

Why Synthesis?

As design complexity increases, manually designing circuits is impossible and very inefficient. Here are some of the limitations of manual design process that is addressed by automated logic synthesis tools.

Error-Prone: For large and complex designs, manual conversion of high level design was prone to human error.

Hard to Verify: The designer could never be sure that the design constraints were going to be met until the gate-level implementation is complete and tested.

Time-Consuming: A significant portion of the design cycle was dominated by the time taken to convert a high-level design into gates.

Hard to reuse: Design reuse was not possible.

Impossible to optimize globally: Each designer would implement design blocks differently. For large designs, this could mean that smaller blocks were optimized but the overall design was not optimal.

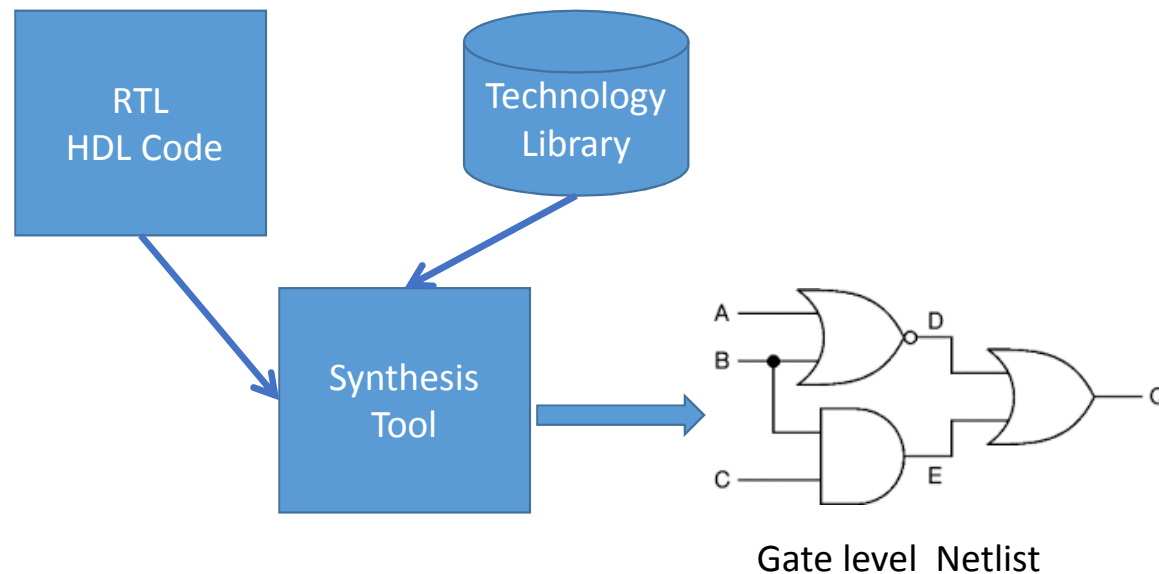
What is logic Synthesis

Logic synthesis is the process of converting a HDL design into an optimized gate-level form. Logic synthesis uses a standard cell library such as basic logic gates like and, or, and nor, or macro cells, such as adder, multiplexers, memory, and flip-flops. Standard cells put together are called technology library. Technology libraries are provided by fabrication house (such as TSMC, UMC, INTC etc.) based on technology of manufacturing. Normally the technology library is known by the transistor size (e.g.. 90nm, 65nm, 45nm or 14nm). The term “14nm” refers to the size of the transistors that are used to make the whole IC or the chip. Reducing size of transistors has multitude of benefits and scaled transistor provide:

- Higher Performance
- Lower Power
- Lower cost per transistor
- Improved leakage
- Higher density(more transistors in same space)

Synthesis Process

Conversion from high-level design to gates is done by synthesis tools(synopsys Design compiler is widely used for logic synthesis), using various algorithms to optimize the design as a whole. Diagram below describes simplified form of synthesis process.



Impact of Logic Synthesis

Automated Logic synthesis tools provides powerful way to design very complex and large circuits efficiently that reduces design cycle time significantly.

- Fewer human error, because designs are described at a higher level of abstraction
- High-level design is done without significant concern about design constraints.
- Conversion from high-level design to gates is fast and accurate.
- Logic synthesis tools can optimize the design globally.
- Logic synthesis tools allow technology-independent design (through synthesis, design can be translated to any technology)
- Design reuse is possible (reuse the higher-level description).

What is Logic Synthesis?

Synthesis = translation + optimization + mapping

Logic synthesis is the process of combining primitive logic functions to form a design netlist that meets functional and design goals.

Translation: Is the process that translates Verilog (HDL) source files into gate level netlist. HDL Compiler translates Verilog HDL descriptions into gate-level netlist.

Optimization: Is the process of optimizing design based on your design constraints (Design goals drive optimization). Design goals could be speed, area or power. To get most out of synthesis designers should use various approaches available to optimize design that meets design goals. There is always a trade-off between the circuit timing and area.

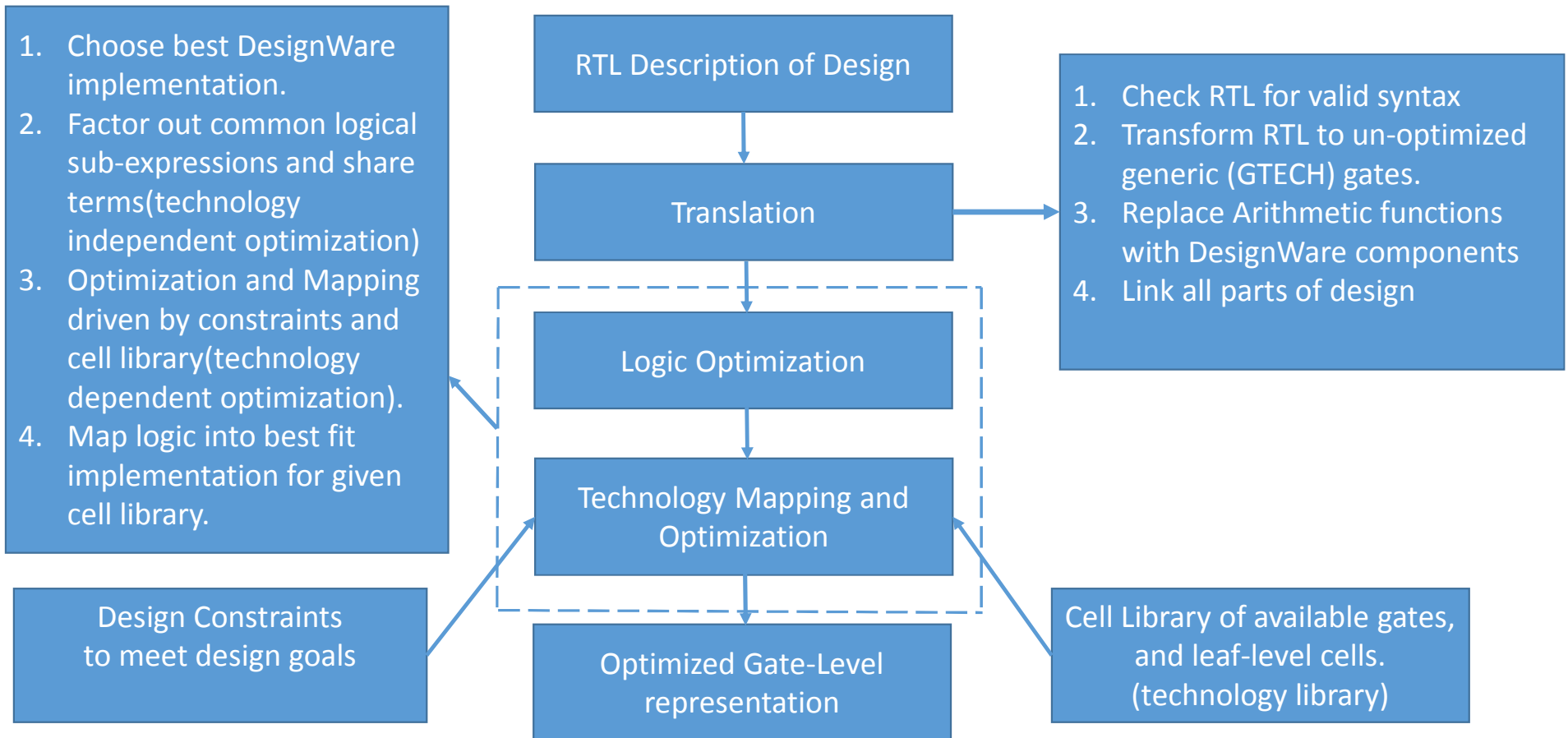
- Area constraints (Transistor count ->The area of the final layout must not exceed a limit).
- Timing constraints (Number of Levels (delays/Speed) -> The circuit must meet certain timing requirements).
- Power (Number of Circuits -> The power dissipation in the circuit must not exceed a threshold).

Mapping: Synthesis tool (Design Compiler) maps design block to gate level design with a user specified library.

Main challenges:

- Use Coding Style that is Synthesizable (use Verilog synthesizable constructs)
- Specify reasonable design constraints
- Use HDL synthesis tool efficiently (Design Partitioning to achieve design goals)

Synthesis Flow



Synthesis Flow (Translation + Optimization + Mapping)

Typical Synthesis flow is described in few steps below.

```
reg [7:0] data_out;  
integer index;  
data_out= 8'h0000;  
if (msb_bits[1:0] == 2'b10)  
    data_out = state[index];  
else  
    data_out = 8'h0000;
```

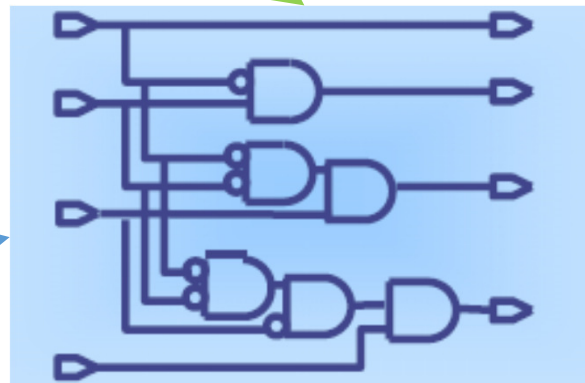
**Hardware Description Language
(HDL):Verilog code**

No timing information

Translation Process(Convert HDL to
Functional Boolean equivalent):

1. HDL syntax/rule checks
2. Optimizes HDL
3. Arithmetic function mapping
4. Sequential function mapping
5. Combinational function mapping

**Translation
(Verilog read)**



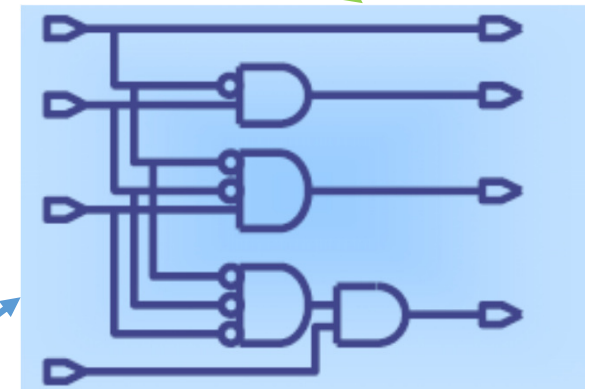
Generic Boolean Gates(GTECH)

timing information

Mapping + Optimization Process:

1. Maps Boolean functions to technology specific primitive
2. Modifies mapping to meet design goals(Design rules, Timing, Area, and Power)

**Optimization + Mapping
(HDL Compiler)**



**Target Technology Library
(standard cells)**

Verilog HDL Compiler Supported Constructs (Synthesizable)

Not all Verilog constructs are synthesizable. Only a subset of Verilog constructs can be synthesized and supported by Verilog HDL Compiler.

Supported Definitions and Declarations

- parameter declarations
- wire, wond, wor, reg declarations(signals and variables)
- Input, output and inout (ports) declarations
- module definitions
- module instantiations
- gate instantiations
- function and task definitions
- Procedural blocks(begin, end, named blocks, disable)
- named Blocks(disable- Disabling of named block supported)

Synthesizable Verilog primitive cells:

- and, or, not, nand, nor, xor, xnor
- bufif0, bufif1, notif0, notif1

Supported Structural Language Constructs

- always statement(Procedural statement)
- assign statement (continuous assignment statement)
- If, else, elseif statements
- case, casex, casez
- for loop, while and forever loop(while and forever loops must contain @(posedge clk) or @(negedge clk)) in order to be synthesizable.

Supported Verilog Operators:

- Binary bit-wise (~, &, |, ^, ~^)
- Unary reduction (&, ~&, |, ~|, ^, ~^)
- Logical (!, &&, ||)
- 2's complement arithmetic (+, -, *)
- Relational (>, <, >=, <=) and Equality (==, !=),
- concatenation({}), replication({{ }})
- Logical shift (>>, <<) and conditional(? :)

Verilog HDL Compiler Unsupported Constructs(not synthesizable)

These are the Verilog constructs not supported by the compiler.

Un-supported Definitions and Declarations

- primitive user-defined
- time declaration
- event declaration
- triand, trior, tri1, tri0, trireg net types
- Ranges and arrays for integers

Unsupported statements:

- initial statement
- repeat statement
- defparam statement
- delay control
- event control
- wait statement
- fork and join statement
- deassign statement
- force and release statement

Unsupported Operators:

- Case equality (===) and Case inequality (!==)
- Division and modulus (division can be done using soft macros like dividers, multipliers, adders, comparators, etc. available in DesignWare library, call synthetic library through DesignWare instantiation)

Unsupported Gate-Level Constructs:

- nmos, pmos, cmos, rnmos, rpmos, rcmos
- pullup, pulldown, tranif0
- rtran, tranif0, tranif1, rtranif0, rtranif1

Unsupported miscellaneous construct:

- hierarchical names within a module

Pre-RTL Coding Preparation:

Prepare High Level Design Specification:

- Write the specification of your design(prepare a high level specification document)
- Partition your design (break your design into major functional blocks)
- Create block level drawing of each functional block in your design.
- Define clearly each functional blocks (Port lists and overall functional description for each block in your design from top to bottom).
- Define Hierarchy for design implementation (Top-down or Bottom-up approach: in practice mostly a combined approach is used for implementation)
- Review your design specification with team and update your design specification based on feedback from team.
- Every team member should have the final design specification before they start coding.

