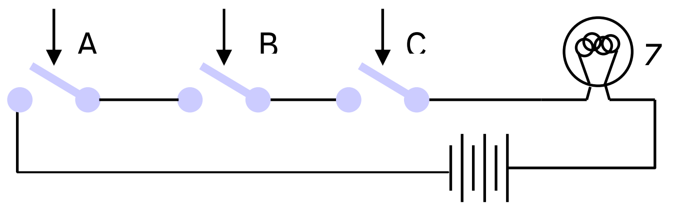
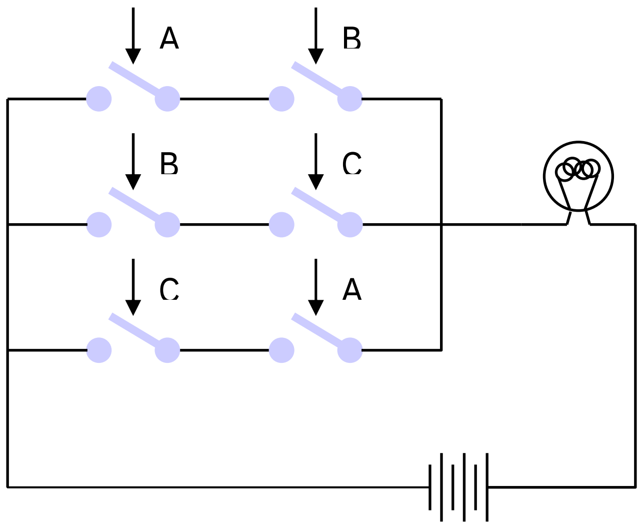
Exercise 1.1

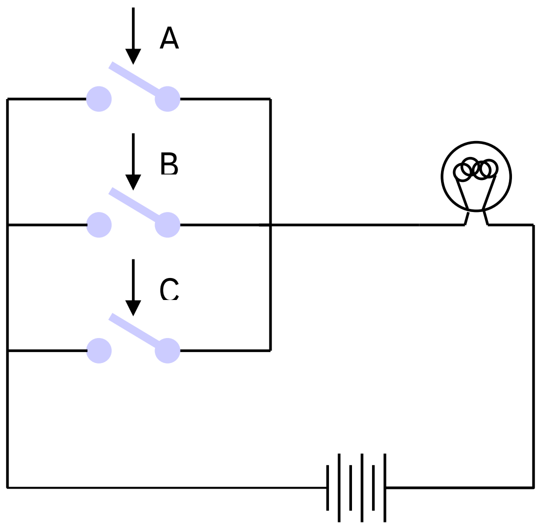
(a) In order for the light bulb to be turned on, all of the switches have to be closed.



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

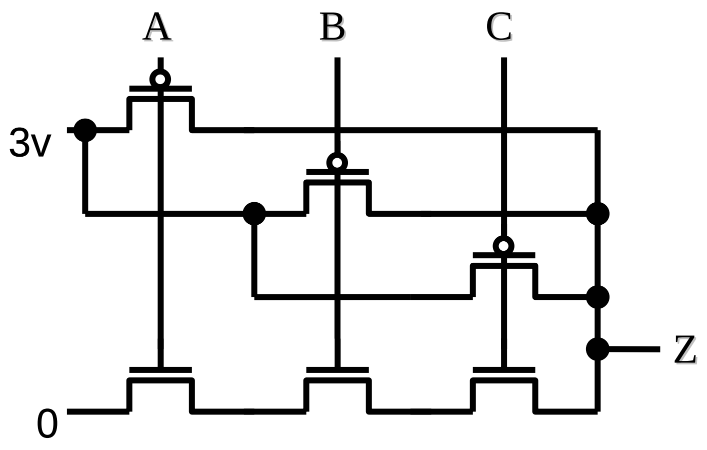


(c) If any one of the switches are closed the light will turn on.



