ELEC-H-310
Digital electronics

Lecture03
Circuit synthesis, Quine Mc.Cluskey logic optimisation

Dragomir MILOJEVIC
2021

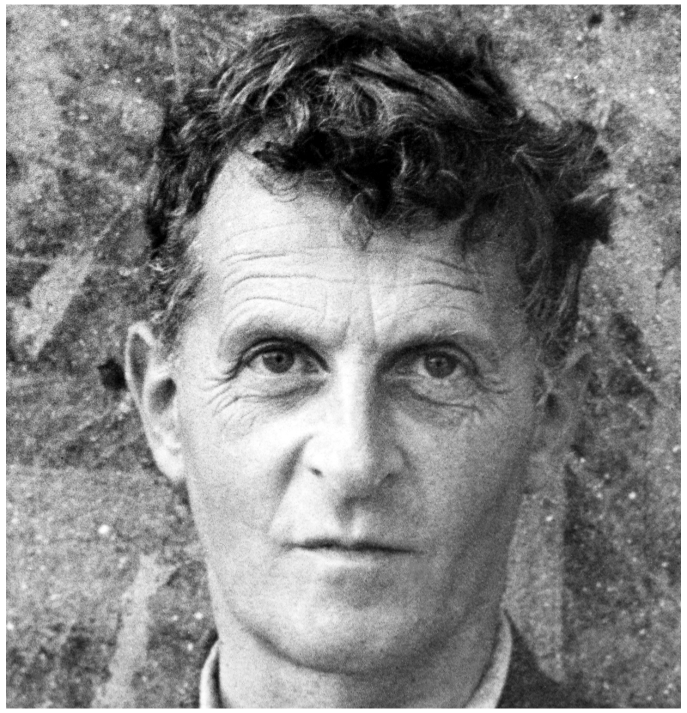
Today

1. Synthesis of combinatorial logic from verbal specifications
2. Quine-Mc.Cluskey logic optimisation method
3. Quine-Mc.Cluskey: logic function covering
4. Hazards in logic circuits
5. Introduction to sequential logic systems

1. Synthesis of combinatorial logic from verbal specifications

Problem

- We describe desired system functionality using **verbal specifications** (i.e. human language)
    - E.g. turn some LEDs on/off depending on the values of some input variables
- We need to translate this (non-formal) system specification into something that can be used by Boolean algebra
- In another words we need to **formalise verbal spec**
- **Formal** in this context means a description that is:
    - **Complete, precise & without any ambiguities**
    - Natural (human) languages are far from being formal…
    - "*What we cannot speak about, we must pass over in silence*", Tractatus Logico-Philosophicus, L.Wittgenstein



- Any of the previously mentioned form: any Boolean expression (whatever the form), a truth table, K-Map etc. are ok …

Synthesis of combinatorial systems

- You now have all the tools to **synthesise a combinatorial logic circuit**
  - Truth tables – analyse all input options and provide desired output
    - **Writing a truth table is formalising your spec**
  - Expression writing – transform the truth table into Boolean equation
  - Optimisation – using any of the methods exposed
- The only thing you need to know is if you are in the presence of the problem that can be solved using **combinatorial logic?**
- **Combinatorial = system output is a function of inputs only!**
- If this the case, the above holds …
- **Advice for combinatorial system synthesis**
  - Start by identifying the inputs & outputs in the system, in fact their number
  - Number of inputs will fix the size of each logic function
  - Number of outputs will fix the number of different logic functions that you need to synthesise (number of Truth Tables, K-Maps, etc.)

Eg. 4-bit BCD to 2421 code converter

- 2421 code is another variant of a **weighted code**
  - we 8421 Binary Coded Decimal (BCD)
  - 2421 encodes 0-9 with different encoding schemes
  - it is a **self-complementary** code
- Problem analysis
  - **OUTPUTS**
    - *2421 code*
    - 4 logic functions (truth tables, K-Maps)
  - **INPUTS**
    - *…4 bit converter…*
    - 4 inputs truth table has $2^4=16$ lines

Décimal	2421
0	0000
1	0001
2	0010
3	0011
4	0100
5	1011
6	1100
7	1101
8	1110
9	1111

Truth table of the converter

- System has 4 binary inputs
- Truth table has $2^4 = 16$ lines
- **What do we do with 16-10=6 remaining lines in the truth table?**
  - We use ***don't cares***!

f_4, f_3, f_2, f_1

Décimal	BCD	2421
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	1011
6	0110	1100
7	0111	1101
8	1000	1110
9	1001	1111
10	1010	----
11	1011	----
12	1100	----
13	1101	----
14	1110	----
15	1111	----

Encoder K-Maps & (optimised) logic functions

f_1	00	01	11	10	x_2x_1
00	0	1	1	0	
01	0	1	1	0	
11	-	-	-	-	
10	0	1	-	-	
x_4x_3					

f_2	00	01	11	10	x_2x_1
00	0	0	1	1	
01	0	1	0	0	
11	-	-	-	-	
10	1	1	-	-	
x_4x_3					

$f_1 = x_1$

$f_2 = x_4 + x_3x_2'x_1 + x_3'x_2$

2. Quine-Mc.Cluskey logic optimisation method

Simplification methods

1. **Axioms & theorems**
2. **K-Maps**
3. **Quine-Mc.Cluskey**

- We will analyse **all possibilities** for logic function simplification
- This is achieved with **systematic analysis of all possible groupings of all minterms & sub-cubes** of a given logic function
- This is an **exhaustive method** since we will extract **all** possible solutions!
- **Bad news**: doing it manually could be time consuming, but it still remains doable for small number of variables and it is a good exercise!
- **Good news**
  - You can easily write a software to automate the whole process
  - Method can be used for problems with huge number of variables
  - You can guarantee best possible solution (if you have access to cost functions of the logic gates that will be used to build the actual circuit)

Quine-McCluskey – overall idea

Split the problem into two different phases:

A. **ANALYSIS phase**

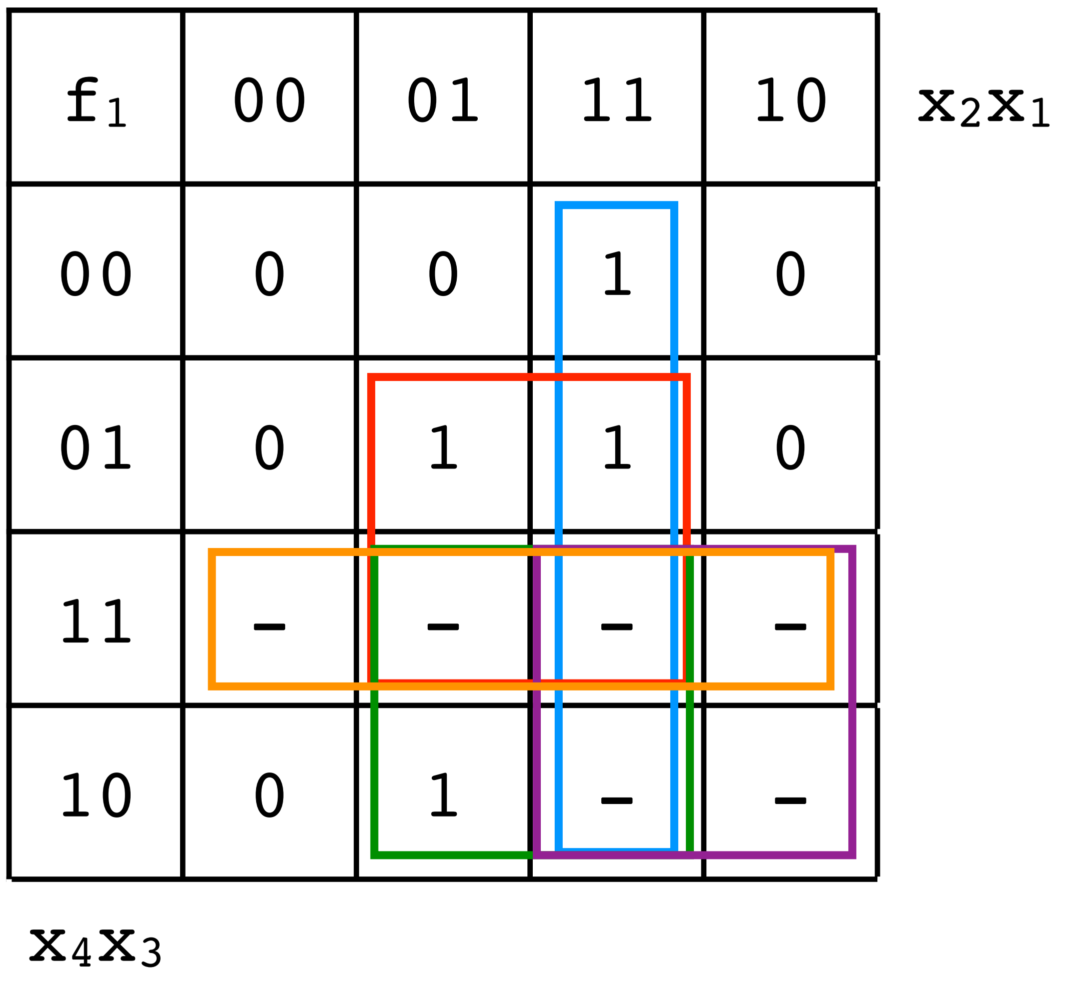
- We systematically search & extract **ALL Prime Implicants**
  - **Prime Implicants** (PIs) – biggest sub-cubes not included in any other sub-cube (*n*-cubes)
- We will introduce a tabular method for easy PIs extraction

B. **SYNTHESIS phase**

- In which we decide on the final logic function expression
- We have seen this: not all PIs are necessary to build the optimal expression, i.e. it is possible that we have a choice
- Two methods that will allow **minimal cover of the logic function**

Extraction of all Prime Implicants

You could do this visually using K-Maps – enumerate **ALL** sub-cubes (including the sub-cubes containing only *don't cares*)



f_1	00	01	11	10
00	0	0	1	0
01	0	1	1	0
11	-	-	-	-
10	0	1	-	-

$IP_1 = x_2x_1$

$IP_2 = x_3x_1$

$IP_3 = x_4x_3$

$IP_4 = x_4x_1$

$IP_5 = x_4x_2$

Problem with the method: once we have more than 5 variables (limit of K-Maps)

Quine-Mc.Cluskey: Analysis phase

Prime Implicants search & extraction (tabular method)

Data preparation step

1. For a logic function `F` of *n* variables, we begin by constructing *m* groups of **all mintermes** of this function

   In each group of mintermes (marked $\mathtt{G_i}$) we will find:

   - `i` logic variables that are equal to `1`;
   - So there are `(n-i)` variables equal to `0`
   - We count the number of `1` in all minterms, and we sort this in the increasing order

Rm. It is possible that we have *m=0* groups (logic function is 0), *m=1* groups, and *m=n*

Quine-Mc.Cluskey: Analysis phase

Remarks for the previous step

a) When building the groups, pay attention on the **position of the LSB** & MSB if your function is specified using decimal interpretation of minterms (typically LSB is on the left and MSB on the right, but always make sure you know what you are doing)

b) First build the corresponding truth table and then do the grouping; do this at least in the beginning to make sure you start on a good basis; different groupings will be done using the tables anyhow

c) Clearly indicate separation between different groups $\mathtt{G_i}$ (here I will use a thick line, you do whatever you want)

These are not mandatory of course, but should help …
(despite all these warnings I do see strange things on the exam)

Quine-Mc.Cluskey: Analysis phase

Example of group G_i construction – Logic function is given as a list of minterms, using their decimal equivalents, so we make a truth table first (LSB is on the right)

`f(a,b,c)=∑(0,1,3,4,5,7)`

	a	b	c	f	
0	0	0	0	1	x
1	0	0	1	1	x
2	0	1	0	0	
3	0	1	1	1	x
4	1	0	0	1	x
5	1	0	1	1	x
6	1	1	0	0	
7	1	1	1	1	x

$\longrightarrow$

G_i groups

	a	b	c
G_0	0	0	0
G_1	0	0	1
	1	0	0
G_2	0	1	1
	1	0	1
G_3	1	1	1

Quine-Mc.Cluskey: Analysis phase

Implicants comparison (algorithm continued)

