Exercise 4.1

In this particular case using a K-map to simplify the problem will not be very useful since adjacent cells in a K-map vary by only one bit, and because this is a parity function every adjacent cell will be opposites.

Starting with the truth table for the function:

Input	Output
0000	1
0001	0
0010	0
0011	1
0100	0
0101	1
0110	1
0111	0
1000	0
1001	1
1010	1
1011	0
1100	1
1101	0
1110	0
1111	1

Using Boolean algebra the function can be expressed as follows:

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

= A’B’ (C’D’ + CD) + A’B (C’D + CD’) + AB’ (C’D + CD’) + AB (C’D’ + CD)

= (A’B’ + AB) (C’D’ + CD) + (A’B + AB’) (C’D + CD’)

= [(A’B’ + AB) (C’D’ + CD) + (A’B + AB’) (C’D + CD’)]””

= [[(A’B’ + AB) (C’D’ + CD)]’ [(A’B + AB’) (C’D + CD’)]’]’”

= [[(A’B’ + AB) (C’D’ + CD)]’ [(A’B + AB’) (C’D + CD’)]’]’”

= [[(A’B’ + AB)’ + (C’D’ + CD)’] [(A’B + AB’)’ + (C’D + CD’)’]]’”

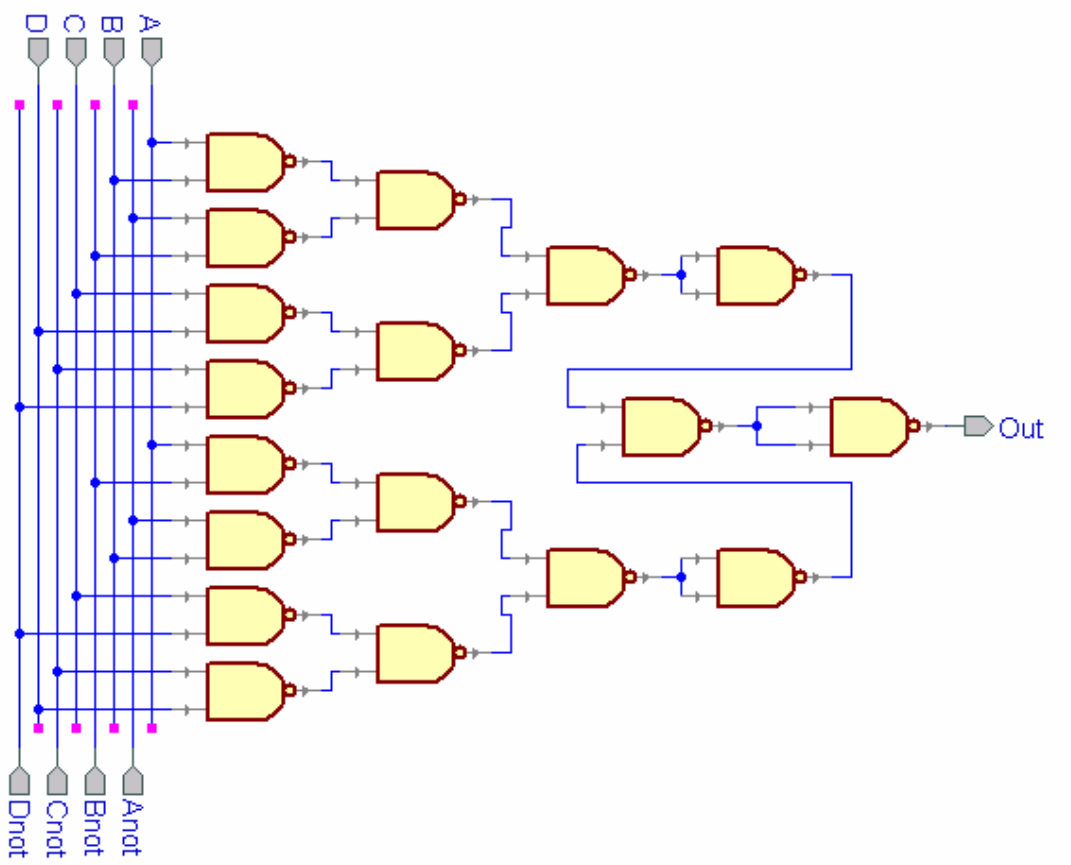
= [[(A’B’)’ (AB)’ + (C’D’)’ (CD)’] [(A’B)’ (AB’)’ + (C’D)’ (CD’)’]]’”

= [[(A’B’)’ (AB)’ + (C’D’)’ (CD)’]’ + [(A’B)’ (AB’)’ + (C’D)’ (CD’)’]’]”

= [[(A’B’)’ (AB)’]’ [(C’D’)’ (CD)’]’ + [(A’B)’ (AB’)’]’ [(C’D)’ (CD’)’]’]”

= [[(A’B’)’ (AB)’]’ [(C’D’)’ (CD)’]’’ [(A’B)’ (AB’)’]’ [(C’D)’ (CD’)’]’’]’

Assuming that both inputs and their complements are available, the diagram below shows how many NAND gates are required:



Since there are four to a package, this will take 3 packages to implement. Going back to the Boolean simplification, and then simplifying to XOR gates gives the following result:

f = (A’B’ + AB) (C’D’ + CD) + (A’B + AB’) (C’D + CD’)

= (A ⊕ B)’ (C ⊕ D)’ + (A ⊕ B) (C ⊕ D)

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

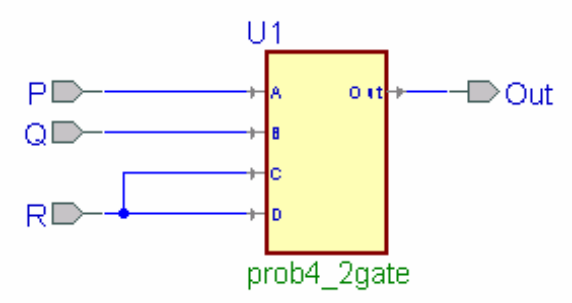
= (A’ ⊕ B) ⊕ (C ⊕ D)

Which can be implemented using a single package of XOR gates.

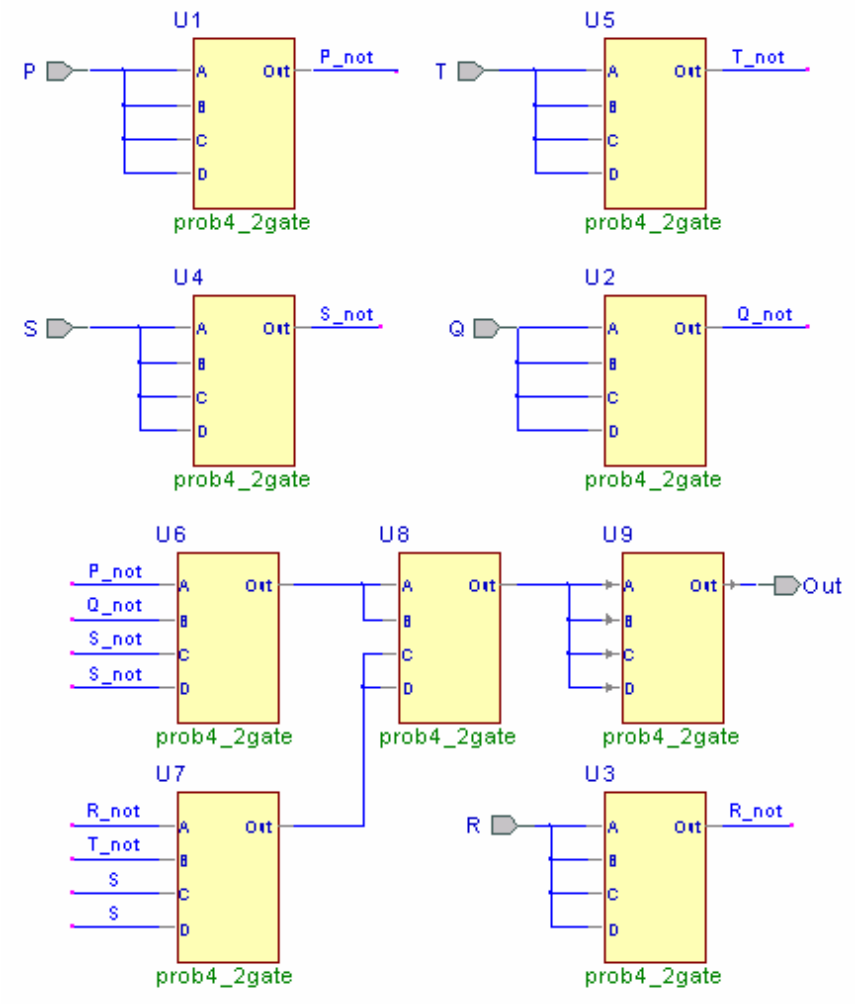
Exercise 4.2

In each of the parts below the prob4_2 blocks can be assumed to implement the function Z = (AB + CD)'.

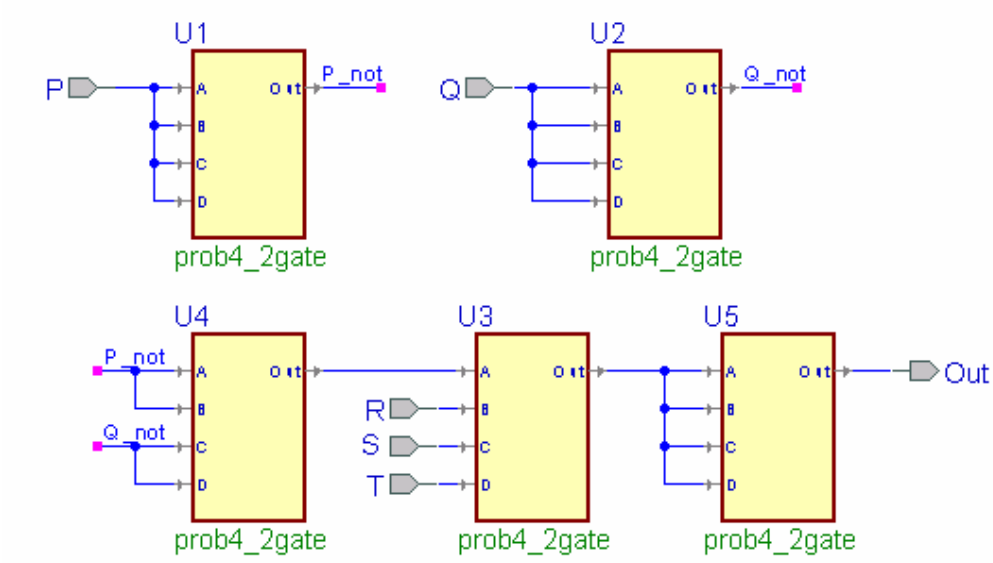
(a)



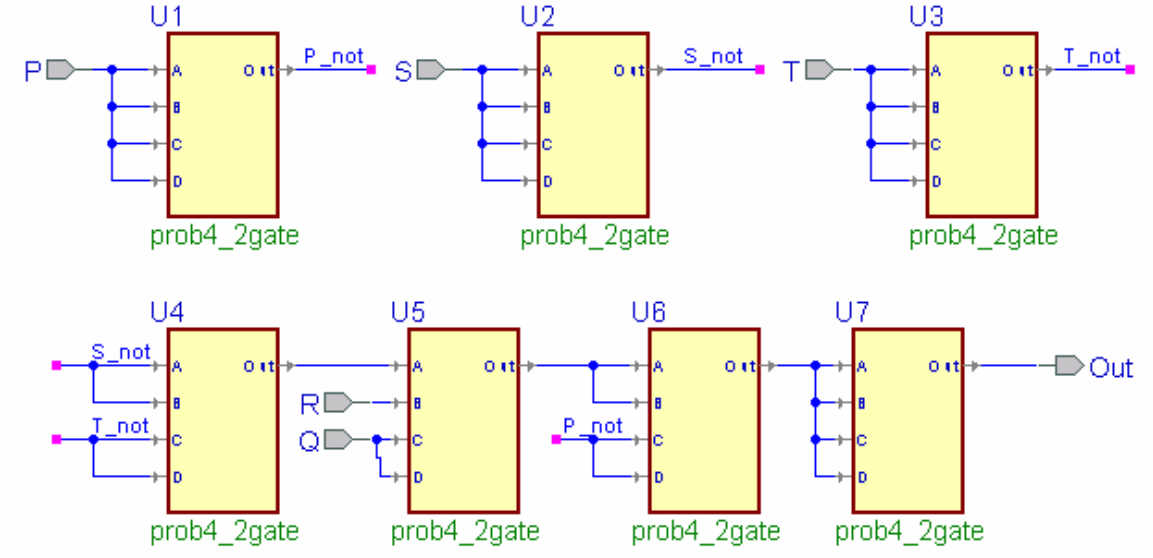
(b)



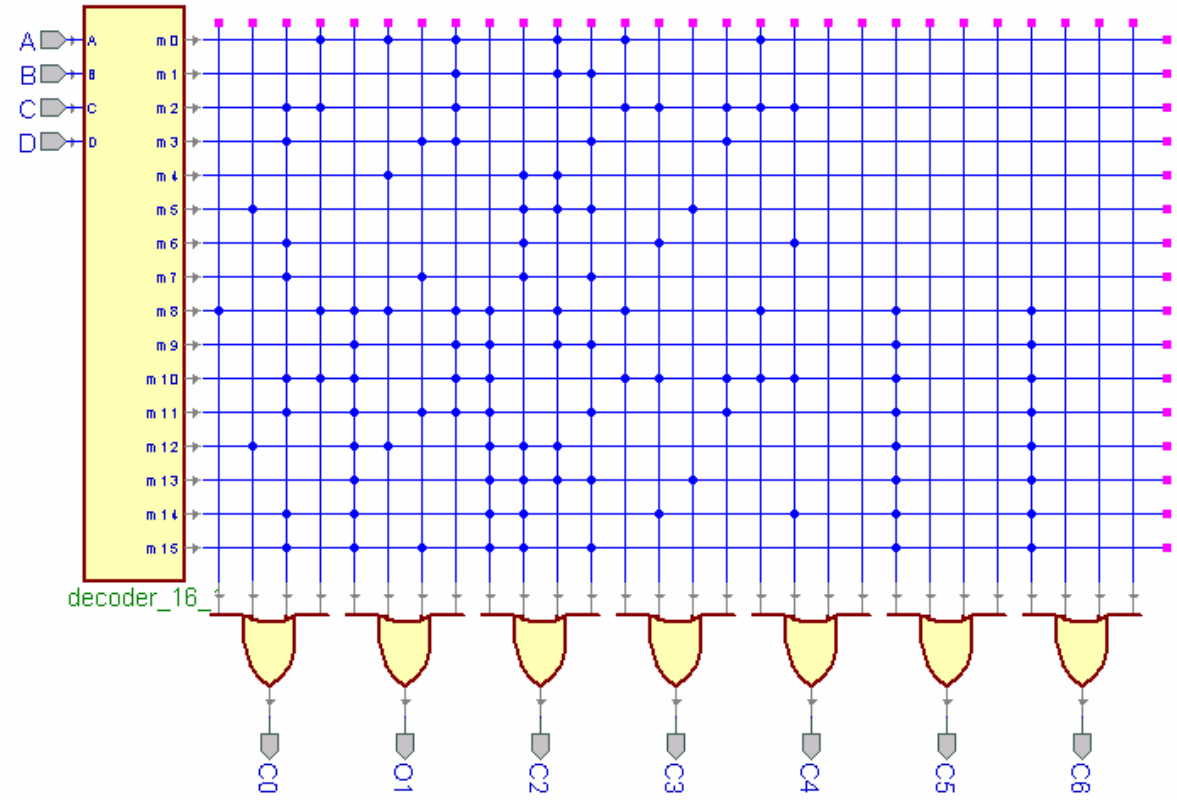
(c)


U1
P
P_not
prob4_2gate
U2
Q
Q_not
prob4_2gate
U4
P_not
Q_not
prob4_2gate
U3
R
S
T
prob4_2gate
U5
Out
prob4_2gate

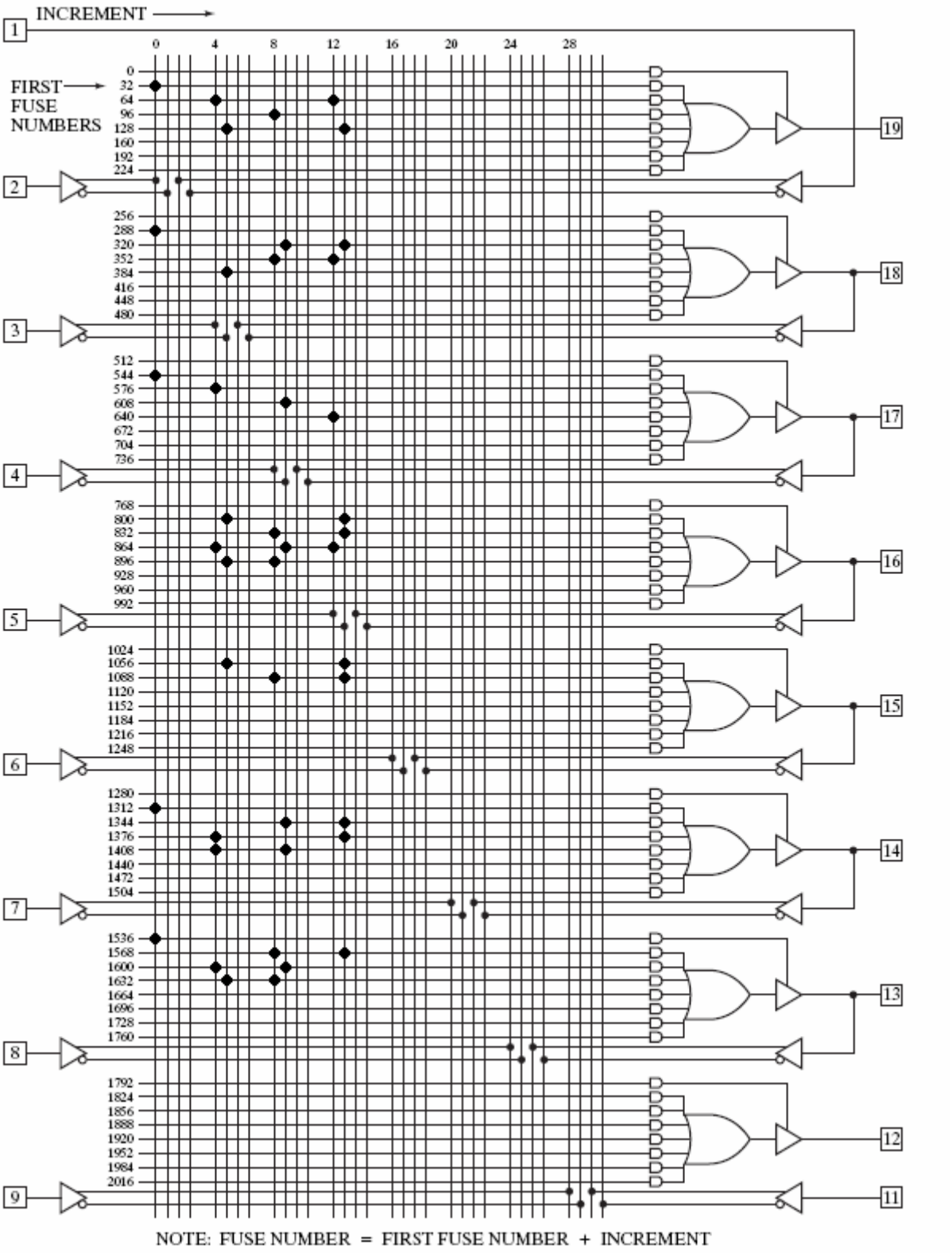
(d)


U1
P
P_not
prob4_2gate
U2
S
S_not
prob4_2gate
U3
T
T_not
prob4_2gate
U4
S_not
T_not
prob4_2gate
U5
R
Q
prob4_2gate
U6
P_not
prob4_2gate
U7
Out
prob4_2gate

Exercise 4.3


A
B
C
D
A
B
C
D
m0
m1
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15
decoder_16_
O0
O1
O2
O3
O4
O5
O6

Exercise 4.4



NOTE: FUSE NUMBER = FIRST FUSE NUMBER + INCREMENT

The table below matches inputs and outputs to the corresponding pin numbers:

PIN	Signal
2	A
3	B
4	C
5	D
13	C_6
14	C_5
15	C_4
16	C_3
17	C_2
18	C_1
19	C_0

Exercise 4.5

As is shown with the multilevel functions given in the chapter, it is incredibly difficult to even factor the equations into a multilevel functions that have a total of 8 outputs from the PLA, and only two outputs that use 4 AND gates. Thus, you cannot fit the solution entirely in a P14H8 PAL.

