CAL POLY
SAN LUIS OBISPO

EE 531: ADVANCED VLSI DESIGN

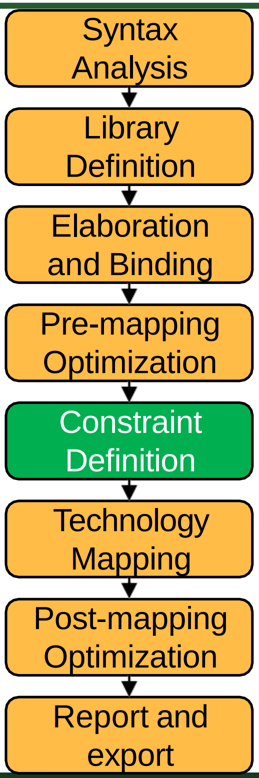
Logic Synthesis: Part 2

Nishith N. Chakraborty

January, 2025

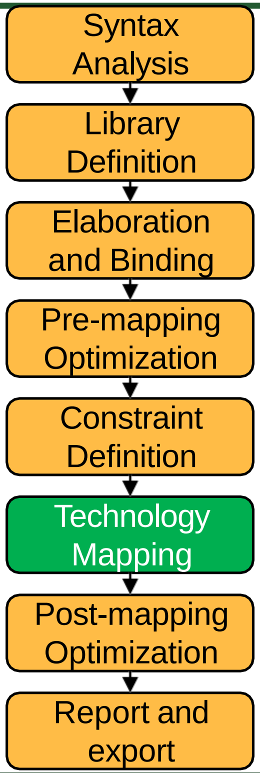
CONSTRAINT DEFINITION

- 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.
- At this point, we can load the design constraints in SDC format.
- Carefully check that all constraints were accepted by the tool!



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

STEP 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:

$t_1 = d + e;$
$t_2 = b + h;$
$t_3 = at_2 + c;$
$t_4 = t_1t_3 + fgh;$
$F = t_4';$

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

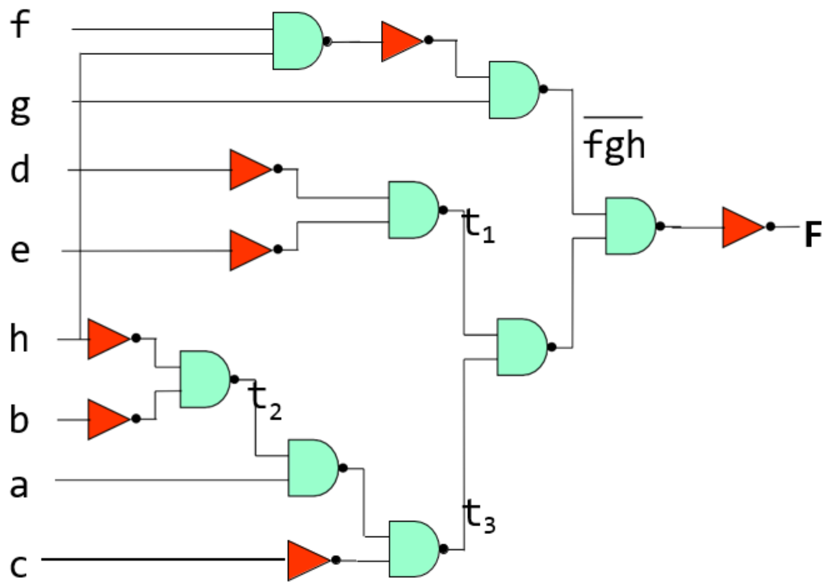
$$t_2 = b + h = \text{NAND}\left(\bar{b}, \bar{h}\right)$$

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

$$t_4 = t_1t_3 + fgh = \text{NAND}\left(\overline{t_1t_3}, \overline{fgh}\right)$$

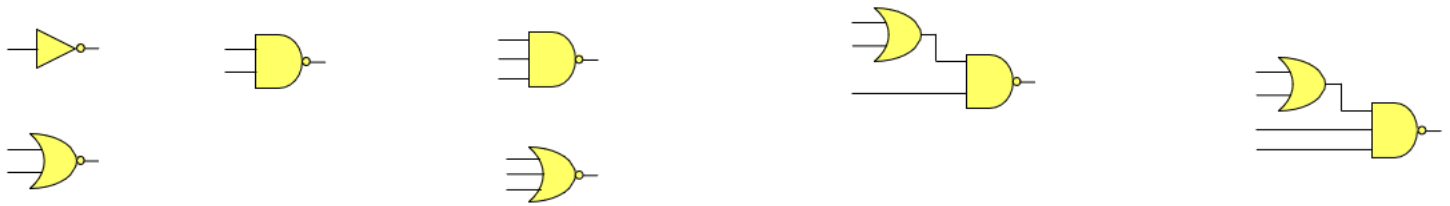
$$fgh = \overline{\overline{fh}} \cdot g = \overline{\overline{\overline{\overline{fh}} \cdot g}} = \overline{\text{NAND}\left(\overline{\text{NAND}(f, h)}, g\right)}$$

$$F = \overline{t_4}$$

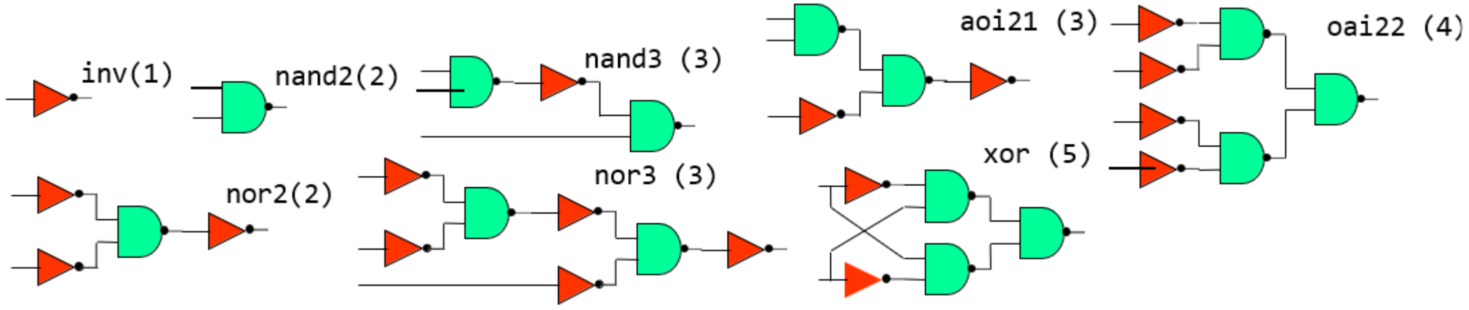


STEP 1: SIMPLE GATE MAPPING

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

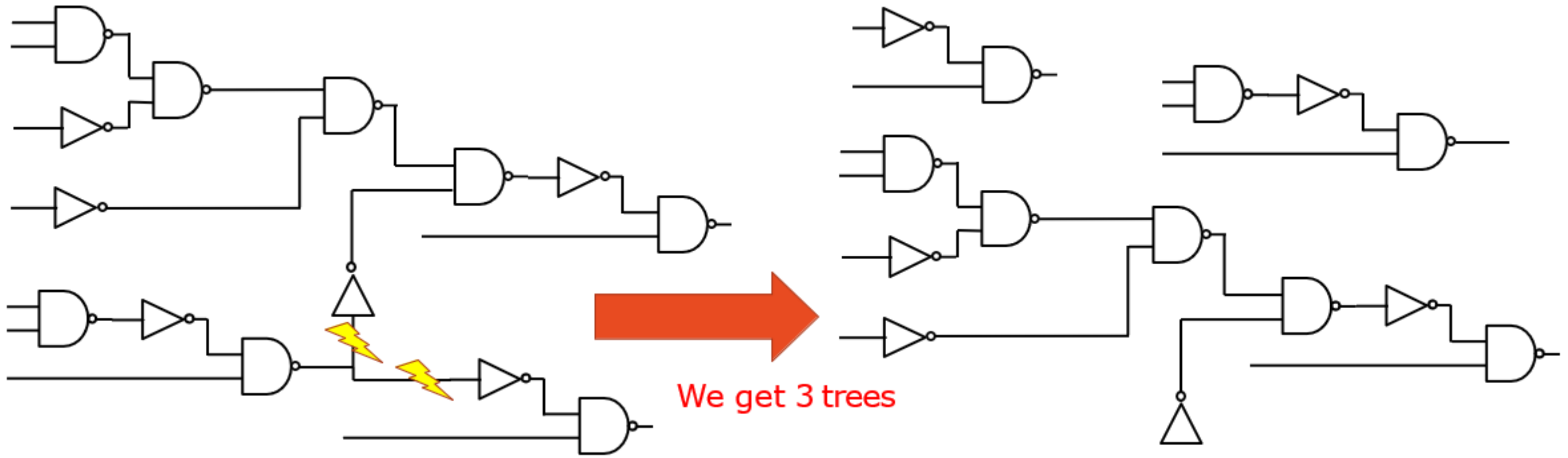


- We need to define the gates with the same NAND2/NOT set:



