Power Optimization

Rajaram Sivasubramanian
Associate Professor
ECE Department
Thiagarajar College of Engineering,Madurai-15

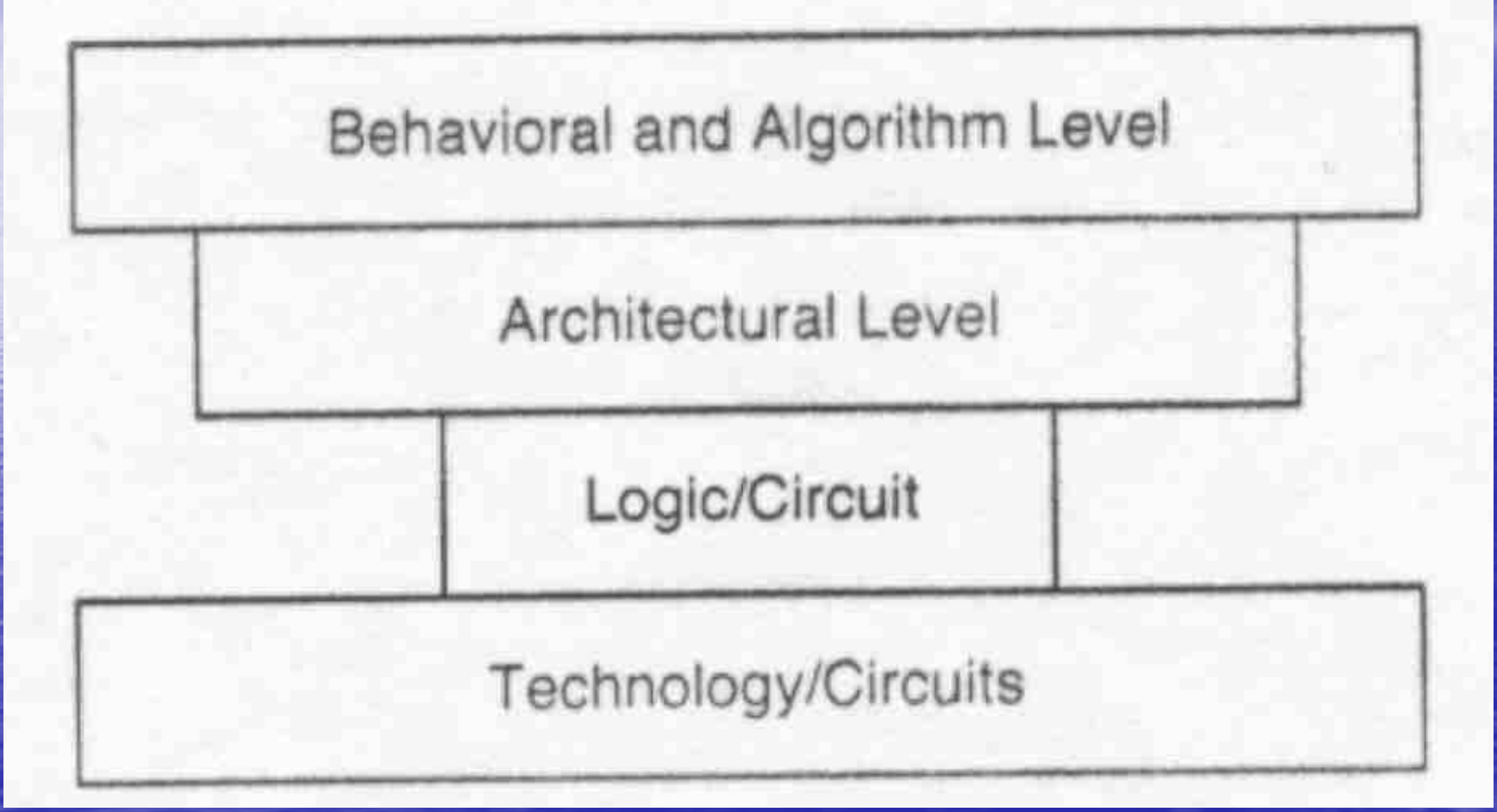
Courtesy : Prof .Bushnell

Motivation

- Conventional automated layout synthesis method:
  - ***Describe design at RTL or higher level***
  - ***Generate technology-independent realization***
  - ***Map logic-level circuit to technology library***
- Optimization goal: shifting from low-area to low-power and higher performance
- Need accurate signal probability/activity estimates
- Consider low-power needs at all design levels

Behavioral-Level Transformations

Algorithm-Level Power Reductions vs. Other Levels



Logic/Circuit Synthesis for Low-Power

Logic-Level Optimizations

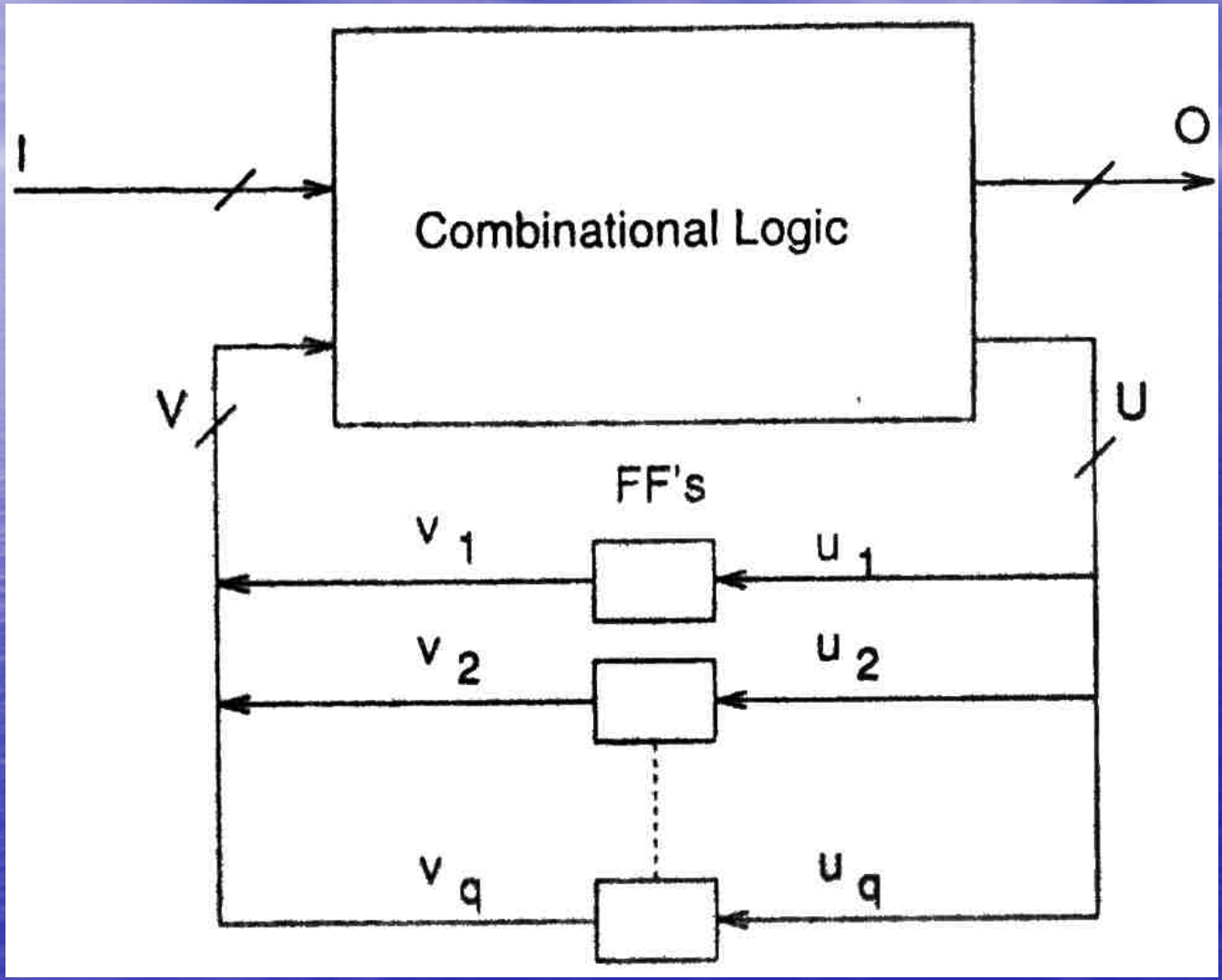
Design Flow

- Behavioral Synthesis – not used very much
- Initial design description: RTL or Logic level
- Logic synthesis widely used
- FSMs:
  - ***State Assignment – opportunity for power saving***
  - ***Logic Synthesis – look for common subfunctions – opportunity for power saving***
    - **Custom VLSI design – size transistors to optimize for power, area, and delay**
    - **Library-based design – technology mapping used to map design into library elements**

FSM and Combinational Logic Synthesis

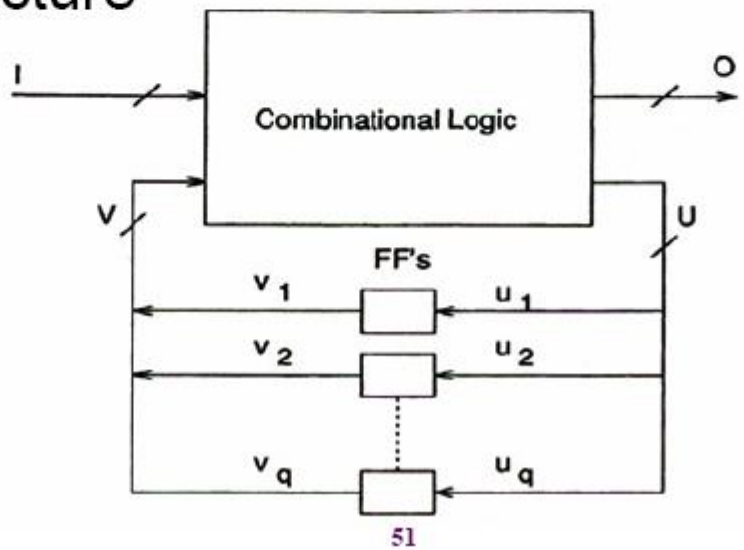
- Consider likelihood of state transitions during state assignment
  - ***Minimize # signal transitions on present state inputs V***
- Consider signal activity when selecting best common sub-expression to pull out during multi-level logic synthesis
  - ***Factor highest-activity common sub-expression out of all affected expressions***

Huffman FSM Representation



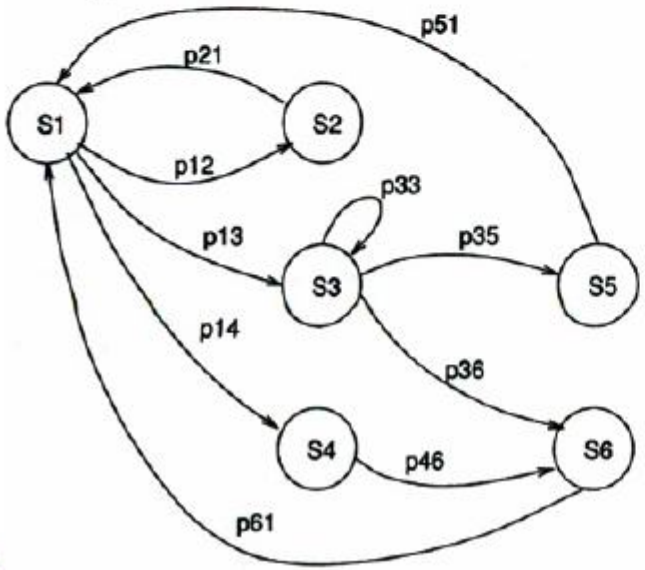
Logic level optimization for low power

- Logic synthesis
  - FSM synthesis
    - Minimize the switching during the state transition
  - Combinational synthesis
    - Power-conscious multi-level logic optimization
- General structure



Representation of FSM

- PSTG: Probabilistic STG
- P_{jm}: State transition probability from S_j to S_m
  - $\sum_m P_{jm} = 1$
- Example



For S_1: $p_{12} + p_{13} + p_{14} = 1$
For S_3: $p_{33} + p_{35} + p_{36} = 1$

Power metrics for FSM synthesis

- A measure for the switching between states
  - Hamming distance $H(S_i, S_j) = \bullet(S_i \text{ XOR } S_j)$
    - $\bullet(x)$: returns the # of 1s in x
- Power estimation
  - Average power

$$Power_{avg} = \frac{1}{2} V_{DD}^2 \sum C_i D(i)$$

  - Normalized power

$$\Phi = \sum_i fanout_i D(i)$$

State assignment for FSM (I)

- Use PSTG shown in the previous slide
- L_{ij}: the label associated with the edge from S_i to S_j
- P_x: signal probability of input x
- value(x): logic value of x from the set {1, 0, -}
- P_{ij} is

$$p_{ij} = \prod_{x \in inputs\_of\_L_{ij}} W_x$$

$$W_x = P_x \quad when \quad value(x) = 1$$
$$= 1 - P_x \; when \quad value(x) = 1$$
$$= 1 \quad when \quad value(x) = 1$$

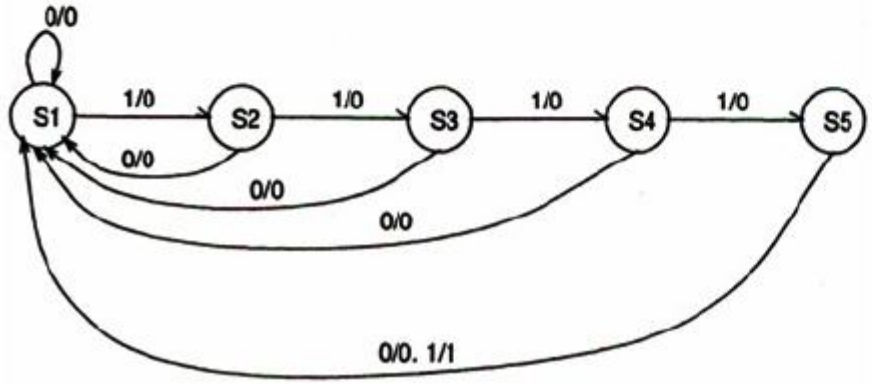
State assignment for FSM (II)

- Objective function
  - Minimize the following cost function
    - $\gamma = \bullet\ \bullet_{\text{over all edges}} p_{ij}H(S_i, S_j)$
  - Oneway to solve the problem: Simulated anealing
    - iterative method
    - For each iteration, either interchage two codes of assign a new code to a state
    - If γ is decreased ➔ accecpted
    - If γ is decreased ➔ accepcted with the probability of $e^{-|\sigma(\gamma)|/\text{Temp}}$
    - Temp is an anealing temperature
    - Terminated when Temperature is under the given value

unassigned code to the state that is randomly picked for exchange. The move is accepted if the new assignment decreases γ. If the move increases the value of the objective function γ, the move is accepted with a probability of $e^{-|\delta(\gamma)|/\text{Temp}}$, where $|\delta(\gamma)|$ is the absolute value of the change in the objective

Example of state assignment for FSM

- A state machine making output as "1" when it observes the five consecutive "1"s from the input
- The signal probability of each input = 0.5
  - Thus, the state transition probability of each edge = 0.5
- Coding 1 ➔ γ = 10, Coding 2 ➔ • •= 5.5
- Same HW cost
- From SPICE simulation Coding 2 consumes 15% of less power

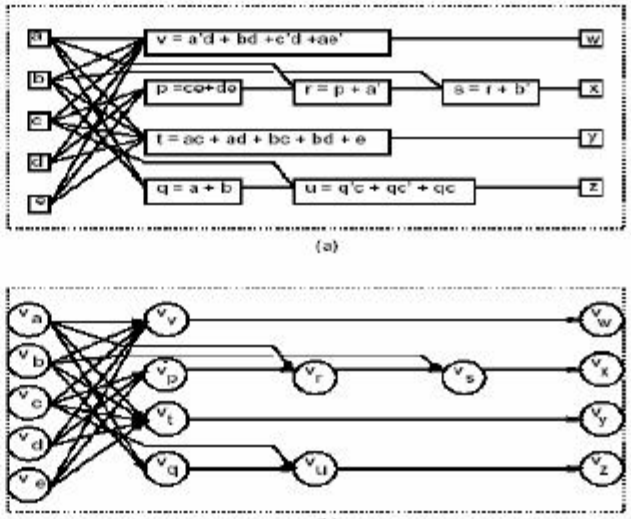


	Coding 1	Coding 2
S1	010	000
S2	101	100
S3	000	111
S4	111	010
S5	100	011

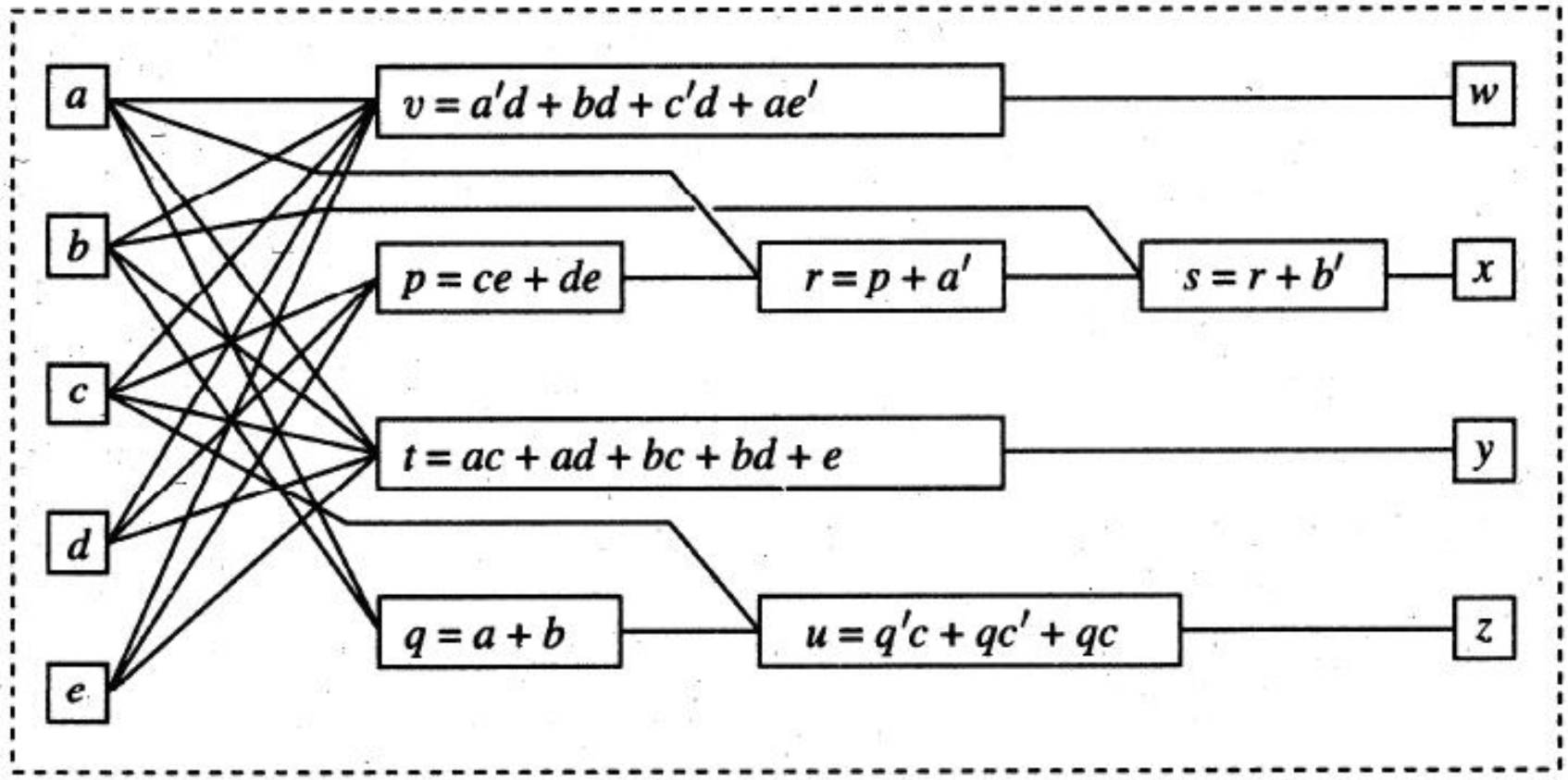
Power-conscious multi-level logic opt. (I)

- Before talking about power…
- Multi-level vs. two-level
  - Multi-level: the path from input to output may have gates more than two ➔ Area / Delay efficient
  - Two-level: AND plane + OR plane ➔ Good for PLA
- Build a logic network for the given input boolean equations

$$
\begin{aligned}
p &= ce + de \\
q &= a + b \\
r &= p + a' \\
s &= r + b' \\
t &= ac + ad + bc + bd + e \\
u &= q'c + qc' + qc \\
v &= a'd + bd + c'd + ae' \\
w &= v \\
x &= s \\
y &= t \\
z &= u
\end{aligned}
$$

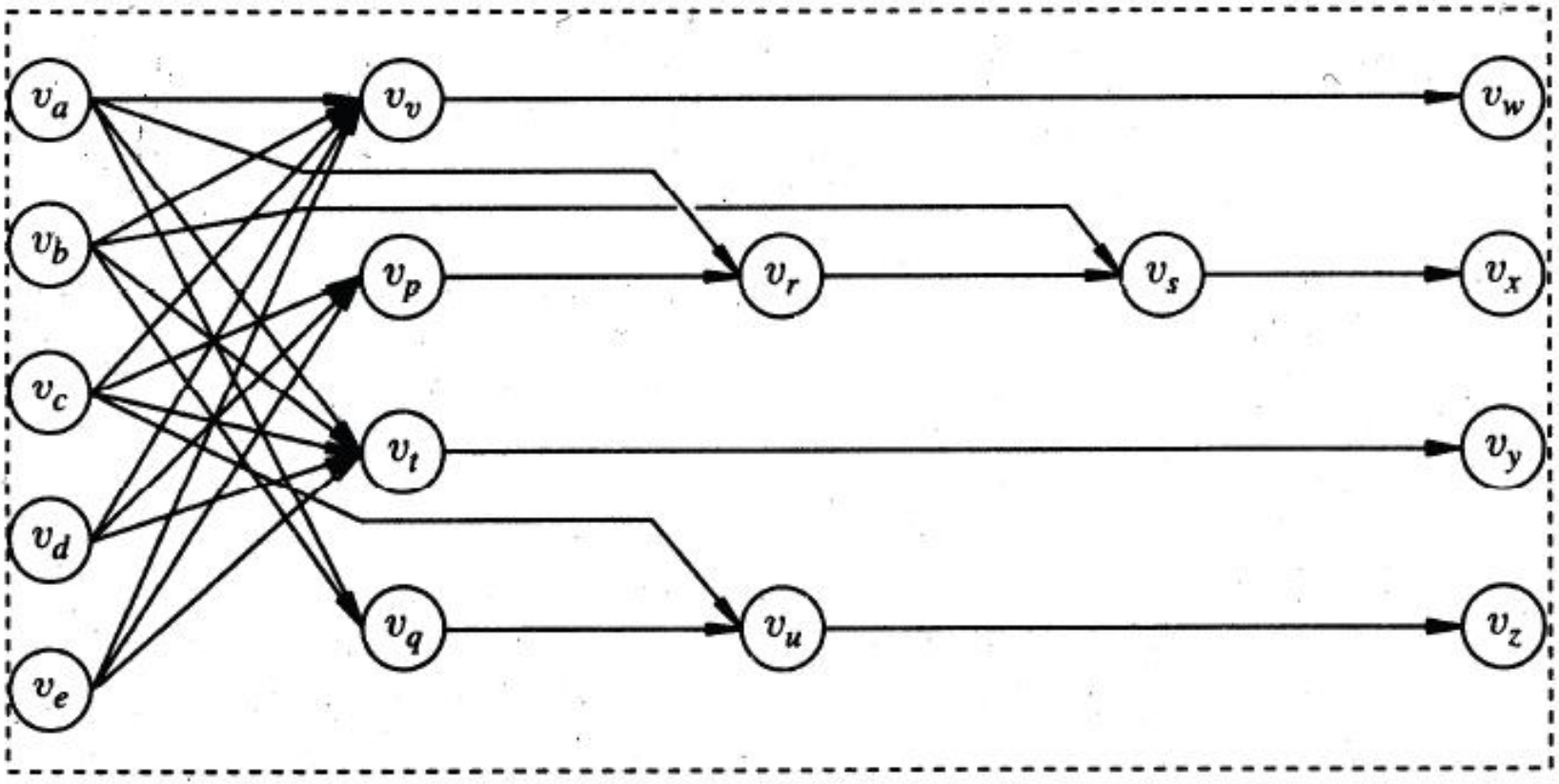


Basic Representation #1: Logic Network



$$
\begin{aligned}
p &= ce + de \\
q &= a + b \\
r &= p + a' \\
s &= r + b' \\
t &= ac + ad + bc + bd + e \\
u &= q'c + qc' + qc \\
v &= a'd + bd + c'd + ae' \\
w &= v \\
x &= s \\
y &= t \\
z &= u
\end{aligned}
$$

Basic Representation #2: Logic Network Graph



Comparison: Logic Network vs. Logic Network Graph

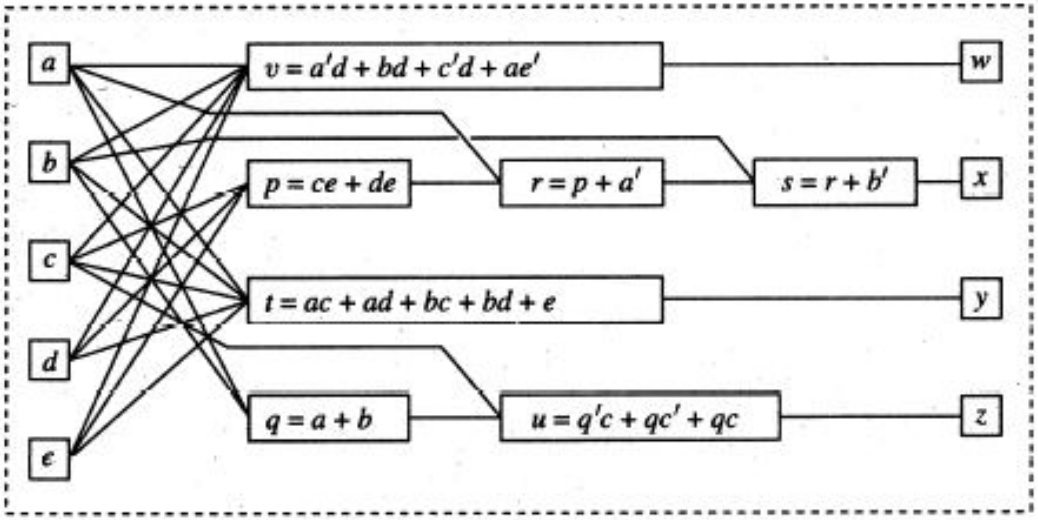
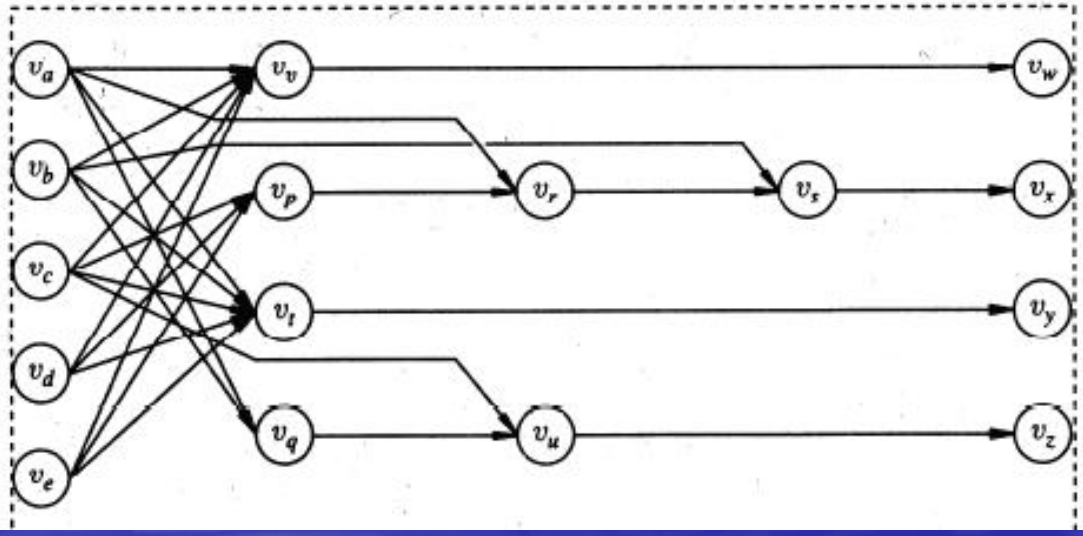


