Week 3: More Advanced Combinatorial Methods

In Week 1 we looked at some general circuit concepts and in week 2 we looked at some techniques for analysing circuits that operate according to the rules of binary logic. In particular we saw how to write out the truth table for a specified function and how to construct the canonical logical form for the logic.

This week we will finish our review of combinatorial logic techniques and look at a couple of simple applications:

- Using Karnaugh Maps for minimising logic
- Implementing logic as NAND or NOR only
- Binary arithmetic
- Binary Codes

Next week we will move on to look at *sequential* logic circuits, i.e. circuits with an output that depends not only on the current state of its inputs but also on the previous states.

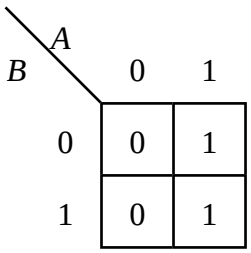
3.1 Using Karnaugh Maps for Minimisation

A Karnaugh map is a way to write out the truth table for a circuit which highlights ways in which the circuit may be simplified. This is important because it allows one to construct circuits using the minimum number of gates. This might also mean minimum cost, or minimum chance of an error, or minimum time delay and so it is a matter of some importance. We will follow the Karnaugh analysis for circuits with one output and up to 4 inputs (i.e. one function of 4 variables).

3.1.1 Constructing K maps

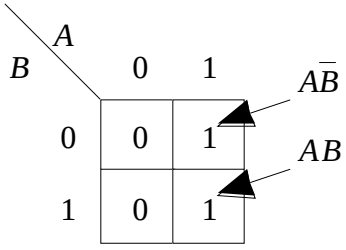
2-variable Karnaugh maps

Consider the canonical expression $F = A\overline{B} + AB$. With a little thought one can see that the variable B is redundant, i.e. its value doesn't affect F. A Karnaugh Map for this function is drawn below

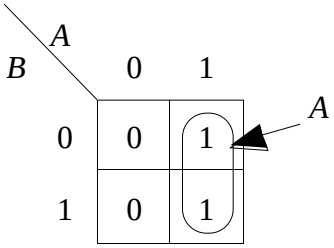


This is like a 2-dimensional histogram with the possible values of the variables listed along the two axes and the result of the function written in the boxes at their intersections.

The two 1's in the diagram correspond to the terms $A\overline{B}$ and AB in the sense that they pick out functions which are *true* when (A is *true* AND $\overline{B}$ is *true*) and when (A is *true* AND B is *true*), which is the same as the "coordinates" of these states on the diagram, as shown below

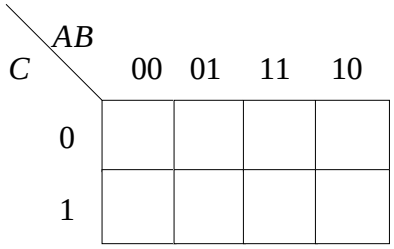


Notice that within this map, moving vertically or horizontally by one square, the variables along the sides change such that just one variable at a time is complemented, i.e. $A \leftrightarrow \overline{A}$ or $B \leftrightarrow \overline{B}$. Clearly if changing the value of a variable makes no difference to the result of an expression then that variable must be redundant. To show this one "loops together" the adjacent 1's on the map as below.



i.e. B is the redundant variable. The looping in this case makes clear that the circuit output F is just given by $F = A$. Clearly this is cheaper and easier to implement than the canonical form $F = A\overline{B} + AB$. We can show the same thing using the Boolean algebra theorems:

$$F = A\overline{B} + AB = A(\overline{B} + B) = A \cdot 1 = A$$



3-variable Karnaugh maps

It is with 3 and 4 variables that K-maps come into their own, allowing one to spot redundancy with relative ease. For three inputs A, B and C one can draw a K-map as shown right. Notice that along the top, the ordering of the AB states is such that only one variable at a time changes: first B then A then B again. The function we used earlier $F = \overline{A}BC + A\overline{B}C + AB\overline{C}$ can be drawn on this map as follows

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

Notice how each minterm corresponds to a 1 on the map at the coordinates where each factor within the minterm is taken as *true*, so the first minterm $\overline{A}BC$ puts a 1 in the box at 011, or (01)(1), since when $\overline{A}$ is *true* A is *false* (0).

We can see that there are no redundant variables on this map i.e. no loops are possible between adjacent squares containing a 1, and so no minimisation of this form is possible. (However see next week's notes about the XOR function). "Adjacent" can mean immediately next to each other vertically or horizontally or via the boundaries, so that square 101 is adjacent to square 001 on the above map because only one variable changes in moving between them.

