(*Lec 3*) Binary Decision Diagrams: Representation

What you know

- Lots of useful, advanced techniques from Boolean algebra
- Lots of cofactor-related manipulations
- A little bit of computational strategy
  - Cubelists, positional cube notation
  - Unate recursive paradigm

What you don't know

- The "right" data structure for dealing with Boolean functions: BDDs
- Properties of BDDs
  - Graph representation of a Boolean function
  - Canonical representation
- Efficient algorithms for creating, manipulating BDDs
  - Again based on recursive divide&conquer strategy

  (Thanks to Randy Bryant for nice BDD pics+slides)

Copyright Notice

Handouts

- **Physical**
  - **Lecture 03 -- BDDs: Representation**
  - **Paper: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, *ACM Computing Surveys*, Sept 1992.**
- **Electronic**
  - **Nothing today**
- **Reminder**
  - **HW1 is due Thu in class**

Where Are We?

- **Still doing Boolean background, now focussed on data structs**

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
	15	16	17	18	19	8
	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

Introduction
Advanced Boolean algebra
JAVA Review
Formal verification
2-Level logic synthesis
Multi-level logic synthesis
Technology mapping
Placement
Routing
Static timing analysis
Electrical timing analysis
Geometric data structs & apps

Readings

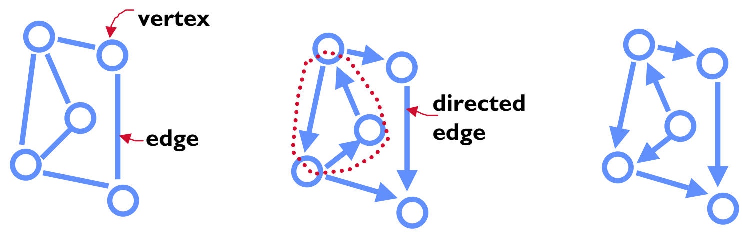
- In De Micheli book
  - pp 75-85 does BDDs, but not in as much depth as the notes
- Randy Bryant paper
  - Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, ACM Computing Surveys, Sept 1992.
  - Lots more detail (some of it you don't need just yet) but very complete, if a bit terse.

BDD History

- A little history…
  - Original idea for Binary Decision Diagrams due to Lee (1959) and Akers (1978)
  - Critical refinement–*Ordered* BDDs–due to Bryant (1986)
    - Refinement imposes some restrictions on structure
    - Restrictions needed to make result *canonical* representation
- A little terminology
  - A BDD is a *directed acyclic graph*
  - **Graph:** vertices connected by edges
  - **Directed:** edges have direction (draw them with an arrow)
  - **Acyclic:** no cycles possible by following arrows in graph
  - Often see this shortened to "DAG"

Graphs

- DAGs -- a reminder of some technicalities…



A graph
vertices + edges

A directed graph
…but not acyclic

A directed acyclic graph
…note that a "loop" is not a directed cycle, you are only allowed to follow edges along *direction* that the arrow points

Binary Decision Diagrams

- Big Idea #1: Binary Decision Diagram
  - Turn a truth table for the Boolean function into a Decision Diagram

Vertices = Edges = Leaf nodes =

  - In simplest case, resulting graph is just a tree

- Aside
  - Convention is that we don't actually draw arrows on the edges in the DAG representing a decision diagram
  - Everybody knows which way they point, implicitly
    - Point from parent to child in the decision tree

