LEC-23: Scan Testing and JTAG

Lecture Notes Sections: 6.5 – 6.7.3

University of Waterloo

Dept of Electrical and Computer Engineering

E&CE 427 Digital Systems Engineering
2002-Winter

Schedule . . . .

wk-01 – 04	VHDL Design and Optimization
wk-05	Functional Validation
wk-06	Performance Analysis, Prediction, and Optimization
wk-07	Design Wrapup
wk-08	IRS and Holiday (No Lecture or Tutorial)
wk-09	Timing Analysis
wk-10	Power Analysis and Reduction
wk-11	Power Reduction; Faults
wk-12	Fault Detection; Built-In Self Test (BIST)
wk-13	**Scan Testing (JTAG)**; Review

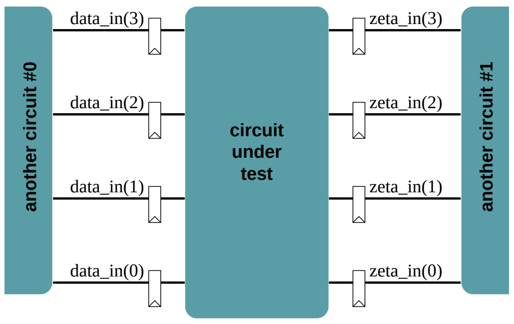
Purpose and List of Concepts . . . .

The purpose of this lecture is to connect the theory of testing and testability to the current techniques of scan testing and the IEEE Standard 1149.1 (aka JTAG).

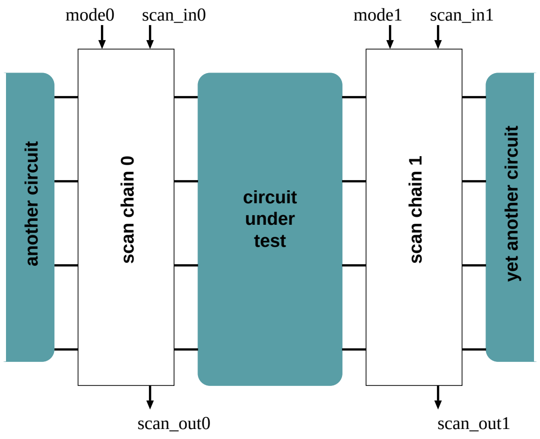
- scan testing
- scan chain
- testing procedure
- time to run a test
- boundary scan testing
- JTAG
- IEEE 1149
- length of time to do a scan test
- hardware to do scan testing

