Outline

- Introduction
- Two-level Logic Synthesis
- Multi-level Logic Synthesis
Introduction

- Cho Moon
  - PhD from UC Berkeley 92
  - Lattice Semiconductor (synthesis) 92 - 96
  - Cadence Design Systems (synthesis, verification and timing analysis) 96 – present

- Why logic synthesis?
  - Ubiquitous – used almost everywhere VLSI is done
  - Body of useful and general techniques – same solutions can be used for different problems
  - Foundation for many applications such as
    - Formal verification
    - ATPG
    - Timing analysis
    - Sequential optimization
RTL Design Flow

- HDL
  - RTL Synthesis
    - Manual Design
    - Module Generators
  - netlist
    - Logic Synthesis
      - netlist
      - Physical Synthesis
        - layout

Library

Module Generators

Slide courtesy of Devadas, et. al
Logic Synthesis Problem

- **Given**
  - Initial gate-level netlist
  - Design constraints
    - Input arrival times, output required times, power consumption, noise immunity, etc...
  - Target technology libraries

- **Produce**
  - Smaller, faster or cooler gate-level netlist that meets constraints

Very hard optimization problem!
Combinational Logic Synthesis

Slide courtesy of Devadas, et. al
Outline

- Introduction
- Two-level Logic Synthesis
- Multi-level Logic Synthesis
- Sequential Logic Synthesis
Two-level Logic Synthesis Problem

- Given an arbitrary logic function in two-level form, produce a smaller representation.
- For sum-of-products (SOP) implementation on PLAs, fewer product terms and fewer inputs to each product term mean smaller area.

F = A B + A B C

F = A B
Boolean Functions

\[ f(x) : B^n \rightarrow B \]
\[ B = \{0, 1\}, \ x = (x_1, x_2, \ldots, x_n) \]
- \( x_1, x_2, \ldots \) are variables
- \( x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots \) are literals
- each vertex of \( B^n \) is mapped to 0 or 1
- the onset of \( f \) is a set of input values for which \( f(x) = 1 \)
- the offset of \( f \) is a set of input values for which \( f(x) = 0 \)
A literal is a variable or its negation $y, \overline{y}$

It represents a **logic function**

Literal $x_1$ represents the logic function $f$, where $f^1 = \{x \mid x_1 = 1\}$

Literal $\overline{x_1}$ represents logic function $g$ where $g^1 = \{x \mid x_1 = 0\}$
Boolean Formulas

Boolean functions can be represented by formulas defined as catenations of

- parentheses - (, )
- literals - $x, y, z, \overline{x}, \overline{y}, \overline{z}$
- Boolean operators - $\lor$ (OR), $\land$ (AND)
- complementation - e.g. $\overline{x + y}$

Examples:

\[
\begin{align*}
    f &= x_1 \times \overline{x_2} + \overline{x_1} \times x_2 \\
    &= (x_1 + x_2) \times (\overline{x_1} + \overline{x_2}) \\
    h &= a + b \times c \\
    &= \overline{a} \times (\overline{b} + \overline{c})
\end{align*}
\]

We will usually replace $\times$ by catenation, e.g. $a \times b \rightarrow ab$. 

Slide courtesy of Devadas, et. al
Logic Functions: \[ f(x) : B^n \rightarrow B \]

There are \(2^n\) vertices in input space \(B^n\).

There are \(2^{2^n}\) distinct logic functions. Each subset of vertices is a distinct logic function: \(f^1 \subseteq B^n\).

There are \(\infty\) number of logic formulas

\[
\begin{align*}
f & = x + y \\
& = x\overline{y} + xy + \overline{x}y \\
& = x\overline{x} + x\overline{y} + y \\
& = (x + y)(x + \overline{y}) + \overline{y}
\end{align*}
\]

SYNTHESIS = Find the "best" formula (or "representation")
Cube Representation

The AND of a set of literal functions ("conjunction" of literals) is a cube

\[ C' = x\overline{y} \]

is a cube

\[ C' = (x = 1)(y = 0) \]

if \( C^1 \subseteq B^n \) and \( C' \) has \( k \) literals, then \( |C^1| \) has \( 2^{n-k} \) vertices

