Why BDDs

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

BDDs as MUX Circuits

\[
S(x) = S(x_1, x_2, x_3) = x_1' y_2 + x_1 y_3 = x_1' (x_2' x_3' + x_2) + x_1 x_2' \\
y_3 = x_2' 1 + x_2 0 = x_2', \quad y_2 = x_2' y_1 + x_2 1, \quad y_1 = x_3' 1 + x_3 0
\]

BDD Examples

\[
f(1,0) = 0 \quad f(1,0) = 1
\]
**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

**Effect of Variable Ordering**

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

- 6 nodes vs. 4 nodes
- \( a \leq b \leq c \leq d \)

**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.

\[ f = ab \]
\[ f_a = b, \quad f_{a\bar{c}} = 0 \]
\[ f_{ab} = 1, \quad f_{ab\bar{e}} = 0 \]

\[ f = a + b \]
\[ f_a = 1, \quad f_{a'} = b \]
\[ f_{a'b} = 1, \quad f_{a'b'} = 0 \]
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:

\[ f(-,0,-,1) = 1 \]

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

Order b, c, d, a

BDD for Linear Inequality

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

Isomorphic subgraphs

Linear growth: \( n \) variable BDD requires \( 2n \) nodes,
<table>
<thead>
<tr>
<th>BDD Advantages</th>
<th>Growth of Parity Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Though size is exponential in worst-case,</td>
<td></td>
</tr>
<tr>
<td>BDDs are well-behaved for many functions.</td>
<td></td>
</tr>
<tr>
<td>• AND and OR of BDDs have polynomial complexity.</td>
<td></td>
</tr>
<tr>
<td>Complementation is constant.</td>
<td></td>
</tr>
<tr>
<td>• Satisfiability and tautology solved in</td>
<td></td>
</tr>
<tr>
<td>constant time.</td>
<td></td>
</tr>
<tr>
<td>• Covering problems can be solved in time</td>
<td></td>
</tr>
<tr>
<td>linear in the size of the BDD representation.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BDD Disadvantages</th>
<th>BDDs for FSMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• BDD size depends on ordering, and finding</td>
<td>• BDDs can compactly represent huge sets</td>
</tr>
<tr>
<td>a good ordering can be difficult.</td>
<td>• Widely used to represent the set of</td>
</tr>
<tr>
<td>• There are functions where SOP/POS</td>
<td>reachable states of a Finite State Machine</td>
</tr>
<tr>
<td>representations are more compact.</td>
<td>• Used in every phase of synthesis and</td>
</tr>
<tr>
<td>• The SOP/POS representation may be closer</td>
<td>verification of sequential circuits</td>
</tr>
<tr>
<td>to the form of the final circuit.</td>
<td></td>
</tr>
</tbody>
</table>

*Diagram of Parity Function:*

```
\overset{a}{\oplus} \overset{b}{\oplus} \overset{c}{\oplus} \ldots?
```

**BDD for FSM:**

```
\overset{a}{\oplus} \overset{b}{\oplus} \overset{c}{\oplus}
```

**Diagram of BDD:**

```
\overset{0}{b} \overset{1}{b} \overset{1}{c} \overset{0}{c}
```
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 ITE Operator

ITE can represent all functions of 2 variables

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

ITE\((F, G, 0) = FG\) \hspace{1cm} \text{AND}
ITE\((F, 1, G) = F + G\) \hspace{1cm} \text{OR}
ITE\((F, 0, G) = F' G\) \hspace{1cm} F < G
ITE\((F, G, 1) = FG + F' = F' + G\) \hspace{1cm} F \leq G
ITE\((F, G', G) = FG' + F'G\) \hspace{1cm} F \oplus G
ITE\((F, G, G') = FG + F'G'\) \hspace{1cm} F \equiv G

Why Ordering is Important?

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.
**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).

