Finite State Machines (FSM)

Lecturer: Guillaume Beslon
(Lecture adapted from Lionel Morel)

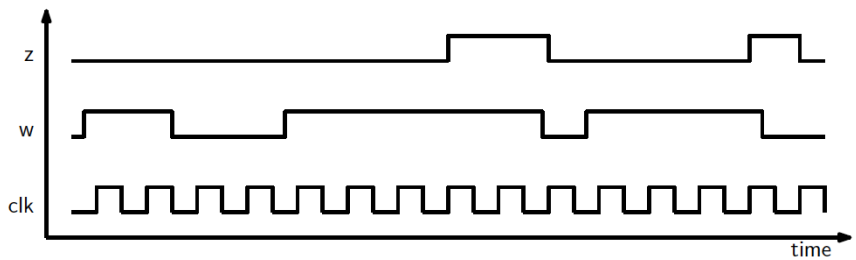
Computer Science and Information Technologies - INSA Lyon

Fall 2026

Previously, on AC ...

Two ways of representing a sequential behavior:

- Time-diagrams(aka **chronogrammes**)
  - Describe a specific sequence of input/output values graphically



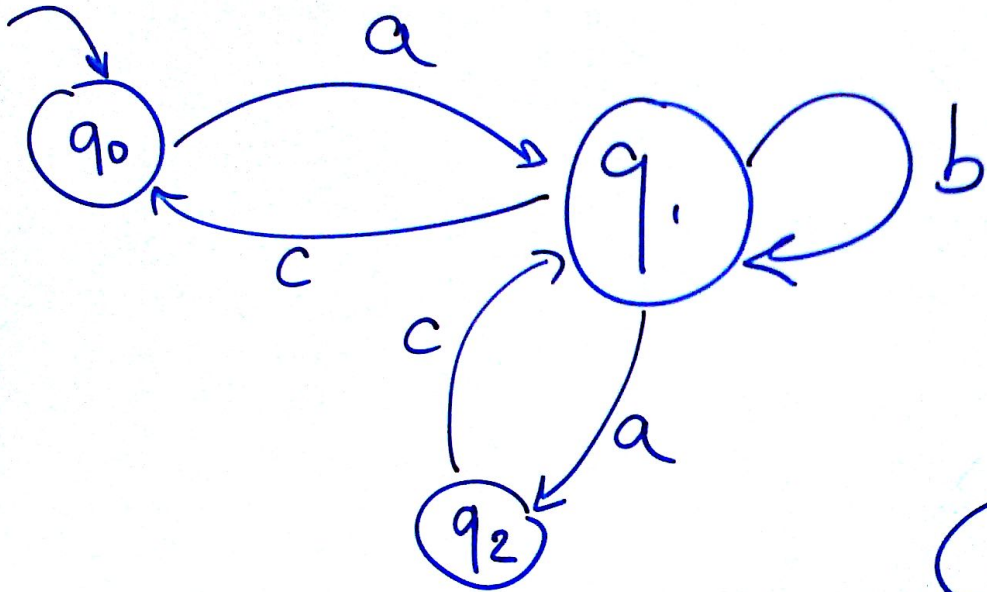
  - But chronograms can only gives a (short) example of the possible behavior
  - They cannot represent arbitrarily long behaviors
- **Finite State Machines (FSM)**:
  - Describe the full system's behavior **formally**

FSM: definition

A Finite State Machine (FSM or “Machine à États Finis”) is a tuple (Q, q_0, I, T) with:

- Q is the set of states (“États”)
- $q_0 \in Q$ is the initial state (“État initial”)
- I is the input alphabet (“Alphabet d’entrée”)
- $T \subseteq Q \times I \times Q$ is a transition function (“Fonction de transition”).

