Introduction to Electronic Design Automation

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Spring 2011

Formal Verification

Part of the slides are by courtesy of Prof. Y.-W. Chang, S.-Y. Huang, and A. Kuehlmann

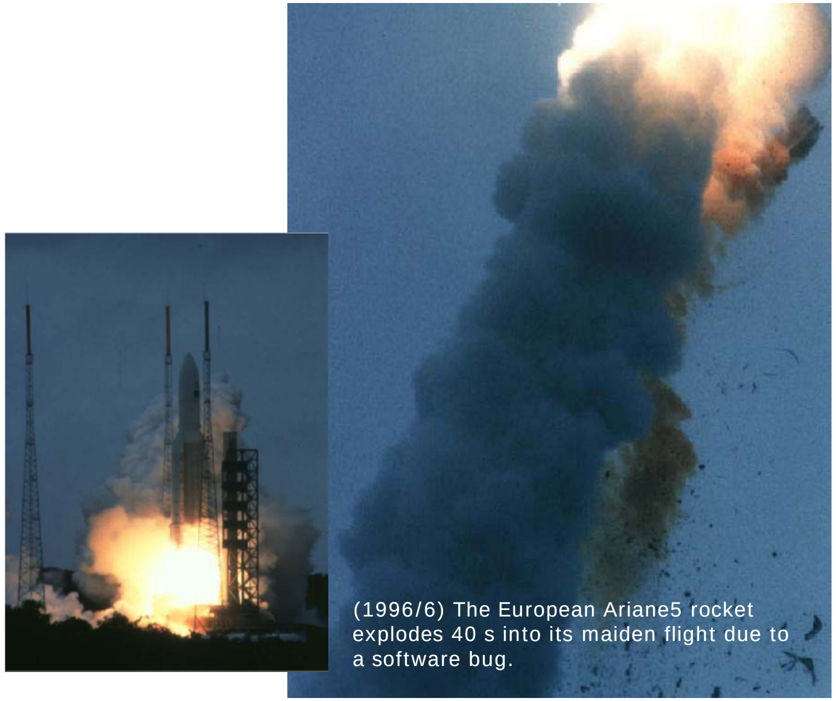
Formal Verification

- Course contents
  - Introduction
  - Boolean reasoning engines
  - Equivalence checking
  - Property checking
- Readings
  - Chapter 9

Outline

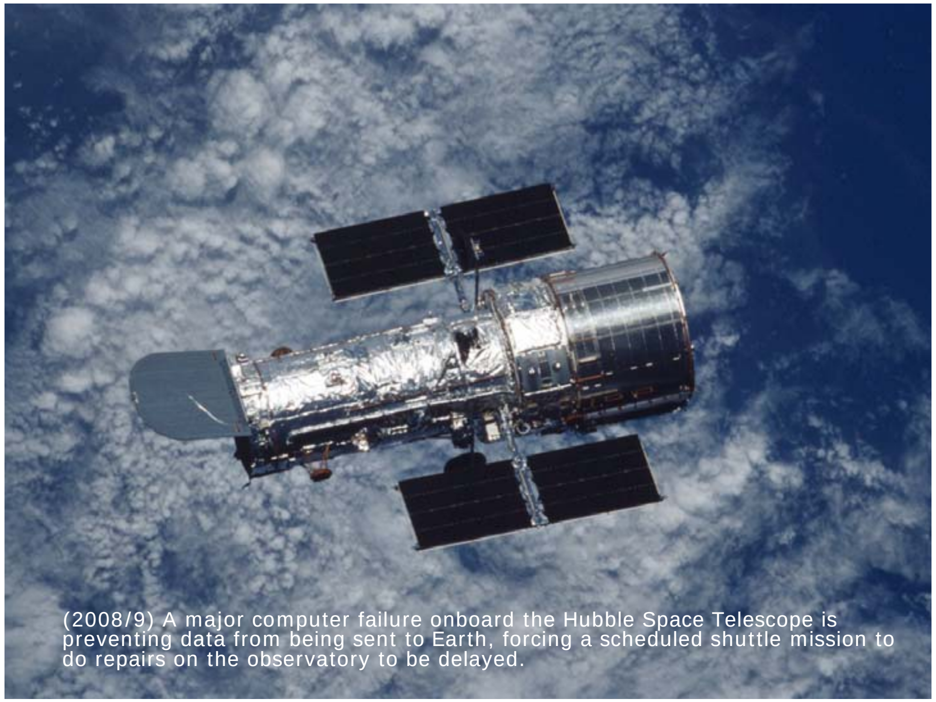
- Introduction
- Boolean reasoning engines
- Equivalence checking
- Property checking

