CAD of VLSI

Tutorial #6

Binary Decision Diagrams

BDDs

- How to Represent functions using BDD’s
- Unique Table
- How to Build Reduced Ordered BDDs
- How to Build Complement Edge BDDs
Binary Decision Diagram

Introduction: draw an ordered **BDD** for the following truth table:

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Ordered BDD (OBDD):**
The variables in all paths always appear in the same order $x_1 < x_2 < x_3$
Introduction: draw an ordered BDD for the following truth table:

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Ordered BDD (OBDD):
The variables in all paths always appear in the same order $x_1 < x_2 < x_3$
Reduced OBDD - ROBDD

Example: Reduce the BDD given on the previous slide.

**Step 1: Duplicate terminal removal**

![Diagram of BDD reduction](image)
Reduced OBDD - ROBDD

Example: Reduce the bdd given on the previous slide.

Step 1: Duplicate terminal removal
Reduced OBDD - ROBDD

Example: Reduce the bdd given on the previous slide.

**Step 2:** Duplicate non-terminal removal
Reduced OBDD - ROBDD

Example: Reduce the bdd given on the previous slide.

Step 2: Duplicate non-terminal removal
Reduced OBDD - ROBDD

Example: Reduce the bdd given on the previous slide.

Step 3: Redundant test removal
Reduced OBDD - ROBDD

Example: Reduce the bdd given on the previous slide.

Step 3: Redundant test removal
Storing BDDs

Shared BDDs:
BDD structures can be shared by different functions.

Example:
\( f = abc \)  \( (g=bc, \ h=c) \)

The bdd for \( F \) can be created as a multi-rooted bdd.
Once the bdds of \( C \) and \( BC \) have been computed, pointers to these structures (\( C \) and \( BC \)) can be used to create the bdd for \( F \).
The Unique Table (UT)

Unique Table:
- A data base storing pointers to all the subgraphs of the BDD.
- The table is unique - there is a single pointer to each node.
- The purpose of the unique table is to avoid redundancies in the multi-rooted BDDs.
- The main Idea is to check for existing nodes before adding.
- The key to the table:
  - The key is a description of the node:
  - (v,H,L) where “v” is the decision variable,
  - “H” and “L” are it’s two sons - subgraphs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key</th>
<th>Node Ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>v</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C:</td>
<td>c</td>
<td>0x01</td>
</tr>
<tr>
<td>BC:</td>
<td>b</td>
<td>0x03</td>
</tr>
<tr>
<td>ABC:</td>
<td>a</td>
<td>0x04</td>
</tr>
</tbody>
</table>
Building BDDs - *wrong way*

**Exercise**: Build a ROBDD for the following function: \( f = a'b + ac \)

**Intuitive solution:**

1) Work bottom up,
   Build Primitives:

2) Build (simple) implicants:
   Remark: what about large implicants?

3) Join the implicants???
   Not reduced, nor unique !!!

Recall the **Shannon expansion**, how it applies to BDDS ?
The Shannon Expansion

Shannon’s expansion: \( f = a f_a + a' f_a \),

The BDD represents recursive application of Shannon’s expansion!

Decomposing the function (co-factoring)
Creates subgraphs that are unique (unless it already exists), and are used in building the function.
Build BDD - Algorithm

Build_BDD (F)
{
    If ( terminal = Terminal_Case( F ) )
    {
        return terminal; }
    else
    {
        v = Top_Variable( F ); // Assumes global variable order
        F_v = Co_Factor( F , v );
        F_v` = Co_Factor( F , v` );
        HSON = Build_BDD (F_v ); // recursive call
        LSON = Build_BDD (F_v` ); // recursive call
        return Unique_Table( v , HSON , LSON );
    }
}
Build BDD - functions

Unique_Table( v , HSON , LSON )
{
    If ( HSON == LSON )
    {    return HSON;  }
else if ( node = Unique_Table_Exists ( v , HSON , LSON ) )
    {    return node;  }
else
{
    new_node = Unique_Table_Insert ( v , HSON , LSON );
    return new_node;
}
}

Terminal_Case( F )
{
    if ( F == ‘0’) {    return BDD(‘0’);  }
else if ( F == ‘1’) {    return BDD(‘1’);  }
else {    return NON_TERMINAL;  }
}
More Terminal Cases

• If we find at some point during our recursive process that \( f_{ab'c} = d \) (not a Terminal_Case) we will continue to cofactor it further to \( f_{ab'cd} = 1 \) and \( f_{ab'cd'} = 0 \) which are both terminal cases.