6.5 Scan Testing in General (Smith 14.6)

6.5.1 Structure and Behaviour of Scan Testing



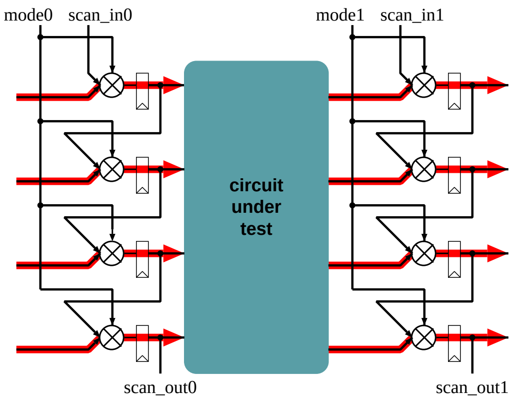
Normal Circuit



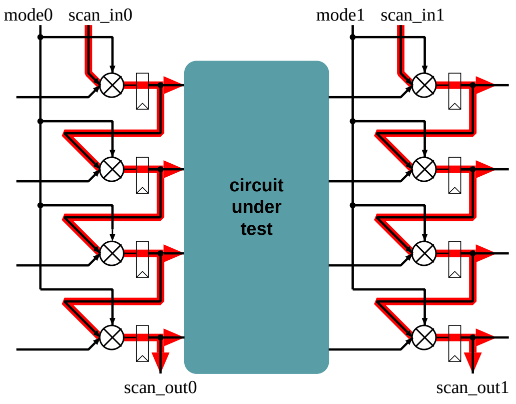
Circuit with Scan Chains Added

6.5.2 Scan Chains

6.5.2.1 Circuitry in Normal and Scan Mode

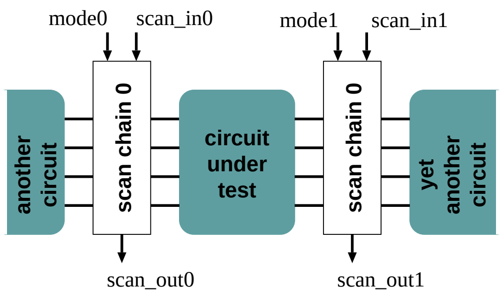


Normal Mode

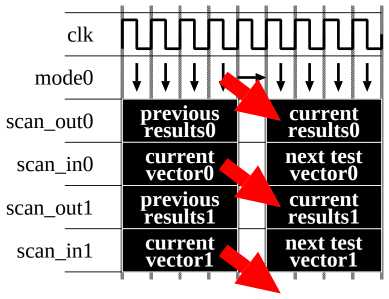


Scan Mode

6.5.2.2 Scan in Operation

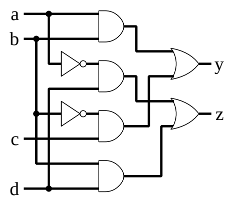


Circuit under test with scan chains

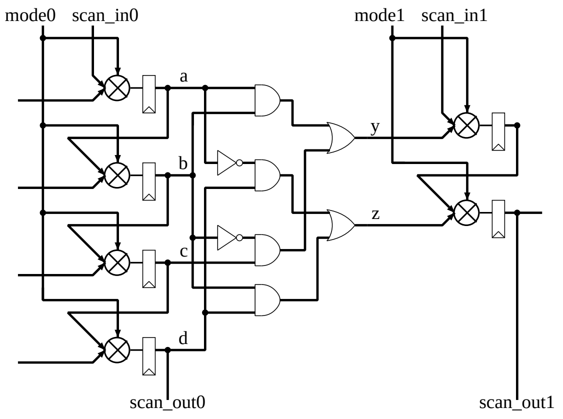


Behaviour of scan testing

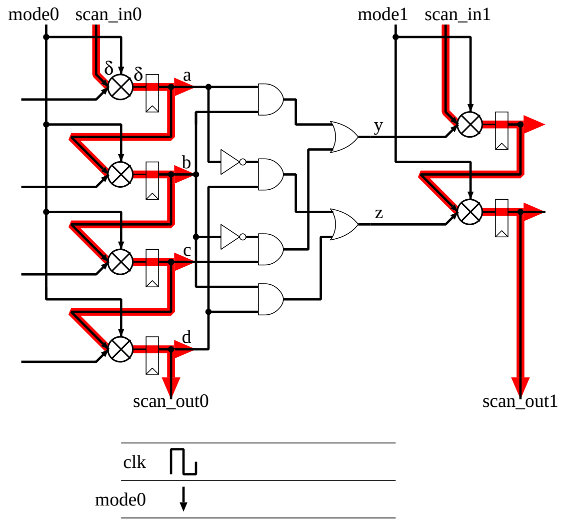
6.5.2.3 Scan in Operation with Example Circuit



