Topic 4

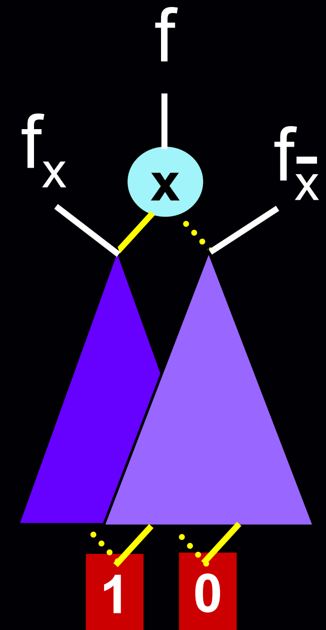
Advanced BDD Techniques

系統晶片驗證
SoC Verification

2025.03.19

Quick Recap: Binary Decision Diagram

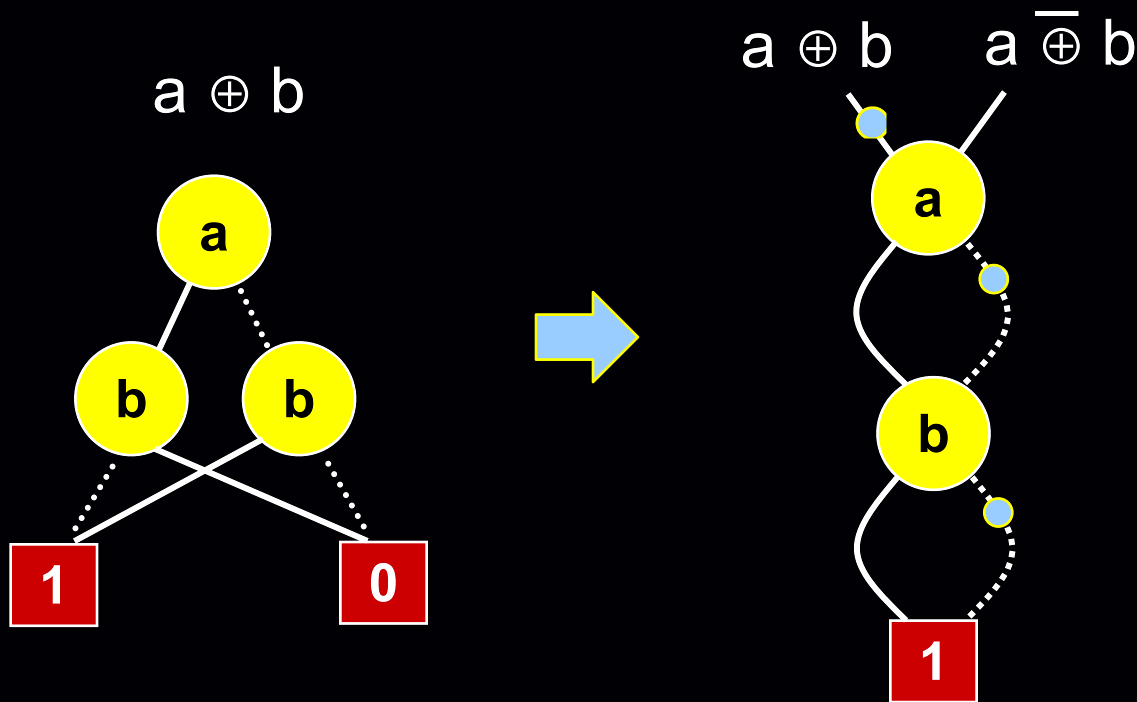
- BDD as Shannon expansion of f
  - $f = x * f_x + \overline{x} * f_{\overline{x}}$
  - $f * g = x * (f_x * g_x) + \overline{x} * (f_{\overline{x}} * g_{\overline{x}})$
  - $f + g = x * (f_x + g_x) + \overline{x} * (f_{\overline{x}} + g_{\overline{x}})$



- Actual implementation:
  Convert operations to ITE
  - $ITE(F, G, H) = F * G + \overline{F} * H$
    $= ITE(v1, ITE(F_{v1}, G_{v1}, H_{v1}), ITE(F_{\overline{v1}}, G_{\overline{v1}}, H_{\overline{v1}}))$

Quick Recap: Binary Decision Diagram

- Use complement edge to denote the negation of BDD:



Quick Recap: ITE Algorithm

- ```
  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_v', G_v', H_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;
  }
  ```

  (where $T = ite(F_v, G_v, H_v)$ and $E = ite(F_{\bar{v}}, G_{\bar{v}}, H_{\bar{v}})$)
  ```

Limitation of BDD

- Sounds good. Looks like if we can build the BDD for the proposition p, we can easily prove or disprove it!!
  - Sounds too good to be true?
  - BDD actually find ALL the solutions at once
- Note: the worst case for BDD is still $O(2^n)$
- Experience shows that BDD is often good for circuit with input size roughly < 200…

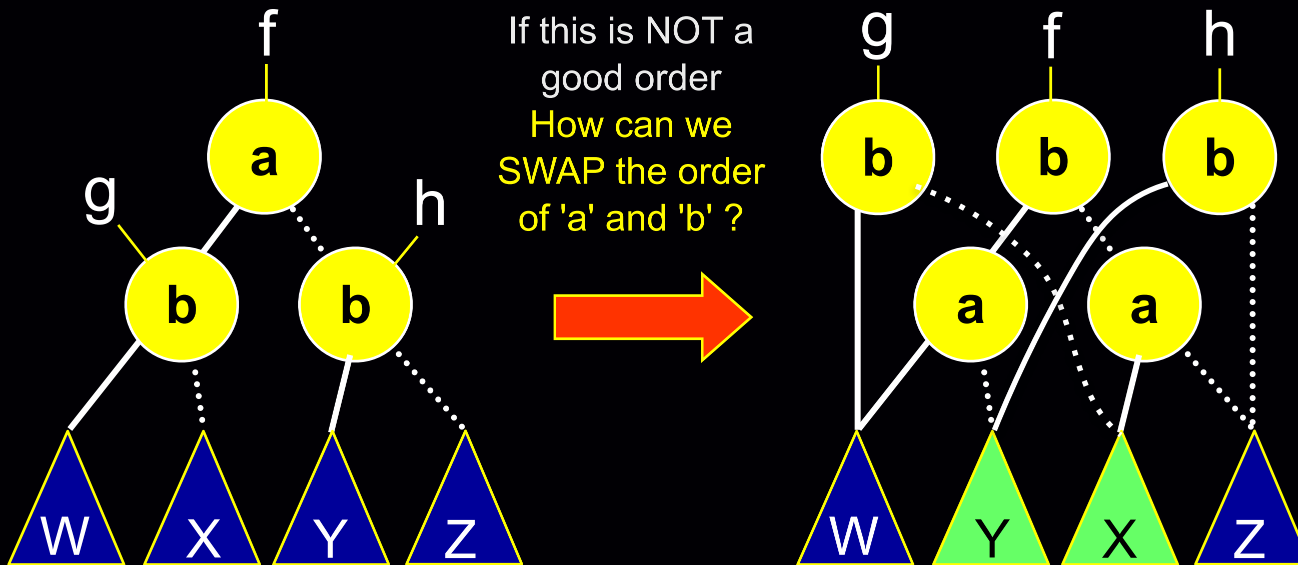
  → Need more heuristics to push the limit!!

Advanced BDD Techniques for Combinational Proof

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

Swapping Adjacent Variables in BDD

- Fujita EDAC 91; Ishiura ICCAD 91
  - Swap the nodes in two different levels



- Need to efficiently identify the nodes in the same level (how?)
  - Unique table = array of hash tables // index by level

=> Need to make sure function references to f, g, h remain unchanged (Where is the save?? Where are g and h?)

Dynamic BDD Variable Reordering

// Remember: different variable ordering leads to different BDD size

→ Try to find a good order dynamically

1. Choose an initial BDD variable ordering
2. Perform BDD construction until memory usage exceeds limit
3. Reorder BDD variables (e.g. sifting, window permutation)
4. Perform garbage collection to free the unreferenced BDD nodes
5. Continue the BDD construction

Sifting Algorithm (Rudell, ICCAD 1993)

- // Given a BDD with n variables (v1... vn)