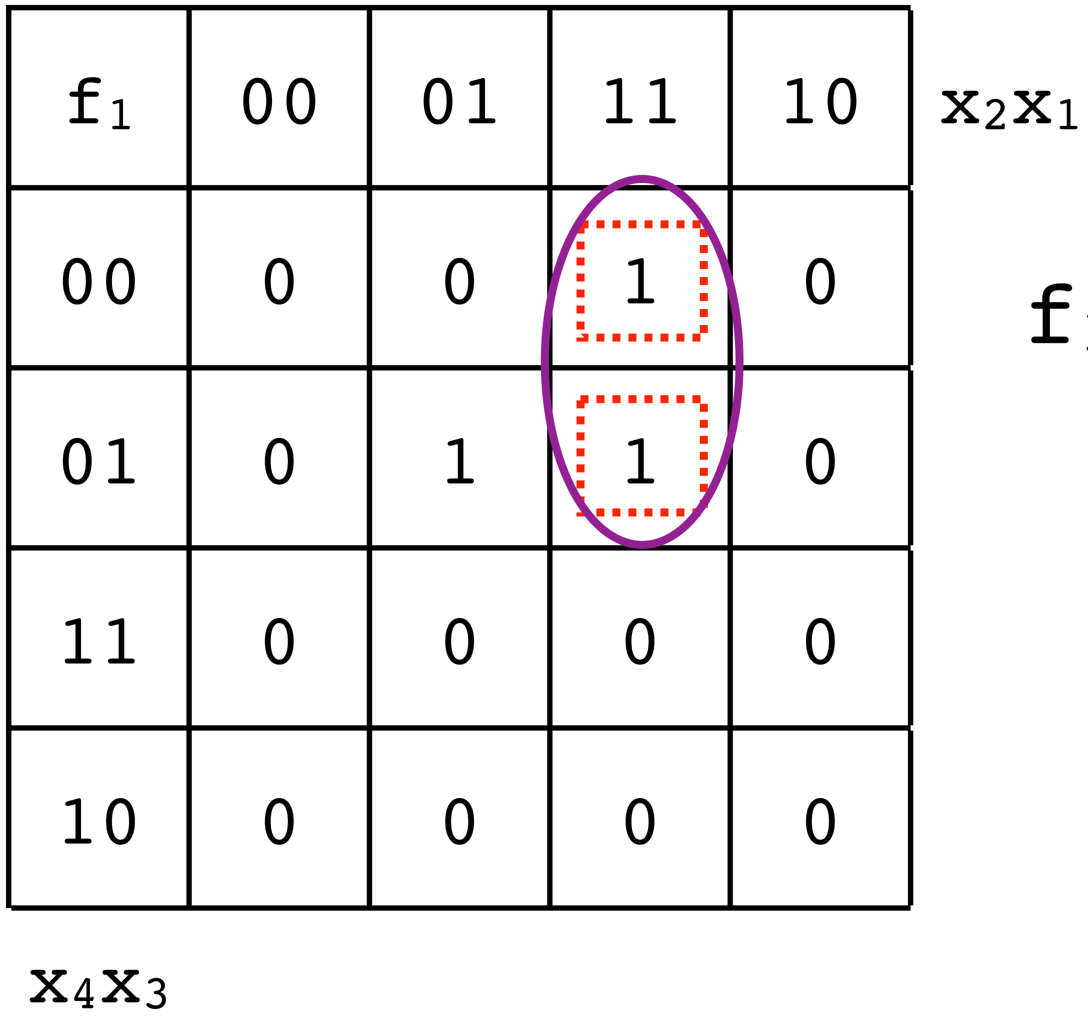
2. We **compare** Hamming distance of **every minterm** in **`Gi`** with **ALL** minterms that belong to group **`Gi+1`** (we do this for all `i`)
   - This is why clear separation between the groups is useful (you miss one in the beginning and your solution is wrong!)
   - If the Hamming distance between two minterms you are comparing is equal to 1 (it is possible that this distance is > 1), we mark this minterm combination as follows:

$$\left.\begin{array}{l}000\\ \\ 010\end{array}\right\}\quad 0\!-\!0 \;\nwarrow \textit{don't care}$$

This is the application of: `x+x'=1`

Quine-Mc.Cluskey: Analysis phase

This is not that different from what we are doing with K-Maps: by combining two minterms, we have found a sub-cube of size 2!



f_1	00	01	11	10	x_2x_1
00	0	0	1	0	
01	0	1	1	0	
11	0	0	0	0	
10	0	0	0	0	
x_4x_3					

$$f_1 = x_4'x_3'x_2x_1 + x_4'x_3x_2x_1$$

$$= x_4'x_2x_1(x_3'+x_3)$$

$$= x_4'x_2x_1$$

Quine-Mc.Cluskey: Analysis phase

Iteration step (algorithm continued)

3. We create a new group $\mathtt{G_i}'$ resulting in all successful comparisons that we could made out of $\mathtt{G_i}$ and $\mathtt{G_{i+1}}$

- Each element of this group $\mathtt{G_i}'$ is a *n*-cube of size 2
- Some of these cubes are sub-cubes of the *n*-cubes of the higher order
- But they could be minterms too …
- After this first pass out of *m* initial groups that we had in $\mathtt{G_i}$ we now have $m—1$ groups in $\mathtt{G_i}'$
- Note that it is also possible **not to have** any pair of minterms for which Hamming distance is equal to 1

Quine-Mc.Cluskey: Analysis phase

Example of G_i' construction

Groups G_i

	a	b	c
G_0	0	0	0
G_1	0	0	1
	1	0	0
G_2	0	1	1
	1	0	1
G_3	1	1	1

All comparisons
(for your reference)

```
G0,G1   000        G2,G3   011
        001                111

        000                101
        100                111
        -------            -------
G1,G2   001
        011

        001
        101

        ~~100~~
        ~~011~~

        100
        101
        -------
```

Hamming distance is >1 (=2) so these two don't match

Groups G_i'

```
G0'     00-   (0,1)
        -00   (0,4)
      ---------------------

G1'     0-1   (1,3)
        -01   (1,5)
        10-   (4,5)
      ---------------------

G2'     -11   (3,7)
        1-1   (5,7)
      ---------------------
```

Quine-Mc.Cluskey: Analysis phase

Prime implicants isolation (algorithm continued)

4. When grouping $\mathtt{G_i}$ and $\mathtt{G_{i+1}}$, some minterms are grouped with others, but it is possible that some have not !
   - All minterms of $\mathtt{G_i}$ that were used to pair with minterms of $\mathtt{G_{i+1}}$ are marked with an `x` in a separate column (I strongly advice you to do this!!!)
   - Group $\mathtt{G_i}'$ is build out of all ***n*-cubes of size 2**
   - We mark all implicants as $\mathtt{PI_k}$ in the order at which they appear
   - If a minterm in $\mathtt{G_i}$ have not been used, this is then a **prime implicant**
   - Note that in the first iteration such *n*-cube is of the size 1 – it is a minterm ! (lonely `1` in the K-Map)

Quine-Mc.Cluskey: Analysis phase

Loop (algorithm continued)

5. The whole process is repeated using the same rules:
    - All elements in group $\mathbf{G_i}'$ gave *n*-cubes of size 2 are used to
    - build $\mathbf{G_i}''$ (*n*-cubes of size 4) and this is of course repeated to
    - generate $\mathbf{G_i}'''$ (*n*-cubes of size 8)

Attention!!! When computing Hamming distance for the elements in $\mathbf{G_i}'$, $\mathbf{G_i}''$ *don't cares* are also taken into account …

```
-000
-100
```

Hamming distance is 1 we can combine

```
-000
--00
```

~~-000~~
~~01-0~~

Hamming distance is 1 we **can not** combine

Quine-Mc.Cluskey: Analysis phase

End

- When do we stop all this?
- Once we can not do the grouping any more !
- That is when it is no longer possible to find pairs to build $\mathbf{G_i^{k+1}}$ out of $\mathbf{G_i^k}$
- In general the convergency of the method is quick, and typically you have less and less things to compare as you move along
- Good to know, since in the beginning you might tell you: "*this is never gonna' end*"

Quine-Mc.Cluskey: Analysis phase

Example of G$_i$'' construction

```
G0'    00-   (0,1)     X
       -00   (0,4)     X
       ---------------------

G1'    0-1   (1,3)     X
       -01   (1,5)     X
       10-   (4,5)     X
       ---------------------

G2'    -11   (3,7)     X
       1-1   (5,7)     X
       ---------------------
```

$\longrightarrow$

```
G0''   -0-   (0,1,4,5)    PI1
       -0-   (0,4,1,5)
       -----------------------------

G1''   --1   (1,3,5,7)    PI2
       --1   (1,5,3,7)
       -----------------------------
```

If we have same PIs we keep only one !

e.g. (0,1,4,5) is same as (0,4,1,5)

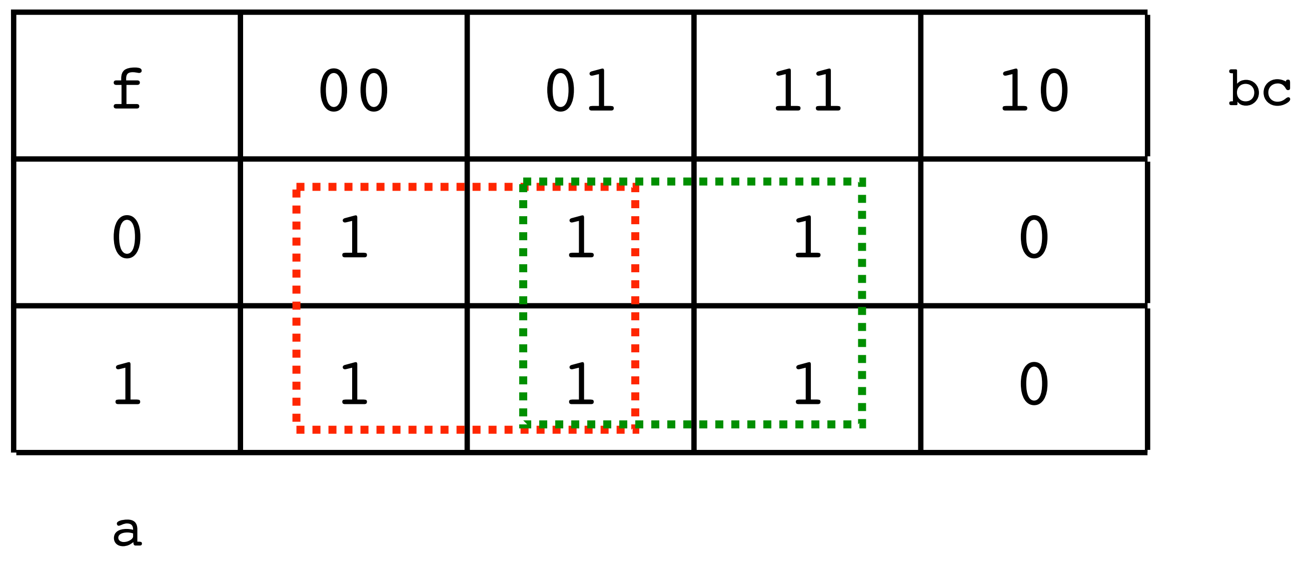
Quine-Mc.Cluskey: Analysis phase

For smaller problems you can always verify the result you've got using K-Maps (I advise you do this in the beginning …)

Take the logic function we used (and for this example extracting all prime implicants using K-Map is much more simpler than using Quine-Mc.Cluskey method)

$f(a,b,c)=\sum(0,1,3,4,5,7)$

PI1=b'
PI2=c



f	00	01	11	10	bc
0	1	1	1	0	
1	1	1	1	0	
a					

Quine-Mc.Cluskey: Analysis phase

Incompletely specified logic functions

- Means that there are don't cares in the logic function:

  `f(a,b,c)=∑m(0,1,3,4,5,7)+∑d(2,6)`

- Solution:

1. We make a hypothesis that all don't cares are set to `1`
   - The idea is to allow all possible simplifications with don't cares (i.e. find all possible minterms)
   - It is important to do this! If you do not do this your solution will be wrong (**Can you explain why?**)
2. Prime Implicants that are either redundant or not useful (such as those composed of don't cares only) will be eliminated during the second phase where we finally synthesise the function

Quine-Mc.Cluskey: Analysis phase

Example of prime implicants that are not useful

f_1	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	~~1~~	0	1	0
10	~~1~~	0	~~1~~	0

x_2x_1

x_4x_3

We find the following 2 implicants:

$IP_1=x_1x_2$

$IP_2=x_4x_1'x_2'$

For the simplified function we do not take IP_2

IP_2 is not useful since it covers only don't cares

3. Minimal function cover

Finding final optimised Boolean expression

Second phase of Quine-McCluskey algorithm: **Minimal function cover**

Two different ways of doing this:

- **Minimal function cover table**
  - Graphical method, efficient, could be automatised; but there are cases where finding the best solution will not be easy … we want something better than this
- **Minimal cover equation** – solves the above problem
  - Algebraic approach
  - **Generates all possible solutions!**
  - We can pick the best solution depending on target technology (here we stop, circtui technologies are out of scope in these lectures and will be covered in later lectures on digital system design such as ELECH409 or ELECH505)
  - Efficient, a bit of more work, but applicable in any case
  - Can be automatised (important)

Tabular method

1. Make a table (after finding out all the implicants – Part A) out of:
    - **mintermes** – written as different lines in the table (could be the other way around, but let's say this is the way it should be done) and
    - **prime implicants** – all sub-cubes that we have identified previously
2. **Goal –** find (easily!) all **essential prime implicants** (because these we need to take anyhow) AND the right combination of **prime implicants** that will enable to **cover completely** the logic function
    - Terme **cover** – refers to "packing" all `1's` in the K-Map into *n*-cubes

f_1	00	01	11	10	x_2x_1
00	0	0	1	0	
01	0	0	1	0	
11	1	0	1	0	
10	1	0	1	0	
x_4x_3					

Tabular method

- We construct a table showing the relationship between the **minterms only** of the logic function and the Prime Implicants we found during analysis phase

- In this table we do not put don't care terms !!!
  (reason we say minterms only in the previous bullet)

- For each Prime Implicant (PI) we indicate ALL minterms that are covered by this PI

- Example: for a logic function of 4 variables, the following PI:

  $x_4x_2'x_1$

  enable us to cover the following minterms:

  $x_4x_3x_2'x_1$ and $x_4x_3'x_2'x_1$

Tabular method: example

Take the following logic function and PIs:

$f(x_4,x_3,x_2,x_1,x_0)=\sum m(\underbrace{0,1,2,5,14,16,18,24,26,30}) + \sum d(3,13,28)$

PI1= $x_4'x_3'x_1'x_0$
PI2= $x_4'x_2x_1'x_0$
PI3= $x_3x_2x_1x_0'$
PI4= $x_4'x_3'x_2'$
PI5= $x_3'x_2'x_0'$
PI6= $x_4x_3x_0'$
PI7= $x_4x_2'x_0'$

Note the absence of don't cares!

