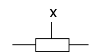
Combinational Logic

- Switches -> Boolean algebra
- Representation of Boolean functions
- Logic circuit elements - logic gates
- Regular logic structures
- Timing behavior of combinational logic
- HDLs and combinational logic
- Incompletely specified functions
- Optimization of combinational logic
- Arithmetic circuits

Switching Algebra

- Based on operation of switches



switch conducts if ON

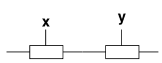
switch does not conduct if OFF

- Values are either 0 or 1 – switch is on or off
  (A1) $X \neq 1 \Rightarrow X = 0$ $X \neq 0 \Rightarrow X = 1$

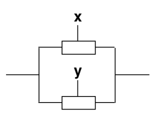
- Complementation – operation which reverses switch state
  (A2) $X = 1 \Rightarrow X' = 0$ $X = 0 \Rightarrow X' = 1$

Switching Algebra (cont'd)

- Switch composition
  - • is series composition



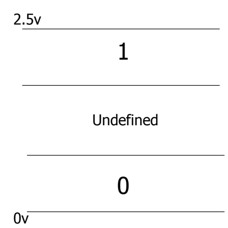
  - + is parallel composition



(A3) $0 \cdot 0 = 0$ $1 + 1 = 1$
(A4) $1 \cdot 1 = 1$ $0 + 0 = 0$
(A5) $1 \cdot 0 = 0 \cdot 1 = 0$ $0 + 1 = 1 + 0 = 1$

Representing Boolean Values

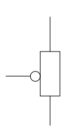
- Use voltage



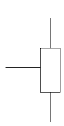
Only Switches We Need

- *How do we make an inverter?*
  - *input 2.5v ⇒ output 0v*
  - *input 0v ⇒ output 2.5v*

Conducts when control is 0v

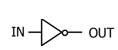


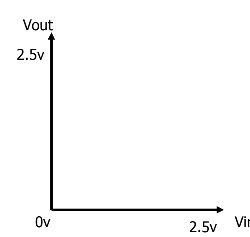
Conducts when control is 5v



Before we Abstract Voltage away Entirely...

- What happens when input isn't 2.5v or 0v?
  - OK for input to be 2.5v - ε
  - OK for input to be 0v + ε



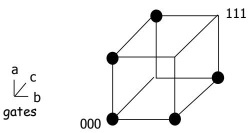


Representing of Boolean Functions

- Boolean expression: a'b + ab' + a'c'
- truth table:

a	b	c	a'b	ab'	a'c'	a'b+ab'+a'c'
0	0	0	0	0	1	1
0	0	1	0	0	0	0
0	1	0	1	0	1	1
0	1	1	1	0	0	1
1	0	0	0	1	0	1
1	0	1	0	1	0	1
1	1	0	0	0	0	0
1	1	1	0	0	0	0

- n-dimensional cube:



- circuit with logic gates

Canonical forms: Sum Of Products

- Truth table is the unique signature of a Boolean function
- Many alternative expressions may have the same truth table
- Canonical form
  - standard form for a Boolean expression
  - Sum-of-products form – a.k.a. disjunctive normal form or minterm expansion

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

011 100 101 110 111

F = A'BC + AB'C' + AB'C + ABC' + ABC

F' = A'B'C' + A'B'C + A'BC'

Canonical Forms: SOP (cont'd)

- Product term (or minterm)
  - ANDed product of literals in which each variable appears exactly once, in true or complemented form (but not both)

A	B	C	minterms	
0	0	0	A'B'C'	m0
0	0	1	A'B'C	m1
0	1	0	A'BC'	m2
0	1	1	A'BC	m3
1	0	0	AB'C'	m4
1	0	1	AB'C	m5
1	1	0	ABC'	m6
1	1	1	ABC	m7

short-hand notation for minterms of 3 variables

F in canonical form:

F(A, B, C) = Σm(3,4,5,6,7)
= m3 + m4 + m5 + m6 + m7
= A'BC + AB'C' + AB'C + ABC' + ABC

canonical form ≠ minimal form

F(A, B, C) = AB'(C + C') + A'BC + AB(C' + C)
= AB' + A'BC + AB
= A(B' + B) + A'BC
= A + A'BC
= A + BC

Product-of-sums canonical form

- Also known as conjunctive normal form
- Also known as maxterm expansion

F = *000* *010* *100*

F = (A + B + C) (A + B' + C) (A' + B + C)