```
for (i = 1 ~ n) {
    perform interchanges to bring vi
        to the top
    perform interchanges to sift vi
        to the bottom
    keeping track of position resulting
        in smallest BDD
    perform interchanges to move vi
        back to best position
}
```

Dynamic BDD Variable Reordering

// Remember: different variable ordering leads to different BDD size

➔ Try to find a good order dynamically

1. Choose an initial BDD variable ordering
2. Perform BDD construction until memory usage exceeds limit
3. Reorder BDD variables (e.g. sifting, window permutation)
4. Perform garbage collection to free the unreferenced BDD nodes
5. Continue the BDD construction

BDD Node Reference Count

- In building BDDs for a formula, it's very likely that some intermediate nodes are no longer used and thus can be freed to save memory
  - ➔ Keep a reference count in each BDD node
  - Increase "1" if referred by some formula (e.g. BddNode), or pointed by nodes in the higher level
    - Reference from unique or computed tables usually does not count (why??)
  - If (reference count == 0) ➔ dead node, can be freed
    - However, need to make sure no reference in unique or computed tables
    - If a node become a dead node ➔ the reference counts of its children are also decremented by 1

  [Heuristic] If (reference count == MAX_REF_COUNT)

  ➔ this node is denoted as saturating and will never be freed

Recall: Class BddNodeInt

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

Garbage Collection

- Reclaim
  - If a lookup (e.g. by some formula) in a computed table returns a dead node
    - → This dead node will become alive
    - → Increase its reference count and recursively reclaim its children
- #(dead nodes) is kept to determine whether garbage collection is needed
- (Brace DAC90)

1. If (load factor of the unique table > 4) &&
   ((#dead nodes) > 10%)
   - → Trigger garbage collection
   - → All the dead nodes in the tables are removed and freed
2. If the condition in 1 still holds, resize the tables and rehash

Advanced BDD Techniques for Combinational Proof

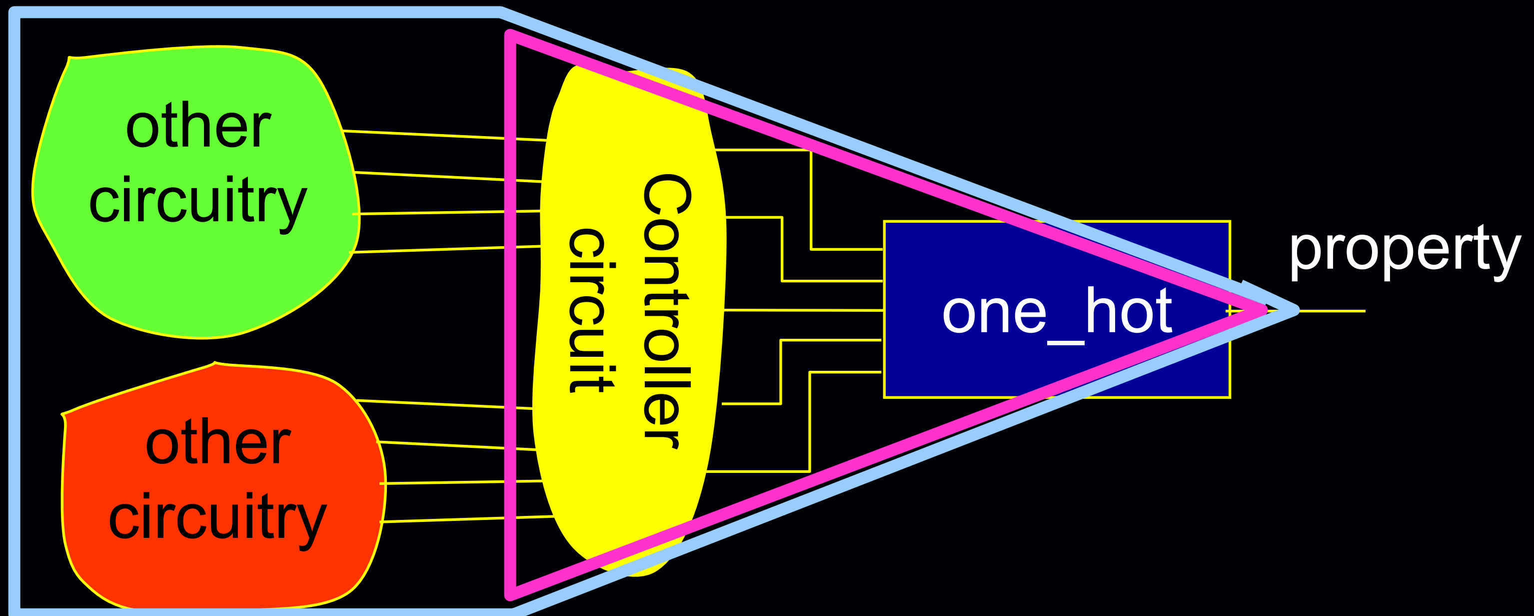
1. Dynamic BDD variable reordering
2. Reducing the BDD size by using cut
   - Local vs. global BDDs
   - Compose operation
3. Partial BDDs
4. Other types of decision diagrams
   - ZDD
   - FDD
   - MTBDD(ADD)
   - BMD

Locality of an Assertion Property

- [Observation]

  Designers usually assure the assertions hold within the local module

  → Proving local instead of global cone



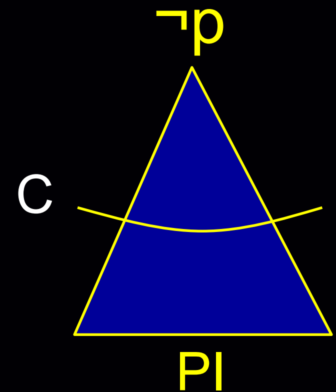
Local vs. Global BDDs

- Since the size of BDD may grow exponentially as the size of function increases, building BDD for a smaller (local) function ***may be*** much faster than for a bigger (global) function.

- Let BDD(¬p) be the global BDD for ¬p

  BDD_C(¬p) be the local BDD for ¬p

  with respect to a cut C

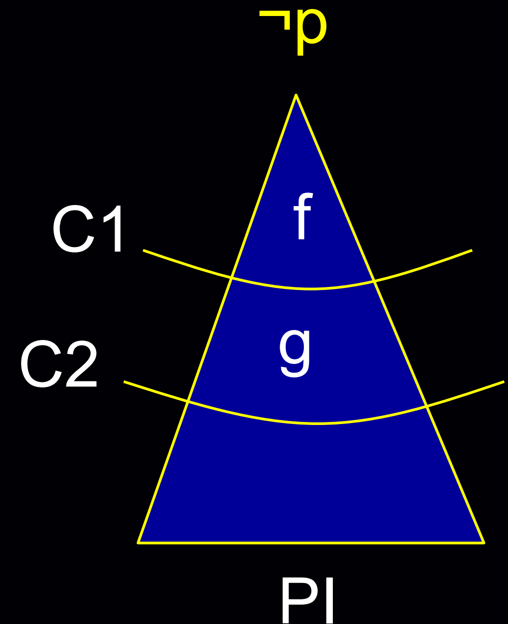


- To prove p :
  - If BDD(¬p) = constant 0, p is always true
  - If BDD(¬p) != constant 0, each cube to 1 is a counter-example for p

Building BDDs using cut

1) Find a cut C = C1
2) Treat C as BDD supports and build BDD for ¬p
3) If $BDD_C(\neg p) = 0$, then $BDD(\neg p)$ is also = 0

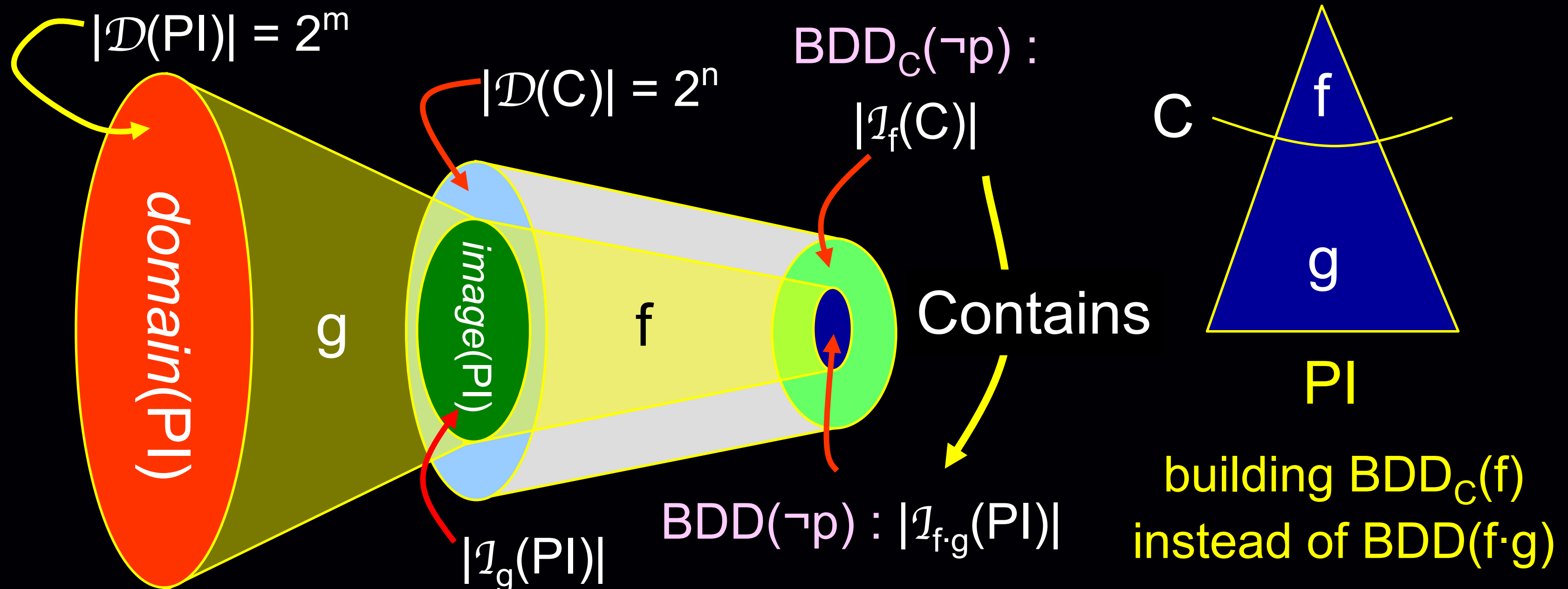
   ➔ p is proven true (why?)
4) If $BDD_C(\neg p)$!= 0, find another cut C2 that is closer to PI.
   Let C = C2. Repeat (2) (why?)



Local cut: a simple abstraction

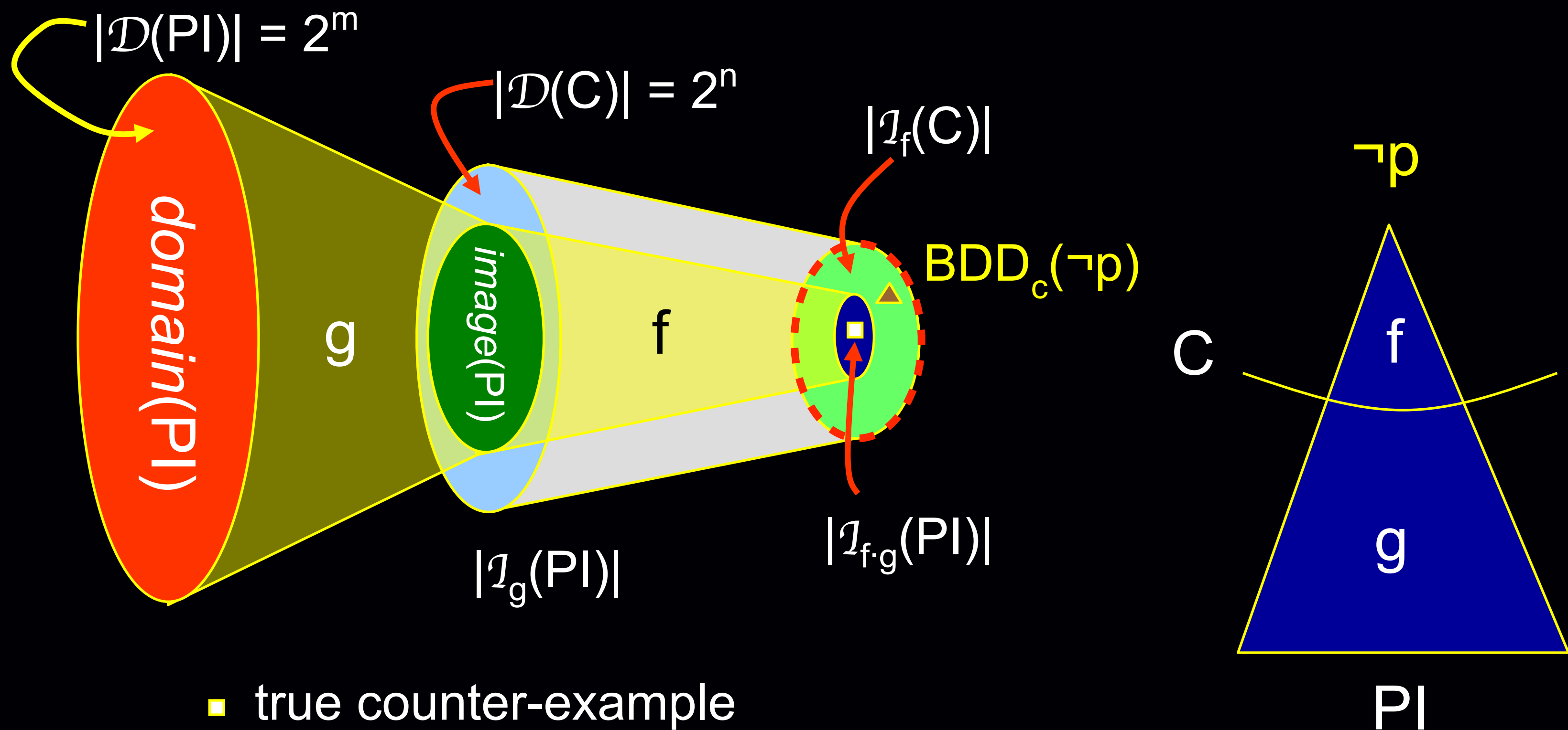
If $BDD_C(\neg p) = 0$, then $BDD(\neg p)$ is also $= 0$

- Let the number of inputs: $|PI| = m$,
  and the number of gates on the cut: $|C| = n$.
  - Let $\mathcal{D}(PI)$ be the domain for the variables of PI, and $\mathcal{I}_g(PI)$ be the image of PI under g



Potential Spurious Counter-Example

When $BDD_c(\neg p)$!= 0.... (i.e. $\mathcal{I}_f(C)$!= ∅)

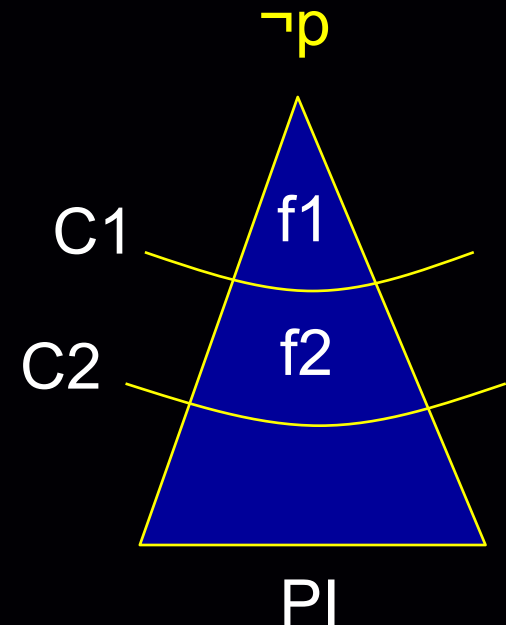


Resolve Non-zero Local BDD

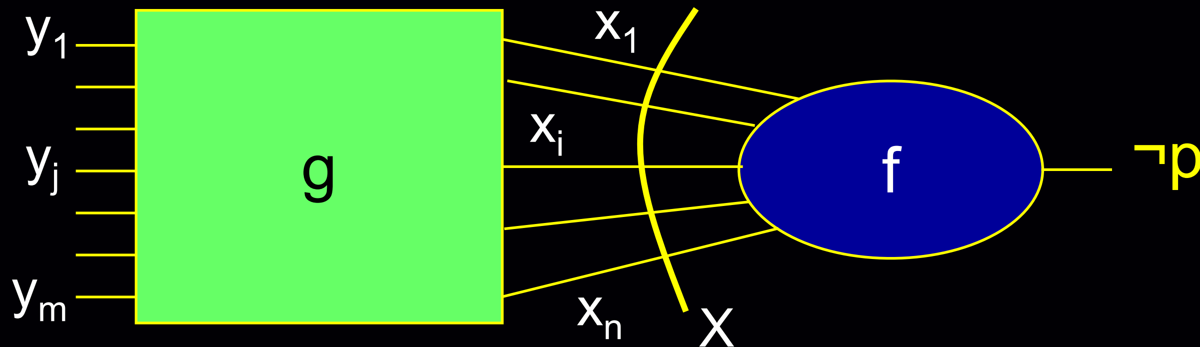
When $BDD_{c1}(\neg p)$!= 0,

→ Find another cut C2 that is closer to PI; construct $BDD_{C2}(\neg p)$

1. Reconstruct
   - Rebuild $BDD_{C2}(\neg p)$
2. Compose
   - Building BDD for $f1 |_{C1 = f2}$



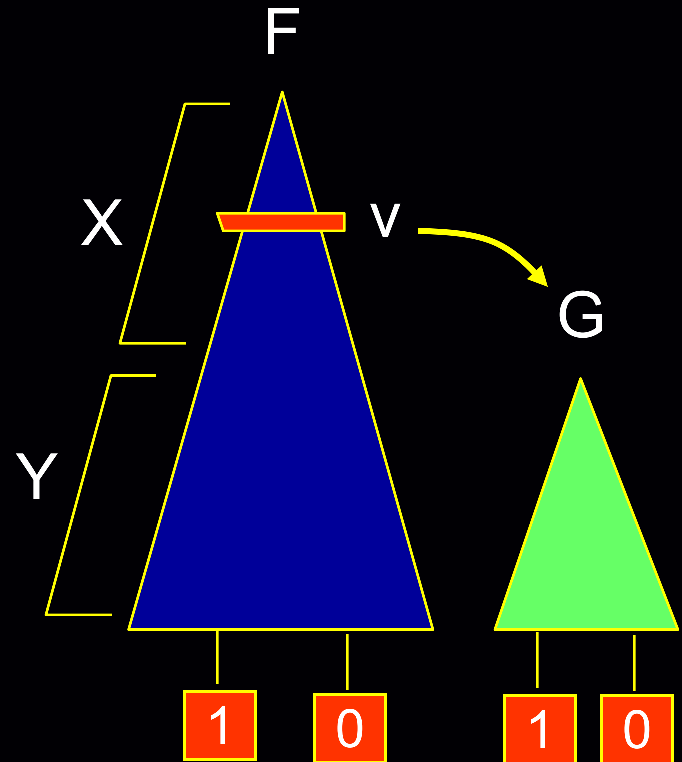
Compose Operation in BDD




