Data Type Conversion

Truth Table
- Enumerate each root-to-1 path (each representing a product term)

BDD
- Recursive Shannon expansion
- Enumerate each root-to-1 path (each representing a product term)

Logic Netlist
- Incremental construction from PIs to POs

Boolean Formula
- Translation using MUXes

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

1. Decide a good variable ordering
2. Topologically sort the signals (from PI's towards PO's)
3. Select the next signal based on the topological order
4. Construct the selected signal's OBDD using its direct fanins' OBDD's

Example

Topological order: \{x_1, x_2, x_3, z_1, z_2\}
Variable order: x_1 < x_2 < x_3

\[
OBDD(z_2) = OBDD(x_3) \cdot OBDD(z_1)
\]
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

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 $I = (f, d, \neg(f+d))$ with onset $f$, dc-set $d$, and offset $\neg(f+d)$. A completely specified function $g$ correctly implements $I$ if $(d + f \cdot g + \neg f \cdot \neg g)$ is a tautology, that is, $f \Rightarrow g \Rightarrow (f+d)$

- **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

---

BDD Applications

- **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: $\sum_i c_i \cdot u_i(x_i)$
  - where $\mu(x_i) = 1$ when $x_i = '1'$ and $\mu(x_i) = 0$ when $x_i = '0'$. Solving minimum-cost satisfiability amounts to computing the shortest path in an ROBDD with weights: $w(v, \eta(v)) = c_i$, $w(v, \lambda(v)) = 0$, variable $x_i = \phi(v)$, 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 $(x_1+x_2)(x_1+x_3)(x_1+x_4)(x_2+x_3)(x_2+x_4)(x_3+x_4)(\neg x_1+\neg x_2+\neg x_3+\neg x_4)$
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) $\iff$ Tautology in DNF (SOP)

  How about Tautology in CNF and SAT in DNF?
SAT Solving

- 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 covert the circuit to an CNF.

\[ \text{Is output always 0?} \]

an AIG

Circuit to CNF

- Naive conversion of circuit to CNF:
  - Multiply out expressions of circuit until two level structure
  - Example: \( y = x_1 \oplus x_2 \oplus x_2 \oplus \ldots \oplus x_n \) (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:

\[
\neg a \land \neg b \lor c \\
\neg a \land \neg c \land \neg b
\]

- Circuit of connected gates:

\[
\neg 1 \lor 2 \lor 4 \land 1 \lor \neg 4 \land \neg 2 \lor \neg 4 \land \\
\neg 2 \lor \neg 3 \lor 5 \land 2 \lor \neg 5 \land 3 \lor \neg 5 \land \\
\neg 2 \lor \neg 3 \lor 6 \land \neg 2 \lor \neg 6 \land 3 \lor \neg 6 \land \\
\neg 4 \lor \neg 5 \lor 7 \land 4 \lor \neg 7 \land 5 \lor \neg 7 \land \\
\neg 5 \lor \neg 6 \lor 8 \land \neg 5 \lor \neg 8 \land \neg 6 \lor \neg 8 \land \\
\neg 7 \lor \neg 8 \lor 9 \land \neg 7 \lor \neg 9 \land \neg 8 \lor \neg 9 \land \\
9
\]

Is output always 0?

Justify to “1”

Circuit to CNF conversion

- can be done in linear size (with respect to the circuit size) if intermediate variables can be introduced

- may grow exponentially in size if no intermediate variables are allowed
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. $(a + b + c)$
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: Karen A. Sakallah, 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

\[
\begin{align*}
\text{Implications:} & \quad (\neg a + \neg b + c) \quad (a + \neg c) \quad (b + \neg c) \\
(\neg a + \neg b + c) & \quad 1 \quad x \\
(\neg a + \neg b + c) & \quad 1 \quad x \\
(\neg a + \neg b + c) & \quad 1 \quad x \\
(\neg a + \neg b + c) & \quad x \quad 0 \\
(\neg a + \neg b + c) & \quad x \quad 0 \\
(\neg a + \neg b + c) & \quad x \quad 0 \\
(a + \neg c) & \quad x \quad 0 \\
(a + \neg c) & \quad x \quad 0 \\
(a + \neg c) & \quad x \quad 0 \\
(b + \neg c) & \quad x \quad 0 \\
(b + \neg c) & \quad x \quad 0 \\
(b + \neg c) & \quad x \quad 0 \\
\end{align*}
\]
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: \((\neg b + \neg c + d)\)

