Formal Hardware Verification: getting started

Mary Sheeran

Making Formal Verification work

Aim for automation (bit level)

Find niches where formal methods work well

Use assertions/ properties first in sim. and then in FV (the acronym is ABV, assertion based verif.)

First question: what exactly do we reason about (in this course)?

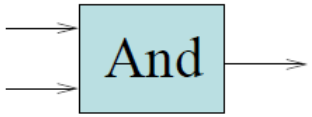
Answer: Finite State Machines (or state transition systems) at the bit level

Always reasoning about models of circuits

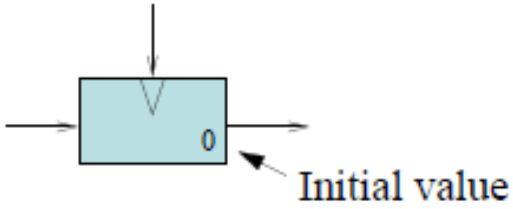
Need to be sure conclusions really apply to the final physical circuit

What is a (gate level) circuit (for us)?

boolean gates



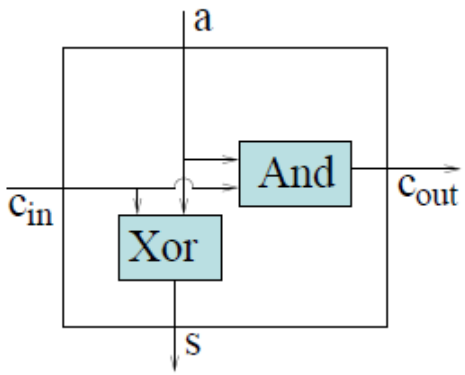
clocked state holding elements (d flip-flops)



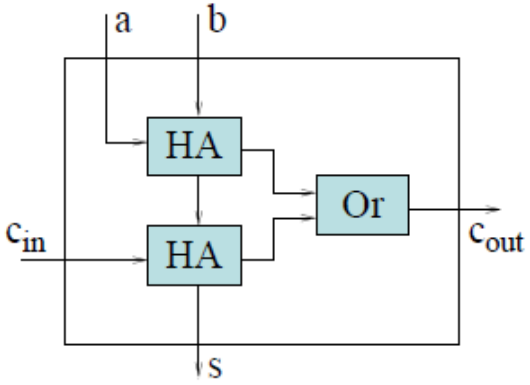
components (or boxes) containing gate level circuits