Exercise 3.1: Draw a K-map for the function $F = AB\overline{C} + \overline{A}\overline{B}C + ABC + A\overline{B}C$ and show that the function can be minimised to $F = AB + \overline{B}C$

Exercise 3.2: Draw a K-map for the function $F = \overline{A}B\overline{C} + AB\overline{C} + ABC$ and show that the function can be minimised to $F = AB + B\overline{C}$

Exercise 3.3: Minimise the two functions whose K-Maps are shown below

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

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

Exercise 3.4: Verify each of the minimisations in Exercises 3.1 to 3.3 directly using the *Crocodile Clips* simulator.

Note: the greatest minimisation is achieved when using the fewest loops (since each loop corresponds to a term in the final expression). So you should make each loop as large as possible, simultaneously encompassing as many redundant variables as possible. It is permissible for loops to overlap each other. See the next section for the full set of rules.

4-variable Karnaugh maps

By now you should be getting the hang of things. A 4-variable *K*-map is shown below.

CD \ AB	00	01	11	10
00				
01				
11				
10				

The looping rules can be summarised as:

- Only adjacent squares can be looped, but that includes end squares.
- Terms can only be looped in groups of 2^N where N is an integer (i.e. 2s, 4s, 8s...).
- The term resulting from a grouping of 2^N has N less variables than the term it groups.
- Loops must only be rectangular, so when 4 squares are looped they must be a line of 4 or a square of 4, but watch out for the corners! – they can be linked too.
- The resulting function must account for all 1's whether they be looped or not.

Exercise 3.4 Minimise the four functions whose K-Maps are shown below. For each function write out both the canonical form and the minimised function. For the two lower functions verify each of the minimisations directly using the *Crocodile Clips* simulator.

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

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

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

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

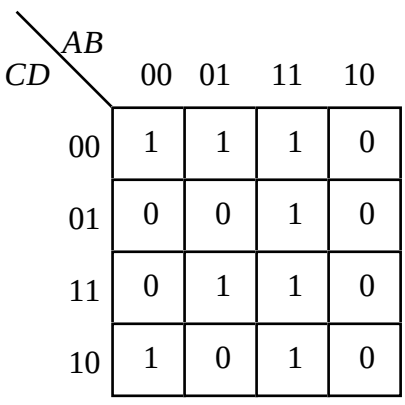
3.1.2 Basic K-map technique

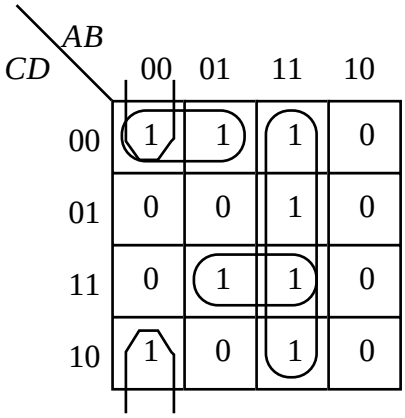
A Karnaugh map or *K*-map is a way to write out the truth table for a circuit which highlights ways in which the circuit may be simplified. The inputs to the circuit are shown along the sides of the map and the circuit output states are specified on the map squares. Importantly the input states must be written in an order such that only a single variable at a time changes. Then, if the circuit outputs a 1 for either value of a variable, it makes no difference if it is changed and that variable must be redundant for that minterm.

Such redundant variable are marked on a *K*-map by looping them together and then identifying the logic that selects that group. Clearly the bigger the group the simpler the logic. For example, for the *K*-map above we could simplify the logic as shown. This predicts that

$$F = AB + \overline{A}\,\overline{C}\,\overline{D} + BCD + \overline{A}\,\overline{B}\,\overline{D}$$

which is considerably simpler than the canonical expression containing 8 minterms each with 4 variables. Getting the general idea of looping is fairly straightforward, but ensuring that one has the minimum logic is matter of some skill. Below we look at some subtleties of the *K*-mapping technique.





3.1.3 "Can't Happen" states

In some applications, certain possible input patterns may never occur in practice for some reliable external reason. For example, suppose a circuit has 4 inputs *A*, *B*, *C*, and *D*, which represent in binary the numbers 0 to 9. Suppose also that for some reason it was not possible for the codes corresponding to the number 10 to 15 to occur. (This is so with a code known as *Binary Coded Decimal* or *BCD*.) In this case we do not care if the logic that we write to deal with these particular inputs produces a logical 1 or a logical 0. Why don't we care? Because these inputs will never occur.

Conventionally one marks such states on a *K*-map with an *X*, and we can optionally include these states in any loop if it will help to simplify our logic, i.e. we can take them as 1's or 0's at our convenience.

