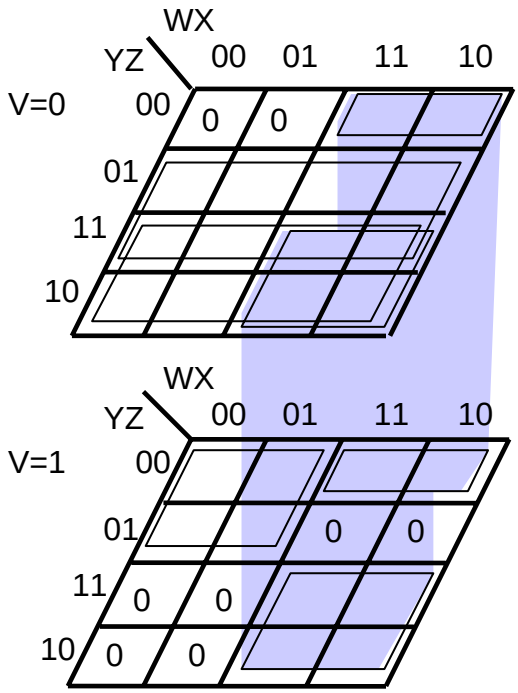
Exercise 3.1

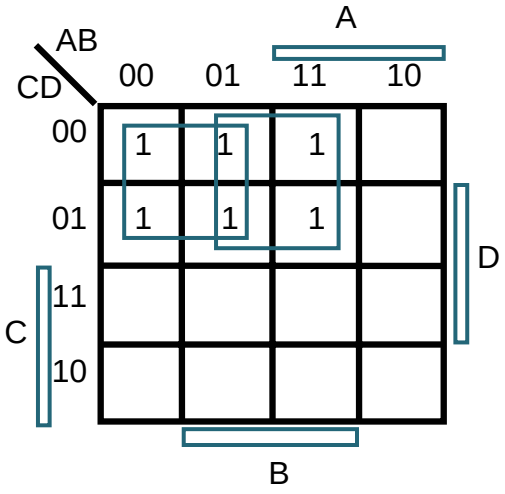
(a) f(V,W,X,Y,Z) = Π M(0,4,18,19,22,23,25,29)



The simplified function has 11 literals:

f = WY + V'W'Y + V'Z + V'Y + WZ'

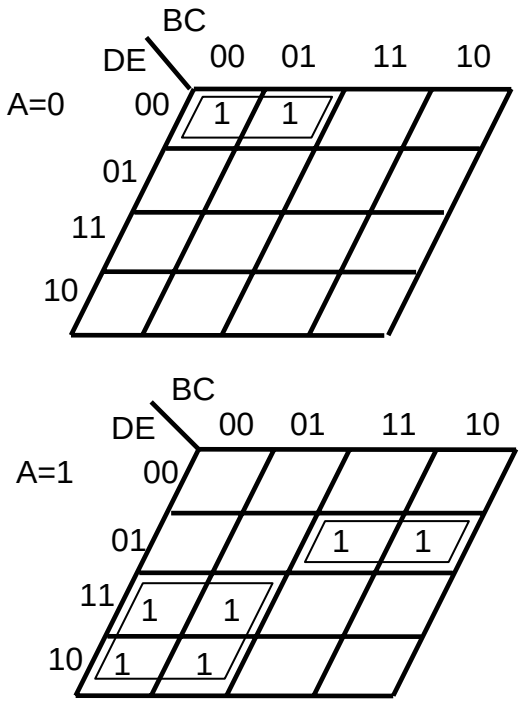
(b) f(A,B,C,D) = Σ m(0,1,4,5,12,13)



The simplified function has 4 literals:

f = A'C' + BC'

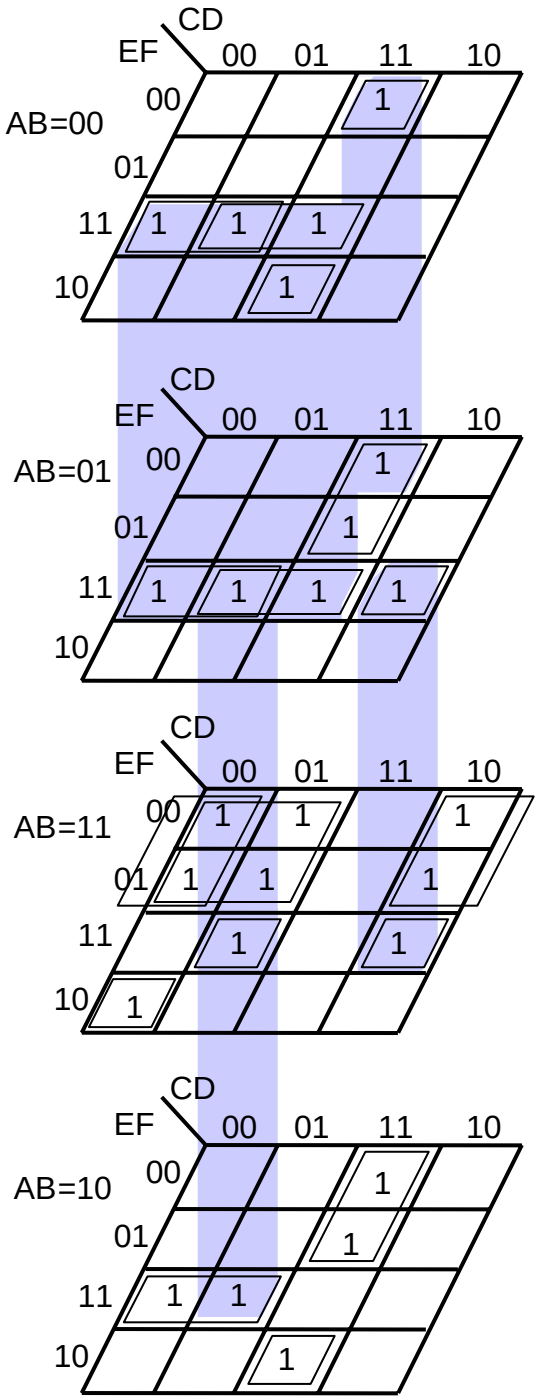
(c) f(A,B,C,D,E) = Σ m(0,4,18,19,22,23,25,29)



The simplified function has 11 literals:

f = AB'D + A'B'D'E' + ABD'E

(d) f(A,B,C,D,E,F) = Σ m(3,7,12,14,15,19,23,27,28,29,31,35,39,44,45,46,48,49,50,52,53, 55,56,57,59)



The simplified function has 46 literals:

f = A'C'EF + C'DEF + B'CDF + A'DEF + A'BCDE' + BCD'EF' + ABC'E' + ABD'E' + ABC'D'EF' + AB'CDE' + AB'C'EF

Exercise 3.2

(a) The following are prime implicants:

WY, V'W'Y, V'Z, V'Y, WZ'

All of the elements are also essential primes because they each contain a '1' that is not covered by any of the other elements.

(b) The following are prime implicants:

A'C', BC'

Both of the elements are also essential primes because they each contain a '1' that is not covered by any of the other elements.

(c) The following are prime implicants:

AB'D, A'B'D'E', ABD'E

All of the elements are also essential primes because they each contain a '1' that is not covered by any of the other elements.

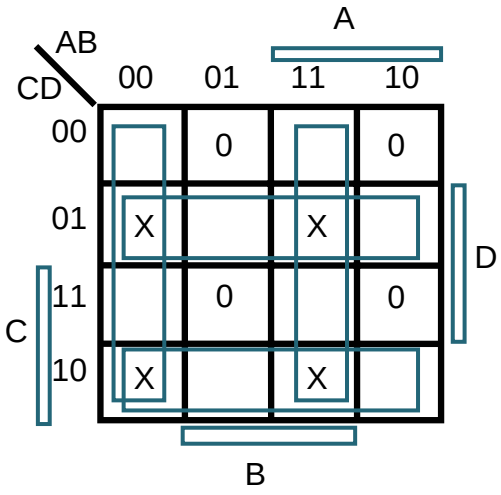
(d) The following are prime implicants:

A'C'EF, C'DEF, B'CDF, A'DEF, A'BCDE', BCD'EF', ABC'E', ABD'E', ABC'D'EF', AB'CDE', AB'C'EF

All of the elements are also essential primes because they each contain a '1' that is not covered by any of the other elements.

Exercise 3.3

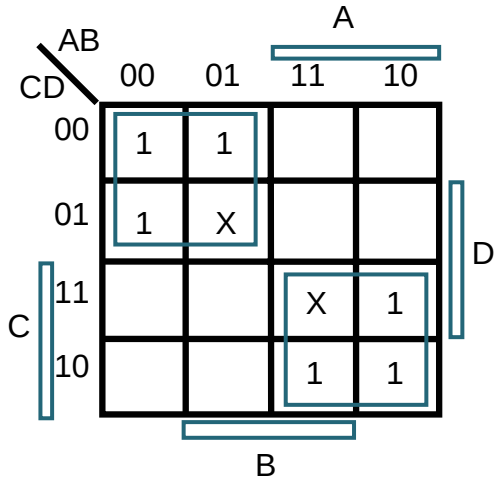
(a) f(W,X,Y,Z) = Π M(4,7,8,11)* Π D(1,2,13,14)



The simplified function is:

f = A'B' + AB + C'D + CD'

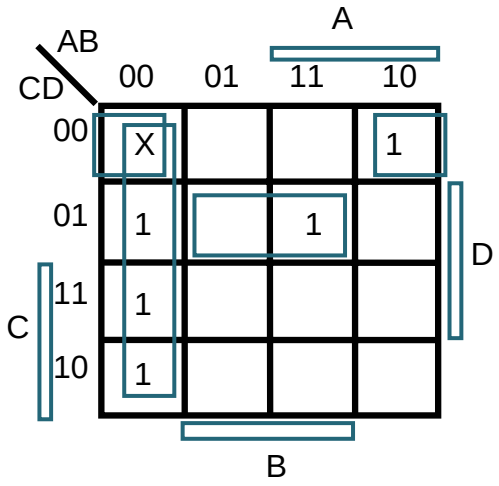
(b) f(A,B,C,D) = Σ m(0,1,4,10,11,14) + Σ d(5,15)



The simplified function is:

f = A'C' + AC

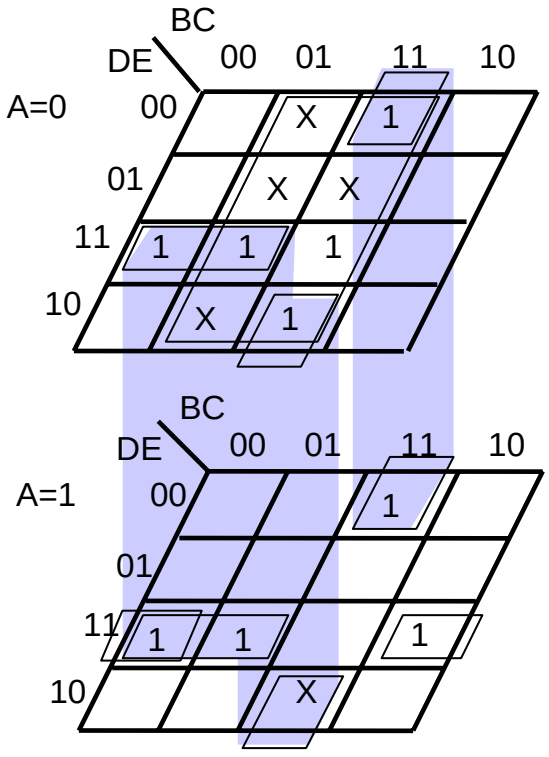
(c) f(A,B,C,D) = Σ m(1,2,3,5,8,13) + Σ d(0)



The simplified function is:

f = A'B' + B'C'D' + BC'D

(d) f(A,B,C,D,E) = Σ m(3,7,12,14,15,19,23,27,28,29,31) + Σ d(4,5,6,13,30)



The simplified function is:

f = A'C + B'DE + BCE' + AC'DE

Exercise 3.4

(a) The following are prime implicants:

A'B', AB, C'D, CD'

All of the elements are also essential primes because they each contain a '1' that is not covered by any of the other elements.

In this function, all of the don't-cares are set to one.

(b) The following are prime implicants:

A'C', AC

All of the elements are also essential primes because they each contain a '1' that is not covered by any of the other elements.

In this function, both of the don't-cares are set to one.

(c) The following are prime implicants:

A'B', B'C'D', BC'D

All of the elements are also essential primes because they each contain a '1' that is not covered by any of the other elements.

In this function, the single don't-care was set to one.

(d) The following are prime implicants:

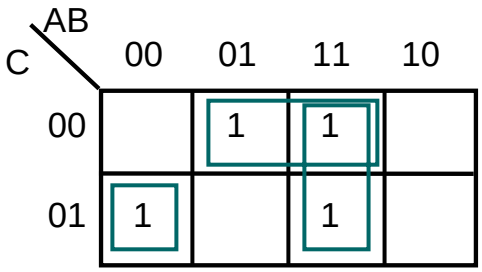
A'C, B'DE, BCE', AC'DE

All of the elements are also essential primes because they each contain a '1' that is not covered by any of the other elements.

In this function, all of the don't-cares are set to one.

Exercise 3.5

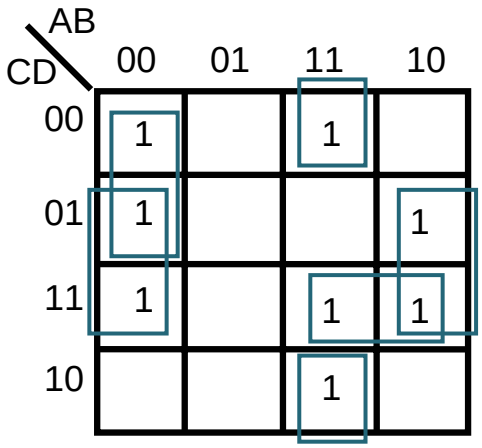
(a) $F(A,B,C) = \Sigma\, m(1,2,6,7)$



The simplified function is:

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

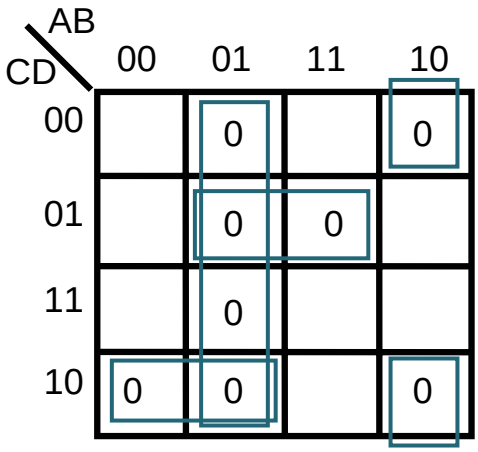
(b) $F(A,B,C,D) = \Sigma\, m(0,1,3,9,11,12,14,15)$



The simplified function is:

$F = B'D + A'B'C' + ACD + ABD'$

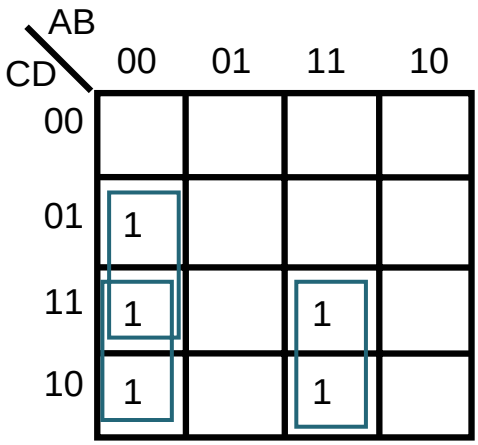
(c) F(A,B,C,D) = Σ m(2,4,5,6,7,8,10,13)



The simplified function is:

F = A'B + BC'D + A'CD' + AB'D

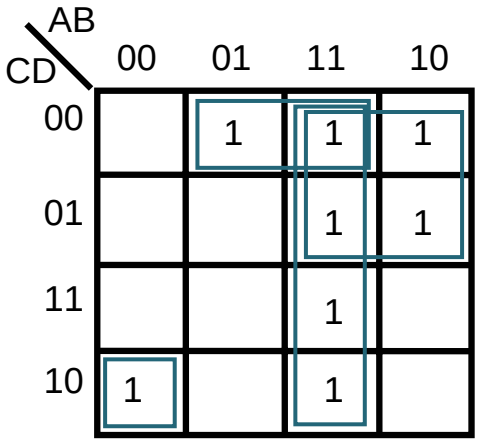
(d) F(A,B,C,D) = (ABC + A'B')(C + D)



The simplified function is:

F = ABC + A'B'C + A'B'D

(e) F(A,B,C,D) = (A’ + B + C)(A + B +C’ + D)(A + B’ + C + D)(A’ + B’)



The simplified function is:

F = AC’ + AB + BC’D’ + A’B’CD’

Exercise 3.6

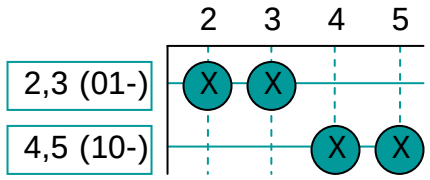
(a) f(X,Y,Z) = Σ m(2,3,4,5)

Column 1	Column 2
010√	01-*
100√	10-*
011√	
101√	

√ implicant
* prime implicant

	2	3	4	5
2,3 (01-)	X	X		
4,5 (10-)			X	X

Initial prime implicant chart



Final configuration

The resulting simplified function is:

f = X'Y + XY'