FSM: graphical representation



$$Q = \{q_0, q_1, q_2\}$$

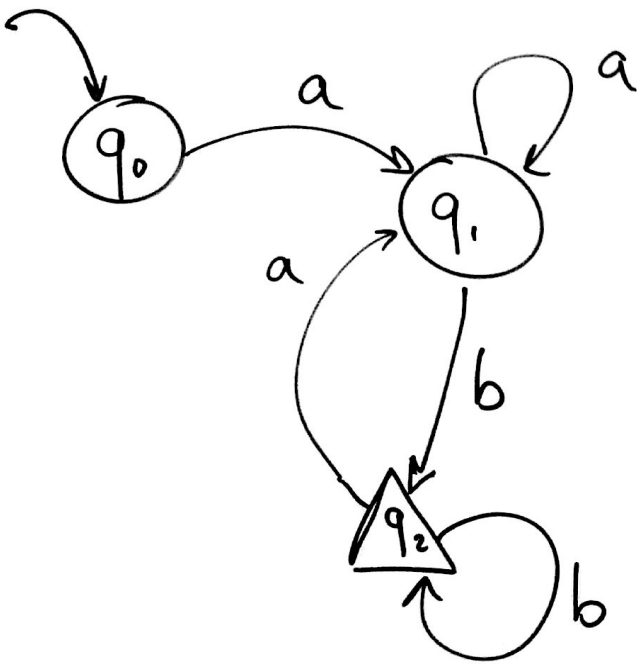
$$\underline{I} = \{a, b, c\}$$

Acceptors (“Accepteurs” ou ‘Reconnaisseurs”)

- Special form of FSM
- Have at least one **accepting state**
- (Regular languages) recognizers

Theorem: Regular languages are the class of languages recognizable by finite state machines (i.e. a language is regular if and only if it is accepted by a finite state machine).

Acceptors — Example



$Q = \{q_0, q_1, q_2\}$

$I = \{a, b, c\}$

By the way ... why are we here?

We want to describe systems that produce certain output sequences based on input sequences...

But our FSMs don't have outputs (yet).

Outputs can be introduced in two different ways:

- when the system is transitioning from one state to another
  $\Rightarrow$ **Mealy Machine**
- when the system lies in one state
  $\Rightarrow$ **Moore Machine**

Mealy Machines

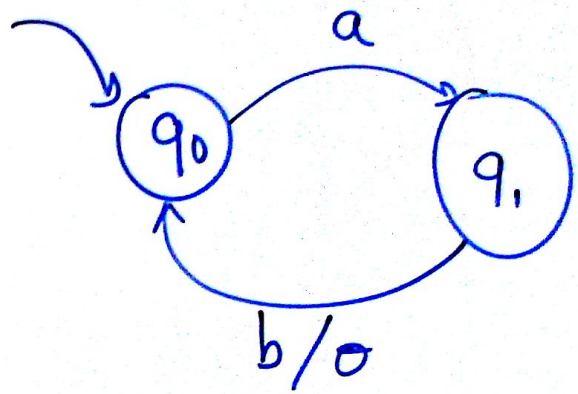
A Mealy machine is a tuple (Q, q_0, I, O, T) with:

- Q is the set of states
- $q_0 \in Q$ is the initial state
- I is the input alphabet
- O is the output alphabet
- $T \subseteq Q \times I \times O \times Q$ is a transition function.

On every transition, we read one input symbol/event and produce one output symbol/event (at least)

