

ELEC-H-310
Digital electronics

Lecture04

Synthesis & optimisation of synchronous logic circuits

Dragomir MILOJEVIC
2021

Today

1. Sequential systems: classes et representation
2. State encoding
3. Sequential system output
4. Example of a very simple sequential system (1st example)
5. Optimisation of primitive state tables
6. D flip-flop synthesis (2nd example)
7. Different memory elements

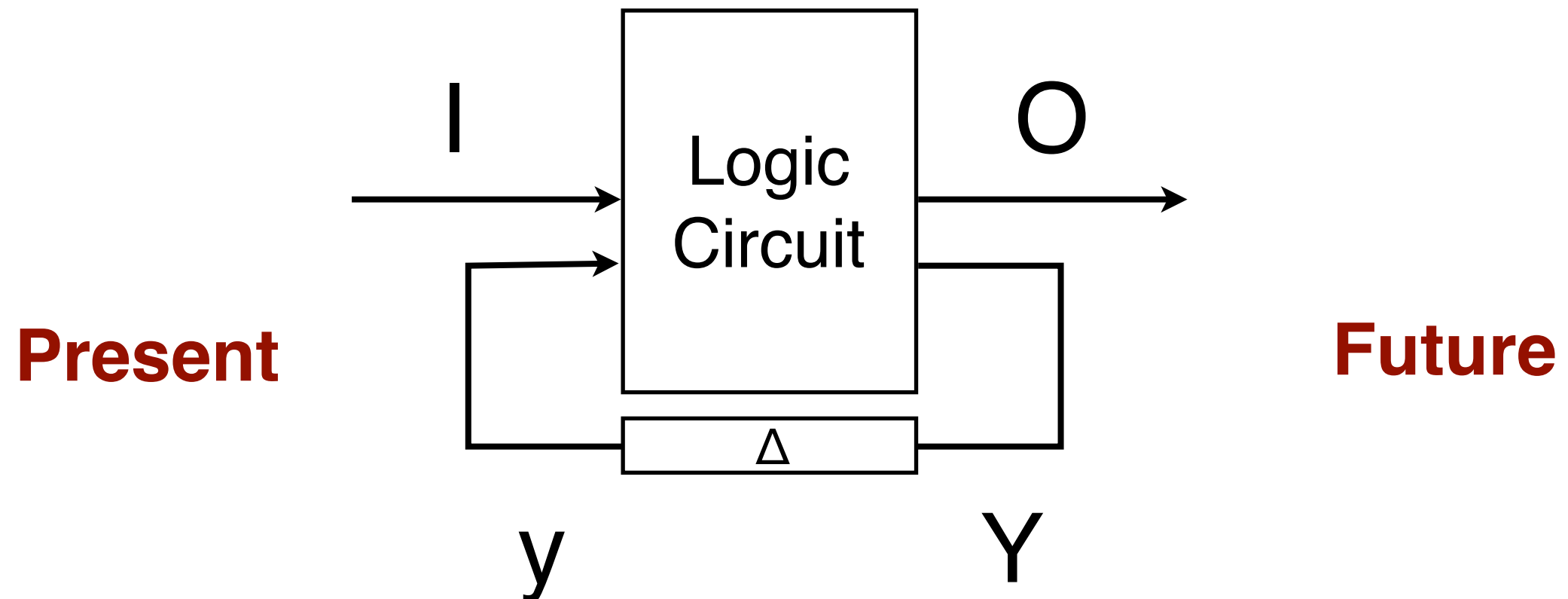
1. Sequential systems: classes et representation

Sequential systems — why and how?

- Combinatorial systems can't be used to control certain process
- Last time we have seen the reservoir problem: level of the liquid in the tank can not be used alone to control the valve that will bring the liquid (we can not make a difference between the state of a “full tank being emptied” and/or “empty tank being filled”)
- So, to compute the system output (i.e. the control of the valve) using the **inputs only** is not enough; we need something more ...
- Sequential systems introduce the notion of **system state**
- Current system state is somehow “stored”, “kept”, “remembered” in the system – there is a notion of **memory** in sequential digital logic systems
- In the example we need to remember if tank was full or empty (there are 2 distinct states)

Memory: present & future

- **State of the system** is materialised using a **feedback loop** which allows us to make a difference between the **present** and **the future**
- Feedback loop = time difference
- Time difference = **delay in the wire**, necessary to distinguish between y (**present state variable**) and Y (**future state variable**)

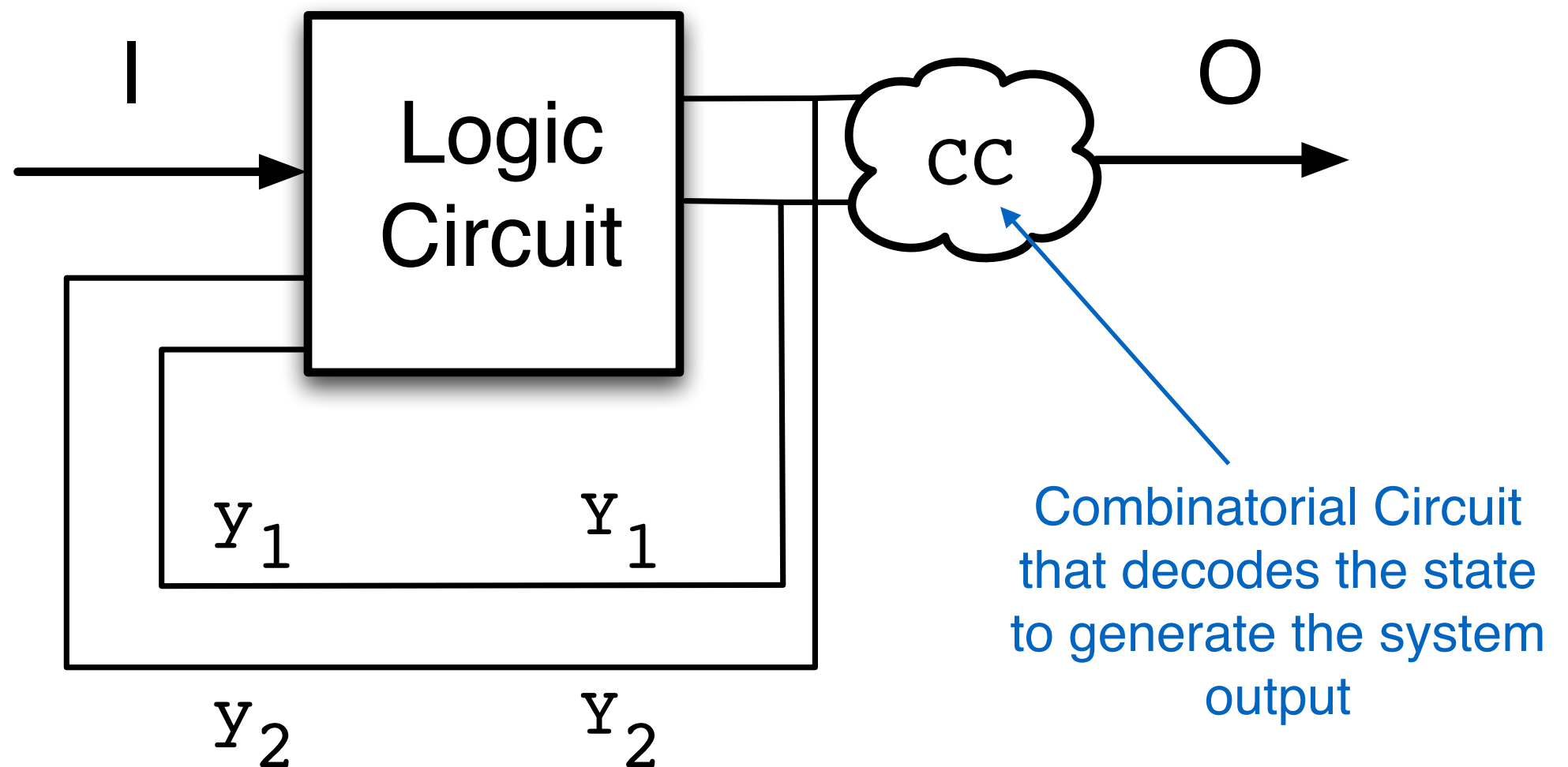


States & sequential logic systems

- System state (internal state) is not necessarily seen by the observer outside of the system
- After all, what counts is what the systems does, or what it produces at the output
- In sequential systems the output is computed as a **function of the current state AND eventually of the system inputs**
- Here “**eventually**” means that we have two different classes of sequential logic circuits:
 - **Moore machine** (not Gordon from Intel!) &
 - **Meally Machine**
- Today let's just point the difference between the two & focus on Moore
- We will see later what the other really mean, and why it is useful

