CAD of VLSI

Tutorial # 7

Algorithms on BDDs

- Represent functions using BDD’s – not other representations as PCN
- Perform operations on functions (on BDD’s)
- Perfrom operations on BDD structure (without changing the function)
Dynamic Variable Reordering

- The order of the variables in the BDD is crucial for its size.
- As the ROBDD is built, the optimal order may change.
- The optimal order can change at any stage of the usage.
- Most BDD packages supply dynamic variable re-ordering.
  (combined with dynamic memory management)
- Variable order can change the BDD size between:
  Exponential to polynomial (by number of variables).
- Many heuristic methods.
Dynamic Variable Reordering - adjacent variables swapping

Recall that \( F_{xy} = F_{yx} \)

The two BDDs are equivalent!
All of the leaves are the same and the routes are identical.
Not much computation required, if we use a unique table.
Variable Reordering - Example:

Function: \( f = a \cdot b + c \cdot d \)

- Change the variable ordering in the BDD for the function \( F \)
- From: “\( a < c < b < d \)” into “\( a < b < c < d \)”
- Note: we will swap only adjacent variables (\( b \iff c \))
Variable Reordering - Example: Result

Function: $f = a \cdot b + c \cdot d$

```
F
  a
  v
  c
  v
  b
  v
  d
  v
0 1

“a < c < b < d”
```

```
F
  a
  v
  c
  v
  b

  v
  d
  v
0 1

“a < b < c < d”
```
Variable Reordering - Example: Steps

Function: $f = a \cdot b + c \cdot d$

“a < c < b < d”
Variable Reordering - Example:

Function: \( f = a \cdot b + c \cdot d \)

```
F_a': F_a'bc = d
F_a'bc' = 0
F_a'b'c = d
F_a'b'c' = 0

F_a: F_{abc} = 1
F_{abc}' = 1
F_{ab'c} = d
F_{ab'c'} = 0
```

```
"a < c < b < d"
"a < b < c < d"
```
Variable Reordering - Example:

Function: $f = a \cdot b + c \cdot d$

"a < c < b < d"

"a < b < c < d"
Variable Reordering - Example:

Function: \( f = a \cdot b + c \cdot d \)

```
F_a':
F_a'bc = d
F_a'bc' = 0
F_a'bc = d
F_a'bc' = 0
```

```
F_a:
F_abc = 1
F_abc' = 1
F_ab'c = d
F_ab'c' = 0
```

```
F_a:
F_abc = 1
F_abc' = 1
F_ab'c = d
F_ab'c' = 0
```

“a < c < b < d”

Local changes!

“a < b < c < d”
Restrict operator -
Restrict the variable \( x \) (of function \( f \)) to value \( k \).

This is co-factoring!

\[
f(x,y,z,\ldots) \mid_{x=k} = f(k,y,z,\ldots)
\]

Example: \( bc + ab’c’ \mid_{b=1} = c \)

Problem: Given the BDD of \( f \) find the BDD of \( f \) with \( x \) restricted to 1 (or 0).

Solution: Redirect any arc into vertex \( v \) having \( \text{var}(v) = x \) to point to \( \text{hi}(v) \) for \( x=1 \) (or \( \text{lo}(v) \) for \( x=0 \))

Restrict - working top down, must remove redundancies.
Restrict - Algorithm

Restrict( F, var, value )
{
    if ( result = AlreadyComputed( F, var, value ) ) return result;
    else if ( Var( F ) > var ) return F;
    else if ( Var( F ) < var )
    {
        u = CreateNode( Var(F), Restrict(Fx', var, value), Restrict(Fx, var, value) );
        InsertComputed( F, var, value, u );
        return u;
    }
    else
    {
        if ( value == 0 ) return Restrict(Fx', var, value);
        else return Restrict(Fx, var, value);
    }
}

x: top var of F
Restrict – Algorithm – Create Node

\text{CrateNode}(v, \text{HSON}, \text{LSON})
\
\{
    \textbf{If} ( \text{HSON} == \text{LSON} )
    \{
        \text{return} \text{HSON};
    \}

