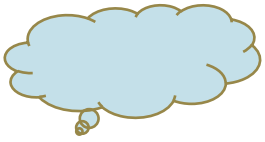
Formal Hardware Verification (some key ideas)

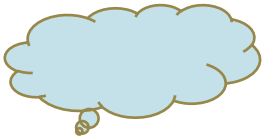
Mary Sheeran

Idealised Flow



High level
Not formal

Idealised Flow

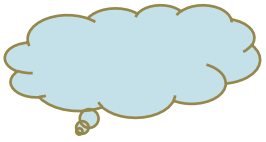


High level
Not formal

Formal spec.

Equally high level
Math, expressive logic

Idealised Flow



High level
Not formal

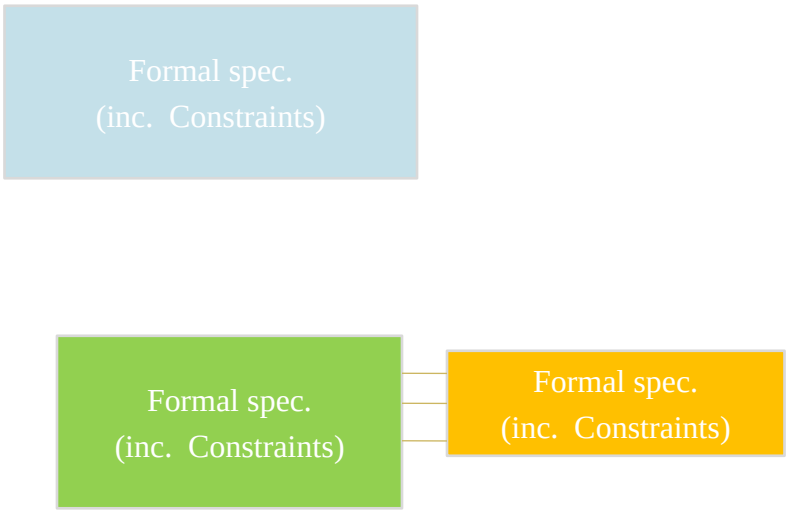
Formal spec.

Equally high level
Math, expressive logic

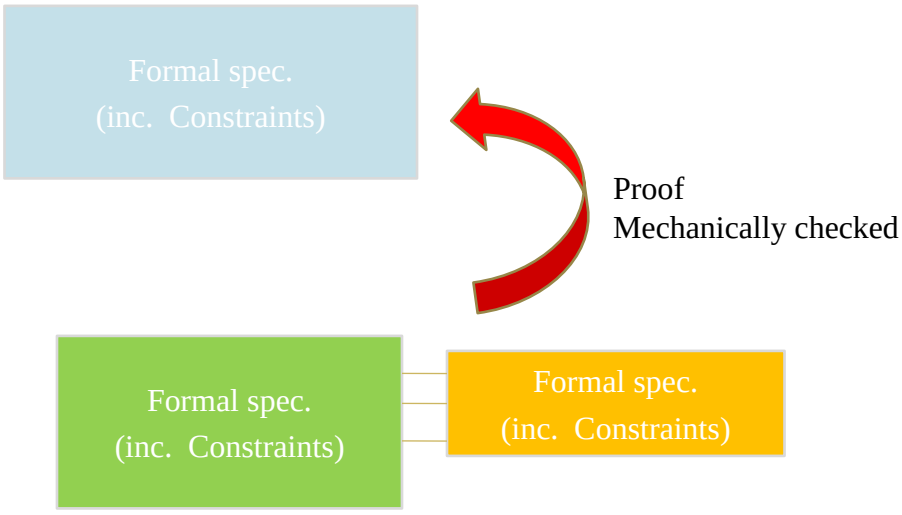
Specification Validation

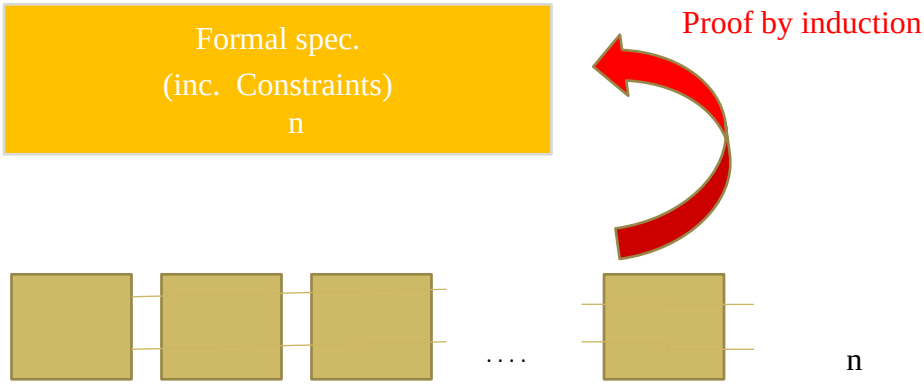
Not a formal process (big demand for tools that assist)