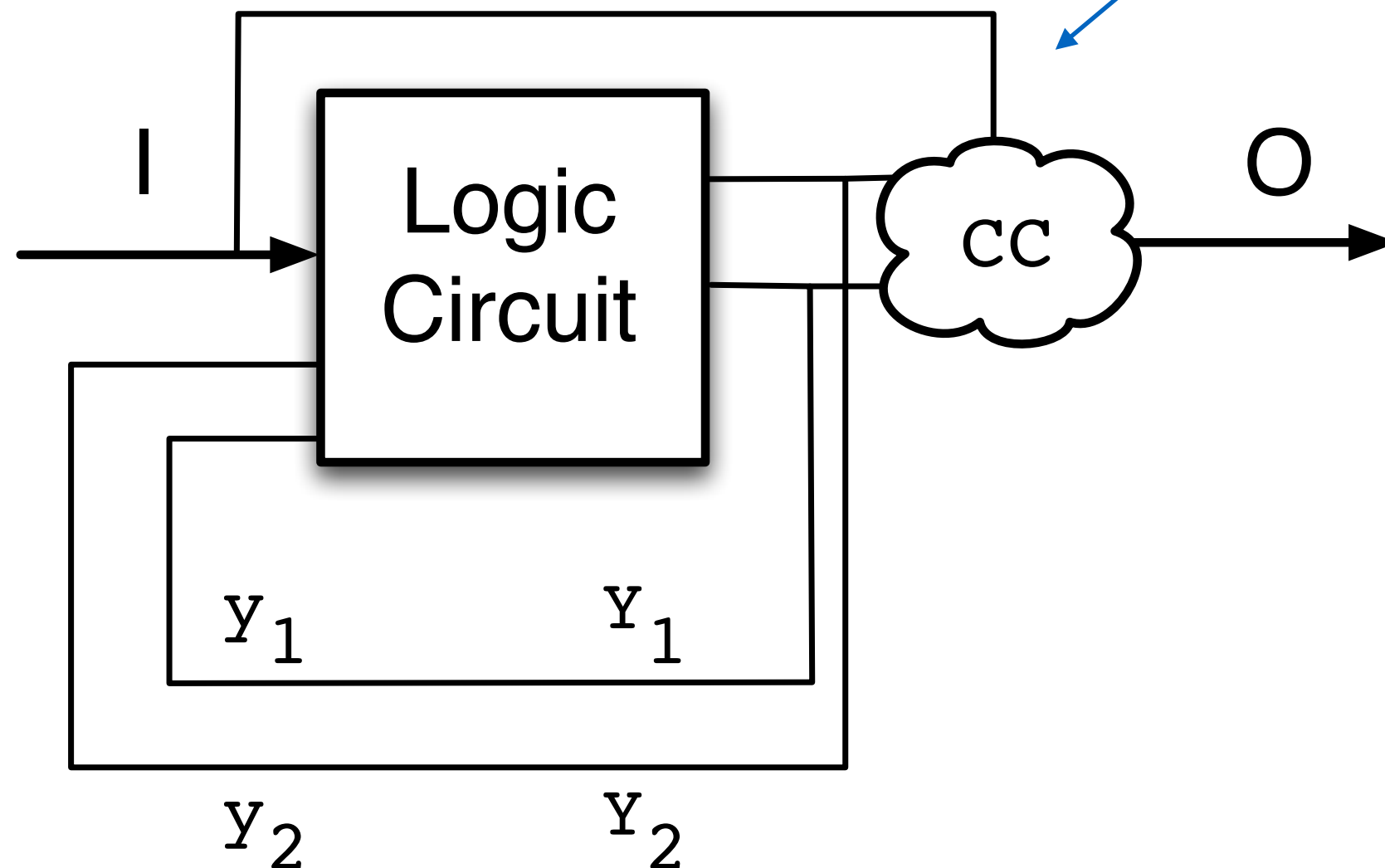
Moore Machine

Output (combinatorial) is function of **state variables** only;
(we focus on this type of machines for a moment)



Mealy machine

Output (combinatorial) is function of **state variables and INPUT**;
(we will see these machines later on)



Sequential circuits — representation & synthesis

- Representing sequential systems using:
 1. **State graphs** – for fun, not that useful, but nice to look at
 2. **State tables** – we will use these extensively
 3. **Logic equations** – important to build the circuit
- All these provide a formal specification of the automata evolution in time
- Note that the time here is seen more as an **event**, rather than some absolute value ...
- One can easily switch from one to another representation (you need one formal representation that is good) – this is an easy part
- Hard one is to derive the **initial system specification from verbal description** of the system behaviour (needs practice)

2. State encoding

State encoding

	ab				
	00	01	11	10	z
1	1	2	1	–	
2	1	2	3	–	
3	–	2	3	4	
4	1	–	–	4	

- State table showed last time used **decimal numbers** to represent states
- But it could be anything: letters, symbols, etc.
- As long as the future predicted exists as the present (whatever we find in the table should exist in the column representing present state)
- Logic circuits use binary: $\{0, 1, -\}$
- To go from one (initial state table) to the other we need to **encode states** (encoded table)

To perform state encoding we need to encode n states:

$$\log_2 n$$

each bit is a state **variable**

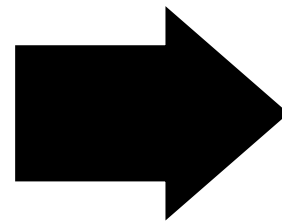
State encoding example

- System has 4 states
- To each state we attribute a binary code
- Let's pick an arbitrary (first to come) code first:

$1 \rightarrow 00 ; 2 \rightarrow 01 ; 3 \rightarrow 11 ; 4 \rightarrow 10$

State table

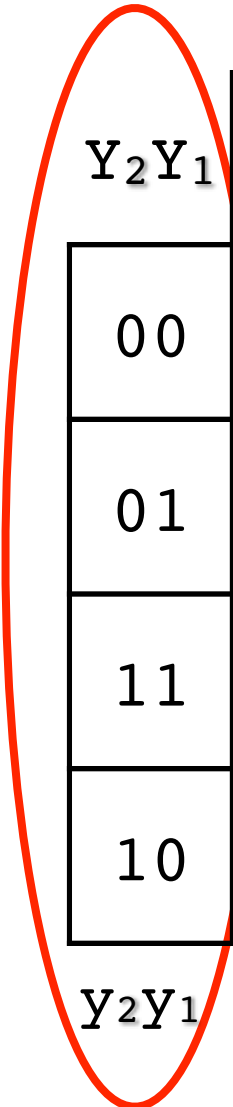

	ab				
	00	01	11	10	z
1	1	2	1	–	0
2	1	2	3	–	0
3	–	2	3	4	1
4	1	–	–	4	1



Encoded state table

	ab				
	00	01	11	10	z
00	00	01	00	–	0
01	00	01	11	–	0
11	–	01	11	10	1
10	00	–	–	10	1

Encoded state table



Y_2Y_1	ab				z
	00	01	11	10	
00	00	01	00	–	0
01	00	01	11	–	0
11	–	01	11	10	1
10	00	–	–	10	1

Y_2Y_1

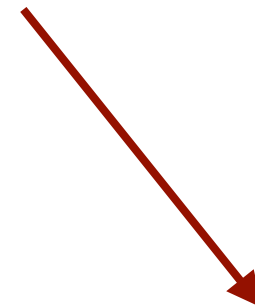
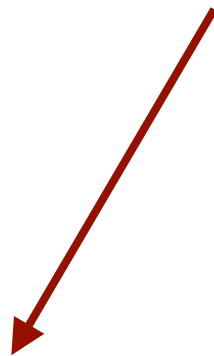
- States are encoded: each bit of a **code** is a separate, dedicated **logic function** to become a dedicated feedback loop
- For the example: 4 states, 2 state variables (y_2y_1), 2 logic functions, 2 different circuits (with 2 different feedback loops)
- Note the difference between y_2y_1 et Y_2Y_1 (lower & upper case)
- Lower case refers to **present**
- Upper case to the **future**

State table is like a crystal ball!

(give it present and inputs and it will tell you the future)

State table with 4 states gives 2 logic equations

	ab				
Y_2Y_1	00	01	11	10	z
00	00	01	00	–	0
01	00	01	11	–	0
11	–	01	11	10	1
10	00	–	–	10	1



Y_2	00	01	11	10
00	0	0	0	–
01	0	0	1	–
11	–	0	1	1
10	0	–	–	1

Y_1	00	01	11	10
00	0	1	0	–
01	0	1	1	–
11	–	1	1	0
10	0	–	–	0

Logic optimisation & final expressions

	ab			
Y ₂	00	01	11	10
00	0	0	0	-
01	0	0	1	-
11	-	0	1	1
10	0	-	-	1

Y₂Y₁

$$Y_2 = ab' + y_1a$$

	ab			
Y ₁	00	01	11	10
00	0	1	0	-
01	0	1	1	-
11	-	1	1	0
10	0	-	-	0

Y₂Y₁

$$Y_1 = a'b + y_1b$$

3. Sequential system output

Simple decoder

- We look only at states that generate $Z=1$

	ab				
	00	01	11	10	z
00	00	01	00	–	0
01	00	01	11	–	0
11	–	01	11	10	1
10	00	–	–	10	1

y_2y_1

$$Z = y_1y_2 + y_1'y_2$$

- This is not great: what will happen if we have arbitrary number of transitions?
- Decoder will follow transitions ... this will generate glitches on output!

