Lecture 7: Operations on BDDs
Logic Design Automation for VLSI 046880

- Reminder of last lecture:
  BDDs - Binary Decision Diagrams
  - Reduced, Ordered, Shared (multi-rooted DAG)
  - Canonical form (often compact, sometimes not)
  - Efficiency tricks: Unique Table, Negation Edges
  - Not just a tree – a forest!

- Today’s lesson:
  - Dynamic Reordering
  - Computations with BDDs
    - Recursive operations based on Shannon expansion
  - Representing sets with BDDs
  - Applications of BDDs
Why 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: garbage collection and dynamic re-ordering
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.

- **Trick:** Two adjacent variable layers can be exchanged by keeping the original **memory locations** for the nodes.

---

More in tutorial
BDD sifting (ala R. Rudell, Synopsys)

Algorithm:
- Shift each BDD variable to the top and then to the bottom
- Move it to the position that resulted with minimal number of BDD nodes
- Stop if lower bound on size is worse then the best found so far
  - Shortcut: two layers can be swapped very cheaply if there is no interaction between them
- Expensive operation, sophisticated trigger condition to invoke it

Grouping of BDD variables:
- For many applications, pairing or grouping variables gives better ordering
  - e.g. current state and next state variables in state traversal
- Grouping them for sifting explores ordering that are otherwise skipped
Operations on BDDs

- **Strategy:**
  - Represent whole circuit in a shared BDD, using identical variable ordering - for all functions: $x_1 < x_2 < x_3 < x_4, \ldots < x_n$
  - Build it incrementally, keeping it reduced all the time
  - Each circuit gate is an operator, that may produce a new BDD node

- So, we need to implement symbolic operations on BDDs:
  - E.g. NOT(f), f AND g, f OR g, f EXOR g, \ldots cofactor(f, x), \ldots
  - The arguments are BDD’s, the result is a new BDD
  - All with the same variable ordering, sharing existing nodes whenever possible - use the unique table to keep the BDD reduced

![Gate Level Model](image)

![BDD Forest Model](image)
Recursive algorithm to APPLY an operation

- Shannon expansion can help, by “divide and conquer”: 
  \[ F \ <op> \ G = v \ ( Fv \ <op> \ Gv ) + v' \ ( Fv' \ <op> \ Fv' ) \]

- To apply operation \(<op>\) between F and G, here is the algorithm:
  - Find \(v\) - it is the “top variable” of F and G
    - top variable means: the first variable in F or G according to order
  - Find cofactors
    - If F depends on \(v\), the cofactors are: Fv and Fv'
    - If F does not depend on \(v\): F
  - Perform the operation on cofactors (simpler problems!)
  - If the sub-problems are not simple enough, use recursion!
  - Finally, create a node labeled \(v\), that points to the results of the two sub-problems
    - If such a node already exists in the unique table, just return its pointer, otherwise create it in the BDD and record in the unique table
Terminal operations

- The termination rules and terminal operations depend on OP
  - Example: \( f_1 \text{ OR } f_2 \)
    - What are “simple enough” cofactors?
    - How to perform the operation on them?

- So, APPLY needs to be re-written for each OP
- We’ll soon find a more general, elegant way to compute APPLY operations
COMPOSE operation

- Composition $g$ of $f_1$ and $f_2$: $g = f_1\mid_{x_i=f_2}$

- Use Shannon: $g = f_1\mid_{x_i=f_2} = f_2 f_1\mid_{x_i=1} + \text{not}(f_2) f_1\mid_{x_i=0}$
  - We already know how to do $\ast +$ operations (APPLY)
  - We need another operation to compute $f_1\mid_{x_i=1}$ $f_1\mid_{x_i=0}$ (RESTRICT)
  - RESTRICT can be implemented recursively too

- The expression above is of the form
  \[
  \text{IF } f_2 \text{ THEN } g = f_1\mid_{x_i=1} \\
  \text{ELSE } g = f_1\mid_{x_i=0}
  \]

- It can be implemented directly as an IF-THEN-ELSE operator
ITE operator (If-Then-Else)

- This is an operation involving 3 functions, yielding a new function.
  - Will be useful for implementing almost everything we’ll ever need!!

