CSE 620: Selective Topics

Introduction to Formal Verification

Master Studies in CSE

Winter 2017

Lecture #3

Dr. Hazem Ibrahim Shehata

Assistant Professor

Dept. of Computer & Systems Engineering

Course Outline

- Computational Boolean Algebra
  - Basics
    - Shannon Expansion
    - Boolean Difference
    - Quantification Operators
      - Application to Logic Network Repair
  - Validity Checking (Tautology Checking)
  - Binary Decision Diagrams (BDD's)
  - Satisfiability Checking (SAT solving)
- Model Checking
  - Temporal Logics ➔ LTL - CTL
  - SMV: Symbolic Model Verifier
  - Model Checking Algorithms ➔ Explicit CTL

VLSI CAD:
Logic to Layout

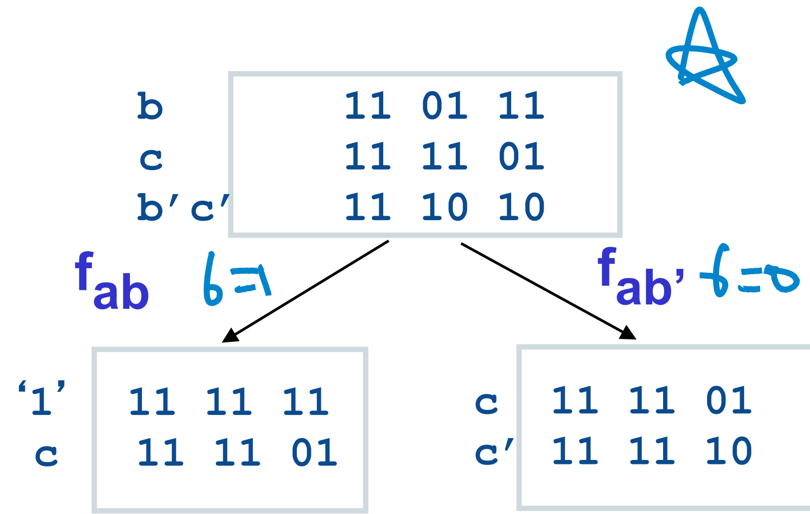
Lecture 3.1

Computational Boolean Algebra Representations: BDD Basics, Part 1

Rob A. Rutenbar
University of Illinois

Representations, Revisited

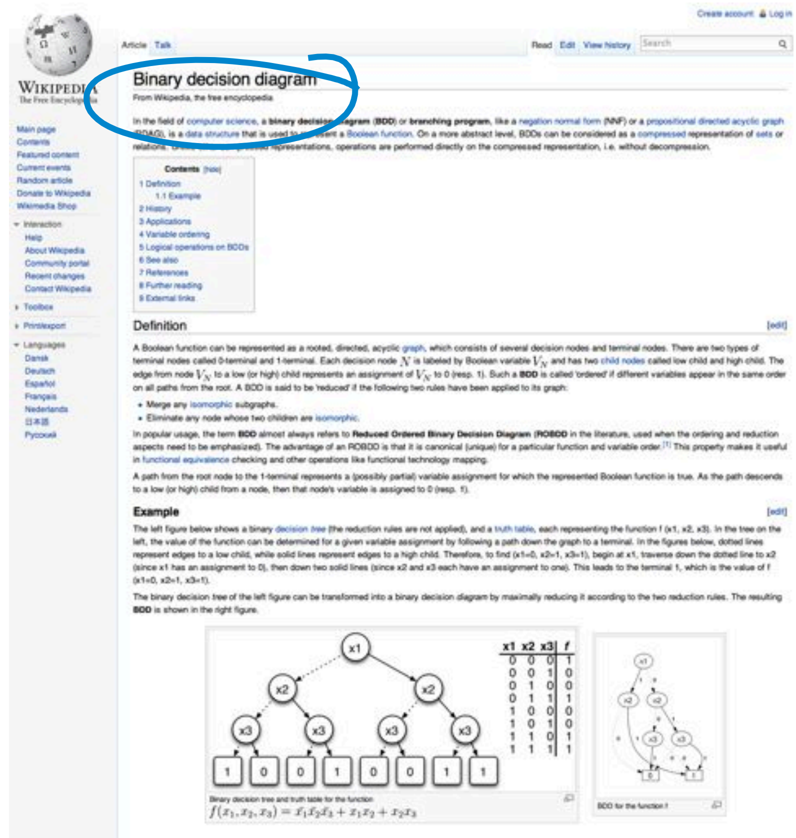
- **URP tautology is a great "warm up" example**
  - Shows big idea: Boolean functions as things we manipulate with software
  - **Data structure + operators**
  - But URP not the real way(s) we do it
- **Let's look at a real, important, *elegant* way to do this…**
  - **Binary Decision Diagrams: BDDs**



Background: BDDs

- **Originally studied by many**
  - Lee first, then Boute and Akers
- **Got seriously useful in 1986**
  - Randy Bryant of CMU made breakthrough of **ROBDDs**
  - Randy also provided many of the nice BDD pictures in this lecture –thanks!

http://www.cs.cmu.edu/~bryant/



Create account Log in

Article Talk Read Edit View history Search

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox
Print/export
Languages
Dansk
Deutsch
Español
Français
Nederlands
日本語
Русский

Binary decision diagram

From Wikipedia, the free encyclopedia

In the field of computer science, a **binary decision diagram** (**BDD**) or **branching program**, like a negation normal form (NNF) or a propositional directed acyclic graph (PDAG), is a data structure that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression.

Contents [hide]

1 Definition
1.1 Example
2 History
3 Applications
4 Variable ordering
5 Logical operations on BDDs
6 See also
7 References
8 Further reading
9 External links

Definition [edit]

A Boolean function can be represented as a rooted, directed, acyclic graph, which consists of several decision nodes and terminal nodes. There are two types of terminal nodes called 0-terminal and 1-terminal. Each decision node N is labeled by Boolean variable V_N and has two child nodes called low child and high child. The edge from node V_N to a low (or high) child represents an assignment of V_N to 0 (resp. 1). Such a **BDD** is called 'ordered' if different variables appear in the same order on all paths from the root. A BDD is said to be 'reduced' if the following two rules have been applied to its graph:

- Merge any isomorphic subgraphs.
- Eliminate any node whose two children are isomorphic.

In popular usage, the term **BDD** almost always refers to **Reduced Ordered Binary Decision Diagram** (**ROBDD** in the literature, used when the ordering and reduction aspects need to be emphasized). The advantage of an ROBDD is that it is canonical (unique) for a particular function and variable order.[1] This property makes it useful in functional equivalence checking and other operations like functional technology mapping.

A path from the root node to the 1-terminal represents a (possibly partial) variable assignment for which the represented Boolean function is true. As the path descends to a low (or high) child from a node, then that node's variable is assigned to 0 (resp. 1).

Example [edit]

The left figure below shows a binary decision tree (the reduction rules are not applied), and a truth table, each representing the function f (x1, x2, x3). In the tree on the left, the value of the function can be determined for a given variable assignment by following a path down the graph to a terminal. In the figures below, dotted lines represent edges to a low child, while solid lines represent edges to a high child. Therefore, to find (x1=0, x2=1, x3=1), begin at x1, traverse down the dotted line to x2 (since x1 has an assignment to 0), then down two solid lines (since x2 and x3 each have an assignment to one). This leads to the terminal 1, which is the value of f (x1=0, x2=1, x3=1).