Combinational gate level examples

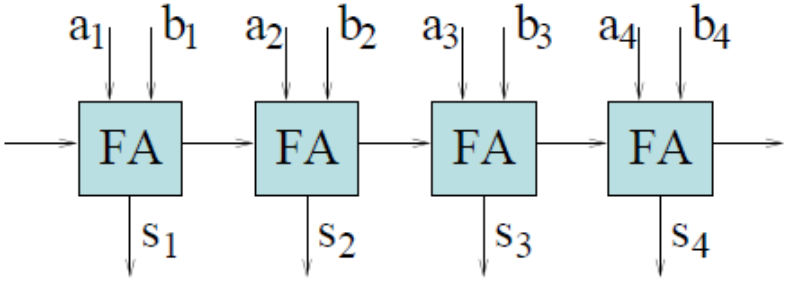
Half adder



Full adder

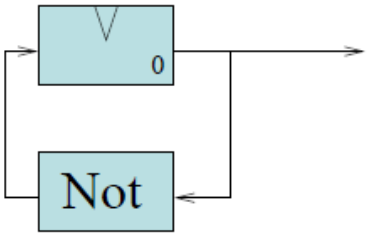


4 bit ripple-carry adder

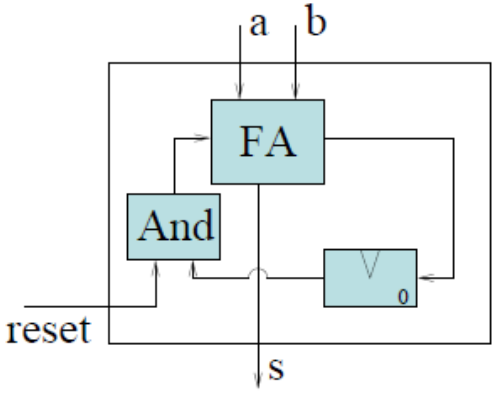


Sequential gate level examples

Oscillator



Sequential adder



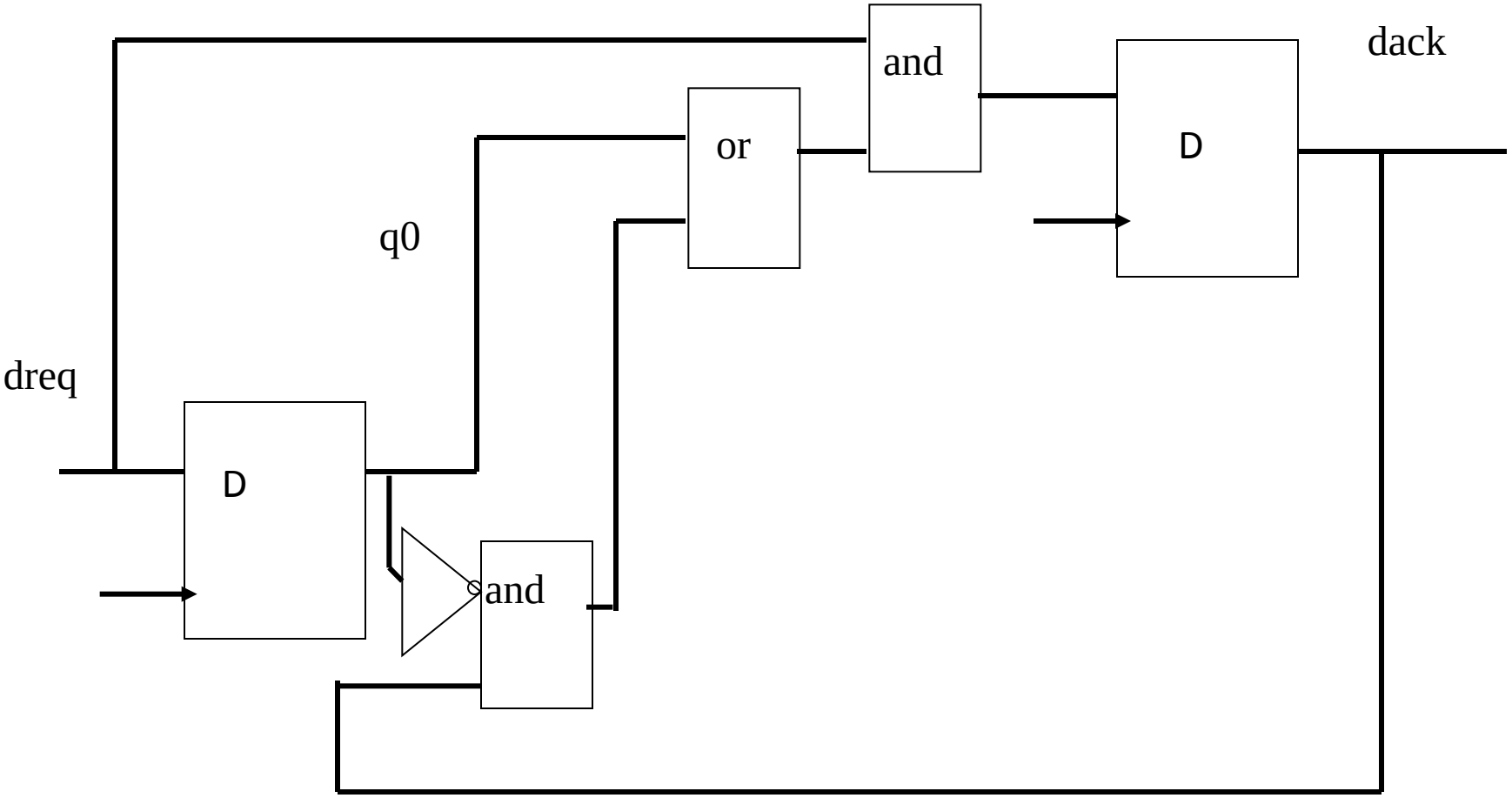
Rules

Each cycle contains at least one flip-flop

Wires can be split but not joined

Single clock

Simple synchronous circuits

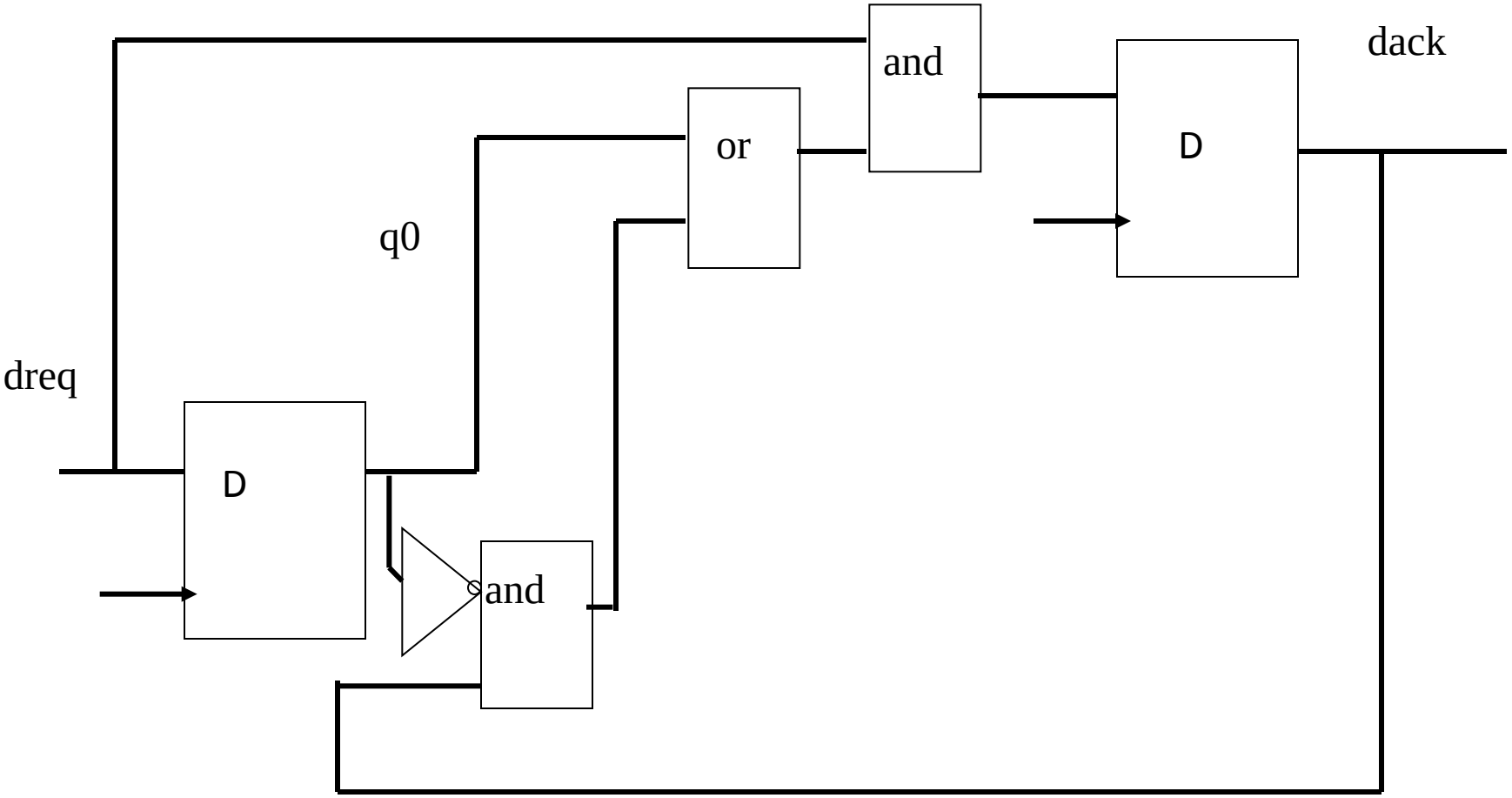

and
dack
or
D
q0
dreq
D
and

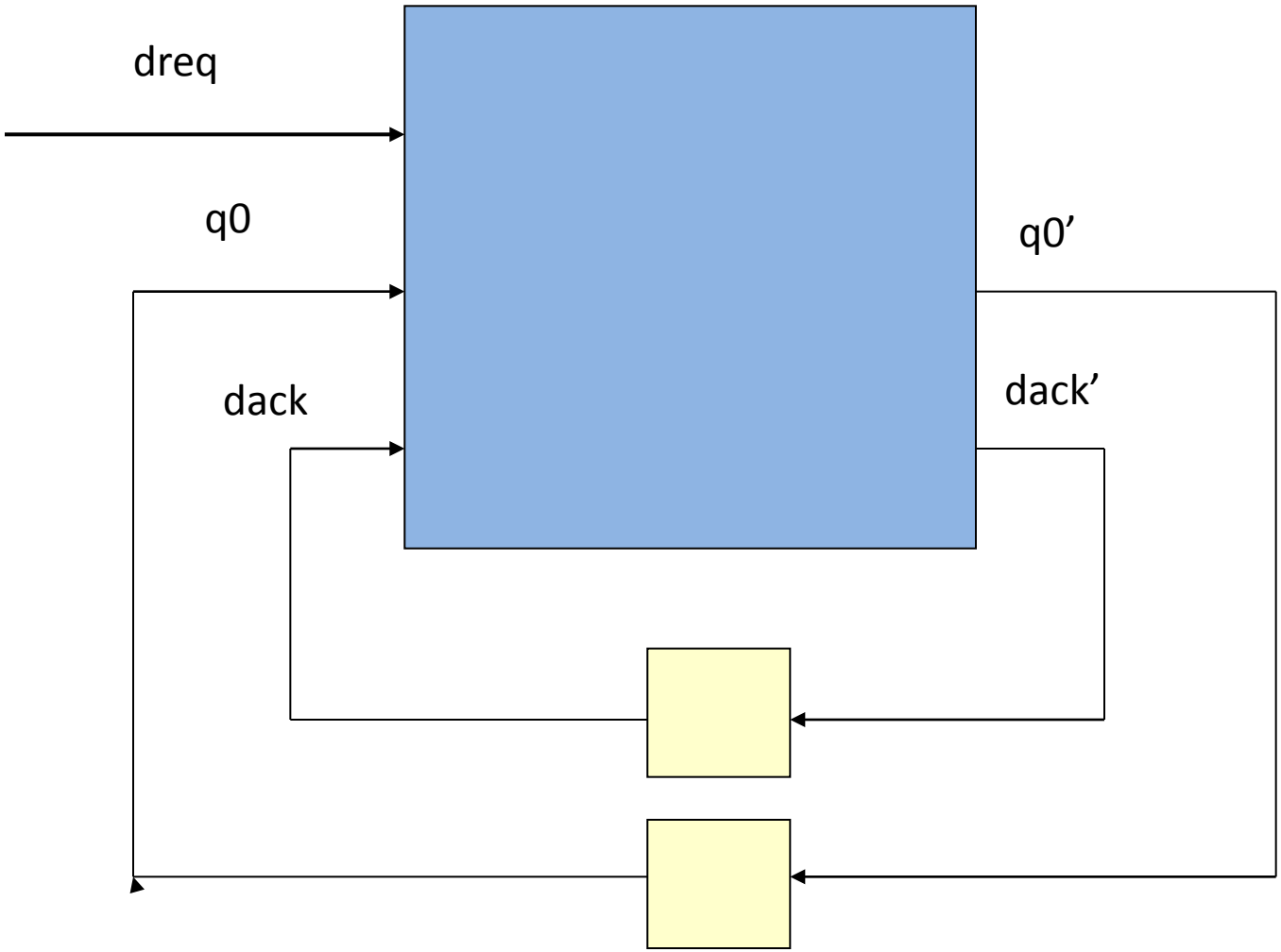
View circuit as a transition system

(dreq, q0, dack) → (dreq', q0', dack')

q0' = dreq

dack' = dreq & (q0 + (¬q0 & dack))


and
dack
or
D
q0
dreq
D
and



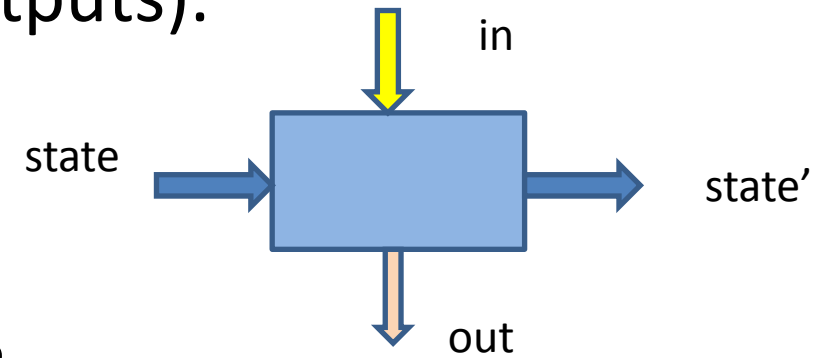
Exercise: draw the contents of the blue box

So the blue box (which contains only boolean gates) captures everything we need to know about the circuit (assuming we know which signals are states, inputs and outputs).

Sometimes pictured as

(see Magnus' slide on BMC)

Often output is a single OK wire checking a property



Can view transition relation as state -> state

(dreq, q0, dack) →

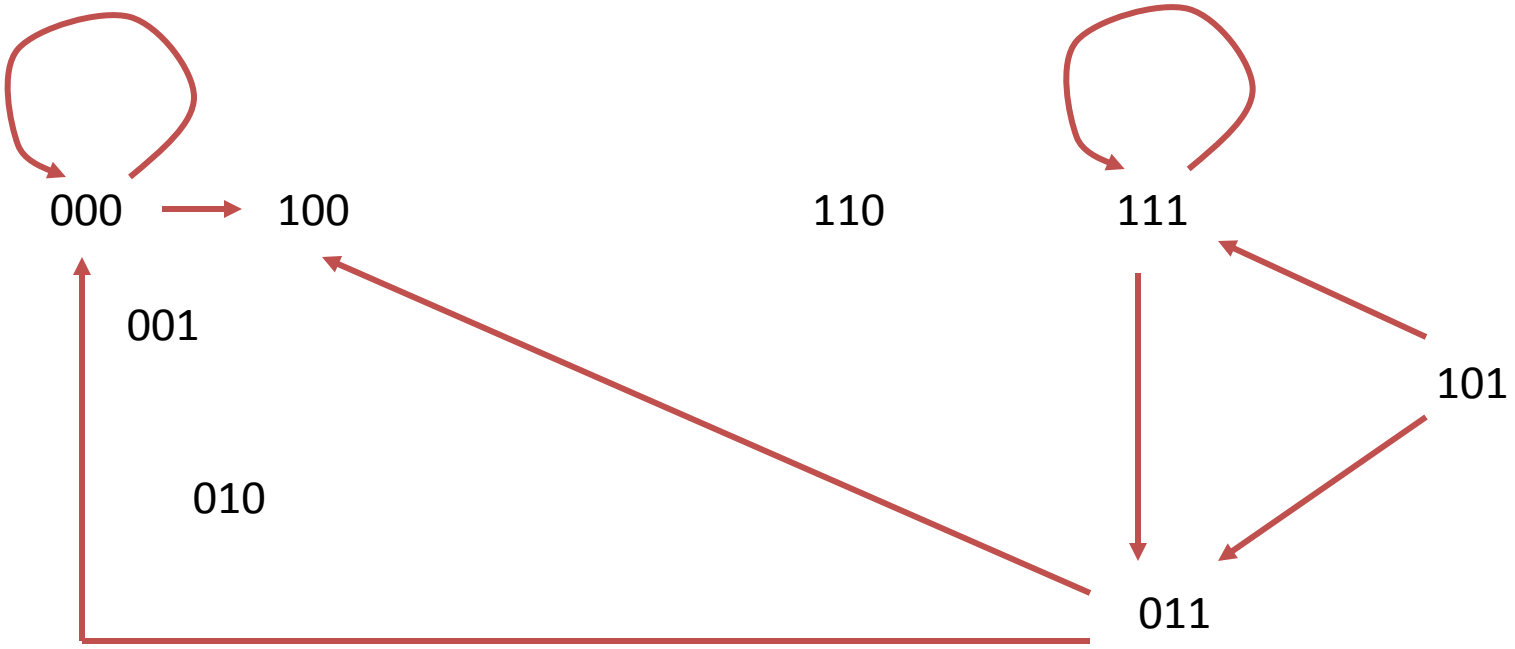
(dreq', dreq, dreq & (q0 || (¬q0 & dack)))

Exercise:

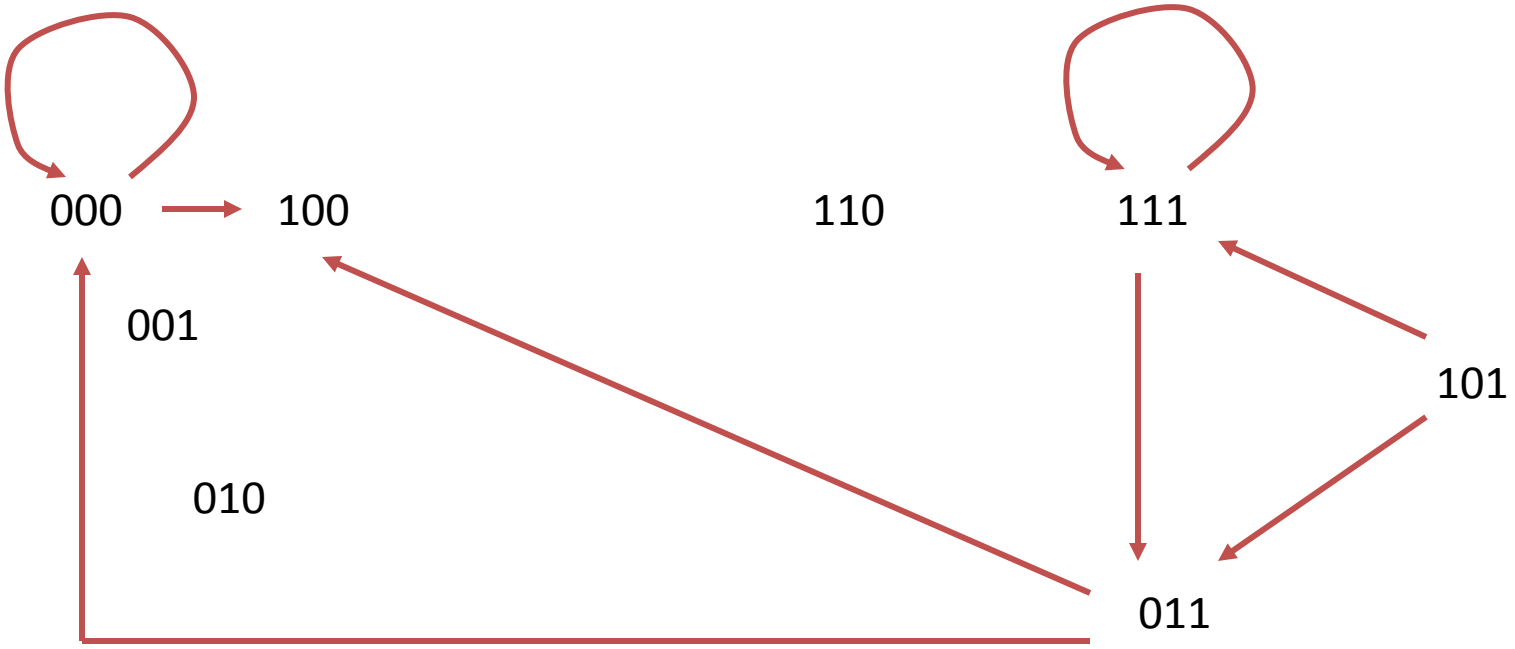
Draw state transition diagram

Q: How many states for a start?

