# Topic 3

## Introduction to
## Binary Decision Diagram (BDD)

系統晶片驗證

SoC Verification

2025.03.05

# Remember...

Formal verification techniques are used in --

1. Equivalence checking
   - Formal engine is used to prove the internal equivalence candidates
2. Property checking
   - Formal engine is used to prove that certain attributes (properties) of the design are always true

➔In either case: proving something is always true

➔Showing that there is no counter example

# Refresh: Proving Invariance Property

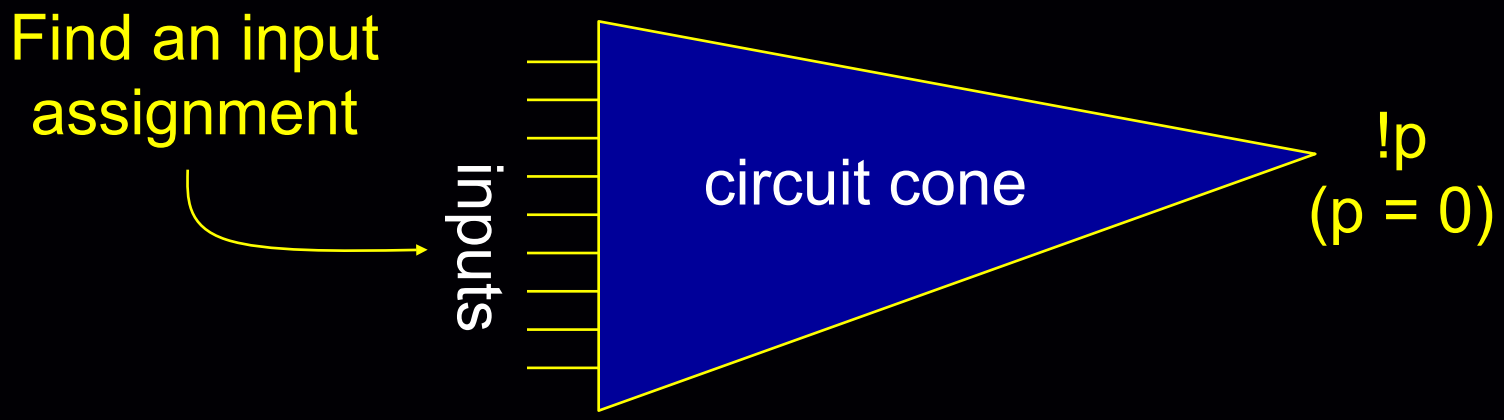
- assert_always(p) ≡ AG (p) ≡ ¬EF (¬p)

- For sequential property ---
  - Compute the set of reachable states, or
  - Generate a sequential trace to (¬p), or
  - Prove that there is no trace to (¬p)

- For combinational property ---
  - Prove (p) is a tautology, or
  - Find an input assignment for (¬p)

# Let’s start from the simpler combinational property

# Tautology Checking ⇔ Test Pattern Generation




- How to solve/prove it?
- p is a function of inputs, so we should define a data structure to represent the function so that we can easily find the input assignment for p = 0.
- In general, function on 'p' can be represented as a multi-level logic equation…

  e.g. p = (a && b || ((c + d) > e)) ^ !g && (a > b)?...
  - String? Netlist? How to operate (e.g. AND, OR) on it?

# Representing Boolean Functions

- As a circuit:
  - Pros: compact in memory usage; easy to perform logic operations(?)
  - Cons: no direct clue in how to acquire the input assignments to satisfy the output target
- As SoP:
  - Pros: simple data structure in implementation; trivial in acquiring input assignments
  - Cons: no easy way to transform problems into SoP form; extra variables are usually needed
- As PoS:
  - Pros: simple data structure in implementation
  - Cons: extra variables are usually needed to construct PoS; NP complete problem to solve the input assignments

➔ More to cover in the “SAT” lecture note

# Enumeration Method (Truth Table)

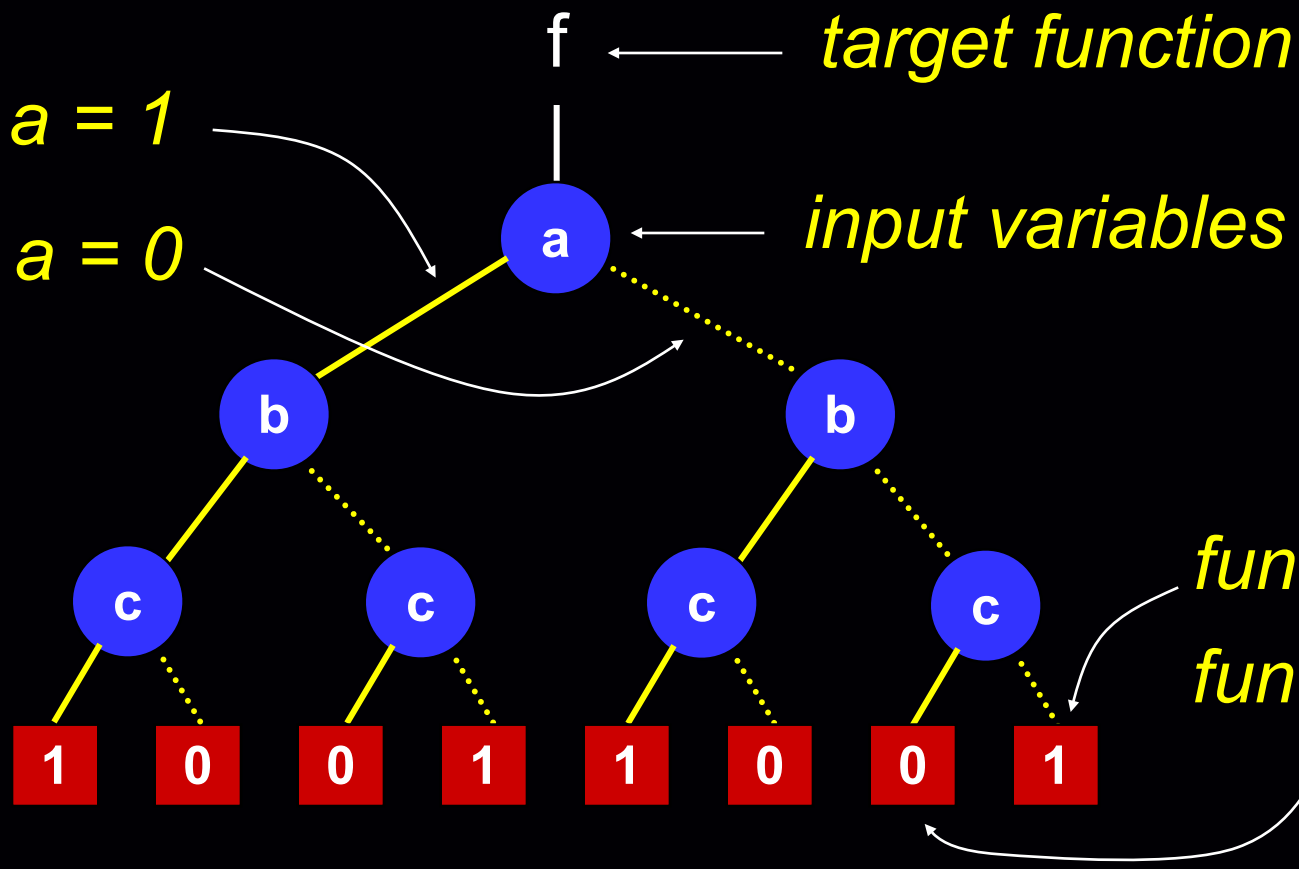
1. For each variable, enumerate its possible values
   - Usually convert a variable into Boolean variables
2. Explicitly list all the combinations of variable values
   - e.g. 2-bit adder

| a1 | a0 | b1 | b0 | f |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| ..... | | | | |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

   - If the truth table can be constructed, to prove the tautology or find a test vector is trivial
   - However, the size of the truth table is exponential in terms of input size (like simulation)

A better data structure to represent truth table?

