ECE 6745 Complex Digital ASIC Design

Topic 3: Basic Front-End Flow

School of Electrical and Computer Engineering
Cornell University

revision: 2026-02-10-09-47

1 Front-End Flow **3**

2 Synthesis **6**

2.1. *Data Structure:* Forest of Trees 7

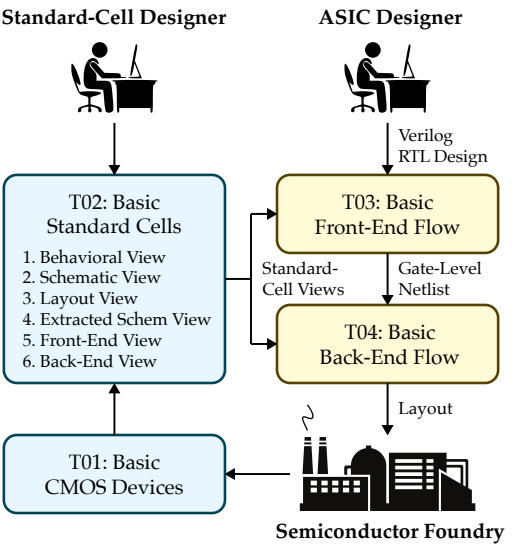
2.2. *Algorithm:* Verilog Reader 8

2.3. *Algorithm:* Technology Mapping 11

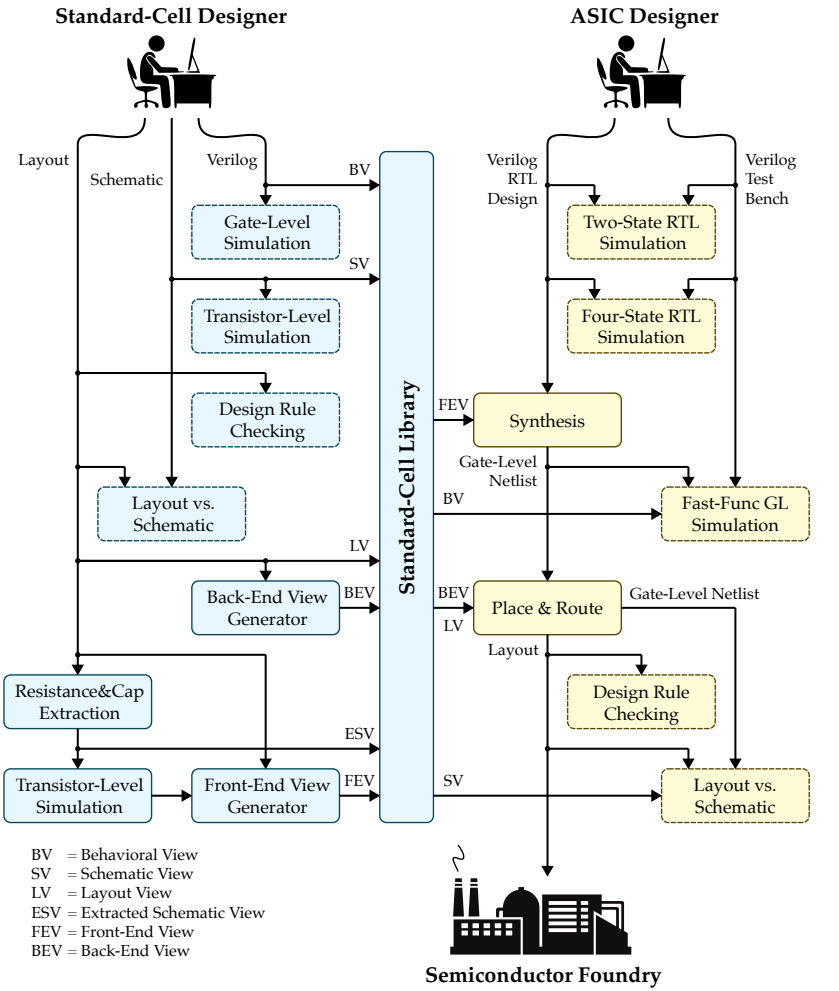
2.4. *Algorithm:* Static Timing Analysis 21

2.5. *Algorithm:* Gate-Level Netlist Writer 24

3 TinyFlow Front-End **25**



1. Front-End Flow



Two-State vs Four-State RTL Simulation

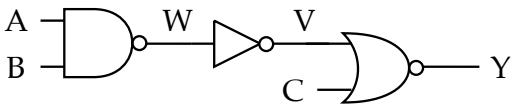
- Two-State Simulation
  - All signal values must be 0 or 1
  - Usually enables much faster simulations
  - Can potentially mask subtle design bugs
- Four-State Simulation
  - Signal values cab be 0, 1, X or Z
  - Can reveal mask subtle design bugs (e.g., X and Z propagation)
  - Usually requires much slower simulations

