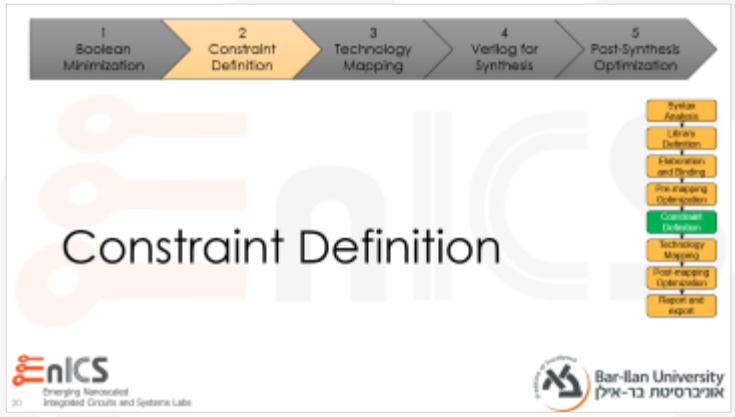
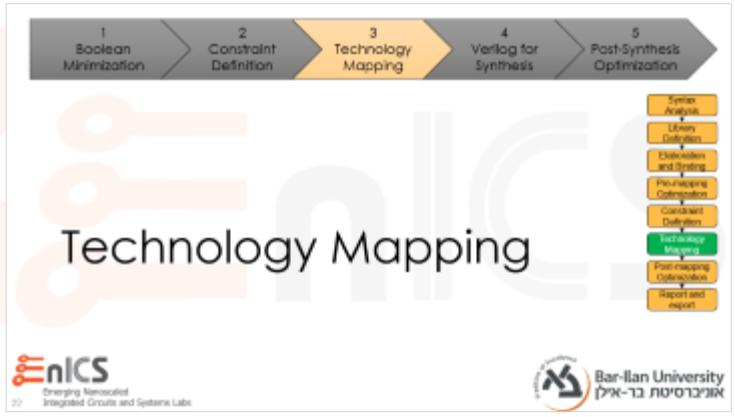
Lecture Outline





Digital VLSI Design

Lecture 4: Logic Synthesis Part 2

Semester A, 2018-19

Lecturer: Dr. Adam Teman

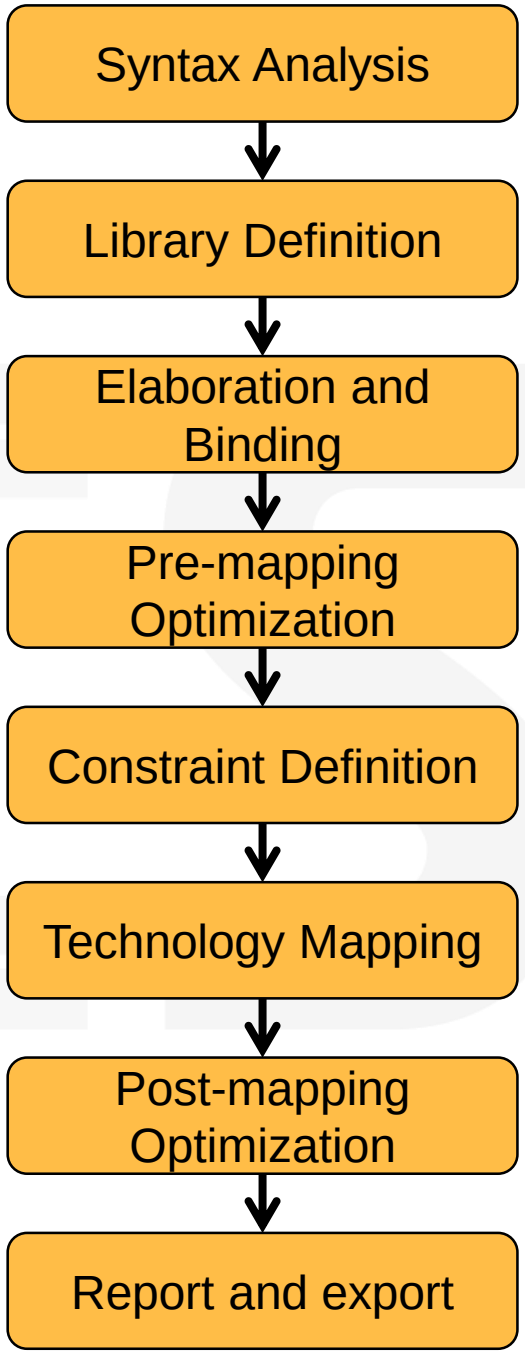
November 16, 2018

Emerging Nanoscaled
Integrated Circuits and Systems Labs

Tradition of Excellence
Bar-Ilan University
אוניברסיטת בר-אילן

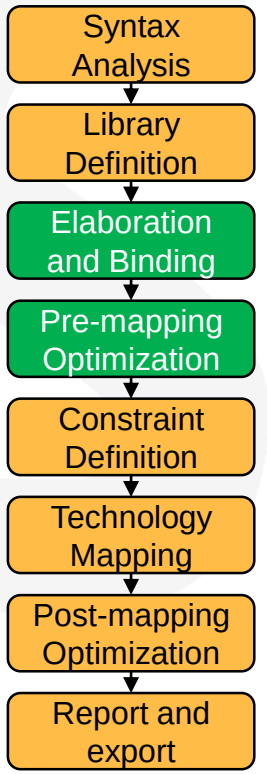
What have we discussed so far?

- **Not too much…**
  - We briefly discussed compilation.
  - And then we really dove down and dirty into standard cell libraries.
- **So at this point:**
  - We have loaded our design into the synthesizer.
  - And we have loaded our standard cell library and IPs.
- **We can move on to discuss the "brains" of the synthesis process.**



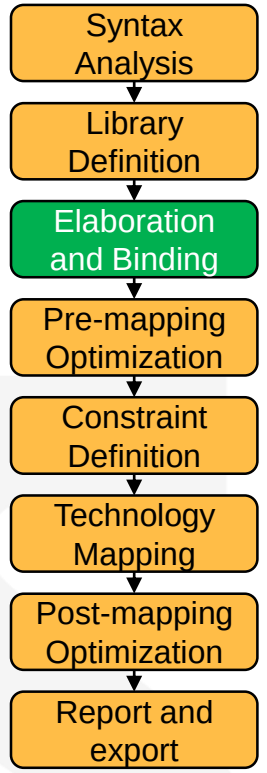
Boolean Minimization

Mapping to Generics and Libs, Basics of Boolean Minimization (BDDs, Two-Level Logic, Espresso)



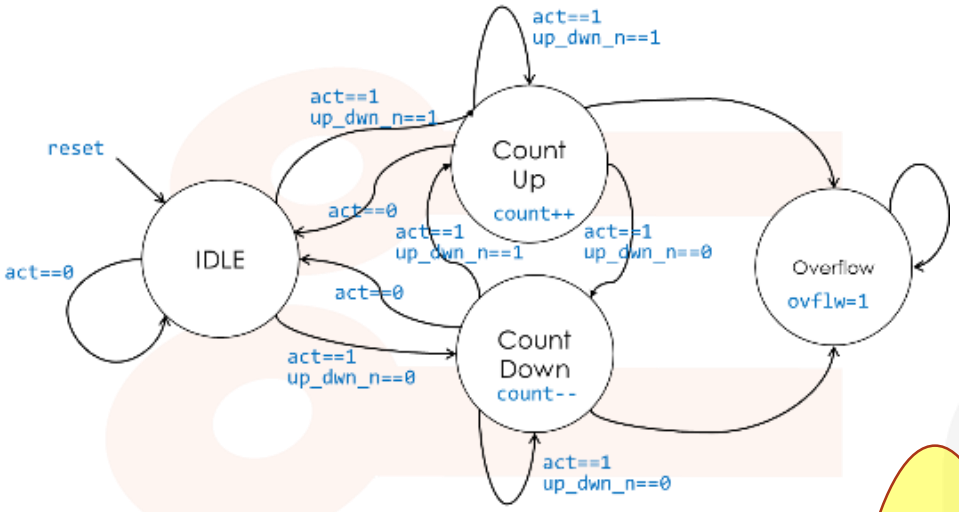
Emerging Nanoscaled Integrated Circuits and Systems Labs
Bar-Ilan University
אוניברסיטת בר-אילן

Elaboration and Binding



- **During the next step of logic synthesis, the tool:**
  - Compiles the RTL into a Boolean data structure (elaboration)
  - Binds the non-Boolean modules to leaf cells (binding), and
  - Optimizes the Boolean logic (minimization).
- **The resulting design is mapped to generic, technology independent logic gates.**
- **This is the core of synthesis and has been a very central subject of research in computer science since the eighties.**

Elaboration Illustrated

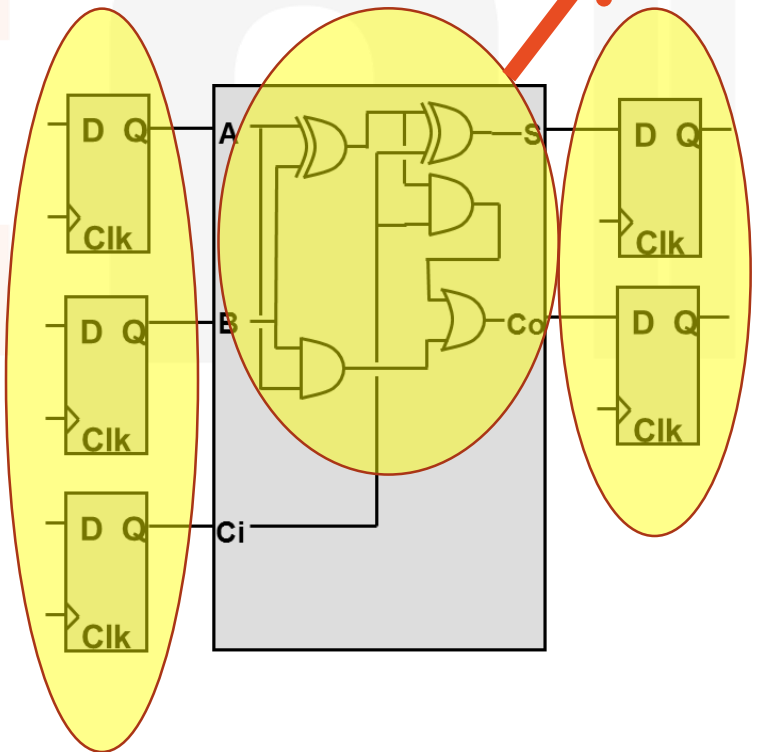


x_1 x_2 x_3	$f(x_1x_2x_3)$
0 0 0	1
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	0
1 0 1	0
1 1 0	1

Boolean Logic

F_1=ACB'+DEF'+A'BCF+…
F_2=C'B'+D'GH'+A'FG'+…
…

Two-level Logic



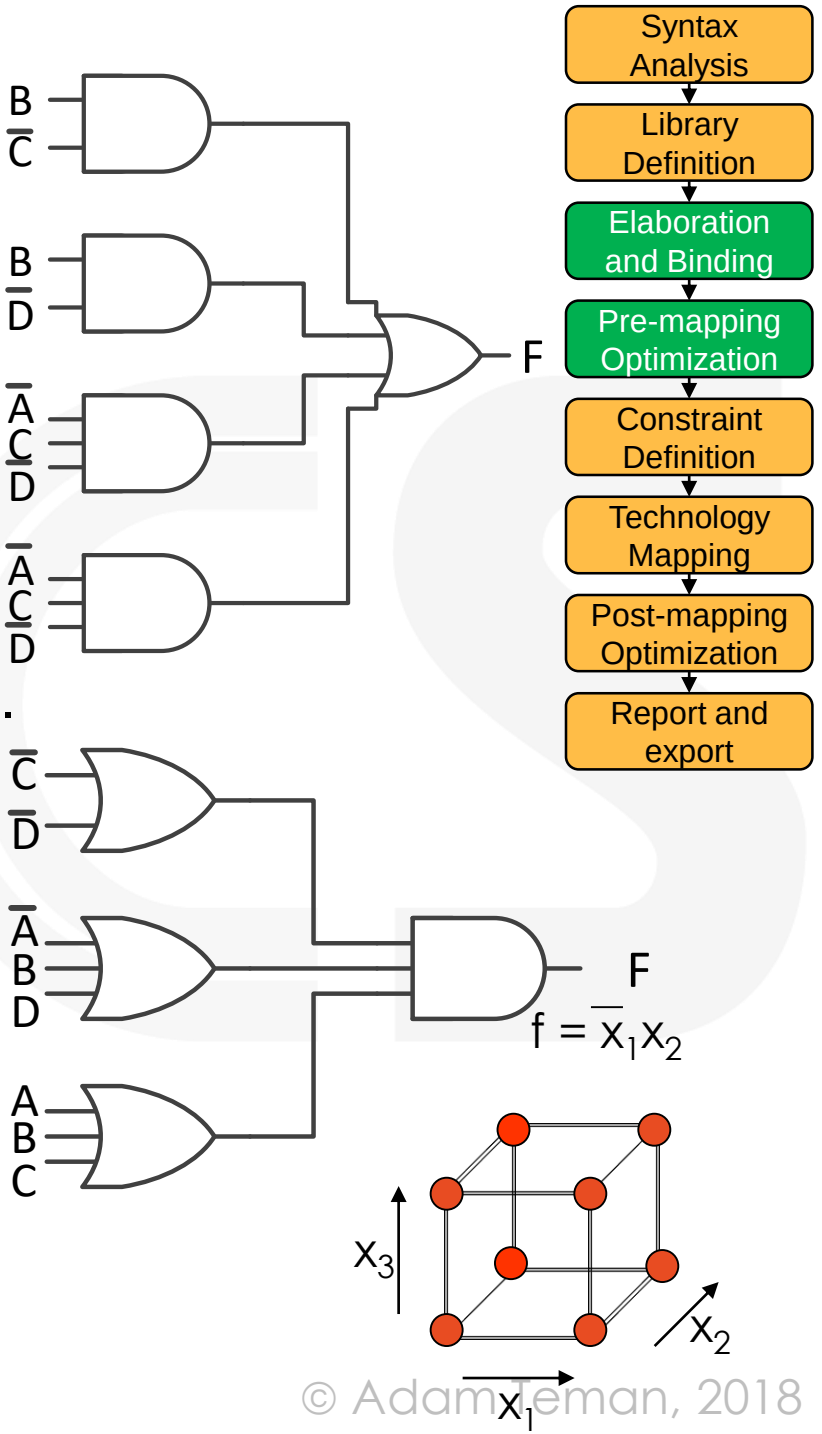
Inferred Registers

- Syntax Analysis
- Library Definition
- Elaboration and Binding
- Pre-mapping Optimization
- Constraint Definition
- Technology Mapping
- Post-mapping Optimization
- Report and export

Two-Level Logic

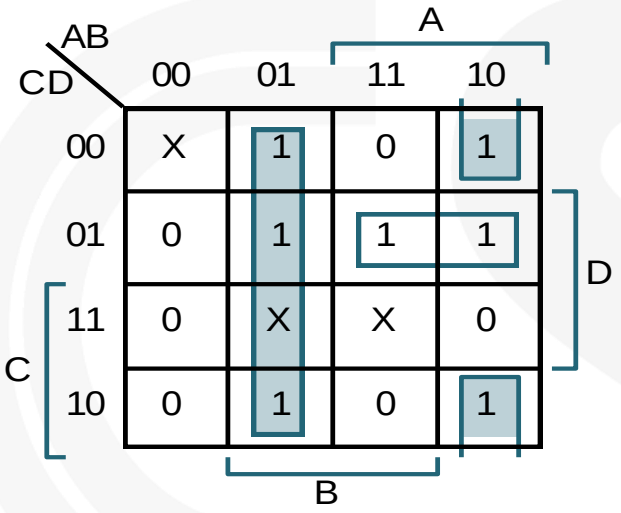
- **During elaboration, primary inputs and outputs (ports) are defined and sequential elements (flip-flops, latches) are *inferred*.**
- **This results in a set of combinational logic clouds with:**
  - Input ports and register outputs are inputs to the logic
  - Output ports and register inputs are the outputs of the logic
  - The outputs can be described as Boolean functions of the inputs.
  - *The goal of Boolean minimization is to reduce the number of literals in the output functions*.

- **Many different data structures are used to represent the Boolean functions:**
  - Truth tables, cubes, Binary Decision Diagrams, equations, etc.
  - A lot of the research was developed upon SOP or POS representation, which is better known as "Two-Level Logic"



Two-Level Logic Minimization

- **In our freshman year we learned about Karnaugh maps:**
  - For n inputs, the map contains 2^n entries
  - Objective is to find the minimum *prime* cover
- **However…**
  - Difficult to automate (NP-complete)
  - Number of cells is exponential (<6 variables)
- **A different approach is the Quine-McCluskey method**
  - Easy to implement in software
  - BUT computational complexity too high
- **Some Berkeley students fell asleep while solving a Quine-McCluskey exercise.**



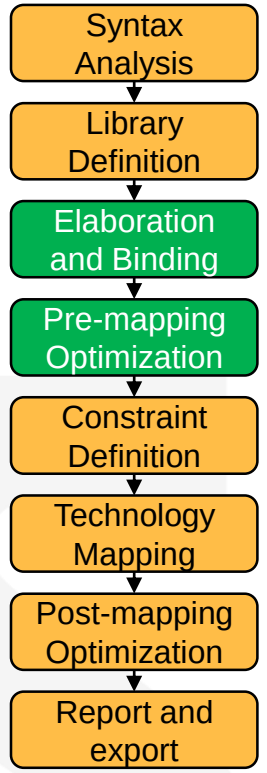
They needed a shot of *Espresso*.

