Lecture 6: Formal verification and BDDs
Logic Design Automation for VLSI 046880

What we already learned:
- Working with cube-lists to represent boolean functions
- Positional cube notation - useful bitwise operations on cubes
- Shannon expansion and cofactors
- Properties of monotonic (unate) boolean functions
- Unate Recursive Paradigm to decide if a function is a tautology
- This was how things used to work from the 50’s to the late 80’s...

Today’s lesson:
- More on cofactors
- Using Binary Decision Diagrams to represent boolean functions.
  - An old basic idea: first proposed in the 50’s, used again in the 70’s, improved significantly by Bryant in the late 80’s, that made a big impact on logic CAD of the 90’s.
Recall the Shannon expansion:

- Given a boolean function $F(x_1, x_2, \ldots x_i, \ldots x_n)$
- Take variable $x_i$
- The cofactor of F for $x_i$ is: $F_{x_i} = F(x_1, x_2, \ldots x_i[1] \ldots x_n)$
- The cofactor of F for $x_i'$ is: $F_{x_i'} = F(x_1, x_2, \ldots x_i[0] \ldots x_n)$
  - These cofactors are new functions (of all the other variables)
- Shannon’s expansion is: $F = x_iF_{x_i} + x_i'F_{x_i'} = (x_i + F_{x_i'})(x_i' + F_{x_i})$
- Repeated further expansions will lead to a sum of minterm products (or to product of maxterms if we use the dual form)
Plots of the cofactors

example: \( f=ab + bc + ac \)

- Remember: the cofactors wrt \( a \) are not functions of \( a \), because we get rid of \( a \) in the Shannon expansion.
- Cofactor wrt \( a \) is: \( f \) when \( a=1 \).
- Cofactor wrt \( a' \) is: \( f \) when \( a=0 \).

\[ \begin{array}{c}
000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
\hline
a & b & c & f & f_a & f_a' \\
\end{array} \]
Operations between 2 functions using their cofactors

- Cofactor of complement of F is the complement of the cofactor of F
- Express 2 functions (F,G) by their 2 pairs of cofactors, then:
  - Cofactors of F*G are AND of the respective cofactors
  - Cofactors of F+G are OR of cofactors...
  - …. Same for any other logic operation between the 2 functions

\[ f \text{ OP } g = x_i \cdot \left( f_{xi} \text{ OP } g_{xi} \right) + x_i' \cdot \left( f_{xi}' \text{ OP } g_{xi}' \right) \]
Operation between the 2 cofactors of a single function $F$

- Product of the cofactors is called the consensus of $F$ wrt $x_i$. It is the part of the function that does not depend on $x_i$.
- Sum of the cofactors is called the smoothing of $F$ wrt $x_i$. It corresponds to deleting $x_i$ from the expression of $F$.
- Exclusive OR of the cofactors is called the boolean derivative, or boolean difference of $F$ wrt $x_i$ ($dF/dx_i$). If it is 0, then $F$ does not depend on $x_i$ ($x_i$ is unobservable). If it is 1, $F$ changes when $x_i$ changes. Useful for testing.

Note: we could have plotted these functions on the b-c plane, because they are independent of a!
Smoothing and Consensus Properties

- Consensus wrt $a$ is the biggest function independent of $a$ that’s contained in $f$
- Smoothing is the smallest function independent of $a$ that contains $f$
Smoothing and Consensus as quantifiers

- Consensus($f$ wrt $x$) is also called the *universal quantifier*; it means “for all $x$, $f$ ….” $\forall_{x\in\{0,1\}}(f = 1)$

- Smoothing($f$ wrt $x$) is also called the *existential quantifier*; it means “there exist $x$ such that $f$ ….” $\exists_{x\in\{0,1\}}(f = 1)$

- Usage of these quantifiers:
  - To derive or check a property $f$, which should be obeyed for all values of input variable $x_i$
  - To check whether there is any input value $x_i$ that would make $f$ satisfied
Smoothing and Consensus Example

- **Given**: The function \( F = ac + a'c' + b \)
  - \( Fa = c + b, \ Fa' = c' + b \)

- **Find**:
  What are the restrictions on \( b \) and \( c \) such that \( F = 1 \)
  a) *for all values of* \( a \)
  b) *There exists a value of* \( a \) *that* \( F = 1 \)

- **Solution**:
  a) “For all values of \( a \)” means consensus of \( F \) vs. \( a \) : \( Fa*Fa'=(c+b)(c'+b)=b \)
    i.e. \( b \) must be 1 for \( F \) to be 1 for all values of \( a \)
  b) “For any value of \( a \)” means smoothing of \( F \) vs. \( a \) : \( Fa+Fa'=(c+..)+(c'+..)=1 \)
    i.e. there is no restriction on \( b \) or \( c \). No matter what are their values there is a value of \( a \) that makes \( F = 1 \)
Back to logic verification: Why logic equivalence checking is hard?