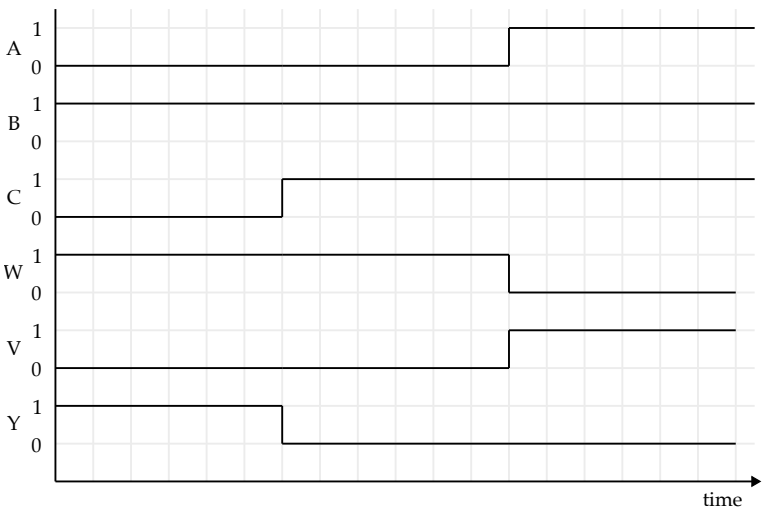
```
1  module AdderRippleCary_2b
2  (
3    input  wire a0,
4    input  wire a1,
5    input  wire b0,
6    input  wire b1,
7    input  wire cin,
8    output wire sum0,
9    output wire sum1,
10   output wire cout
11 );
12
13   assign sum0 = (~a0 & ~b0 &  cin) | (~a0 &  b0 & ~cin) |
14                 ( a0 & ~b0 & ~cin) | ( a0 &  b0 &  cin);
15
16   wire c1; // forgot to assign to c1
17
18   assign sum1 = (~a1 & ~b1 &  c1) | (~a1 &  b1 & ~c1) |
19                 ( a1 & ~b1 & ~c1) | ( a1 &  b1 &  c1);
20
21   assign cout = (a1 & b1) | (a1 & c1) | (b1 & c1);
22
23 endmodule
```

- Two-state simulation would initialize `c1` to either 0 or 1
- Four-state simulation would initialize `c1` to Z

Fast-Functional Gatel-Level Simulation

- Simulate a gate-level netlist without any delays
- Verifies the synthesized gate-level netlist is functionally correct
- Does not verify any timing constraints



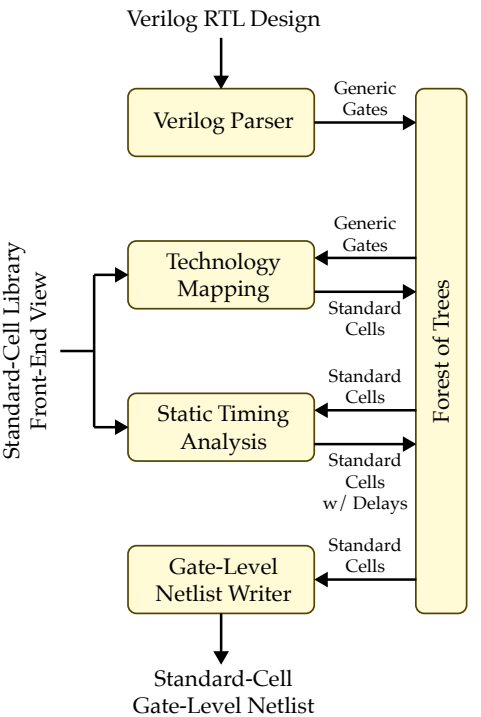


Good ASIC designers are paranoid!

- Never trust an RTL designer → use two-state and four-state simulation to verify a design before using the front-end ASIC flow
- Never trust an ASIC tool → use fast-functional gate-level simulation before using the back-end ASIC flow

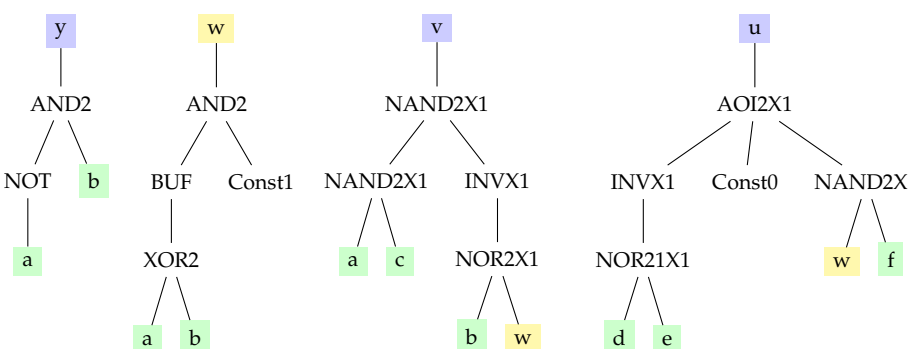
2. Synthesis

- Synthesis takes as input a Verilog RTL design and produces a standard-cell gate-level netlist
- Synthesis Data Structures
  - Forest of trees of generic-gate and standard-cell nodes
- Synthesis Algorithms
  - Verilog Reader: Parses Verilog RTL design into forest of trees of generic logic gates
  - Technology Mapping: Maps trees of generic gates to trees of standard cells
  - Static Timing Analysis: Statically analyze all paths to find critical path
  - Gate-Level Netlist Writer: Outputs a standard-cell gate-level netlist



2.1. *Data Structure:* Forest of Trees

- List of input ports, output ports, and wires
- Forest of trees of nodes with one tree per output port and wire
  - Nodes represent either generic gates or standard cells
  - Edges represent nets between generic gates or standard cells
  - Root of the tree is the associated output port or wire
  - Leaves of the tree are input ports, wires, or constants
- Generic-Gate Nodes
  - These represent generic logic gates *not standard cells!*
  - Do not have any associated area or delay
  - Named without X1 suffix
  - BUF, INV, NOT, AND2, OR2, XOR2, NAND2, NOR2, XNOR2
- Standard-Cell Nodes
  - These represent the actual standard cells
  - Have any associated area and delay from front-end view
  - Usually named with X1 suffix
  - INVX1, NAND2X1, NOR2X1, AOI2X1, TIEHI, TIELO



