Introduction to Electronic Design Automation

Jie-Hong Roland Jiang
江介宏
Department of Electrical Engineering
National Taiwan University
Spring 2013

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
  - Layout vs. schematic
  - Design rule check

**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 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
```

R.O.B.D.D.

Ordered BDD (OBDD)

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

```
f = \overline{x}_1 \overline{x}_2 \overline{x}_3 + \overline{x}_1 x_2 \overline{x}_3 + \overline{x}_1 \overline{x}_2 x_3 + x_1 \overline{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

- 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
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

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

Truth table

<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>

OBDD

Example (cont’d)

<table>
<thead>
<tr>
<th>ROBDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>

Canonicity

- 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:
    \[ f = \prod_{j=1}^{k} x_{j-1} \oplus x_j \]

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
    \( (e.g., x_1 x_2 + x_2 x_3 + x_3 x_4) \)
    \( x_1 < x_2 < x_3 < x_4 \) 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

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

struct vertex *find_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 \( x \)

- \( h = x \cdot h_{x=1} + x' \cdot h_{x=0} \)
- \( h = x \cdot (f <op> g)_{x=1} + x' \cdot (f <op> g)_{x=0} \)

(f \( <op> \) g)\(_{x=1} \) for \( <op> = \) AND, OR, NAND, NOR.

Example

Existential quantification

Let \( \exists x \_ [f(x_1,y_1,\ldots,y_n)] = g(y_1,\ldots,y_n) \). 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.

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

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

Example:

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

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

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

ITE Operator

- 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 &lt; g</td>
<td>ite(f, 0, g)</td>
</tr>
<tr>
<td>0100</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>0101</td>
<td>f ≥ g</td>
<td>ite(f, g, 0)</td>
</tr>
<tr>
<td>1000</td>
<td>XOR(f, g)</td>
<td>ite(f, g, 0)</td>
</tr>
<tr>
<td>1010</td>
<td>NOT(g)</td>
<td>g</td>
</tr>
<tr>
<td>1100</td>
<td>NAND(f, g)</td>
<td>ite(f, 0, g)</td>
</tr>
<tr>
<td>1110</td>
<td>f ≥ g</td>
<td>ite(f, 0, g)</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Recursive Formulation of ITE

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

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

ITE Operator

- Example

\[ I = \text{ite}(F, G, H) \]
  \[ = \text{ite}(a, \text{ite}(F_a, G_a, H_a), \text{ite}(F_{\overline{a}}, G_{\overline{a}}, H_{\overline{a}})) \]
  \[ = \text{ite}(a, \text{ite}(1, C, J), \text{ite}(B, 0, D)) \]
  \[ = \text{ite}(a, C, \text{ite}(b, \text{ite}(1, 0, 1), \text{ite}(0, 0, D))) \]
  \[ = \text{ite}(a, C, \text{ite}(b, 0, D)) \]
  \[ = \text{ite}(a, C, J) \]

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

```c
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 {
        i = π(i);
        η = apply_ite(Fx, Gx, Hx, i + 1);
        λ = apply_ite(Fη, Gη, Hη, i + 1);
        if (η == λ)
            return η;
        else
            return old_Cor_new(x, η, λ);
    }
}
```

- ITE algorithm processes the variables in the order used in the BDD package
- \( π(i) \) gives the \( i \)th variable from the top; \( π^*(x) \) 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
  \[ F_{x} = η(F) \quad \text{if} \quad π^*(φ(F)) = i \]
- \( F_{x} \) can be calculated similarly

- The time complexity of the algorithm is \( O(|F| \cdot |G| \cdot |H|) \)

ITE Operator

- Example

```
G = ite(G, 0, 1)
```

- Example (cont’d)

```
\[ \overline{G} = \text{ite}(G, 0, 1) \]
```

- 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
Data Type Conversion

Truth Table
- recursive Shannon expansion
- enumerate each root-to-1 path (each representing a product term)

