(*Lec 10*) Technology Mapping

- What you know: Synthesis--*all* there is
  - 2-level ESPRESSO style
  - Multi-level style
    - Structural modifications to Boolean Logic Network
    - Subexpression extraction: cube & kernel extraction
    - Vertex simplification via multi-level don't cares
  - Clever representation schemes to make these doable
    - PCN inside ESPRESSO
    - BDDs everywhere else

- What you don't know
  - How a synthesized multi-level logic network gets turned into real, live, *usable* gates in your implementation

Copyright Notice

© Rob A. Rutenbar 2001
All rights reserved.

You may not make copies of this material in any form without my express permission.

Where Are We?

- *After* logic synthesis--how to map to *real* library of gates?

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
Midsem	15	16	17	18	19	8
break	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

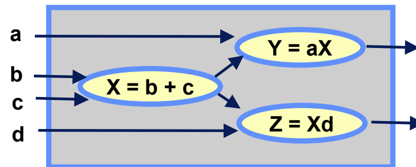
Introduction
Advanced Boolean algebra
JAVA Review
Formal verification
2-Level logic synthesis
Multi-level logic synthesis
Technology mapping
Placement
Routing
Static timing analysis
Electrical timing analysis
Geometric data structs & apps

Readings

- De Micheli
  - Chapter 10 is all about is all about technology mapping, which he calls "cell library binding"
  - Read 10.1, 10.2, 10.3 (but only 10.3.1 here) and also 10.6

Tech Mapping: The Problem

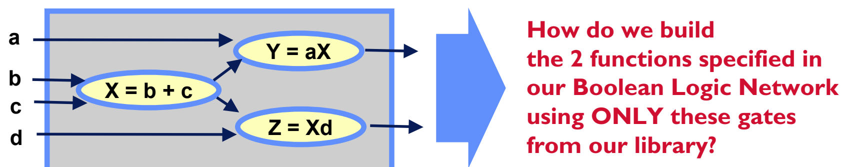
- Multi-level model is still a little abstract
  - Structure of the Boolean logic network is fixed
  - ESPRESSO-style 2-level simplification done on each node of network...
  - But that still doesn't say what the *actual* gate-level netlist should be
- Trivial example



Tech Mapping: Problem

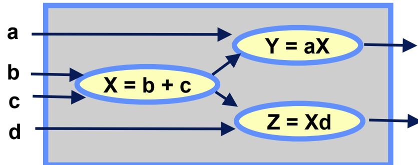
- Suppose we have these gates in our "library"
  - This is referred to as the "technology" we are allowed to use to actually build this optimized network

  - OA21 is in OR-AND, a so-called complex gate in our library



Tech Mapping: Simple Example

AND2 OR2 OA21



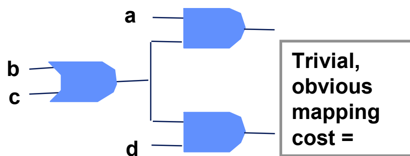
Trivial, obvious mapping

Alternative un-obvious mapping

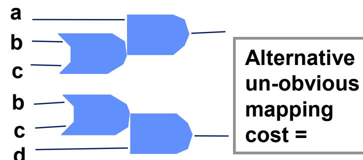
Tech Mapping: Simple Example

- Why choose a non-obvious mapping?
  - Answer 1: *Cost*. Suppose each gate in lib has a cost associated with it. Think of this as, say, the silicon area of the gate

AND2 cost = 4 OR2 cost=4 OA21 cost=5



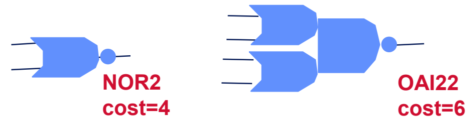
Trivial, obvious mapping cost =



Alternative un-obvious mapping cost =

Tech Mapping

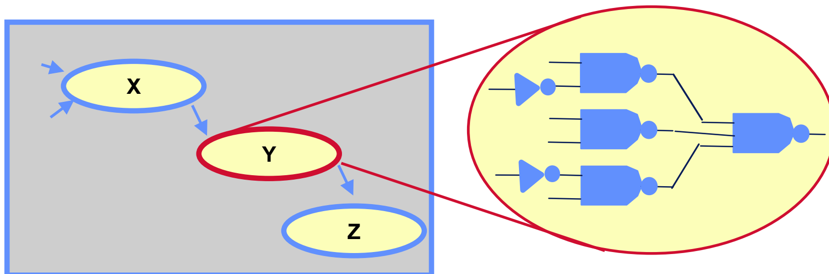
- Why choose a non-obvious mapping?
  - **Answer 2: obvious parts not in your library**
  - **Example: your library is only NOR and OR-AND-INVERT gates**



