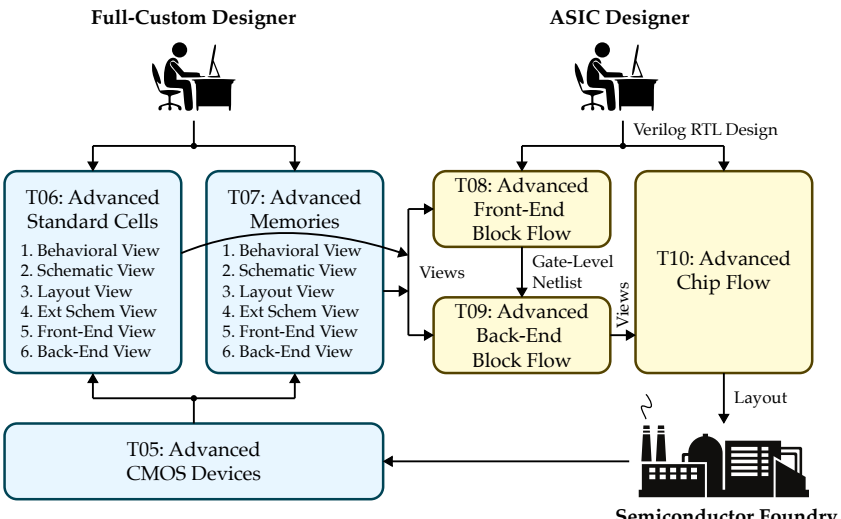
ECE 6745 Complex Digital ASIC Design

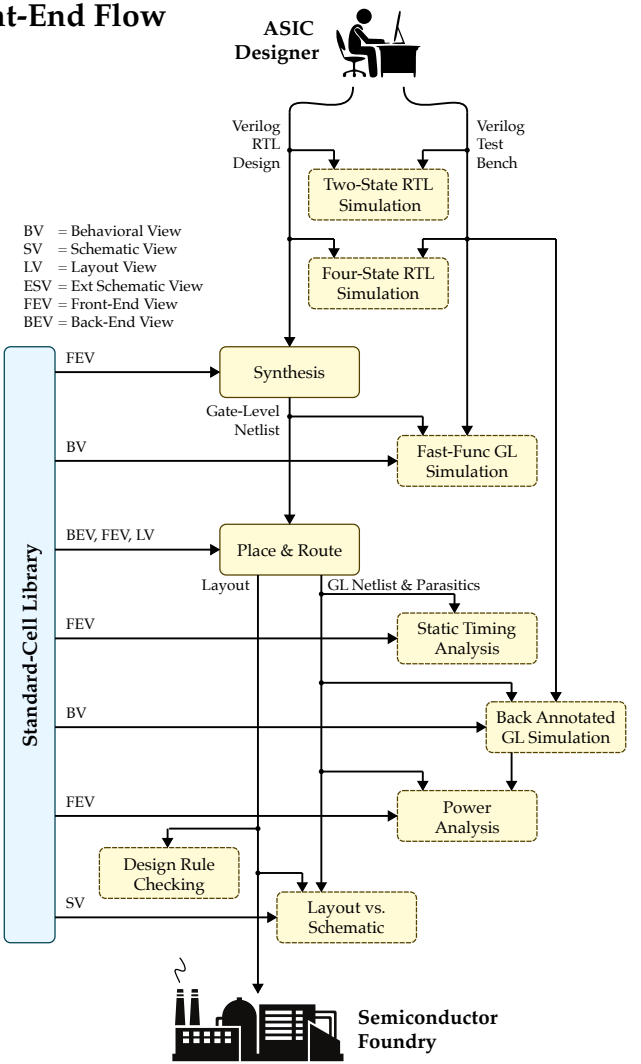
Topic 8: Advanced Front-End Flow

School of Electrical and Computer Engineering
Cornell University

revision: 2026-03-17-12-23

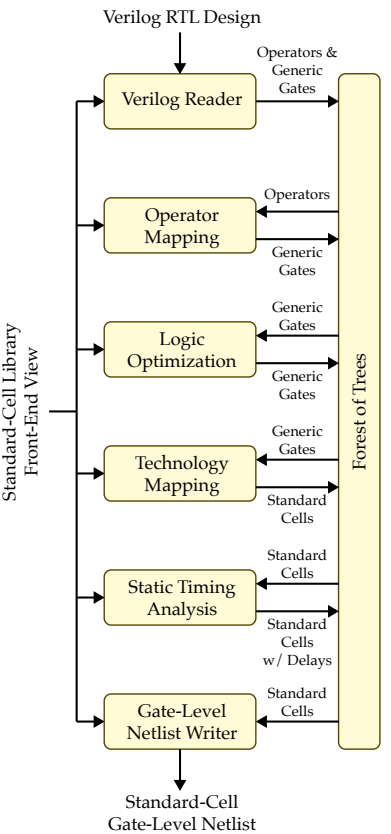


1. Front-End Flow



Synthesis

- Synthesis takes as input a Verilog RTL design and produces a standard-cell gate-level netlist
- Synthesis Data Structure
  - Forest of trees of operator, generic-gate, and standard-cell nodes
- Synthesis Algorithms
  - Verilog Reader: Parses Verilog RTL design into forest of trees of registers, operators, and generic gates
  - Operator Mapping: Maps multi-bit operators to trees of generic gates
  - Logic Optimization: Optimizes forest of trees to minimize area while meeting timing constraints (technology independent metrics)
  - Technology Mapping: Maps trees of generic gates to trees of standard cells trying to minimize area while meeting timing constraints (technology-dependent metrics)
  - Static Timing Analysis: Statically analyzes all paths to ensure design meets both setup and hold timing constraints
  - Gate-Level Netlist Writer: Outputs a standard-cell gate-level netlist



1.1. *Algorithm:* Verilog Reader

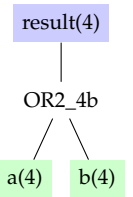
- Lexing: Breaking Verilog source code into meaningful tokens
- Parsing: Organizing tokens into an abstract syntax tree (AST)
- Foresting: Converting AST into a forest of trees

