We already learned how logic simulators work and about the complexity of Dynamic Verification Environment.

Today’s lesson:
- Why and how we may reduce our simulation time and efficiently ensure correctness of complex circuits
- Logic equivalence checking

```verilog
input in;
output out;
output [7:0] counter;
reg [7:0] counter;
wire counter_overflow = (counter == 8'hff);
wire counter_underflow = (counter == 8'h00);
always @(posedge clk or negedge xreset)
begin
  if (xreset == 0)
    counter <= 0;
  else if (en && !counter_overflow)
    counter <= counter + 1;
  else if (en && !counter_underflow)
    counter <= counter - 1;
  else
    counter <= counter;
end
```
Why we want to avoid simulation?

Problems with simulation

- Incomplete: it can help verify that some properties are obeyed for some inputs (not all properties for all possible inputs)
- Cannot prove correctness: Exhaustive simulation is impractical
- May help find some bugs, but no guarantee to find all of them
- Data dependent: it is only as good as the set of input vectors
- Inefficient: Requires time-consuming analysis to debug
- Inaccurate: “bugs” may be caused by modeling idiosyncrasies
- Takes a lot of resources (#1 CPU consumer in CAD)
- Re-simulation at each representation level would be prohibitive!
What can be done to reduce simulation?

- **Static checks:**
  - Inspect the design without “exercising” it over time
  - Data-independent
  - Direct identification of design problems

- **Invest in avoiding bugs instead of detecting bugs**
  - Design methodology can help in this
  - Automatic code generation, minimize manual design (which is error-prone)
Comparison with software design

- The ‘design & simulate’ cycle is similar to programming and run-time debugging.
- Helpful approaches to minimize debugging in software:
  - Structured programming
  - Strongly-typed programming languages
  - Coding style conventions
  - Object-oriented programming
  - Smart compilers and CASE tools doing syntax and semantic checking
- These are all *state* and *methodology-based* approaches
  - Are there analogous approaches to hardware design?
Validate one abstraction level, Check equivalence with lower levels

- Simulate at a high abstraction level only, don’t repeat at lower levels:
  - As the design gets refined through successive representations, keep checking that it is still equivalent to the previously-validated model at the higher abstraction level.

- Of course, the equivalence checking should be static
  - Not by simulating both models and comparing results….
A Verification Plan for RTL-based flow

- System Spec.
  - RTL Simulation
  - Static RTL checking:
    - properties
    - design rules

- RTL
  - RTL Validation
    - Combinational logic equivalence
    - State-machine equivalence
  - Some circuit simulations
    - Static ERC
    - Static timing

- Schematics
  - Verify Schematics
    - Check connectivity equivalence (LVS)

- Layout
  - Verify Layout
    - No simulation!
    - Static DRC

Avoid simulation as much as possible. Rely on static checks and equivalence verification.
Static Verification in CAD

- **DRC: Layout Design Rule Checking**
  - Min. sizes, min. spacing, area, overlap rules, etc.

- **ERC: Electrical rule checking**
  - Circuit configurations, composition rules, reliability rules

- **Other kinds of methodology rule checking**
  - E.g. RTL coding style (lint), RTL dead code, RTL power...

- **Static delay analysis**
  - Also: noise analysis, power analysis, reliability analysis, etc.

- **LVS: Layout Versus Schematics**
  - Extract layout netlist and compare to schematics

- **Logic functional equivalence checking**
  - RTL Vs. gates, gates Vs. gates, state-machine equivalence
What we said so far?

- Simulation limitations
  - Can’t claim for correctness
  - Not complete

- How can we avoid simulating every abstraction level?
  - Statically (not simulation) prove lower abstraction equals higher

- Different types of equivalences can be proven statically

- We start by comparing Logic of Gate Level vs. the RTL
  - We skip details of how RTL is elaborated to gates as a first step
  - So we compare two gate level models
RTL to Schematics Equivalence

- Why do we need to do this?
  - If schematics are formally equivalent to RTL no need to simulate

- Grand plan:
  - Assume we have matching Flip-Flops in both models
  - Break-down both models into registers and Combinational Blocks
  - Verify the external connections of each CB
  - Verify each CB for logic function equivalence
Gates-to-Gates Equivalence

- Each RTL CB logic functions are given as a (levelized) logic network
- Schematic CBs are either:
  1) modeled as gate-level logic networks, or
  2) modeled as transistor-level networks
    - In 2), use a switch-level analyzer to extract logic equations for the CB
- Now: we have pairs of CBs which are structurally different, and we need to verify that their logic functions are the same
  - Let’s look at a single output of each CB, to simplify explanations
Gate-to-gates Equivalence as a Tautology decision problem

- The basic problem: given a boolean function $F$, decide whether it is a tautology ($==$1)
- Tautology decision is an intractable problem
  - Worst case is exponential with the number of input variables
- How to approach it?

