Technology Mapping

Decomposition

Pattern Matching

Covering

Ryan Moffitt

Technology Mapping

- Given a Boolean network which represents a logic circuit and a cell library, technology mapping is the process of binding nodes in the network to cells in the library

Xiaoqing and Saluja, "A New Method Towards Achieving Global Optimality in Technology Mapping", p. 9

Technology Mapping

- Translates logic equations into a network of technology cells
- Transforms each and every cell in the network
- A three step procedure
  - Decomposition
  - Pattern matching
  - Covering

Logic Decomposition

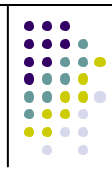
- Circuit modified to be a simple input for covering
  - Limits networks to cells with few primitive functions having few inputs
    - Practical systems are using 2-input NANDS and Inverters

When Logic Decomposition is Used

- When inputs of a cell contain both critical and non-critical parts.
- A gate that can not be sized up anymore due to gate library constraints.

Hassoun and Sasao, "Logic Synthesis and Verification", p. 229

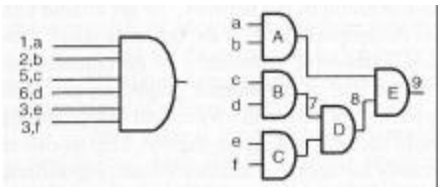
Decomposing



- Network decomposed into two input primitive cells
  - SOP represented by OR cell with one input per product term.
    - Product terms are represented by an AND cell with as many inputs as variables in the product term.
- Cells with more then two inputs can be decomposed recursively.

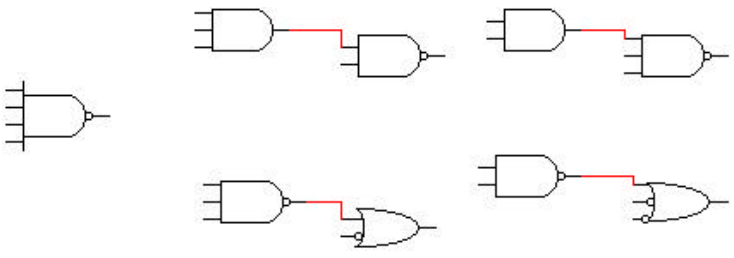
Hassoun and Sasao, "Logic Synthesis and Verification", p. 117

Example



Hassoun and Sasao, "Logic Synthesis and Verification", p. 118

Example for a 4 input NAND



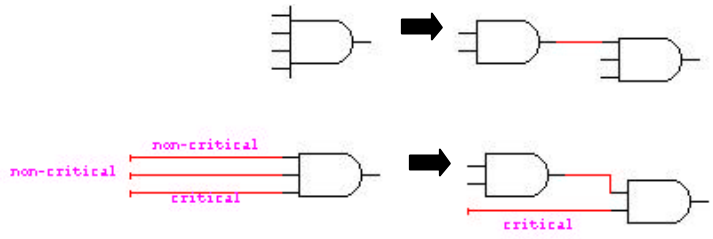
Lian and Lin, "Layout-based Logic Decompostion for Timing Optimization", p. 231

Simple Gate Decomposition

- Taking a larger gate and decomposing it into smaller gates
  - Ex. A 4 input AND gate can be made into a two input AND gate followed by a three input AND gate
- Used to reduce delay
- Advantage is the circuit remains mapped after the transform
- Disadvantage is area increases

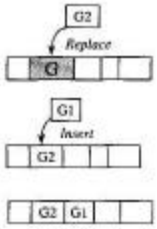
Hassoun and Sasao, "Logic Synthesis and Verification", p. 156-157

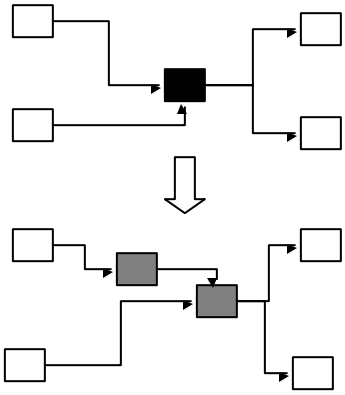
Simple Gate Decomp. Example



Lian and Lin, "Layout-based Logic Decompostion for Timing Optimization", p. 230

Change in overall Circuit





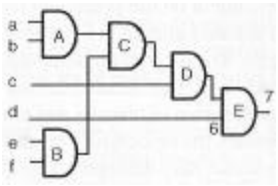
Lian and Lin, "Layout-based Logic Decompostion for Timing Optimization", p. 232

Mapping for Delay

- Sort cells from inputs to outputs
  - Guarantees that all predecessor cells will be decomposed
- Decompose each cell by extracting the two earliest arriving inputs and creating a new cell for them, feeding the output into the old cell
- Timing incremented
- Re-decompose if inputs > 2
- Unbalanced arrival times for the signals
  - Skew decomposition towards the arrival times

Hassoun and Sasao, "Logic Synthesis and Verification", p. 118

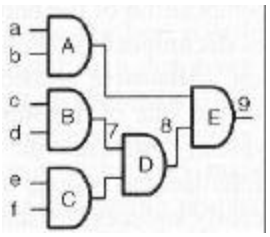
Delay Mapping Example



Hassoun and Sasao, "Logic Synthesis and Verification", p. 118

Input Swapping

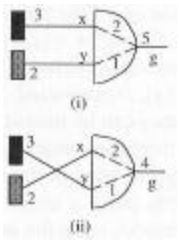
- More Timing Driven
- Example
  - Swap input b with the topmost signal of cell B
  - Swap input c with the bottom signal of cell A
  - Transformed (b) into (c)



Hassoun and Sasao, "Logic Synthesis and Verification", p. 118

Pin Permutation

- Minimizes delay by connecting the signal with the largest delay to the input with the smallest internal delay in a symmetric gate.



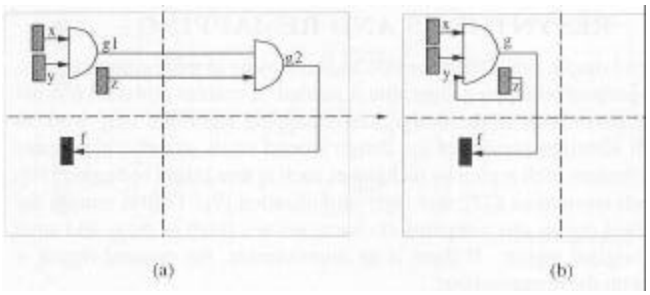
Hassoun and Sasao, "Logic Synthesis and Verification", p. 158-159

Rule-Based Optimization

- Special case of remapping
- Rules were based on technology library and were handcrafted for that technology
- Identifies a portion of the circuit that is isomorphic
  - Ex. If a chain of two AND gates has a delayed input x, x should be moved to the second AND gate, this would improve the delay time
  - A NAND-NOT-NOR should be replaced with an AND-OR-INVERT.

Hassoun and Sasao, "Logic Synthesis and Verification", p. 158

Simple Gate Collapsing



Hassoun and Sasao, "Logic Synthesis and Verification", p. 157

Multiple Decompositions

- Allows more freedom during pattern generation and covering
  - Embeds multiple decompositions in the graph
- Adding pairs of cascaded inverters into the subject Directed Acyclic Graph (DAG)
  - Can find patterns for positive and negative functions as well as input inversions

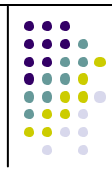
Hassoun and Sasao, "Logic Synthesis and Verification", p. 118

Placement

- Decomposition effects the circuit structure and the placement of the cells
- Ideal to pick decompositions which result in good placement
  - Results in a lower cost system with less congestion

Hassoun and Sasao, "Logic Synthesis and Verification", p. 119

Pedram Decomposition



- Decomposes a DAG so that signals coming from nearby regions of the network enter the tree at topologically near points
- A companion placement is done on the DAG before decomposition

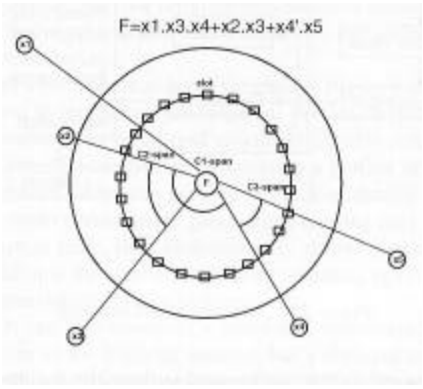
Hassoun and Sasao, "Logic Synthesis and Verification", p. 119

Companion Placement Example

- Decompose vertex F with inputs x1,…, x5
- Replace each net with a direct connection from its source to all its sinks.
  - Circularly traversing around the node determines a unique ordering for the input signals

Example Diagram

- Slots are placed on an imaginary circle
- Span for each cube = the shortest circular distance between all of its variables
- Cost is zero if the slot falls within the span of the cube, otherwise it is the distance from the slot to the nearest edge of that cube

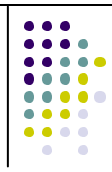


Results

- Ordering of the cubes that can drive the decomposition
- Cubes with signals that come from the same input direction have a lower cost and therefore more likely to be extracted.
- Biases decomposition to create a network with simplified wiring.