- For “hardware people”:
  Assume x is a vector $x_1, x_2, x_3, \ldots$ of input variables

- For “software people”:

- For mathematicians:
  $ITE(I,T,E)(x) = I(x) \cdot T(x) + I'(x) \cdot E(x)$

ITE($I, T, E$) {
  if ($I(x_1, x_2, x_3, \ldots) = 1$)
  then return $T(x_1, x_2, x_3, \ldots)$
  else return $E(x_1, x_2, x_3, \ldots)$
}

More in tutorial
ITE compared to BDD-node

- A regular BDD-node is an if-then-else decision, based on testing a primary input variable v
  - (v,G,H) is its key in unique table
    - v is an integer
    - G,H are pointers

- ITE is a decision based on a functions I, T, E
  - Their values are computed by separate BDDs
  - ITE is a composition of functions
  - Recall how to compose f1(f2) with ITE:
    - I = f_2
    - T = f_1|_{x_i=1}
    - E = f_1|_{x_i=0}
Implementing ITE on our BDD

- Again….. Shannon expansion can help, by “divide and conquer”!:

\[ \text{ITE}(I,T,E) = x \cdot \text{ITE}(I_x, T_x, E_x) + x' \cdot \text{ITE}(I_x', T_x', E_x') \]

- Now we have a regular BDD-node labeled with some primary input variable \( x \).
- This is recursive!
- Questions:
  - Who is \( x \)?
  - How to compute the 6 cofactors?
  - When to terminate recursion?
  - How to keep our shared BDD reduced?
Who is X?

- In principle, it can be any input variable
- Practically, we want to pick $x$, such that cofactors will be trivial
- There “natural winner” is….. $X =$ top variable of functions I,T,E
  - The cofactors will be already existing inside the reduced BDD!

Bad choice: $c$ (or any other “deep” variable)
why?

Good choice: $b$ or above
what are the cofactors if $x=b$? What are they if $x=a$ or $x=w$?
When to terminate recursion?

- Termination rules:
  \[
  \text{ITE}(0, T, E) = E \\
  \text{ITE}(1, T, E) = T \\
  \text{ITE}(I, 1, 0) = I \\
  \text{ITE}(I, 0, 1) = I' \\
  \text{ITE}(I, F, F) = F
  \]

- All of these are existing pointers in the BDD -> time to return!

- Otherwise, add a new node, labeled x, into the BDD
  - Before creating a new node, check in the Unique Table whether it already exists: use the key (x,G,H)
  - Skip the creation in case H==G, to keep BDD reduced

\[
\begin{align*}
G &= \text{ITE}(I_x, T_x', E_x') \\
H &= \text{ITE}(I_x, T_x, E_x)
\end{align*}
\]
Computed Table: a hash table for Efficiency

- During the recursive ITE algorithm, we often make identical calls - ITE(F,G,H) with same F,G,H!
  - Why work hard just to re-discover an existing pointer?

- Keep a log of all recent ITE calls, with their results
  For efficient look-up, the log will also be a hash-table
  It’s called the **Computed Table**. Its Key=(F,G,H)

- Computed table Vs. Unique table:
Effect of computed table on complexity

- Without computed table, the ITE algorithm might have exponential time complexity if we are unlucky: $O\left(2^N\right)$

- With the computed table, worst-case complexity is the product of graph sizes for functions $I$, $T$, $E$: $O\left(|I| \cdot |T| \cdot |E|\right)$
  - Often, $T$ or $E$ are constants: 0 or 1
  - If we are unlucky, the computed table will blow up with all possible keys.... Assume infinite memory, or impalement as a cache, with possibility of a miss...
  - Usually, complexity is proportional to the size of the result graph!

1. $ITE(I,T,E)$
2. $ITE(I_x,T_x,E_x)$
4. $ITE(I_x,T_x,E_x)$

How many possible keys?

$ITE(I,T,E)$
The ITE algorithm

ITE (I,T,E) {
    If (terminal case) {
        return result;
    } else if (computed_table_has_entry(I,T,E)) {
        return result;
    } else {
        v = top_variable(I,T,E);
        Highson = ITE( I_v, T_v, E_v);
        Lowson = ITE( I_v', T_v', E_v');
        if (Highson=Lowson) return Lowson;
        R = find_or_add_unique_table(v, Highson, Lowson);
        insert_computed_table({I,T,E},R);
        return R;
    }
}
Expressing logic operations by ITE
(APPLY by using ITE)