		PI1	PI2	PI3	PI4	PI5	PI6	PI7
00000	0				x	x		
00001	1	x			x			
00010	2				x	x		
00101	5	x	x					
01110	14			x				
10000	16					x		x
10010	18					x		x
11000	24						x	x
11010	26						x	x
11110	30			x			x	

Tabular method: essential prime implicants

We first look for Essential Prime Implicants (EPIs): these are **the only ones** to cover certain minterms (`PI3` is the only one to cover `01110`)

		PI1	PI2	PI3	PI4	PI5	PI6	PI7
00000	0				x	x		
00001	1	x			x			
00010	2				x	x		
00101	5	x	x					
01110	14			x				
10000	16					x		x
10010	18					x		x
11000	24						x	x
11010	26						x	x
11110	30			x			x	

Tabular method: others?

We need to cover all mintermes: we need to chose PIs so that all minterms are covered; a choice of a given PI implies that we do not **have to cover all other minterms covered by this PI**

No need to cover them later

Already covered by `PI3`

		PI1	PI2	PI3	PI4	PI5	PI6	IP7
00000	0				x	x		
00001	1	x			x			
00010	2				x	x		
00101	5	x	x					
01110	14			x				
10000	16					x		x
10010	18					x		x
11000	24						x	x
11010	26						x	x
11110	30			x			x	

Tabular method in practice

- Sometimes the choice of PIs is more or less simple from the table
- In the example given on previous slide (on purpose) **it is not**!
- This motivates an alternative method for logic function cover
- We would like to **formalise the choice of PIs**, so that we could eventually analyse all possible covers!
- If this is the case, we could generate all possible expressions for F that all could be potentially optimal
- By knowing the gate cost, we could then apply any optimisation method to pick the right one

Formalising the choice of PIs using
minimal cover equation

Reasoning behind

		PI1	PI2	PI3	PI4	PI5	PI6	IP7
00000	0				x	x		
00001	1	x			x			
00010	2				x	x		
00101	5	x	x					
01110	14			x				
10000	16					x		x
10010	18					x		x
11000	24						x	x
11010	26						x	x
11110	30			x			x	

To cover the minterm 00000 we have the choice between `PI4` **OR** `PI5`

We can write this as: (i_4+i_5) – attention here we use `i` lowercase to avoid confusion with normal PIs

For minterm 00001 we have the choice between PI1 **OR** PI4: (i_1+i_4)

How to interpret this equation?

		PI1	PI2	PI3	PI4	PI5	PI6	PI7
00000	0				x	x		
00001	1	x			x			
00010	2				x	x		
00101	5	x	x					
01110	14			x				
10000	16					x		x
10010	18					x		x
11000	24						x	x
11010	26						x	x
11110	30			x			x	

We need to cover **ALL** mintermes, so we link different expressions with a logical **AND**:

$$1=(i_4+i_5)(i_1+i_4)...$$

Minimal cover equation completed

		PI1	**PI2**	**PI3**	**PI4**	**PI5**	**PI6**	**PI7**
00000	0				x	x		
00001	1	x			x			
00010	2				x	x		
00101	5	x	x					
01110	14			x				
10000	16					x		x
10010	18					x		x
11000	24						x	x
11010	26						x	x
11110	30			x			x	

$$1=(i_4+i_5)(i_1+i_4)(i_4+i_5)(i_1+i_2)i_3$$
$$(i_5+i_7)(i_5+i_7)(i_6+i_7)(i_6+i_7)(i_3+i_6)$$

Simplifying minimal cover equation

$$1=(\mathtt{i}_4+\mathtt{i}_5)(\mathtt{i}_1+\mathtt{i}_4)\cancel{(\mathtt{i}_4+\mathtt{i}_5)}(\mathtt{i}_1+\mathtt{i}_2)\mathbf{i_3}(\mathtt{i}_5+\mathtt{i}_7)(\mathtt{i}_5+\mathtt{i}_7)(\mathtt{i}_6+\mathtt{i}_7)(\mathtt{i}_6+\mathtt{i}_7)\cancel{(\mathtt{i}_3+\mathtt{i}_6)}$$

- Essential Prime Implicants appear as isolated terms, here $\mathtt{i}_3$
- If any other term of this equation has an Essential Prime Implicant we can drop that term …
  - In the above the term ($\mathtt{i}_3+\mathtt{i}_6$) can be dropped.
    Can you explain why?
- Simplification of double terms (**Can you explain why?**):
  $(\mathtt{i}_4+\mathtt{i}_5)(\mathtt{i}_4+\mathtt{i}_5) = (\mathtt{i}_4+\mathtt{i}_5)$

Simplification de l’eq. de couverture

doubles

$$1=(i_4+i_5)(i_1+i_4)\cancel{(i_4+i_5)}(i_1+i_2)i_3(i_5+i_7)\cancel{(i_5+i_7)}(i_6+i_7)\cancel{(i_6+i_7)}$$

$$= i_3(i_4+i_5)(i_1+i_4)(i_1+i_2)(i_5+i_7)(i_6+i_7)$$

distribute

$$= i_3(i_4+i_5)(i_1+i_2i_4)(i_7+i_5i_6) \quad \{\rightarrow (i_1=i_1+i_1i_2+i_1i_4)\}$$

$$= i_3(i_4+i_5)(i_1i_7+i_1i_5i_6+i_2i_4i_7+i_2i_4i_5i_6)$$

$$= i_3(i_4i_1i_7 + \boxed{i_4i_1i_5i_6} + \boxed{i_2i_4i_7} + \boxed{i_2i_4i_5i_6} + i_5i_1i_7 + \boxed{i_1i_5i_6} + \boxed{i_5i_2i_4i_7} + \boxed{i_2i_4i_5i_6})$$

same

$$= i_3(i_4i_1i_7 + i_1i_5i_6 + i_1i_5i_7 + i_2i_4i_7 + i_2i_4i_5i_6)$$

Simplifying minimal cover equation

$$1=i_3(i_4i_1i_7 + i_1i_5i_6 + i_1i_5i_7 + i_2i_4i_7 + i_2i_4i_5i_6)$$

$$=i_3i_4i_1i_7+i_3i_1i_5i_6+i_3i_1i_5i_7+i_3i_2i_4i_7+i_3i_2i_4i_5i_6$$

How do we interpret this:

Each term of this expression is a **possible solution!**

For this particular example we have 5 possible solutions:

F_1, F_2, F_3, F_4, F_5 since the equation has 4 terms

Interpreting minimal cover equation

- Each solution (term of the minimal cover equation) indicates which prime implicants **we will have to take**
- In the example for the first solution 1, we have to take:

  IP_3 AND IP_4 AND IP_1 AND IP_7

- But when we write the final expression, we need to use OR and not AND since the expression is the Sum of Products
- We then write:

$$F_1 = IP_3 + IP_4 + IP_1 + IP_7$$

- **All possible** expressions of the logic function F:

$$F_1 = IP_3 + IP_4 + IP_1 + IP_7$$

$$F_2 = IP_3 + IP_1 + IP_5 + IP_6$$

$$F_3 = IP_3 + IP_1 + IP_5 + IP_7$$

$$F_4 = IP_3 + IP_2 + IP_4 + IP_7$$

$$F_5 = IP_3 + IP_2 + IP_4 + IP_5 + IP_6$$

Finding expressions

$IP1 = x_4'x_3'x_1'x_0$
$IP2 = x_4'x_2x_1'x_0$
$IP3 = x_3x_2x_1x_0'$
$IP4 = x_4'x_3'x_2'$
$IP5 = x_3'x_2'x_0'$
$IP6 = x_4x_3x_0'$
$IP7 = x_4x_2'x_0'$

$F_1 = i_3i_4i_1i_7 = IP_3 + IP_4 + IP_1 + IP_7$

$F_2 = i_3i_1i_5i_6 = IP_3 + IP_1 + IP_5 + IP_6$

$F_3 = i_3i_1i_5i_7 = IP_3 + IP_1 + IP_5 + IP_7$

$F_4 = i_3i_2i_4i_7 = IP_3 + IP_2 + IP_4 + IP_7$

$F_5 = i_3i_2i_4i_5i_6 = IP_3 + IP_2 + IP_4 + IP_5 + IP_6$

- Now that we have all the expressions, we can chose the best possible solution: the one with the **least number** of **smallest possible** gates
- In the example above F_5 be excluded (2 gates of 4 inputs & 3 gates of 3 inputs, so 5 gates in all)
- Between F_1, F_2, F_3, F_4 we can chose any of the three one (2 gates of 4 & 2 gates of 3 inputs)
- Final choice will depend on the technology used to implement the circuit
- Out of scope for us … smallest one is good enough