Fig. 8.3(a)

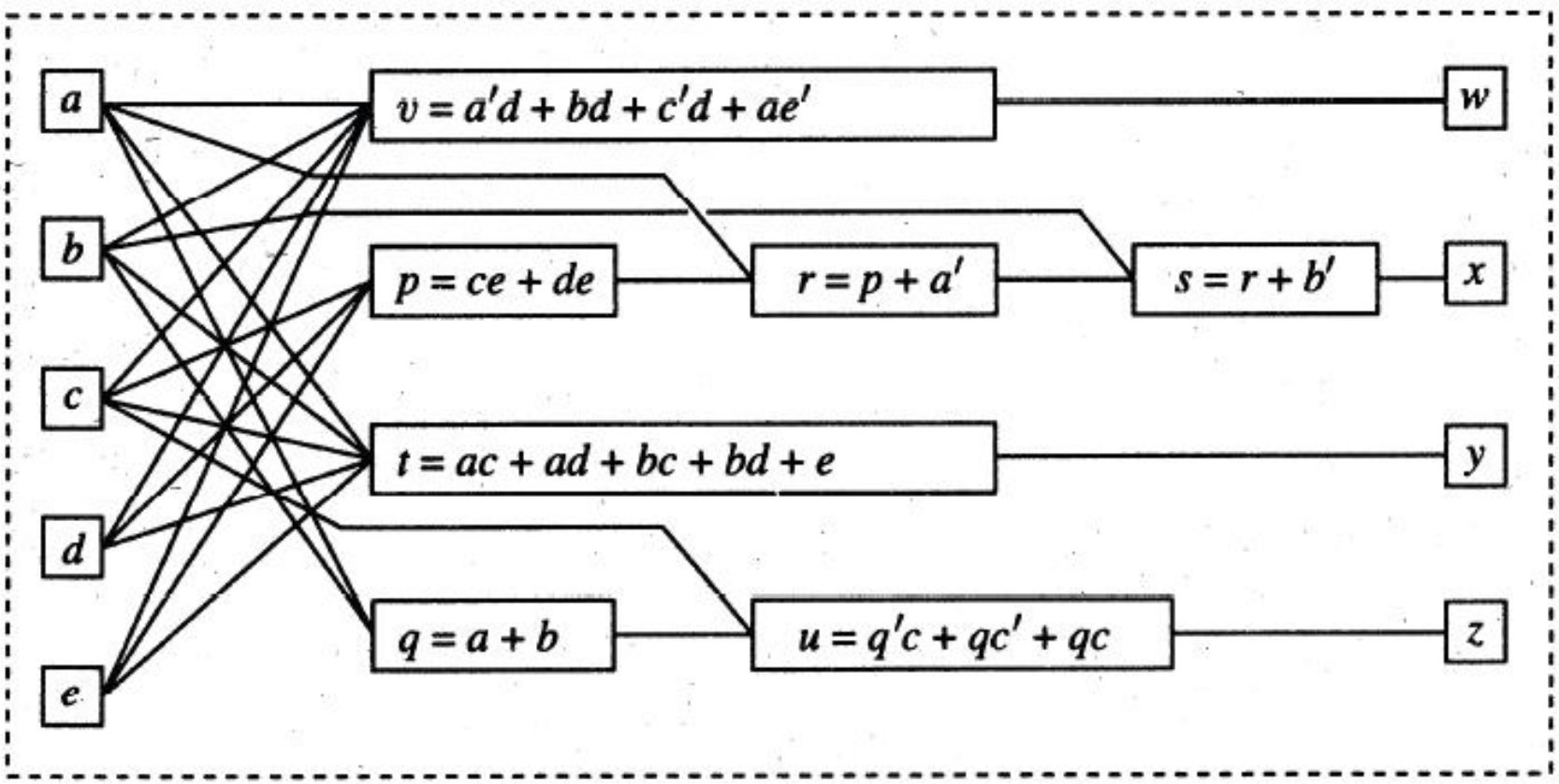
Logic network



Logic network graph

$$\mathbf{f} = \begin{bmatrix} a'd + bd + c'd + ae' \\ a' + b' + c + d \\ ac + ad + bc + bd + e \\ a + b + c \end{bmatrix}$$

Multi-Level Optimization Example: Initial Design



Transform #1: ELIMINATE (collapse)

Eliminate a node: "collapse" network structure

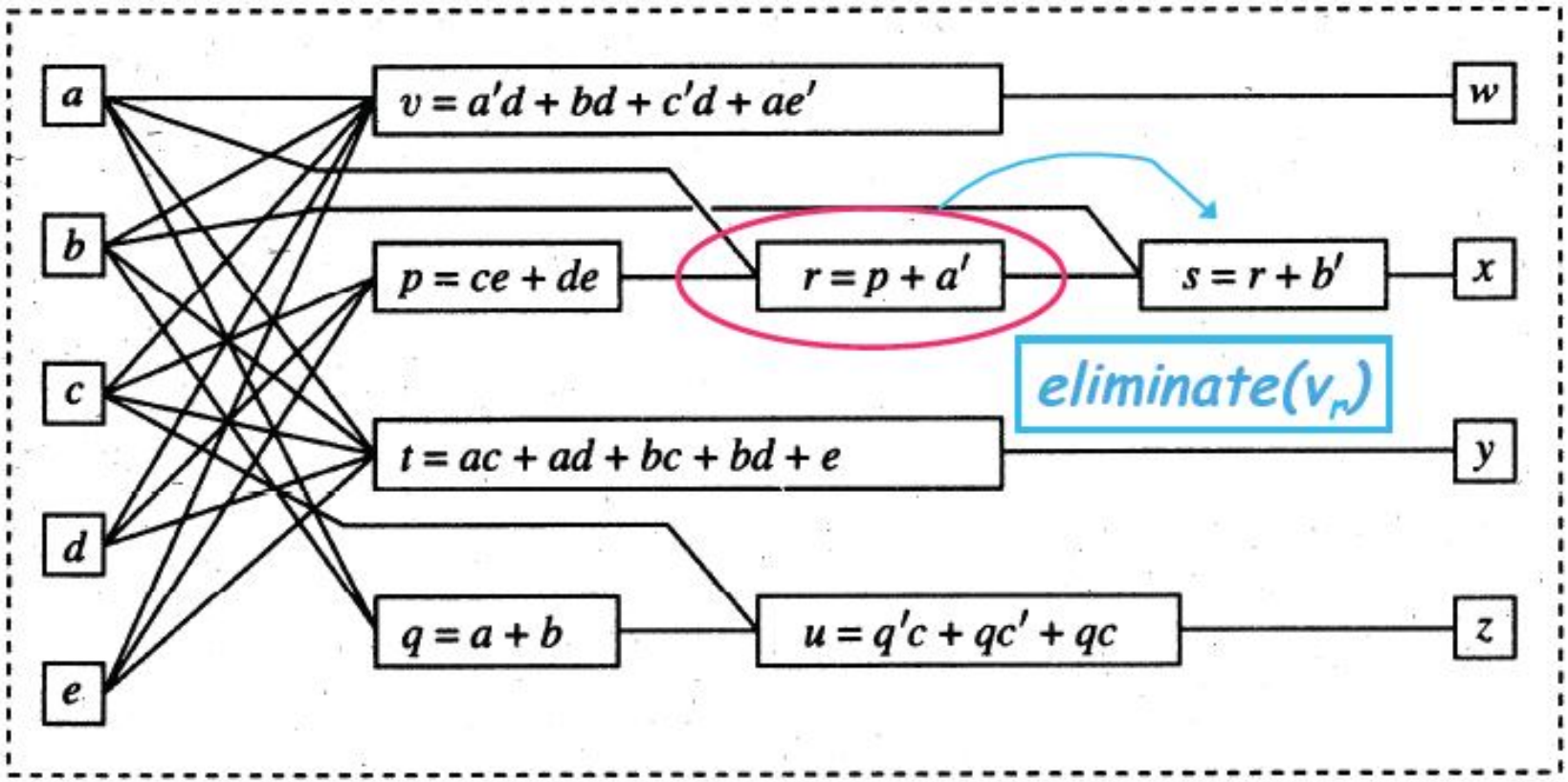
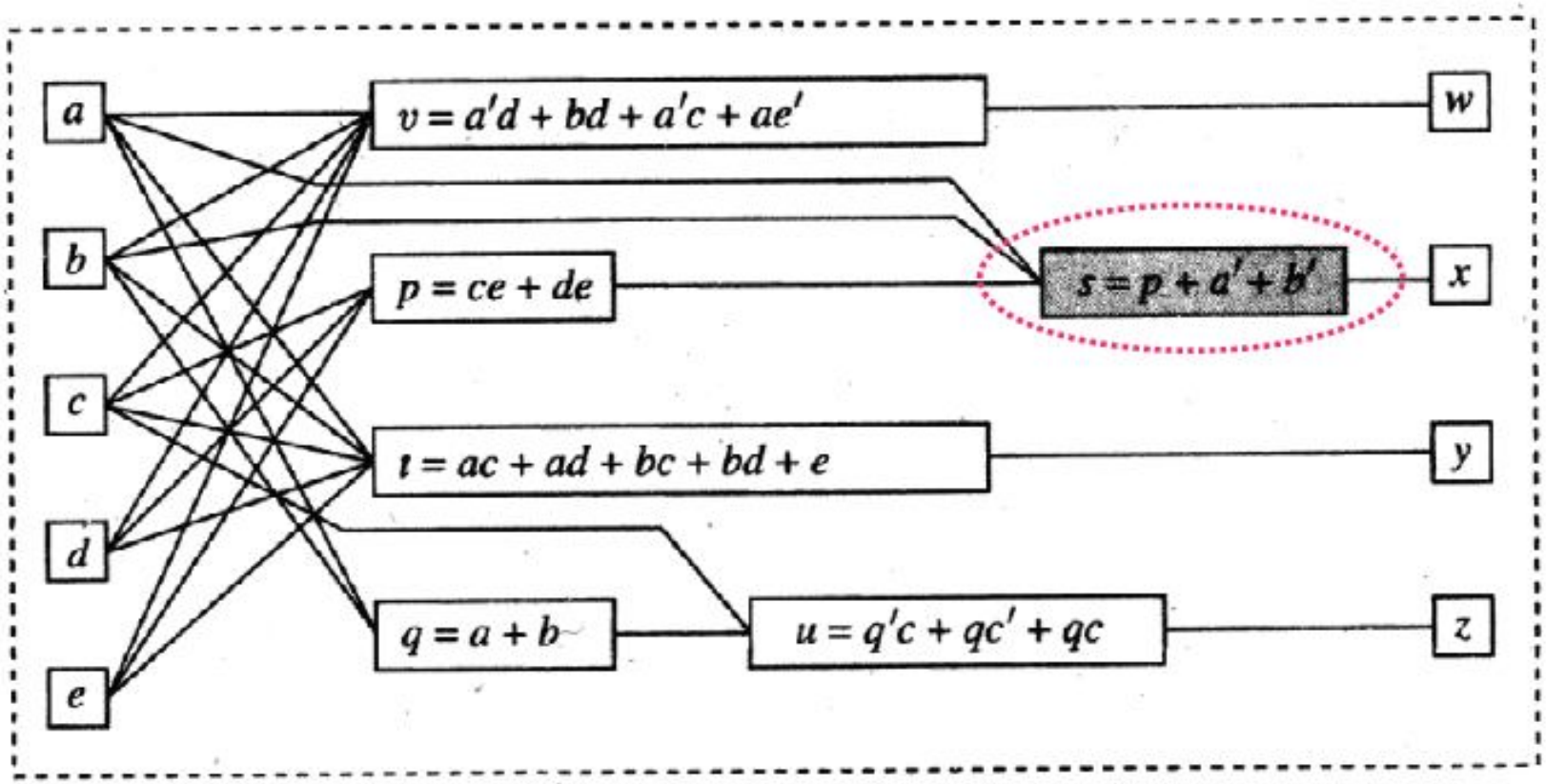


Fig. 8.3(a)

before

Transform #1: ELIMINATE (collapse)

Eliminate a node: "collapse" network structure



after

Transform #2: DECOMPOSE

Break 1 larger node into several smaller nodes

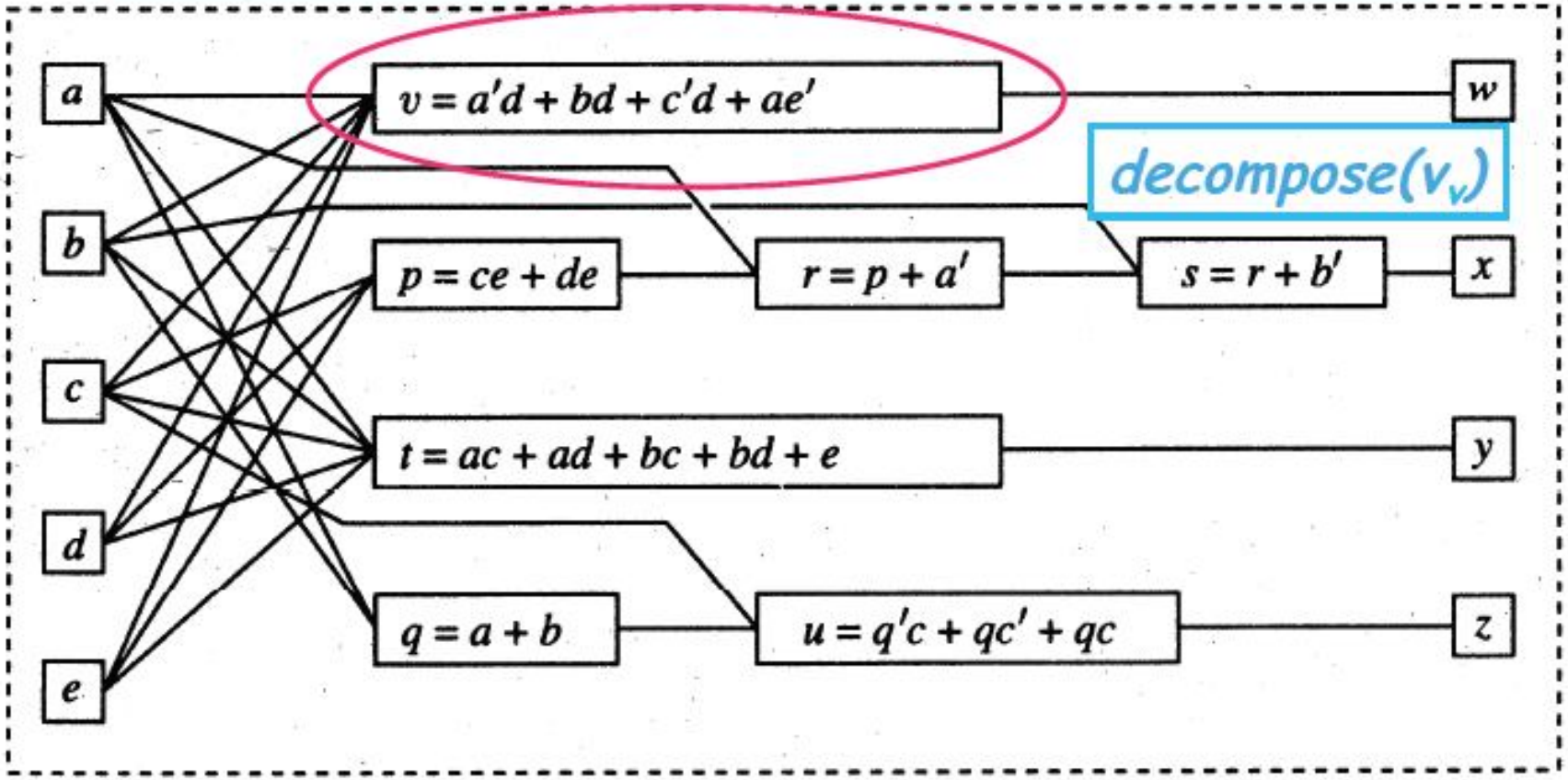
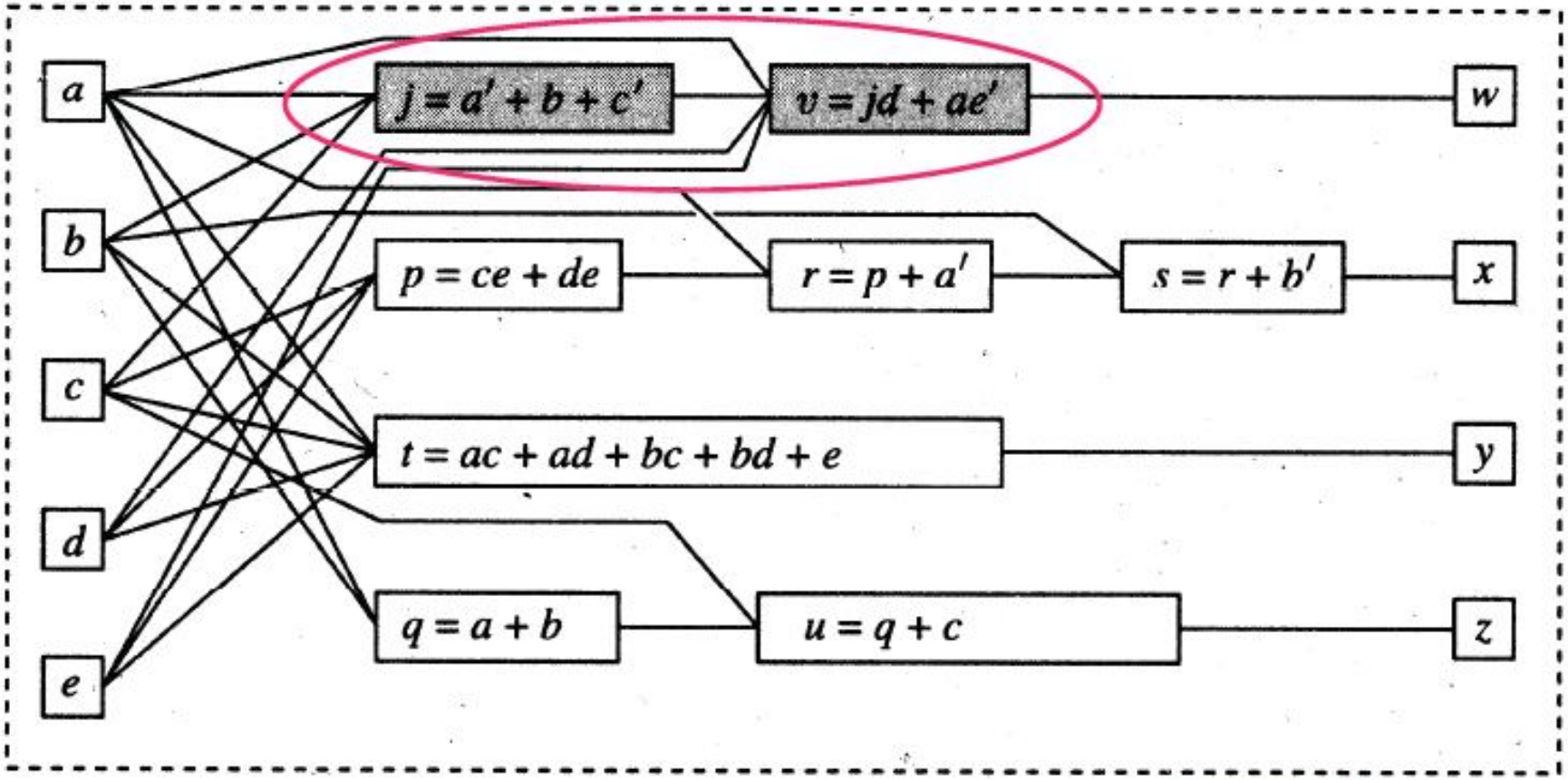


Fig. 8.3(a)

before

Transform #2: DECOMPOSE

Break 1 larger node into several smaller nodes



after

Transform #3: EXTRACTION

Create/extract "common subexpression" for 2 or more nodes

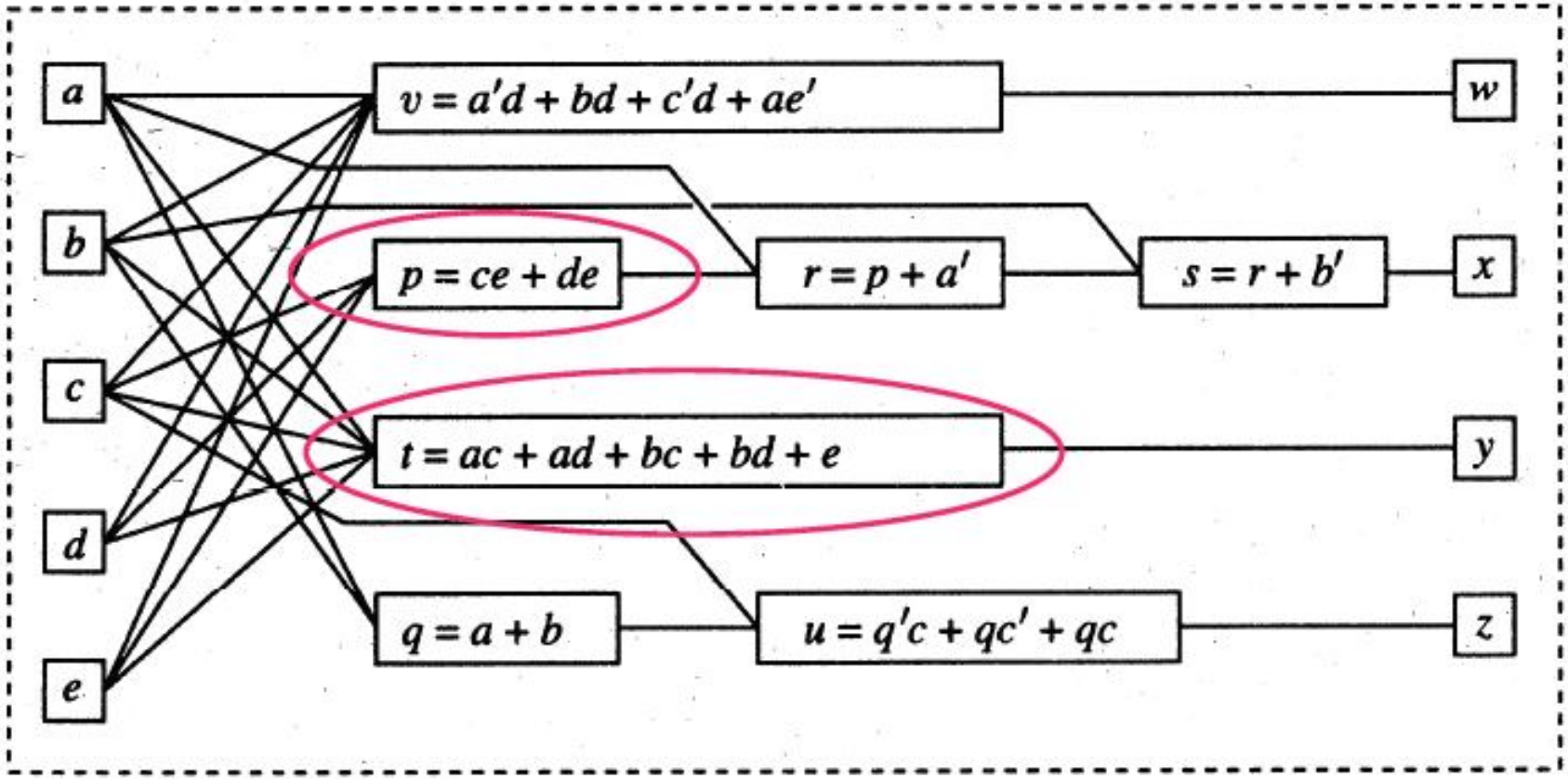
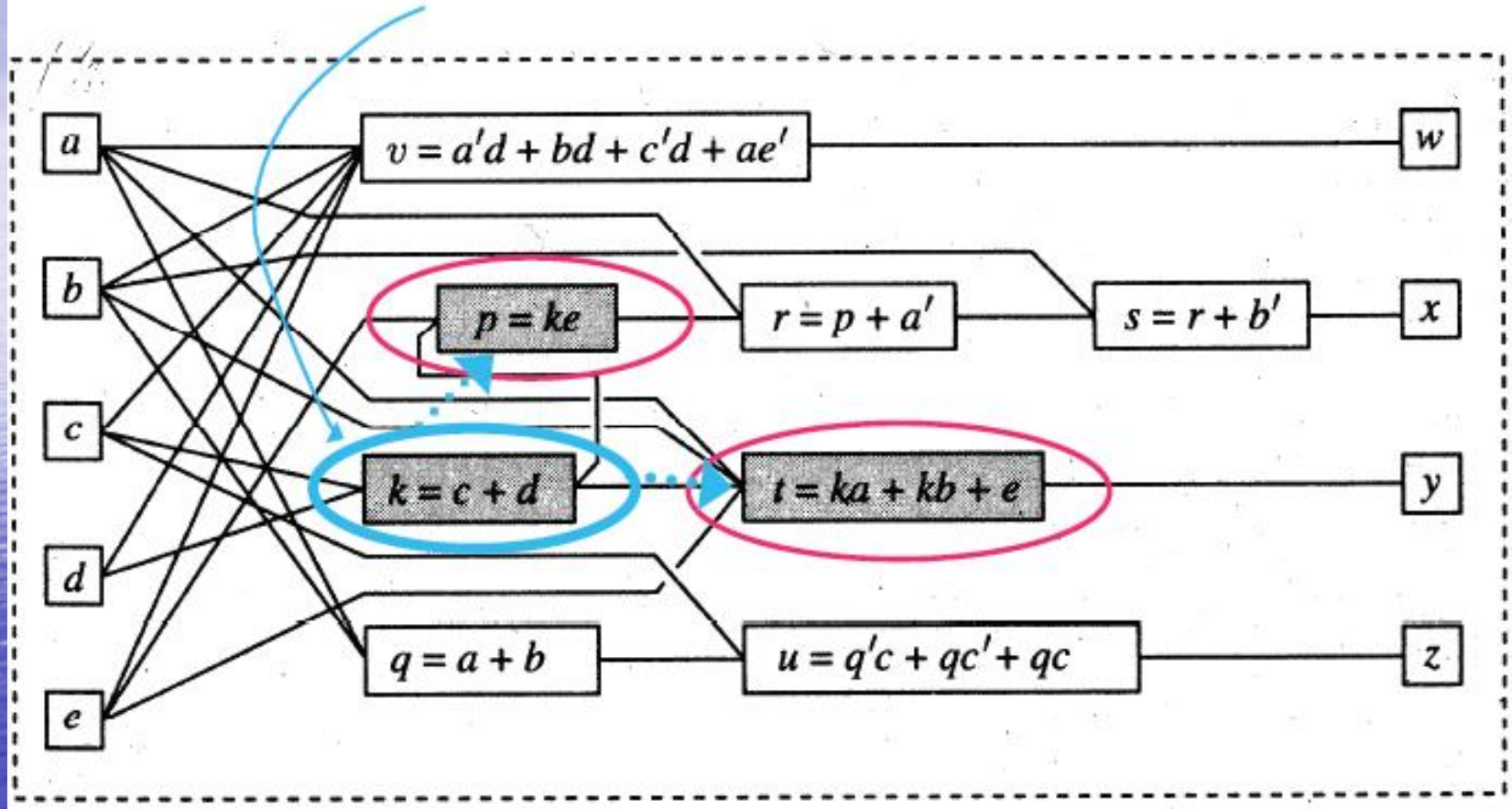