Refinement



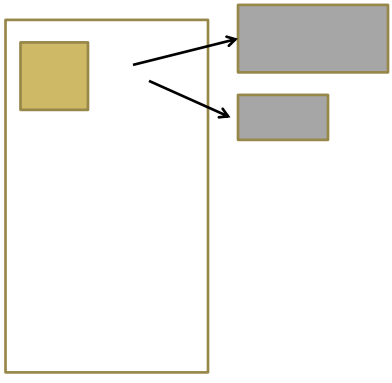
Refinement





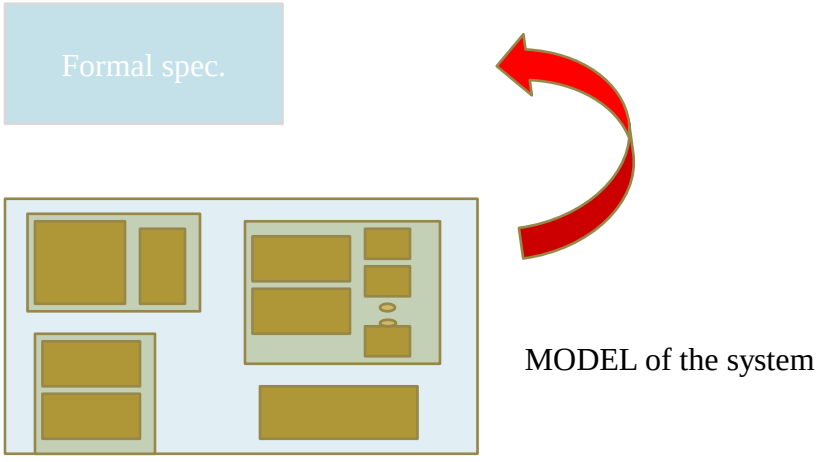
and so on recursively ….

Stop when reach library components that have physical implementation(s)



Design verification

Run proof from bottom up



Pros

Hierarchy is a (the?) way to manage complexity. Scalable.

This approach can span abstraction levels and in particular can start high up close to the original informal spec.

The proof is based on the circuit structure

It is mechanically checked.

Can prove generic (or parameterised) systems. (One proof gives a lot.)

Cons

Interactive theorem proving is difficult and time-consuming (often tedious too)

May need the lowest level components to be rather abstract to make it feasible

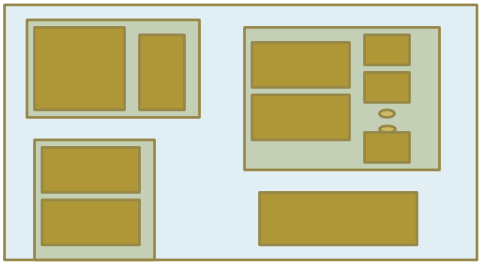
Hard to make the link to the very low level physical details. Risk leaving a gap to what is actually implemented

Idealised Implementation

Keep exact structure

Conservative design rules used to ensure that the abstract behaviour of the silicon is faithfully reflected in the system model

Link between implementation and design is checked in Implementation verification



Remember that the model captures only a simplified version of the behaviour.
Usually only function

Implementation verification

Often done by extracting a model from the actual layout (look in it to find where the transistors or gates are and how they are connected)

Make a model of this result and compare with the design (using Equivalence Checking (EC))

To make this feasible the design (golden model) has to be close to the actual implementation

Post-silicon verification

Did the manufacturing work?