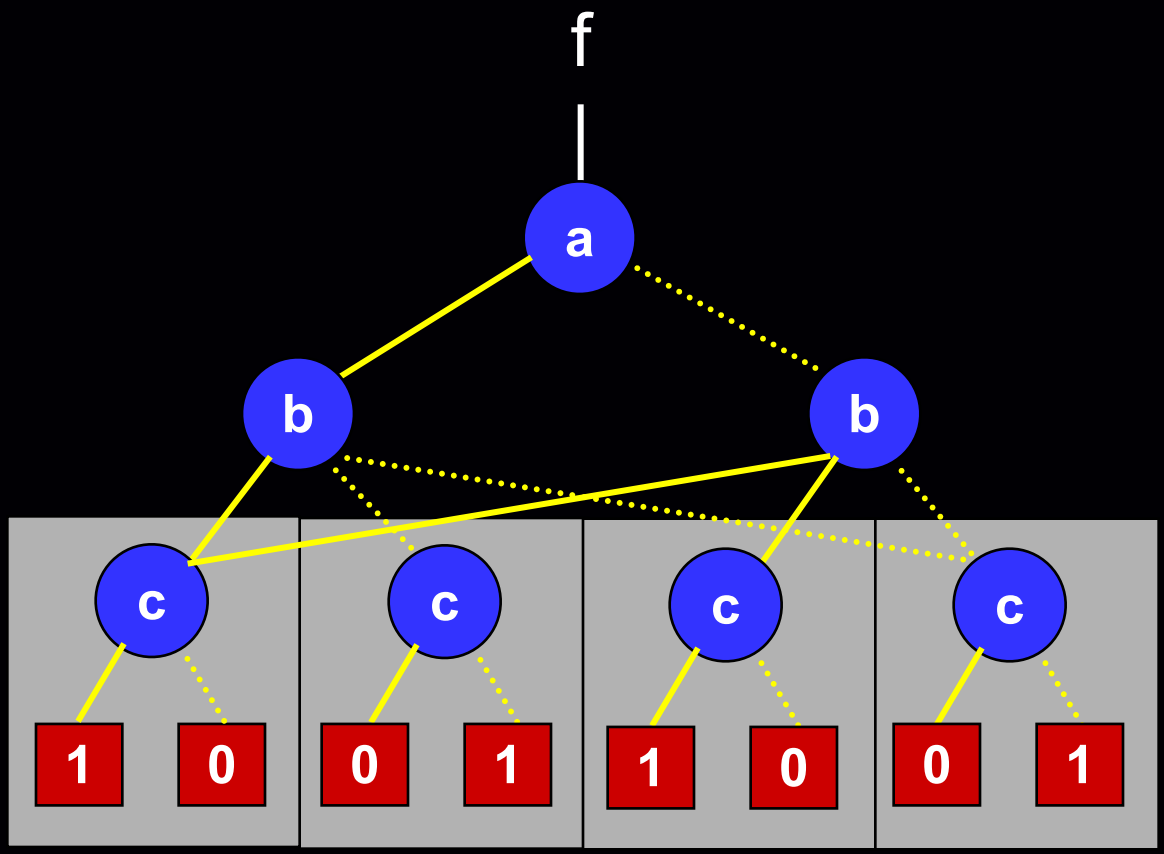
# Binary Decision Tree



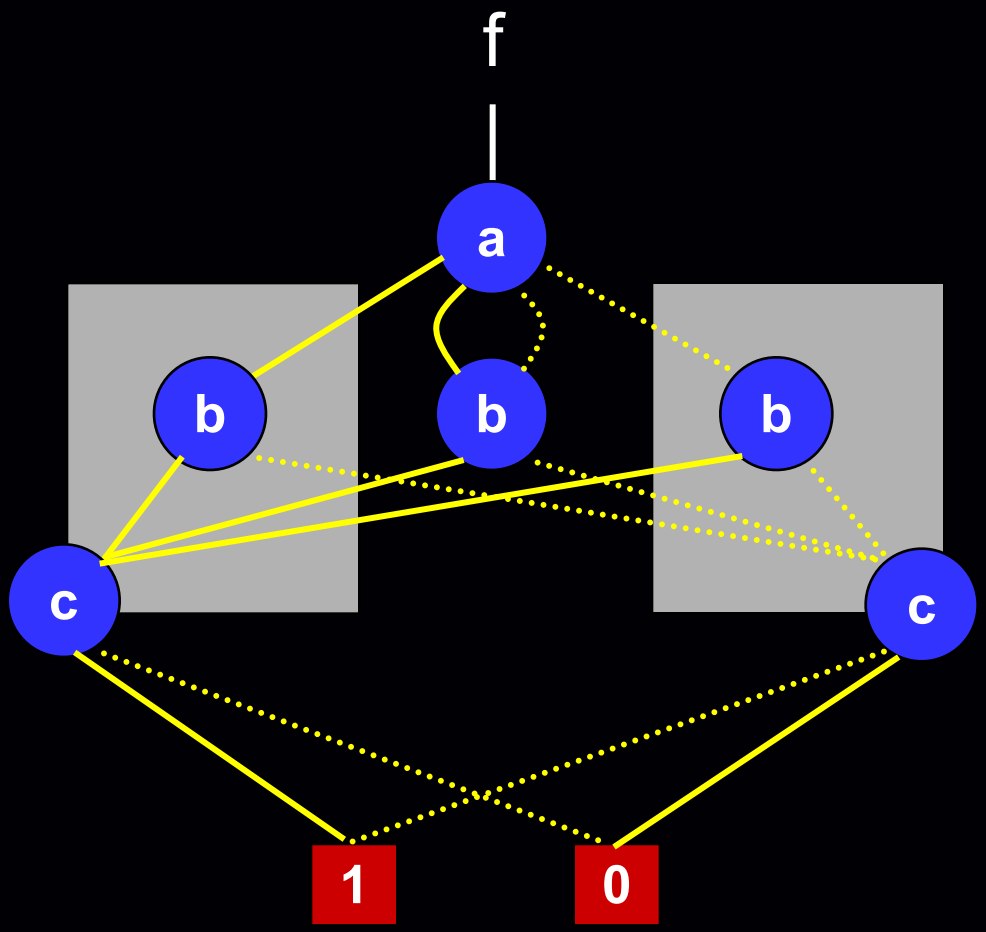

| a | b | c | f |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

- Still exponential in size

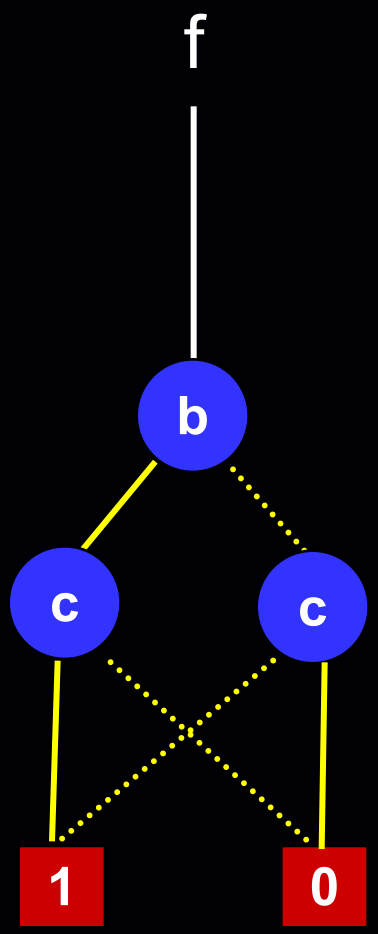
# Binary Decision Diagram



# Binary Decision Diagram

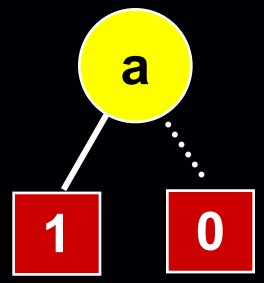


# Reduced Ordered Binary Decision Diagram (ROBDD)

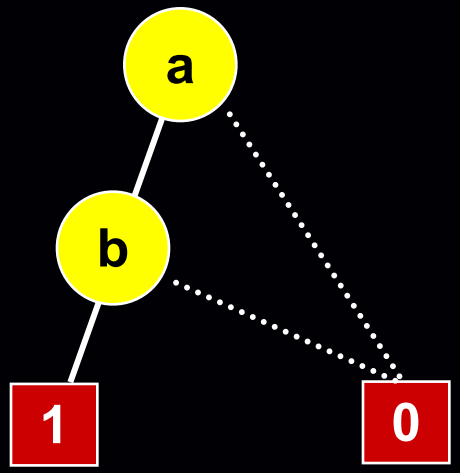



| a | b | c | f |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

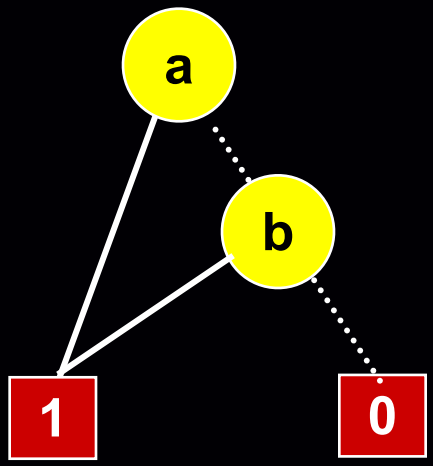
# More BDD Examples



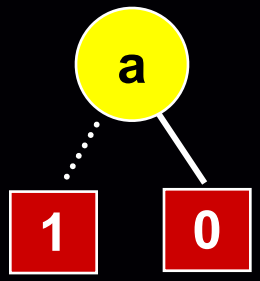

a



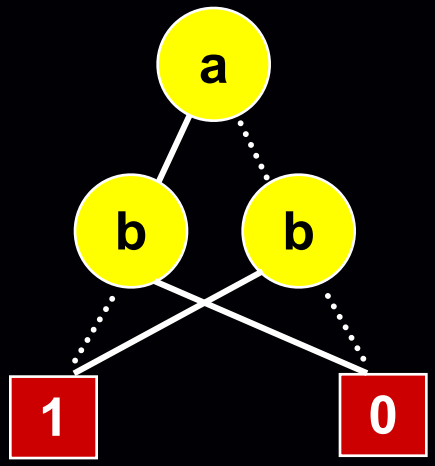

a && b




a || b




!a




a ⊕ b

# Reduced Ordered Binary Decision Diagram (ROBDD)

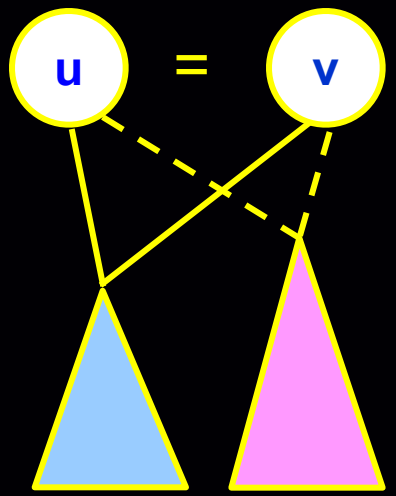
- A graphical representation of truth table
  1. Each level corresponds to an input variable
     → Set of inputs is called "support"
  2. Each node (and its sub-graph) represents a function
  3. Each path represents *a cube* of the function
     → i.e. an input pattern for this function
  4. Functions with equivalent functionality (sub-graph) are merged together
     - Always canonical

# ROBDD Reduction Rules (1/2)

1. Uniqueness
   - No two distinct nodes u and v have the same level and left- and right-successors
   1. level(u) = level(v)
   2. left(u) = left(v)
   3. right(u) = right(v)

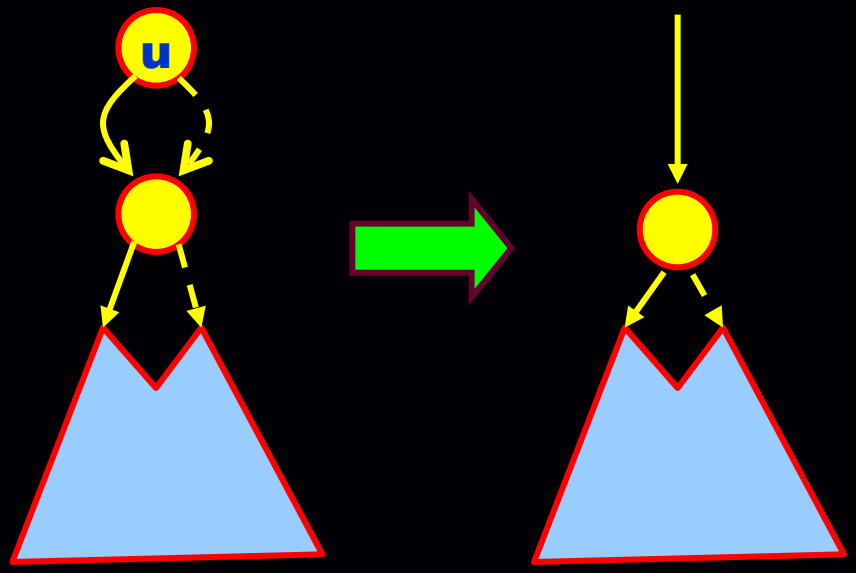
   → node(u) = node(v)

Hash(level, left, right) → BDD node



# ROBDD Reduction Rules (2/2)

2. Non-redundant tests
   - No variable node u has identical left- and right- successor.
     - For each node, left(u) ≠right(u)