(1995/1) Intel announces a pre-tax charge of 475 million dollars against earnings, ostensibly the total cost associated with replacement of the flawed processors.



(1996/6) The European Ariane5 rocket explodes 40 s into its maiden flight due to a software bug.

(2003/8) A programming error has been identified as the cause of the Northeast power blackout, which affected an estimated 10 million people in Canada and 45 million people in the U.S.



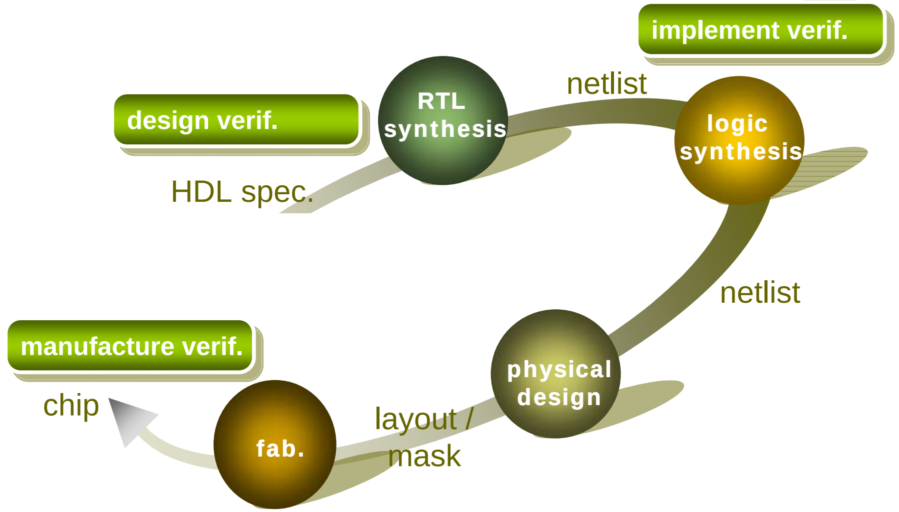
(2008/9) A major computer failure onboard the Hubble Space Telescope is preventing data from being sent to Earth, forcing a scheduled shuttle mission to do repairs on the observatory to be delayed.

Design vs. Verification

- Verification may take up to 70% of total development time of modern systems !
  - This ratio is ever increasing
  - Some industrial sources show 1:3 head-count ratio between design and verification engineers

