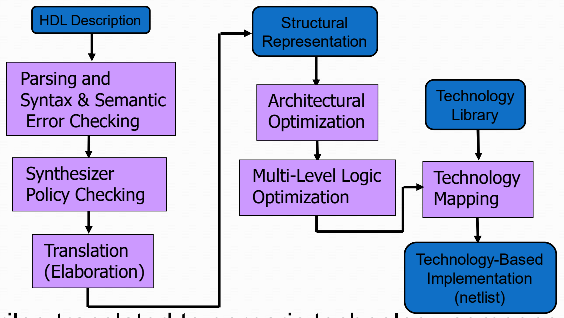
ECE 551
Digital Design And Synthesis

Fall '21

Synthesis Flow
Synthesis Optimizations

1

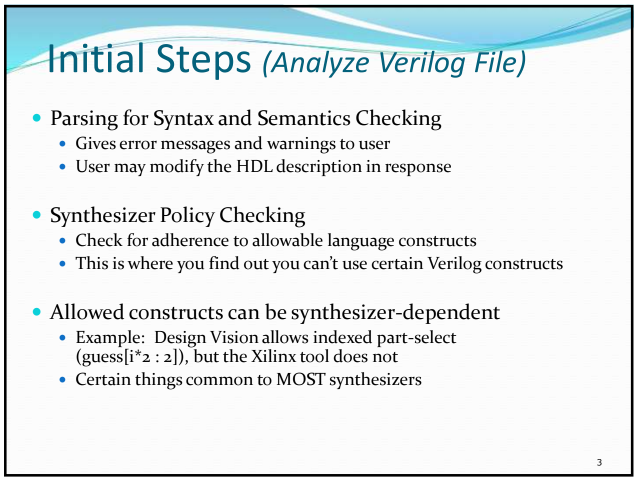
Internal Synthesizer Flow


HDL Description
Parsing and Syntax & Semantic Error Checking
Synthesizer Policy Checking
Translation (Elaboration)
Structural Representation
Architectural Optimization
Multi-Level Logic Optimization
Technology Library
Technology Mapping
Technology-Based Implementation (netlist)

verilog translated to gernaric technology component e.g. adder, mux, flop

2

2

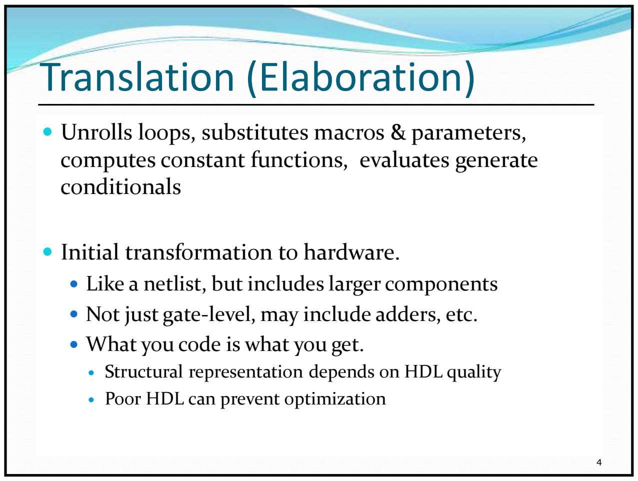


Initial Steps *(Analyze Verilog File)*

- 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
  - This is where you find out you can't use certain Verilog constructs
- Allowed constructs can be 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

