ECE 551
Digital Design And Synthesis

Fall ‘16

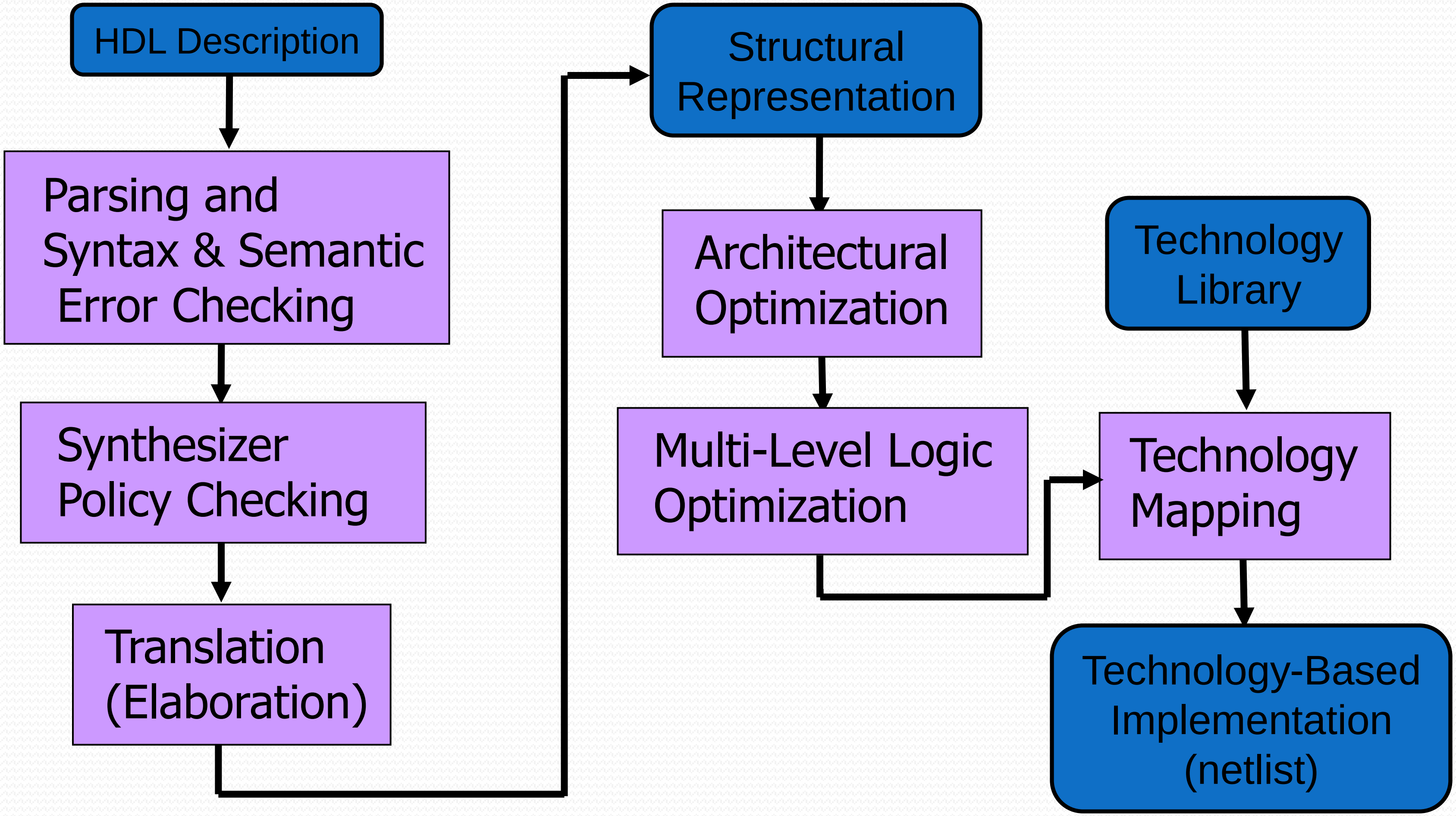
Synthesis Flow
Synthesis Optimizations

Administrative Matters

- HW4 Due Fri Nov 11th

- Project spec out soon. Make sure teams are formed. Stay after class today if not yet part of a team of 3-4 students.