Hint (partial answer)

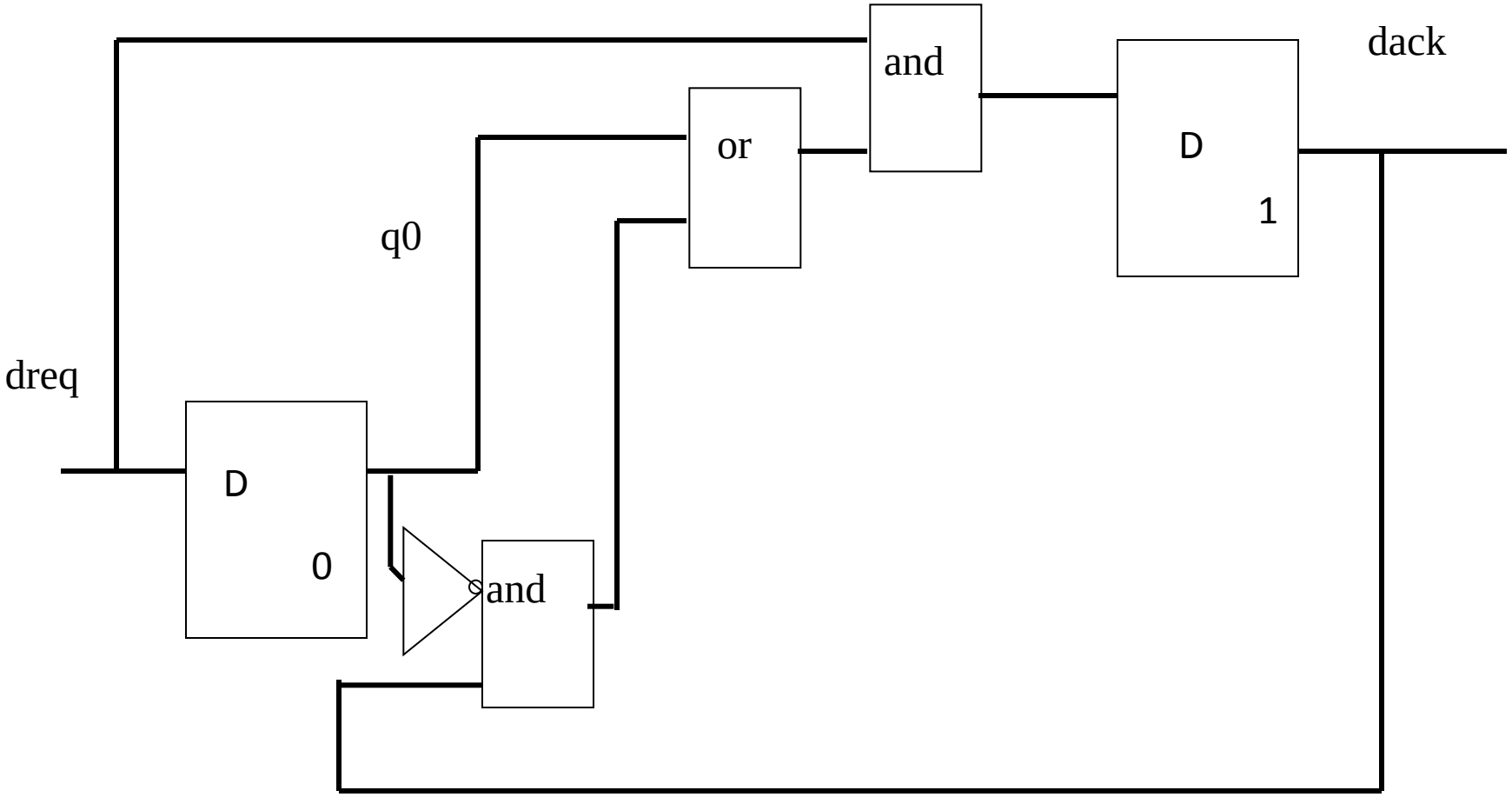


Question

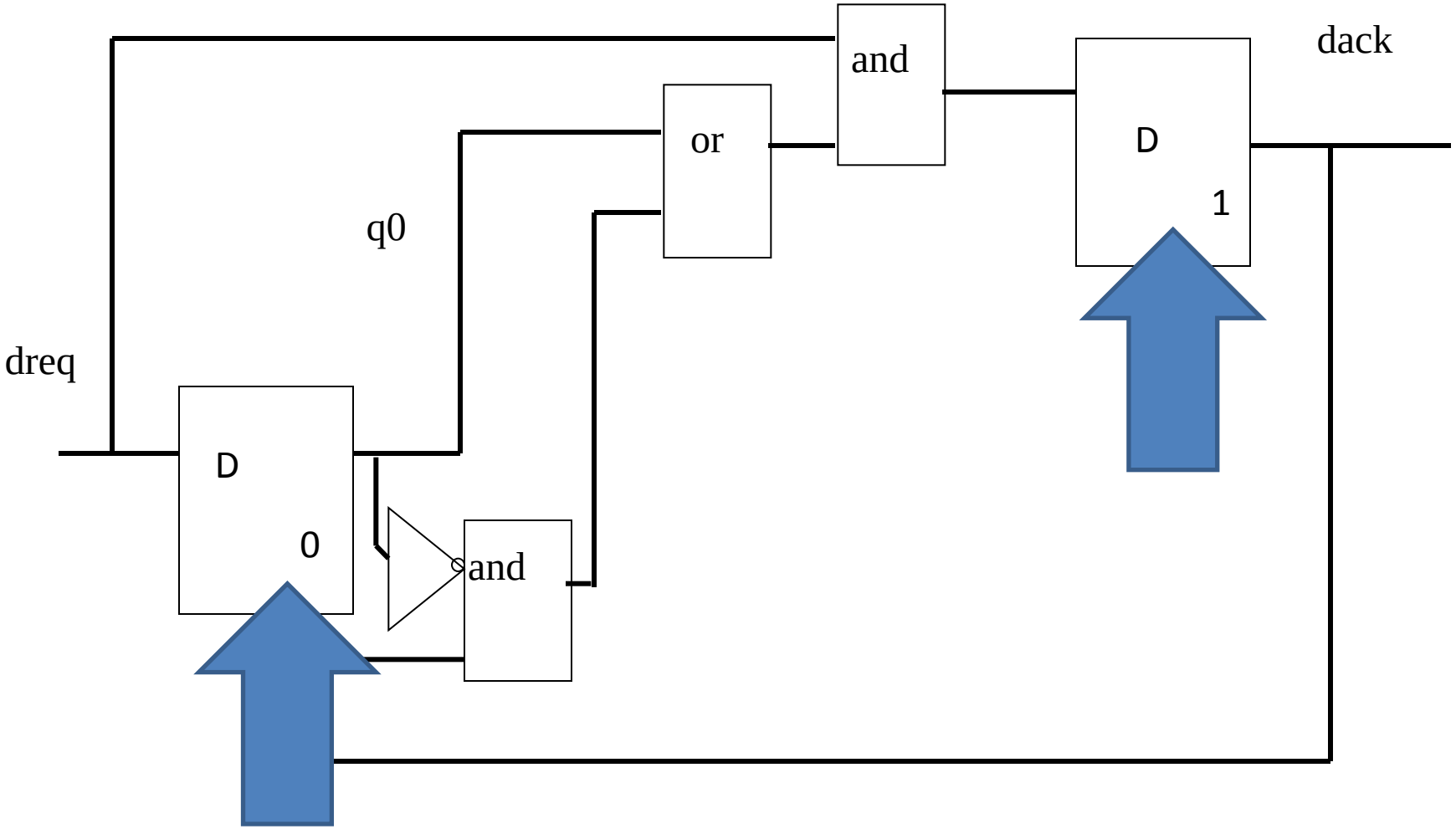


How many arrows should there be out of each state? Why so?
(Complete the diagram in spare time)

Formal tools operate on Netlists



Formal tools operate on Netlists



input to SMV model checker

```
MODULE main
VAR w1 : boolean;
VAR w2 : boolean;
VAR w3 : boolean;
VAR w4 : boolean;
VAR w5 : boolean;
VAR i0 : boolean;
VAR w6 : boolean;
VAR w7 : boolean;
VAR w8 : boolean;
VAR w9 : boolean;
VAR w10 : boolean;
DEFINE w4 := 0;
DEFINE w5 := i0;
ASSIGN init(w3) := w4;
ASSIGN next(w3) := w5;
DEFINE w7 := !(w3);
DEFINE w9 := 1;
DEFINE w10 := w5 & w6;
ASSIGN init(w8) := w9;
ASSIGN next(w8) := w10;
DEFINE w6 := w7 & w8;
DEFINE w2 := w3 | w6;
```

MC builds internal representation of transition system

We reason about models of circuits

Circuits we will design and verify are synchronous with a single clock =>

very easy move to a formal representation. No worries about accuracy of modelling.

In JG as we use it, if the circuit is not at the gate level, a netlist is synthesised to enable formal reasoning (e.g. comparison with the user's gate level implementation)

We have no worries about the accuracy of our modelling

We reason about models of circuits

Real life circuits typically have multiple clocks => translation to transition system a bit more complicated with a finer view of time. Still a single abstract clock but now the real clocks are viewed as inputs.

To reason about finer details of circuits (e.g. gate delays) then a finer degree of modelling is used to get to the transition system. More delay elements in the model. No longer one to one match.

Lots of real life circuits also break the rules (e.g. with clocks that depend on each other), leading to special hacks in the formal tools

It is also possible (but much harder) to reason about analogue or asynchronous circuits

Questions?

Key ideas 1: Binary Decision Diagrams

Vital enabling technology (along with SAT solving)

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

Canonical form (constant time comparison)

Used in Symbolic Model Checking

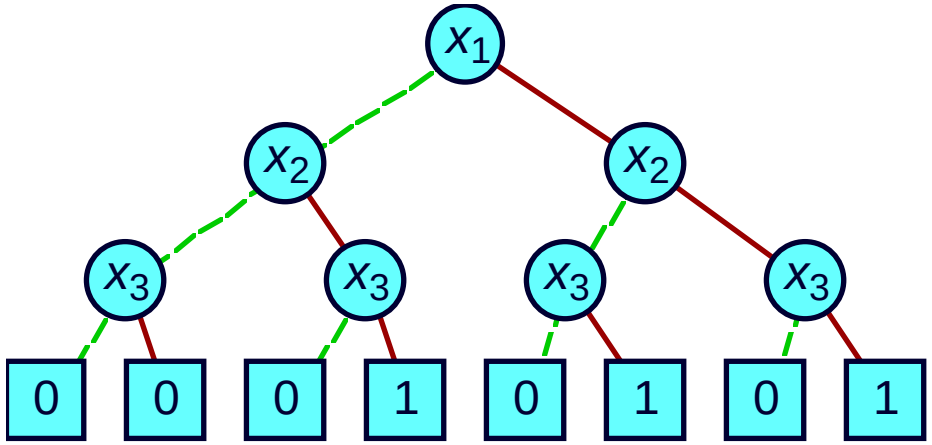
Following slides are by Bryant (used with thanks!)

Decision Structures

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

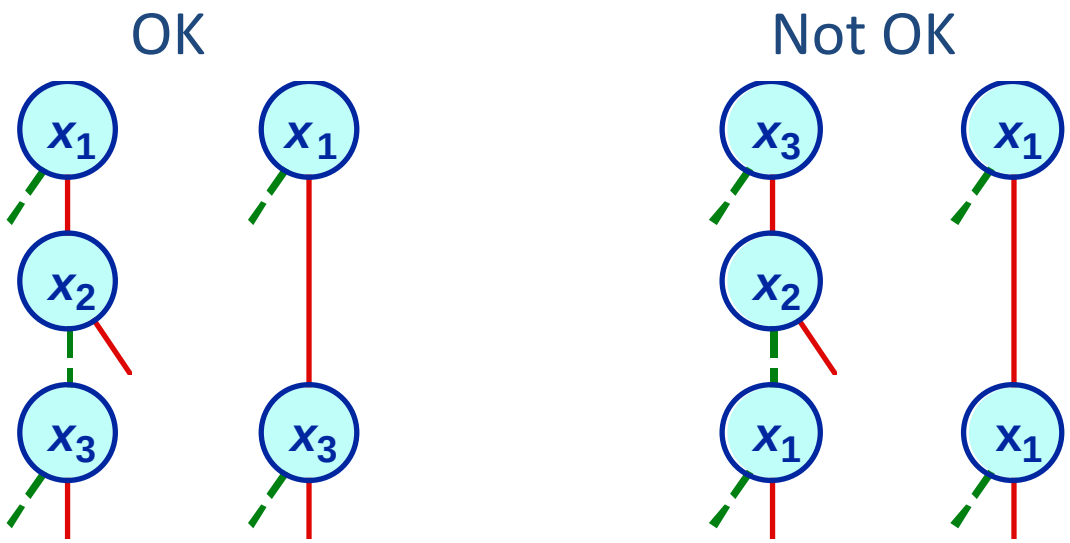
Decision Tree



- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.

