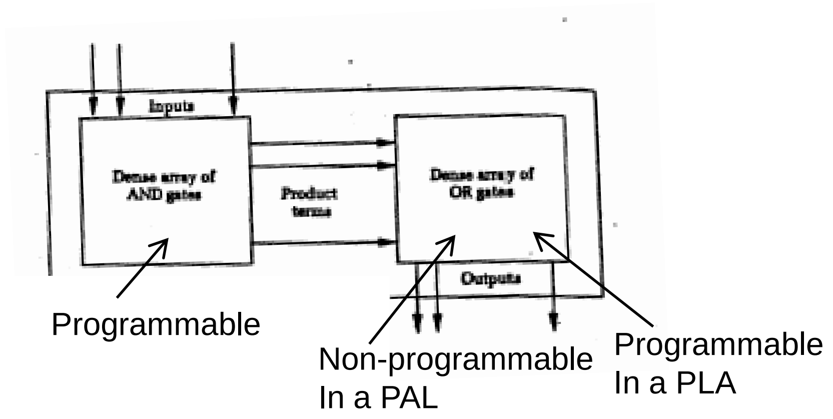
PLA/PALs and PLA Design

Programmable Logic Arrays (PLAs) and Programmable Array Logics (PALs)



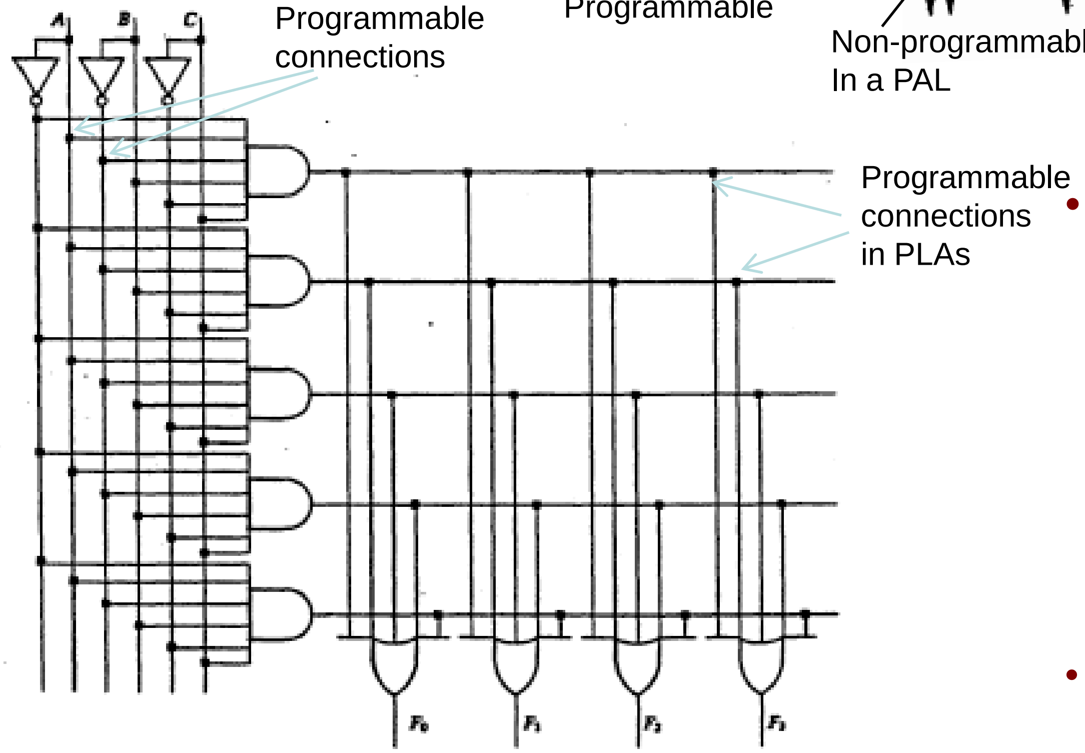


Figure PLA before programming.

- PLA/PAL advantages
  - Ease in circuit implementation (don't have to worry about placing and routing individual gates on a chip)
  - The wiring is well-patterned and regular; thus easy to pre-estimate delay and area costs of the PLAs/PALs on a chip
- Disadvantage:
  - Not CMOS, so higher power consumption (esp. when o/p = 0)

A functional schematic; the implementation is more streamlined

PLAs and PALs (Contd)

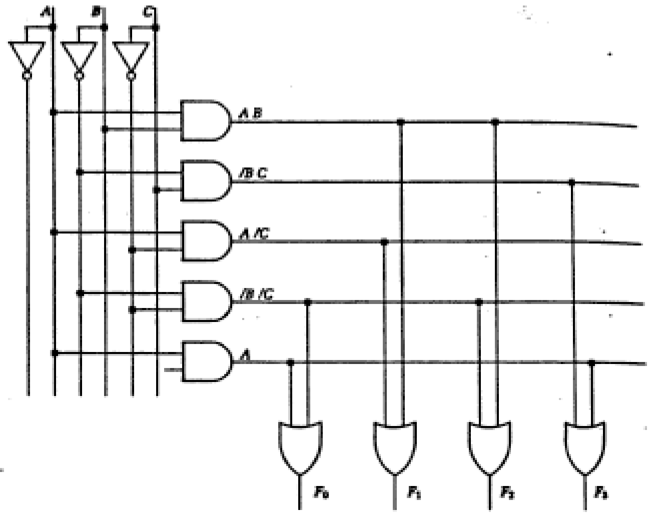


Figure . PLA after programming.

A more streamlined functional schematic:

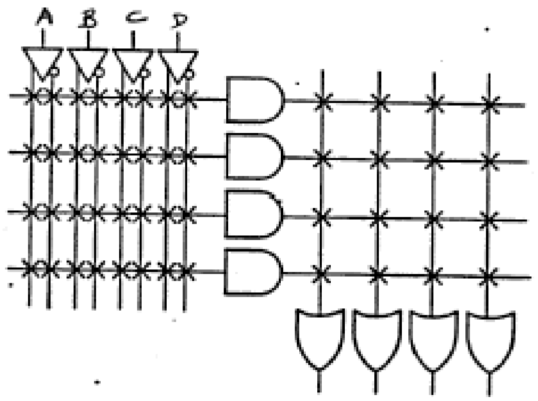


Figure Notation for four-input, four-output, four-product-term array.

