Introduction of Logic Synthesis and Optimization

Speaker: Prof. Zhufei Chu
E-Mail: chuzhufei@nbu.edu.cn
Homepage: https://zfchu.github.io/
EECS, Ningbo University
Outline

- Introduction
- Logic Representations
- Logic Optimization
- Technology Mapping
- Overview of Logic Synthesis Tools
- Demo
- Conclusions
Introduction
What is logic synthesis?

residue = 16’h0000;
if (high_bits == 2’h10)
    residue = state_table[index];
else state_table[index] = 16’h0000;

[Courtesy: https://vlsi-backend-adventure.com/logic_synthesis.html]
Digital Computing Systems

From circuits to systems
IoT (Internet of Things)

[Courtesy: J. Rabaey]

©Zhufei Chu
Design Objective

Mobile Computing
- Low power
- Portable

Aeronautics and Astronautics
- High reliability
- Add redundant backups

Data center\Cloud computing
- High performance
- Parallel、Multi-core

Performance

Power

Area

Accuracy

Approximation

©Zhufei Chu
Digital Integrated Circuit Design
Overall Process

- HDL spec.
- RTL synthesis
- logic netlist
- logic synthesis
- circuit netlist
- layout / mask
- physical design
- fab.
- chip

[Courtesy: R. Jiang]
Logic Synthesis Flow

Translation of High-Level Design into GTECH Netlist

- High-level & datapath optimization
- Multiplexer optimization & mapping
- Sequential mapping
- Technology-independent logic optimization

Technology mapping
- Timing analysis & optimization (buffering/gate sizing/pin sweeping)
- Register retiming
- Leakage, power, and area optimization

Post Optimization of Netlist

[Courtesy: Dr. Yu Huang, HiSilicon]
Critical Issues in Logic Synthesis Research

Methods of Boolean Function Representation
• Truth table
• SOP/POS/CNF
• BDD
• DAG
• …

Methods of Boolean Function Optimization
• General algebraic method
• Boolean algebraic method
• Exact synthesis method
• …

Methods of Boolean Function Mapping
• Cover-based method
• Tree-based method
• Oriented to FPGA/ASIC
• …

SAT Solver / BDD Reasoning Engine
Logic Synthesis Technique Map

Goals
- Power
- Performance
- Area
- Accuracy

Types
- Accurate
- Approximate

PPA
- Rewriting
- Divisor extraction
- Mapping/Unmapping
- Fast optimization using one cut per node
- Replacing nodes by constants
- Merging almost-equivalent nodes
- A more general simulation-guided synthesis

Methods
- Exact
- Heuristic

Object
- Sequential
- Combinational

Data Structure
- AIG: And-Inverter graphs
- BDD: Binary decision diagram
- CNF: Conjunctive normal form
- DNF: Disjunctive normal form
- TT: Truth table
- MIG: Majority-inverter graphs
- XMG: XOR-majority graph
- XAG: XOR-and graph

©Zhufei Chu
Logic Synthesis Tools

The development history of logic synthesis tools

70's
- Few logic synthesis algorithms and tools existed in the 70’s
- Link to place and route for automatic design
  - Innovative methods at IBM, Bell Labs, Berkeley, Stanford

Early 80’s
- First prototype synthesis tools in the early 80’s
  - YLE (Yorktown Logic Editor) [Brayton]
  - MIS (Multi-level Interactive System) [Berkeley]
  - Espresso [IBM+Berkeley]

Late 80’s
- First logic synthesis companies in the late 80’s
  - Synopsys and others
Logic Synthesis Tools

- Commercial logic synthesis tools
  - Vivado [Xilinx]
  - Fusion Compiler [Synopsys]
  - Genus [Cadence]

- Academic logic synthesis tools
  - SIS [Berkeley]
  - ABC [Berkeley]
  - Yosys [Clifford Wolf]
  - The EPFL Logic Synthesis Libraries [EPFL]
  - CIRKIT [EPFL] [Ningbo University]
  - ALSO [Ningbo University]
Logic Representations
Logic representation zoo

- $f = x_1 x_2 + x_2 x_3 + x_1 x_3$
  - Sum-of-Products, SOP
- $f = (x_1 + x_2)(x_2 + x_3)(x_1 + x_3)$
  - Disjunctive Normal Form, DNF
- $f = (1110 1000)_2 = 0xe8$ (Truth tables, TT)
- BDD
- AIG
- ...
Truth tables

Truth table representation is
- intractable for large $n$
- canonical

Example:

$$f = abc\bar{d} + ab\bar{c}d + \bar{a}bcd + \bar{a}bc\bar{d} + \bar{abc}d + \bar{abc}d + abd + abcd$$
Example:

\[ ab\bar{c} + \bar{a}bd + b\bar{d} + \bar{b}\bar{e}f \]

