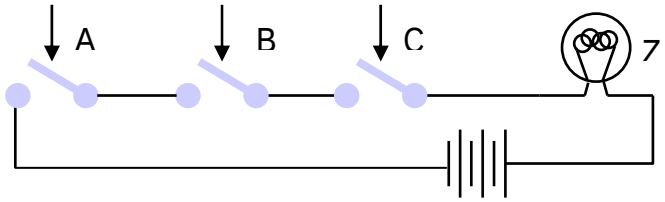
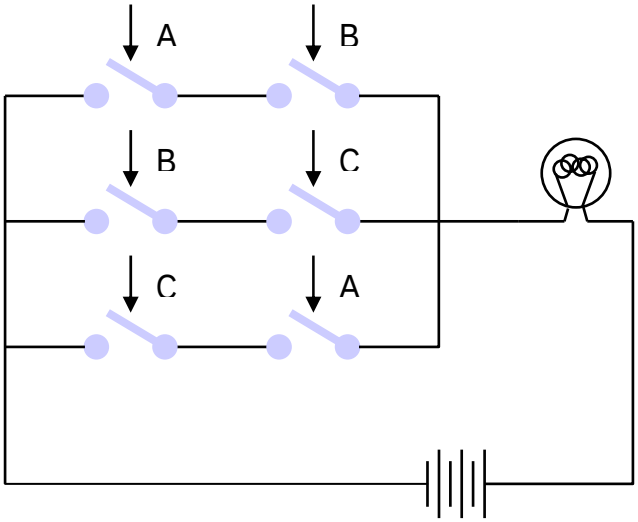
Exercise 1.1

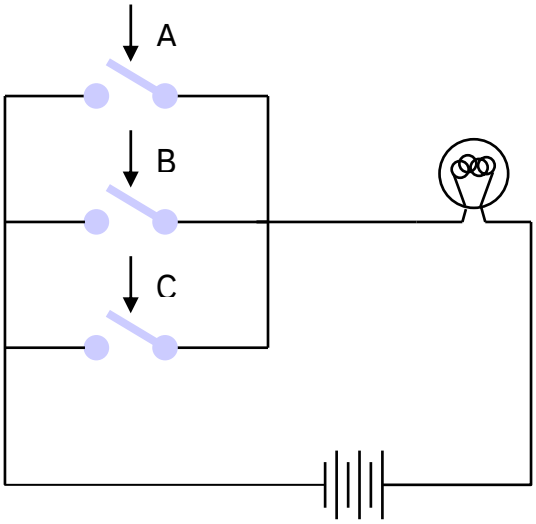
(a) In order for the light bulb to be turned on, all of the switches have to be closed.



(b) If any two of the switches are closed the light will turn on.

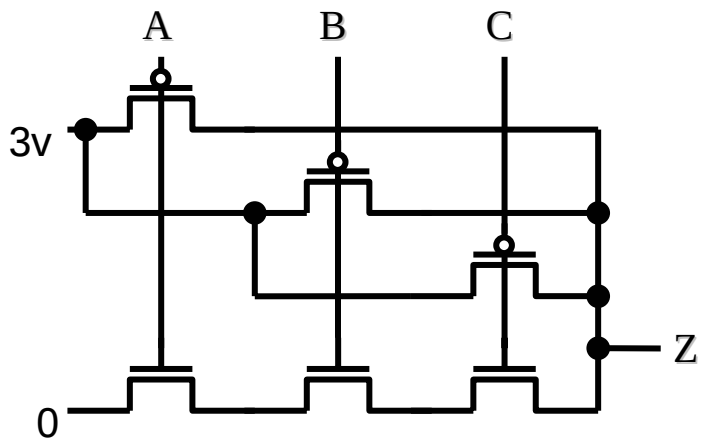


(c) If any one of the switches are closed the light will turn on.



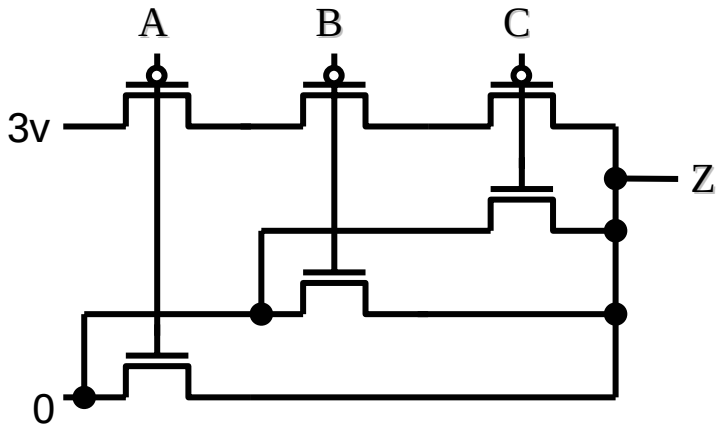
Exercise 1.2

(a) Three-input nand gate is constructed by putting three p-type transistors in parallel



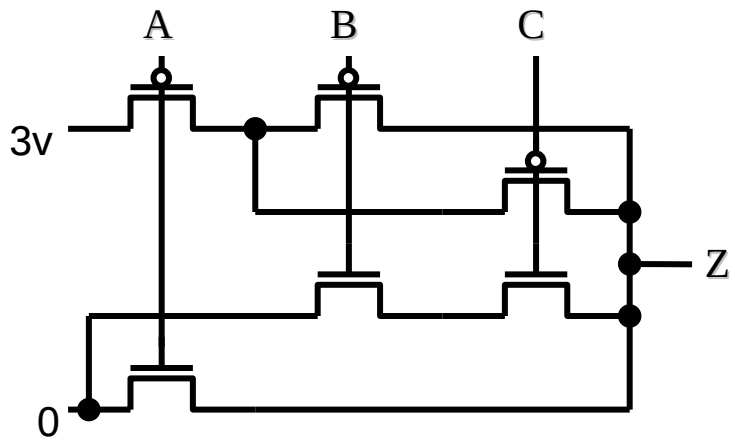
attached to high and three n-type transistors in series attached to low.

(b) Three-input nor gate is constructed by putting three p-type transistors in series

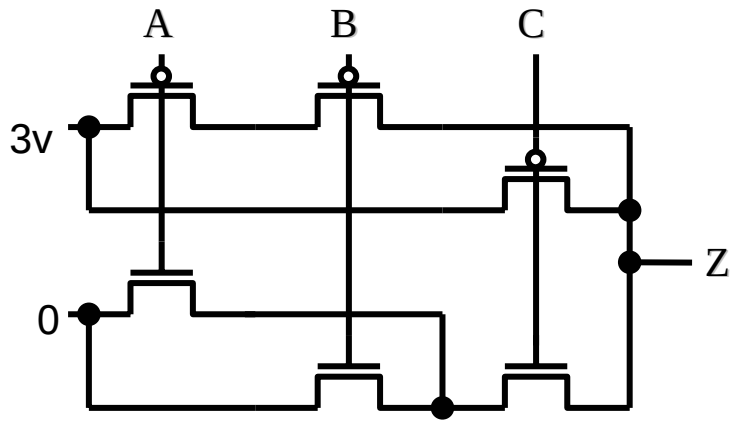


attached to high and three n-type transistors in parallel attached to low.

(c) Three-input gate for (A or (B and C))', which can also be written A' and (B' or C').



(d) Three-input gate for ((A or B) and C)', which can also be written (A' and B') or C'.



Exercise 1.3

There are several different ways to encode a deck of cards using binary numbers. Each scheme has its advantages and disadvantages. Here are two:

(a) Observe that the cards come in groups of 13 (suit). Therefore each card has 2 values that distinguish it: suit (Hearts, Clubs, Diamonds, Spades) and value (Ace, 2, ..., King). Since the suit can be one of four values, we need 2 bits to encode the suit (0, 1, 2, 3 assigned to clubs, diamonds, hearts, spades, respectively). The value ranges from 1 to 13 (1=ace, 2, 3, ... , 10, 11=jack, 12=queen, 13=king). Values of 0, 14, and 15 are unused. This encoding requires 6 bits:

$V_3\ V_2\ V_1\ V_0\ S_1\ S_0$

(b) We may want to make it easy to distinguish face cards. Another possible encoding would include a bit for face cards and non-face cards (F). A jack, queen, and king would be encoded with F=1 and the value=0001, value=0002, and value=0003, respecitively. Numbered cards and the ace would be encoded with F=0 and value equal to their number with the ace being a 1. The values 0, 11, 12, 13, 14, 15 would not be used when F=0, the values 0, 4, ... , 15 would not be used when F=1. This encoding uses 9 bits:

$F\ V_3\ V_2\ V_1\ V_0\ C\ D\ H\ S$

Exercise 1.4

(a) First encoding in 1.3. The diamond suit is numbered 1 and the jack is the card with value 11. Its encoding with this scheme is 1011 concatenated with 01 to yield 101101 or:V_3 and V_2' and V_1 and V_0 and S_1' and S_0

Second encoding in 1.3. The jack is a face card with value of 0001. When we concatenate F = 1,
value =0001, and suit, CDHS = 0100, we get an encoding of 100010100 or:F and V_3' and V_2' and V_1' and V_0 and D

(b) First encoding in 1.3. We only need to consider the value of the card. A seven is encoded as 0111 or:

V_3' and V_2 and V_1 and V_0

Second encoding in 1.3. A seven is not a face card and its value is 0111:F' and V_3' and V_2 and V_1 and V_0

(c) First encoding in 1.3. Any heart will have S_1 and S_0 both zero:S_1' and S_0'

Second encoding in 1.3. We only need to consider H:

H

Exercise 1.5

To figure out the logic equations for each day, d29, d30, and d31, simply look at the column in the truth table, Figure 1.18, that corresponds to the day and OR all the months together that have a *1*. Since February has two possibilities for number of days, you also need to AND *leap* to the month.

d29 = (m8' AND m4'AND m2 AND m1' AND leap)

d30 = (m8' AND m4 AND m2' AND m1') OR (m8' AND m4 AND m2 AND m1') OR (m8 AND m4' AND m2' AND m1) OR (m8 AND m4' AND m2 AND m1)

d31 = (m8' AND m4' AND m2' AND m1) OR (m8' AND m4' AND m2 AND m1) OR (m8' AND m4 AND m2' AND m1) OR (m8' AND m4 AND m2 AND m1) OR (m8 AND m4' AND m2' AND m1') OR (m8 AND m4' AND m2 AND m1') OR (m8 AND m4 AND m2' AND m1')

Exercise 1.6

Deriving d31 in terms of d28, d29, and d30 can be done easily by just saying if it's not d28 and it's not d29 and it's not d30 then it's day d31.

d31 = (d28' AND d29' AND d30')

Exercise 1.7

To derive an expression for the months that contain ‘R’, which we will label ‘mR’, begin by constructing a truth table as in Figure 1.18.

month	mR
0000 -----	----
0001 (Jan)	1
0010 (Feb)	1
0011 (Mar)	1
0100 (Apr)	1
0101 (May)	0
0110 (Jun)	0
0111 (Jul)	0
1000 (Aug)	0
1001 (Sep)	1
1010 (Oct)	1
1011 (Nov)	1
1100 (Dec)	1
1101 -----	----
1110 -----	----
1111 -----	----

Once the truth table is constructed, simply run down the ‘mR’ column and “OR” all of the months together that have a ‘1’ in their row.

mR = (m8’ AND m4’ AND m2’ AND m1) OR (m8’ AND m4’ AND m2 AND m1’)
OR (m8’ AND m4’ AND m2 AND m1) OR (m8’ AND m4 AND m2’ AND m1’)
OR (m8 AND m4’ AND m2’ AND m1) OR (m8 AND m4’ AND m2 AND m1’)
OR (m8 AND m4’ AND m2 AND m1) OR (m8 AND m4 AND m2’ AND m1’)

Exercise 1.8

Encoding the months from 0 (January) to 11 (December) rather than 1 to 12 changes the binary number for each month by one. The new truth table will look like this:

month	leap	d28	d29	d30	d31
0000	---	0	0	0	1
0001	0	1	0	0	0
0001	1	0	1	0	0
0010	---	0	0	0	1
0011	---	0	0	1	0
0100	---	0	0	0	1
0101	---	0	0	1	0
0110	---	0	0	0	1
0111	---	0	0	0	1
1000	---	0	0	1	0
1001	---	0	0	0	1
1010	---	0	0	1	0
1011	---	0	0	0	1
1100	---	---	---	---	---
1101	---	---	---	---	---
1110	---	---	---	---	---
1111	---	---	---	---	---

Once the truth table is constructed, simply run down each column corresponding to d29, d30, and d31 and "OR" together all the months that have a "1" in their row.

d29 = (m8' AND m4' AND m2' AND m1 AND leap)

d30 = (m8' AND m4' AND m2 AND m1) OR (m8' AND m4 AND m2' AND m1)
OR (m8 AND m4' AND m2' AND m1') OR (m8 AND m4' AND m2 AND m1')

d31 = (m8' AND m4' AND m2' AND m1') OR (m8' AND m4' AND m2 AND m1')
OR (m8' AND m4 AND m2' AND m1') OR (m8' AND m4 AND m2 AND m1')
OR (m8' AND m4 AND m2 AND m1) OR (m8 AND m4' AND m2' AND m1)
OR (m8 AND m4' AND m2 AND m1)

Exercise 1.9

This new encoding scheme can be represented by slightly modifying the truth table from Figure 1.18. Basically, d28, d29, d30, and d31 is true if the number of days that it represents is greater than or equal to the number of days in a particular month.

month	leap	d28	d29	d30	d31
0000	---	---	---	---	---
0001	---	1	1	1	1
0010	0	1	0	0	0
0010	1	1	1	0	0
0011	---	1	1	1	1
0100	---	1	1	1	0
0101	---	1	1	1	1
0110	---	1	1	1	0
0111	---	1	1	1	1
1000	---	1	1	1	1
1001	---	1	1	1	0
1010	---	1	1	1	1
1011	---	1	1	1	0
1100	---	1	1	1	1
1101	---	---	---	---	---
1110	---	---	---	---	---
1111	---	---	---	---	---

Since every month has at least 28 days in it,

d28 = true

Since d29 only has one case where it's not true, it is more compact to write that one case and then negate it.

d29 = (m8' AND m4' AND m2 AND m1' AND leap')'

Since d30 only has two cases where it's not true, it is more compact to write those two cases and negate them.

d30 = ((m8' AND m4' AND m2 AND m1' AND leap') OR (m8' AND m4' AND m2 AND m1' AND leap))'

Factor out the month:

d30 = ((m8' AND m4' AND m2 AND m1') AND (leap' OR leap))'

Since (1 AND 0) = 1:

d30 = ((m8’ AND m4’ AND m2 AND m1’) AND 1)’

Or, simply:

d30 = (m8’ AND m4’ AND m2 AND m1’)’

Finally, d31 is unchanged from the original encoding scheme.

d31 = (m8’ AND m4’ AND m2’ AND m1) OR (m8’ AND m4’ AND m2 AND m1) OR (m8’ AND m4 AND m2’ AND m1) OR (m8’ AND m4 AND m2 AND m1) OR (m8 AND m4’ AND m2’ AND m1’) OR (m8 AND m4’ AND m2 AND m1’) OR (m8 AND m4 AND m2’ AND m1’)

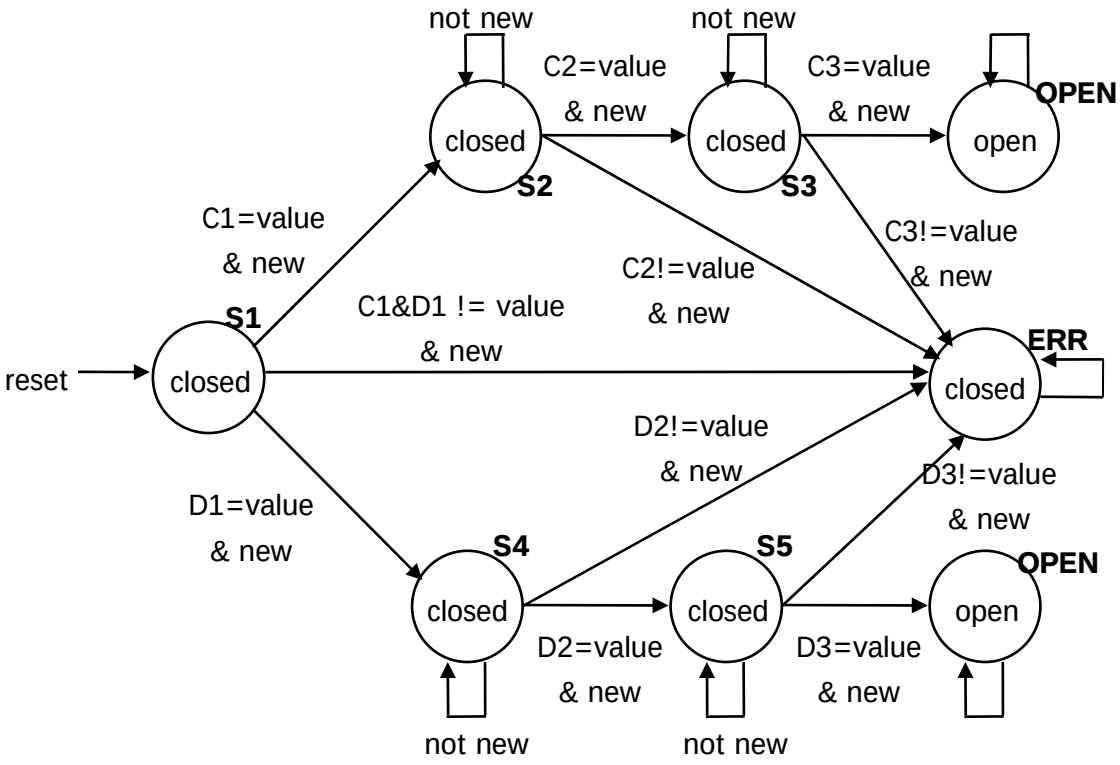
Exercise 1.10

To accomplish the goal of having two combinations which work to open the lock the original C code needs to branch after reading in the first correct value based on which of the two combinations it needs to check the last two values against. There are several ways to accomplish this.

(a) Combination lock program:

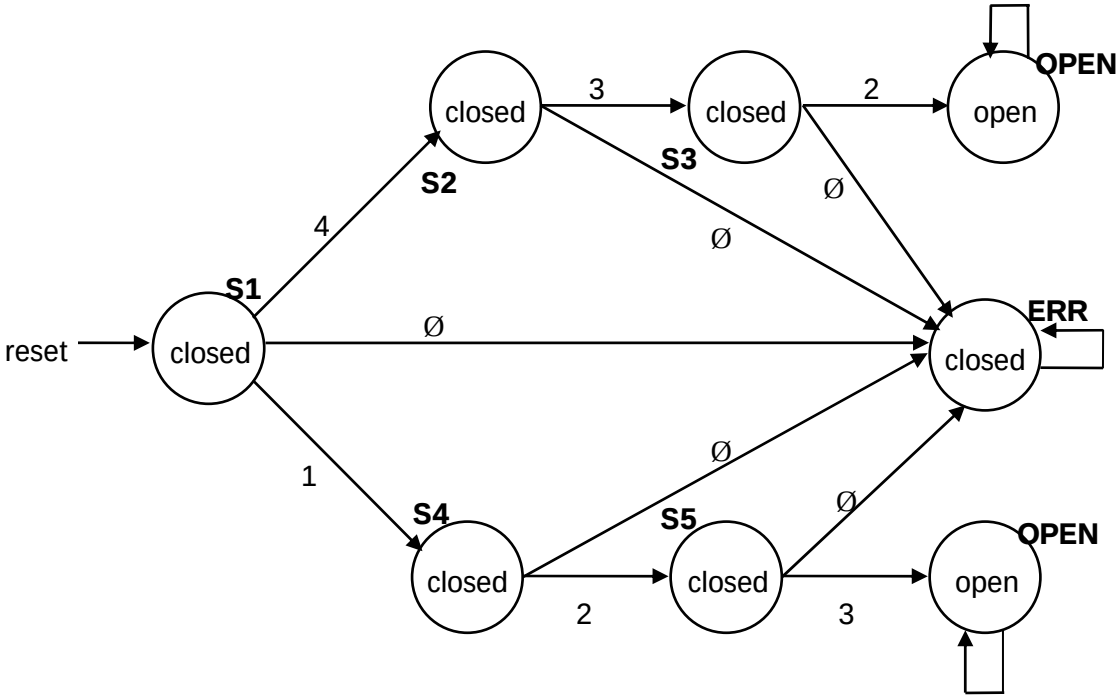
```
integer combination_lock ( ) {
    integer v1, v2, v3;
    integer error = 0;
    static integer c[3] = 3, 4, 2;
    static integer d[3] = 1, 2, 3;

    while (!new_value( ));
    v1 = read_value( );
    if (v1 == c[1]) {
        while (!new_value( ));
        v2 = read_value( );
        if (v2 != c[2]) then error = 1;

        while (!new_value( ));
        v3 = read_value( );
        if (v3 != c[3]) then error = 1;
    }else{
    if (v1 == d[1]) {
        while (!new_value( ));
        v2 = read_value( );
        if (v2 != d[2]) then error = 1;

        while (!new_value( ));
        v3 = read_value( );
        if (v3 != d[3]) then error = 1;
    }else
        error = 1;

    if (error == 1) then return(0); else return (1);
}
```

(b) State diagram for the modified combination lock code:



Exercise 1.11

The number along each arc represents the value that the input has to match in order to transition to the next state. If the input value is incorrect, the state machine will transition along the arc Ø.



Exercise 1.12

Since the states use one-hot encoding, we only need to look at one bit of the current state, S, to determine which state we are in. The next state functions derived from Figure 1.25:

NS_1 = (reset' new' S_0) + (reset)

NS_2 = (reset' new equal S_0) + (reset' new' S_1)

NS_3 = (reset' new equal S_1) + (reset' new' S_2)

NS_4 = (reset' new equal S_2) + (reset' S_3)

Exercise 1.13

A slightly different approach to encoding the state table of Figure 1.24 would be to use three bits rather than four. There are five different states, so three bits is the minimum number we could pick. 001_2 is the start state, 100_2 is the open state, and 000_2 is the error state. This would be an ideal implementation if we are limited by the amount of storage we have for saving state data. S is the current state and the next state functions are as follows:

NS_1 = (reset' new' S_2' S_1' S_0) + (reset)

NS_2 = (reset' new equal S_2' S_1' S_0) + (reset new' S_2' S_1 S_0')

NS_3 = (reset' new equal S_2' S_1 S_0') + (reset new' S_2' S_1 S_0)

NS_4 = (reset' new equal S_2' S_1 S_0) + (reset' S_2 S_1' S_0')

Exercise 1.14

We can use the lower two bits of the next state, which are NS_0 and NS_1, to encode the multiplexer control signals. The reason this works is because we are only concerned with the multiplexer output when the next state is S1, S2, or S3, which corresponds to NS being 001_2, 010_2, or 011_2, respectively. The rest of the cases are don't-cares.

$Mux_1 = NS_0$

$Mux_2 = NS_1$

Exercise 1.15

(a) All of the control signals are true:

A and B and C

(b) Any two of the control signals are true:

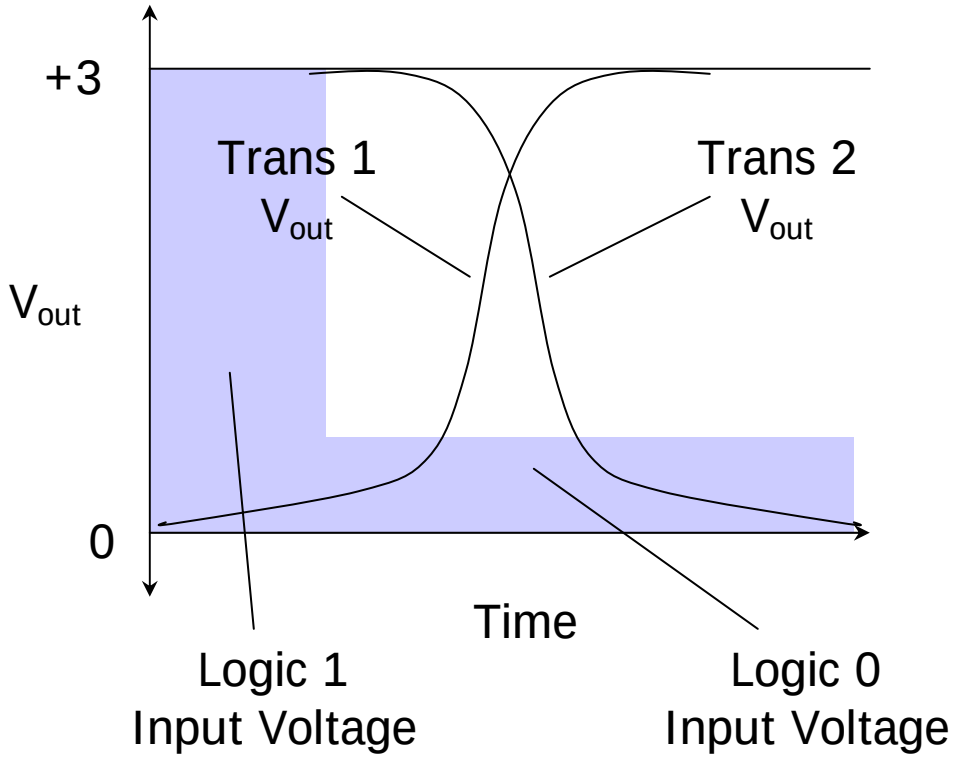
(A and B) or (B and C) or (C and A)

(c) Any one of the control signals is true:

A or B or C

Exercise 1.16

Initially, at time 0, the output of the two inverters will be a logic one because the input voltage is 3 volts, which is inverted by the first inverter and then again by the second inverter. As the voltage drops, the first inverter will eventually switch to a logic 1, which will cause the output of the second inverter to switch to a logic 0 after a little more time elapses. This is what the graph of voltage versus time would look like:



Exercise 1.17

Truth table for: A + B = S

A_1	A_0	B_1	B_0	S_2	S_1	S_0
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

Where, A and B are 2-bit operands and S is the 3-bit result.

Exercise 1.18

Truth table for the three functions from exercise 1.1

(a) All control signals are true:

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

(b) Any two control signals are true:

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

(c) Any one control signal is true:

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

Exercise 1.19

Logic expressions for each of the three truth tables in exercise 1.18

(a) All of the control signals are true:

$$Z = A \cdot B \cdot C$$

(b) Any two of the control signals is true:

$$Z = A \cdot B + B \cdot C + C \cdot A$$

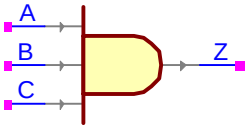
(c) Any one of the control signals is true:

$$Z = A + B + C$$

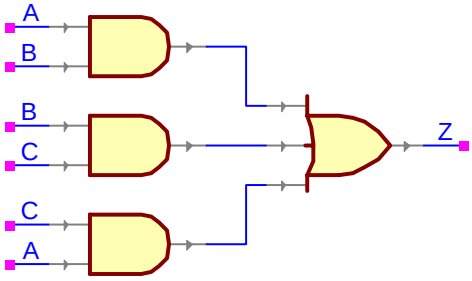
Exercise 1.20

Logic schematics for each of the functions from exercise 1.19

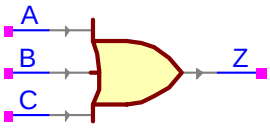
(a) A • B • C:



(b) A • B + B • C + C • A:

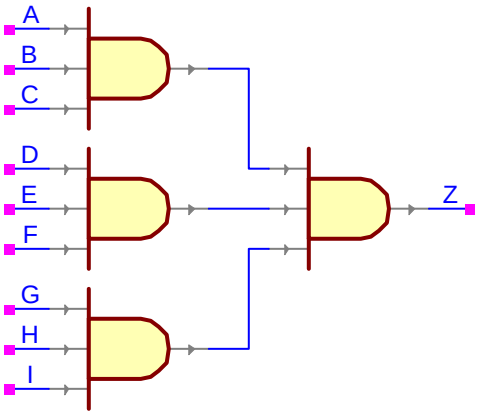


(c) A + B + C:



Exercise 1.21

A nine-input and gate can be constructed from four three-input and gates.



Exercise 1.23

(a) A washing machine that sequences through soak, wash, and spin cycles for a preset period of time would need a sequential circuit to achieve this. The reason being, that as the washing machine advances from one cycle to the next, it needs to know which cycle it is currently in to determine the next cycle. This is basically a state machine, where each cycle is a state and the next state depends on the previous state.

(b) An arithmetic circuit that divides two numbers is simply combinational logic. The reason for this is because it does not have to save any data to compute the quotient of two numbers. The circuit could be sequential if it was pipelined, meaning that it takes more than one clock cycle to compute the answer. In this case it would contain pipeline registers that save the partially computed answer.

(c) This machine is similar to the washing machine in that it has a number of states it has to cycle through as it dispenses the coins. Since the circuit needs to save its current state to be able to determine the next state, this is a sequential circuit.

(d) A digital alarm clock is a sequential circuit because it has to save a preset time and be able to go into an "alarm" state when the current time matches the preset time.

(e) A circuit that compares two numbers is a combinational circuit because it doesn't need to save anything to determine if the two numbers on its input match.

(f) This is a combinational circuit. This circuit configuration is exactly what an XOR logic gate does. If the inputs are different the output is one, if they are the same, the output is a zero.

(g) This circuit is sequential because it has to save the number of ones that have been input. It uses this saved value to determine whether it needs to output a one or zero.

(h) This is an example of using a multiplexer as a lookup table, which is a combinational circuit.

Exercise 1.24

The problem states that the digital system we are constructing takes a 4-bit binary number, N, from 0000_2 to 1111_2 and outputs a 1-bit binary number, F_4, set to 1 if the input is a multiple of 4 and 0 otherwise. So we are looking for numbers divisible by 4.

(a) Truth table:

N	F_4
0000	1
0001	0
0010	0
0011	0
0100	1
0101	0
0110	0
0111	0
1000	1
1001	0
1010	0
1011	0
1100	1
1101	0
1110	0
1111	0

(b) The Boolean equation for F_4 is simply the "OR" of all of the ones.

$F_4 = N_8'N_4'N_2'N_1' + N_8'N_4N_2'N_1' + N_8N_4'N_2'N_1' + N_8N_4N_2'N_1'$

(c) To implement this circuit, it would take four 4-input AND gates and one 4-input OR gate.

Exercise 1.25

(a) Truth table, Boolean equations and complexity of implementing F_2 and F_8:

N	F_2	F_8
0000	1	1
0001	0	0
0010	1	0
0011	0	0
0100	1	0
0101	0	0
0110	1	0
0111	0	0
1000	1	1
1001	0	0
1010	1	0
1011	0	0
1100	1	0
1101	0	0
1110	1	0
1111	0	0

$F_2 = N_8'N_4'N_2'N_1' + N_8'N_4'N_2N_1' + N_8'N_4N_2'N_1' + N_8'N_4N_2N_1' + N_8N_4'N_2'N_1' + N_8N_4'N_2N_1' + N_8N_4N_2'N_1' + N_8N_4N_2N_1'$

$F_8 = N_8'N_4'N_2'N_1 + N_8N_4'N_2'N_1'$

F_2 would take eight 4-input AND gates and one 8-input OR gate to implement.

F_8 would take two 4-input AND gates and one 2-input OR gate to implement.

(b) To implement F_4 in terms of F_2, you need to also include the N_2 bit in the equation.

$F_4 = F_2N_2'$

(c) To implement F_8 in terms of F_4, you need to also include the N_4 bit in the equation.

$F_8 = F_4N_4'$

Exercise 1.26

The calendar subsystem generates a 5-bit binary value, D, that corresponds to the number of days in the month. The month, M, can be represented by the 4-bit binary numbers 0001_2 to 1100_2. A leap year, L, is represented by an additional bit, which is 1 for a leap year and 0 for a non-leap year. Since the number of days in February depends on whether or not it is a leap year, February has two entries in the truth table.

(a) The truth table:

M	L	D
0000	---	-----
0001	---	11111
0010	0	11100
0010	1	11101
0011	---	11111
0100	---	11110
0101	---	11111
0110	---	11110
0111	---	11111
1000	---	11111
1001	---	11110
1010	---	11111
1011	---	11110
1100	---	11111
1101	---	-----
1110	---	-----
1111	---	-----

(b) The highest order three bits of D are all the same:

$$D_4\ldots D_2 = M_3'M_2'M_1'M_0 + M_3'M_2'M_1M_0' + M_3'M_2'M_1M_0 + M_3'M_2M_1'M_0' + M_3'M_2M_1'M_0 + M_3'M_2M_1M_0' + M_3'M_2M_1M_0 + M_3M_2'M_1'M_0' + M_3M_2'M_1'M_0 + M_3M_2'M_1M_0' + M_3M_2'M_1M_0 + M_3M_2M_1'M_0'$$

$$D_1 = M_3'M_2'M_1'M_0 + M_3'M_2'M_1M_0 + M_3'M_2M_1'M_0' + M_3'M_2M_1'M_0 + M_3'M_2M_1M_0' + M_3'M_2M_1M_0 + M_3M_2'M_1'M_0' + M_3M_2'M_1'M_0 + M_3M_2'M_1M_0' + M_3M_2'M_1M_0 + M_3M_2M_1'M_0'$$

$$D_0 = M_3'M_2'M_1'M_0 + M_3'M_2'M_1M_0'L + M_3'M_2'M_1M_0 + M_3'M_2M_1'M_0 + M_3'M_2M_1M_0 + M_3M_2'M_1'M_0' + M_3M_2'M_1M_0' + M_3M_2M_1'M_0'$$

(c) $D_4\ldots D_2$ can be implemented with 12 4-input AND gates, 1 12-input OR gate, and 4 3-input, 6 2-input, and 2 1-input NOT gates. D_1 requires 11 4-input AND gates, 1 11-input OR gate, and 3 3-input, 6 2-input, and 2 1-input NOT gates. D_0 will take 7 4-input AND gates, 1 5-input AND gate, 1 8-input OR gate, and 3 3-input, 4 2-input, and 1 1-input NOT gates to implement.

Exercise 1.27

We are given the four outputs: d28, d29, d30, and d31 as inputs to our component and translating them into the 5-bit binary value representing how many days are in that month. Since only one of the four inputs can be true at a given time, we only need to represent four cases with our truth table.

(a) The truth table:

d28	d29	d30	d31	D
1	0	0	0	11100
0	1	0	0	11101
0	0	1	0	11110
0	0	0	1	11111

(b) The highest order three bits of D are all the same:

$D_4 \ldots D_2$ = d28d29'd30'd31 + d28'd29d30'd31' + d28'd29'd30d31' + d28'd29'd30'd31

D_1 = d28'd29'd30d31' + d28'd29'd30'd31

D_0 = d28'd29d30'd31' + d28'd29'd30'd31

(c) $D_4 \ldots D_2$ can be implemented with 4 4-input AND gates, 1 4-input OR gate, and 4 3-input NOT gates. D_1 requires 2 4-input AND gates, 1 2-input OR gate, and 3 3-input NOT gates. D_0 will also take 2 4-input AND gates, 1 4-input OR gate, and 3 3-input NOT gates to implement.

(d) The calendar subsystem from exercise 1.9 takes a total of 21 gates to implement. This component requires a total of 21 gates, also, for a grand total of 42 gates. The calendar subsystem from exercise 1.26 uses a total of 65 gates. We can see that using the calendar subsystem from exercise 1.9 along with the component in this exercise to convert the four outputs into a binary representation of the number of days in a month is a much better solution.

Exercise 1.28

We know that no matter what month it is, the first three bits of D, $D_4 – D_2$, will always be true. Also, since only one of the inputs can be true at once, we don't need to include the inputs that are not true in our Boolean equations. After these simplifications the Boolean equations for D are:

(a) $D_4 \ldots D_2 = 1$

$D_1 = d30 + d31$

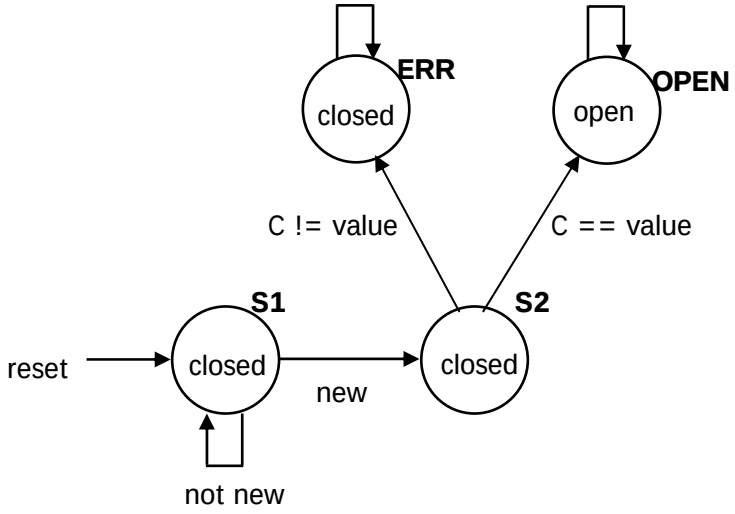
$D_0 = d29 + d31$

(b) $D_4 \ldots D_2$ don't require any logic gates. D_1 and D_0 can be implemented with only a single 2-input OR.

Exercise 1.29

Rather than reading in one value at a time, we can read in the whole combination and then transition to a state where we can determine whether or not the correct combination was entered.

(a) The new state diagram looks like this:



(b) The new design has a total of four states: S1, S2, OPEN, and ERROR.

reset	new	equal	S	NS	open/closed
1	-	-	-	S1	closed
0	0	-	S1	S1	closed
0	1	-	S1	S2	closed
0	-	0	S2	ERR	closed
0	-	1	S2	OPEN	open
0	-	-	OPEN	OPEN	open
0	-	-	ERR	ERR	Closed

(c) The original output function for open was:

open = (reset' new equal S3) + (reset' OPEN)

The new output function for open is:

open = (reset' equal S2) + (reset' OPEN)

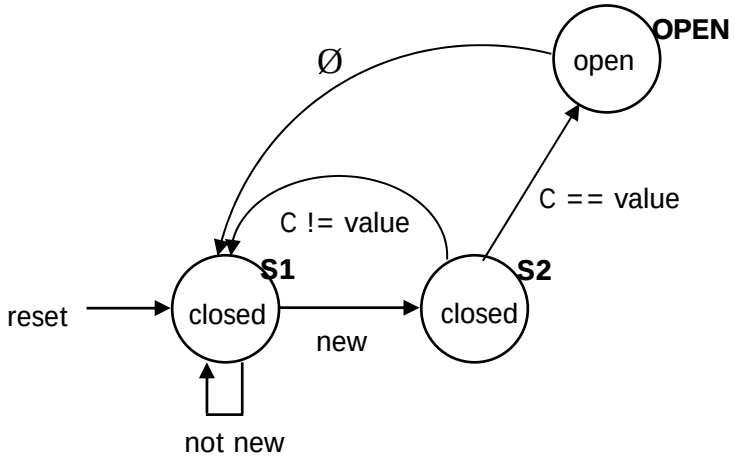
The complexity of the old open function is: 1 4-input AND, 1 2-input AND, and 1 2-input OR. The complexity of the new open function is: 1 3-input AND, 1 2-input AND, and 1 2-input OR.

So the new implementation for the open function is slightly better than the old one. However, the comparator for the new design will be much larger than the comparator in the old design.

Exercise 1.30

To achieve the automatic reset, we can delete the ERROR state and have state S2 transition back to the start state on a bad combination. Also, the OPEN state needs to transition automatically back to the start state after one cycle. This can be achieved by adding an arc back to the start state with the empty-set transition, Ø.

The new state diagram looks like this:

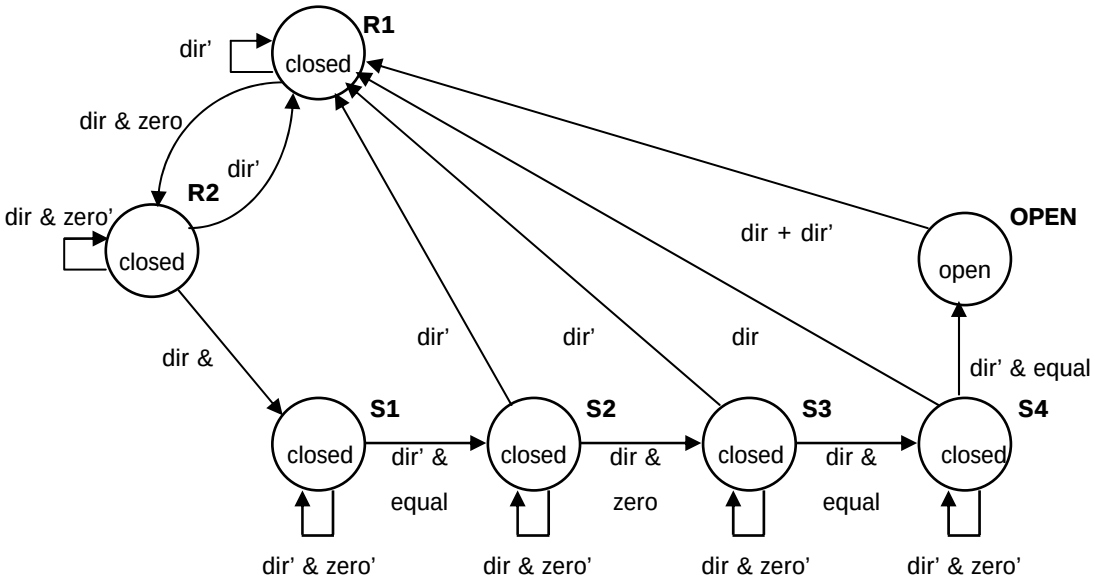


Exercise 1.31

(a) To implement this design, we will need the following input/outputs: dir, which is 1 for clockwise rotation and 0 for counter clockwise rotation; zero, to indicate whether or not we have passed 0; equal, to indicate if the current position matches the combination; state, to indicate the current state; next state, to indicate the next state to transition to; and open/close, to indicate whether or not the lock should open. The state table for this system looks as follows:

dir	zero	equal	state	next state	open/closed
0	-	-	R1	R1	closed
1	0	-	R1	R1	closed
1	1	-	R1	R2	closed
0	-	-	R2	R1	closed
1	0	-	R2	R2	closed
1	1	-	R2	S1	closed
1	-	-	S1	S1	closed
0	-	0	S1	S1	closed
0	-	1	S1	S2	closed
0	-	-	S2	R1	closed
1	0	-	S2	S2	closed
1	1	-	S2	S3	closed
0	-	-	S3	R1	closed
1	-	0	S3	S3	closed
1	-	1	S3	S4	closed
1	-	-	S4	R1	closed
0	-	0	S4	S4	closed
0	-	1	S4	OPEN	open
0	-	-	OPEN	R1	closed
1	-	-	OPEN	R1	closed

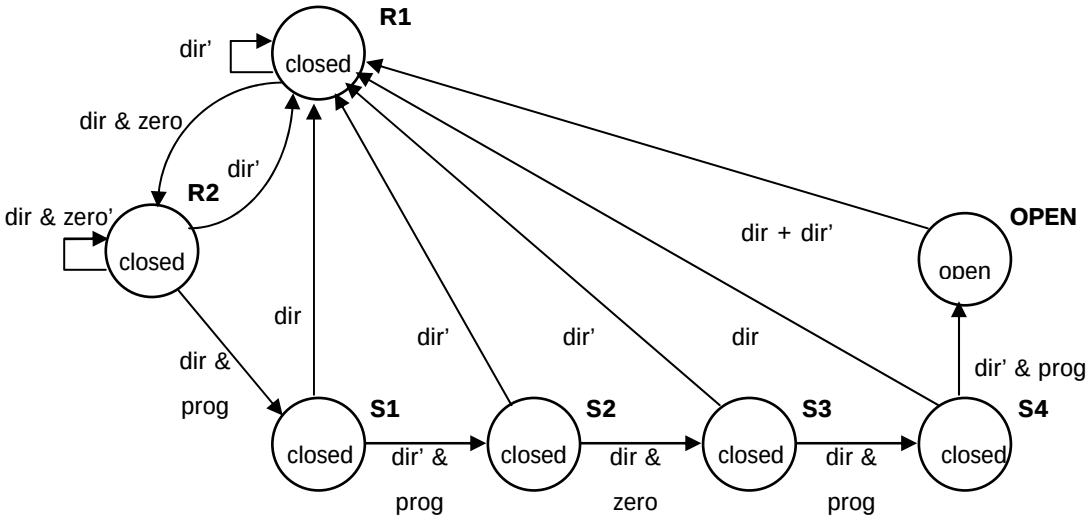
(b) For this system, we are going to need 2 reset states so that we can keep track of how many times we have passed by 0 before we transition into the start state. Since there is no way for the user to tell if he/she has transitioned back to the reset state, it is okay to transition from any state back to the first reset state. Also, the zero input is only set to 1 if the dial has passed 0 once, and 0 otherwise.



Exercise 1.32

(a) Essentially, all that you need to be able to program a new combination is an input that indicates that you are programming in a new combination. The new input is prog.

(b) The revised state diagram looks as follows:

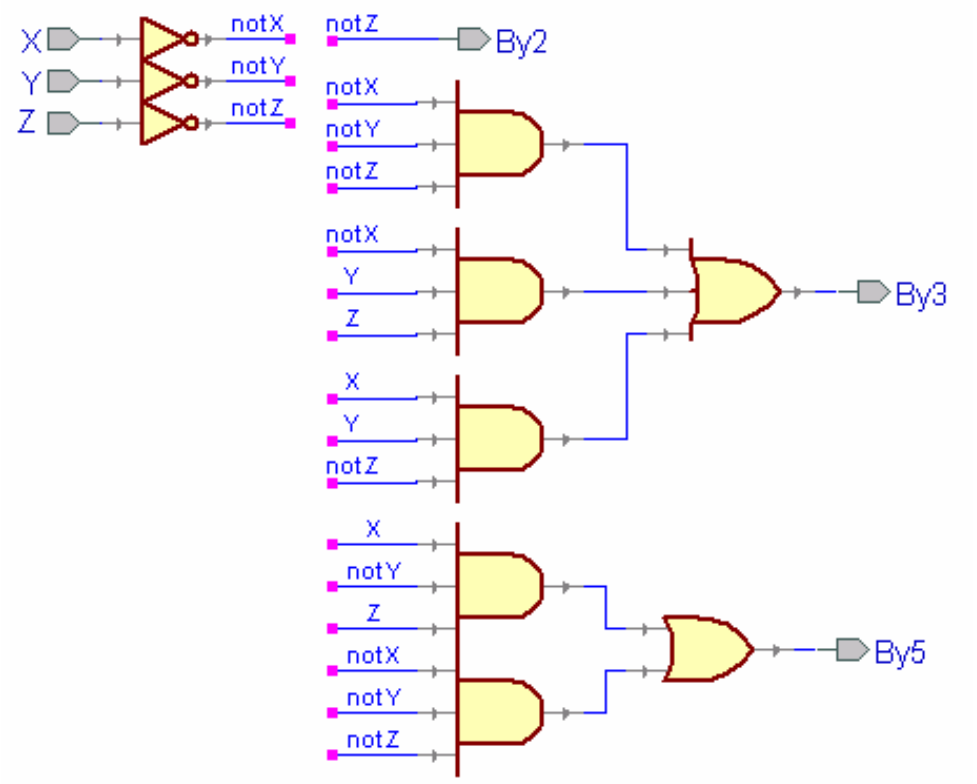


Exercise 2.1

The truth table for the functions looks as follows:

X	Y	Z	By2	By3	By5
0	0	0	1	1	1
0	0	1	0	0	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	1	0	1	1	0
1	1	1	0	0	0

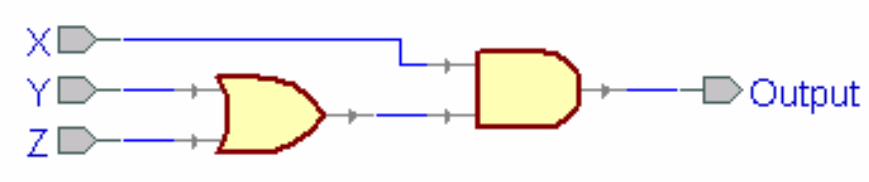
This translates into the following circuit. Please note that wires with the same name are considered to be connected.



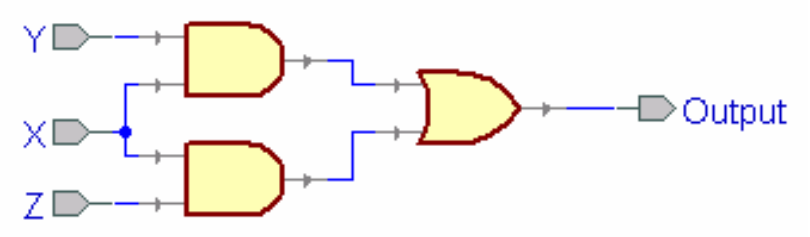
Exercise 2.2

Each of the formulas for parts a-e converts almost directly into logic gates.

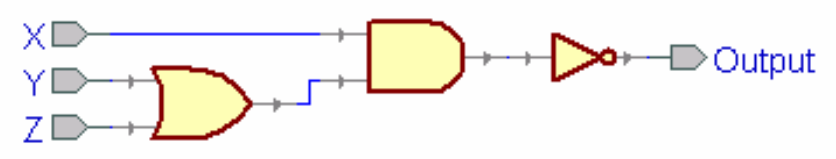
(a)



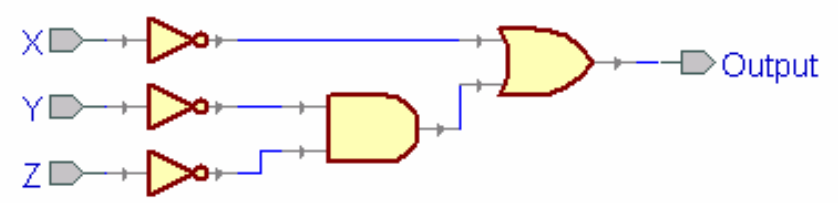
(b)



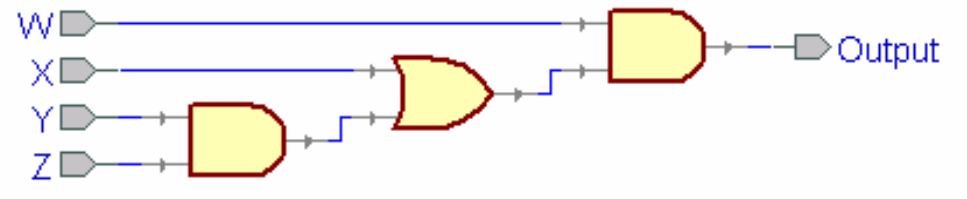
(c)



(d)



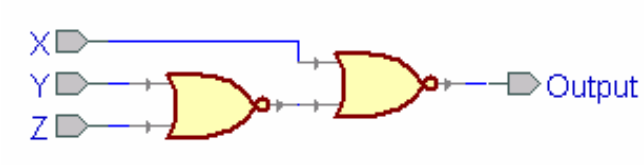
(e)



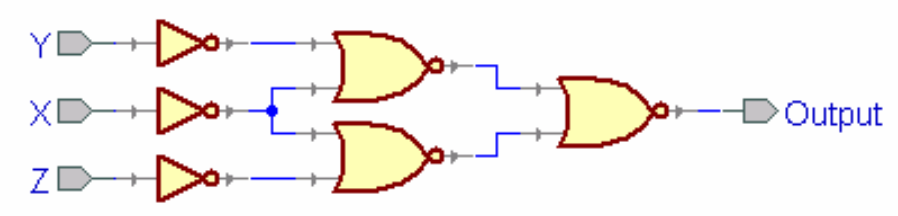
Exercise 2.3

Each of the formulas for both parts converts almost directly into logic gates.

(a)

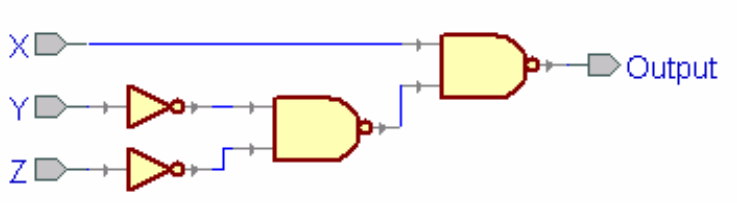


(b)

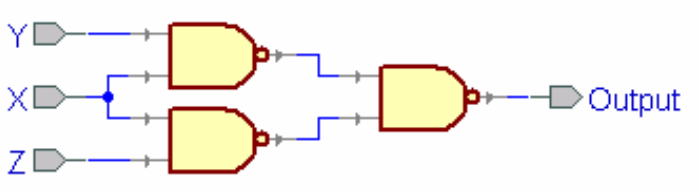


Exercise 2.4

(a) The formula converts easily to NAND and NOT gates.



(b) This solution uses DeMorgan's law to show that [[XY]' [XZ]'] ' = XY + XZ



Exercise 2.5

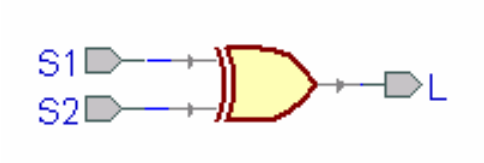
This question asks for two inputs and one output. Let the input S1 correspond to one switch, S2 correspond to the other switch, and let L correspond to the light.

Assume that when the light is installed both S1 and S2 are in the off position when L is in the off position.

What this gives is the following truth table for the circuit:

S1	S2	L
0	0	0
0	1	1
1	0	1
1	1	0

By inspection of the truth table, this function can be realized using an XOR gate as follows:



Exercise 2.6

Paragraphs that apply to the entire problem (or a single part problem) are not indented. This paragraph is "MainBody", while lettered paragraphs below are "IndentedBody". The title at the top is format "Heading".

(a) (X + Y)(X + Y') = X

Using 8D: X + (YY') = X
Using 5D: X + 0 = X
Using 1D: X = X

(b) X (X + Y) = X

Using 8: XX + XY = X
Using 3D: X + XY = X
Using 8: X (1 + Y) = X
Using 2: X (1) = X
Using 1D: X = X

(c) (X + Y') Y = XY

Using 8: XY + YY' = XY
Using 5D: XY + 0 = XY
Using 1: XY = XY

(d) (X + Y)(X' + Z) = XZ + X'Y

Using 8: (X + Y) X' + (X + Y) Z = XZ + X'Y
Using 8: XX' + YX' + XZ + YZ = XZ + X'Y
Using 5D: 0 + YX' + XZ + YZ = XZ + X'Y
Using 1: X'Y + XZ + YZ = XZ + X'Y
Using 3: X'Y (1) + XZ (1) + YZ (1) = XZ + X'Y
Using 5: X'Y (Z + Z') + XZ (Y + Y') + YZ (X + X') = XZ + X'Y
Using 8: X'YZ + X'YZ' + XYZ + XY'Z + XYZ + X'YZ = XZ + X'Y
Using 3: X'YZ + X'YZ' + XYZ + XY'Z = XZ + X'Y
Using 8: X'Y (Z + Z') + XZ (Y + Y') = XZ + X'Y
Using 5 and 1D: X'Y + XZ = XZ + X'Y

Exercise 2.7

(a) By definition: $(X + Y)^D = XY$

By definition: $(XY)^D = X + Y$

(b) Using 12: $[(X + Y)']^D = [X'Y']^D$

Using part (a): $[X'Y']^D = X' + Y'$

Using 12D: $X' + Y' = (XY)'$

This that NOR and NAND are duals of each other, since each function has one dual by definition, showing the direction from NOR to NAND is sufficient.

(c) Using part (a): $[XY' + X'Y]^D = (X + Y')(X' + Y)$

Using 8: $(X + Y')(X' + Y) = X'(X + Y') + Y (X + Y')$
Using 8: $X'(X + Y') + Y (X + Y') = XX' + X'Y' + XY + YY'$
Using 5D: $XX' + X'Y' + XY + YY' = X'Y' + XY$

The first step may appear kind of confusing since it uses the fact that AND and OR are duals three times. Again since each function has only one dual, it is sufficient to show XNOR is the dual of XOR to get XOR is the dual of XNOR.

(d) $(XY' + X'Y)' = (X'Y' + XY)$

Using 12: $(XY')' (X'Y)' = (X'Y' + XY)$
Using 12D: $(X' + Y) (X + Y') = X'Y' + XY$
Using 8: $(X' + Y) X + (X' + Y) Y' = X'Y' + XY$
Using 8: $X'X + XY + X'Y' + YY' = X'Y' + XY$
Using 5D: $X'Y' + XY = X'Y' + XY$

Exercise 2.8

The objective for proving using the truth tables is to show that when you apply the functions to every possible input, they agree on all of the inputs. This is the case in all of the parts below.

(a)

X	Y	Z	XY + YZ + XZ'	YZ + XZ'
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

(b)

A	B	(A + B') B	AB
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

(c)

A	B	C	(A + B) (A' + C)	AC + A'B
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

(d)

A	B	C	ABC + A'BC + A'B'C + A'BC' + A'B'C'	BC + A'B' + A'C'
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Exercise 2.9

	BC + A'B' + A'C' = ABC + A'
Using 5:	(A + A') BC + A'B' + A'C' = ABC + A'
Using 8:	ABC + A'BC + A'B' + A'C' = ABC + A'
Using 8:	ABC + A' (B' + C' + BC) = ABC + A'
Using 5:	ABC + A' (B'(C + C') + C' (B + B') + BC) = ABC + A'
Using 8:	ABC + A' (B'C + B'C' + BC' + B'C' + BC) = ABC + A'
Using 8:	ABC + A' [B (C + C') + B' (C + C')] = ABC + A'
Using 5:	ABC + A' [B + B'] = ABC + A'
Using 5:	ABC + A' = ABC + A'

Exercise 2.10

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

(b) [ABC + B (C’ + D’)] ’
= (ABC)’ [B (C’ + D’)]’
= (A’ + B’ + C’) [B’ + (C’ + D’)’]
= (A’ + B’ + C’) [B’ + CD]

(c) (X’ + Y’)’ = XY

(d) (X + YZ’)’
= X’ (YZ’)’
= X’ (Y’ + Z)

(e) [(X + Y) Z]’
= (X + Y)’ + Z’
= X’Y’ + Z’

(f) [X + (YZ)’]’ = X’YZ

(g) [X (Y + ZW’ + V’S)]’
= X’ + (Y + ZW’ + V’S’)’
= X’ + Y’ (ZW’)’ (V’S)’
= X’ + Y’ (Z’ + W) (V + S’)

Exercise 2.11

Note these solutions only take the complements, and do not simplify any further than using DeMorgan's laws.

(a) f (A, B, C, D) = [A + (BCD)'][(AD)' + B (C' + A)]

f ' (A, B, C, D) = [[A + (BCD)'][(AD)' + B (C' + A)]]'

= [A + (BCD)']' + [(AD)' + B (C' + A)]'

= A'BCD + AD [B (C' + A)]'

= A'BCD + AD [B' + (C' + A)']

= A'BCD + AD [B' + A'C]

(b) f (A, B, C, D) = A'BC + (A' + B + D)(ABD' + B')

f ' (A, B, C, D) = [A'BC + (A' + B + D)(ABD' + B')]'

= (A'BC)' [(A' + B + D)(ABD' + B')]'

= (A + B' + C')[(A' + B + D)' + (ABD' + B')']

= (A + B' + C')[AB'D' + B (ABD')']

= (A + B' + C')[AB'D' + B (A' + B' + D)']

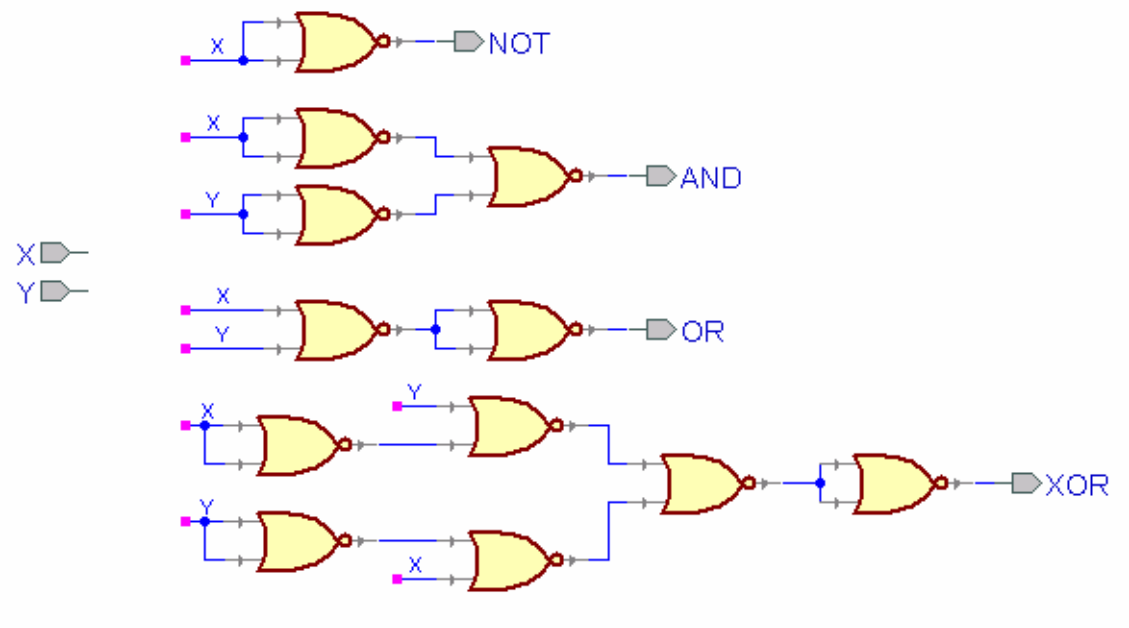
Exercise 2.12

[[X (XY)’]’ [Y (XY)’]’]’ = XY’ + X’Y

Using 12: [X (XY)’] + [Y (XY)’] = XY’ + X’Y
Using 12D: [X (X’ + Y’)] + [Y (X’ + Y’)] = XY’ + X’Y
Using 11: XY’ + X’Y = XY’ + X’Y

Exercise 2.13

The following figure demonstrates the implementation of the NOT, AND, OR, and XOR gates using the two-input NOR gate. The basis for many of these functions uses a combination of DeMorgan's laws and other Boolean Algebra simplifications.



Similarly NAND is a universal logic element since NAND and NOR are duals of each other, any function that NOR implements, NAND can implement the dual of that function. Since XOR and XNOR are complements of each other, combining NOT and XNOR can be implemented with NAND gates.

XOR is not a universal logic element, since it cannot implement NOT just using the inputs X and Y without leaving an input disconnected.

Exercise 2.14

Here define H_{1S} to be the sum output and H_{1C} to be the carryout of the first half adder that takes inputs A and B. H_{2S} is the sum output and H_{2C} is the carryout of the second half adder that takes H_{2S} and C_{in} as inputs. The output of H_{2S} is equivalent to S for the full adder, and OR is equivalent to C_{out} of the full adder.

A	B	C_{in}	H_{1S}	H_{1C}	H_{2S}	H_{2C}	OR	S	C_{out}
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	1	0
0	1	0	1	0	1	0	0	1	0
0	1	1	1	0	0	1	1	1	1
1	0	0	1	0	1	0	0	1	0
1	0	1	1	0	0	1	1	0	1
1	1	0	0	1	0	0	1	0	1
1	1	1	0	1	1	0	1	1	1

Exercise 2.15

The waveform behavior will be different because in the direct implementation from Boolean equations the equations can be simplified to contain at most two gate delays assuming that there are no limitations on the fanout of each logic gate. In the case of the full adder, it takes two gate delays for each half-adder, plus an additional gate delay for the OR-gate. Thus the direct equations will lead to a much faster circuit than the hierarchical form.

Exercise 2.16

Let the variables A_0, B_0 represent the least significant bit of the two bit inputs and A and B, and let A_1, B_1 be the most significant bits respectively.

$S_0 = A_0B_0' + A_0'B_0$

$C_{out0} = A_0B_0$

$S_1 = A_1B_1C_{out0} + A_1B_1'C_{out0}' + A_1'B_1C_{out0}' + A_1'B_1'C_{out0}$

$= A_1B_1\ A_0B_0 + A_1B_1'\ (\ A_0B_0\ )' + A_1'B_1\ (\ A_0B_0\ )' + A_1'B_1'A_0B_0$

$= A_1B_1\ A_0B_0\ + A_1B_1'\ (\ A_0'+B_0'\ ) + A_1'B_1(\ A_0' + B_0'\ )' + A_1'B_1'A_0B_0$

$= A_0B_0A_1B_1 + A_0'A_1B_1' + B_0'A_1B_1' + A_0'A_1'B_1 + B_0'A_1'B_1 + A_0B_0A_1'B_1'$

$C_{out1} = B_1C_{out0} + A_1C_{out0} + A_1B_1$

$= B_1A_0B_0 + A_1A_0B_0 + A_1B_1$

Exercise 2.17

(a) f (X, Y) = XY + XY'

Using 8: XY + XY' = X (Y + Y')
Using 5: XY + XY' = X

(b) f (X, Y) = (X + Y)(X + Y')

Using 8: f (X, Y) = X + XY' + XY + YY'
Using 5D: f (X, Y) = X + XY' + XY
Using 8: f (X, Y) = X + X (Y' + Y)
Using 5: f (X, Y) = X + X
Using 3: f (X, Y) = X

(c) f (X, Y, Z) = Y'Z + X'YZ + XYZ

Using 8: f (X, Y, Z) = Z (Y' + X'Y + XY)
Using 8: f (X, Y, Z) = Z (Y' + Y (X' + X))
Using 5: f (X, Y, Z) = Z (Y' + Y)
Using 5: f (X, Y, Z) = Z

(d) f (X, Y, Z) = (X + Y) (X' + Y + Z) (X' + Y + Z')

Using 8: f (X, Y, Z) = (XX' + XY + XZ + X'Y + Y + YZ) (X' + Y + Z')
Using 5D and 10: f (X, Y, Z) = (XZ + Y) (X' + Y + Z')
Using 8: f (X, Y, Z) = XX'Z + XYZ + XZZ' + X'Y + Y + YZ'
Using 5D and 2D: f (X, Y, Z) = XYZ + X'Y + Y + YZ'
Using 10: f (X, Y, Z) = Y

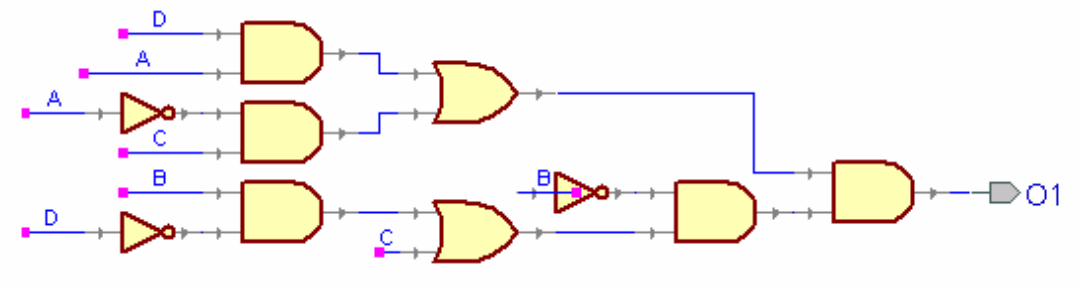
(e) f (W, X, Y, Z) = X + XYZ + X'YZ + X'Y + WX + WX'

Using 8: f (W, X, Y, Z) = X (1 + W) + YZ (X + X') + X'Y + X'W
Using 2: f (W, X, Y, Z) = X + YZ (X + X') + X'Y + X'W
Using 5: f (W, X, Y, Z) = X + YZ + X'Y + X'W

Exercise 2.18

Paragraphs that apply to the entire problem (or a single part problem) are not indented. This paragraph is “MainBody”, while lettered paragraphs below are “IndentedBody”. The title at the top is format “Heading”.

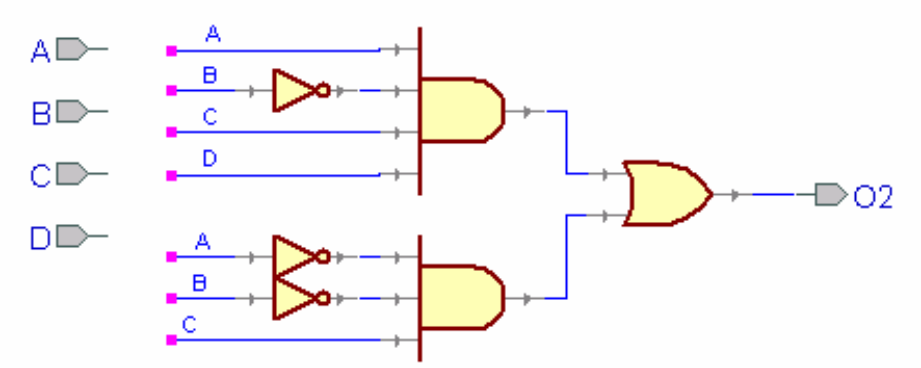
(a)



(b) Using 8: (AD + A’C)[B’ (C + BD’)] = (AD + A’C) [B’C + B’BD’]

Using 3: (AD + A’C)[B’C + B’BD] = (AD + A’C) B’C

Using 8: (AD + A’C) B’C = AB’CD + A’B’C



Exercise 2.19

Each of the following solutions is put into their respected canonical form.

(a) A’B’C’D’ + A’B’C’D + A’B’CD’ + A’BCD + AB’C’D’ + AB’C’D + AB’CD’ + ABCD

(b) (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’)

(c) A’B’CD + A’BC’D’ + A’BC’D + A’BCD’ + AB’CD + ABC’D’+ ABC’D + ABCD’

(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’) (A + B + C + D)

Exercise 2.20

Each of the following solutions is put into their respected canonical form.

(a) A’B’C’D + A’B’CD’ + A’B’CD + A’BC’D + AB’C’D’ + ABC’D

(b) (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’) (A + B + C + D’) (A + B + C + D)

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

(d) (A’ + B’ + C’ + D) (A’ + B’ + C + D’) (A’ + B’ + C + D) (A’ + B + C’ + D) (A + B + C’ + D)

Exercise 2.21

(a) f (A, B, C) = AB + B'C' + AC'

= ABC' + ABC + A'B'C' + AB'C' + AB'C' + ABC'

= A'B'C' + AB'C' + ABC' + ABC

= $\sum$ m(0, 4, 6, 7)

(b) M(0, 4, 6, 7) = (A' + B' + C') (A + B' + C') (A + B + C') (A + B + C)

Exercise 2.22

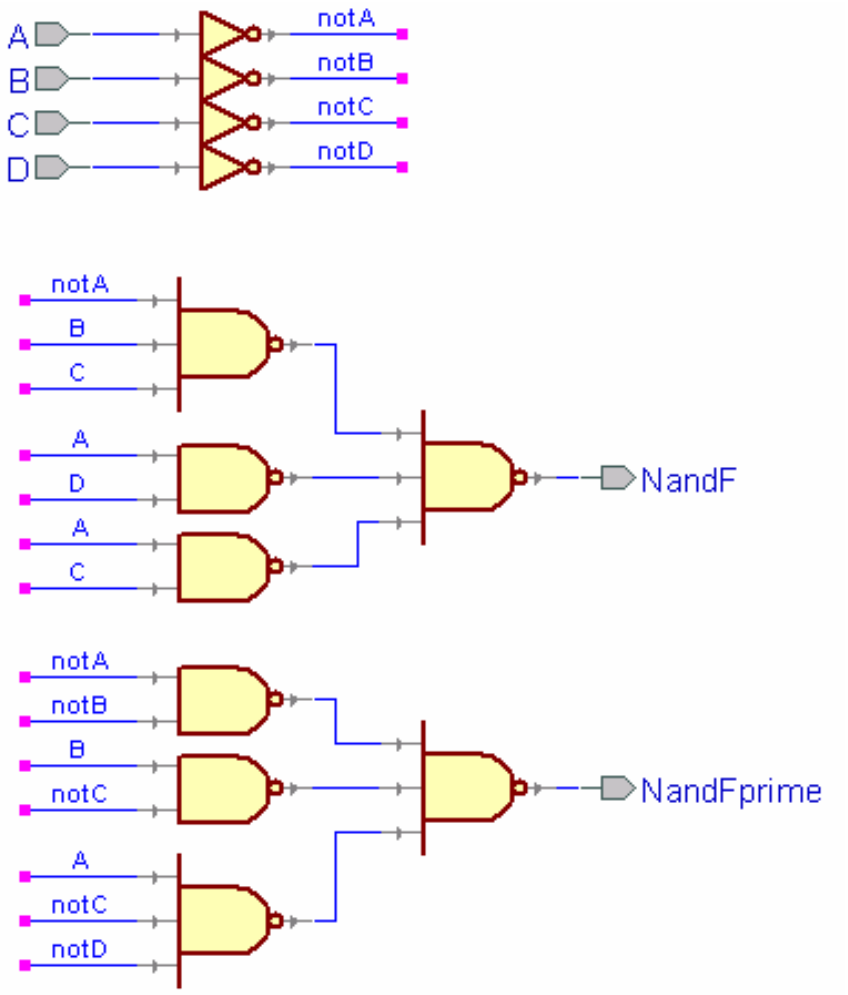
(a) $F(A, B, C, D) = \prod M(0, 1, 2, 3, 4, 5, 8, 12)$

(b) $F(A, B, C, D) = (A' + B' + C' + D') (A' + B' + C' + D) (A' + B' + C + D')$
$(A' + B' + C + D) (A' + B + C' + D') (A' + B + C' + D)$
$(A + B' + C' + D')(A + B + C' + D')$
$= (A'+B'+C')(A'+B'+C)(A'+B+C')(A+C'+D')$
$= (A'+B')(A'+B+C')(A+C'+D')$

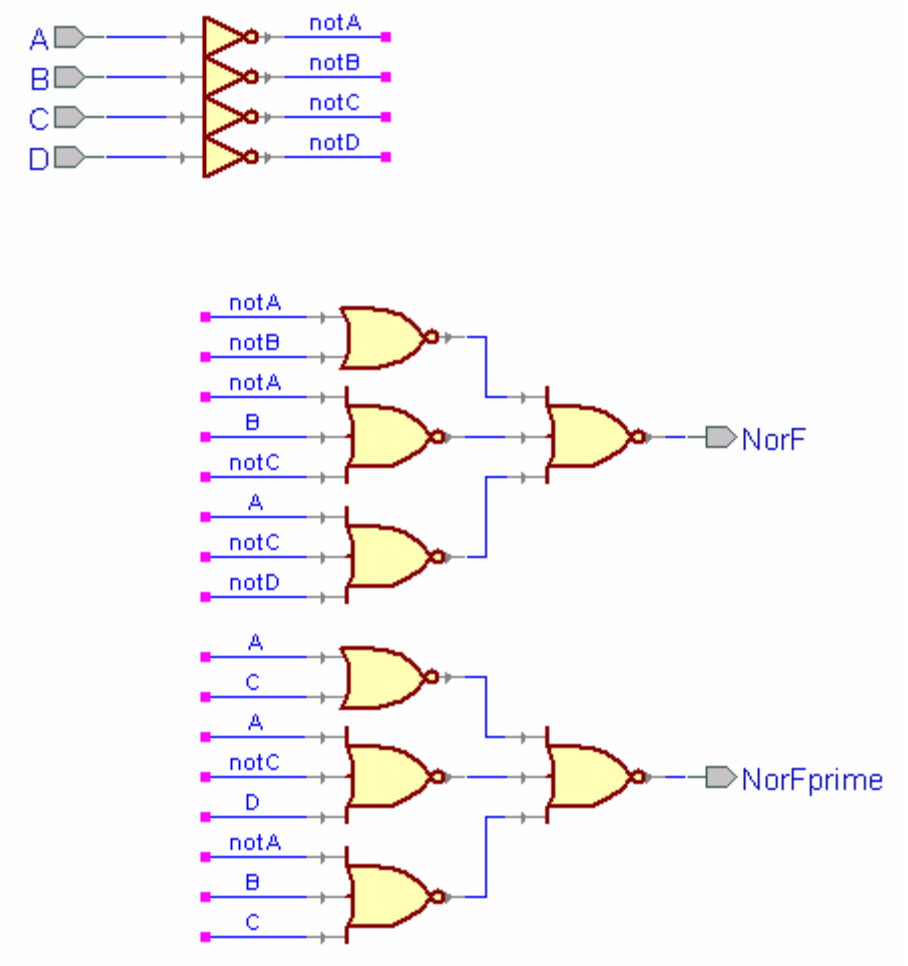
(c) $F'(A, B, C, D) = \prod M(6, 7, 9, 10, 11, 13, 14, 15)$
$= (A' + B + C + D') (A' + B + C + D) (A + B' + C' + D)$
$(A + B' + C + D') (A + B' + C + D) (A + B + C' + D)$
$(A + B + C + D')(A + B + C + D)$
$= (A' + B + C)(A + C' + D) (A + C)$

(d) $F'(A,B,C,D) = \sum M(0, 1, 2, 3, 4, 5, 8, 12)$
$= A'B'C'D' + A'B'C'D + A'B'CD' + A'B'CD + A'BC'D' + A'BC'D$
$+ AB'C'D' + ABC'D'$
$= A'B'C' + A'B'C + A'BC' + AC'D'$
$= A'B' + BC' + AC'D'$

(e) Note wires with the same name are considered to be connected to each other. Using the canonical sum-of-products formula and DeMorgen's laws, the transformation into NAND gates is fairly straightforward.



(f) Using the product-of-sums form, transformation to NOR gates is fairly straightforward.



Exercise 2.23

(a) Since the function is given in minimized product-of-sums form, it is easiest to convert it first to canonical product-of-sums form. By taking the complement of the canonical form, and converting the $\prod$M to $\sum$m the canonical sum-of-products is reached.

F(W, X, Y, Z) = (W + X' + Y') (W' + Z') (W + Y)

= (W' + X' + Y' + Z') (W' + X' + Y + Z') (W' + X + Y' + Z')
(W' + X + Y + Z') (W + X' + Y' + Z') (W + X' + Y' + Z)
(W + X' + Y + Z') (W + X' + Y + Z) (W + X + Y + Z')
(W + X + Y + Z)

= $\prod$M (0, 2, 4, 6, 8, 9, 10, 11, 14, 15)

= $\sum$m (1, 3, 5, 7, 12, 13)

(b) $\sum$m (1, 3, 5, 7, 12, 13) = W'X'Y'Z + W'X'YZ + W'XY'Z + W'XYZ + WXY'Z' + WXY'Z

= W'X'Z + W'XZ + WXY'

= W'Z + WXY'

(c) $\sum$m(0, 2, 4, 6, 8, 9, 10, 11, 14, 15) = W'X'Y'Z' + W'X'YZ' + W'XY'Z' + W'XYZ'+ WX'Y'Z' + WX'Y'Z + WX'YZ' + WX'YZ + WXYZ' + WXYZ

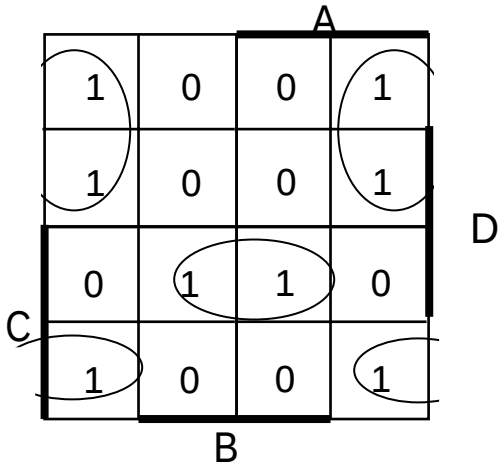
= W'X'Z' + W'XZ' + WX'Y' + WX'Y + WXY

= W'Z' + WX' + WY

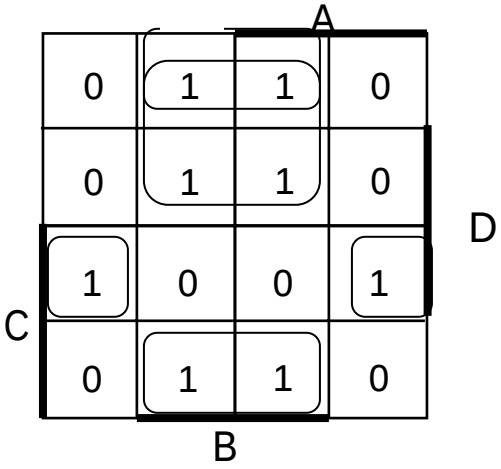
(d) $\prod$M (1, 3, 5, 7, 12, 13) = (W' + X' + Y' + Z) (W' + X' + Y + Z)
(W' + X + Y' + Z) (W' + X + Y + Z) (W+ X + Y' + Z')
(W + X + Y' + Z)

Exercise 2.24

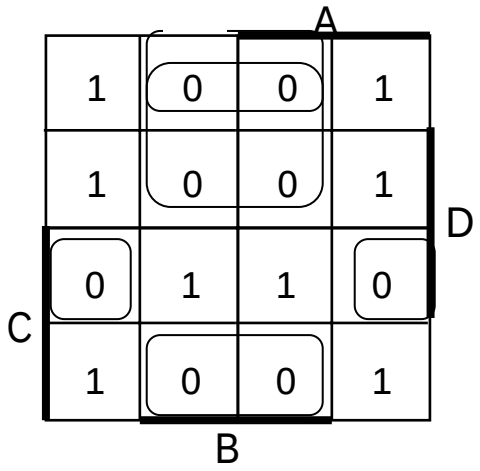
(a) F(A, B, C, D) = B'C' + BCD + B'CD'



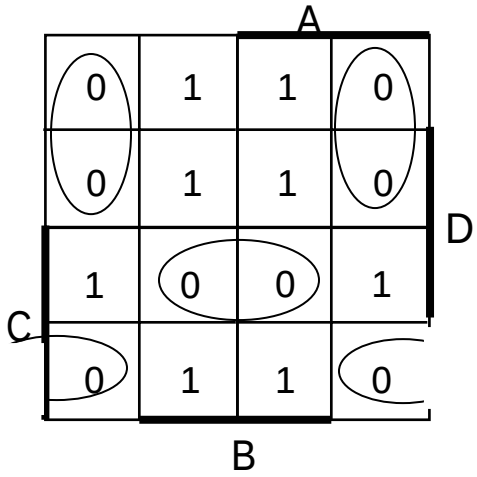
F'(A, B, C, D) = BD' + BC' + B'CD



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



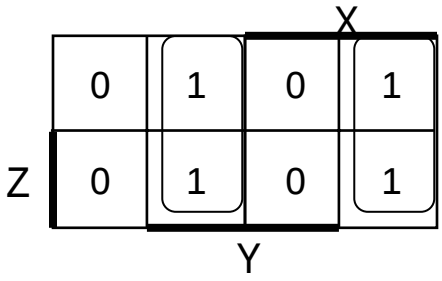
F'(A, B, C, D) = (B' + C') (B + C + D) (B' + C + D')



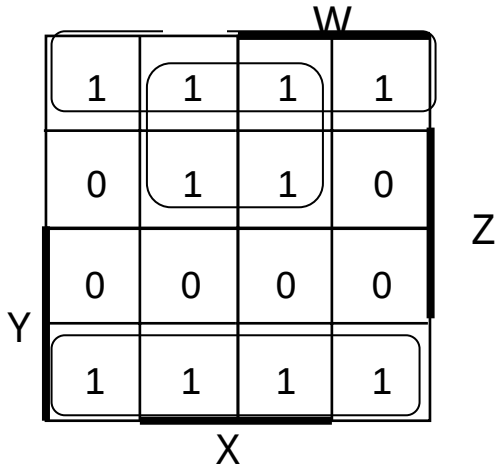
Exercise 2.25

Paragraphs that apply to the entire problem (or a single part problem) are not indented. This paragraph is “MainBody”, while lettered paragraphs below are “IndentedBody”. The title at the top is format “Heading”.