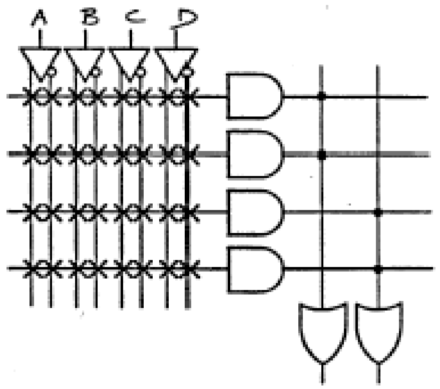


Figure . Example of a PAL organization with a constrained OR array

– TO PROGRAM REMOVE THE X'S FROM THE AND ARRAY CORRESPONDING TO LITERALS NOT NEEDED IN THE PRODUCT TERM. SIMILARLY, REMOVE X'S IN THE OR ARRAY (OF PLAs) FOR PRODUCT TERMS NOT USED FOR A FUNCTION (O/P OF AN OR GATE, LINE).

PLAs (Contd)

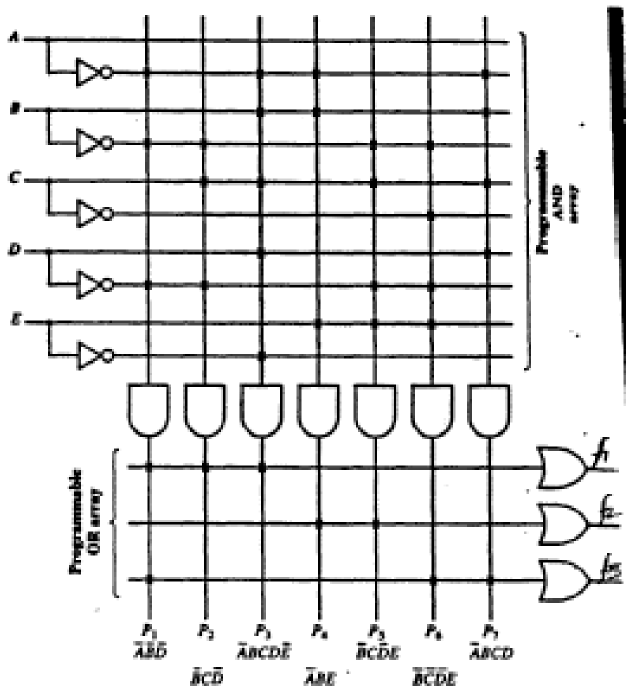
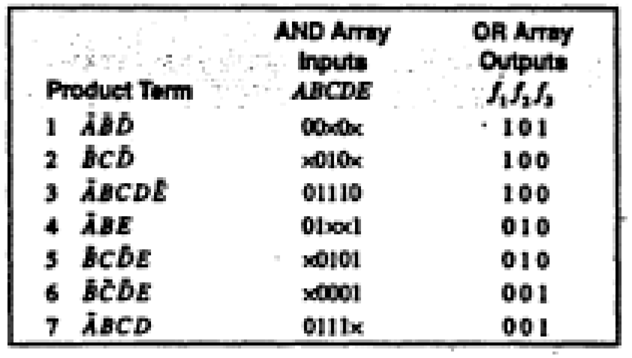


Figure 5.7 PLA for Example 5.1.

Alternative notation (Hor top lines are literal and bottom lines are OR lines; vertical lines are And lines)

TABLE 5.1 PLA TABLE FOR EXAMPLE 5.1



Product Term		AND Array Inputs $ABCDE$	OR Array Outputs $f_1 f_2 f_3$
1	$\bar{A}\bar{B}\bar{D}$	00x0x	1 0 1
2	$\bar{B}C\bar{D}$	x010x	1 0 0
3	$\bar{A}BCD\bar{E}$	01110	1 0 0
4	$\bar{A}BE$	01xx1	0 1 0
5	$\bar{B}C\bar{D}E$	x0101	0 1 0
6	$\bar{B}\bar{C}\bar{D}E$	x0001	0 0 1
7	$\bar{A}BCD$	0111x	0 0 1

PLAs: NOR-NOR Implementation

$z_1 = ab + \overline{\bar{a}\bar{b}c}$
$z_2 = ab$
$z_3 = a + \bar{b}c$

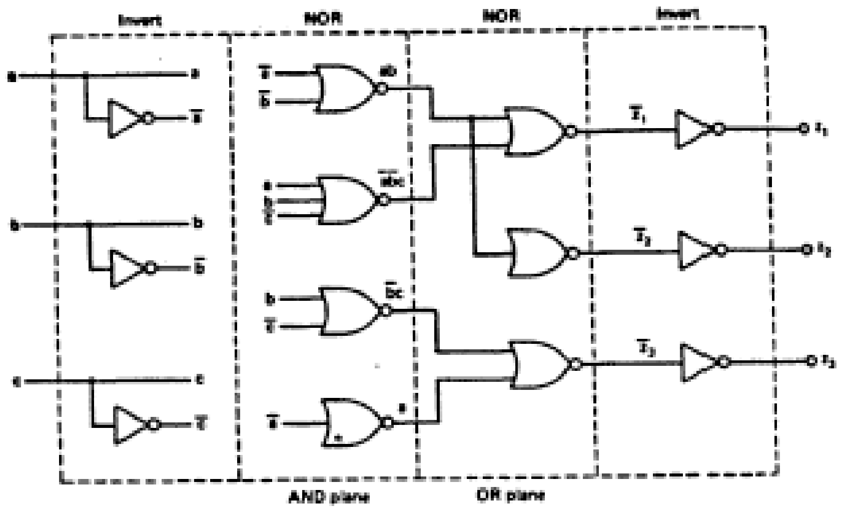


Figure General functional structure of a NOR–NOR PLA.