Hassoun and Sasao, "Logic Synthesis and Verification", p. 119

Layout Area and Congestion



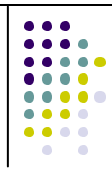
- Little placement information is provided
  - Potential wire congestion and routing problems
- Bhat and Pedram global point placement
  - Each node of the subject graph is assigned an (x,y) coordinate.
    - Location used in estimating wiring cost, and modify the area calculations, or modify the delay cost and take the wiring delays into account

Bhat and Pedram, "Layout Driven Technology Mapping", p. 101

Gate Delay Models

- Load Independent Delay (LIDM)
  - $\delta(i,g) = \alpha(i,g)$
    - g: gate, i: input pin, $\delta(i,g)$: delay, $\alpha(i,g)$: intrinsic delay
- Load Dependent Delay (LDDM)
  - $\delta(i,g) = \alpha(i,g)+\beta(i,g)Cg$
    - $\beta(i,g)$: load coefficient, Cg: load capacitance

Gate Delay Models



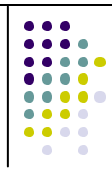
- Input-Slew Dependent Delay (ISDDM)
  - $\tau(h) = \phi(h) + \rho(h)Ch$
    - h: chip, $\tau(h)$: slew, Ch, cap. load seen by h
  - $\delta(i,g) = \alpha(i,g) + \beta(i,g)Cg + \kappa(i,g)\tau(h) + \upsilon(i,g)Cg\tau(h)$
  - $\beta\ \alpha\ \rho\ \phi\ \upsilon$ and κ are determined by gathering delay and slew data for a range of input slew and output load values
- ISDDM is popular in industrial libraries, but most time related research has used the other two models (LIDM and LDDM)

Different Delays

- Delay on a Path
  - Delay from a PI to a PO, or a latch output to a latch input is the sum of the pin to pin delays δ through the gates plus the delay in the wires on that path
- Circuit Delay
  - Maximum of all path delays in the circuit

Hassoun and Sasao, "Logic Synthesis and Verification", p. 143-144

Traces



- Forward Delay Trace
  - Computes signal arrival times at every pin for every gate
- Backward Delay Trace
  - Computes required times of signals at every pin of every gate

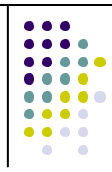
Hassoun and Sasao, "Logic Synthesis and Verification", p. 144

Slack and Criticality

- Slack
  - Difference between its required time and its arrival time for a pin
- Critical
  - A pin/pad is critical if its slack is negative
- ε-criticality
  - If a pin/pad slack lies within ε of the minimum slack of the circuit

Hassoun and Sasao, "Logic Synthesis and Verification", p. 144

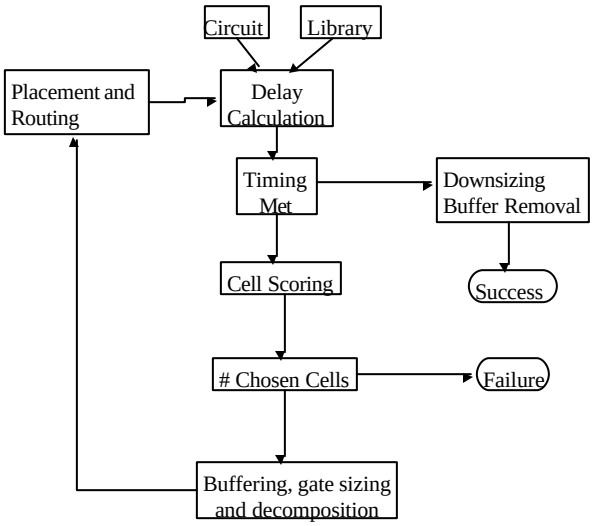
Gate Resizing



- Gate library contains several different sizes of each gate in the circuit
- Objective it to select the size of each gate so that a certain objective is met (i.e. circuit delay, circuit area) without violating any constraints.
- Minimal effect on the placement and routing of cells

Hassoun and Sasao, "Logic Synthesis and Verification", p. 144-147

Gate Resizing Algorithm



Lian and Lin, "Layout-based Logic Decomposition for Timing Optimization", p 230

Improving Speed

- Logic Decomposition
- Gate Sizing
  - Size Up increases speed
  - Size Down decreases area
- Buffering
  - Useful in nets with large fanouts
  - Inserted into critical nets to decouple fanout loading
- Wire Sizing

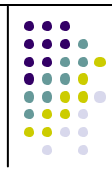
Lian and Lin, "Layout-based Logic Decomposition for Timing Optimization", p 230

Advantages and Disadvantages

- Disadvantages
  - Logic Decomposition increases area
  - Gate sizing, adding buffers and wire sizing are all limited by the library and the netlist structure
- Advantages
  - Decomposition isn't limited to the netlist structure
  - Gate sizing, Buffering, and Wire Sizing don't change the netlist structure

Lian and Lin, "Layout-based Logic Decomposition for Timing Optimization", p 231

Resynthesis and Remapping Algorithm



- Identify a region of the design around critical gates
- Resynthesize these regions
- Remap the resynthesized regions
- Compares the new region with the old region
- If there is an improvement the original is replaced

Hassoun and Sasao, "Logic Synthesis and Verification", p. 158

Experimental Results

	SISGS	G		GD		GB		GBD	
Circuit	delay	delay	improve (%)	delay	improve (%)	delay	improve (%)	delay	improve (%)
C1355	11.94	11.11	6.95	9.92	16.92	10.34	13.40	9.5	20.44
C5315	19.13	15.24	20.33	14.73	23.00	15.19	20.60	14.48	24.31
s298	5.56	4.56	17.99	4.48	19.42	4.56	17.99	4.5	19.06
s1423	14.23	13.29	6.61	12.12	14.83	13	8.64	12.1	14.97
s5378	13.27	10.11	23.81	9.75	26.53	9.75	26.53	9.57	27.88
s9234	10.93	8.84	19.12	8.25	24.52	8.77	23.52	8.1	25.89
s13207	14.65	13.83	5.60	12	18.09	13.7	18.09	11.84	19.18
s15850	21.4	16.93	20.89	15.51	27.52	16.08	27.52	15.3	28.50
s35932	108.18	97.29	10.07	88.62	18.08	91.38	18.08	83.22	23.07
s38417	43.52	29.34	32.58	20.24	53.49	23.57	53.49	21.35	50.94

G: Gate Resizing, D: Decomposition, B: Buffer Insertion

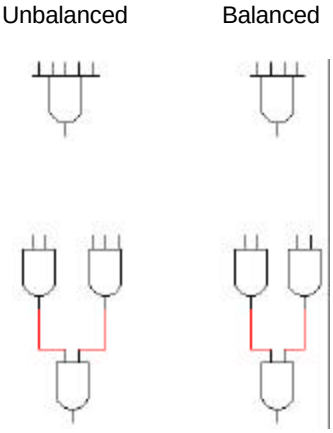
Lian and Lin, "Layout-based Logic Decomposition for Timing Optimization", p 232

Optimizing Time in Multi-Level Logic

- Recent systems use simple 2-way decomposition
- Decomposition relies on precise linear models for gate delays.
- Performs locally optimal m-way balanced and unbalanced decompositions to achieve maximal timing gain.
  - Take the output load and are characterized by a near minimal area increase.
  - Speedup is nearly twice compared to 2-way decomposition

Paulin and Poirot, "Logic Decomposition Algorithms for the Time Optimization of Multi-Level Logic", p. 329

Balanced and Unbalanced



Paulin and Poirot, "Logic Decomposition Algorithms for the Time Optimization of Multi-Level Logic", p. 332

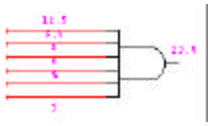
2-Way vs. M-Way

- 2-Way decomposition
  - All gates are the same including number of inputs
- M-Way decomposition
  - Gates can be different including number of inputs per gate
  - Results in a lower area then 2-way

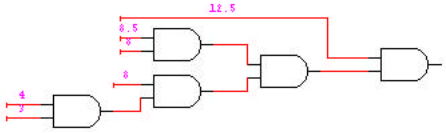
Paulin and Poirot, "Logic Decomposition Algorithms for the Time Optimization of Multi-Level Logic", p. 329

Example

Original



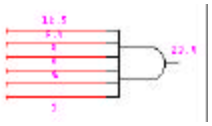
2-Way Decomposition



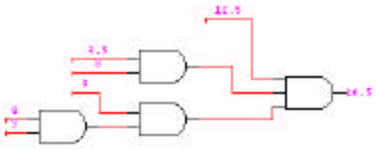
Paulin and Poirot, "Logic Decomposition Algorithms for the Time Optimization of Multi-Level Logic", p. 331