Advantages:
- Easy to manipulate and minimize
- Many algorithms available
- Two-level theory applies

Disadvantages:
- Not representative of logic complexity. For example:
  \[ f = ad + ae + bd + be + cd + ce \quad \bar{f} = \bar{a}\bar{b}\bar{c} + \bar{d}\bar{e} \]
- Not easy to estimate logic size and performance
- Difficult to estimate progress during logic manipulation
POS

- Product of Sum (POS) representation of Boolean function
- Describes solution using a set of constraints
  - very handy in many applications because new constraints can just be added to the list of existing constraints
  - very common in AI community

Example:

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

- SAT on CNF (POS) \iff Tautology on DNF (SOP)
Binary Decision Diagrams (BDD)

- General idea: Representation of a logic function as graph (DAG)
  - S. B. Akers proposed in 1978, R.E Bryant extended in 1986 (ROBDD)
  - Based on recursive Shannon expansion
    - one root node, two terminals 0, 1
    - each node, two children, and a variable

- Reduced and ordered (ROBDD)
  - Reduced:
    - any node with two identical children is removed
    - two nodes with isomorphic BDD’s are merged
  - Ordered:
    - Co-factoring variables (splitting variables) always follow the same order along all paths

\[
f = v f_v + \overline{v} f_{\overline{v}}
\]
Reduced ordered BDD

**BDD Reduction Rules -1**
- with both edges pointing to same node

\[ f = a g(b) + \bar{a} g(b) = g(b) \]

**BDD Reduction Rules -2**
- Merge duplicate nodes

\[ f_1 = \bar{a} g(b) + a h(c) = f_2 \]

\[ f = f_1 = f_2 \]
BDD Example

BDD Ordered

\[
f = ab + \overline{a}c + b\overline{c}d
\]
Logic network

- Logic Network is a generic representation of a logic circuit.
- Several common elements of logical networks:
  - Primary Input/Output, PI/PO
  - Node
  - Fanin/Fanout, FI/FO
  - Transitive fanin/fanout cone, TFI/TFO
Logic network

- \( f_4 = x_1 + x_2 \)
- \( f_5 = x_1(x_1 + x_2) + \overline{x_1}(x_1 + x_2) \)
- \( f_5 = x_1f_4 + \overline{x_1}f_4 \)
- \( f_6 = x_3(x_1f_4 + \overline{x_1}f_4) \)
DAGs

\[ f = a \oplus (bc + cd + bd) \]
Historical perspective

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1950-1970</td>
</tr>
<tr>
<td>50</td>
<td>1980</td>
</tr>
<tr>
<td>100</td>
<td>1990</td>
</tr>
<tr>
<td>100000</td>
<td>2000</td>
</tr>
</tbody>
</table>

- TT
- SOP
- BDD
- AIG
- CNF
- Espresso, MIS, SIS
- SIS, VIS, MVSIS
- ABC
LUT network

Logic network

LUT network
LUT network

\[ f = a \oplus (bc + cd + bd) \]
Logic Optimization
Two-level vs multi-level

\[ f = a\overline{b}c + ad + b\overline{c}d \]

深度 = 2

深度 = 4

多级

多级
Two-level logic synthesis

\[ f = a\bar{b}c + ad + b\bar{c}d \] logic expression based on SOP

- 3 product implicants, 8 literals
- \#implicants \rightarrow \text{number of rows in the PLA}
- \#literals \rightarrow \text{number of transistors}

- multiple AND gates in the first level
- a large OR gate in the second level
Quine-McCluskey Algorithm

- Minimization of a second-order logic formula
  - Reducing the SOP expression to its minimal literals
  - Willard Quine proposed it in 1952, Edward J. McCluskey expanded it in 1956.

- Base idea
  - Combining adjacent minterms and eliminating redundant factors, e.g.: \( xy + x\bar{y} = x \)
  - A computable version of the Karnaugh map that is easy to implement in programming
  - Simplification applicable to any complex logic function
Two-level logic synthesis

- Espresso

- Decrease

- Loop

- Expand

- Non-redundant

©Zhufei Chu
Multi-level logic synthesis

- Elimination
- Decomposition
- Extraction
- Simplification
- Substitution
- ...