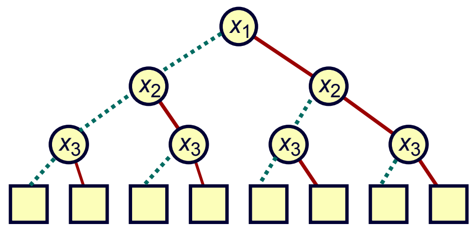
- Look at a simple example…

Binary Decision Diagrams

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

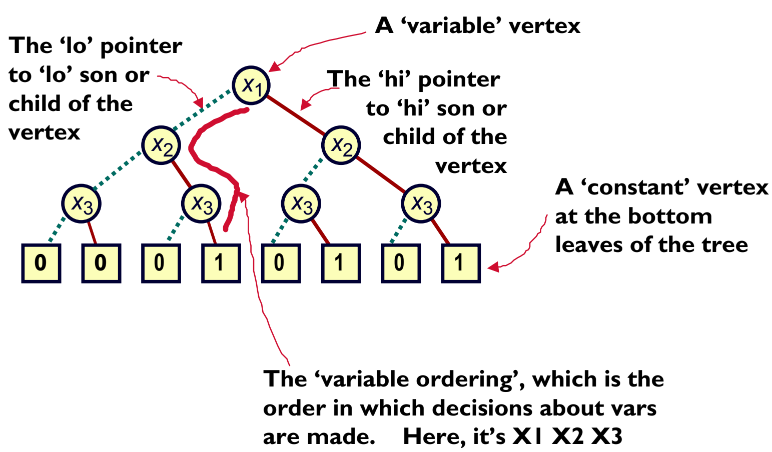
Decision Tree



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

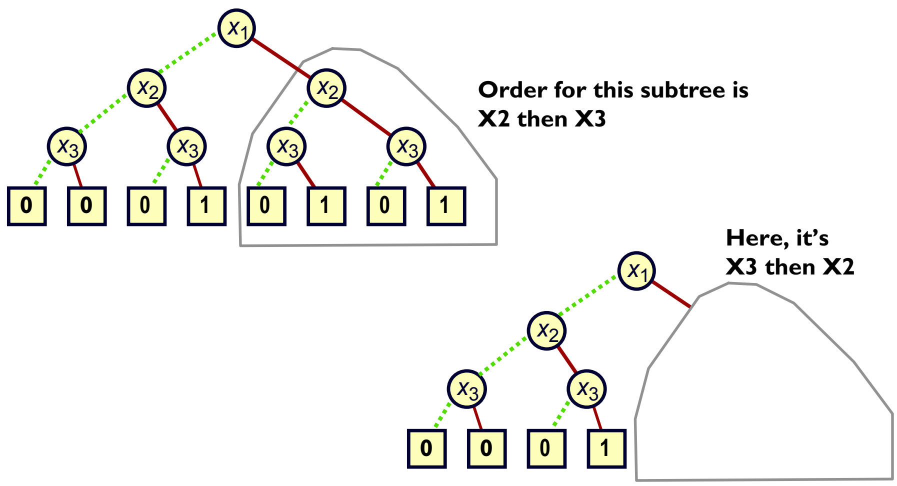
Binary Decision Diagrams

- Some terminology



Ordering

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



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 vars take particular values
  - But we have not yet specified anything about the order of decisions
  - This decision diagram is **not canonical** for this function

- **Reminder: canonical forms**
  - Representation that does *not* depend on the logic gate implementation of a Boolean function
  - Same function (ie, truth table) of same vars always produces this exact same representation
  - Example: a **truth table** is canonical
    a **minterm list**, for our function $f = \Sigma\, m(3,5,7)$, is canonical

Binary Decision Diagrams

- What's wrong with this representation?
  - It's not canonical,
  - Way too big to be useful
  - ...in fact it's every bit as big as a truth table: 1 leaf per row

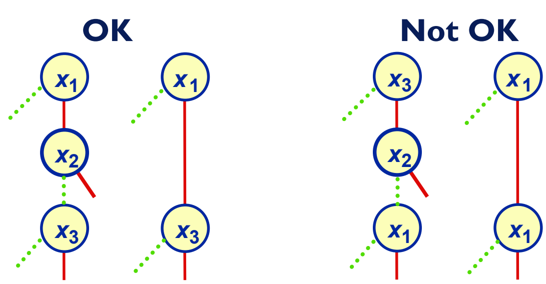
- Big idea #2: Ordering
  - Restrict **global ordering** of variables

    Means:

  - 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

Total Ordering

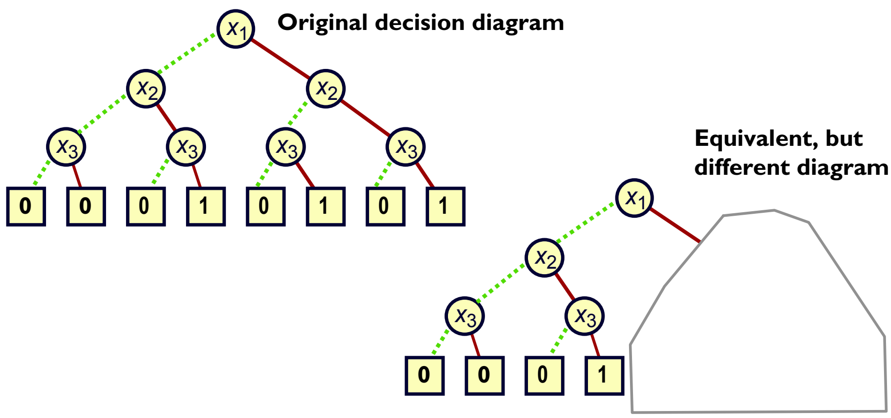
- Assign arbitrary *total ordering* to variables
  - $x_1 < x_2 < x_3$
  - Variables must appear in *this specific* order along all paths



- Properties
  - No conflicting variable assignments along path (see each var at most once walking down the path).
  - Simplifies manipulation

Binary Decision Diagrams

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

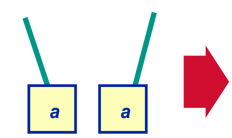


Binary Decision Diagrams

- Big Idea #3: Reduction
  - Identify redundancies in the DAG that can remove unnecessary nodes and edges
  - Removal of X2 node and its children, replacement with X3 node is an example of this sort of reduction
- Why are we doing this?
  - To combat size problem: want DAGs as *small* as possible
  - To achieve canonical form: for same function, given total variable order, want there to be exactly *one* graph that represents this function

Reduction Rules

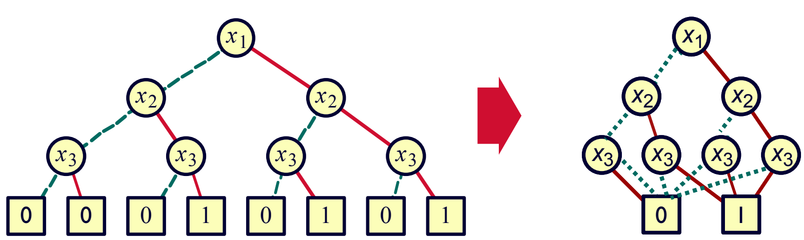
- Reduction Rule 1: Merge equivalent leaves



- 'a' is either a constant 1 or constant 0 here
- Just keep one copy of the leaf node
- Redirect all edges that went into the redundant leaves into this one kept node

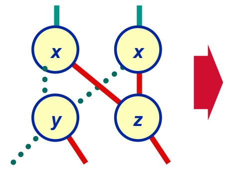
Reduction Rules

- Apply Rule 1 to our example...