\textbf{else if} ( \text{n}ode = \text{Unique\_Table\_Exists}(v, \text{HSON}, \text{LSON}) )
    \{
        \text{return} \text{n}ode;
    \}

\textbf{else}
    \{
        \text{new\_node} = \text{Unique\_Table\_Insert}(v, \text{HSON}, \text{LSON});
        \text{return} \text{new\_node};
    \}
\}
Restrict - Example

Restrict $b$ to 1

Function: $f = b \cdot c + a \cdot b^\prime \cdot c^\prime$
Restrict - Example

Restrict $b$ to 1

For all the nodes of ‘b’
Restrict - Example

Restrict b to 1

For all the nodes of ‘b’
Delete the nodes.
Redirect the above arcs to:
the output of the ‘1’ (blue) arcs.
Restrict - Example

Restrict $b$ to 1

Redundant
Restrict - Example

Restrict $b$ to 1

Redundant
ITE: If Then Else operator.

Generates a function out of 3 input functions.

$$\text{ITE}(F, G, H) = F \cdot G + F' \cdot H$$

Can be read as: If $F == 1$ Then $G$ Else $H$.

Shannon expansion for ITE:

$$\text{ITE}(I, T, E) = x \cdot \text{ITE}(I_x, T_x, E_x) + x' \cdot \text{ITE}(I_{x'}, T_{x'}, E_{x'})$$
Expressing logic operations by ITE

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>= 0</td>
<td>$ITE(0, 0, 0)$</td>
</tr>
<tr>
<td>AND (F, G)</td>
<td>= $F \cdot G$</td>
<td>$ITE(_, _, _)$.</td>
</tr>
<tr>
<td>F</td>
<td>= F</td>
<td>$ITE(_, _, _)$.</td>
</tr>
<tr>
<td>G</td>
<td>= G</td>
<td>$ITE(1, G, G)$.</td>
</tr>
<tr>
<td>XOR(F, G)</td>
<td>= $F \cdot G' + F' \cdot G$</td>
<td>$ITE(_, _, _)$.</td>
</tr>
<tr>
<td>OR(F, G)</td>
<td>= $F + G$</td>
<td>$ITE(F, 1, G)$.</td>
</tr>
</tbody>
</table>
# Expressing logic operations by ITE

Recall $\text{ITE}(F, G, H) = FG + F'\text{H}$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>= 0</td>
<td>$\text{ITE} (0, 0, 0)$</td>
</tr>
<tr>
<td>AND (F, G)</td>
<td>= $F \cdot G$</td>
<td>$\text{ITE} (F, G, 0)$</td>
</tr>
<tr>
<td>F</td>
<td>= F</td>
<td>$\text{ITE} (1, F, F)$</td>
</tr>
<tr>
<td>G</td>
<td>= G</td>
<td>$\text{ITE} (1, G, G)$</td>
</tr>
<tr>
<td>XOR(F,G)</td>
<td>= $F \cdot G' + F' \cdot G$</td>
<td>$\text{ITE} (F, G', G)$ *</td>
</tr>
<tr>
<td>OR(F,G)</td>
<td>= $F + G$</td>
<td>$\text{ITE} (F, 1, G)$</td>
</tr>
</tbody>
</table>
Expressing logic operations by ITE

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>ITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOR(F, G)</td>
<td>$(F + G)'$</td>
<td>$ITE(F, 0, G')$</td>
</tr>
<tr>
<td>XNOR(F, G)</td>
<td>$(F XOR G)'$</td>
<td>$ITE(_, _, _) $</td>
</tr>
<tr>
<td>NOT(G)</td>
<td>$G'$</td>
<td>$ITE(G, 0, 1) $</td>
</tr>
<tr>
<td>$F \geq G$</td>
<td>$F + G'$</td>
<td>$ITE(F, 1, G') $</td>
</tr>
<tr>
<td>NOT(F)</td>
<td>$F'$</td>
<td>$ITE(F, 0, 1) $</td>
</tr>
<tr>
<td>$F \leq G$</td>
<td>$F' + G$</td>
<td>$ITE(F, G, 1) $</td>
</tr>
<tr>
<td>NAND(F, G)</td>
<td>$(F \cdot G)'$</td>
<td>$ITE(_, _, _) $</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$ITE(1, 1, 1) $</td>
</tr>
</tbody>
</table>