The binary decision tree of the left figure can be transformed into a binary decision diagram by maximally reducing it according to the two reduction rules. The resulting **BDD** is shown in the right figure.

x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Binary decision tree and truth table for the function
$f(x_1,x_2,x_3) = \bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2 + x_2x_3$

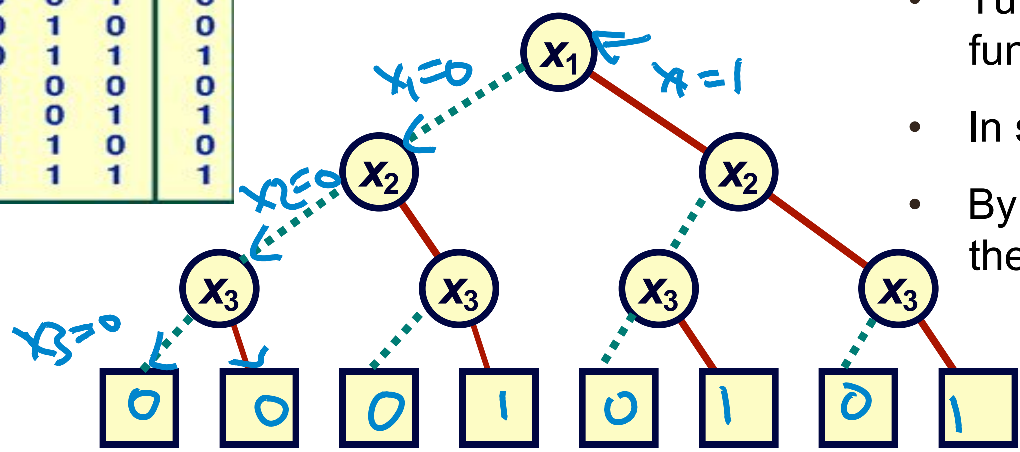
BDD for the function f

Binary Decision Diagrams for Truth Tables

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



- **Big Idea #1: BDD**
  - Turn a truth table for the Boolean function into a **Decision Diagram**
  - In simplest case, graph is just a tree
  - By convention – don't draw arrows on the edges, we know where they go

(down!)

- **Vertex** represents a variable; **edge** out is a **decision** (0 or 1)
- Follow **green** (***dashed***) line for value 0
- Follow **red** (***solid***) line for value 1
- Function value determined by ***leaf value***.

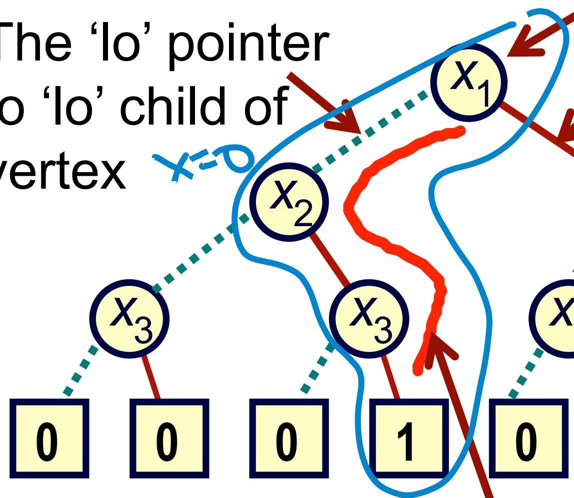
Binary Decision Diagrams

- **Some terminology**

A 'variable' vertex

The 'lo' pointer to 'lo' child of vertex x=0

The 'hi' pointer to 'hi' child of of vertex x=1

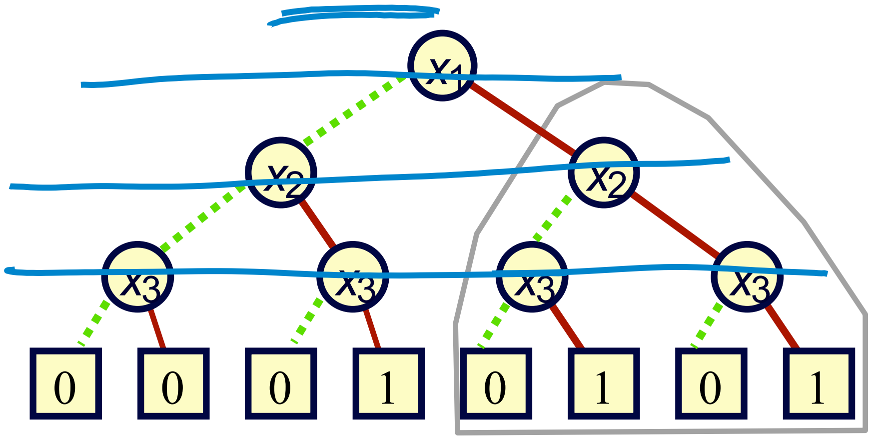


A 'constant' vertex at the bottom leaves of the tree

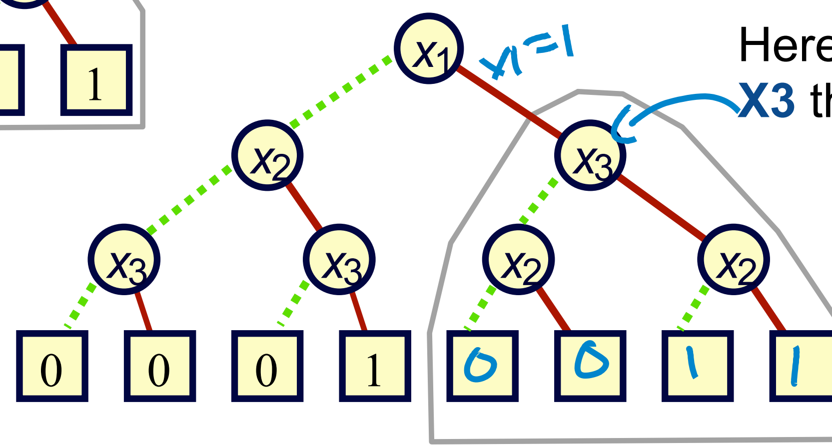
The **'variable ordering'**, which is the order in which decisions about variables are made. Here, it's **X1 X2 X3**

Ordering

- **Note: Different variable orders are possible**



Order for this subtree is
X2 then **X3**



Here, it's
X3 then **X2**

different

Binary Decision Diagrams

- **Observations**
  - Each path from root to leaf traverses variables in a **some** order
  - Each such path constitutes a row of the truth table, ie, a decision about what output is when variables take particular values
  - But we have not yet specified anything about the **order** of decisions
  - This decision diagram is **not unique** for this function

- **Terminology: Canonical form**
  - Representation that does **not** depend on gate implementation of a Boolean function
  - Same function of same variables always produces this exact **same** representation
  - Example: a **truth table** is canonical. ***We want a canonical form data structure.***

(up to order)

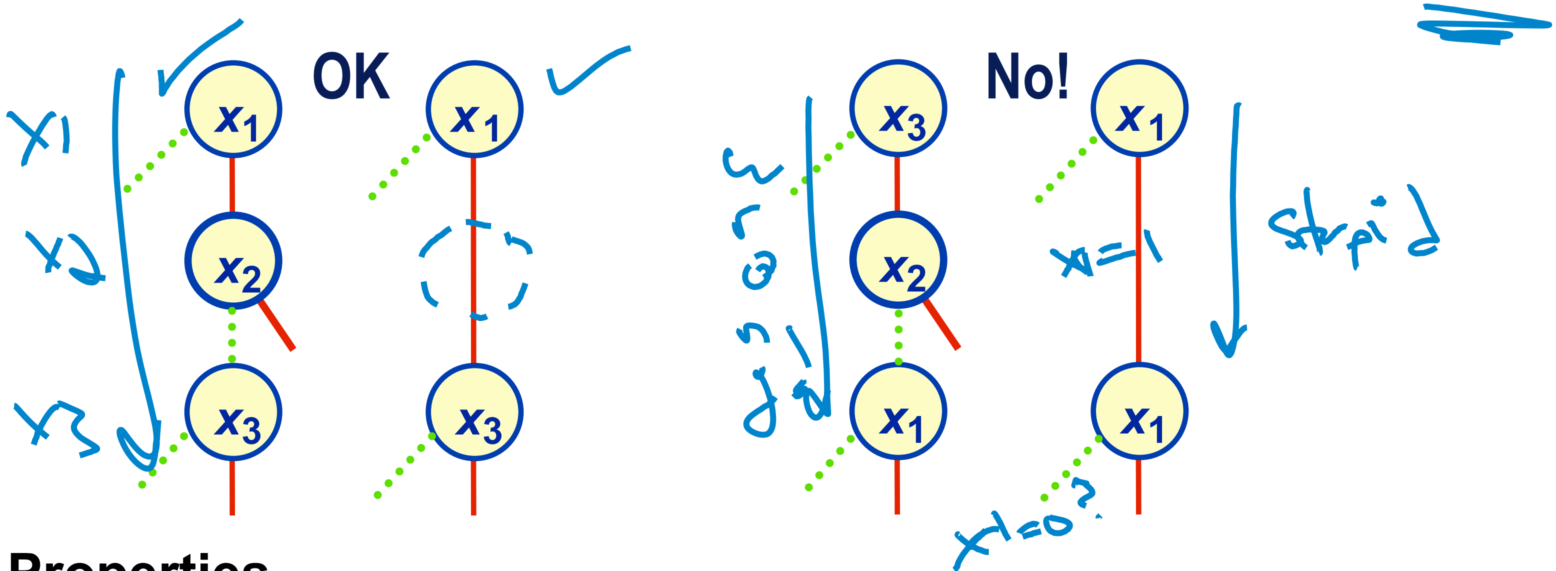
Binary Decision Diagrams

- **What's wrong with this diagram representation?**
  - It's **not canonical**, and it is way **too big** to be useful (it's as big as truth table!)