- Verilog grammar for multi-bit wire declarations and assign statements with multi-bit operators

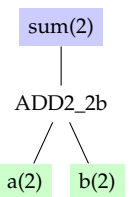
```
module: "module" MNAME port_decl_list? ";" stmt* "endmodule"

port_decl_list: "(" port_decl ( "," port_decl )* ")"
port_decl: DIR "wire" range? SIGNAL

?stmt: wire_decl | assign

wire_decl: "wire" range? SIGNAL ";"
assign:    "assign" SIGNAL "=" expr ";"

range: "[" INT ":" INT "]"

?expr: SIGNAL | LITERAL | "(" expr ")"
     | "~" expr        -> not
     | expr "+" expr   -> add
     | expr "-" expr   -> sub
     | expr "<" expr   -> lt
     | expr ">" expr   -> gt
     | expr "==" expr  -> eq
     | expr "&" expr   -> and
     | expr "|" expr   -> or
     | expr "^" expr   -> xor
     | expr "?" expr ":" expr -> ternary

SIGNAL:  /[a-zA-Z_][a-zA-Z0-9_]*/
LITERAL: /\d+'[bB][01]+/ | /\d+/
INT:     /\d+/
DIR:     ( "input" | "output" )
MNAME:   /[a-zA-Z_][a-zA-Z0-9_]*/
```

- Realistic SystemVerilog grammar would have thousands of rules
- Read `always_comb` block into trees
- Read `always_ff` block into register at root of tree

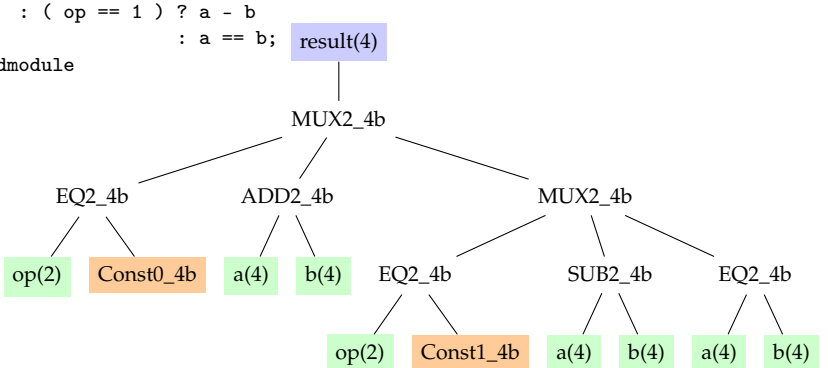
```
module SimpleModule
(
  input  wire [3:0] a,
  input  wire [3:0] b,
  output wire [3:0] result
);
  assign result = a | b;
endmodule
```



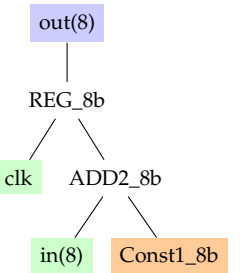

```
module Adder_2b
(
  input  wire [1:0] a,
  input  wire [1:0] b,
  output wire [1:0] sum
);
  assign sum = a + b;
endmodule
```



