can be left on-board, possibly accessed by booting with a reserved input signal via a test pin. Alternatively software could be downloaded by the test equipment (although this takes longer).

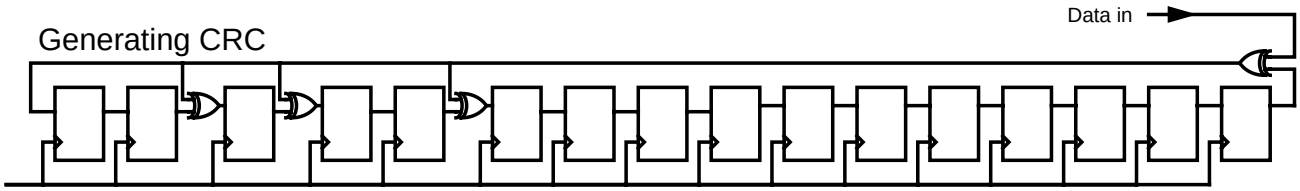
There needs to be some reliable method of signalling a test pass such as a particular output pattern or a signature in accessible RAM; the absence of a pass after the appropriate time is assumed to be a failure. The pass signature should be unlikely to appear by accident.

Linear Feedback Shift Registers

These can be made in different ways



… and applied



More parallel implementations are possible (at a cost) to accommodate multiple bits/cycle

LFSRs

The requirement

The requirement for a test of a block of logic is usually a *representative* set of input patterns checked against their expected output patterns. This could be done by precalculating and storing the corresponding input and output sets and applying them during the test. There are a couple of drawbacks to this.

- Size: the patterns would require ROMs which may be quite large
- Reliability: chip faults are random in area; a large test circuit makes a fault in the test logic more likely; discarding a working chip because its test logic is faulty is uneconomic

A solution

Generate a predetermined sequence of test patterns using a 'counter'. This saves storing the input patterns.

'Checksum' the output patterns to give a single 'signature' word. This saves storing the output patterns, except for a single result.

A simple counter will not usually provide good test coverage with a 'small' sequence of inputs. A better mechanism is to provide 'random' inputs; as these need to be a repeatable sequence this is really a pseudo-random sequence.

LFSRs are small, fast, simple(?) circuits which can generate pseudo-random sequences. Each state is calculated from the previous state by shifting and using a few exclusive-OR gates to alter the bits.

The principle can also be used to generate **Cyclic Redundancy Checks** (CRCs).

Other uses

- LFSRs are used extensively in digital communications, including DAB and DVB.
- LFSRs can be used to 'encrypt' data – but not very well!

LFSR design

The choice of the position of the 'taps' in the LFSR is important in setting the length of the pseudo-random sequence before it repeats. Ideally, with N flip-flops, it should be 2^N states long. (In practice only 2^N-1 is achievable because the 'all zeroes' state repeats itself.) Only a subset of the possible arrangements will generate the maximal sequence. These are typically expressed as *polynomials*: the example shown is: $x^{16} + x^{14} + x^{13} + x^{11} + 1$

Suggestion: don't try to memorise data like polynomials – look them up.

There is plenty of (interesting) mathematics behind these circuits but that need not concern us here.

Checksums

A 'checksum' is a hash function of a set of input words. At its simplest it can be (for example) the exclusive-OR of all the words in a block of data, modulo the word length. This type of function is easy to compute in software. Simple checksum algorithms can be prone to particular bit errors; for example in summing all the data if a bit is stuck-at-zero then it could easily escape detection.

Slightly more sophisticated algorithms can improve the protection. For example adding the words (with propagating carries) will help, especially if the overflowing carry is returned to the least significant bit.

A checksum will not find every possible fault but can give reasonable coverage over a range of 'expected' problems. A more sophisticated algorithm – such as a **CRC** – 'jumbles' the bits much more and so gives better coverage form most typical faults.

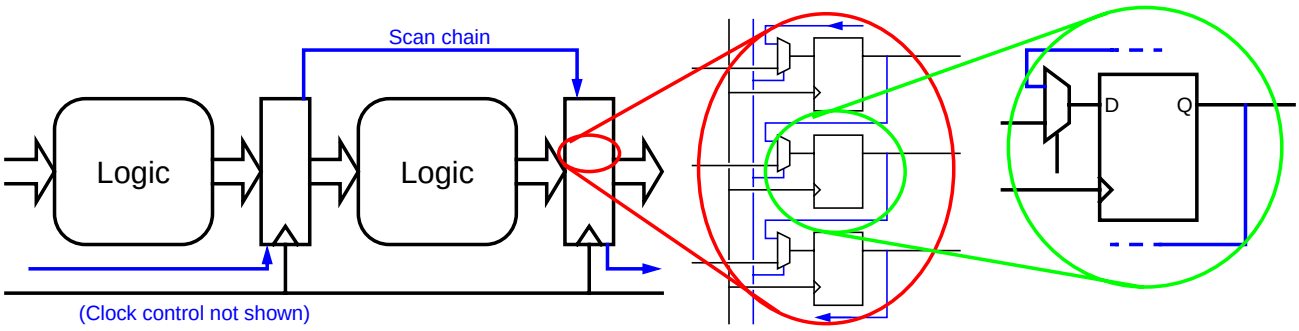
Simple software checksums are frequently applied to on-chip ROM to increase confidence in code and data integrity. Often an extra data word is appended to the block, calculated so that the resultant checksum will be zero; this makes checking slightly easier (no need to modify any *code* when the data changes). Note that 'ROM' can include programmable memories such as Flash memory where there are potential 'wear out' mechanisms.