  - **Even if you *wanted* to think about mapping your network into nice, vanilla ANDS and ORs and NOTs, *you can't, because they are not in your library*.**
  - **Examples: in some Gallium Arsenide (GaAs) IC technologies, you *only* get NOR gates, period. In some dynamic CMOS logic styles, NORs are also the only thing you can do easily.**

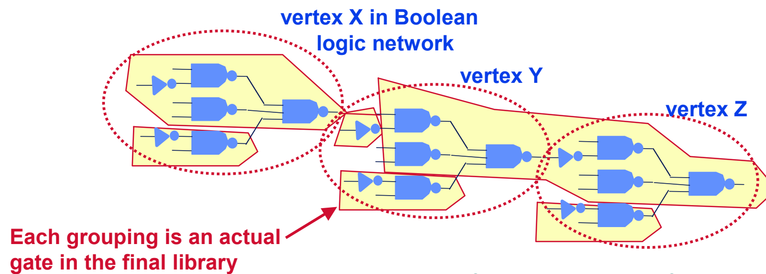
Tech Mapping: What Multilevel Synthesis Does

- Helpful model to use: Multi-level synthesis does *this*...
  - **Structures the multiple-vertex Boolean logic network "well"**
  - **Minimizes guts of each vertex in the network "well", ie, min literals**
  - **But this is not real logic gates. This is "uncommitted" logic, or "technology independent" logic**
  - **Think of it like this: it's only NANDs and NOTs, nothing else**



What "Technology Mapping" Does

- Model
  - So, synthesis gives you a network with the right *overall* structure...
  - ...but not in terms of the gates you actually have to *implement* it with
- Tech mapping
  - "Maps" output of synthesis, in "technology independent" form, into your actual gate library (Also called "binding" to the technology library)
  - Important point: tech mapper is no longer required to respect the boundaries of vertices in the original Boolean logic network



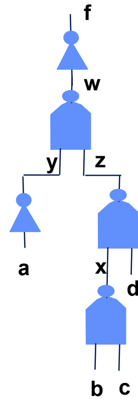
Each grouping is an actual gate in the final library

Technology Mapping as Tree Covering

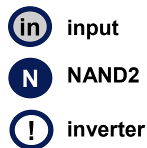
- One particularly useful, simple model of problem
  - Your logic network to be mapped is a **tree** of simple gates
    - Easiest to assume absolutely minimal gate types
    - Example: tech-independent form is 2 input NAND and NOT
  - Your library of actual gate types is also available in this form
    - Each gate can be represented as a tree of NAND2 and NOT
    - Each gate also has an associated cost
- Problem
  - "Cover" the tree that represents your tech-independent logic--called the **subject tree**...
  - ...with minimum cost set of gates (each a **pattern tree**) from lib
  - Reduces tech mapping to: **matching problem + a minimization problem**

Tree Covering Example

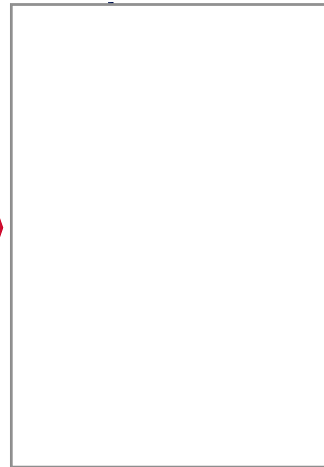
- Here is your *subject* tree to be matched
  - Assume it pops out of synthesis as only NAND2 and NOT



We get tired of drawing the gates all the time, so adopt a simpler set of symbols:

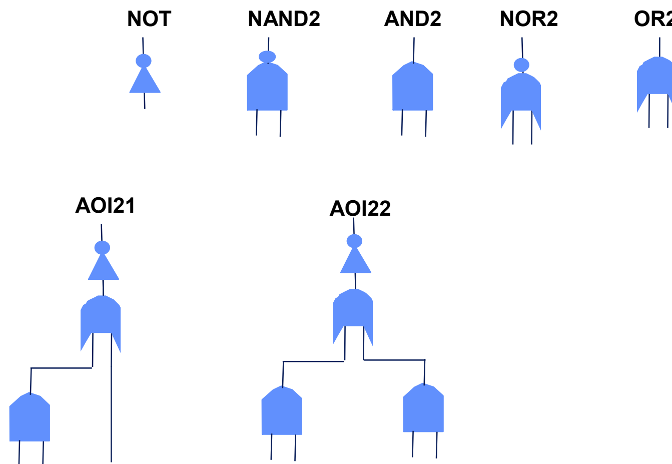


input
NAND2
inverter



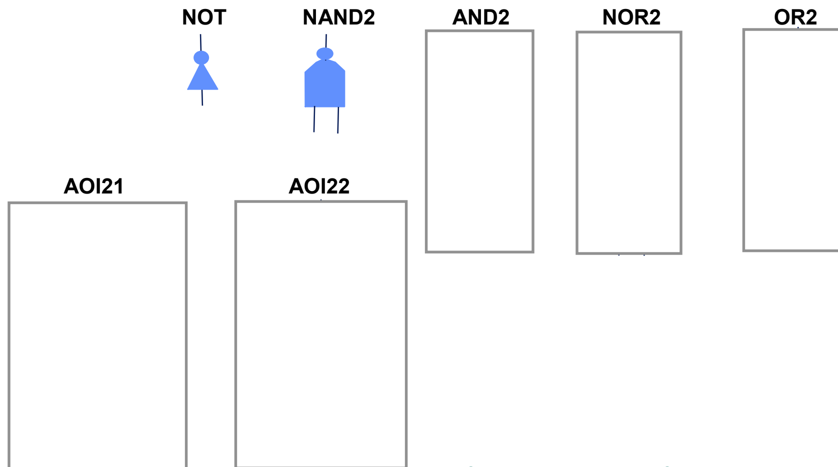
Tree Covering: Your Technology Library

- And, here is your library--at least, in *ideal* form



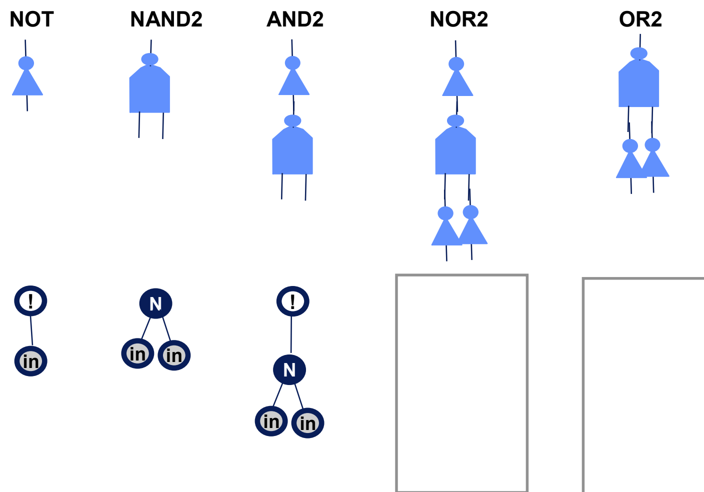
Tree Covering: Representing Your Libnrary

- Oops!
  - To use tree covering, must represent your library using *same gates* as your synthesis tool -- here, NAND2 + NOT



Tree Covering: Pattern Trees

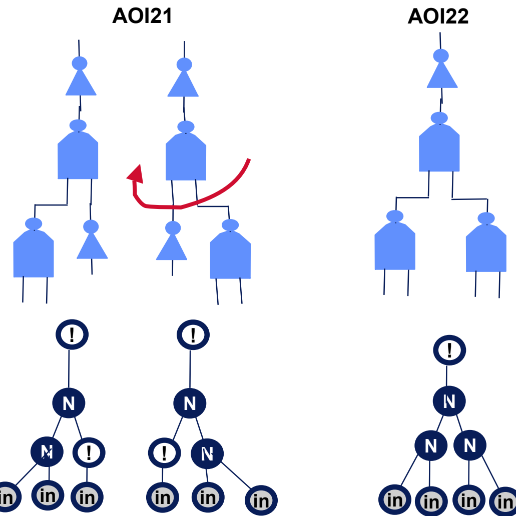
- Represent each gate as a *pattern* tree, using notation



