Logic Synthesis and Verification

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Fall 2010

Boolean Function Representation & Reasoning

Reading:
Logic Synthesis in a Nutshell
Section 2

most of the following slides are by courtesy of Andreas Kuehlmann

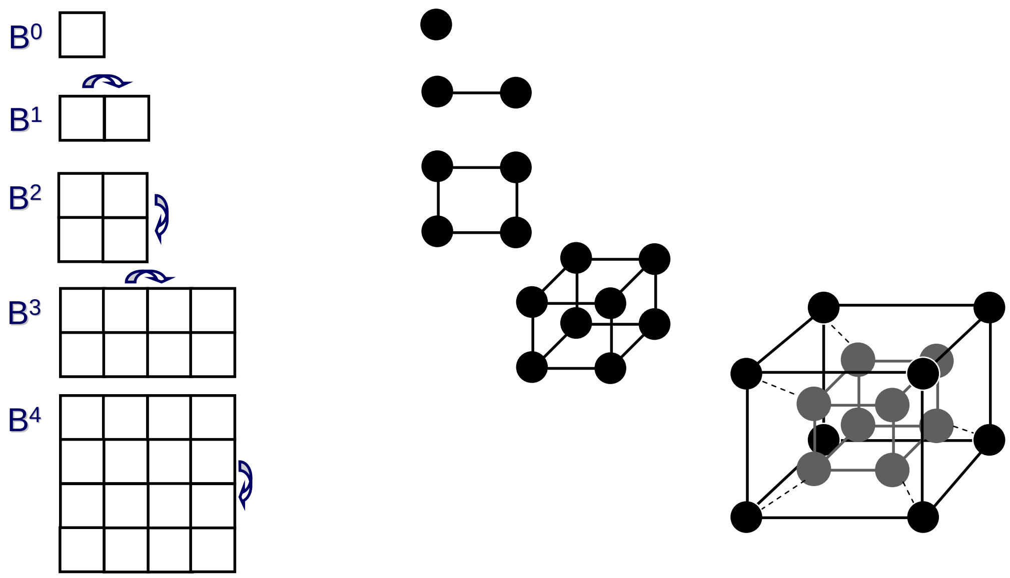
Assumption

- Unless otherwise said, from now on we are concerned with two-element Boolean algebra (i.e. **B** = {0,1})

Boolean Space

- B = {0,1}
- B^2 = {0,1}×{0,1} = {00, 01, 10, 11}

Karnaugh Maps: Boolean Lattices:



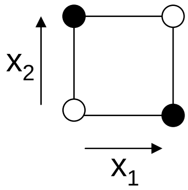
Boolean Function

- For **B** = {0,1}, a Boolean function f: $\mathbf{B}^n \rightarrow \mathbf{B}$ over variables $x_1,\ldots,x_n$ maps each Boolean valuation (truth assignment) in $\mathbf{B}^n$ to 0 or 1

Example

$f(x_1,x_2)$ with f(0,0) = 0, f(0,1) = 1, f(1,0) = 1, f(1,1) = 0

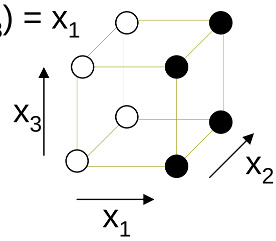
	x_2 = 0	x_2 = 1
x_1 = 0	0	1
x_1 = 1	1	0



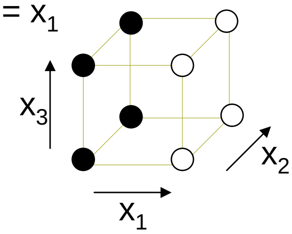
Boolean Function

- Onset of f, denoted as f^1, is $f^1 = \{v \in \mathbf{B}^n \mid f(v)=1\}$
  - If $f^1 = \mathbf{B}^n$, f is a tautology
- Offset of f, denoted as f^0, is $f^0 = \{v \in \mathbf{B}^n \mid f(v)=0\}$
  - If $f^0 = \mathbf{B}^n$, f is unsatisfiable. Otherwise, f is satisfiable.
- f^1 and f^0 are sets, not functions!
- Boolean functions f and g are equivalent if $\forall v \in \mathbf{B}^n$. f(v) = g(v) where v is a truth assignment or Boolean valuation
- A literal is a Boolean variable *x* or its negation x′ (or x, ¬x) in a Boolean formula