Fig. 8.3(a)

before

Transform #3: EXTRACTION

Create/extract "common subexpression" for 2 or more nodes



after

Transform #4: SIMPLIFICATION

Perform optimization (usually Boolean) within a single node

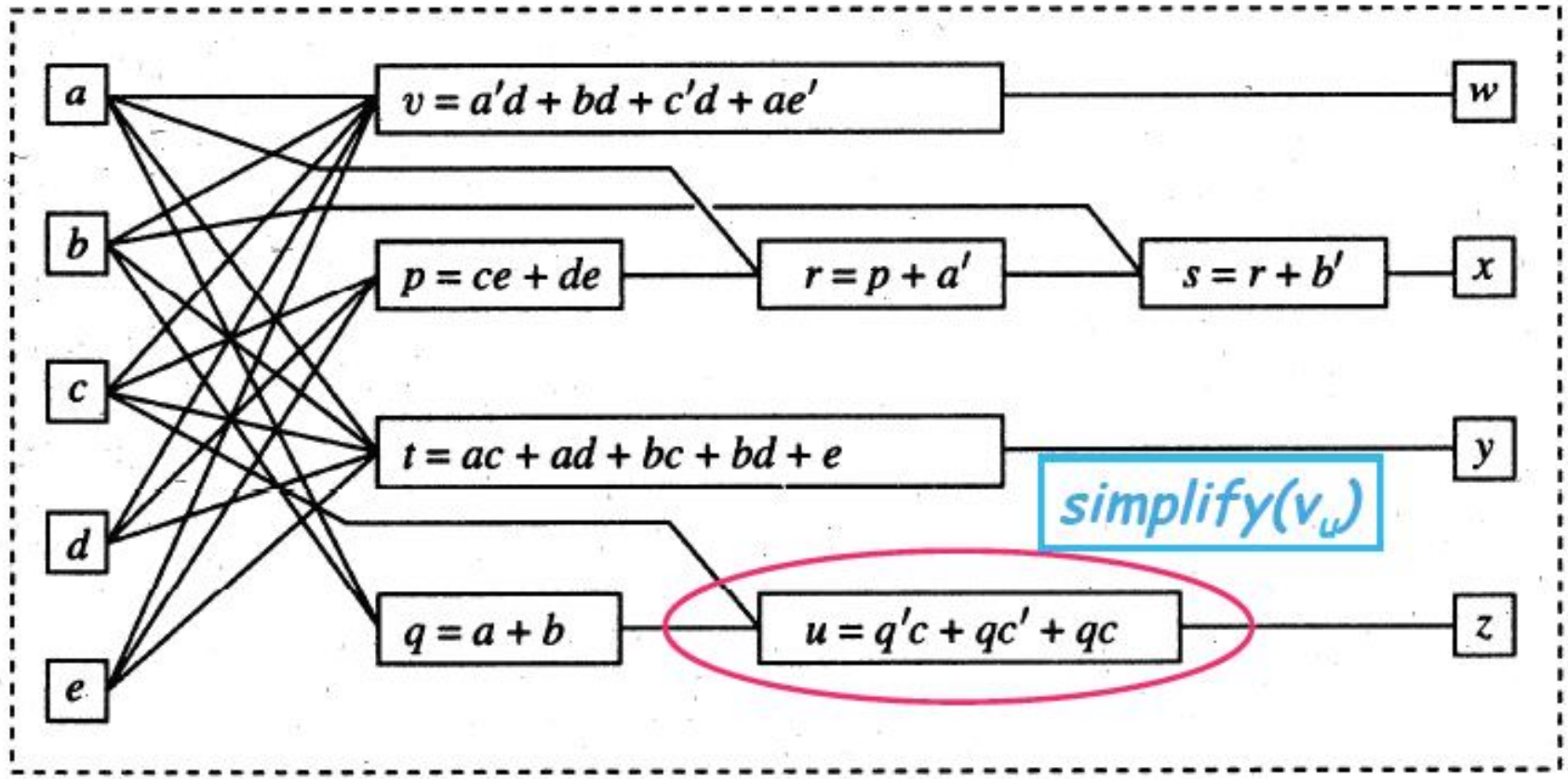
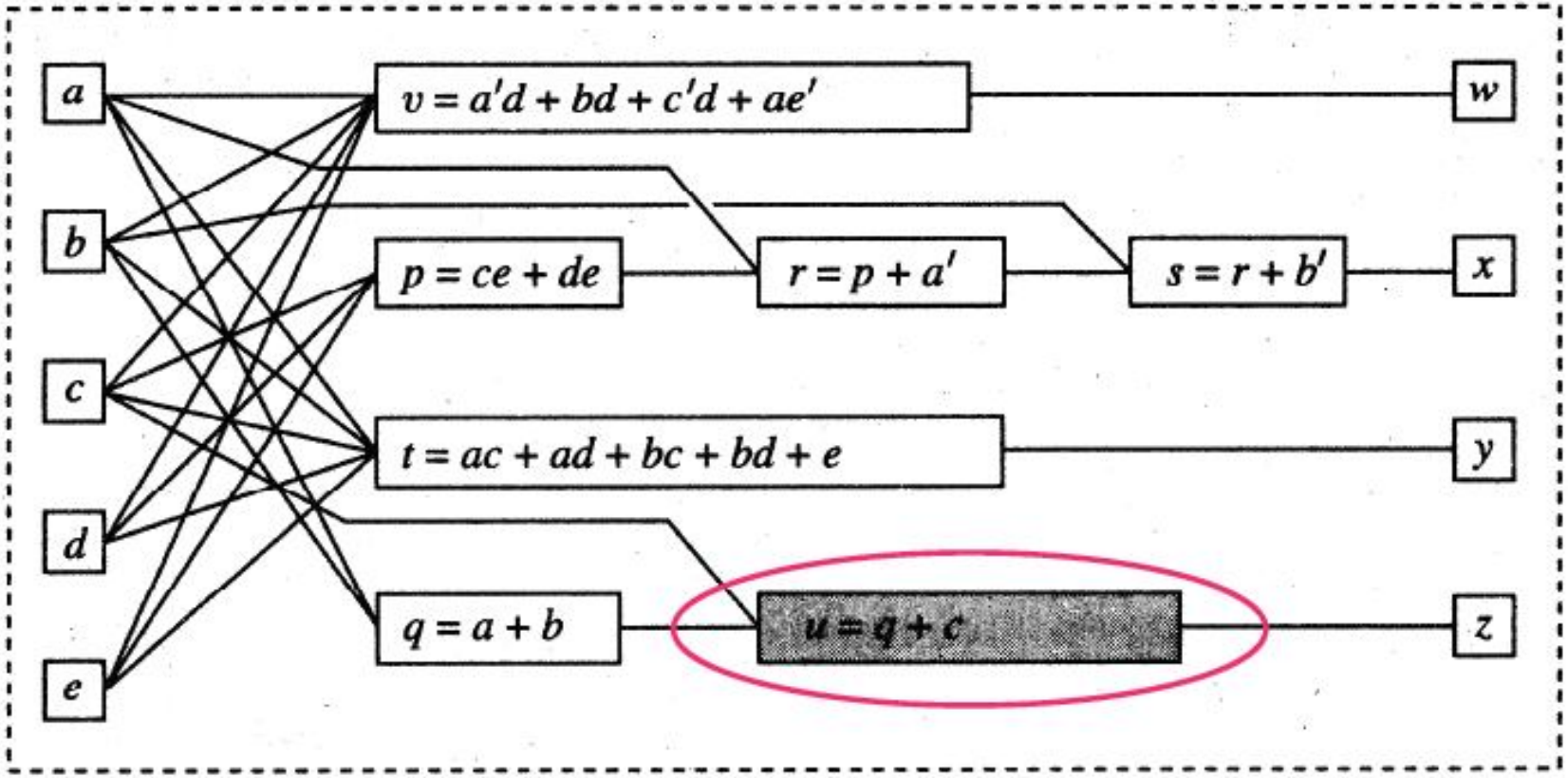


Fig. 8.3(a)

before

Transform #4: SIMPLIFICATION

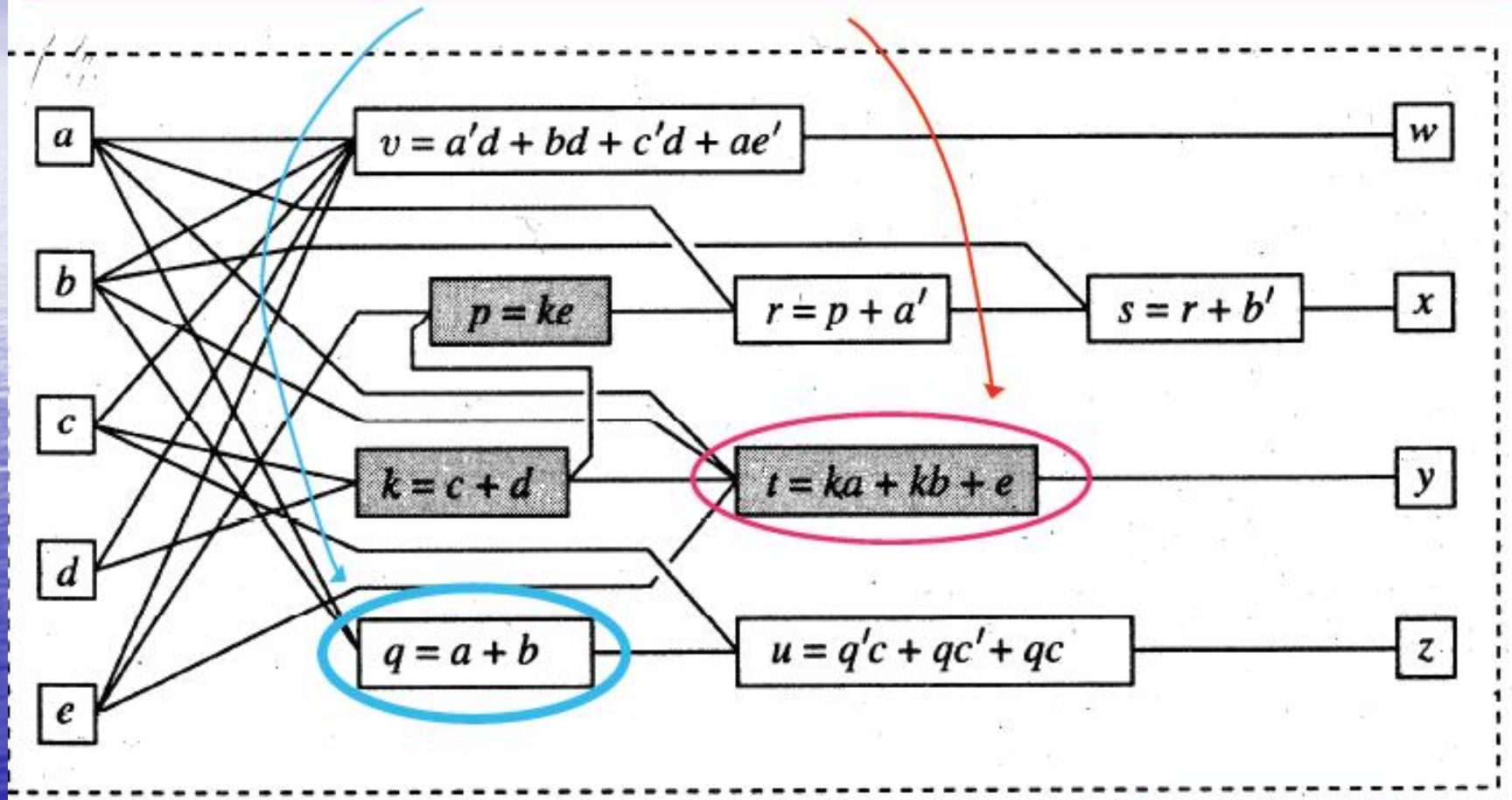
Perform optimization (usually Boolean) within a single node



after

Transform #5: SUBSTITUTION

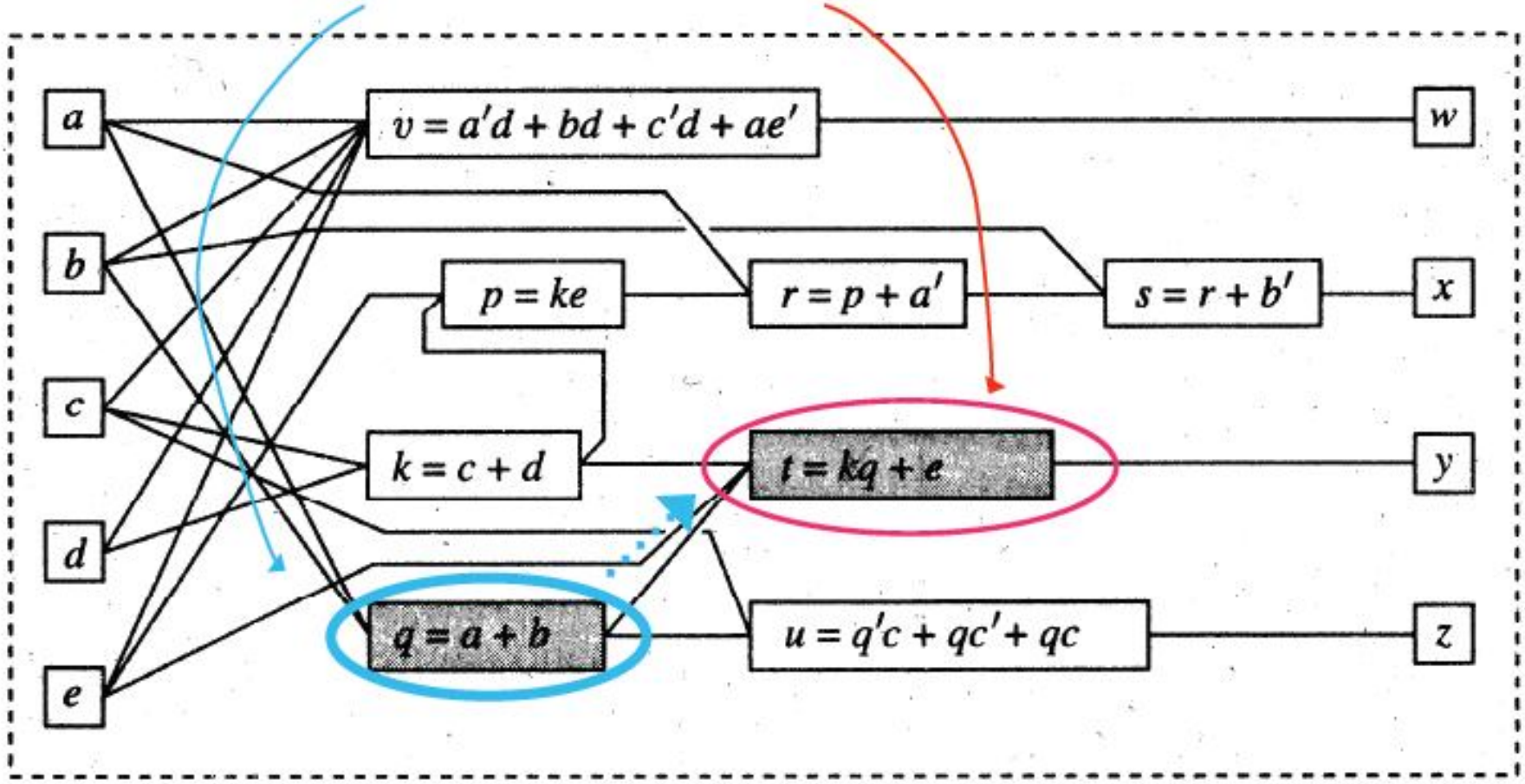
Find an existing "common subexpression" for 1 or more nodes



before

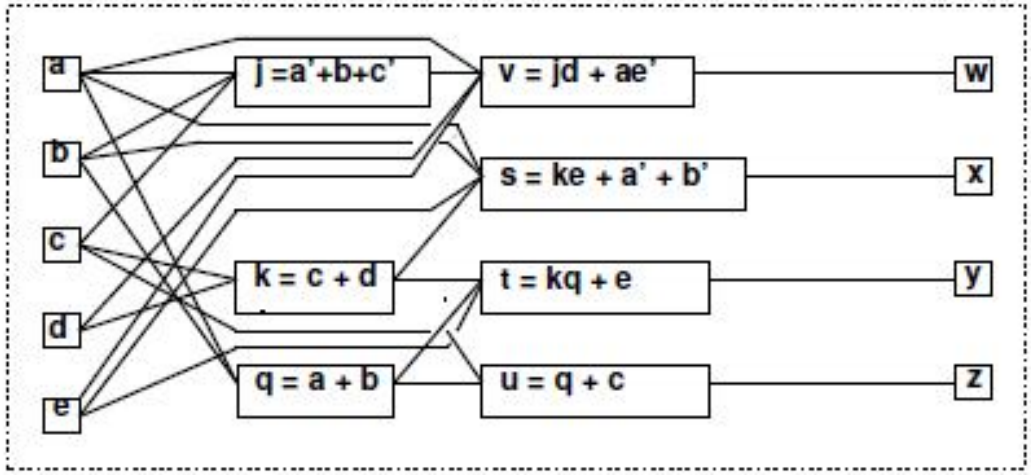
Transform #5: SUBSTITUTION

Find an existing "common subexpression" for 1 or more nodes



after

$$
\begin{aligned}
j &= a' + b + c' \\
k &= c + d \\
q &= a + b \\
s &= ke + a' + b' \\
t &= kq + e \\
u &= q + c \\
v &= jd + ae'
\end{aligned}
$$



Power-conscious multi-level logic opt. (II)

- Major procedures
  - Kernel
    - finds some or all cube-free multiple or single-cube divisors of each of the functions
    - retains those divisors that are common to two or more functions
    - To factor out the kernels, use algebraic division method
      - Quotient → Kernel
  - Substitution
    - simplify the network by using the best few common divisors factored out
    - repeated until no common divisors can be found

Introduction to our technique

- Terminology
  - ***Literal**: A variable or a constant eg. a,b,2,3.14*
  - ***Cube**: Product of literals e.g. $+3a^2b$, $-2a^3b^2c$*
  - ***SOP**: Sum of cubes e.g. $+3a^2b - 2a^3b^2c$*
  - ***Cube-free expression**: No literal or cube can divide all the cubes of the expression*
  - ***Kernel**: A cube free sub-expression of an expression, e.g. 3 – 2abc*
  - ***Co-Kernel**: A cube that is used to divide an expression to get a kernel, e.g. a^2b*

Power-conscious multi-level logic opt. (III)

- Example of Kernel computation
  - Function: fx = ace + bce + de + g
  - Kernel set
    - Divide fx by a ➔ Get ce ➔ Not cube free
    - Divide fx by b ➔ Get ce ➔ Not cube free
    - Divide fx by c ➔ Get ae + be ➔ Not cube free
    - Divide fx by ce ➔ Get a + b ➔ Cube free ➔ Kernel
    - Divide fx by d ➔ Get e ➔ Not cube free
    - Divide fx by e ➔ Get ac + bc + d ➔ cube free ➔ Kernel
    - Divide fx by g ➔ Get 1 ➔ Not cube free
    - Expression fx is a kernel of itself because cube free
  - K(fx) = {(a+b); (ac+bc+d); (ace+bce+de+g)}

Kernels and Kernel Intersections

DEFINITION:

An expression is cube-free if no cube divides the expression evenly (i.e. there is no literal that is common to all the cubes).

ab + c is cube-free

ab + ac and abc are not cube-free

Note: a cube-free expression must have more than one cube.

DEFINITION:

The primary divisors of an expression F are the set of expressions

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

Kernels and Kernel Intersections

DEFINITION:

The kernels of an expression F are the set of expressions
K(F) = {G | G $\in$ D(F) and G is cube-free}.

In other words, the kernels of an expression F are the cube-free primary divisors of F.

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

Example:

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

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

kernels	co-kernels
a+b+c	df, ef
d+e	af, bf, cf
(a+b+c)(d+e)f+g	1

Kernels: Example

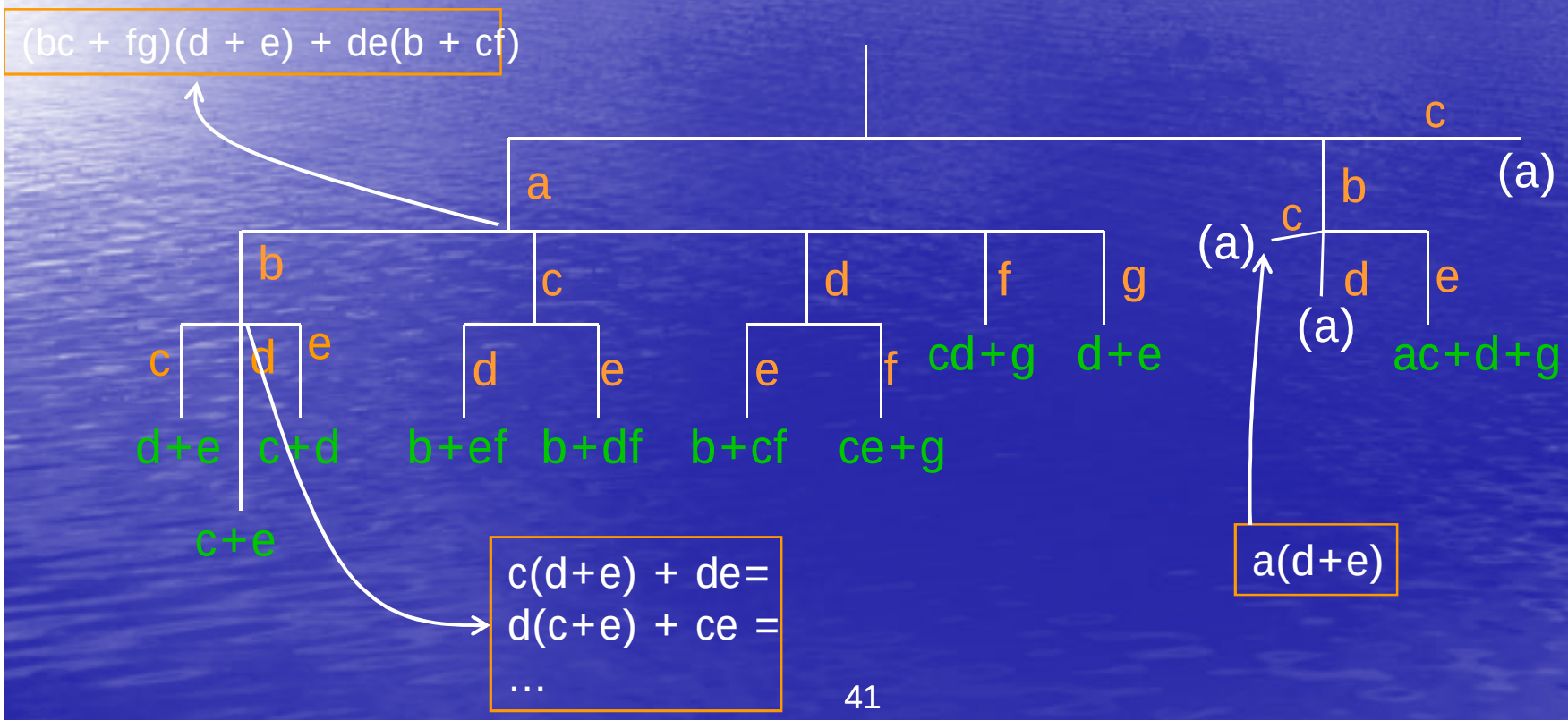
F = adf + aef + bdf + bef + cdf + cef + bfg + h

= (a+b+c)(d+e)f + bfg + h

cube	Prim. Div.	Kernel	Co-kernel	level
a	df+ef	NO	NO	--
b	df+ef+fg	NO	NO	--
bf	d+e+g	YES	YES	0
cf	d+e	YES	YES	0
df	a+b+c	YES	YES	0
fg	b	NO	NO	--
f	(a+b+c)(d+e)+bg	YES	YES	1
1	F	YES	YES	2