Mealy — Graphical Example





$I = \{a, b\}$

$O = \{o\}$

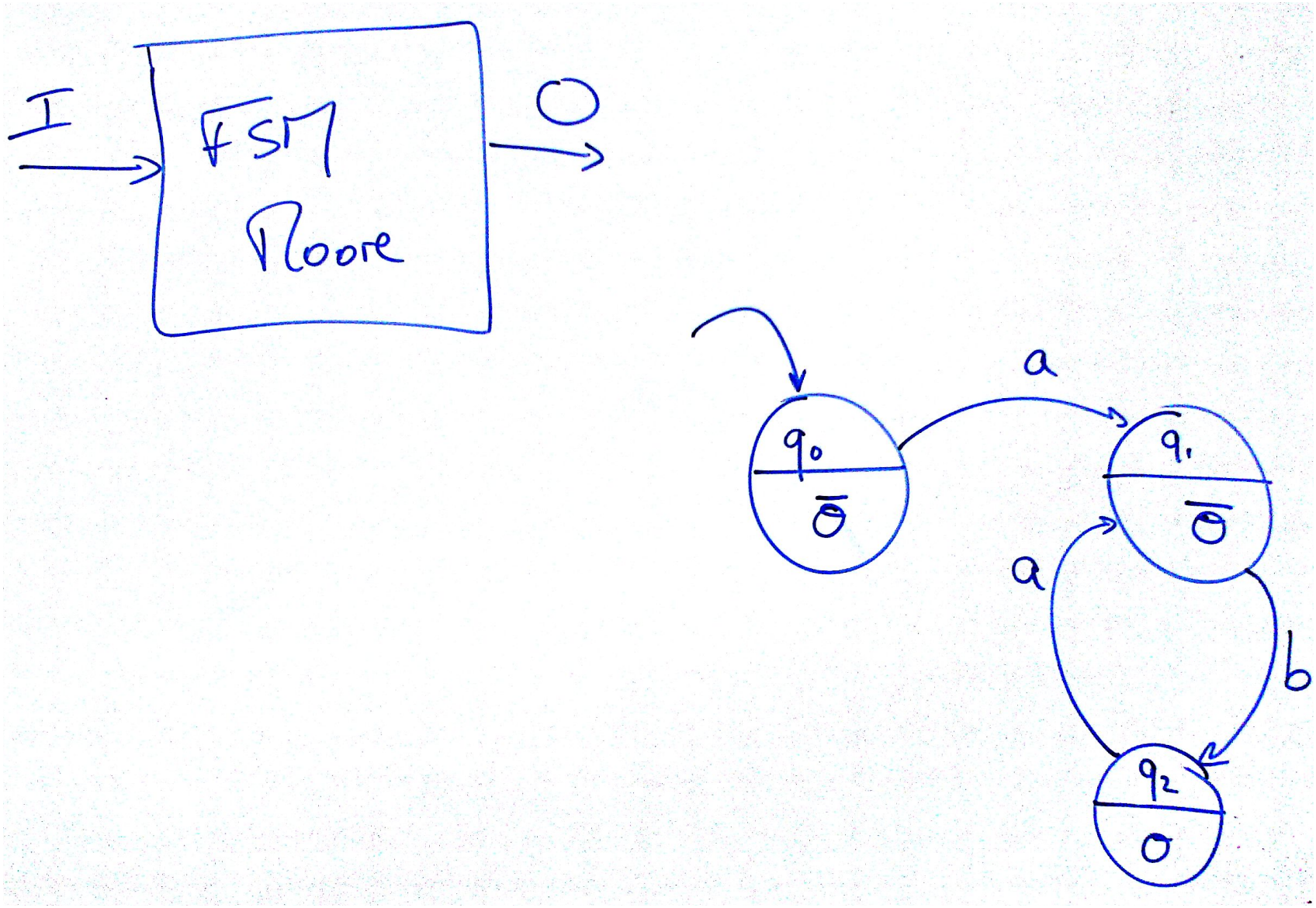
"emit o every time you see an 'a' followed by a 'b'"

Moore Machines

A Moore machine is a tuple (Q, q_0, I, O, T, F) with:

- Q is the set of states
- $q_0 \in Q$ is the initial state
- I is the input alphabet
- O is the output alphabet
- $T \subseteq Q \times I \times Q$ is a transition function.
- $F \subseteq Q \to O$ **is an output function.**

