The problem of logic verification: show that two circuits implement the same boolean function.

- Problem: efficient representation of Boolean functions
  - DNF: linear for OR of n variables, exponential for XOR
  - Reed-Muller: linear for XOR of n variables, exponential for OR
- Problem: efficient application of Boolean operations
  - example:
    DNF ➞ Negation ➞ DNF, e.g.:
    ab + cd + ef + gh ➞ (ab + cd + ef + gh) ➞ ?
- Possible solution in many cases: binary decision diagrams (BDD’s)
2. Binary Decision Diagrams

2.1 BDD concepts

- Idea: decompose a function into two sub-functions which do not depend on a certain variable, e.g., $x_i$

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Idea: Decompose a function into two sub-functions which do not depend on a certain variable, e.g., $x_i$

- Apply Boole’s expansion theorem in a systematic way to all variables

- Represent result graphically

- The application of Boole’s expansion theorem to all variables leads to a decision tree.

Example: XOR in 3 variables

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f(a ⊕ b ⊕ c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- The application of Boole’s expansion theorem to all variables leads to a decision tree.

Example: XOR in 3 variables
2. Binary Decision Diagrams

2.1 BDD concepts

- The application of Boole’s expansion theorem to all variables leads to a decision tree. Example: XOR in 3 variables

\[
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

- Variable ordering: order in which Boole’s expansion theorem is applied

Order: a, b, c

- Concepts:
  - Nodes
  - Directed edges
  - Edge labelings
  - (Direct) successors of node a
  - Root node
  - Paths
  - Leafs or Terminal nodes
Decision trees are ordered (identical variable ordering on all paths) or free.

Example of a free decision tree:

A fully expanded decision tree has $2^n$ leaf nodes.

Example of a (free) decision tree which is not fully expanded:

Observation: there are identical sub-trees.
2. Binary Decision Diagrams

2.1 BDD concepts

- Merging identical sub-trees results in a decision-graph

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a•b•c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2. Binary Decision Diagrams

2.1 BDD concepts

- Shannon: A symbolic analysis of relay and switching circuits (1938)
  \[ \sum_{k=1}^{n} x_k \text{ for } n \text{ odd, } \left( \sum_{k=1}^{n} x_k \right)' \text{ for } n \text{ even} \]

Figure 28.

Some simple examples of BDD's:
2. Binary Decision Diagrams

2.1 BDD concepts

- AND-, OR-, XOR-operation in $n$ variables

- One path to the 1 leaf-node corresponds to a product – an implicant of the function. Example:

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a ⊕ b ⊕ c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
2 Problems:
- Given a binary decision diagram. How to derive the Boolean function represented by the BDD?
- Given a Boolean function. How to derive the BDD for it?
- First: BDD ➔ Boolean function

A node \( v \) of a BDD is characterized by a triple \((x, v_0, v_1)\), where \(v_0, v_1\) are the successors of \(v\).

The leaf nodes 0 and 1 represent the Boolean functions 0 and 1.

According to Boole's expansion theorem, the Boolean function \(bf(v)\) is associated with node \(v\) as follows (where \(\text{var}(v)\) is the variable of node \(v\)):

\[
bf(v) = \overline{\text{var}(v)} \cdot bf(v_0) + \text{var}(v) \cdot bf(v_1)
\]

The function associated with a node can be determined only if the functions associated with the successor nodes are known.

Example: which Boolean function is represented by the following BDD?

The "Bottom-up procedure": 1. Step
2. Binary Decision Diagrams

2.1 BDD concepts

⇒ “Bottom-up procedure”: 2. Step

\[ b_f(v) = \text{var}(v) \cdot b_f(v_0) + \text{var}(v) \cdot b_f(v_1) \]

\[ = b \cdot 0 + b \cdot 1 = b \]

Functions 0 and 1

⇒ “Bottom-up procedure”: 3. Step

\[ b_f(v) = \text{var}(v) \cdot b_f(v_0) + \text{var}(v) \cdot b_f(v_1) \]

\[ = a \cdot 0 + a \cdot b = a \cdot b \]

\[ = b \cdot 0 + b \cdot 1 = b \]