Internal Synthesizer Flow



Initial Steps *(Analyze Verilog File)*

- Parsing for Syntax and Semantics Checking
  - Gives error messages and warnings to user
  - User may modify the HDL description in response

- Synthesizer Policy Checking
  - Check for adherence to allowable language constructs
  - Check for usage recommendations

- This is where you find out you can't use certain Verilog constructs
- This is synthesizer-dependent
  - Example: Design Vision allows indexed part-select (guess[i*2 : 2]), but the Xilinx tool does not
  - Certain things common to MOST synthesizers

Translation (Elaboration)

- Unrolls loops, substitutes macros & parameters, computes constant functions, evaluates generate conditionals
- Builds a structural representation of the design
- Like a netlist, but includes larger components
  - Not just gate-level, may include adders, etc.
- Gives additional errors or warnings to the user
  - Issues in initial transformation to hardware.
- Affects quality achieved by optimization steps
  - Structural representation depends on HDL quality
  - Poor HDL can prevent optimization

Importance of Translation

- It is important for the tool to recognize the sort of logic structures you are trying to describe.
- If it sees a 32-bit full adder, the tool has built-in solutions for optimizing adders
  - Ripple-carry, carry-save, carry look-ahead, etc.
- If it just sees a Boolean function with 65 inputs, it has to work a lot harder to achieve the same results
  - Do you think it can invent a CLA on the fly?

Optimization in Synthesis

- None of these are guaranteed!
  - Most synthesizers will make at least some attempt

- Detect and eliminate redundant logic
- Detect combinational feedback loops
- Exploit don't-care conditions
- Try to detect unused states (logic states you can't get to)
- Synthesize optimal, multilevel realizations subject to:
  - constraints on area and/or speed
  - available technology (library)

Optimization Process

- Optimization modifies the initial netlist resulting from elaboration.
  - Architecture choices made first (CLA,RCA,…)
  - Boolean logic level optimization next
  - Maps to cells from the technology library
  - Attempts to meet all specified constraints
- The process is divided into major phases
  - All or some selection of the major phases may be performed during optimization
  - Phase selection can be controlled by the user

Optimization Phases

- Architectural optimization
  - High-level optimizations that occur before the design is mapped to the logic-level
  - Based on constraints and high-level coding style
  - Level of parallelism?
  - Building block choices like adder architecture (DW components)
  - After optimization circuit function is represented by a generic, technology-independent netlist (GTECH)

Architectural Optimization

- In Synopsys, types include:
  - Sharing common mathematical subexpressions
  - Sharing resources
  - Selecting DesignWare implementations
  - Reordering operators
  - Identifying arithmetic expressions for datapath synthesis

Architectural Optimization

- Examples:
  - Replace an adder used as a counter with incrementor
  - Replace adder and separate subtractor with adder/subtractor if not used simultaneously

    ```
    if (~sub) z = a + b; else z = a - b;
    ```

  - Performs selection of pre-designed components (Synopsys DesignWare)
    - adders, multipliers, shifters, comparators, muxes, etc.

- Need good code for synthesizer to do this
- Designer still knows more about the project

Logic/Gate-Level Optimization

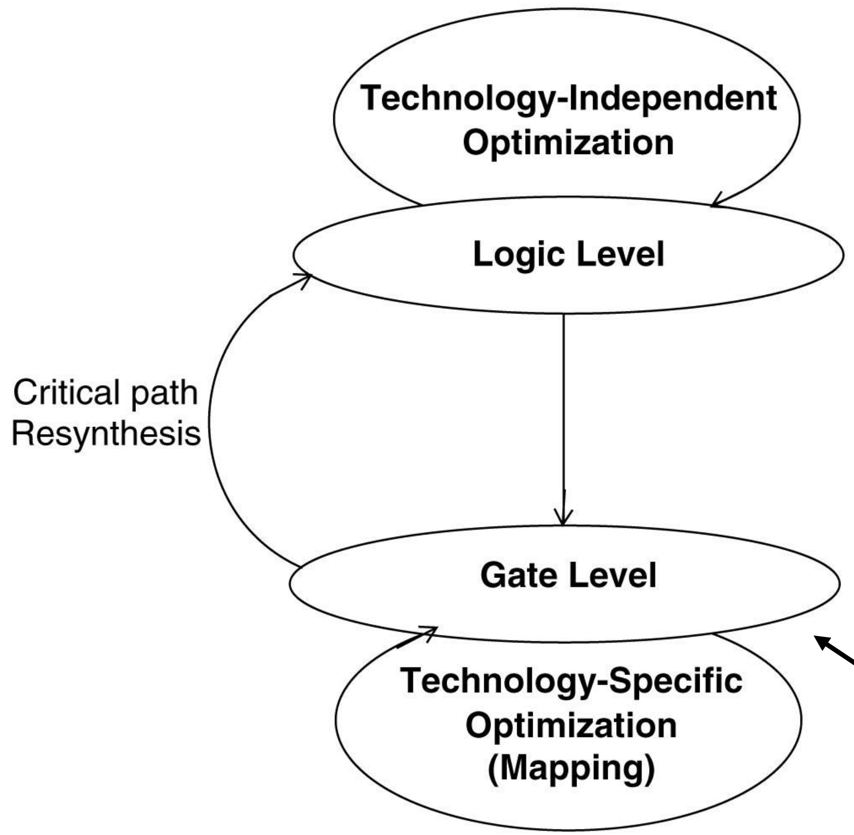
- Works on the generic netlist created by logic synthesis
- Produces a technology-specific netlist.
- In Synopsys, it consists of four stages:
  - Mapping
  - Delay optimization
  - Design rule fixing
  - Area optimization
- This phase often runs in multiple iterations if constraints are not met on the first try