STEP 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

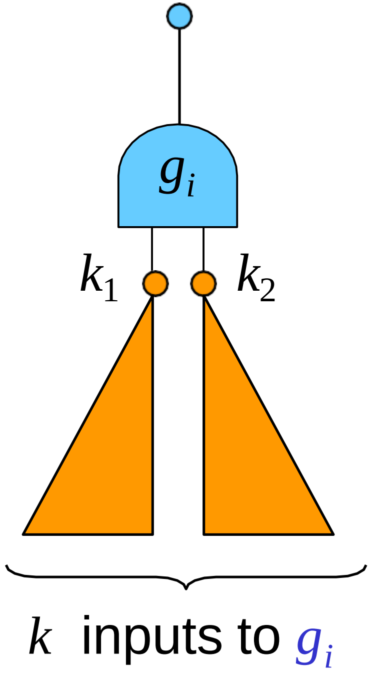


STEP 3: MINIMUM TREE COVERING

- 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:

$$\text{cost}\left(i\right)=\min_{\text{k}}\left\{\text{cost}\left(g_i\right)+\sum\ \text{cost}\left(k_i\right)\right\}$$

  - where k_i are the inputs to gate *g*.

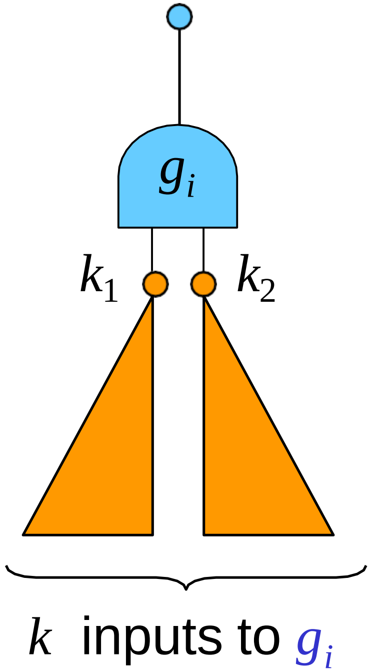


STEP 3: MINIMUM TREE COVERING

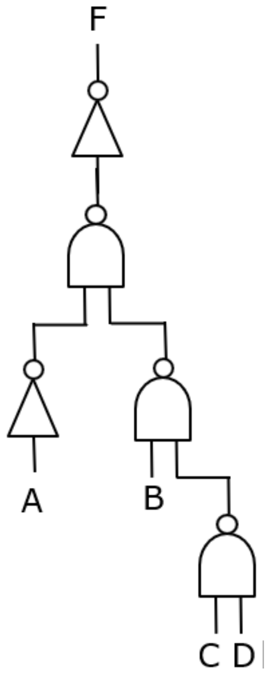
- Now, we can apply a recursive algorithm to achieve a minimum cover:
  - Start at the output of the graph.
  - For each node, find all the matching target patterns.
  - The cost of node *i* for using gate *g* is:

$$\mathrm{cost}(i) = \min_{k}\left\{\mathrm{cost}(g_i) + \sum \mathrm{cost}(k_i)\right\}$$

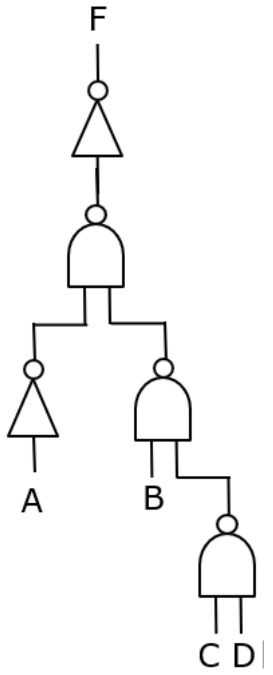
  - where k_i are the inputs to gate *g*.
- For simplicity, we will redraw our graph and show an example:
  - Every NOT is just an empty circle: $\textcircled{\textbf{I}}$
  - Every NAND is just a full circle: ● N
  - Every input is just a box: $\boxed{\mathbf{A}}$

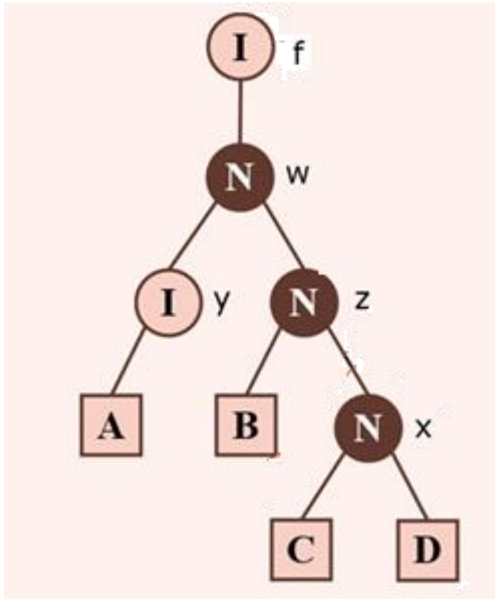


MINIMUM TREE COVERING: EXAMPLE

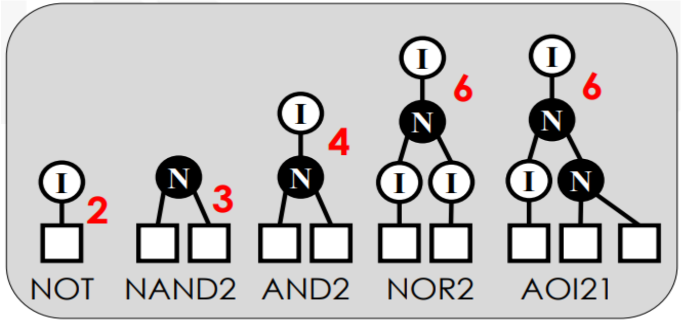


MINIMUM TREE COVERING: EXAMPLE

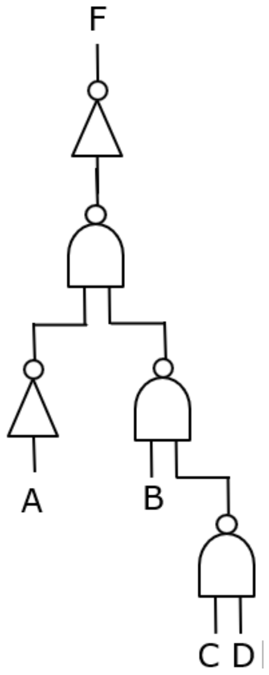


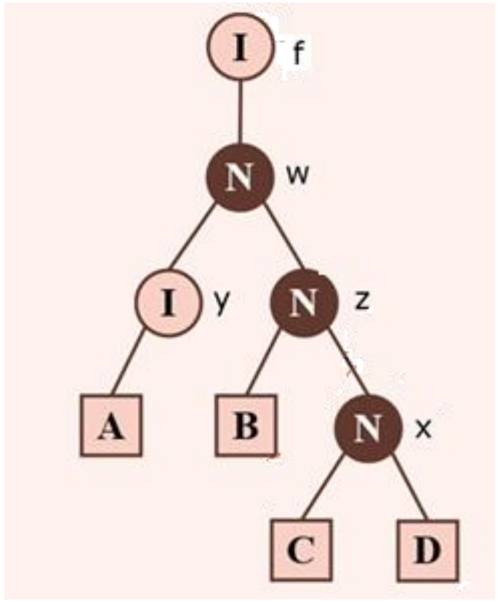


f:	NOT	2 + min(w)
	AND2	4 + min(y)+min(z)
	AOI21	6 + min(x)