- Verification plays a key role to reduce design time and increase productivity

IC Design Flow and Verification

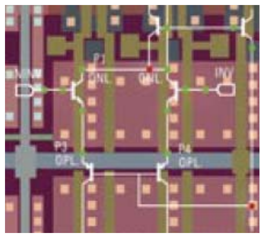


Scope of Verification

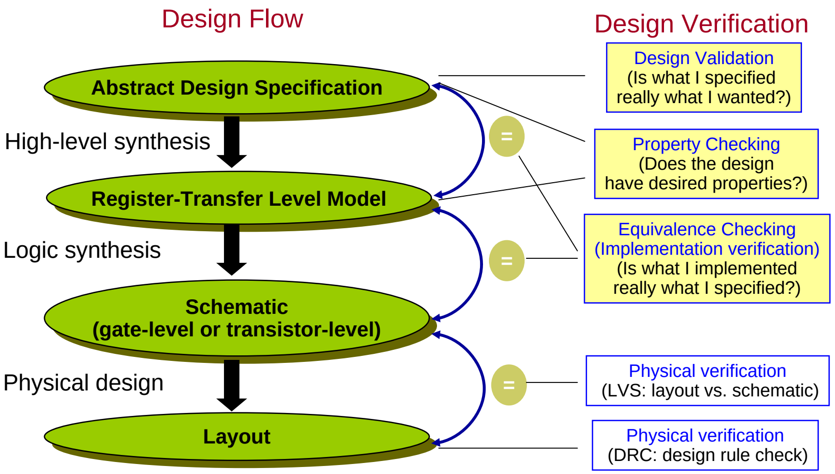
- Design flow
  - A series of transformations from abstract specification all the way to layout
- Verification enters design flow in almost all abstraction levels
  - Design verification
    - Functional property verification (main focus)
  - Implementation verification
    - Functional equivalence verification (main focus)
    - Physical verification
    - Timing verification
    - Power analysis
    - Signal integrity check
      - Electro-migration, IR-drop, ground bounce, cross-talk, etc.
  - Manufacture verification
    - Testing

Verification

- Design/Implementation Verification
  - Functional Verification
    - Property checking in system level
      - PSPACE-complete
    - Equivalence checking in RTL and gate level
      - PSPACE-complete
  - Physical Verification
    - DRC (design rule check) and LVS (layout vs. schematic check) in layout level
      - Tractable
- Manufacture Verification
  - Testing
    - NP-complete
- "Verification" often refers to functional verification



Functional Verification



Functional Verification Approaches

- Simulation (software)
  - Incomplete (i.e., may fail to catch bugs)
  - Time-consuming, especially at lower abstraction levels such as gate- or transistor-level
  - Still the most popular way for design validation
- Emulation (hardware)
  - FPGA-based emulation systems, emulation system based on massively parallel machines (e.g., with 8 boards, 128 processors each), etc.
  - 2 to 3 orders of magnitude faster than software simulation
  - Costly and may not be easy-to-use
- Formal verification
  - a relatively new paradigm for property checking and equivalence checking
  - requires no input stimuli
  - perform exhaustive proof through rigorous logical reasoning

Informal vs. Formal Verification

- Informal verification
  - Functional simulation aiming at locating bugs
  - Incomplete
    - Show existence of bugs, but not absence of bugs
- Formal verification
  - Mathematical proof of design correctness
  - Complete
    - Show both existence and absence of bugs

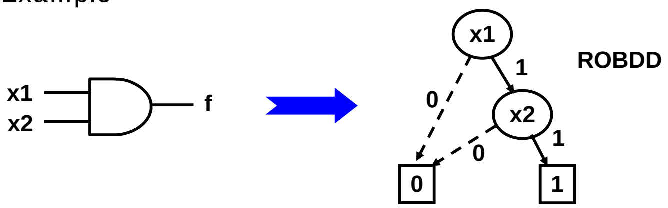
We will be focusing on formal verification

Outline

- Introduction
- Boolean reasoning engines
  - BDD
  - SAT
- Equivalence checking
- Property checking

Binary Decision Diagram (BDD)

- Basic features
  - ROBDD
    - Proposed by R.E. Bryant in 1986
    - A directed acyclic graph (DAG) representing a Boolean function f: $B^n \rightarrow B$
      - Each non-terminal node is a decision node associated with a input variable with two branches: 0-branch and 1-branch
      - Two terminal nodes: 0-terminal and 1-terminal
  - Example



Binary-Decision Diagram (BDD)

- Cofactor of Boolean function:
  - Positive cofactor w.r.t. x_i: $f_{xi} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$
  - Negative cofactor w.r.t. x_i: $f_{\neg xi} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$
  - Example

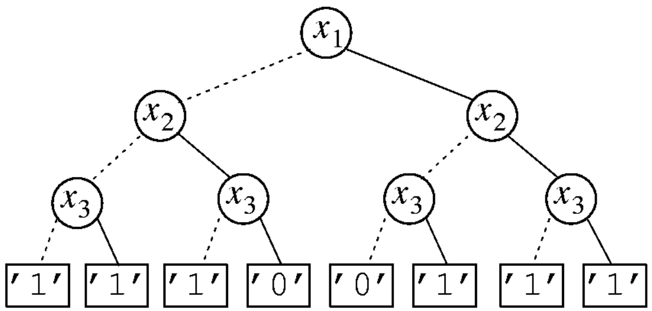
    $f = x_1' x_2' x_3' + x_1' x_2' x_3 + x_1 x_2' x_3 + x_1 x_2 x_3' + x_2 x_3$

