EE4218 L7 - Technology Mapping

1 General introduction

Recall that after *Architectural Synthesis*, we had *Logic Synthesis*.

Logic synthesis consisted of the following sub-steps . . .

1. Logic Optimization
   1.1. Minimizing undesirable redundancies; Using K-Maps, boolean identities, etc...
2. Technology Mapping (ie. Library Binding)
   2.1. Mapping logic to library cells
   2.2. The 'binding' that is going on here is of a **microscopic** level, in contrast to the **macroscopic** 'binding' that happened during architectural synthesis

We will focus on Technology Mapping now.

2 Introduction to Technology Mapping

Maps the technology **independent** network resulting from logic optimization, to logic cells that are available in a technology **dependent** cell library

Whilst performing technology mapping, we utilize algorithms to furthur minimize area while still meeting constraints (eg. timing constraints).

2.1 Types of technology mapping

Broadly speaking, it depends on whether we are developig for ASICs or FPGAs.

- ASICs
  - Usually library cells are more primitive than that of FPGAs
  - Limited set of pre-designed cells
- FPGAs
  - LUT-based (Xilinx): Each FPGA cell is termed a *Configurable Logic Block*
  - MUX-based (Actel): Each FPGA cell consists of a number of multiplexers

2.1.1 (ASIC) cell library

As an example, the IBM standard-cell library used for the POWER4 is as follows

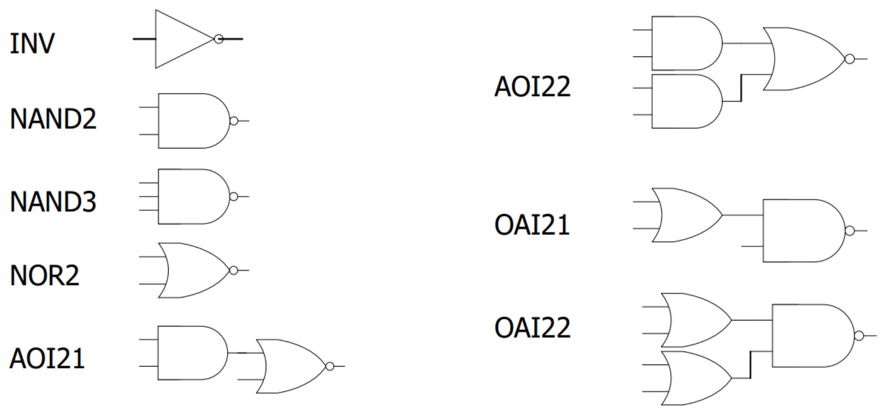
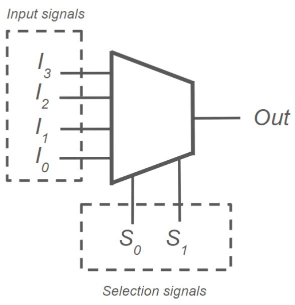


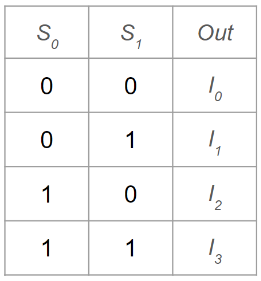
Figure 1: Designer has the following cells to build upon

2.1.2 (FPGA) LUT-based cells

2.1.2.1 Multiplexer

Let's begin with a *4-1* MUX.





S_0	S_1	Out
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Figure 2: *4-1* MUX, with associated truth table

- In a *4-1* MUX, we have 2 *selection* signals which select amongst 2^2 *input* signals.
- Note that **each** of the $\{I_0, I_1, I_2, I_3\}$ signals could be 0 or 1, therefore this *4-1* MUX has $2^{2^{num\_input\_signals}} = 16$ possible **input-output** combinations.
- Recognize that a *logic function* is a *mapping* from input to output (singular in this case), hence we could represent 16 different combinational functions this way.
- Herein is the underlying logic behind an LUT - if we can configure the value of these input signals, then we can construct 16 different combinational functions.

2.1.2.2 Constructing an LUT

Using the ideas above, the *4-1* MUX above can thus be converted into a 2LUT (2-input LUT)

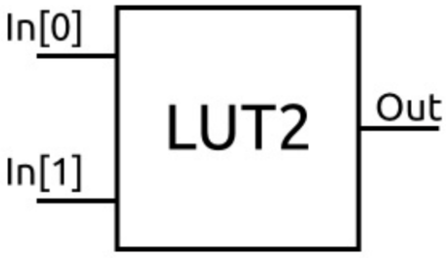


Figure 3: 2LUT
Selection signals are our *input addresses*

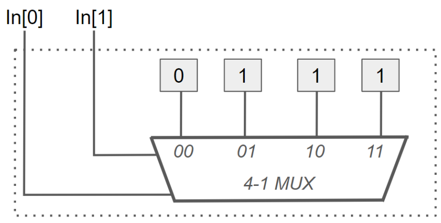


Figure 4: **0110** represents the **output** of a particular truth table (ie. combinational function) that this 2LUT implements

- The LUT consists of a block of SRAM that is indexed by the LUT's inputs
- Output of the LUT is whatever value is in the indexed location in it's SRAM

Let's examine the internals of the LUT.

- LUT is built out of ...
  1. SRAM bits to hold some configurable memory (*CRAM*), we term this the *LUT-mask*
  2. Set of multiplexers to select the bit of CRAM that will be directed to output
- To implement a k-LUT (an LUT that can implement any k-input function), we require ...
  1. 2^k SRAM bits. Note that there are 2^{2^k} possible combinational functions, but we are setting the SRAM bits (ie. outputs) only for a **certain** combinational function we want to implement. Therefore, since the combinational function has k inputs, it should have 2^k possible outputs.
  2. $2^k : 1$ MUX, to select the output for our **certain** combinational function

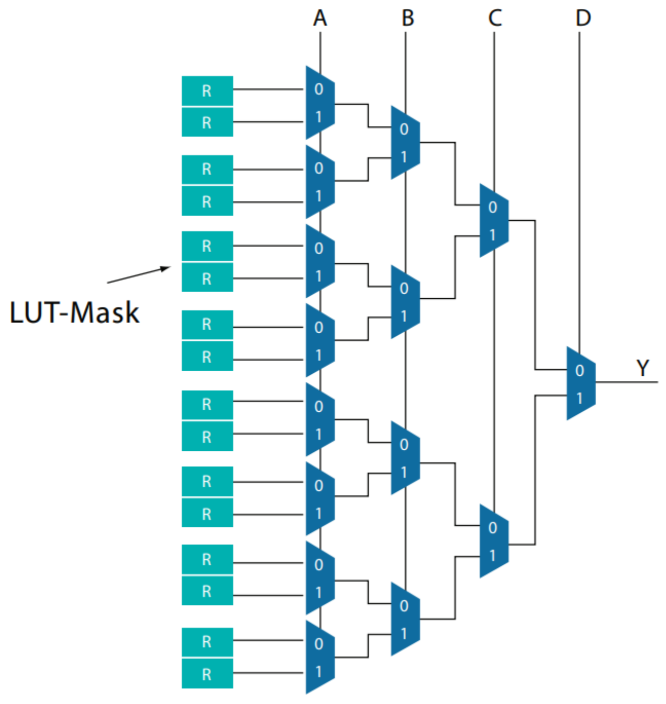


Figure 5: 4LUT

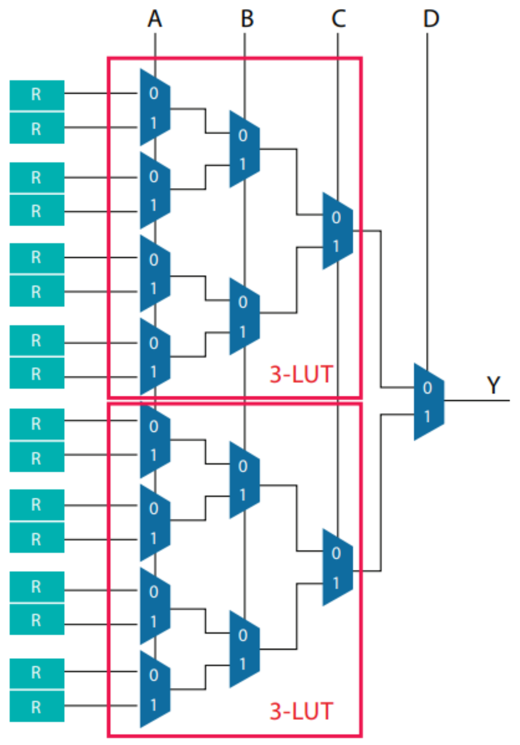


Figure 6: 4LUT implemented with two 3LUT and a 2:1 MUX

2.1.2.3 CLB

Finally, we can understand how the CLB is built.

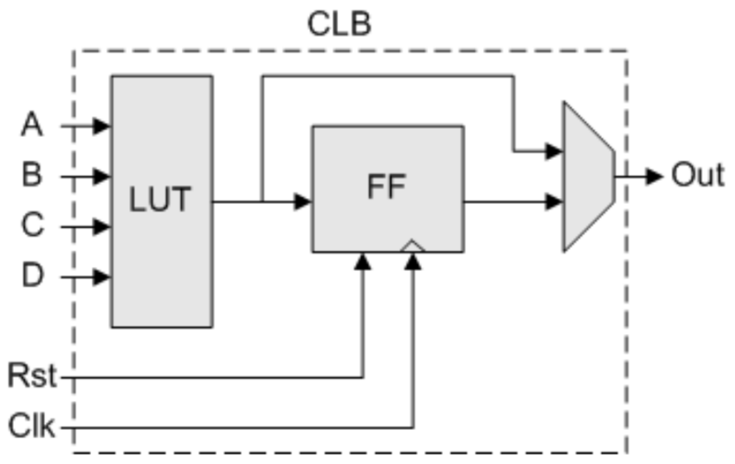


Figure 7: Configurable Logic Block

2.1.3 (FPGA) MUX-based cells

2.1.3.1 Tweaking the Multiplexer

Remember the *4-1* MUX earlier?

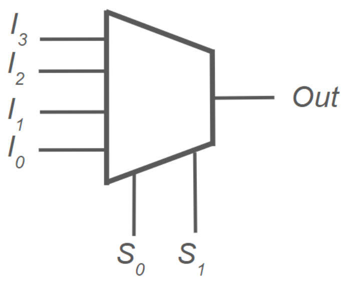


Figure 8: *4-1* MUX

Previously, the *4-1* MUX could only implement a 2-input combinational function (via $\{S_0, S_1\}$), since $\{I_0, I_1, I_2, I_3\}$ were 'fixed' at binary values 0 or 1. However, what if we allowed $\{I_0, I_1, I_2, I_3\}$ to act as inputs as well? Then the *4-1* MUX can implement up to a 6-input combinational function!