$f(x_1, x_2, x_3) = x_1$

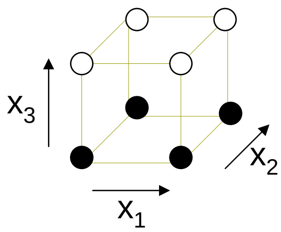


$f(x_1, x_2, x_3) = \overline{x_1}$



Boolean Function

- There are 2^n vertices in $\mathbf{B}^n$
- There are 2^{2^n} distinct Boolean functions
  - Each subset $f^1 \subseteq \mathbf{B}^n$ of vertices in $\mathbf{B}^n$ forms a distinct Boolean function f with onset f^1



$x_1x_2x_3$		f
0 0 0		1
0 0 1		0
0 1 0		1
0 1 1		0
1 0 0	$\Rightarrow$	1
1 0 1		0
1 1 0		1
1 1 1		0

Boolean Operations

Given two Boolean functions:

$f : \mathbf{B}^n \rightarrow \mathbf{B}$

$g : \mathbf{B}^n \rightarrow \mathbf{B}$

- $h = f \wedge g$ from AND operation is defined as
  $h^1 = f^1 \cap g^1$; $h^0 = \mathbf{B}^n \setminus h^1$
- $h = f \vee g$ from OR operation is defined as
  $h^1 = f^1 \cup g^1$; $h^0 = \mathbf{B}^n \setminus h^1$
- $h = \neg f$ from COMPLEMENT operation is defined as
  $h^1 = f^0$; $h^0 = f^1$

Cofactor and Quantification

Given a Boolean function:

$f : \mathbf{B}^n \rightarrow \mathbf{B}$, with the input variable $(x_1,x_2,\ldots,x_i,\ldots,x_n)$

- Positive cofactor on variable x_i
  $h = f_{xi}$ is defined as $h = f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Negative cofactor on variable x_i
  $h = f_{\neg xi}$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n)$
- Existential quantification over variable x_i
  $h = \exists x_i.\ f$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n) \vee f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Universal quantification over variable x_i
  $h = \forall x_i.\ f$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n) \wedge f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Boolean difference over variable x_i
  $h = \partial f/\partial x_i$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n) \oplus f(x_1,x_2,\ldots,1,\ldots,x_n)$

Representation of Boolean Function

- Represent Boolean functions for two reasons
  - to represent and manipulate the actual circuit we are implementing
  - to facilitate Boolean reasoning
- Data structures for representation
  - Truth table
  - Boolean formula
    - Sum of products (Disjunctive "normal" form, DNF)
    - Product of sums (Conjunctive "normal" form, CNF)
  - Boolean network
    - Circuit (network of Boolean primitives)
    - And-inverter graph (AIG)
  - Binary Decision Diagram (BDD)

Boolean Function Representation Truth Table

- Truth table (function table for multi-valued functions):
  The truth table of a function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ is a tabulation of its value at each of the 2^n vertices of $\mathbf{B}^n$.

  In other words the truth table lists all mintems

  Example: f = a′b′c′d + a′b′cd + a′bc′d + ab′c′d + ab′cd + abc′d + abcd′ + abcd

  The truth table representation is
  - impractical for large *n*
  - canonical

  If two functions are the equal, then their canonical representations are isomorphic.

	abcd	f		abcd	f
0	0000	0	8	1000	0
1	0001	1	9	1001	1
2	0010	0	10	1010	0
3	0011	1	11	1011	1
4	0100	0	12	1100	0
5	0101	1	13	1101	1
6	0110	0	14	1110	1
7	0111	0	15	1111	1

Boolean Function Representation Boolean Formula

- A Boolean formula is defined inductively as an expression with the following formation rules (syntax):

formula ::=		‘(‘ formula ‘)’	
	|	Boolean constant	(**true** or **false**)
	|	<Boolean variable>	
	|	formula “+” formula	(OR operator)
	|	formula “·” formula	(AND operator)
	|	¬ formula	(complement)