- **Big idea #2: Ordering**
  - **Restrict** global ordering of variables
  - **Means:** Every path from root to a leaf visits variables in SAME order
  - Note
    - It's OK to **omit** a variable if you don't need to check it to decide which leaf node to reach for final value of function

Ordering BDD Variables

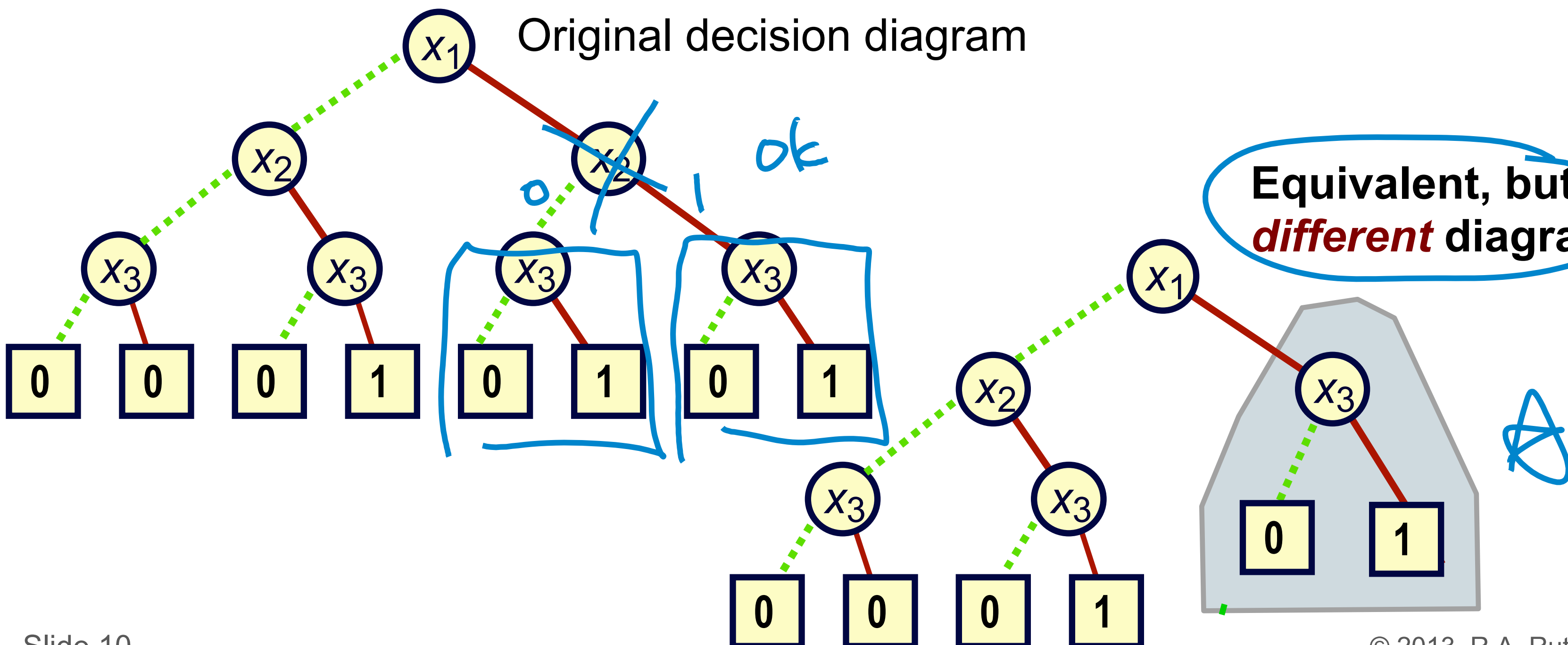
- **Assign arbitrary *ordering* to vars: x1 < x2 < x3**
  - **Variables must appear in this specific order along all paths; ok to skip vars**



- **Properties**
  - No conflicting assignments along path (see each var at most **once on path**).

Binary Decision Diagrams

- **OK, now what's wrong with it?**
  - Variable ordering simplifies things... But still **too big**, and **not canonical**



VLSI CAD:
Logic to Layout

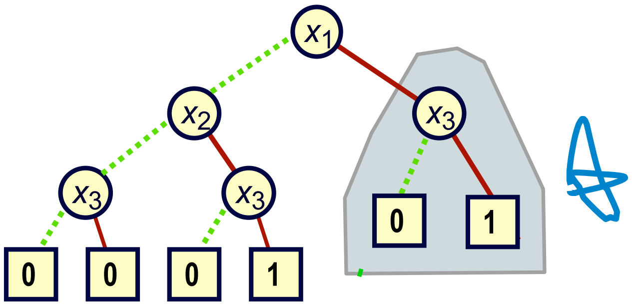
Lecture 3.2

Computational Boolean Algebra Representations: BDD Basics, Part 2

Rob A. Rutenbar
University of Illinois

Binary Decision Diagrams

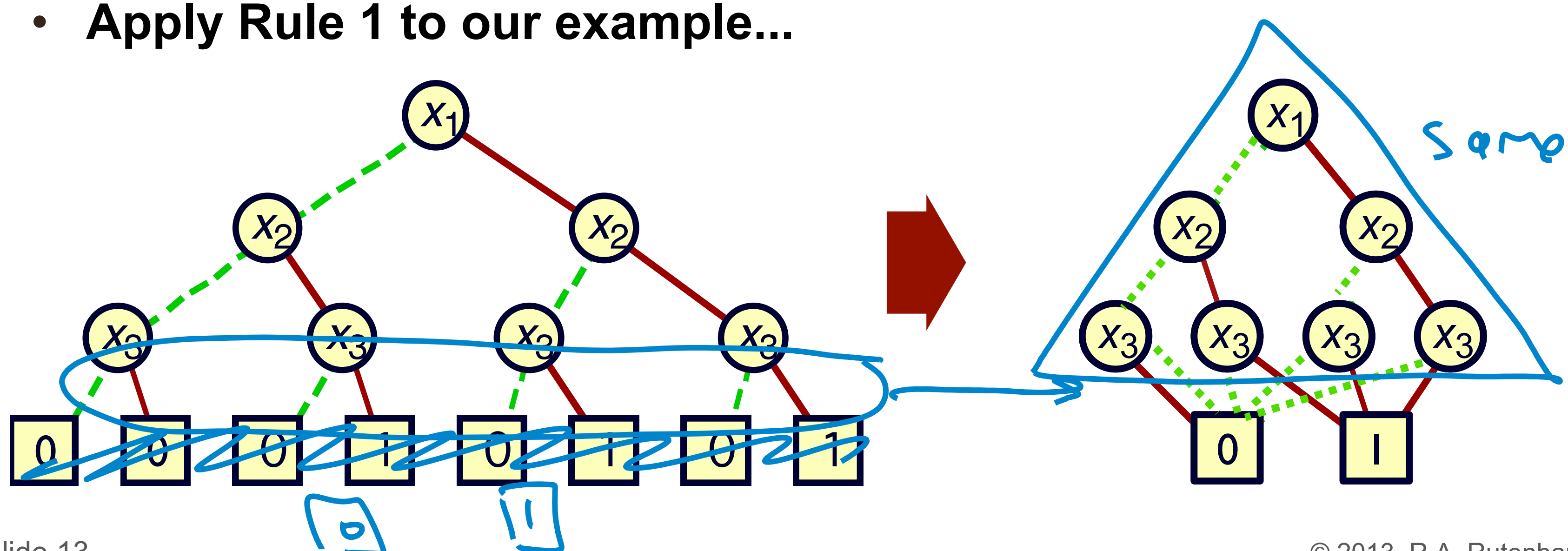
- **Big Idea #3:** ***Reduction***
  - Identify **redundancies** in the graph that can remove unnecessary nodes and edges
  - Removal of **X2** node and its children, replace with **X3** node is an example of this