Output of a sequential system

- Two types of outputs: for **stable states** and **transitions**
- For stable states we have this information in the state table itself, we know what the output value should be: we simply copy those
- For transitions (grey cells in the tables below): we decide on the value, **but we want to minimise the total number of transitions to only one**


	ab				z
	00	01	11	10	
00	00	01	00	–	0
01	00	01	11	–	0
11	–	01	11	10	1
10	00	–	–	10	1

	ab				z
	00	01	11	10	
00	0		0	–	
01		0		–	
11	–		1		
10		–	–	1	

$y_2 y_1$

Output function: rule on transitions

- In other words: if there should be a transition at the output, **we can allow only one such change at a time !**
- We can analyse all possibilities for a single transition between 2 stable states
- One state is called **departure** (from where), the other is **destination**



Departure state	0	1	0	1
Transition	0	1	–	–
Arrival state	0	1	1	0

Same:

We need to
maintain same
value

Different:

We can put a
don't care

Output function: final synthesis

- Presence of extra transitions enables better output logic function **simplification** (presence of *don't cares*)
- Output values for transitions (grey in the table below) have been fixed applying the rule from the previous slide

	ab				
	00	01	11	10	z
00	00	01	00	–	0
01	00	01	11	–	0
11	–	01	11	10	1
10	00	–	–	10	1

	ab				
z	00	01	11	10	
00	0	0	0	–	
01	0	0	–	–	
11	–	–	1	1	
10	–	–	–	1	

$y_2 y_1$

$$z = y_2$$

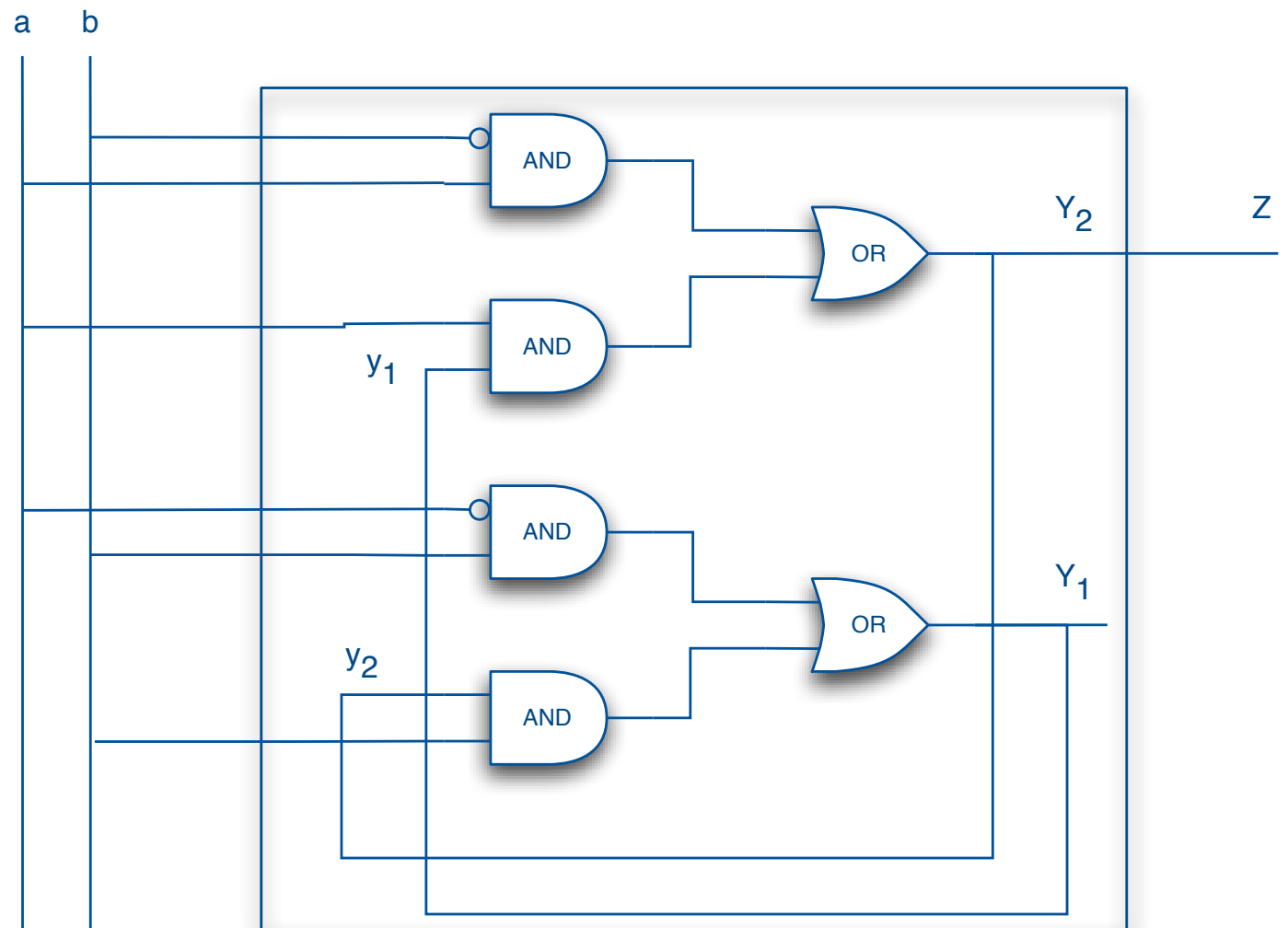
Logic diagram of a sequential circuit

With all Boolean expressions, we can now draw the circuit;
Attention ! Feedback logic functions: make a difference between the present state (y_i) and future states (Y_i)

$$Y_2 = ab' + y_1a$$

$$Y_1 = a'b + y_2b$$

$$Z = y_2$$



4. Complete example of a (very simple) sequential system

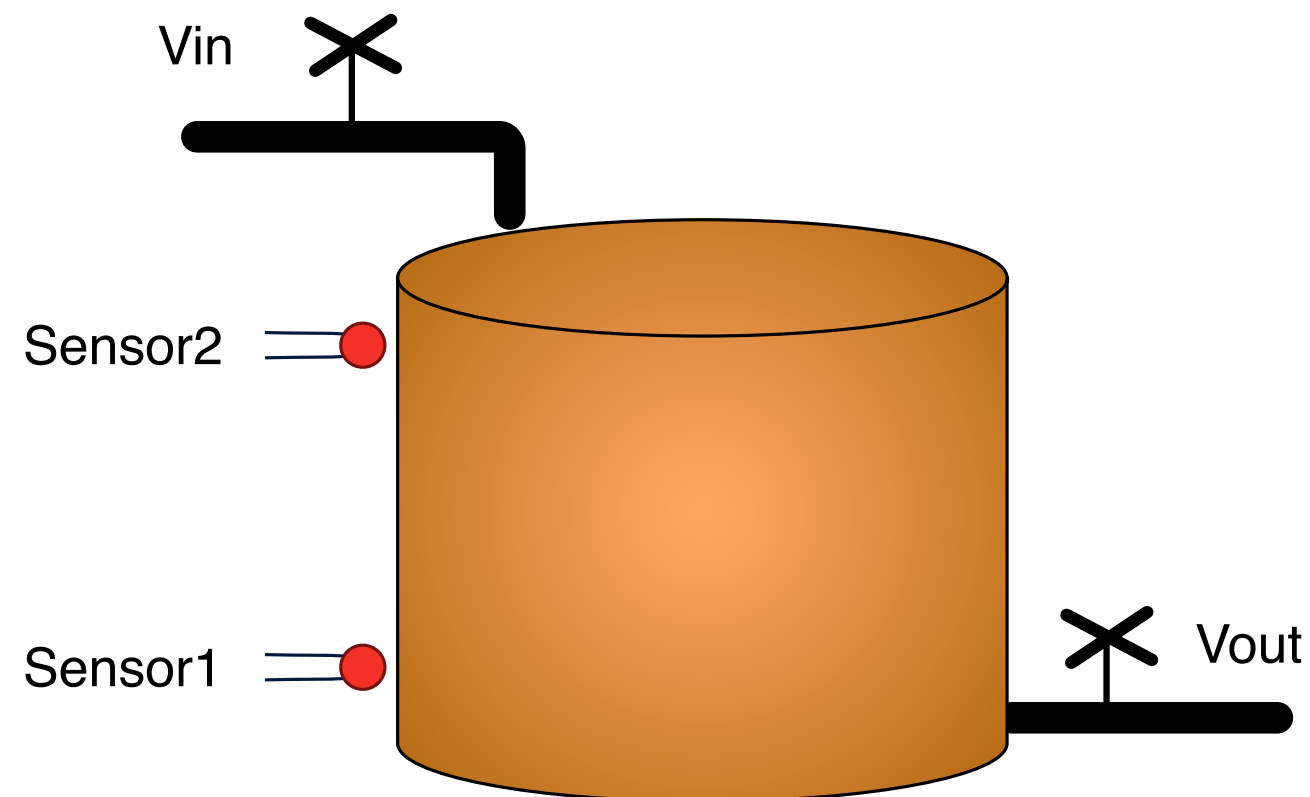
Synthesis steps

For the example of the reservoir:

1. Write state table
2. Draw state graph
3. Encode the states
4. Derive K-Maps corresponding to feedback logic functions
5. Derive the output for transitions
6. Enumerate all logic functions
7. Draw the circuit diagram

1. State table

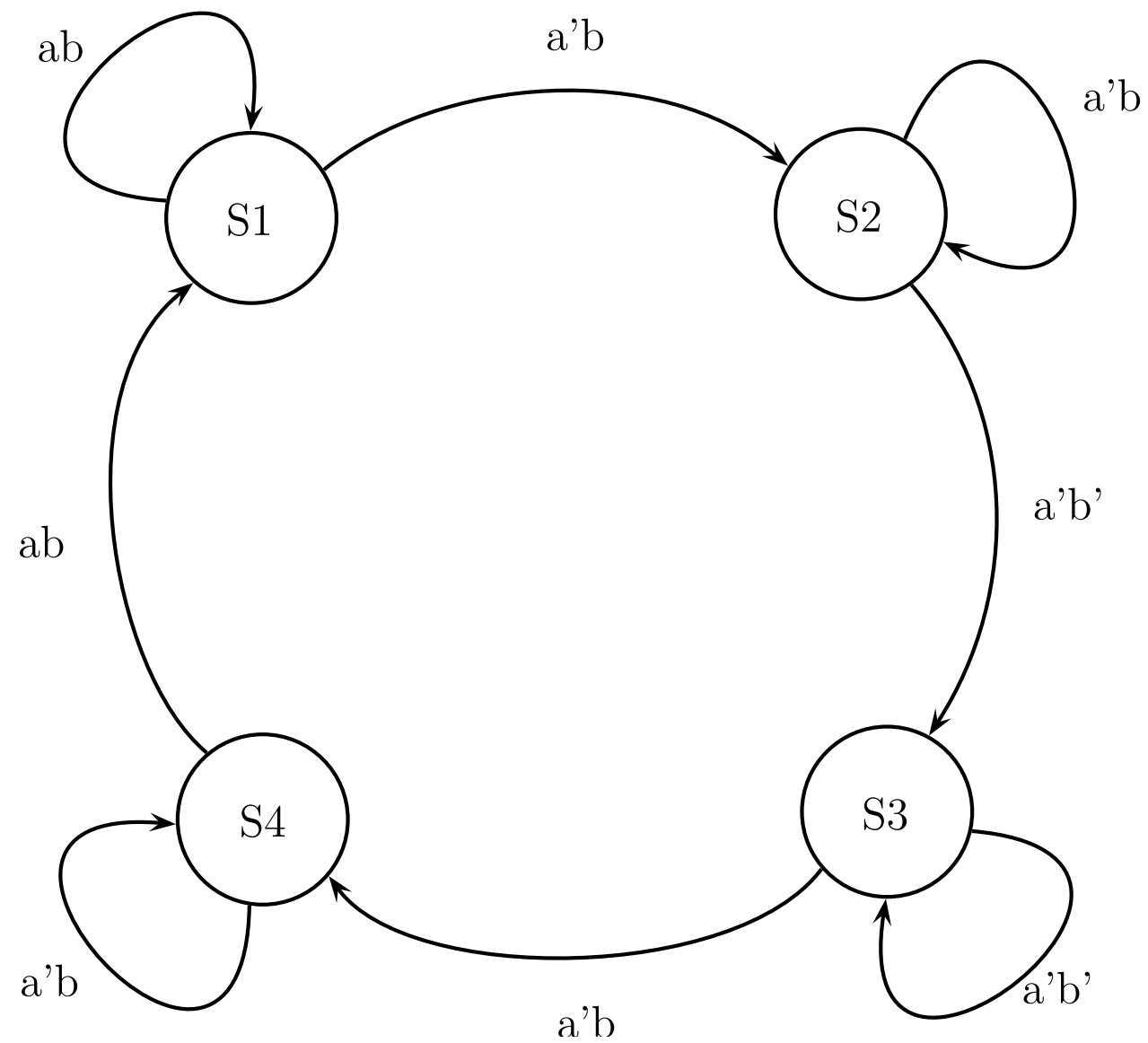
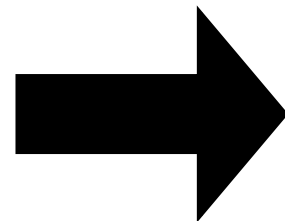
- We suppose for initial state a **full reservoir** (state 1 with $ab=11$)
- From there the system can be only emptied (input $ab=01$) to go in 2
- From 2 the system could either become full again, or further be emptied (this is a new state 3); note that instantaneous inversion of inputs is not possible in this case ($ab=01$ to become $ab=10$)



	ab				
	00	01	11	10	Z
1	-	2	1	-	0
2	3	2	1	-	0
3	3	4	-	-	1
4	3	4	1	-	1

2. State graph

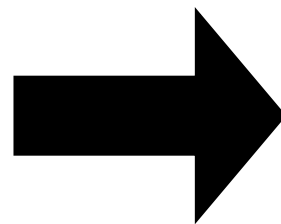
	ab				
	00	01	11	10	Z
1	-	2	1	-	0
2	3	2	1	-	0
3	3	4	-	-	1
4	3	4	1	-	1



3. State encoding

We make the following choice: $1 \rightarrow 00$; $2 \rightarrow 01$; $3 \rightarrow 11$; $4 \rightarrow 10$
(more on this later on ... take the choice free for now)

	ab				
	00	01	11	10	z
1	–	2	1	–	0
2	3	2	1	–	0
3	3	4	–	–	1
4	3	4	1	–	1



	ab				
	00	01	11	10	z
00	–	01	00	–	0
01	11	01	00	–	0
11	11	10	–	–	1
10	11	10	00	–	1

4. Feedbacks logic functions

	ab			
Y ₂	00	01	11	10
00	-	0	0	-
01	1	0	0	-
11	1	1	-	-
10	1	1	0	-

$$Y_2 = a'b' + y_2a'$$

	ab				
	00	01	11	10	z
00	-	01	00	-	0
01	11	01	00	-	0
11	11	10	-	-	1
10	11	10	0	-	1

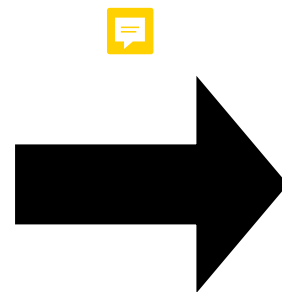
	ab			
Y ₁	00	01	11	10
00	-	1	0	-
01	1	1	0	-
11	1	0	-	-
10	1	0	0	-

$$Y_1 = a'b' + y_2'a'$$

5. Outputs

Values of the outputs are fixed for stable states;
for transitions: **one change only is allowed!**