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

F' = (A + B + C') (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' + C')

Canonical Forms: POS

- Product-of-sums form – a.k.a. conjunctive normal form or maxterm expansion
- Sum term (or maxterm)
  - ORed sum of literals in which each variable appears exactly once, in true or complemented form (but not both)

A	B	C	maxterms	
0	0	0	A+B+C	M0
0	0	1	A+B+C'	M1
0	1	0	A+B'+C	M2
0	1	1	A+B'+C'	M3
1	0	0	A'+B+C	M4
1	0	1	A'+B+C'	M5
1	1	0	A'+B'+C	M6
1	1	1	A'+B'+C'	M7

short-hand notation for maxterms of 3 variables

F in canonical form:

F(A, B, C) = ΠM(0,1,2)
= M0 • M1 • M2
= (A + B + C) (A + B + C') (A + B' + C)

canonical form ≠ minimal form

F(A, B, C) = (A + B + (C • C')) (A + (B • B') + C)
= (A + B) (A + C)
= A + BC

Incompletely specified functions

- Example: binary coded decimal increment by 1
  - BCD digits encode the decimal digits 0 – 9 in the bit patterns 0000 – 1001

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

off-set of W

on-set of W

don't care (DC) set of W

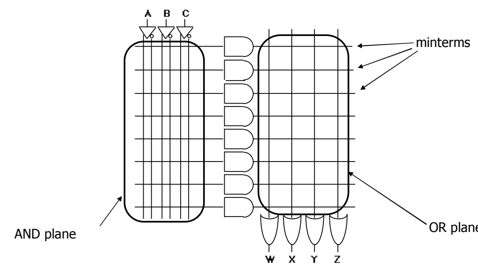
these inputs patterns should never be encountered in practice – we **"don't care"** about associated output values, and this can be exploited in minimization

Notation for incompletely specified functions

- Don't cares and canonical forms
  - so far, only represented on-set
  - must also represent don't-care-set
  - need two of the three sets (on-set, off-set, dc-set)
- Canonical representations of the BCD increment by 1 function:
  - Z = m0 + m2 + m4 + m6 + m8 + d10 + d11 + d12 + d13 + d14 + d15
  - $Z = \Sigma$ [m(0,2,4,6,8) + d(10,11,12,13,14,15)]
  - Z = M1 • M3 • M5 • M7 • M9 • D10 • D11 • D12 • D13 • D14 • D15
  - $Z = \Pi$ [M(1,3,5,7,9) • D(10,11,12,13,14,15)]

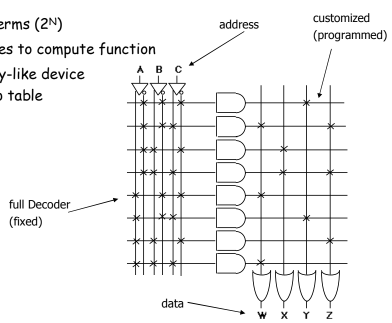
Regular Two-Level Logic

- Basis is canonical form
- Note notation for high-fanin gates



Complete Minterm Array

- Provide all minterms (2^N)
- Connect OR gates to compute function
- This is a memory-like device
  - a.k.a. look up table



Memory Example

- Example: combinational logic implementation (two-level canonical form)

F0 = A' B' C + A B' C' + A B' C

F1 = A' B' C + A' B C' + A B C

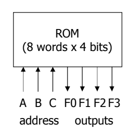
F2 = A' B' C' + A' B' C + A B' C'

F3 = A' B C + A B' C' + A B C'

Truth Table

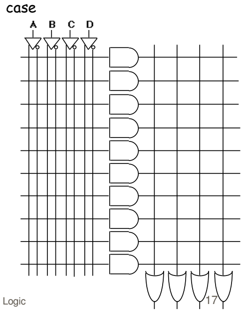
Address			Data			
A	B	C	F0	F1	F2	F3
0	0	0	0	0	1	0
0	0	1	1	1	1	0
0	1	0	0	1	0	0
0	1	1	0	0	0	1
1	0	0	1	0	1	1
1	0	1	1	0	0	0
1	1	0	0	0	0	1
1	1	1	0	1	0	0

