

Unit 2: Equivalence Checking

Abstract

We address the problem of checking whether two combinational designs are equivalent. We first show that this problem is intimately connected to the notion of composition. We then prove some Boolean functions, which leads us to an algorithm for equivalence checking.

1 Composition

Big designs are made out of smaller design by composition.

- Connect some POs of $N1$ to PIs of $N2$, and vice versa.
- PI's of $N1$ and $N2$ which are not POs of others are PIs of new netlist
- Some POs of original are designated POs of the composition
- Resulting structure is required to be a netlist
 - How could it not be a netlist?

Example 1 (Composition) General idea: Figure 1.

Example 2 (Composing two netlists) Concrete example: Figure 2.

Why did we stress the absence of compositional cycles?

Example 3 (Combinational cycles) Intuition fails when you have combinational cycles.

Upshot: when you have cycles, can't characterize designs by languages.

Will never be an issue as far as we are concerned. (However, do lose some generality.) References: PhD writeups from Vigyan Singhal, Tom Shiple.

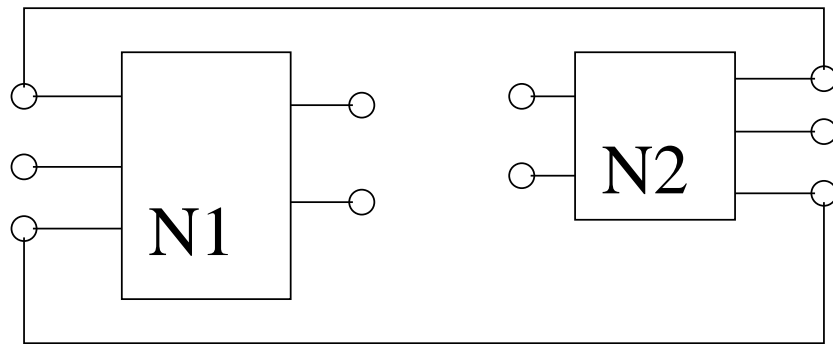


Figure 1: Netlist composition.

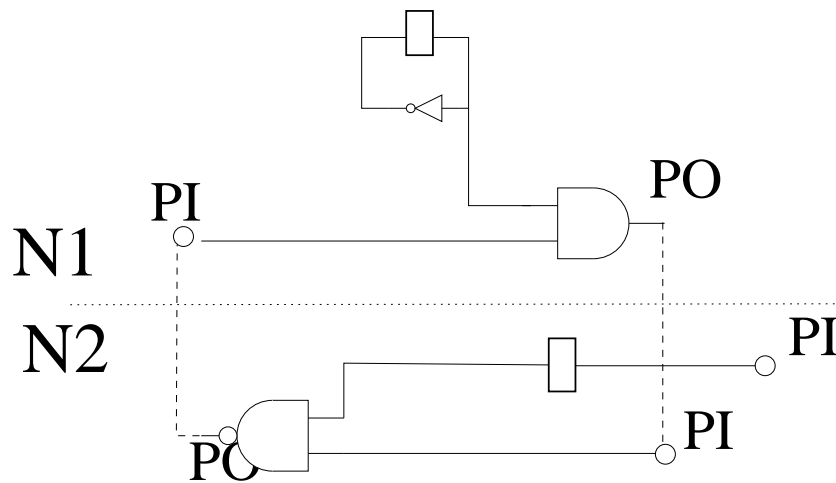


Figure 2: Netlist composition.

2 Alternate semantics

$$y_1 = F_1(u_1, u_2, u_3, x_1, x_2)$$

$$y_2 = F_2((u_1, u_2, u_3, x_1, x_2)$$

1. Asynchronous Interleaving: exactly one updates
2. Fully Asynchronous: either or both can change
3. Hybrid of synch/asynch