Example

$f = (x_1 \cdot x_2) + (x_3) + \neg(\neg(x_4 \cdot (\neg x_1)))$

typically “·” is omitted and ‘(‘, ‘)’ and ‘¬’ are simply reduced by priority,

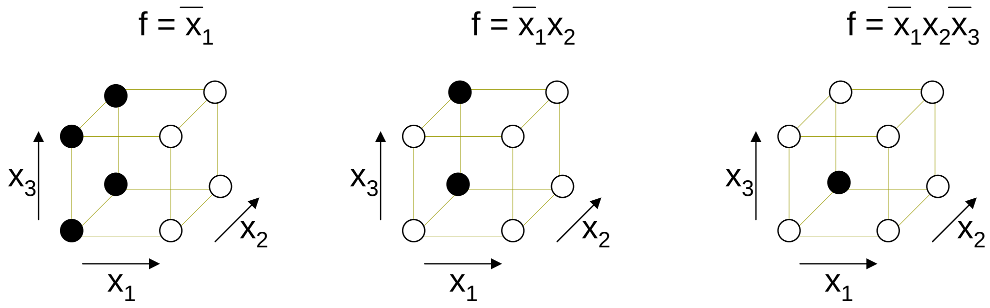
e.g. $f = x_1\, x_2 + x_3 + x_4 \neg x_1$

Boolean Function Representation
Boolean Formula in SOP

- A cube is defined as a conjunction of literals, i.e. a product term.

Example

$C = x_1x_2'x_3$ represents the function with onset: $f^1 = \{(x_1=1,x_2=0,x_3=1)\}$ in the Boolean space spanned by x_1,x_2,x_3, or $f^1 = \{(x_1=1,x_2=0,x_3=1, x_4=0), (x_1=1,x_2=0,x_3=1,x_4=1)\}$ in the Boolean space spanned by x_1,x_2,x_3,x_4, or …



Boolean Function Representation
Boolean Formula in SOP

- If $C \subseteq f^1$, C the onset of a cube c, then c is an implicant of f
- If $C \subseteq \mathbf{B}^n$, and c has k literals, then $|C| = 2^{n-k}$, i.e., C has 2^{n-k} elements

Example

$c = xy'$ $(c{:}\mathbf{B}^3 \rightarrow \mathbf{B})$, $C = \{100, 101\} \subseteq \mathbf{B}^3$

$k = 2$, $n = 3$ $\quad |C| = 2 = 2^{3-2}$

- An implicant with n literals is a minterm

Boolean Function Representation
Boolean Formula in SOP

- A function can be represented by a sum-of-cubes (products):

  $f = ab + ac + bc$

  Since each cube is a product of literals, this is a sum-of-products (SOP) representation or disjunctive normal form (DNF)

- An SOP can be thought of as a set of cubes F

  $F = \{ab, ac, bc\}$

- A set of cubes that represents f is called a cover of f.

  $F_1=\{ab, ac, bc\}$ and $F_2=\{abc, abc', ab'c, a'bc\}$

  are covers of

  $f = ab + ac + bc$.

- Mainly used in circuit synthesis; seldom used in Boolean reasoning

Boolean Function Representation
Boolean Formula in POS

- Product-of-sums (POS), or conjunctive normal form (CNF), representation of Boolean functions
  - Dual of the SOP representation

  Example

  $\varphi = (a+b'+c)\ (a'+b+c)\ (a+b'+c')\ (a+b+c)$

- A Boolean function in a POS representation can be derived from an SOP representation with De Morgan's law and the distributive law

- Mainly used in Boolean reasoning; rarely used in circuit synthesis (due to the asymmetric characteristics of NMOS and PMOS)

Boolean Function Representation
Boolean Network

- Used for two main purposes
  - as target structure for logic implementation which gets restructured in a series of logic synthesis steps until result is acceptable
  - as representation for Boolean reasoning engine

- Efficient representation for most Boolean problems
  - memory complexity is similar to the size of circuits that we are actually building

- Close to the input and output representations of logic synthesis

Boolean Function Representation
Boolean Network

A Boolean network is a directed graph C(G,N) where G are the gates and $N \subseteq (G \times G)$ are the directed edges (nets) connecting the gates.