**Example 1**  \( C' = x\overline{y} \subseteq B^3 \)

\[ k = 2 \]

\[ n = 3 \]

\[ |C^1| = 2 = 2^{3-2} \]

if \( k = n \), the cube is a min-term

Slide courtesy of Devadas, et. al
Operations on Logic Functions

- (1) Complement: \( f \longrightarrow \overline{f} \)
interchange ON and OFF-SETS

- (2) Product (or intersection or logical AND)
  \( h = f \cdot g \) or \( h = f \cap g \)

- (3) Sum (or union or logical OR):
  \( h = f + g \) or \( h = f \cup g \)
Sum-of-products (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” representation.

- A SOP can be thought of as a set of cubes \( F \)

  \[ F = \{ab, ac, bc\} = C \]

- A set of cubes that represents \( f \) is called a cover of \( f \).
  \( F=\{ab, ac, bc\} \) is a cover of \( f = ab + ac + bc \).
Prime Cover

- A cube is prime if there is no other cube that contains it
  (for example, b c is not a prime but b is)

- A cover is prime iff all of its cubes are prime
Irredundant Cube

- A cube of a cover $C$ is irredundant if $C$ fails to be a cover if $c$ is dropped from $C$.

- A cover is irredundant iff all its cubes are irredundant (for example, $F = a \cdot b + a \cdot c + b \cdot c$).
Quine-McCluskey Method

- We want to find a minimum prime and irredundant cover for a given function.
  - Prime cover leads to min number of inputs to each product term.
  - Min irredundant cover leads to min number of product terms.

- Quine-McCluskey (QM) method (1960’s) finds a minimum prime and irredundant cover.
  - Step 1: List all minterms of on-set: $O(2^n)$ $n = \#\text{inputs}$
  - Step 2: Find all primes: $O(3^n)$ $n = \#\text{inputs}$
  - Step 3: Construct minterms vs primes table
  - Step 4: Find a min set of primes that covers all the minterms: $O(2^m)$ $m = \#\text{primes}$
QM Example (Step 1)

- $F = a' b' c' + a b' c' + a b' c + a b c + a' b c$
- List all on-set minterms

<table>
<thead>
<tr>
<th>Minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a' b' c'$</td>
</tr>
<tr>
<td>$a b' c'$</td>
</tr>
<tr>
<td>$a b' c$</td>
</tr>
<tr>
<td>$a b c$</td>
</tr>
<tr>
<td>$a' b c$</td>
</tr>
</tbody>
</table>
QM Example (Step 2)

- \( F = a' b' c' + a b' c' + a b' c + a b c + a' b c \)

- Find all primes.

<table>
<thead>
<tr>
<th>primes</th>
<th>b' c'</th>
<th>a b'</th>
<th>a c</th>
<th>b c</th>
</tr>
</thead>
</table>
QM Example (Step 3)

- $F = a' \ b' \ c' + a \ b' \ c' + a \ b' \ c + a \ b \ c + a' \ b \ c$

- Construct minterms vs primes table (prime implicant table) by determining which cube is contained in which prime. $X$ at row $i$, column $j$ means that cube in row $i$ is contained by prime in column $j$.

<table>
<thead>
<tr>
<th></th>
<th>$b' \ c'$</th>
<th>$a \ b'$</th>
<th>$a \ c$</th>
<th>$b \ c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a' \ b' \ c'$</td>
<td>$X$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a \ b' \ c'$</td>
<td>$X$</td>
<td>$X$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a \ b' \ c$</td>
<td></td>
<td>$X$</td>
<td>$X$</td>
<td></td>
</tr>
<tr>
<td>$a \ b \ c$</td>
<td></td>
<td>$X$</td>
<td>$X$</td>
<td></td>
</tr>
<tr>
<td>$a' \ b \ c$</td>
<td></td>
<td></td>
<td></td>
<td>$X$</td>
</tr>
</tbody>
</table>
QM Example (Step 4)

- \( F = a' b' c' + a b' c' + a b' c + a b c + a' b c \)
- Find a minimum set of primes that covers all the minterms
  
  “Minimum column covering problem”