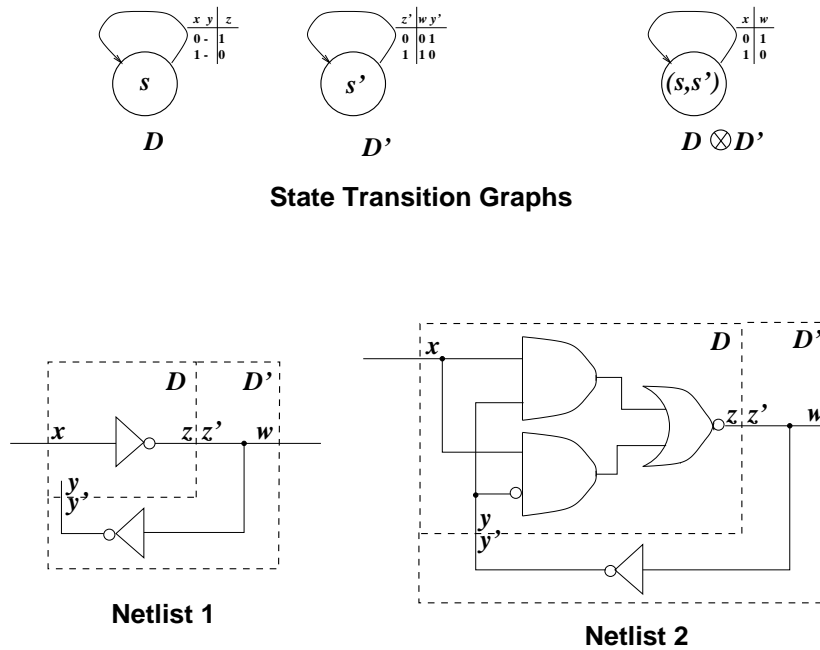


Figure 3: Seemingly equivalent designs.

Example 4 (Alternate semantics) See Figure 5.

3 Netlists—discussion

Limitations

- timing
- words, integers, complex functions
- synchronization

Advantages

- simple, “synthesizable”
- expressive, captures many designs you’d like to verify
- actually used internally (VIS, Synopsys’ design compiler)

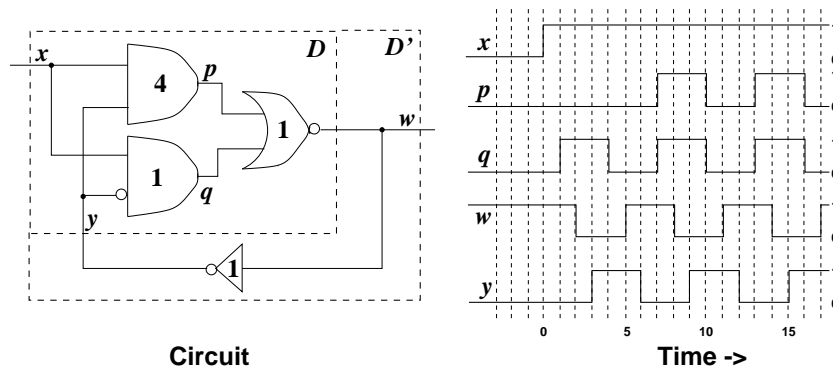


Figure 4: Bad things results when you allow combinational cycles.

4 Equivalence checking

How to check if two netlists are equivalent?

- Recall; η_1, η_2 are equivalent wrt initial states s_0, p_0 if $L_{s_0}^{\eta_1} = L_{p_0}^{\eta_2}$

Do we need to consider an infinite number of inputs sequences???!!!

Answer: no, can solve using finite state machines (will see later).

4.1 Equivalence for combinational netlists

First observe—can do on an output by output basis, so it suffices to consider single output netlists.

1. One approach: randomly simulate (incomplete)
2. Next approach: try all inputs (design has 50 PIs?)

Suppose we had a way of checking to see if the output of a netlist was 1; then I claim we would be done.

- Use “product construction” from Figure 6.