Example Continued

Original



M-Way Decomposition



Paulin and Poirot, "Logic Decomposition Algorithms for the Time Optimization of Multi-Level Logic", p. 331

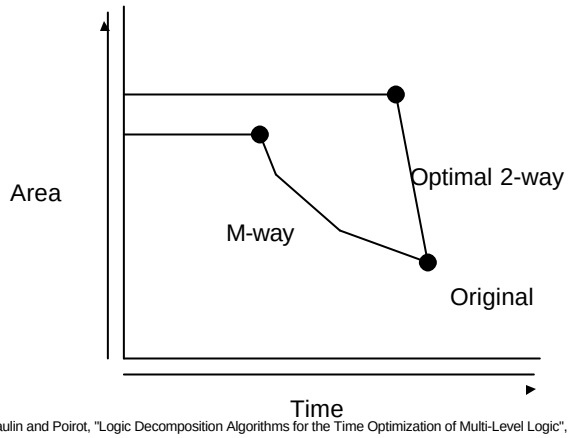
Results

- Original Area: 9; Original Time: 23.5

- 2-Way Decomposition
  - Area: 25; 278% bigger
  - Time: 23; 2.17% faster
- M-Way Decomposition
  - Area: 21; 233% bigger
  - Time: 20.5; 15% faster

Paulin and Poirot, "Logic Decomposition Algorithms for the Time Optimization of Multi-Level Logic", p. 331

Time-Area Tradeoffs



Paulin and Poirot, "Logic Decomposition Algorithms for the Time Optimization of Multi-Level Logic", p. 331

Results

	no decomposition		2-way		m-way	
Name	size	delay	size	delay	size	delay
bw	140.0	23.0	140.0	23.0	146.5	21.3
duke2	295.0	32.5	307.0	29.1	305.0	28.9
f51m	111.5	21.8	112.0	21.3	112.0	21.0
misex1	45.0	15.2	46.5	15.2	46.5	15.0
rd53	36.0	17.4	42.0	17.4	43.0	13.8
rd73	81.5	22.4	85.0	18.5	87.5	19.3
sao2	152.0	28.4	157.5	25.5	159.0	24.7
big2	125.0	22.0	125.0	22.0	127.0	21.4
ctrl2	266.0	28.0	268.5	27.8	268.5	27.8
fifowrite	126.0	23.5	126.0	26.5	129.5	20.3
flammand	149.5	25.5	149.5	25.5	152.0	23.9
icdma	104.5	22.4	104.5	22.4	105.5	22.2
planet	645.0	35.6	705.0	31.7	705.5	31.7
uar	395.0	33.8	397.0	31.7	401.0	30.6
TOTAL	2672.0	351.5	2765.5	334.0	2788.5	321.9
Ratios			3.50%	-5.20%	4.40%	-9.20%
dA/dt			0.67		0.48	

Paulin and Poirot, "Logic Decomposition Algorithms for the Time Optimization of Multi-Level Logic", p. 333

Summary

- Logic Decomposition
  - Simplifying the circuit for speed, area, and better covering
- Simple Decompositions
- Mapping for Delay
  - Makes sure the slowest signals are inputs to the latest gates
- Improving Speed
  - Gate sizing, buffering, and decomposition
- M-Way Decomposition
  - results in a slightly larger circuit but gives a faster circuit compared to 2-way decomposition

Technology Mapping

Decomposition

Pattern Matching

Covering

Hakan Karsligil

Technology Mapping

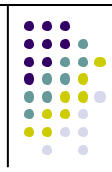
- Technology mapping transforms a technology independent logic network into gates implemented in a technology library [1].
- Conventional technology mapping can be described as a three step procedure:

[1] L. Stok and V. Tiwari, "Technology Mapping." Logic Synthesis and Verification

Technology Mapping

- Decomposition
- Pattern Matching
  - Common matchers are:
    - Structural matcher
    - Boolean matcher using BDDs
    - PLA matchers
- Covering

Decomposition



- Decomposition:
  - Decomposing technology independent circuits in terms of primitive cells to have a simple logic structure to aid the technology mapping process is referred to as Decomposition [1].
  - Circuits need to be modified to be a simple input for covering.

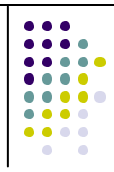
[1] L. Stok and V. Tiwari, "Technology Mapping."
Logic Synthesis and Verification

Pattern Matching

- One of the crucial tasks for technology mapping is the process of matching, which tries to determine which cells in the library may be used to implement a set of nodes in the *subject Boolean network* [2].
- Pattern Matching
  - Common matchers are:
    - Structural matcher
    - Boolean matcher using BDDs
    - PLA matchers

2. A new structural pattern matching algorithm for technology mapping
Zhao, M.; Sapatnekar, S.S., Design Automation Conference, 2001. Proceedings , 2001, Page(s): 371 -376

Structural Matching Algorithm [2]



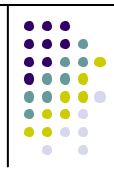
- The matcher is based on a key observation that the matches for a node in a subject Boolean network are related to the matches for its children.
- The structural relationships between the library cells are modeled using a lookup table.

2. A new structural pattern matching algorithm for technology mapping
Zhao, M.; Sapatnekar, S.S., Design Automation Conference, 2001. Proceedings , 2001, Page(s): 371 -376

Structural Matching Algorithm [2]

- A straightforward method is to match each pattern at each node of the subject Boolean network.
- In general, the subject Boolean networks and library cells are decomposed into the same set of the simple functions, called *base functions*.

Structural Matching Algorithm [2]

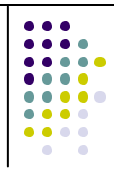


- The proposed structural matching algorithm:
  - The algorithm relates the matching of a node to the matching of its children.
  - Given a node in a subject network, the valid matches of the current node can be obtained from the valid matches of its children and the node type of the current node.

Structural Matching Algorithm [2]

- The proposed structural matching algorithm:
  - Each canonical structure or substructure of a library cell that could be represented by a tree is abstracted into a unique index, called pattern index.
  - The structural relationships between these structures or substructures are modeled into a lookup table, called pattern table.

Structural Matching Algorithm [2]



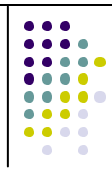
- First, we generate a pattern table from a given library; during the technology mapping procedure, we use this pattern table and perform the matching procedure.
- Pattern table consists of four parts: AND table, OR table, isGate array, and isInvGate array.

Pattern_Table_Generation Algorithm

Input: A Library
Output: AND table; OR table; isGate array; isInvGate array

1. Initialize each entry of the pattern table with invalid value
2. For each cell of library
3. Form a multiple-input AND/OR/NOT tree T
4. pattern_index = *Look_up*(T)
5. isGate[pattern_index] = 1
6. If there is inverter at the root node of T
7. isInvGate[pattern_index] = 1

Procedure *Look_up(tree T)*



Input: An AND/OR/NOT tree
Output: Pattern index of the tree; Entries in the pattern table
Global variable: newIndex(store number of the patterns)

1. If T is a leaf
2. return leaf_index
3. If T is an inverter before a leaf
4. return inv_leaf_index
5. If Pattern_index(T) is a valid value
6. Return Pattern_index(T)
7. NodeType = type of root node of tree T

Procedure *Look_up(tree T)*

8. For each two-input decomposition of root node of tree T, T'
9. indexL= *Look_up*(leftchild_of_T')
10. indexR= *Look_up*(rightchild_of_T')
11. If [NodeType,IndexL,IndexR] is a valid value
12. Pattern_index(T)=[NodeType,IndexL,IndexR]
13. Return Pattern_index(T)
14. Pattern_index(T)=[NodeType,IndexL,IndexR]=newIndex
15. Increment newIndex
16. Return Pattern_index(T)

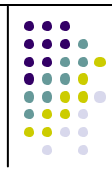
Structural Matching Algorithm [2]

- The isGate and isInvGate arrays are used to reflect the corresponding relations between a tree structure with a cell.
- The isGate array is an *M*-dimensional vector, where *M* is the number of pattern indices, that is used to indicate whether the pattern index representing a tree structure is a cell of a library or not.