Functions 0 and 1

- There are many variants of binary decision diagrams
- Most useful and common: OBDD's (Ordered Binary Decision Diagrams, Bryant 1986)
- OBDD properties:
  - Ordered: The variables appear in a fixed ordering on all paths
    - Technically, an index (a positive integer) is associated with each variable \( \text{index(var(v))} \)
    - For each node \( v \) with successors \( v_0 \) and \( v_1 \) we have:
      \[ \text{index(var(v))} < \text{index(var(v_0))} \text{ and } \text{index(var(v))} < \text{index(var(v_1))} \]
OBDD properties (cont'd.):

- **Reduced**:
  - The function represented by one node is different from the functions of all other nodes
  - The two successors of each node are distinct

**Reduction example:**

Several representations of 1

Identical successors

Simplified representations exist, e.g.,

- 1-edges to the right, 0-edges to the left
- Edges to 0 omitted
- etc.

Example:

\[(a \oplus b) \cdot (c \oplus d) \cdot (e \oplus f)\]

or: 0 edges are dashed lines

Now: Boolean function \(\text{OBDD}\)

Example above: \((a \oplus b) \cdot (c \oplus d) \cdot (e \oplus f)\)

Let

\[F = (a \oplus b) \cdot (c \oplus d) \cdot (e \oplus f) = (a \oplus b) \cdot r\]

Variable ordering \(a, b, c, d, e, f\)

Following Boole's expansion theorem, we have the following cofactors of \(F\) w.r.t. \(a\):

\[
F_a = (0 \oplus b) \cdot r = b \cdot r
\]

\[
F_a = (1 \oplus b) \cdot r = \overline{b} \cdot r
\]
2. Binary Decision Diagrams

2.1 BDD concepts

More expansions:

\[ F_a = (0 \oplus b) \cdot r = b \cdot r \]
\[ F_b = (1 \oplus b) \cdot r = \bar{b} \cdot r \]

\[ F_{ab} = 0 \]
\[ F_{ba} = F_{ab} = r \]
\[ F_{ba} = 0 \]

The problem of reduction:

- In the example above it was easy to detect \( F_{ab} = F_{ba} = r \) and to merge the nodes for \( F_{ab} \) and \( F_{ba} \).
- Redundant nodes have to be removed.

- Redundant nodes
  - Either represent the same function
  - Or have identical successors (easy to detect)
- How to know that two nodes represent the same function?

Two functions

\[ f = a f_a + \bar{a} f_b \]
\[ g = a g_b + \bar{a} g_b \]

are equal iff they have identical cofactors

\[ f_{11} \]
\[ f_{01} \]
\[ f_{00} \]
\[ g_{11} \]
\[ g_{01} \]
\[ g_{00} \]

This results in a simple bottom-up-procedure: redundant nodes are eliminated in the bottom-level first, the in the second level, etc.
2. Binary Decision Diagrams

2.1 BDD concepts

1. Step: 0/1 leaves

2. Step: c nodes

3. Step b nodes

We can decide that the two b-nodes do not represent the same function by means of the c-nodes.

Example: derive the OBDD for the following function, variable order r,e,g

\[ p = e\bar{g} + \bar{r} \bar{g} + \bar{r}\bar{e}g \]

\[ p_r = e\bar{g} + \bar{g} = \bar{g} \]

\[ p_r = e\bar{g} + \bar{e}g \]

Traffic-Light Checker

\[ r \]

\[ e \]

\[ g \]

\[ p \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 1 \]

\[ 0 \]

\[ 1 \]

\[ 0 \]

\[ 1 \]

\[ 0 \]

\[ 1 \]
2. Binary Decision Diagrams

2.1 BDD concepts

- Reduction was necessary in the original concept by R. Bryant (1986), but can be avoided completely (s. Sect. 2.3)

- OBDD's can be implemented easily by means of 2:1-Multiplexors

---

---
2. Binary Decision Diagrams

2.1 BDD concepts

- Given a certain variable ordering, OBDD’s are canonical representations of Boolean functions, i.e., there exists exactly one OBDD-representation for each Boolean function.
- Two circuits implementing the same function have identical OBDD’s.