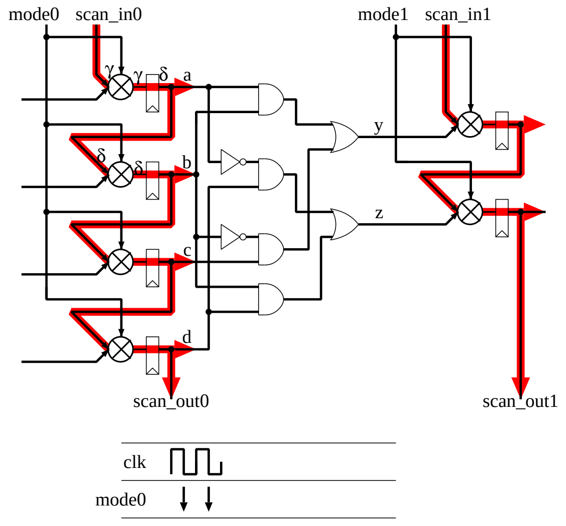
Circuit under test



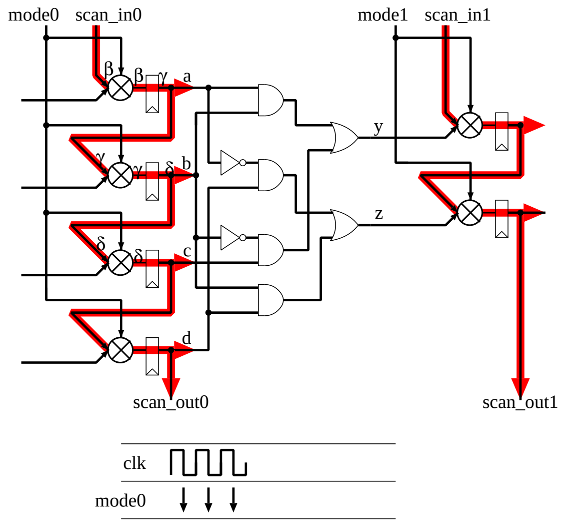
Circuit under test with scan test circuitry



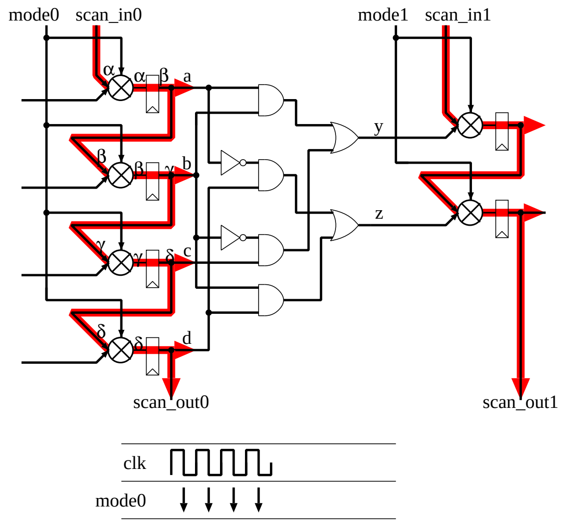
Start Loading Test Vector (Load δ)