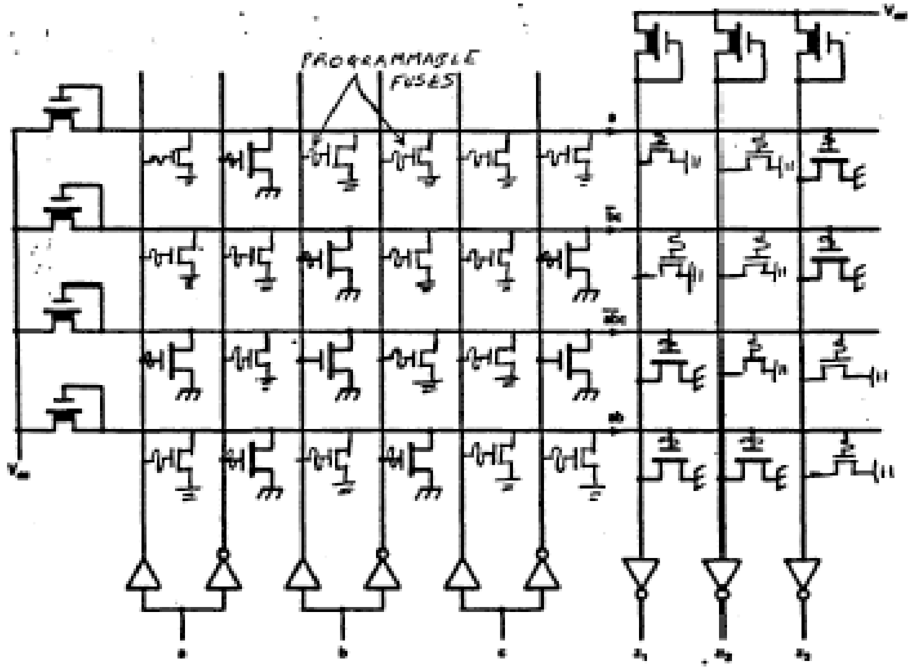
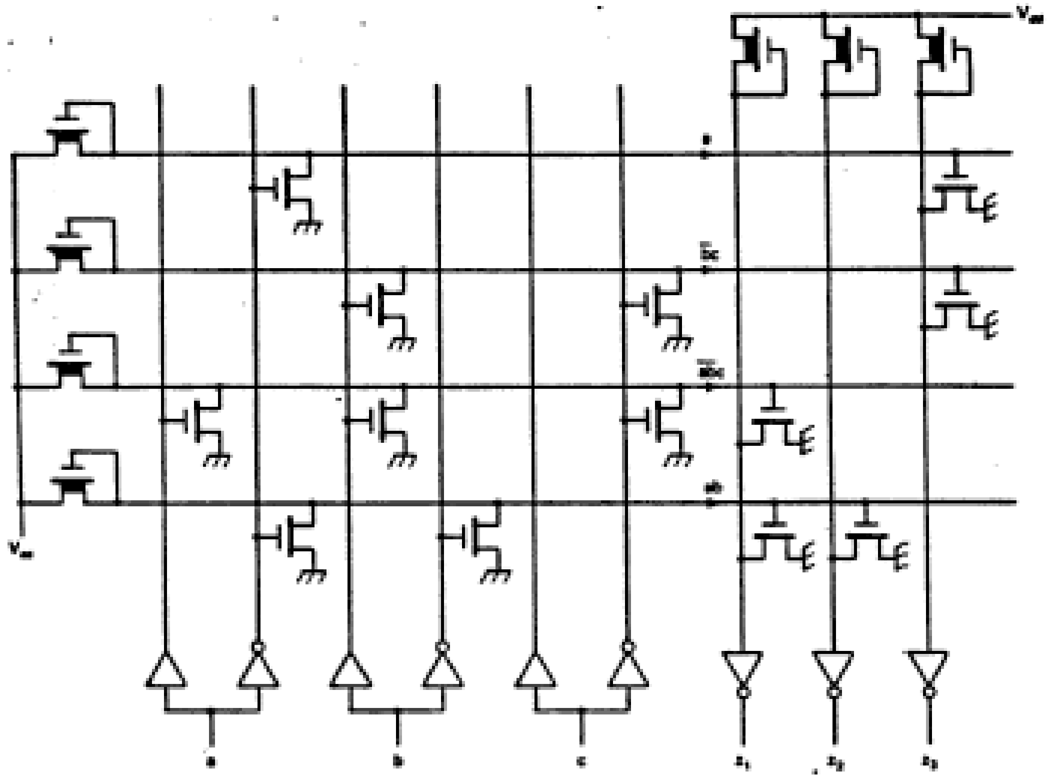


Figure Circuit schematic for an nMOS PLA.

PLAs (Contd)

nMOS PLA PROGRAMMING FOR: $z_1 = ab + \bar{a}\bar{b}c$; $z_2 = ab$

$z_3 = a + \bar{b}c$

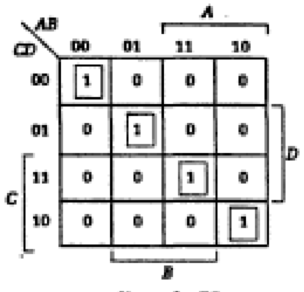


PLAs (Contd)

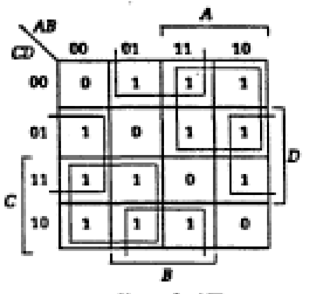
Two-Bit Magnitude Comparator Our next task is to design a comparator circuit. The circuit takes two 2-bit binary numbers as inputs, denoted by *AB* and *CD*, and computes the four functions $AB = CD$ (*EQ*), $AB \neq CD$ (*NE*), $AB < CD$ (*LT*), and $AB > CD$ (*GT*).

Figure shows the K-maps for the four functions with boxed prime implicants. This yields the following reduced equations for the output functions:

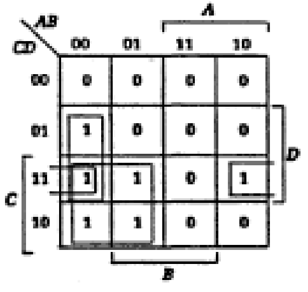
$$EQ = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}B\bar{C}D + ABCD + A\bar{B}C\bar{D}$$
$$NE = A\bar{C} + \bar{A}C + \bar{B}D + B\bar{D}$$
$$LT = \bar{A}C + \bar{A}\bar{B}D + \bar{B}CD$$
$$GT = A\bar{C} + AB\bar{D} + B\bar{C}\bar{D}$$