Example

Consider the *K*-map shown above but suppose that the inputs *AB* = 10 never occur in practice. In this case we could write the *K*-map as shown with *X*s in the *AB* = 10 column. We could now use a new looping which includes a larger loop and hence a term with fewer variables. (Remember the looping rules: 2^N in squares or rectangles)

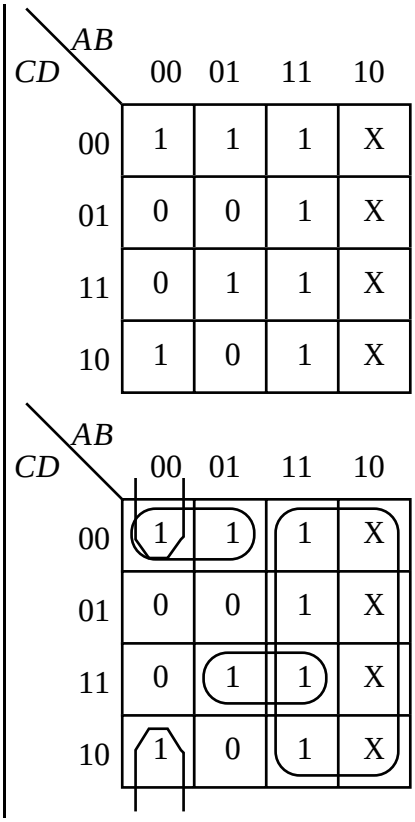
In this case the logic simplifies to

$$F = A + \overline{A}\,\overline{C}\,\overline{D} + BCD + \overline{A}\,\overline{B}\,\overline{D}$$

which may be compared with the previous version

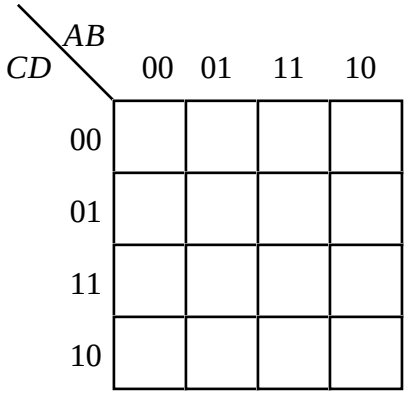
$$F = AB + \overline{A}\,\overline{C}\,\overline{D} + BCD + \overline{A}\,\overline{B}\,\overline{D}$$

Notice that the new version has saved one AND gate.



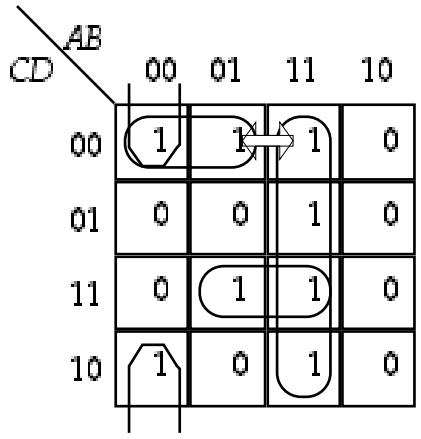
Exercise 3.5 As out lined above, suppose a circuit has 4 inputs *A*, *B*, *C*, and *D*, which represent in binary the numbers 0 to 9. The numbers 10 to 15 do not occur. Use a *K*-map to design logic which will produce an output of 1 when and only when the input is 4, 5 or 6.

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

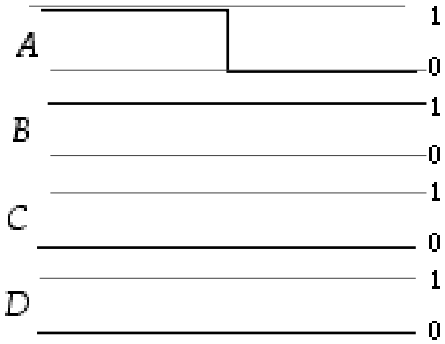


(Hint: Remember the procedure:
Truth Table ⇒ *K*-map ⇒ Logic)

3.1.4 Pulse Trains and Static Hazards



The inputs to logic circuits will obviously change with time and cause the outputs of the circuits to switch also. A *static hazard* is the term used to describe the possibility that in switching between two desired states an undesired state may occur transiently. Recall that as the inputs to a circuit change the circuit will jump between its possible output states. For example, on the *K*-map shown right, changing the input from $ABCD = 0100$ to 1100 will cause the circuit to switch as shown. The inputs to the circuit may be represented as a function of time as shown. Notice that both states should still give an output of 1, but that the 1 derives from two different terms in the expression for the logic, $F = AB + \overline{A}\,\overline{C}\,\overline{D} + BCD + \overline{A}B\overline{D}$