Verilog RTL Coding Style

- **Code for readability (always think of the poor guy who has to read your RTL code)**
 - Use a general naming convention through out your code (use meaning full name for ports, signals, parameters etc.)
 - Add comments (add comments appropriately to explain blocks, functions, ports, signals, and variables, or groups of signals or variables near the code they describe)
 - Add Headers(Header file include Legal statement(confidentiality, copyright, restrictions on reproduction), Filename , Author and Description of function and list of key features of the module)
 - Use indentation to improve the readability of continued code lines and nested loops (use spaces)
 - Do not use Verilog reserved words for names of any elements in code.
 - Declare the ports in order, first input then output ports.

Verilog General Naming Conventions

- Use lowercase letters for all signal names, variable names, and port names (**reset**, **clk**, **data_in**, **data_out**).
- Use uppercase letters for names of constants and user defined types(**`define BUS_SIZE 8**).
- Use meaningful names for signals, ports, functions, and parameters (Do not use *ra* for a RAM address bus, use *ram_addr*)
- Use the same name or similar names for ports and signals that are connected (reset, RESET)
- Use short but descriptive names for parameters(**parameter FIFO_DEPTH = 16**)
- For active low signals, end the signal name with an underscore followed by a lowercase character (e.g. **reset_n**: *active low reset signal*).
- Use a distinctive suffix for state variable names (**<name>_cs** for the current state and **<name>_ns** for the next state)

Verilog RTL Coding Practices for Synthesis

Good coding practices helps to achieve better results in synthesis. Design Partitioning and optimization process results in better circuit. Here are some of useful guidelines that helps to accurately simulate hardware, modeled using Verilog. Adherence to these guidelines will also remove 90-100% of the Verilog race conditions encountered by most Verilog designers and avoid simulation and synthesis mismatch.

- When modeling sequential logic, use nonblocking assignments(`<=`) including modeling latches.
- When modeling combinational logic with an `always` block, use blocking assignments(`=`).
- When modeling both sequential and combinational logic within the same `always` block, use nonblocking assignments.
- Do not mix blocking and nonblocking assignments in the same `always` block.
- Do not make assignments to the same variable from more than one `always` block.

Verilog RTL Coding Practices for Synthesis Continue..

- Use Verilog Constructs that are synthesizable in RTL Coding
- Practice Synchronous design for synthesis (Asynchronous circuits are more difficult to design and should be used carefully in design: Even though asynchronous design has several advantages synchronous design is practiced more).
- Avoid Combination Feedback Loop(HDL Compiler will automatically open up asynchronous logic loops and Without disabling the combinational feedback loop, the static timing analyzer can't resolve).
- Specify the output of a combinational behavior for all possible cases of its inputs(For example, If a case statement is not a full case, it will infer a latch).
- A variable assigned within an always block that is not fully specified will synthesize as latch.
- In a combinational always block, Sensitivity list must be specified completely, otherwise pre-synthesis and post-synthesis simulation may mismatch.

Verilog RTL Coding Practices for Synthesis Continue..

- Keep related combinational logic/block together in the same module.
- Avoid unintentional latch inference.
- For each block in a hierarchical design, register all output signals (Makes output drive strengths and input delays).
- Separate Modules that Have Different Design Goals (This allows Synthesis tools can perform speed optimization on the critical path logic, while performing area optimization on the noncritical path logic).
- Partition design so every module input signal is synchronized to the same clock domain before entering the module(makes module completely synchronous)
- Initialize all registers with reset signal.
- Eliminate Glue logic at top level.
- Resource sharing for optimization(For synthesis tools to consider resource sharing all relevant resources all relevant resources need to be within the same always block.
- Use Parentheses Properly for optimization (out = ((a+(b+c))+d+e)+f;)

Logic Synthesis and Optimization

Technology independent optimization:

- **High Level Optimization**
 - *Resource Sharing*
 - *Implementation Selection*
 - *Arithmetic Optimization*
- **Logic Level Optimization-minimize number of literals** (Logic Level Optimization is independent of technology library)
 - *Flattening*
 - *Structuring*

Technology dependent optimization:

- **Gate level optimization**
 - *Technology mapping(Speed, Area and Power)*
 - *In-Place*
 - *Boundary*

High Level Optimization

Optimization through Resource Sharing: HDL Compiler automatically selects the best structure for your design based on your high level architecture. One of the techniques that HDL Compiler employs is Optimization through resource sharing.

//An Example of Resource Sharing is shown below:

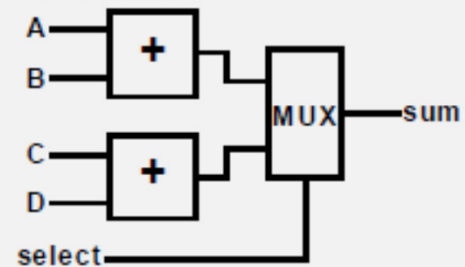
//Given the following HDL description:

```
if (select)
    sum = A + B;
else
    sum = C + D;
```

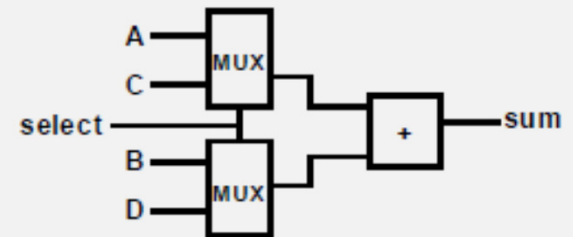
/* Resources can be shared only if they are in the same process, A1 + B1 and C1 + D1 can share resources because they are in same always block process. But A1 + B1 can not do resource sharing with A2 + B2 because they are in different process */

```
always@(*)          always@(*)
  if (select1)       if(select2)
    Z1 = A1 + B1;    Z2 = A2 + B2;
  else               else
    Z1 = C1 + D1;    Z2 = C2 + D2;
```

One possible implementation:

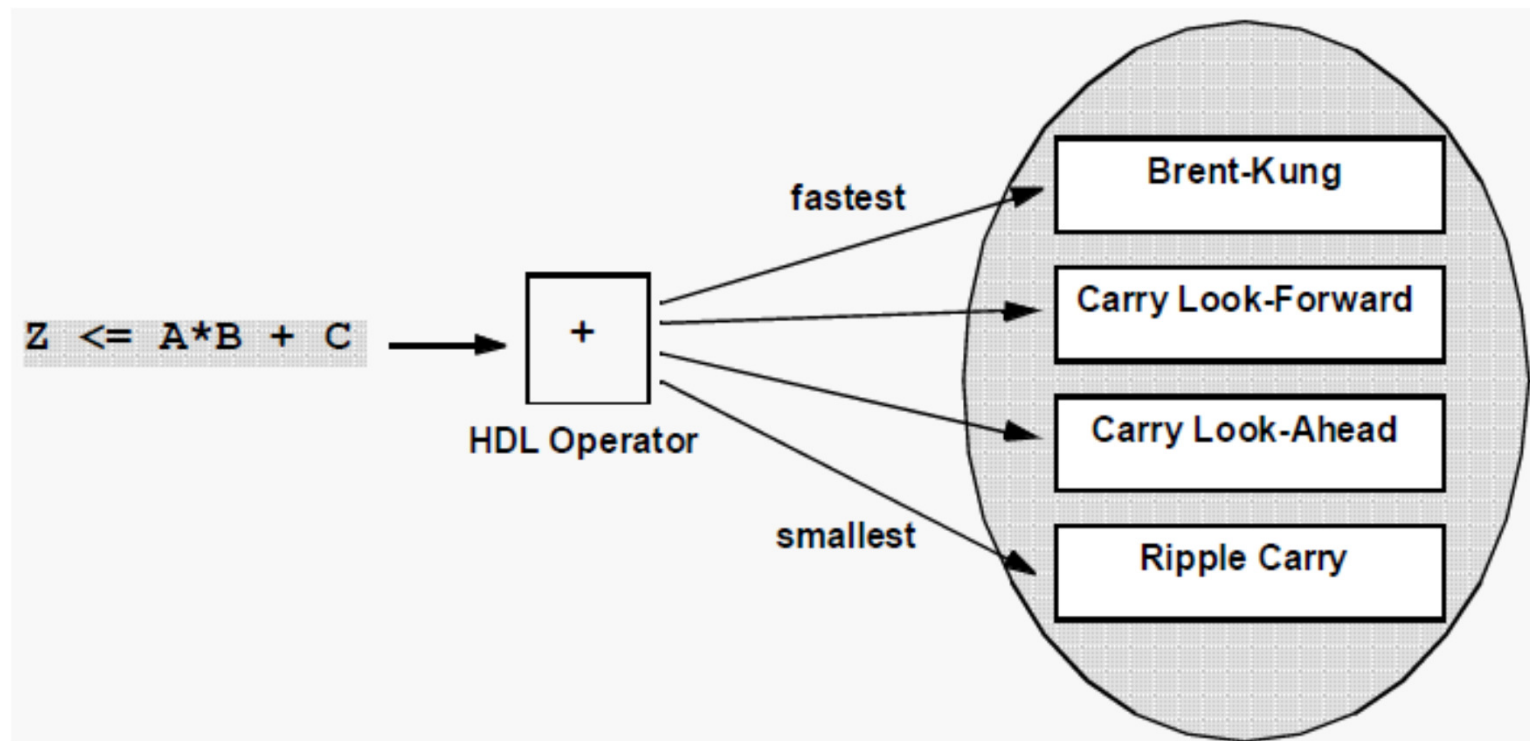


Another, more efficient implementation.



High Level Optimization

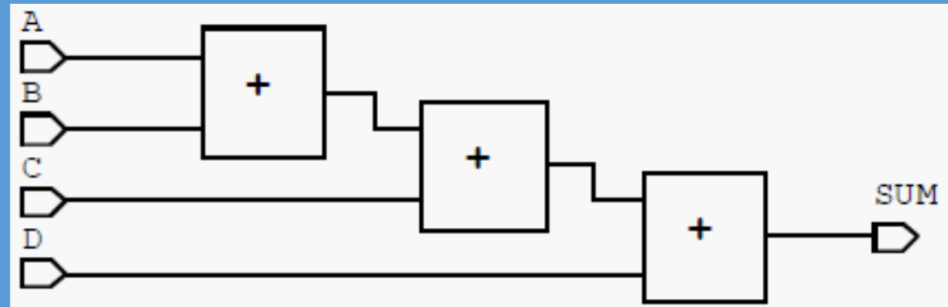
Optimization through Implementation Selection: Different implementations of the DesignWare components have different area and timing characteristics. Design constraints determine the appropriate DesignWare Component.



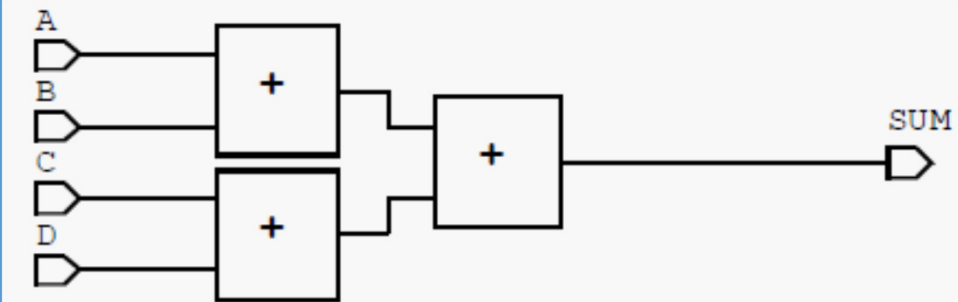
High Level Optimization

Arithmetic Optimization: Using Operator re-ordering (Architectural Level)

Example: $SUM = A + B + C + D$
Initial Order
- from left to right



Example: $SUM = A + B + C + D$
Optimized for Speed
- same delay for all inputs

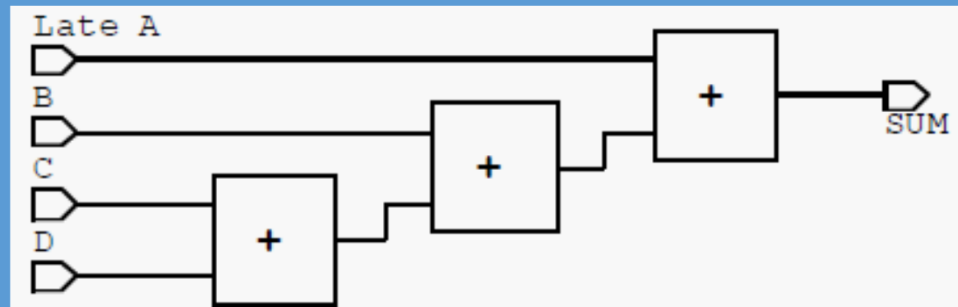


High Level Optimization

Arithmetic Optimization: Using Operator re-ordering (Architectural Level)

Example: $SUM = A + B + C + D$

Optimized for Speed
- re-ordering due to large
input delay for A



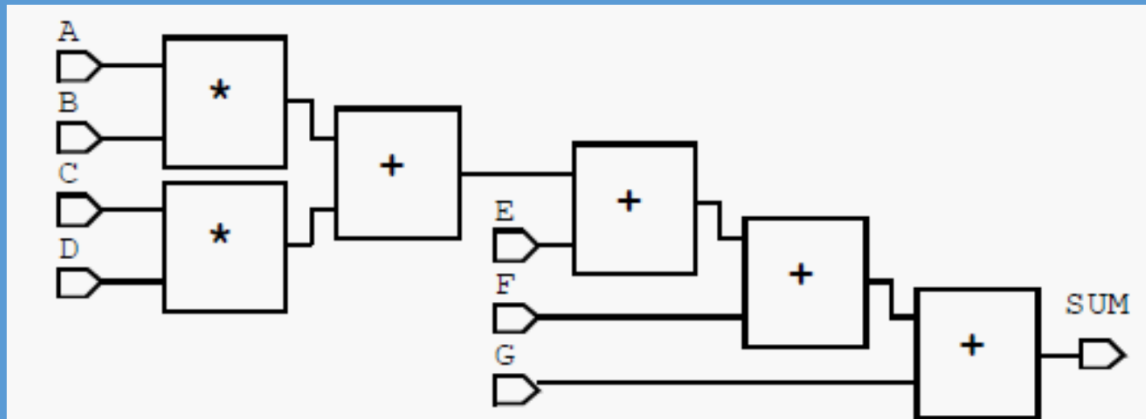
Note: Operators can not be re-arranged if initial order is overridden by use of parenthesis in HDL.