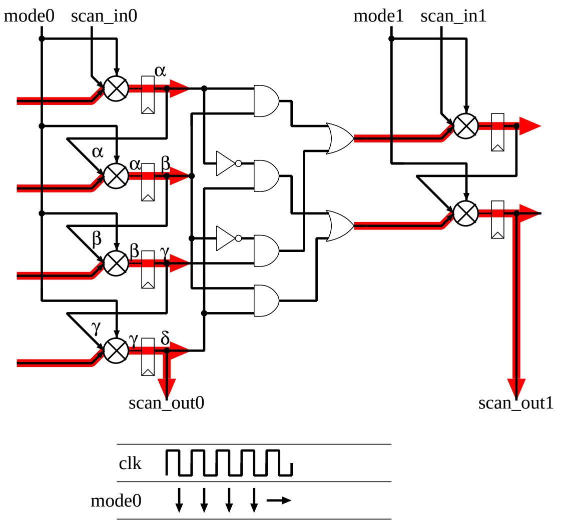
Load γ



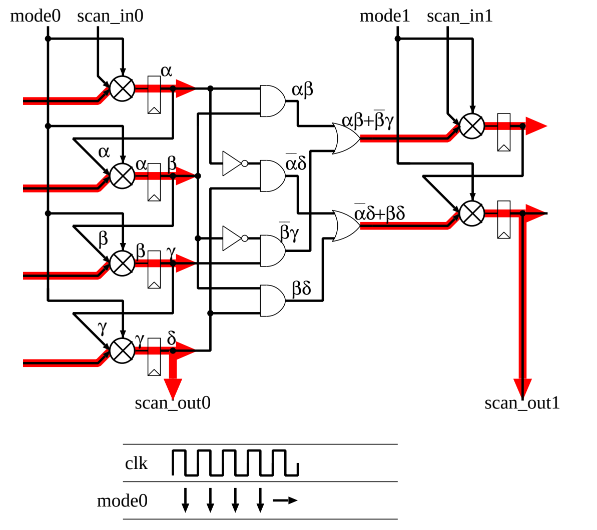
Load β



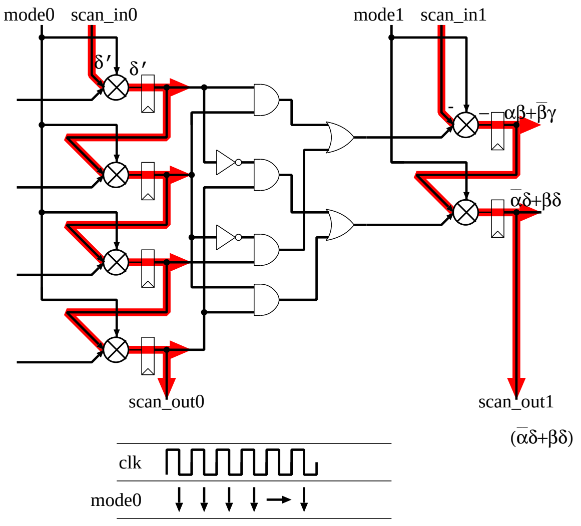
Load α



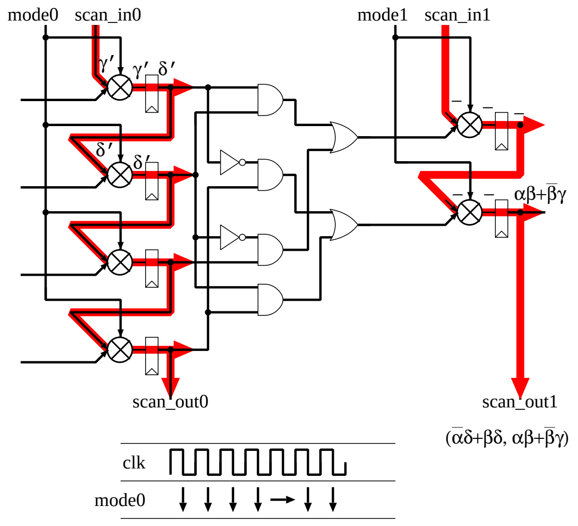
Run Test Vector



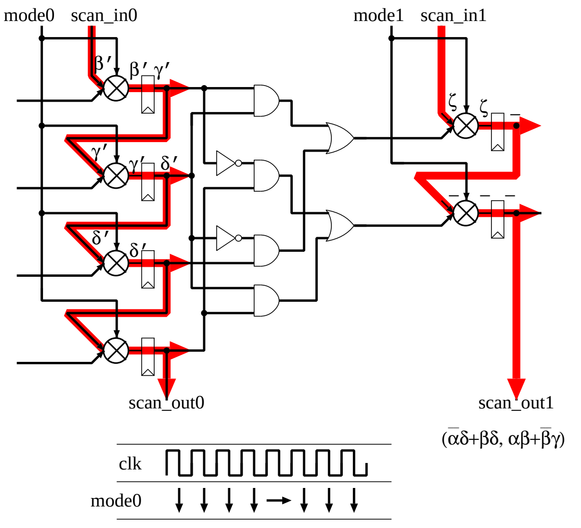
Test Values Propagate


mode0
scan_in0
mode1
scan_in1
δ′
$\alpha\beta+\bar{\beta}\gamma$
$\bar{\alpha}\delta+\beta\delta$
scan_out0
scan_out1
$(\bar{\alpha}\delta+\beta\delta)$
clk
mode0

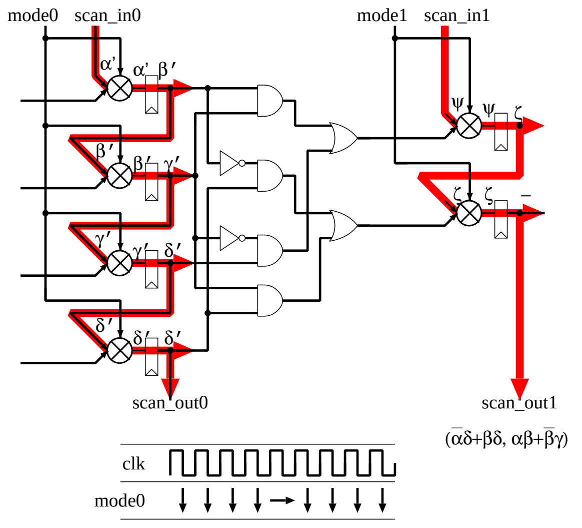
Flop-In Result, Start (Un)loading Test Vector


mode0
scan_in0
mode1
scan_in1
γ′
δ′
$\alpha\beta+\bar{\beta}\gamma$
scan_out0
scan_out1
$(\bar{\alpha}\delta+\beta\delta,\ \alpha\beta+\bar{\beta}\gamma)$
clk
mode0