- **Why are we doing this?**
  - **GRAPH SIZE**: Want result as ***small*** as possible
  - **CANONICAL FORM:** For **same function**, given **same variable order**, want there to be exactly ***one*** graph that represents this function

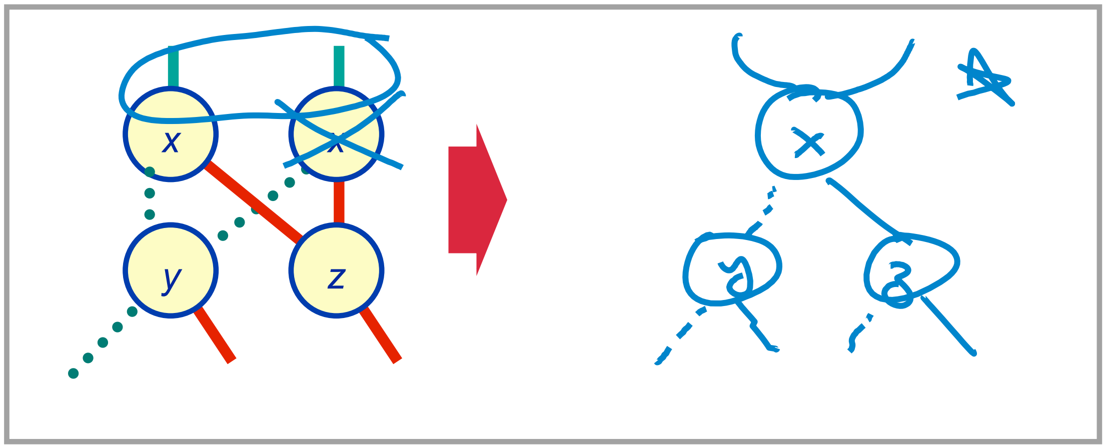
Reduction Rules

- **Reduction Rule 1:** ***Merge equivalent leaves***
  - Just keep one copy of each constant leaf – anything else is totally wasteful
  - Redirect all edges that went into the redundant leaves into this one kept node
- **Apply Rule 1 to our example...**



Reduction Rules

- **Reduction Rule 2:** ***Merge isomorphic nodes***



Remove redundant node (extra 'x' node).
Redirect all edges that went into the redundant node into the one copy that you kept (edges into right 'x' node now into left as well)

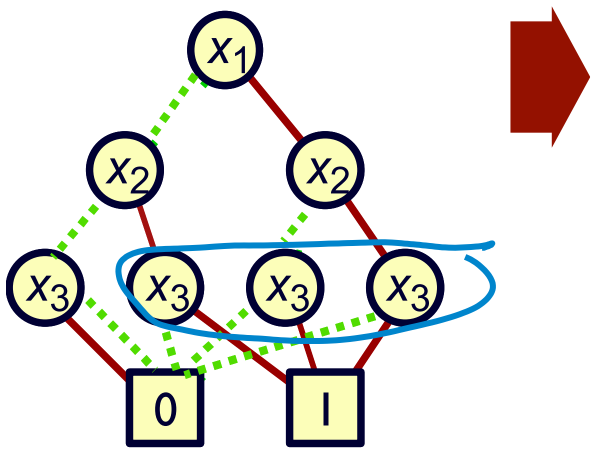
lo: y
hi: z
same

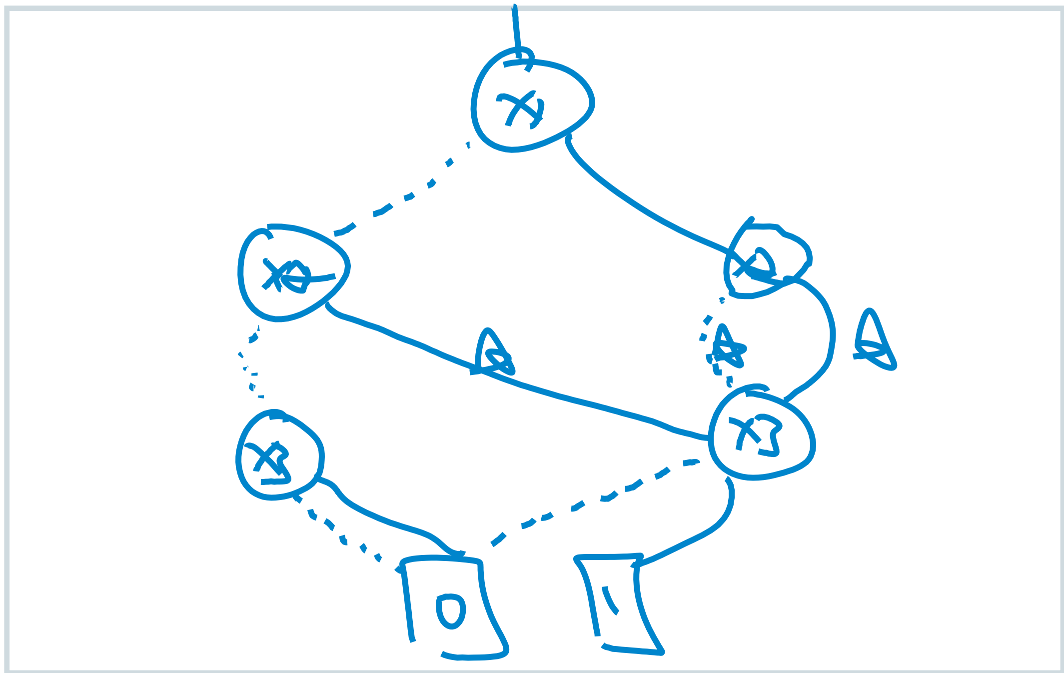
x

- **Isomorphic = 2 nodes with *same* variable and *identical* children**
  - Cannot tell these nodes apart from how they contribute to decisions in graph
  - Note: means ***exact same physical*** child nodes, not just children with same label

Reduction Rules

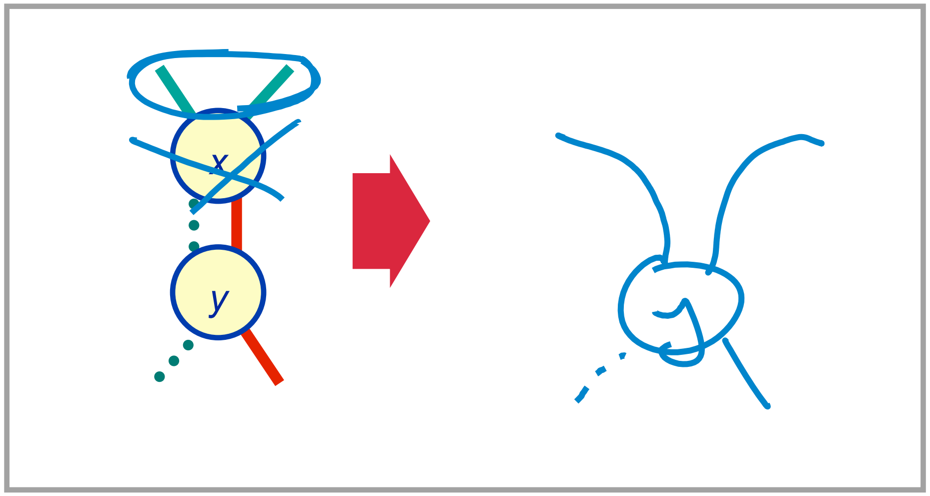
- **Apply Rule 2 to our example**





Reduction Rules

- **Reduction Rule #3:** ***Eliminate Redundant Tests***

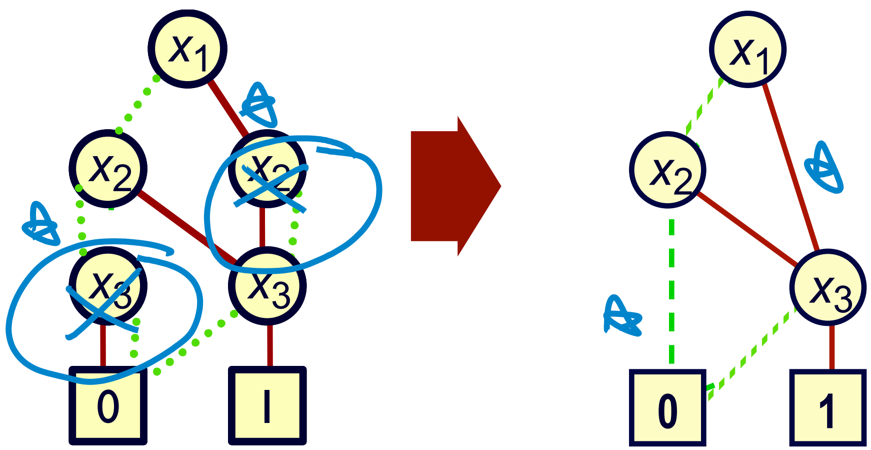


Remove redundant node
Redirect all edges into redundant node (x) into child (y) of the removed node

- **Test:** means a variable node: redundant since children go to same node...
  - ...so we don't care what value **x** node takes in this diagram

Reduction Rules

- **Apply Rule #3 to our example … and we're finished!**



- **Aside: How to apply the rules?**
  - For now, **iteratively**: when you can't find another match, graph is reduced
  - Is this how programs really do it? **No** – we will talk about that later… !!