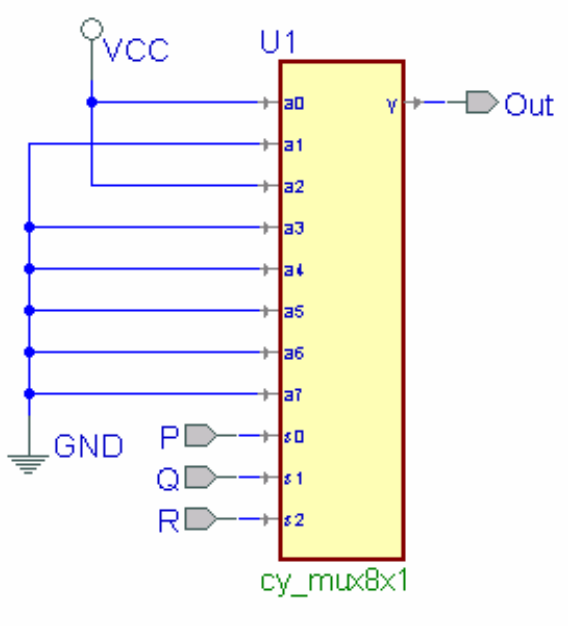
Exercise 4.6

The main difference between solutions 4.3 and 4.4 is that in 4.3 you have fully programmable OR plane, whereas in 4.4 you have a fully programmable AND plan. The advantage of the OR plane is that you can utilize common product terms for each function; however, this leads to OR gates with $2^{\text{\# of inputs}}$ fan in. In the case of the PAL, the advantage is being able to have smaller fan-in OR gates, and only 2*(# of inputs) fan-in on the AND gates.

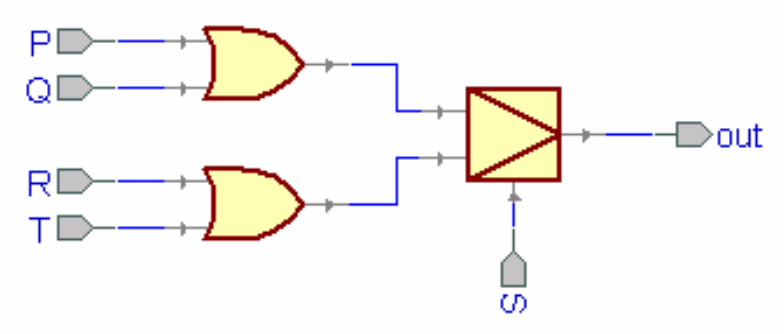
The advantage of the PLA implementation is that both of these strategies can be combined in order to reduce the number of AND gates and the number of OR gates. However, a PLA is generally slower because the programmable planes tend to slow the circuit down a bit.

Exercise 4.7

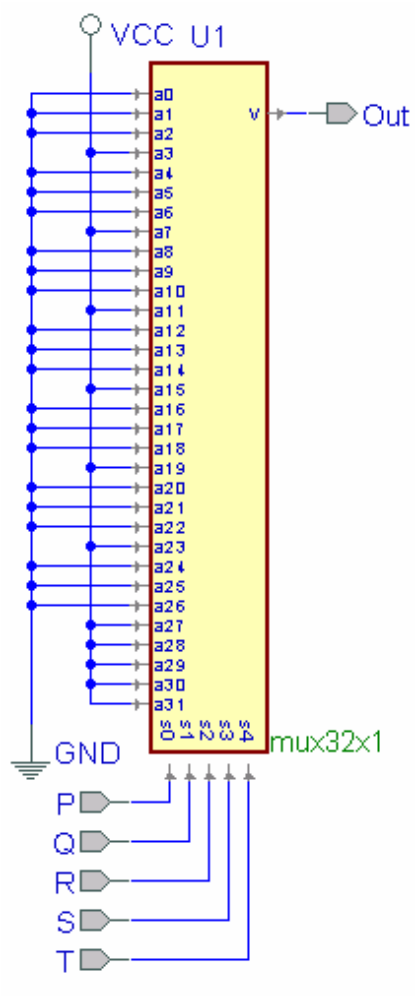
(a)



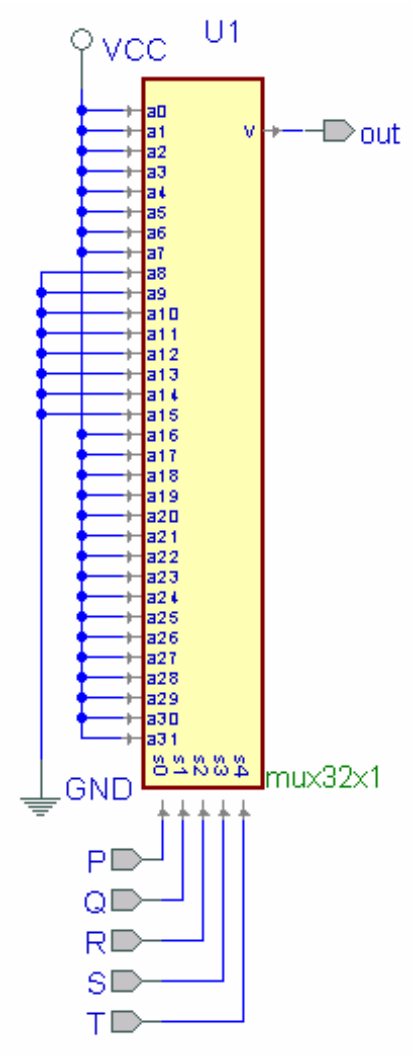
(b)



(c)


VCC
U1
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
a10
a11
a12
a13
a14
a15
a16
a17
a18
a19
a20
a21
a22
a23
a24
a25
a26
a27
a28
a29
a30
a31
v
Out
S0
S1
S2
S3
S4
mux32x1
GND
P
Q
R
S
T

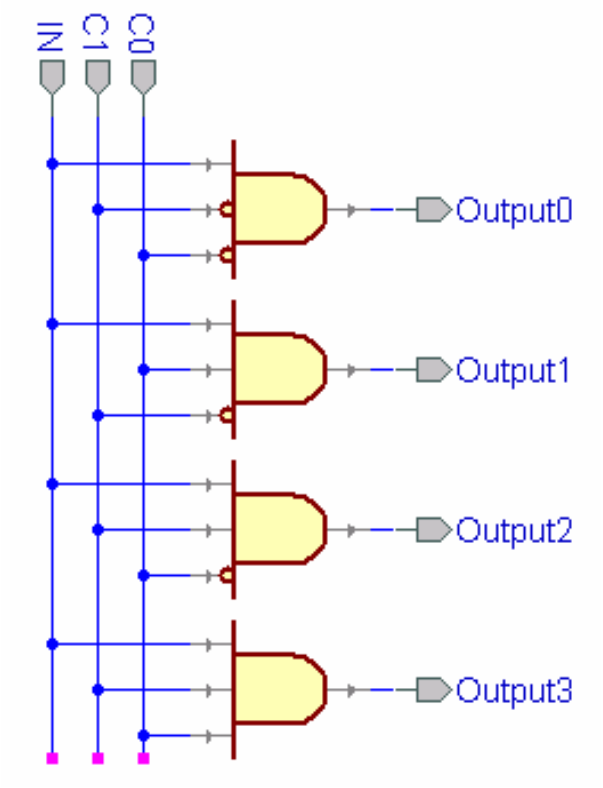
(d)


VCC
U1
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
a10
a11
a12
a13
a14
a15
a16
a17
a18
a19
a20
a21
a22
a23
a24
a25
a26
a27
a28
a29
a30
a31
v
out
s0
s1
s2
s3
s4
GND
mux32x1
P
Q
R
S
T

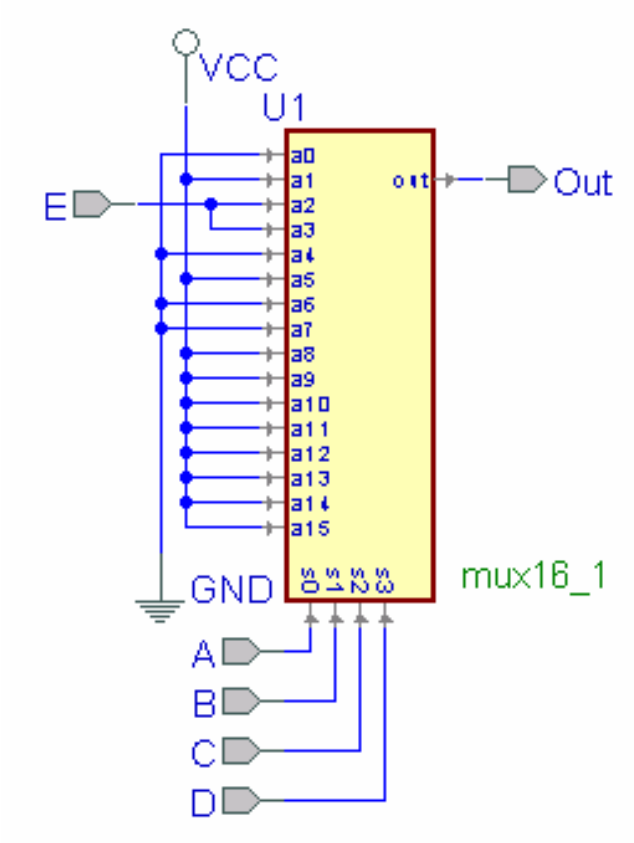
Exercise 4.8

(a) A multiplexer with n control bits takes 2^n inputs, and based on the binary value of the control bits outputs, suppose this number is i, the i-th input is connected to the output bit. A demultiplexer takes a single input and has 2^n outputs. Based on the binary value of the n control bits, it will pass the input value into the i-th output bit. A decoder is the same as a demultiplexer except that in general the input bit is viewed more as an enable signal in this case.

(b) The function below implements a 2:4 demultiplexer.

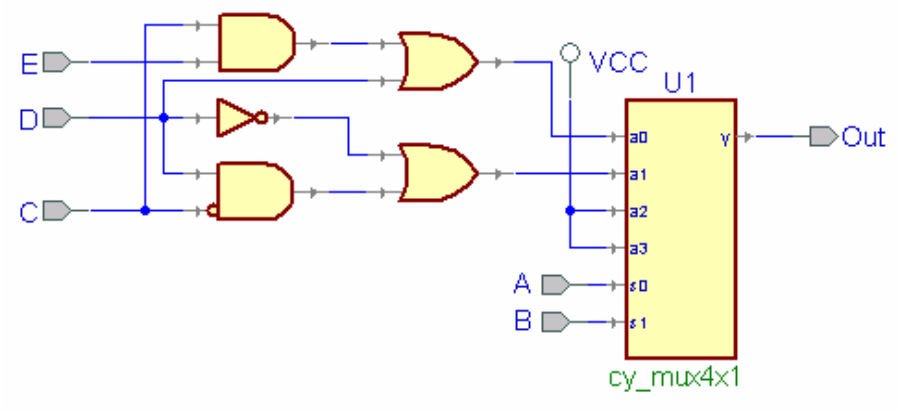


Exercise 4.9

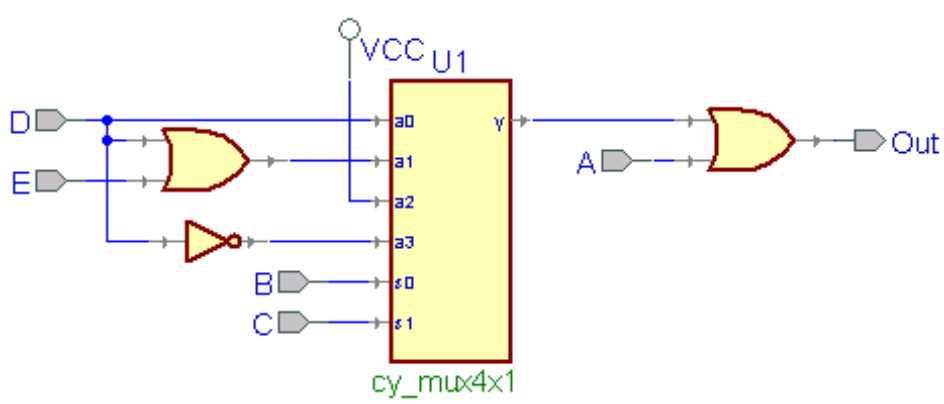

VCC
U1
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
a10
a11
a12
a13
a14
a15
out
Out
E
GND
s0
s1
s2
s3
mux16_1
A
B
C
D

Exercise 4.10

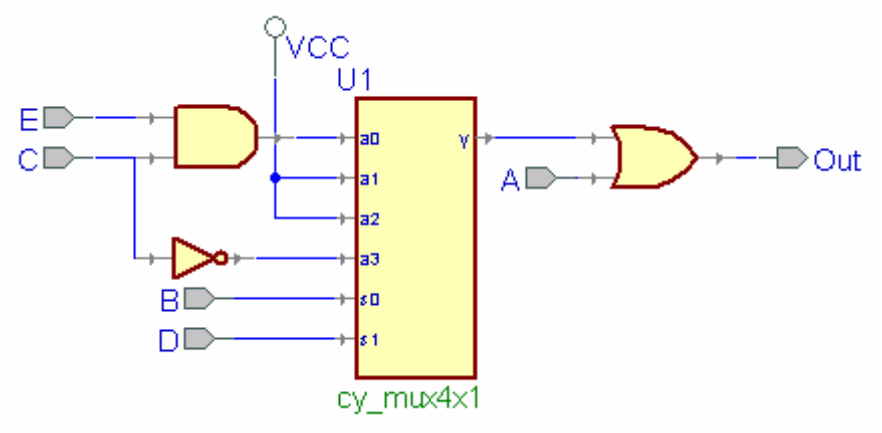
(a)



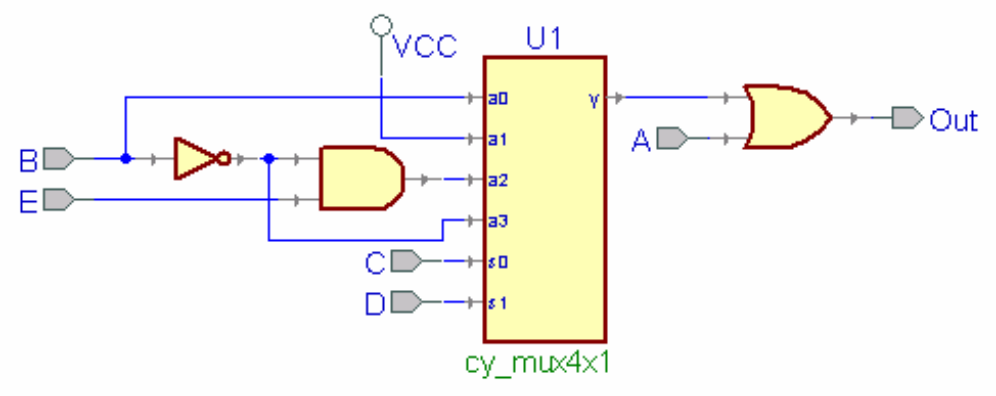
(b)



(c)



(d)