Some of the vertices are designated:
Inputs: $I \subseteq G$
Outputs: $O \subseteq G$
$I \cap O = \emptyset$

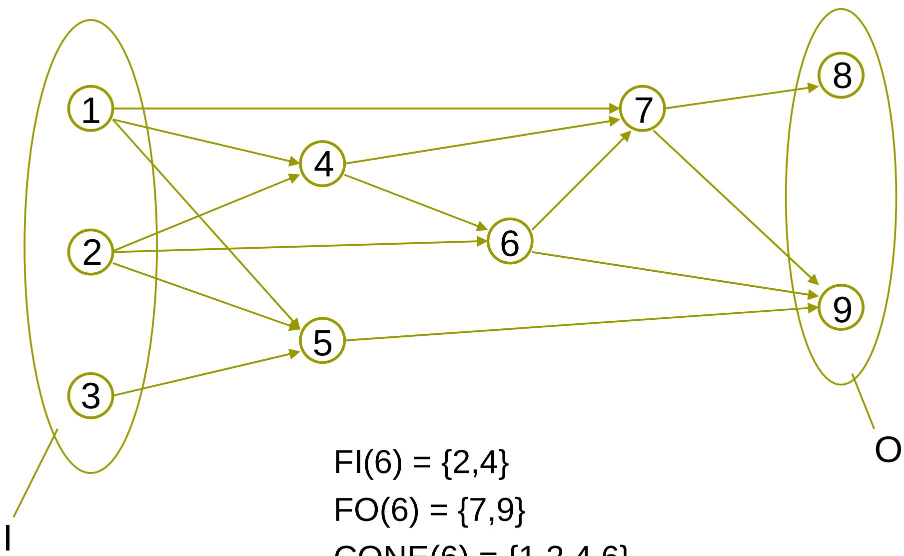
Each gate g is assigned a Boolean function f_g which computes the output of the gate in terms of its inputs.

Boolean Function Representation
Boolean Network

- The fanin FI(g) of a gate g are the predecessor gates of g: FI(g) = {g' | (g',g) ∈ N} (N: the set of nets)
- The fanout FO(g) of a gate g are the successor gates of g: FO(g) = {g' | (g,g') ∈ N}
- The cone CONE(g) of a gate g is the transitive fanin (TFI) of g and g itself
- The support SUPPORT(g) of a gate g are all inputs in its cone:
  SUPPORT(g) = CONE(g) ∩ I

Boolean Function Representation
Boolean Network

Example



FI(6) = {2,4}
FO(6) = {7,9}
CONE(6) = {1,2,4,6}
SUPPORT(6) = {1,2}

Boolean Function Representation
Boolean Network

- Circuit functions are defined recursively:

$$h_{g_i} = \begin{cases} x_i & \text{if } g_i \in I \\ f_{g_i}(h_{g_j}, \ldots, h_{g_k}),\ g_j, \ldots, g_k \in FI(g_i) & \text{otherwise} \end{cases}$$

If G is implemented using physical gates with positive (bounded) delays for their evaluation, the computation of h_g depends in general on those delays.

Definition

A circuit C is called combinational if for each input assignment of C for $t \to \infty$ the evaluation of h_g for all outputs is independent of the internal state of C.

Proposition

A circuit C is combinational if it is acyclic. (converse not true!)

Boolean Function Representation
Boolean Network

General Boolean network:

- Vertex can have an arbitrary finite number of inputs and outputs

- Vertex can represent any Boolean function stored in different ways, such as:
  - SOPs (e.g. in SIS, a logic synthesis package)
  - BDDs (to be introduced)
  - AIGs (to be introduced)
  - truth tables
  - Boolean expressions read from a library description
  - other sub-circuits (hierarchical representation)

- The data structure allows general manipulations for insertion and deletion of vertices, pins (connection ports of vertices), and nets
  - general but far too slow for Boolean reasoning

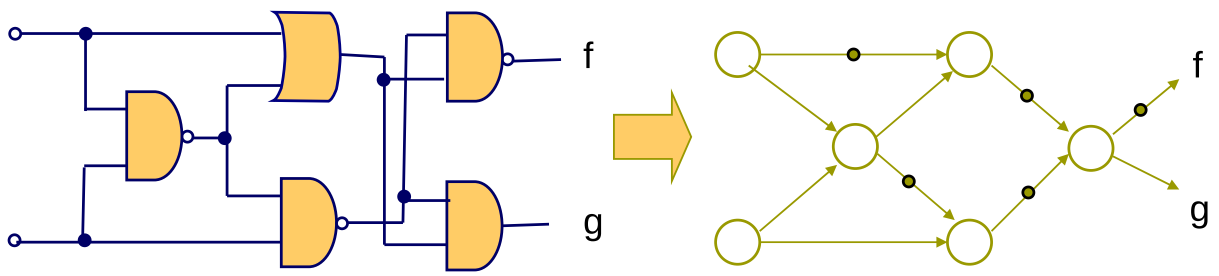
Boolean Function Representation
Boolean Network