# ROBDD Characteristics

1. Canonicity
   - f = g iff BDD(f) = BDD(g)
   - e.g.

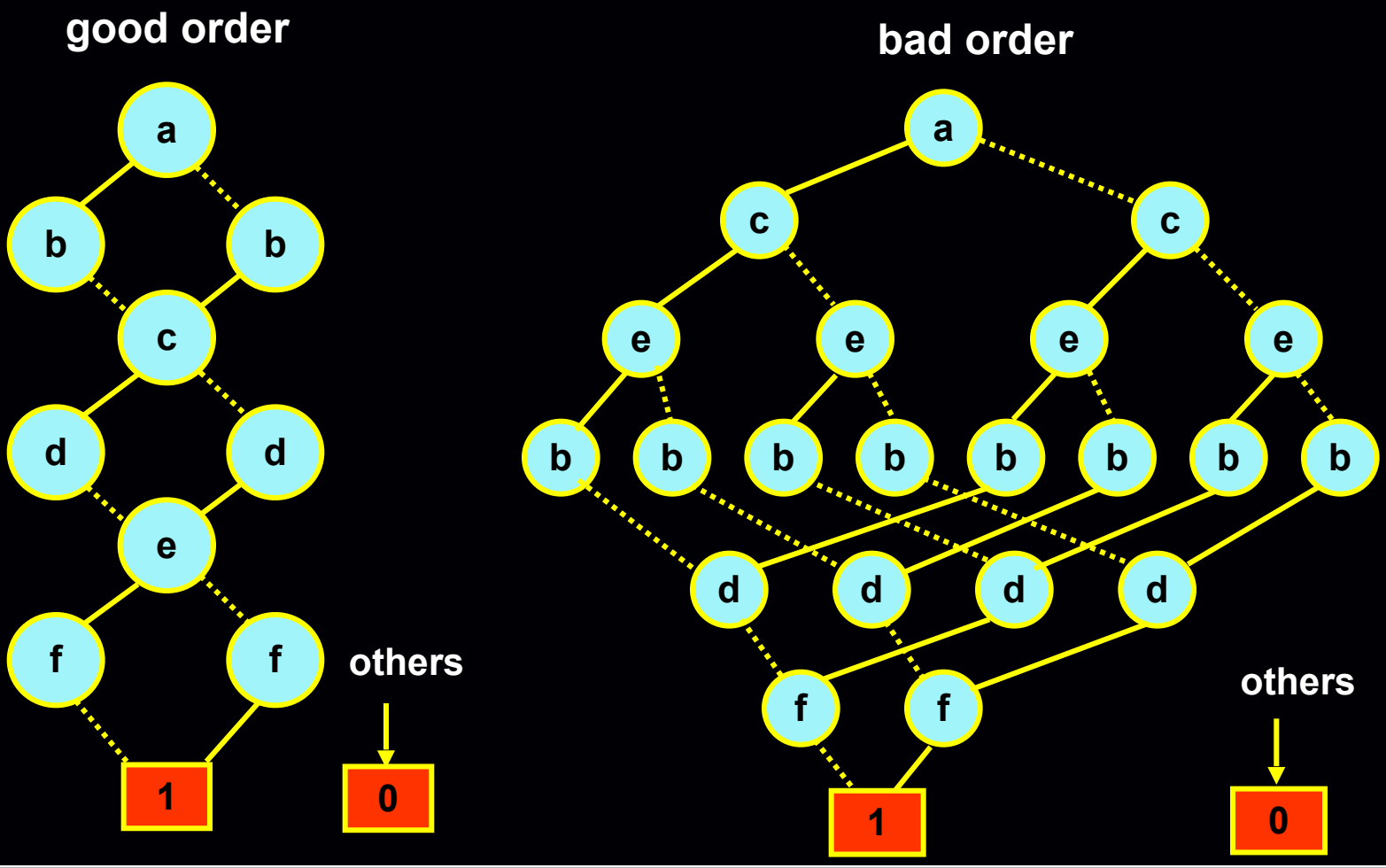
     f = ab + ac

     g = a(b+c)

     ➔ BDD(f) = BDD(g)
2. Ordering dependency
   - Given the same functionality, different input variable orderings may lead to significant difference in the number of BDD nodes

# Example on Variable Ordering

$$z = (a \oplus b) \cdot (c \oplus d) \cdot (e \oplus f)$$



# The Influence of Variable Ordering

- Size of BDD
  - Can vary from linear to exponential in the number of the variables, depending on the ordering
- Hard-to-Build BDD
  - Data path components (e.g., multipliers) cannot be represented in polynomial space, regardless of the variable ordering
- Heuristics of Ordering

  (1) Put variables that influence most on the top of BDD

  (2) Minimize the distance between strongly related variables

  (e.g., x1x2 + x2x3 + x3x4)

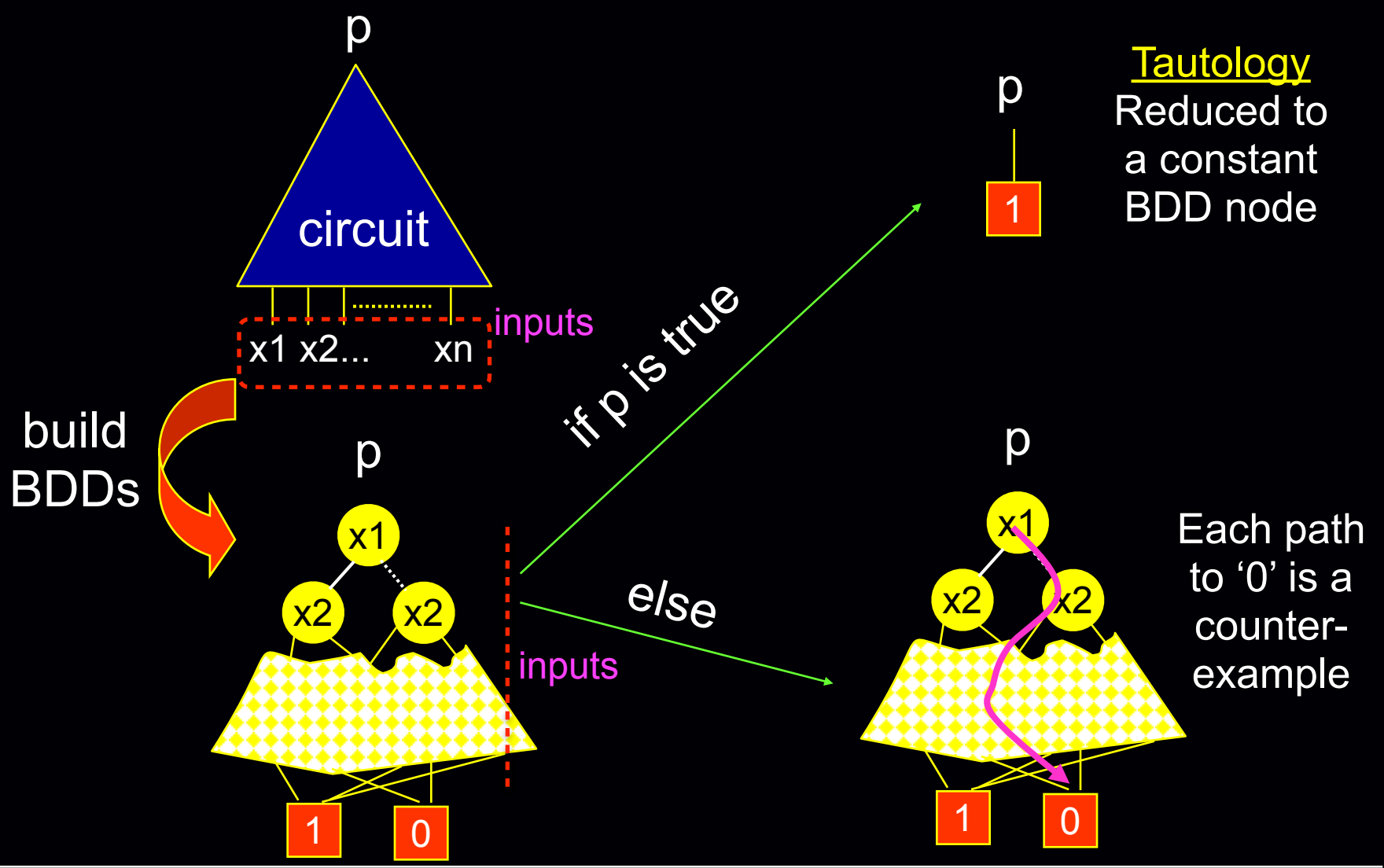
  x1→x2→x3→x4 is better than x1→x4→x2→x3
- Dynamic variable reordering (e.g. “sifting”, covered later)

Now we know BDD can be used to represent functions

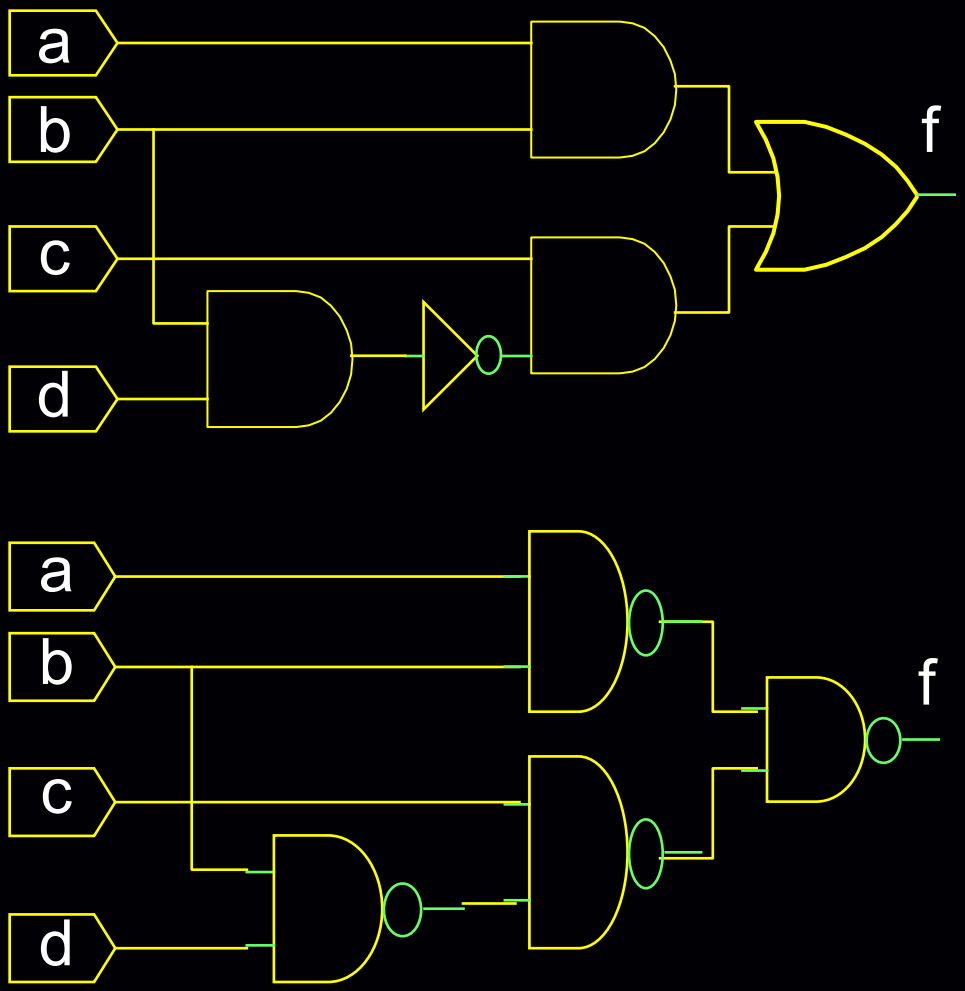
To prove a (circuit) invariance property