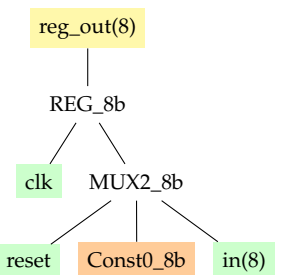

```
module ALU
(
  input  wire [3:0] a,
  input  wire [3:0] b,
  input  wire [1:0] op,
  output wire [3:0] result
);
  assign result
    = ( op == 0 ) ? a + b
    : ( op == 1 ) ? a - b
                  : a == b;
endmodule
```

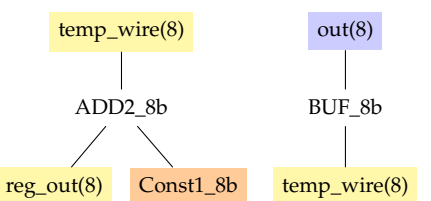


```
module RegIncrV1
(
  input  wire       clk,
  input  wire [7:0] in,
  output wire [7:0] out
);

  always_ff @( posedge clk ) begin
    out <= in + 1;
  end

endmodule
```



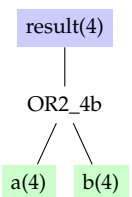

```
module RegIncrV2
(
  input  logic       clk,
  input  logic       reset,
  input  logic [7:0] in,
  output logic [7:0] out
);

  logic [7:0] reg_out;

  always_ff @( posedge clk ) begin
    if ( reset )
      reg_out <= 0;
    else
      reg_out <= in;
  end

  logic [7:0] temp_wire;

  always_comb begin
    temp_wire = reg_out + 1;
  end

  assign out = temp_wire;

endmodule
```

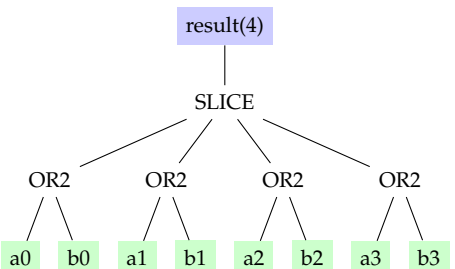


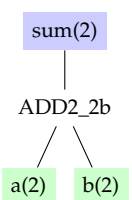


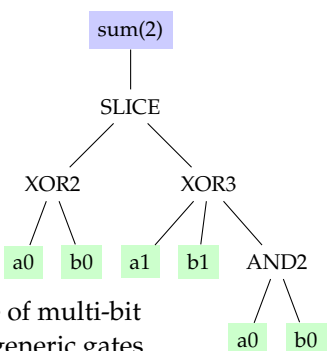
1.2. *Algorithm:* Operator Mapping

- Use tree rewrite rules to map multi-bit operators to other multi-bit operators and/or generic gates; "bit blast" multi-bit signals
- Continue to apply rules to all trees until reach fixed point where all trees only contain generic gates and single-bit signals

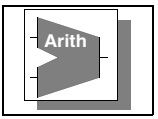


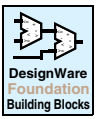






- Can rewrite a multiplier into tree of multi-bit adders which are rewritten into generic gates
- Rewrite rules can be parameterized by the operator and/or bitwidth
- Multiple rewrite rules for single operator possible in which case this becomes logic optimization





DW01_add

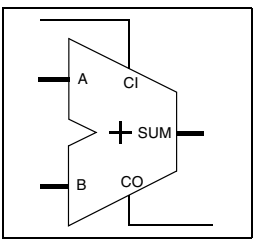
Adder

Version, STAR, and myDesignWare Subscriptions: IP Directory

Features and Benefits

Revision History

- Parameterized word length
- Carry-in and carry-out signals



Description

DW01_add adds two operands `A` and `B` with a carry-in `CI` to produce the output `SUM` with a carry-out `CO`.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
A	*width* bits	Input	Input data
B	*width* bits	Input	Input data
CI	1 bit	Input	Carry-in
SUM	*width* bits	Output	Sum of (`A` + `B` + `CI`)
CO	1 bit	Output	Carry-out

Table 1-2 Parameter Description

Parameter	Values	Description
width	≥1	Word length of `A`, `B`, and `SUM`

Table 1-3 Synthesis Implementations[a]

