Boolean Functions representations

Paolo PRINETTO
Politecnico di Torino (Italy)
University of Illinois at Chicago, IL (USA)
Paolo.Prinetto@polito.it
prinetto@uic.edu
www.testgroup.polito.it
www.comitato-girotondo.org
This lecture presents most of the methods today used to represent Boolean Functions, pointing out, for each of them, pro’s and con’s, and their typical application areas.
Prerequisites

- Lecture 3.4
**Homework**

- No particular homework is foreseen
Further readings

- No particular suggestion
Boolean Functions are mostly represented by stating their on-set and don’t-care set.

These can, in turn, be represented according to several criteria, each particularly suited for a target application.
Boolean Function representation
Boolean Function representation

Exhaustive Representations

Compact Representations

ITE Expressions

Graphs

Boolean Function
An example

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$s_1' s_2' + s_1 s_3' + s_1 s_2$

Function $f$

Input $i$: 3
Output $o$: 1
00: 1
10: 1
11: 1
End

Slide # 3-5.9
© Prinetto - 2008
Problems to face

- Check whether two representations are equivalent (equivalence checking)
Problems to face

- Check whether two representations are equivalent (equivalence checking)
- Find an equivalent representation having “minimum cost” (minimization)
Problems to face

• Check whether two representations are equivalent (equivalence checking)
• Find an equivalent representation having “minimum cost” (minimization)
• Guarantee that, for each representation method, each function have a unique representation (canonicity).
Caveat

The three above problems have different complexity and different solutions for each representation method.
Boolean Function representation

Exhaustive Representations

Compact Representations

ITE Expressions

Graphs

Boolean Function

Exhaustive Representations

Compact Representations

ITE Expressions

Graphs

Boolean Function
Exhaustive representations explicitly represent the values assumed by the function for each of the possible combinations of the input variables $\Rightarrow 2^n$ entries !!!
Boolean Function

- Truth tables
- Karnaugh maps
- Exhaustive Representations
- Compact Representations
- ITE Expressions
- Graphs
Boolean Function

Exhaustive Representations

Truth tables

Karnaugh maps

Compact Representations

ITE Expressions

Graphs
The **Truth Table** specifies the value the function gets for each input combination:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>f(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Usage

- Sometimes helpful in manual design, as preliminary step to generate the equivalent Karnaugh maps.
Karnaugh maps, first introduced by Maurice Karnaugh in the ’50s, are an alternative representation of truth tables.

An n input variable boolean function is represented by a matrix of $2^n$ cells.
Karnaugh maps (cont’d)

Each cell is identified by binary row-column coordinates formed from the combination of the input variables:
**Karnaugh maps (cont’d)**

Each cell shows the value of the function when its input variables get the values of the corresponding row and column:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>f (x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

![Karnaugh map diagram]
Karnaugh maps (cont’d)

Values are assigned to rows and columns in such a way that *logically adjacent cells* (see slides # 70-77 of Lecture 3.4) be also physically adjacent.

Rows and columns are thus usually labeled to follow the reflected Gray code sequence:

```
  00 01 11 10
  0   1   
  1   
```

```
  a
  b
  c
```

© Prinetto - 2008
3 input maps

\[
\begin{array}{cccc}
\text{c} & 00 & 01 & 11 & 10 \\
\hline
0 & & & & \\
1 & & & & \\
\end{array}
\]
3 input maps

Logically adjacent columns
4 input maps
4 input maps

Logically adjacent rows

Logically adjacent columns
## 5 input maps

<table>
<thead>
<tr>
<th>de</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Logically adjacent columns**
“Overlapping” maps

c = 1

c = 0
Representation by Karnaugh maps
An example

<table>
<thead>
<tr>
<th>( s_3 )</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

function \( f \)
Usage

• Karnaugh maps are used just in manual design, when the # of input variables is very small (<6).
On a Karnaugh map, a k-cube corresponds to a set of $2^k$ logically adjacent cells, i.e., to a set of $2^k$ cells such that the set of combinations of the corresponding $2^k$ input variable satisfies the conditions to represent a k-cube (Lecture 3-4, slide 11):

- $n-k$ input variables must get a same constant value
- $k$ input variables assume all the $2^k$ possible combinations of values,
Examples