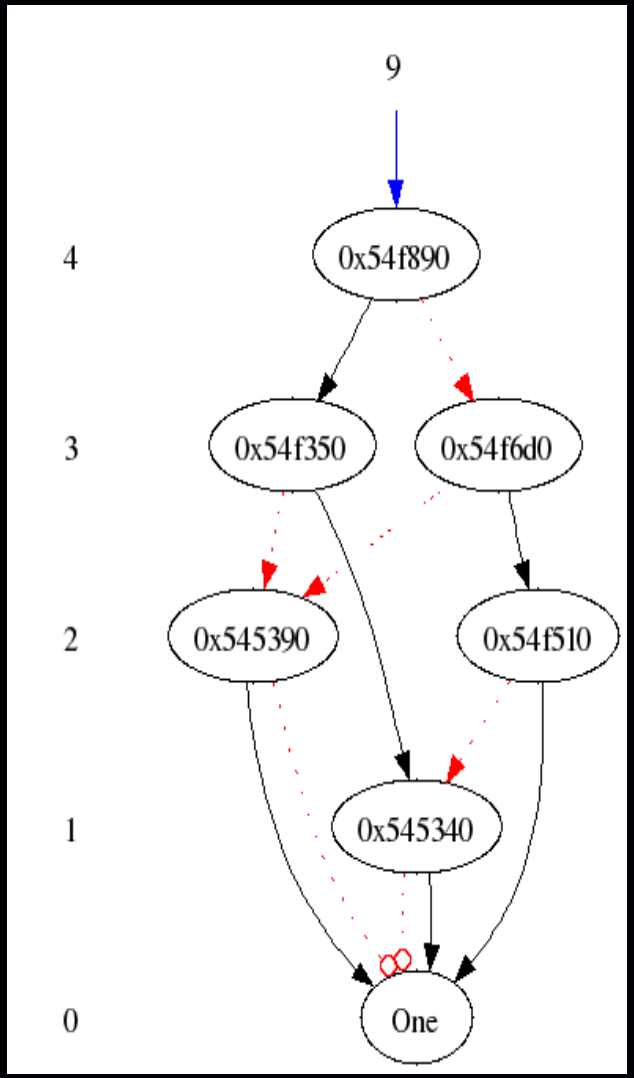
→ Need to construct BDD for the function of the circuit

# Proving Invariance Property by BDD



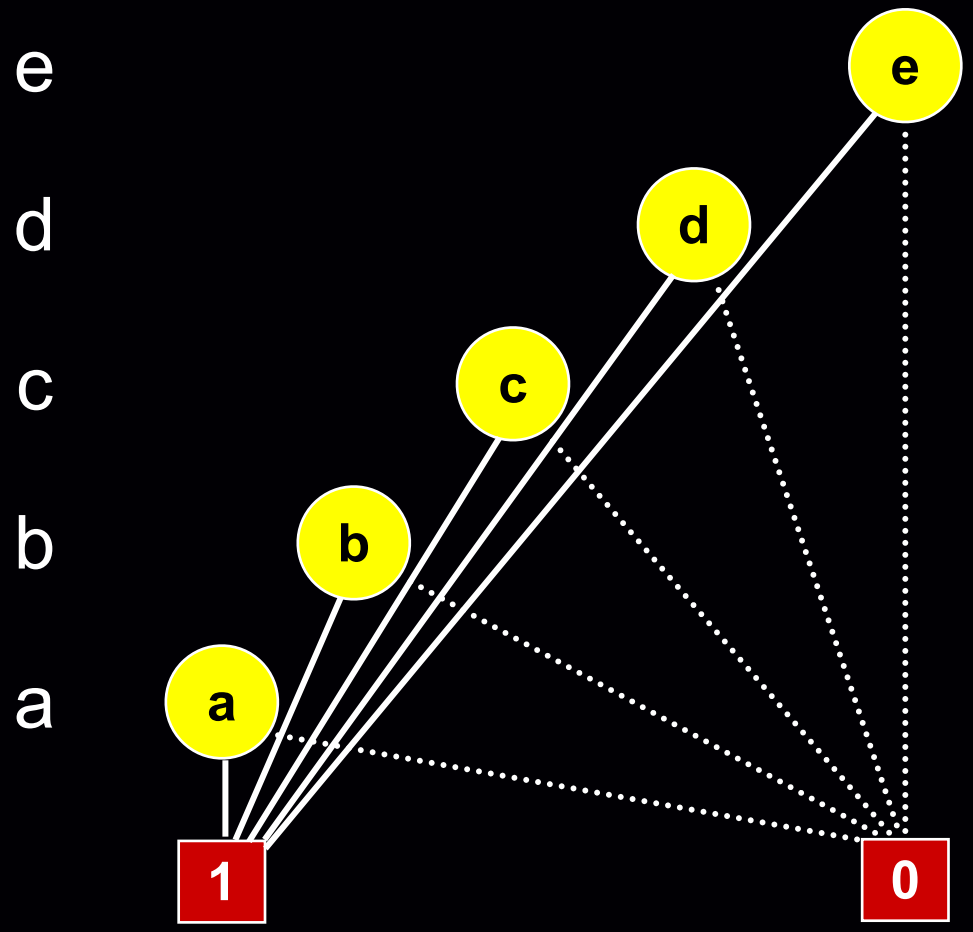
# Circuit to BDD Example (C17)



# How to construct BDDs from circuit?

# First, determine the variable order and construct the BDDs for PIs



# Building BDD from Circuit

- Perform topological sorting on the gates in the circuit
  - ➔ Order the gates from PIs to Pos

```
for each gate in the sorted list {
   // all the BDDs of its fanins have been built
   construct BDD for this gate
   record (gate, BDD node)
}
```

- OR ➔ // Use recursive call (post-order traversal)

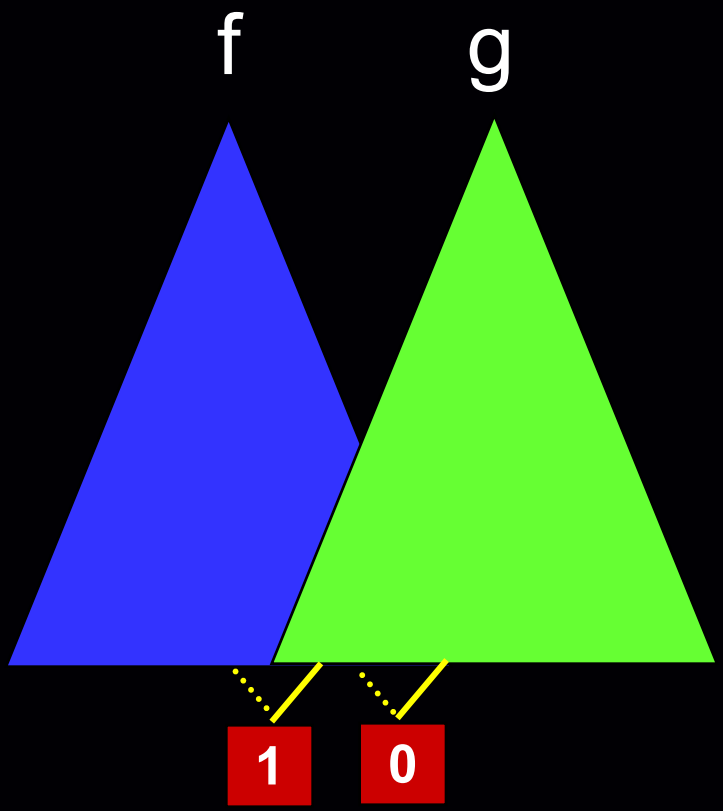
```
for each circuit PO { buildBDD(poGate); }
buildBDD(gate) {
   for each fanin (gate, fanin)
      faninBDDs.push_back(buildBDD(fanin));
   buildBDDNode(type, gate, faninBDDs);
}
```

➔ Basic step:

Given a set of (fanin) BDDs, perform BDD operation for the gate type

# In other words...

Given a gate y = AND(f, g), and BDDs for f and g ---



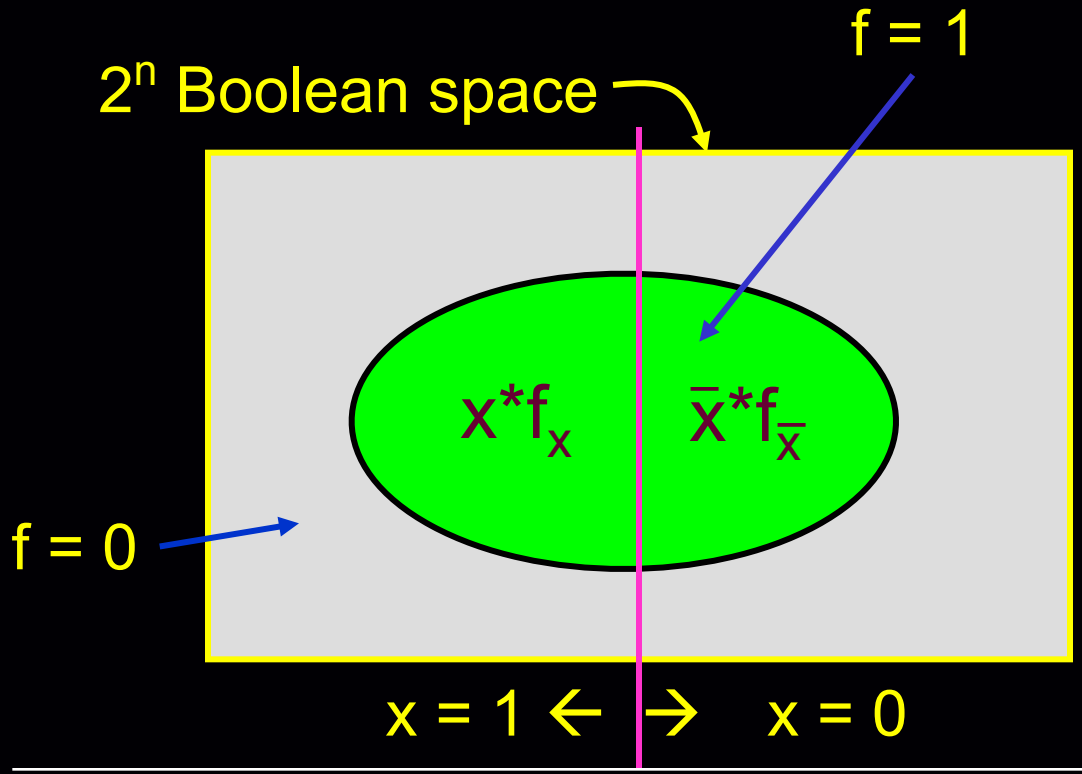

What's the BDD for y = (f && g)?