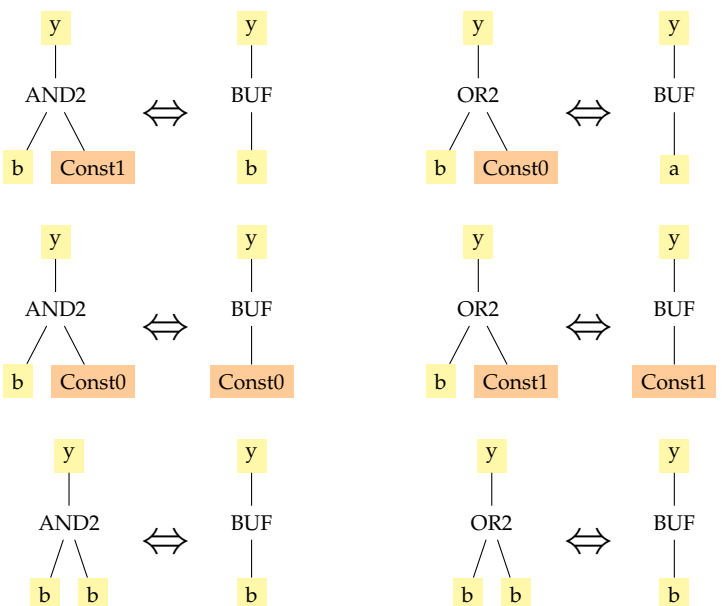
Implementation	Function	License Feature Required
rpl	Ripple-carry synthesis model	none
cla	Carry-look-ahead synthesis model	none
pparch	Delay-optimized flexible parallel-prefix	DesignWare
apparch	Area-optimized flexible architecture that can be optimized for area, for speed, or for area, speed	DesignWare

a. During synthesis, Synopsys synthesis tools select the appropriate architecture for your constraints. Alternatively, you may use the set_implementation command to choose one specific implementation from those listed in this table. For details, see the documentation for your Synopsys synthesis tool.

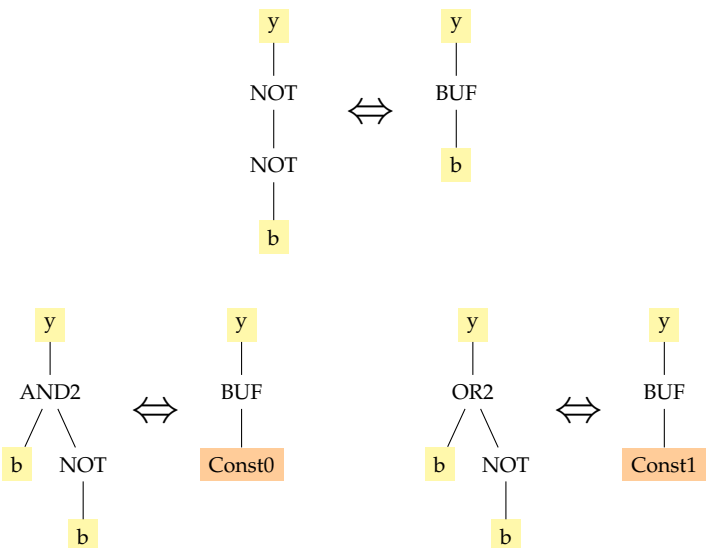
1.3. *Algorithm:* Logic Optimization

- Local transformations use rewrite rules based on Boolean algebra
  - Do not create new wires
  - Do not restructure large parts of the tree
  - Do not look at more than one tree at a time

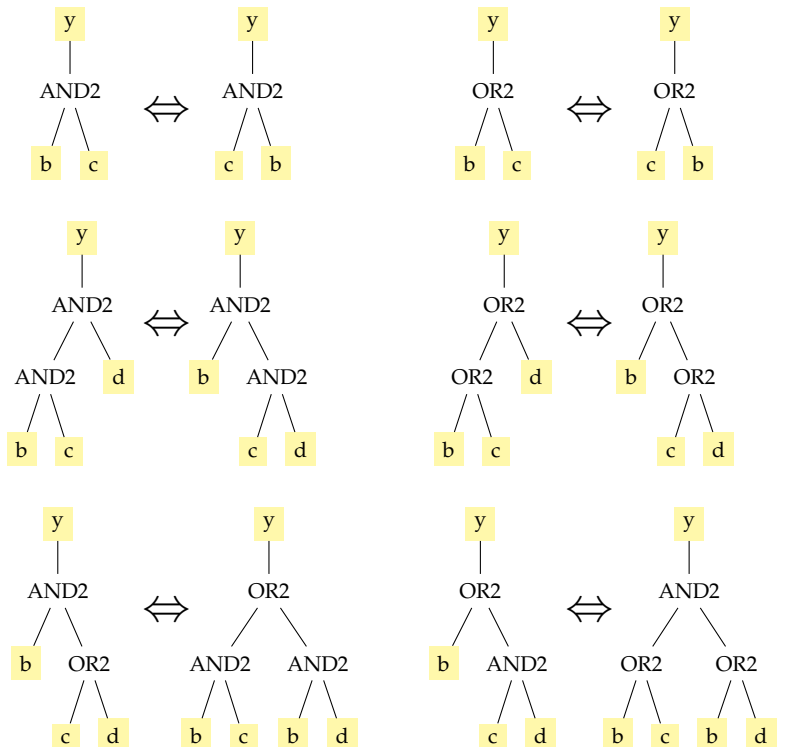
Number	Theorem	Dual	Name
T1	B • 1 = B	B + 0 = B	Identity
T2	B • 0 = 0	B + 1 = 1	Null Element
T3	B • B = B	B + B = B	Idempotency