BDD
- translation using MUXes
- recursive Shannon expansion
- enumerate each root-to-1 path (each representing a product term)

Logic Netlist
- incremental construction from PIs to POs

Boolean Formula

Formula to BDD

Given a Boolean formula
\[ f = x_3 \cdot (x_1 + x_2) \]
Use variable order: \(x_1 < x_2 < x_3\)

Shannon expansion on \(x_1\)
\[ f = x_1 \cdot f_{x_1=1} + x_1' \cdot f_{x_1=0} \]
\[ = x_1 \cdot x_3 + x_1' \cdot x_2 \cdot x_3 \]

Shannon expansion on \(x_2\) and \(x_3\)
\[ f = x_1 \cdot x_3 + x_1' \cdot (x_2 \cdot x_3 + x_2' \cdot 0) \]

Perform reduction on the resulting BDD to a canonical form

Netlist to BDD

Decide a good variable ordering

Topologically sort the signals (from PI's towards PO's)

more signal's OBDD to build?

no

select the next signal based on the topological order

yes

construct the selected signal's OBDD using its direct fanins OBDD's

Netlist to BDD

Example

Topological order: \(\{x_1, x_2, x_3, z_1, z_2\}\)
variable order: \(x_1 < x_2 < x_3\)

\(\text{OBDD}(z_2) = \text{OBDD}(x_3) \cdot \text{OBDD}(z_1)\)
BDD to Netlist

- **MUX-based translation**
  - replace each decision node by a MUX
  - replace 0-terminal by GND, and 1-terminal by VDD
  - reverse the direction of every edge
  - specify the root node as the output node

```
+-----------------+-----------------+-----------------+-----------------+
| 0   | 1   | 0   | 1   |
| x1  | x2  | x3  | MUX |
+-----------------+-----------------+-----------------+-----------------+
| 0   | 1   | 0   | 1   |
| GND | MUX | MUX | GND |
+-----------------+-----------------+-----------------+-----------------+
| 0   | 1   | 0   | 1   |
```

output function

BDD Features

- **Strengths**
  - ROBDD is a **compact representation** for many Boolean functions
  - ROBDD is **canonical**, given a fixed variable ordering
  - Many Boolean operations are of **polynomial time complexity** in the input BDD sizes

- **Weaknesses**
  - In the worst case, the size of a BDD is $O(2^n)$ for n-input Boolean functions

BDD Applications

- **Boolean function verification**
  - Compare a specification $f$ to an implementation $g$, assuming their ROBDDs are $F$ and $G$, respectively.
  - For fully specified functions $f$ and $g$, the verification is trivial (pointer comparison) because of the strong canonicity of the ROBDD
    - Strong canonicity: the representations of identical functions are the same
  - For an incompletely specified function $I = (f, d, \neg (f+e))$ with onset $f$, dc-set $d$, and offset $-(f+e)$, A completely specified function $g$ correctly implements $I$ if $(d + f g + \neg f \neg g)$ is a tautology, that is, $f \Rightarrow g = (f+e)$

- **Satisfiability checking**
  - A Boolean function $f$ is **satisfiable** if there exists an input assignment for which $f$ evaluates to ‘1’
  - Any Boolean function whose ROBDD is not equal to ‘0’ is satisfiable

- **Min-cost satisfiability**
  - Suppose that choosing a Boolean variable $x_i$ to be ‘1’ costs $c_i$. Then, the **minimum-cost satisfiability** problem asks to minimize $\sum c_i u_i(x_i)$ where $u_i(x_i) = 1$ when $x_i = '1'$ and $u_i(x_i) = 0$ when $x_i = '0'$.
  - Solving minimum-cost satisfiability amounts to computing the shortest path in an ROBDD with weights: $w(v, \eta(v)) = c_i \epsilon v, \lambda(v) = 0$, variable $x_i = \delta(v)$, which can be solved in linear time

