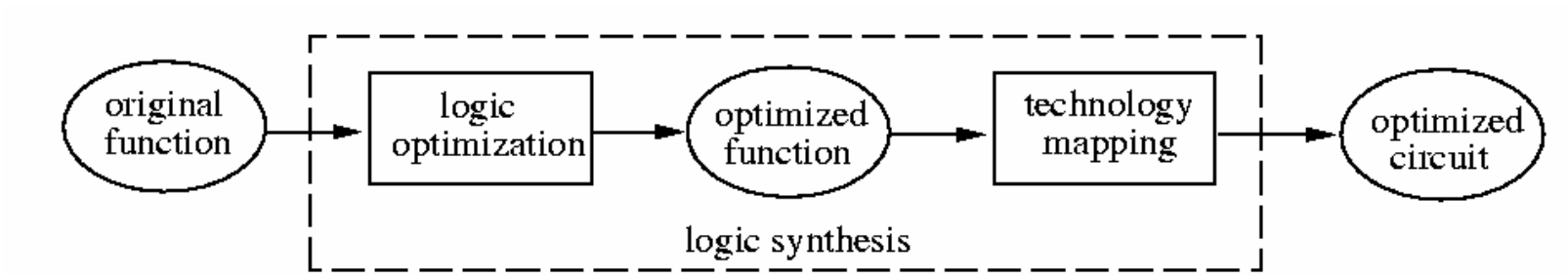
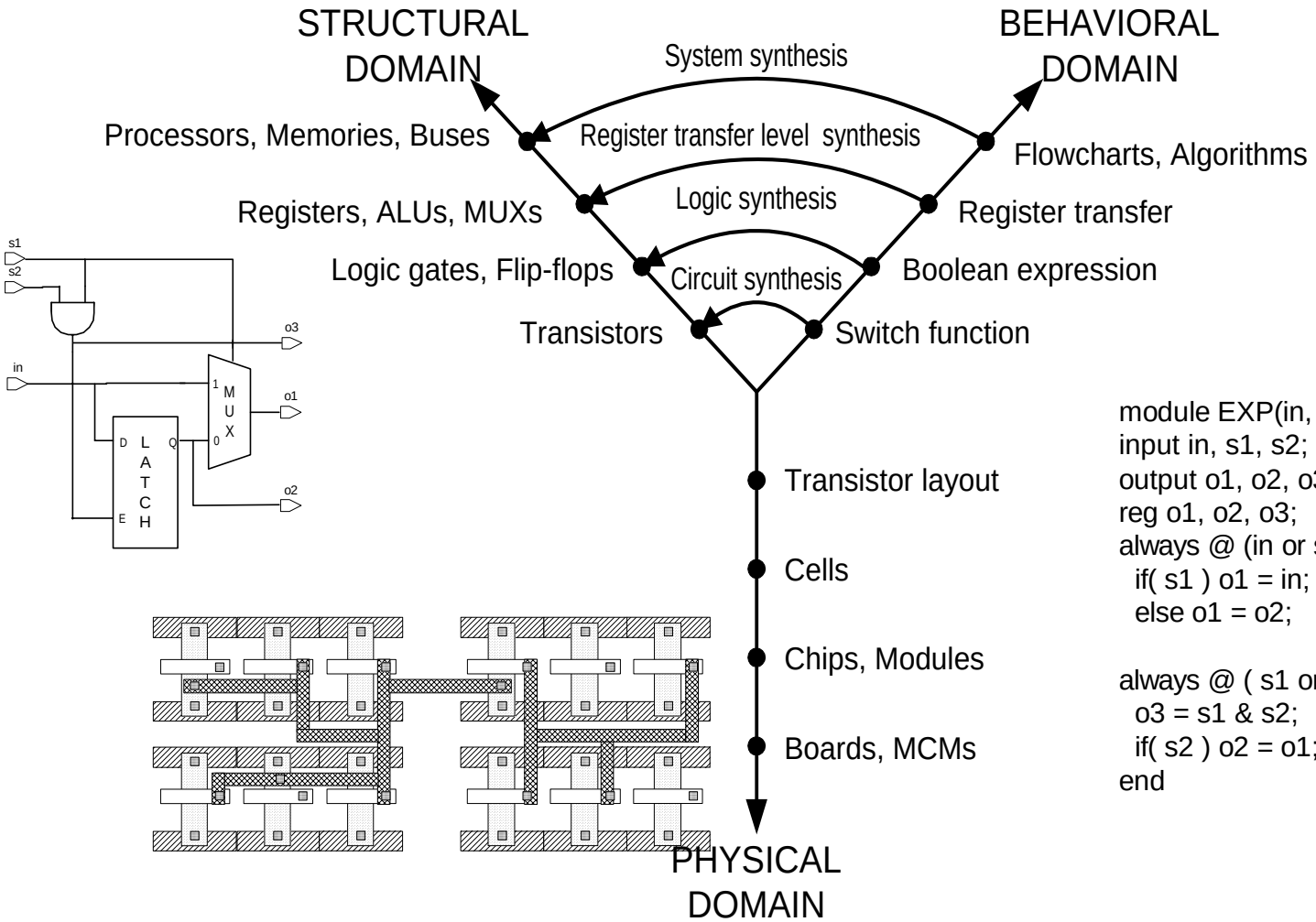
Unit 3: Logic Synthesis

- Course contents
  - Synthesis overview
  - RTL synthesis
  - Logic optimization
  - Technology mapping
  - Timing optimization
  - Synthesis for low power
  - * Retiming
- Readings
  - Chapter 11



*: optional

Levels of Design



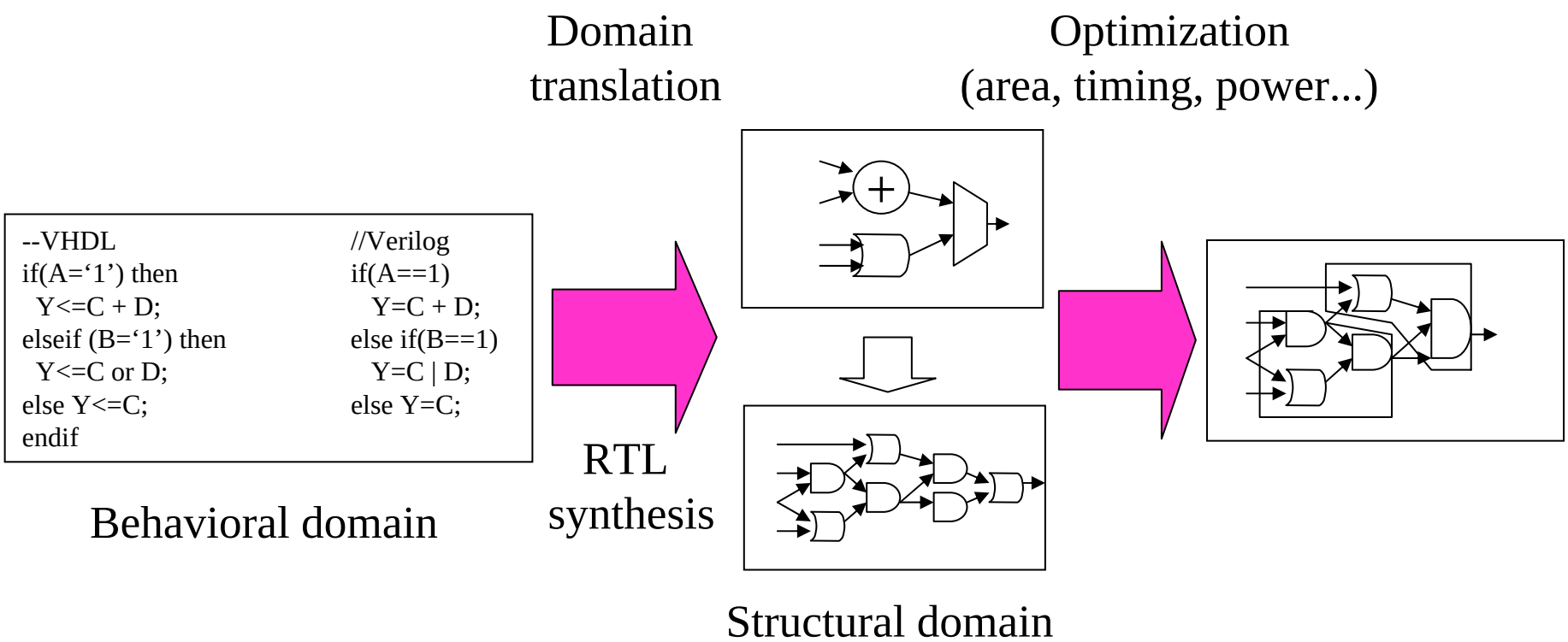

```
module EXP(in, s1, s2, o1, o2, o3);
input in, s1, s2;
output o1, o2, o3;
reg o1, o2, o3;
always @ (in or s1 or s2)
  if( s1 ) o1 = in;
  else o1 = o2;

always @ ( s1 or s2 or o1) begin
  o3 = s1 & s2;
  if( s2 ) o2 = o1;
end
```

Synthesis

- Translate HDL descriptions into logic gate networks (structural domain) in a particular library
- Advantages
  - Reduce time to generate netlists
  - Easier to retarget designs from one technology to another
  - Reduce debugging effort
- Requirement
  - **Robust** HDL synthesizers

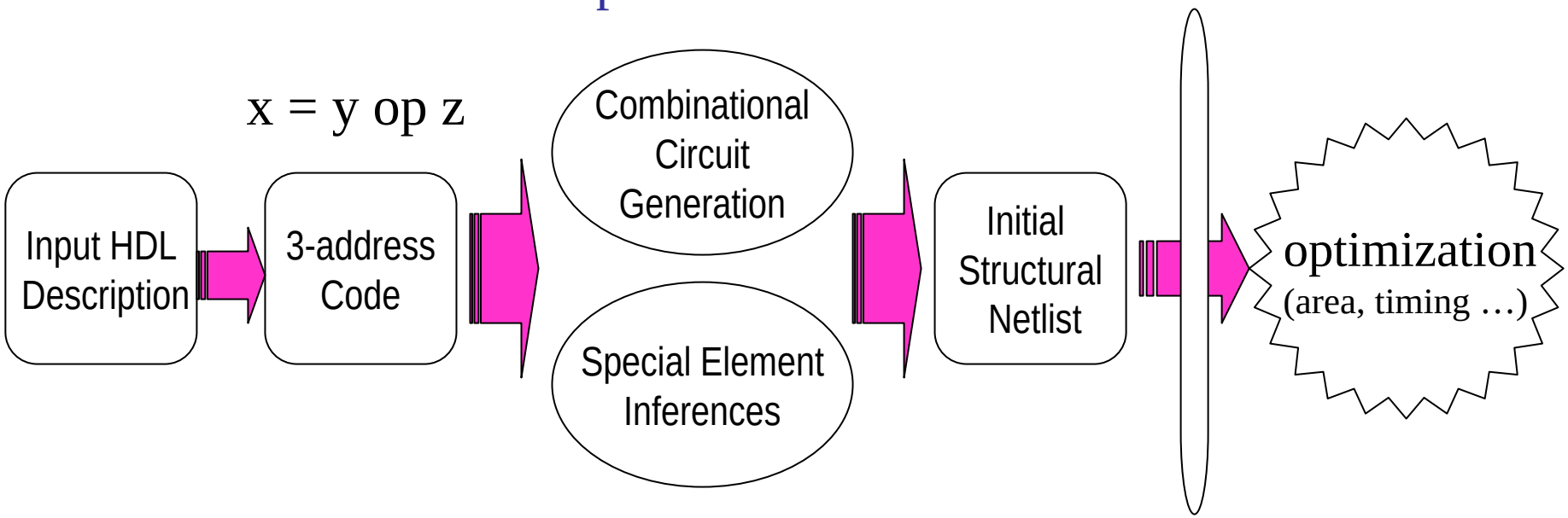
HDL Synthesis

Synthesis = Domain Translation + Optimization



Domain Translation

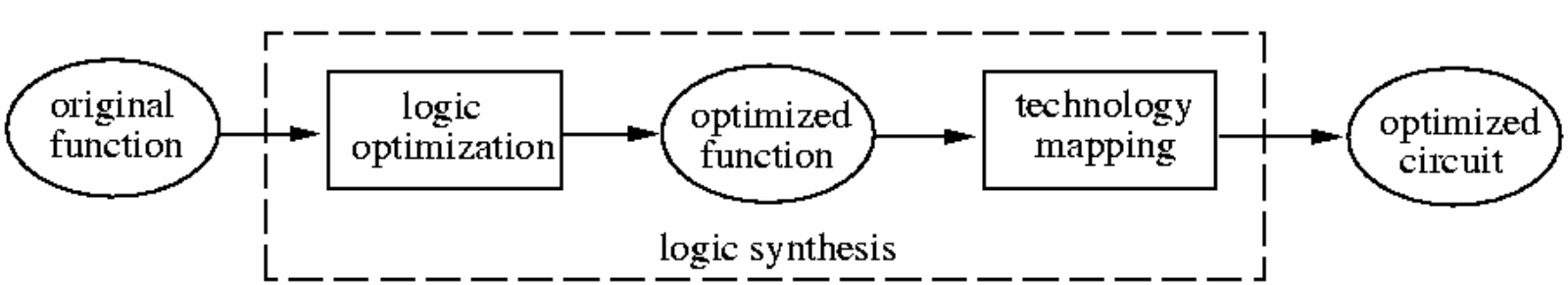
Consistent with data manipulation functions



Consistent with special semantics

Optimization

- **Technology-independent** optimization: **logic optimization**
  - Work on Boolean expression equivalent
  - Estimate size based on # of literals
  - Use simple delay models
- **Technology-dependent** optimization: **technology mapping/library binding**
  - Map Boolean expressions into a particular cell library
  - May perform some optimizations in addition to simple mapping
  - Use more accurate delay models based on cell structures



Two-Level Logic Optimization

- Two-level logic representations
  - Sum-of-product form
  - Product-of-sum form
- Two-level logic optimization
  - Key technique in logic optimization
  - Many efficient algorithms to find a near minimal representation in a practical amount of time
  - In commercial use for several years
  - Minimization criteria: **number of product terms**
- Example: $F = XYZ + X\bar{Y}\bar{Z} + X\bar{Y}Z + \bar{X}YZ + XY\bar{Y}Z$

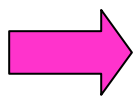
$F = X\bar{Y} + YZ$

Multi-Level Logic Optimization

- Translate a combinational circuit to meet performance or area constraints
  - Two-level minimization
  - Common factors or kernel extraction
  - Common expression resubsitution
- In commercial use for several years
- Example:

$$f1 = abcd + abce + \bar{a}\bar{b}cd + a\bar{b}\bar{c}\bar{d} + \bar{a}c + cdf + ab\bar{c}\bar{d}\bar{e} + a\bar{b}\bar{c}d\bar{f}$$

$$f2 = bdg + \bar{b}dfg + \bar{b}\bar{d}g + b\bar{d}eg$$



$$f1 = c(\bar{a} + x) + a\bar{c}\bar{x}$$

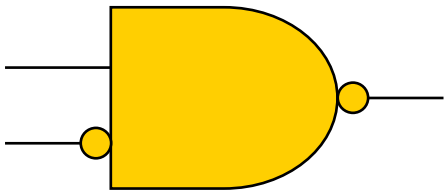
$$f2 = gx$$

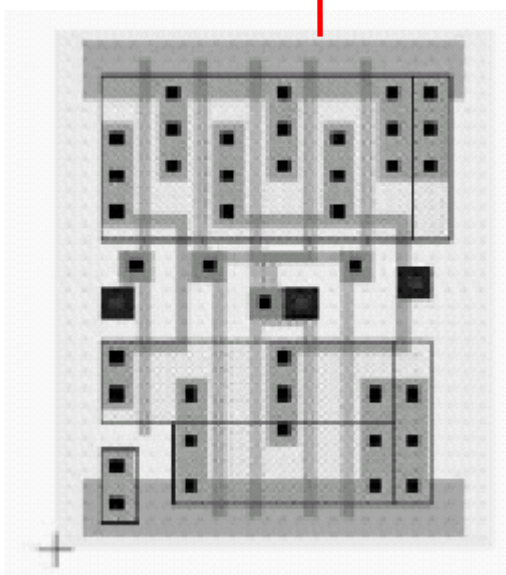
$$x = d(b + f) + \bar{d}(\bar{b} + e)$$

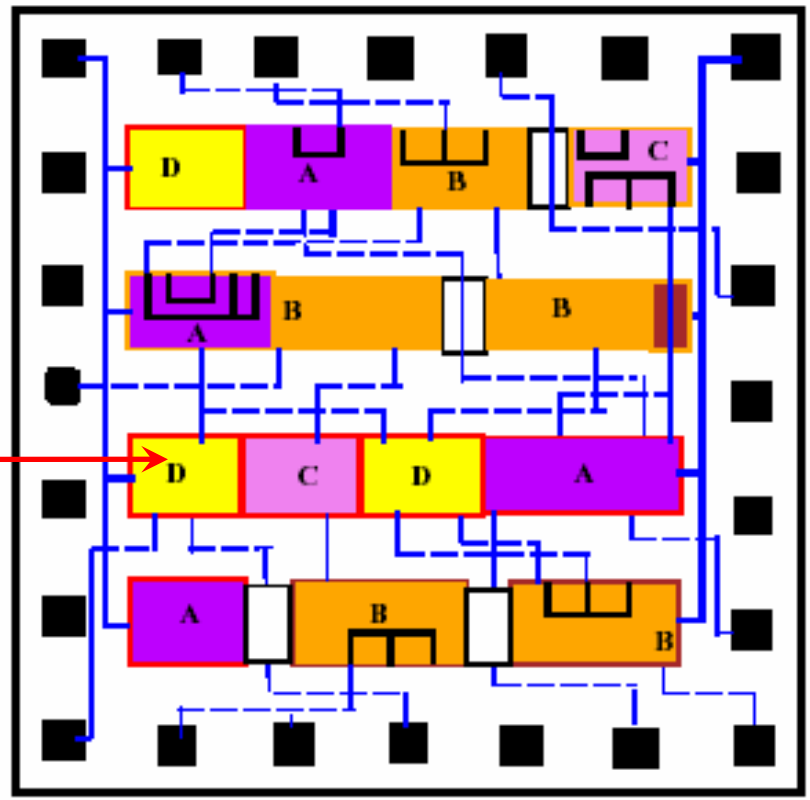
Technology Mapping

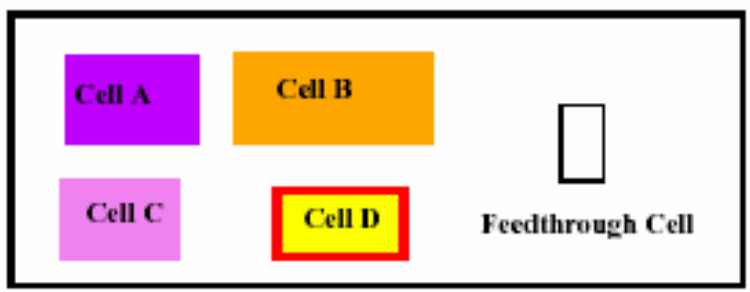
- Goal: translation of a technology independent representation (e.g. Boolean networks) of a circuit into a circuit in a given technology (e.g. standard cells) with optimal cost
- Optimization criteria:
  - Minimum area
  - Minimum delay
  - Meeting specified timing constraints
  - Meeting specified timing constraints with minimum area
- Usage:
  - Technology mapping after technology independent logic optimization
  - Technology translation

Standard Cells for Design Implementation



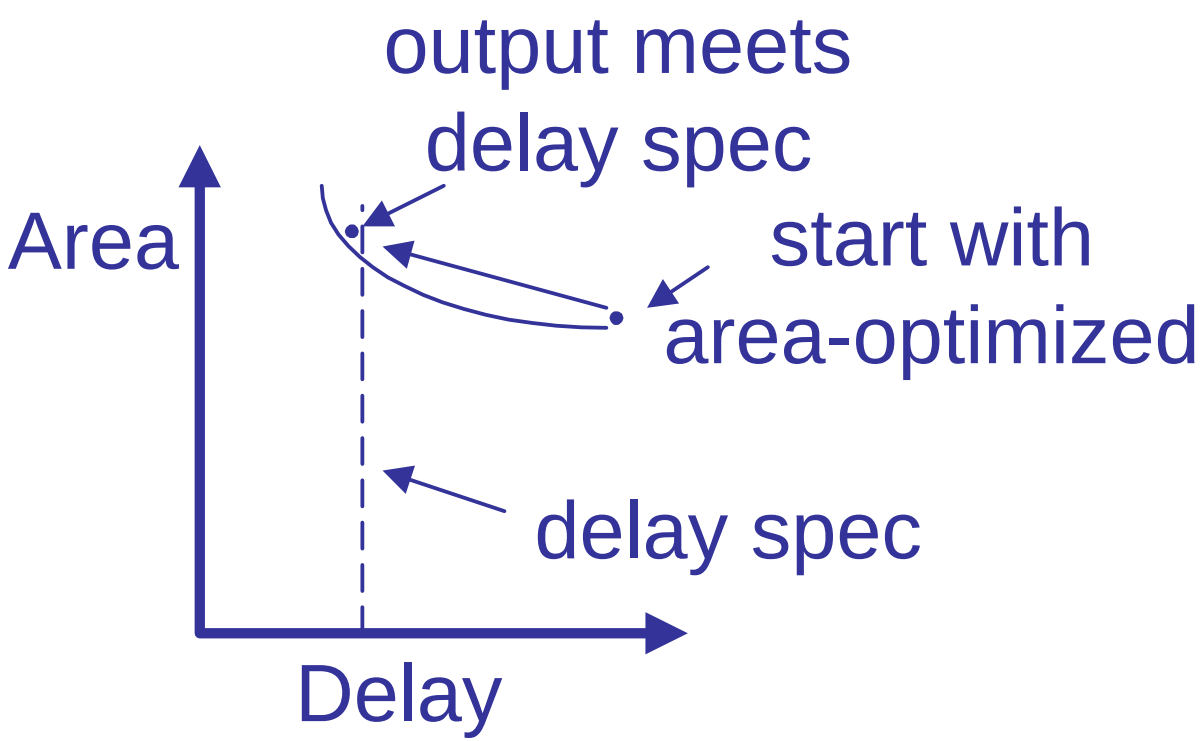






Timing Optimization

- There is always a trade-off between area and delay
- Optimize timing to meet delay spec. with minimum area

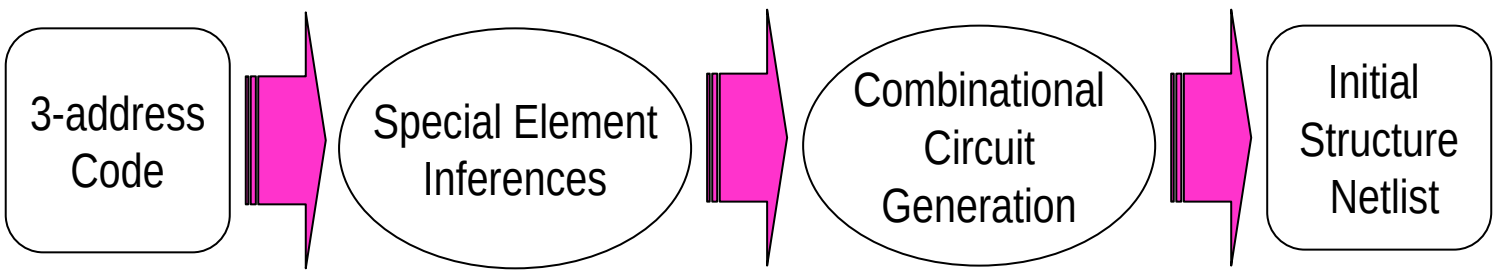


Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level logic optimization
  - Multi-level logic optimization
- Technology mapping
- Timing optimization
- Synthesis for low power
- Retiming

Typical Domain Translation Flow

- Translate original HDL code into 3-address format
- Conduct special element inferences before combinational circuit generation
- Conduct special element inferences process by process (local view)

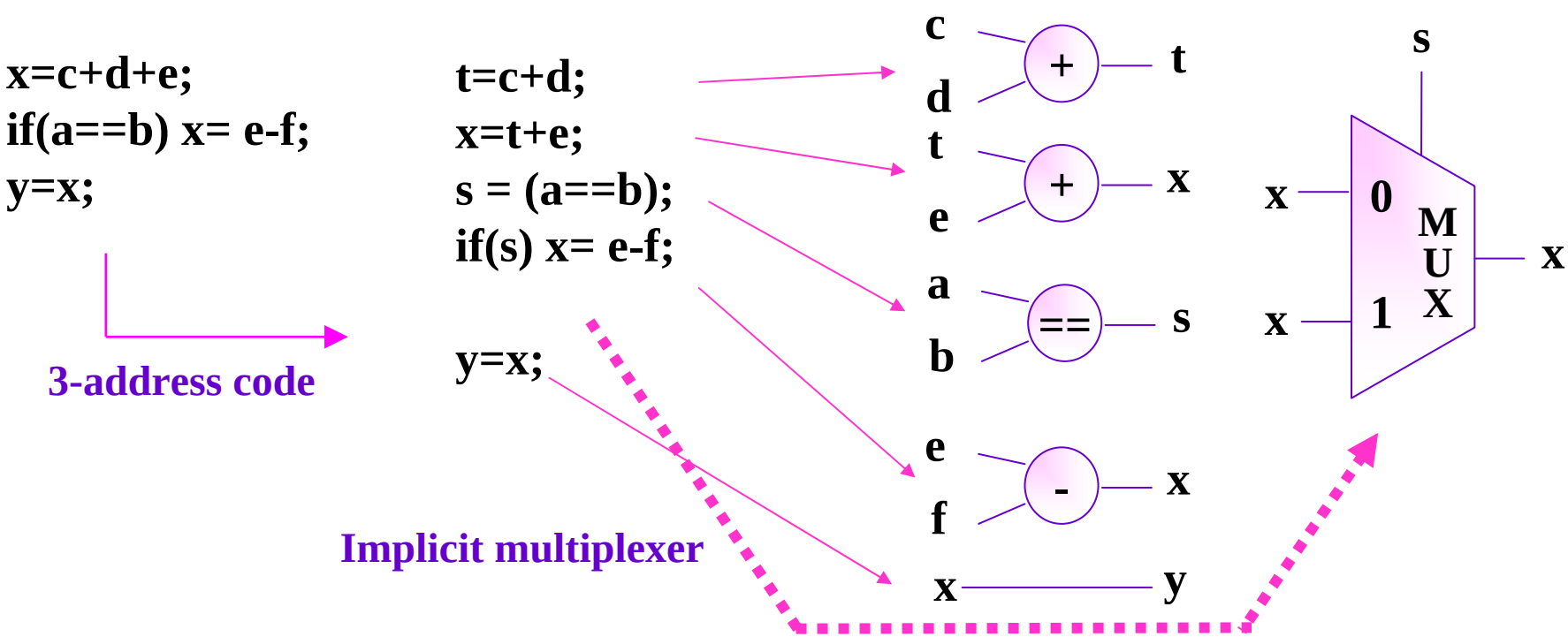


Combinational Circuit Generation

- Functional unit allocation
  - Straightforward mapping with 3-address code
- Interconnection binding
  - Using control/data flow analysis

Functional Unit Allocation

- 3-address code
  - x = y op z in general form
  - Function unit op with inputs y and z and output x



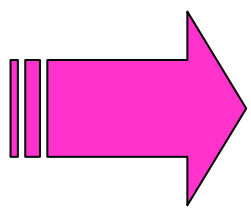
Interconnection Binding

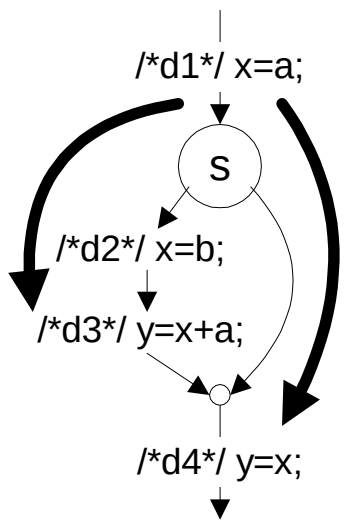
- Need the dependency information among functional units
  - Using **control/data flow analysis**
  - A traditional technique used in compiler design for a variety of code optimizations
  - Statically analyze and compute the set of assignments reaching a particular point in a program

Control/Data Flow Analysis

- Terminology
  - A **definition** of a variable x
    - An assignment assigns a value to the variable x
  - d1 can reach d4 but cannot reach d3
    - d1 is killed by d2 before reaching d3
- A definition can only be affected by those definitions being able to reach it
- Use a set of data flow equations to compute which assignments can reach a target assignment

```
/*d1*/  x = a;
        if(s) begin
/*d2*/      x = b;
/*d3*/      y = x + a;
        end
/*d4*/  y = x;
```





Combinational Circuit Generation

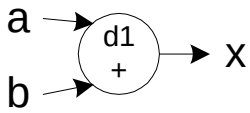
```
always @ (x or a or b or c or d or s)
begin
/*d1*/   x = a + b;
/*d2*/   if ( s ) x = c – d;
/*d3*/   else x = x;
/*d4*/   y = x;
end
```

Input HDL

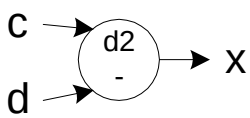
```
always @ (x or a or b or c or d or s)
begin
/*d1*/   x = a + b;
/*d2*/   if ( s ) x = c – d;
/*d3*/   else x = x;
/*d4*/   x = s mux x;
/*d5*/   y = x;
end
```

Modified 3-address code

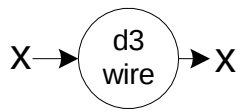
In[d1]={d4, d5}



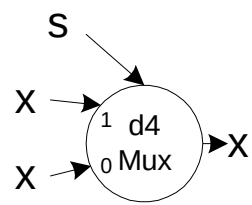
In[d2]={d1, d5}



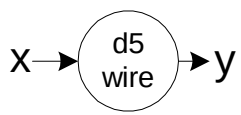
In[d3]={*d1, d5}



In[d4]={*d2, *d3, d5}

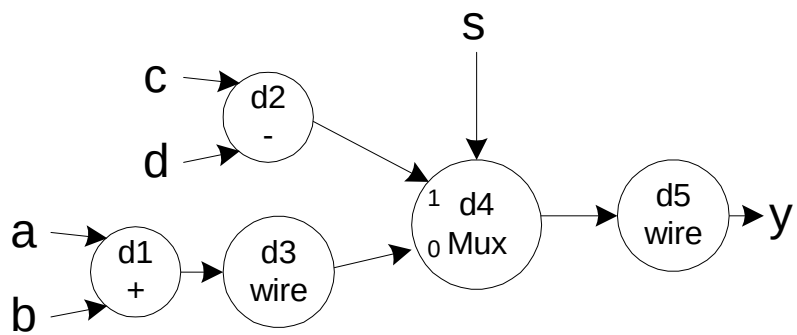


In[d5]={*d4, d5}

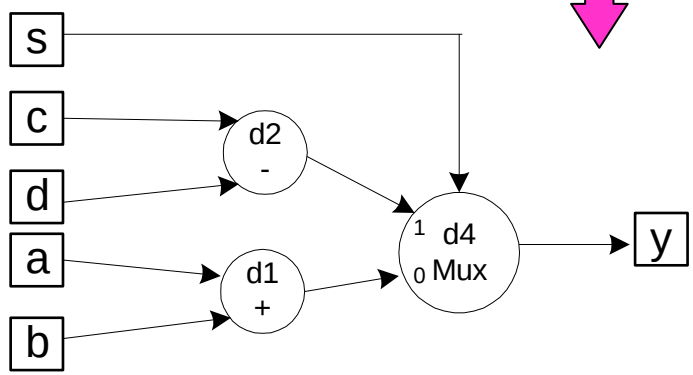


Functional unit allocation

computed by control/ data flow analysis



Interconnection binding



Final result

Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level logic optimization
  - Multi-level logic optimization
- Technology mapping
- Timing optimization
- Synthesis for low power
- Retiming

Special Element Inferences

- Given a HDL code at RTL, three special elements need to be inferred to keep the special semantics
  - Latch (D-type) inference
  - Flip-Flop (D-type) inference
  - Tri-state buffer inference
- Some simple rules are used in typical approaches

```
reg Q;
always@(D or en)
  if(en) Q = D;
```

Latch inferred!!

```
reg Q;
always@(posedge clk)
   Q = D;
```

Flip-flop inferred!!

```
reg Q;
always@(D or en)
  if(en) Q = D;
  else   Q = 1'bz;
```

Tri-state buffer inferred!!