Scan paths

Some systems divide ‘naturally’

- IP blocks such as processors, peripherals, memories
- Pipeline stages

… or logic can be divided ‘artificially’, with latches



- The latches are made of flip-flops
- ‘Normal’ flip-flops can be replaced (‘automatically’) by scan flip-flops.
- If the clock is stopped the data are retained around each ‘block’
- They can be scanned in/out by turning the registers into a single, large shift register.

Boundary scan

The problem with testing a big circuit is that there are many reachable states.

The complexity grows super-linearly with the size. This could be exponential although in practice it may be closer to the cube of the size. Nevertheless this soon becomes unmanageable.

Partitioning

Apply the usual divide-and-conquer approach. If a block of size N has complexity: N^3 then dividing it in two has a complexity $2 \times \left(\frac{N}{2}\right)^3 = \frac{N^3}{4}$

> Note: the cubic relationship is *illustrative*, not a mathematical rule. (It is reasonably representative.)

In the general case, a large design can be subdivided into many blocks which can be tested separately – and possibly in parallel.

> Caveat: it is not always guaranteed that these blocks will be physically separated on the chip, although it is quite likely.

Scan path

A scan path is a means of improving the testability of a circuit by improving:

- controllability
- observability

by subdividing logic. Using serial access this is quite cheap in terms of I/O connections, especially when performed from outside the chip. It is also very slow when the scan chains get long, of course.

Long scan chains are sometimes subdivided into parallel sections so selected parts can be shifted in fewer clock cycles.

Boundary scan is a scan path that surrounds an IP block. It allows the function to be frozen, the peripheral states to be read out and new states inserted at low cost.

It can be used to check connections from chip to chip across a PCB but a more common use is to examine blocks within a chip.

Debugging aid

In addition to chip test, boundary scan is used for software debug. Consider a case where a processor clock has been halted: the address and data buses can be determined to see what it was doing at the time. Then a sequence like the following can be applied:

- Ignore the address, force the instruction input to a store and clock
- Read the store bus to get that register’s contents
- Repeat until all the state is extracted
- Work out the state before you started; display for the programmer
- Possibly make changes
- Force a load instruction into the processor, provide the appropriate data and clock
- Repeat until the state is restored
- Continue normal execution or single step …

The memory contents may be accessed similarly by performing load cycles.

Although slow in machine terms this allows full debug access deep within a chip.

The usual boundary scan interface conforms to IEEE Std. 1149.1-1990, commonly known as **JTAG** (Joint Test Action Group) and is a five wire serial interface. Because it is serial any number of devices can be ‘daisy chained’ without needing extra pins.

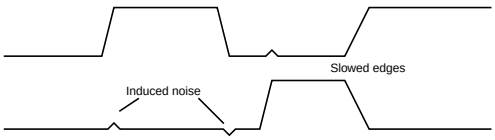
Summary

- Design should be right first time, every time
  - Hardware designers typically come pretty close to this
  - Cost of failure >$1,000,000 and 3 months
- Extensive simulation is used to achieve this
  - Various ‘levels’ of simulation (speed vs. accuracy)
  - Other CAD tools (e.g. timing analysis)
  - This consumes time & money, of course
- Some manufactured chips will not work
  - Identify and discard as soon as economics dictates
  - Add features to make this testing cheaper (faster, simpler)
  - Fault tolerant devices? – may be an issue for the future?

Some other things that can go wrong

Crosstalk

Crosstalk is noise which is induced by one signal in another. Any proximate wires will be ‘linked’ by some capacitance. An edge on one wire will cause the other to move (a little) in the same direction. If the capacitance is large then this noise could be large enough to affect signal integrity. However what is more likely is that it may slow down an edge in the opposite direction on the second wire. This may affect timing assumptions.



To alleviate this it is a good idea to vary which tracks run adjacent to each other when there are long, parallel wires (such as buses). This phenomenon – and ‘cure’ – have been observed since the early days of telegraphs and telephones (hence the name).

Power supply noise

When signals switch they source or sink charge via the power rails. Charge is conducted across the chip via metal tracks - typically much wider tracks than are used for signals. Widening the tracks reduces their resistance (per unit length) and thus reduces the voltage drop. Even so, at a given gate the power supplies will be closer together than the nominal value. It is important to keep this voltage drop within acceptable bounds.

The demand for charge varies as gates switch, so the supply current varies. The sudden demands for charge cause extra, transient drop in the supply voltage (noise). This can be combatted by providing capacitors as ‘reservoirs’ of charge as near to the gates as possible. These are connected between the power rails and are known as ‘decoupling’ (or ‘bypass’) capacitors. They are almost always present external to the chip but some effort may also be made to increase the effect on-chip.

This is a rare case where capacitance is the digital designer’s friend!

Electromagnetic Compatibility (EMC)

Sometimes may be described as Electromagnetic Interference (EMI) or Radio Frequency Interference (RFI).

Switching currents leak energy through the emission of electromagnetic radiation. At the frequencies of digital circuits this tends to concentrate around the UHF radio bands; these bands are used for microwave communication including wireless LANs and mobile ’phones.

There are two aspects to EMC problems:

- radiated energy interfering with other equipment
- passing communications interfering with the chip

The first is ‘controlled’ by various standards imposing limits on the energy radiated.