Logic/Gate-Level Optimization

- Mapping
  - Generates a gate level implementation
  - Tries to meet timing and area goals
- Delay optimization
  - Tries to fix delay violations from mapping phase.
  - Does not fix design rule violations or meet area constraints.
- Design rule fixing
  - Tries to correct design rule violations
    - Inserting buffers or resizing existing cells
  - If necessary, violates optimization constraints
- Area optimization
  - Tries to meet area constraints, which have lowest priority

Combinational Optimization

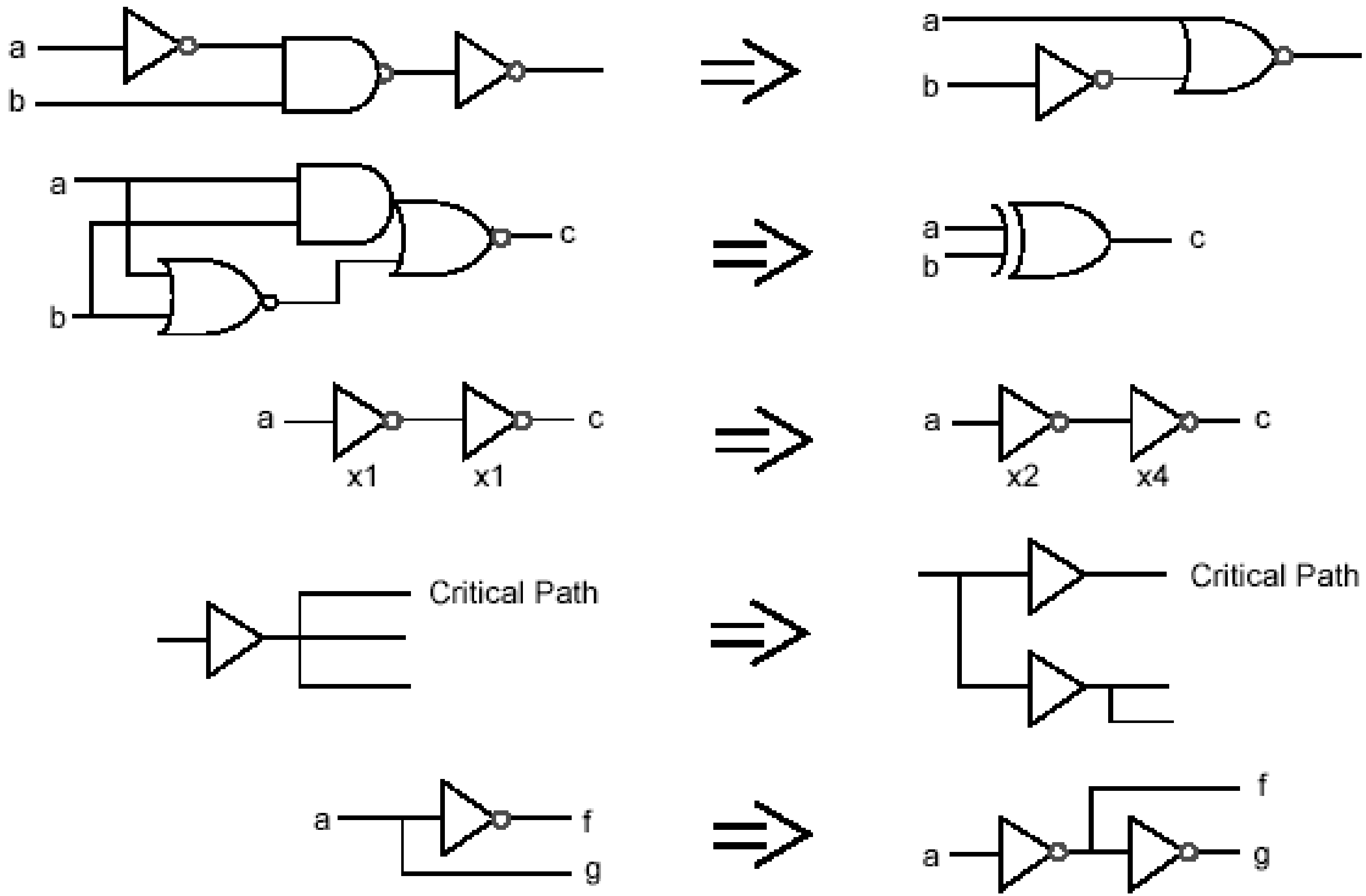
Figure 1-3 Combinational Optimization Phases