Syntax Analysis
Library Definition
Elaboration and Binding
Pre-mapping Optimization
Constraint Definition
Technology Mapping
Post-mapping Optimization
Report and export

Espresso Heuristic Minimizer

- **Start with an SOP solution.**
  - Expand
    - Make each cube as *large as possible* without covering a point in the OFF-set.
    - Increases the number of literals (worse solution)
  - Irredundant
    - Throw out *redundant* cubes.
    - Remove smaller cubes whose points are covered by larger cubes.
  - Reduce
    - The cubes in the cover are reduced in size.
  - In general, the new cover will be different from the initial cover.
    - "expand" and "irredundant" steps can possibly find out a new way to cover the points in the ON-set.
    - Hopefully, the new cover will be smaller.

```
ESPRESSO(F) {
  do {
      reduce(F);
      expand(F);
      irredundant(F);
  } while (fewer terms in F);
  verify(F);
}
```

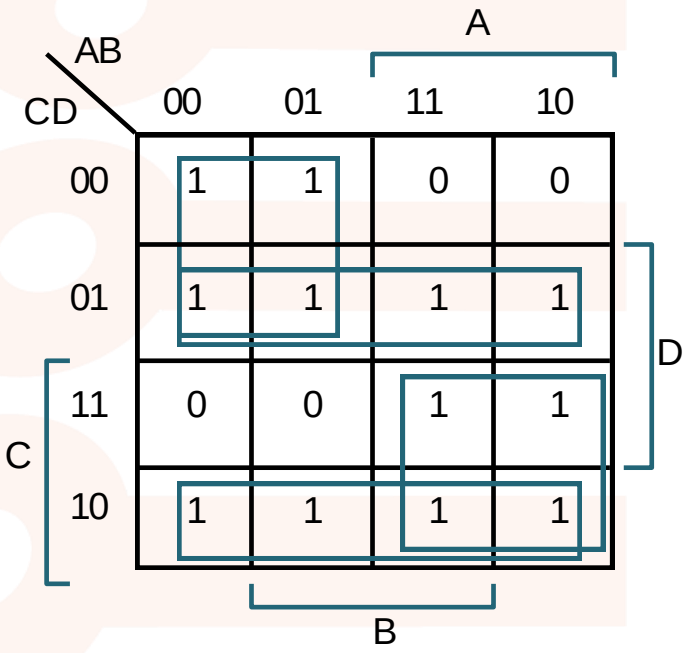


Espresso Example

$$f = \bar{A}\bar{C} + \bar{C}D + AC + C\bar{D}$$

$$f = \bar{A}\bar{C} + A\bar{C}D + AC + \bar{A}C\bar{D}$$

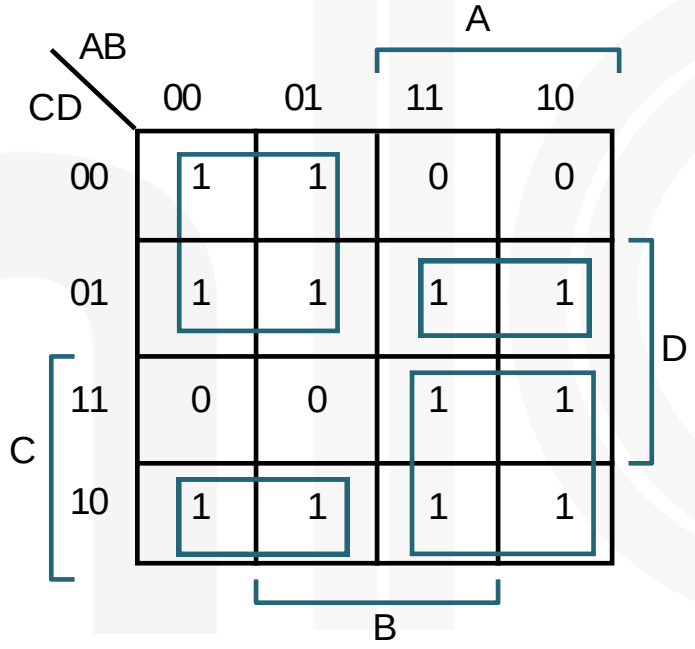
Starting SOP Form:



Initial Set of Primes found by Steps1 and 2 of the Espresso Method

4 primes, irredundant cover, but not a minimal cover!

Reduce:



Result of REDUCE: Shrink primes while still covering the ON-set

Choice of order in which to perform shrink is important

```
ESPRESSO(F) {
  do {
       reduce(F);
       expand(F);
       irredundant(F);
  } while (F smaller);
  verify(F);
}
```

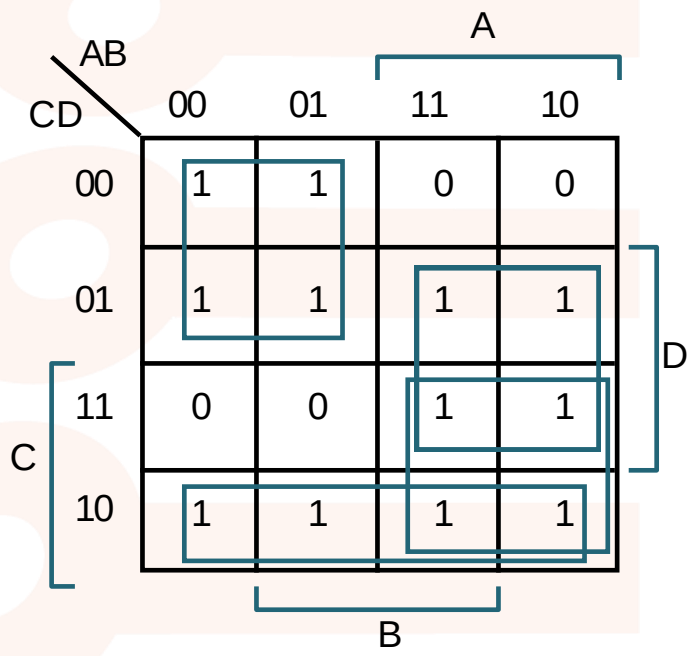
Espresso Example

$$f = \bar{A}\bar{C} + AD + AC + C\bar{D}$$

$$f = \bar{A}\bar{C} + AD + C\bar{D}$$

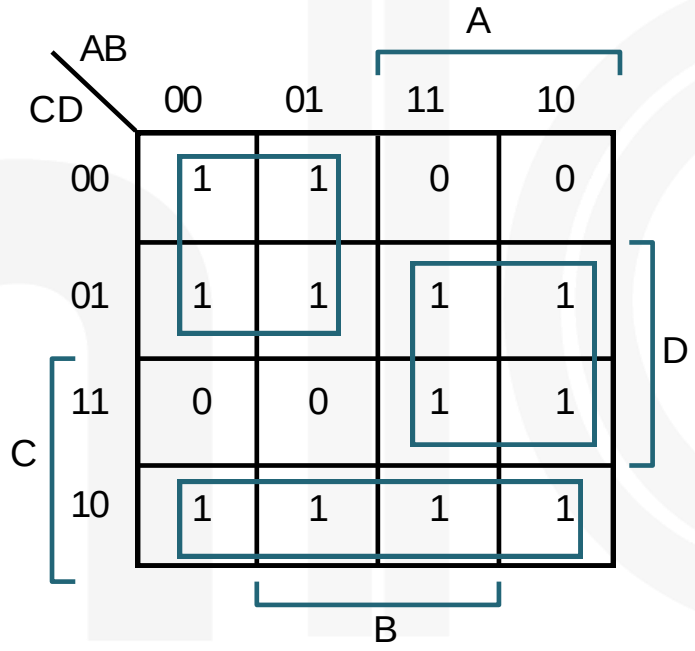
Only 6 literals!

Expand:



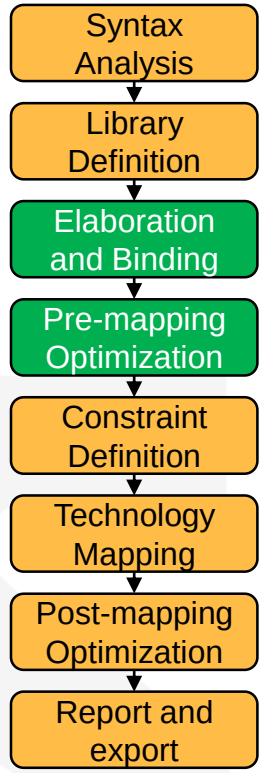
Second EXPAND generates a different set of prime implicants

Irredundant:



IRREDUNDANT COVER found by final step of espresso

Only three prime implicants!

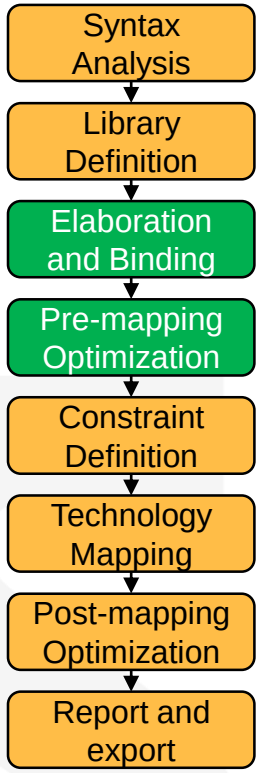



```
ESPRESSO(F) {
  do {
      reduce(F);
      expand(F);
      irredundant(F);
  } while (F smaller);
  verify(F);
}
```

Multi-level Logic Minimization

- **Two-level logic minimization has been widely researched and many famous methods have come out of it.**
  - However, often it is better and/or more practical to use many levels of logic (remember *logical effort*?).

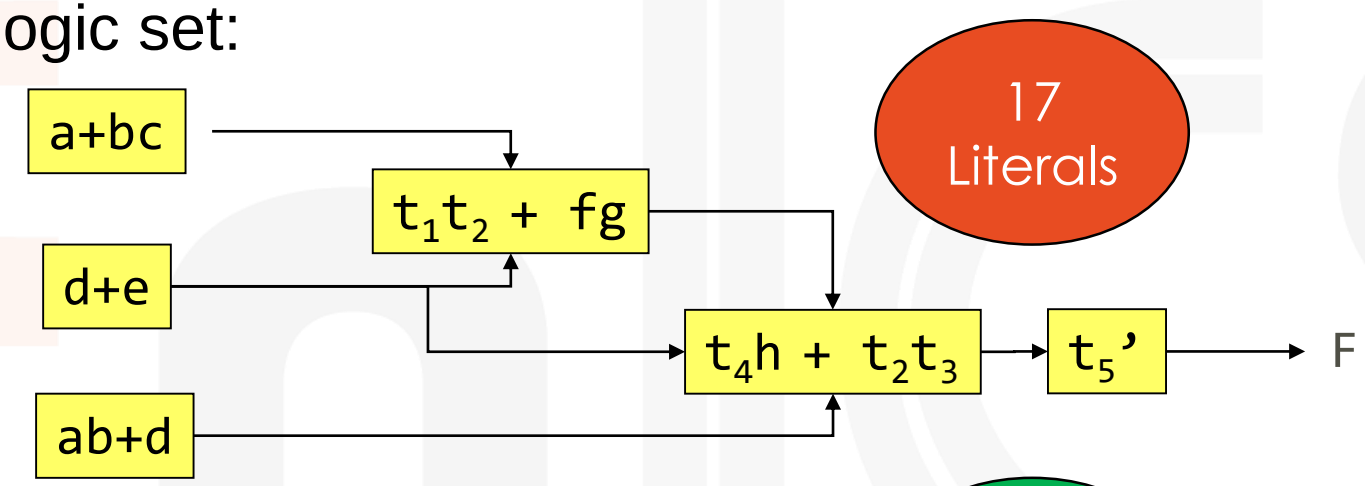
- **Therefore, a whole new optimization regime, known as *multi-level logic minimization* was developed.**
  - We will not cover multi-level minimization in this course, however, you should be aware that the output of logic minimization will generally be multi-level and not two-level.



Multi-level Logic Minimization

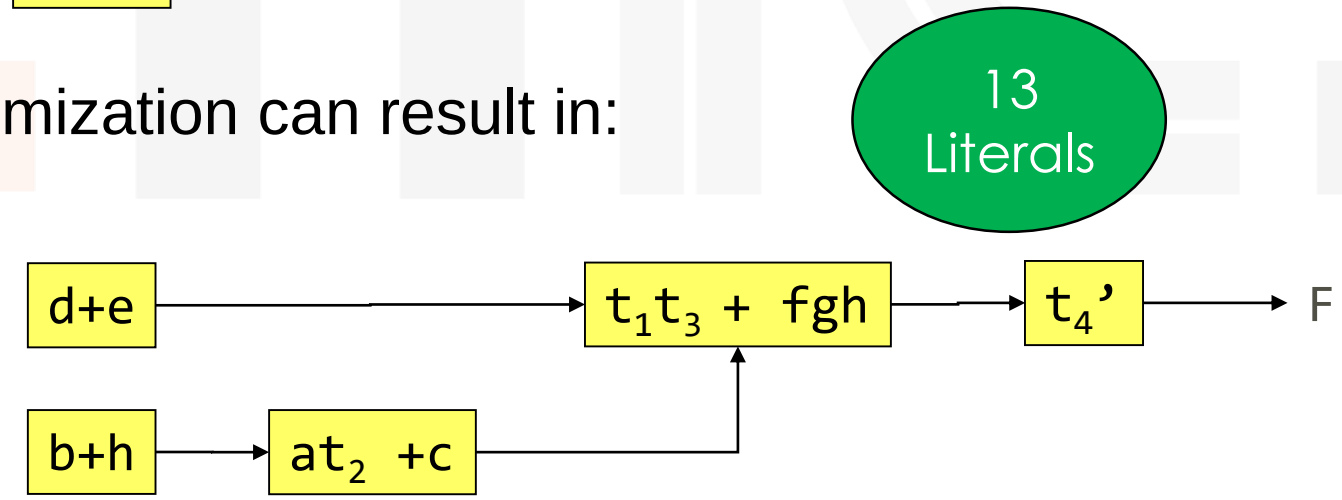
- **For example:**
  - Given the following logic set:

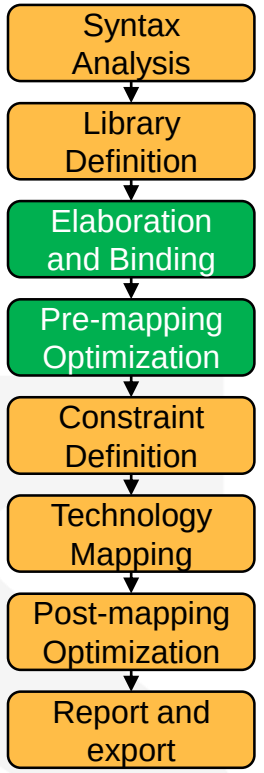
```
t1 = a + bc;
t2 = d + e;
t3 = ab + d;
t4 = t1t2 + fg;
t5 = t4h + t2t3;
F = t5';
```



  - Multi-level Logic Minimization can result in:

```
t1 = d + e;
t2 = b + h;
t3 = at2 + c;
t4 = t1t3 + fgh;
F = t4';
```