Variable Ordering

- Assign arbitrary total ordering to variables
  - e.g., $x_1 < x_2 < x_3$
- Variables must appear in ascending order along all paths

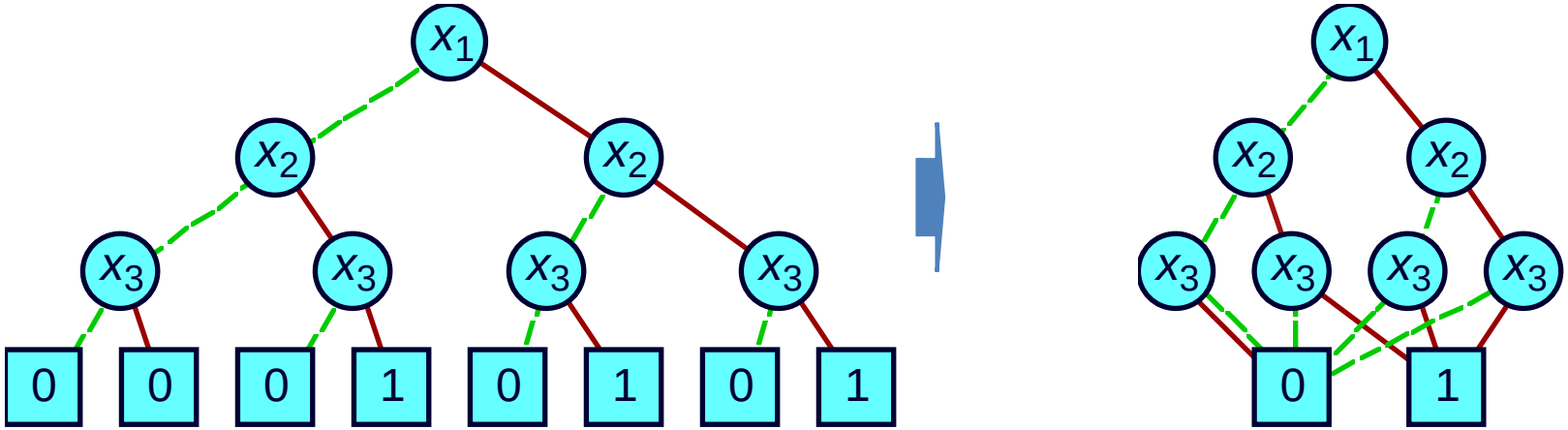


Properties

- No conflicting variable assignments along path
- Simplifies manipulation

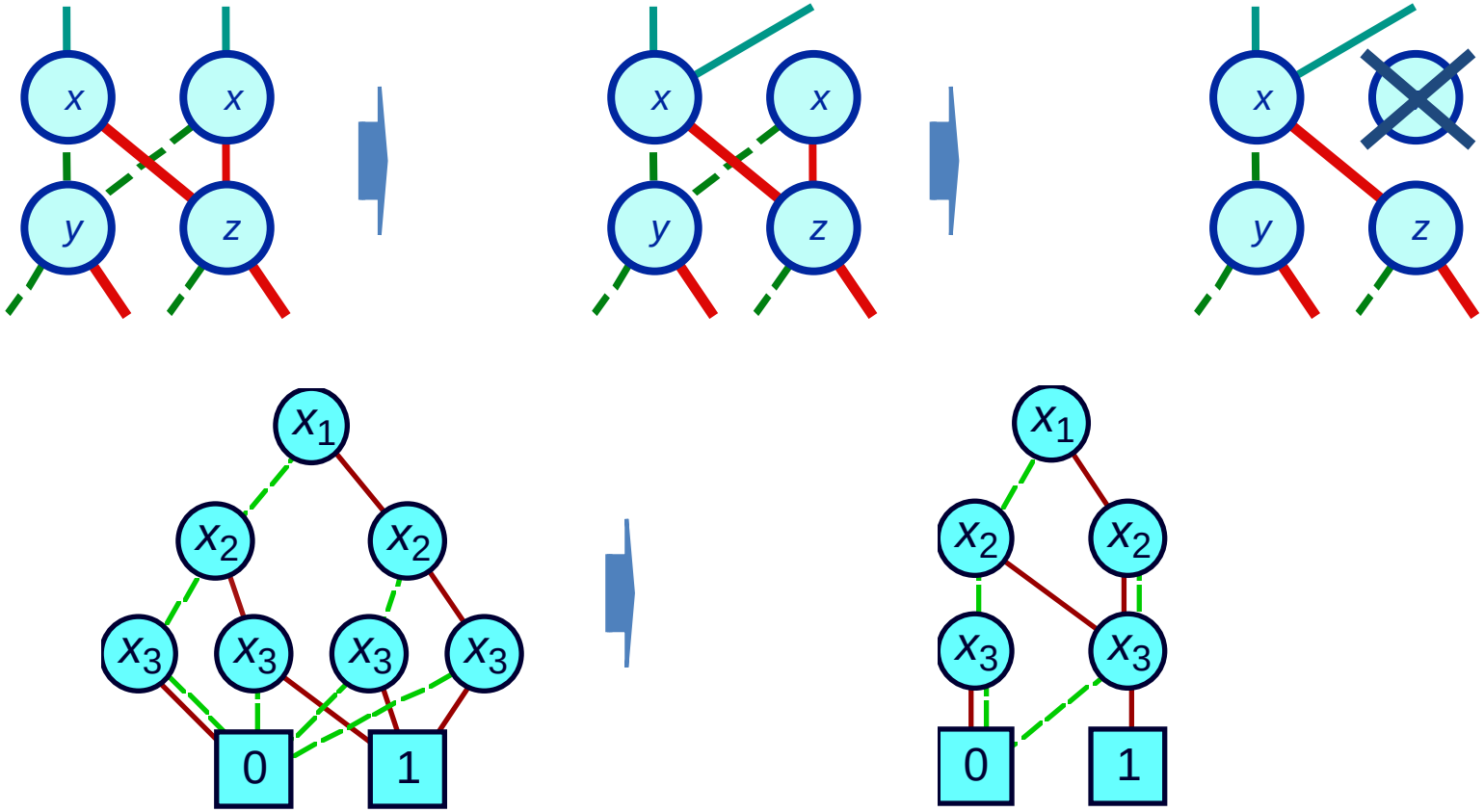
Reduction Rule #1

Merge equivalent leaves



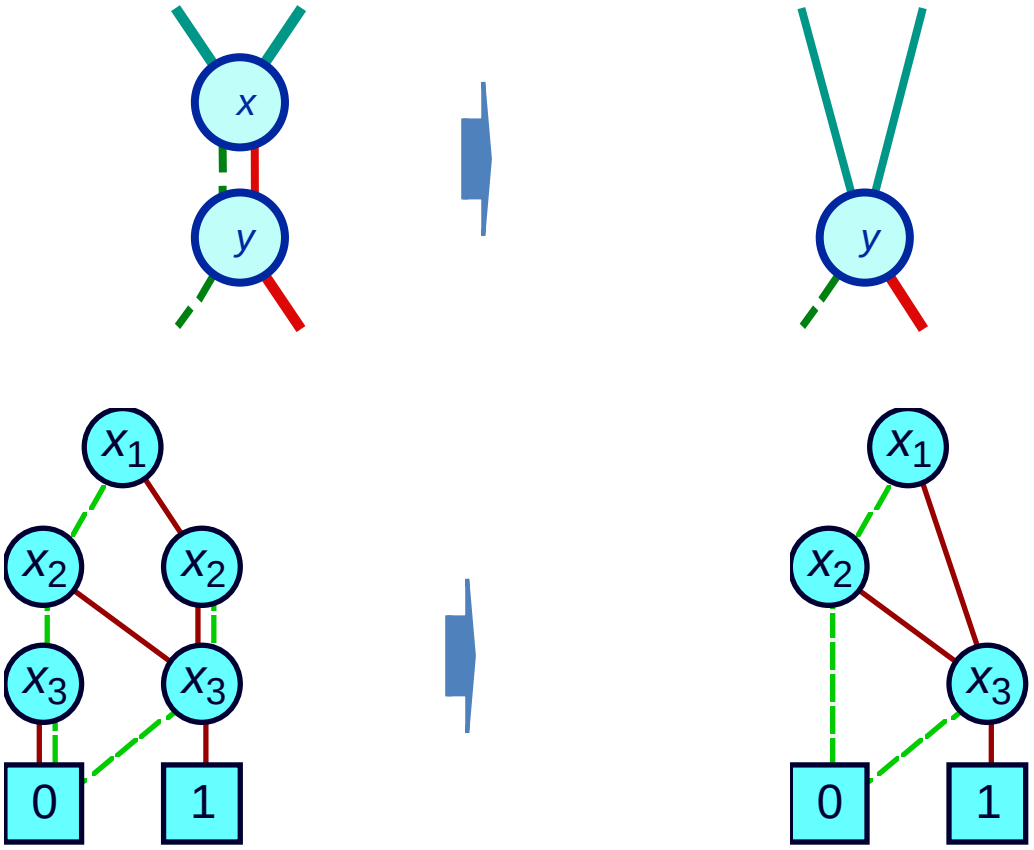
Reduction Rule #2

Merge isomorphic nodes

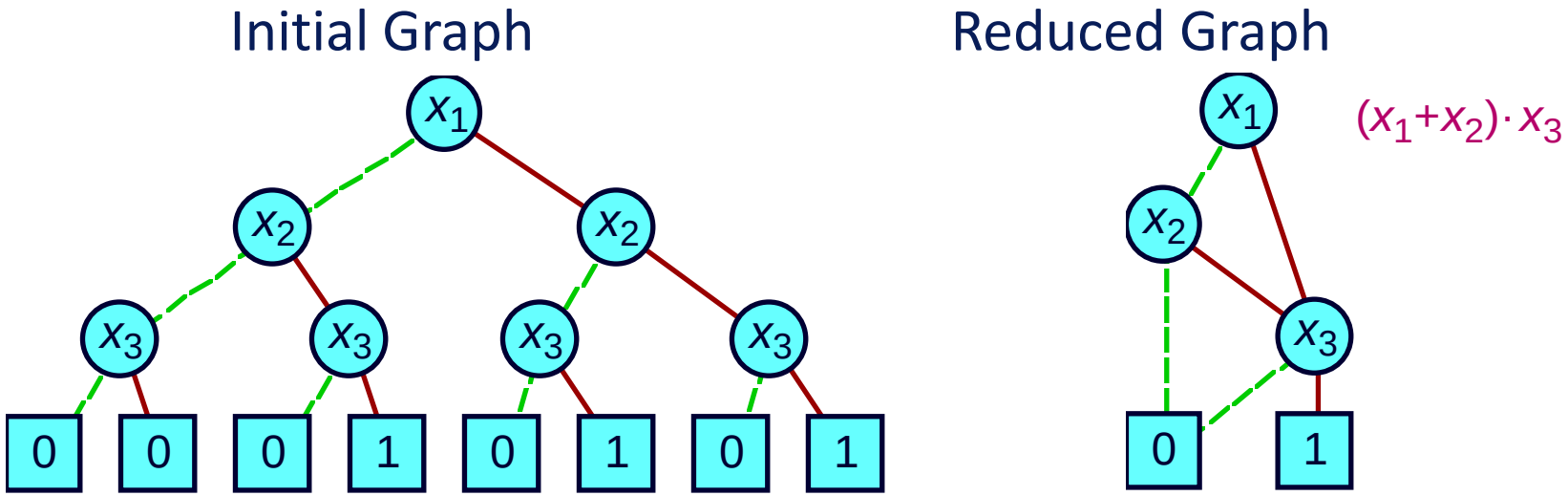


Reduction Rule #3

Eliminate Redundant Tests



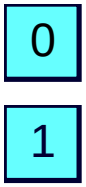
Example OBDD



- Canonical representation of Boolean function
  - ☐ For given variable ordering
  - – Two functions equivalent if and only if graphs isomorphic
    - Can be tested in linear time
  - – Desirable property: *simplest form is canonical*.

Example Functions

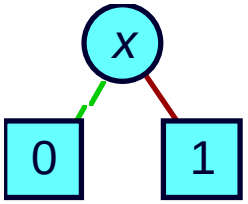
Constants



Unique unsatisfiable function

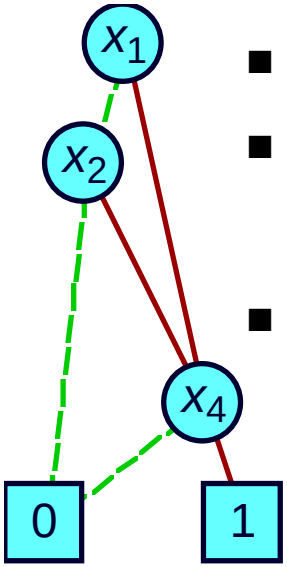
Unique tautology

Variable



Treat variable as function

Typical Function



- $(x_1 \vee x_2) \wedge x_4$
- No vertex labeled x_3
  - independent of x_3
- Many subgraphs shared

Exercise: make OBDD for

You an label the arcs with 0 and 1 instead of using colours

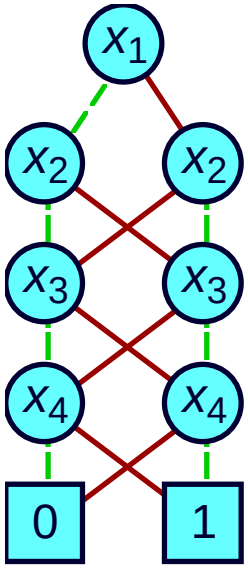
Exercise: make OBDD for

$$x \oplus y \oplus z$$

How does it look for $x1 \oplus x2 \oplus x3 \oplus x4$
and for odd parity in general?