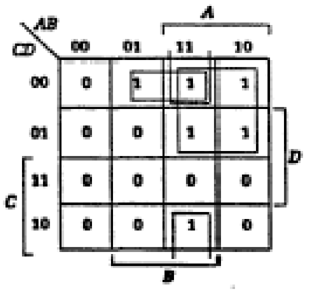
K-map for *EQ*



K-map for *NE*



K-map for *LT*



K-map for *GT*

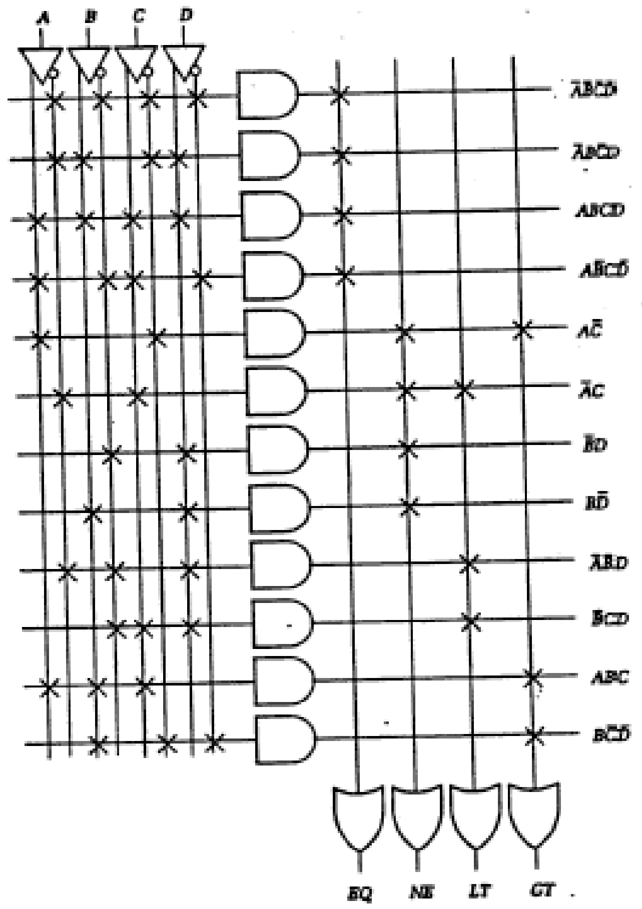
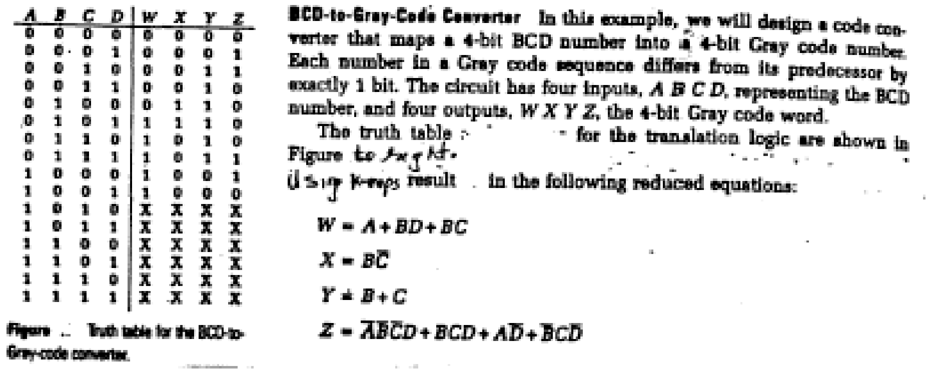


Figure PLA implementation of magnitude comparator.

Programmable Array Logic (PALs)



A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	1	0	0	1
1	0	0	1	1	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

Figure .. Truth table for the BCD-to-Gray-code converter.

BCD-to-Gray-Code Converter In this example, we will design a code converter that maps a 4-bit BCD number into a 4-bit Gray code number. Each number in a Gray code sequence differs from its predecessor by exactly 1 bit. The circuit has four inputs, *A B C D*, representing the BCD number, and four outputs, *W X Y Z*, the 4-bit Gray code word.

The truth table for the translation logic are shown in Figure to right.
Using K-maps result in the following reduced equations:

$$W = A + BD + BC$$

$$X = B\overline{C}$$

$$Y = B + C$$

$$Z = \overline{A}B\overline{C}D + BCD + A\overline{D} + \overline{B}C\overline{D}$$

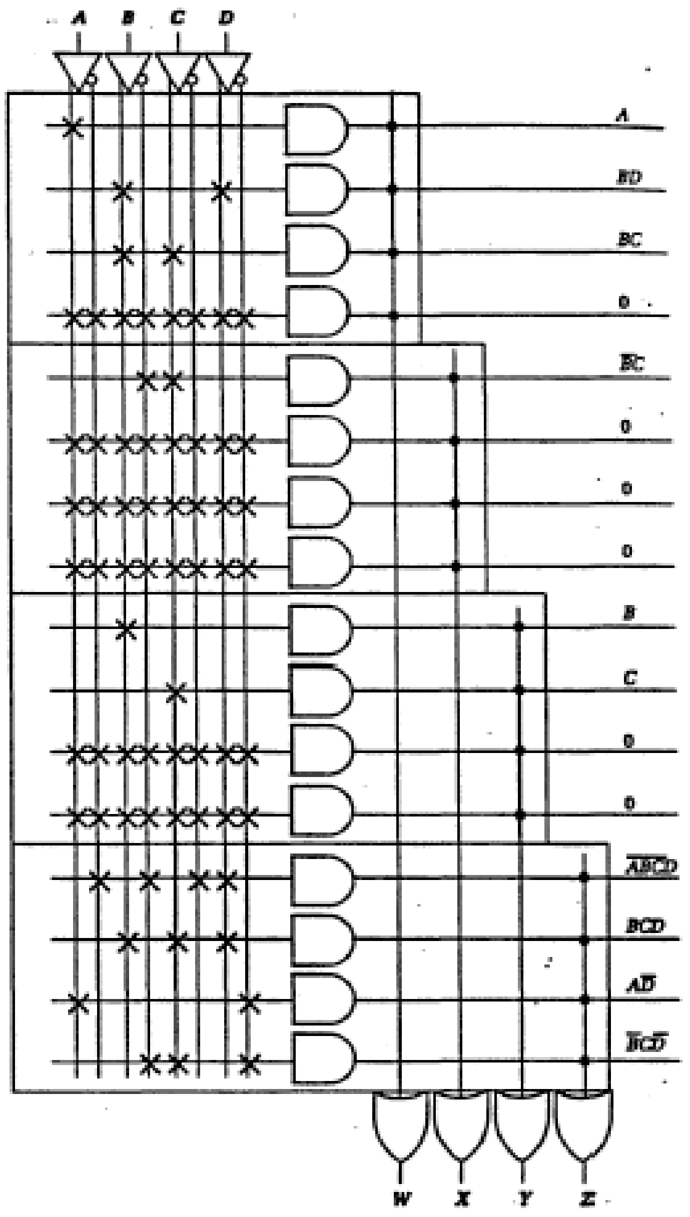
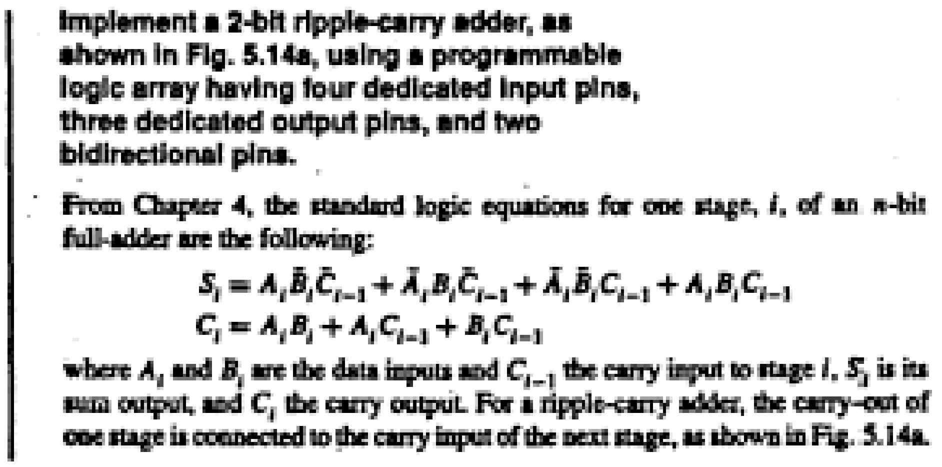


Figure PAL Implementation of BCD-to-Gray-code converter.