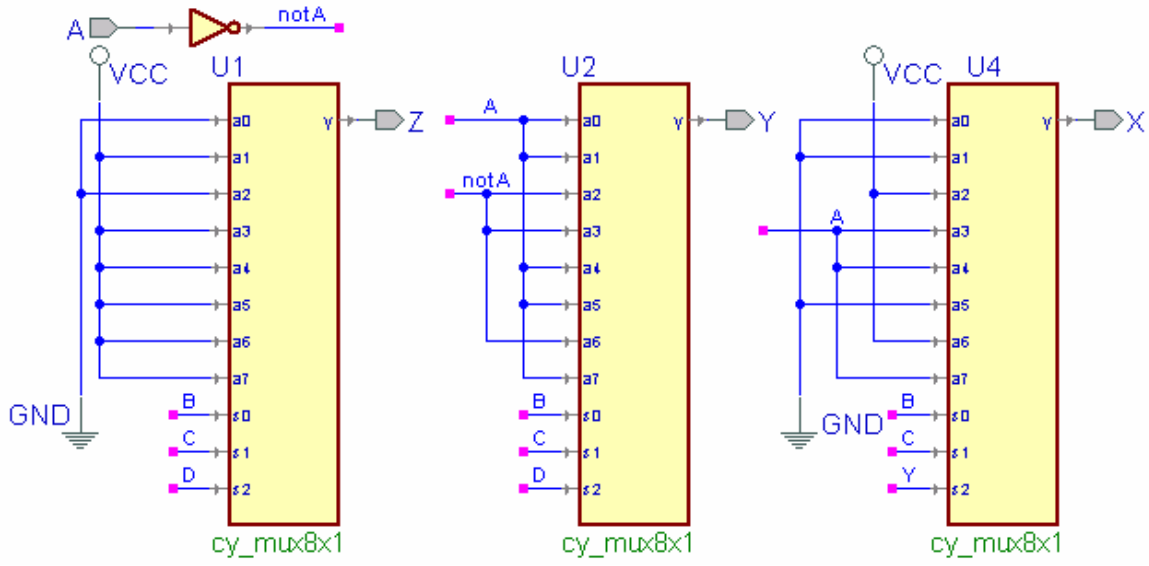
Exercise 4.11

AB	CD	XYZ
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

X = A’BCD + AB’CD’ + AB’CD + ABC’D + ABCD’ + ABCD

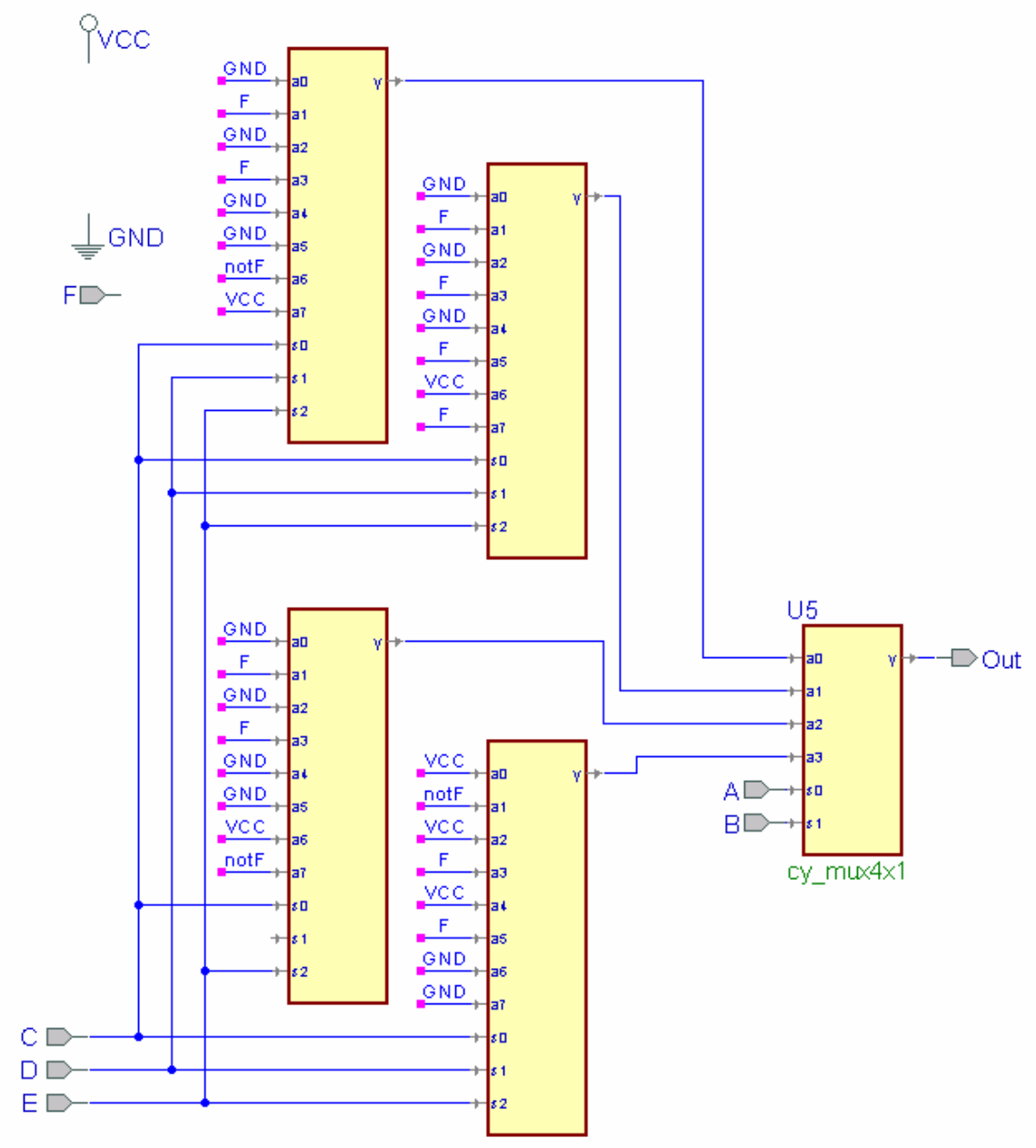
Y = A’B’CD’ + A’B’CD + A’BC’D + A’BCD’ + AB’C’D’ + AB’C’D + ABC’D’ + ABCD

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



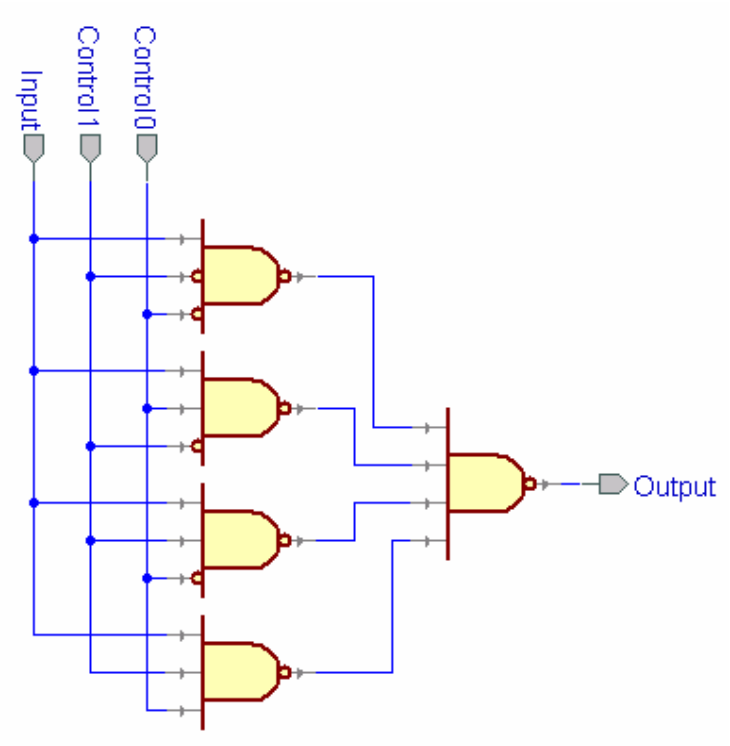
Exercise 4.12

Assuming F' is also available:



(a) To implement this function, this would take 5 packages, 4 of which would be 8:1 multiplexers and the last one would be a 4:1 multiplexer.

(b) The component below implements a 4:1 multiplexer. Since each control signal only needs to be inverted once for the entire chip, only one package of inverters is needed. A 32:1 multiplexer can be implemented using 10 of this component, and 2:1 multiplexer (where the 2:1 multiplexer uses three 2-input NAND gates).

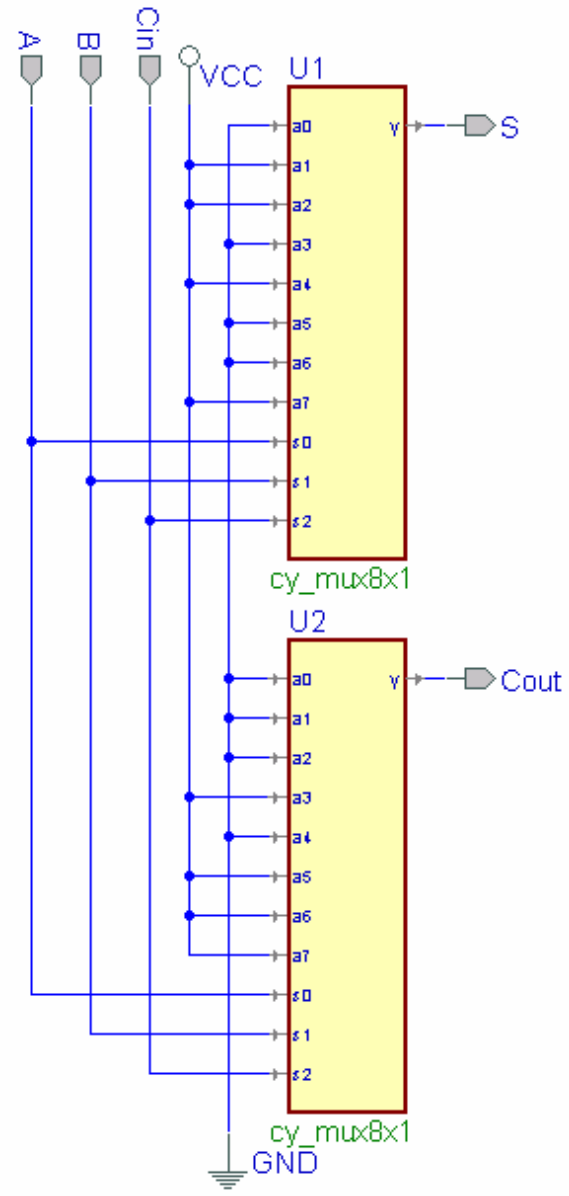


The table below demonstrates how many of each type of gate is required, and how many packages for each.

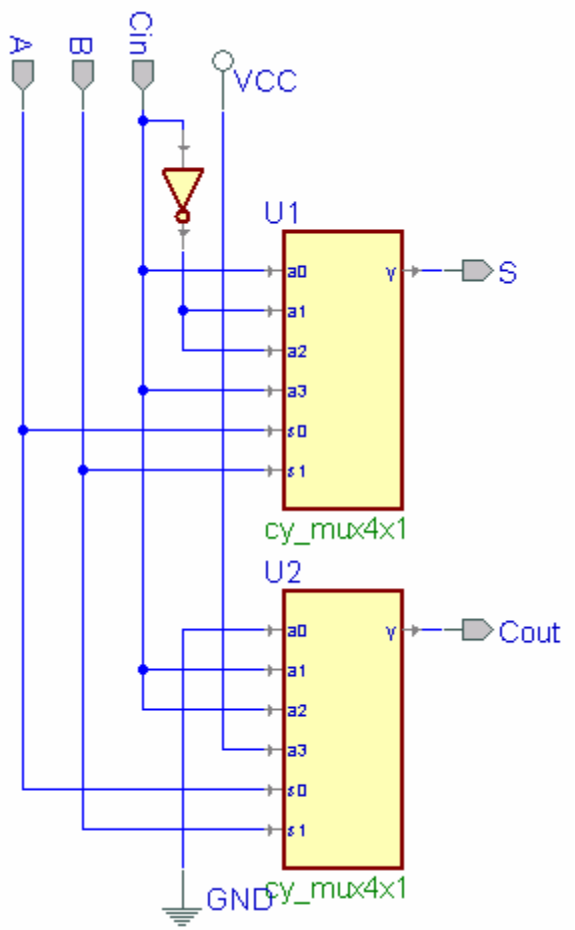
Gate Type	Number of Gates	Number of Packages
Inverters	6	1
2-input NAND	3	1
3-input NAND	40	14
4-input NAND	10	5
TOTAL	**59**	**21**

Exercise 4.13

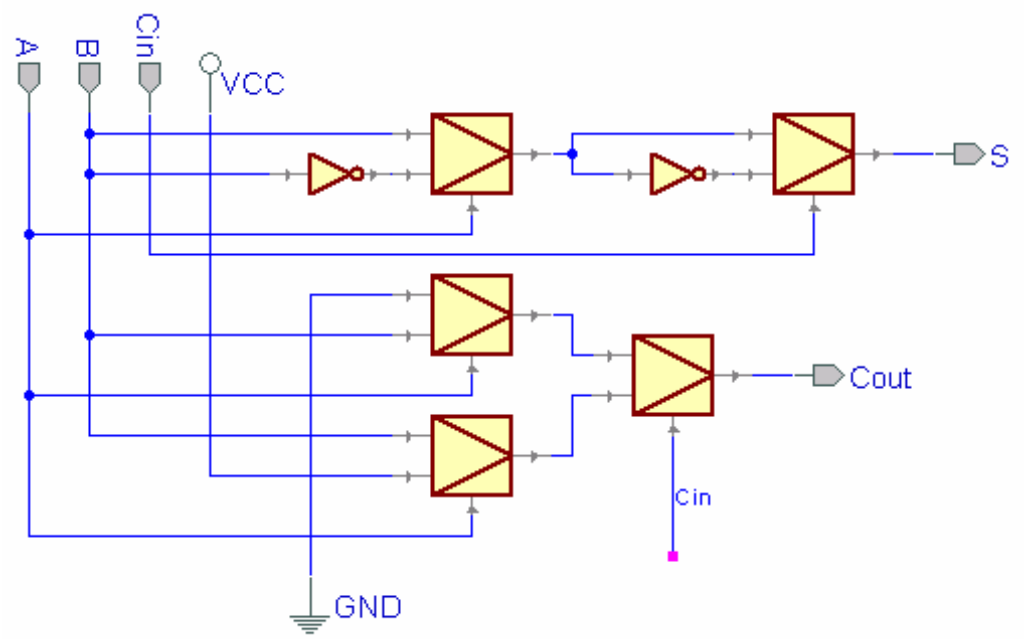
(a)



(b)


A
B
Cin
VCC
U1
a0
a1
a2
a3
s0
s1
y
S
cy_mux4x1
U2
a0
a1
a2
a3
s0
s1
y
Cout
cy_mux4x1
GND

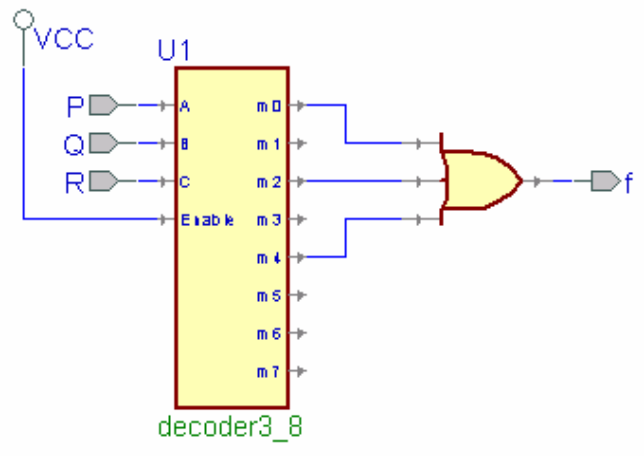
(c) Only five 2:1 multiplexers are needed to implement the function:



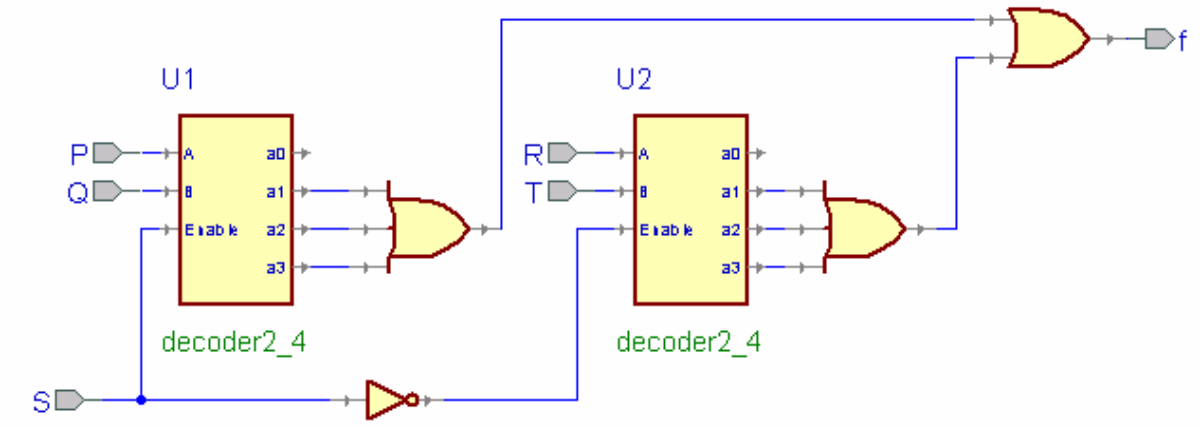
(d) Yes it is possible as shown in part (c).

Exercise 4.14

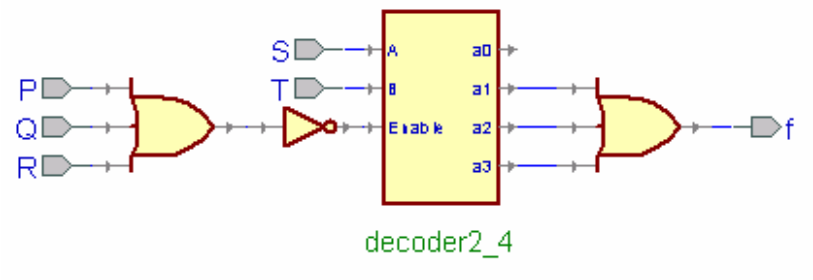
(a)



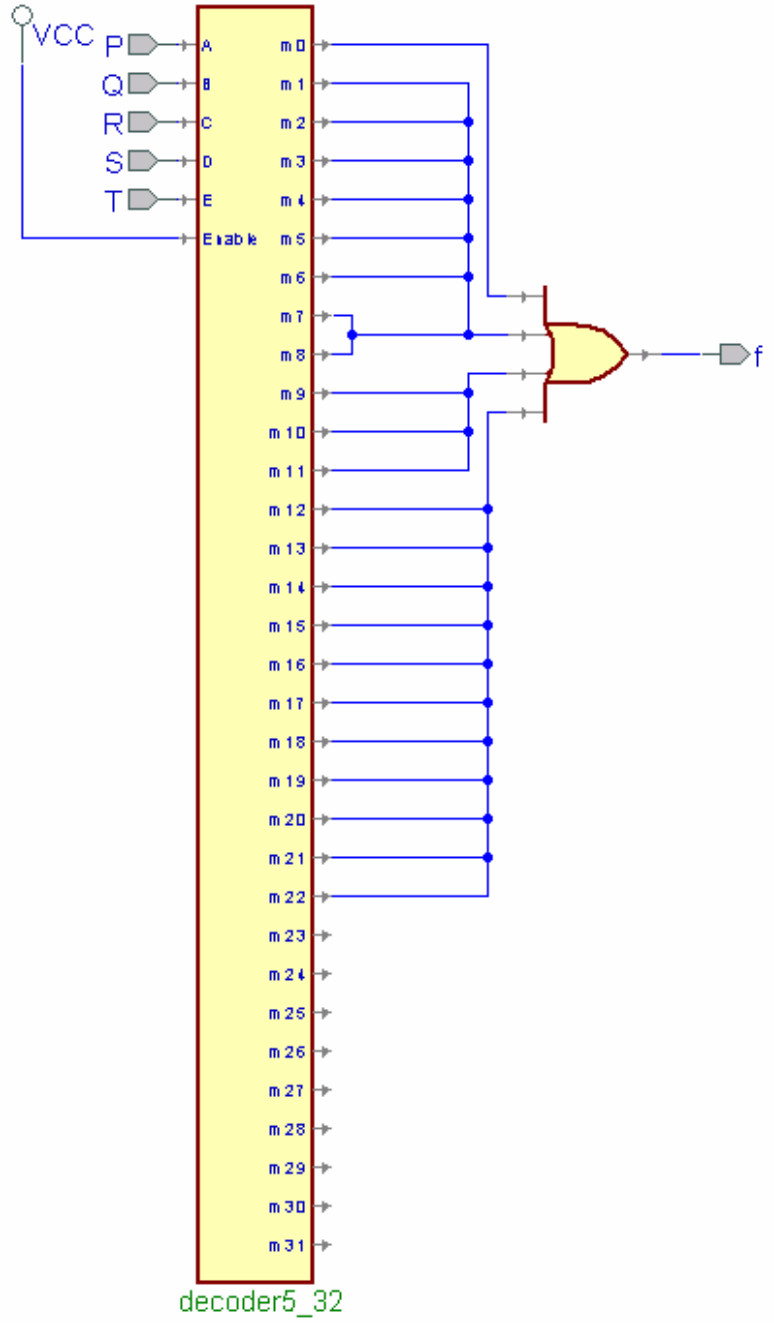
(b)



(c)