$F_{RTL} = F_{SCH}$

$F == 1$ for all inputs iff $F_{RTL} = F_{SCH}$
A divide-and-conquer strategy for tautology decision

- A recursive approach to prove tautology
  - Break down the function F into 2 smaller ones
  - …. And if the answer is still not obvious, recursively break down again and again….. Until the functions are clearly ==1 or not
  - If we are unlucky, the recursion tree will be exponential...
  - If we are lucky, we can terminate recursion fairly early!

- How to split function F into 2 smaller ones?
  - Select one of the input variables Xi (splitting variable)
  - Express F as a Shannon Expansion with regard to Xi.
  - What is Shannon expansion?
Shannon Expansion

- Given a boolean function $F(x_1, x_2, \ldots x_i, \ldots x_n)$
- Take variable $x_i$
- The cofactor of $F$ for $x_i$ is: $F_{x_i} = F(x_1, x_2, \ldots x_i = 1, \ldots x_n)$
- The cofactor of $F$ for $x_i'$ is: $F_{x_i'} = F(x_1, x_2, \ldots x_i = 0, \ldots x_n)$
  - These cofactors are new functions (of all the other variables)

- Shannon’s expansion is: $F = x_i F_{x_i} + x_i' F_{x_i'}$
  - This is a ‘divide and conquer’ approach to $F$
  - Handy for logic design when $x_i$ is a late-arriving signal. Why?
  - Easy to prove since $x_i$ can be either 0 or 1

- To check if $F$ is a tautology, check the cofactors!
Review of Boolean algebra

- \((0,1,+,\times)\)  
  \(+ = \text{OR} \quad \times = \text{AND}\)

- Commutative, distributive in both operations

- Identity elements: 0 for +, 1 for *

- Complement: \(a+a' = 1, a*a' = 0\)

- Associativity: \(a+(b+c) = (a+b)+c \quad a(bc) = (ab)c\)

- Idempotence: \(a+a = a \quad a*a = a\)

- Absorption: \(a+ab = a \quad a*(a+b) = a\)

- De Morgan: \((a+b)' = a'b \quad (ab)' = a'+b'\)

- Involution: \((a')' = a\)
A **literal** is a boolean variable or its complement
- examples: a, a’, b’, x, …

A **cube** (also called **implicant**) is a product of literals
- Actually it is a misleading name…. Because: it may be a point or a plane
- n boolean variables create an n-dimensional space
  - Each variable can be 0 or 1, so the only points in this space are the corners of an n-dimensional geometrical cube
  - A product of n literals is 1 corner (also called **minterm**)
  - A product of 0 literals is the whole space
  - A product of k literals is $2^{(n-k)}$ corners
Literals and Cubes/Implicants

- Representing a boolean function:
  - Mark the corners where $F=1$
  - e.g. $F=ab+bc+ac$
Off/ON/Don’t Care sets

- ON set = set of points (minterms) where $F=1$
- OFF set = set of points where $F=0$
- If the function is incompletely specified, there is a Don’t Care set of points ($F$ can be either 0 or 1)
- Operation on a function is equivalent to operations on minterm sets:
  - Union: $f_{on} \cup g_{on} = f \ OR \ g$
  - Intersection: $f_{on} \cap g_{on} = f \ AND \ g$
  - Containment checking: $f_{on} \supseteq g_{on} = f \geq g \quad g \Rightarrow f$
ON Covers, OFF Covers

- A cover is a list of cubes (implicants) which cover all the minterms in the ON set (or OFF set) that may include some of the DC set
  - There are many possible covers for boolean functions…. That’s the root of our logic equivalence verification problem!!
  - It is also the root of the logic minimization problem (synthesis - we’ll get to it within some weeks)
Representing Boolean functions

- Cube plots

- Truth tables: $2^n$ rows; impractical for $n \approx 16$

- Implicant tables: smaller than truth tables
  - Select implicants from the cube
  - How to obtain smallest table?
  - $0/1/-$ marks variable as pos/neg/d.c.
  - Note this is not a unique representation!
  - Can share with other functions
Representing Boolean functions