- **Combinatorial optimization**
  - Many combinatorial optimization problems can also be formulated in terms of the satisfiability problem
  - 0-1 integer linear programming can be formulated as a minimum-cost satisfiability problem although the translation may not be efficient
  - E.g., the constraint $x_1 + x_2 + x_3 + x_4 = 3$ can be written as $(x_1 + x_2)(x_1 + x_3)(x_1 + x_4)(x_2 + x_3)(x_2 + x_4)(x_3 + x_4)(x_3 + x_4)(x_4 + x_4)(-x_1 + -x_2 + -x_3 + -x_4)$
Outline

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

SAT Solving

- SAT problem: Given a Boolean formula $\varphi$ in CNF, find an input assignment such that $\varphi$ valuates to true
- SAT solving is a decision procedure over CNFs
  - Example
    $$\varphi = (a+b'+c)(a'+b+c)(a+b'+c')(a+b+c)$$
    is SAT (e.g. under $a=1$, $b=1$, $c=0$)
- SAT in CNF (POS) $\Leftrightarrow$ Tautology in DNF (SOP)
  - How about Tautology in CNF and SAT in DNF?

Circuit to CNF

- Given a circuit, suppose we would like to know if some signal is always zero. This can be formulated as a SAT problem if we can convert the circuit to an CNF.
  - an AIG
  - Is output always 0?

Naive conversion of circuit to CNF:
  - Multiply out expressions of circuit until two level structure
  - Example: $y = x_1 \oplus x_2 \oplus ... \oplus x_n$ (Parity function)
    - circuit size is linear in the number of variables
    - generated chess-board Karnaugh map
    - CNF (or DNF) formula has $2^{n-1}$ terms (exponential in #vars)

Better approach:
  - Introduce one variable per circuit vertex
  - Formulate the circuit as a conjunction of constraints imposed on the vertex values by the gates
  - Uses more variables but size of formula is linear in the size of the circuit
Circuit to CNF

Example

- Single gate:
  \[ \text{AND} \quad a \quad b \rightarrow c \quad (\neg a + \neg b + c)(a + \neg c)(b + \neg c) \]

- Circuit of connected gates:

\[ \sim 1 \sim 2 \sim 3 \sim 4 \sim 5 \sim 6 \sim 7 \sim 0 \]

Is output always 0?

Justify to "1"

\[ \sim 1 + 2 + 4)(1 + \sim 4)(\sim 2 + \sim 4) \]
\[ \sim 2 + \sim 3 + 5)(2 + \sim 5)(3 + \sim 5) \]
\[ (2 + \sim 3 + 6)(\sim 2 + \sim 6)(3 + \sim 6) \]
\[ (\sim 4 + \sim 5 + 7)(\sim 4 + \sim 7)(5 + \sim 7) \]
\[ (5 + 6 + 8)(\sim 5 + \sim 8)(\sim 6 + \sim 8) \]
\[ (\sim 7 + 8 + 9)(\sim 7 + \sim 9)(\sim 8 + \sim 9) \]
\[ (9) \]

DPLL-Style SAT Solving

SAT(clause set S, literal v)
1. S := S_v //cofactor each clause of S w.r.t. v
2. If no clauses in S, return T
3. If a clause in S is empty (FALSE), return F
4. If S has a unit clause with literal u, then return SAT(S, u) //implication
5. Choose a variable x with value not yet assigned
6. If SAT(S, x), return T
7. If SAT(S, \neg x), return T
8. Return F

SAT Solving with Case Splitting

Example

1. \[ (a + \neg b + c) \]
2. \[ (a + b + \neg c) \]
3. \[ (\neg a + b + \neg c) \]
4. \[ (a + c + d) \]
5. \[ (\neg a + c + \neg d) \]
6. \[ (\neg a + c + \neg d) \]
7. \[ (\neg b + \neg c + \neg d) \]
8. \[ (\neg b + \neg c + \neg d) \]