- Because it’s hard to represent boolean functions
  - Truth tables are canonical…… but size $2^n$ is impractical for $n>{\sim}16$
  - Sum of minterms or product of maxterms - same problem
  - 2-level forms - we used cube lists (SOP), and there are many possible covers - not canonical
    - Size of minimal representation depends on function (e.g. EXOR is large!)
    - Looking for a minimal cover is a tough problem
    - Passing from POS to SOP is hard: E.g. ANDing two SOPs
    - Complementation is hard
    - Tautology decision is hard (also: satisfiability decision is hard)

- All we need for equivalence checking is a compact, efficient, canonical representation:
  - Bring 2 functions to canonical form and see if it’s the same
  - This canonical form should be efficient
  - .... These are properties of ROBDD: Reduced Ordered BDDs
What is a BDD?

Binary Decision Diagram

- Let’s start from a Decision Tree for function $f$
  - Each vertex is a binary decision
    - Testing a single variable value
    - If 1 leads to the THEN branch or if 0 to the ELSE
  - Leaf nodes are function values
  - In this example, BDD:
    - Use variable order: $a, b, c$
    - Each variable is tested once in a path
- Testing all the variables, is exactly like a truth table
  - What’s the equivalent of a minterm?
  - Where is the shortcut in our example?
- Optimizations
  - It would be nice to have just one vertex per variable
  - Can we do it here?
BDD as “IF THEN ELSE” (s/w analogy)

BDD as Steering Logic (h/w analogy)

if A = '1' then
  if B = '1' then
    if C = '1' then parity <= '1';
    else parity <= '0';
    endif;
  else
    if C = '1' then parity <= '0';
    else parity <= '1';
    endif;
  endif;
else
  if B = '1' then
    if C = '1' then parity <= '0';
    else parity <= '1';
    endif;
  else
    if C = '1' then parity <= '1';
    else parity <= '0';
    endif;
  endif;
end if;

Steering Logic: multiplexer-based circuits
ROBDD - Reduced Ordered BDD

- Ordering = “layering” the BDD by order of variables
  - a < b < c < …… (we’ll use integers to number them)
  - They must appear **in this order** along any path
- Reduction = removing any redundancies:
  - No vertex with THEN = ELSE
  - No isomorphic sub-graphs (i.e. exact same sub graphs)
- Purpose: Get a compact, unique form (canonical!)
- Consequences of reduction:
  - All the 1-leaves are isomorphic. Merge them. Same for 0-leaves.
  - Merge vertices with same variable and same children
  - Multiple arrows can lead to same vertex
  - Remove redundant tests (THEN son)=(ELSE son)
  - Keep merging and reducing…
  - The result: no longer a tree, just a DAG
  - In practice, reduction rules are observed during building of the BDD – bottom up!
Relation of BDD to Shannon expansion

- The children of a node labeled ‘a’ are the cofactors w.r.t. a!

\[ F = a f_a + a' f_a' \]

- The whole BDD represents recursive applications of Shannon’s expansion….

  - Keep splitting and cofactoring with regard to all variables
  - End result is a complete BDD for F
Relation of BDD to Shannon expansion

- The children of a node are the cofactors!
- The whole BDD represents recursive applications of Shannon expansion

- Some functions and their BDDs:
  - \( f = 1 \)
  - \( f = 0 \)
  - \( f = a \)
  - \( f = ab \)
  - \( f = a + b \)
  - \( f = a \text{ XOR } b \)
Some properties of ROBDDs

- Tautology checking is trivial
- Satisfiability (is it possible to get a 1) is also easy
  - Just find any path to the leaf 1! (check that the BDD is not ==0)
- AND and OR have the same complexity
- Complementation is cheap (later)
- However:
  - The size of a BDD might be exponential in the number of variables
  - For most functions size depends on choosing a good variable ordering

<table>
<thead>
<tr>
<th>Function</th>
<th>Best order BDD size</th>
<th>Worst order BDD size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adders</td>
<td>linear</td>
<td>exponential</td>
</tr>
<tr>
<td>Symmetric e.g. majority, parity</td>
<td>linear</td>
<td>quadratic</td>
</tr>
<tr>
<td>Multipliers</td>
<td>exponential</td>
<td>exponential</td>
</tr>
</tbody>
</table>

- No magic solution to variable ordering (it is NP-hard)
The next idea: Shared BDDs

- So every node is a root of a sub-graph representing some function which is a “piece of f” - a cofactor of the function corresponding to parent node.

- We can allow a multi-rooted DAG to share the same BDD data structure by many functions:
  - It becomes a “warehouse of function parts” (shared)
  - Every piece exists just once (Reduced)
  - Every piece must exist in a unique form (Ordered → Canonical)
  - Think about resemblance to folded hierarchy in a structural h/w model.