Kerneling Illustrated

abcd + abce + adfg + aefg + adbe + acdef + beg



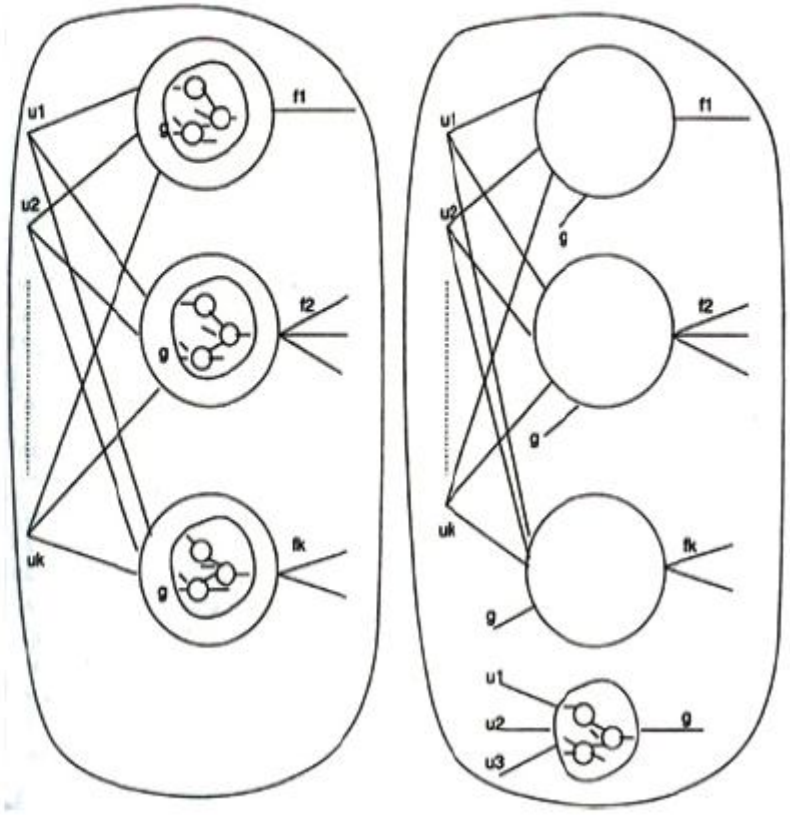
Kerneling Illustrated

co-kernels	kernels
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
abd	c + e
abe	c + d
ac	b(d + e) + def
acd	b + ef

Note: f/bc = ad + ae = a(d + e)

Power-conscious multi-level logic opt. (IV)

- A common sub-expression of functions
  - $g = g(u_1, u_2, \ldots, u_K)$, $K >= 1$
- Functions: $f_1, f_2, \ldots, f_L$, $L >= 2$
- Internal nodes of g: $v_1, v_2, \ldots, v_M$, $M >= 0$
- A example of factoring
  - g is factored out
  - Signal probabilities and acitivities are unchanged
  - Capcitiances are changed



Power-conscious multi-level logic opt. (V)

- Power saving for inputs

$$(L-1)V_{dd}^2 C_0 \sum_{k=1}^{K} n_{u_k} D(u_k)$$

  - $D(u_k)$: activity at node u_k
  - n_{uk}: # of gates belonging to node g and driven by signal u_k
  - C_0: the load capacitance due to a fanout equal to one gate
- Power saving for internal nodes

$$(L-1)V_{dd}^2 C_0 \sum_{k=1}^{K} n_{v_m} D(v_m)$$

- Total power saving

$$\Delta W(g) = (L-1)V_{dd}^2 C_0 \left(\sum_{k=1}^{K} n_{u_k} D(u_k) + \sum_{k=1}^{K} n_{v_m} D(v_m)\right)$$

Power-conscious multi-level logic opt. (VI)

- Impact on area

$$\Delta A(g) = (L-1)[T(g)-1]-1$$

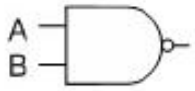
- Net saving

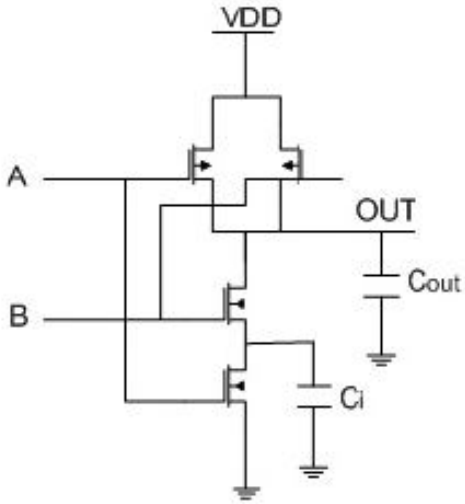
$$S(g) = \alpha_W \cdot \frac{\Delta W(g)}{W_T} + \alpha_A \cdot \frac{\Delta A(g)}{A_T}$$

  - W_T: Average power of the input Boolean network
  - A_T: Area of the input Boolean network
  - $\alpha_W + \alpha_A = 1$

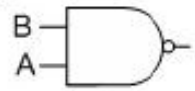
Equivalent pin ordering

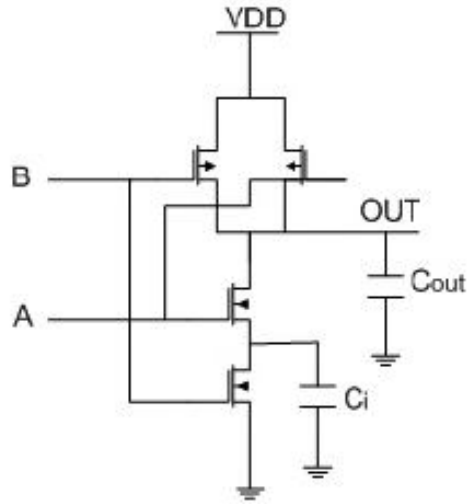
- Change the input connection based on the signal probability and signal activity
  - Suppose that B is high and A is switching from low to high





Charges in C_{out} and C_i are discharged



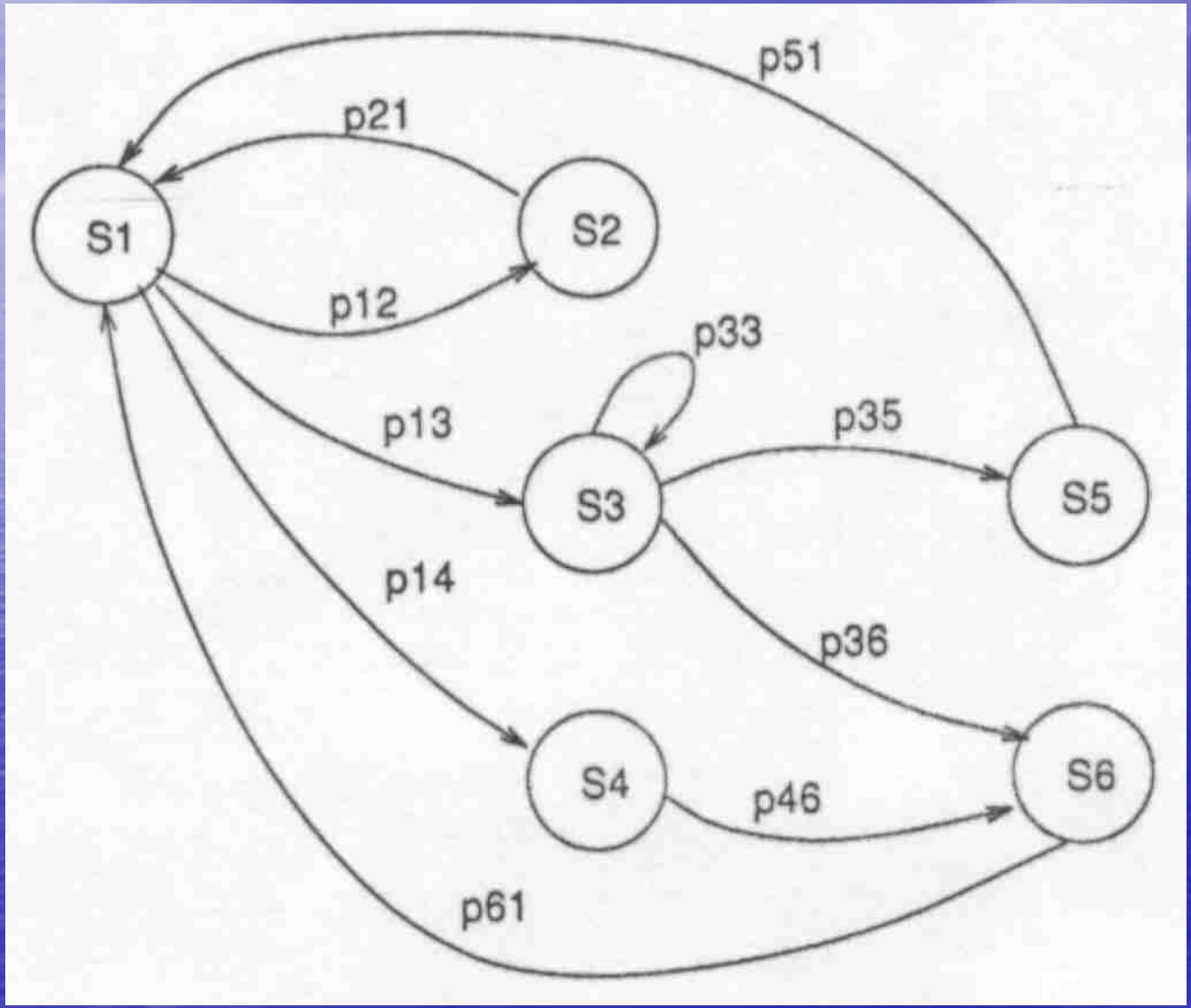


Charges in C_{out} are discharged

Probabilistic State Transition Graphs (STGs)

- Edges showing state transitions not only indicate input values causing transitions and resulting outputs
- Also have labels p_{ij} giving conditional probability of transition from state S_i to S_j
  - ***Given that machine is in state S_i***
  - ***Directly related to signal probabilities at primary inputs***
  - $$\sum_m p_{jm} = 1$$
- Introduce self-loops in STG for don't care situations to transform incompletely-specified machine into completely-specified machine

Example



Relationship Between State Assignment and Power

- Hamming distance between states S_i and S_j:
  - ***H (S_i, S_j) = # bits in which the assignments differ***
- Average Power: $\text{Power}_{\text{avg}} = \frac{1}{2} V_{DD}^2 \sum C_i D(i)$
  - ***D (i) = signal activity at node i***
  - ***Approximate C_i with fanout factor at node i***
- Average power proportional to:

$$\Phi = \sum_i \text{fanout}_i D(i)$$

Handling Present State Inputs

- Find state transitions (Si, Sj) of highest probability
- Minimize H (Si, Sj) by changing state assignment of Si, Sj
- Requires system simulation of circuit over many clock periods, noting signal values and transitions
- If one-hot design is used, note that H = 2 for all states
  - ***Impossible to obtain optimum power reduction***
  - ***Uses too many flip-flops***
- Optimization cost function:

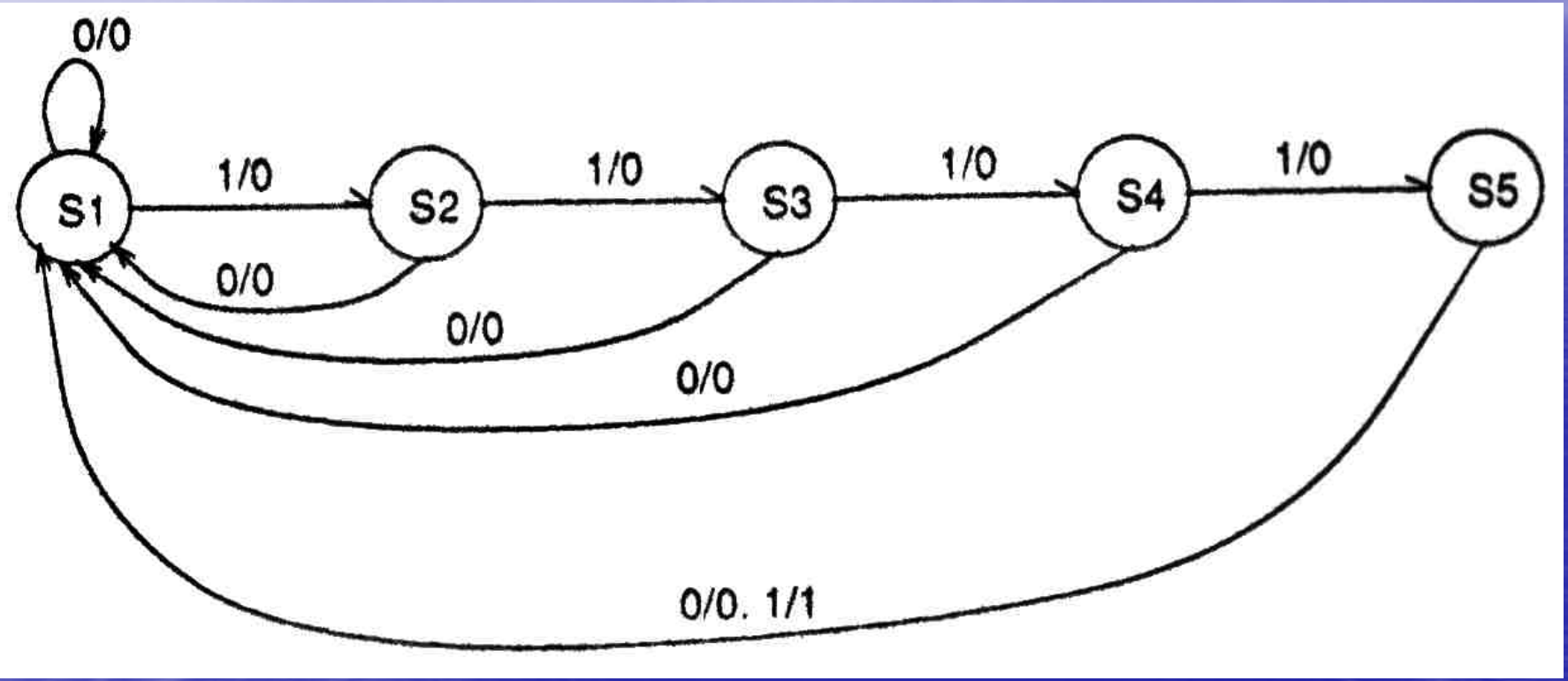
$$\gamma = \sum_{\text{over all edges}} p_{ij} H(S_i, S_j)$$

Simulated Annealing Optimization Algorithm

- Allowed moves:
  - ***Interchange codes of two states***
  - ***Assign an unassigned code to a state that is randomly picked for an exchange***
- Accept move if it decreases g
- If move increases g, accept with probability:

$$e^{-|d(g)|/Temp}$$

Example State Machine



State Assignments

- Coding 1 uses 15% more power than coding 2

	Coding 1	Coding 2
S1	010	000
S2	101	100
S3	000	111
S4	111	010
S5	100	011

Multi-Level Logic Optimization for Low Power

- Combinational logic is F (I, V)
  - ***I = set of primary inputs***
  - ***V = present state inputs***
- Need to estimate probabilities and activities of V inputs (same as next state outputs but delayed one clock period) in order to synthesize logic for minimum power
  - ***Use methods of Chapter 3***
- Randomly generate PI signals with probabilities and activities conforming to a given distribution
  - ***Get D (v_i) = transition activity at input v_i (transitions / clock period)***
  - ***Get from fast state transition diagram simulation***

Power-Driven Multi-Level Logic Optimization

- Use Berkeley MIS tool
  - ***Takes set of Boolean functions as input***
  - ***Procedure kernel finds all cube-free multiple or single-cube divisors of each Boolean function***
  - ***Retains all common divisors***
  - ***Factors out best few common divisors***
  - ***Substitution procedure simplifies original functions to use factored-out divisor***
- Original criteria for selecting common divisor:
  - ***Chip area saving***
- New criterion: power saving

Boolean Expression Factoring

- $g = g(u_1, u_2, \ldots, u_K)$, $K \geq 1$ is common sub-expression
- When g factored out of L functions, signal probabilities and activities at all circuit nodes are unchanged
- Capacitances at output of driver gates $u_1, u_2, \ldots, u_K$ change
- Each drives $L-1$ fewer gates than before
- Reduced power:

$$(L-1)V_{dd}^2 C_0 \sum_{k=1}^{K} n_{u_k} D(u_k)$$

- $D(x)$ = activity at node x
- n_{uk} = # gates belonging to node g and driven by u_K

Factoring (continued)

- Only one copy now of g instead of L copies
  - ***L-1 fewer copies of internal nodes $v_1, v_2, \ldots, v_m$ in factored-out hardware for switching and dissipating power***
- Power saving:

$$(L-1)V_{dd}^2 C_0 \sum_{m=1}^{M} n_{v_m} D(v_m)$$

- Total power saving:

$$\Delta W(g) = (L-1)V_{dd}^2 C_0 \left(\sum_{k=1}^{K} n_{u_k} D(u_k) + \sum_{m=1}^{M} n_{v_m} D(v_m) \right) \quad (4.21)$$

Factoring (concluded)

- T (g) = # literals in factored form of g
- Area saving: $\Delta A(g) = (L - 1)[T(g) - 1] - 1$
- Net saving of power and area:

$$S(g) = \alpha_W \cdot \frac{\nabla W(g)}{W_T} + \alpha_A \cdot \frac{\Delta A(g)}{A_T} \qquad (4.22)$$

Optimization Cost Criteria

The accepted optimization criteria for multi-level logic are to *minimize* some function of:

1. Area occupied by the logic gates and interconnect (approximated by literals = transistors in technology independent optimization)
2. Critical path delay of the longest path through the logic
3. Degree of testability of the circuit, measured in terms of the percentage of faults covered by a specified set of test vectors for an approximate fault model (e.g. single or multiple stuck-at faults)
4. Power consumed by the logic gates
5. Noise Immunity
6. Wireability

while simultaneously satisfying upper or lower bound constraints placed on these physical quantities

Factored Form

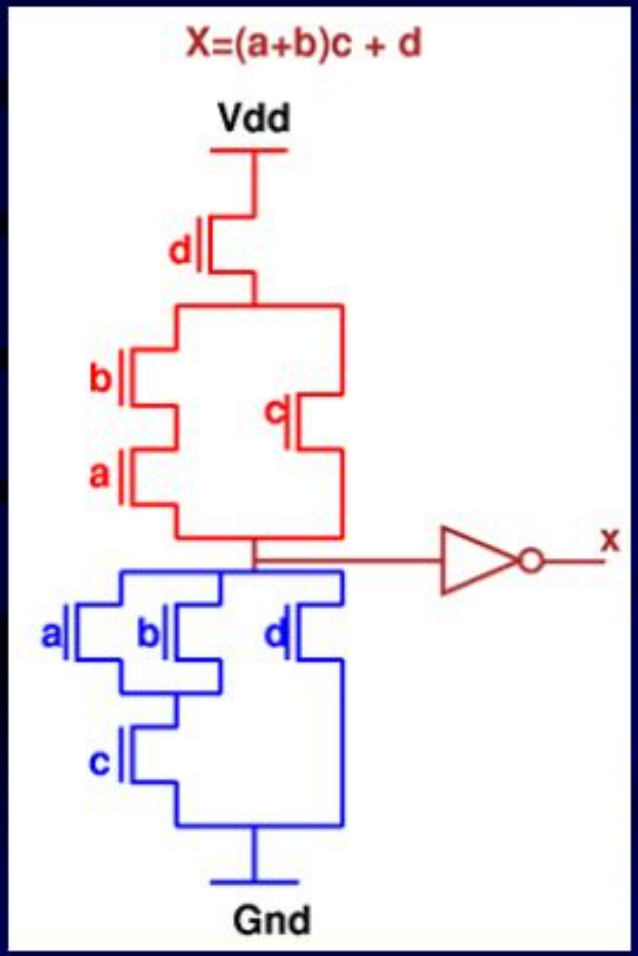
$$a$$
$$a'$$
$$ab'c$$
$$ab + c'd$$
$$(a+b)(c+a'+de)+f$$

- Where a, a′, b′, c, are called literals
- Factored form can be derived from a SOP

$$ace + ade + bce + bde + e'$$

$$e(a+b)(c+d)+e'.$$

CMOS view



Note:

literal count ∝ transistor count ∝ area

(however, area also depends on wiring)

Definition

- A factored form is
  - A product or a sum, where
  - A product is
    - Either a single literal
    - Or a product of factored form
  - A sum is
    - either a single literal
    - Or a sum of factored form

$a + b'c$

$((a' + b)cd + e)(a + b') + e'$

Yes

$(a + b)'c$

No

Factored form is not unique

$ab + c(a + b)$
$bc + a(b + c)$
$ac + b(a + c).$

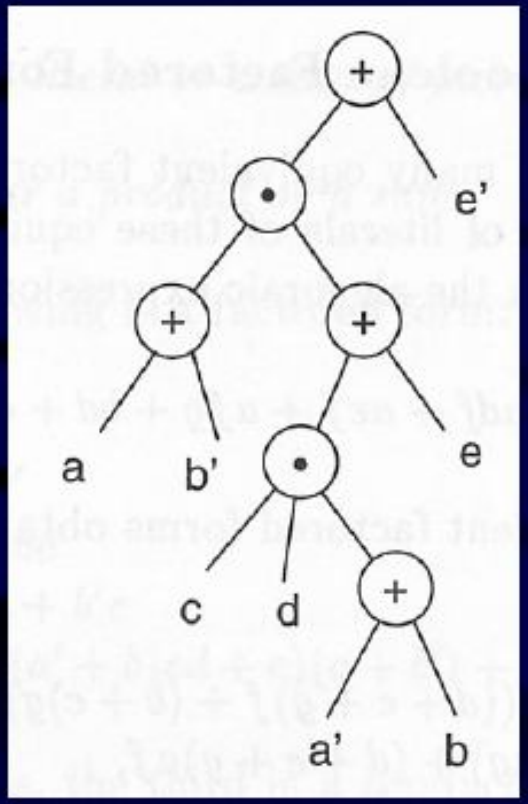
$$abg + acg + adf + aef + afg + bd + ce + be + cd,$$

All equivalent

$$(b + c)(d + e) + ((d + e + g)f + (b + c)g)a, \quad \text{(12 literals)}$$
$$(b + c)(d + e + ag) + (d + e + g)af, \quad \text{(11 literals)}$$
$$(af + b + c)(ag + d + e). \quad \text{(8 literals)}$$

- There are 12 literals in the first form
- There is only 8 literals in the 3rd one
- Take the first one and multiply out, we can get the original expression
  - Without using xx′=0 and xx=x
- Take the 3rd one and multiply out, we get a different expression
  - Because we have afag

Factoring tree



$$((a' + b)cd + e)(a + b') + e',$$

- A sub-tree is called a factor
  - (a+b')
  - cd(a'+b)
- Two trees are equivalent if they represents the same function
- Two trees are syntactically equivalent if they are isomorphic
  - (a+b)(c+d)e = (a+b)e(c+d)

Example

$$\begin{aligned} F &= adf + aef + bdf + bef + cdf + cef + bfg + h \\ &= (a+b+c)(d+e)f + bfg + h \end{aligned}$$

kernel	co-kernel	level
$d+e$	af, cf	0
$d+e+g$	bf	0
$a+b+c$	df, ef	0
$(a+b+c)(d+e)+bg$	f	1
$((a+b+c)(d+e)+bg)f+h$	1	2

F/a = df+ef is not a cube-free divisor

A kernal may have more than 1 co-kernel

If the original expression is cube free, a co-kernel can be the “1”

Common subexpressions

$f = abc' + adg + ac'f$

$g = ac'd + adf$ (15)

ac' : 3

ad : 3

⇓

$u = ac'$

$f = ub + adg + uf$

$g = ud + adf$ (14)

30 tran

⇓

24 tran

⇓

$u = ac'$

$f = u(b+f) + adg$

$g = d(u+af)$ (12)

1) Kernel Extraction : Consider the Boolean Function

F = uvy + vwy + xy + uz + vz.

- Identify all co-kernel/kernel paris of F. State their levels.

2) Weak Algebraic Division (10 points):

We have studied an algorithm to perform weak algebraic division in class that can be used to decompose a function F, algebraically,as Fdividend = Gdivisor ·Hquotient+Rremainder. Divide F = ab+ac+ad′+bc+bd′ by the following:

- G = a + b. What is the quotient and the remainder?
- G = c+d′. What is the quotient and the remainder?

Optimization Algorithm

Algorithm: ***Power Dissipation Driven Multilevel Logic Optimization***

Inputs: Boolean network F, input signal probability $P(x_i = 1)$ and transition activity $D(x_i)$ for each primary input x_i, N_0.

Output: Optimized Boolean network F', $P(s = 1)$ and $D(s)$ for each node in the optimized network.

Step 0: Compute $P(s = 1)$ and $D(s)$ for each node s in F.

Step 1: Repeat steps 2–4.

Step 2: $G' = \bigcup_{f \in F} K(f)$, where $K(f)$ = set of all divisors of f. The set of kernels (cube-free divisors) is computed for each function. G' is the union of all the sets of kernels.

Step 3: $G = \{g | g(\in G') \wedge (g \in K(f_i)) \wedge (g \in K(f_j)) \wedge (i \neq j)\}$. The set of kernel intersections, G, is the set of those kernels that apply to more than one function.

Step 4: Repeat steps 5–7 N_0 times.

Step 5: Find g, p_g, d_g such that

$$(g \in G) \wedge (\forall h \in G)[S(g) \geq S(h)] \wedge [p_g = P(g = 1)] \wedge [d_g = D(g)]$$

Optimization Algorithm (concluded)

If $\Delta A(g) < 0$, terminate procedure. The kernel intersection g brings about the largest net saving. The signal probability and transition activity of the output signal of g are remembered. If the area component of net saving is negative, there are no more multiple-cube divisors common to two or more functions and so we stop.

Step 6: For all f such that $f \in F \wedge g \in K(f)$, substitute variable g in f in place of the subexpression $g(u_1, u_2, \ldots, u_K)$. Each function, which has the expression g as one of its kernels, has the new variable g substituted into it in place of the expression.

Step 7: $F = F \cup \{g\}, G = G - \{g\}$. Here, $P(g = 1) = p_g$, $D(g) = d_g$. The new function g is added to the set of functions F. The newly added node is assigned signal probability and activity values from step 5.

At the beginning of the optimization procedure, signal probabilities and activities for each internal and output node is computed. Each time a common divisor g = $g(u_1, u_2, .., u_K)$ is factored out, the $P(u_k = 1)$ and $A(u_k)$, 1 $\leq$ k $\leq$ K, are known but $P(v_m = 1)$ and $A(v_m)$, 1 $\leq$ m $\leq M$, are not. The latter are computed when $\Delta R(g)$ is being evaluated and are retained. Thus once again $P(s = 1)$ and $A(s)$, for each node s are known.

The parameter N_0 is used to control the number of kernel intersections (cube free divisors common to two or more functions) which are substituted into all the functions before the set of kernel intersections is recomputed. Recomputing after a single substitution is wasteful as only some of the functions have changed. On the other hand, with each substitution, some of the kernel intersections become invalid.

Algorithm : Reliability driven multilevel logic optimization

Inputs : Boolean network F, input signal probability $P(x_i = 1)$ and activity $A(x_i)$ for each primary input x_i, N_0

Output : Optimized Boolean network F', $P(s = 1)$ and $A(s)$ for each node in the optimized network.

Step *0* : Compute $P(s=1)$ and $A(s)$ for each node s in F.

Step 1 : Repeat steps **2** through 4.