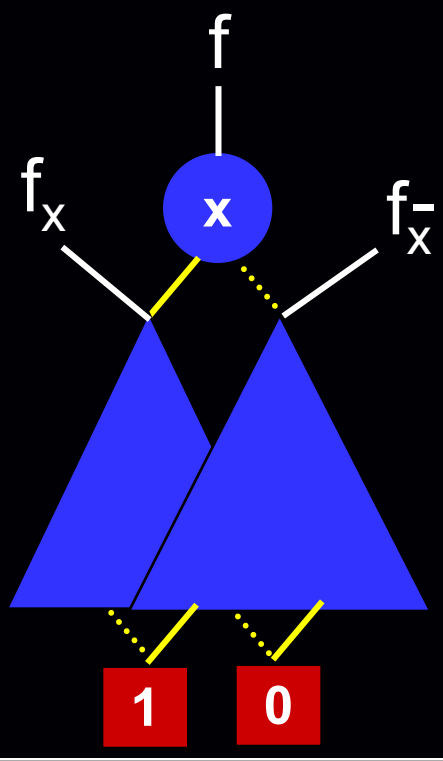

BDD(f && g) = ?

# BDD Cofactors: Binary Function Decomposition

- Shannon expansion of f
  - $f = x * f_x + \bar{x} * f_{\bar{x}}$

➔ $f_x$ and $f_{\bar{x}}$ are called positive/negative cofactors



# Cofactor Examples

- $f = a\bar{b}c + \bar{a}d + b\bar{d}$

➔

$f_a = \bar{b}c + b\bar{d}$ // remove a, drop $\bar{a}$ term,
// keep non-a term (i.e. $f|_{a=1}$)

$f_{\bar{a}} = d + b\bar{d}$

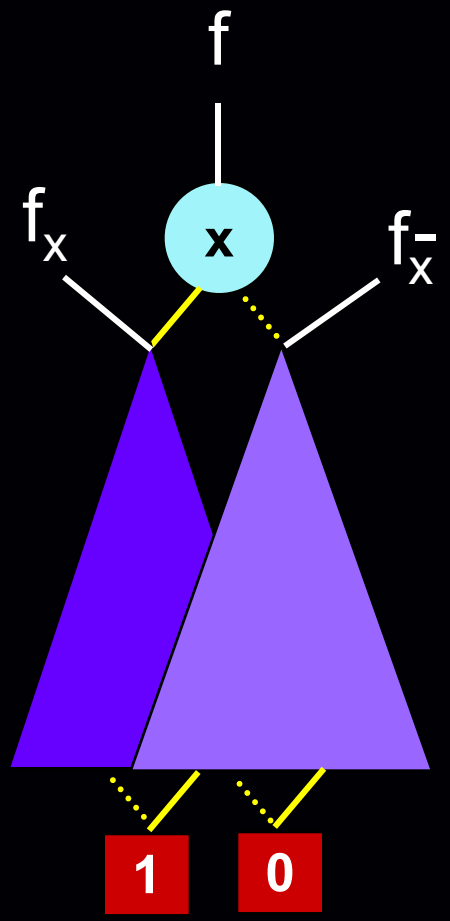
$a * f_a = ?$

$\bar{a} * f_{\bar{a}} = ?$

$a * f_a + \bar{a} * f_{\bar{a}} = ?$ What is $f_e$?

# BDD Operations

- Shannon expansion of f
  - $f = x * f_x + \overline{x} * f_{\overline{x}}$
- f * g
  $= (x * f_x + \overline{x} * f_{\overline{x}}) *$
  $(x * g_x + \overline{x} * g_{\overline{x}})$
  $= x * (f_x * g_x) + \overline{x} * (f_{\overline{x}} * g_{\overline{x}})$
- f + g
  $= x * (f_x + g_x) + \overline{x} * (f_{\overline{x}} + g_{\overline{x}})$
- Operation:
  perform on cofactors individually → Recursion



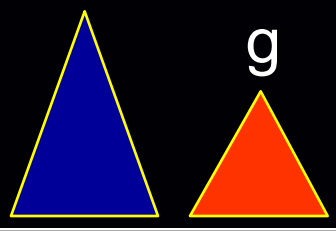
# Recursive Function Operations

- Boolean operation of 2 BDDs can be performed recursively on their cofactors
  - $f * g = x * (f_x * g_x) + \bar{x} * (f_{\bar{x}} * g_{\bar{x}})$
  - $f + g = x * (f_x + g_x) + \bar{x} * (f_{\bar{x}} + g_{\bar{x}})$

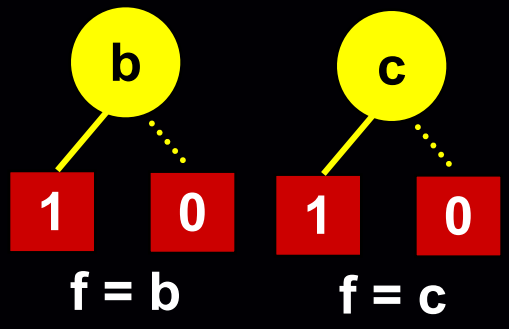
➔ Terminal cases for this recursion are the operations on constants "0" and "1"

- Let u, v be the top variables of BDDs f, and g
  - If (u > v) // i.e. u is closer to the top

$$f * g = u * (f_u * g) + \bar{u} * (f_{\bar{u}} * g)$$

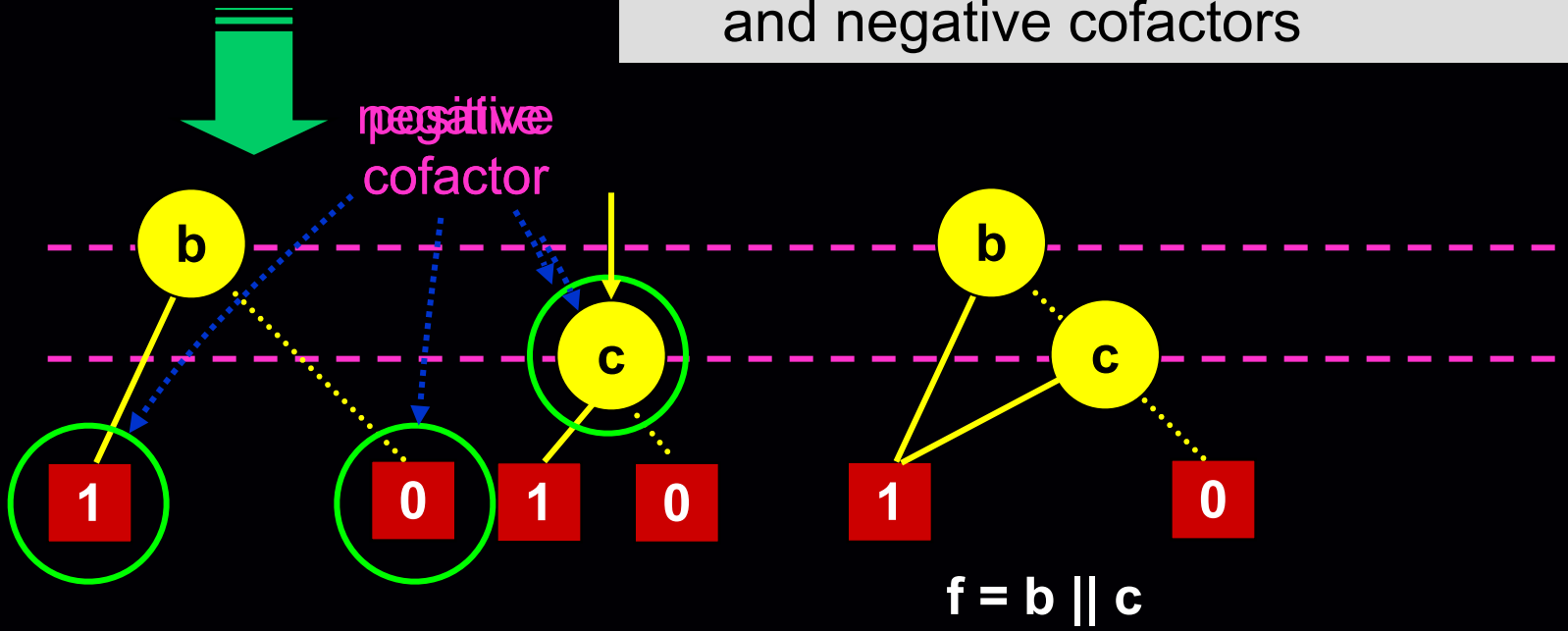


# BDD Operation Example

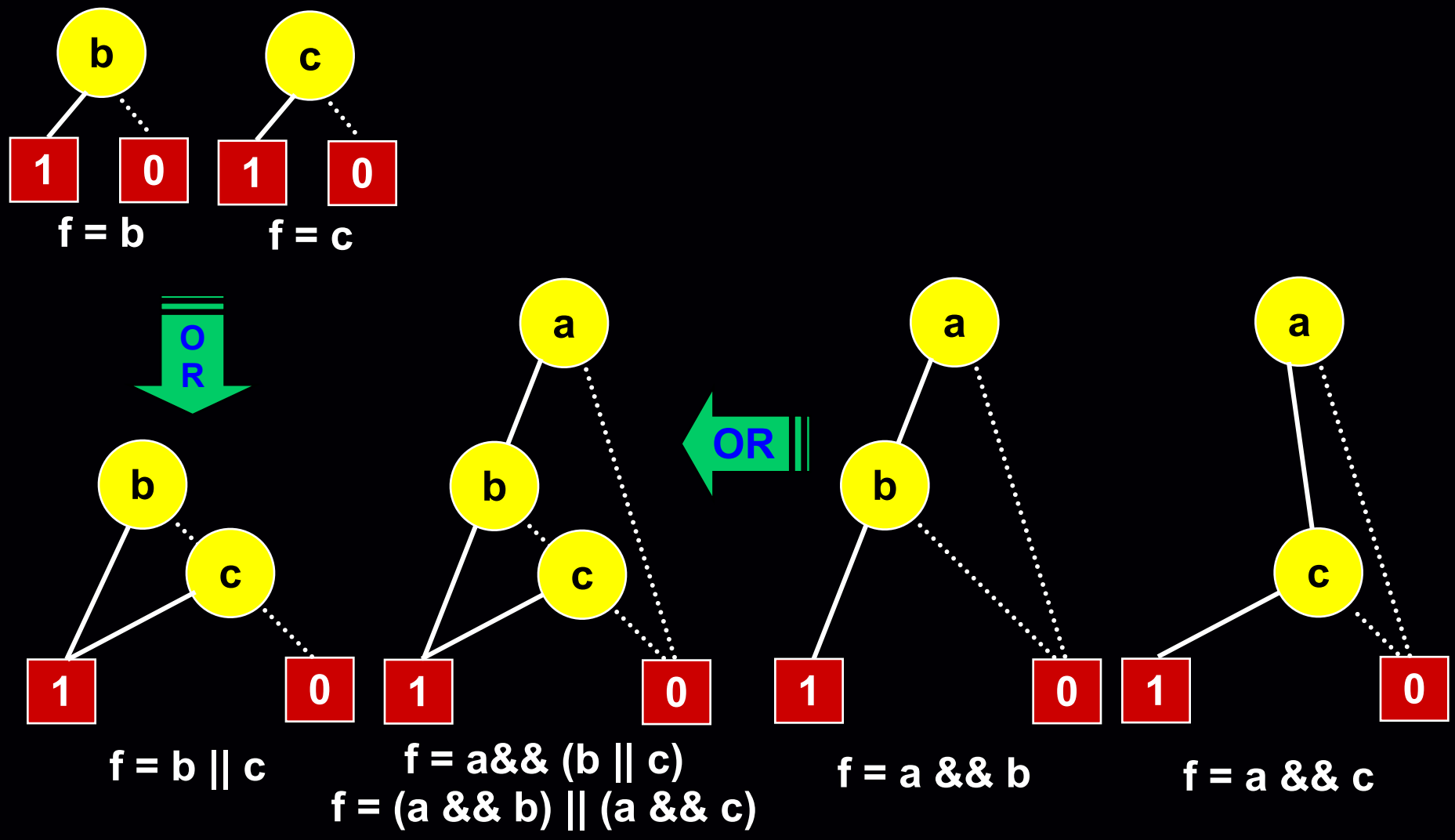



(e.g. Perform “b || c”)