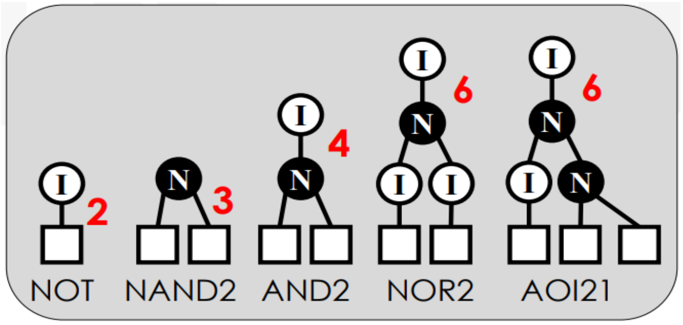
MINIMUM TREE COVERING: EXAMPLE



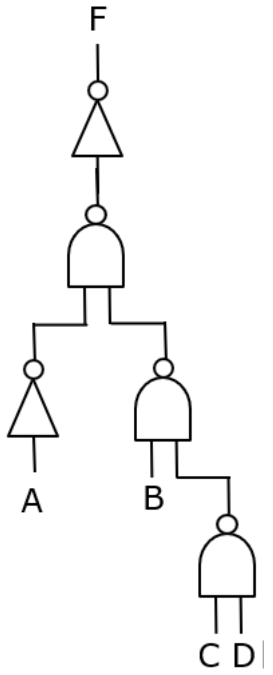


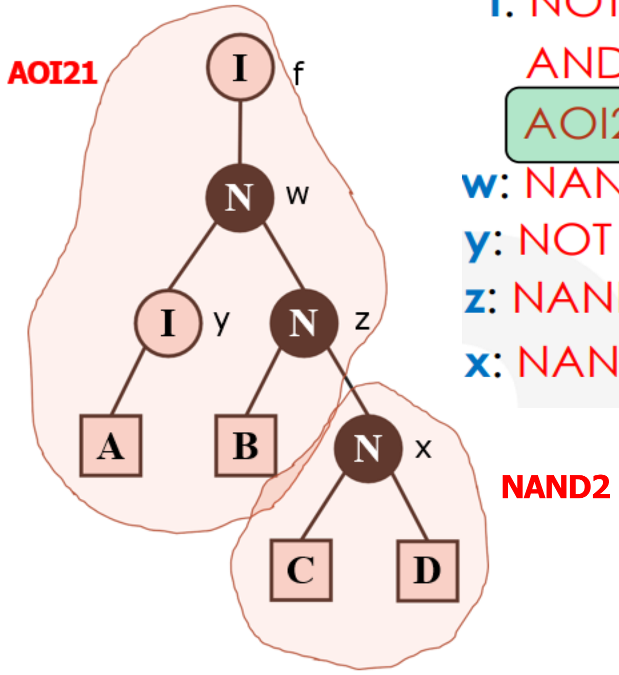
f: NOT $2 + \min(w)$
AND2 $4 + \min(y) + \min(z)$
AOI21 $6 + \min(x)$

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



MINIMUM TREE COVERING: EXAMPLE





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

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

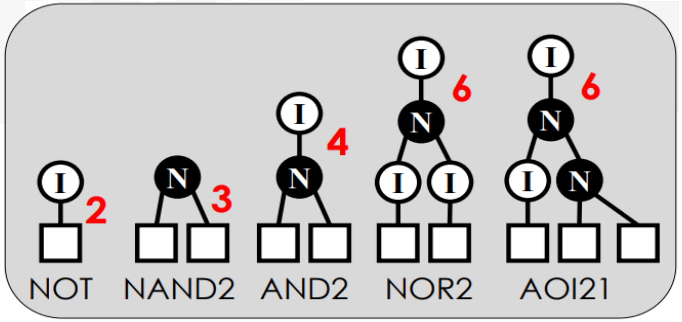
AOI21 6 + min(x) = 6 + 3 = 9

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

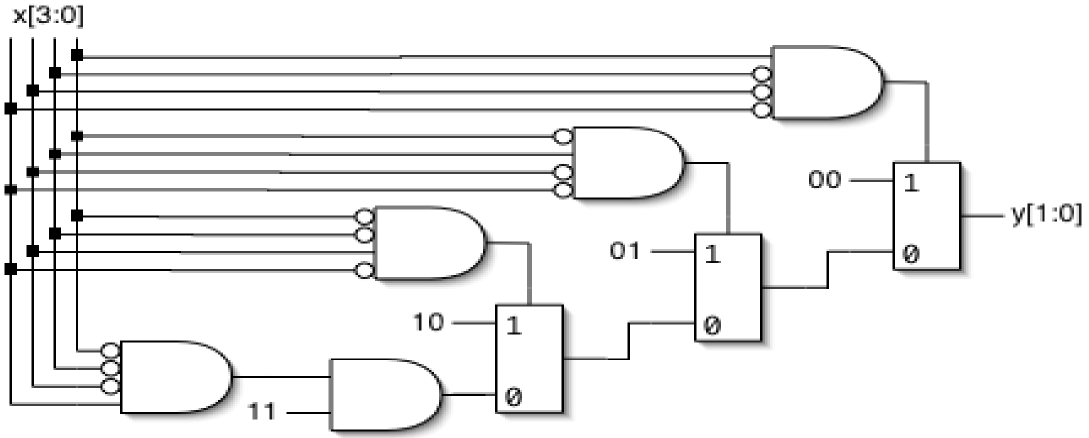


Verilog for Synthesis: Revisited

VERILOG TO CIRCUIT

- 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:
    - The result is known as “priority logic”

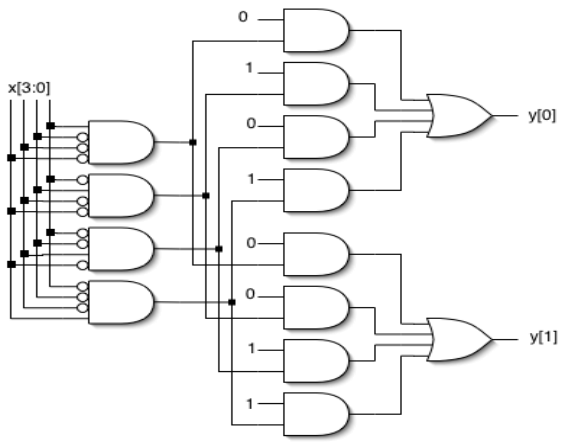
```
always @(x)
  begin : encode
    if      (x == 4'b0001) y = 2'b00;
    else if (x == 4'b0010) y = 2'b01;
    else if (x == 4'b0100) y = 2'b10;
    else if (x == 4'b1000) y = 2'b11;
    else y = 2'bxx;
  end
```