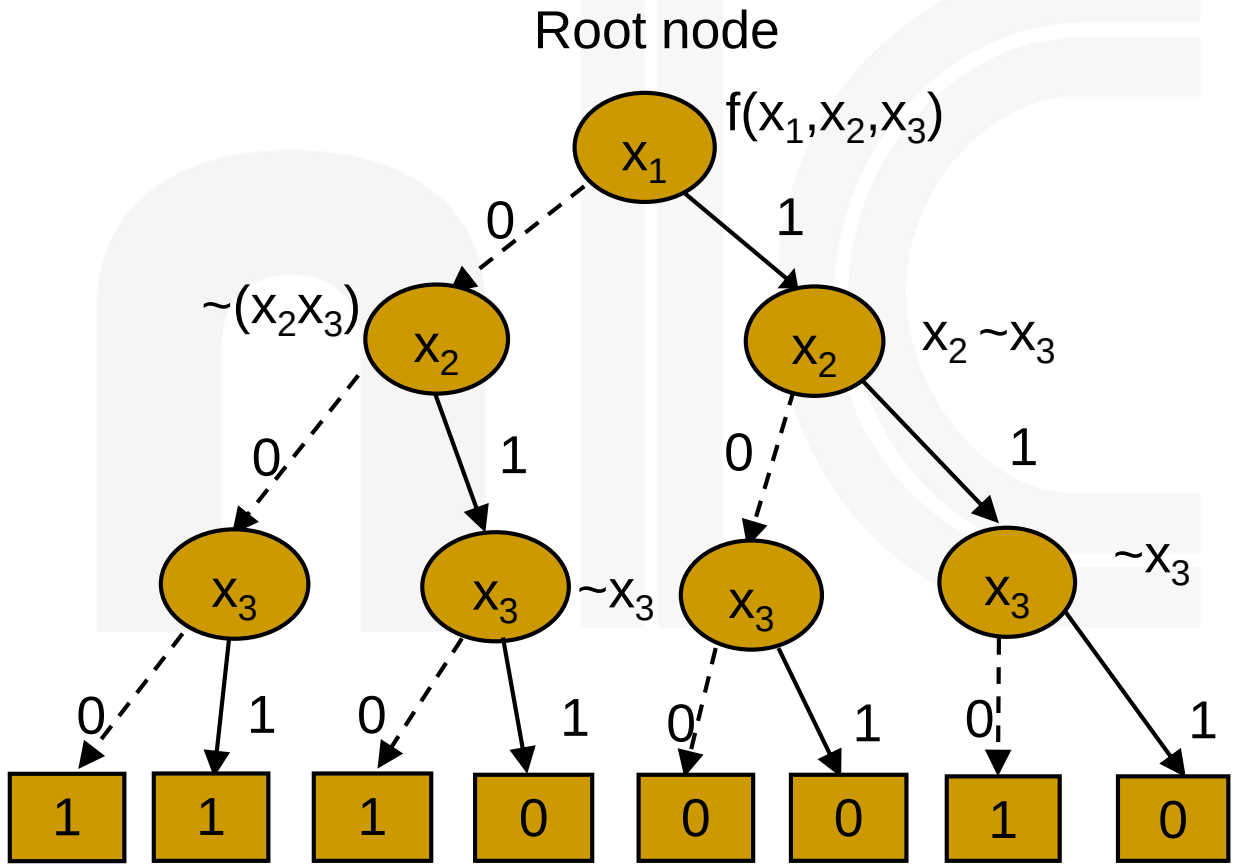




Binary Decision Diagrams (BDD)

- **BDDs are DAGs that represent the truth table of a given function**

x_1 x_2 x_3	$f(x_1x_2x_3)$
0 0 0	1
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	0
1 0 1	0
1 1 0	1



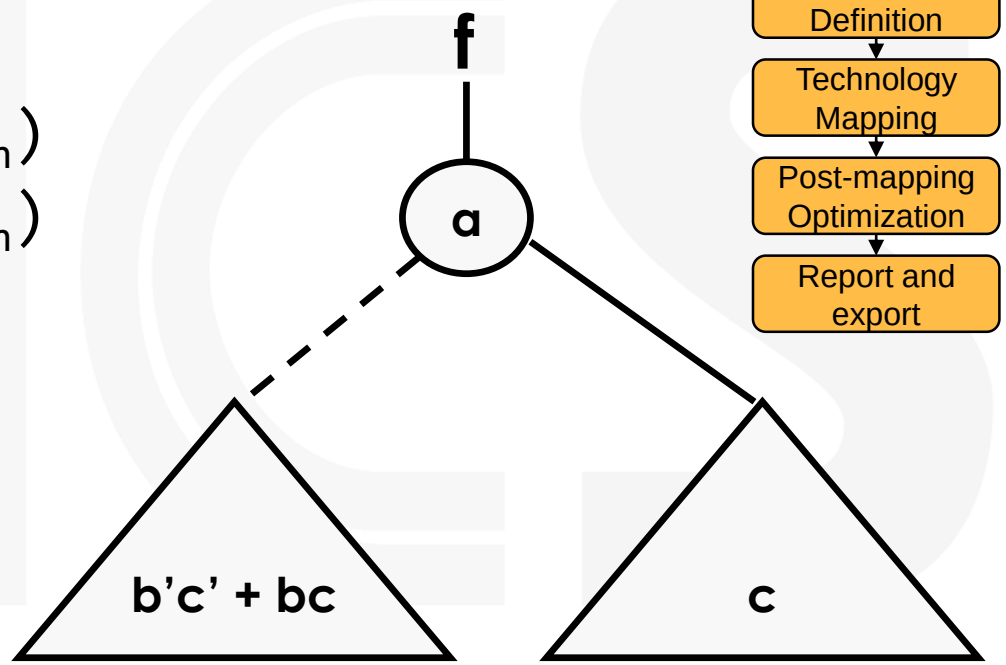
$$f(x_1, x_2, x_3) = \sim x_1 \sim x_2 \sim x_3 + \sim x_1 \sim x_2 x_3 + \sim x_1 x_2 \sim x_3 + x_1 x_2 \sim x_3$$

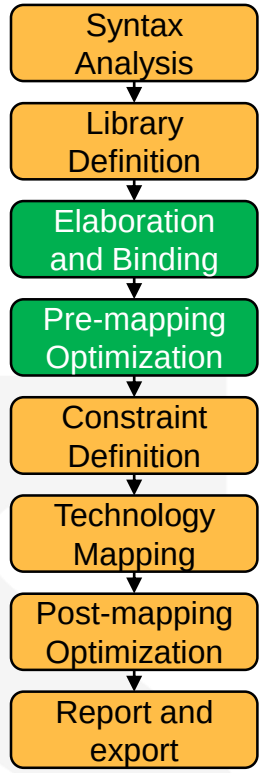
Binary Decision Diagrams (BDD)

- **The Shannon Expansion of a function relates the function to its *cofactors*:**
  - Given a Boolean function $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$
  - Positive cofactor: $f_i^1 = f(x_1, x_2, \ldots, 1, \ldots, x_n)$
  - Negative cofactor: $f_i^0 = f(x_1, x_2, \ldots, 0, \ldots, x_n)$
- **Shannon's expansion theorem states that**
  - $f = x_i' \, f_i^0 + x_i \, f_i^1$
  - $f = (x_i + f_i^0)(x_i' + f_i^1)$
- **This leads to the formation of a BDD:**
  - Example: $f = ac + bc + a'b'c'$

$$= a' (b'c' + bc) + a (c + bc)$$

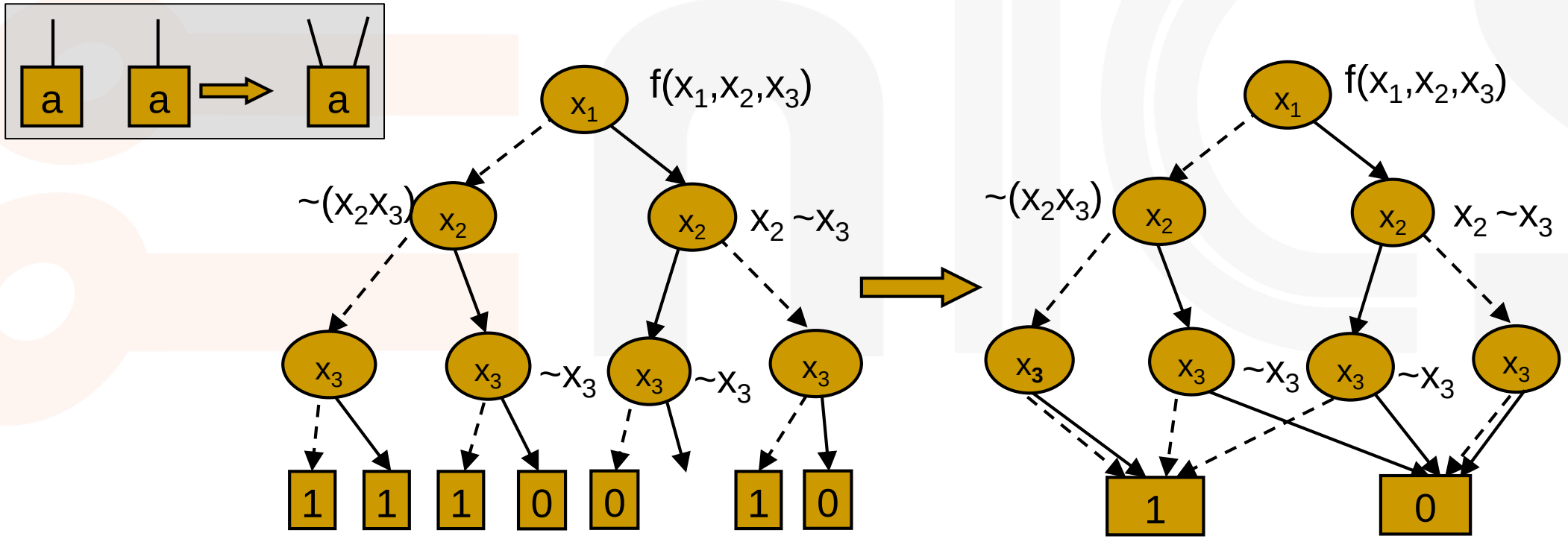
$$= a' (b'c' + bc) + a (c)$$





Reduced Ordered BDD (ROBDD)

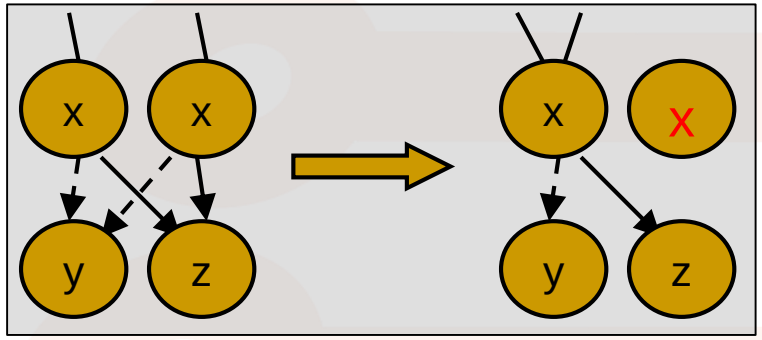
- **BDDs can get very big.**
  - So let's see if we can provide a reduced representation.
- **Reduction Rule 1: Merge equivalent leaves**

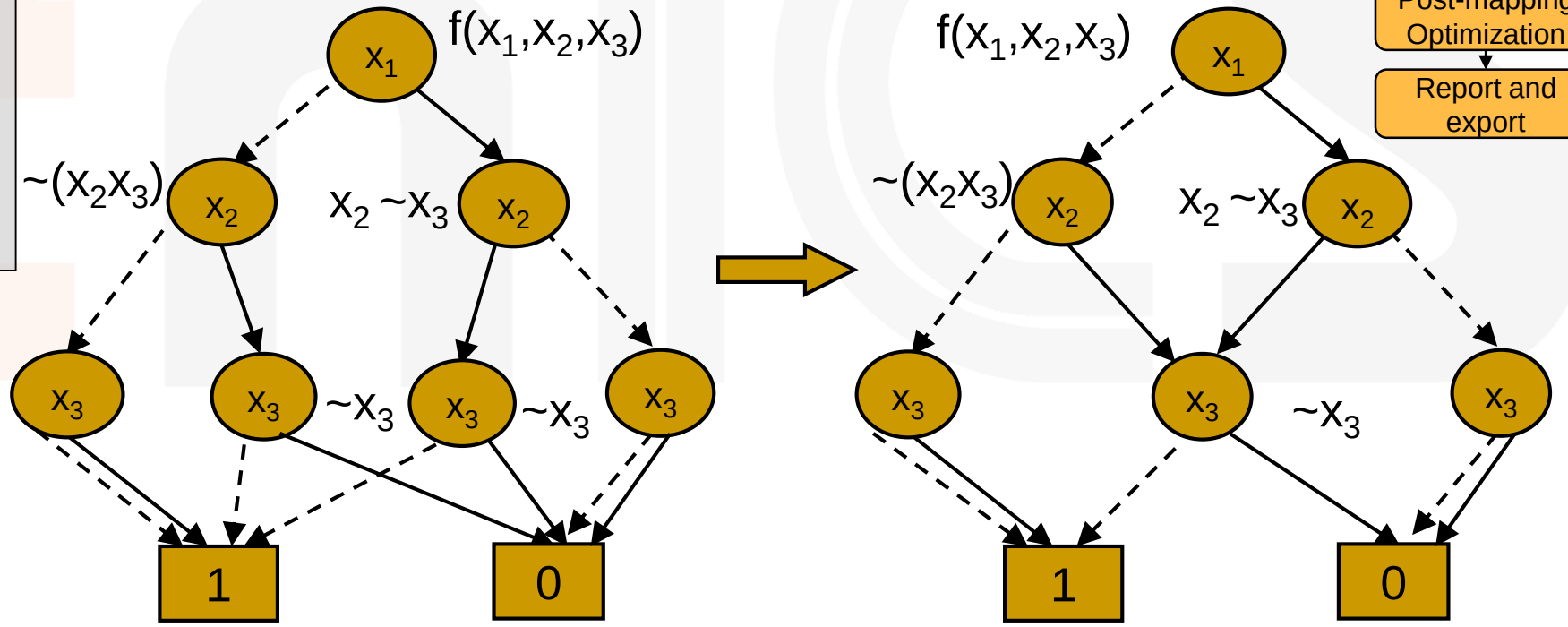


$f(x_1, x_2, x_3) = \sim x_1 \sim x_2 \sim x_3 + \sim x_1 \sim x_2 x_3 + \sim x_1 x_2 \sim x_3 + x_1 x_2 \sim x_3 = \sim x_1 \sim x_2 + \sim x_1 x_2 \sim x_3 + x_1 x_2 \sim x_3$

Reduced Ordered BDD (ROBDD)

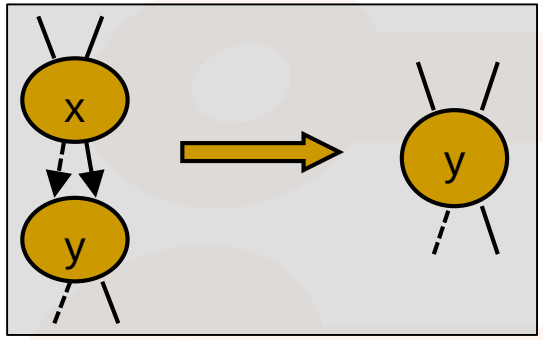
- **BDDs can get very big.**
  - So let's see if we can provide a reduced representation.
- **Reduction Rule 2: Merge isomorphic nodes**

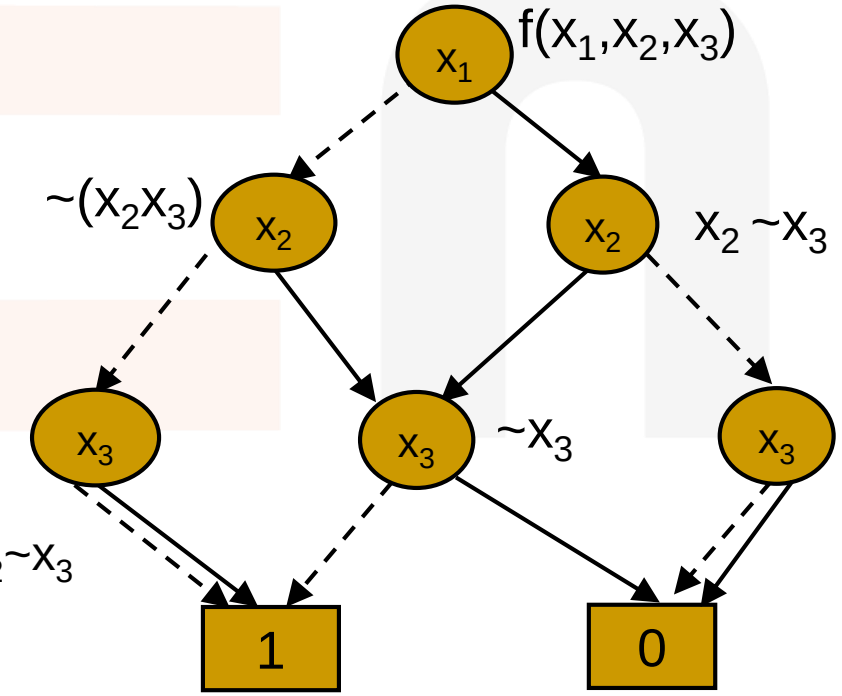




Reduced Ordered BDD (ROBDD)