Block Diagram



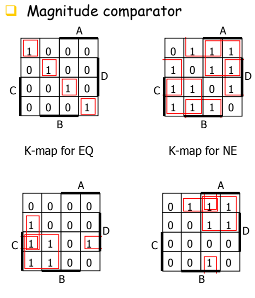
Programmable Logic Array

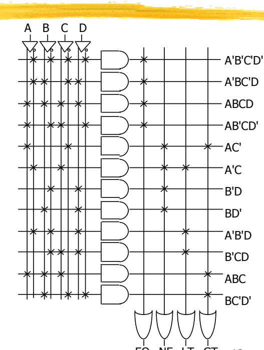
- Observation - don't need all the minterms
- Result - supply enough for the "average" case
  - "a ROM that cheats" - T. Kehl
- Program the ANDs for minterms
- Program the ORs for outputs
- More expensive than ROM
- Slower than ROM



PLAs: another design example

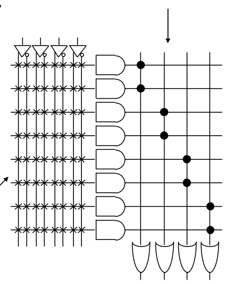
- Magnitude comparator





PALs

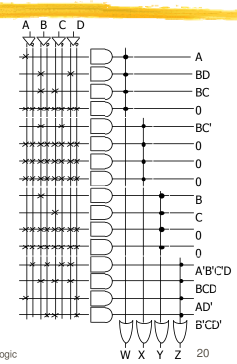
- Programmable array logic (PAL)
  - constrained topology of the OR array
  - innovation by Monolithic Memories
  - faster and smaller OR plane
  - "PLA per pin"



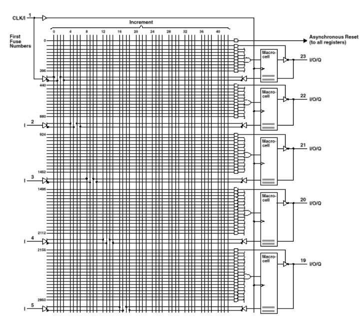
PALs: Design example (cont'd)

- Code converter: programmed PAL

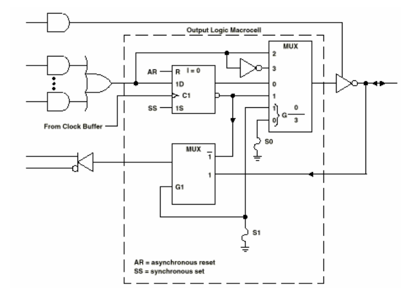
4 product terms
per each OR gate



Typical PLD: 22V10



PLD Output



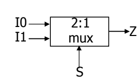
PALs

- Typical PAL/PLD: 22V10
  - 22 pins, 10 outputs (can be used as inputs/feedback)
  - Output pins have a register which can be bypassed
    - Register output feeds back to inputs
  - Different # of product terms/pin
    - Some 8, some 16
- Common problems - too few product terms
  - Especially for arithmetic-like functions
- Solutions
  - Change output polarity
    - the complement of a function may use fewer product terms
  - Implement using multi-level logic (more later)
    - factor into multiple functions (e.g. Shannon expansion)
  - Pre-encode inputs - e.g. (A+B),(A'+B),(A+B'),(A'+B')
  - Re-encode outputs - generate more outputs that can be combined

Multiplexers/selectors

- Multiplexers/selectors: general concept
  - 2^n data inputs, n control inputs (called "selects"), 1 output
  - used to connect one of 2^n inputs to the single output
  - control signal pattern forms binary index of input connected to output
  - e.g. 2-1 mux