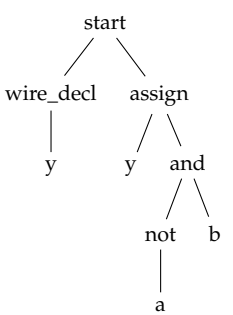
2.2. *Algorithm:* Verilog Reader

- 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

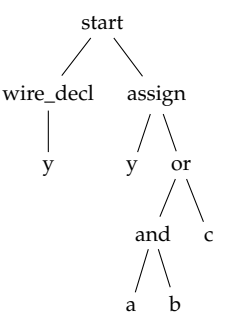
- Verilog grammar for single-bit wire declarations and assign statements with single-bit operators
- Written in extended Backus-Naur form (EBNF)
- Assume precedence inferred from rule order

```
start: stmt+
?stmt: wire_decl | assign

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

?expr: SIGNAL | LITERAL | "(" expr ")"
     | "~" expr       -> not
     | expr "&" expr -> and
     | expr "|" expr -> or
     | expr "^" expr -> xor

SIGNAL:  /[a-zA-Z_][a-zA-Z0-9_]*/
LITERAL: /\d+'[bB][01]+/ | /\d+/
```

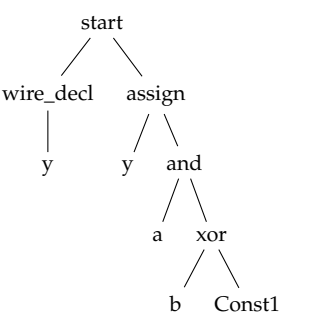
```
wire y;
assign y = ~a & b;
```




```
wire y;
assign y = (a & b) | c;
```



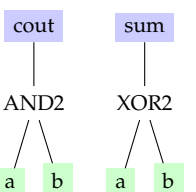

```
wire y;
assign y = a & (b ^ 1);
```

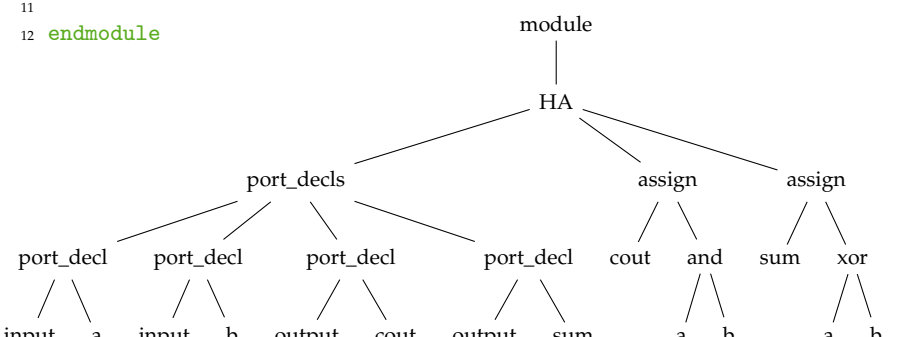


- Extend grammar to support module interface declaration

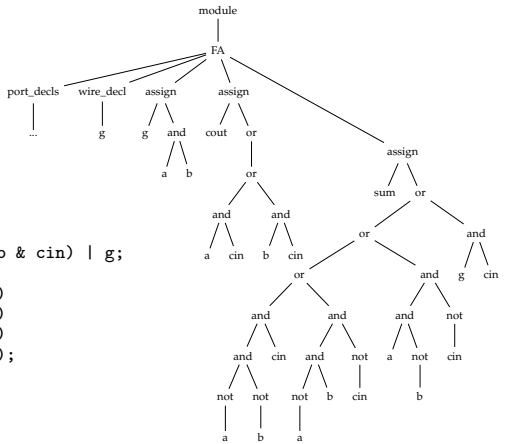
```
?start: module
module: "module" MNAME port_decl_list? ";" stmt* "endmodule"

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

DIR:   ( "input" | "output" )
MNAME: /[a-zA-Z_][a-zA-Z0-9_]*/
...
```

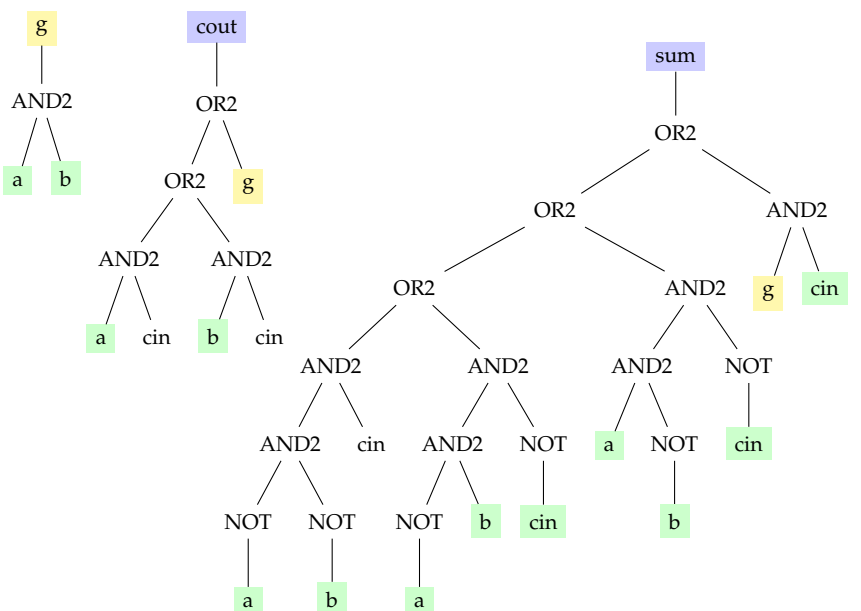
```
1  module HA
2  (
3    input  wire a,
4    input  wire b,
5    output wire cout,
6    output wire sum
7  );
8
9    assign cout = a & b;
10   assign sum = a ^ b;
11
12 endmodule
```





```
1  module FA
2  (
3    input  wire a,
4    input  wire b,
5    input  wire cin,
6    output wire cout,
7    output wire sum
8  );
9
10   wire g;
11   assign g = a & b;
12
13   assign cout = (a & cin)  | (b & cin) | g;
14
15   assign sum = (~a & ~b &  cin)
16              | (~a &  b & ~cin)
17              | ( a & ~b & ~cin)
18              | ( g      &  cin);
19
20 endmodule
```