Number	Theorem	Dual	Name
T1	$B \bullet 1 = B$	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	$B + B = B$	Idempotency
T4	$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	$B + \overline{B} = 1$	Complements

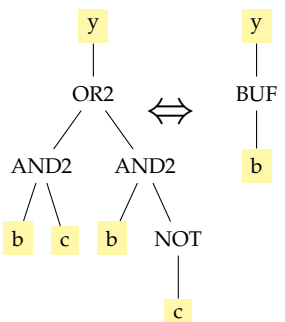


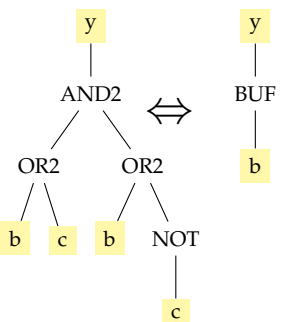
#	Theorem	Dual	Name
T6	B•C = C•B	B+C = C+B	Commutativity
T7	(B•C) • D = B • (C•D)	(B + C) + D = B + (C + D)	Associativity
T8	B • (C + D) = (B•C) + (B•D)	B + (C•D) = (B+C) (B+D)	Distributivity

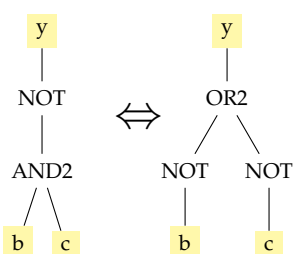


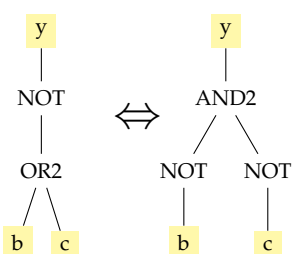
Kernel-based factoring systematically finds common divisors in Boolean expressions by identifying kernels and then using them to factor logic into shared subexpressions

#	Theorem	Dual	Name
T6	B•C = C•B	B+C = C+B	Commutativity
T7	(B•C) • D = B • (C•D)	(B + C) + D = B + (C + D)	Associativity
T8	B • (C + D) = (B•C) + (B•D)	B + (C•D) = (B+C) (B+D)	Distributivity
T9	B • (B+C) = B	B + (B•C) = B	Covering
T10	(B•C) + (B•$\overline{C}$) = B	(B+C) • (B+$\overline{C}$) = B	Combining
T11	(B•C) + ($\overline{B}$•D) + (C•D) = (B•C) + ($\overline{B}$•D)	(B+C) • ($\overline{B}$+D) • (C+D) = (B+C) • ($\overline{B}$+D)	Consensus
T12	$\overline{B•C•D...}$ = $\overline{B}$+$\overline{C}$+$\overline{D}$...	$\overline{B+C+D...}$= $\overline{B}$•$\overline{C}$•$\overline{D}$...	De Morgan's