Maps for 3 variables

1-cube $\Rightarrow$ $K=1 \Rightarrow 2^k = 2^1 = 2$ adjacent cells
Maps for 3 variables

2-cube $\Rightarrow K=2$ $\Rightarrow 2^k = 2^2 = 4$

adjacent cells
Maps for 4 variables

1-cube $\Rightarrow K=1 \Rightarrow 2^k = 2^1 = 2$

adjacent cells
Maps for 4 variables

2-cube  ⇒  K=2  ⇒  $2^k = 2^2 = 4$ adjacent cells
Maps for 4 variables

2-cube $\Rightarrow K=2 \Rightarrow 2^k = 2^2 = 4$

adjacent cells
Maps for 4 variables

3-cube \( \Rightarrow \) \( K=3 \) \( \Rightarrow \) \( 2^k = 2^3 = 8 \)

adjacent cells
Maps for 5 variables

1-cube ⇒ K=1 ⇒ \(2^k = 2^1 = 2\) adjacent cells
Maps for 5 variables
2-cube $\Rightarrow K=2$ $\Rightarrow$
$2^k = 2^2 = 4$ adjacent cells
Maps for 5 variables

2-cube $\Rightarrow K=2 \Rightarrow 2^k = 2^2 = 4$ adjacent cells
Boolean Function representation

Exhaustive Representations

Compact Representations

ITE Expressions

Graphs

Boolean Function
Compact Representations

Compact representations represent a function $f$ resorting to another function $C$ that is a cover of $f$, i.e., by a function that includes all the vertices of the on-set and no vertex of the off-set of $f$:

$$\text{ons}(f) \subseteq C \subseteq \text{ons}(f) \cup \text{dcs}(f)$$
Boolean Function representation

Exhaustive Representations

Compact Representations

Boolean Expressions

Set of Cubes

ITE Expressions

Graphs
The function is represented by a **Boolean Expressions**, i.e., by a proper set of cubes, each represented according to the **Algebraic notation** (see Lecture 3-4, slides 43 on).
An example

\[
\begin{array}{c|c|c|c}
  & 00 & 01 & 11 & 10 \\
\hline
 0 & 1 & 0 & 1 & 1 \\
 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[s_1 s_2 + s_1 s_3' + s_1 s_2\]

**function** \(f\)
Caveat

A Boolean function can be represented by an infinite # of Boolean expressions
The following Boolean expressions all represent the same Boolean function:

- $a + x$
- $a \cdot x' + x$
- $a + a' \cdot x$
- $a + a + x$
- $(a + x) \cdot a + a' \cdot x$
Usage

Boolean expressions are typically used in manual handling (and, in particular, in manual synthesis), and in literature (both textbooks and papers).
Alternatives representation

• According to the Algebraic notation, a cube can be represented:
  – as *Product term* or
  – as a *Sum term*
Alternatives representation

- According to the Algebraic notation, a cube can be represented:
  - as a Product term or
  - as a Sum term

When cubes are represented as Product terms, the expressions get the form of logical sums (OR, “+”) of product terms, and are usually referred to as *Sum-Of-Product (SOP) expressions*
Alternatives representation

• According to the Algebraic notation, a cube can be represented:
  - as Product term or
  - as a Sum term

Example:

\[ f = x'yz' + xz \]

When cubes are represented as Product terms, the expressions get the form of logical sums (OR, “+”) of product terms, and are usually referred to as Sum-Of-Product (SOP) expressions.
Alternatives representation

- According to the Algebraic notation, a cube can be represented:
  - as Product term or
  - as a Sum term

When cubes are represented as Sum terms, the expressions get the form of logical product (AND) of sum terms, and are usually referred to as Product-of-sums (POS) expressions.
Alternatives representation

- According to the Algebraic notation, a cube can be represented:
  - as Product term
  - as a Sum term

Example:

\[ f = (x+y+z')(x'+y') \]

When cubes are represented as Sum terms, the expressions get the form of logical product (AND) of sum terms, and are usually referred to as Product-of-sums (POS) expressions.
Remarks

In the sequel of the course, for sake of simplicity, but without any loss in generality, only Sum-Of-Product expressions will be used.
Equivalence checking with Boolean Expressions