Very Hard because have few pins for pumping data in and out (Formal methods used here too, more needed)

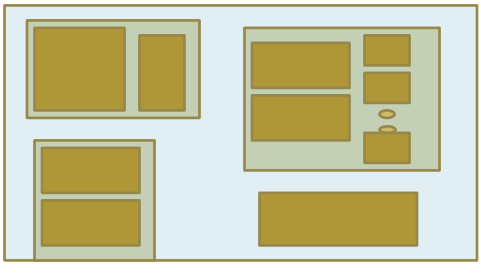
Specification validation (not formal)

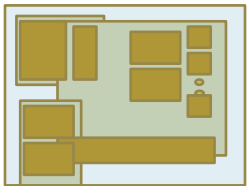
Design Verification

Implementation Verification

Post-silicon Verification

Reality gets in the way ☹





Pipelining

State encoding

Physical design messes up logical structure

Optimisations

Spec. is dragged downwards

What can we do??

Aim for automation (bit level)

Find niches where formal methods work well

Use assertions / properties first in sim. and then in FV

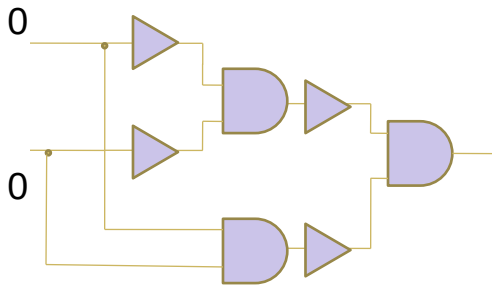
Idea 1: make simulators a little cleverer

Symbolic simulation

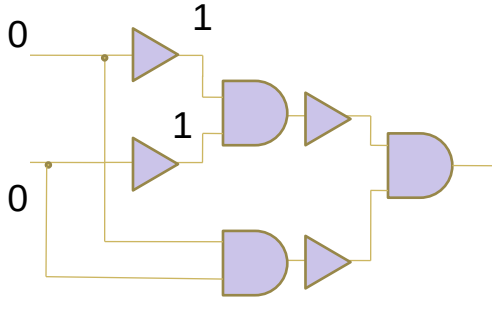
Take a simulator (can be quite low level, accurate one)

Make it work not only on 0, 1, X (unknown)
(or a larger group of constants) but ALSO on symbols

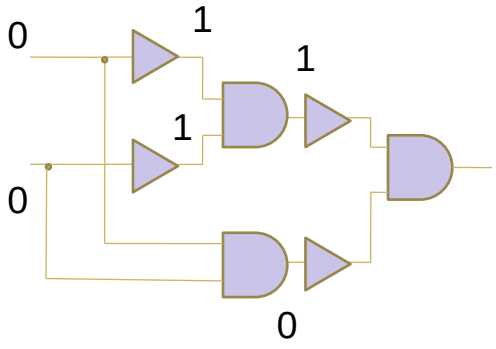
Ordinary simulation xor ?



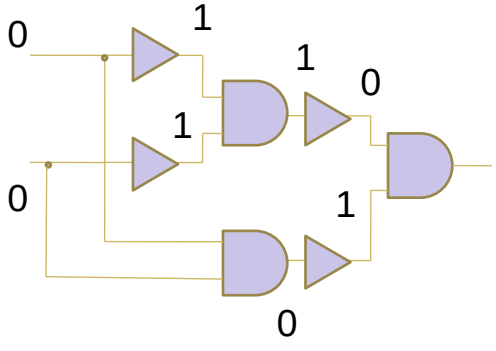
simulation



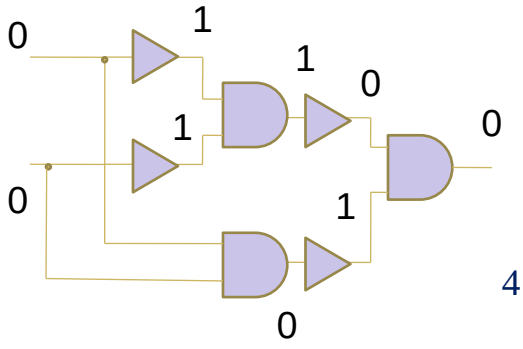
simulation



simulation



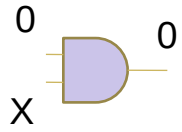
simulation



4 runs to check exhaustively

Q: how many for n inputs?

Symbolic simulation Idea 1

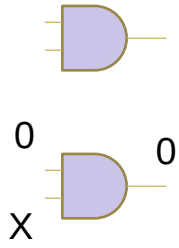


Use X values

Halves number of sim. runs!