Reduction Rules

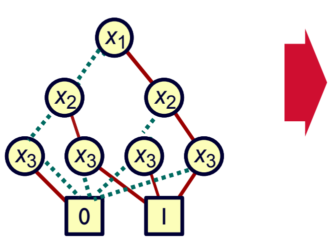
- Reduction Rule 2: Merge isomorphic nodes



- **Isomorphic:** Means 2 nodes with *same* var and *identical* children
  - You cannot tell these nodes apart from how they contribute to decisions as you decend thru DAG
  - Note: means *exact same physical* child nodes, *not* just children with same *labels*
- Remove redundant node (extra 'x' node here)
- 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)

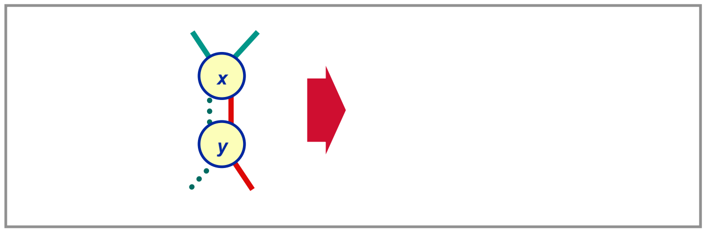
Reduction Rules

- Apply Rule 2 to our example



Reduction Rules

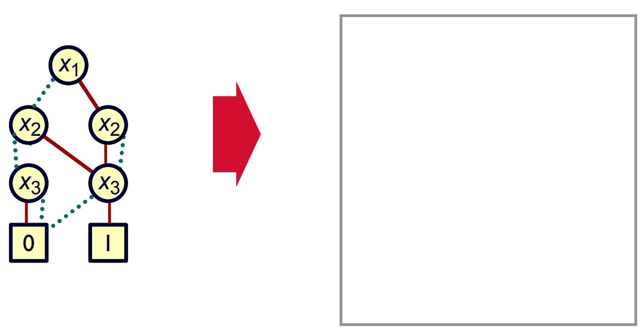
Reduction Rule #3: Eliminate Redundant Tests



- Test: means a variable node here...
  - It's redundant since both of its children go to same node...
  - ...so we don't care what value x node takes in this diagram
- Remove redundant node
- Redirect all edges into the redundant node (x) into the one child node (y) of the removed node

Reduction Rules

Apply Rule #3 to our example



Binary Decision Diagrams

How to apply the rules?

- For now, just iteratively, keep trying to find places the rules "match" and do the reduction
- When you can't find any more matches, the graph is reduced

Is this how programs really do it?

- Nope, there's some magic one can do with a clever hash table, but more about that later, when we start doing algorithms to manipulate BDDs
- Roughly speaking, in real programs you build the BDDs correctly on the fly--you never build a bad, noncanonical one then try to fix it.

BDDs: Big Results

Recap: what did we do?

- Start with any old BDD
- ...ordered the variables => Ordered BDD (OBDD)
- ...reduced the DAG => Reduced Ordered BDD (ROBDD)

Big result

  - Same function always generates exactly same DAG...
  - ...for a given variable ordering

  - ie, they are identically the same graph
- Nice property to have: *simplest form of DAG is canonical*

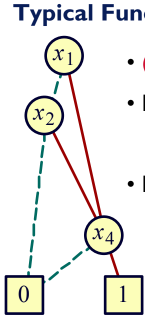
BDDs: Representing Simple Things

- Note: can represent *any* function as a ROBDD
  - Here is the ROBDD for the function f(x1,x2,...xn) = 0
  - Here is the ROBDD for the function f(x1,x2,...xn) = 1
  - Here is the ROBDD for the function f(x1, ..., x, ..., xn) = x

Binary Decision Diagrams

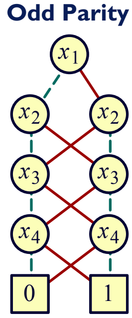
- Assume variable order is X1, X2, X3, X4

Typical Function



- $(x_1 + x_2)x_4$
- **No vertex labeled** x_3
  - independent of x_3
- **Many subgraphs shared**

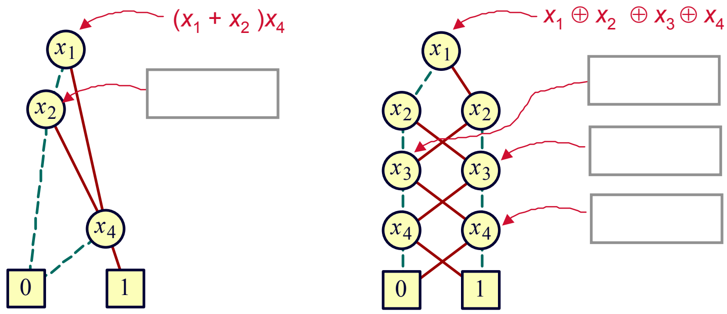
Odd Parity



Linear representation

Sharing in BDDs

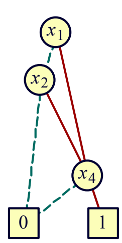
- Technical aside
  - *Every node* in a BDD (in addition to the root) represents *some* Boolean function in a canonical way



  - BDDs are incredibly good at extracting and representing this kind of *sharing* of subfunctions in subgraphs

BDD Applications

- Aside: some nice, immediate applications
  - *Tautology checking*
    - Was complex with the cubelist representation, required divide &conquer algorithm, lots of manipulation
    - With BDDs, it's trivial. Just see if the BDD for function == 1
  - *Satisfiability* == can you find assignment of 0s & 1s to vars to make the function == 1?
    - No idea how to do it with cubelists
    - With BDDs, any path to 1 node from root is a solution