Comments:

- More AND lines needed than called for by the SOP expressions (and hence than in a PLA) due to the static hardwired nature of the OR plane
- A little faster than in the OR plane than PLAs due to the pre-charging phase seeing only one drain capacitance (of one nMOS) on each OR line

PLAs with Feedback

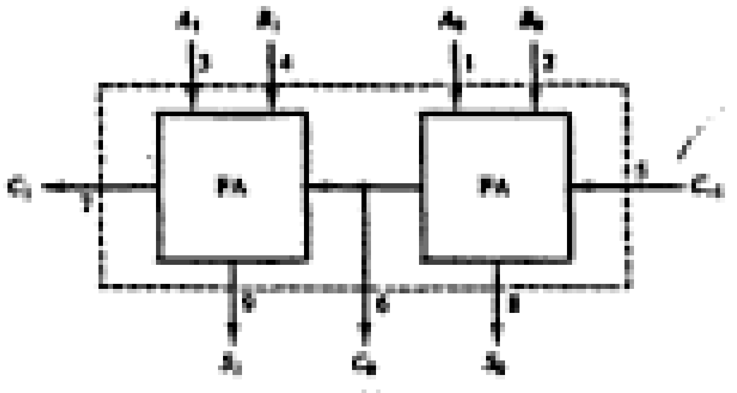


Implement a 2-bit ripple-carry adder, as shown in Fig. 5.14a, using a programmable logic array having four dedicated input pins, three dedicated output pins, and two bidirectional pins.

From Chapter 4, the standard logic equations for one stage, i, of an n-bit full-adder are the following:

$$S_i = A_i\bar{B}_i\bar{C}_{i-1} + \bar{A}_iB_i\bar{C}_{i-1} + \bar{A}_i\bar{B}_iC_{i-1} + A_iB_iC_{i-1}$$
$$C_i = A_iB_i + A_iC_{i-1} + B_iC_{i-1}$$

where A_i and B_i are the data inputs and C_{i-1} the carry input to stage i, S_i is its sum output, and C_i the carry output. For a ripple-carry adder, the carry-out of one stage is connected to the carry input of the next stage, as shown in Fig. 5.14a.

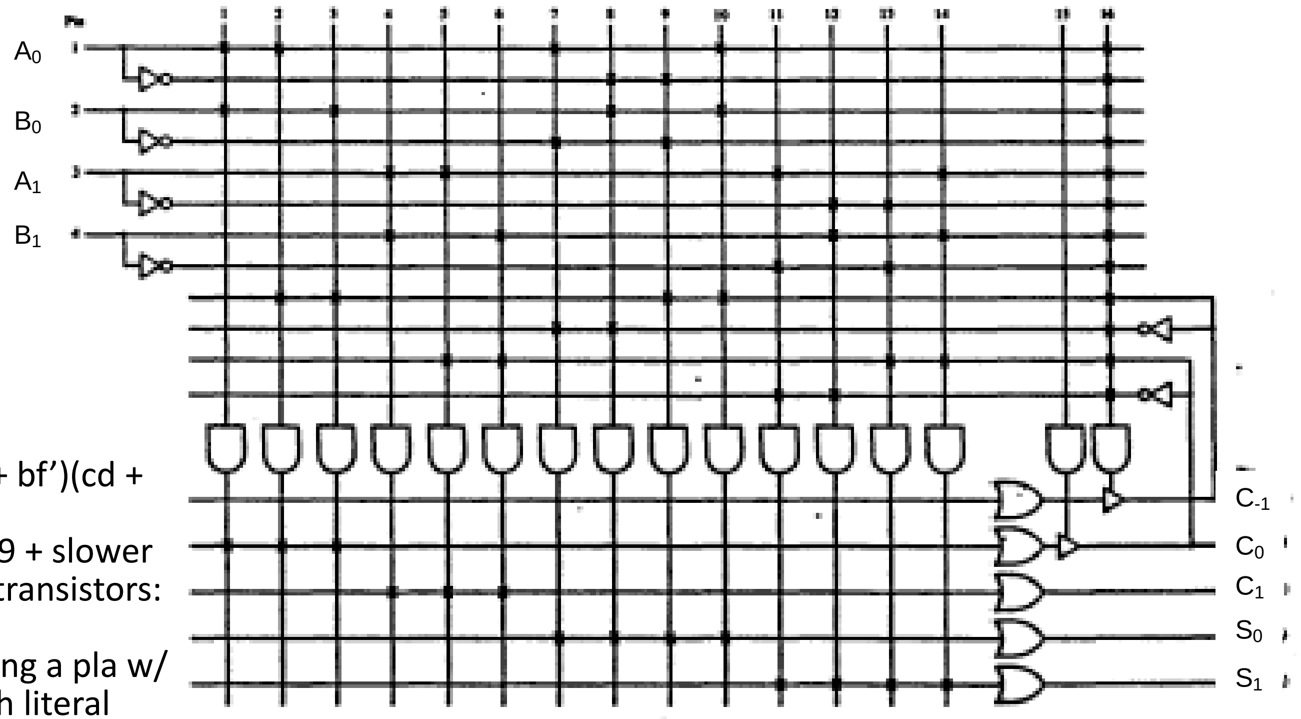


(a)

- PLAs w/ feedback are, in general, useful for implementing "decomposed" circuits, in which the internal o/ps of a subckt is taken as a primary i/p of another sub-circuit.
- Another way of looking at the same thing is that PLAs w/ feedback can implement factored non-SOP expressions.
- E.g. ab'(cd + de) + ce'(ac + bd').

Feedback expressions

- A cost saving example: (ab' + a'f + bf')(cd + d'e + c'e').
- AND line cost of flat expression = 9 + slower (each literal driving more AND line transistors: max = 3).
- AND line cost of factored expr. using a pla w/ feedback: 3 + 3 + 1 = 7 + faster (each literal drives fewer AND line transistors: max = 1).
- However, overall slower as 2 sets of AND-OR delays are incurred due to feedback



(b)

Figure 5.14 Two-bit ripple-carry adder, using I/O and feedback lines. (a) Block diagram with pin numbers. (b) Programmable logic realization.