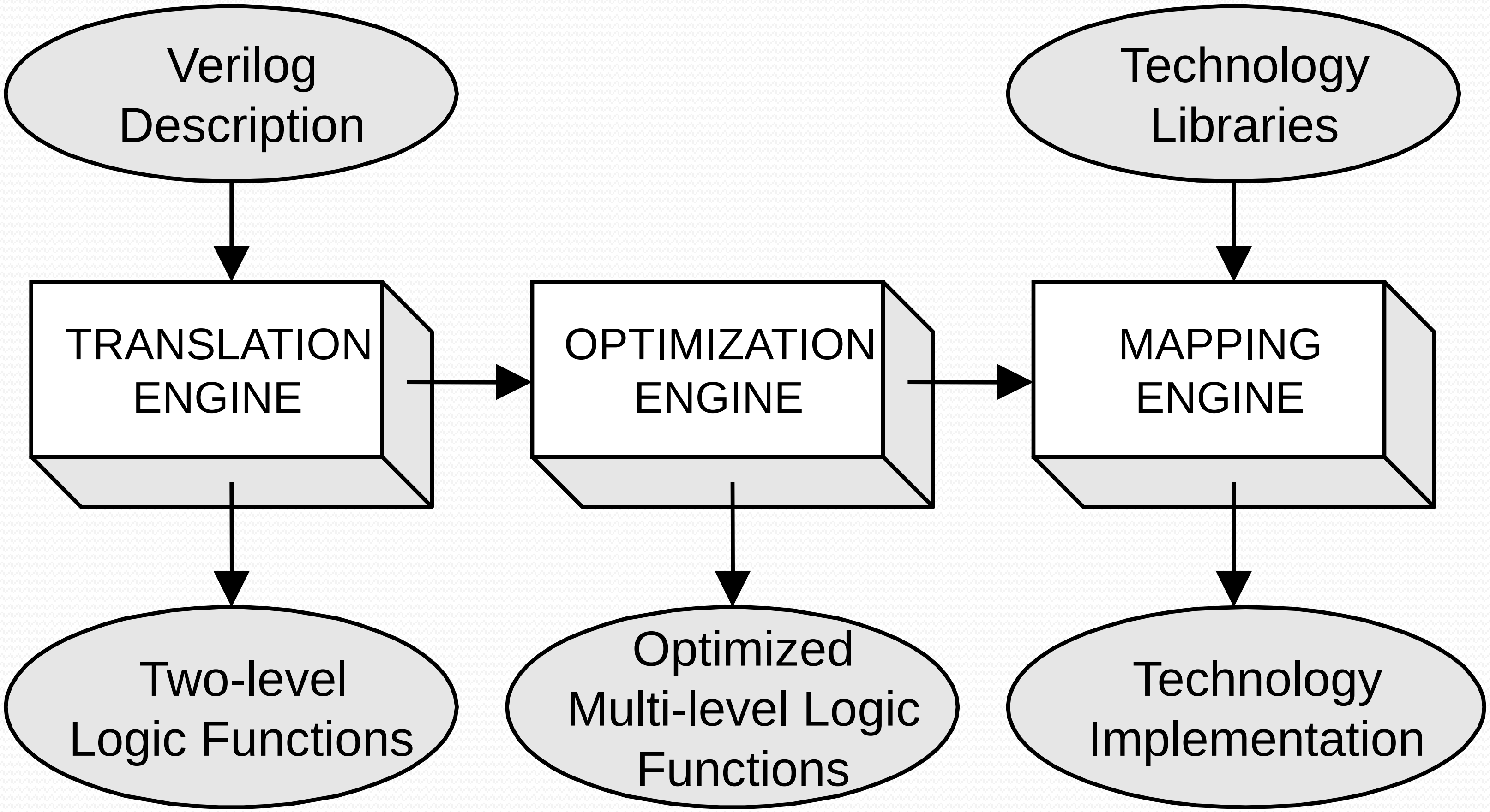
Boolean equation manipulation
- ✓Boolean reduction
- ✓Factoring
- ✓Sharing common terms