2.2 Variable Orderings

- The variable ordering has a critical impact on the size of the OBDD (\(= \# \text{nodes} \)).
- There are static and dynamic procedures to determine "good" orderings.

Classical example (Bryant 1986):

\[
 f = x_1x_2 + x_3x_4 + x_5x_6
\]
2. Binary Decision Diagrams
2.2 Variable orderings

Example: n-bit adder:
- Order R₁: aₙ, bₙ, aₙ₋₁, bₙ₋₁, ..., a₀, b₀
- Order R₂: aₙ, aₙ₋₁, ..., a₀, bₙ, bₙ₋₁, ..., b₀

<table>
<thead>
<tr>
<th>n</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁: time</td>
<td>0.02</td>
<td>0.03</td>
<td>0.11</td>
<td>0.19</td>
</tr>
<tr>
<td>#nodes</td>
<td>35</td>
<td>75</td>
<td>155</td>
<td>315</td>
</tr>
<tr>
<td>R₂: time</td>
<td>0.39</td>
<td>16.34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#nodes</td>
<td>750</td>
<td>196574</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

— Calculating the best order may result in exponential runtime.
— For a given circuit, "good" orderings can be heuristically determined.

— Example: Distribution of a "weight"

Sum of weights: x=1/2, y=1/4, z=1/4, ⇒ first use x for expansion

— Sifting: dynamic ordering procedure (Rudell ICCAD’93)
— Basic step: exchange two adjacent variables (Fujita et al. EDAC’91)
Principle: exchange 0-1 and 1-0 path

Sifting-procedure:
- Calculate variable with max. #nodes (the "thickest" part of the OBDD)
- Shift variable over OBDD by pairwise exchange of adjacent variables
- Shift variable to a position where #nodes is minimal
2.2 Variable orderings

Movie "Sifting" by Stefan Höreth:
2. Binary Decision Diagrams

2.2 Variable orderings

Diagram 1:

Diagram 2:

Diagram 3:

Diagram 4:
2. Binary Decision Diagrams

2.2 Variable orderings

[Diagram of a Binary Decision Diagram with nodes labeled V0, V1, V2, V3, V4, V5, V6, V7, and V8, showing the structure of the diagram with decision paths for different variable orderings.]
2. Binary Decision Diagrams

2.2 Variable orderings

Diagram 1:

Diagram 2:

Diagram 3:

Diagram 4:
2. Binary Decision Diagrams

2.2 Variable orderings

2.3 OBDD Construction

- Principle: build OBDD while traversing the circuit from inputs to outputs

- OBDD-Package

- C-Program Traverser

- C-Program ≥1

- C-Program &
2. Binary Decision Diagrams

2.3 OBDD construction

- **AND-operation of two OBDD's**
  - Assumption: nodes are represented as triples
  - \((x,v_0,v_1)\)
  - \(\text{vár low high}
  - \text{access-functions}

```
function AND(bdd1, bdd2):
  IF bdd1=0 OR bdd2=0 THEN return 0;
  ELSEIF bdd1=1 THEN return bdd2;
  ELSEIF bdd2=1 THEN return bdd1;
  ELSE
    var1:=var(bdd1); var2:=var(bdd2);
    IF index(var1) < index(var2) THEN
      x:=var1; v0:= AND(low(bdd1), low(bdd2)), v1:= AND(high(bdd1), high(bdd2));
    ELSEIF
      x:=var2; v0:= AND(low(bdd1), bdd2), v1:= AND(high(bdd1), bdd2);
    ELSEIF
      IF v0 = v1 THEN return v0 ELSE return (x,v0,v1); ...
```
2. Binary Decision Diagrams

2.3 OBDD construction

### Example 1

- BDD 1: \(a \geq 1\) (Node 3)
- BDD 2: \(c\) (Node 5)

Variables: \(a, b, c\)

- \(b = 1\) and \(c = 0\)
- \(b = 0\) and \(c = 1\)

Indexing: \(\text{index}(a) < \text{index}(b)\)

### Example 2

- BDD 1: \(b = 1\) (Node 4)
- BDD 2: \(c\) (Node 5)

Variables: \(b, c\)