3

3

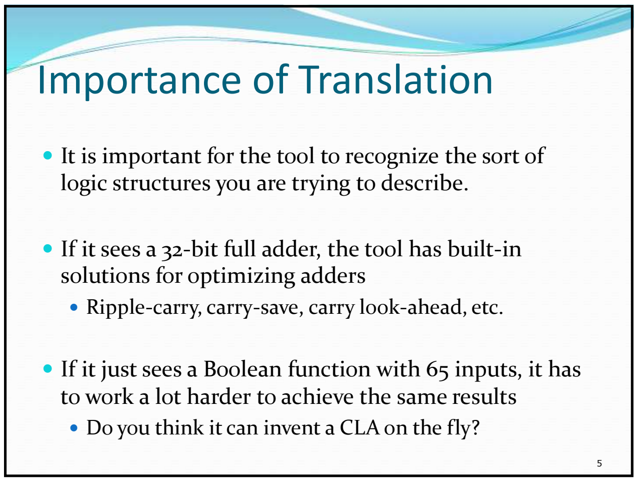


Translation (Elaboration)

- Unrolls loops, substitutes macros & parameters, computes constant functions, evaluates generate conditionals
- Initial transformation to hardware.
  - Like a netlist, but includes larger components
  - Not just gate-level, may include adders, etc.
  - What you code is what you get.
    - Structural representation depends on HDL quality
    - Poor HDL can prevent optimization

4

4

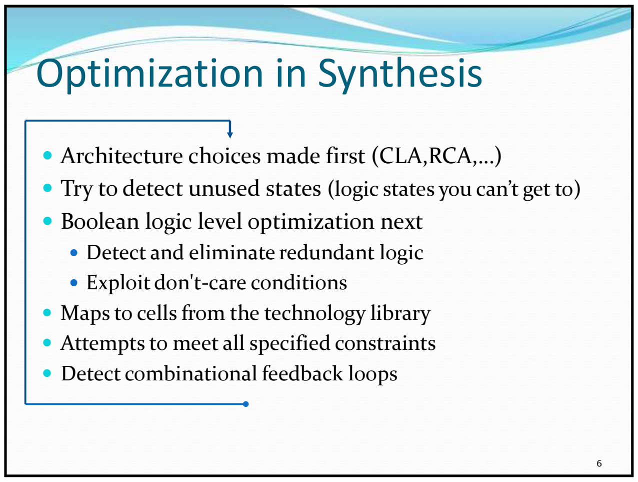


Importance of Translation

- 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?

5

5

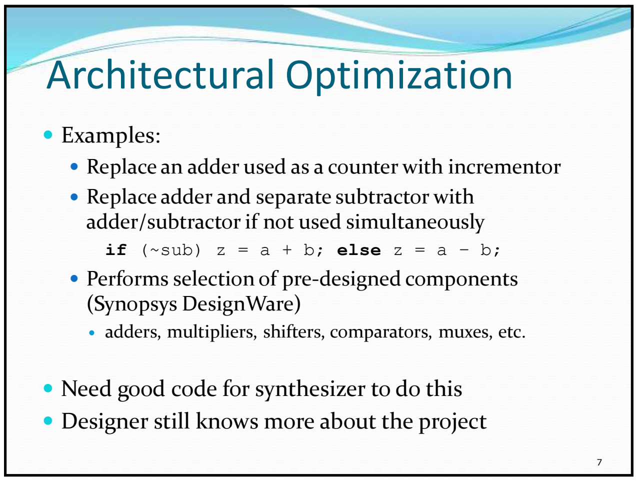


Optimization in Synthesis

- Architecture choices made first (CLA,RCA,…)
- Try to detect unused states (logic states you can't get to)
- Boolean logic level optimization next
  - Detect and eliminate redundant logic
  - Exploit don't-care conditions
- Maps to cells from the technology library
- Attempts to meet all specified constraints
- Detect combinational feedback loops