Source: Karim A. Salehiah, Univ. of Michigan
**SAT Solving with Implication**

- Implication in a CNF formula are caused by unit clauses
  - A unit clause is a clause in which all literals except one are assigned (to be false)
    - The value of the unassigned variable is implied
      - Example
        
        \[(a + \neg b + c)\]
        
        \[a = 0, b = 1 \Rightarrow c = 1\]

**Implications in CNF**

- Example
  - \[\begin{align*}
    a & \quad \text{AND} \\
    b & \quad c
  \end{align*}\]
    \[\neg a + \neg b + c (a + c) (b + \neg c)\]

  - Implications:
    
    \[\begin{align*}
      & (-a + -b + c) \\
      & (a + -c) \\
      & (b + -c)
    \end{align*}\]

Source: Karem A. Sakallah, Univ. of Michigan

---

**SAT Solving with Implication**

- Example
  - 1: \((a + b + c)\)
  - 2: \((a + b + \neg c)\)
  - 3: \((\neg a + b + \neg c)\)
  - 4: \((a + c + d)\)
  - 5: \((\neg a + c + d)\)
  - 6: \((\neg a + c + \neg d)\)
  - 7: \((\neg b + \neg c + \neg d)\)
  - 8: \((\neg b + \neg c + d)\)

Source: Karem A. Sakallah, Univ. of Michigan

---

**SAT Solving with Learning**

- Example
  - 1: \((a + b + c)\)
  - 2: \((a + b + \neg c)\)
  - 3: \((\neg a + b + \neg c)\)
  - 4: \((a + c + d)\)
  - 5: \((\neg a + c + d)\)
  - 6: \((\neg a + c + \neg d)\)
  - 7: \((\neg b + \neg c + \neg d)\)
  - 8: \((\neg b + \neg c + d)\)
  - 9: \((\neg b + \neg c)\)
  - 10: \((\neg b + \neg c + \neg d)\)
  - 11: \((\neg b + \neg c + d)\)

Source: Karem A. Sakallah, Univ. of Michigan
Implementation Issues

- Track sensitivity of clauses for changes (two-literall-watch scheme)
  - clause with all literals but one assigned → implication
  - clause with all literals but two assigned → sensitive to a change of either literal
  - all other clauses are insensitive and need not be observed

- Learning:
  - learned implications are added to the CNF formula as additional clauses
    - limit the size of the clause
    - limit the "lifetime" of a learned clause, will be removed after some time

Quantification over CNF and DNF

- Recall a quantified Boolean formula (QBF) is $Q_1 x_1, Q_2 x_2, \ldots, Q_n x_n \cdot \varphi$
  where $Q_i$ is either a existential ($\exists$) or universal quantifier ($\forall$), $x_i$ is a Boolean variable, and $\varphi$ is a Boolean formula.

- Existential (respectively universal) quantification over DNF (respectively CNF) is easy
  - One approach to quantifier elimination is by back-and-forth CNF-DNF conversion!

- Solving QBFs with QBF-solvers

Outline

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

Equivalence Checking in Microprocessor Design
Equivalence Checking in ASIC Design

- RTL Specification
- Cell-Based Synthesis
- Standard Cell Implementation
- Engineering Changes (ECOs)
- Final Implementation

Equivalence Checking

- Property Checking
- Equivalence Checking

Outline

- Introduction
- Boolean reasoning engines
- Equivalence checking
  - Combinational equivalence checking
  - Sequential equivalence checking
- Property checking

History of Equivalence Checking

- SAS (IBM 1978 - 1994):
  - standard equivalence checking tool running on mainframes
  - based on the DBA algorithm (“BDDs in time”)
  - verified manual cell-based designs against RTL spec
  - handling of entire processor designs
    - application of “proper cutpoints”
    - application of synthesis routines to make circuits structurally similar
    - special hacks for hard problems