Structural Matching Algorithm [2]

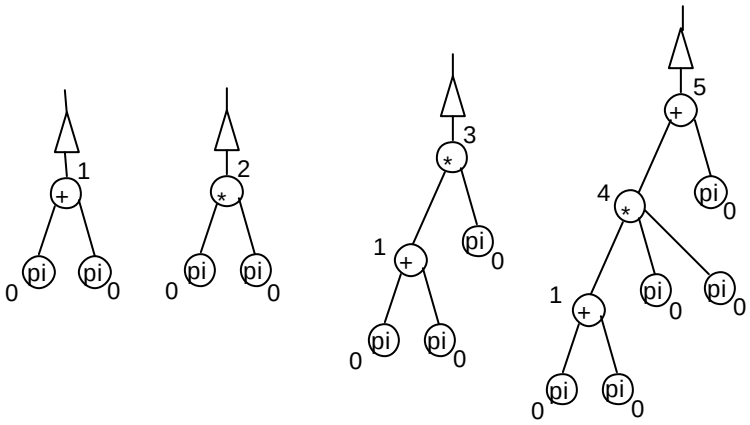
- The M-dimensional isInvGate vector is used to indicate whether there is an inverter at the root of a tree or not.
- The inverters in a tree structure are pushed to the root or the leaf nodes of the tree.

Example



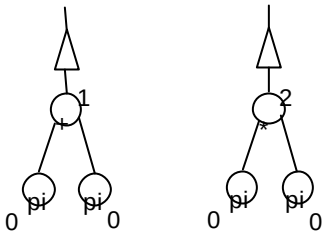
- *Leaf_index* is denoted by “0”.
- Each pattern except for pattern 4 corresponds to a cell of the library.
- All elements of isGate are set to “1” except at index 4.
- All of them are inverting complex gates and thus the corresponding elements of array *isInvGate* are set to “1”.

Example – Library Cells



An example of library composed of four cells

Example



AND	0	1	2	3	4
0	2	3	-	4	-
1	3	-	4	-	-
2	-	4	-	-	-
3	4	-	-	-	-
4					

isGate

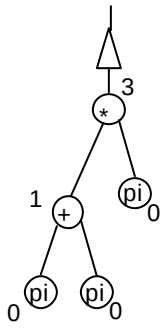
1	1
2	1
3	1
4	0
5	1

OR	0	1	2	3	4
0	1	-	-	-	5
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	5				

isInvGate

1	1
2	1
3	1
4	0
5	1

Example



AND	0	1	2	3	4
0	2	3	-	4	-
1	3	-	4	-	-
2	-	4	-	-	-
3	4	-	-	-	-
4	-	-	-	-	-

OR	0	1	2	3	4
0	1	-	-	-	5
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	5	-	-	-	-

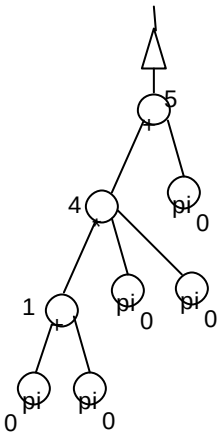
isGate

1	1
2	1
3	1
4	0
5	1

isInvGate

1	1
2	1
3	1
4	0
5	1

Example



AND	0	1	2	3	4
0	2	3	-	4	-
1	3	-	4	-	-
2	-	4	-	-	-
3	4	-	-	-	-
4	-	-	-	-	-

OR	0	1	2	3	4
0	1	-	-	-	5
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	5	-	-	-	-

isGate

1	1
2	1
3	1
4	0
5	1

isInvGate

1	1
2	1
3	1
4	0
5	1

Matching Part

- The relationship between matching of a node of the subject Boolean network and matching of its children will be explored.
- A matching method utilizing these two relationships will be shown.

Procedure Matching(current_node N)

Input: Match sets of N's children; The pattern table

Output: Match set of N

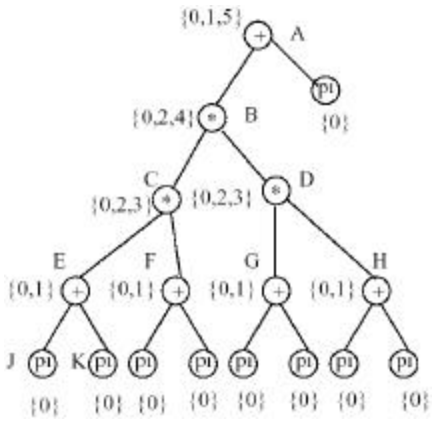
- NT= node type of N
- For each match of the left child, l
- For each match of the right child, r
- k = [NT,l,r]
- If (k is valid value)
- Insert k into the match set of N
- If isGate[k]==1
- cell_handling(k)
- Insert leaf_index, inv_leaf_index into the match set of N

Example - Matching

AND	0	1	2	3	4
0	2	3	-	4	-
1	3	-	4	-	-
2	-	4	-	-	-
3	4	-	-	-	-
4	-	-	-	-	-

OR	0	1	2	3	4
0	1	-	-	-	5
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	5	-	-	-	-

Lookup tables

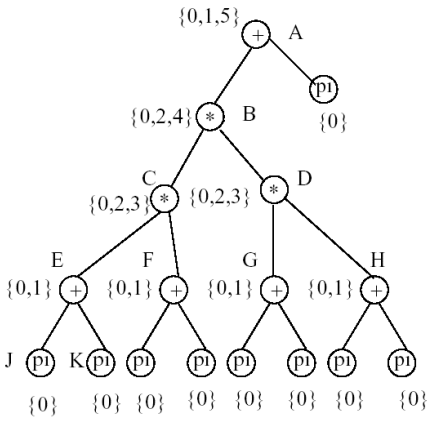


Matching of the subject network

Example - Matching

- This example uses the same library as the table generation example.
- The elements within curly braces next to each note constitute the match set of the node.
- All primary inputs are initialized with the match set {leaf_index}, where *leaf_index* is denoted by “0”.

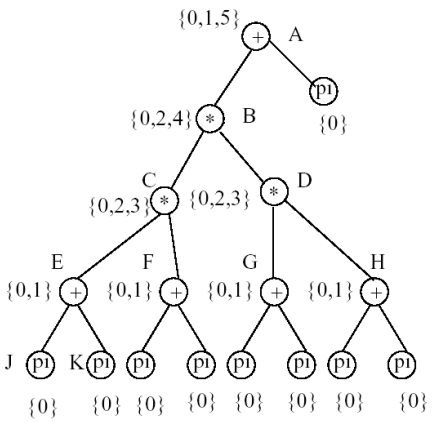
Example - Matching



Matching of the subject network

- At note B, by looking for every element of set {0,2,3}x{0,2,3} in the AND table, a match set of {0,2,4} is obtained.

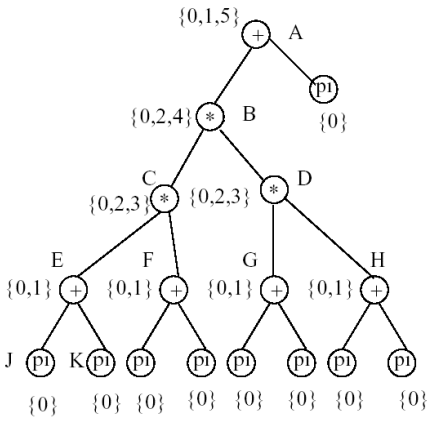
Example - Matching



Matching of the subject network

- This is because the lookups at [AND, 0, 0] and [AND, 0, 3] return patterns 2 and 4, respectively, and the other combinations return invalid values and therefore ignored.

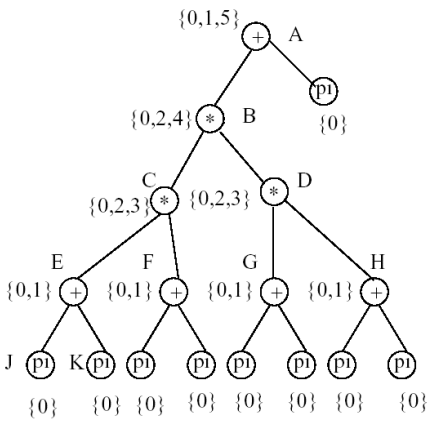
Example - Matching



Matching of the subject network

- The lookups return patterns 2 and 4. In addition, node B can be a leaf of a cell and thus pattern 0 also belongs to its match set, i.e. {0, 2, 4}.

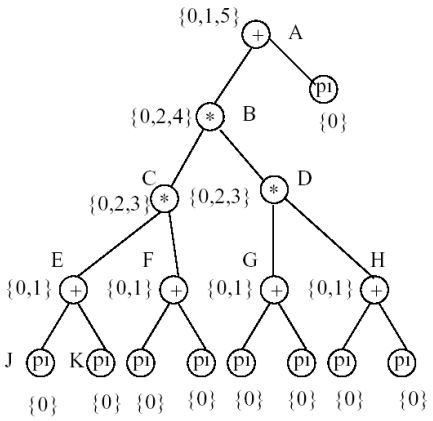
Example - Matching



Matching of the subject network

- When we look at *isGate* array, we can see that not every match set {0, 2, 4} of note B is a cell: pattern 4 is merely a substructure of pattern 5, and only pattern 2 is a real cell in the library.