Tree Covering: Pattern Trees

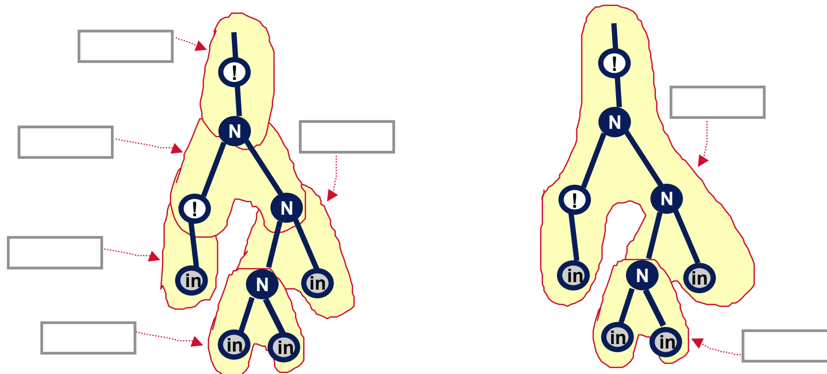
- Symmetries matter!

2 pattern trees for AOI21 since there are 2 ways we could match this gate against a target tree



Tree Covering Example

- Notice in this form it's a very *clean* covering problem
  - Every "blob" is one gate in our lib, matching somewhere on subject tree
  - Note that a grey "in" input node allowed to match *anything* in tree



Not very clever tree cover = 3 NAND2s + 2 NOTs

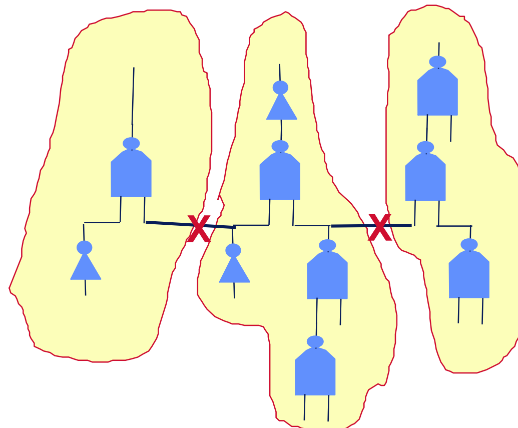
Clever tree cover = 1 AOI21 + 1 NAND2

TechMap via Tree Covering

- What do we need?
- Tree-ifying the input netlist
  - Few assumptions need mentioning
- Tree matching
  - For every node in your subject tree, need to know all the target pattern trees in your library that can match here
- Minimum cost covering
  - Given you know what can match at each node of subject tree, which ones do you pick for a minimum cost cover?

Tree-ifying the Netlist

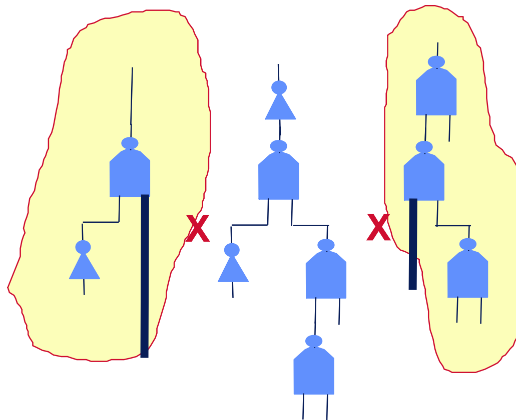
- These algorithms only work on *trees*, not DAGs



Must split this DAG with fanouts into 3 separate trees, map each separately
This entails some clear loss of optimality, since cannot map *across* trees

Tree-ifying Netlist

- Our algorithms mandate trees, not DAGs
  - Every place there is fanout from gate output > 1, you have to cut
  - Clearly loses some optimizations
  - There are ways around this, but we won't look at them here...



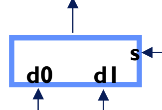
Aside: How Restrictive is "Tree" Assumption?

- Subject graph and each pattern graph must be *trees*
- Subject tree: must *tree-ify* it
- What about *pattern* trees?
  - Are there any well-behaved 'gates' you would like in lib, but != trees?
  - Unfortunately, yeah...