Hence, we can furthur expand on the truth table of the MUX.

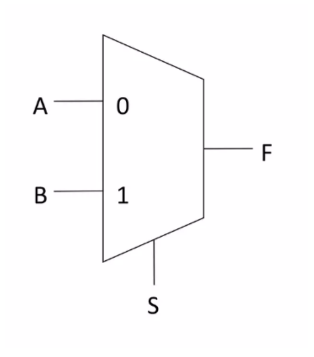
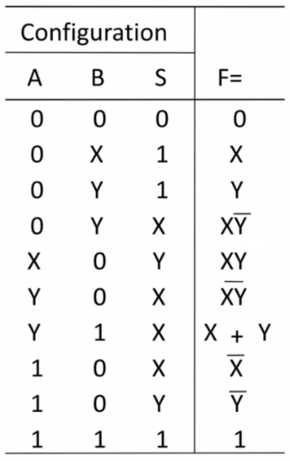


Figure 9: 2-1 MUX



Configuration			
A	B	S	F=
0	0	0	0
0	X	1	X
0	Y	1	Y
0	Y	X	$X\overline{Y}$
X	0	Y	XY
Y	0	X	$\overline{X}Y$
Y	1	X	X + Y
1	0	X	$\overline{X}$
1	0	Y	$\overline{Y}$
1	1	1	1

Figure 10: Expanded truth table.
Note that it is non-exhaustive, since we expect there to be 3^3 possible combinations

This new way of viewing MUXes will make sense soon.

2.1.3.2 Shannon's decomposition theorem

Allows for any logic function to be expanded and broken down into parts. This means that it is possible to implement any logical function using multiplexers.

$$\begin{aligned} f(x_1, x_2, \ldots, x_n) &= x_1 \cdot f(1, x_2, \ldots, x_n) + \overline{x_1} \cdot f(0, x_2, \ldots, x_n) \\ &= \big(x_1 + f(0, x_2, \ldots, x_n)\big) \cdot \big(\overline{x_1} + f(1, x_2, \ldots, x_n)\big) \\ &= x_1 \cdot f(1, x_2, \ldots, x_n) \oplus \overline{x_1} \cdot f(0, x_2, \ldots, x_n) \end{aligned}$$

Figure 11: Shannon's decomposition theorem

Let's apply Shannon's decomposition theorem now.

1. Suppose we have a function $F(a, b, c)$
2. Expanding it using SDT, we get $F = a \cdot F_a + \bar{a} \cdot F_{\bar{a}}$
3. Therefore, F can be implemented as a 2-1 MUX as seen below.

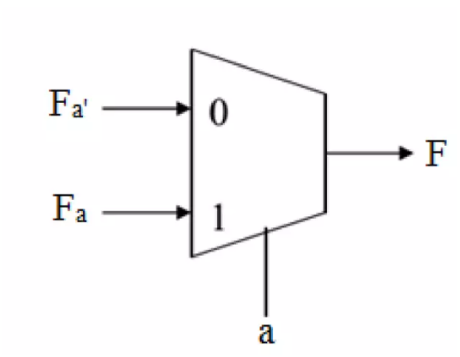


Figure 12: When $a = 1, F_a$ is channeled to output
When $a = 0, F_{\bar{a}}$ is channeled to output

We can recursively apply SDT ...

1. $F = a \cdot F_a + \bar{a} \cdot F_{\bar{a}}$

1.1. $F_a = b \cdot F_{ab} + \bar{b} \cdot F_{a\bar{b}}$

1.2. $F_{\bar{a}} = b \cdot F_{\bar{a}b} + \bar{b} \cdot F_{\bar{a}\bar{b}}$

2. $F = \{a \cdot (b \cdot F_{ab} + \bar{b} \cdot F_{a\bar{b}})\} + \{\bar{a} \cdot (b \cdot F_{\bar{a}b} + \bar{b} \cdot F_{\bar{a}\bar{b}})\}$

3. $F = \{a \cdot b \cdot F_{ab} + a \cdot \bar{b} \cdot F_{a\bar{b}}\} + \{\bar{a} \cdot b \cdot F_{\bar{a}b} + \bar{a} \cdot \bar{b} \cdot F_{\bar{a}\bar{b}}\}$

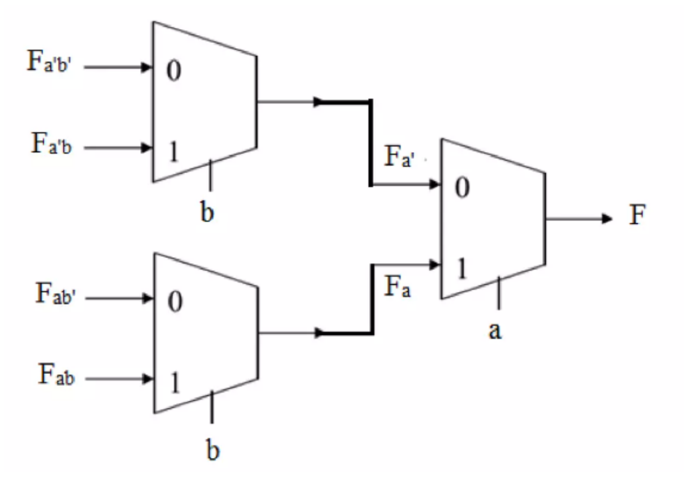


Figure 13: We see a 'tree' of MUXes forming...

2.1.3.3 Logic modules

Actel uses the above MUXes to form MUX-based logic cells.

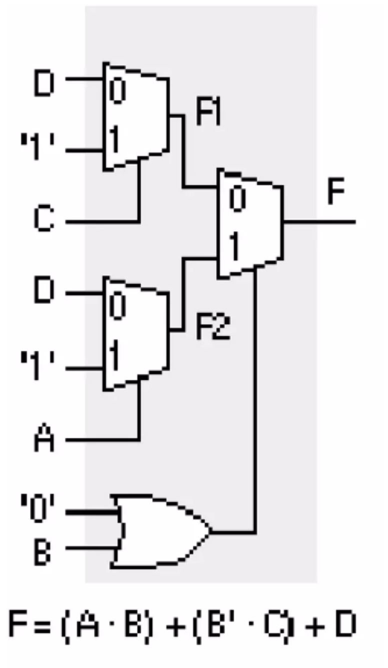


Figure 14: Implementation of combinational logic, using what Actel terms *ACT1 Logic Module*

2.2 Flow of Technology Mapping

Technology Mapping can be broadly broken down into 3 steps . . .

1. Decomposition
   - Restructures the logic network **and library cells** using **base** primitives
   - The purpose of this is so that *pattern matching* at the later stage will be easier (since they are all expressed in terms of the same base primitives)
   - Decomposition to base primitives decreases the level of optimization that can be done (tradeoff for increased simplicity)
2. Partitioning
   - Partitions the large network into smaller sub-networks
   - The purpose of this is so that the *Matching and Covering* stage is easier to handle (ie. divide and conquer strategy)
   - Partitioning introduces module boundaries, decreasing the level of optimization that can be done (tradeoff for increased simplicity)
3. Matching and Covering
   - Determines which library cells may be used to implement a set of nodes in our logic network

3 Logic Decomposition

We decompose the logic network **and library cells** using a choice of base primitives, which creates a new representation.

Note that logic decomposition is a heuristic that does not provide unique solutions.

Figure 15: *NAND2* and *INV* gates are popular choices for base primitives, since **every** boolean function can be implemented with a **combination** of them

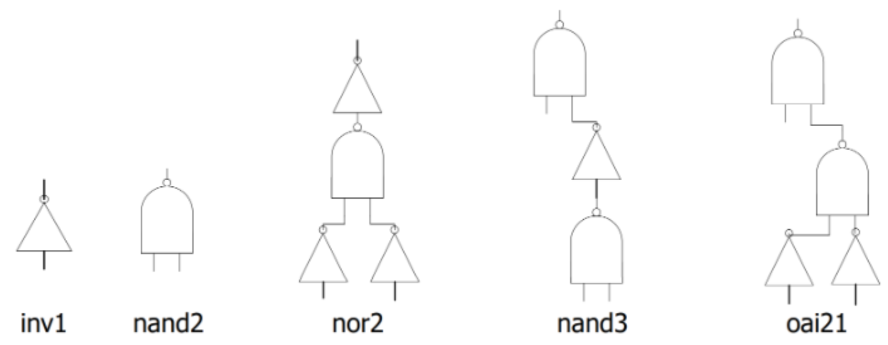


Figure 16: Decomposed library cells (of the IBM POWER4). We term these *Pattern Trees*

Suppose we have a complex gate in our logic network, shown below.

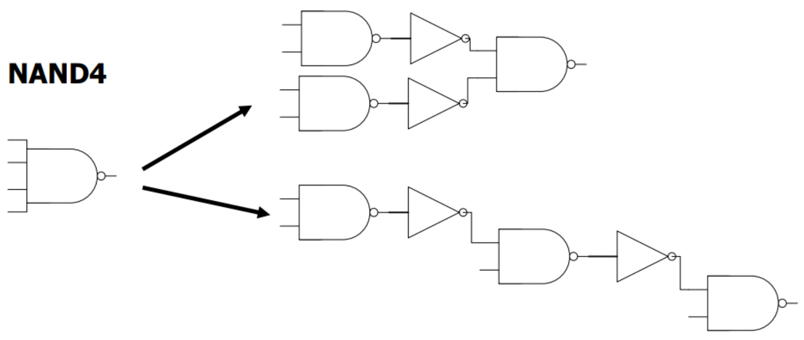


Figure 17: *NAND4* gate may be decomposed in two different ways

We note that the two decompositions are *functionally* identically, but *structurally* (and hence timing-wise) different. Therefore, complex gates may be *matched* by completely different patterns.