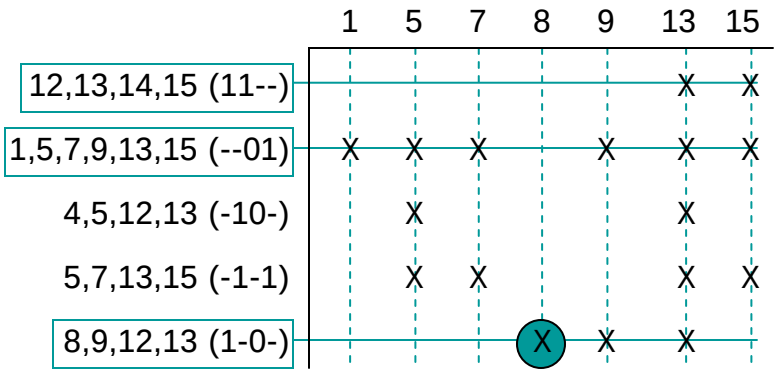
(b) f(A,B,C,D) = Σ m(1,5,7,8,9,13,15) + Σ d(4,12,14)

Column 1	Column 2	Column 3
0001√	0-01√	--01*
0100√	-001√	-10-*
1000√	010-√	1-0-*
	-100√	
0101√	100-√	-1-1*
1001√	1-00√	11--*
1100√		
	01-1√	
0111√	-101√	
1101√	1-01√	
1110√	110-√	
	11-0√	
1111√		
	-111√	
	11-1√	
	111-√	

√ implicant
* prime implicant

	1	5	7	8	9	13	15
12,13,14,15 (11--)						X	X
1,5,7,9,13,15 (--01)	X	X	X		X	X	X
4,5,12,13 (-10-)		X				X	
5,7,13,15 (-1-1)		X	X			X	X
8,9,12,13 (1-0-)				X	X	X	

Initial prime implicant chart



Final configuration

The resulting simplified function is:

f = AB + C'D + AC'

(c) f(A,B,C,D) = Σ m(1,2,3,4,5,6,7,8,9,10,11,12)

Column 1	Column 2	Column 3
0001√	00-1√	0--1*
0010√	0-01√	-0-1*
0100√	-001√	0-1-*
1000√	001-√	-01-*
	0-10√	
0011√	-010√	01--*
0101√	01-0√	10--*
0110√	-100√	
1001√	100-√	
1010√	10-0√	
1100√	1-00√	
0111√	0-11√	

1011√			-011√									
			01-1√									
			011-√									
			10-1√									
			101-√									

√ implicant
* prime implicant

	1	2	3	4	5	6	7	8	9	10	11	12
1,3,5,7 (0--1)	X		X		X		X					
1,3,9,11 (-0-1)	X		X						X			X
2,3,6,7 (0-1-)		X	X			X	X					
2,3,10,11 (-01-)		X	X							X	X	
4,5,6,7 (01--)				X	X	X	X					
8,9,10,11 (10--)								X	X	X	X	
4,12 (-100)				X								X
8,12 (1-00)								X				

Initial prime implicant chart

	1	2	3	4	5	6	7	8	9	10	11	12
1,3,5,7 (0--1)	X		X		X		X					
1,3,9,11 (-0-1)	X		X						X			X
2,3,6,7 (0-1-)		X	X			X	X					
2,3,10,11 (-01-)		X	X							X	X	
4,5,6,7 (01--)				X	X	X	X					
8,9,10,11 (10--)								X	X	X	X	
4,12 (-100)				X								X
8,12 (1-00)								X				

Final configuration

The resulting simplified function is:

f = B'D +B'C + A'B + AC'D'

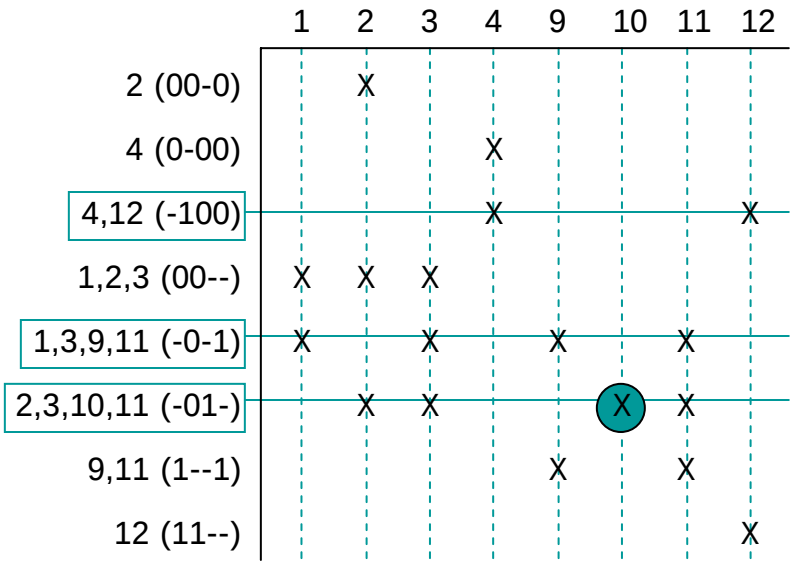
(d) f(A,B,C,D) = Σ m(1,2,3,4,9,10,11,12) + Σ d(0,13,14,15)

Column 1	Column 2	Column 3
0000√	000-√	00--*
	00-0*	
0001√	0-00*	-0-1*
0010√		-01-*
0100√	-001√	
	-010√	1--1*
0011√	001-√	11--*
1001√	-100*	
1010√		
1100√	-0-11√	
	10-1√	
1011√	1-01√	
1101√	110-√	
1110√	11-0√	
1111√	1-11√	
	11-1√	
	111-√	

√ implicant
* prime implicant

	1	2	3	4	9	10	11	12
2 (00-0)		X						
4 (0-00)				X				
4,12 (-100)				X				X
1,2,3 (00--)	X	X	X					
1,3,9,11 (-0-1)	X		X		X		X	
2,3,10,11 (-01-)		X	X			X	X	
9,11 (1--1)					X		X	
12 (11--)								X

Initial prime implicant chart



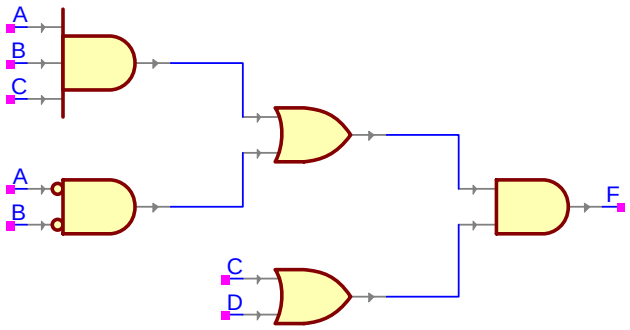
Final configuration

The resulting simplified function is:

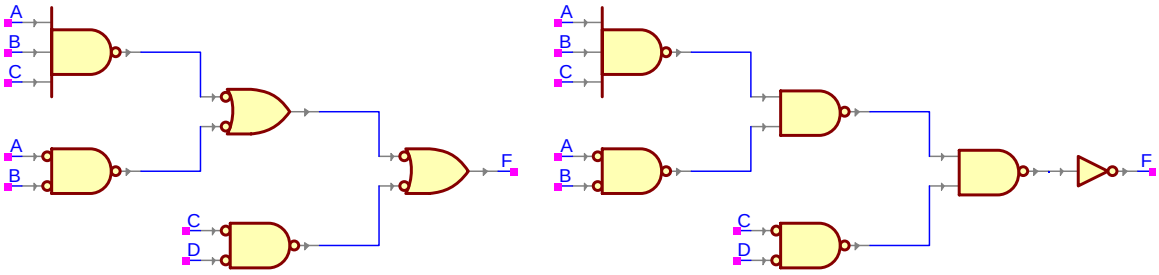
f = BC'D' + B'D + B'C

Exercise 3.8

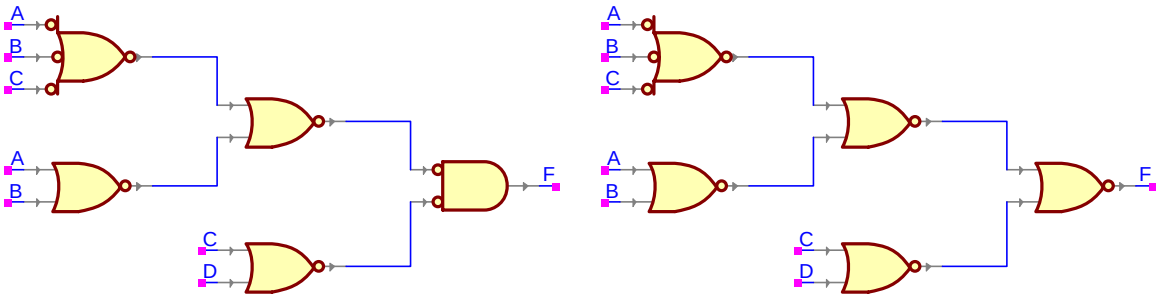
(a) F(A,B,C,D) = (ABC + A'B')(C + D)



Initial Circuit

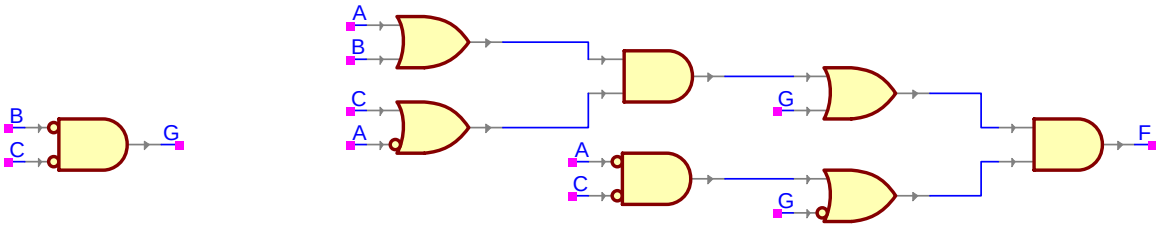


NAND Gate Implementation

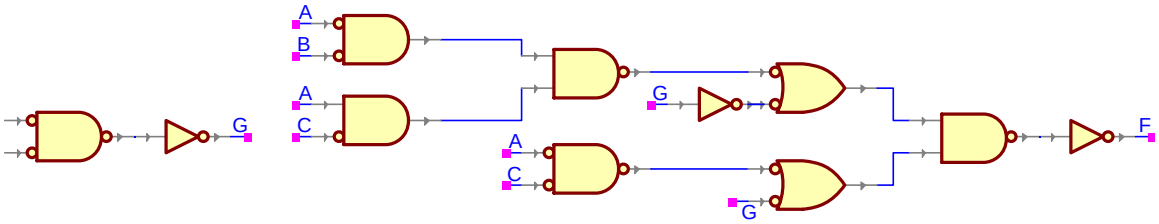


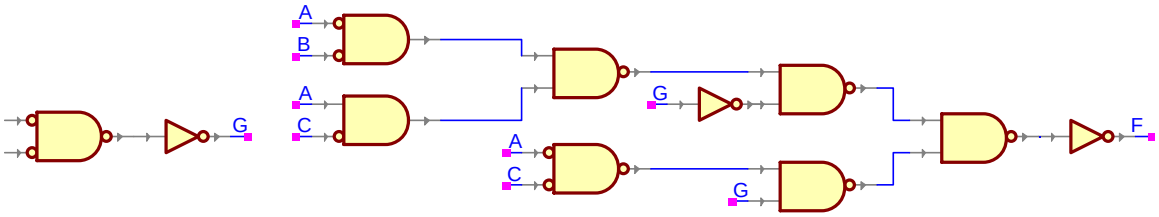
NOR Gate Implementation

(b) G(B,C) = B'C' F(A,B,C,G) = [(A + B)(A' + C) + G](A'C' + G)

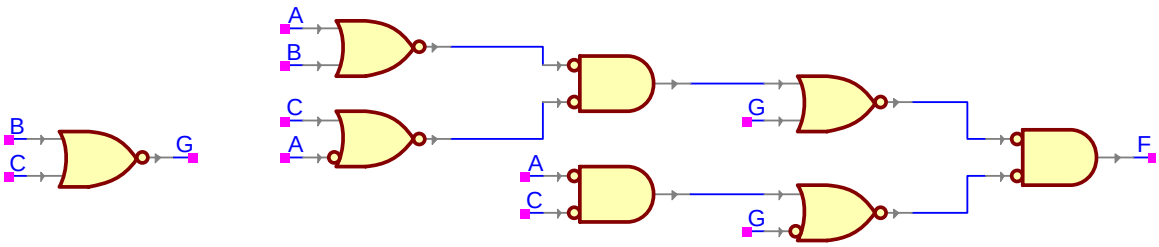


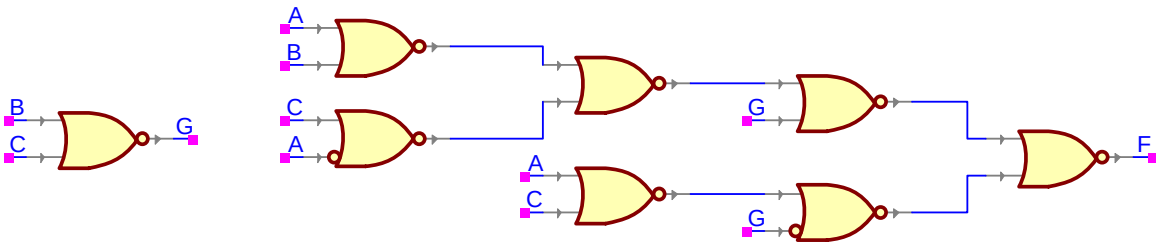
Initial Circuit





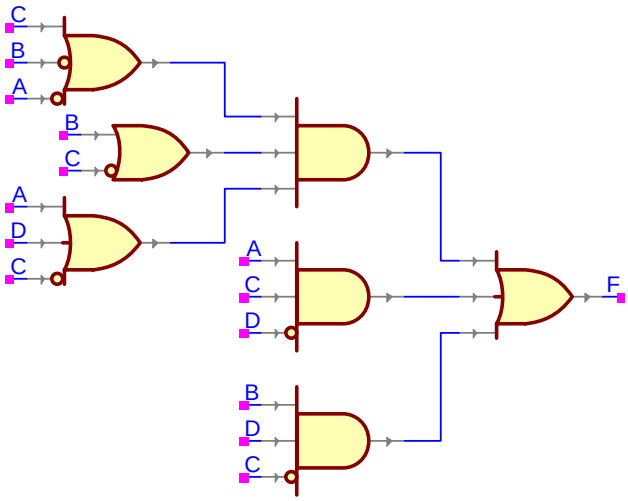
NAND Gate Implementation



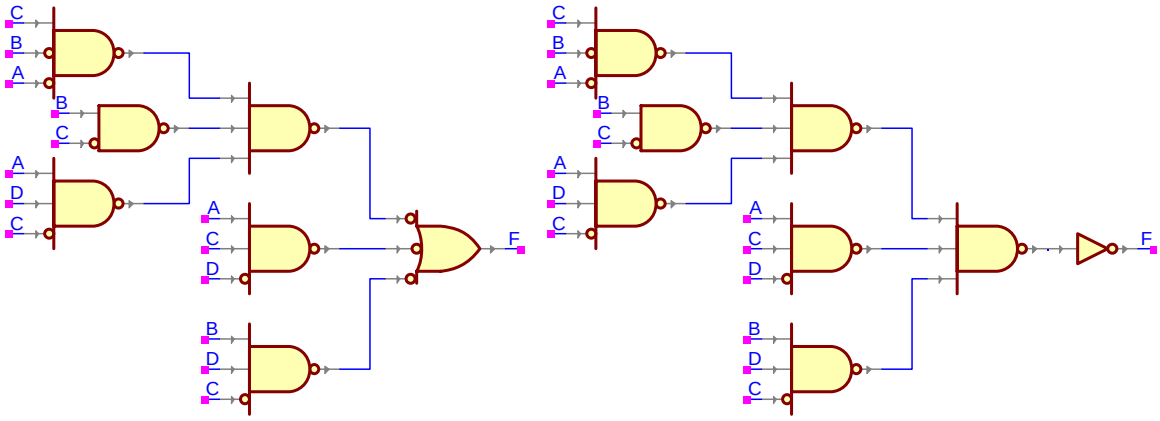


NOR Gate Implementation

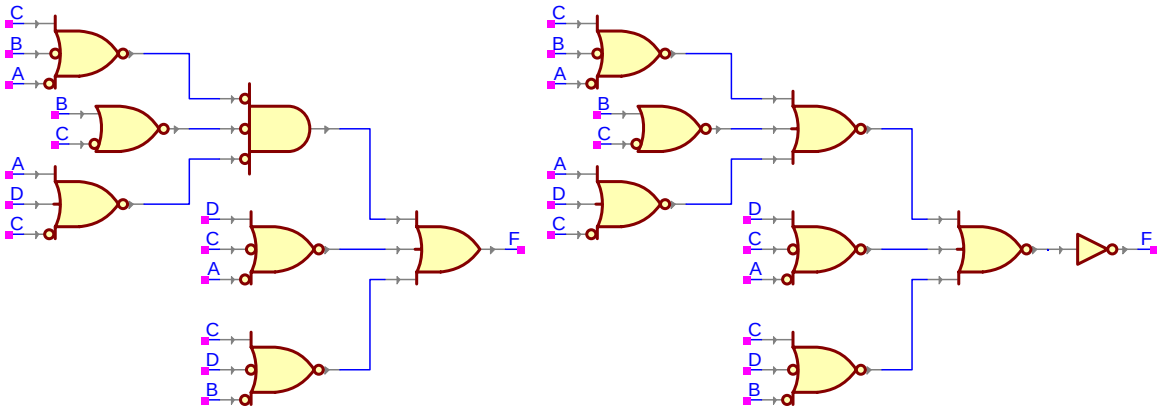
(c) F(A,B,C,D) = [(A' + B' + C)(B + C')(A + B' + D')] + ACD' + BC'D