Step 2 : $G' = \bigcup_{f \in F} K(\mathrm{f})$, where $K(\mathrm{f}) =$ set of all divisors of f. Set of kernels (cube free divisors) is computed for each function. G' is the union of all the sets of kernels.

Step *3* : $G = \{\mathrm{g} \mid (\mathrm{g} \in G') \wedge (\mathrm{g} \in K(f_i)) \wedge (\mathrm{g} \in K(f_j)) \wedge (\mathrm{i} \neq \mathrm{j})\}$. G, the set of kernel intersections, is the set of those kernels which apply to more than one function.

Step 4 : Repeat steps 5 through 7 N_0 times

Step 5 : Find g, p_g, d_g such that

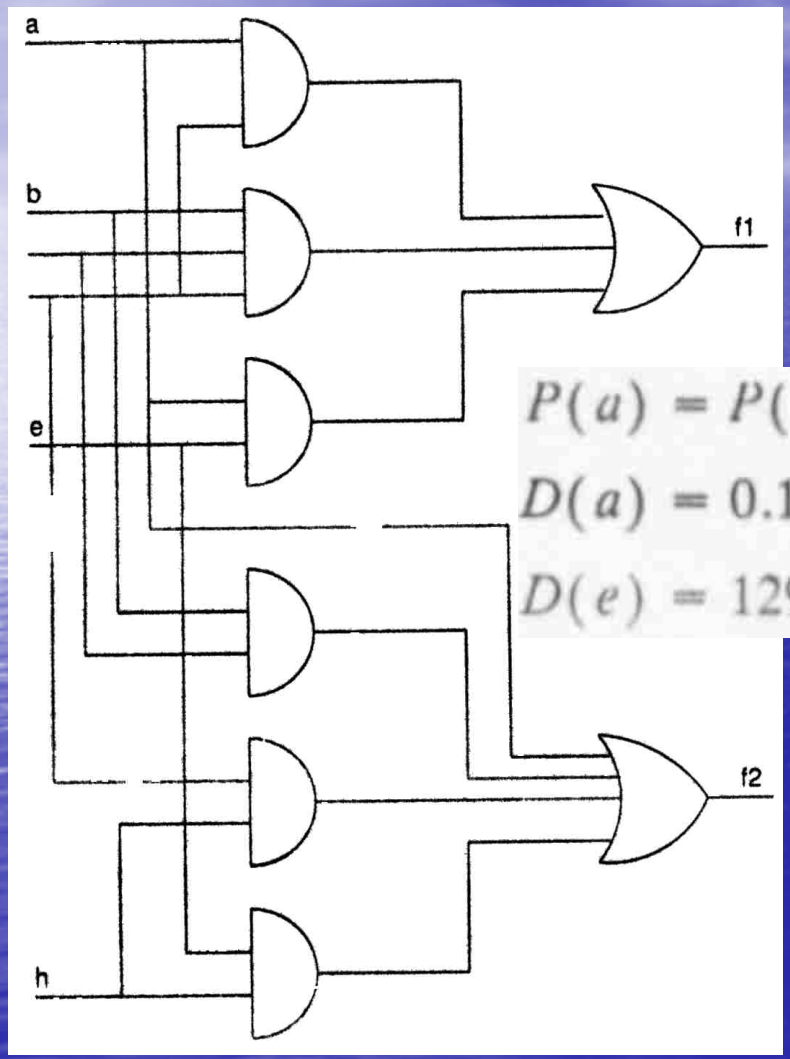
$$(g \in G) \wedge (\forall h \in G)[S(g) \geq S(h)] \wedge (p_g = P(g=1)) \wedge (d_g = A(g))$$

If $\Delta A(g) < 0$, terminate procedure. g is the kernel intersection which brings about largest saving. The signal probability and activity of the output signal of g are remembered. If the area component of total saving is negative, there are no more multiple-cube divisors common to two or more functions and so we stop.

Step *6* : For all f such that $\mathrm{f} \in F \wedge \mathrm{g} \in K(f)$, substitute variable g in f in place of the subexpression $g(u_1, u_2, .., u_K)$. Each function, which has the expression g as one of its kernels, has the new variable g substituted into it in place of the expression.

Step **7**: $F = \mathrm{F} \cup \{g\}$, $\mathrm{G} = \mathrm{G} - \{\mathrm{g}\}$. $\mathrm{P}(\mathrm{g}=1) = p_g$, $A(g) = d_g$. New function g is added to the set of functions F. The newly added node is assigned signal probability and activity values from step 5.

Example Unoptimized Circuit

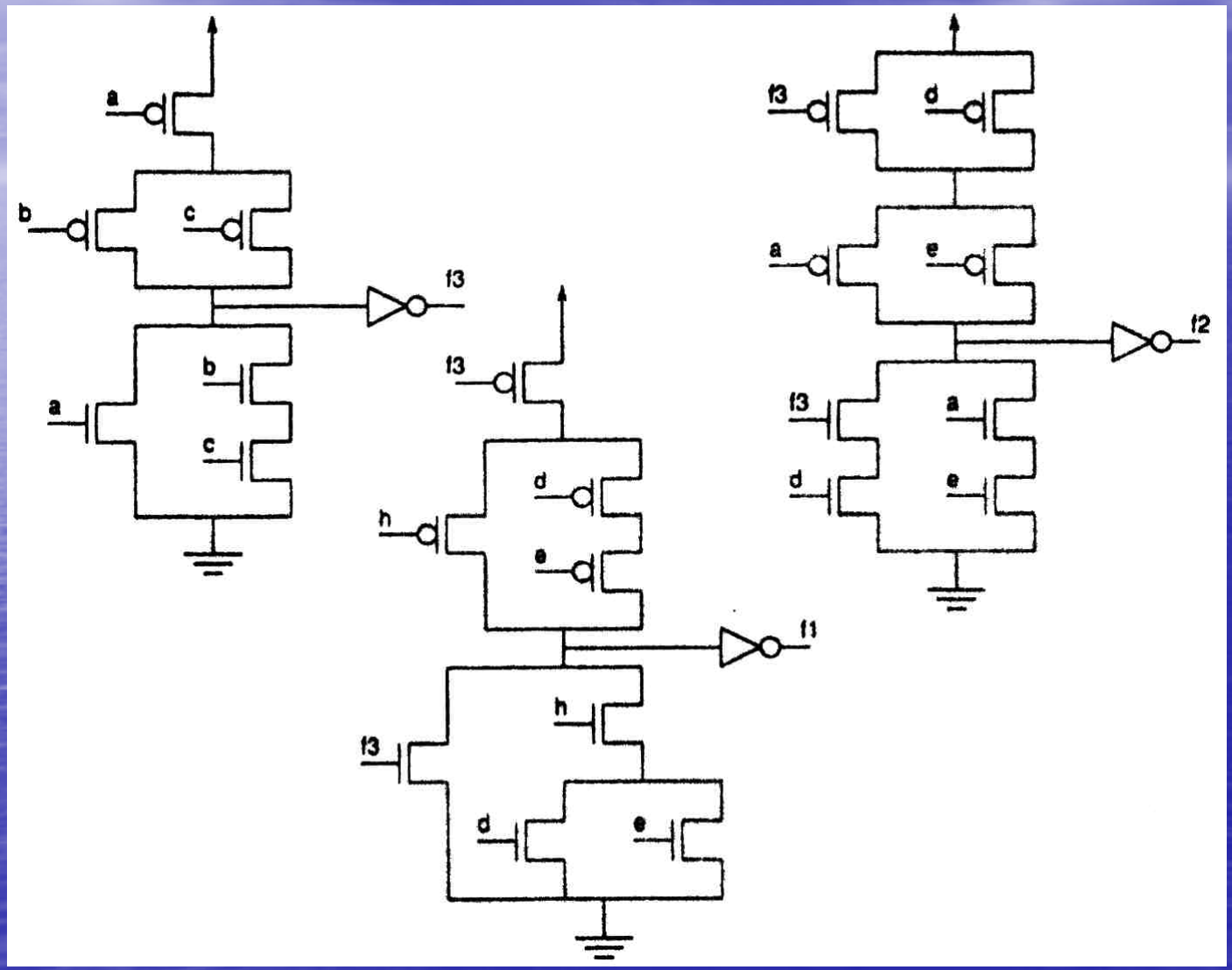


$$P(a) = P(b) = P(c) = P(d) = P(e) = P(h) = 0.5$$

$$D(a) = 0.1, D(b) = 0.6, D(c) = 3.6, D(d) = 21.6,$$

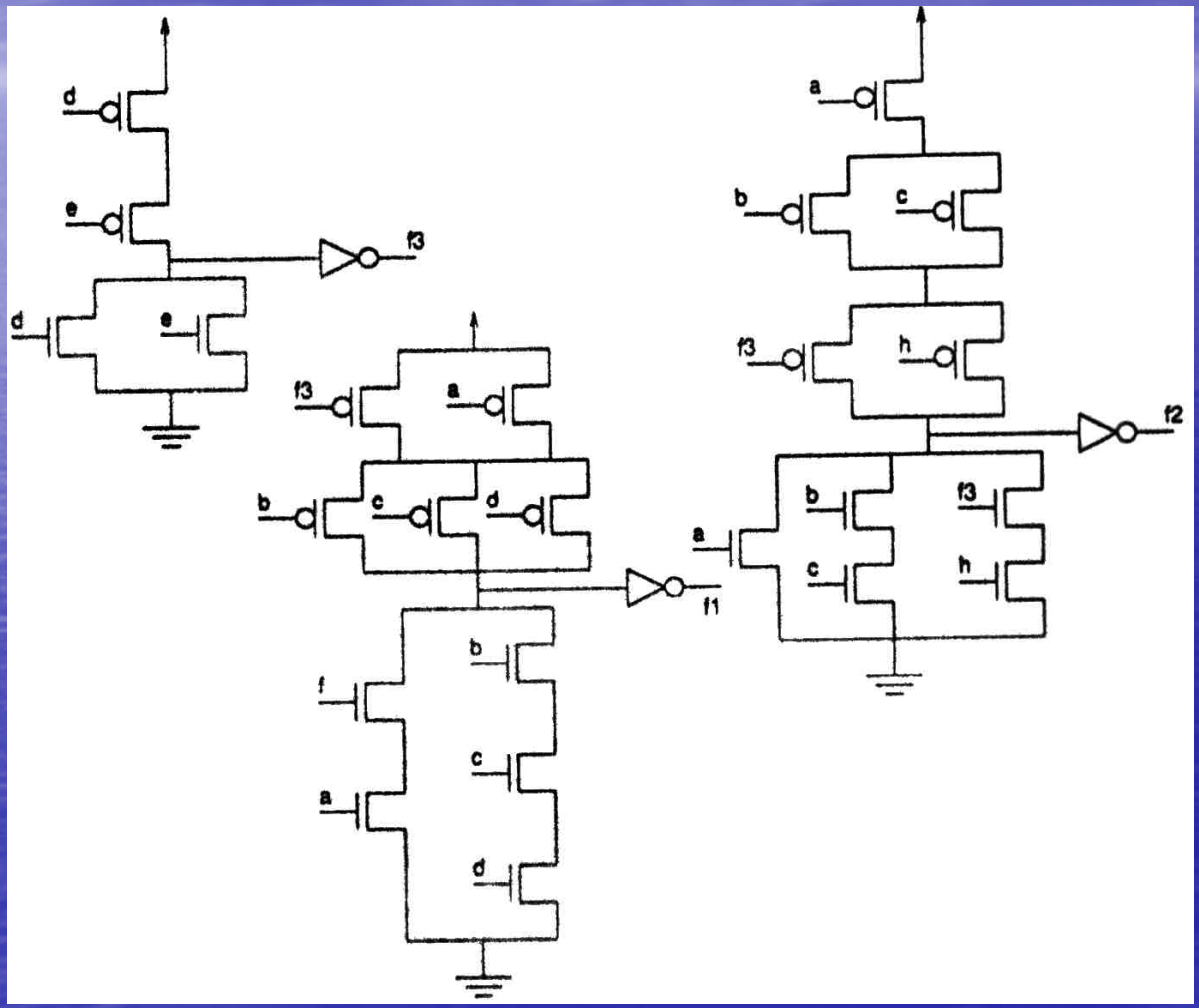
$$D(e) = 129.6, D(h) = 3.6$$

Optimization for Area Alone

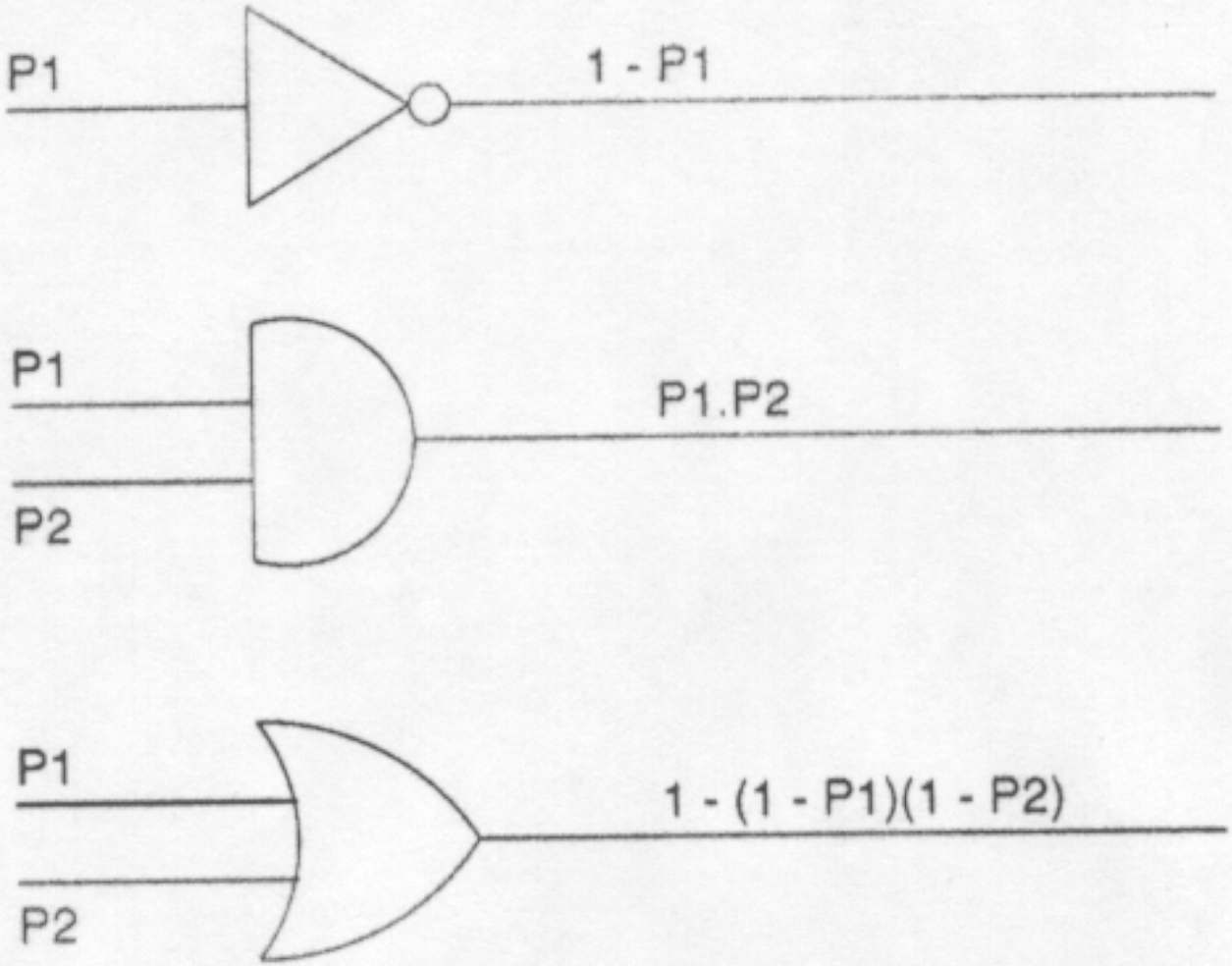


Optimization for Low-Power Alone

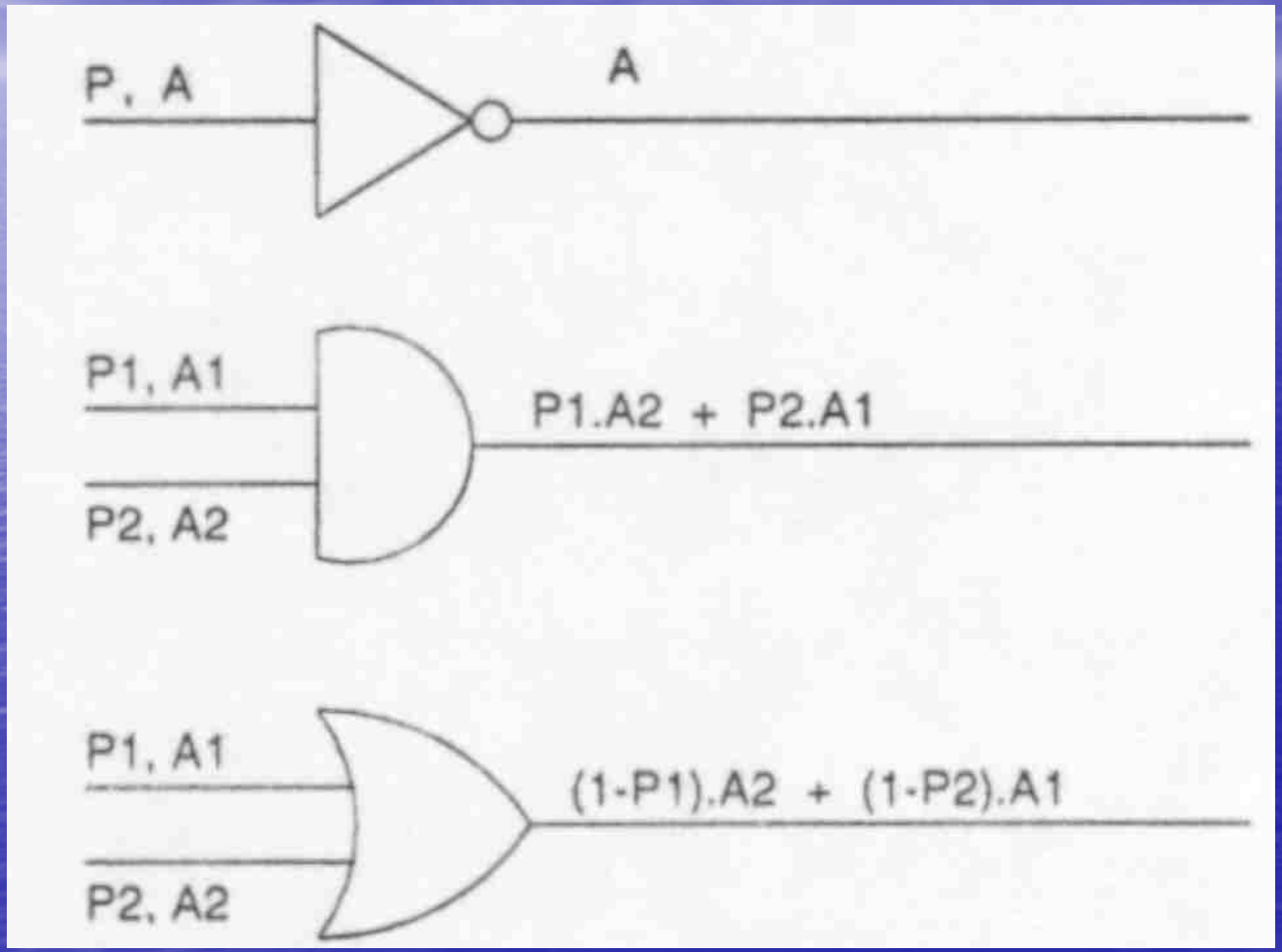
- Large area but reduces power from 476.12 to 423.12



Example Signal Probabilities



Propagating Combinational Signal Activities



If $\Delta A(g) < 0$, terminate procedure. The kernel intersection g brings about the largest net saving. The signal probability and transition activity of the output signal of g are remembered. If the area component of net saving is negative, there are no more multiple-cube divisors common to two or more functions and so we stop.

Step 6: For all f such that $f \in F \wedge g \in K(f)$, substitute variable g in f in place of the subexpression $g(u_1, u_2, \ldots, u_K)$. Each function, which has the expression g as one of its kernels, has the new variable g substituted into it in place of the expression.

Step 7: $F = F \cup \{g\}, G = G - \{g\}$. Here, $P(g = 1) = p_g$, $D(g) = d_g$. The new function g is added to the set of functions F. The newly added node is assigned signal probability and activity values from step 5.

An Example

Let us consider a small circuit to illustrate the application of the above procedure.

Let $F = \{f_1, f_2\}$ be a two-output circuit given by

$$f_1 = ad + bcd + ae \quad f_2 = a + bc + de + eh$$

The signal probabilities and the transition activities at the primary inputs are assumed to be

$$P(a) = P(b) = P(c) = P(d) = P(e) = P(h) = 0.5$$
$$D(a) = 0.1, D(b) = 0.6, D(c) = 3.6, D(d) = 21.6,$$
$$D(e) = 129.6, D(h) = 3.6$$

Since F is a small circuit, we recompute the set of kernel intersections after every substitution, that is, $N_0 = 1$.

Figure 4.24 shows the circuit F as an interconnection of logic gates.

The area and power dissipation of the unoptimized circuit are

$$A_T(F) = 6 + 6 = 12 \qquad W_T(F) = 503.35 \text{ units}$$

The sets of kernels for f_1 and f_2 are computed:

$$K(f_1) = \{a + bc, d + e\} \qquad K(f_2) = \{a + bc, d + e\}$$

The union of the sets of kernels of all the functions, G', is computed as

$$G' = \{a + bc, d + e\}$$

The set of kernel intersections, G, that is, those kernels that apply to two or more functions, is computed as

$$G = \{a + bc, d + e\}$$

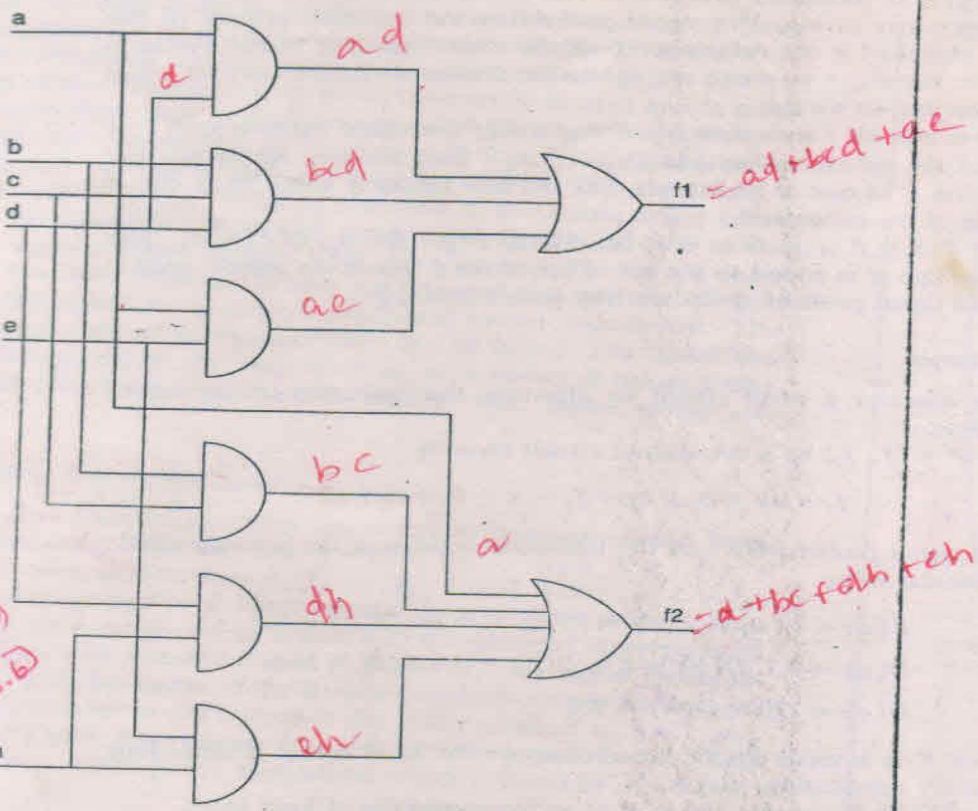


Figure 4.24 Unoptimized circuit.

Let us first consider $\alpha_A = 1.0$, $\alpha_W = 0$ (area optimization only). The net saving due to each of the kernel intersections, $g \in G$, is determined and the kernel intersection corresponding to the largest net saving is selected:

$$g = a + bc, \Delta A(g) = 1, \quad \Delta W(g) = 6.4, \quad P(g) = 0.625,$$
$$D(g) = 1.125, \quad S(g) = 0.083$$
$$g = d + e, \Delta A(g) = 0, \quad \Delta W(g) = 151.2, \quad P(g) = 0.75,$$
$$D(g) = 75.6, \quad S(g) = 0$$

Hence, $g = a + bc$ is selected. It is substituted into functions in F and added to F to give F^*:

$$F^* = \{f_1, f_2, f_3\}$$

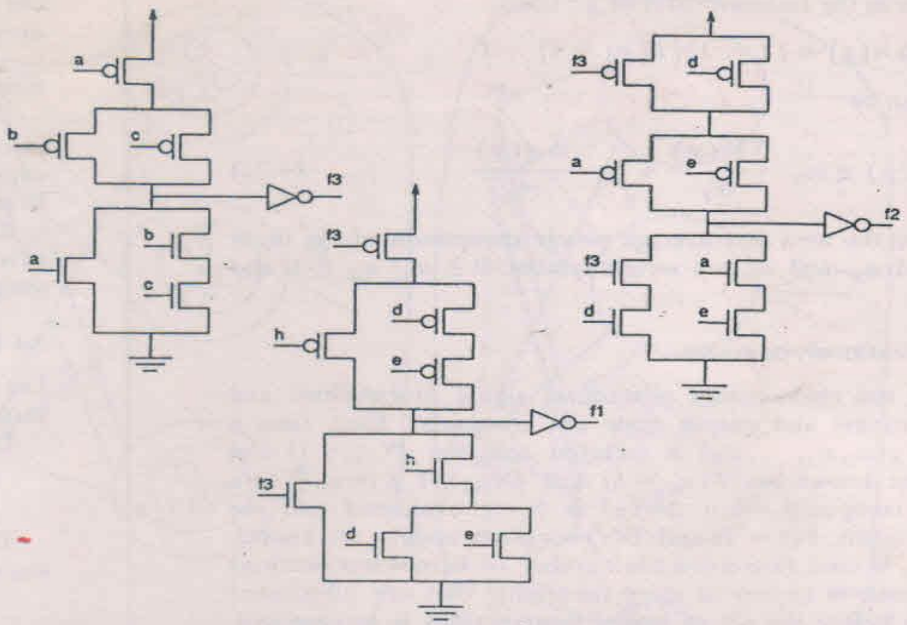


Figure 4.25 Complex gate implementation of the circuit optimized for area alone.

where

$$f_3 = a + bc$$
$$f_1 = f_3 d + ae$$
$$f_2 = f_3 + dh + eh$$

The total area and power dissipation of circuit F^* are

$$A_T(F^*) = 3 + 4 + 4 = 11 \qquad W_T(F^*) = 476.5 \text{ units}$$

No more kernel intersections can be found and the procedure terminates. The complex logic gate implementation of the optimized circuit is shown in Figure 4.25. It requires 28 transistors.