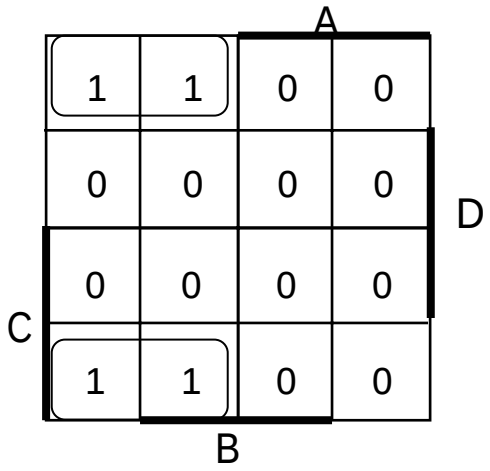
(a) f(X, Y, Z) = AB + AB’



(b) f(W, X, Y, Z) = Z’ + XY’

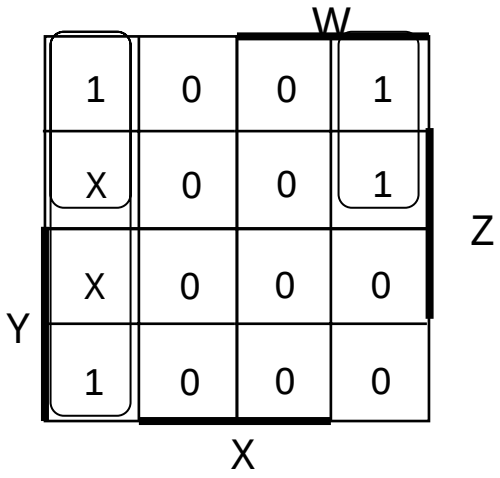


(c) f(A, B, C, D) = A’D’

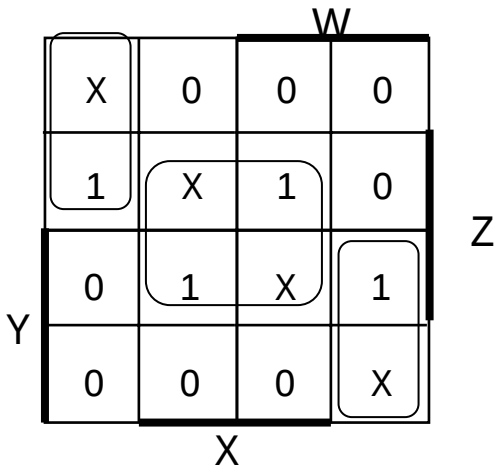


Exercise 2.26

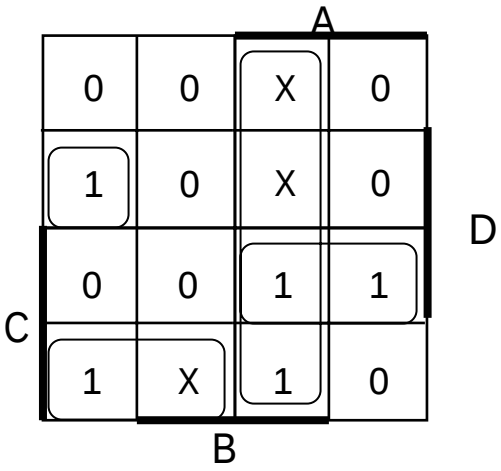
(a) f(W, X, Y, Z) = X'W' + X'Y'



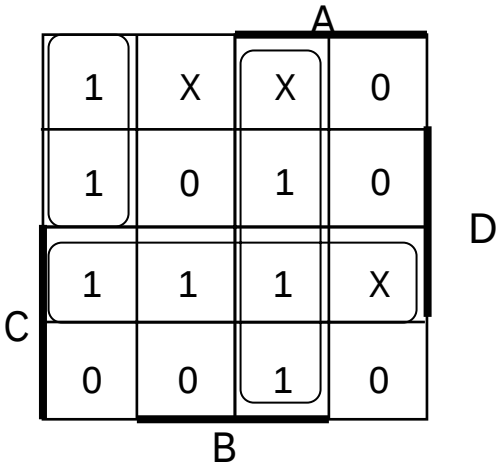
(b) f(W, X, Y, Z) = XZ + WX'Y + W'X'Y'



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

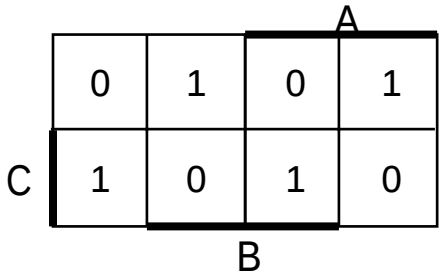


(d) f(A, B, C, D) = A’B’C’ + CD + AB

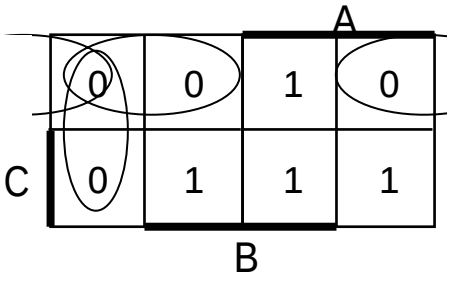


Exercise 2.27

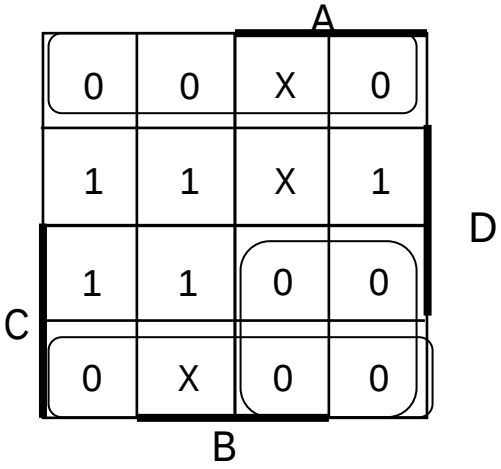
(a) f(A, B, C) = (A’ + B’ + C’) (A’ + B + C) (A + B + C) (A + B’ + C)



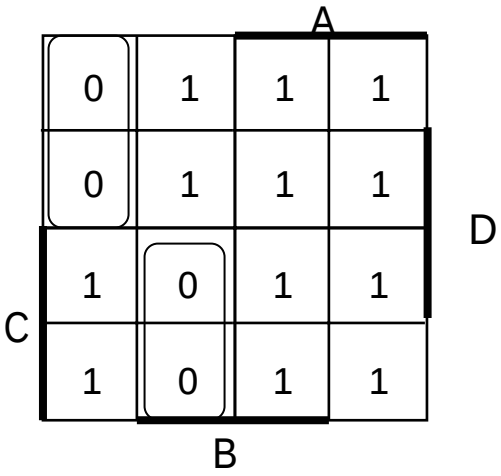
(b) f(A, B, C) = (A’ + B’) (A’ + C’) (A + B’)



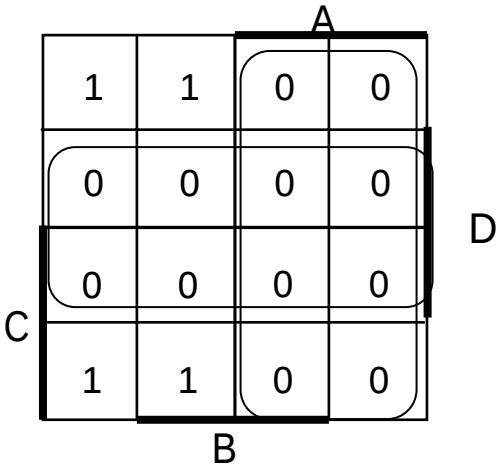
(c) f(A, B, C, D) = D’ (A + C)



(d) f(A, B, C, D) = (A’ + B + C) (A’ + B’ + C’)



(e) f(A, B, C, D) = AD



Exercise 2.28

(a) In this case S cannot be simplified any further.
S(A, B, C) = A'B'C + A'BC' + AB'C' + ABC

(b) F(A, B, C) = A'B'C' + A'B'C + AB'C' + AB'C + ABC' + ABC

Using 8: = A'B'(C' + C) + AB' (C' + C) + AB (C' + C)
Using 5: = A'B' + AB' + AB
Using 3: = A'B' + AB' + AB' + AB
Using 8: = (A' + A) B' + A (B' + B)
Using 5: = B' + A

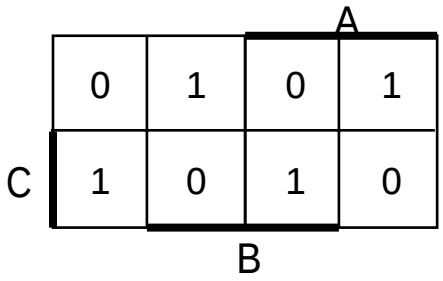
(c) G(A, B, C, D) = A'B'C'D' + A'B'CD' + AB'C'D' + AB'CD + ABC'D' + ABCD

Using 8: = A'B'(C' + C)D' + A(B + B')C'D' + A(B + B')CD
Using 5: = A'B'D' + AC'D' + ACD

Exercise 2.29

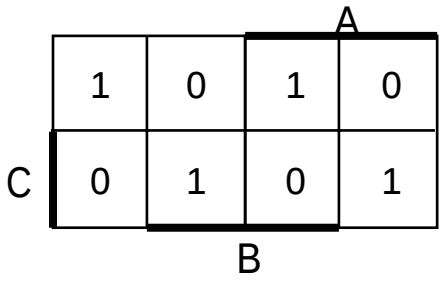
Solution for part 2.28 part (a). Note since neither the function nor its complement can be simplified in this case, the Karnaugh maps will be exactly the same for (a) and (b), and also for (c) and (d). Because of this, only the maps for part (a) and (c) are shown.

(a) S(A, B, C) = A'B'C + AB'C + ABC + AB'C'



(b) S(A, B, C) = (A' + B' + C') (A' + B + C) (A + B + C) (A + B' + C)

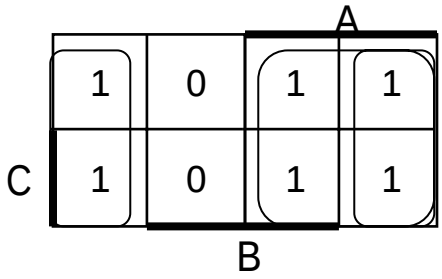
(c) S'(A, B, C) = A'B'C' + A'BC + ABC' + AB'C



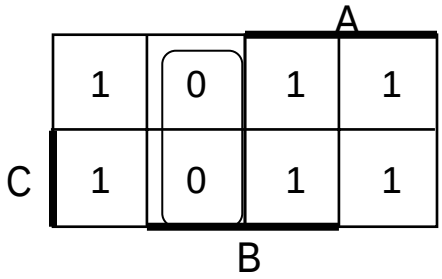
(d) S'(A, B, C) = (A' + B' + C) (A' + B + C') (A + B + C) (A + B' + C')

Solution for part 2.28 part (b):

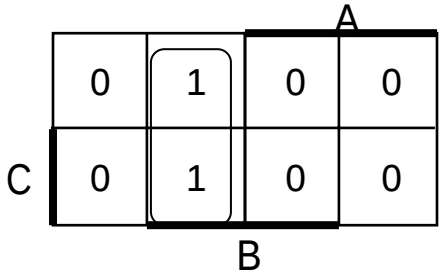
(a) F(A, B, C) = B’ + A



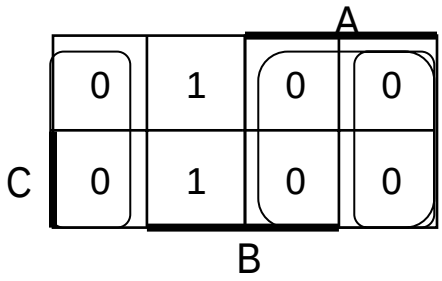
(b) F(A, B, C) = BA’



(c) F’(A, B, C) = B + A’

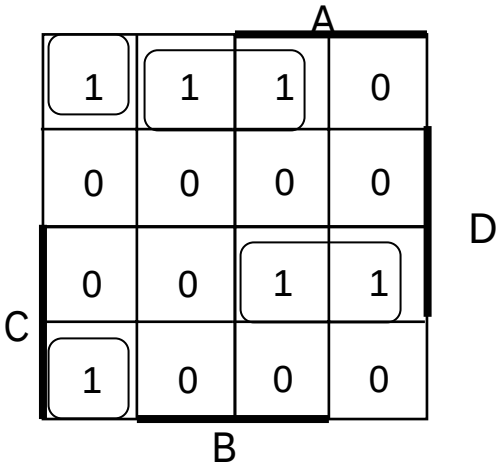


(d) F’(A, B, C) = AB’

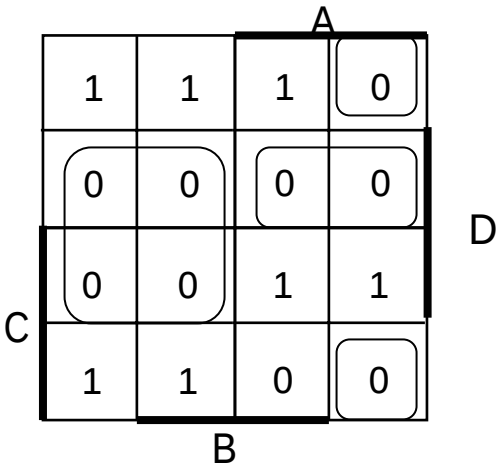


Solution for part 2.28 part (c):

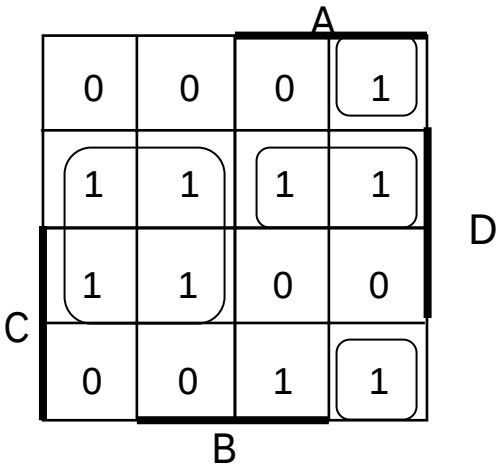
(a) G(A, B, C, D) = A'BD' + BC'D' + ACD



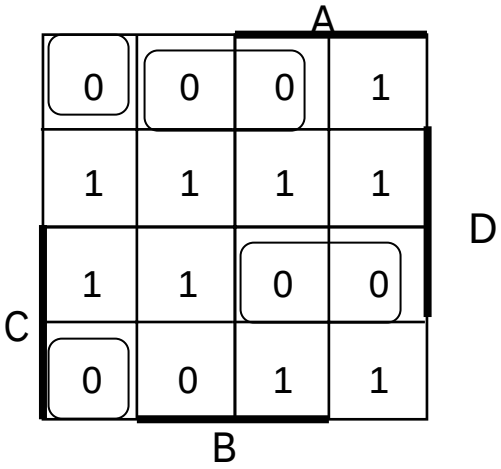
(b) G(A, B, C, D) = (A' + D) (A + C' + D) (A + B' + D')



(c) G’(A, B, C, D) = A’D + AC’D + AB’C’



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



Exercise 2.30

(a) W(A, B, C) = A'BC' + A'BC + AB'C' + AB'C

Using 8: = A'B(C' + C) + AB' (C + C')
Using 5: = A'B + AB' = A ⊗ B

(b) X(A, B, C) = A'B'C' + A'BC + AB'C' + ABC

Using 8: = (A' + A) B'C' + (A' + A) BC
Using 5: = B'C' + BC

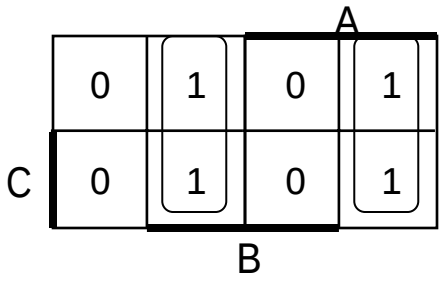
(c) Y(A, B, C, D) = A'B'C'D' + A'B'C'D + A'B'CD' + A'B'CD + AB'C'D' + AB'CD'

Using 8: = A'B'C' (D + D') + AB' (C + C') D' + A'B'C (D + D')
Using 5: = A'B'C' + AB'D' + A'B'C
Using 8: = A'B'(C + C') + AB'D'
Using 5: = A'B' + AB'D'
Using 2: = A'B' (1 + D') + AB'D'
Using 8: = A'B' + A'B'D' + AB'D'
Using 8: = A'B' + (A' + A) B'D'
Using 5: = A'B' + B'D'

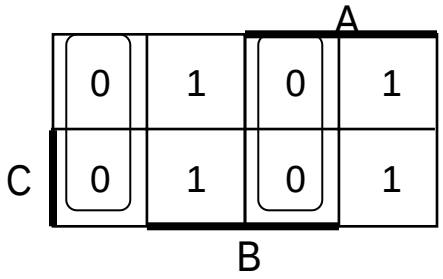
Exercise 2.31

Solution for part 2.30 part (a).

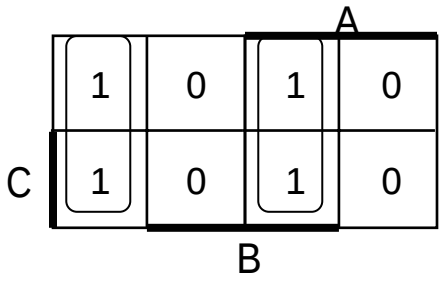
(a) W(A, B, C) = A'B + AB'



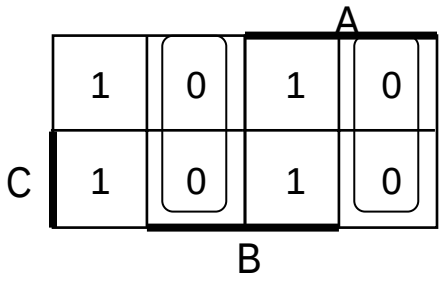
(b) W(A, B, C) = (A + B) (A' + B')



(c) W'(A, B, C) = A'B' + AB

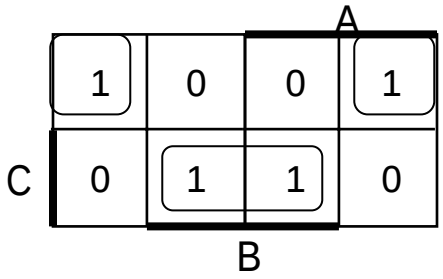


(d) W’(A, B, C) = (A’ + B’ + C) (A’ + B + C’) (A + B + C) (A + B’ + C’)

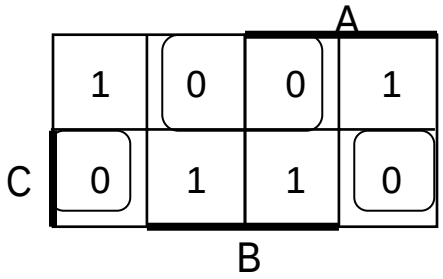


Solution for part 2.30 part (b).

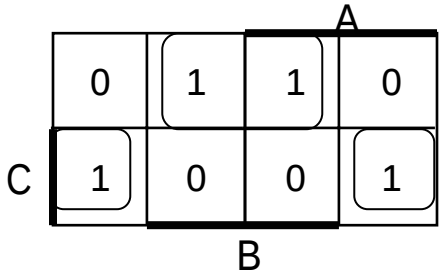
(a) X(A, B, C) = BC+B'C'



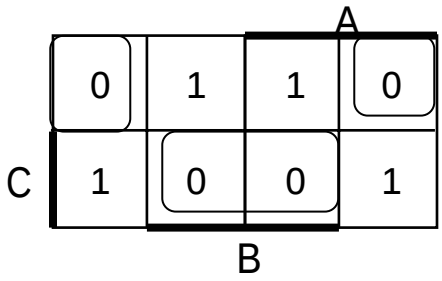
(b) X(A, B, C) = (B' + C) (B + C')



(c) X'(A, B, C) = B'C + BC'

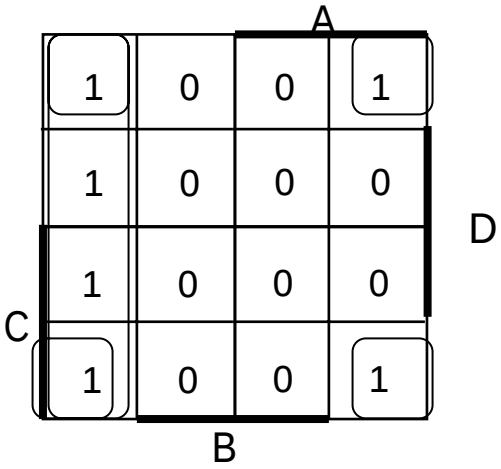


(d) X’(A, B, C) = (B + C) (B’ + C’)

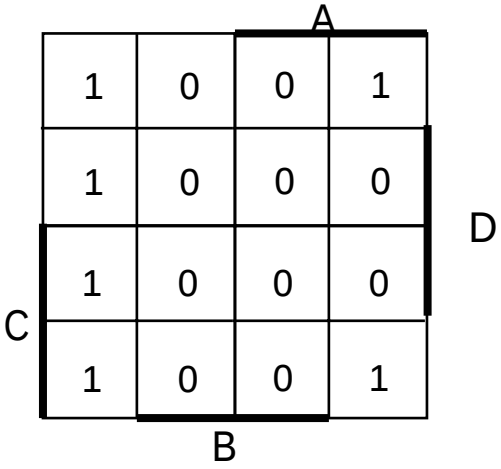


Solution for part 2.30 part (b).

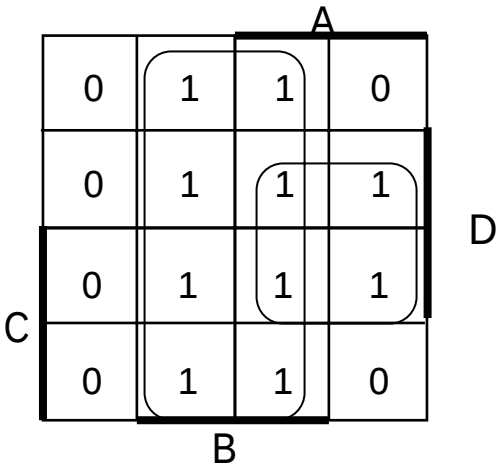
(a) Y(A, B, C, D) = A'B' + B'D'



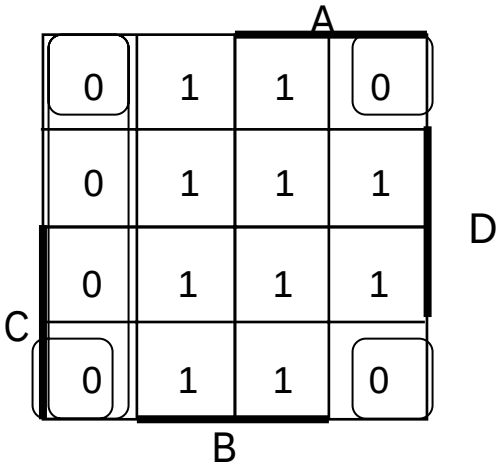
(b) Y(A, B, C, D) = B (A + D)



(c) Y’(A, B, C) = B + AD



(d) Y’(A, B, C) = (A’ + B’) (B’ + D’)



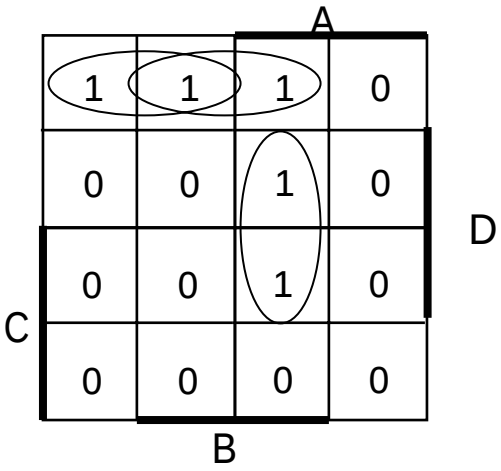
Exercise 2.32

The following two functions are equivalent as show by the K-maps below:

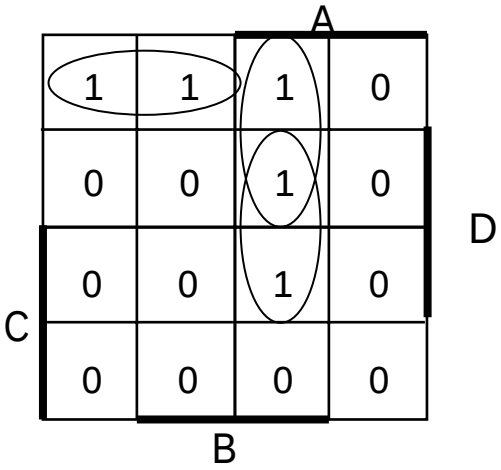
$$f_1(A, B, C, D) = ABD + BC'D' + A'C'D'$$

$$f_2(A, B, C, D) = ABD + ABC' + A'C'D'$$

f_1:

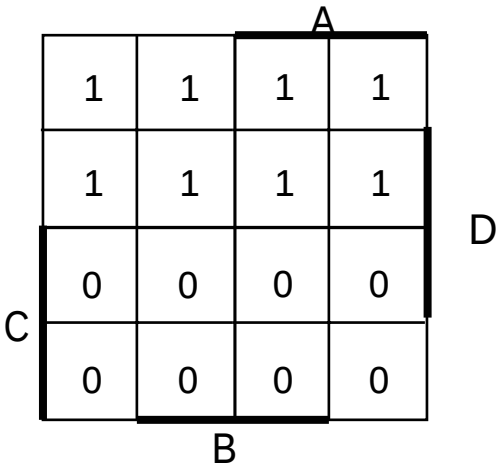


f_2:



Exercise 2.33

(a) The K-map below gives the sum-of-products form C' and product-of-sums form C, each of which has one literal and one term.



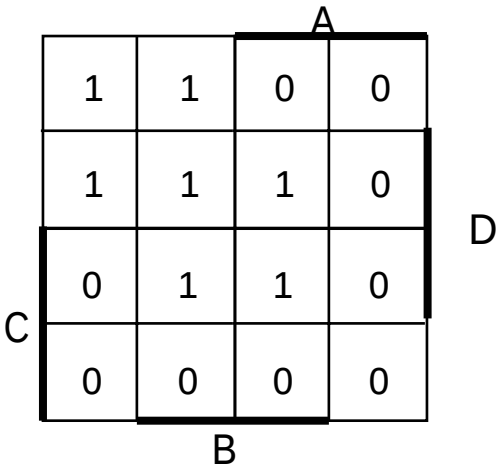
(b) The minimized sum-of-products for the following K-map is:

$$BD + A'C'$$

This equation has 4 literals and two terms. The minimized product-of-sums is:

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

This equation has eight literals and four terms.



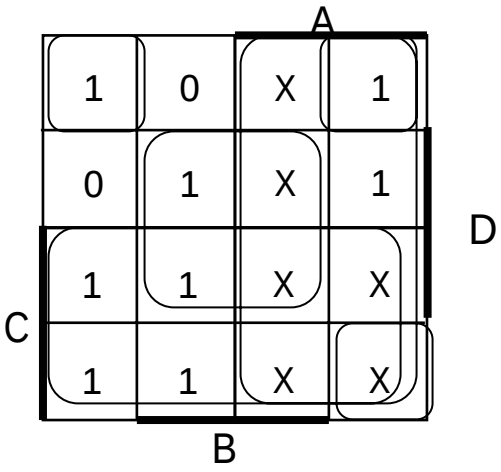
(c) By taking the complement of the previous K-map, the sum-of-products form now has eight literals and four terms, and the product-of-sums now has four literals and 2 terms:

Sum-of-products: CD’ + AB’ + B’C + AD’

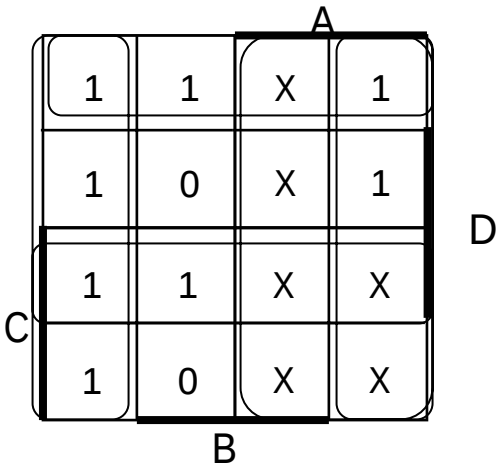
Product-of-sums: (B + D) (A’ + C’)

Exercise 2.34

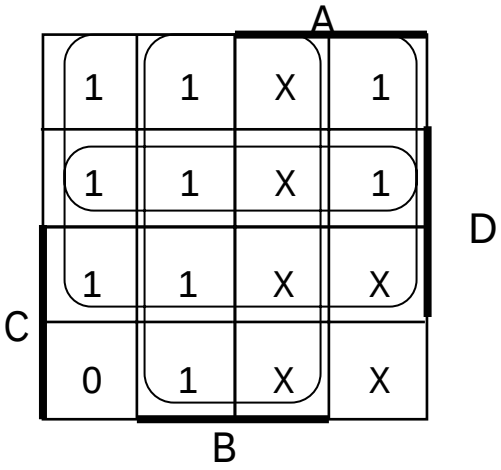
$C_0 = A + C + BD + B'D'$



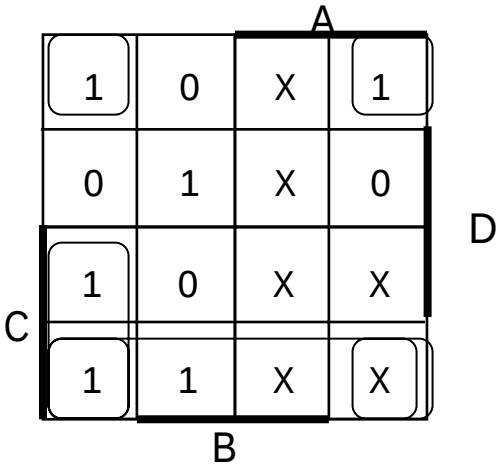
$C_1 = A + B' + C'D' + CD$



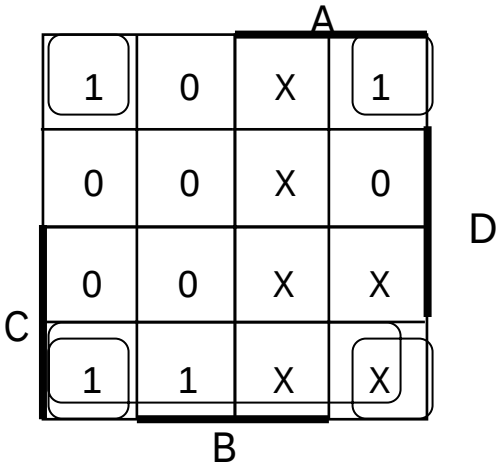
$C_2 = B + C' + D$



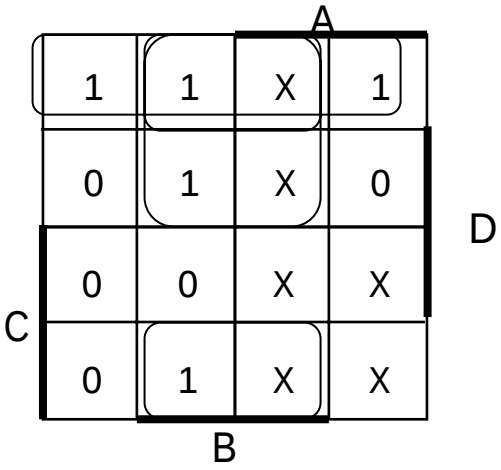
$C_3 = A'BC'D + B'D' + CD' + A'B'C$



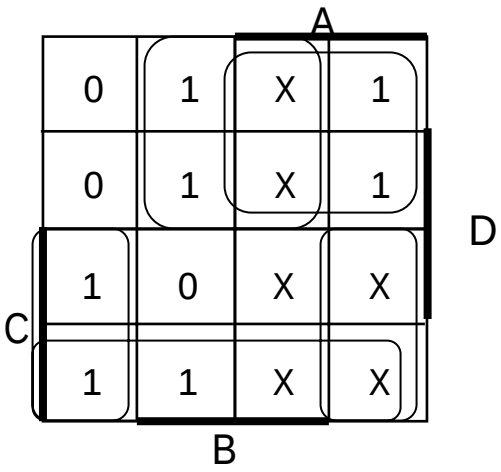
$C_4 = CD' + B'D'$



$C_5 = AC' + BD + BC'$



$C_6 = AC' + BC' + B'C + CD'$



Exercise 2.35

The truth table below provides an easy method for determining the canonical sum-of-products form:

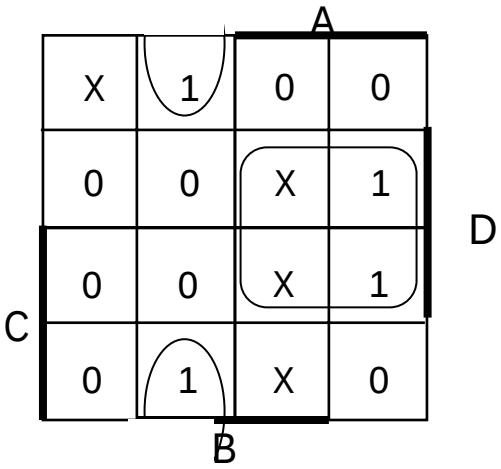
Month Code	d30	d31
0000	X	X
0001	0	1
0010	0	0
0011	0	1
0100	1	0
0101	0	1
0110	1	0
0111	0	1
1000	0	1
1001	1	0
1010	0	1
1011	1	0
1100	0	1
1101	X	X
111X	X	X

$$d30 = \sum m(4, 6, 9, 11) + \sum d(0, 13, 14, 15)$$

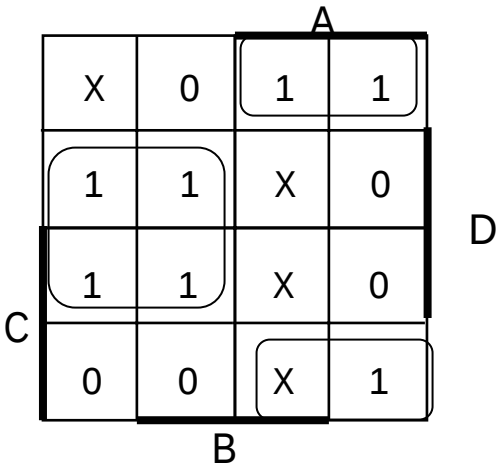
$$d31 = \sum m(1, 3, 5, 7, 8, 10, 12) + \sum d(0, 13, 14, 15)$$

These produce the following K-maps:

$$d30 = AD + A'BD'$$



d31 = A’D + AD’



Exercise 2.36

The truth table below provides an easy method for determining the canonical sum-of-products form (note the slight difference in encoding from the solution of Exercise 2.35):

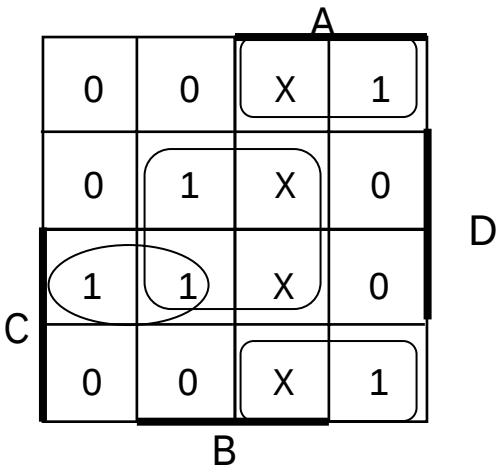
Month Code	d30	d31
0000	0	1
0001	0	0
0010	0	1
0011	1	0
0100	0	1
0101	1	0
0110	0	1
0111	0	1
1000	1	0
1001	0	1
1010	1	0
1011	0	1
11XX	X	X

$$d30 = \sum m(3, 5, 8, 10) + \sum d(12, 13, 14, 15)$$

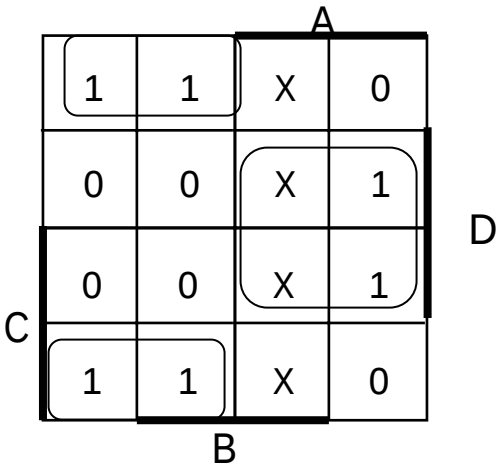
$$d31 = \sum m(0, 2, 4, 6, 7, 9, 11) + \sum d(12, 13, 14, 15)$$

These produce the following K-maps:

$$d30 = BD + AD' + A'CD$$



d31 = AD + A’D’



This minimized form turns out to be worse than the original encoding, since d30 now takes three terms instead of two, and two more literals. There isn’t much impact on d31 since it uses the same number of terms and literals with both encodings.

Exercise 2.37

Paragraphs that apply to the entire problem (or a single part problem) are not indented. This paragraph is “MainBody”, while lettered paragraphs below are “IndentedBody”. The title at the top is format “Heading”.

(a) $ABCD + ABDE = ABC (D + E)$

(b) $ACD + BC + ABE + BD = ACD + B (C + E + D)$

(c) $AC + ADE + BC + BDE = A (C + DE) + B (C + DE) = (A + B) (C + DE)$

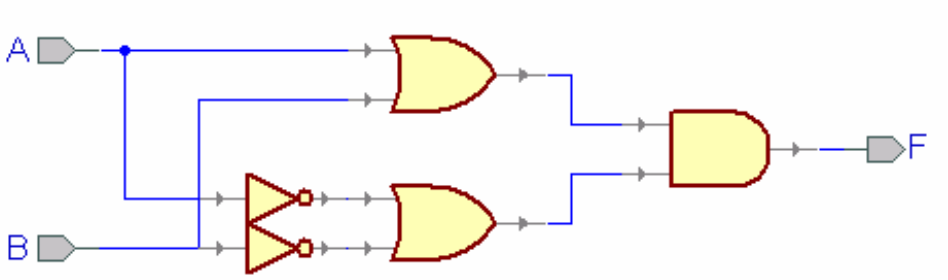
(d) $AD + AE + BD + BE + CD + CE + AF = (A + B + C) E + (A + B + C) D + AF$
$= (A + B + C) (E + D) + AF$

(e) $ACE + ACF + ADE + ADF + BCE + BCF + BDF$
$= (AC + AD + BC) E + (AC + AD + BC) F + BDF$
$= (AC + AD + BC) (E + F) + BDF$

Exercise 2.38

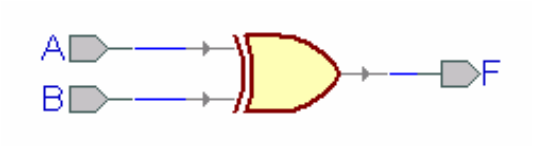
The following function implements the function in Figure Ex. 2.38 without using any NAND or NOR operations, as demonstrated by the figure.

$$F(A, B) = A' (A + B) + B' (A + B) = (A' + B') (A + B)$$



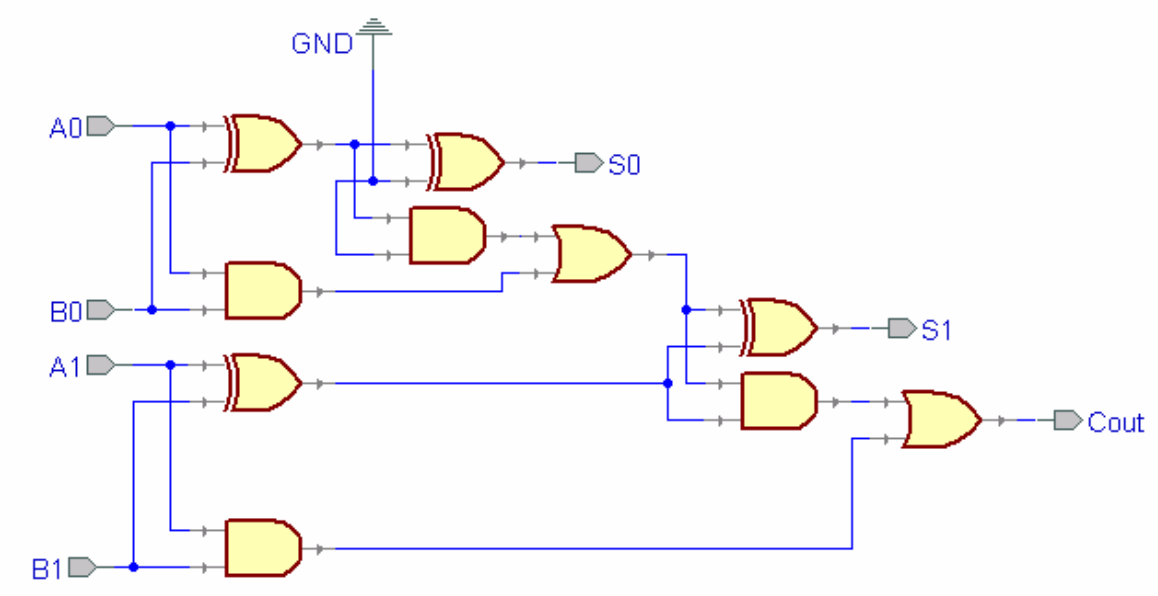
A minimized implementation using the fewest gates comes out to be the following function:

$$F(A, B) = (A' + B') (A + B) = (A'B + AB') = A \otimes B$$



Exercise 2.39

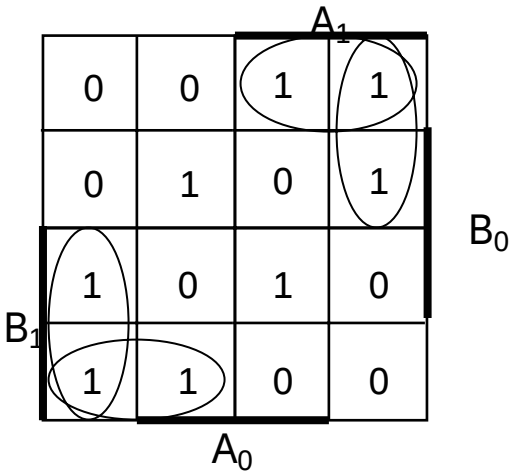
The method described in Figure 2.20 gives the following implementation of the full adder:



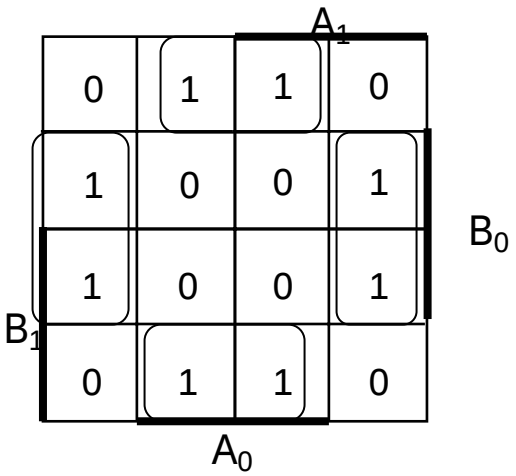
The truth table below is used to create K-maps, in order to create a minimized two level representation.

A1	A0	B1	B0	S1	S0	Cout
0	0	0	0	0	0	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	0	1	0
0	1	0	1	1	0	0
0	1	1	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	1	0
1	0	1	0	0	0	1
1	0	1	1	0	1	1
1	1	0	0	1	1	0
1	1	0	1	0	0	1
1	1	1	0	0	1	1
1	1	1	1	1	0	1

$S_1 = A_1\ A_0\ B_1\ B_0 + A_1'\ A_0\ B_1'\ B_0 + A_1\ B_1'\ B_0' + A_1\ A_0'\ B_1'$



$S_0 = A_0\ B_0' + A_0'\ B_0$



$C_{out} = A_1\ B_1 + A_1\ A_0\ B_0 + A_0\ B_1\ B_0$

Since the implementation of this will be fairly straightforward, a diagram is not being provided. The first implementation uses a variety of two-input gates including AND, OR, and XOR, whereas the second one uses a variety of AND and OR gates that can fanin up to 4 inputs. In the case of total number of gates the first implementation has ten gates, whereas the second implementation has eleven gates. The first implementation will have fewer wires, since factorization helps reduce the number of times the same solutions are computed. The second implementation will be faster though, since the worst case is through two gates with 4 inputs, whereas the first implementation has to travel through six gates along the worst path. The delays incurred from a larger fanin are not enough to cover the delays due to the 2 extra gates in this case.

Exercise 2.40

Paragraphs that apply to the entire problem (or a single part problem) are not indented. This paragraph is “MainBody”, while lettered paragraphs below are “IndentedBody”. The title at the top is format “Heading”.

(a)

SHIFT	i_0	i_1	o_0	o_1
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

(b)

IN	SELECT	o_0	o_1
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

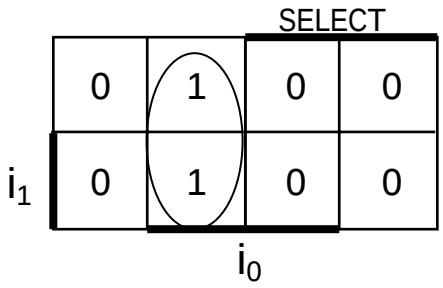
(c)

SELECT	i_0	i_1	OUT
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

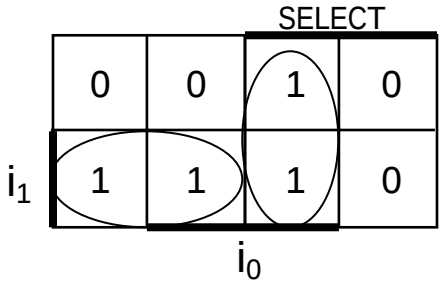
Exercise 2.41

Note: the solutions provided are dependent on the truth table in the answer to Exercise 2.40.

(a) $o_0 = \sum m(2, 3) = SELECT \bullet i_0$



$o_1 = \sum m(1, 3, 6, 7) = SELECT \bullet i_0 + SELECT' \bullet i_1$

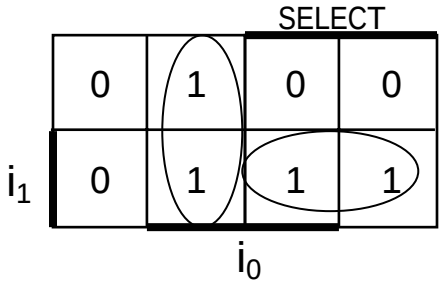


(b) $o_0 = \sum m(2) = SELECT' \bullet IN$

$o_1 = \sum m(3) = SELECT \bullet IN$

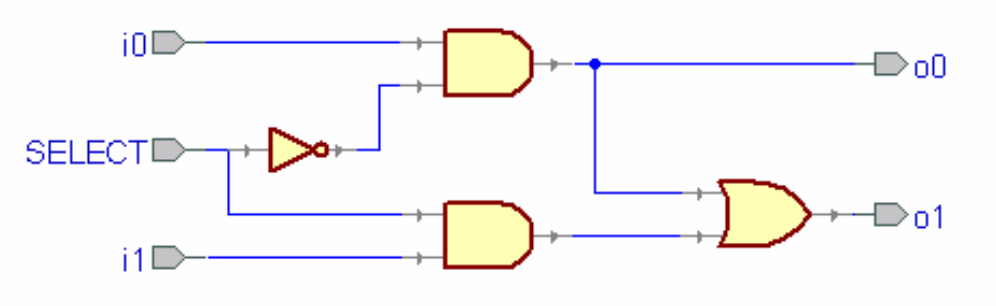
In this case it does not make sense to use a K-map for simplification since each function consists of only one term.

(c) $OUT = \sum m(2, 3, 5, 7) = SELECT' \bullet i_0 + SELECT \bullet i_1$

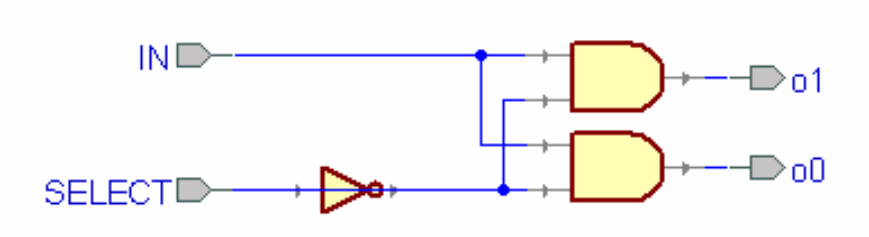


Exercise 2.42

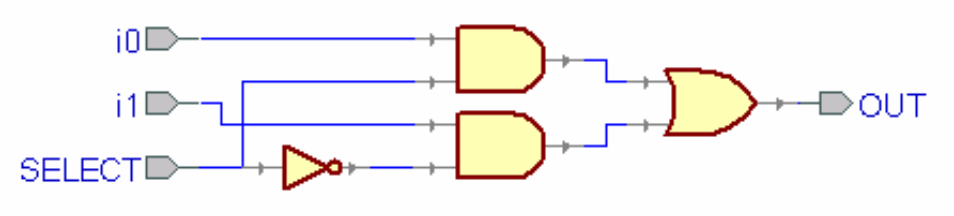
(a)



(b)



(c)



Exercise 2.43

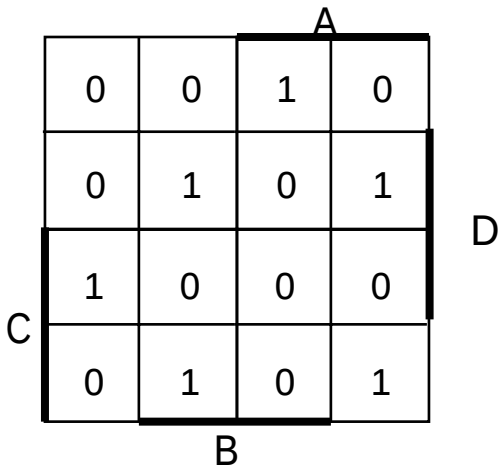
(a)

Input	Output
0000	0
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	0

(b) $\sum m(3, 5, 6, 9, 10, 12)$

(c) $\prod M(0, 1, 2, 4, 7, 8, 11, 13, 14, 15)$

(d) The sum-of-products K-map does not simplify since there are no adjacencies.



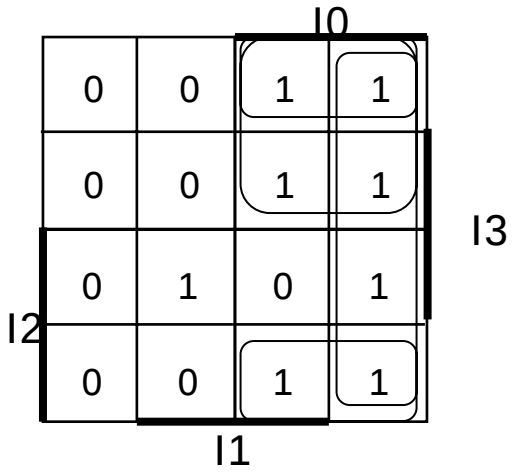
Exercise 2.44

(a)

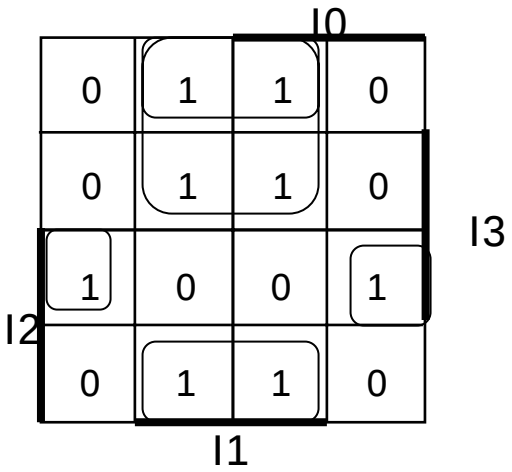
Input	Output
0000	0001
0001	0010
0010	0011
0011	0100
0100	0101
0101	0110
0110	0111
0111	1000
1000	1001
1001	1010
1010	1011
1011	1100
1100	1101
1101	1110
1110	1111
1111	0000

(b) Note: I_0 and O_0 are the most significant bits in each of their corresponding bit streams.

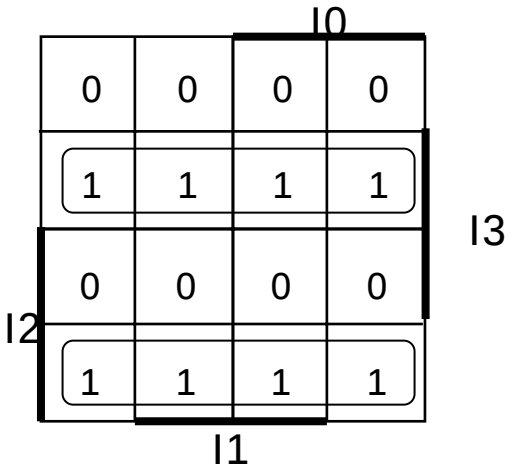
$$O_0 = I_0\, I_2' + I_0\, I_1' + I_0\, I_3' + I_0'\, I_1\, I_2\, I_3$$



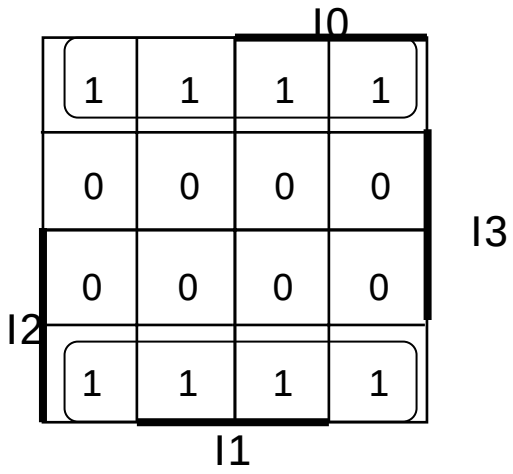
$O_1 = I_1 I_3' + I_1 I_2' + I_1' I_2 I_3$



$O_2 = I_2' I_3 + I_2 I_3'$

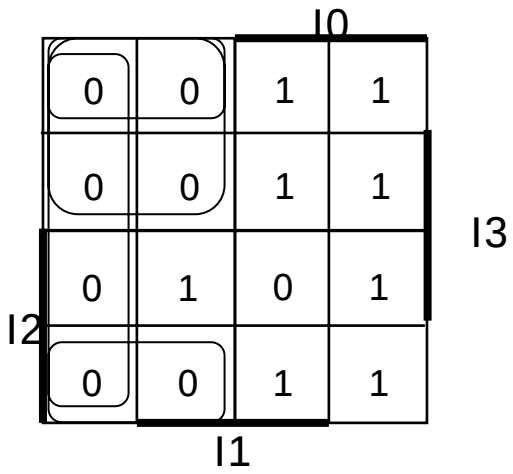


$O_3 = I_3'$

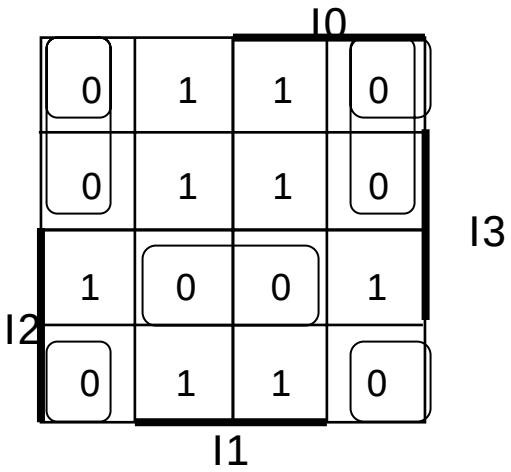


(c)

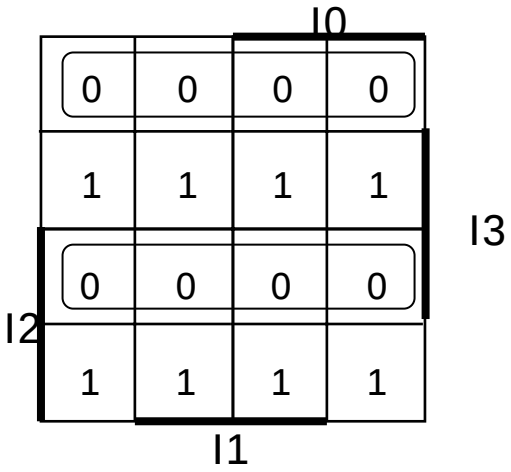
$O_0 = (I_0' + I_2') (I_0' + I_1') (I_0' + I_3') (I_0 + I_1 + I_2 + I_3)$



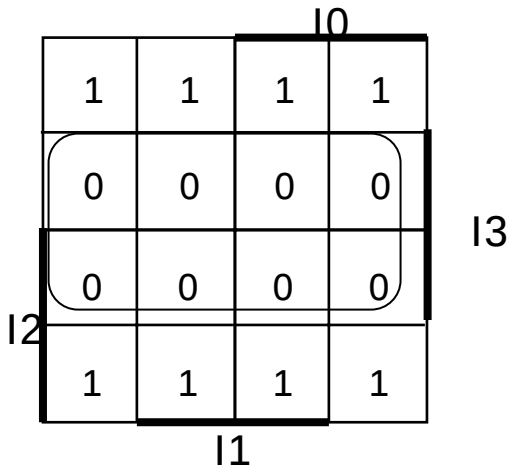
$O_1 = (I_1' + I_3') (I_1' + I_2') (I_1 + I_2 + I_3)$



$O_2 = (I_2 + I_3) (I_2' + I_3')$



$O_3 = I_3$



It turns out that both implementations are equivalent in the number of literals used for each of the Output bits.

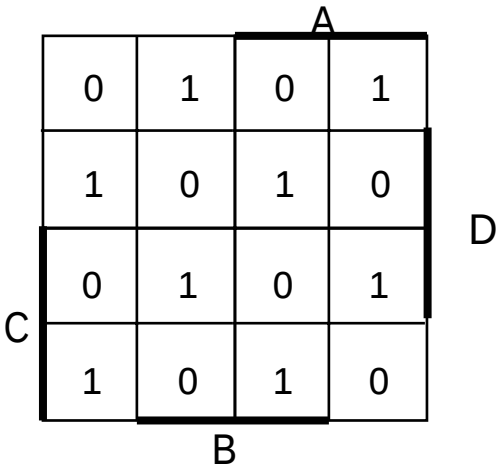
Exercise 2.45

(a)

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

(b) In the K-map, A represents the most significant bit of the input bit string. In this case the K-map method is not very useful in eliminating terms. The minimized form using the K-map is in fact the canonical sum-of-products form:

$$\sum m(1, 2, 4, 7, 8, 11, 13, 14)$$



(c) The truth table below shows that A ⊕ B ⊕ C ⊕ D is equivalent to the Output function. Logically this makes sense because if an even number of inputs are asserted, then the XOR of those inputs will always be 0. If an odd number of inputs are asserted, then *2n* asserted inputs before it produce a 0, which when XORed with the last asserted bit produces a 1.

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

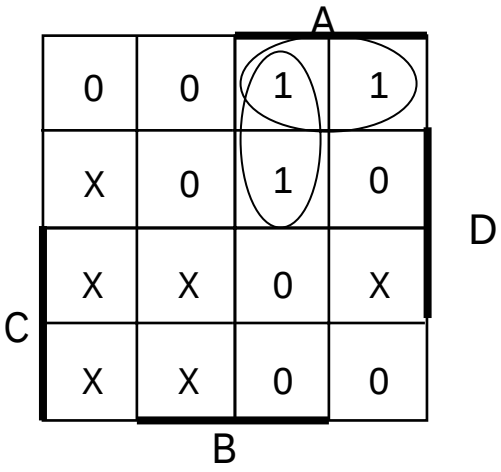
Exercise 2.46

(a)

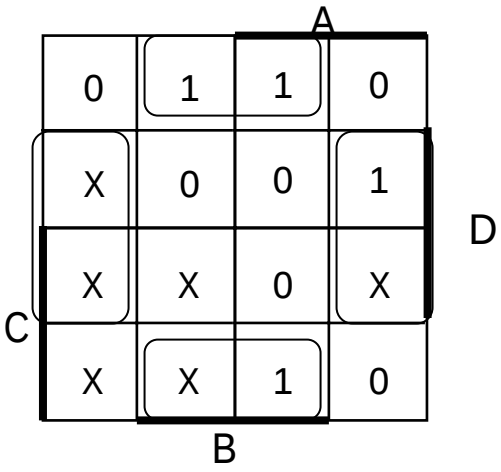
A	B	C	D	F	G
0	0	0	0	0	0
0	0	0	1	X	X
0	0	1	0	X	X
0	0	1	1	X	X
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	X	X
0	1	1	1	X	X
1	0	0	0	1	0
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	X	X
1	1	0	0	1	1
1	1	0	1	1	0
1	1	1	0	0	1
1	1	1	1	0	0

(b) In the case of F, the don't cares do not provide any help in simplifying the sum-of-products expression. They do help in simplifying G however.

F = ABC' + AC'D'

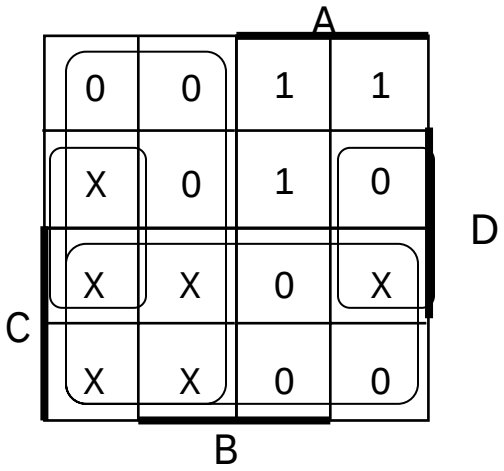


G = B'D + BD'

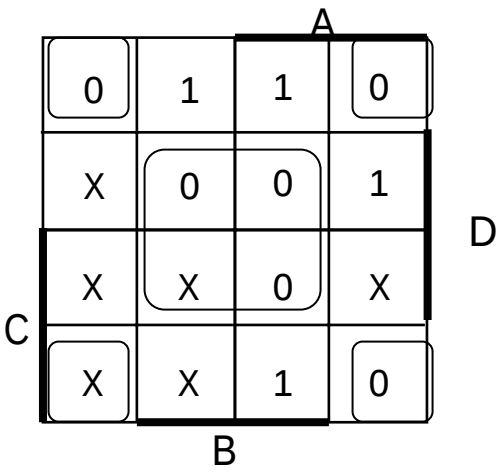


(c) The product-of-sums implementation turns out to be simpler than the sum-of-products implementation. This is because F now has only 4 literals instead of 6. Both implementations of G are about the same since they each have 4 literals and 2 terms.

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

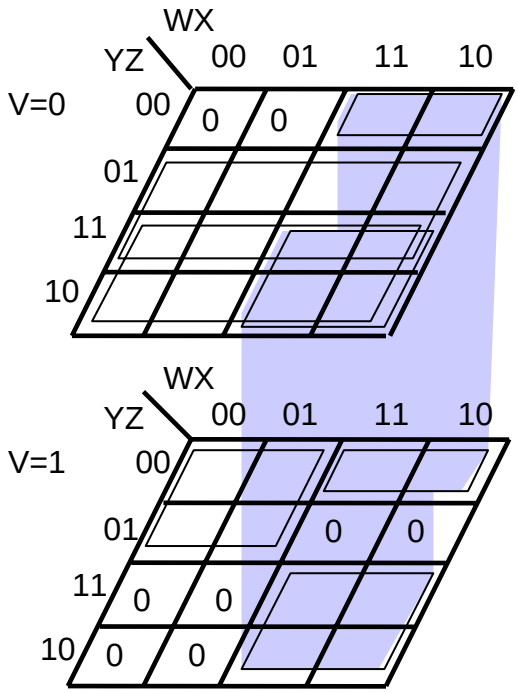


G = (B + D) (B’ + D’)



Exercise 3.1

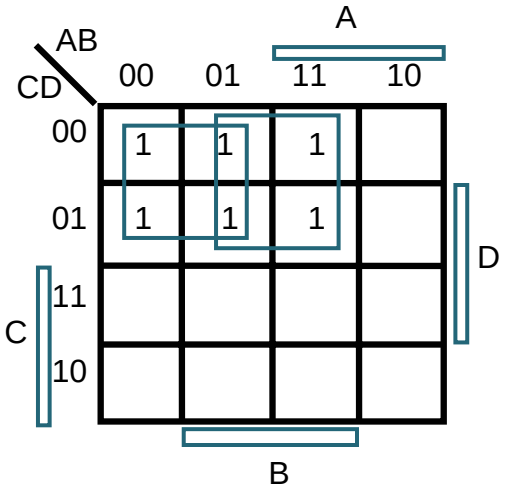
(a) f(V,W,X,Y,Z) = Π M(0,4,18,19,22,23,25,29)



The simplified function has 11 literals:

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

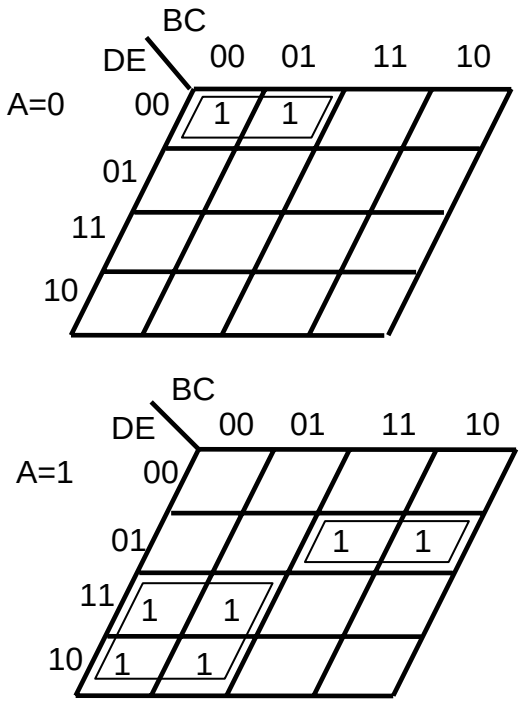
(b) f(A,B,C,D) = Σ m(0,1,4,5,12,13)



The simplified function has 4 literals:

f = A'C' + BC'

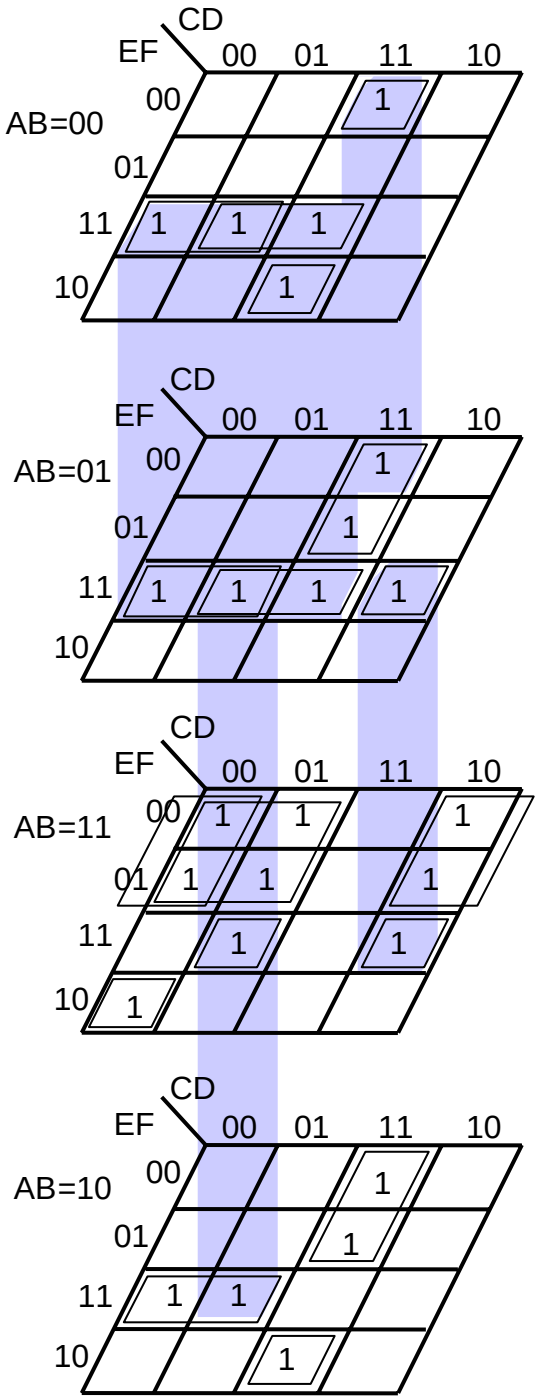
(c) f(A,B,C,D,E) = Σ m(0,4,18,19,22,23,25,29)



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.

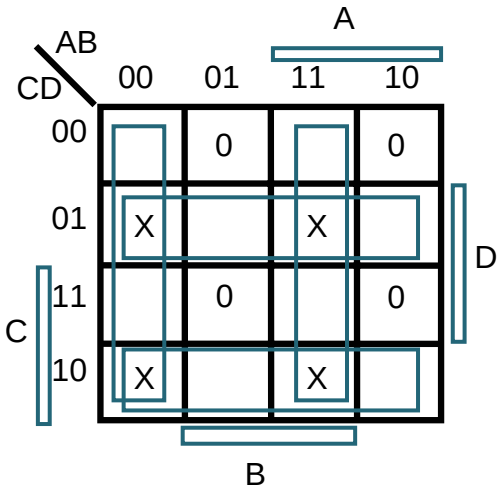
(d) The following are prime implicants:

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

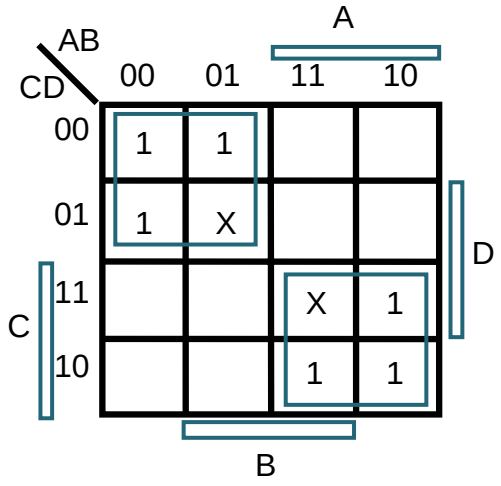
(a) f(W,X,Y,Z) = Π M(4,7,8,11)* Π D(1,2,13,14)



The simplified function is:

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

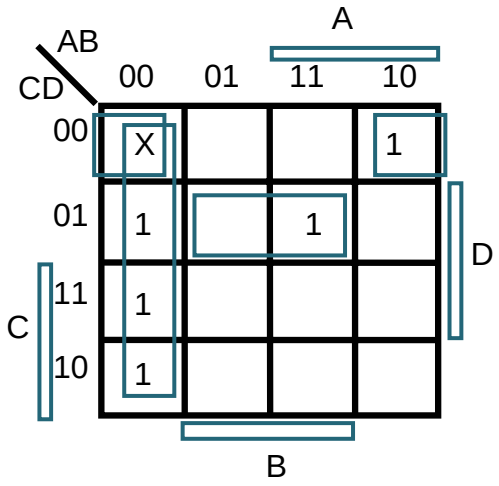
(b) f(A,B,C,D) = Σ m(0,1,4,10,11,14) + Σ d(5,15)



The simplified function is:

f = A'C' + AC

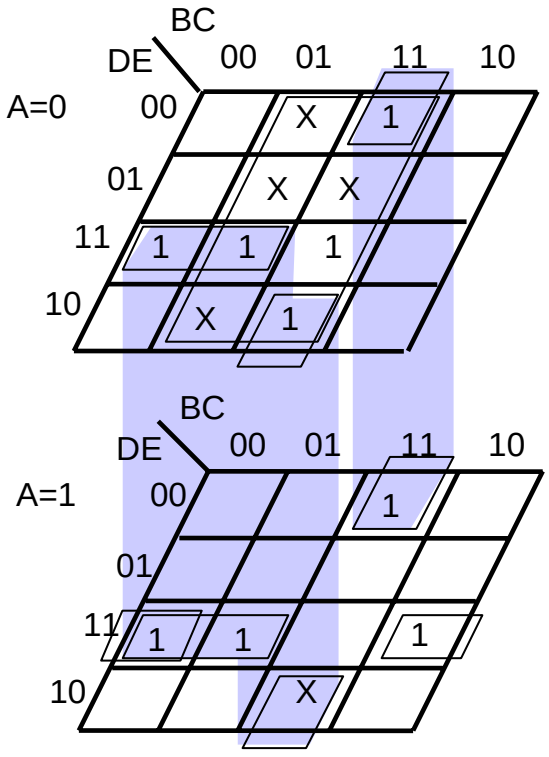
(c) f(A,B,C,D) = Σ m(1,2,3,5,8,13) + Σ d(0)



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:

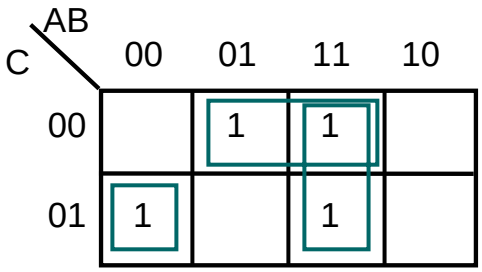
A'C, B'DE, BCE', AC'DE

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

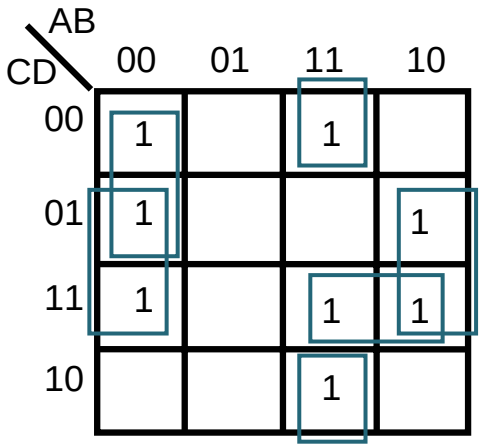
(a) $F(A,B,C) = \Sigma\, m(1,2,6,7)$



The simplified function is:

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

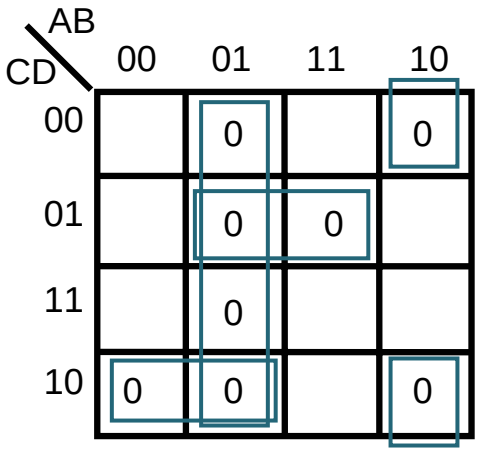
(b) $F(A,B,C,D) = \Sigma\, m(0,1,3,9,11,12,14,15)$



The simplified function is:

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

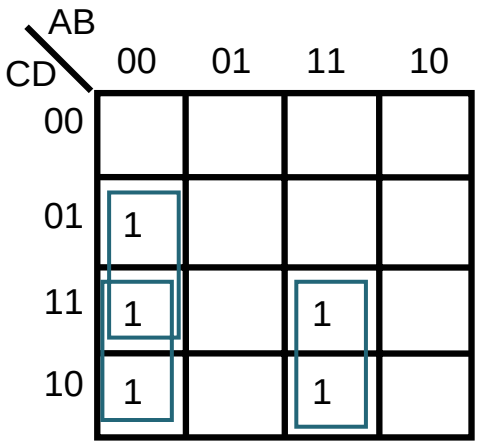
(c) F(A,B,C,D) = Σ m(2,4,5,6,7,8,10,13)



The simplified function is:

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

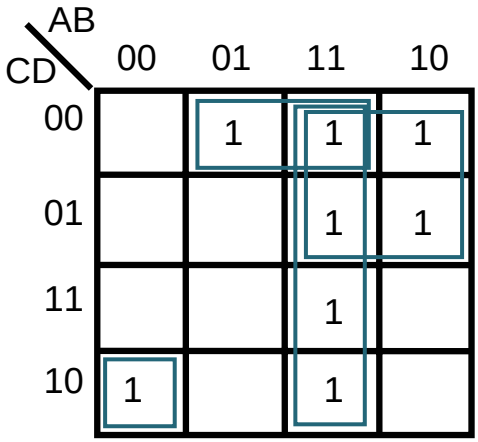
(d) F(A,B,C,D) = (ABC + A'B')(C + D)



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

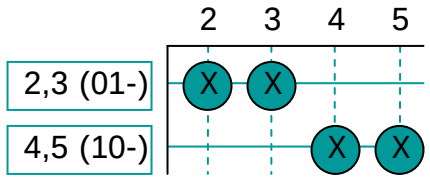
(a) f(X,Y,Z) = Σ m(2,3,4,5)

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'

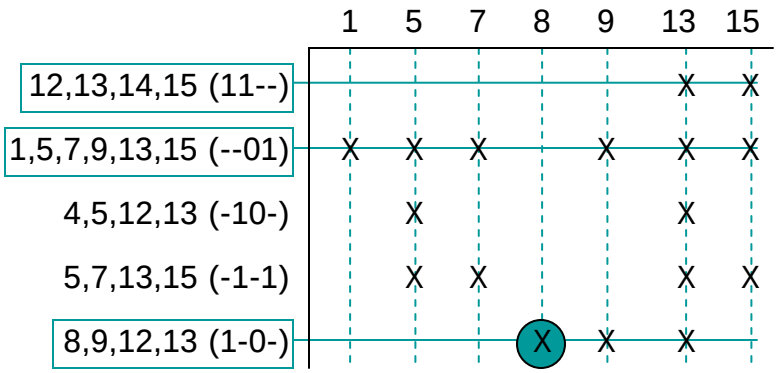
(b) f(A,B,C,D) = Σ m(1,5,7,8,9,13,15) + Σ d(4,12,14)

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'

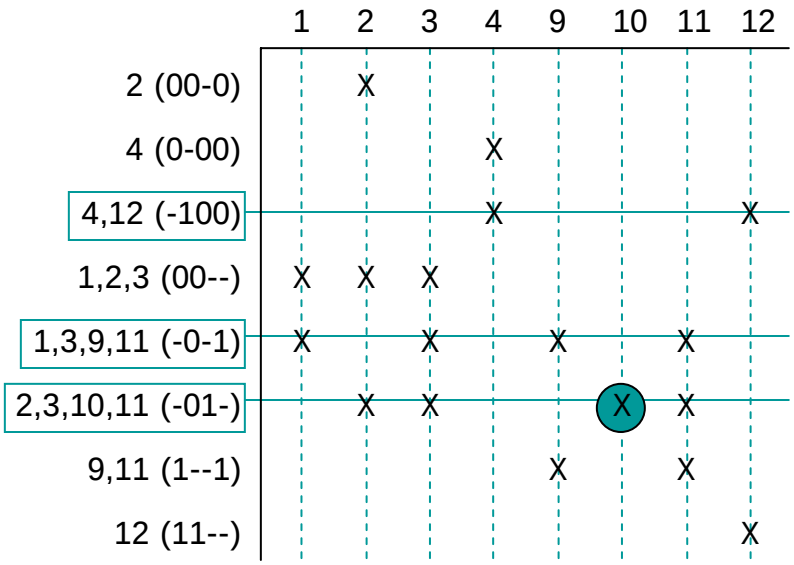
(d) f(A,B,C,D) = Σ m(1,2,3,4,9,10,11,12) + Σ d(0,13,14,15)

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



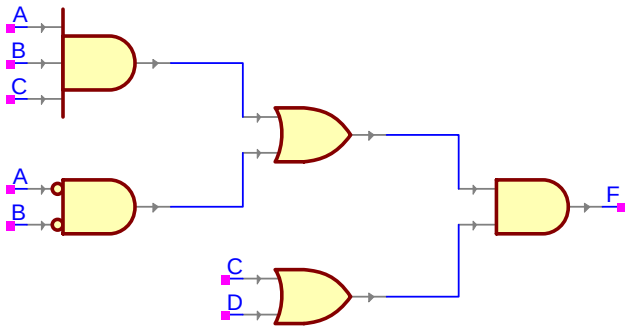
Final configuration

The resulting simplified function is:

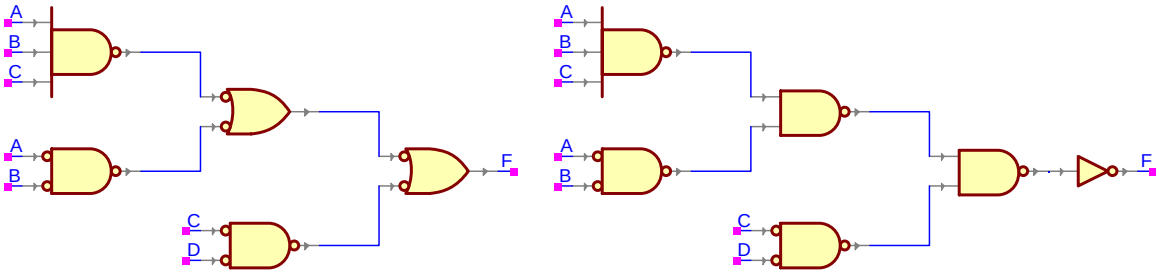
f = BC'D' + B'D + B'C

Exercise 3.8

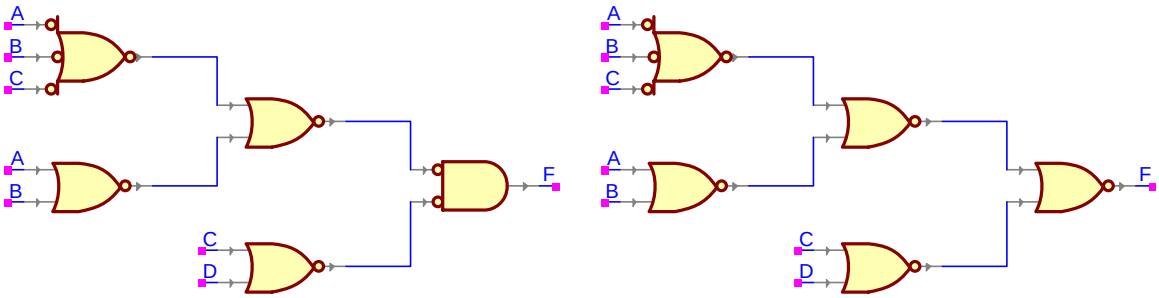
(a) F(A,B,C,D) = (ABC + A'B')(C + D)



Initial Circuit

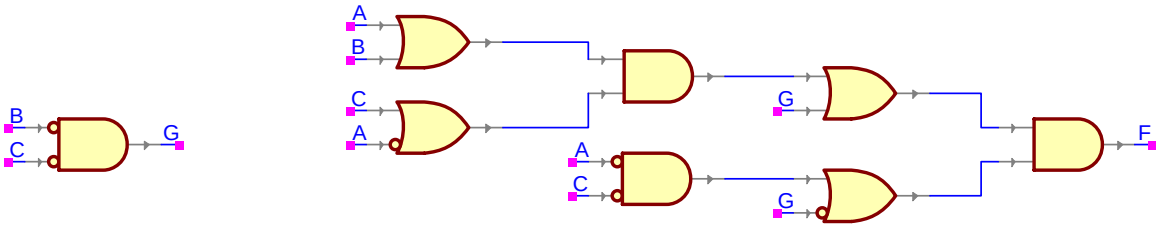


NAND Gate Implementation

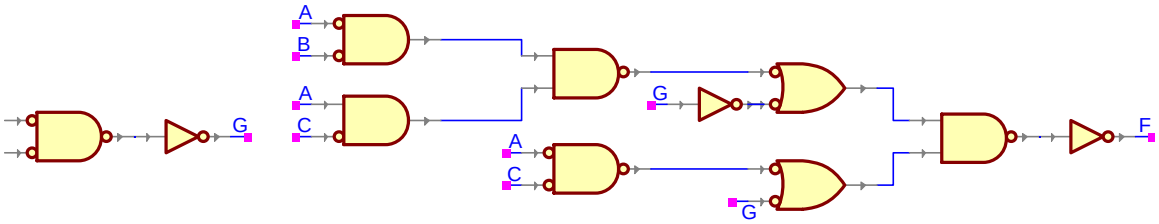


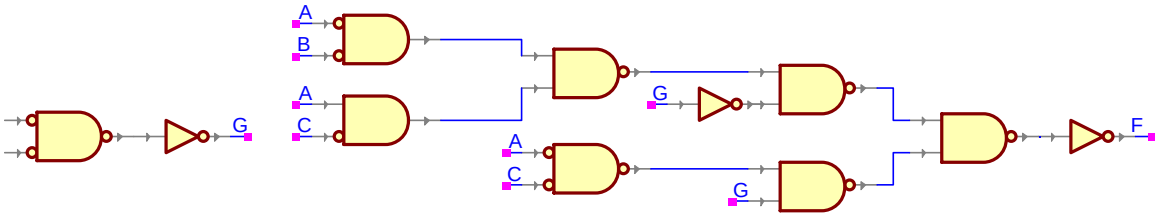
NOR Gate Implementation

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

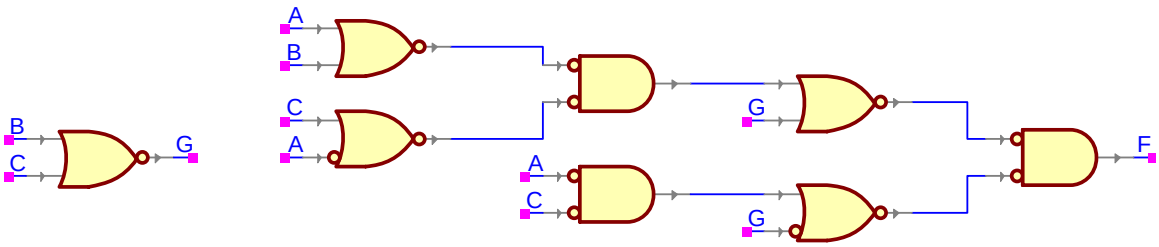


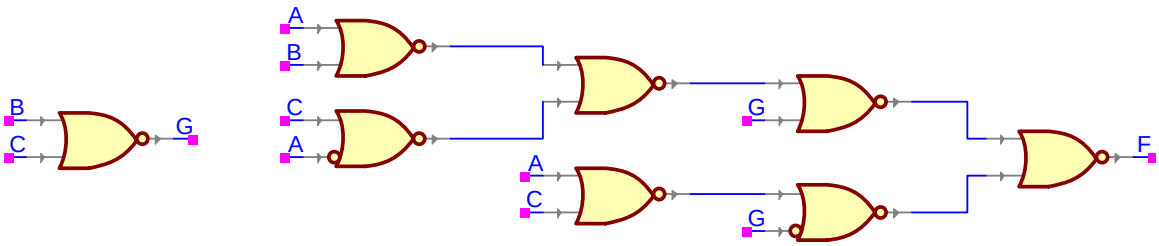
Initial Circuit





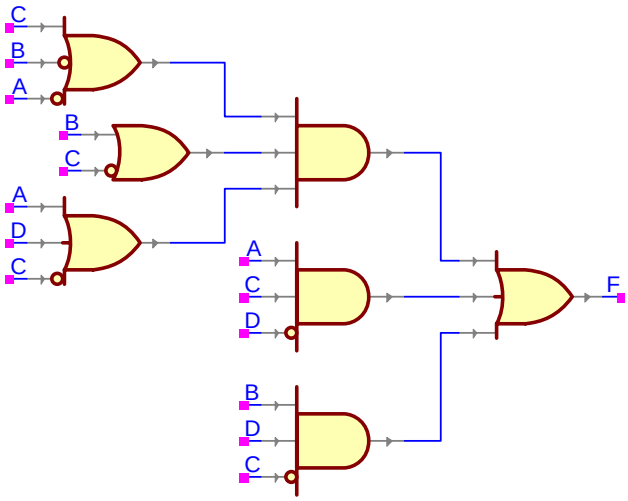
NAND Gate Implementation



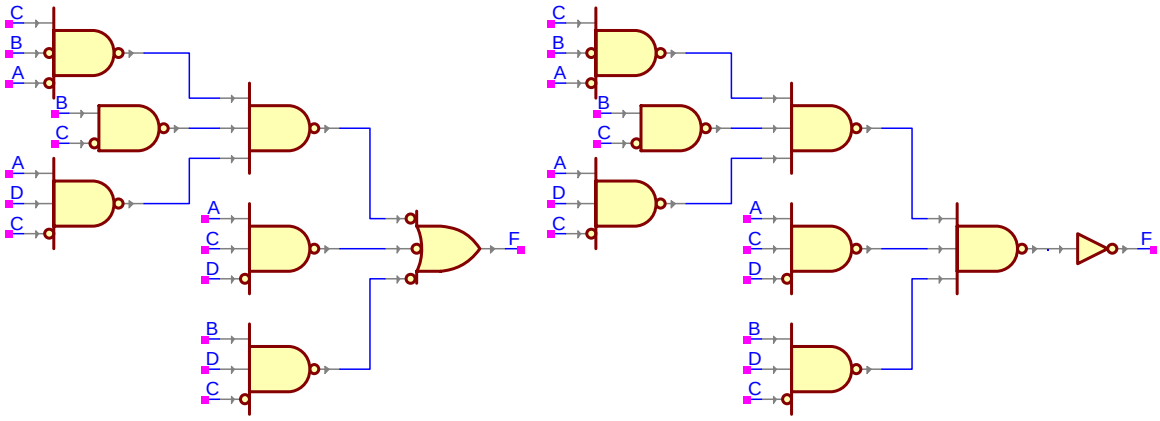


NOR Gate Implementation

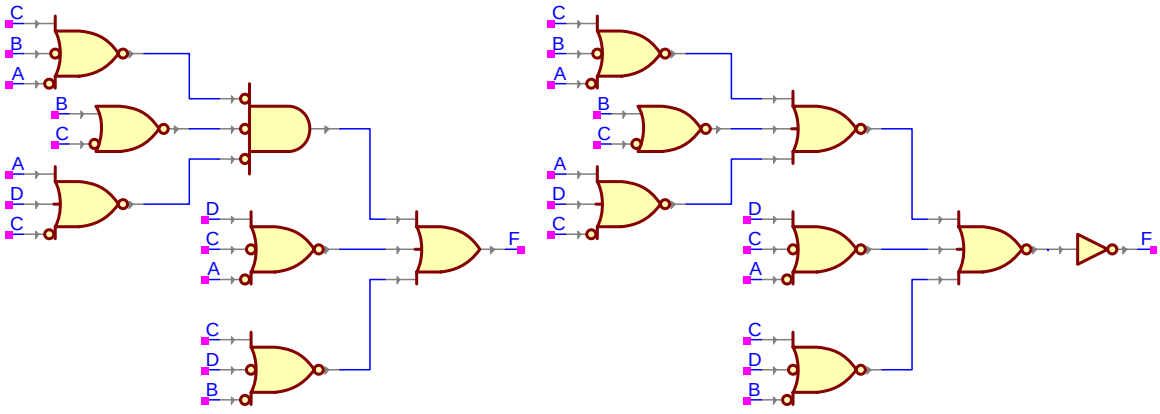
(c) F(A,B,C,D) = [(A' + B' + C)(B + C')(A + B' + D')] + ACD' + BC'D