- \(b = 1\) and \(c = 0\)
- \(b = 0\) and \(c = 1\)

Indexing: \(\text{index}(b) < \text{index}(c)\)

### Example 3

- BDD 1: \(b = 0\) (Node 1)
- BDD 2: \(c\) (Node 5)

Variables: \(b, c\)

- \(b = 0\) and \(c = 0\)
- \(b = 0\) and \(c = 1\)

Indexing: \(\text{index}(b) = \text{index}(c)\)

### Example 4

- BDD 1: \(b = 1\) (Node 4)
- BDD 2: \(c\) (Node 5)

Variables: \(b, c\)

- \(b = 0\) and \(c = 0\)
- \(b = 0\) and \(c = 1\)

Indexing: \(\text{index}(b) < \text{index}(c)\)
2. Binary Decision Diagrams

2.3 OBDD construction

bdd1  bdd2
var1=b  var2=c \implies \text{index(var1) < index(var2)}

x:=var1 := b
v0:= and(low(bdd1),bdd2),
v1:= and(high(bdd1),bdd2)

var2=c    \implies \text{index(var1) < index(var2)}

x:=var1 := a
v0:= and(low(bdd1),bdd2),
v1:= and(high(bdd1),bdd2)
2. Binary Decision Diagrams

2.3 OBDD construction

- "OBDD-Packages" manage two tables:
  - The unique table (ut) with entries:
    - For uniqueness of OBDD's

- The computed table (ct) with entries
  - Stores previously calculated results

- Many steps of recursion may be saved

- Reduction was needed in the original OBDD procedures
- OBDD uniqueness is guaranteed by
  - Checking in the unique-table (ut) if the OBDD was calculated before
  - Testing for identical successor nodes
- In addition, it is checked in the computed table (ct) if the result was calculated before
  - Many steps of recursion may be saved
2. Binary Decision Diagrams

2.3 OBDD construction

**function** AND(bdd1, bdd2):

*IF* (AND, bdd1, bdd2, x) ∈ ct *THEN* return x;

*IF* bdd1=0 *OR* bdd2=0 *THEN* return 0;

ELSEIF bdd1=1 *THEN* return bdd2;

ELSEIF bdd2=1 *THEN* return bdd1;

ELSE var1 := var(bdd1); var2 := var(bdd2);

*IF* var1 = var2 *THEN*:

x := var1; v0 := AND(low(bdd1), low(bdd2)),

v1 := AND(high(bdd1), high(bdd2));

ELSEIF index(var1) < index(var2) *THEN*:

x := var1;

v0 := AND(low(bdd1), bdd2),

v1 := AND(high(bdd1), bdd2);

ELSEIF ...

*IF* v0 = v1 *THEN* return v0

ELSEIF (x, v0, v1) ∉ ut *THEN* put in ut; ELSE return (x, v0, v1); ...

---

### The computed table is essential for the efficiency of the algorithms:

- In principle, two additional steps of recursion may result at each step
- The number of steps may grow exponentially in the number of variables

Using the computed table with entries

<table>
<thead>
<tr>
<th>Operation</th>
<th>bdd1</th>
<th>bdd2</th>
<th>Result bdd</th>
</tr>
</thead>
</table>

the number of recursions is reduced to |n1*|n2| where |n1| and |n2| are the number of nodes of bdd1 and bdd2, respectively.
2. Binary Decision Diagrams

2.3 OBDD construction

- General result:
  - If two OBDD's with m and n nodes are logically combined then the resulting OBDD has \( \leq mn \) nodes
  - This is due to the fact that not more than \( mn \) distinct functions are generated!

- Negated edges:
  - The OBDD of a function \( f \) and the OBDD of the negated function are very similar: exchange the terminal nodes 0 and 1
  - Orthogonality of negation: negate a function by negating its cofactors

- Problem: non-canonical representation!

2. Binary Decision Diagrams

2.3 OBDD construction

- Solution:
  - Only the 0-edge can be a negated edge
  - 1 terminal leaf only (or the dual version ...)