Next we consider $\alpha_A = 0, \alpha_W = 1.0$, which causes optimization for low-power dissipation. Once again each of the kernel intersections $g \in G$ is evaluated and the best is selected:

$$g = a + bc, P(g) = 0.625, \quad D(g) = 1.125, \quad \Delta A(g) = 1,$$
$$\Delta W(g) = 6.4, \quad S(g) = 0.013$$
$$g = d + e, \Delta A(g) = 0, \quad \Delta W(g) = 151.2, \quad P(g) = 0.75,$$
$$D(g) = 75.6, \quad s(g) = 0.3$$

This time $g = d + e$ is selected. It is substituted into functions in F and added to F to give F^{**}:

$$F^{**} = \{f_1, f_2, f_3\}$$

where

$$f_3 = d + e$$
$$f_1 = f_3 a + bcd$$
$$f_2 = a + bc + f_3 h$$

The total area and power dissipation of circuit F^{**} are

$$A_T(F^{**}) = 2 + 5 + 5 = 12 \qquad W_T(F^{**}) = 423.12 \text{ units}$$

No more kernel intersections can be found. The complex logic gate implementation of the optimized circuit is shown in Figure 4.26. It requires 30 transistors.

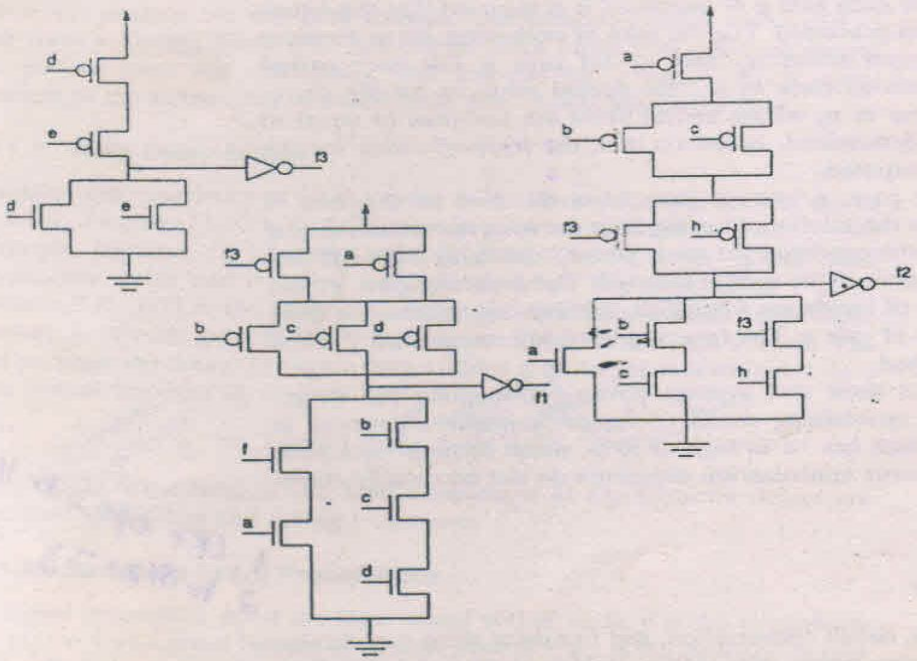


Figure 4.26 Complex gate implementation of the circuit optimized for power.

Results

- On the MCNC Benchmarks:
- Two-stage process
  - ***State assignment problem***
  - ***Multi-level combinational logic synthesis based on power dissipation and area reduction***
- Result:
  - ***25% reduction in power***
  - ***5% increase in area***

Technology Mapping for Low Power

- Problem statement:
  - ***Given Boolean network optimized in a technology-independent way and a target library, bind network nodes to library gates to optimize a given cost***
- Method:
  - ***Decompose circuit into trees***
  - ***Use dynamic programming to cover trees***
  - ***Cost function:***

$$\text{Area}(g) + \sum_{n_i \in \text{inputs}\ (g)} \text{MinArea}(n_i) \qquad (4.23)$$

  - ***Traverse tree once from leaves to root***

Extension for Low-Power Design

- Power dissipation estimate:

$$\text{Power} = \sum_{i=1}^{i=n} \frac{1}{2} V_{dd}^2 a_i C_i f \qquad (4.24)$$

- Estimate partial power consumption of intermediate solutions
- Cost function:

$$\text{power}(g, n) = \text{power}(g) + \sum_{n_i \in \text{inputs}\ (g)} \text{MinPower}(n_i) \qquad (4.25)$$

- MinPower (n_i) is minimum power cost for input pin n_i of g
- power (g) = 0.5 f V_{DD}^2 a_i C_i
- Formulation:

$$\text{Minimize} \quad wP + (1 - w)R \quad \text{such that } T \le T_{\max}$$

  - ***R = Total Area, w gives their relative importance***
  - ***f = frequency, T = circuit delay***

Top-Level Mapping Algorithm

- Overall process:
  - ***From tree leaves to root, compute trade-off curves for matching gates from library***
  - ***From root to leaves:***
    - **Select minimum-cost solution**
- Reduces average power by 22% while keeping the same delay
  - ***Sometimes increases area as much as 39%***

Circuit-Level Optimizations

Algorithm Components

1. Find which gate to examine next
2. Use a set of transformations for the gate
3. Compute overall power improvement due to transformations
4. Update the circuit after each transformation

Gate Delay Model

- For every input terminal I_i and output terminal O_j of every gate:
  - ***$T^{i}_{i,j}(G)$ – fanout load independent delay (intrinsic)***
  - ***$R_{i,j}(G)$ – additional delay per unit fanout load***
- Total gate propagation delay from input to output:

$$T^{i}_{i,j}(G) + R_{i,j}(G)C_j(G)$$

  - ***Normalize all activities d_y by dividing them by clock activity ($2f$)***
- Probability of rising or falling transition at y:

$$p_y^{\updownarrow} = d_y \qquad (4.27)$$

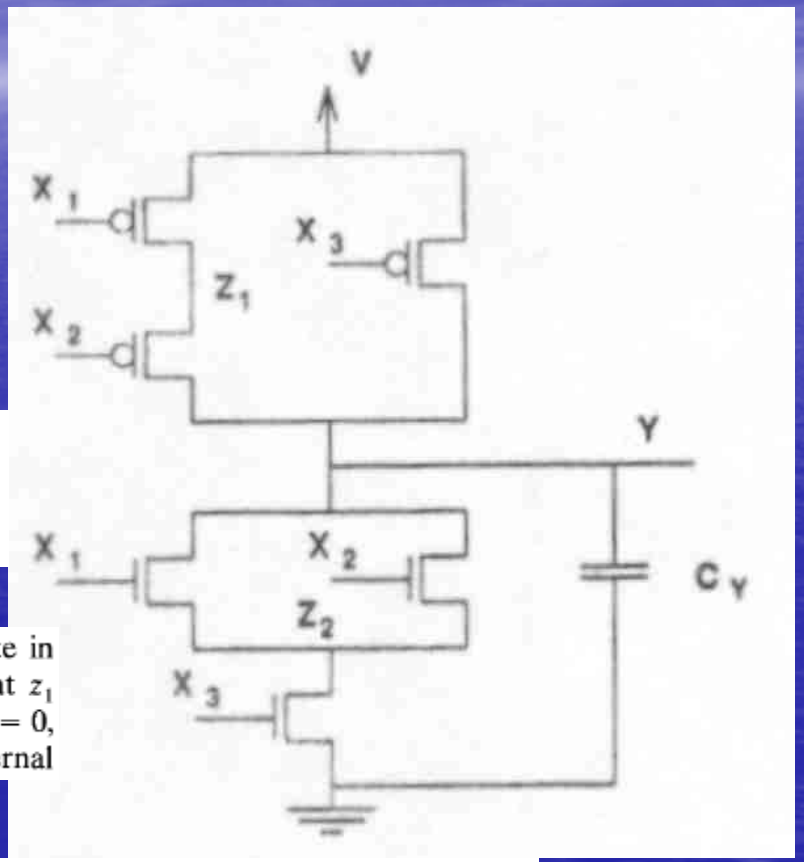
$$p_y^{\uparrow} = p_y^{\downarrow} = \tfrac{1}{2} p_y^{\updownarrow} \qquad (4.28)$$

CMOS Gate Usage

- Deep sub-micron technology:
- Delay of NAND/NOR to INVERTER delay lessens in deep sub-micron technology
  - ***Series transistor connection V_{ds} and V_{gs} smaller than that for inverter transistor***
- Encourages wider use of complex CMOS gates
- Important to order series transistors correctly
  - ***Delay varies by 20%***
  - ***Power varies by 10%***

CMOS Gate Power Consumption

- For series-connected transistors, signal with lower activity should be on transistor closest to power supply rail



Given two signals A and B that are connected to the two equivalent inputs x_1 and x_2 of the gate in Figure 4.27, if A has much longer signal activity than B then A should be connected to x_2 and B to x_1. This results in lower power consumption.

capacitance at the gate output. For example, when the input to the gate in Figure 4.27 is ($x_1 = \downarrow, x_2 = 1, x_3 = 1$), there may be a rising transition at z_1 though y remains at 0. On the other hand, when the input is $x_1 = \uparrow, x_2 = 0$, $x_3 = 1$, there is a falling transition at both z_1 and y. Therefore, internal

$$W_{av} = V_{dd}^2 f\left(C_y d_y + \sum C_{z_i} d_{z_i}\right) \qquad (4.29)$$

Calculating Transition Probability

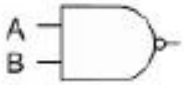
- Hard to find $p_{z_i}^{\updownarrow}$
  - ***Hard to determine prior state of internal circuit nodes***
  - ***Assume that when state cannot be determined, a transition occurred (upper power limit)***
  - ***More accurate bound: Observe that # conducting paths from node to V_{dd} must change from 0 to > 0 followed by similar change in # conducting paths to V_{ss}***
    - **Use # conducting paths that is smaller**
    - $$p_{z_i}^{\updownarrow} = \begin{cases} p_{z_i, V_{dd}}^{\uparrow} & \text{if } p_{z_i, V_{dd}}^{\uparrow} \leq p_{z_i, V_{ss}}^{\uparrow} \\ p_{z_i, V_{ss}}^{\uparrow} & \text{otherwise} \end{cases}$$
  - ***Use serial-parallel graph edge reduction techniques***

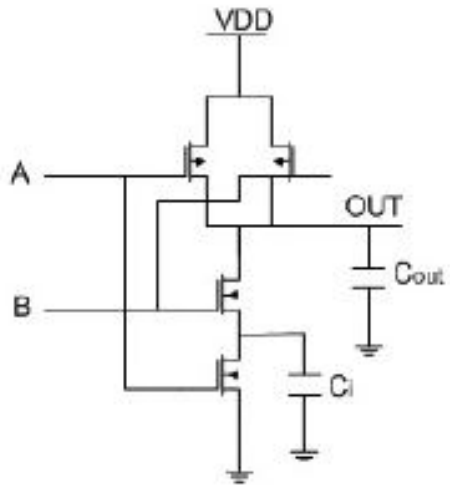
Transistor Reordering

- Already know delay of longest paths through each gate input from static timing analyzer
- Should (for NAND or NOR) connect latest arriving signal to input with smallest delay
  - ***Break gate inputs into permutable sets and swap inputs***
  - ***Hard to compute which input order is best – can afford to enumerate all possible orderings and try them***
    - **Compute prob. (signal is switching while all other signals in permutable set are on) – gives maximum internal node C charging / discharging**

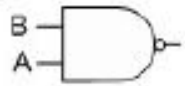
Equivalent pin ordering

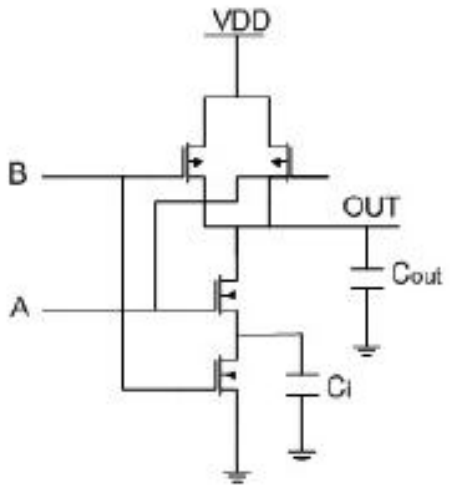
- Change the input connection based on the signal probability and signal activity
  - Suppose that B is high and A is switching from low to high





Charges in C_{out} and C_i are discharged





Charges in C_{out} are discharged

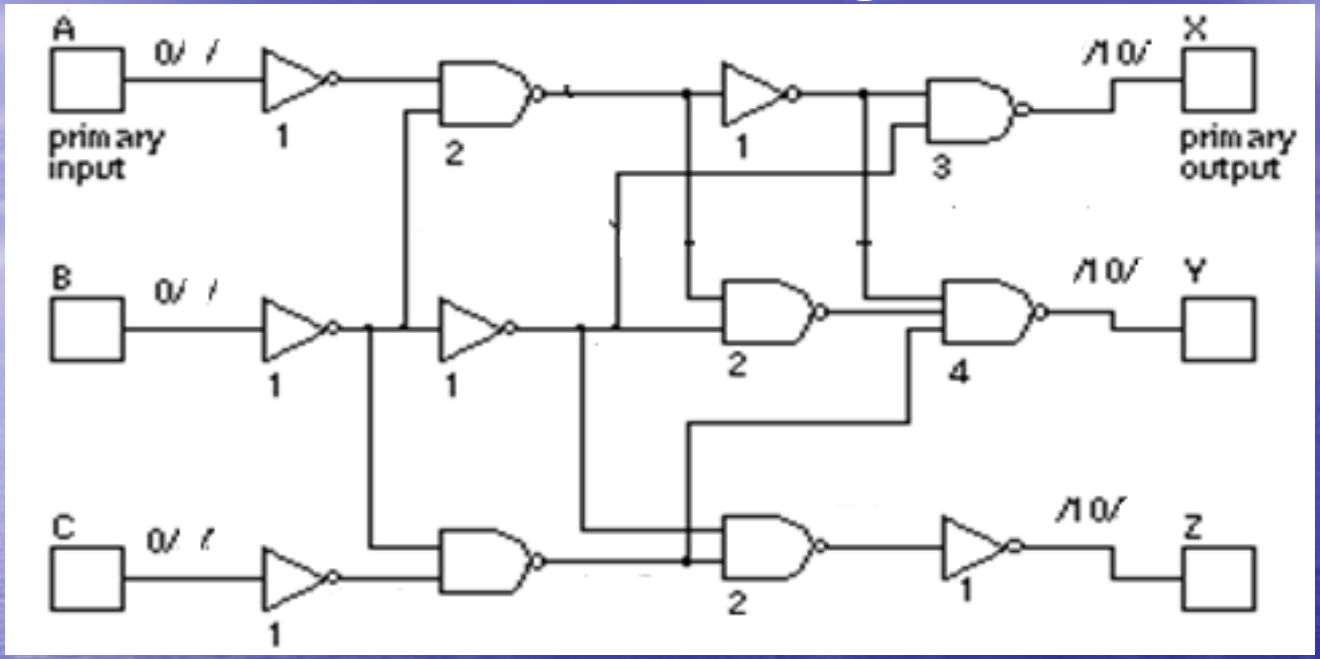
Optimization Algorithm

- Try to meet circuit performance goal (do forwards and then backwards graph traversal)
- During backwards traversal:
  - ***If a gate delay is larger than specified delay, reorder inputs to decrease delay***
  - ***End up with valid backwards delays for gates, but not valid forward delays***
- Repeat forward traversal if input reordering was done
  - ***Continue reordering inputs if gate path delay specification is exceeded***
- Continue alternating forward/backwards traversals until no more reorderings happen, then proceed to power minimization

Power Minimization

- Repeat alternating forward and backward traversals
- Change: Determine delay increase for input order corresponding to least estimated power dissipation
  - ***If increase less than available path slack, reorder inputs***
- Available slack: difference between:
  1. ***Larger of maximum acceptable delay and longest path delay***
  2. ***Delay of longest path through gate***
- Results on MCNC benchmarks – reduced power by 7 to 8 %, with no critical path delay increase, and very little area penalty

Zero Slack Algorithm



- Arrival time in forward direction from primary input to primary output.
- Choose Maximum arrival time so that all the inputs are reaching the gate. So the gate generate output.
- Required time in backward direction from primary output to primary input.
- Choose Minimum required time – how fast the output from gate reach the input of next gate. So that signal is propagated to next level.
- Slack= Difference between Maximum arrival time- Minimum required time

Zero Slack Algorithm

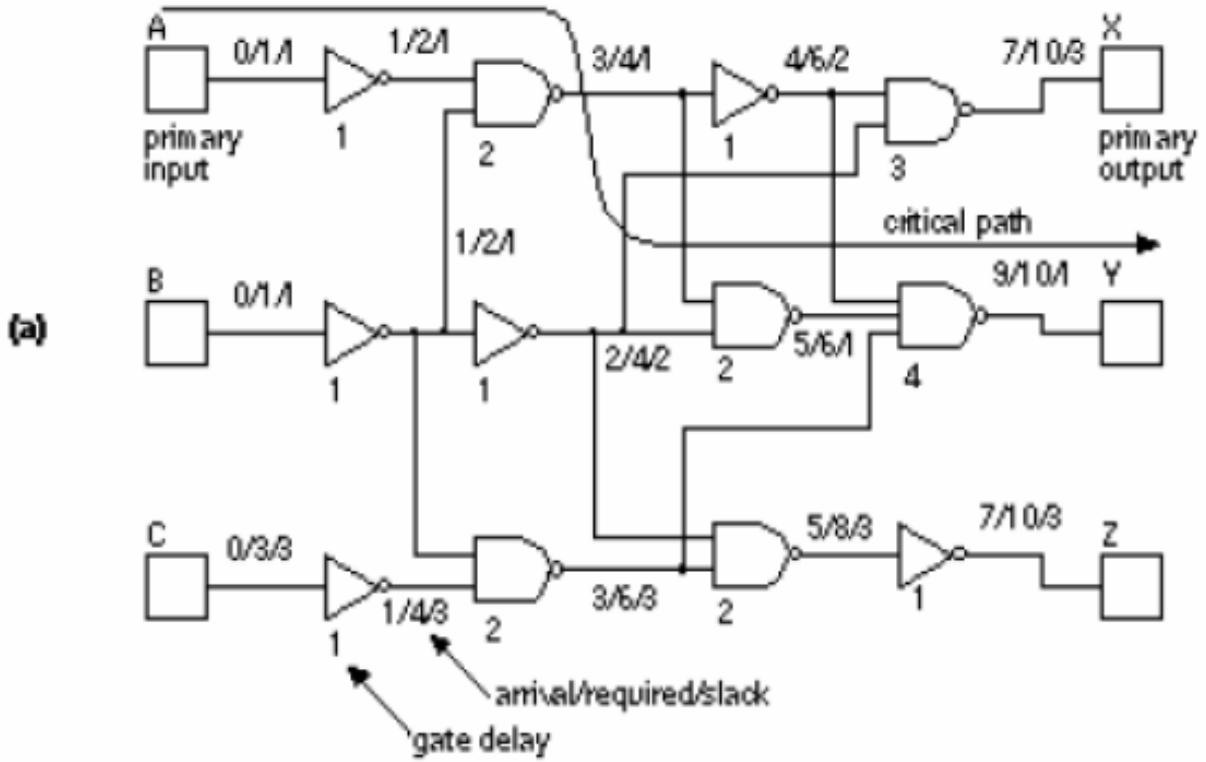


FIGURE 6: The zero-slack algorithm.
(a) The circuit with no net delays.

Zero Slack Algorithm

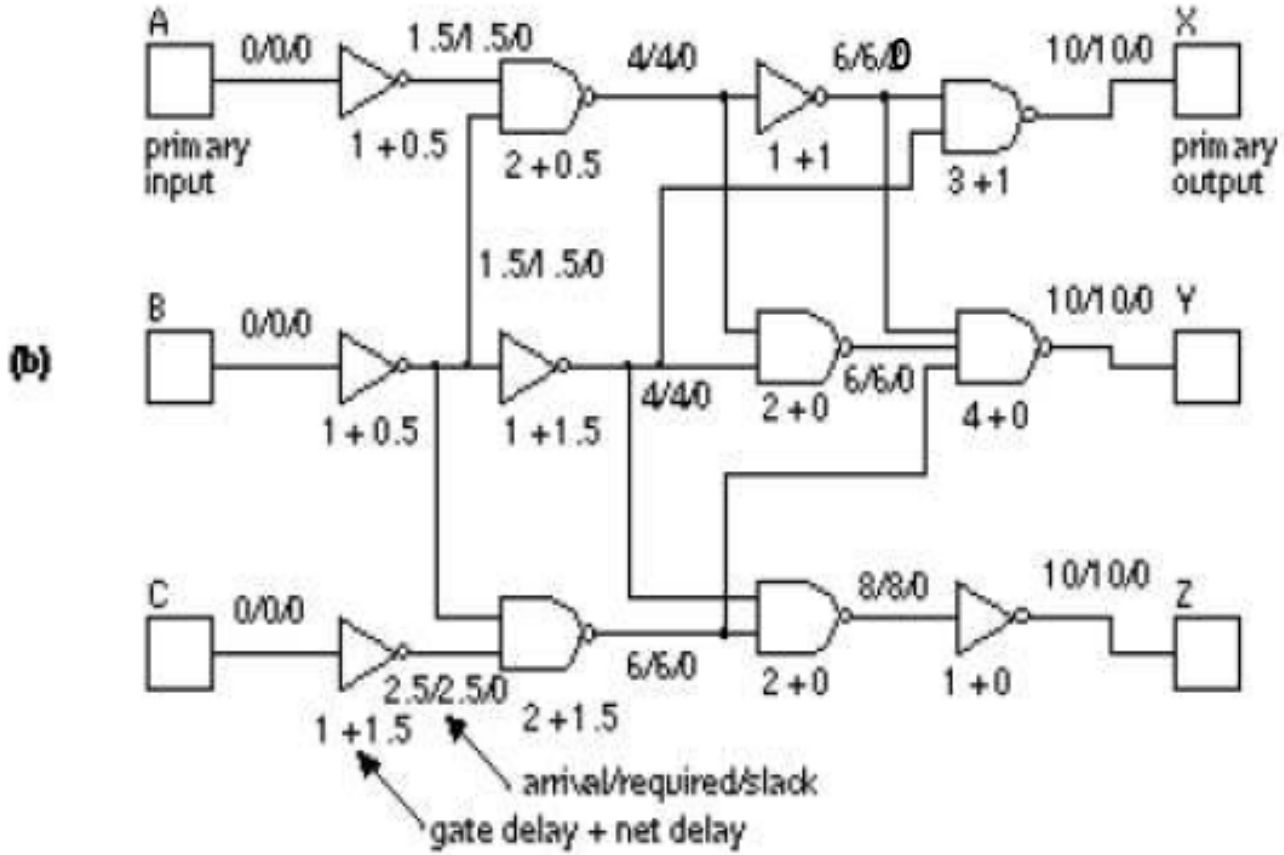


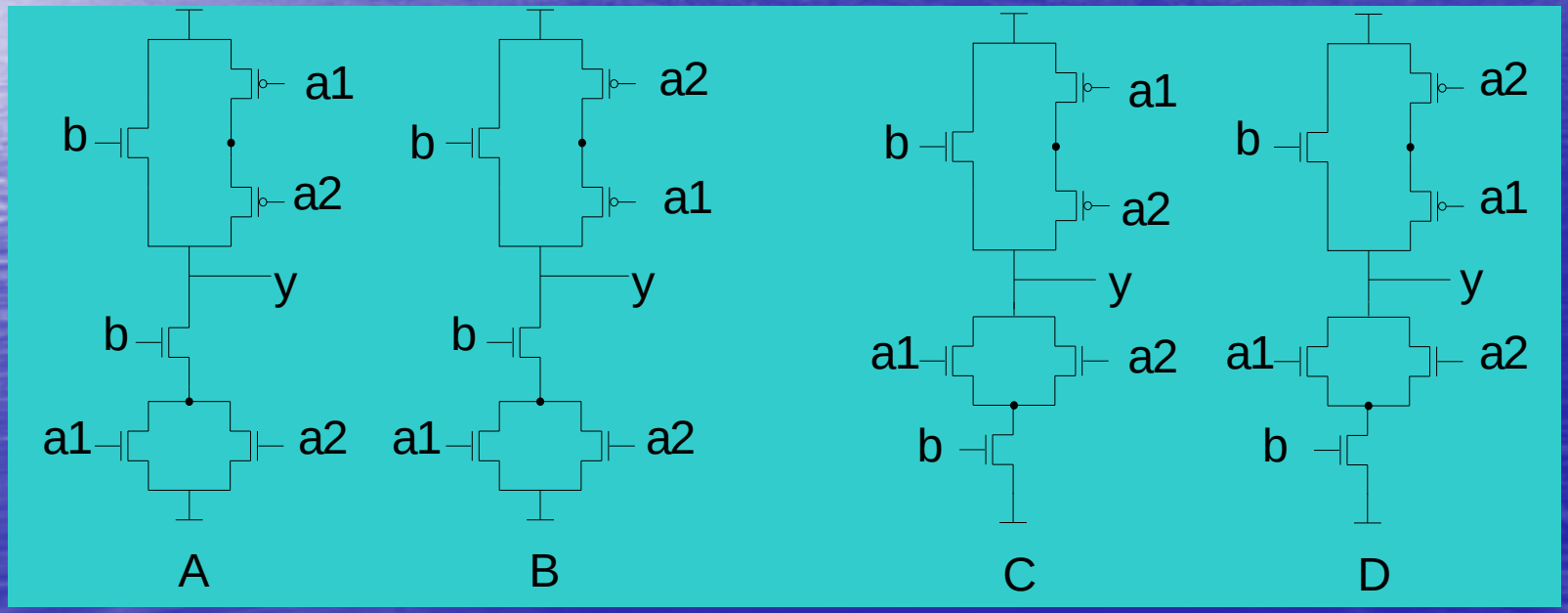
FIGURE 6: The zero-slack algorithm.

(b) The zero-slack algorithm adds net delays (at the outputs of each gate, equivalent to increasing the gate delay) to reduce the slack times to zero.