6

6



Architectural Optimization

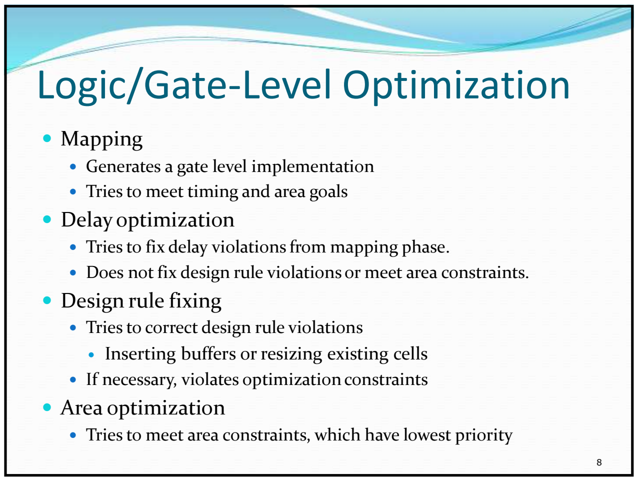
- Examples:
  - Replace an adder used as a counter with incrementor
  - Replace adder and separate subtractor with adder/subtractor if not used simultaneously

```
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

7

7

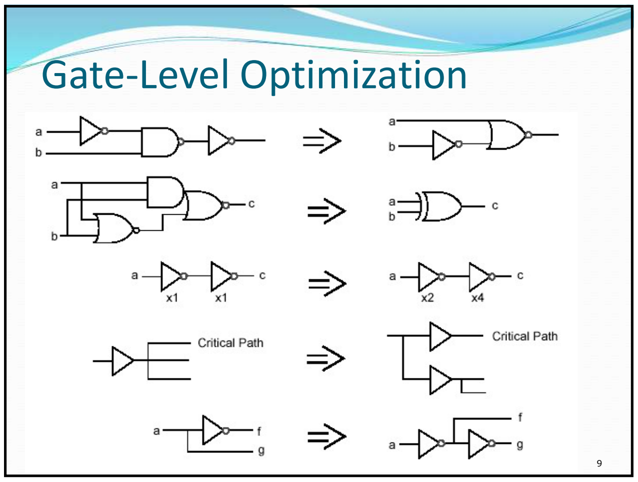


Logic/Gate-Level Optimization

- 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

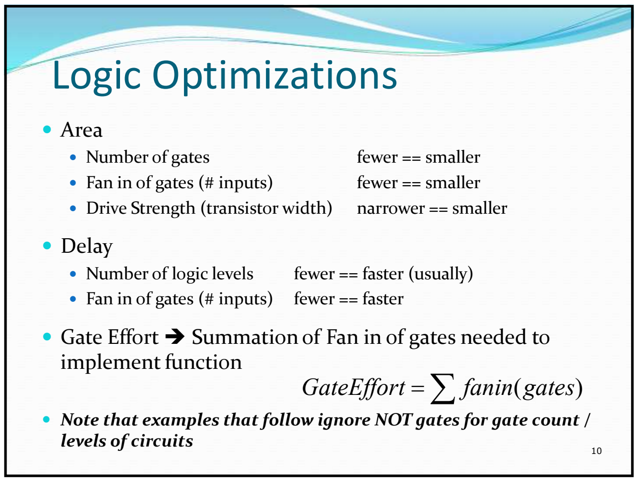
8

8



Gate-Level Optimization

9



Logic Optimizations

- Area

Number of gates	fewer == smaller
Fan in of gates (# inputs)	fewer == smaller
Drive Strength (transistor width)	narrower == smaller

- Delay