    $f_{x1} = x_2' x_3 + x_2 x_3' + x_2 x_3$

    $f_{x1'} = x_2' x_3' + x_2' x_3 + x_2 x_3$
- Shannon expansion: $f = x_i f_{xi} + x_i' f_{xi'}$
  - A complete expansion of a function can be obtained by successively applying Shannon expansion on all variables until either of the constant functions '0' or '1' is reached

Ordered BDD (OBDD)

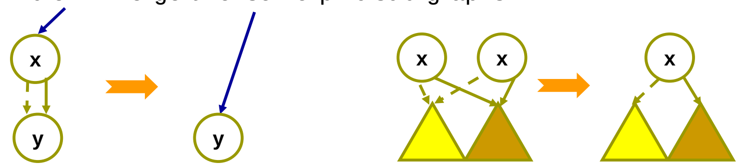
- Complete Shannon expansion can be visualized as a binary tree
  - Solid (dashed) lines correspond to the positive (negative) cofactor



$$f = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3$$

Reduced OBDD (ROBDD)

- Reduction rules of ROBDD
  - Rule 1: eliminate a node with two identical children
  - Rule 2: merge two isomorphic sub-graphs



- Reduction procedure
  - Input: An OBDD
  - Output: An ROBDD
  - Traverse the graph from the terminal nodes towards to root node (i.e., in a bottom-up manner) and apply the above reduction rules whenever possible

ROBDD

- An OBDD is a directed tree *G*(*V*,*E*)
- Each vertex *v* ∈ *V* is characterized by an associated variable ϕ(*v*), a *high* subtree η(*v*) (high(*v*), the 1-branch) and a *low* subtree λ(*v*) (low(*v*), the 0-branch)
- Procedure to reduce an OBDD:
  - Merge all identical leaf vertices and appropriately redirect their incoming edges
  - Proceed from bottom to top, process all vertices: if two vertices *u* and *v* are found for which ϕ(*u*) = ϕ(*v*), η(*u*) = η(*v*), and λ(*u*) = λ(*v*), merge *u* and *v* and redirect incoming edges
  - For vertices *v* for which η(*v*) = λ(*v*), remove *v* and redirect its incoming edges to η(*v*)

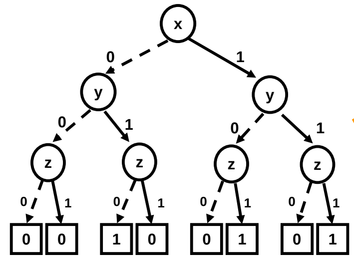
ROBDD

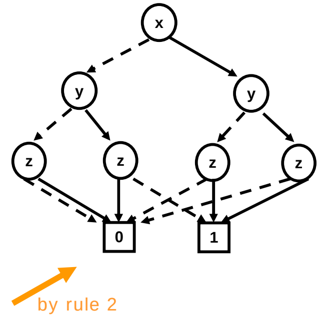
- Example
  - f = x'yz' + xz
  - variable order: x < y < z