QM for PLA/PAL Optimization

- Hardware cost optimization for ***programmable PLAs/PALs***:
  - PLA/PAL hardware cost in this case is only the total # of PIs across all functions.
  - The # of literals in a selected PI is of no consequence as the PLA/PAL is designed so that each PI can have up to the max # of literals, and using a PI w/ fewer literals does not reduce PI cost—nMOS transistors populate the entire (i.e., every location of the) AND-plane and OR-plane matrix for programmability, and there is no issue of reducing their numbers. The # of PIs across all functions is = # of AND lines in a PLA, and smaller this #, smaller can be the PLA/PAL size (in terms of # of AND lines).
  - Thus PI cost should be 1 (each chosen PI ➔ 1 AND line)
  - In *multifunction design* (only for PLAs not for PALs):
    - PI cost reduces from 1 to 0 after the PI becomes an EPI for one function, since there will be no more AND lines reqd for this PI if it is chosen for other functions.
    - Also, each OR array can have the max # of AND terms (# of AND lines = total # of chosen PIs) w/o any additional cost (unlike in a gate based design where each additional PI in a function ➔ an additional input for the 2nd level OR gate).
    - A multi-function PI can thus be chosen for each function for which it is a PI (and for which it covers any MTs in the current state of the PIT) once it becomes an EPI for any function. Thus Rule 6 of multifunction QM not needed for PLA cost minimization, and Rule 7 can be used w/o the sweep-up phase. *However, these rules are useful for delay minimization in the OR plane (reducing the # of nMOS gates driven by an AND line = degree of sharing of the PI).*

- Programmable PLA/PAL hardware cost = total # of AND lines = # of PIs in the final expression.
- Note: # of OR lines fixed by # of funcs., so cost optimization not an issue in the OR plane

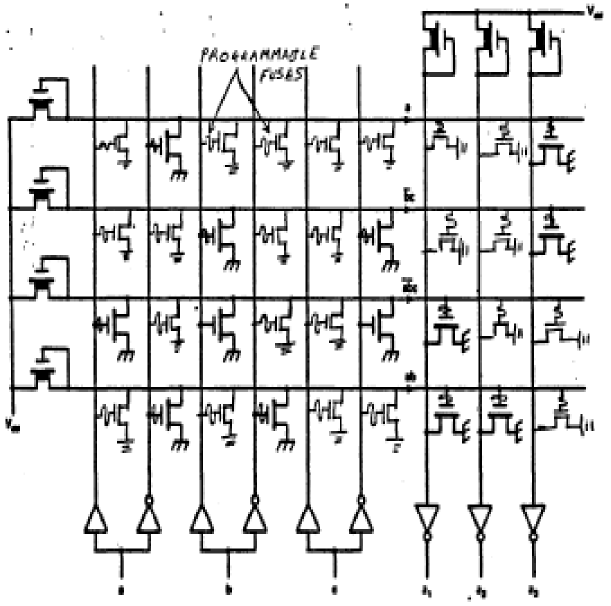
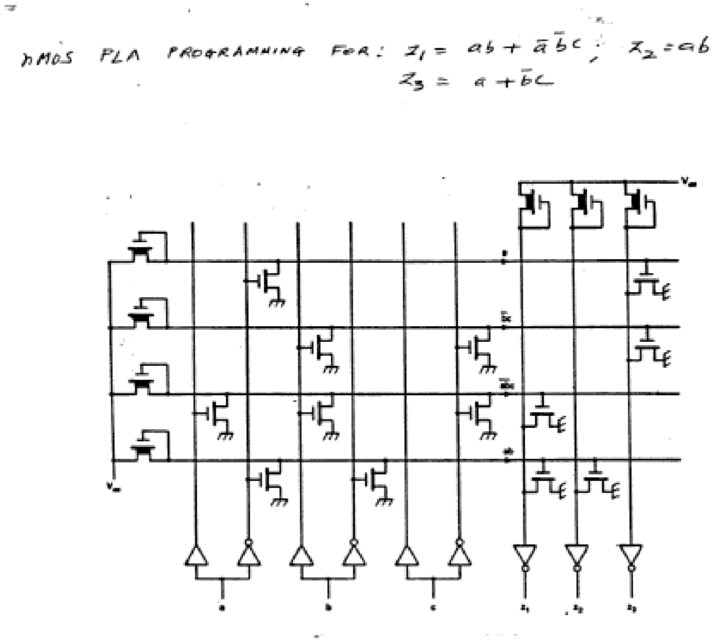


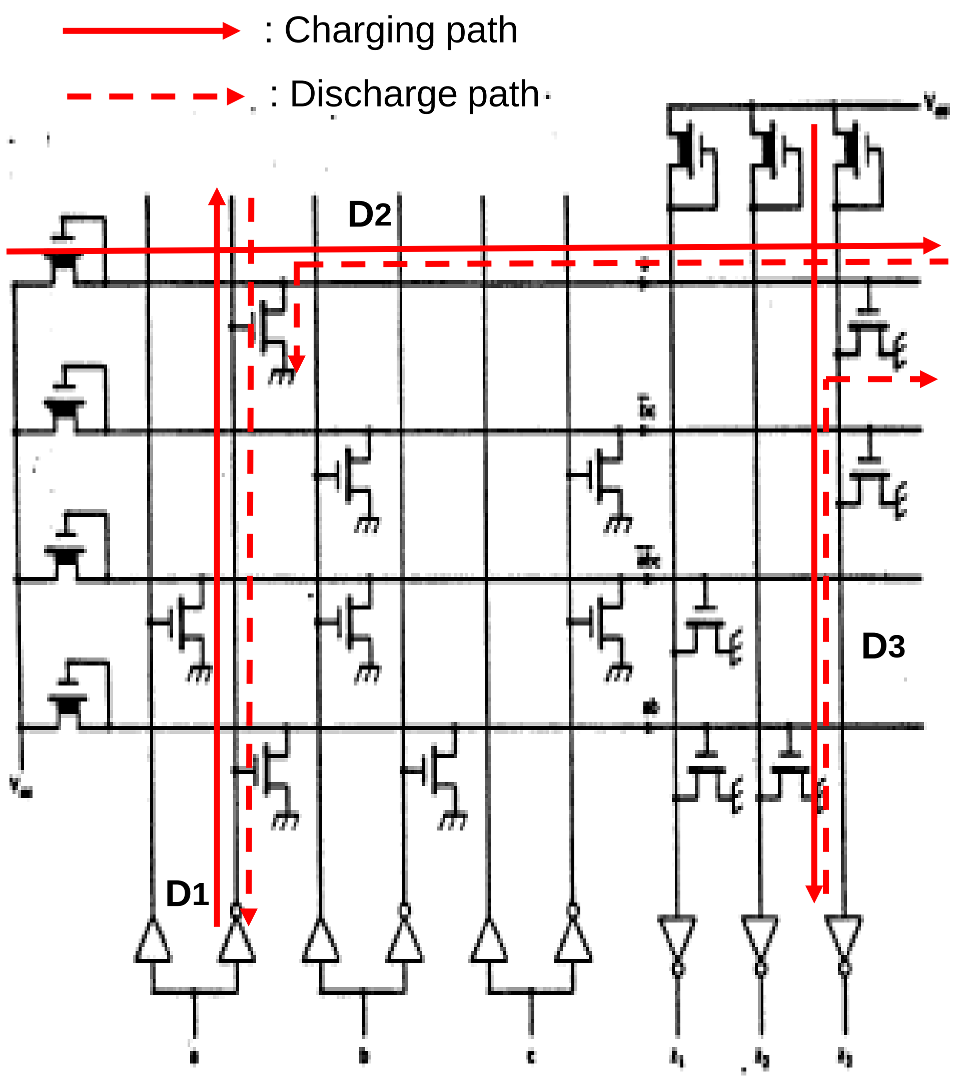
Figure   Circuit schematic for an nMOS PLA.