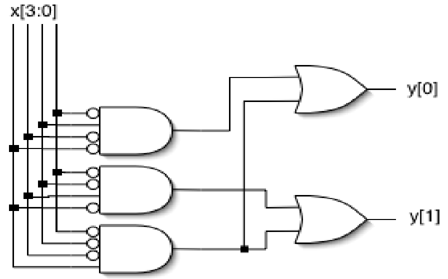


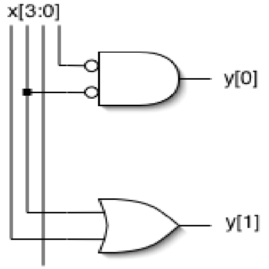
VERILOG TO CIRCUIT

- 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
```

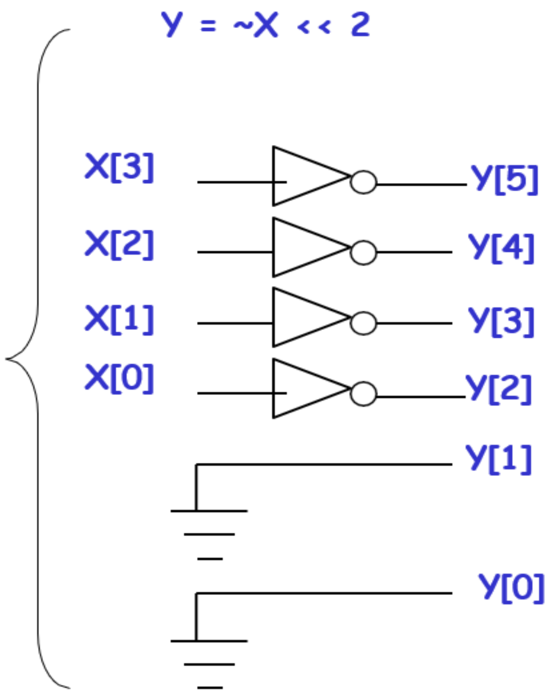






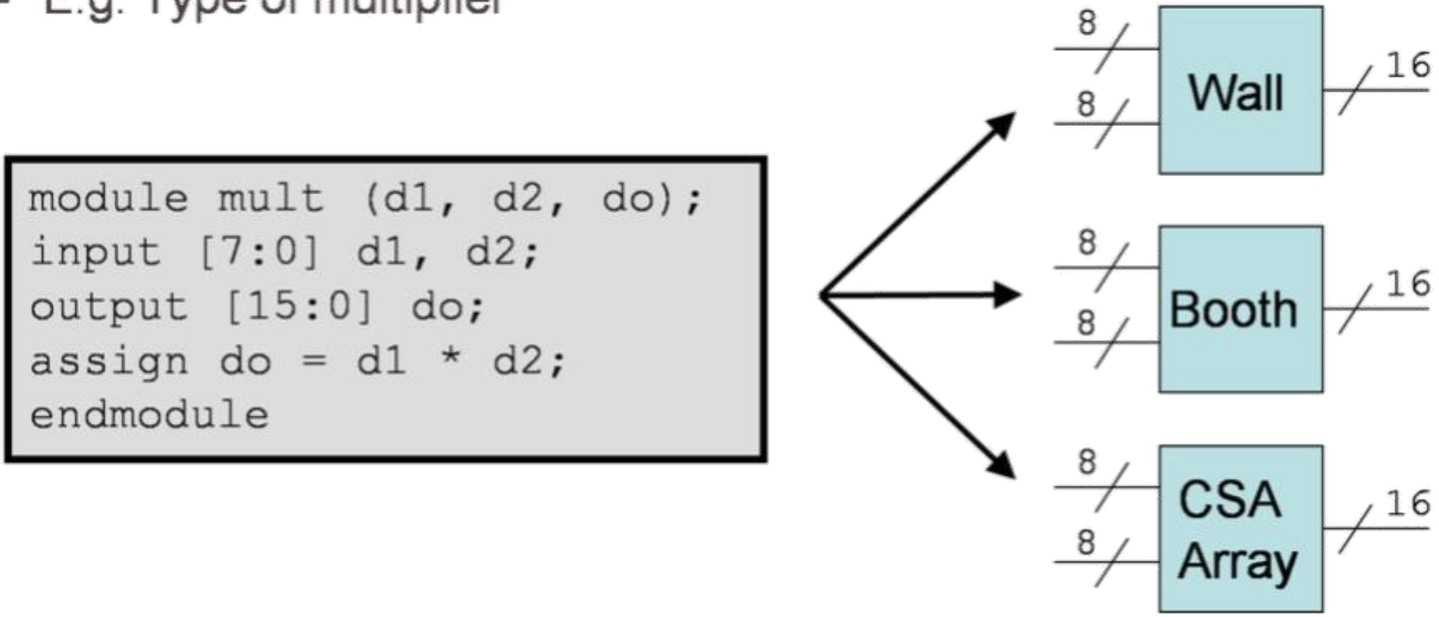
A FEW POINTS ABOUT OPERATORS

- 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



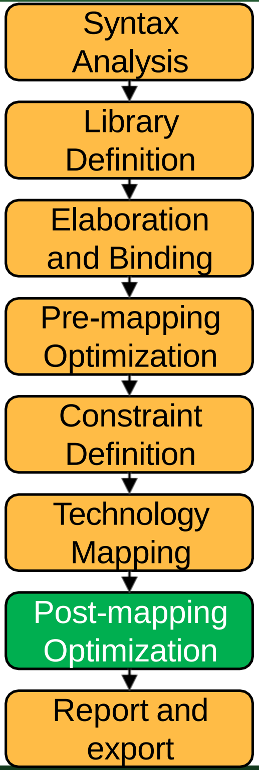
DATAPATH SYNTHESIS

- 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
```

OPTIMIZATION



OPTIMIZATION OPTIONS FOR SYNTHESIS

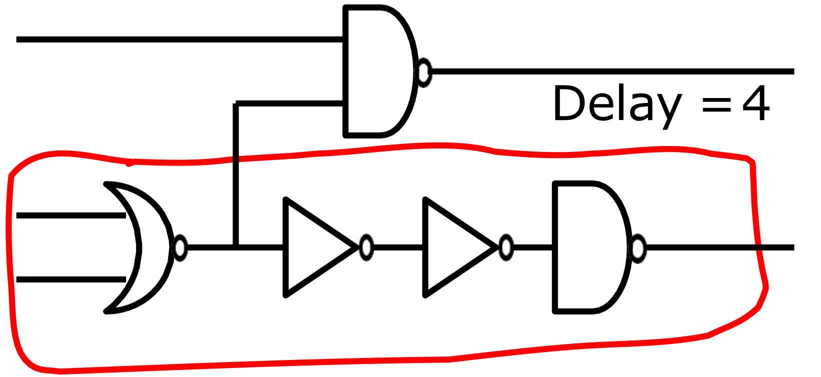
- Another look at RTL to physical layout (GDSII*) design flows
- Metrics used for optimization
  - Speed
  - Area
  - Power
  - Others?
- ASIC/SoC design choices that allow optimizations

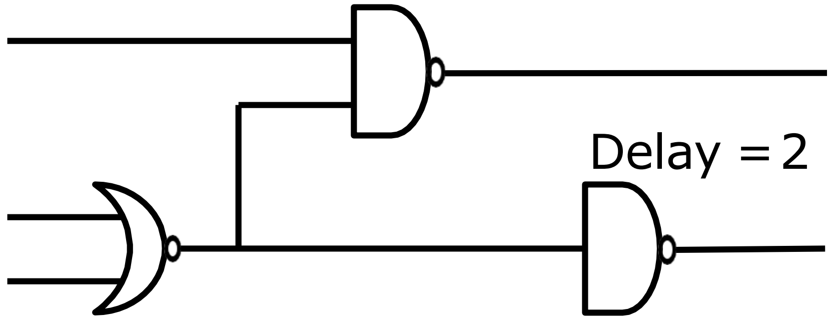
TIMING DRIVEN SYNTHESIS

- Traditionally, timing has been one of the major drivers in ASIC/SoC design
- The goal is to minimize the delay on the *critical path* of the design so that the frequency is maximized
- CAD tools must include accurate and robust models for estimating the delay through the circuit(s)
- During synthesis, several choices can be made to reduce delay:
  - Optimize number of logic levels
  - Logic family used (if in standard cell library)
  - Cell sizing (usually have multiple sized std. cells)

HOW TO OPTIMIZE TIMING?