• However, building a BDD-Key for \( f_{ab'c} = d \) (or even \( g = d' \)) is very easy:

• So we can from now on say that any function which is equal to a literal (such as \( a \) or \( b, c, \ldots \)) or even a negated one (\( a' \) or \( b', \ldots \)) is also a Terminal_Case.

• Actually even functions like \( p = a + b \) or \( q = cd \) can be also easily built and said to be terminal cases too.
**Exercise 1**

**Exercise**: Build a shared ROBDD for the following functions:

A) \( f = (a + b) \cdot c + d \)
B) \( g = ac' + d \)
C) \( h = f + g \)

- We will use the Build_BDD algorithm
- The variable ordering is: \( a < b < c < d \) , \((a) is the top\).
- Computation is always “bottom up”.
  First compute \( f \) and \( g \) and then “\( f \text{ op } g \)”
  Recall that: \( f \text{ op } g = x(f_x \text{ op } g_x) + x'(f_x \cdot \text{ op } g_x) \)
Exercise 1-A

- We begin with an empty Unique_Table:
  For our convenience we add two more columns (for humans only).

<table>
<thead>
<tr>
<th>Boolean exp’</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- We recall our algorithm:

- We begin with our function:
  \[ f = (a + b) \cdot c + d \]
  (performing Build_BDD \((f)\))

- Obviously \(f\) is not a Terminal_Case so we break it into cofactors \(f_a, f_a'\):
  \[ f_a = c + d; \quad f_a' = bc + d \]
- and enter recursively to compute \(f_a\).
Build_BDD (F) {  // F= f_a = c + d
  If ( terminal = Terminal_Case( F ) ) {
    return terminal;
  } else {
    v = Top_Variable( F );    // v=b
    F_v = Co_Factor( F , v );  // F_v = f_v = c + d
    F_v = Co_Factor( F , v' ); // F_v = f_v = b + c + d
    HSON = Build_BDD ( F_v );
    LSON = Build_BDD ( F_v );
    return Unique_Table( v , HSON , LSON );
  }  }

Exercise 1-A (cont’)

(performing Build_BDD (f_a))

• Obviously f_a = c + d is also not a Terminal_Case so the recursion continues to f_ab = c + d ; f_ab' = c + d.
  (performing Build_BDD (f_ab))

• f_ab; f_ab’ are also not Terminal_Cases by themselves, so we continue and finally reach : f_abc = 1; f_abc' = d

• While f_abc = 1 is Terminal_Case, f_abc’ = d is not but it is trivial to (perform Build_BDD (f_abc’)) (which returns 0x01)

• For this expanded Terminal_Case the Key is simply: {d, 1, 0}

<table>
<thead>
<tr>
<th>Boolean exp’</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>d; f_abc’</td>
<td>0x001</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Exercise 1-A (cont')

(performing Build_BDD (f_{ab'}))

- Now that we have finished with all of the recursion tree under \( f_{ab}=c+d \), and we know that \( v=c \), HSON=1, LSON=0x01 we can build another BDD part with Key={c, 1, 0x01} (returns 0x002)

- Having finished with Build_BDD (f_{ab}) the next step is (performing Build_BDD (f_{ab'}))

- But because \( f_{ab'}=c+d \) exactly equals \( f_{ab}=c+d \) for which the relevant BDD part was already built it returns the same pointer LSON=HSON=0x02.

- Humans can add another Label to 0x02:

<table>
<thead>
<tr>
<th>Boolean exp'</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d )</td>
<td>( d; f_{abc'} )</td>
<td>0x01</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( c+d )</td>
<td>( f_{ab}; f_{ab'} )</td>
<td>0x02</td>
<td>c</td>
<td>1</td>
<td>0x01</td>
</tr>
</tbody>
</table>
Exercise 1-A (cont’)

(performing Build_BDD (f_a))

• Having finished with Build_BDD (f_ab') the next step is calling Unique_Table from within the recursion hierarchy level of Build_BDD (f_a).

• Because we in the case of HSON==LSON nothing is need to be done except returning HSON=0x02. That is again no new line in the Unique_Table is needed, one can only add additional label to the line 0x02.