BDD for x1 ⊕ x2 ⊕ x3 ⊕ x4

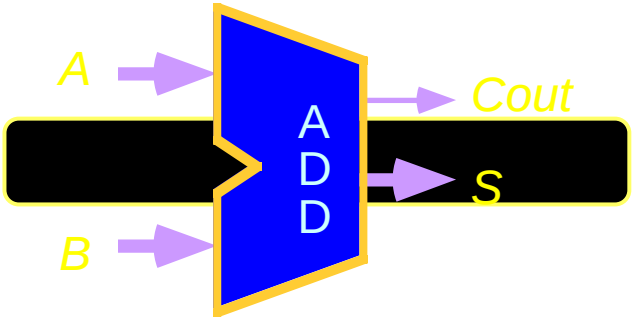
Odd Parity

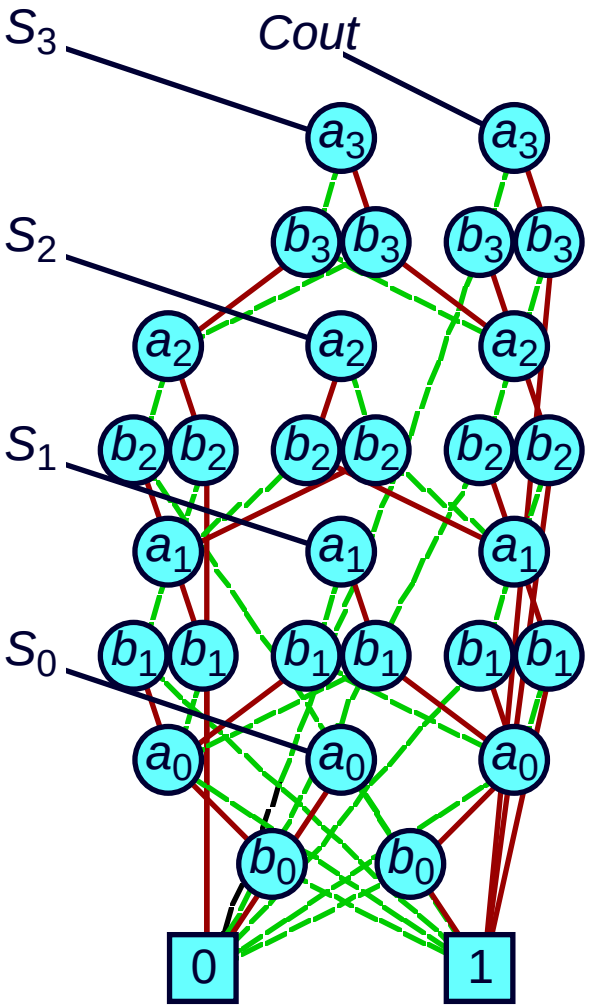


Linear representation

Representing Circuit Functions

- Functions
  - All outputs of 4-bit adder
  - Functions of data inputs





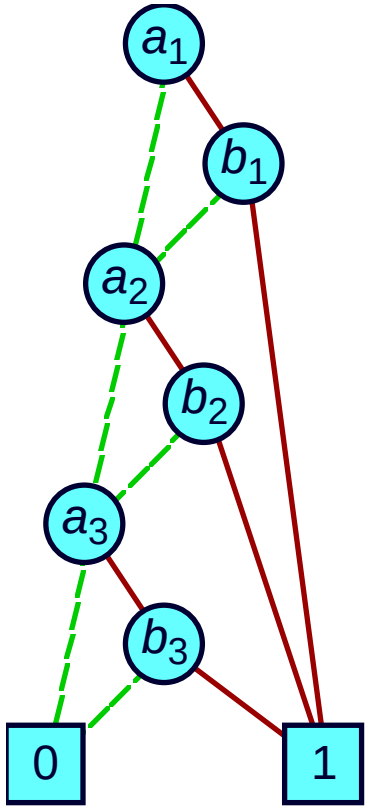
Shared Representation

- Graph with multiple roots
- 31 nodes for 4-bit adder
- 571 nodes for 64-bit adder
  - *Linear growth*

Effect of Variable Ordering

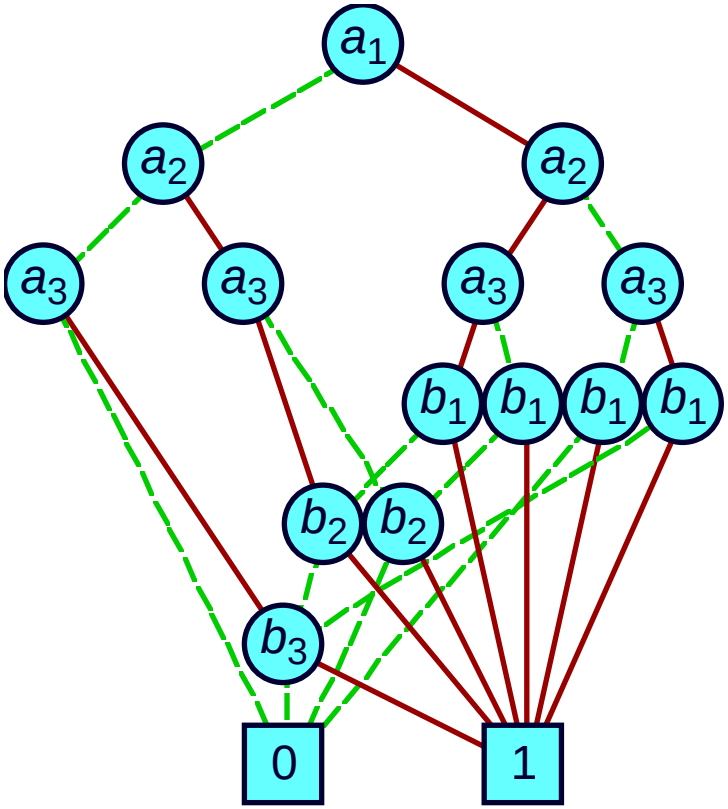
$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$$

Good Ordering



Linear Growth

Bad Ordering



Exponential Growth

Selecting Good Variable Ordering

- Intractable Problem
  - Even when problem represented as OBDD
    - I.e., to find optimum improvement to current ordering
- Application-Based Heuristics
  - Exploit characteristics of application
  - E.g., Ordering for functions of combinational circuit
    - Traverse circuit graph depth-first from outputs to inputs
    - Assign variables to primary inputs in order encountered

Dynamic Variable Reordering

  - Richard Rudell, Synopsys
- Periodically Attempt to Improve Ordering for All BDDs
  - Part of garbage collection
  - Move each variable through ordering to find its best location
- Has Proved Very Successful
  - Time consuming but effective
  - Especially for sequential circuit analysis

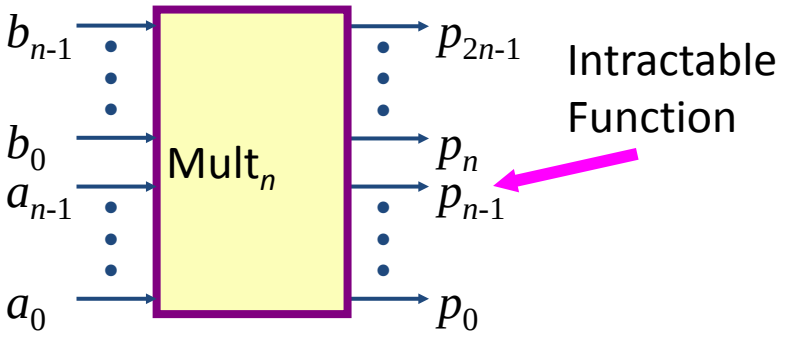
Sample Function Classes

Function Class	Best	Worst	Ordering Sensitivity
ALU (Add/Sub)	linear	exponential	High
Symmetric	linear	quadratic	None
Multiplication	exponential	exponential	Low

- General Experience
  - Many tasks have reasonable OBDD representations
  - Algorithms remain practical for up to 100,000 node OBDDs (Note from MS: remember this was written in 1999)
  - Heuristic ordering methods generally satisfactory

Lower Bound for Multiplication

- Bryant, 1991
- Integer Multiplier Circuit
  - *n*-bit input words *A* and *B*
  - 2*n*-bit output word *P*
- Boolean function
  - Middle bit (*n*-1) of product
- Complexity
  - Exponential OBDD for all possible variable orderings



Actual Numbers

- 40,563,945 BDD nodes to represent all outputs of 16-bit multiplier
- Grows 2.86x per bit of word size

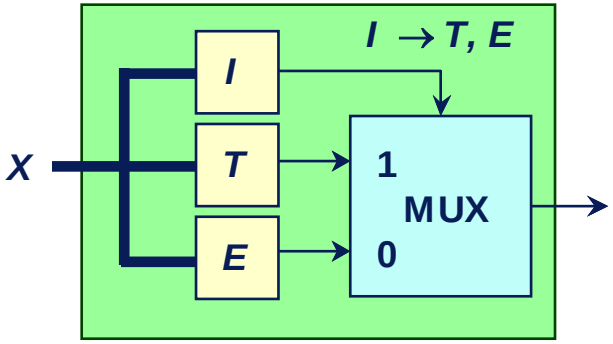
Symbolic Manipulation with OBDDs

- Strategy
  - Represent data as set of OBDDs
    - Identical variable orderings
  - Express solution method as sequence of symbolic operations
  - Implement each operation by OBDD manipulation
- Algorithmic Properties
  - Arguments are OBDDs with identical variable orderings.
  - Result is OBDD with same ordering.
  - “Closure Property”
- Contrast to Traditional Approaches
  - Apply search algorithm directly to problem representation
    - E.g., search for satisfying truth assignment to Boolean expression.

If-Then-Else Operation

Concept

- Basic technique for building OBDD from logic network or formula.