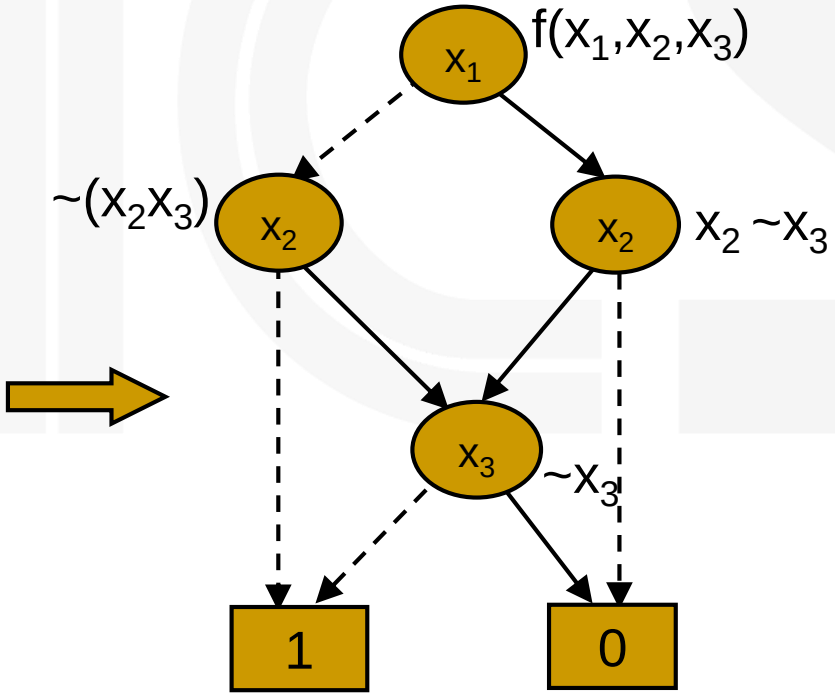
- **BDDs can get very big.**
  - So let's see if we can provide a reduced representation.
- **Reduction Rule 3: Eliminate Redundant Tests**

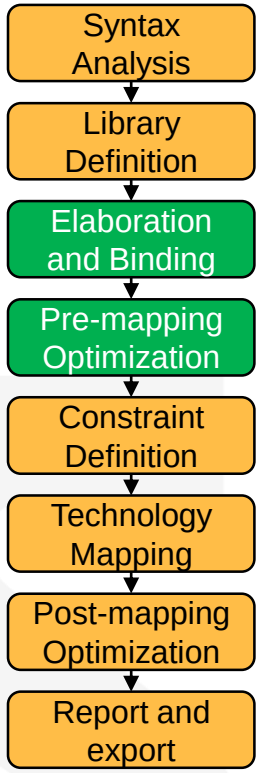




$f(x_1, x_2, x_3) = \sim x_1 \sim x_2 +$

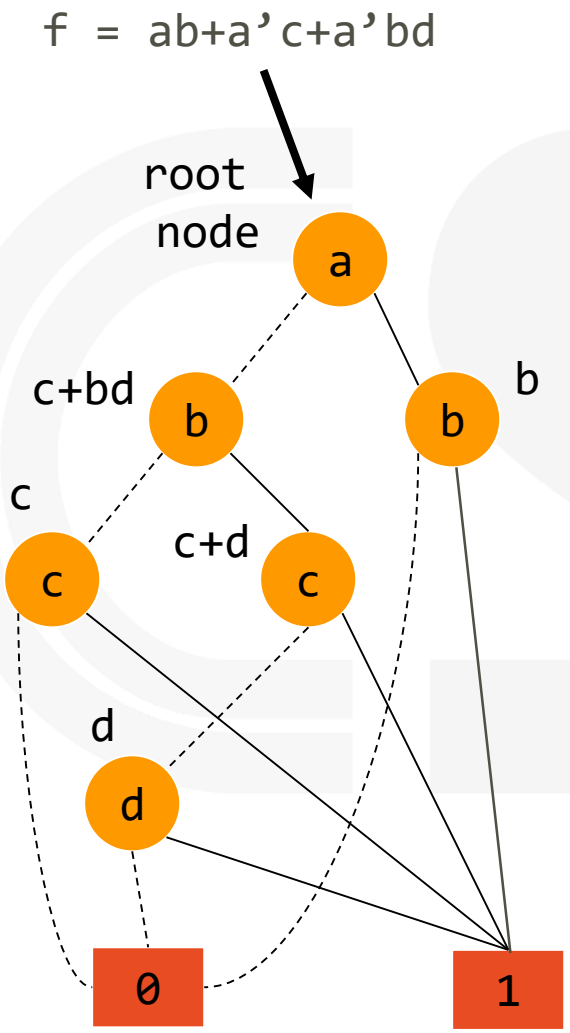
$\sim x_1 x_2 \sim x_3 + x_1 x_2 \sim x_3$

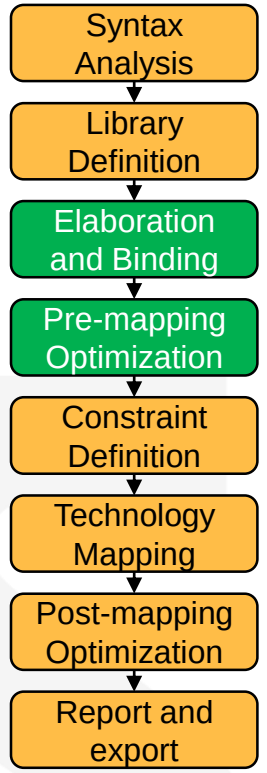




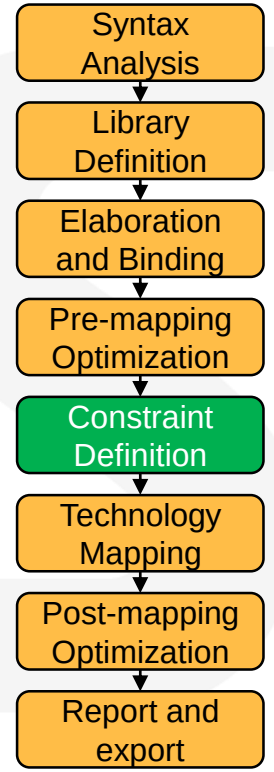
Binary Decision Diagrams (BDD)

- **Some benefits of BDDs:**
  - Check for *tautology* is trivial.
    - BDD is a constant 1.
  - Complementation.
    - Given a BDD for a function f, the BDD for f' can be obtained by interchanging the terminal nodes.
  - Equivalence check.
    - Two functions f and g are equivalent if their BDDs (under the same variable ordering) are the same.
- **An Important Point:**
  - The size of a BDD can vary drastically if the order in which the variables are expanded is changed.
  - The number of nodes in the BDD can be exponential in the number of variables in the worst case, even after reduction.

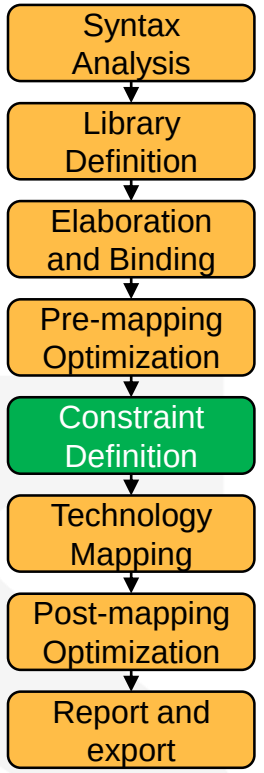




Constraint Definition



Constraint Definition



- **Following Elaboration, the design is loaded into the synthesis tool and stored inside a data structure.**
- **Hierarchical ports (inputs/outputs) and registers can be accessed by name.**

```
set in_ports [get_ports IN*]
set regs [get_cells -hier *_reg]
```

- **At this point, we can load the design constraints in SDC format, as we will learn in *Lecture 5*.**

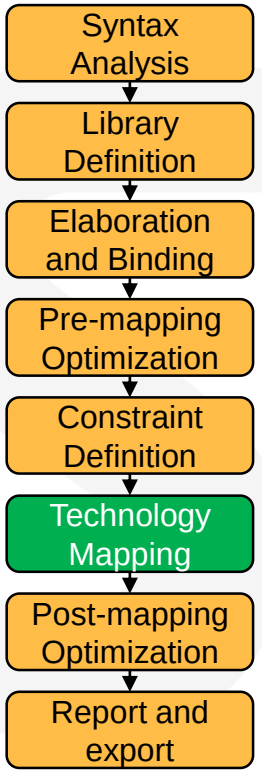
```
read_sdc -verbose sdc/constraints.sdc
```

  - For example, to create a clock and define the target frequency:

```
create_clock -period $PERIOD -name $CLK_NAME [get_ports $CLK_PORT]
```

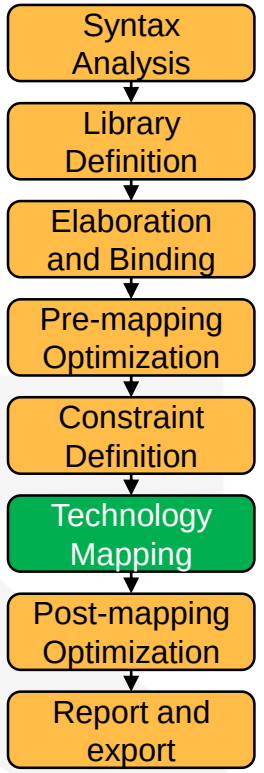
- **Carefully check that all constraints were accepted by the tool!**

Technology Mapping



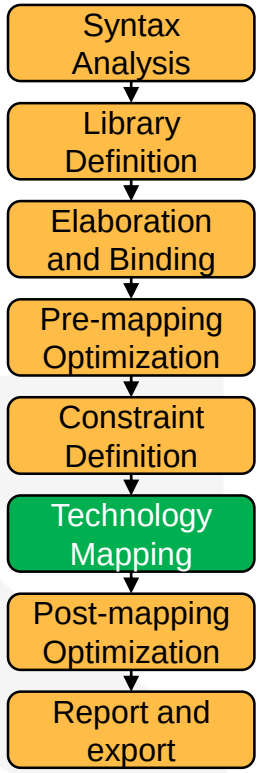
Technology mapping

- **Technology mapping is the phase of logic synthesis when gates are selected from a technology library to implement the circuit.**
- **Why technology mapping?**
  - Straight implementation may not be good.
  - For example, `F=abcdef` as a 6-input AND gate causes a long delay.
  - Gates in the library are pre-designed, they are usually optimized in terms of area, delay, power, etc.
  - Fastest gates along the critical path, area-efficient gates (combination) off the critical path.
- **Can apply a minimum cost tree-covering algorithm to solve this problem.**



Technology Mapping Algorithm

- **Using a recursive tree-covering algorithm, we can easily, and almost optimally, map a logic network to a technology library.**
- **This process incurs three steps:**
  - Map netlist and tech library to simple gates
    - Describe the netlist with only NAND2 and NOT gates
    - Describe SC library with NAND2 and NOT gates and associate a cost with each gate
  - Tree-ifying the input netlist
    - Tree covering can only be applied to trees!
    - Split tree at all places, where fanout > 2
  - Minimum Cost Tree matching
    - For each node in your tree, recursively find the minimum cost target pattern at that node.
- **Let us briefly go through these steps**



1. Simple Gate Mapping

- **Apply De Morgan laws to your Boolean function to make it a collection of NAND2 and NOT gates.**
  - Let's take the example of multi-level logic minimization:

```
t1 = d + e;
t2 = b + h;
t3 = at2 + c;
t4 = t1t3 + fgh;
F = t4';
```

$$t_1 = d + e = \text{NAND}\left(\bar{d}, \bar{e}\right)$$

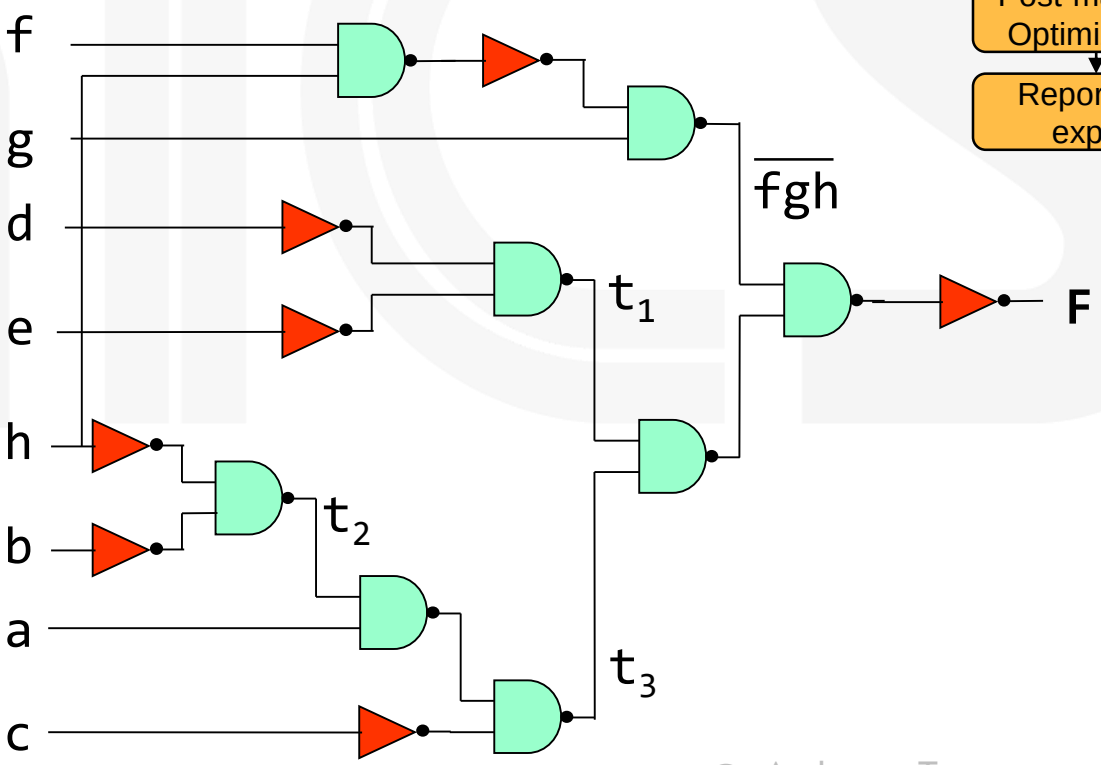
$$t_2 = b + h = \text{NAND}\left(\bar{b}, \bar{h}\right)$$

$$t_3 = at_2 + c = \overline{\overline{at_2} \cdot \bar{c}} = \text{NAND}\left(\text{NAND}\left(a, t_2\right), \bar{c}\right)$$

$$t_4 = t_1 t_3 + fgh = \text{NAND}\left(\overline{t_1 t_3}, \overline{fgh}\right)$$

$$fgh = \overline{\overline{fh} \cdot g} = \overline{\overline{\overline{fh} \cdot g}} = \overline{\text{NAND}\left(\overline{\text{NAND}\left(f, h\right)}, g\right)}$$

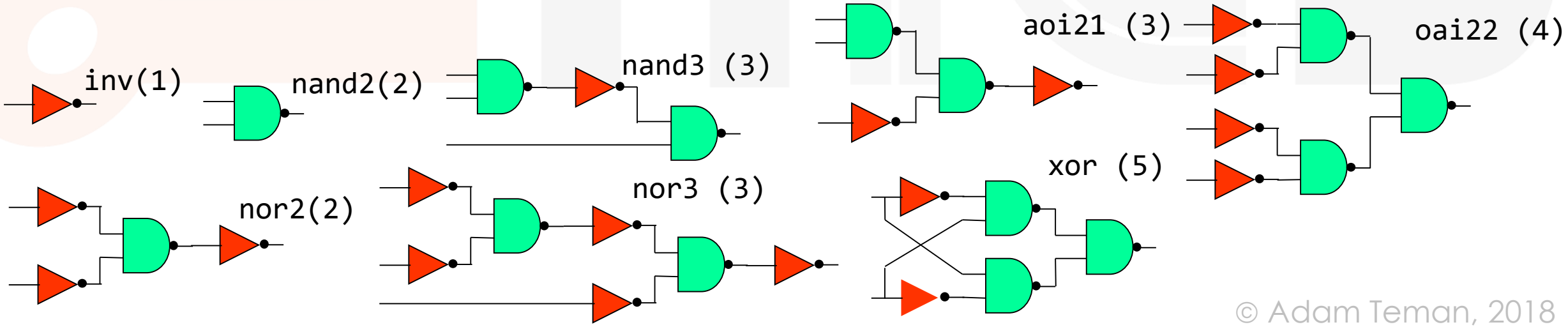
$$F = \bar{t}_4$$



1. Simple Gate Mapping

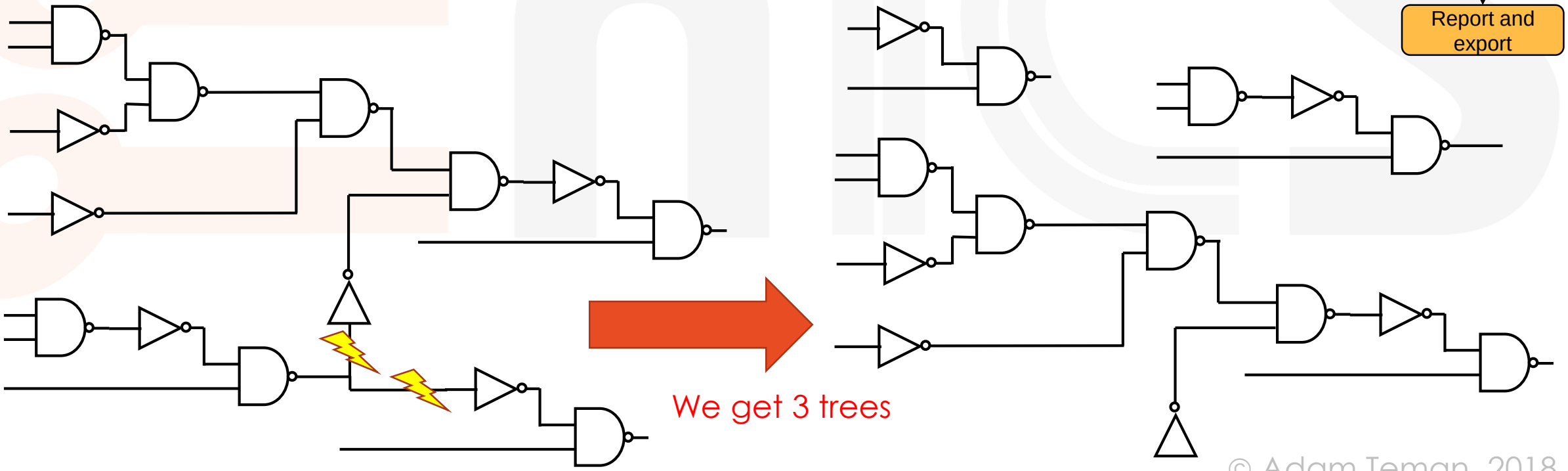
- **And then, given a set of gates (standard cell library) with cost metrics (area/delay/power):**