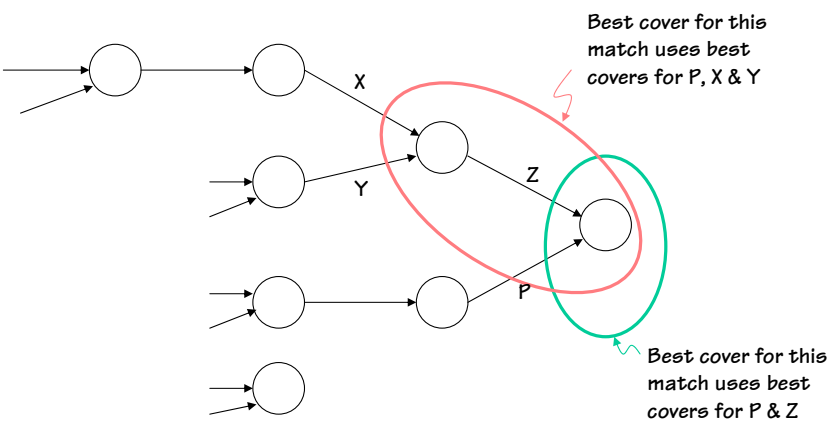


2.3. *Algorithm:* Technology Mapping

- Canonicalize: Replace every generic gate with equivalent tree of just NAND2 and INV generic gates; determine equivalent trees of NAND2 and INV generic gates for each standard cell
- Cover: Use *dynamic programming* to determine optimal way to cover each node in a canonicalized tree of generic gates with equivalent standard cells; goal is to minmize total area
- Traceback: Determine final optimal cover of all trees in the forest

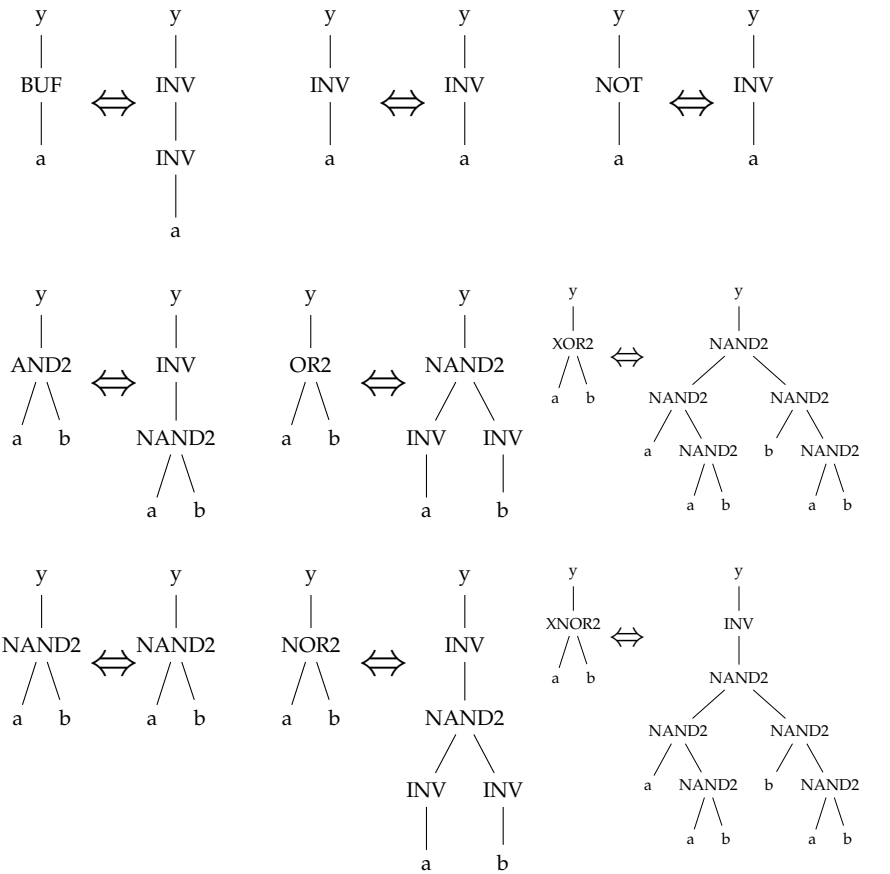
Dyanmic Programming

- Dynamic programming can be used if two properties are true:
  - Optimal Substructure: an optimal solution to the whole problem can be constructed from optimal solutions of subproblems
  - Overlapping Subproblems: the problem resuses the same subproblem many times, meaning we can reuse previous optimal solutions

