Lecture 16:

Testing,
Design for Testability

Overview

Reading

W&E 7.1-7.3 - Testing

Introduction

Up to this place in the class we have spent all of time trying to figure out what we want on a chip, how to implement the desired functionality. We also need to figure out a method of testing to see if the chip works after it is manufactured. There are two types of testing a designer is interested in. The first is to check out the design, to make sure it is correct, and implements the desired functionality. We have talked some about this testing already (checking out your verilog code, or using verilog to generate tests for irsim).

Unfortunately the manufacturing process for ICs is not perfect, so we need to check each chip created to see if it matches the original design. Given the complexity of today's chips, this can be a very difficult task, unless some planning is done up front. This planning is called design for testability.

Levels of Specification and Simulation

Design testing uses the different abstraction levels:

- The goal of design is a hierarchy of levels of implementation, where each level is “correct” with respect to the above level of specification.
  - Architectural -> Behavioral -> Logic -> Circuit -> Layout -> Devices
- Each level of implementation has a more detailed behavior than its specification cares about. The specification is a “subset” of its implementation. The specification does not imply a unique implementation.
- The goal of simulation is to make sure that all levels of implementation “agree” on the same results as specified.
  - ProgramExecution -> C-model -> Verilog -> Gates -> Spice -> Trans. Models
- Errors occur when simulation or testing fails to detect that a given level of implementation doesn’t produce the same results as its specification says it should. Always think about comparing between two levels.
- Design errors are corrected by changing the implementation AND the simulation/specification to make sure it gets checked.

Testing Roles

1. Debugging a Design - Did we design it correctly?
   Why doesn't the chip match simulation
   (oops - did not simulate that case. Specification inadequate)
   (oops - design mismatch. Implementation wrong.)
   Designers working with chip trying to locate bugs.
   Want vectors to isolate problems and minimize engineers' time.

2. Acceptance Testing – Does this particular fabricated chip work?
   Pass/Fail result. It is a quality test of the part before it is sold
   Want few, good, vectors, each vector should check many possible faults
   Time is money for this type of testing.

3. Failure Analysis – Why did this part fail?
   Try to improve the test coverage (if faulty part passed our tests)
   Try to improve the chip (if many parts are failing for the same reason.
   Maybe there is a marginal circuit
   Maybe a design rule really needs to be larger

Defects in Fabrication

In the fabrication of chips, there are defects that get introduced. These defects come from many sources:

1. Defects in the base silicon.
2. Dust or contamination in the fab or on wafer.
3. Dust on mask.
4. Misalignment.
5. Over-etch or under-etch.

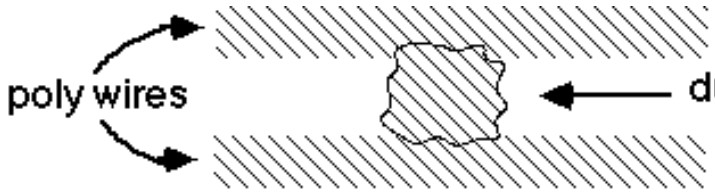
   ...

   → sometimes you don't get the exact pattern you printed.

Thus, sometimes, the implementation of the fabricated die doesn't match its specification, and you get wires you don't want, don't get wires that you want, or find out that the transistors don't operate as you expect

Particle Defects

Particles can disrupt either “light-field” or “dark-field” patterns on the mask or on the wafer.



Shorts

So wires end up shorted together

or

Opens

missing a piece.

Design Errors vs. Production Errors

Design Testing

- When you are checking out your design, all you need to do is test that every cell works, but you don't worry as much about checking that every instance of every cell is working. If register 6 works, register 7 will work too (but you do need to check the decoder.)
- Check every unique cell definition.

Production Testing

- Defects are random, they can occur anywhere on the chip.
- Need to check every unique instance (every transistor and every wire)

Testing

Testing for Design:

- If one register bit works, that cell was designed correctly.

Testing for Production:

- Need to test every bit in the register to make sure they all were fabricated correctly.
- Need some metric to indicate the coverage of the tests. This is usually done by measuring **fault coverage**, which is the percentage of the faults are covered by your tests. **Fault grading** is the process of measuring fault coverage.
- Tests have to be pretty good if you don't want to sell parts that are faulty. The lower the yield, the higher test coverage required to avoid selling a bad part.
- Probability shipping a bad die approximately (1 - C) * B) where C is the test coverage and B is the bad yield, both normalized from 0..1

Chip Yields

New, state of the art huge part	< 10% (maybe 1%)
Mature (old) part	> 50% (maybe 90%)

- Problem is as yields go up, people build larger die, forcing chip yield down. The reason? It makes the system cheaper. The number of good die/wafer need not be large for the newest processor chip, since the price of that chip can be many $100s. By adding more stuff to the die (floating point, larger caches), you make the chip more expensive, but the systems cheaper.