Example - Matching

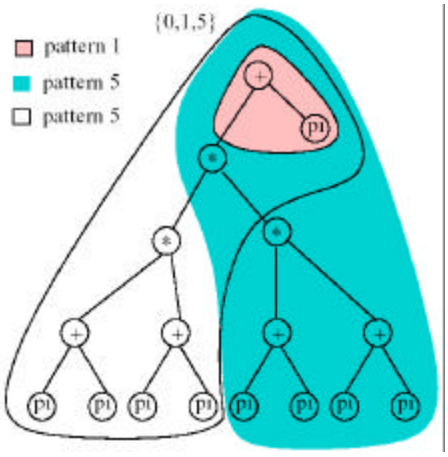


Matching of the subject network

- The objective of keeping pattern 4 in node B is to check the possibility of matching pattern 5 to the parent of node B.

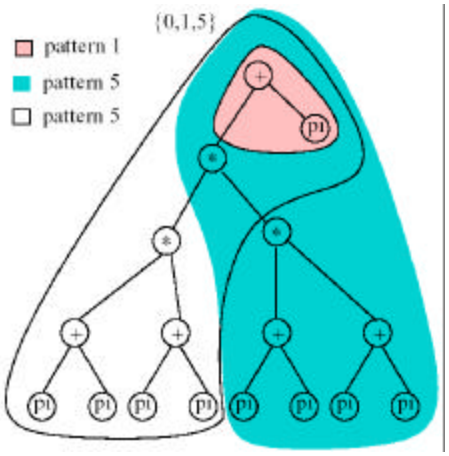
Example - Matching

- Figure illustrates the matches for Node A.
- {0, 1, 5} is the match set of Node A, of which pattern 1 and 5 are valid matched cells.



Example - Matching

- Two matches correspond to the same pattern 5.
- This is because both [AND, 0, 3] and [AND, 3, 0} return the same pattern 4 during the matching procedure at node B.



Pattern Matching

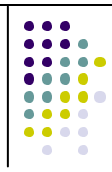
- Pattern Matching
  - Common matchers are:
    - Structural matcher
    - Boolean matcher using BDDs
    - PLA matchers

Boolean Matching [3]

- The method described in the paper relies on signature calculation for variables of Boolean functions.
- Signatures induce an ordering of the variables which is used to construct a BDD.

3. Efficient Boolean matching in technology mapping with very large cell libraries
Schlichtmann, U.; Brglez, F.
Custom Integrated Circuits Conference, 1993., Proceedings of the IEEE 1993 , 1993
Page(s): 3.6.1 -3.6.6

Boolean Matching [3]



- Boolean matching addresses the problem of determining the equivalence of two Boolean functions regardless of the structure of their representation under an arbitrary permutations of their inputs.
- Given two Boolean functions:
  $f(x_1,\ldots,x_n)$ *and* $g(y_1,\ldots,y_n)$, *we search for* $Z(x) : \{x_1,\ldots,x_n\} \Rightarrow \{y_1,\ldots,y_n\}$ *such that* $f(Z(x_1,\ldots,x_n)) = g(y_1,\ldots,y_n)$.

Boolean Matching [3]

- Boolean library matching contains the problem of Boolean matching. Given a library of cells, each represented with a Boolean function, and a specific Boolean function, one must determine whether the library contains a cell that implements the specific Boolean function.

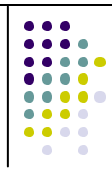
Boolean Matching [3]

- Assuming a library of k cells, each having *n* inputs, a brute force approach to the problem of Boolean library matching requires $k * n!$ verifications of two Boolean functions in the worst case.

Boolean Matching [3]

- Proposal:
  - A new approach to Boolean matching that efficiently prunes the search space by combining shared Reduced Ordered Binary Decision Diagrams (ROBDDs) with a unique ordering of variables.
  - The approach can efficiently deal with extremely large libraries, constraint only by available memory resources. Cells with reconvergent fanout poses no problems in this method.

General Strategy [3]



- Shared ROBBDs are not only memory efficient means of representing Boolean functions, they also guarantee a strong canonical representation.
  - This means that two identical ROBDDs will always reside in the same memory location.
- Therefore, equivalence of two Boolean functions can be tested by simply comparing the pointers to the ROBDDs of the respective functions

General Strategy [3]

- The key to efficient Boolean matching of an arbitrary Boolean function in *n* variables to an n-input cell in the library is the ability to generate a unique variable ordering for both the Boolean function and each of the cells in the library.

Boolean Matching [3]

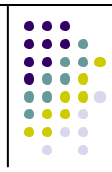
- We want to test if two Boolean functions are equivalent, we have to solve the problem of establishing a correspondence between the input variables of these two functions [4].
- Signatures are used to represent each function.

4. Limits of using signatures for permutation independent Boolean comparison
Mohnke, J.; Molitor, P.; Malik, S.
Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95.,
IFIP International Conference on Hardware Description Languages.
IFIP International Conference on Very Large Scale Integration., Asian and South Pacific , 1995
Page(s): 459 -464

Boolean Matching [3]

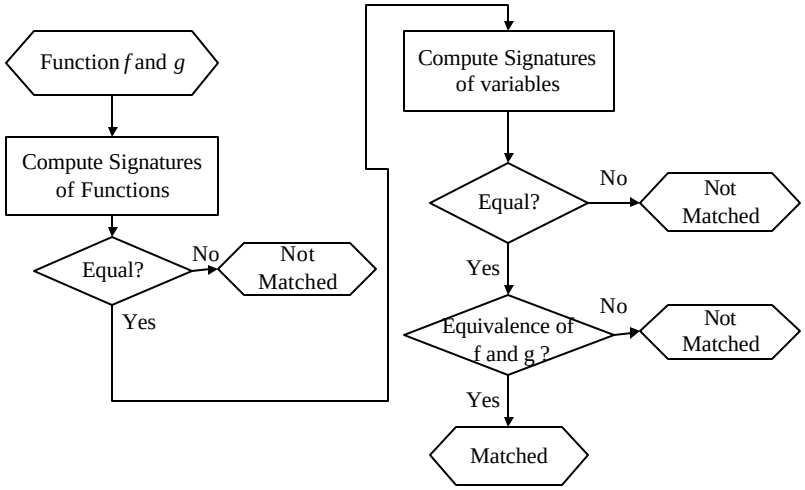
- A signature for an input variable x_i of a Boolean function f is a description of x_i which provides special information about that variable in terms of f [4].
- A signature may be a value or a vector of values as well as a special function.

Boolean Matching [3]



- We can use a signature to identify an input variable x_i to establish a correspondence between this variable x_i of f with variable x_k of any other Boolean function g [4].
- If we are to compute a unique signature for each input variable of f, then the correspondence problem is solved.

A General Paradigm of Boolean Matching



Signature Based Variable Ordering

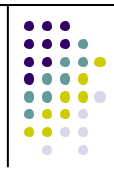
- Let $\underline{x}$ denote a vector ($x_1, x_2,\ldots,x_n$) of Boolean variables.
- Let $|f(\underline{x})|$ denote the number of minterms covered by $f(\underline{x})$
- The cofactor $f_{x_i}(\underline{x})$ of $f(\underline{x})$ with respect to x_i is obtained by setting x_i=1 in $f(\underline{x})$
- A **variable auto-signature** $S_{x_i}(f(\underline{x}))$ is a k-tuble of integers, each representing the number of minterms associated with Boolean function derived from $f(\underline{x})|_{x_i \notin \underline{x}}$

Signature Based Variable Ordering

- A **variable cross signature** $S_{x_i x_j}(f(\underline{x}))$ is a k-tuble of integers, each representing the number of minterms associated with a Boolean function derived from $f(\underline{x})\big|_{x_i x_j \notin \underline{x}}$
- Finally, we define a **function super-signature in natural order** as

$$s_{x_i x_j} = \begin{bmatrix} S_{x_1} & S_{x_1 x_2} & .... & S_{x_1 x_n} \\ S_{x_2 x_1} & S_{x_2} & & \\ .... & & & \\ S_{x_n x_2} & & & S_{x_n} \end{bmatrix} \qquad (1)$$

Signature Based Variable Ordering



- An ordering on S_{x_i} is imposed, which in turn orders the variables x_i
- If all S_{x_i} are distinct, the order of x_i is unique and there is no further need to evaluate the cross-signatures in (1).
- If auto-signatures are not unique, then the cross-signatures could be useful in providing additional information to find a unique ordering of the variables

Signature Based Variable Ordering

- We define the ordering relation $\prec$ for integers and vectors.
- For integers a, b it is defined as a $\prec$ b $\Leftrightarrow$ a<b.
- In the following figures we will consider a situation where the variable auto-signature S_{x_i} does not generate any useful information.

Signature Based Variable Ordering

$$f(\underline{x}) \quad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \quad (a)$$

- Given an on-set of a function $f(\underline{x})$ n=4 variables, we generate the function super-signature in natural order in (b).