Final simplified Boolean equations

$IP_1 = x_4'x_3'x_1'x_0$

$IP_2 = x_4'x_2x_1'x_0$

$IP_3 = x_3x_2x_1x_0'$

$IP_4 = x_4'x_3'x_2'$

$IP_5 = x_3'x_2'x_0'$

$IP_6 = x_4x_3x_0'$

$IP_7 = x_4x_2'x_0'$

$F_1 = IP_3 + IP_4 + IP_1 + IP_7$

$F_2 = IP_3 + IP_1 + IP_5 + IP_6$

$F_3 = IP_3 + IP_1 + IP_5 + IP_7$

$F_4 = IP_3 + IP_2 + IP_4 + IP_7$

$F_1 = x_3x_2x_1x_0' + x_4'x_3'x_2' + x_4'x_3'x_1'x_0 + x_4x_2'x_0'$

$F_2 = x_3x_2x_1x_0' + x_4'x_3'x_1'x_0 + x_3'x_2'x_0' + x_4x_3x_0'$

$F_3 = x_3x_2x_1x_0' + x_4'x_2x_1'x_0 + x_4'x_3'x_2' + x_4x_2'x_0'$

$F_4 = x_3x_2x_1x_0' + x_4'x_2x_1'x_0 + x_4'x_3'x_2' + x_4x_2'x_0'$

4. Hazards

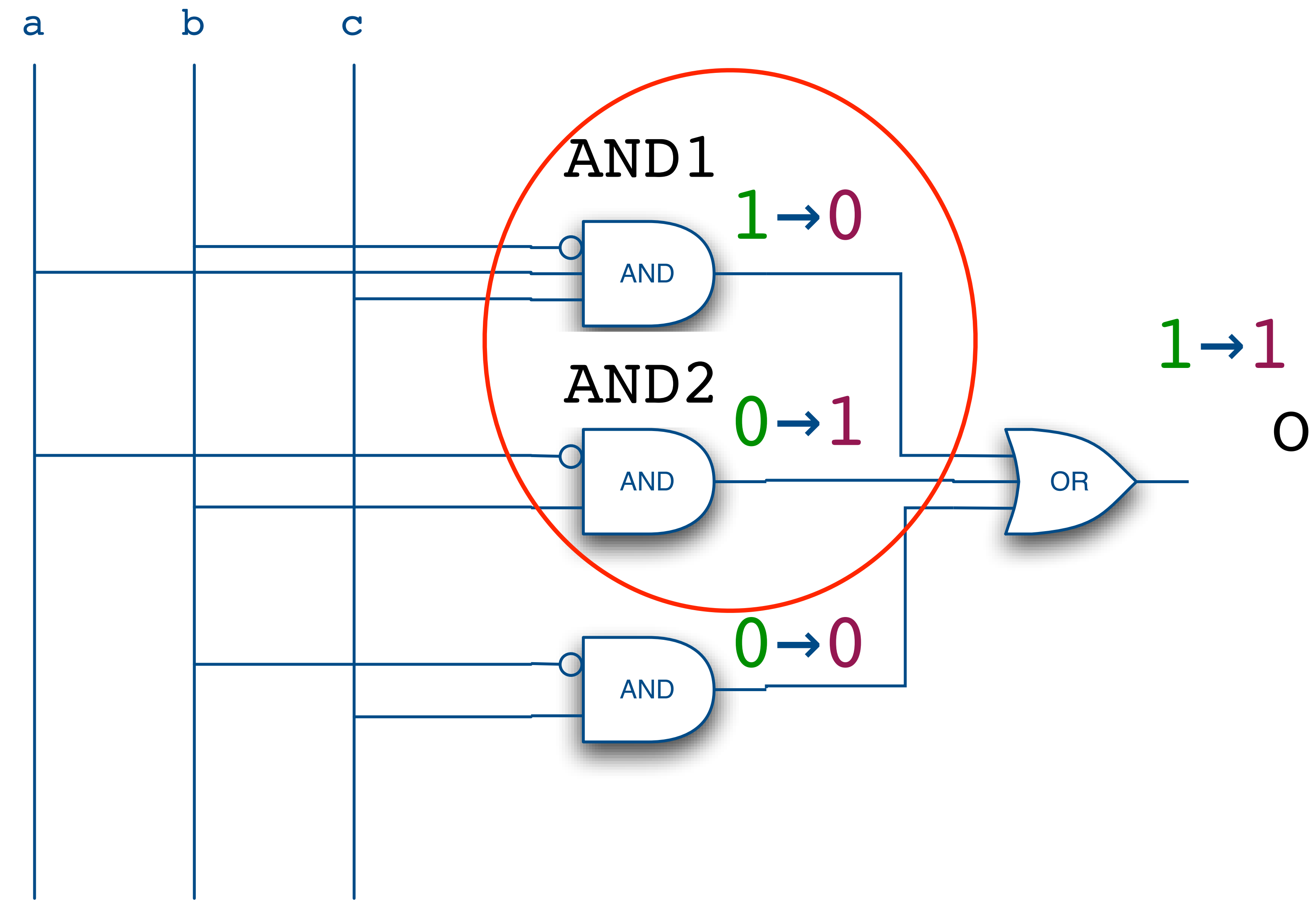
We speak of these in real circuits …

- Until now we considered our circuits to be **ideal**
- We assumed no **gate** and/or **wire delays**, i.e. they switch instantly
- This means that when the input changes (e.g. Alice pushes the button), the output is produced **instantaneously**
- Sum-of-Products (i.e. AND and OR gates) will calculate the right output immediately
- Obviously in the real world this is not the case, no matter how fast is your circuit (after all this is just the change in scale)
- Real circuits have delays and this fact **will affect overall system operation**
- Worse … these delays are not homogeneous, i.e. they will vary from component to component and in different parts of the circuit

Gate delays and end result computation

Logic function is cabled as a set of AND and OR gates, i.e. every minterm will results in an AND gate.

Input changes so that the top and the bottom gates change their outputs

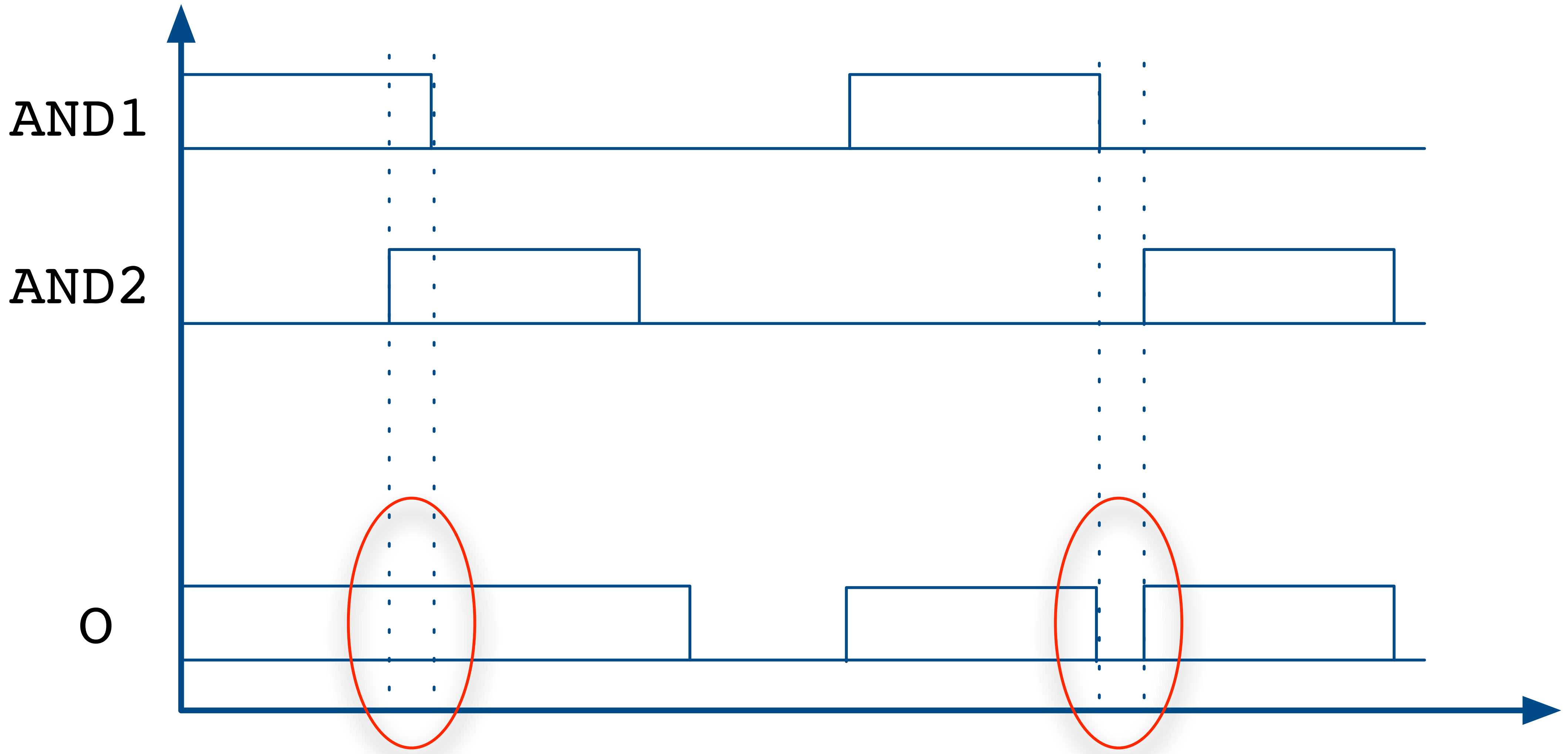


What happens when different AND gates switch, knowing that this doesn’t happen instantaneously?

Two (VERY MUCH) different situations:

AND2 turns on **before** gate AND1 turns off…

AND2 turns on **after** AND1 turned off …



Assuming that OR reacts quickly (this is possible), during a short amount of time we will see a zero on O

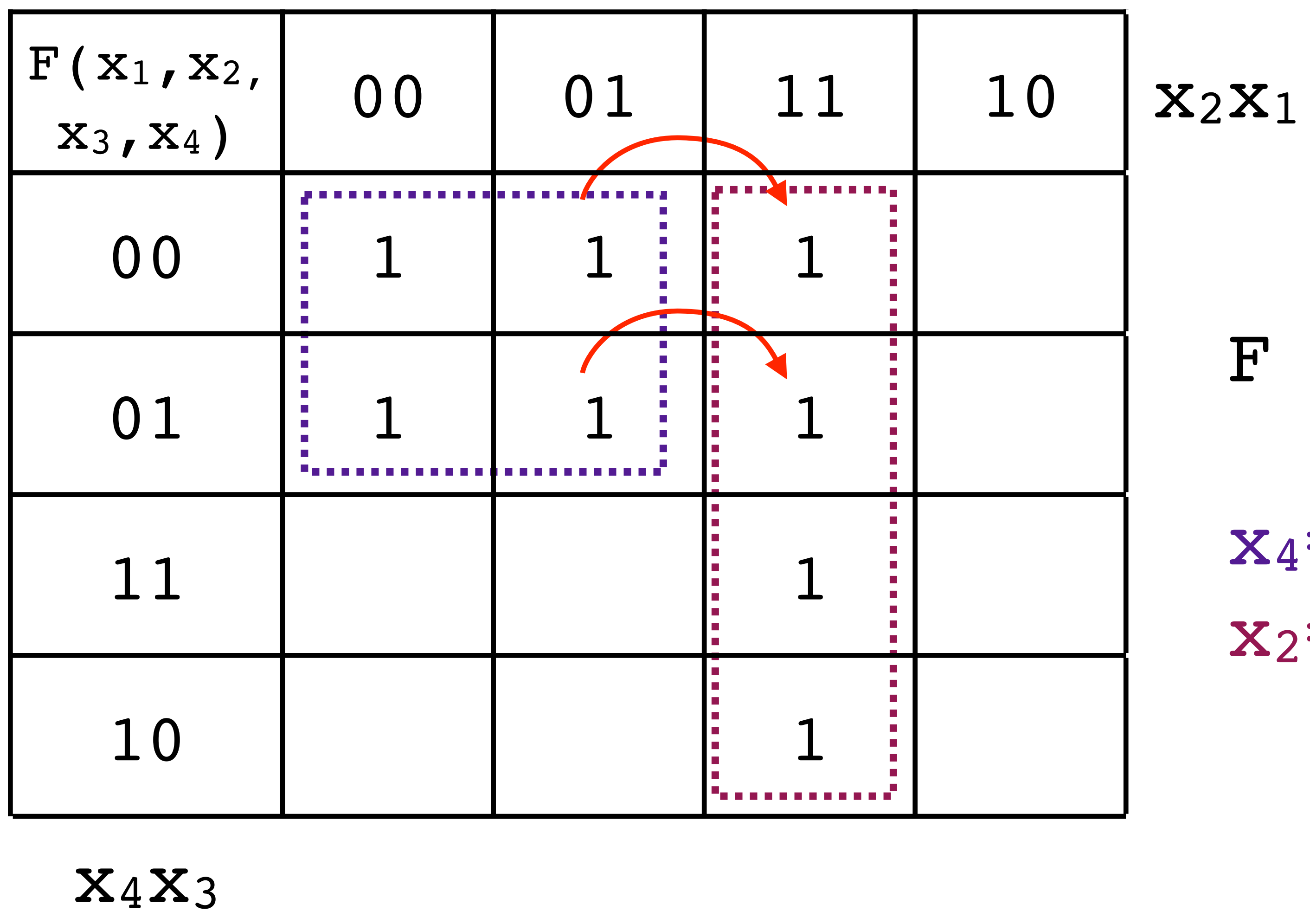
AND1 turned off …

Hazards

- We speak about glitches or hazards defined as: **non-desired impulse of a (very) short duration**
- **Are glitches safe or not?**
- It depends on what we do with the output
  - If the output is used by someone/something not sensible to such variations – **we do not care !**
    - Example: human reading some output shown on LED/LCD screen
    - Our sampling rate is roughly 30ms (30 images/sec)
    - If the glitch duration is less then that, we will not notice anything
  - If the system is used to control another digital systems, it is likely that the controlled system will be able to capture this glitch; sometimes this could lead to uncontrolled behaviour
  - In general we do everything to avoid hazards

Hazards in an K-Map

We switch from one *n*-cube to the other; each *n*-cube, i.e. term will correspond to a AND logic gate in an SoP; this means that when one gates switches from 1 to 0, the other will do so from 0 to 1; **we move from one to another n-cube** !