$Z = S' I_0 + S I_1$



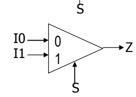
S	Z
0	I_0
1	I_1

functional form

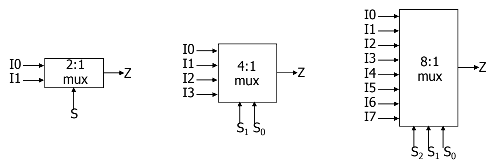
logical form

I_1	I_0	S	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

two alternative forms
for a 2:1 Mux truth table

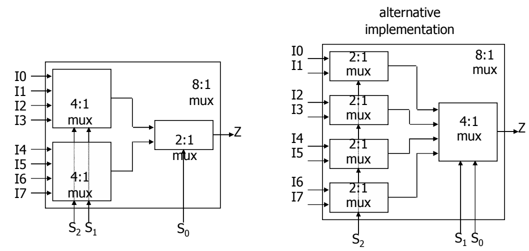


Multiplexers/selectors (cont'd)



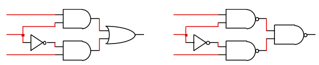
Cascading multiplexers

- Large multiplexers can be implemented by cascading smaller ones using a tree structure

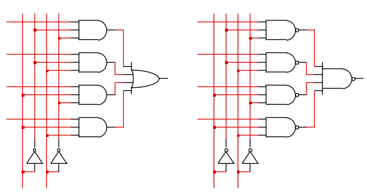


Gate level implementation of muxes

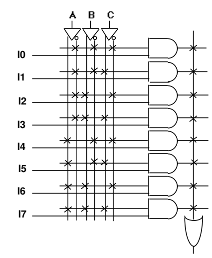
- 2:1 mux



- 4:1 mux

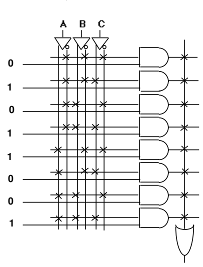


Multiplexer Logic



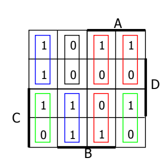
Multiplexer Logic (cont'd)

- What function does this compute?



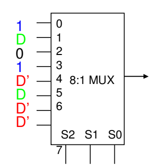
Multiplexer Logic (cont'd)

- Multiplexors as a general-purpose logic block
  - a 2^n:1 multiplexor can implement any function of n variables by connecting the inputs to 0/1 (a lookup table)
  - a 2^{n-1}:1 multiplexor can implement any function of n variables by connecting inputs to one variable or its complement
  - etc.
- Example: F(A,B,C,D) can be implemented by an 8:1 MUX



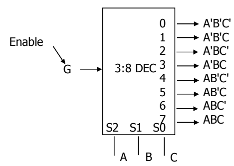
choose A,B,C as control variables

multiplexer implementation



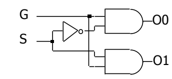
decoders

- General idea:
  - Convert a binary number into a 1-hot number
    - n inputs (address)
    - 2n outputs
    - enable input (optional)
      0 -> all outputs 0
      can be used as data input
        demultiplexer

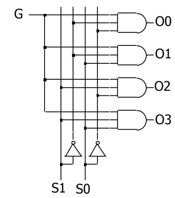


Gate level implementation of demultiplexers

- 1:2 decoder

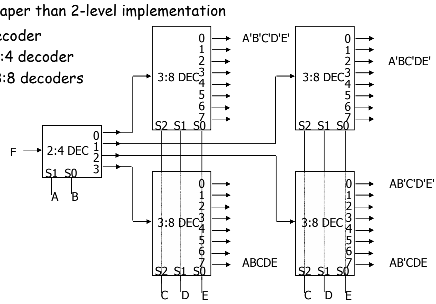


- 2:4 decoder



Cascading decoders

- Use a tree structure
  - cheaper than 2-level implementation
- 5:32 decoder
  - 1x2:4 decoder
  - 4x3:8 decoders

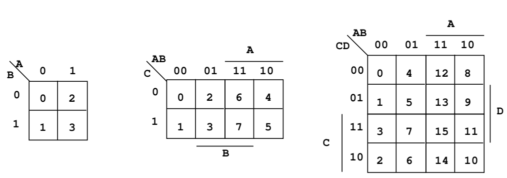


Two-Level Logic Minimization

- Useful for regular two-level implementation methods, e.g. PLA, PAL
- Exact procedure is straightforward
  - Good CAD tools
- OK for “small” functions - ~10 input variables
- In general, for large functions
  - Algorithm is too slow (exponential in number of inputs)
  - Even if you could find the minimal circuit, it has exponential size
- What to do?
  -
  -

