Why BDDs

- BDDs are a **Canonical** representation method for boolean functions.
- Two functions are represented by identical BDDs if and only if they are the same.
- BDDs are often more compact than SOP
  - Parity trees, adders, control logic
  - Multipliers are bad, however, and a few others
BDD Examples

\[
\begin{align*}
\text{for } f(1,0) &= 0 \\
\text{for } f(1,0) &= 1
\end{align*}
\]

BDDs as MUX Circuits

\[
S(x) = S(x_1, x_2, x_3)
\]

\[
S = x_1'y_2 + x_1y_3 = x_1'(x_2'x_3 + x_2) + x_1x_2'
\]

\[
y_3 = x_2'1 + x_20 = x_2', \quad y_2 = x_2'y_1 + x_21, \quad y_1 = x_3'1 + x_30
\]
Effect of Variable Ordering

\[ f = abc + b'd + c'd \]

6 nodes vs. 4 nodes

\[ a \leq b \leq c \leq d \]

BDDs vs Decision Trees
(for given ordering)

- A decision tree is just the result of recursive Boole expansion
- 2 decision tree nodes merge into 1 if they are isomorphic
- A decision tree node is redundant if its two children are identical
Building (RO)BDDs

- Order variables;
- Recursively cofactor the given SOP, in the specified order, until the cofactors become constants or single variables;
- Merge any pair of equivalent nodes;
- Delete any node which has identical children;
- Repeat the merge and delete steps, until neither applies to any node.

\[
\begin{align*}
  f &= ab \\
  f_a &= b, \quad f_{a\overline{0}} = 0 \\
  f_{ab} &= 1, \quad f_{ab\overline{0}} = 0 \\
  f &= a + b \\
  f_a &= 1, \quad f_{a\overline{0}} = b \\
  f_{a\overline{0}b} &= 1, \quad f_{a\overline{0}b\overline{0}} = 0
\end{align*}
\]
Building (RO)BDDs

\[ f = abc + b'd + c'd \]

Order b, c, d, a

---

Building (RO)BDDs

Each node of the BDD corresponds to one of the cofactors of the given function.

Each path corresponds to a minimum satisfying assignment:

\[ f(-,0,-,1) = 1 \]
BDD for Linear Inequality

\[ f = a \leq b \leq c \iff (a \Rightarrow b)(b \Rightarrow c) \]
\[ = (a'b + b'c) = a'bc + a'c + bc \]

Order \( a, b, c \)

Isomorphic subgraphs

Linear growth: \( n \) variable BDD requires \( 2n \) nodes,
Growth of Parity Function

\[ a \oplus b \quad \text{and} \quad a \oplus b \oplus c \]

...?

BDD Advantages

- Though size is exponential in worst-case, BDDs are well-behaved for many functions.
- AND and OR of BDDs have polynomial complexity. Complementation is constant.
- Satisfiability and tautology solved in constant time.
- Covering problems can be solved in time linear in the size of the BDD representation.
BDDs for FSMs

- BDDs can compactly represent huge sets
- Widely used to represent the set of reachable states of a Finite State Machine
- Used in every phase of synthesis and verification of sequential circuits

BDD Disadvantages

- BDD size depends on ordering, and finding a good ordering can be difficult.
- There are functions where SOP/POS representations are more compact.
- The SOP/POS representation may be closer to the form of the final circuit.
**Why Ordering is Important?**

\[ f = ab + cd + ef \]

\[
\begin{array}{c}
\text{a} \leq \text{b} \leq \text{c} \leq \text{d} \leq \text{e} \leq \text{f}
\end{array}
\]

**Design Considerations for BDDs**

- Shared BDDs - multiple functions should share common subexpressions.
- Unique Table - allows checking for a node before creating it, producing reduced BDDs.
- Strong Canonicity - easy equivalence check.
- Attributed edges - ex. complement edges.
- Computed table - cache of recent results.
- Memory management & dynamic ordering.
The Apply Algorithm

- Calculate $f\langle op \rangle g$ where $\langle op \rangle$ is some binary connective using expansion theorem:
  
  $$f\langle op \rangle g = v (f_v \langle op \rangle g_v) + v' (f_{v'} \langle op \rangle g_{v'})$$

- If $f$ does not depend on $v$, $f_v = f_{v'} = f$.
- Otherwise, if $v$ is the top variable, cofactors of $f$ are the two children of the top node.

