Reading material

- de Micheli pp. 75 - 85
- R. Bryant, “Graph-based algorithms for Boolean function manipulation”, IEEE Transactions on computers, C-35, No 8, August 1986; can be downloaded from
  - http://www.cs.cmu.edu/~bryant/
- CUDD-package manual
  - http://vlsi.colorado.edu/~fabio/
Binary Decision Diagram

• Graph-based representation for Boolean functions \( f: B^n \rightarrow B^m \)
  – directed acyclic graph
  – one root node, terminal nodes 0, 1
  – each non-terminal node has two children and is labeled by a variable

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

Binary Decision Diagram

• Mathematical foundation for BDDs is the Shannon decomposition theorem:

\[
f(x_1,x_2, ...,x_n) = x'_1 \cdot f|_{x_1=0} + x_1 \cdot f|_{x_1=1}
\]

where

\[f|_{x_1=0} := f(0,x_2, ...,x_n), f|_{x_1=1} := f(1,x_2, ...,x_n)\]

are subfunctions of \( f \), called co-factors
• Let \( v \) be a node of a ROBDD labeled by some variable \( x_i \), then
  - low(\( v \)) is the subgraph pointed by the edge corresponding to \( x_i = 0 \)
    • this subgraph represents \( f|_{x_i=0} \)
  - high(\( v \)) is the subgraph pointed by the edge corresponding to \( x_i = 1 \)
    • this subgraph represents \( f|_{x_i=1} \)
  - \( \text{index}(v) = i \)
**Ordering rules**

- no variable appears more than once along a path
- in all paths variables appear in the same order

![Diagram](image1.png)

- ordered order = a, c, b
- not ordered

**Reduction rules**

1) no vertex $v$ with $\text{low}(v) = \text{high}(v)$

![Diagram](image2.png)

2) no distinct vertices $v$ and $u$ such that the subgraphs rooted by $v$ and $u$ are isomorphic

- these two rules guarantee that each node represents a distinct logic function
Reduced Ordered Binary Decision Diagram (ROBDD)

- canonical representation (fixed ordering)
- easy manipulation algorithms
- compact for many practical functions

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

Properties of ROBDD

- function is given by tracing all the paths to 1 terminal node: \( f = b' + a'c' = ab' + a'c'b + a'c' \)
- cubes given by paths are pair-wise disjoint
Advantages of ROBDDs

- The number of cubes we represent might be exponential
- But the size of a ROBDD is measured by its nodes, not by its paths
- A ROBDD can represent an exponential number of paths with a linear number of nodes

Problems with ROBDDs

- Some functions have exponential-size ROBDDs
  - multipliers, HWB
- Size of a ROBDD depends critically on variable ordering
  - Adders - exponential for bad orderings, linear for good orderings
- Finding a best variable ordering requires exponential time
Example

Two different orderings, same function

Algorithm for BDD manipulation

• Basic algorithms for BDD manipulation:
  – **Reduce**: BDD G reduced to canonical form, complexity $O(|G| \log |G|)$
  – **Apply**: performs $f_1 \bullet f_2$ for some operation “$\bullet$", complexity $O(|G_1| \cdot |G_2|)$
  – **Compose**: substitutes $f_2$ in a variable $x$ of $f_1$, complexity $O(|G_1|^2 \cdot |G_2|)$
  – **Satisfy-one**: checks whether $f(x_1, \ldots, x_n) = 1$ for some assignment $x_1, \ldots, x_n$, $O(n)$
A simple reduction algorithm:

1) Merge all identical terminal nodes and appropriately redirect their incoming edges
2) Proceeding from bottom to top:
   • if a node \( v \) is found such that \( \text{low}(v) = \text{high}(v) \), remove \( v \) and redirect its incoming edges to \( \text{low}(v) \) (reduction rule 1)
   • if two nodes \( v \) and \( u \) with \( \text{index}(v) = \text{index}(u) \) are found such that \( \text{low}(v) = \text{low}(u) \) and \( \text{high}(v) = \text{high}(u) \), remove \( v \) and redirect its incoming edges to \( u \) (reduction rule 2)

Apply

• The procedure \textit{Apply} provides the basic method for creating the BDD representation of a function given by a Boolean expression
• It takes BDDs for functions \( f_1, f_2 \) and a binary operation \( \bullet \), and produces a BDD for the function \( f_1 \bullet f_2 \) defined as

\[
[f_1 \bullet f_2](x_1,\ldots,x_n) = f_1(x_1,\ldots,x_n) \bullet f_2(x_1,\ldots,x_n)
\]
Apply