High Level Optimization

Arithmetic Optimization: Forced Operator order (Architectural Level)

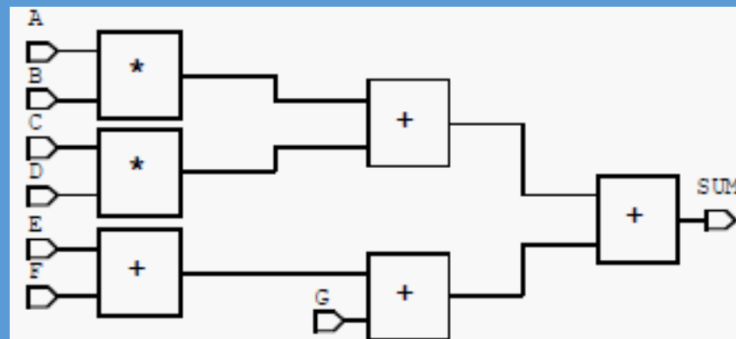
Example: $SUM \leq A * B + C * D + E + F + G$

The compiler parses expressions from left to right by default



$SUM \leq (A * B + C * D) + ((E + F) + G)$

The use of parenthesis will override this order
- the reordering of operators is now constrained.



High Level Optimization

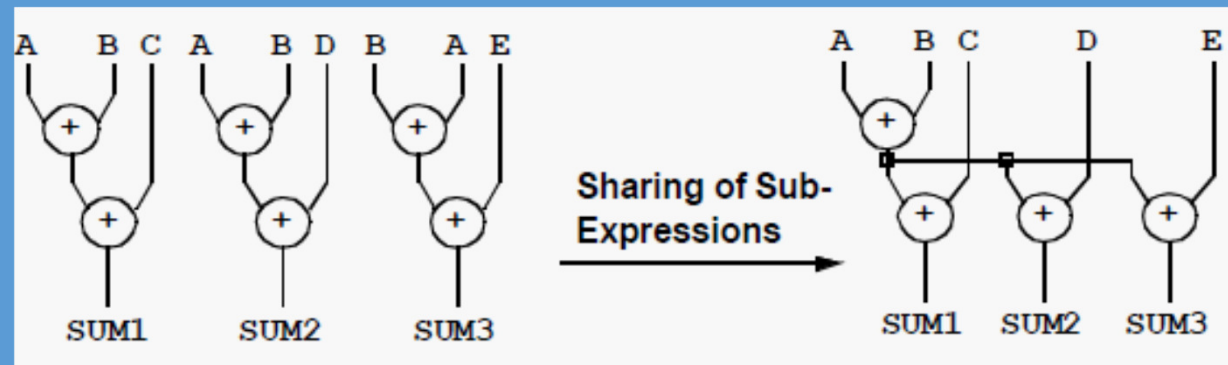
Arithmetic Optimization: Common Sub expression sharing

SUM1 $\leq A + B + C$

SUM2 $\leq A + B + D$

SUM3 $\leq B + A + E$

Order of within the Sub-Expressions is not important, but the positions must be the Same.



Logic Level Optimization

Logic Level Optimization:

- Operate with Boolean representation of a circuit
- Has a *global effect* on the overall area/speed characteristic of a design
- Strategy:
 - Structure
 - Flatten
 - If both are true, the design is first flattened and then structured

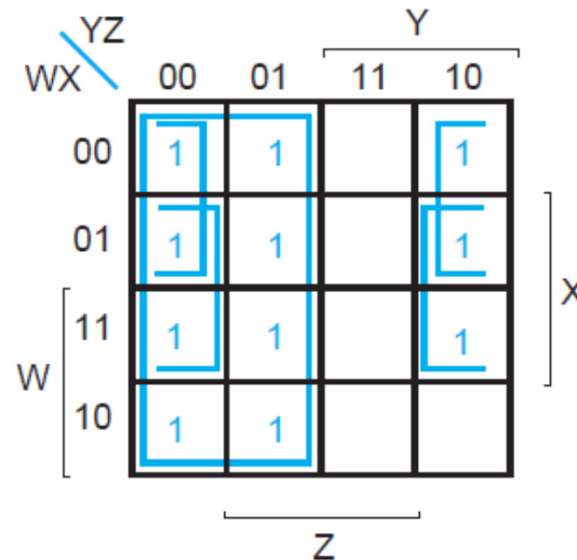
Structuring:

- *Adds intermediate variables and thus logic structure* (Factors out common sub-expression as intermediate variable)
- *Sharing expressions -> area efficiency (negative effect on delay)*
- The default logic-level optimization strategy; suitable for structured circuits (e.g. adders and ALU's).
- *Timing Driven Structuring(TDS): The default Synopsys optimization strategy takes delay constraints into account while structuring. Tries to optimize the critical path, by flattening it and Adds structure to less critical paths. It is very important to accurately constraint designs, that will result in optimized design* (meet design goals based on speed / area optimization).
- *Boolean Optimization: Use Boolean algebra rules to optimize area at the expense of delay. Boolean Optimization and TDS should not be ON at the same time. Boolean Optimization creates deep logic, which TDS cannot undo.*

Logic Level Minimization using K-Map

Logic Minimization using K-Map: K-Map is used to minimize the Boolean expression so final expression will have minimum number of literals and product terms that will produce an optimized circuit. Lets consider the example below.

Simplify the Boolean function $F(W, X, Y, Z) = \sum(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$.



$$F = \bar{Y} + \bar{W}\bar{Z} + X\bar{Z}$$

The map method of simplification is convenient if # of variables ≤ 4 . Tabulation method(Quin McClusky) is preferred for function with large # of variables.

Logic Level Minimization using Quine-McCluskey method

Example: $f(a,b,c,d) = \text{Sum } m(0,1,2,6,8,9,7,14)$

Step1: Find Prime Implicants:

Step1: Group according to no. of 1's				Step2: Compare adjacent groups				Step3 Continue comparison process			
Group0	0	0000	✓	0,1	000-	✓		0,1,8,9	-00-		
		-----		0,2	00-0			0,8,1,9	-00-		
Group1	1	0001	✓	0,8	-000	✓					
	2	0010	✓		-----						
	8	1000	✓	1,9	-001	✓					
Group2	6	0110	✓	2,6	0-10						
	9	1001	✓	8,9	100-	✓					
		-----		6,7	011-						
Group3	7	0111	✓	6,14	-110						
	14	1110	✓								

Note: Unchecked items form Prime implicants. They are $a'b'd' + a'cd' + a'bc + bcd' + b'c'$

Logic Level Minimization using Quine-McCluskey method

Step2: Find Essential Prime Implicants:

	0	1	2	6	7	8	9	14
0,1,8,9 ($b'c'$)	x	x				x	x	
0,2 ($a'b'd'$)	x	x						
2,6 ($a'cd'$)			x	x				
6,7 ($a'bc$)				x	x			
6,14 (bcd')				x				x

Note: If a minterm is covered by only one prime implicant that prime implicant is called an essential prime implicant.

Result is: $f(a,b,c,d) = b'c' + a'cd' + a'bc + bcd'$

Logic Level Optimization

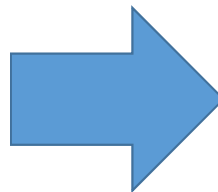
Logic Level Structuring Example: This example shows the logic before structuring and after structuring.

//Before Structuring

$f1 = acd + bcd + e$

$g1 = ae' + be'$

$h1 = cde$



//After Structuring

/*factoring out sub-expression as intermediate variable*/

$x = a + b$ and $y = cd$

$f1 = xy + e$

// $(acd + bcd + e = cd(a + b) + e) = xy + e$

$g1 = xe'$

// $ae' + be' = e'(a + b) = xe'$

$h1 = ye$

// $cde = ye$

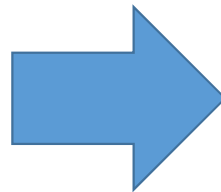
Logic Level Optimization

Flattening:

- Remove all intermediate variable(Hierarchy is preserved)
- Goal is to create a 2 level S.O.P. (2 level S.O.P does not always mean 2 level hardware (Speed is the motive) due to Library limitations.
- Flatten is default OFF; Use when you have a timing goal and have don't cares(x) in your HDL code

//Before Flattening

```
f0 = at  
f1 = d + t  
f2 = t'e  
t = b + c
```



//After Flattening

```
f0 = ab + ac  
f1 = b + c + d  
f2 = b'c'e
```

Gate Level Optimization(Technology dependent)

Gate Level Optimization:

- Select components to meet timing, design rule & area goals specified for the circuit
- Has a *local effect* on the area/speed characteristics of a design
- Strategy
 - **Mapping**
 - Combinational mapping
 - Sequential mapping

Combination circuit Vs Sequential Circuit Mapping:

Combinational Mapping:

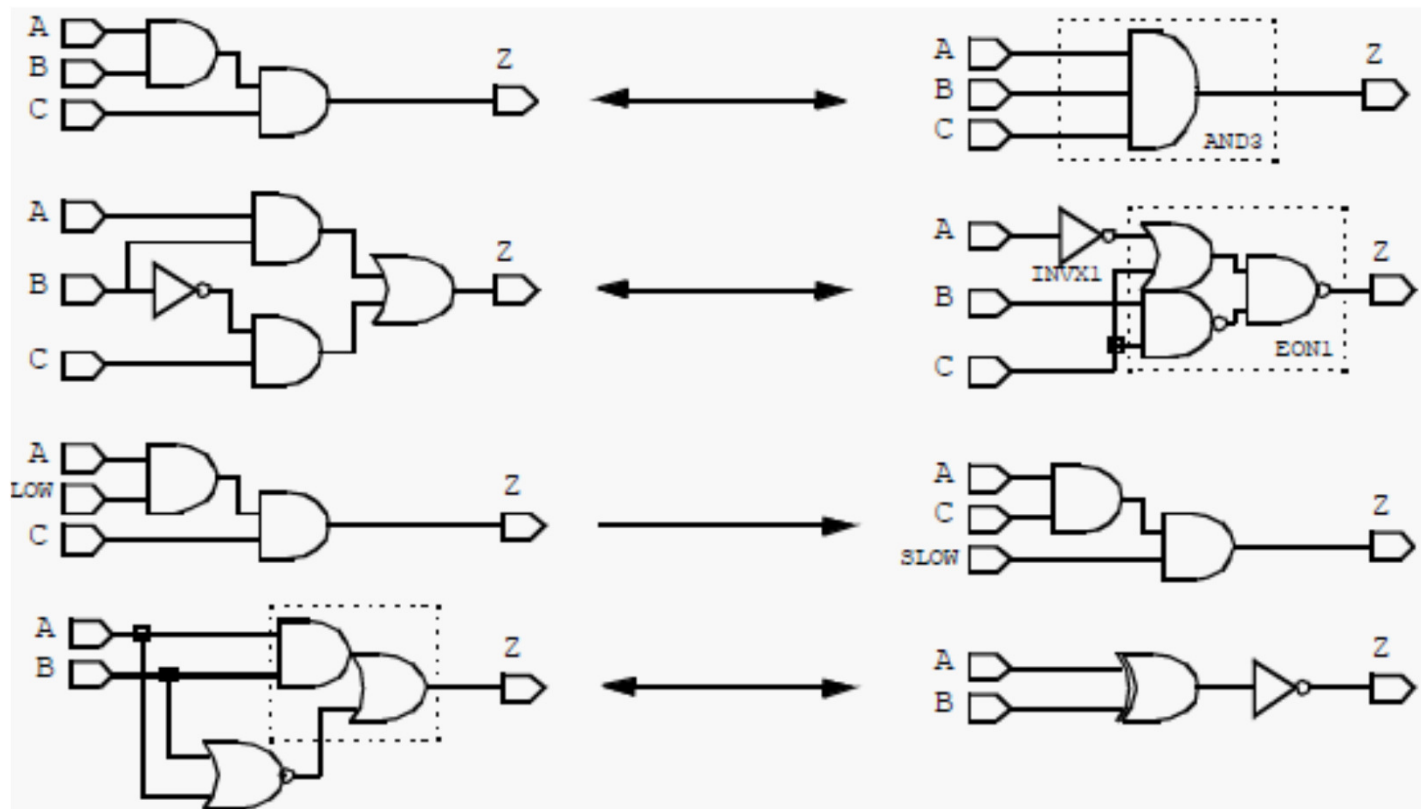
- Mapping rearranges components, combining and re-combining logic into different components
- May use different algorithms such as cloning, resizing or buffering
- Try to meet the design rule constraints and timing/area goals