Unfortunately: nobody knows how to answer this question efficiently (for general designs)—“NP-complete” (Garey-Johnson 1979)

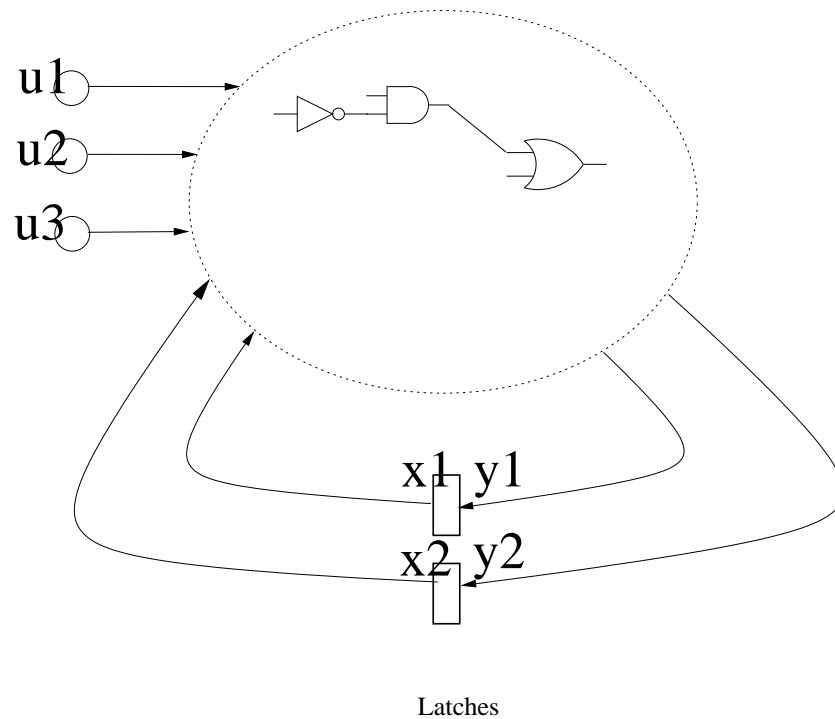


Figure 5: Alternate semantics for netlists.

4.1.1 Divide and Conquer

Natural CS'ish idea—split into (easier) subproblems.

Example 5 (Divide and Conquer) See Figure 7.

But how many subproblems does it introduce? Same as before, i.e., 2^n !!!

However:

1. may stop earlier
2. may have already done

This is one of the basic ideas behind BDD-based methods.

5 Cofactoring Boolean functions

Let $f : \{0, 1\}^n \rightarrow \{0, 1\}$ be a Boolean function.

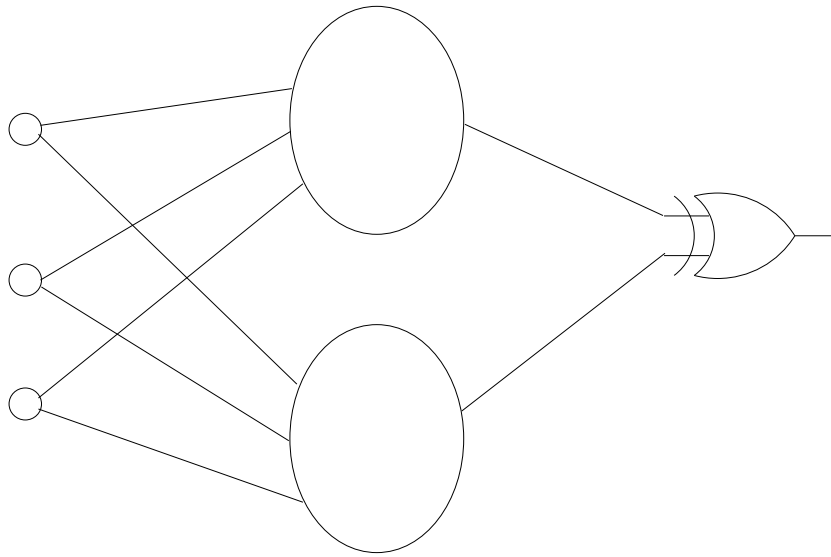


Figure 6: Product construction.

- Think of f as a netlist on inputs x_1, x_2, \dots, x_n

Definition 1 The function f cofactored wrt x_k is the function $f_{x_k}(x_1, \dots, x_{k-1}, x_{k+1}, \dots, x_n) : \{0, 1\}^{n-1} \mapsto \{0, 1\}$ where $f_{x_k}(x_1, \dots, x_{k-1}, x_{k+1}, \dots, x_n) = f(x_1, \dots, x_{k-1}, 1, x_{k+1}, \dots, x_n)$.