Preliminaries

- Sequential section
  - Edge triggered always statement
- Combinational section
  - All signals whose values are used in the always statement are included in the sensitivity list

```
reg Q;
always@(posedge clk)
   Q = D;
```

Sequential section

Conduct flip-flop inference

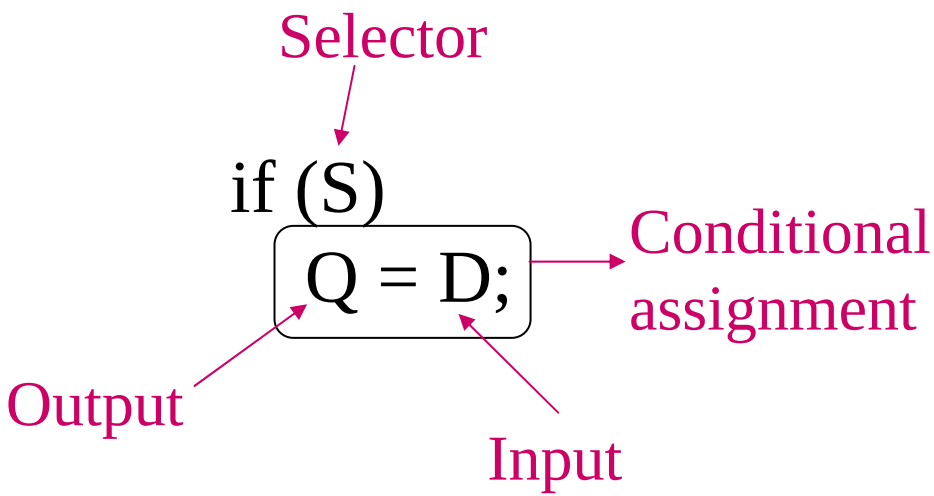
```
reg Q;
always@(in or en)
  if(en) Q=in;
```

Combinational section

Conduct latch inference

Terminology (1/2)

- Conditional assignment
- Selector: S
- Input: D
- Output: Q

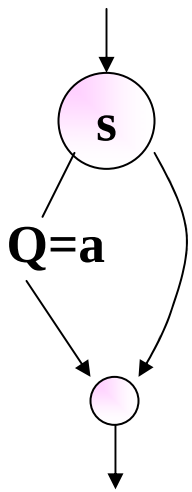


Terminology (2/2)

- A variable Q has a *branch* for a value of selector **s**
  - The variable Q is assigned a value in a path going through the branch

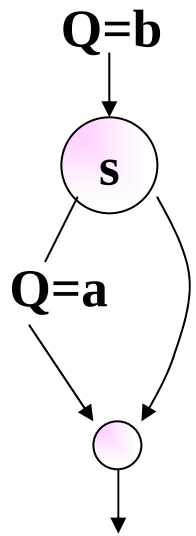
if(s) Q=a;

Q has no branch for the false value of the selector s



Q=b;
if(s) Q=a;

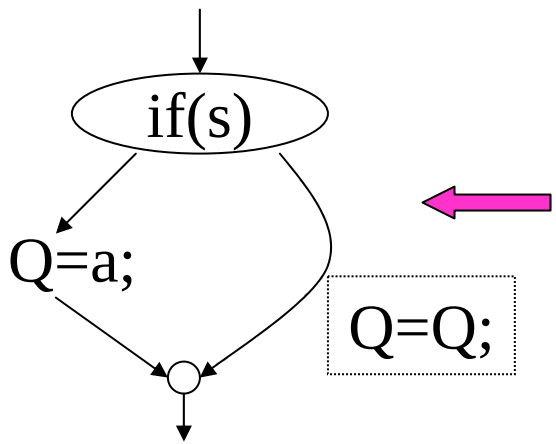
Q has a branch for the false value of the selector s



Rules of Latch Inference (1/2)

- Condition 1: There is no branch associated with the output of a conditional assignment for a value of the selector
  - Output depends on its previous value implicitly

```
always@(s or a)
if(s) Q=a;
```

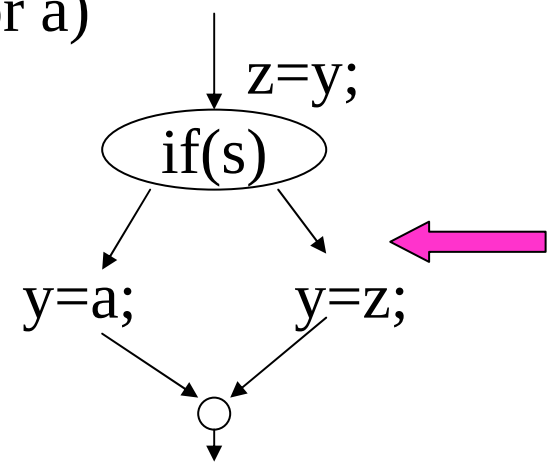


Q depends on its previous value at this branch

Rules of Latch Inference (2/2)

- Condition 2: The output value of a conditional assignment depends on its previous value explicitly

```
always@(s or z or y or a)
begin
  z = y;
  if(s) y=a;
  else y=z;
end
```



y depends on its previous value at this branch via the assignment z=y;

Typical Latch Inference

- Conditional assignments are not completely specified
  - Check if the *else-clause* exists
  - Check if all case items exist
- Outputs conditionally assigned in an if-statement are not assigned before entering or after leaving the if-statement

```
always@(D or S)
if(S) Q = D;
```

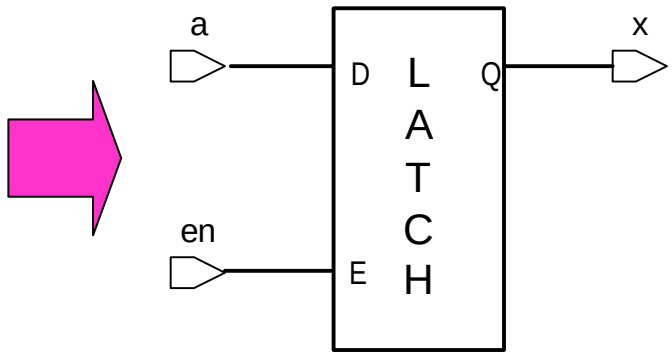
→ Infer latch for Q

```
always@(S or A or B)
begin
Q = A;
if(S) Q = B;
end
```

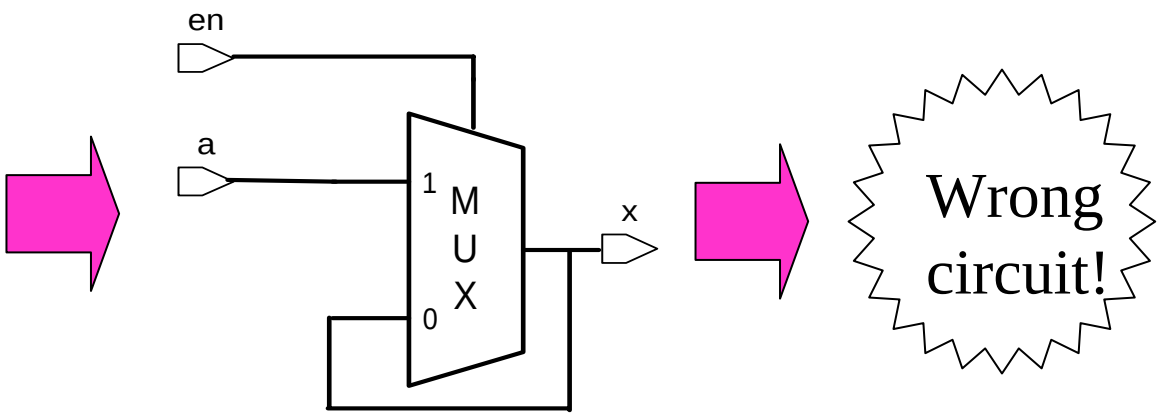
→ Do not infer latch for Q

Typical Coding Style Limitation (1/2)

```
always@(a or en)
  if(en) x=a;
```




```
always@(a or en)
  if(en) x=a;
  else x=x;
```



Latch description

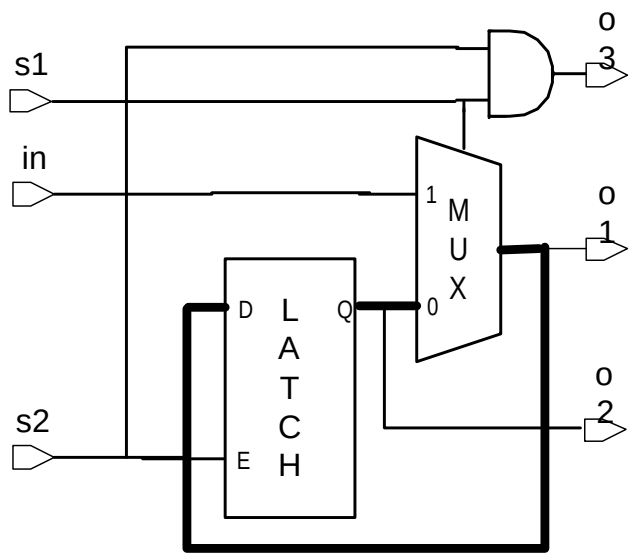
Typical Coding Style Limitation (2/2)

- Process by process
  - No consideration on the dependencies across processes
  - No warrantee on the consistency of memory semantics

```
module EXP(in, s1, s2, o1, o2, o3);
input in, s1, s2;
output o1,o2,o3;
reg o1, o2, o3;
always@(in or s1 or o2)
/*d1*/    if(s1) o1=in;
/*d2*/    else o1=o2;
always@( s1 or s2 or o1) begin
/*d3*/    o3=s1&s2;
/*d4*/    if(s2) o2=o1;
end
endmodule
```

o1 depends on its value via o2 at d4

Infer a latch for o2



Asynchronous feedback loop

Terminology

- Clocked statement: edge-triggered always statement
  - Simple clocked statement

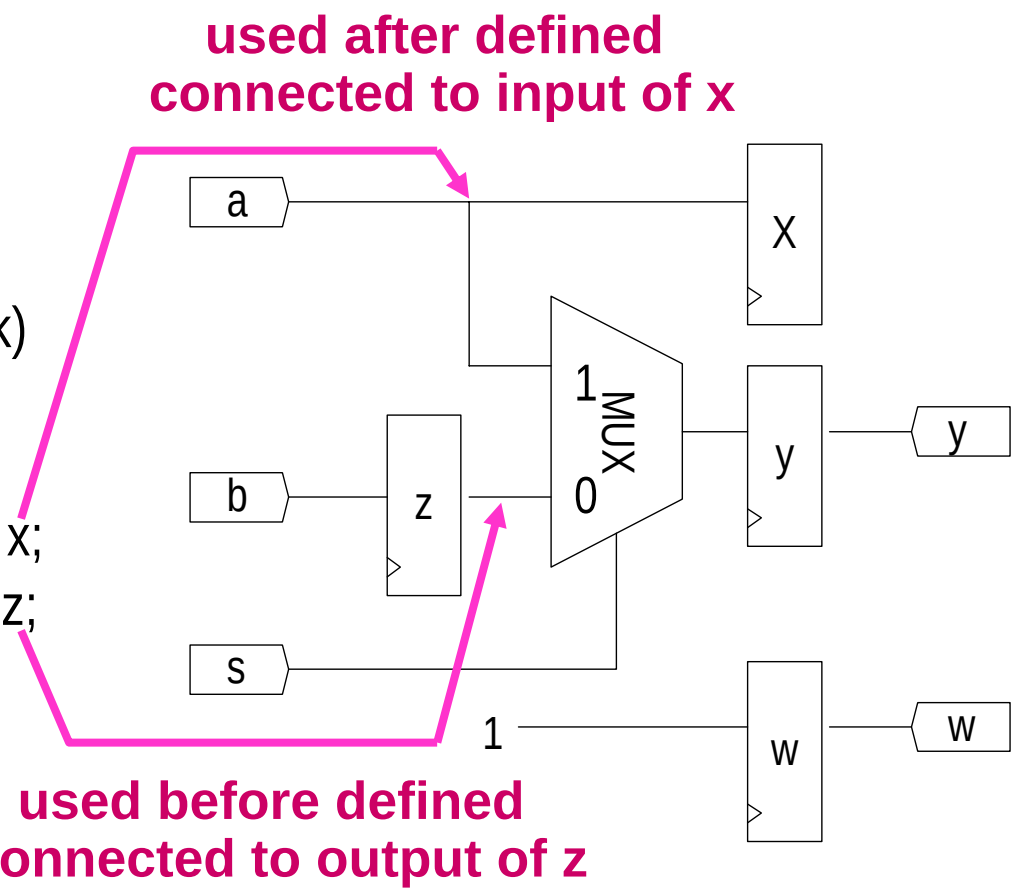
    e.g., always @ (**posedge** clock)
  - Complex clocked statement

    e.g., always @ (**posedge** clock or **posedge** reset)
- Flip-flop inference must be conducted only when synthesizing the **clocked statements**

Infer FF for Simple Clocked Statements (1/2)

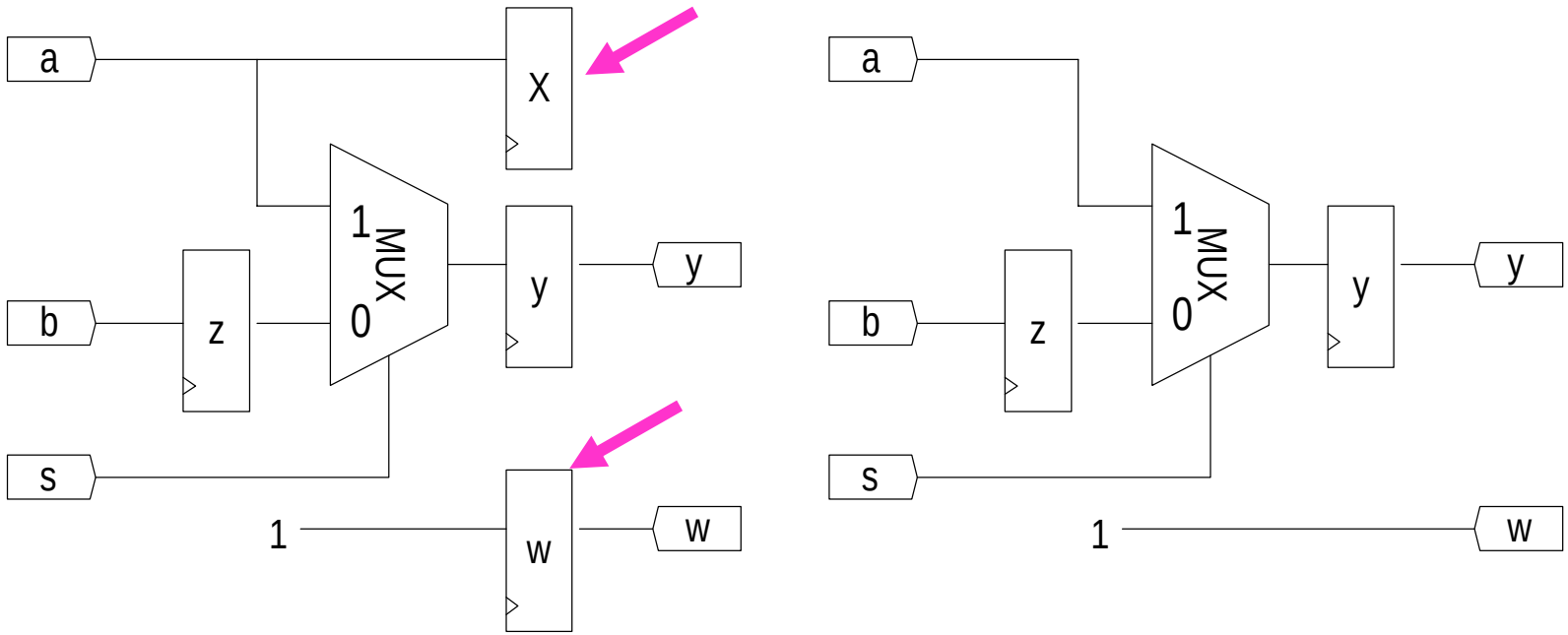
- Infer a flip-flop for **each variable** being assigned in the simple clocked statement

```
input a, b, s, clk;
output y, w;
reg x, w, y, z;
always @ (posedge clk)
begin
/* d1 */      x = a;
/* d2 */      if ( s )   y = x;
/* d3 */      else       y = z;
/* d4 */      z = b;
/* d5 */      w = 1'b1;
end
```



Infer FF for Simple Clocked Statements (2/2)

- Two post-processes
  - Propagating constants
  - Removing the flip-flops without fanouts



Infer FF for Complex Clocked Statements

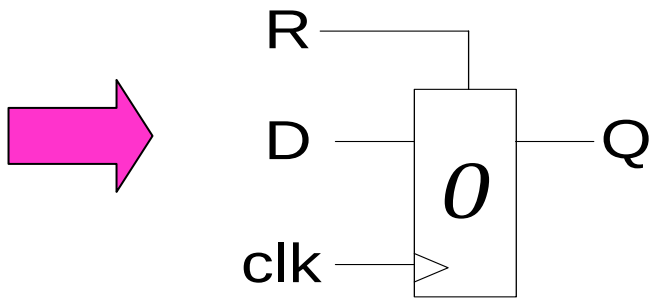
- The edge-triggered signal not used in the following operations is chosen as the clock signal
- The usage of asynchronous control pins requires the following syntactic template
  - An if-statement immediately follows the always statement
  - Each variable in the event list except the *clock signal* must be a selective signal of the if-statements
  - Assignments in the blocks B1 and B2 must be constant assignments (e.g., x=1, etc.)

always @ (**posedge** clock or **posedge** reset or **negedge** set)

```
if(reset) begin B1 end
else if ( !set) begin B2 end
else begin B3 end
```

Typical Coding Style Limitation

```
always @ (posedge clk or posedge R)
if(R) Q = 0;
else Q = D;
```

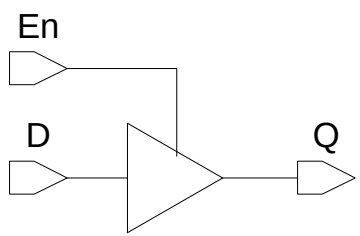



```
always @ (posedge clk or posedge R)
begin
Q = D;
if(R) Q = 0;
end
```

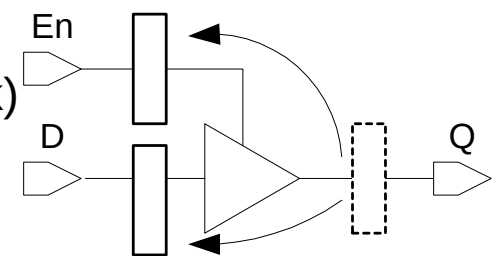
Typical Tri-State Buffer Inference (1/2)

- If a data object Q is assigned a high impedance value ‘Z’ in a multi-way branch statement (if, case, ?:)
  - Associated Q with a tri-state buffer
- If Q associated with a tri-state buffer has also a memory attribute (latch, flip-flop)
  - Have the **Hi-Z propagation problem**
    - Real hardware cannot propagate Hi-Z value
  - Require two memory elements for the control and the data inputs of tri-state buffer

```
reg Q;
always @ (En or D)
if(En) Q = D;
else Q = 1'bz;
```



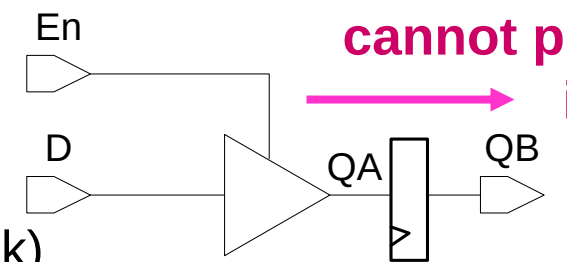

```
reg Q;
always @ (posedge clk)
if(En) Q = D;
else Q = 1'bz;
```



Typical Tri-State Buffer Inference (2/2)

- It may suffer from mismatches between synthesis and simulation
  - Process by process
  - May incur the Hi-Z propagation problem

```
reg QA, QB;
always @ (En or D)
if(En) QA = D;
else QA = 1'bz;

always @ (posedge clk)
QB = QA;
```



cannot propagate Hi-Z in real hardware

assignment can pass Hi-Z to QB in simulation

Comments on Special Element Inference

- Typical synthesizers
  - Use ad hoc methods to solve latch inference, flip-flop inference and tri-state buffer inference
  - Incur extra limitations on coding style
  - Do not consider the dependencies across processes
  - Suffer from synthesis/simulation mismatches
- A lot of efforts can be done to enhance the synthesis capabilities
  - It may require more computation time
  - Users' acceptance is another problem

Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level logic optimization
  - Multi-level logic optimization
- Technology mapping
- Timing optimization
- Synthesis for low power
- Retiming

Two-Level Logic Optimization

Basic idea: Boolean law $x+x'=1$ allows for grouping $x1x2+x1\,x'2=x1$

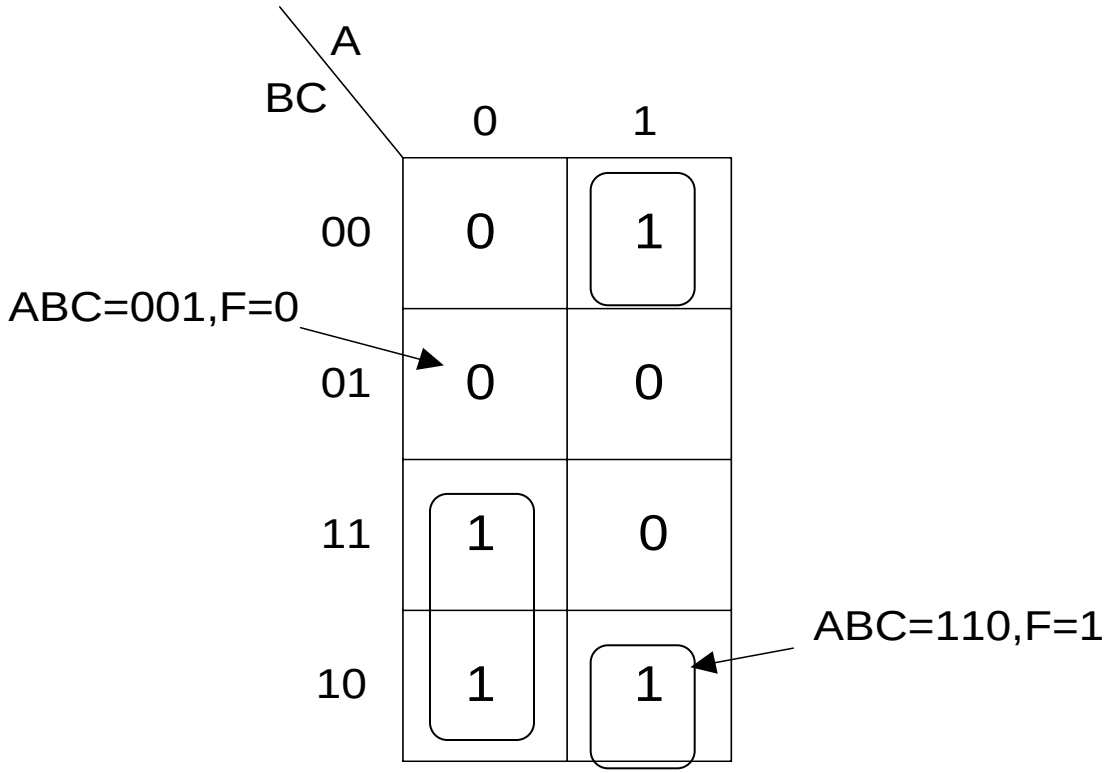
Approaches to simplify logic functions:

- Karnaugh maps [Kar53]
- Quine-McCluskey [McC56]

3-Variable Karnaugh Maps

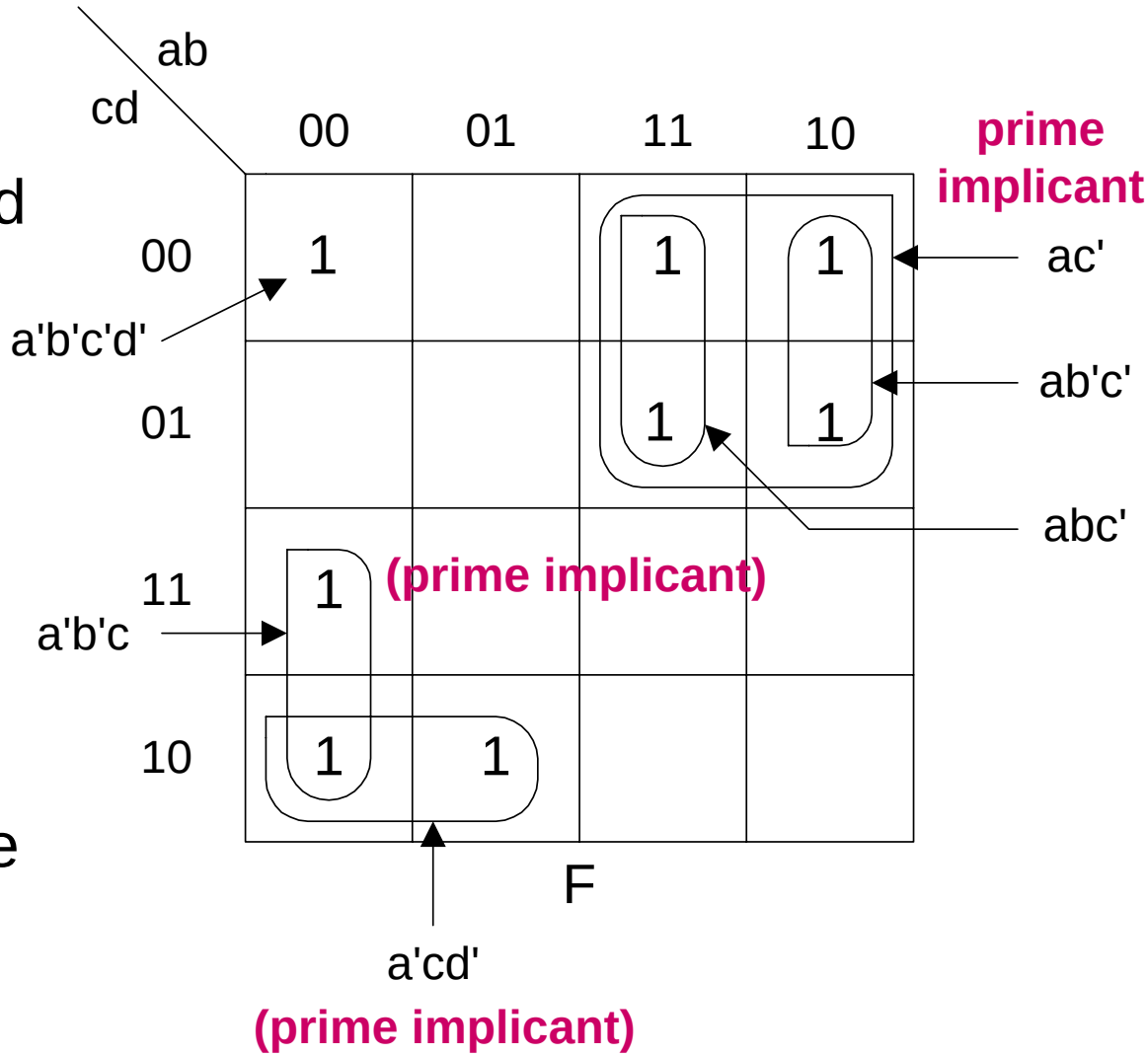
- Example

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



Implicant

- **IMPLICANT**: any single 1 or any group of 1's combined together on a map of the function F
  - ab'c', abc
- **PRIME IMPLICANT**: an implicant that cannot be combined with another terms to eliminate a variable
  - a'b'c, a'cd, ac'

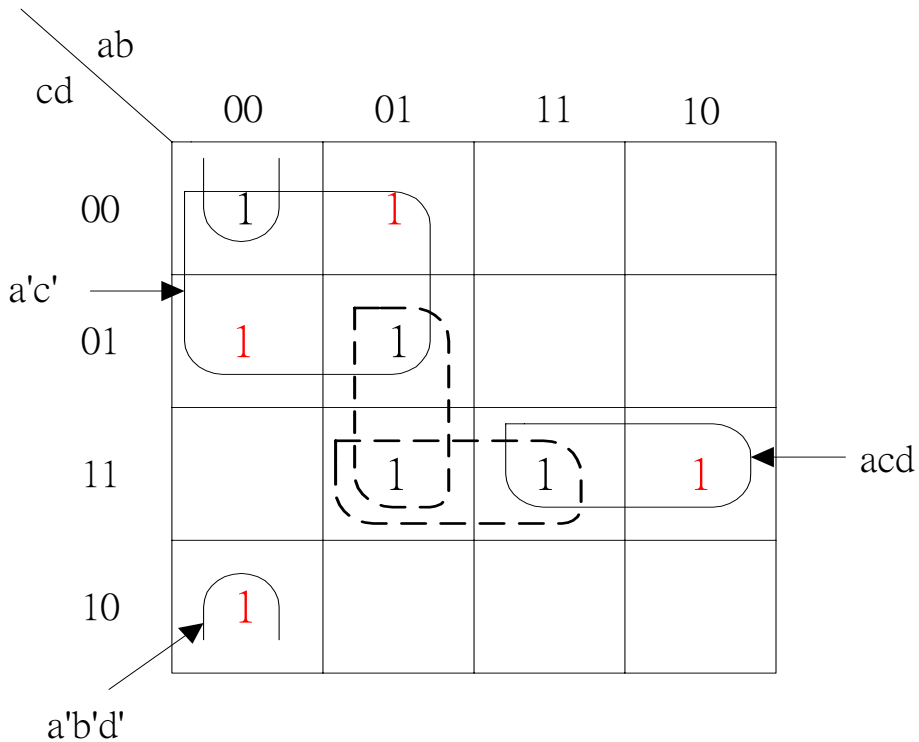


Minimum Form

- A sum-of-products expression containing a non-prime implicant cannot be minimum
  - Could be simplified by combining the nonprime term with additional minterm
- To find the minimum sum-of-products
  - Find a minimum number of prime implicants which cover all of the 1's
  - Not every prime implicant is needed
  - If prime implicants are selected in the wrong order, a nonminimum solution may result

Essential Prime Implicant

- If a minterm is covered by **only one** prime implicant, that prime implicant is **ESSENTIAL** and must be included in the minimum sum-of-products