The equivalence between two boolean expressions can be verified:

- *manually*, resorting to a set of re-write rules, mainly based on postulates, properties and theorems of boolean algebras
- *automatically*, resorting to ad-hoc tools, such as *tautology checkers* or, more in general, *theorem provers*. 
Example of re-write rule application

Prove that:

\[(c' + abd + b'd + a'b)(c + ab + bd) = b(a + c)(a' + c') + d(b + c)\]
Solution

\[(c' + abd + b'd + a'b) (c + ab + bd) =\]
\[c' (ab + bd) + c (abd + b'd + a'b) =\]
\[abc' + bc'd + abcd + b'cd + a'bc =\]
\[abc' + abcd + bc'd + a'bc + b'cd =\]
\[ab (c' + cd) + bc'd + a'bc + b'cd =\]
\[ab (c' + d) + bc'd + a'bc + b'cd =\]
\[ab (c' + d) + bc'd + a'bc + a'bd + b'cd =\]
\[abc' + abd + bc'd + a'bc + a'bd + b'cd =\]
\[abd + a'bd) + abc' + bc'd + a'bc + b'cd =\]
\[ bd (a + a') + abc' + bc'd + a'bc + b'cd = \]
\[ bd (1) + abc' + bc'd + a'bc + b'cd = \]
\[ bd + abc' + bc'd + a'bc + b'cd = \]
\[ bd + bc'd + abc' + a'bc + b'cd = \]
\[ bd + abc' + a'bc + b'cd = \]
\[ d (b + b'c) + abc' + a'bc = \]
\[ d (b + c) + abc' + a'bc = \]
\[ d (b + c) + b (ac' + a'c) = \]
\[ d (b + c) + b (a + c) (a' + c') \]

q.e.d.
Equivalence checker

Equivalence checker: \( f_1 = f_2 \) ?

- \( f_1 := \text{boolean function of circuit #1} \)
- \( f_2 := \text{boolean function of circuit #2} \)
Canonicity with Boolean Expressions

An SOP expression is *canonical* iff is composed of minterms, only.
An SOP expression is *canonical* iff it is composed of minterms, only.

\[
f = x' y z' + x y' z + x y z\]

canonical
An SOP expression is *canonical* iff it is composed of minterms, only.

- **Canonical**: $f = x'yz' + x'yz + xyz$
  - $f = x'yz' + xz$
  - $f = x'yz' + xz$
- **Not Canonical**: $f = x'yz' + x'yz + xyz$

---

**Canonicity with Boolean Expressions**

Canonicity refers to the property of a Boolean expression where it is composed only of minterms. In an SOP (Sum of Products) form, if an expression contains only minterms, it is considered canonical. Expressions that include other terms besides minterms are not canonical.
Canonicity with Boolean Expressions (cont’d)

A POS expression is canonical iff is composed of maxterms, only.
Canonicity with Boolean Expressions (cont’d)

A POS expression is *canonical* iff is composed of maxterms, only.

\[ f = (x+y+z') \]
\[ (x'+y'+z) \]
\[ (x+y'+z') \]
Canonicity with Boolean Expressions (cont’d)

A POS expression is *canonical* iff it is composed of maxterms, only.

\[ f = (x+y+z') (x'+y') \]
not canonical

\[ f = (x+y+z') (x'+y'+z) (x+y'+z') \]
canonical
Boolean Function

- Boolean Expressions
- Graphs
- Exhaustive Representations
- Compact Representations
- Set of Cubes
- ITE Expressions
The function is represented by a proper set of cubes, each represented according to the **Cubic notation** (see Lecture 3-4, slides 25 on).
ons = \{ 000, 010, 110, 100 \} = \{ - - 0 \}
ofs = \{ 001, 101 \} = \{ - 0 1 \}
dcs = \{ 011, 111 \} = \{ - 1 1 \}
Usage

Set of cubes are typically used as inputs for EDA tools.
An example

\[ s_1 s_2 \\
\hline
s_3 & 00 & 01 & 11 & 10 \\
0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 \\
\]

\[ s_1' s_2' + s_1 s_3' + s_1 s_2 \]

\textbf{function } f \textbf{ }

\[
\begin{align*}
.i & 3 \\
.o & 1 \\
00 & 1 \\
10 & 1 \\
11 & 1 \\
.e &
\end{align*}
\]
ITE Expressions

A possible representation for Boolean functions consists of if-then-else (ITE, for short) expressions.