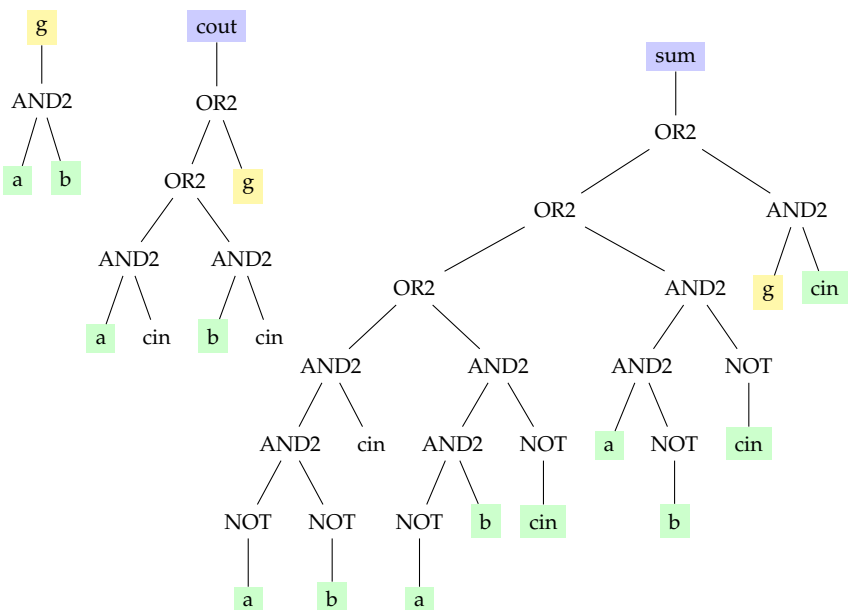




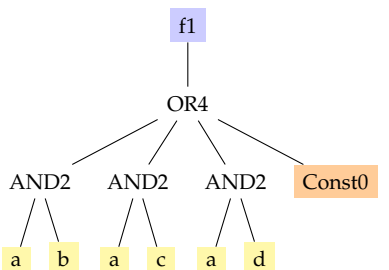


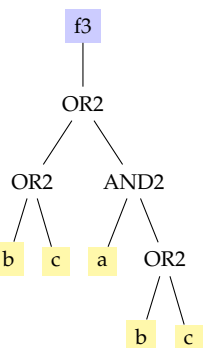
- Global transformations manipulate multiple trees at once

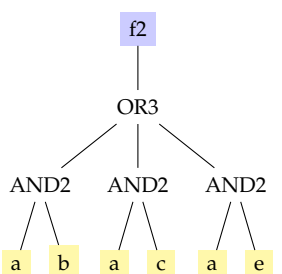
Extract	Extract subtree, create new wire and new tree, make original tree reference new tree
Extract-and-Share	Extract same subtree many times
Inline	Inline tree into the referencing tree
Inline-and-Eliminate	Inline same tree many times, then eliminate tree



- We now have a large number of transformations
- Cost function evaluates any given forest of trees
  - Assign abstract area cost for each generic gate (e.g., number of inputs)
  - Assign abstract delay for each generic gate (e.g., constant delay model)
  - Cost = area + k * delay
- Repeat until no improvement
  - Step 1: Try a transformation
  - Step 2: Evaluate cost
  - Step 3: If cost improves apply transformation
- Occassionally choose bad transformations to avoid local minimum





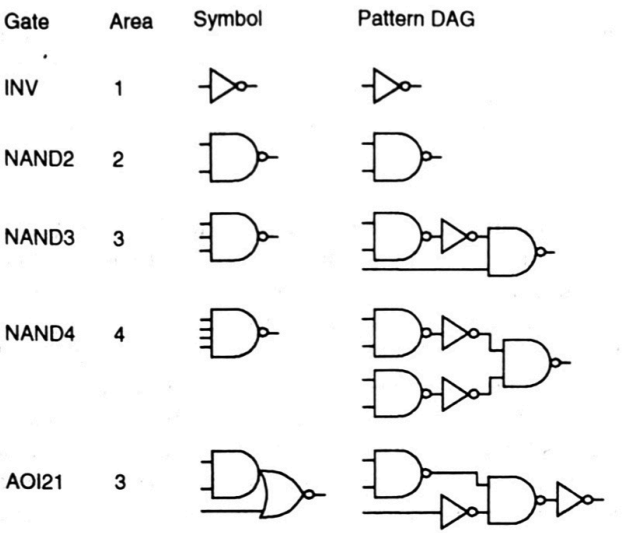


$$f_1 = ab + ac + ad + 0$$
$$f_2 = ab + ac + ae$$
$$f_3 = (b + c) + a(b + c)$$

$$f_1 = ab + ac + ad + 0$$
$$f_2 = ab + ac + ae$$
$$f_3 = (b + c) + a(b + c)$$

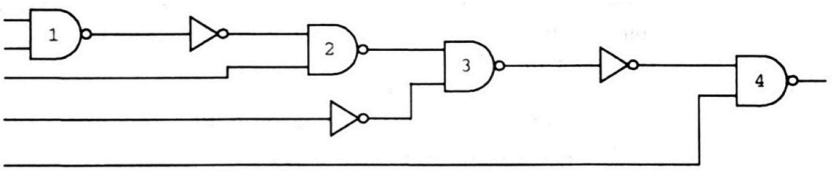
1.4. *Algorithm:* Technology Mapping

- Basic technology mapping algorithm only considered area
- Advanced technology mapping algorithms are timing driven
  - Constant load
  - Quantized loads
  - Quantized loads with area minimization