- ```
  // Let F = BDD_X(f) on X
  //     G: {G_1,..., G_n} be the BDDs for {x_1,...,x_n}
  //
  ComposeBDD(F, X, G) {
      Let R = F;
      for (variable x_i in X) {
          R = Compose(R, x_i, G_i); // remove x_i from R
      }
      return R; // support is now only on Y
  }
  ```
  ```

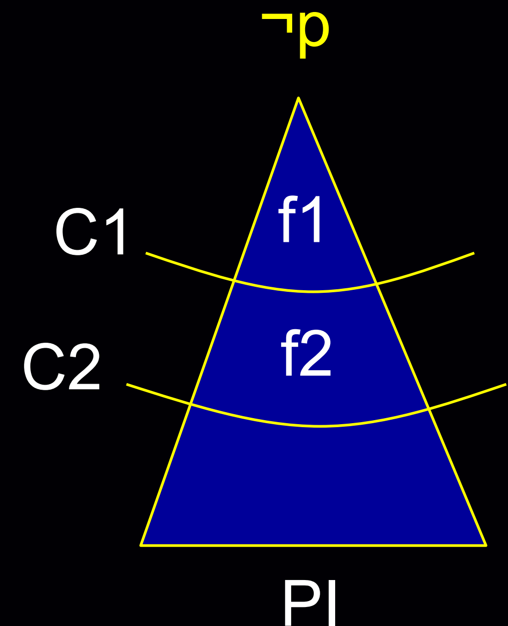
Compose Operation in BDD

```
// Let F1, F0 be the
// positive and negative
// cofactors of F
Compose(F, v, G){
    let u = top_var(F);
    if (u < v) // F is lower
    // F is independent of v
        return F;
    else if (v == u)
    // why not ITE(v,F1,F0)?
        return ITE(G,F1,F0);
    // else F higer than v
    T = Compose(F1, v, G);
    E = Compose(F0, v, G);
    return ITE(u, T, E);
}
```



Practical Issue about Compose Operation

- Bottom-line
  - Compose operation can be very expensive
    - Compose(F, v, G): $O(|F|^2 \times |G|)$
  - Very sensitive to the cuts
- In practice
  - Making BDD(f2) small (don't push too far)
  - Or just rebuild BDD from C2 (don't use "compose")



Advanced BDD Techniques for Combinational Proof

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

Partial BDD

1. Case splitting
   - Divide and conquer
2. Witness generation
   - Depth-first-search approach
3. Partial BDD construction

e.g. “Partial Binary Decision Diagrams”, Whitney J. Townsend and Mitchell A. Thornton, 34th IEEE Southeastern Symposium on System Theory, pp. 422-425, Huntsville, AL, March 18-19, 2002

Case Splitting

1. Select some "important" variables (usually controlling PIs)
2. Enumerate the value combinations of these variables
3. For each value assignment, perform constant propagation
4. Build BDDs
5. Repeat 2-4 for all combinations

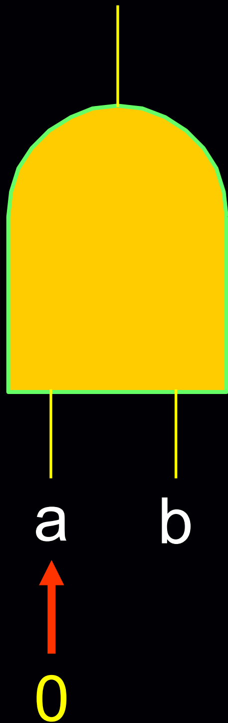
Witness Generation

- To prove AG(p), we just need to generate a witness for EF(¬p) ← counter example
- However, when we build the BDD for EF(¬p), all the counter examples are generated

  → Could be an over-kill

- How to use BDD to generate partial counter-examples?

A Rough Idea....

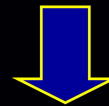


Witness (a, 0) first

→ Just compute BDD(¬a)

→ Ignore b; as don't-care...

Any problem??



What about a's fanins?

(a recursive procedure?)

How about the complexity?

(Become a SAT problem?)

No, you should not ignore BDD(b)...

Witness-BDD Generation Algorithm

- Terminology
  - Output controlling value (of a gate)
    ➔ Fanin's value is determined
    - e.g. value '1' for AND gate output
  - Output non-controlling value (of a gate)
    - e.g. value '0' for AND gate output
    - e.g. value '0' or '1' for XOR gate output
- Basic BDD operations
  1. ConstructBDD(gate, value, isCare)
     - If output has controlling value, recursively construct BDDs for all its fanins
  2. WitnessBDD(gate, value)
     - If output has non-controlling value, try to construct "minimal" BDDs to witness the output value

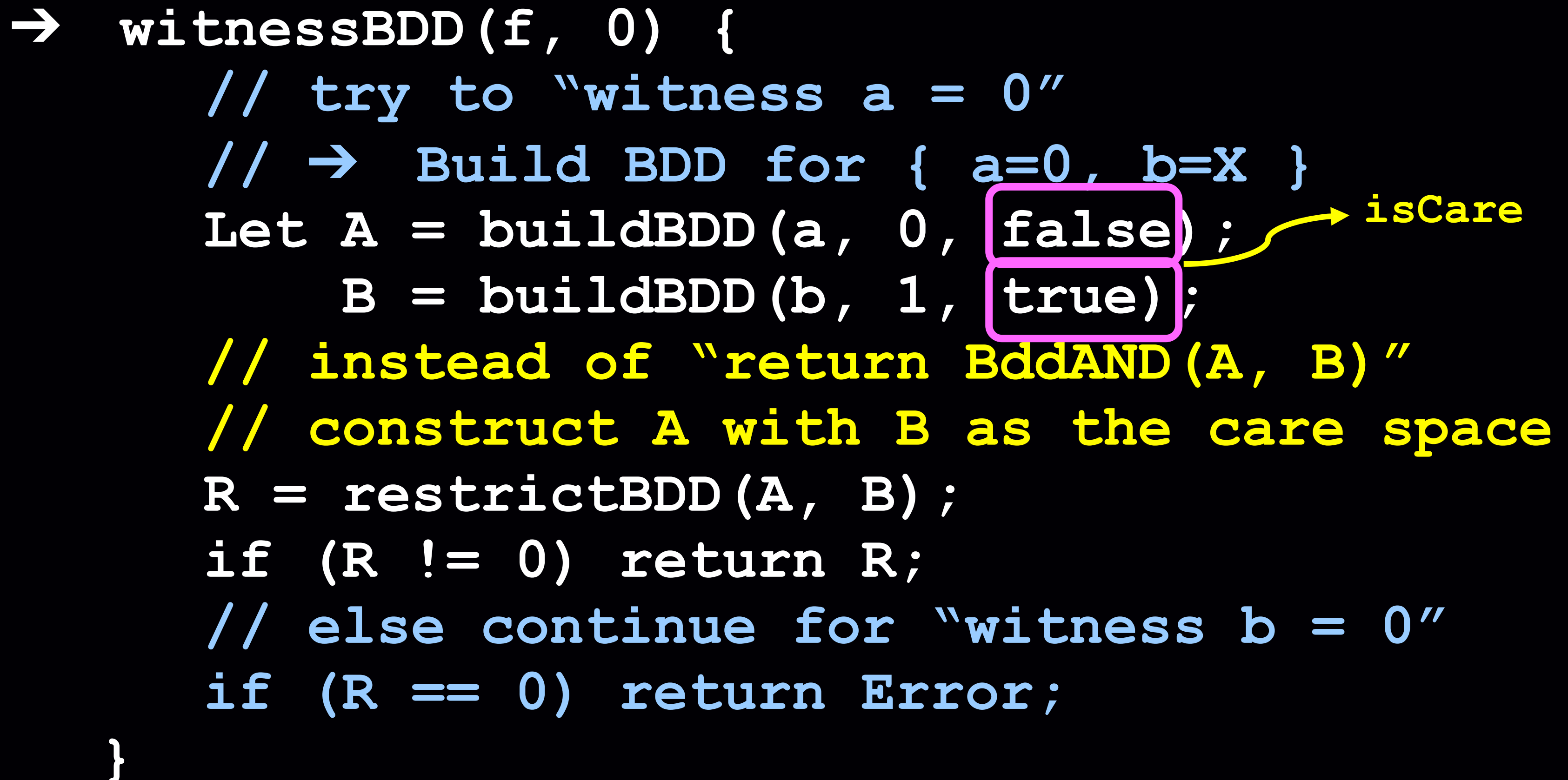
Witness-BDD Generation Algorithm

```
buildBDD(gate, value, isCare) {
    hashedBDD = bddMap(gate);
    if (hashedBDD != NULL)   // already built
        return hashedBDD;
    if (isCare || value is output-controlling) {
        for_each_fanin(fanin, faninV)
            buildBDD(fanin, faninV, isCare);
        constructBDD(type, gate, faninBDDs, isCare);
    }
    else { // non-controlling
        return witnessBDD(gate, value);
    }
}
// Note: If witnessBDD fails to witness a non-zero
   BDD, then the process fails.
```

explained later

WitnessBDD(gate, value) Operation

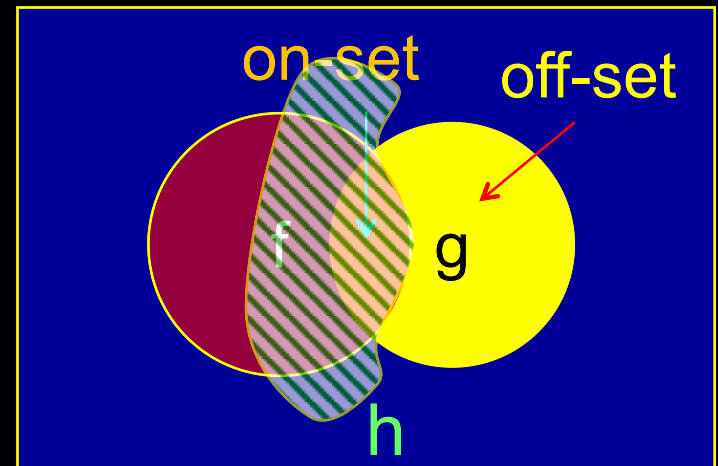
- Without loss of generosity, let's consider a 2-input AND gate "f = AND (a, b)" ---



Restrict Operation in BDD

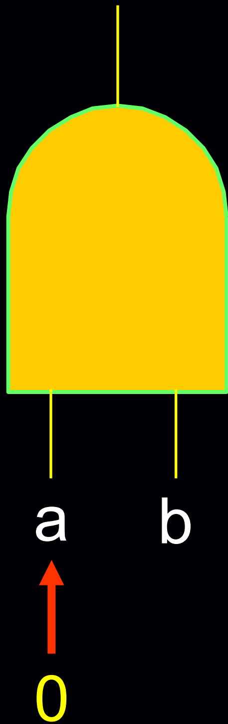
[Coudert & Madre, ICCAD90]

- "h = restrict(f, g)" is a function equivalent to "f if g"
  // if !g, f don't care...; i.e. g is the care space
  - The result is guaranteed to be simpler than 'f' (i.e. less BDD nodes)
- Conditions
  - if $(f \wedge g)$, then h = 1 (on)
  - if $(\neg f \wedge g)$, then h = 0 (off)
  - $(f \wedge g) < h < (f \vee \neg g)$