- **AND** \( (F, G) = \text{ITE} (F, G, 0) \)
- **OR** \( (F, G) = \text{ITE} (F, 1, G) \)
- **NOR** \( (F, G) = \text{ITE} (F, 0, G') \)
- **EXOR** \( (F, G) = \text{ITE} (F, G', G) \)
- **EXNOR** \( (F, G) = \text{ITE} (F, G, G') \)
- **NOT** \( (F) = \text{ITE} (F, 0, 1) \)
- **NAND** \( (F, G) = \text{ITE} (F, G', 1) \)
- **IMPLY** \((F \Rightarrow G) = \text{ITE} (F, G, 1) \)

\*

**IMPLY** = \( G + F' \)
“Implication Check” operation

- We often want to check whether \( F \subseteq G \)
  - If \( F \subseteq G \), it means \( F \Rightarrow G \) (F implies G)
    - The proposition \( F \Rightarrow G \) is true if \( G \) is true whenever \( F \) is true.
      If \( F \) is false, the proposition is true regardless of \( G \).
      (in other words: “IF \( F \) then \( G \), Else 1”)

- So: \( F \Rightarrow G = F' + G \)
  - Exercise: express the following fully in terms of AND and \( \Rightarrow \): \( (s'+h) \ast (h'+m) \Rightarrow (s'+m) \)

- This amounts to checking if \( F'+G==1 \) (tautology)

- ITEconstant: A modified ITE algorithm that efficiently returns whether the function is a constant \( ==0, ==1 \), or a non-constant.
  - Useful to decide whether \( F \) implies \( G \) \( (F \Rightarrow G \iff F'+G) \), by testing ITE Constant \((F,G,1)=1 \)
  - It is more efficient than building \( F'+G \) and checking tautology, because:
    - It terminates early whenever a non-constant cofactor is detected
    - It does not build the intermediate-result BDD
New idea: Represent sets by functions

- Why is this a good idea?
  - Because we can represent the functions symbolically by BDDs
  - We’ll see that large sets can often be represented by small BDDs!

- How to do this?
  - Say we have $N$ objects, from which we pick elements of our sets
  - Step 1: Domain Encoding; Assign a binary number to each object
    - We’ll need $\log_2(N)$ bits for it. But we can use more… any encoding OK
  - Step 2: Build a set; Choose objects that will be in the set
  - Step 3: Define a characteristic function of the set, as follows:
    $$f(x) = \begin{cases} 1 & \text{if } x \text{ is in our set} \\ 0 & \text{otherwise} \end{cases}$$

- example:
  - Days of the week, encoded:
    - Sunday=001, Monday=010, Tuesday=011, … Saturday=111.
  - $S$ represents the set {Sunday, Tuesday, Thursday}
  - $T$ represents the set {Friday, Saturday, Sunday}

<table>
<thead>
<tr>
<th>x2</th>
<th>x1</th>
<th>x0</th>
<th>T</th>
<th>S</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>1</td>
<td>1</td>
<td>1</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>1</td>
<td>0</td>
<td>1</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>1</td>
<td>0</td>
<td>1</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>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Symbolic operations on sets represented by characteristic functions represented by BDDs

- Operation on sets can be implemented by Boolean operations on their characteristic functions
  - Take our weekdays example:
    - The characteristic function of $S \cup T$ is Boolean OR (+) of their functions
    - The characteristic function of $S \cap T$ is Boolean AND (*) of their functions
    - To test if a set $S$ is empty: If the $S == 0$
    - To test if a set $S$ is a subset of another set $T$: if $S \Rightarrow T$

- What’s the advantage of working with the characteristic functions? (instead of working directly with the elements of the sets?)
A small BDD can represent a large set

- Assume we have a state machine with 20 states.
- Let's use 5 bits to binary-encode them as follows: \{0-3,8-15,16-23\}.
- This called ‘natural encoding’. 4,5,6,7,24-31 are unused codes.
- A BDD representing the characteristic function the of 20 states:
  x4 and x5 are not referenced in the BDD.

\[
\begin{array}{cccccc}
\text{x1} & \text{x2} & \text{x3} & \text{x4} & \text{x5} & \text{f} & \text{#elements} \\
0 & 0 & 0 & - & - & 1 & 4 \ [0-3] \\
0 & 1 & - & - & - & 1 & 8 \ [8-15] \\
1 & 0 & - & - & - & 1 & 8 \ [16-23] \\
\end{array}
\]

- Now, if we add an unreferenced variable x6, and keep the BDD unchanged, it will represent a set of 40 elements :\{0-7,16-31,32-47\}… because unreferenced variables mean more paths to 1!
- With n variables, same BDD represents \(5 \times 2^{(n-3)}\) elements.
A small BDD can represent a large set

- Assume we have a state machine with 20 states.
- Let’s use 5 bits to binary-encode them as follows: \{0-3,8-15,16-23\}.
- This called ‘natural encoding’. 4,5,6,7,24-31 are unused codes.
- A BDD representing the characteristic function the set of 20 states: x4 and x5 are not referenced in the BDD.