- Verity (IBM 1992 - today):
  - originally developed for switch-level designs
  - today IBMs standard EC tool for any combination of switch-, gate-, and RTL designs
History of Equivalence Checking

- Chrysalis (1994 - Avanti - now Synopsys):
  - based on ATPG technology and cutpoint exploitation
  - very weak if many cutpoints present
  - did not adopt BDDs for a long time

- Formality (1997 - Synopsys)
  - multi-engine technology including strong structural matching techniques

- Verplex (1998 - now Cadence)
  - strong multi-engine based tool
  - heavy SAT-based
  - very fast front-end

Combinational EC

- Given two combinational circuits $C_1$ and $C_2$, are their outputs equivalent under any possible input assignment?

\[ x \quad C_1 \quad y_1 \quad ? \quad C_2 \quad y_2 \]

Miter for Combinational EC

- Two combinational circuits $C_1$ and $C_2$ are equivalent if and only if the output of their "miter structure" always produces constant 0

Approaches to Combinational EC

- Basic methods:
  - random simulation
    - good at identifying inequivalent signals
  - BDD-based methods
  - structural SAT-based methods
BDD-based Combinational EC

Procedure
1. Construct the ROBDDs $F_1$ and $F_2$ for circuits $C_1$ and $C_2$, respectively
   - Variable orderings of $F_1$ and $F_2$ should be the same
2. Let $G = F_1 \oplus F_2$. If $G = 0$, $C_1$ and $C_2$ are equivalent; otherwise, they are inequivalent
   - No false negative or false positive
     - False negative: circuits are equivalent; however, verifier fails to tell
     - False positive: circuits are inequivalent; however, verifier says otherwise

SAT-based Combinational EC

Procedure
1. Convert the miter structure into a CNF
2. Perform SAT solving to verify if the output variable cannot be evaluated to true under every input assignment (i.e. UNSAT)

Combinational EC

- Pure BDD and plain SAT solving cannot handle all logic cones
  - BDDs can be built for about 80% of the cones of high-speed designs and less for complex ASICs
  - Plain SAT blows up in CPU time on a miter structure
- Contemporary method highly exploit structural similarities between two circuits to be compared

Memory statistics of BDD-based EC on a PowerPC processor design

- 95% of all circuits
Combinational EC

- Runtime statistics of BDD-based EC on a PowerPC processor design

Necessity of Structure Similarity

- Pure BDDs are incapable of verifying equivalence of large circuits
  - Even more so for arithmetic circuits (e.g. BDDs blow up in representing multipliers)
- Identifying structure similarity helps simplify verification tasks
  - E.g. structure hashing in AIGs

Structure and Verification

- Structure-independent techniques
  - Exhaustive simulation
  - Decision diagrams
- Structure-dependent techniques
  - Graph hashing
  - SAT based cutpoint identification
Summary

- Combinational EC is considered to be solvable in most industrial circuits (w/ multi-million gates)
  - Computational efforts scale almost linearly with the design size
  - Existence of structural similarities
    - Logic transformations preserve similarities to some extent
    - Hybrid engine of BDD, SAT, AIG, simulation, etc.
  - Cutpoint identification

- Unsolved for arithmetic circuits
  - Absence of structural similarities
    - Commutativity ruins internal similarities
  - Word- vs. bit-level verification

Outline

- Introduction
- Boolean reasoning engines
- Equivalence checking
  - Combinational equivalence checking
  - Sequential equivalence checking
- Property checking

Sequential EC

- Given two sequential circuits (and thus FSMs), do they produce the same output sequence under any possible input sequence?