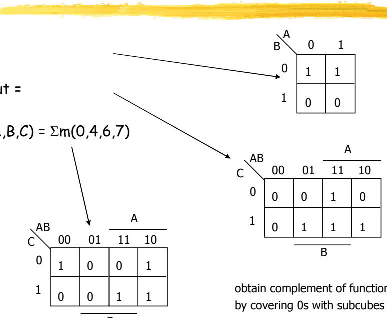
Two-Level Simplification

- Map of cube with wrap-around (hard to draw for more than 4 dimensions)
- Karnaugh map: an alternative to truth-tables to help visualize adjacencies
- Computer-based methods are necessary for practical applications
- Numbering scheme based on Gray-code – e.g., 00, 01, 11, 10
  - only a single bit changes in code for adjacent map cells

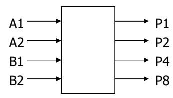


Two-Level Simplification (cont’d)

- F =
- Cout =
- f(A,B,C) = Σm(0,4,6,7)



Design example: 2x2-bit multiplier

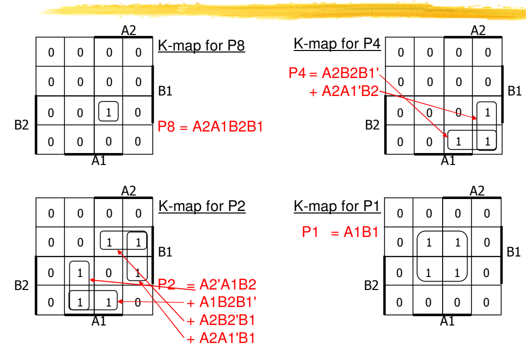


block diagram
and
truth table

A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

4-variable K-map
for each of the 4
output functions

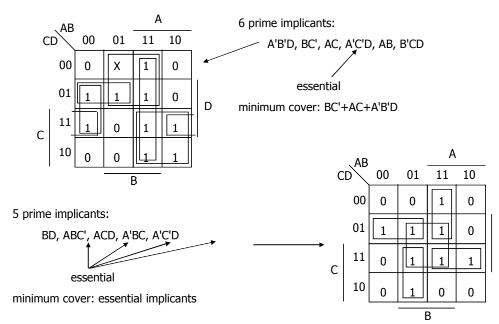
Design example: 2x2-bit multiplier (cont'd)



Definitions for Two-Level Simplification

- Implicant – single element of the ON-set or DC-set or any group of these elements that can be combined to form a subcube
- Prime implicant – implicant that cannot be combined with another to form a larger subcube
- Essential prime implicant – a prime implicant is essential if it is the only one to cover an element of the ON-set (will participate in ALL possible covers of the ON-set)
  - Note: don't cares are used to form prime implicants but cannot make the prime implicant essential
- Objective:
  - grow implicant into prime implicants (minimize literals per term)
  - cover the ON-set with as few prime implicants as possible (minimize number of product terms)
  - essential primes participate in all possible covers

Examples to Illustrate Terms



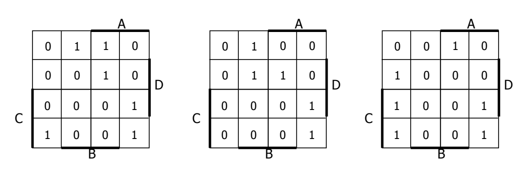
Algorithm for Two-Level Minimization

- Algorithm: minimum sum of products expression from a K-map
  - step 1: choose an element of the ON-set
  - step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - remember to consider top/bottom row, left/right column, and corner adjacencies, this forms prime implicants (number of elements always a power of 2)
  - repeat steps 1 and 2 to find all prime implicants
  - step 3: revisit the 1's in the K-map
    - if covered by single prime implicant, it is essential, and participates in final cover, the 1s it covers do not need to be revisited
  - step 4: if there remain 1s not covered by essential prime implicants, then select the smallest number of prime implicants that cover the remaining 1s

Multiple-Output Functions

- Additional optimization: share terms between functions
- e.g.