Sequential Mapping:

- Optimize the mapping to sequential cells from technology library
- Analyze combinational circuit surrounding a sequential cell to see if it can absorb the logic attribute with HDL
- Try to save speed and area by using a more complex sequential cell

Gate Level Optimization: Combinational Mapping

Combinational Mapping: Maps the combinational parts of the design to the current technology library to meet design goals



Technology mapping + Optimization

- A **cover** is a collection of pattern graphs such that:
 - Every node of the subject graph is **contained** in one (or more) pattern graphs
 - Each **input** required by a pattern graph is an **output** of some other graph (i.e. the inputs of one gate come as outputs of other gates.)
- For minimum area, the cost of the cover is the **sum of the areas** of the gates in the cover

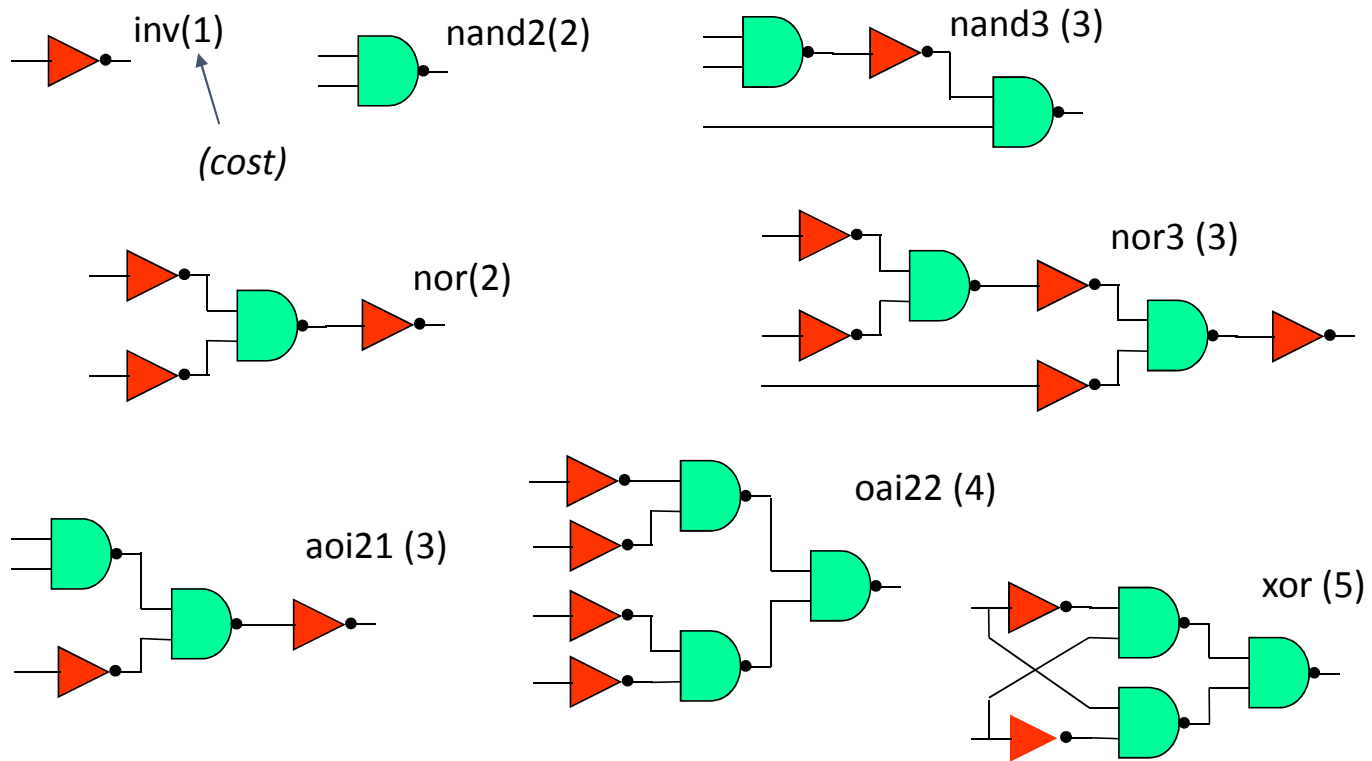
Technology mapping problem:

- Find a **minimum cost covering** of the **subject graph** by choosing from the collection of **pattern graphs** (from cell library).

Two phases of technology mapping:

- Pattern **matching** (find all ways a library pattern covers the nodes in the subject graph)
- Tree **covering**

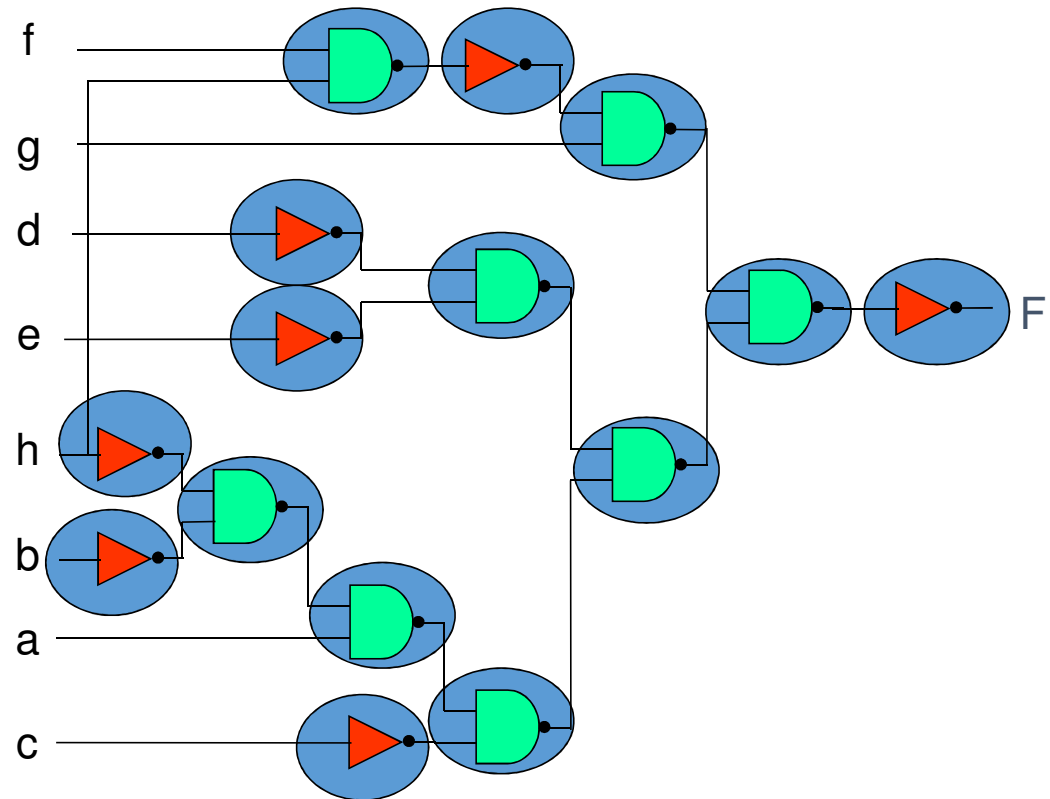
Pattern Graphs for the IWLS Library



Subject graph covering

$$\begin{aligned} t_1 &= d + e; \\ t_2 &= b + h; \\ t_3 &= at_2 + c; \\ t_4 &= t_1t_3 + fgh; \\ F &= t_4'; \end{aligned}$$

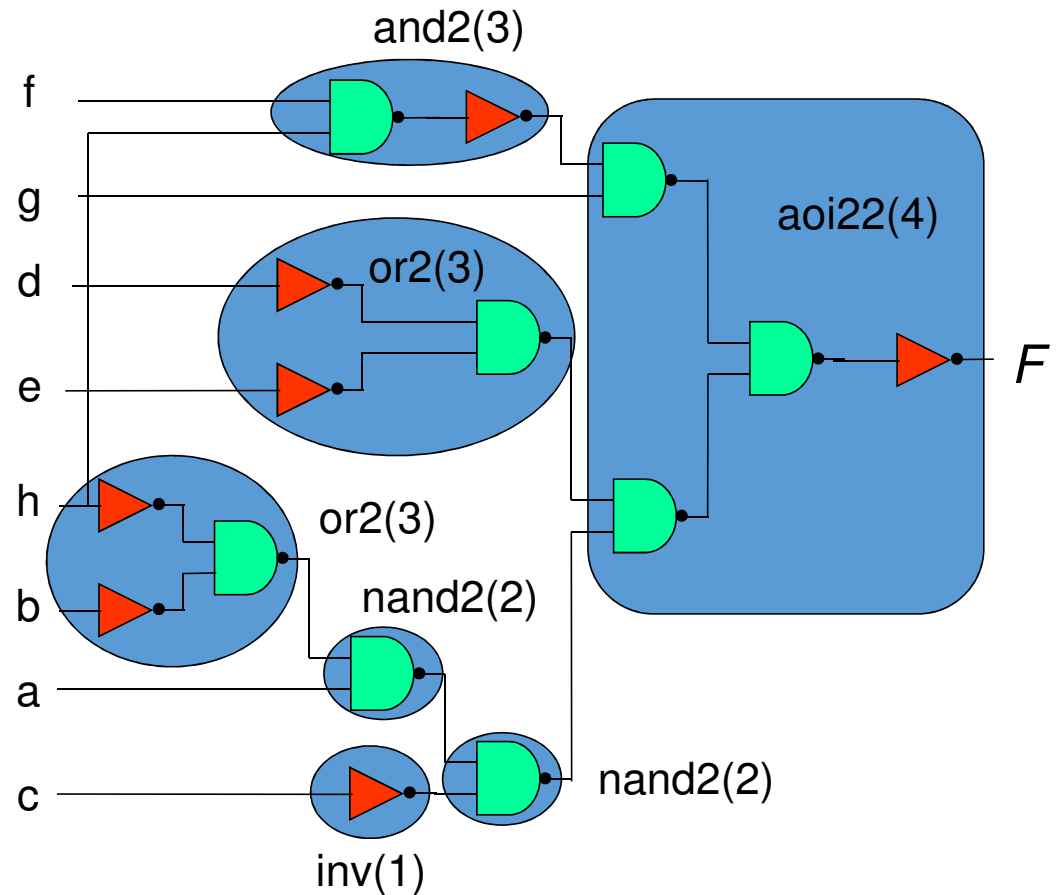
Total cost = 23



Better Covering

$$\begin{aligned}
 t_1 &= d + e; \\
 t_2 &= b + h; \\
 t_3 &= at_2 + c; \\
 t_4 &= t_1t_3 + fgh; \\
 F &= t_4';
 \end{aligned}$$

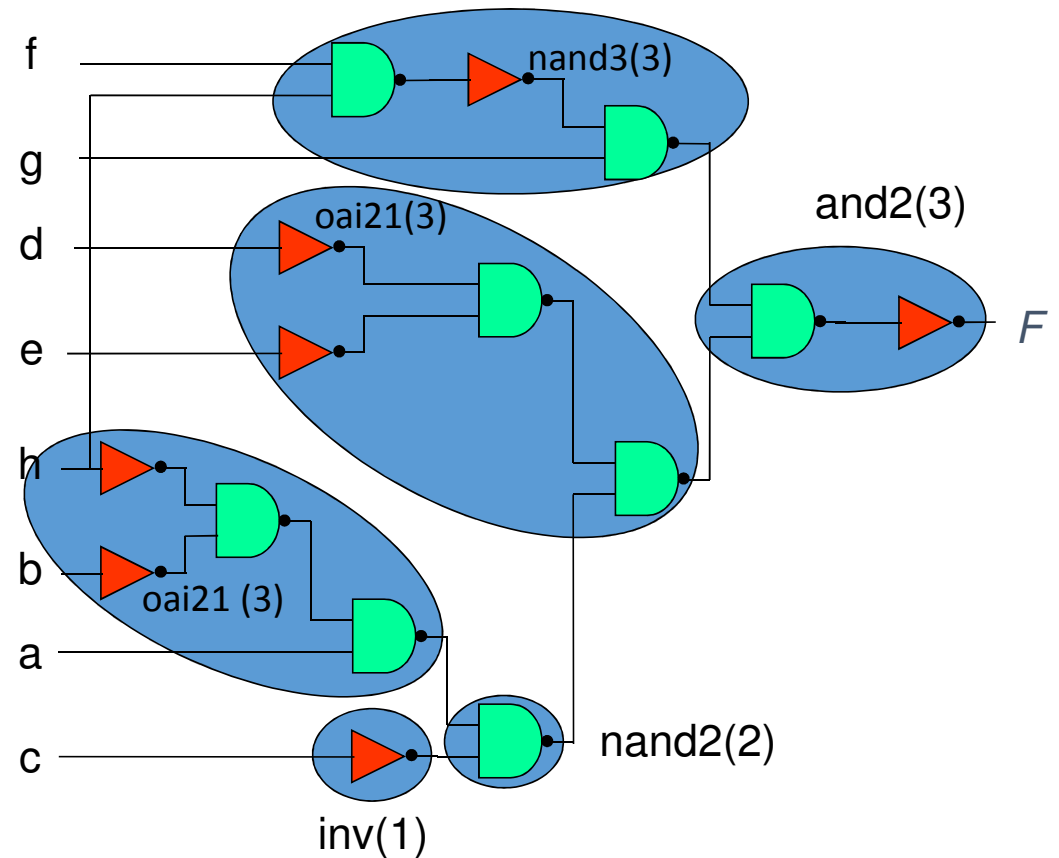
Total cost = 18



Alternate Covering

$$\begin{aligned}
 t_1 &= d + e; \\
 t_2 &= b + h; \\
 t_3 &= at_2 + c; \\
 t_4 &= t_1t_3 + fgh; \\
 F &= t_4';
 \end{aligned}$$

Total area = 15



Technology mapping using DAG Covering

Mapping Via DAG Covering: A *directed acyclic graph* (DAG!) is a directed graph that contains no cycles. A DAG may be used to represent common subexpressions in an optimizing compiler.