Miter for Sequential EC

- Two FSMs $M_1$ and $M_2$ are equivalent if and only if the output of their product machine always produces constant 0.

\[
x \xrightarrow{\lambda_1, \delta_1} y_1 \xrightarrow{?} \equiv \xrightarrow{?} 0
\]

\[
x \xrightarrow{\lambda_2, \delta_2} y_2 \xrightarrow{?} \equiv 0
\]
Product Machine

The product FSM $M_{1\times2}$ of FSMs $M_1 = (Q_1, I_1, \Sigma, \Omega, \delta_1, \lambda_1)$ and $M_2 = (Q_2, I_2, \Sigma, \Omega, \delta_2, \lambda_2)$ is a six-tuple $(Q_{1\times2}, I_{1\times2}, \Sigma, \Omega, \delta_{1\times2}, \lambda_{1\times2})$, where

- State space $Q_{1\times2} = Q_1 \times Q_2$
- Initial state set $I_{1\times2} = I_1 \times I_2$
- Input alphabet $\Sigma$
- Output alphabet $\{0, 1\}$
- Transition function $\delta_{1\times2} = (\delta_1, \delta_2)$
- Output function $\lambda_{1\times2} = (\lambda_1 \oplus \lambda_2)$

Sequential EC

Approaches for combinational EC do not work for sequential EC because two equivalent FSMs need not have the same transition and output functions

- False negatives may result from applying combinational EC on sequential circuits

One solution to sequential EC is by reachability analysis

Two FSMs $M_1$ and $M_2$ are equivalent if and only if the output of their product FSM $M_{1\times2}$ is constant 0 under all input assignments and all reachable states of $M_{1\times2}$

Reachability Analysis

Given an FSM $M = (Q, I, \Sigma, \Omega, \delta, \lambda)$, which states are reachable from the initial state set $I$?

Symbolic Reachability Analysis

Reachability analysis can be performed either explicitly (over a state transition graph) or implicitly (over transition functions or a transition relation)

- Implicit reachability analysis is also called symbolic reachability analysis (often using BDDs and more recently SAT)

Image computation is the core computation in symbolic reachability analysis
Reachability Onion Ring

Computing Reachable States

- **Input**: Sequential system represented by a transition relation and an initial state (or a set of initial states)
  - Transition functions can be converted into a transition relation

- **Computation**: Image computation using Boolean operations on characteristic functions (representing state sets)

- **Output**: A characteristic function representing the set of reachable states

Relation

- **Definition.** Relation $R \subseteq X \times Y$ is a subset of the Cartesian product of two sets $X$ and $Y$. If $(x, y) \in R$, then we alternatively write “$x \mathbin{R} y$” meaning $x$ is related to $y$ by $R$.

Charlotte Function

- **Relation** $R \subseteq X \times Y$ can be represented by a characteristic function: a Boolean function $F_R(x, y)$ taking value 1 for those $(x, y) \in R$ and 0 otherwise.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

9 nodes

Courtesy of A. Mishchenko
Transition Relation

Definition. A transition relation \( T \) of an FSM \( M = (Q, I, \Sigma, \Omega, \delta, \lambda) \) is a relation \( T \subseteq (\Sigma \times Q) \times Q \) such that \( T(\sigma, q_1, q_2) = 1 \) iff there is a transition from \( q_1 \) to \( q_2 \) under input \( \sigma \).

\( \delta: (\Sigma \times Q) \rightarrow Q \)
\( T: (\Sigma \times Q) \times Q \rightarrow \{0,1\} \)

Assume \( \delta = (\delta_1, \ldots, \delta_n) \). Then

\[
T(x, s, s') = (s_1' = \delta_1(x, s)) \land (s_2' = \delta_2(x, s)) \land \cdots \land (s_n' = \delta_n(x, s))
\]

where \( x, s, s' \) are primary-input, current-state, and next-state variables, respectively.

Quantified Transition Relation

Definition

Let \( M = (Q, I, \Sigma, \Omega, \delta, \lambda) \) be an FSM

\( T_\exists \) quantified transition relation

\[
T_\exists(x, s, s') = \exists x.(s_1' = \delta_1(x, s)) \land (s_2' = \delta_2(x, s)) \land \cdots \land (s_n' = \delta_n(x, s))
\]

\( (p, q) \in T_\exists \) if there exists an input assignment bringing \( M \) from state \( p \) to state \( q \)

Only concerns about the reachability of the FSM’s transition graph.