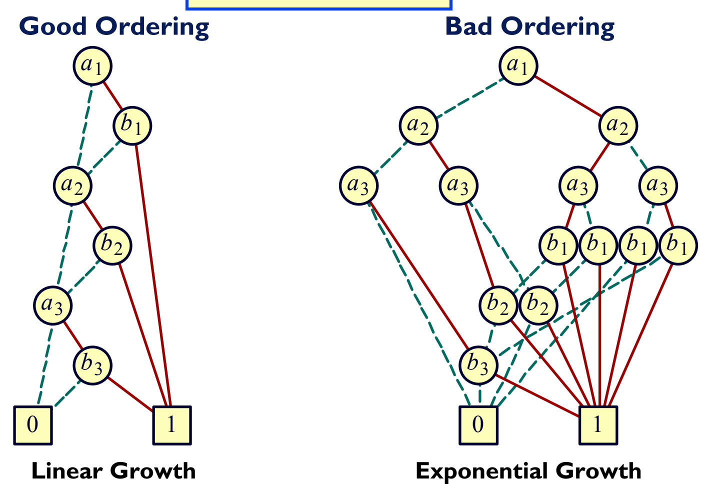


Satisfiability: $x_1 \; x_2 \; x_3 \; x_4$ =

BDD Variable Ordering

- **Question: Does variable ordering matter? YES!**

$a_1 b_1 + a_2 b_2 + a_3 b_3$

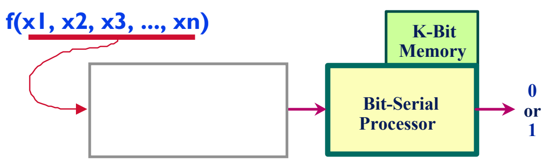


Variable Ordering: Consequences

- **Interesting problem**
  - Some problems that are known to be exponentially hard to solve work out to be very easy on BDDs
  - Trouble is, they are only easy when the size of the BDD that represents the problem is "reasonable"
  - Some input problems make nice (small) BDDs, others make pathological (large) BDDs
  - No universal solution (or else we'd always be able to solve exponentially hard problems easily)
- **How to handle?**
  - **Variable ordering heuristics:** make nice BDDs for reasonable probs
  - Basic characterization of which problems *never* make nice BDDs

Variable Ordering

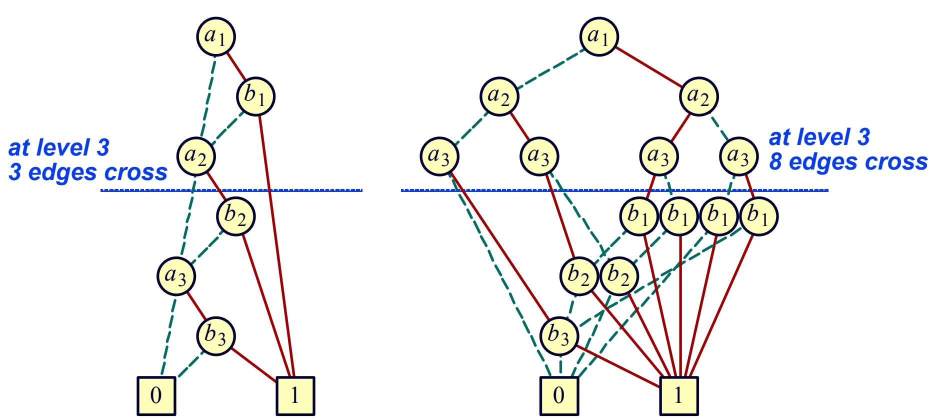
- Analogy to “bit-serial” computing useful here…



- Operation
  - Suppose this machine reads your function inputs 1 bit at a time…
  - …ie, in a certain variable order.
  - Stores information about previous inputs to correctly deduce function value from remaining inputs.
- Relation to OBDD Size
  - If this ‘machine’ requires *K* bits of memory at step i…
  - …then the OBDD has ~ 2^K branches crossing level i.

Variable Ordering: Example

$a_1 b_1 + a_2 b_2 + a_3 b_3$

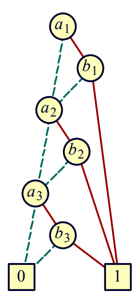


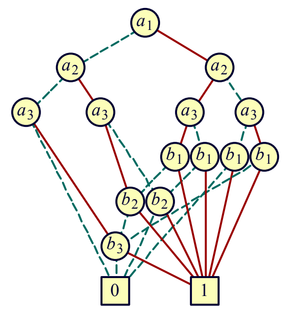
Variable Ordering: Intuition

Idea: Local Computability

- Inputs that are **closely related** should be kept near each other in the variable order
- Groups of inputs that can **determine** the function value by themselves should be close together

$$a_1\, b_1 + a_2\, b_2 + a_3\, b_3$$

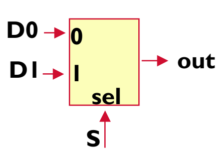




Variable Ordering: Intuition

Idea: Power to control the output

- The inputs that **"greatly affect"** the output should be early in the variable order
- "Greatly affect" means almost always changes the output when this input changes
- Example: multiplexer

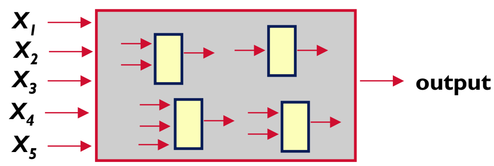


order: S < D0 < D1

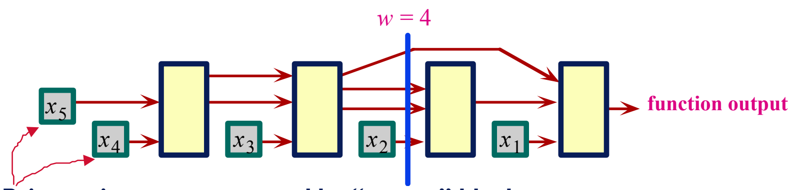
order: D1 < D0 < S

Variable Ordering

- What use is any of this? Suggests ordering heuristic...
  - Suppose I have a logic network like this...