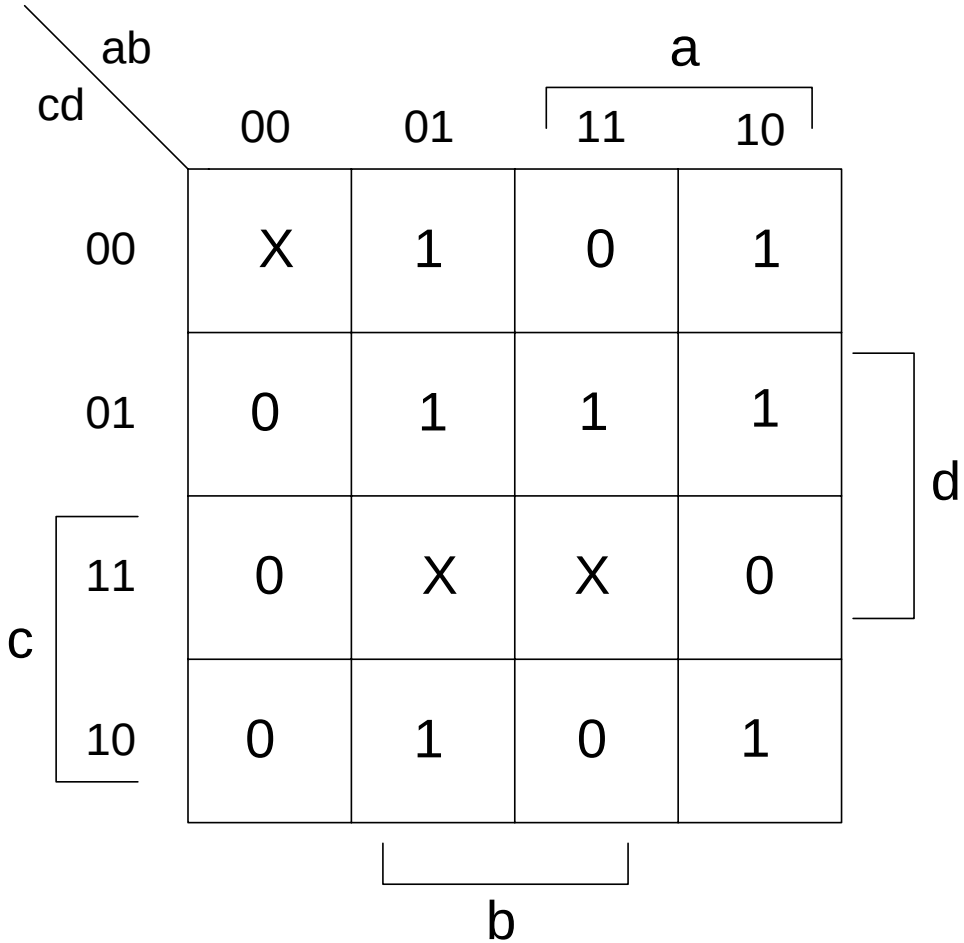


Note: 1's in red color are covered by only one prime implicant. All other 1's are covered by at least two prime implicants

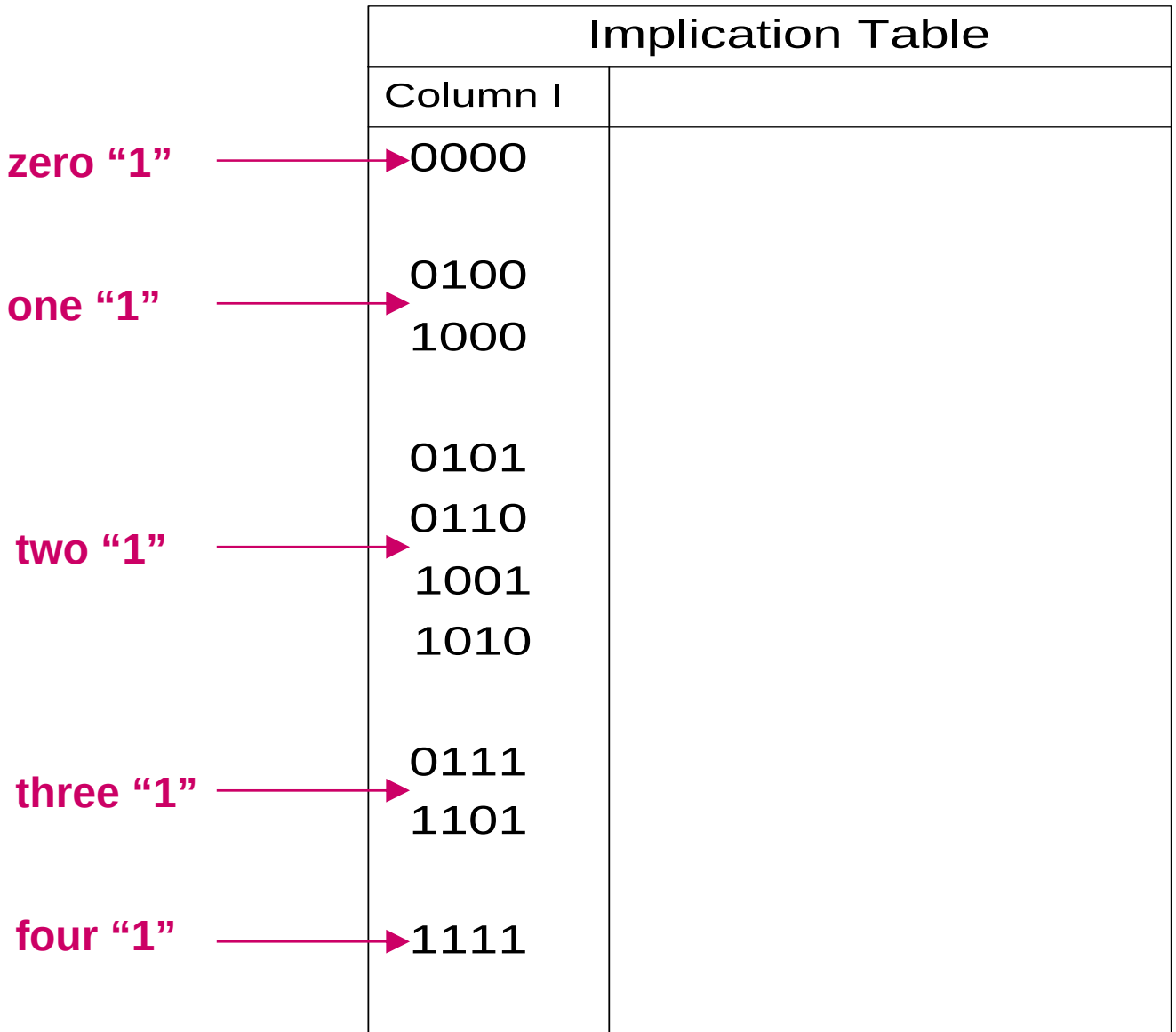
Classical Logic Minimization

- Theorem:[Quine,McCluskey] There exists a minimum cover for F that is prime
  - Need to look just at primes (reduces the search space)
- Classical methods: two-step process
  1. Generation of all prime implicants
  2. Extraction of a minimum cover (covering problem)

Primary Implicant Generation (1/5)



Primary Implicant Generation (2/5)



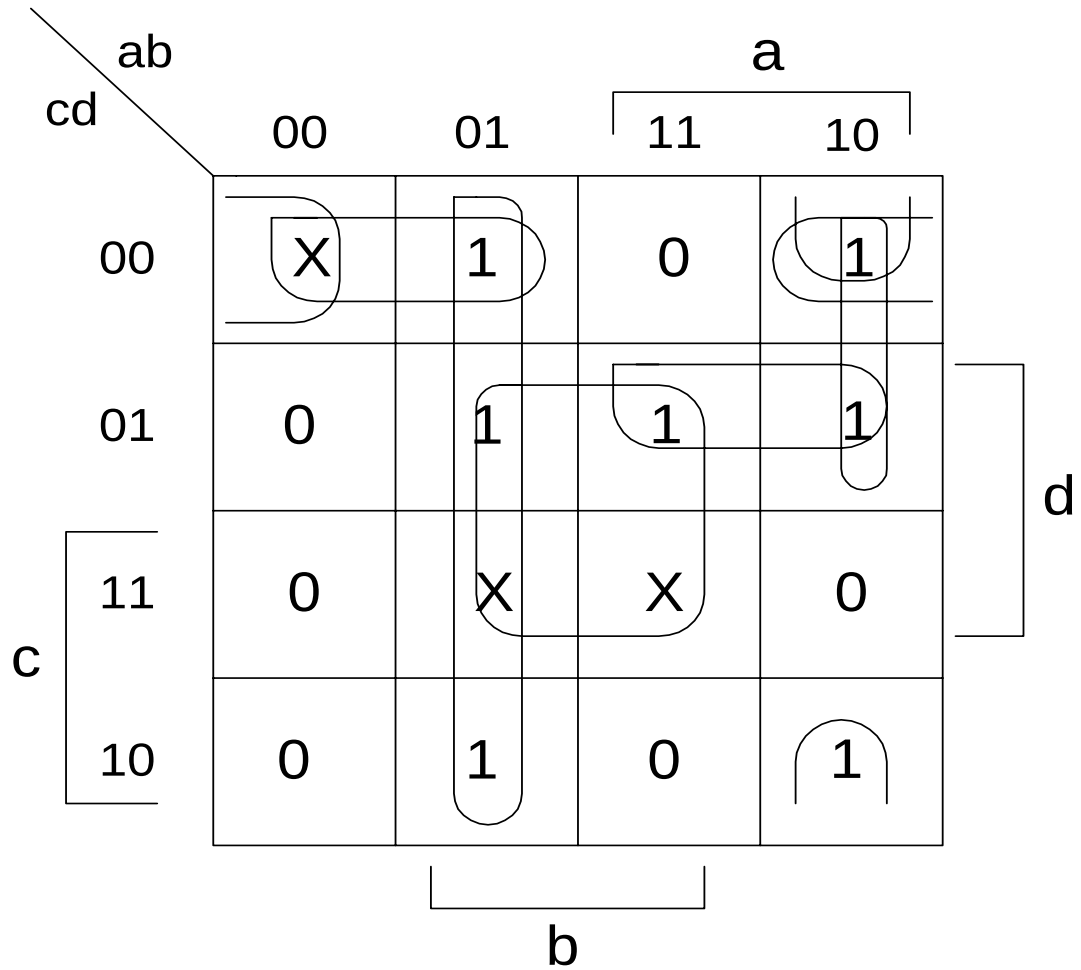
Primary Implicant Generation (3/5)

Implication Table		
Column I	Column II	
0000 |	0-00	
	-000	
0100 |		
1000 |	010-	
	01-0	
0101 |	100-	
0110 |	10-0	
1001 |		
1010 |	01-1	
	-101	
0111 |	011-	
1101 |	1-01	
1111 |	-111	
	11-1	

Primary Implicant Generation (4/5)

Implication Table		
Column I	Column II	Column III
0000 |	0-00 *	01-- *
	-000 *	
0100 |		-1-1 *
1000 |	010- |	
	01-0 |	
0101 |	100- *	
0110 |	10-0 *	
1001 |		
1010 |	01-1 |	
	-101 |	
0111 |	011- |	
1101 |	1-01 *	
1111 |	-111 |	
	11-1 |	

Primary Implicant Generation (5/5)



Prime Implicants:

0-00 = a'c'd'

100- = ab'c'

1-01 = ac'd

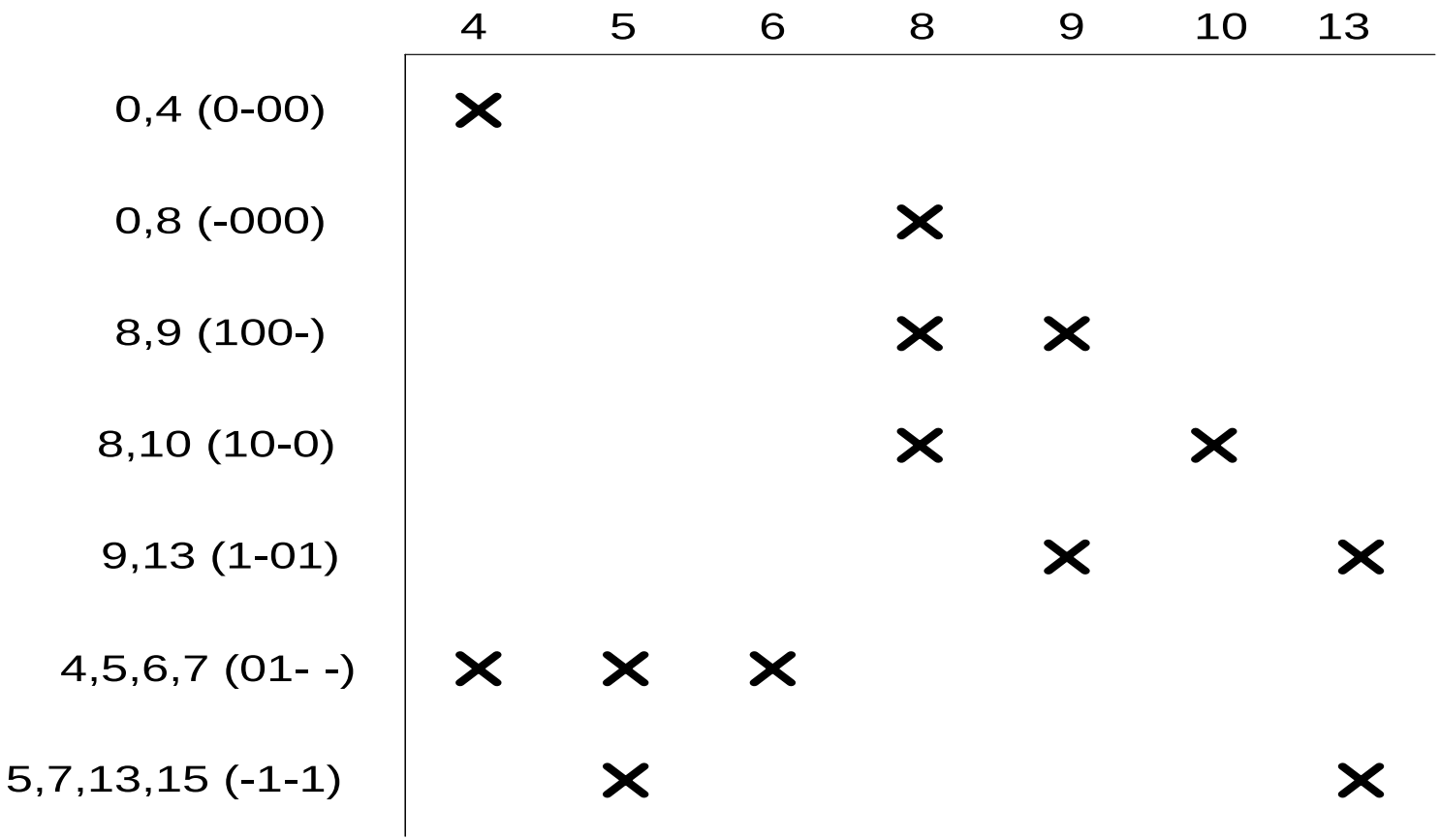
-1-1 = bd

-000 = b'c'd'

10-0 = ab'd'

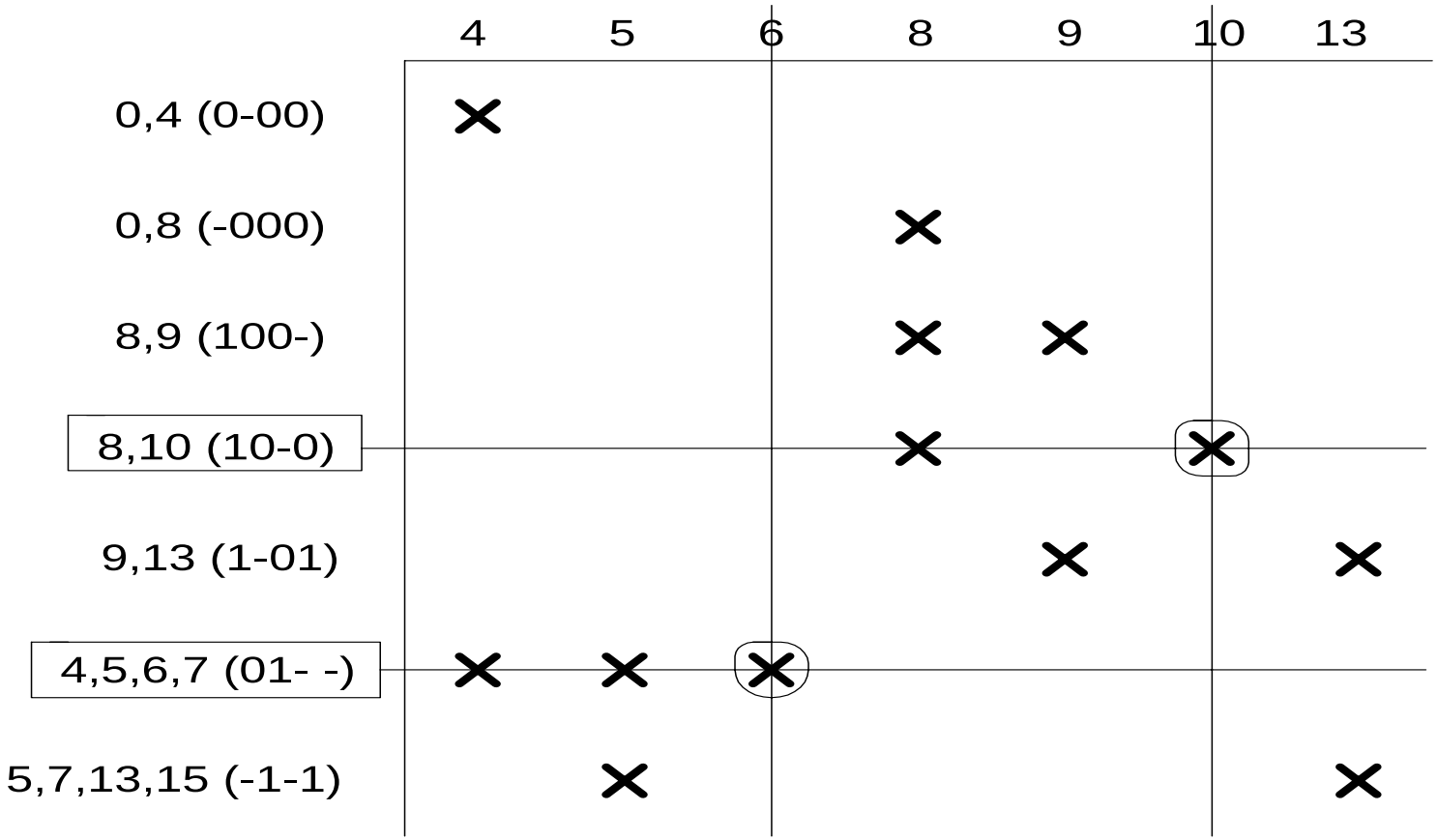
01-- = a'b

Column Covering (1/4)



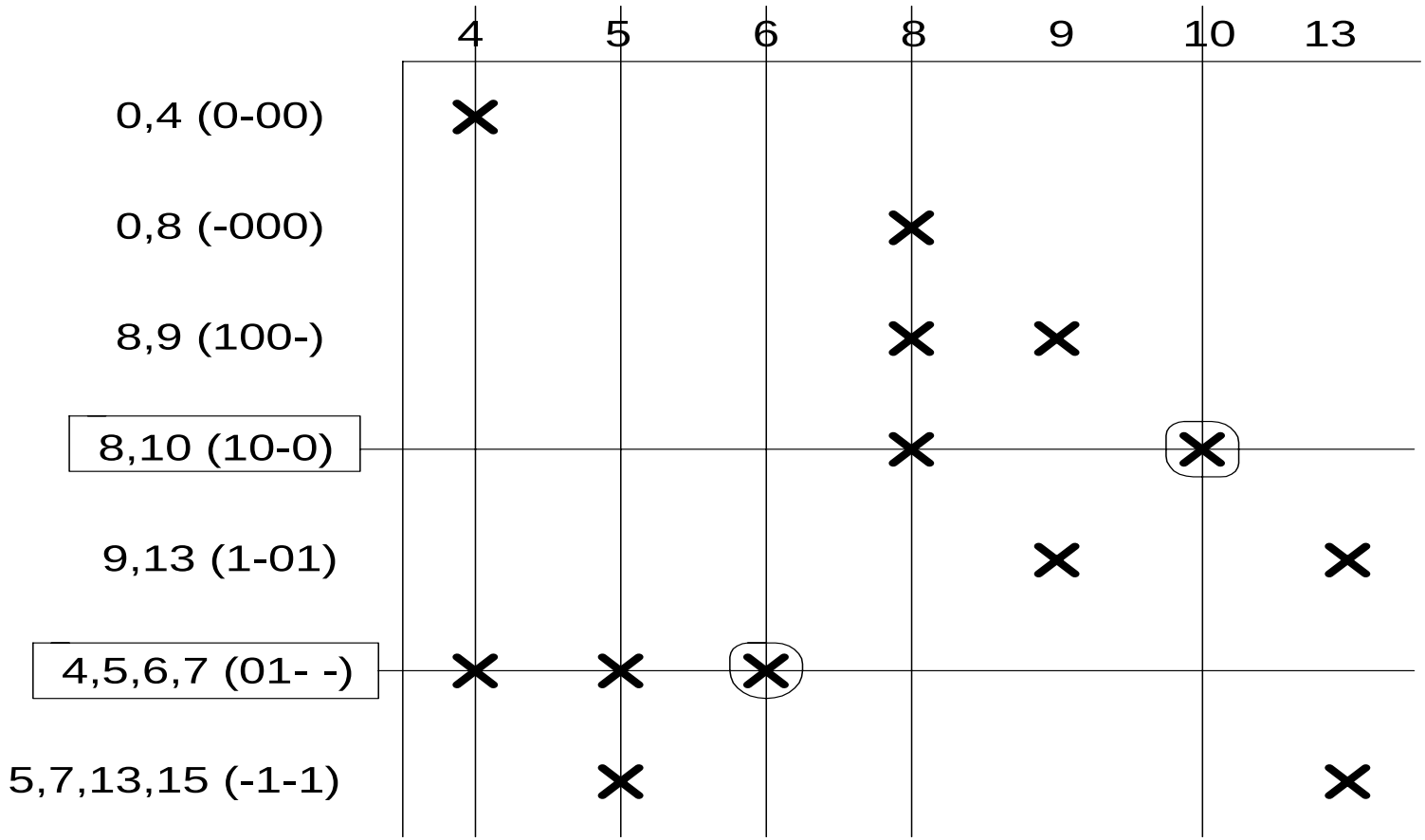
rows = prime implicants
columns = ON-set elements
place an "X" if ON-set element
is covered by the prime implicant

Column Covering (2/4)



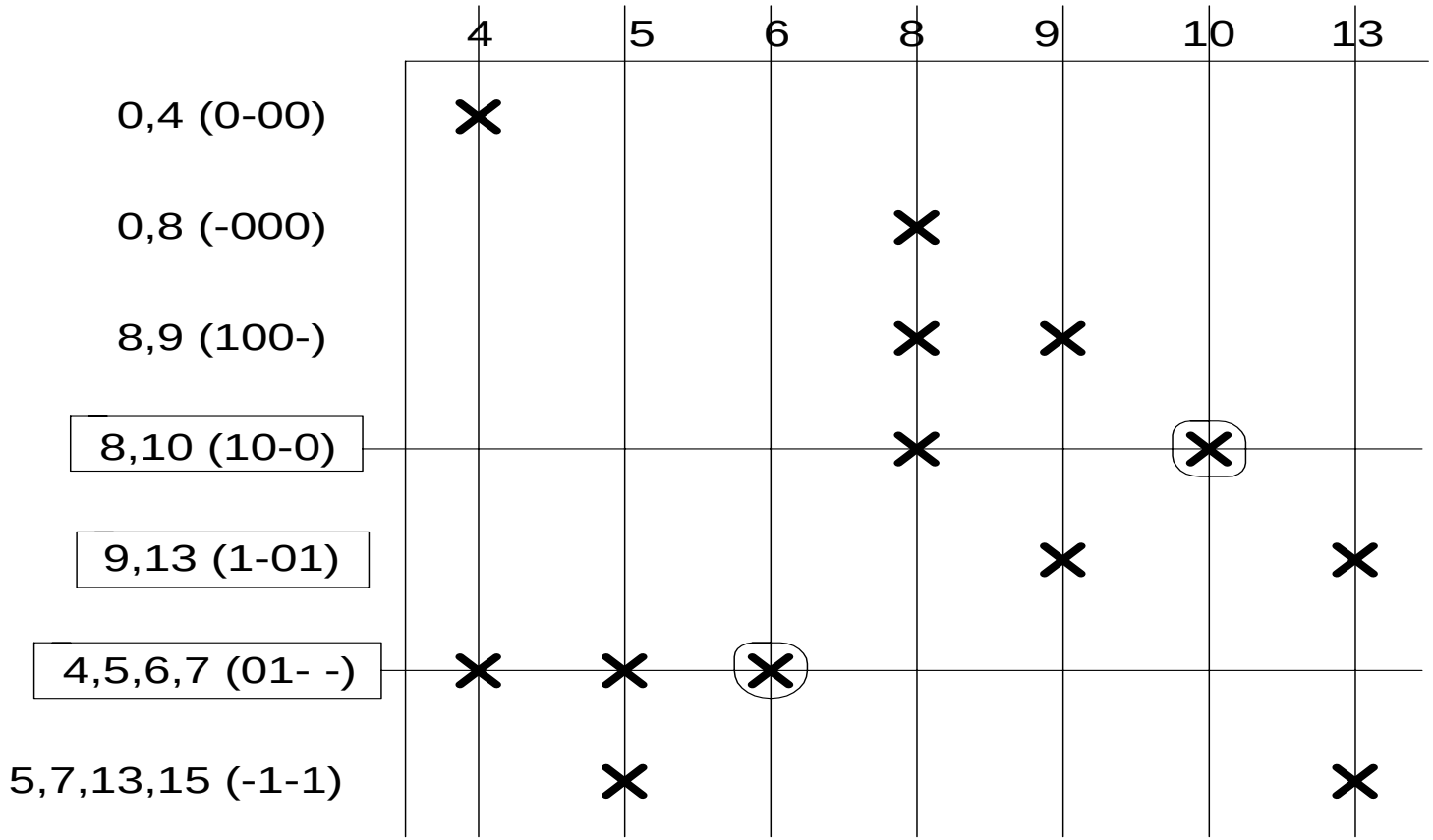
If column has a single X, then the implicant associated with the row is essential. It must appear in minimum cover

Column Covering (3/4)



Eliminate all columns covered by essential primes

Column Covering (4/4)



Find minimum set of rows that cover the remaining columns

f = ab'd' + ac'd + a'b

Petrick's Method

- Solve the satisfiability problem of the following function

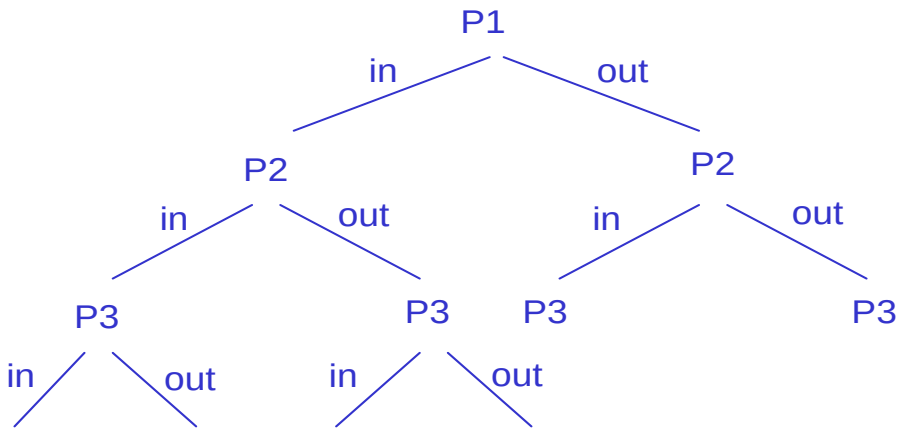
P = (P1+P6)(P6+P7)P6(P2+P3+P4)(P3+P5)P4(P5+P7)=1

		4	5	6	8	9	10	13
P1	0,4 (0-00)	×						
P2	0,8 (-000)				×			
P3	8,9 (100-)				×	×		
P4	8,10 (10-0)				×		×	
P5	9,13 (1-01)					×		×
P6	4,5,6,7 (01- -)	×	×	×				
P7	5,7,13,15 (-1-1)		×					×

- Each term represents a corresponding column
- Each column must be chosen at least once
- All columns must be covered

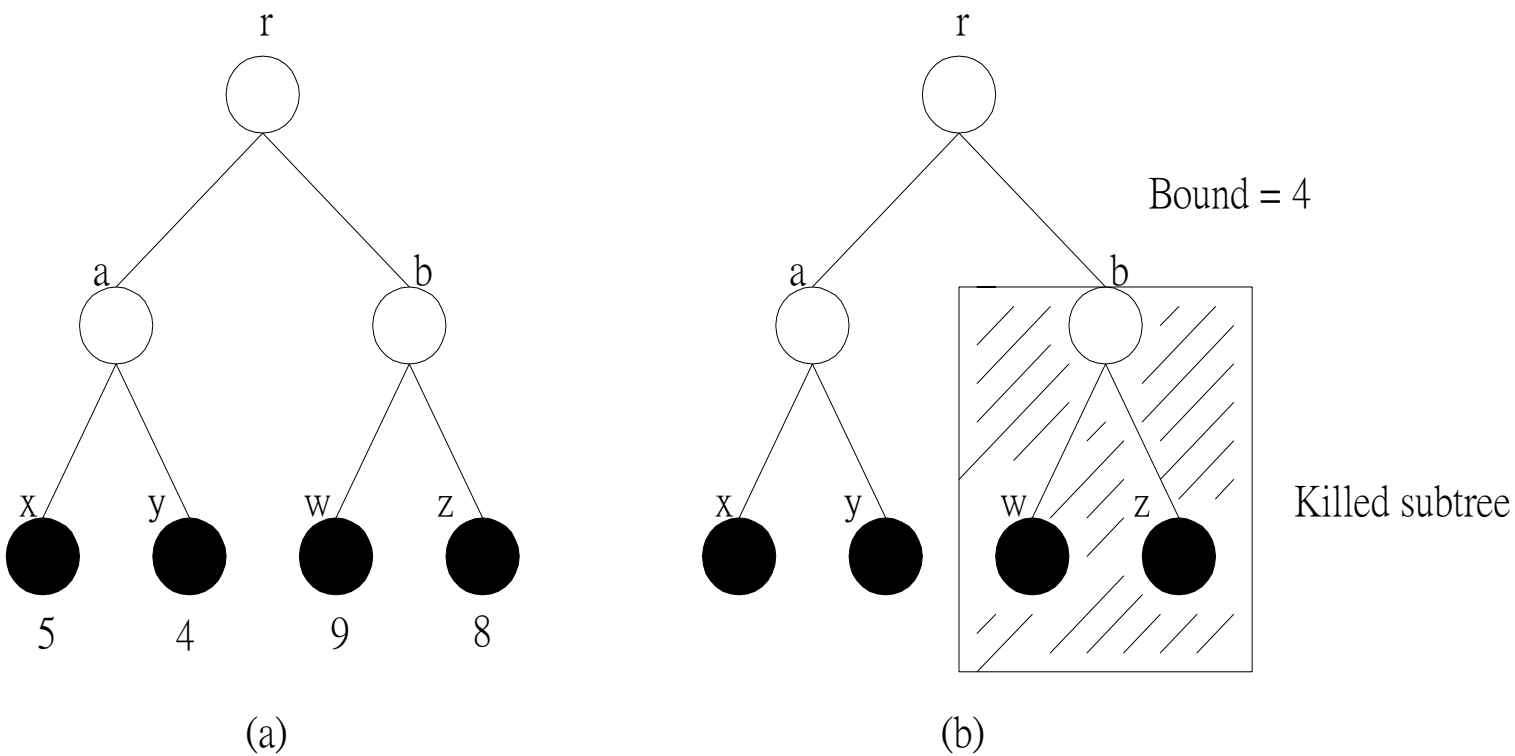
Brute Force Technique

- Brute force technique: Consider all possible elements



- Complete branching tree has $2^{|P|}$ leaves!!
  - Need to prune it
- Complexity reduction
  - Essential primes can be included right away
    - If there is a row with a singleton “1” for the column
  - Keep track of best solution seen so far
    - Classic ***branch and bound***