$(ITE = any-op)$
Expressing logic operations by ITE

Recall $\text{ITE}(F, G, H) = FG + F'H$

\begin{align*}
\text{NOR}(F, G) &= (F + G)' \quad &= \quad \text{ITE} (F, 0, G') \\
\text{XNOR}(F, G) &= (F \text{ XOR } G)' \quad &= \quad \text{ITE} (F, G, G') \\
\text{NOT}(G) &= G' \quad &= \quad \text{ITE} (G, 0, 1) \\
F \geq G &= F + G' \quad &= \quad \text{ITE} (F, 1, G') \\
\text{NOT}(F) &= F' \quad &= \quad \text{ITE} (F, 0, 1) \\
F \leq G &= F' + G \quad &= \quad \text{ITE} (F, G, 1) \\
\text{NAND}(F, G) &= (F \cdot G)' \quad &= \quad \text{ITE} (F, G', 1) \\
1 &= 1 \quad &= \quad \text{ITE} (1, 1, 1)
\end{align*}
The ITE data structures

Shannon expansion for ITE:
\[ \text{ITE}(I,T,E) = x \times \text{ITE}(I_x, T_x, E_x) + x' \times \text{ITE}(I_{x'}, T_{x'}, E_{x'}) \]

Meaning that an ITE node can be expressed as a Unique Table entry.
Assuming \( x \) is the ITE’s top variable:
\[ \text{ITE}(I,T,E) = (x, \text{ITE}(I_x, T_x, E_x), \text{ITE}(I_{x'}, T_{x'}, E_{x'})) \]

The search of an ITE in the Unique Table is not trivial.
Co-factoring (Restrict) all 3 functions and building ITE’s from it first.

The Computed Table saves this possibly redundant computation.
For every (depending on the memory available) ITE computed:
The triple \((I,T,E)\) is kept as an index of the Computed Table, pointing to the output function (pointer into the Unique Table).
The ITE data structures

ITE(I,T,E) = x * ITE(I_x, T_x, E_x) + x' * ITE(I_x', T_x', E_x')

Unique Table

<table>
<thead>
<tr>
<th>Var</th>
<th>Hi_son</th>
<th>Lo_son</th>
<th>pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>ITE(I_x, T_x, E_x)</td>
<td>ITE(I_x', T_x', E_x')</td>
<td></td>
</tr>
</tbody>
</table>

Computed Table

<table>
<thead>
<tr>
<th>If</th>
<th>Then</th>
<th>Else</th>
<th>pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>T</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

Example: \{ F = ITE(1, F, F) \}
The ITE algorithm

ITE (I, T, E)
{
    if (result = terminal_case(I, T, E)) return result;
    else if (result = computed_table_has_entry(I, T, E)) return result;
    else
    {
        v = top_variable(I, T, E);
        Highson = ITE( I_v, T_v, E_v);  // ↺ recursion here !
        Lowson = ITE( I_v', T_v', E_v'); // ↺ recursion here !
        if (Highson==Lowson) return Lowson; // = return Highson
        R = find_or_add_unique_table(v, Highson, Lowson);
        insert_computed_table({I, T, E}, R);
        return R;
    }
}
If Highson == Lowson ...

- When building the BDD for $F = \text{ITE}(I, T, E)$ with variable “a” at the root node, if we find that the Lowson == Highson it means that the function is independent of “a” and that this node is redundant.
- In this case a pointer to the Lowson (== Highson) is returned.
- Note this BDD is already in the database as it was created by a previous recursive call.
Terminal cases

Note that for example:

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

The BDDs of \( F \) and \( G \) are known whenever one of the above cases is reached, the recursion can be terminated and a pointer to \( F \) returned.

