Cycles in Combinational Circuits

- Digital circuits are called combinational if they are memory-less: they have outputs that depend only on the current values of the inputs.

- Common misconception: combinational circuits cannot contain cycles.

- There exists a class of combinational circuits whose minimum implementation MUST necessarily be cyclic.

\[ f_1 = x_1'x_2' + x_1'x_3' + x_2'x_3' \]
\[ f_2 = x_1x_2 + x_1x_3 + x_1'x_2'x_3' \]
\[ f_3 = x_1x_2' + x_2'x_3 + x_1'x_2x_3' \]

Cyclic Combinational Circuits

• Can they arise during synthesis?
• Recall Boolean optimization using DCs
  – Cycles result when you allow variables in the transitive fanout of a node to appear in it’s don’t cares (SDCs + ODCs) and use them in optimizing the node
• Many design automation tools break when they see combinational cycles
• Algorithms can be enhanced to work with cycles at the cost of modest slowdown


(my Design Automation course project!)

Summary: Technology-independent Multi-level Synthesis

• Boolean network model
• Algebraic transformations
• Boolean optimization using SDCs and ODCs
ECE 595Z
Digital VLSI Design Automation

Module 5 (Lectures 14-20): Multi-level Synthesis
Lecture 19

Anand Raghunathan
MSEE 348
raghunathan@purdue.edu
Putting it together ...

**Multi-level Minimization in Practice** :
MIS / SIS
Multi-level Minimization in Practice: MIS / SIS

- Implement a wide range of useful optimization steps, exposed as commands to user
- Scripts: User specified “recipes” or sequences of steps

Example:

```
script.rugged

sweep
eliminate -1
simplify -m nocomp
eliminate -1

sweep
eliminate 5
simplify -m nocomp
resub -a
fx
resub -a
sweep
eliminate -1
sweep
full_simplify -m nocomp
```
Anatomy of a synthesis script

- Command: **sweep**
- Removes all nodes with a constant (0 or 1) function and all nodes with only 1 input
- Periodically “clean up” such nodes produced by other operations

```plaintext
script.rugged

sweep
eliminate -1
simplify -m nocomp
eliminate -1

sweep
eliminate  5
simplify -m nocomp
resub -a
fx
resub -a
sweep

eliminate -1
sweep
full_simplify -m nocomp
```
Anatomy of a synthesis script

• Example: `sweep`

```bash
sis> read_eqn sweep.eqn
sis> print
    F = a
    \{G\} = F
    \{H\} = F
    \{Q\} = a + a'
sis> sweep
sis> print
    \{Q\} = a + a'
    \{G\} = a
    \{H\} = a
```
Anatomy of a synthesis script

• Command: `eliminate <threshold>`

• Eliminates all nodes whose “value” is ≤ threshold by collapsing them into their fanouts

• Value represents the number of literals saved by keeping the node
  - Approximated by number of times a node output appears in the factored form of its fanouts

```
script.rugged

sweep
eliminate -1
simplify -m nocomp
eliminate -1

sweep
eliminate 5
simplify -m nocomp
resub -a
fx
resub -a
sweep
eliminate -1
sweep
full_simplify -m nocomp
```
Anatomy of a synthesis script

• Example: eliminate
Anatomy of a synthesis script

- Command: `simplify`
- Minimize SOP expression for each node in the network using a subset of the implicit don’t cares
- “-m nocomp” means use ESPRESSO without computing the full off set
- Multiple options for how to compute don’t cares, default uses a subset of transitive fanin of the node
- `full_simplify`: similar, except full-blown computation of don’t cares

```plaintext
script.rugged

sweep
eliminate -1
simplify -m nocomp
eliminate -1

sweep
eliminate 5
simplify -m nocomp
resub -a
fx
resub -a
sweep
eliminate -1
sweep
full_simplify -m nocomp
```
Anatomy of a synthesis script

• Example: \textit{simplify}

\[
\begin{align*}
F &= a + a'b + c \\
G &= a + a' \\
G_1 &= a + b + c
\end{align*}
\]
Anatomy of a synthesis script

- Command: **resub**
- Re-substitute each node into every other node in the network
  - Explores using both the node output and its complement
- “-a” : use algebraic division
- Keeps iterating until network (literal count) improves

```
script.rugged

sweep
eliminate -1
simplify -m nocomp
eliminate -1

sweep
eliminate 5
simplify -m nocomp
resub -a

fx
resub -a
sweep
eliminate -1
sweep
full_simplify -m nocomp
```
Anatomy of a synthesis script

• Example: resub

Resub example 1

F = ab
G = ab+c
H = ab+e

resub

F = ab
G = F +c
H = F+e

Resub example 2

F = ab
G = ab+c
H = a’ + b’ + cd

resub

F = ab
G = F +c
H = F’ +cd

Note: F was complemented
Anatomy of a synthesis script

• Command: **fx**
  - Finds all single-cube and double-cube divisors of nodes in the network
• Greedily extracts the “best” divisor as a node
• Usually followed by resub to see if the extracted factors are worth keeping
• Also see: commands **gcx, gkx**
  - Use the techniques we spoke about in class for kernels / co-kernels
Anatomy of a synthesis script