BDDs: Big Results

- **Recap: What did we do?**
  - Start with a BDD, order the vars, reduce the diagram
  - Name: **Reduced Ordered BDD (ROBDD)**
- **Big result**

ROBDD is a canonical form (data structure) for any Boolean function!

  - **Same function** always generates exactly **same graph**... for **same var ordering**
  - Two functions identical if and only if ROBDD graphs are isomorphic (ie, same)
- **Nice property to have: Simplest form of graph is canonical**

VLSI CAD:
Logic to Layout

Lecture 3.3

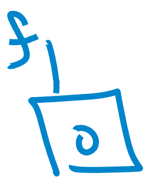
Computational Boolean Algebra Representations: BDD Sharing

Rob A. Rutenbar
University of Illinois

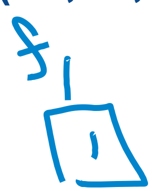
BDDs: Representing Simple Things

- **Note: can represent any function as a ROBDD**

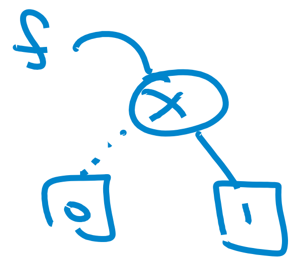
ROBDD for **f(x1,x2,...xn) = 0**



ROBDD for **f(x1,x2,...xn) = 1**



ROBDD for **f(x1, …, x, …xn) = x**



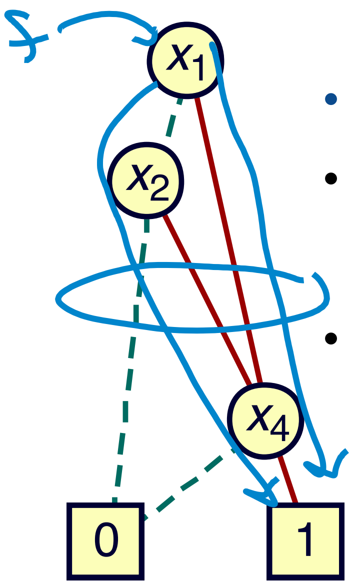
NOTE: In a ROBDD, a Boolean function is really just **a pointer to the root node of** a canonical graph data structure

BDD: Examples

- **Assume variable order is X1, X2, X3, X4**

Typical Function

x1 x4 + x2 x4

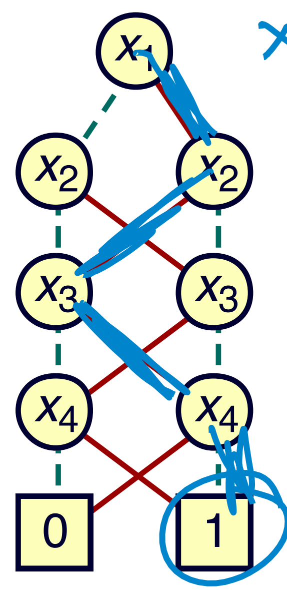


- **(X1 + X2)X4**
- No vertex labeled **X3**
  - – independent of **X3**
- Many subgraphs **shared**