Truth table

xyz	f
000	0
001	0
010	1
011	0
100	0
101	1
110	0
111	1

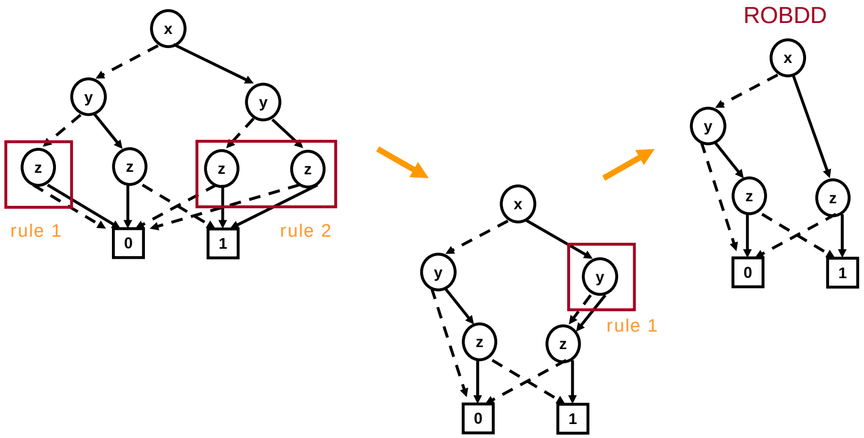
OBDD





ROBDD

- Example (cont'd)



Canonicity

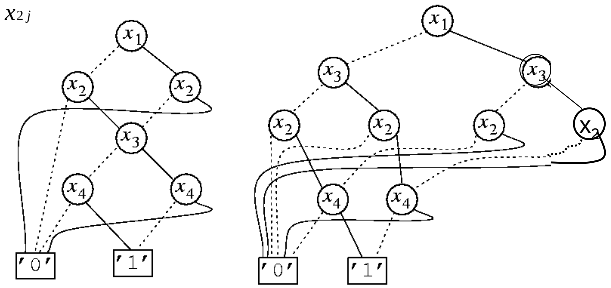
- Canonicity requirements
  - A BDD representation is not canonical for a given Boolean function unless the following constraints are satisfied:

1. Simple BDD – each variable can appear only once along each path from the root to a leaf
2. Ordered BDD – Boolean variables are ordered in such a way that if the node labeled x_i has a child labeled x_k, then order(x_i) < order(x_k)
3. Reduced BDD – no two nodes represent the same function, i.e., redundancies are removed by sharing isomorphic sub-graphs

ROBDD Properties

- ROBDD is a canonical representation for a fixed variable ordering
- ROBDD is compact in representing many Boolean functions used in practice
- Variable ordering greatly affects the size of an ROBDD
  - E.g., the parity function of k bits:

$$f = \prod_{j=1}^{k} x_{2j-1} \oplus x_{2j}$$



Effects of Variable Ordering

- BDD size
  - Can vary from linear to exponential in the number of the variables, depending on the ordering
- Hard-to-build BDD
  - Datapath components (e.g., multipliers) cannot be represented in polynomial space, regardless of the variable ordering
- Heuristics of ordering
  - (1) Put the variable that influence most on top
  - (2) Minimize the distance between strongly related variables

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

    x1 < x2 < x3 < x4 is better than x1 < x4 < x2 < x3

BDD Package

- A BDD package refers to a software program that supports Boolean manipulation using ROBDDs. It has the following features:
  - It provides convenient API (application programming interface)
  - It supports the conversion between the external Boolean function representation and the internal ROBDD representation
  - Multiple Boolean functions are stored in shared ROBDD
  - It can create new functions from existing ones (e.g., $h = f \bullet g$)

BDD Data Structure

- A triplet (ϕ,η,λ) uniquely identifies an ROBDD vertex
- A unique table (implemented by a hash table) that stores all triplets already processed

```
struct vertex {
  char *ϕ;
  struct vertex *η, *λ;
  ...
}
```

```
struct vertex *old_or_new(char *ϕ, struct vertex *η, *λ)
{
  if ("a vertex v = (ϕ, η, λ) exists")
    return v;
  else {
    v ← "new vertex pointing at (ϕ, η, λ)";
    return v;
  }
}
```

Building ROBDD

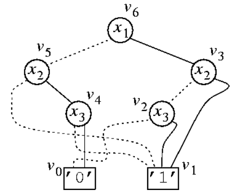
```
struct vertex *robdd_build(struct expr f, int i)
{
  struct vertex *η, *λ;
  struct char *φ;

  if (equal(f, '0'))
    return v_0;
  else if (equal(f, '1'))
    return v_1;
  else {
    φ ← π(i);
    η ← robdd_build(f_φ, i + 1);
    λ ← robdd_build(f_φ', i + 1);
    if (η = λ)
      return η;
    else
      return old_or_new(φ, η, λ);
  }
}
```