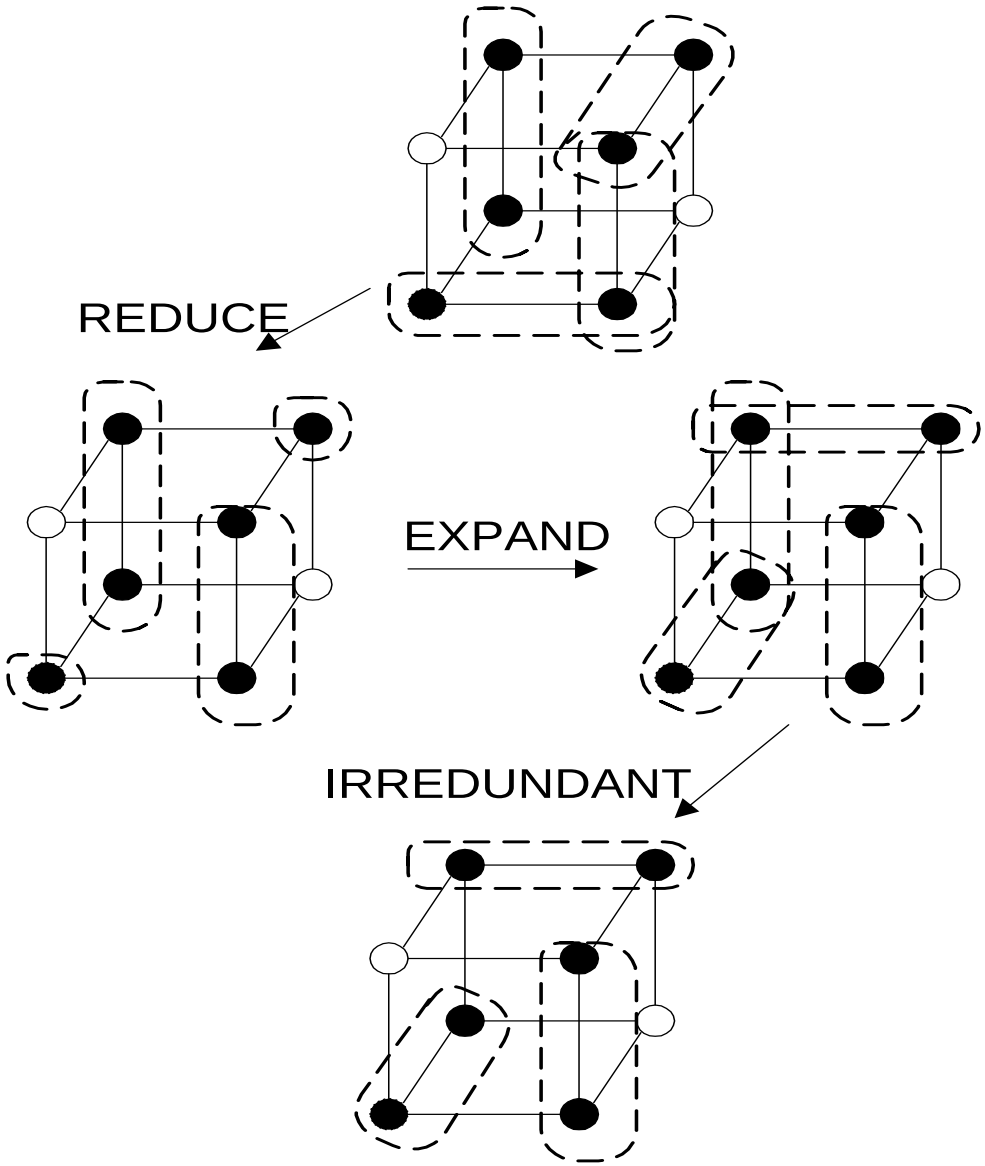
Branch and Bound Algorithm



Heuristic Optimization

- Generation of ***all*** prime implicants is impractical
  - The number of prime implicants for functions with *n* variables is in the order of $3^n/n$
- Finding an *exact* minimum cover is NP-hard
  - Cannot be finished in polynomial time
- Heuristic method: avoid generation of all prime implicants
- Procedure
  - A minterm of ON(f) is selected, and expanded until it becomes a prime implicant
  - The prime implicant is put in the final cover, and all minterms covered by this prime implicant are removed
  - Iterated until all minterms of the ON(f) are covered
- "ESPRESSO" developed by UC Berkeley
  - The kernel of synthesis tools

ESPRESSO - Illustrated

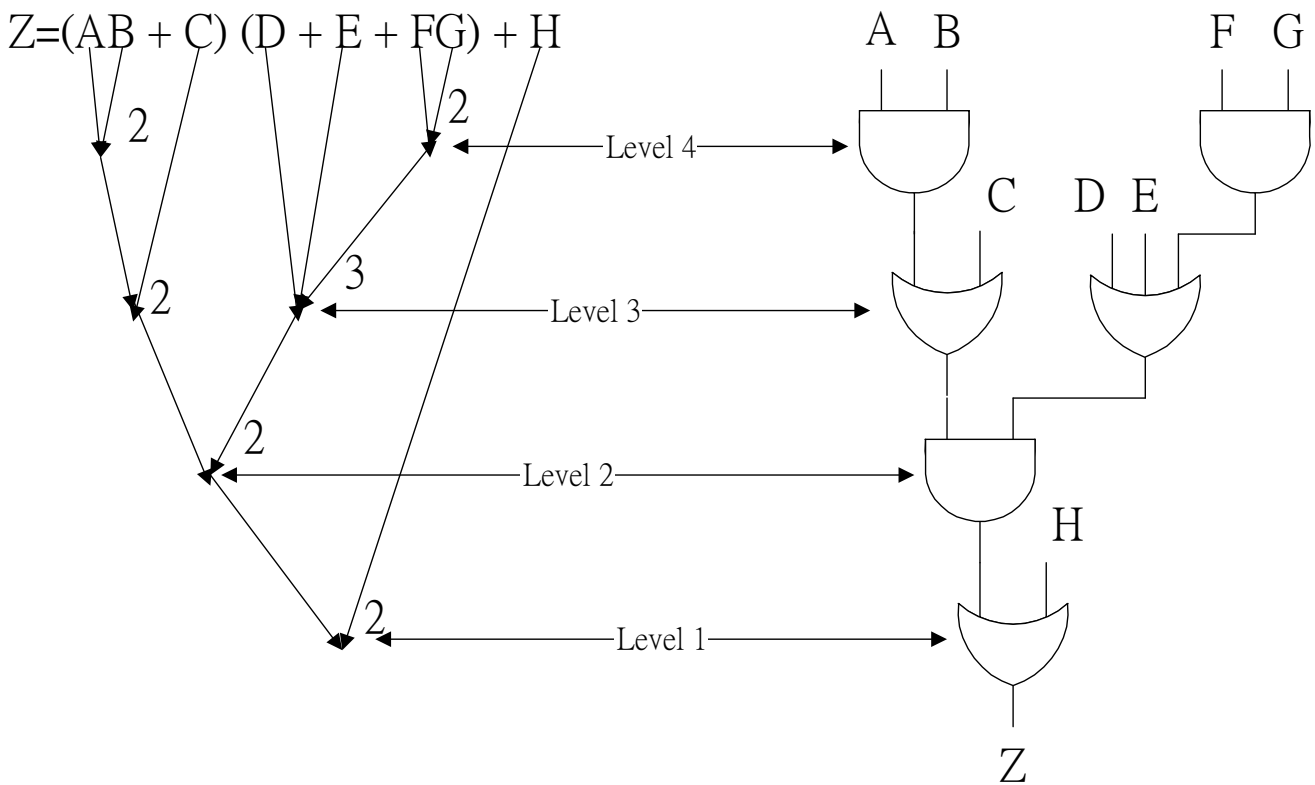


Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level logic optimization
  - Multi-level logic optimization
- Technology mapping
- Timing optimization
- Synthesis for low power
- Retiming

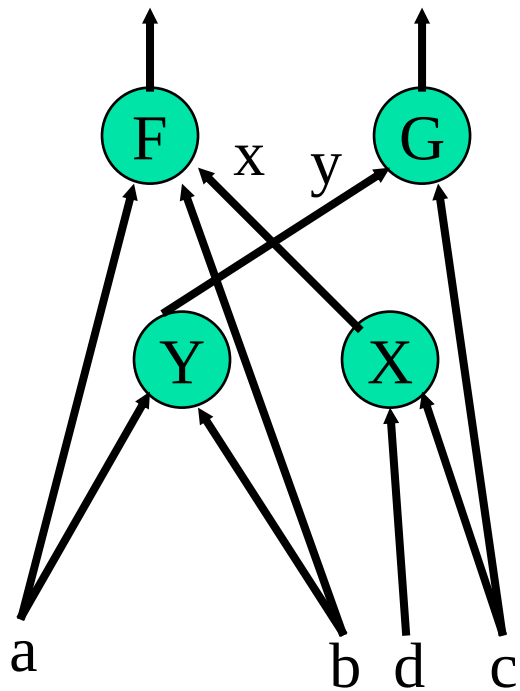
Multi-Level Logic

- Multi-level logic:
  - A set of logic equations with no cyclic dependencies
- Example: Z = (AB + C)(D + E + FG) + H
  - 4-level, 6 gates, 13 gate inputs



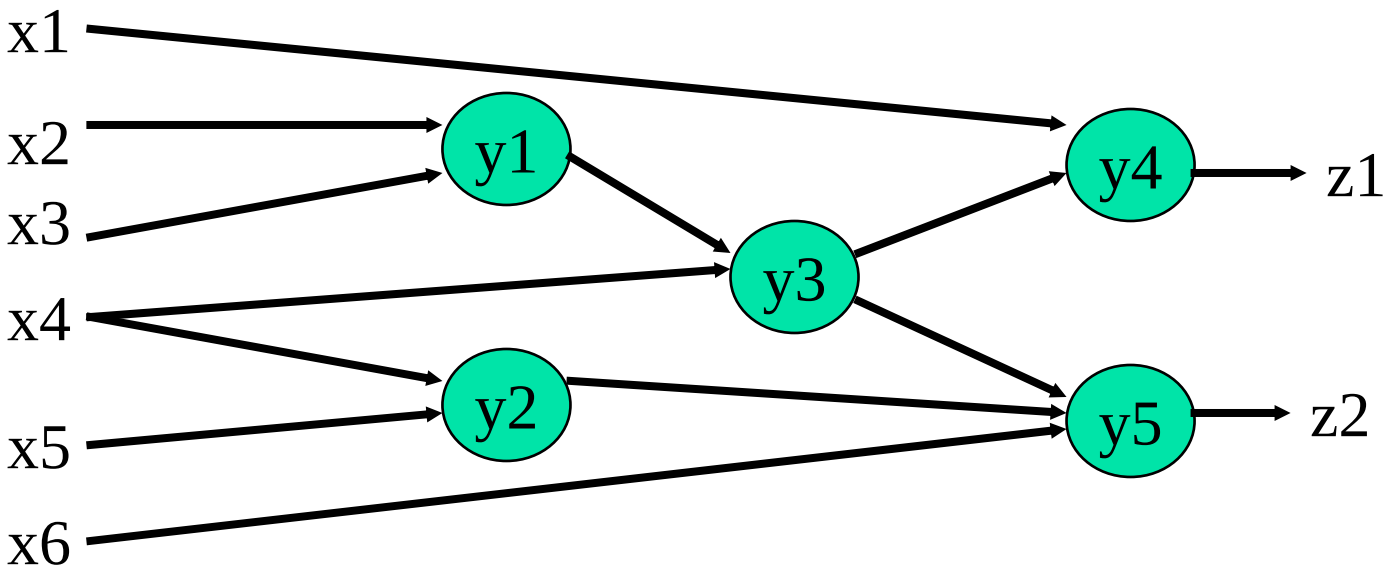
Boolean Network

- Directed acyclic graph (DAG)
- Each source node is a primary input
- Each sink node is a primary output
- Each internal node represents an equation
- Arcs represent variable dependencies



fanin of y : a, b
fanout of x : F

Boolean Network : An Example



$y1 = f_1(x2, x3) = x2' + x3'$

$y2 = f_2(x4, x5) = x4' + x5'$

$y3 = f_3(x4, y1) = x4'y1'$

$y4 = f_4(x1, y3) = x1 + y3'$

$y5 = f_5(x6, y2, y3) = x6y2 + x6'y3'$

Multi-Level v.s. Two-Level

- Two-level:
  - Often used in control logic design

    $f_1 = x_1x_2 + x_1x_3 + x_1x_4$

    $f_2 = x_1'x_2 + x_1'x_3 + x_1x_4$
  - Only x_1x_4 shared
  - Sharing restricted to common cube
- Multi-level:
  - Datapath or control logic design
  - Can share $x_2 + x_3$ between the two expressions
  - Can use complex gates

    $g_1 = x_2 + x_3$

    $g_2 = x_2x_4$

    $f_1 = x_1y_1 + y_2$

    $f_2 = x_1'y_1 + y_2$

    (y_i is the output of gate g_i)

Multi-Level Logic Optimization

- Technology independent
- Decomposition/Restructuring
  - Algebraic
  - Functional
- Node optimization
  - Two-level logic optimization techniques are used

Decomposition / Restructuring

- Goal : given initial network, find best network
- Two problems:
  - Find good **common subfunctions**
  - How to perform **division**
- Example:

  $f_1 = abcd + abce + ab'cd' + ab'c'd' + a'c + cdf + abc'd'e' + ab'c'df'$

  $f_2 = bdg + b'dfg + b'd'g + bd'eg$

  minimize (in sum-of-products form):

  $f_1 = bcd + bce + b'd' + b'f + a'c + abc'd'e' + ab'c'df'$

  $f_2 = bdg + dfg + b'd'g + d'eg$

  decompose:

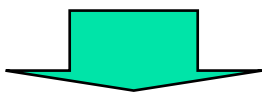
  $f_1 = c(a' + x) + ac'x'$ $x = d(b + d) + d'(b' + e)$

  $f_2 = gx$

Basic Operations (1/2)

1. decomposition
(single function)

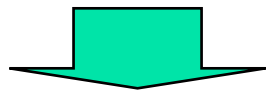
f = abc + abd + (ac)'d' + b'c'd'



f = xy + (xy)'
x = ab
y = c + d

2. extraction
(multiple functions)

f = (az + bz')cd + e
g = (az + bz')e'
h = cde

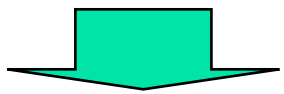


f = xy + e
g = xe'
h = ye
x = az + bz'
y = cd

Basic Operations (2/2)

3. factoring
(series-parallel decomposition)

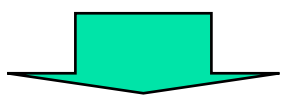
$f = ac + ad + bc + bd + e$



$f = (a + b)(c + d) + e$

4. substitution
(with complement)

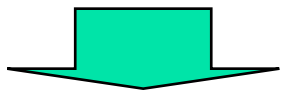
$g = a + b$
$f = a + bc + b'c'$



$f = g(a + c) + g'c'$

5. elimination

$f = ga + g'b$
$g = c + d$



$f = ac + ad + bc'd'$
$g = c + d$

Division

- Division: p is a Boolean divisor of f if $q \neq \phi$ and r exist such that $f = pq + r$
  - p is said to be a factor of f if in addition $r = \phi$:

    $$f = pq$$

  - q is called the **quotient**
  - r is called the **remainder**
  - q and r are **not unique**
- ***Weak division***: the unique algebraic division such that r has as few cubes as possible
  - The quotient q resulting from weak division is denoted by f / p (it is ***unique***)

Weak Division Algorithm (1/2)

Weak_div(f, p):

U = Set $\{u_j\}$ of cubes in f with literals not in p deleted

V = Set $\{v_j\}$ of cubes in f with literals in p deleted

/* note that $u_j v_j$ is the j-th cube of f */

$V^i = \{v_j \in V : u_j = p_i\}$

$q = \cap V^i$

$r = f - pq$

return(q, r)

Weak Division Algorithm (2/2)

- Example

common expressions

$f = acg + adg + ae + bc + bd + be + a'b$

$p = ag + b$

$U = ag + ag + a + b + b + b + b$

$V = c + d + e + c + d + e + a'$

$V^{ag} = c + d$

$V^{b} = c + d + e + a'$

$q = c + d = f/p$

Algebraic Divisor

- Example:

  X = (a + b + c)de + f

  Y = (b + c + d)g + aef

  Z = aeg + bc

- Single-cube divisor: ae
- Multiple-cube divisor: b + c
- Extraction of common sub-expression is a global area optimization effort

Some Definitions about Kernels

- Definition: An expression is ***cube-free*** if no cube divides the expression evenly
  - *ab* + *c* is cube-free
  - *ab* + *ac* = *a* (*b* + *c*) is not cube-free
- Note: a cube-free expression must have more than one cube
  - *abc* is not cube-free
- Definition: The ***primary divisors*** of an expression f are the set of expressions

  $$D(f) = \{f/c \mid c \text{ is a cube}\}$$

  - To find cube-free divisor

Kernels

- Definition: The ***kernels*** of an expression *f* are the set of expressions

  $K(f) = \{g \mid g \in D(f)$ and g is cube free$\}$

- The kernels of an expression f are K(f) = {f/c}, where
  - / denote algebraic polynomial division
  - c is a cube
  - No cube divide f/c evenly (without any remainder)
- The cube c used to obtain the kernel is the ***co-kernel*** for that kernel

Co-Kernels

- Definition: A cube *c* used to obtain the kernel *k* = *f/c* is called a ***co-kernel*** of *k*. *C(f)* is used to denote the set of co-kernels of *f*.
- Example

  x = adf + aef + bdf + bef + cdf + cef + g

  = (a + b + c)(d + e)f + g

Kernel	***Co-kernel***
a + b + c	df, ef
d + e	af, bf, cf
(a + b + c)(d + e)f + g	1

Kernels of Expressions

- Example:

$$f = x_1x_2x_3 + x_1x_2x_4 + x_3'x_2$$

$$K = \{x_1x_3 + x_1x_4 + x_3', x_3 + x_4\}$$

  - x_1x_2 is the co-kernel for the kernel $x_3 + x_4$
- Kernels can be used to factor an expression

$$f = x_2(x_1(x_3 + x_4) + x_3')$$

- Key in finding ***common divisors*** between expressions

Common Divisor

- Theorem (Brayton & McMullen):

  f and *g* have a multiple-cube common divisor if and only if the intersection of a kernel of *f* and a kernel of *g* has more than one cube

$f_1 = x_1(x_2x_3 + x_2'x_4) + x_5$

$f_2 = x_1(x_2x_3 + x_2'x_5) + x_4$

$K(f_1) = \{x_2x_3 + x_2'x_4,$

$\quad x_1(x_2x_3 + x_2'x_4) + x_5\}$

$K(f_2) = \{x_2x_3 + x_2'x_5,$

$\quad x_1(x_2x_3 + x_2'x_5) + x_4\}$

$K_1 \cap K_2 = \{x_2x_3, x_1x_2x_3\}$

– f_1 and f_2 have no multiple-cube common divisor

$f_1 = x_1x_2 + x_3x_4 + x_5$

$f_2 = x_1x_2 + x_3'x_4 + x_5$

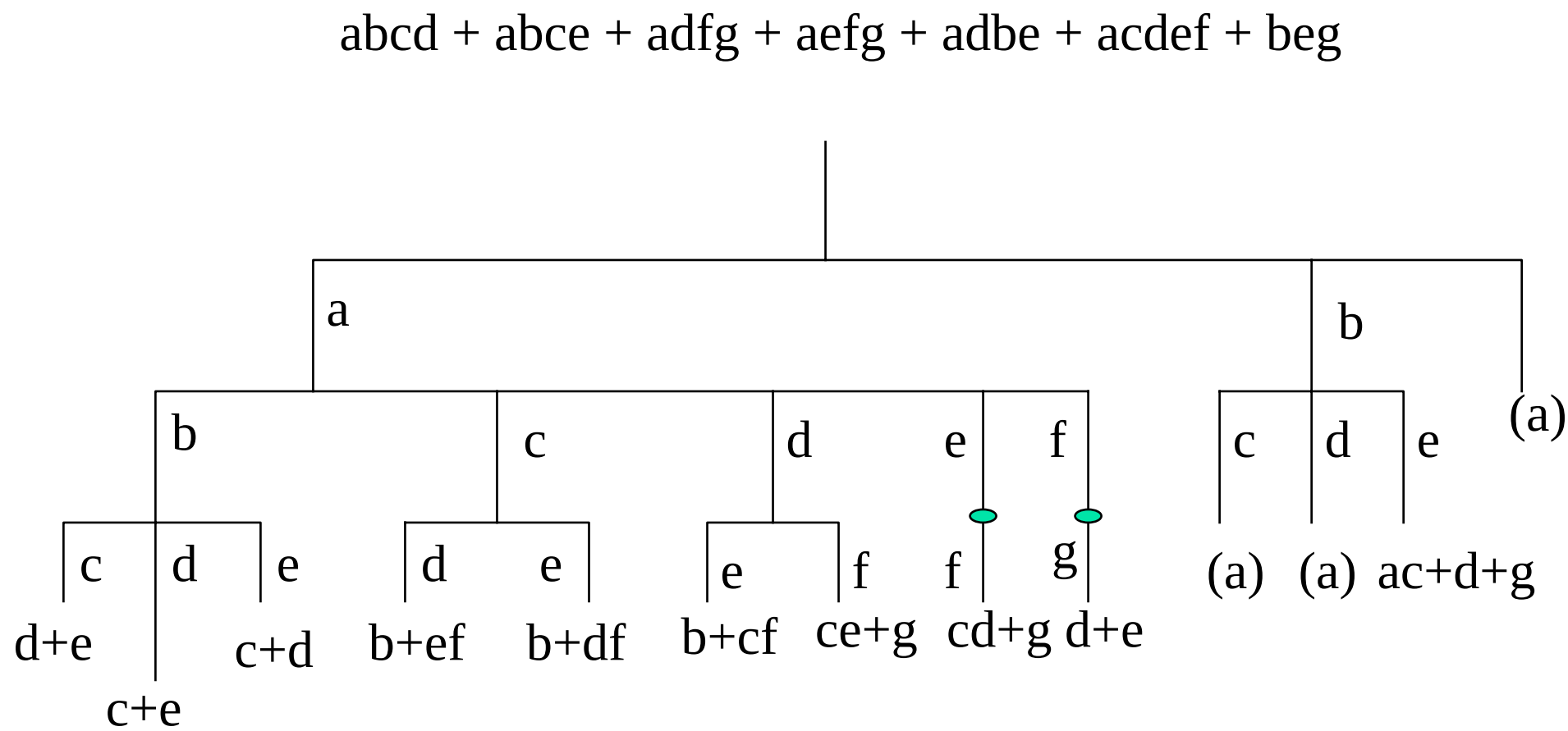
$K(f_1) = \{ x_1x_2 + x_3x_4 + x_5\}$

$K(f_2) = \{ x_1x_2 + x_3'x_4 + x_5\}$

$K_1 \cap K_2 = \{ x_1x_2 + x_5\}$

– f_1 and f_2 have multiple-cube common divisor

Find Out All Kernels (1/2)



Find Out All Kernels (2/2)

co-kernel	*kernel*
1	a((bc + fg)(d + e) + de(b + cf))) + beg
a	(bc + fg)(d + e) + de(b + cf)
ab	c(d + e) + de
abc	d + e
.	.
ac	b(d + e) + def
acd	b + ef
.	.
bc	ad + ae

They can be obtained in n^2 time
where n is number of cubes in this expression.

Cube-Literal Matrix

- Cube-literal matrix

$f = x_1x_2x_3x_4x_7 + x_1x_2x_3x_4x_8 + x_1x_2x_3x_5 + x_1x_2x_3x_6 + x_1x_2x_9$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
$x_1x_2x_3x_4x_7$	1	1	1	1	0	0	1	0	0
$x_1x_2x_3x_4x_8$	1	1	1	1	0	0	0	1	0
$x_1x_2x_3x_5$	1	1	1	0	1	0	0	0	0
$x_1x_2x_3x_6$	1	1	1	0	0	1	0	0	0
$x_1x_2x_9$	1	1	0	0	0	0	0	0	1

Cube-Literal Matrix & Rectangles (1/2)

- A rectangle (R, C) of a matrix A is a subset of rows R and columns C such that

$$A_{ij} = 1 \forall\ i \in R,\ j \in C$$

  - Rows and columns need not be continuous
- A prime rectangle is a rectangle not contained in any other rectangle
  - A prime rectangle indicates a co-kernel kernel pair

Cube-Literal Matrix & Rectangles (2/2)

- Example:

  R = {{1, 2, 3, 4},{1, 2, 3}}

  - co-kernel: $x_1x_2x_3$
  - kernel: $x_4x_7 + x_4x_8 + x_5 + x_6$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
$x_1x_2x_3x_4x_7$	1	1	1	1	0	0	1	0	0
$x_1x_2x_3x_4x_8$	1	1	1	1	0	0	0	1	0
$x_1x_2x_3x_5$	1	1	1	0	1	0	0	0	0
$x_1x_2x_3x_6$	1	1	1	0	0	1	0	0	0
$x_1x_2x_9$	1	1	0	0	0	0	0	0	1

Rectangles and Logic Synthesis

- Single cube extraction

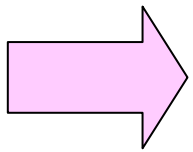
F = abc + abd + eg

G = abfg

H = bd + ef

({1,2,4},{1,2}) <=> ab

({2,5},{2,4}) <=> bd



F = Xc + XY + eg

G = Xfg

H = Y + ef

X = ab

Y = bd

		a	b	c	d	e	f	g
		1	2	3	4	5	6	7
abc	1	1	1	1	0	0	0	0
abd	2	1	1	0	1	0	0	0
eg	3	0	0	0	0	1	0	1
abfg	4	1	1	0	0	0	1	1
bd	5	0	1	0	1	0	0	0
ef	6	0	0	0	0	1	1	0

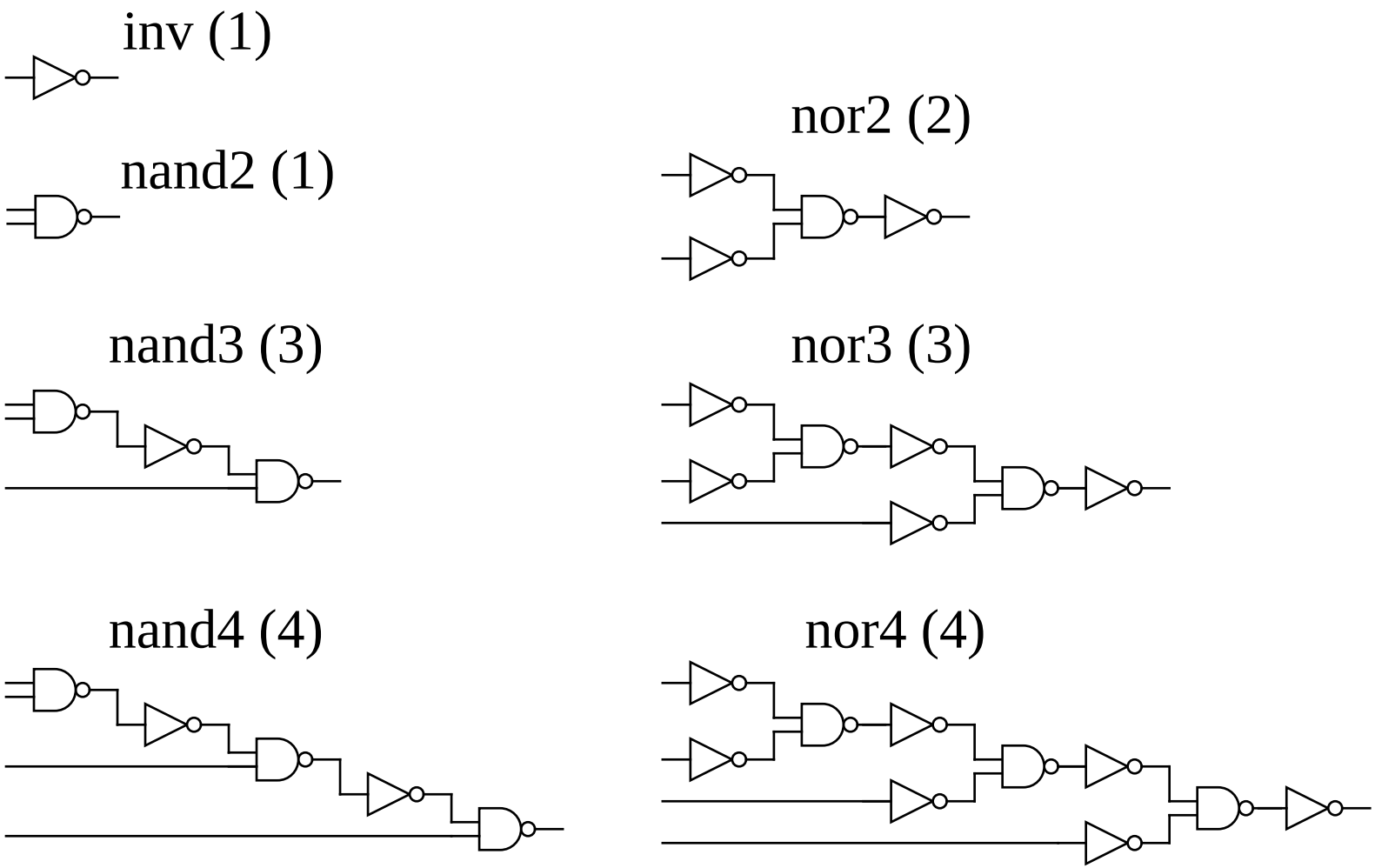
Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level logic optimization
  - Multi-level logic optimization
- Technology mapping
- Timing optimization
- Synthesis for low power
- Retiming

Technology Mapping

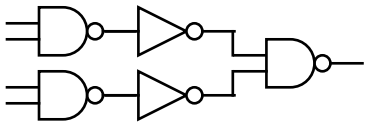
- General approach:
  - Choose base function set for canonical representation
    - Ex: 2-input NAND and Inverter
  - Represent optimized network using base functions
    - Subject graph
  - Represent library cells using base functions
    - Pattern graph
  - Each pattern associated with a cost which is dependent on the optimization criteria
- Goal:
  - Finding a minimal cost covering of a subject graph using pattern graphs

Example Pattern Graph (1/3)

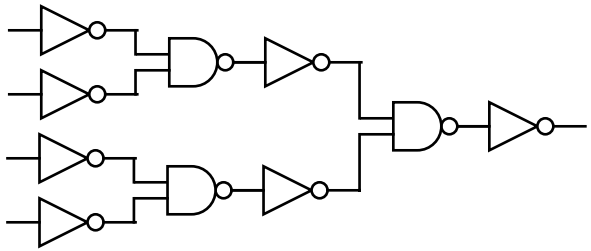


Example Pattern Graph (2/3)

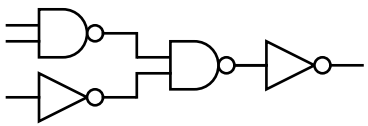
nand4 (4)



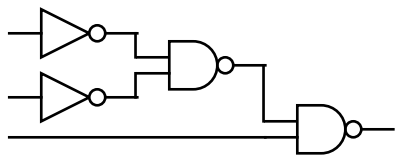
nor4 (4)