Initial Circuit

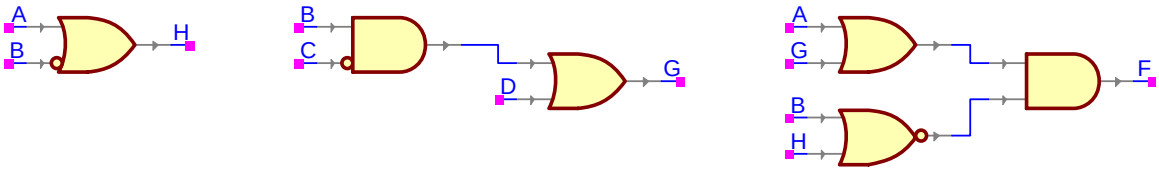


NAND Gate Implementation

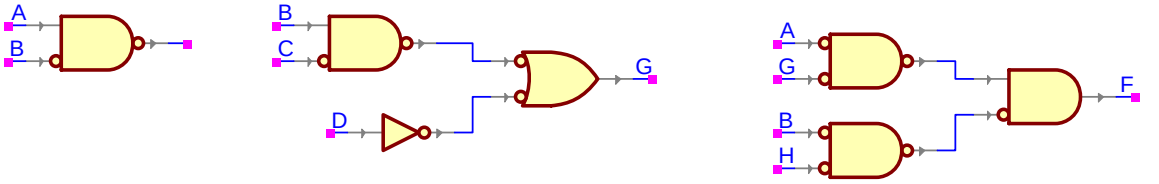


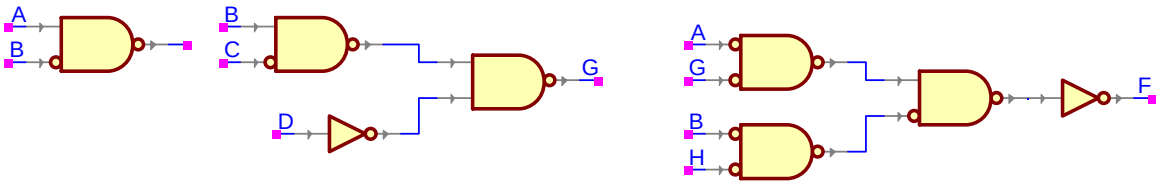
NOR Gate Implementation

(d) H(A,B) = A + B’ G(B,C,D) = BC’ + D F(A,B,G,H) = (A + G)(B + H)’

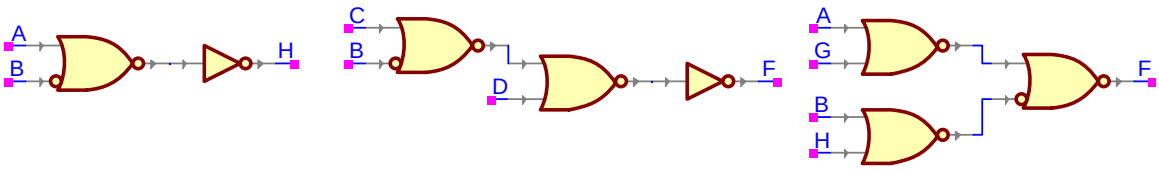


Initial Circuit





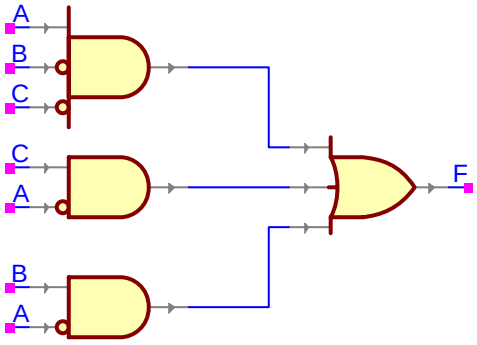
NAND Gate Implementation



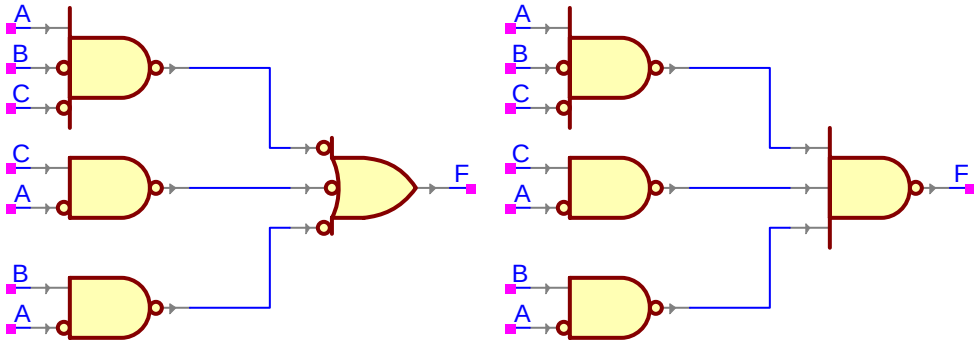
NOR Gate Implementation

Exercise 3.9

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

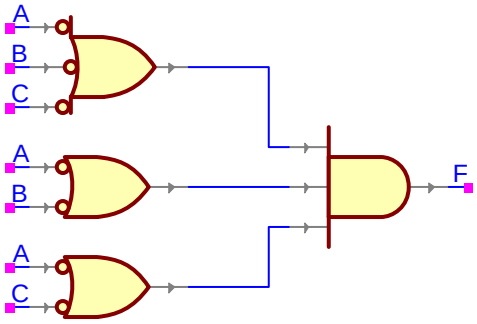


Initial Circuit

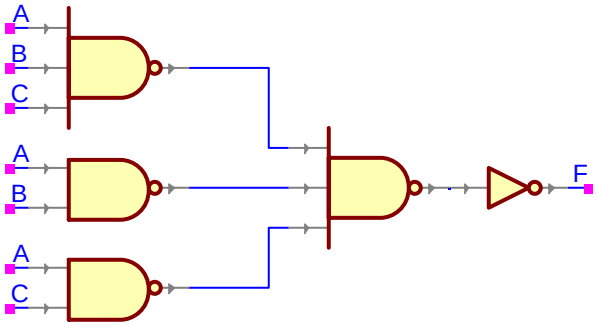


NAND Gate Implementation

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

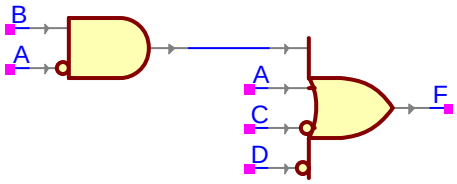


Initial Circuit

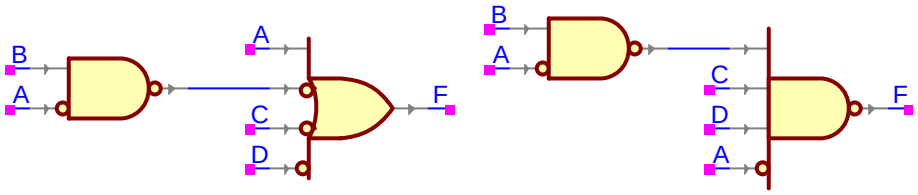


NAND Gate Circuit

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

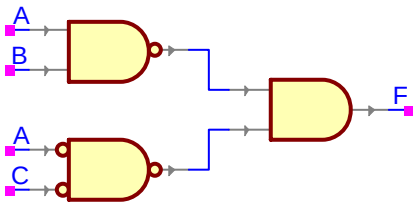


Initial Circuit

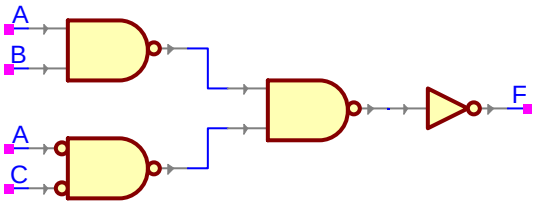


NAND Gate Implementation

(d) F = (AB)'(A'C')'

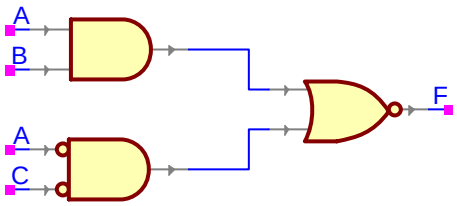


Initial Circuit

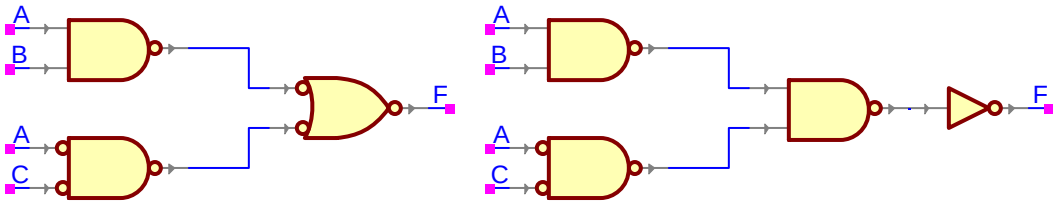


NAND Gate Implementation

(e) F = (AB + A'C')'



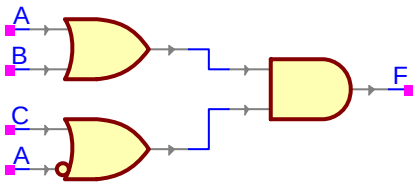
Initial Circuit



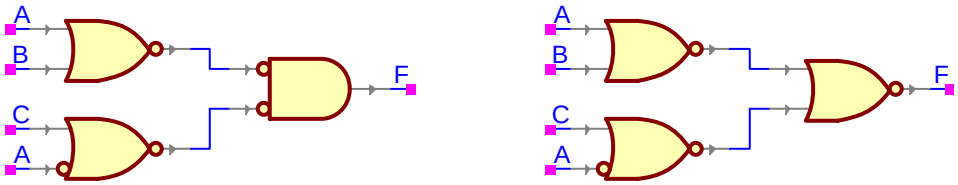
NAND Gate Implementation

Exercise 3.10

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

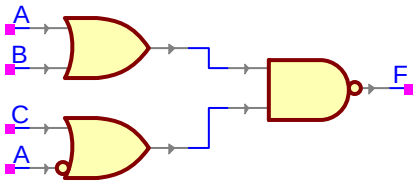


Initial Circuit

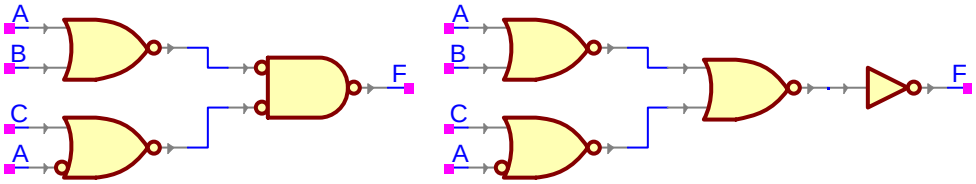


NOR Gate Implementation

(b) F = [(A + B)(A' + C)]'

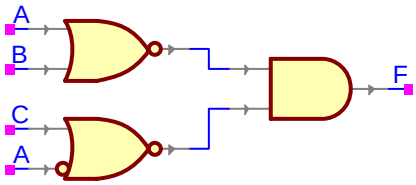


Initial Circuit

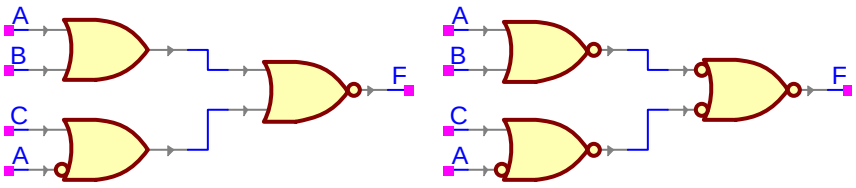


NOR Gate Implementation

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

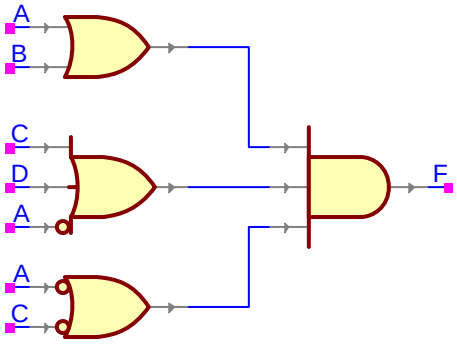


Initial Circuit

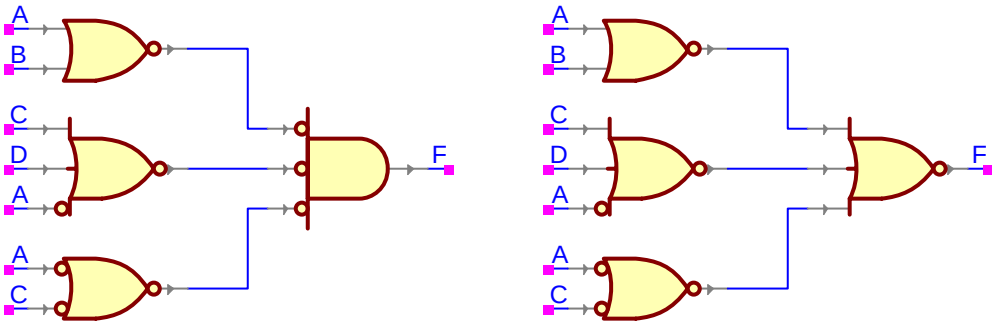


NOR Gate Implementation

(d) F = (A + B)(A’ + C + D)(A’ + C’)

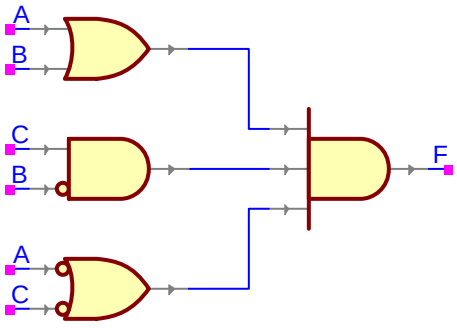


Initial Circuit

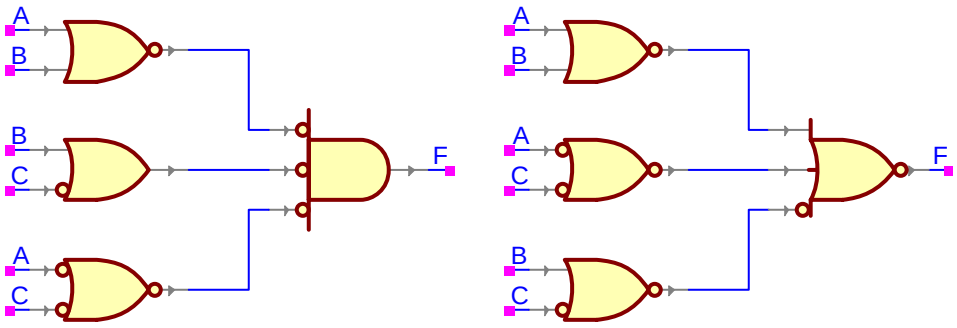


NOR Gate Implementation

(e) F = (A + B)(B’ + C)(A’ + C’)