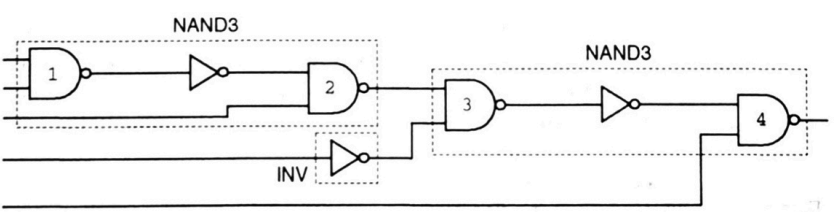


Constant Load

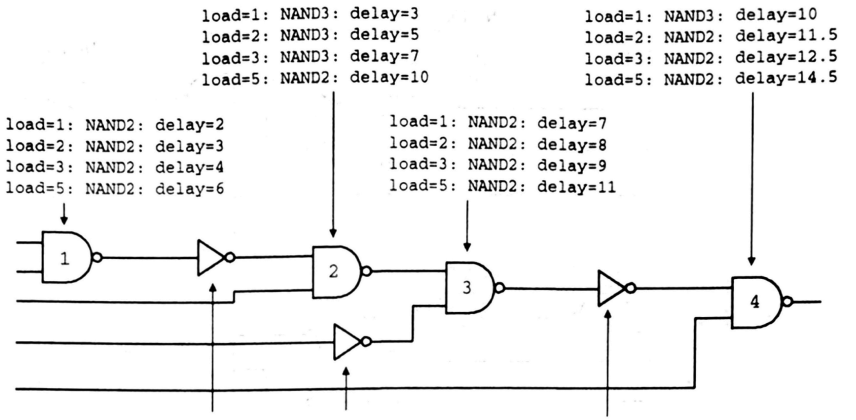
- Assume constant load of 1

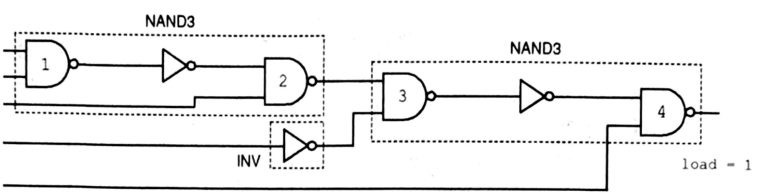


Node	Cover	Cell Delay	Total Delay	Optimal?
0				
1				
2				
3				
4				
5				
6				

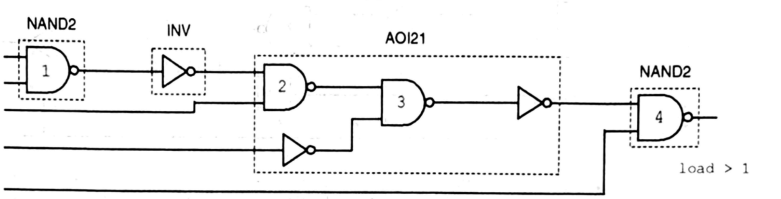


Quantized Load





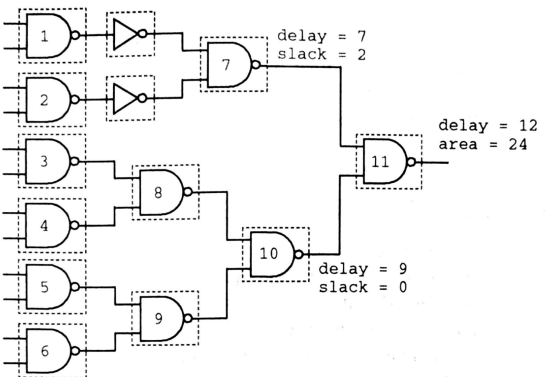
(a)



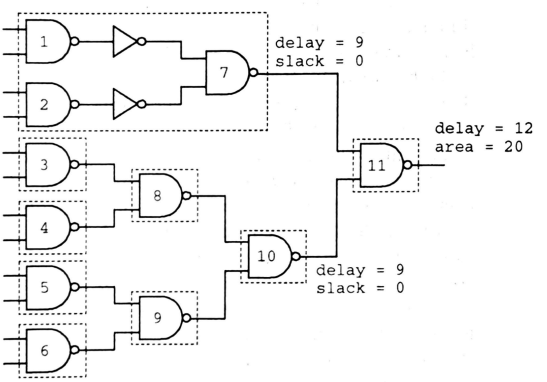
(b)

Quantized Load with Area Minimization

- Determine delays at each node using quantized loads
- Assume a max delay timing constraint from any input to output
- During backtrace choose minimum area solution that still meets max timing constraint at that node



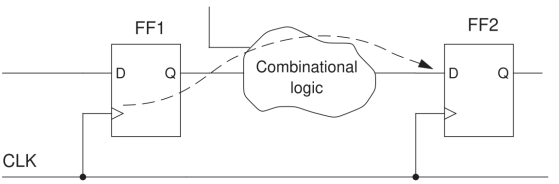
(a)