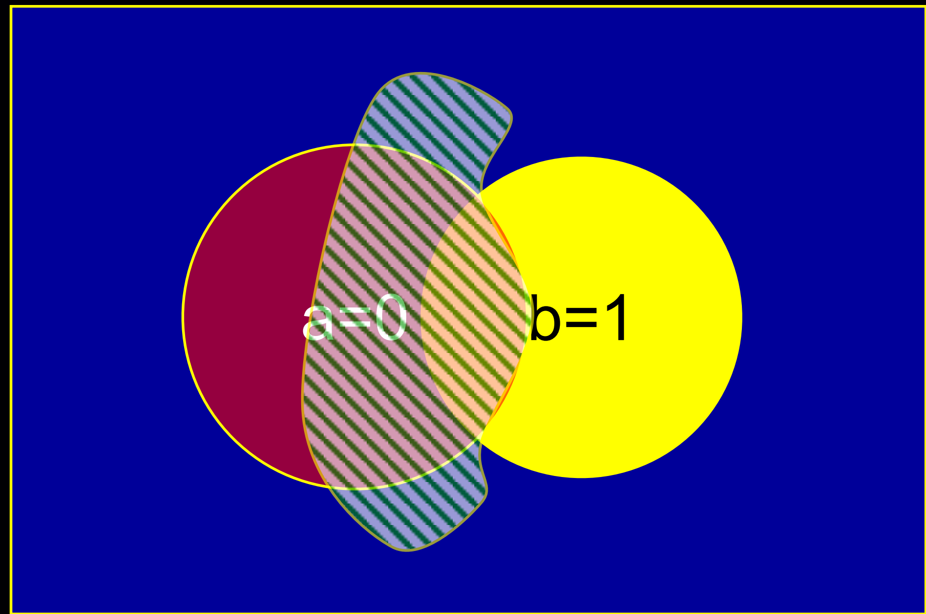


Using Restrict Operator

h = witness (p, 0)



h = restrict (¬a, b)



$(f \wedge g) \rightarrow h \rightarrow (f \vee \neg g)$

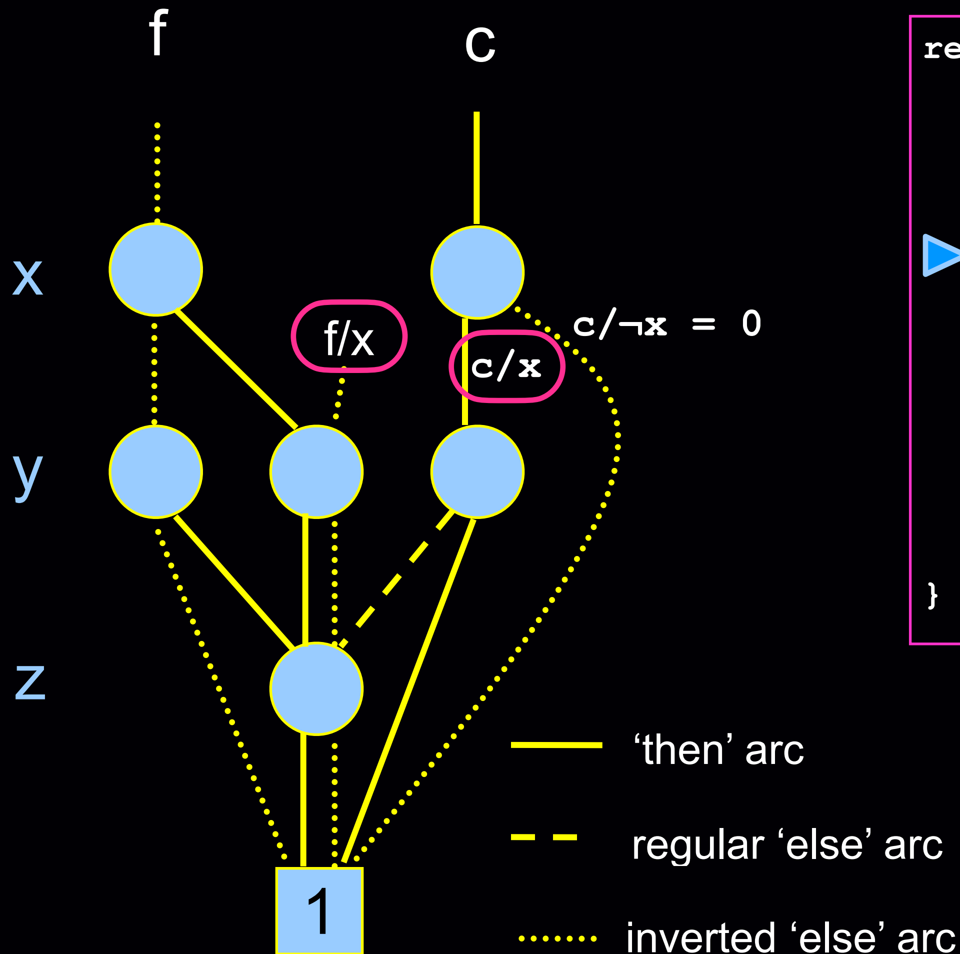
BDD Restrict Algorithm in Details

- Let c.root be the root node of BDD function c; (c/¬a, c/a) be the Shannon's expansion of the function c with respect to a

- ```
  restrict (f, c) {
      if (c = 0), return error;
      if (c = 1), return f;
      if (f = 0 or f = 1), return f;
      let a = c.root;
      if (c/¬a = 0), return restrict(f/a, c/a);
      if (c/a = 0), return restrict(f/¬a, c/¬a);
      if (f/¬a = f/a), return restrict(f, c/¬a ∨ c/a);
      return (¬a ∧ restrict(f/¬a, c/¬a)) ∨
             (a ∧ restrict(f/a, c/a));
  }
  ```
  ```

An example for "Restrict"

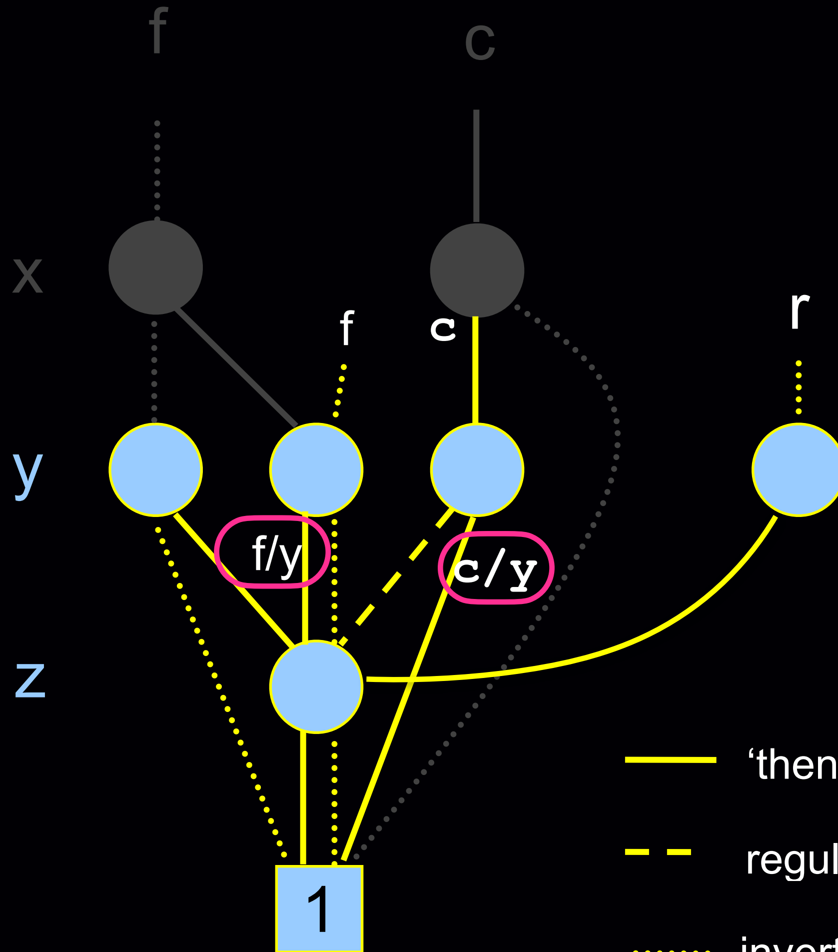
r = restrict (f, c) = f ⇓ c




```
restrict (f, c) {
   if (c = 0), return error;
   if (c = 1), return f;
   if (f = 0 or f = 1), return f;
   let x = c.root;
   if (c/¬x = 0)
      return restrict(f/x, c/x);
   if (c/x = 0)
      return restrict(f/¬x, c/¬x);
   if (f/¬x = f/x)
      return restrict(f, c/¬x ∨ c/x);
   return (¬x ∧ restrict(f/¬x, c/¬x)) ∨
          (x ∧ restrict(f/x, c/x));
}
```

An example for “Restrict”

r = restrict (f, c) = f ⇓ c

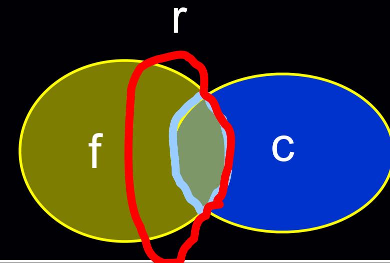



```
restrict (f, c) {
    if (c = 0), return error;
    if (c = 1), return f;
    if (f = 0 or f = 1), return f;
    let y = c.root;
    if (c/¬y = 0)
        return restrict(f/y, c/y);
    if (c/y = 0)
        return restrict(f/¬y, c/¬y);
    if (f/¬y = f/y)
        return restrict(f, c/¬y ∨ c/y);
    return (¬y ∧ restrict(f/¬y, c/¬y)) ∨
           (y ∧ restrict(f/y, c/y));
}
```

‘then’ arc

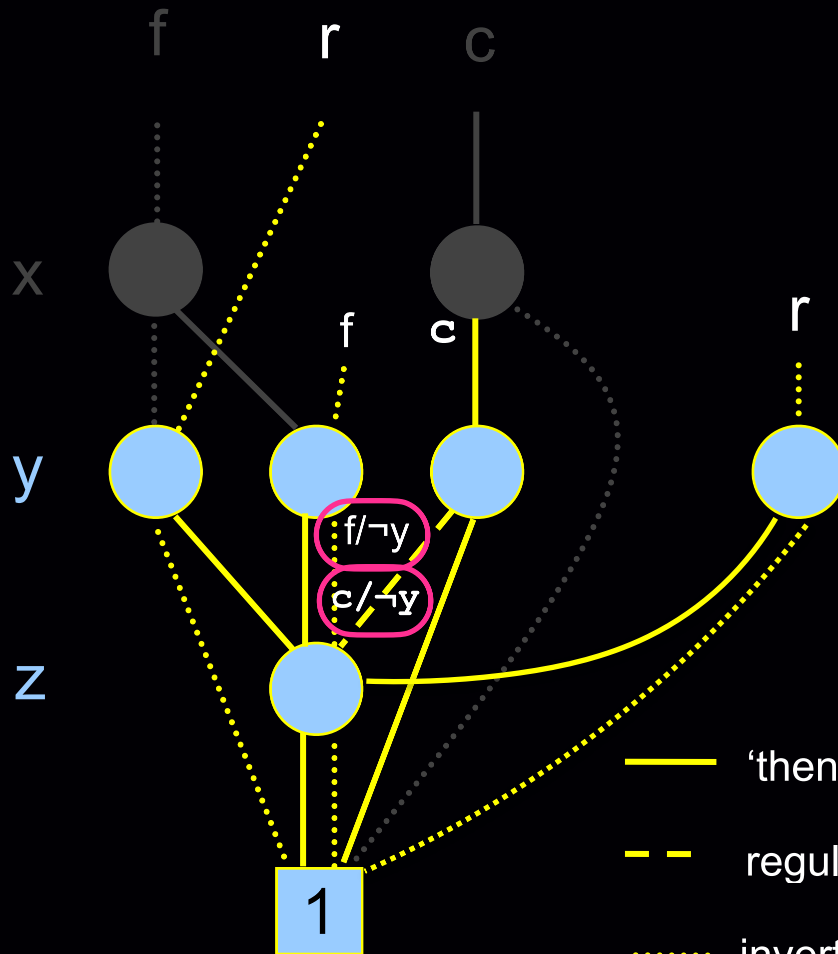
regular ‘else’ arc

inverted ‘else’ arc



An example for “Restrict”

r = restrict (f, c) = f ⇓ c

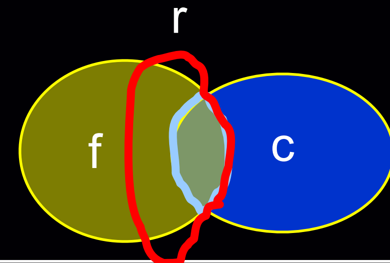