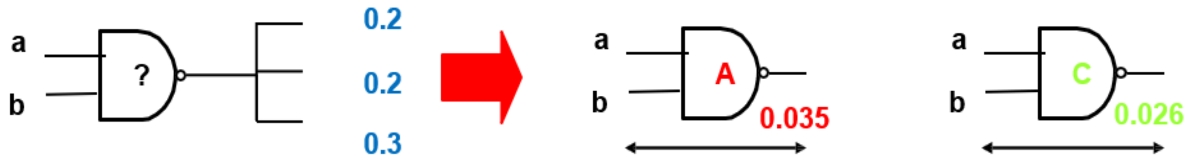
- There are many 'transforms' that the synthesizer applies to the logic to improve the cost function:
  - Resize cells
  - Buffer or clone to reduce load on critical nets
  - Decompose large cells
  - Swap connections on commutative pins or among equivalent nets
  - Move critical signals forward
  -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

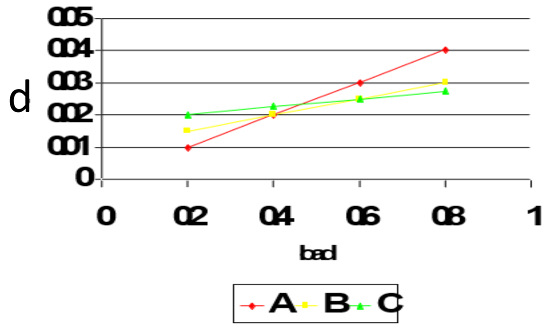




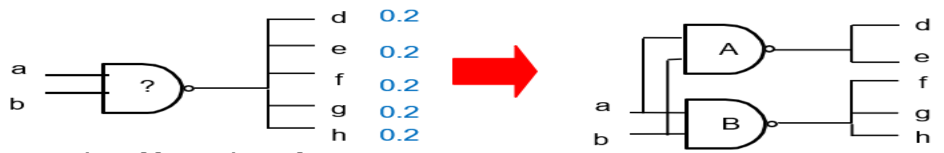
RESIZING, CLONING AND BUFFERING

- Resize a logic gate to better drive a load:

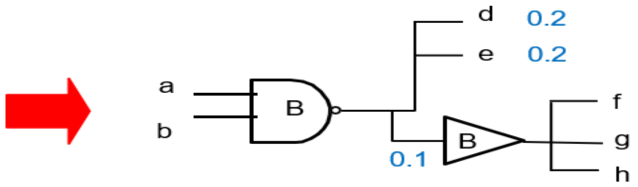




- Or make a copy (clone of the gate) to distribute the load:



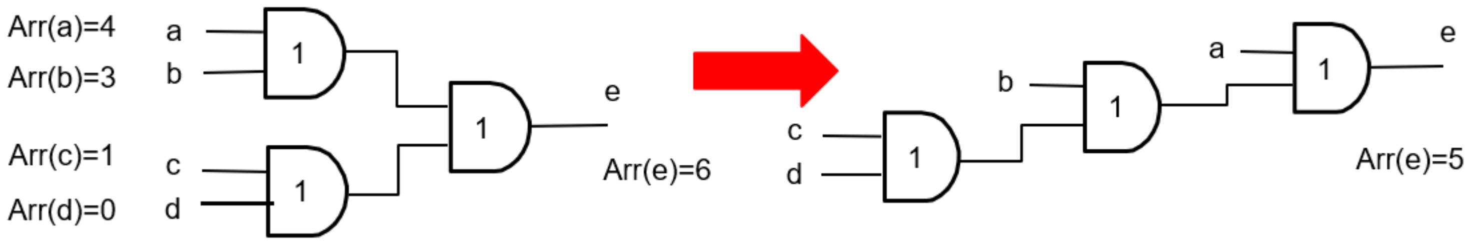
- Or just buffer the fanout net:



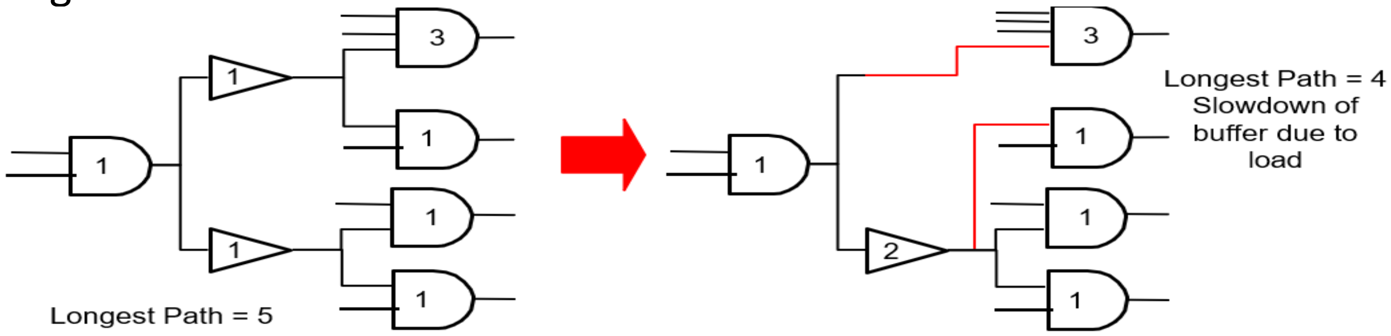
CAL POLY
SAN LUIS OBISPO

REDESIGN FAN-IN/FAN-OUT TREES

- Redesign Fan-In Tree

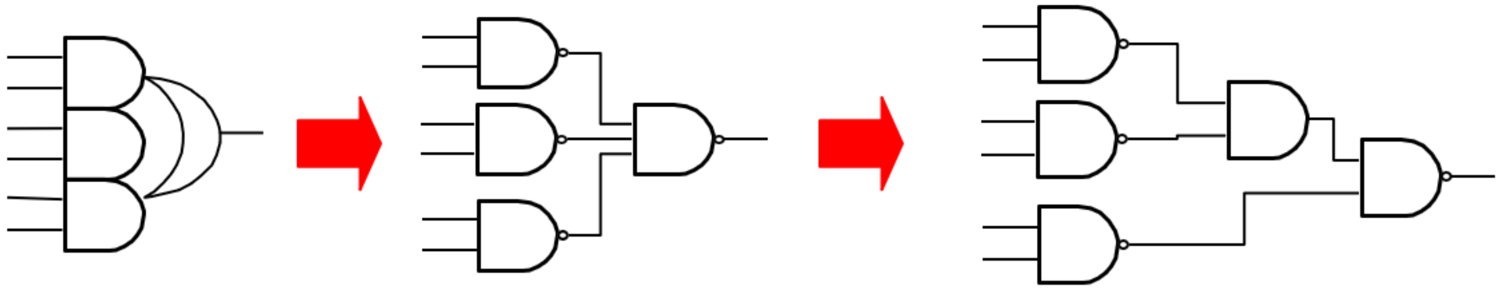


- Redesign Fan-Out Tree

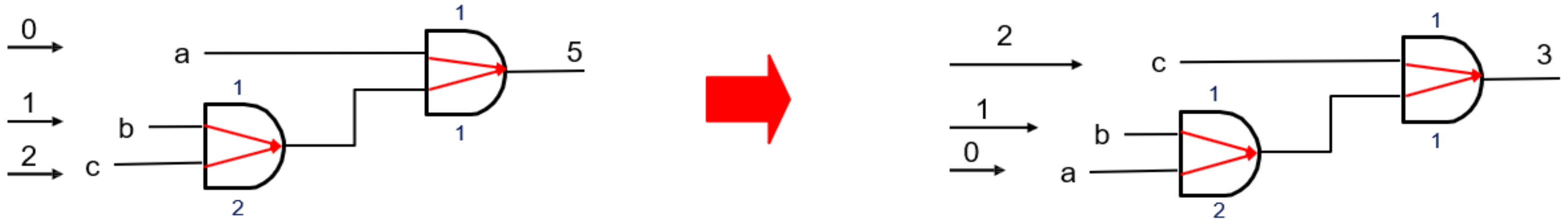


DECOMPOSITION AND SWAPPING

- Consider decomposing complex gates into less complex ones:



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



WORST NEGATIVE SLACK

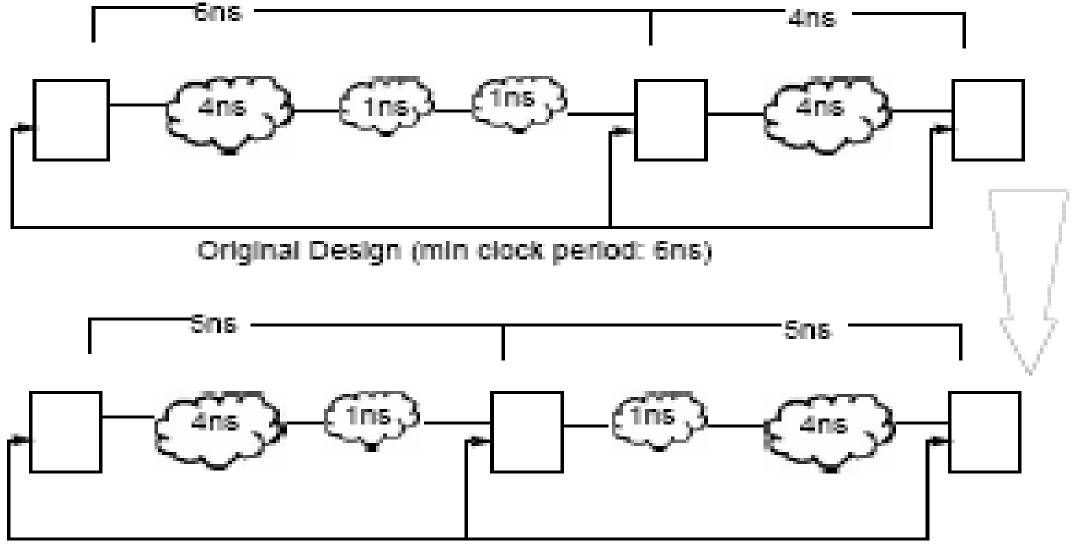
- Design Vision (and many other tools) uses Worst Negative Slack (WNS) to achieve timing requirements
- Slack = Design Delay – Predicted Delay
    - ➢ Design Frequency is essentially the target which is usually higher than Market Frequency
    - ➢ Predicted Frequency is the frequency of the current design determined by low-level simulation
- Negative slack occurs when the design does not meet the timing requirements
- Worst Negative Slack refers to the critical path, the path with the most delay

RETIMING THE DESIGN

- Retiming: technique for improving performance of sequential circuits by repositioning registers
  - Reduces cycle time or area with no I/O latency change
- Pipelining is a subset of retiming
- Retiming redistributes sequential elements at appropriate locations to meet requirements
- Retiming does not change combinational logic

RETIMING FOR TIMING

- Improving clock period or timing slack common
- Design Compiler distributes the registers within the design to provide minimum cycle time

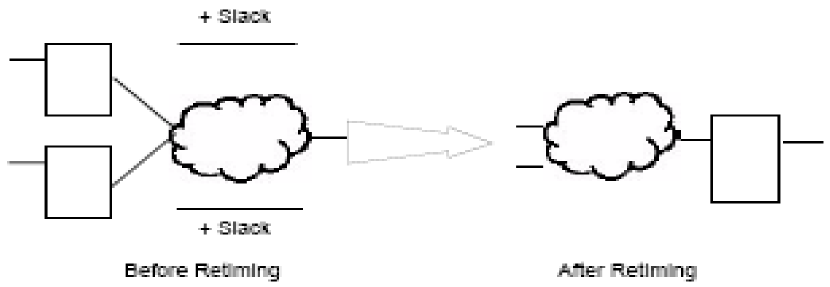


Original Design (min clock period: 6ns)

Retimed Design (min clock period:5ns)

RETIMING FOR AREA

- When retiming for area, Design Vision moves registers to minimize register count without worsening the critical path in the design



POWER AS A DESIGN METRIC

- Power determined by four major factors:
  - Capacitance being driven (*C*)
  - Voltage (V_{DD})
  - Frequency (*f*)
  - Activity factor (α)

$$P = \alpha \cdot C \cdot V_{DD}^2 \cdot f$$

- Low-power design techniques focus on these factors for controlling the power consumption of a design

LOW-POWER DESIGN TECHNIQUES

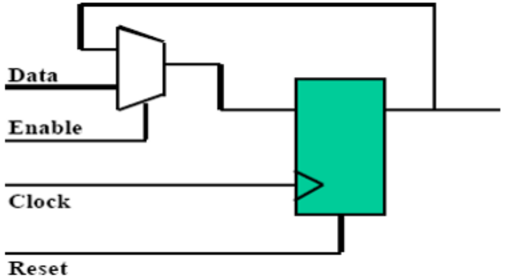
- Dynamic Voltage Scaling (DVS)
  - lower V_{DD} during runtime for quadratic savings
- Frequency scaling
- Sleep mode transistors

- Note: These are all techniques for reducing dynamic power; as technology scales, static power is becoming more of a concern

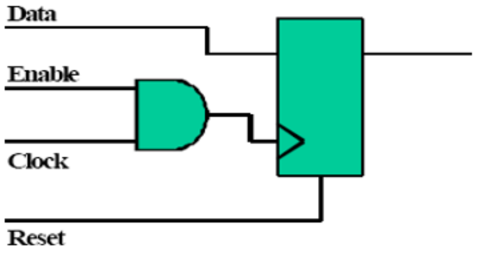
CAL POLY
SAN LUIS OBISPO

CLOCK GATING

- Dynamic power control through synthesis typically due to clock gating
- Usually this means shutting off the clock to flip flop(s)
- Example to the right:
  - Conceptually the same
  - Implementation 1 clocks the flip flop every cycle
  - Implementation 2 only clocks when enabled
    - the lower power design



Implementation 1

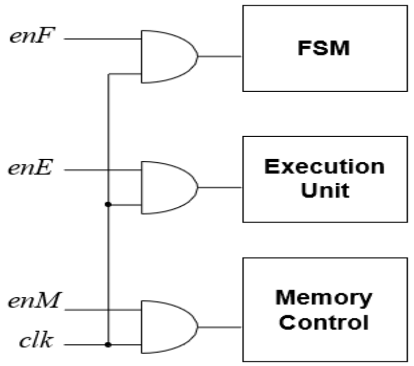


Implementation 2

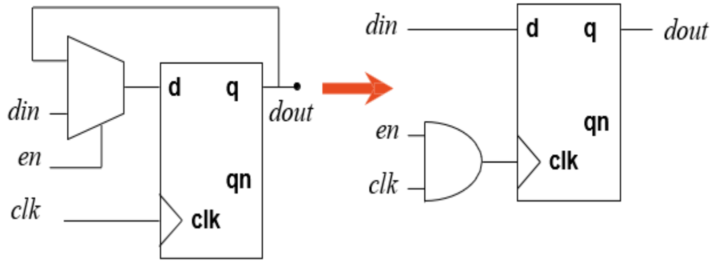
CLOCK GATING LEVELS

- 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 its 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.

Global Clock Gating



Local Clock Gating



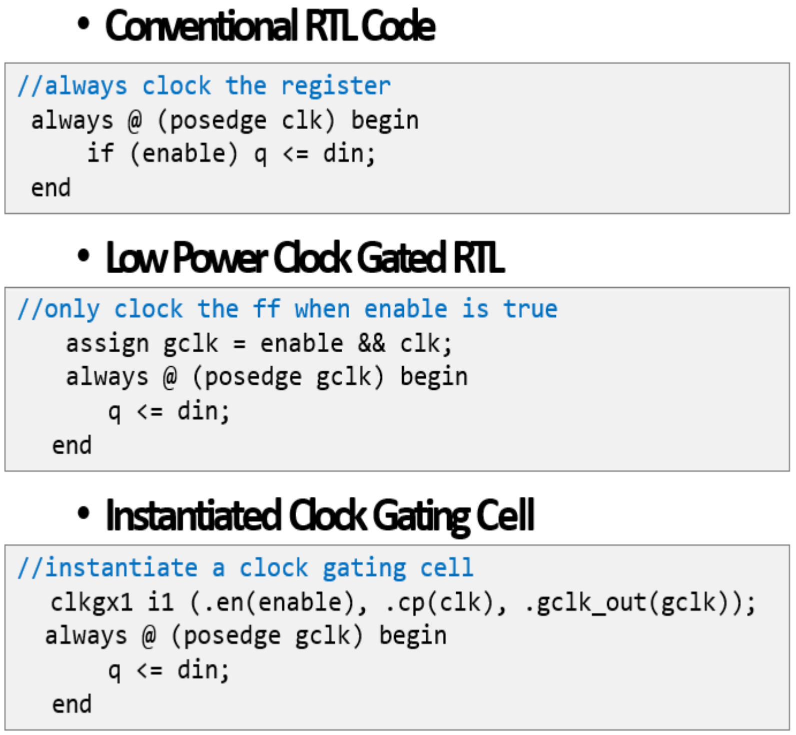
SYNTHESIZED CLOCK GATING

- To implement clock gating for power control during synthesis, tools analyze design at elaboration stage
- Most likely, gating structures are not applied to every register – cost in power of gating would exceed savings on the flip flops
- Synthesis tool tries to find gating enable signals within the design that can control the clock for a register bank