Continue (Un)loading Test Vector


mode0
scan_in0
mode1
scan_in1
β′
γ′
δ′
ζ
scan_out0
scan_out1
$(\bar{\alpha}\delta+\beta\delta,\ \alpha\beta+\bar{\beta}\gamma)$
clk
mode0

Finish (Un)loading Test Vector


mode0
scan_in0
mode1
scan_in1
α′
β′
γ′
δ′
ψ
ζ
scan_out0
scan_out1
$(\bar{\alpha}\delta+\beta\delta,\ \alpha\beta+\bar{\beta}\gamma)$
clk
mode0

Run Next Test Vector

6.5.3 Summary of Scan Testing

- Adding scan circuitry
  1. Registers around circuit to be tested are grouped into scan chains
  2. Replace each flop with mux + flop
  3. Flops and muxes wired together into scan chains
  4. Each scan chain is connected to dedicated I/O pins for loading and unloading test vectors
- Running test vectors
  1. Put scan chain in “scan” mode
  2. Load in test vector (one element of vector per clock cycle)
  3. Put scan chain in “normal” mode
  4. Run circuit for one clock cycle — load result of test into flops
  5. Unload results of current test vector while simultaneously loading in next test vector (one element of vector per clock cycle)

6.5.4 Example: Time to Test a Chip

A 800MHz chip has scan chains of length 20,000 bits, 18,000 bits, 21,000 bits, 22,000 bits, and two of 15,000 bits.
500,000 test vectors are used for each scan chain.
The tests are run at 80% of full speed.

Question: *Calculate the total test time.*

Answer:

We can load and unload all of the scan chains at the same time, so time will be limited by the longest (22,000 bits).

For the first test vector, we have to load it in, run the circuit for one clock cycle, then unload the result.

Loading the second test vector is done while unloading the first.

$$\begin{aligned}
\mathit{TimeTot} &= \mathit{ClockPeriod} \\
&\quad \times(\mathit{MaxLengthVec} + \mathit{NumVecs} \times (\mathit{MaxLengthVec} + 1)) \\
&= (1/(0.80 \times 800 \times 10^6)) \times (22,000 + 500,000 \times (22,000 + 1)) \\
&= 17 \mathit{secs}
\end{aligned}$$

6.6 Boundary Scan

Boundary scan originated as technique to test wires on printed circuit boards (PCBs).

Goal was to replace “bed-of-nails” style testing with technique that would work for high-density PCBs (lots of small wires close together)

Now used to test both boards and chip internals.

Used both on boundaries (I/O pins) and internal flops.

Standardized by IEEE (1149) and previously by JTAG:

- 4 required signals (Scan Pins: `TDI`, `TDO`, `TCK`, `TMS`)
- 1 optional signal (Scan Pin: `TRST`)
- protocol to connect circuit under test to tester and other circuits
- state machine to drive test circuitry on chip
- Boundary Scan Description Language (BSDL): structural language used to describe which features of JTAG a circuit supports

JTAG circuitry now commonly built-into FPGAs and ASICS, or part of a cell-library. Rarely is a JTAG circuit custom-built as part of a larger part. So, you'll probably be choosing and using JTAG circuits, not constructing new ones.

Using JTAG circuitry is usually done by giving a description of your printed circuit board (PCB) and the JTAG components on each chip (in BSDL) to test generation software. The software then generates a sequence of JTAG commands and data that can be used to test the wires on the circuit board for opens and shorts.