Input:

- Technology independent, optimized *logic network*
- Description of the gates in the *library* with their cost






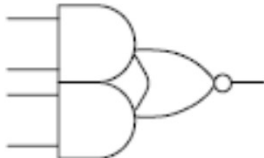
Output:

- *Netlist of gates* (from library) which minimizes total cost

General Approach:

- Construct a subject DAG for the network
- Represent each gate in the target library by pattern of DAG's
- Find an optimal-cost covering of subject DAG using the collection of pattern DAG's

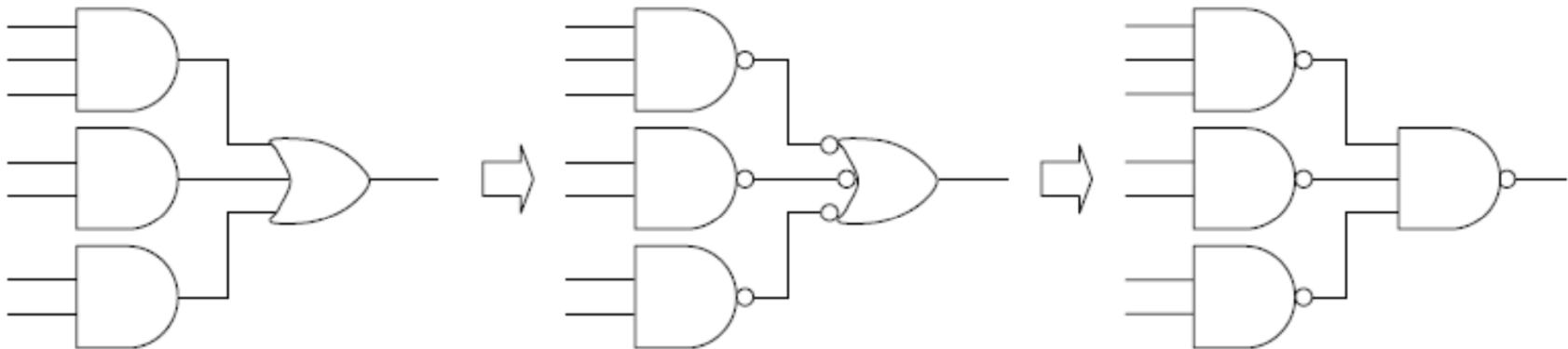
Technology Cell Library Example

Cell name	cost	symbol	Cell name	cost	symbol
INV	2		NAND4	5	
NAND2	3		AOI21	4	
NAND3	4		AOI22	5	

Area Optimal Technology Flow

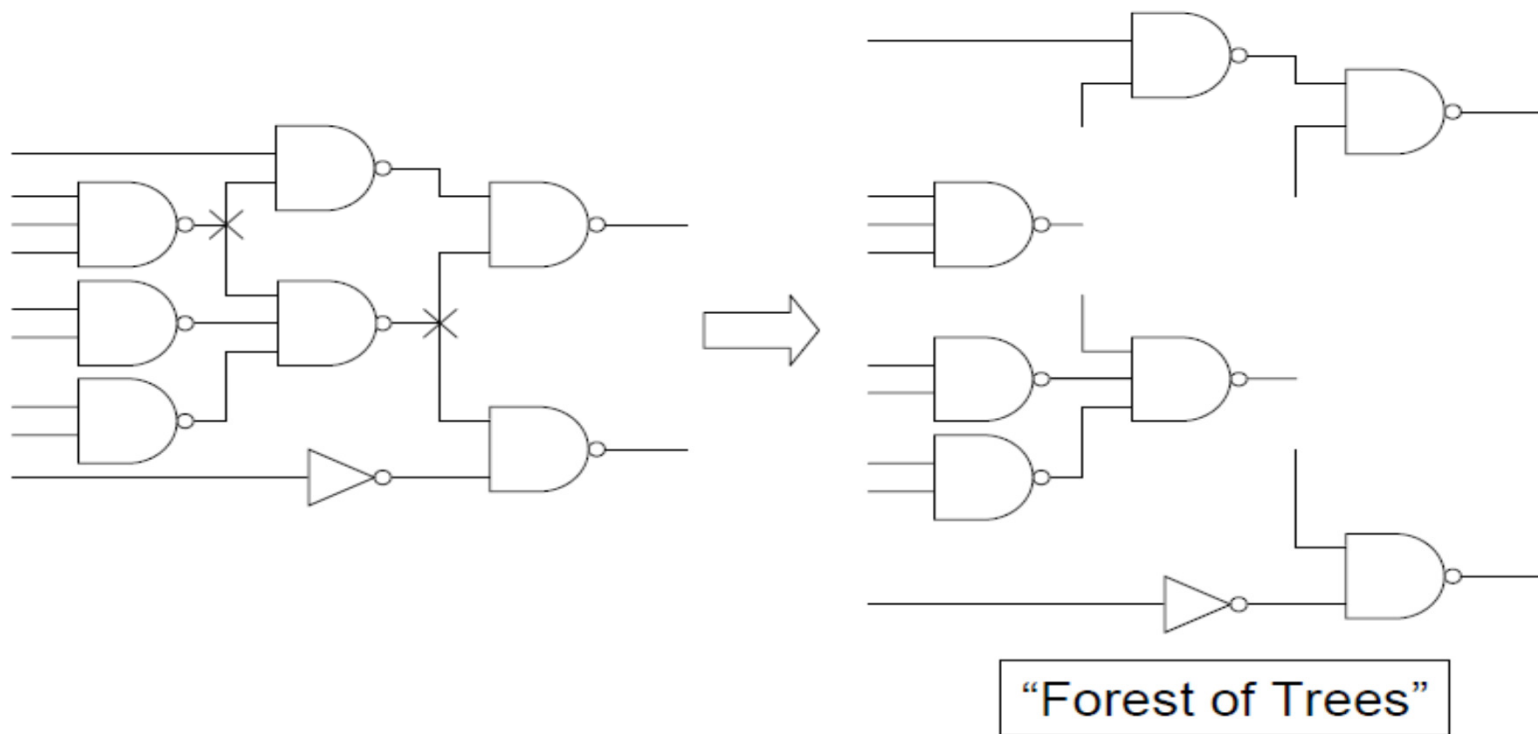
1. Transform the optimized Boolean Network into NAND network
2. Decompose NAND network into trees
 - Fan-out : # of destination pins for output pin of a node
 - Tree : DAG (directed acyclic graph) where all nodes have a fan-out of 1 ->Fast algorithms exist for solving the Optimal Tree Covering Problem
3. Transform each NAND-tree into NAND2-tree
 - Balanced NAND2 decomposition
4. For each NAND2-tree, obtain the optimal tree covering in terms of circuit area by *dynamic programming*
5. At each node of Boolean Network, covert the sum-of-product form into NAND-NAND form.

Example: $F = abc + de + fg = (abc) + (de) + (fg) = (abc)(de)(fg)$



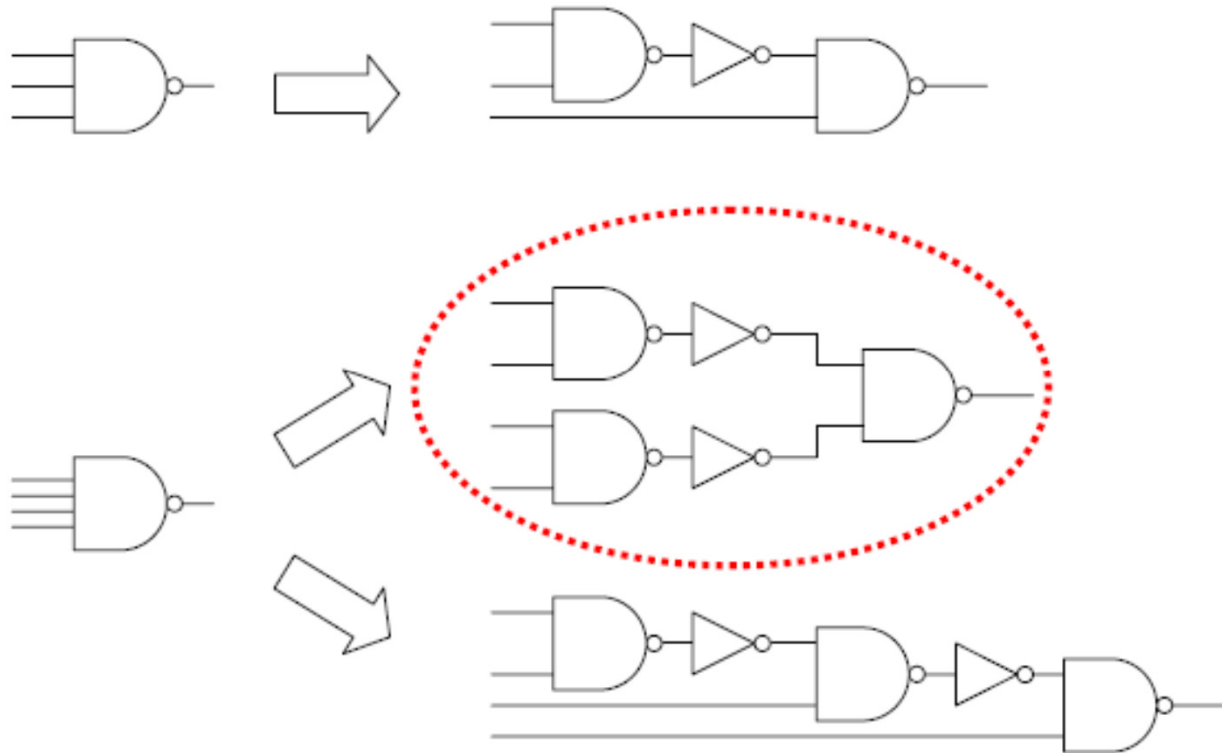
DAG-to-Tree Decomposition

If the gate output has a fan-out of more than 1, disconnect all pins from the arc (net).



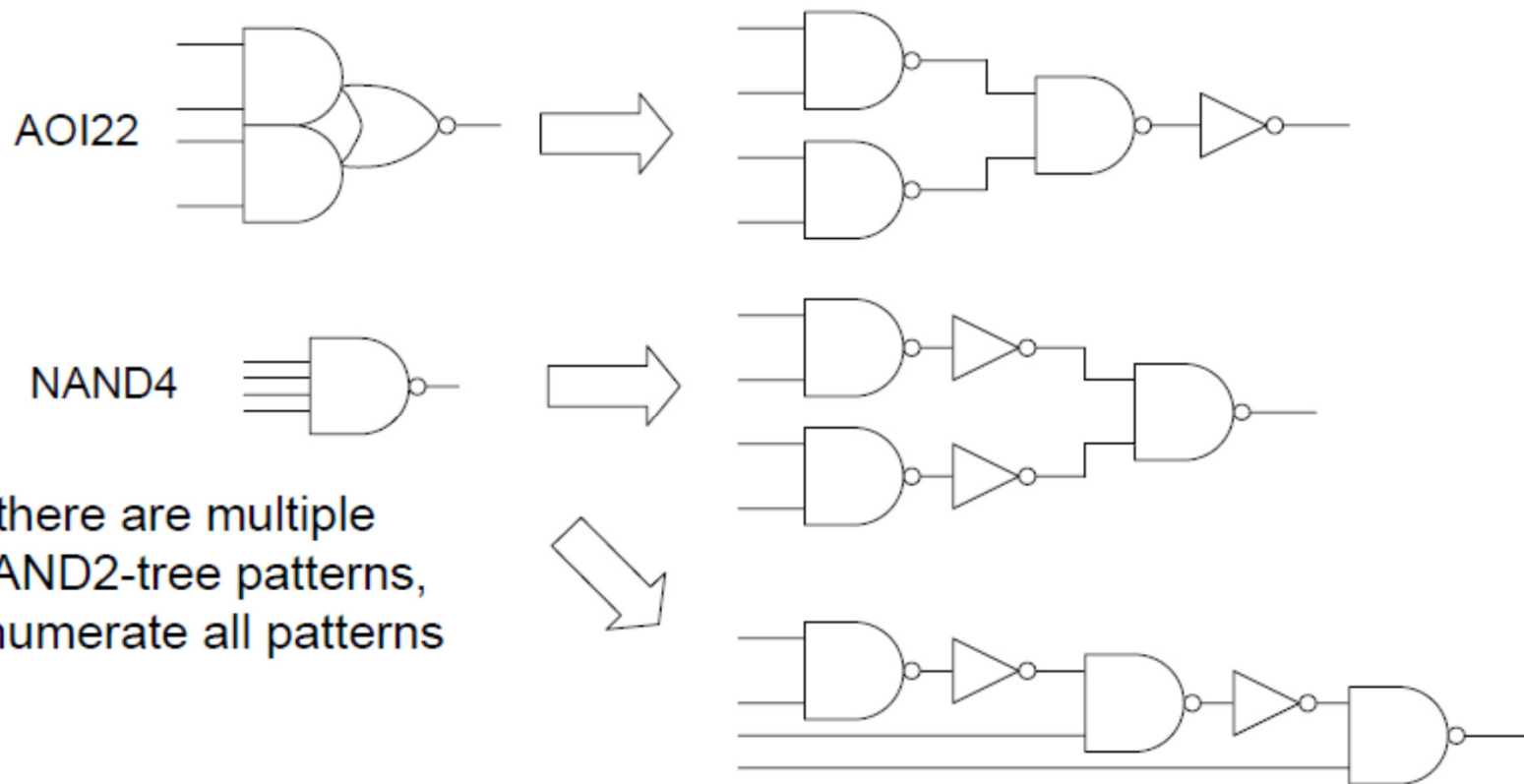
NAND2 Decomposition(NAND2-Tree)

- For each NAND gate on the NAND-tree, decompose into NAND2 gates
- If there are multiple decomposition solutions, choose the tree with the smallest height (balanced tree decomposition)






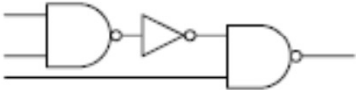

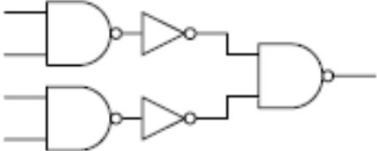
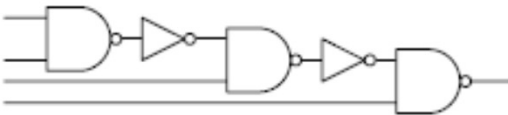

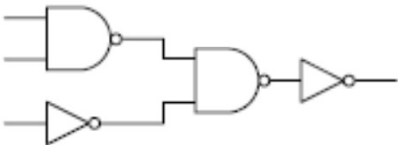

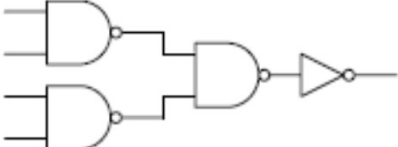


Cell Patterns

For each cell in the library, enumerate all functionally equivalent NAND2-trees and register them as *cell patterns*.