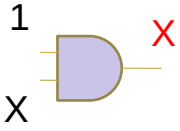
Why?

Symbolic simulation Idea 1



Use X values

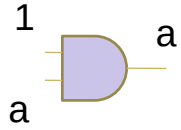
Halves number of sim. runs!



BUT may lose information

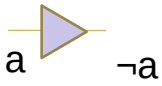
(try on xor example)

Symbolic simulation Idea 2



Use symbolic values

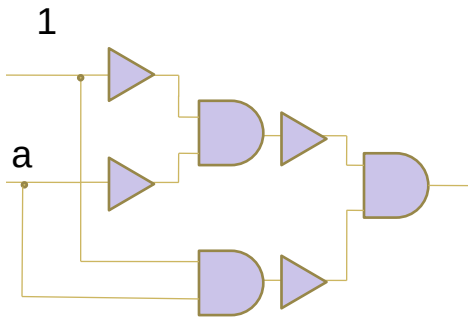
Think of giving input values names rather than constant values



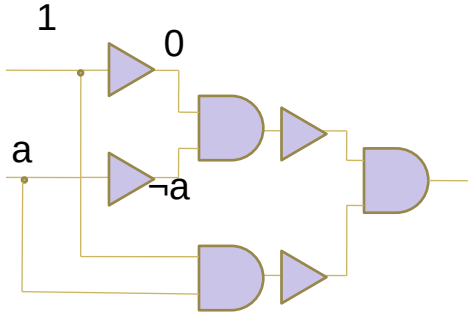
Build up an expression in terms of (some of the) inputs

May Rep. Using Binary Decision Diagrams (BDDs)

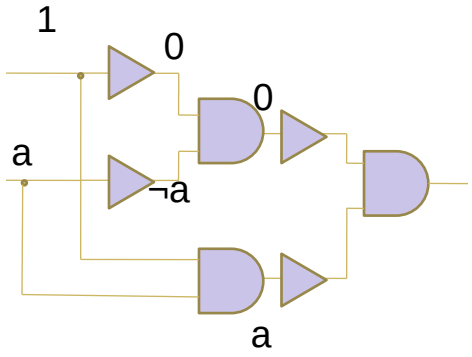
Symbolic simulation



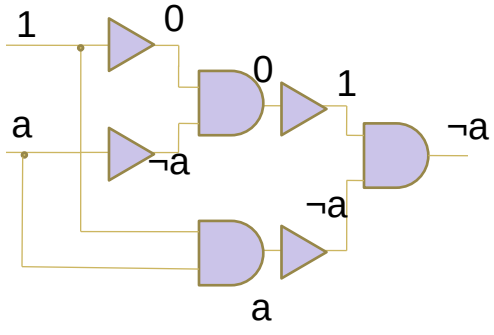
Symbolic simulation



Symbolic simulation



Symbolic simulation



1X X

1a ¬a

Symbolic simulation

Widely used (applies also to sequential circuits)

Forms basis of model checking method called Symbolic Trajectory Evaluation (STE)

User must make judicious choice of 0,1 X a, b, …

X halves sim runs, but may result in X at a point vital to the verification

Symbolic variable halves sim. runs without losing info.
BUT BDD somewhere in the sim. may grow too big

Questions?

Binary Decision Diagrams

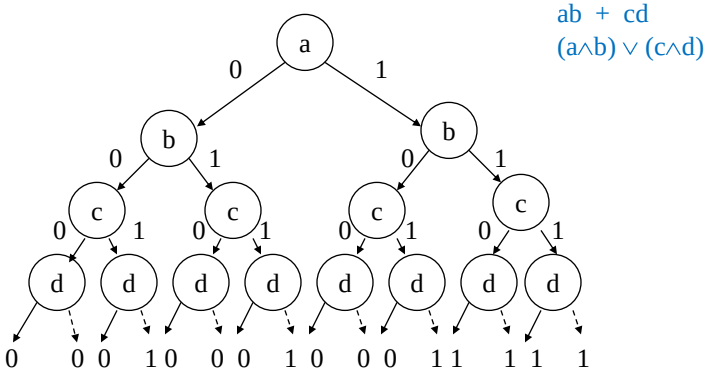
Vital enabling technology

Data structure for representing a Boolean function
(current form introduced by Bryant, known earlier)

Canonical form (constant time comparison)