<table>
<thead>
<tr>
<th></th>
<th>b' c'</th>
<th>a b'</th>
<th>a c</th>
<th>b c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a' b' c'</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b' c'</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b' c</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>a b c</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>a' b c</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Essential primes
ESPRESSO – Heuristic Minimizer

- Quine-McCluskey gives a minimum solution but is only good for functions with small number of inputs (< 10)

- ESPRESSO is a heuristic two-level minimizer that finds a “minimal” solution

  ESPRESSO(F) {
    do {
      reduce(F);
      expand(F);
      irredundant(F);
    } while (fewer terms in F);
    verfiy(F);
  }

  
ESPRESSO ILLUSTRATED

Reduce

Irredundant

Expand
Outline

- Introduction
- Two-level Logic Synthesis
- Multi-level Logic Synthesis
Multi-level Logic Synthesis

- Two-level logic synthesis is effective and mature
- Two-level logic synthesis is directly applicable to PLAs and PLDs

But...

- There are many functions that are too expensive to implement in two-level forms (too many product terms!)
- Two-level implementation constrains layout (AND-plane, OR-plane)

Rule of thumb:
- Two-level logic is good for control logic
- Multi-level logic is good for datapath or random logic
Boolean network:

- directed acyclic graph (DAG)
- node logic function representation \( f_j(x, y) \)
- node variable \( y_j: y_j = f_j(x, y) \)
- edge \((i, j)\) if \( f_j \) depends explicitly on \( y_i \)

Inputs \( x = (x_1, x_2, \ldots, x_n) \)

Outputs \( z = (z_1, z_2, \ldots, z_p) \)
Multi-level Logic Synthesis Problem

- **Given**
  - Initial Boolean network
  - Design constraints
    - Arrival times, required times, power consumption, noise immunity, etc…
  - Target technology libraries

- **Produce**
  - a minimum area netlist consisting of the gates from the target libraries such that design constraints are satisfied
Modern Approach to Logic Optimization

- Divide logic optimization into two subproblems:
  - **Technology-independent optimization**
    - determine overall logic structure
    - estimate costs (mostly) independent of technology
    - simplified cost modeling
  - **Technology-dependent optimization (technology mapping)**
    - binding onto the gates in the library
    - detailed technology-specific cost model

- Orchestration of various optimization/ transformation techniques for each subproblem

Slide courtesy of Keutzer
Technology-Independent Optimization

Simplified cost models

- **Area** = sum of factored form literals in all nodes
  - Number of product terms is not a good measure of area in multi-level implementation
  
  \[ f = ad + ae + bd + be + cd + ce \]  
  \[ f' = a'b'c' + d'e' \]  

  *The only difference between \( f \) and \( f' \) is inversion*

  \[ f = (a+b+c)(d+e) \]  
  \[ f' = a'b'c' + d'e' \]

- **Delay** = levels of logic on critical paths
Technology-Independent Optimization

Technology-independent optimization is a bag of tricks:

- Two-level minimization (also called simplify)
- Constant propagation (also called sweep)
  \[ f = a \ b + c; \ b = 1 \Rightarrow f = a + c \]
- Decomposition (single function)
  \[ f = abc+abd+a'c'd'+b'c'd' \Rightarrow f = xy + x'y'; \ x = ab; \ y = c+d \]
- Extraction (multiple functions)
  \[ f = (az+bz')cd+e \quad g = (az+bz')e' \quad h = cde \]
  \[ \downarrow \]
  \[ f = xy+e \quad g = xe' \quad h = ye \quad x = az+bz' \quad y = cd \]
More Technology-Independent Optimization

- More technology-independent optimization tricks:
  - Substitution
    \[ g = a+b \quad f = a+bc \]
    \[ \downarrow \]
    \[ f = g(a+b) \]
  - Collapsing (also called elimination)
    \[ f = ga+g'b \quad g = c+d \]
    \[ \downarrow \]
    \[ f = ac+ad+bc’d’ \quad g = c+d \]
  - Factoring (series-parallel decomposition)
    \[ f = ac+ad+bc+bd+e \Rightarrow f = (a+b)(c+d)+e \]
Summary of Typical Recipe for TI Optimization