6.6.1 Boundary Scan History

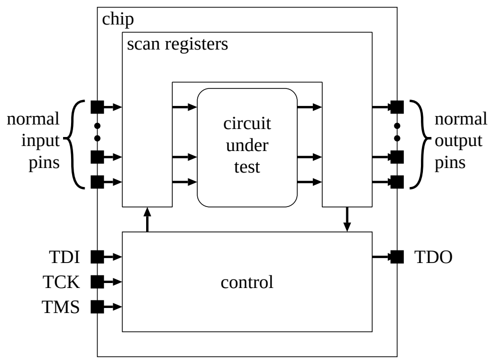
1985 JETAG: Joint European Test Action Group

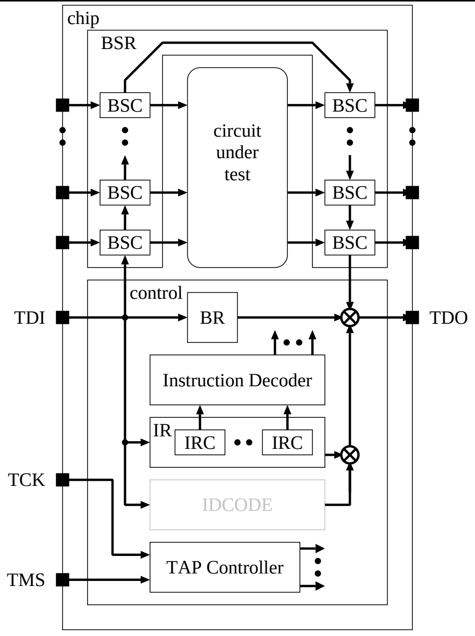
1986 JTAG (North American companies joined)

1990 JTAG 2.0 formed basis for IEEE 1491 "Test access port and boundary scan architecture"

6.6.2 Scan Pins

`TDI`	$\longrightarrow$	test data input: input testvector to chip
`TDO`	$\longleftarrow$	test data output: output result of test
`TCK`	$\longrightarrow$	test clock: clock signal that test runs on
`TMS`	$\longrightarrow$	test mode select: controls scan state machine
`TRST`	$\longrightarrow$	test reset (optional): resets the scan state machine





6.6.3 Scan Registers and Cells

Basic Building Blocks . . . .

`TDR`		Test data register The boundary scan registers on a chip
`DR`	Fig 14.2	Data register cell Often used as a Boundary scan cell (BSC)

JTAG Components . . . .

	Fig 14.8	Top level diagram
`BSR`	Fig 14.5	Boundary scan register A chain of boundary scan cells (BSCs)
`BSC`	Fig 14.2	Boundary scan cell Connects external input and scan signal to internal circuit. Acts as wire between external input and internal circuit in normal mode.
`BR`	Fig 14.3	Bypass-register cell Allows direct connection from TDI to TDO. Acts as a wire when executing BYPASS instruction.
`IDCODE`		Device identification register data register to hold manufacturer's name and chip identifier. Used in IDCODE instruction.
`IR cell`	Fig 14.4	Instruction register cell Cells are combined together as a shift register to form an instruction register (IR)
`IR`	Fig 14.6	Instruction register Two or more IR cells in a row. Holds data that is shifted in on TDI, sends this data in parallel to instruction decoder.
IDecode	Table 14.4	Instruction decoder Reads instruction stored in instruction register (IR) and sends control signals to bypass register (BR) and boundary scan register (BSR)
	Fig 14.7	TAP Controller State machine that, together with instruction decoder, controls the scan circuitry.

6.6.4 Scan Instructions

This the set of required instructions, other instructions are optional.

`EXTEST`	Test board-level interconnect. Drive output pins of chip with hard-coded test vector. Sample results on inputs.
`SAMPLE`	Sample result data
`PRELOAD`	Load test vector
`BYPASS`	Directly connect TDI to TDO. This is used when several chips are daisy chained together to skip loading data into some chips.
`IDCODE`	Output manufacturer and part number

6.6.5 TAP Controller

The TAP controller is required to have 16 states and obey the state machine shown in Fig 14.7.

6.6.6 Other descriptions of JTAG/IEEE 1194.1

Texas Instruments introductory seminar on IEEE 1149.1
`http://www.ti.com/sc/docs/jtag/seminar1.pdf`
Texas Instruments intermediate seminar on IEEE 1149.1
`http://www.ti.com/sc/docs/jtag/seminar2.pdf`
Sun midroSPARC-IIep scan-testing documentation
`http://www.sun.com/microelectronics/whitepapers/wpr-0018-01/`
Intellitech JTAG overview:
`http://www.intellitech.com/resources/technology.html`
Actel's JTAG description:
`http://www.actel.com/appnotes/97s05d15.pdf`
Description of JTAG support on Motorola Coldfile microprocessor:
`http://e-www.motorola.com/collateral/MCF5307TR-JTAG.pdf`