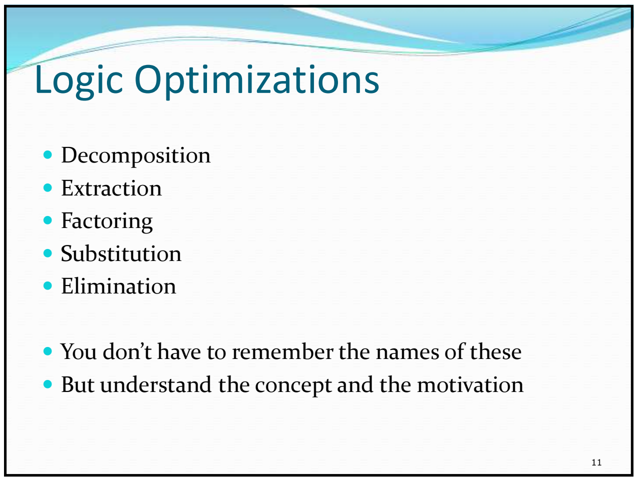
Number of logic levels	fewer == faster (usually)
Fan in of gates (# inputs)	fewer == faster

- 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***

10

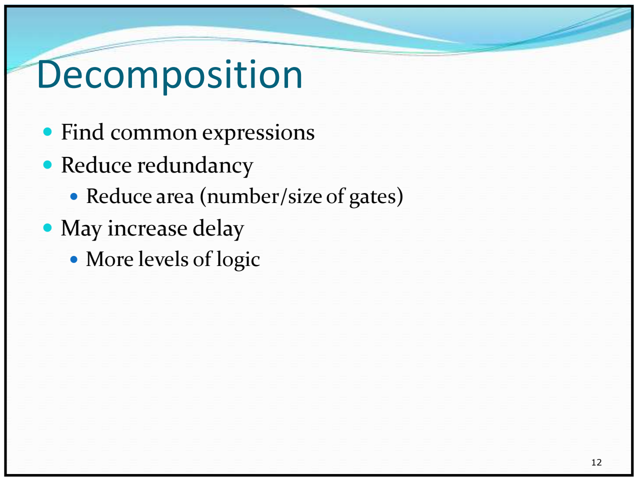


Logic Optimizations

- Decomposition
- Extraction
- Factoring
- Substitution
- Elimination

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

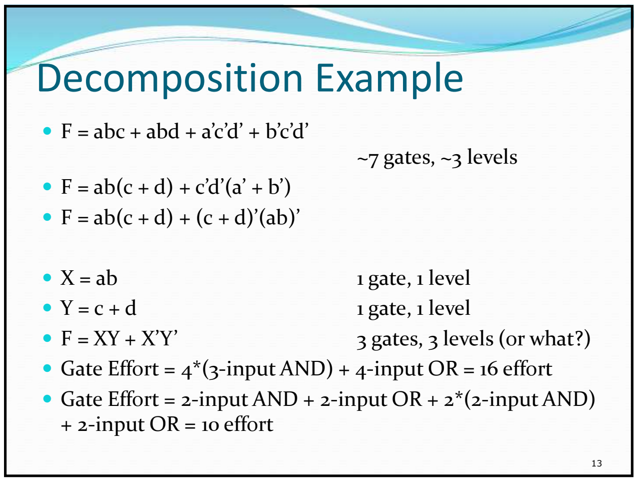
11



Decomposition

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

12



Decomposition Example

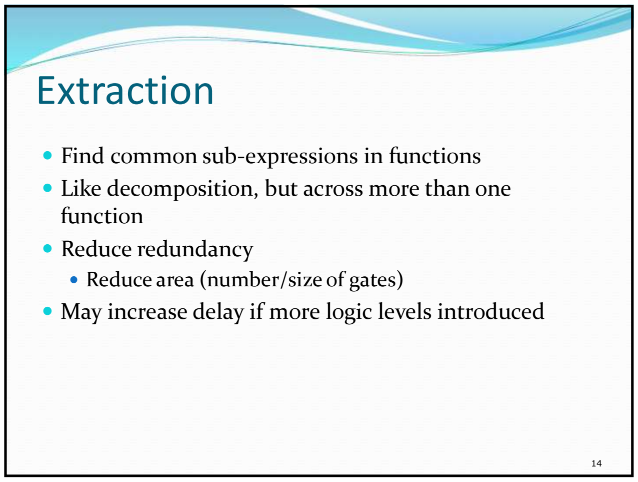
- F = abc + abd + a'c'd' + b'c'd'
  ~7 gates, ~3 levels
- F = ab(c + d) + c'd'(a' + b')
- F = ab(c + d) + (c + d)'(ab)'