Moore — Graphical Example



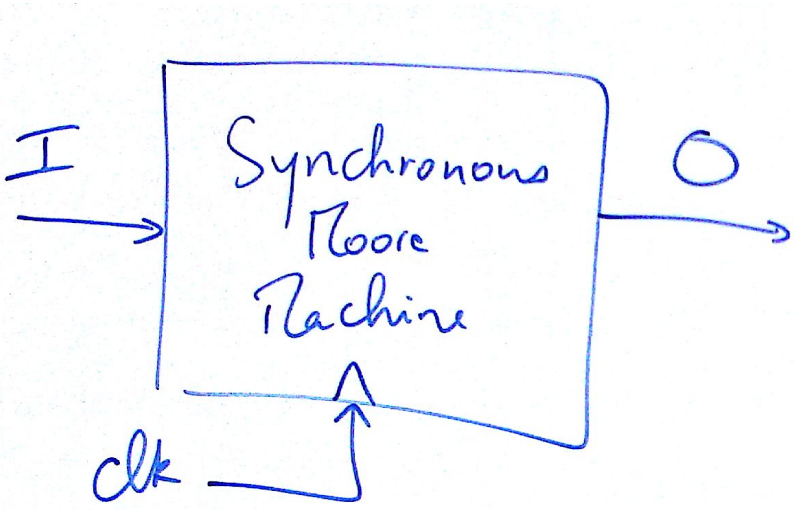
Mealy or Moore?

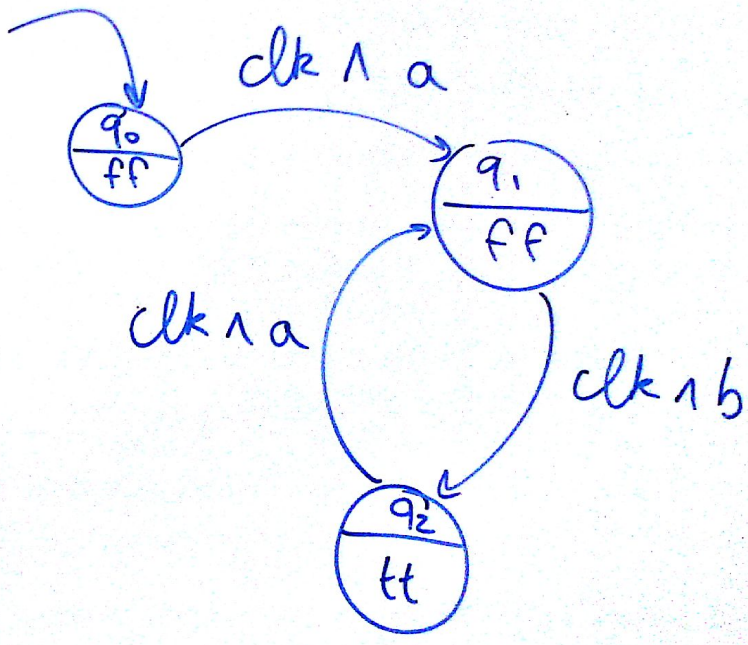
- Equivalent in expressiveness
- Translation from one to the other is always possible (at some costs)
- Mealy describes a very “event-driven” vision of the world
- In Computer Architecture, we will use Moore machines: Events (i.e. rising or falling edges) trigger state changes in registers and output are maintained for certain time.