Signature Based Variable Ordering

$s_{x_i x_j} =$

$$\begin{bmatrix} \boxed{2} & 2 & 1 & 0 \\ 2 & \boxed{2} & 1 & 2 \\ 1 & 1 & \boxed{2} & 1 \\ 0 & 2 & 1 & \boxed{2} \end{bmatrix} \quad (b)$$

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 2 & 2 & 1 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix} \quad (c)$$

- $S_{x_i} = \left| f_{x_i}(\underline{x}) \right|$ and

  $S_{x_i x_j} = \left| f_{x_i \underline{x}_j}(\underline{x}) \right|$

  which can both be generated efficiently from BDDs. The values of S_{x_i} are shown on the diagonal of the matrix in (b).

Signature Based Variable Ordering

$s_{x_i x_j} =$

$$\begin{bmatrix} \boxed{2} & 2 & 1 & 0 \\ 2 & \boxed{2} & 1 & 2 \\ 1 & 1 & \boxed{2} & 1 \\ 0 & 2 & 1 & \boxed{2} \end{bmatrix} \quad (b)$$

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 2 & 2 & 1 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix} \quad (c)$$

- The values for S_{x_i} are equal for all variables, they cannot be used to order the variables.
- Thus, we need to determine the values for the cross signature $S_{x_i x_j} = \left| f_{x_i \underline{x}_j}(\underline{x}) \right|$

Signature Based Variable Ordering

- We now have a column of n values for each variable.
- Each column characterizes the corresponding variable in it's relationship to the other variables.

Signature Based Variable Ordering

$$s_{x_i x_j} = \begin{bmatrix} 2 & 2 & 1 & 0 \\ 2 & 2 & 1 & 2 \\ 1 & 1 & 2 & 1 \\ 0 & 2 & 1 & 2 \end{bmatrix} \quad (b)$$

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 2 & 2 & 1 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix} \quad (c)$$

- We reorder the values within each column in increasing size and then take each column as a vector and order the vectors according to the relation $\prec$ (c).

Signature Based Variable Ordering

- Conclusions for figure (a), (b), and (c)
  - We have a unique characterization for x_2 and x_3.
  - $f(\underline{x})$ is symmetric with respect to x_1 and x_4.
    - These two variables can be exchanged without affecting the function.
  - Thus, we are not able to distinguish between x_1 and x_4
- In (b), if we hadn't reordered the values within each column, we would have distinguished x_1 and x_4 , incorrectly.

Signature Based Variable Ordering

- In the following figures, we will use the information provided by the auto-signatures on the diagonal to gain better distinguishable variables.
- In (d), x_1 and x_5 are already uniquely distinguished by the auto-signatures.

$$f(\underline{x}) \begin{matrix} \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} x1 & x2 & x3 & x4 & x5 \\ \end{matrix} \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix} \quad (d)$$

Signature Based Variable Ordering

$$s_{x_i x_j} = \begin{array}{c} \begin{matrix} x1 & x2 & x3 & x4 & x5 \end{matrix} \\ \left[\begin{array}{c|c|c|c|c} \boxed{2} & 1 & 2 & 3 & 2 \\ \hline 0 & \boxed{3} & 1 & 2 & 1 \\ 1 & 1 & \boxed{3} & 1 & 1 \\ 2 & 2 & 1 & \boxed{3} & 2 \\ \hline 0 & 0 & 0 & 1 & \boxed{4} \end{array}\right] \end{array} \quad (e)$$

- The ordering of variables in this figure is already consistent with the relation $\prec$ on the auto-signatures.
- This ordering partitions the matrix into sub-matrices as shown in (e).

Signature Based Variable Ordering

- We now reorder the values in each column within a partition according to $\prec$, but not across partition boundaries.
- Finally, we take the values of each column as a vector and apply the relation $\prec$ to the these vectors. In (f), the order is the final order.

$$\begin{array}{c} \begin{matrix} x1 & x2 & x3 & x4 & x5 \end{matrix} \\ \left[\begin{array}{c|c|c|c|c} 2 & 1 & 2 & 3 & 2 \\ \hline 0 & 3 & 1 & 2 & 1 \\ 1 & 1 & 3 & 1 & 1 \\ 2 & 2 & 1 & 3 & 2 \\ \hline 0 & 0 & 0 & 1 & 4 \end{array}\right] \end{array} \quad (f)$$

Boolean Matching [3]

- Boolean Library Matching:
  - The variables of each cell are ordered according to their signatures
  - Build the ROBDD for the function and retain it in the memory
  - During the technology mapping phase, check the library for a subcircuit that is to be mapped using a procedure which is similar to building the library.

Boolean Matching [3]

- During the technology mapping phase:
  - Determine the signatures for the variables of the subcircuit's Boolean function.
  - Based on the signatures, an ordering on the variables is imposed and an ROBDD for the subcircuit's Boolean functions using this variable ordering is built.
  - If such a ROBDD already existed in memory, the subcircuit is implemented using the respective library cell.

Boolean Matching [3]

- In this method, it is no guarantee that the signatures will yield a unique variable ordering, some variables may remain undistinguished.
  - There is always the case for symmetric variables.
  - If not symmetric, we need to examine all variable orderings induced by the possible permutations of these undistinguished variables.
  - Therefore, one have to permute those variables of a subcircuit's Boolean function that are not yet uniquely characterized and build ROBDDs using the permuted variable orderings.

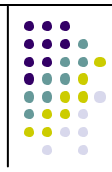
Boolean Matching [3]

- Each such ROBDD is checked against the library until either an isomorphic ROBDD is detected in the library or all possible permutations have been unsuccessfully tried.

Technology Mapping

- Decomposition
- Pattern Matching
  - Common matchers are:
    - Structural matcher
    - Boolean matcher using BDDs
    - PLA matchers
- Covering

PLA Matchers



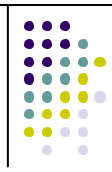
- They are also based on functional representation.
- Similar to Boolean matcher, but different unique representation.
  - PLA tables are used instead of BDDs.
- Has advantages for functions for which BDDs are large.
  - Enables some speedup in removing input variables during the matching.

Conclusion

- A structural matcher requires both the library cells and the subject graph to be composed in the same fashion [5].
- Boolean matchers allow one to find matches based on the logic function, independent from actual local decomposition and under arbitrary input and output phase assignments and input permutations [5].

[5] S.Hassoun and T. Sasao, "*Logic Synthesis and Verification*."

Technology Mapping



- Decomposition
- Pattern Matching
  - Common matchers are:
    - Structural matcher
    - Boolean matcher using BDDs
    - PLA matchers
- Covering

Covering

- This third and final phase consists of identifying the best possible matches for the circuit such that every node in the circuit is covered at least once and the functionality is maintained.

Technology Mapping

Decomposition

Pattern Matching

Covering

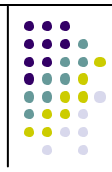
Jeffrey Rosenberg

Covering

- Occurs after decomposition and pattern matching
- Goal is to cover the entire network and optimize some objective function
- The cost function needs to be minimized by finding a complete cover of the subject graph

Hassoun and Sasao, "Logic Synthesis and Verification", p. 121

Directed Acyclic Graph (DAG)



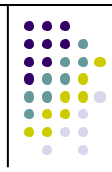
- A Boolean network can be treated as a DAG so that a cover can be searched for directly
- Cells in the library can also be represented as a DAG, or subject graph

Hassoun and Sasao, "Logic Synthesis and Verification", p. 116

DAG Covering

- As a row covering problem
  - Each row represents a node in the subject DAG
  - Each column represents a pattern match
  - Goal is to find a set of columns to cover all rows with minimal cost
  - Doesn't work for large circuits
  - A solution: Tree Covering

Tree Covering



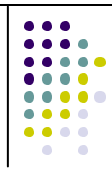
- Subject DAG is separated into trees, and covering is done for each tree
  - Efficient programming algorithms do exist for optimum tree covering, so this makes tree covering more practical for large circuits
- Easiest division is usually at each multiple fanout point

Tree Covering

- Different algorithms are used depending on the goal: What needs to be minimized?
  - Tree covering for Area
    - Minimize circuit size
  - Tree covering for Delay
    - Minimize arrival time
  - Tree covering for Power
    - Minimize circuit power

Hassoun and Sasao, "Logic Synthesis and Verification", p. 122-134

Tree Covering For Area



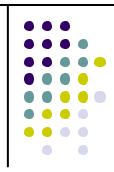
- For a match *m*, the costs of the best matches at the input to *m* are independent of each other
- The best match at a node is independent of the best matches at nodes in its transitive fanout