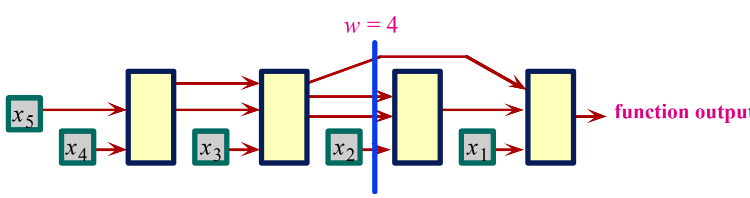


  - Now, redraw to represent circuit as *linear* arrangement of its gates
  - Constraint: all the output-to-input wires go left-to-right in this order
  - Called a *topological* ordering



Primary inputs represented by "source" blocks

Variable Ordering

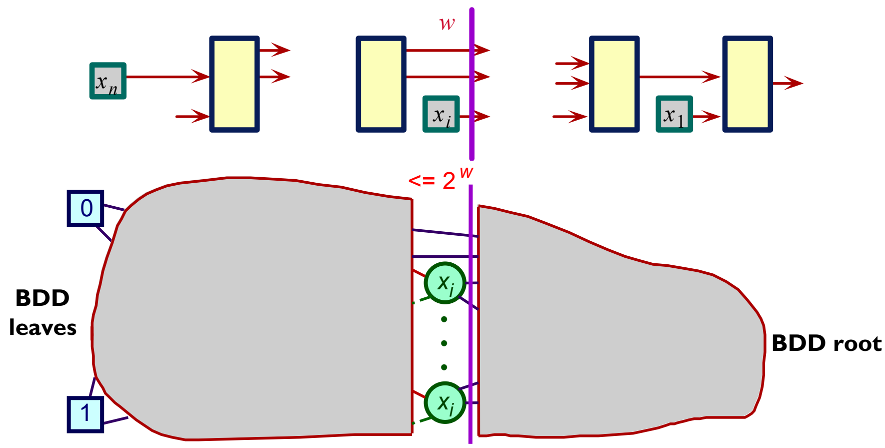


- Parameters
  - Number of primary inputs = n
  - "Bandwidth" = w = number of wires cut at widest point

- Useful result: Size upper bound [Berman, IBM]
  - Can represent with OBDD with <= $n\ 2^w$ nodes
  - Order variables in *reverse* of source block ordering
    - Means list vars right to left in the above picture...

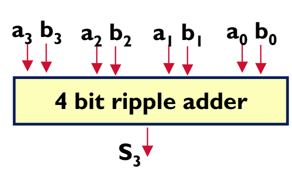
Variable Ordering

- Reasoning here goes like this...
  - All info about vars > i encoded in w bits...
  - ...so at most 2^w distinct decisions, which bounds number of branch destinations from levels < i to levels <= i

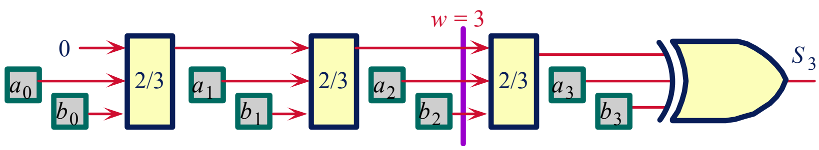


Variable Ordering

- Linear circuit example: 4 bit adder sum, MSB
  - How to order vars for a simple 4-bit carry ripple adder, Sum MSB?



- Answer: Use nice property of our adder circuit
  - It has *Constant* bandwidth => Linear OBDD size



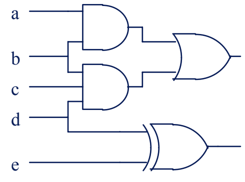
Aside: Variable Ordering

- Generalization
  - Many carry chain circuits have constant bandwidth
  - Examples
    - Comparators
    - Priority encoders
    - ALUs

Variable Ordering Heuristics

- Heuristic ordering methods
  - Take advantage of this "linear ordering" idea
  - Input: gate-level logic network we want to build a BDD for
  - Output: global variable ordering to use
  - Method: topological analysis, aka, "walking" the network graph...

Input Netlist



Ordering

b < a < d < c < e?

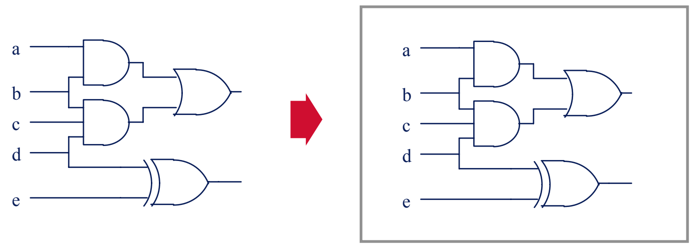
a < b < c < d < e?

e < d < c < b < a?