Initial Circuit

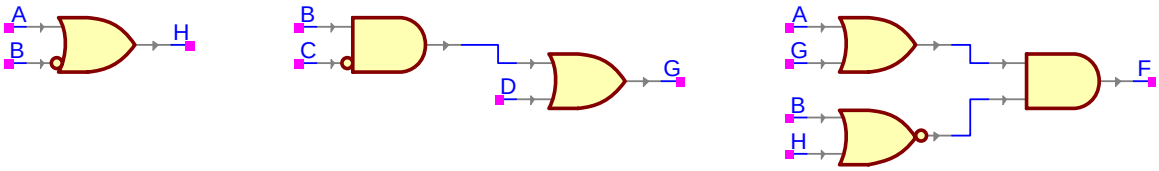


NAND Gate Implementation

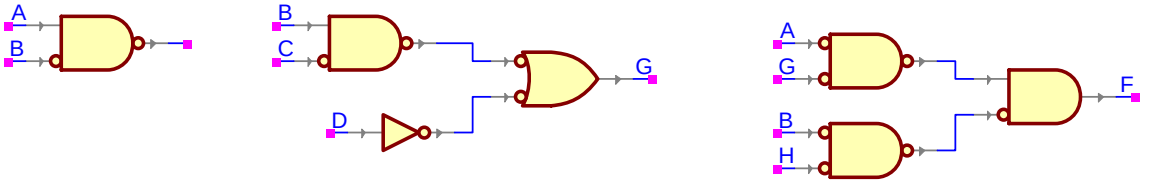


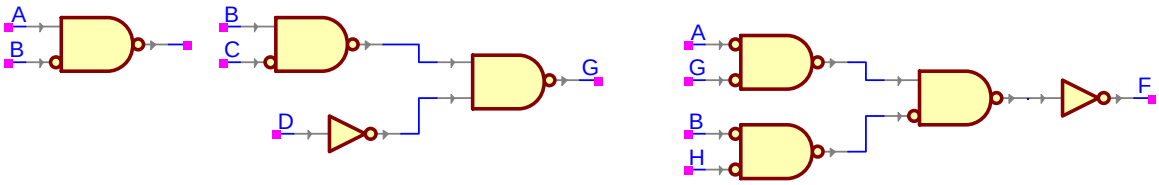
NOR Gate Implementation

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

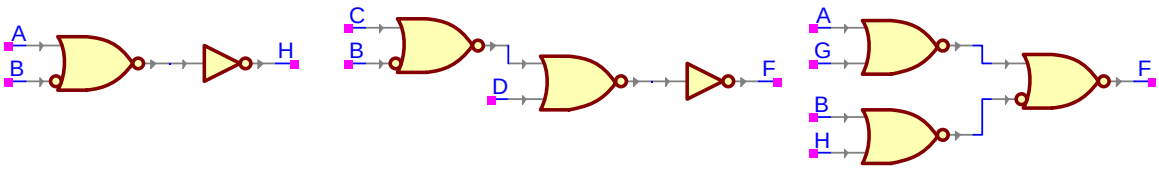


Initial Circuit





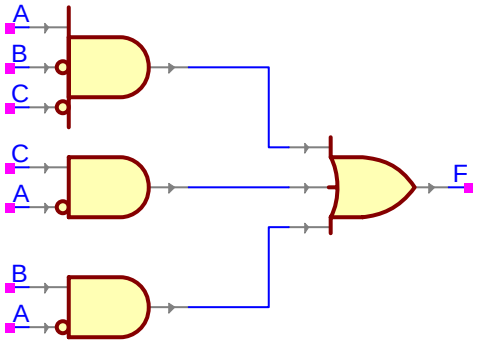
NAND Gate Implementation



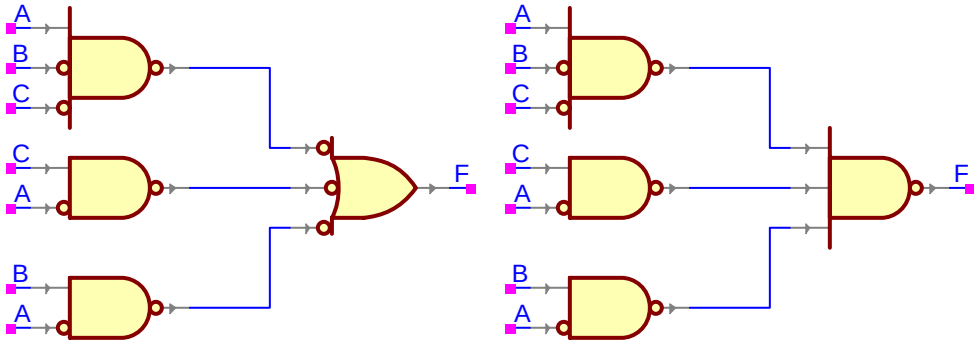
NOR Gate Implementation

Exercise 3.9

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

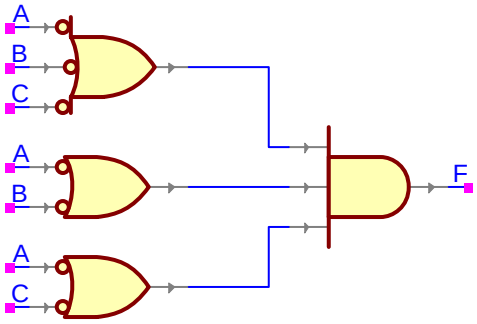


Initial Circuit

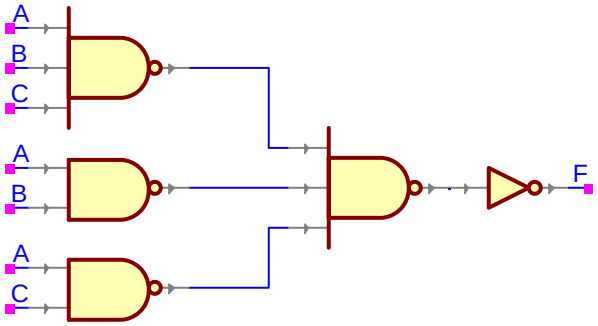


NAND Gate Implementation

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

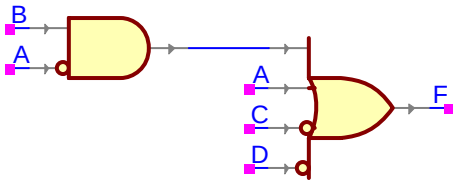


Initial Circuit

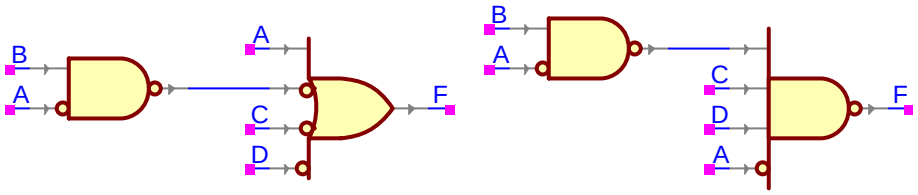


NAND Gate Circuit

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

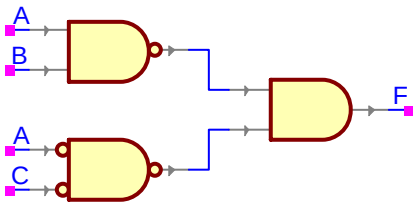


Initial Circuit

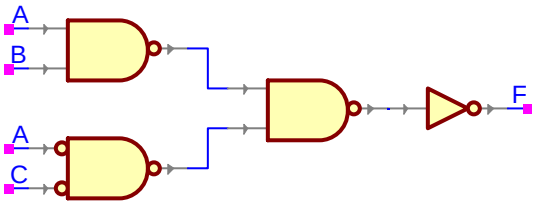


NAND Gate Implementation

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

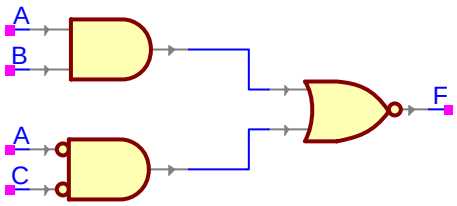


Initial Circuit

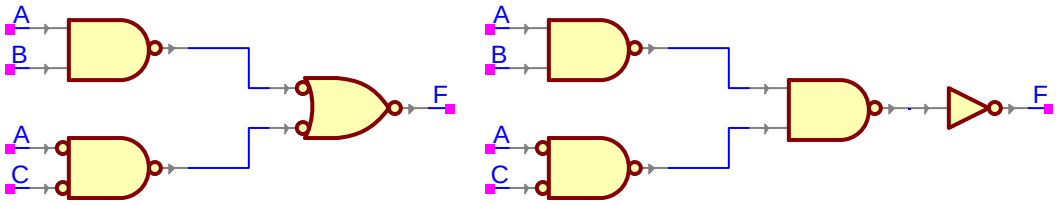


NAND Gate Implementation

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



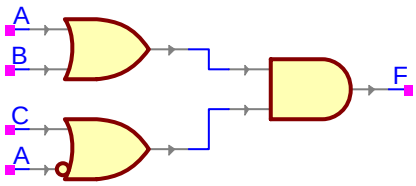
Initial Circuit



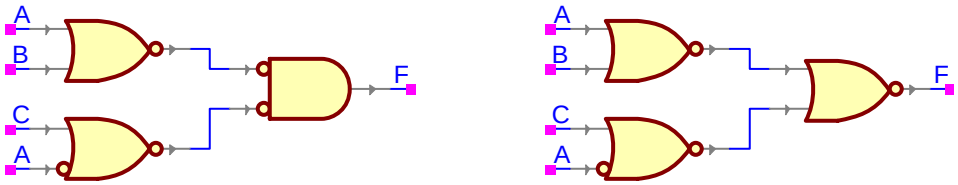
NAND Gate Implementation

Exercise 3.10

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

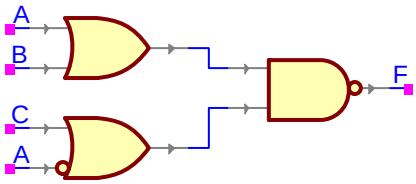


Initial Circuit

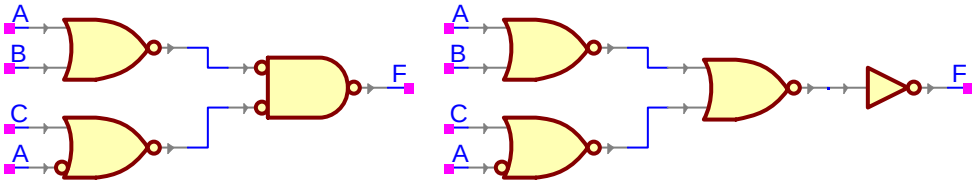


NOR Gate Implementation

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

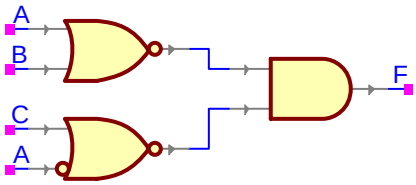


Initial Circuit

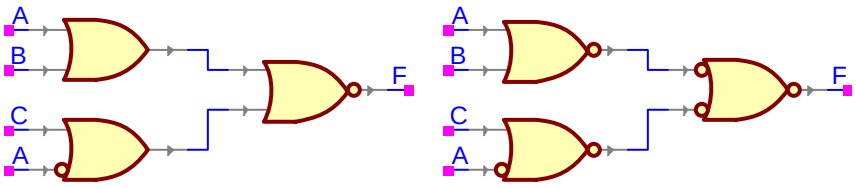


NOR Gate Implementation

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

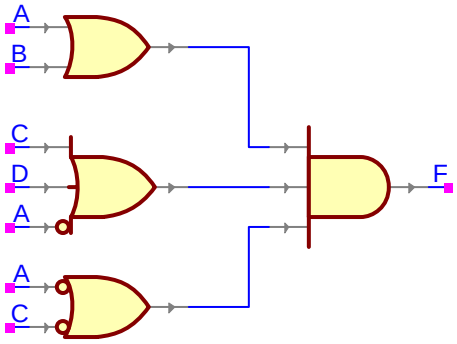


Initial Circuit

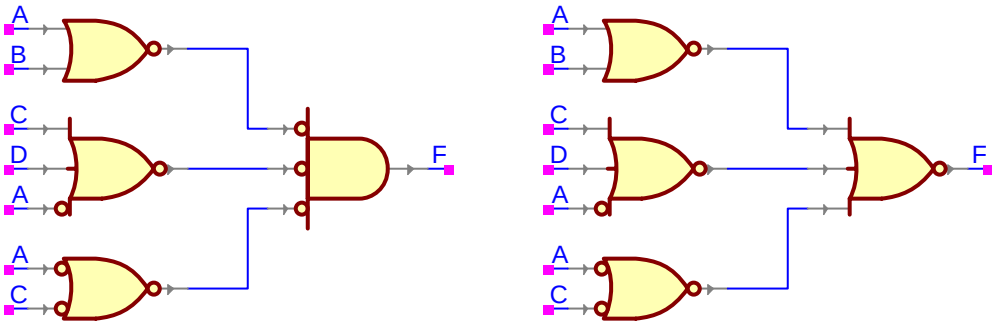


NOR Gate Implementation

(d) F = (A + B)(A’ + C + D)(A’ + C’)

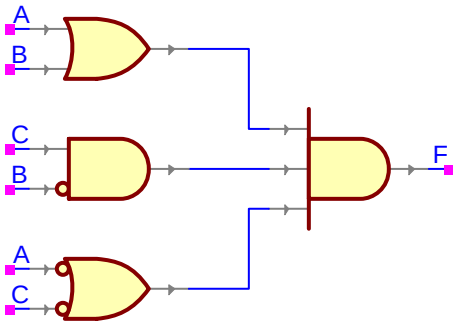


Initial Circuit

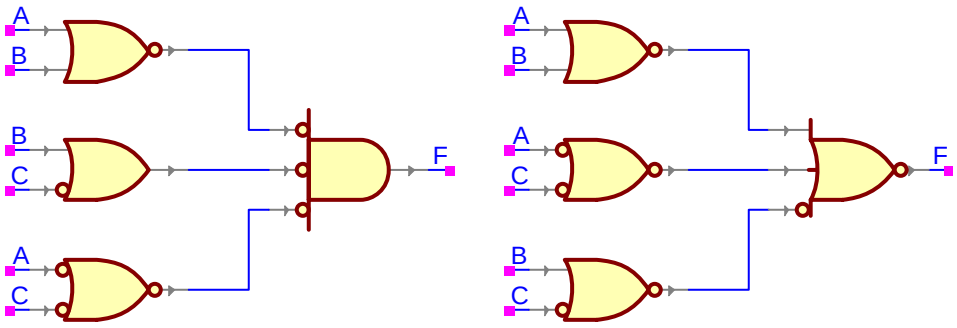


NOR Gate Implementation

(e) F = (A + B)(B’ + C)(A’ + C’)



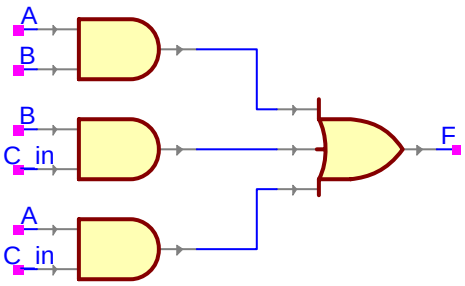
Initial Circuit



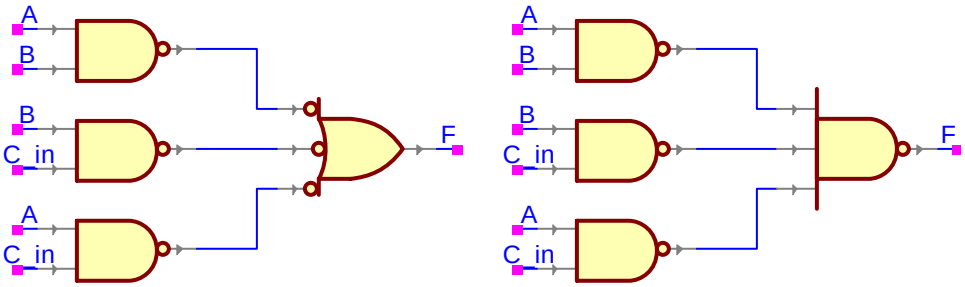
NOR Gate Implementation

Exercise 3.11

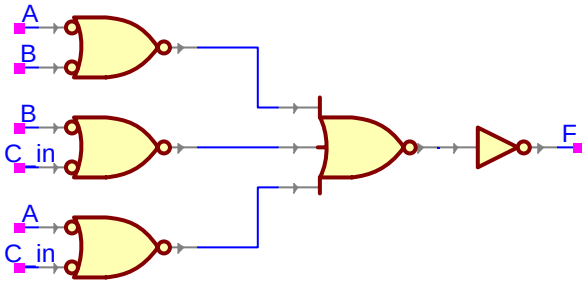
(a) F(A,B,Cin) = AB + BCin + ACin



Initial Circuit

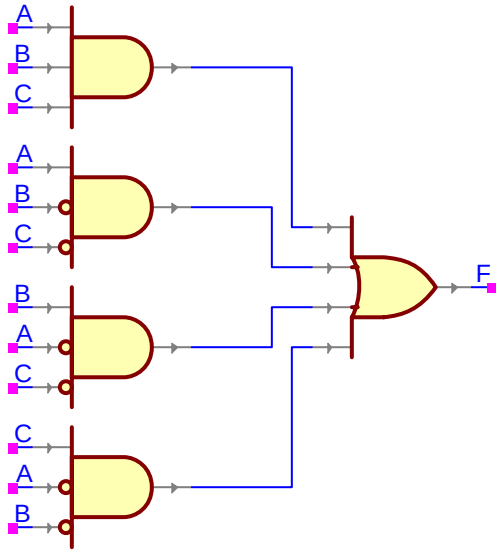


NAND Gate Implementation

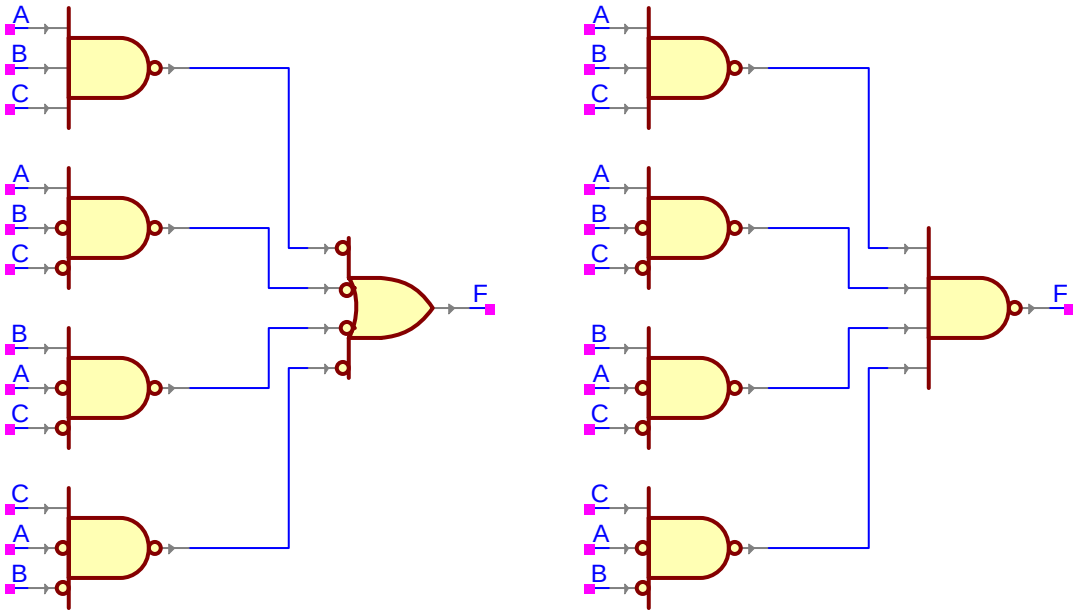


NOR Gate Implementation

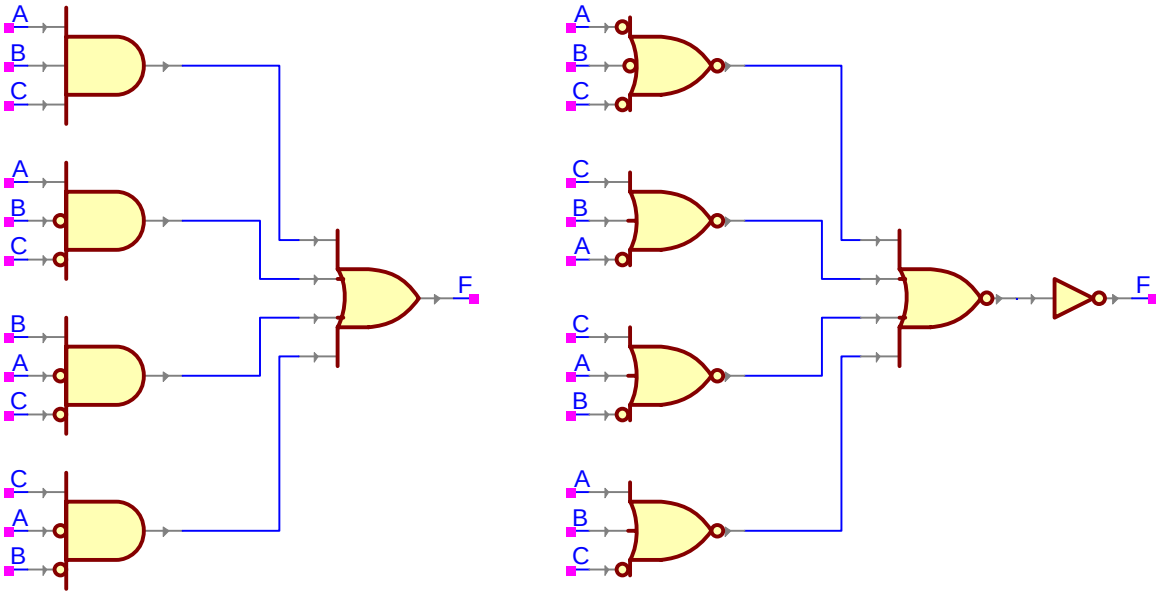
(b) F(A,B,Cin) = A xor B xor Cin



Initial Circuit



NAND Gate Implementation



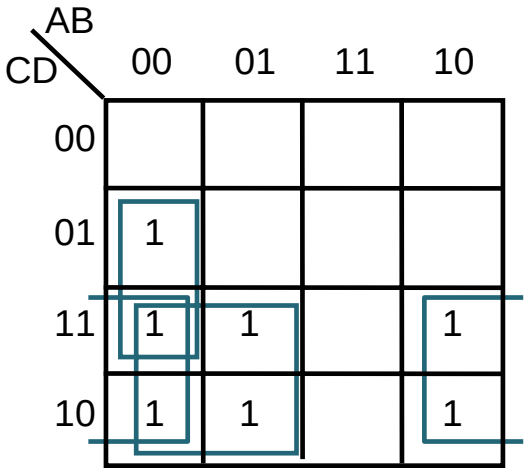
NOR Gate Implementation

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

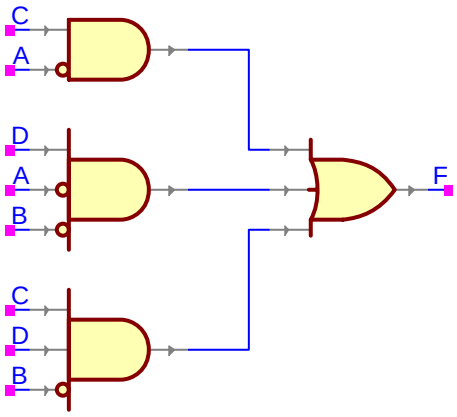
Truth table for F:

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

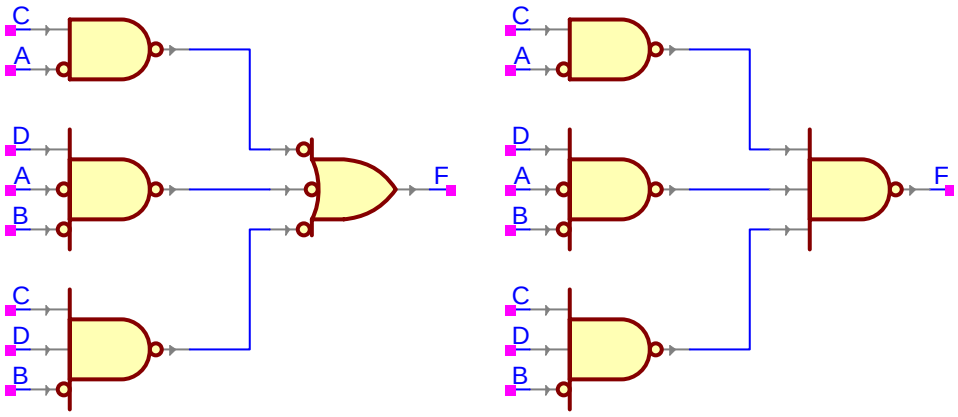
K-map for F:



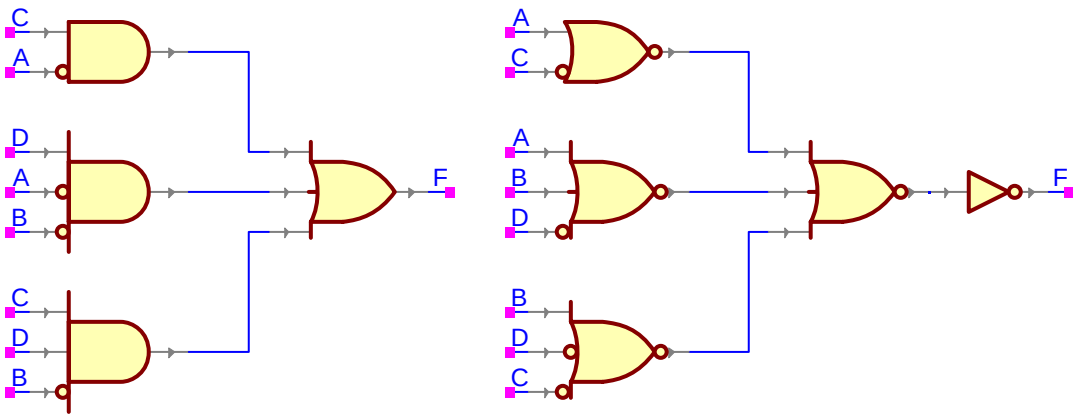
F = A'C + A'B'D + B'CD



Initial Circuit



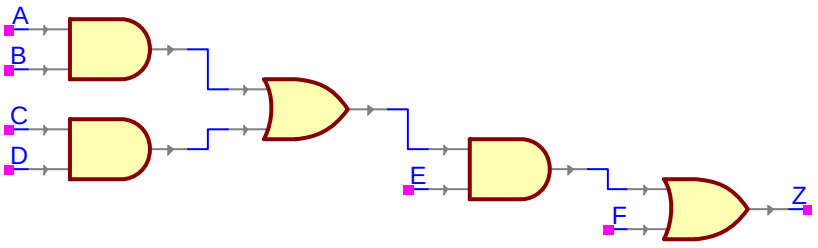
NAND Gate Implementation



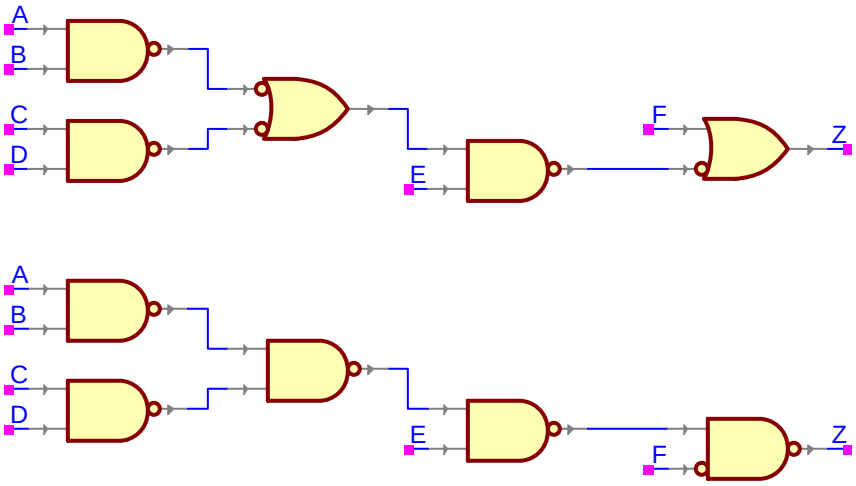
NOR Gate Implementation

Exercise 3.12

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

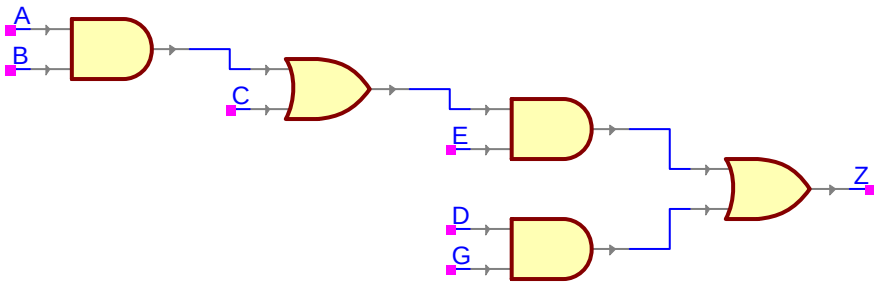


Initial Circuit

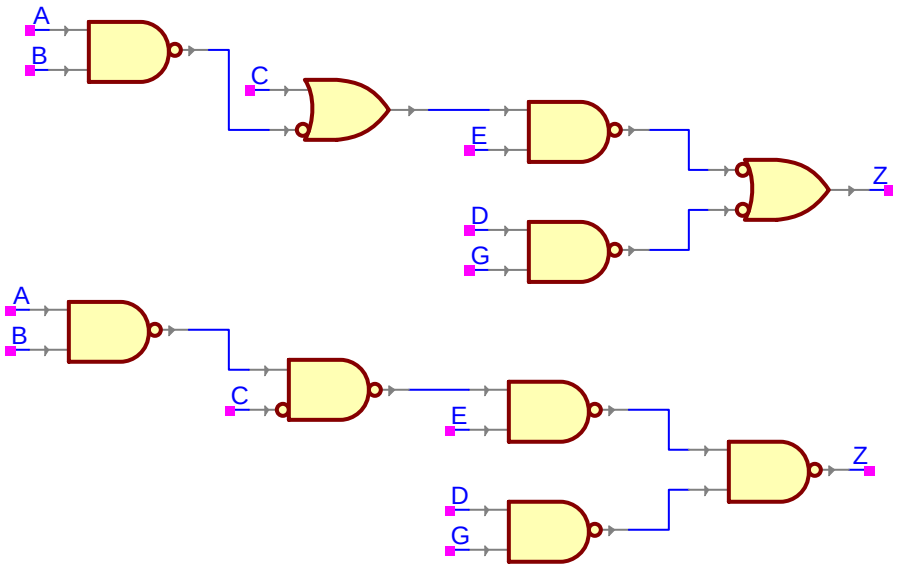


NAND/NOR Gate Implementation

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

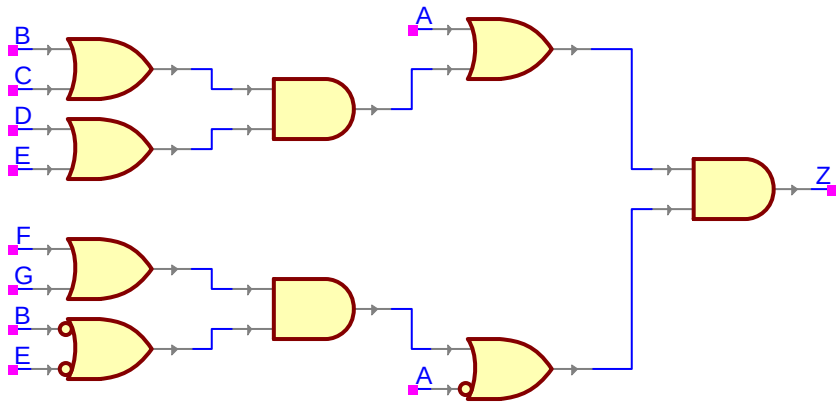


Initial Circuit

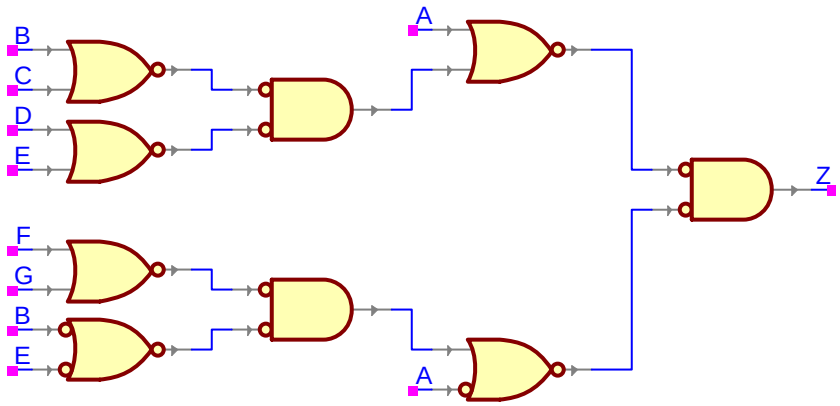


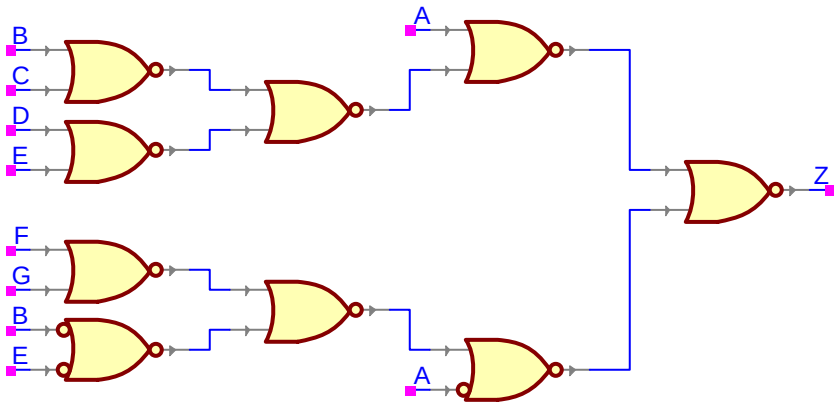
NAND/NOR Gate Implementation

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



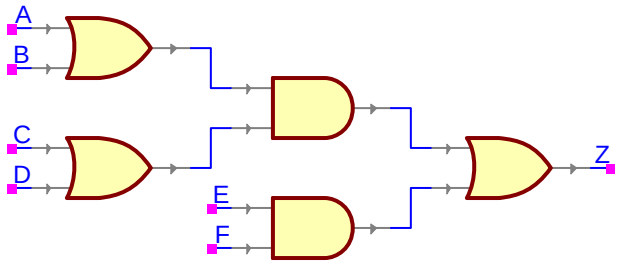
Initial Circuit



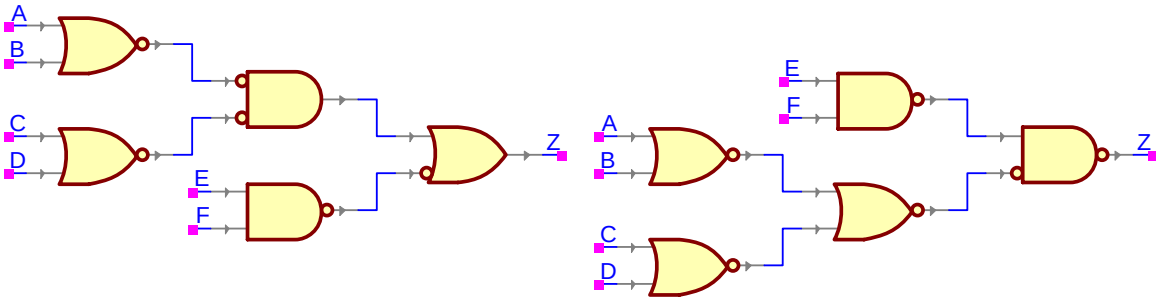


NAND/NOR Gate Implementation

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

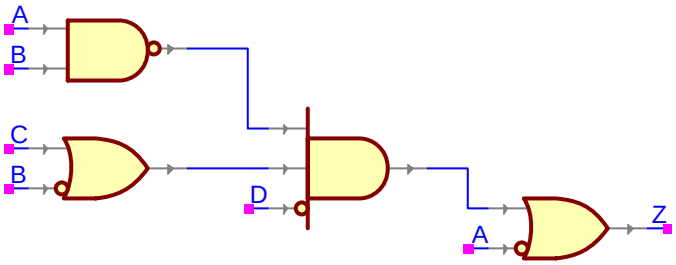


Initial Circuit

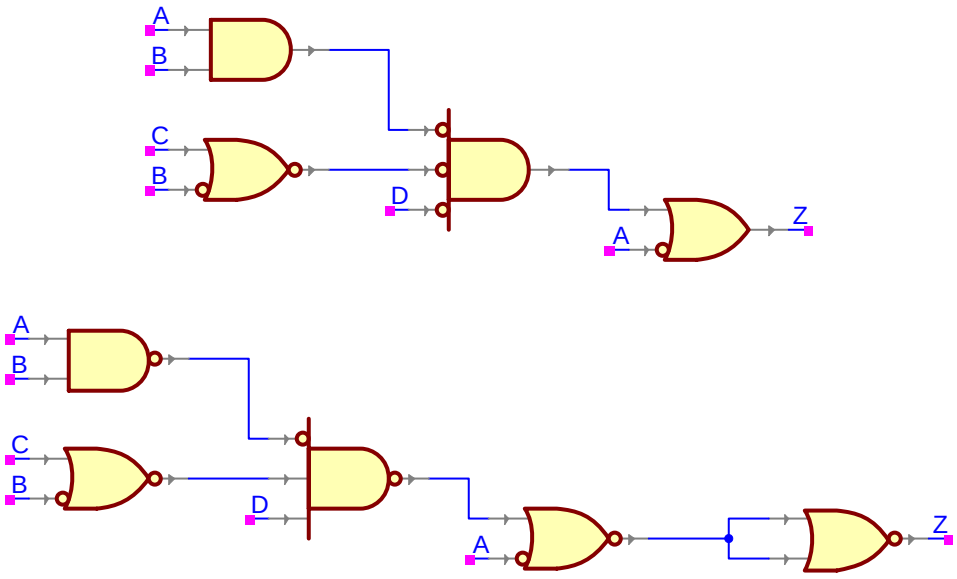


NAND/NOR Gate Implementation

(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)'

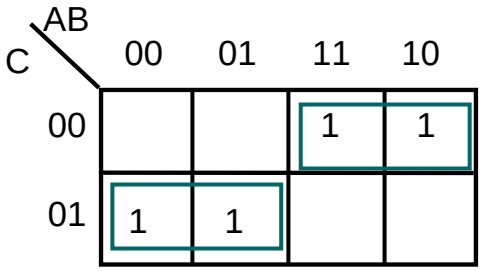
(b) Truth table:

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:

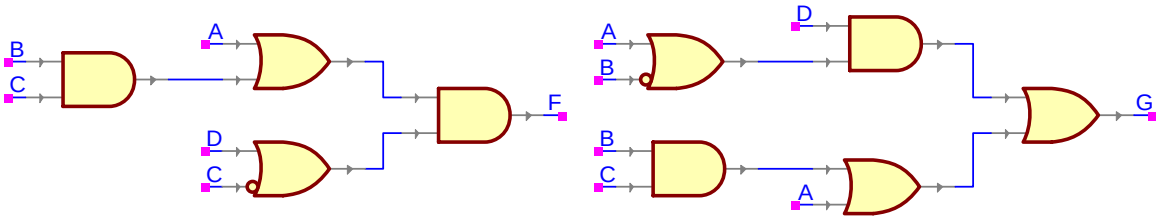
Z = $\sum$ m(1,3,4,6)

(d) K-map:



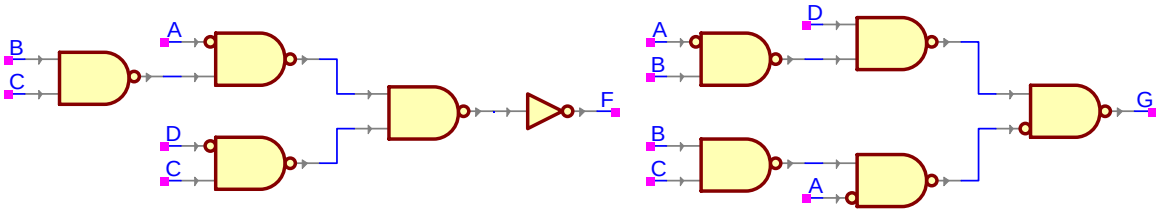
Exercise 3.16

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

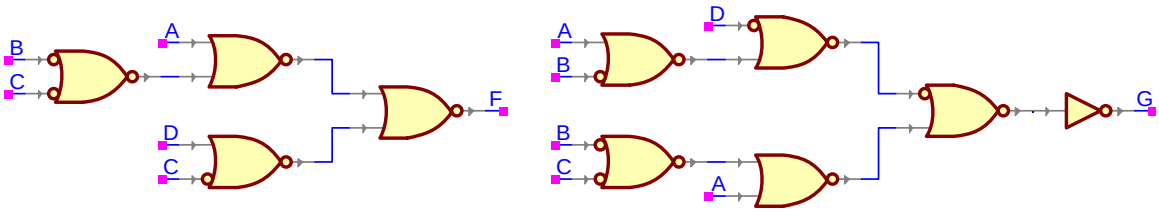


Initial Circuits

(a) NAND-only implementations:



(b) NOR-only implementations:

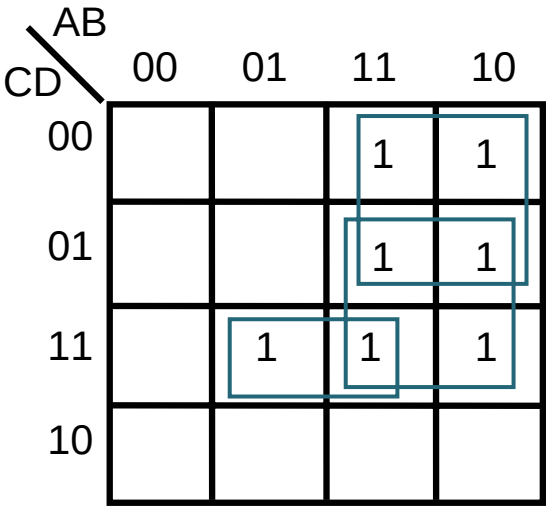


(c) Truth tables for F and G:

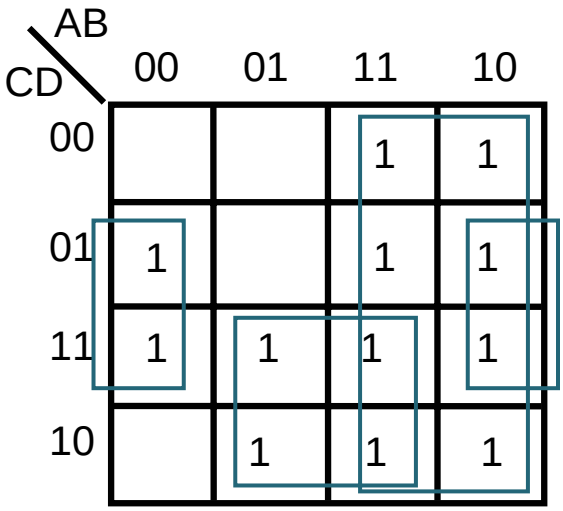
A	B	C	D	F	G
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	1
0	1	1	1	1	1
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	1	1	1
1	1	0	0	1	1

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

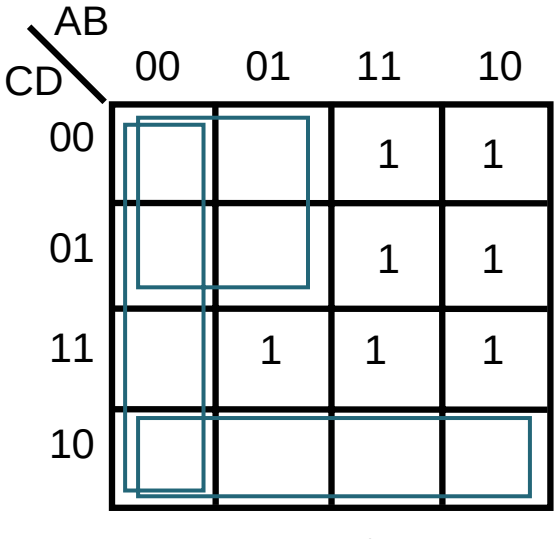


K-Map for G

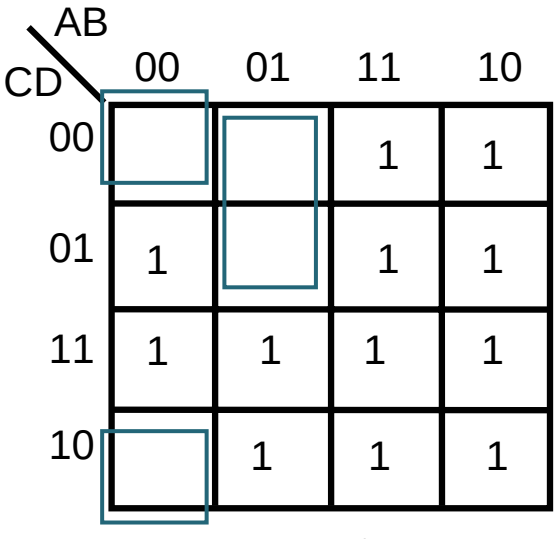
F = AC' + AD + BCD

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.

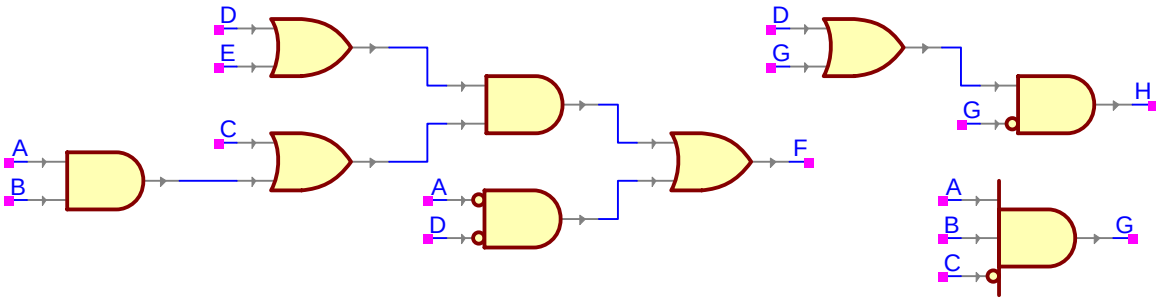
So in both cases the product-of-sums implementation was less complex.

Exercise 3.17

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

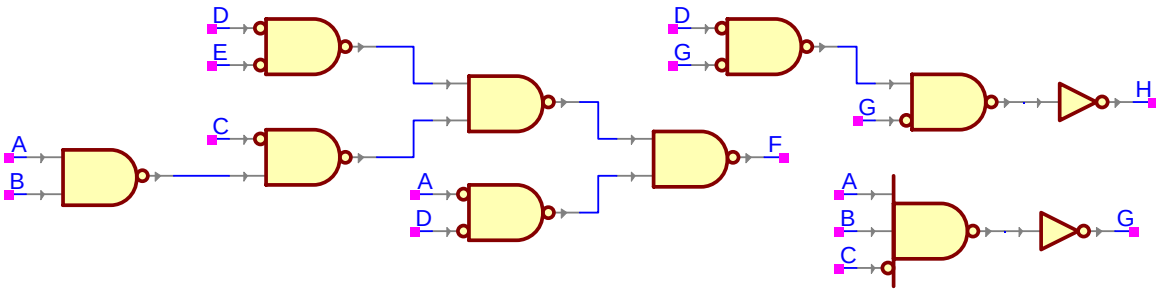
G(A,B,C) = (ABC')

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

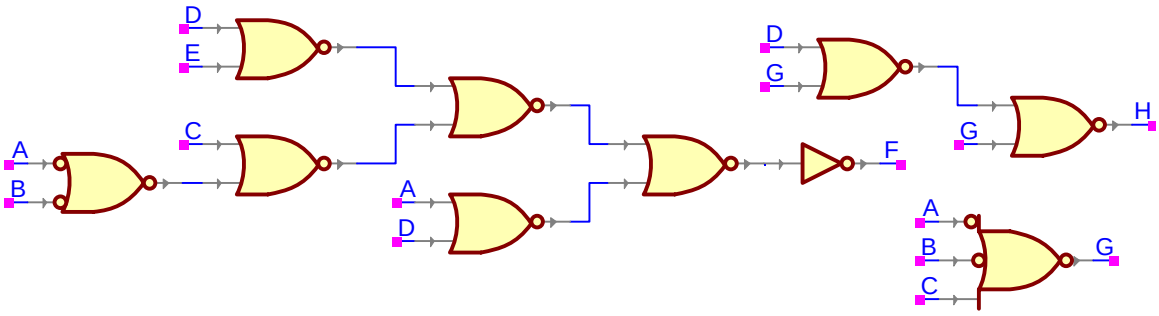


Initial Circuits

(a) NAND-only implementations:



(b) NOR-only implementations:



(c) Minimized Sum-of-Products:

Truth tables for F, G, and H:

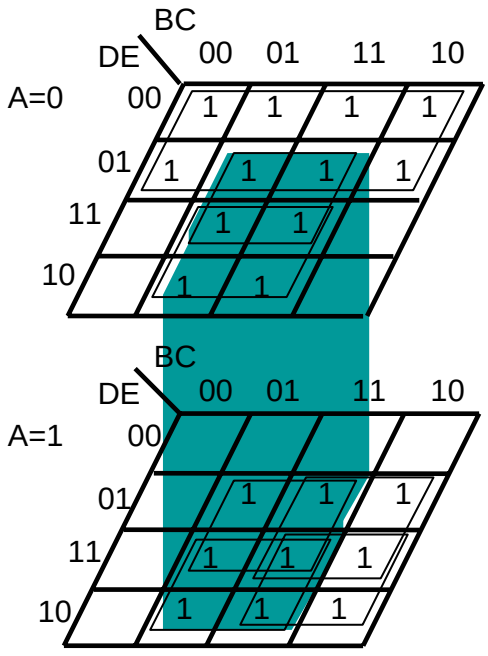
A	B	C	D	E	F	G	H
0	0	0	0	0	1	0	0
0	0	0	0	1	1	0	0
0	0	0	1	0	0	0	1
0	0	0	1	1	0	0	1
0	0	1	0	0	1	0	0
0	0	1	0	1	1	0	0
0	0	1	1	0	1	0	1
0	0	1	1	1	1	0	1
0	1	0	0	0	1	0	0
0	1	0	0	1	1	0	0
0	1	0	1	0	0	0	1
0	1	0	1	1	0	0	1
0	1	1	0	0	1	0	0
0	1	1	0	1	1	0	0
0	1	1	1	0	1	0	1
0	1	1	1	1	1	0	1
1	0	0	0	0	0	0	0
1	0	0	0	1	0	0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

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

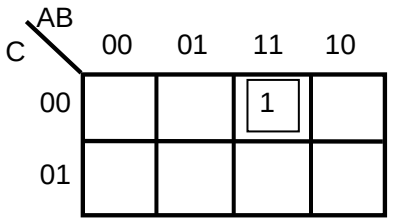
$G(A,B,C) = \Sigma\, m(6)$

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

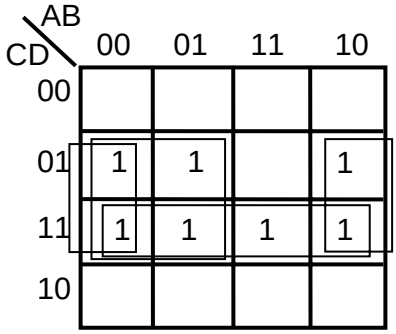
K-Maps:



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’

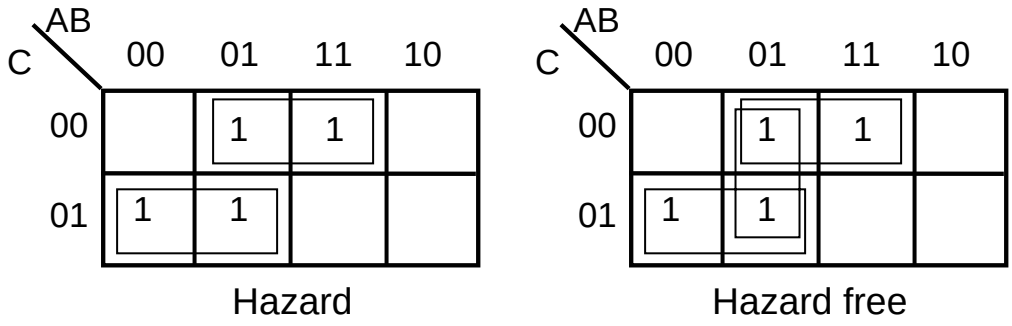
H’ = D’ + ABC’

H’’ = (D’)’(ABC’)’

H(A,B,C,D) = D(A’ + B’ + C)

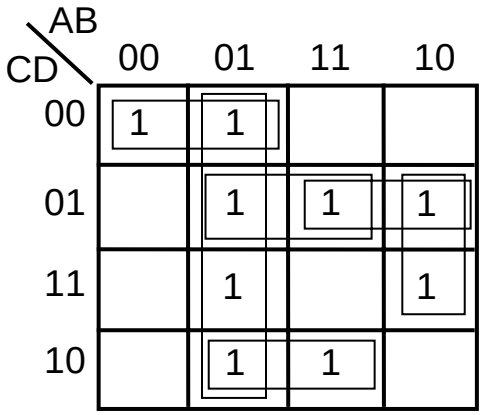
Exercise 3.18

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



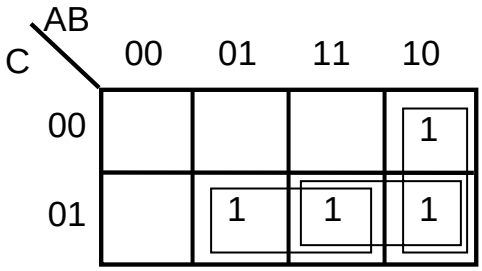
F(A,B,C) = A'B + A'C + BC'

(b) The K-map for F:



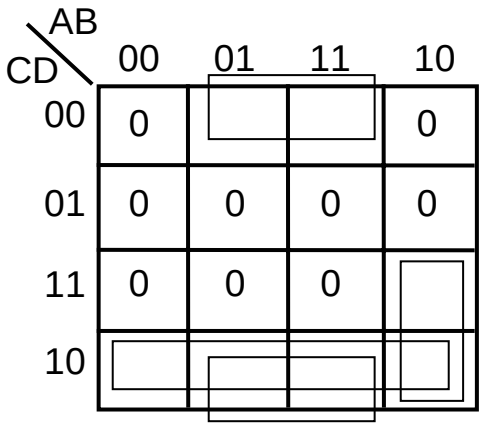
F(A,B,C,D) = A'C'D' + AC'D + AB'D + BC'D + BCD' + A'B

(c) The K-map for F:



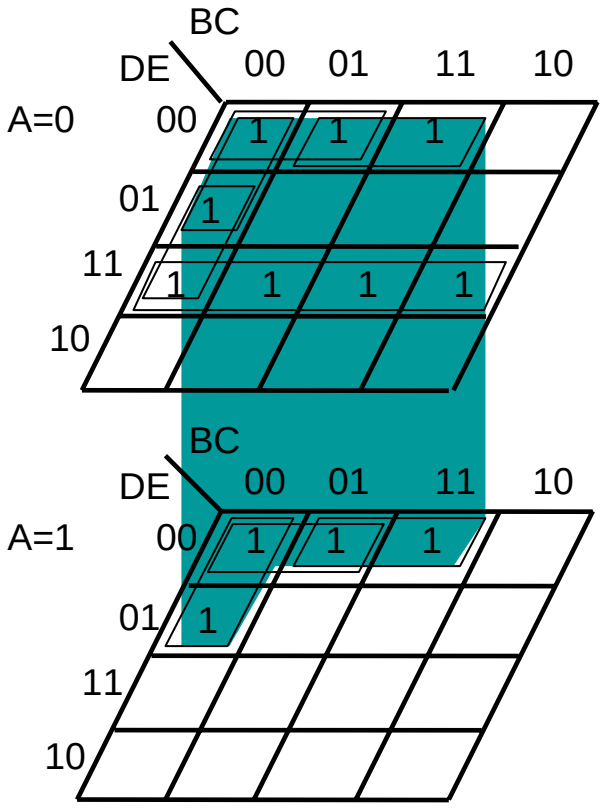
F(A,B,C) = AB' + AC + BC

(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’

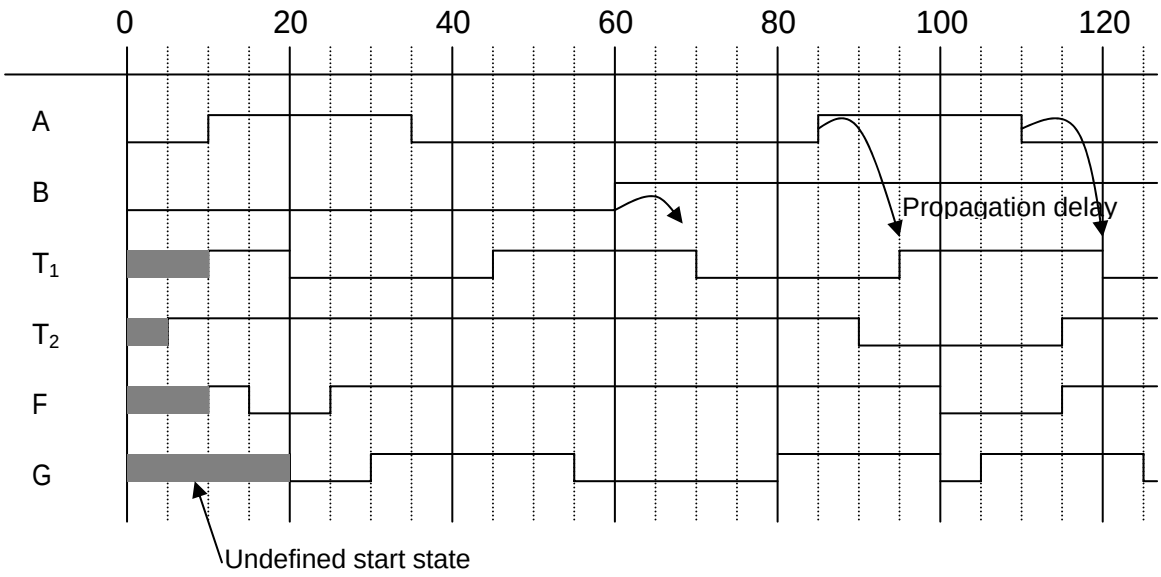
Exercise 3.19

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:

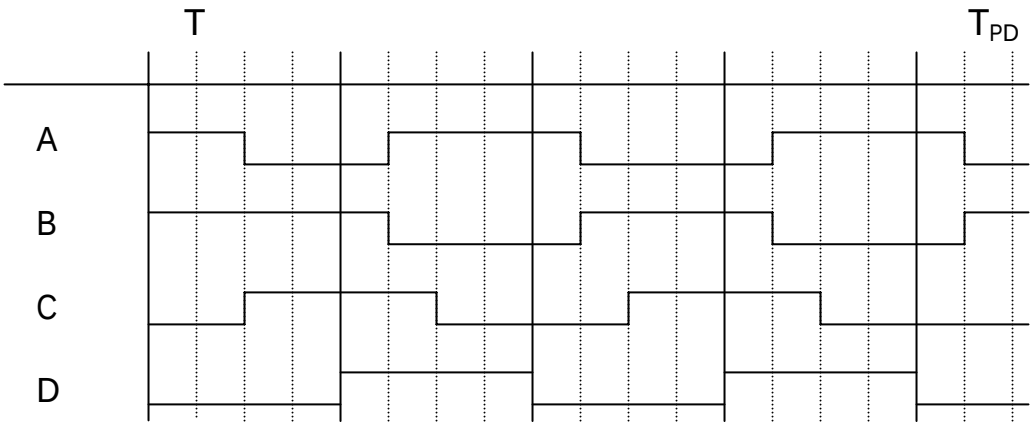
A = 1

B = 1

C = 0

D = 0

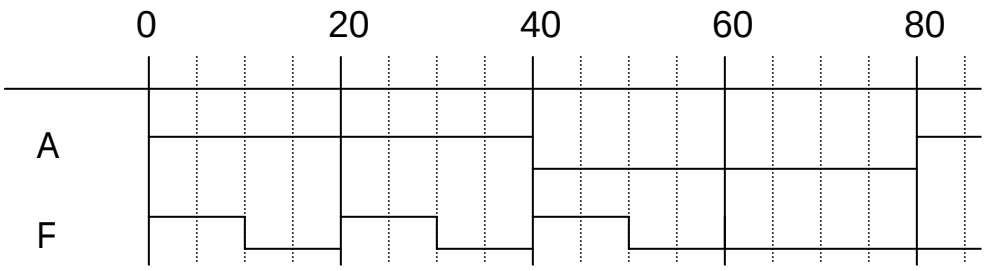
(b) Output response wave form after time T:



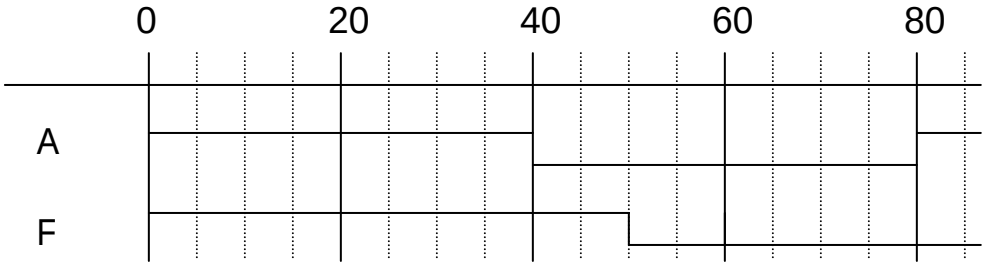
Exercise 3.21

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.



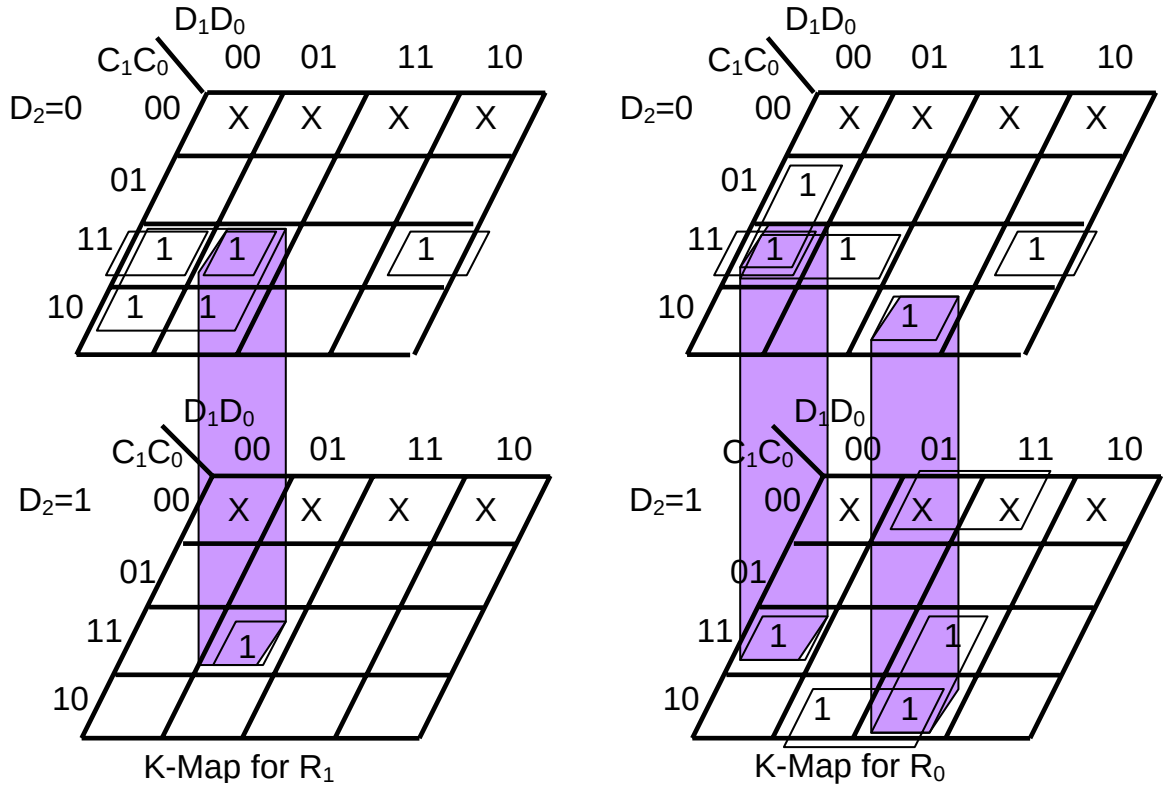
Exercise 3.22

(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

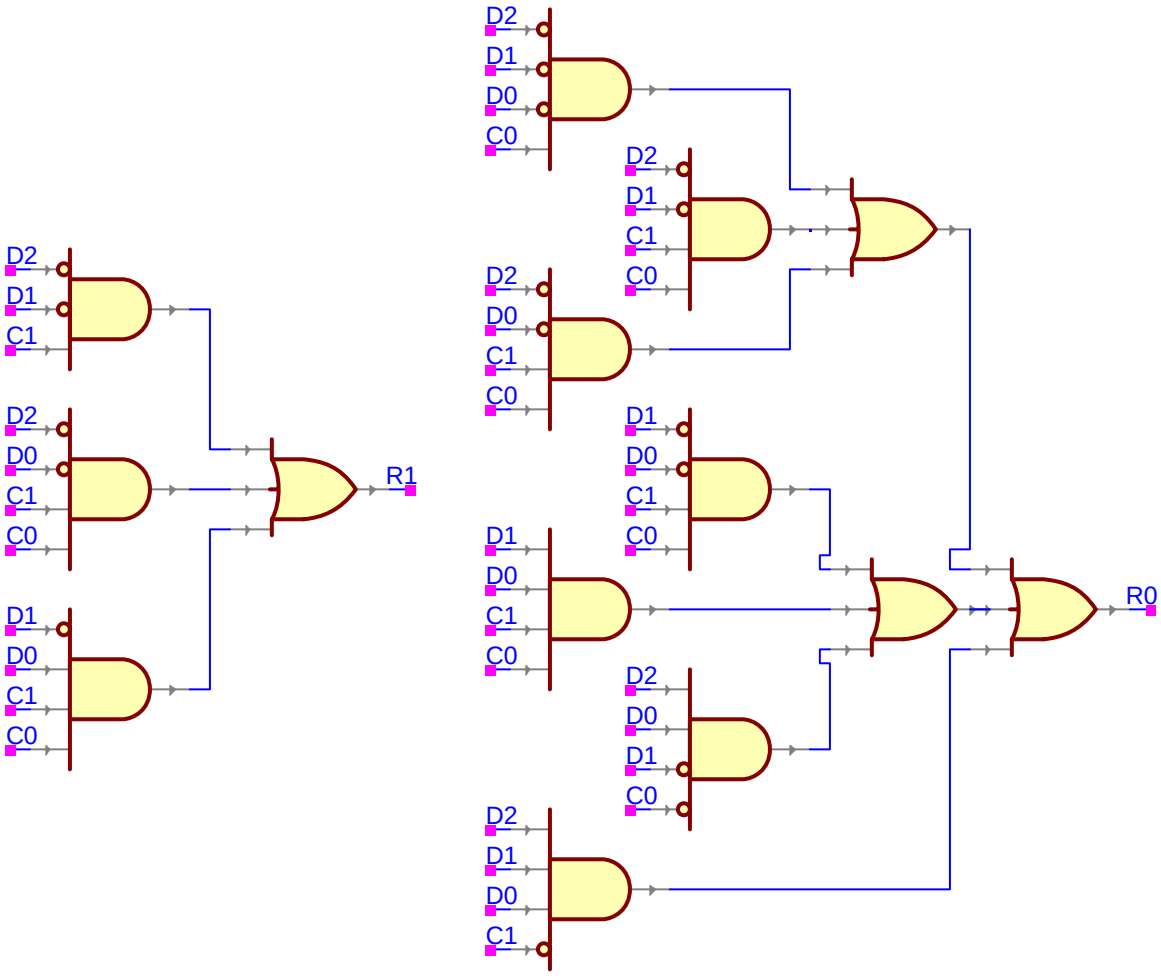
(b) K-Maps for R_1 and R_0:



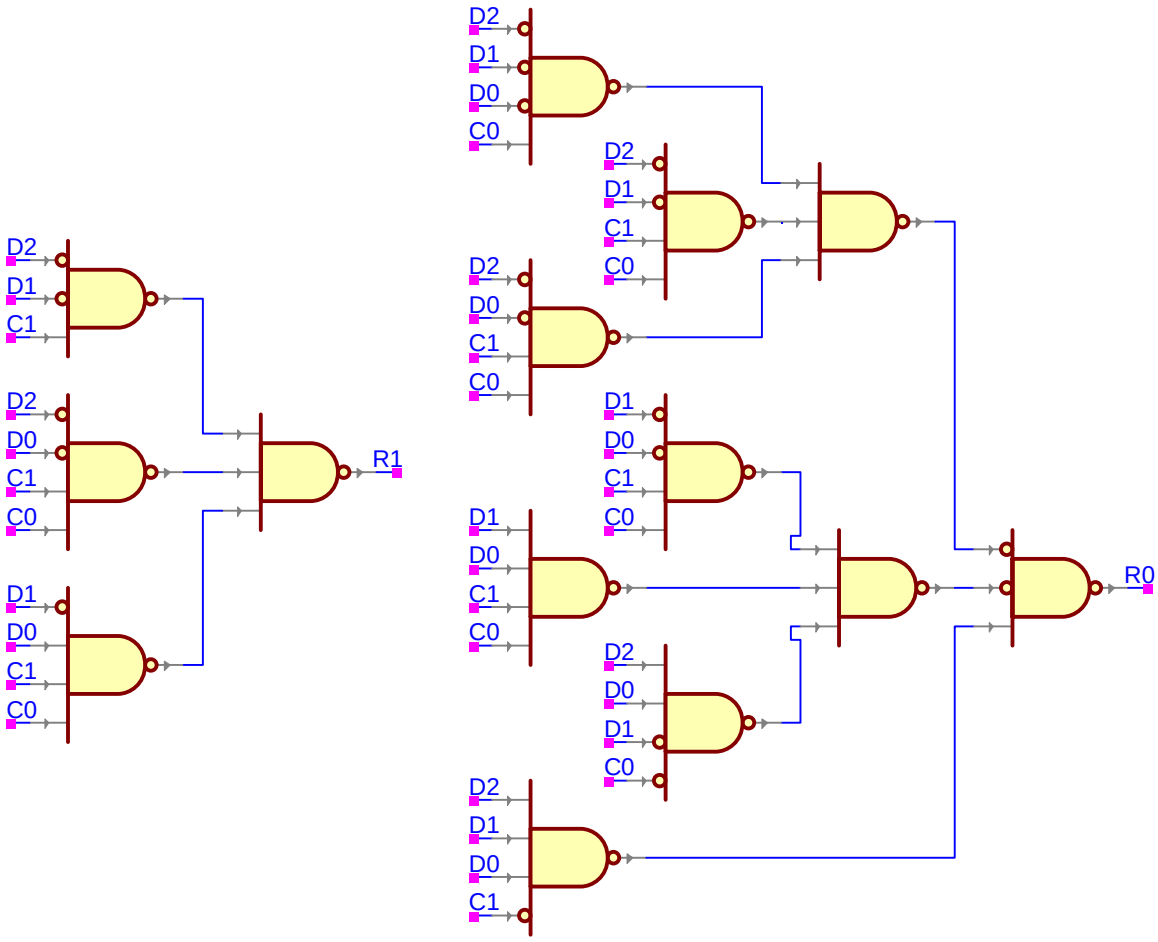
$R_1 = D_2'D_1'C_1 + D_2'D_0'C_1C_0 + D_1'D_0C_1C_0$

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

(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

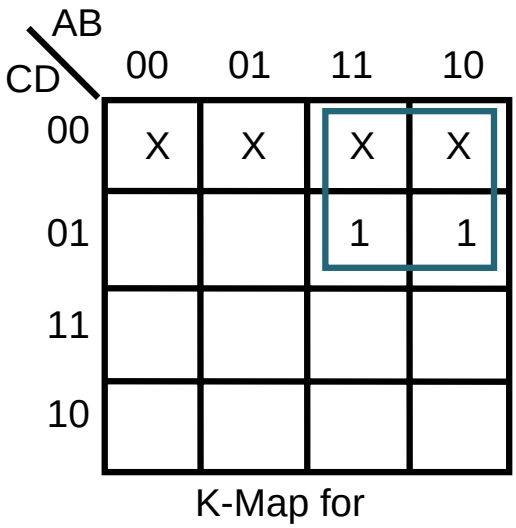
(b) Minimized sum-of-products:

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

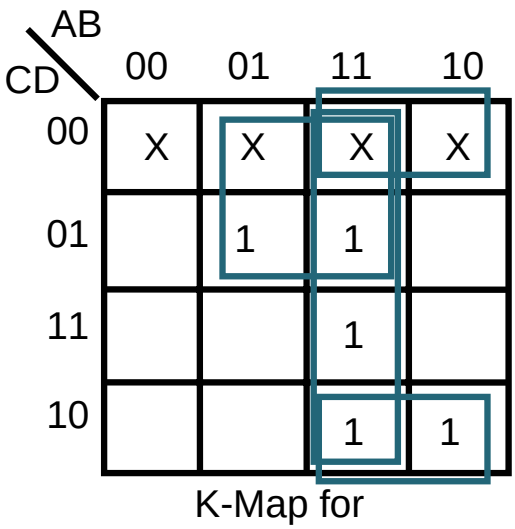
$X(A,B,C,D) = \Sigma\, m(5,10,13,14,15) + \Sigma\, d(0,4,8,12)$

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

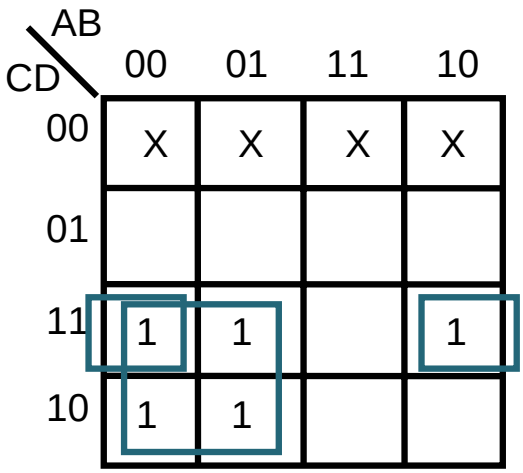
$Z(A,B,C,D) = \Sigma\, m(1,3,7,11,14) + \Sigma\, d(0,4,8,12)$



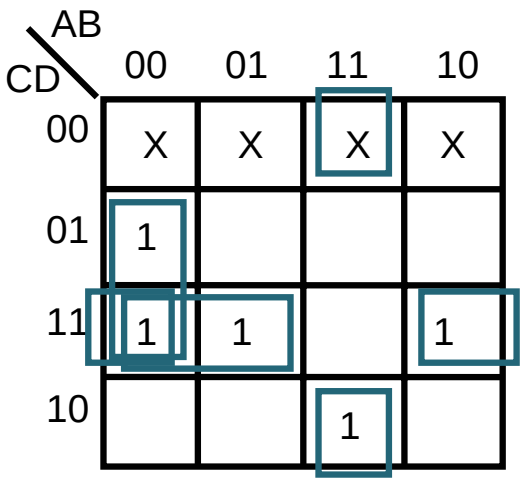
K-Map for



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:

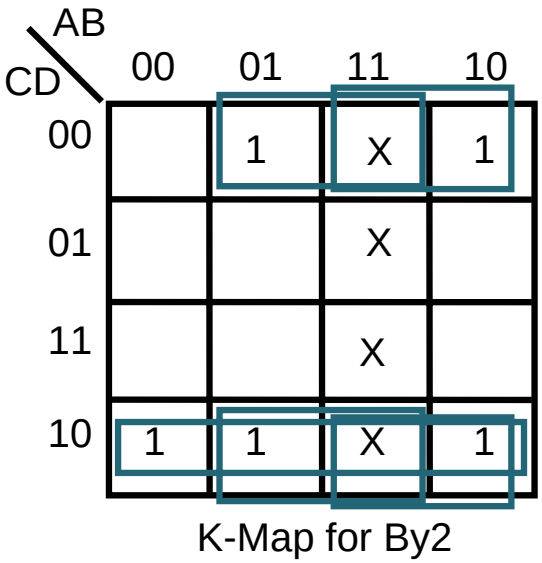
A	B	C	D	By2	By3	By6
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	1	0	0	1	0	0
0	1	0	1	0	0	0
0	1	1	0	1	1	1
0	1	1	1	0	0	0
1	0	0	0	1	0	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	0	1	1	0	0	0
1	1	0	0	X	X	X
1	1	0	1	X	X	X
1	1	1	0	X	X	X
1	1	1	1	X	X	X

(b) Minimized sum-of-products:

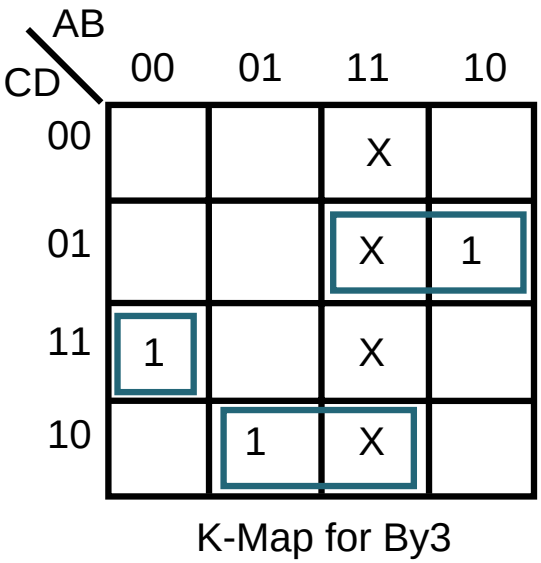
$By2 = \Sigma\, m(2,4,6,8,10) + \Sigma\, d(12,13,14,15)$

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

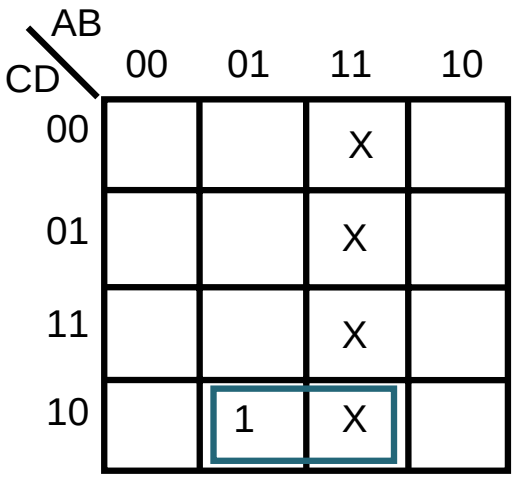
$By6 = \Sigma\, m(6) + \Sigma\, d(12,13,14,15)$



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:

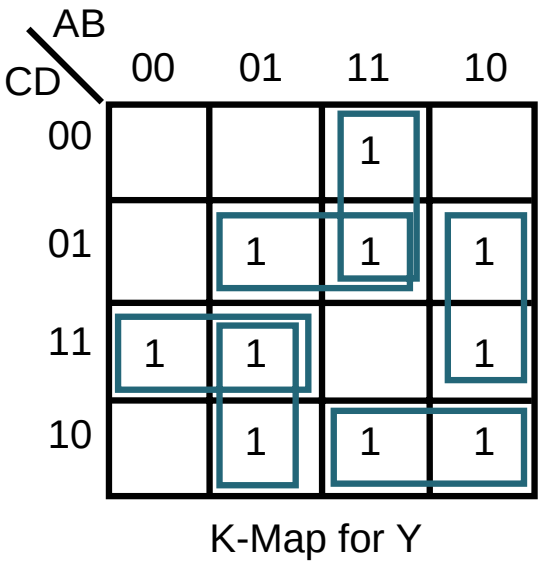
A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	0
0	1	1	1	0	1	1
1	0	0	0	0	0	1
1	0	0	1	0	1	0
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	0	1	0
1	1	0	1	0	1	1
1	1	1	0	0	1	1
1	1	1	1	1	0	0

(b) Minimized sum-of-products:

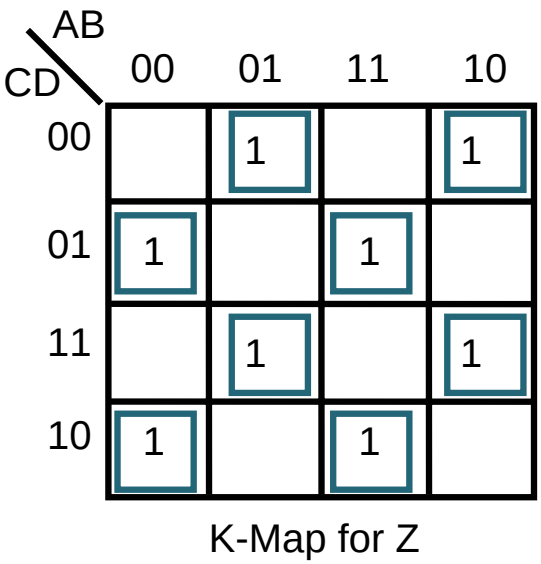
$X(A,B,C) = \Sigma\, m(15)$

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

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



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:

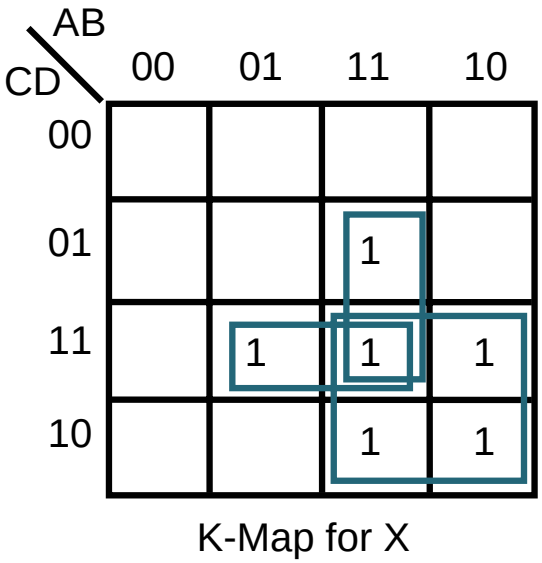
A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

(b) Minimized sum-of-products:

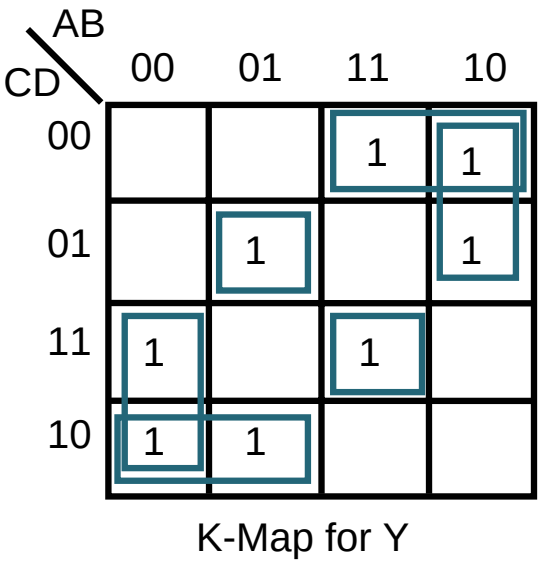
$X(A,B,C,D) = \Sigma\, m(7,10,11,13,14,15)$

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

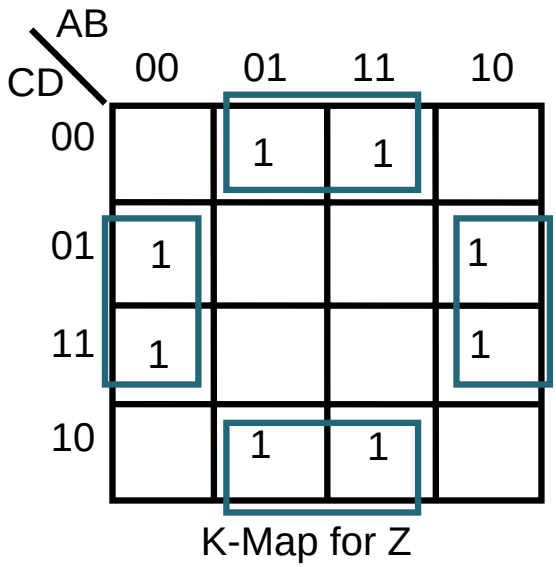
$Z(A,B,C,D) = \Sigma\, m(1,3,4,6,9,11,12,14)$



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

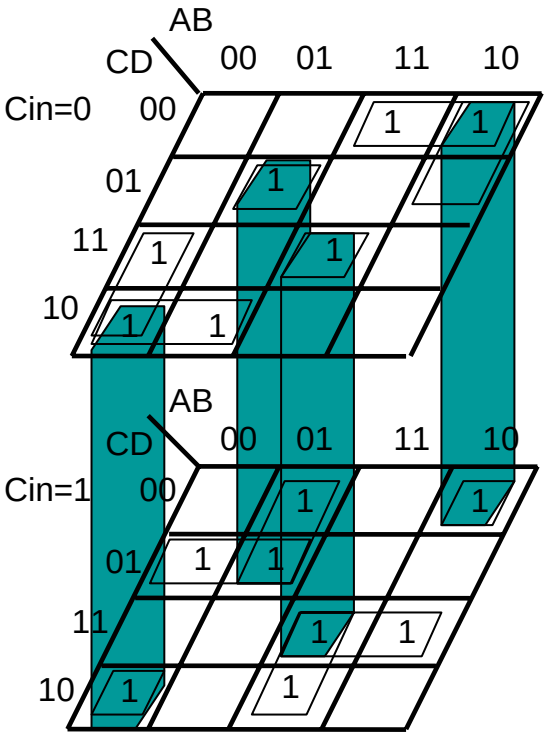
Z(A,B,C,D) = BD' + B'D

(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.