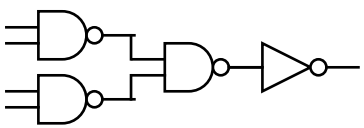
aoi21 (3)



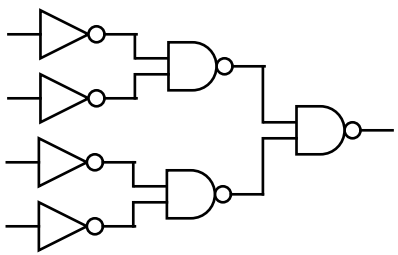
oai21 (3)



aoi22 (4)

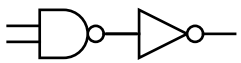


oai22 (4)

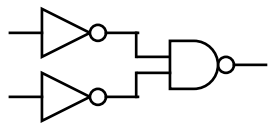


Example Pattern Graph (3/3)

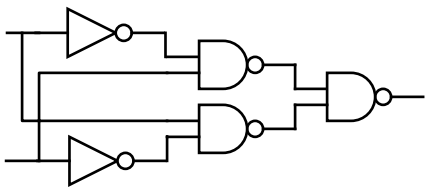
and2 (3)



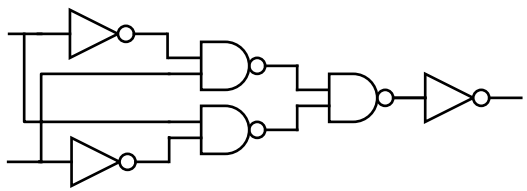
or2 (3)



xor (5)

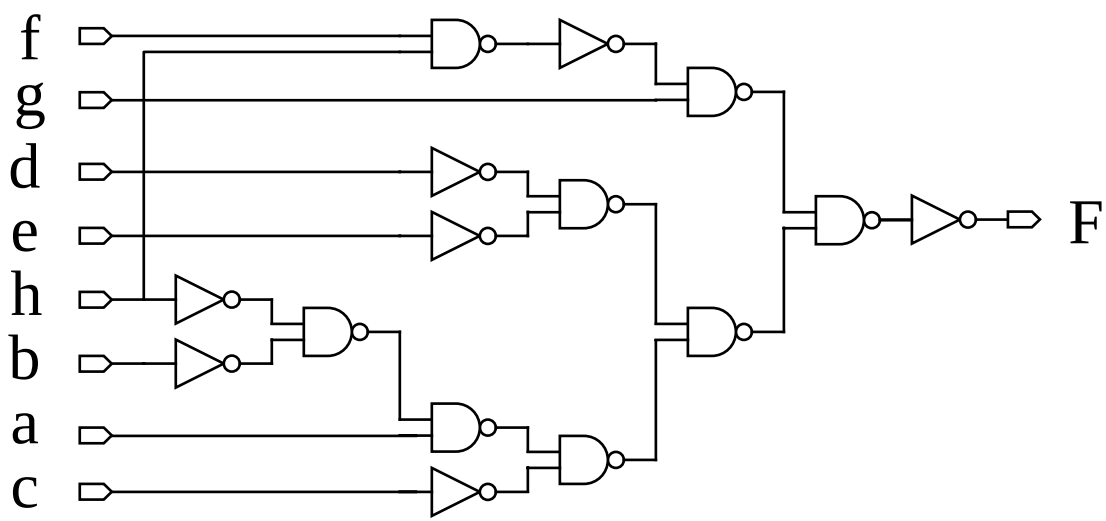


xnor (5)

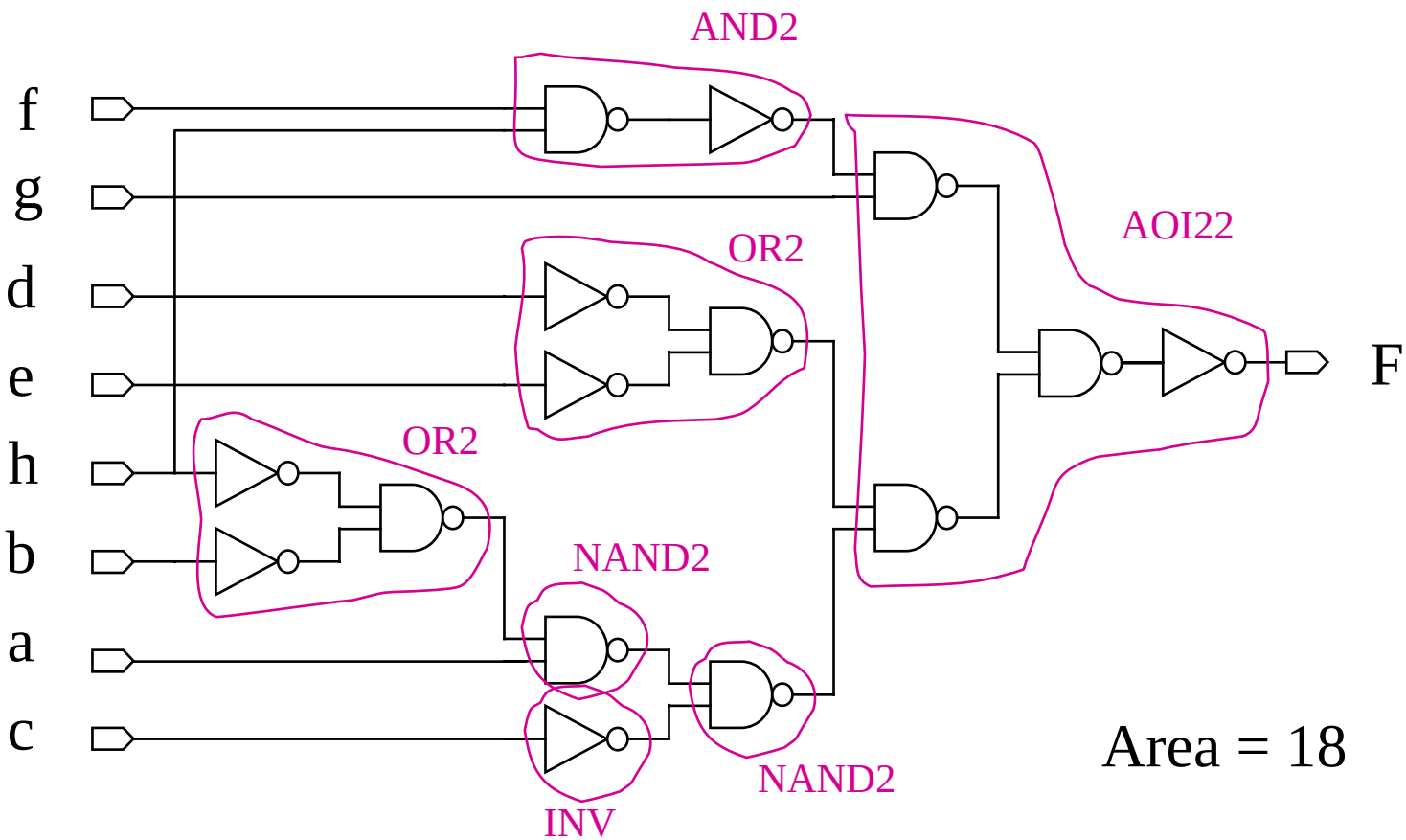


Example Subject Graph

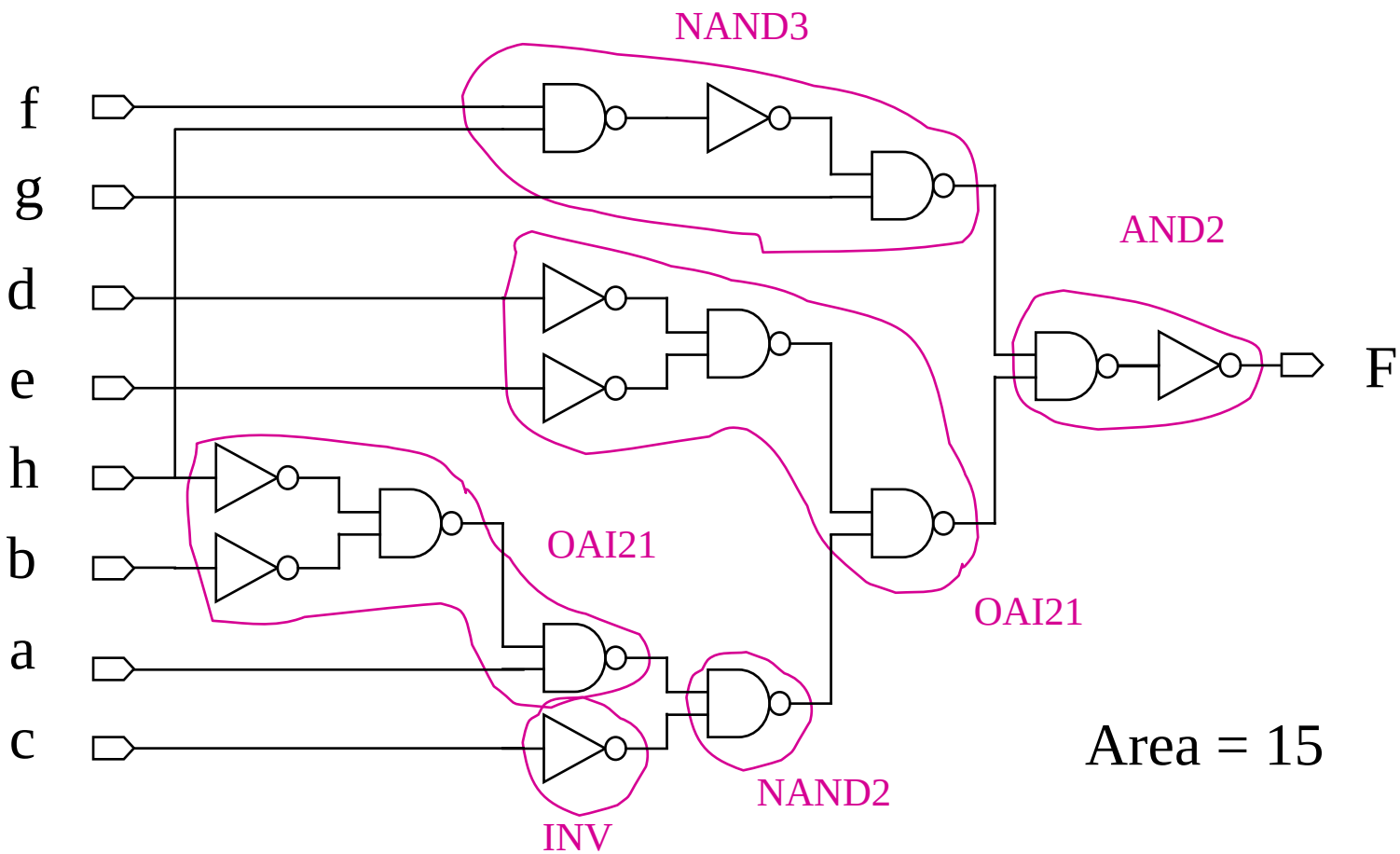
t1 = d + e;
t2 = b + h;
t3 = a t2 + c;
t4 = t1 t3 + f g h;
F = t4’;



Sample Covers (1/2)

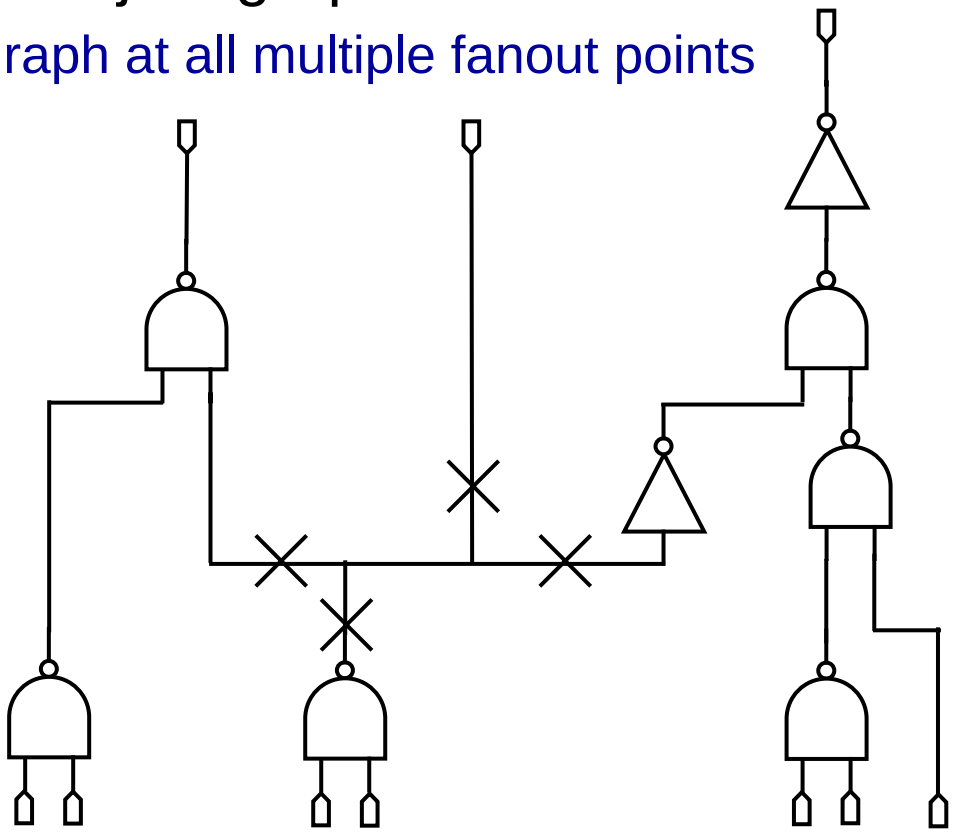


Sample Covers (2/2)



DAGON Approach

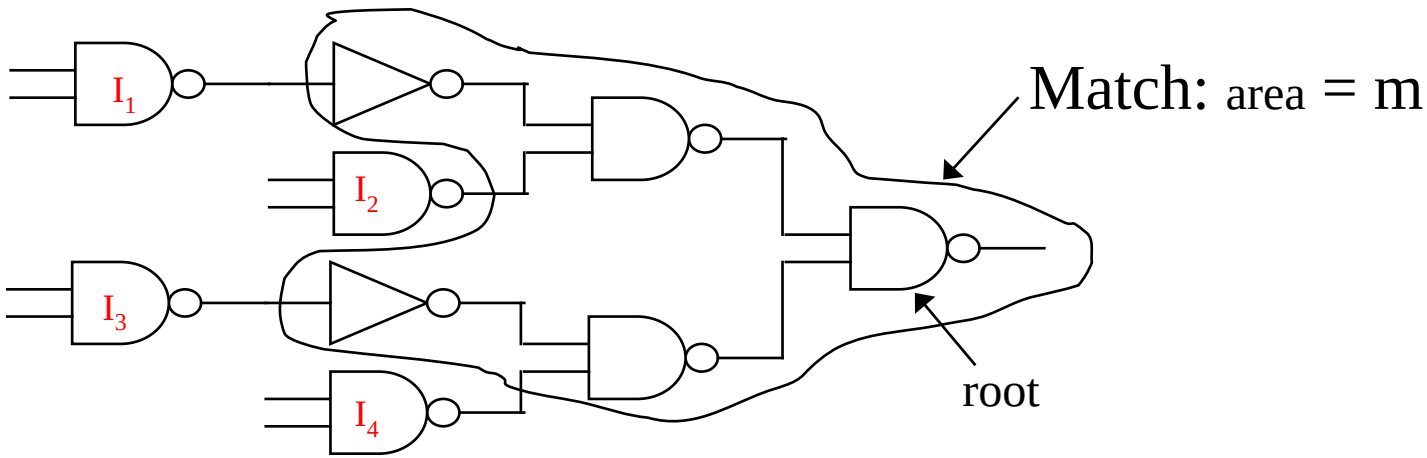
- Partition a subject graph into trees
  - Cut the graph at all multiple fanout points



- Optimally cover each tree using dynamic programming approach
- Piece the tree-covers into a cover for the subject graph

Dynamic Programming for Minimum Area

- Principle of optimality: optimal cover for the tree consists of a match at the root plus the optimal cover for the sub-tree starting at each input of the match

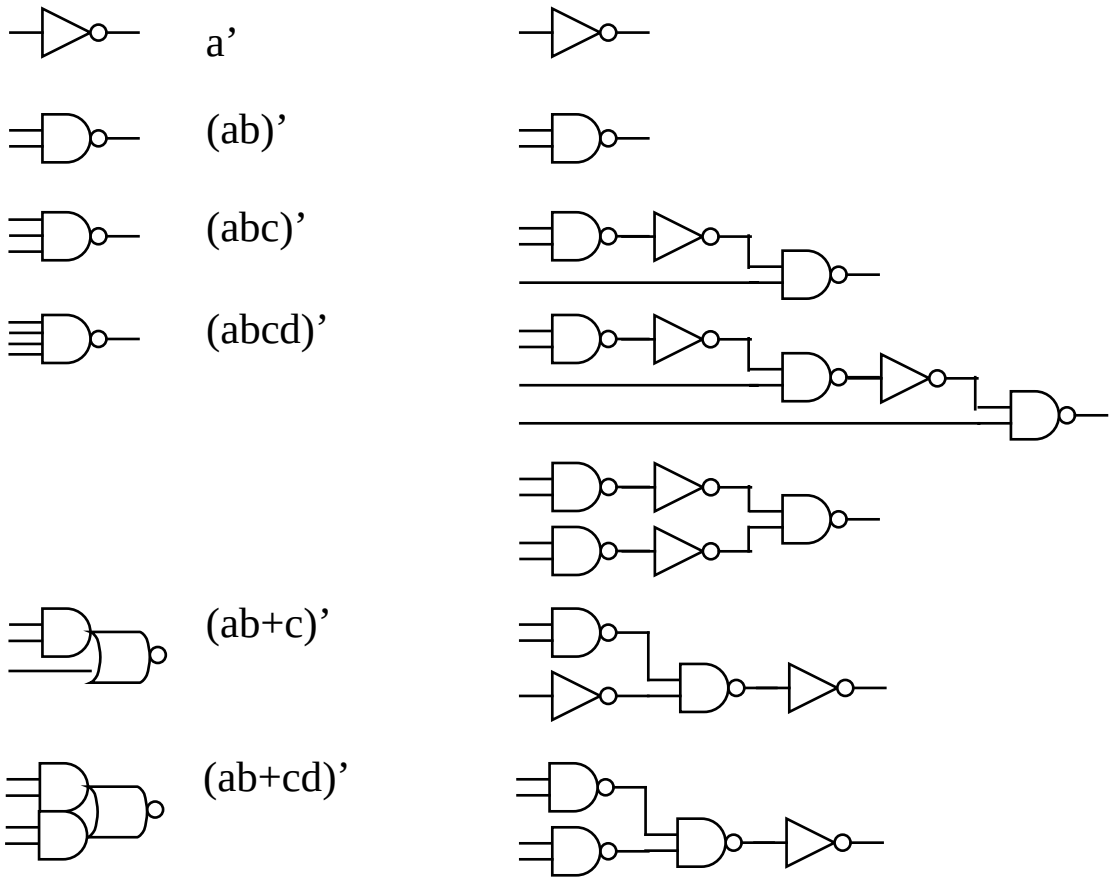


$$A(root) = m + A(I_1) + A(I_2) + A(I_3) + A(I_4)$$

cost of a leaf = 0

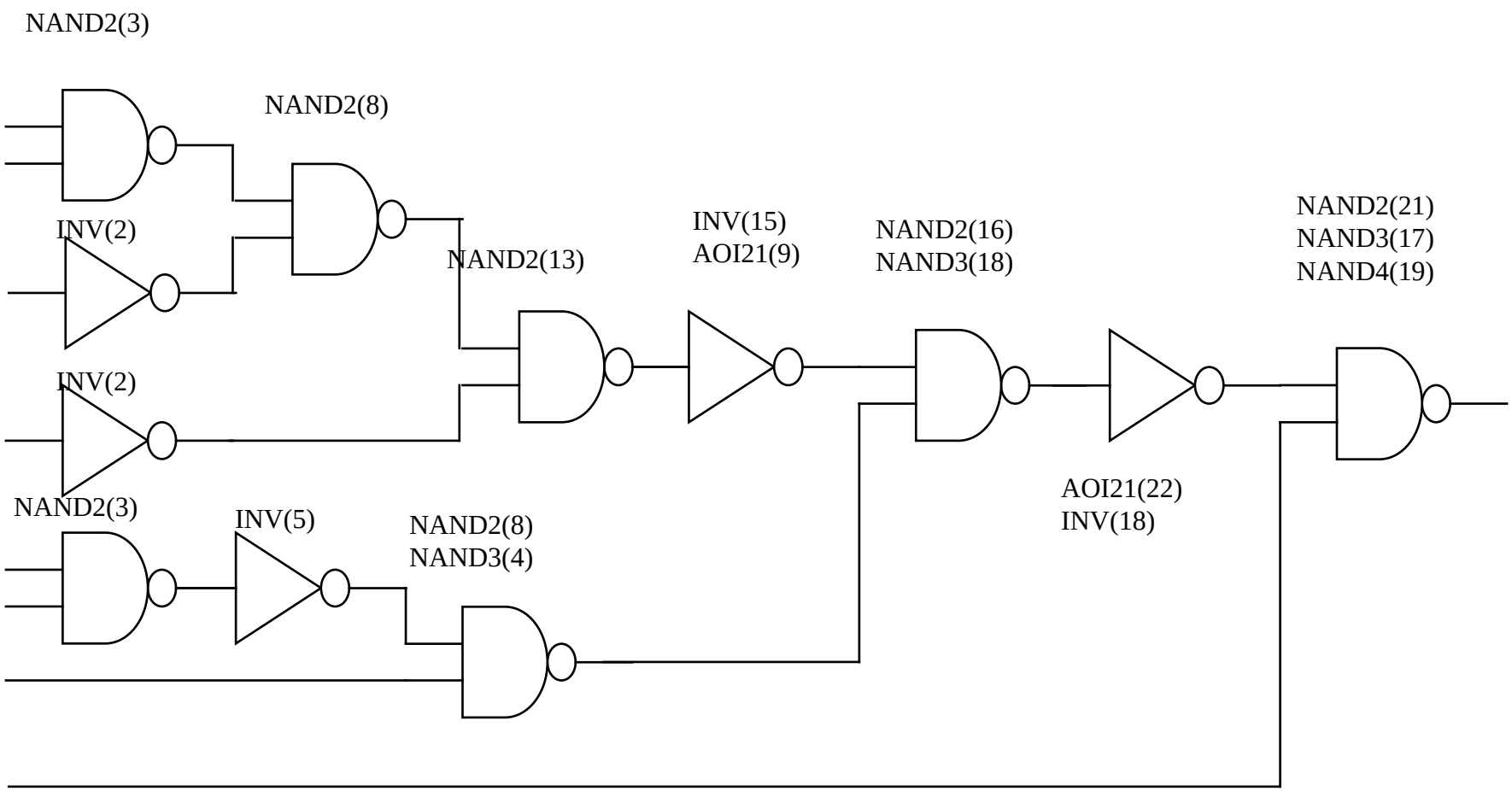
A Library Example

INV	2	a’	
NAND2	3	(ab)’	
NAND3	4	(abc)’	
NAND4	5	(abcd)’	
AOI21	4	(ab+c)’	
AOI22	5	(ab+cd)’	



Library Element Canonical Form

DAGON in Action

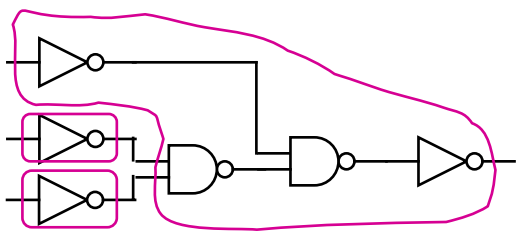


Features of DAGON

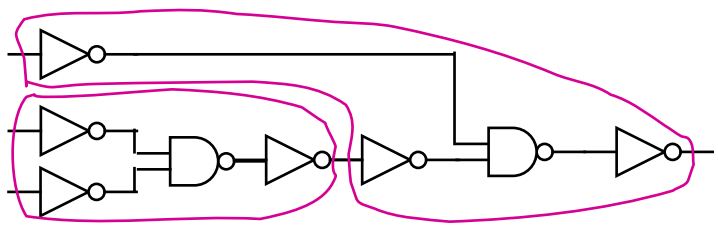
- Pros. of DAGON:
  - Strong algorithmic foundation
  - Linear time complexity
    - Efficient approximation to graph-covering problem
  - Given locally optimal matches in terms of both area and delay cost functions
  - Easily “portable” to new technologies
- Cons. Of DAGON:
  - With only a local (to the tree) notion of timing
    - Taking load values into account can improve the results
  - Can destroy structures of optimized networks
    - Not desirable for well-structured circuits
  - Inability to handle non-tree library elements (XOR/XNOR)
  - Poor inverter allocation

Inverter Allocation

- Add a pair of inverters for each wire in the subject graph
- Add a pattern of a wire that matches two inverters with zero cost
- Effect: may further improve the solution



2 INV
1 AIO21



2 NOR2

Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level logic optimization
  - Multi-level logic optimization
- Technology mapping
- Timing optimization
- Synthesis for low power
- Retiming

Delay Model at Logic Level

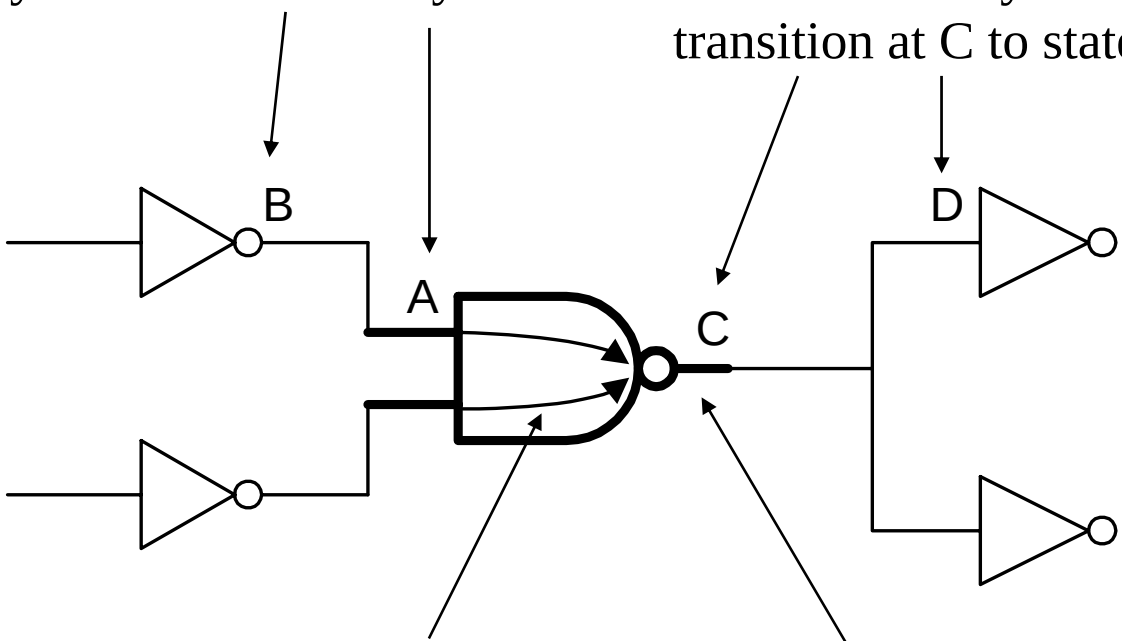
1. unit delay model
   - Assign a delay of 1 to ***each gate***
2. unit fanout delay model
   - Incorporate an ***additional*** delay for ***each fanout***
3. library delay model
   - Use delay data in the library to provide more accurate delay value
   - May use linear or non-linear (tabular) models

Linear Delay Model

Delay = Dslope + Dintrinsic + Dtransition + Dwire

Ds : Slope delay : delay at input A caused by the transition delay at B

Dw : Wire delay : time from state transition at C to state transition at D

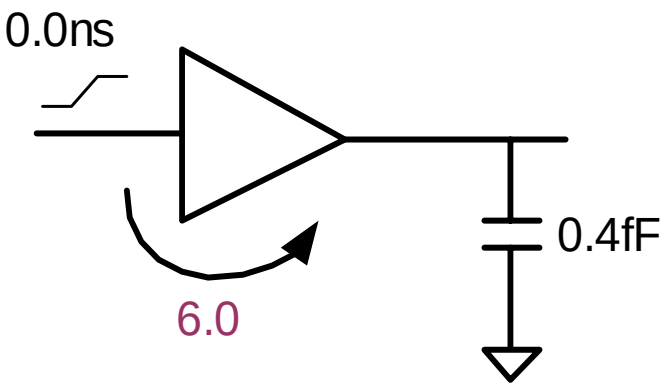


D_I : Intrinsic delay : incurred from cell input to cell output

D_T : Transition delay : output pin loading, output pin drive

Tabular Delay Model

- Delay values are obtained by a look-up table
  - Two-dimensional table of delays (m by n)
    - with respect to input slope (m) and total output capacitance (n)
  - One dimensional table model for output slope (n)
    - with respect to total output capacitance (n)
  - Each value in the table is obtained by real measurement



Total Output Load (fF)

Input Transition (ns)	0.2	0.3	0.4	0.5
0	3	4.5	6	7
0.1	5	8	10.7	13

Cell Delay (ps)

- Can be more precise than linear delay model
  - table size ↑ ➔ accuracy ↑
- Require more space to store the table

Arrival Time and Required Time

- arrival time : calculated from input to output
- required time : calculated from output to input
- slack = required time - arrival time

A(j): arrival time of signal j

R(k): required time or for signal k

S(k): slack of signal k

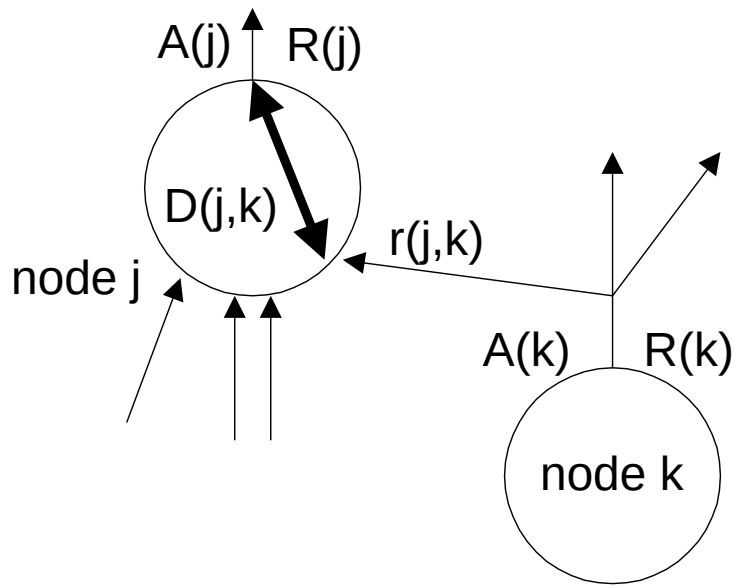
D(j,k): delay of node j from input k

$A(j) = \max_{k \in FI(j)} [A(k) + D(j,k)]$

$r(j,k) = R(j) - D(j,k)$

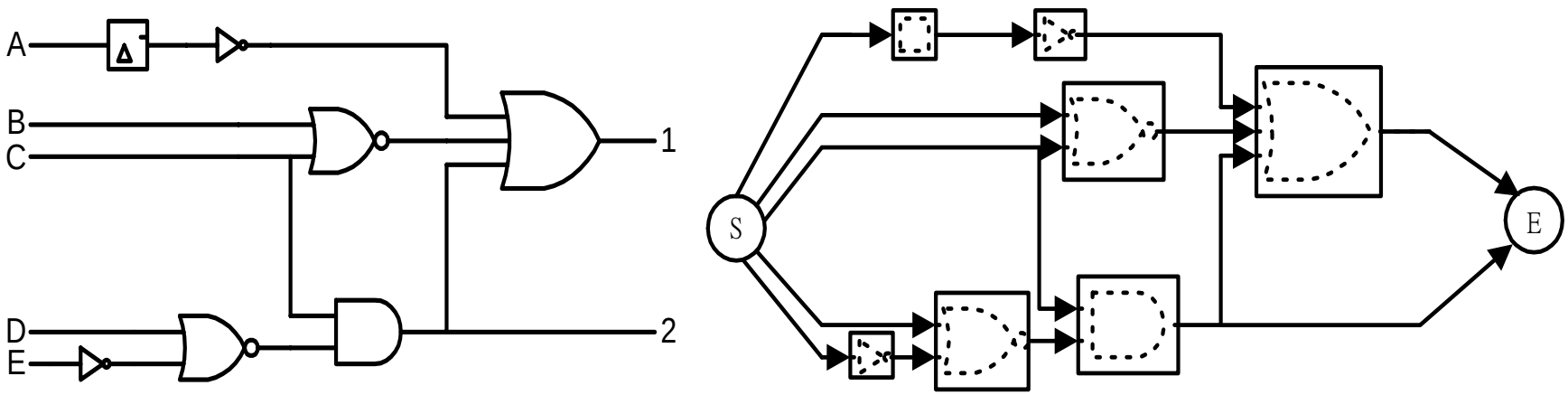
$R(k) = \min_{j \in FO(k)} [r(j,k)]$

$S(k) = R(k) - A(k)$



Delay Graph

- Replace logic gates with delay blocks
- Add start (S) and end (E) blocks
- Indicate signal flow with directed arcs