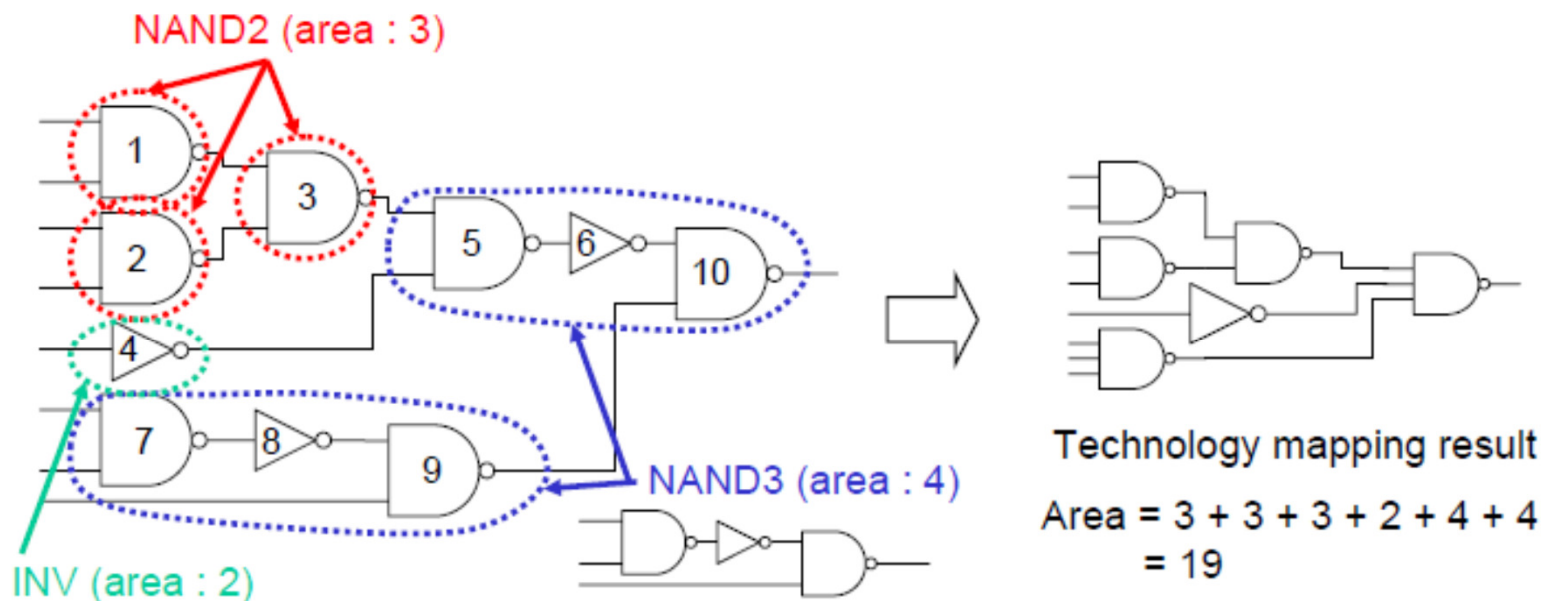


Cell Library Example

Cell name	cost	symbol	Primitive DAG (NAND2+INV representation)
INV	2		
NAND2	3		
NAND3	4		
NAND4	5		 
AOI21	4		
AOI22	5		

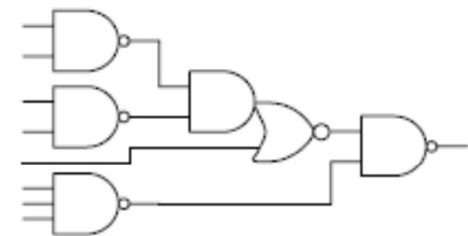
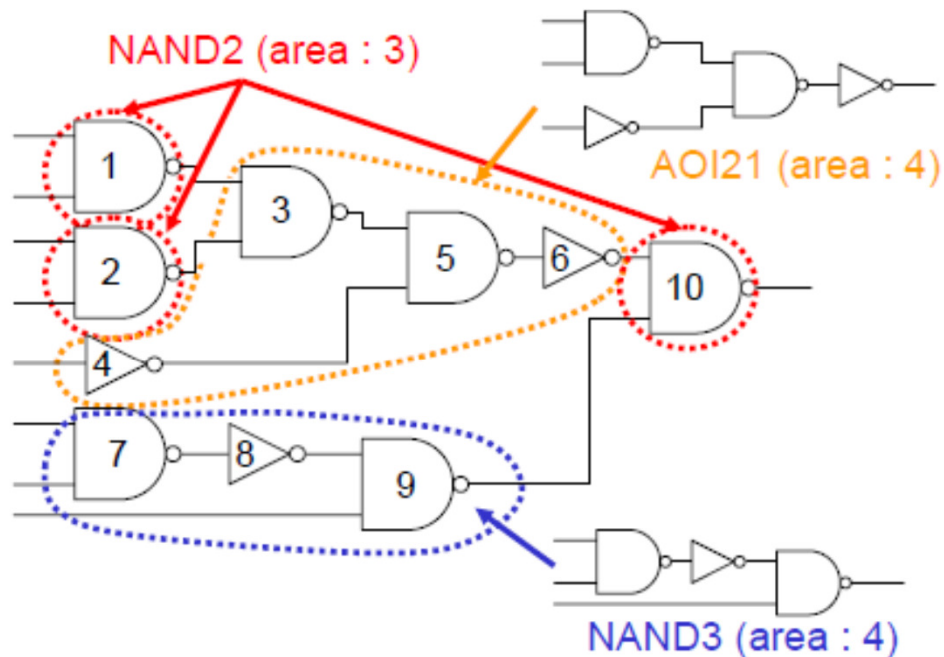
Technology Mapping as Tree Covering Problem (1)

- Cover the NAND2-tree with registered cell patterns with the minimum cost (circuit area, speed, etc.)
- Each node must be covered by exactly one pattern



Technology Mapping as Tree Covering Problem (2)

- Cover the NAND2-tree with registered cell patterns with the minimum cost (circuit area, speed, etc.)
- Each node must be covered by exactly one pattern

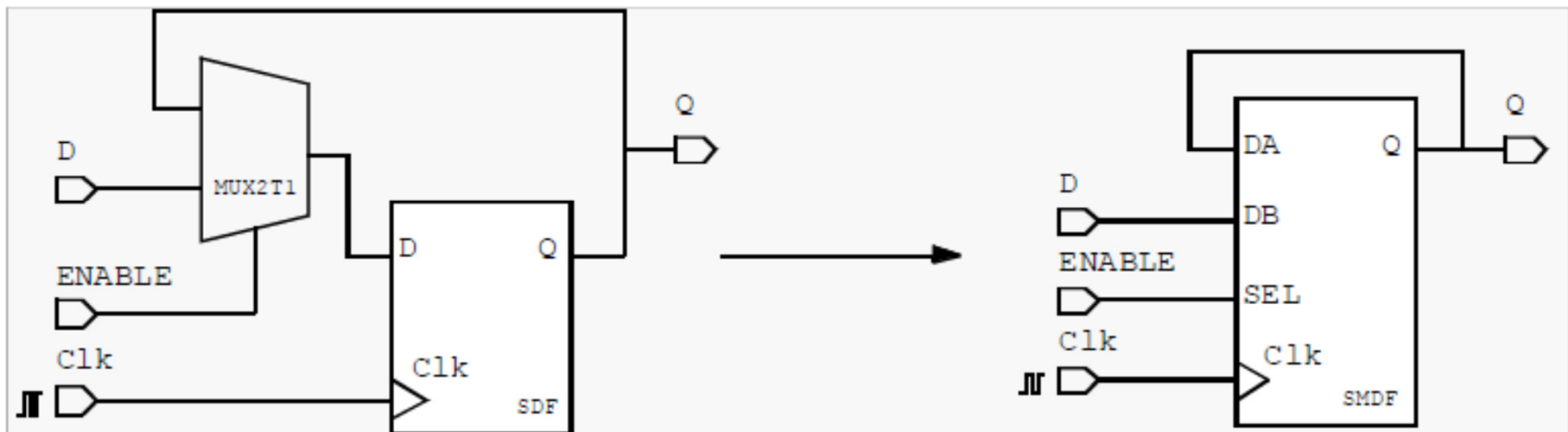


Technology mapping result

$$\begin{aligned} \text{Area} &= 3 + 3 + 4 + 4 + 3 \\ &= 17 \end{aligned}$$

Gate Level – Sequential Mapping

- Maps the sequential parts of the design to the current technology library to meet design goals
- To save speed and Area by using more complex sequential cells)



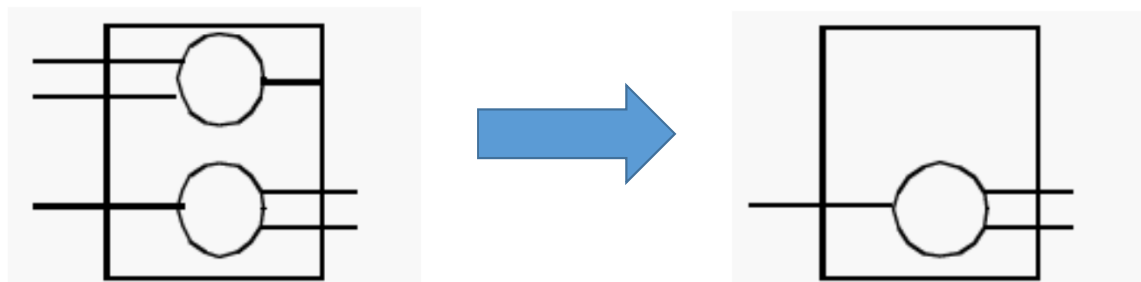
In Place and Boundary Optimization

In Place Optimization:

- Pre-layout fan-out based estimates of net length, resistance and capacitance could differ from post-layout numbers.
- To change mapping to take these differences into account use in place optimization.

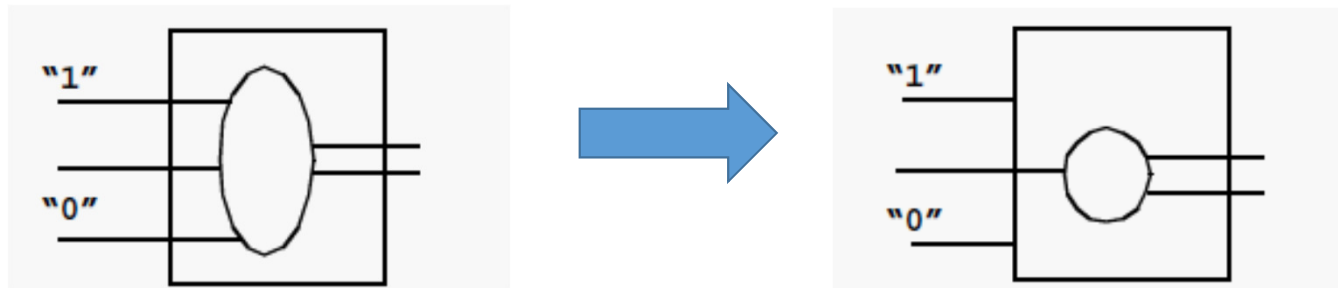
Boundary Optimization:

- Removal of un-connected logic across boundaries

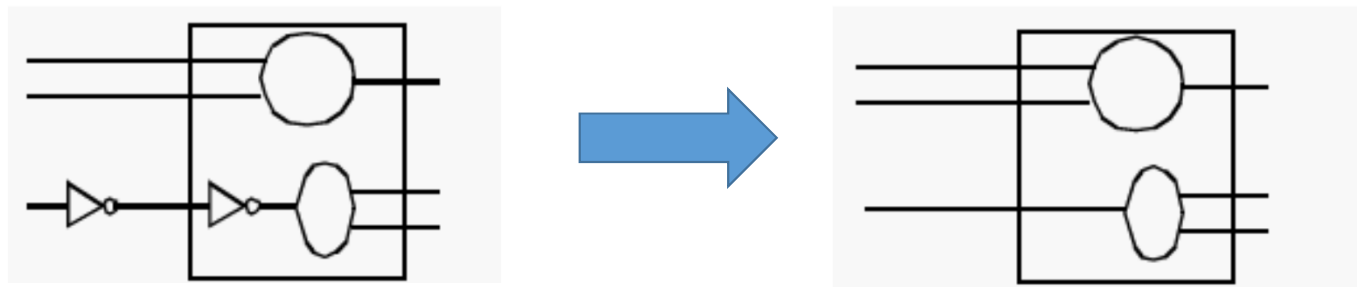


Boundary Optimization

- **Propagation of constants to reduce logic**



- **Removal of double inverting logic across boundaries**



Coding For Synthesis(Continue...)