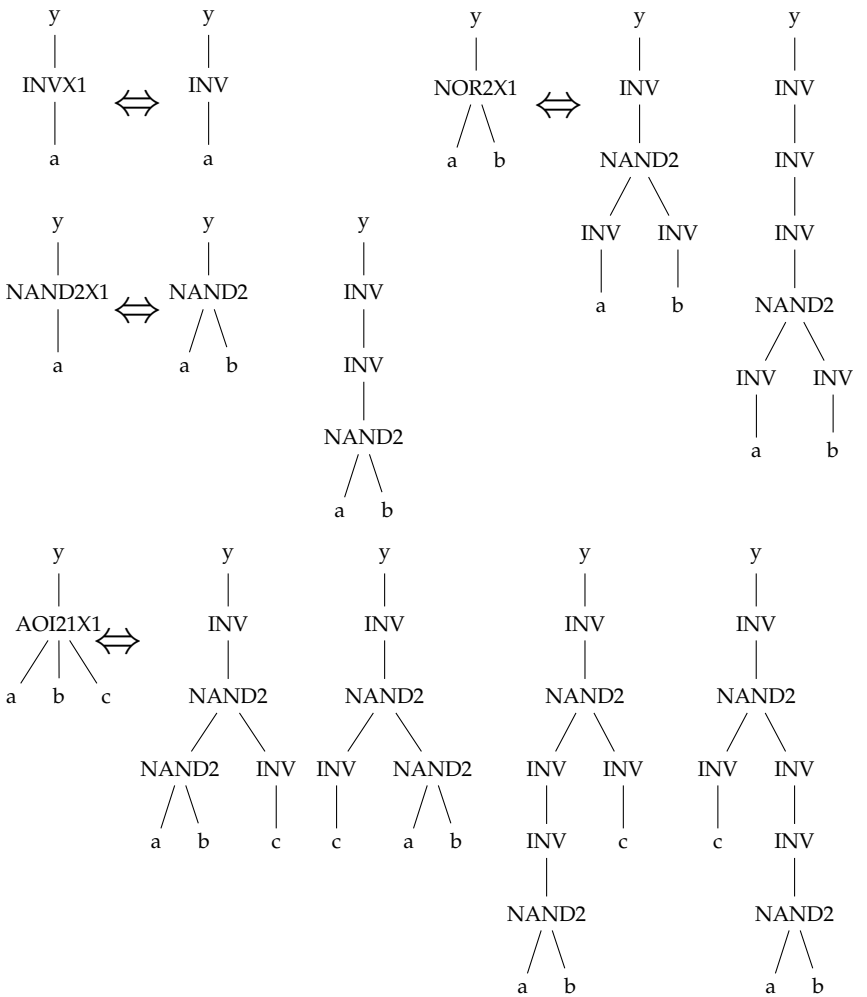


Canonicalize

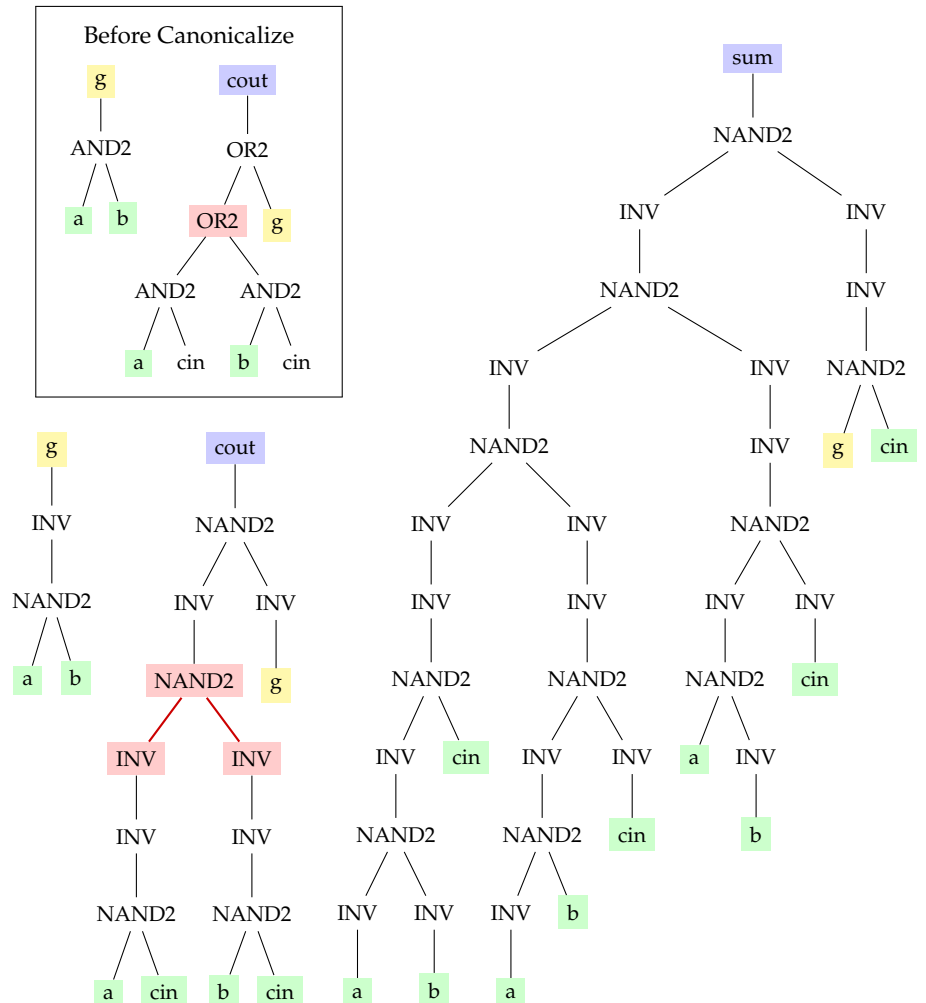
- Every generic gate has a NAND2 and INV canonical form