Mapping Optimizations
- ✓Gate mapping based on CF
- ✓Applying De-Morgans
- ✓Sizing Gates & Buffering

Gate-Level Optimization



Logic-Level Optimizations



Logic Optimizations

- Area
  - Number of gates fewer == smaller
  - Fan in of gates (# inputs) fewer == smaller
  - Drive Strength (transistor width) narrower == smaller
- Delay
  - Number of logic levels fewer == faster (usually)
  - Fan in of gates (# inputs) fewer == faster
- Gate Effort ➔ Summation of Fan in of gates needed to implement function

$$GateEffort = \sum fanin(gates)$$

- ***Note that examples that follow ignore NOT gates for gate count / levels of circuits***

Logic Optimizations

- Decomposition
- Extraction
- Factoring
- Substitution
- Elimination

- You don't have to remember the names of these
- But understand the concept and the motivation

Decomposition

- Find common expressions
- Reduce redundancy
  - Reduce area (number/size of gates)
- May increase delay
  - More levels of logic

Decomposition Example

- F = abc + abd + a'c'd' + b'c'd'
  ~7 gates, ~3 levels
- F = ab(c + d) + c'd'(a' + b')
- F = ab(c + d) + (c + d)'(ab)'

- X = ab 1 gate, 1 level
- Y = c + d 1 gate, 1 level
- F = XY + X'Y' 3 gates, 3 levels (or what?)
- Gate Effort = 4*(3-input AND) + 4-input OR = 16 effort
- Gate Effort = 2-input AND + 2-input OR + 2*(2-input AND) + 2-input OR = 10 effort

Extraction

- Find common sub-expressions in functions
- Like decomposition, but across more than one function
- Reduce redundancy
  - Reduce area (number/size of gates)
- May increase delay if more logic levels introduced

Extraction Example

- F = (a + b)cd + e 3 gates, 3 levels
- G = (a + b) e’ 2 gates, 2 levels
- H = cde 1 gate, 1 level

- Define common terms: X = a + b, Y = cd 1 gate, 1 level (each)
- F = XY + e 3 gates, 3 levels
- G = Xe’ 2 gate, 2 levels
- H = Ye 2 gate, 2 levels

- Before:
  - (3) 2-input ORs, (2) 3-input ANDs, (1) 2-input AND
  - Gate Effort = 6 + 6 + 2 = 14
- After
  - (2) 2-input ORs, (4) 2-input ANDs
  - Gate Effort = 4 + 8 = 12

Factoring

- Traditional two-level logic is sum-of-products
- Sometimes better expressed by product-of-sums
  - Fewer literals => less area
- May increase delay if logic equation not completely factored (becomes multi-level)

Factoring Example

- Definitely good:
  - F = ac + ad + bc + bd Gate Effort = 8 + 4
  - F = (a + b)(c + d) Gate Effort = 4 + 2
- Maybe good:
  - F = ac + ad + e Gate Effort = 7
  - F = a(c + d) + e Gate Effort = 6

  - Factoring may improve area…
  - But will likely increase delay (tradeoff)

Substitution

- Similar to Extraction (in fact a sub-case of extraction)
- When one function is subfunction of another
- Reduce area
  - Fewer gates
- Can increase delay if more logic levels

Substitution Example

- G = a + b   1 gate, 1 level
- F = a + b + c   1 gate, 1 level

- F = G + c   2 gate, 2 levels

- **Before:**
  - (1) 2-input OR, (1) 3-input OR => Gate Effort = 5
- **After**
  - (2) 2-input ORs (but increased levels) => Gate Effort = 4