- **Expressions:**
  - 2-level expressions: sum of products, product of sums
    - Can be converted directly to/from implicant tables
  - Special cases:
    - Sum of minterms, product of maxterms: canonical (but impractical)
  - Multi-level expressions: arbitrarily parenthesized

- **Binary Decision Diagrams (BDDs):**
  - Reduced Ordered BDDs are unique (canonical)
  - The most useful data structure

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
a & b & c & f_1 & f_2 \\
\hline
1 & - & 1 & 1 & 1 \\
1 & 1 & - & 1 & 1 \\
- & 1 & 1 & 1 & 0 \\
\hline
\end{array}
\]

\[f_1 = ab + ac + bc\]

\[f_2 = a \ ( \ ...(...) \ )\]
Back to tautology checking…
(Start working with implicant tables)

- Let’s assume we already have a tool to convert any combinational logic network into an implicant table
  - Can be done by symbolic substitutions and ‘removal of parentheses’ - until we get a sum of products (2-level expression)
  - We’ll learn later about minimization of such 2-level expressions
  - So we have F represented by an implicant table

- We want to:
  - Select an input variable to “split the function” with
  - Apply Shannon to F (=generate the cofactors as implicant tables)
  - Look at each cofactor and have rules to decide whether it is a tautology, or not, or whether we can’t tell and need to “split again”

- These operations are easy to do if we store implicant tables using Positional Cube Notation …
Positional Cube Notation (PCN)

In implicant tables, use 2 bits to encode each input variable $X_i$, as follows:

- $00 =$ Void. This implicant should be deleted (whole row)
- $01 =$ 1 $X_i$ appears in the product term
- $10 =$ 0 $X'_i$ appears in the product term
- $11 =$ Don’t Care The term has no $X_i$ or $X'_i$

Example 1: $f = ab + bc + ac$

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>01</td>
</tr>
</tbody>
</table>

Example 2: $g = a'b' + ac$

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>01</td>
</tr>
</tbody>
</table>

More in tutorial
Positional Cube Notation Usefulness

- Intersection of implicants (cubes) is their **bitwise product**
- \( S_L = \) “Size” of literal \( S_L = \) number of 1’s in its field (1 or 2)
- \( I = \) Size of implicant = product of sizes of literals

Example 1: \( f = ab + bc + ac \)
\[
\begin{array}{ccccccc}
a & b & c & S_a & S_b & S_c & I \\
01 & 01 & 11 & 1 & 1 & 2 & 2 \\
11 & 01 & 01 & 2 & 1 & 1 & 2 \\
01 & 11 & 01 & 1 & 2 & 1 & 2 \\
\end{array}
\]

Example 2: \( g = a'c' + ac \)
\[
\begin{array}{ccccccc}
a & b & c & S_a & S_b & S_c & I \\
10 & 11 & 10 & 1 & 2 & 1 & 2 \\
01 & 11 & 01 & 1 & 2 & 1 & 2 \\
\end{array}
\]

- How does tautology look like?
  - But is it a unique representation?
So how to do the Shannon expansion?  
Cofactoring is easy with PCN!

- Look at each cube, in the Xi column
  - To take the cofactor for Xi:
    11 means this term does not depend on Xi  --> keep this term
    10 means Xi’ is in the term  --> delete this term
    01 means Xi  is in the term  --> write 11 (to remove Xi from cofactor)
  - To take the cofactor for Xi’:
    11  -->
    10  -->
    01  -->

- Example: \( f = ab + ac' + abc \)
  write down the PCN for the following

\[
\begin{array}{ccc}
| f | & | f_a | & | f_{b'} | \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a &amp; b &amp; c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 &amp; 01 &amp; 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 &amp; 11 &amp; 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 &amp; 01 &amp; 01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &amp; b &amp; c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 &amp; 01 &amp; 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 &amp; 11 &amp; 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 &amp; 01 &amp; 01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &amp; b &amp; c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 &amp; 01 &amp; 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 &amp; 11 &amp; 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 &amp; 01 &amp; 01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{array}
\]

Can we tell whether  \( f \) or its cofactors are tautologies?
Rules to identify a tautology
(and the usefulness of unate functions)