- Propagate constants
- Simplify: two-level minimization at Boolean network node
- Decomposition
- Local “Boolean” optimizations
  - Boolean techniques exploit Boolean identities (e.g., $a a' = 0$)
    Consider $f = a b' + a c' + b a' + b c' + c a' + c b'$
  - Algebraic factorization procedures
    $f = a (b' + c') + a' (b + c) + b c' + c b'$
  - Boolean factorization produces
    $f = (a + b + c) (a' + b' + c')$

Slide courtesy of Keutzer
Technology-Dependent Optimization

Technology-dependent optimization consists of

- Technology mapping: maps Boolean network to a set of gates from technology libraries

- Local transformations
  - Discrete resizing
  - Cloning
  - Fanout optimization (buffering)
  - Logic restructuring

Slide courtesy of Keutzer
Technology Mapping

Input
1. Technology independent, optimized logic network
2. Description of the gates in the library with their cost

Output
- Netlist of gates (from library) which minimizes total cost

General Approach
- Construct a subject DAG for the network
- Represent each gate in the target library by pattern DAG’s
- Find an optimal-cost covering of subject DAG using the collection of pattern DAG’s
- Canonical form: 2-input NAND gates and inverters
DAG Covering

- DAG covering is an NP-hard problem

- Solve the sub-problem optimally
  - Partition DAG into a forest of trees
  - Solve each tree optimally using tree covering
  - Stitch trees back together

Slide courtesy of Keutzer
Tree Covering Algorithm

- Transform netlist and libraries into canonical forms
  - 2-input NANDs and inverters

- Visit each node in BFS from inputs to outputs
  - Find all candidate matches at each node N
    - Match is found by comparing topology only (no need to compare functions)
  - Find the optimal match at N by computing the new cost
    - New cost = cost of match at node N + sum of costs for matches at children of N
  - Store the optimal match at node N with cost

- Optimal solution is guaranteed if cost is area

- Complexity = $O(n)$ where $n$ is the number of nodes in netlist
Tree Covering Example

Find an “optimal” (in area, delay, power) mapping of this circuit into the technology library (simple example below)

Slide courtesy of Keutzer
## Elements of a library - 1

<table>
<thead>
<tr>
<th>Element/Area Cost</th>
<th>Tree Representation (normal form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVERTER</td>
<td><img src="image" alt="Inverter Circuit" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image" alt="Inverted Circuit" /></td>
</tr>
<tr>
<td>NAND2</td>
<td><img src="image" alt="NAND2 Circuit" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="image" alt="NAND2 Circuit" /></td>
</tr>
<tr>
<td>NAND3</td>
<td><img src="image" alt="NAND3 Circuit" /></td>
</tr>
<tr>
<td>4</td>
<td><img src="image" alt="NAND3 Circuit" /></td>
</tr>
<tr>
<td>NAND4</td>
<td><img src="image" alt="NAND4 Circuit" /></td>
</tr>
<tr>
<td>5</td>
<td><img src="image" alt="NAND4 Circuit" /></td>
</tr>
</tbody>
</table>

*Slide courtesy of Keutzer*
Elements of a library - 2

<table>
<thead>
<tr>
<th>Element/Area Cost</th>
<th>Tree Representation (normal form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOI21 4</td>
<td>![AOI21 tree representation]</td>
</tr>
<tr>
<td>AOI22 5</td>
<td>![AOI22 tree representation]</td>
</tr>
</tbody>
</table>

Slide courtesy of Keutzer
Can we do better with tree covering?
Optimal tree covering - 1

``subject tree``

Slide courtesy of Keutzer
Optimal tree covering - 2

``subject tree``

Slide courtesy of Keutzer
Optimal tree covering - 3

Cover with ND2 or ND3?

1 NAND2 + subtree = 4

Area cost 8

Slide courtesy of Keutzer
Optimal tree covering – 3b

Label the root of the sub-tree with optimal match and cost

Slide courtesy of Keutzer
Cover with INV or AO21?