4 Partioning

- Reduce a *MIMO* (multi-input-multi-output) network into a collection of *MISO* (multi-input-single-output) sub-networks, with each sub-network termed as a *subject graph*
- This allows for a *Divide and Conquer* approach to the *Covering problem* that we will see later

4.1 Partioning algorithm

1. Mark vertices with multiple-out degrees; Edges connected to these marked vertices define the partition boundary
2. Partition accordingly to form *subject graphs* (ie. break the circuit at fanout points)
3. Note that we may furthur (recursively) partition the subject graphs

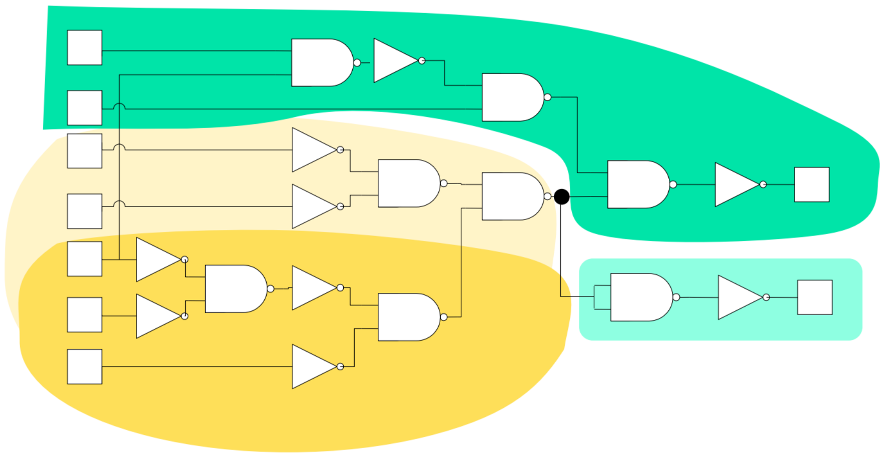


Figure 18: Partitioned graph

5 Pattern Matching and Covering

There are two main types of pattern matching . . .

- Structural Matching
- Boolean Matching

5.1 Structural Matching

Intuitively, it is matching by looking at shapes. We match the logic network with the library cells recursively, until the entire network is mapped.

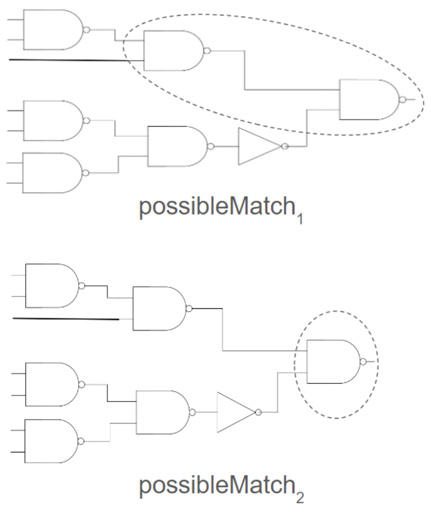


Figure 19: Possible matches (depending on which library cell we map to)

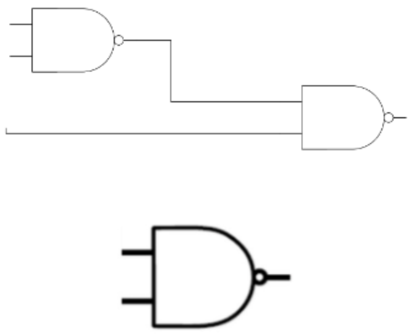


Figure 20: Library cells available

As seen in the example above, there may be multiple matches available for the same set of nodes. We need an algorithm to find which are the best matches that we eventually use to cover the entire network.

5.1.1 Recursive algorithm for Structural Matching

Recursive algorithm to identify if *pattern tree* (decomposed library cells) is *isomorphic* (ie. same shape) to a subgraph of the subject tree. Note that it works only when **one** type of base primitive is used in the decomposition (furthur note: *Inverter* can be seen as *NAND* gate with inputs shorted).

5.1.1.1 Some definitions before we get started

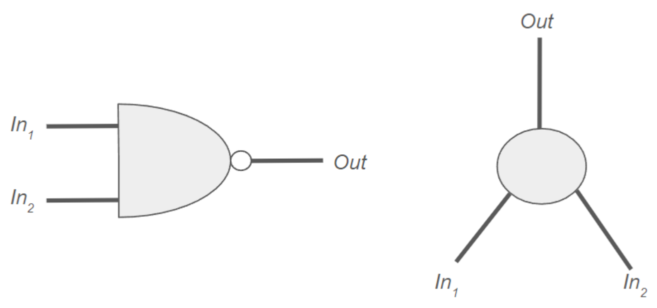
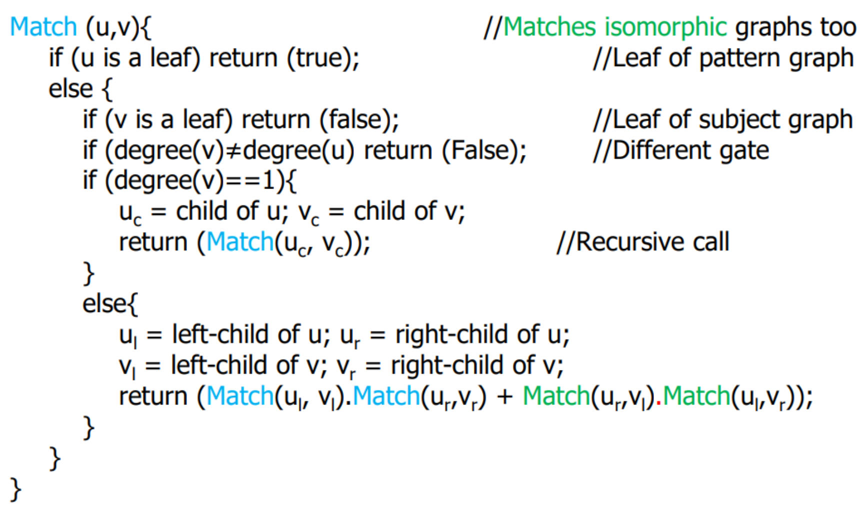


Figure 21: Vertex with degree=2

Note that orientation of the tree!

- Degree of vertex used to indicate the number of children
- u is the root of the **pattern** graph
- v is a vertex of the **subject** graph

5.1.1.2 Algorithm



```
Match (u,v){                                    //Matches isomorphic graphs too
    if (u is a leaf) return (true);                       //Leaf of pattern graph
    else {
        if (v is a leaf) return (false);                  //Leaf of subject graph
        if (degree(v)≠degree(u) return (False);           //Different gate
        if (degree(v)==1){
            $u_c$ = child of u; $v_c$ = child of v;
            return (Match($u_c$, $v_c$));                 //Recursive call
        }
        else{
            $u_l$ = left-child of u; $u_r$ = right-child of u;
            $v_l$ = left-child of v; $v_r$ = right-child of v;
            return (Match($u_l$, $v_l$).Match($u_r$,$v_r$) + Match($u_r$,$v_l$).Match($u_l$,$v_r$));
        }
    }
}
```

Figure 22: Recursive algorithm for matching

5.1.1.3 Algorithm - Example 1

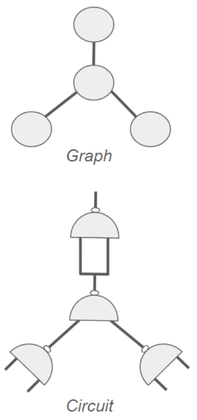


Figure 23: Subject graph, with corresponding circuit

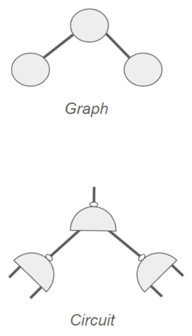
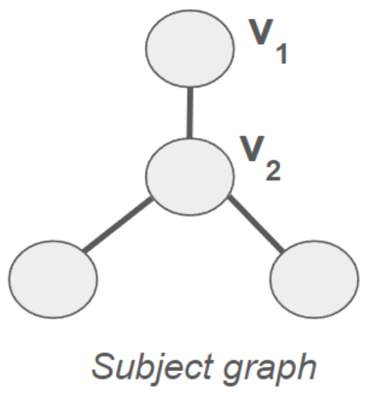
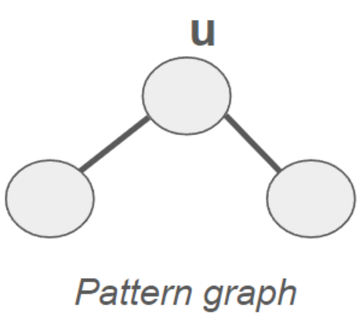


Figure 24: Pattern graph, with corresponding circuit





- If we start from vertex v_1 of the subject graph, then we have that $degree(v_1) = 1 \neq degree(u) = 2$. Thererfore no match.
- If we start from vertex v_2 of the subject graph (assume that v_1 was matched by other library cells), then we have that $degree(v_2) = 2 = degree(u) = 2$. Therefore match *might* be possible. We will enter the *else* statement for furthur checks, eventually finding that match is possible.

5.1.1.4 Algorithm - Example 2

We need to find all possible patterns in the subject network first, before we can decide which combination of patterns is optimal.