EXOR gate

Boolean eqn:

2:1 MUX



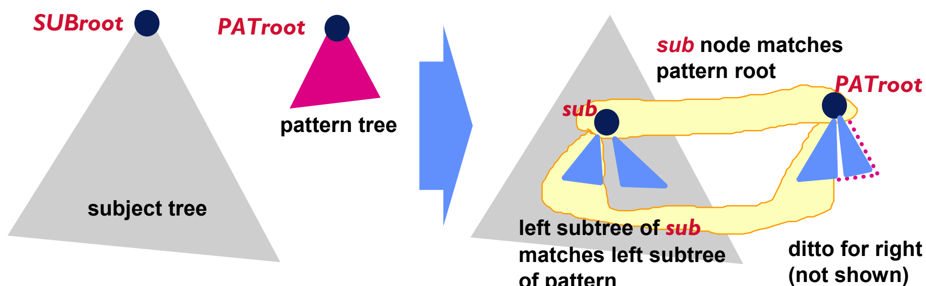
Boolean eqn:

Tree Matching

- There are several approaches
  - Elegant, complicated approach: *FSM matching*
    - Treat subject tree like a special "string" of characters
    - Turn the pattern library into a Finite State Machine (FSM)
    - Run the subject "string" thru the FSM, it tells you each place in subject tree that ANY tree in library matches
    - Fast, cool, hard to describe quickly (see book)
  - Straightforward, not-so-fast, easy approach: *Recursive matching*
    - Inputs: subject tree (root), a specific target pattern tree (root)
    - Outputs: each node in subject tree marked if pattern matches
    - Note: need to run this algorithm for every target tree in library

Recursive Tree Matching

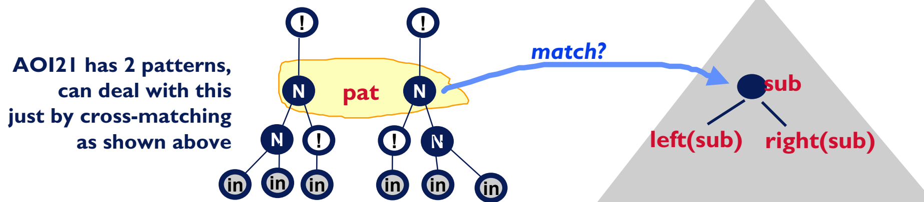
- Ideas
  - Start with subject tree root *SUBroot*, pattern tree root *PATroot*
  - Do a tree walk on subject tree, visit *every* node--*sub*--of subject tree
  - At *each* node of sub in walk, you call MATCH(*sub, PATroot*)
  - The pattern tree matches here at *sub* if
    - *sub* node type same as *PATroot* node type, AND
    - ...each child of *sub* node recursively matches approp. child of *PATroot*



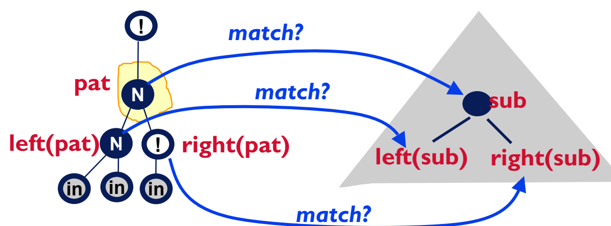
Tree Matching

- Well, it's slightly messier than that
  - Inverters have only *one* child
  - Node types
    - NAND matches NAND
    - NOT matches NOT
    - INPUT matches *anything*
  - To handle asymmetric targets, must try BOTH these child matchings:
    - match (left(*sub*), left(*pat*)) && match (right(*sub*), right (*pat*))
    - match (left(*sub*), right (*pat*)) && match (right(*sub*), left (*pat*))

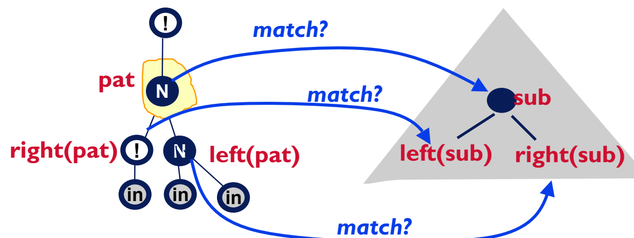
AOI21 has 2 patterns, can deal with this just by cross-matching as shown above