- **We need to define the gates with the same NAND2/NOT set:**



2. Tree-ifying

- **To apply a tree covering algorithm, we must work on a tree!**
  - Is any given logic network a tree?
  - No!
  - We must *break the tree* at any node with fanout>2

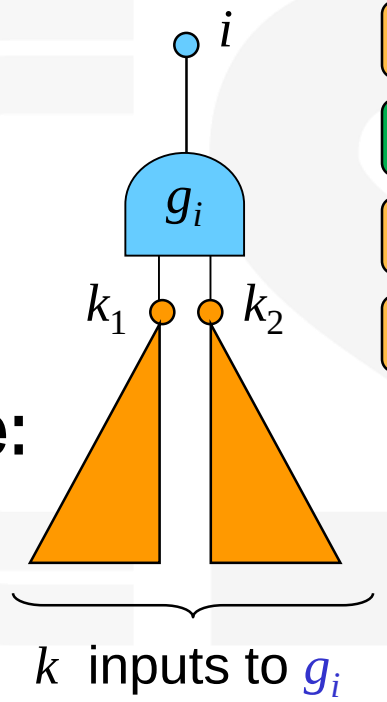


3. Minimum Tree Covering

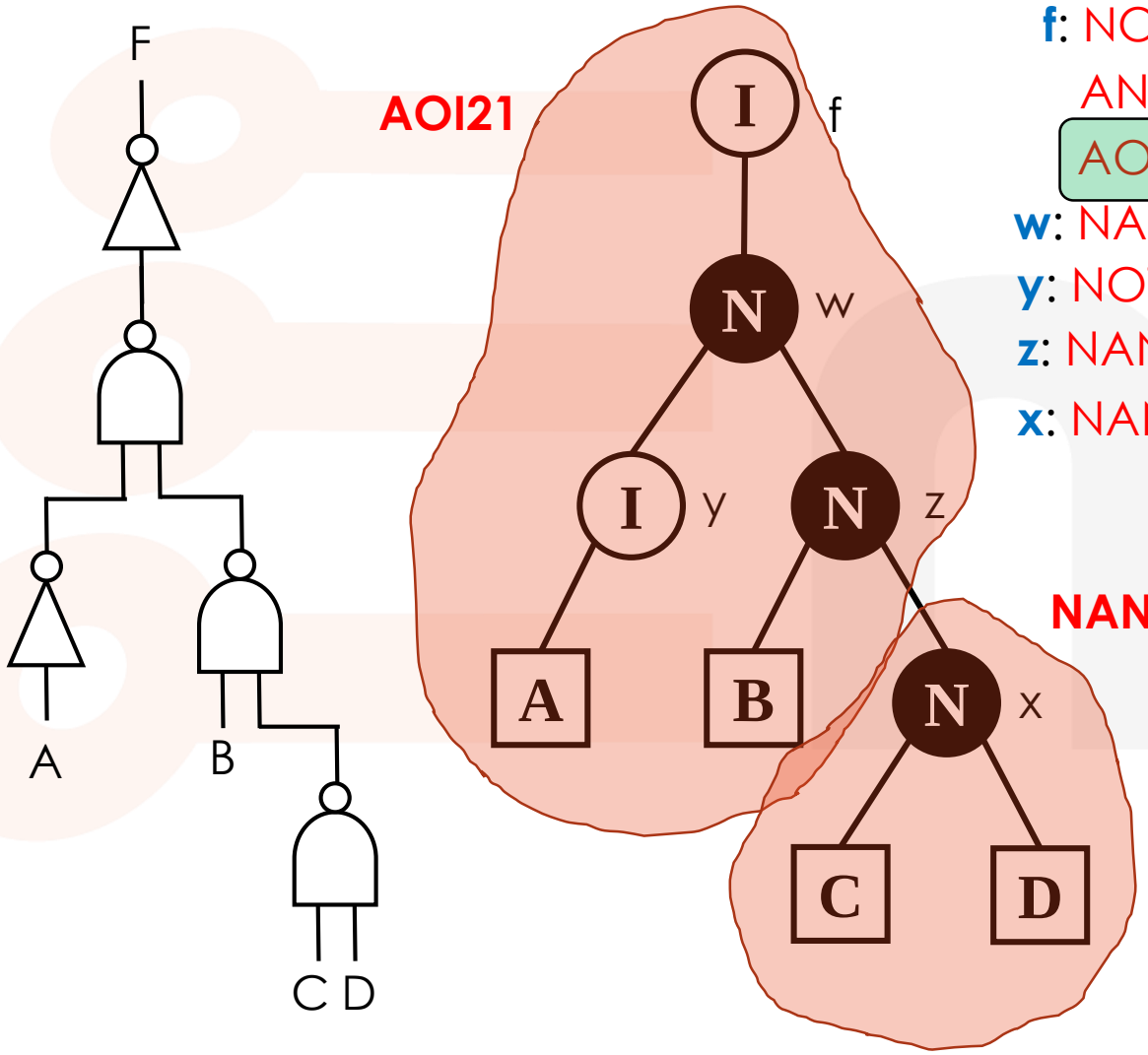
- **Now, we can apply a recursive algorithm to achieve a minimum cover:**
  - Start at the output of the graph.
  - For each node, find all the matching target patterns.
  - The cost of node i for using gate g is:

$$\text{cost}(i) = \min_k \left\{ \text{cost}(g_i) + \sum_k \text{cost}(k_i) \right\}$$

    - where k_i are the inputs to gate g.
- **For simplicity, we will redraw our graph and show an example:**
  - Every NOT is just an empty circle: (I)
  - Every NAND is just a full circle: (N)
  - Every input is just a box: [A]



3. Minimum Tree Covering - Example



f: NOT 2 + min(w) = 2 + 11 = 13

AND2 4 + min(y)+min(z) = 4 + 2 + 6 = 12

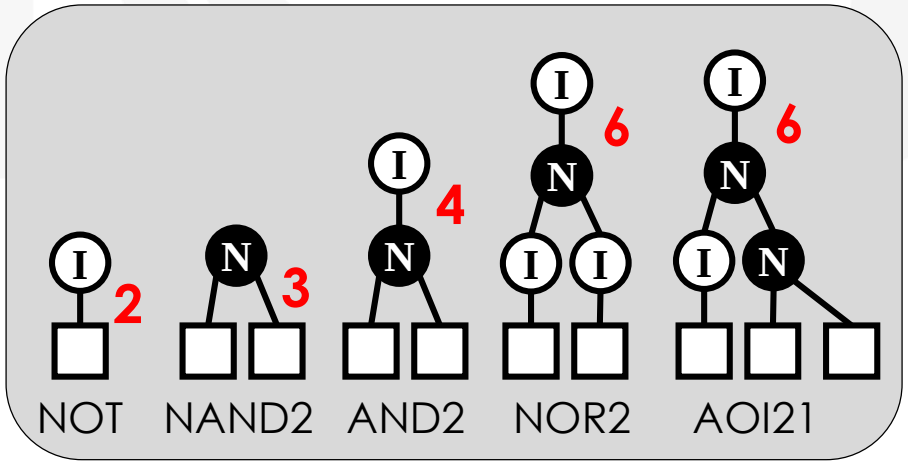
AOI21 6 + min(x) = 6 + 3 = 9

w: NAND2 3 + min(y)+min(z) = 3 + 2 + 6 = 11

y: NOT 2

z: NAND2 3 + min(x) = 3 + 3 = 6

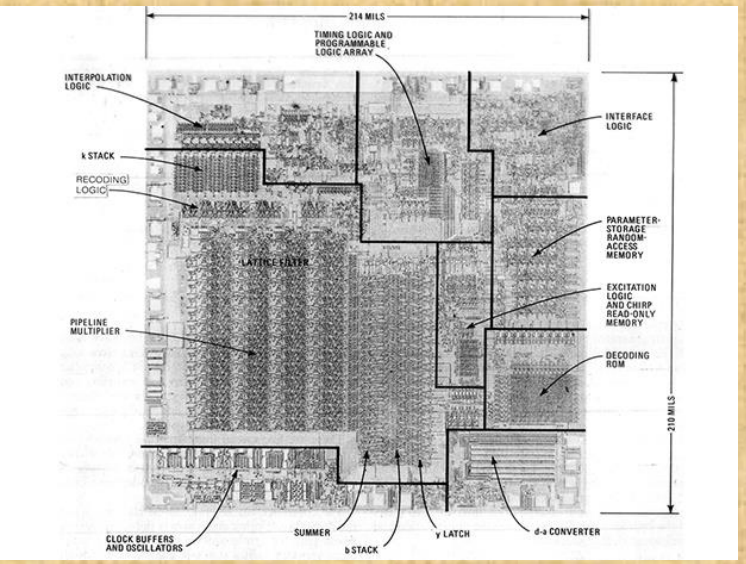
x: NAND2 3



The Chip Hall of Fame

- **Since we're speaking about synthesis, how about a very famous synthesizer**

Texas Instruments TMC0281 Speech Synthesizer



- Used in Texas Instrament's "Speak & Spell" and used by E.T. to "phone home"
- The first single-chip speech synthesizer.
- Release date: 1978 Chip Size: 44 mm^2
- The sound emerges from a combination of buzzing, hissing and popping.

2017 Inductee to the IEEE Chip Hall of Fame

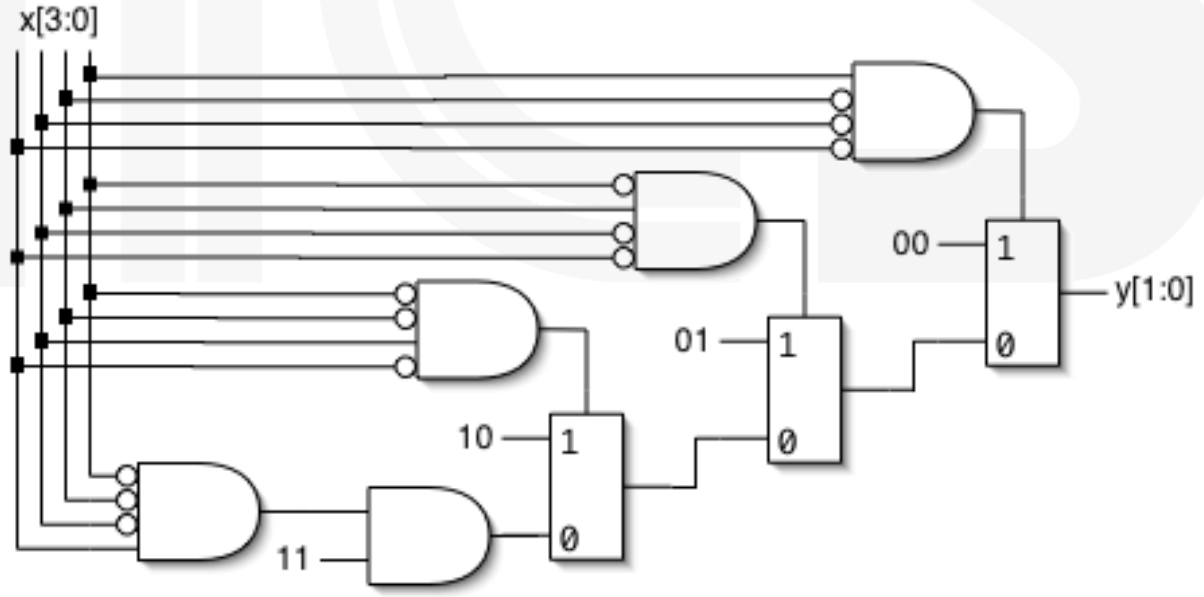
Verilog for Synthesis - revisited

Some things we may have missed

- **Now that we've seen how synthesis works, let's revisit some of the things we may have skipped or only briefly mentioned earlier…**
- **Let's take a simple 4→2 encoder as an example:**
  - Take a one-hot encoded vector and output the position of the '1' bit.
  - One possibility would be to describe this logic with a nested `if-else` block:

```
always @(x)
  begin : encode
    if      (x == 4'b0001) y = 2'b00;
    else if (x == 4'b0010) y = 2'b01;
    else if (x == 4'b0100) y = 2'b10;
    else if (x == 4'b1000) y = 2'b11;
    else y = 2'bxx;
  end
```

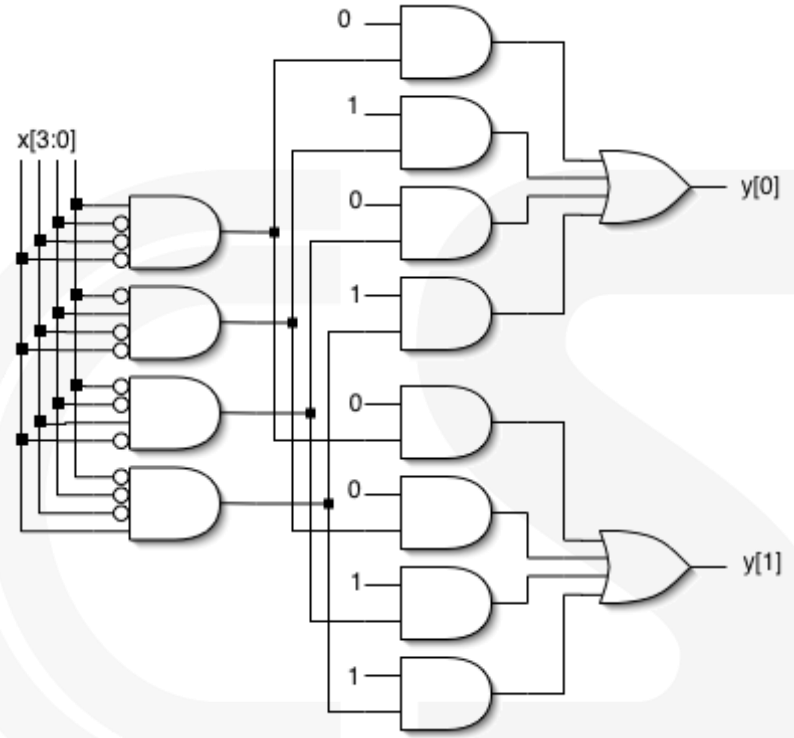
  - The result is known as "priority logic"
    - i.e., some bits have priority over others…

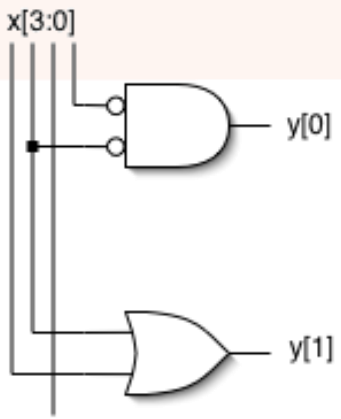


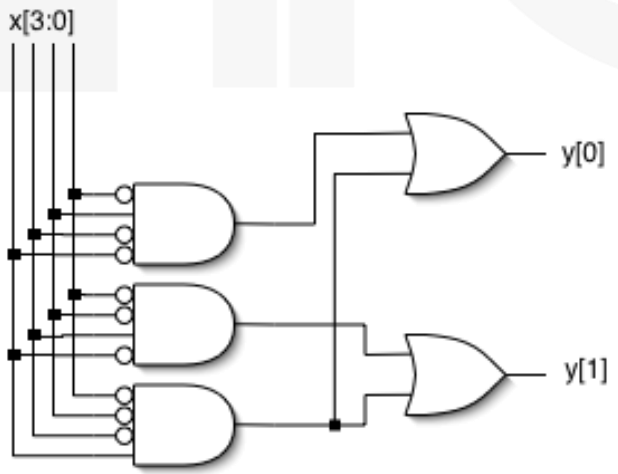
Some things we may have missed

- **It would have been better to use a `case` construct:**
  - All cases are matched in parallel
  - And better yet, synthesis can optimize away the constants and other Boolean equalities:

```
always @(x)
  begin : encode
    case (x)
      4'b0001: y = 2'b00;
      4'b0010: y = 2'b01;
      4'b0100: y = 2'b10;
      4'b1000: y = 2'b11;
      default: y = 2'bxx;
    endcase
  end
```