<table>
<thead>
<tr>
<th>Bool’ exp’</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>d; f_{ab}c'</td>
<td>0x01</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c+d</td>
<td>f_a; f_{ab}; f_{ab}'</td>
<td>0x02</td>
<td>c</td>
<td>1</td>
<td>0x01</td>
</tr>
</tbody>
</table>
Exercise 1-A (cont’)

(performing Build_BDD (f ) )

• We return to the top recursion hierarchy of f itself. We have finished with all of the recursion tree under $f_a = c + d$, and we know that $v = a$ and HSON=0x02. (& we know $f_a = c + d; f_a' = bc + d$)

• It is now time to compute LSON by further recursion:

(performing Build_BDD (f_a') )

• After cofactoring of $f_a'$ w.r.t. $v = b$ we find that $f_a'b = c + d$ and $f_a'b' = d$ both of which were calculated before, so their HSON and LSON are already known: HSON=0x02, LSON=0x01.

• Humans can add more labels.

<table>
<thead>
<tr>
<th>Bool’ exp’</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>$d; f_{abc}; f_{a'b'}$</td>
<td>0x01</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$c+d$</td>
<td>$f_a; f_{ab}; f_{ab'}; f_{a'b}$</td>
<td>0x02</td>
<td>c</td>
<td>1</td>
<td>0x01</td>
</tr>
</tbody>
</table>

Build_BDD (F) {       // F= $f = (a + b) \cdot c + d$
    If ( terminal = Terminal_Case( F ) ) {        return terminal;
} else {
    v = Top_Variable( F );       // v=a
    F_v = Co_Factor( F , v );       // F_v = f_a = c+d
    F_v = Co_Factor( F , v' );       // F_v = f_a' = bc+d
    HSON = Build_BDD (F_v );
    LSON = Build_BDD (F_v );
    return Unique_Table( v , HSON , LSON );
} }
Exercise 1-A (cont’)

(performing `Build_BDD(f_a')`)

- The next step is to call `Unique_Table` from within the recursion hierarchy level of `Build_BDD(f_a')`, where v=b and HSON=0x02, LSON=0x01.
- This creates a new Key entry in the `Unique_Table` \{b,0x02,0x01\}:
- Humans can add the label $f_a'$ to this newly created line 0x03.

<table>
<thead>
<tr>
<th>Bool' exp'</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>d; $f_{abc}$; $f_{a'b}$</td>
<td>0x01</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c+d</td>
<td>$f_a$; $f_{ab}$; $f_{ab'}$; $f_{a'b}$</td>
<td>0x02</td>
<td>c</td>
<td>1</td>
<td>0x01</td>
</tr>
<tr>
<td>bc+d</td>
<td>$f_a'$</td>
<td>0x03</td>
<td>b</td>
<td>0x02</td>
<td>0x01</td>
</tr>
</tbody>
</table>
Humans can draw the BDD represented in the Unique_Table: `Simple` or `Elaborated` →

<table>
<thead>
<tr>
<th>Bool' exp'</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>d; f_{abc}'; f_{a'b'}</td>
<td>0x01</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c+d</td>
<td>f_a; f_{ab}; f_{ab'}; f_{a'b}</td>
<td>0x02</td>
<td>c</td>
<td>1</td>
<td>0x01</td>
</tr>
<tr>
<td>bc+d</td>
<td>f_a'</td>
<td>0x03</td>
<td>b</td>
<td>0x02</td>
<td>0x01</td>
</tr>
<tr>
<td>(a+b)c+d</td>
<td>f</td>
<td>0x04</td>
<td>a</td>
<td>0x02</td>
<td>0x03</td>
</tr>
</tbody>
</table>
Exercise 1-B

(performing Build_BDD (g ) )

• Now we need to build the BDD for the function \( g = ac' + d \).
• Obviously \( g \) is not a Terminal_Case so we break it into coffactors \( g_a, g_a' : \ g_a=c'+d \; \; g_a'=d \)
• and enter recursively to compute \( g_a \).