→ Always going to have to test low-yielding parts.

→ Need high fault coverage to guarantee shipping good quality parts.

Design Testing

This type of testing has been the focus of most of this class.

- Done mostly with simulators
- Goal is to exercise the chip through **all** its operations.
- The 'all' part is hard, since
  - It is hard to remember / think up all possible cases
  - Simulators run much slower than the chip
- When chips come out of fabrication and fail in system
  - Check to see if the simulator fails where the real chips fail.
  - If it does, debug the chip in the simulator

    (it is easier to peak and poke at nodes)
  - If it doesn't, you need to probe the chip

    Need to find out why the simulation doesn't match the actual part.
- Goal is to find a chip that works, and keep simulation model consistent

Probing Technology

- Normally you only have access to the pads.
- **Micro-probes** (nano-probes, pico-probes) are all very thin needles manipulated on a wafer probe station that can get be moved under the microscope and dropped down on specific wires. But often the added capacitance disrupts the circuit so much, that no information is gained about what was happening before the probe changed the circuit.
- With an active FET amplifier in the tip, the best micro-probe models can reduce the added capacitance to under 100fF
- Voltage levels can be sensed non-mechanically by **e-beam** testers, which are like sampling digital oscilloscopes that recreate a waveform by repeated sampling at slightly different delay offsets.
- Both microprobing and e-beam probing can usually only get to the top layer of metal. Sometimes it is possible to scrape insulation away, to get to the next lower level of metal if it isn't covered by the top layer. Sometimes laser or ion beam drilling can be used too.

Single-Die Repair Technology

When a problem is discovered, and a correction proposed, it is desirable to check the correction quicker than waiting many weeks for the re-fabrication cycle.

It is possible to make changes to individual dies by using focused ion-beam(FIB) machines. Together with laser cutting, this allows the capability to both **ablate** (remove) and **deposit** (add) wires. But the wires are much lower quality (hundreds of ohm/square), and deposition rates are slow (on the order of a minute per 100 microns).

Much better to get the chips right in the first place!

Production Testing

Once the design is fully correct, production testing verifies correct fabrication for each new die.

But this isn't trivial.

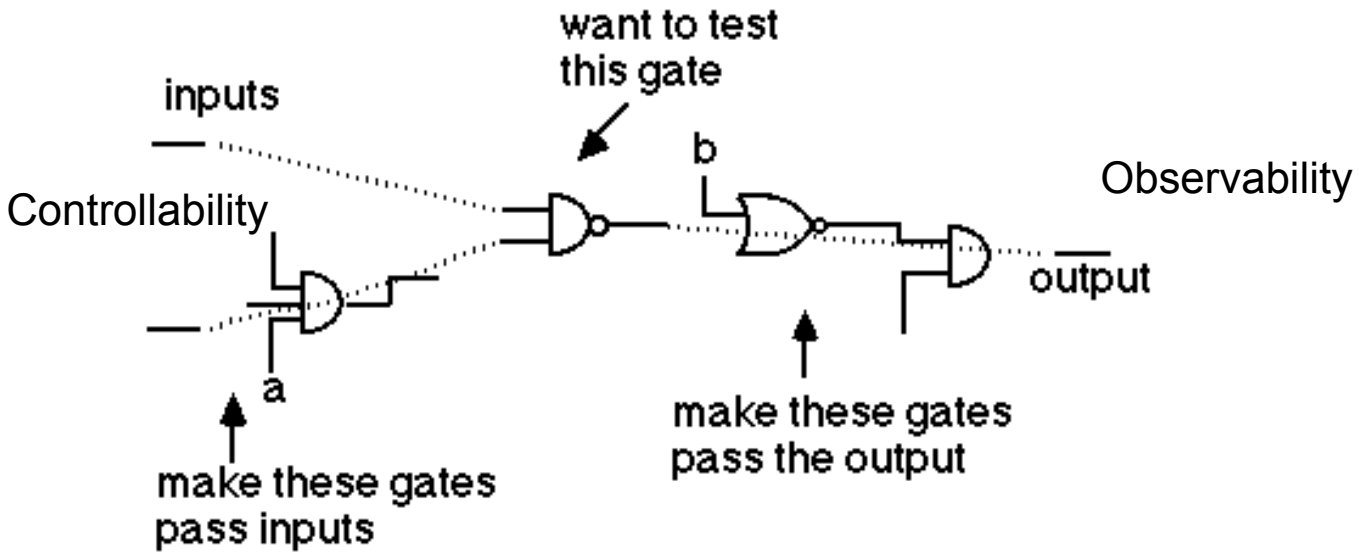
1. Need to test all the gates in a chip.
2. Only get to force chip inputs and observe chip outputs. (Production testing is done with automated equipment, never microprobing or e-beam, obviously)

   # gates >> # inputs/outputs.
3. Production testing is always done by applying a sequence of **vectors** (apply new input to pins, and measure outputs from pins), usually one vector per clock cycle.