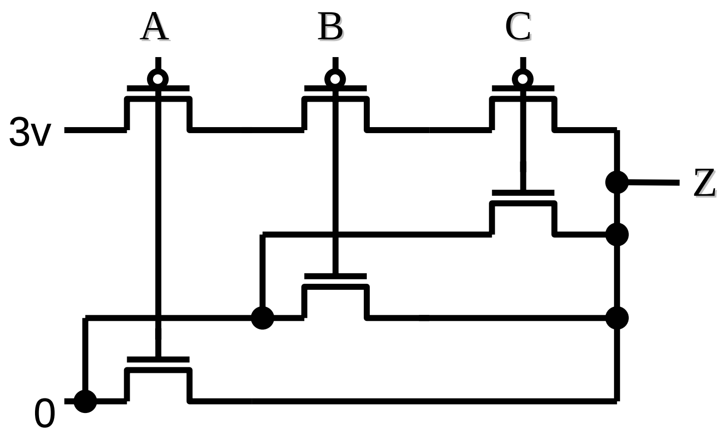
Exercise 1.2

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



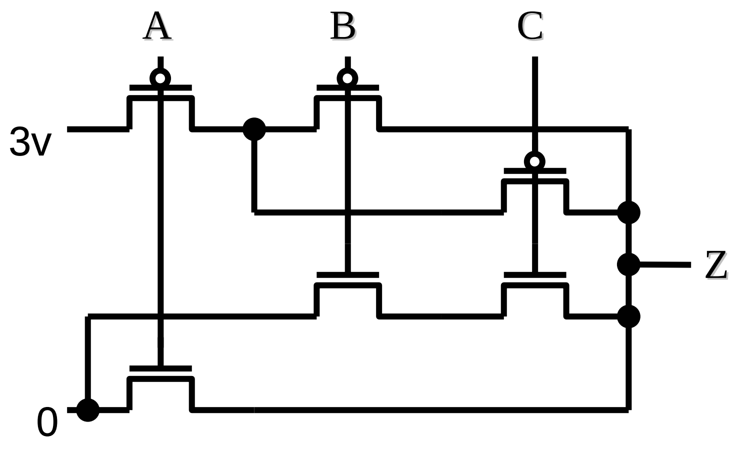
attached to high and three n-type transistors in series attached to low.

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

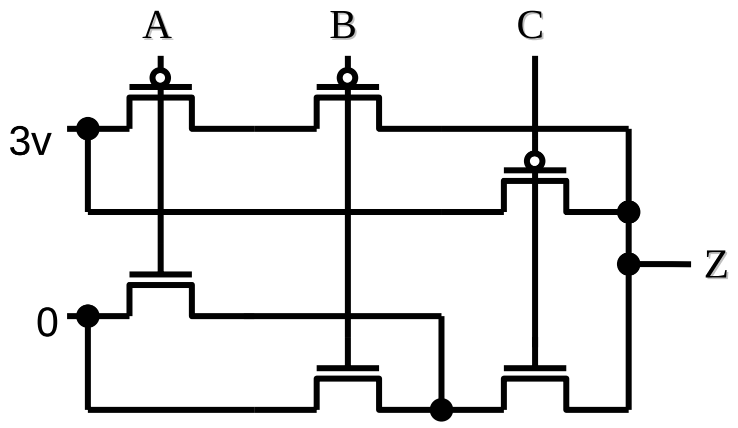


attached to high and three n-type transistors in parallel attached to low.

(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’)