Asymmetry: In More Gruesome Detail



OR…



Tree Matching

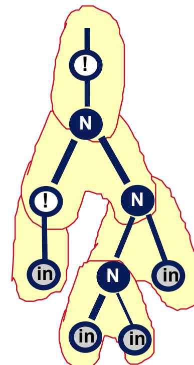
Algorithm outline

```
MATCH( sub, pat) {
    if ( nodetype( pat ) == INPUT )
    then return (true);    // input matches anything

    if ( sub is a leaf of subject graph)
    then return (false);   // cannot be a match

    if ( nodetype(pat) !=  nodetype(sub) )
    then return (false) ;  // cannot be a match

    if ( nodetype(pat) == NOT ) {     // only 1 child to recurse on
    then return(  MATCH( child(sub), child(pat) );

    // it must be NAND2, so 2 children to recurse on
    //  just do the case where we assume pattern is asymmetric
    return( MATCH( left(sub), left(pat)) && MATCH(right(sub),right(pat))
              || MATCH( left(sub), right(pat)) && MATCH(right(sub),left(pat))  );

}
```


Minimum Cost Tree Covering

Where are we?

- We tree-ified subject tree
- For each pattern tree in our library...
- ...we walked the nodes of the subject tree, root to leaves, recursively
- ...and at each node visited, we asked: MATCH(sub node, pat node)?
- Result: *each node of subject now labeled with which pattern trees match it*



One candidate cover of a subject tree with 5 patterns from target library

Next problem:

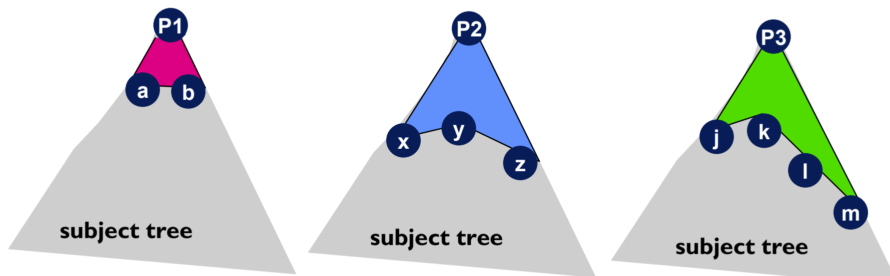
- What is the *best cover* of the subject tree with patterns from target lib?

Minimum Cost Covering of Subject Tree

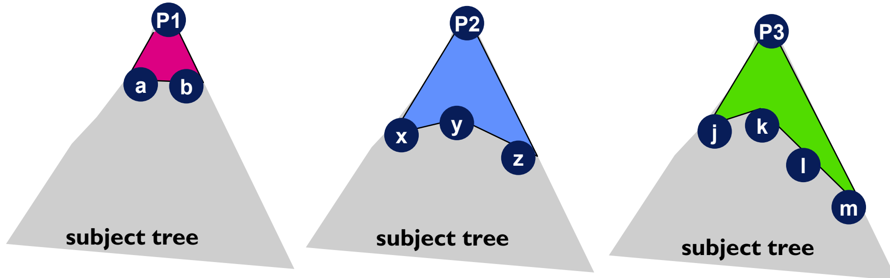
- **Key insight**
  - If pattern P is min cost match at some node of subject tree...
  - ...then it must be that each *leaf* of pattern tree is also the *root* of some min cost matching pattern
  - Leads to a recursive algorithm
  - (A *dynamic programming* algorithm, if you know that terminology...)
- **Pick an illustrative example to see how this works**

Min Cost Tree Covering

- **Assume 3 different patterns match at *root* of subject**
  - Pattern P1 has 2 leaf nodes: a b
  - Pattern P2 has 3 leaf nodes: x y z
  - Pattern P3 has 4 leaf nodes: j k l m
  - Which is *cheapest* pattern if we know cost of each pattern?