© Zhufei Chu
Elimination

- Eliminate a function from the network
  - Similar to Gaussian elimination
- Perform variable substitution

Example:
- \( f = r + a; \ r = b + c; \)
- \( f = a + b + c; \)
Elimination

- Eliminate a function from the network
  - Similar to Gaussian elimination
- Perform variable substitution

Example:
- \( f = r + a; \ r = b + c; \)
- \( f = a + b + c; \)
Decomposition

- Decompose a function into smaller functions
  - Opposite to elimination
- Introducing new variables into the network
- Example:
  - \( f = ad + bd + ac; \)
  - \( j = a + b; f = jd + ac; \)
Decomposition

- Decompose a function into smaller functions
  - Opposite to elimination
- Introducing new variables into the network
- Example:
  - \( f = ad + bd + ac; \)
  - \( j = a + b; f = jd + ac; \)
Extraction

- Finding common sub-expressions between two (or more) expressions
  - Extracting new sub-expressions as a new function
  - Introducing new blocks into the circuit

Example:
- \( p = bc + bd; t = ac + ad + b \);
- \( p = (c + d) b; t = (c + d) a + b \);
- \( k = c + d; p = kb; t = ka + b \);
Finding common sub-expressions between two (or more) expressions
- Extracting new sub-expressions as a new function
- Introducing new blocks into the circuit

Example:
- $p = bc + bd; t = ac + ad + b$
- $p = (c + d) b; t = (c + d) a + b$
- $k = c + d; p = kb; t = ka + b$
Simplification

- Simplifying local functions
  - Using heuristic minimization techniques, such as Espresso
  - Modifying the fanin of the target node
- Example:
  - \( f = a'b + ab' + ab \);
  - \( f = a + b \);

\[ f = a'b + ab' + ab \]
Simplification

- Simplifying local functions
  - Using heuristic minimization techniques, such as Espresso
  - Modifying the fanin of the target node
- Example:
  - \( f = a'b + ab' + ab \)
  - \( f = a + b \)
Substitution

- Simplifying local functions by introducing new variables

Example:

- $f = ka + kb + c$;
- $f = kq + c$;
- Because $q = a + b$ is already part of the network
Substitution

- Simplifying local functions by introducing new variables

Example:

- \( f = ka + kb + c; \)
- \( f = kq + c; \)
- Because \( q = a + b \) is already part of the network
Optimization approach

- Algorithmic approach
  - Define an algorithm for each transformation type
  - Algorithm is an *operator* on the network
  - Algorithms are sequenced by *scripts*

- Rule-based approach
  - Rule data base
    - Set of pattern pairs
  - Pattern replacement is driven by rules

- AI-based approach
  - AI explores *combinations*(*scripts*) of existing optimizations
  - AI-based novel optimization method

Most modern tools use the algorithmic approach to synthesis, even though rules are used to address specific issues, AI has a better QoR
Exact synthesis

Exact synthesis refers to synthesizing logic representations that exactly meets a specification.

- It is not used in opposition to approximate synthesis.
- Exact synthesis can be implemented in several methods
  - Boolean Satisfiability (SAT)
  - Integer Linear Programming (ILP)
  - Satisfiability modular theory (SMT)
  - ...

We may ask “Does there exist a logic network $N$ such that $N$ implements $f$ with exactly $r$ gates?”
SAT-based exact synthesis

- SAT-based exact synthesis essentially solves the **two** tasks
  - Find logic network topology (**DAG**)
  - Assign **Boolean operators** to vertices

SAT-based exact synthesis
Exact synthesis algorithm

- Given a Boolean function $f$, for size-optimum synthesis, take the SAT encoding method as an example:
  - Start with $r \leftarrow 0$
  - Encode the question as SAT formulas $F_r$
  - Feed the formulas to a SAT solver and wait for its result
  - If the result is SAT, we obtained an optimal result, otherwise, we set $r \leftarrow r + 1$ and continue the encoding

- Hence, the size-optimum problem can be solved by a sequence of SAT formulas.
Exact synthesis - Encode

**Variables**

- **Simulation:** $x_{it}$: $t^{th}$ bit of $x_i$’s truth table
- **Output:** $g_{hi}$: $[g_h = x_i]$ for $1 \leq j < k < i$
- **Selection:** $s_{ijk}$: $[x_i = x_i \circ_i x_k]$ for $1 \leq j < k < i$
- **Operands:** $f_{ipq}$: $\circ_i(p, q)$ for $0 \leq p, q \leq 1, p + q > 0$

**Running example**

<table>
<thead>
<tr>
<th>$t$</th>
<th>$x_{4t}$</th>
<th>$x_{5t}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1 0 0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 0 0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 0 0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 0 0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 0 0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 0 0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 0 0 0 1 0 0</td>
<td></td>
</tr>
</tbody>
</table>