```
restrict (f, c) {
   if (c = 0), return error;
   if (c = 1), return f;
   if (f = 0 or f = 1), return f;
   let y = c.root;
   if (c/¬y = 0)
      return restrict(f/y, c/y);
   if (c/y = 0)
      return restrict(f/¬y, c/¬y);
   if (f/¬y = f/y)
      return restrict(f, c/¬y ∨ c/y);
   return (¬y ∧ restrict(f/¬y, c/¬y)) ∨
          (y ∧ restrict(f/y, c/y));
}
```

‘then’ arc

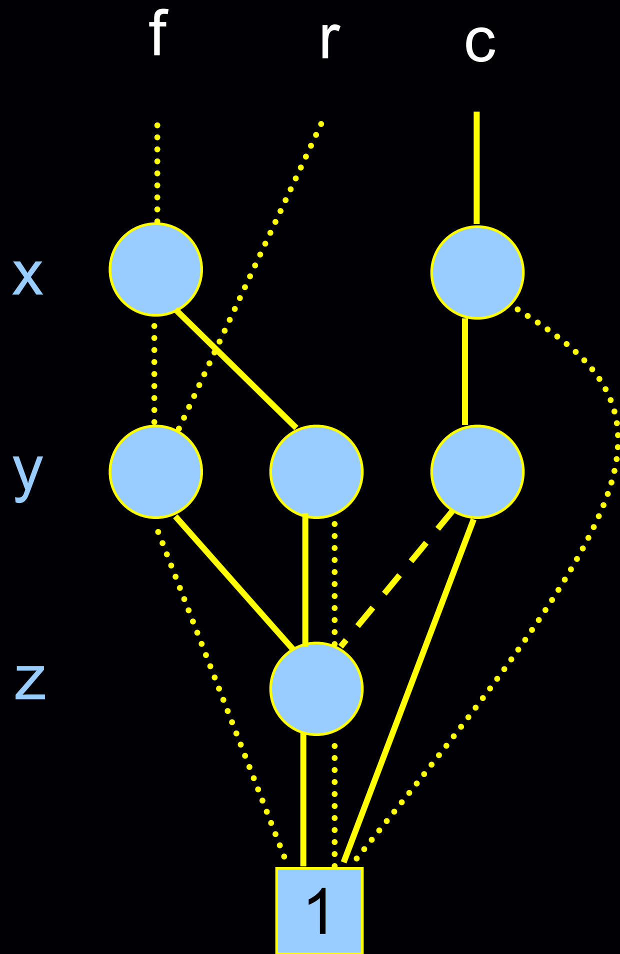
regular ‘else’ arc

inverted ‘else’ arc



An example for "Restrict"

r = restrict (f, c) = f ⇓ c

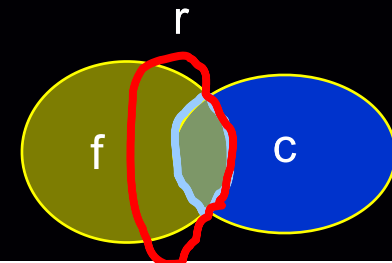