Example: Dynamic Weight Assignment Heuristic

Concrete example: Minato's heuristic

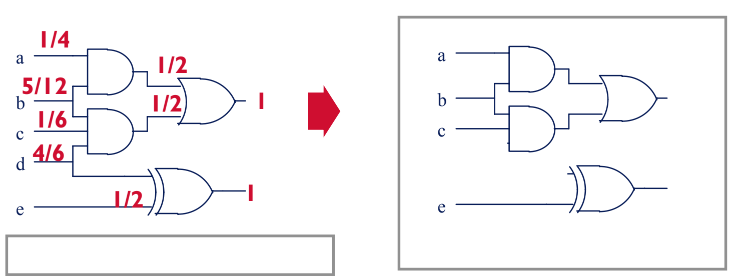
- Pick a primary output; put a weight "1" there
- For each gate with weights on its output but not its input, "push" the weight thru to the inputs, dividing by the number of inputs. Each input gets equal weight.
- If there is fanout (one wire goes to >= 2 inputs) then ADD the weights to get the new weight for this wire.
- If there is more than 1 output, start with the one that has the deepest logic depth from the inputs
- Continue till all primary inputs are labeled



Dynamic Weight Assignment

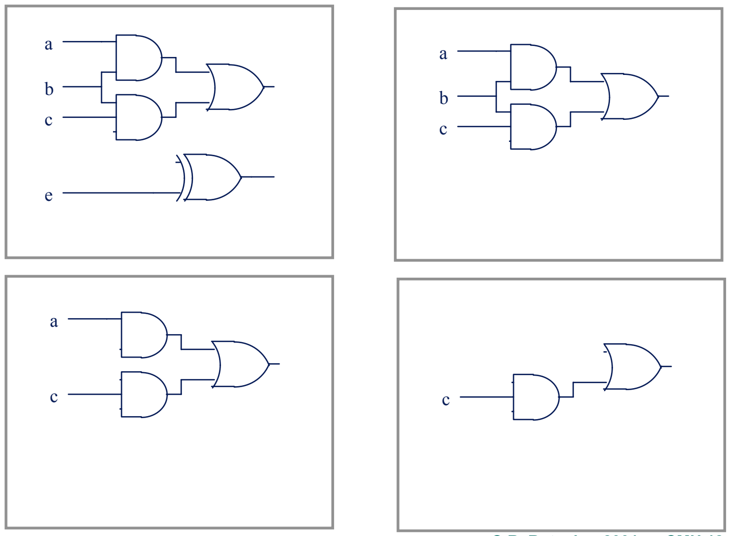
Minato's heuristic

- Pick the primary input with the biggest weight. Put it first in var order.
- Erase the subcircuit (wires, input pins, entire gates if they only one "active" pin left) that are reachable only from this primary input we selected.
- Go back and reassign the weights again in the new, smaller circuit.



Dynamic Weight Assignment

- **Just continue**



Dynamic Weight Assignment

- **Minato's method**
  - Iteratively picks the next variable in the order using the simple weight propagation idea
  - Tries to order all vars starting from the "deepest" output
  - Deletes the ordered var, erases wires/gates, repeats till all ordered

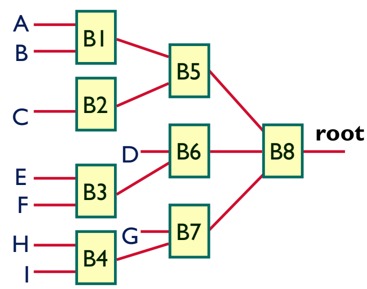
- **How well does it work?**
  - Fairly well. Very simple to do. Lots better than random order.
  - OK complexity == O(#gates • #primary inputs)

- **Notes**
  - There are other, better, more complex heuristics
  - Also, the ordering does NOT have to be static, it can change dynamically as the BDD is used

Variable Ordering Heuristics

- Alternative: Suppose your network is a tree
  - Start at the output
  - Do a postorder traversal of tree
  - Write down variables in order visited by the tree walk
- Remember postorder walk?
  - Visits the nodes, ie, gates, in a deterministic order
  - Ignore primary inputs (for now)

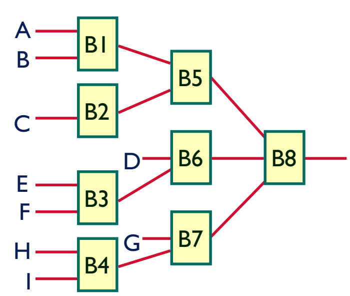
```
postorder (TreeNode) {
  if (TreeNode.TopChild != null)
 postorder( TreeNode.TopChild)
  if (TreeNode.BotChild != null)
 postorder( TreeNode.BotChild)
  write out TreeNode name
}
```

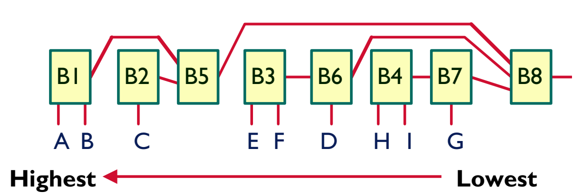


Nodes finished as:

Variable Ordering Heuristics

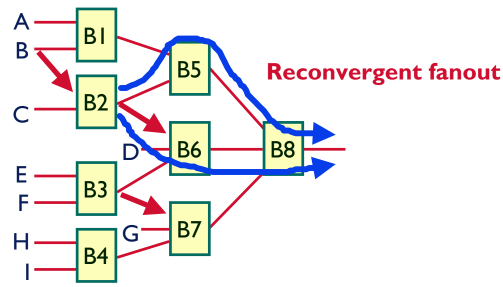
- In our case
  - Tree might not be binary -- not a big deal
  - Just use some consistent order for exploring the children nodes
  - *Visits variables in reverse order*
- Why is this a good heuristic?
  - It makes a linear ordering of ckt
  - Bandwidth is *O(logN)* for N blocks
  - OBDD size is $O(N^2)$