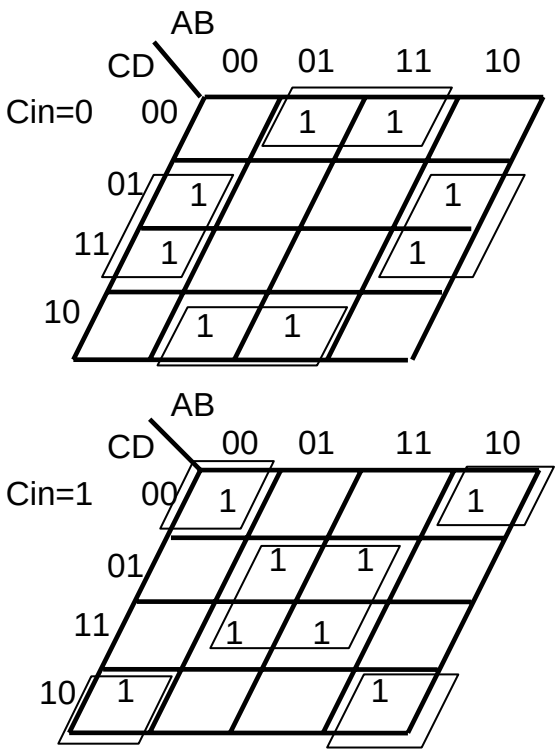
Exercise 3.27

Truth table for the generalized 2-bit adder:

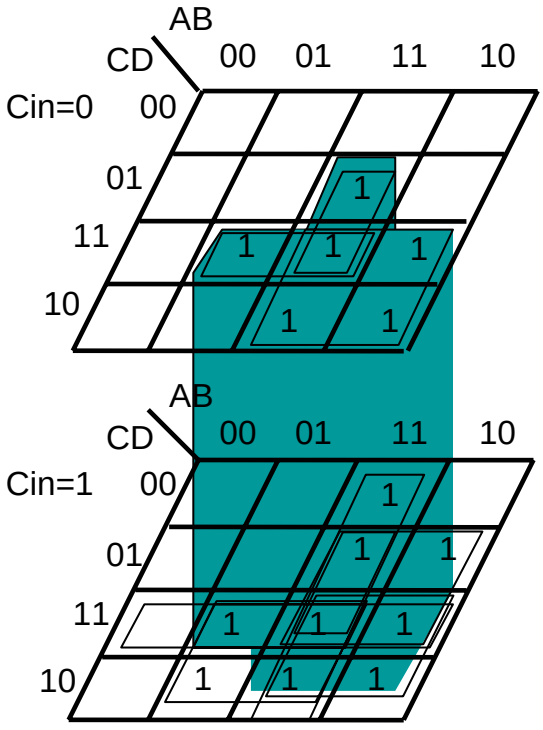
Cin	A	B	C	D	Cout	Y	Z
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	0	1	1	0	1	1
0	0	1	0	0	0	0	1
0	0	1	0	1	0	1	0
0	0	1	1	0	0	1	1
0	0	1	1	1	1	0	0
0	1	0	0	0	0	1	0
0	1	0	0	1	0	1	1
0	1	0	1	0	1	0	0
0	1	0	1	1	1	0	1
0	1	1	0	0	0	1	1
0	1	1	0	1	1	0	0
0	1	1	1	0	1	0	1
0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1
1	0	0	0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

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’)’]]’”

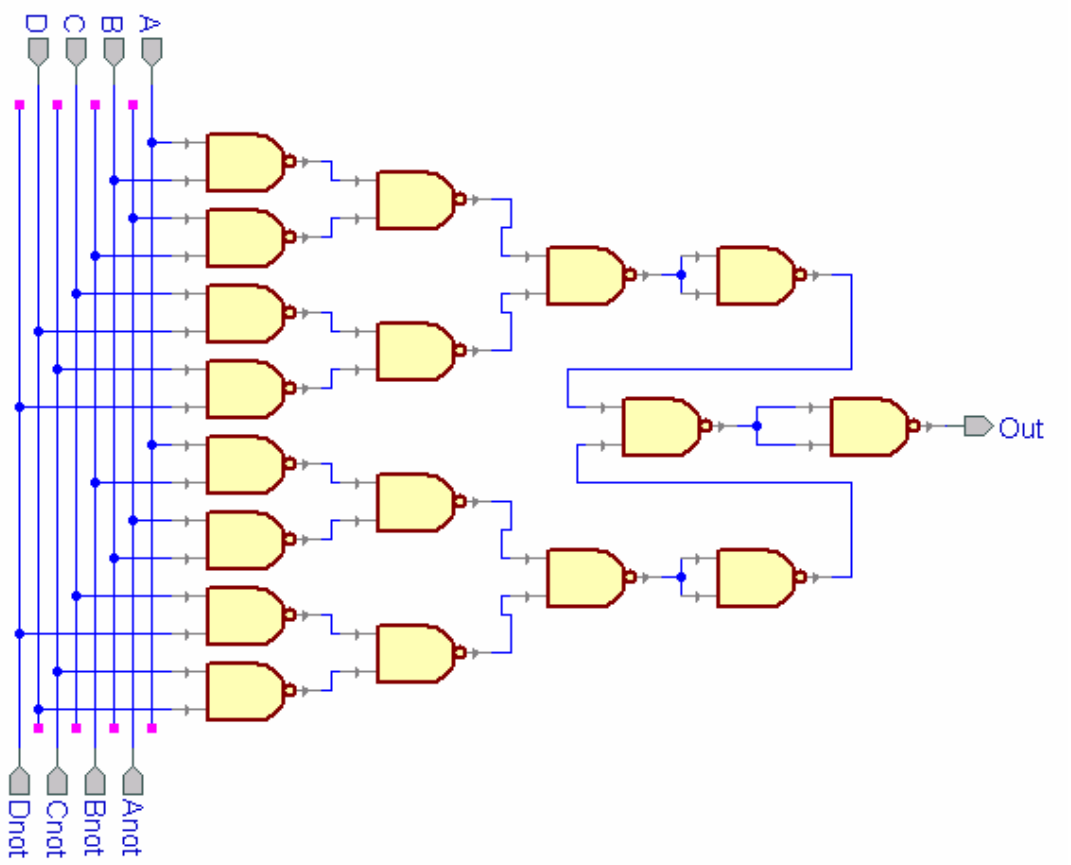
= [[(A’B’)’ (AB)’ + (C’D’)’ (CD)’] [(A’B)’ (AB’)’ + (C’D)’ (CD’)’]]’”

= [[(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)

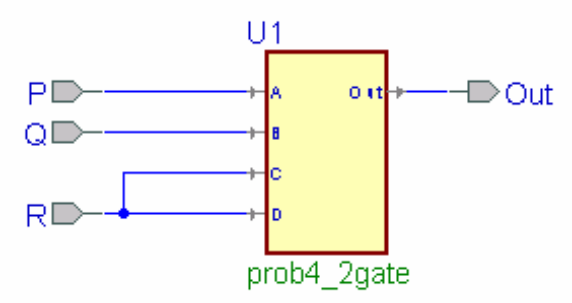
= (A’ ⊕ B) ⊕ (C ⊕ D)

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

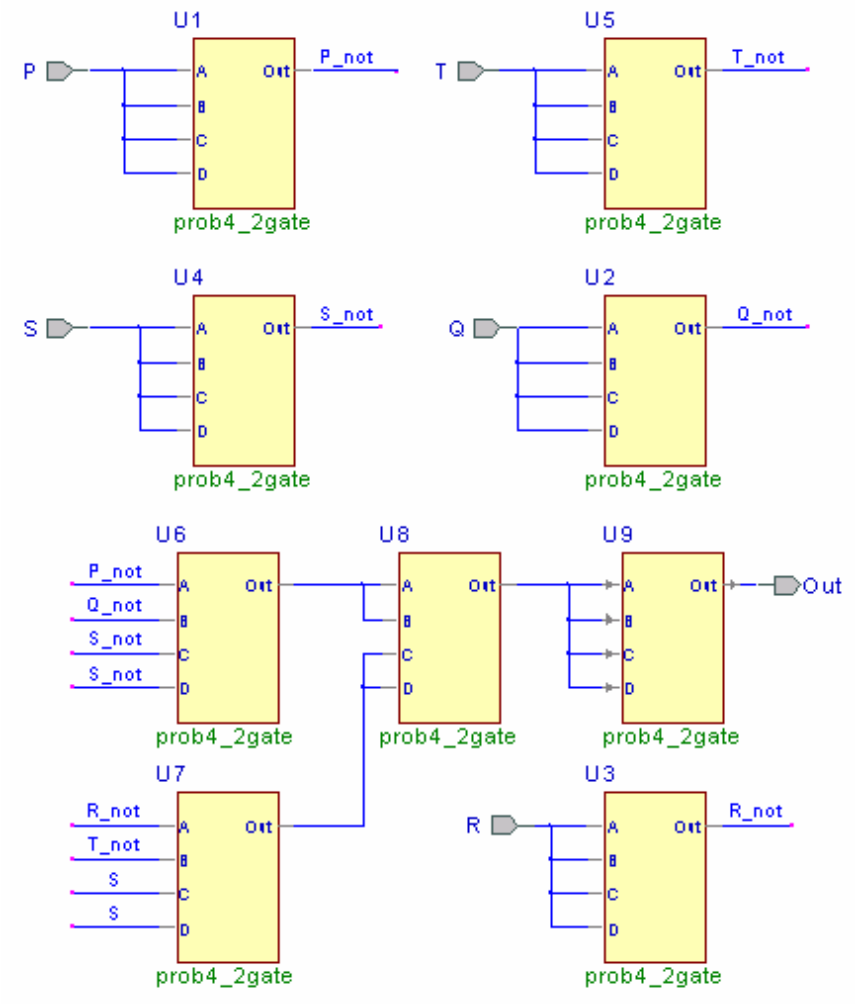
Exercise 4.2

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

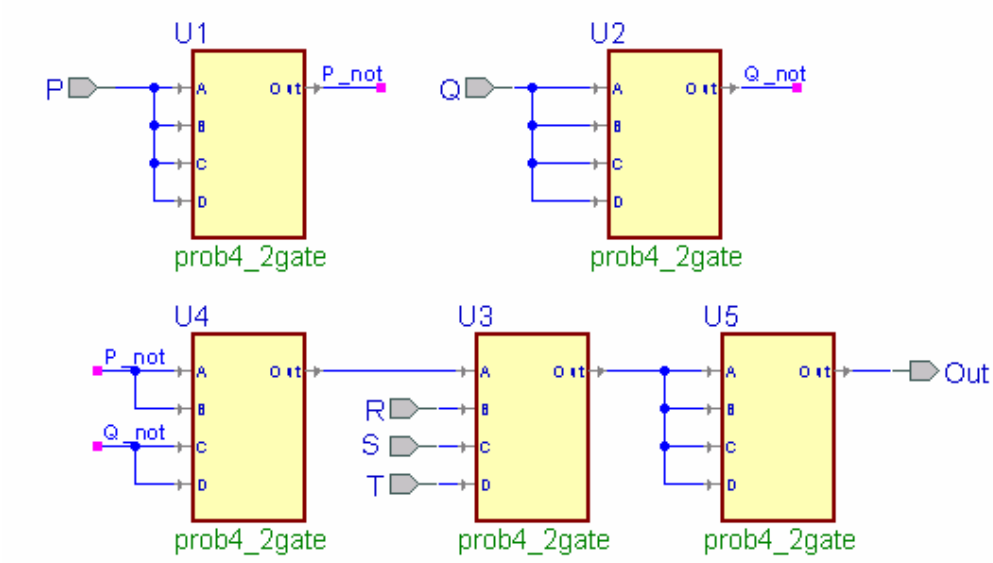
(a)



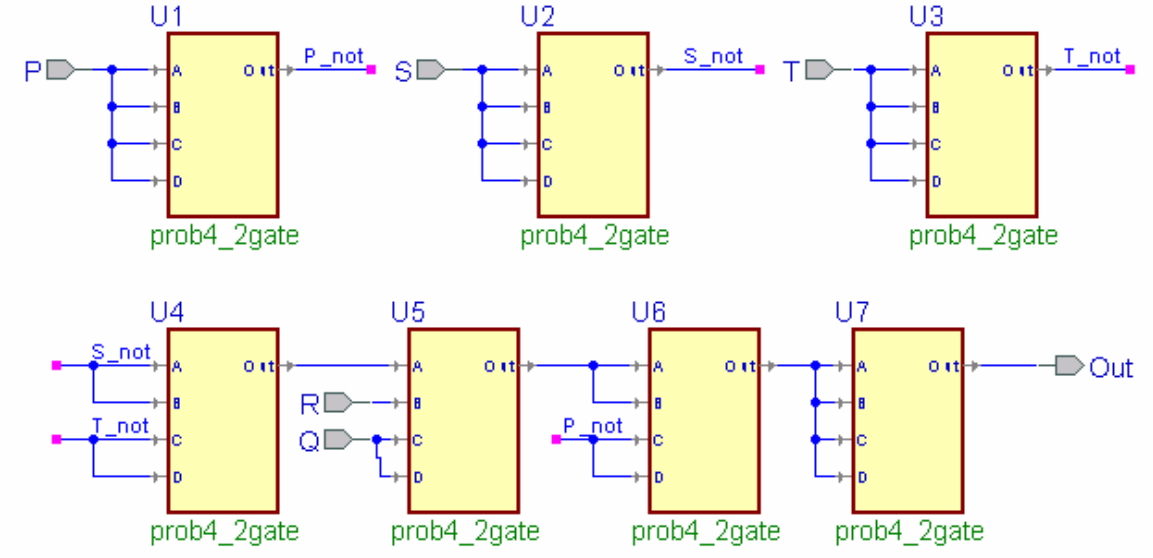
(b)



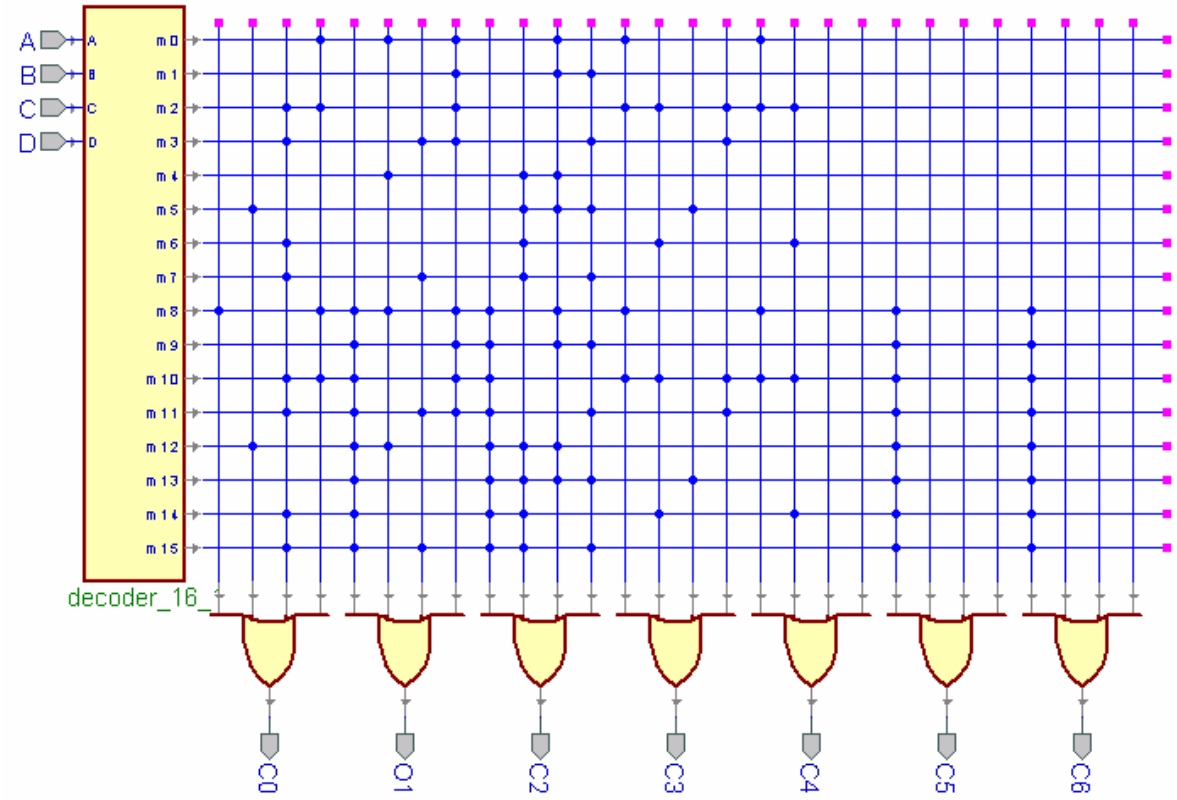
(c)


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

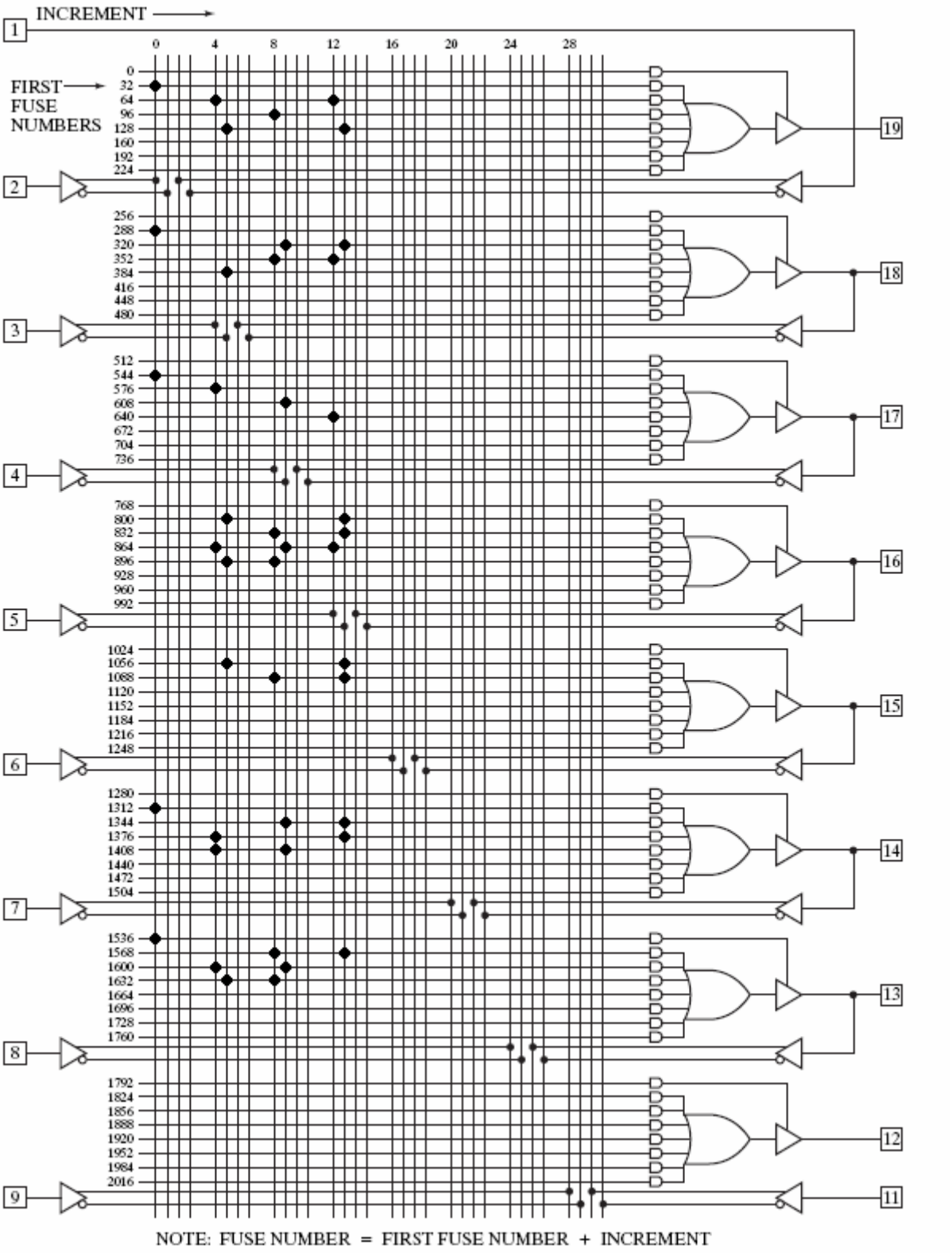
(d)


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.

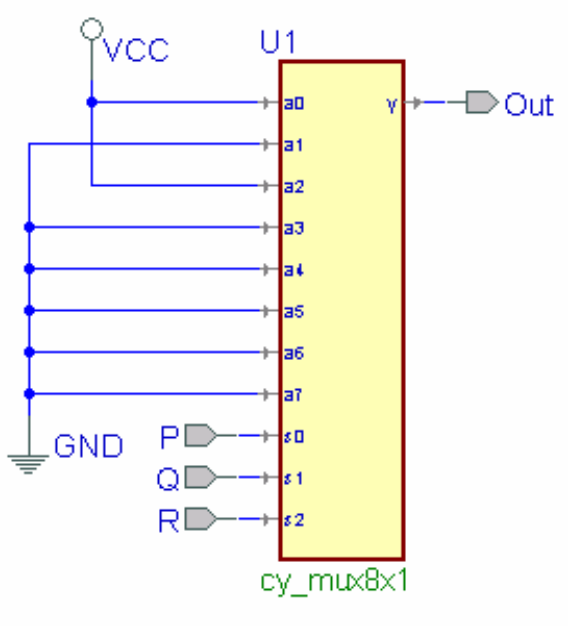
Exercise 4.6

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.

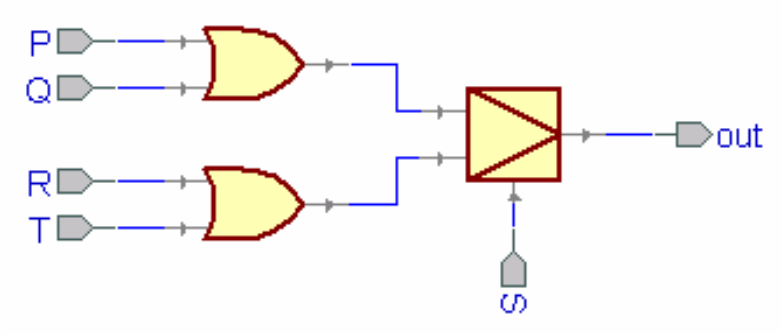
The advantage of the PLA implementation is that both of these strategies can be combined in order to reduce the number of AND gates and the number of OR gates. However, a PLA is generally slower because the programmable planes tend to slow the circuit down a bit.

Exercise 4.7

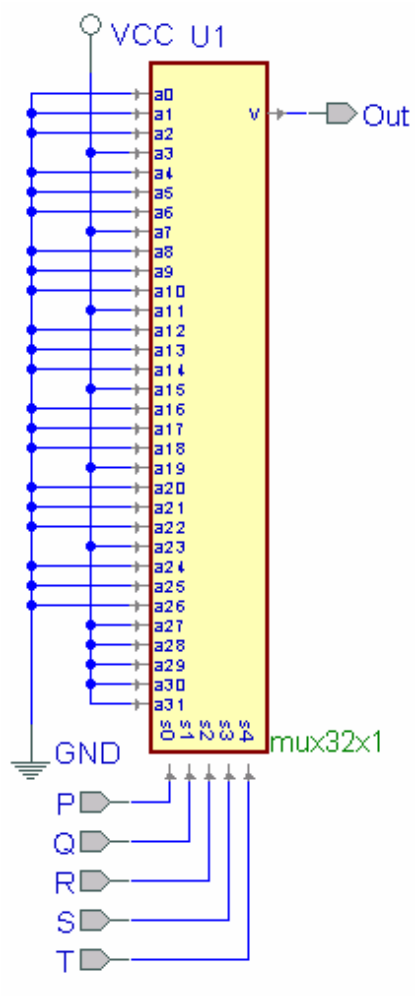
(a)



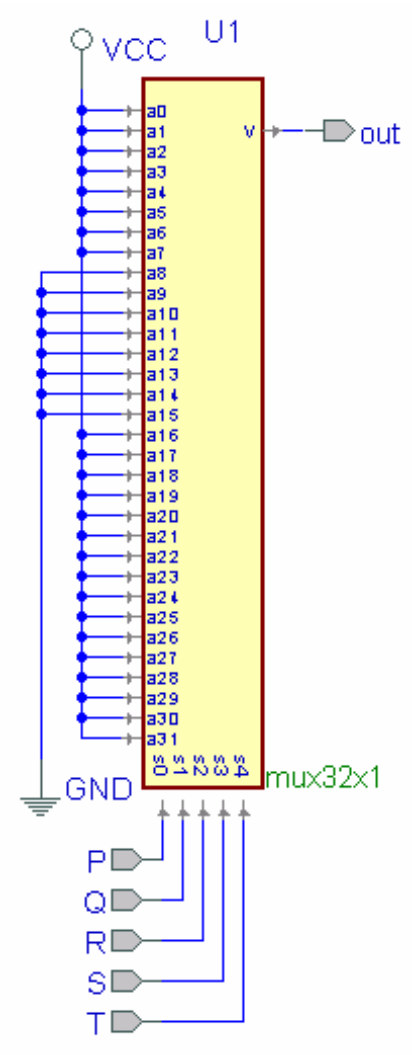
(b)



(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

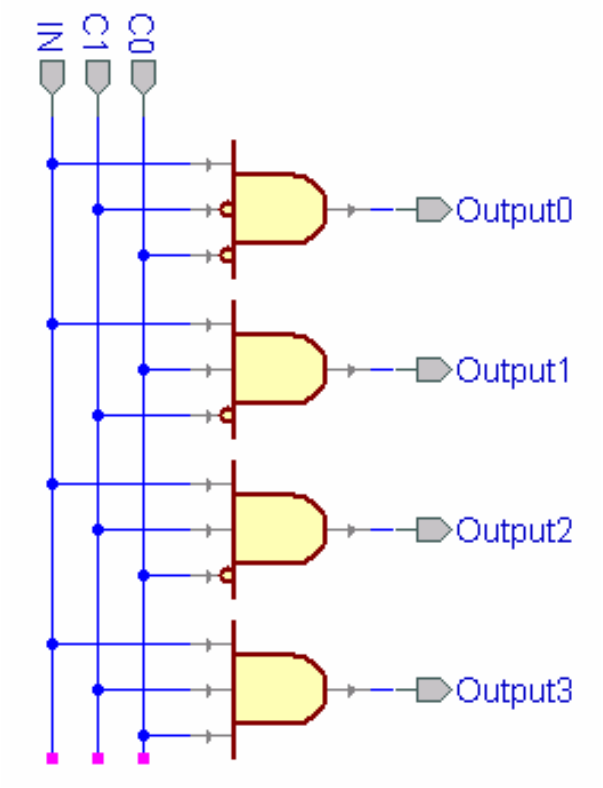
(d)


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

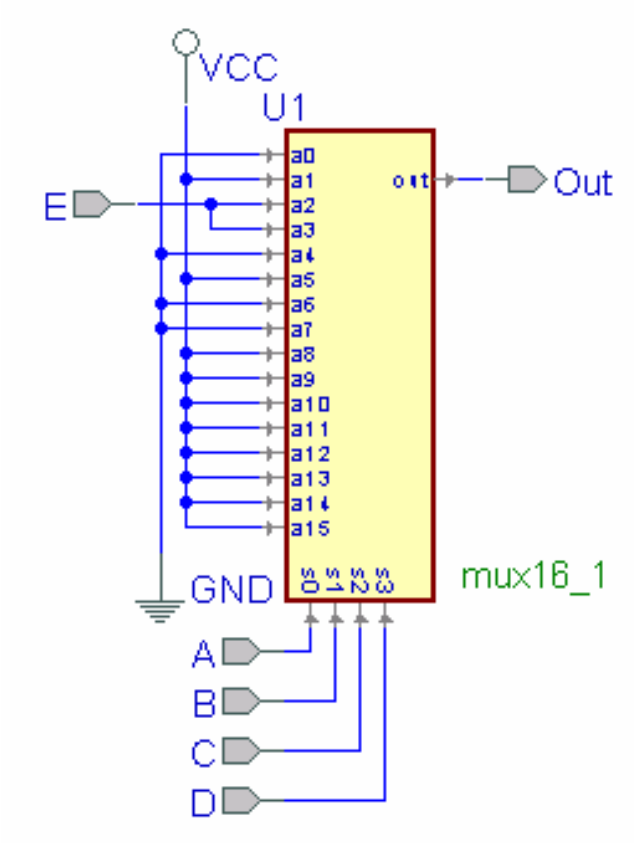
Exercise 4.8

(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.

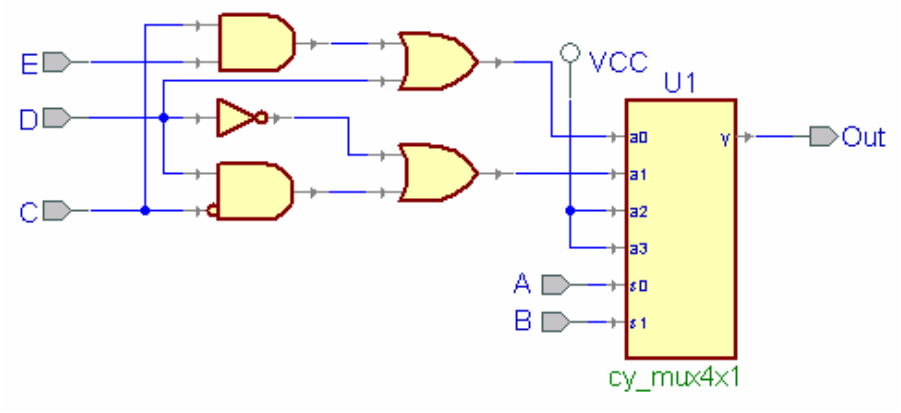


Exercise 4.9

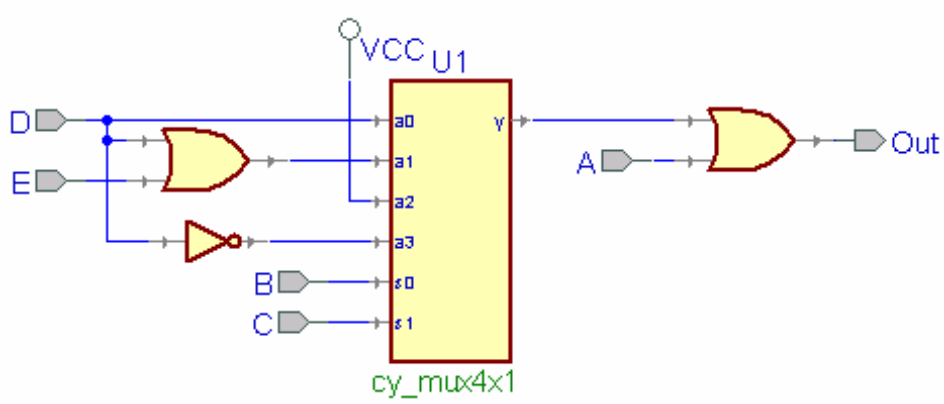

VCC
U1
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
a10
a11
a12
a13
a14
a15
out
Out
E
GND
s0
s1
s2
s3
mux16_1
A
B
C
D

Exercise 4.10

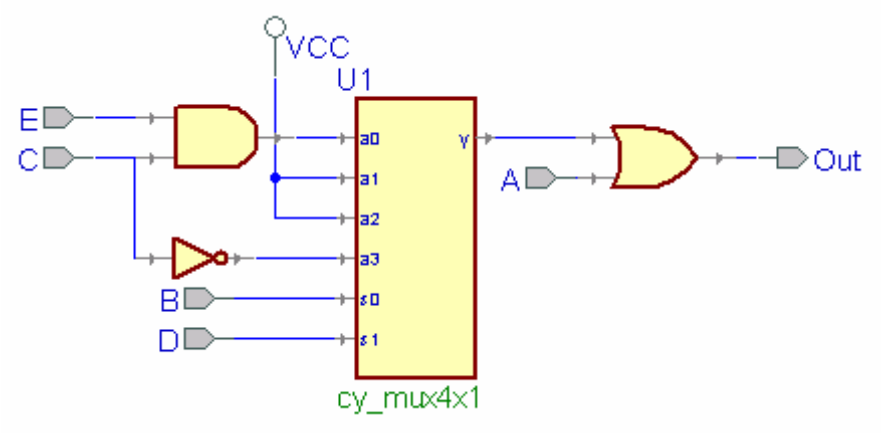
(a)



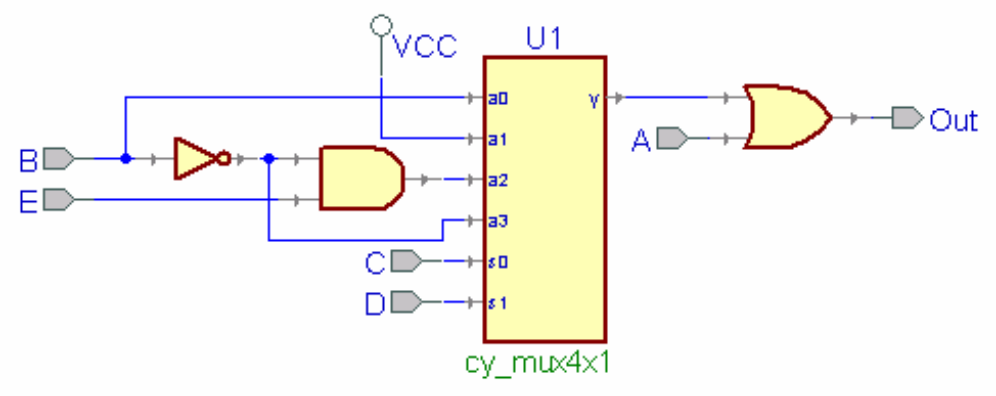
(b)



(c)



(d)



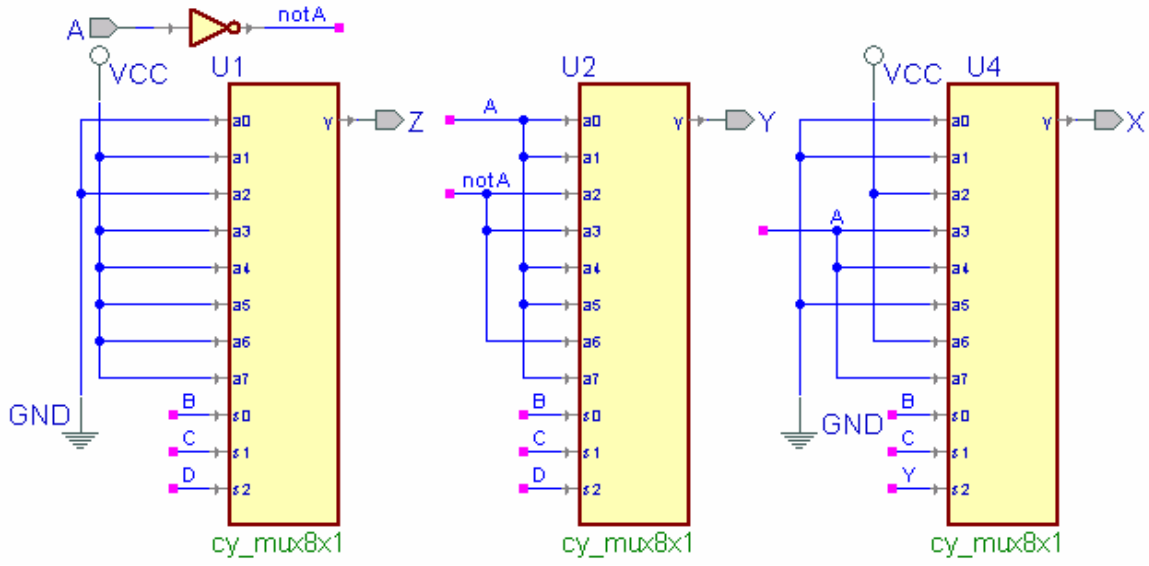
Exercise 4.11

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

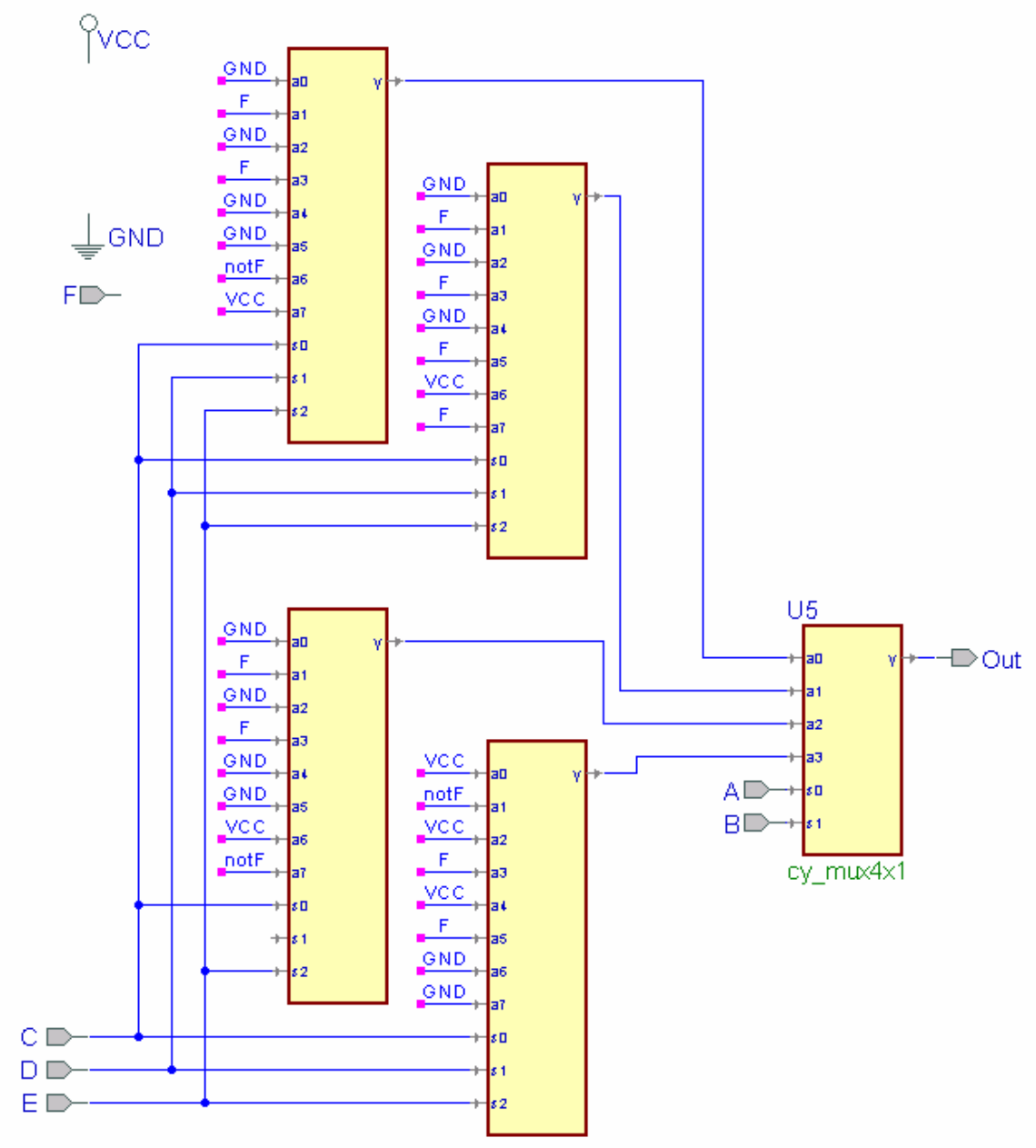
Y = A’B’CD’ + A’B’CD + A’BC’D + A’BCD’ + AB’C’D’ + AB’C’D + ABC’D’ + ABCD

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



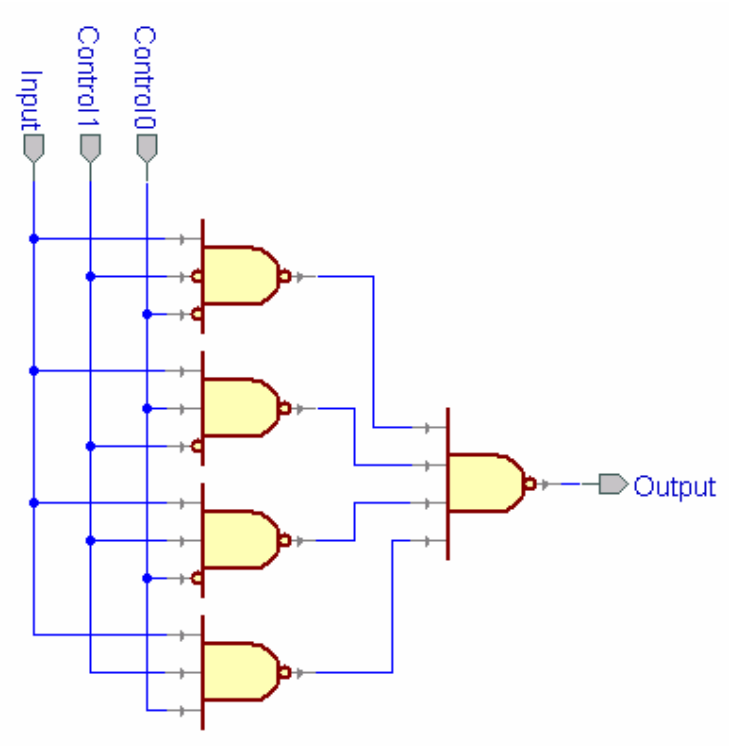
Exercise 4.12

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).

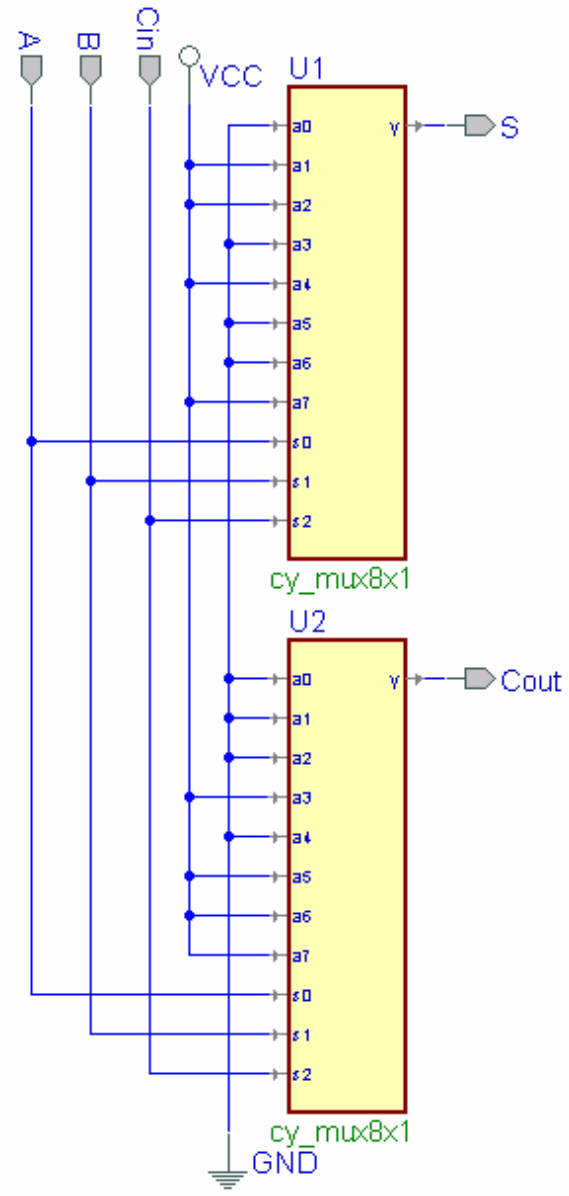


The table below demonstrates how many of each type of gate is required, and how many packages for each.

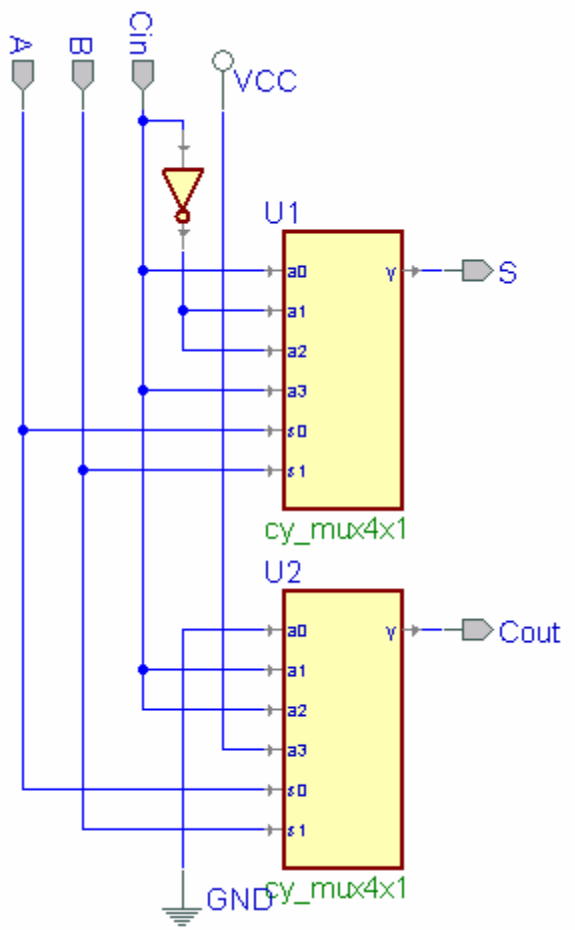
Gate Type	Number of Gates	Number of Packages
Inverters	6	1
2-input NAND	3	1
3-input NAND	40	14
4-input NAND	10	5
TOTAL	**59**	**21**

Exercise 4.13

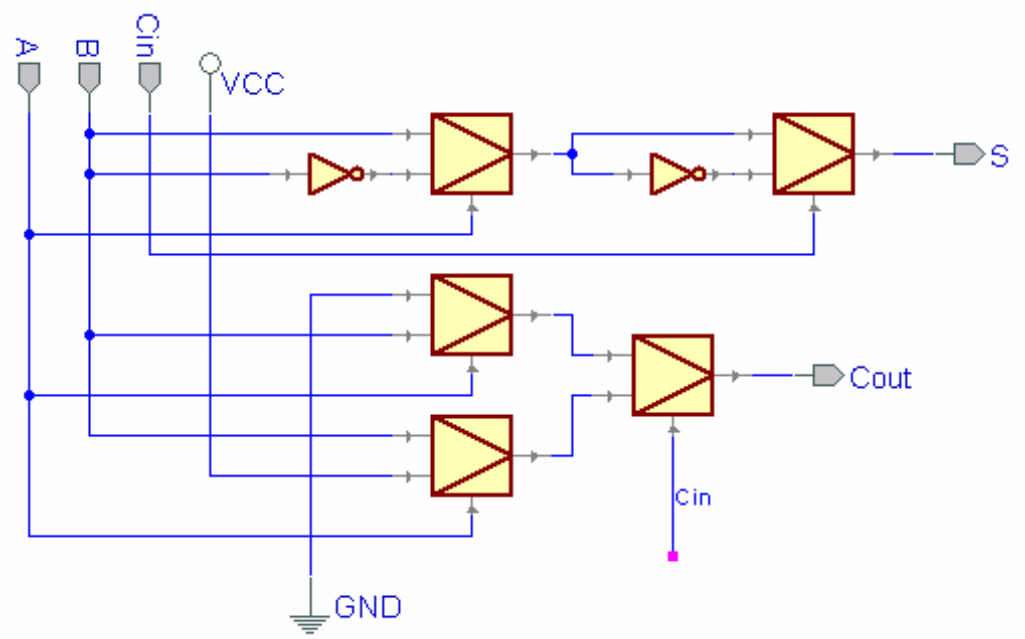
(a)



(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

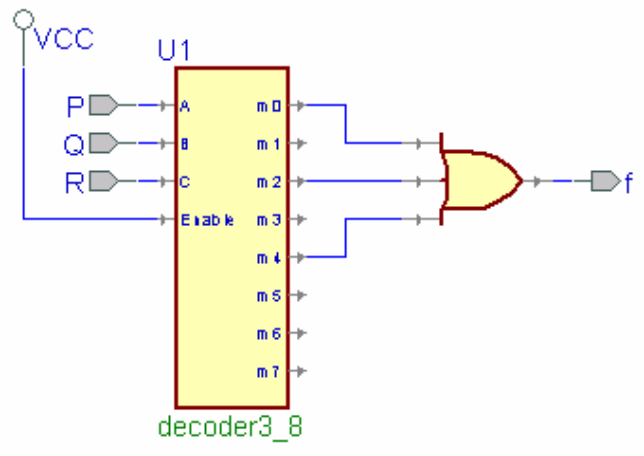
(c) Only five 2:1 multiplexers are needed to implement the function:



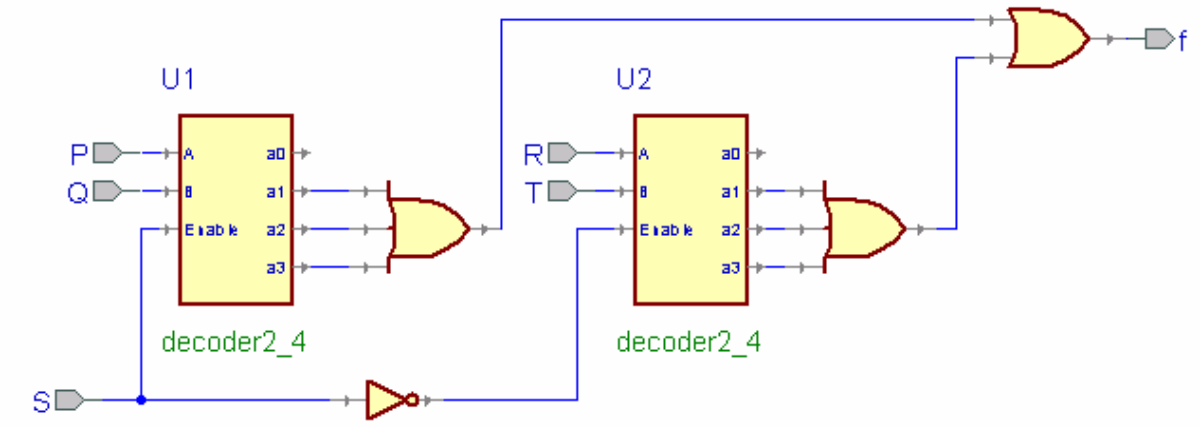
(d) Yes it is possible as shown in part (c).

Exercise 4.14

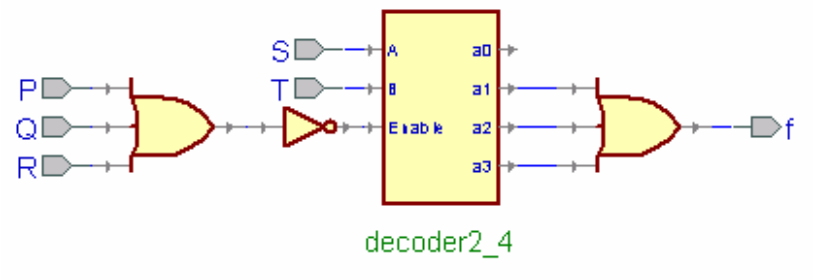
(a)



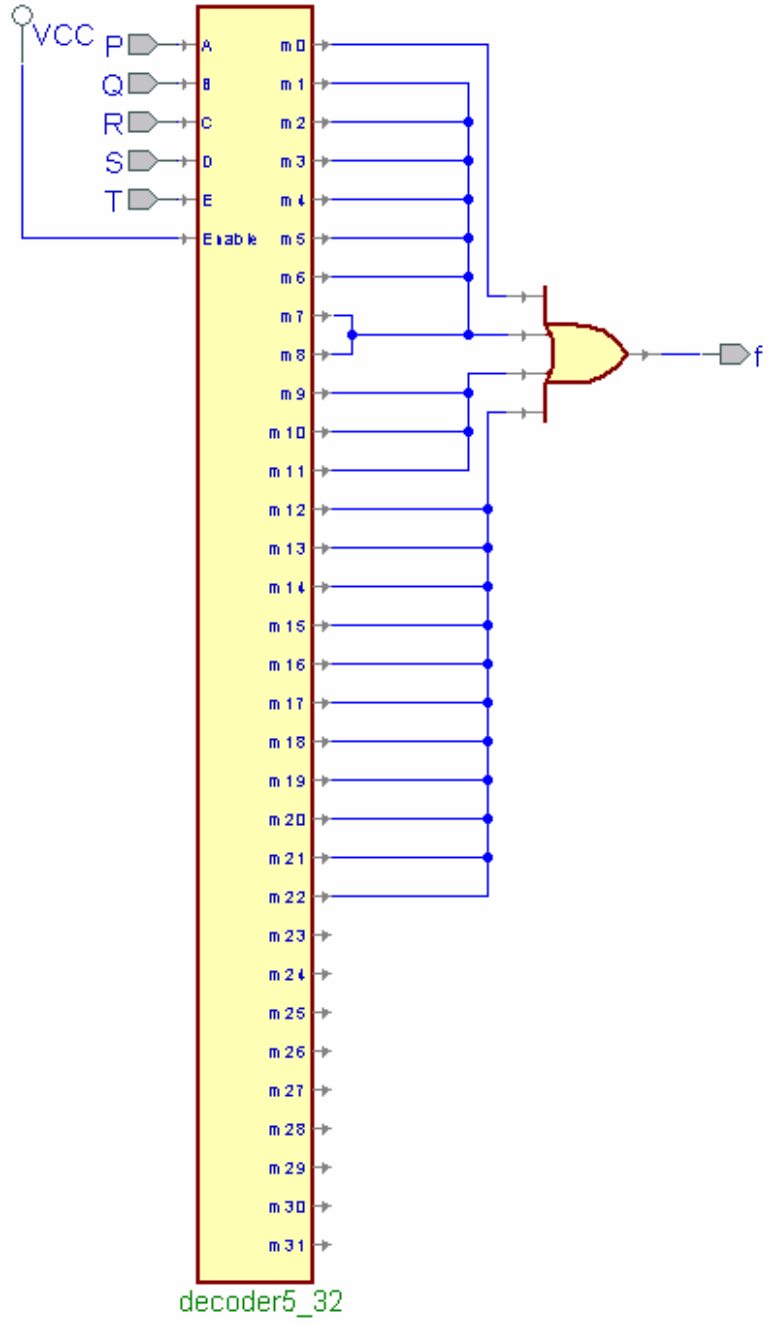
(b)



(c)

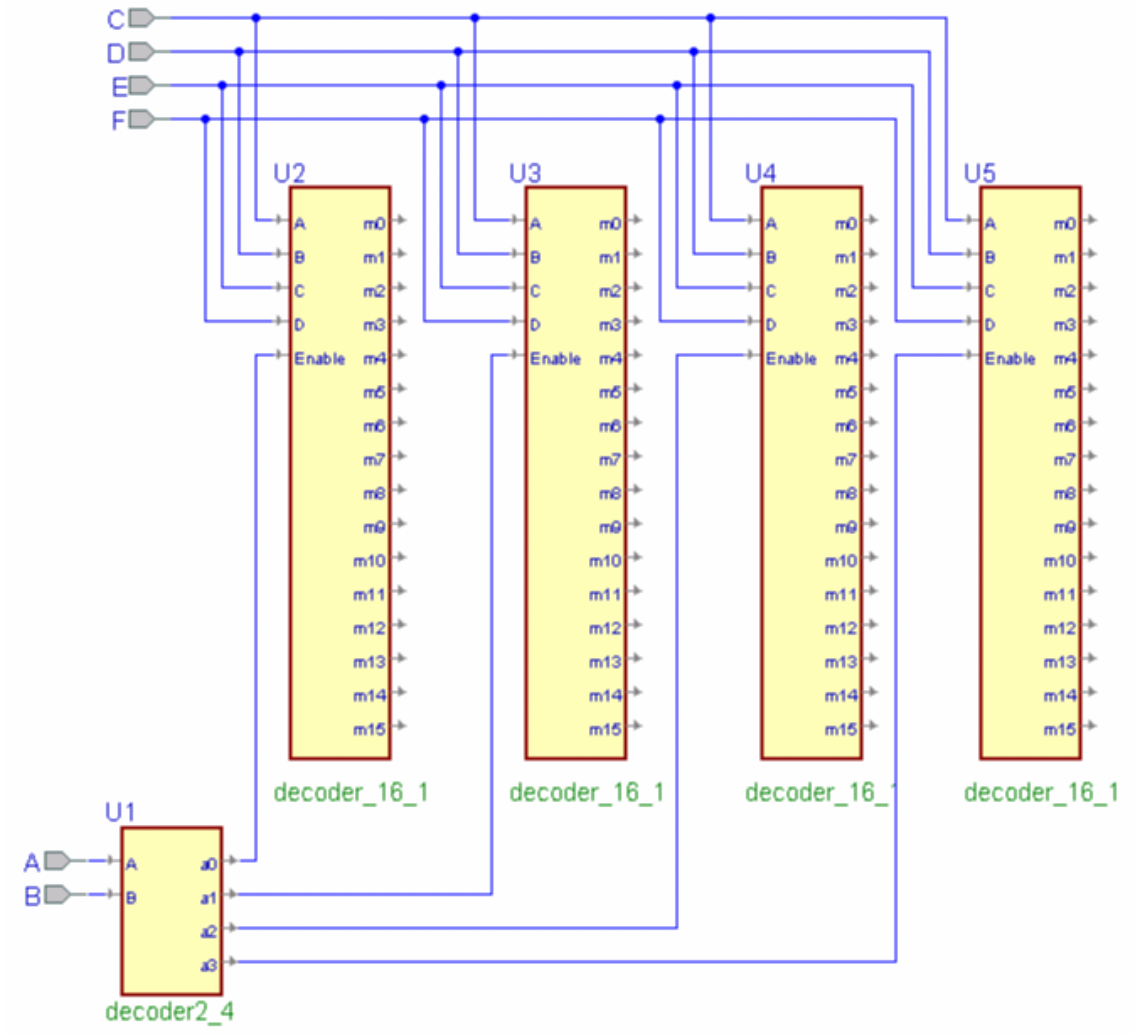


(d) Note: in order to simplify the diagram a single or gate is used. In this case the dots connecting output wires should be interpreted as separate connections to the OR gate since the drawing tool used does not have an or gate with sufficient fan-in. When a 23-input OR gate is not available, using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

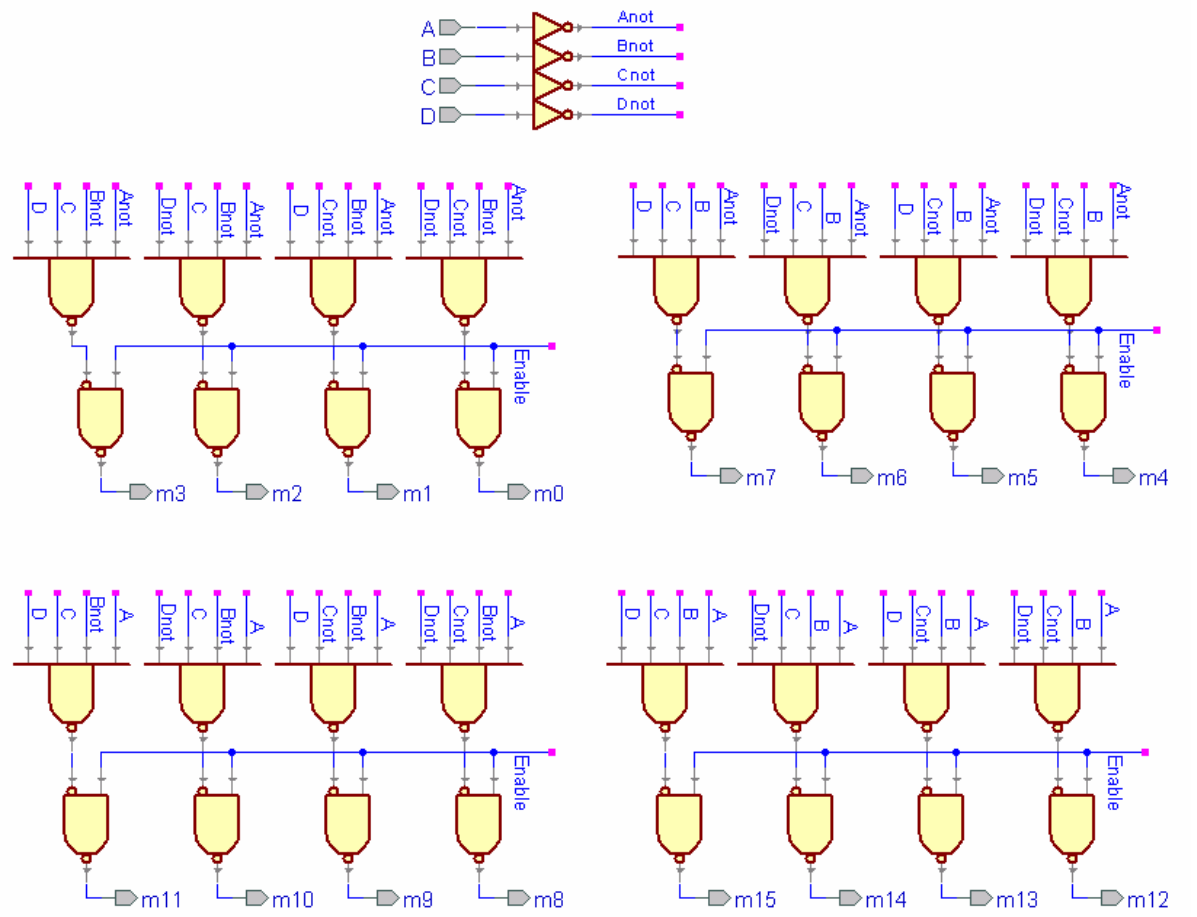


Exercise 4.15

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.

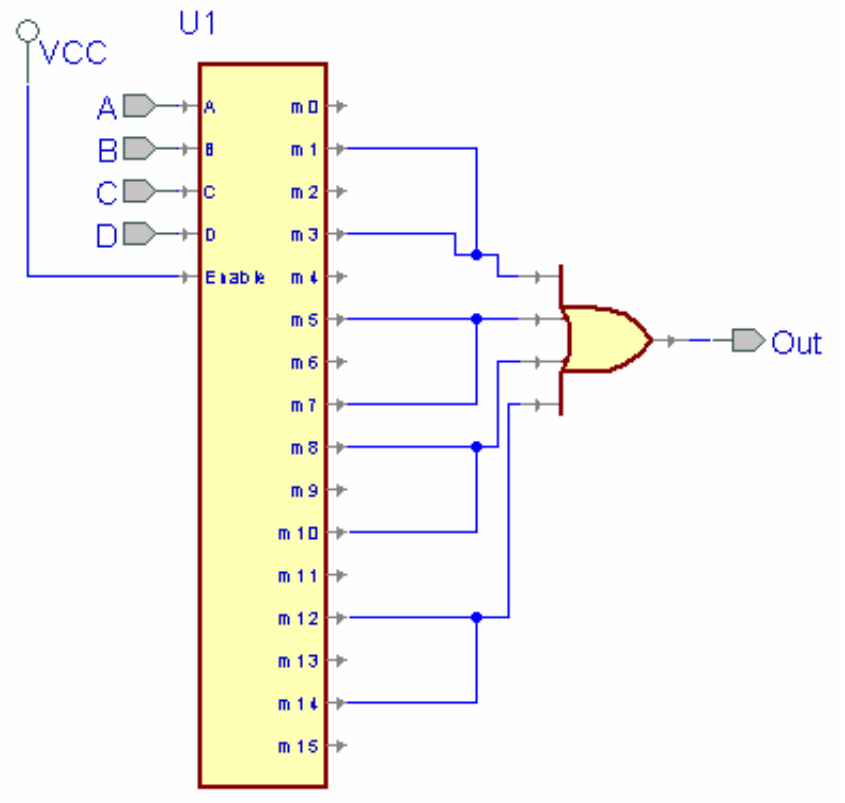


Exercise 4.16



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'

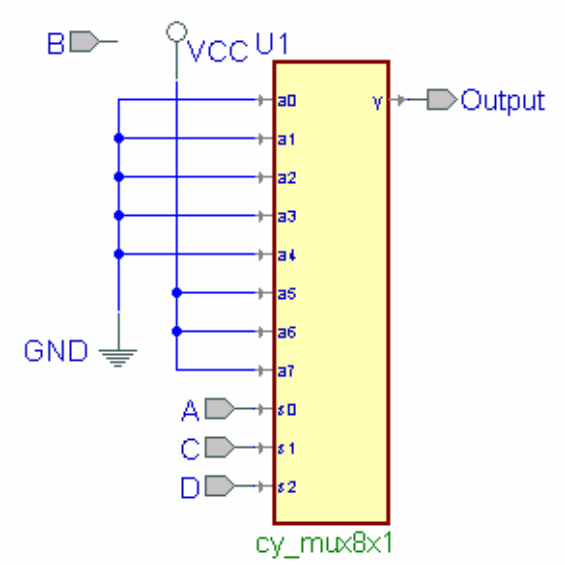
= A'D + AD'

= A ⊕ D

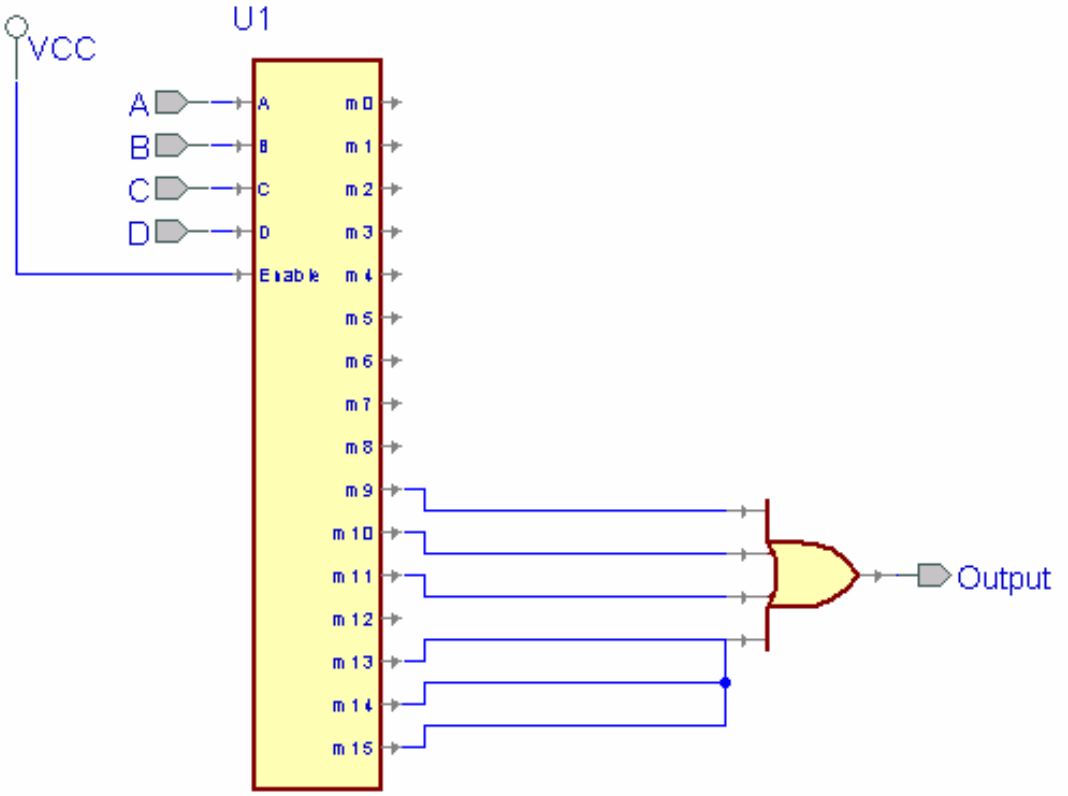
This can be implemented using a single discrete gate which has significantly less fan-in than the 8-input OR gate, and this is accomplished with only one level of logic instead of two.

Exercise 4.18

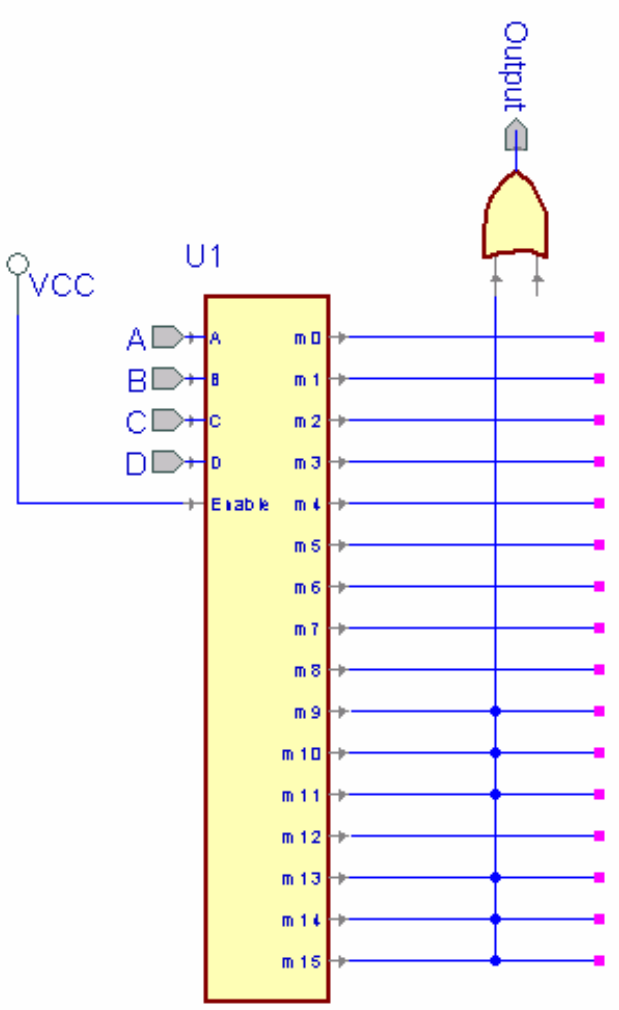
(a) By simplifying the function, it is realized that AB'C is already covered by AC, so B is not needed as an input to the multiplexer.



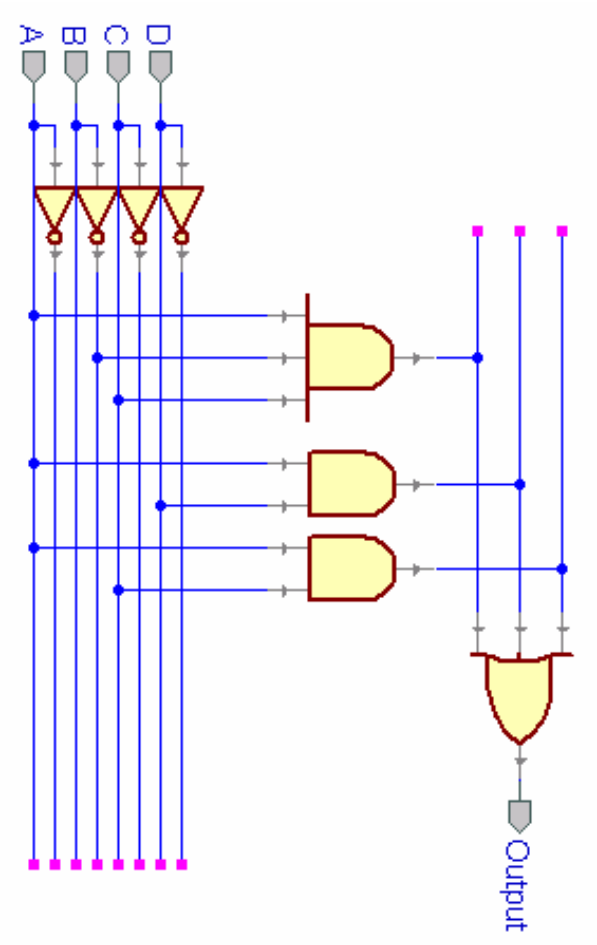
(b)



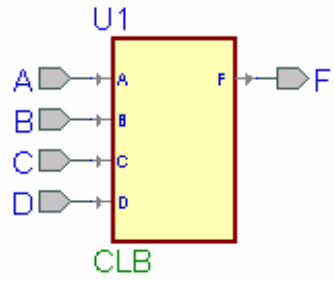
(c)


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

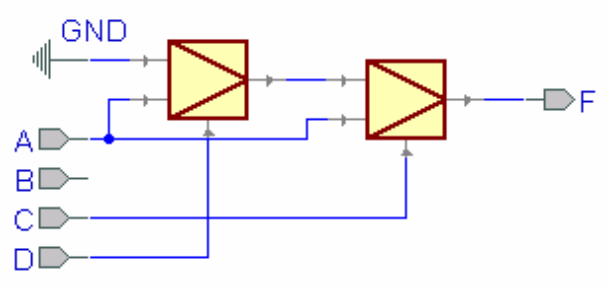
(d)



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

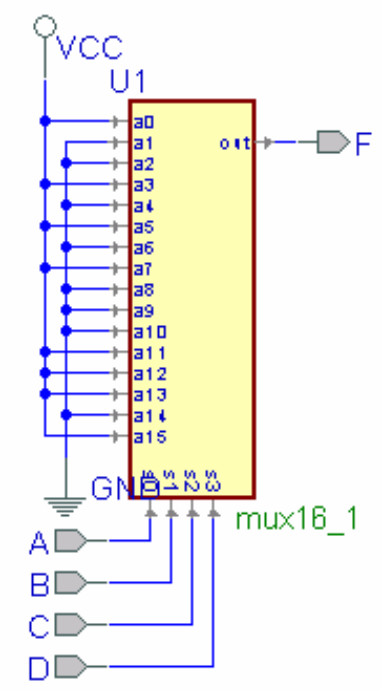


(f)

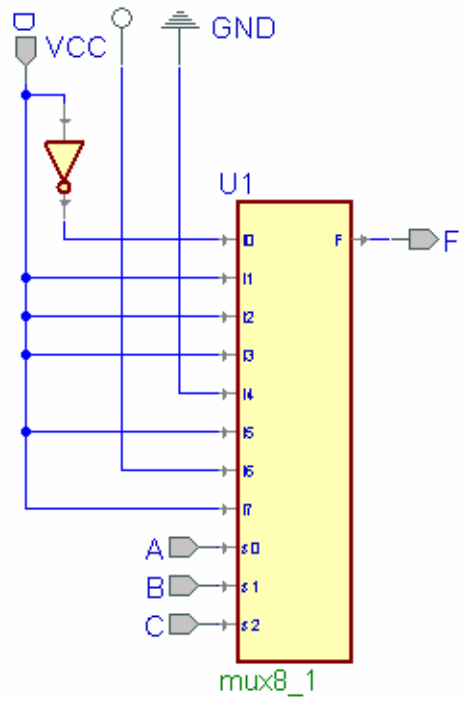


Exercise 4.19

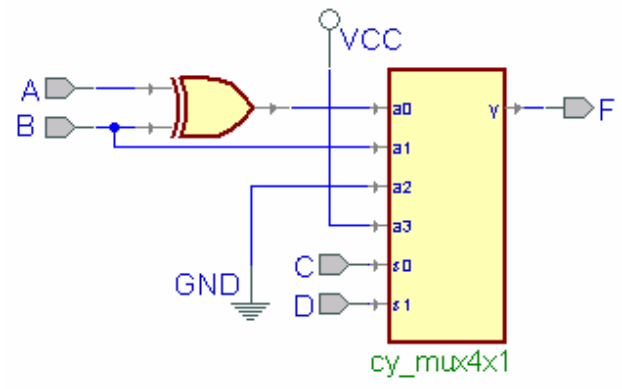
(a)



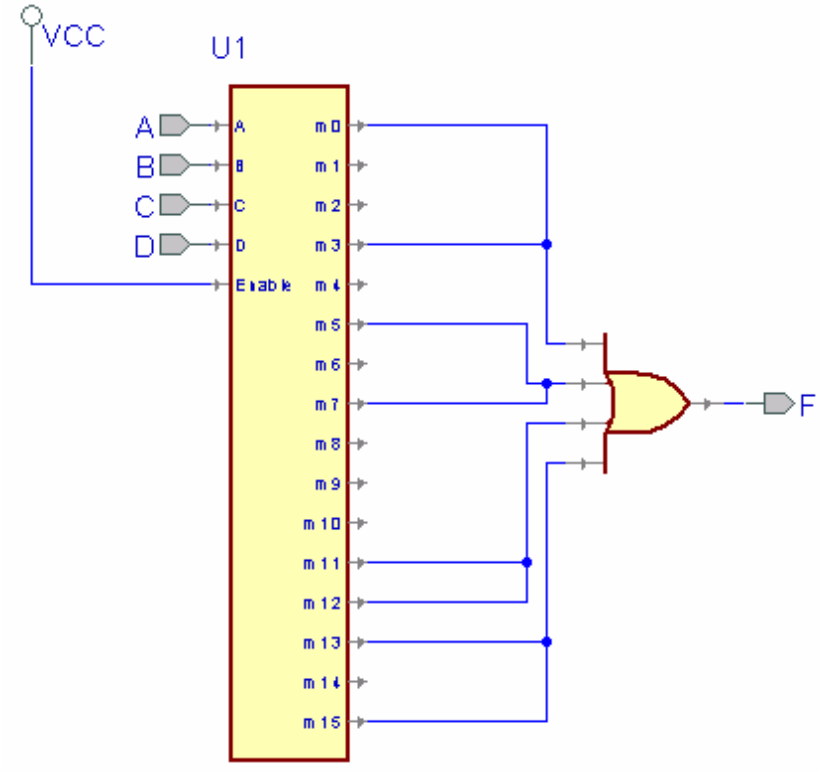
(b)



(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)



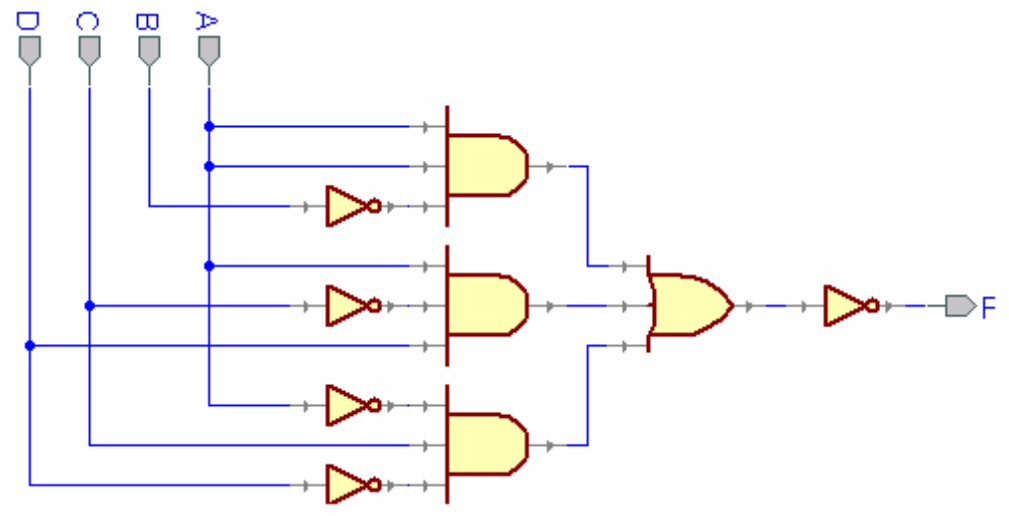
Exercise 4.20

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

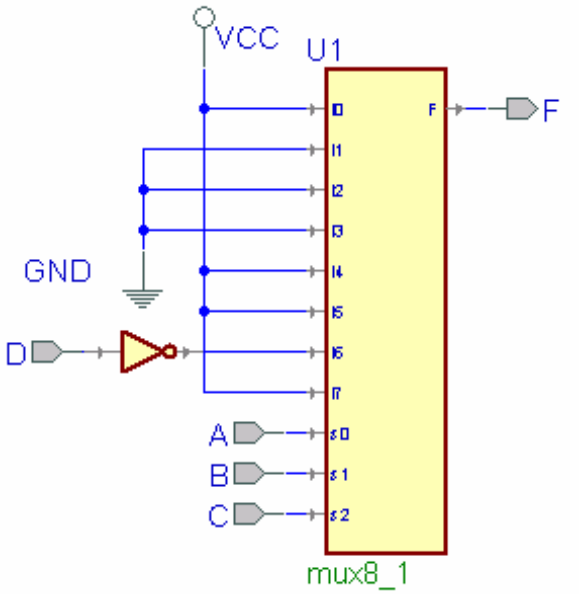
(b) f(A, B, C, D) = $\sum$ m(1, 8, 11, 12, 14, 15) + $\sum$ d(0, 2, 9, 10)

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

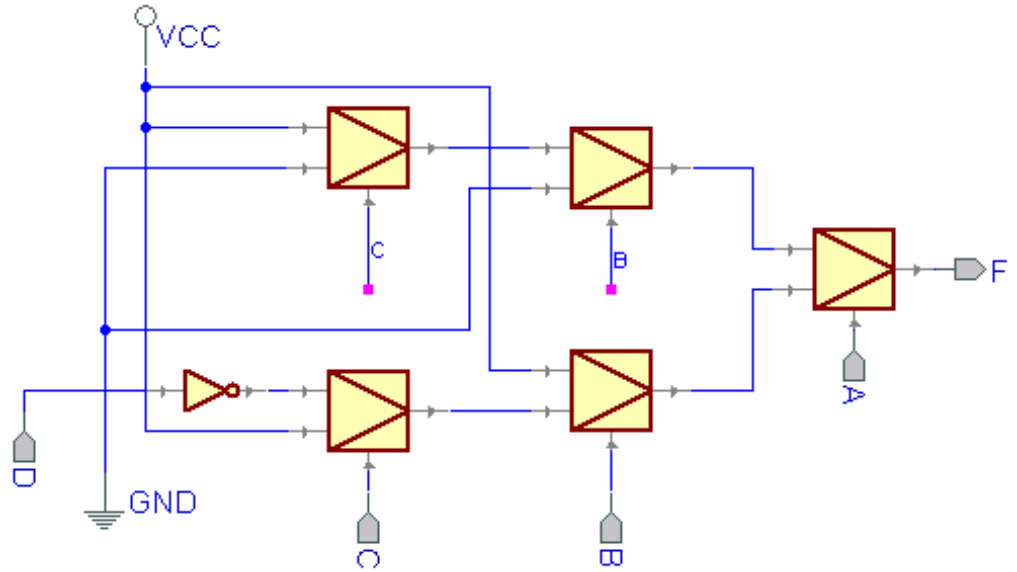
(d) The solution below assumes that each input and its complement is available. Utilizing DeMorgan's laws, the AND-OR-INVERT gate is the same as doing a product-of-sums solution, except each of the inputs into the AND gates must be inverted.



(e)

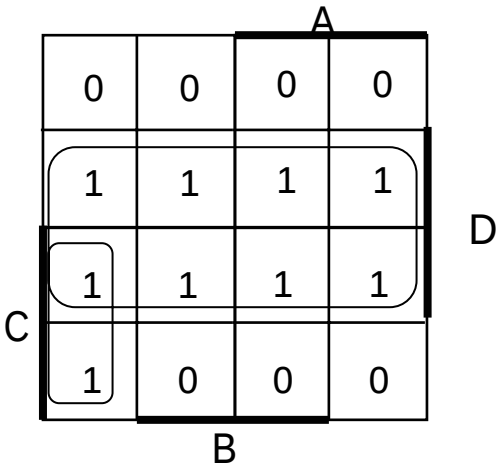


(f)

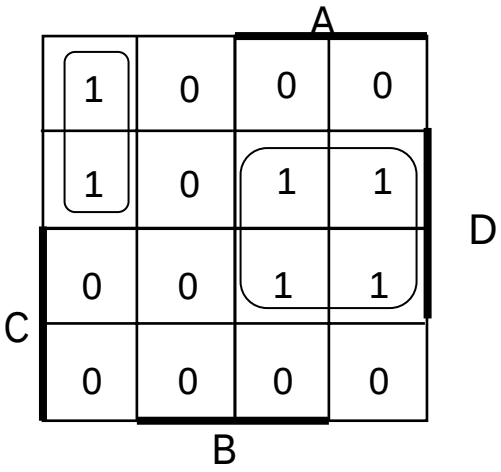


Exercise 4.21

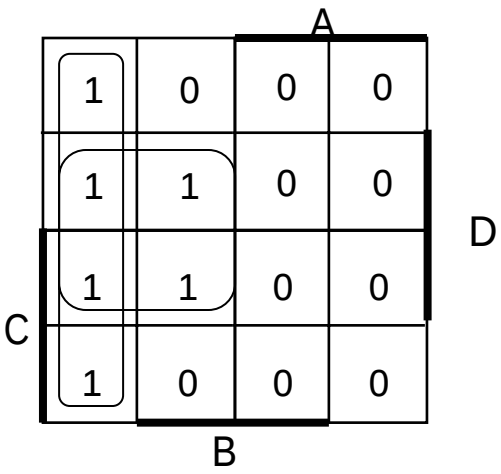
(a) The following K-maps show the minimum sum-of-products form for each of the 3 equations. This solution has 6 distinct product terms.



X = D + A'B'C



Y = AD + A'B'C'



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

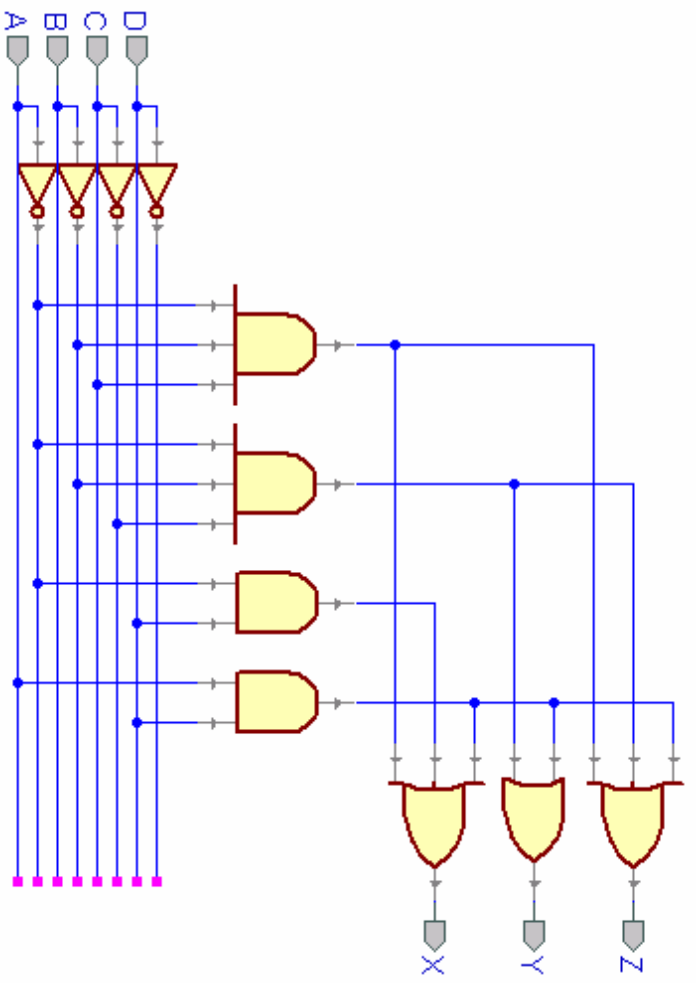
(b) The following solution only has 4 distinct product terms:

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

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

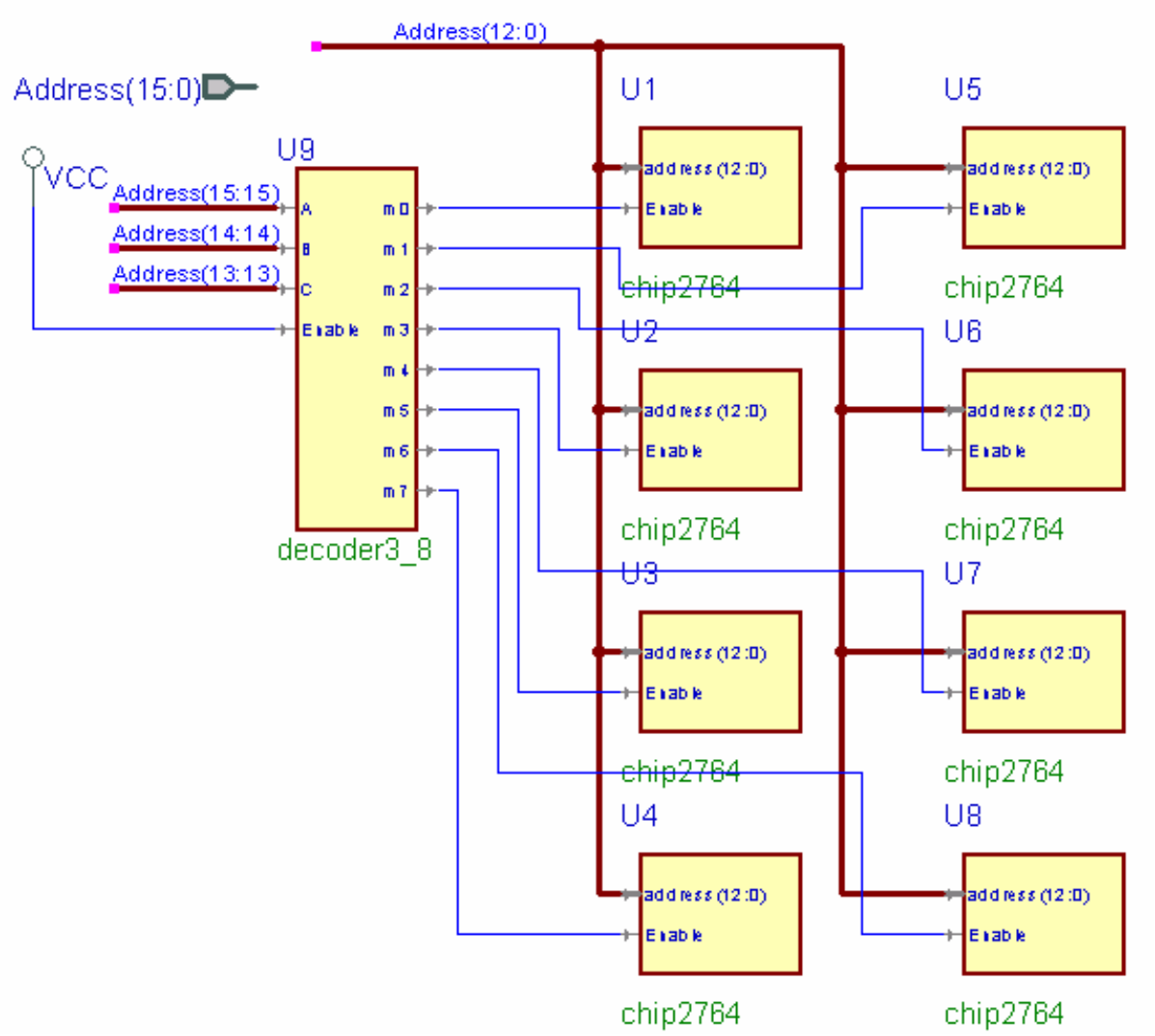
$$Z = A'D + A'B'C' + A'B'C$$

(c)

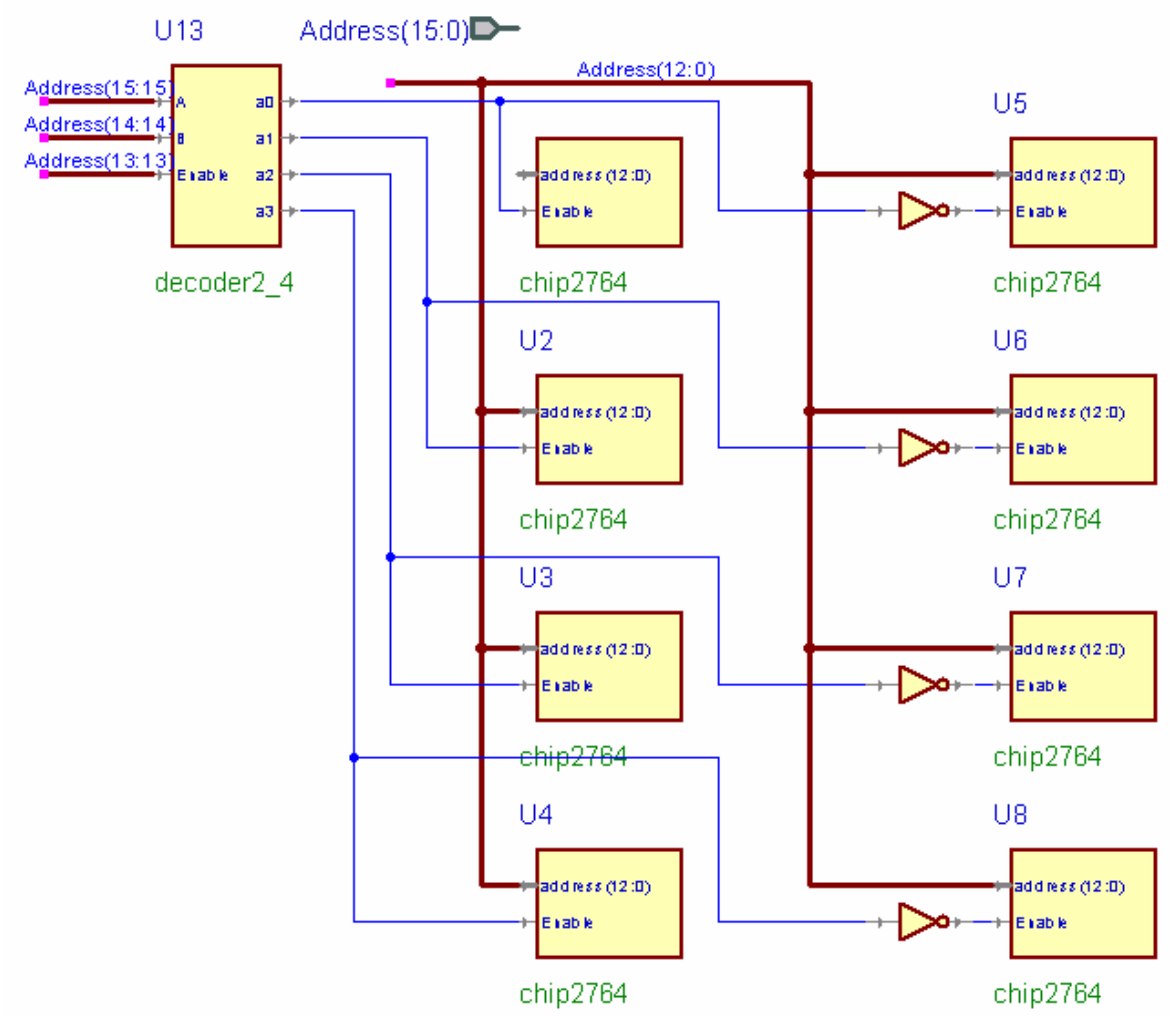


Exercise 4.22

(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.