```
restrict (f, c) {
   if (c = 0), return error;
   if (c = 1), return f;
   if (f = 0 or f = 1), return f;
   let x = c.root;
   if (c/¬x = 0)
      return restrict(f/x, c/x);
   if (c/x = 0)
      return restrict(f/¬x, c/¬x);
   if (f/¬x = f/x)
      return restrict(f, c/¬x ∨ c/x);
   return (¬x ∧ restrict(f/¬x, c/¬x)) ∨
          (X ∧ restrict(f/x, c/x));
}
```

'then' arc

regular 'else' arc

inverted 'else' arc



buildBDD(gate, value, isCare) // When isCare = true

- For "buildBDD(B)" in "R = restrictBDD(A, B);"
  - We cannot call "witnessBDD" under the recursion of buildBdd(B) because if we do, the returned BDD(B) can be smaller and thus misses some "care-space" for "restrictBDD(A, B)"
  - Because if we do so, we may "witness" a cube that is in (!A && B) // i.e. off-set
- Set "isCare = true" to avoid going into "witnessBDD()"

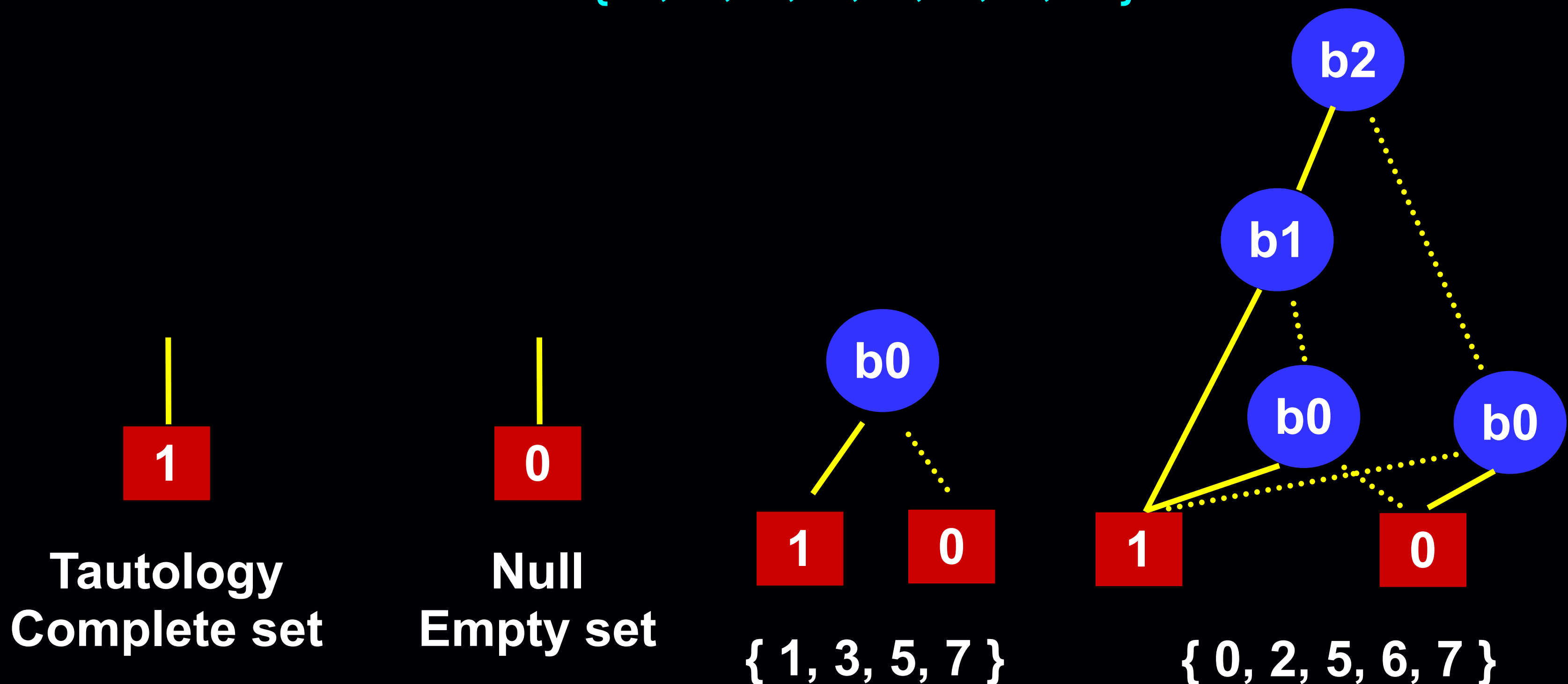
```
buildBDD(f, v, isCare=false){
  // v is output non-controlling
  witnessBDD(g, u) {
    A = buildBDD(a, u1, false)
    B = buildBDD(b, u2, true){
      only constructBDD called
    }
    restrictBDD(A, B)
  }
}
```

Advanced BDD Techniques for Combinational Proof

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

BDD to Represent a Set

- Other than Boolean function, BDD can also represent sets over Boolean variables
  - e.g. Use 3-variable BDDs to represent any subset of the numbers in { 0, 1, 2, 3, 4, 5, 6, 7 }



Zero-Suppressed BDD (ZDD)

- However, BDD may not be good for sparse set representation, set cover/union operation, etc
  - ➔ ZDD can do better

- Proposed: S. Minato, DAC 93
- A good tutorial:
  - "An Introduction to Zero-Suppressed Binary Decision Diagrams", Alan Mishchenko, 2001

BDD vs. ZDD

- BDD: function representation
- ZDD: set cover representation

➔ Conversion between BDD and ZDD is easy

- Remember, BDD has two reduction rules

1. Uniqueness: hash(level, left, right)
2. Non-redundant test: eliminate the node whose positive and negative children pointing to the same node

➔ ZDD does NOT have (2)

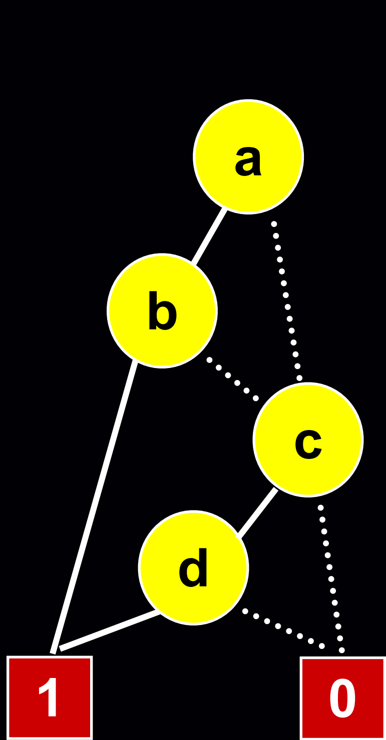
- Instead, ZDD's non-redundancy rule
  - Remove the node whose "positive" edge pointing to a constant '0' node

ZDD Examples

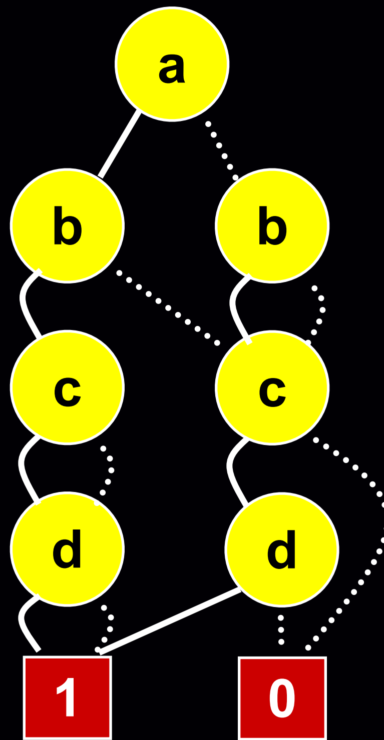
- Function: ab + cd

- set of subsets: { {a,b}, {a,c}, {c} }

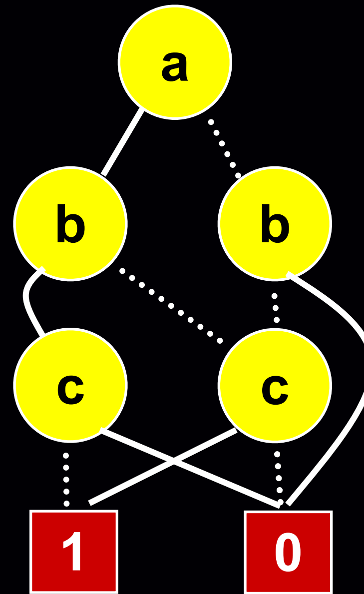
→ **characteristic function: { (abc) | (110), (101), (001) }**



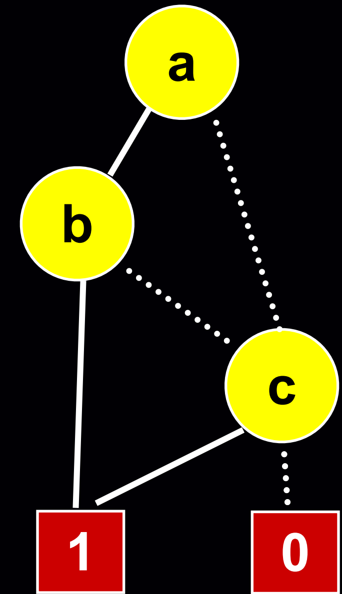
BDD



ZDD



BDD



ZDD

Intuitions on the ZDD non-redundancy rule

- If we treat the "cubes" in BDDs as strings of 1/0's, what does ZDD "suppress" in its representation?
  - e.g. in the previous example, characteristic function:

    { (abc) | (110), (101), (001) }
- The suppressed ZDD nodes are ---

1. Ending 0's
2. Consecutive 0's, maybe except for the leading 0
   - e.g. 100010010 → 1(0)--1(0)-1-

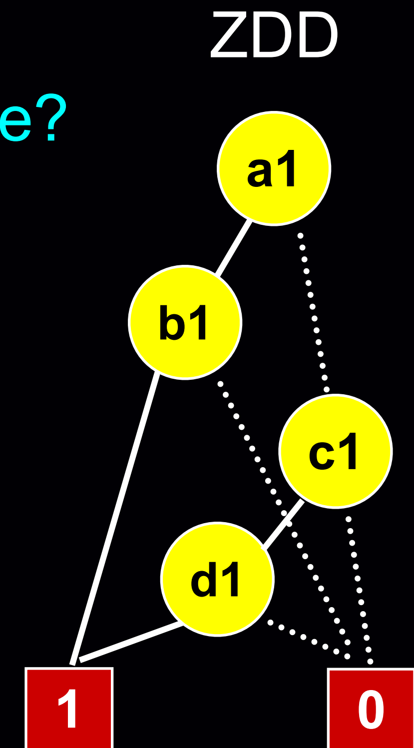
Complexity in Representing Set of Subsets

- ZDD upper bound
  - Total number of elements appearing in all subsets of a set
- BDD upper bound
  - The number of subsets multiplied by the number of all elements that can appear in them

ZDD to Represent Cube Covering

- Goal: use DDs to record the cubes and perform operations (union/intersect, etc) on them
  - E.g. F = ab + cd has 2 cubes
  - How does BDD represent them?
  - Are they same cubes when we retrieve?
    - No, need extra vars to distinguish 0/1/X
- Approach:
  - Each input is split into 2 variables
  - ➔ positive and negative literals
  - ➔ Char. function for cover { ab, cd }

$$= a1\overline{a0}b1\overline{b0}\,\overline{c1}\,\overline{c0}\,\overline{d1}\,\overline{d0}$$
$$+ \overline{a1}\,\overline{a0}\,\overline{b1}\,\overline{b0}c1\overline{c0}d1\overline{d0}$$
$$= 10100000 + 00001010$$



Advanced BDD Techniques for Combinational Proof

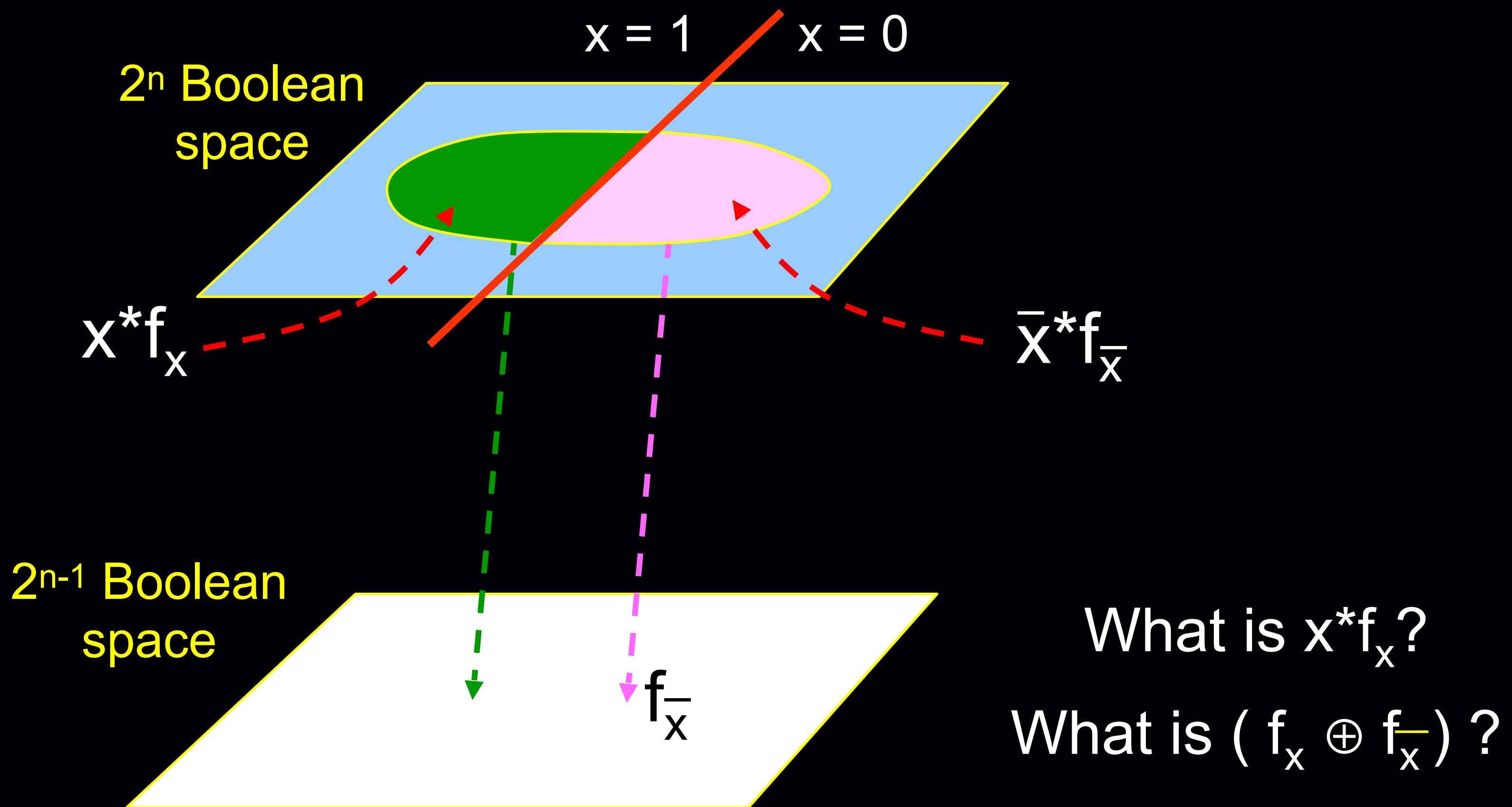
1. Dynamic BDD variable reordering
