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
    - 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

### Design Flow

- **Abstract Design Specification**
- **Register-Transfer Level Model**
  - **Schematic** (gate-level or transistor-level)
  - **Physical design**
  - **Layout**

#### Design Verification

- **Design Validation**
  - (Is what I specified really what I wanted?)
- **Property Checking**
  - (Does the design have desired properties?)
- **Equivalence Checking**
  - (Implementation verification)
  - (Is what I implemented really what I specified?)
- **Physical verification**
  - (LVS: layout vs. schematic)

### 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 $f: \mathbb{B}^n \rightarrow \mathbb{B}$
      - Each non-terminal node is a decision node associated with an input variable with two branches: 0-branch and 1-branch
      - Two terminal nodes: 0-terminal and 1-terminal
  - Example

```
    x1  x2  f
   0    1
      0
```

- **Cofactor of Boolean function**:
  - Positive cofactor w.r.t. $x_i$: $f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$
  - Negative cofactor w.r.t. $x_i$: $f_{\overline{x_i}} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$
  - Example
    
    \[ f = x_1 \overline{x_2} x_3 + x_1 x_2 x_3 + x_1 x_2 x_3 + x_1 \overline{x_2} x_3 + x_1 x_2 x_3 + x_1 x_2 x_3 \]
    
    \[ f_{x_1} = \overline{x_2} x_3 + x_2 x_3 \]
    
- **Shannon expansion**: $f = x_i f_{x_i} + x_i' f_{\overline{x_i}}$
  - 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

```
    x1
    / \  \
  x2  x2
  / \ / \  \
x3 x3 x3 x3
```

\[ f = \overline{x_1} x_2 x_3 + \overline{x_1} x_2 \overline{x_3} + \overline{x_1} x_2 x_3 + x_1 \overline{x_2} \overline{x_3} + x_1 x_2 x_3 + x_1 x_2 \overline{x_3} + x_1 x_2 x_3 \]

Reduced OBDD (ROBDD)

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

```
    x  y
    \--\  \--\  \\
    \  \  \  \  \\
    \  \  \  \  \\
    \  \  \  \  \\
    \  \  \  \  \\
```

- **Reduction procedure**
  - Input: An OBDD
  - Output: An ROBDD
  - Traverse the graph from the terminal nodes towards 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 $v \in V$ is characterized by an associated variable $\phi(v)$, a *high* subtree $\eta(v)$ (high($v$), the 1-branch) and a *low* subtree $\lambda(v)$ (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 $\phi(u) = \phi(v)$, $\eta(u) = \eta(v)$, and $\lambda(u) = \lambda(v)$, merge $u$ and $v$ and redirect incoming edges
  - For vertices $v$ for which $\eta(v) = \lambda(v)$, remove $v$ and redirect its incoming edges to $\eta(v)$

Example

- $f = x'yz' + xz$
- Variable order: $x < y < z$

<table>
<thead>
<tr>
<th>$xyz$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>

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 $\text{order}(x_i) < \text{order}(x_k)$
    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., $x_1x_2 + x_2x_3 + x_3x_4$)
    $x_1 < x_2 < x_3 < x_4$ is better than $x_1 < x_4 < x_2 < x_3$.

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., $h = f \cdot g$).

BDD Data Structure

- A triplet $(\phi, \eta, \lambda)$ uniquely identifies an ROBDD vertex.
- A unique table (implemented by a hash table) that stores all triplets already processed.

```
struct vertex {
    char *phi;
    struct vertex *eta, *lambda;
}
```

```
struct vertex *add_or_new(char *phi, struct vertex *eta, *lambda)
{
    if ("a vertex v = (phi, eta, lambda) exists")
        return v;
else
    v = "new vertex pointing at (phi, eta, lambda)";
    return v;
}
```
Building ROBDD

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.

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 $x$
    - $h = x \cdot h_{x=1} + x' \cdot h_{x=0}$
    - $h = x \cdot (f <op> g)_{x=1} + x' \cdot (f <op> g)_{x=0}$
    - $h = x \cdot (f_{x=1} <op> g_{x=1}) + x' \cdot (f_{x=0} <op> g_{x=0})$
    - $(f <op> g)_x = (f_x <op> g_x)$ for $<op> = \text{AND, OR, NAND, NOR}$

Example

Existential quantification

Let $\exists x_1 [f(x_1, y_1, ..., y_n)] = g(y_1, ..., y_n)$. Then $g(y_1, ..., y_n) = 1$ iff $f(0, y_1, ..., y_n) = 1$ or $f(1, y_1, ..., y_n) = 1$
ROBDD Manipulation

Separate algorithms could be designed for each operator on ROBDDs, such as AND, NOR, etc. However, the universal if-then-else operator \( \text{ite} \) is sufficient.