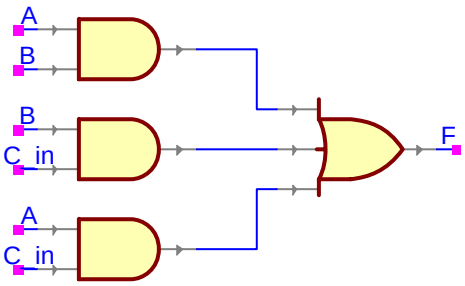
Initial Circuit



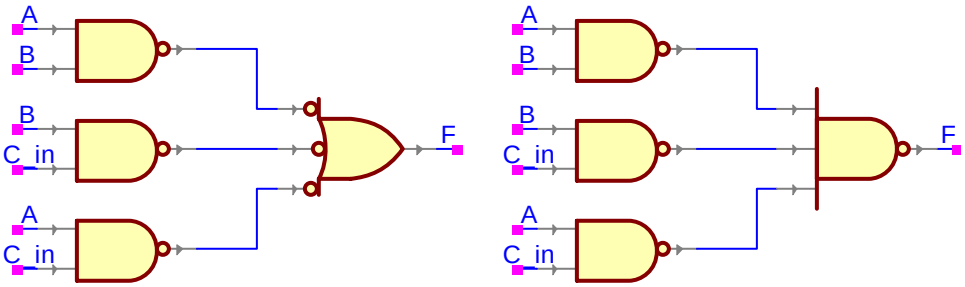
NOR Gate Implementation

Exercise 3.11

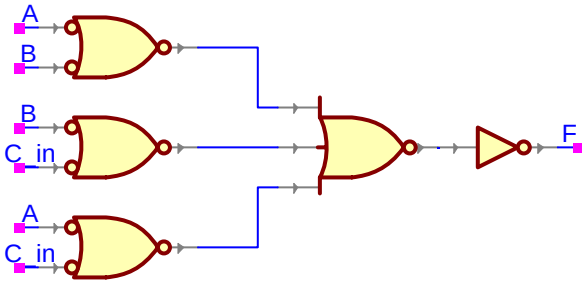
(a) F(A,B,Cin) = AB + BCin + ACin



Initial Circuit

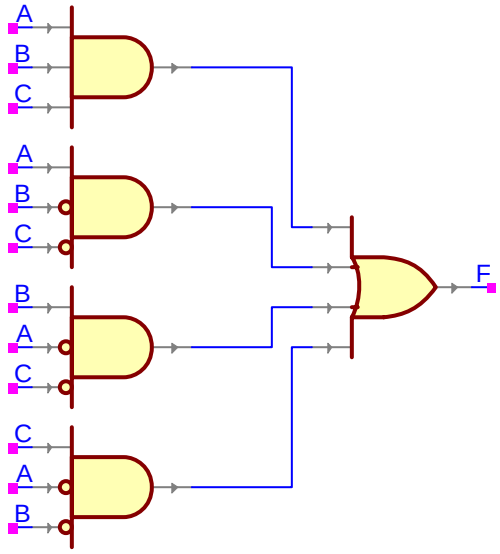


NAND Gate Implementation

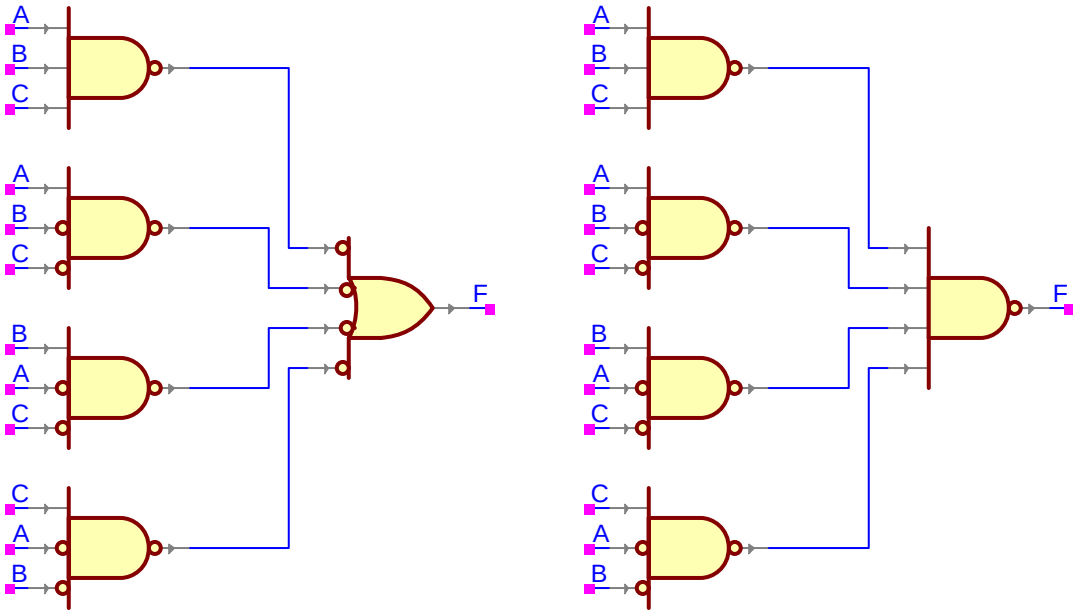


NOR Gate Implementation

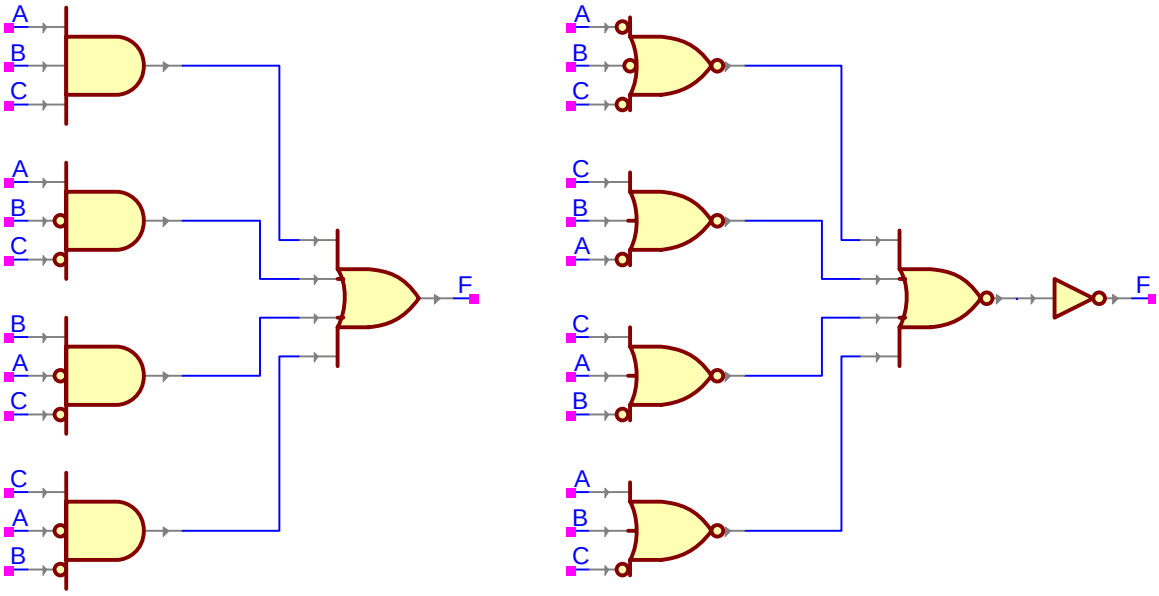
(b) F(A,B,Cin) = A xor B xor Cin



Initial Circuit



NAND Gate Implementation



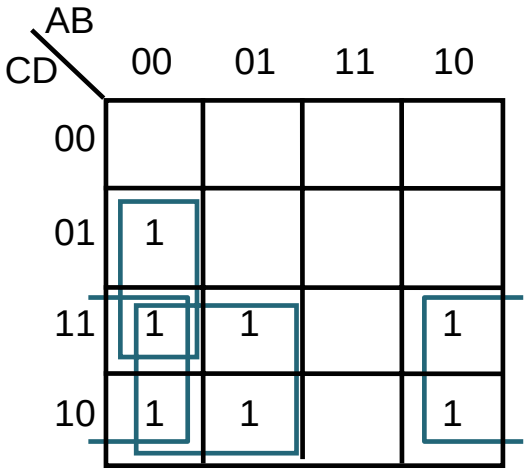
NOR Gate Implementation

(c) The function, F, is true when the 2-bit value AB is strictly less than the 2-bit quantity CD.

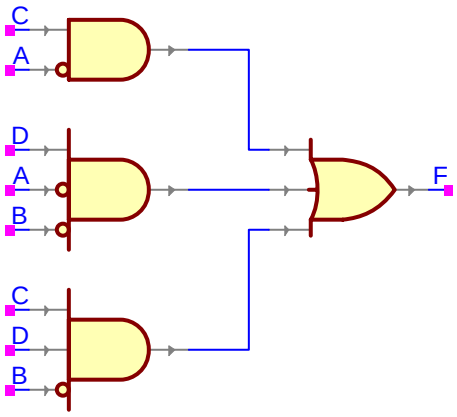
Truth table for F:

A	B	C	D	F(AB < CD)
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

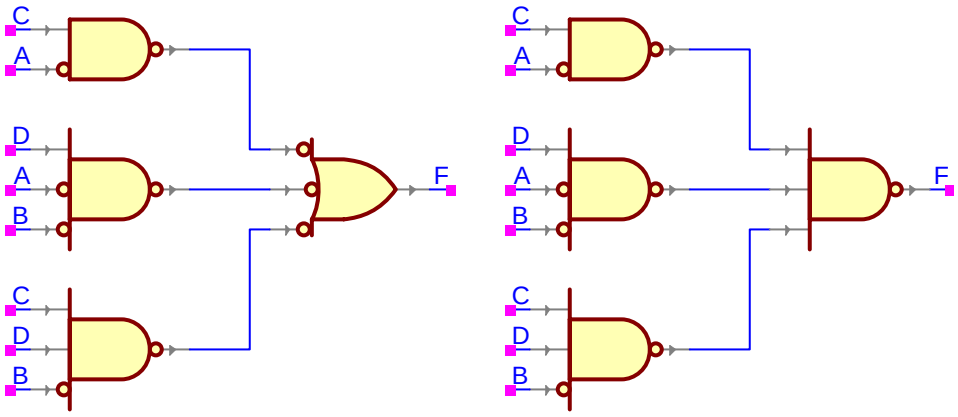
K-map for F:



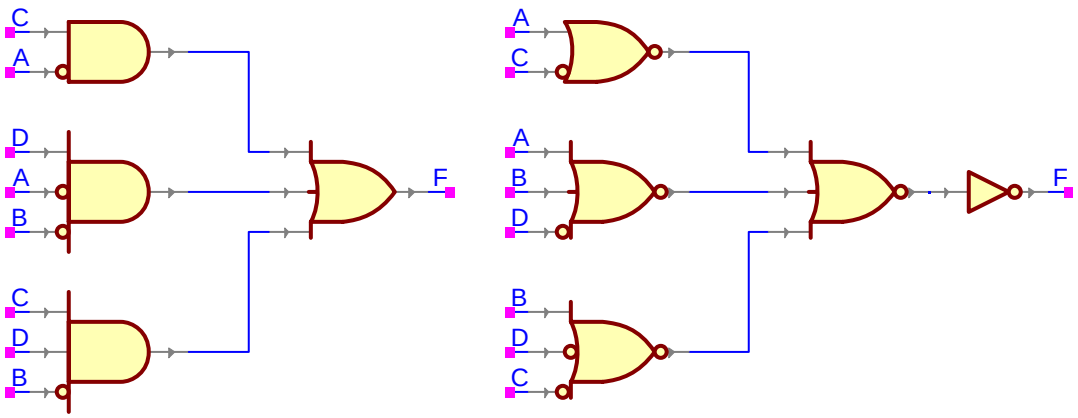
F = A'C + A'B'D + B'CD



Initial Circuit



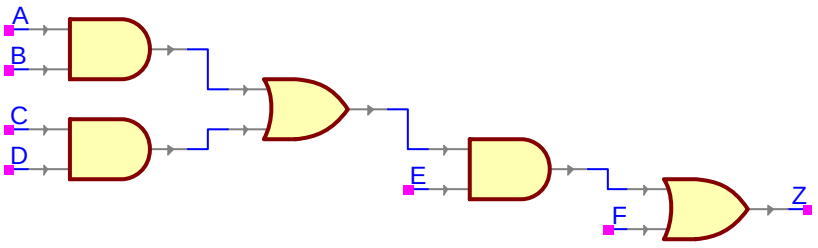
NAND Gate Implementation



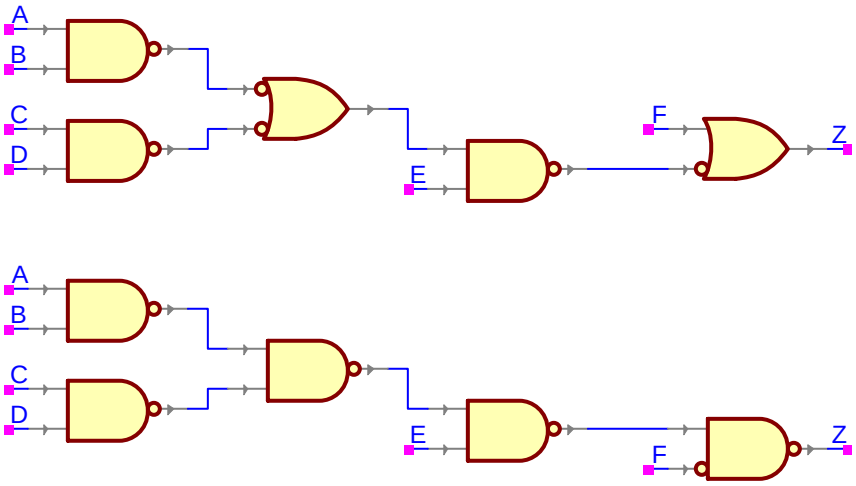
NOR Gate Implementation

Exercise 3.12

(a) $Z = (AB + CD)E + F$

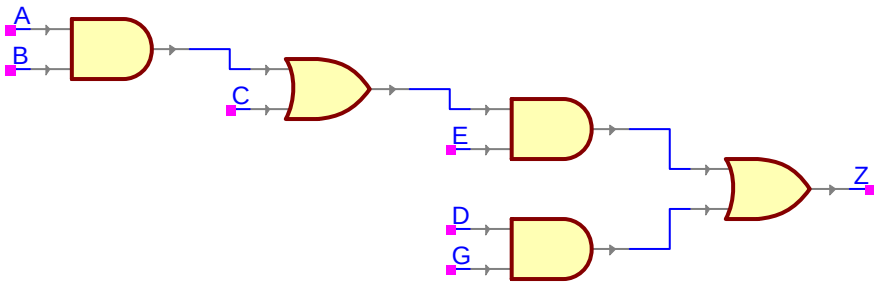


Initial Circuit

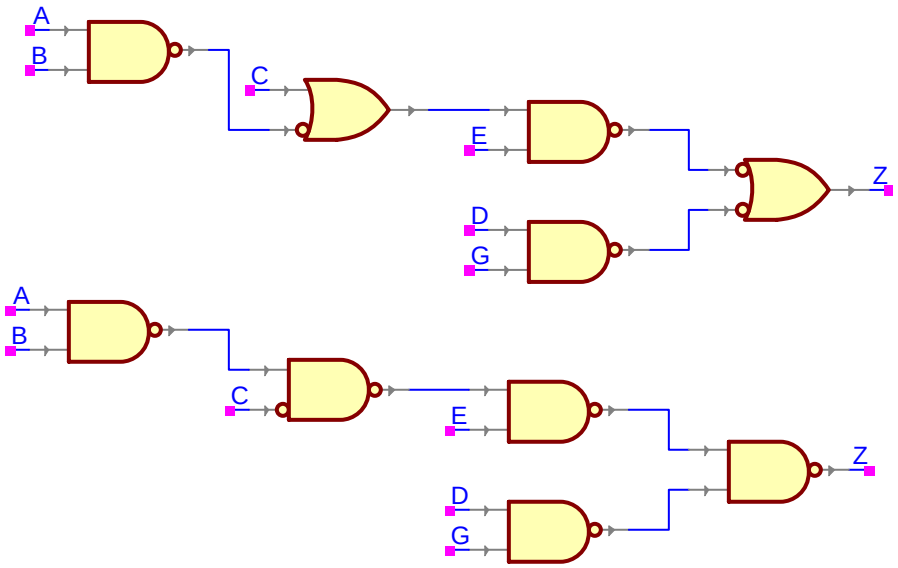


NAND/NOR Gate Implementation

(b) $Z = (AB + C)E + DG$

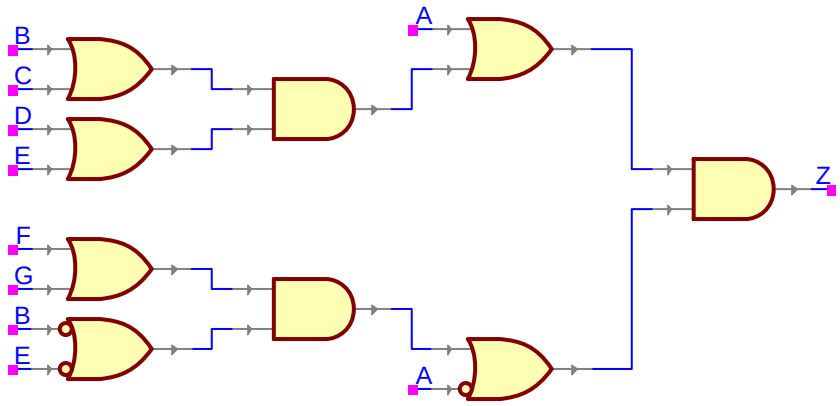


Initial Circuit

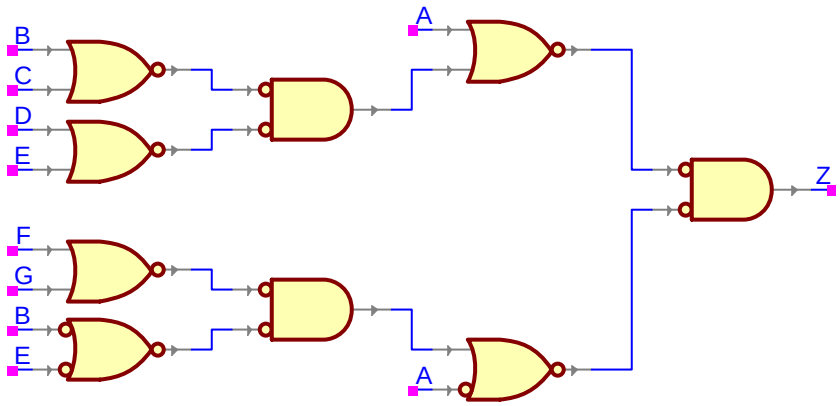


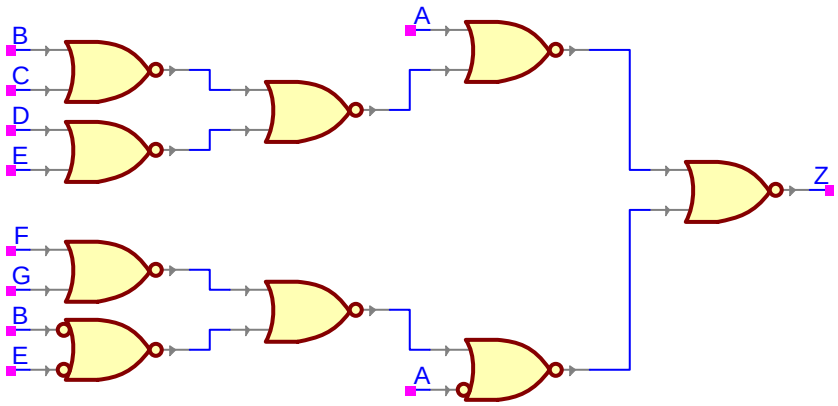
NAND/NOR Gate Implementation

(c) Z = {A + [(B + C)(D + E)]}{[(F + G)(B' + E')] + A'}



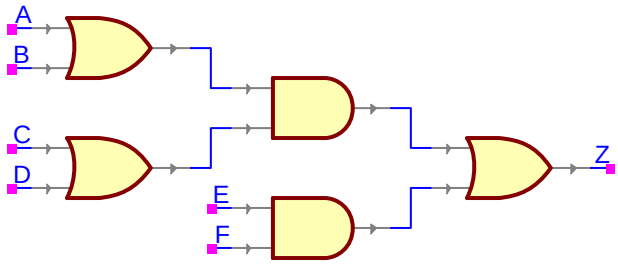
Initial Circuit



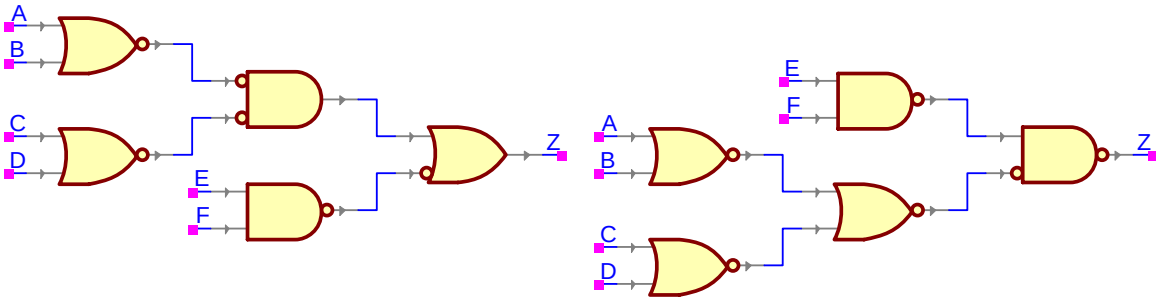


NAND/NOR Gate Implementation

(d) Z = (A + B)(C + D) + EF

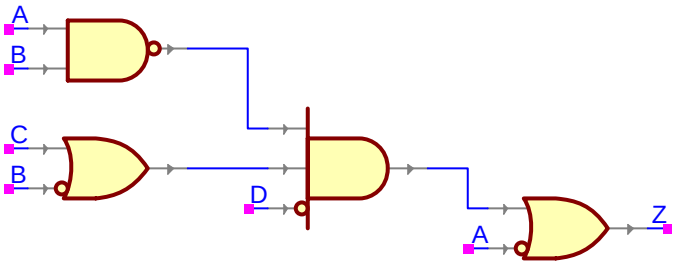


Initial Circuit

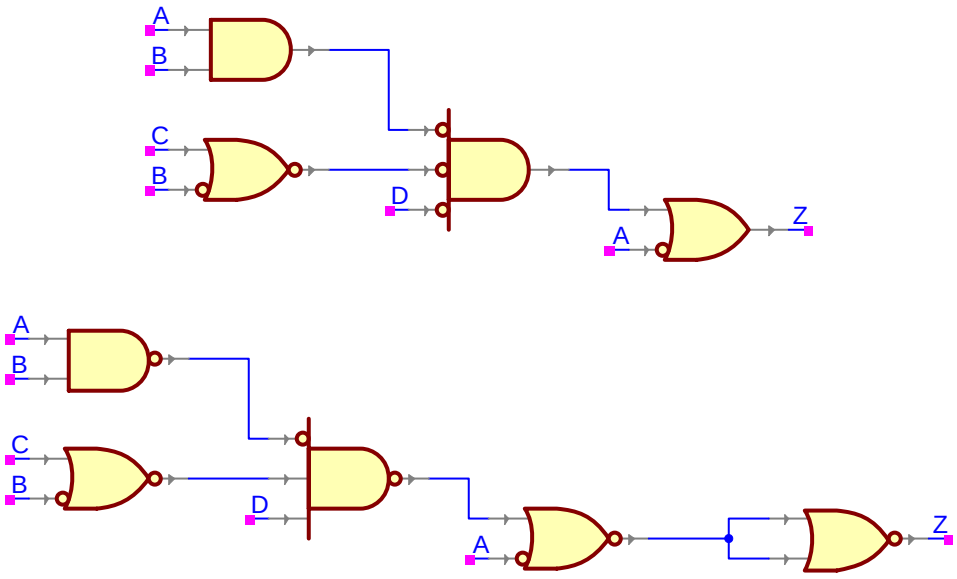


NAND/NOR Gate Implementation

(e) Z = (AB)'(B' + C)D' + A'



NAND/NOR Gate Implementation



NAND/NOR Gate Implementation

Exercise 3.13

We will show the multi-level forms for the full adder are equivalent to the two-level forms using Boolean algebra.

Recall from Section 2.2 that the two-level functions for Sum and Cout are:

Sum = AB'Cin' + A'BCin' + A'B'Cin + ABCin

Cout = ABCin + ABCin' + AB'Cin + A'BCin

The multi-level functions for Sum and Cout are:

Sum = Cin'X' + CinX

Cout = AB + CinY

X = A'B' + AB

Y = A + B

We'll start by negating X to get X' and then substitute these into the function for Sum:

X' = (A'B' + AB)' = (A'B')'(AB)' = (A + B)(A' + B') = AB' + A'B

Sum = Cin(AB' + A'B) + Cin(A'B' + AB) =>
AB'Cin' + A'BCin' + A'B'Cin + ABCin

Next, we'll manipulate Y to get it into the correct form. We will do this by multiplying Y by (A + A') and (B + B'). Recall that this is equivalent to multiplying by one in Boolean algebra.

Y = (A + B)(A + A')(B + B') =>
AAB + AAB' + AA'B + AA'B' + ABB + ABB' + A'BB + A'BB'

Also recall that multiplying a term by its compliment yields 0 and AA is equivalent to A. So, after removing terms that evaluate to zero and combining terms, we are left with:

Y = AB + AB' + A'B

Before we substitute Y into Cout, we have to multiply Cout by (Cin + Cin').

Cout = (AB + CinY)(Cin + Cin') = ABCin + ABCin' + CinCinY + CinCin'Y

Or simply:

Cout = ABCin + ABCin' + CinY

Now we can substitute Y into Cout.

Cout = ABCin + ABCin' + Cin(AB + AB' + A'B) =>
ABCin + ABCin' + AB'Cin + A'BCin

Exercise 3.14

We will use a truth table to show that the multi-level form of the 2-bit adder is equivalent to the multi-level form. The multi-level functions for the 2-bit adder are:

Z = B'D + BD'

Y = W'A'C + W'AC' + WAC + WA'C'

X = AC + W(A + C)

W = BD

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

Exercise 3.15

(a) Boolean representation of the circuit:

Z = (B xor C)(A xor C)' + (A xor B)(B xor C)'

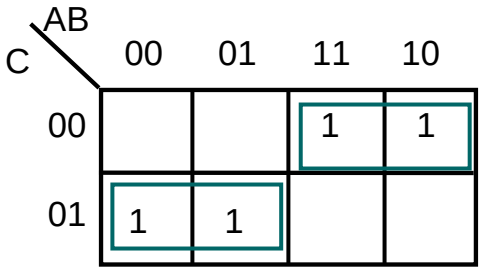
(b) Truth table:

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(c) Sum-of-products function:

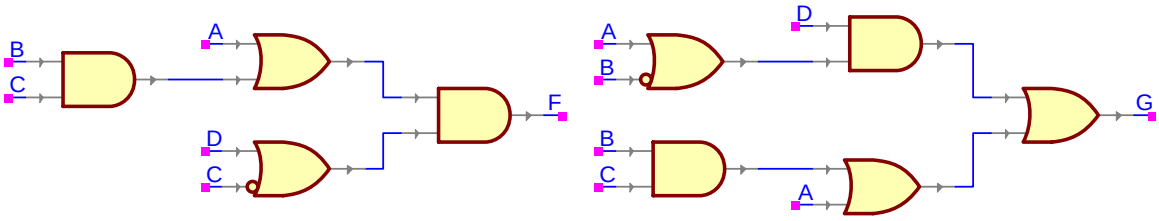
Z = $\sum$ m(1,3,4,6)

(d) K-map:



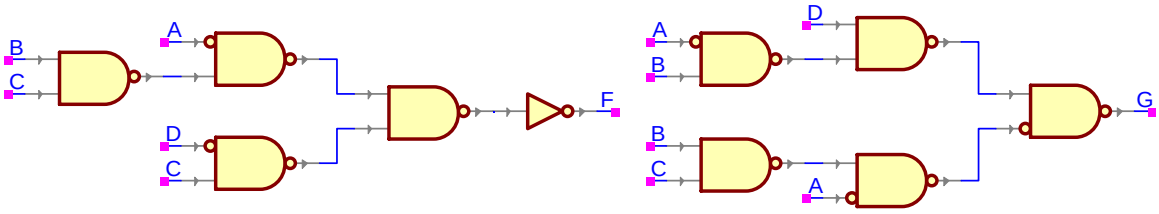
Exercise 3.16

F(A,B,C,D) = (A + (BC))(C' +D)
G(A,B,C,D) = ((A + B')D) + (A + (BC))

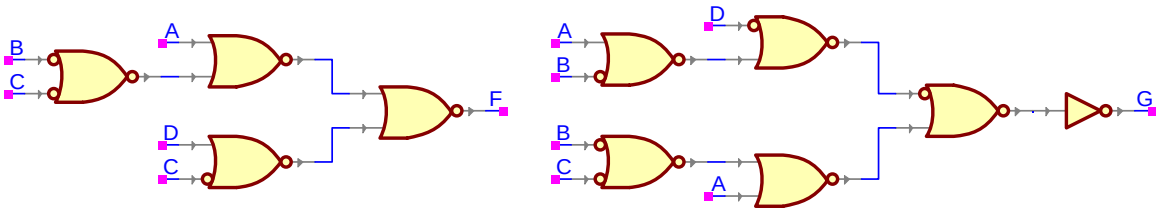


Initial Circuits

(a) NAND-only implementations:



(b) NOR-only implementations:

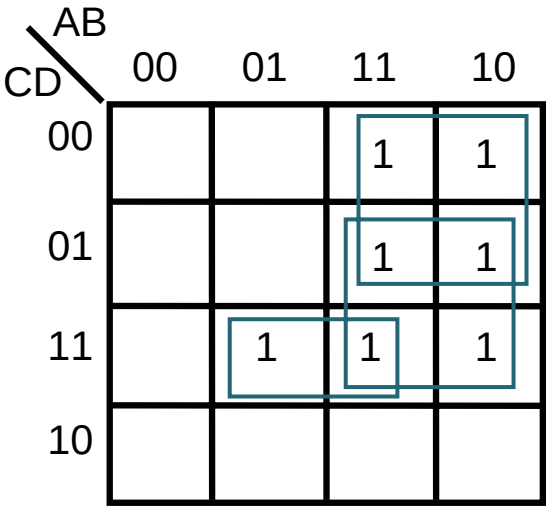


(c) Truth tables for F and G:

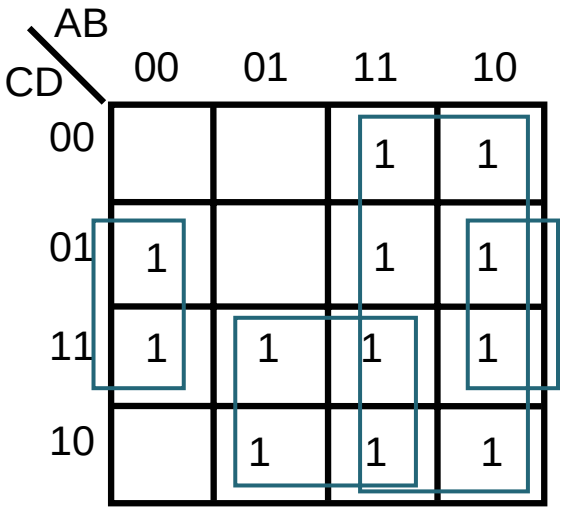
A	B	C	D	F	G
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	1
0	1	1	1	1	1
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	1	1	1
1	1	0	0	1	1

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

K-Maps for F and G:



K-Map for F

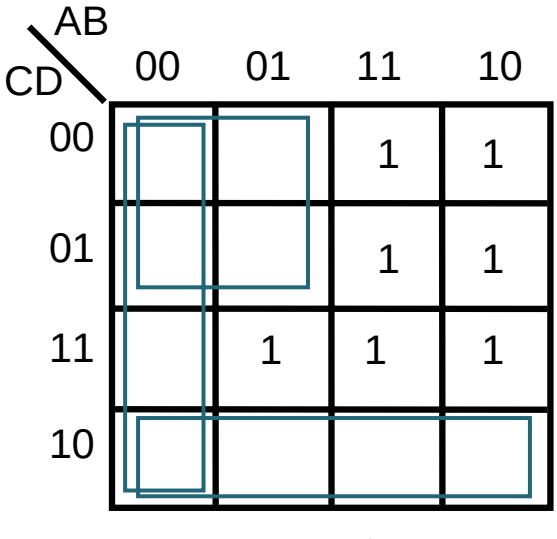


K-Map for G

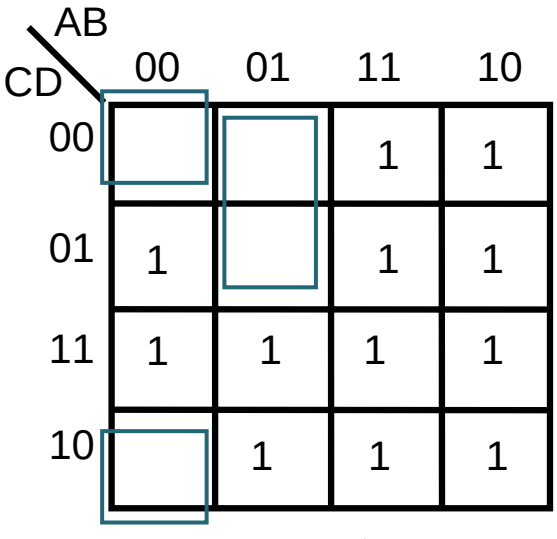
F = AC' + AD + BCD

G = A + BC + B'D

(d) To find the simplified product-of-sums form, we get the sum-of-products for the 0's in the k-maps and then negate them.



K-Map for



K-Map for

F' = A'C' + A'B' + CD'

G = ABC' + A'B'D'

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

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

G'' = (ABC' + A'B'D')' = (ABC')'(A'B'D')' =>

G = (A' + B' + C)(A + B + D)

(e) The implementation complexities for the sum-of products implementations are:

F has 7 literals with two 2-input AND, one 3-input AND, and one 3-input OR gates.

G has 5 literals with two 2-input AND, and one 3-input OR gates.

The complexities for the product-of-sums implementations are:

F has 6 literals with three 2-input AND and one 3-input OR gates.

G also has 6 literals with two 3-input AND and one 2-input OR gates.

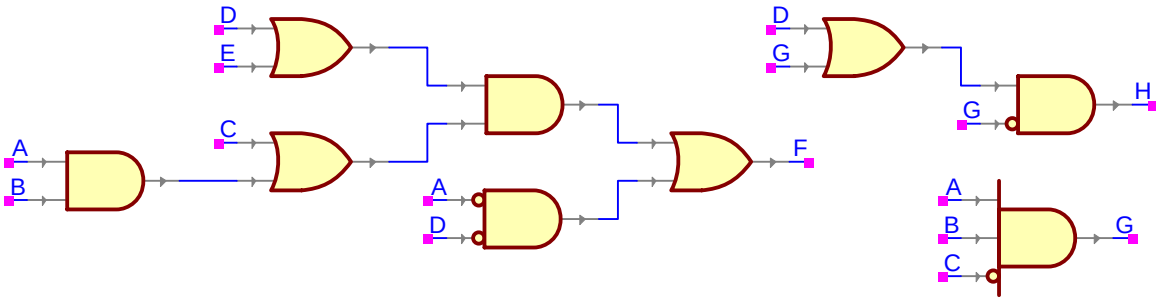
So in both cases the product-of-sums implementation was less complex.

Exercise 3.17

F(A,B,C,D,E) = (((AB) + C)(D + E)) + (A'D'))

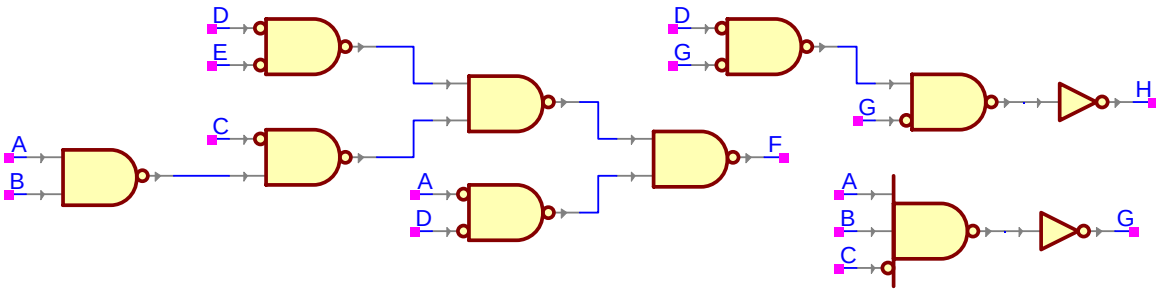
G(A,B,C) = (ABC')

H(A,B,C,D) = (D + G)G'

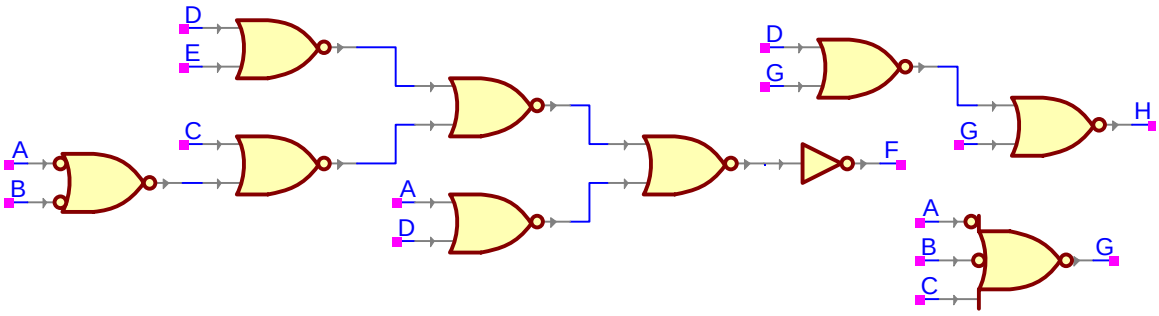


Initial Circuits

(a) NAND-only implementations:



(b) NOR-only implementations:



(c) Minimized Sum-of-Products:

Truth tables for F, G, and H:

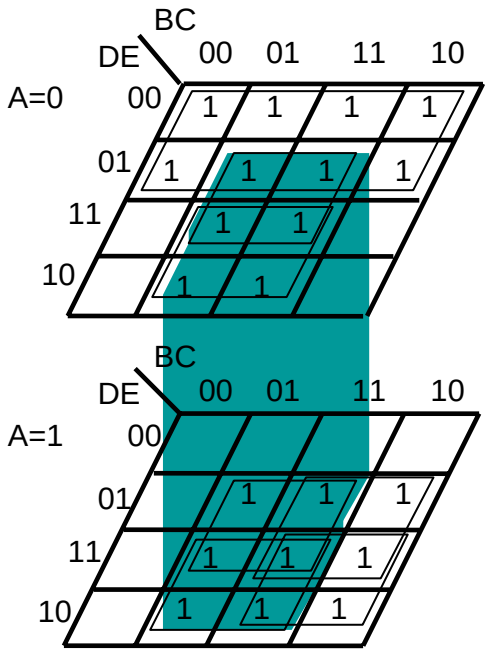
A	B	C	D	E	F	G	H
0	0	0	0	0	1	0	0
0	0	0	0	1	1	0	0
0	0	0	1	0	0	0	1
0	0	0	1	1	0	0	1
0	0	1	0	0	1	0	0
0	0	1	0	1	1	0	0
0	0	1	1	0	1	0	1
0	0	1	1	1	1	0	1
0	1	0	0	0	1	0	0
0	1	0	0	1	1	0	0
0	1	0	1	0	0	0	1
0	1	0	1	1	0	0	1
0	1	1	0	0	1	0	0
0	1	1	0	1	1	0	0
0	1	1	1	0	1	0	1
0	1	1	1	1	1	0	1
1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0
1	0	0	1	0	0	0	1
1	0	0	1	1	0	0	1
1	0	1	0	0	0	0	0
1	0	1	0	1	1	0	0
1	0	1	1	0	1	0	1
1	0	1	1	1	1	0	1
1	1	0	0	0	0	1	0
1	1	0	0	1	1	1	0
1	1	0	1	0	1	1	0
1	1	0	1	1	1	1	0
1	1	1	0	0	0	0	0
1	1	1	0	1	1	0	0
1	1	1	1	0	1	0	1
1	1	1	1	1	1	0	1

$F(A,B,C,D,E) = \Sigma\, m(0,1,4,5,6,7,8,9,12,13,14,15,21,22,23,25,26,27,29,30,31)$

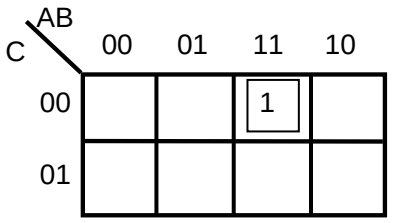
$G(A,B,C) = \Sigma\, m(6)$

$H(A,B,C,D) = \Sigma\, m(1,3,5,7,9,11,15)$

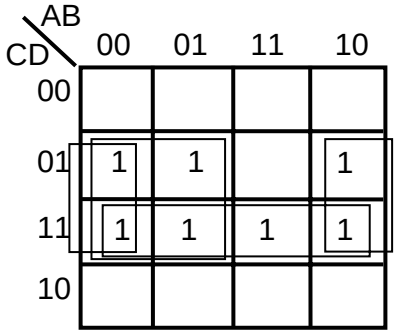
K-Maps:



K-Map for F



K-Map for G



K-Map for H

F(A,B,C,D,E) = A'D' + CE + CD + ABD + ABE

G(A,B,C) = ABC'

H(A,B,C,D) = A'D + B'D + CD

(d) Minimized Product-of-Sums:

F’ = A’C’D + B’C’DE’ + AB’C’ + AD’E’

F’’ = (A’C’D)’(B’C’DE’)’(AB’C’)’(AD’E’)’

F(A,B,C,D,E) = (A + C + D’)(B + C + D’ + E)(A’ + B + C)(A’ + D + E)

G’ = A’ + B’ + C

G’’ = (A’ + B’ + C)’

G(A,B,C) = ABC’

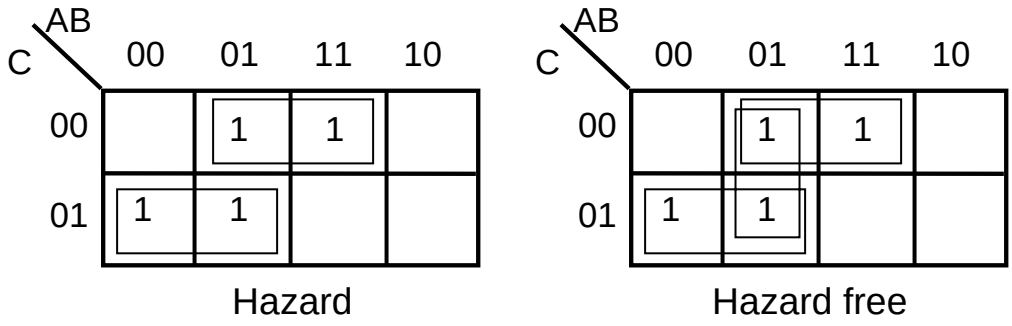
H’ = D’ + ABC’

H’’ = (D’)’(ABC’)’

H(A,B,C,D) = D(A’ + B’ + C)

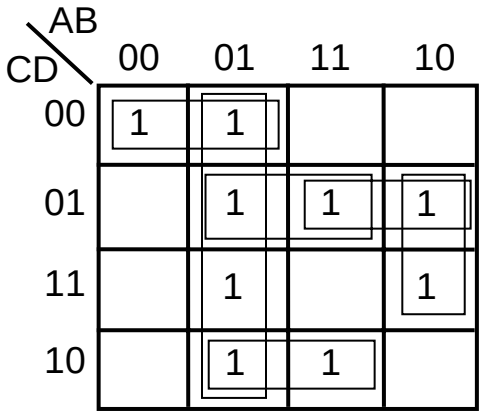
Exercise 3.18

(a) The K-map for this function shows that there are adjacent groups 1's without a circle:



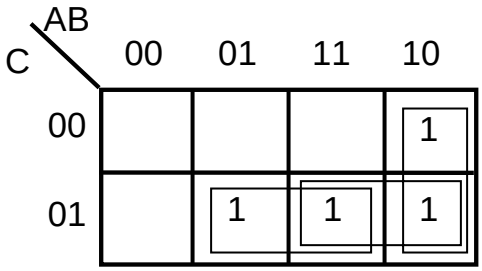
F(A,B,C) = A'B + A'C + BC'

(b) The K-map for F:



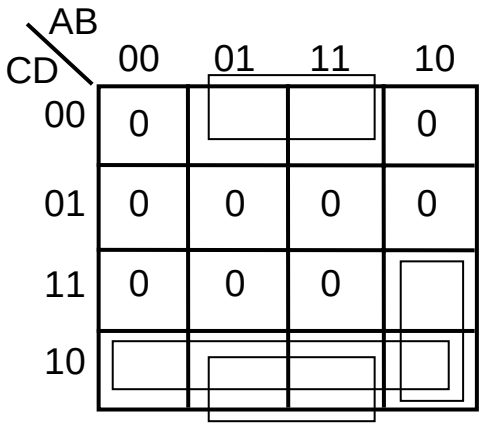
F(A,B,C,D) = A'C'D' + AC'D + AB'D + BC'D + BCD' + A'B

(c) The K-map for F:



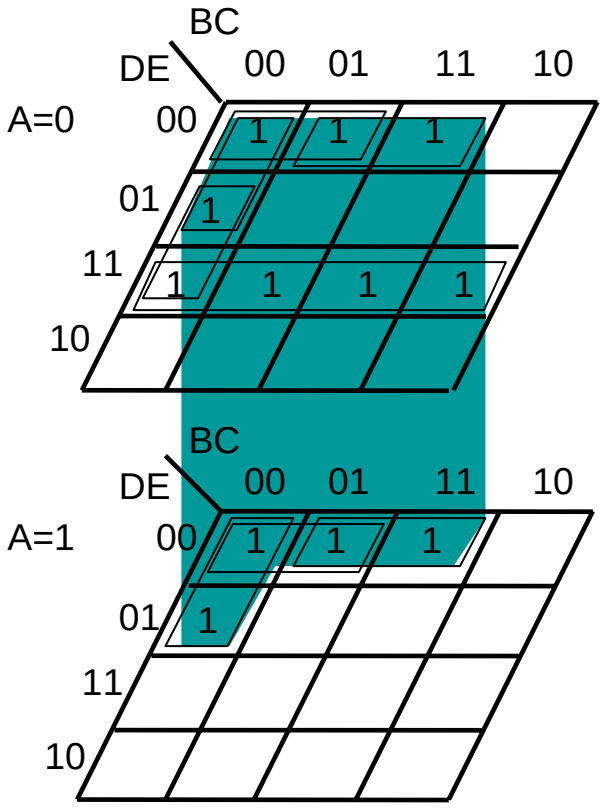
F(A,B,C) = AB' + AC + BC

(d) The K-map for F:



F(A,B,C,D) = BD’ + CD’ + AB’C

(e) The K-map for F:



F(A,B,C,D,E) = A’DE + B’D’E’ + B’C’D’ + B’C’E + CD’E’

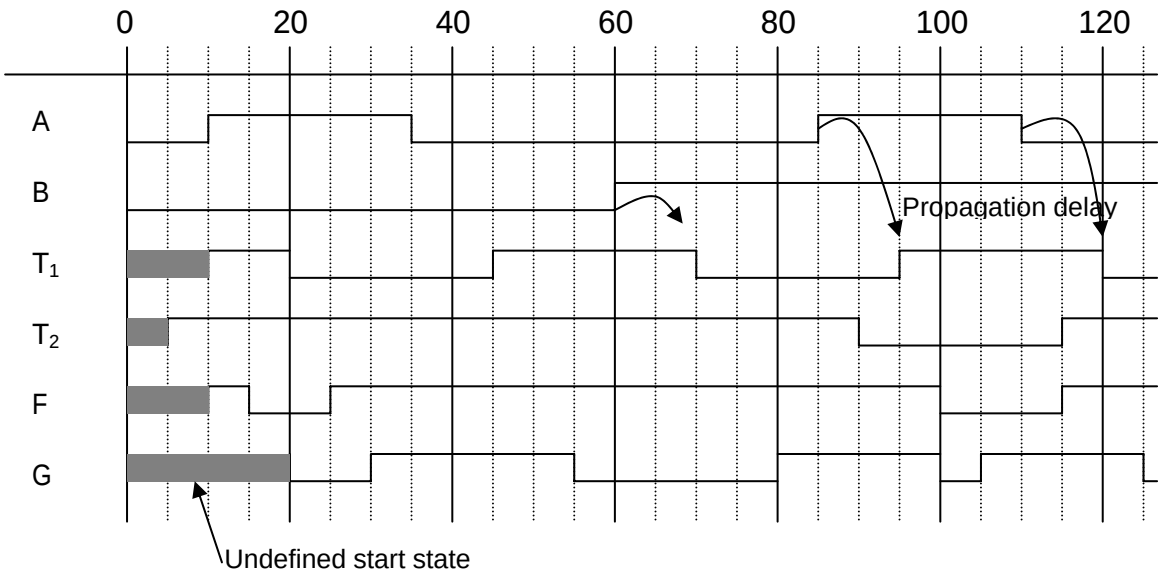
Exercise 3.19

In this output response waveform, T_1 represents the output of the first XOR gate and T2 represents the output of the first NAND gate. The propagation delay for the XOR gate is 10 time units and the delay for the NAND gate is 10 time units. We are assuming that T_{PHL} is equal to T_{PLH}.

Here is the truth table for F and G:

A	B	F	G
0	0	1	0
0	1	1	1
1	0	1	1
1	1	0	1

Output response waveform:



Exercise 3.20

(a) The steady-state starting condition for the circuit is as follows:

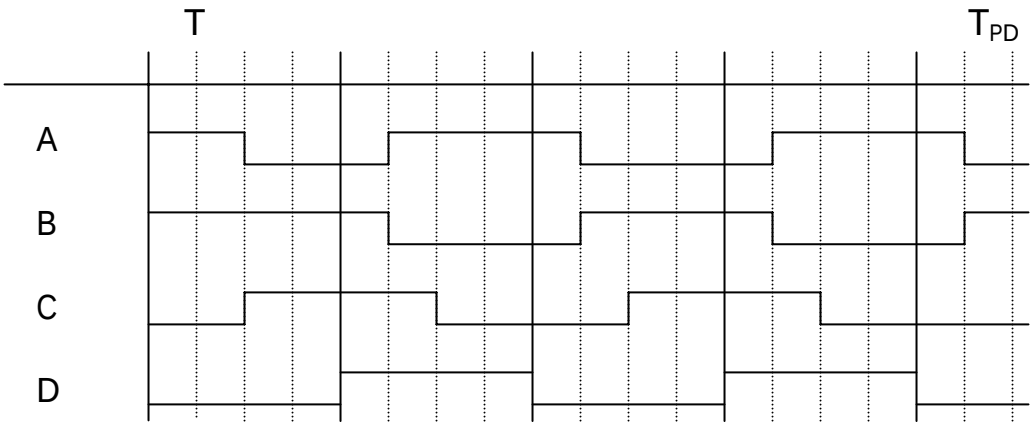
A = 1

B = 1

C = 0

D = 0

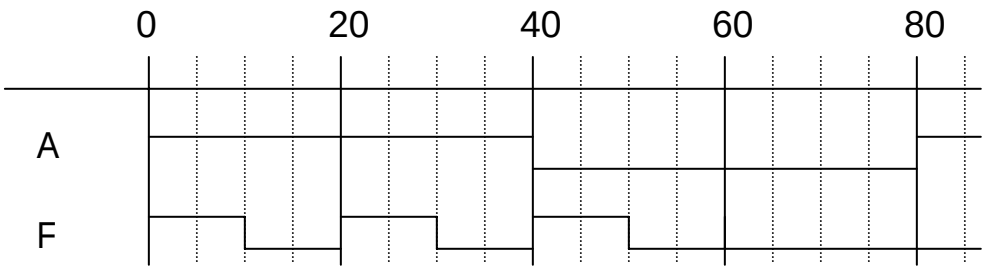
(b) Output response wave form after time T:



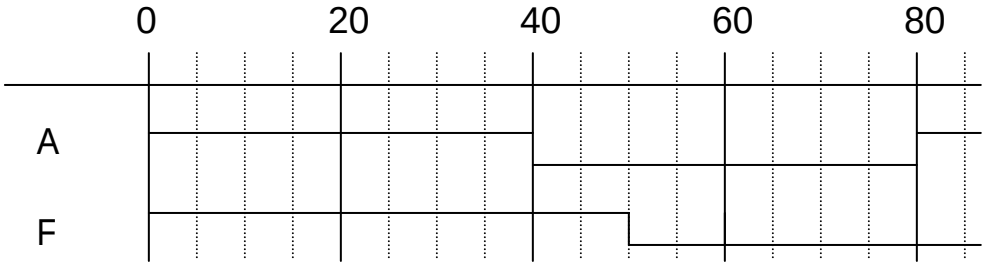
Exercise 3.21

In each of the output waveforms, the oscillating signal from the switch is A, and the output from the NOR gate is F.

(a) This circuit oscillates when the switch is open and is steady when the switch is closed.



(b) This circuit does not oscillate when the switch is open or closed. The initial output of the NOR gate could either be high or low when the switch is open.



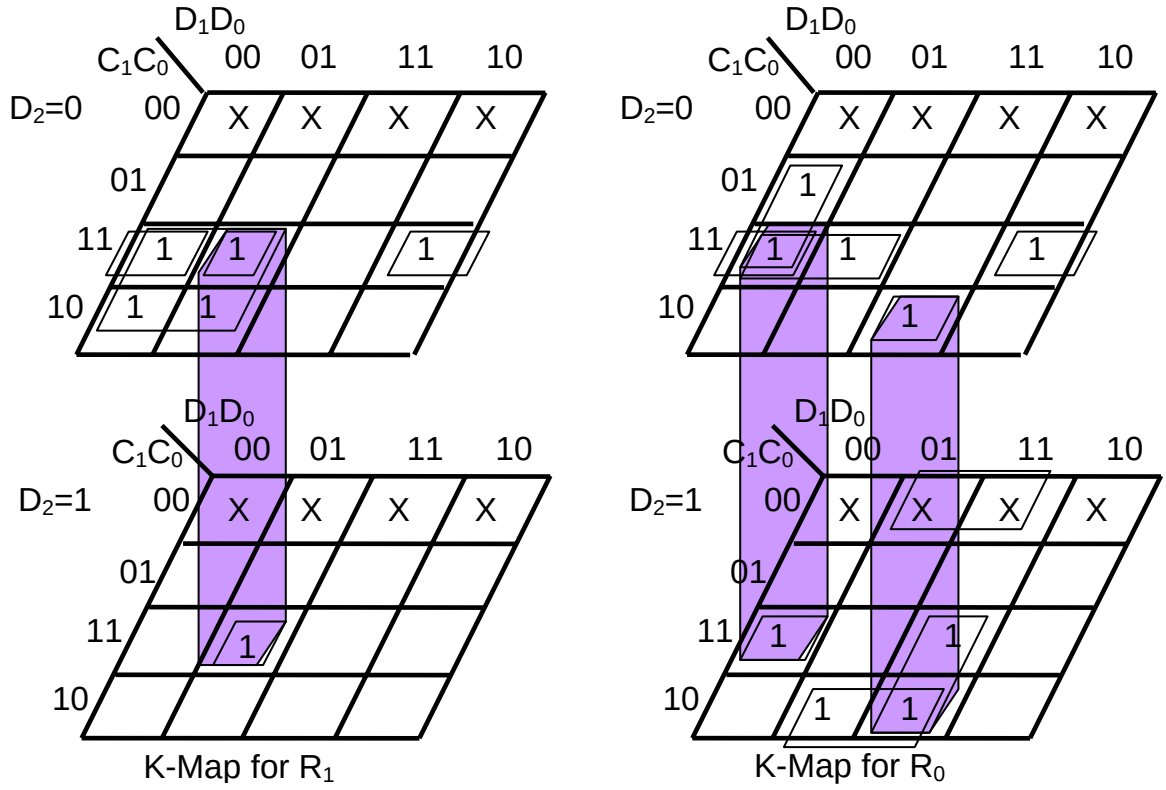
Exercise 3.22

(a) Here is the truth table for R_0 and R_1:

Note that since division by zero will never be requested, there are eight don't-cares for both R_1 and R_0.