- note that f_{x_k} doesn't depend on x_k

Similarly, f cofactored wrt \bar{x}_k (denoted by $f_{\bar{x}_k}$ is the function $f(x_1, \dots, x_{k-1}, 0, x_{k+1}, \dots, x_n)$.

Recall $f_{x_1} = f(1, x_2, \dots, x_n)$

- Can define $f_{x_1 x_2 \bar{x}_3} = f(1, 1, 0, x_4, \dots, x_n)$

Properties of cofactoring:

1. $f_{x_i x_j} = f_{x_j x_i}$ “commutes”
2. $(f + g)_{x_i} = (f_{x_i} + g_{x_i})$ “distributes over disjunction”
3. $(f \cdot g)_{x_i} = (f_{x_i} \cdot g_{x_i})$ “distributes over conjunction”

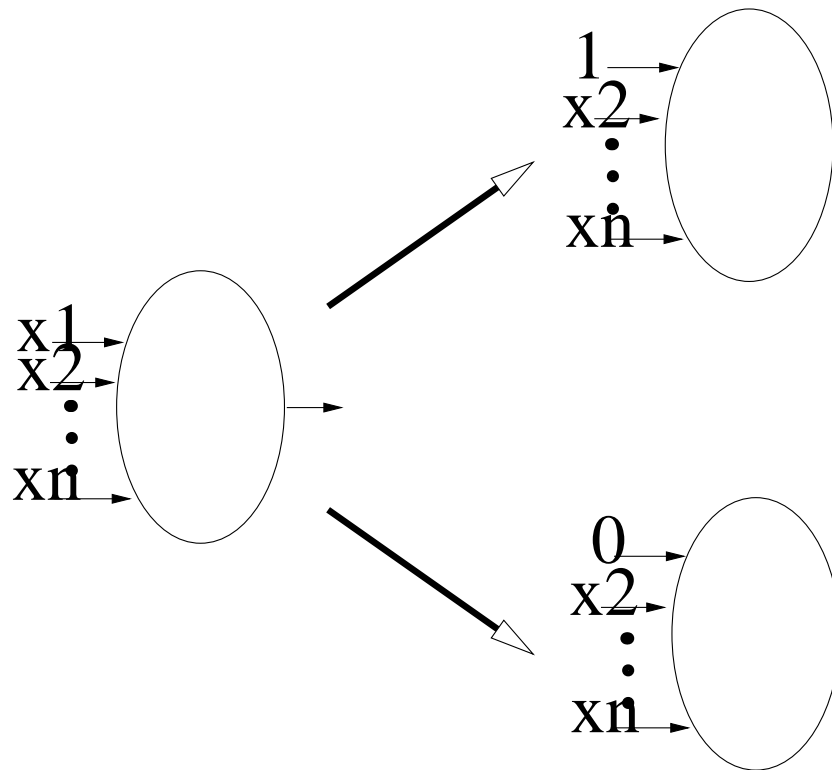


Figure 7: Divide and Conquer—observe one less variable.

5.1 Quantification

Given a function $f(x_1, \dots, x_n)$; suppose we want a function $g(x_2, \dots, x_n)$ such that $g(\alpha_2, \dots, \alpha_n) = 1$ exactly when there is a value for x_1 (call it c) so that $f(c, \alpha_2, \dots, \alpha_n) = 1$.

How can we obtain g ? Get a good idea from netlists—see Figure 9.

Observe: $g = f_{x_1} + f_{\bar{x}_1}$; we'll write $g = (\exists x_1)f$; often referred to as the *existential quantification* of f by x_1 .

- dual: $(\forall x_1)f$ —no matter what value (call it c) we give to x_1 , $f(c, x_2, \dots, x_n) = 1$. This is referred to as the *universal* quantification of f by x_1 .