- The procedure directly builds the compact ROBDD structure
- A simple symbolic computation system is assumed for the derivation of the cofactors
- $\pi(i)$ gives the i^{th} variable from the top

Building ROBDD

- Example

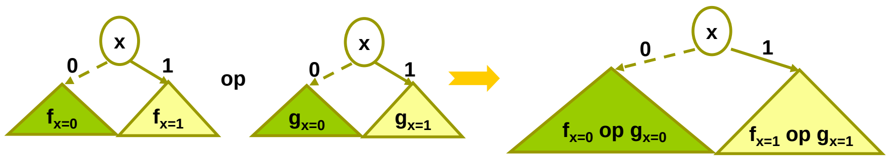
$\text{robdd\_build}(\overline{x_1}\cdot\overline{x_3}+\overline{x_2}\cdot x_3+x_1\cdot x_2, 1)$
$\xrightarrow{\eta}$ $\text{robdd\_build}(\overline{x_2}\cdot x_3+x_2, 2)$
  $\xrightarrow{\eta}$ $\text{robdd\_build}('1', 3)$
    v_1
  $\xrightarrow{\lambda}$ $\text{robdd\_build}(x_3, 3)$
    $\xrightarrow{\eta}$ $\text{robdd\_build}('1', 4)$
      v_1
    $\xrightarrow{\lambda}$ $\text{robdd\_build}('0', 4)$
      v_0
    $v_2 = (x_3, v_1, v_0)$
  $v_3 = (x_2, v_1, v_2)$



$\xrightarrow{\lambda}$ $\text{robdd\_build}(\overline{x_3}+\overline{x_2}\cdot x_3, 2)$
  $\xrightarrow{\eta}$ $\text{robdd\_build}(\overline{x_3}, 3)$
    $\xrightarrow{\eta}$ $\text{robdd\_build}('0', 4)$
      v_0
    $\xrightarrow{\lambda}$ $\text{robdd\_build}('1', 4)$
      v_1
    $v_4 = (x_3, v_0, v_1)$
  $\xrightarrow{\lambda}$ $\text{robdd\_build}(\overline{x_3}+x_3, 3)$
    $\xrightarrow{\eta}$ $\text{robdd\_build}('1', 4)$
      v_1
    $\xrightarrow{\lambda}$ $\text{robdd\_build}('1', 4)$
      v_1
    v_1
  $v_5 = (x_2, v_4, v_1)$
$v_6 = (x_1, v_3, v_5)$

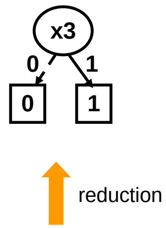
Recursive BDD Operation

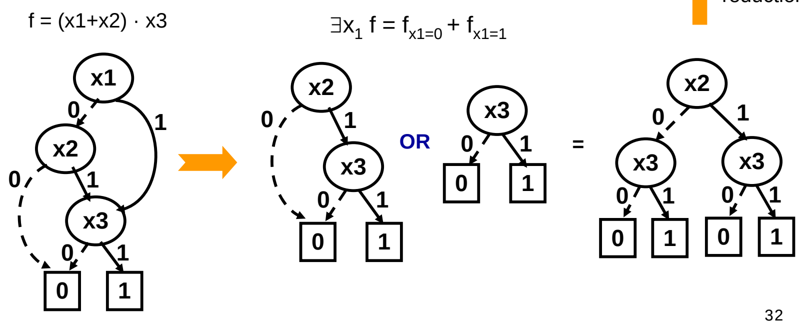
- Construct the ROBDD h = f <op> g from two existing ROBDDs f and g, where <op> is a binary Boolean operator (e.g. AND, OR, NAND, NOR)
  - A recursive procedure on each variable x
    - h $= x \cdot h_{x=1} + x' \cdot h_{x=0}$
      $= x \cdot (f \text{ <op> } g)_{x=1} + x' \cdot (f \text{ <op> } g)_{x=0}$
      $= x \cdot (f_{x=1} \text{ <op> } g_{x=1}) + x' (f_{x=0} \text{ <op> } g_{x=0})$
      - $(f \text{ <op> } g)_x = (f_x \text{ <op> } g_x)$ for <op> = AND, OR, NAND, NOR