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

13

13

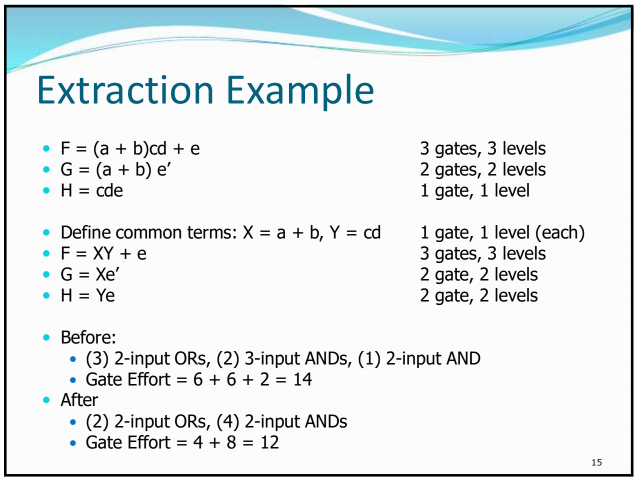


Extraction

- 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

14

14



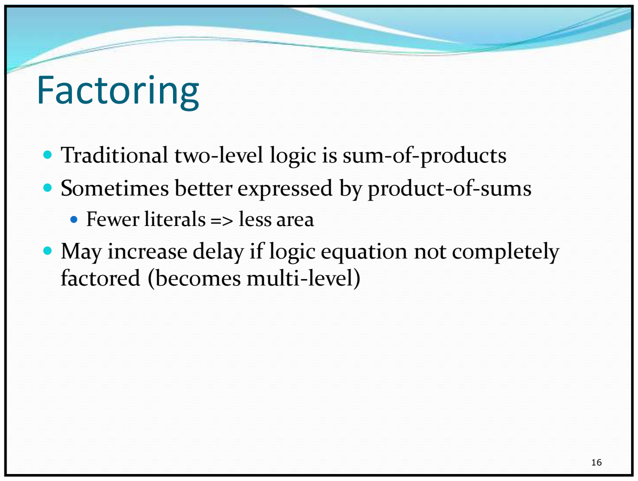
Extraction Example

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

15

15

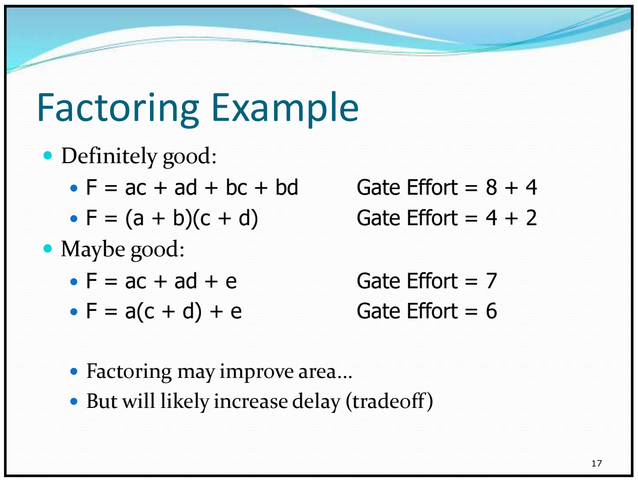


Factoring

- 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)

16

16



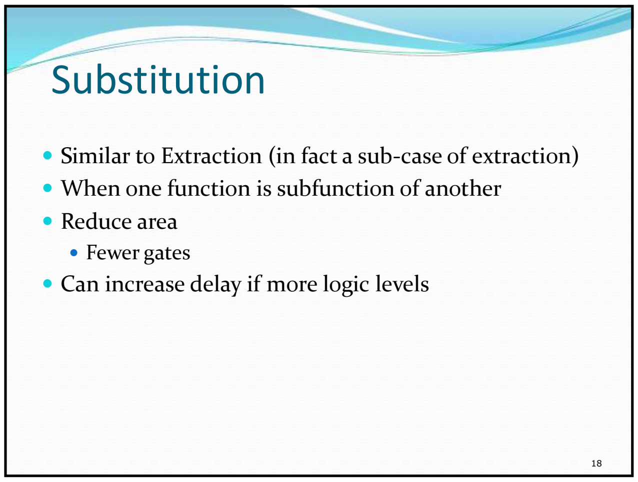
Factoring Example

- 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)

17

17

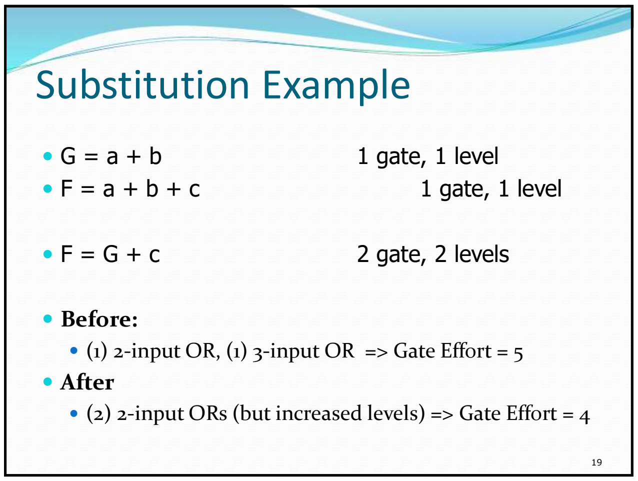


Substitution

- 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

18

18



Substitution Example