![Diagram showing BDDs](image-url)
The Unique Table

- We want to keep our BDD always reduced: no duplicate parts
- Whenever we want to add a node to the BDD, labeled by variable v, with functions G and H as the high son and low son - we must first check if such a node already exists
- Instead of going through all the nodes to check this, use a hash table or a sorted container, for storing a direct pointer to such a node if it exists.
  - Finding in the hash-table takes constant time, or in sorted container log(n)
- The key for the hash or compare functions is composed from (v, G,H)
  - v is the node’s variable - represented by an integer
  - v is called the top variable of f
  - G,H are the children functions - represented by pointers

The key to pointer to f is: \( V,G,H \)
Now we have a **strong canonical form**

- Strong canonicity: Equivalence test between two functions is just a comparison between their unique id’s
- Two equivalent functions must share the same subgraph in our BDD, and have the same pointer to its root
- With our unique table, another unique id of a function is the key \((v,G,H)\)

If \(f_i = f_j\) they must have the same **pointer**!
Complemented edges (negation)

- **Purpose:** To save memory, represent a function and its complement by the same subgraph in the BDD
- Let’s use a *negation attribute (denoted by a dot)* on the edge leading into the complemented function
  - It means “invert the final value you get”
  - Implemented in memory by an attribute bit
- To preserve canonicity:
  - Use dots only on *Else* pointers (use 0 - side of each pair below)
Examples of complemented edges

\[ \text{NOT}(b) \]

0 \quad 1

\[ \text{NOT}(a) + \text{NOT}(b) \]

0 \quad 1

0 \quad 1

0 \quad 1

0 \quad 1

0 \quad 1

Odd parity

1

1

1
The Importance of Variable Ordering

Example: \( f=ax+by+cz \)

Use order:
\[ a<x<b<y<c<z \]
GOOD!

Use order:
\[ a<b<c<x<y<z \]
...BAD!

Spring 2017
What makes a good ordering?

- Keep together variables which are closely related

\[ f = ax + by + cz \]

- Inputs which have “big effect” should be early in the order
  - big effect = every change in this variable is likely to change the output
Levelizing the circuit can help in variable ordering

- Given a logic network, want to build a BDD
- Order the variables in reverse order of the topological sort!
- Bandwidth $w$ is the number of wires crossing the widest cut-line
- Maximum of $2^w$ decisions between levels $i$ and $i+1$ \(^{[1]}\)
  - Hence at most $n^* 2^w$ nodes in the BDD, where $n$ is the number of input variables
- ...But how about swapping the order of $x_2$ and $x_3$? Why not?.....

Depth-first-search ordering heuristics

- Idea: Try to group together input variables whose influence “converges” into one signal
- Method: Do a depth-first-search, going backwards from the output. When you find a primary input, append it to the current ordering.
- Use the reverse order in BDD
- Improvements:
  - Give priority to nets with multiple fanouts
  - Compute DFS depths, give priority to nets with deeper depths

DFSOrder (node, order_list) {
    if (not in_list_already(node, order_list))
        foreach fanin DFSOrder (fanin, order_list);
        if primary_input append (order_list, node);
}
Breadth vs. Depth first graph traversal
BFS vs. DFS

- We’ll see these terms many times. They are easy to understand on TREES, but are essentially the same on directed graphs:
  - Breadth-first means: work on all nodes you can access (“go sideways”), before you go to explore further nodes. Done with a FIFO (queue).
    - E.g. Levelizing is a breadth-first ordering
  - Depth-first means: move forward (“deeper”) as far as you can get, before you do any work. Done with a LIFO (stack).
    - In recursive routines, calculate results after you return from recursive call
Weight Assignment Ordering heuristic

- **Rationale:** we are looking for inputs that have a big effect on the output

- **Heuristic:**
  - Assign weight=1.0 to the output
  - Walk backwards, the weight at a gate’s output is distributed equally to the inputs of the gate
  - At each fanout point, accumulate the weights backwards into the driver gate
  - Pick the primary input with biggest weight: it has the biggest effect on the output in a topological sense
  - Repeat the same procedure after deleting part of the circuit which depends only on the chosen input
  - If there are many outputs, begin with the output of deepest depth from the primary inputs
Dynamic variable reordering

- Typical applications build and dispose of many BDDs
- Program may “space-out” if memory is not well-managed
- The BDD software package can identify and maintain the variable ordering, *dynamically re-ordering* as necessary to save memory
  - Re-ordering is based on adjacent pair-exchanges
  - This is an interesting minimization problem by itself...
- Unused memory is recovered by *garbage collection*
- These 2 mechanisms are essential for robust BDD packages
BDDs Intermediate Summary

- Binary Decision Diagrams as a data structure
  - Ordered
  - Reduced
  - Shared (multi-rooted DAG)
  - Canonical form
  - Unique Table
  - Negation edges
  - Variable ordering heuristics

- Next
  - Applying operations between functions in BDDs
  - More Operations on BDDs
  - More efficiency improvements
  - Using BDDs for verification
  - The problems of Finite-State-Machine verification
References