Obviously many more redundant cases exist.

Would you be able to hold yourself until the “Set of equivalent triples” slide?
Example of ITE recursion

\[
R = \text{ITE} ( F, G, H ) =
\]

\[
= ( \_, \text{ITE}[\_, \_, \_, \_], \text{ITE}[\_, \_, \_, \_]) \quad \text{// co-factor (==\_\_\_\_\_\_Restrict!)}
\]

\[
= ( \_, \text{ITE}[\_, \_, \_, \_], \text{ITE}[\_, \_, \_, \_])
\]

\[
= (a, C, \quad (b, \text{ITE}[\_, \_, \_, \_], \text{ITE}[\_,\_,\_,\_])) \quad \text{// co-factor}
\]

\[
= (a, C, \quad (b, \text{ITE}[\_, \_, \_, \_], \text{ITE}[\_,\_,\_,\_])) \quad \text{// trivial}
\]

\[
= (a, C, \quad (b, \_, \_, \_))
\]

\[
\text{ITE} (I, T, E) \{
\text{if (result = terminal_case(I, T, E)) return result;}
\text{else if (result = computed_table_has_entry(I, T, E)) return result;}
\text{else } \{
\text{v = top_variable(I, T, E);} 
\text{Hson = ITE(I_v, T_v, E_v);} 
\text{Lson = ITE(I_v', T_v, E_v);} 
\text{if (Hson==Lson) return Lson;} 
\text{R = find/add_unique_table(v, Hson, Lson);} 
\text{insert_computed_table({I, T, E}, R);} 
\text{return R; } \} \}
\]
Example of ITE recursion

\[ R = \text{ITE}(F, G, H) = \]
\[ = (a, \text{ITE}[F_a, G_a, H_a], \text{ITE}[F_{a'}, G_{a'}, H_{a'}]) \] // co-factor (== Restrict!)
\[ = (a, \text{ITE}[1, C, H], \text{ITE}[B, 0, H]) \]
\[ = (a, C, (b, \text{ITE}[B_b, 0_b, H_b], \text{ITE}[B_{b'}, 0_{b'}, H_{b'}])) \] // co-factor
\[ = (a, C, (b, \text{ITE}[1, 0, 1], \text{ITE}[0, 0, D])) \] // trivial
\[ = (a, C, (b, 0, D)) \]
Set of equivalent triples

\[ F = a' + bd \]
\[ G = (a + b) \cdot d \]
\[ H = 0 \]

\[ F_1 = bd \cdot (a + c') \]
\[ G_1 = 1 \]
\[ H_1 = a'bcd \]

\[ \text{ITE}(F, G, H) = \text{ITE}(F_1, G_1, H_1) = bd \]

Finding all triples are difficult, but some are easy to identify, can speed up computation:

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

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

A useful check - $F \leq G$ or $F \Rightarrow G$ i.e. if $F == 1$ then $G == 1$

Equivalent to proving that: $F' + G = 1$

We could compute $ITE(F,G,1)$ and check for a tautology.

We will use the efficient $ITE_{Constant}(F,G,1)$ instead.

$ITE_{Constant}$ returns 1, 0 or non-constant

If $ITE_{Constant}(F,G,1) == 1$ then $F \Rightarrow G$
The ITE CONSTANT algorithm

ITE_CONSTANT (I, T, E) {
    if (result = terminal_case(I, T, E)) return result;
    else if (result = computed_table_has_entry(I, T, E)) return result;
    else {
        v = top_variable(I, T, E);
        Highson = ITE(v, T, E);
        Lowson = ITE(v', T, E);
        if (Highson == Lowson) return Lowson;
        R = find_or_add_unique_table(v, Highson, Lowson);
        insert_computed_table({I, T, E}, R);
        return R;
    }
}

ITE (I, T, E) = x * ITE(I_x, T_x, E_x) + x' * ITE(I_{x'}, T_{x'}, E_{x'})
ITE (I, T, E) = (x, ITE(I_x, T_x, E_x), ITE(I_{x'}, T_{x'}, E_{x'}))