Recursive BDD Operation

- Existential quantification
  Let $\exists x_1\ [f(x_1,y_1,\ldots,y_n)] = g(y_1,\ldots,y_n)$.
  Then $g(y_1,\ldots,y_n) = 1$ iff
  $f(0,y_1,\ldots,y_n)=1$ or $f(1,y_1,\ldots,y_n)=1$





ROBDD Manipulation

- Separate algorithms could be designed for each operator on ROBDDs, such as AND, NOR, etc. However, the universal if-then-else operator '*ite*' is sufficient.
  z = *ite*(*f*,*g*,*h*), *z* equals *g* when *f* is true and equals *h* otherwise:
  - Example:

$$z = ite(f,g,h) = f \cdot g + \bar{f} \cdot h$$
$$z = f \cdot g = ite(f,g,'0')$$
$$z = f + g = ite(f,'1',g)$$

- The *ite* operator is well-suited for a recursive algorithm based on ROBDDs ($\phi(v) = x$):

$$v = ite(F,G,H) = (x, ite(F_x, G_x, H_x), ite(F_{\bar{x}}, G_{\bar{x}}, H_{\bar{x}}))$$

ITE Operator

- ITE operator ite(f,g,h) = fg + f'h can implement any two variable logic function. There are 16 such functions corresponding to all subsets of vertices of $\mathbf{B}^2$:

Table	Subset	Expression	Equivalent Form
0000	0	0	0
0001	AND(f, g)	f g	ite(f, g, 0)
0010	f > g	f g′	ite(f, g′, 0)
0011	f	f	f
0100	f < g	f′g	ite(f, 0, g)
0101	g	g	g
0110	XOR(f, g)	f ⊕ g	ite(f, g′, g)
0111	OR(f, g)	f + g	ite(f, 1, g)
1000	NOR(f, g)	(f + g)′	ite(f, 0, g′)
1001	XNOR(f, g)	f ⊕ g′	ite(f, g, g′)
1010	NOT(g)	g′	ite(g, 0, 1)
1011	f ≥ g	f + g′	ite(f, 1, g′)
1100	NOT(f)	f′	ite(f, 0, 1)
1101	f ≤ g	f′ + g	ite(f, g, 1)
1110	NAND(f, g)	(f g)′	ite(f, g′, 1)
1111	1	1	1

Recursive Formulation of ITE

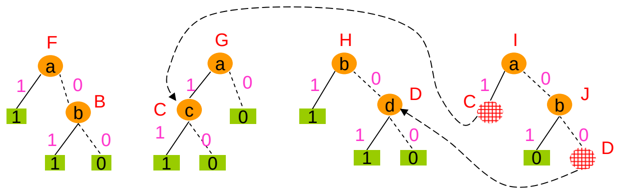
- Ite(f,g,h)

$$= f\,g + f'\,h$$
$$= v\,(f\,g + f'\,h)_v + v'\,(f\,g + f'\,h)_{v'}$$
$$= v\,(f_v\,g_v + f'_v\,h_v) + v'\,(f_{v'}\,g_{v'} + f'_{v'}\,h_{v'})$$
$$= ite(v, ite(f_v,g_v,h_v), ite(f_{v'},g_{v'},h_{v'}))$$

where *v* is the top-most variable of BDDs *f*, *g*, *h*

ITE Operator

- Example



F,G,H,I,J,B,C,D are pointers

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

Check: F = a + b
G = ac
H = b + d
ite(*F*, *G*, *H*) = (a + b)(ac) + a′b′(b + d) = ac + a′b′d