QM for PLA/PAL Optimization (contd.)

- Hardware cost optimization for ***non-programmable PLAs***:
  - A *non-programmable PLA* is one in which the AND and OR array layouts are similar to that of their programmable cousins, but nMOS transistors w/ their gate connections hardwired to the needed literal line (in the AND array) or the AND line in the OR array
  - PLA hardware cost in this case *could be: total literal cost (= total # of transistors in the AND array) + total # of PIs across all functions (= # of AND lines → corresponding transistors, one per AND line, in the OR array) + $\Sigma_{\text{each PI PIj}}$ [degree of sharing of PIj] (= total trans's. in OR plane)*
  - Further, # of AND lines is more important to reduce than # of transistors in the AND plane, as its affects the area of the PLA, while the # of transistors have a much smaller impact on this). Thus we can assign a larger weight $\beta > 1$ to # of PIs in the above expression to get the *final cost* of a non-programmable PLA to be: ***total literal cost + β*(total # of PIs across all functions)+ $\Sigma_{\text{each PI PIj}}$ [degree of sharing of PIj]***
  - Minimization of the above cost formulation in QM/QM+ is achieved by setting PI cost = # of literals in it + β (AND line cost) + 1. Thus in *multifunction design*:
    - Multi-function PI cost reduces from above to 1 (for "extra" OR plane transistor cost if chosen later for other functions) after it becomes an EPI for one function, since the AND plane transistor cost + AND line cost for this PI has been incurred and will not be incurred further if chosen later for other functions. reqd for this PI if it is chosen for other functions
    - Thus either Rule 6 or Rule 7 followed by the sweep-up phase is needed in cost minimization of a (multi-function) non-programmable PLA.


nMOS PLA PROGRAMMING FOR: $z_1 = ab + \bar{a}\bar{b}c$; $z_2 = ab$
$z_3 = a + \bar{b}c$

PLA/PAL Delay



• Assume charge & discharge times on a line are approx. the same.

• Charge/Discharge time on a line *i*

$= R_d * (C_L^i + C_W)$

where R_d is driver/sink (for chrg/disch, resp) transistor resistance, C_L^i is the total transistor gate cap on *i*, and C_W is the wire cap on *i*. We ignore drain/source cap as that is much smaller. We also ignore wire res. here.

• From the PLA circuit opt. point of view, R_d and C_W are constants. Thus we can minimize delay in this design phase by minimizing the max # of transistor (gate) connections across all lines (which minimizes ***$\max_i\{C_L^i\}$*** on each relevant set of lines: literal and AND lines.

• For the literal line (delay D_1) this means min. the max # of PIs a literal belongs to

• For the AND line (del. D_2) this means min. the max # functions they belong to (i.e., the degree of sharing—this is one example of the conflict betw delay and h/w cost.

• For the OR line (del. D_3) there are no trans. gate connections, and drain/src connections are fixed and the same for all lines. Thus there is no delay optimization for this line.

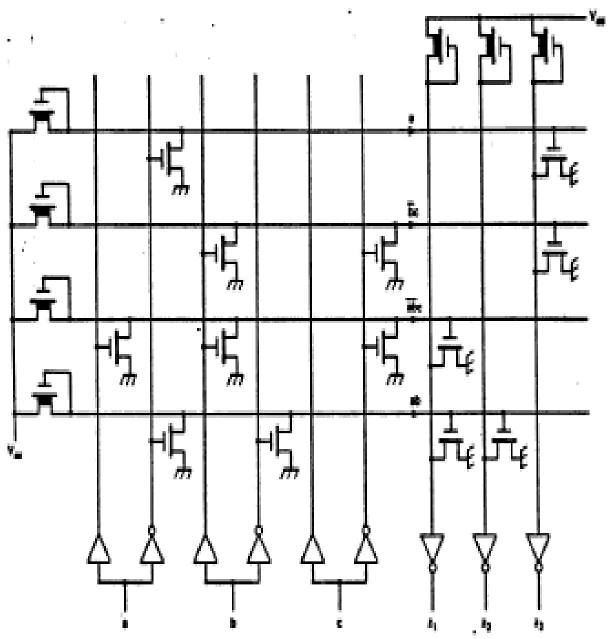
• **PLA Delay = delay of literal line + delay of AND line + delay of OR line =** **D1 +D2 + D3**

• **Critical path delay in PLA = max $_{\text{all interconn 1,2,3 lines}}$ (D1 +D2+ D3) <= max D_1+max D_2+max D_3**

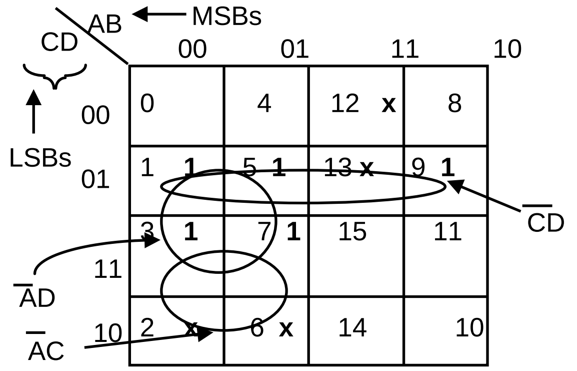
QM for PLA Hardware (H/W) + Delay Opt.