Odd Parity

== 1 iff odd #'s

x1 ⊕ x2 ⊕ x3 ⊕ x4

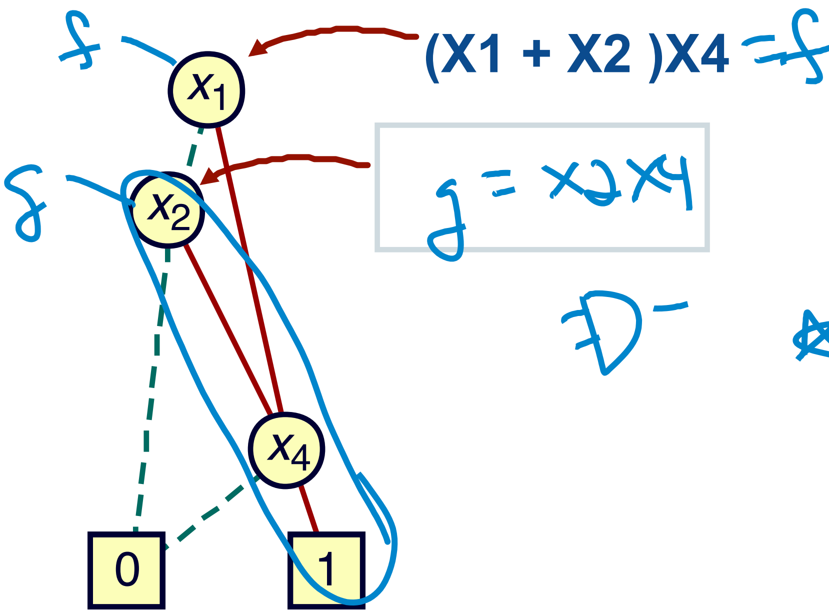


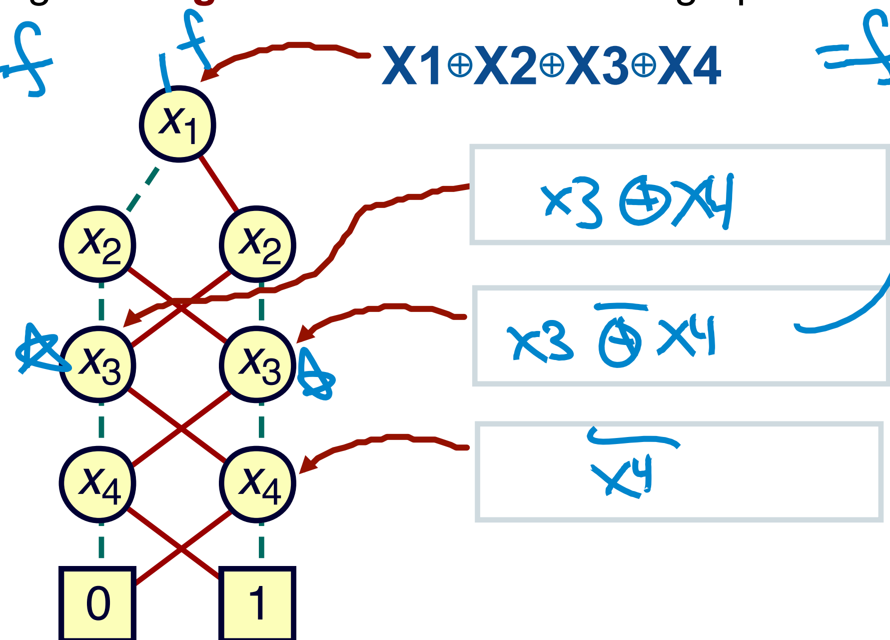
Linear representation

x1 x2 x3 x4
1 1 1 0

Sharing in BDDs

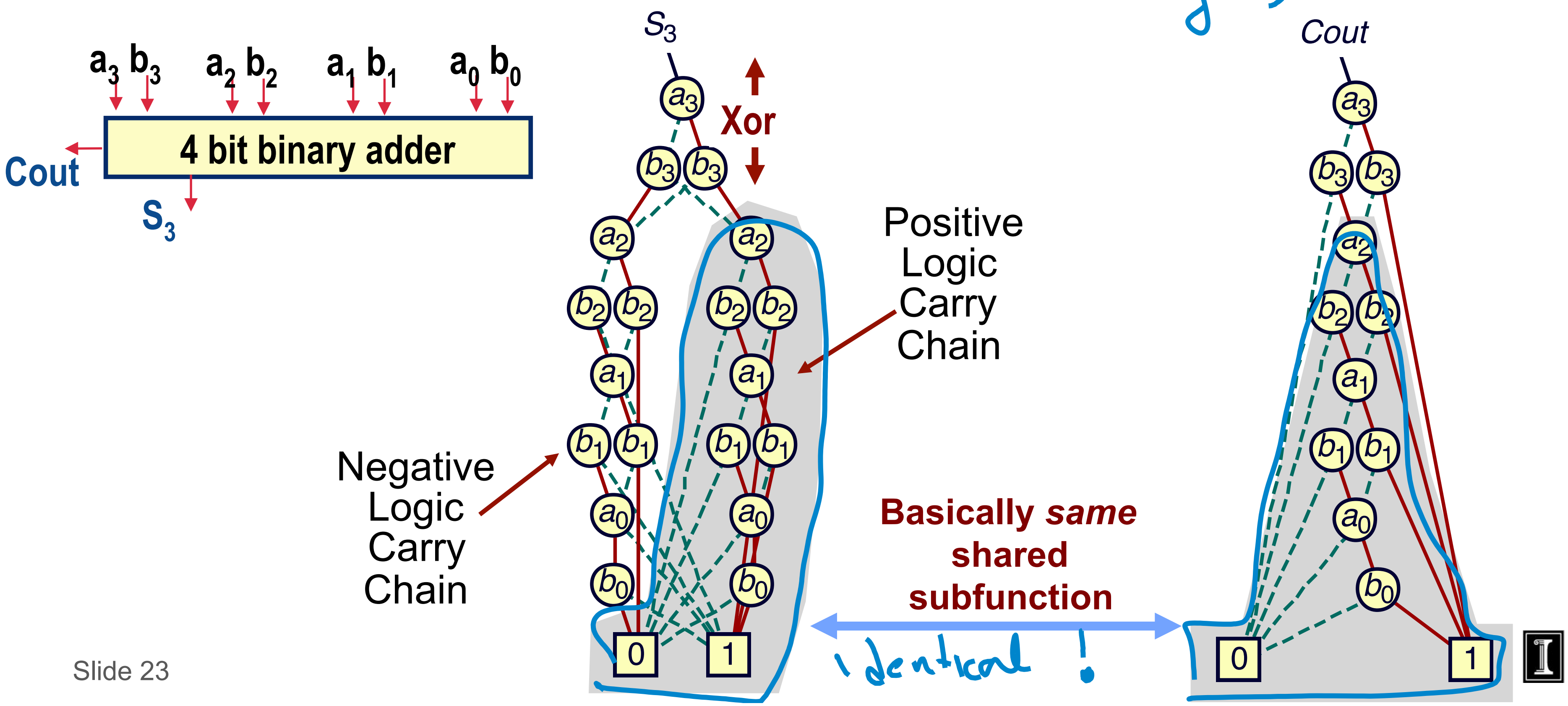
- **Very important technical point**
  - **Every** BDD node (not *just* root) represents **some** Boolean function in a **canonical** way
  - BDDs good at extracting & representing **sharing of subfunctions** in subgraphs





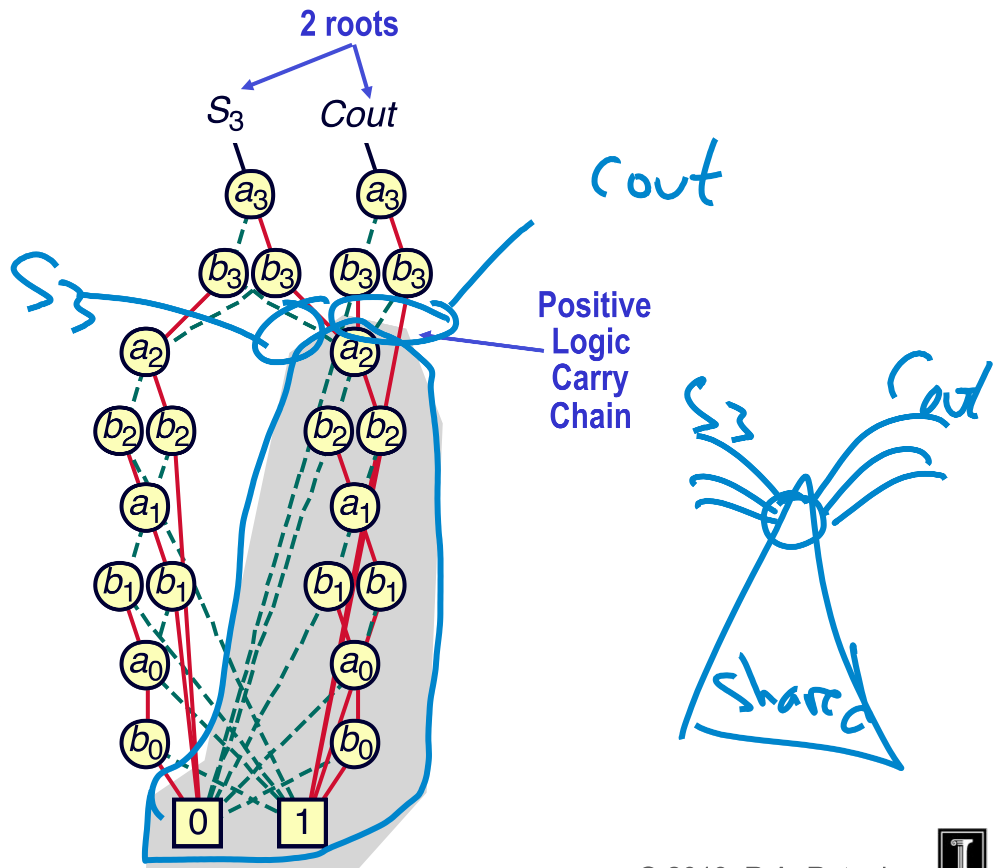
BDDs Sharing: Consider a 4bit Adder

- **Look at sum S3 msb & carry out Cout** (no carry in)



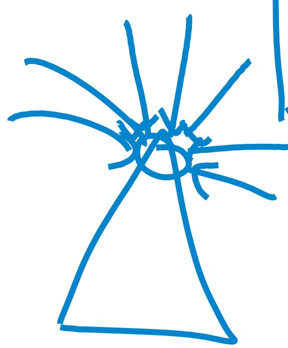
BDD Sharing: Multi-rooted Graph

- **Don't represent it twice!**
  - BDD can have multiple 'entry points', or roots
  - Called a **multi-rooted BDD**
- **Recall (again!)**
  - ***Every*** node in a BDD represents some Boolean function
  - This multi-rooting idea just explicitly exploits this to better share stuff

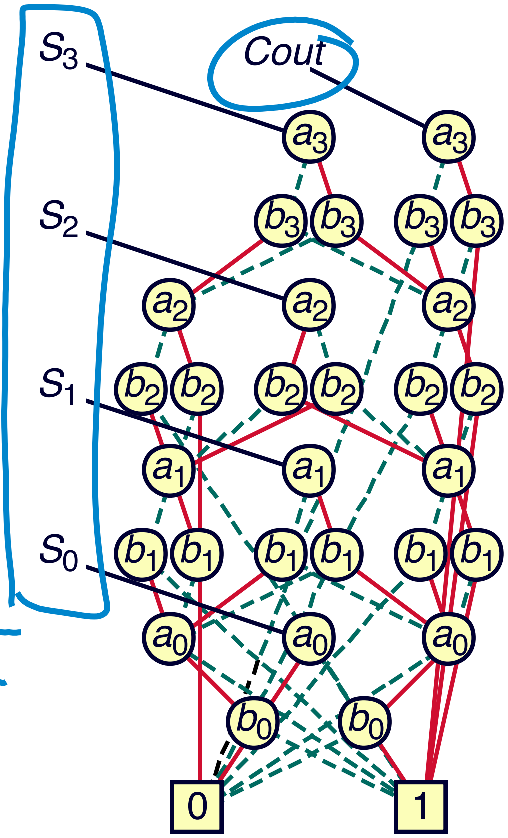


Sharing: Multi-rooted BDD Graphs

- **Why stop at 2 roots?**
  - Big savings for **sets of functions**
  - Minimize size over **several separate** BDDs by maximum sharing
- **Example: Adders**
  - Separately
    - **51** nodes for **4-bit** adder
    - **12,481** for **64-bit** adder
  - Shared
    - **31** nodes for **4-bit** adder
    - **571** nodes for **64-bit** adder