\[ z = \text{ite}(f, g, h), \]  
\( z \) equals \( g \) when \( f \) is true and equals \( h \) otherwise:

Example:

\[ z = \text{ite}(f, g, h) = f \cdot g + f' \cdot h \]

\[ z = f \cdot g = \text{ite}(f, g, '0') \]

\[ z = f + g = \text{ite}(f, g, '1') \]

The \( \text{ite} \) operator is well-suited for a recursive algorithm based on ROBDDs (\( \varphi(v) = x \)):

\[ v = \text{ite}(F, G, H) = (x, \text{ite}(F_x, G_x, H_x), \text{ite}(F_{x'}, G_{x'}, H_{x'})) \]

Recursive Formulation of ITE

\( \text{Ite}(f, g, h) \)

\[ = f \cdot g + f' \cdot h \]

\[ = \text{v} (f \cdot g + f' \cdot h)_{\text{v}} + \text{v'} (f \cdot g + f' \cdot h)_{\text{v'}} \]

\[ = \text{v} (f_{\text{v}} g_{\text{v}} + f'_{\text{v}} h_{\text{v}})_{\text{v}} + \text{v'} (f_{\text{v'}} g_{\text{v'}} + f'_{\text{v'}} h_{\text{v'}})_{\text{v'}} \]

\[ = \text{ite}(\text{v}, \text{ite}(f_{\text{v}} g_{\text{v}}, h_{\text{v}}), \text{ite}(f_{\text{v'}} g_{\text{v'}}, h_{\text{v'}})) \]

where \( \text{v} \) is the top-most variable of BDDs \( f, g, h \)

ITE Operator

\( \text{ITE operator ite}(f, g, h) = fg + f' \cdot h \) can implement any two variable logic function.

There are 16 such functions corresponding to all subsets of vertices of \( B^2 \):

<table>
<thead>
<tr>
<th>Subset</th>
<th>Expression</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>AND(f, g)</td>
<td>ite(f, g, 0)</td>
</tr>
<tr>
<td>0010</td>
<td>f \cdot g</td>
<td>ite(f, g', 0)</td>
</tr>
<tr>
<td>0011</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>0100</td>
<td>f &lt; g</td>
<td>f</td>
</tr>
<tr>
<td>0101</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>0110</td>
<td>XOR(f, g)</td>
<td>ite(f, g', 0)</td>
</tr>
<tr>
<td>0111</td>
<td>f + g</td>
<td>ite(f, g, 0)</td>
</tr>
<tr>
<td>1000</td>
<td>NOR(f, g)</td>
<td>ite(f, 0, g)</td>
</tr>
<tr>
<td>1001</td>
<td>f \oplus g</td>
<td>ite(f, g, 0)</td>
</tr>
<tr>
<td>1010</td>
<td>NOT(g)</td>
<td>g</td>
</tr>
<tr>
<td>1011</td>
<td>f \leq g</td>
<td>f \oplus g</td>
</tr>
<tr>
<td>1100</td>
<td>NOT(f)</td>
<td>ite(f, 0, 1)</td>
</tr>
<tr>
<td>1101</td>
<td>f \leq g</td>
<td>ite(f, g, 1)</td>
</tr>
<tr>
<td>1110</td>
<td>NAND(f, g)</td>
<td>ite(f, 0, g)</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example:

\[ I = \text{ite}(F, G, H) = (a, \text{ite}(F_{a}, G_{a}, H_{a}), \text{ite}(F_{a'}, G_{a'}, H_{a'})) \]

\[ = \text{ite}(a, \text{ite}(1, C, J), \text{ite}(B_{a}, 0, D)) \]

Check:

\[ F = a + b \]

\[ G = ac \]

\[ H = b + d \]

\( \text{ite}(F, G, H) = (a + b)(ac) + a'b'(b + d) = ac + a'b'd \)
ITE Operator

The ITE (If-Then-Else) operator is a fundamental concept in the BDD (Binary Decision Diagram) package. It is used to represent conditional statements in a compact form.

### Algorithm

```c
struct vertex *apply_ite(struct vertex *F, *G, *H, int i)
{
    char x;
    struct vertex *π, *λ;
    if (F == v1)
        return G;
    else if (F == v0)
        return H;
    else if (G == v1 && H == v0)
        return F;
    else {
        i ← π(i);
        π ← apply_ite(Fx, Gx, Hx, i + 1);
        λ ← apply_ite(F, *G, *H, i + 1);
        if (π = λ)
            return π;
        else
            return old_or_new(x, π, λ);
    }
}
```

### Example

- **G** = `ite(G, 0, 1)`
- **H** = `F \oplus G`
- **F** = `ite(F, G, G)`

### 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.