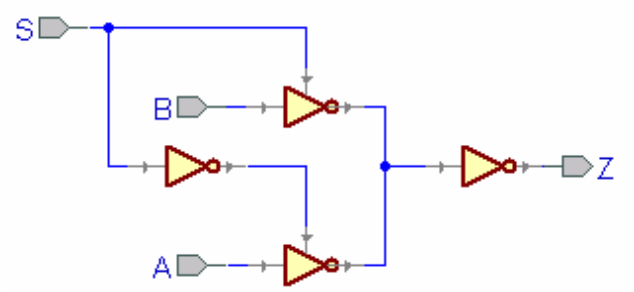


(b) The trick is to use the Enable signal on the decoder as another address bit. By inverting the enable on half of the 2764 chips, it will enable when the address bit is low and be disabled when that address bit is high.

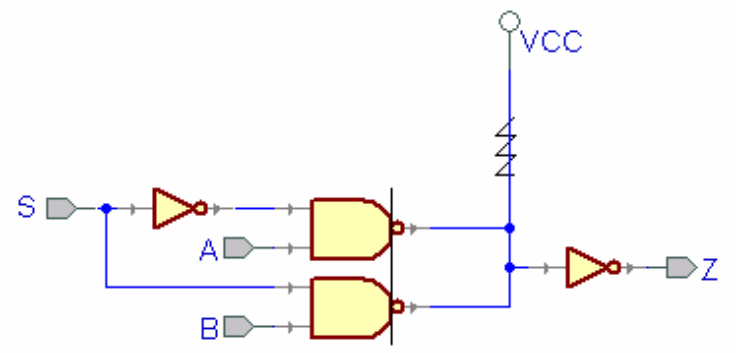


Exercise 4.23

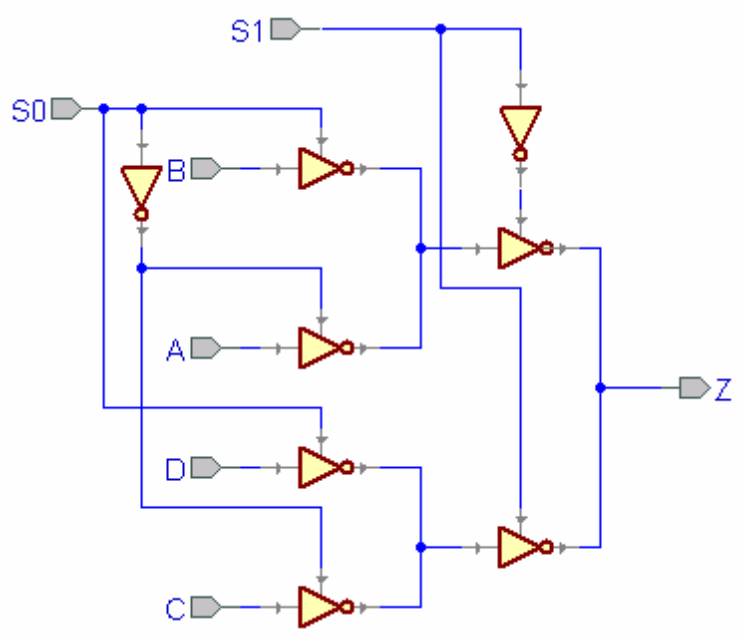
(a)

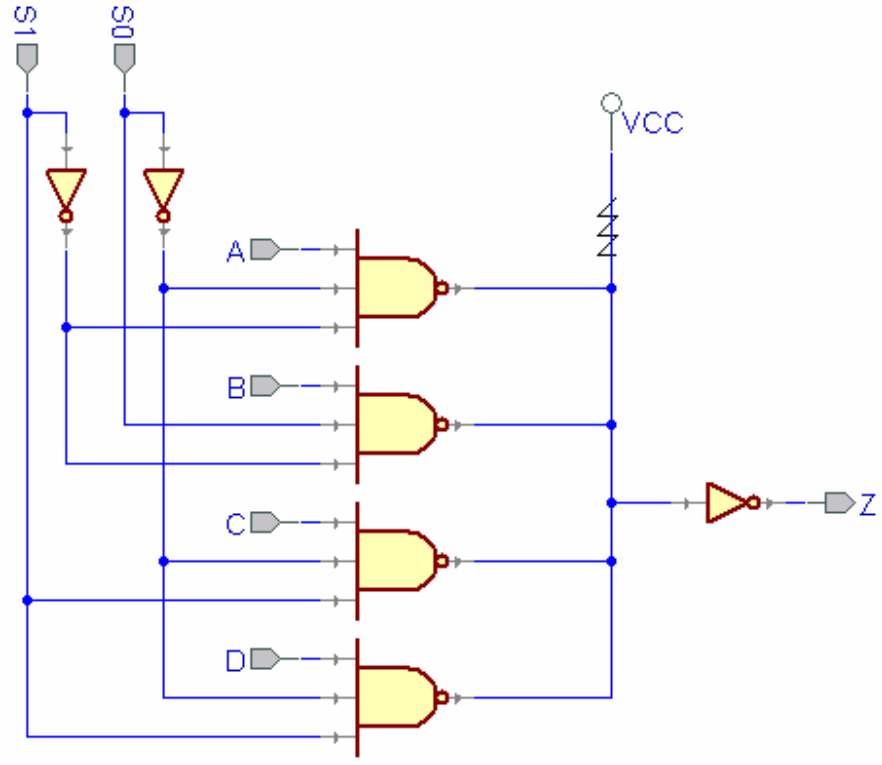


(b)



(c)





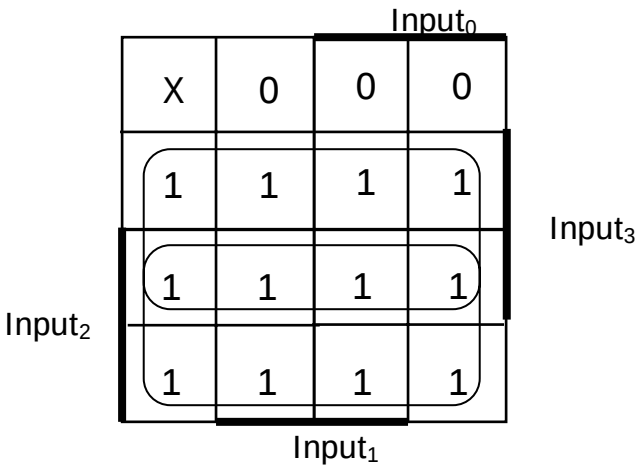
Exercise 4.24

(a)

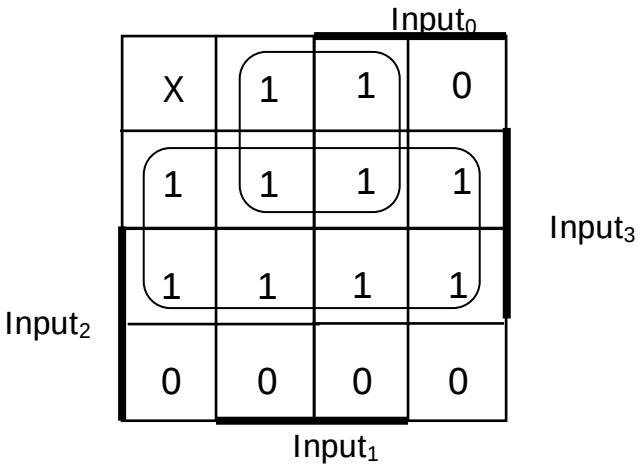
$Input_0$	$Input_1$	$Input_2$	$Input_3$	$Output_0$	$Output_1$
0	0	0	0	X	X
0	0	0	1	1	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	1	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	1	1	1
1	1	1	0	1	0
1	1	1	1	1	1

(b)

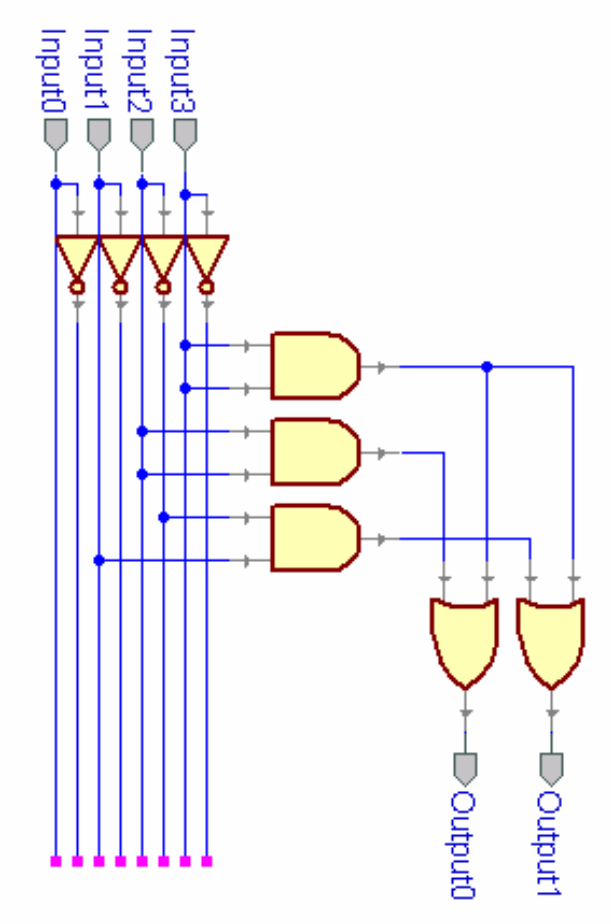
$Output_0 = Input_3 + Input_2$



$Output_1 = Input_1\ Input_2' + Input_3$



(c)



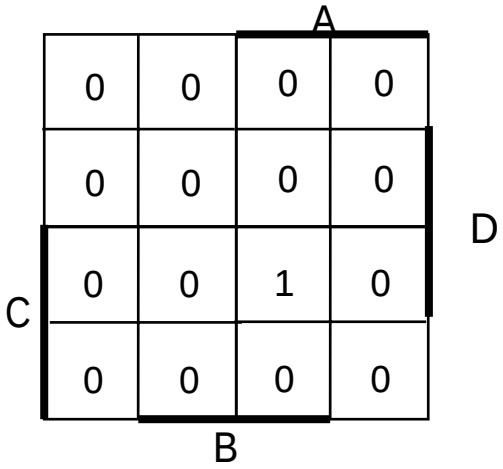
Exercise 4.25

(a)

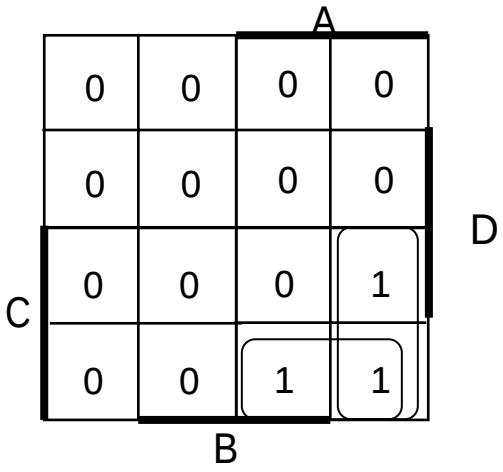
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

(b)

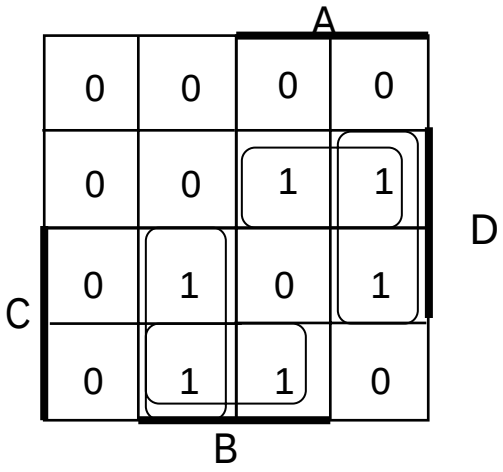
W = ABCD



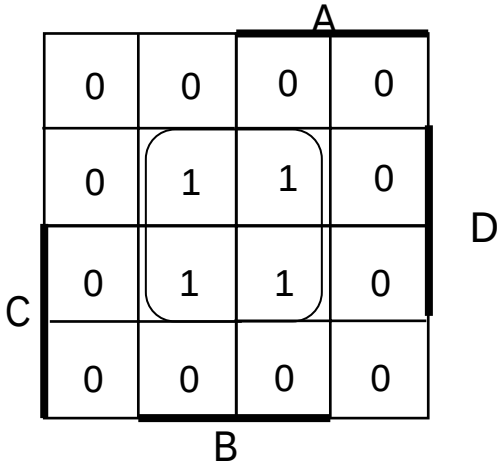
X = ACD’ + AB’C



Y = AB’D + AC’D + A’BC + BCD’



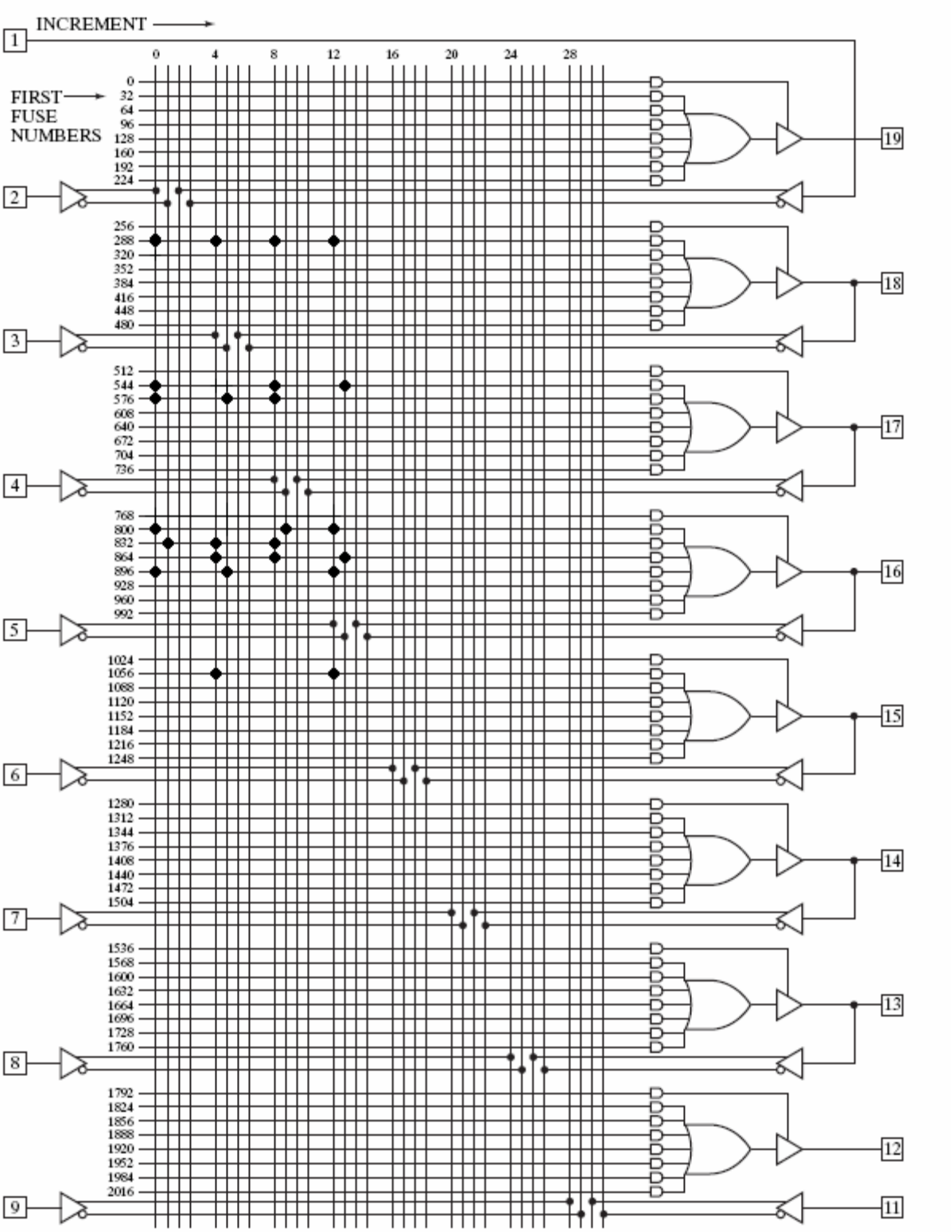
Z = BD



(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:

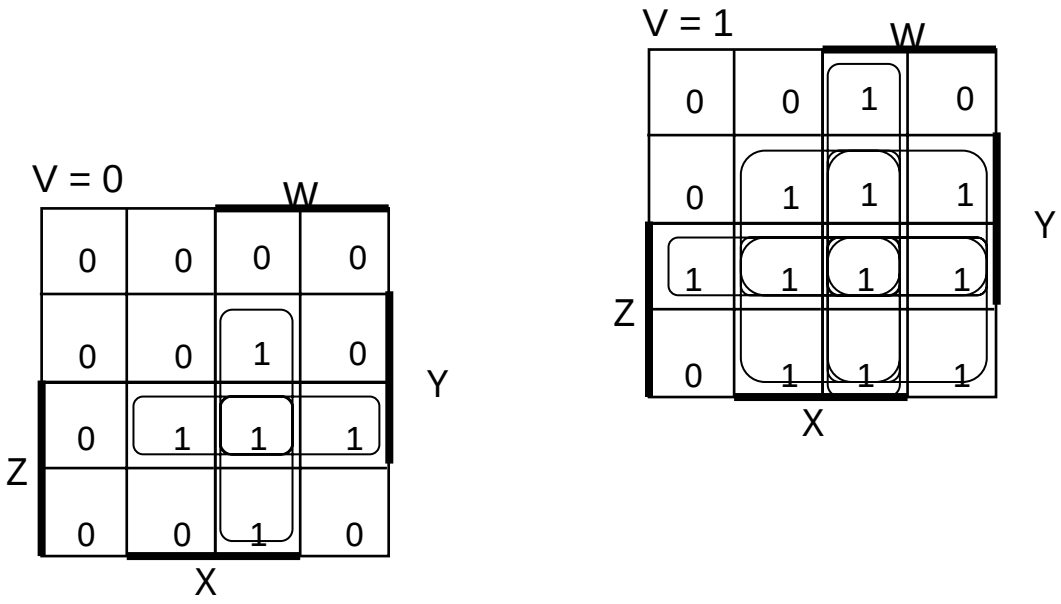
$$V = AB + BC + AC$$

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

(b)

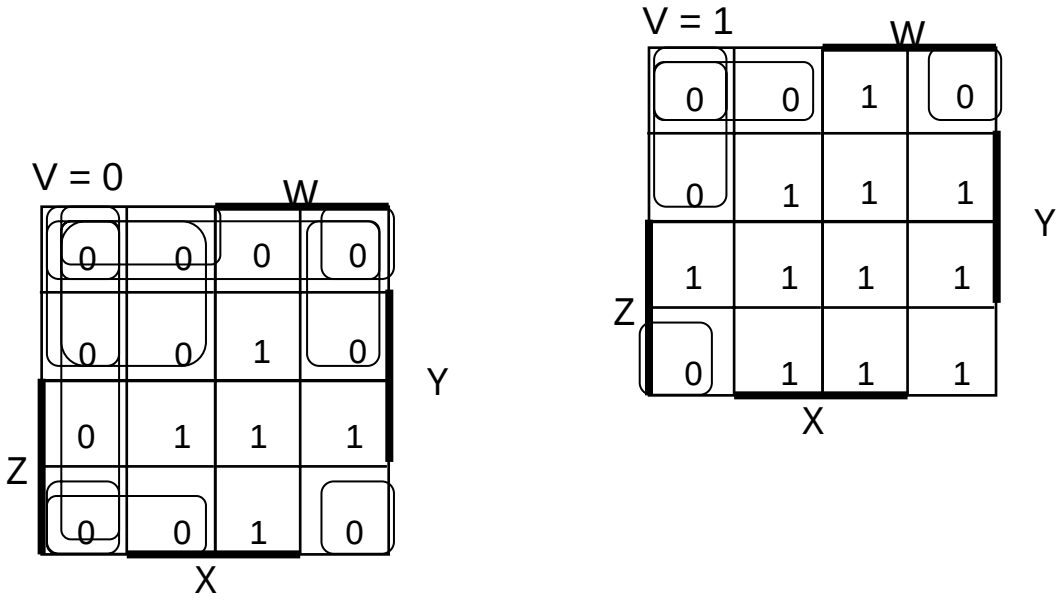
VWXYZ	Output
00000	0
00001	0
00010	0
00011	0
00100	0
00101	0
00110	0
00111	1
01000	0
01001	0
01010	0
01011	1
01100	0
01101	1
01110	1
01111	1
10000	0
10001	0
10010	0
10011	1
10100	0
10101	1
10110	1
10111	1
11000	0
11001	1
11010	1
11011	1
11100	1
11101	1
11110	1
11111	1

(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’)

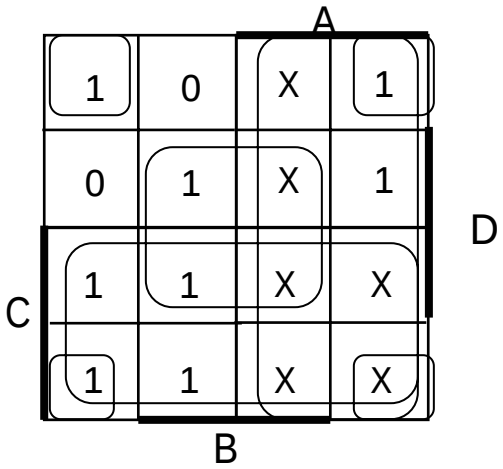
(e) This would require 3 CLB chips. Each circuit would use a separate CLB chip. In the case where only 4-input CLB's are available, each of the terms in the minimized sum-of-products form can be implemented with 3 or less terms. This means that dividing Circuit #3 into 4 CLB's with VWXY, VWXZ, VXYZ,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

Exercise 4.27

X = C’ + D’

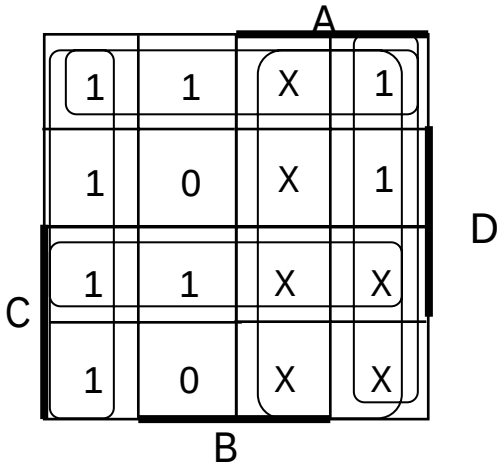
Y = B’C’

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’ √

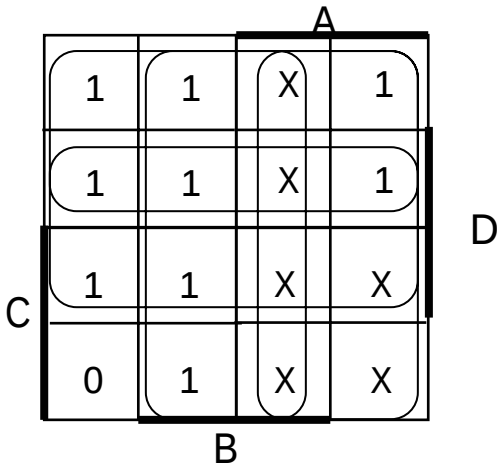
C_1 = Y + A’C_5’ + C’D’C_6
= B’C’ + A’ (C + D) (B’ + D) (B’ + C) + AC’D’ + BC’D’
= B’C’ + A’ (B’C + B’D + CD) + AC’D’ + BC’D’
= B’C’ + A’B’C + A’B’D + A’CD + AC’D’ + BC’D’



= A + C’D’ + CD + B’ √

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

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

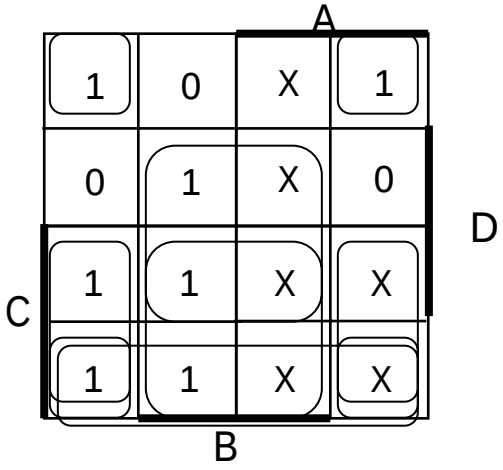


$= A + B + C' + D$ √

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

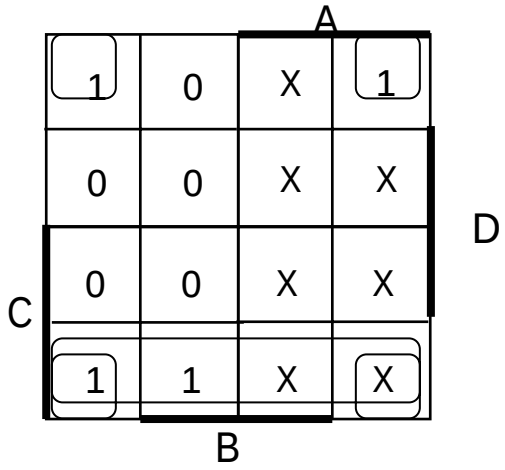
$= B'C'D' + A'CD' + BD\,(B'C'D' + AB'C' + A'BC' + A'BD') + A'B'\,(C' + D')'$

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



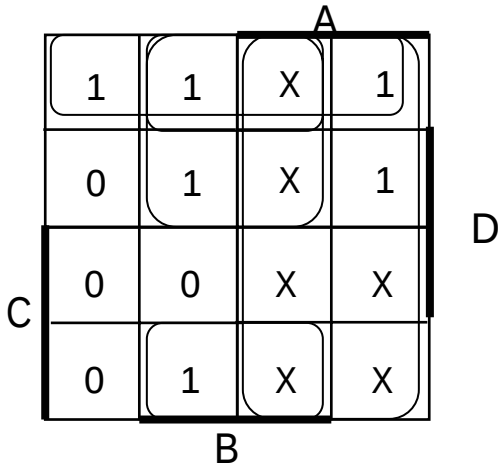
$= B'D' + CD' + BC'D + B'C$ √

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



$= B'D' + CD'$ √

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



$= A + C'D' + BD' + BC'$ √

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

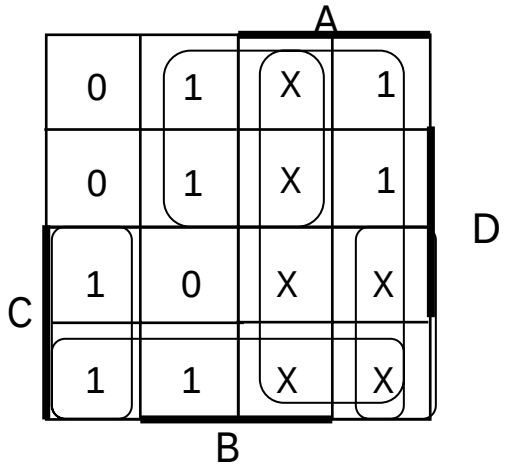
$= AB'C'D' + C (B'C'D' + AB'C' + A'BC' + A'BD') + (B'C'D' + A'CD')'$
$(B'C'D' + AB'C' + A'BC' + A'BD') + A'B'C$

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

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

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

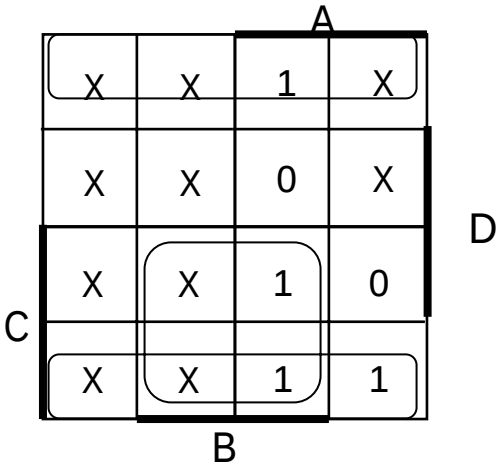
$= AB'C'D' + A'BCD' + AB'C'D + A'BC'D + A'BC' + A'B'C$



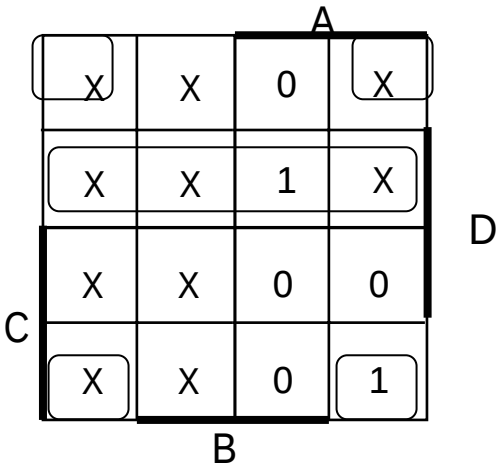
$= A + CD' + BC' + B'C$ √

Exercise 4.28

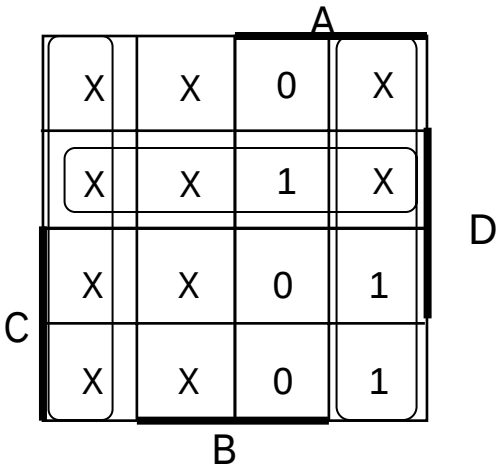
(a) The following K-maps determine the minimized sum-of-products forms for each of the 7 terms for the LED display decoder.



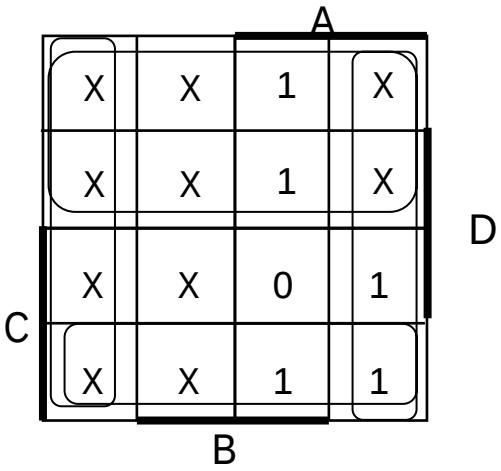
$C_0 = BC + D'$



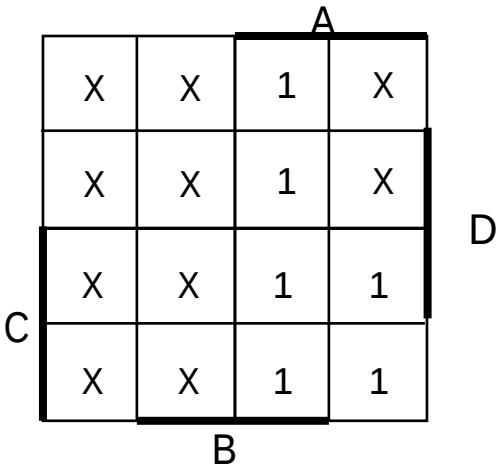
$C_1 = C'D + B'D'$



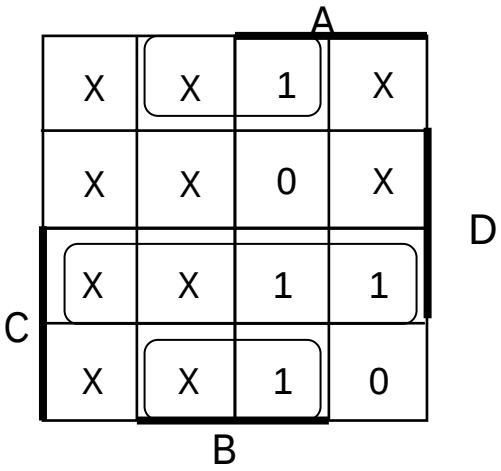
$C_2 = B' + C'D$



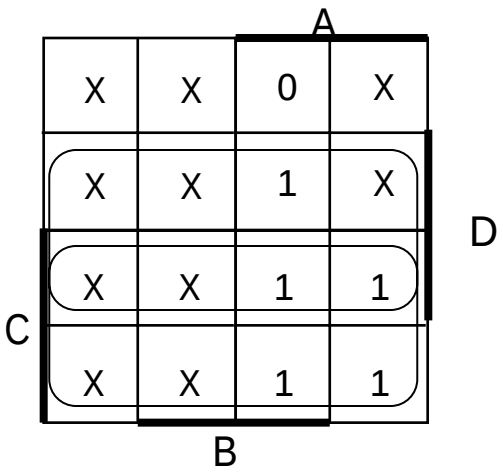
$C_3 = C' + B' + D'$



$C_4 = 1$

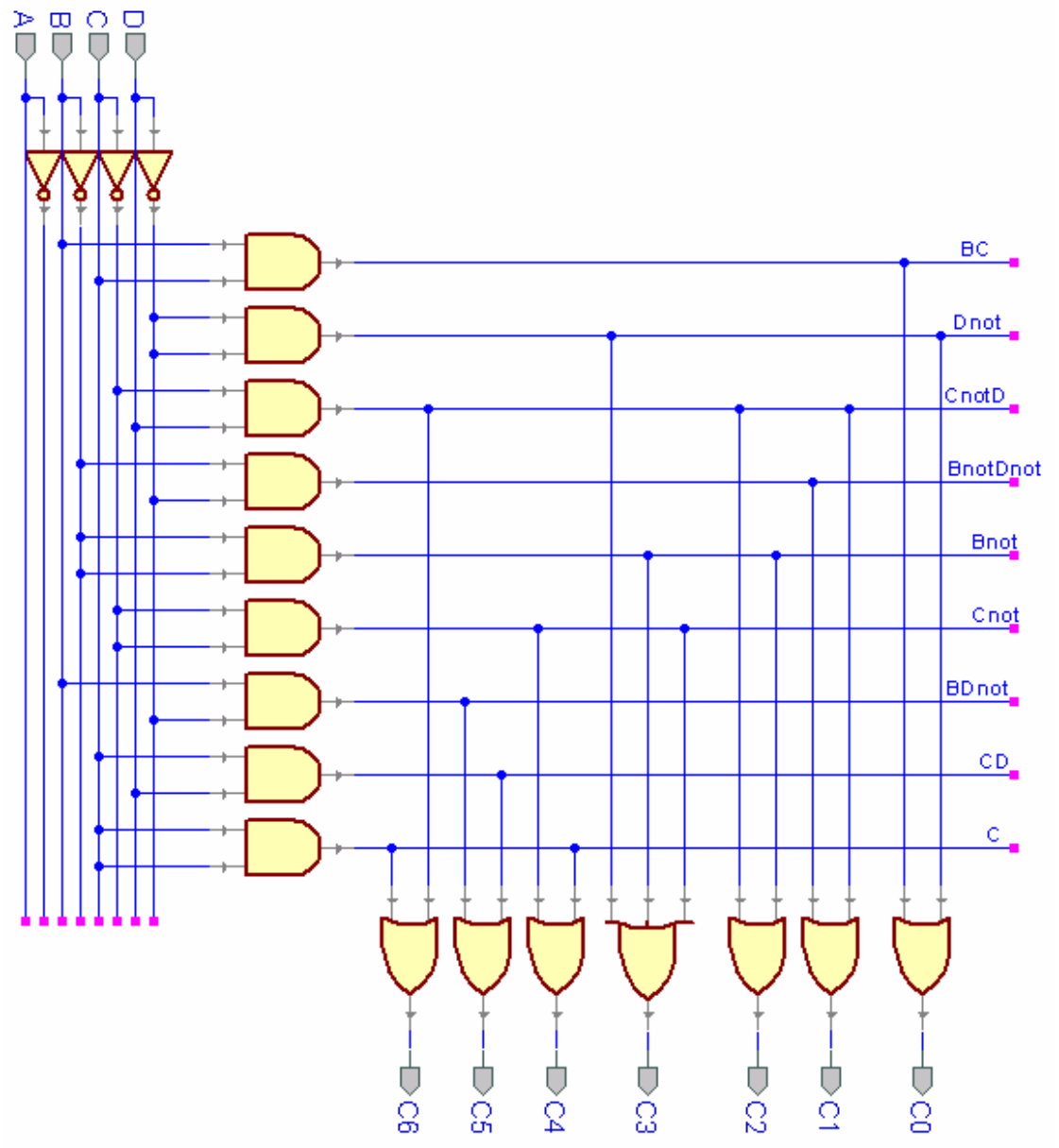


$C_5 = BD' + CD$

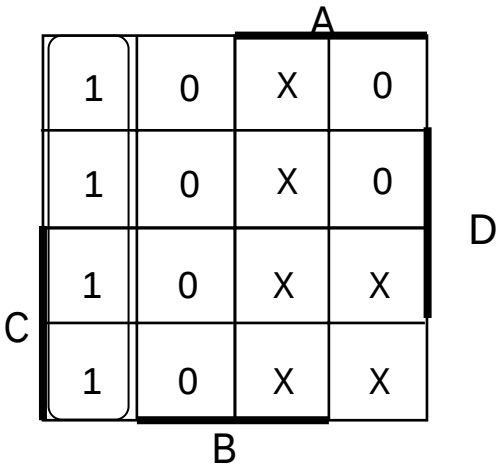


$C_6 = C + D$

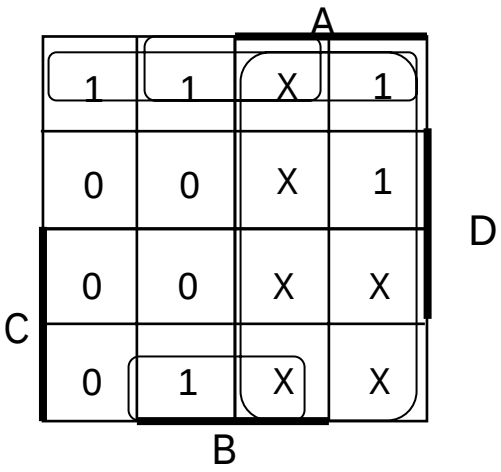
(b) Notice in the implementation some of the expressions in the minimized sum-of-products form are implemented using other available terms. For example, C_4 uses C' and C to implement the "true" function.



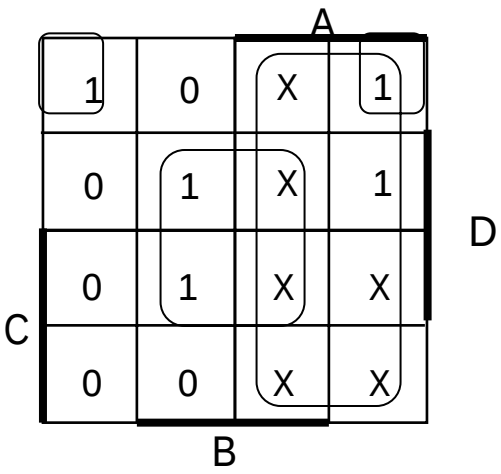
Exercise 4.29



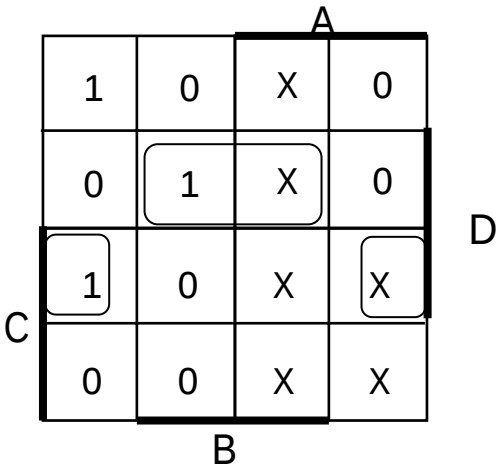
$C_0 = A'B'$



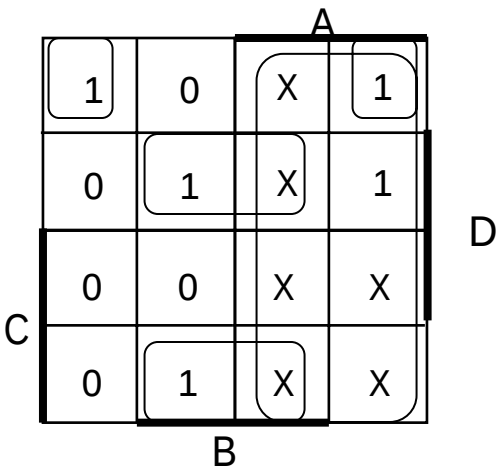
$C_1 = C'D' + A + BD'$



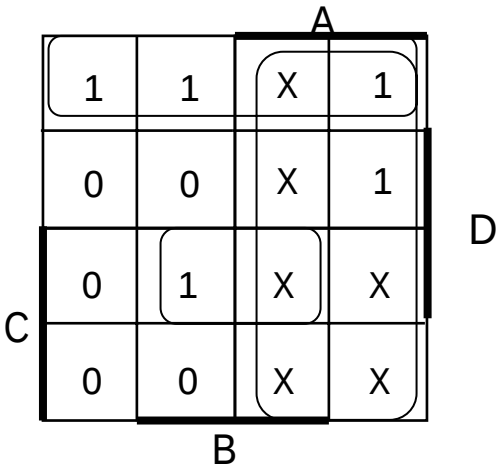
$C_2 = B'C'D' + A + BD$



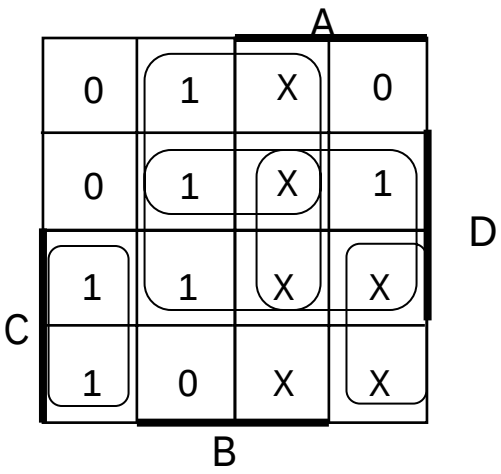
$C_3 = A'B'C'D' + BC'D + B'CD$



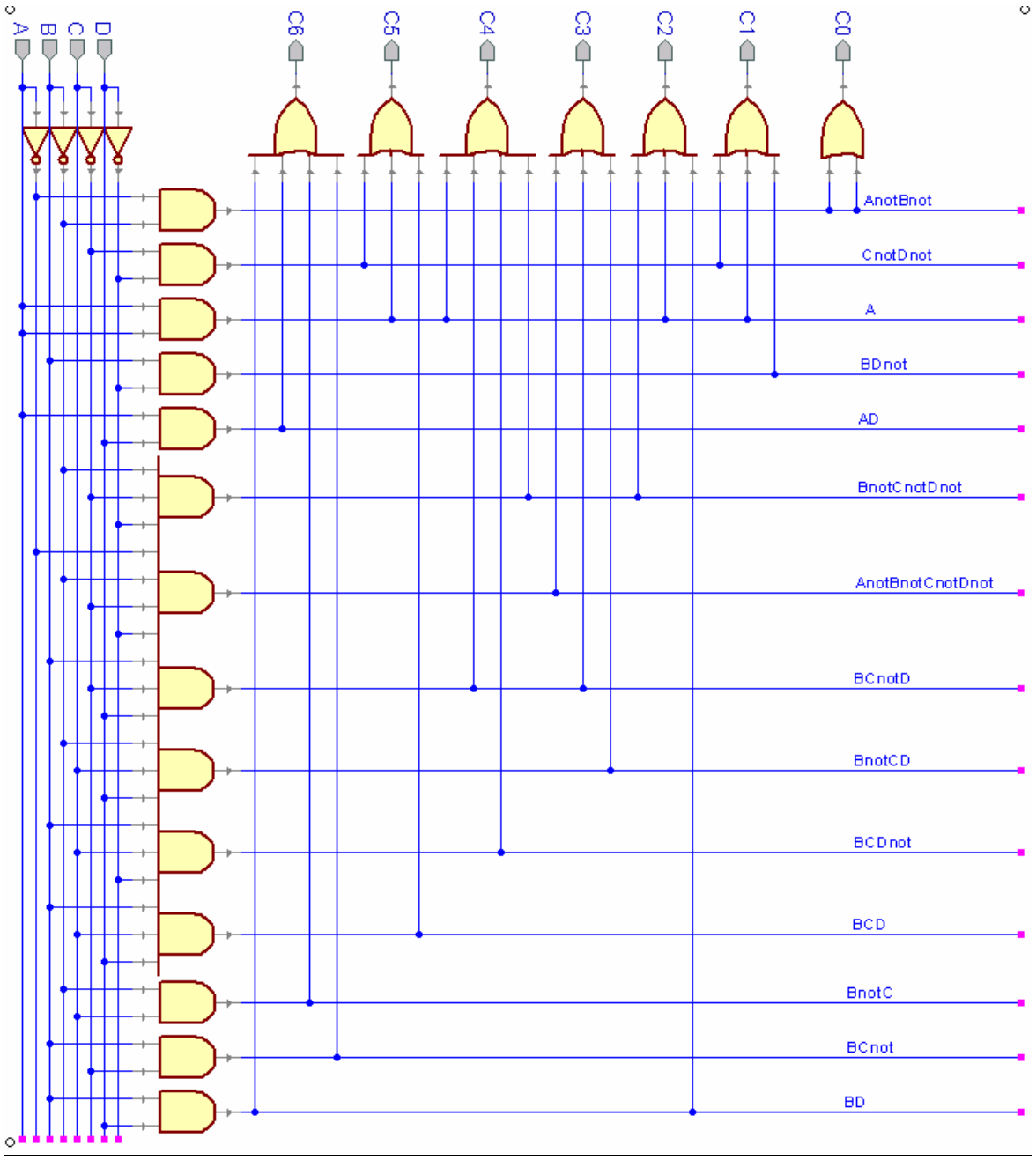
$C_4 = A + BC'D + BCD' + B'C'D'$



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

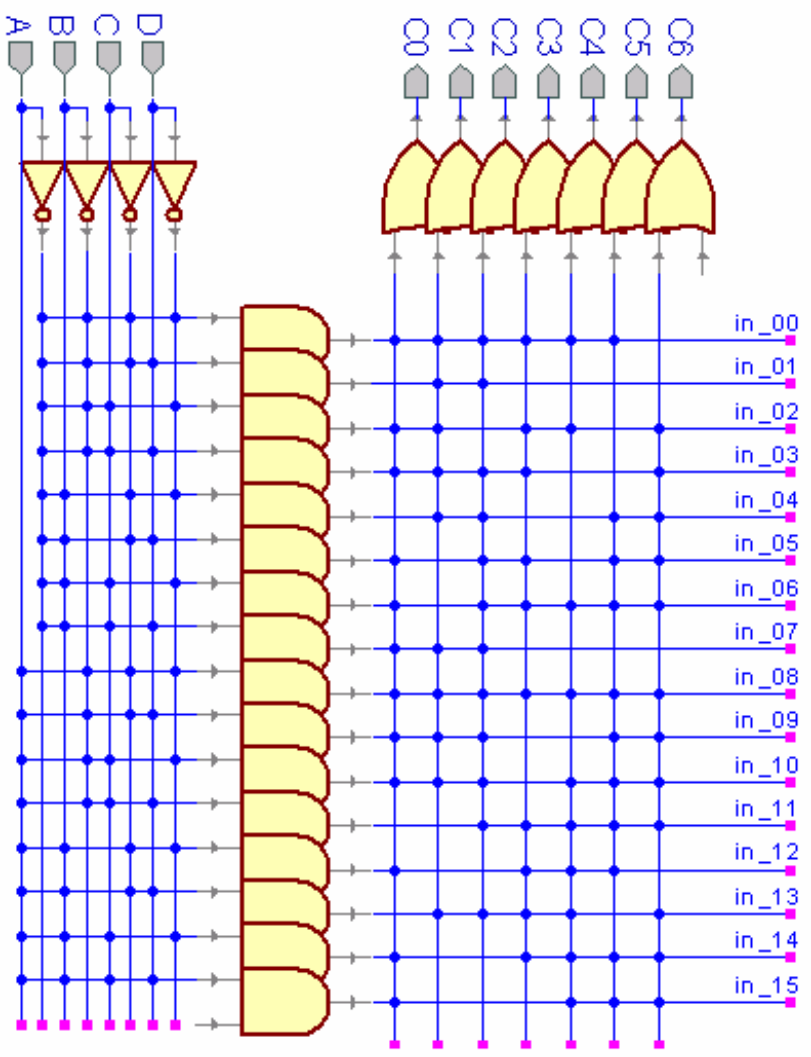


$C_6 = BD + AD + B'C + BC'$


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

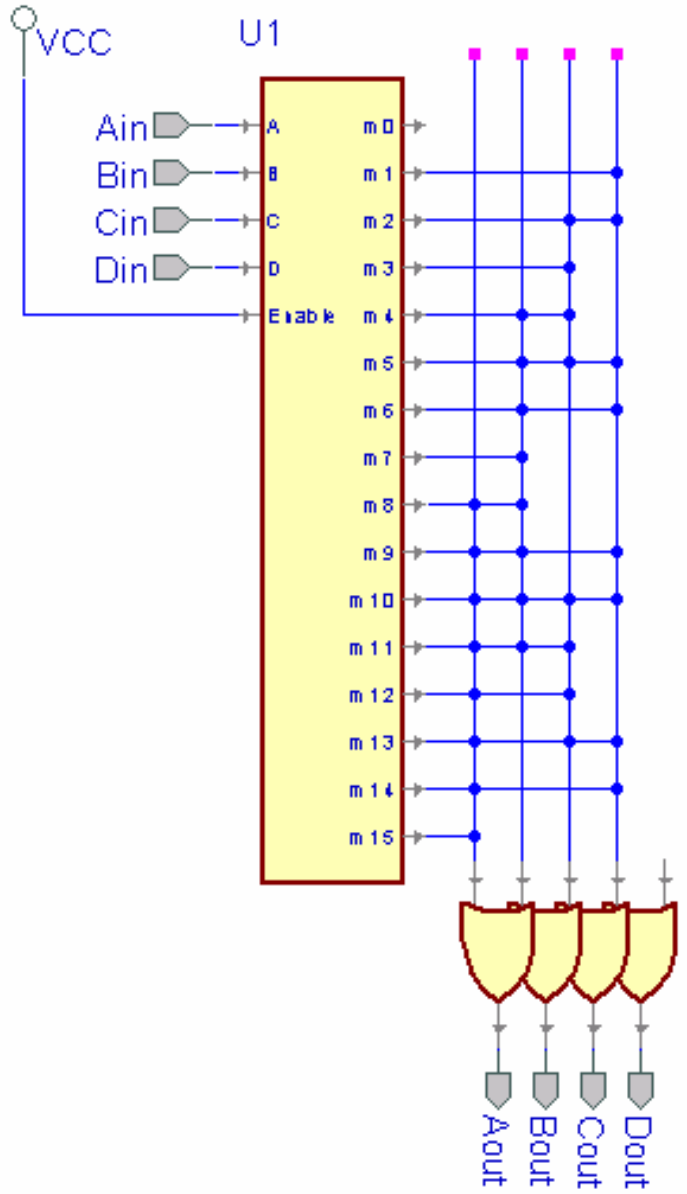
It is really tempting to use K-maps to try and simplify each of the output functions. However, using the K-map method eventually leaves it so each function is harder to implement using existing terms, and usually gives a lot more than 16 AND terms. Instead of using the K-map, using the brute force 0000 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.



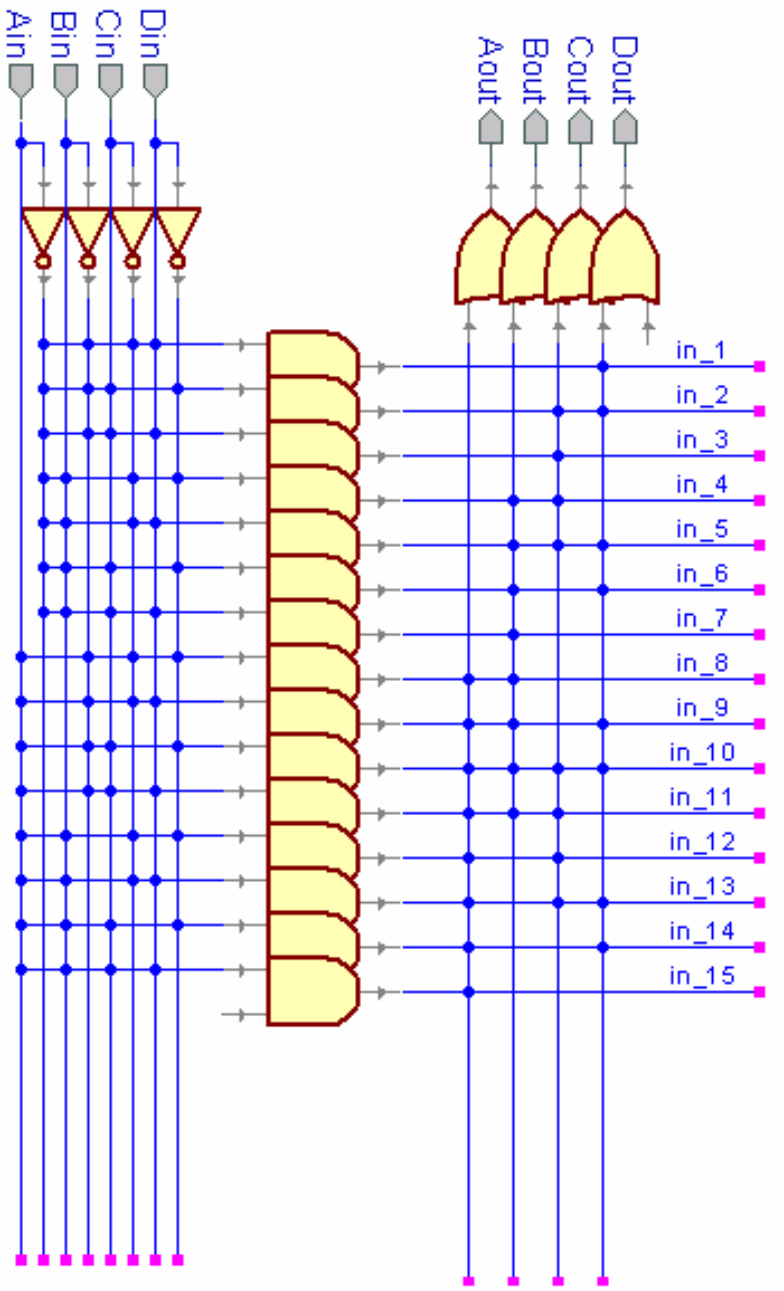
Exercise 4.31

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



Exercise 5.1

(a) The input signals to this system are: # dimes and # of nickels in the reservoir, N_D and N_N, respectively, and coin deposited, C. The outputs are: no change, E, # dimes and # of nickels to return, R_D and R_N, respectively.

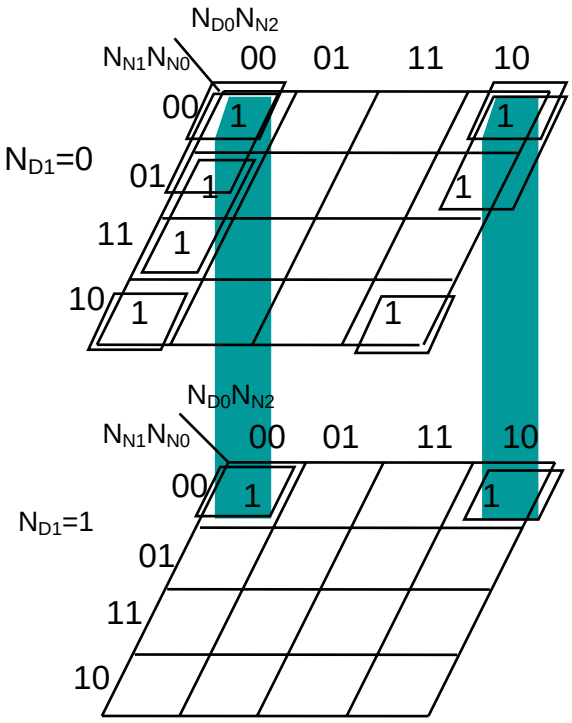
(b) N_D and N_N are the number of dimes and nickels left in the machine. C tells us that a quarter has been deposited so we need to make change. E signals to the machine that it should light up a "No Change Available" sign. R_D and R_N tell the machine how many dimes and nickels to return.

(c) Minimized gate-level implementation:

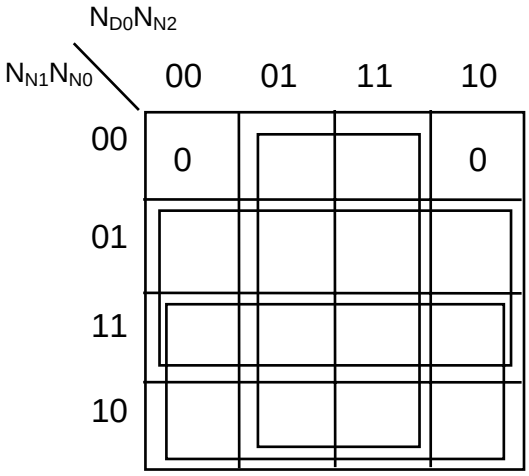
N_{D1}	N_{D0}	N_{N2}	N_{N1}	N_{N0}	E	R_{D1}	R_{D0}	R_{N2}	R_{N1}	R_{N0}
0	0	0	0	0	1	0	0	0	0	0
		0	0	1	1	0	0	0	0	0
		0	1	0	1	0	0	0	0	0
		0	1	1	1	0	0	0	0	0
0	0	1	0	0	1	0	0	0	0	0
		1	0	1	0	0	0	1	0	1
		1	1	0	0	0	0	1	0	1
		1	1	1	0	0	0	1	0	1
0	1	0	0	0	1	0	0	0	0	0
		0	0	1	1	0	0	0	0	0
		0	1	0	1	0	0	0	0	0
		0	1	1	0	0	1	0	1	1
0	1	1	0	0	0	0	1	0	1	1
		1	0	1	0	0	1	0	1	1
		1	1	0	0	0	1	0	1	1
		1	1	1	0	0	1	0	1	1
1	0	0	0	0	1	0	0	0	0	1
		0	0	1	0	1	0	0	0	1
		0	1	0	0	1	0	0	0	1
		0	1	1	0	1	0	0	0	1
1	0	1	0	0	0	1	0	0	0	1
		1	0	1	0	1	0	0	0	1
		1	1	0	0	1	0	0	0	1
		1	1	1	0	1	0	0	0	1
1	1	0	0	0	1	1	0	0	0	1
		0	0	1	0	1	0	0	0	1
		0	1	0	0	1	0	0	0	1
		0	1	1	0	1	0	0	0	1
1	1	1	0	0	0	1	0	0	0	1
		1	0	1	0	1	0	0	0	1
		1	1	0	0	1	0	0	0	1
		1	1	1	0	1	0	0	0	1

The input C is not included in the truth table because we never return change unless C is 1. However, we do still need to compute E.

If we look at the truth table carefully, we can see that not all of the outputs will require 5-variable K-maps. R_{D1} and R_{D0} can be implemented with 4-variable K-maps, which means that we will have to AND N_{D1} to the output those functions. Similarly, R_{N2} only requires a 3-variable K-map. Also, notice that R_{D0} and R_{N1} are the same.



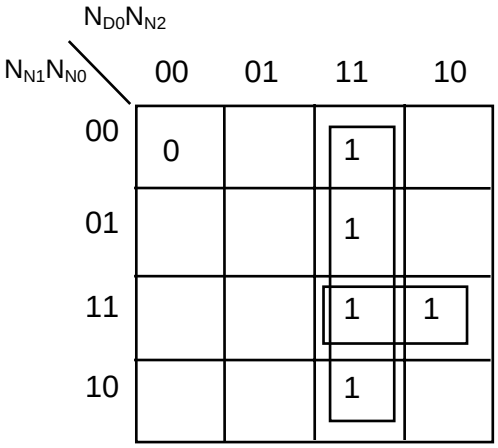
K-map for E



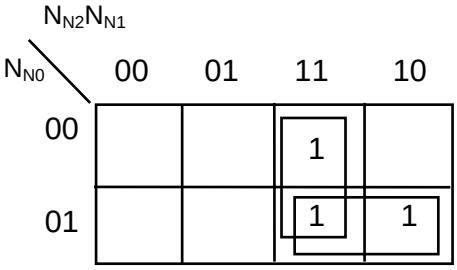
K-map for R_{D1}

$E = N_{D1}'N_{N2}'N_{N0}' + N_{N2}'N_{N1}'N_{N0}' + N_{D1}'N_{N2}'N_{N1}' + N_{D1}'N_{D0}'N_{N2}' + N_{D1}'N_{D0}'N_{N1}'N_{N0}'$

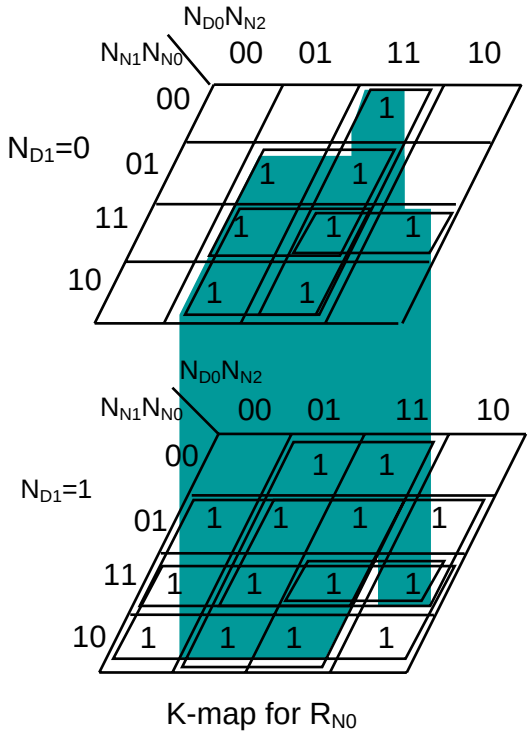
$R_{D1} = N_{D1}(N_{N2} + N_{N1} + N_{N0})$



K-map for R_{D0} & R_{N1}



K-map for R_{N2}



K-map for R_{N0}

$R_{D0} = R_{N1} = (N_{D0}N_{N2} + N_{D0}N_{N1}N_{N0})N_{D1}'$

$R_{N2} = (N_{D0}N_{N2} + N_{D0}N_{N1})N_{D1}'N_{D0}'$

$R_{N0} = N_{D0}N_{N2} + N_{N2}N_{N0} + N_{N2}N_{N1} + N_{D0}N_{N1}N_{N0} + (N_{N2} + N_{N1} + N_{N0})N_{D1}$

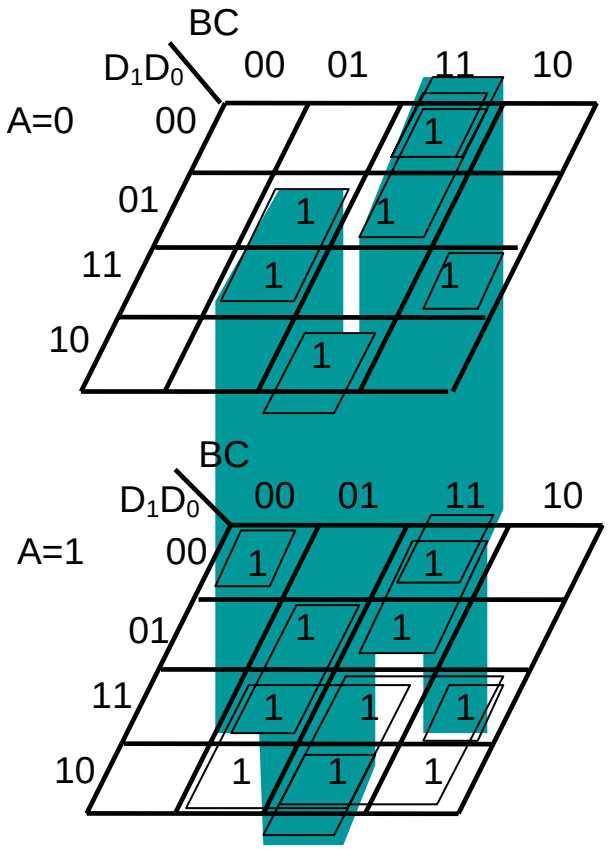
Exercise 5.2

This logic unit is a simple ALU which uses the control bits, A, B, and C, to determine the operation to be performed on D_1 and D_0.

(a) Truth table for Z:

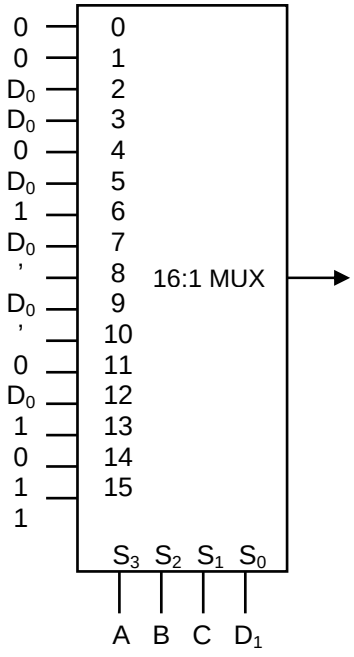
A	B	C	D_1	D_0	Z
0	0	0	0	0	0
			0	1	0
			1	0	0
			1	1	0
0	0	1	0	0	0
			0	1	1
			1	0	0
			1	1	1
0	1	0	0	0	0
			0	1	0
			1	0	0
			1	1	1
0	1	1	0	0	1
			0	1	1
			1	0	1
			1	1	0
1	0	0	0	0	1
			0	1	0
			1	0	0
			1	1	0
1	0	1	0	0	0
			0	1	1
			1	0	1
			1	1	1
1	1	0	0	0	0
			0	1	0
			1	0	1
			1	1	1
1	1	1	0	0	1
			0	1	1
			1	0	1
			1	1	1

K-map for Z:

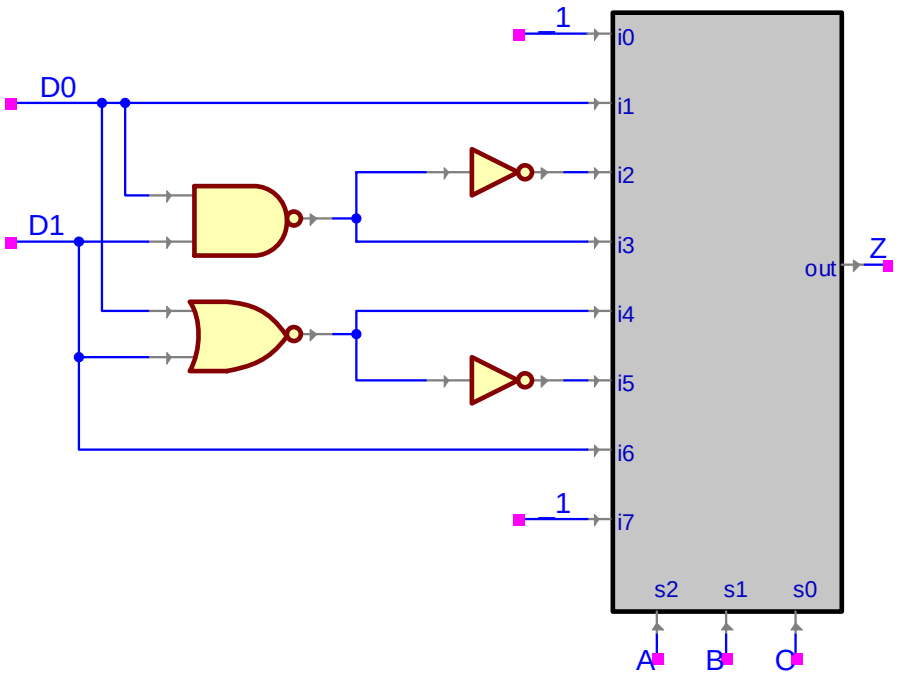


$Z(A,B,C,D_1,D_0) = BCD_1' + B'CD_0 + BCD_0' + BC'D_1D_0 + ABD_1 + ACD_1 + AB'C'D_1'D_0'$

(b) 16:1 Mux for Z:



(c) 8:1 MUX for Z:

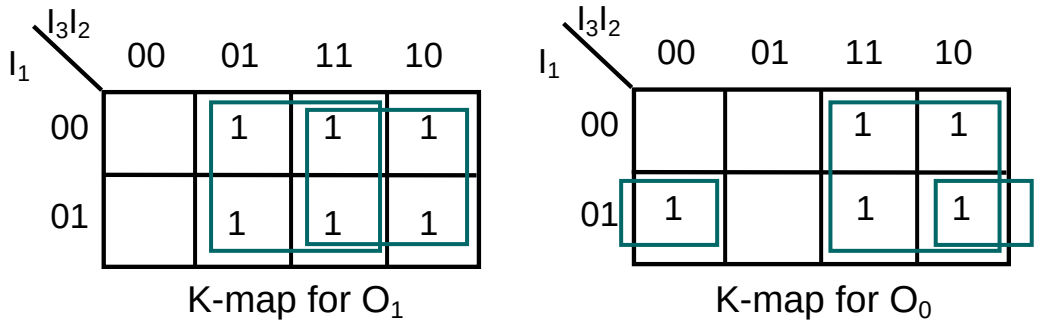


Exercise 5.3

(a) Truth table for O_1 and O_0:

I_3	I_2	I_1	O_1	O_0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

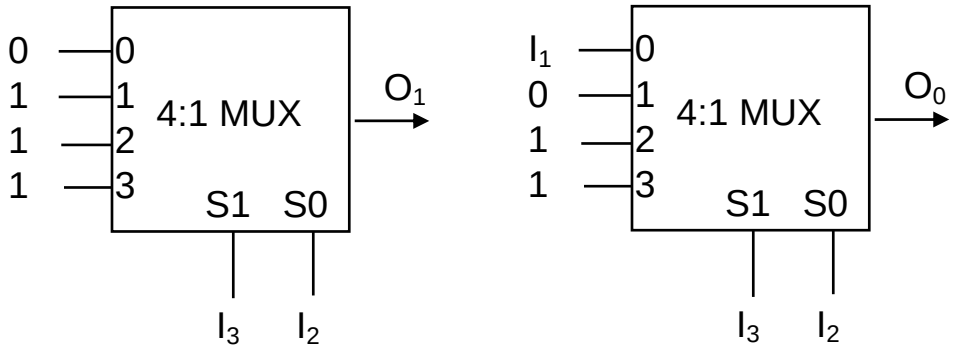
(b) Gate-level implementation:



$O_1 = I_3 + I_2$

$O_0 = I_3 + I_2'I_1$

(c) 4:1 multiplexor implementation:



Exercise 5.4

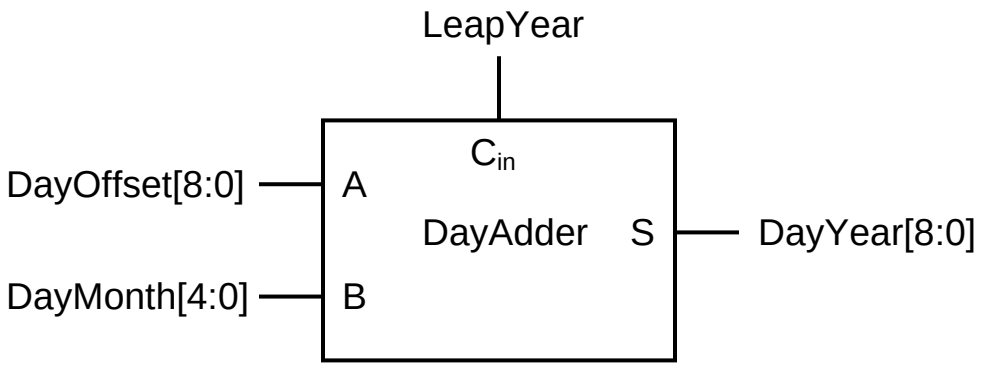
(a) The day offset block takes the month as input and returns a 9-bit value which corresponds to the number of days in the year up to that month.

Block Diagram for DayOffset

Verilog module:

```
module DayOffset(input wire [3:0] M, output reg [8:0] D);

   always@(*)
         begin
               case(M)
                     4'b0001: D = 0;
                     4'b0010: D = 31;
                     4'b0011: D = 59;
                     4'b0100: D = 90;
                     4'b0101: D = 120;
                     4'b0110: D = 151;
                     4'b0111: D = 181;
                     4'b1000: D = 212;
                     4'b1001: D = 243;
                     4'b1010: D = 273;
                     4'b1011: D = 304;
                     4'b1100: D = 334;
                     default: D = 0;
               endcase
         end
endmodule
```

(b) The adder takes the output from the DayOffset module, the day of the month, and a carry-in, which is tied to a leap year signal. Tying the leap year to the carry in allows us to add an extra day if it a leap year.



Block diagram for DayAdder

(c) This implementation of a day-of-the-year system would require 9 4-input LUTs to implement.

Exercise 5.5

(a) Truth table for the half-subtractor D = A - B:

A	B	B_I	D	B_L
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

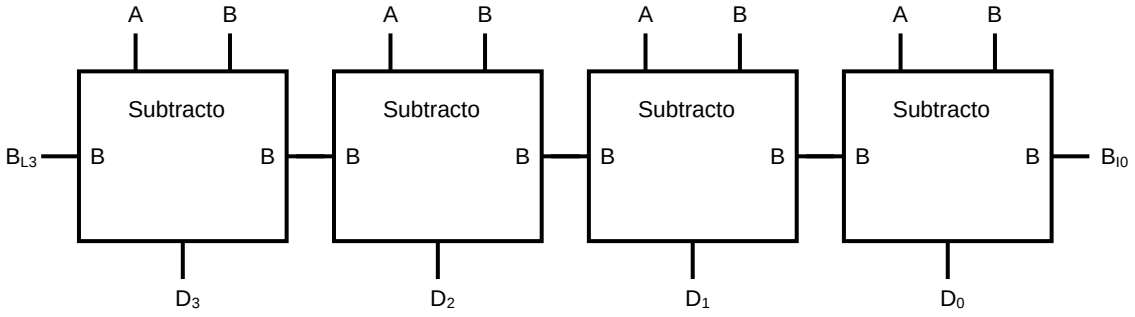
$D = A'B'B_I + A'BB_I' + AB'B_I' + ABB_I$

Or simply:

D = A xor B xor B_I

$B_L = A'B + A'B_I + BB_I$

(b) Block diagram for a 4-bit subtractor:



(c) This multi-bit subtractor works with 2's compliment numbers just as the full-adder does.

(d) Underflow can be detected by checking if the MSB tries to borrow from the left: B_{L3} is 1.

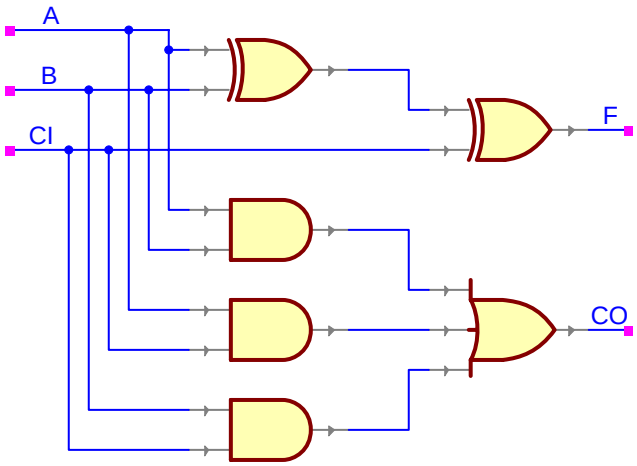
Exercise 5.6

The basic module we will use is the half-adder. As long as the operands are 2's compliment numbers, we can easily use the adder to subtract two numbers. We do this by inverting the bits of one operand and setting the carry-in of the first half-adder to one.

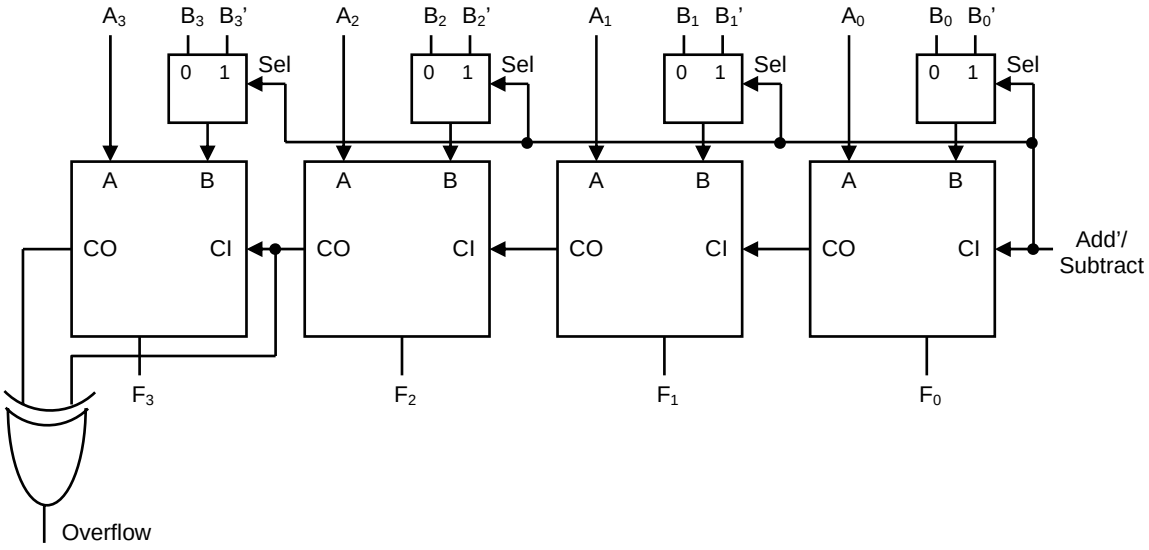
The carry-in, CI, and borrow-in, BI, can be shared, as well as the carry-out, CO, and borrow from left, BL, signals. Here are the functions for each block:

F = (A xor B xor CI)

CO = (AB + ACI + BCI)



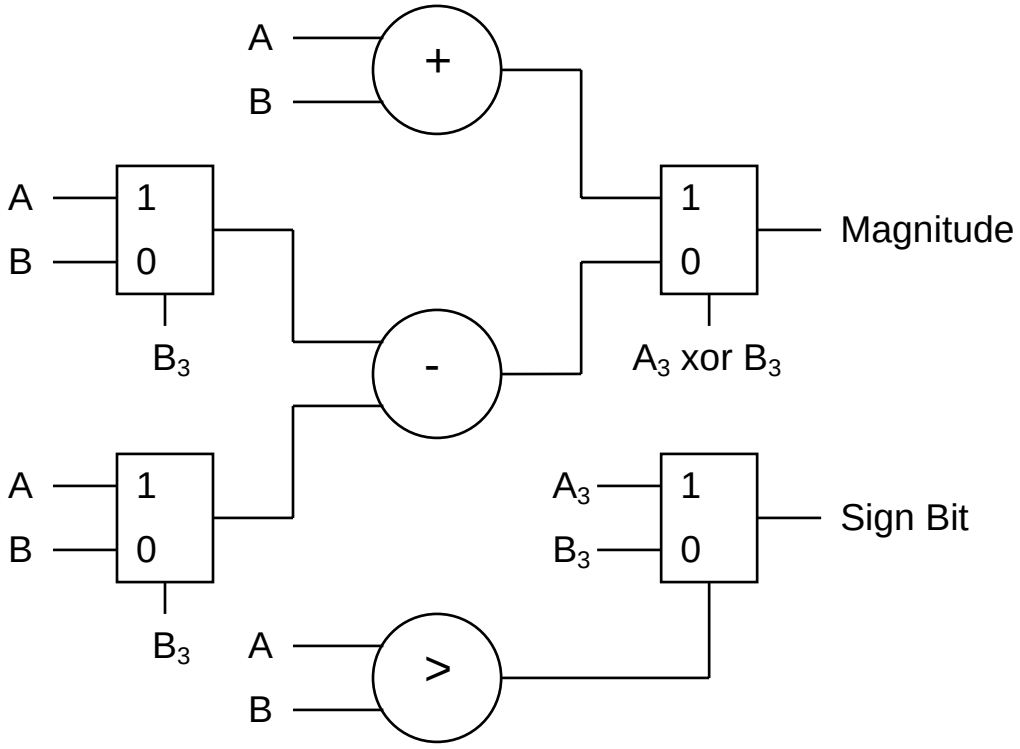
Circuit diagram for each block



Combinational adder/subtractor block diagram

Exercise 5.7

A and B are the 3-bit magnitudes of the operands. The sign bits are compared and if equal, the result is the sum of the magnitudes with the sign of the operands. If B is negative, it is subtracted from A. If B is positive, A must be negative (since we are assuming that they are unequal), so A is subtracted from B. The sign bit is chosen from the result of the comparison, but the selection is relevant only if the signs differ. We use once mux for both cases to save logic. Overflow is passed through the magnitude mux from the adder and subtractor blocks.