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

- Track sensitivity of clauses for changes (two-literal-watch scheme)
  - clause with all literals but one assigned $\rightarrow$ implication
  - clause with all literals but two assigned $\rightarrow$ 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

Architectural Specification (informal) → Property Checking
RTL Specification (Verilog, VHDL) → Cycle Simulation → Test Programs
Circuit Implementation (Schematic) → Equivalence Checking
Layout Implementation (GDS II) → Circuit Simulation
Equivalence Checking in ASIC Design

Equivalence checking is one of the most important problem in design verification
- It ensures logic transformation process (e.g. two-level, multi-level logic minimization, retiming and resynthesis, etc.) does not introduce errors

Two types of equivalence checking
- Combinational equivalence checking
  - Check if two combinational circuits are equivalent
- Sequential equivalence checking
  - Check if two sequential circuits are equivalent
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 IBM's 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?

![Diagram](https://via.placeholder.com/150)
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 valuated 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

---

Combinational EC

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

![Graph showing memory statistics vs. circuit size](image-url)
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
Combinational EC

- Evidence of vast existence of structure similarities

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.
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 $M_{1\times2}$
- Need to know the set of 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.** A 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 “$x R y$” meaning $x$ is related to $y$ by $R$.

![Diagram of relations](image1.png)

<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>1</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>
</tbody>
</table>

**Characteristic 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.

![Diagram of characteristic function](image2.png)

<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>1</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
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 $\delta = (\delta_1, ..., \delta_k)$. Then

$$T(\bar{x}, \bar{s}, \bar{s}') = (s_1' = \delta_1(\bar{x}, \bar{s})) \land (s_2' = \delta_2(\bar{x}, \bar{s})) \land \cdots \land (s_k' = \delta_k(\bar{x}, \bar{s}))$$

$$= \prod_i (s_i' = \delta_i(\bar{x}, \bar{s}))$$

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

- **Quantified transition relation $T_\exists$**

$$T_\exists(\bar{s}, \bar{s}') = \exists \bar{x}. (s_1' = \delta_1(\bar{x}, \bar{s})) \land (s_2' = \delta_2(\bar{x}, \bar{s})) \land \cdots \land (s_k' = \delta_k(\bar{x}, \bar{s}))$$

$$= \exists \bar{x}. \prod_i (s_i' = \delta_i(\bar{x}, \bar{s}))$$

- $(p, q) \in T_\exists$ if there exists an input assignment bringing the $M$ from state $p$ to state $q$
- only concerns about the **reachability** of the FSM's transition graph
### Transition Relation

**Example**

![Transition Diagram](image)

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

*Courtesy of A. Mishchenko*
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: \( a, b, c \)

Output space: \( x, y \)

Image: \( 00, 01, 10, 11 \)

Care set: \( 000, 001, 010, 011, 100, 101, 110, 111 \)

 Courtesy of A. Mishchenko
Image Computation

- **Image Computation**
  
  \[ \text{Image}(C(x), T(x, y)) = \exists x [C(x) \land T(x, y)] \]

- 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 are implemented using BDDs

- To avoid large intermediate results (during and after the product computation), operation AND-EXIST is used, which 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 \(\text{Img}(C, F) = \{ w \in B^n \mid (v, w) \in F \text{ and } v \in C \}\) in \(B^n\).

- Characteristic function
  - for reachable next-state computation
  
  \[ N_i(s') = \text{Img}(R_i(s), T_\exists(s, s')) \]
  
  \[ = \exists s . (R_i(s) \land T_\exists(s, s')) \]
  
  \[ = \exists s . (R_i(s) \land (\exists x. \prod_{i} (s_i' \equiv \delta_i(x, s)))) \]