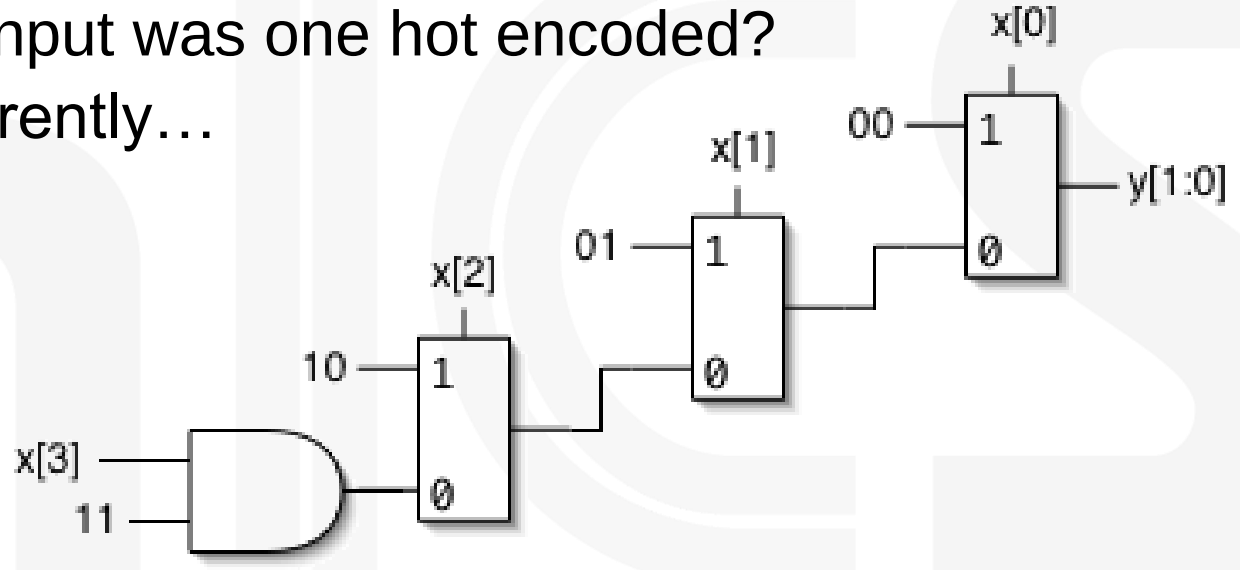




Some things we may have missed

- **In the previous example, if the encoding was wrong (i.e., not one-hot), we would have propagated an x in the logic simulation.**
  - But what if we *guarantee* that the input was one hot encoded?
  - Then we could write our code differently…

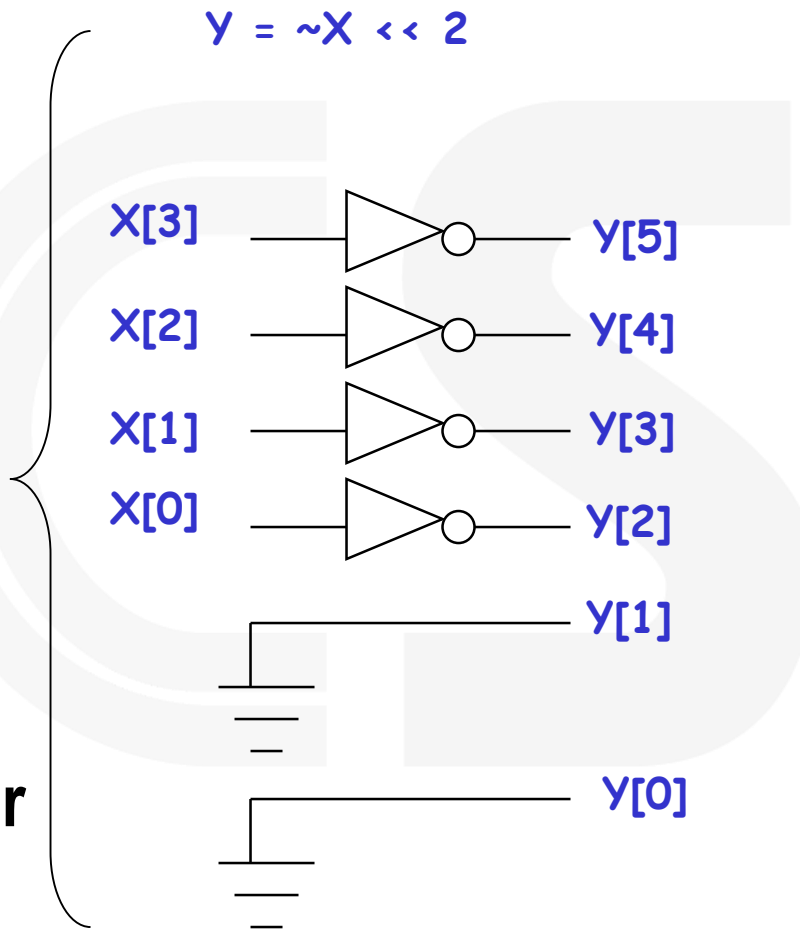
```
always @(x)
  begin : encode
    if (x[0])      y = 2'b00;
    else if (x[1]) y = 2'b01;
    else if (x[2]) y = 2'b10;
    else if (x[3]) y = 2'b11;
    else y = 2'bxx;
  end
```



  - In fact, we have implemented a "*priority decoder*" (the least significant '1' gets priority)

A few points about operators

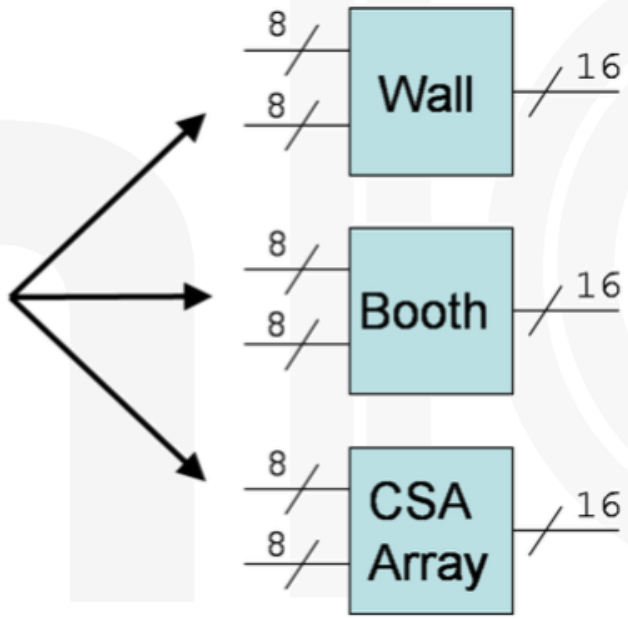
- **Logical operators map into primitive logic gates**
- **Arithmetic operators map into adders, subtractors, …**
  - Unsigned or signed 2's complement
  - Model carry: target is one-bit wider that source
  - Watch out for *, %, and /
- **Relational operators generate comparators**
- **Shifts by constant amount are just wire connections**
  - No logic involved
- **Variable shift amounts a whole different story → shifter**
- **Conditional expression generates logic or MUX**



Datapath Synthesis

- **Complex operators (Adders, Multipliers, etc.) are implemented in a special way**
  - E.g. Type of multiplier

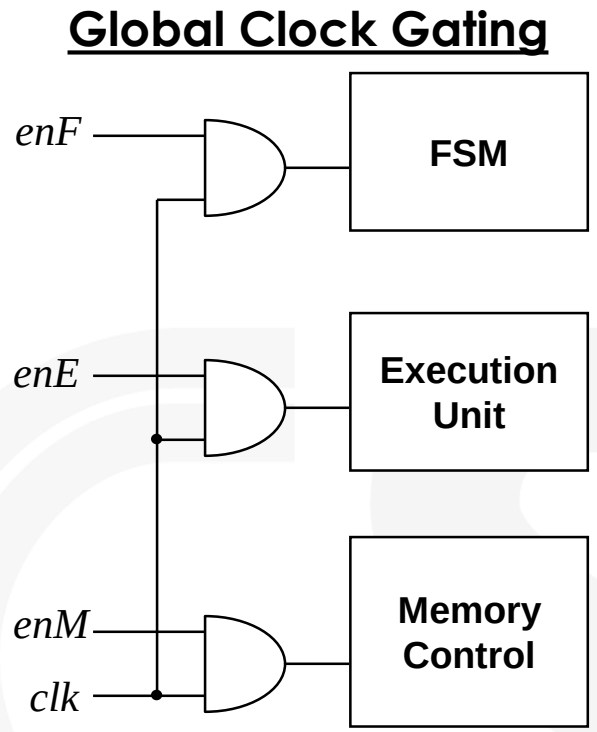
```
module mult (d1, d2, do);
input [7:0] d1, d2;
output [15:0] do;
assign do = d1 * d2;
endmodule
```

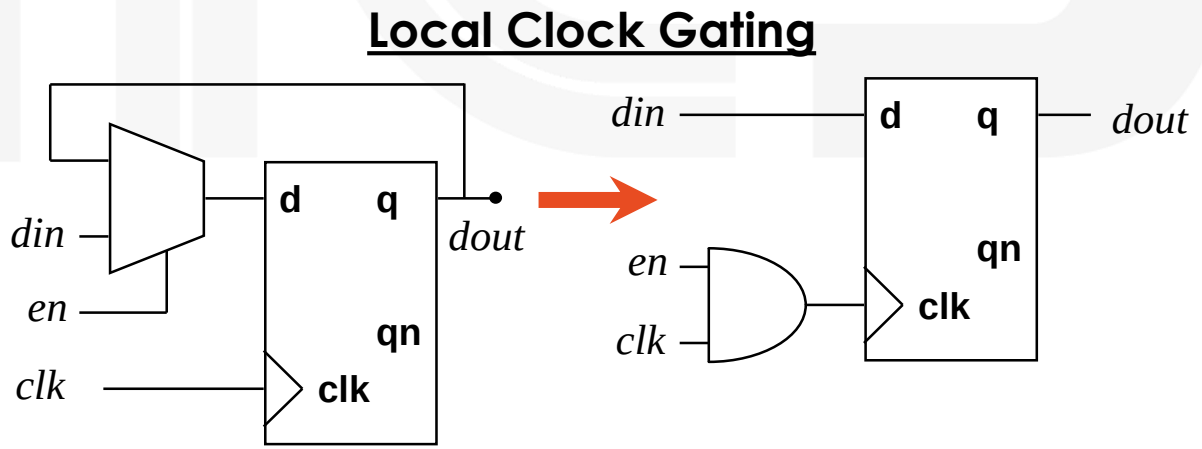


- **Pre-written descriptions can be found in *Synopsys DesignWare* or *Cadence ChipWare* IP libraries.**

Clock Gating

- **As you know, since a clock is continuously toggling, it is a major consumer of dynamic power.**
  - Therefore, in order to save power, we will try to turn off the clock for gates that are not in use.
- **Block level (Global) clock-gating**
  - If certain operating modes do not use an entire module/component, a clock gate should be defined in the RTL.
- **Register level (Local) clock-gating**
  - However, even at the register level, if a flip-flop doesn't change it's output, internal power is still dissipated due to the clock toggling.
  - This is very typical of an enabled signal sampling, and therefore can be automatically detected and gated by the synthesis tool.





Clock Gating

- **Local clock gating: 3 methods**
  - Logic synthesizer finds and implements local gating opportunities
  - RTL code explicitly specifies clock gating
  - Clock gating cell explicitly instantiated in RTL
- **Global clock gating: 2 methods**
  - RTL code explicitly specifies clock gating
  - Clock gating cell explicitly instantiated in RTL

- **Conventional RTL Code**

```
//always clock the register
 always @ (posedge clk) begin
     if (enable) q <= din;
 end
```

- **Low Power Clock Gated RTL**

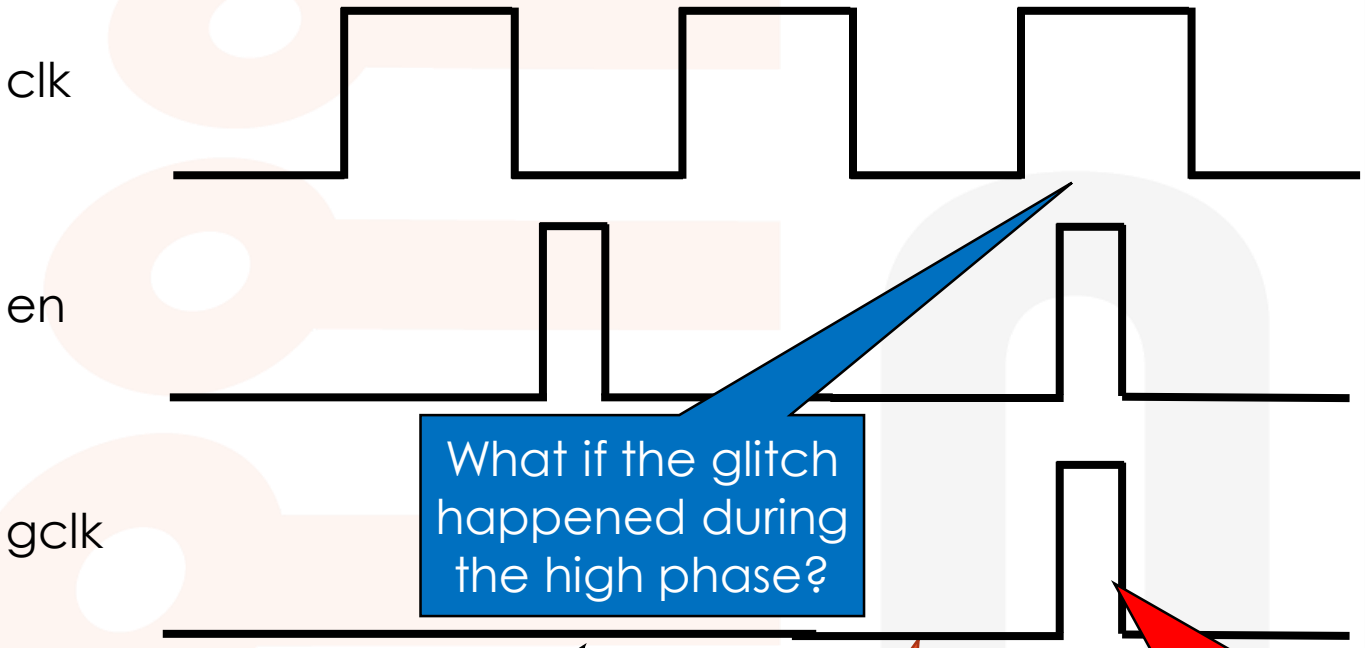
```
//only clock the ff when enable is true
    assign gclk = enable && clk;
    always @ (posedge gclk) begin
       q <= din;
   end
```

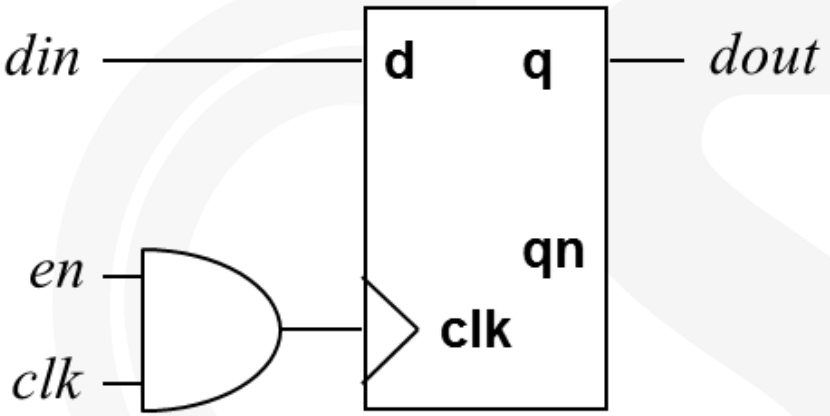
- **Instantiated Clock Gating Cell**

```
//instantiate a clock gating cell
   clkgx1 i1 (.en(enable), .cp(clk), .gclk_out(gclk));
  always @ (posedge gclk) begin
       q <= din;
   end
```

Clock Gating – Glitch Problem

- **What happens if there is a glitch on the enable signal?**





What if the glitch happened during the high phase?