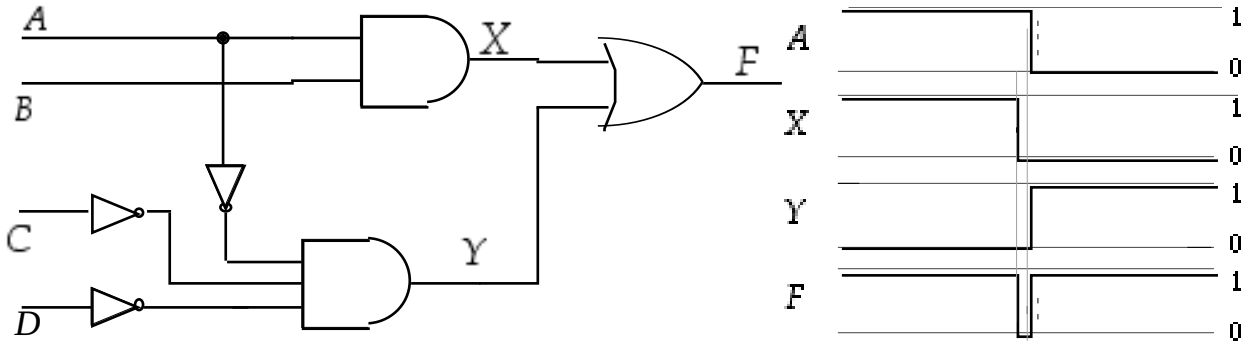


Initially the active term

$F = AB + \overline{A}\,\overline{C}\,\overline{D} + BCD + \overline{A}B\overline{D}$

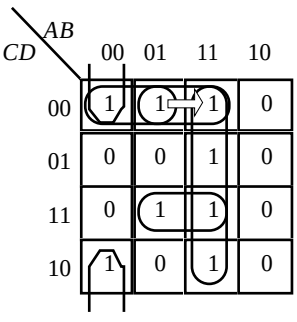
Transition makes this the active term

If we examine the partial logic for the first two terms…



Then we see that as *A* makes a transition from 1 to 0 the output of the first AND gate *X* goes from 1 to 0, while the output of the second AND gate *Y* goes from 0 to 1. Depending on the delay times of the logic it is possible for both inputs to the OR gate to fall transiently to zero, and hence the output *F* may fall to zero. Even though the spurious state may exist for only a few nanoseconds, in some logic applications the consequences will be significant.

The difficulties of static hazards may be countered by adding in logic which is in a strict sense "superfluous" but which prevents dangerous transient states being entered into. For example, in this case we can add an extra loop on the *K* map as shown (surrounding the arrow ⇐). The OR gate partial logic for this additional loop is $B\overline{C}\,\overline{D}$. The final OR gate now always has at least one 1 input and so is in no danger of transiently giving 0 output.



3.1.5 XOR logic on a *K*-map

Some of you may now be becoming adept at spotting logical patterns on a *K*-map. This section is just to point out that XOR logic yields a particular pattern on a *K*-map. The pattern to look for is a pair of diagonally linked 1 outputs. Clearly one cannot loop these in the conventional manner, but using XOR logic does succinctly provide the desired result. The logic shown in this K-map can be achieved with two XOR gates and an AND: $(B \oplus D) \cdot (A \oplus C)$

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

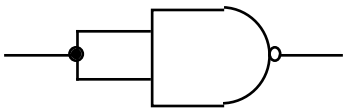
3.2 Using just one type of logic NAND and NOR

Frequently it is required to implement logical functions using just a single type of logic gate, most commonly NAND or NOR. Last week many of you did this without bidding in the exercises. The reasons for implementing logic using only one type of gate is generally practical: its easier to stock (or to manufacture) only type of gate and to build up circuits with identical "building blocks". This technique is illustrated in the Exercise below.

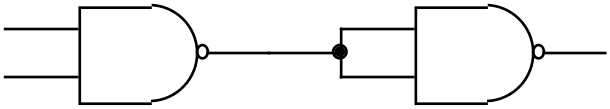
Exercise 3.6

The NOT, AND and OR functions are shown here using only NAND gates. Verify that the operation of the circuits shown is correct by constructing their truth tables.

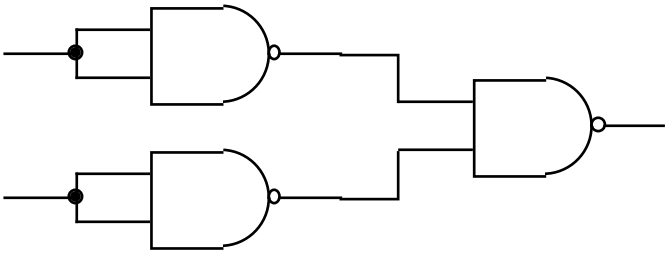
NOT



AND



OR



Exercise 3.7 Using Crocodile Clips and the NAND gate boards, implement an XOR function using only NAND gates.
[Note: this can be done using four NAND gates, but it requires insight to see how.]