sum



full 4bit adder

VLSI CAD:
Logic to Layout

Lecture 3.4
Computational Boolean Algebra Representations: BDD Ordering

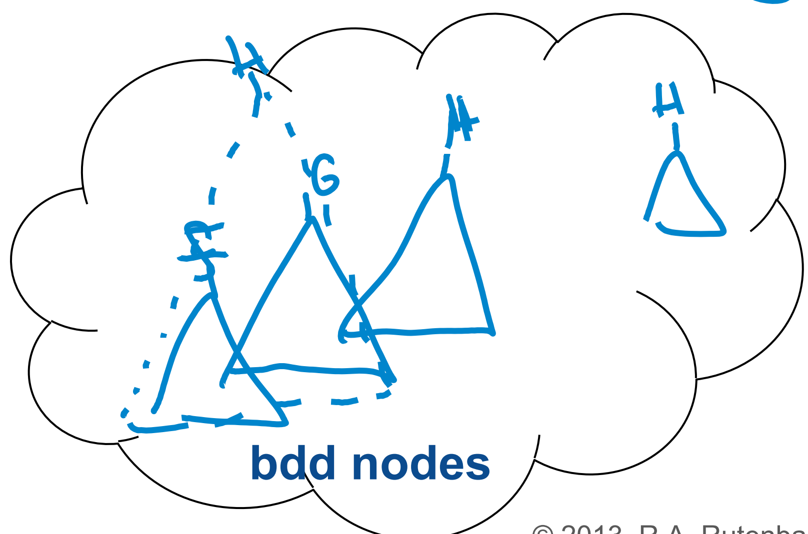
Rob A. Rutenbar
University of Illinois

How Are BDDs Really Implemented

- **Recursive methods, like URP**
  - Shannon cofactor divide & conquer is key
- **As a set of operators (ops) on the BDDs**
  - AND, OR, NOT, EXOR, CoFactor…
  - ∀Quant, ∃Quant, Satisfy, etc
- **Operate on a universe of Boolean data as input/output**
  - Constants 0,1; vars; Bool functions represented as **shared BDD graphs**

- **Big trick:** Implement each op so if inputs **shared**, **reduced**, **ordered**, → outputs are too

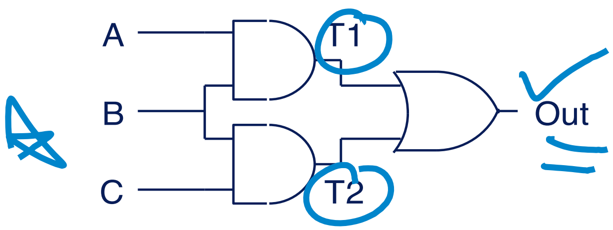
H = (bdd)OP(bdd F, bdd G)

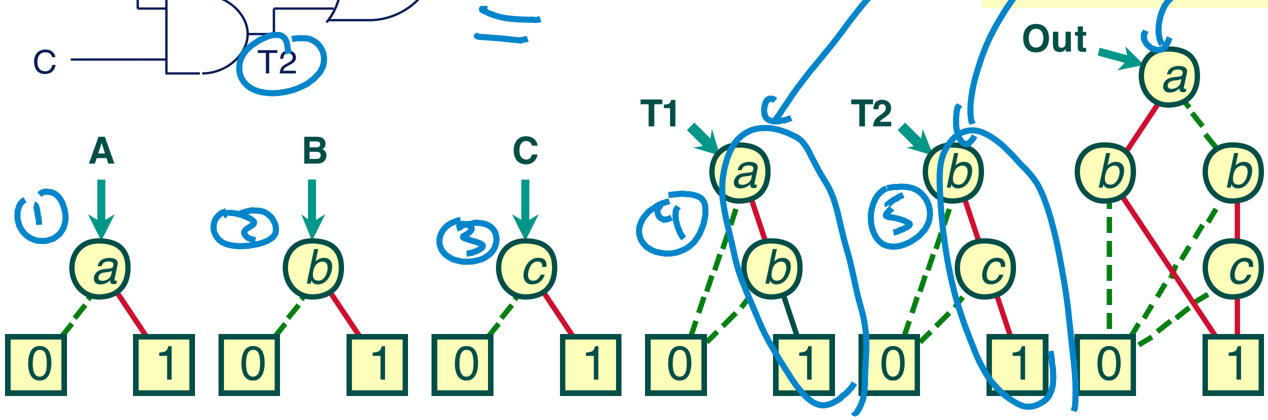


BDDs: Build Up Incrementally…

- **For example: by walking a gate-level network**
  - Each **input** is a BDD, each **gate** becomes an **operator** that produces a new **output** BDD
  - Build BDD for Out as a **script** of calls to basic BDD operators

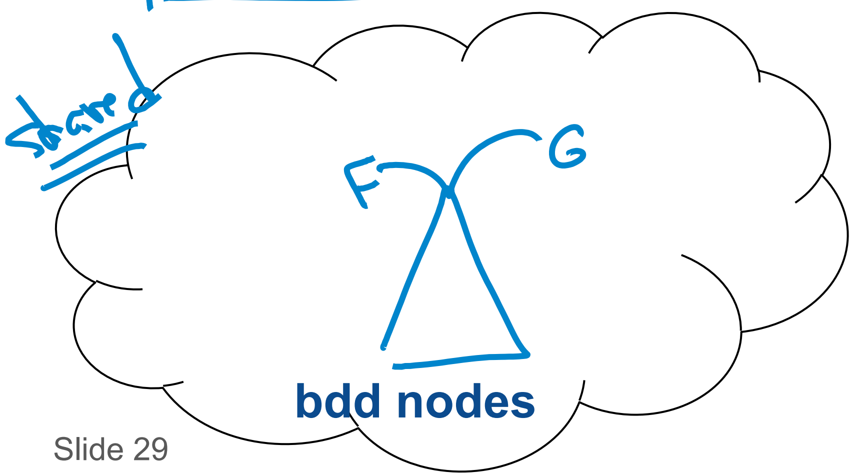
BDD operator script
1. A = CreateVar(“A”)
2. B = CreateVar(“B”)
3. C = CreateVar(“C”)
4. T1 = And(A,B)
5. T2 = And(B,C)
6. Out = Or(T1,T2)



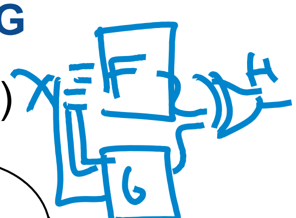


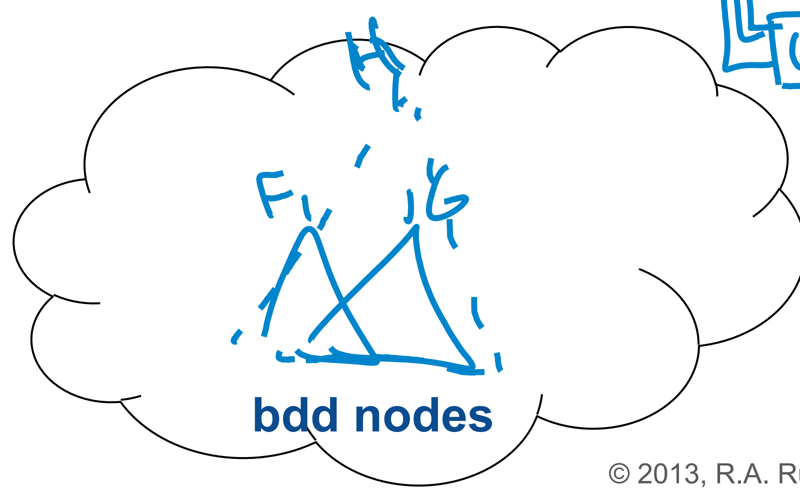
Apps: Comparing Logic Implementations

- **Are these two Boolean functions F, G the same?**
  - Build BDD for **F**. Build BDD for **G**
  - Compare pointers to roots of **F, G**
  - If pointers are **same** (!!), **F==G**



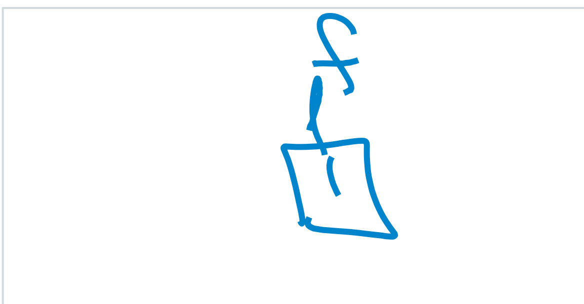
- **What inputs make functions F, G give different answers?**
  - Build BDD for **F**. Build BDD for **G.**
  - Build the BDD for **H = F⊕G**
  - Ask "**satisfiable**" for **H** (!!)