Synchronous Moore Machine

We want to described **Synchronous Circuits**

Our automata should ultimately be driven by a clock

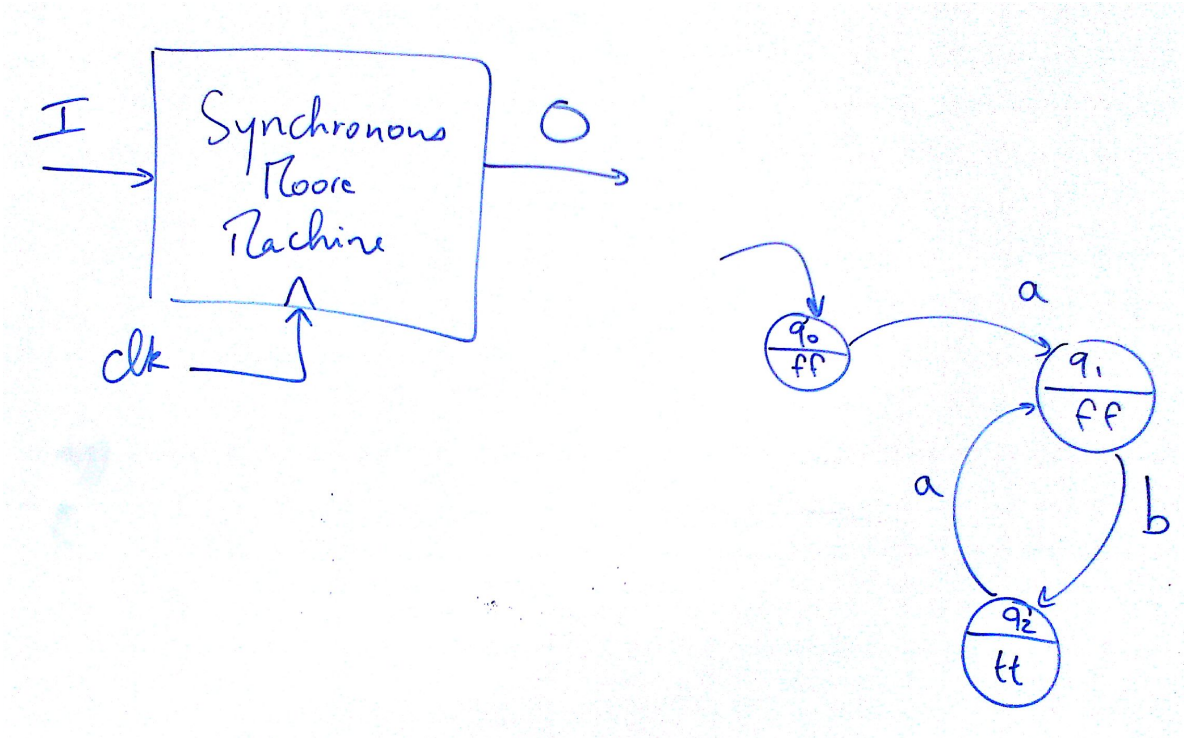




Synchronous Moore Machine

Most of the time, however, we will **not include clock** in description

$\Rightarrow$ Diagrams are more readable.



But **CLK** is always here, IMPLICITELY!!!

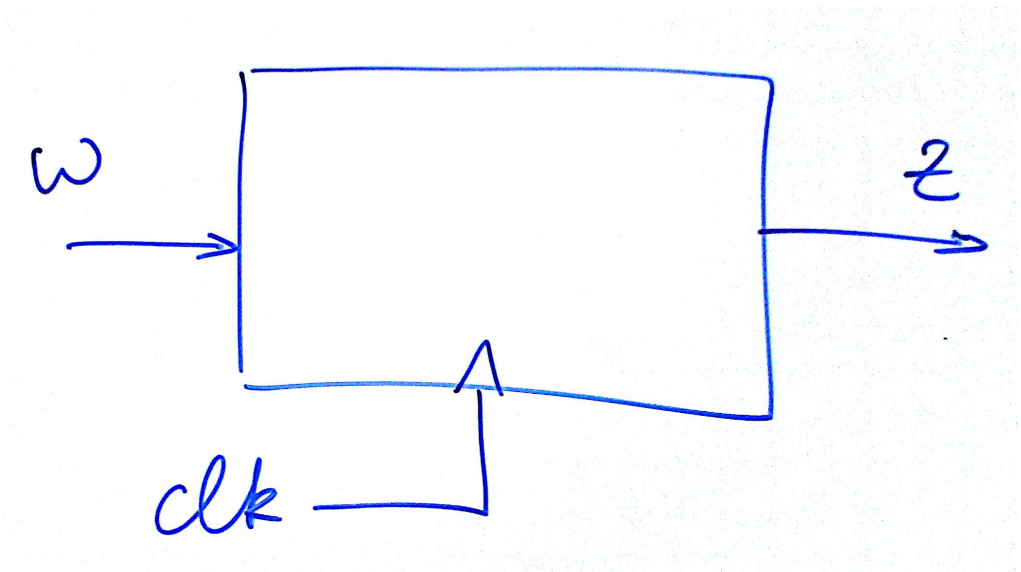
Running Example - specification (again!?)