Arguments *I*, *T*, *E*

- Functions over variables *X*
- Represented as OBDDs

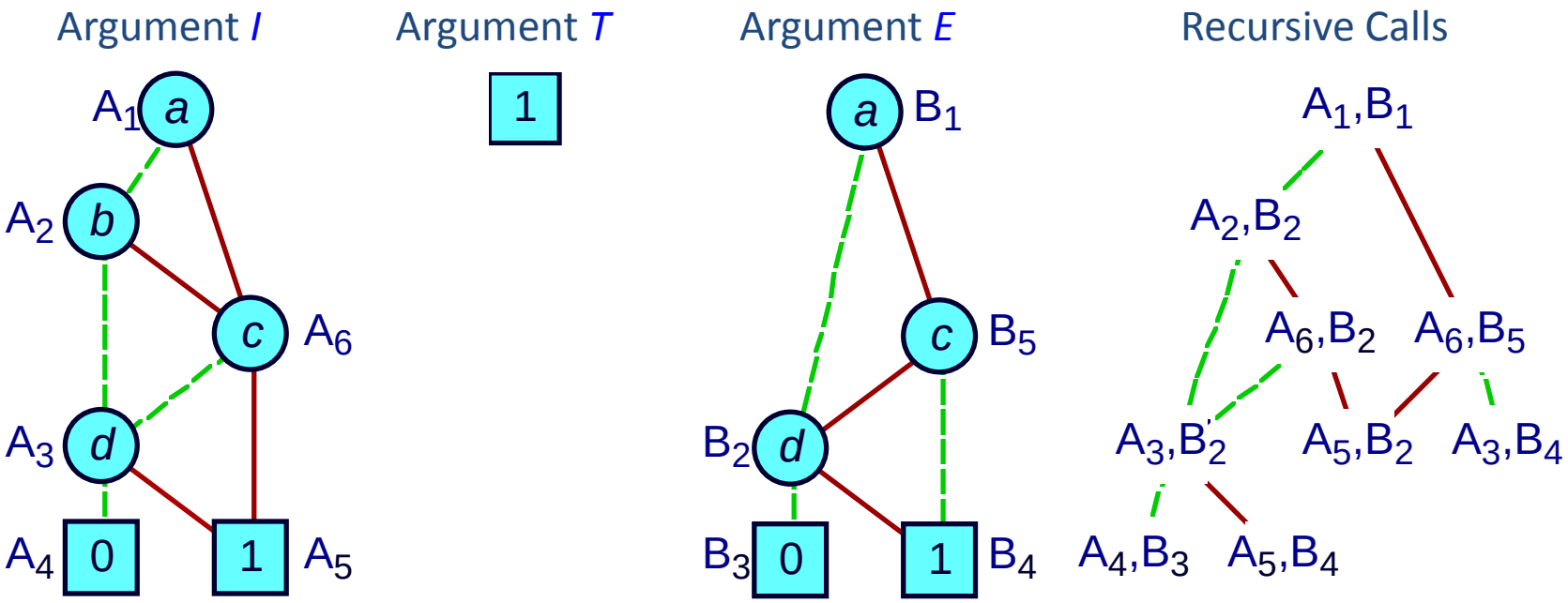
Result

- OBDD representing composite function
- $(I \wedge T) \vee (\neg I \wedge E)$

Implementation

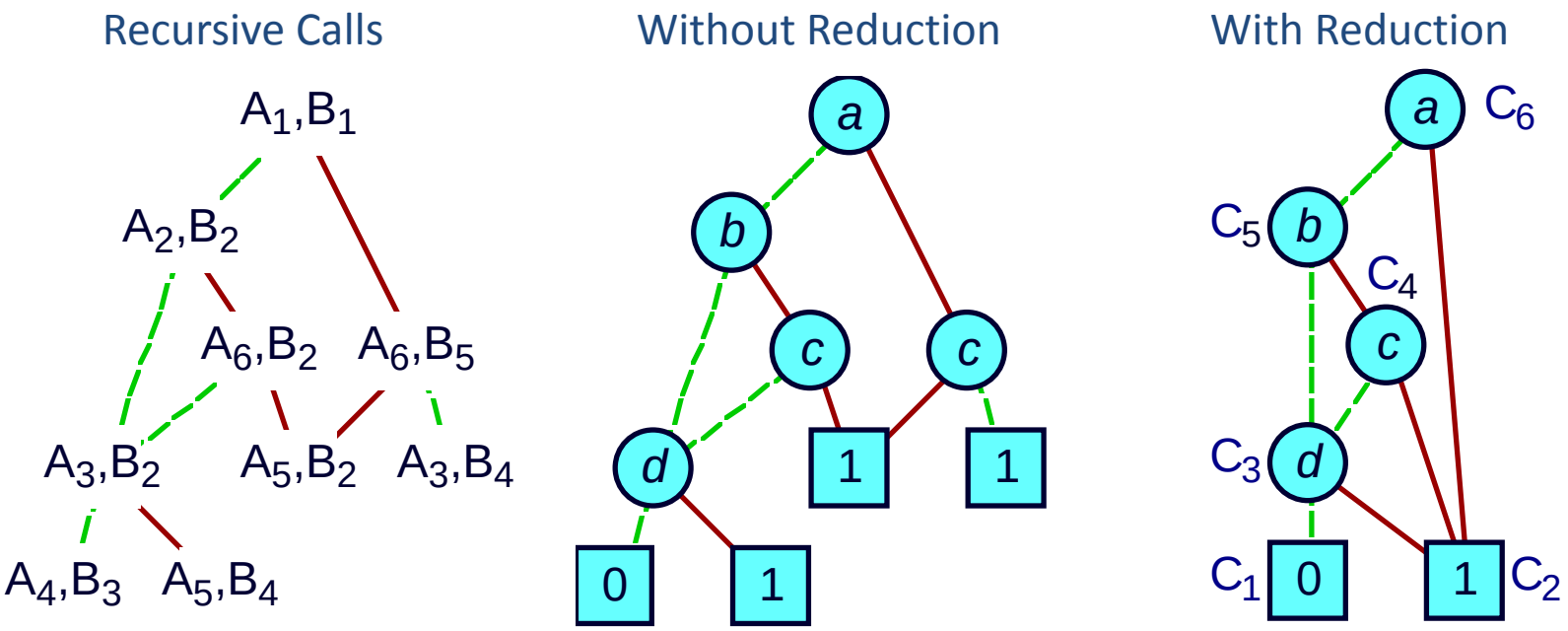
- Combination of depth-first traversal and dynamic programming.
- Worst case complexity product of argument graph sizes.

If-Then-Else Execution Example



- Optimizations
  - Dynamic programming
  - Early termination rules

If-Then-Else Result Generation



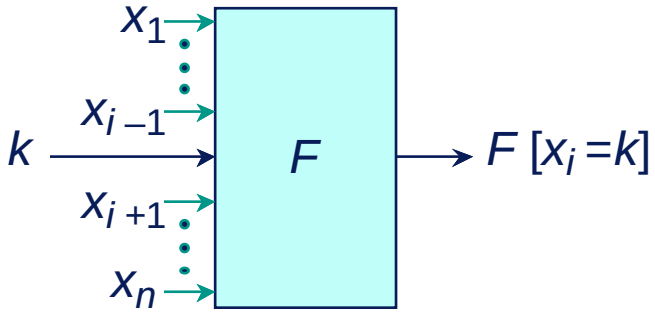
- Recursive calling structure implicitly defines unreduced BDD
- Apply reduction rules bottom-up as return from recursive calls
  - Generates reduced graph

Restriction Operation

- Concept
  - Effect of setting function argument x_i to constant k (0 or 1).
  - Also called Cofactor operation (UCB)

F_X equivalent to $F\,[x = 1]$

$F_{\overline{X}}$ equivalent to $F\,[x = 0]$



Implementation

- Depth-first traversal.
- Complexity near-linear in argument graph size

Derived Operations

Express as combination of If-Then-Else and Restrict

Preserve closure property

- Result is an OBDD with the right variable ordering

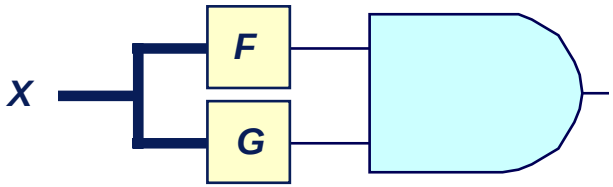
Polynomial complexity

- Although can sometimes improve with special implementations

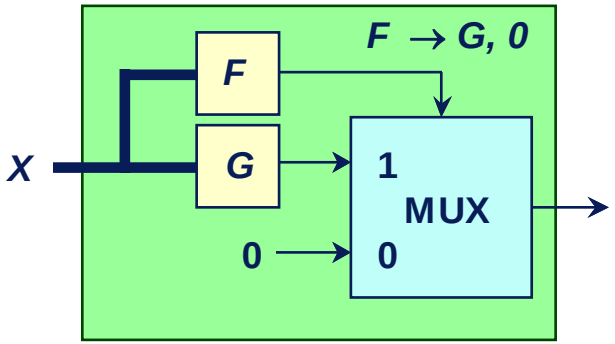
Derived Algebraic Operations

– Other operations can be expressed in terms of If-Then-Else

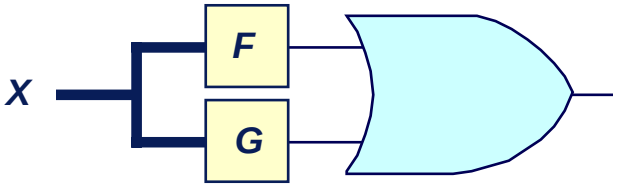
And(*F*, *G*)



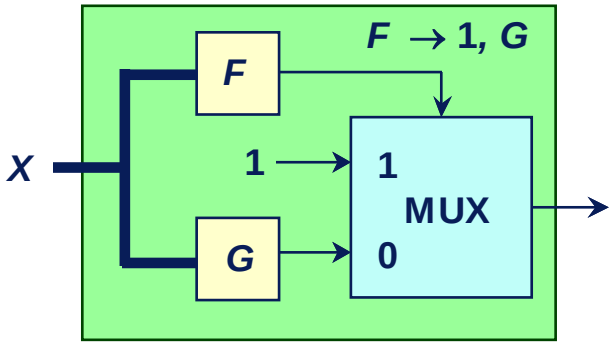
If-Then-Else(*F*, *G*, 0)



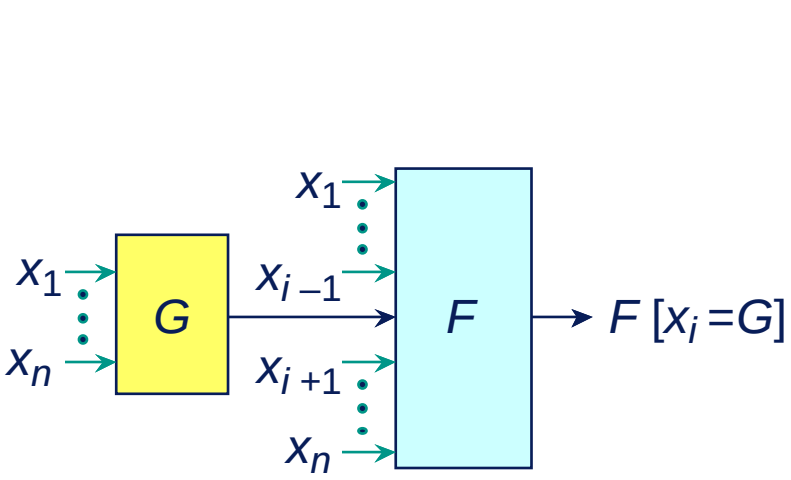
Or(*F*, *G*)

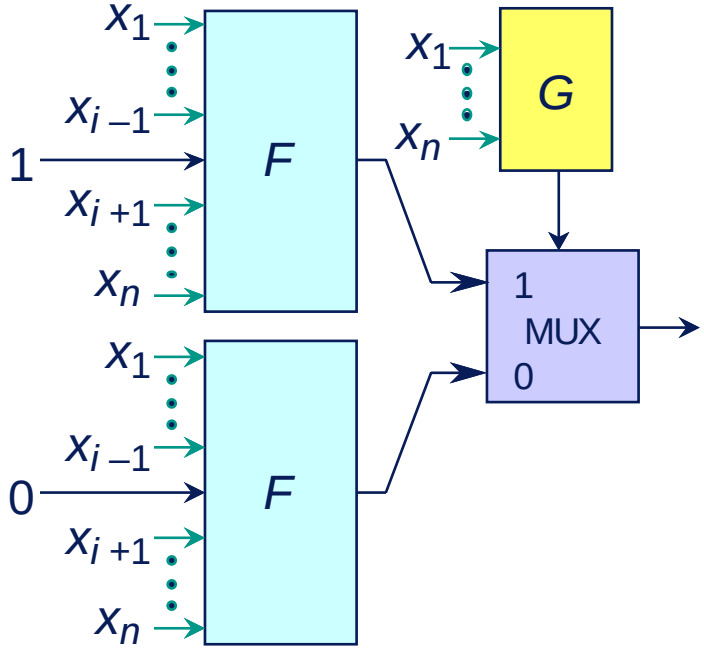


If-Then-Else(*F*, 1, *G*)



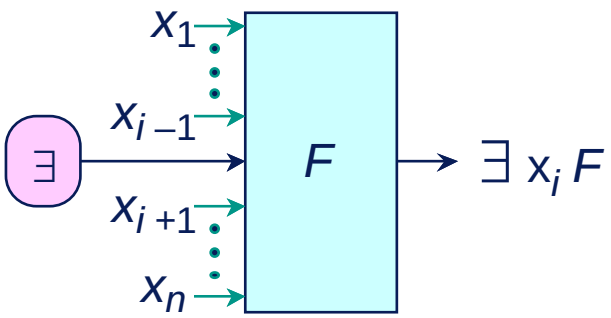
Functional Composition

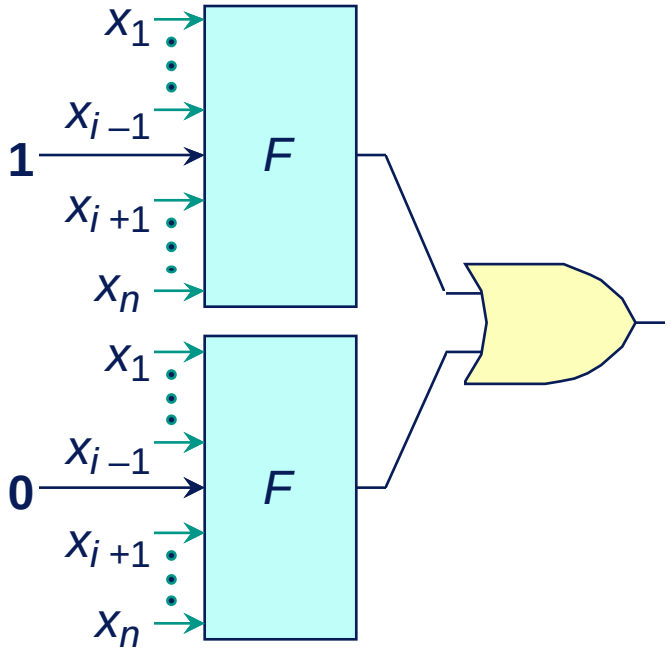




- Create new function by composing functions F and G.
- Useful for composing hierarchical modules.

Variable Quantification



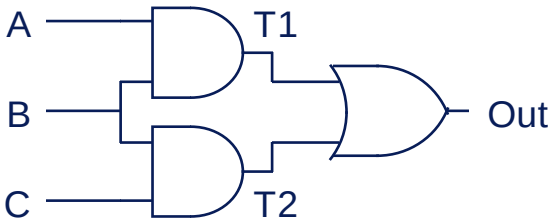


- Eliminate dependency on some argument through quantification
- Combine with AND for universal quantification.

Generating OBDD from Network

Task: Represent output functions of gate network as OBDDs.