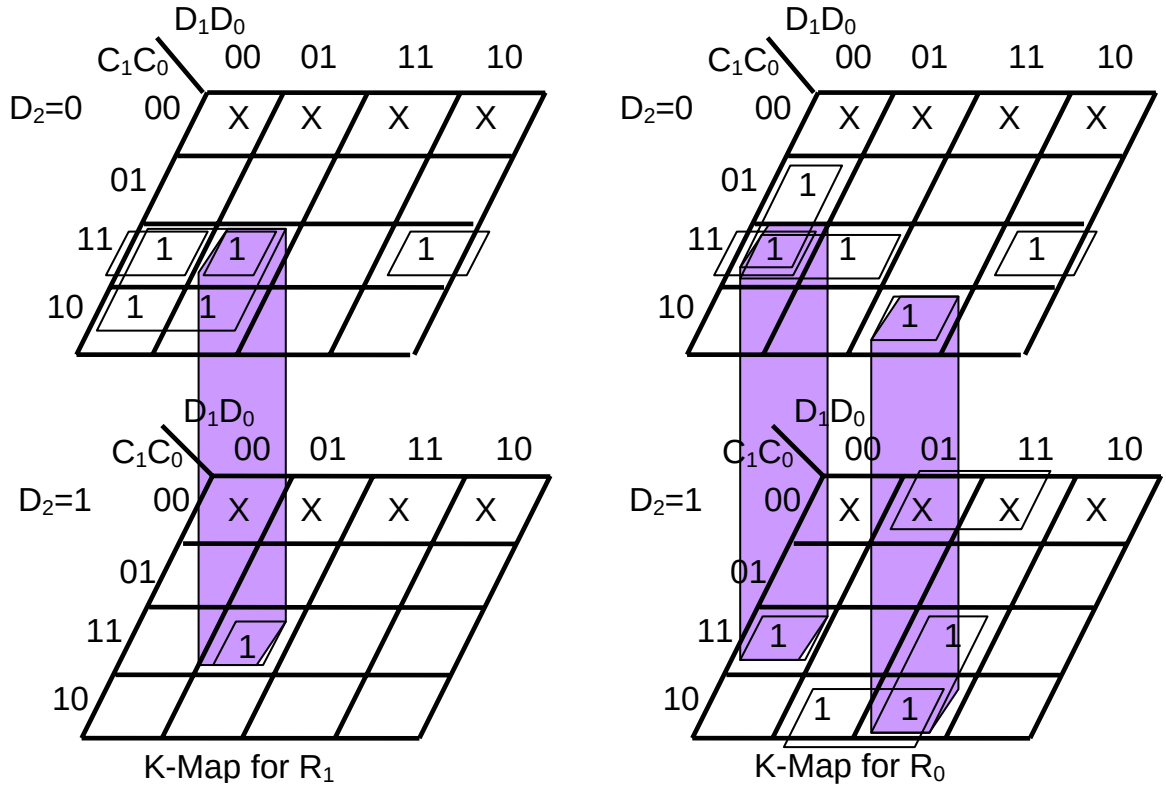
Exercise 5.8

(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

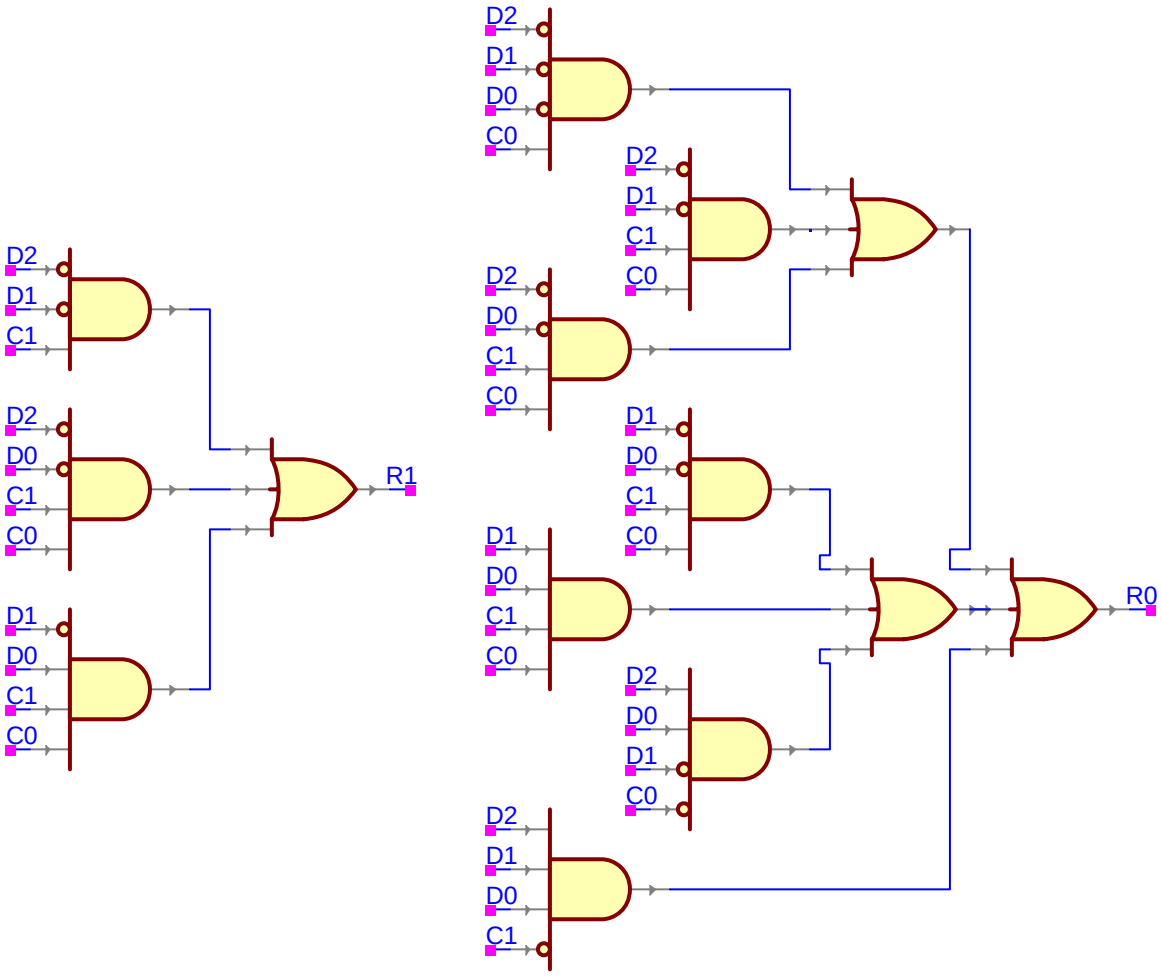
(b) K-Maps for R_1 and R_0:



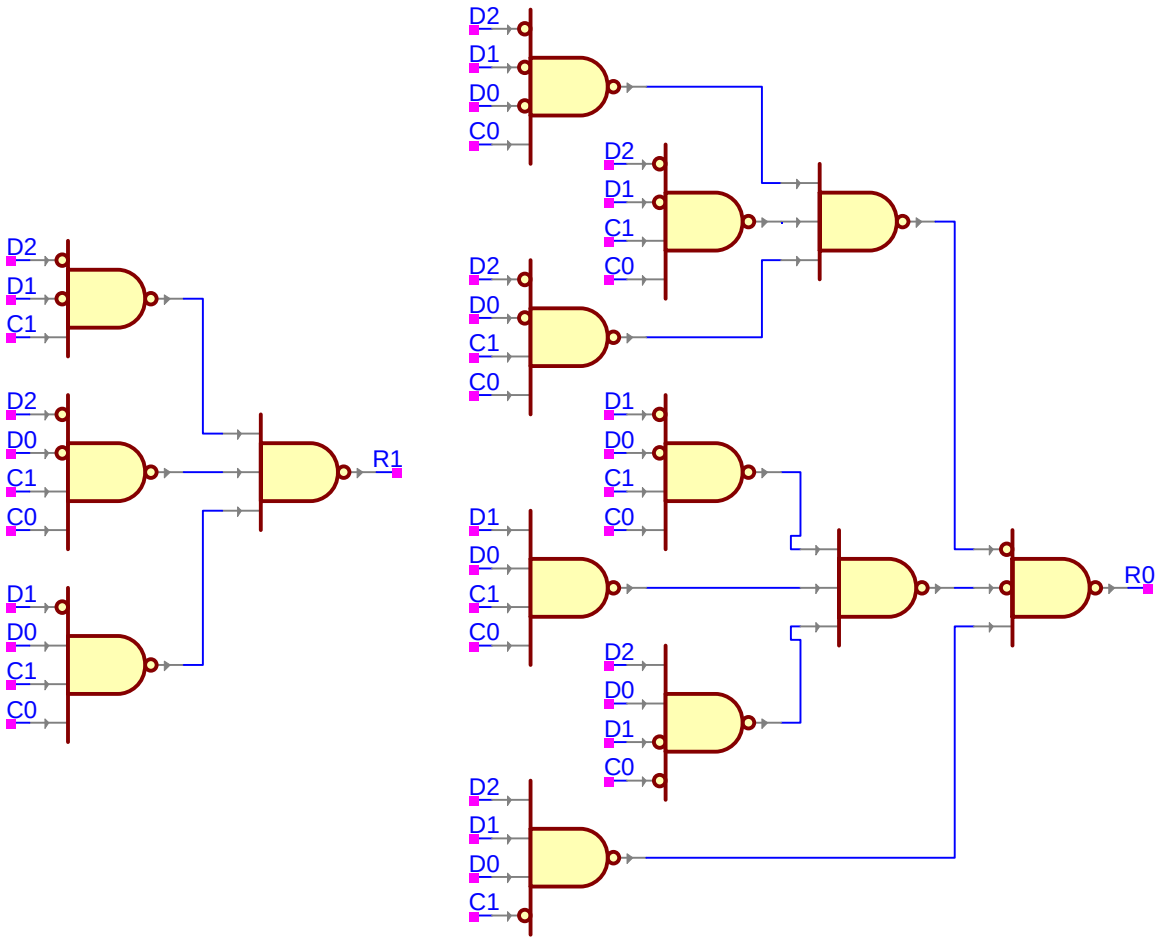
$R_1 = D_2'D_1'C_1 + D_2'D_0'C_1C_0 + D_1'D_0C_1C_0$

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

(c) Circuit Schematics:



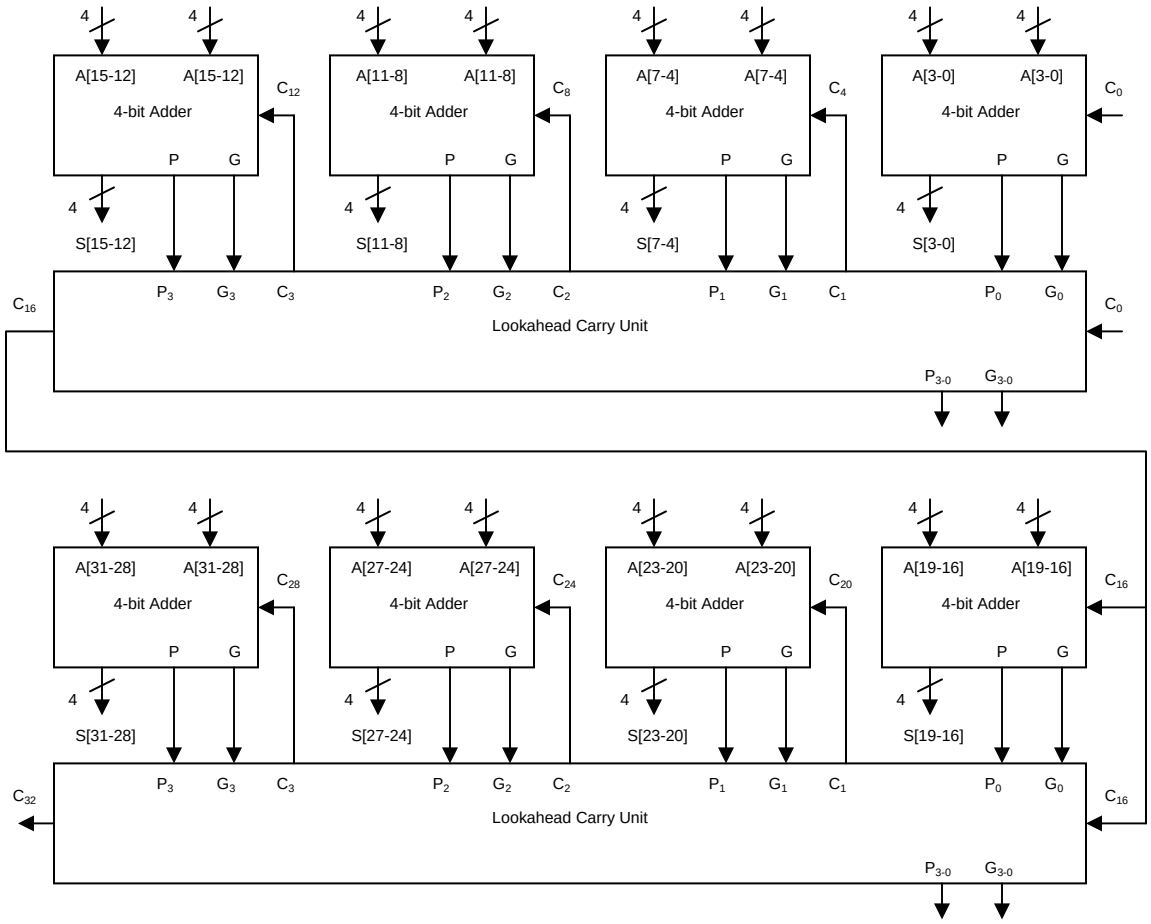
Initial Circuits



NAND-gate implementation

Exercise 5.9

(a) Block diagram for adder with carry lookahead:



32-bit Carry Lookahead Adder

The same idea is used for the 64-bit adder. We would use 4 of the 16-bit carry lookahead adders connected in series.

(b) The 32-bit implementation will have a critical delay of 13 time units. The 64-bit implementation will have a critical delay of 23 time units.

Exercise 5.10

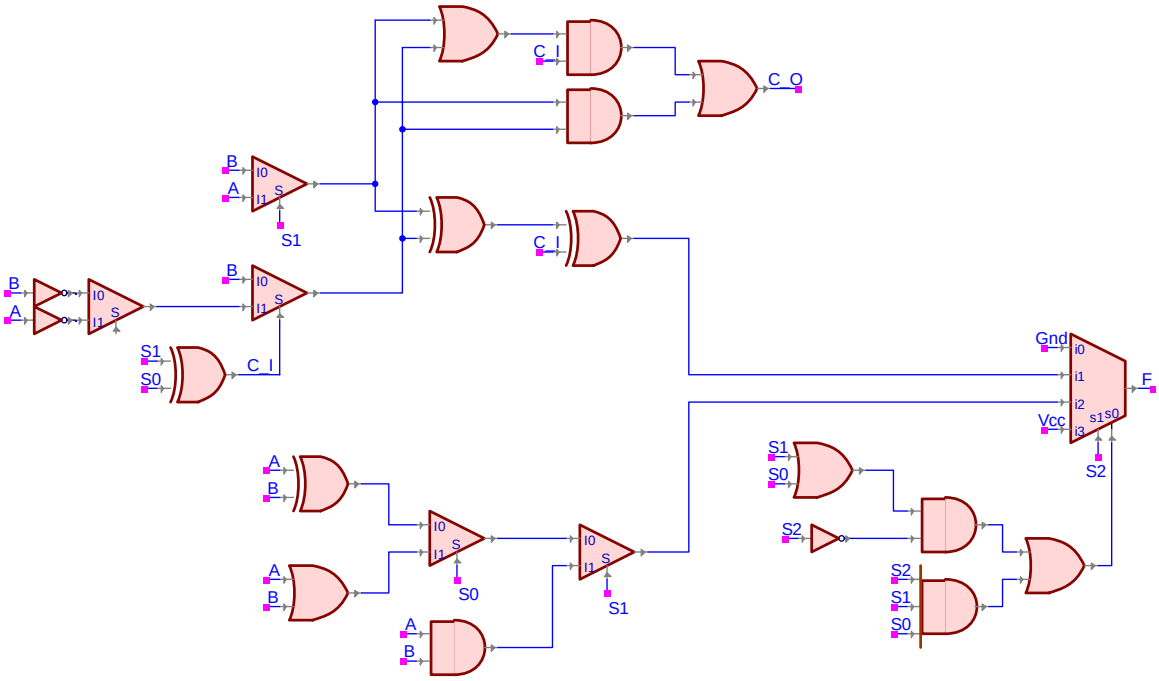
The critical path in carry-select adders is the highest order sum bit plus the delay through the multiplexor. Assuming that the 8-bit carry lookahead adder is implemented using two 4-bit adders and carry logic, the critical delay for the 8-bit adder will be 7. The multiplexor will add 2 to the overall delay, which makes the critical path for a 16-bit carry-select adder 9.

This is much better than a 16-bit ripple-carry adder which would have a critical delay of 18. The 16-bit carry-lookahead adder is slightly faster, with a critical delay of 8. However, as the adder gets bigger, the carry-select adder will be much faster than any other adder. A 32-bit carry-select adder, implemented in the same way as the 16-bit adder, has a critical delay of just 1 more.

Exercise 5.11

The main functionality of the ALU can be broken into four parts: 0, arithmetic, logic, and 1. Each of these is pieces shares functionality. We can use the control signals and small multiplexors to get the correct functionality. The four components come together with a 4:1 multiplexor, which is controlled with some logic that selects the correct input for a given S2-S0 value.

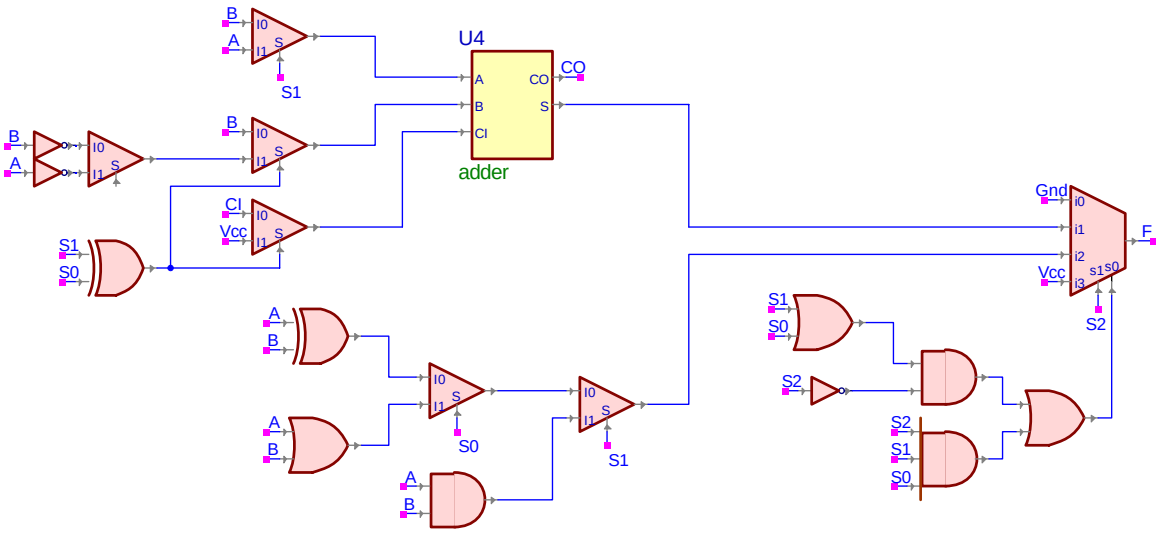
The separate components are grouped together in the following circuit schematic:



Exercise 5.12

The main functionality of the ALU can be broken into four parts: 0, arithmetic, logic, and 1. Each of these is pieces shares functionality. We can use the control signals and small multiplexors to get the correct functionality. The four components come together with a 4:1 multiplexor, which is controlled with some logic that selects the correct input for a given S2-S0 value.

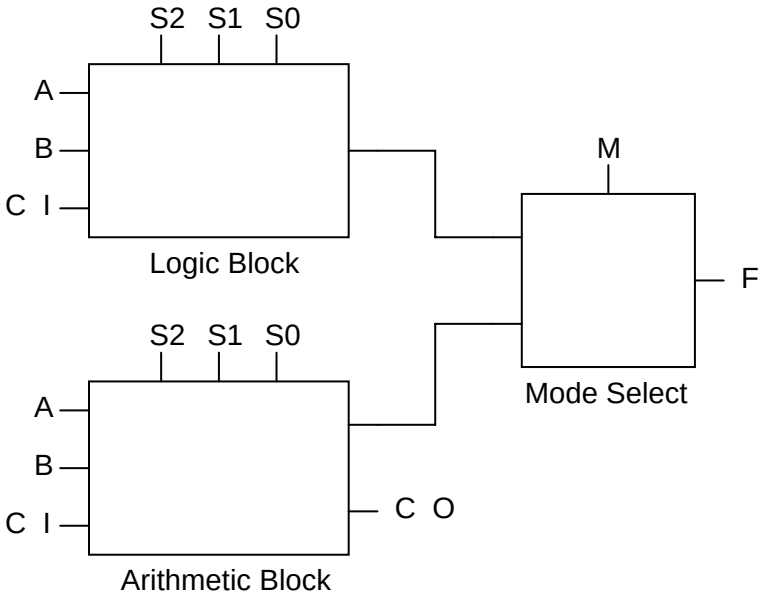
The separate components are grouped together in the following circuit schematic:



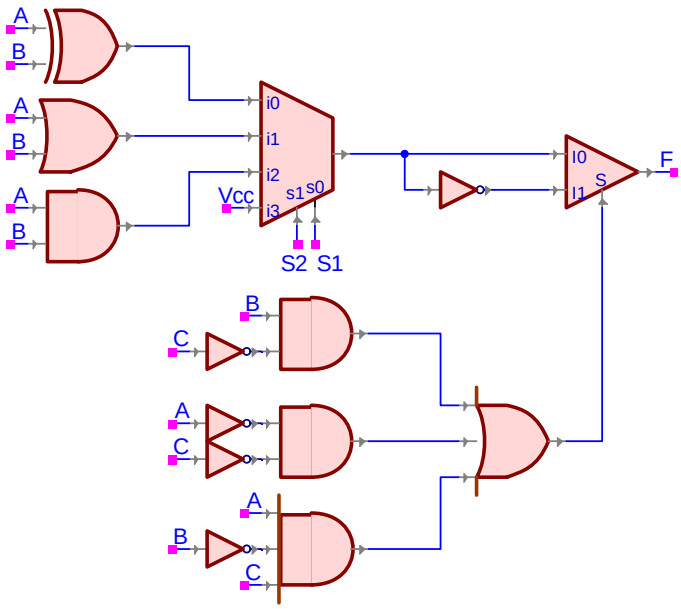
Exercise 5.13

Since this is a large design, we will implement it in a hierarchical-fashion using block diagrams at the top level and describe each of the components separately.

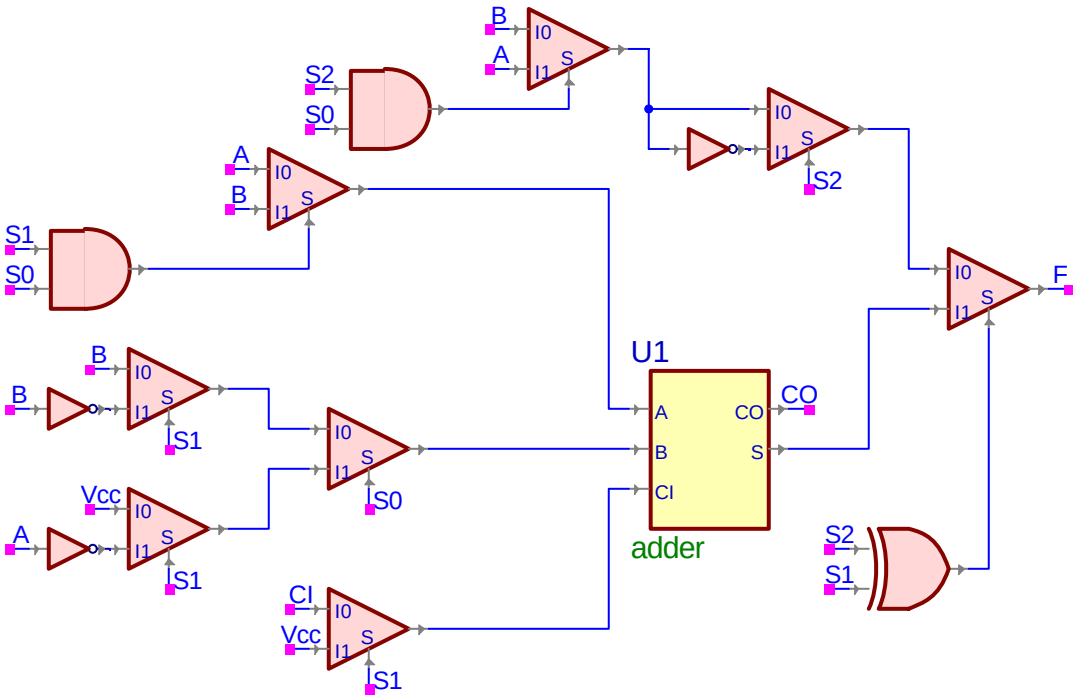
The top-level block diagram looks as follows:



The logic block is implemented with a 4:1 and 2:1 multiplexor along with some combinational logic. It works by first doing the logical operation and selecting the correct one with the 4:1 mux, then it negates the result if it needs to. One thing to note is this block doesn't do anything with CO, which is okay because we don't care what its value is for logic operations. The resulting circuit looks as follows:



This logic block uses a bunch of 2:1 multiplexors to select the correct inputs for the half-adder. It then uses more 2:1 multiplexors to output the correct value. The resulting circuit schematic looks as follows:



Lastly, the mode select block is just a 2:1 multiplexor that selects the arithmetic block if M is 1 and the logic block if M is 0.

Exercise 5.14

Since we implemented the ALU in Exercise 5.12 with an adder module, changing the design to use a carry-lookahead adder is simple. All of the wires for A and B would change into 4-bit busses. The carry-in input to the adder would remain unchanged. The half-adder would be replaced by a 4-bit carry-lookahead adder, which would take the two 4-bit A and B operands and the CI. It would output a 4-bit result and a carry-out. Everything else in the design would remain unchanged except for the fact that A and B are now 4-bit values.

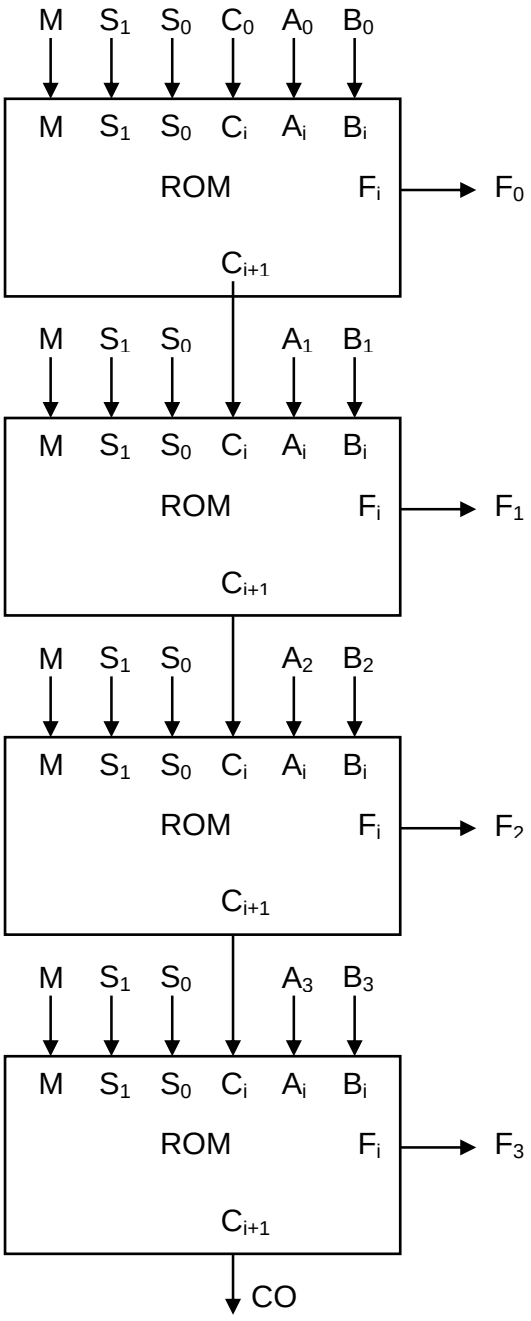
Exercise 5.15

Since we implemented the ALU in Exercise 5.13 with an adder module, changing the design to use a carry-lookahead adder is simple. All of the wires for A and B would change into 4-bit busses. The carry-in input to the adder would remain unchanged. The half-adder would be replaced by a 4-bit carry-lookahead adder, which would take the two 4-bit A and B operands and the CI. It would output a 4-bit result and a carry-out. Everything else in the design would remain unchanged except for the fact that A and B are now 4-bit values.

Exercise 5.16

One convenient property of a ROM is if you have a truth table, it is simple to implement a ROM. Each of the inputs to the truth table become the addresses inputs to the ROM and the outputs from the truth table are the outputs from the ROM. So, M, S_1, S_0, C_i, A_i, and B_i, become are the address inputs from most to least significant. The two outputs from the ROM are: F_i, and C_{i+1}.

We can implement a 4-bit ALU by cascading four of these ROMs as follows:



Exercise 5.17

The function for F_i is:

$F_i = S_iB_i$ xor (MC_i xor (S_0 xor A_i))

It is much easier to expand complicated expression if you express them as in terms of their operations with single variables and then substitute the complex expressions back into the equation at the end. So F_i becomes:

$F_i = W$ xor X, where $W = S_iB_i$, and $X = MC_i$ xor (S_0 xor A_i)

We can further simplify X as follows:

$X = Y$ xor Z, where $Y = MC_i$ and $Z = S_0$ xor A_i

Now we can begin our expansion:

$F_i = WX' + W'X$

$X = YZ' + Y'Z$

$Z = S_0A_i' + S_0'A_i$

Now we need to find expressions for W', X', Y', and Z' before we begin to substitute in:

$W' = S_i' + B_i'$

$X' = YZ + Y'Z'$

$Y' = M' + C_i'$

$Z' = S_0A_i + S_0'A_i'$

Now we can substitute everything back in and collect the terms. The resulting expression is:

$F_i = MS_1S_0C_iA_i'B_i + MS_1S_0'C_iA_iB_i + M'S_1S_0A_iB_i + M'S_1S_0'A_i'B_i + S_1S_0C_i'A_iB_i + S_1S_0'C_i'A_i'B_i + MS_1'S_0C_iA_i + MS_1'S_0'C_iA_i' + M'S_1'S_0A_i' + M'S_1'S_0'A_i + S_1'S_0C_i'A_i' + S_1'S_0'C_i'A_i + MS_0C_iA_iB_i' + MS_0'C_iA_i'B_i' + M'S_0A_i'B_i' + M'S_0'A_iB_i' + S_0C_i'A_i'B_i' + S_0'C_i'A_iB_i'$

This is correct because for every row in the truth table that F_i is true, we have a term in the expression that corresponds to that entry.

Applying the same method for C_{i+1}, we get the following expression:

$C_{i+1} = MS_1S_0C_iA_iB_i + MS_1S_0'C_iA_i'B_i + M'S_1S_0A_iB_i + M'S_1S_0'A_i'B_i + S_1S_0C_i'A_iB_i + S_1S_0'C_i'A_i'B_i + MS_0C_iA_i' + MS_0'C_iA_i$

Again, all of the terms in the expression match up to the rows in the truth table which are true for C_{i+1}.

Exercise 5.18

BCD addition is similar to binary addition except that if the result is greater than nine, we need to add six to it to keep it in BCD form.

(a) 0001 + 0100 = 0101

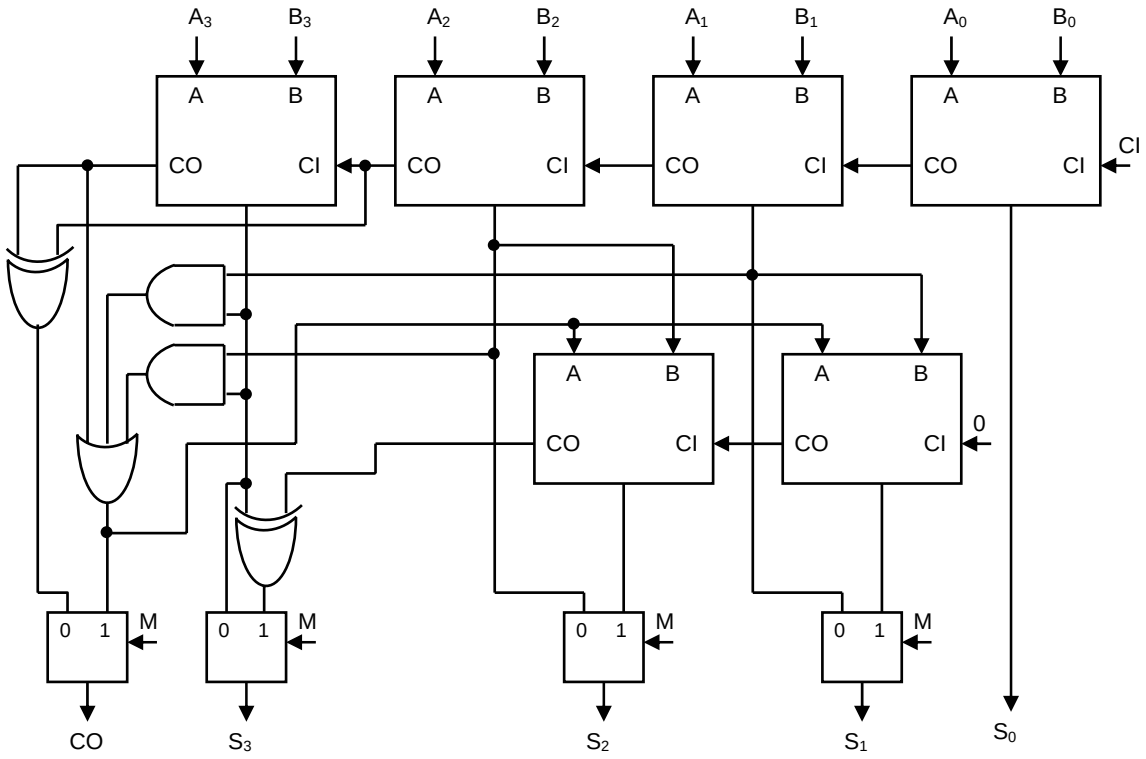
(b) 1000 + 1001 = 1 0001 + 0110 = 1 0111

(c) 0111 + 0011 = 1010 + 0110 = 1 0000

(d) 1001 1001 + 0001 0001 = 1010 1010 + 0110 0110 = 1 0001 0000

Exercise 5.19

By adding 2:1 mulitplexors on the outputs that are different for the BCD adder, CO, S_3, S_2, and S_1, we are able to create an adder that works as a standard binary adder or a BCD adder. The input M is the control signal for the added multiplexors. The resulting circuit looks as follows:



Exercise 5.20

A = 1101 and B = 1011. The result bits are in bold.

S_0: A_0B_0 =>	**1**	
S_1: A_1B_0 =>	0	
A_0B_1 =>	+ 1	
	1	
S_2: A_0B_2 =>	0	
A_1B_1 =>	+ 0	
	0	
A_2B_0 =>	+ 1	
	1	
S_3: A_0B_3 =>	1	
A_1B_2 =>	+ 0	
	1	
A_2B_1 =>	+ 1	
	1 0	Carry-out for second S4 add carry-in
A_3B_0 =>	+ 1	
	1	
S_4: A_1B_3 =>	0	
A_2B_2 =>	+ 0	
	0 1	Carry-in from S3
A_3B_1 =>	+ 1	
	1 0	Carry-out for second S5 add operand
	+ 0	Operand from third S3 add carry-out
	0	
S_5: A_2B_3 =>	1	
A_3B_2 =>	+ 0	
	1	
	+ 1	Operand from second S5 add carry-out
	1 **0**	
S_6: A_3B_3 =>	1 1	Carry-in from S5
	+ 1	Operand from second S5 add carry-out
	1 **0**	
S_7:	**1**	Bit from S6 carry-out

Result: 10001111, which is correct.

Exercise 5.21

A = 1101 and B = 1011. The ones that are carry-outs/ins are underlined and the result bits are in bold.

P_0: $(A_0B_0) + 0 = \mathbf{1}$

P_1: $(A_1B_0) + 0 = 0 + (A_0B_1) = \mathbf{1}$

P_2: $(A_2B_0) + 0 = 1 + (A_1B_1) = 1 + (A_0B_2) = \mathbf{1}$

P_3: $(A_3B_0) + 0 = 1 + (A_2B_1) = \underline{1}\ 0 + (A_1B_2) = 0 + (A_0B_3) = \mathbf{1}$

P_4: $(A_3B_1) + 0 + \underline{1} = \underline{1}\ 0 + (A_2B_2) = 0 + (A_1B_3) = \mathbf{0}$

P_5: $(A_3B_2) + \underline{1} = 1 + (A_2B_3) = \underline{1}\ \mathbf{0}$

P_6: $(A_3B_3) + 0 + \underline{1} = \underline{1}\ \mathbf{0}$

P_7: **1**

Result: 10001111, which is correct.

Exercise 5.22

The worst-case propagation delay through the multiplier in Figure 5.25 is 15 gates. This is the carry-out path through the first adders in S_1 and S_2, then the Sum paths for the first two adders and the carry-out path for the last adder in S_3, then the carry-out path through the last adder in S_4, S_5, and S_6.

Exercise 5.23

Designing a 2's compliment multiplier is not as difficult as it may seem at first. A nice property about multiplying signed numbers is the sign of the output is relatively easy to figure out. If the signs are the same, the result is positive, if they are different, the result is negative. This can be accomplished by XORing the highest-order bit of both operands together.

The operands need to be converted into a positive number before they are passed into the magnitude multiplier. This can be accomplished by flipping the bits and adding 1 if the high order bit is a one.

Putting the result back into 2's compliment is achieved in the same fashion. If the result of the XOR on the high order bits of the operands is a 1, the result's bits need to be flipped and have 1 added to them to turn it back into a negative 2's compliment number.

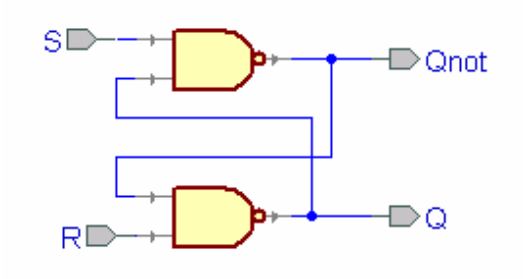
One thing to note is the range for a 4x4 2's compliment multiplier is -64 to 49. So the multiplier would still work if the result was 7 bits, rather than eight.

Exercise 5.25

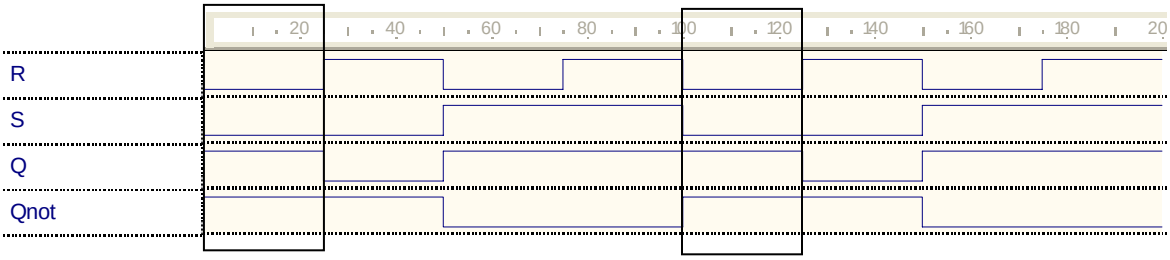
The 4x4-combinational multiplier, in Figure 5.26, can be slightly rearranged to provide better propogation delay. The way to do this is to have the carry-outs pass their value to the adder that is diagonal from them rather than in front of them. This will net a propagation delay which is 4 gates better. The reason for this is that the carry-out has to propagate through one more gate then the sum does, so the adders in each row will all have valid inputs 1 gate delay sooner.

Exercise 6.1

The cross-coupled NAND gates would produce the following latch:



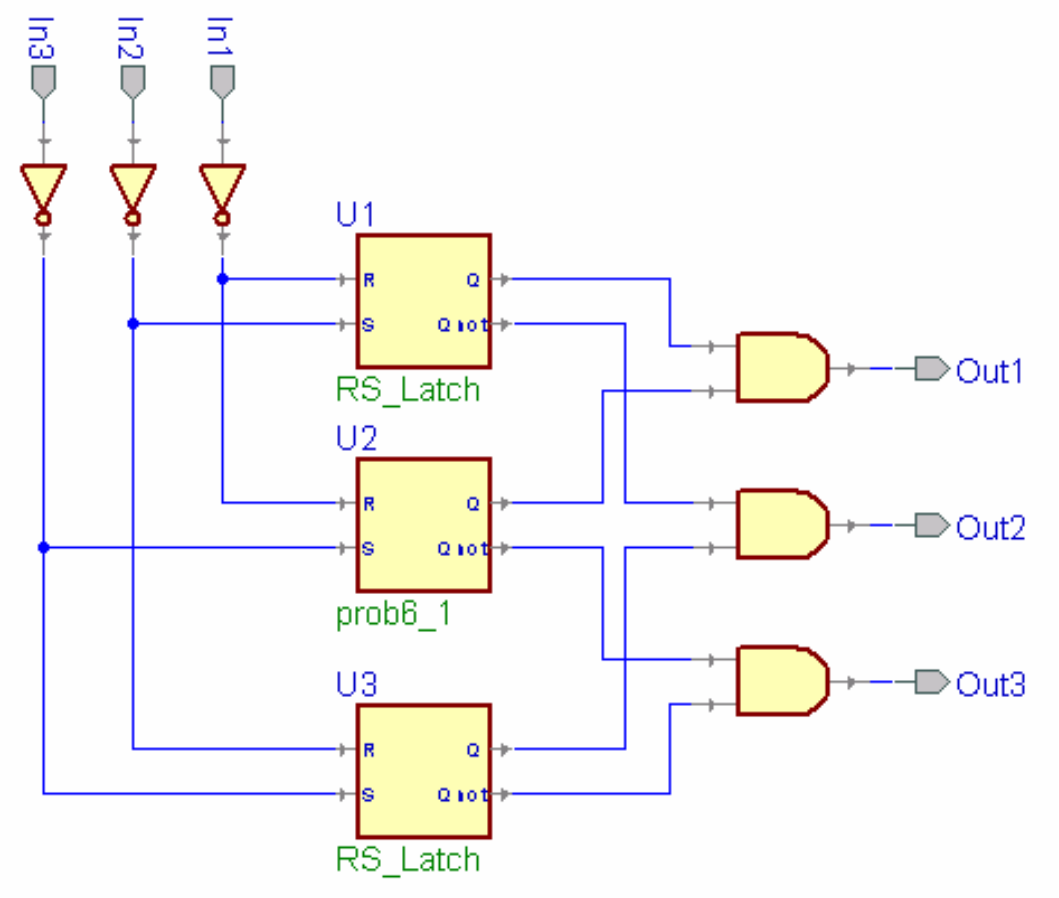
This produces the following waveform diagram:



The shaded regions above denote forbidden inputs because they cause both Q and Q' to be asserted. The input signal S causes Q^+ to be asserted, and the input signal R causes Q^+ to be reset; however when both R and S are asserted Q^+ will take whatever the input value of Q is.

S	R	Q	Q^+
0	0	X	Forbidden
0	1	X	0
1	0	X	1
1	1	0	0
1	1	1	1

Exercise 6.2



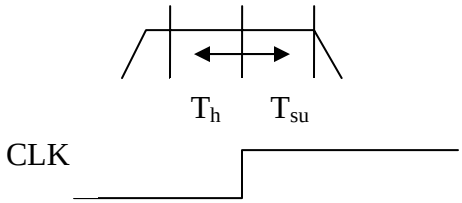
Extending this solution to either n = 12 or n = 30 inputs requires

$$\sum_{i=2}^{n}(i-1)$$

R-S latches. Where each output would have an AND gate with n-1 inputs. This would cause significant delay and could be quite noticeable.

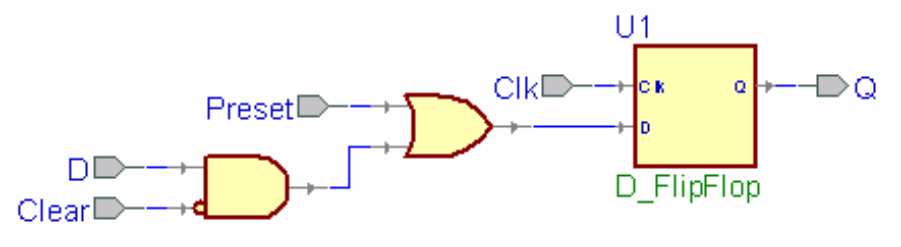
Exercise 6.3

If there were such a thing as negative setup and hold time it might look something like the following:

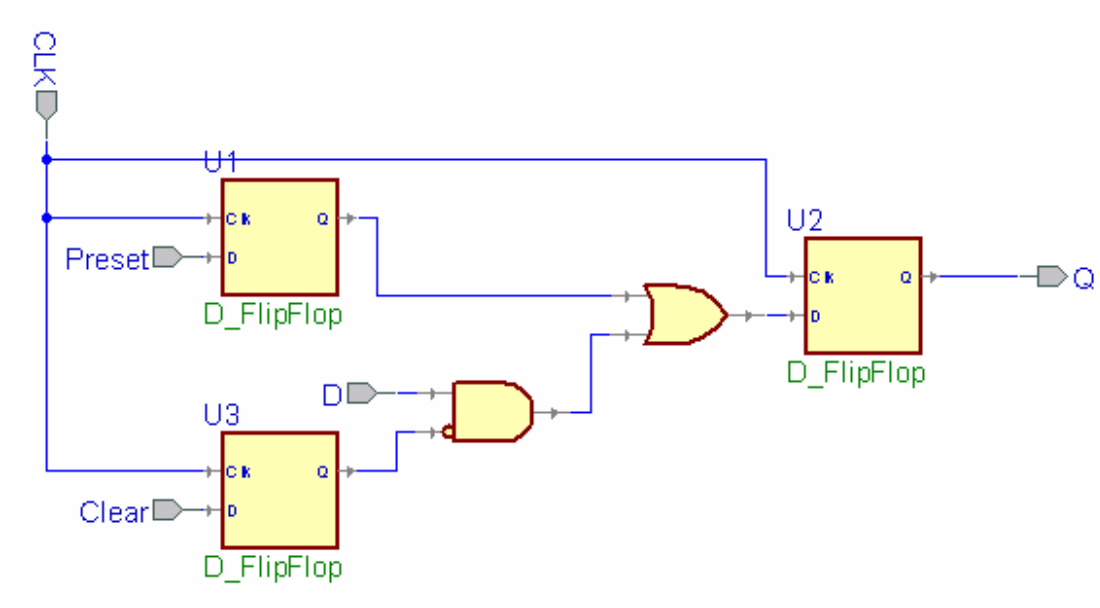


This however does not make a lot of sense because T_{su} is defined to be the time before the CLK event in which the signal must remain stable, and T_h is the time after the CLK event in which the signal must remain stable. Thus negative times would in effect switch the function of T_{su} and T_h so that T_{su} is the amount of time after the clock and T_h is the time before the clock.

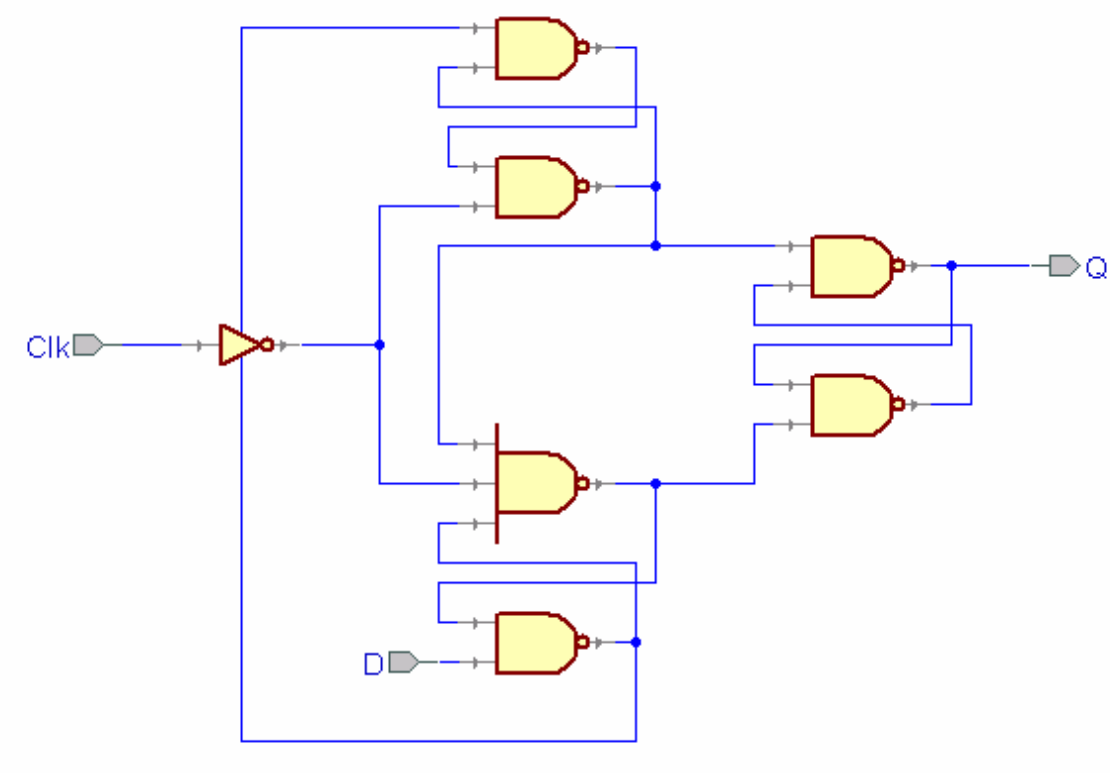
Exercise 6.4



Exercise 6.5

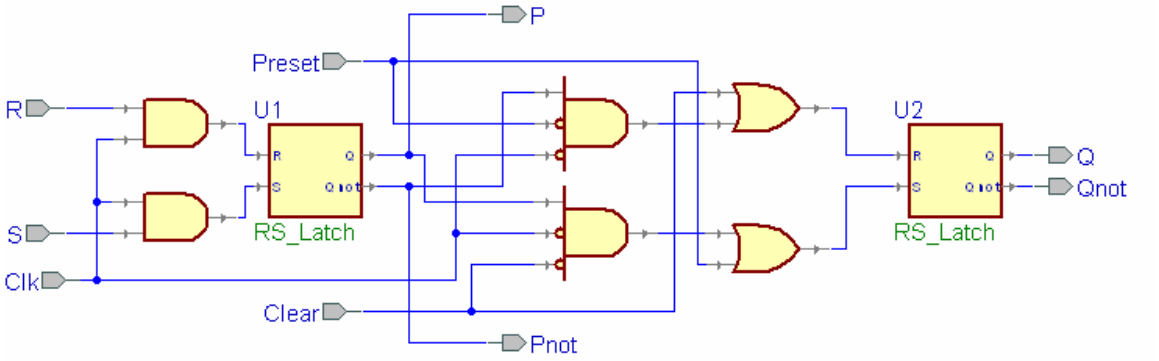

CLK
U1
Clk
Q
D
Preset
D_FlipFlop
U2
Clk
Q
D
D_FlipFlop
Q
D
U3
Clk
Q
D
Clear
D_FlipFlop

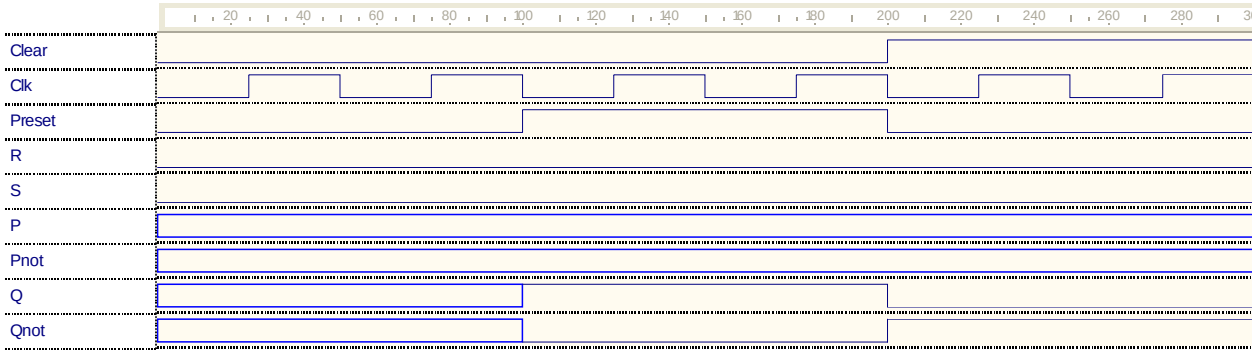
Exercise 6.6



Removing the inverter on the clock will make this a positive-edge triggered D-flip flop.

Exercise 6.7


P
Preset
R
U1
U2
Q
Qnot
S
RS_Latch
RS_Latch
Clk
Clear
Pnot


20
40
60
80
100
120
140
160
180
200
220
240
260
280
Clear
Clk
Preset
R
S
P
Pnot
Q
Qnot

Exercise 6.8

Ones-catching happens when one of the inputs glitches and gets asserted while the clock is high. This causes the Master Output to pickup the signal and get set, even though the glitch is finished before the clock goes low. After the Master Output picks up the glitch, the Slave Outputs catch the one as well.

Zeroes-catching can happen depending on the construction of the Master-Slave flip-flop. If instead of constructing the flip-flop with only NOR gates, NAND gates were used (i.e. a R'-S' flip-flop), then the Master-Slave will pick up zeroes instead of ones.

Exercise 6.9

J-K flip-flop's characteristic equation is:

$$Q+ = QK' + Q'J$$

Substituting D' for K and D for J:

$$Q+ = QD + Q'D = (Q + Q')D = D$$

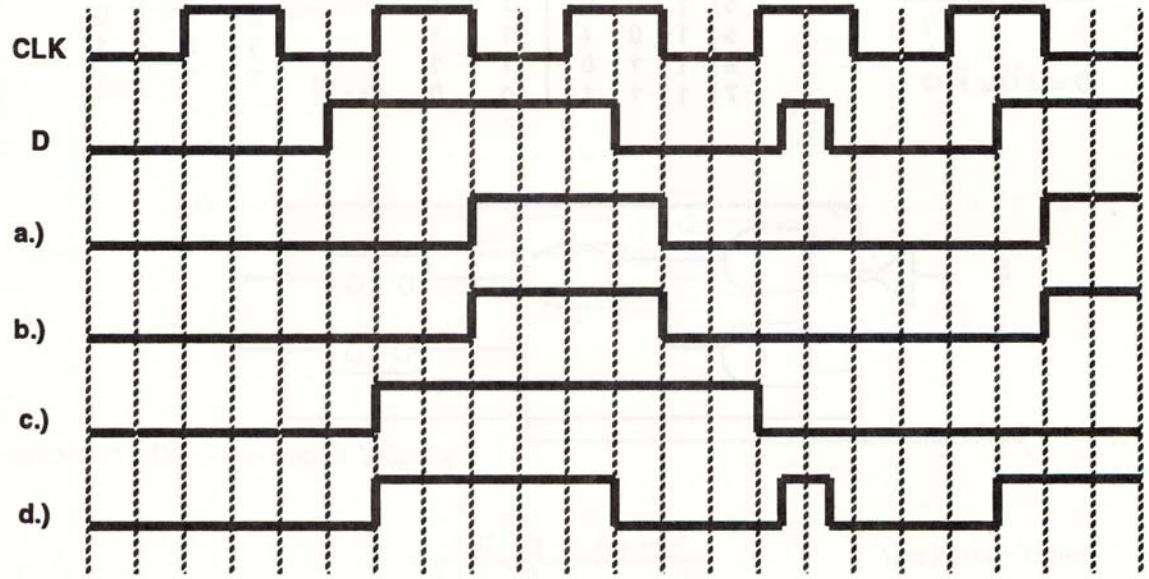
Which is precisely the characteristic equation for a D flip-flop:

$$Q+ = D$$

Exercise 6.10

KEY:

a.) negative edge-triggered flip-flop
b.) master/slave flip-flop
c.) positive edge-triggered flip-flop
d.) clocked latch



Exercise 6.11

(a) False. The level-sensitive latch continuously updates its outputs based on its inputs only when the clock is asserted. When the clock is unasserted, the outputs hold their previous state, and the inputs can fluctuate in order to compute the next values for the outputs.

(b) False. The time after the clock edge until the outputs change is called the propagation delay, or T_p. As stated at the end of 6.2.1, T_p must be greater than T_h in order for flip-flops to be cascaded properly.

(c) True. Provided that forbidden inputs are not fed into the clocked latches or flip-flops. Because there is zero setup or hold times, then events occur instantaneously after a clock-edge, this means that there is no time at which a signal could be misread during the setup or hold times.

(d) False. While it is true that a master-slave flip-flop does behave similarly to a clocked latch, and that its output can change only near an edge of the clock, depending on the design of the master-slave flip-flop, this change can occur at the rising edge, the falling edge, or both (but both is rarely done in practice).

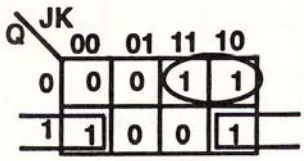
(e) False. The D Flip-flop is constructed using an R-S master-slave latch with an additional inverter for the signal D'. In the case of the J-K flip flop, there is an R-S master-slave latch as well as two additional AND gates for the inputs and feedback. Therefore the J-K takes one more gate than the D Flip-flop.

Exercise 6.12

a) Master-slave RS flip-flop
b) Master-slave D flip-flop
c) Master-slave T flip-flop
d) Clocked RS latch
e) Positive edge-triggered D flip-flop

Exercise 6.13

Implement a JK flip-flop from a D flip-flop

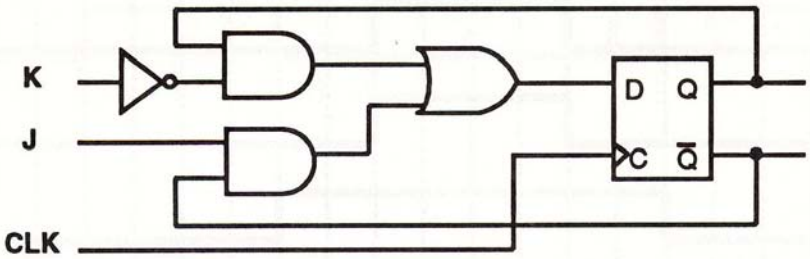


$D = J\overline{Q} + \overline{K}Q$

	J	K	Q	Q+	D
0	0	0	0	0	0
1	0	0	1	1	1
2	0	1	0	0	0
3	0	1	1	0	0
4	1	0	0	1	1
5	1	0	1	1	1
6	1	1	0	1	1
7	1	1	1	0	0

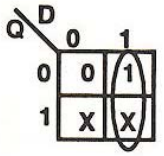
Excitation Table

Q	Q+	D
0	0	0
0	1	0
1	0	1
1	1	1

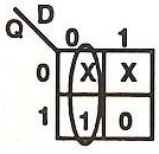


Exercise 6.14

Implement a D flip-flop from a J-K flip-flop



J = D

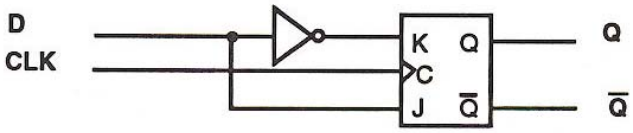


K = $\overline{D}$

	D	Q	Q+	J	K
0	0	0	0	0	X
1	0	1	0	X	1
2	1	0	1	1	X
3	1	1	1	X	0

Excitation Table

Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



Exercise 6.15

Implement a D flip-flop from a T flip-flop

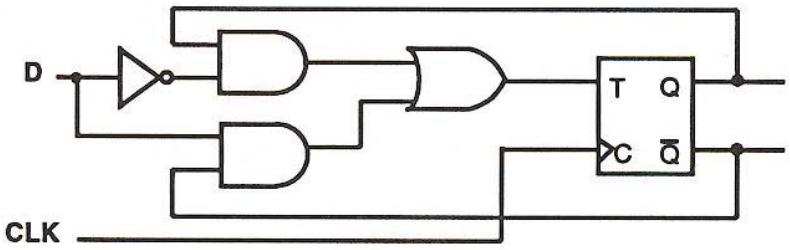
Q \ D	0	1
0	0	1
1	1	0

	D	Q	Q+	T
0	0	0	0	0
1	0	1	0	1
2	1	0	1	1
3	1	1	1	0

Excitation Table

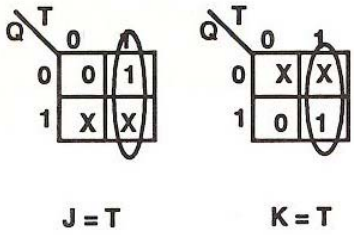
Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

$T = D\,\overline{Q} + \overline{D}\,Q$



Exercise 6.16

Implement a T flip-flop from a J-K flip-flop



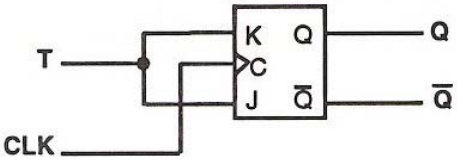
J = T

K = T

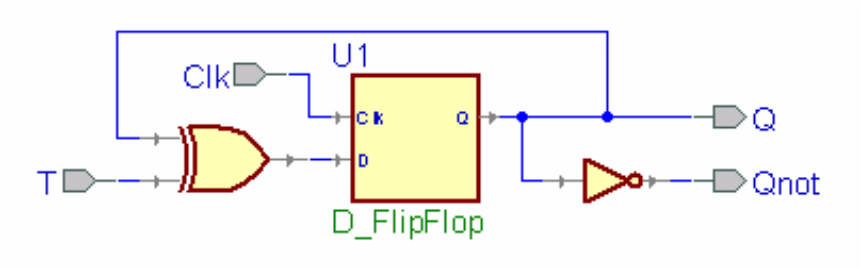
	T	Q	Q+	J	K
0	0	0	0	0	X
1	0	1	1	X	0
2	1	0	1	1	X
3	1	1	0	X	1

Excitation Table

Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



Exercise 6.17



In order to get the T flip-flop behavior out of the D flip-flop, whenever T is unasserted, Q will have to maintain its current value, and whenever T is asserted, Q^+ will take on Q'. This is achieved by using the equation: $Q^+ = Q \otimes T$.

Exercise 6.18

The worst case time skew is calculated by taking the equation:

$$T_p > T_{skew} + T_h$$

Which in this case takes on the form:

$$3.6 > T_{skew} + 0.5$$

or

$$3.1 > T_{skew}$$

Exercise 6.19

(a) The problem specifies that T_{su} = 20 ns, T_{pd} = 13 ns, and T_h = 5 ns. Frequency is measured in cycles per second, so $f = 1 / T_{period}$, where T_{period} is defined to be greater than $T_{su} + T_{pd}$. Therefore the math shows:

$$T_{period} > T_{pd} + T_{comb} + T_{su}$$

$$T_{period} > 13 \text{ ns} + 0 \text{ ns} + 20 \text{ ns}$$

$$T_{period} > 33 \text{ ns}$$

$$f < 1 / T_{period}$$

$$f < 30.3 \text{ MHz}$$

(b) Since the clock period must account for the worst-case propagation delay we will use 100ns for the combinational logic delay. The calculation is the same as part (a). Thus:

$$T_{period} > T_{pd} + T_{comb} + T_{su}$$

$$T_{period} > 13 \text{ ns} + 100 \text{ ns} + 20 \text{ ns}$$

$$T_{period} > 133 \text{ ns}$$

$$f < 1 / T_{period}$$

$$f < 7.5 \text{ MHz}$$

Exercise 6.20

The system failure could be caused by the fact that an asynchronous signal is received every 200 ns, and the period of the 25 MHz receiver is 40 ns. In general, clock skew will provide enough difference between when a signal is sent and received, such that the signal is received properly and respects the setup and holding times of the synchronizer.

There are three common ways for dealing with this problem:

- Increasing the cycle time of the receiver, so that it has more time to determine respect the setup and hold times of its components.
- Adding a second synchronizer, which helps lower the probability of a synchronizer failure.
- Implement a handshaking methodology between the client and server, so that it is easy to confirm whether or not information has been communicated effectively.

The first option severely has the side-effect that it severally slows down the receiver, and is not very tolerable in today's high-performance requirements.

The second option helps reduces the error, but also reduces how quickly a signal can be interpreted and used in the receiving system. This can make a huge difference in systems that are trying to communicate quickly.

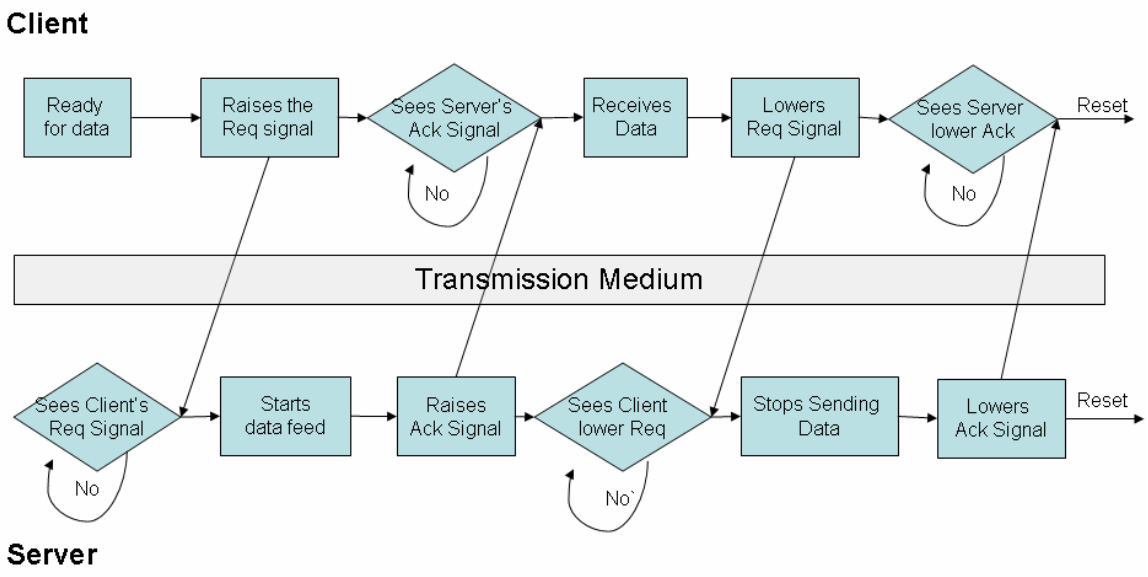
The final option is more commonly practiced, because it allows both the sender and receiver to communicate when they are ready to both send and receive signals. This coordination, helps reduce the amount of errors, and does not impose to much of a time penalty.

Exercise 6.21

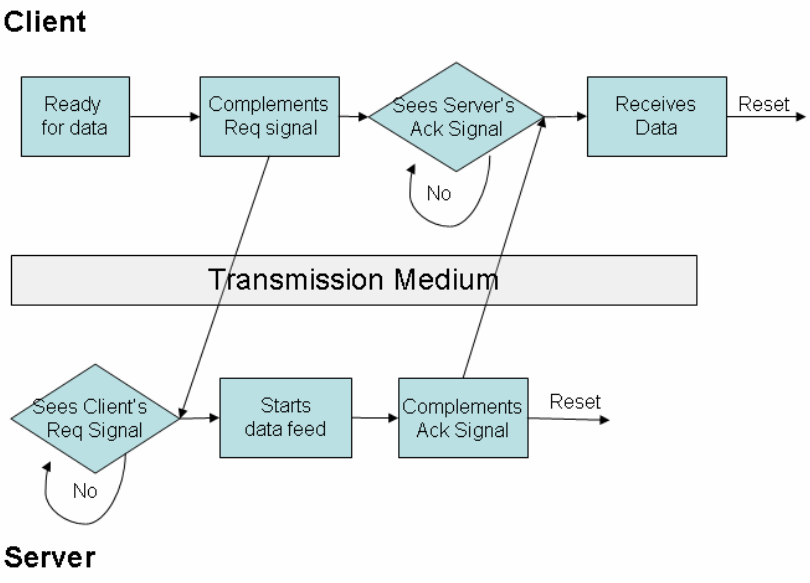
Placing two synchronizers in series helps reduce the probability of synchronizer failure because it is in effect providing the system two clock periods to stabilize instead of one. This effect is very similar to increasing the clock period of the receiver so that it has more time to decide. Even though it seems logical to put more synchronizers in series to even further reduce the probability of synchronizer failure, this approach has the severe penalty in that each synchronizer adds another full clock period before the input can be used. The Oscilliscope sketch in Figure 6.41 demonstrates that, in general, signals have low probability of staying in a metastable state for long durations of time. This means that after two synchronizers, the series is paying heavy time delays for minor improvements to the probability.

Exercise 6.22

The figure below shows the process for four-cycle handshaking:

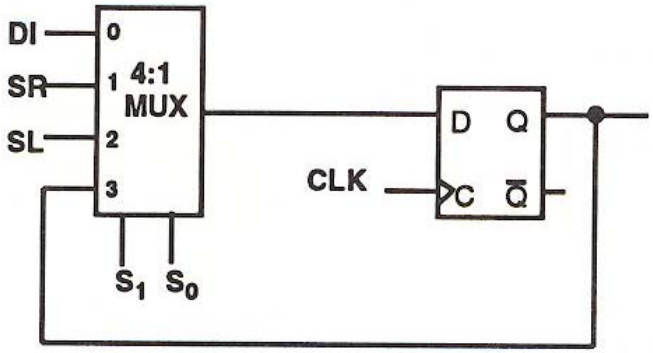


In this figure, reset means that both Client and Server are back in their initial states, with the client preparing to receive data, and the server waiting for the client's request.



In two-cycle handshaking, the Client and Server both keep track of the state provided by the other. This way the Server can recognize when the Client complements the Req signal, and the Client can in turn recognize the Server complementing the Ack signal. In general, maintaining the state in a two-cycle handshaking model is more complicated than implementing the four-cycle handshaking model.

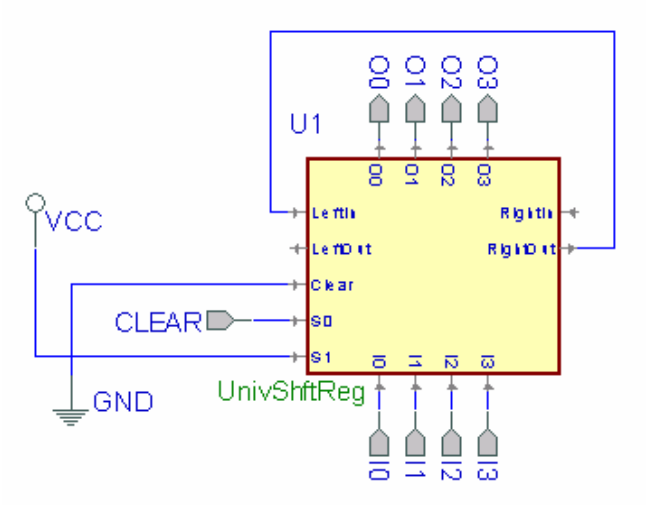
Exercise 6.23



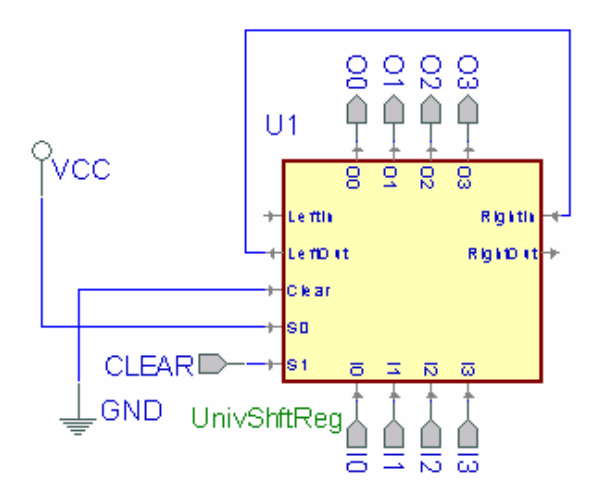
Exercise 6.24

In the diagrams below, the CLEAR signal is used to do a parallel load of data into the universal shift register. The clear functionality is not required, but in general will provide more interesting, and useable shift registers.

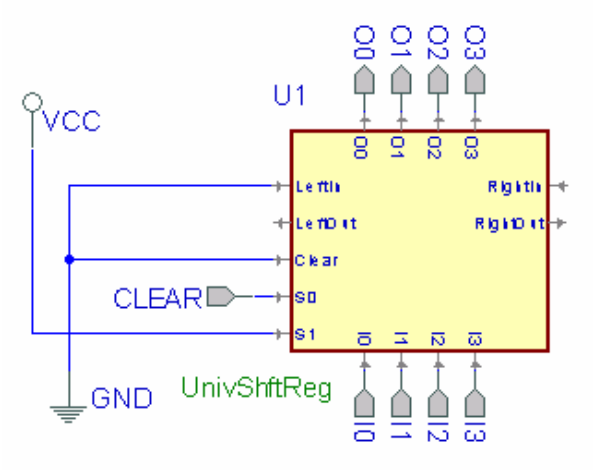
(a) For the circular shift right, S1 will always be enabled, and the output from RightOut will be fed into LeftIn. If CLEAR is enabled as well, the shift register will load in values I0…I3 instead of performing the circular shift.



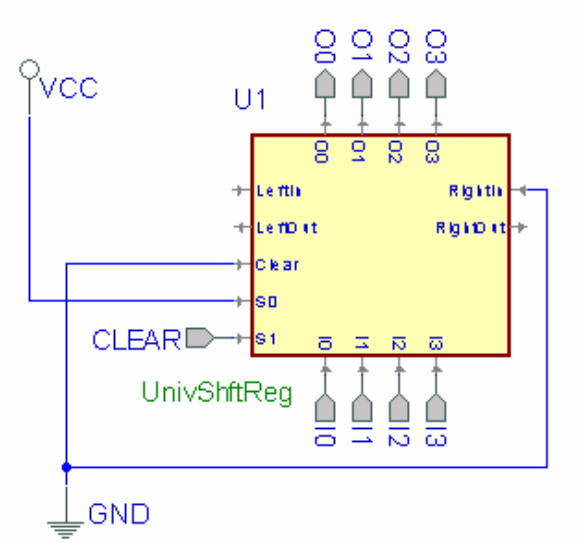
(b) The implementation of the left circular shift register is very similar to that of the right:



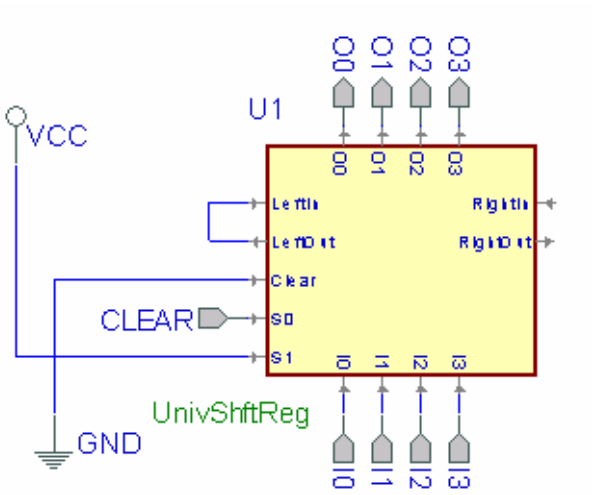
(c) The logic shift right:



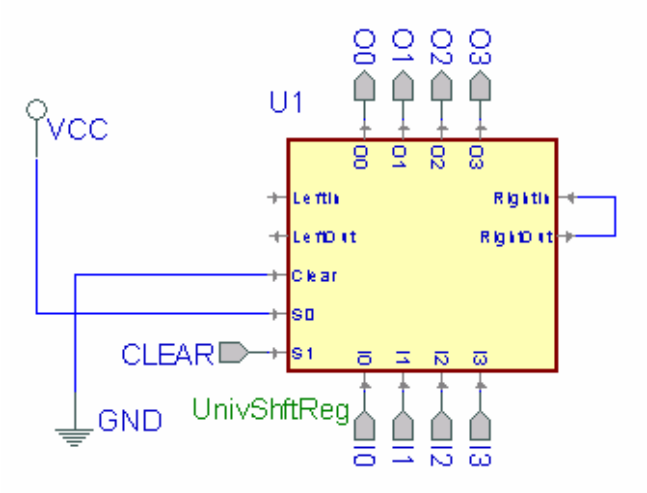
(d) The logic shift left:



(e) Arithmetic shift right:

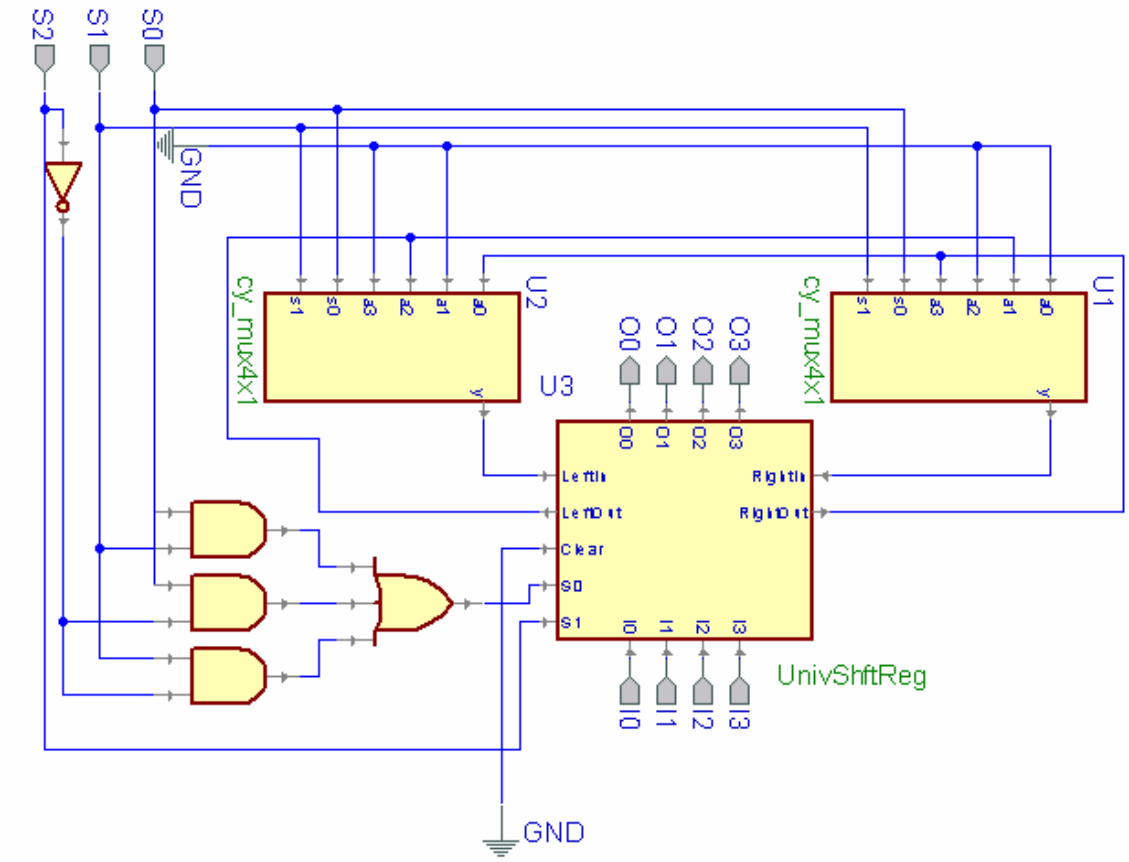


(f) Arithmetic shift left:



Exercise 6.25

(a) Here is the full shift register system:



(b) Since the 74194 uses S_0 and S_1 as its control signals, and those are already being used in this problem, let C_0 and C_1 represent the 74194 control signals instead.

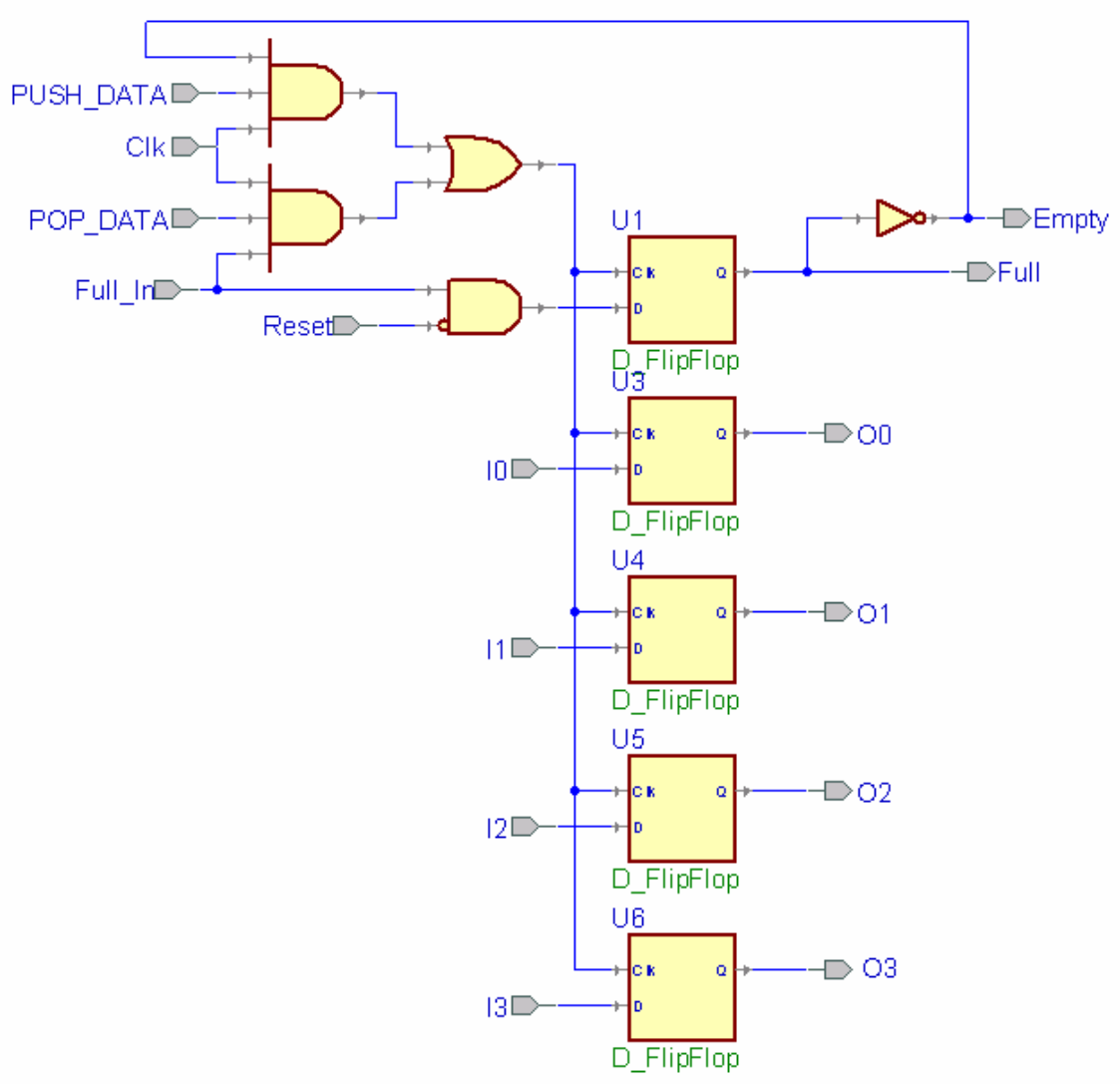
$C_0 = S_0S_1 + S_0S_2' + S_1S_2'$

$C_1 = S_2$

Notice that the shift register input functions are only dependent on S_0 and S_1. This is because S_3 alternates whether or not the function is a "left" or "right" operation, except where hold and preset are concerned. However, hold and preset do not look at the input values, and thus are don't-cares in terms of the the LeftIn and RightIn inputs.

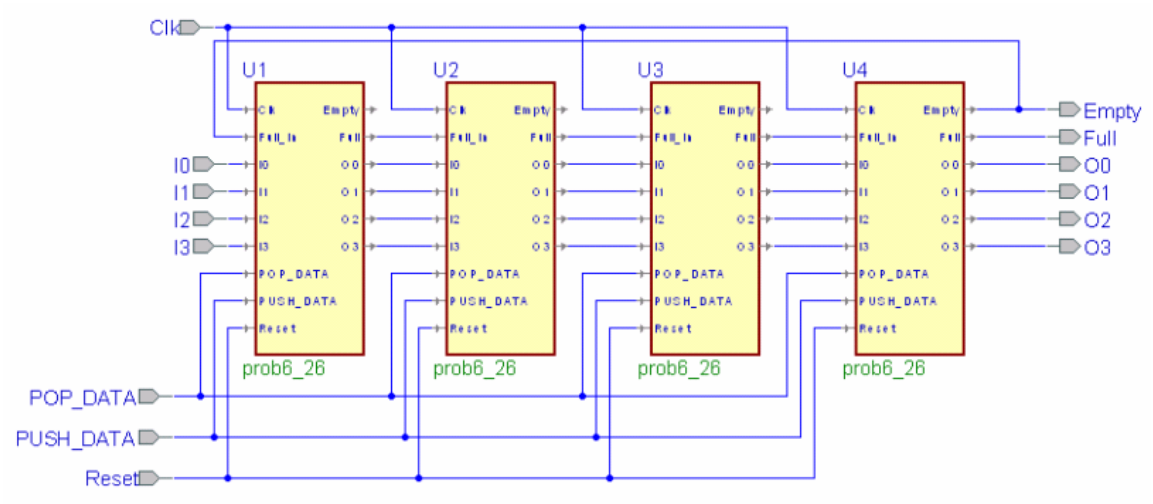
Exercise 6.26

Since there is a lot of repeated functionality, the implementation given is broken up into two main modules. The first module handles a single 4-bit word, and the shift register responsible for the Full and Empty status.



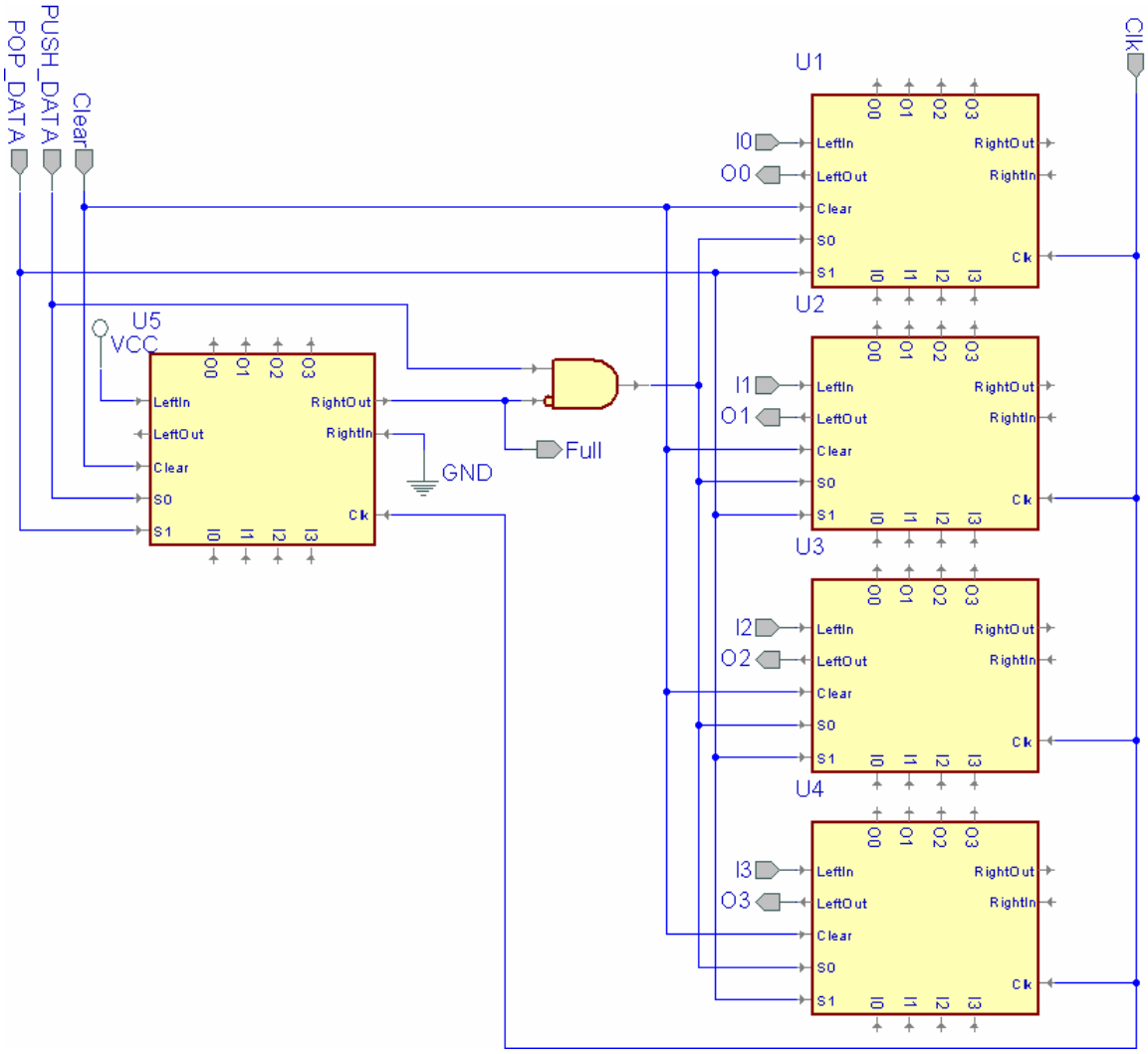
The restraints on Pushing and Popping are maintained, since the Clock signal will only be enabled if the module feeding Full_In is asserted and it is a Pop instruction or this module is outputting Empty and it is a Pop instruction.

The diagram below shows how to chain the individual components together.