- Apply can be used to complement a function (compute \( f \oplus 1 \))
- to compute intersection (\( \cdot = \cdot \))
- to compute union (\( \cdot = + \))
- Note: in cube representations we needed 3 different procedures for that

ITE operator

- If-then-else: \( \text{ite}(f,g,h) \) equals to \( g \) when \( f \) is 1 and equals to \( h \) otherwise:
  \[
  \text{ite}(f,g,h) = f \cdot g + f' \cdot h
  \]
- Each node is written as a triple: \( f = (v,g,h) \) where \( g = f_v \) and \( h = f_{\overline{v}} \), where \( v \) is the root node of \( f \)
- We read this triple as:
  \[
  f = \begin{cases} 
  g & \text{if } v \\
  h & \text{else}
  \end{cases} = \text{ite}(v,g,h) = v g + v' h
  \]

p. 17 - Advanced Logic Design – L4 - Elena Dubrova

p. 18 - Advanced Logic Design – L4 - Elena Dubrova
ITE operator

- Ite can be used to implement any binary operation, like AND, OR
  \[ f \cdot g = \text{ite}(f, g, 0) \]
  \[ f + g = \text{ite}(f, 1, g) \]
- Ite can be implemented by extending Apply procedure to ternary operations

ITE operator

<table>
<thead>
<tr>
<th>Table</th>
<th>Subset</th>
<th>Expression</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>AND(f, g)</td>
<td>(fg)</td>
<td>(\text{ite}(f, g, 0))</td>
</tr>
<tr>
<td>0010</td>
<td>(f &gt; g)</td>
<td>(f \oplus g)</td>
<td>(\text{ite}(f, g, 0))</td>
</tr>
<tr>
<td>0011</td>
<td>(f)</td>
<td>(f)</td>
<td>(f)</td>
</tr>
<tr>
<td>0100</td>
<td>(f &lt; g)</td>
<td>(f \oplus g)</td>
<td>(\text{ite}(f, 0, g))</td>
</tr>
<tr>
<td>0101</td>
<td>(g)</td>
<td>(g)</td>
<td>(g)</td>
</tr>
<tr>
<td>0110</td>
<td>XOR(f, g)</td>
<td>(f \oplus g)</td>
<td>(\text{ite}(f, g, g))</td>
</tr>
<tr>
<td>0111</td>
<td>OR(f, g)</td>
<td>(f + g)</td>
<td>(\text{ite}(f, 1, g))</td>
</tr>
<tr>
<td>1000</td>
<td>NOR(f, g)</td>
<td>(f \oplus g)</td>
<td>(\text{ite}(f, 0, g))</td>
</tr>
<tr>
<td>1001</td>
<td>XNOR(f, g)</td>
<td>(f \oplus g)</td>
<td>(\text{ite}(f, g, g))</td>
</tr>
<tr>
<td>1010</td>
<td>NOT(g)</td>
<td>(\overline{g})</td>
<td>(\text{ite}(g, 0, 1))</td>
</tr>
<tr>
<td>1011</td>
<td>(f \geq g)</td>
<td>(f + \overline{g})</td>
<td>(\text{ite}(f, 1, g))</td>
</tr>
<tr>
<td>1100</td>
<td>(f \leq g)</td>
<td>(f)</td>
<td>(\text{ite}(f, 0, 1))</td>
</tr>
<tr>
<td>1101</td>
<td>(f \leq g)</td>
<td>(f + g)</td>
<td>(\text{ite}(f, g, 1))</td>
</tr>
<tr>
<td>1110</td>
<td>NAND(f, g)</td>
<td>(fg)</td>
<td>(\text{ite}(f, g, 1))</td>
</tr>
<tr>
<td>1111</td>
<td>(\overline{1})</td>
<td>(\overline{1})</td>
<td>(\overline{1})</td>
</tr>
</tbody>
</table>
Variable ordering problem

• Exact algorithms are intractable
• Heuristics are used:
  – static: use properties to group the variables together (e.g. keep symmetric variable together)
  – dynamic sifting: use the property that exchanging adjacent variables has only a local effect on the BDD; the variables are sifted from top to bottom one by one and the best position is remembered

Static variable ordering

• variable ordering is computed up-front based on the problem structure
• works very well for many combinational functions that come from circuits we actually build
  – general scheme: control variables first
• works bad for unstructured problems
  – e.g. using BDDs to represent arbitrary sets
• lots of research in ordering algorithms
  – simulated annealing, genetic algorithms
  – give better results but extremely costly
**Dynamic variable ordering**

**Theorem (Friedman):** Permuting any top part of the variable order has no effect on the nodes labeled by variables in the bottom part. Permuting any bottom part of the variable order has no effect on the nodes labeled by variables in the top part.