- Every standard cell has a list of NAND2 and INV cananoical forms



Full Adder

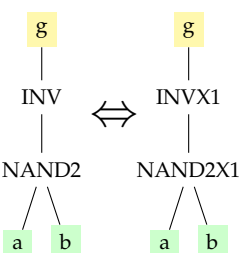

Before Canonicalize
g
cout
AND2
OR2
a
b
OR2
g
AND2
AND2
a
cin
b
cin
sum
NAND2
INV
INV
NAND2
INV
INV
NAND2
INV
INV
NAND2
g
cin
g
cout
INV
NAND2
NAND2
INV
INV
INV
INV
NAND2
a
b
NAND2
g
INV
INV
NAND2
NAND2
NAND2
INV
INV
INV
INV
INV
INV
INV
cin
INV
INV
NAND2
NAND2
cin
a
INV
NAND2
NAND2
INV
INV
INV
b
a
cin
b
cin
a
b
a
b

Cover

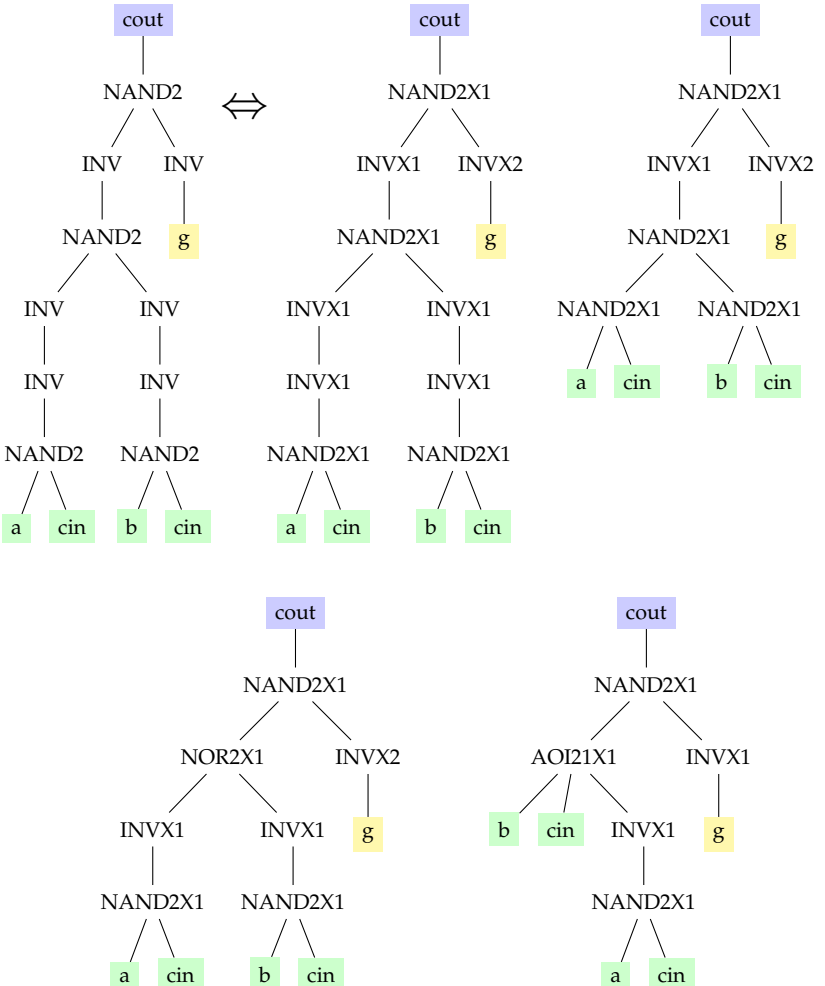
- Goal is to optimally cover the canonicalized forest of trees using the canonical versions of the standard cells
- Area model from front-end view

Standard Cell	Area (λ^2)	Normalized Area
INVX1	1536	3
NAND2X1	2048	4
NOR2X1	2048	4
AOI21X1	3072	5

- Optimally covering g tree in full adder is trivial (area = 3+4 = 7)

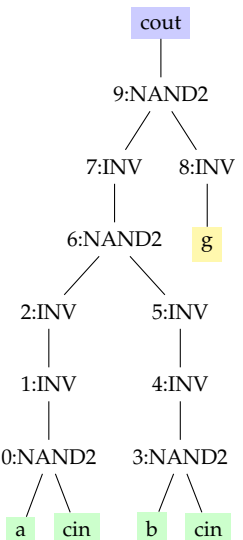


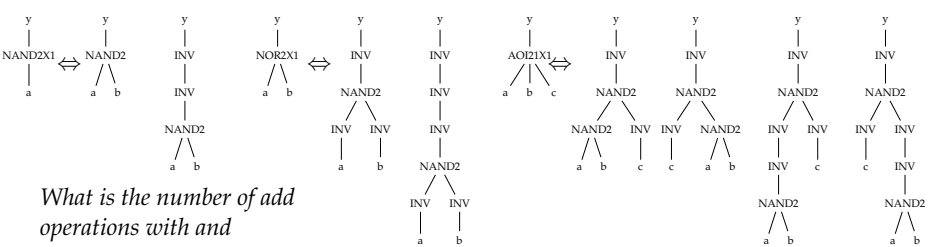
- Many possible covers of cout tree in full adder


cout
NAND2
INV
INV
NAND2
g
INV
INV
INV
INV
NAND2
NAND2
a
cin
b
cin
⇔
cout
NAND2X1
INVX1
INVX2
NAND2X1
g
INVX1
INVX1
INVX1
INVX1
NAND2X1
NAND2X1
a
cin
b
cin
cout
NAND2X1
INVX1
INVX2
NAND2X1
g
NAND2X1
NAND2X1
a
cin
b
cin
cout
NAND2X1
NOR2X1
INVX2
INVX1
INVX1
g
NAND2X1
NAND2X1
a
cin
b
cin
cout
NAND2X1
AOI21X1
INVX1
b
cin
INVX1
g
NAND2X1
a
cin