D_2	D_1	D_0	C_1	C_0	R_1	R_0
0	0	0	0	0	X	X
0	0	0	0	1	0	1
0	0	0	1	0	1	0
0	0	0	1	1	1	1
0	0	1	0	0	X	X
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	0	1	1	1	1	1
0	1	0	0	0	X	X
0	1	0	0	1	0	0
0	1	0	1	0	0	0
0	1	0	1	1	1	1
0	1	1	0	0	X	X
0	1	1	0	1	0	0
0	1	1	1	0	0	1
0	1	1	1	1	0	0
1	0	0	0	0	X	X
1	0	0	0	1	0	0
1	0	0	1	0	0	0
1	0	0	1	1	0	1
1	0	1	0	0	X	X
1	0	1	0	1	0	0
1	0	1	1	0	0	1
1	0	1	1	1	1	0
1	1	0	0	0	X	X
1	1	0	0	1	0	0
1	1	0	1	0	0	0
1	1	0	1	1	0	0
1	1	1	0	0	X	X
1	1	1	0	1	0	0
1	1	1	1	0	0	1
1	1	1	1	1	0	1

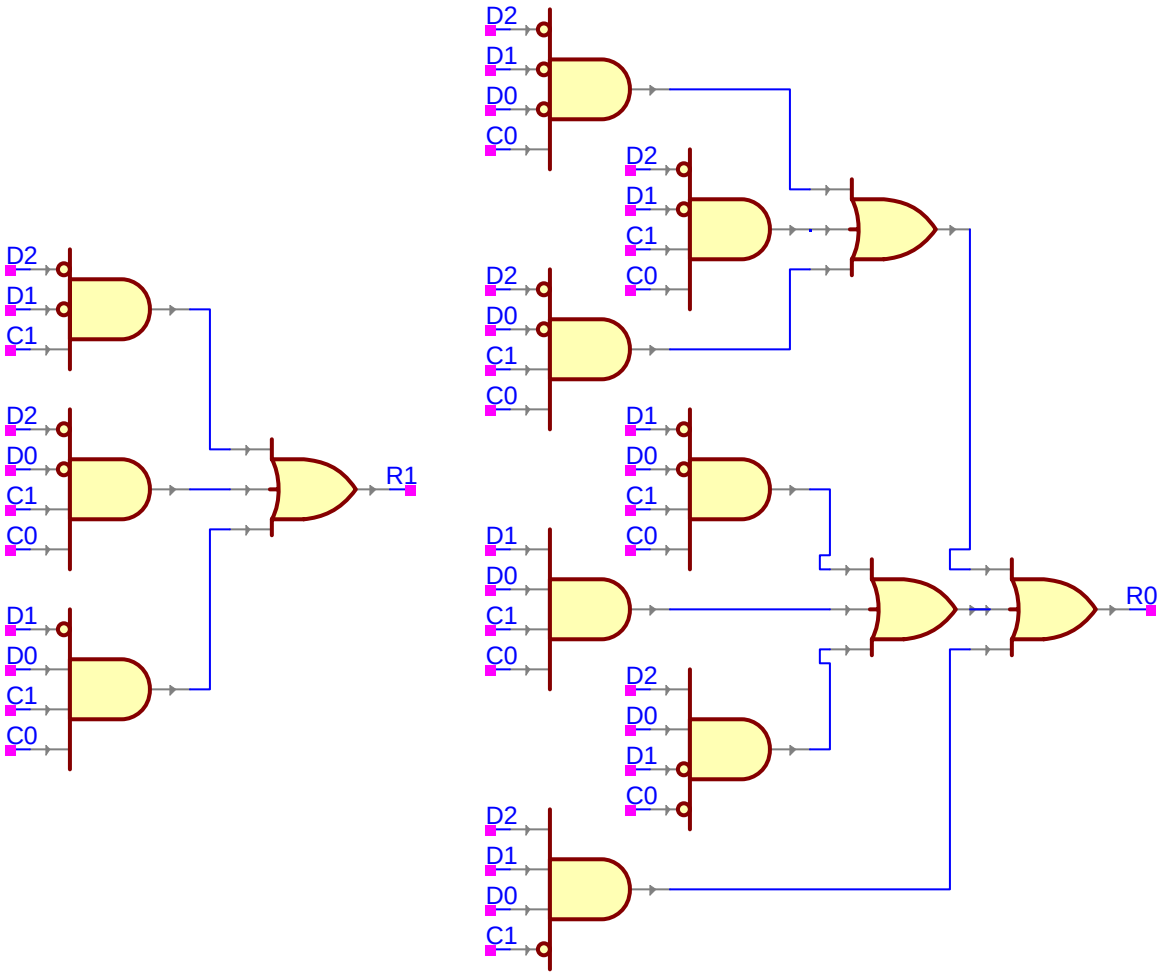
(b) K-Maps for R_1 and R_0:



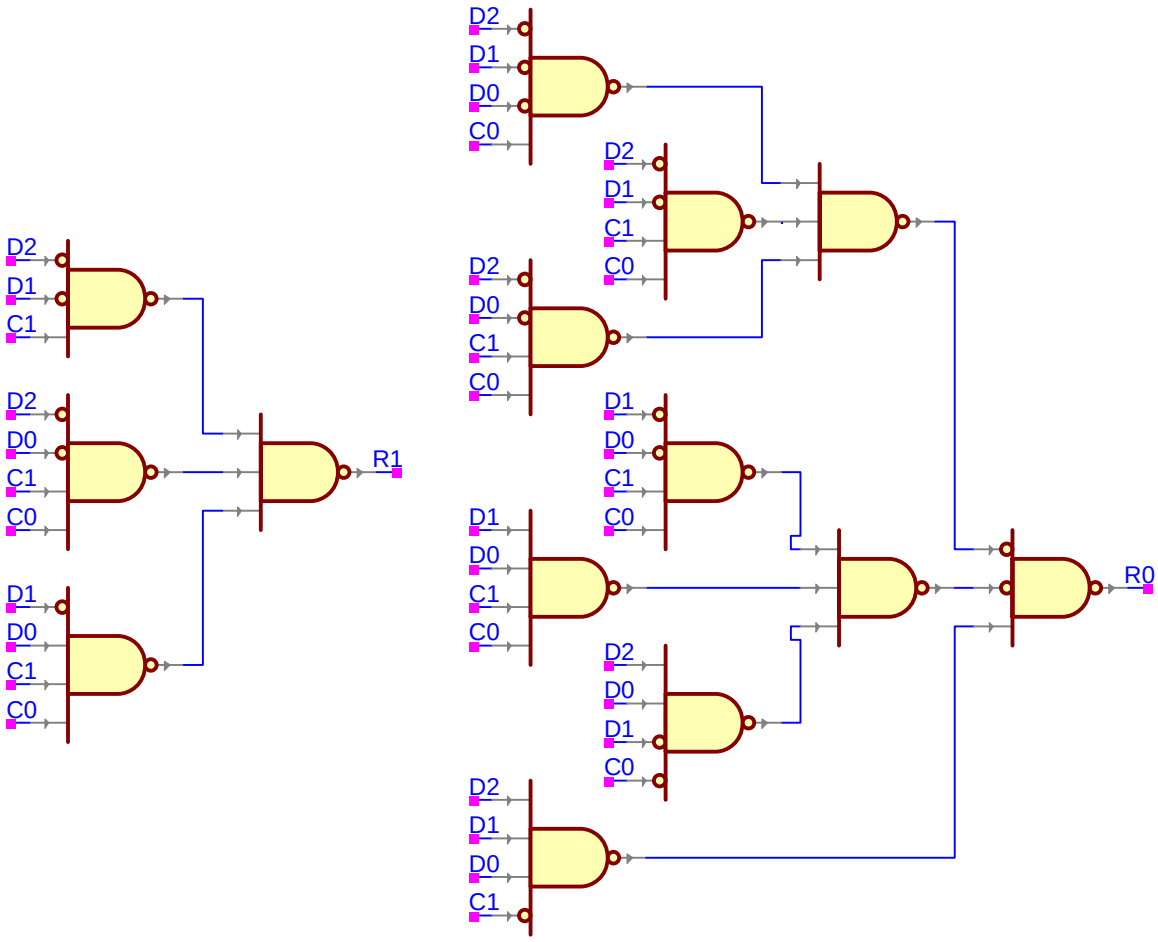
$R_1 = D_2'D_1'C_1 + D_2'D_0'C_1C_0 + D_1'D_0C_1C_0$

$R_0 = D_2'D_1'D_0'C_0 + D_2'D_1'C_1C_0 + D_2'D_0'C_1C_0 + D_1'D_0'C_1C_0 + D_1D_0C_1C_0' + D_2D_1D_0C_1 + D_2D_0C_0'$

(c) Circuit Schematics:



Initial Circuits



NAND-gate implementation

(d) There is a definite advantage when sharing expressions with this design. A lot of the expressions share subexpressions, which makes the overall design less expensive as well as reducing the fan-in to some of the logic gates.

$R_1 = C1(D_2'D_1' + C_0(D_2'D_0' + D_1'D_0))$

$R_0 = C_1C_0(D_2'D_1' + D_2'D_0' + D_1'D_0') + D_1D_0(C_1C_0' + D_2C_1) + D_1'(D_2'D_0'C_0 + D_2D_0C_0')$

Exercise 3.23

(a) Since division by zero is illegal, we will assume that this will not happen. If it does, we don't care what happens. Here is the truth table for WX and YZ:

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

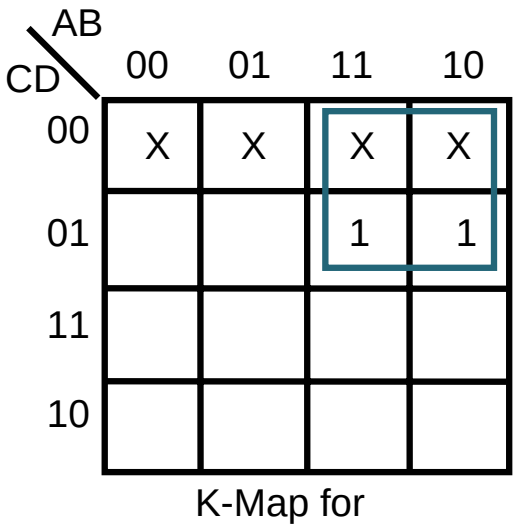
(b) Minimized sum-of-products:

$W(A,B,C,D) = \Sigma\, m(9,13) + \Sigma\, d(0,4,8,12)$

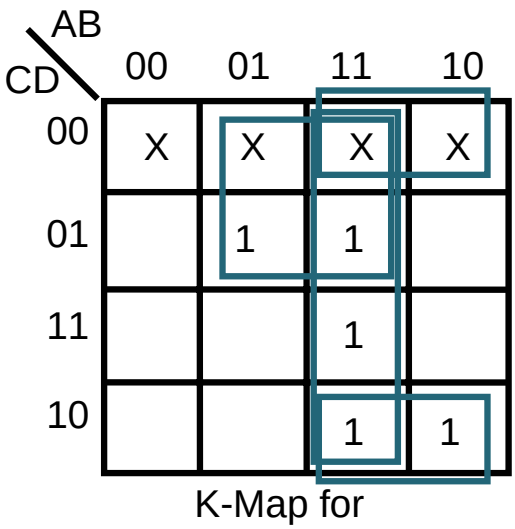
$X(A,B,C,D) = \Sigma\, m(5,10,13,14,15) + \Sigma\, d(0,4,8,12)$

$Y(A,B,C,D) = \Sigma\, m(2,3,6,7,11) + \Sigma\, d(0,4,8,12)$

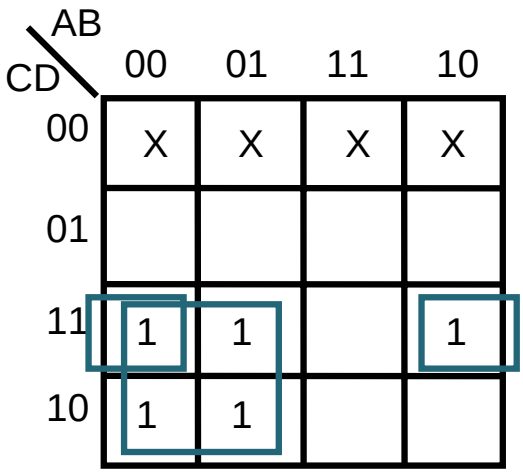
$Z(A,B,C,D) = \Sigma\, m(1,3,7,11,14) + \Sigma\, d(0,4,8,12)$



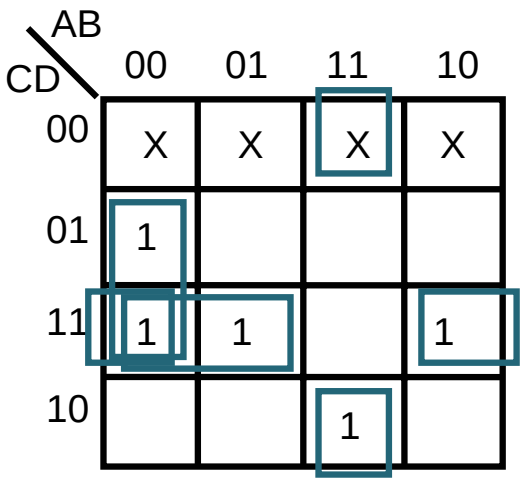
K-Map for



K-Map for



K-Map for Y



K-Map for Z

W(A,B,C,D) = AC’

X(A,B,C,D) = AB + BC’ + AD’

Y(A,B,C,D) = A’C + B’CD

Z(A,B,C,D) = ABC’ + A’CD + A’B’D + B’CD

(c) Minimized product-of-sums:

W’ = A’ + C

W’’ = (A’ + C)’

W = AC’

X’ = B’D + A’C

X’’ = (B’D + A’C)’ = (B’D)’(A’C)’

X = (B + D’)(A + C’)

Y’ = C’ + AB + AD’

Y’’ = (C’ + AB + AD’)’ = (C’)’(AB)’(AD’)’

Y = C(A’ + B’)(A’ + D)

Z’ = A’D’ + BC’ + AC’ + B’D’ + ABD

Z’’ = (A’D’ + BC’ + AC’ + B’D’ + ABD)’ = (A’D’)’(BC’)’(AC’)’(B’D’)’(ABD)’

Z = (A + D)(B’ + C)(A’ + C)(B + D)(A’ + B’ + D’)

Exercise 3.24

(a) Truth tables for By2, By3, and By6:

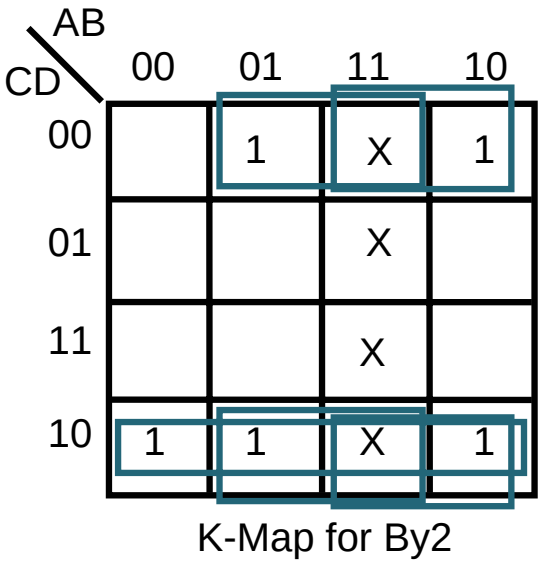
A	B	C	D	By2	By3	By6
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	1	0	0	1	0	0
0	1	0	1	0	0	0
0	1	1	0	1	1	1
0	1	1	1	0	0	0
1	0	0	0	1	0	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	0	1	1	0	0	0
1	1	0	0	X	X	X
1	1	0	1	X	X	X
1	1	1	0	X	X	X
1	1	1	1	X	X	X

(b) Minimized sum-of-products:

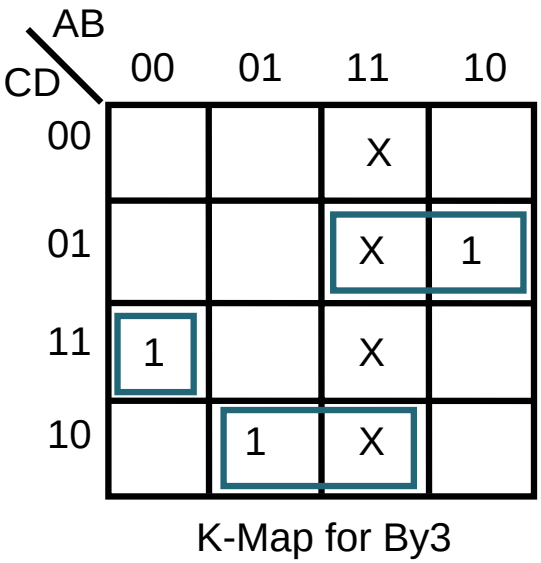
$By2 = \Sigma\, m(2,4,6,8,10) + \Sigma\, d(12,13,14,15)$

$By3 = \Sigma\, m(3,6,9) + \Sigma\, d(12,13,14,15)$

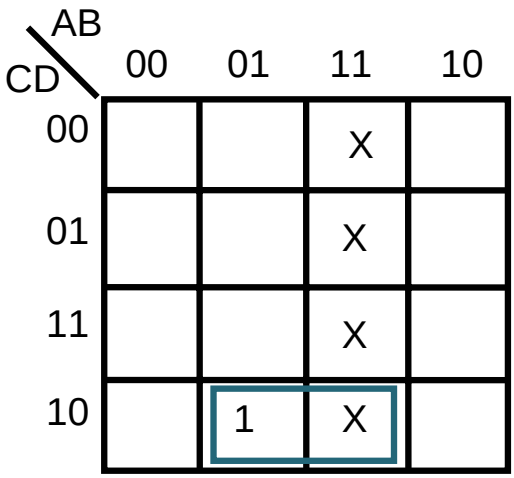
$By6 = \Sigma\, m(6) + \Sigma\, d(12,13,14,15)$



K-Map for By2



K-Map for By3



K-Map for By6

By2(A,B,C,D) = AD’ + BD’ + C D’

By3(A,B,C,D) = AC’D + BCD’ + A’B’CD

By6(A,B,C,D) = BCD’

(c) Yes, By2 could be simplified by factoring out the D’. Also, By3 could be simplified by using the result from By6 for its second expression.