Min Cost Tree Cover



- Cheapest cover of root of subject is mincost(*root*) =

  min (patterncost(P1) + [ ] + [ ] ,

  patterncost(P2) + [ ] + [ ] + [ ] ,

  patterncost(P3) + [ ] + [ ] + [ ] + [ ]

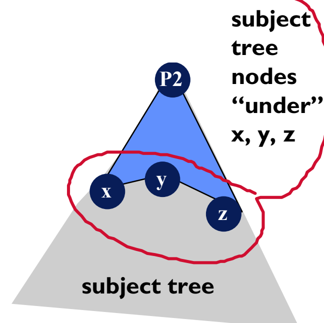
)

- Each box above means we must recurse on mincost(node)

Min Cost Tree Cover

- Naive Algorithm

```
mincost( treenode) {
   cost = ∞
   foreach( pattern P matching at subject treenode) {

      let L = {nodes in subject tree corresponding to leaf nodes in P
               when P is placed with its root at treenode }

      newcost = patterncost(P)
      foreach( node n in L ) {
         newcost = newcost +  mincost( n );
      }
      if ( newcost < cost )
      then  {
         cost = newcost;
         treenode.selected = P;
   }
```

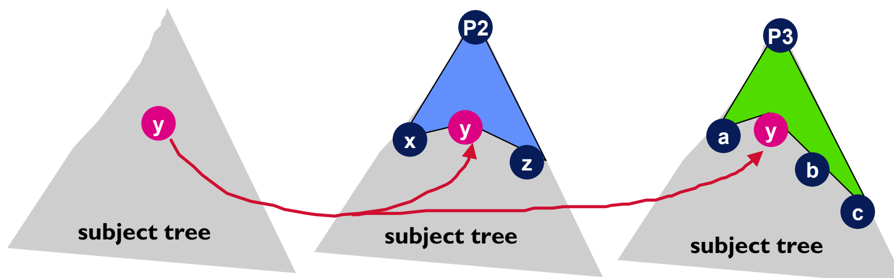


Min Cost Tree Cover

- What's wrong with this?
  - Will revisit *same* treenode many times during recursions...
  - ...and it will recompute the min cost cover for that node each time.
- Can we do *better*...?
  - Sure, just keep a table with min cost for each node
  - Starts with value ∞ and then when the node's cost get computed, this value gets updated
  - Check first to see if the node has already been visited before computing it--saves computing it multiple times.

Illustration

- Node "y" in this subject tree
  - Will get its mincost cover computed (mincost(y)) when we put P2 at the root of the subject tree...
  - ...and again when we put P3 at the root
  - Why not just compute it *once*, first time, *save* it, and look it up later?



Min Cost Tree Cover

- **Improved**
  - Assume table[] gets set up so table[node] = ∞ for all nodes at start

```
mincost( treenode) {
   if ( table[treenode]  ) < ∞
   then return( table[treenode]);

   cost = ∞
   foreach( pattern P matching at subject treenode) {

      let L = {nodes in subject tree corresponding to leaf nodes in P
                 when P is placed with its root at treenode }

      newcost = patterncost(P)
      foreach( node n in L ) {
         newcost = newcost +  mincost( n );
      }
      if ( newcost < cost )
      then  {
         cost = newcost;   table[treenode] = newcost
         treenode.selected = P;
   }
```


Min Cost Tree Cover Example

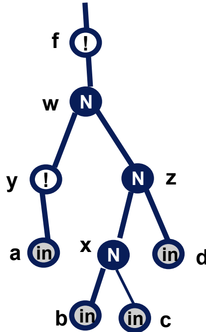
Subject tree

At treenode	Can Match	With min cost
f		
w		

All our pattern trees

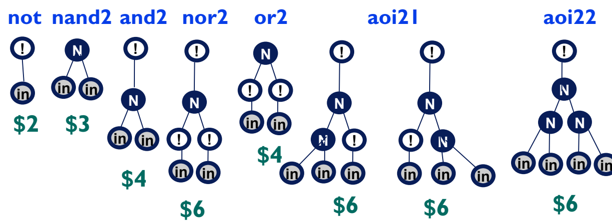
Min Cost Tree Cover Example, cont

Subject tree



At treenode	Can Match	With min cost
y		
z		
x		

All our pattern trees



Min Cost Tree Cover

- Example
  - If costs for NOT, NAND2, AND2, AOI21 are as shown…
  - Best cover is to use one AOI21 and one NAND2
- Turns out to be several nice extensions possible
  - Can tweak algorithm a little to minimize delay instead of cost
  - Need to deal with some messy issues associated with capacitive loads for driven gates, but some simple discrete loads can be modeled and handled.