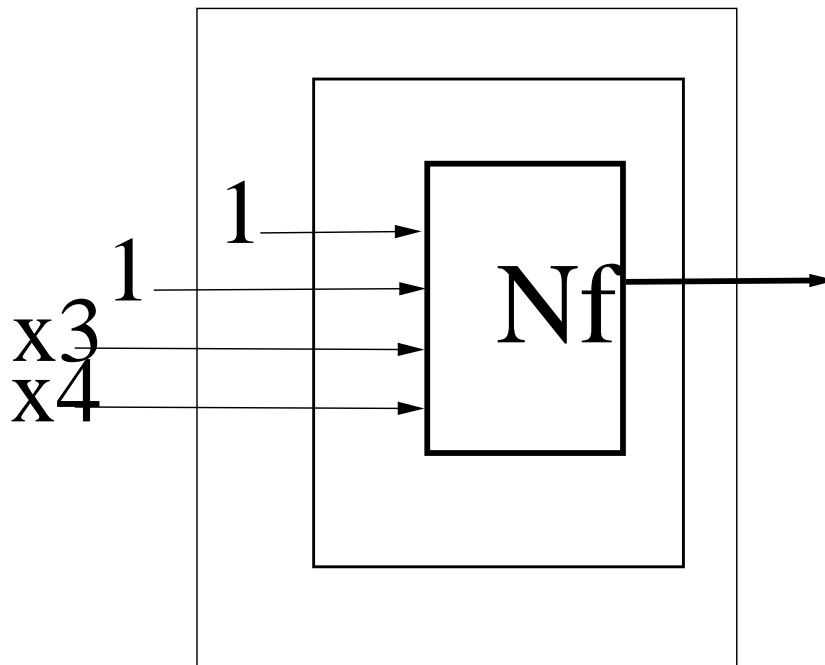


Figure 8: Cofactoring Boolean functions.

6 Decomposing Boolean functions

Theorem 1 (Shannon Expansion) *Let f be a Boolean function defined over the variables x_1, x_2, \dots, x_n .*

Then f can be expressed as:

$$f(x_1, \dots, x_n) = x_k \cdot f_{x_k}(x_1, \dots, x_n) + \bar{x}_k \cdot f_{\bar{x}_k}(x_1, \dots, x_n)$$

Proof: Write down the netlist for the expression on the right. See Figure 10.

■

Upshot—given $f_x, f_{\bar{x}}$ we can reconstruct f

Example 6 ($f = abc + a'b + a'b'c$) *See Figure 11.*

Observe—this representation is quite wasteful. Can perform some simplifications:

1. children same \Rightarrow collapse

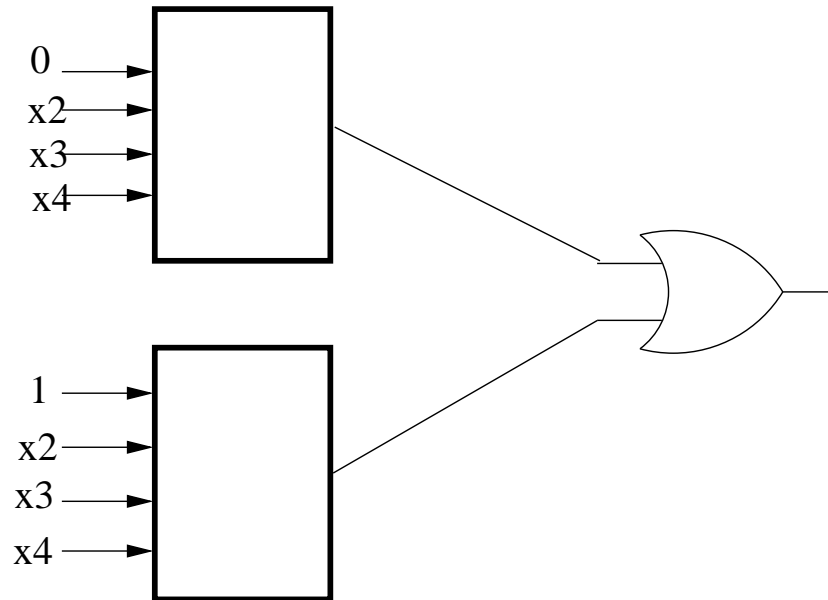


Figure 9: Quantification using netlists.