$k$ = 2 3 4

$g_{14} = 1$, $g_{15} = 0$, $g_{24} = 0$, $g_{25} = 1$

$p, q = 0, 1, 1, 0, 1, 1$

$f_{Apq}$ = 0 0 1

$f_{5pq}$ = 1 1 0


©Zhufei Chu
Exact synthesis - Encode

Main clauses

\[
\begin{align*}
((s_{ijk} \land (x_{it} \oplus \bar{a}) \land (x_{jt} \oplus \bar{b}) \land (x_{kt} \oplus \bar{c})) \rightarrow (f_{ibc} \oplus \bar{a})) & \quad \text{Constrain the right Boolean operation} \\
(\bar{s}_{ijk} \lor (x_{it} \oplus a) \lor (x_{jt} \oplus b) \lor (x_{kt} \oplus c) \lor (f_{ibc} \oplus \bar{a})) & \\
(\bar{g}_{hi} \lor (\bar{x}_{it} \oplus g_{h}(t_1, \ldots, t_n))) & \quad \text{Constrain the output value they point to} \\
\bigvee_{i=n+1}^{n+r} g_{hi} & \quad \text{Each output is realized by the network} \\
\bigvee_{i=1}^{i-1} \bigvee_{k=1}^{k-1} \bigvee_{j=1}^{j-1} s_{ijk} & \quad \text{Each gate has exactly two inputs}
\end{align*}
\]

Additional constraints

- Breaking symmetry
  - E.g.
    - Only non-trivial operands
    - Use all steps
    - ...

- DAG Topology
  - E.g.
    - Fences
    - Partial DAGs
    - ...

Technology Mapping
The recursive tree-covering algorithm can be used to implement ASIC mapping in a simple and near-optimal manner.

Step:

- Mapped netlist and the logic gate of the technology library
  - Describing a netlist using only NAND2 and NOT logic gates.
  - The logic gates in the technology library are the same as above, and their corresponding costs are calculated.

- Tree transformation
  - Splitting nodes with multiple outputs into trees.

- Matching minimum implementation cost
  - Recursively searching the library for matching logic gates to implement at minimum cost.
ASIC technology mapping

- **Simple gate mapping**

  - Applying De Morgan's law to a Boolean function to convert it into a representation using only NAND2 and NOT logic gates, for example:

  - \[ t_1 = d + e; \]
  - \[ t_2 = b + h; \]
  - \[ t_3 = at_2 + c; \]
  - \[ t_4 = t_1t_3 + fgh; \]
  - \[ F = t_4'; \]

  - \[ t_1 = d + e = \text{NAND}(\overline{d}, \overline{e}) \]
  - \[ t_2 = b + h = \text{NAND}(\overline{b}, \overline{h}) \]
  - \[ t_3 = at_2 + c = at_2 \cdot \overline{c} = \text{NAND}(\text{NAND}(a, t_2), \overline{c}) \]
  - \[ t_4 = t_1t_3 + fgh = \text{NAND}(t_1, t_3, \overline{fgh}) \]
  - \[ fgh = \overline{fh} \cdot g = \overline{fh} \cdot g \]
  - \[ F = \overline{t_4} = \text{NAND}(\text{NAND}(f, h), g) \]
ASIC technology mapping

Simple gate mapping

- For the logic gates in a library of technology, they can be represented using both NAND2 and NOT logic gates.
ASIC technology mapping

- Tree transformation
  - To apply tree covering algorithm, we must first have a tree!

Obtain three trees from left to right.
Minimum Spanning Tree Cover

- Implementing minimum vertex cover using a recursive algorithm.
  - Start with the output of the graph.
  - For each node, find all the ways it can be matched.
  - Node $i$ cost function is as follows:
    \[
    \text{cost}(i) = \min_k \left\{ \text{cost}(g_i) + \sum_k \text{cost}(k_i) \right\}
    \]
  - $g_i$ represents a logical gate, and $k_i$ represents the input to $g_i$. 
FPGA technology mapping

- How is it different from ASIC (standard cells)
  - Structural in nature, simpler
  - Any function with k inputs can be mapped into a k-LUT
  - Typically implemented by cut mapping

FPGA architecture: k-LUT

\[ f = x_1'x_2' + x_1x_2 \]

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>
FPGA technology mapping

A possible mapping onto 3-LUTs
Overview of Logic Synthesis Tools
Industry tools

- Synopsys Design Compiler (DC)
  - For ASIC, DC NXT, Fusion Compiler
  - For FPGA, Synplify
- Cadence Genus
  - For ASIC
- FPGA Vendor
  - Xilinx Vivado
  - Altera Quartus Prime
  - Pango Design Suite (PDS)
  - ...
Academic tools

- **ABC**
  - [https://github.com/berkeley-abc/abc](https://github.com/berkeley-abc/abc)
  - Alan Mishchenko, UC Berkeley
  - Predecessor: SIS, MIS, VIS

- **Yosys**
  - [https://github.com/YosysHQ/yosys](https://github.com/YosysHQ/yosys)
  - Claire Xenia Wolf, use ABC for logic optimization

- **EPFL open-source logic synthesis library**
  - [https://github.com/lsils/lstools-showcase](https://github.com/lsils/lstools-showcase)

- **ALSO**
  - [https://gitee.com/zfchu/also](https://gitee.com/zfchu/also)

- Espresso...
Demo
ALSO
Q & A

chuzhufei@nbu.edu.cn

https://zfchu.github.io/

©Zhufei Chu