If there was no state, the problem would merely be hard, but not impossible. With state, the problem is nearly impossible unless the designer helps.

Testing Combinational Logic

Problem is that you don't have direct control over the inputs to the gate you want to test, and you can't directly observe the output either.



This problem would be easy except for reconvergent fanout.--That is, that node "a" and node "b" can be the same node.

In general this is a 'hard' problem.

Reconvergent Fanout

In real logic circuits you need to choose a consistent set of assignments along the input paths, so the output path is also activated.

This choice is hard (impossible) to do without some back-tracking (exponential time in the worst-case)

But the average case is not very bad, and there are a number of algorithms / programs to help generate the test vectors:

D-algorithm

Podem

(Take McCluskey's class here at Stanford …)

Problem is with State --

- Don't have direct control of inputs, since most inputs are really the outputs of latches and therefore the result of the previous inputs. We will address this problem soon, but first there are other issues …

Are All Faults Detectable?

The process of controlling inputs and observing outputs is called “sensitizing” a path to a particular fault.

Sometimes, there is no combination of inputs that will sensitize a particular fault. Does this mean the fault doesn’t matter? Sometimes it really doesn’t matter, because there may be redundant logic in the circuit. Test generation programs therefore help to find redundant logic that can then just be removed from a design.

But usually, the redundant logic is there for a reason, such as in a carry-lookahead adder, the logic helps to make the overall critical path shorter. If there was a fault in some of the redundant logic, the observable results might still be correct, but slower.

But production testers are usually applying the vectors more slowly than real operation, so they can’t detect the speed degradation. It is better to change the design by bring out an internal node to an output, so that the faults will be observable with a change in logic result that the automated tester can see.

Fault Models

Most automatic test pattern generation (ATPG) programs use a simple Stuck-At-One (ST1) and Stuck-at-Zero (ST0) model of faults.

But, most real faults are actually opens or shorts. Unfortunately, generating tests for these kinds of faults is much more ugly. Generally, people and programs just use stuck-at models and hope they will catch the real faults anyway.

In CMOS, an open node stores a value. If it is changing slowly enough, then a sequence of stuck-at tests for that node will catch it. Need to try to drive it to one value, then try to drive it to the other value to detect that it was open and didn't change. So, if you have a complete set of patterns testing every node for ST1 and ST0, then just apply it twice to make sure every node gets either ST1-ST0-ST1-ST0 or ST0-ST1-ST0-ST1 tests.

Shorted nodes can only be sensitized if the two drivers are in opposite states. Without information about geometric locations, there are $(N-1)^2$ possible shorts in an N node circuit. If you know the layout the number drops to a few times N.

IDDQ Testing

Measuring the quiescent power supply current (IDDQ) as a method of testing is another good idea in CMOS circuits.

Once pure fully-complementary static CMOS circuits have driven their outputs they only draw picoamps from the power supply to compensate for leakage.

So, design the chip with a way of turning off all DC power consumption (like ratioed-logic NOR gates), and then after every vector, the power supply current will settle down. Even on big chips with millions of transistors, the leakage current of all the transistors together is less than (or about the order of) the on-current of a single transistor that is stuck on.

So, measuring IDDQ is a great way to detect shorts which will cause fighting output drivers to consume current.

But, don't have time to measure IDDQ after every test vector, so just measure it after a few well chosen ones, and hope.

Testing Circuits with State

Suppose you have a 32 bit counter:

- With no inputs but clock and the outputs there are two problems:
  - The tester does not know where the counter starts. Thus it can't simply compare the counter output to the 'expected' value.
  - To test the counter takes 2^{32} clock cycles. This is a long time to test just 32 counter cells.
- Adding a reset fixes the first problem.
  - All counters (and state in general) need a reset to enable the tester to get the chip in a known starting state. Testers never synchronize or change vectors based on "if" conditional statements like a program.
- Adding a load input fixes the second problem.
  - With a load it takes O(32) steps to test the counter.
  - To efficiently test state machines, you need a direct way of getting the state machine into the state you want to test. The easiest way of doing this it to provide a mechanism to load the state in the state machines.