- Examples: variable and negated variable

\[
\begin{align*}
\text{variable: } & g \\
\text{negated variable: } & \neg g
\end{align*}
\]
Example: XOR function

Cofactor calculation using OBDD's
- cof(x, pol, OBDD): x variable, pol polarity 1 or 0
- Easy if variable = top-variable, e.g., cof(a, 0, OBDD):

Generally: Replace pointers to the variable by the pointer to the 1-(0-)successor:

Example: determine the 0-cofactor for variable d:
2. Binary Decision Diagrams

2.3 OBDD construction

- **Functional substitution**: substitute function $g$ for variable $x$
  - The paper-and-pencil method is: replace all occurrences of $x$ textually by $g$
  - How to do that with an OBDD-representation?
    \[
    f[x \leftarrow g] = \overline{g} \cdot f_x + g \cdot f_x
    \]

    **Rationale:**
    \[
    \begin{align*}
    f &= x \cdot (f_x) + \overline{x} \cdot (f_x) \\
    f[x \leftarrow g] &= \overline{g} \cdot f_x + g \cdot f_x
    \end{align*}
    \]

    **Note:** $\exists x : (f(x = g)) = [f_x(0 = g)] + [f_x(1 = g)]$
    \[
    = f_x \cdot \overline{g} + f_x \cdot g \\
    = f[x \leftarrow g]
    \]

    Functional substitution can be reduced to the application of the $\exists$-operator

- **OBDDs are used in many CAD-tools for synthesis, verification and simulation**
  - Many public domain OBDD-packages
  - Many are based on the $\text{ite}(p, f, g)$-operator (if $p$ then $f$ else $g$)
  - CUDD package (Boulder Univ.)

**Using Boolean operations plus cofactor-calculation more advanced Boolean operations like the $\exists$- and $\forall$-quantifier and functional substitutions can be implemented**
2. Binary Decision Diagrams

2.4 FDD’s and OKFDD’s

- OBDD’s are based on Boole’s expansion theorem
- OBDD’s represent the systematic decomposition in all variables
- Q: Are there other types of “decomposition”? How many?

### Boole’s expansion:
\[ f = x \cdot f_x + \overline{x} \cdot f_{\overline{x}} \]

- There are more types of expansion (exactly two more):
  - Positive Davio-expansion
    \[ f = f_x \oplus x \cdot (f_x \oplus f_{\overline{x}}) \]
  - Negative Davio-expansion
    \[ f = f_x \oplus \overline{x} \cdot (f_x \oplus f_{\overline{x}}) \]

### FDD’s (Functional Decision Diagrams, Kebschull et al. 92)
- Same graph structure, but different interpretation:
  - Rule: variable = 1 → XOR both branches to get the value of f
FDD’s are canonical representations
- FDD’s obey a different rule of reduction:

\[
(f_x \oplus f_x) : \text{if the Boolean difference is 0, then } f \text{ does not depend on } x
\]

Orthogonality of XOR and AND for FDD’s:
- \( f \oplus g = f_x \oplus x^*(f_x \oplus f_x) \oplus g_x \oplus x^*(g_x \oplus g_x) = (f_x \oplus g_x) \oplus x^*[f_x \oplus f_x] \oplus (g_x \oplus g_x) \)
  \(\) Yes!
- \( f \land g = (f_x \land x^*(f_x \land f_x)) \land (g_x \land x^*(g_x \land g_x)) = (f_x \land g_x) \land x^*[f_x \land f_x] \oplus (f_x \land f_x) \land x^*[g_x \land g_x] \)
  \(\) No!

All 4 combinations have to be considered for the AND of 2 FDD’s

ObD and FDD for 4-bit adder

OKFDD’s (Ordered Kronecker FDD’s, Drechsler et al. 94)
- Allows any of the three types of decomposition for each variable
- The type of decomposition is stored in a decomposition type list (DTL)

\[
\begin{align*}
  f &= a^*[0 \oplus c^* (0 \oplus 1)] \oplus b^*[0 \oplus c^* (0 \oplus 1)] \oplus 1 \\
  &= a^*[0 \oplus c] + a^*c
\end{align*}
\]

\[
\begin{align*}
  f &= x \cdot f_x + \overline{x} \cdot f_x \\
  f &= f_x \land x \cdot (f_x \lor f_x) \\
  f &= f_x \lor x \cdot (f_x \land f_x)
\end{align*}
\]
2. Binary Decision Diagrams