$F(x_1,x_2,x_3,x_4)$	00	01	11	10
00	1	1	1	
01	1	1	1	
11			1	
10			1	

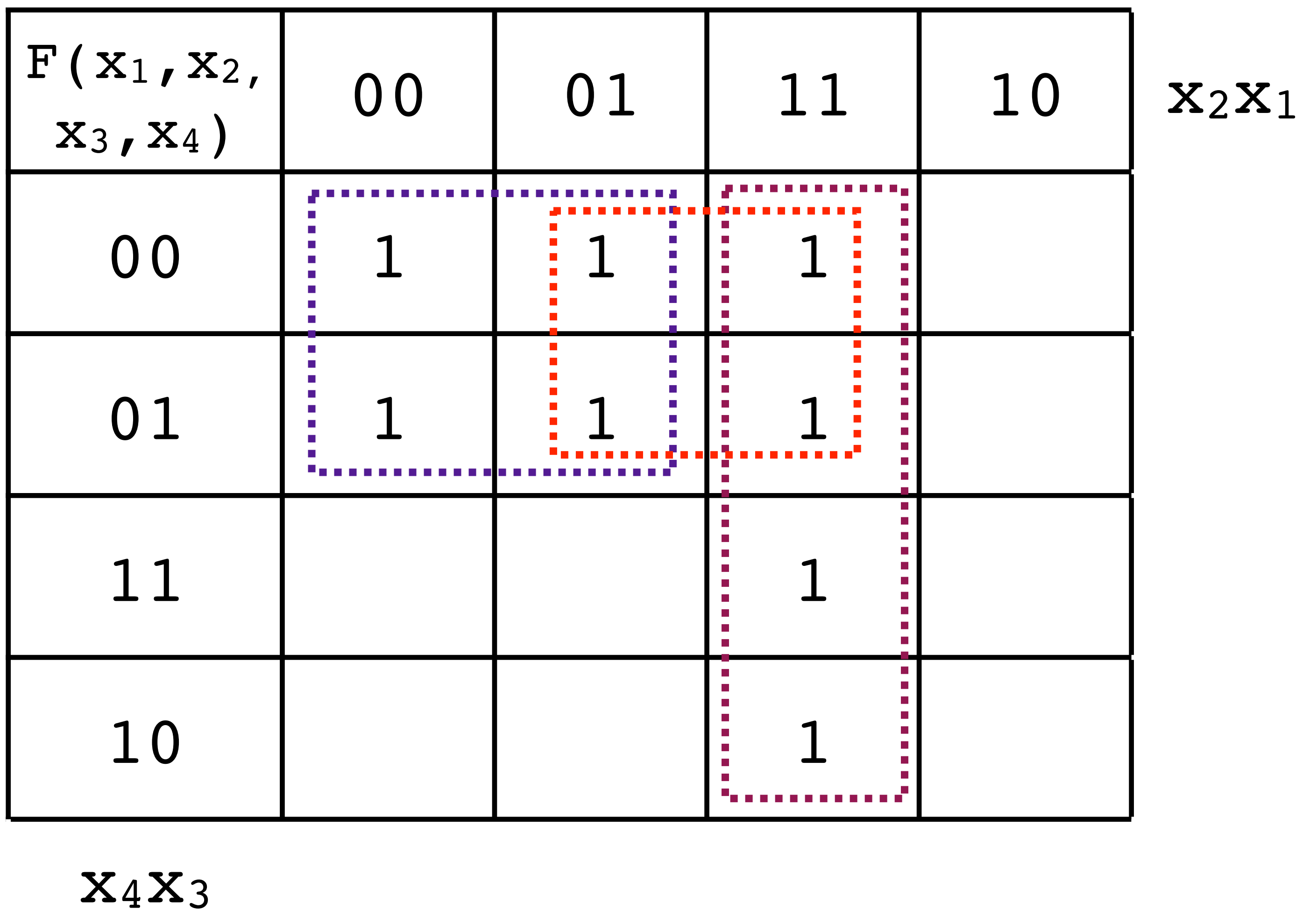
$$F = x_4'x_2' + x_2x_1$$

$x_4=x_2=0 \rightarrow x_4'x_2'=1$

$x_2=1, x_1=1 \rightarrow x_2x_1=1$

Solution (partial) to hazard problem

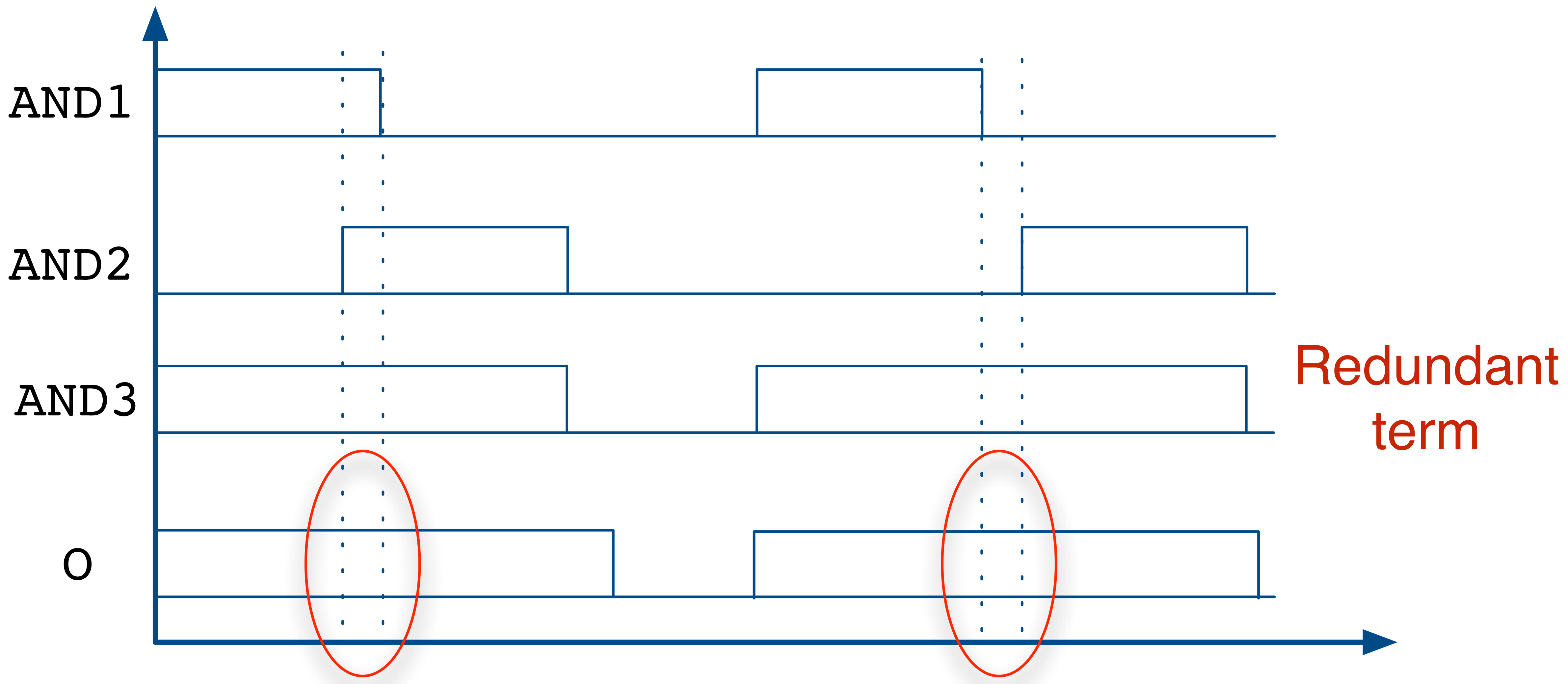
Add a **redundant term** to maintain the output to 1 during the transition: it is an AND gate that will output 1 during the transition



$F(x_1,x_2,x_3,x_4)$	00	01	11	10
00	1	1	1	
01	1	1	1	
11			1	
10			1	

$$F = x_4'x_2' + x_2x_1 + x_4'x_1$$

Wave diagram with the redundant term



Added AND gate **maintains** the output to 1 when two other AND gates are switching!

5. Sequential systems

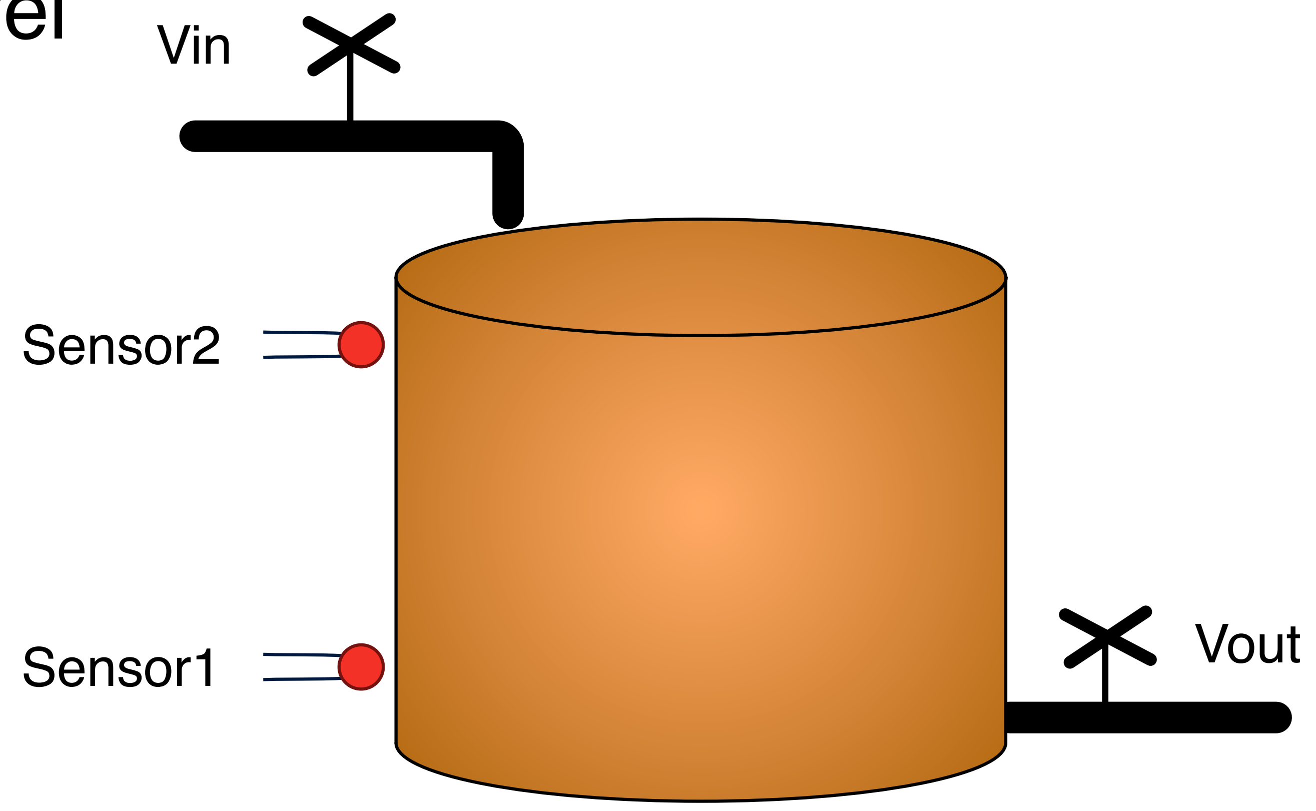
ToC revisited

Three chapters

Limitation(s) of combinatorial systems

Consider the following problem: we want to use a **digital system** to control the process of filling a tank with some kind of liquid. To enable the control we have access to **two sensors**: one sensor on the bottom of the tank (possibly signalling that the tank is empty) and the other one on the top (signalling that the tank is full).

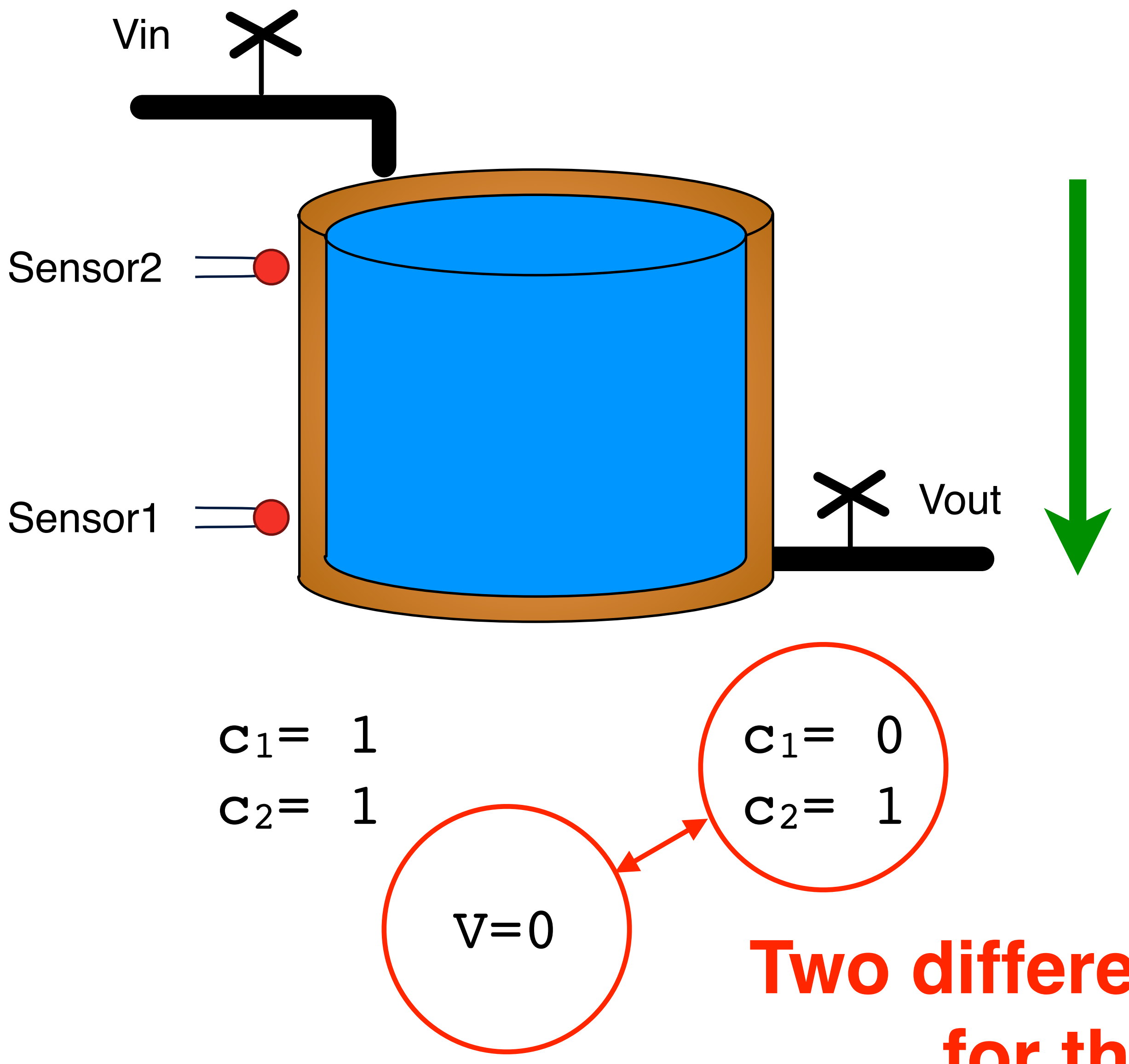
The system should control the level of the liquid, so that it never goes below the Sensor1 or above the Sensor2, by opening and/or closing the valve Vin. Note that valve Vout acts as a random variable emptying the tank at the pace we can not predict in advance …