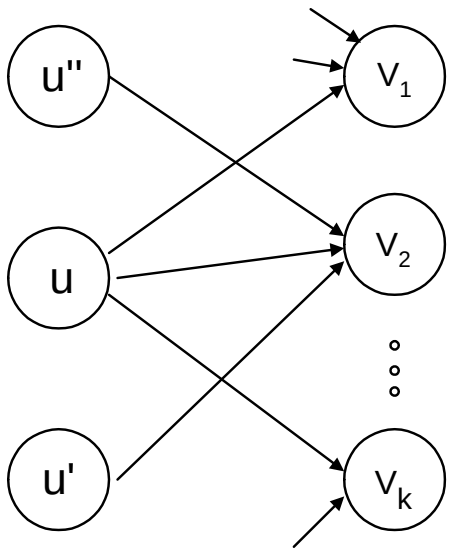


Longest and Shortest Path

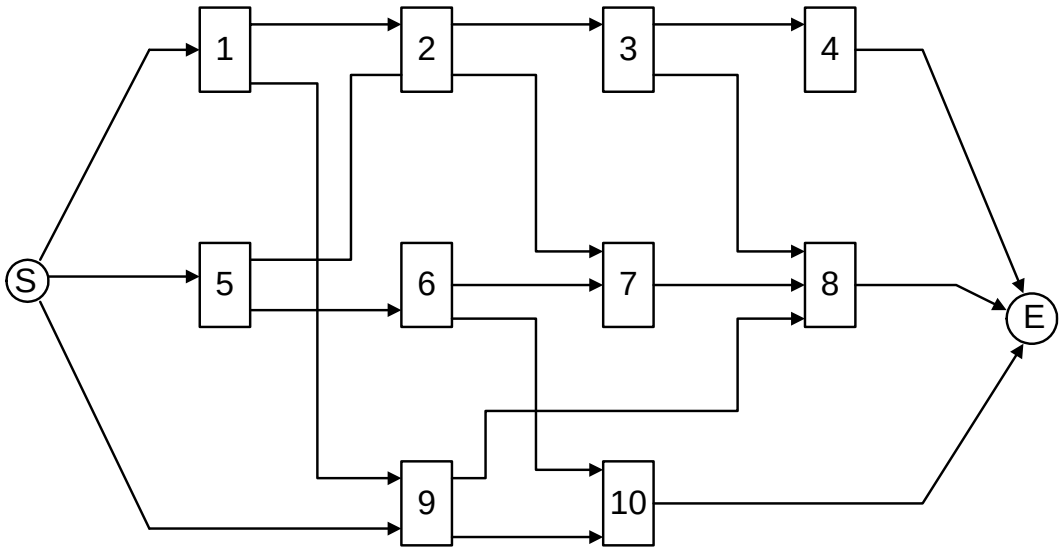
- If we visit vertices in precedence order, the following code will need executing only once for each *u*

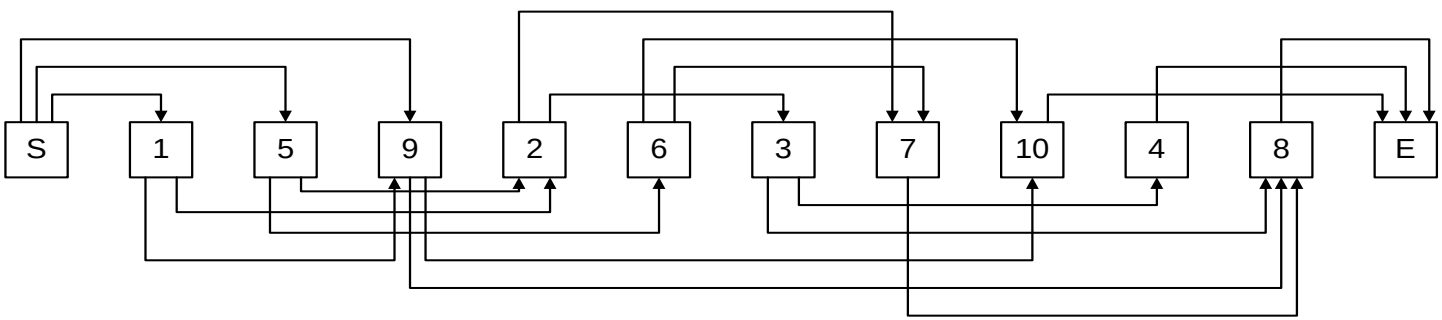
Update Successors[u]

```
1  for each vertex v ∈ Adj[u] do
2      if  A[v] < A[u] + Δ[u]   // longest
3          then A[v] ← A[u] + Δ [u]
4          LP[v] ← u  fi
5      if  a[v] >a[u] + δ[u]   // shortest
6          then a[v] ← a[u] + δ[u]
7          SP[v] ← u  fi
```

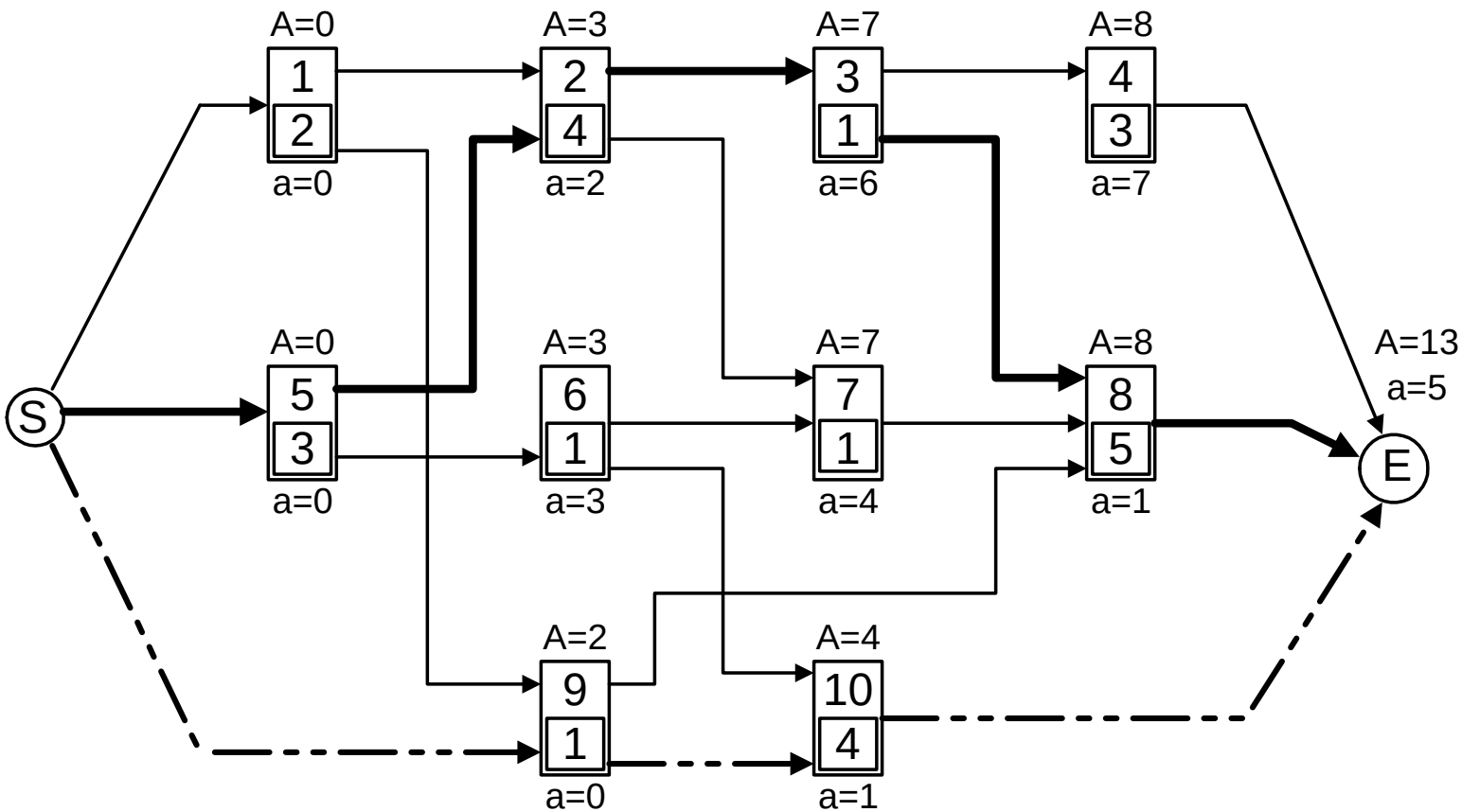


Delay Graph and Topological Sort





Delay Calculation



A=3 → longest path delay

2 → node number

4 → gate delay

a=2 → shortest path delay

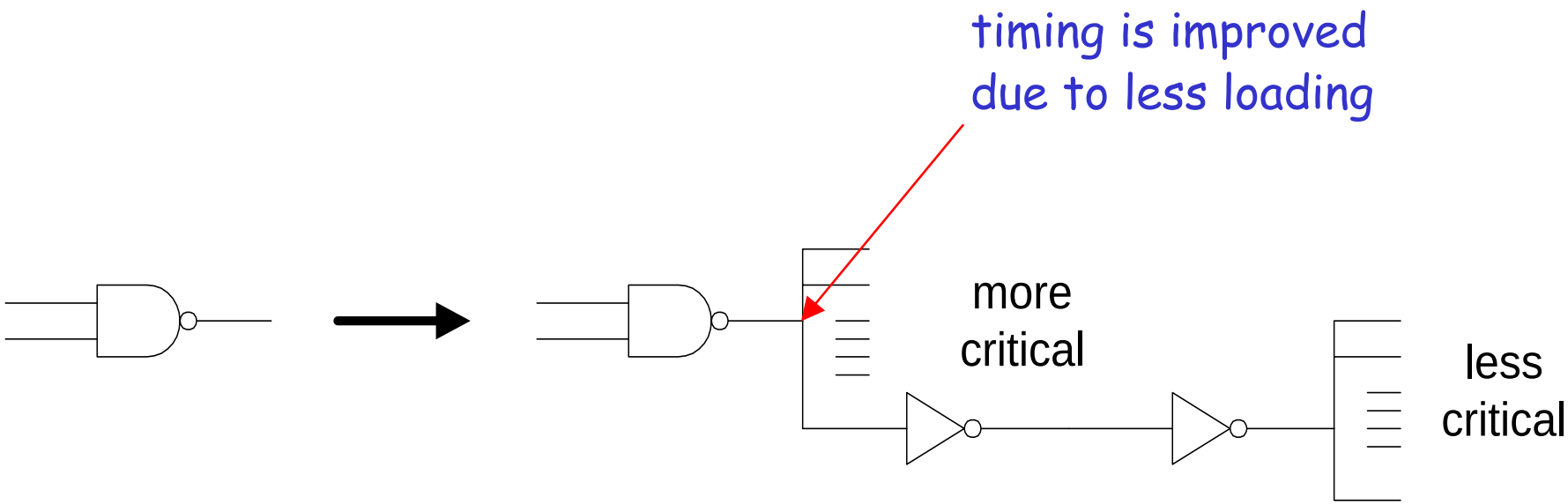
P.S: The longest delay and shortest delay of each gate are assumed to be the same.

Timing Optimization Techniques (1/8)

- Fanout optimization
  - Buffer insertion
  - Split
- Timing-driven restructuring
  - Critical path collapsing
  - Timing decomposition
- Misc
  - De Morgan
  - Repower
  - Down power
- Most of them will increase area to improve timing
  - Have to make a good trade-off between them

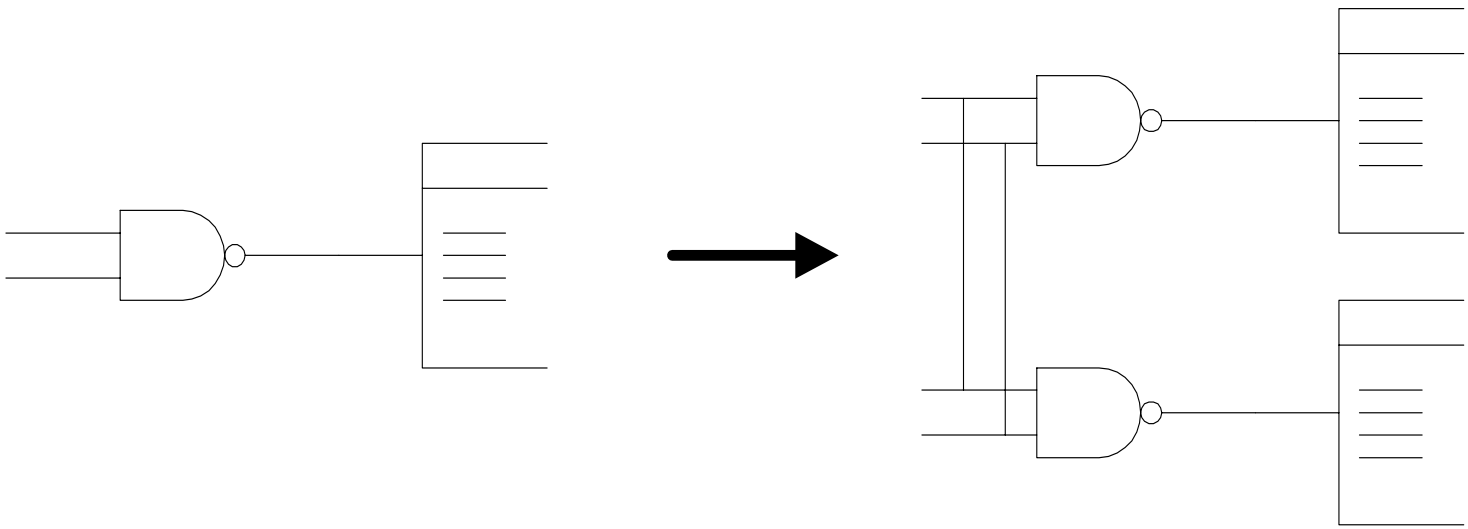
Timing Optimization Techniques (2/8)

- **Buffer insertion**: divide the fanouts of a gate into critical and non-critical parts and drive the non-critical fanouts with a buffer



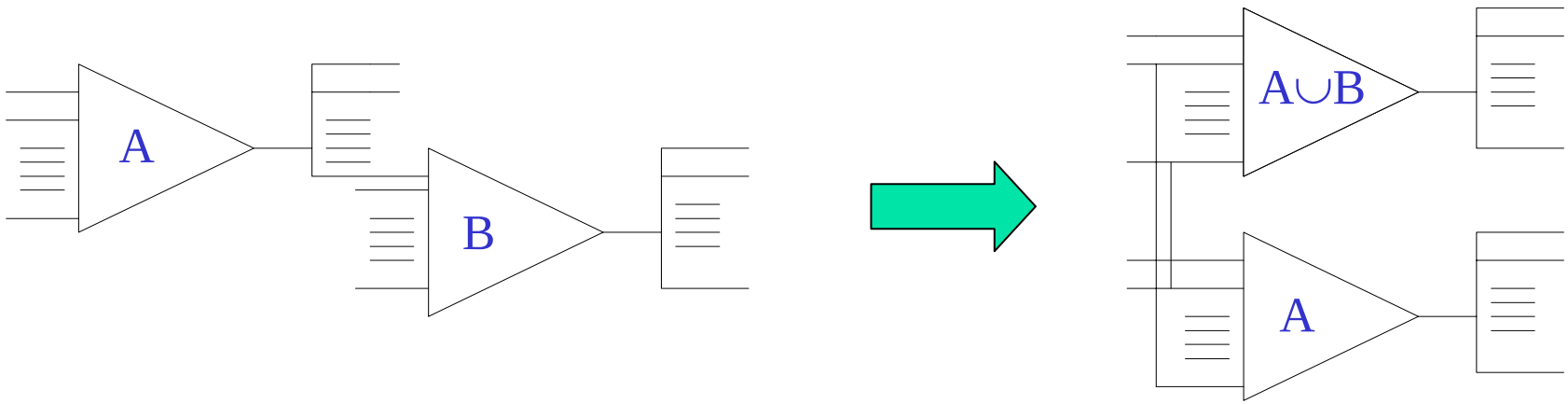
Timing Optimization Techniques (3/8)

- **Split**: split the fanouts of a gate into several parts. Each part is driven with a copy of the original gate.



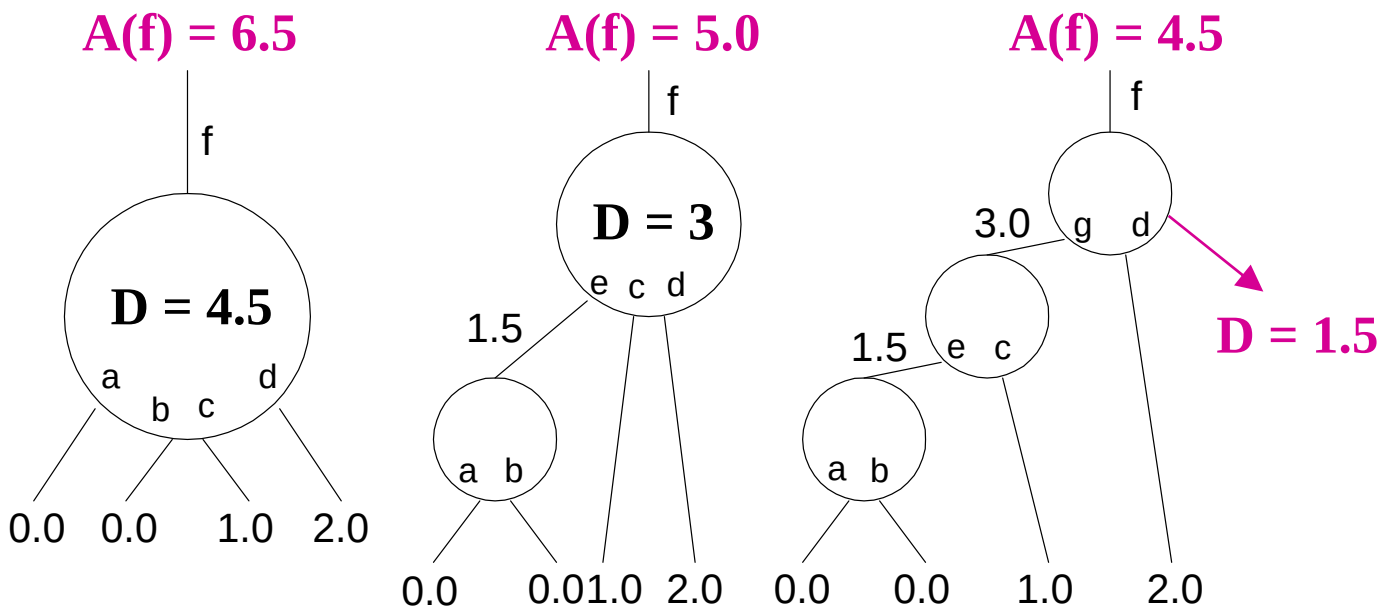
Timing Optimization Techniques (4/8)

- **Critical path collapsing**: reduce the depth of logic networks



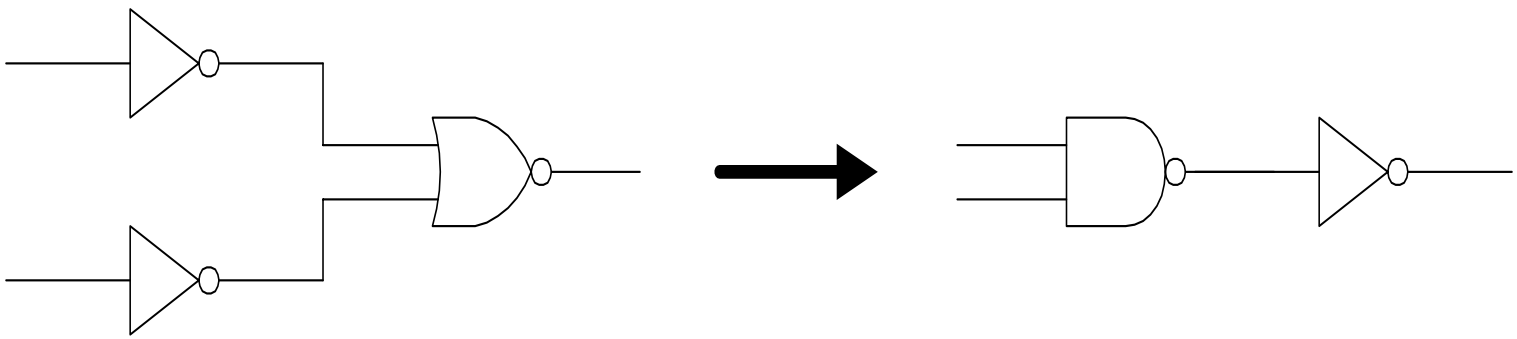
Timing Optimization Techniques (5/8)

- **Timing decomposition**: restructuring the logic networks to minimize the arrival time



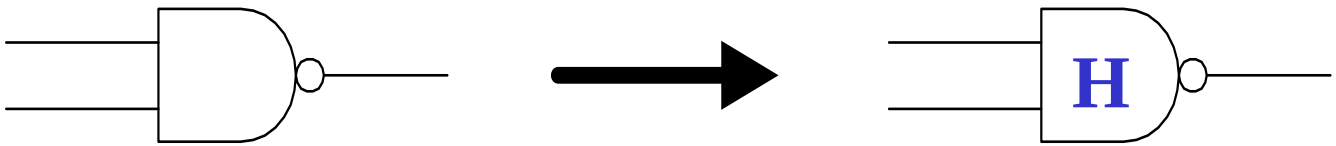
Timing Optimization Techniques (6/8)

- **De Morgan**: replace a gate with its dual, and reverse the polarity of inputs and output
  - NAND gate is typically faster than NOR gate



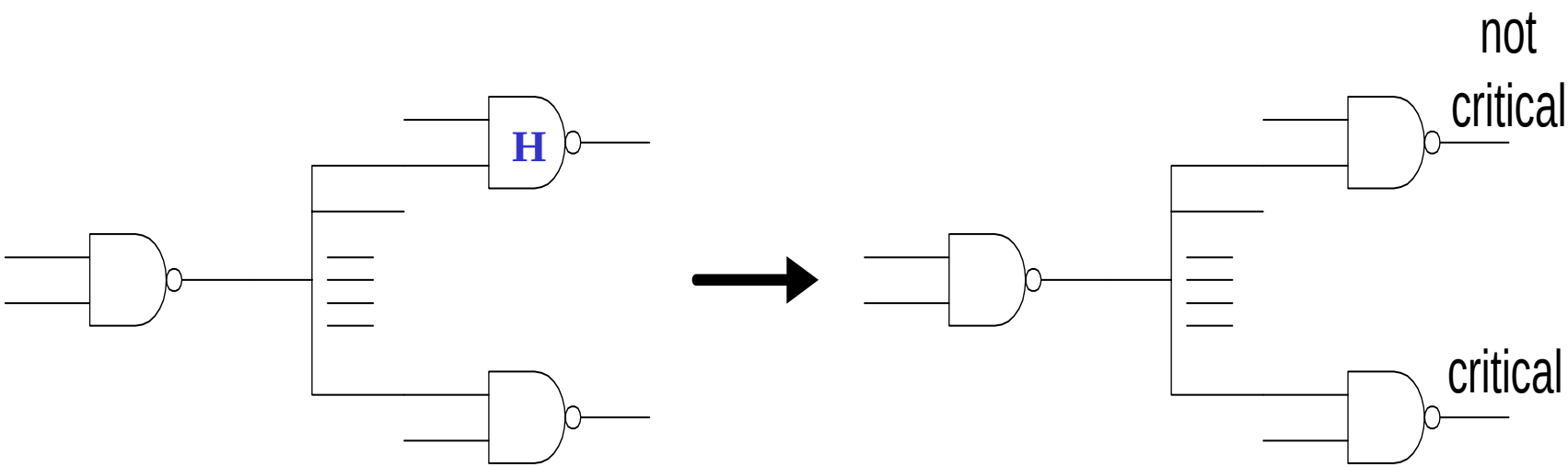
Timing Optimization Techniques (7/8)

- **Repower**: replace a gate with one of the other gate in its logic class with higher driving capability



Timing Optimization Techniques (8/8)

- **Down power**: reducing gate size of a non-critical fanout in the critical path



Restructuring Algorithm

While (circuit timing improves) do

    select regions to transform

    collapse the selected region

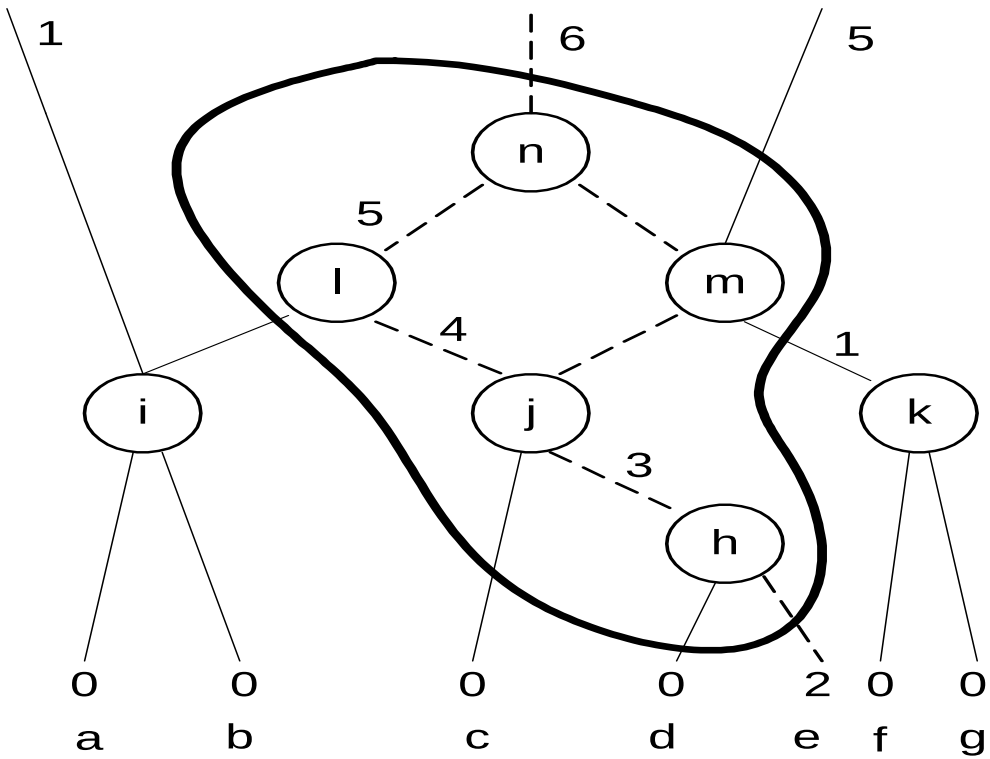
    resynthesize for better timing

done

- Which regions to restructure ?
- How to resynthesize to minimize delay ?

Restructuring Regions

- All nodes with slack within ε of the most critical signal belong to the ε-*network*
- To improve circuit delay, necessary and sufficient to improve delay at nodes on cut-set of ε-*network*



Find the Cutset

- The weight of each node is W = Wxt + α * Wxa
  - Wxt is potential for speedup
  - Wxa is area penalty for duplication of logic
  - α is decided by various area/delay tradeoff
- Apply the maxflow-mincut algorithm to generate the cutset of the ε-*network*

Controlling the Algorithm

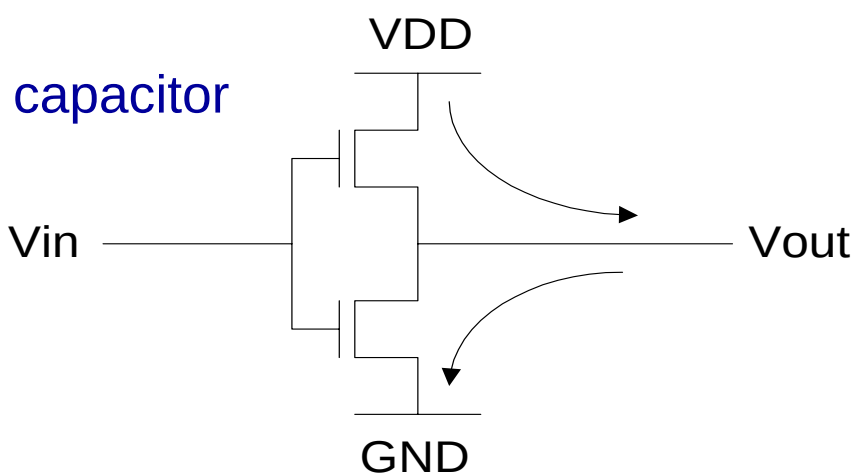
- ε: Specify the size of the ε-*network*
  - Large ε might waste area without much reduction in critical delay
  - Small ε might slow down the algorithm
- *d*: The depth of the *d-critical-fanin-section*
  - Large *d* might make large change in the delay
  - Large *d* might increase run time rapidly due to the collapsing effort and the large number of divisor
- α: Control the tradeoff between area and speed
  - Large α avoids the duplication of logic
  - α = 0 implies a speedup irrespective of the increase in area

Outline

- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level logic optimization
  - Multi-level logic optimization
- Technology mapping
- Timing optimization
- Synthesis for low power
- Retiming

Power Dissipation

- Leakage power
  - Static dissipation due to leakage current
  - Typically a smaller value compared to other power dissipation
  - Getting larger and larger in deep-submicron process
- Short-circuit power
  - Due to the short-circuit current when both PMOS and NMOS are open during transition
  - Typically a smaller value compared to dynamic power
- Dynamic power
  - Charge and discharge of a load capacitor
  - Usually the major part of total power consumption



Power Dissipation Model

$$P = \frac{1}{2} \bullet C \bullet V_{dd}^2 \bullet D$$

- Typically, ***dynamic power*** is used to represent total power dissipation

  P: the power dissipation for a gate

  C: the load capacitance

  V_{dd}: the supply voltage

  D: the transition density

- To obtain the power dissipation of the circuit, we need
  - The node capacitance of each node (obtained from layout)
  - The transition density of each node (obtained by computation)

The Signal Probability

- Definition: The signal probability of a signal x(t), denoted by P_x^1 is defined as :

$$P_x^1 \equiv \lim_{T\to\infty} \frac{1}{T} \int_{-T/2}^{+T/2} x(t)\, dt$$

where T is a variable about time.