	ab				
	00	01	11	10	z
1	–	2	1	–	0
2	3	2	1	–	0
3	3	4	–	–	1
4	3	4	1	–	1



	ab				
z	00	01	11	10	
0	–	0	0	–	
1	–	0	0	–	
11	1	1	–	–	
10	1	1	–	–	

6. All logic functions (feedback & output)

	ab			
Y ₂	00	01	11	10
00	-	0	0	-
01	1	0	0	-
11	1	1	-	-
10	1	1	0	-

$$Y_2 = a'b' + y_2a'$$

	ab			
Y ₁	00	01	11	10
00	-	1	0	-
01	1	1	0	-
11	1	0	-	-
10	1	0	0	-

$$Y_1 = a'b' + y_2'a'$$

	ab			
Z	00	01	11	10
00	-	0	0	-
01	-	0	0	-
11	1	1	-	-
10	1	1	-	-

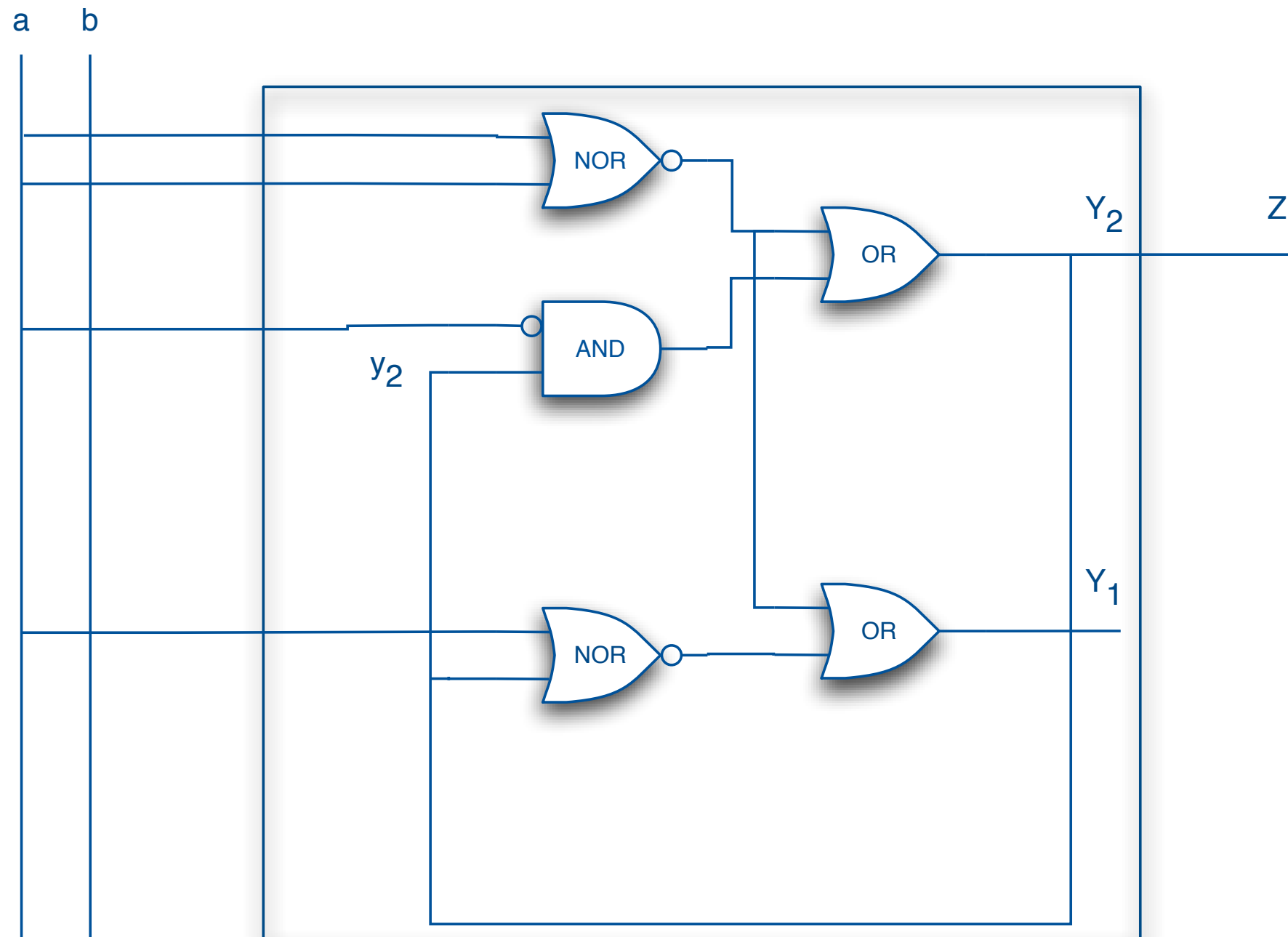
$$Z = y_2$$

7. Final logic circuit

$$Y_1 = a' b' + y_2' a'$$

$$Y_2 = a' b' + y_2 a'$$

$$Z = y_2$$



3. Optimisation of primitive state tables

Optimisation & implementation

Sequential system synthesis in three steps:

1. Primitive state table (Huffman Table)

- ✦ Using verbal system specification (or state graphs for example) we first derive **Primitive State Table** (AKA Huffman table)
- ✦ **Primitive State Table** — we expect one **stable state per line of the table**

2. State encoding

- ✦ Chose unique binary code per state in the system
- ✦ Few codes are possible, for now we pick the first code that we could imagine
- ✦ Code choice arbitrary for a moment; not going to be the case in the future!

3. Logic Equations

- ✦ From encoded state table we derive logic expression in their optimised form (using K-Maps or Quine-McCluskey)

Optimisation & implementation

- Final logic circuit complexity after synthesis will be influenced by the **complexity** (i.e. **size**) of the initial state table
- Encoding: mapping of $\log_2 n$ bits of the binary code to n states
- Each bit here represents:
 - one logic **function**,
 - one **memory element** (delay)
- By reducing the number of states in the system we can simplify the corresponding logic circuit (save resources means cheaper, faster, less power hungry circuit)
- This is similar to optimisation of logic functions used for combinatorial systems

Reducing the number of states

- Optimising the state table is possible thanks to the **notion of state equivalence**
- Two (stable) states are equivalent iff:
 1. they produce the **same output** (we are in the case of Moore Machine, output depends on the state only)

AND

2. for **all** combinations of inputs, all **future** states are the same or **equivalents** (two states will have the same future)
- This can be extended to three and more states; in that case all pairs of states need to be equivalents

Equivalence: identical states and state fusion

- **Identical states** — then 2 (or more) stable states are at the same place (for the same input combination)

	ab				
	00	01	11	10	z
1	1	2	3	5	0
6	6	2	3	5	0

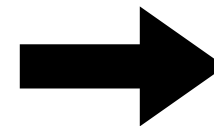
- **State fusion** — stable states are found at different input combinations, but will generate the same future

	ab				
	00	01	11	10	
1	1	2	3	7	1
7	1	2	3	7	1

State table after optimisation

**Identical
state**

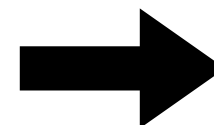
	ab				
	00	01	11	10	Z
1	1	2	3	5	0
6	6	2	3	5	0



	ab				
	00	01	11	10	Z
1	1	2	3	5	0

**State
fusion**

	ab				
	00	01	11	10	Z
1	1	2	3	7	1
7	1	2	3	7	1



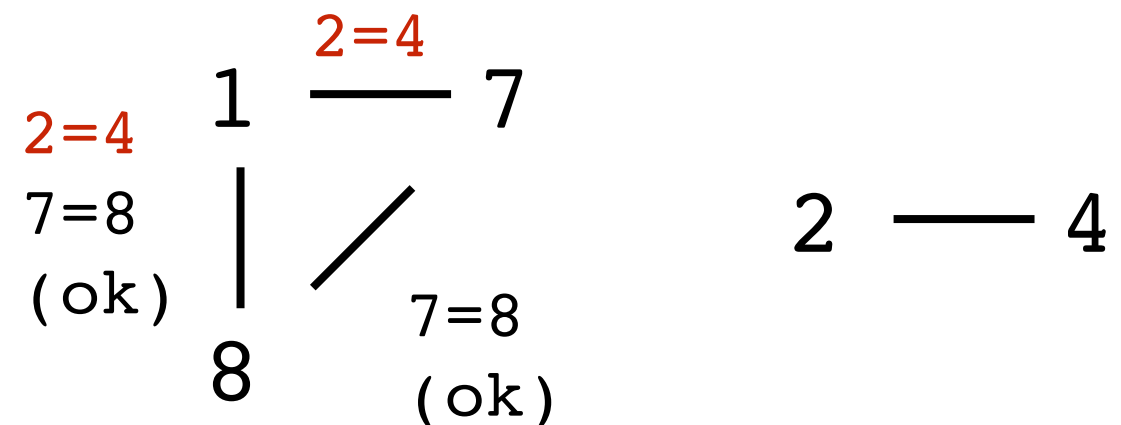
	ab				
	00	01	11	10	Z
1	1	2	3	1	1

In a given state table, finding all identical states and performing state fusion will yield state table **OPTIMISATION**

Fusing more than 2 states

- It is possible to fuse two or more states, if we can fuse all **pairs of states, two-by-two**; this can be verified using **fusion graphs**
- These graphs show **fusion conditions** that need to be checked before
- Let's take an example:

	ab				
	00	01	11	10	Z
1	1	2	8	7	1
7	1	4	8	7	1
8	1	4	8	8	1



- To be able to fuse **1,7 and 8**: we need to fuse **7-8** and **2-4**!
- These other states are not shown here

Systematic search of equivalences

- Draw a table with $(n)(n-1)/2$ cells; where n is the number of states in the initial state table; we call this table: **equivalences conditions table**
- We consider that each state is equivalent to itself — combination of n elements **without repetitions**
- Each table cell contains the condition of the potential equivalence between two stable states, and could have three options:
 - **Not equivalent** — we note this with and **x**; in the case of Moore machine any two pair of states with different outputs will be marked as such; 1st pass is always only about comparing the output value
 - **Equivalent** — we mark this OK; these two states can be fused without any conditions (this does not necessarily mean that we do fuse them)
 - **Conditional** — some other pair of states need to be equivalent in order for these two to be equivalent