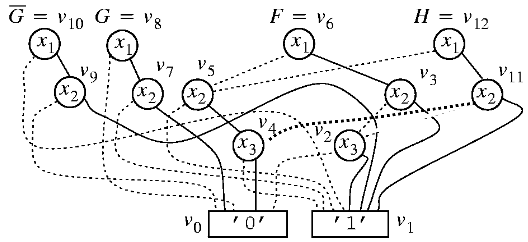
ITE Operator

```
struct vertex *apply_ite(struct vertex *F, *G, *H, int i)
{
  char x;
  struct vertex *η, *λ;

  if (F = v1)
    return G;
  else if (F = v0)
    return H;
  else if (G = v1 && H = v0)
    return F;
  else {
    x ← π(i);
    η ← apply_ite(Fx, Gx, Hx, i + 1);
    λ ← apply_ite(Fx̄, Gx̄, Hx̄, i + 1);
    if (η = λ)
      return η;
    else
      return old_or_new(x, η, λ);
  }
}
```

- ITE algorithm processes the variables in the order used in the BDD package
  - $\pi(i)$ gives the i^{th} variable from the top; $\pi^{-1}(x)$ gives the index position of variable x from the top
- Cofactor: Suppose F is the root vertex of the function for which F_x should be computed. Then
  $F_x = \eta(F)$ if $\pi^{-1}(\phi(F)) = i$
  - $F_{x'}$ can be calculated similarly
- The time complexity of the algorithm is $O(|F|\cdot|G|\cdot|H|)$

ITE Operator

- Example

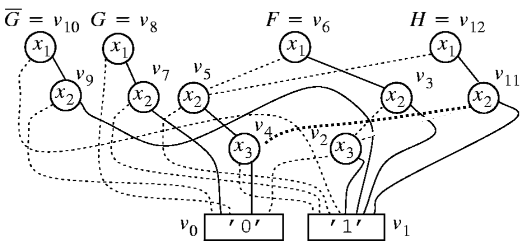


$\overline{G} = \text{ite}(G, 0, 1)$

$\text{apply\_ite}(v_8, v_0, v_1, 1)$
$\xrightarrow{\eta} \text{apply\_ite}(v_7, v_0, v_1, 2)$
$\quad\xrightarrow{\eta} \text{apply\_ite}(v_0, v_0, v_1, 3)$
$\quad\quad v_1$
$\quad\xrightarrow{\lambda} \text{apply\_ite}(v_1, v_0, v_0, 3)$
$\quad\quad v_0$
$\quad v_9 = (x_2, v_1, v_0)$
$\xrightarrow{\lambda} \text{apply\_ite}(v_0, v_0, v_1, 2)$
$\quad v_1$
$v_{10} = (x_1, v_9, v_1)$

ITE Operator

- Example (cont'd)



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

$\text{apply\_ite}(v_6, v_{10}, v_8, 1)$
$\xrightarrow{\eta} \text{apply\_ite}(v_3, v_9, v_7, 2)$
$\quad\xrightarrow{\eta} \text{apply\_ite}(v_1, v_1, v_0, 3)$
$\quad\quad v_1$
$\quad\xrightarrow{\lambda} \text{apply\_ite}(v_2, v_0, v_1, 3)$
$\quad\quad\xrightarrow{\eta} \text{apply\_ite}(v_1, v_0, v_1, 4)$
$\quad\quad\quad v_0$
$\quad\quad\xrightarrow{\lambda} \text{apply\_ite}(v_0, v_0, v_1, 4)$
$\quad\quad\quad v_1$
$\quad\quad v_4 = (x_3, v_0, v_1)$
$\quad v_{11} = (x_2, v_1, v_4)$
$\xrightarrow{\lambda} \text{apply\_ite}(v_5, v_1, v_0, 2)$
$\quad v_5$
$v_{12} = (x_1, v_{11}, v_5)$

BDD Memory Management

- Ordering
  - Finding the best ordering minimizing ROBDD sizes is intractable
  - Optimal ordering may change as ROBDDs are being manipulated
    - An ROBDD package may reorder the variables at different moments
    - It can move some variable closer to the top or bottom by remembering the best position, and repeat the procedure for other variables
- Garbage collection
  - Another important technique, in addition to variable ordering, for memory management