Standard Cell	Area
INVX1	3
NAND2X1	4
NOR2X1	4
AOI21X1	5

Node	Cover	Area	Optimal?
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			

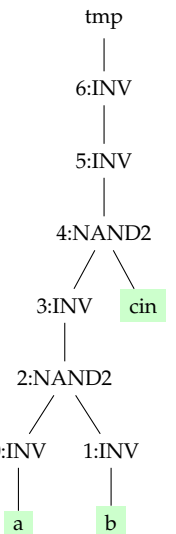


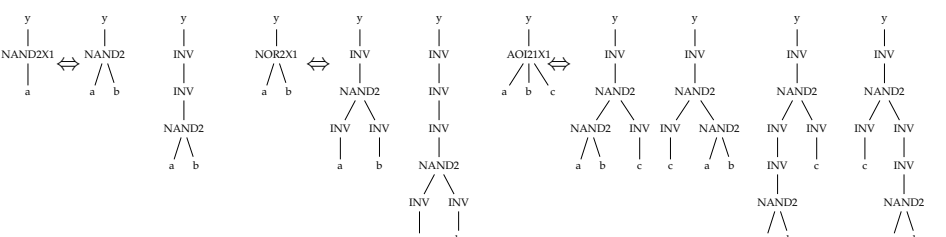


What is the number of add operations with and without dynamic programming?

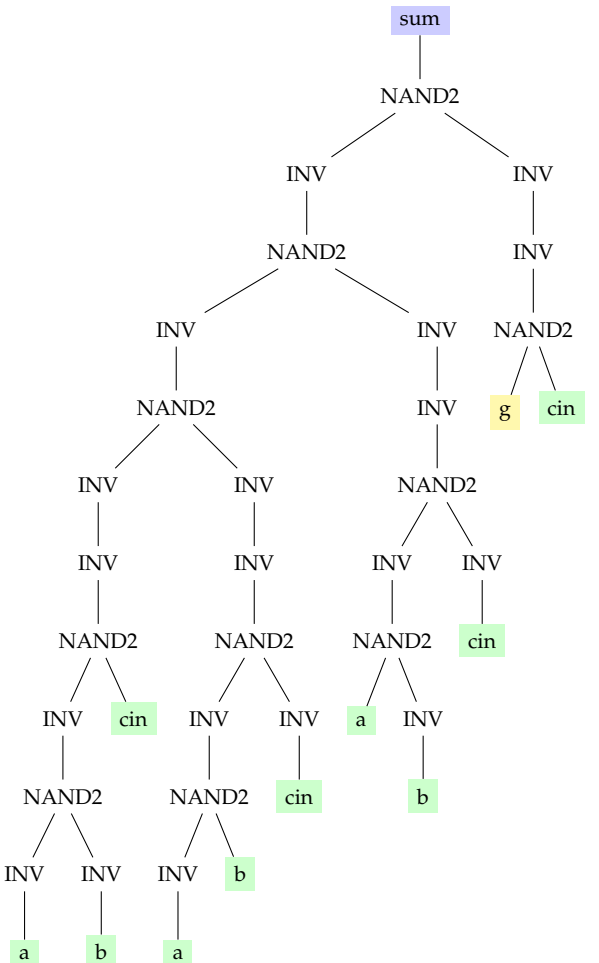
Standard Cell	Area
INVX1	3
NAND2X1	4
NOR2X1	4
AOI21X1	5

Node	Cover	Area	Optimal?



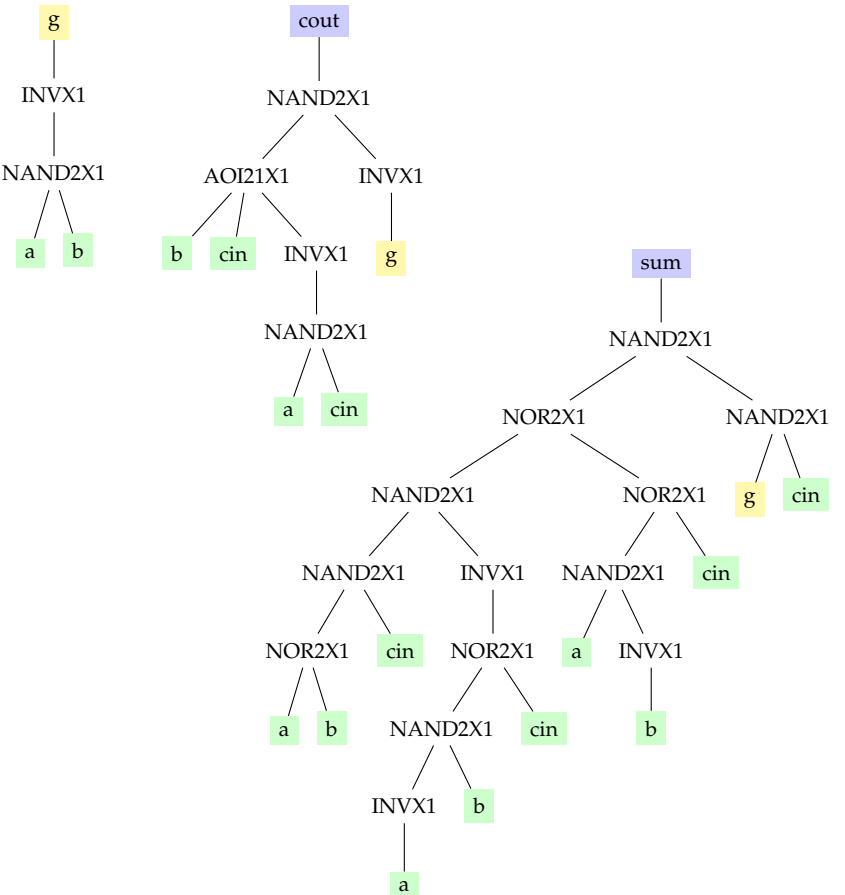


- How many canonical forms of AOI21X1 can you find in this tree?