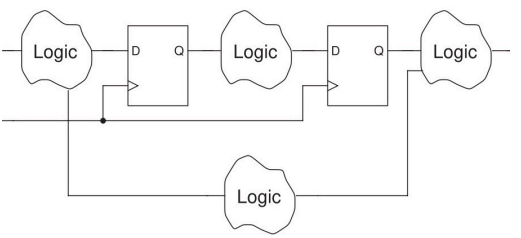


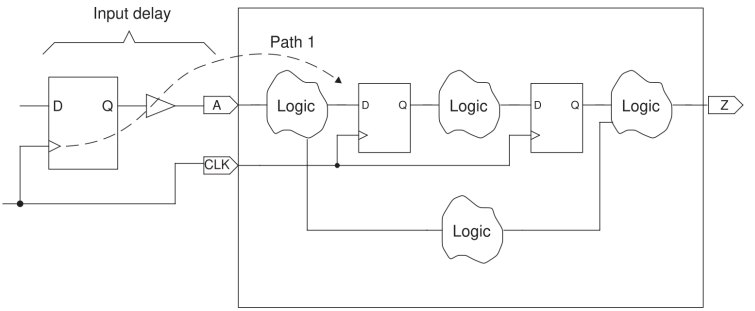
(b)

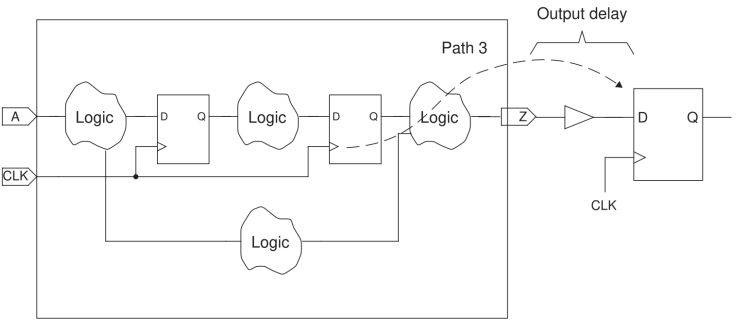
1.5. *Algorithm:* Static Timing Analysis

- Basic static timing analysis only found critical path
- Advanced static timing analysis considers many timing constraints

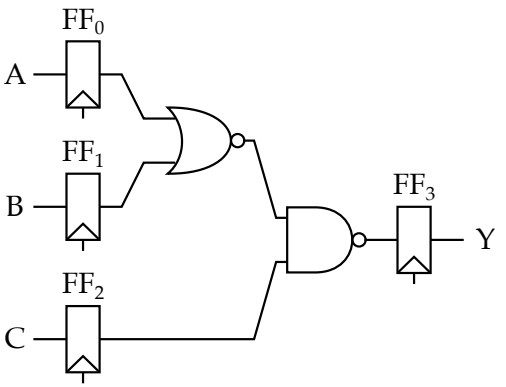





Input delay
Path 1
D
Q
A
Logic
D
Q
Logic
D
Q
Logic
Z
CLK
Logic


Output delay
Path 3
A
Logic
D
Q
Logic
D
Q
Logic
Z
D
Q
CLK
CLK
Logic

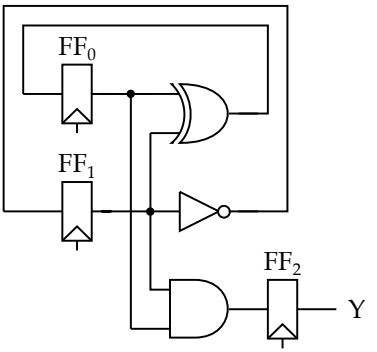
- Static timing analysis analyzes checks to see if all paths satisfy all applicable constraints
- Consider simplified STA with a constant delay model



	t_{pd}	t_{cd}
NAND2	2τ	1τ
NOR2	3τ	1τ
FF (t_{cq})	9τ	2τ

FF (t_{setup})	10τ
FF (t_{hold})	1τ
T_C	30τ

Path Start Point	**Path End Point**	_____ **Constraint**	_____ **Constraint**



	t_{pd}	t_{cd}
NOT	1τ	1τ
AND2	3τ	1τ
XOR2	7τ	1τ
FF (t_{cq})	9τ	2τ

FF (t_{setup})	10τ
FF (t_{hold})	1τ
T_C	23τ

Path Start Point	**Path End Point**	_____ **Constraint**	_____ **Constraint**

1.6. *Algorithm:* Gate-Level Netlist Writer