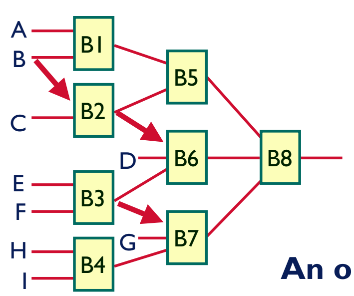
Variable Ordering Heuristics

- What if network is not a tree?
  - **More general, more common case**
  - **Some terminology: Reconvergent fanout**
    - **When one input or intermediate output has *multiple* paths to the final network output, fanout is called *reconvergent***
    - **If you don't have a tree, you have this**



Variable Ordering Heuristics

- For general logic networks
  - **Still try to do a depth-first walk of the graph, output to inputs**
  - **Try to walk the graph like it was a tree, giving priority to nets that have multiple fanouts**



An ordering...

B < A < C < D < E < F < G < H < I

Ordering: Results

Function Class	Best	Worst
Addition	linear	exponential
Symmetric	linear	quadratic
Multiplication	exponential	exponential

General Experience

- Many tasks have reasonable OBDD representations
- Algorithms remain practical for up to millions of OBDD nodes.
- Heuristic ordering methods are generally OK, though it may take effort to find a heuristic that works well for your problem
- So-called dynamic variable ordering -- reordering your BDD vars as your BDD gets used, to improve the size -- is essential today

Binary Decision Diagrams

Variants and optimizations

- Refinements to OBDD representation
- Do not change fundamental properties

Primary Objective

- *Reduce memory requirement*
- Critical resource
- Constant factors matter

Secondary Objective

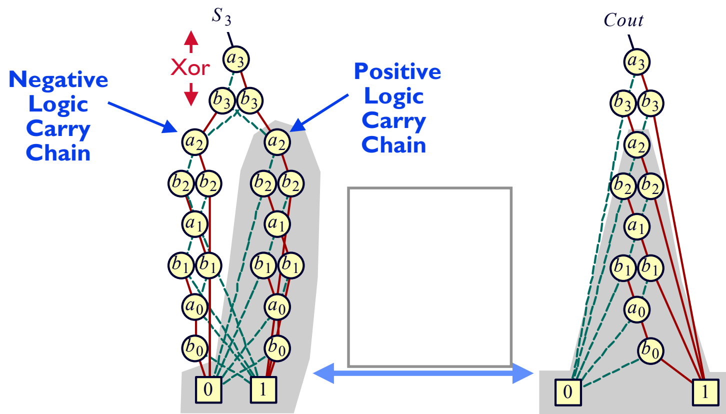
- *Improve Algorithmic Efficiency*
- Make commonly performed operations faster

Common Optimizations

- Share nodes among multiple functions
- Negated arcs

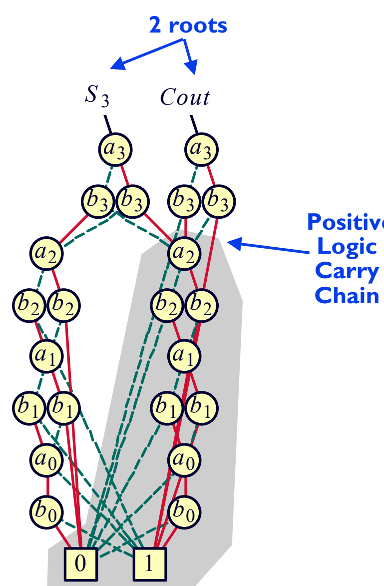
Binary Decision Diagrams: Sharing

- **Sharing, revisited**
  - **We mentioned BDDs good at representing shared subfunctions**
  - **Consider this example from a 4 bit adder: sum msb and carry out**



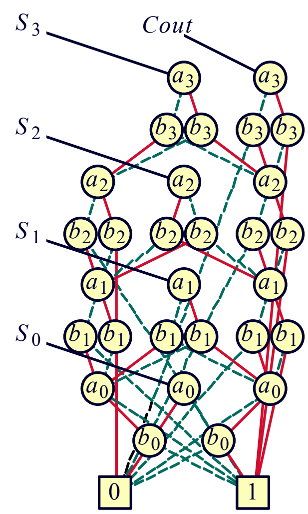
Sharing: Multi-rooted DAG

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



Sharing: Multi-rooted DAG

- **Why stop at 2 roots?**
  - For many collections of functions, there is considerable sharing
  - Idea is to minimize size wrt several separate BDDs by max sharing
- **Example: Adders**
  - Separately
    - 51 nodes for 4-bit adder
    - 12,481 for 64-bit adder
    - *Quadratic growth*
  - Shared
    - 31 nodes for 4-bit adder
    - 571 nodes for 64-bit adder
    - *Linear growth*

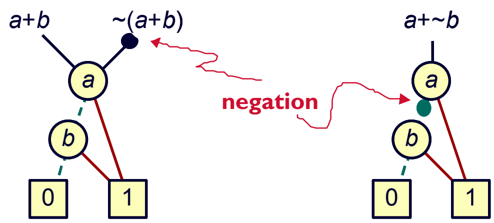


BDD Sharing: Issues

- **Storage model**
  - Single, multi-rooted DAG
  - Function represented by pointer to node in DAG
  - Be careful to apply reduction ops globally to keep all canonical
    - Every time you create a new function, gotta go look in your big multi-rooted DAG to see if it already exists, inside, somewhere
- **Storage management**
  - User cannot know when storage for node can be freed
  - Must implement automatic garbage collection...
    - ...or not try to free any storage
  - *Significantly more complex programming task*
- **Algorithmic efficiency**
  - Functions equivalent if and only if pointers equal
    - if (p1 == p2) …
  - Can test in constant time