Example

<table>
<thead>
<tr>
<th>x</th>
<th>CS</th>
<th>( s_1 \ s_2 )</th>
<th>NS</th>
<th>( s_1' \ s_2' )</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>00</td>
<td>B</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>0,1</td>
<td>A</td>
<td>00</td>
<td>A</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>B</td>
<td>10</td>
<td>B</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>10</td>
<td>A</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>C</td>
<td>01</td>
<td>B</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>01</td>
<td>A</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>other</td>
<td></td>
<td></td>
<td>A</td>
<td>00</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

Transition Relation

Transition Relation

Example
Image Computation

- Given a mapping of one Boolean space (input space) into another Boolean space (output space)
  - For a set of minterms (care set) in the input space
    - The image is the set of related minterms from the output space
  - For a set of minterms in the output space
    - The pre-image is the set of related minterms in the input space

Example

Input space  Output space

\( \text{abc} \)

\( \text{Care set} \)

\( \text{Image} \)

\( \text{Output space} \)

\( \text{000} \)

\( \text{001} \)

\( \text{010} \)

\( \text{011} \)

\( \text{100} \)

\( \text{101} \)

\( \text{110} \)

\( \text{111} \)

\( \text{a} \)

\( \text{b} \)

\( \text{c} \)

\( \text{x} \)

\( \text{y} \)

Image Computation

- Example

\[ \text{Image}(C(x), T(x,y)) = \exists x\left[ C(x) \land T(x,y) \right] \]

- Implicit methods by far outperform explicit ones
  - Successfully computing images with more than \( 2^{100} \) minterms in the input/output spaces

- Operations \( \land \) and \( \exists \) are basic Boolean manipulations and are implemented in BDD packages
  - To avoid large intermediate results (during and after the product computation), BDD \( \land \)-\( \exists \) operation performs product and quantification in one pass over the BDD

Symbolic Image Computation

- Definition. Let \( F: B^m \times B^n \) be a projection and \( C \) be a set of minterms in \( B^m \). Then the image of \( C \) is the set \( \text{Img}(C, F) = \{ w \in B^n | (v, w) \in F \land v \in C \} \) in \( B^n \).

- Characteristic function
  - for reachable next-state computation

\[ N'(s') = \text{Img}(R(s), T(s, s')) = \exists s.(R'(s) \land T(s, s')) = \exists s.(R'(s) \land (\exists x \prod_i (s_i' \equiv \delta_i(x, s))))) \]
Symbolic Pre-Image Computation

- Definition. Let $F: B^m \times B^n$ be a projection and $C$ be a set of minterms in $B^m$. Then the pre-image of $C$ is the set $\text{PreImg}(C, F) = \{ v \in B^m \mid (v, w) \in F \text{ and } w \in C \}$ in $B^n$.

- Characteristic Function
  - for reachable previous-state computation
    
    $N_i(s) = \text{PreImg}(R_i(s'), T_i(s, s'))$
    
    $= \exists s'. (R_i(s') \land T_i(s, s'))$
    
    $= \exists s'. (R_i(s') \land (\exists x_i (s_i = \delta(x, s))))$

Reachability Analysis

- Forward Reachability

$$\text{ForwardReachability}(\text{Transition Relation } T, \text{Initial State } I )$$

1. $i := 0$
2. $R^i := I$
3. repeat
   1. $R_{new} = \text{Image}(R^i, T)$;
   2. $i := i + 1$
   3. $R^i := R^{i-1} \lor R_{new}$
4. until $R^i = R^{i-1}$
5. return $R^i$

- The procedures can be realized using BDD package.

- Backward reachability analysis can be done in a similar manner with pre-image computation and starting from final states to see if they can be reached from initial states.