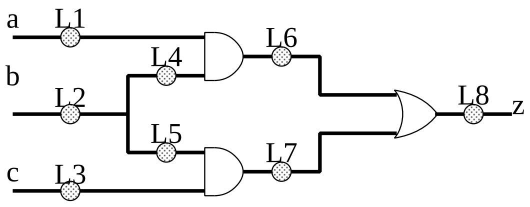
6.7 Summary and Conclusions on Testing

6.7.1 Faults

Faults are manufacturing defects. Common occurences are **opens** (wire is broken) and **shorts** (two wires are connected together).

When working with faults, we work with wire **segments**, not signals. In the circuit below, there are 8 different wire segments (L1–L8). Each wire segment corresponds to a logically distinct fault location. All physical faults on a segment affect the same set of signals, so they are grouped together into a "logical fault". If a signal has a fanout of 1, then there is one wire segment. A signal with a fanout of n, where $n > 1$, has $n+1$ wire segments — one for the source signal and one for each gate of fanout.

For signal "b" in the circuit here, the fanout is 2, so there are three wire segments (L2, L4, and L5).



Although there are many different bad behaviours that faults can lead to, the simple model of **single-stuck-at-faults** has proven very capable of finding real faults in real circuits.

single	assume that at most wire segment in circuit has a fault.
stuck-at-0 (`s@0`)	assume that the faulty behaviour is that the segment is hardwired to 0.
stuck-at-1 (`s@1`)	assume that the faulty behaviour is that the segment is hardwired to 1.

6.7.2 Testing

Faults are detected by stimulating circuits (real, manufactured circuit, not a simulation!) with **test-vectors** and checking that real circuit gives correct output.

Standard practice in testing is to test circuits for single stuck-at faults. Mathematics and empirical evidence demonstrate that testing a circuit for single stuck-at faults will also detect many other types of faults and will often detect multiple faults.

Some faults are undetectable. Undetectable stuck-at faults are located in **redundant** parts of a circuit. These redundant parts are added to prevent timing hazards. As such, a stuck-at fault in redundant circuitry will not affect the steady state behaviour of the circuit, but could allow timing glitches to occur.

If a circuit has 100% single stuck-at fault coverage with a suite of test vectors, then each stuck-at fault in the circuit can be detected by one or more vectors in the suite. This also means that the circuit has no undetectable faults, and hence, no redundant circuitry.

It is possible that achieving 100% coverage for single stuck at faults will allow defective chips to pass if they have faults that are not stuck-at-1 or stuck-at-0, or if they have multiple faults.

I think, but haven't seen a proof, that achieving 100% single stuck-at coverage will detect all combinations of multiple stuck-at faults. But, if you do not achieve 100% coverage, then a stuck-at fault that you aren't testing for can mask (hide) a fault that you are testing for.

There are two ways to generate vectors and check result: built-in tests and scan testing.

Both require:

- generate test vectors
- overide normal datapath to send test-vectors, rather than normal inputs, as inputs to flops
- compare outputs of flops to expected result

6.7.2.1 Scan Testing

In scan testing, the generation and checking are done off-chip. This has the advantage of flexibility and reduced on-chip hardware, but increases the length of time required to run a test. We want to individually drive and read every flop in the circuit.

Even without using any I/O pins for testing purposes, chips are already I/O bound, so scan-testing must be very frugal in its use of pins. Flops are connected together in a "scan chain" with one input pin and one output pin.

If the length (number of flops) of a scan chain is n, then it takes $2n+1$ clock cycles to run a single test: n clock cycles to scan in the test vector, 1 clock cycle to execute the test vector, and n cycles to scan out the results. Once the results are scanned out, they can be compared to the expected results for a correctly working circuit.

If we run 2 or more tests (and chips generally are subjected to hundreds of thousands of tests), then we speed things up by scanning in the next test vector while we scan out the previous result.