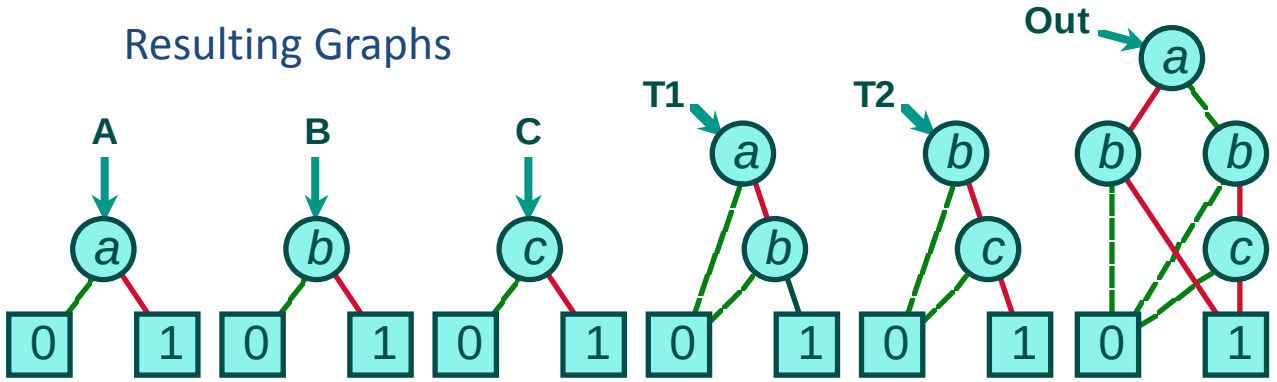
Network



Evaluation

- A ← new_var ("a");
- B ← new_var ("b");
- C ← new_var ("c");
- T1 ← And (A, 0, B);
- T2 ← And (B, C);
- Out ← Or (T1, T2);

Resulting Graphs

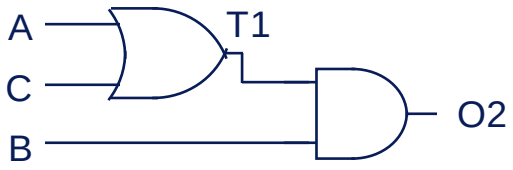


Checking Network Equivalence

Task: Do two networks compute same Boolean function?

Method: Compute OBDDs for both networks and compare

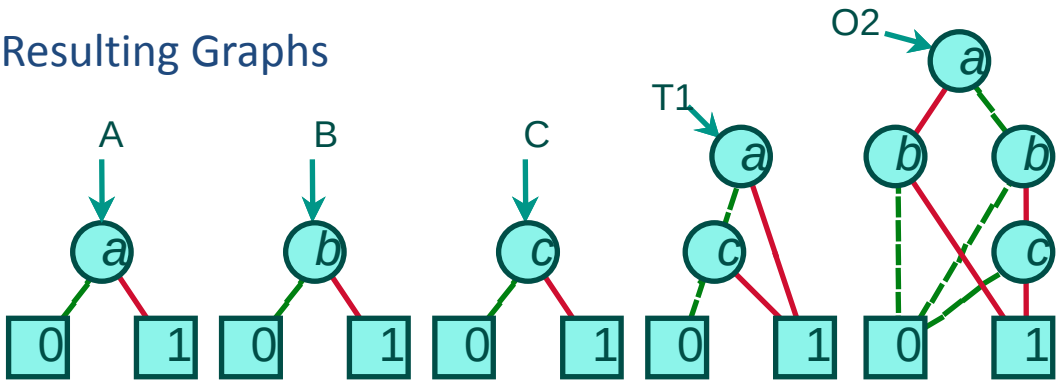
Alternate Network



Evaluation

```
T1  ← Or (A, C);
O2  ← And (T1, B);
if (O2 == Out)
        then Equivalent
        else Different
```

Resulting Graphs



end of Bryant's slides, with thanks

First form of FV
Equivalence Checking (EC,CEC)

Boolean network comparison, also known as combinational equivalence checking

Straight BDD comparison works for moderately sized circuits. For larger circuits, more sophisticated methods are used.

Invisible to user, automatic, effective

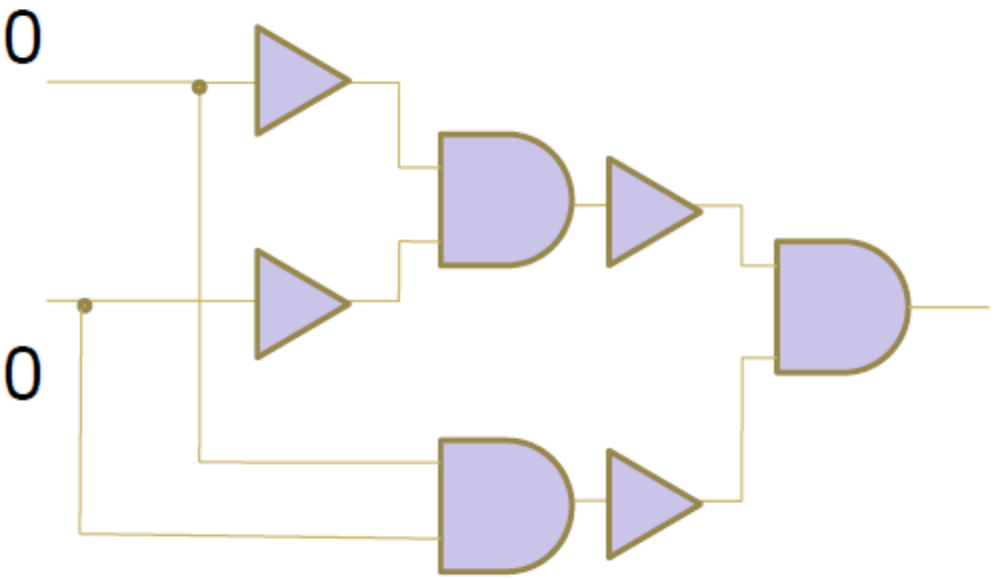
Second form of FV

Symbolic simulation

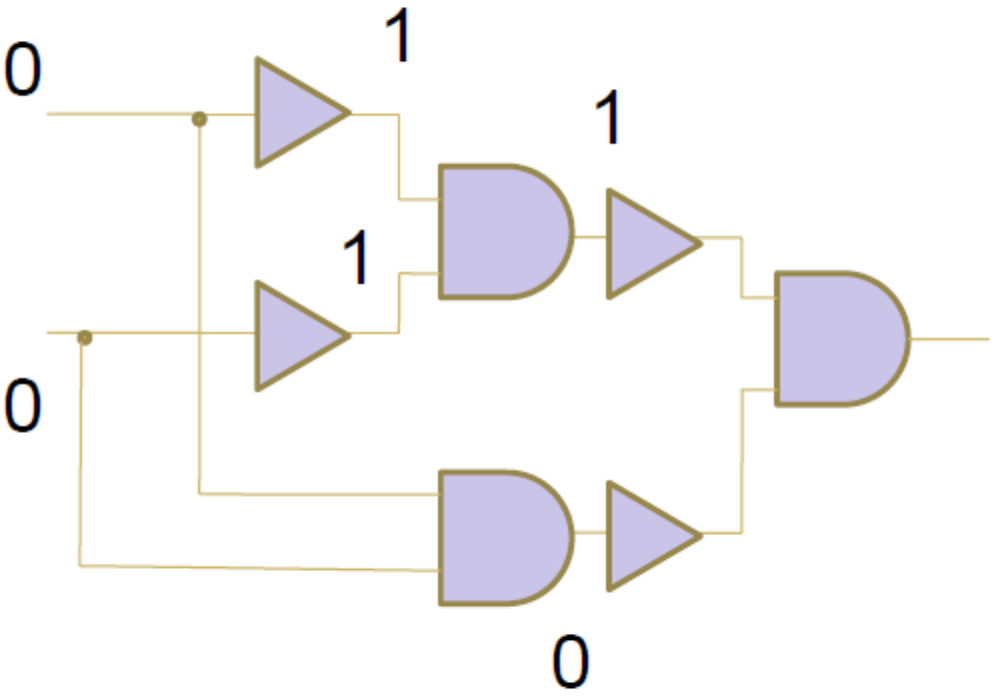
Take a simulator (can be quite low level, accurate one)

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

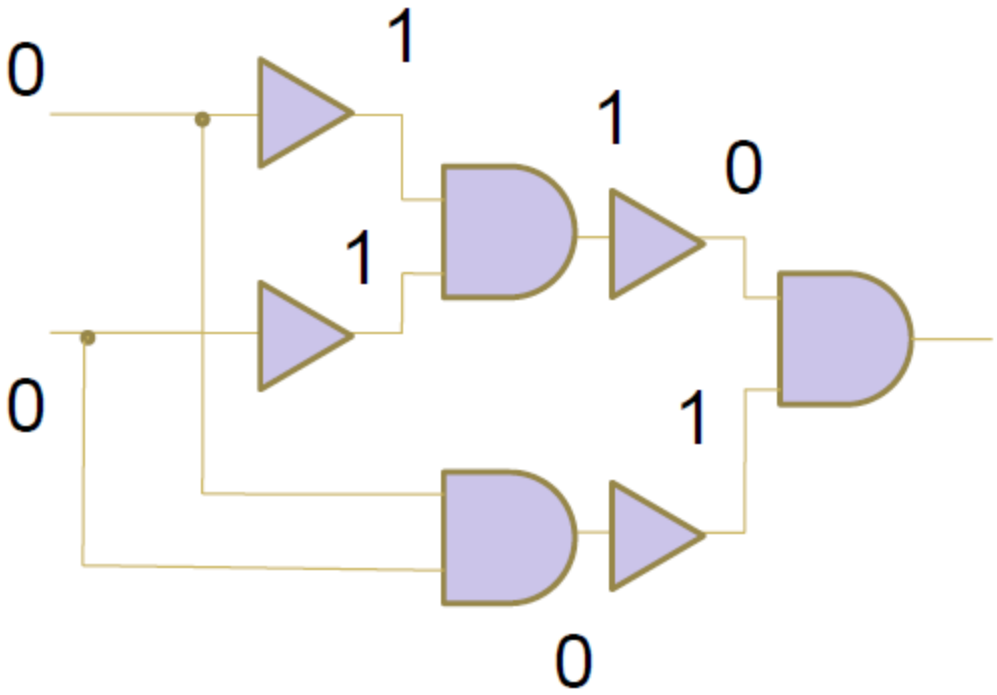
Ordinary simulation xor ?



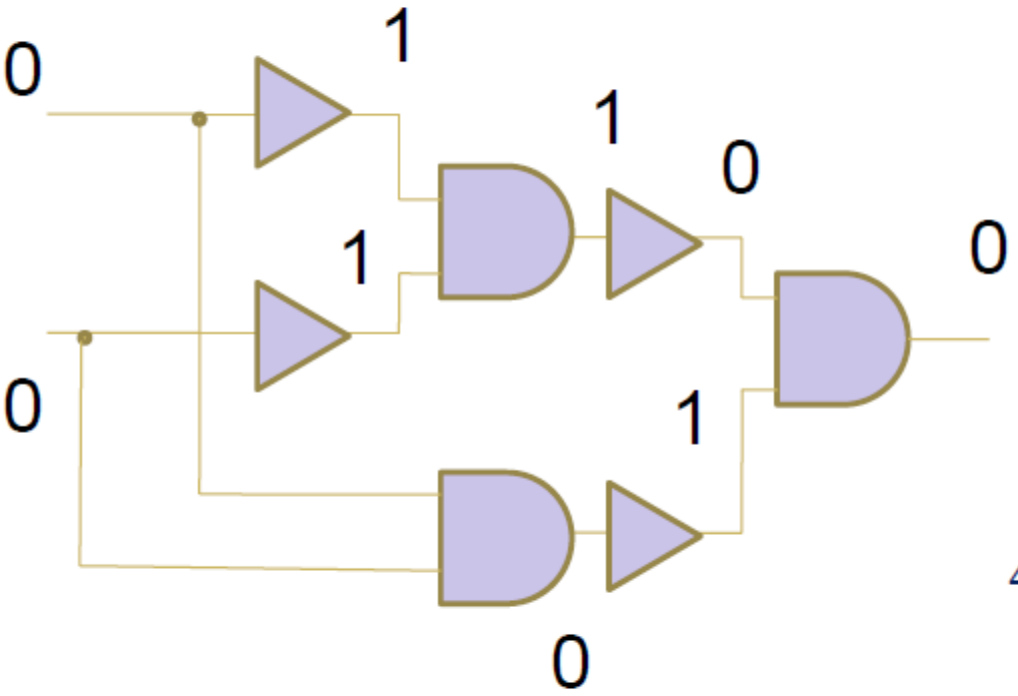
simulation



simulation



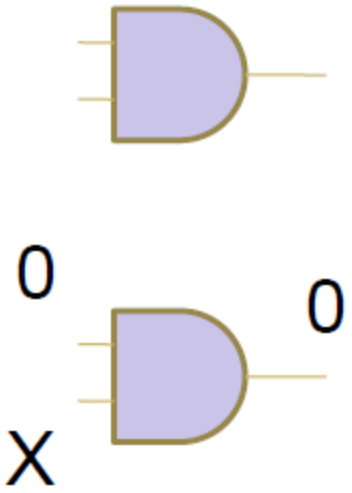
simulation



4 runs to check exhaustively

Q: how many for n inputs?