Exercise 3.25

(a) Truth table for X, Y, and Z:

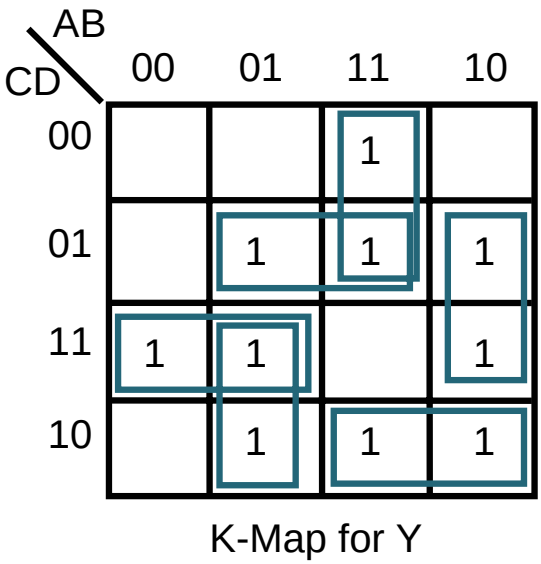
A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	0
0	1	1	1	0	1	1
1	0	0	0	0	0	1
1	0	0	1	0	1	0
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	0	1	0
1	1	0	1	0	1	1
1	1	1	0	0	1	1
1	1	1	1	1	0	0

(b) Minimized sum-of-products:

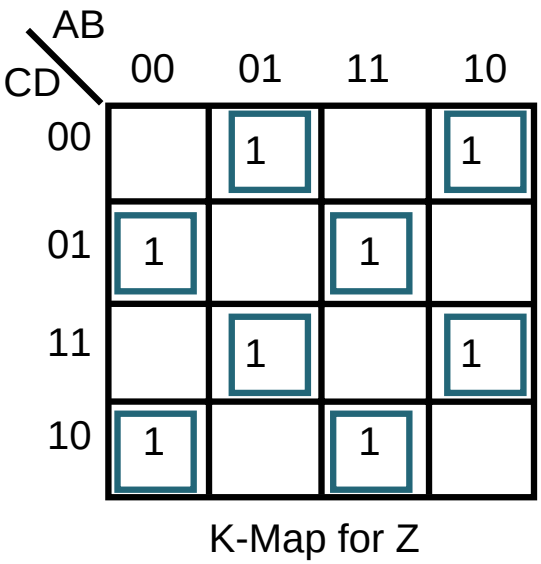
$X(A,B,C) = \Sigma\, m(15)$

$Y(A,B,C) = \Sigma\, m(3,5,6,7,9,10,11,12,13,14)$

$Z(A,B,C) = \Sigma\, m(1,2,4,7,8,11,13,14)$



K-Map for Y



K-Map for Z

Since X only has one minterm, there is no need to construct a K-map for it.

X(A,B,C) = ABCD

Y(A,B,C) = ABC’ + AC’D + BC’D + B’CD + A’BC + ACD’

Z(A,B,C) = A’B’C’D + A’B’CD’ + A’BC’D’ + A’BCD + AB’C’D’ + AB’CD + ABC’D + ABCD’

(c) Minimzed product-of-sums:

X’ = A’ + B’ + C’ + D’

X’’ = (A’ + B’ + C’ + D’)’

X = ABCD

Y’ = A’C’D’ + A’B’C’ + A’B’D’ + B’C’D’ + ABCD

Y’’ = (A’C’D’ + A’B’C’ + A’B’D’ + B’C’D’ + ABCD)’

Y = (A + C + D)(A + B +C)(A + B + D)(B + C + D)(A’ + B’ + C’ + D’)

Z’ = A’B’C’D’ + A’B’CD + A’BC’D + ABC’D’ + A’BCD’ + ABCD + AB’C’D + AB’CD’

Z’’ = (A’B’C’D’ + A’B’CD + A’BC’D + ABC’D’ + A’BCD’ + ABCD + AB’C’D + AB’CD’)’

Z = (A + B + C + D)(A + B + C’ + D’)(A + B’ + C + D’)(A’ + B’ + C + D)(A + B’ + C’ + D)(A’ + B’ + C’ + D’)(A’ + B + C + D’)(A’ + B + C’ + D)

Exercise 3.26

(a) Truth table for the 3-bit sum XYZ:

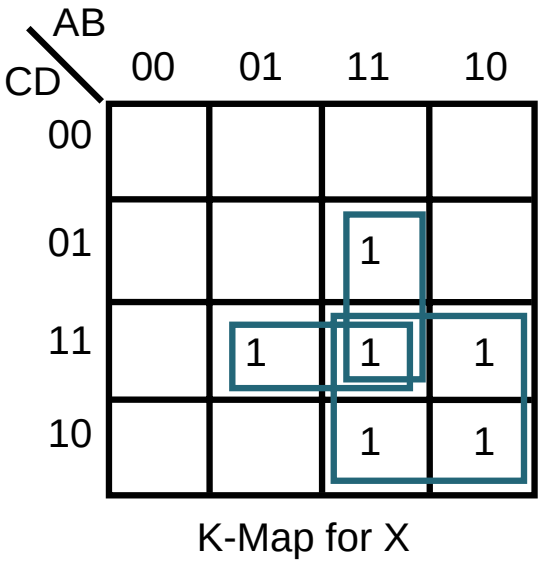
A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

(b) Minimized sum-of-products:

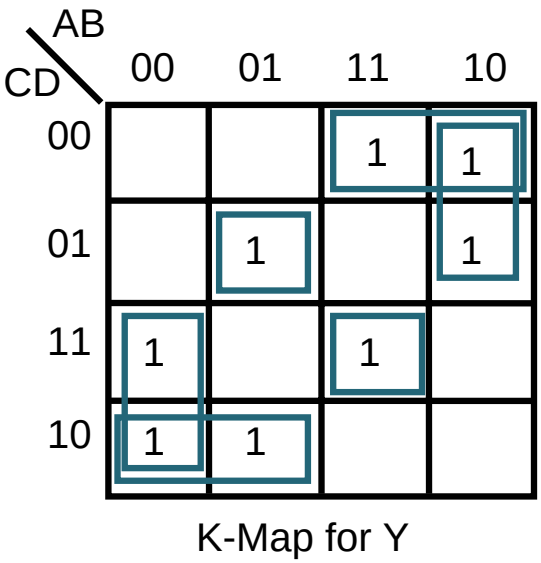
$X(A,B,C,D) = \Sigma\, m(7,10,11,13,14,15)$

$Y(A,B,C,D) = \Sigma\, m(2,3,5,6,8,9,12,15)$

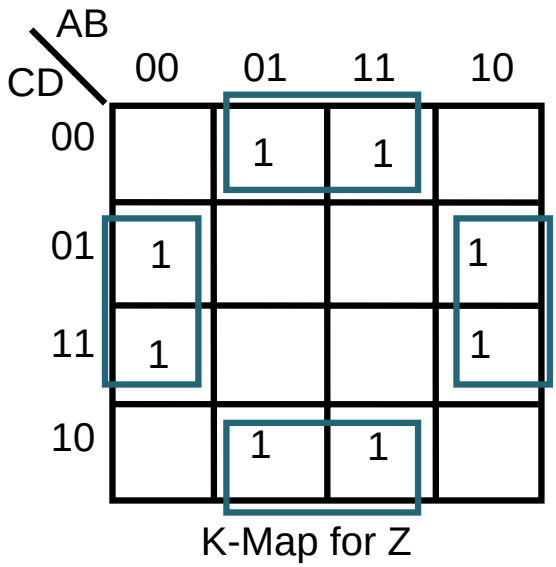
$Z(A,B,C,D) = \Sigma\, m(1,3,4,6,9,11,12,14)$



K-Map for X



K-Map for Y



X(A,B,C,D) = AC + BCD + ABD

Y(A,B,C,D) = AC'D' + AB'C' + A'B'C + A'CD' + A'BC'D + ABCD

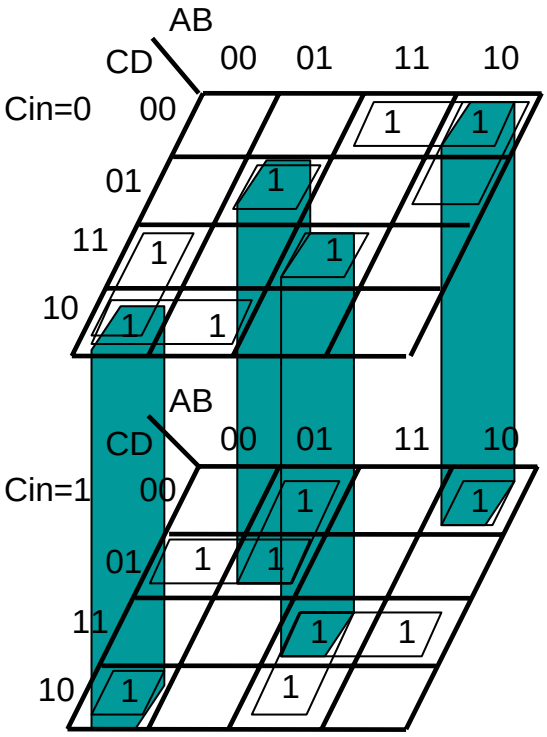
Z(A,B,C,D) = BD' + B'D

(c) The full adder has a worse delay than the 2-bit adder that was designed in this problem. The critical path for the 2-bit adder in this exercise is 2 gate delays because it was implemented with two-level logic. The critical path for the full adder is the carry out, which is n + 2, where n is the number of bits being summed. So a 2-bit full adder would have a delay of 4 gates.

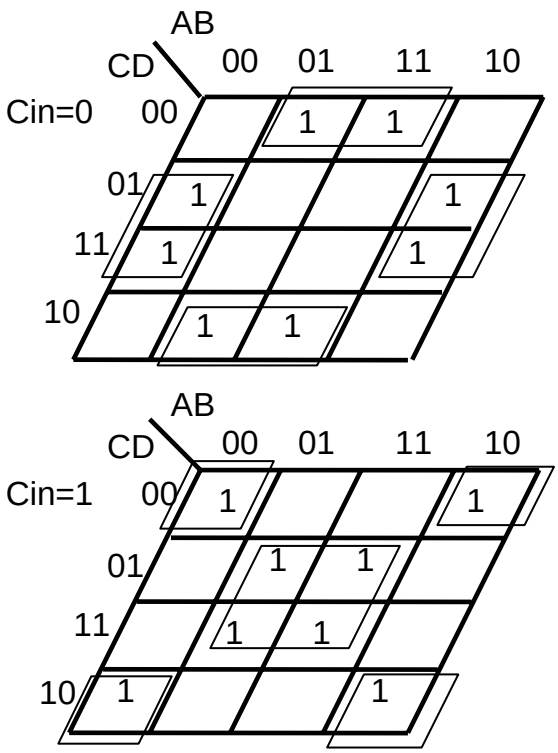
Exercise 3.27

Truth table for the generalized 2-bit adder:

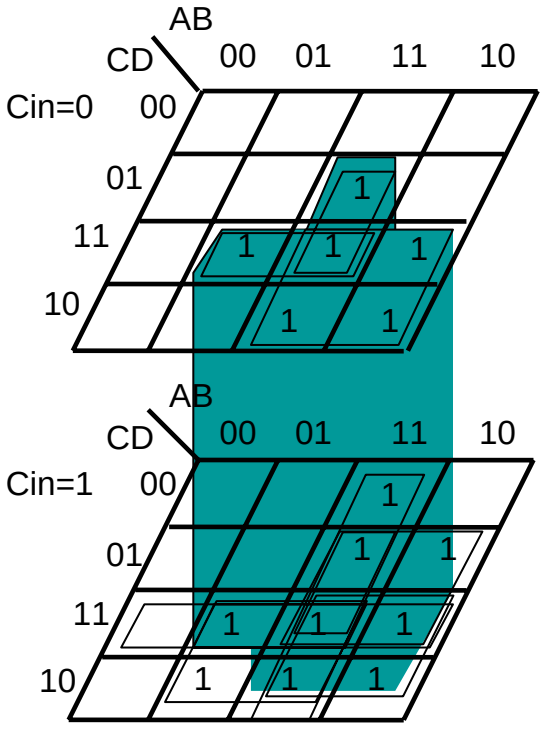
Cin	A	B	C	D	Cout	Y	Z
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	0	1	1	0	1	1
0	0	1	0	0	0	0	1
0	0	1	0	1	0	1	0
0	0	1	1	0	0	1	1
0	0	1	1	1	1	0	0
0	1	0	0	0	0	1	0
0	1	0	0	1	0	1	1
0	1	0	1	0	1	0	0
0	1	0	1	1	1	0	1
0	1	1	0	0	0	1	1
0	1	1	0	1	1	0	0
0	1	1	1	0	1	0	1
0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1
1	0	0	0	1	0	1	0
1	0	0	1	0	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	0	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	0	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	0	1	1
1	1	0	0	1	1	0	0
1	1	0	1	0	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	0	1	0	0
1	1	1	0	1	1	0	1
1	1	1	1	0	1	1	0
1	1	1	1	1	1	1	1



K-Map for Y



K-Map for Z



K-Map for Cout

Cout(A,B,C,D,Cin) = AC + ABD + BCD + ADCin + BCCin + ABCin + CDCin

Y(A,B,C,D,Cin) = A’B’CCin’ + A’CD’Cin’ + AC’D’Cin’ + AB’C’Cin’ + AB’C’D’ + A’B’CD’ + A’BC’D+ ABCD + A’BC’Cin + A’C’DCin + ACDCin + ABCCin

Z(A,B,C,D,Cin) = BD’Cin’ + B’DCin + BDCin + B’D’Cin

Exercise 3.28

Verilog has the modulo operator, %, that provides the functionality that we are trying to achieve in Exercise 3.22. The verilog module looks as follows:

```
module remainder(input wire [2:0] D, input wire [1:0] C, output wire [1:0] R);

  //Taking the mod of D with C will give us the remainder of dividing D by C

  assign R = D % C;

endmodule
```

Exercise 3.29

The 2-bit combinational divider from Exercise 3.23 can be implemented in verilog as follows:

```
module divider(input wire A, input wire B, input wire C, input wire D,
       output wire W, output wire X, output wire Y, output wire Z);

       //Registers to store the 2-bit input and outputs
       wire [1:0] in_1, in_2;
       reg [1:0] quotient, remainder;

       //Assign the appropriate bits to the operands
       assign in_1[1] = A;
       assign in_1[0] = B;
       assign in_2[1] = C;
       assign in_2[0] = D;

       //Combinational block
       always@(*)
              begin
                     quotient = in_1 / in_2;
                     remainder = in_1 % in_2;
              end

       //Assign the output bits the appropriate values
       assign W = quotient[1];
       assign X = quotient[0];
       assign Y = remainder[1];
       assign Z = remainder[0];

endmodule
```

Exercise 3.30

This module determines if the input value is divisible by 2, 3, and 6 by taking the modulo of the input with 2, 3, and 6 and sets the outputs to the appropriate value. Note that this works with any input value from 0000_2 to 1111_2.

```
module div_by_2_3_6(input wire A, input wire B, input wire C, input wire D,
       output reg By2, output reg By3, output reg By6);

       //A place to store the input
       wire [3:0] value;

       //Assign value with the input by taking each bit, shifting it left by
       //the appropriate amount and ORing them together.
       assign value = (A << 3) | (B << 2) | (C << 1) | D;

       //Combinational block
       always@(*)
              begin
                     //Check if divisible by 2
                     if((value % 2) == 0)
                            By2 = 1;
                     else
                            By2 = 0;

                     //Check if divisible by 3
                     if((value % 3) == 0)
                            By3 = 1;
                     else
                            By3 = 0;

                     //Check if divisible by 6
                     if((value % 6) == 0)
                            By6 = 1;
                     else
                            By6 = 0;
              end

endmodule
```

Exercise 3.31

This module works by adding each of the bits, A, B, C, and D together and storing the result into a 3-bit register, which is parsed into separate bits for the output.

```
module ones_count(input wire A, input wire B, input wire C, input wire D,
       output wire X, output wire Y, output wire Z);

       //A place to store the 3-bit output
       wire [2:0] out;

       //Add each bit together
       assign out = A + B + C + D;

       //Parse the output into seperate bits
       assign X = out[2];
       assign Y = out[1];
       assign Z = out[0];

endmodule
```

Exercise 3.32

This module adds the two 2-bit inputs together by shifting the high-order bit of the inputs to the left once and OR's it with the low-order bit then adds the result to the other input.

```
module adder(input wire A, input wire B, input wire C, input wire D,
       output wire X, output wire Y, output wire Z);

       //A place to store the 3-bit output
       wire [2:0] sum;

       //Shift the high-order bit left once, OR it with the low-order
       //bit for each input, and sum the results
       assign sum = ((A << 1) | B) + ((C << 1) | D);

       //Parse the output into seperate bits
       assign X = sum[2];
       assign Y = sum[1];
       assign Z = sum[0];

endmodule
```