Optimization: Negation Arcs

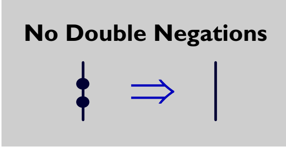
- Concept
  - Dot on arc represents complement operator
    - Inverts function value of BDD reachable "below the dot"
  - Can appear on internal or external arc



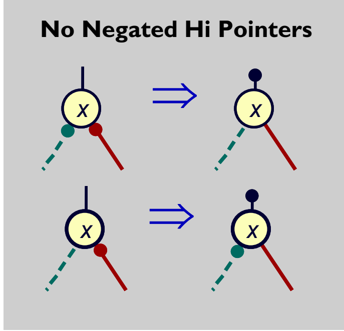
Canonical Form

- Must have *conventions* for use of negative arcs
  - Express as series of transformation rules
  - These are really nothing more than DeMorgan laws

Rule #1



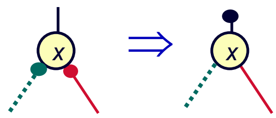
Rule #2



Aside: Why Does This Work...?

- Just like Shannon expansion, applied again
  - ..with prudent use of the basic DeMorgan laws.

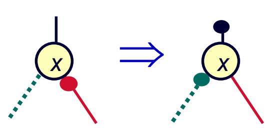
No Negated Hi Pointers



Aside: Why Does This Work...?

- Just like Shannon expansion, applied again

No Negated Hi Pointers



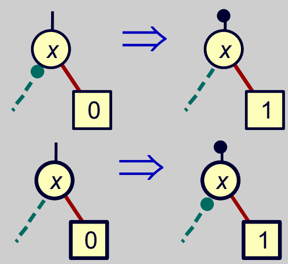
Transformation Rules (Cont.)

Rule #3

No Negated Constants

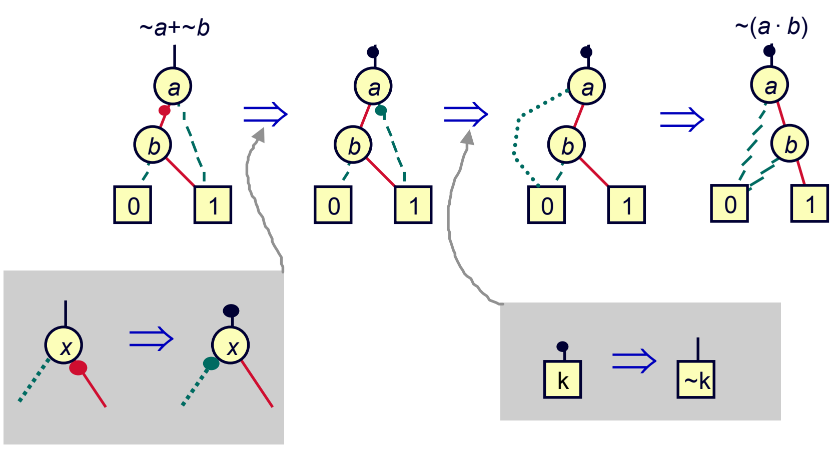
Rule #4

No Hi Pointers to 0

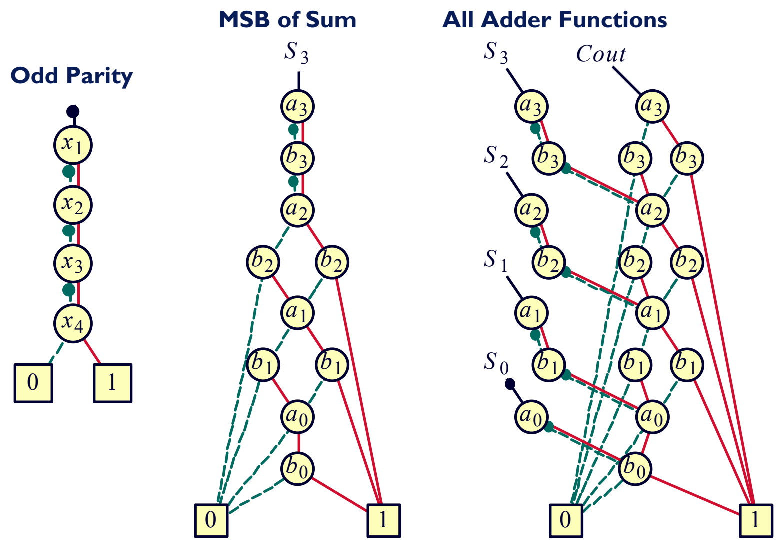


Transformation Example

- Example of applying the rules
  - **Tends to get "nand-like" DAGs**



Negation Arc Examples



Effect of Negation Arcs

- Storage savings
  - At most 2X reduction in number of nodes
- Aside: can people *really* do this “negation” thing in their heads by looking at a normal BDD?
  - Nope
  - Takes lots of practice even to be able *read* these things
  - Just useful because of the 2X space efficiency
- Algorithmic improvement
  - Can complement function in constant time

Summary

- OBDD
  - Reduced graph representation of Boolean function
  - Canonical for given variable ordering
- Selecting good variable ordering critical
  - Minimize OBDD size
  - Circuit embeddings provide effective guidance
- Variants and optimizations
  - Reduce storage requirements
  - Improve algorithmic efficiency
  - Complicate programming and debugging