- Decide and place variables in individual levels
- Start from the top variable, recursively apply the computation for its positive and negative cofactors



# BDD Operation Example



In short,

1. f = a && (b || c)
2. f = (a && b) || (a && c)

➔ will result in the same BDD

➔ independent of building orders

(ROBDD canonicity characteristics)

# Recursive BDD Operations on Cofactors

- Do we implement BDD operations this way?
  - (repeated computation)

    For example, $\overline{(f * g)}$ and $(\bar{f} + \bar{g})$ result in the same BDD. However, we need to recursively compute it twice...

    ➔ No "caching" effect....
  - (not good enough)

    For each new operator, need to define the evaluation of its terminal cases

# Introducing the "ITE Operator" --- to enhance BDD construction efficiency

- ITE stands for "If-Then-Else"
- ITE(F, G, H) = F * G + $\bar{F}$ * H
  - e.g. For Shannon Expansion
    - f = ITE(x, $f_x$, $f_{\bar{x}}$) = x * $f_x$ + $\bar{x}$ * $f_{\bar{x}}$  // x: top variable
- All unary/binary Boolean operations can be implemented by "ite" operators
  - AND(F, G) = ITE(F, G, 0)
  - OR(F, G) = ITE(F, 1, G)
  - NOT(F) = ITE(F, 0, 1)

# Using ITE Operators for Boolean Functions

| Table | Name | Expression | Equivalent form |
|---|---|---|---|
| 0000 | 0 | 0 | 0 |
| 0001 | AND(F,G) | $F \cdot G$ | $ite(F, G, 0)$ |
| 0010 | F > G | $F \cdot \overline{G}$ | $ite(F, \overline{G}, 0)$ |
| 0011 | F | $F$ | $F$ |
| 0100 | F < G | $\overline{F} \cdot G$ | $ite(F, 0, G)$ |
| 0101 | G | $G$ | $G$ |
| 0110 | XOR(F,G) | $F \oplus G$ | $ite(F, \overline{G}, G)$ |
| 0111 | OR(F,G) | $F + G$ | $ite(F, 1, G)$ |
| 1000 | NOR(F,G) | $\overline{F + G}$ | $ite(F, 0, \overline{G})$ |
| 1001 | XNOR(F,G) | $\overline{F \oplus G}$ | $ite(F, G, \overline{G})$ |
| 1010 | NOT(G) | $\overline{G}$ | $ite(G, 0, 1)$ |
| 1011 | F ≥ G | $F + \overline{G}$ | $ite(F, 1, \overline{G})$ |
| 1100 | NOT(F) | $\overline{F}$ | $ite(F, 0, 1)$ |
| 1101 | F ≤ G | $\overline{F} + G$ | $ite(F, G, 1)$ |
| 1110 | NAND(F,G) | $\overline{F \cdot G}$ | $ite(F, \overline{G}, 1)$ |
| 1111 | 1 | 1 | 1 |

# Using ITE to enhance BDD construction efficiency

- For example,

BDD ( $\overline{f * g}$ )
= $\overline{ITE ( f, g, 0 )}$

BDD ( $\overline{f} + \overline{g}$ )
= ITE ( $\overline{f}$, 1, $\overline{g}$ )
= ITE ( f, $\overline{g}$, 1 )
= $\overline{ITE ( f, g, 0 )}$

Same ITE operation!!

➔ DeMorgan's rule $\overline{(f * g)} = (\overline{f} + \overline{g})$

➔ ITE conversion rules to be covered later

# How to apply ITE in BDD construction?

1. Given two BDDs, A and B
   ➔ Convert the Boolean operation on (A, B) to ITE representation, say ITE(F, G, H)

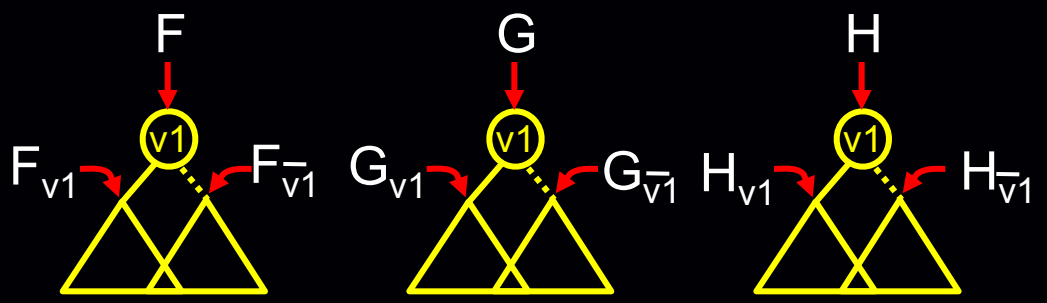
2. Normalize ITE(F, G, H) to ITE(F', G', H')
   ➔ Check: has ITE(F', G', H') been computed?
   ➔ If yes, retrieve the previously computed result

3. Otherwise, recursively compute ITE(F', G', H')
   ➔ Record the { ITE(F', G', H'), result } pair in a cache/hash

# Recursive Algorithm for ITE Operation

- Let Z = ITE(F, G, H), and { v1, v2, ... } be the variable order of Z from top to bottom
  - Z = ITE(F, G, H)

    = ITE(v1, ITE($F_{v1}$, $G_{v1}$, $H_{v1}$), ITE($F_{\bar{v1}}$, $G_{\bar{v1}}$, $H_{\bar{v1}}$)) // 2 cases




= ITE(v1, ITE(v2, ITE($F_{v1v2}$, $G_{v1v2}$, $H_{v1v2}$), // 4 cases

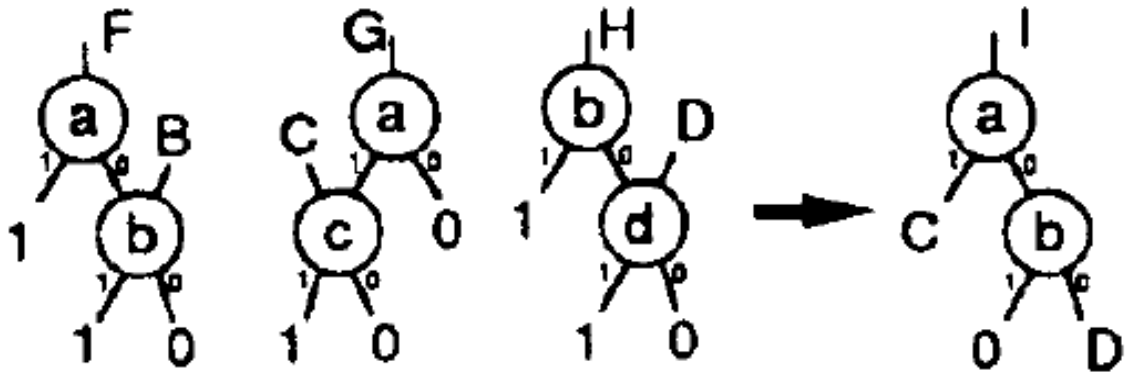
ITE($F_{v1\bar{v2}}$, $G_{v1\bar{v2}}$, $H_{v1\bar{v2}}$)),

ITE(v2, ITE($F_{\bar{v1}v2}$, $G_{\bar{v1}v2}$, $H_{\bar{v1}v2}$),

ITE($F_{\bar{v1}\bar{v2}}$, $G_{\bar{v1}\bar{v2}}$, $H_{\bar{v1}\bar{v2}}$)))

= ... // until terminal cases

# Example of ITE Operations




$$
\begin{aligned}
I &= ite(F, G, H) \\
&= (a, ite(F_a, G_a, H_a), ite(F_{\overline{a}}, G_{\overline{a}}, H_{\overline{a}})) \\
&= (a, ite(1, C, H), ite(B, 0, H)) \\
&= (a, C, (b, ite(B_b, 0_b, H_b), ite(B_{\overline{b}}, 0_{\overline{b}}, H_{\overline{b}}))) \\
&= (a, C, (b, ite(1, 0, 1), ite(0, 0, D))) \\
&= (a, C, (b, 0, D))
\end{aligned}
$$

# Terminal Cases for ITE Algorithm

- What do the terminal cases mean?
  - ITE on constant values?

- No, actually we can terminate earlier
  - F = ite(G, F, F) = ite(1, F, G) = ite(0, G, F) = ite(F, 1, 0)

  ➔ Return F

  (FYI, the only 4 cases you need to check in your BDD pkg)

- What are NOT terminal cases?
  - ite(F, G, 0) = F*G;  ite(F, G, 1) = not(F) + G
  - ite(F, 0, 1) = not(F)
  - ite(0/1, 0/1, H) ➔ covered by terminal cases
  - How about ite(F, 0, 0) and ite(F, 1, 1)??

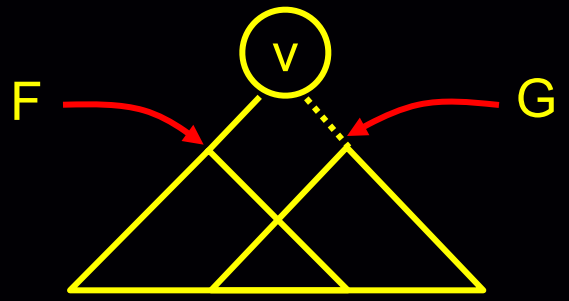
# How to apply ITE in BDD construction?