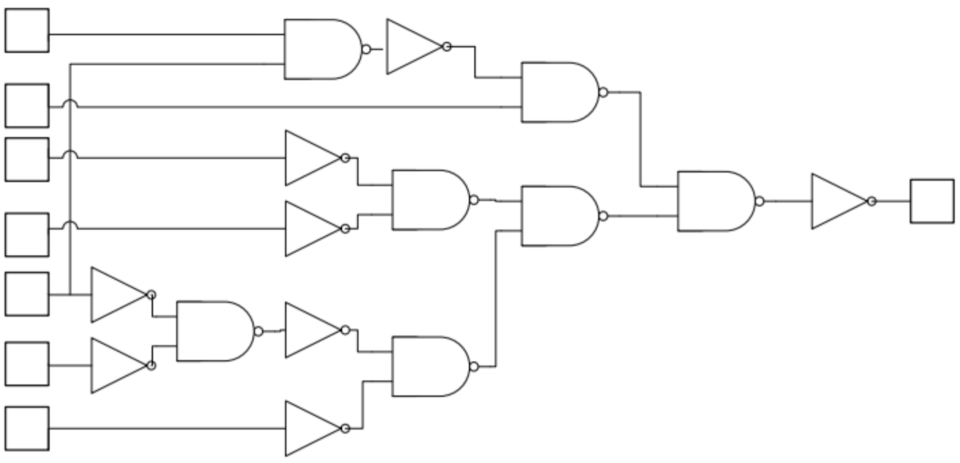


Figure 25: Subject network

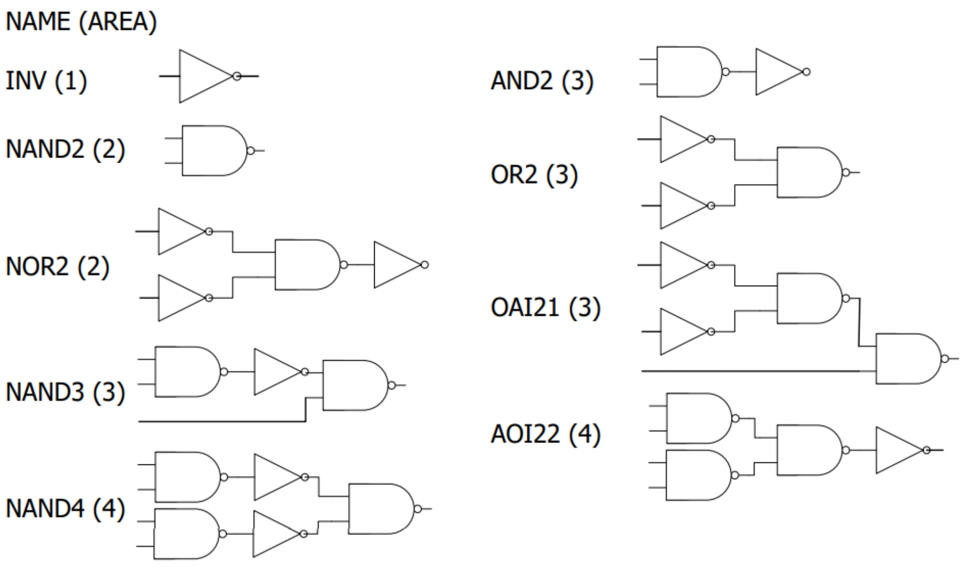


Figure 26: Cell Library

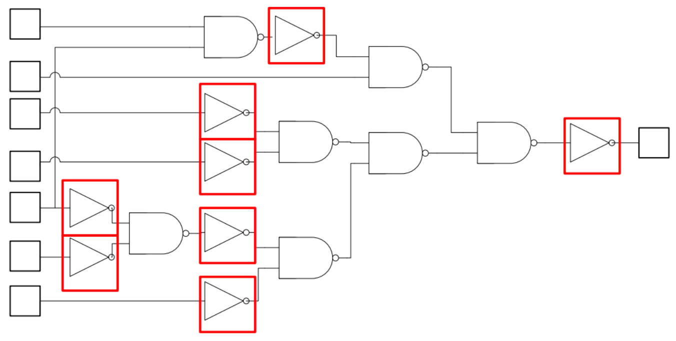


Figure 27: *INV* patterns

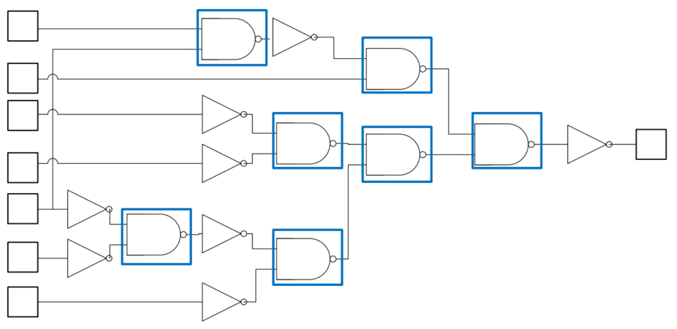


Figure 28: *NAND2* patterns

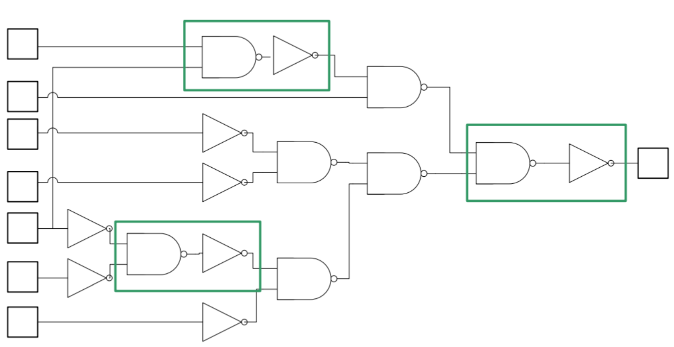


Figure 29: *AND2* patterns

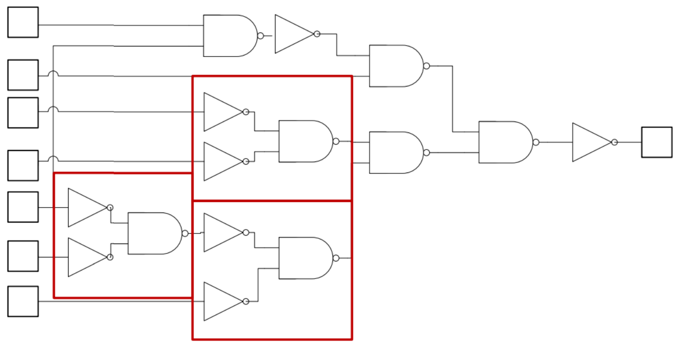


Figure 30: *OR2* patterns

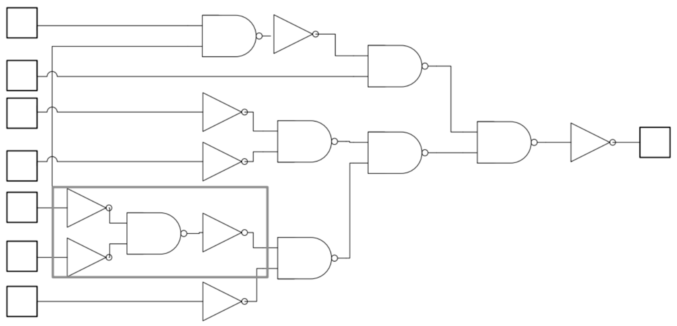


Figure 31: *NOR2* patterns

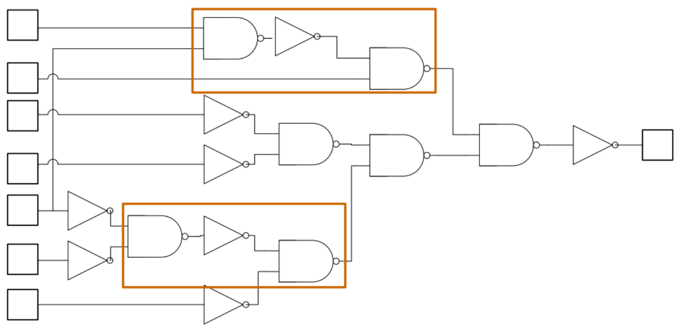


Figure 32: *NAND3* patterns

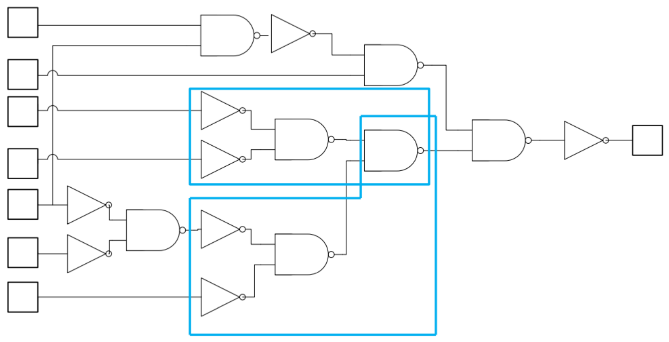


Figure 33: *OAI21* patterns

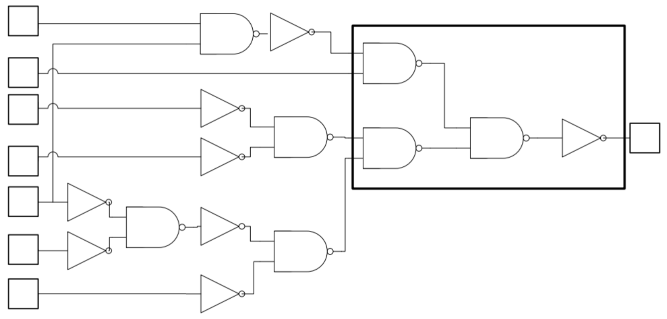


Figure 34: *AOI22* patterns

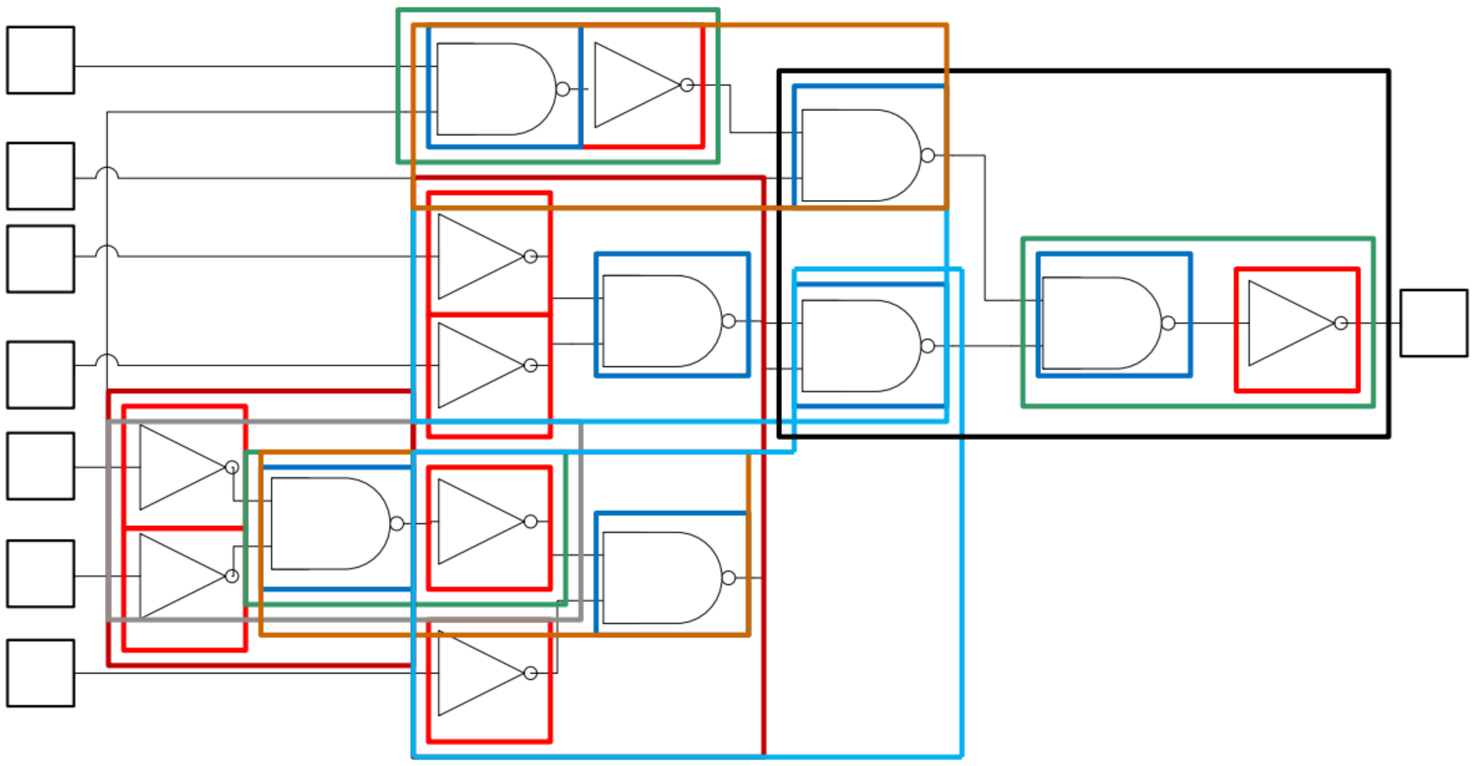


Figure 35: All patterns (superimposed)

Now the question of *optimal coverage*. Which different combinations shall we use to map the entire network?