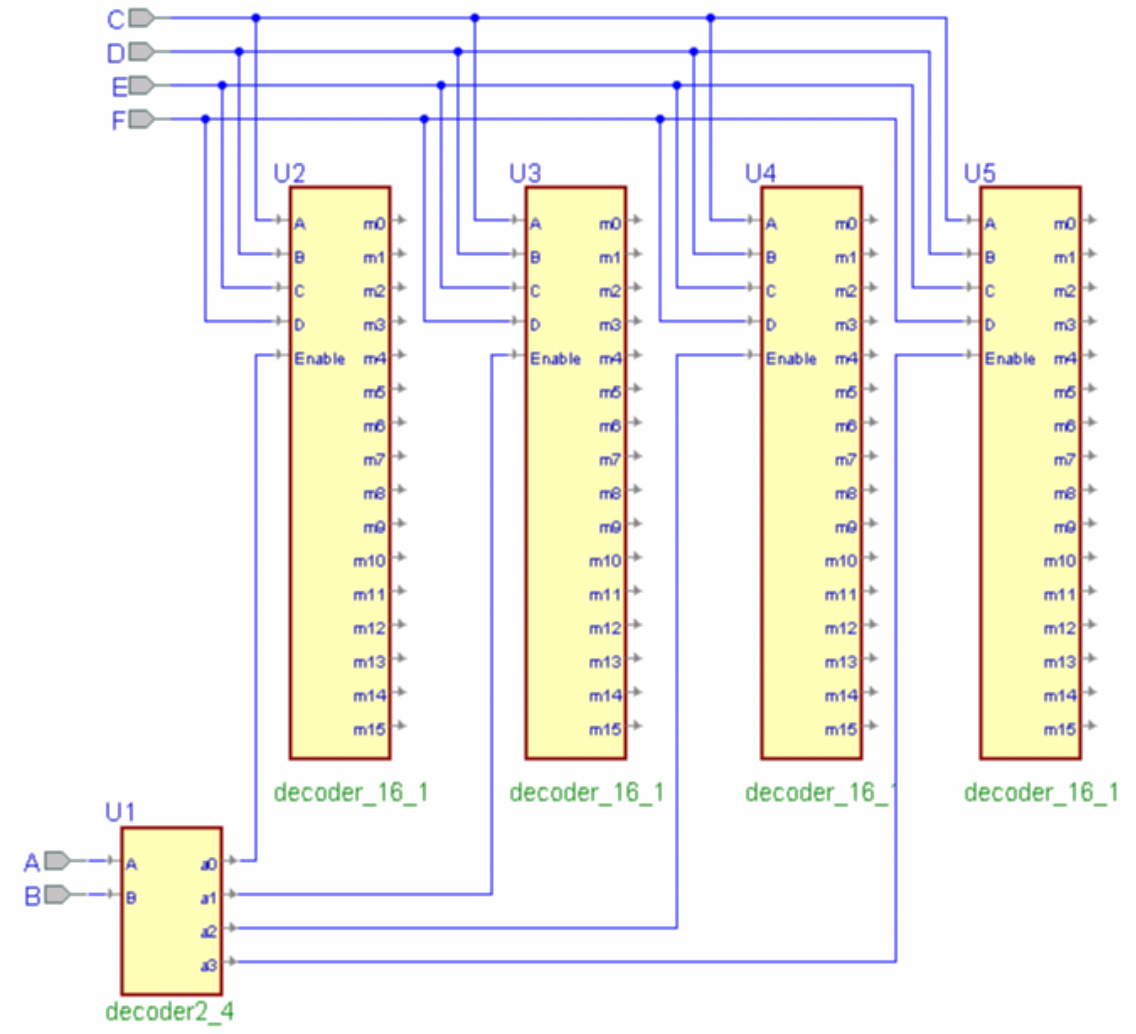


(d) Note: in order to simplify the diagram a single or gate is used. In this case the dots connecting output wires should be interpreted as separate connections to the OR gate since the drawing tool used does not have an or gate with sufficient fan-in. When a 23-input OR gate is not available, using a hierarchy of smaller OR gates will accomplish the same thing.

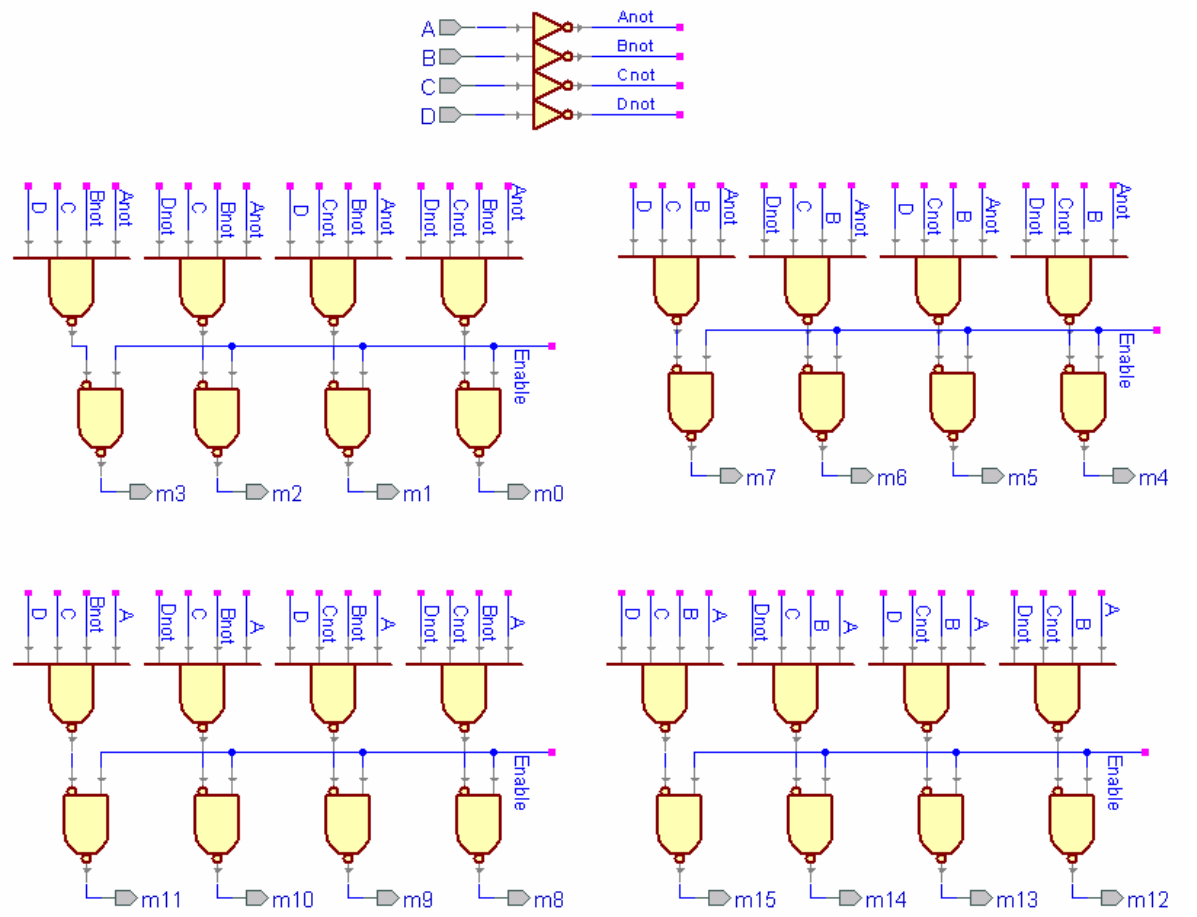


Exercise 4.15

This implementation uses a 2:4 decoder to enable a set of 4:16 decoders. Thus the first 16 outputs will be enabled when A'B' is asserted, the second 16 outputs when A'B and so on.

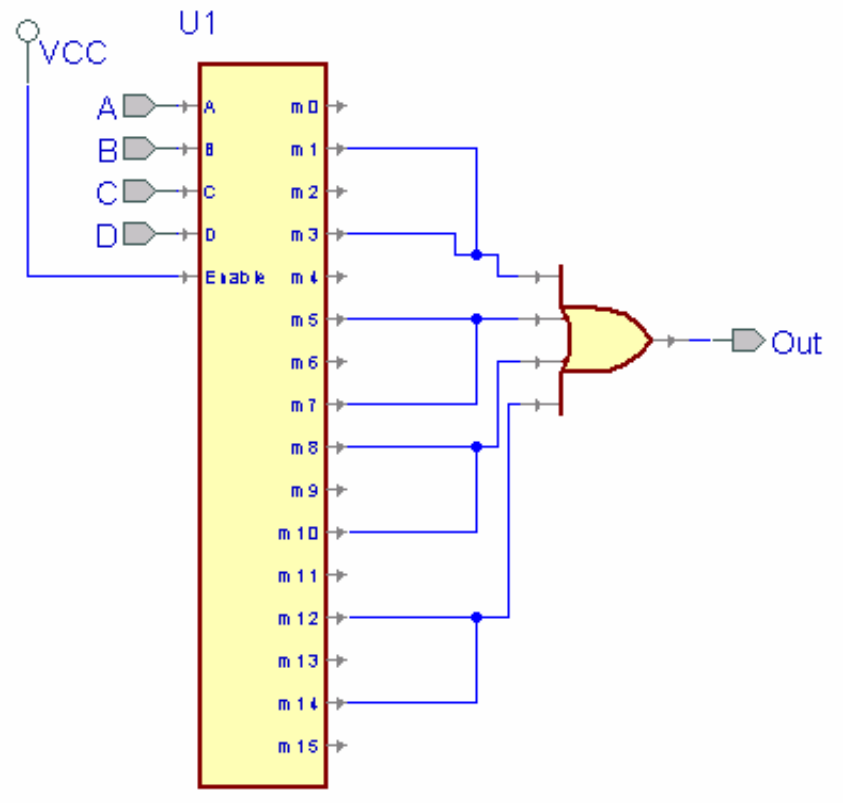


Exercise 4.16



Exercise 4.17

(a) Note: Due to the limitations of the software being used, the dots connecting wires should be considered separate inputs to the OR gate. In this case the OR-gate is an 8-input OR gate.



(b) Using the naïve implementation in discrete gates, this function takes four 3-input AND gates and one 4-input OR gate. Which compared to the single package for the decoder and single package for the 8-input OR gate seems pretty bad in terms of number of gates and wires required. However, by simplifying the function first:

f = A'B'D + A'BD + AC'D' + ACD'

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

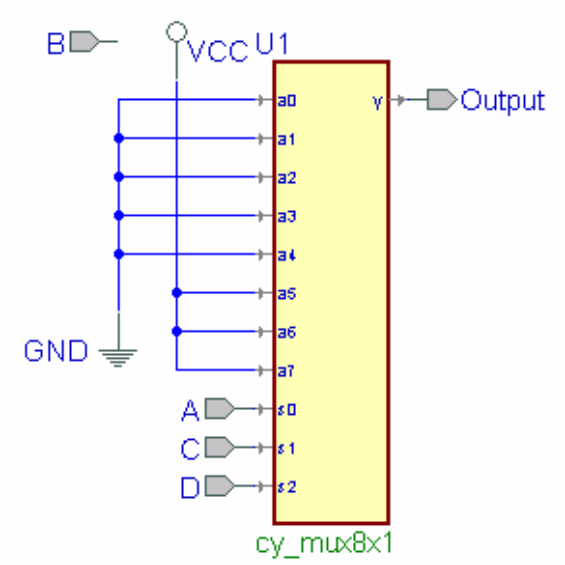
= A'D + AD'

= A ⊕ D

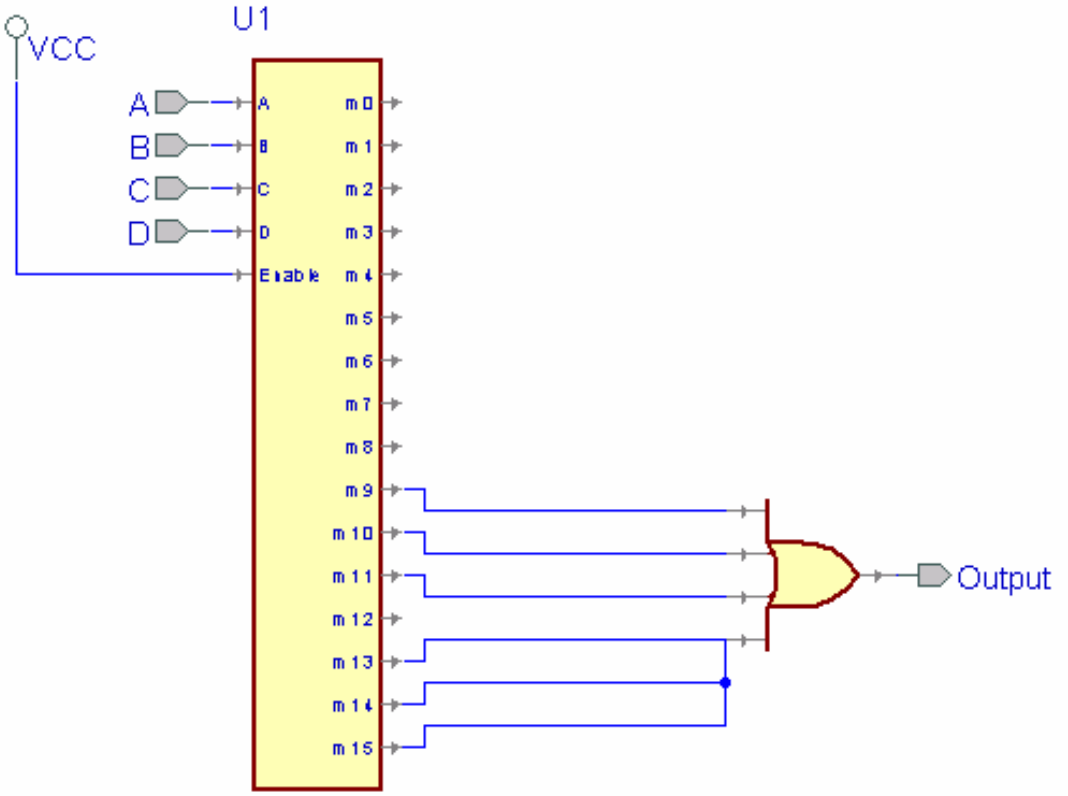
This can be implemented using a single discrete gate which has significantly less fan-in than the 8-input OR gate, and this is accomplished with only one level of logic instead of two.

Exercise 4.18

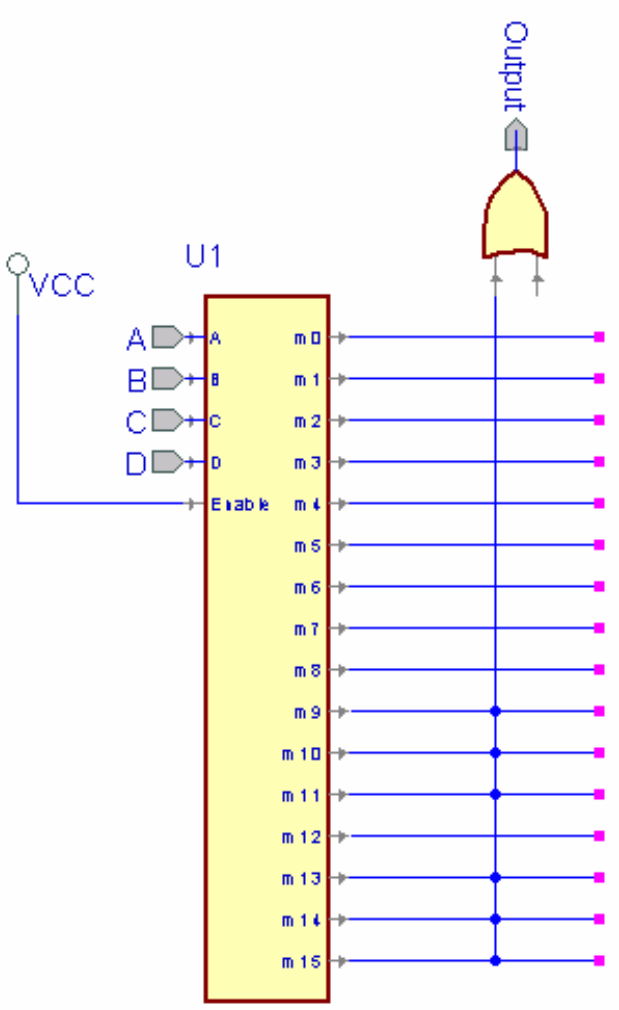
(a) By simplifying the function, it is realized that AB'C is already covered by AC, so B is not needed as an input to the multiplexer.



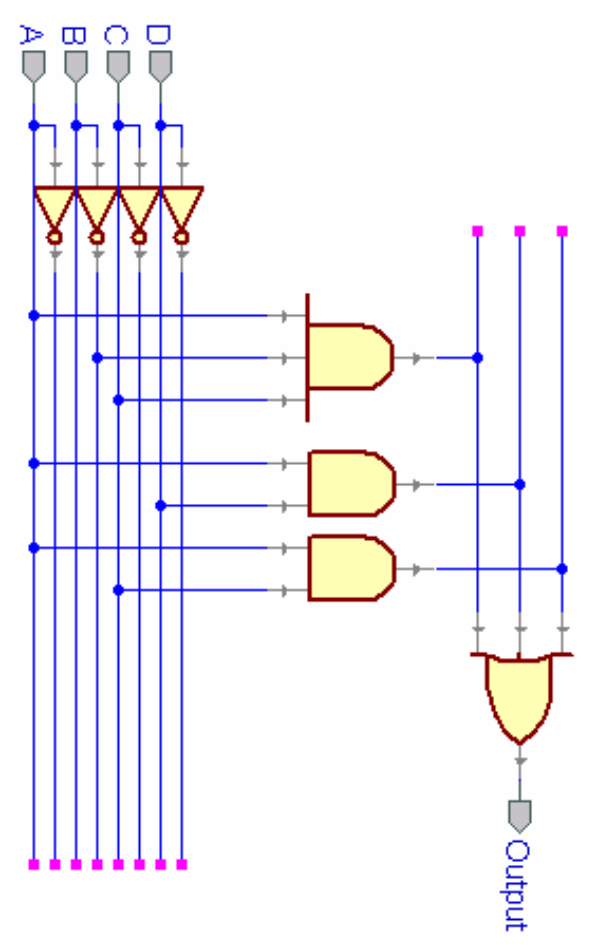
(b)



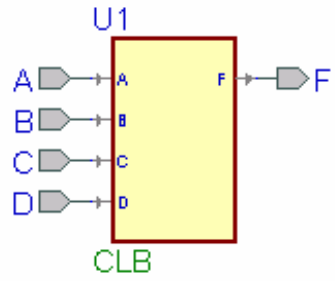
(c)


Output
U1
VCC
A
B
C
D
Enable
A
B
C
D
m0
m1
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15

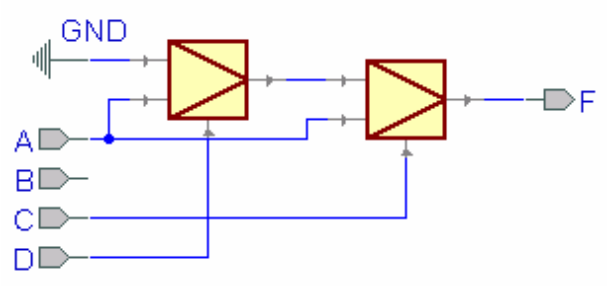
(d)



(e) Program the CLB such that the output function F corresponds to the function of four variables passed in to the block.