Transistor Reordering

- Logically equivalent CMOS gates may not have identical energy/delay characteristics

$$y = \overline{(a1 + a2)b}$$



Transistor Reordering cont'd

	Normalized $\mathbf{P_{dyn}}$				
Activity (transitions / s)	(A)	(B)	(C)	(D)	max. savings
(1) $A_{a1} = 10\ K$, $A_{a2} = 100\ K$, $A_b = 1\ M$					
(2) $A_{a1} = 1\ M$, $A_{a2} = 100\ K$, $A_b = 10\ K$					

➔ For given logic function and activity:
Signal with highest activity → closest to output
to reduce charging/discharging internal nodes

Transition Probabilities for CMOS Gates

Example: Static 2 Input NOR Gate

If A and B with same input signal probability:

Truth table of NOR2 gate

A	B	Out
1	1	0
0	1	0
1	0	0
0	0	1

$P_{A=1} = 1/2$
$P_{B=1} = 1/2$

Then:

$P_{Out=0} = 3/4$
$P_{Out=1} = 1/4$

$P_{0\rightarrow1}$

$C_{eff} = P_{0\rightarrow1} * C_L = 3/16 * C_L$

Transition Probabilities cont'd

- A and B with different input signal probability:
- P_A and P_B : Probability that input is 1
- P_1 : Probability that output is 1

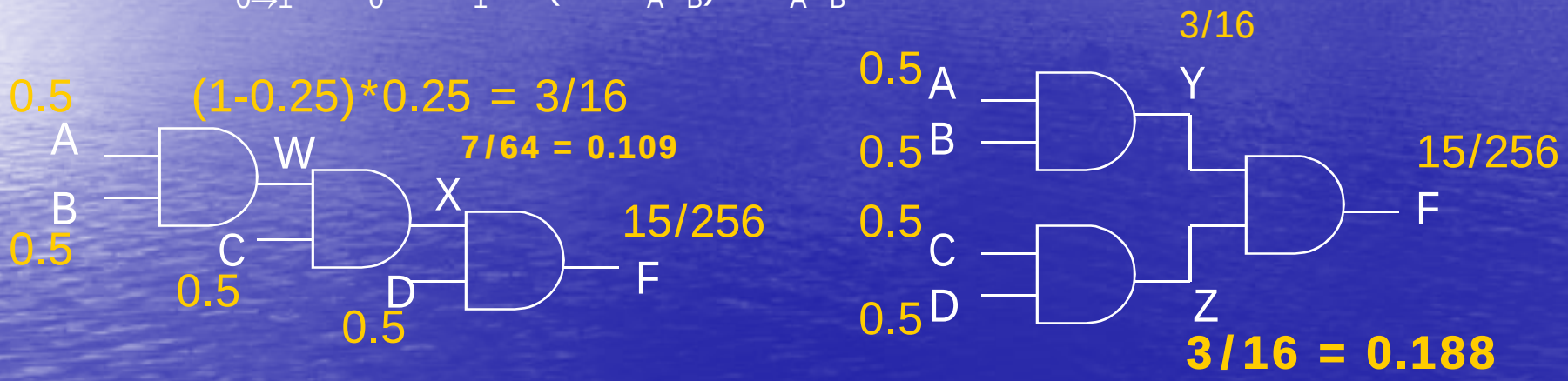
- Switching activity in CMOS circuits: $P_{0\rightarrow1} = P_0 * P_1$
- For 2-Input NOR: $P_1 = (1-P_A)(1-P_B)$
- Thus: $P_{0\rightarrow1} = (1-P_1)*P_1 = [1-(1-P_A)(1-P_B)]*[(1-P_A)][1-P_B]$

	$P_{0\rightarrow1} = P_{out=0} * P_{out=1}$
NOR	$(1 - (1 - P_A)(1 - P_B)) * (1 - P_A)(1 - P_B)$
OR	$(1 - P_A)(1 - P_B) * (1 - (1 - P_A)(1 - P_B))$
NAND	$P_AP_B * (1 - P_AP_B)$
AND	$(1 - P_AP_B) * P_AP_B$
XOR	$(1 - (P_A + P_B - 2P_AP_B)) * (P_A + P_B - 2P_AP_B)$

Logic Restructuring

- Logic restructuring: changing the topology of a logic network to reduce transitions

AND: $P_{0\to1} = P_0 * P_1 = (1 - P_A P_B) * P_A P_B$



➔ Chain implementation has a lower overall switching activity than tree implementation for random inputs

- BUT: Ignores glitching effects

Source: Timmernann, 2007

Input Ordering

(1-0.5x0.2)*(0.5x0.2)=**0.09**

(1-0.2x0.1)*(0.2x0.1)=**0.0196**

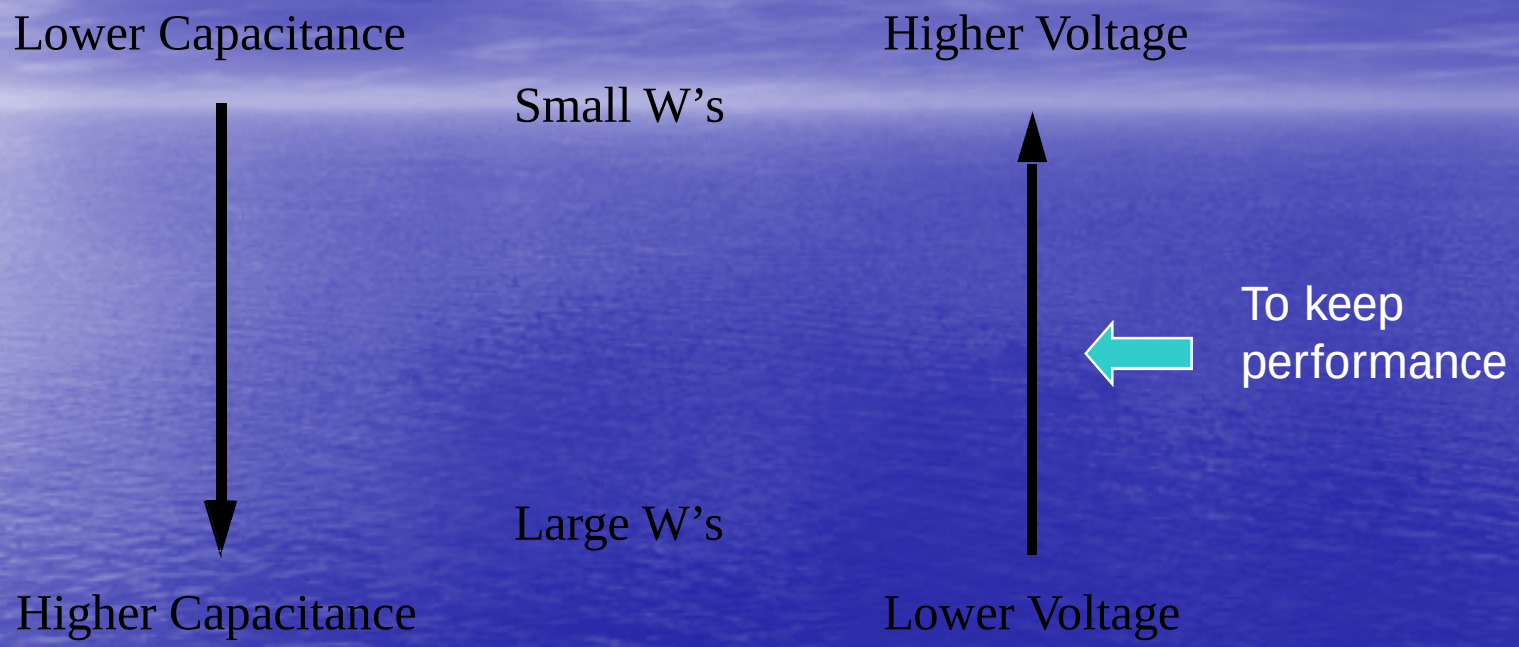
AND: $P_{0\rightarrow1} = (1 - P_A P_B) * P_A P_B$

Beneficial: postponing introduction of signals with a high transition rate (signals with signal probability close to 0.5)

Transistor Resizing Methods

- Datta, Nag & Roy: resized transistors on critical paths to reduce power and shorten delay
- Wider transistors speed up critical path and reduce power because you get sharper edges, and therefore less short-circuit power dissipation
  - ***Penalty – larger transistors increase node C, which can increase delay and power***
  - ***Increased drive for present block, and greater transition time for preceding block (due to larger load C_L) may increase present block short-circuit current***
  - ***Simulated annealing algorithm tries to optimize gates on N most critical paths***

Transistor Sizing for Power Minimization

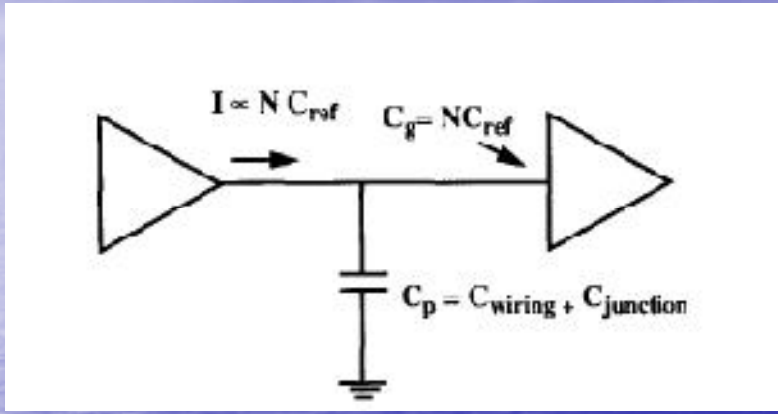


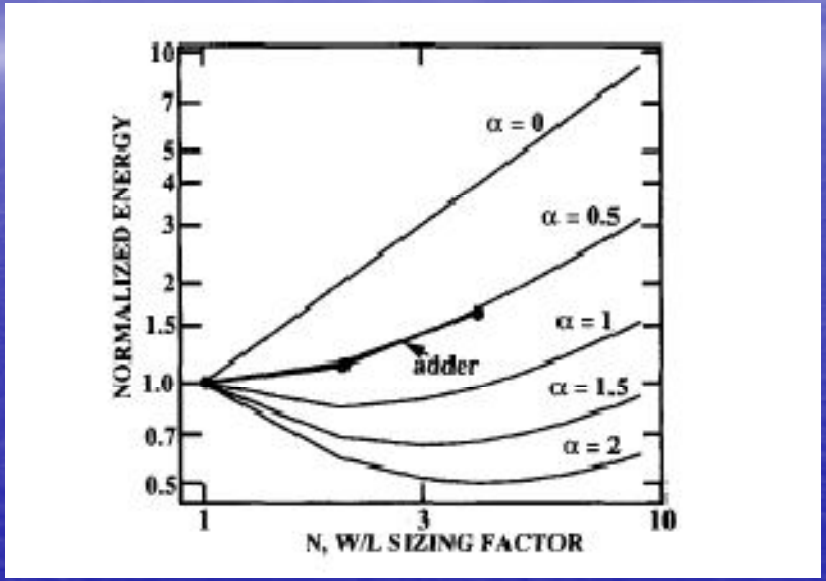
- Larger sized devices: only useful only when interconnects dominate
- Minimum sized devices: usually optimal for low-power

Source: Timmernann, 2007

Transistor Sizing

- Optimum transistor sizing





• The first stage is driving the gate capacitance of the second and the parasitic capacitance

• input gate capacitance of both stages is given by NCref, where Cref represents the gate capacitance of a MOS device with the smallest allowable (*W/L*)

Transistor Sizing

- When there is no parasitic capacitance contribution (i.e., $\alpha = 0$), the energy increases linearly with respect to N and the solution of utilizing devices with the smallest (W/L) ratios results in the lowest power.
- At high values of α, when parasitic capacitances begin to dominate over the gate capacitances, the power decreases temporarily with increasing device sizes and then starts to increase, resulting in a optimal value for N.
- The initial decrease in supply voltage achieved from the reduction in delays more than compensates the increase in capacitance due to increasing N.
- after some point the increase in capacitance dominates the achievable reduction in voltage, since the incremental speed increase with transistor sizing is very small
- Minimum sized devices should be used when the total load capacitance is not dominated by the interconnect

Summary

- Logic-level multi-level logic optimization is effective
  - ***State assignment***
  - ***Modified MIS algorithm***
- Logic-level Technology mapping
  - ***Tree-covering algorithm is effective***
- Circuit-level operations are effective
  - ***Transistor input reordering***
  - ***Transistor resizing***

HA Cell

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

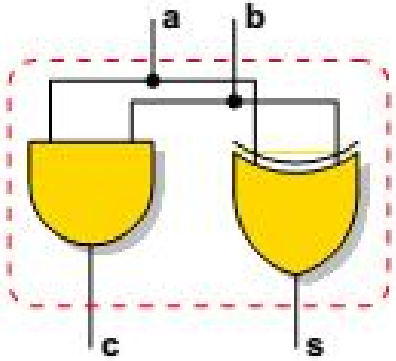
Arithmetic equation:

$$2c + s = a + b$$

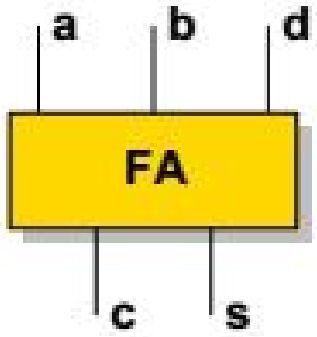
Logic equation:

$$s = a \oplus b$$
$$c = ab$$

Gate-level implementation of the HA:



FA Cell



a	b	d	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

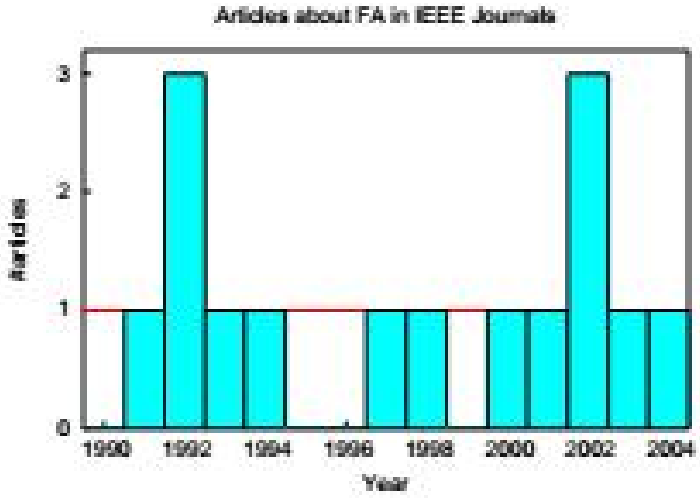
Arithmetic equation:

$$2c + s = a + b + d$$

Logic equation:

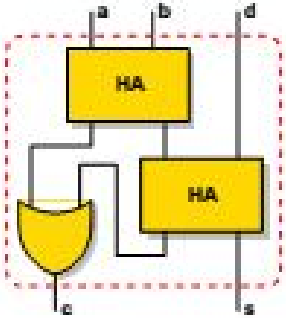
$$s = a \oplus b \oplus d$$

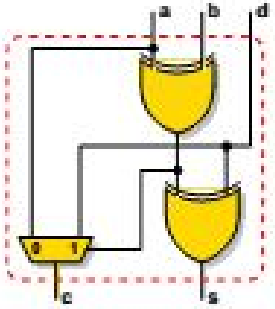
$$c = ab + ad + bd$$

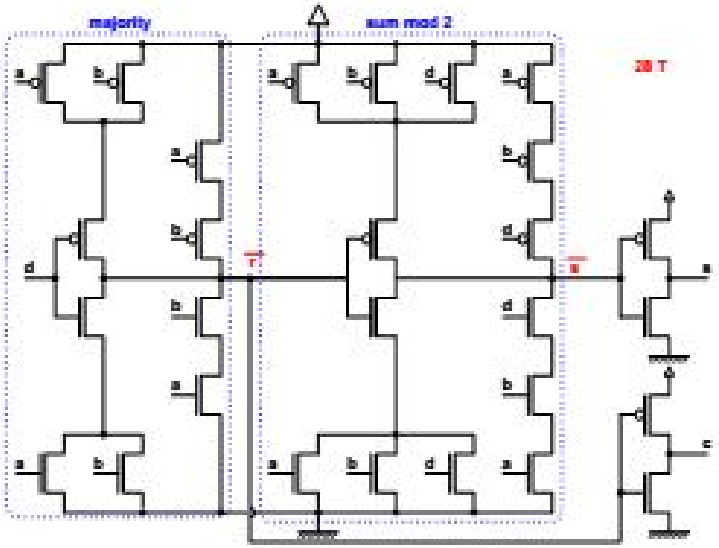


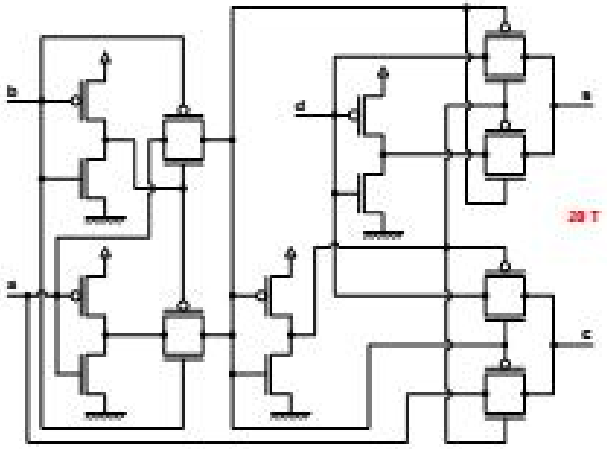
There many implementations of the FA cell

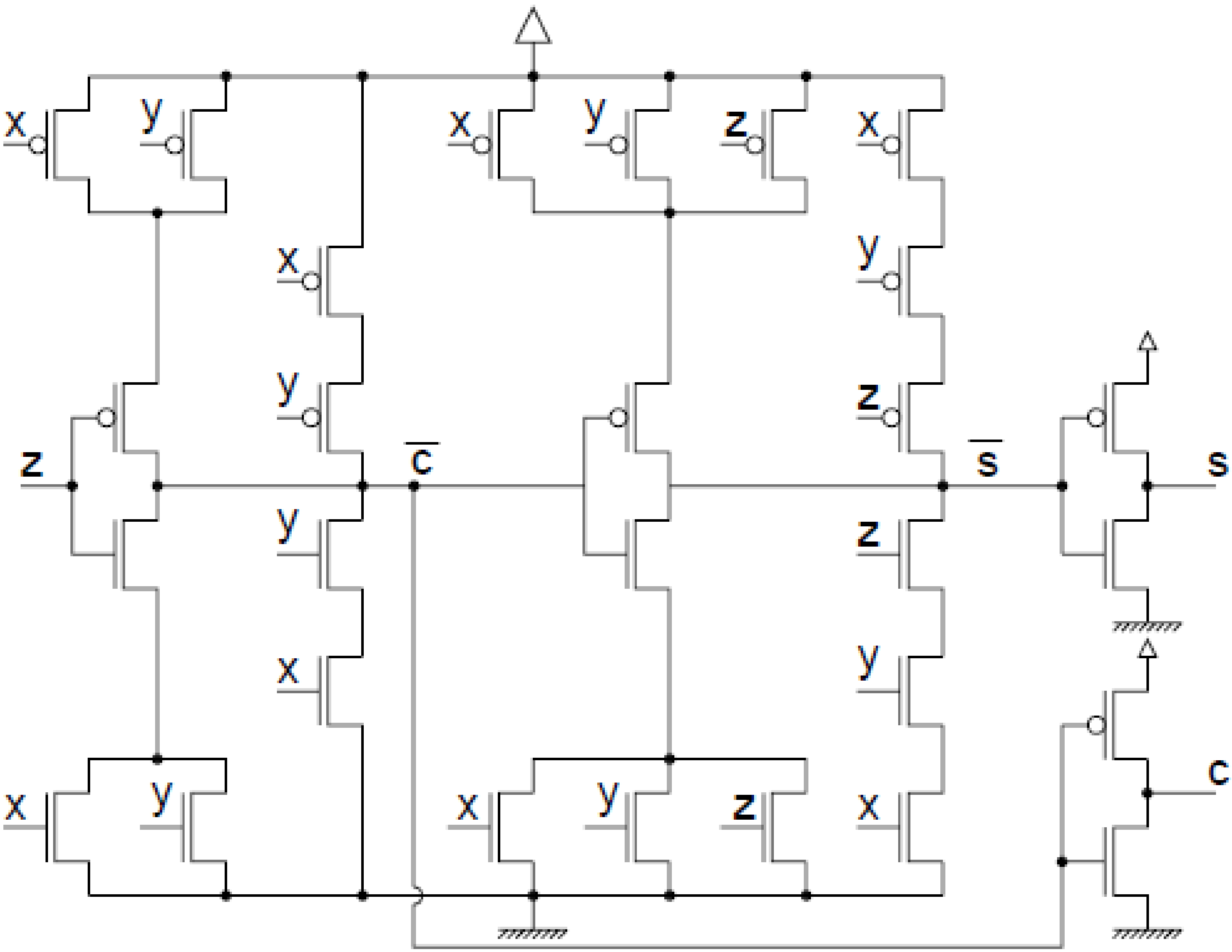
FA Implementations





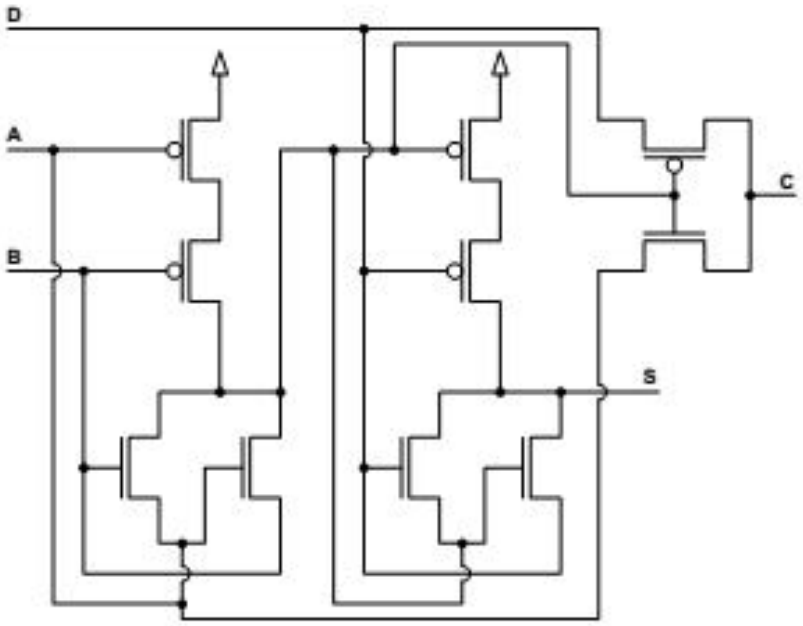





x
y
x
y
z
x
x
y
y
z
z
$\bar{c}$
$\bar{s}$
s
y
z
x
y
c
x
y
x
y
z
x

Example of Low-Power FA Cell

10-transistor solution[1] (some output signals are weak signals):

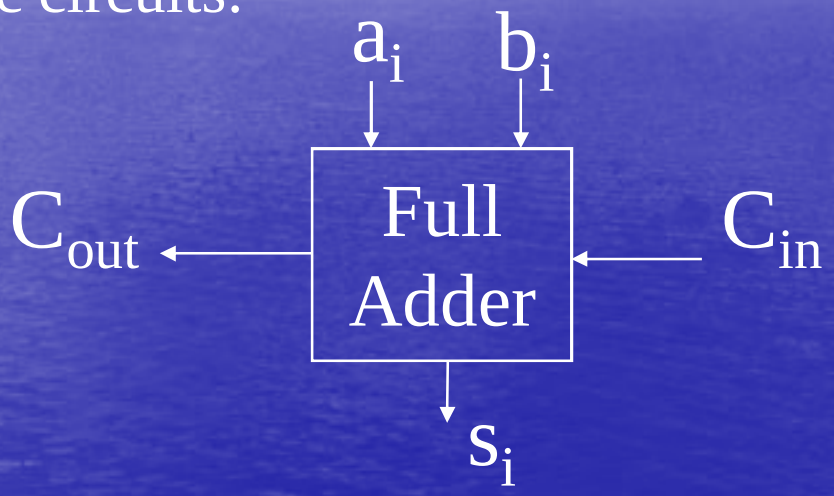


A	B	D	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0_w	1
0	1	1	1	0
1	0	0	0_w	1
1	0	1	1_w	0
1	1	0	1	0
1	1	1	1_w	1_w

[1] H. T. Bui, Y. Wang and Y. Jiang. *Design and analysis of low-power 10-transistor full adders using novel XOR–XNOR gates.* IEEE Trans. CaS, jan. 2002.