- Note that a *greedy* approach **may not be optimal**.
- We have to try out all different combinations and see which is optimal. This will be covered later.

5.2 Problems with Structural Matching

5.2.1 Structural Matching works on trees only

5.2.1.1 Recap: Properties of trees

- Each node in the tree can be connected to many children, but must be connected to **exactly one parent** (except for the root node).
- This constraint means that there are no 'cycles' or 'loops' in the tree
- What implications does this have for us?
  - Circuits with a **fan-out** node cannot be represented with a tree
  - Therefore, we cannot apply *matching* across fan-out nodes!

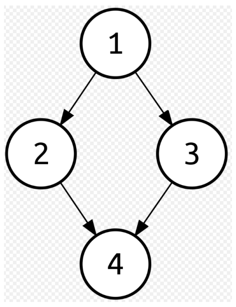


Figure 36: Not a tree. Node 4 has more than one parent.

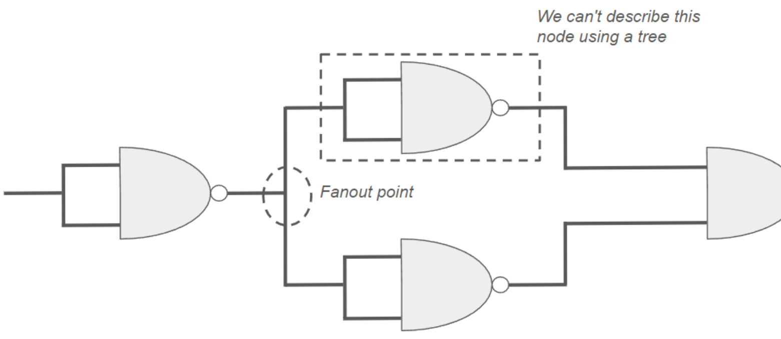


Figure 37: Circuits with fan-out nodes are an issue

This means that *XOR* gates cannot be matched, since *XOR* gates require fan-out (by definition)

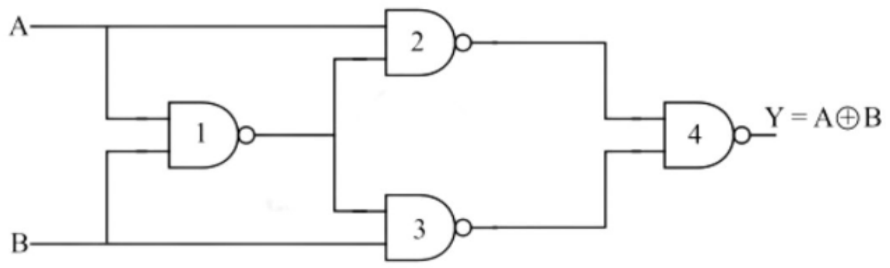


Figure 38: $A \oplus B = (A \cdot \bar{B}) + (\bar{A} \cdot B)$; Thus *XOR* gates require fan-out

5.2.2 Structural Matching has imperfect matching

- Different structures **with the same functionality** are not identified by structural matching.
- This means that we miss out on potential matches, meaning our final result is not fully optimized

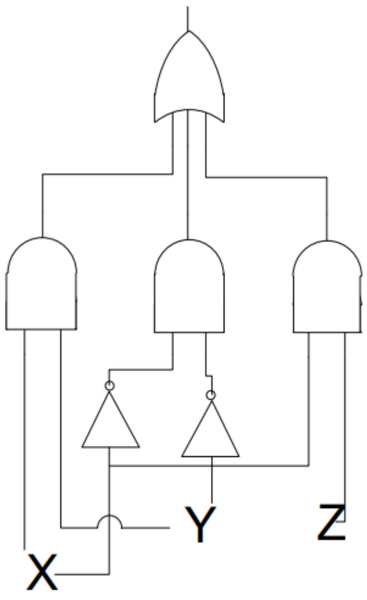


Figure 39: $g = xy + \bar{x}\bar{y} + xz$

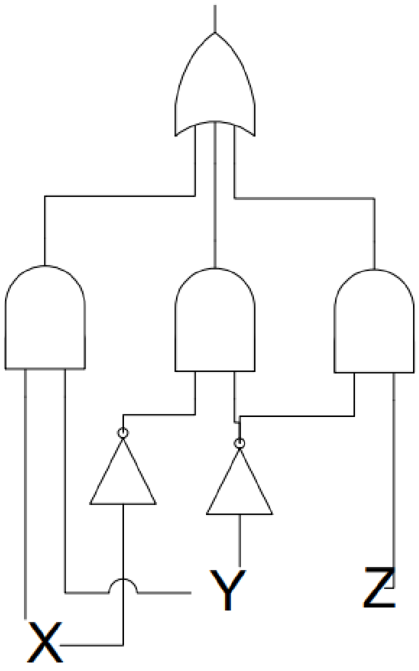


Figure 40: $f = xy + \bar{x}\bar{y} + \bar{y}z$

In the example above, both functions f and g are **functionally identical** (you can check using truth table), but **structurally different**. Structural matching will not identify that they are the same.

The fix to this issue is to do matching by *boolean logic*, rather than by *structure* (ie. isomorphism). This is called *Boolean Matching.*

6 Boolean Matching

Boolean Matching relies on matching the *pattern* to the *subject* logically

- Decomposition independent
- Patterns match via *structural match* $\implies$ Patterns match via *boolean match*
  - Structurally matched pattern must be logically equivalent
- Patterns match via *boolean match* $\not\Rightarrow$ Patterns match via *structural match*
  - Two logically equivalent patterns may have different structures (as we saw in Fig39 and Fig40)

6.1 Boolean Matching - General ideas

1. Define the *cluster function* (what we previously termed *subject graph*) as $f(X) = f(x_1, x_2, \ldots, x_n)$; ie. n input variables
2. Define the *pattern function* (what we previously termed *pattern graph*) as $g(Y) = f(y_1, y_2, \ldots, y_m)$; ie. m cell inputs
3. WLOG, assume $n = m$
4. Matching of functions f and g, involves comparing f and g for *equivalence* (ie. checking if they are logically equivalent)
   4.1. Intuitively, we are examining all *permutations* to find valid assignment(s) of variables from **cluster function** to variables of **pattern function**

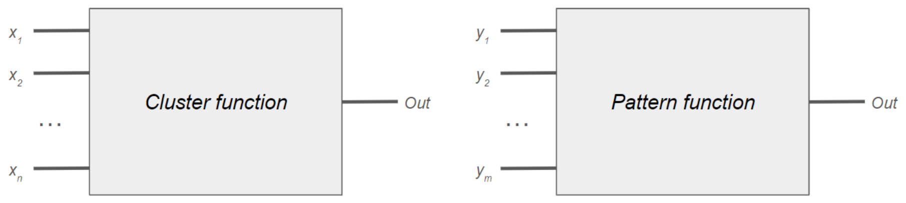


Figure 41: Abstracted view of cluster and pattern functions

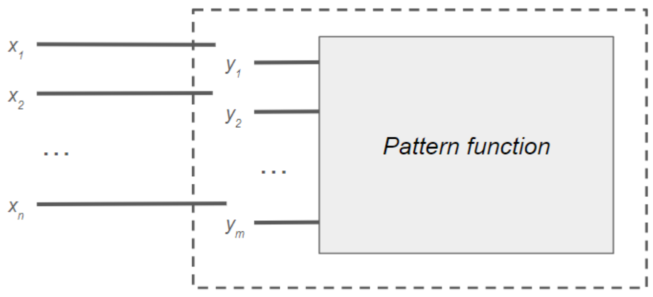


Figure 42: There are many different ways (*permutations*) to connect $\{x_1, \ldots, x_n\}$ with $\{y_1, \ldots, y_n\}$. We need to examine all permutations to find if logical equivalence exists (and if so, under which case is it mapping is it optimal?)

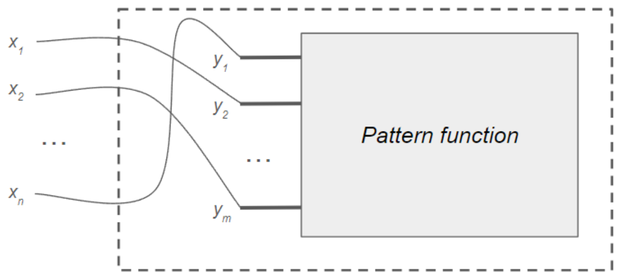


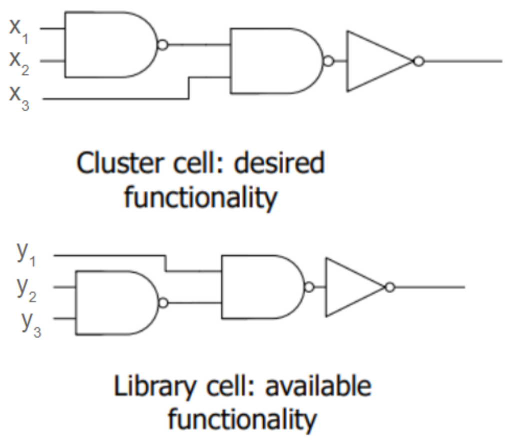
Figure 43: One possible connection might be this. Note that y_1 of pattern function does not correspond to x_1 of cluster function

6.2 Equivalence of functions

Functions may be logically equivalent under different conditions, let's examine an example.

6.2.1 Permutation of input variables

Ordering of input *cluster variables* needs to be changed



x1
x2
x3
Cluster cell: desired functionality
y1
y2
y3
Library cell: available functionality

Figure 44: Can desired functionality be implemented with library cell?