CLOCK GATING INSTANTIATION

- 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**

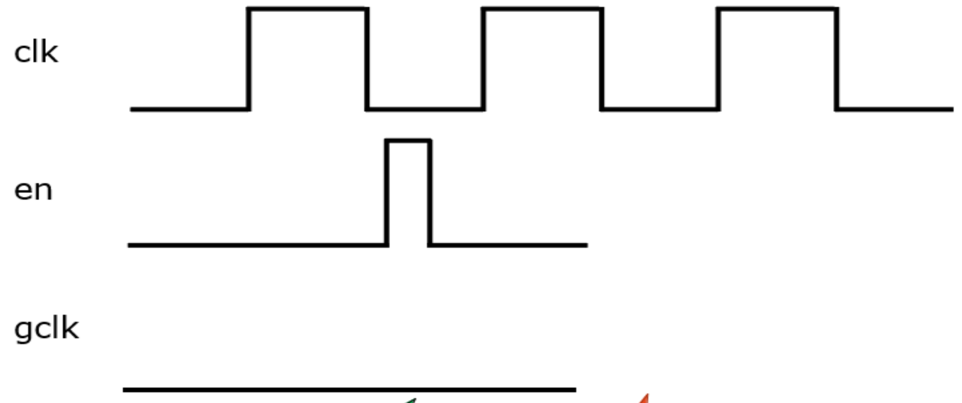
```
//only clock the ff when enable is true
   assign gclk = enable && clk;
   always @ (posedge gclk) begin
      q <= din;
  end
```