1. Given two BDDs, A and B
   ➔ Convert the Boolean operation on (A, B) to ITE representation, say ITE(F, G, H)

2. Normalize ITE(F, G, H) to ITE(F', G', H')
   ➔ Check: has ITE(F', G', H') been computed?
   ➔ If yes, retrieve the previously computed result

3. Otherwise, recursively compute ITE(F', G', H')
   ➔ Record the { ITE(F', G', H'), result } pair in a cache/hash

# Hash/Cache in BDD constructions




1. Unique table
   v: top variable
   F, G: BddNode
   ➔ Hash(v, F, G) to an unique BddNode
   ➔ Ensure “canonicity”
2. Computed table (cache)
   F, G, H: BddNode
   ➔ Cache { ITE(F, G, H), result } to the computed BddNode
   ➔ Don’t compute the same thing again!!

Complexity of ITE(F, G, H) can be as good as O(|F|*|G|*|H|)
(Why??)

# Remember: using ITE to enhance sharing

- For example,

BDD ( $\overline{f * g}$ )
= $\overline{\text{ITE ( f, g, 0 )}}$

BDD ( $\overline{f} + \overline{g}$ )
= ITE ( $\overline{f}$, 1, $\overline{g}$ )
= ITE ( f, $\overline{g}$, 1 )
= $\overline{\text{ITE ( f, g, 0 )}}$

Same ITE operation!!

➔What about ITE(F, 0, 1) and ITE(F, 1, 0)?

➔Do they share the same sub-BDD nodes?

# Improving the BDD node sharing

- Complexity to compute $\overline{F}$?

➔ Fact: F and $\overline{F}$ are different nodes in BDD

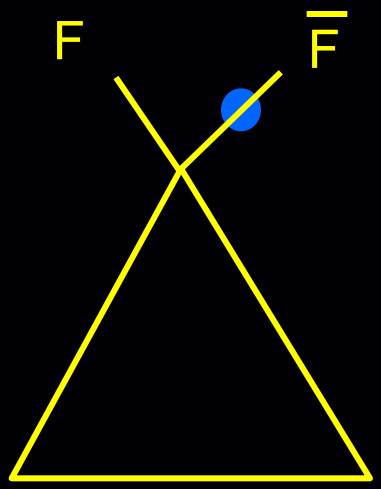
- NOT(F) = ite(F, 0, 1) is not a terminal case
  - Need to go through every path of BDD(F) and at the end, construct BDDs with interchanged terminal nodes (0, 1)

- Any better way?

  ➔ Complement edge

# BDDs with Complement Edges

- A complement edge is a flagged edge which denotes the function is complemented (inverted)
  - Usually use a “bubble” to denote the inverted edge




  - [Note] Only constant “1” node is kept; constant “0” is denoted with complement edge
- Can we still guarantee the canonicity?

# Equivalence with Complement Edges

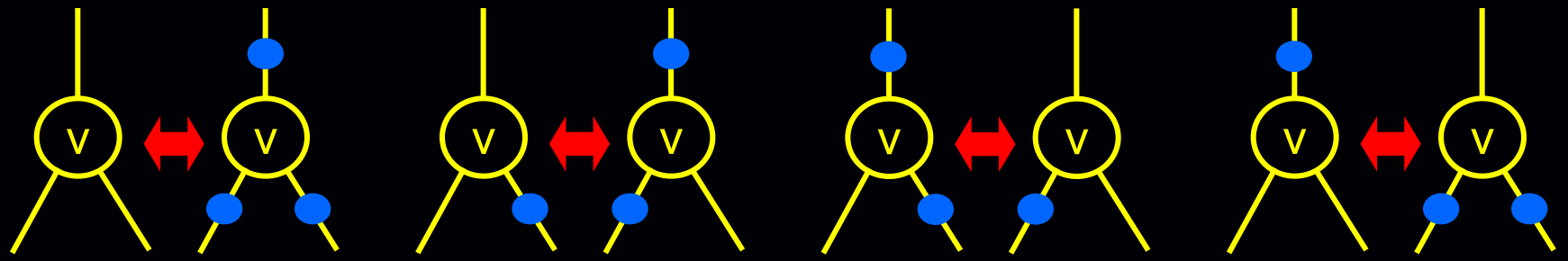
- $(v, F_v, F_{\bar{v}}) \equiv v * F_v + \bar{v} * F_{\bar{v}}$

$$= \overline{\overline{v * F_v + \bar{v} * F_{\bar{v}}}}$$

$$= \overline{(\bar{v} + \bar{F}_v) * (v + \bar{F}_{\bar{v}})}$$

$$= \overline{v * \bar{F}_v + \bar{v} * \bar{F}_{\bar{v}}}$$

$$= \overline{(v, \bar{F}_v, \bar{F}_{\bar{v}})}$$



# Canonicity with Complement Edges

- What's the problem?
  - If we allow (v, $F_v$, $F_{\bar{v}}$) and (v, $\bar{F}_v$, $\bar{F}_{\bar{v}}$) to co-exist in BDDs, they will have different hash keys and thus hash to different nodes (why?)

  ➔ But by using complement edges, they should point to the same node!!

- Solution
  - The "then" child should NOT have bubble

  ➔ If happens, apply the rules in the previous page

# Improving ITE Cache Hit Rate

- Observation
  - There may be ITE(F1, F2, F3) = ITE(G1, G2, G3), where $F_i$ != $G_i$ for some i
    - e.g. ITE(F, G, 0) = $\overline{ITE(\overline{F}, 1, \overline{G})}$ // AND(F, G) = NOR($\overline{F}$, $\overline{G}$)
    - e.g. F + G

    ITE(F, 1, G) = ITE(G, 1, F) = ITE(F, F, G) = ITE(G, G, F)

  ➔ ITE function may be recursively called many times and return the same result
- Objective
  - Rearrange the ITE parameters so that the cache hit rate of the computed table can be higher

    [Keypoint] Think how the computed cache works!!

# Equivalent ITE Operations

- Identical parameters ➔ Constant parameter
  - ite(F, F, G) ➔ ite(F, 1, G) // F + G
    ite(F, G, $\overline{F}$) ➔ ite(F, G, 0) // F * G
    ite(F, $\overline{G}$, F) ➔ ite(F, G, 1)
    ite(F, $\overline{F}$, G) ➔ ite(F, 0, G)
- Symmetrical parameters
  - ite(F, 1, G) = ite(G, 1, F)
    ite(F, G, 0) = ite($\overline{G}$, $\overline{F}$, 0)
    ite(F, G, 1) = ite($\overline{G}$, F, $\overline{1}$)
    ite(F, 0, G) = ite($\overline{G}$, 0, $\overline{F}$)
    ite(F, G, $\overline{G}$) = ite(G, F, $\overline{F}$)
- Complement parameters
  - ite(F, G, H) = ite($\overline{F}$, H, G) = $\overline{\text{ite(F, } \overline{G}, \overline{H})}$ = $\overline{\text{ite}(\overline{F}, \overline{H}, \overline{G})}$

**Which one to choose??**

# Equivalent ITE Operation Rules

1. If contains identical or complement parameters
   ➔ Reduce to constant parameter
2. If contains symmetrical parameters
   ➔ The first parameter is given with the smallest "top" variable;
   ➔ If tied, choose the one with smaller pointer address
3. If contains complement edge parameters
   ➔ The first and second parameters cannot be complement edges

ite(F, F, G) ➔ ite(F, 1, G) // F + G
ite(F, G, $\overline{F}$) ➔ ite(F, G, 0) // F * G
ite(F, $\overline{G}$, F) ➔ ite(F, G, 1)
ite(F, $\overline{F}$, G) ➔ ite(F, 0, G)

ite(F, 1, G) = ite(G, 1, F)
ite(F, G, 0) = ite($\overline{G}$, $\overline{F}$, 0)
ite(F, G, 1) = ite($\overline{G}$, F, $\overline{1}$)
ite(F, 0, G) = ite($\overline{G}$, 0, $\overline{F}$)
ite(F, G, $\overline{G}$) = ite(G, F, $\overline{F}$)

ite(F, G, H) = ite($\overline{F}$, H, G)
$\overline{\text{ite}(F, \overline{G}, \overline{H})}$ = $\overline{\text{ite}(\overline{F}, \overline{H}, \overline{G})}$

# Recursive Algorithm for ITE Operation

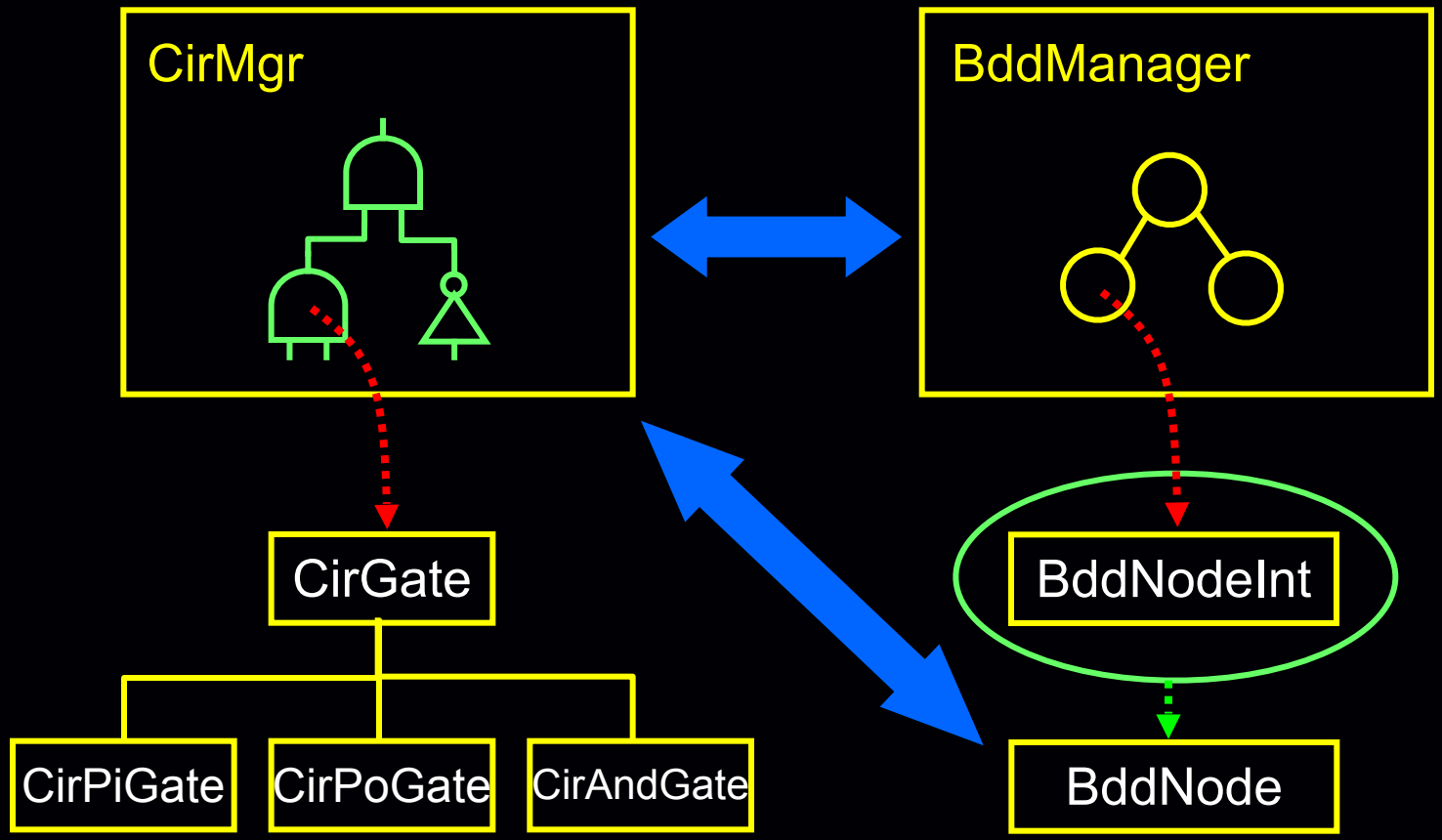
- ite(F, G, H) {
  Standardize parameters (F, G, H);
  if (terminal case)
  return result;
  if (ite(F, G, H) in computed table)
  return result;
  let v be the top variable;
  T = ite($F_v$, $G_v$, $H_v$);
  E = ite($F_{\bar{v}}$, $G_{\bar{v}}$, $H_{\bar{v}}$);
  Process complement edge info for T & E
  if (T == E) return T;
  if ((v, T, E) in the unique table)
  return result;
  let R = BddNode(v, T, E);
  insert R into unique table;
  return R;
  }

03/12

In the coming PA(s), we will use this BDD package for BDD constructions.