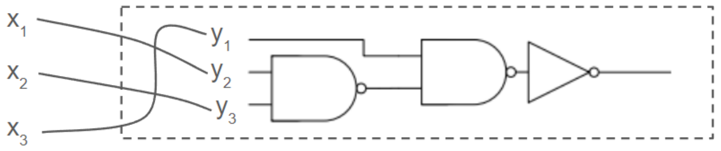


Figure 45: One possible permutation that is valid

Mathematically speaking, we have $g(Y) = f(\rho(X))$ where ρ is some permutation of X

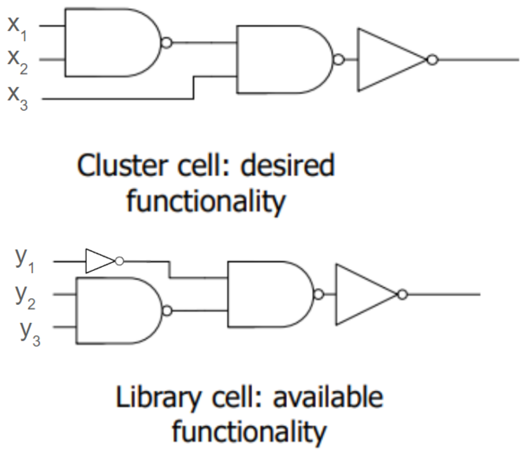
For example ...

1. Suppose $f(X) = x_1\bar{x_3} + x_2x_4$
2. Suppose $g(Y) = y_2\bar{y_4} + y_1y_3$
3. ρ reorders X and maps ...
   - 3.1. $x_1 \rightarrow y_2$
   - 3.2. $x_2 \rightarrow y_1$
   - 3.3. $x_3 \rightarrow y_4$
   - 3.4. $x_4 \rightarrow y_3$
4. Hence, $g(Y) = f(\rho(X))$

If functions are equivalent when variable order is allowed to change, we term that *P-equivalent*

6.2.2 Negation of input variables

Polarity of inputs can be altered, assuming that inputs come from FFs (which provide input and it's complemented version)



x1
x2
x3
Cluster cell: desired functionality
y1
y2
y3
Library cell: available functionality

Figure 46: We cannot directly connect cluster variables to pattern variables now

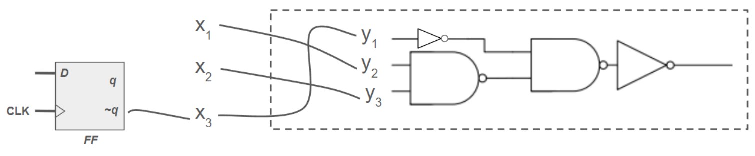


Figure 47: However, taking $\bar{Q}$ from the input FF solves our problems

Mathematically speaking, we have $g(Y) = f(\phi(X))$ where ϕ maps x_i (elements of X) either to y_i or $\bar{y_i}$

For example ...

1. Suppose $f(X) = x_1 + x_2 + x_3$
2. Suppose $g(Y) = \bar{y_1} + \bar{y_2} + y_3$
3. ϕ maps ...
    3.1. $x_1 \rightarrow \bar{y_1}$
    3.2. $x_2 \rightarrow \bar{y_2}$
    3.3. $x_3 \rightarrow y_3$
4. Hence, $g(Y) = f(\phi(X))$

If functions are equivalent when polarity of input variable is allowed to change, we term that *N-equivalent*

6.2.3 Negation of output variables

Same concept of negation of input variables, but this time at the output

For example ...

1. Suppose $f(X) = x_1 + x_2$
2. Suppose $g(Y) = \bar{y_1} \cdot \bar{y_2}$
3. Do $g(Y) = \bar{f}(X)$

6.2.4 Summary

We thus have three categories of equivalence ...

- *NPN-equivalent*: Equivalent under input negation, input permutation, output negation
- *PN-equivalent*: Equivalent under input permutation, output negation
- *P-equivalent*: Equivalent under input permutation

In the **worst-case** scenario of *NPN-equivalence*, we have to check $(2^n \times n! \times 2)$ mappings just to determine if logical equivalence exists!

- 2^n: n input variables may be negated (either 0 or 1)
- $n!$: n input variables have $n!$ different permutations (orderings)
- 2: output variable may be negated (either 0 or 1)

That is alot of mappings to consider. *Boolean signatures* give us a way to reduce the number of possibilities.

7 Boolean Signatures

Boolean signatures are a compact representation that characterizes **some** of the properties of a boolean function.

- Boolean signatures do not match $\implies$ No match is possible
- Boolean signatures match $\implies$ Match **may** be possible
    - Each boolean function has a **unique** signature, but a signature can be related to > 1 functions (this problem is termed *aliasing*)
    - Therefore, signature matching is *necessary*, but **not** a *sufficient* condition
    - Only if **all** possible boolean signatures match, then we can say that function match is possible (but this is not a practical way to do so)

Therefore, the main idea is to use some simple boolean signatures to filter out the solution space, before we start checking function equivalence. If we know that some functions already have non-matching boolean signatures, we need not consider them during the function equivalence stage.

Note that whether the boolean signatures match, also depends on equivalence type we are checking for (NPN, PN, P)?

7.1 Example of boolean signatures

7.1.1 Function symmetry

How many sets of variables are pairwise interchangable without affecting the logic?

- $\bar{A}\bar{B} + ABC$; A, B are pairwise interchangable (1 set)
- $\bar{B}(A + \bar{C}) + C$; No variables are pairwise interchangable (0 set)

7.1.2 Number of unate/binate variables

- Unate variables: Variables with which a function *monotonically* increases/decreases
- Binate variables: Not unate variables

An example . . . $A\bar{C} + \bar{A}B$

- B, C are unate variables (B **always** pulls towards logic high, C **always** pulls towards logic low)
- A is binate variable (A can pull to logic high or low, depending on circumstances)

7.2 Summary

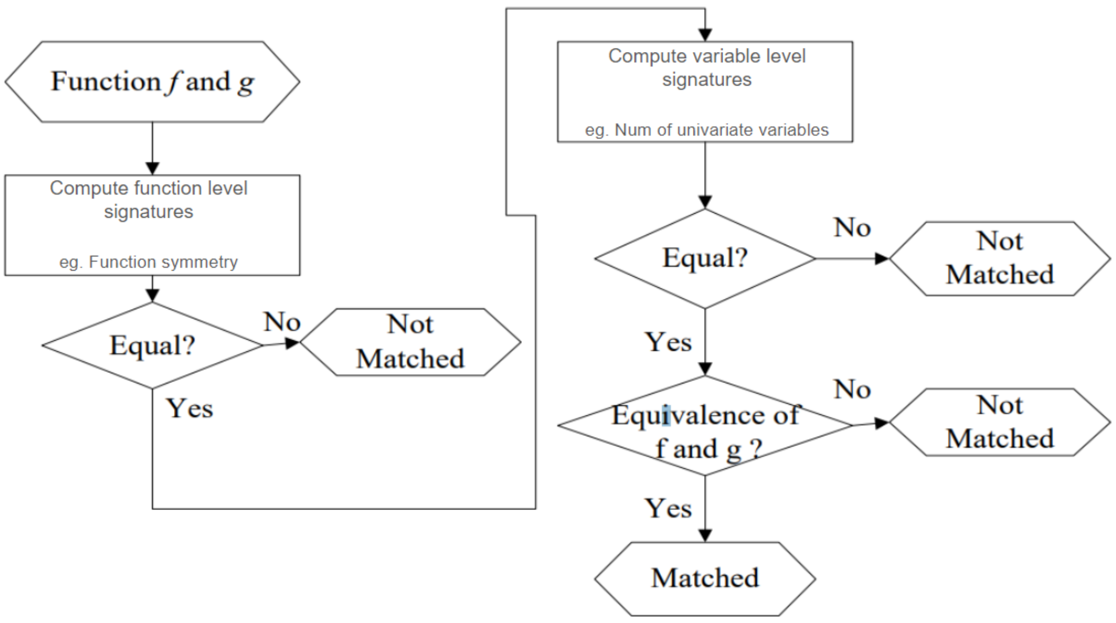


Figure 48: Flowchart

8 Covering

Once we find matches between the pattern-tree (decomposed library cells) and subgraphs of the subject tree, which match shall we pick to implement our actual circuit?

Formally speaking . . .

- Define *covering* as a collection of pattern graphs st. every node of the subject graph is contained in one (or more) of the pattern graphs
- We then want to find the **minimum** cost covering - which particular collection of pattern graphs will achieve this?

8.1 Covering of trees and DAGs

8.1.1 Trees

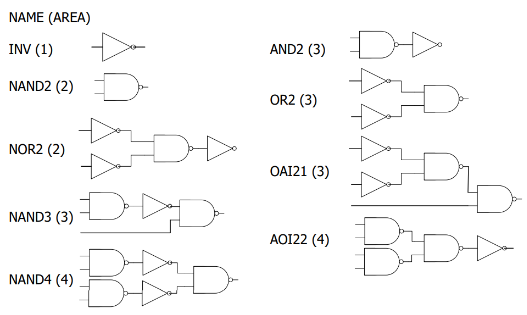


Figure 49: Cell Library

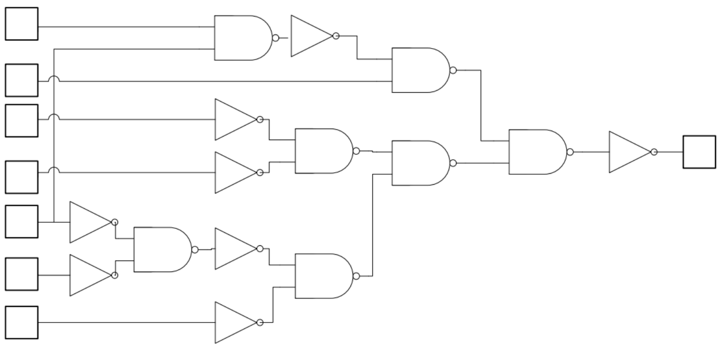


Figure 50: Subject graph **with no fanout nodes**

- Consider the example above, where the subject graph **has no fanout nodes**.
- We know that we can convert the subject graph to a *tree*, since there are no fan-out nodes
- We also know that we can covert the cell library to it's appropriate tree representation
- Therefore, the covering problem to cover the 'subject-tree' using 'pattern-trees', which **has a P solution**.

