Lab 2 - BCD Detector

Objectives:

- To understand the Sum-of-Product (SOP) form (To apply the Karnaugh map method to simplify a Boolean equation (To apply "don't care" states to a Karnaugh Map (K-map) (To create a MultiSim simulation of a digital circuit

Your References: Lessons 2, & 3

Components Needed: (quantities unknown)

- 7400 IC – quad two-input NAND gate
- 7402 IC – two-inputr (NOT) gate
- 7408 IC – quad two-input (AND) gate
- 7411 IC – triple three-input AND
- 7432 IC – quad two-input OR gate

Discussion:

The Karnaugh map (K-map) method of simplification is a powerful graphical technique. The advantages of using K-maps for reduction is that it is easier to see when the expression has been fully simplified and the K-map method provides a structured approach to Boolean simplification.

In this experiment, you will design a BCD detector, implement it in MultiSim, simplify the design using a K-map, implement the new designs and do a cost analysis. Start with a truth table for your first design. Think about which 4-input digital numbers make a valid BCD code. Pay special attention to the specifications that follow. We will look at one of this BCD detector as changes are made. The specification for the BCD detector is as follows:

- Components:
  - 7404 IC – quad two-input NAND gate
  - 7404 IC – two inverter (NOT gate)
  - 7408 IC – quad two-input (AND) gate
  - 7432 IC – four-input NAND gate
  - 7432 IC – quad two-input OR gate
- Need to minimize the overall cost of the BCD detector
- Cost breakdown of the components used:
  - Each two-input AND, OR, NAND, NOR, NOT, XOR, ... 1
  - Each other Component (LED, switch, etc) is $0
  - Each individual connection is charged by $1 per wire connection

```
from BasicGateOpts import Signal
from BasicGateOpts import KarnaughMap as Map

# Inputs: Four from binary switches (A, B, C, D) representing the binary number
# Outputs: One LED turns ON to indicate a valid BCD number as an input, turns OFF for an invalid BCD number

import numpy as np
OK = lambda x: np.logical_xor(x, axisparity)
OR = lambda x: np.logical_or(x, axisparity)
NOT = lambda x: np.logical_not(x, axisparity)
NOT = lambda x: np.logical_not(x, axisparity)
Signal(truth_cols = f"A,B,C,D,{f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16}", show_legend =False).display()
```

Truth Table

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

KMAP

	00	01	11	10
	CD	CD	CD	CD
00 AB	1	1	1	1
01 AB	1	1	1	1
11 AB	0	0	0	0
10 AB	1	1	0	0

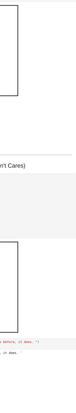
KMAP Legend

	00	01	11	10
	CD	CD	CD	CD
00 AB	1	1	1	2
01 AB	4	5	6	7
11 AB	12	13	14	15
10 AB	8	9	10	11

[1.0] First Prototype (SOP from TT):

$A \cdot B' \cdot C' + A \cdot B' \cdot D + A \cdot B \cdot C + A' \cdot B' \cdot D + A' \cdot B \cdot C' + A' \cdot B \cdot D + A' \cdot B' \cdot C + A' \cdot B' \cdot C'$

```
from BasicGateOpts import LogicFlatter
lp = LogicFlatter()
circuit = {
    'assign': {
        'F': (
            '+',
            ('*', ('*', 'A', 'B'), 'C'),
            ('*', ('*', 'A', 'B'), ('~', 'D')),
            ('*', ('*', ('~', 'A'), ('~', 'B')), ('~', 'D')),
            ('*', ('*', ('~', 'A'), ('~', 'B')), 'C'),
            ('*', ('*', ('~', 'A'), 'B'), ('~', 'D')),
            ('*', ('*', ('~', 'A'), 'B'), 'C'),
            ('*', ('*', ('~', 'A'), ('~', 'B')), ('~', 'C'))
        )
    }
}
lp.make_logic(circuit,IN_ON_FOUR,2500)
```



From the diagram, we need:

- 25 7404 ICs (inv gates) = US$
- 5 7408 ICs (and gates) = ceil(25x two-input and gates)/4 gates per IC) = 208
- 15 7432 ICs (OR gates) = 45$
- 4 switched to control the system = $9
- 1 LED = $0
- 40 x $1 = 25 wires = 75$

Total BOM cost = 2455

[2.3] First Reuction: (using only AND, OR, NOT)

$A + B \cdot C$

```
lp = LogicFlatter()
circuit = {
    'assign': {
        'F': (
            '+',
            ('*', ('~', 'A')),
            ('*', 'B', 'C'),
            ('*', ('~', 'B'), ('~', 'C'))
        )
    }
}
lp.make_logic(circuit)
```

From the diagram, we need:

- 2 7404 ICs (inv gates) = US$
- 1 7408 ICs (and gates) = ceil(2x two-input and gates)/4 gates per IC) = 20$
- 15 7432 ICs (OR gates) = 45$
- 4 switched to control the system = $9
- 1 LED = $0
- 8 x $1 = 25 wires = 75$

Total BOM cost = 201

[4.1] NAND only Reduction (Bubble Method)(No Don't Cares)

```
from BasicGateOpts import LogicFlatter
lp = LogicFlatter()
circuit = {
    'assign': {
        'F': (
            '+',
            ('*', 'A'),
            ('*', 'B', 'C')
        )
    }
}
lp.make_logic(circuit)
```

```
display("Verifying that the reduced circuit has the same truth table as before, it does.")
Signal(f1_expression='(A) + (B & C)').display()
```

Verifying that the reduced circuit has the same truth table as before, it does.