Avoid Combinational Feedback Loop:

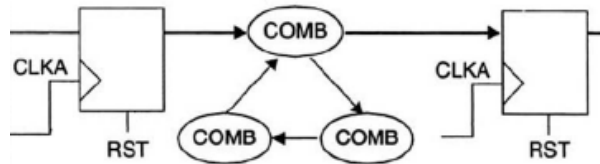


```
always @ (a or x)
  if (a)
    x = x + 1'b1;
  else
    x = x;
```



```
always @ (posedge clk)
  temp_x <= x;

always @ (a or temp_x)
  if (a)
    x = temp_x + 1'b1;
  else
    x = temp_x;
```



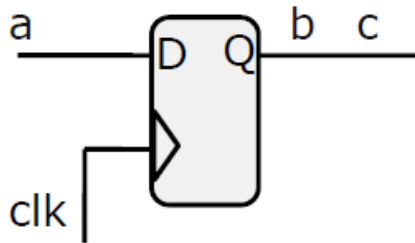
Coding For Synthesis(Continue...)

Use Non-blocking for sequential Logic Coding:



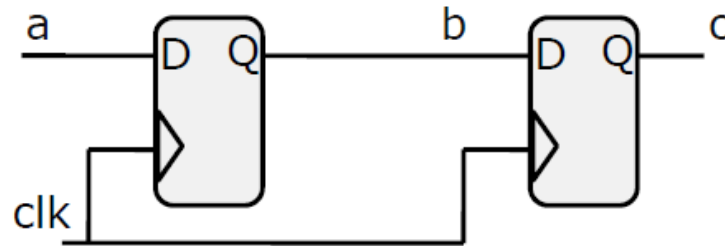
```
always @ (posedge clk )  
begin  
    b = a;  
    c = b;  
end
```

Just like "a=c;"



```
always @ (posedge clk )  
begin  
    b <= a;  
    c <= b;  
end
```

Just like "shift register"



Coding For Synthesis(Continue...)

Sensitivity List should be Complete for combinational Block: For combinational logic blocks, the sensitivity list must include every signal that is read by the Process.

- Signals that appear on the right side of an assign statement
- Signals that appear in a conditional expression



```
// c and enable missing from list
always @ (a or b)
begin
    if(enable)
        sum = a + b;
    else
        sum = b + c;
end
```



```
/*Just use always@(*) for
combinational */
always @ (a or b or c or enable)
begin
    if(enable)
        sum = a + b;
    else
        sum = b + c;
end
```

Note: Incomplete sensitivity list may cause behavior of pre-synthesis design to be different from that of the post-synthesis netlist.

Coding For Synthesis(Continue...)

Sensitivity List for sequential block: For sequential logic block, the sensitive list must include :

- Clock signal (posedge clk or negedge clk).
- If an asynchronous reset signal is used, include reset in the sensitivity list.



//nothing on sensitivity list

```
always @ ( )  
  if(!rst_n)  
    q <= 1'b0;  
  else  
    d <= d;
```



/* Flip flop with Asynchronous active
low rst_n */

```
always @ (posedge clk or negedge  
rst_n)  
  if(!rst_n)  
    q <= 1'b0;  
  else  
    d <= d;
```

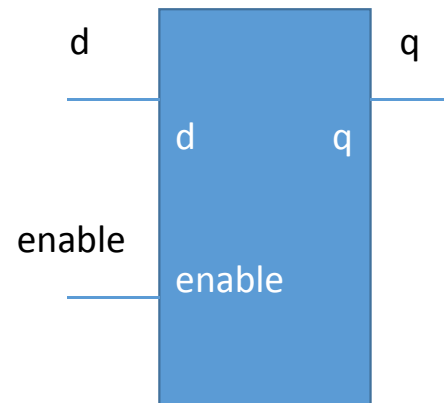
Coding For Synthesis(Continue...)

When Synthesis tool Infer Latch(Avoid latches and flip flop should be used instead) :

In D-Latch example below, If q were simply a combinational function of d, the synthesizer could create the appropriate combinational logic. But since there are times when the always block executes but q isn't assigned (e.g., when enable = 0), the synthesizer has to arrange to remember old value of q even if d is changing → it will infer the need for a storage element (latch, register). Sometimes this inference happens even when you don't mean it to. You have to be careful to always ensure an assignment happens each time through the block if you don't want storage elements to appear in your design.

```
// D-Latch
```

```
reg q;  
always@(enable or d)  
  if(enable)  
    q <= d;
```



Coding For Synthesis(Continue...)

How to Avoid unwanted Latches?

Rule No.1: If the procedure has several paths, every path must evaluate all outputs.

Method1:

//Set all outputs to some value at the start of the procedure. Later on
//different values can overwrite those values.

always @(*)

begin

x=0; y=0; z=0; // all outputs set to 0 at start

if (a)

x=2;

else if (b)

y=3;

else

z=4;

end

Coding For Synthesis(Continue...)

How to Avoid unwanted Latches:

Method2:

// Be sure every branch of every if and case generate every output

always @(*)

begin

if (a)

begin

x=2; y=0; z=0; // every output is assigned

end

else if (b)

begin

x=0; y=3; z=0; // every output is assigned

end

else

begin

x=0; y=0; z=4; // every output is assigned

end

end

Coding For Synthesis(Continue...)

Rule No.2: All inputs used in the procedure must appear in the trigger list(sensitivity list).

Right-hand side variables:

//Except variables both calculated and used in the procedure.

```
always @(a or b or c or x or y) // x and y not required
begin
    x = a; y=b; z=c;
    w = x + y;
end
```

Branch controlling variables:

//Be sure every branch of every if and case generate every output.

```
always @(a or b)
begin
    if (a)
        begin x=2; y=0; z=0; end
    else if (b) begin x=0; y=3; z=0; end
    else      begin x=0; y=0; z=4; end
end
```

Coding For Synthesis(Continue...)

Rule No.3: All possible inputs used control statements must be covered.

```
/*End all case statements with the default case whether you need it or not*/
case(state)
...
default: next_state = reset;
endcase

/* Do not forget the self loops in your state graph for each if include else*/
If(a | b&c)
    next_state = S1;
elseif(c&d)
    next_state = S2;
else
    next_state = reset;
```


Coding For Synthesis(Continue...)

Latch Inference example:



/ Unspecified Assignments in 'case' or 'if – else' Statements */*

```
if (...)  
begin  
    a <= 1;  
    b <= 0;  
end // latch inferred for the signal c  
  
else if (...)  
    a <= 0; // latch inferred for b and c  
  
else  
    c <= 1; // latch inferred for a and b
```



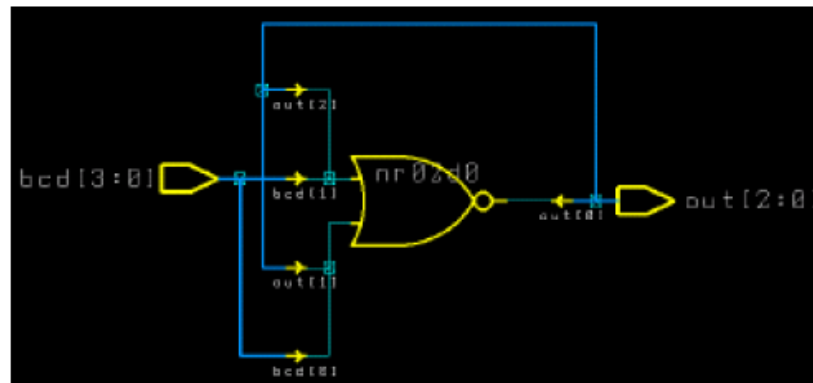
/ Always assign known values to all the signals in a case or if-else statements*

```
*/  
if (...)  
begin  
    a <= 1; b <= 0; c <= 0;  
end  
else if (...)  
begin  
    a <= 0; b <= 0; c <= 0;  
end  
else  
begin  
    a <= 0; b <= 0; c <= 1;  
end
```

Coding For Synthesis(Continue...)

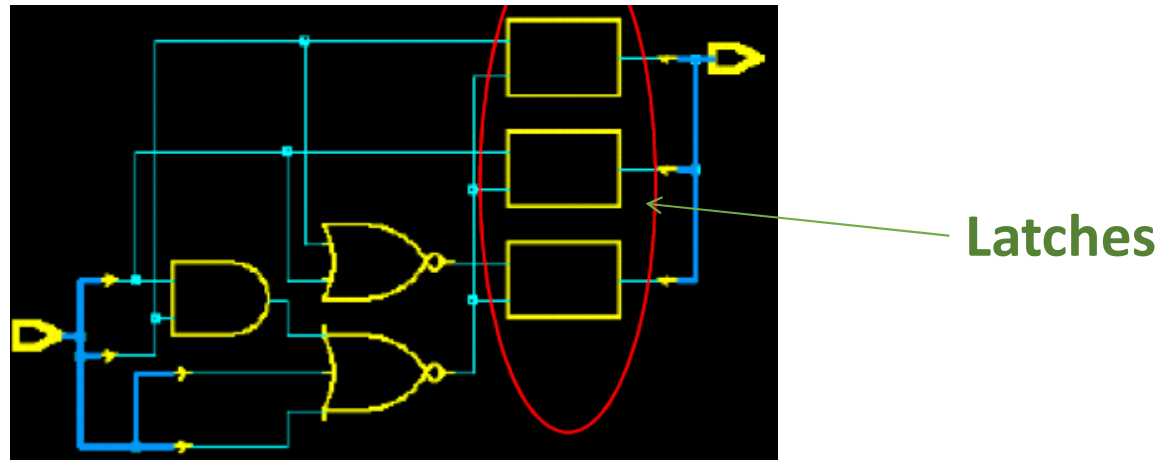
Case statement: A case statement is called a full case if all possible branches are specified.

```
// All possible branches of bcd are specified using default, no latch inference .  
//if case statement contains a default clause, HDL compiler assumes all conditions are covered.  
always @(bcd)  
  case (bcd)  
    4'd0:   out=3'b001;  
    4'd1:   out=3'b010;  
    4'd2:   out=3'b100;  
    default: out=3'bxxx;  
  endcase
```



Coding For Synthesis(Continue...)

Case statement: If a case statement is not a full case, it will infer latches.

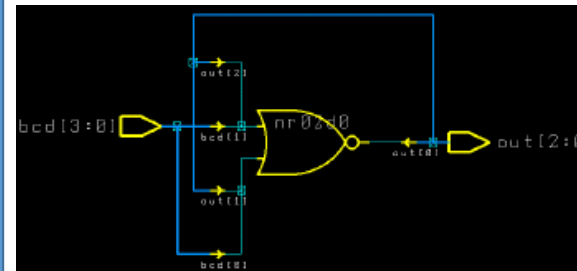


Coding For Synthesis(Continue...)

Synopsys full case Directive: If you do not specify all possible branches, but you know the other branches will never occur, you can use “//synopsys full_case” directive to specify full case. The //synopsys full_case directive asserts that all possible clauses of a case statement have been covered and that no default clause is necessary. This directive has two uses; it avoids the need for default logic, and it can avoid latch inference from a case statement by asserting that all necessary conditions are covered by the given branches of the case statement.

```
/* All possible branches of bcd” are  
covered through synopsys full case  
directive */
```

```
always @(bcd)  
  case (bcd) //synopsys full_case  
    4'd0:   out=3'b001;  
    4'd1:   out=3'b010;  
    4'd2:   out=3'b100;  
  endcase
```

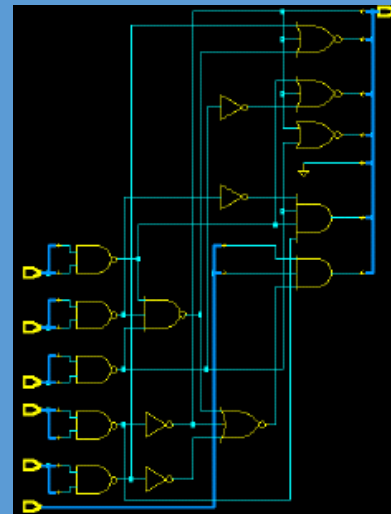


Coding For Synthesis(Continue...)

Synopsys Parallel case Directive: If HDL Compiler can't determine that case branches are parallel, its synthesized hardware will include a priority decoder.

/*This example shows a casez statement that is not parallel because if the 3-bit irq bus is 3'b011, 3'b101, 3'b110 or 3'b111, more than one case item could potentially match the irq value. This will simulate like a priority encoder where irq[2] has priority over irq[1], which has priority over irq[0]. This example will also infer a priority encoder when synthesized */

```
always @(irq)
begin
    {int2, int1, int0} = 3'b0;
    casez (irq)
        3'b1??: int2 = 1'b1;
        3'b?1?: int1 = 1'b1;
        3'b??1: int0 = 1'b1;
    endcase
end
```

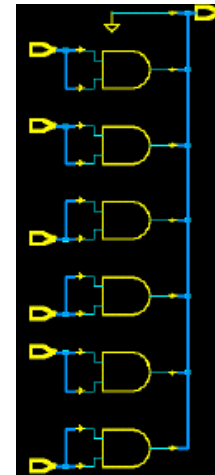


Coding For Synthesis(Continue...)