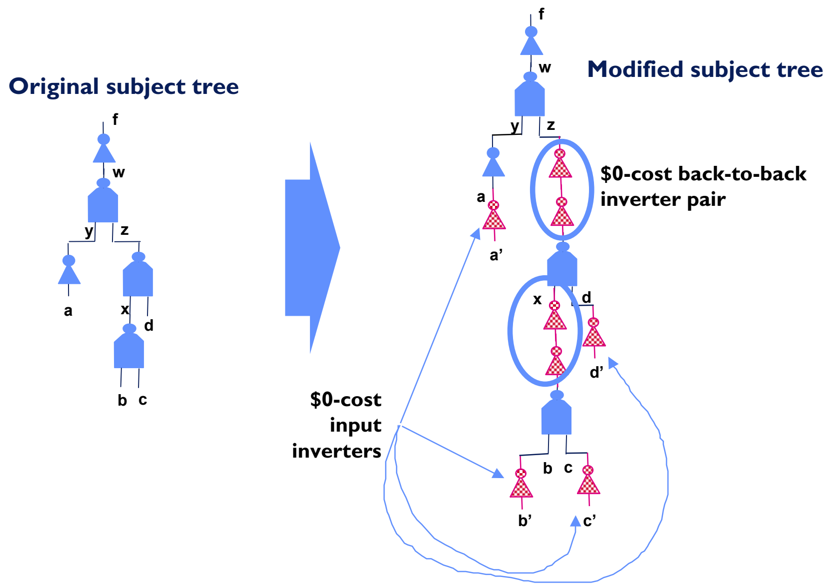
Issues

- Pro
  - Easy, simple algorithm
  - Works great for trees
- Con
  - Not everything is a tree
  - Most subject netlists are NOT trees, need to chop up into trees
  - Some patterns cannot be trees (EXOR, MUX)
- Comments
  - Heuristic tricks for dealing with most of these
  - Also, other tech mapping approaches

Polarity Assignment

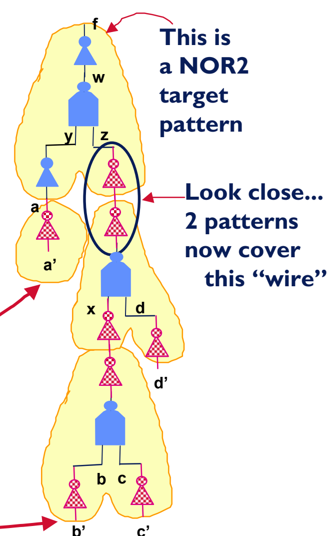
- One cool trick worth mentioning
  - What if you can't match a pattern just because you don't have the right polarities (true / complemented form) on internal nodes of subject tree?
  - Just don't use that good pattern? No -- *fix the polarity*
- *Do this to netlist*
  - At every *internal* wire in netlist (from a gate output to a gate input) where there is *no* inverter, replace with *back-to-back inverters*
  - At every input, add one more *zero-cost-inverter*
- *Do this to pattern library*
  - Add *back-to-back inverter* pattern with cost = 0, on every *internal* wire (from a gate output to a gate input) in your pattern trees
  - Add *zero-cost-inverter* pattern, only matches at inputs, cost=0

Inverter Trick



Inverter Trick

- **Why do this??**
  - Can get different covers!
  - Notice here, if you had to create a', b', c', d', or if you already have each of a,b,c,d in both forms....
  - ...you can get a nice cover made up of 3 NOR2s and one extra inverter
  - Of course, this inverter is really bogus -- can really get away with nuking it and just inputting 'a' to the NOR2...
  - ...but do need different polarity now on these inputs



Summary

- **Technology mapping ...**
  - Synthesis gives you "uncommitted" or "technology independent" design, eg, NAND2 and NOT
  - Mapping turns this into real gates in your own library
  - Can determine difference between good implementation and a bad one
  - Still a very hot problem
- **Tree covering**
  - One nice, simple, elegant approach to the problem
  - 3 parts: **tree-ify** input, **match** all lib patterns, find min cost **cover**
- **Tricks**
  - Adding extra inverters (with cost) can get you out of nasty problems where a nice pattern does't match because of wrong polarity at inputs

© R. Rutenbar 2001, CMU 18-760, Fall 2001 43