Elimination (Flattening)

- Opposite of previous optimizations
- Goal is to reduce delay
  - Make signals travel though as few logic levels as possible
- But will likely increase area
  - Gate replication / redundant logic

Elimination Example

- G = c + d — 1 gate, 1 level
- F = Ga + G' b — 3 gates, 3 levels

- G = c + d — 1 gate, 1 level
- F = ac + ad + bc’d’ — 4 gates, 2 levels

- **Before:**
  - (2) 2-input ORs, (2) 2-input ANDs
- **After:**
  - (1) 2-input OR, (1) 3-input OR, (2) 2-input ANDs, (1) 3-input AND (but fewer levels)

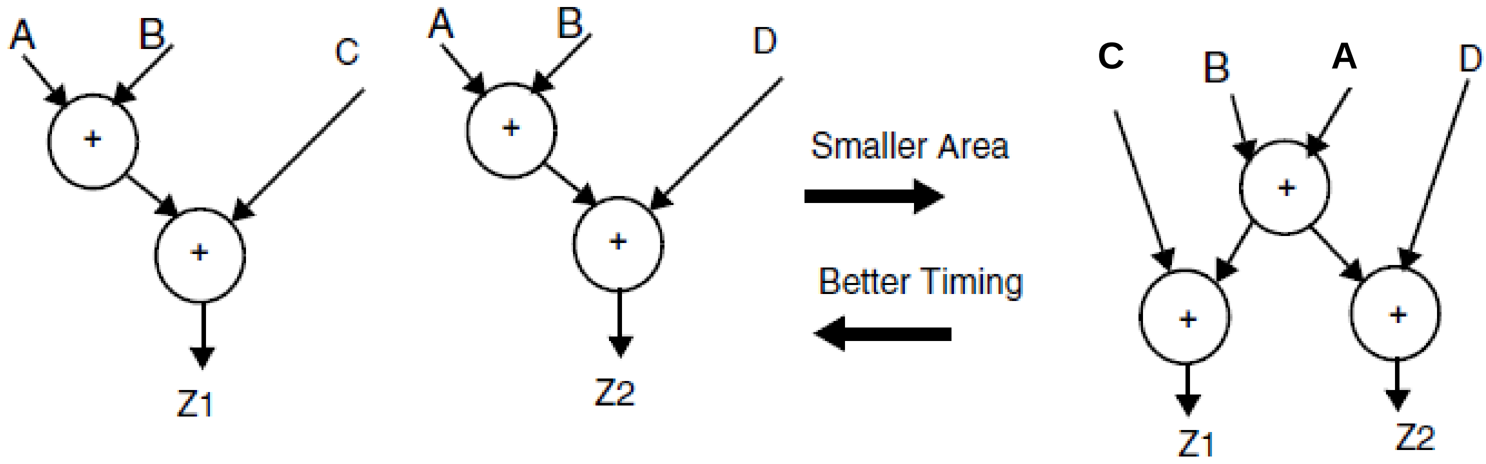
compile_ultra Optimizations

- High effort, maximum optimization
- Automatic hierarchical ungrouping
  - Ungroups small modules before mapping
  - Ungroups critical path based on delay
- Automatic datapath extraction
  - E.g. carry-save adders
- Boundary optimization
  - Propagates logic across hierarchical boundaries (constants, NC inputs/outputs, NOT)
- Sequential inversion
  - Sequential elements can have their outputs inverted