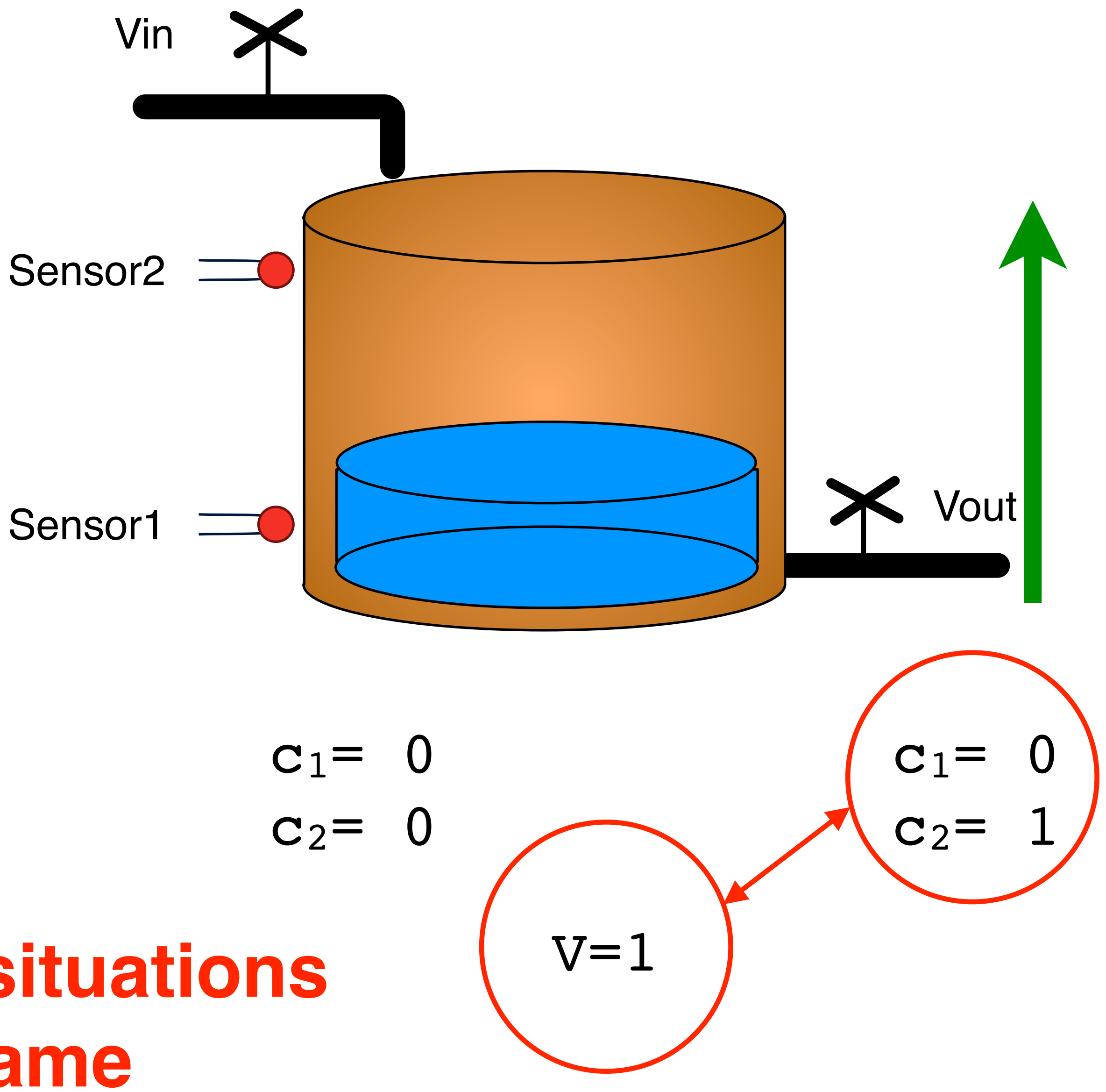


Limitation(s) of combinatorial systems

Imagine the tank already filled with liquid and Vout being open:

Imagine now that level in the tank went below Sensor1 & Vin being open:





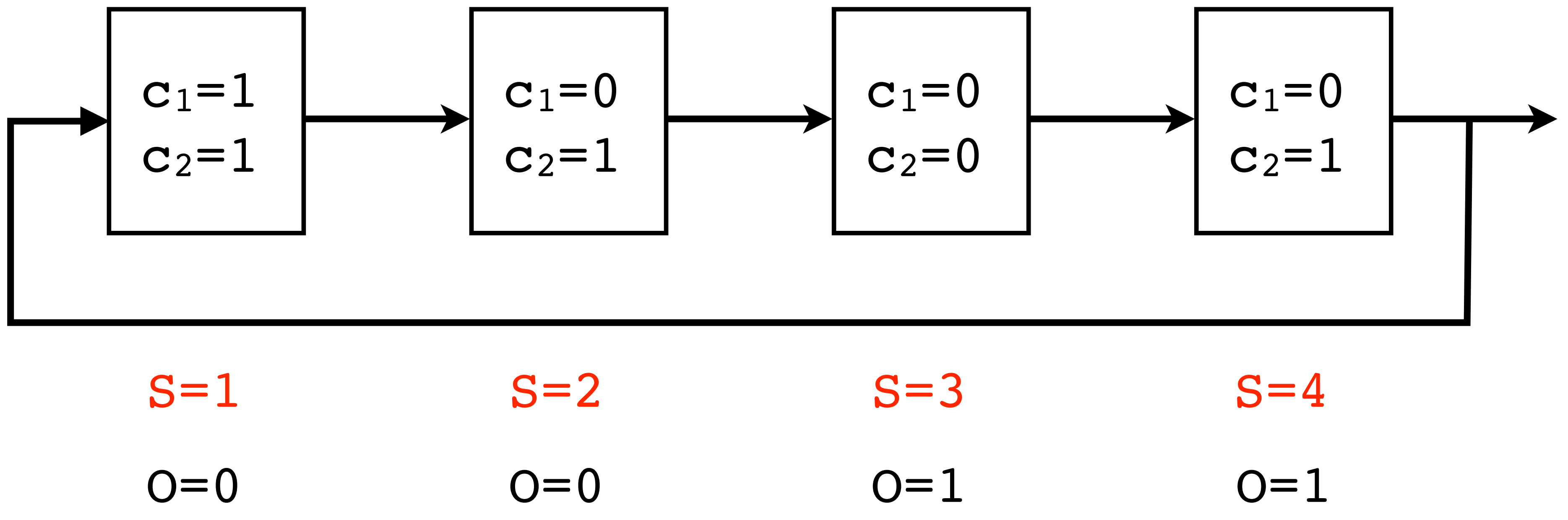
Two different situations for the same combination of inputs!

Limitation(s) of combinatorial systems

- We need to somehow make the difference between the following two **situations**:
  - Tanks is full and is being emptied (Vin should remain closed) &
  - Tank is empty and is being filled (Vin is opened)
- In order to be able to distinguish between these two situations we need to **memorise** the previous **situation** in which the system has been (we should open Vin and keep it open only if the system has reached the bottom sensor level)
- We speak about two things:
  - ***memorise*** – we need some kind of a **memory element device** to store the situation
  - ***situation*** – better known as **the system state**

System evolution as sequence of states

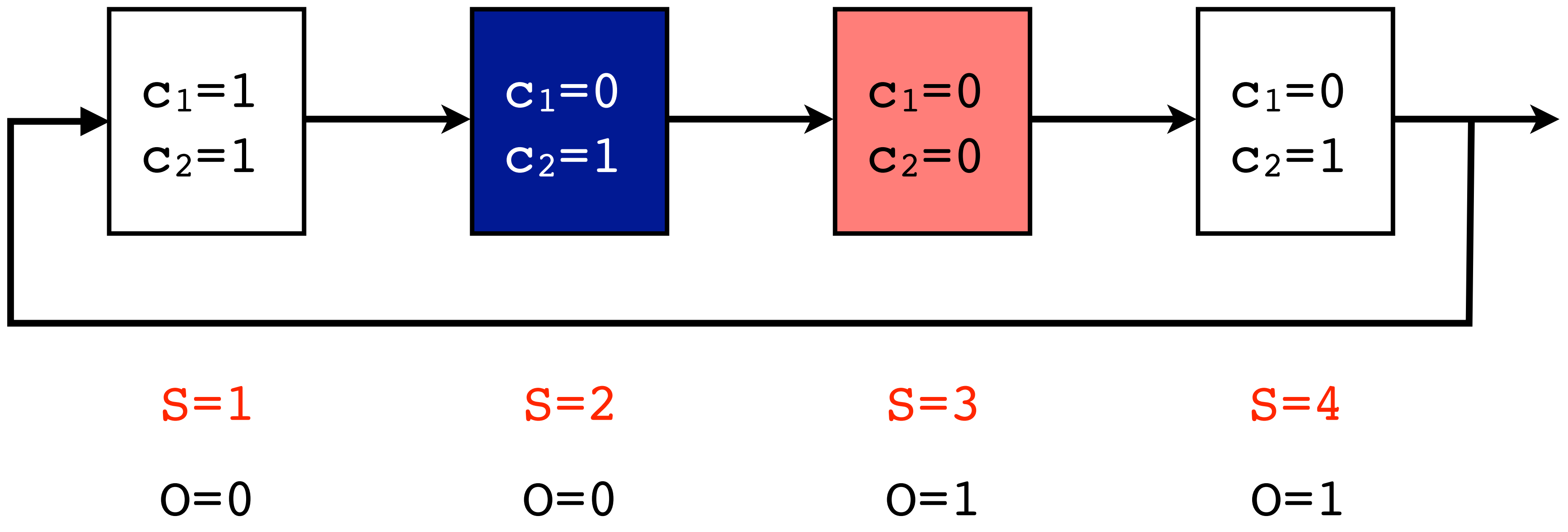
For the example of the tank:



- If inputs `01` AND system is in state `2` (`S2`), **keep Vin closed** (tank is being full and now is emptied)
- If inputs `01` AND system in state `4` (`S4`), i.e. we went below the level of the bottom sensor, **then open the valve Vin** to fill the tank
- **Provide description of states in English?**

System evolution: from current to next state

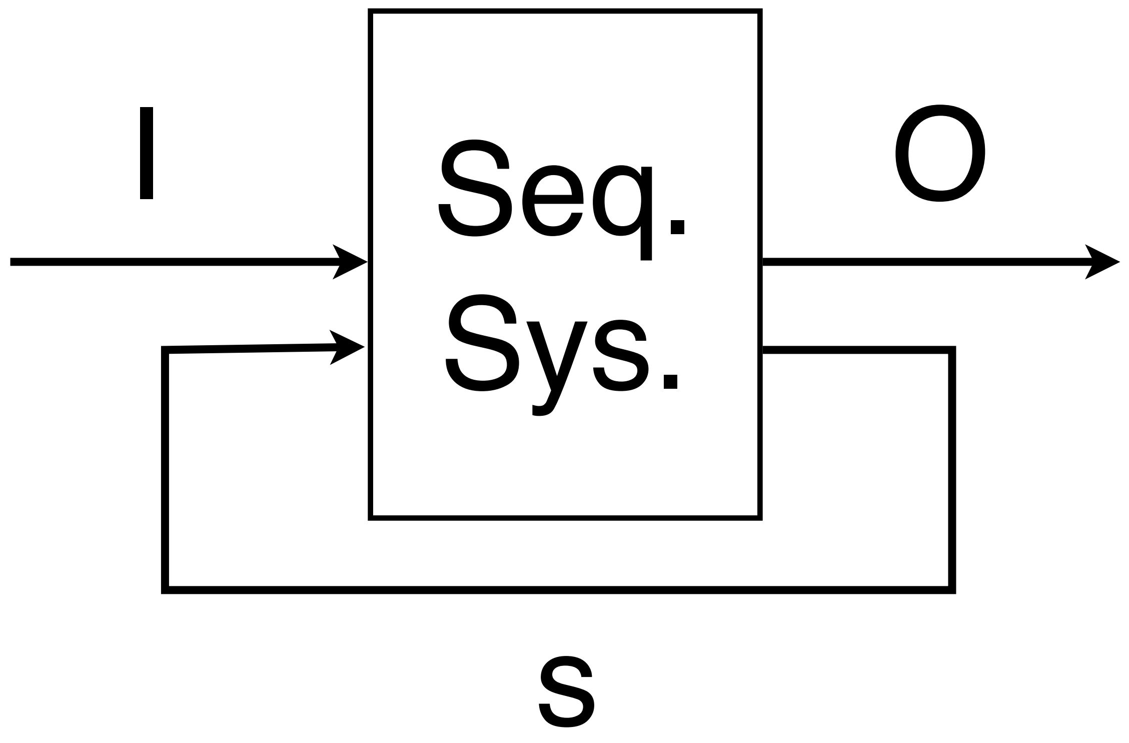
Let us imagine that we are in the S2:



- This is the present (or current state), the system can't go nowhere else except in the (next state)
- This is possible only if $c_2=0$
- We see that the system moves from one state to another (**state transition**), depending on the **present state** & the **current inputs**

Sequential systems: more formal definition

In sequential systems the outputs depend not only on the **system inputs**, but also on the **internal state**. The system **autonomously** decides on the **future state** depending on **inputs and the current state**.