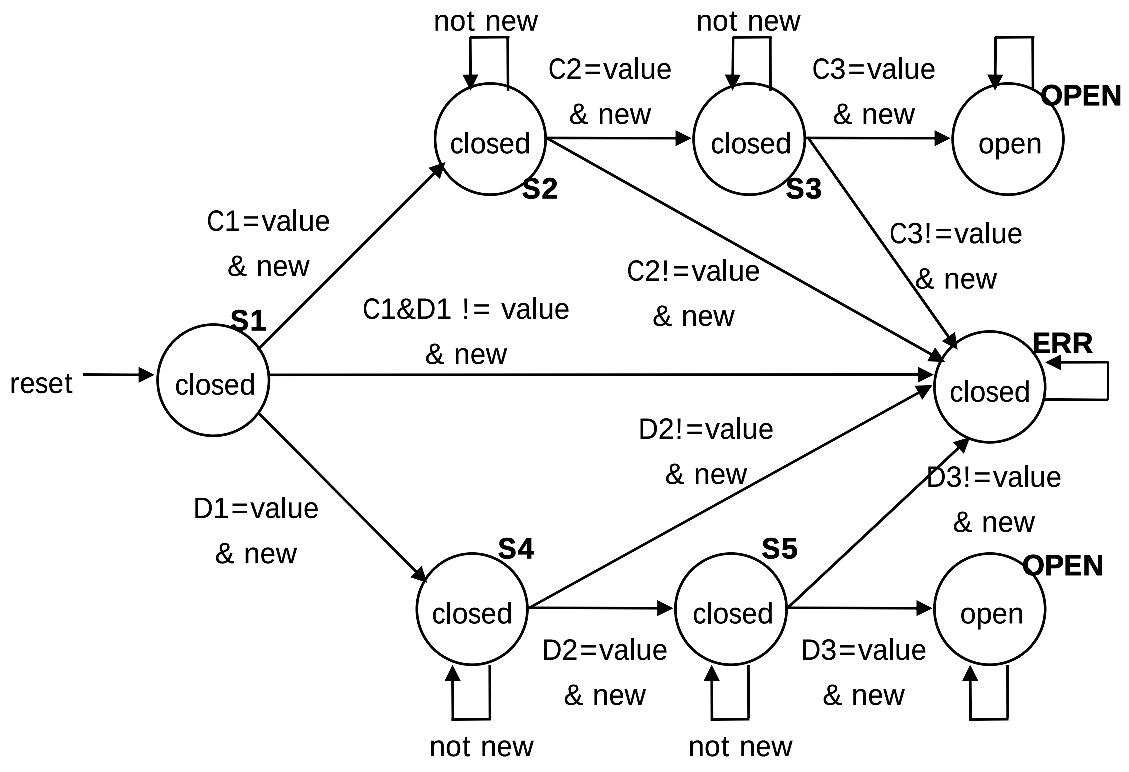
Exercise 1.10

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:

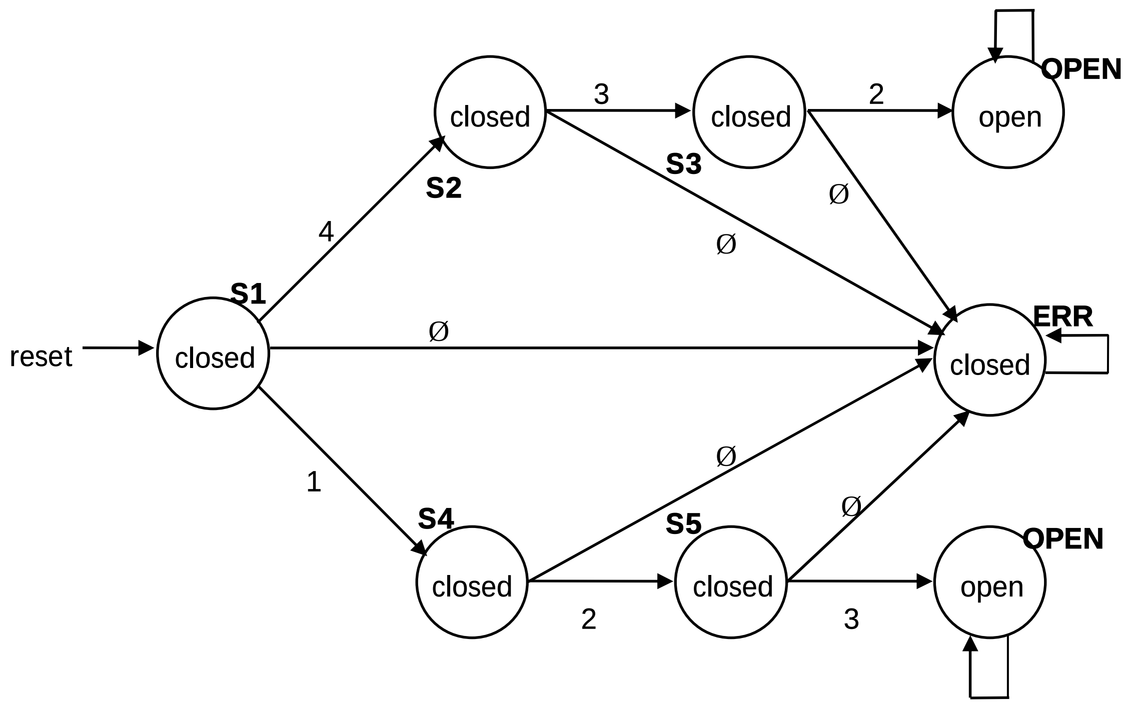
```
integer combination_lock ( ) {
    integer v1, v2, v3;
    integer error = 0;
    static integer c[3] = 3, 4, 2;
    static integer d[3] = 1, 2, 3;

    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:

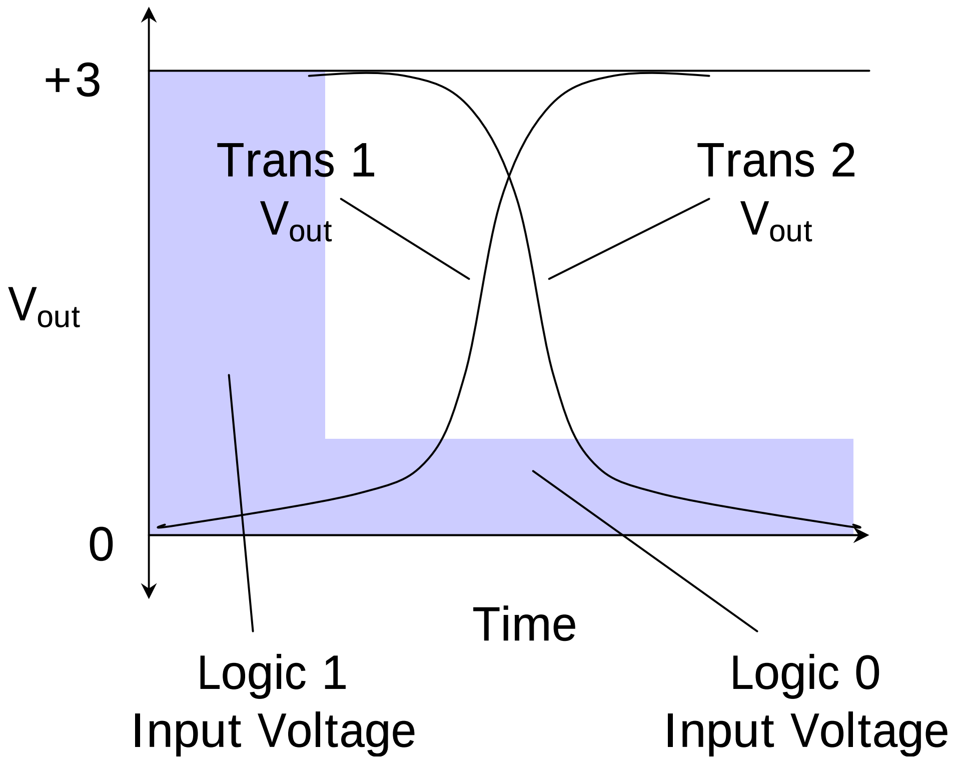
(A and B) or (B and C) or (C and A)

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

A or B or C

<u>Exercise 1.16</u>

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:

$$Z = A \bullet B \bullet C$$

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

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

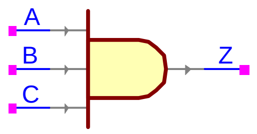
(c) Any one of the control signals is true:

$$Z = A + B + C$$

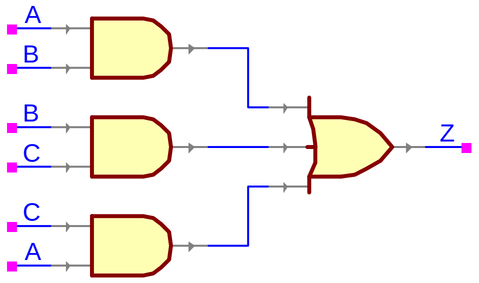
Exercise 1.20