How to Ensure a Job Offer Once You Have the on-site Interview

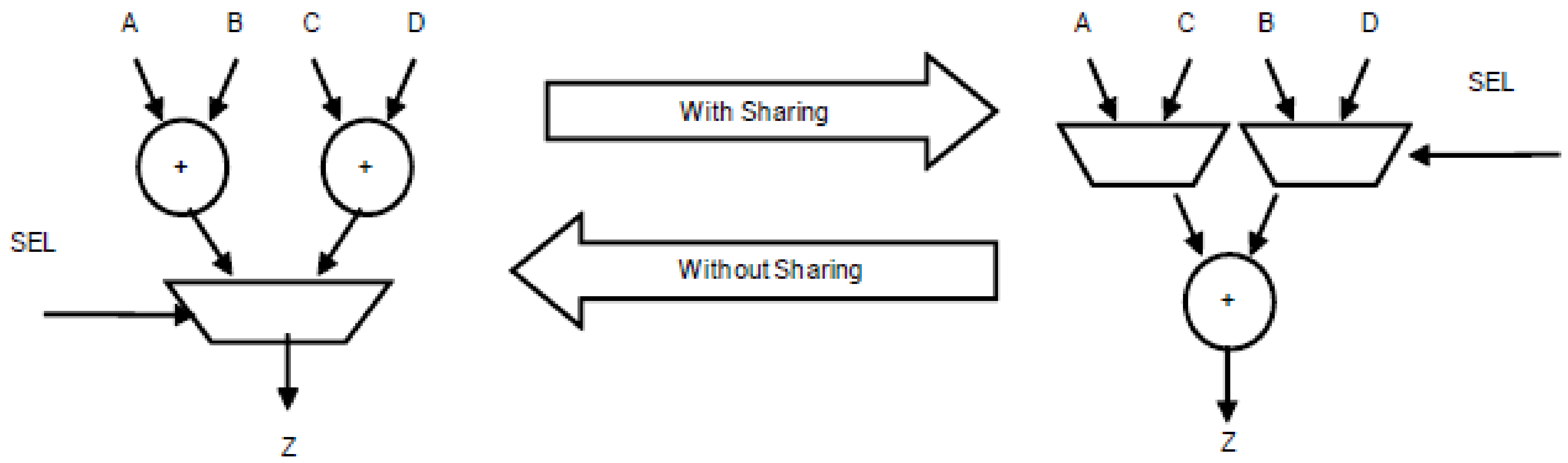
Sharing and Unsharing

- Expression sharing may be overridden later due to timing
  - Z1 <= A + B + C
  - Z2 <= A + B + D
  - Arrival time is A < B < D < C



Sharing and Unsharing

- Mutually exclusive operations can share resources
  - if(SEL) Z = A + B
  - else Z = C + D