- **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
```

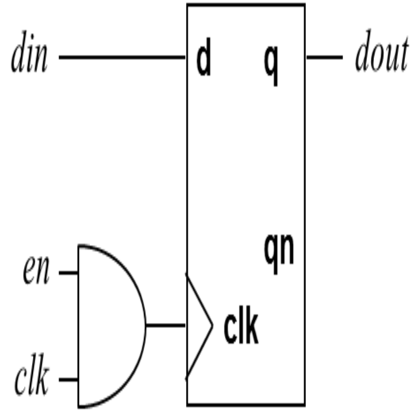
CLOCK GATING – GLITCH PROBLEM

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



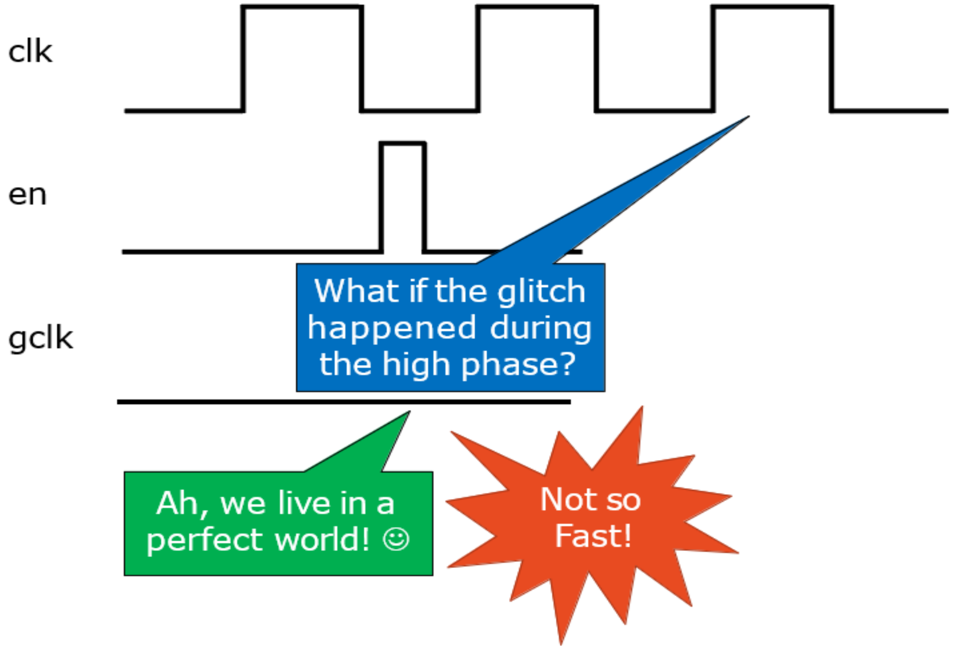
Ah, we live in a perfect world! ☺

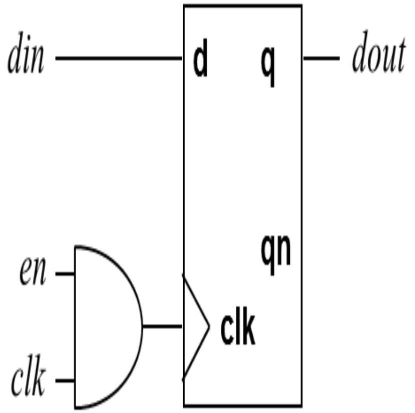
Not so Fast!



CLOCK GATING – GLITCH PROBLEM

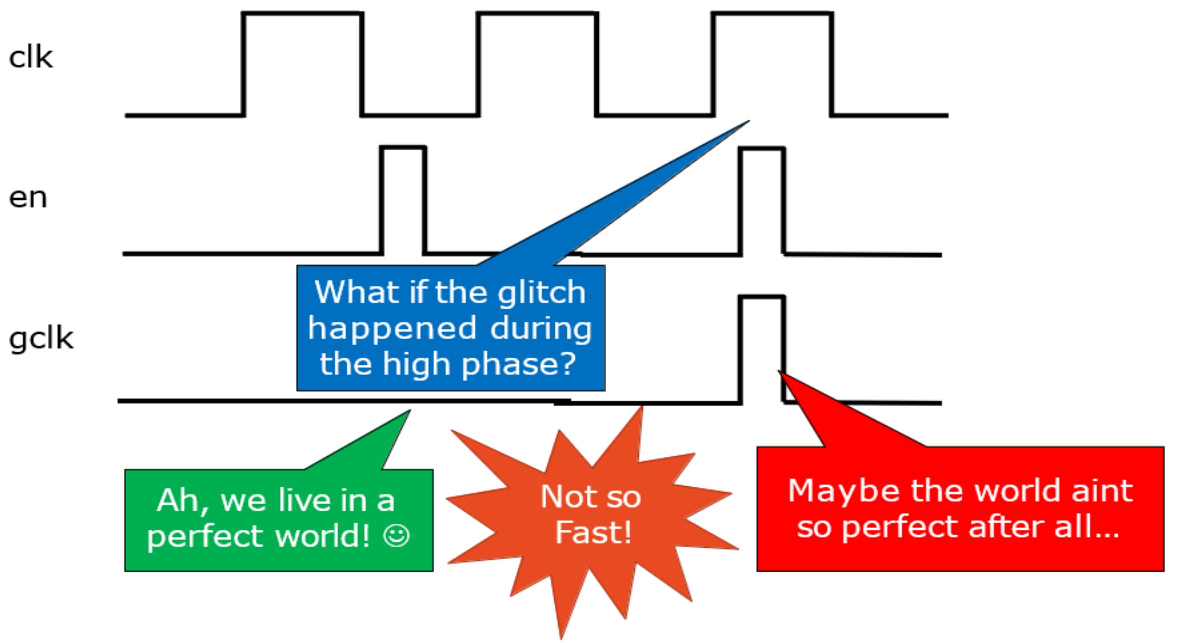
- What happens if there is a glitch on the enable signal?

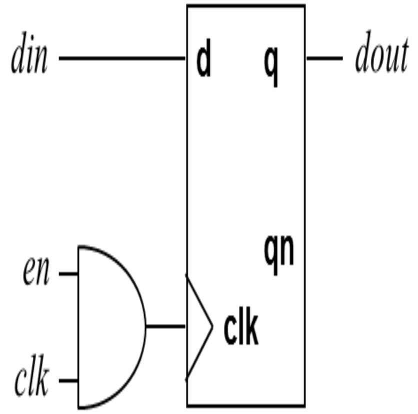

clk
en
gclk
What if the glitch happened during the high phase?
Ah, we live in a perfect world! ☺
Not so Fast!



CLOCK GATING – GLITCH PROBLEM

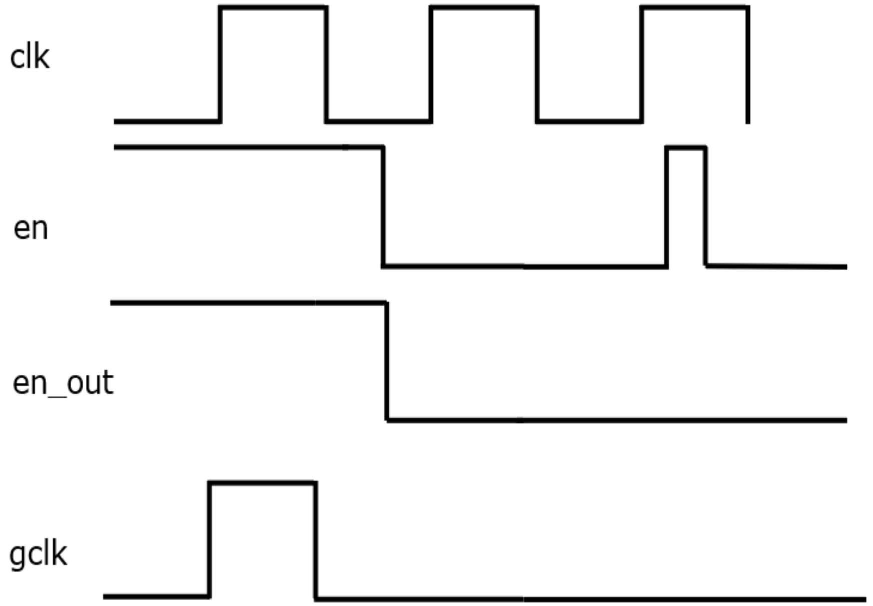
- What happens if there is a glitch on the enable signal?

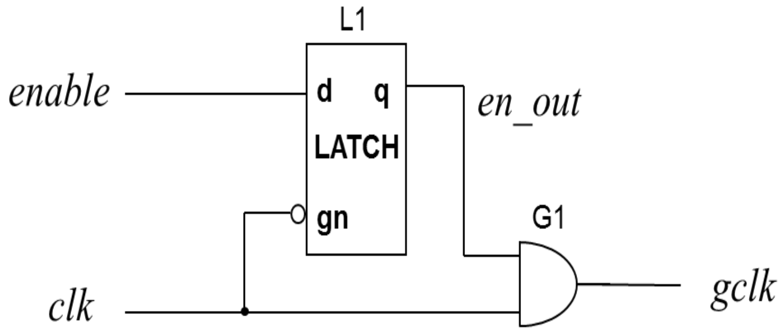

clk
en
gclk
What if the glitch happened during the high phase?
Ah, we live in a perfect world! ☺
Not so Fast!
Maybe the world aint so perfect after all…



SOLUTION: GLITCH-FREE CLOCK GATE

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

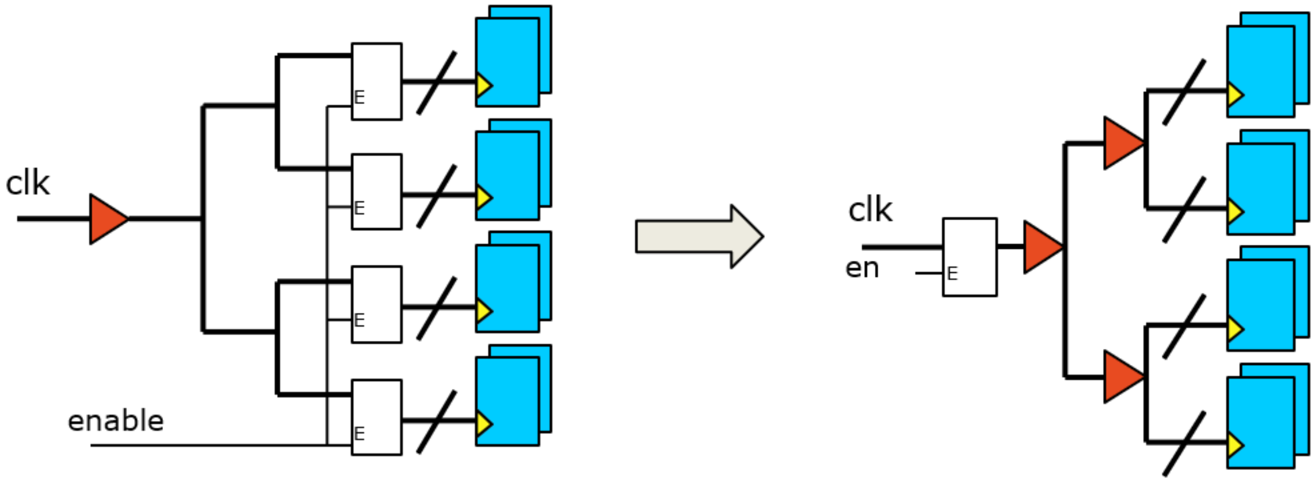




```
//clock gating with glitch prevention latch
always @ (enable or clk)
  begin
     if (!clk)
       en_out <= enable;
  end
assign gclk = en_out && clk;
```

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

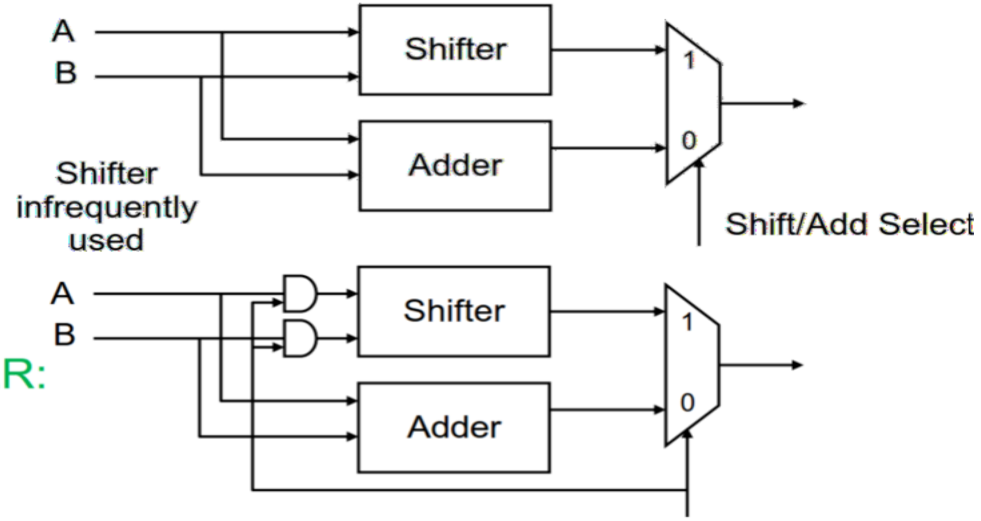
- While clock gating is very well understood and automated, a similar situation occurs due to the toggling of data signals that are not used.
- These situations should be recognized and data gated.

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;
```



CAL POLY
SAN LUIS OBISPO

Thank you!