CS2100 - L18 - Simplification Week 10 + 11

18.1 - Function simplification (algebraic simplification)

18.2 - Karnaugh maps (K-maps)

- Terminology
- Algorithm
- 2 variables
- 3 variables
- 4 variables
- Simplified POS expression
- "Don't care" conditions

18.1 - Function simplification (algebraic simplification)

Function Simplification

- Why simplify?
  - Simpler expression uses fewer logic gates.
  - Thus cheaper, uses less power, (sometimes) faster
- Techniques
  - **Algebraic**
    - Using theorems
    - Open-ended; requires skills
  - **Karnaugh Maps (K-Map)**
    - Easy to use
    - Limited to no more than 6 variables
  - **Quine-McCluskey** (non-examinable)
    - Suitable for automation
    - Can handle many variables (but computationally intensive)

Algebraic Simplification

- Aims to minimise
  - Number of literals, and
  - Number of terms
- Sometimes the two aims conflict, so let's aim at reducing the number of literals in this section
- **Difficult** – needs good algebraic manipulation skills

18.2 - Karnaugh maps (K-maps)

Terminology

Terminology: **I, PI and EPI**

Implicant:

- A product term that could be used to cover several minterms of the function

Prime Implicant:

- The maximal (largest) possible implicant for a group of minterms

Essential Prime Implicant:

- Prime Implicant that contains 1 or more **unique minterm**

Algorithm

Simplification Algorithm Rephrased

1. Draw the ***Prime Implicant(s)*** for each minterm in the K-Map

2. Using (1), take all ***Essential Prime Implicants***

3. Choose the smallest collection of ***Prime Implicant*** for the rest of minterms not covered in (2)

EX: **Generalize Observation**

- A group of size 1 can remove __0__ literals
- A group of size 2 can remove __1__ literals
- ……<use your power of deduction>……
- So, a group of size X can remove __$\log_2 X$__ literals

2^k k

2 variables

2-Variables K-Map

x	y	F
0	0	m0
0	1	m1
1	0	m2
1	1	m3

		y
	x'.y'	x'.y
x	x.y'	x.y

		y
	m0	m1
x	m2	m3

- Other layouts are possible:
  - e.g. swap X and Y, reverse the rows / columns etc
- We will use this one due to its easily remembered association to the truth table entries

3 variables

3-Variables K-Map

- There are 8 squares in a 3-variable K-map

a	b	c	F
0	0	0	m0
0	0	1	m1
0	1	0	m2
0	1	1	m3
1	0	0	m4
1	0	1	m5
1	1	0	m6
1	1	1	m7

∀ squares, each square has a neighbour with exactly 1 complement

b=1, c=1

a \ bc	00	01	11	10
0	000 a'.b'.c'	001 a'.b'.c	011 a'.b.c	010 a'.b.c'
1 (a)	100 a.b'.c'	101 a.b'.c	111 a.b.c	110 a.b.c'

b: columns 11, 10; c: columns 01, 11

a \ bc	00	01	11	10
0	m0	m1	m3	m2
1 (a)	m4	m5	m7	m6

b: columns 11, 10; c: columns 01, 11

1	1		1
			1
			1
1	1		1

All 4 corners are considered one group

4 variables

4-Variables K-Map

- 16 square cells in a 4-variable K-map:
  - Each cell has **4 neighbours**
  - Note the wraps around along the edges

∀ squares, each square has a neighbour with exactly 1 complement

$F(\underbrace{w, x}_{\text{horizontal}}, \overbrace{y, z}^{\text{vertical}})$

y

wx \ yz	00	01	11	10
00	***m0***	*m1*	*m3*	*m2*
01	*m4*	*m5*	*m7*	*m6*
11	*m12*	*m13*	*m15*	*m14*
10	*m8*	*m9*	*m11*	*m10*

x

w

z

Can you identify the 4 neighbours of minterm **m0**?

4-Var K-Map: Alternative Placement

- Beware that there is another popular 4 variable K-Map placements:

w

yz \ wx	00	01	11	10
00	*m0*	*m4*	*m12*	*m8*
01	*m1*	*m5*	*m13*	*m9*
11	*m3*	*m7*	*m15*	*m11*
10	*m2*	*m6*	*m14*	*m10*

z

y

x

Can you spot the **differences**?

Simplified POS expression

Simplified POS Expression

- To find the Simplified POS expression of a function F:
  1. Find the Simplified **SOP expression** for **the F'**
  2. Negate (1) to get simplified **POS expression of F**

- Find the simplified POS for F = Σm(0,2,4,5,6,8,10,12,13,14)

K-map of F

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

K-map of F'

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

SOP of **F' = B'·D + C·D** ➔ POS of **F =** ________________

"Don't care" conditions

Don't Care Condition (1/3)

- In certain problems, some outputs are not specified or are invalid
- Hence, these outputs can be either '1' or '0'
- They are called don't-care conditions, denoted by **X** (or sometimes, **d**)
- Example: **ABCD** represents a digit (0..9). Function **P(A,B,C,D)** counts whether there are **even** number of '1's in the binary representation
  - Input (1010…1111) are invalid and we don't care about the output for them

A	B	C	D	P
0	0	0	0	**1**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**1**
0	1	0	0	**0**
0	1	0	1	**1**
0	1	1	0	**1**
0	1	1	1	**0**
1	0	0	0	**0**
1	0	0	1	**1**
1	0	1	0	**X**
1	0	1	1	**X**
1	1	0	0	**X**
1	1	0	1	**X**
1	1	1	0	**X**
1	1	1	1	**X**

Don't Care Condition (2/3)

- Don't-care conditions can be used to help simplify Boolean expression further in K-maps
  - Could be chosen to be **either** '1' or '0', depending on which choice **results in a simpler expression**
- We can use the notation **Σd** to denote the set of don't-care minterms.
  - e.g., the function **P** in the previous slide can be written as:

$$P = \Sigma m(0, 3, 5, 6, 9) + \Sigma d(10, 11, 12, 13, 14, 15)$$