Let's take a look at how it is implemented.

# Class Hierarchy: Circuit vs. BDD



# Overview of the BDD Class Hierarchy

- class BddManager
  - BDD implementation management
    - Input variable list, terminal 0/1 nodes
    - Unique table, computed table
  - Interface to construct BDDs
- class BddNodeInt (private class)
  - Nodes (pointers) in the diagram
  - Implementation details
- class BddNode
  - BddNodeInt* + edge info
  - Wrapper class (objects) on top of BddNodeInt
  - Encapsulates implementation details

# Target usage of BddManager and BddNode

```
int
main()
{
   // #input, |hash|, |cache|
   BddManager bm(2, 127, 61);

   BddNode a(bm.getSupport(1));
   BddNode b(bm.getSupport(2));

   BddNode c = a & b;

   cout << c << endl;
}
```

Output

```
[2](+) 0x517db0 (1)
  [1](+) 0x516620 (3)
    [0](+) 0x515010 (8)
    [0](-) 0x515010 (8) (*)
  [0](-) 0x515010 (8) (*)
```

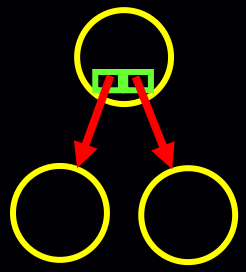
==> Total #BddNodes : 3

➔ Why the reference count for "[0](+) 0x515010 " is '8'?

# BDD Manager

- Initialize unique and computed tables
- Record the terminal (constant 1) node
- Initialize input variables
- Statistics
  - Number of nodes
  - Number of operations
  - Number of dead nodes

# "BddNode c = a & b" ?



- Note: "pointers" are needed to connect and share nodes in a graph.
  - For each node, its left and right children are pointers to other BDD nodes.
  - However, pointers cannot perform operators such as "&".
- Solution: A "wrapper" class to wrap the pointer so that operator overloading can be implemented.

# class BddNode

- Wrapper class so that users don't need to worry about the reference count, pointer address, etc
- 
```
class BddNode {
   BddNodeInt*   _node;
public:
   BddNode(BddNodeInt* n) :_node(n); // do incRefCount
   ~BddNode();                       // do decRefCount
   BddNode& operator = (const BddNode&);
   BddNode operator && (const BddNode&) const;
   BddNode operator || (const BddNode&) const;
   BddNode operator ! (const BddNode&) const;
   BddNode& operator &= (const BddNode&);
   BddNode& operator ||= (const BddNode&);
   bool operator == (const BddNode&) const;
   bool operator != (const BddNode&) const;
   bool operator !() const;
};
```

# class BddNodeInt

- Pointer Nodes for internal BDD graph representation
  - Graphical connection
  - Pointers used in unique table returned results
  - ➔ Basic building blocks for BDDs
  - ➔ Need to be careful about the size of this class
- A classical 12-Byte implementation // for 32-bit platform

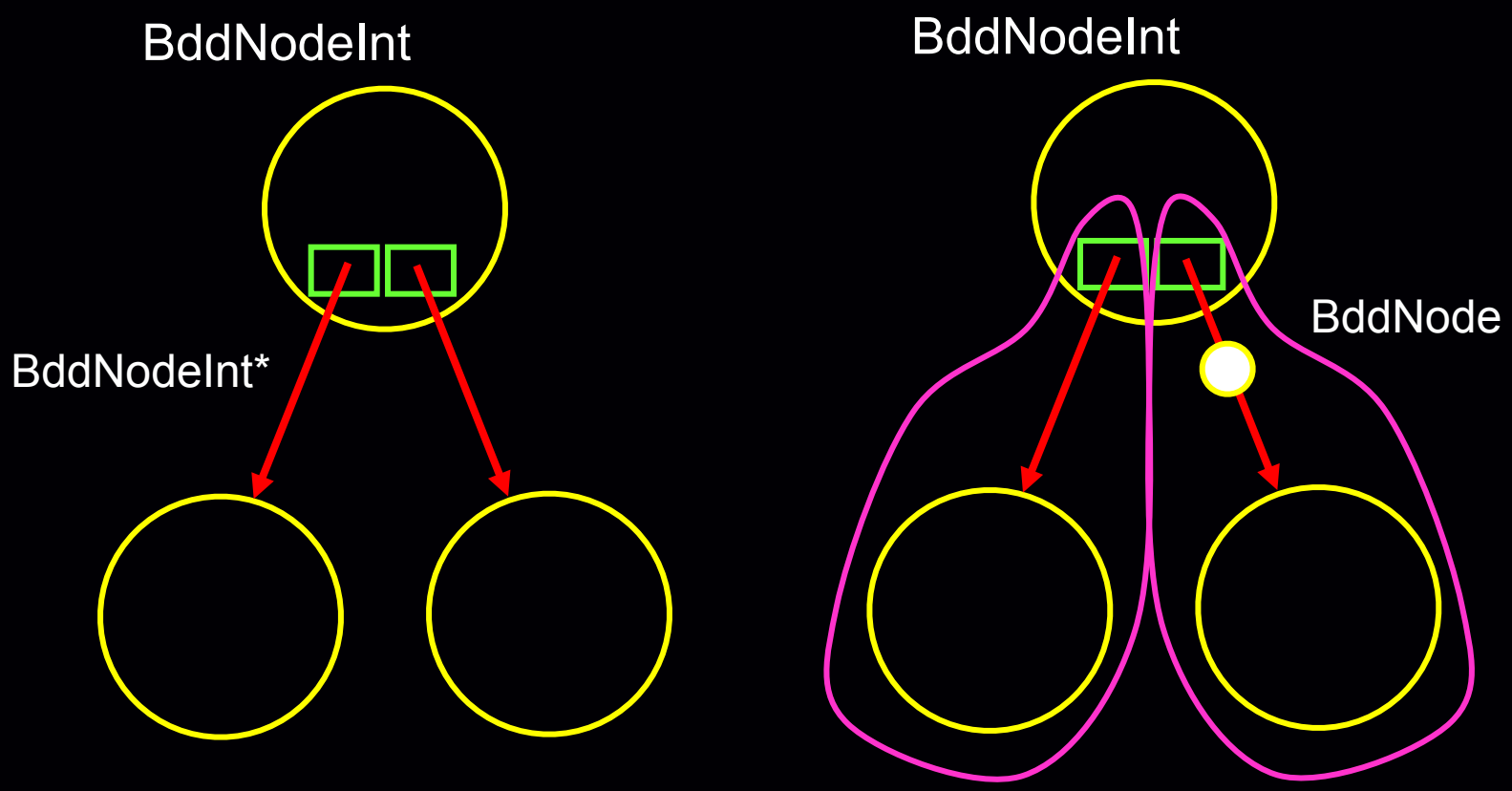
```
class BddNodeInt {
   friend class  BddManager;
   friend class  BddNode;
   unsigned      _level     : 16;  // or 24
   unsigned      _refCount  : 16;  // or 8
   BddNodeInt*   _left;
   BddNodeInt*   _right;
};
```

# How about Complement Edges?

- Option #1
  - Add another field "_flag"  (another 4 Bytes)
- Option #2
  - Reduce the bit width of _level or _refCount
- Options #3 (suggested in DAC90 paper)
  - Use the fact the pointer address is multiplier of 8 (64-bit machine)
  - BddNodeInt* p = ...;

```
size_t pv = (size_t)p;
pv = pv + COMPLEMENT_FLAG;
```

# BddNodeInt vs. BddNode



# Class BddNode Revised

- Original implementation in slide p58

```
class BddNode {
private:
    BddNodeInt*   _node;
}
```

  - Use the fact the pointer address is multiplier of 8 (for 64-bit machine)

# Class BddNode Revised

```
class BddNode {
public:
   // BddNode constructor and destructor are the
   // only place to increase or decrease _refCount
   BddNode(...);
   ~BddNode();
private:
   size_t        _nodeV;

   BddNodeInt* getBddNodeInt() const {
      return (BddNodeInt*)(_nodeV & PTR_MASK); }
   bool isNegEdge() const {
      return (_nodeV & BDD_NEG_EDGE); }
   bool isPosEdge() const { return !isNegEdge(); }
}
```

# Class BddNodeInt Revised

```
class BddNodeInt {
    friend class  BddManager;
    friend class  BddNode;
    BddNodeInt*     _left;
    BddNodeInt*     _right;
    unsigned short  _level;
    unsigned short  _refCount;
};
```

# Class BddNodeInt Revised

```
class BddNodeInt {
    friend class  BddManager;
    friend class  BddNode;
    BddNode         _left;
    BddNode         _right;
    unsigned short  _level;
    unsigned short  _refCount;
};
```

# Revisit the test program...

```
int
main()
{
    // #input, |hash|, |cache|
    BddManager bm(2, 127, 61);

    BddNode a(bm.getSupport(1));
    BddNode b(bm.getSupport(2));

    BddNode c = a & b;

    cout << c << endl;
}
```

Output

```
[2](+) 0x517db0 (1)
  [1](+) 0x516620 (3)
    [0](+) 0x515010 (8)
    [0](-) 0x515010 (8) (*)
  [0](-) 0x515010 (8) (*)
```

==> Total #BddNodes : 3

➔ Why the reference count for "[0](+) 0x515010 " is '8'?

## What to expect in the next lecture note...

1. Dynamic BDD variable reordering
2. Reducing the BDD size by using cut
   - Local vs. global BDDs
   - Compose operation
3. Partial BDDs
4. Other types of decision diagrams
   - ZDD
   - FDD
   - MTBDD(ADD)
   - BMD