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
<th>f</th>
<th>#elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>4 [0-3]</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>8 [8-15]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>8 [16-23]</td>
</tr>
</tbody>
</table>

- Now, if we add an un referenced variable x6, and keep the BDD unchanged, it will represent a set of 40 elements :\{0-7,16-31,32-47\}... because un referenced variables mean more paths to 1!
- With n variables, same BDD represents \(5 \times 2^{(n-3)}\) elements.
Digital Applications of BDDs

- **Verification**
  - Combinational equivalence
  - Formal Equivalence Verification
  - FSM equivalence
  - Symbolic Simulation
  - Symbolic Model Checking

- **Synthesis**
  - Don’t care set representation
  - State minimization
  - Sum-of-Products minimization

- **Test**
  - False path identification
Generating ROBDD from Network

**Task:** Represent output functions of gate network as OBDDs.

### Network
![Network Diagram](image)

### Evaluation
- **A:** `new_var("a");`
- **B:** `new_var("b");`
- **C:** `new_var("c");`
- **T1:** `ITE(A, 0, B);` // and
- **T2:** `ITE(B, 0, C);` // and
- **Out:** `ITE(T1, 1, T2);` // or

### Resulting Graphs
![Resulting Graphs](image)
Checking Network Equivalence

- **Task:**
  Do two networks compute same Boolean function?

- **Method:**
  Compute ROBDDs for both networks and compare

- If built using same v,G,H hash – should be trivial
Finite State System Analysis

- Systems Represented as Finite State Machines
  - Sequential circuits
  - Communication protocols
  - Synchronization programs

- Analysis Tasks
  - State reachability
  - State machine comparison
  - Temporal logic model checking

- Traditional Methods Impractical for Large Machines
  - Polynomial in number of states
  - Number of states exponential in number of state variables.
  - Example: single 32-bit register has 4,294,967,296 states!
BDD Summary

- Binary Decision Diagrams as a data structure
  - Ordered
  - Reduced
  - Shared (multi-rooted DAG)
  - Canonical form
  - Unique Table
  - Negation edges
  - Variable ordering heuristics
  - Applying operations between functions using their BDDs
  - ITE algorithm
  - Computed Table for time-saving during recursion
  - Boolean derivative wrt a variable (XOR of cofactors)
  - Quantification operations between cofactors (universal, existential)
  - Characteristic functions
  - BDDs to represent large sets (set→characteristic function →BDD)
Ignore next slides for now

STOP HERE
Symbolic FSM Representation

- Represent set of transitions as function $\delta(\text{Old, New})$
  - Yields 1 if can have transition from state $\text{Old}$ to state $\text{New}$
- Represent as Boolean function
  - Over variables encoding states

**Diagram:**

- FSM
  - States: 00, 01, 10, 11
  - Transitions: 00 to 01, 00 to 10, 01 to 00, 01 to 11, 10 to 00, 10 to 11, 11 to 00, 11 to 11

- Symbolic Representation
  - Variables: $o_1, o_2$ for encoded old state
  - Variables: $n_1, n_2$ for encoded new state
Reachability Analysis

- **Task**
  - Compute set of states reachable from initial state \( Q_0 \)
  - Represent as Boolean function \( R(S) \)
  - Never enumerate states explicitly

Initial

Given

Compute

\[ R_0 = Q_0 \]
Breadth-First Reachability Analysis

- $R_i$ – set of states that can be reached in $i$ transitions
- Reach fixed point when $R_n = R_{n+1}$
  - Guaranteed since finite state
Iterative Computation

- $R_{i+1}$ – set of states that can be reached $i+1$ transitions
  - Either in $R_i$
  - or single transition away from some element of $R_i$
Example: Computing $R_1$ from $R_0$

$\exists \text{Old} \left[ R_0(\text{Old}) \land \delta(\text{Old}, \text{New}) \right]$