Delay optimization (Initial Cost Formulation):

- PLA delay = max(delay from any input literal to an o/p)
- Each literal signal incurs a delay (D_1) that is proportional to the # of transistors it drives. It is thus important to balance the # of transistors driven by each literal ➔ min(max # of chosen PIs that each literal is in)
- Each PI signal's delay on the AND line (D_2) is also similarly proportional to the # of transistors in the OR array that it drives (i.e., on the # of functions it belongs to). So this delay can be minimized by min(max # of functions each PI is in).). However, PI sharing is useful for cost min. → so we ignore this aspect of delay here. (Can we augment QM to consider extra delay on AND line when PI is shared, and tradeoff delay w/ cost as we do below for literal lines?)
- But either Rule 6 or Rule 7 followed by the sweep-up phase should be used to reduce *unnecessary sharing*, as the latter can increase the PI signal's delay.
- Thus for combined h/w & delay (D_1 only) min., start w/ a PI cost = 1 (for h/w cost; this will min # of AND lines). After a PI *g* is chosen, increase the cost of each PI *h* by w if *g* and *h* share at least 1 common literal that currently is in the largest # of chosen PIs so far. Otherwise, there is no increase in g's cost.
- This is done, since such literals in *g* will drive an additional transistor gate if g is subsequently chosen, thus increasing the D_1 delay. It is should thus be "expensive" to choose a PI such as g.
- *w* is the ``norm. weight'' of the importance of delay compared to hardware cost (AND lines). E.g., w = 0.2 ➔ we can sacrifice delay by up to (1/0.2)*a = 5α units in order to save 1 AND line, where α <= 1 is the estimated fraction of the contribution of D_1 to the total PLA/PAL delay. On the other hand, w = 2 ➔ we can sacrifice a delay increase of up to (1/2)*a = 0.5α units to save 1 AND line or conversely, we can sacrifice up to 1 AND line to not incur a delay increase of 0.5α units.
- The covering rule can be applied taking this cost into consideration (e.g., for two PIs that cover each other, delete the higher-cost one, and by not deleting a lower-cost covered PI—either PI-pair based covering or least cost/MTs based heur. can be used to break such a "pseudo-cyclic" or "deadlocked" table)



Ex: f(A,B,C,D) = Σ m(1,3,5,7,9) + Σ d(2,6,12,13)

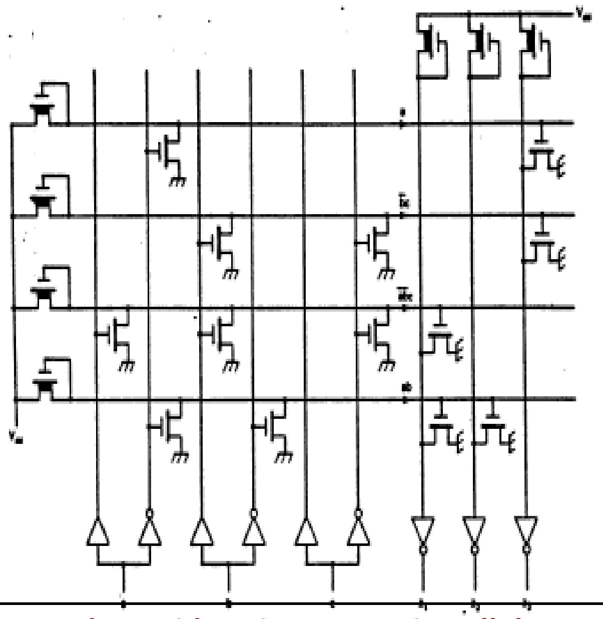


- cost(C'D) = 1 + 0.2 = cost(A'D) = cost(A'C) (w = 0.2)
- C'D is an EPI → choose it → cost(A'D) incr. to 1.2 + 0.2 (due to common literal D driving max trans. [1] so far) = 1.4
- A'C good covers A'D, A'D bad covers A'C → delete A'D → A'C is a p-EPI → choose A'C
- Final soln: **f = C'D + A'C** has a smaller delay in a PLA than the alternate soln. of C'D + A'D

QM for PLA Hardware (H/W) + Delay Opt.

Delay optimization (Initial Cost Formulation):

- PLA delay = max(delay from any input literal to an o/p)
- Each literal signal incurs a delay (D_1) that is proportional to the # of transistors it drives. It is thus important to balance the # of transistors driven by each literal ➔ min(max # of chosen PIs that each literal is in)
- Each PI signal's delay on the AND line (D_2) is also similarly proportional to the # of transistors in the OR array that it drives (i.e., on the # of functions it belongs to). So this delay can be minimized by min(max # of functions each PI is in). However, PI sharing is useful for cost min. → so we ignore this aspect of delay here. (Can we augment QM to consider extra delay on AND line when PI is shared, and tradeoff delay w/ cost as we do below for literal lines?)
- But either Rule 6 or Rule 7 followed by the sweep-up phase should be used to reduce *unnecessary sharing*, as the latter can increase the PI signal's delay.
- Thus for combined h/w & delay (D_1 only) min., start w/ a PI cost = 1 (for h/w cost; this will min # of AND lines). After a PI *g* is chosen, increase the cost of each PI *h* by w if *g* and *h* share at least 1 common literal that currently is in the largest # of chosen PIs so far. Otherwise, there is no increase in g's cost.
- This is done, since such literals in *g* will drive an additional transistor gate if g is subsequently chosen, thus increasing the D_1 delay. It is should thus be "expensive" to choose a PI such as g.
- *w* is the ``norm. weight'' of the importance of delay compared to hardware cost (AND lines). E.g., w = 0.2 ➔ we can sacrifice delay by up to (1/0.2)*a = 5α units in order to save 1 AND line, where α <= 1 is the estimated fraction of the contribution of D_1 to the total PLA/PAL delay. On the other hand, w = 2 ➔ we can sacrifice a delay increase of up to (1/2)*a = 0.5α units to save 1 AND line or conversely, we can sacrifice up to 1 AND line to not incur a delay increase of 0.5α units.
- The covering rule can be applied taking this cost into consideration (e.g., for two PIs that cover each other, delete the higher-cost one, and by not deleting a lower-cost covered PI—either PI-pair based covering or least cost/MTs based heur. can be used to break such a "pseudo-cyclic" table)



Advanced Considerations—Not in Syllabus:

- An alternate cost is to have the cost grow slowly as opposed to being binary w/ a sharp step (go from 0 to 1 as soon as a literal goes from being in max-1 to max # of PIs so far).
- So when PI g is selected, for every PI h that g has common literal(s) with, determine among these literals, the max. # k of chosen PIs they occur in
- E.g., incr. cost by $w*(e/(e-1))*(1-e^{-(k/kmax)})$ OR $w(e^{-(1-[k/kmax])})$ OR $w*k/kmax$, where kmax is the globally max. # of chosen PIs that a literal (*a critical literal*) is currently present in.
- This allows us to "anticipate" and consider costs of literals that can get critical in the near-future in the QM opt. process.
- But this does not do "look-ahead" (check what PIs are remaining and for each literal estimate how many it belongs to will be chosen. **What is a good look-ahead estimate?**