Systematic search of equivalences

Example – For 11 state table the **equivalences conditions table**:

2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
	1	2	3	4	5	6	7	8	9	10

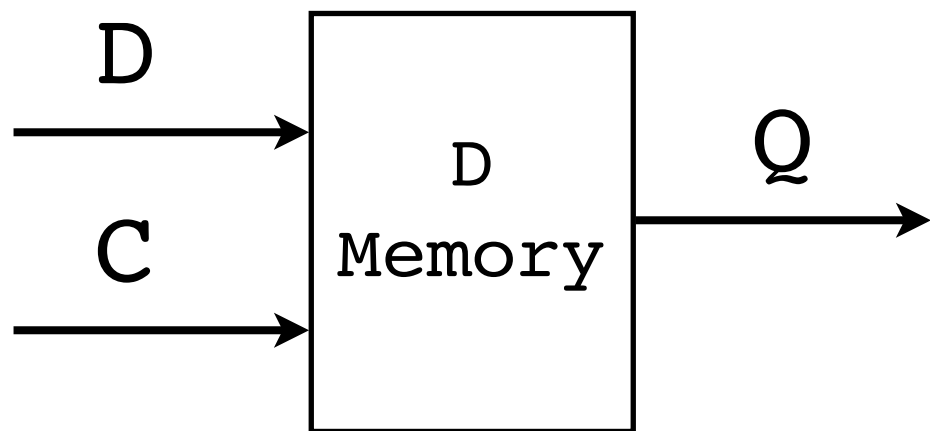
As for use & applications let's see a concrete example

4. D flip-flop synthesis (2nd example)

Problem

Establish **primitive state table**, find **optimised state table**, write encoded state table & derive logic equations for **D – memory element**

D – memory

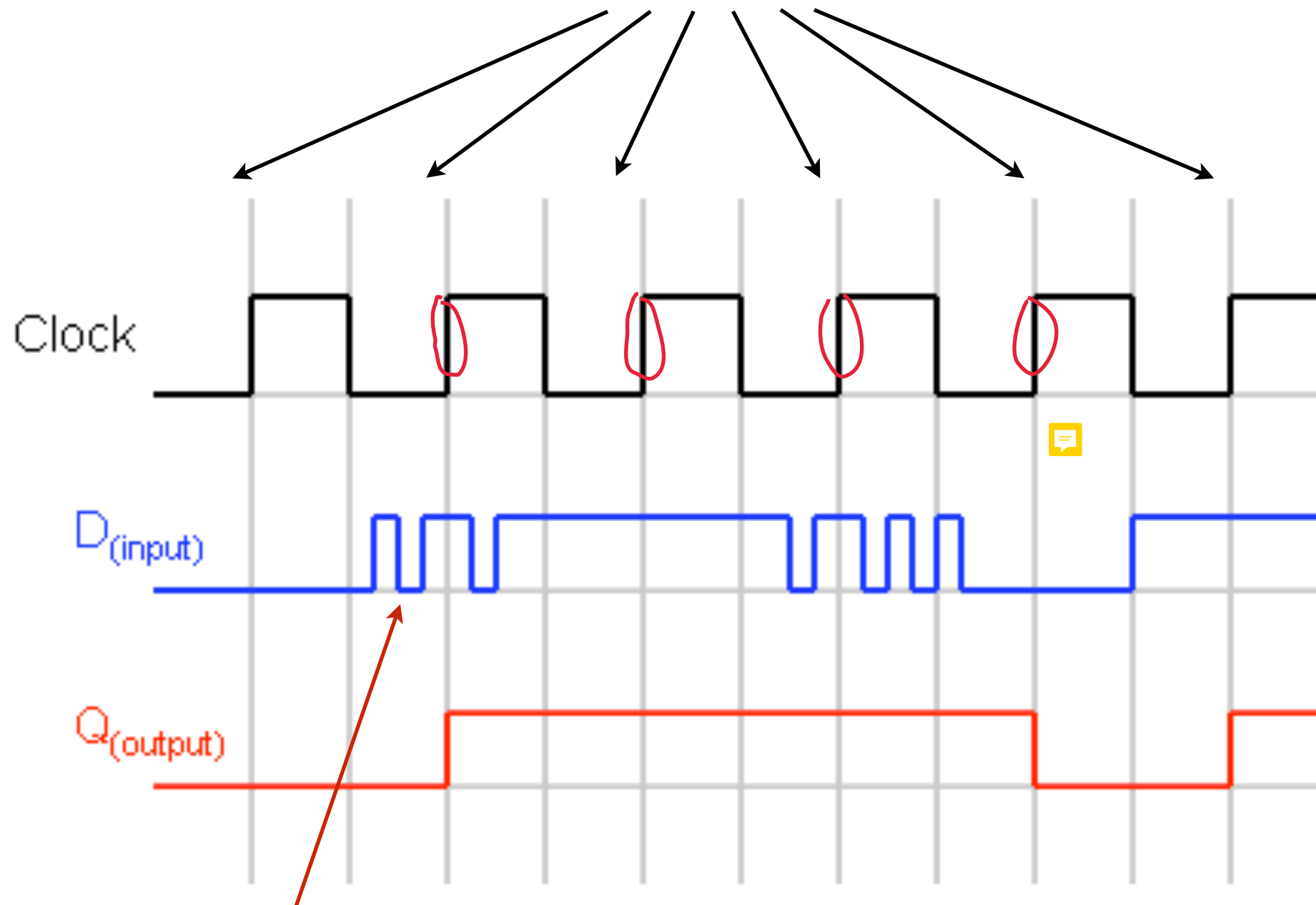
- Two inputs: D (*data*) & C (*control*)
 - One output Q
 - Output value controlled by D **and** C
- 
- Two variants:
 - **Latches** – sensitive to the **level** of C (we don't like these)
 - **Flip-flop** – sensitive to the **transition** of C (0 to 1 or inverse)

Specification of the **edge triggered D-FF**

- Assume we have a periodic control signal for C — clock that controls the D-FF device
- Output Q takes the value of D **only** when **rising edge of** Clock
- Rising edge is the **transition from 0 to 1** (falling edge from 1 to 0)
- In between two rising edges, the old value of D is maintained in the memory (any variation of the input is ignored)
- If there is a **simultaneous** variation of the C and D (rising edge of C and value of D that switch from 0 to 1 or inverse), Q is taking the old value of D, the one just before the rising edge
- Whatever else happens at the input is ignored, the output maintains the previous value (memorisation)

D Flip-Flop behaviour— timing diagram

Write to memory only at rising edge of a clock signal



(input values outside these moments are ignored)

D Flip-Flop — Primitive state table

Let's consider the evolution that will set the output to 0 

1. Let's start with an **initial state** — we can do that, and make it so that the system always “wakes up” in this initial state — we call this state **1** (note the bold):

we already saved 0 ($Q=0$) and there is no change in inputs ($CD=00$ and remains as such)

2. We consider that D is first set to 1 (D moves from 0 to 1) **and before** C is switched on — state **2**

Output is kept to 0

3. And then C switches on — state **4**

Output is set to 1 when the following input sequence is seen for CD: $CD=00, 01, 11$

(i.e. machine moves between states **1, 2 et 4**)

		CD				
		00	01	11	10	Q
1	1		2			0
2	1		2	4		0
3						0
4				4		1
5						0
						0

D Flip-Flop — Primitive state table

Another evolution is possible from state 1

1. Imagine that C switched on before D from state 1 (we have $CD=00, 10$) → this is a new situation (state) that we call 3

C could switch back to (and again), this is ignored as long as D doesn't change (**red loop**)

2. If D turns on now, FF should ignore this input change since it arrived **after** the rising edge of C (this already occurred)
3. We created state 5, different from 4 because the output should be kept to 0, since the input sequence is not good

	CD				Q
	00	01	11	10	
1	1	2		3	0
2	1	2	4	3	0
3			5	3	0
4			4		1
5			5		0
					0

Complete primitive state table

Memorising (and keeping memorised) value of 1

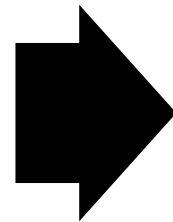
1. Initially $CD=00$
2. CD input sequence is: $00, 01, 11$
3. Output Q set to 1
(system in state 4)
4. Then imagine $C=0$
5. Then whatever comes at input D ,
output Q will keep the value of 1 —
succession of states $7, 9, 7, 9, \dots$

	CD				
	00	01	11	10	Q
1	1	2	5	3	0
2	1	2	4	3	0
3	6	2	5	3	0
4	9	7	4	8	1
5	6	2	5	3	0
6	6	2	5	3	0
7	9	7	4	8	1
8	9	7	4	8	1
9	9	7	5	10	1
10	6	2	5	10	0