Remember...

Input: `w`

Output: `z`

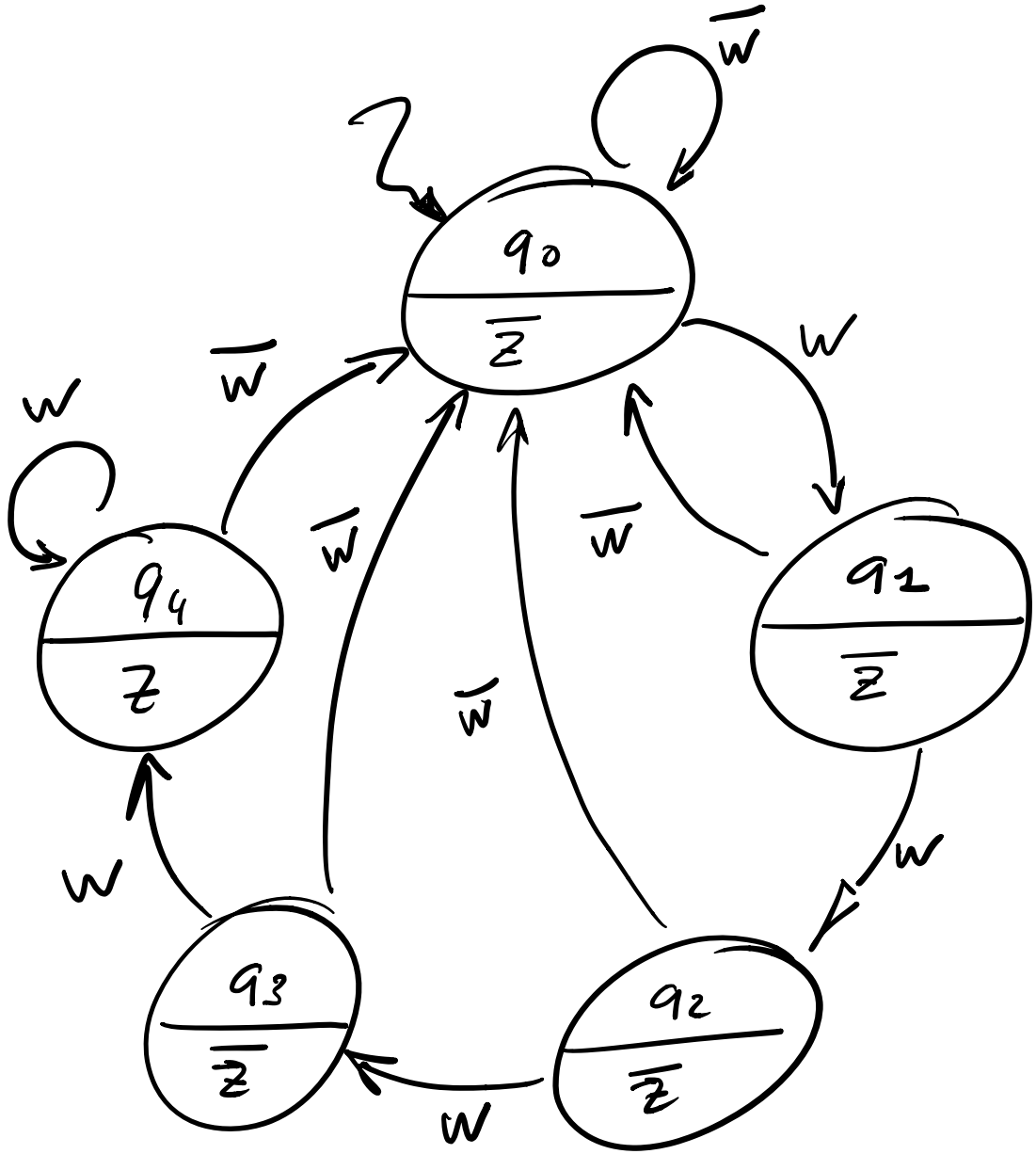
Behavior: `z` is true whenever `w` has been true for at least 4 clock cycles in a row.