``subject tree``

1 AO21 4
+ subtree 1 3
+ subtree 2 2

Area cost 15

1 Inverter 2
+ subtree 13

Area cost 9

Slide courtesy of Keutzer
Label the root of the sub-tree with optimal match and cost

Slide courtesy of Keutzer
Optimal tree covering - 5

Cover with ND2 or ND3?

``subject tree``

NAND2
- subtree 1 9
- subtree 2 4
- 1 NAND2 3

Area cost 16

NAND3
- subtree 1 8
- subtree 2 2
- subtree 3 4
- 1 NAND3 4

Area cost 18

Slide courtesy of Keutzer
Optimal tree covering – 5b

Label the root of the sub-tree with optimal match and cost

``subject tree``

Slide courtesy of Keutzer
Optimal tree covering - 6

Cover with INV or AOI21?

``subject tree``

INV
subtree 1 16
1 INV 2
subtree 1 13
subtree 2 5
1 AOI21 4

Area cost 18
Area cost 22

Slide courtesy of Keutzer
Optimal tree covering – 6b

Label the root of the sub-tree with optimal match and cost

``subject tree``

Slide courtesy of Keutzer
Optimal tree covering - 7

Cover with ND2 or ND3 or ND4?

``subject tree``

Slide courtesy of Keutzer
Cover 1 - NAND2

``subject tree``

Cover with ND2?

Slide courtesy of Keutzer

Area cost 21
Subject tree

Cover with ND3?

``subject tree``

<table>
<thead>
<tr>
<th>Subtree</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtree 1</td>
<td>9</td>
</tr>
<tr>
<td>Subtree 2</td>
<td>4</td>
</tr>
<tr>
<td>Subtree 3</td>
<td>0</td>
</tr>
<tr>
<td>1 NAND3</td>
<td>4</td>
</tr>
</tbody>
</table>

Area cost 17

Slide courtesy of Keutzer
``subject tree``

Cover with ND4?

`````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````
Optimal Cover was Cover 2

``subject tree``

INV
ND2
AOI21
ND3
INV
ND2
2 ND3
AOI21
Area cost 17

Slide courtesy of Keutzer
Summary of Technology Mapping

- **DAG covering formulation**
  - Separated library issues from mapping algorithm (can’t do this with rule-based systems)

- **Tree covering approximation**
  - Very efficient (linear time)
  - Applicable to wide range of libraries (std cells, gate arrays) and technologies (FPGAs, CPLDs)

- **Weaknesses**
  - Problems with DAG patterns (Multiplexors, full adders, …)
  - Large input gates lead to a large number of patterns
Local Transformations

- **Given**
  - Technology-mapped netlist
  - Target technology libraries
  - Some violations (timing, noise immunity, power, etc…)

- **Produce**
  - New netlist that corrects the given violations without introducing new violations

- **Approach: another bag of tricks**
  - Discrete resizing
  - Cloning
  - Buffering
  - Logic restructuring
  - More…
Discrete Resizing

Note that some arrival and required times become invalid.
Cloning

Note that loads at a, b increase
Buffering

![Graph showing the buffering effect with load and delay parameters.]

- **Load** vs. **Delay (d)**
  - Graph indicates the delay (d) in response to load changes.
  - Three lines correspond to different buffering cases (A, B, C).

- **Nodes and Weights**
  - Diagram includes nodes a, b, d, e, f, g, h with associated weights.
  - Node labels and weights are consistent with the graph's context.
  - Graphs illustrating the buffering effect with load and delay parameters.
Logic Restructuring 1

- Nodes in critical section that fan out outside of critical section are duplicated

Slides courtesy of Keutzer
Logic Restructuring 2

- Place timing-critical nodes closer to output
  - Make them pass through fewer gates
  - After collapse, a divisor is selected such that substituting $k$ into $f$ places critical signal $c$ and $d$ closer to output

Collapse critical section

Re-extract factor $k$
Summary of Local Transformations

- Variety of methods for delay optimization
  - No single technique dominates

- The one with more tricks wins? No!

- Need a good framework for evaluating and processing different transforms
  - Accurate, fast timing engine with incremental analysis capability
    - don’t want to retime the whole design for each local transform
  - Simultaneous min and max delay analysis
    - How does fixing the setup violation affect the existing hold checks?