DFF – State equivalence table **1st pass**

Primitive state table opt. – Moore Machine, conditions on outputs

	CD				
	00	01	11	10	Q
1	1	2	5	3	0
2	1	2	4	3	0
3	6	2	5	3	0
4	9	7	4	8	1
5	6	2	5	3	0
6	6	2	5	3	0
7	9	7	4	8	1
8	9	7	4	8	1
9	9	7	5	10	1
10	6	2	5	10	0



2										
3										
4	X	X	X							
5					X					
6					X					
7	X	X	X			X	X			
8	X	X	X			X	X			
9	X	X	X			X	X			
10					X			X	X	X
	1	2	3	4	5	6	7	8	9	

Different outputs
are marked as x

DFF – State equivalence table 2nd pass

	CD				Q
	00	01	11	10	
1	1	2	5	3	0
2	1	2	4	8	0
3	6	2	5	3	0
4	9	7	4	8	1
5	6	2	5	3	0
6	6	2	5	3	0
7	9	7	4	8	1
8	9	7	4	8	1
9	9	7	5	10	1
10	6	2	5	10	0

2	4-5 3-8								
3	1-6	1-6 4-5 3-8							
4	X	X	X						
5	1-6	1-6 4-5 3-8	OK	X					
6	OK	1-6 4-5 3-8	OK	X	OK				
7	X	X	X	OK	X	X			
8	X	X	X	OK	X	X	OK		
9	X	X	X	8-10 4-5	X	X	4-5 8-10	4-5 8-10	
10	1-6 3-10	1-6 4-5 3-10	OK	X	3-10	3-10	X	X	X
	1	2	3	4	5	6	7	8	9

We add conditions for state fusion

DFF – State equivalence table 3rd pass

2	4-5 3-8	4≠5 results all red cells								2	4-5 3-8	1=6, 3=10 results all (light) green cells							
3	1-6	1-6 4-5 3-8								3	1-6	1-6 4-5 3-8							
4	X	X	X							4	X	X	X						
5	1-6	1-6 4-5 3-8	OK	X						5	1-6	1-6 4-5 3-8	OK	X					
6	OK	1-6 4-5 3-8	OK	X	OK					6	OK	1-6 4-5 3-8	OK	X	OK				
7	X	X	X	OK	X	X				7	X	X	X	OK	X	X			
8	X	X	X	OK	X	X	OK			8	X	X	X	OK	X	X	OK		
9	X	X	X	8-10 4-5	X	X	4-5 8-10	4-5 8-10			9	X	X	X	8-10 4-5	X	X	4-5 8-10	4-5 8-10
10	1-6 3-10	1-6 4-5 3-10	OK	X	3-10	3-10	X	X	X	10	1-6 3-10	1-6 4-5 3-10	OK	X	3-10	3-10	X	X	X
	1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9

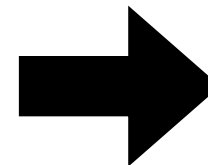
New state table after removing equivalent states

Equivalent states

1, 6 → 1

3, 10 → 3

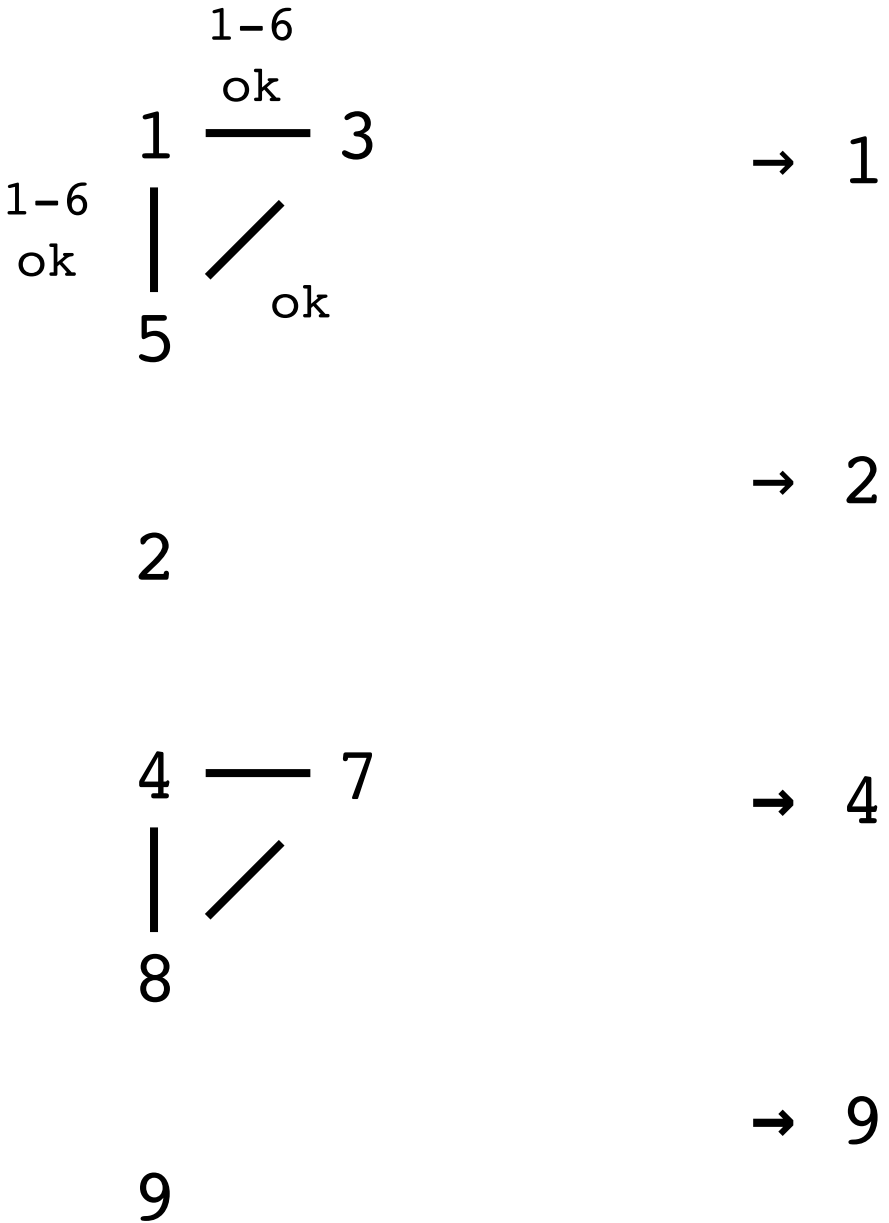
	CD				Q
	00	01	11	10	Q
1	1	2	5	3	0
2	1	2	4	8	0
3	6	2	5	3	0
4	9	7	4	8	1
5	6	2	5	3	0
6	6	2	5	3	0
7	9	7	4	8	1
8	9	7	4	8	1
9	9	7	5	10	1
10	6	2	5	10	0



	CD				Q
	00	01	11	10	Q
1	1	2	5	3	0
2	1	2	4	8	0
3	1	2	5	3	0
4	9	7	4	8	1
5	1	2	5	3	0
7	9	7	4	8	1
8	9	7	4	8	1
9	9	7	5	3	1

State fusion graphs

2	4-5 3-8								
3	1-6	1-6 4-5 3-8							
4	X	X	X						
5	1-6	1-6 4-5 3-8	OK	X					
6	OK	1-6 4-5 3-8	OK	X	OK				
7	X	X	X	OK	X	X			
8	X	X	X	OK	X	X	OK		
9	X	X	X	8-10 4-5	X	X	4-5 8-10	4-5 8-10	
10	1-6 3-10	1-6 4-5 3-10	OK	X	3-10	3-10	X	X	X
	1	2	3	4	5	6	7	8	9



Re-writing state table after optimisation

Equivalent states

1, 6 → 1

3, 10 → 3

State fusion

1, 3, 5 → 1

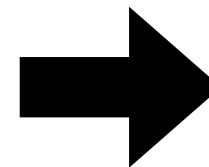
2 → 2

4, 7, 8 → 4

9 → 9

	CD				
	00	01	11	10	Q
1	1	2	5	3	0
2	1	2	4	8	0
3	1	2	5	3	0
4	9	7	4	8	1
5	1	2	5	3	0
7	9	7	4	8	1
8	9	7	4	8	1
9	9	7	5	3	1

Initial state table



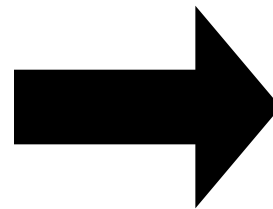
	CD				
	00	01	11	10	Q
1	1	2	1	1	0
2	1	2	4	4	0
4	9	4	4	4	1
9	9	4	1	1	1

Final state table

Final state table

We pick new codes (ordered): 1,2,3 et 4

	CD				
	00	01	11	10	Q
1	1	2	1	1	0
2	1	2	4	4	0
4	9	4	4	4	1
9	9	4	1	1	1



	CD				
	00	01	11	10	Q
1	1	2	1	1	0
2	1	2	3	3	0
3	4	3	3	3	1
4	4	3	1	1	1

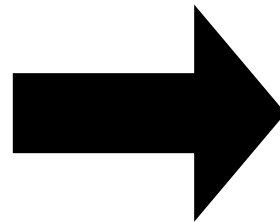
Optimisation: we moved from 10 to 4 states, meaning 2 state variables instead of 4. This is a **significantly less** resources to do exactly the same thing!