Addition of Binary Numbers

Full Adder. The full adder is the fundamental building block of most arithmetic circuits:

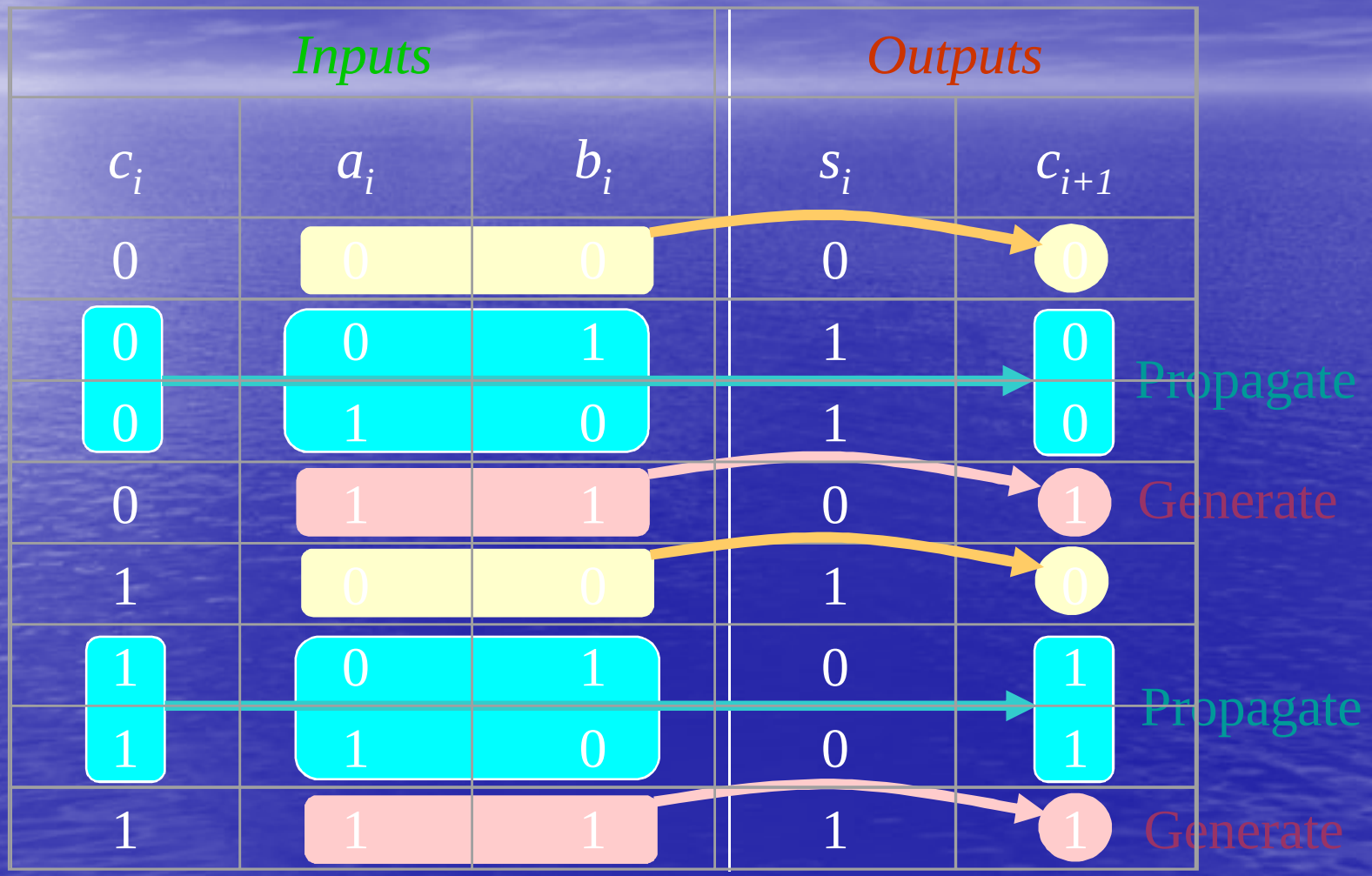


The sum and carry outputs are described as:

$$s_i = a_i \bar{b}_i \bar{c}_i + \bar{a}_i b_i \bar{c}_i + \bar{a}_i \bar{b}_i c_i + a_i b_i c_i$$

$$c_{i+1} = \bar{a}_i b_i c_i + a_i \bar{b}_i c_i + a_i b_i \bar{c}_i + a_i b_i c_i = a_i b_i + a_i c_i + b_i c_i$$

Addition of Binary Numbers



Full-Adder Implementation

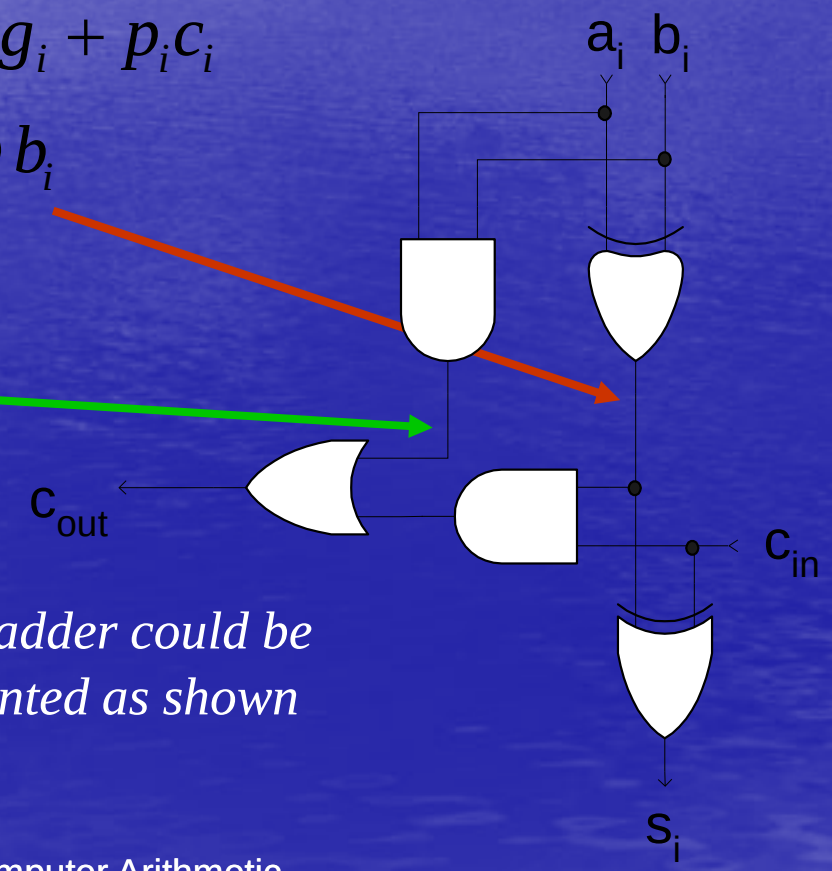
Full Adder operations is defined by equations:

$$s_i = a_i\bar{b}_i\bar{c}_i + \bar{a}_i b_i\bar{c}_i + \bar{a}_i\bar{b}_i c_i + a_i b_i c_i = a_i \oplus b_i \oplus c_i = p_i \oplus c_i$$

$$c_{i+1} = \bar{a}_i b_i c_i + a_i\bar{b}_i c_i + a_i b_i = g_i + p_i c_i$$

Carry-Propagate: $p_i = a_i \oplus b_i$
and Carry-Generate g_i

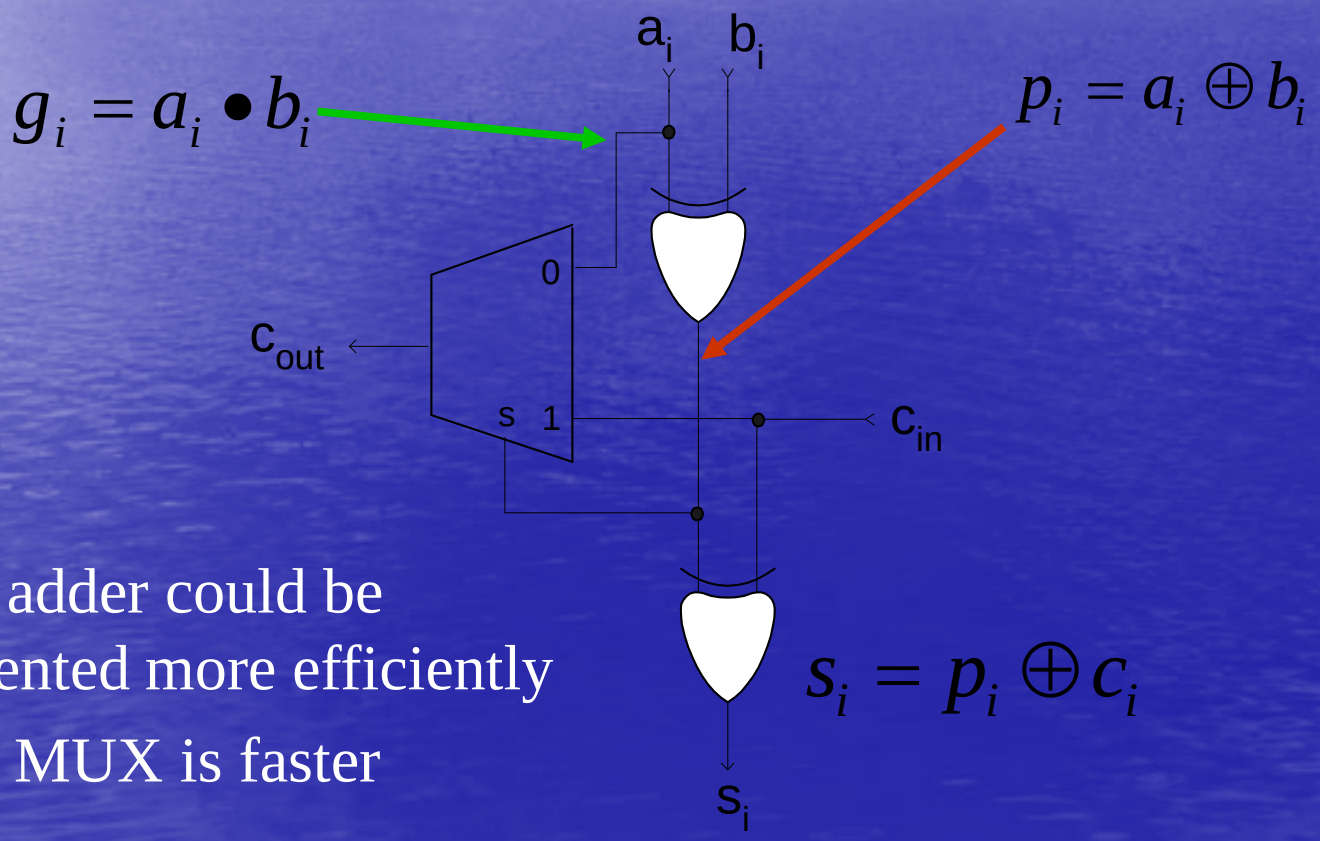
$$g_i = a_i \bullet b_i$$



One-bit adder could be implemented as shown

High-Speed Addition

$$c_{i+1} = g_i + p_i c_i$$



One-bit adder could be implemented more efficiently because MUX is faster

Array Multipliers with Lower Power Consumption

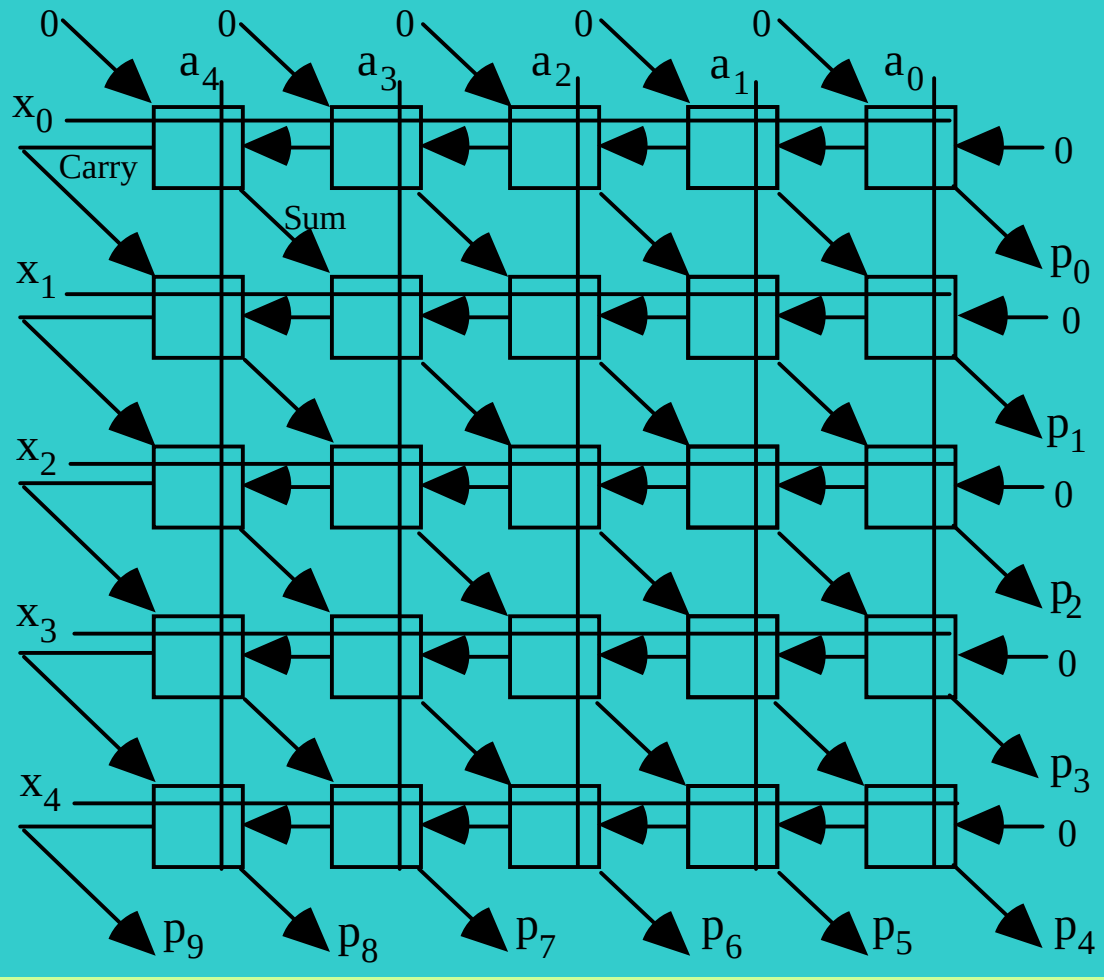
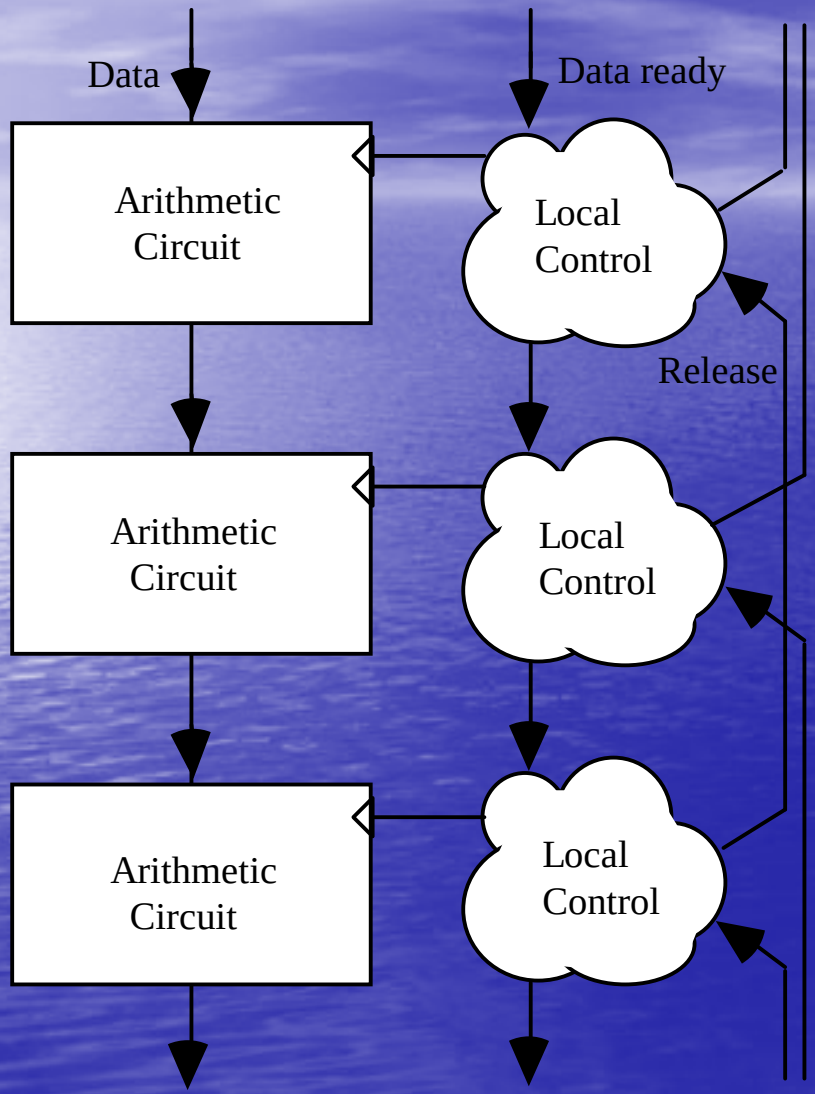


Fig. 26.5 An array multiplier with gated FA cells.

New and Emerging Methods



Dual-rail data encoding with transition signaling:

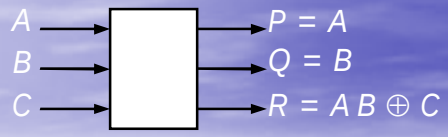
Two wires per signal

Transition on wire 0 (1) indicates the arrival of 0 (1)

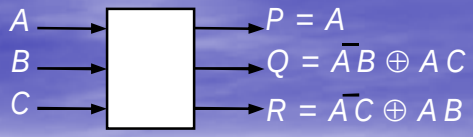
Dual-rail design does increase the wiring density, but it offers the advantage of complete insensitivity to delays

Part of an asynchronous chain of computations.

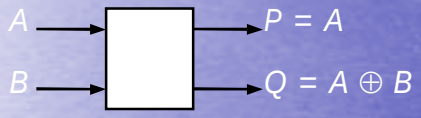
The Ultimate in Low-Power Design



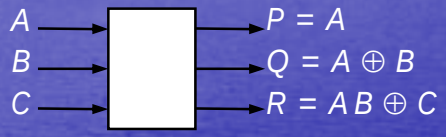
(a) Toffoli gate



(b) Fredkin gate

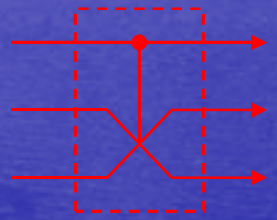


(c) Feynman gate

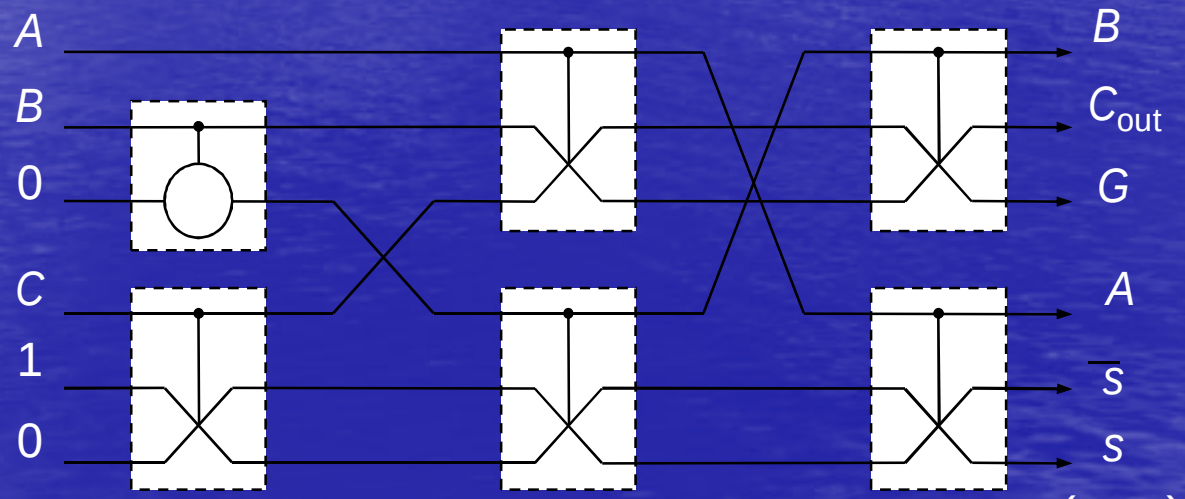


(d) Peres gate

Some reversible logic gates.



Reversible binary full adder built of 5 Fredkin gates, with a single Feynman gate used to fan out the input B. The label "G" denotes "garbage."



Problem 3.1 (5 points) **And-Or-Invert Gates:** And-Or-Invert (AOI) gates are often included in standard cell libraries to reduce the area of synthesized combinational logic because they implement common logic functions with fewer devices. Draw a static CMOS gate which implements the AOI function $Y = \overline{A \cdot (B + C + D)}$.

Problem 3.2 (10 points) **Activity Factors for And-Or-Invert Gates:** Write down the truth table for the AOI gate for Y. Determine the transition activities $\alpha_{0\to1}$ of the output assuming the inputs are independent and uniformly distributed.

Problem 3.3 (25 points) **And-Or-Invert Gates versus Basic Gates:** (1) Draw the simplest possible implementation of the logic function Y using 2-input basic gates (NOR, OR, NAND, AND, XOR). (2) Assuming the inputs are independent and their probabilities of equaling 1 are 0.5, derive the activity factors for the outputs and any internal nodes of the 2-input gate implementation based on the transition probabilities formulas from Lecture 3. Assume the self-loading capacitance of a gate is the same as its input capacitance (i.e., both are equal to some unit capacitance C_u), and all gates are equivalent in terms of capacitance. (3) How much more efficient is the AOI implementation than the 2-input gate implementation in terms of *effective* capacitance (write answer in units of C_u)?

3.1

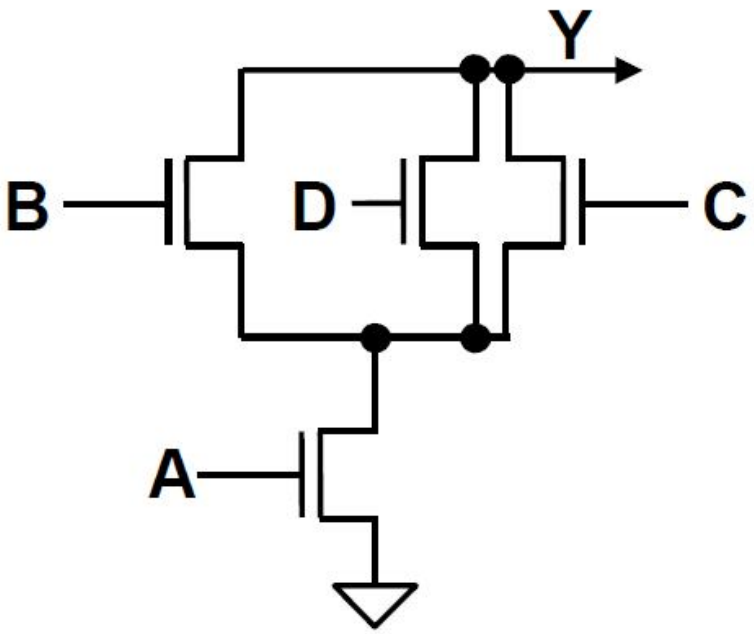


Figure 4: Y-output AOI gate pulldown network.

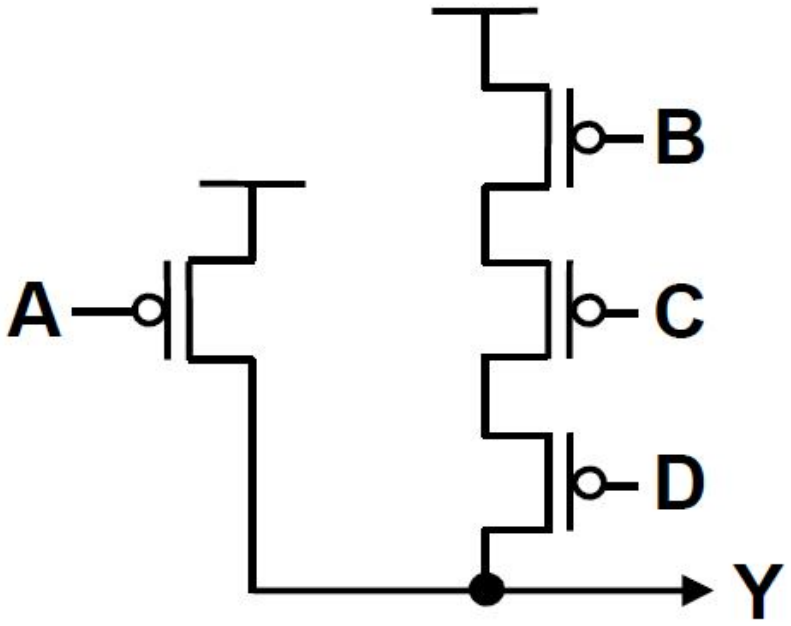


Figure 5: Y-output AOI gate pullup network.

3.2

	A	B	C	D	Y
	0	0	0	0	1
	0	0	0	1	1
	0	0	1	0	1
	0	0	1	1	1
	0	1	0	0	1
	0	1	0	1	1
	0	1	1	0	1
3.2	0	1	1	1	1
	1	0	0	0	1
	1	0	0	1	0
	1	0	1	0	0
	1	0	1	1	0
	1	1	0	0	0
	1	1	0	1	0
	1	1	1	0	0
	1	1	1	1	0

Table 3: Truth table for function $Y = \overline{A \cdot (B + C + D)}$.

Given the logic equation for Y, computing the transition activity factor $\alpha_{0\to1}$ is easy once the truth table for output Y is determined. This is shown in Table 3. Using the equation from lecture, the transition probability is:

$$\alpha_{0\to1} = \frac{N_0}{2^N}\frac{N_1}{2^N} \qquad \alpha_{0\to1}(Y) = \frac{7}{16} \times \frac{9}{16} = \frac{63}{256}$$

where N_0 and N_1 are the numbers of 0's and 1's in the output column of the truth table.

3.3

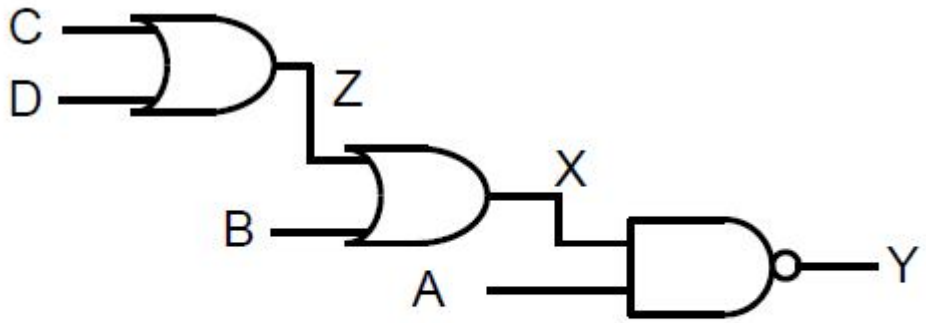


Figure 6: Y-output two input gate implementation.

The AOI gate implementation for the Y equation has only approximately 30% of the switched capacitance of the two input gate implementation. AOI gates are generally quite efficient for implementing miscellaneous logic equations, hence they are often included in standard cell libraries designed for synthesis (**10 points**).

3.3

$$
\begin{aligned}
P_{0\to 1}(Z) &= (1-p_D)(1-p_C)[1-(1-p_D)(1-p_C)] \\
&= (1-0.5)(1-0.5)[1-0.5\cdot 0.5] \\
&= \frac{3}{16} = 0.1875 \\
p_Z &= \frac{N_1}{4} = \frac{3}{4} = 0.75
\end{aligned}
$$

$$
\begin{aligned}
P_{0\to 1}(X) &= (1-p_B)(1-p_Z)[1-(1-p_B)(1-p_Z)] \\
&= (1-0.5)(1-0.75)[1-0.5\cdot 0.25] \\
&= \frac{7}{64} = 0.1094 \\
p_X &= [1-(1-p_B)(1-p_Z)] = 1-\frac{1}{8} = 0.875 \\
P_{0\to 1}(Y) &= p_A p_X (1-p_A p_X) \\
&= 0.5\cdot 0.875(1-0.5\cdot 0.875) \\
&= \frac{63}{256} = 0.2461
\end{aligned}
$$

3.3

From the transition activity factors calculated earlier, the effective capacitances for the Y AOI gate is $C_{sw}(Y) = \alpha_{0\to1}C_u = \frac{63}{256}C_u$. The effective capacitances are $0.241C_u$. This is based on the self-loading or output capacitances of the gates equalling C_u. The two input gates have to include the activity and the capacitances of the intermediate nodes as well as the output. These are summarized below:

$$\begin{aligned} C_{sw}(Y) &= \alpha_{0\to1}(Z)(2C_u) + \alpha_{0\to1}(X)(2C_u) + \alpha_{0\to1}(Y)C_u \\ &= 0.8399C_u \end{aligned}$$