- If there is a cube of all 1’s (‘universal cube’) → Yes, it’s a Tautology!
- If there is a column of all 10 or 01 → No (variable never takes a value)
- If all columns except one are 11
  and that column has both 01 and 10 → Yes (….+ x+x’ + ….)
- More rules can be derived if the function is monotonic = unate functions
  - Why? Because if we know that f(0,0,0,…0)=1 and the function is monotonic increasing in all variables, we can be sure that f=1 for all other inputs
  - OK, so how do we know if f is monotonic in a variable?
    - In the expression of f, the variable appears only in one polarity
    - In PCN, the column does not contain both 01 and 10
  - If the function is unate in all variables,
    and there is no cube of all 1’s → No, not a tautology
    (unate is a tautology only if it contains the universal cube 11 11 11... )
Proof of the Unate Tautology Condition

- **Theorem:**
  - Let $A$ be a specific cover PCN of a unate function $F$
  - Then $F == 1$ iff $A$ has a row of all “11”s

- **Proof:**
  - $\leq$
    - A row of all “11”s is the tautology cube
  - $\Rightarrow$
    - Assume no row of all “11”s. Without loss of generality, suppose the function is positive unate. Then each row has at least one “01” in it. Consider the minterm(0,0,…,0). This is not contained in any row of $A$. Hence $A != 1$
    - So some row must be all “11”s
How to select a variable to “split” with?

- Heuristic: try to get rid of the variable which is most “binate” (that’s the opposite of unate).

- Example:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>01</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>11</td>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>

- a is binate (2 true a, 2 complement a’)
- b is unate
- c is unate
- d is binate (3 true d, 1 complement d’)

- We choose variable a to do the next Shannon expansion with!
The Tautology Algorithm

Tautology ( f - in PCN representation) {
    if (f is unate) {
        return (f meet the unate tautology termination rules)
    } else if (test for any other termination rules = yes or no)
        return (1 or 0 accordingly)
    } else {
        /* cannot decide, split the function to cofactors */
        x= “most binate” variable in f
        return (Tautology( f_{x} ) && Tautology(f_{x}')) /*recursion*/
    }
}
Generalization:
The Unate Recursive Paradigm (URP)

- This scheme of computation is a general paradigm, proposed by Brayton, who suggested to look for unate functions within a recursion:
  - If you have a large problem, divide it up to two sub-problems
  - Use Shannon expansion, express the problem by cofactors
  - Every operation between functions can be applied to the cofactors, so the Shannon expansion is very useful:
    \[ f \boxplus g = X_i(f_{x_i} \boxdot g_{x_i}) + X_i'(f_{i'} \boxdot g_{i'})| \boxdot \in \{+,* ,\oplus\} \]
  - Try to make cofactors unate, because unate functions have nice properties and this may help solve the sub-problems more easily
  - If no solution yet, recursively apply this procedure - until the sub-problems are easy enough to terminate

- This computational strategy is also the basis of Synthesis of 2-level logic minimization (Espresso) – we will learn it – stay tuned
Another trick: Partitioning

Instead of using Shannon to split the function, it may be possible to express it as a sum of 2 other functions which depend on different variables:

\[ F = G + H \]

(where G and H have no variables in common)

If we succeed, then F is tautologous if either G or H are

- Significant reduction in worst-case problem size

How to do the partitioning?

- Pick a cube, take the columns where the cube has 01 or 10
- Add the cubes which are affected by any of these columns
- Repeat these steps on the added cubes, until no more columns can be added
- If there are any columns left over - they are the second partition!

Example:

Start with first cube, take col 1 and 3

Add cubes in row 3,4

Take col 4, no more columns can be added

H=columns 1,3,4   G=columns 2,5
Effect of Don’t Cares on equivalence checking

- What do we do to verify the implementation of a function which is incompletely specified?
  - Remember: incompletely specified functions have don’t cares

- We actually want to ensure that our function does not intersect with the OFF SET of the specified function
  - It can be converted to a tautology decision too:
    - Use 2 auxiliary, complete functions, to represent the specification:
      - \( f = \) ON set of the specification
      - \( d = \) DON’T CARE set of the specification
    - \( g = \) the implementation (no don’t cares here)
    - check tautology:
      - \( d + f \cdot g + f' \cdot g' = 1 \)
Summary

Whenever applicable, static checking has inherent advantages over dynamic checking (=simulation)

Combinational logic equivalence verification (=tautology decision)

Shannon expansion and cofactors as a way to “split” functions into smaller functions

Positional Cube Notation as a useful data structure

Looking for unate (monotonic) functions

Unate Recursive Paradigm to solve complex computations:
  - Choose a splitting variable
  - Do a Shannon expansion and proceed with each cofactor separately
  - Using rules to decide about unateness (termination rules)
  - If can’t decide - keep splitting the problem recursively