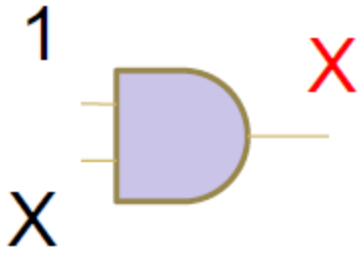
Symbolic simulation Idea 1



Use X values

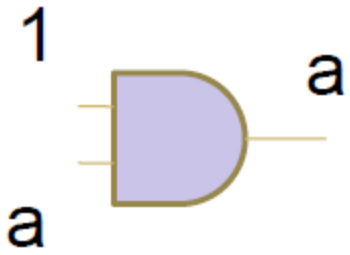
Halves number of sim. runs!

BUT may lose information



(try on xor example)

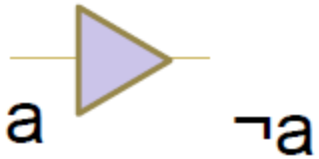
Symbolic simulation Idea 2



Use symbolic values

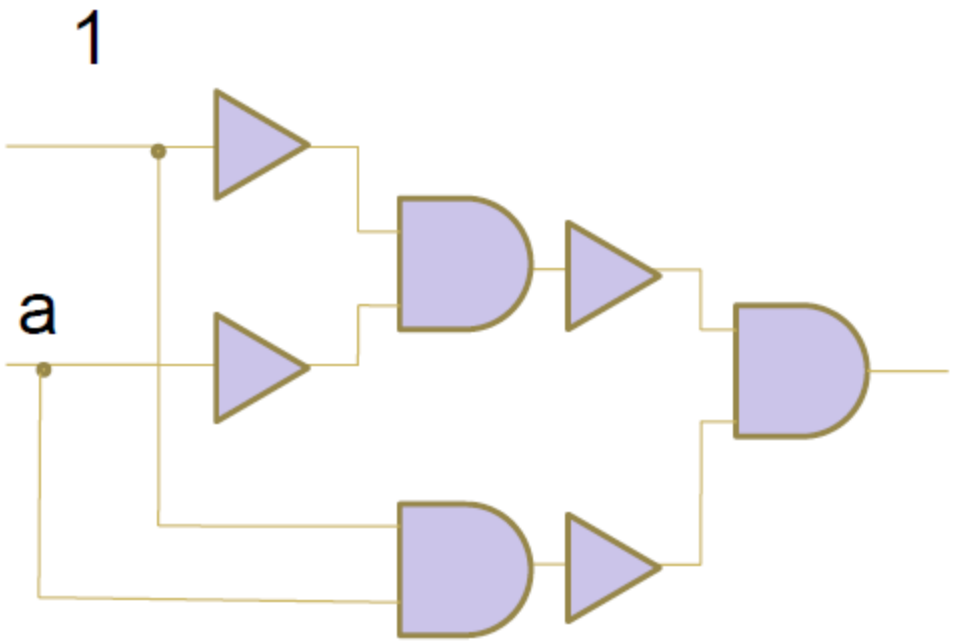
Think of giving input values names rather than constant values

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

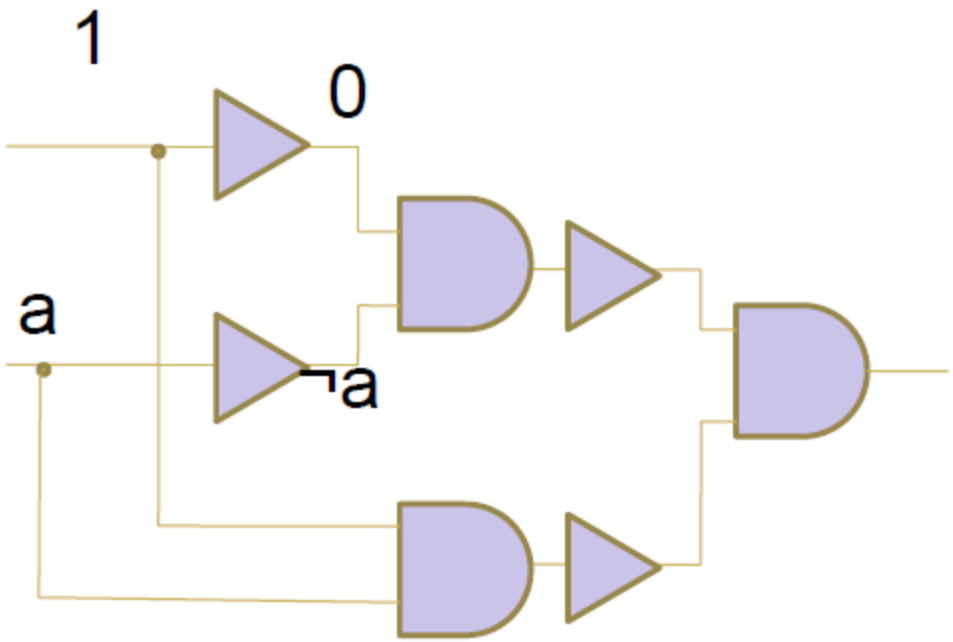


May Rep. Using Binary Decision Diagrams (BDDs)

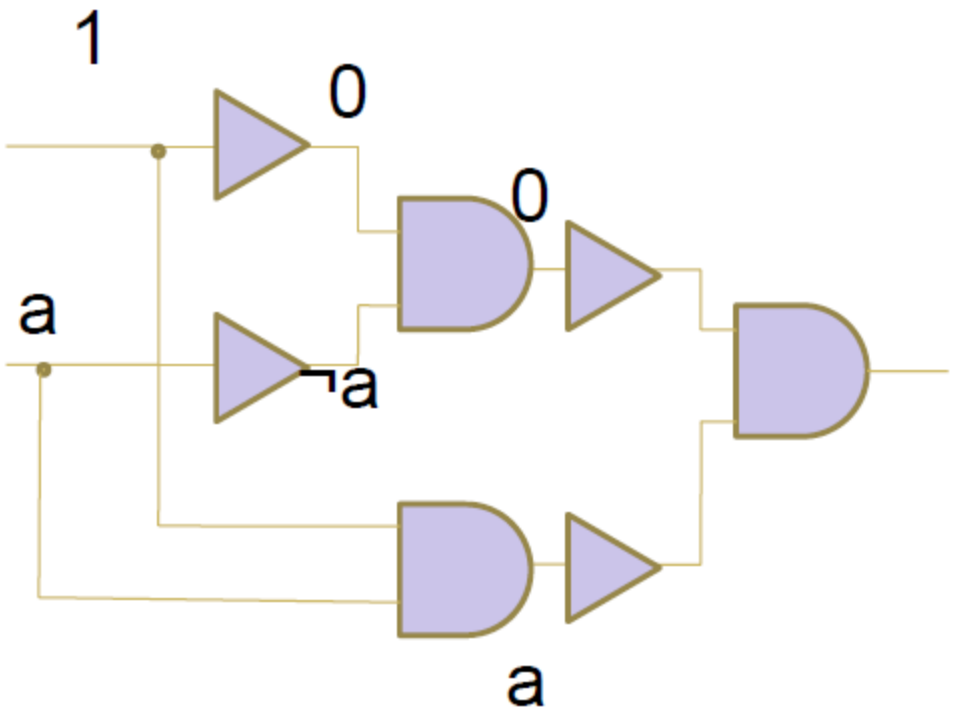
Symbolic simulation



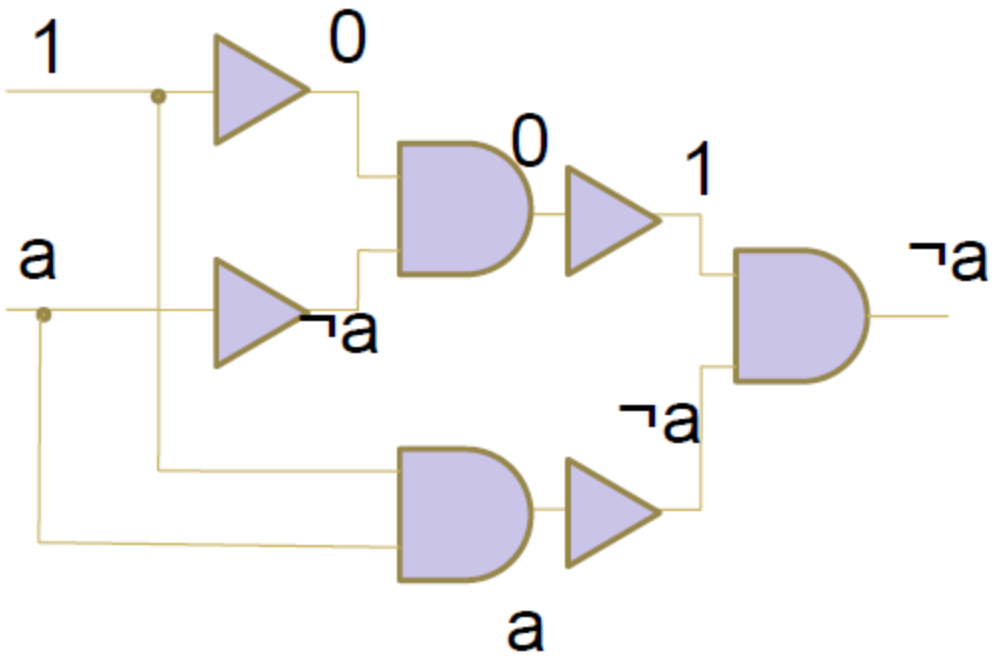
Symbolic simulation



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

Pro and Cons of BDDs

+ Powerful operations (create, manipulate,test)
  polynomial complexity, composable
+ Usually stay small enough
  given good variable order
+ Provide quantification operations (unlike plain SAT)

- sometimes explode in size
- important circuits (multipliers and shifters) are problematic => yet more special hacks in the tools

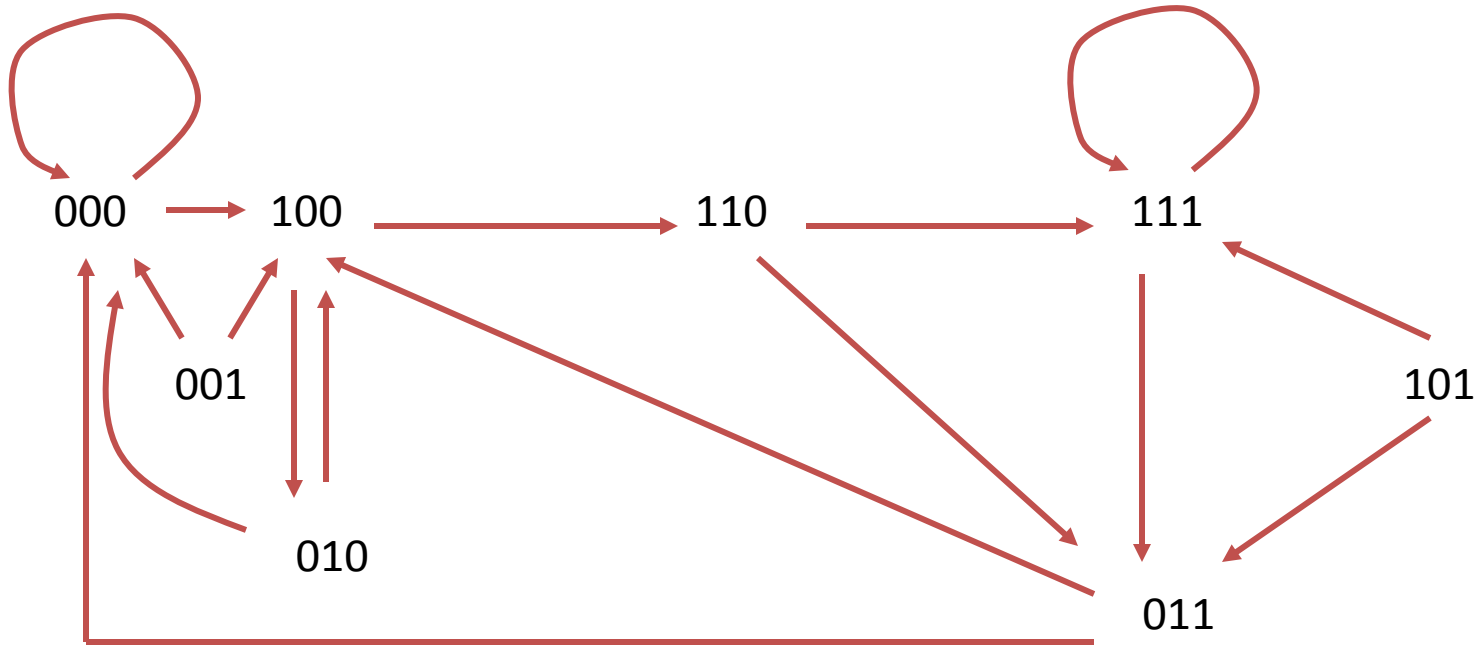
In practice used together with SAT and other engines

Questions?

Next step in FV methods: Symbolic Model Checking (week after next)

Next week: PSL

Answer to earlier question



More questions:
Given the initialisation of the state holding elements shown in slide 17, how many initial states are there and why? What are they?
Write down the corrresponding binary relation as a set of pairs of states