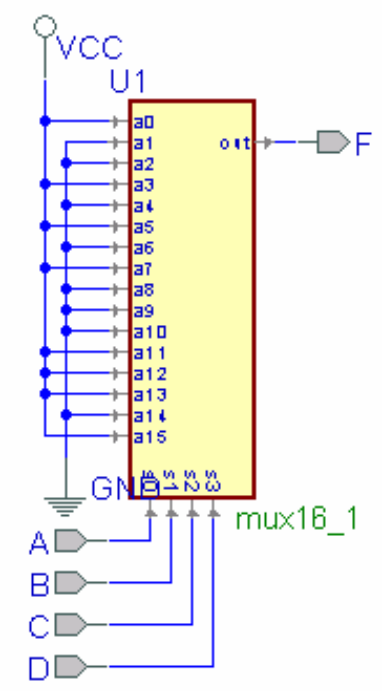


(f)

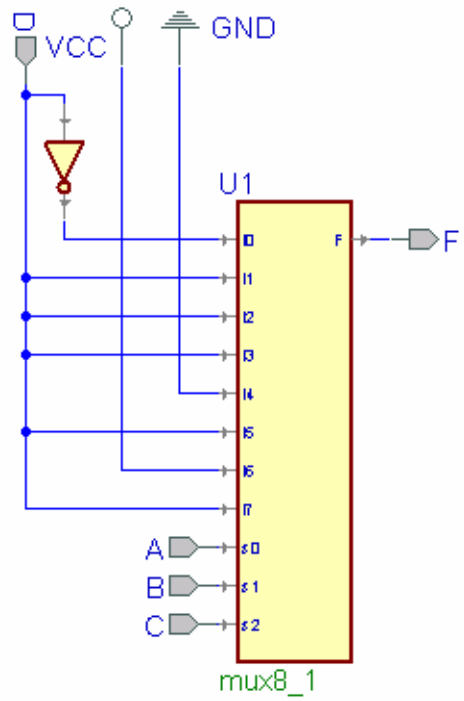


Exercise 4.19

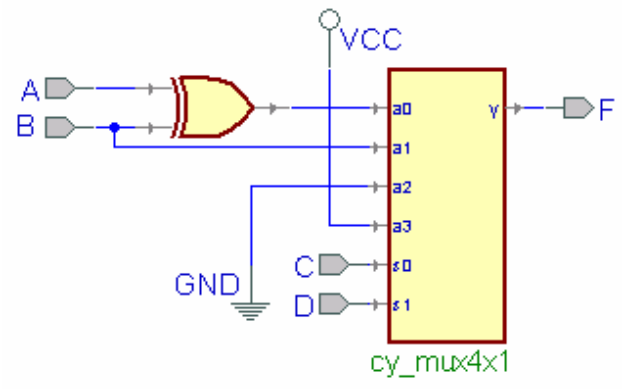
(a)



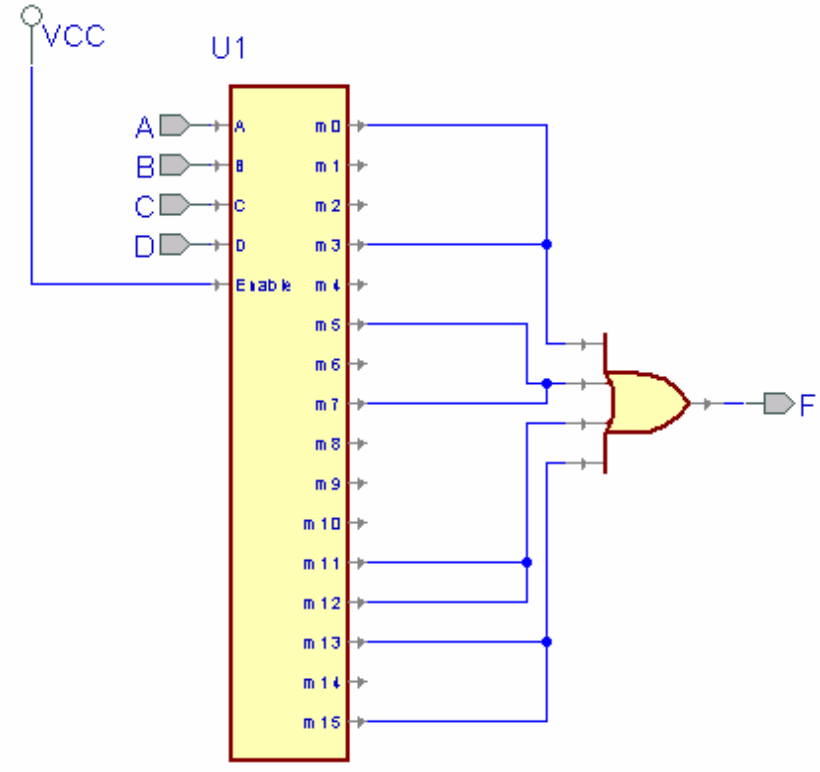
(b)



(c) Note: The Hint in the book appears to be incorrect for this problem, putting A and B on the control inputs requires more than just a single OR gate to implement the input functions. Putting C and D on the inputs was found to only require a single XOR gate to implement the function.



(d)



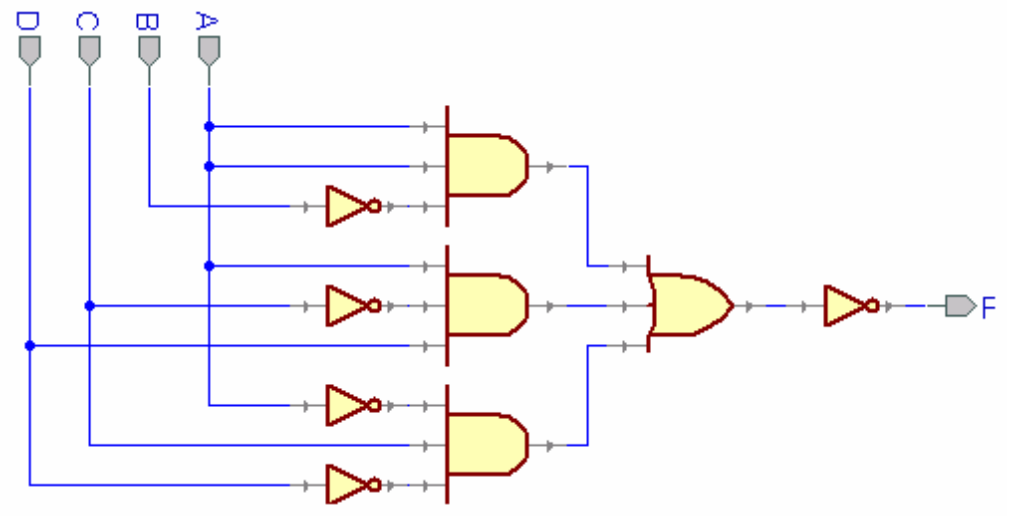
Exercise 4.20

(a) f(A, B, C, D) = $\prod$ M(3, 4, 5, 6, 7, 13) + $\prod$ D(0, 2, 9, 10)

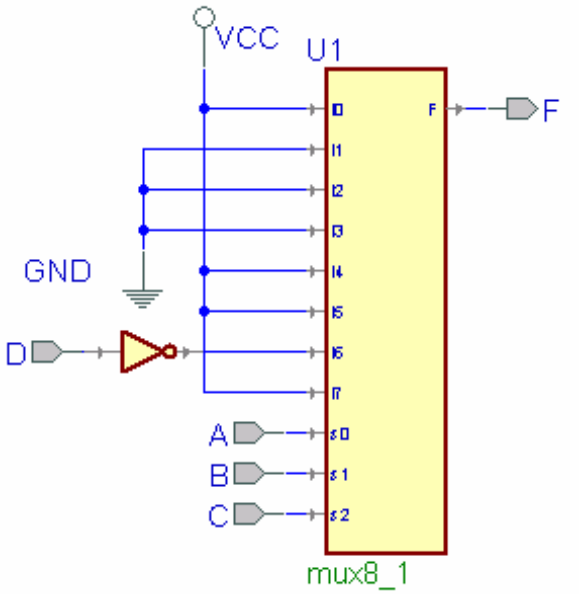
(b) f(A, B, C, D) = $\sum$ m(1, 8, 11, 12, 14, 15) + $\sum$ d(0, 2, 9, 10)

(c) f(A, B, C, D) = AC + B'C + AD'

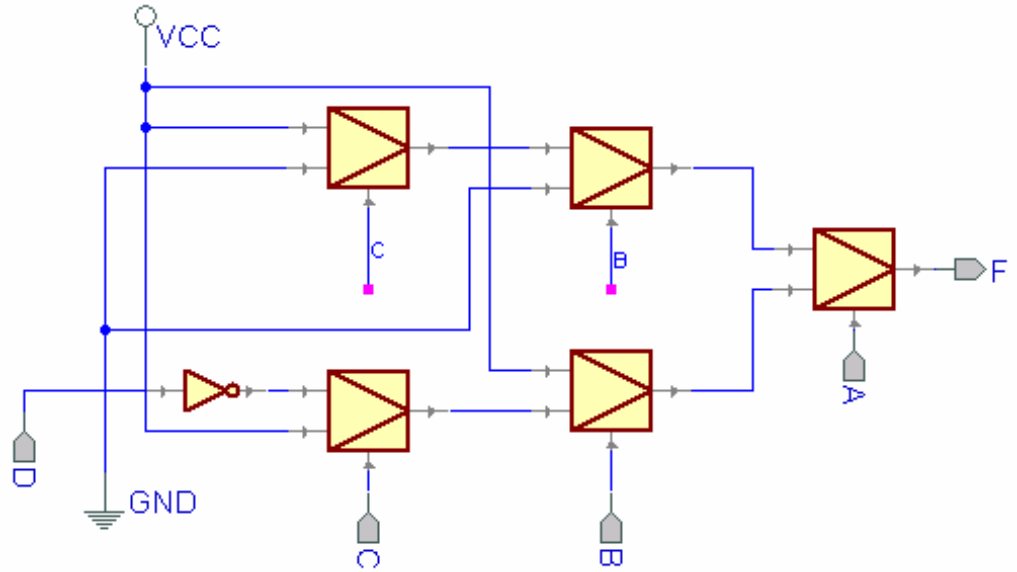
(d) The solution below assumes that each input and its complement is available. Utilizing DeMorgan's laws, the AND-OR-INVERT gate is the same as doing a product-of-sums solution, except each of the inputs into the AND gates must be inverted.



(e)

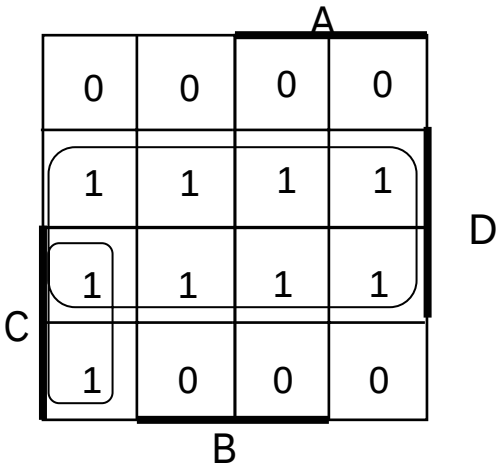


(f)

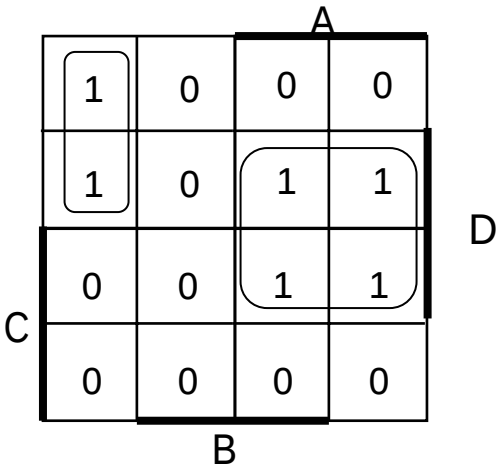


Exercise 4.21

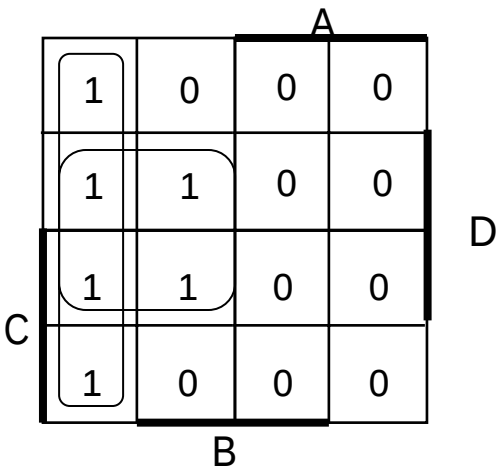
(a) The following K-maps show the minimum sum-of-products form for each of the 3 equations. This solution has 6 distinct product terms.



X = D + A'B'C



Y = AD + A'B'C'



$$Z = A'D + A'B'$$

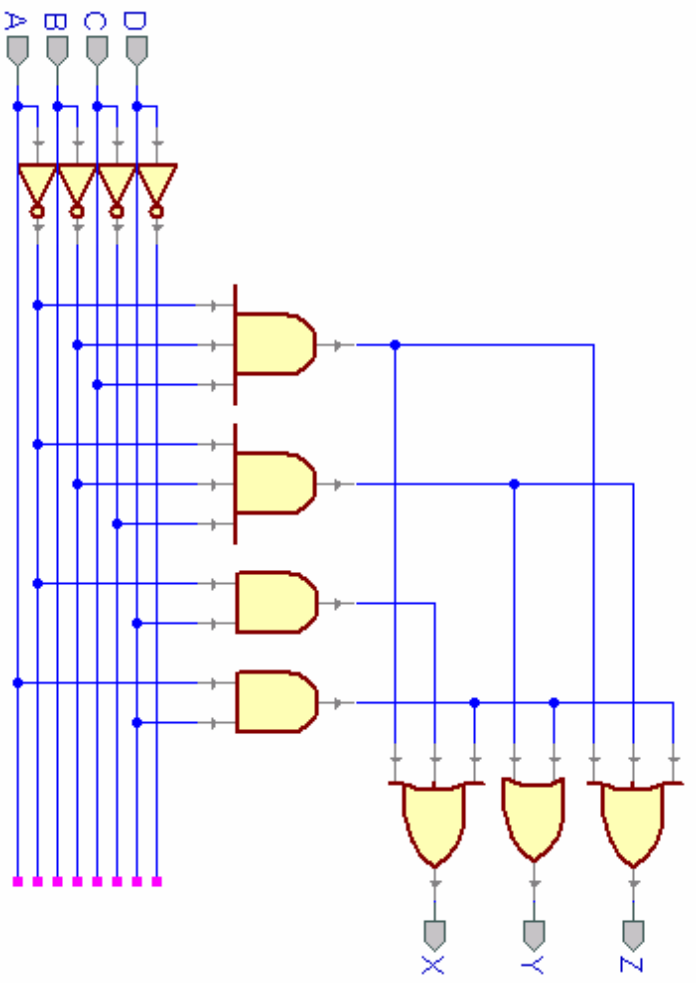
(b) The following solution only has 4 distinct product terms:

$$X = A'D + AD + A'B'C$$

$$Y = AD + A'B'C'$$

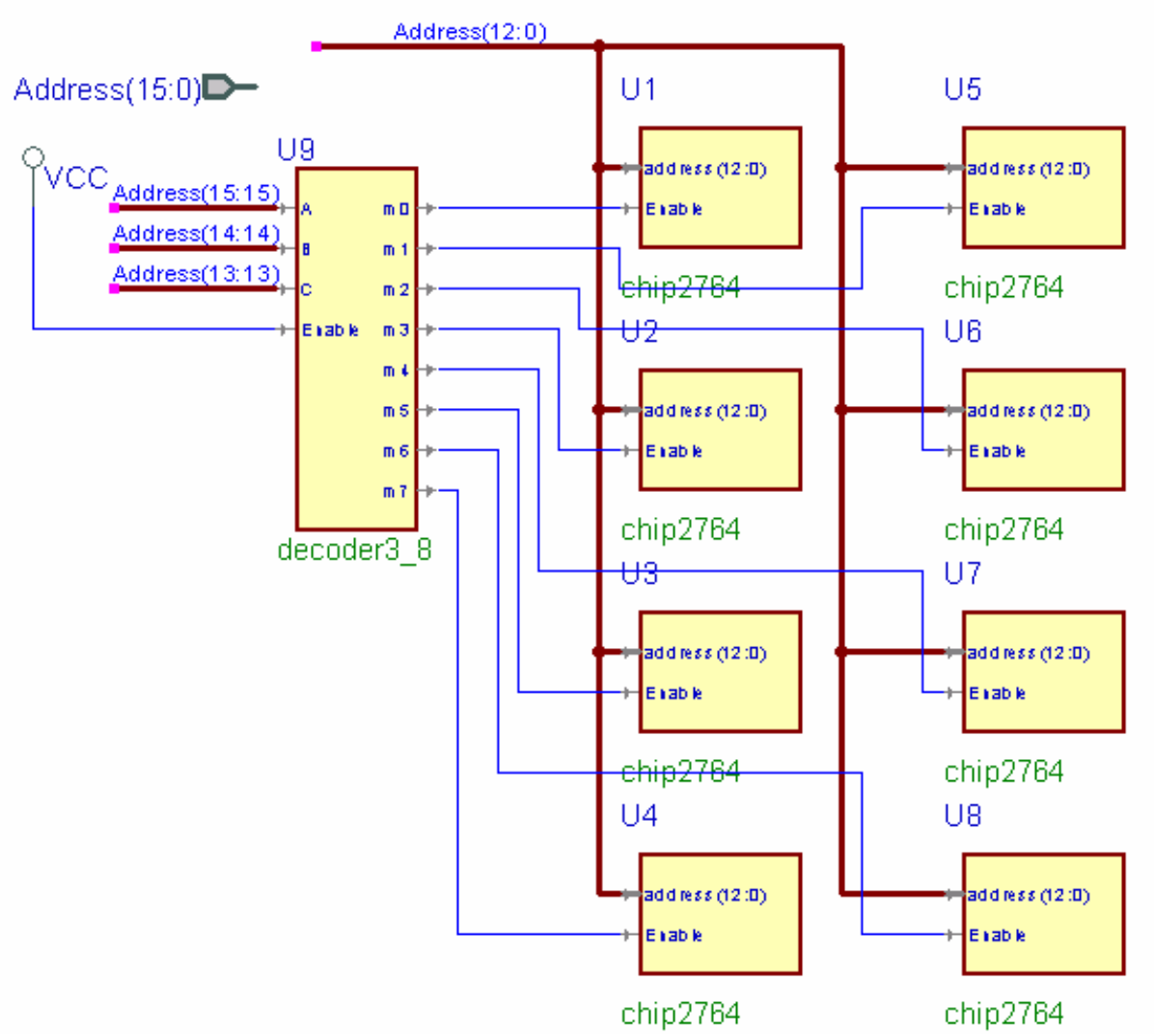
$$Z = A'D + A'B'C' + A'B'C$$

(c)