---

**Dynamic variable ordering**

- Dynamic variable ordering is done as follows:
  - shift a BDD variable to the top and then to the bottom and see which position had minimal number of BDD nodes
  - fix the variable to this position
  - repeat for all variables
**BDD package (CUDD)**

- A BDD package is a software program that can manipulate ROBDDs:
  - It can store multiple Boolean functions, sharing all vertices that can be shared.
  - It can create new functions by combining existing ones (e.g. $f = g \cdot h$)
  - It can convert the internal representation back to an external one

---

**Basic data structures**

- The basic component is DdNode, which is a structure with several fields

```c
typedef struct DdNode {
    struct DdNode *low, *high; /* pt to low, high child */
    int index;                 /* index of the variable that */
                               /* labels the node, in 0...n */
    char value;                /* range in {0,1} of terminal nodes */
                               /* x for non-terminal nodes */
    int id;                    /* reference count - identifier */
    ...                       /* unique to that node */
}... /* unique to that node */
```
Efficient Implementation

Unique Table:
- avoids duplication of existing nodes
  - Implemented by a hash-table

Computation Table:
- avoids re-computation of existing results
  - Implemented by a software cache

Building a BDD

- It is normally more efficient to build BDDs “bottom-up”
- ITE-procedure Cudd_bddIteVar can be used
- Other procedures for binary operations, like Cudd_bddAnd or Cudd_bddOr can also be used
Example of building a BDD using CUDD package

• Building BDD for \( f = x'_0 \land x'_1 \land x'_2 \land x'_3 \):

```c
DdManager *manager;
DdNode *f, *var, *tmp;
int i;

f = Cudd_readOne(manager); /* start from const 1*/
Cudd_ref(f); /* referencing f */
for(i=3; i >= 0; i--) {
    /* building bottom-up */
    var = Cudd_bddIthVar(manager,i);
    tmp = Cudd_bddAnd(manager, Cudd_Not(var),f); /*assigning to a
tmp variable, old f should be kept to free its nodes */
    Cudd_ref(tmp);
    Cudd_recursiveDeref(manager,f);
    f = tmp;
}
```

Garbage Collection

• It is very important to free and reuse memory of unused BDD nodes. This is done by garbage collection.
• Timing is very crucial because garbage collection is expensive. There are two approaches:
  1. do immediately when a node gets freed
     – bad because dead nodes get often reincarnated in next operation
  2. regular garbage collections based on statistics collected during BDD operations
     – better
Extension - complemented edges

• Complemented edge indicates that the function associated with it is the complement of the function being pointed by the edge
  – reduces memory requirements
  – BUT MORE IMPORTANT:
    • makes some operations more efficient (NOT, ITE)

![Diagram of two different BDDs leading to only one BDD using complement pointer]

Solution: Always choose one on **left**, i.e. the “then” edge must have **no** complement edge.
Other extensions

- Many different extensions of BDDs are possible
  - algebraic DDs: for $B^n \rightarrow M$ functions
    - multi-terminal BDDs
    - decision tree is binary
    - multiple leaves, including real numbers, sets or arbitrary objects
    - efficient for matrix computations and other non-integer applications
  - MDDs: for $M^n \rightarrow M$ functions
    - can be implemented using a regular BDD package with binary encoding
    - advantage that binary BDD variables for one MV variable do not have to stay together -> potentially better ordering
  - FDDs: Free BDDs
    - variable ordering differs; not canonical anymore

Zero Suppressed BDD’s - ZBDD’s

ZBDD’s were invented by Minato to efficiently represent sparse sets. They have turned out to be useful in implicit methods for representing primes (which usually are a sparse subset of all cubes).

Different reduction rules:
- **BDD**: eliminate all nodes where dotted edge and solid edge point to the same node.
- **ZBDD**: eliminate all nodes where the solid edge points to 0. Connect incoming edges to node pointed by dotted edge.
- **For both**: share equivalent nodes.
Summary of BDDs

• BDDs give a compact representation for many practical functions as well as fast manipulation algorithms
  – widely used in CAD-tools nowadays
• They are very convenient for formal verification (equivalence checking)
  – comparing of two functions is done by comparing two pointers to the root nodes (constant-time operation)