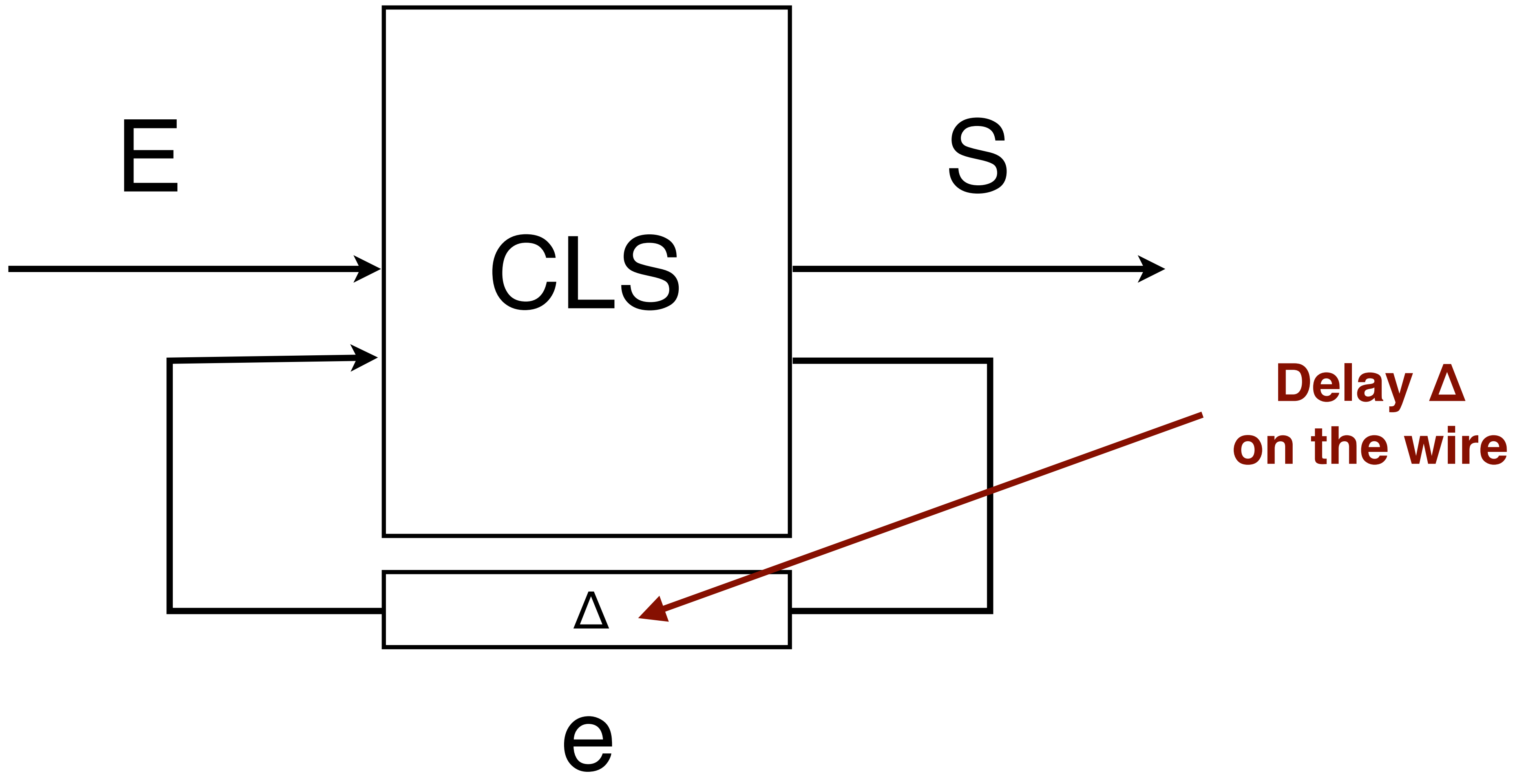
`O`	–	outputs
`I`	–	inputs
`S`	–	Future states
`s`	–	Present states

$$O = f(I, s)$$
$$S = f(I, s)$$

Where we can find `s` et `S` ?

Difference: present/future states

The difference between present & future states can be modelled (represented on paper) using **delays.** We add some propagation time on the feedback wire — and this is actually what happens at the level of a physical circuit.



Representing sequential systems

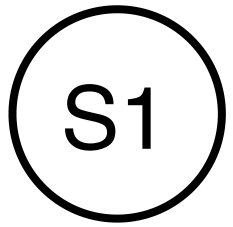
1. **State graphs – Graphical** representation; provides a neat view of all **states** and **state transitions** depending on the machine inputs; nice to see, but not preferred way to work with (lack of in-depth view of the system behaviour); not used for synthesis !!!

2. **State table (Hufmann table, transition matrix, etc.)**
   Tabular representation of all system states and transitions; preferred way of describing state machines (sequential systems) because it forces us to be **systematic** when synthesising the system (we will have to look into all possible outcomes)

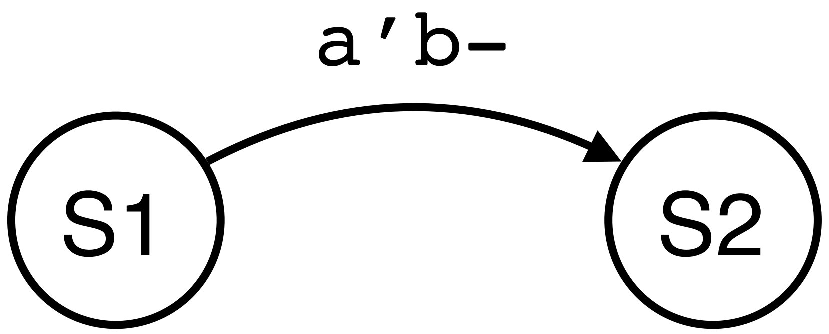
3. **Logic Equations –** Boolean expressions for different logic functions; this is typically derived from the above to build the circuit (these expression will be optimised)

State graphs

- System state indicated using **circles**



- State transition (occurs between stable states) is indicated using **oriented arcs**, from present state to the next state



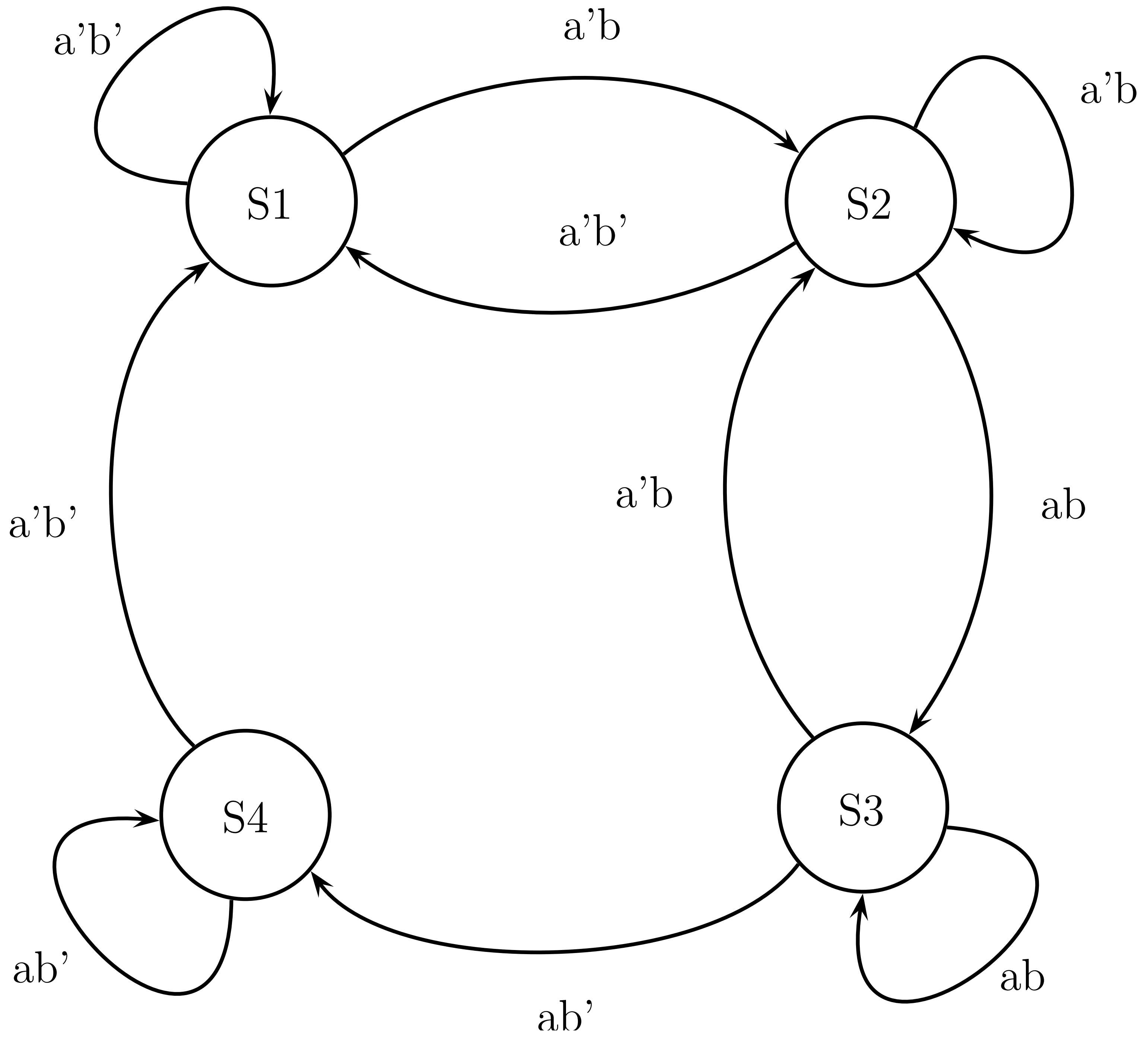
- Note that transition **condition(s)** (on inputs) are written on the top of the arc and we use normal Boolean formalism to do that

  Example
  We now here that we have 3 variables `abc`: since `a’b-`; meaning when `a=0 AND b=1`, whatever the value of `c`