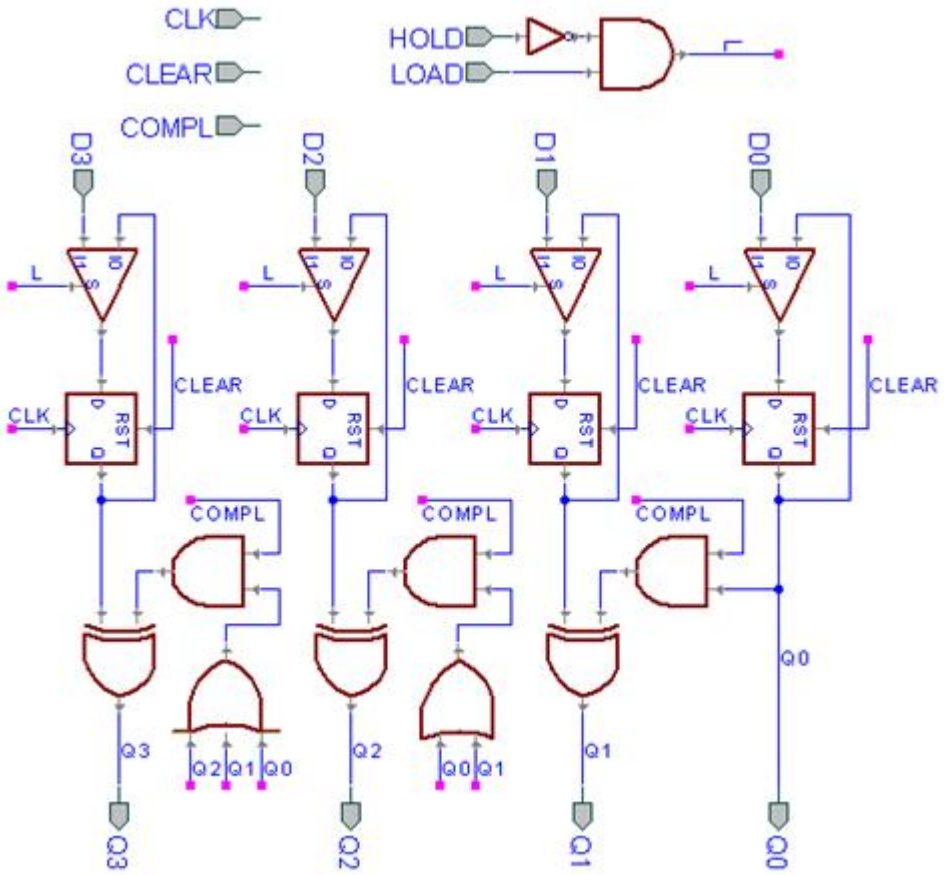
Exercise 6.27

Using 5 universal shift registers, the following implementation implements the 4-bit LIFO stack. The stack is determined to be full if the left-most shift register in the diagram has RightOut asserted. This is to say that whenever something is pushed onto the stack, a logical 1 will move over one space to the right in the shift register. Whenever a pop occurs, a logic 0 is shifted onto the left of the register.

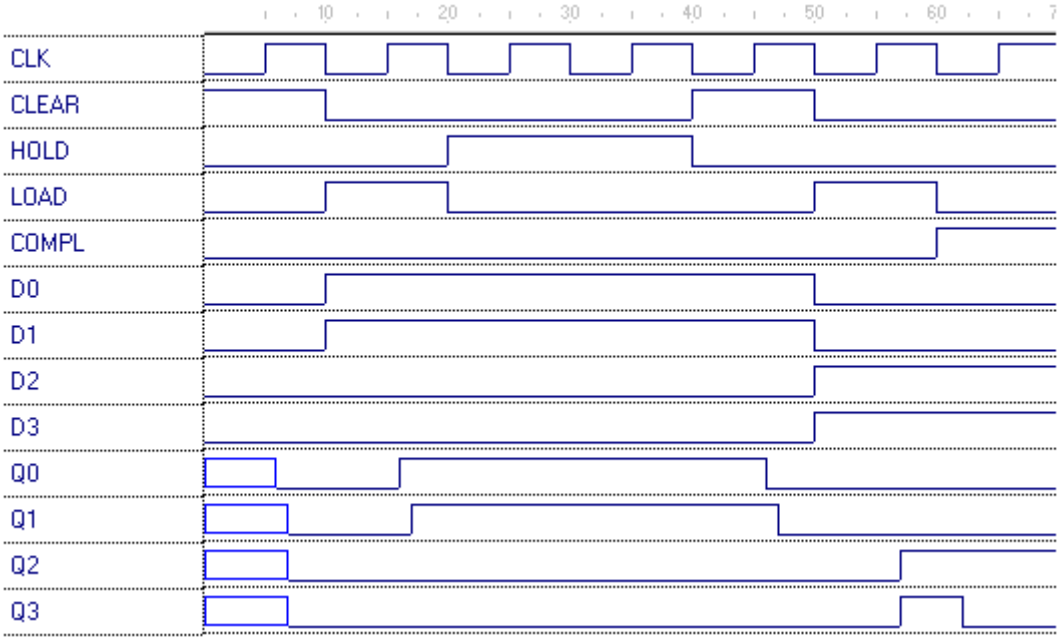


Exercise 6.28

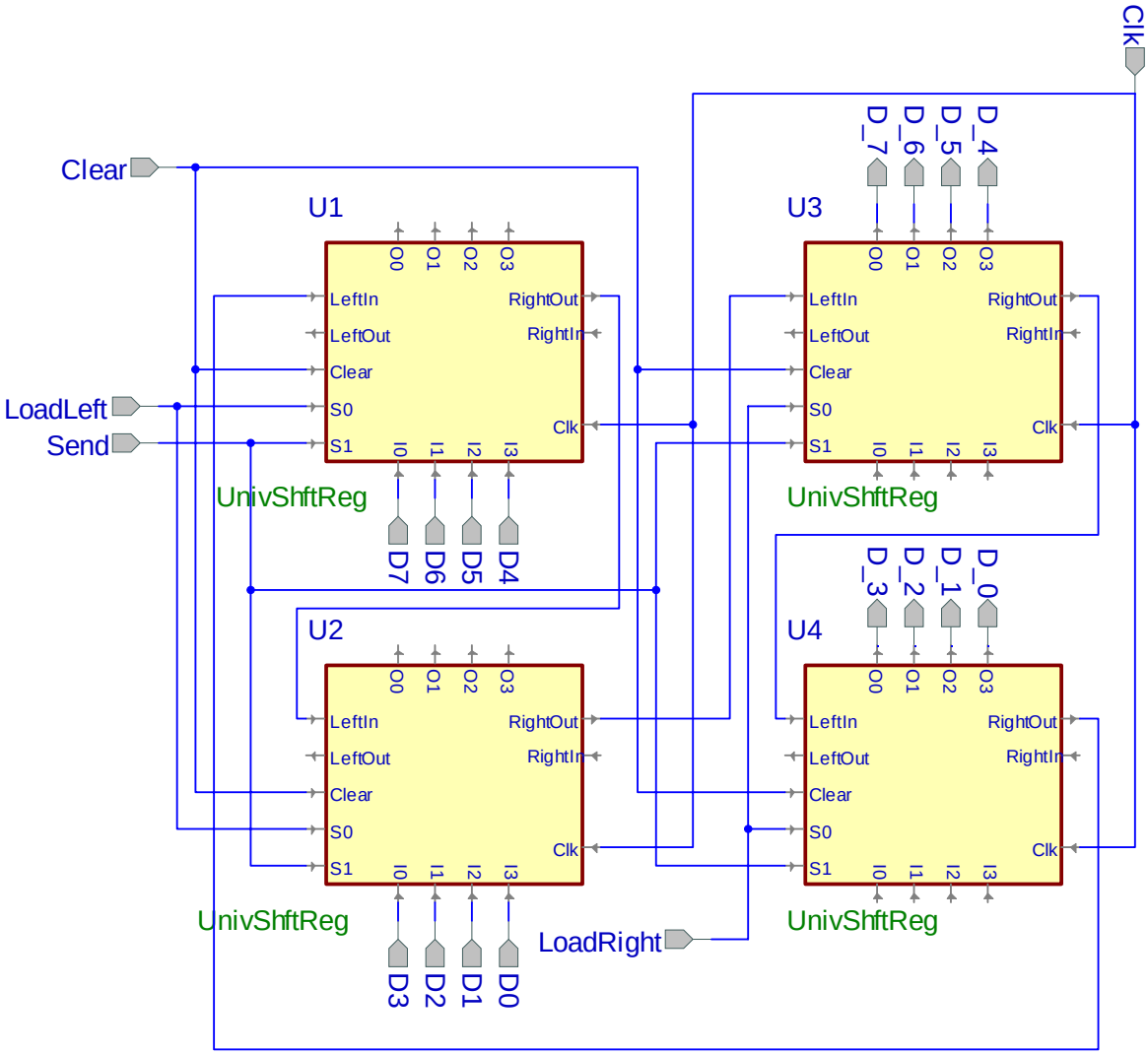
One of many possible implementations:



Simulation waveforms for the specified sequence of operations:



Exercise 6.29



Exercise 6.30

A master-slave flip-flop looks like the following when expressed in Verilog. Note the two `always` blocks, triggered by different edges of the clock. This description is reset-dominant.

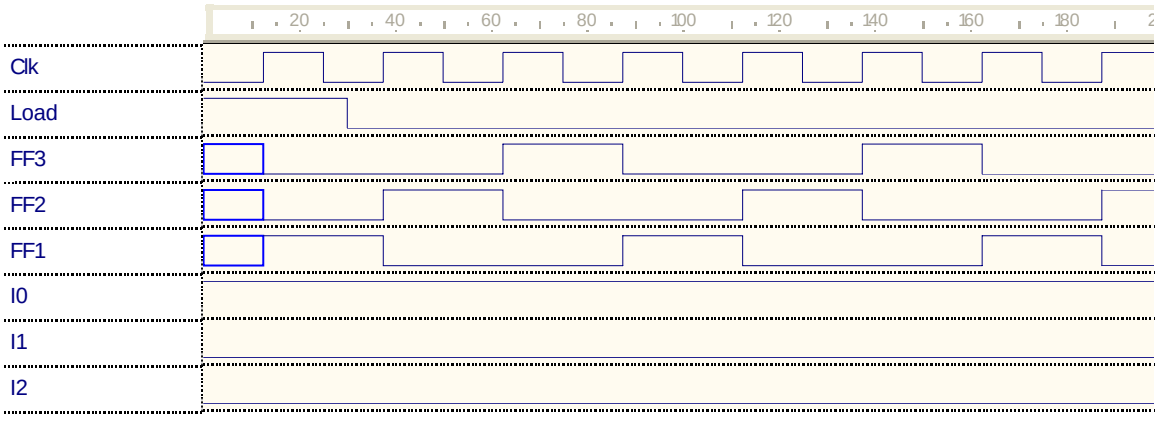
```
module prob6_30 (P, Q, Clk, R, S);

input R, S, Clk;
output P, Q;
reg P, Q;

always @(posedge Clk) begin
      if ( R ) P = 1'b0;
      else if ( S ) P = 1'b1;
end

always @(negedge Clk) begin
      if ( P ) Q = 1'b1;
      else Q = 1'b0;
end

endmodule
```

Exercise 6.31

The implementation below uses a Load signal to initialize FF_1 with I0, FF_2 with I1, and FF_3 with I2. If the load signal is not present, then the module performs a circular shift behavior.

```
module prob6_31 (Load, I0, I1, I2, Clk, FF1, FF2, FF3);

input Load, I0, I1, I2, Clk;
output FF1, FF2, FF3;
reg FF1, FF2, FF3;

always @(posedge Clk) begin
      if (Load) begin
            FF1 = I0;
            FF2 = I1;
            FF3 = I2;
      end
      else begin
            FF3 <= FF2;
            FF2 <= FF1;
            FF1 <= FF3;
      end
end

endmodule
```

The waveform on the next page demonstrates the functionality of this module.



Exercise 6.32

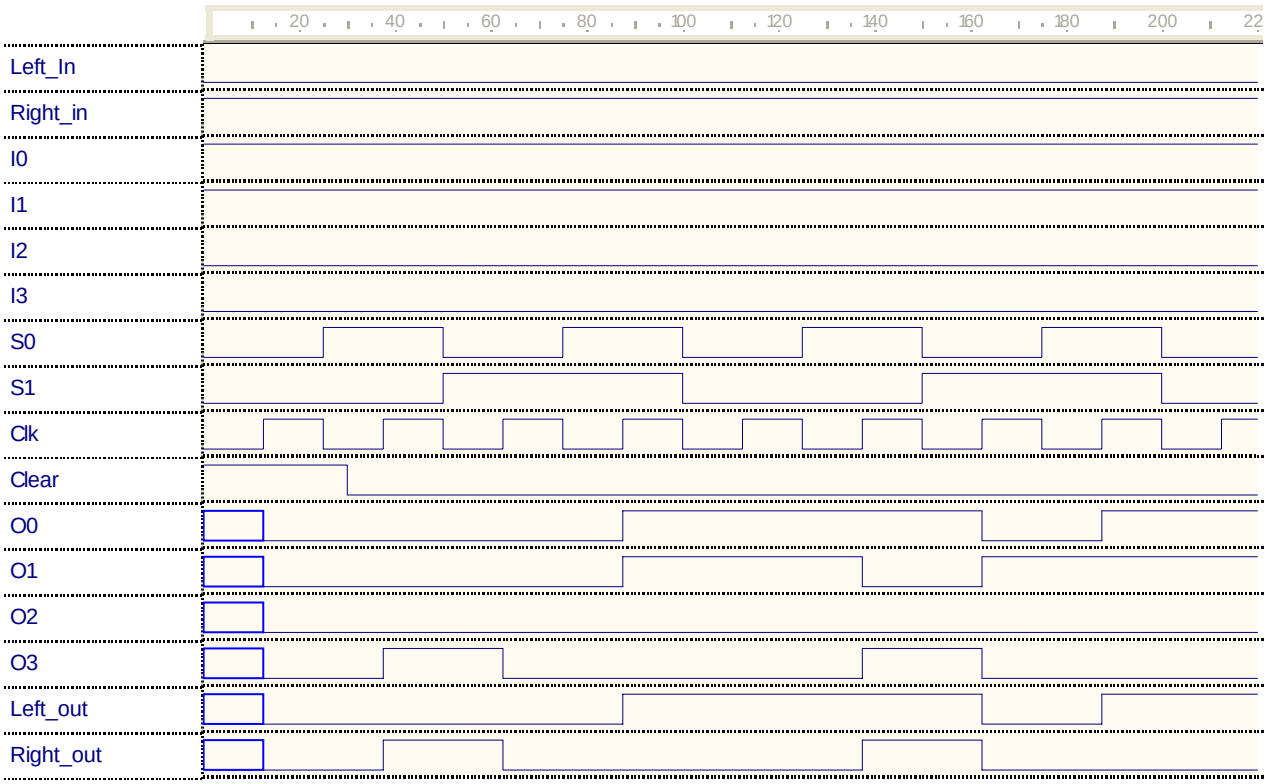
Here is the implementation for the 1 bit module that is shown in Figure 6.54.

```
module prob6_32_1bit (In, Left, Right, Clear, S0, S1, Clk, Out);

input In, Left, Right, Clear, S0, S1, Clk;
output Out;
reg Out;

always @(posedge Clk) begin
      if ( Clear ) Out = 1'b0;
      else begin
            if ( !S0 && !S1 ) Out = Out;
            if ( !S0 && S1 ) Out = Left;
            if ( S0 && !S1 ) Out = Right;
            if ( S0 && S1 ) Out = In;

      end
end

endmodule
```

The 4 bit implementation is shown on the next page using 4 instances of the `prob6_32_1bit` module and appropriate wiring done by name:

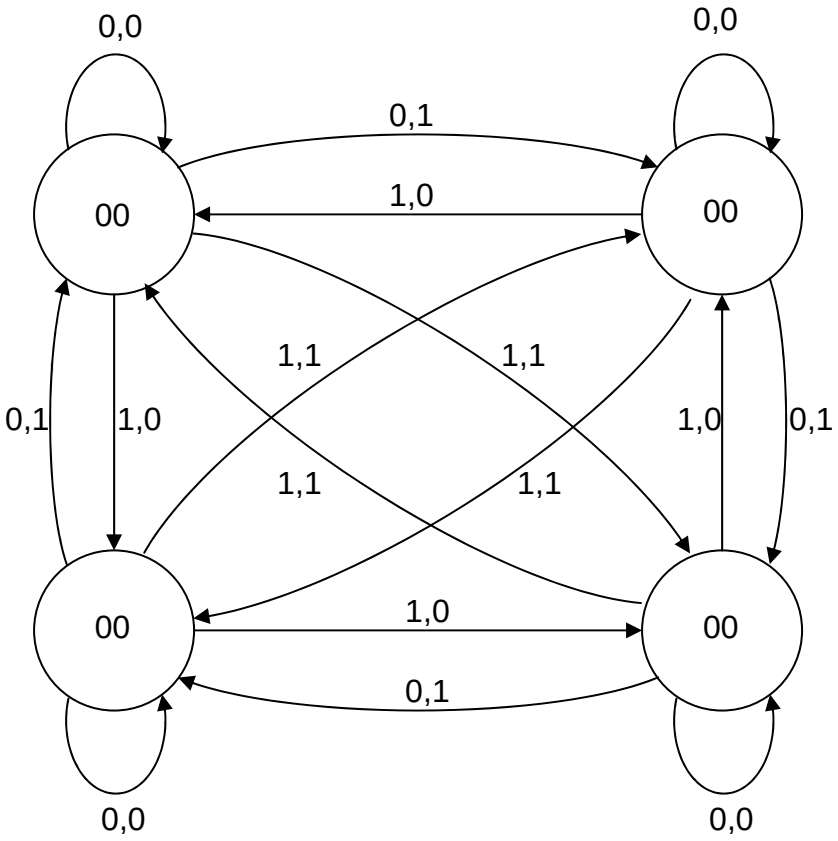
```
module prob6_32_4bit (I0, I1, I2, I3, Left_in, Right_in, Clear, S0, S1,
Clk, O0, O1, O2, O3, Left_out, Right_out);

input I0, I1, I2, I3, Left_in, Right_in, Clear, S0, S1, Clk;
output O0, O1, O2, O3, Left_out, Right_out;

prob6_32_1bit FirstBit( .In(I0), .Left(Left_In), .Right(O1), .Clk(Clk),
.Clear(Clear), .S0(S0), .S1(S1), .Out(O0));
prob6_32_1bit SecondBit( .In(I1), .Left(O0), .Right(O2), .Clk(Clk),
.Clear(Clear), .S0(S0), .S1(S1), .Out(O1));
prob6_32_1bit ThirdBit( .In(I2), .Left(O1), .Right(O3), .Clk(Clk),
.Clear(Clear), .S0(S0), .S1(S1), .Out(O2));
prob6_32_1bit FourthBit( .In(I3), .Left(O2), .Right(Right_in),
.Clk(Clk), .Clear(Clear), .S0(S0), .S1(S1), .Out(O3));

assign Right_out = O3;
assign Left_out = O0;

endmodule
```

Here is a timing waveform showing that the operations work correctly:



Exercise 7.1

(a) State diagram and transition table:

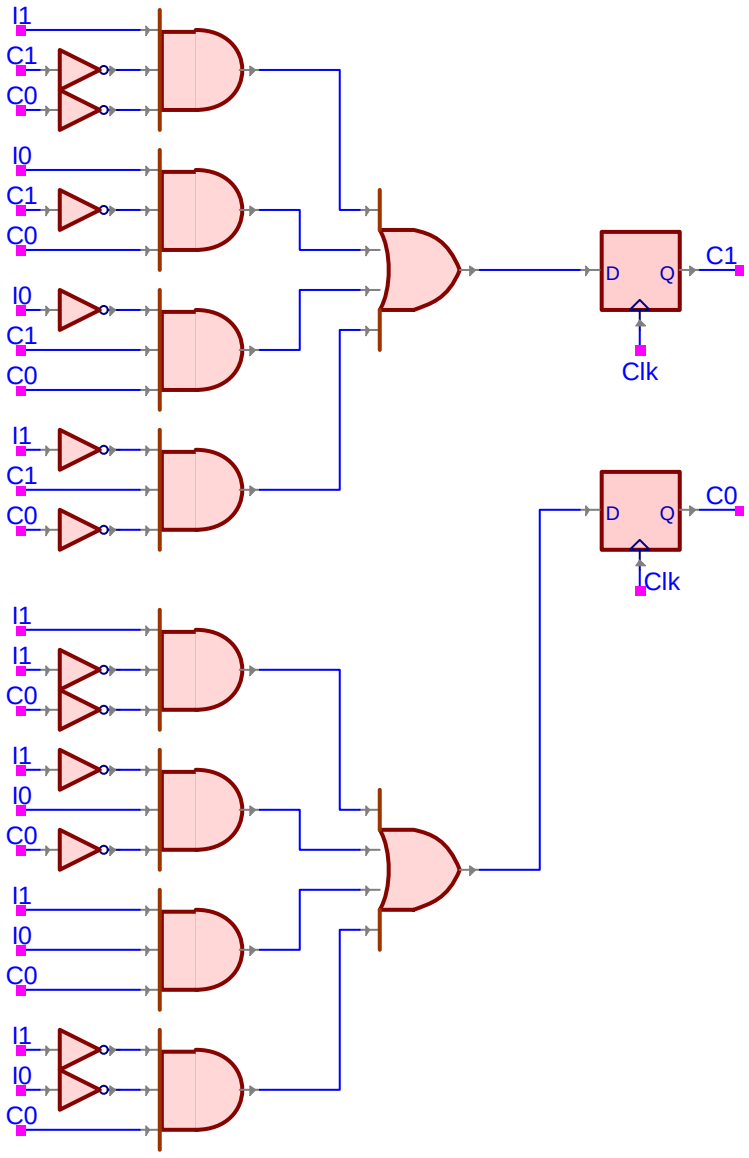


I_1	I_0	C_1	C_0	C_1+	C_0+
0	0	0	0	0	0
		0	1	0	1
		1	0	1	0
		1	1	1	1
0	1	0	0	0	1
		0	1	1	0
		1	0	1	1
		1	1	0	0
1	0	0	0	1	1
		0	1	0	0
		1	0	0	1
		1	1	1	0
1	1	0	0	1	0
		0	1	1	1
		1	0	0	0
		1	1	0	1

(b) Circuit schematic:

$C_1 = I_1C_1'C_0' + I_0C_1'C_0 + I_0'C_1C_0 + I_1'C_1C_0'$

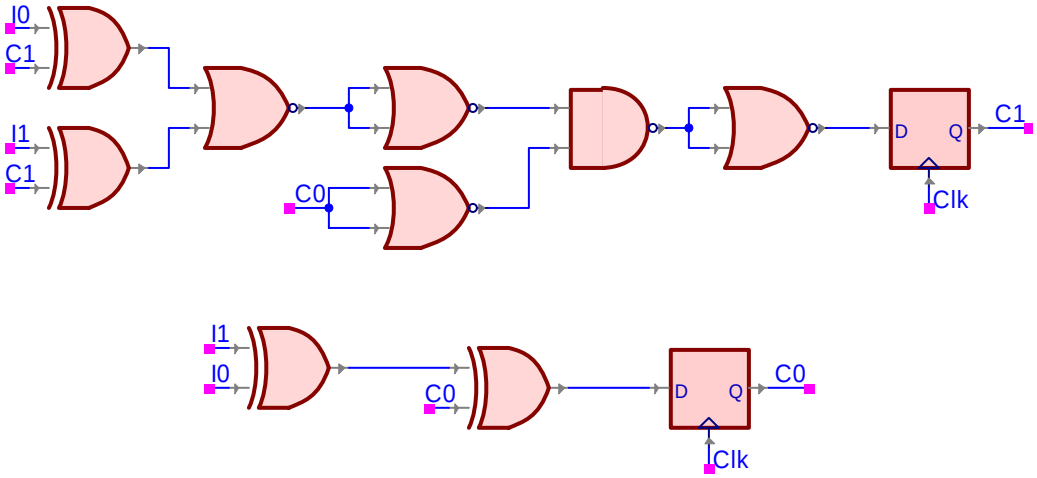
$C_0 = I_1I_0'C_0' + I_1'I_0C_0' + I_1I_0C_0 + I_1'I_0'C_0$



(c) NAND/NOR/XOR implementation:

$C_1 = [(I_0 \text{ xor } C_1) + (I_1 \text{ xor } C_1)]C_0'$

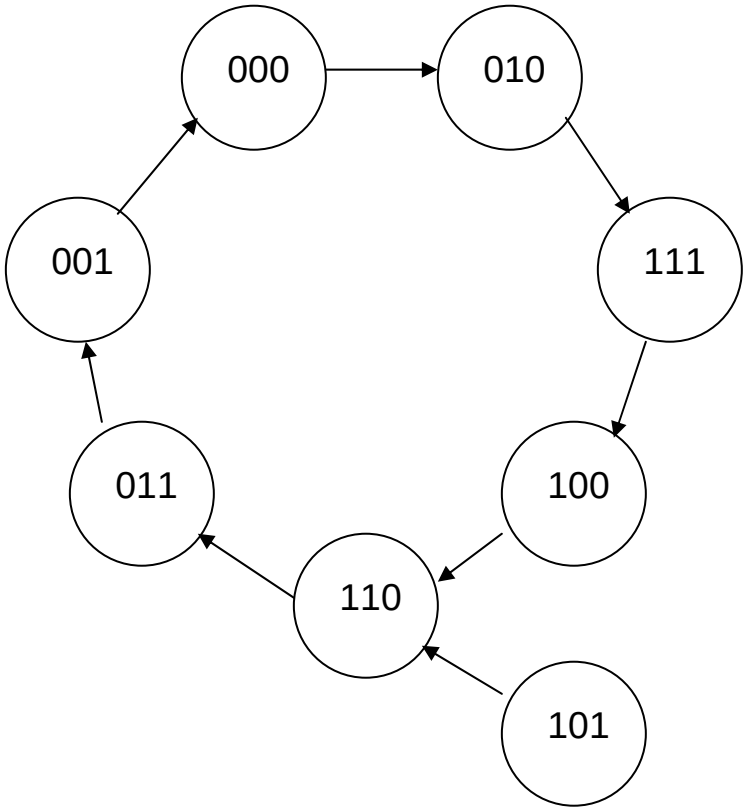
$C_0 = (I_1 \text{ xor } I_0) \text{ xor } C_0$



Exercise 7.2

We begin designing the three bit counter by first drawing the state transition graph, then the state transition table. Since state 101 is not part of the sequence, we need to make sure that it has a transition out to a state that is part of the sequence.

State diagram:



State transition table:

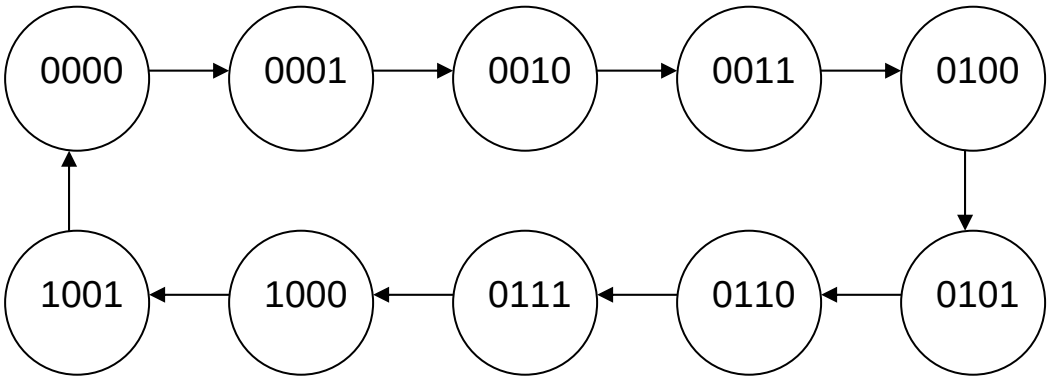
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	1	1	0
1	1	0	0	1	1
1	1	1	1	0	0

Exercise 7.3

(a) 3-bit counter:

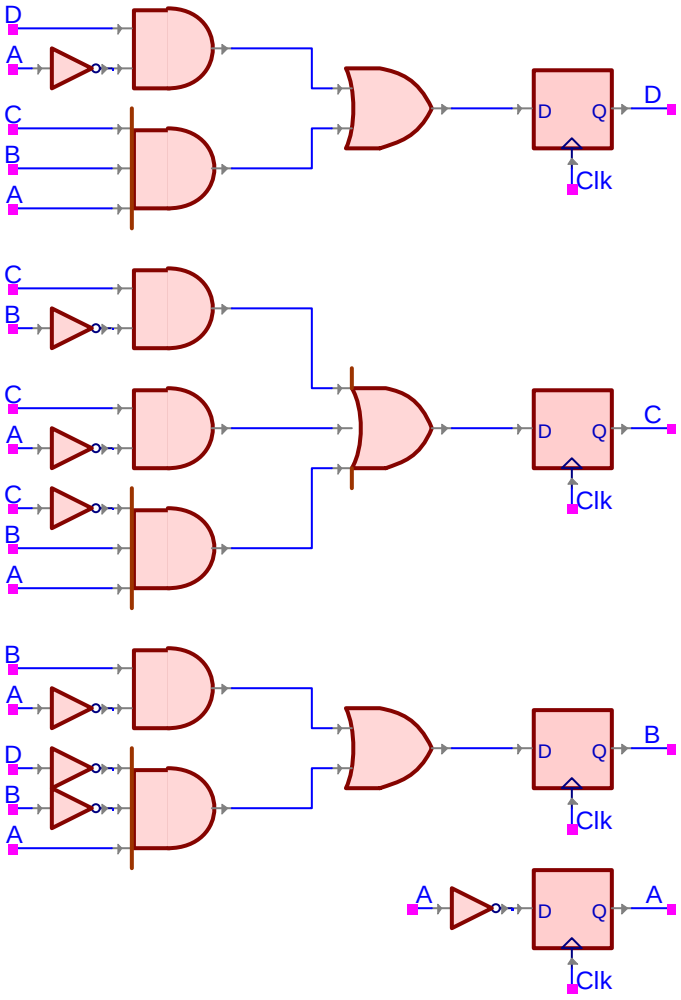
D	C	B	A	D+	C+	B+	A+
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0

State transition table



State diagram

(b) Circuit schematic:



(c) Expressions for a self-starting counter:

D+ = DA’ + DCB’ + DC’B + D’CBA

C+ = CB’ + CA’ + C’BA

B+ = BA’ D’B’A + CB’A

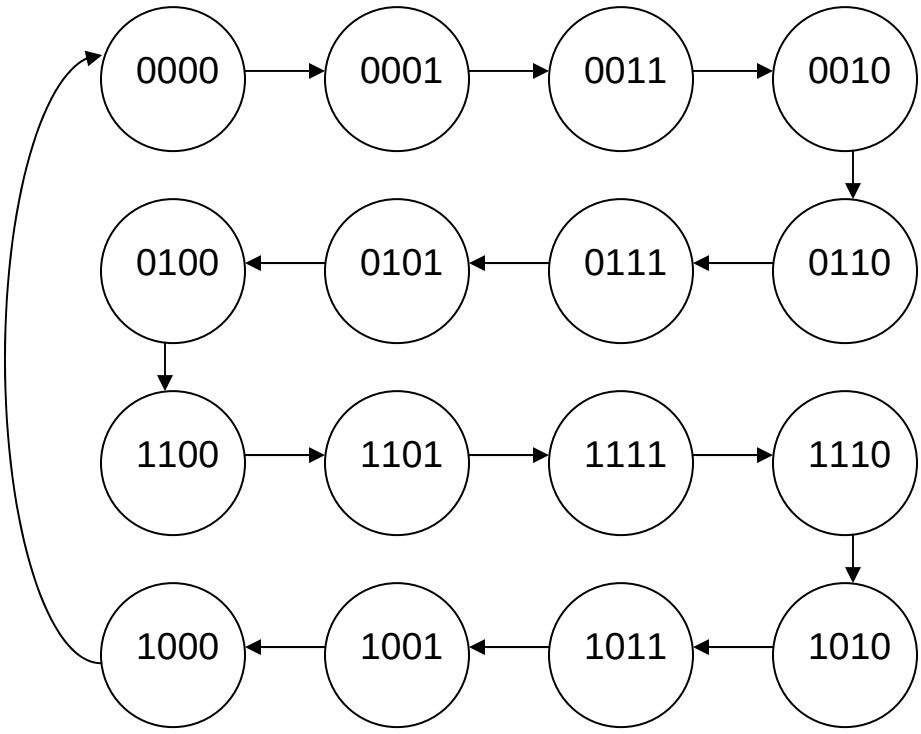
A+ = A’

Exercise 7.4

(a) Gray-code counter:

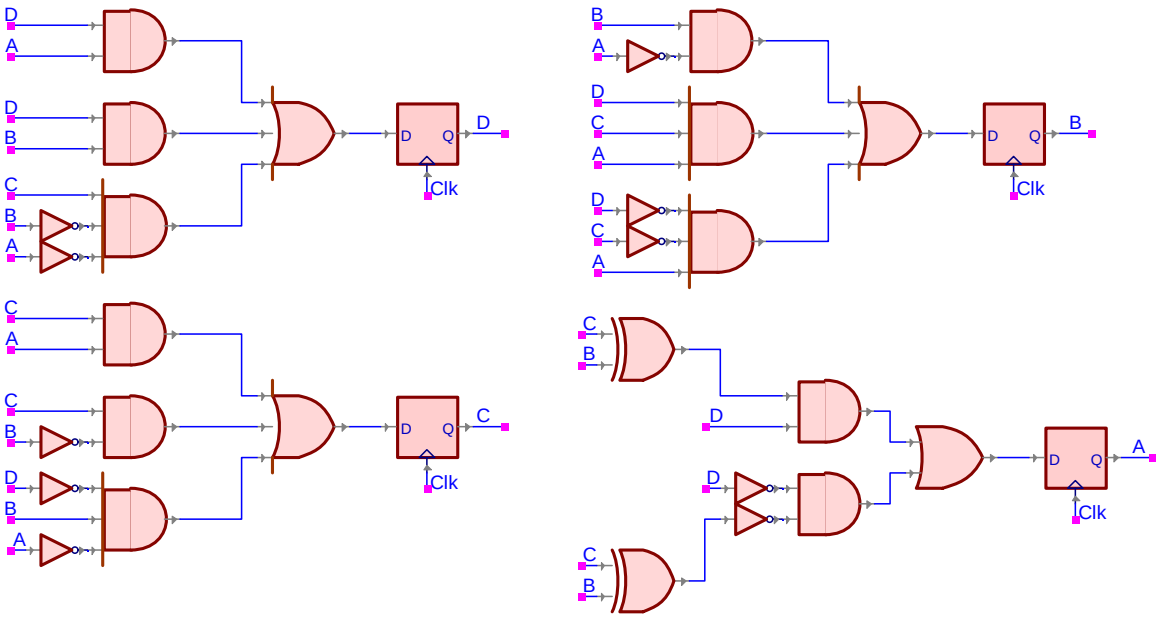
D	C	B	A	D+	C+	B+	A+
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	1
0	0	1	0	0	1	1	0
0	0	1	1	0	0	1	0
0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	1	1
0	1	1	1	0	1	0	1
1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	1	1
1	0	1	1	1	0	0	1
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	1
1	1	1	0	1	0	1	0
1	1	1	1	1	1	1	0

State-transition table



State Diagram

(b) Circuit Schematic:



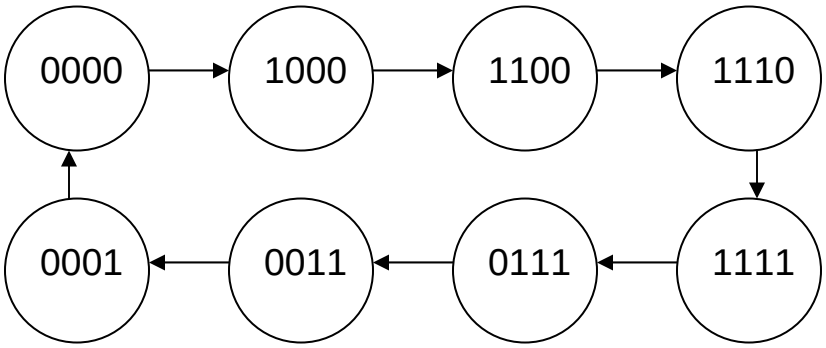
(c) We don't need to worry about the self-starting because every possible state is part of the sequence that the Gray-code counter goes through. So no matter what state the counter starts out in, it will always count correctly.

Exercise 7.5

4-bit Johnson Counter:

D	C	B	A	D+	C+	B+	A+
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	-	-	-	-
0	0	1	1	0	0	0	1
0	1	0	0	-	-	-	-
0	1	0	1	-	-	-	-
0	1	1	0	-	-	-	-
0	1	1	1	0	0	1	1
1	0	0	0	1	1	0	0
1	0	0	1	-	-	-	-
1	0	1	0	-	-	-	-
1	0	1	1	-	-	-	-
1	1	0	0	1	1	1	0
1	1	0	1	-	-	-	-
1	1	1	0	1	1	1	1
1	1	1	1	0	1	1	1

State Transition Table



State Diagram

If we assume that the states that are not part of the sequence will never be entered, we can use the outputs for those states as don't-cares, which will make our implementation less complex. The following are the next-state expressions:

D+ = CA' + BA'

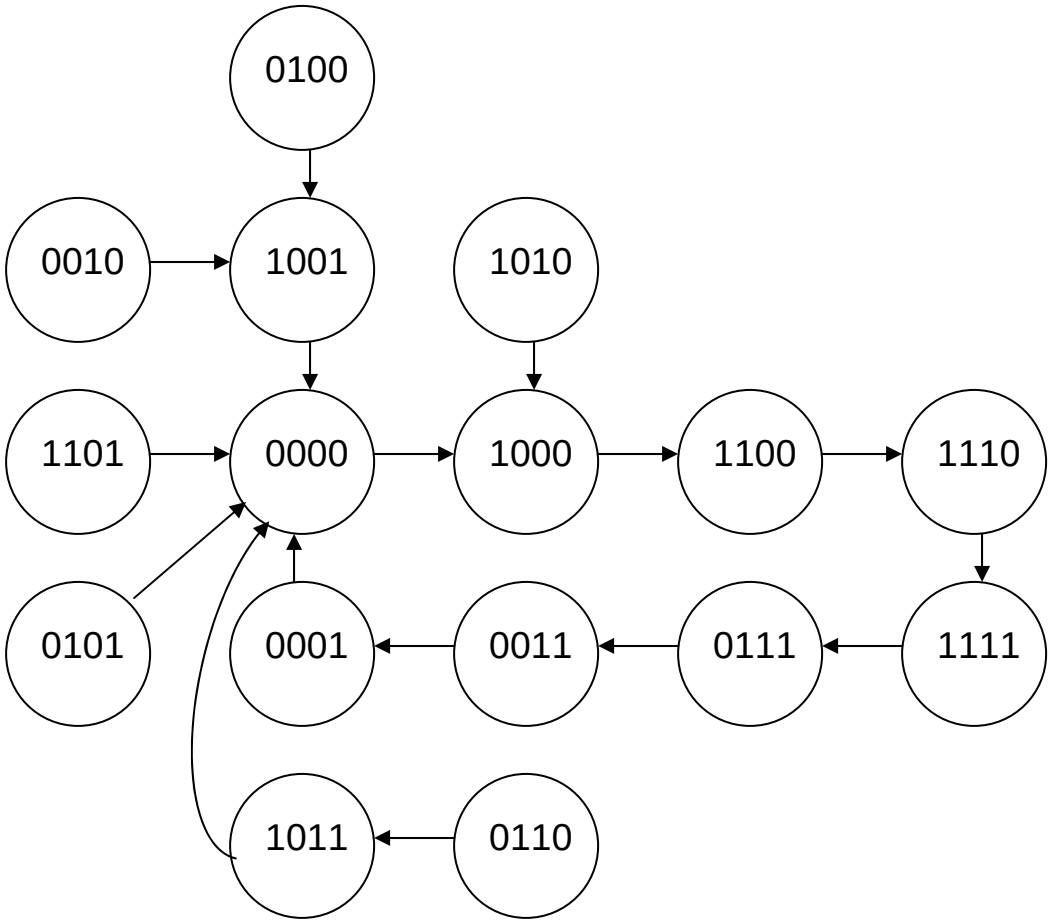
C+ = DB'A' + DCB

B+ = CB + CA'

A+ = D'B + CB

Exercise 7.6

The solution for Exercise 7.5 is self-starting. All of the states eventually lead into the counting sequence. The following state diagram illustrates the transitions from each of the 16 possible states:



State diagram for the 4-bit Johnson counter

Exercise 7.7

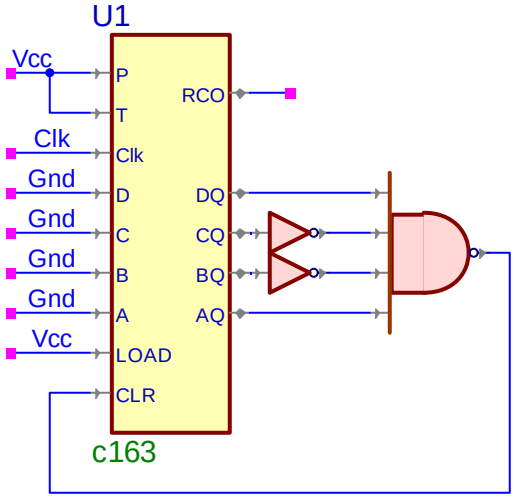
An asynchronous load or reset signal may be acceptable when we need to immediately change the contents of a register without waiting for an extra clock cycle. This could be important in communication between two FSMs or counters. Of course, we would have to make sure that asynchronous behaviors in two communicating FSMs were not inadvertently connected to create an asynchronous feedback cycle (that is, a loop not interrupted by waiting for a clock signal).

Exercise 7.8

The Load signal on the counter-chip tells the counter to load the value that is on the parallel load inputs. We can see from the timing diagram that when the counter reaches 1111_2, the compliment of the RCO signal causes Load to be driven low and the counter circles back to the value at the parallel load inputs. If we take the RCO compliment and AND it with an active-low reset signal, we will be able to force the counter to start counting in its first valid state whenever the reset is driven low. Since we are using an AND gate with the reset and RCO signals, the functionality of the counter will be exactly the same but we will be able to start out in the first valid count sequence whenever we wish. One assumption that we must make about the reset signal is that it is asserted for at least one clock cycle. If it doesn't, the counter may behave unpredictably.

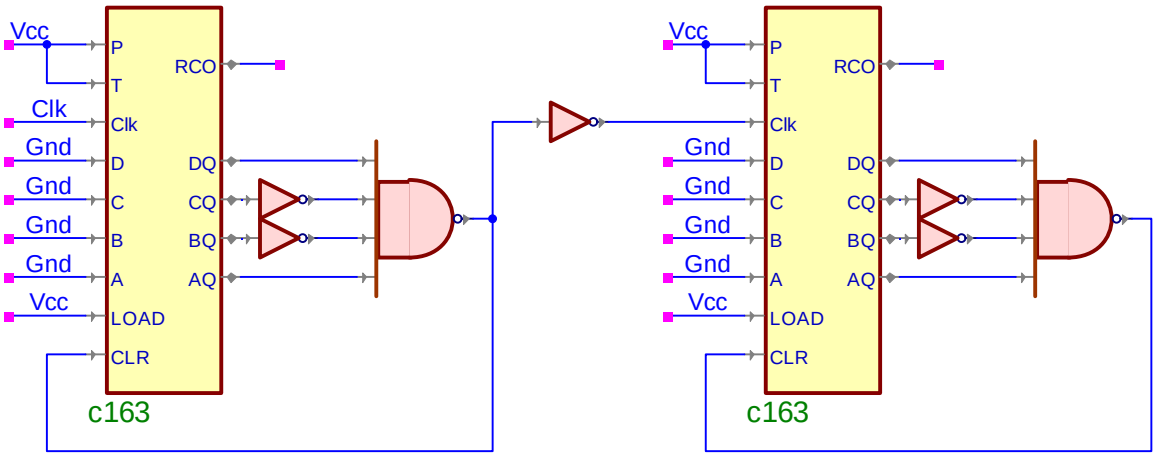
Exercise 7.9

To build a BCD counter out of a 163 counter component, we need to have the counter roll back to zero when 1001_2 is reached. We can accomplish this by checking the value of the parallel outputs and when it reaches 1001_2, drive the CLR input low, which will cause the next output to be zero. The following circuit diagram illustrates this:



1-Digit BCD Counter

We can add another bit by using two 1-digit BCD counters. The second counter will be for the most significant bit. This counter should only be incremented every time the least significant counter rolls over. We can achieve this functionality by using the same signal that we use to roll over the low order counter as a clock signal for the high order counter. We have to invert the output of it so that when the low order counter rolls over, the signal will go high for one clock cycle causing the high order counter to increment itself. This can be generalized to an n-digit BCD counter by using n BCD counters and using the CLR output from the n-1 counter as the n counter Clk. The following circuit schematic illustrates this with a 2-digit BCD counter:

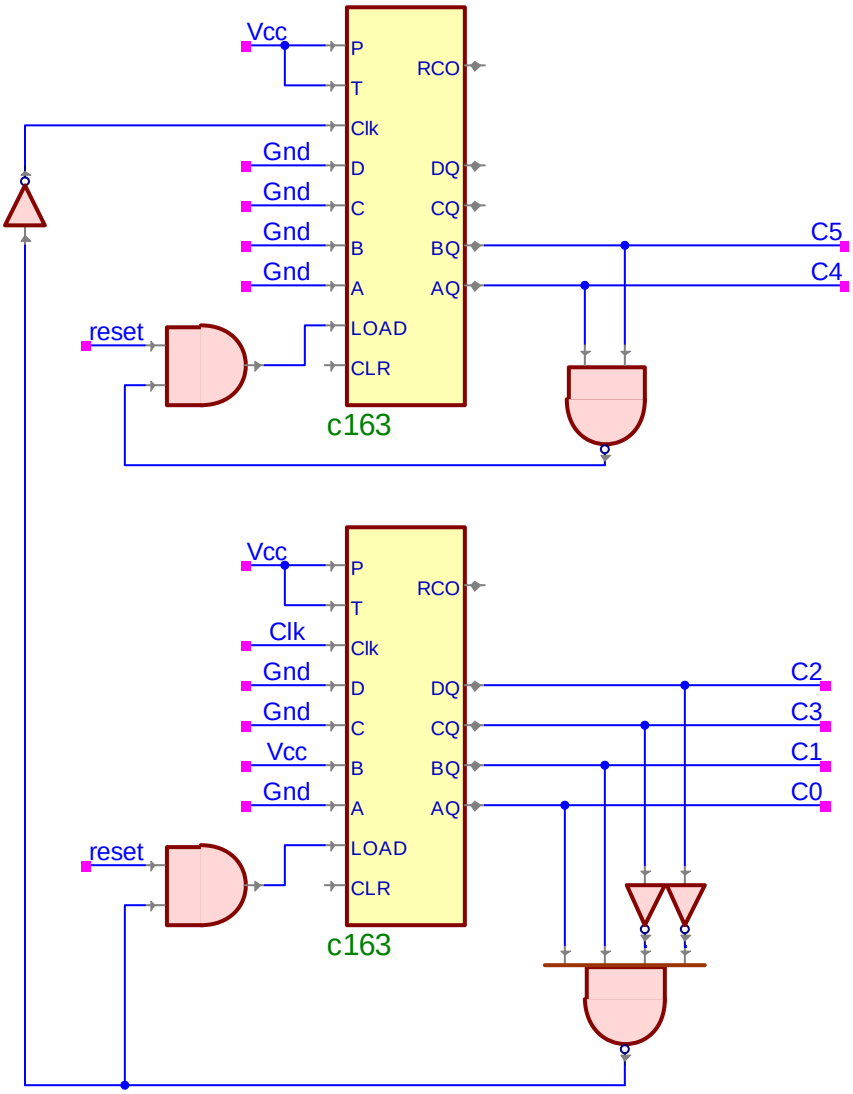


Low order BCD counter

High order BCD counter

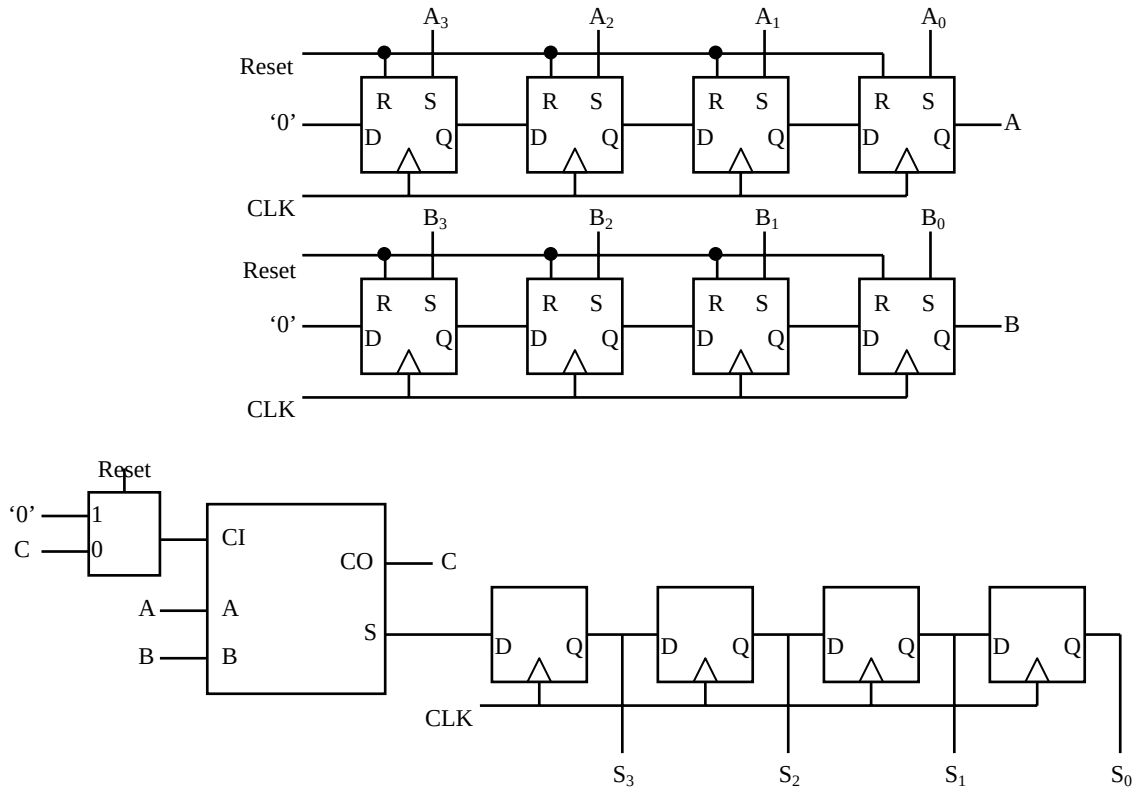
Exercise 7.10

So, essentially what we have is two cutoff/offset counters that are set to roll over at their corresponding correct values. When they roll over, they load the starting value from the parallel inputs and continue counting. C5-C0 are the bit values for the 6-bit output. The reset is also tied to the load, which will cause the counters to reset to their start value. The following circuit schematic illustrates a 6-bit BCD counter:



Exercise 7.11

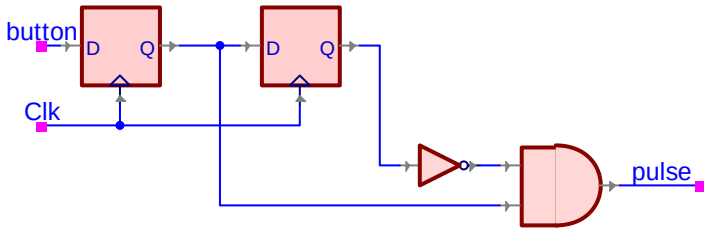
(a) Circuit schematic for the 4-bit serial adder:



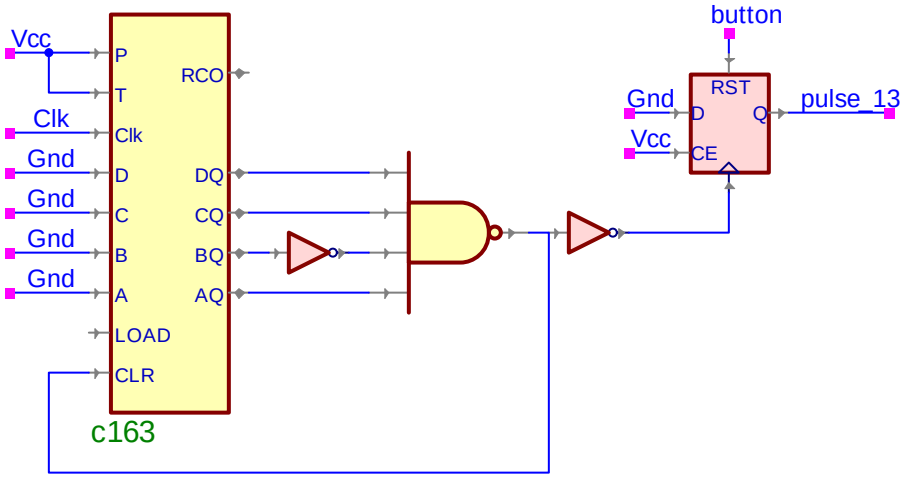
(b) The only control signal in this implementation is Reset. When the Reset signal is asserted, each bit of the two operands is loaded into the registers in parallel. When the reset signal goes low, each value is shifted to the right on every clock cycle. The Reset signal also acts as the control for a 2:1 multiplexor, which loads the first carry-in with a zero. The carry-out feeds back into the carry-in, which causes it to be used for the subsequent add. After the entire 4-bit value has been shifted through the adder, the result can be accessed on the output of each sum register.

Exercise 7.12

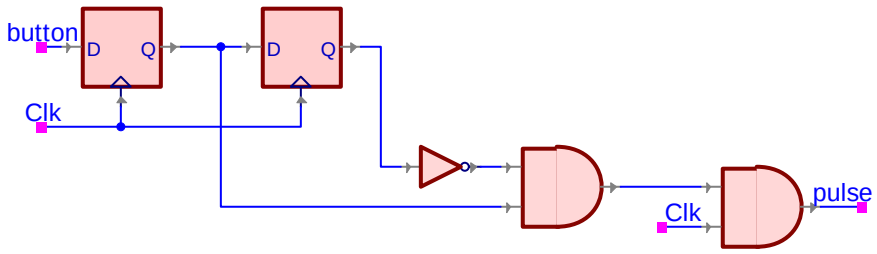
(a) The way to guarantee that the clock signal only goes high for one clock cycle is to only assert it when the input signal is changing from low to high. We can do this by using a 2-bit shift register and checking the value of each bit. When the second one is low and the first one is high, we assert the output signal.



(b) The simplest way to implement this design is to use a register which loads a one into it when it is reset. The button signal is used as the reset for this register. The clock input to this register is tied to the output signals from the 163 counter in a manner that will cause the clock to cycle once on the 13th cycle and load a zero.

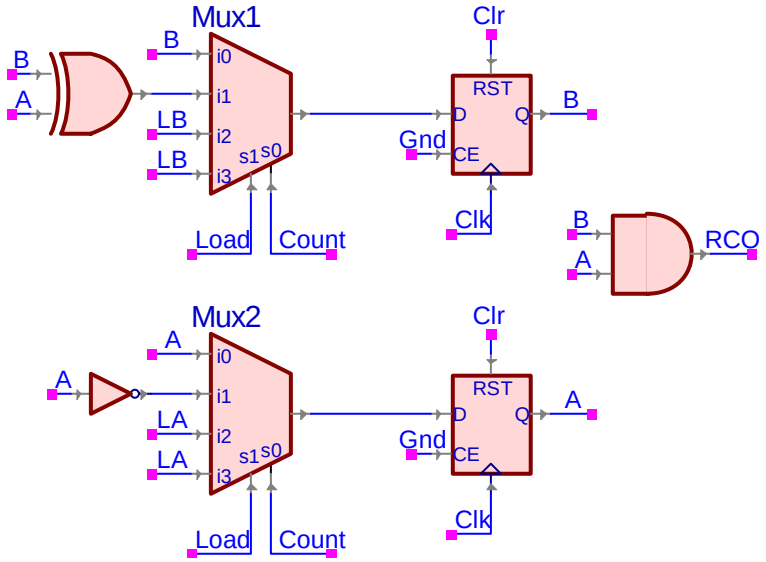


(c) A system that will provide a single clock step can be implemented with an extension of the design in part a. The portion of the design from part a guarantees that the output will be exactly one clock cycle. ANDing this output with the clock signal will cause it to cycle from low to high and back down to low again.



Exercise 7.13

The first piece that we need to design is the next state logic. 4:1 multiplexors work well to implement this functionality. We will use load and count as the control signals for these multiplexors. When load is zero, we are either counting or not counting. In the case that we are not counting, the multiplexor just needs to feed back the current value that the flip-flop has in it. If we are counting, the multiplexor for the B flip-flop is simply the sum of the A and B flip-flops, which can be implemented with an 2-input XOR. The mutliplexor for the A flip-flop is just the current output inverted when we are counting. If the load signal is high, the input to both multiplexors is simply their respective load values. To achieve the clear functionality, we tie the CLR signal to the reset input of the flip-flops and tie the CE input to the flip-flops to Gnd. This will reset both of the flip-flops to zero when the CLR signal is asserted and it will have precedence over both the count and the load signals. Lastly, the RCO signal is just the outputs of the A and B flip-flops ANDed together. The resulting circuit schematic looks as follows:



2-bit up-counter

Exercise 7.14

The biggest problem associated with using both positive and negative edge-triggered flip-flops is that they latch their inputs at different times and the output for each one is not valid at the same time as the output of the other one. This can cause serious problems if the combinational logic which computes the next state of one flip-flop relies on the output of the other flip-flop. For example, if we have flip-flop A and B, each of which is triggered on different clock edges, the combinational logic of A may be half way done computing the next state for A and the output of B changes. This would cause unpredictable behavior in the combinational logic for A.

Exercise 7.15

Using both flip-flops clocked on both clock edges changes timing constraints dramatically. Since we are assuming we are not keeping negative and positive edge-triggered flip-flops segregated, instead of having the single constraint:

$$T_{period} > T_{pd} + T_{comb} + T_{su}$$

We will now have two constraints, one for each of the two parts of the clock cycle, one for when the clock is high and one for when the clock is low:

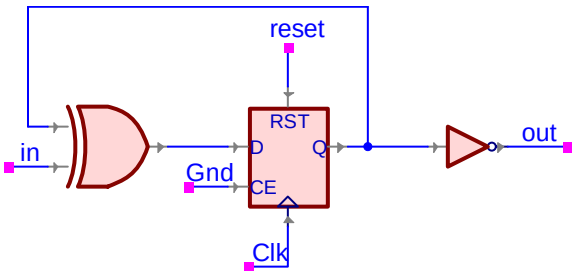
$$T_{high} > T_{pd} + T_{comb\text{-}high} + T_{su}$$
$$T_{low} > T_{pd} + T_{comb\text{-}low} + T_{su}$$

Therefore, the constraints are much tighter as T_{high} and T_{low} add up to T_{period}. If the clock's duty-cycle is 50%, then things can be made to be pretty similar to a single clock-edge system if the combinational logic delays can be evenly split between the two parts. If the clock duty-cycle is 90%, then one constraint will be very tight (10% of the clock period) while the other will be almost a full period (90% to be exact).

There is rarely an advantage to clocking a single system with both clock edges as it makes it more difficult for the designer to decide where to place combinational logic.

Exercise 7.16

Since we already have an odd parity checker, designing an even parity checker is trivial. An even parity checker is just the inverse of an odd one. So we can simply add an inverter to the output of the odd parity checker. The resulting circuit schematic looks as follows:

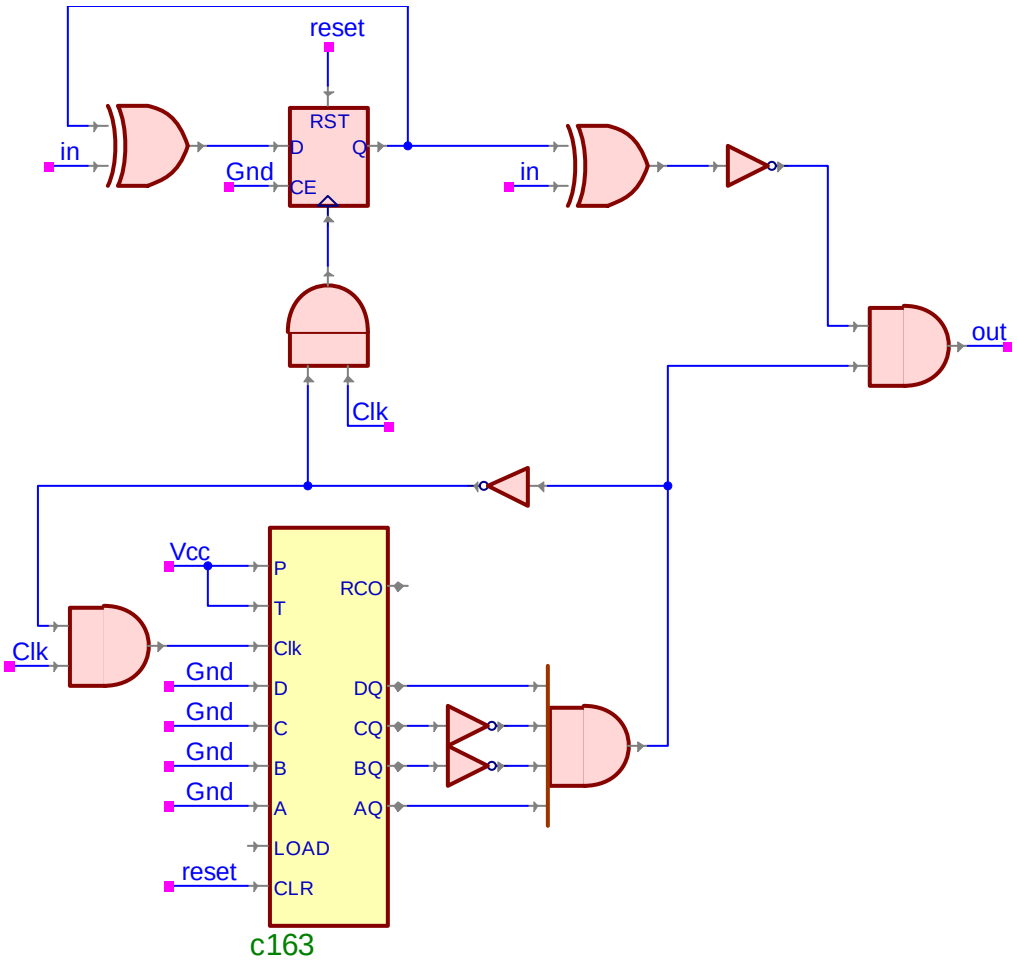


Even parity checker

Exercise 7.17

(a) It is possible to designe a state diagram with a small number of states to implement this 9th bit parity checker system. The system doesn't need to track all possible combinations of 9 bits. If it did, the system would have 2^{9th} or 512 states. In reality, the state diagram only needs to have 9 states. The first 8 states would count how many bits the input has seen and update the parity checker for each one. The last state would compare the current parity checker value with the 9th bit and assert OK or ERROR.

(b) The way that this design works is when the system is reset, the counter clears its contents out to be zero. Each clock cycle the counter counts up one and the parity checker reads in a new value. When the counter reaches 9, the clock inputs to the counter and flip-flop are disabled, the current value in the flip-flop is XORed with the 9th bit of the input and inverted. If this value is a one, it means that the parity bit and 9th input bit are the same and the output is asserted.



9-bit serial parity checker

Exercise 7.18

Let L be the number of flip-flops, J be the number of inputs, and K be the number of outputs.

(a) The maximum number of states is: $2^L = 8$

The minimum number of states is: $2^L = 8$

(b) The maximum number of transitions starting in a particular state is: $2^J = 4$

The minimum number of transitions starting in a particular state is: $2^J = 4$

(c) The maximum number of transitions that can end in a particular state is: $2^J * 2^L = 32$

The minimum number of transitions that can end in a particular state is: 0

(d) The maximum number of binary patters that can be displayed on the outputs is:

$\text{Min}(2^K, 2^J * 2^L) = 32$

The maximum number of binary patters that can be displayed on the outputs is:

1

Exercise 7.19

Let L be the number of flip-flops, J be the number of inputs, and K be the number of outputs.

(a) The maximum number of states is: $2^L = 32$

The minimum number of states is: $2^L = 32$

(b) The maximum number of transitions starting in a particular state is: $2^J = 8$

The minimum number of transitions starting in a particular state is: $2^J = 8$

(c) The maximum number of transitions that can end in a particular state is: $2^J * 2^L = 256$

The minimum number of transitions that can end in a particular state is: 0

(d) The maximum number of binary patters that can be displayed on the outputs is:

$\text{Min}(2^K, 2^L) = 32$

The maximum number of binary patters that can be displayed on the outputs is:

1

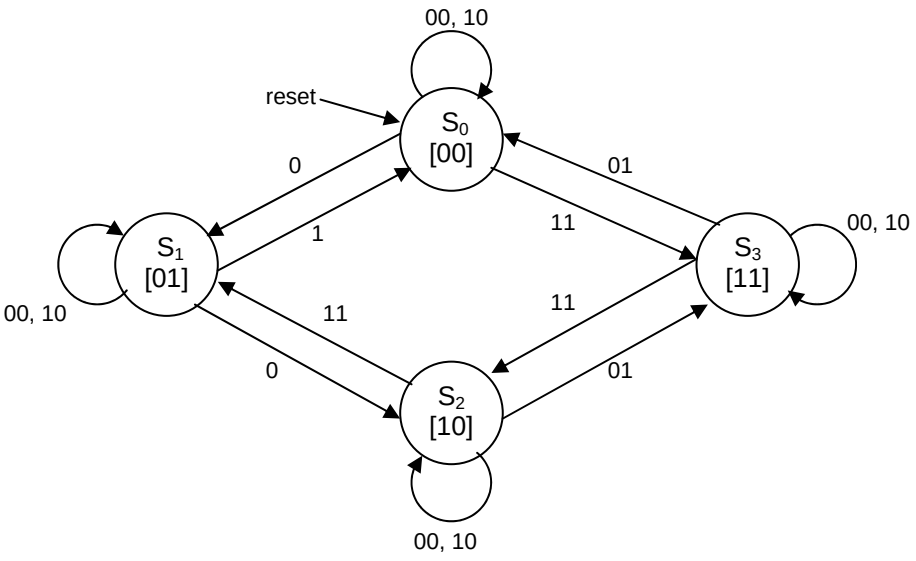
Exercise 7.20

The counter circuit in this exercise has 2 flip-flops, which tells us that the state diagram will have four states. We will assume that we don't care what the transitions are if the input C is 0. This leaves us with a total of eight entries in our truth table that we care about. To figure out these values we will trace through the circuit. Once we have the next state values, we can draw our state diagram.

D	C	S_1	S_0	S_1+	S_0+
0	0	0	0	-	-
		0	1	-	-
		1	0	-	-
		1	1	-	-
0	1	0	0	0	1
		0	1	1	0
		1	0	1	1
		1	1	0	0
1	0	0	0	-	-
		0	1	-	-
		1	0	-	-
		1	1	-	-
1	1	0	0	1	1
		0	1	0	0
		1	0	0	1
		1	1	1	0

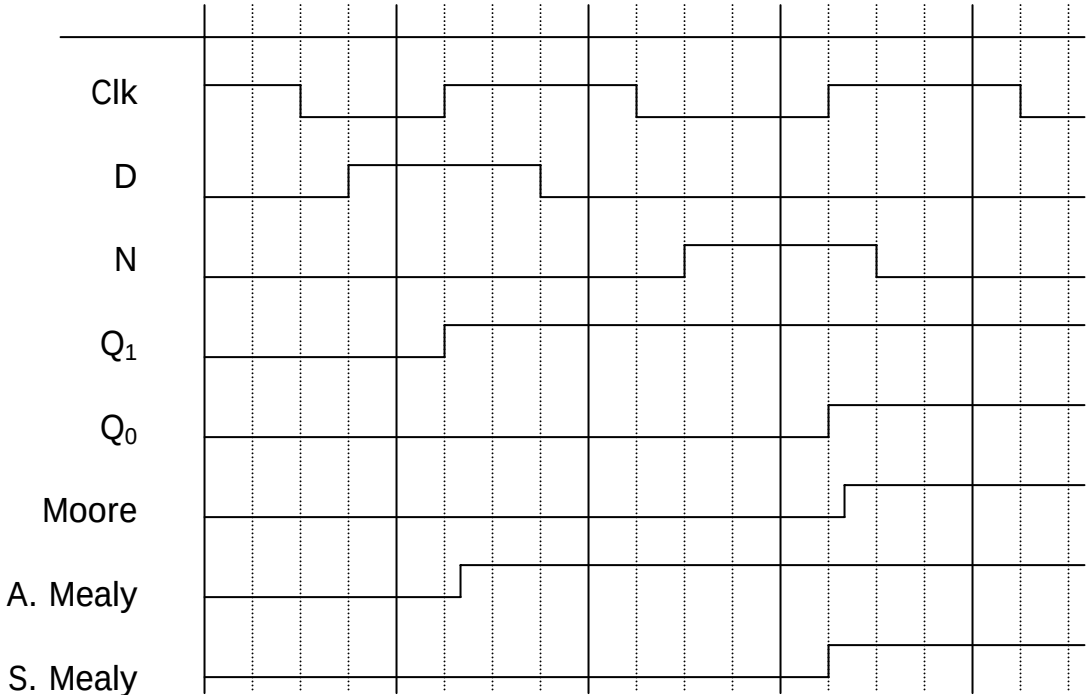
Truth table

As we can see from this truth table the forward sequence of the counter is: 00, 01, 10, 11, and the reverse sequence is: 00, 10, 01, 11. The transitions are labeled DC, where D is the count direction input and C is the count input. The state diagram for this design looks as follows:



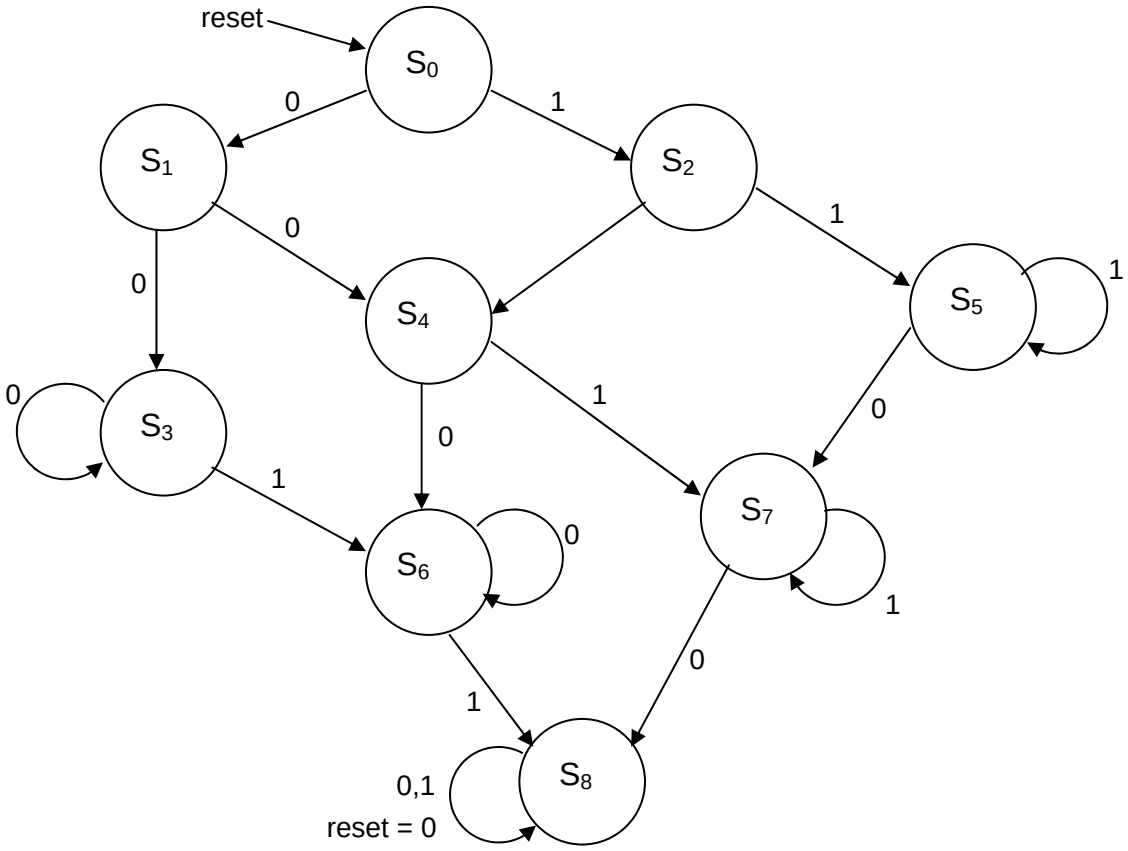
Exercise 7.21

A dime and a nickel are asynchronously inserted into the vending machine. At the first rising edge of the clock, Q_1 latches a 1. At the second rising edge of the clock, Q_0 latches a 1 from the nickel. One gate delay later the Moore machine's Open is asserted. The asynchronous Mealy machine's Open is asserted one gate delay after the nickel is inserted. The synchronous Mealy machine's Open is asserted when the Open register latches a 1. We can see that the synchronous Mealy machine is the only implementation that exhibits the behavior we are looking for. The timing diagram looks as follows:



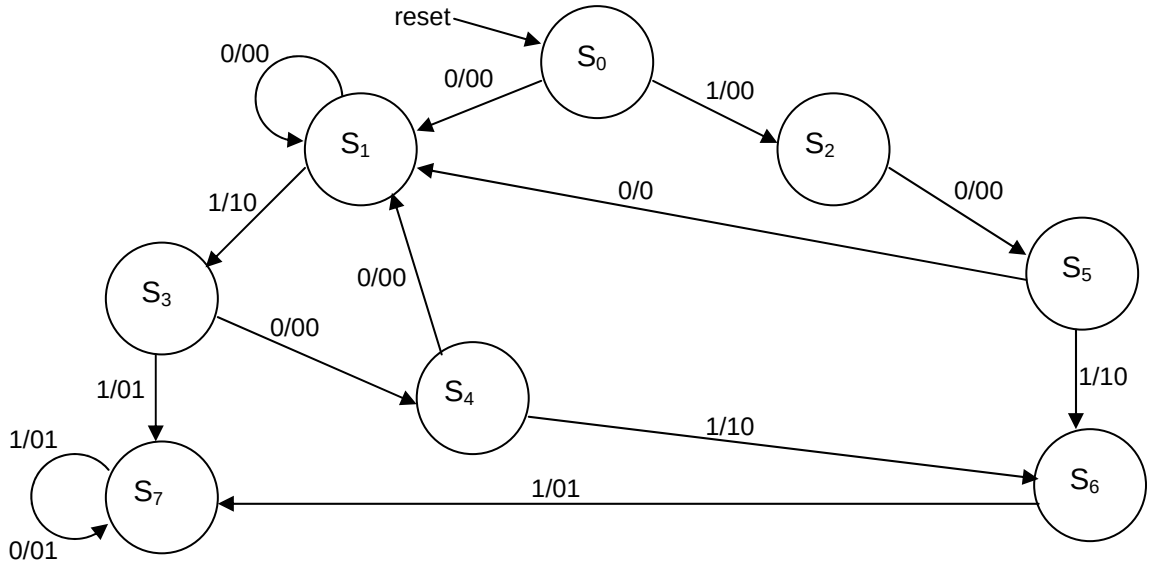
Exercise 7.22

The state machine needs to be designed so that we must see at least 2 zeros in a row and 2 ones in a row before the output is asserted. So the output is 0 in all states except the final state, S_8. The state diagram looks as follows:



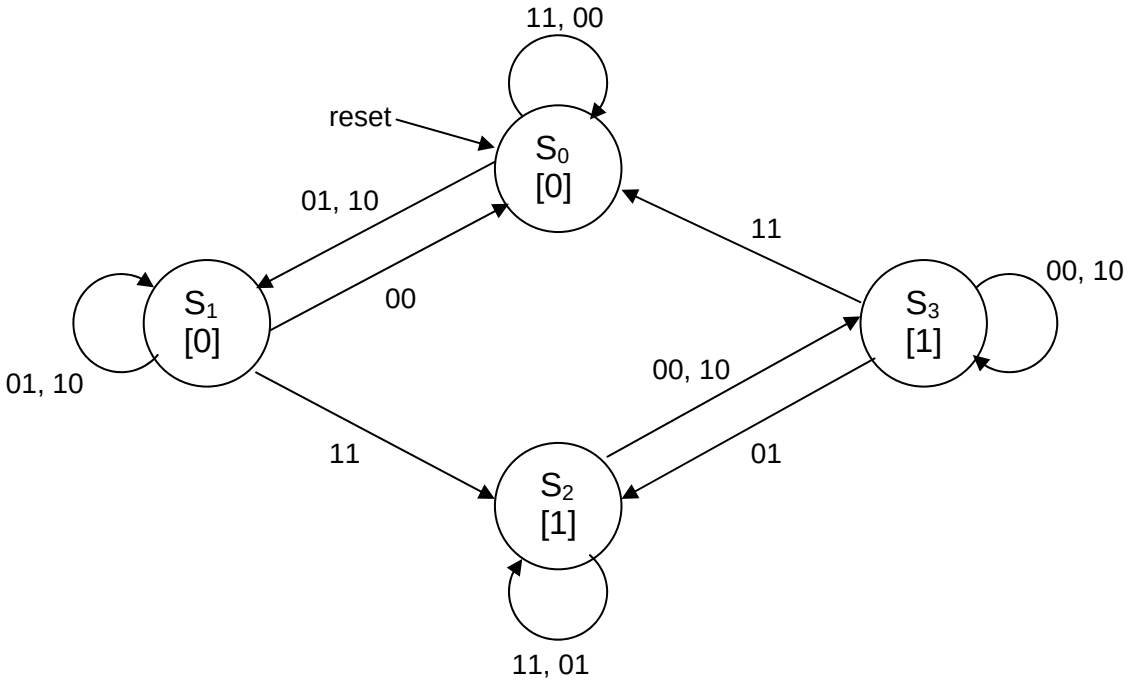
Exercise 7.23

This design can be implemented with 8 states because the system only needs to keep track of the last 3 input bits. The transitions are labeled X/Z_1Z_2, where X is the current input and Z_1 and Z_2 are the current outputs. State S_7 is the trap state that the system goes into if it sees the sequence 011. The resulting state diagram looks as follows:



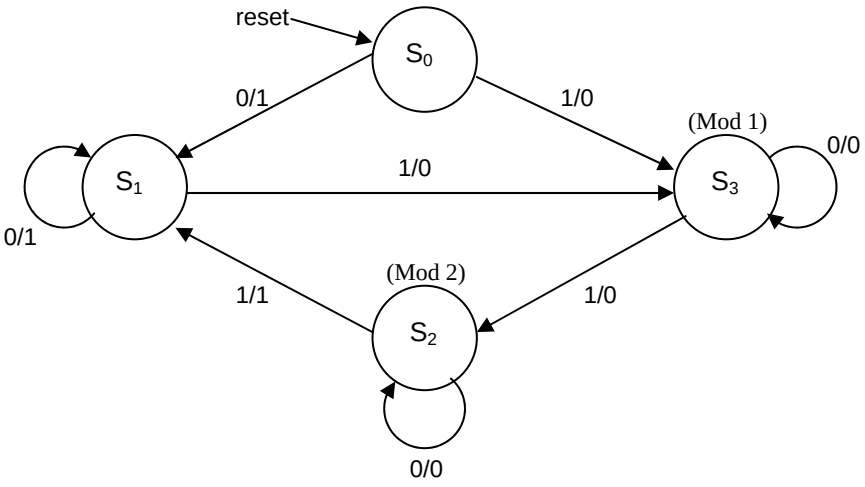
Exercise 7.24

Since this system only needs to remember 2 bits of the input sequence, the state diagram only needs four states. The input is shown next to each transition and the output is shown in brackets within each state. The resulting state diagram looks as follows:

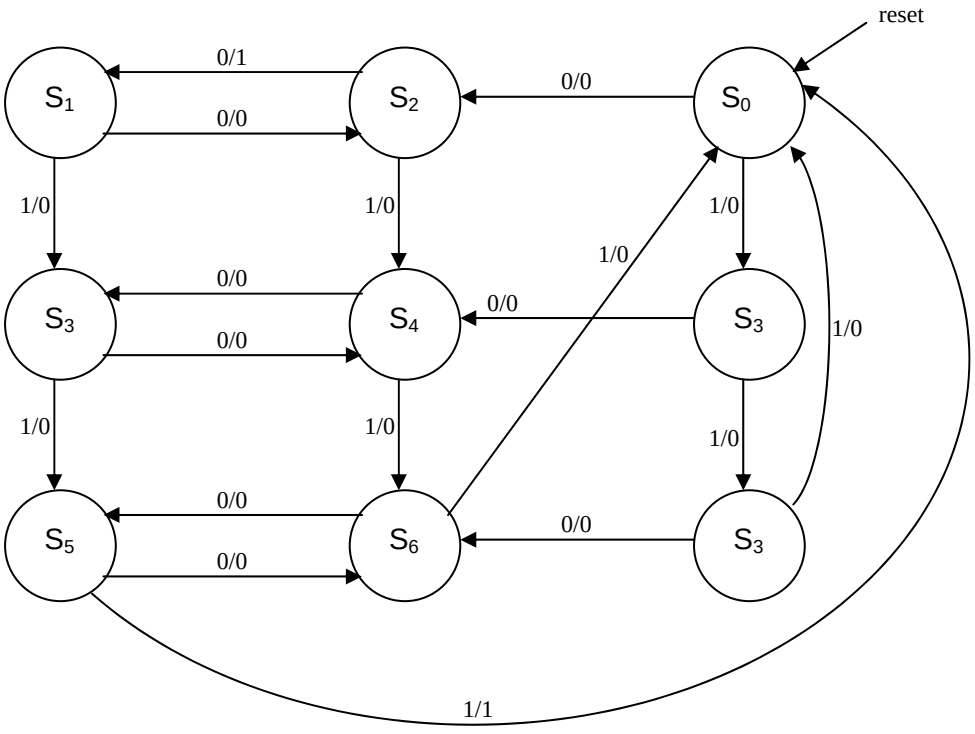


Exercise 7.25

(a) This system can be implemented with four states. Essentially, all it does is count how many ones have been received. State S_1 is the only state which has the output asserted. The transitions are labeled X/Z, where X is the input and Z is the output. The resulting state diagram looks as follows:

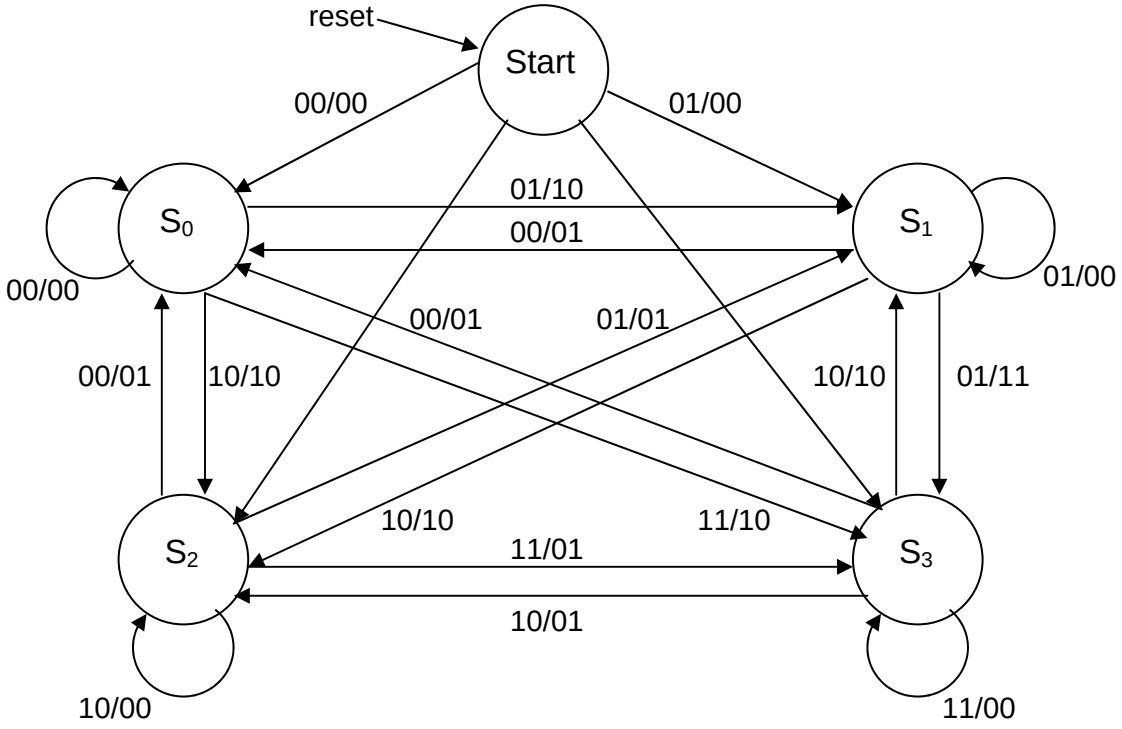


(b) This system is similar to the system in part a except that it needs to keep track of whether or not the number of zeros received is even. This basically doubles the number of states plus it needs an extra state for the start state giving a total of nine. The transitions are labeled X/Z, where X is the input and Z is the output. The resulting state diagram looks as follows:

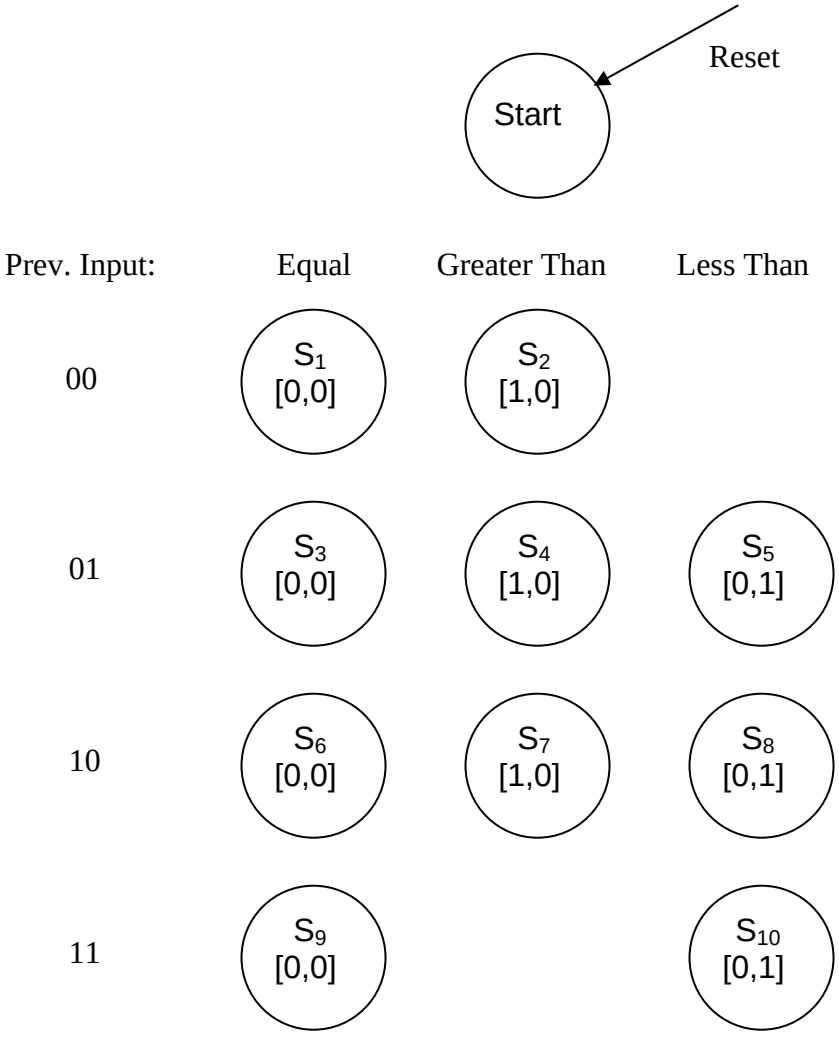


Exercise 7.26

(a) The Mealy machine for this system has four states for each of the combination of the previous input and the current input plus one for the start state. The transistions are labeled X_1X_2/Z_1Z_2, where X_1 and X_2 are the inputs and Z_1 and Z_2 the outputs. The resulting state diagram looks as follows:

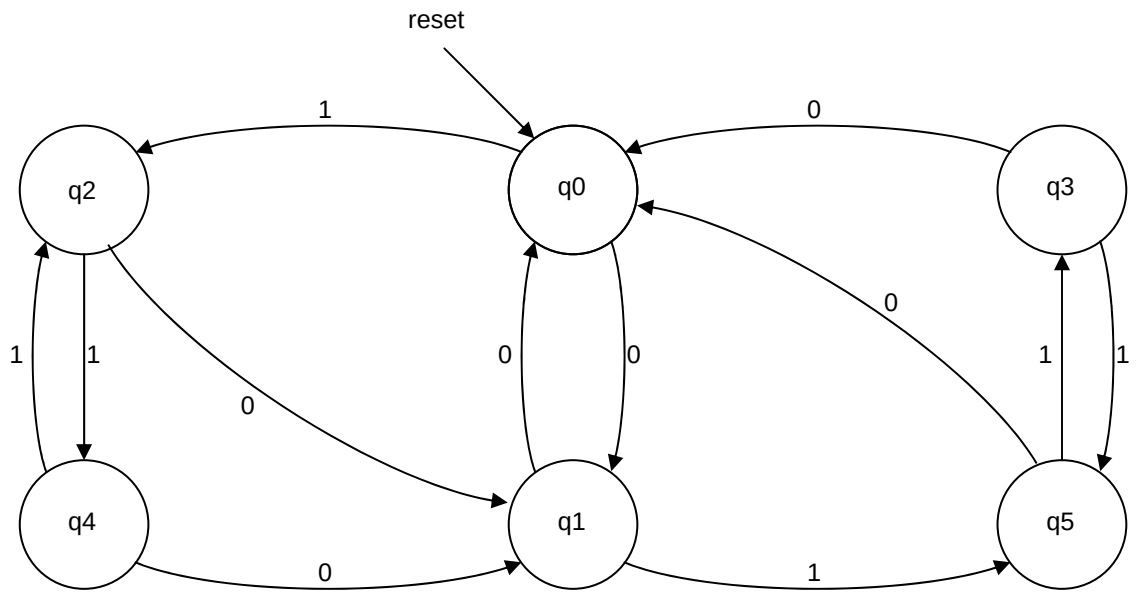


(b) Since the outputs of a Moore machine only depend on the current state, the Moore-implementation of this system is much more complicated. In order to simplify the diagram the transition arrows have been left out. Assume a transition arrow from each of the states to each state S_1 through S_{10} (this corresponds to 90 transitions, hence, the reason they were omitted). Each tier of the state diagram represents the previous value seen. The state diagram looks as follows:



Exercise 7.27

States q0, q2, and q4 represent an even number of 0's, with states q0, q1, and q2 representing an even number of zeros. The output would be one in states q1 and q3. The state diagram looks as follows:



Exercise 7.28

There are six different states that the lights can be in. The following chart illustrates one complete cycle of the traffic lights:

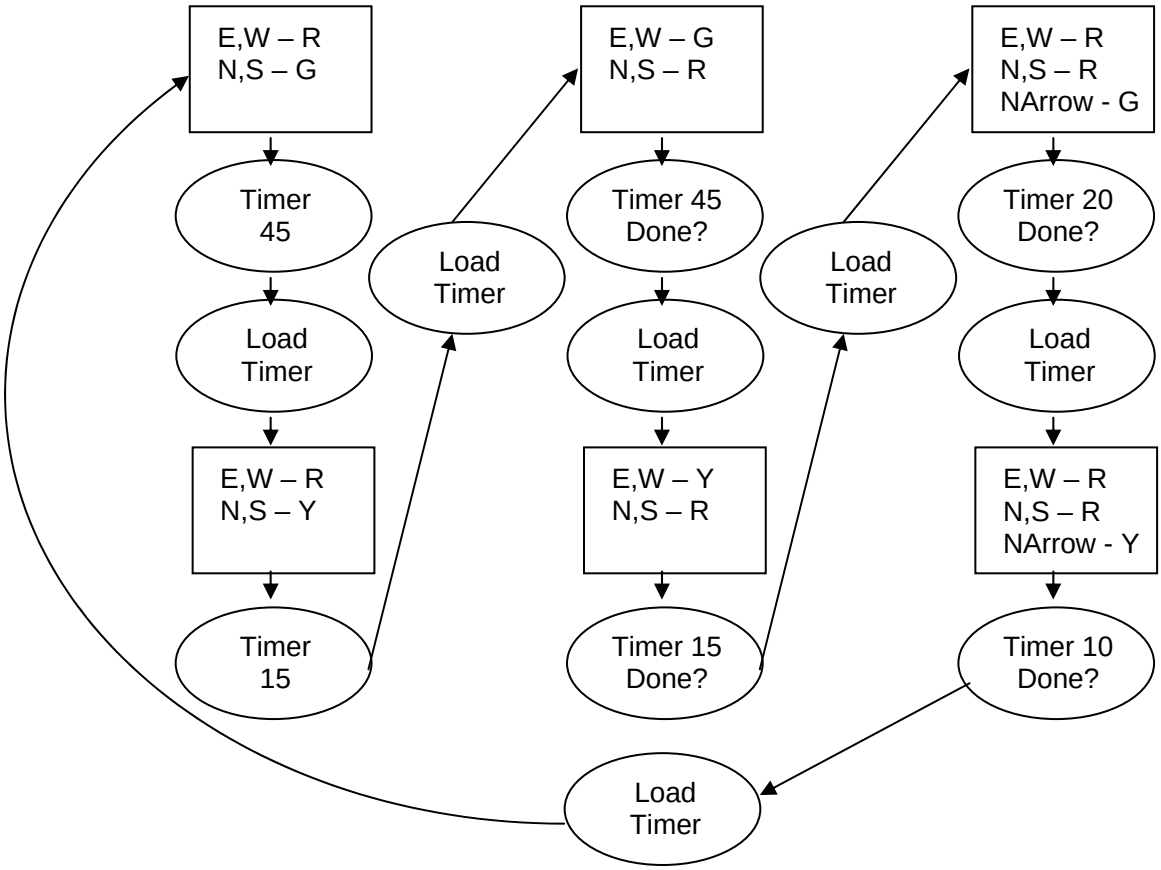
East	Red	Red	Green	Yellow	Red	Red	Red
West	Red	Red	Green	Yellow	Red	Red	Red
South	Green	Yellow	Red	Red	Red	Red	Green
North	Green	Yellow	Red	Red	G Arrow	Y Arrow	Green
	45	15	45	15	20	10	
	45	60	105	120	140	150	

The input signals to this system include:

Timer 45 done, Timer 15 done, Timer 20 done, Timer 10 done, and Reset

The output signals include:

East Red, East Yellow, East Green, West Red, West Yellow, West Green,
South Red, South Yellow, South Green, North Red, North Yellow, North Green,
N Arrow Yellow, N Arrow Green,
Load 45 Timer, Load 15 Timer, Load 20 Timer, Load 10 Timer



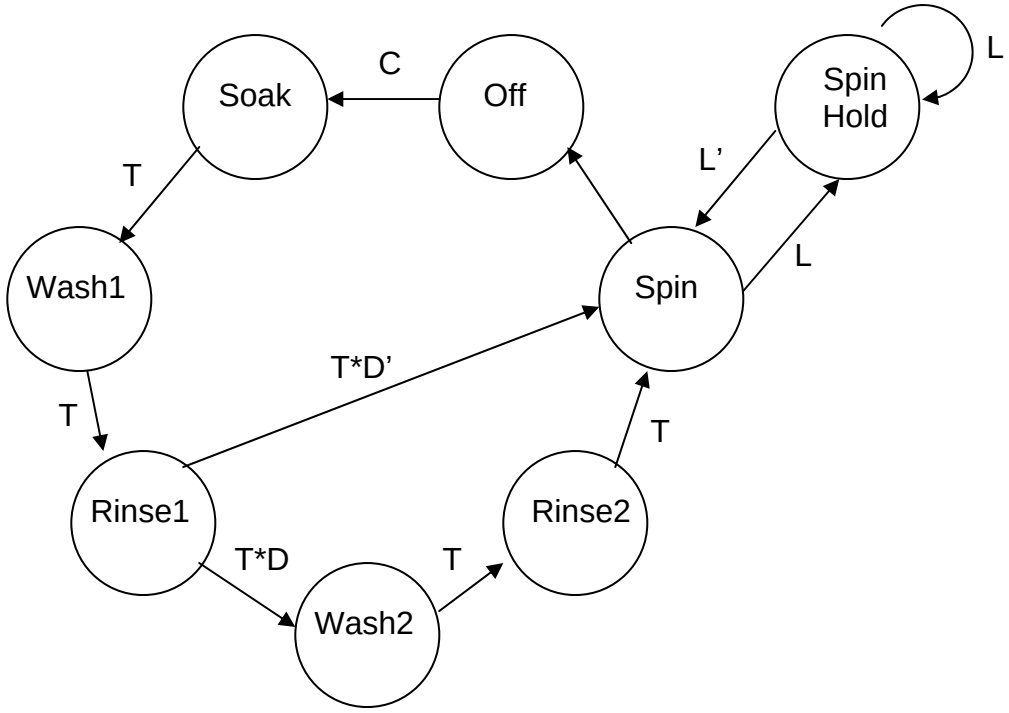
Traffic Controller State diagram

Exercise 7.29

The inputs for the washing machine design are as follows:

Coin = C, Timer = T, Double Wash = D, Lid Open = L

The state diagram looks as follows:



Exercise 7.30

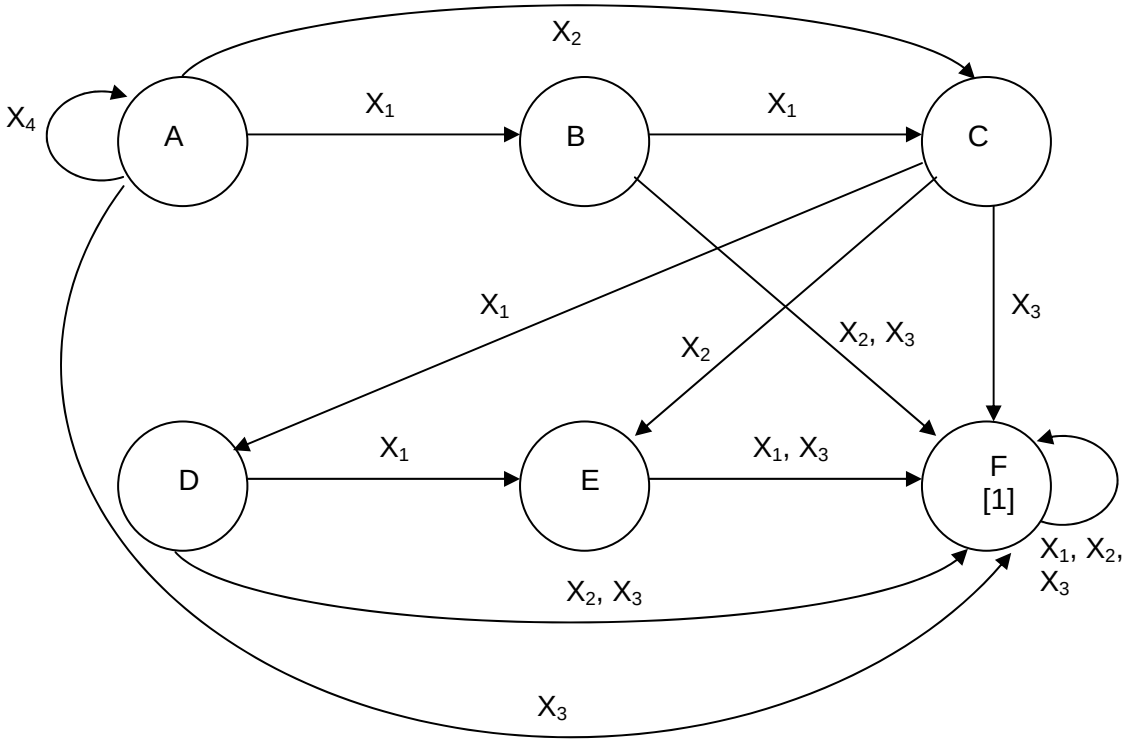
The candy vending machine has a total of 6 states. F is the only state in which the output is enabled. The states are as follows:

A = Reset, B = 5 cents, C = 10 cents, D = 15 cents, E = 20 cents, F = 25 cents or more

The inputs are as follows:

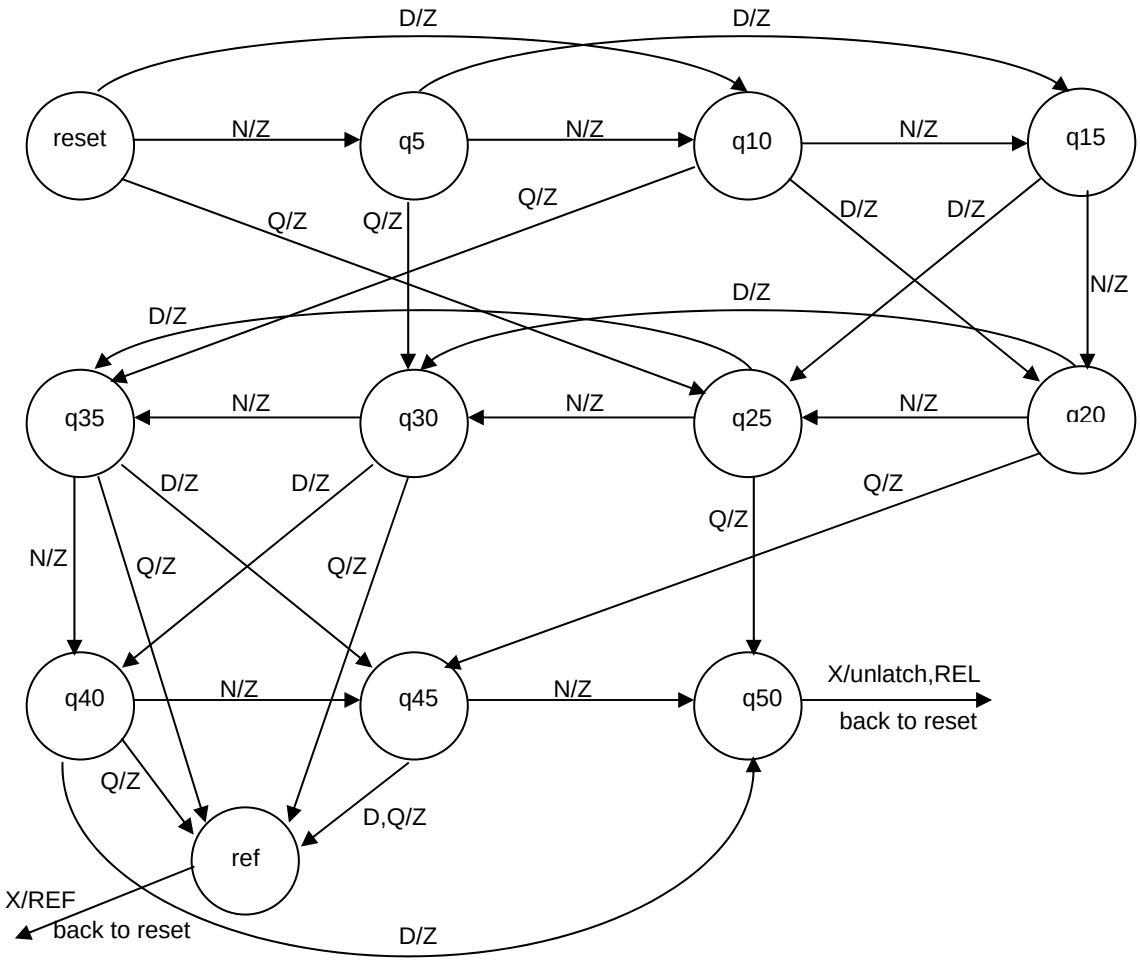
X_1 = 5 cents, X_2 = 10 cents, X_3 = 25 cents, X_4 = reset

The state diagram is as follows:

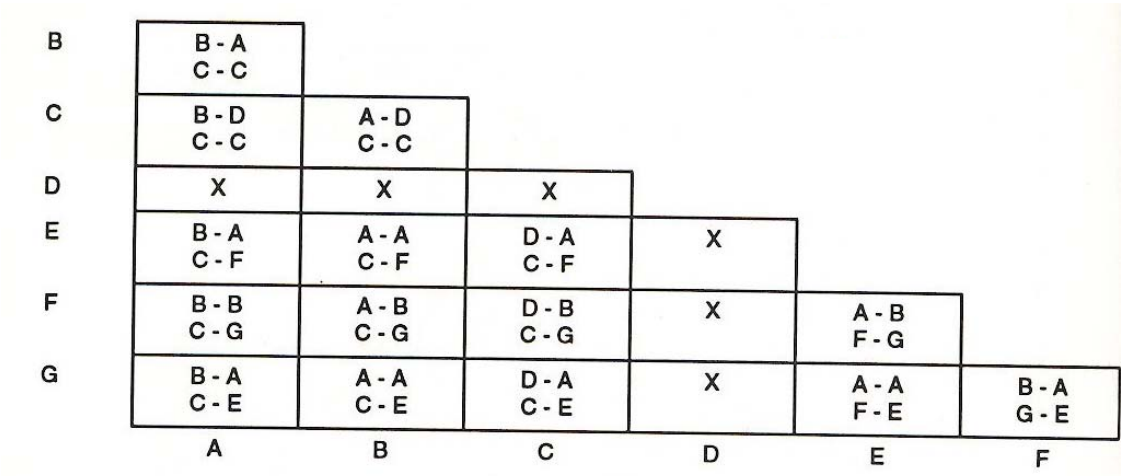


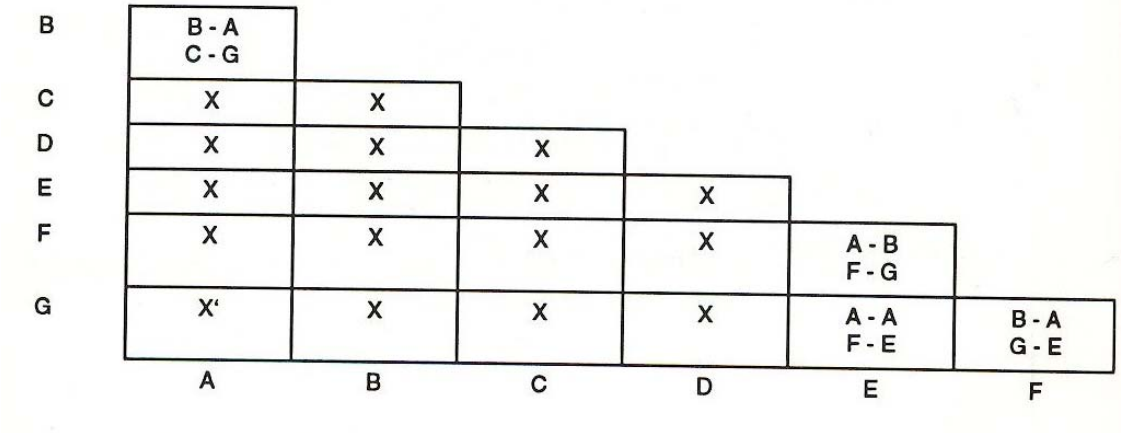
Exercise 7.31

The newspaper vending machine has a state that represents how much money has been deposited so far. Each state has three output transitions, which correspond to a nickel, dime and quarter being deposited. The transitions are labeled X/Z, where X represents the input and Z represents the output. If there is no input on a transition it is labeled X, similarly, if there is no output signals asserted it is labeled Z. The only state that a newspaper is released in is state q50. If the user deposits more than 50 cents, the system jumps to a ref state where the money deposited is refunded. The resulting state diagram is as follows:



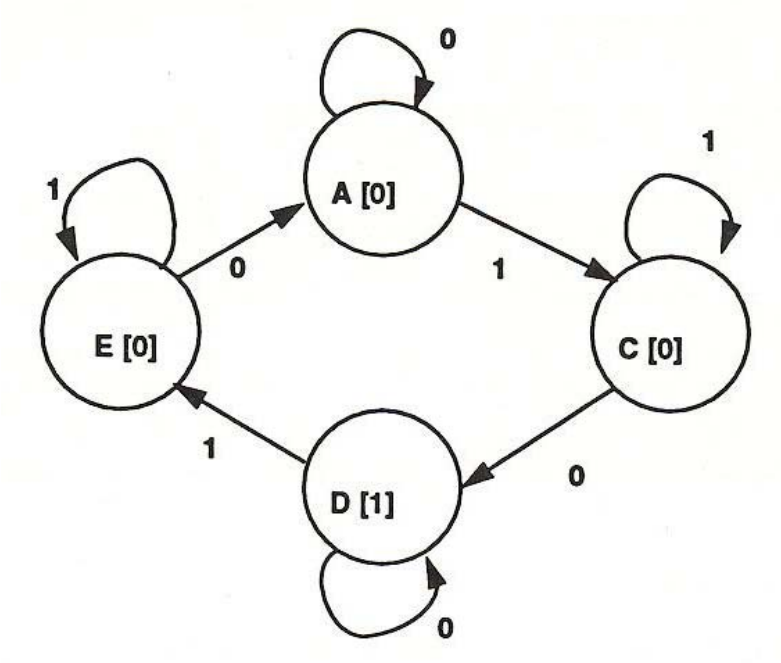
Exercise 8.1


B
C
D
E
F
G
B - A
C - C
B - D
C - C
A - D
C - C
X
X
X
B - A
C - F
A - A
C - F
D - A
C - F
X
B - B
C - G
A - B
C - G
D - B
C - G
X
A - B
F - G
B - A
C - E
A - A
C - E
D - A
C - E
X
A - A
F - E
B - A
G - E
A
B
C
D
E
F


B
C
D
E
F
G
B - A
C - G
X
X
X
X
X
X
X
X
X
X
X
X
X
X
X
X
A - B
F - G
X'
X
X
X
A - A
F - E
B - A
G - E
A
B
C
D
E
F

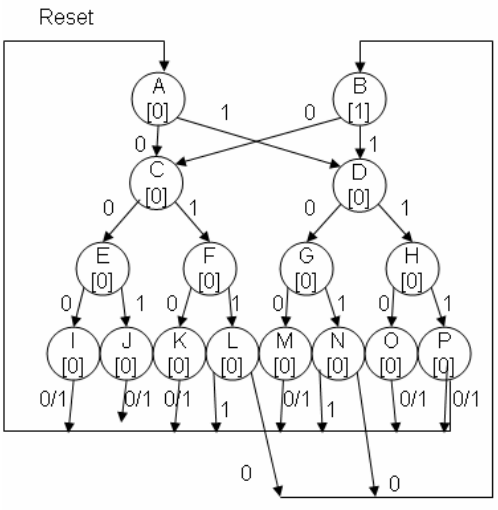
Exercise 8.2

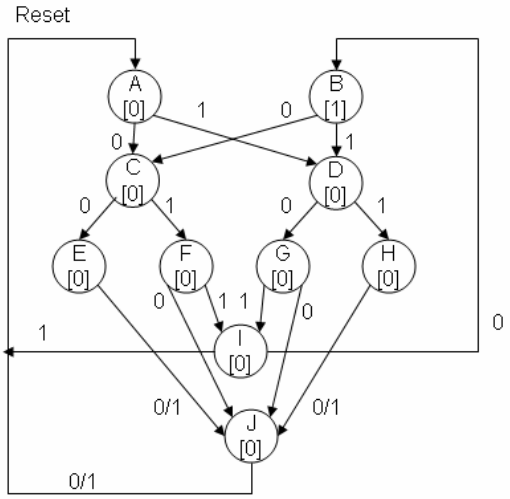
States A and B (only A is shown below) are equivalent as are E, F, and G (only E is shown below).

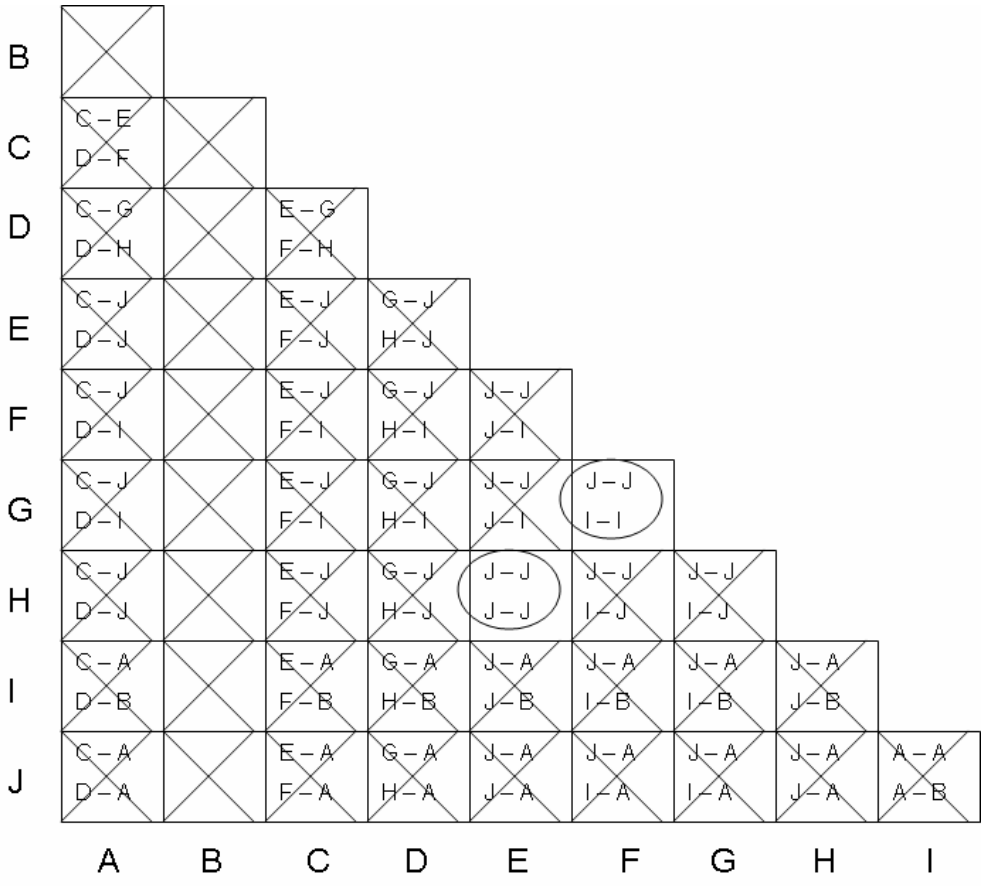


Exercise 8.3

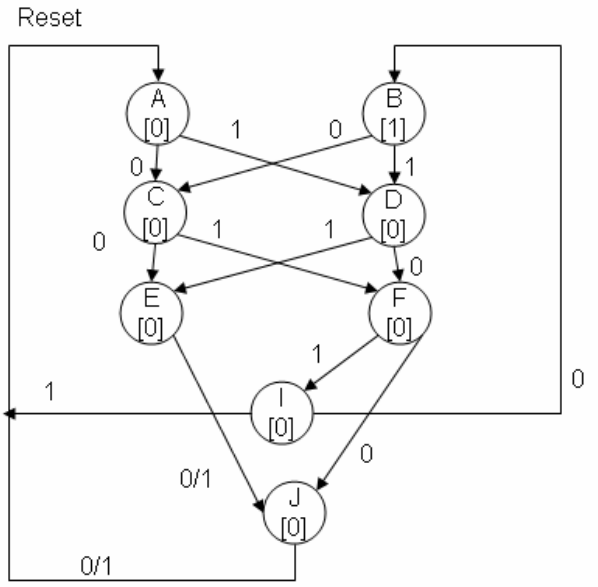
Since there are a lot of states in this example, some pre-reduction has been done before the implication chart to make the implication chart smaller. The image on the left, is the state diagram before any reduction, and the image on the right is the state diagram being used in the implication chart.



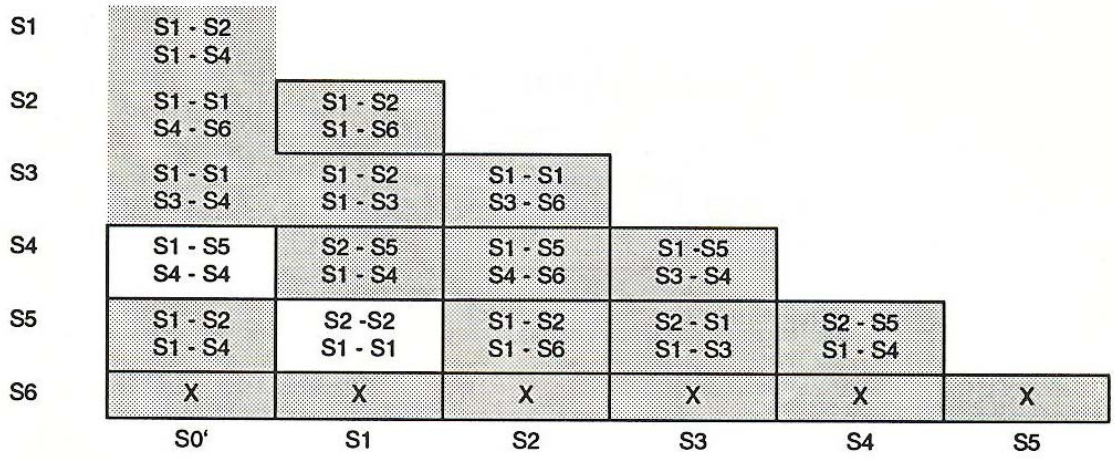




The final state diagram comes out as follows:



Exercise 8.4



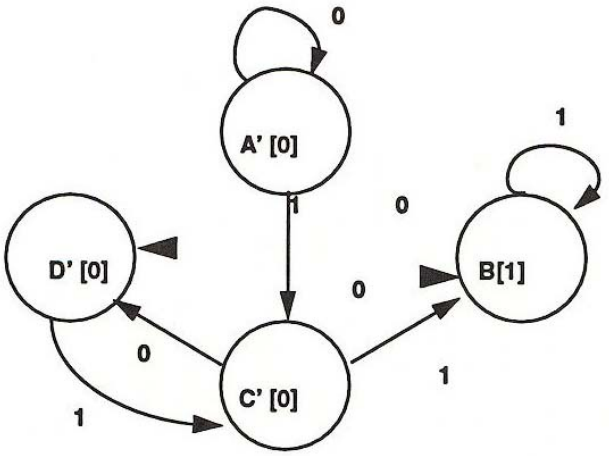
S0 = S4, S1 = S5.

Exercise 8.5

	A	B	C	D	E	F
B	X					
C	A - D C - B	X				
D	A - B C - C	X	D - B B - C			
E	A - E C - F	X	D - E G - F	B - E C - F		
F	A - G C - B	X	D - G B - B	B - G C - B	E - G F - B	
G	A - B C - F	X	D - B B - F	B - B C - F	E - B F - F	G - B B - F

A = E, C = F, D = G.

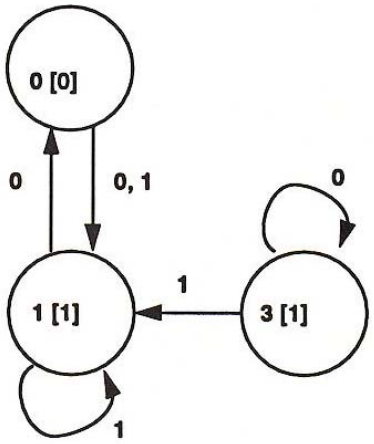
Any string of 2 or more 1s or a double 0 will cause a 1 output.



Exercise 8.6

1	X					
2	x	0 - 0 1 - 5				
3	x	0 - 3 1 - 1	0 - 3 1 - 5			
4	x	0 - 0 1 - 5	0 - 0 5 - 5	0 - 3 1 - 5		
5	x	0 - 0 1 - 5	0 - 0 5 - 5	0 - 3 1 - 5	0 - 0 5 - 5	
6	5 - 5 2 - 4	X	x	x	x	x
	0	1	2	3	4	5

1 = 2 = 4 = 5

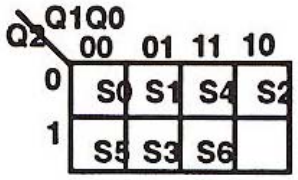


Exercise 8.7

a.) Minimum bit change heuristic

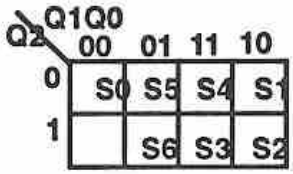
7 states, 3 variable K-map

Assign S0 = 000 = (Q2, Q1, Q0)
S0 adjacent to S1, S2
Either S4 adjacent to S1, S2, or S3 adjacent to S1, try both cases.



b.) State assignment guidelines

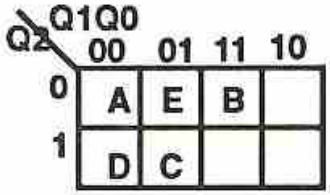
Highest priority: 2x (S5, S6), 2x (S1, S2) (S3, S4)
Medium priority: 2x (S3, S4), (S1, S2), (S5, S6)
Lowest priority: 0/0: (S4, S0, S6, S5)
1/0: (S0, S2, S1, S4, S5)
0/1: (S1, S2, S3)
1/1: (S6, S3)



Satisfy all high and medium priority.
Try to satisfy as many lowest priority as possible.

Exercise 8.8

High Priority: (B, C), (E, A)
Medium Priority: (A, D), (D, C)
Lowest Priority: 0/0: (E, D, C, B)
1/0: (B, C, A)



Exercise 8.9

Let Q_1, Q_0 be the current state bits and P_1, P_0, be the next state bits with C, T_L, T_S as inputs and S_T, H_1, H_0, F_1, F_0.

Suppose the random 2-bit assignment is:

$HG = Q_0'Q_1'$

$HY = Q_0Q_1$

$FG = Q_0'Q_1$

$FY = Q_0Q_1'$

Produces the following function:

$P_0 = T_S' Q_0$

$P_1 = Q_1 W$

$ST = T_S Q_0 + Q_0' Z$

$H_1 = Q_1 \text{ xor } Q_0$

$H_0 = Q_0Q_1$

$F_1 = Q_1 \text{ xnor } Q_0$

$F_0 = Q_0Q_1'$

$W = Q_0 + Q_0'CT_L'$

$X = Q_0$

$Y = C' + T_L$

$Z = CT_L Q_1' + Y Q_1$

This encoding has 28 literals.

Using the random 3-bit assignment:

$HG = Q_0Q_1'Q_2'$

$HY = Q_0Q_1Q_2'$

$FG = Q_0'Q_1'Q_2$

$FY = Q_0'Q_1Q_2$

Generates the encoding:

$P_0 = Q_0Q_1' + T_S F_0 + T_S' H_0$

$P_1 = T_S' Q_1$

$P_2 = FG + T_S H_0 + T_S' F_0$

$ST = Q_1' W + T_S Q_1$

$H_1 = Q_1$

$H_0 = Q_0Q_1$

$F_1 = Q_2$

$F_0 = Q_0'Q_1$

$Y = C' + T_L$

$W = CT_L\ Q_0 + Y\ Q_2$

This encoding has only 30 literals.

Finally trying a four-bit encoding:

$HG = Q_0Q_1'Q_2'Q_3'$

$HY = Q_0Q_1Q_2'Q_3'$

$FG = Q_0'Q_1'Q_2Q_3$

$FY = Q_0'Q_1'Q_2'Q_3$

Generates:

$P_0 = P_1 + HG$

$P_1 = T_S'\ Q_1$

$P_2 = T_S'\ Q_2$

$P_3 = FG + P_2$

$ST = CT_L\ HG + T_S\ X + Y\ FG$

$H_1 = Q_3$

$H_0 = Q_1$

$F_1 = Q_0$

$F_0 = Q_2$

$FG = Q_2'Q_3$

$HG = Q_0Q_1'$

$X = Q_1 + Q_2$

$Y = C + T_L$

This encoding also gives 27 literals.

Exercise 8.10

Using the solution for 8.9, shows that varying the number of bits used to encode the states, and choosing different bit encodings can provide more or less optimal solutions. For example, the 4-bit encoding used has fewer literals than both the 2 and 3 bit encoding, and the 3-bit encoding has more literals than the 2-bit encoding. Finding the most optimal encoding when varying both the number of bits and the different encodings of each state is an extremely difficult problem. This is because it is not inherently easy to determine whether or not one encoding will be better than all others, unless all the other encodings are tried. For *m* states and *n* bits, $m! / (m - 2^n)!$ where $n \leq \log_2 m$ different possible encodings.

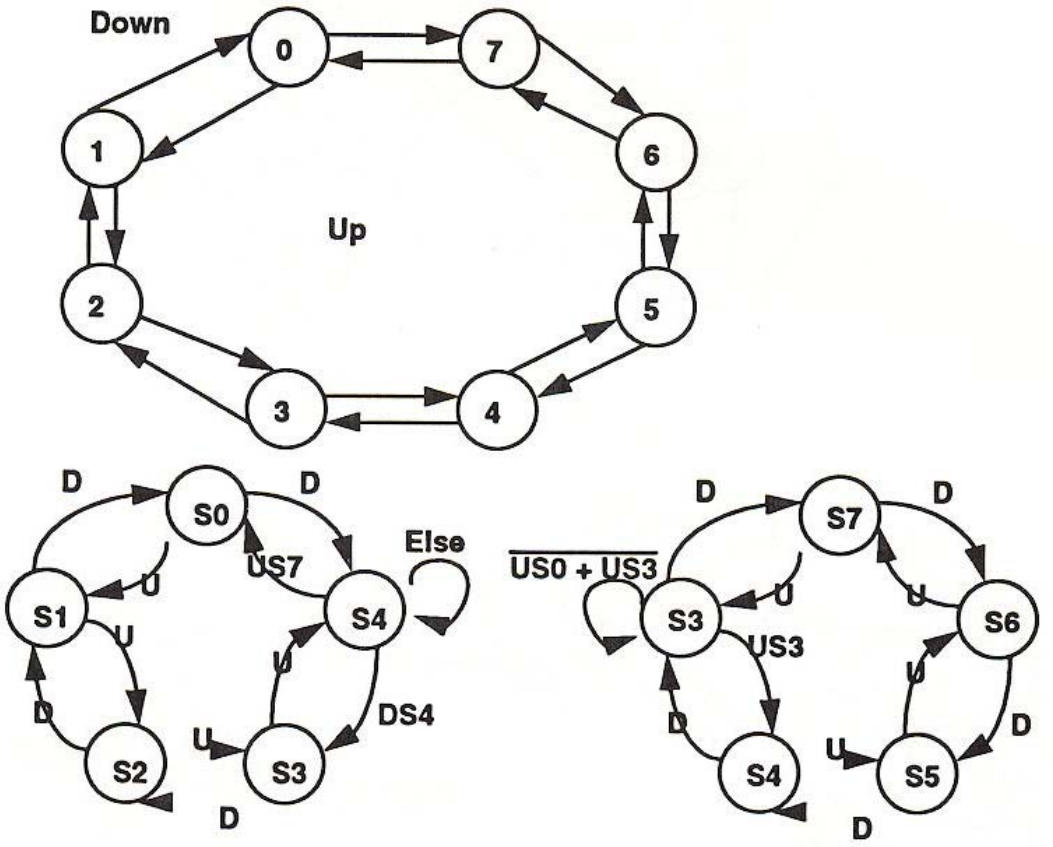
Exercise 8.11

This problem was not completely specified and should be ignored until the text is corrected.

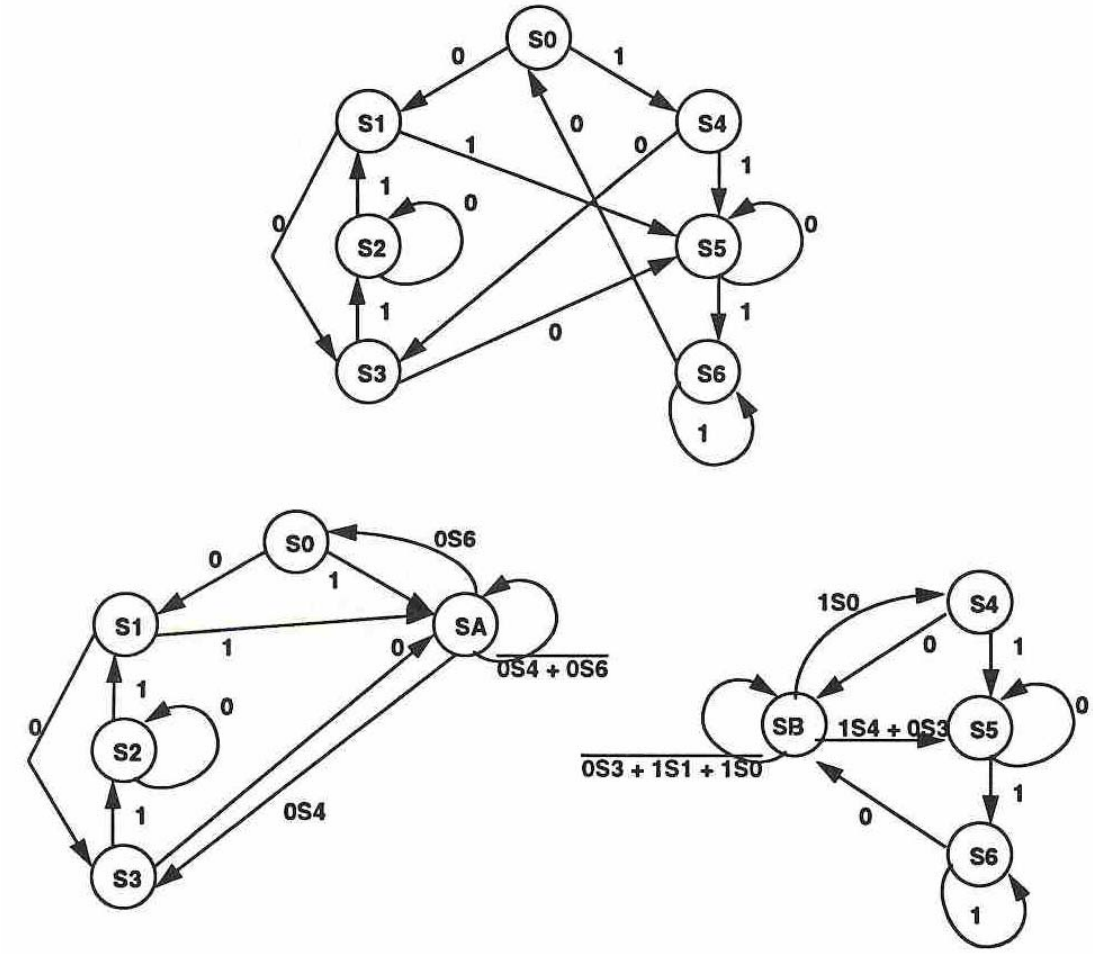
Exercise 8.12

This problem was not completely specified and should be ignored until the text is corrected.

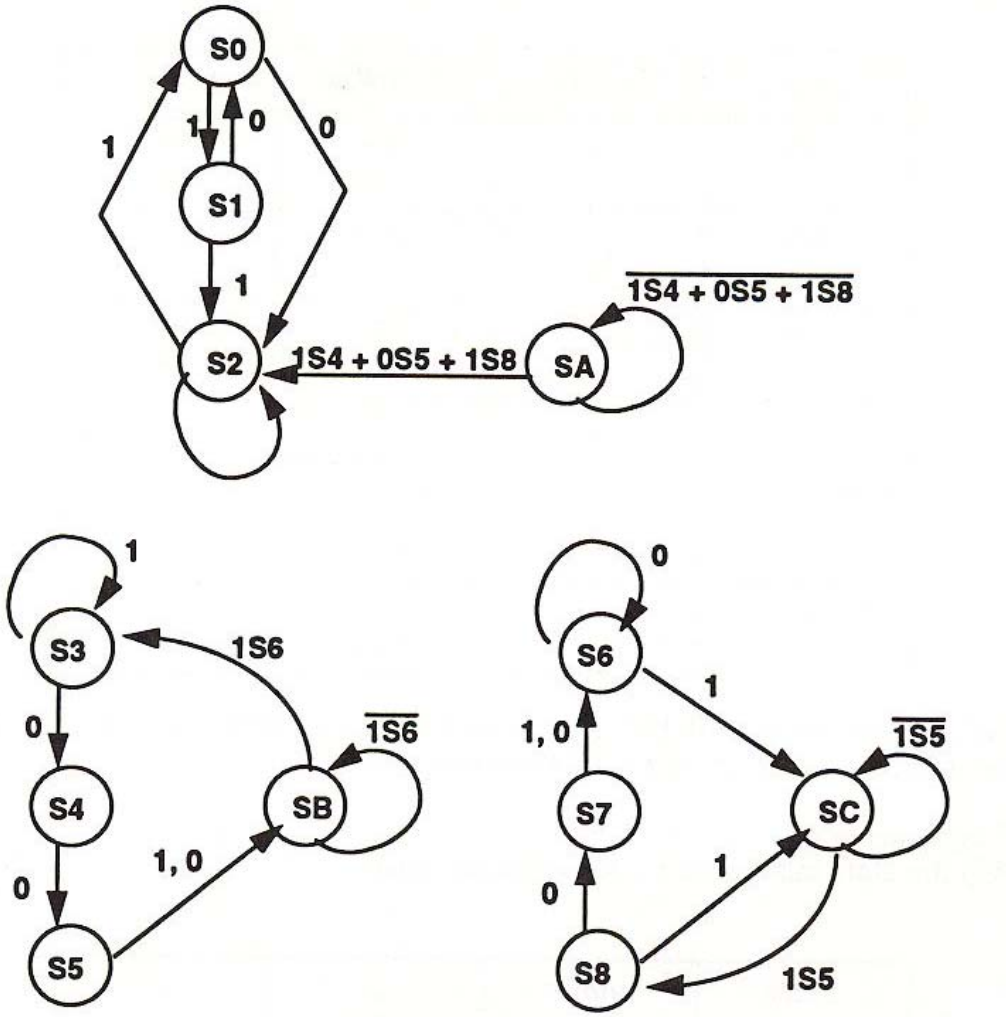
Exercise 8.13



Exercise 8.14



Exercise 8.15



Exercise 8.16

a.) Mealy partitioning rule modification

Each branch that enters the idle state must have an "idle" output, and transfers out of the idle state must be associated with the proper output.

b.) Moore partitioning rule modification

The idle state must have an "idle" output. Combine them by tri-stating one section when in the idle state.

Exercise 8.17

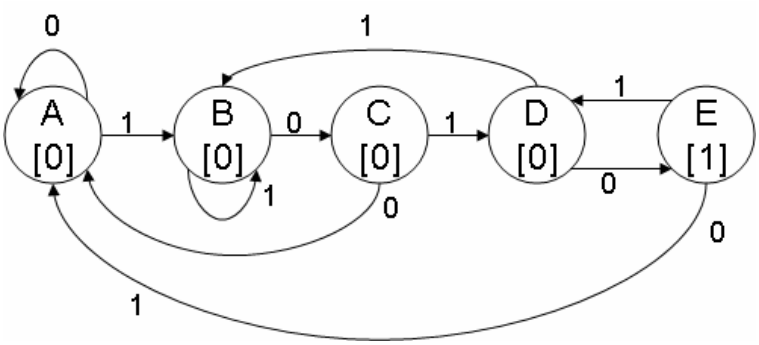
State	Input	Next State	Output
000	0 1	000 010	0 0
001	0 1	100 000	1 0
010	0 1	001 100	0 1
011	0 1	001 010	1 0
100	0 1	101 011	1 0
101	0 1	011 101	0 1

For implementation with D-FFs, minimize the above table with espresso or K-maps. The resulting equations are the next state and output functions.

Exercise 8.18

At the writing of this solution manual, there was a mistake in the printing of Figure Ex 8.18. This is due to the non-deterministic behavior illustrated by the two 0 arcs leaving state D. The arc from D to A on 0 should actually be from E to A.

The actual figure should look like:



The next-state function for this FSM looks like the following:

Current State	Encoding	Input	Reset	Next State	Output
Any	XXX	X	1	000	0
A	000	0	0	000	0
	000	1	0	001	0
B	001	0	0	011	0
	001	1	0	001	0
Invalid	010	X	0	000	0
C	011	0	0	000	0
	011	1	0	111	0
Invalid	100	X	0	000	0
E	101	0	0	000	1
	101	1	0	111	1
Invalid	110	X	0	000	0
D	111	0	0	101	0
	111	1	0	001	0

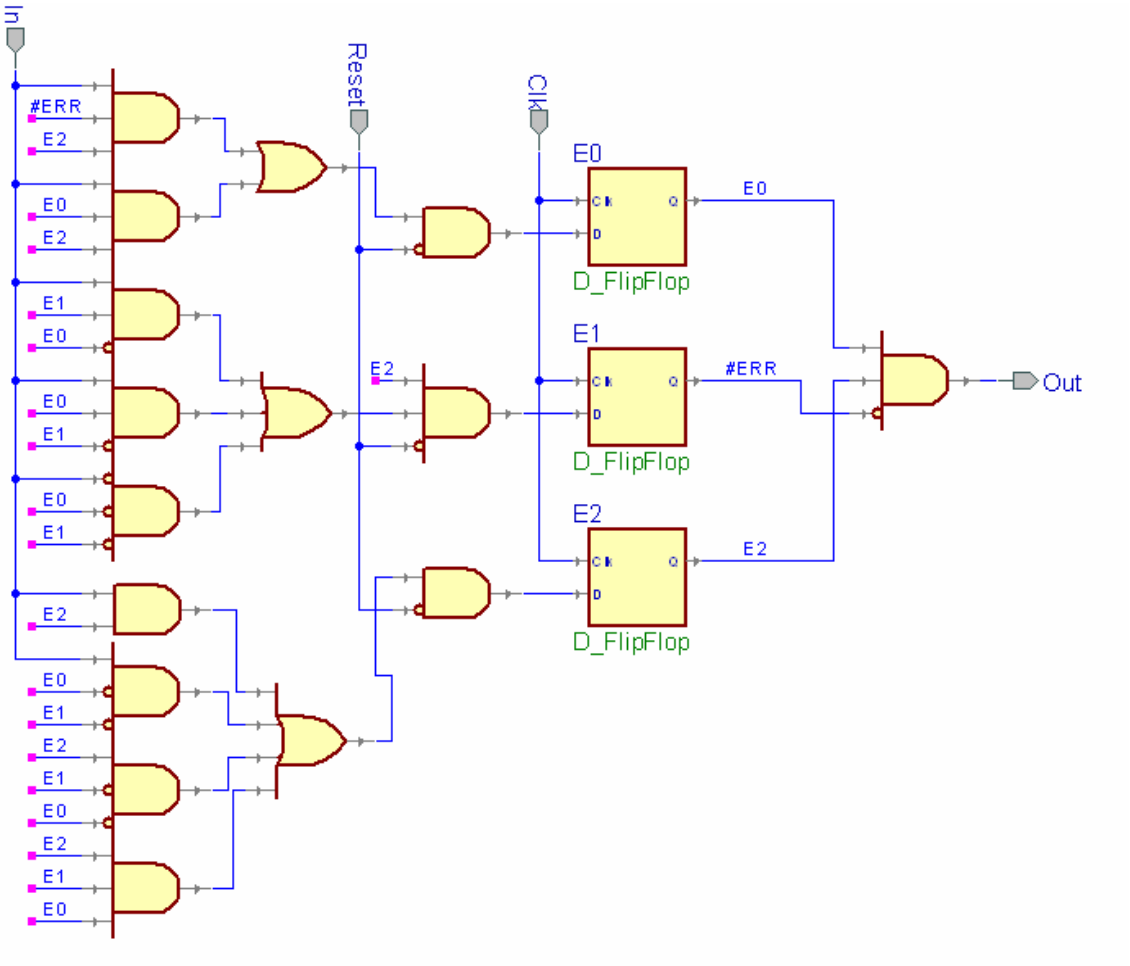
Using K-maps, the Encoding bits E_0, E_1, E_2 and input I translate into N_0, N_1, N_2 and O in the following reduced equations:

$N_0 = \text{Reset}' (I\ E_1\ E_2 + I\ E_0\ E_2)$

$N_1 = \text{Reset}' (I\ E_0\ E_1' \ E_2 + I\ E_0' \ E_1\ E_2 + I' \ E_0' \ E_1\ E_2)$

$N_2 = \text{Reset}' (I\ E_2 + E_0\ E_1\ E_2 + E_0' \ E_1' \ E_2 + I\ E_0' \ E_1' + E_0' \ E_1' \ E_2)$

$O = E_0\ E_1' \ E_2$


In
Reset
Clk
#ERR
E2
E0
E2
E1
E0
E0
E1
E0
E1
E2
E0
E1
E2
E1
E0
E2
E1
E0
E2
E0
E1
E0
D_FlipFlop
E1
#ERR
D_FlipFlop
E2
E2
D_FlipFlop
Out

Exercise 8.19

TS	TL	C	Reset	State	HR	HY	HG	FR	FY	FG
X	X	X	1	6'bxxxxxx	0	0	1	1	0	0
X	0	X	0	Highwaygreen	0	0	1	1	0	0
X	X	0	0	Highwaygreen	0	0	1	1	0	0
X	1	1	0	Highwaygreen	0	1	0	1	0	0
0	X	X	0	Highwayyellow	0	1	0	1	0	0
1	X	X	0	Highwayyellow	1	0	0	0	0	1
X	0	1	0	Farmroadgreen	1	0	0	0	0	1
X	1	X	0	Farmroadgreen	1	0	0	0	1	0
X	0	0	0	Farmroadgreen	1	0	0	0	1	0
0	X	X	0	Farmroadyellow	1	0	0	0	1	0
1	X	X	0	Farmroadyellow	0	0	1	1	0	0

HR = (Highwayyellow)(TS) + (FarmroadGreen) + (Farmroadyellow)(TS)'

HY = (Highwaygreen)(C)(TL) + (Highwayyellow)(TS)'

HG = (Highwaygreen)(C)' + (Highwaygreen)(TL)' + (Farmroadyellow)(TS)

FR = (Farmroadyellow)(TS) + (Highwaygreen) + (Highwayyellow)(TS)'

FY = (Farmroadgreen)(C) + (Farmroadgreen)(TL) + (Farmroadyellow)(TS)'

FG = (Farmroadgreen)(C)(TL)' + (Highwayyellow)(TS)

Exercise 8.20

```
module prob8_20 ( Out ,Clk ,Reset ,In );
input Clk, Reset, In ;
wire Clk, Reset, In ;
output Out ;

parameter S0 = 3'b000;
parameter S1 = 3'b001;
parameter S2 = 3'b010;
parameter S3 = 3'b011;
parameter S4 = 3'b100;
parameter S7 = 3'b101;
parameter S10 = 3'b110;
parameter Sinv = 3'b111; // invalid state

reg Out;
reg [3:1] state ;

always @(posedge Clk) begin
      Out = 0;
      if (Reset)
            state = S0;
      else case (state)
            S0:
                  if (In) state = S2;
                  else state = S0;
            S1:
                  if (In) state = S4;
                  else state = S3;
            S2:
                  if (In) state = S3;
                  else state = S4;
            S3:
                  state = S7;
            S4:
                  if (In) state = S10;
                  else state = S7;
            S7:
                  state = S0;
            S10:
                  begin
                        state = S0;
                        if (!In) Out = 1;
                  end
            Sinv: // Does the same as reset
                  state = S0;
      endcase

end

endmodule
```

Exercise 8.21

Note that there is an error in the diagram for figure 8.13, state S_3 has two outputs marked 11 and is missing an output for 01. This solution assumes that the arc from S_3 to S_0 should be 01 instead of 11.

```
module prob8_21 ( Out ,In ,Clk);

input [2:0] In;
wire [2:0] In ;
input Clk;
wire Clk;

output Out ;
wire Out ;

parameter I0 = 2'b00;
parameter I1 = 2'b01;
parameter I2 = 2'b10;
parameter I3 = 2'b11;

parameter S0 = 3'b000;
parameter S1 = 3'b001;
parameter S2 = 3'b010;
parameter S3 = 3'b011;
parameter S4 = 3'b100;
parameter S5 = 3'b101;
parameter S6 = 3'b110;
parameter S7 = 3'b111;

reg [3:1] state;
assign Out = !state[1]; // the last state bit determines if the
               // state is even or odd, even states
                           // are the only ones that output a 1

always @(posedge Clk) begin
      case (state)
            S0:
                  case (In)
                        I1: state = S1;
                        I2:   state = S2;
                        I3:   state = S3;
                  endcase
            S1:
                  case (In)
                        I0:   state = S0;
                        I1:   state = S3;
                        I3:   state = S5;
                  endcase
            S2:
                  case (In)
                        I0: state = S1;
                        I1:   state = S3;
                        I3:   state = S4;
                  endcase
```

```
            S3:
                  case (In)
                        I0:   state = S1;
                        I1:   state = S0;
                        I2:   state = S4;
                        I3:   state = S5;
                  endcase
            S4:
                  case (In)
                        I0:   state = S0;
                        I1:   state = S1;
                        I2:   state = S2;
                        I3:   state = S5;
                  endcase
            S5:
                  case (In)
                        I0:   state = S1;
                        I1:   state = S4;
                        I2:   state = S0;
                  endcase
            S6:
                  state = S0;
            S7:
                  state = S0;
      endcase
end

endmodule
```

Exercise 8.22

This solution divides the partition work into three modules. The prob8_22 module acts as a connecting block between the two partitions. Partition A handles the left side of Figure 8.38 and Partition B handles the right side.

```
module prob8_22 ( Reset ,Clk ,U ,D );
input Reset, Clk, U, D;
wire Reset, Clk, U, D, S0, S2, S3, S5;
PartitionA partA( Reset, Clk, U, D, S3, S5, S0, S2);
PartitionB partB( Reset, Clk, U, D, S0, S2, S3, S5);
endmodule

module partA ( Reset ,Clk ,U ,D ,S3 ,S5 ,S0 ,S2);
      input Reset, Clk, U, D, S5, S3;
      output S0, S2;
      wire Clk, U, D, S5, S3, S0, S2;

      parameter state0 = 2'b00;
      parameter state1 = 2'b01;
      parameter state2 = 2'b10;
      parameter stateA = 2'b11;

      reg [2:1] state;

      assign S0 = !state[1] && !state[2];
      assign S2 = state[1] && state[2];

      always @(posedge Clk) begin
            if (Reset) state = state0;
            else case (state)
                  state0:
                        begin
                              if (U) state = state1;
                              if (D) state = stateA;
                        end
                  state1:
                        begin
                              if (U) state = state2;
                              if (D) state = state0;
                        end
                  state2:
                        begin
                              if (U) state = stateA;
                              if (D) state = state1;
                        end
                  stateA:
                        begin
                              if (U & S5) state = state0;
                              if (D & S3) state = state2;
                        end
            endcase
      end
endmodule
```

```
module partB ( Reset ,Clk ,U ,D ,S0 ,S2 ,S3 ,S5 );
      input Reset, Clk, U, D, S0, S2;
      output S3, S5;
      wire Reset, Clk, U, D, S0, S2;

      parameter state3 = 3'b00;
      parameter state4 = 3'b01;
      parameter state5 = 3'b10;
      parameter stateB = 3'b11;

      reg [2:1] state;

      assign S3 = !state[1] & !state[2];
      assign S5 = state[1] & state[2];

      always @(posedge Clk) begin
            if (Reset) state = stateB;
            else case (state)
                  state3:
                        begin
                              if (U) state = state4;
                              if (D) state = stateB;
                        end
                  state4:
                        begin
                              if (U) state = state5;
                              if (D) state = state3;
                        end
                  state5:
                        begin
                              if (U) state = stateB;
                              if (D) state = state4;
                        end
                  stateB:
                        begin
                              if (U & S2) state = state3;
                              if (D & S0) state = state5;
                        end
            endcase
      end
endmodule
```

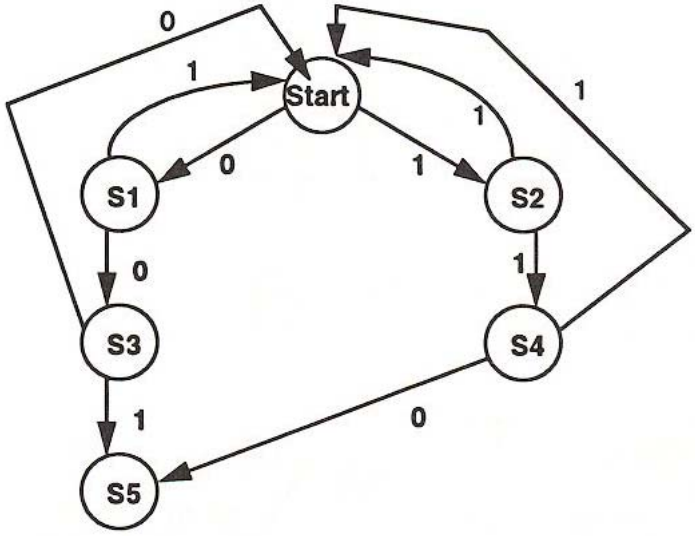
Exercise 8.23

It is possible to rectify this problem without adding any additional states to either state machine. The trick is to have every state between T_{04} and T_{19} output TS. This way if the timer gets past T_{04} before the other state machine can react, it will eventually catch the TS signal from a later state. This does not harm the other state machine, because once the other state machine recognizes TS, the controller is in the FG or HG states and does not care about TS again until it transitions to FY or HY and resets the timer to T_{00} with ST.

Exercise 8.24

0 is asserted in states Start and S1 to S4. 1 is asserted in S5. Various state assignments may be used, including

State	Coding
Start	000
S1	001
S2	010
S3	011
S4	110
S5	111



Exercise 8.25

This solution uses T flip-flops. For an implementation using D flip-flops, simply use K-maps for Q1+ and Q0+ directly.

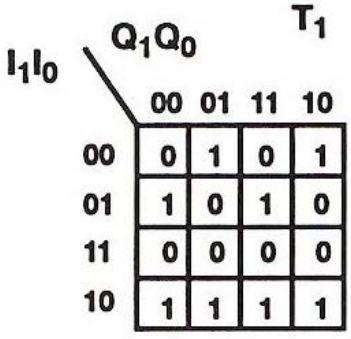
a.) State table

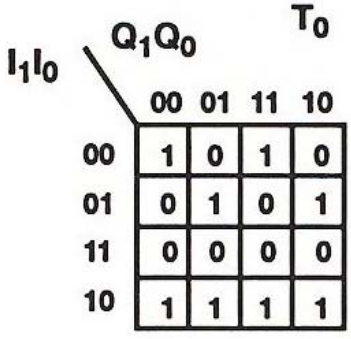
Q1 Q0	I1 I0	Q1+ Q0+	T1 T0
00	0 0	0 1	0 1
	0 1	1 0	1 0
	1 0	1 1	1 1
	1 1	0 0	0 0
0 1	0 0	1 1	1 0
	0 1	0 0	0 1
	1 0	1 0	1 1
	1 1	0 1	0 0
1 0	0 0	0 0	1 0
	0 1	1 1	0 1
	1 0	0 1	1 1
	1 1	1 0	0 0
1 1	0 0	1 0	0 1
	0 1	0 1	1 0
	1 0	0 0	1 1
	1 1	1 1	0 0

b.) Inputs to T1 and T0

$$T1 = \bar{Q}_1\bar{Q}_0\bar{I}_1I_0 + I_1\bar{I}_0 + \bar{Q}_1Q_0\bar{I}_0 + Q_1\bar{Q}_0\bar{I}_1I_0 + Q_1\bar{Q}_0\bar{I}_0$$

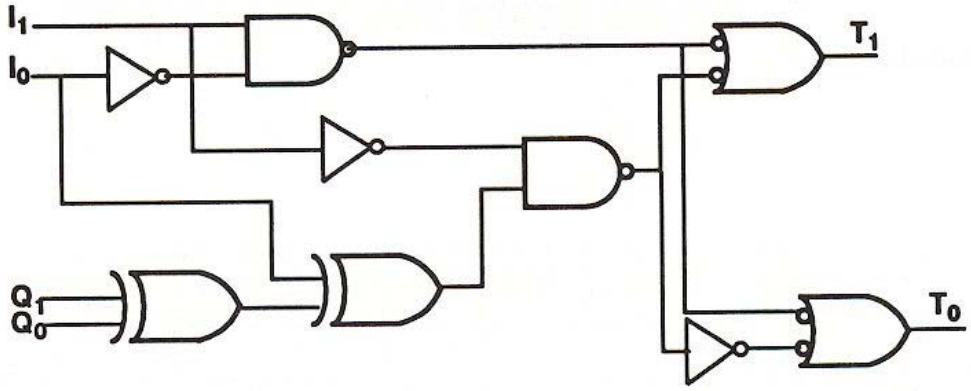
$$T2 = \bar{Q}_1Q_0\bar{I}_1I_0 + I_1\bar{I}_0 + Q_1Q_0\bar{I}_0 + Q_1\bar{Q}_0\bar{I}_1I_0 + \bar{Q}_1\bar{Q}_0I_0$$





c.) Schematic implementation

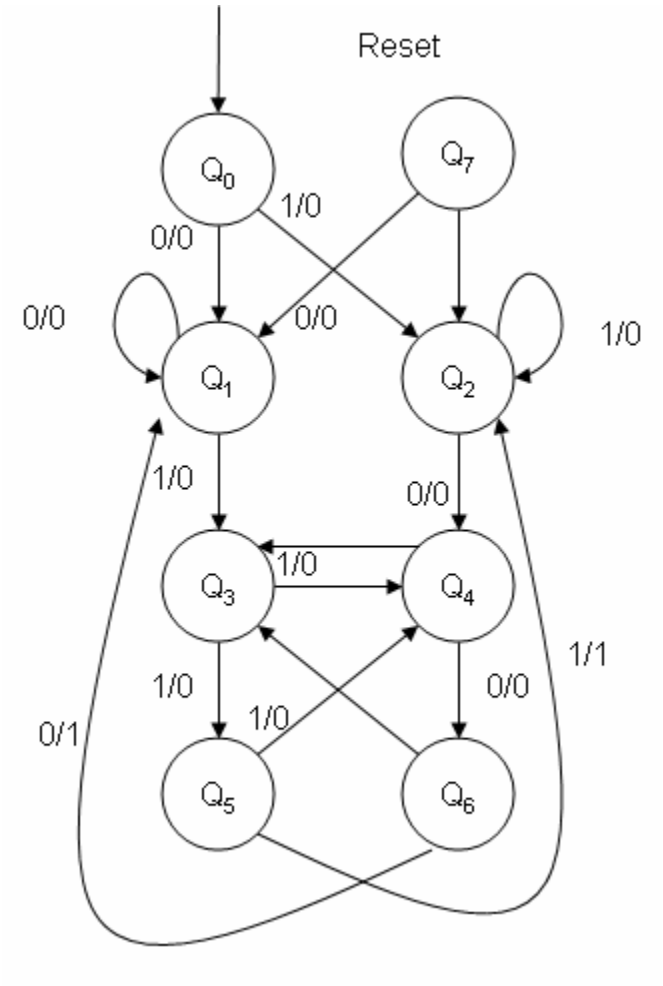
$$T_1 = I_1\bar{I}_0 + (Q_1 \oplus Q_0 \oplus \bar{I}_0)\bar{I}_1$$
$$T_0 = I_1\bar{I}_0 + (Q_1 \oplus Q_0 \oplus \bar{I}_0)\bar{I}_1$$



Other schematics may be possible.

Exercise 8.26

(a) The state diagram below will implement the function:



(b) The next-state function looks like

Input	Current State	Next State	Output
0	Q_0	Q_1	0
1	Q_0	Q_2	0
0	Q_1	Q_1	0
1	Q_1	Q_3	0
0	Q_2	Q_4	0
1	Q_2	Q_2	0
0	Q_3	Q_4	0
1	Q_3	Q_5	0
0	Q_4	Q_6	0
1	Q_4	Q_3	0
0	Q_5	Q_1	0
1	Q_5	Q_4	1
0	Q_6	Q_1	1
1	Q_6	Q_3	0
0	Q_7	Q_1	0
1	Q_7	Q_2	0

(c) By row-matching, state seven can be left out of the minimized state diagram.

(d) The table below shows the encoding assignment:

State	Encoding
Q_0	010
Q_1	110
Q_2	000
Q_3	011
Q_4	101
Q_5	001
Q_6	100

Q_0 = Reset
$Q_1 = I' (Q_0 + Q_1 + Q_6)$
$Q_2 = I (Q_0 + Q_2 + Q_5)$
$Q_3 = I (Q_1 + Q_4 + Q_6)$
$Q_4 = I' (Q_2 + Q_3 + Q_5)$
$Q_5 = IQ_3$
$Q_6 = I'Q_4$

Suppose the Encoding bits are E_0, E_1, E_2 where 0 is the high order bit. Using the state equivalent of a K-map, the equations for each state become:

Q_0 = Reset
$Q_1 = I'E_2' (E_0 + E_1)$
$Q_2 = I E_0' (E_1' + E_0')$
$Q_3 = I E_0$
$Q_4 = I'E_0' (E_1' + E_2)$
$Q_5 = I E_0'E_1E_2$
$Q_6 = I'E_0E_1'E_2$
Output = $I E_0'E_1'E_2 + I' E_0E_1'E_2'$

E_2 \ E_0E_1	00	01	11	10
0	Q_2	Q_0	Q_1	Q_6
1	Q_5	Q_3	x	Q_4

This uses the high-medium-low priority heuristic fairly well.

For the highest priority, on I = 0, the states Q_2, Q_3, and Q_5 have the same output, Q_0 and Q_1 have the same output, and each of these are adjacent. On I = 1, the states Q_1, Q_4, and Q_6 have the same output, and Q_0 and Q_2 have the same output, and each of these is adjacent. Thus for all of the states, the high-priority condition is met.

For medium priority, $Q_3 \rightarrow Q_5$ or Q_4 and both of these are adjacent.

For low priority, on input 0, Q_0, Q_2, Q_3, and Q_5 all output 0, and on input 1, Q_1, Q_6, Q_4 and x all output 0, meeting the low priority conditions.

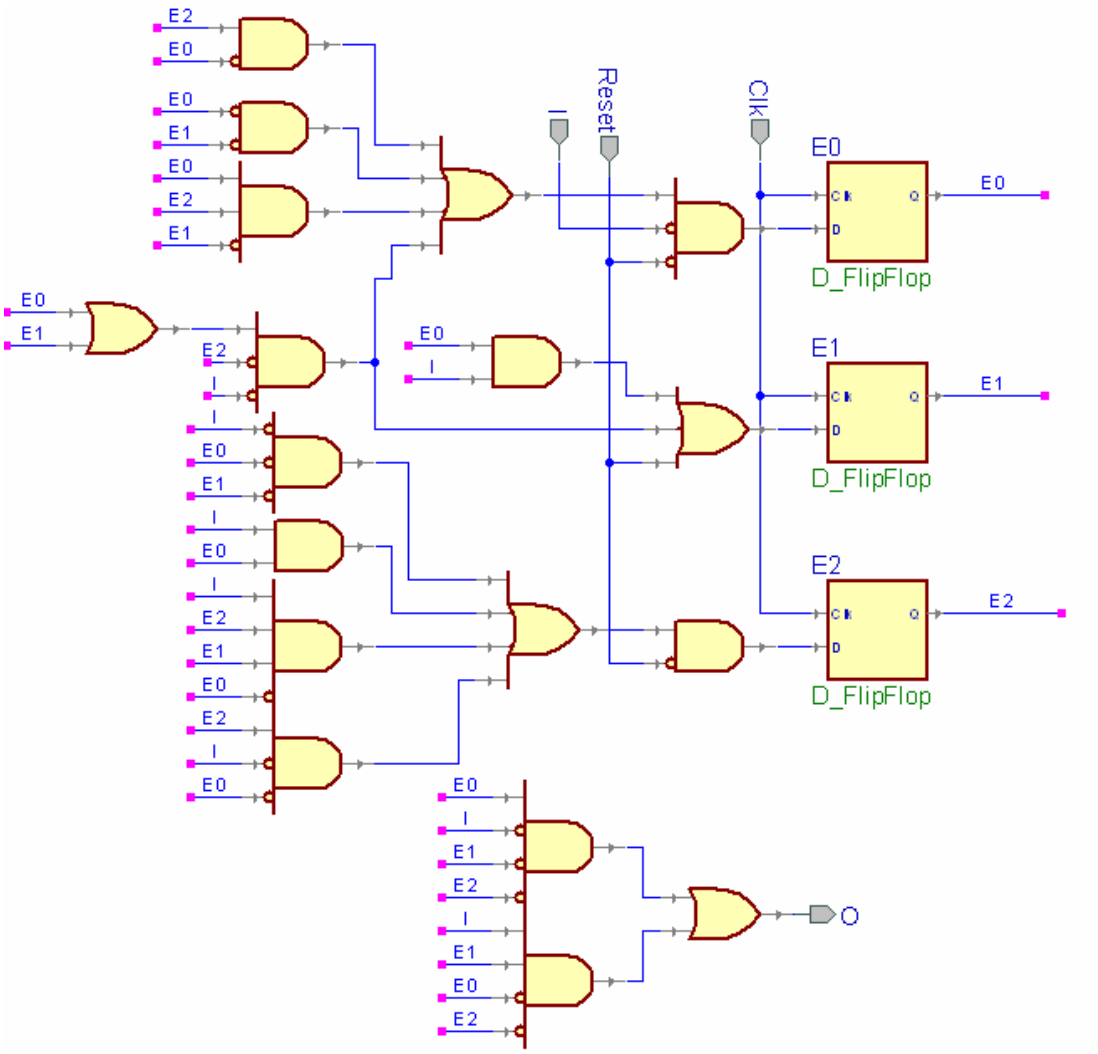
(e) The equations are:

$E_0^* = Reset' I' [E_0E_1'E_2 + E_0'E_1' + E_0'E_2 + E_2' (E_0 + E_1)]$

$E_1^* = Reset + I'E_2' (E_0 + E_1) + I E_0$

$E_2^* = Reset' [I E_0'E_1E_2 + I E_0 + I'E_0'E_1' + I'E_0'E_2]$

Output = $I E_0'E_1'E_2 + I' E_0E_1'E_2'$



(f) Here is the coded implementation:

```
module prob8_26f ( I ,Clk ,Reset ,O );

input I, Clk, Reset;
wire I, Clk, Reset;

output O ;
reg O ;

parameter Q0 = 3'b010;
parameter Q1 = 3'b110;
parameter Q2 = 3'b000;
parameter Q3 = 3'b011;
parameter Q4 = 3'b101;
parameter Q5 = 3'b001;
parameter Q6 = 3'b100;

reg [3:1] state;
```

```
always @(posedge Clk) begin
   O = (!I & (state == Q5)) | (I & (state == Q6));
   case (state)
         Q0:
               begin
                     if (I) state = Q2;
                     else state = Q1;
               end
         Q1:
               begin
                     if (I) state = Q3;
                     else state = Q1;
               end
         Q2:
               begin
                     if (I) state = Q2;
                     else state = Q4;
               end
         Q3:
               begin
                     if (I) state = Q5;
                     else state = Q4;
               end
         Q4:
               begin
                     if (I) state = Q3;
                     else state = Q6;
               end
         Q5:
               begin
                     if (I) state = Q2;
                     else state = Q4;
               end
         Q6:
               begin
                     if (I) state = Q3;
                     else state = Q1;
               end
   endcase

end

endmodule
```

Exercise 9.1

Not yet available.

Exercise 9.2

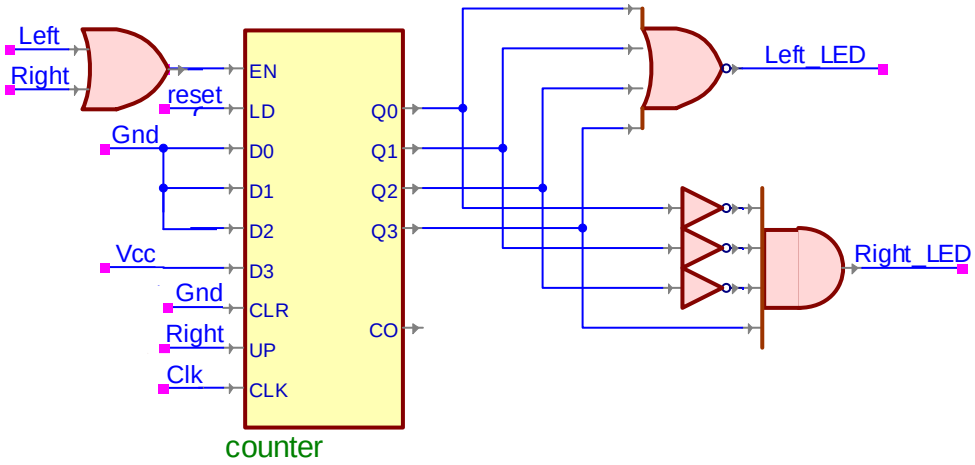
Not yet available.

Exercise 9.3

Not yet available.

Exercise 9.4

Using the counter from Figure 9.5, we can easily construct this left-right LED design. When the system is reset, we need to start out in state nine. This can be accomplished by hard-wiring the load inputs of the counter to 4, which corresponds to the middle of nine states from 0 to 8, and tying the load input to reset. If we assume that either the left or right input will be asserted at once, never at the same time (or if they are, right will take precedence), then we can tie the right signal to the up input of the counter so that when right is asserted it will advance to the next higher state. ORing the left and right inputs and using that as the enable signal of the counter will ensure that the counter only changes when right or left is asserted. Finally, two gates use the outputs of the counter to set the Left_LED and Right_LED as well as disabling the counter from advancing further. The resulting circuit schematic looks as follows:



Exercise 9.5

Not yet available.

Exercise 9.6

Not yet available.

Exercise 9.7

Not yet available.

Exercise 9.8

Not yet available.

Exercise 9.9

Not yet available.

Exercise 9.10

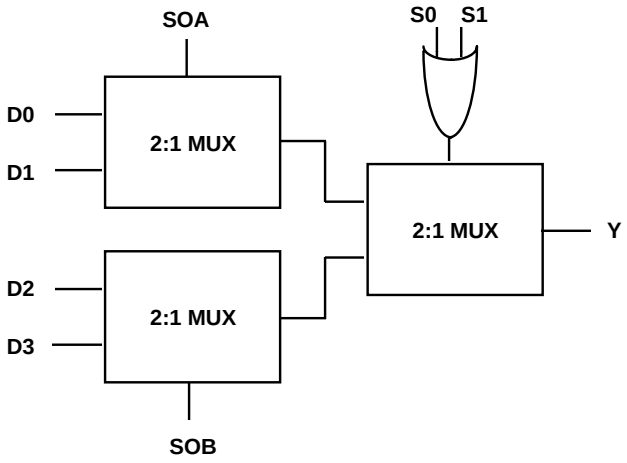
Not yet available.

Exercise 9.11

Not yet available.

Exercise 9.12

Recall that the block diagram of the Actel combinational logic C-cell looks as follows:



(a) Y = AB

D0	D1	SOA	D2	D3	SOB	S0	S1
0	A	B	-	-	-	0	0

(b) Y = (AB)'

D0	D1	SOA	D2	D3	SOB	S0	S1
1	-	0	1	0	B	A	0

(c) Y = (A + B)'

D0	D1	SOA	D2	D3	SOB	S0	S1
1	0	B	0	-	0	A	0

(d) Y = A xor B

D0	D1	SOA	D2	D3	SOB	S0	S1
0	1	B	1	0	B	A	0

(e) Y = AB + C

D0	D1	SOA	D2	D3	SOB	S0	S1
0	A	B	1	-	0	C	0

(f) (A + B) + C

D0	D1	SOA	D2	D3	SOB	S0	S1
0	-	0	B	1	A	C	0

(g) AB + AC + BC

D0	D1	SOA	D2	D3	SOB	S0	S1
0	C	B	C	1	B	A	0

Exercise 9.13

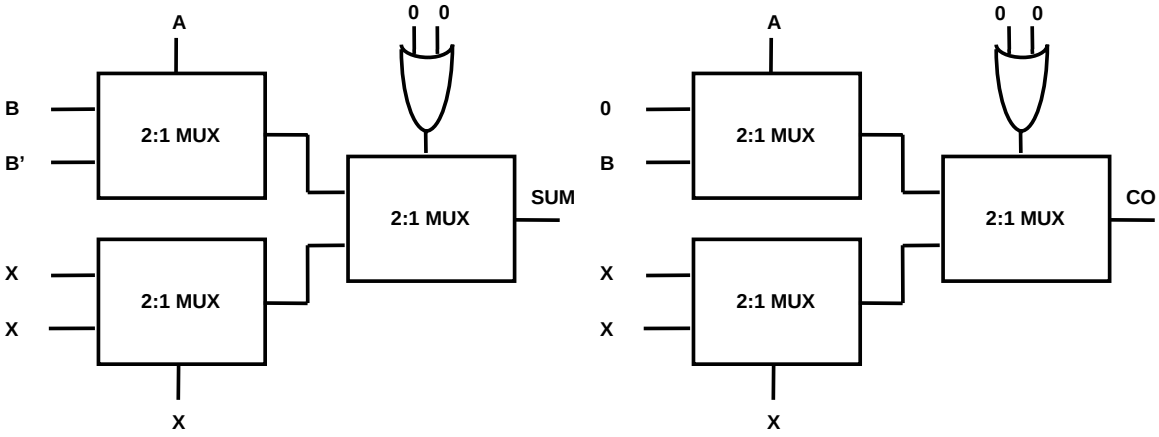
Not yet available.

Exercise 9.14

Half adder Actel Logic Module:

SUM = A'B + AB'

CO = AB

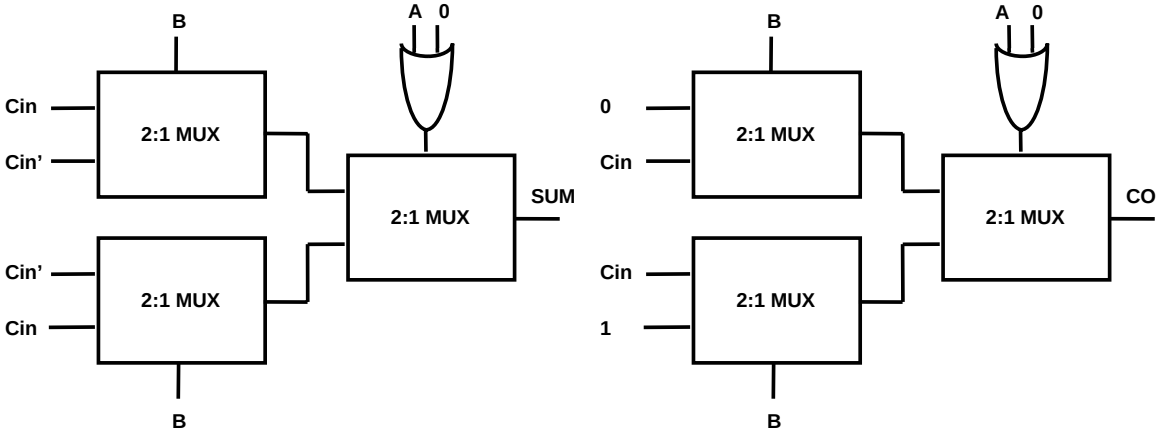


Half Adder

(h) Full adder Actel Logic Module:

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

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



Exercise 9.15

Not yet available.

Exercise 9.16

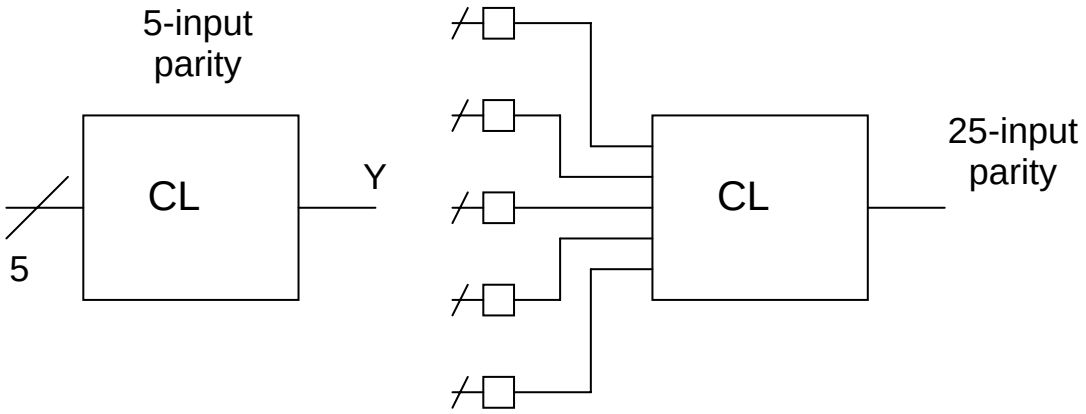
Not yet available.

Exercise 9.17

Not yet available.

Exercise 9.18

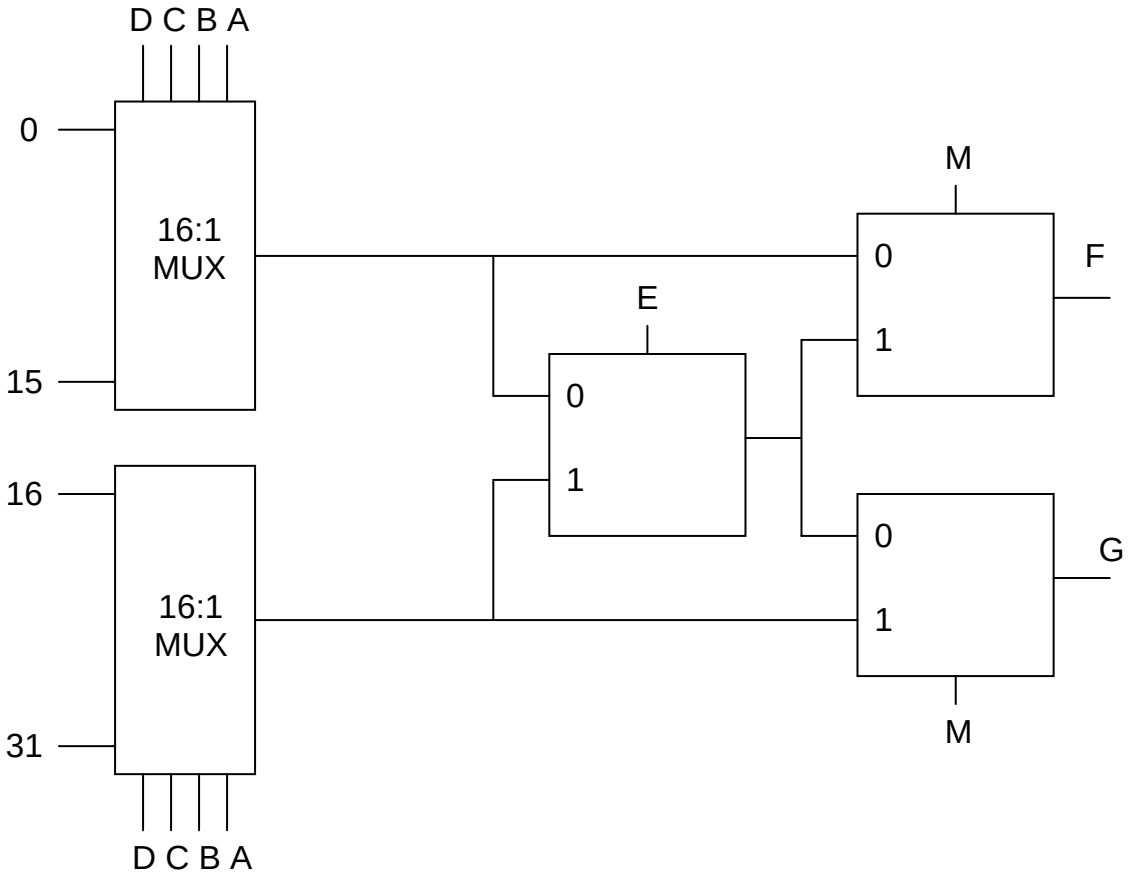
(a) 25-input parity function in 2 level CLB structure:



(b) A 3 level CLB structure can implement a $5^3 = 125$ input parity function.

Exercise 9.19

(a)



(b) F = A xor B xor C xor D xor E

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

(c) F(A,B,C) = A xor B xor C

G(A,B,C) = AB + BC + AC

ABC	AB	BC	AC	F	G
000	0	0	0	0	0
001	0	0	0	1	0
010	0	0	0	1	0
011	0	1	0	0	1
100	0	0	0	1	0
101	0	0	1	0	1
110	1	0	0	0	1
111	1	1	1	1	1

Exercise 9.20

Not yet available.

Exercise 9.21

Not yet available.

Exercise 9.22

Not yet available.

Exercise 9.23

Not yet available.

Exercise 9.24

Not yet available.