Design for Testability

An absolute requirement for Modern Chips.

Partition the chip to make controllability of all state and therefore the testing, easier. Think about changing a sparsely connected FSM into a fully-connected graph, where every node can be entered directly by a transition from every other node.

Wide Range of Options:

- Explicit built-in-self-test (BIST)
- Additions to the (micro)code to make it go through a test sequence
- Extra Test Buses
- Link all registers/latches into scan chains, to make state scannable.

There are some automatic test pattern generation programs that do a really good job, but only if all state is scannable. This is a motivation to make the state scannable even if it seems excessive.

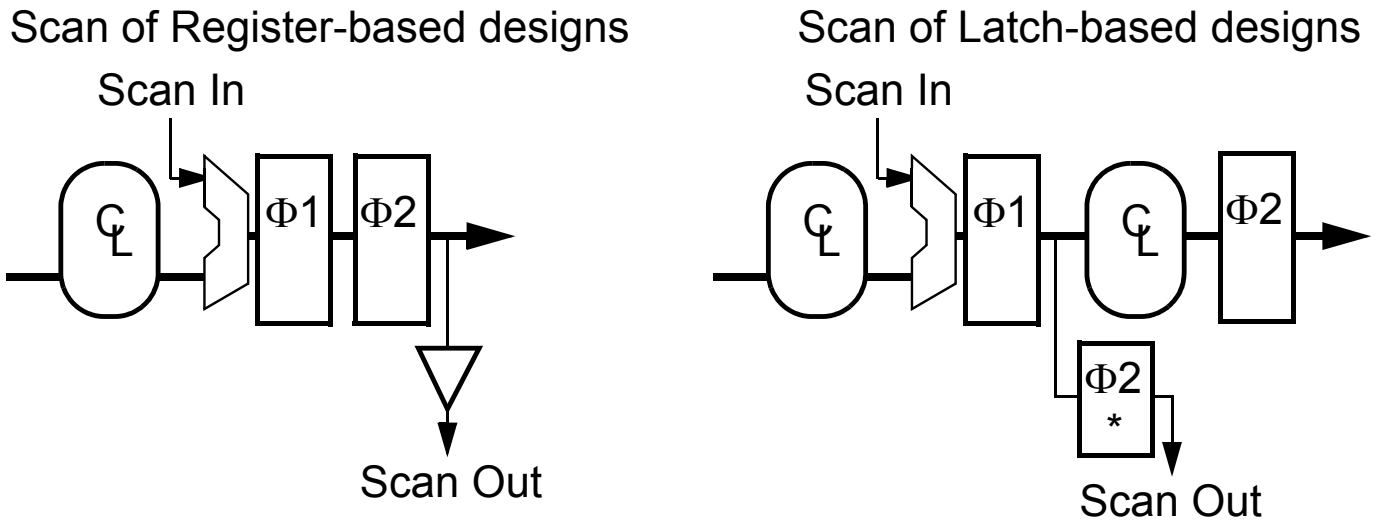
“Live within your tools”

Scan

Scan techniques are often used in testing, since they take only a few (about 4 to 6) additional pins, but have the potential to control a lots of things (state, other output pads, etc.). Trade bandwidth for pins. Techniques requiring everything to be scannable are often called by the buzzword LSSD (level-sensitive scan design) which really just means “full-scan”. (Original meaning also implied strict 2 phase clocking)

- Convert testing a sequential machine into solely the problem of testing a block of combinational logic with ALL primary inputs controllable through scan, and all primary outputs observable through scan.
- Accomplish this by changing all the state latches into a latch that can become a shift-register, and linking them all together in **scan chains**.
- A state can be shifted in, all the logic can compute, and then the state can be shifted out at the same time the next state is shifted in.
  Do a complete scan between every applied test vector.
- Scan great for both debug and production wafer or packaged part testing
- Partial-scan is possible, but ATPG tools are not as automated.

LSSD



- In test mode, registers shift data and form large shift registers.
- Since all the CL blocks can be 'directly' driven and observed, we can use a test generation program for combinational logic.
- In a latch-based chip, need to add extra Φ2 latches so the scan chain doesn't go through combinational logic.
- In both cases, scan-in muxes can be combined into Φ1 latches.

Design for LSSD

Does not affect user-visible architecture but

- Makes registers or latches more complex (larger and slightly slower)

  $\rightarrow$ area penalty.

- The additional wiring taken up by the scan wires is usually minuscule, but this is only true when the CAD flow can generate a good scan ordering, which must be determined after cell placement (and then backannotated into logic netlist).