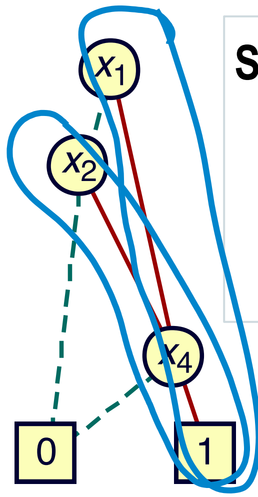


More Very Useful BDD Applications

- **Tautology checking**
  - Was complex with cubelist URP, subtle algorithm, lots of work
  - With BDDs, it's **trivial.** Just build the BDD, check if BDD graph ==



- **Satisfiability**
  - Find values **(0,1)** for vars so **F == 1?**
  - No idea how to do it with cubelists
  - Any **path** from root to "1" leaf is soln!



Satisfiability: X1 X2 X3 X4 =

1 - - 1

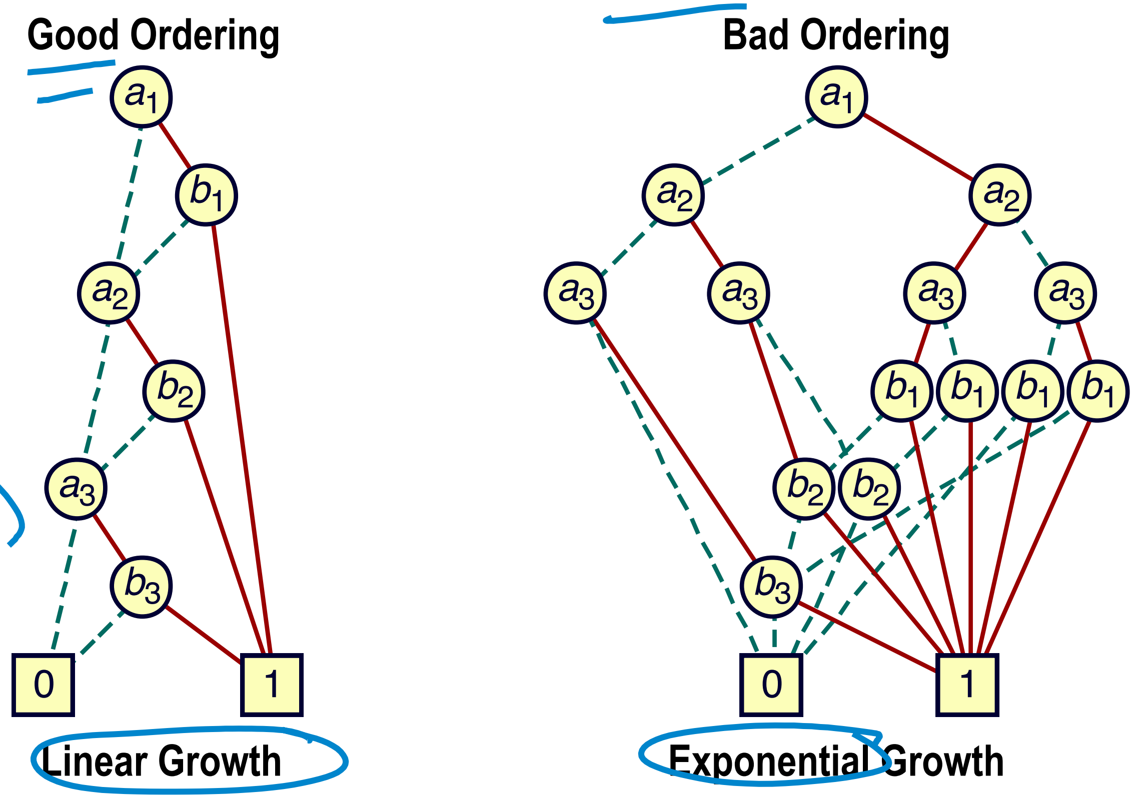
0 1 - 1

another OP

BDDs: Seem Too Good To Be True?!

- Problem : Variable ordering ***matters***. Ex: a1•b1 + a2•b2 + a3•b3

n=3



size = O(n)

size = O(2^n)

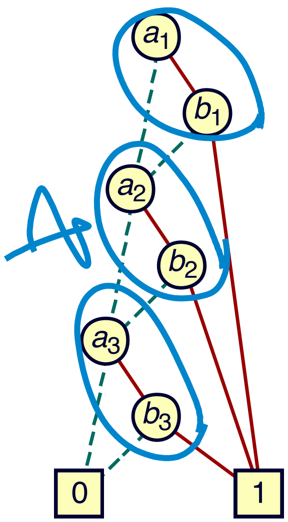
Variable Ordering: Consequences

- **Interesting problem**
  - Some problems known to be **exponentially hard** are very **easy** done with BDDs(!)
  - Trouble is, they are easy only if size of the BDD for the problem is "reasonable"
  - Some problems make nice (small) BDDs, others make pathological (large) BDDs
  - No universal solution (or could always to solve exponentially hard problems easily)

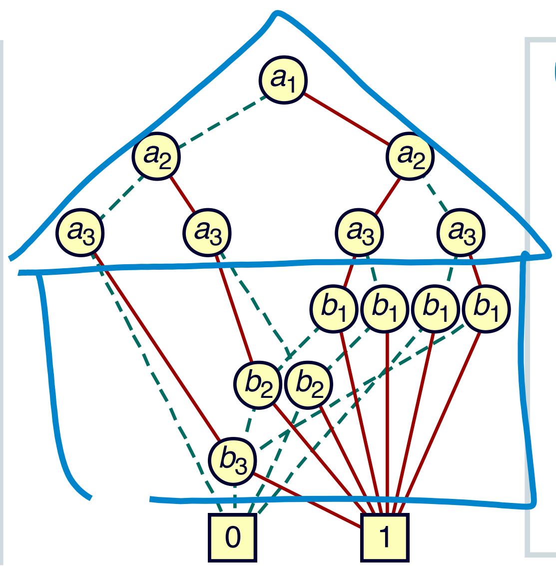
- **How to handle?**
  - **Variable ordering heuristics:** make nice BDDs for reasonable probs
  - **Characterization:** know which problems *never* make nice BDDs (eg, multipliers)
  - **Dynamic ordering:** let the BDD software package pick the order… on the fly

Variable Ordering: Intuition

- **Rules of thumb for BDD ordering** **Ex: (a1•b1 + a2•b2 + a3•b3)**
  - **Related inputs** should be *near* each other in order;   **groups** of inputs that can **determine function by themselves** should be (i) *close* together, and (ii) near *top* of BDD



$a_i \cdot b_i$ can together determine F( )



worst order! all as first then all bs

Aside: Variable Ordering

- **Arithmetic circuits are important logic; how are their BDDs?**
  - Many **carry chain circuits** have easy linear sized ROBDD orderings: Adders, Subtractors, Comparators, Priority encoders
  - Rule is **alternate** variables in the BDD order: **a0 b0 a1 b1 a2 b2 … an bn**
- **So – are all arithmetic circuits easy then?   No, sorry**

Function Class	Best Order	Worst Order
Addition	linear	exponential
Multiplication	exponential	exponential

- **General experience with BDDs**
  - Many tasks have reasonable OBDD sizes; algorithms practical to **~100M** nodes
  - People spend a lot of effort to find orderings that work…

BDD Summary

- **Reduced, Ordered, Binary Decision Diagrams, ROBDDs**

  BDDs

  - Canonical form – a data structure – for Boolean functions
  - Two boolean functions the same if and only if they have identical BDD
  - A Boolean function is just a pointer to the root node of the BDD graph
  - Every node in a (shared) BDD represents some function
  - With shared BDD, test (**F==G**) → check pointers for equality, point to same root
  - Basis for much of today's general manipulation or Boolean stuff
- **Problems**
  - Variable ordering matters; sometimes BDD is just too big
  - Often, we just want to know **SAT** – don't need to build the whole function