2.4 FDD's and OKFDD's

- OBDD's/FDD's/OKFDD's in comparison
  - OKFDD's have OBDD's and FDD's as subclasses
  - OBDD's:
    - AND, OR, XOR of two OBDD's of size n and m requires max. n*m operations
  - FDD's/OKFDD's:
    - XOR requires max. n*m, but AND and OR may need exponentially many operations!
      - However: #nodes of FDD/OKFDD may be exponentially smaller than #nodes of the OBDD (and vice versa)
      - Important for logic synthesis
  - OKFDD's: determining the decomposition-type list (DTL) is an additional problem

2.5 Integer-Valued Decision Diagrams

- So far type \( B^n \rightarrow B^n \), now: type \( B^n \rightarrow Z \):
  - MTBDD's (Multi Terminal Binary Decision Diagrams, Clarke et al. DAC '93)
  - BMD's (Binary Moment Diagrams, Bryant/Chen DAC '95)

Example: multiplier circuits

<table>
<thead>
<tr>
<th>Word length</th>
<th>OBDD nodes</th>
<th>MTBDD</th>
<th>BMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.183</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10.766</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interest in other types of decision diagrams
2. Binary Decision Diagrams

2.5 Integer-valued decision diagrams

MTBDD:
\[ f = (1 - x)f_x + x*f_y \]
where +, -, * are the addition, subtraction and multiplication, respectively

BMD:
\[ f = f_x + x*(f_y - f_x) \]

HDD’s (Clarke/Zhao): combination of MTBDD/BMD, one decomposition types for each variable (~ OKFDD’s)

Example of application (Fujita et a. ‘96):
Vector-matrix operations employing MTBDD’s
Idea:
- encode rows and columns by means of boolean variables
- elements ~ leafs

Example 2*2 Matrix:

\[
\begin{array}{cc|cc}
    & f_{xy} & f_{y} & f_{x} \\
    & f_{x} & f_{y} & f_{xy} \\
\end{array}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>11</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>43</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>43</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

2. Binary Decision Diagrams

2.5 Integer-valued decision diagrams

- Type \( B^0 \rightarrow \mathbb{Z} \) and attributed edges
  - EVBDD’s (Edge Valued Binary Decision Diagrams, Lai et al. ICCAD ’93)
  - *BMD’s (Multiplicative Binary Moment Diagrams, Bryant/ Chen DAC ’95)

- Rule: variable = 1 => add both branches, multiply by weight

- EVBDD:
  \[ f = a + (1 - x)f_x + x*f_y \]
where +, -, * are the addition, subtraction and multiplication, respectively

- *BMD:
  \[ f = m*(f_x + x*(f_y - f_x)) \]

K*BMD’s (Drechsler EDTC ‘96): one decomposition type for each variable (~ OKFDD’s, HDD’s), additive + multiplicative weights

*PHDD (Chen/Bryant ICCAD ‘97): multiplicative power hybrid decision diagrams for floating-point circuits
For *BMD’s we have for an edge without weight:
\[ f = f_x + x^*(f_x - f_1) = f_x + x f_1 \]

*BMD’s are canonical representations provided that:

1. Rule:

2. Rule: the weight on an edge equals the gcd of the weights of the successor edges

\[ \text{gcd}(2, 4) = 2 \]

Example: *BMD for \( f = 4x + 2 \)

\[ f = f_x + x^*(f_x - f_1) = 2 + x^*(6 - 2), \quad \text{gcd}(2, 4) = 2 \]

Next example: *BMD for \( f = 3y + 4x + 2 \)

\[ f = f_x + y^*(f_x - f_1) = (4x + 2) + y^*((4x + 5) - (4x + 2)) \]

\[ = (4x + 2) + y^*3 \]

a leave "n" is a 1 node with weight n
3. Rule: sign of t is sign of left branch

- For "BMD"s with range \(\{0, 1\}\), the boolean operations can be reduced to integer addition, subtraction and multiplication:
  - \(\overline{f}\)
  - \(f \text{ and } g\) \(= f'g\)
  - \(f \text{ or } g\) \(= f + g - f'g\)
  - \(f \text{ xor } g\) \(= f + g - 2f'g\)
2. Binary Decision Diagrams