Truth Table

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

KMAP

	00	01	11	10
	CD	CD	CD	CD
00 AB	1	1	1	1
01 AB	1	1	1	1
11 AB	0	0	0	0
10 AB	1	1	0	0

KMAP Legend

	00	01	11	10
	CD	CD	CD	CD
00 AB	1	1	1	2
01 AB	4	5	6	7
11 AB	12	13	14	15
10 AB	8	9	10	11

Final Part cost:

- 1 7420 ICs (nand gates) = 4$
- 1 7432 IC (or gate) = 4$
- 3 switches to control the system = $9
- 1 LED = $0
- 5 wires = $5

Total reduced BOM cost= 21$

Therefore, there was a 9.5x cost reduction

[4.2] Don't Care Reduction

See above KMAP legend, setting [1] and [2] to don't care would result in the following groups:

(0,1,2,3,4,5,6,7) = A'

(0,1,2,3,8,9,10,11) = B'

Therefore, the final reduction is A' + B'.

```
from BasicGateOpts import LogicFlatter
lp = LogicFlatter()
circuit = {
    'assign': {
        'F': (
            '+',
            ('~', 'A'), ('~', 'B')
        )
    }
}
lp.make_logic(circuit)
```


Final Part cost:

- 2 7404 ICs (inv gates) = 4$
- 1 7432 IC (or gate) = 4$
- 1 switch to control the system = $3
- 1 LED = $0
- 5 wires = $5

Total reduced BOM cost: 19$

[4.4] NAND only Don't Care Reduction

Convert the don't care logic diagram to use only NAND gates.

Note: I used the demorgans law here

```
lp = LogicFlatter()
circuit = {
    'assign': {
        'F': (
            '+',
            ('~', 'A'), ('~', 'B')
        )
    }
}
lp.make_logic(circuit)
```


Final Part cost :

- 1 7420 ICs (nand gates) = 4$
- 2 switches to control the system = 6$
- 1 led =2$
- 5 wires = 5$

Total reduced BOM cost: 13$

```
display("Note: DC and Ci systems are don't cares. The truth table is equivalent to the starting circuit so this is valid")
Signal(f1_expression='not ( A and B)').display()
```

Note: DC and Ci systems are don't cares. The truth table is equivalent to the starting circuit so this is valid

Note, the final circuit can reduce to a single nand gate:

Truth Table

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

KMAP

	00	01	11	10
	CD	CD	CD	CD
00 AB	1	1	1	1
01 AB	1	1	1	1
11 AB	0	0	0	0
10 AB	1	1	0	0

KMAP Legend

	00	01	11	10
	CD	CD	CD	CD
00 AB	1	1	1	2
01 AB	4	5	6	7
11 AB	12	13	14	15
10 AB	8	9	10	11

[R] Results

[R.1] "Why isn't Input D needed in the BCD detector design?" :

Because D was factored out during minimum SOP reduction. In the Don't care reduction, C is also eliminated

[R.2] Cost Table

```
import pandas as pd

pd.DataFrame({
    'Design': ['Prototype', 'First Cut', 'Dup Cost', 'DNL', 'Connection Cost', 'TS', 'Total Cost', 1#1],
    'Prototype': [ 25, 15, 4, 1, 25, 9, 2455, 1#1],
    'First Cut': [ 2, 15, 4, 1, 8, 9, 201, 1#1],
    'Dup Cost': [ 4, 15, 4, 1, 5, 9, 21, 1#1],
    'DNL': [ 4, 15, 4, 1, 5, 9, 19, 1#1],
    'Connection Cost': [ 4, 15, 4, 1, 5, 9, 13, 1#1],
    'Total Cost': [ 10, 1#1, 1#1, 1#1, 1#1, 1#1, 1#1, 1#1],
})
```

Design	Prototype	First Cut	Dup Cost	DNL	Connection Cost	TS	Total Cost	1#1
Prototype	25	15	4	1	25	9	2455	1#1
First Cut	2	15	4	1	8	9	201	1#1
Dup Cost	4	15	4	1	5	9	21	1#1
DNL	4	15	4	1	5	9	19	1#1
Connection Cost	4	15	4	1	5	9	13	1#1
Total Cost	10	1#1	1#1	1#1	1#1	1#1	1#1	1#1