Each function is represented by a sequence of nested if-then-else statements.
**Definition of ITE**

The *ITE* representation for a Boolean function is built from the following syntactic rules:

\[
\text{expr ::= 0 | 1 | v | ITE(expr_1, expr_2, expr_3)}
\]

where:

- 0 and 1 are the Boolean constants
- \(v\) is a single Boolean variable
- ITE(a,b,c) stands for
  “if a evaluates to 1 (is true), then the value of the expression is that of b, else it is that of c”
Examples

\[ a \cdot b \leftrightarrow \text{ITE}( a, b, 0 ) \]
Examples

\[ a \cdot b \leftrightarrow \text{ITE}(a, b, 0) \]

\[ a + b \leftrightarrow \text{ITE}(a, 1, b) \]
Examples

\[ a \cdot b \leftrightarrow \text{ITE}( a, b, 0 ) \]

\[ a + b \leftrightarrow \text{ITE}( a, 1, b ) \]

\[ a' \leftrightarrow \text{ITE}( a, 0, 1 ) \]
Examples

\[ a \cdot b \leftrightarrow \text{ITE}( a, b, 0 ) \]
\[ a + b \leftrightarrow \text{ITE}( a, 1, b ) \]
\[ a' \leftrightarrow \text{ITE}( a, 0, 1 ) \]
\[ a \oplus b \leftrightarrow \text{ITE}( a, \text{ITE}( b, 0, 1 ), b ) \]
Examples

\[ a \cdot b \leftrightarrow \text{ITE}( a, b, 0 ) \]

\[ a + b \leftrightarrow \text{ITE}( a, 1, b ) \]

\[ a' \leftrightarrow \text{ITE}( a, 0, 1 ) \]

\[ a \oplus b \leftrightarrow \text{ITE}( a, \text{ITE}( b, 0, 1 ), b ) \]

\[ (a+b) \cdot (c+d) \leftrightarrow \text{ITE}( \text{ITE}( a, 1, b ), \text{ITE}( c, 1, d ), 0 ) \]
Equivalence

The following identity holds:

\[
\text{ITE}( a, b, c ) = a \cdot b + a' \cdot c
\]

**Corollary:**
It is possible to mix \textit{ITE} expressions with usual Boolean operators, since each of the two forms can be expressed in terms of the other one.
ITE( a, a, b ) = ITE( a, 1, b )
ITE( a, b, a ) = ITE( a, b, 0 )
ITE( a, b, a’ ) = ITE( a, b, 1 )
ITE( a, a’, b ) = ITE( a, 0, b )

ITE( a, 1, b ) = ITE( b, 1, a )
ITE( a, b, 0 ) = ITE( b, a, 0 )
ITE( a, b, 1 ) = ITE( b’, a’, 1 )
ITE( a, 0, b ) = ITE( b’, 0, a’ )
ITE( a, b, b’ ) = ITE( b, a, a’ )

ITE( a, b, c ) = ITE( a’, c, b ) = ITE( a, b’, c’ )’ = ITE( a’, c’, b’ )’
Two-input functions

Any two-input boolean function can be expressed resorting to just one ITE expression.
Two-input functions

*Any* two-input boolean function can be expressed resorting to just one ITE expression.