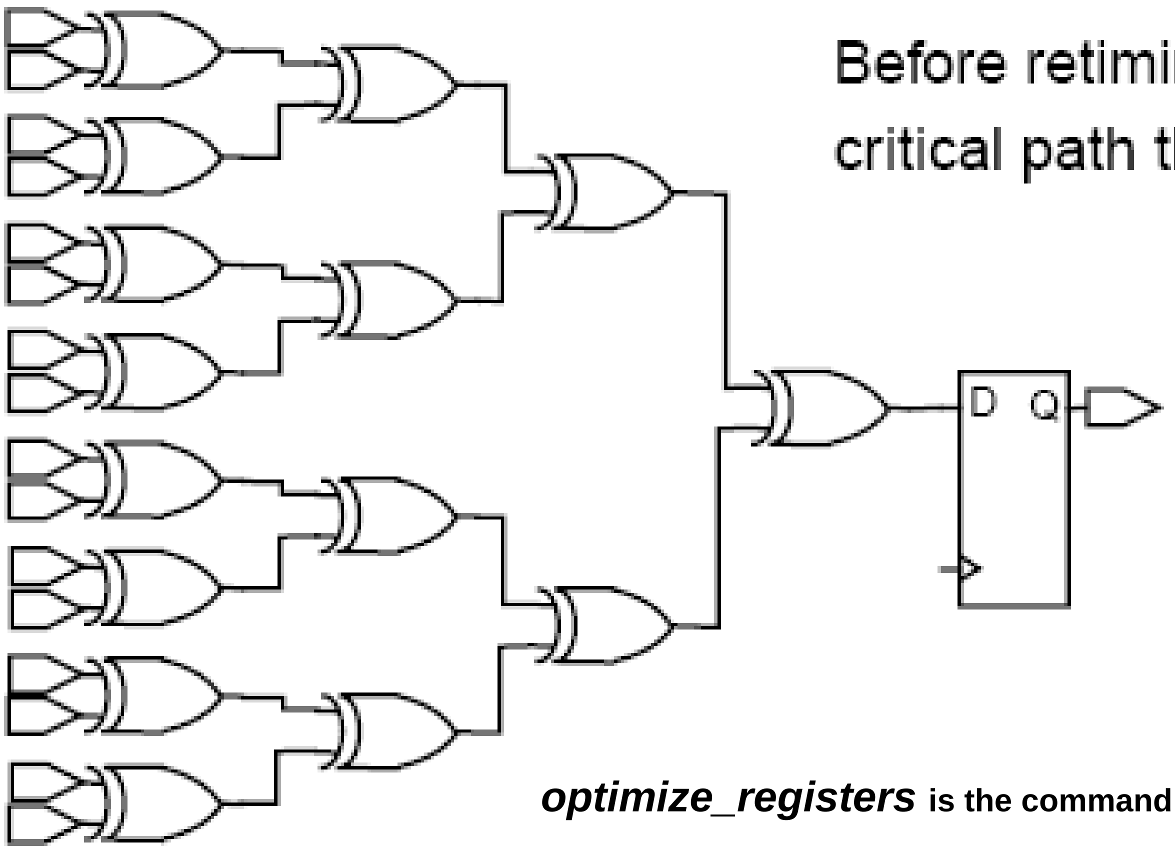
Sequential Inversion

- *set compile_seqmap_enable_output_inversion true*
- Allows the mapping of sequential elements in the design to library cells whose outputs are inverted.
- This can reduce area and or speed
- Not applicable for compile_ultra, because compile_ultra already does this.

Register Retiming

- At the HDL level, determining the optimal placement of registers is difficult and tedious at best, or just plain impossible at worst
- The register retiming tool moves registers through the synthesized combinational logic network to improve timing and/or area
  - Equalize delay (i.e. reduce critical path delay by increasing delay in other paths)
  - Reduce the number of flip-flops if timing criteria are met
    - Usually propagate registers forward
- Can also automatically pipeline combinational logic modules

Register Retiming Example [1]



Before retiming:

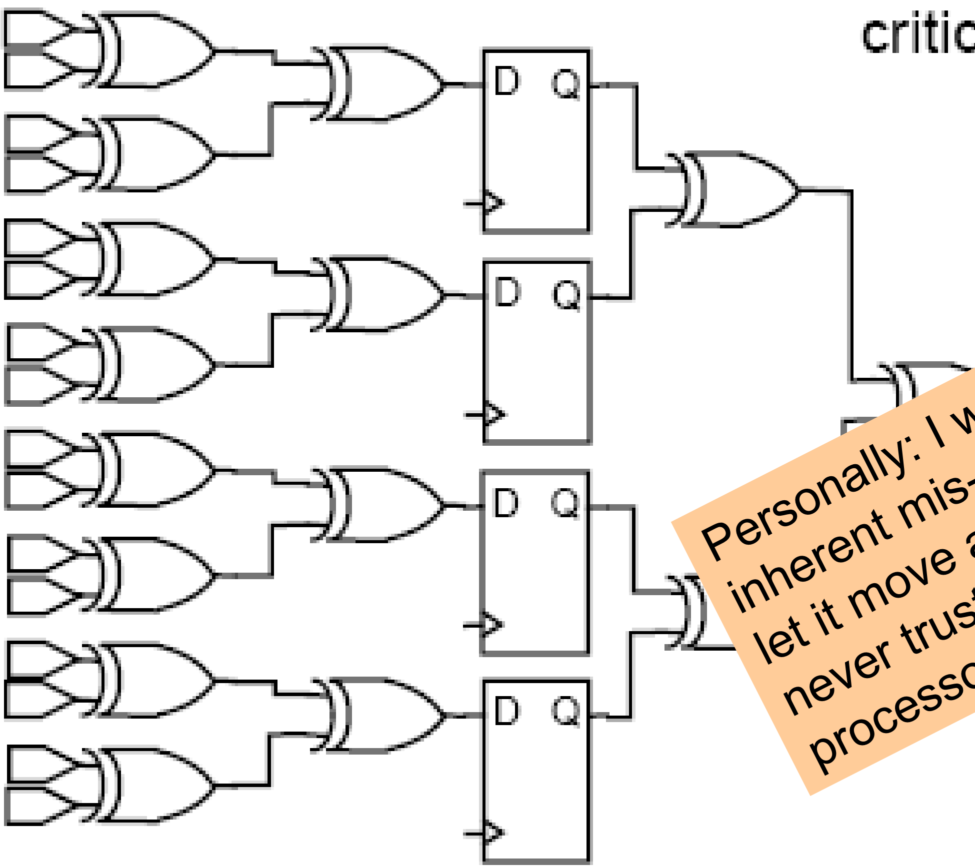
critical path through four gates

optimize_registers **is the command in Design Compiler**

Register Retiming Example [2]

After retiming:

critical path through two gates



Personally: I would **never use this**. I have an inherent mis-trust of CAD tools. I would never let it move any of my flops, and certainly never trust it to rearrange my pipelined processor.