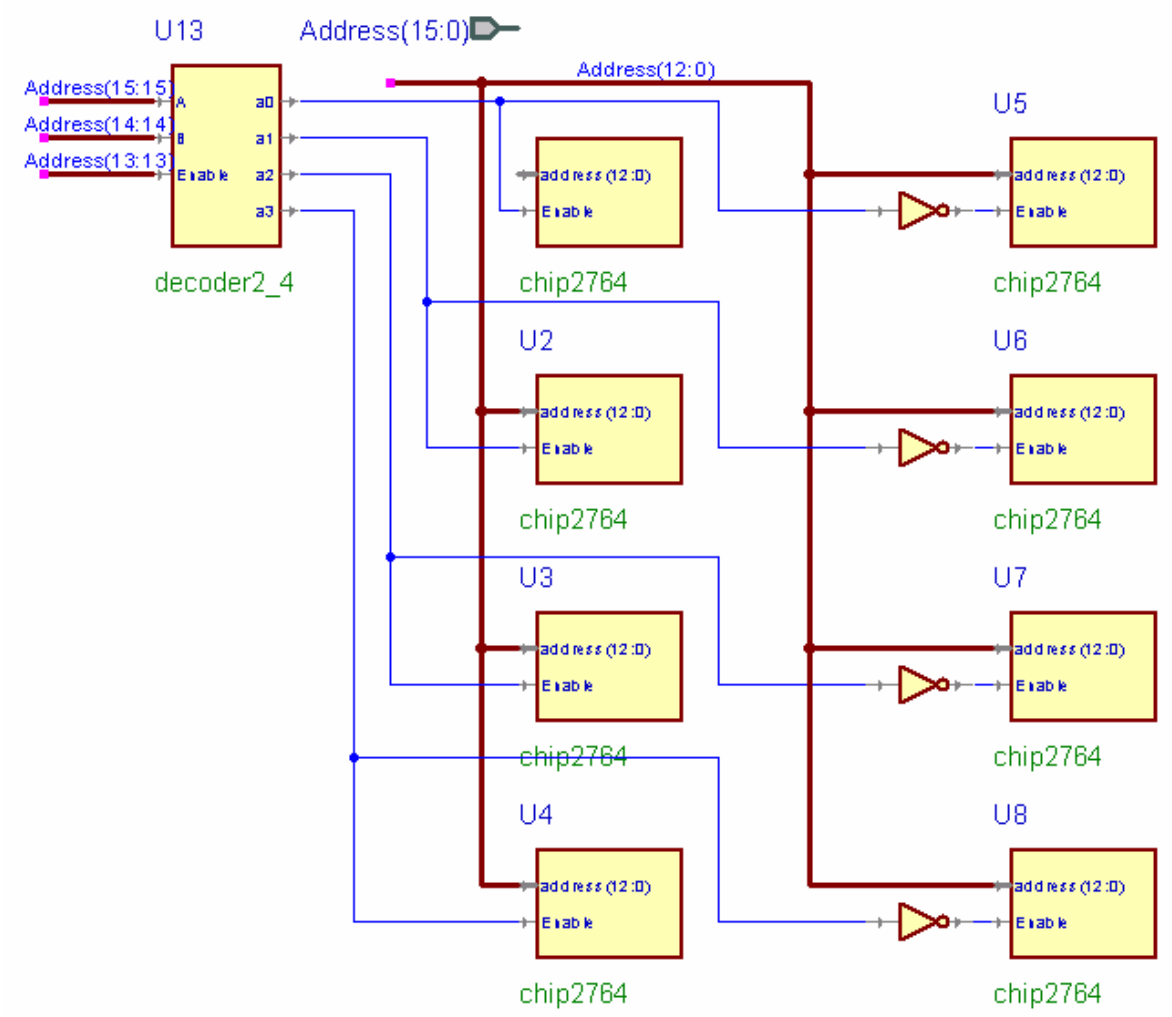


Exercise 4.22

(a) To simplify the diagram, output bits from the 2764 chips have not been shown. Dealing with the output would involve applying the OR function for each bit of the 2764 outputs with all the other 2764 chips. Note that the thicker wires are busses and are being used to reduce the number of wires in the diagram.

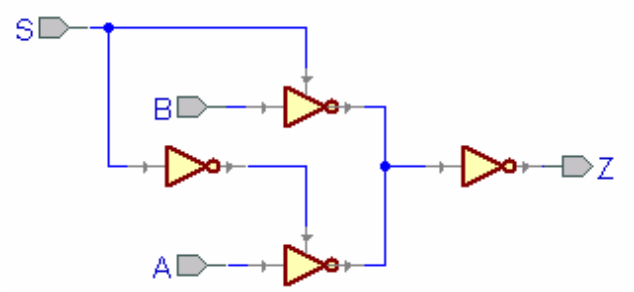


(b) The trick is to use the Enable signal on the decoder as another address bit. By inverting the enable on half of the 2764 chips, it will enable when the address bit is low and be disabled when that address bit is high.

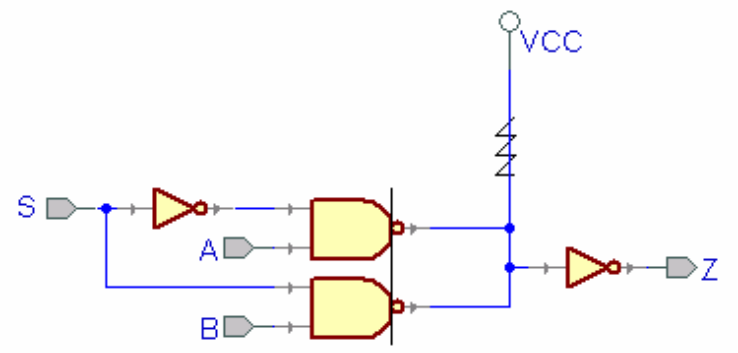


Exercise 4.23

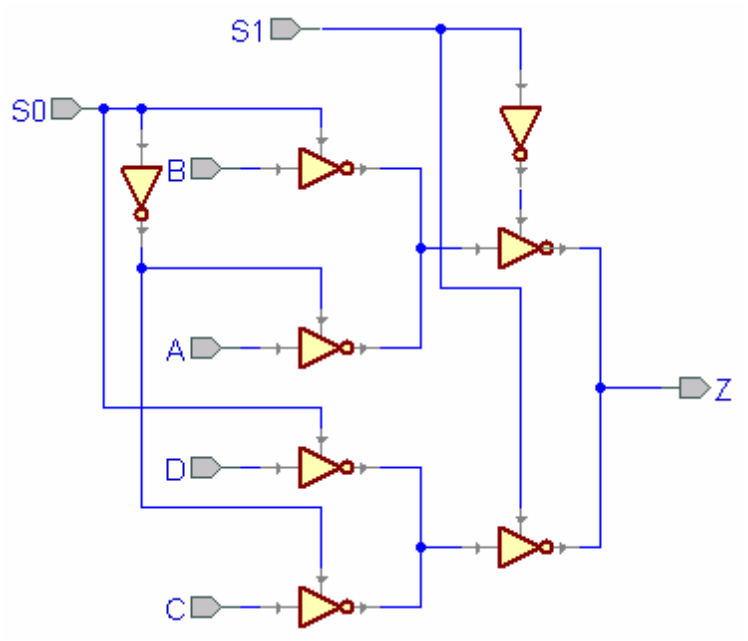
(a)

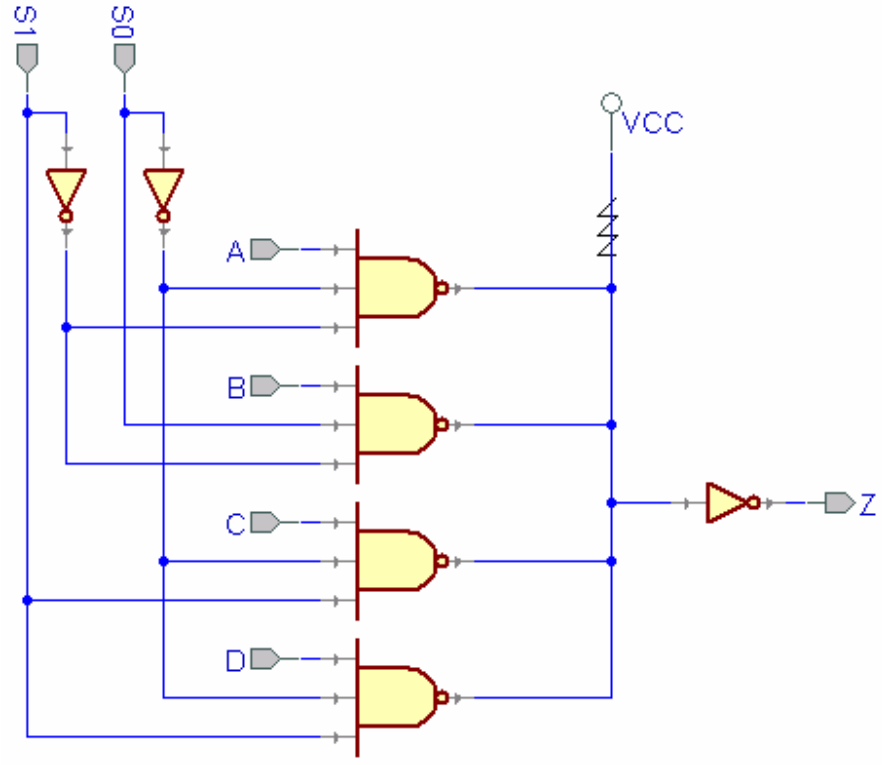


(b)



(c)





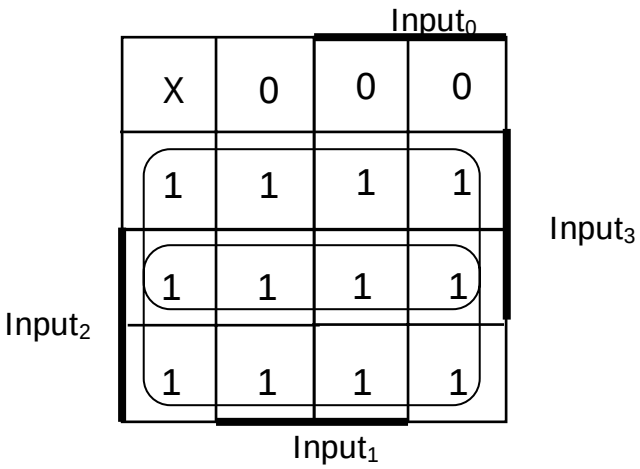
Exercise 4.24

(a)

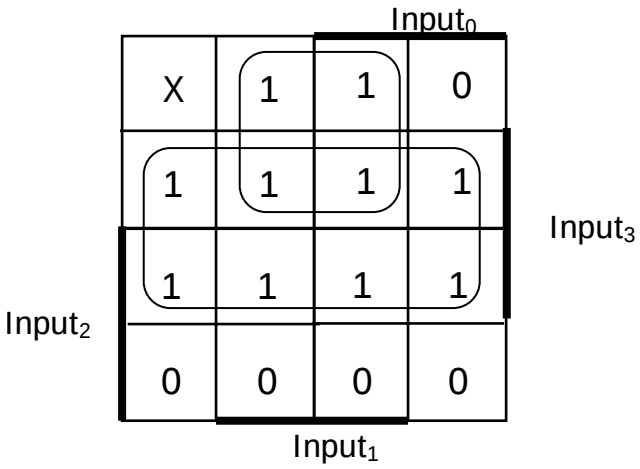
$Input_0$	$Input_1$	$Input_2$	$Input_3$	$Output_0$	$Output_1$
0	0	0	0	X	X
0	0	0	1	1	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	1	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	1	1	1
1	1	1	0	1	0
1	1	1	1	1	1

(b)

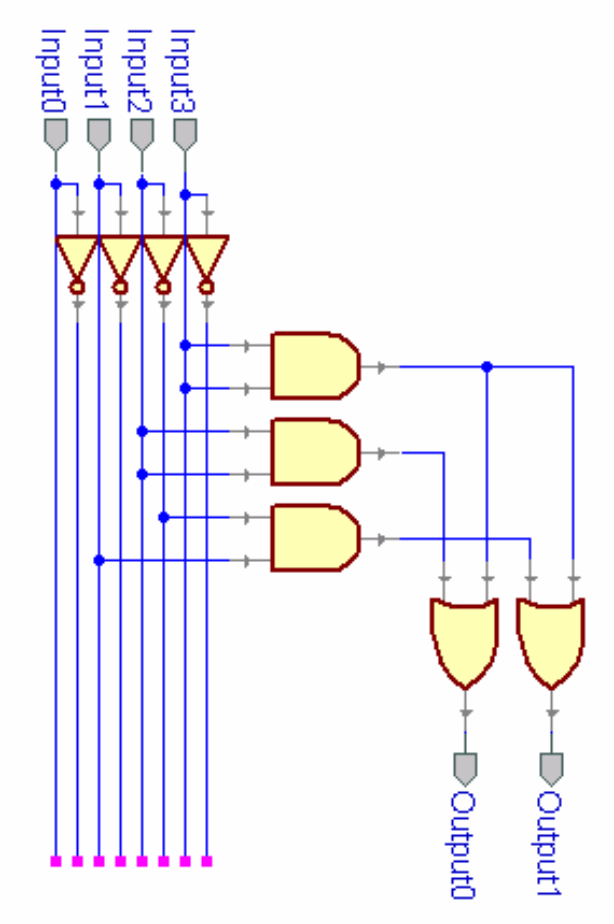
$Output_0 = Input_3 + Input_2$



$Output_1 = Input_1\ Input_2' + Input_3$



(c)



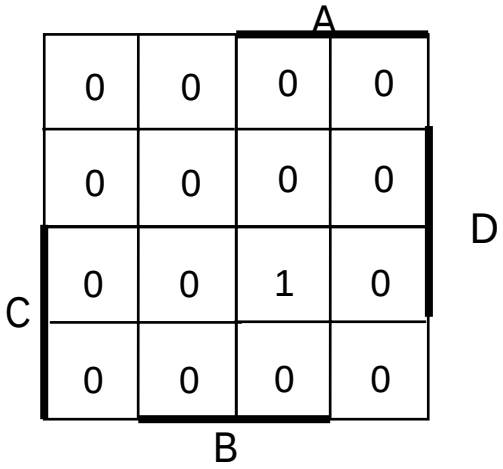
Exercise 4.25

(a)

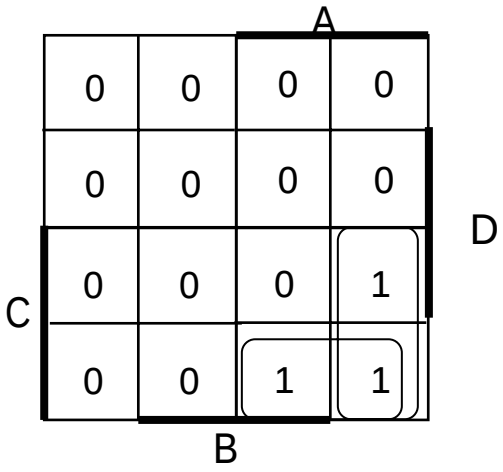
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

(b)

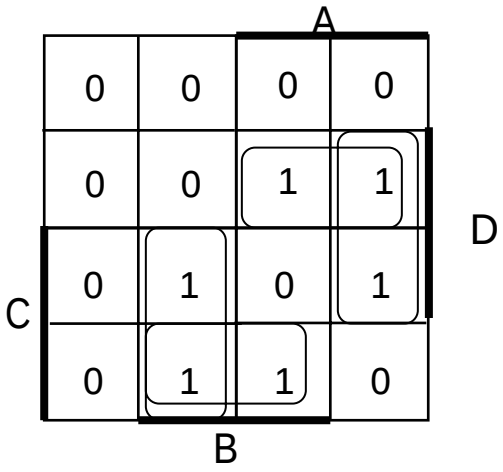
W = ABCD



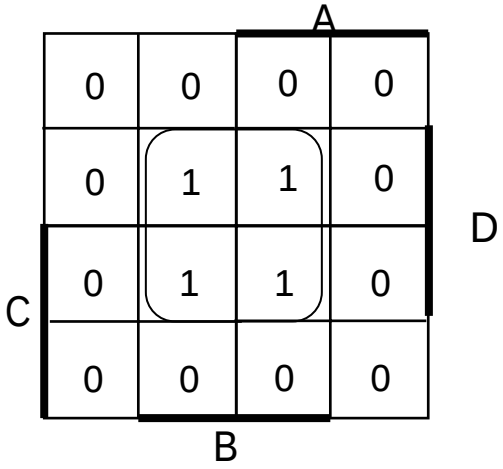
X = ACD’ + AB’C



Y = AB’D + AC’D + A’BC + BCD’



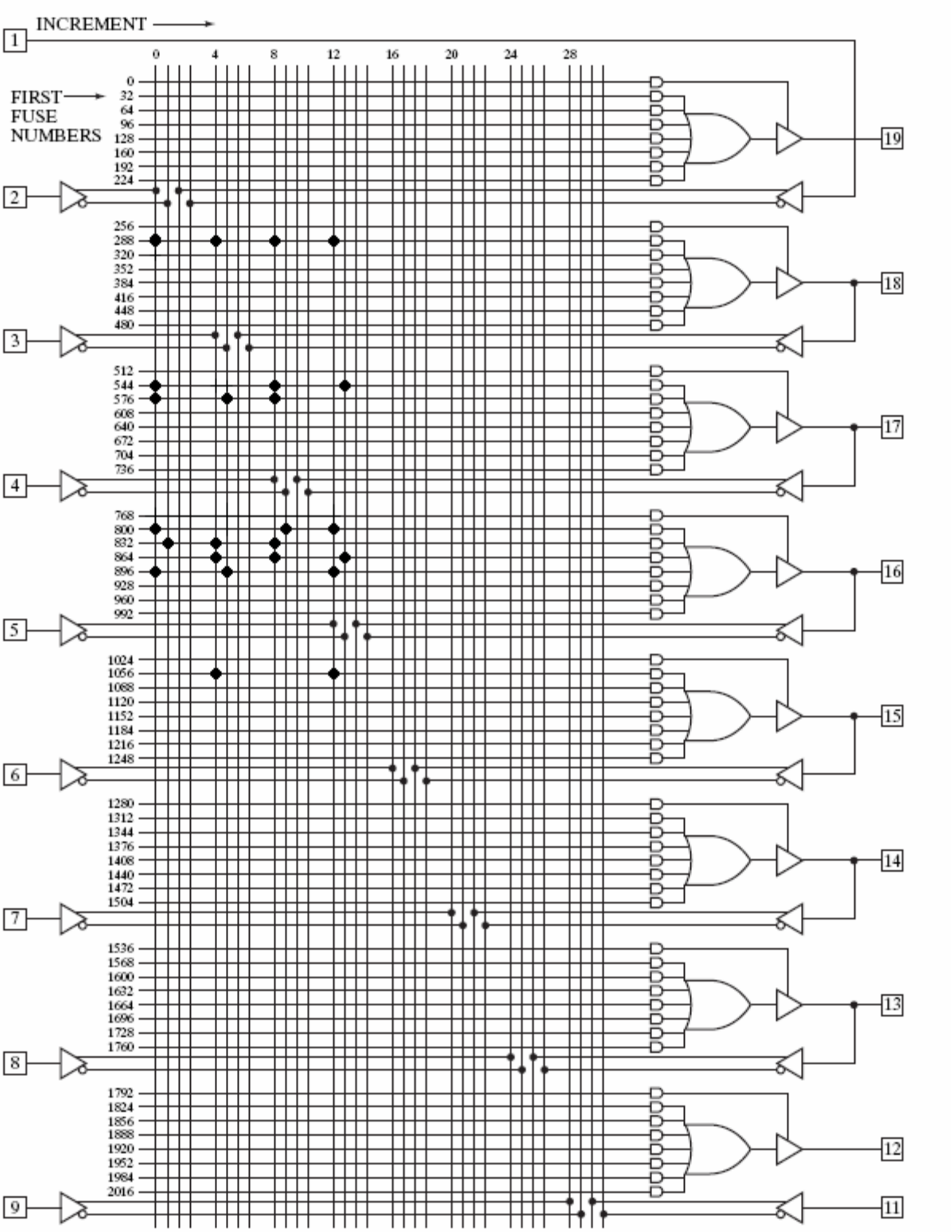
Z = BD



(c) The terms are mapped to the following pins:

Term	Pin #
A	2
B	3
C	4
D	5
W	18
X	17
Y	16
Z	15

This solution requires only one P16H8 PAL.