2. subgraphs isomorphic \Rightarrow merge

Ordered BDD: order in which variables are expanded are fixed

Combine reduced and ordered \Rightarrow reduced-ordered BDD (also called ROBDD or just BDD)

Benefits of ROBDDs:

1. compact representation
2. efficient operation

Observe each node is a distinct logic function (indeed a cofactor of f)

Theorem 2 (Bryant 1986) *Boolean functions f_1 and f_2 , are equal iff their ROBDDs are isomorphic (under same ordering)*

Algorithm for isomorphism?

- Both functions are 0, or 1
- labels agree and $(f_1)_v \approx (f_2)_v$ and $(f_1)_{\bar{v}} \approx (f_2)_{\bar{v}}$

Need to cache intermediate results.

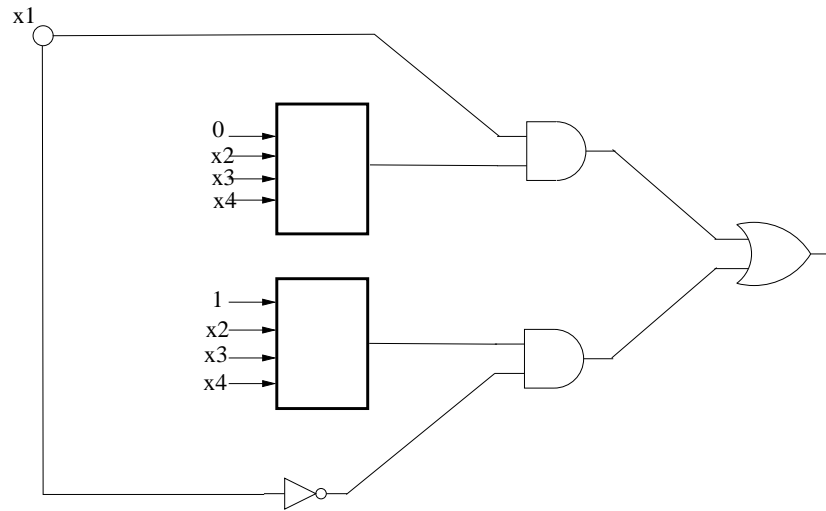


Figure 10: Shannon expansion.

7 Operations on BDDs

What are the basic operations we'd like to perform on BDDs?

1. Tautology
2. Equivalence
3. Cofactoring
4. Boolean— \wedge , \neg , \vee
5. Composition $F(x_1, \dots, x_n); x_1 = G(x_2, \dots, x_n)$
6. Quantification \exists, \forall

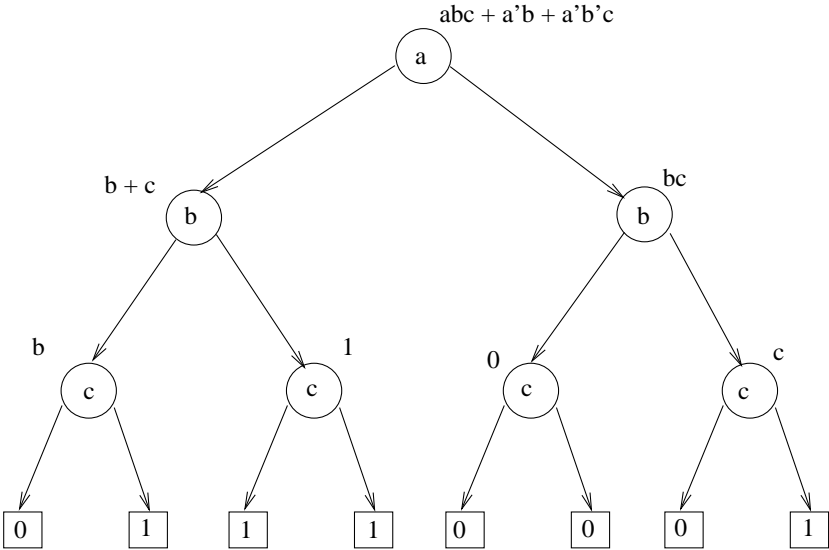


Figure 11: Recursive decomposition using the Shannon expansion.