(performing Build_BDD (g_a ) )

• As \( g_a \) doesn’t depend on \( b \) the recursion would yield the same for \( g_a=g_{ab}=g_{ab}' \), so we (perform Build_BDD (g_{ab} ) )

<table>
<thead>
<tr>
<th>Bool’ exp’</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d )</td>
<td>( d; f_{abc}; f_{a'b} )</td>
<td>0x01</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( c+d )</td>
<td>( f_a; f_{ab}; f_{ab'}; f_{a'b} )</td>
<td>0x02</td>
<td>c</td>
<td>1</td>
<td>0x01</td>
</tr>
<tr>
<td>( bc+d )</td>
<td>( f_a' )</td>
<td>0x03</td>
<td>b</td>
<td>0x02</td>
<td>0x01</td>
</tr>
<tr>
<td>( (a+b)c+d )</td>
<td>( f )</td>
<td>0x04</td>
<td>a</td>
<td>0x02</td>
<td>0x03</td>
</tr>
</tbody>
</table>
Build_BDD (F) {  // F= g_{ab} = c' + d
    If ( terminal = Terminal_Case( F ) ) {
        return terminal;
    } else {
        v = Top_Variable( F );  // v=c
        F_v = Co_Factor( F , v );  // F_v = g_{abc} = d
        F_{v'} = Co_Factor( F , v' );  // F_{v'} = g_{abc'} = 1
        HSON = Build_BDD (F_v);
        LSON = Build_BDD (F_{v'});
        return Unique_Table( v , HSON , LSON );
    }
}

Exercise 1-B (cont’)

(performing Build_BDD (g_{ab}.) )

- As $g_{ab} = c' + d$ and is not a Terminal_Case we break it into cofactors and find $g_{abc} = d$; $g_{abc'} = 1$ which are already computed and a Terminal_Case respectively, i.e. we found that HSON=0x01 and LSON=1. With v=c, the new Key={c,0x01,1}:
- Humans make labels and make a shortcut to $g_a$:

<table>
<thead>
<tr>
<th>Bool’ exp’</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>$d; f_{abc'}; f_{a'b'}$</td>
<td>0x01</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c+d</td>
<td>$f_a; f_{ab}; f_{ab'}; f_{a'b}$</td>
<td>0x02</td>
<td>c</td>
<td>1</td>
<td>0x01</td>
</tr>
<tr>
<td>bc+d</td>
<td>$f_a'$</td>
<td>0x03</td>
<td>b</td>
<td>0x02</td>
<td>0x01</td>
</tr>
<tr>
<td>(a+b)c+d</td>
<td>$f$</td>
<td>0x04</td>
<td>a</td>
<td>0x02</td>
<td>0x03</td>
</tr>
<tr>
<td>c’+d</td>
<td>$g_a; g_{ab}; g_{ab'}$</td>
<td>0x05</td>
<td>c</td>
<td>0x01</td>
<td>1</td>
</tr>
</tbody>
</table>
Exercise 1-B (cont’)

(performing `Build_BDD(g)`)

- We now finished with all the recursion sub-tree on the \(g_a\) side.
- We are back to the recursion level of `Build_BDD(g)` and now is
  the time to compute LSON but as we found earlier that \(g_a' = d\) it
  is clear that LSON=0x01 (another human label added).
- Currently \(v=a\) so the call to `Unique_Table` creates a new entry
  with the Key:

```plaintext
{ a, 0x05, 0x01 }
```

<table>
<thead>
<tr>
<th><code>Bool’ exp’</code></th>
<th><code>Label(s)</code></th>
<th>ID</th>
<th><code>v</code></th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d)</td>
<td>(d; f_{abc}; f_{a'b'}; g_a')</td>
<td>0x01</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(c+d)</td>
<td>(f_a; f_{ab}; f_{ab'}; f_{a'b})</td>
<td>0x02</td>
<td>c</td>
<td>1</td>
<td>0x01</td>
</tr>
<tr>
<td>(bc+d)</td>
<td>(f_a')</td>
<td>0x03</td>
<td>b</td>
<td>0x02</td>
<td>0x01</td>
</tr>
<tr>
<td>((a+b)c+d)</td>
<td>(f)</td>
<td>0x04</td>
<td>a</td>
<td>0x02</td>
<td>0x03</td>
</tr>
<tr>
<td>(c'+d)</td>
<td>(g_a; g_{ab}; g_{ab'})</td>
<td>0x05</td>
<td>c</td>
<td>0x01</td>
<td>1</td>
</tr>
<tr>
<td>(ac'+d)</td>
<td>(g)</td>
<td>0x06</td>
<td>a</td>
<td>0x05</td>
<td>0x01</td>
</tr>
</tbody>
</table>
Exercise 1-B (cont’)

Humans can draw the BDD represented in the Unique_Table:

Simple or Elaborated

<table>
<thead>
<tr>
<th>Bool’ exp’</th>
<th>Label(s)</th>
<th>ID</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>d; f_{abc}; f_{a’b’}; g_{a’}</td>
<td>0x01</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c+d</td>
<td>f_{a}; f_{ab}; f_{ab’}; f_{a’b}</td>
<td>0x02</td>
<td>c</td>
<td>1</td>
<td>0x01</td>
</tr>
<tr>
<td>bc+d</td>
<td>f_{a’}</td>
<td>0x03</td>
<td>b</td>
<td>0x02</td>
<td>0x01</td>
</tr>
<tr>
<td>(a+b)c+d</td>
<td>f</td>
<td>0x04</td>
<td>a</td>
<td>0x02</td>
<td>0x03</td>
</tr>
<tr>
<td>c’+d</td>
<td>g_{a}; g_{ab}; g_{ab’}</td>
<td>0x05</td>
<td>c</td>
<td>0x01</td>
<td>1</td>
</tr>
<tr>
<td>ac’+d</td>
<td>g</td>
<td>0x06</td>
<td>a</td>
<td>0x05</td>
<td>0x01</td>
</tr>
</tbody>
</table>
Exercise 1-C

The function is: \( h = f + g \)

First we’ll decompose \( h \) (some of functions):

\[
\begin{align*}
    h &= a(f_a + g_a) + a'(f'_a + g'_a) \\
    h_a &= f_a + g_a = c + d + c' + d = 1 \\
    h_a' &= f_a' + g_a' = b \cdot c + d + d = b \cdot c + d = f_a'.
\end{align*}
\]

Meaning we already have the sums in the unique table.

We have to build the new node:
Exercise 1-C (h)

Build bdd and **UT** entry for: $h = f + g$

Simple BDD add cofactors:

<table>
<thead>
<tr>
<th>Label</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>b</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>c</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>f_{ab}</td>
<td>c</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>f_{a'}</td>
<td>b</td>
<td>f_{ab}</td>
<td>D</td>
</tr>
<tr>
<td>f</td>
<td>a</td>
<td>f_{ab}</td>
<td>f_{a'}</td>
</tr>
<tr>
<td>g_{ab}</td>
<td>c</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>a</td>
<td>g_{ab}</td>
<td>D</td>
</tr>
<tr>
<td>h</td>
<td>a</td>
<td>1</td>
<td>30</td>
</tr>
</tbody>
</table>
Exercise 1-C (h)

Build bdd and UT entry for: \( h = f + g \)

Simple BDD add cofactors:

<table>
<thead>
<tr>
<th>Label</th>
<th>v</th>
<th>Hson</th>
<th>Lson</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>b</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>c</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( f_{ab} )</td>
<td>c</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>( f_{a^c} )</td>
<td>b</td>
<td>( f_{ab} )</td>
<td>D</td>
</tr>
<tr>
<td>( f )</td>
<td>a</td>
<td>( f_{ab} )</td>
<td>( f_{a^c} )</td>
</tr>
<tr>
<td>( g_{ab} )</td>
<td>c</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>( g )</td>
<td>a</td>
<td>( g_{ab} )</td>
<td>D</td>
</tr>
<tr>
<td>( h )</td>
<td>a</td>
<td>1</td>
<td>( f_{a^c} )</td>
</tr>
</tbody>
</table>

Partial BDD:
ROBDD with Complement Edges

Consider the two following BDDs:

- These are two different BDDs.
- There is no need to store them both!
- We can store just one, with a polarity value.
- BDD building recursion can be stopped if $F'$ is found (not only if $F$ is found).
• The **dot** notation:
• The **dot** means: “invert the final result you get”.
• The **dot** can be only on “**else**” edge to keep the BDD canonical.
• Using complement edges reduces the number of BDDs, but increases the data and work on the remaining BDDs.
• Not very intuitive - hard to draw and read.
Exercise: Build ROBDD with Complement Edges

\[ f = \text{abcd}' + \text{ab'd} + \text{a'c} + \text{a'c'd} \]
Exercise: Build ROBDD with Complement Edges

\[ f = abd' + ab'd + a'c + a'c'd \]
Exercise: Build ROBDD with Complement Edges

\[ f = abd' + ab'd + a'c + a'c'd \]
Exercice: Build ROBDD with Complement Edges

\[ f = abd' + ab'd + a'c + a'c'd \]
Exercise: Build ROBDD with Complement Edges

\[ f = abd' + ab'd + a'c + a'c'd \]
Exercise: Build ROBDD with Complement Edges

\[ f = abd' + ab'd + a'c + a'c'd \]

Try to manually verify the final BDD…