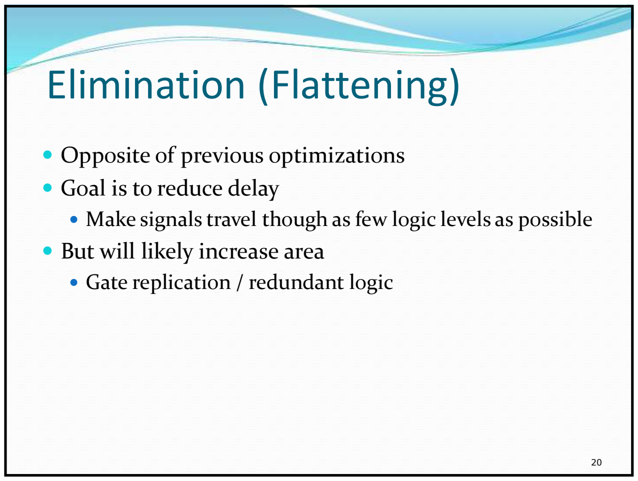
- G = a + b 1 gate, 1 level
- F = a + b + c 1 gate, 1 level

- 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

19

19



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

20

20

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

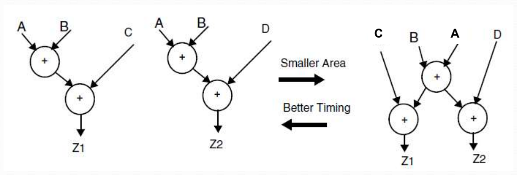
- **Before:**
  - (2) 2-input ORs, (2) 2-input ANDs
- **After:**
  - (1) 2-input OR, (1) 3-input OR, (2) 2-input ANDs, (1) 3-input AND (but fewer levels)

21

21

Sharing and Unsharing

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

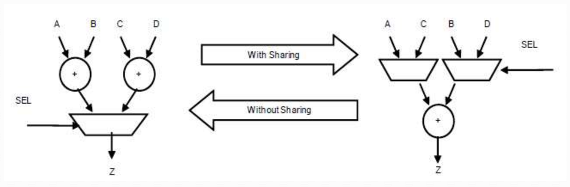


22

22

Sharing and Unsharing

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



23

23