Logic schematics for each of the functions from exercise 1.19

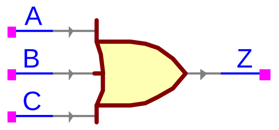
(a) A • B • C:



(b) A • B + B • C + C • A:

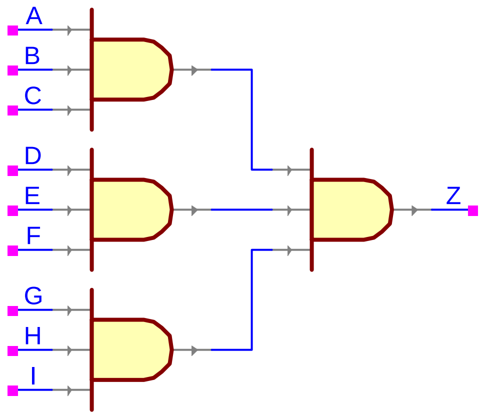


(c) A + B + C:



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.

<u>Exercise 1.27</u>

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) $D_4 \ldots D_2 = 1$

$D_1 = d30 + d31$

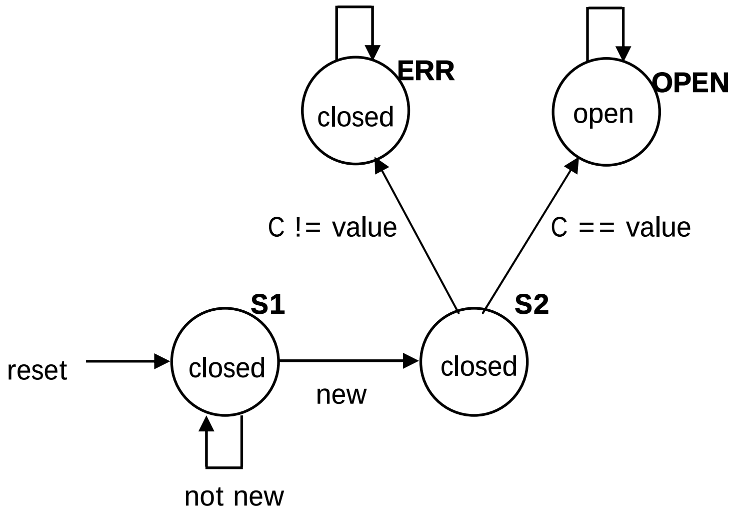
$D_0 = d29 + d31$

(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.

<u>Exercise 1.29</u>

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)

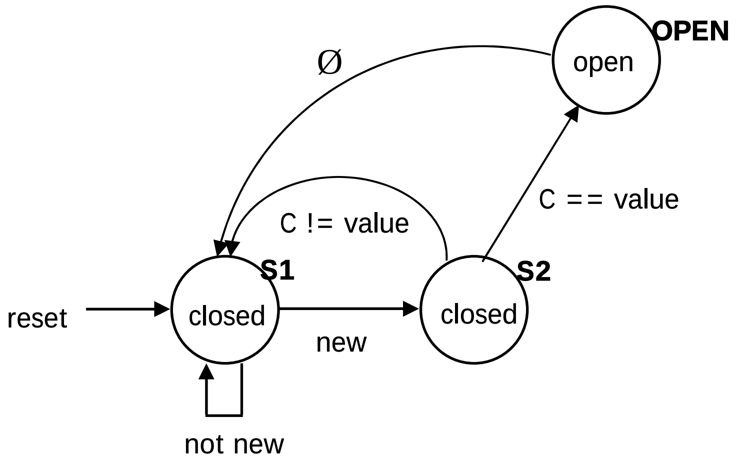
The complexity of the old open function is: 1 4-input AND, 1 2-input AND, and 1 2-input OR. The complexity of the new open function is: 1 3-input AND, 1 2-input AND, and 1 2-input OR.