Encoded state table

We can (still!) chose codes arbitrarily, so say:

$1 \rightarrow 00, 2 \rightarrow 01, 3 \rightarrow 11, 4 \rightarrow 10$

CD					
	00	01	11	10	Q
1	1	2	1	1	0
2	1	2	3	3	0
3	4	3	3	3	1
4	4	3	1	1	1



CD					
	00	01	11	10	Q
00	00	01	00	00	0
01	00	01	11	11	0
11	10	11	11	11	1
10	10	11	00	00	1

K-Maps & logic equations

Optimised state table implementation

	CD			
Y ₂	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	1	1	1	1
10	1	1	0	0

Y₂Y₁

$$Y_2 = y_1 C' + y_2 C$$

	00	01	11	10
00	00	01	00	00
01	00	01	11	11
11	10	11	11	11
10	10	11	00	00

	CD			
Y ₁	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	0	1	1	1
10	0	1	0	0

Y₂Y₁

$$Y_1 = C' D + y_1 C$$

5. Different memory elements

Flip-flops

- One or two inputs that control the value stored in the FF
- Presence (or not) of a periodic control signal (Clock)
- Normal output (Q) and inverted output (Q')
- Depending on the FF type
 - **SR (Set/Reset)** – When $S=1$, then output $Q=1$, no matter the previous state. Combination $SR=11$ is forbidden. Output $Q=0$, when $R=1$, no matter the previous state
 - **JK (Jack Kilby** – not sure if coincidence ...); same as in SR, except that forbidden input is now allowed and produces flipped system state (if in state 0 then 1, and inversely)
 - **D (Data)** – is following the input
 - **T (Toggle)** – input changes (flips) the system state (when $T=1$, Q inverts)
- Different way of specifying behaviour of different FFs
 - **Functional table** (similar to state table) or **Excitation tables**
 - **Logic equations**
- You don't need to know by heart these, but you will have to learn how to switch from one to another

Possible outputs for any flip-flops

- Output Q can have one of the two values (0 or 1)
- There is:
 - the **present state Q** and there is
 - the **future state Q^+**
- Between present and future we can have only 4 possible combinations for any FF, and these combinations are codified as follows:

Q	Q ⁺	Operation	Code
0	0	Maintain 0	μ_0
0	1	Enable (turn on)	ε
1	0	Disable (turn off)	δ
1	1	Maintain 1	μ_1

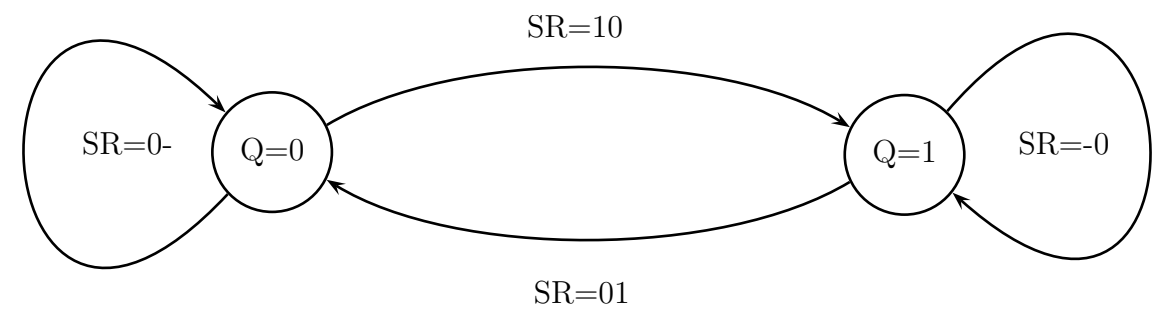
SR

Q^+	SR			
	00	01	11	10
0	0	0	–	1
1	1	0	–	1

State table

S	R	Q^+
0	0	Q
0	1	0
1	0	1
1	1	–

Operation table



State graph

	Q	Q^+	S	R
μ_0	0	0	0	–
ϵ	0	1	1	0
δ	1	0	0	1
μ_1	1	1	–	0

Excitation table

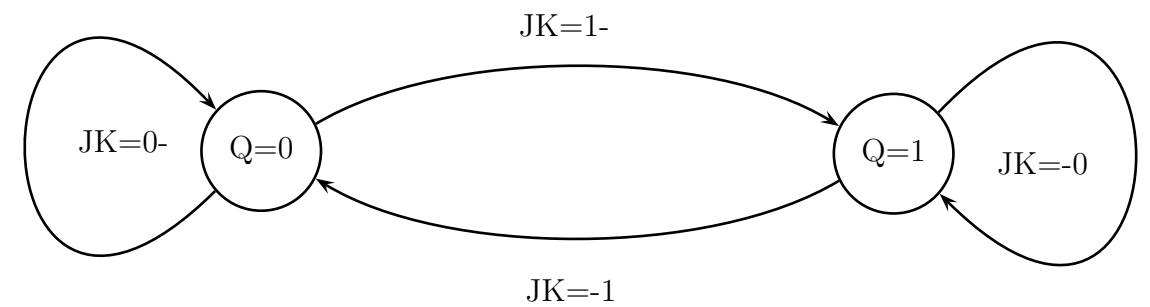
JK

	JK			
Q^+	00	01	11	10
0	0	0	1	1
1	1	0	0	1

State table

J	K	Q^+
0	0	Q
0	1	0
1	0	1
1	1	Q'

Operation table



State graph

	Q	Q^+	J	K
μ_0	0	0	0	—
ε	0	1	1	—
δ	1	0	0	1
μ_1	1	1	—	0

Excitation table

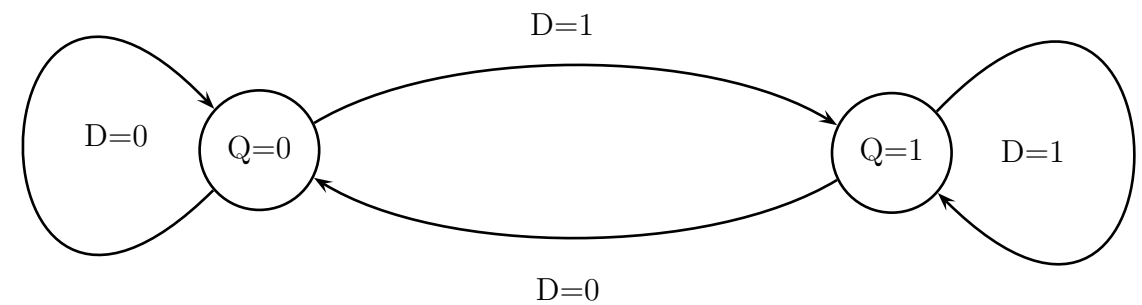
D

		D
Q^+	0	1
0	0	1
1	0	1

State table

D	Q^+
0	0
1	1

Operation table



State graph

	Q	Q^+	D
μ_0	0	0	0
ε	0	1	1
δ	1	0	0
μ_1	1	1	1

Excitation table

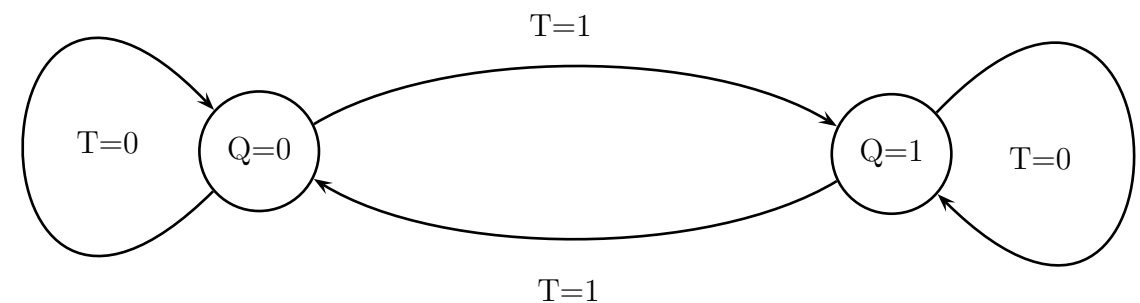
T

		T
Q^+	0	1
0	0	1
1	1	0

State table

T	Q^+
0	0
1	1

Operation table



State graph

	Q	Q^+	T
μ_0	0	0	0
ε	0	1	1
δ	1	0	1
μ_1	1	1	0

Excitation table

Schematic representation of FFs

With and without control signal (clock) Clk (small triangle)