Hassoun and Sasao, "Logic Synthesis and Verification", p. 122

Tree Covering For Area

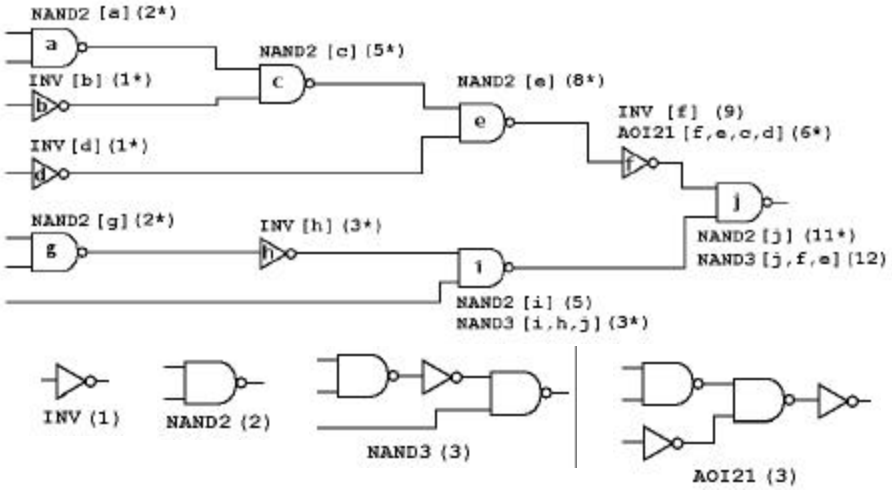
- Goal is to minimize
  $area(m) + S_{v_i \in inputs(m)} min\_area(v_i)$
- v_i are nodes input to m
- $min\_area(v_i)$ is the cost of the minimum cost match at v_i

Tree Covering For Area
Algorithm



```
min_area_tree_cover(v, P) {
/* v is vertex in subject tree, P is set of patterns in library */
   for (w ∈ fanin(v))
         min_area_tree_cover(w, P);
   cost(v) = infinity;
   for (m ∈ match(v, P)) {
         /* match(v, P) the set of library patterns that match at v */
         cost(v, m) = area(m)+S_{vi ∈ inputs(m)} cost(vi);
         if (cost(v, m) < cost(v)) {
           /* best_match(v) stores the best match at v, and cost(v) its area */
           best_match(v) = m;
           cost(v) = cost(v, m);
         }
   }
}
```

Example (Figure 5.4)



Hassoun and Sasao, "Logic Synthesis and Verification", p. 123

Tree Covering For Delay

- Propagation delay of a gate *g* is given by the formula a + b.g
  - a is the intrinsic delay of *g*
    - Not dependent on the load
  - b is the delay per unit load factor
  - g is the load being driven by *g*
    - Dependent on number, type, and size of gates being driven by *g*

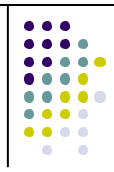
Hassoun and Sasao, "Logic Synthesis and Verification", p. 124

Tree Covering For Delay

- A two pass approach
  - Pass one
    - Leaves to root
    - Find minimum cost match for each possible load value for each node
    - The cost is the delay at the input with the latest arrival time
    - The cost of a match *m* for load g is

$$max_{v_i \in inputs}(m)(a_{i,m} + b_{i,m}g + cost(v_i, g_{i,m}))$$

Tree Covering For Delay

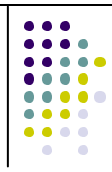


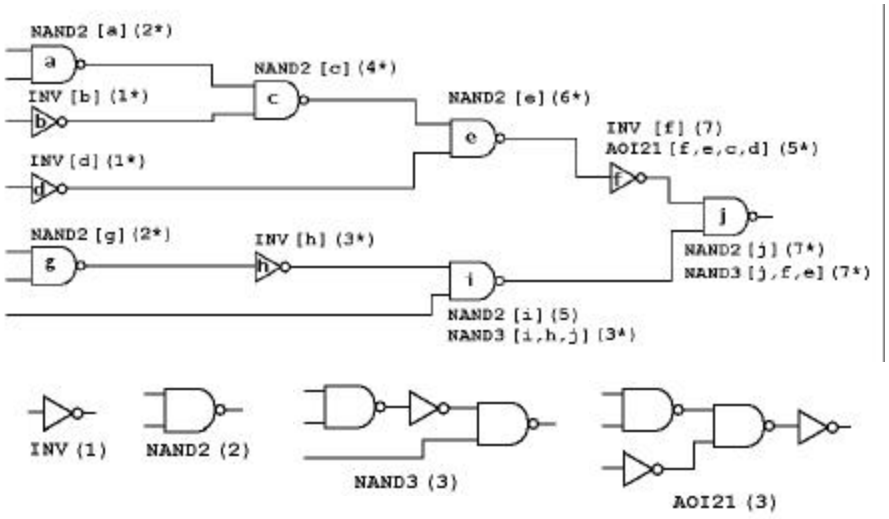
- A two pass approach
  - Pass two
    - Root to leaves
    - For each node visited, the best match is picked based on the matches stored in the first pass

Tree Covering For Delay Algorithm

```
min_delay_tree_cover(v, P) {
/* v is vertex in subject tree, P is set of patterns in the library */
   for (w ∈ fanin(v))
            min_delay_tree_cover(w, P);
   for (g ∈ G(v)) {
   /* G(v) is the set of possible loads at v */
            cost(v, g) = infinity;
   }
   for (m ∈ match(v, P), g ∈ G(v)) {
            cost(v, g, m) = max_{vi ∈ inputs(m)}(a_{i,m} + b_{i,m}g + cost(v_i, g_{i,m}));
            if (cost(v, g, m) < cost(v, g)) {
               /* match(v, g) stores the best match at v for g */
               /* cost(v, g) stores propagation delay up to v */
               match(v, g) = m;
               cost(v, g) = cost(v, g, m);
            }
   }
}
```


Example, Covering For Delay



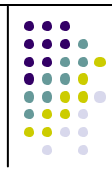


Delay and Area Constraints

- A much more difficult problem
- No known algorithm exists to find an optimal solution - there are drawbacks to every approach
  - Solution quality must be traded-off against runtime and memory requirements

Hassoun and Sasao, "Logic Synthesis and Verification", p. 126

Tree-Bound Covering Method



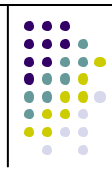
- A subject graph is divided into a set of trees so that the roots of the trees are only nodes with primary outputs or multiple fanouts
- Optimality can't be achieved across tree boundaries, so this makes it difficult when a subject graph is partitioned into a large number of small trees

Xiaoqing and Saluja, "A New Method Towards Achieving Global Optimality in Technology Mapping", p. 9

Tree-Crossing Covering Method

- Provides a solution to the tree-bound covering problem
- Covering starts at a primary output and continues until either a primary input or an already covered node is encountered

Graph Covering



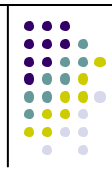
- For this method, a match table is built for the subject graph
  - Each row corresponds to a match
  - Each column corresponds to a node in the subject graph
  - A ‘0’ at row *i* column *j* means that match *i* does not cover node *j*
  - A ‘1’ at row *i* column *j* means that match *i* covers node *j*

Graph Covering

- Various methods
  - Greedy method - select a match that covers the most nodes
  - Select a match that maximizes (nodes covered/cost)
  - Select a match at the node with the fewest matches
- The amount of coverage of these methods is often worse than that of tree covering
  - Simple, inefficient heuristics

Xiaoqing and Saluja, "A New Method Towards Achieving Global Optimality in Technology Mapping", p. 10

DAG Covering



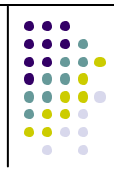
- Tree covering is the simplest way to cover DAGs, but there are disadvantages
  - Inherent deficiency of dealing only with trees
  - Increase in the number of load values
- More generic algorithms are required
- Delay optimal DAG mapping can be enabled by introducing practical load independent models

Hassoun and Sasao, "Logic Synthesis and Verification", p. 126

DAG Covering

- Gain-based delay model
  - Normalize the gain, for example, *gain* = g/C_{in}
    - g is the capacitive load, C_{in} is the gate's input capacitance
  - Gain is not directly dependent on the size of the gate
  - Delay equation $d = l.gain + a$
    - a is the intrinsic delay of the gate
    - l is a constant depending on the logic function

DAG Covering



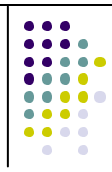
- Gain-based delay model
  - Delay is independent of the load of the cells
  - Dependent solely on the gain

Hassoun and Sasao, "Logic Synthesis and Verification", p. 127

DAG Covering For Delay

- The wavefront
  - Subgraph of the DAG
    - Every path from input to output goes through the subgraph
  - Boundary closer to primary outputs is “head”
  - Boundary closer to primary inputs is “tail”