Specialized Boolean network:

- Non-canonical representation in general
  - computational effort of Boolean reasoning is due to this non-canonicity (c.f. BDDs)
- Vertices have fixed number of inputs (e.g. two)
- Vertex function is stored as label (e.g. OR, AND, XOR)
- Allow on-the-fly compaction of circuit structure
  - Support incremental, subsequent reasoning on multiple problems

Boolean Function Representation
And-Inverter Graph

- AND-INVERTER graphs (AIGs)
  vertices: 2-input AND gates
  edges: interconnects with (optional) dots representing INVs
- Hash table to identify and reuse structurally isomorphic circuits

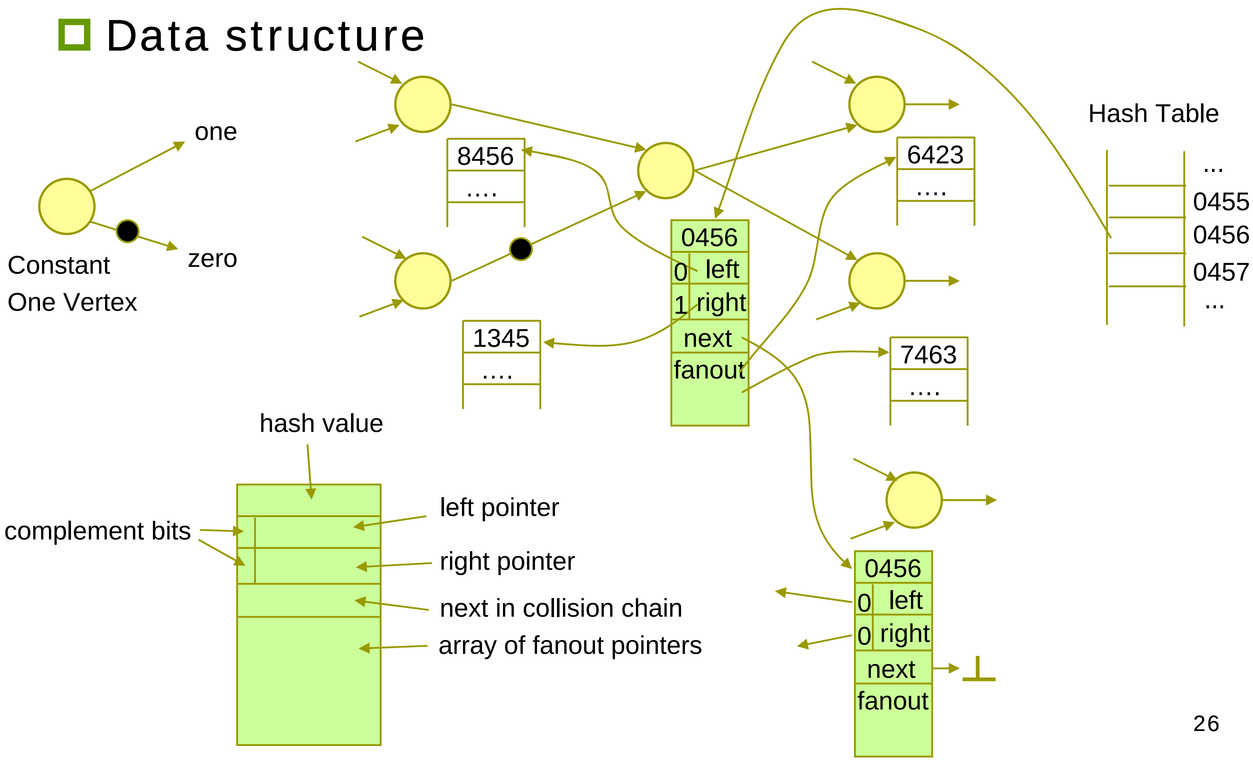


Boolean Function Representation
And-Inverter Graph

- Data structure for implementation
  - Vertex:
    - pointers (integer indices) to left- and right-child and fanout vertices
    - collision chain pointer
    - other data
  - Edge:
    - pointer or index into array
    - one bit to represent inversion
  - Global hash table holds each vertex to identify isomorphic structures
  - Garbage collection to regularly free un-referenced vertices