Traceback

- Start from root to determine optimal cover for entire tree



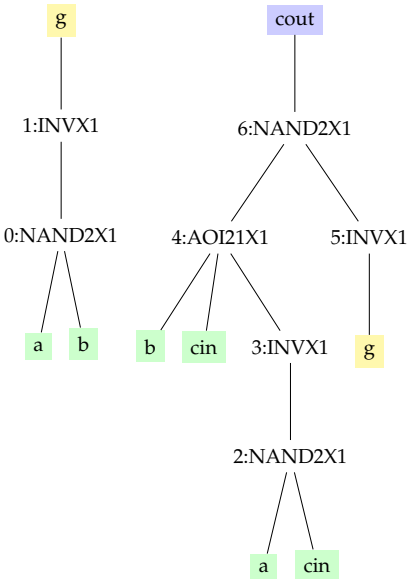
2.4. *Algorithm:* Static Timing Analysis

- Calculate Loads: Calculate the output load of each standard cell taking into account fanout wires
- Calculate Arrivals: Starting from the leaves determine max arrival time for all nodes
- Traceback: Determine final critical path across all trees in forest

- Key information required from the front-end view
  - *Input capacitance* of each input pin for each standard cell
  - *Linear delay equation* for every output pin for each standard cell

$$
\begin{aligned}
C_{g,invx1} &= 3C \\
C_{g,nand2x1} &= 4C \\
C_{g,nor2x1} &= 5C \\
C_{g,aoi21x1} &= 5C \text{ or } 6C
\end{aligned}
$$

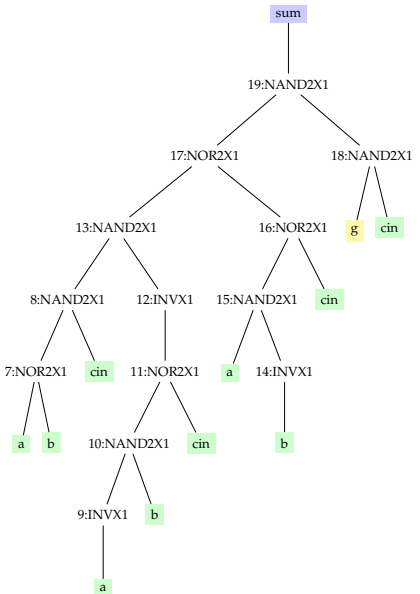
$$
\begin{aligned}
t_{pd,invx1} &= 3RC + RC_L \\
t_{pd,nand2x1} &= 6RC + RC_L \\
t_{pd,nor2x1} &= 6RC + RC_L \\
t_{pd,aoi21x1} &= 11RC + RC_L
\end{aligned}
$$



	C_L	t_i	t_L	t_{pd}	Path Delay
0					
1					
2					
3					
4					
5					
6					

$$C_{g,invx1} = 3C$$
$$C_{g,nand2x1} = 4C$$
$$C_{g,nor2x1} = 5C$$
$$C_{g,aoi21x1} = 5C \text{ or } 6C$$

$$t_{pd,invx1} = 3RC + RC_L$$
$$t_{pd,nand2x1} = 6RC + RC_L$$
$$t_{pd,nor2x1} = 6RC + RC_L$$
$$t_{pd,aoi21x1} = 11RC + RC_L$$



$$C_{g,invx1} = 3C$$
$$C_{g,nand2x1} = 4C$$
$$C_{g,nor2x1} = 5C$$
$$C_{g,aoi21x1} = 5C \text{ or } 6C$$

$$t_{pd,invx1} = 3RC + RC_L$$
$$t_{pd,nand2x1} = 6RC + RC_L$$
$$t_{pd,nor2x1} = 6RC + RC_L$$
$$t_{pd,aoi21x1} = 11RC + RC_L$$

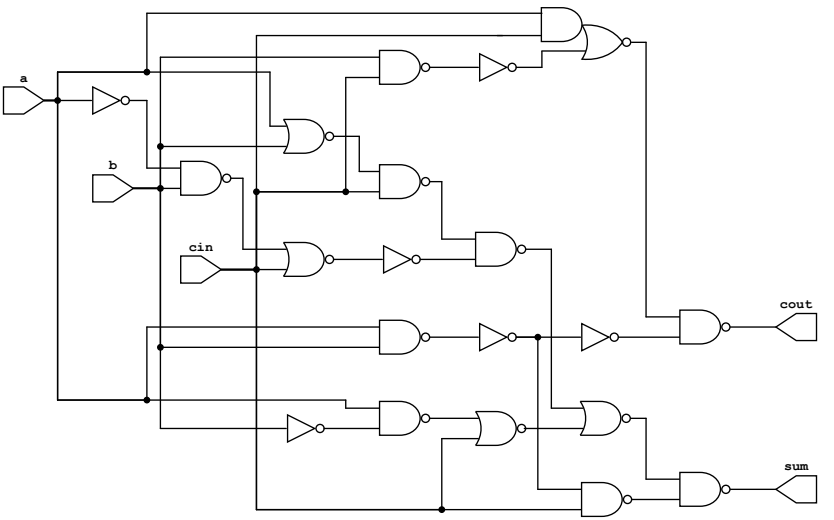
	C_L	t_i	t_L	t_{pd}	Path Delay
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					

$$C_{g,a} =$$

$$C_{g,b} =$$

$$C_{g,cin} =$$

2.5. *Algorithm:* Gate-Level Netlist Writer



3. TinyFlow Front-End