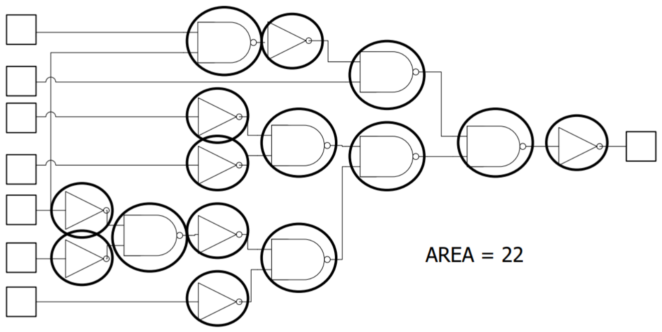


Figure 51: Trivial covering

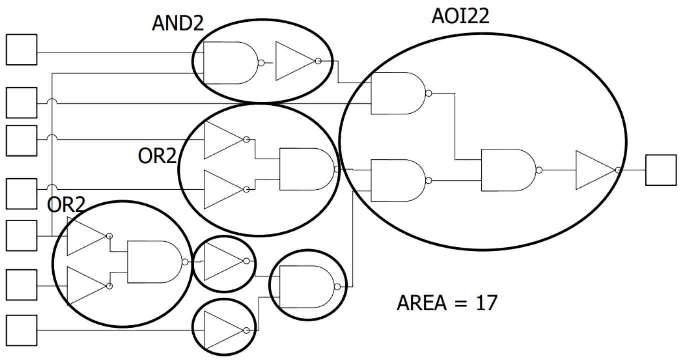


Figure 52: Better covering

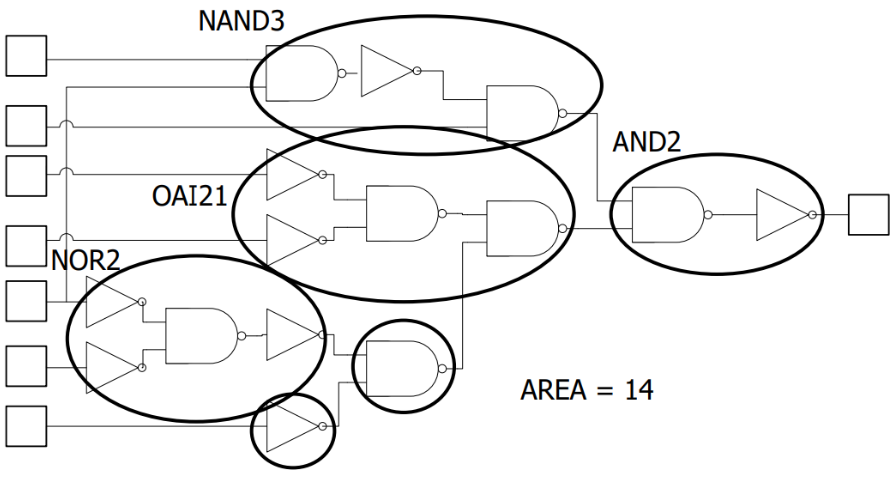


Figure 53: Even better covering

8.1.2 DAGs

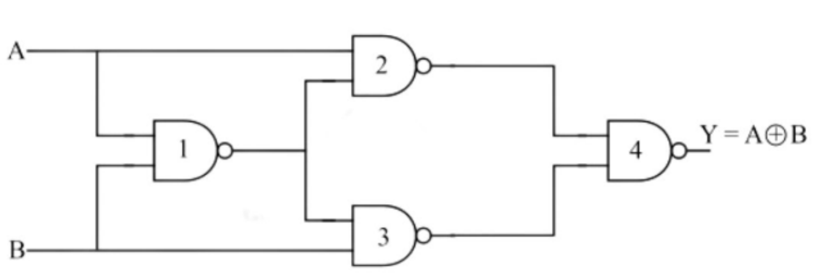


Figure 54: *XOR* gate

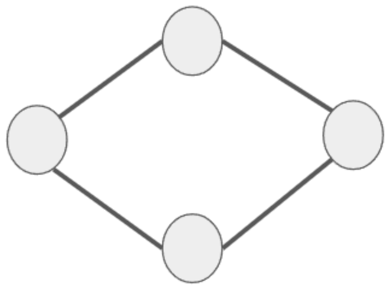


Figure 55: *XOR* gate representation - DAG

Covering a 'subject-**DAG**' using a 'pattern-tree' is termed *DAG-covering*, and is **NP-hard**.

8.1.2.1 Treeifying

Partioning a circuit with fan-out nodes (ie. DAG) at the fan-out node points, thereby forming a collection of **trees**.

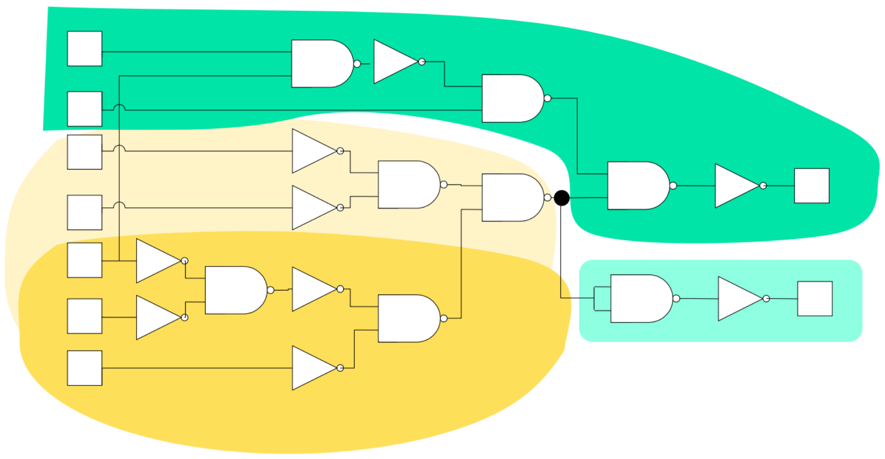


Figure 56: DAG is now transformed into a collection of subnetworks, where each subnetwork is a tree

- We can then use P solution of tree-covering on each of these sub-networks.
- The tradeoff is lowered optimization - We are not able to do inter-subnetwork optimizations

Also note that when the DAG is broken into trees at the fan-out points, some of the logic is duplicated.

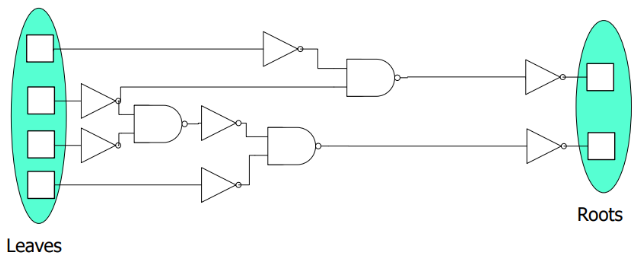


Figure 57: DAG
($\exists$ route between leaf(input) to **two** roots(outputs))

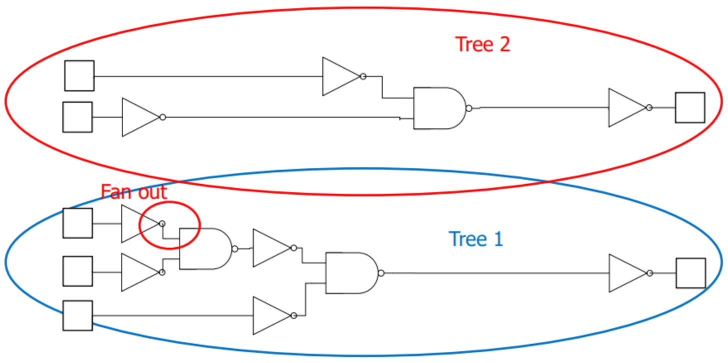


Figure 58: Logic at fan-out node is duplicated

8.2 Optimal tree covering using Dynamic Programming

- We use the *memoization* DP approach, visiting the subject tree **bottom-up** (it's easier this way)
- At each vertex, we attempt to match the **locally** rooted sub-tree to all library cells
  - Find the best match and record it at the vertex (ie. Memoization)
- Work your way up to the root

8.2.1 DP example

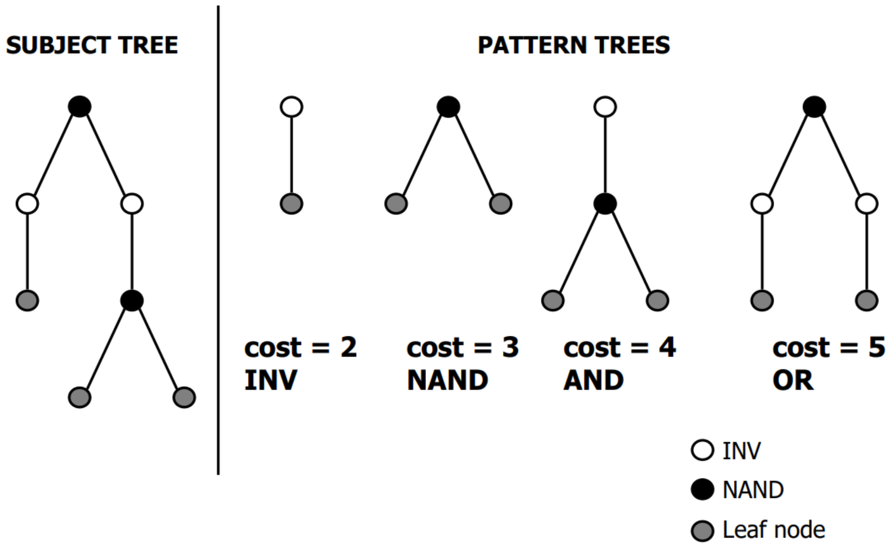


Figure 59: Subject and Pattern trees

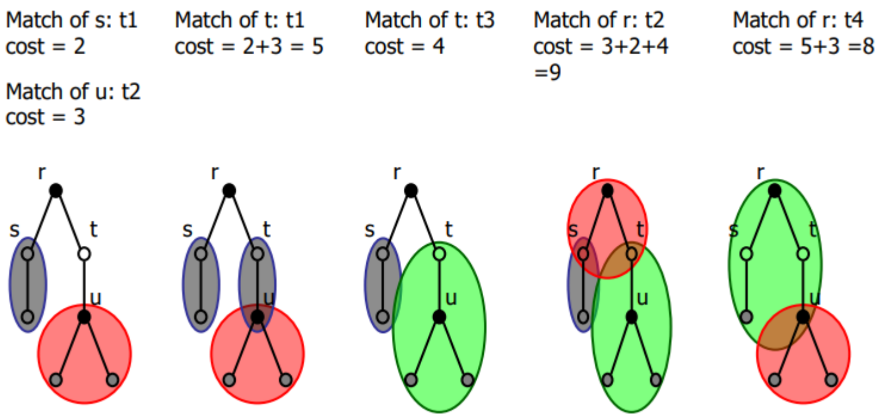


Figure 60: Optimally cover the Subject tree using Pattern trees
Approach this using DP (Memoization)

9 Technology Mapping on FPGAs

For FPGAs, the library cells are *LUTs* (higher abstraction level than ASICs).