2. Reducing the BDD size by using cut
   - Local vs. global BDDs
   - Compose operation
3. Partial BDDs
4. Other types of decision diagrams
   - ZDD
   - ▶ FDD
   - MTBDD(ADD)
   - BMD

Before we go into FDD

- In Boolean logic, let a, b and c be Boolean variables, "$\oplus$" be XOR, "+/-" be plus/minus operations under Boolean field
  - $a + b = a \oplus b$
  - $a – b = a \oplus b$ // so $a + b = a – b$
  - If $a + b = c$, then $a + c = b$
- Let f be a Boolean function and x be one of its depending variable. Let df/dx be the Boolean difference f_d of f on x such that when there is a change on x, there will be a change on f.
  - $f_d = df/dx = f_x \oplus f_{\bar{x}}$

Something about co-factors...

- Fallacy: $f_x \wedge f_{\bar{x}} = \varnothing$



Reed-Muller Expansion (ExOR-polynomial)

- Let $f(x, Y) = x \cdot f_x(Y) + \overline{x} \cdot f_{\overline{x}}(Y)$ .... (1)
- Let $f_1(Y)$, $f_2(Y)$ be some functions of Y.
  Then f can be decomposed as $f_1 \oplus f_2 \cdot x$ ....(2)
  - What are f_1 and f_2?
    Rewrite: $f = f_2 \cdot x \oplus f_1$
  - What do they mean? What does $\oplus$ mean?
- Let x = 0. Compare (1) and (2). We have $f_1 = f_{\overline{x}}$
- Repeat for x = 1. We have: $f_x = f_1 \oplus f_2$
  - So, what are f_1 and f_2 (in terms of f_x and $f_{\overline{x}}$)?

  $f_1 = f_{\overline{x}}$

  $f_2 = f_x \oplus f_{\overline{x}} = f_d$
- Recursively decompose f_1 and f_2, we have:
  $f = f_d \cdot x \oplus f_{\overline{x}}$
  - $f_1 = f_3 \oplus f_5 \cdot y$ $f_2 = f_4 \oplus f_6 \cdot y$

=> $f = f_1 \oplus f_2 \cdot x = f_3 \oplus f_4 \cdot x \oplus f_5 \cdot y \oplus f_6 \cdot xy$

Function Decomposition

- Shannon Expansion (sum of product form)
  - $f = x\, f_x + \overline{x}\, f_{\overline{x}}$
- Reed-Muller Expansion (ExOR-polynomial)
  - $f = a_0 \oplus a_1 \cdot x_0 \oplus a_2 \cdot x_1 \oplus .... \oplus a_n x_{n-1}$
    $\oplus\, a_{n+1} \cdot x_0 \cdot x_1 \oplus ....$
    $....$
    $\oplus\, a_{2^{n-1}} \cdot x_0 \cdot x_1 \cdot x_2 \cdot x_3 .... \cdot x_{n-1}$ // Total 2^n terms

[In recursive form]

- $f = f_{\overline{x}} \oplus (f_x \oplus f_{\overline{x}}) \cdot x = f_{\overline{x}} \oplus f_d \cdot x$ (positive davio)
  $= f_x \oplus (f_{\overline{x}} \oplus \overline{f_x}) \cdot x = f_x \oplus f_d \cdot \overline{x}$ (negative davio)

Note:
$f_d = df/dx$
$= (f_x \oplus f_{\overline{x}})$

➔ f = (negative cofactor) $\oplus$ (Boolean difference) $\cdot\, x$
= (positive cofactor) $\oplus$ (Boolean difference) $\cdot\, \overline{x}$

[Sideline Help] Extracting Cofactors

- Let $F = x \cdot A + \overline{x} \cdot B + C$
  - ➔ positive cofactor: $F_x = A + C$
  - ➔ negative cofactor: $F_{\overline{x}} = B + C$
  - ➔ Boolean difference: $F_d = F_x \oplus F_{\overline{x}} = ??$

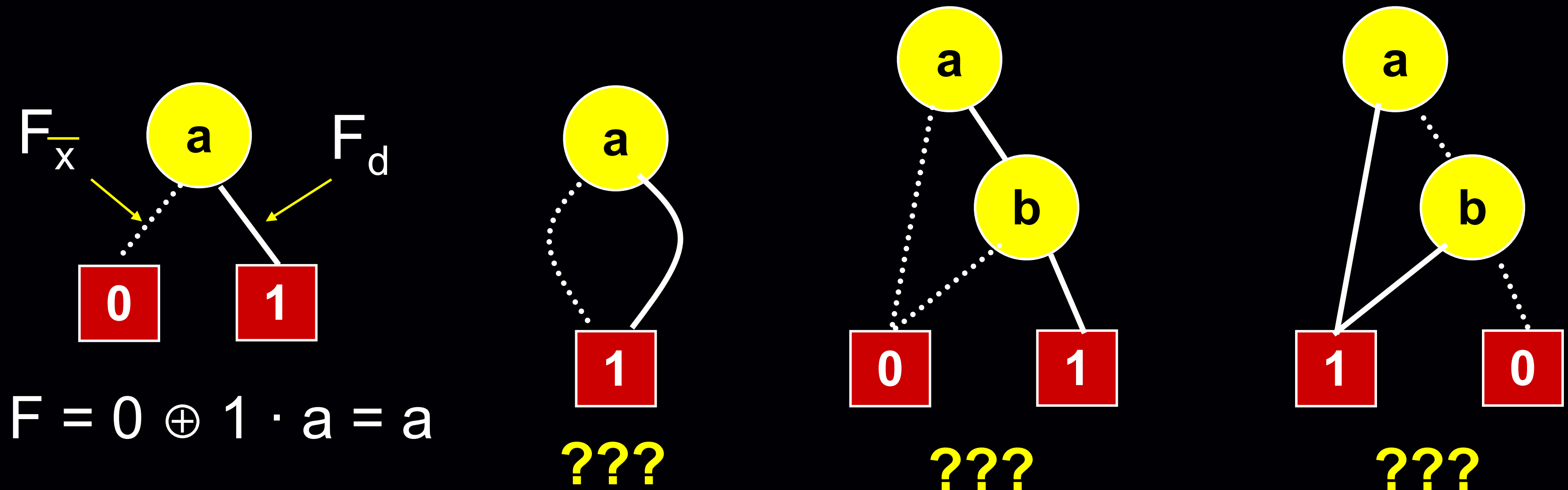
$$F_d = (A \oplus B) \cdot \overline{C}$$

Any intuitive explanation?

Ordered Functional Decision Diagram

[OFDD, Kebschull, EDAC 92]

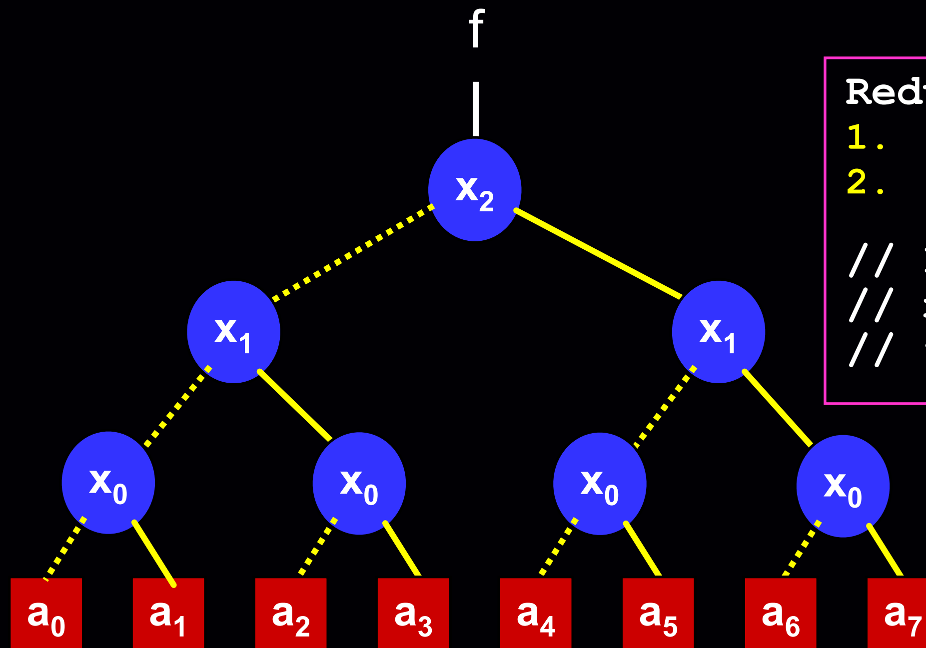
- Structurally similar to BDD, but each node use "positive (or negative) davio to explain
  - Still canonical
  - Good for XOR-like circuits
  - No positive edge to const 0



Reed-Muller Expansion for OFDD

- Reed-Muller Expansion (ExOR-polynomial)
  - $f = a_0 \oplus a_1 \cdot x_0 \oplus a_2 \cdot x_1 \oplus a_3 \cdot x_0 \cdot x_1 \oplus \dots$
    $\oplus\, a_{2^n-1} \cdot x_0 \cdot x_1 \cdot x_2 \cdot x_3 \dots \cdot x_{n-1}$

  [In recursive form]
  - $f = f_{\bar{x}} \oplus (f_x \oplus f_{\bar{x}}) \cdot x = f_{\bar{x}} \oplus f_d \cdot x$ (positive davio)
    $= f_x \oplus (f_x \oplus f_{\bar{x}}) \cdot \bar{x} = f_x \oplus f_d \cdot \bar{x}$ (negative davio)
- Let $f = a_0 \oplus a_1 \cdot x_0 \oplus a_2 \cdot x_1 \oplus a_3 \cdot x_0 \cdot x_1 \oplus a_4 \cdot x_2 \oplus a_5 \cdot x_0 \cdot x_2 \oplus a_6 \cdot x_1 \cdot x_2 \oplus a_7 \cdot x_0 \cdot x_1 \cdot x_2$