- Example: fx

\[ F = ab + c + x \]
\[ G = abx + cx + d \]
\[ H = ab + d \]

Note: fx creates new nodes by extracting common factors and resub substitutes existing network nodes into each other.
Anatomy of a synthesis script

- Overview of strategy used in script.rugged
- Four phases of optimization
  - Simpler to more complex
- Uses algebraic division for extracting factors and substitution
- Boolean optimization for node simplification

script.rugged

Simple clean-up
- sweep
- eliminate -1
- simplify -m nocomp
- eliminate -1

Round of “easy” factoring
- sweep
- eliminate 5
- simplify -m nocomp
- resub -a
- fx
- resub -a
- sweep

Round of aggressive factoring
- eliminate -1
- sweep
- full_simplify -m nocomp

Optimize each node
- sweep
- eliminate -1
- simplify -m nocomp
- resub -a
- fx
Summary

• Multi-level synthesis
  – Technology independent (completed)
    • Boolean network model
    • Operations: Extraction, Substitution, Elimination, Decomposition, Simplification
    • Algebraic model for factoring
      – Kernels / co-kernels
      – Algorithms using 0-1 matrices
    • Boolean optimization using don’t cares
    • Synthesis in MIS / SIS
  – Technology mapping
    • We will cover this next
Further Reading

  - Shows that only double-cube divisors are sufficient to detect whether common multi-cube divisors exist
  - Excellent results: Synthesized circuits have similar quality to kernel-based factoring, but 10X faster!
Technology Mapping : From Boolean Networks to Gates
Technology Mapping in the Logic Synthesis Flow

- Technology independent optimization produces a good “rough” structure for the network
- Technology mapping realizes the network using gates from a cell library
Technology Mapping

• Given
  – A Boolean network (already optimized using technology independent optimizations)
  – A library that contains cells (gates) that can be used, with models for area, delay, power

• Determine how to implement the given network using gates from the library *(optimally)*

Unmapped network
Cell Library

- Contains variety of primitives (cells, or simple and complex gates)
  - Commercial libraries have dozens (or hundreds) of logic cells
  - Each function in different “drive strengths”
  - Richer cell libraries usually lead to better quality of results, while increasing the complexity of technology mapping
Approaches to Technology Mapping

- **Rule-based** (LSS, SOCRATES)
- **Structural Pattern Matching** (DAGON, MIS/SIS)

  - Represent each node of the network as a set of base functions (primitive gates):
    - Must be complete
    - Typically 2-input NAND and INVERTER
    - Network becomes a *subject graph*
  
  - Each gate of the library is likewise represented using the base set. This results in *pattern graphs*.
  
  - Represent each gate in all possible ways
    - **Cover** the subject graph with pattern graphs

- **Boolean matching**
  - Exploit Boolean relationships to find more / better matches
  - Use BDD representations

Our focus
Subject Graph

- Decompose each node of the Boolean network into base functions
  - 2-input NAND and INVERTER

- **Subject graph** is a directed acyclic graph (DAG)

- Not unique, any decomposition is OK
Pattern Graph

- Each gate in the library is represented using the same base functions
  - **Pattern Graphs**
  - Not unique
    - Represent each gate in all possible ways

NAND4

Pattern Graphs for NAND4
Pattern Graphs for a Simple Library

- nand2(2)
- inv(1)
- nand3(3)
- nand4(4)
- nand4(4)
- aoi21(3)
- aoi22(4)
- and2(3)
- xor(5)
- nor2(2)
- nor3(3)
- nor4(4)
- or2(3)
- xnor(5)

Cost
Technology Mapping as a Graph Covering Problem

- A **cover** is a collection of pattern graph instances such that
  - Every node of the subject graph is contained in one or more instances.
  - Each input required by a pattern graph instance is a primary input or the output of some other pattern graph instance
- Need to find the **minimum cost** cover
  - For now, we assume that cost of the cover = sum of the costs of pattern graph instances

Example:

Cost of the cover (8 nand2 + 7 inv) =
Technology Mapping as a Graph Covering Problem

- Multiple solutions exist!

Example

Cost of the cover (1 aoi22 + 2 or2 + 1 and2 + 2 nand2 + 1 inv) =
Technology Mapping as a Graph Covering Problem

• Multiple solutions exist!

Example

\[
\text{Cost of the cover (2 oai21 + 1 and2 + 1 nand3 + 1 nand2 + 1 inv) } =
\]

Need a systematic approach to explore the design space
Technology Mapping Using Graph Covering

• General Approach
  – Construct a subject DAG (Directed Acyclic Graph) for the Boolean network
  – Represent each gate in the target library by pattern DAGs
  – Find an optimal-cost covering of subject DAG using the collection of pattern DAGs

• Challenge: Complexity of DAG covering
  – NP-hard
  – Remains NP-hard even when all nodes have in-degree ≤ 2

Two solution approaches

Binate Row Covering Problem

Decompose DAG into trees

If subject graph and pattern graph are trees (each vertex has an out-degree of 1), then an efficient algorithm exists!