Example of the state graph

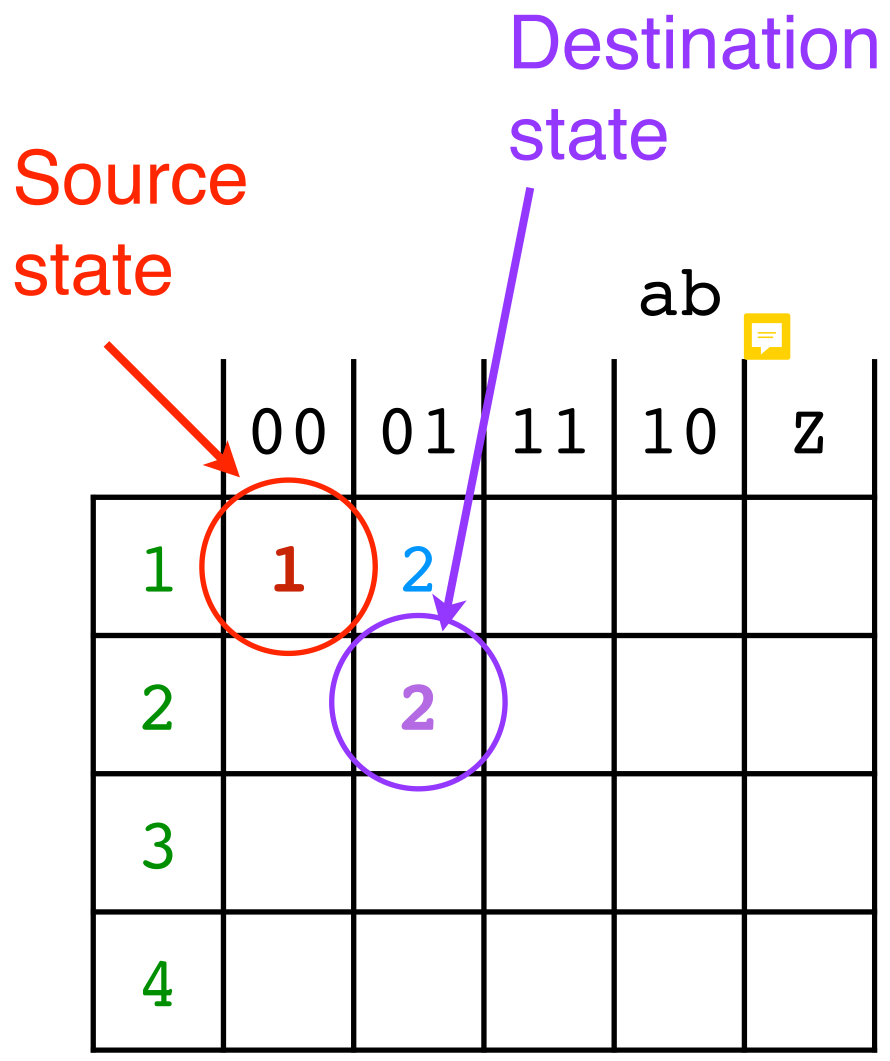
4 states and

2 inputs



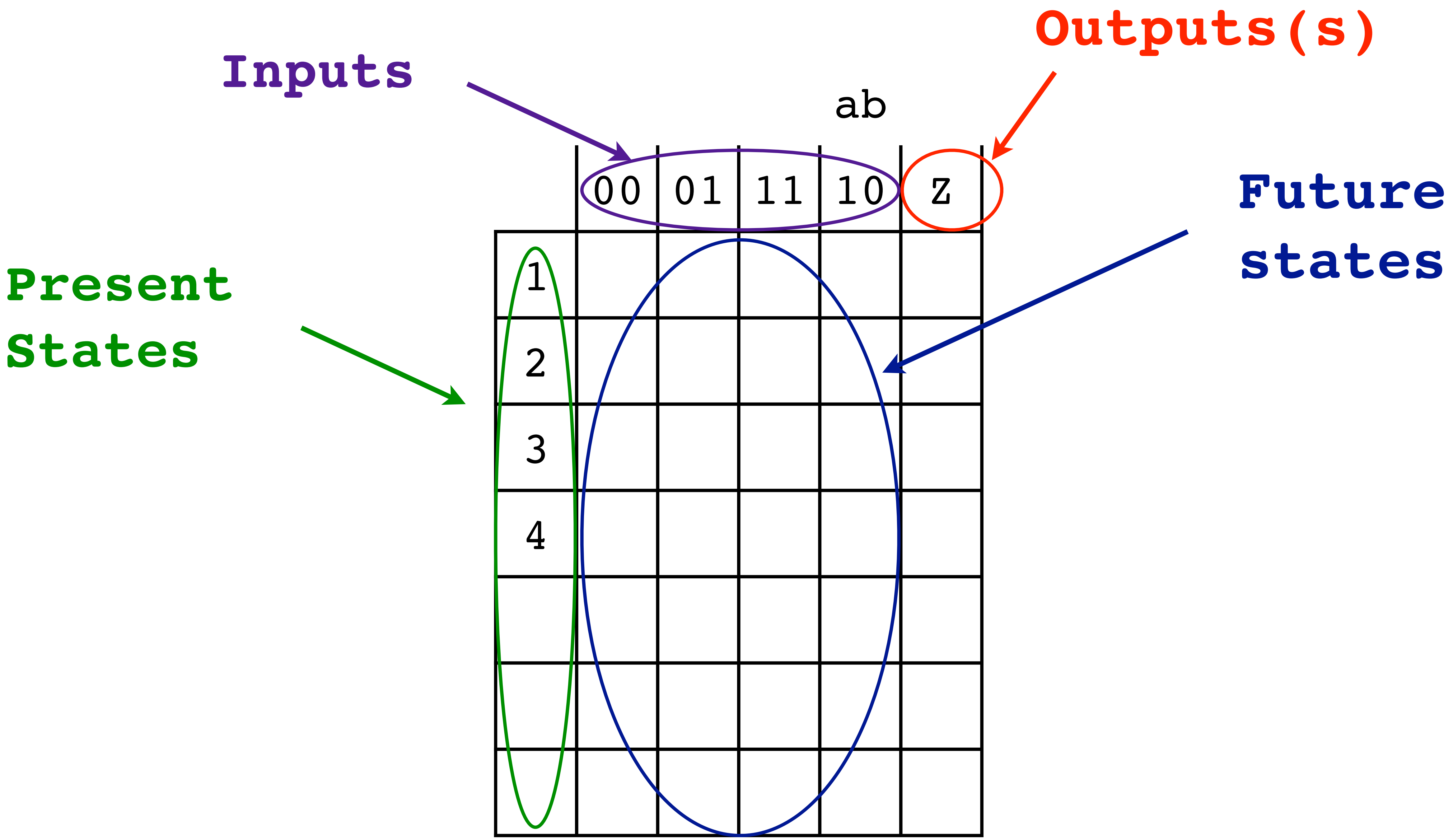
State tables

- Indicates the way the system evolves depending on the **present state** and **inputs**
- The number of lines in the table fixes the number of states
- First column indicates the present states
- Other columns indicate all possibilities of input variables (so their number is 2^n)
- **Stable** states are marked in **bold** (like the circles in state diagrams) → system remains in stable state as long as the input remains unchanged
- Transition state are used to go from one to another state(arcs in the state graphs)
- Transition take pace between Source & Destination states



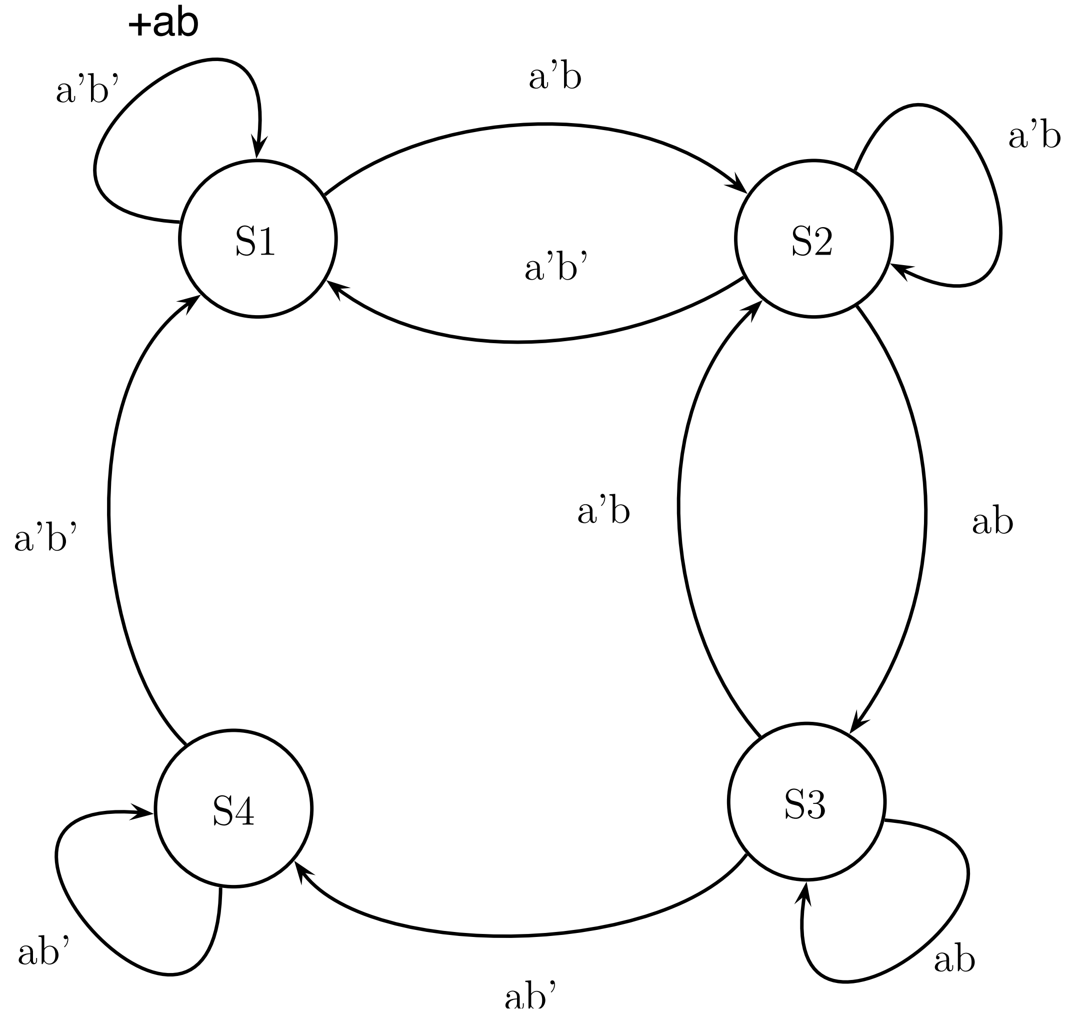
	00	01	11	10	Z
1	**1**	2			
2		**2**			
3					
4					

State tables



State graph to state table

We can easily switch from on to the other:



	ab				
	00	01	11	10	Z
1	**1**	2	**1**	-	
2	1	**2**	3	-	
3	-	2	**3**	4	
4	1	-	-	**4**	

Note that non-existing transitions
are marked using *don't cares* (e.g. S1 to S3)

How to read state table?

a. System is in state `1`, inputs are set to `00` → system **remains** in this state `1` as long as **inputs remain** `00`

b. Imagine now that ab change from 00 to 01
Alice (or worse Donald …) pushed the button!

c. System “**leaves**” the current state

d. We switch the column, so that this new situation corresponds to new input combination (we remain in the same line)

e. We read what is written in this new cell; this becomes now our **new destination**

f. New destination could be a stable state (system remains in that state) or a transition; if it is a transition, the process will continue until we reach a stable state (here state **4**)

	ab				
	00	01	11	10	Z
1	**1**	2			
2		3	**2**		
3		4	**3**		
4		**4**	3		

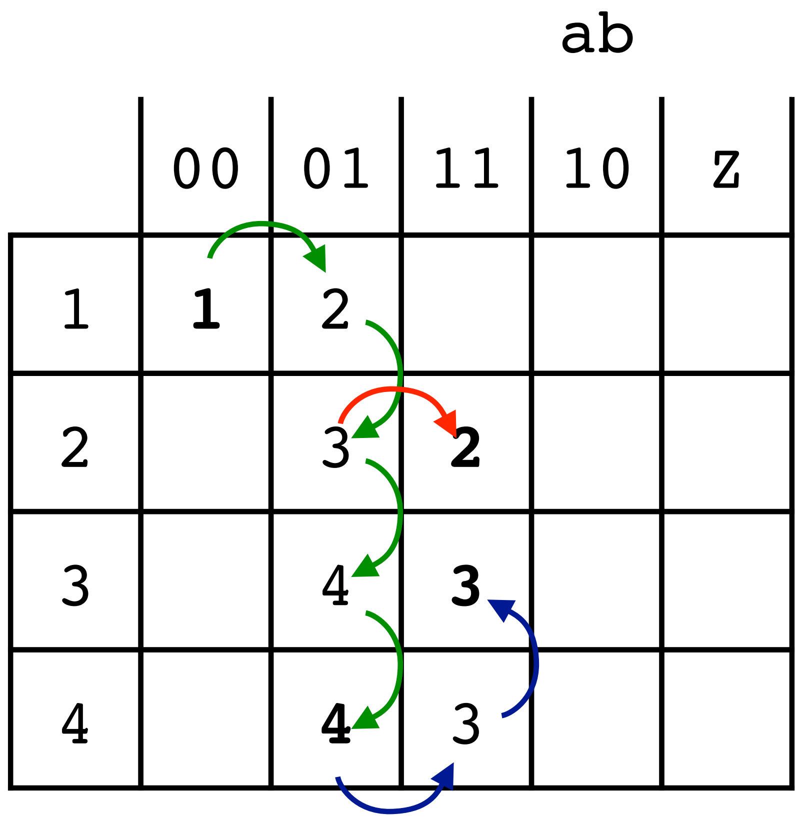
Important hypothesis for Alice

In a state table:

- The number of transitions between two stable states is not limited
- **It is mandatory that the inputs do not change during the transition (transitions are not instantaneous!)**
- If this hypothesis is not satisfied the system behaviour is not guaranteed any more

Example

- Imagine a transition from 1 to 4: so passing through 2, 3 et 4, that will be triggered by input sequence `00`, `01`, `11`
- Imagine that Alice pushed button `a` during transition 3, the system will end in state 2, which is stable (so it will remain there)
- For sequence `00`, `01`, `11` we ended up in `2` and not in `3`



	ab 00	01	11	10	Z
1	**1**	2			
2		3	**2**		
3		4	**3**		
4		**4**	3		