**The ITE Recursion**

\[ f = \text{ITE}(F, G, H) = FG + F \text{\&} H \]
\[ = v(FG + F \text{\&} H) + v(FG + F \text{\&} H)_{v \text{\&} c} \]
\[ = v(F_v G_v + F_v H_v) + v(F_v G_v + F_v H_v)_{v \text{\&} c} \]
\[ = (v,\text{ITE}(F_v, G_v, H_v),\text{ITE}(F_v G_v H_v)) \]

**Terminal Cases:**

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

---

**Top Variables**

- Top variable of \( f \) is \( x_1 \).
- Top variable of \( g \) is \( x_2 \).

**BDD Data Structure**

- Node 2, variable \( x_i = x_1 \)
- Node 3, variable \( x_i = x_2 \)

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( 4 )</th>
<th>( 2 )</th>
<th>( 5 )</th>
<th>( 3 )</th>
</tr>
</thead>
<tbody>
<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>
**BDD Graph Data Structure**

Node $i$ represents a Boolean function $f$

$$f = x_{l(i)} f_x x_{l(i)} + x_{l(i)} f_x x_{l(i)}$$

Splitting variable $(l(i), j_0, j_1) \cup (v, f_v, f_v)$

$v = x_{l(i)}$

To avoid searching this (often large) data structure, we use a **Hash Table**

---

**Shared BDDs**

Every node represents a Boolean function

$$a \leq b \quad a \leq b \leq c \quad a \leq b \leq c \leq d \quad ...$$

---

**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

---

**BDD Graph Data Structure**

- BDD data structures are compact:
  - ~22 bytes/node
  - 1 million nodes needs about 22MB

- However storage is still the main issue
  - Space out on 1GB machines common
  - Dynamic reordering necessary
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

\[
I = \text{ITE}(F, G, H) \\
= (a, \text{ITE}(F_a, G_a, H_a), \text{ITE}(a G a' H a' a G a' H a')) \\
= (a, \text{ITE}(1, C, H), \text{ITE}(B_0, H)) \\
= (a, C, (b, \text{ITE}(B_b, 0, H_b), \text{ITE}(B_b, 0, H_b))) \\
= (a, C, (b, \text{ITE}(1, 0, 1), \text{ITE}(0, 0, D))) \\
= (a, C, (b, 0, D))
\]

Procedure I TE(F, G, H)

\[
\begin{align*}
\text{(result,term)} & = \text{TERMINAL\_CASE}(F, G, H) \\
\text{if}(\text{term}) & \text{ return(result)} \\
\text{(result,comp)} & = \text{IN\_COMPUTED\_TABLE}(F, G, H) \\
\text{if}(\text{comp}) & \text{ return(result)} \\
\nu & = \text{TOP\_VARIABLE}(F, G, H) \\
T & = \text{ITE}(F_v, G_v, H_v) \\
E & = \text{ITE}(F_v, G_v, H_v) \\
\text{if}(T = E) & \text{ return(T)} \\
R & = \text{FIND\_OR\_ADD\_UNIQUE\_TABLE}((v, T, E)) \\
& \text{INSERT\_COMPUTED\_TABLE}((F, G, H), R) \\
& \text{return}(R)
\end{align*}
\]
### Example of Complement Dots

<table>
<thead>
<tr>
<th>( a \oplus b )</th>
<th>( a \oplus b \oplus c )</th>
<th>( a \oplus b \oplus c \rightarrow a \Lambda b \oplus a \Lambda b \Lambda c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( a \oplus b \oplus c )</td>
<td>( a \Lambda b \oplus a \Lambda b \Lambda c )</td>
</tr>
<tr>
<td>( b \oplus c )</td>
<td>( b \oplus c \oplus b )</td>
<td>( b \oplus c \oplus b \oplus c )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

- Can appear only on 0-edges (E-edges) or outputs
- Function evaluates to 1 on paths with even # of \( \cdot \)

### 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

### 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

### 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

Positive cofactor on left