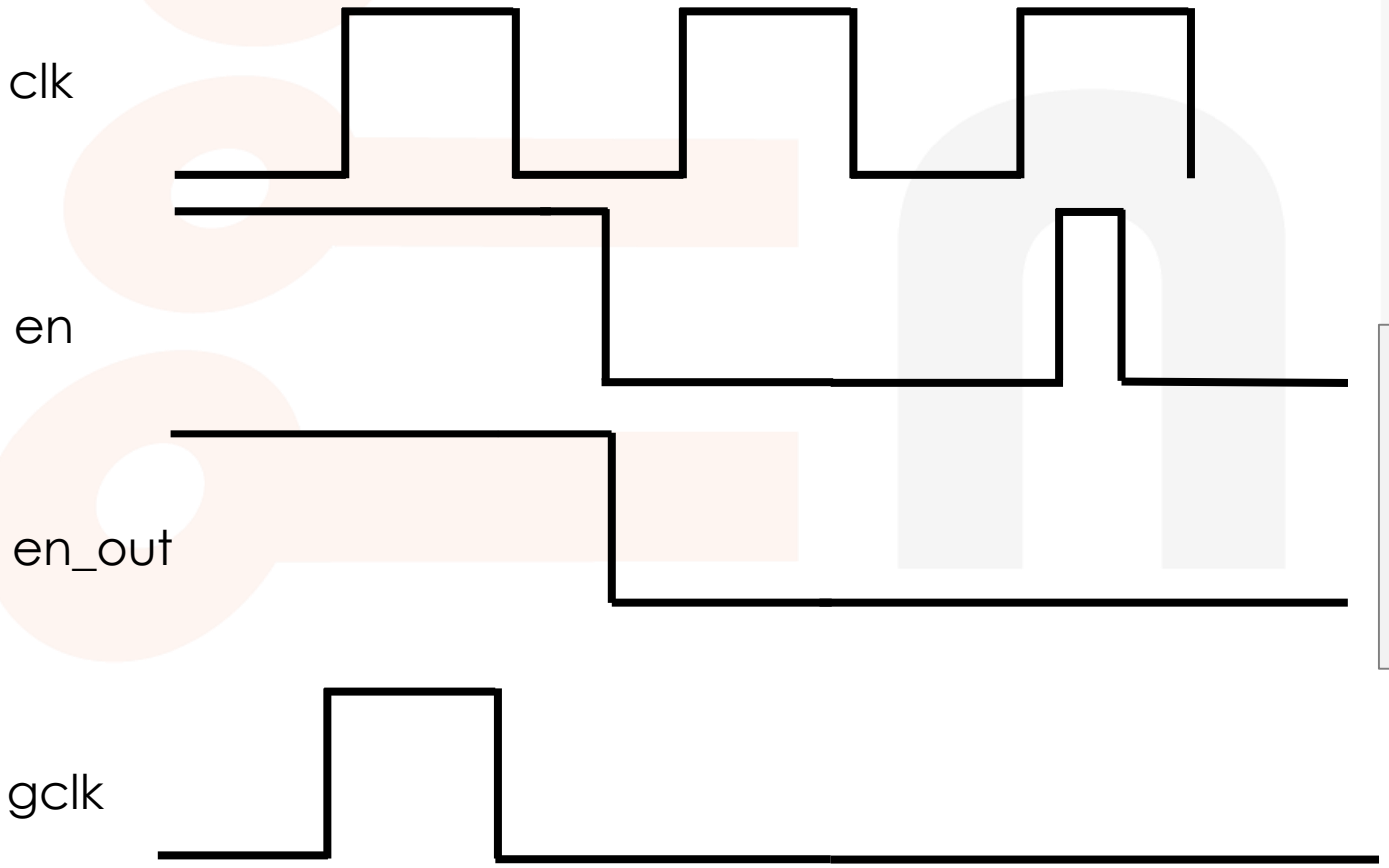
Ah, we live in a perfect world! ☺

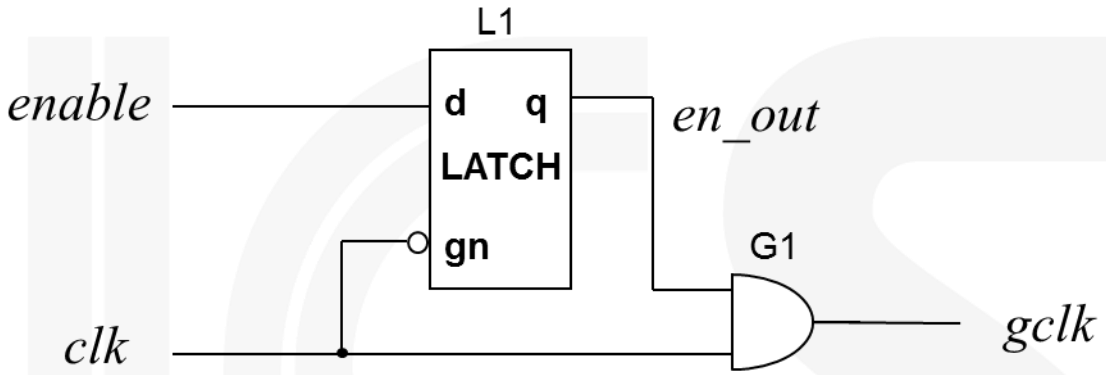
Not so Fast!

Maybe the world aint so perfect after all…

Solution: Glitch-free Clock Gate

- **By latching the enable signal during the positive phase, we can eliminate glitches:**

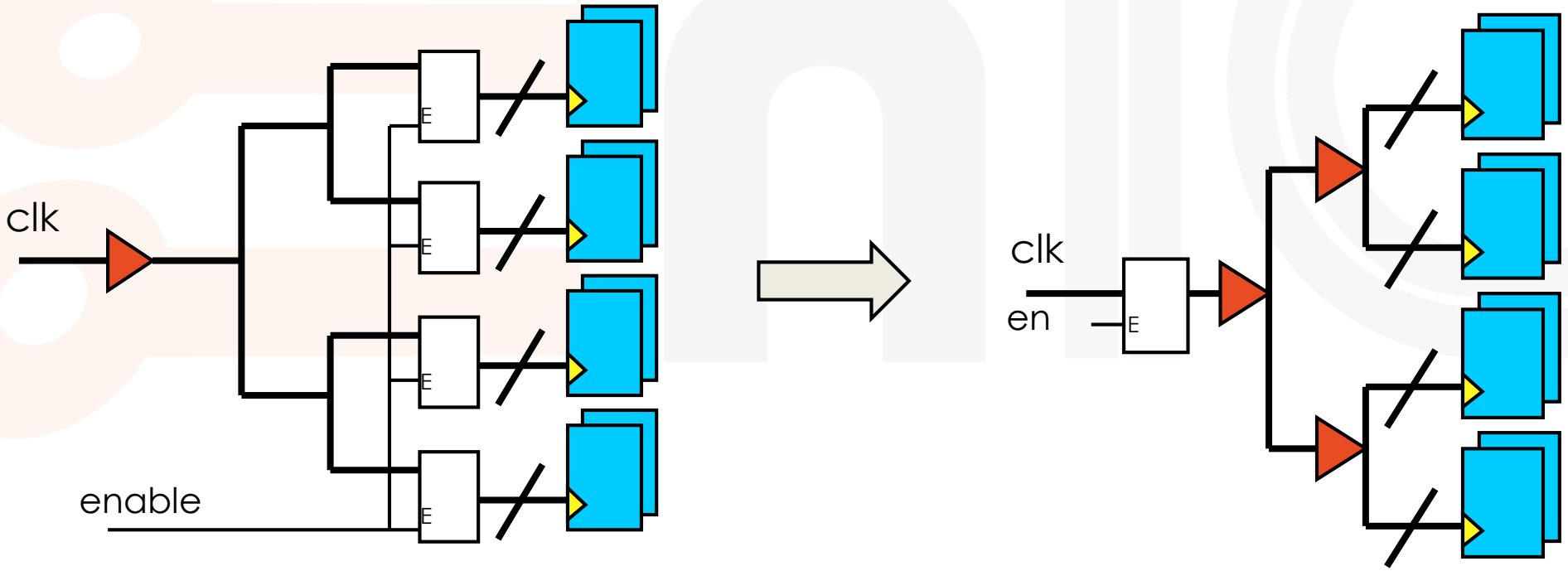





```
//clock gating with glitch prevention latch
always @ (enable or clk)
  begin
     if (!clk)
       en_out <= enable;
  end
assign gclk = en_out && clk;
```

Merging clock enable gates

- **Clock gates with common enable can be merged**
  - Lower clock tree power, fewer gates
  - May impact enable signal timing and skew.

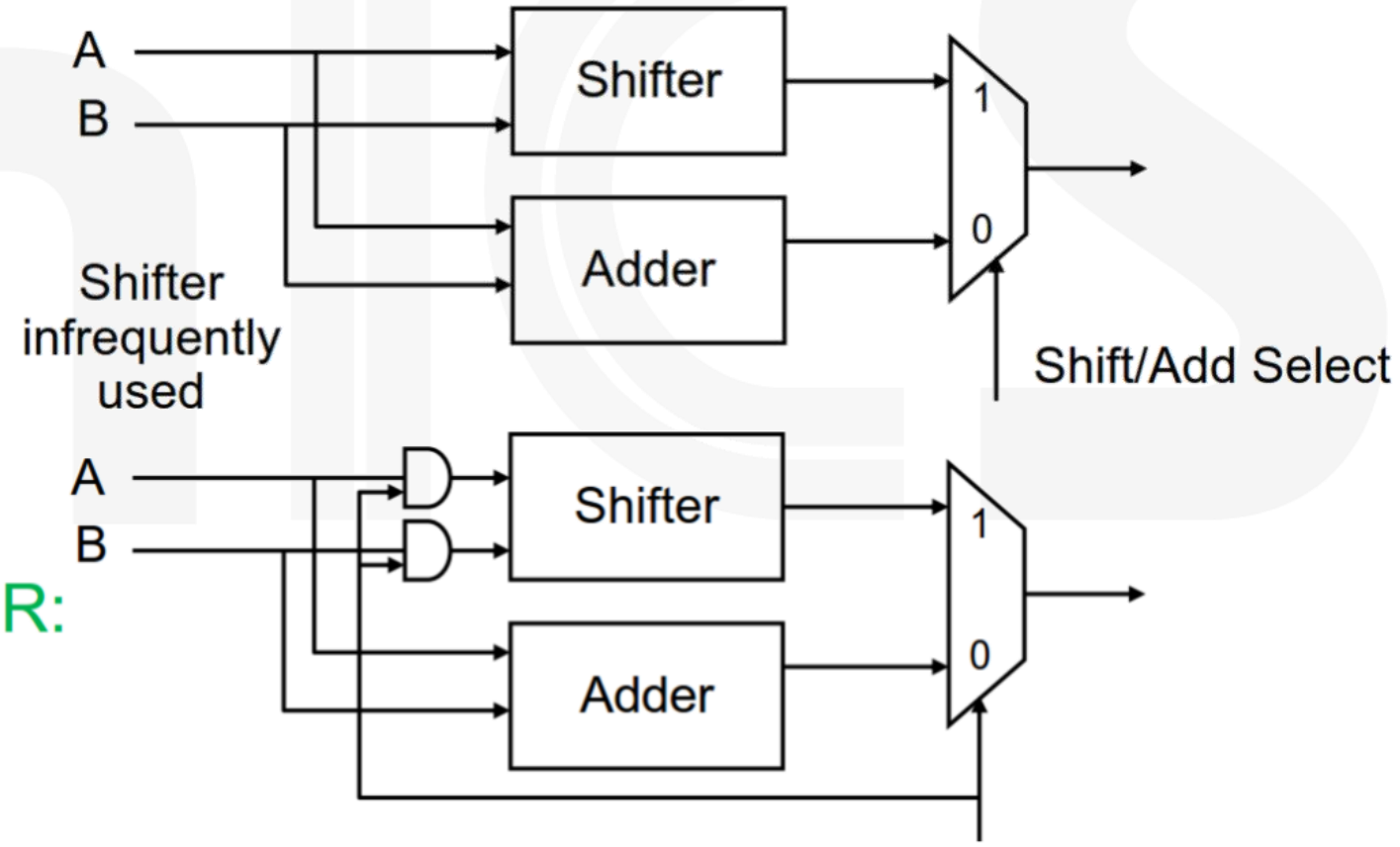


Data Gating

- While clock gating is very well understood and automated, a similar situation occurs due to the toggling of data signals that are not used.
- These situations should be recognized and data gated.

BAD:

```
assign add_out = A+B;
assign shift_out = A<<B;
assign out = shift_add ? shift_out : add_out;
```



BETTER:

```
assign shift_in_A = A && shift_add;
assign shift_in_B = B && shift_add;
assign shift_out = shift_in_A << shift_in_B;
assign out = shift_add ? shift_out : add_out;
```

Design and Verification – HDL Linting

- **HDL Linting tools provide a quick easy check of likely coding inconsistencies:**
  - Simulation problems
  - Synthesis Problems
  - Simulation Synthesis mismatches
  - Clock gating
  - Latch inference
  - Clock Domain Crossing issues
  - Nonsensical assignments / implicit bit widths issues
- **Not for checking syntactic correctness**
  - Use your simulator for that.
    (Will generally be more helpful)
- **Alternatively some synthesis tools will give you basic lint warnings**
  - For simulation-synthesis mismatch errors

Simulation/Synthesis Miss-matches

```
always @(a)
    z = a & b;
```

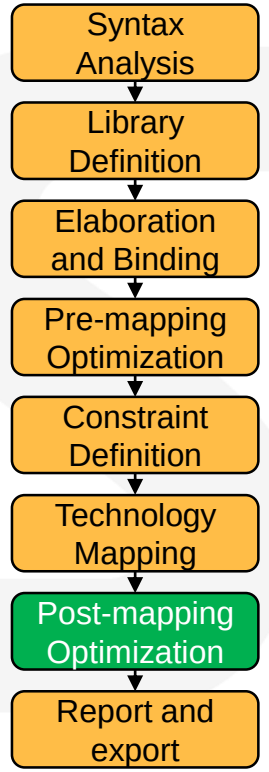
Latch Inference

```
always @(a or b or c)
    if (c) z = a & b;
```

Clock Gating

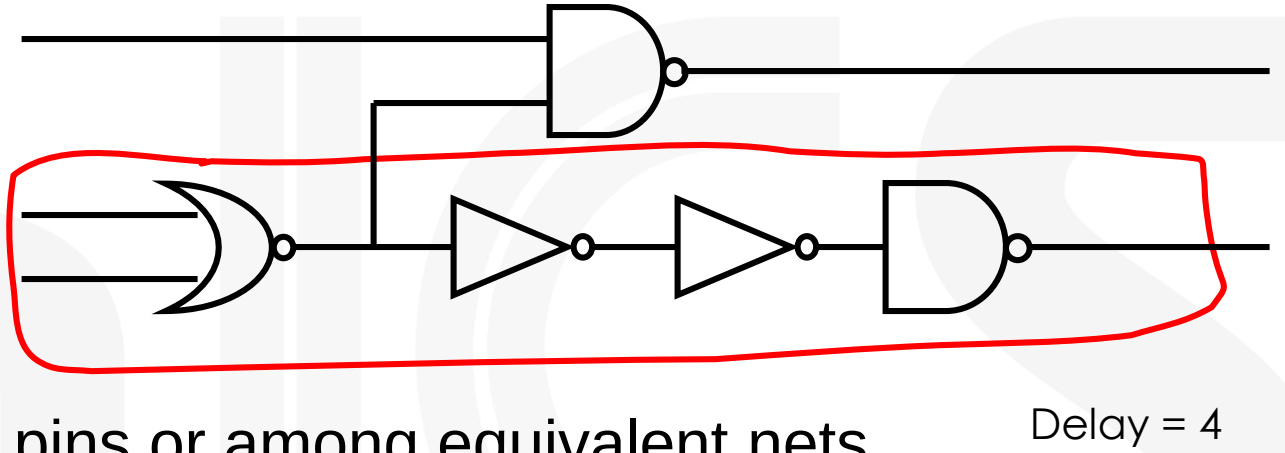
```
assign clka = clk & cond;
always @(posedge clka)
    z <= a & b;
```

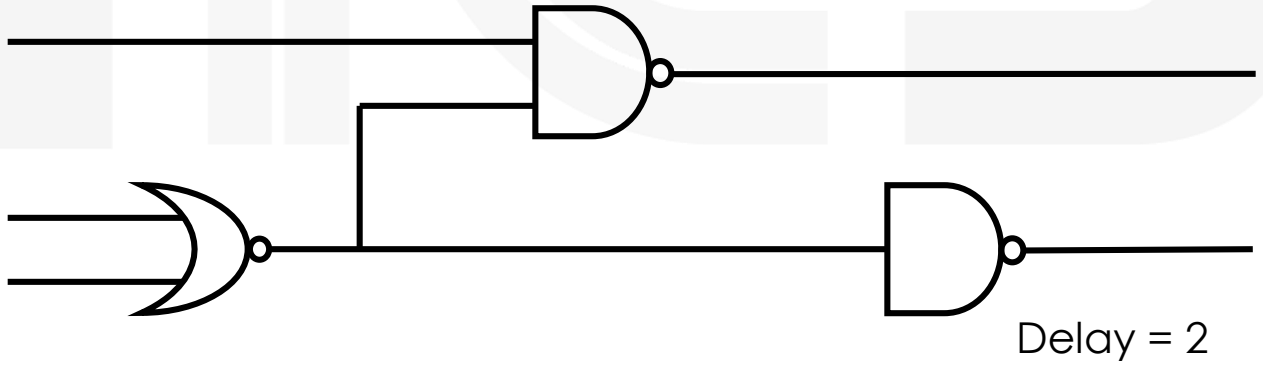
Timing Optimization



How can we optimize timing?