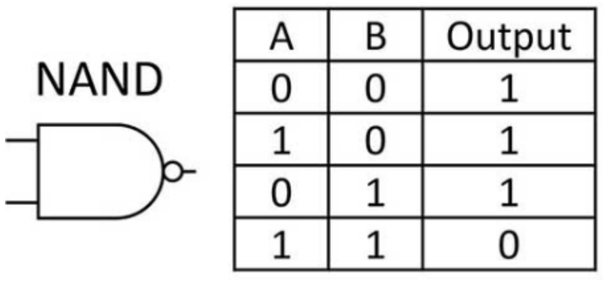
9.1 How to do matching with LUTs?

Unlike previous sections, we do not decompose CLBs (library cell of FPGAs) using primitives (eg. *NAND2* and *INV*) and try to match using them, as it is not practical. For example, a 4-input LUT can implement $2^{2^4} = 256$ possible combinational functions. Decomposing all 256 of those combinational functions is not a good idea.

So how do we match with LUTs? We simply configure the *LUT-mask* to match the output of the cluster function.

9.1.1 LUT Matching - Example

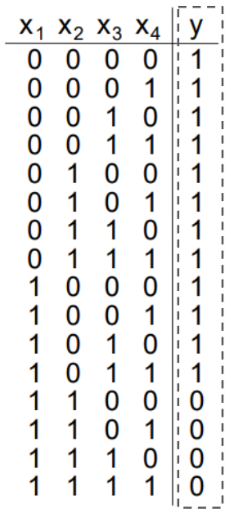
Suppose we want to match a *NAND*-gate with a *4LUT*.



A	B	Output
0	0	1
1	0	1
0	1	1
1	1	0

Figure 61: *NAND* truth table

We derive the truth table for a *4LUT* that matches the *NAND*'s truth table, and simply 'copy' y over to the *LUT-mask*.



x_1	x_2	x_3	x_4	y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Figure 62: Notice that x_3 and x_4 inputs are 'useless'

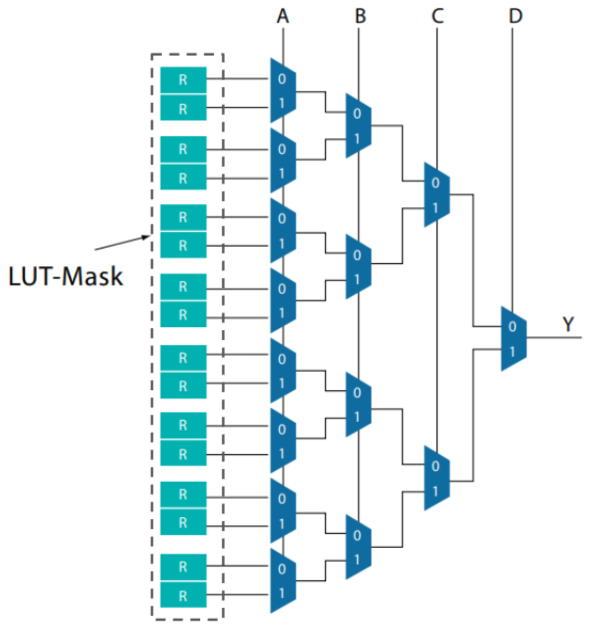


Figure 63: LUT-mask of 4LUT

We can see that it is very inefficient (in terms of hardware resources) to match the *NAND*-gate with a *4LUT*.

9.2 Mapping and Covering on FPGAs

Simply put, FPGA mapping involves transforming the original *subject* network into another network, where **every** node corresponds to an $nLUT$ (where $nLUT$ is the one afforded by our library cell).

Covering is then finding the optimal mapping to utilize, suject to some constraints (ie. use the minimal number of LUTs possible).

Assume that we have decomposed our subject network using whatever primitives were available. Below is how we would do FPGA mapping and covering . . .

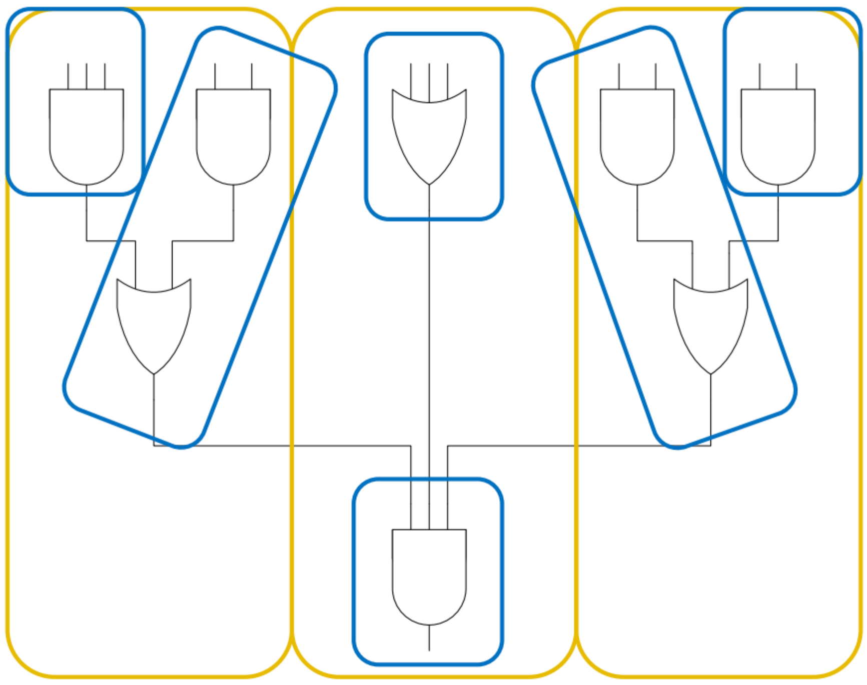


Figure 64: We require 3 $5LUT$s or 6 $3LUT$s to map & cover this circuit
Notice that a $5LUT$ can implement a $4LUT$, one of the inputs will just be 'X'

9.3 Bin Packing

- So far, all our SOP expressions can each 'fit' into a single LUT, but this is not always the case.
- Consider an SOP expression of a single-output function, suppose it has at most n variables
  - If we have an $nLUT$, then the **entire** SOP expression can be implemented by one $nLUT$ (which is what we previously assumed)
- Suppose this SOP expression has $> n$ variables now
  - If we only have an $nLUT$, then the SOP expression needs to be 'split' up and assigned to different $nLUT$s
  - This ideas is similar (but not quite identical) to the *bin packing problem*, where we must pack a given set of objects (here product terms) of bounded size into bins (here tables)
- It is not exactly similar to the *bin packing problem*, since partitioning alone is not sufficient. We require **more bins** to hold the combined partitioned terms
  - Suppose *table size* is $n = 3$ (ie. It's a $3LUT$)
  - Suppose *SOP expression* is $f = ab + cd$
  - 2 $3LUT$s are needed to implement ab and cd, but 1 more $3LUT$ is needed to combine them together
  - Notice that there is room for optimization. For each $3LUT$, one of the inputs is unused. Could we squeeze additional logic into there? This leads us to *Modified Bin Packing.*

9.3.1 Modified Bin Packing (Heuristic algorithm)

- Iterate the following steps until all product terms have been 'binned'
  1. Select product term with the most variables

    2. Place it into any table where it fits
    3. If no table has enough place to fit it, create a new table and place it there

- When all product terms are 'binned', do the following steps
    1. Find the table t_i with the fewest number of **unused** variables (alternatively, table with most number of **used** variables) and associate a variable v_i with it's **output**
    2. Assign v_i to the first table t_j that can accept it
    3. Repeat from (1) until only one table is left

9.3.1.1 Modified Bin Packing - Example

- Suppose *table size* is $n = 3$ (ie. It's a *3LUT*)
- Suppose *SOP expression* is $f = ab + cd$

1. One table is created for ab, call it t_1
2. cd cannot fit in t_1, create another table t_2 for it
3. All terms finished. Let's pick t_1 as the table with fewest number of unused variables
4. Associate variable v_1 with t_1 (representing the table's output)
5. Fit v_1 into the unused **input** of t_2
6. Terminate algorithm, only one table (t_2) left
7. Final output of is $f = v_1 + cd$, which can be implemented by 2 *3LUT*s, compared to 3 *3LUT*s before

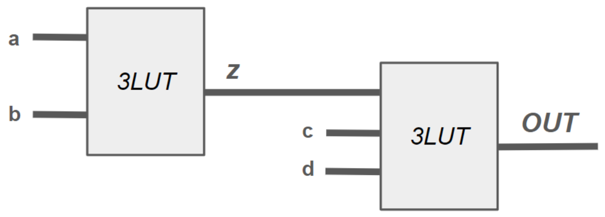


Figure 65: Modified Bin Packing

9.4 CLB packing

Usually, the CLB's in the FPGAs have more than a singe LUT within them.

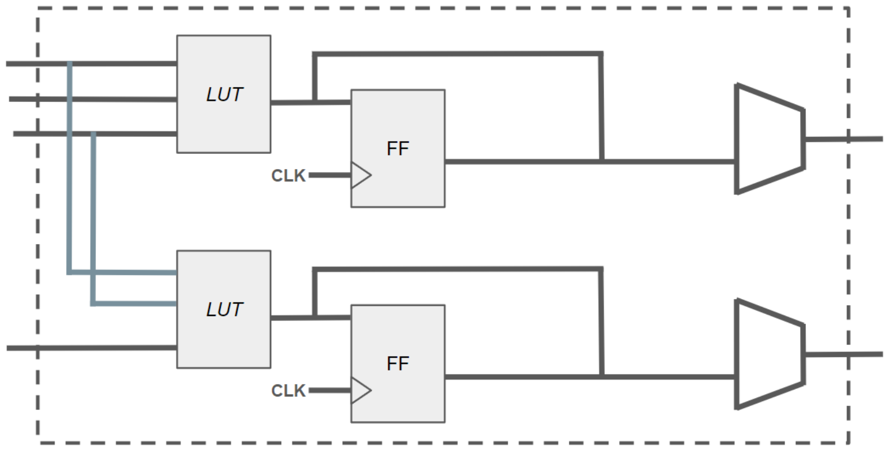


Figure 66: Logic Block
Observe that the LUTs can share inputs as well

Logic block packing packs several LUTs and registers into one logic block under the following constraints

- Number of LUTs in a logic block
- Number of **distinct** input signals

Therefore, our optimization goal is to ...

- Pack LUTs st. routing signals are **minimized** between logic blocks
  - External routing is expensive, internal routing within the logic block is cheaper
- Fill each logic block to it's maximal capacity, to reduce to total number of logic blocks in the FPGA fabric

We can apply the same ideas of 'bin-packing' as covered in the previous sections.

9.5 Summary

Technology mapping on FPGAs involves ...

- Mapping
- Covering
- **Packing**