$f_1(a,b,c,d) = \Sigma m(2,4,10,11,12,13)$
$f_2(a,b,c,d) = \Sigma m(4,5,10,11,13)$
$f_3(a,b,c,d) = \Sigma m(1,2,3,10,11,12)$



Multiple-Output Function Minimization

- Intersect functions to find all shared terms

f_1 (A over columns 3–4, B over columns 2–3, D on rows 2–3, C on rows 3–4)

0	1	1	0
0	0	1	0
0	0	0	1
1	0	0	1

f_2

0	1	0	0
0	1	1	0
0	0	0	1
0	0	0	1

f_3

0	0	1	0
1	0	0	0
1	0	0	1
1	0	0	1

f_{12}

0	1	0	0
0	0	1	0
0	0	0	1
0	0	0	1

f_{13}

0	0	1	0
0	0	0	0
0	0	0	1
1	0	0	1

f_{23}

0	0	0	0
0	0	0	0
0	0	0	1
0	0	0	1

f_{123}

0	0	0	0
0	0	0	0
0	0	0	1
0	0	0	1

Multiple-Output Function Minimization

- Multiple-Output Prime Implicant
  - Implies one or more functions
    - (intersection of function implicants)
  - Not covered by any other implicant of the same function(s)
  - e.g.
- Minimal cover comprises only MOPI's

Multiple-Output Function Minimization

- Multiple-output functions
- Consider prime implicants for all outputs and whether they can be shared
- Consider conjunction of outputs (all combinations)

$f_1(a,b,c,d) = \Sigma m(2,4,10,11,12,13)$
$f_2(a,b,c,d) = \Sigma m(4,5,10,11,13)$
$f_3(a,b,c,d) = \Sigma m(1,2,3,10,11,12)$
← original three functions

$f_{12}(a,b,c,d) = \Sigma m(4,10,11,13)$
$f_{13}(a,b,c,d) = \Sigma m(2,10,11,12)$
$f_{23}(a,b,c,d) = \Sigma m(10,11)$
← terms that can be shared between pairs of function

$f_{123}(a,b,c,d) = \Sigma m(10,11)$
← terms that can be shared between all three

M.O.P.I's

- Enumerate all the MOPIs

$a = \sum m(2,3,10,11)$
$b = \sum m(4,12)$
$c = \sum m(12,13)$
$d = \sum m(4,5)$
$e = \sum m(5,13)$
$f = \sum m(1,3)$
$g = \sum m(2,10)$
$h = \sum m(2,10)$
$i = \sum m(4)$
$j = \sum m(13)$
$k = \sum m(12)$

- Find essential MOPIs
- Determine a minimal cover

Finding a Cover

f,g,h,j are essential:

		2	4	10	11	12	13	4	5	10	11	13	1	2	3	10	11	12
a	3													X	X	X	X	
b	1		X			X												
c	1					X	X											
d	2							X	X									
e	2								X			X						
~~f~~	~~3~~												(X)		X			
~~g~~	~~13~~	(X)		X										X		X		
~~h~~	~~123~~			X	(X)					(X)	(X)					X	X	
i	12		X					X										
~~j~~	~~13~~					X												(X)
k	12						X					X						

$(b+i)(c+k)(d+i)(d+e)(e+k) = 1$ every column must be covered

$cei+bcde+eik+dik+bdk = 1$ multiply out

bcde is clearly out of consideration – it requires one more product term
any one of cei, eik, dik, or bdk will do for minimum number of product terms or for minimum number of literals (they are all equal cost)

$f_1 = c+g+h+i$
$f_2 = e+h+i$
$f_3 = f+g+h+j$

Problems with 2-Level Optimization

- Number of prime implicants grows rapidly with the number of inputs
  - upper bound: 3**n / n, where n is the number of inputs
- Finding a minimum cover is NP-complete, i.e., a computational
  - expensive process not likely to yield to any efficient algorithm
- Solution: use heuristics, trade minimality of answer for speed
- Espresso: don't generate all prime implicants
  - judiciously select a subset of primes that still covers ON-set
  - operate as a human would in finding primes in a K-map