NOTE: FUSE NUMBER = FIRST FUSE NUMBER + INCREMENT

Exercise 4.26

(a) It should be recognized that V and W are the outputs of a full adder function. Using 3 variable K-maps reveals that W cannot be simplified, and V can be simplified to the following function:

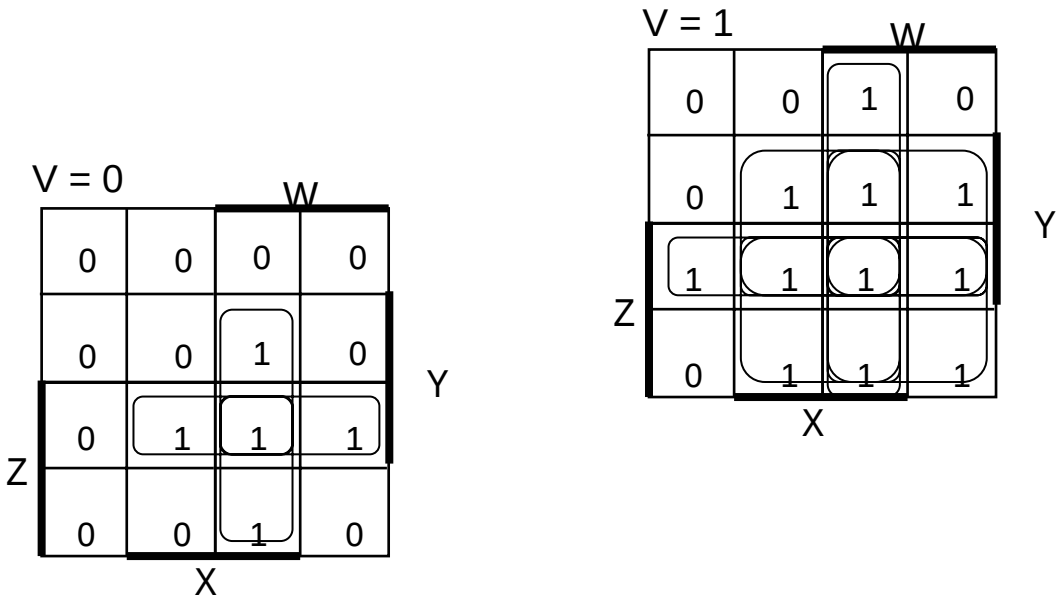
$$V = AB + BC + AC$$

$$W = A'B'C + A'BC' + ABC + AB'C'$$

(b)

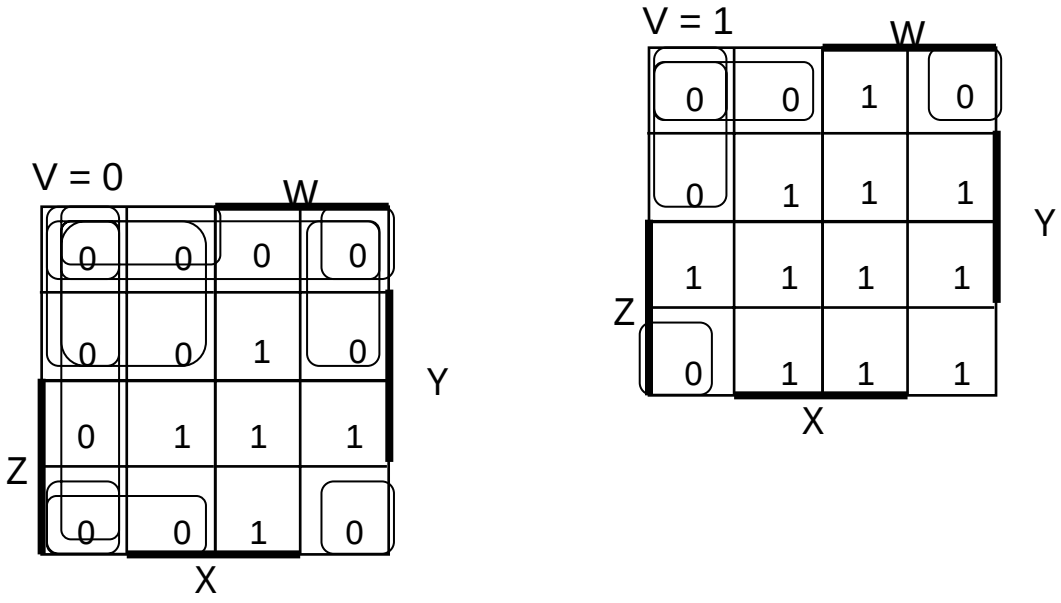
VWXYZ	Output
00000	0
00001	0
00010	0
00011	0
00100	0
00101	0
00110	0
00111	1
01000	0
01001	0
01010	0
01011	1
01100	0
01101	1
01110	1
01111	1
10000	0
10001	0
10010	0
10011	1
10100	0
10101	1
10110	1
10111	1
11000	0
11001	1
11010	1
11011	1
11100	1
11101	1
11110	1
11111	1

(c)



Q = WXY + WXZ + WYZ + XYZ + VYZ + VXY + VXZ + VWX + VWZ + VWY

(d)



Q = (W’ + X’ + Y’) (W’ + X’ + Z’) (W’ + Y’ + Z’) (X’ + Y’ + Z’) (V’ + W’ + X’) (V’ + W’ + Y’) (V’ + W’ + Z’) (V’ + X’ + Y’) (V’ + X’ + Z’) (V’ + Y’ + Z’)

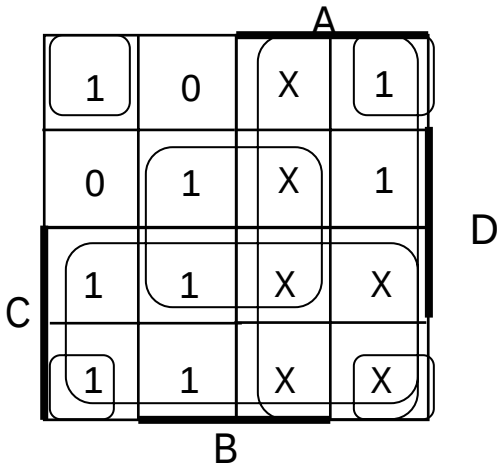
(e) This would require 3 CLB chips. Each circuit would use a separate CLB chip. In the case where only 4-input CLB's are available, each of the terms in the minimized sum-of-products form can be implemented with 3 or less terms. This means that dividing Circuit #3 into 4 CLB's with VWXY, VWXZ, VXYZ, WXYZ plus one additional CLB to determine if any of these are asserted.

Exercise 4.27

X = C’ + D’

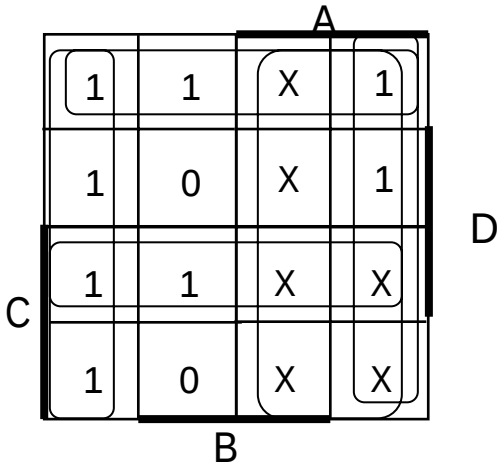
Y = B’C’

C_0 = C_3 + A’BX’ + ADY
= B’C’D’ + A’CD’ + A’BC’D + A’B’CD + A’BCD + AB’C’D



= A + BD + C + B’D’ √

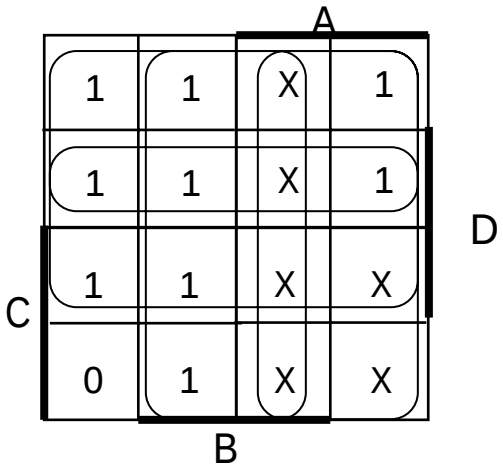
C_1 = Y + A’C_5’ + C’D’C_6
= B’C’ + A’ (C + D) (B’ + D) (B’ + C) + AC’D’ + BC’D’
= B’C’ + A’ (B’C + B’D + CD) + AC’D’ + BC’D’
= B’C’ + A’B’C + A’B’D + A’CD + AC’D’ + BC’D’



= A + C’D’ + CD + B’ √

$C_2 = C_5 + A'B'D + A'CD$

$= B'C'D' + AB'C' + A'BC' + A'BD' + A'B'D + A'CD$

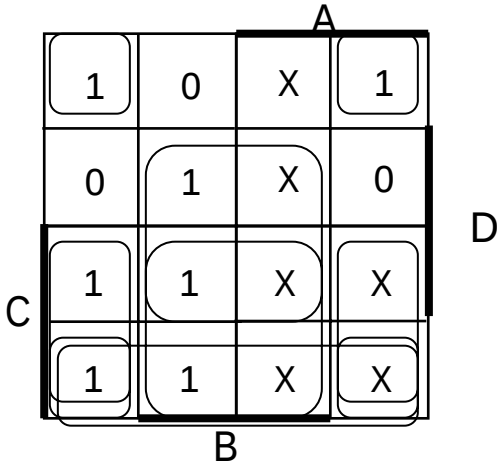


$= A + B + C' + D$ √

$C_3 = C_4 + BDC_5 + A'B'X'$

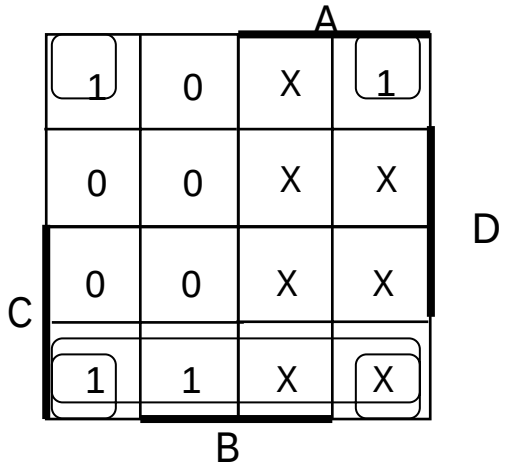
$= B'C'D' + A'CD' + BD\,(B'C'D' + AB'C' + A'BC' + A'BD') + A'B'\,(C' + D')'$

$= B'C'D' + A'CD' + A'BC'D + A'B'CD$



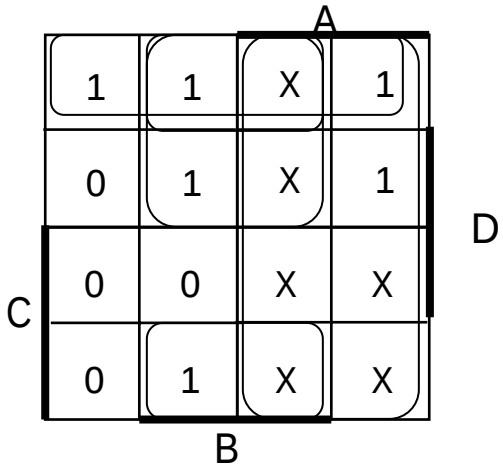
$= B'D' + CD' + BC'D + B'C$ √

$C_4 = D'Y + A'CD'$
$= B'C'D' + A'CD'$



$= B'D' + CD'$ √

$C_5 = C'C_4 + AY + A'BX$
$= C' (B'C'D' + A'CD') + A (B'C') + A'B (C' + D')$
$= B'C'D' + AB'C' + A'BC' + A'BD'$



$= A + C'D' + BD' + BC'$ √

$C_6 = AC_4 + CC_5 + C_4'C_5 + A'B'C$

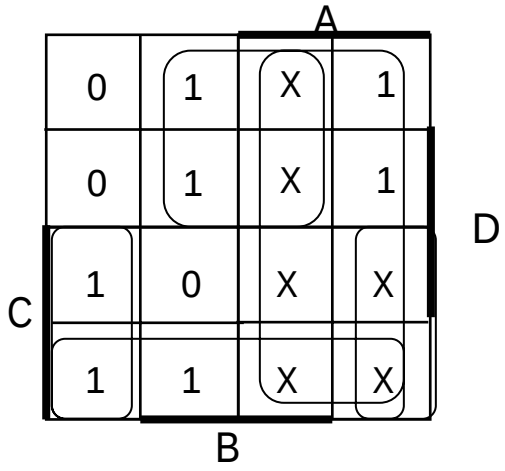
$= AB'C'D' + C (B'C'D' + AB'C' + A'BC' + A'BD') + (B'C'D' + A'CD')'$
$(B'C'D' + AB'C' + A'BC' + A'BD') + A'B'C$

$= AB'C'D' + A'BCD' + (B'C'D')' (A'CD')' (B'C'D' + AB'C' + A'BC' + A'BD')$
$+ A'B'C$

$= AB'C'D' + A'BCD' + (B + C + D) (A + C' + D) (B'C'D' + AB'C' + A'BC'$
$+ A'BD') + A'B'C$

$= AB'C'D' + A'BCD' + (AB + AC + D + BC') (B'C'D' + AB'C' + A'BC'$
$+ A'BD') + A'B'C$

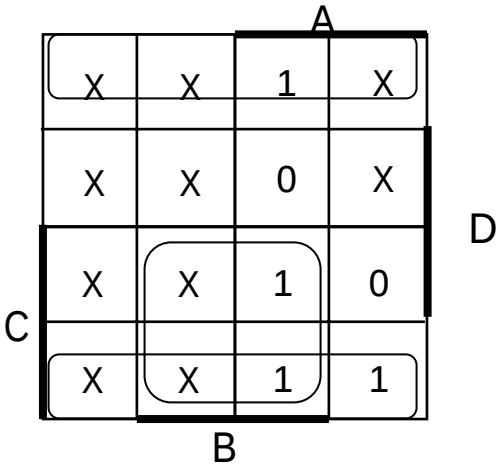
$= AB'C'D' + A'BCD' + AB'C'D + A'BC'D + A'BC' + A'B'C$



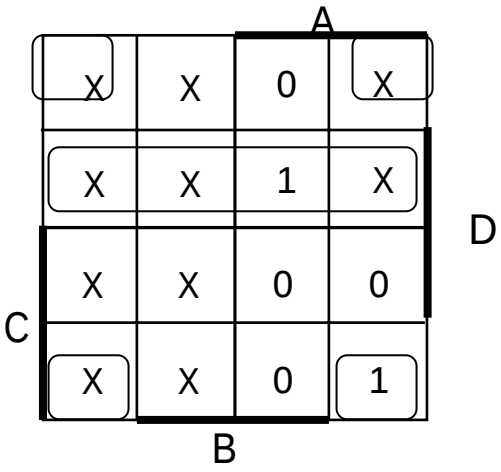
$= A + CD' + BC' + B'C$ √

Exercise 4.28

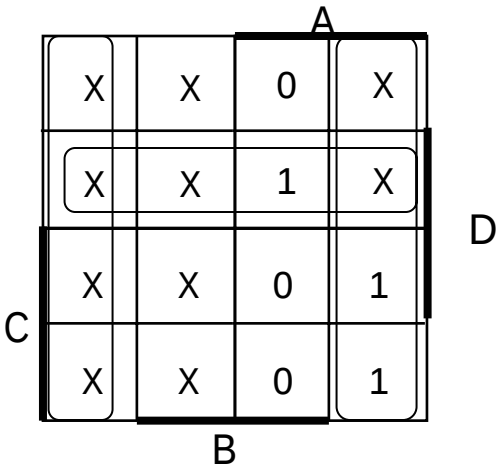
(a) The following K-maps determine the minimized sum-of-products forms for each of the 7 terms for the LED display decoder.