2.5 Integer-valued decision diagrams

– 4 variable OR (a boolean function)

\[ x_1 = 0, \ x_2 = 1, \ x_3 = 0, \ x_4 = 1 \]

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

– Example: 2-Bit multiplication

\[ x_1, x_0, y_1, y_0 \]

Result:

\[
\begin{equation}
(x_1 \cdot 2^1 + x_0 \cdot 2^2)(y_1 \cdot 2^1 + y_0 \cdot 2^2)
\end{equation}
\]

\[ = 2^0 x_0 \cdot (y_1 \cdot 2^1 + y_0 \cdot 2^2) + 2^1 x_1 \cdot (y_1 \cdot 2^1 + y_0 \cdot 2^2) + x_0 \cdot (y_1 \cdot 2^1 + y_0 \cdot 2^2) + y_1 \cdot 2^1 + y_0 \cdot 2^2 \]

2.6 Bit-Vector Expressions

– Bit-vectors:

\[ \text{Used for the compact representation of complex digital hardware} \]

\[ \text{More adequate than single bits in many cases} \]

\[ \text{Examples: data-paths, arithmetic circuits, register-transfer-level (rtl) descriptions, storage elements, ...} \]

\[ \text{Provided by many hardware description languages (HDL's) as a basic data-type} \]
2. Binary Decision Diagrams

2.6 Bit-vector expressions

--- Example: specification of 74181 ALU

Generic bit-vector function "A PLUS B"

<table>
<thead>
<tr>
<th>S3</th>
<th>S2</th>
<th>S1</th>
<th>S0</th>
<th>M = H</th>
<th>Cn = H</th>
<th>M = L</th>
<th>Cn = L</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>F = not(A)</td>
<td>F = A</td>
<td>F = A PLUS 1</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>F = not(A)</td>
<td>F = A</td>
<td>F = A PLUS 1</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>F = not(A)</td>
<td>F = A</td>
<td>F = A PLUS 1</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>F = not(A)</td>
<td>F = A</td>
<td>F = A PLUS 1</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>F = A PLUS A and(A)</td>
<td>F = A PLUS A and(A) PLUS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>F = A PLUS A and(A)</td>
<td>F = A PLUS A and(A) PLUS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>F = A PLUS A and(A)</td>
<td>F = A PLUS A and(A) PLUS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>F = A PLUS A and(A)</td>
<td>F = A PLUS A and(A) PLUS 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Typical bit-vector functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAE(n)</td>
<td>fan-out</td>
<td>FAE(1,3) = &quot;111&quot;</td>
</tr>
<tr>
<td>ADC(A,B,C)</td>
<td>addition</td>
<td>ADC(&quot;11&quot;, &quot;01&quot;, &quot;1&quot;) = &quot;101&quot;</td>
</tr>
<tr>
<td>ADD(A,B)</td>
<td>addition modulo</td>
<td>ADD(&quot;11&quot;, &quot;01&quot;) = &quot;00&quot;</td>
</tr>
<tr>
<td>INC(A)</td>
<td>increment</td>
<td>INC(&quot;11&quot;) = &quot;000&quot;</td>
</tr>
<tr>
<td>DCR(A)</td>
<td>decrement</td>
<td>DCR(&quot;11&quot;) = &quot;110&quot;</td>
</tr>
<tr>
<td>RSH(C,V)</td>
<td>shift</td>
<td>RSH(0, &quot;111&quot;) = &quot;011&quot;</td>
</tr>
<tr>
<td>LSH(V,C)</td>
<td>shift</td>
<td>LSH(&quot;111&quot;, 0) = &quot;110&quot;</td>
</tr>
<tr>
<td>ROL(A)</td>
<td>rotate left</td>
<td>ROL(&quot;001&quot;) = &quot;010&quot;</td>
</tr>
<tr>
<td>ROR(A)</td>
<td>rotate right</td>
<td>ROR(&quot;010&quot;) = &quot;001&quot;</td>
</tr>
<tr>
<td>MPX1(A,S)</td>
<td>multiplexor 1 Dim.</td>
<td>MPX1(&quot;0010&quot;, &quot;10&quot;) = &quot;1&quot;</td>
</tr>
<tr>
<td>MINT(A,N)</td>
<td>minterm</td>
<td>MINT(A(1:2), 0) = (not A(1)) and (not A(2))</td>
</tr>
<tr>
<td>GT(A,B)</td>
<td>A greater B</td>
<td>GT(&quot;100&quot;, &quot;10&quot;) = 1</td>
</tr>
<tr>
<td>LESS(A,B)</td>
<td>A less B</td>
<td>LESS(&quot;100&quot;, &quot;01&quot;) = 0</td>
</tr>
</tbody>
</table>