For many semi-custom design, full-scan makes sense since it allows the production test vectors to be generated with good test coverage.

For full custom, it use is growing.

- While datapaths are pretty easy to test, and you don't need to scan architecturally-visible registers, just read them, there is a great fear about debugging chips that can't be probed. You need some internal access, and scan is one way to get it.

Built-In-Self-Test

The buzzword BIST is best used to refer to the portions of a design where an additional FSM has been added solely for the purposes of running through a test sequence where the results are NOT observable by the tester on every clock cycle.

Typically, BIST is added to test large (RAM) arrays where the number of clock cycles required by conventional vector-based tests would be just too excessive.

Additionally, BIST can be used to test the delay performance of circuits which would also not be accessible by a tester applying vectors slowly.

The result of BIST is meant to be a pass/fail result, not diagnostic information. Typically, the BIST controller compresses the results into a “signature” which can be compared against the signature of a known good chip.

Unlike other methods of design for testability, adding BIST usually doesn’t really help in providing any controllability for design debug.

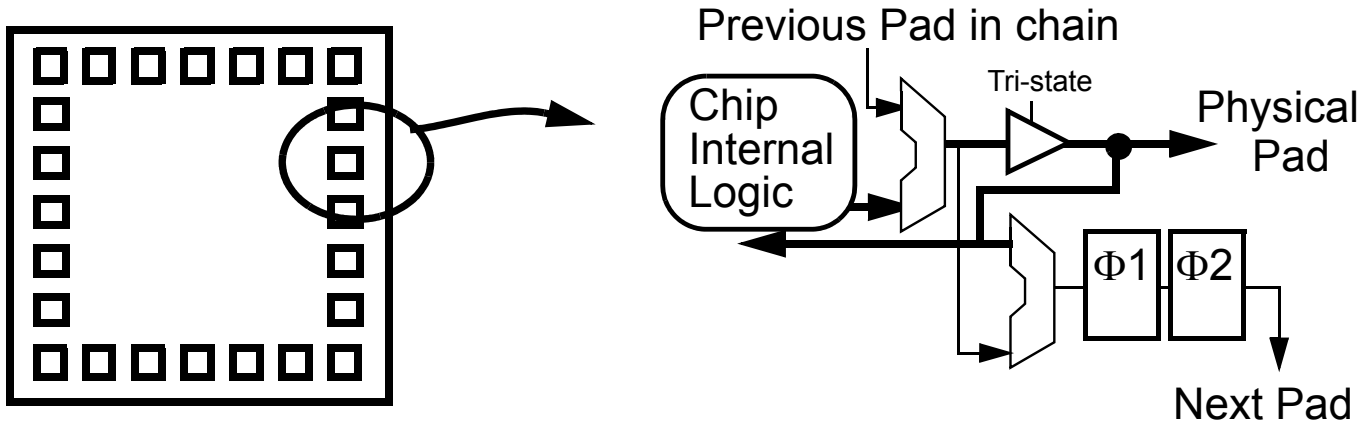
Helping out Board-level Testing

With the increase in chip pin count, and board complexity, printed circuit board testing is becoming an issue too:

- Old ‘bed-of nails’ testing does not work
  - This is where the board was pulled against a large number of pin points. The points could be either sensed or driven, allowing the tester access (both control and observe) to the board level signals.
- Leads of surface-mount components are inaccessible after soldering, so can only test by being able to control the pads of the chips.
- Custom and semi-custom chips are starting to routinely have features for board testing built in. Boards can be easily tested if all the chips on the boards use the same standard
- JTAG (an IEEE standard) adds a pad-boundary scan to allow the tester to control the I/O pins on each chip to test the board’s wires.
- JTAG can also give access to the internal scan of the individual chips

JTAG

Each chip contains a JTAG controller that can recognize when it is being individually addressed by the board-level test sequence. The controller generates the signals to control the muxes in each pad.



Using ability to make every pin either an input or output, the tester drives a value on a pin, and then checks the input pins of the other chips that should be connected to this chip. If they don't see the same value, the 'board' is not working correctly (could be solder connections, or a broken via in the board)

Overall System Test Strategy

To enhance controllability and observability both inside of chips and at the board level, use the techniques discussed in this lecture:

- Add reset signals and muxes to allow more direct control.
- Bring key signals out from inside of complex combinational logic blocks.
- Add Scan to latches/registers.
- Add BIST around RAMs or other arrays, using signature compression to compare the results.
- Use JTAG to give pad access to the board-level test controller
- Parallelize test of different chips at the board-level
- Define the tests that can be run or initiated as “diagnostics” by the customer in a finished system.