The ITE Operator

ITE can represent all functions of 2 variables

$$ITE(F,G,H) = FG + F'H$$

- $ITE(F,G,0) = FG$  AND
- $ITE(F,1,G) = F + G$  OR
- $ITE(F,0,G) = F'G$  $F < G$
- $ITE(F,G,1) = FG + F' = F' + G$  $F \leq G$
- $ITE(F,G',G) = FG' + F'G$  $F \oplus G$
- $ITE(F,G,G') = FG + F'G'$  $F \equiv G$
The ITE Recursion

\[ f = \text{ITE}(F, G, H) \]
\[ = FG + F \emptyset H \]
\[ = v(FG + F \emptyset H)_v + v \emptyset (FG + F \emptyset H)_{v \emptyset} \]
\[ = v(F G_v + F \emptyset H_v) + v \emptyset (F G_{v \emptyset} + F \emptyset H_{v \emptyset}) \]
\[ = (v, \text{ITE}(F_{v}, G_{v}, H_{v}), \text{ITE}(F_{v \emptyset} G_{v \emptyset} H_{v \emptyset})) \]

Terminal Cases:
\[ F = \text{ITE}(1, F, G) = \text{ITE}(0, G, F) \]
\[ = \text{ITE}(F, 1, 0) = \text{ITE}(G, F, F) \]

BDD Manipulation

- To implement ITE efficiently, we need to understand two important data structures:
  - The unique table (insures BDD is reduced).
  - The computed table (stores recent calculations).
BDD Data Structure

Node 2, variable $x_i = x_1$

Node 3, variable $x_i = x_2$

\[ f = x_1(x_2x_3) + x_3(x_2) \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>2</th>
<th>5</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L$</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$T$</td>
<td></td>
<td></td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td></td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Top Variables

Top variable of $f$ is $x_1$.

Top variable of $g$ is $x_2$. 

Shared BDDs

Every node represents a Boolean function

\[ a \leq b \quad a \leq b \leq c \quad a \leq b \leq c \leq d \quad \ldots \]

BDD Graph Data Structure

Node \( i \) represents a Boolean function \( f \)

\[
f = x_{f(i)} f x_{f(i)} + x_{l(i)} f x_{l(i)}
\]

Splitting variable \( v = x_{l(i)} \)

To avoid searching this (often large) data structure, we use a Hash Table
BDD Graph Data Structure

- BDD data structures are compact:
  - ~22 bytes/node
  - 1 million nodes needs about 22 MB
- However storage is still the main issue
  - Space out on 1 GB machines common
  - Dynamic reordering necessary

Insuring Uniqueness

- Internally, each node is represented by a triple:
  \[(label(i), j_0, j_1) \leftrightarrow (v, f_{v'}, f_v)\]
- The label of the node is a variable of the function (the same at each level of the DAG)
- The children \((j_0, j_1)\) of node \(i\) represent the cofactors of the node function
- Each triple is stored in memory and is never duplicated
The Unique Table

Reduction rules insure that each triple is unique

$x_{l(i)}$ = variable at level $i$

Key

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>$l(i)$</th>
<th>$l(j_0)$</th>
<th>$l(j_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hashing Function (p235)

The Computed Table

Similar to Unique Table, except:

- The hash key becomes the 4-tuple (ITE, F, G, H)
- The size of the table is fixed in advance, so that the computed table can “fill up”, and new table entries start replacing older entries
- This makes the Computed Table a cache, instead of a hash table
Procedure \text{ITE}(F,G,H)\{ 
    (result,term) = \text{TERMINAL\_CASE}(F,G,H) 
    if(term) \text{return}(result) 
    (result,comp) = \text{IN\_COMPUNTED\_TABLE}(F,G,H) 
    if(comp) \text{return}(result) 
    v = \text{TOP\_VARIABLE}(F,G,H) 
    T = \text{ITE}(F_v,G_v,H_v); 
    E = \text{ITE}(F_{v'},G_{v'},H_{v'}) 
    if(T = E) \text{return}(T) 
    R = \text{FIND\_OR\_ADD\_UNIQUE\_TABLE}((v,T,E)) 
    \text{INSERT\_COMPUNTED\_TABLE}((F,G,H),R) 
    \text{return}(R) 
\}

I = \text{ITE}(F,G,H) 
= (a, \text{ITE}(F_a,G_a,H_a), \text{ITE}(F_{a\varphi},G_{a\varphi},H_{a\varphi})) 
= (a, \text{ITE}(1,C,H), \text{ITE}(B,0,H)) 
= (a,C,(b,\text{ITE}(B_b,0,H_b), \text{ITE}(B_{b\varphi},0,H_{b\varphi})))) 
= (a,C,(b,\text{ITE}(1,0,1), \text{ITE}(0,0,D)))) 
= (a,C,(b,0,D))
ROCBDDs

- In BDDs (ROBDDs) no two distinct nodes represent the same function
- In ROCBDDs (reduced, ordered, complemented) no two nodes have the same or complementary functions
- This magic is worked with a “complement dot” edge attribute and a transformation of some sub-BDDs to a standard form
- 2X (best case) reduction possible

Example of Complement Dots

- Can appear only on 0-edges (E-edges) or outputs
- Function evaluates to 1 on paths with even # of •
Complementation rules (for canonicity):

- 1-edge (T-edge, +cofactor) must be regular
- If complemented function needed on 1-edge, replace appropriate right subgraph with corresponding left subgraph

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

Positive cofactor on left

BDD Proliferation

- Larger Base Boolean Algebras, more constants (ADDs, MTBDDs; matrix algebra)
- Edge Attribution
  - Complement dots (ROCBDDs)
  - Numerical Values (EVBDDs)
- Reduction Rule Variants
  - Zero Suppressed BDDs (ZBDDs do Large sets)
  - Binary Moment Diagrams (BMDs, multipliers)
- Alphabet Soup Hybrids