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:

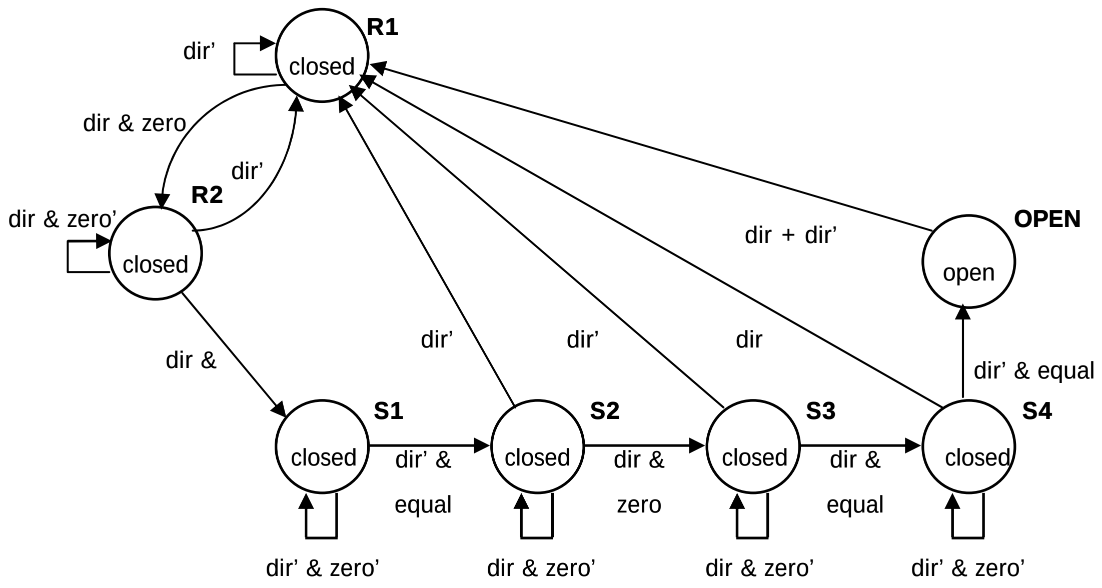


Exercise 1.31

(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

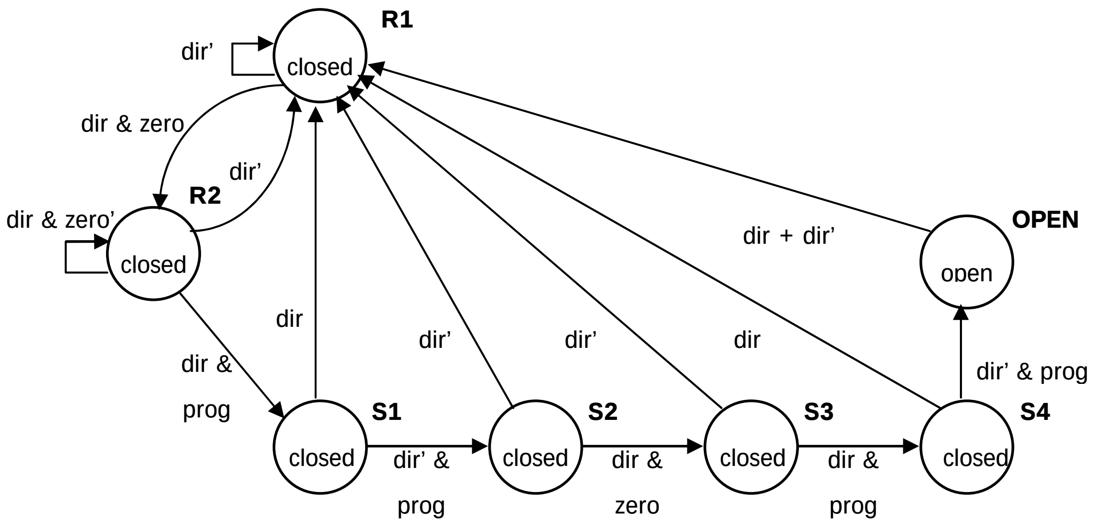
(b) For this system, we are going to need 2 reset states so that we can keep track of how many times we have passed by 0 before we transition into the start state. Since there is no way for the user to tell if he/she has transitioned back to the reset state, it is okay to transition from any state back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


R1
closed
dir'
dir & zero
dir'
R2
dir & zero'
closed
dir
dir'
dir'
dir
dir + dir'
OPEN
open
dir &
prog
dir' & prog
S1
S2
S3
S4
closed
closed
closed
closed
dir' &
prog
dir &
zero
dir &
prog