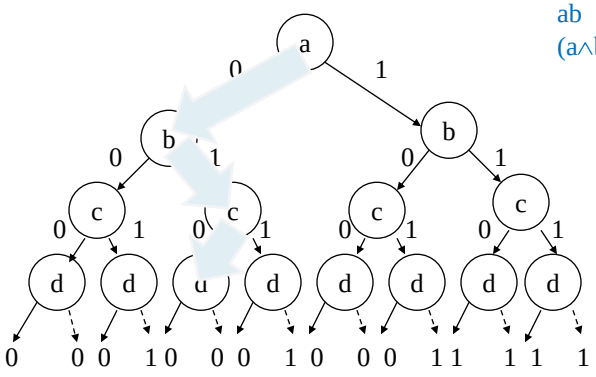
Used in Symbolic Model Checking

Ordered Decision Tree



Ordered Decision Tree

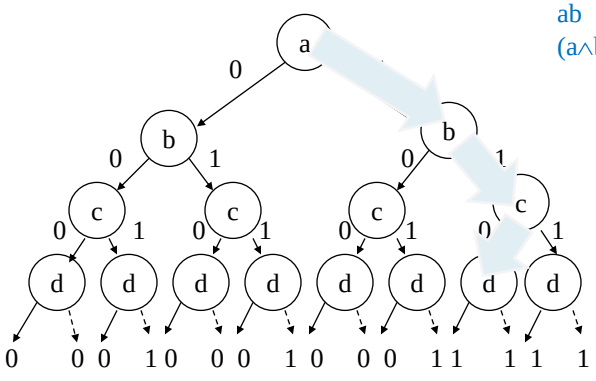
ab + cd
(a∧b) ∨ (c∧d)



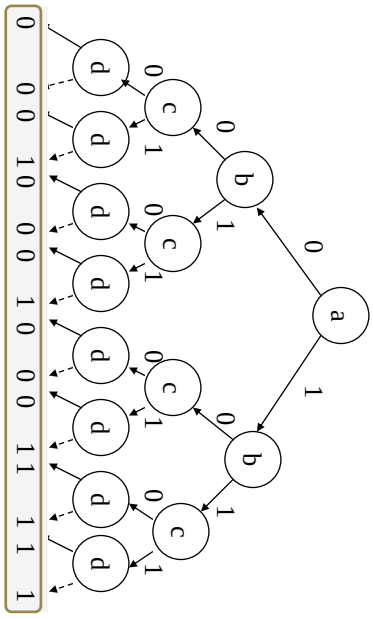
Every path from root to leaf obeys the variable ordering (a,b,c,d)

Ordered Decision Tree

ab + cd
(a∧b) ∨ (c∧d)



Every path from root to leaf obeys the variable ordering (a,b,c,d)



a b c d	
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
. . .	

truth table

To get OBDD

Combine isomorphic subtrees (same label, same children)

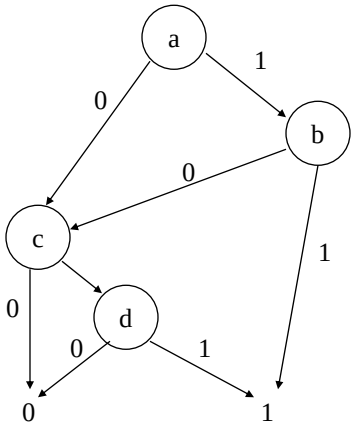
Eliminate redundant nodes (those with two identical children)

until no more reductions possible

Tree becomes a graph

(O)BDD

ab + cd
(a∧b) ∨ (c∧d)



Make (O)BDD for

$x \oplus y \oplus z$

$\oplus$ is xor

Above method just conceptual

In reality generated and manipulated in fully reduced form

Sharing exploited everywhere (hashing)

Efficient (polynomial time) algorithms for all usual operations (and, or etc., quantification)

Representation is canonical (for a given variable ordering)

Pros

Comparing Boolean functions cheap

[could use for what?]

Many small and usual functions have small BDDs

[example parity above

How big BDD for n inputs?

Exercise: How would it look in Conjunctive Normal Form (CNF)?]

Cons

Some usual and important functions have GIGANTIC BDDs

Q: How big is the BDD for a 16-bit binary multiplier?

Shifters are also problematic

Getting the variable order right is vital

Can make the difference between linear and exponential size!

Next Step

Model checking (week after next)