In the following table, all two-input Boolean functions are enumerated and the corresponding ITE expressions are shown.
<table>
<thead>
<tr>
<th>truth table</th>
<th>name</th>
<th>expression</th>
<th>ITE expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0</td>
<td>0</td>
<td>ITE(f, g, 0)</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>AND</td>
<td>f \cdot g</td>
<td>ITE(f, g', 0)</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>&gt;</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>&lt;</td>
<td>f' \cdot g</td>
<td>f</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>EXOR</td>
<td>f \oplus g</td>
<td>g</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>OR</td>
<td>f + g</td>
<td>ITE(f, g, g')</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>NOR</td>
<td>(f + g)'</td>
<td>ITE(f, 0, g')</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>EXNOR</td>
<td>f \oplus g</td>
<td>ITE(f, g, g')</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td></td>
<td>(f + g)'</td>
<td>ITE(f, 1, g)</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>g'</td>
<td>g'</td>
<td>ITE(g, 0, 1)</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>≥</td>
<td>f + g'</td>
<td>ITE(f, g, 1)</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>f'</td>
<td>f'</td>
<td>ITE(f, 0, 1)</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>≤</td>
<td>f' + g</td>
<td>ITE(f, g, 1)</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>NAND</td>
<td>(f \cdot g)'</td>
<td>ITE(f, g', 1)</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>truth table</td>
<td>name</td>
<td>expression</td>
<td>ITE expression</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>------------</td>
<td>----------------</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>AND</td>
<td>f · g</td>
<td>ITE(f, g, 0)</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>&gt;</td>
<td>f · g’</td>
<td>ITE(f, g’, 0)</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td></td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td></td>
<td>f’ · g</td>
<td>ITE(f, 0, g)</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td></td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The column labeled “truth table” refers to the four values of the function corresponding to the four possible truth assignments of the input variables “f” and “g”: 00, 01, 10, 11, respectively.
Simple ITE expressions

An ITE expression is said to be *simple* when:

- it is a constant

or:

- it is of the form $\text{ITE}(a, b, c)$ and:
  - the first operand, $a$, is a variable (i.e., neither a constant, nor an expression itself)
  - the other operands, $b$ and $c$, are simple ITE expressions.

In the latter case, the variable $a$ is said to be the "splitting variable" for the expression $\text{ITE}(a, b, c)$. 
Atomic ITE expressions

An ITE expression is said to be *atomic* when:

- it is a constant

or:

- it is of the form \( \text{ITE}(a, b, c) \), and
  - it is *simple*
  - the *then-part*, \( b \), and the *else-part*, \( c \), are constants
  - \( b \neq c \).
Note

Atomic ITE expressions correspond to literals, only:

- $a = \text{ITE}(a, 1, 0)$
- $a' = \text{ITE}(a, 0, 1)$.

From the above definitions, it follows that the innermost ITE operators in any simple ITE expression must be atomic ITE expressions.
Usage

ITE expressions are typically used in BDD construction.
Equivalence checking

The several equivalences between different forms of ITE expressions, when used as powerful rewriting rules, make the check for equivalence between general boolean expressions much easier.
Boolean Function representation

Exhaustive Representations

Compact Representations

ITE Expressions

Graphs

Boolean Function

Representations
An example

<table>
<thead>
<tr>
<th>s₁ s₂</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₃ = 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>s₃ = 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$s₁' s₂' + s₁ s₃' + s₁ s₂$

Function $f$

Input: 3
Output: 1
00 → 1
10 → 1
11 → 1
01 → 0

Definition of BDD

Given a boolean function $f$ of $n$ boolean variables,

$$f : B^n \rightarrow B,$$

the Binary Decision Diagram (BDD) associated to $f$ is the binary Directed Acyclic Graph (DAG) corresponding to a simple ITE expression representing $f$. 
DAG construction

The DAG is built as follows:

- for each constant 0 or 1 a leaf node is created
- for each ITE operator a node is created
- the node is labeled with the splitting variable
- two outgoing edges are created, labeled with the two boolean constants 0 and 1
- the edge labeled as 0 points to a node representing the else-part, the edge labeled as 1 points to the then-part.
Example

\[ a \cdot b = \text{ITE}(a, b, 0) \]

\[ a \cdot b = \text{ITE}(a, \text{ITE}(b, 1, 0), 0) \]

Simple ITE

General ITE

BDD
Convention

Whenever possible, drawings is simplified by adopting the following conventions:

- edges are directed from top to bottom
- the 0-labeled edge is the leftmost one.
Function evaluation by BDDs

Given an assignment to the input variables, the value of function $f$ can be found by traversing the BDD, starting from the root, choosing at each step the right or the left branch according to the value of the current variable, until a leaf is reached.

The value of the leaf is the value of the constant it is labeled with.
Малые Автюхи, Калинковичский район, Республики Беларусь