Boolean Function Representation
And-Inverter Graph

- Data structure



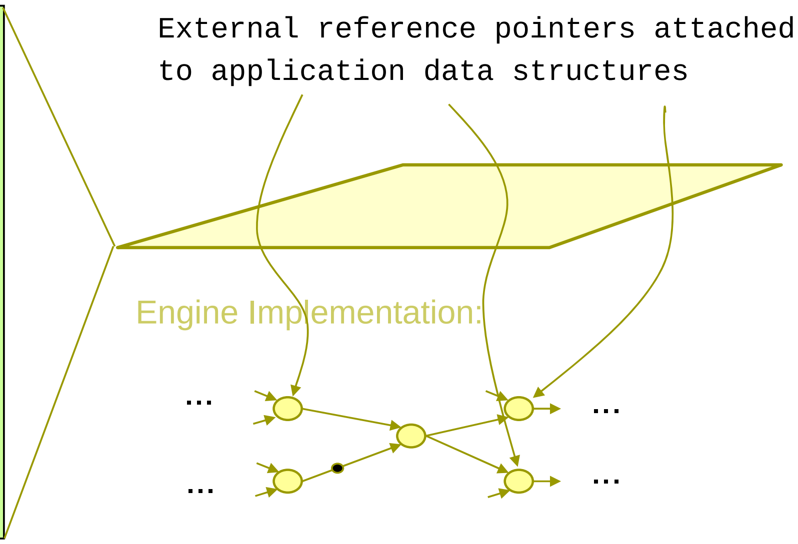
Boolean Function Representation
And-Inverter Graph

- AIG package for Boolean reasoning

  Engine application:

  - traverse problem data structure and build Boolean problem using the interface
  - call SAT to make decision

```
Engine Interface:
void INIT()
void QUIT()
Edge VAR()
Edge AND(Edge p1,
         Edge p2)
Edge NOT(Edge p1)
Edge OR(Edge p1
        Edge p2)
...
int  SAT(Edge p1)
```



Boolean Function Representation
And-Inverter Graph

- Hash table look-up

```
Algorithm HASH_LOOKUP(Edge p1, Edge p2) {
  index = HASH_FUNCTION(p1,p2)
  p     = &hash_table[index]
  while(p != NULL) {
   if(p->left == p1 && p->right == p2) return p;
   p = p->next;
  }
  return NULL;
}
```

- Tricks:
  - keep collision chain sorted by the address (or index) of p
  - use memory locations (or array indices) in topological order for better cache performance

Boolean Function Representation
And-Inverter Graph

- AND operation

```
AND(Edge p1,Edge p2){
  if(p1 == const1) return p2
  if(p2 == const1) return p1
  if(p1 == p2)     return p1
   if(p1 == ¬p2)    return const0
   if(p1 == const0 || p2 == const0) return const0

   if(RANK(p1) > RANK(p2)) SWAP(p1,p2)

   if((p = HASH_LOOKUP(p1,p2)) return p
   return CREATE_AND_VERTEX(p1,p2)
}
```


Boolean Function Representation
And-Inverter Graph

- NOT operation

```
NOT(Edge p) {
  return TOOGLE_COMPLEMENT_BIT(p)
}
```

- OR operation

```
OR(Edge p1,Edge p2){
  return (NOT(AND(NOT(p1),NOT(p2))))
}
```

Boolean Function Representation
And-Inverter Graph

- Cofactor operation

```
POSITIVE_COFACTOR(Edge p,Edge v){
  if(IS_VAR(p)) {
    if(p == v) {
      if(IS_INVERTED(v) == IS_INVERTED(p)) return const1
      else                                 return const0
  }
  else                                      return p
}
if((c = GET_COFACTOR(p,v)) == NULL) {
  left  = POSITIVE_COFACTOR(p->left, v)
  right = POSITIVE_COFACTOR(p->right, v)
  c = AND(left,right)
  SET_COFACTOR(p,v,c)
}
if(IS_INVERTED(p)) return NOT(c)
else               return c
}
```


Boolean Function Representation
And-Inverter Graph

- Similar algorithm for **NEGATIVE_COFACTOR**

- Existential and universal quantifications can be built from AND, OR and COFACTORS

Exercise: Prove $(f \cdot g)_v = f_v \cdot g_v$ and $(\neg f)_v = \neg(f_v)$

Question: What is the worst-case complexity of performing quantifications over AIGs?