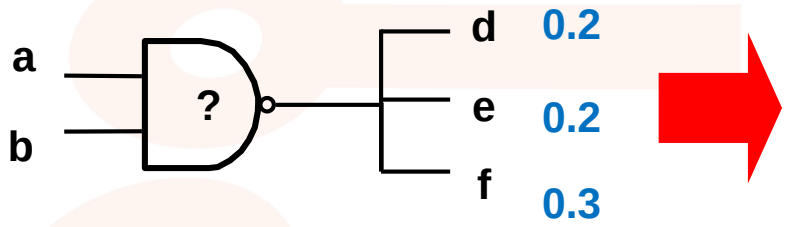
- **There are many 'transforms' that the synthesizer applies to the logic to improve the cost function:**
  - Resize cells
  - Buffer or clone to reduce load on critical nets
  - Decompose large cells
  - Swap connections on commutative pins or among equivalent nets
  - Move critical signals forward
  - Pad early paths
  - Area recovery
- **Simple example:**
  - Double inverter removal transform:

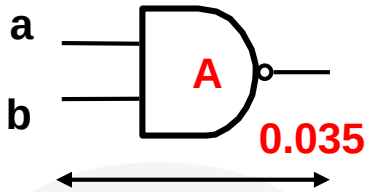


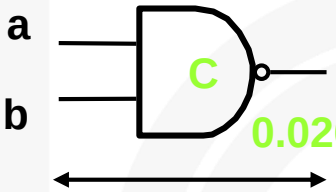


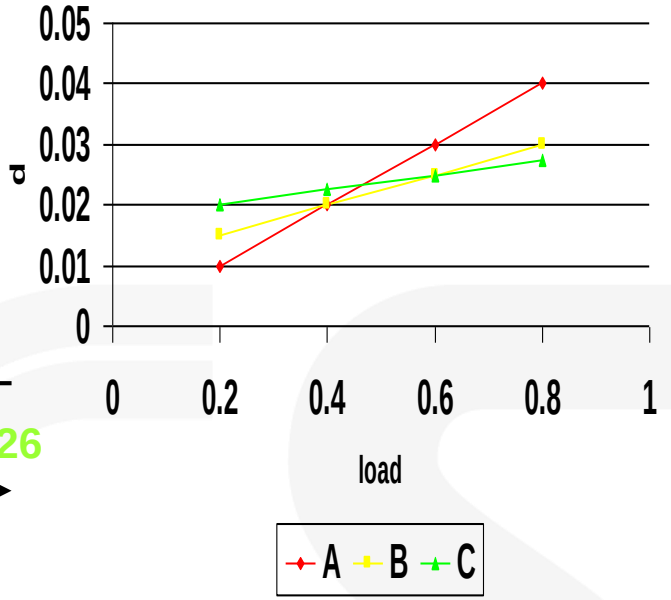
Resizing, Cloning and Buffering

- Resize a logic gate to better drive a load:

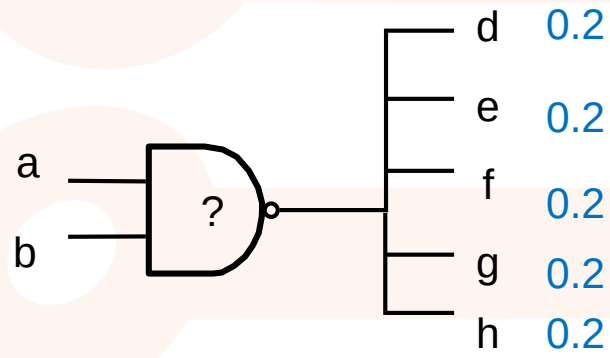


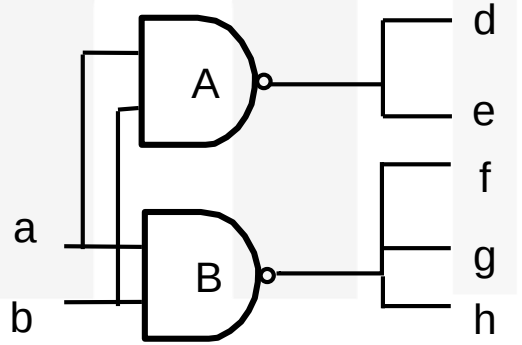




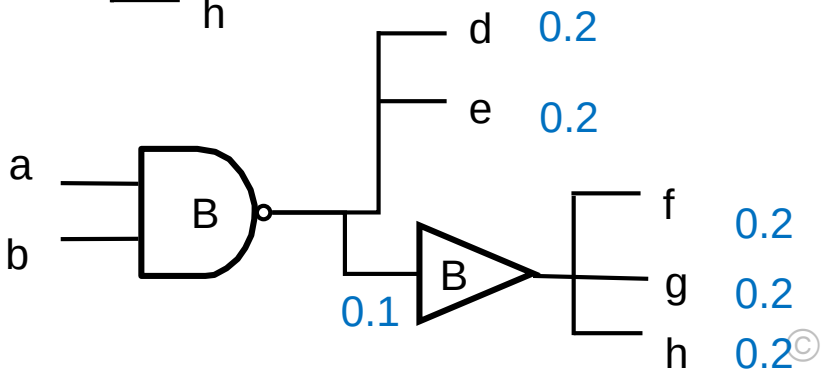


- Or make a copy (clone of the gate) to distribute the load:



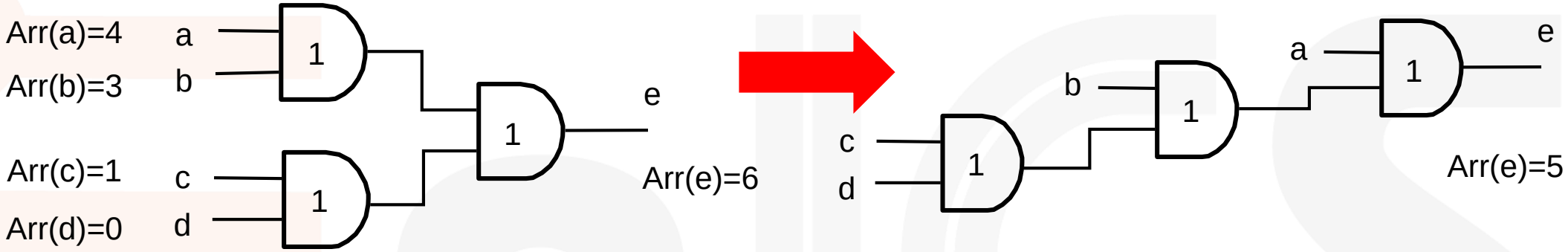


- Or just buffer the fanout net:

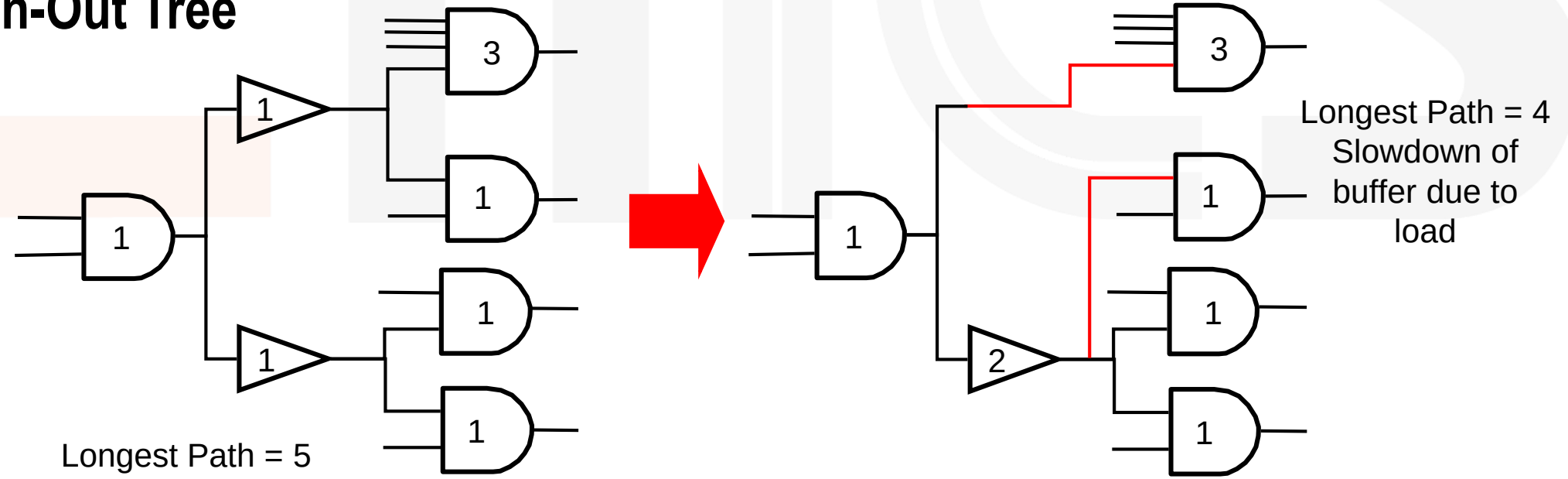


Redesign Fan-In/Fan-out Trees

- **Redesign Fan-In Tree**

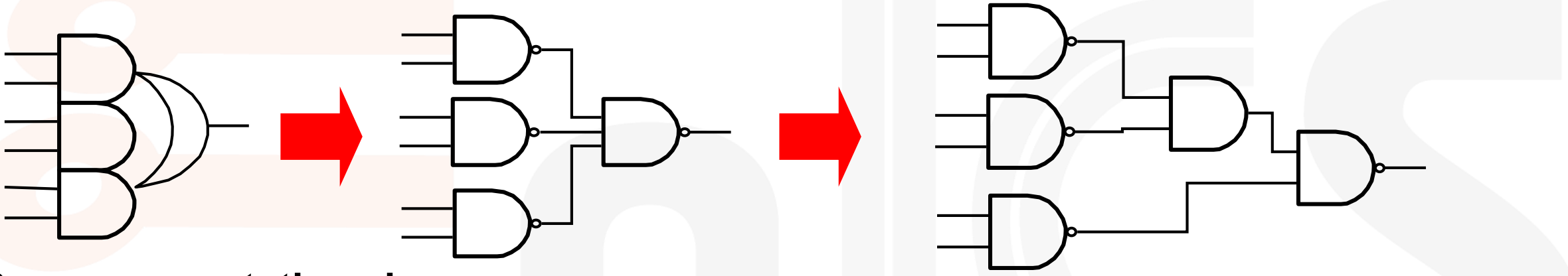


- **Redesign Fan-Out Tree**

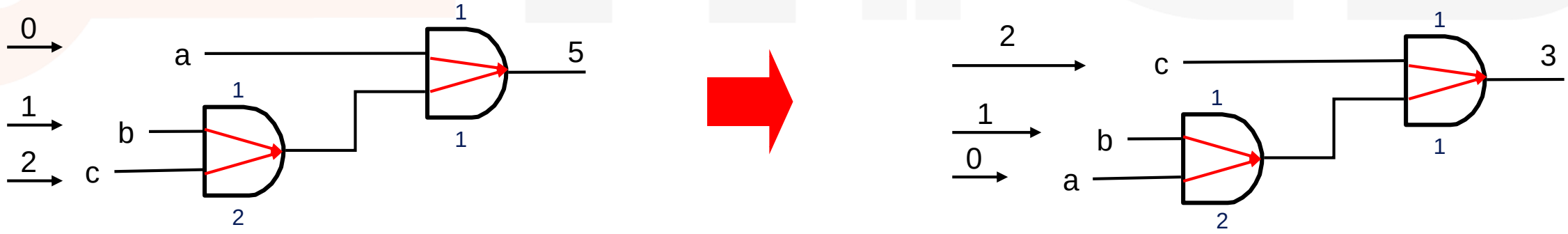


Decomposition and Swapping

- **Consider decomposing complex gates into less complex ones:**

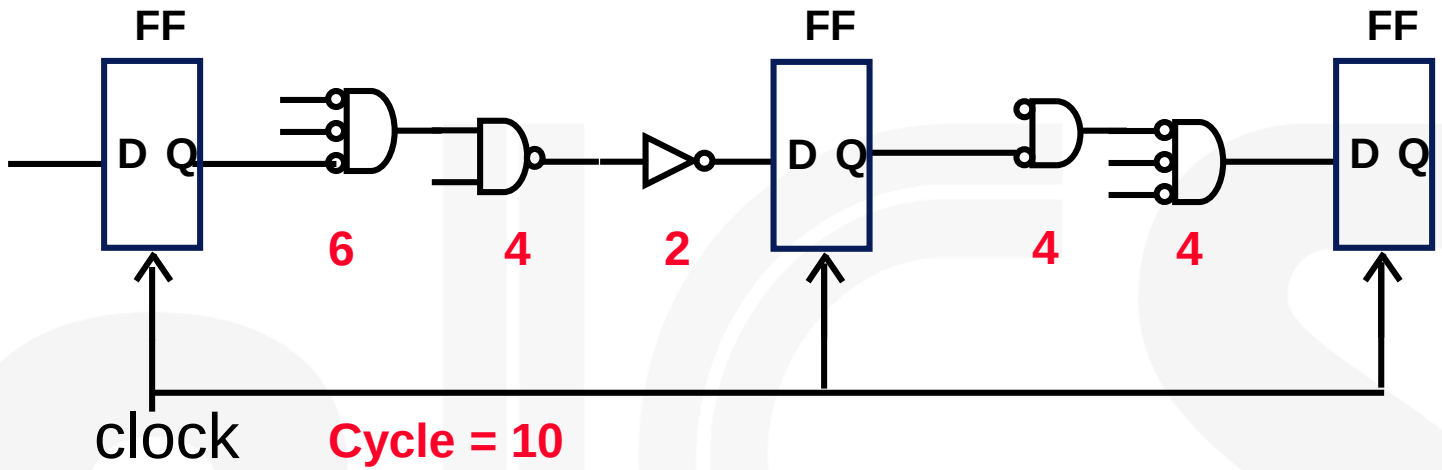


- **Swap commutative pins:**
  - Simple sorting on arrival times and delays can help

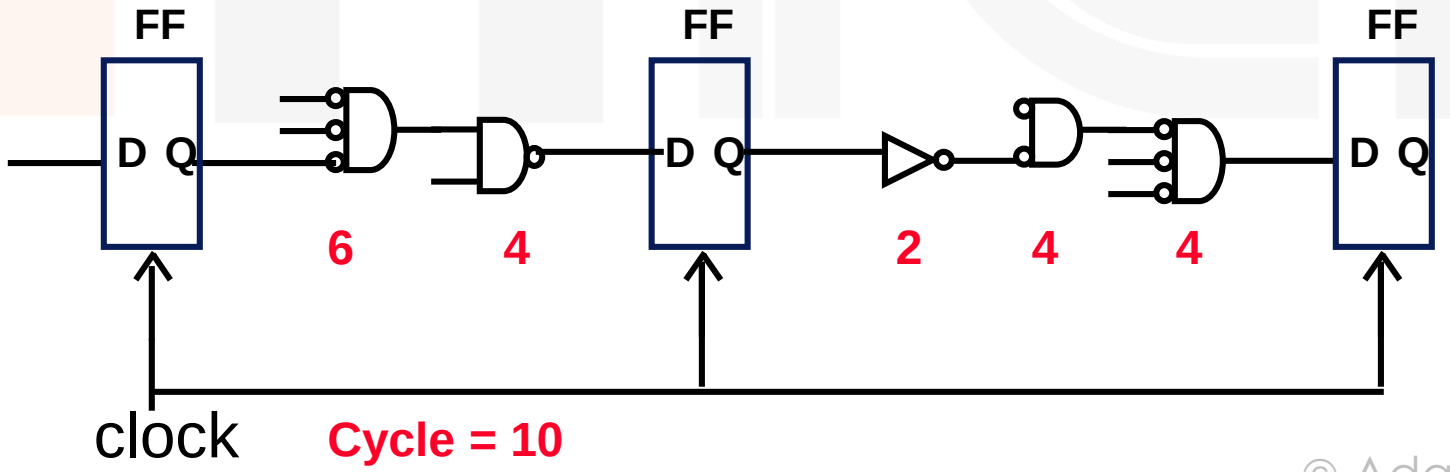


Retiming

- **Given the following network:**



  - How would you meet the $10ns$ clock cycle time?
  - Re-order sequential elements and combinational logic



Main References

- Rob Rutenbar “From Logic to Layout”
- IDESA
- Rabaey, “Low Power Design Essentials”
- vlsicad.ucsd.edu ECE 260B – CSE 241A
- Roy Shor, BGU
- Synopsys slides