Synopsys Parallel case Directive: A "parallel" case statement is a case statement in which it is only possible to match a case expression to one and only one case item. If it is possible to find a case expression that would match more than one case item, the matching case items are called "overlapping" case items and the case statement is not parallel.

```
/* You can declare a case statement as parallel  
case with the "//synopsys_parallel_case"  
directive */
```

```
always @(irq)  
begin  
    {int2, int1, int0} = 3'b0;  
    casez (irq) //synopsys_parallel_case  
        3'b1??: int2 = 1'b1;  
        3'b?1?: int1 = 1'b1;  
        3'b???1: int0 = 1'b1;  
    endcase  
end
```

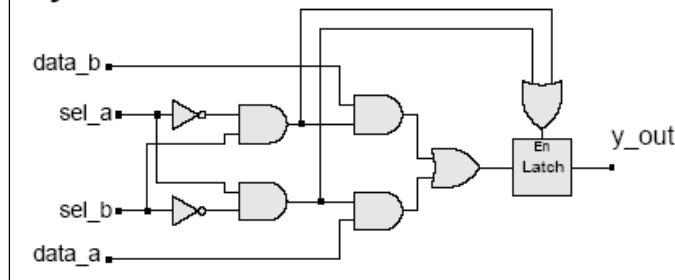


Coding For Synthesis(Continue...)

If ... else statement: Unintentional latches can also be generated resulting from an incomplete conditional branch.

```
/*all conditional branches of the if ... else statement is not specified  
resulting in latch inference , else branch condition missing */  
always@(sel_a or sel_b or data_a or data_b)  
  if ({sel_a, sel_b} == 2'b10)  
    y_out = data_a;  
  else if ({sel_a, sel_b} == 2'b01)  
    y_out = data_b;
```

Synthesis result:

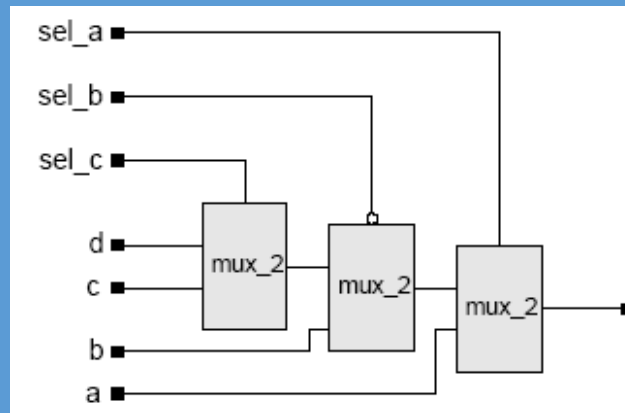


Coding For Synthesis(Continue...)

If ... else statement Priority Logic: When the branching of a conditional (if) is not mutually exclusive, or when the branches of a case statement are not mutually exclusive, the synthesis tool will create a priority structure.

/ Operation of Priority encoder is such that if two or more single bit inputs are at logic 1, then the input with highest priority will take precedence. */*

```
always @ (sel_a or sel_b or sel_c or a or b or c or d)
begin
  if (sel_a == 1)
    y = a;
  else if (sel_b == 0)
    y = b;
  else if (sel_c == 1)
    y = c;
  else
    y = d;
end
```



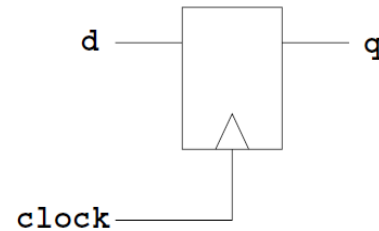
Coding For Synthesis(Continue...)

positive edge triggered D flip flop:

Verilog Code:

```
always @ (posedge clock)
```

```
    q <= d;
```

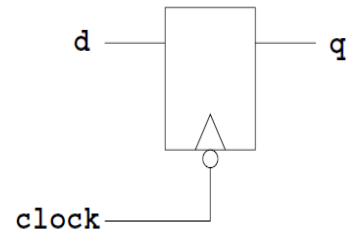


Negative edge triggered D flip flop:

Verilog Code:

```
always @ (negedge clock)
```

```
    q <= d;
```



Coding For Synthesis(Continue...)

D flip flop with Synchronous Reset:

Verilog Code:

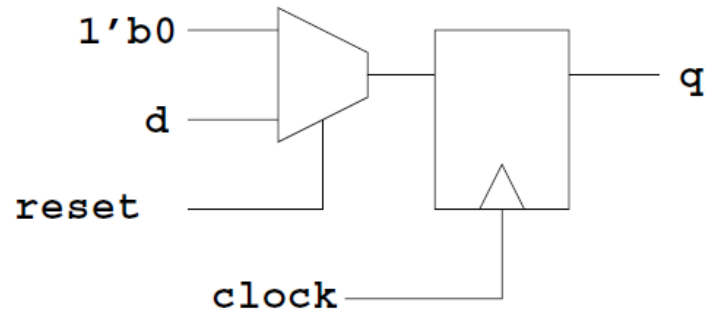
```
always @ (posedge clock)
```

```
if (reset) // active high reset
```

```
    q <= 1'b0;
```

```
else
```

```
    q <= d;
```



D flip flop with Asynchronous Reset:

Verilog Code:

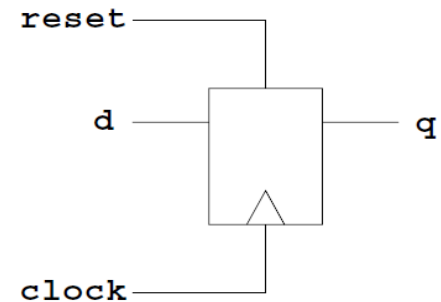
```
always @ (posedge clock or posedge reset)
```

```
if (reset) //active high reset
```

```
    q <= 1'b0;
```

```
else
```

```
    q <= d;
```



Coding For Synthesis(Continue...)

D flip flop with Synchronous Reset(Active low reset):

Verilog Code:

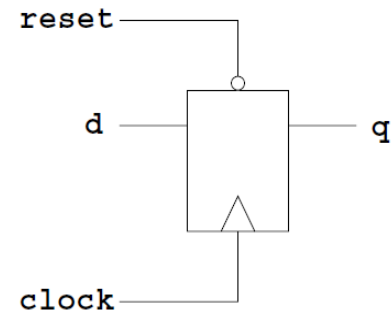
```
always @ (posedge clock or negedge reset)
```

```
if (~reset)
```

```
    q <= 1'b0;
```

```
else
```

```
    q <= d;
```



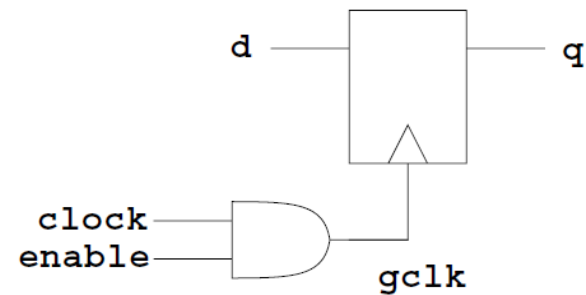
D flip flop with Gated clock:

Verilog Code:

```
wire gclk = (clock && enable);
```

```
always @ (posedge gclk)
```

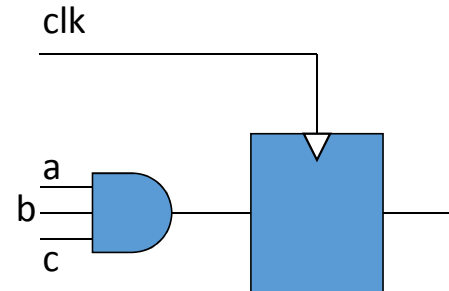
```
    q <= d;
```



Coding For Synthesis(Continue...)


Registered Combinational Logic: Combinational logic that is included in a synchronous behavior will be synthesized with registered output(Makes output drive strengths and input delays).

```
module reg_and ( y, a, b, c, clk );  
    input a, b, c, clk;  
    output y;  
    reg y;  
  
    always @ ( posedge clk )  
        y = a & b & c;  
endmodule
```




Coding For Synthesis(Continue...)

Signal assignment in multiple always block: Multiple always blocks may not assign values to the same signal in the same time step. This would result in the signal having an indeterminate value. HDL Compiler will provide warning “assignment in multiple always blocks” .



```
always @ (posedge clk)
begin
    a <= 1;
    b <= 0;
end
// multiple assignment to signal a
always @ (posedge clk or reset)
begin
    a <= 0;
    c <= 1;
end
```



```
always @ (posedge clk)
begin
    a <= 1;
    b <= 0;
end
//Only one behavioral block can have an
//assignment to a signal
always @ (posedge clk or reset)
begin
    c <= 1;
end
```

Coding For Synthesis(Continue...)

Assignment statements mis-ordered: If assignment statements are mis-ordered in a combinational always block, synthesis tools typically build the logic as if the statements had been ordered correctly but the pre-synthesis simulation will be wrong.



```
/* The following model is coded incorrectly to
model an and-or gate. The pre-synthesis
simulation will not correctly update the ored
output y after changes on the a and b inputs.
There will be a mis-match between pre-synthesis
and post-synthesis simulations using this model */
module AND_OR1 (output reg y, input a, b, c); reg
tmp;
always @*
    begin
        y = tmp | c; //OR logic
        tmp = a & b; // AND logic
    end
endmodule
```



```
/* The following model is coded correctly to model
an and-or gate where the a and b inputs are anded
together, and the result is ored with the c input.
There will be no mis-match between pre-synthesis
and post-synthesis simulations using this model */
```

```
module AND_OR2 (output reg y, input a, b, c);
reg tmp;
always @*
    begin
        tmp = a & b; // AND Logic
        y = tmp | c; //OR Logic
    end
endmodule
```

Coding For Synthesis(Continue...)

How Synthesis treats casex? The Verilog casex statement treats all z , x , and ? bits as don't cares, whether they appear in the case expression or in the case item being tested. In the following model, if the en (enable) goes unknown during simulation, the en_mem output will be driven high. In a synthesized gate-level model, the outputs would most likely go unknown indicating a design problem. This is a mis-match between pre-synthesis and post-synthesis simulations.

```
module endec_x (output reg en_mem, en_cpu, en_io, input [1:0] en);
always @*
begin
    en_mem=1'b0; en_cpu=1'b0; en_io =1'b0;
    casex (en)
        2'b1?: en_mem=1'b1;
        2'b01: en_cpu=1'b1;
        default: en_io =1'b1;
    endcase
end
endmodule
```

Note: It is too easy for a pre-synthesis simulation to have startup problems that cause signals to go unknown and to be treated as a don't care by the casex statement. For this reason, in general, the casex statement should be avoided for synthesis RTL coding(use case instead of casex).

Coding For Synthesis(Continue...)

How Synthesis treats casez? The Verilog casez statement treats all z and ? bits as don't cares, whether they appear in the case expression or in the case item being tested. In the following model, if both en (enable) bits go high during simulation, the en_mem output will be driven high. In a synthesized gate-level model, the outputs would most likely go unknown indicating a design problem. This is a mismatch between pre-synthesis and post synthesis simulations.

```
module endec_z (output reg en_mem, en_cpu, en_io, input [1:0] en);
always @*
begin
    en_mem=1'b0; en_cpu=1'b0; en_io =1'b0;
    casez (en)
        2'b1?: en_mem=1'b1;
        2'b01: en_cpu=1'b1;
        default: en_io =1'b1;
    endcase
end
endmodule
```

Note: It is unlikely (but not impossible) that a pre-synthesis simulation would experience stray high impedance values on most design signals. For this reason, in general, the casez statement is safe to use but, noting the above potential for problems, they should be used with caution(With casez, a problem would occur if an input were initialized to a high impedance state).

Coding For Synthesis(Continue...)

Making 'X' Assignments? When making assignments in RTL code, sometimes it is tempting to assign the 'X' value. The 'X' assignment is interpreted as an unknown by the Verilog simulator (with the exception of casex as previously discussed), but is interpreted as a "don't care" by synthesis tools. Making 'X' assignments can cause mismatches between pre- and post-synthesis simulations. The x-output can be useful to help find bugs in the design during pre-synthesis simulations. It can also help direct the synthesis tool to optimize the design based on a don't care assignment. Note: the code8b synthesizes to a smaller and faster implementation than the code8a due to synthesis optimization.



```
/*Modules code8a will give a simulation mismatch if the select lines ever take on the value of 2'11.*/  
module code8a (y, a, b, c, s);  
output y; input a, b, c; input [1:0] s; reg y;  
always @(a or b or c or s)  
begin y = 1'bx;  
case (s)  
2'b00: y = a;  
2'b01: y = b;  
2'b10: y = c;  
endcase  
end  
endmodule
```



```
/*module code8b will have no such mismatch. This mismatch can be valuable if the select line combination of 2'b11 is never expected, */  
module code8a (y, a, b, c, s);  
output y; input a, b, c; input [1:0] s; reg y;  
always @(a or b or c or s)  
case (s)  
2'b00: y = a;  
2'b01: y = b;  
2'b10, 2'b11: y = c;  
endcase  
end  
endmodule
```

Coding For Synthesis(Continue...)

Timing Delays(Synthesis Ignores delay)? Synthesis tools ignore time delay in a model. Adding time delays to a Verilog pre-synthesis simulation can cause a mis-match between pre-synthesis and post-synthesis simulations and in general should be avoided. In the following delay-line model, the latch enable output is delayed in a pre-synthesis simulation but the delay will be removed from the post-synthesis implementation, potentially causing a delayed latch signal to be enabled too soon.

```
/* Adding delay elements to a synthesized model typically requires instantiation of the delay  
element in the pre-synthesis RTL model */
```

```
`timescale 1ns/1ns  
module delay1 (latch_en_dly, latch_en);  
input    latch_en;  
Output   latch_en_dly;  
reg      latch_en_dly;  
  
always @*  
    latch_en_dly <= #25 latch_en;  
endmodule
```

Design Compiler Flow