Bit-vector functions are necessary for input/output specifications, i.e., for the abstraction from internal details

I/O-Specification

---

2. Binary Decision Diagrams

2.6 Bit-vector expressions

- Bit-vectors in VHDL
  - Type bit_vector predefined
    - signal X: bit_vector (1 to 16) or: (16 downto 1)
  - Selection of single elements X(4) or slices X(2 to 4)
  - Constant-denotation (B)"1001"
  - Assignments X(2 to 4) <= X(8 to 10)
  - Overloaded boolean primitives, e.g., "0101" AND "0011" = "0001"
  - Concatenation &, e.g., X(2 to 4) & X(5 to 7) = X(2 to 7)
Verification problems:
- How to demonstrate the equality of arbitrary bit-vector expressions?
- Do we have to reason formally about tuples, etc.?

Decision procedure: procedure to decide the truth of a statement in some domain
- For generic expressions (may contain expressions of arbitrary length), inductive reasoning is typically used
  - Example: prove $\text{ADD}(A, B) = \text{ADD}(B, A)$ for arbitrary vectors $A$ and $B$
  - Typically a theorem prover is employed
    - You have to derive the proof in large part by yourself
    - The theorem prover checks if the proof is correct
      - Not automated, needs user interaction

For fixed-length expressions, the problem becomes much simpler
- Example: prove $\text{ADD}(A, B) = \text{ADD}(B, A)$ for 32-bit vectors $A$ and $B$
- Several decision procedures exist:
  - The problem can be reduced to OBDD's
  - The problem can be reduced to an integer-linear programming (ILP) problem
  - A specific decision procedure was given by Cyrluk et al. (CAV’97) for a restricted repertoire of bit-vector functions

Reduction to OBDD's ("bit-blasting"):
- Translate expression using bit-vector-functions into multi-level gate-networks
  - e.g., $A \text{ PLUS } B$, where $A$ and $B$ are two 4-bit vectors, is transformed into the gate-network of a four-bit adder
- Then as before!
Technique offers the general possibility to carry out proofs involving complex bit-vector expressions

\[ \text{ADD}(A, B) = \text{ADD}(B, A) \]

\( (\text{ADD}(A, B) > \text{ADD}(B, C)) \rightarrow (A > C) \)

\( (A > B) = \text{NOT}(\text{ADD}(0 \& A, 1 \& \text{NOT}(B))(1)) \)

where A, B, C have fixed length by declaration
Typically takes < 1 sec. for 64-bit vectors

Example: verification of ALU’s
 Verification of VHDL-specification using bit-vector-operations vs. network of standard-cells

<table>
<thead>
<tr>
<th>Wordlength</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-time</td>
<td>1.0</td>
<td>1.5</td>
<td>2.8</td>
<td>6.6</td>
</tr>
</tbody>
</table>

32-Bit ALU: 2 * 32 boolean functions in up to 77 variables

Some references:
- Hassoun/Sasao (Eds.): Logic Synthesis and Verification, Springer
  — Book-Chapters on BDD’s, SAT, Equivalence checking
- Hachtel/Somenzi: Logic Synthesis and Verification Algorithms, Springer
2. Binary Decision Diagrams

2.6 Bit-vector expressions

- Written exam in the summer
  - between 18. Juli and 7. October 2011
  - please follow Doodle link
    http://www.doodle.com/y4igrfy6wcrmx24h