```
Reduction Rules:
1.  No repeated node
2.  No positive edge to
    constant 0
// Positive and
// negative edges to
// the same node is OK!
```

Ordered Kronecker Functional Decision Diagram (OKFDD)

[Drechsler, DAC 94]

- For each level of variable, choose one of the following expansion
  - Shannon
  - Positive Davio
  - Negative Davio
  - → Still canonical
- Provide more flexibility, and thus the BDD size could be smaller

03/19

Advanced BDD Techniques for Combinational Proof

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

DDs for Arithmetic Manipulation

- BDD, FDD, OKFDD, etc are useful in verifying Boolean functions
- What if the property contains word-level signals and arithmetic operations?
  - e.g. 8a + 5ab = 20
  - Word-level signals → decomposed into bits
- e.g. A truth table for arithmetic functions

```
an... a2 a1   bm... b2 b1   | f
----------------------------|---
 0...  0  0     0...  0  0  | 12
 0...  0  0     0...  0  1  | -8
 ...........................| ..
 1...  1  1     1...  1  1  |  7
```

→ Decompose ‘f’ into bits and use multiple functions (BDDs) for operations ?? (Pseudo-Boolean Functions)

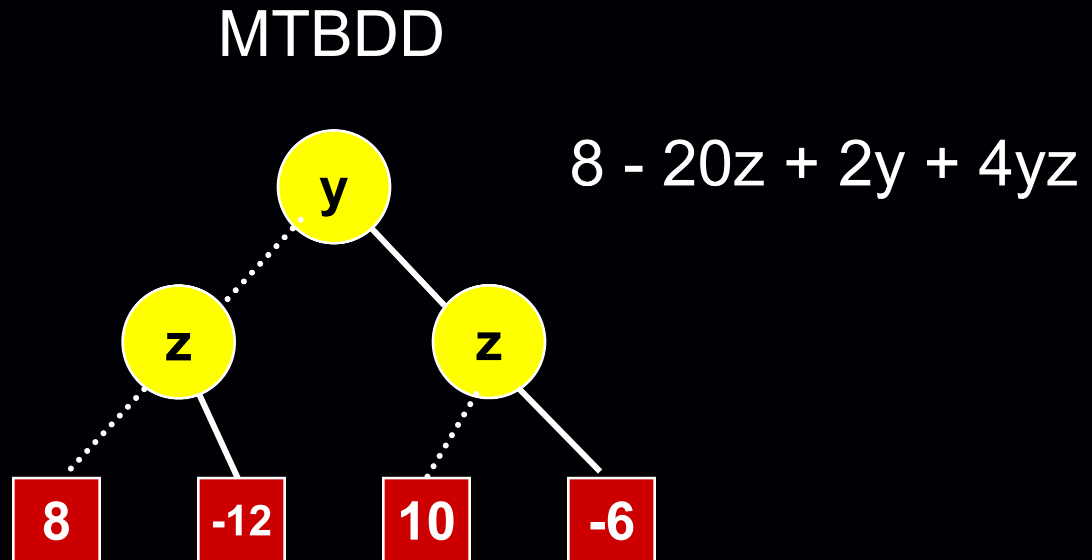
- f_0(an,..., a2, a1, bm,.., b2, b1), f_1(an,..., a2, a1, bm,.., b2, b1),...

Multi-Terminal BDD (also called ADD)

- Expanding the terminal of BDDs to multi-value nodes

  e.g. $8 - 20z + 2y + 4yz$, where y and z are Boolean variables

y	z	F
0	0	8
0	1	-12
1	0	10
1	1	-6



Boolean Moment Diagram (BMD)

[Bryant, DAC95]

- Just like FDD to BDD, where we use "Reed-Muller" (Boolean difference) instead of "Shannon Expansion"

  → We can apply "Reed-Muller Expansion" on MTBDD too

- Let x be Boolean → $\overline{x} = (1 - x)$

→ $f = x \cdot f_x + \overline{x} \cdot f_{\overline{x}} = x \cdot f_x + (1 - x) \cdot f_{\overline{x}}$

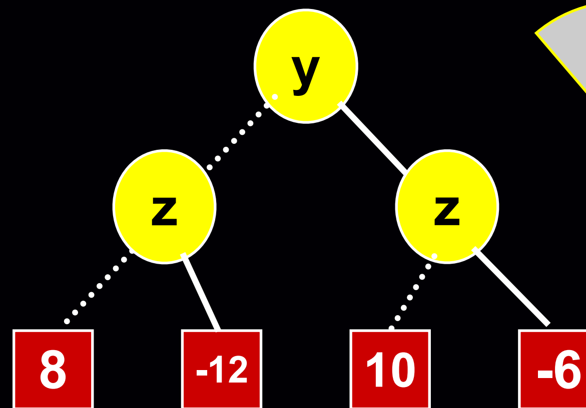
$= f_{\overline{x}} + x \cdot (f_x - f_{\overline{x}})$ ← postive davio form

From MTBDD to BMD

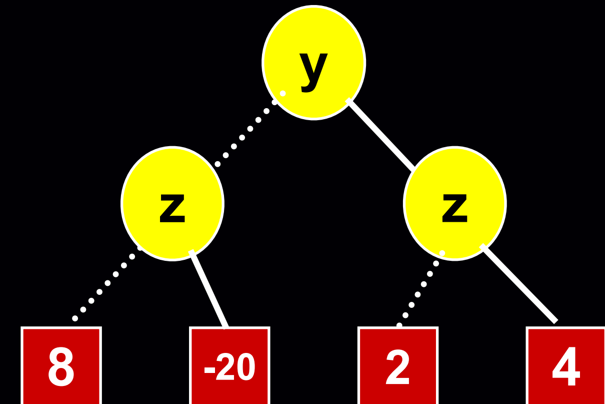
Function

y	z	F
0	0	8
0	1	-12
1	0	10
1	1	-6

MTBDD



BMD



$$F(y, z) = \begin{array}{ll} 8\,(1-y)(1-z)\ + & \ldots(00) \\ -12\,(1-y)\,z\ + & \ldots(01) \\ 10\,y\,(1-z)\ + & \ldots(10) \\ -6\,y\,z & \ldots(11) \end{array}$$

$$= 8 - 20z + 2y + 4yz$$

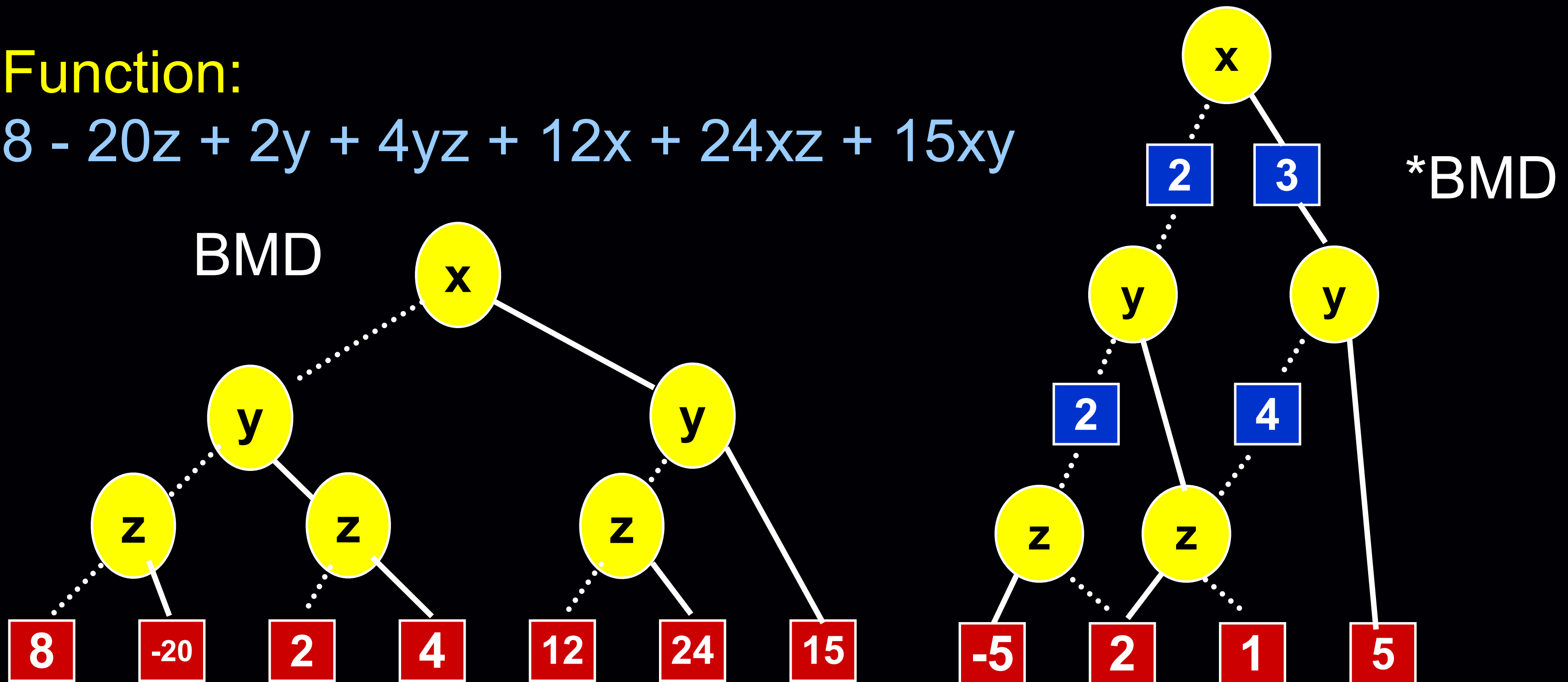
*BMD

- Similar to EVBDD, where an integer value is annotated on the edge as the weight

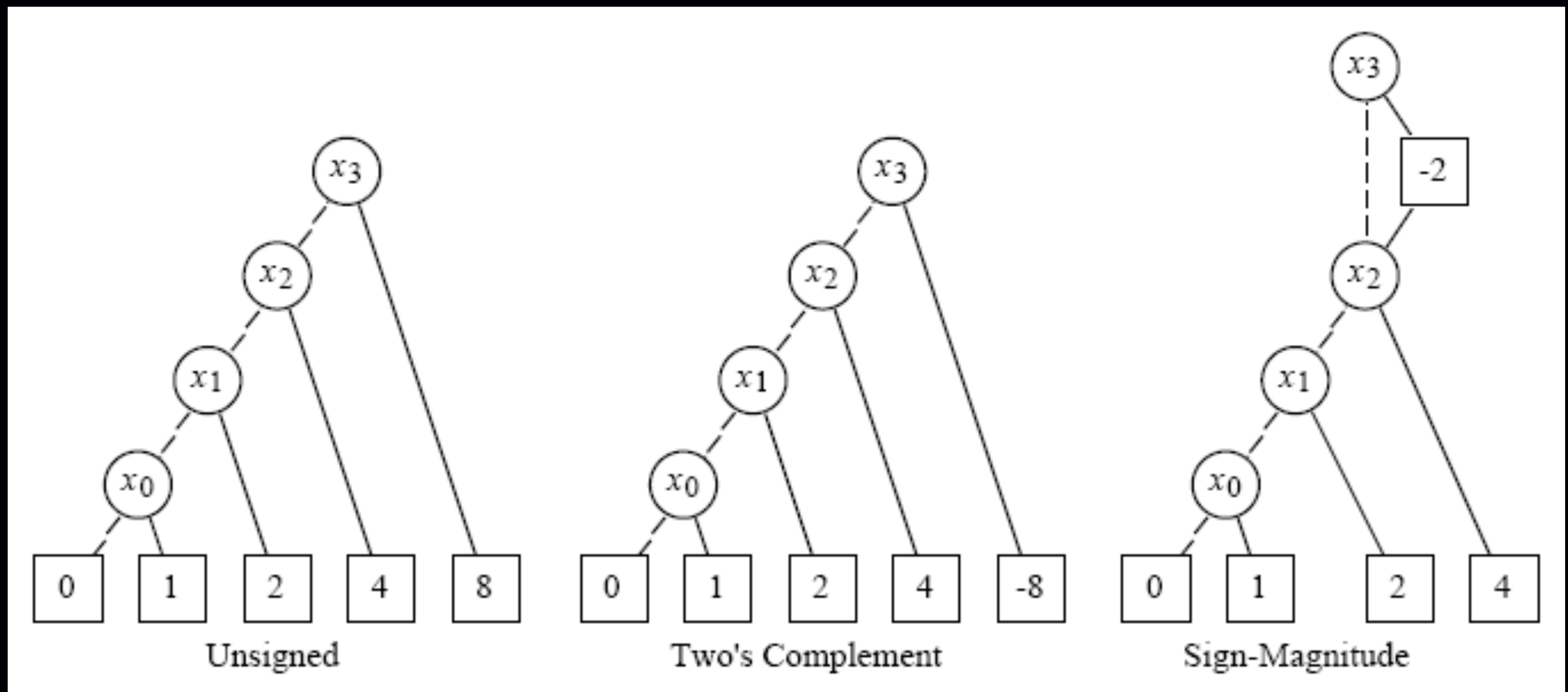
  ➔ We can factor out the terminal nodes and lift the common divisors to the edge values

Function:

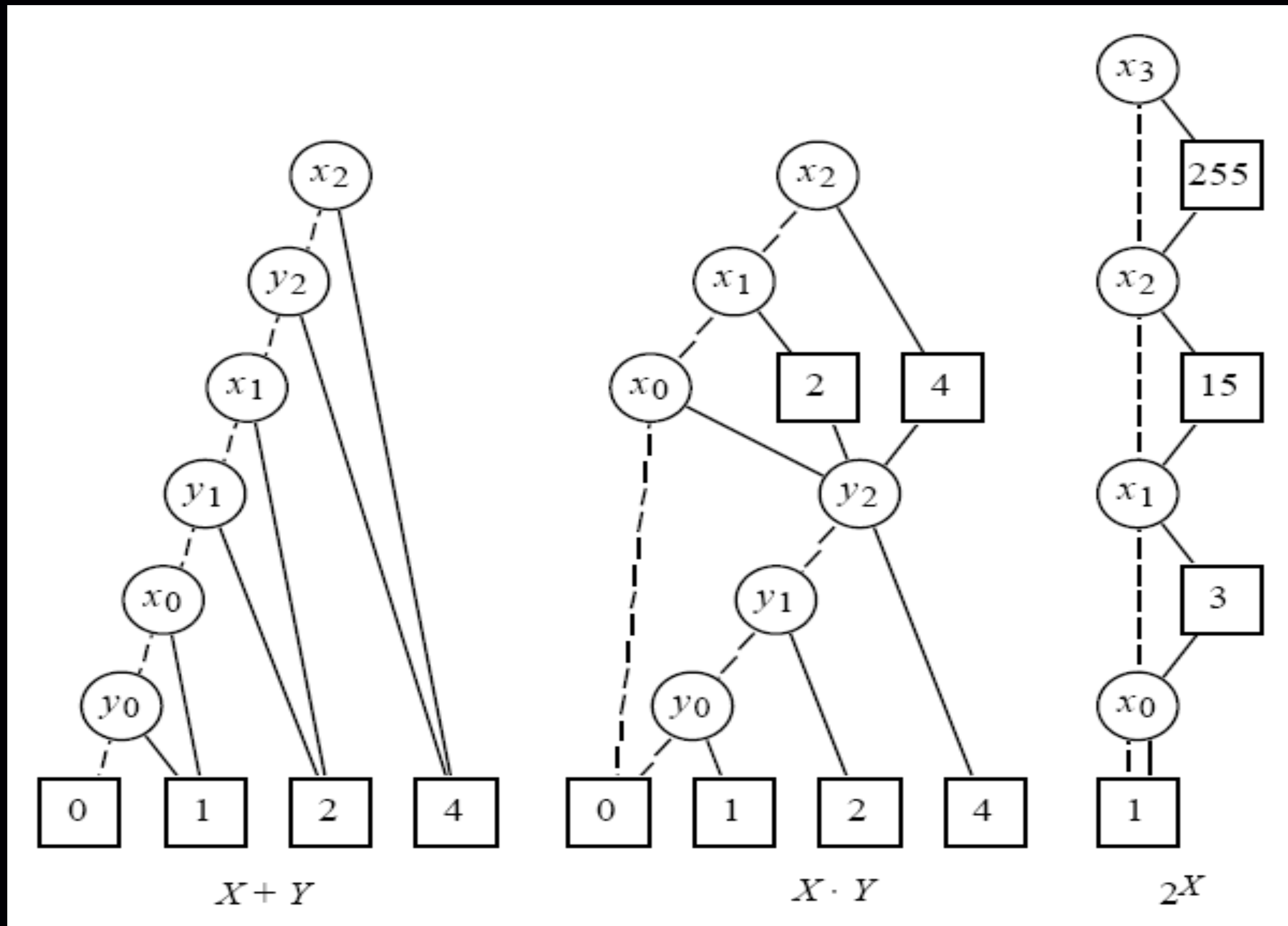
8 - 20z + 2y + 4yz + 12x + 24xz + 15xy



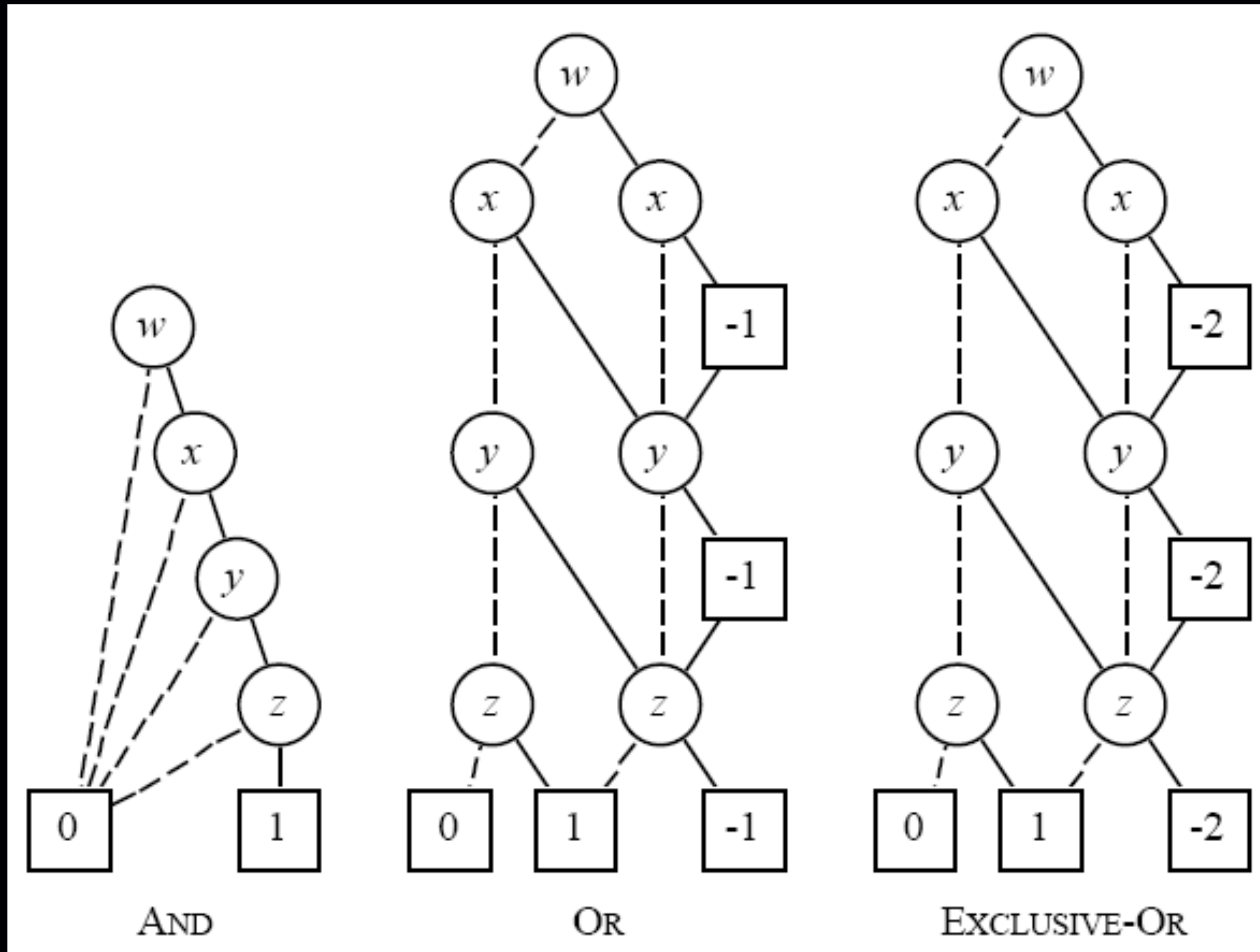
*BMD for Representation of Integers



*BMD for Word-Level Sum, Product, and Exponentiation

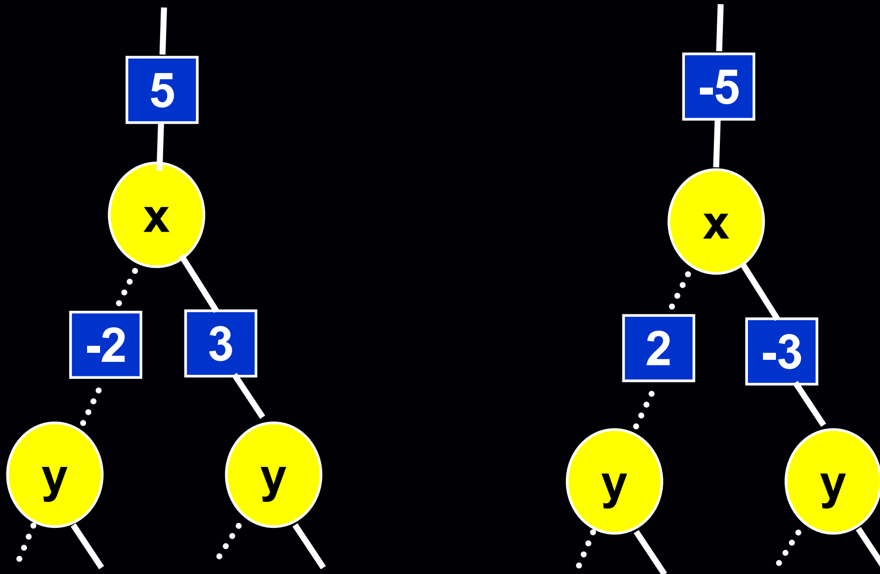


*BMD for Boolean Functions



Canonicity for *BMD

- Are these the same?



→ Need rules to ensure the canonicity!

→ Negative edge value on “pos-edge” only

Conclusion on various DDs

- We have seen various types decision diagrams
  - BDD
  - FDD
  - OKFDD
  - MTBDD (ADD)
  - EVBDD
  - BMD
  - *BMD

  ➔ There are more that are not covered in this class...
- A comprehensive study on the comparison / classification of the various DDs can be found at:
  - "Decision Diagrams for Discrete Functions: Classication and Unied Interpretation", Stankovic and Sasao, ASPDAC 98