- P_x^0 is defined as the probability of a logic signal X(t) being equal to 0.
- $P_x^0 = 1 - P_x^1$

Transition Density

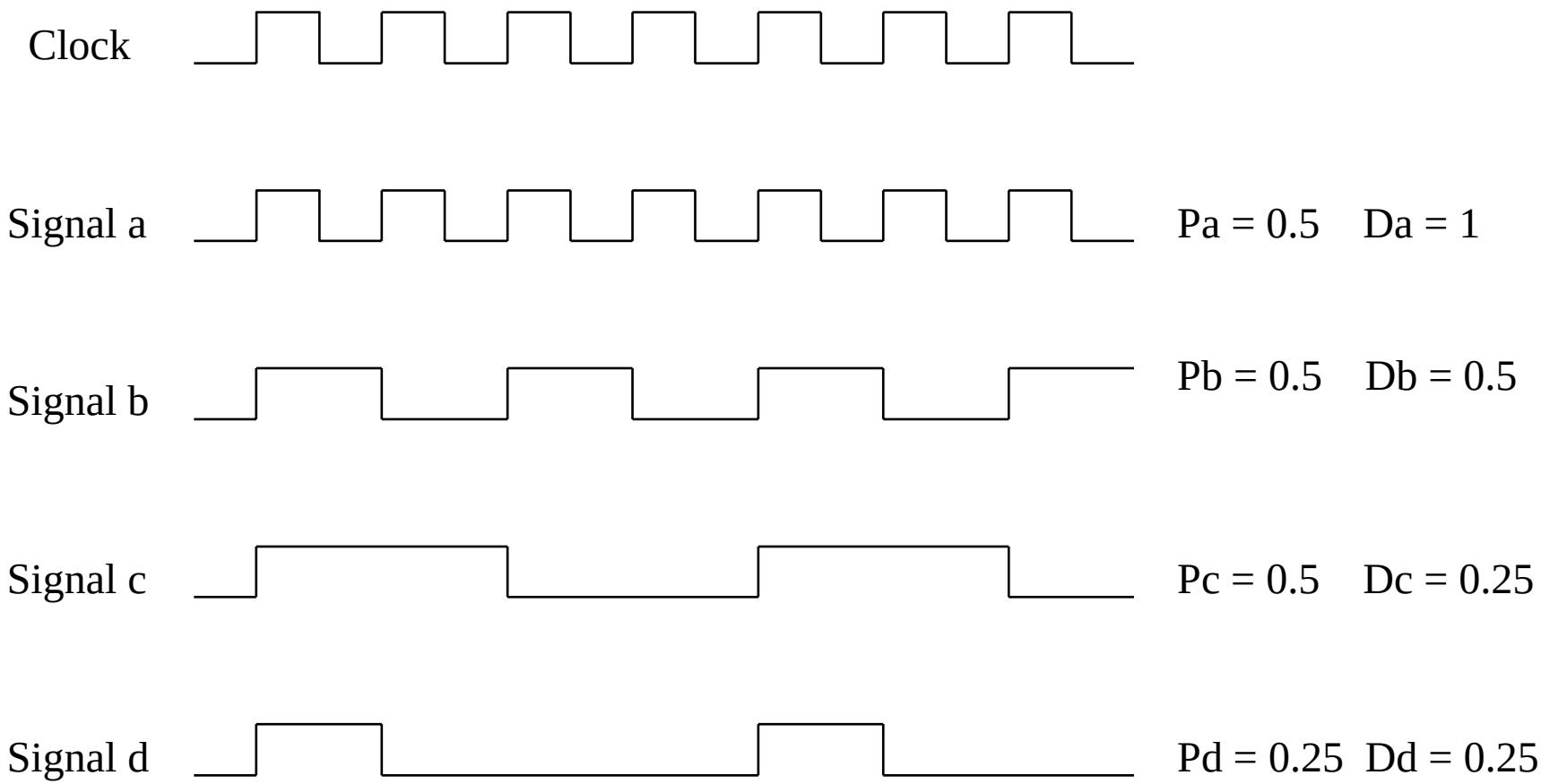
- Definition: The transition density Dx of a logic signal x(t), $t \in (-\infty, \infty)$, is defined as

$$D_X \equiv \lim_{T \to \infty} \frac{n_X(T)}{T \cdot f_C}$$

where f_c is the clock rate or frequency of operation.

- Dx is the expected number of transitions happened in a clock period.
- A circuit with clock rate 20MHz and 5 MHz transitions per second in a node, transition density of this node is 5M / 20M = 0.4

Signal Probability and Transition Density



Signal Probability and Transition Density

$$P_j^{10} + P_J^{11} = P_j^{1-} = P_j^1$$

$$P_j^{01} + P_J^{11} = P_j^{1-} = P_j^1$$

$$P_j^{10} = P_j^{01}$$

$$D_j = P_j^{10} + P_j^{01}$$

$$D_j \leq 2 \times P_j^1$$

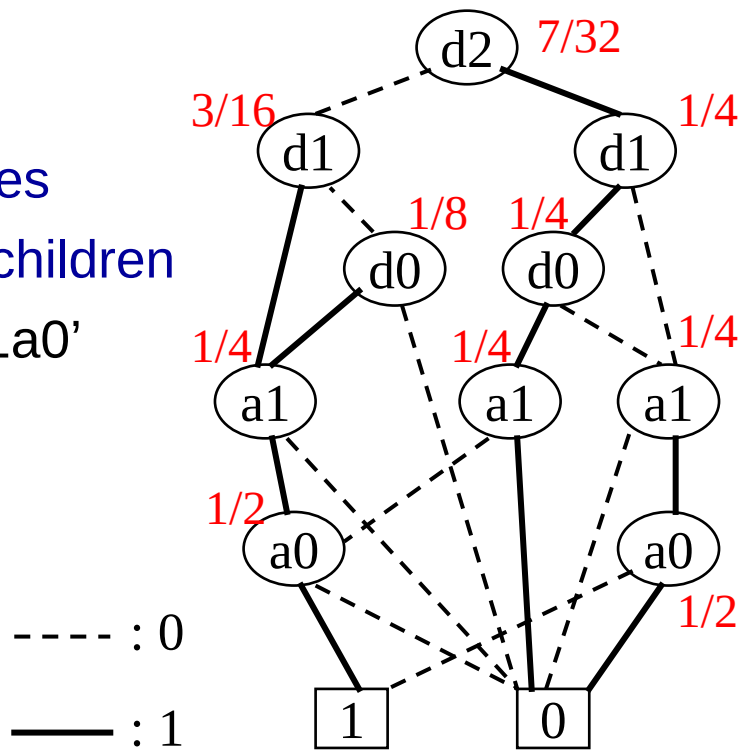
$$D_j \leq 2 \times P_j^0$$

P.S: P^{ab} is the probability of changing from logic state a to b

The Calculation of Signal Probability

- BDD-based approach is one of the popular way
- Definition
  - p(F) : fraction of variable assignments for which F = 1
- Recursive Formulation
  - p(F) = [p(F[x=1]) + p(F[x=0])] / 2
- Computation
  - Compute bottom-up, starting at leaves
  - At each node, average the value of children
- Ex: F = d2'(d1+d0)a1a0 + d2(d1'+d0')a1a0' + d2d1d0a1'a0

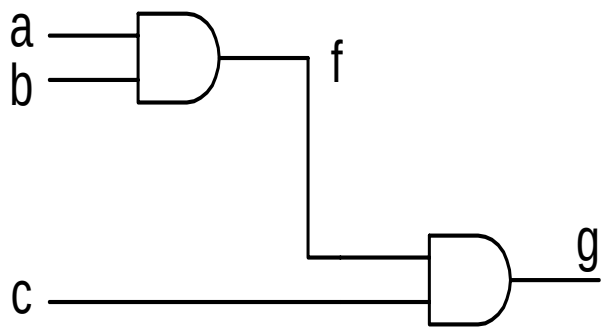
  p(F) = 7/32 = 0.21875



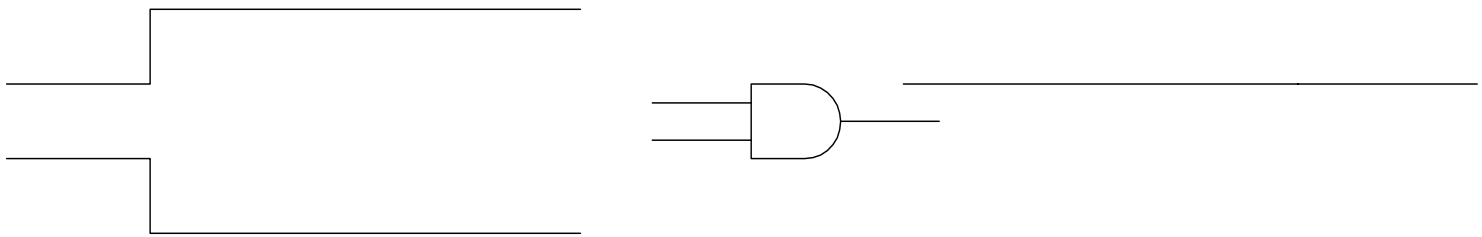
The Calculation of Transition Density

- Transition density of cube
  - f = ab
  - $D_f = D_a P_b + D_b P_a - 1/2\ D_a D_b$
  - $D_a P_b$ means that output will change when b=1 and a has changes
  - 1/2 $D_a D_b$ is the duplicate part when both a and b changes
- n-input AND :
  - a network of 2 -input AND gate in zero delay model
  - 3-input AND gate

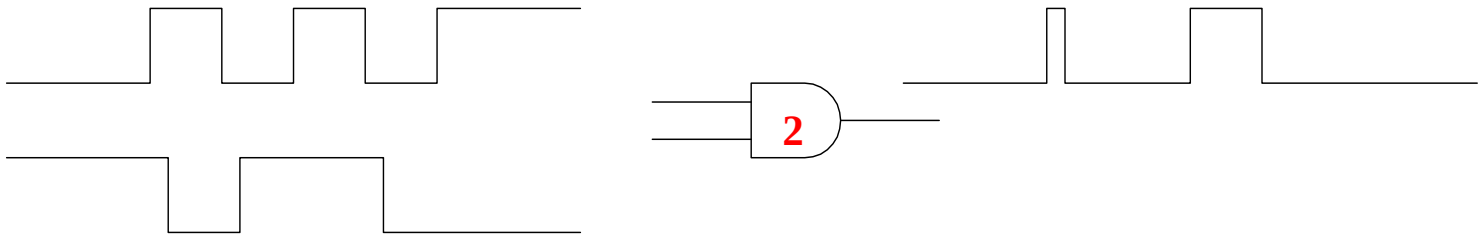
    $D_g = D_f P_c + D_c P_f - 1/2\ D_f D_c$
- Inaccuracy of this simple model :
  - Temporal relations
  - Spatial relations



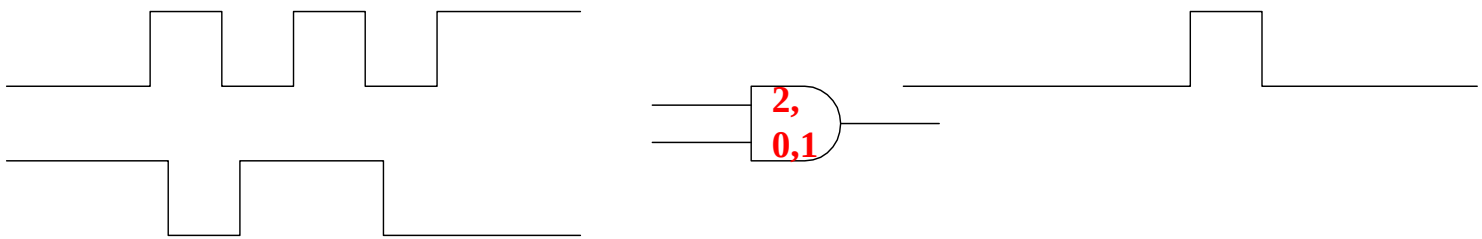
The Problem of Gate Delay and Inertial Delay



(1) Without considering the Gate Delay and Inertial Delay

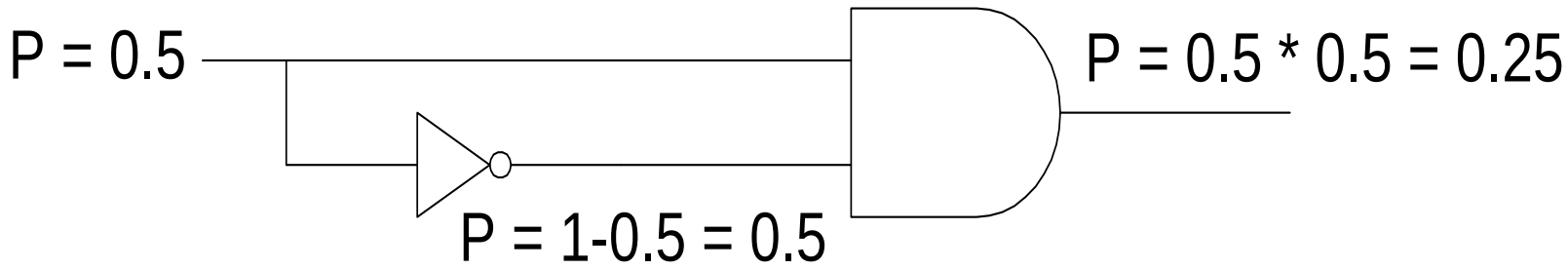


(2) Without considering Inertial Delay

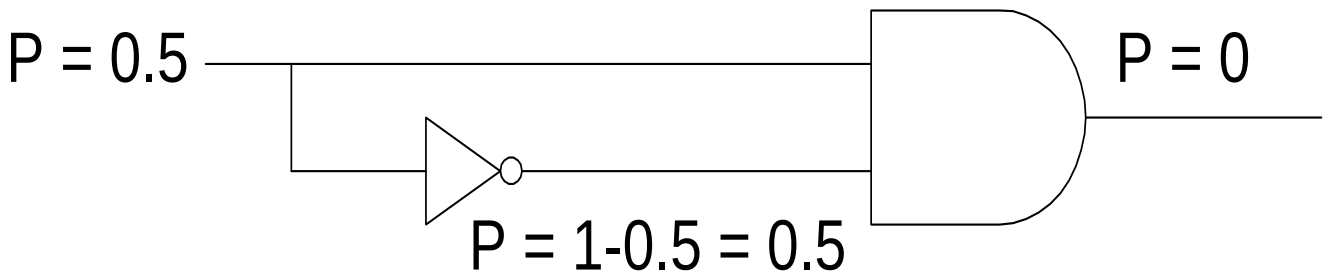


(3) Practical condition

The Problem of Spatial Correlation



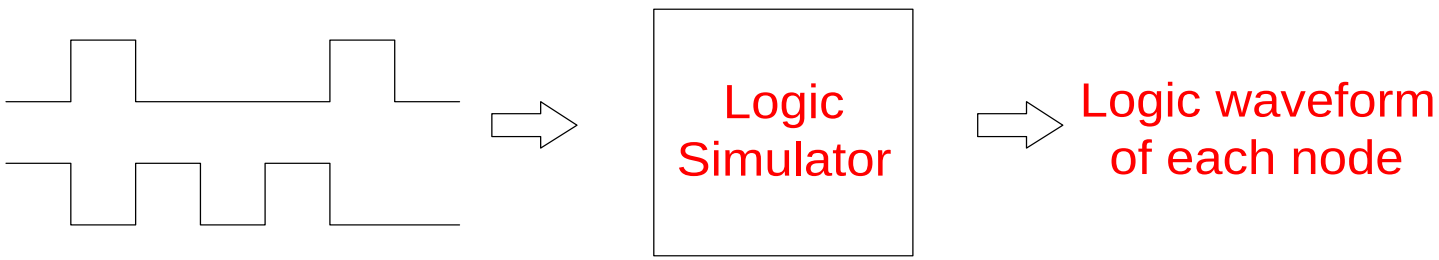
(a) Without considering Spatial Correlation



(b) Practical condition

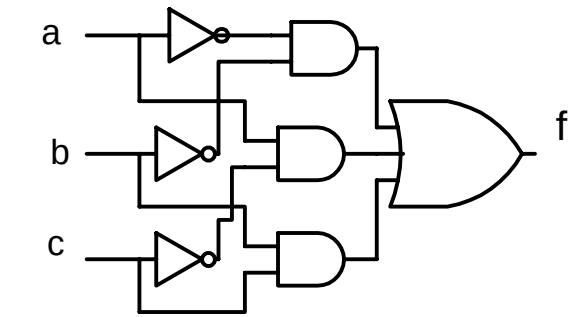
Simulation-Based Computation

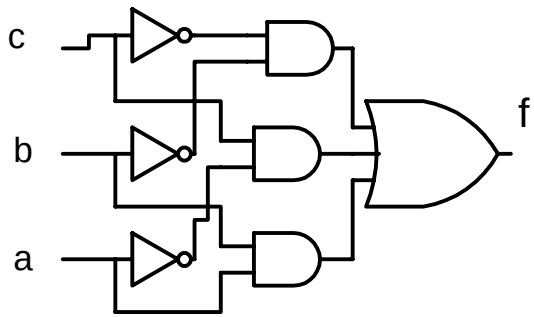
- Input-pattern dependent
- Too many input patterns

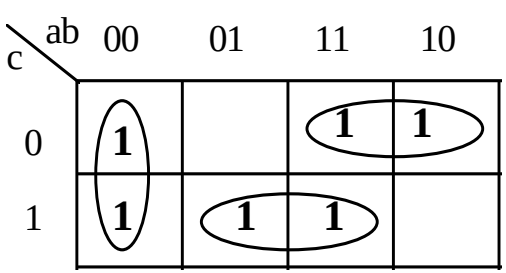


Logic Minimization for Low Power (1/2)

- Consider an example:



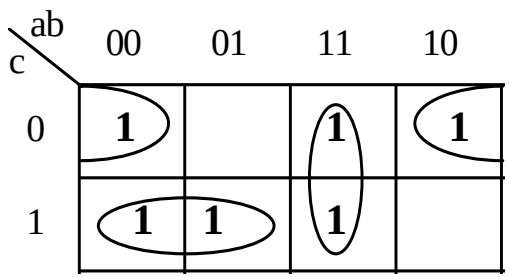




f = a'b' + ac' + bc

P = 108.7 μW

(a)



f = b'c' + a'c + ab

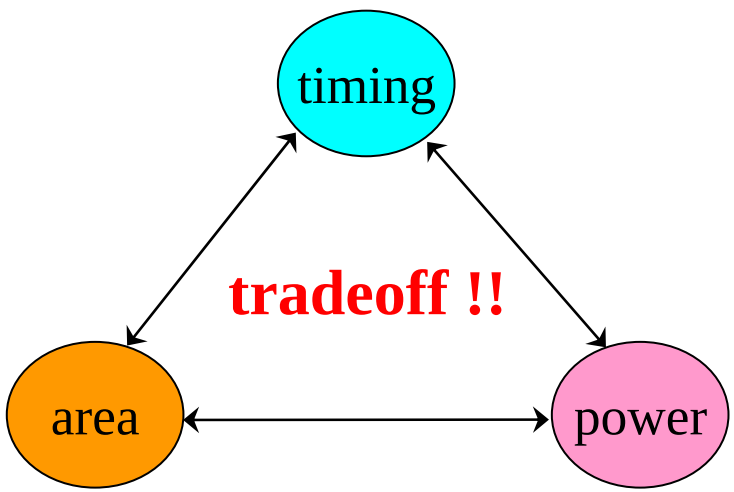
P = 115.5 μW

(b)

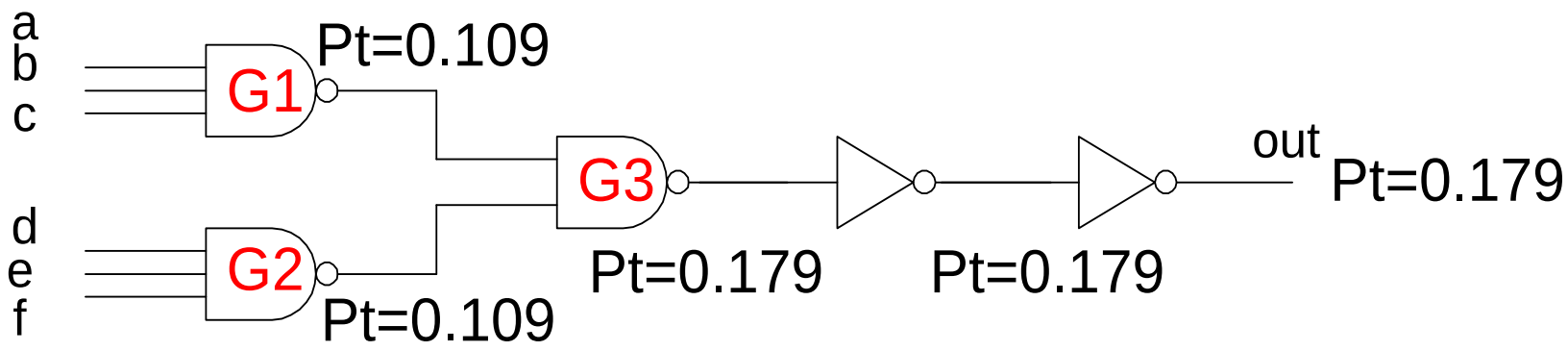
- Different choices of the covers may result in different power consumption

Logic Minimization for Low Power (2/2)

- Typically, the objective of logic minimization is to minimize
  - NPT : the number of product terms of the cover
  - NLI : the number of literals in the input parts of the cover
  - NLO : the number of literals in the output parts of the cover
- For low power synthesis, the power dissipation has to be added into the cost function for best covers



Technology Mapping for Low Power (1/3)

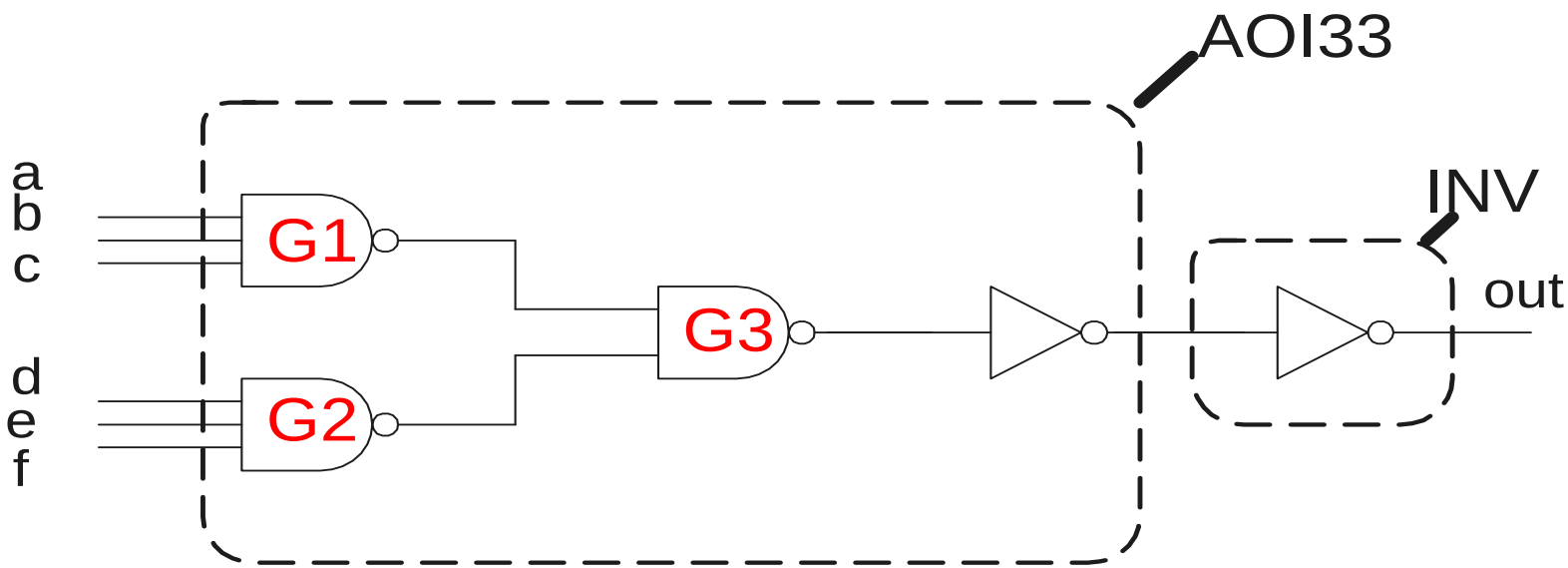


(a) Circuit to be mapped

Gate Type	Area	Intrinsic Cap.	Input Load
INV	928	0.1029	0.0514
NAND2	1392	0.1421	0.0747
NAND3	1856	0.1768	0.0868
AOI33	3248	0.3526	0.1063

(b) Characteristics of Library

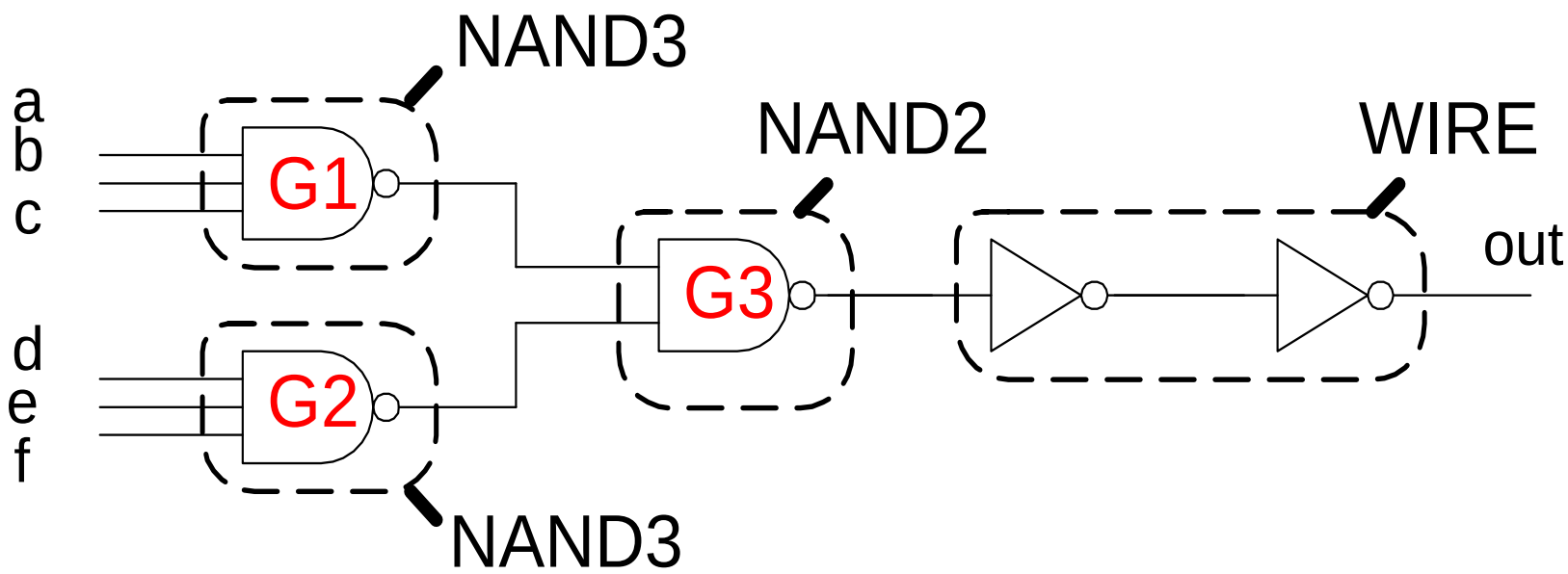
Technology Mapping for Low Power (2/3)



Area Cost: 4176
Power Cost: 0.0907

(a) Minimun-Area Mapping

Technology Mapping for Low Power (3/3)



Area Cost: 5104
Power Cost: 0.0803

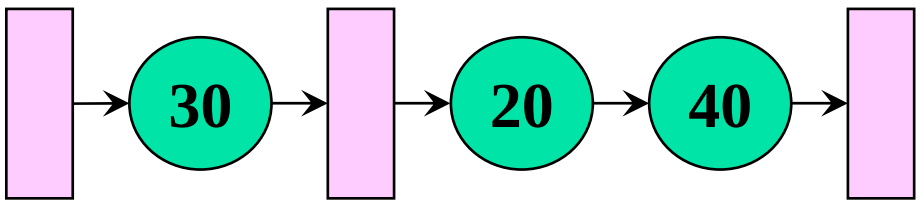
(b) Minimun-Power Mapping

Outline

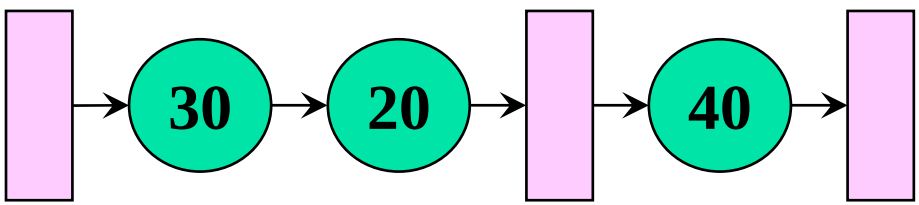
- Synthesis overview
- RTL synthesis
  - Combinational circuit generation
  - Special element inferences
- Logic optimization
  - Two-level logic optimization
  - Multi-level logic optimization
- Technology mapping
- Timing optimization
- Synthesis for low power
- Retiming

Retiming

- Exploit the ability to move registers in a circuit
  - To minimize the cycle time
  - To minimize the the number of registers for a given cycle time