Sequential Equivalence Checking

- Let $R(s)$ be the characteristic function of the reachable state set of the product FSM $M_{1\times2}$ obtained from forward reachability analysis. Then FSMs $M_1$ and $M_2$ are equivalent if and only if
  
  $R(s) \rightarrow (\lambda_{1\times2}(x,s)=0)$
  
  is valid for all valuations on input variables $x$ and state variables $s$.

- This can be checked in constant time for BDD

- Example
  - Are $M_1$ and $M_2$ equivalent?
Sequential Equivalence Checking

- **Example (cont’d)**
  - Product FSM of M1 and M2

![Diagram of FSMs](image)

Sequential Equivalence Checking

- **Example (cont’d)**
  - Forward reachability analysis
    \[
    \text{Img}(C,T) = [\exists \bar{x}, \bar{s}. T(\bar{x}, \bar{s}, \bar{s}') \land C(\bar{s}')]
    \]

![Diagram of reachability analysis](image)

Sequential Equivalence Checking

- **Example (cont’d)**
  - Backward reachability analysis
    \[
    \text{PreImg}(C,T) = [\exists \bar{x}, \bar{s}. T(\bar{x}, \bar{s}, \bar{s}') \land C(\bar{s}')]
    \]

![Diagram of backward reachability analysis](image)

Remarks on Sequential EC

- **Industrial equivalence checkers almost exclusively use an combinational EC paradigm even for sequential EC**
  - Sequential EC is too complex and can only be applied to design with a few hundred state bits
  - Structure similarity should be identified to simplify sequential EC

- **Besides sequential equivalence checking, reachability analysis is useful in sequential circuit optimization**
  - In sequential optimization, *unreachable states* can be used as *sequential don’t cares* to optimize a sequential circuit
Outline

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

Model Checking

- A specific model-checking problem is defined by
  \[ M \models \phi \]
  - “implementation” (system model)
  - “specification” (system property)
  - “satisfies”, “implements”, “refines” (satisfaction relation)

Model Checking

- \( M \models \phi \)
  - Check if system model \( M \) satisfies a system property \( \phi \)
  - System model \( M \) is described with a state transition system
    - finite state or infinite state
  - Temporal property \( \phi \) can be described with three orthogonal choices:
    1. operational vs. declarative: automata vs. logic
    2. may vs. must: branching vs. linear time
    3. prohibiting bad vs. desiring good behavior: safety vs. liveness
  - Different choices lead to different model checking problems.

Property Checking

- Safety property: Something “bad” will never happen
  - Safety property violation always has a finite witness
    - if something bad happens on an infinite run, then it happens already on some finite prefix
  - Example
    - Two processes cannot be in their critical sections simultaneously

- Liveness property: Something “good” will eventually happen
  - Liveness property violation never has a finite witness
    - no matter what happens along a finite run, something good could still happen later
  - Example
    - Whenever process P1 wants to enter the critical section, provided process P2 never stays in the critical section forever, P1 gets to enter eventually

For finite state systems, liveness can be converted to safety!
Safety Property Checking

- Safety property checking can be formulated as a reachability problem
  - Are bad states reachable from good states?
- Sequential equivalence checking can be considered as one kind of safety property checking
  - \( M \) : product machine
  - \( \phi \) : all states reachable from initial states has output 0

Model Checking

- Data structure evolution
  - State graph (late 70s-80s)
    - Problem size \( \sim 10^4 \) states
  - BDD (late 80s-90s)
    - Problem size \( \sim 10^{20} \) states
    - Critical resource: memory
  - SAT (late 90s-)
    - GRASP, SATO, chaff, berkmin
    - Problem size \( \sim 10^{100} \) (?) states
    - Critical resource: CPU time

Remarks on Model Checking

- Model checking is a very rich subject developed since early 1980's
- It is a variation of mathematical logic and is concerned with automatic temporal reasoning
- Reference
  M. Clarke, O. Grumberg, and D. Peled.