The Wavefront Algorithm

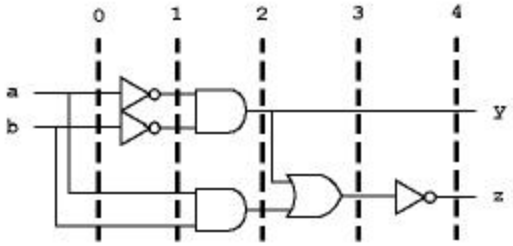


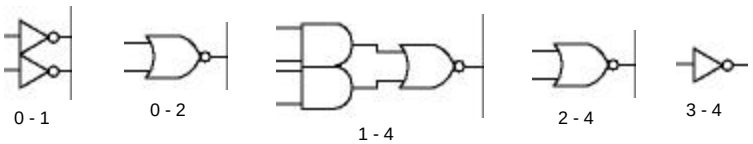
```
wavefront_map(design, wave_width) {
   /* wave_width = width of the wavefront */
   Levelize the design;
   max_levels = maximum number of levels in the design
   head_level = 1;
   while (head_level b max_levels) {
          head_nets = list of all nets on head_level;
          tail_level = head_level - wave_width;
          if (tail_level b 0) tail_level = 0;
          /* Generate matches on the head of the wavefront */
          foreach net, n in head_nets generate_and_implement(design, n, tail_level);
          /* Perform covering on the tail of the wavefront */
          if (tail_level b 0) cover_nets(design, tail_level);
          increment head_level;
   }
```

The Wavefront Algorithm

```
    /* Move the tail, if it has not yet moved */
    if (tail_level b 0) tail_level = 1;
    /* Perform covering on the remaining uncovered levels */
    while (tail_level b max_levels) {
            cover_nets(design, tail_level);
            increment tail_level;
    }
}
```

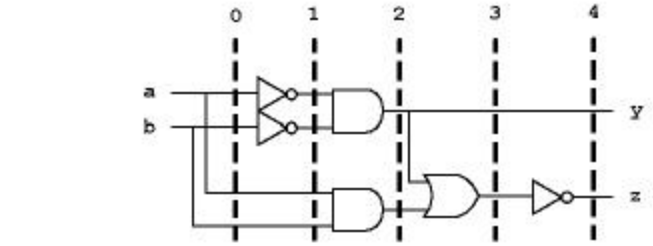

Example (Figure 5.5)

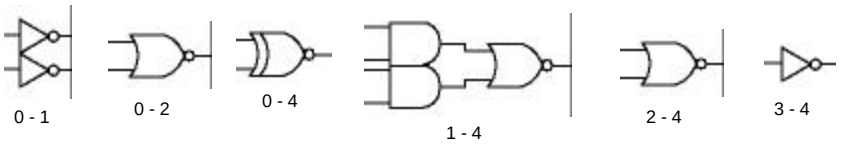




Wave width = 3

Example (Figure 5.5)

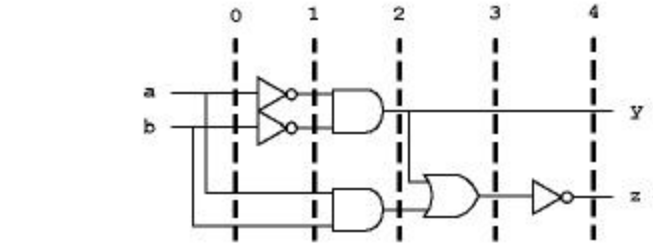


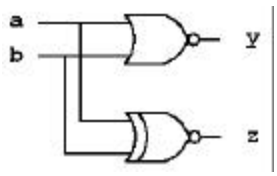


Wave width = 4

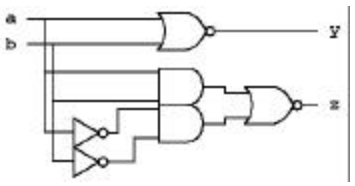
Hassoun and Sasao, "Logic Synthesis and Verification", p. 130

Example (Figure 5.5)





Wave width = 4



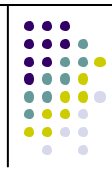
Wave width = 3

Hassoun and Sasao, "Logic Synthesis and Verification", p. 130

The Wavefront Algorithm

- Can handle more complicated cost functions
  - Mapping for different fall/rise times
    - The fastest match for both falling and rising delays are maintained
    - Separate falling and rising times are propagated through different paths in the graph
    - The delay can be minimized by a second pass from outputs to inputs

The Wavefront Algorithm



- Can handle more complicated cost functions
  - Optimizing for area
    - A two pass approach
    - First pass: Keep more than one match (fastest match, smallest match, etc.)
    - Second pass: Choosing slower patterns in non-critical regions will minimize area

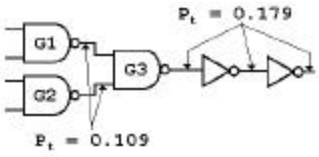
The Wavefront Algorithm

- Can handle more complicated cost functions
  - Dynamic decomposition
    - Primitives with many inputs can be preserved until the head of the wavefront gets there
    - Input arrival times are computed through patterns of already mapped logic
    - Number of logic levels between late arriving signals and output of original primitive is minimized by decomposing large primitives into NAND gates

Mapping For Minimum Power

- Goal is to minimize $S_{g \in gates}\ C_o(g)*p_t(g)$
  - $C_o(g)$ is the output capacitance of the final mapped circuit
  - $p_t(g)$ is the average transition probability

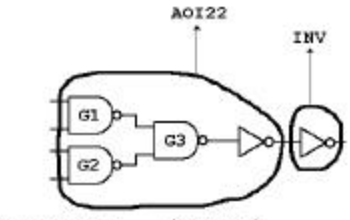
Example (Figure 5.6)



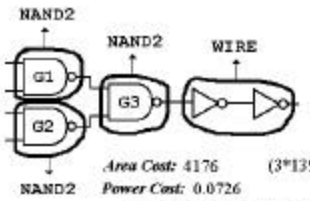
Gate Type	Area	Intrinsic Cap.	Input Load Cap.
INV	928	0.1029	0.0514
NAND2	1392	0.1421	0.0747
NAND3	1856	0.1769	0.0869
AOI22	2320	0.3410	0.1033

Table of gate information

P_t: average transition probability



Area Cost: 3248 (2320 + 928)
Power Cost: 0.0887 [0.179*(0.3410 + 0.0514 + 0.1029)]



Area Cost: 4176 (3*1392)
Power Cost: 0.0726
[0.109*2*(0.1421 + 0.0747) + 0.179*0.1421]

Hassoun and Sasao, "Logic Synthesis and Verification", p. 133

Mapping For Minimum Power

- In this case, the cost is dependent on the transition probability
- If $p_t(g)$ is the same at every node, then mapping for power is identical to mapping for area

Hassoun and Sasao, "Logic Synthesis and Verification", p. 132

Tree Mapping For Power

- Given a match *m*, the power cost of the best matches at the inputs of *m* are independent of each other
  - Similar to the property for covering for area/delay
- $cost(v, m) = power\_cost(m) + \sum_{v_i \in inputs(m)} min\_power(v_i)$
- Goal is to minimize $cost(v, m)$

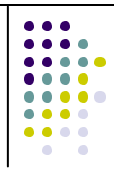
Tree Mapping For Power

- $power\_cost(m)$ is given by $p_t^*(C_{int} + C_o)$
  - C_{int} is intrinsic capacitance
  - C_o is load capacitance
- p_t is the same for all matches, so $p_t^*C_o$ is the same for all matches at a node
  - C_o is not needed to determine the best match at a single node

Tree Mapping For Power

- The actual cost of the best match at node v_i, *min_power*(v_i), is needed to determine the best match at the fanout of node v_i
- This is given by *cost*(v_i) + $g_{i,m}$**p*$_t$(v_i)
  - $g_{i,m}$ is the load on v_i by its fanout node

Tree Mapping For Power With Delay Constraints



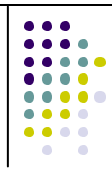
- Cost of a match consists of both the power cost and the delay time
  - The match with lowest power is chosen only if the arrival time is not greater than the requirement
  - If the arrival time is greater than the requirement, the match with the earliest arrival time is selected
- The required time may not be met, but the solution will be the fastest possible one

Reliability

- A network that is power optimized is not necessarily optimized for reliability
  - Power minimization: Average drain current for each gate is minimized
  - Hot-carrier reliability enhancement: Targets only gates with the most serious hot-carrier effect

Hassoun and Sasao, "Logic Synthesis and Verification", p. 135

Conclusions



- Decomposition
  - Restructures the Boolean function into the subject graph
- Pattern matching
  - Finds matches between patterns and regions of cells in the subject graph
- Covering
  - Uses patterns to cover the subject graph and minimize the cost function

Synopsis

- The goal of Technology Mapping is to transform a technology-independent description of a logic circuit into a technology-specific representation
- The technology-dependent implementation should optimize some cost metric (delay, area or power dissipation).