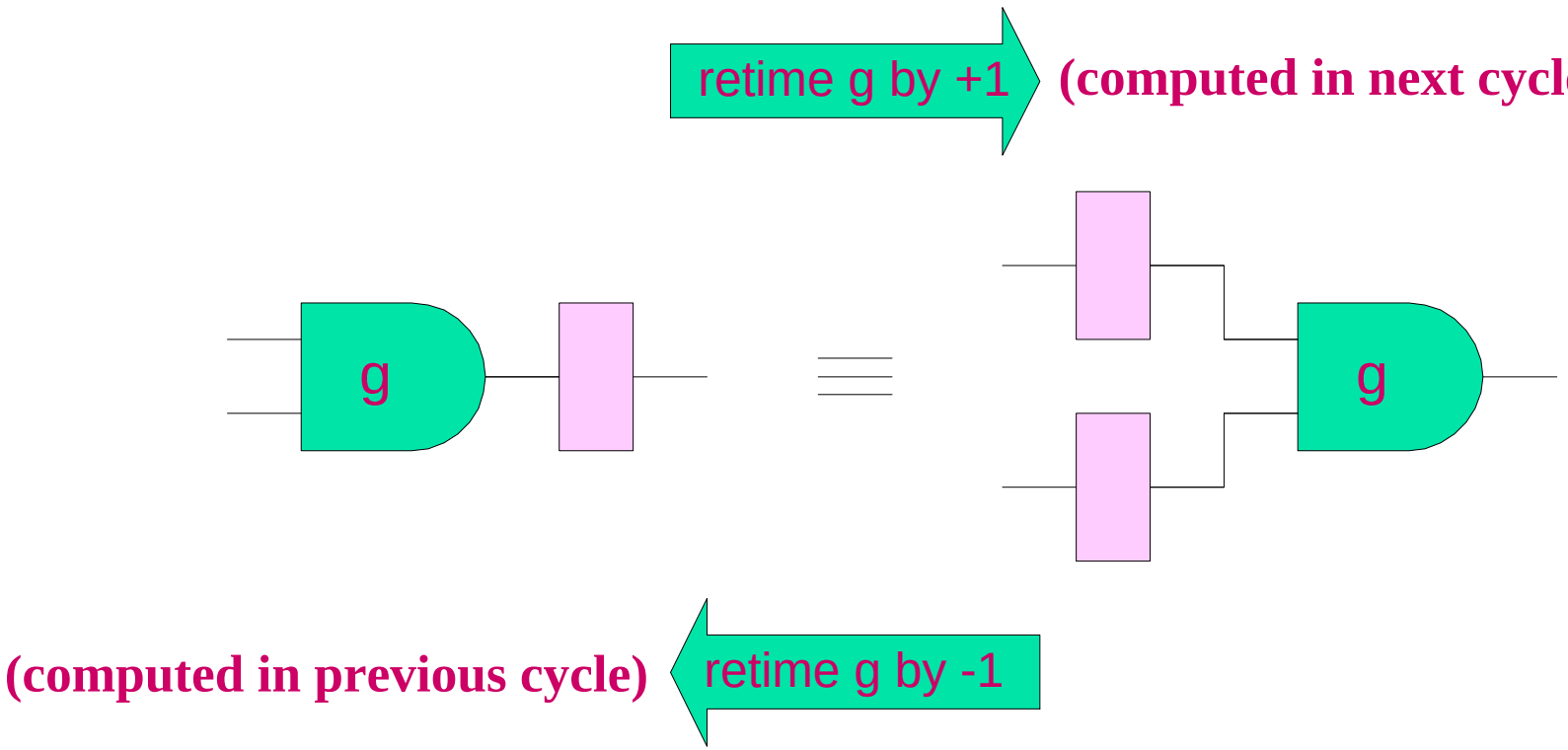
cycle time = 60



cycle time = 50

Moving Registers

- Combinational logic not modified



Formulation

- Directed graph:
  - Nodes: combinational logic
  - Edges: connections (possible latched) between logic
- Weights
  - Nodes: combinational logic propagation delay
  - Edges: number of registers
- Path delay $d(P)$: sum of node delays along a path
- Path weight $w(P)$: sum of edge weights along a path
- Clock period: $\Phi(G) = \max\{d(p) \mid w(p) = 0\}$

Some Definitions

- $W(u,v)$ is defined as the minimum number of registers on any path from vertex u to vertex v
- The critical path p is a path from u to v such that $w(p)=W(u,v)$
- $D(u,v)$ is defined as the maximum total propagation delay on any critical path from u to v

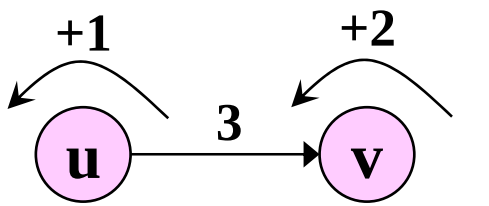
Synchronous Circuits

A synchronous circuit must satisfy following rules:

- D1: The propagation delay $d(v)$ is non-negative for each vertex v
  - Infeasible in real cases
- W1: The register count $w(e)$ is non-negative for each edge e
  - Infeasible in real cases
- W2: In any directed cycle, there is some edge with positive register count
  - No combinational loops

Retiming: Formulation

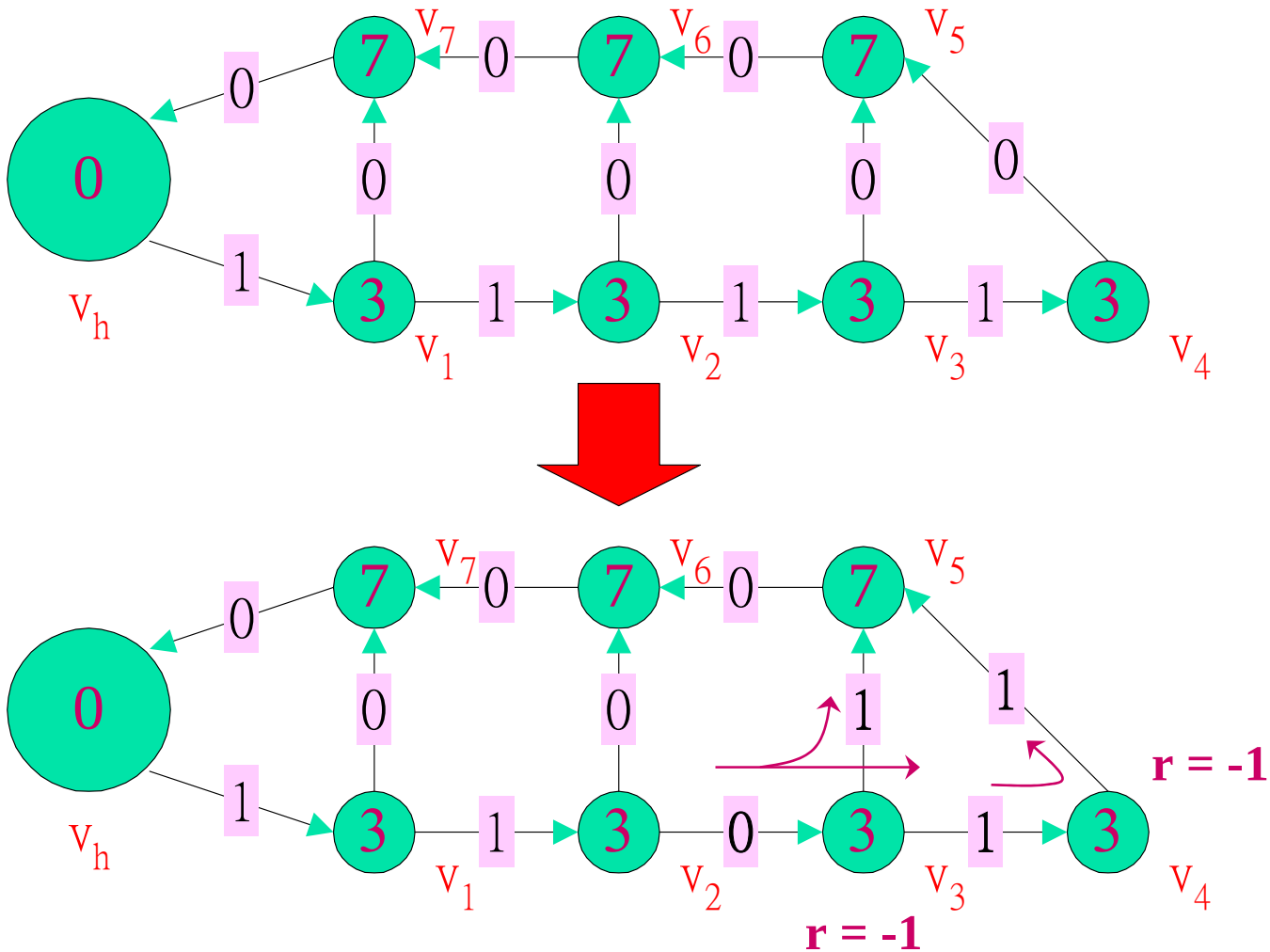
- Assign an integer-valued labeling r to each vertex
  - $w_r(u,v)=w(u,v)+r(v)-r(u)$
  - $w_r(p)=w(p)+r(v)-r(u)$



$W_r(u,v) = 3 + 2 - 1 = 4$

- Corollary: For any cycle p, $w_r(p)=w(p)$
- Legal retiming needs only being checked against condition W1: non-negative edge weight
- Corollary: Let G be a synchronous circuit and r be a retiming on G. Then the retimed graph G_r satisfies condition W2

Relocating Registers



Optimal Retiming (1/3)

- Problem: Given a graph G, find a legal retiming r of G such that the clock period $\Phi(G_r)$ of the retimed circuit G_r is as small as possible.
- Lemma: Let G be a synchronous circuit, and let c be any positive real number, the following are equivlent:
  1. $\Phi(G) \leq c$
  2. For all vertices u and v, if $D(u,v) > c$, then $W(u,v) \geq 1$
- Lemma:
  - A path p is a critical path of $G_r \Leftrightarrow$ it is a critical path of G
  - $W_r(u,v)=W(u,v)+r(v)-r(u)$
  - $D_r(u,v)=D(u,v)$
- Corollary: $\Phi(G_r)=D(u,v)$ for some u,v

Optimal Retiming (2/3)

- Theorem: r is a legal retiming of G such that $\Phi(G_r) \leq c$ if and only if
  1. $r(v_h)=0$
  2. $r(u)-r(v) \leq w(e)$ for every edge $e(u,v)$

     -- keep the register count non-negative
  3. $r(u)-r(v) \leq W(u,v)-1$ for every vertices u and v such that $D(u,v) > c$

     -- pipeline the long path (register count > 1)
- Solve the integer linear programming problem
  - Bellman-Ford method in $O(|V|^3)$
- The set of r's determine new positions of the registers

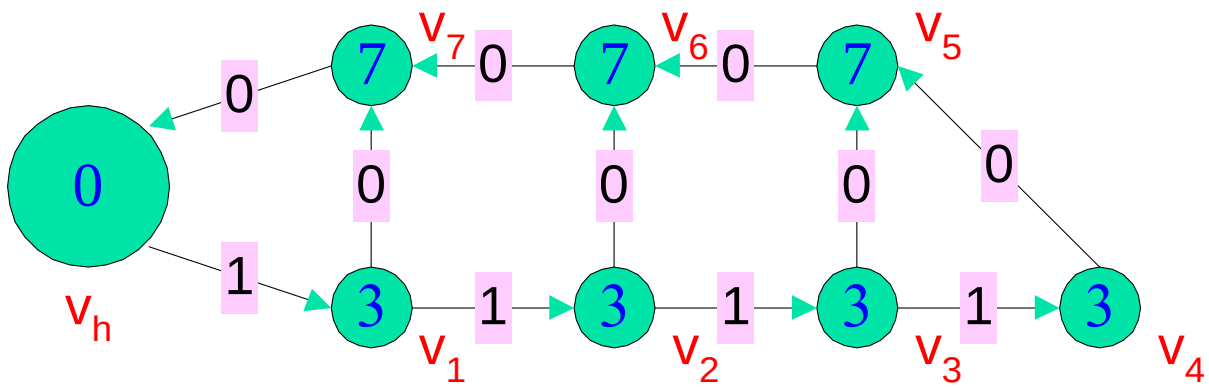
Optimal Retiming (3/3)

- Algorithm of optimal retiming:
  1. Compute W and D
  2. Sort the elements in the range of D
  3. Binary search the minimum achievable clock period by applying Bellman-Ford algorithm to check the satisfication of the Theorem
  4. Derive the $r(v)$ from the minimum achievable clock period found in Step 3
- Complexity $O(|V|^3 \lg|V|)$

All-Pair Shortest-Paths

- *W* and *D* can be computed by solving the ***all-pair shortest-paths*** problem
  - Floyd-Warshall method: $O(|V|^3)$
  - Johnson's method: $O(|V| |E| lg |V|)$
- Algorithm WD:
  1. Weight each edge $e(u,v)$ with the ordered pair $(w(e),-d(u))$
  2. Solve the all-pair shortest-paths problem
     - Add two weights by component-wise addition
     - Compares weights using lexicographic ordering
  3. Each shortest-path weight (x,y) between vertices u and v
     - $W(u,v)=x$
     - $D(u,v)=d(v)-y$

Examples: W and D Matrixes



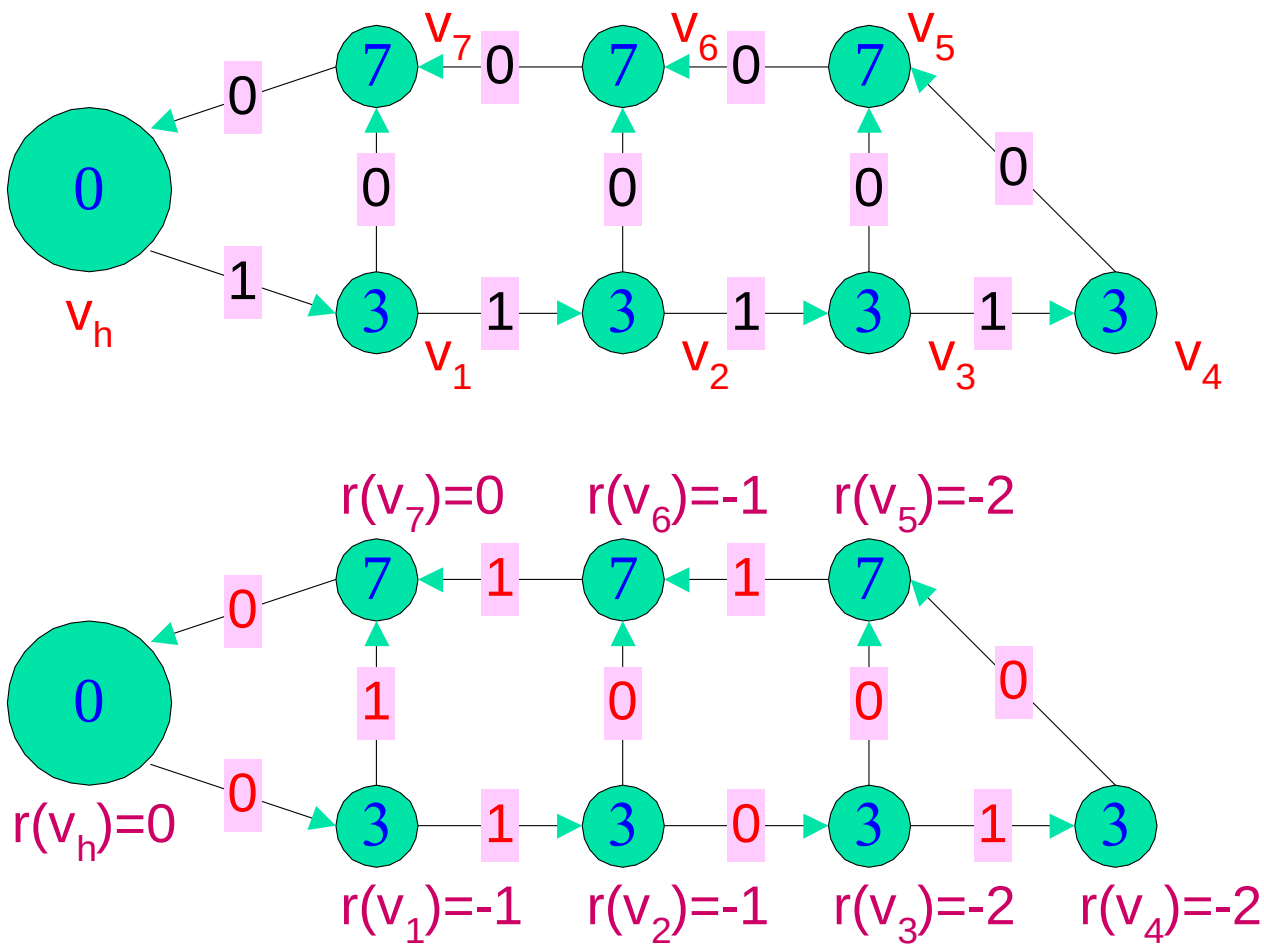
W	v_h	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_h	0	1	2	3	4	3	2	1
v_1	0	0	1	2	3	2	1	0
v_2	0	1	0	1	2	1	0	0
v_3	0	1	2	0	1	0	0	0
v_4	0	1	2	3	0	0	0	0
v_5	0	1	2	3	4	0	0	0
v_6	0	1	2	3	4	3	0	0
v_7	0	1	2	3	4	3	2	0

(source)

D	v_h	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_h	0	3	6	9	12	16	13	10
v_1	10	3	6	9	12	16	13	10
v_2	17	20	3	6	9	13	10	17
v_3	24	27	30	3	6	10	17	24
v_4	24	27	30	33	3	10	17	24
v_5	21	24	27	30	33	7	14	21
v_6	14	17	20	23	26	30	7	14
v_7	7	10	13	16	19	23	20	7

(destination)

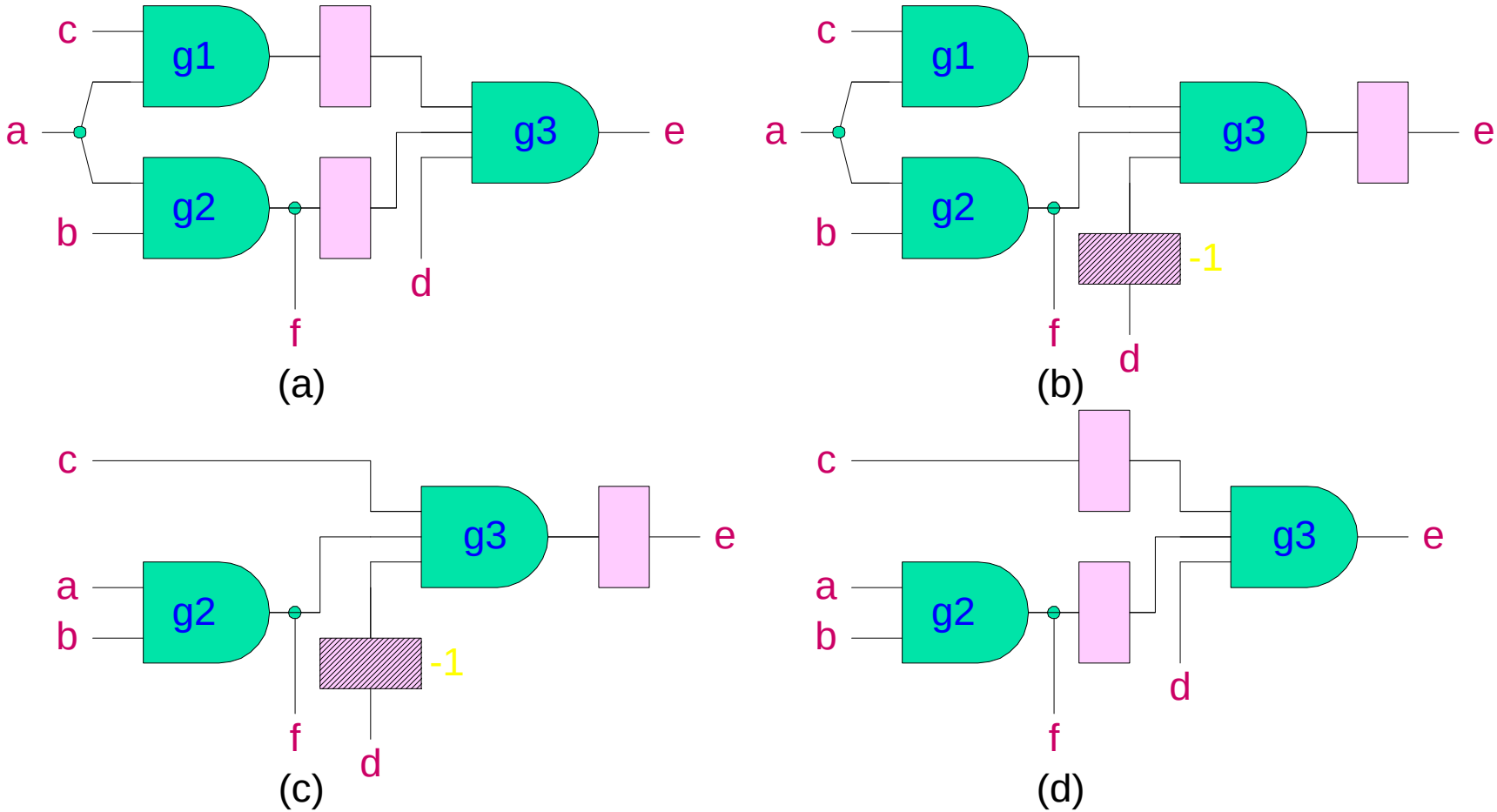
Retimed Correlator



Retiming and Resynthesis

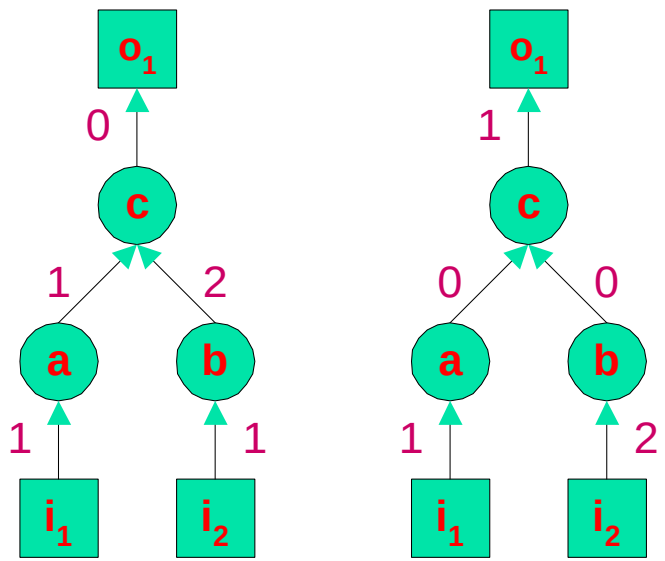
- Migrate all registers to the periphery of a sub-network
  - Peripheral retiming
- Optimize the sub-network with any combinational technique
  - Resynthesis
- Replace registers back in the sub-network
  - Retiming
- This procedure may further improve the timing across the registers

Examples of Resynthesis



Peripheral Retiming

- A peripheral retiming is a retiming such that
  - $r(v)=0$ where v is an I/O pin
  - $w(u,v)+r(v)-r(u)=0$ where $e(u,v)$ in an internal edge
- Move all registers to the peripheral edges
- Leave a purely combinational logic block between two set of registers
- Example:



Conditions for Peripheral Retiming

- No two paths between any input *i* and any output *j* have different edge weights
- Exist α_i and β_j , $1 \le i \le m$, $1 \le j \le n$ such that $W_{i,j} = \alpha_i + \beta_j$ (m: no. of inputs; n: no. of outputs)
- $W_{i,j} = \sum_{path\ i_i \rightarrow o_j} w(e)$ if all paths between input *i* and output *j* have the same weight
- Complexity $O(e \cdot \min(m,n))$

Examples of Peripheral Retiming

- Example 1:

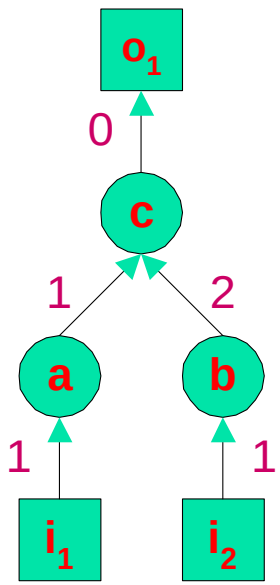
  $W_{1,1}=2,\ W_{2,1}=3,$

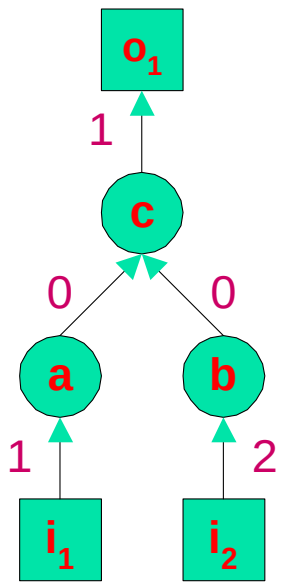
  $\Rightarrow \alpha_1=1,\ \alpha_2=2,\ \beta_1=1$

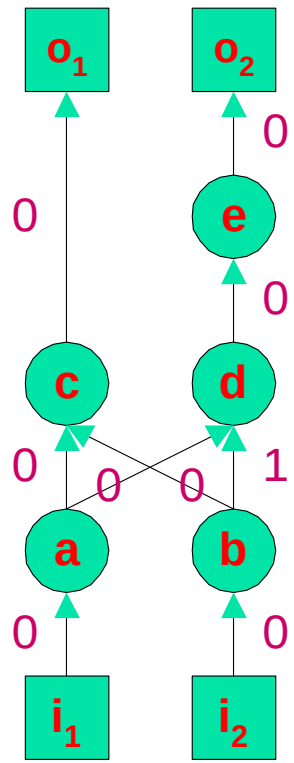
- Example 2:

  $W_{1,1}=0, W_{1,2}=0, W_{2,1}=0,\ W_{2,2}=1$

  $\Rightarrow$ no solution

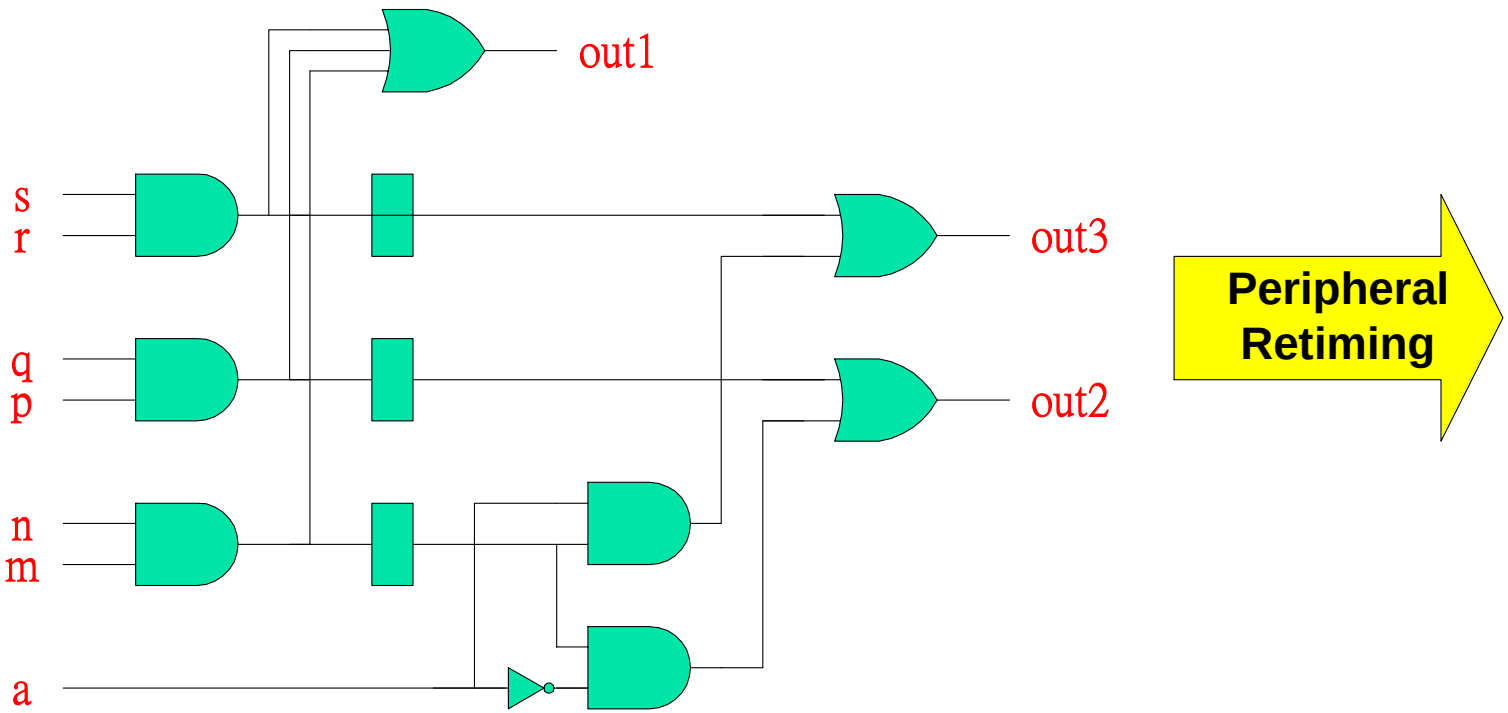




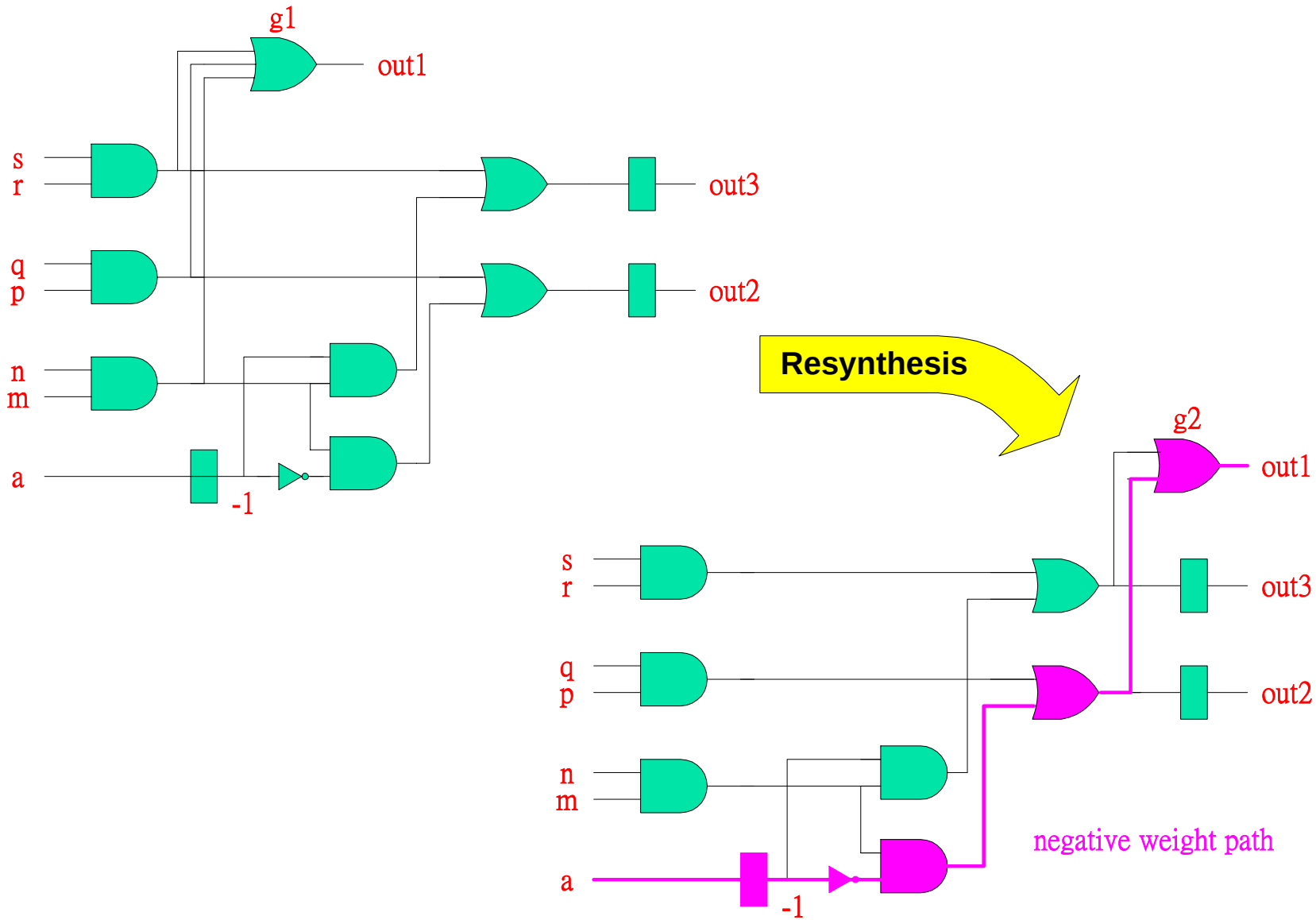


Legal Resynthesis Operations (1/2)

- Any that do not create a path with negative weight
- Resynthesis could create pseudo-dependency between any input and output
- Example:



Legal Resynthesis Operations (2/2)



Effects of Retiming and Resynthesis

- Area optimization:
  - No significant improvement
  - Limitation on existing combinational optimization techniques
  - Some circuits (pipelined datapaths) have inherently no potential for further optimization using retiming and resynthesis techniques
- Performance optimization of pipelined circuits:
  - Significant improvements for pipelined arithmetic circuits