ScanLength	=	number of flip flops in a scan chain
NumVectors	=	number of test vectors in test suite
TimeScan	=	number of clock cycles to run test suite
	=	$\mathsf{NumVectors} \times (\mathsf{ScanLength} + 1) + \mathsf{ScanLength}$

To find a test vector that will detect a fault:

1. build Boolean equation (or Karnaugh map) of correct circuit
2. build Boolean equation (or Karnaugh map) of faulty circuit
3. compare equations (or Karnaugh maps), regions of difference represent test vectors that will detect fault

Because it takes so much time to perform a scan test, reducing the number of test vectors that are needed is very important.

fault1 **dominates** fault2 is defined as: "any test vector that will detect fault1 will also detect fault2."

Summary of Technique to Find and Order Test Vectors:

1. identify all possible faults
2. gate collapsing
3. node collapsing
4. intelligent collapsing

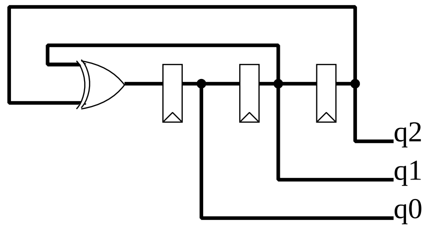
5. fault domination
6. determine required test vectors
7. choose minimal set of test vectors to detect remaining faults
8. order test vectors based on number of faults detected (NOTE: when iterating through this step, need to take into account faults detected by earlier test vectors)

6.7.2.2 Built-In Self Test (BIST)

With built-in self test, the circuit tests itself. Both test vector generation and checking are done using linear feedback shift registers (LFSRs).

The figure below shows an LFSR that generates all possible 3-bit vectors except 000. (An n bit LFSR that generates $2^n - 1$ different vectors is called a “maximal-length LFSR”.)

Assume that reset initializes the circuit to 111. The sequence that is generated is: 111, 011, 001, 100, 010, 101, 110. This sequence is repeated, so the number after 110 is 111.



Each linear feedback shift register has a characteristic polynomial, that corresponds to the behaviour of the signal that is the input to the first flip-flop in the shift register.

The exponents in the polynomial correspond to the delay x^0 is the input to the shift register, x^1 is the output of the first flip-flop, x^2 is the output of the second, etc. The coefficient is 1 if there’s a feedback tap from the output of the flop.

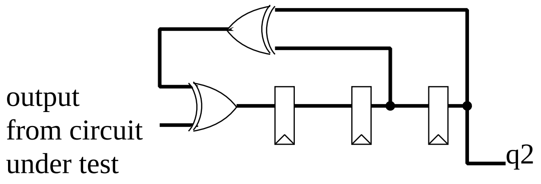
Checking is done by building one signature analyzer circuit for each signal tested. The circuit returns true if the signal generates the correct sequence of outputs for the test vectors. Doing this with complete accuracy would require storing 2^n bits of information for each output for a circuit with n inputs. This would be as expensive as the original circuit. So, BIST uses mathematics similar to error correction/detection to approximate whether the outputs are correct. This technique is called “signature analysis” and originated with Hewlett-Packard in the 1970s.

The checking is done with an LFSR, similar to the BIST generation circuit. The checking circuit is designed to output a 1 at the end of the sequence of $2^n - 1$ test results if the sequence of results matches the correct circuit. We could do this with an LFSR of $2^n - 1$ flops, but as said before, this would be at least as expensive as duplicating the original circuit.

The checking LFSR is designed similarly to a hashing function or parity checking circuit. If it returns 0, then we know that there is a fault in the circuit. If it returns a 1, then there is probably not a fault in the circuit, but we can’t say for sure.

There is a tradeoff between the accuracy of the analyzer and it’s area. The more accurate it is, the more flip flops are required.

The LFSR here recognizes the sequence 1, 0, 1, 1, 1, 0, 0:



It could be used, in conjunction with the maximal-length LFSR above, to detect faults in a circuit that, when stimulated with the sequence with the sequence 111, 011, 001, 100, 010, 101, 110; outputs the sequence 1, 0, 1, 1, 1, 0, 0.

6.7.3 Scan vs Self Test

Scan

- $\Uparrow$ less hardware
- $\Downarrow$ slower
- $\Uparrow$ well defined coverage
- $\Uparrow$ test vectors are easy to modify

Self Test

- $\Downarrow$ more hardware
- $\Uparrow$ faster
- $\Downarrow$ ill defined coverage
- $\Downarrow$ test vectors are hard to modify