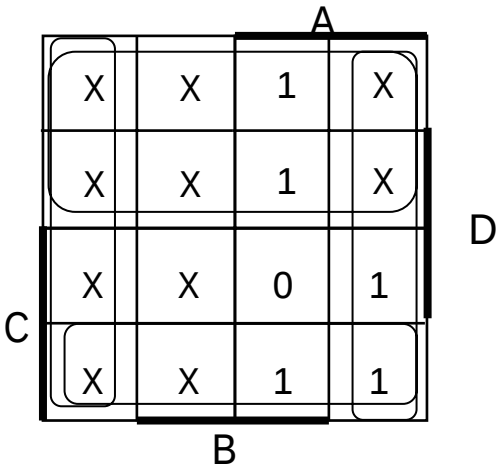
$C_0 = BC + D'$



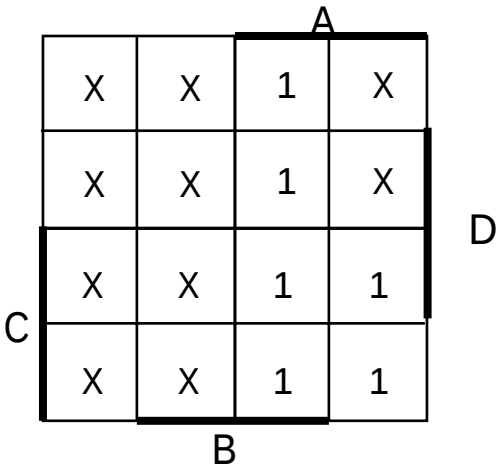
$C_1 = C'D + B'D'$



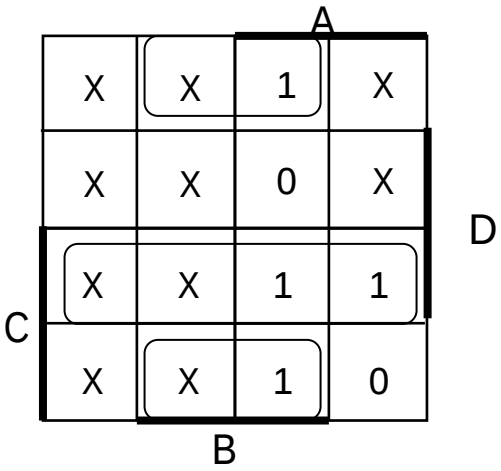
$C_2 = B' + C'D$



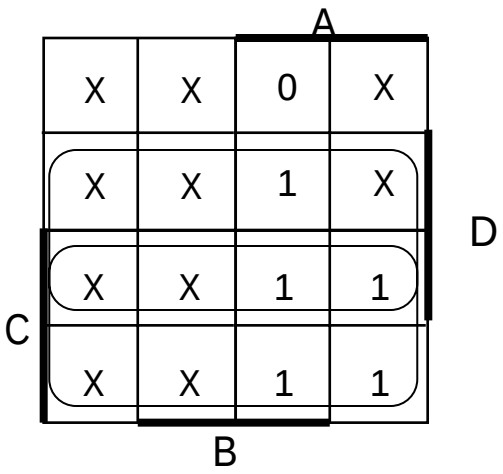
$C_3 = C' + B' + D'$



$C_4 = 1$

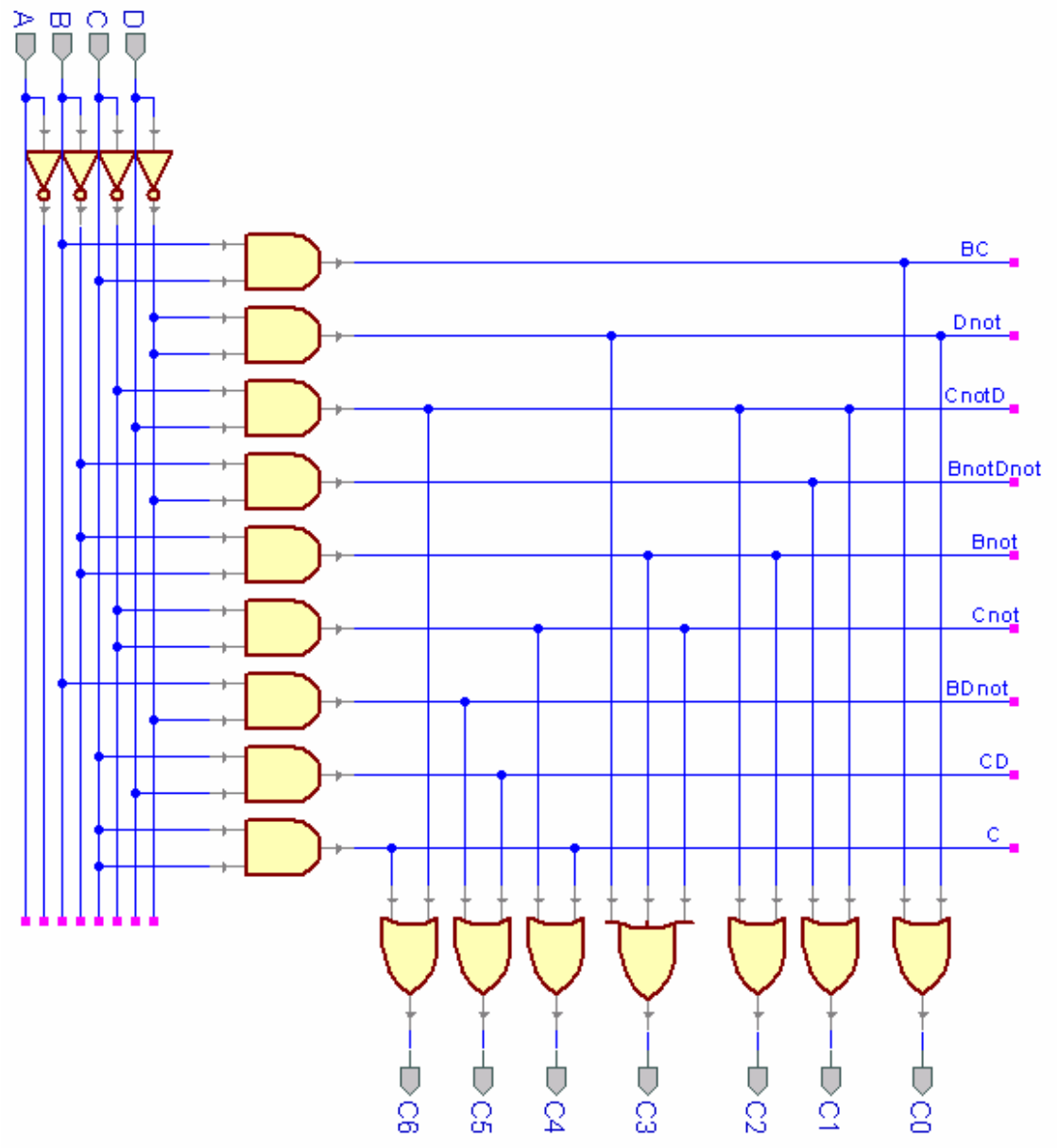


$C_5 = BD' + CD$

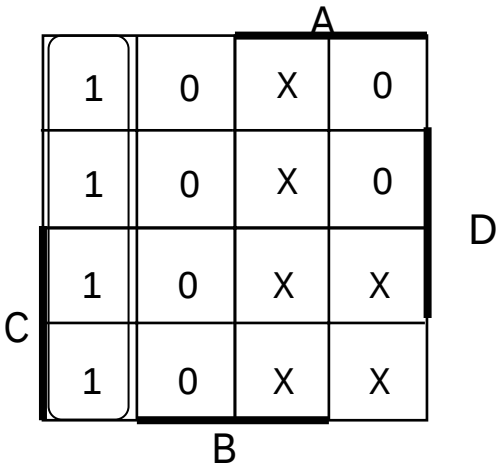


$C_6 = C + D$

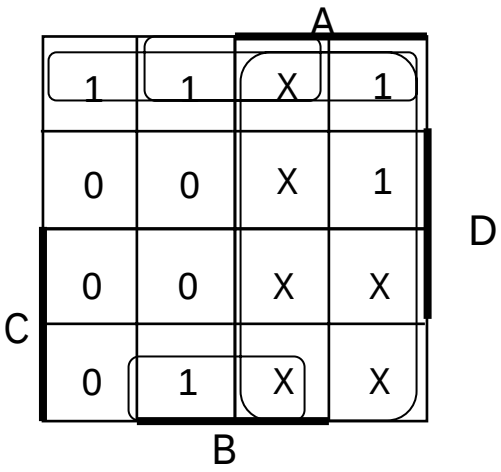
(b) Notice in the implementation some of the expressions in the minimized sum-of-products form are implemented using other available terms. For example, C_4 uses C' and C to implement the "true" function.



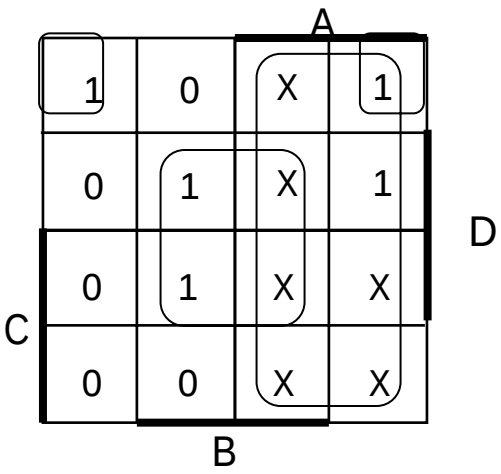
Exercise 4.29



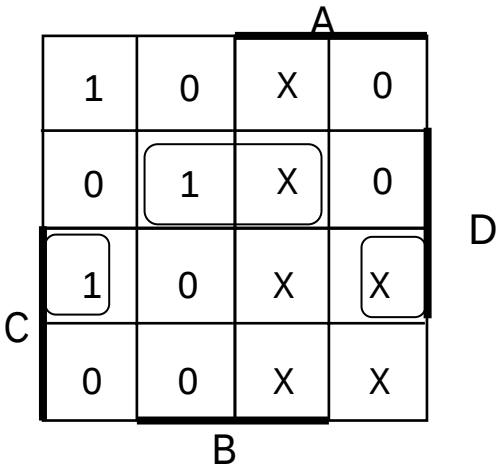
$C_0 = A'B'$



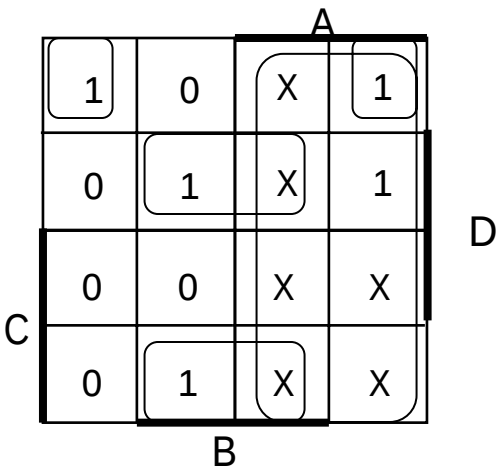
$C_1 = C'D' + A + BD'$



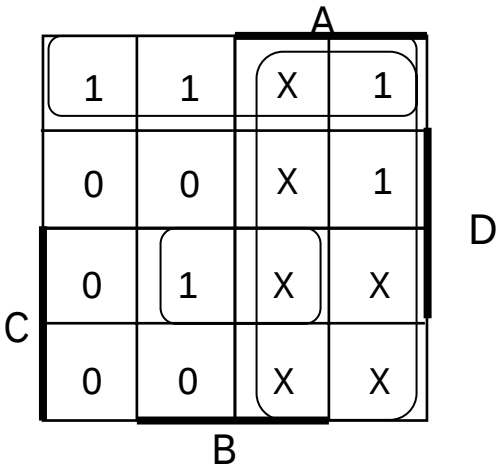
$C_2 = B'C'D' + A + BD$



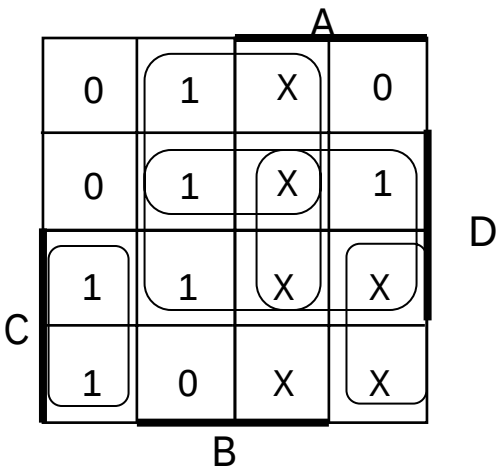
$C_3 = A'B'C'D' + BC'D + B'CD$



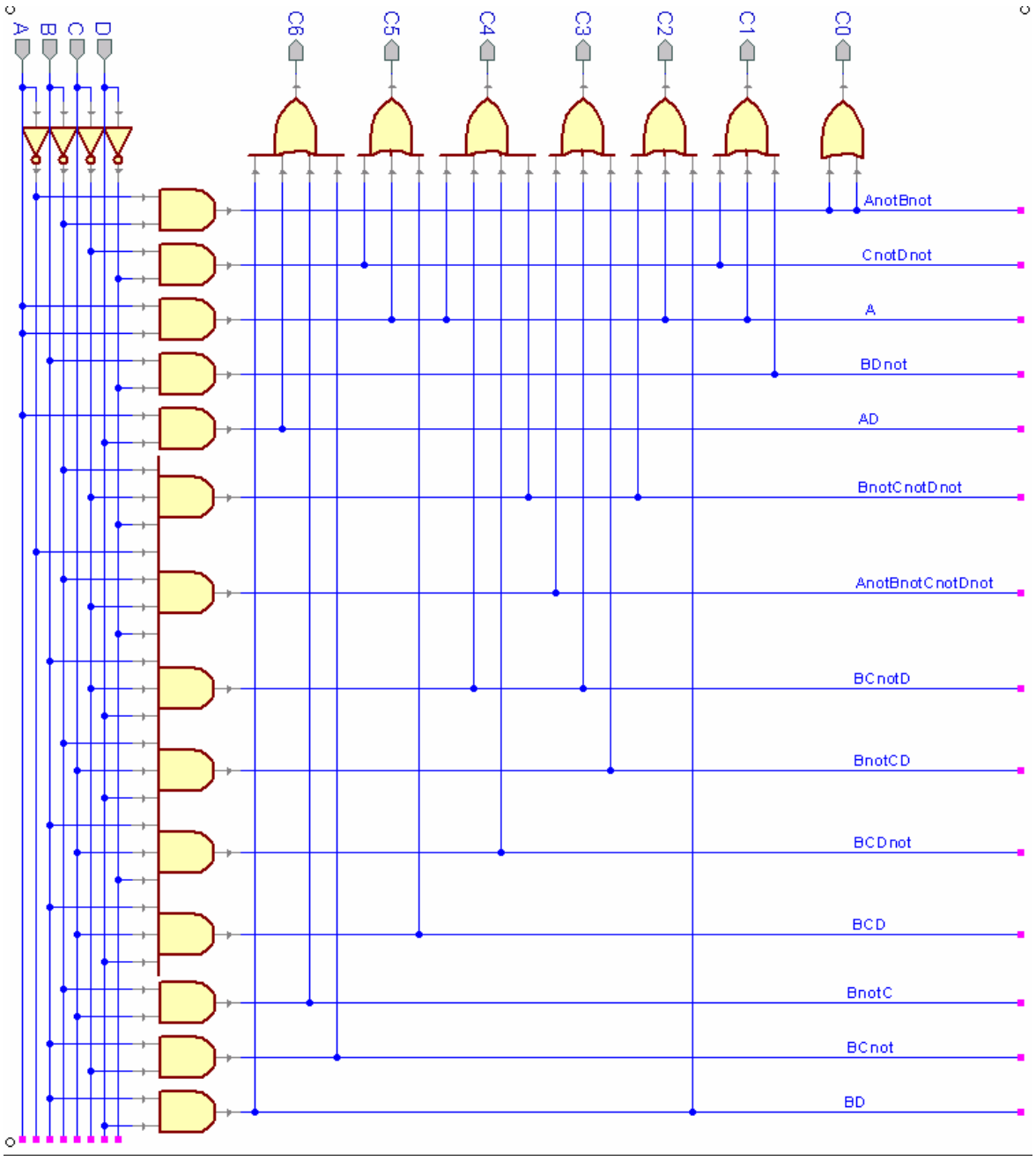
$C_4 = A + BC'D + BCD' + B'C'D'$



$C_5 = C'D' + A + BCD$

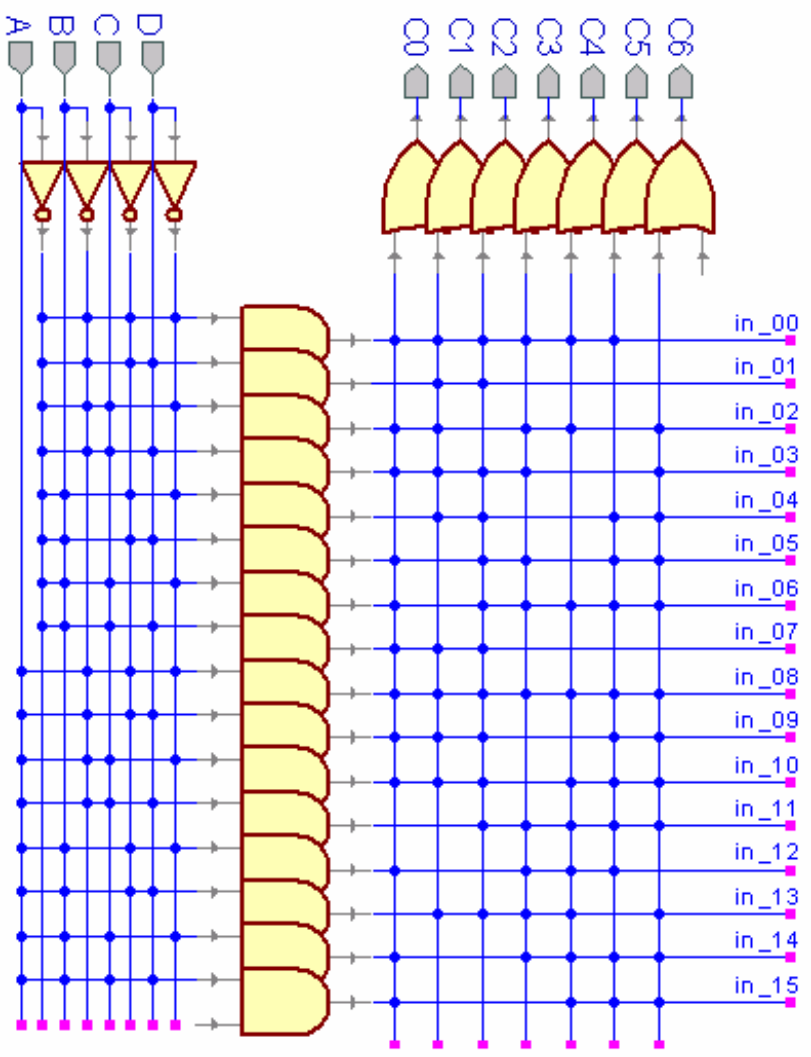


$C_6 = BD + AD + B'C + BC'$


A
B
C
D
C6
C5
C4
C3
C2
C1
C0
AnotBnot
CnotDnot
A
BDnot
AD
BnotCnotDnot
AnotBnotCnotDnot
BCnotD
BnotCD
BCDnot
BCD
BnotC
BCnot
BD

Exercise 4.30

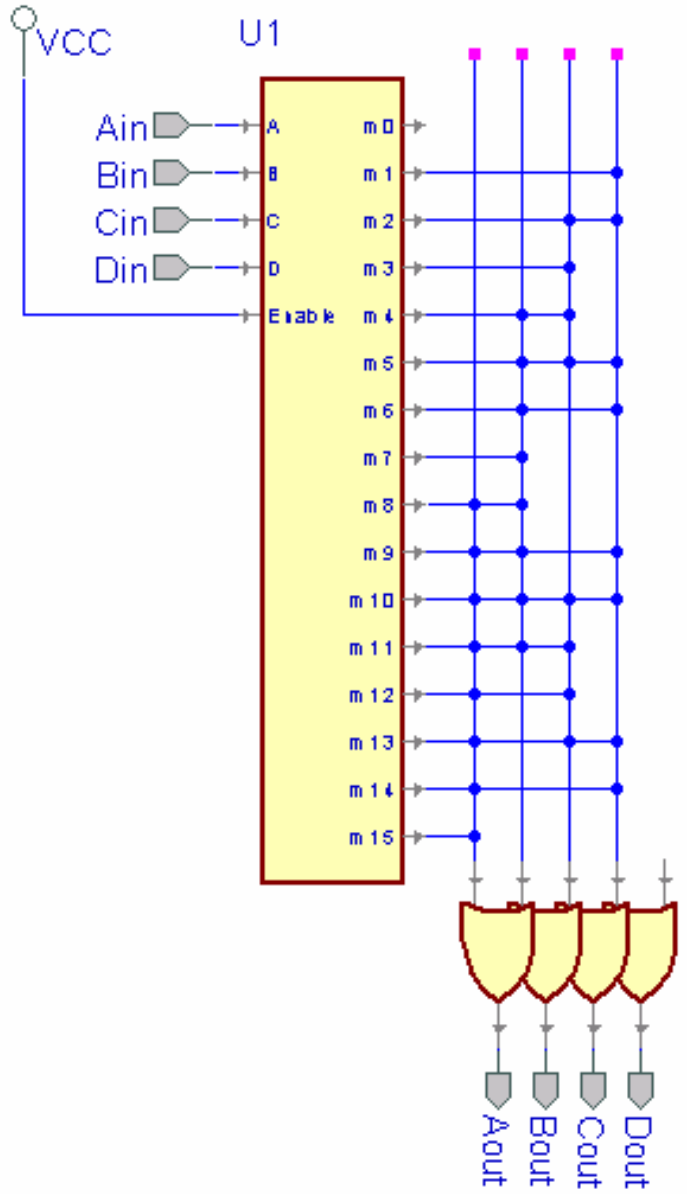
It is really tempting to use K-maps to try and simplify each of the output functions. However, using the K-map method eventually leaves it so each function is harder to implement using existing terms, and usually gives a lot more than 16 AND terms. Instead of using the K-map, using the brute force 0000 to 1111 leaves only 16 AND terms and 7 OR terms as expected. However each of the OR terms will have larger fan-in than if the K-map method was used. Note: the diagram below utilizes the short-hand for a PLA implementation in order to preserve space.



Exercise 4.31

A_{in}	B_{in}	C_{in}	D_{in}	A_{out}	B_{out}	C_{out}	D_{out}
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	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

ROM Implementation



PLA Implementation