Running Example

Blackboard

Running Example



Reactivity and Determinism

In Hardware an **FSMs need to be reactive and deterministic**:

Let's note c_{ij} the condition triggering transition between state s_i and s_j.
Let $Succ_i \in Q$ the set of all successors of state $s_i \in Q$.

Reactive: $\Sigma_{s_j \in Succ_i} c_{ij} = 1$
$\rightarrow$ in any given state, for any input value, the next state is known (i.e., the component is never blocked).

Deterministic: $\forall (s_j, s_k) \in Succ_i \times Succ_i,$ with $j \neq k, c_{ij}.c_{ik} = 0$
$\rightarrow$ in any given state, for any input value, there exist only one next possible state

Blackboard

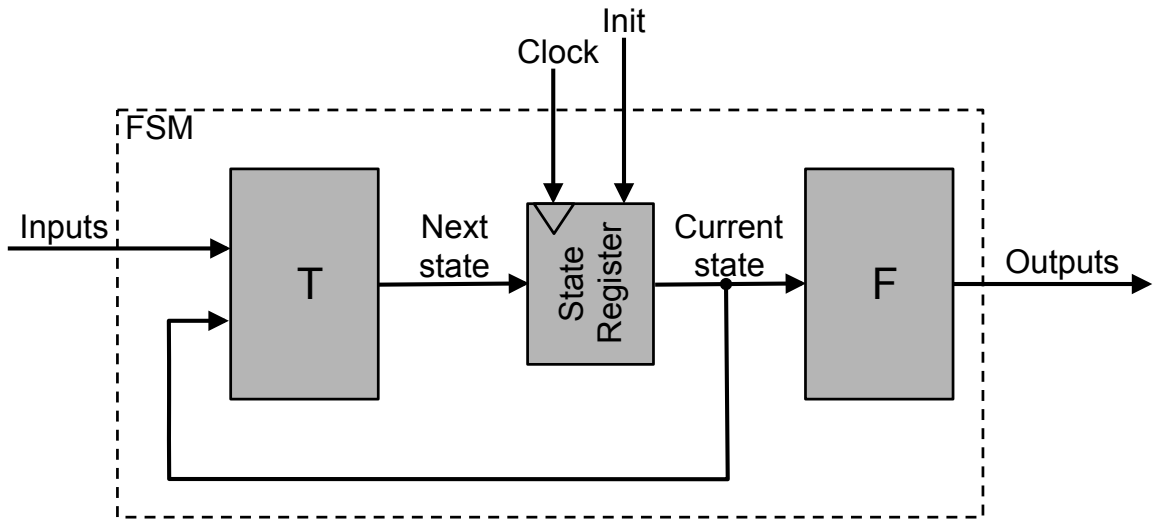
What Next?

FSM are theoretical objects
We need to transform them into **circuits** (“FSM Synthesis”)

We will proceed exactly like we did for combinatorial circuits:

Moore Machine $\rightarrow$ Truth Tables $\rightarrow$ Logical Equations $\rightarrow$ Circuits

Principle



- The **State Register** store the current FSM state
  → It is a sequential circuit synchronized by a **clock**
  → `Init` enforces q_0 (initial state) in the state register
- **T** implements the **transition function** of the FSM
  → **T** is a combinatorial circuit
- **F** implements the **output function** of the FSM
  → **F** is a combinatorial circuit

⇒ **At each time step** the loop ensures the transition from the current state to the next state according to **T**

FSM Synthesis

Method:

1. Input/output signals encoding
2. State encoding
3. Build truth tables for transition function and output function
4. Add state encoding to the truth tables
5. Translate truth tables into boolean functions
6. Build combinatorial circuits from boolean functions

1 Input/Output Encoding

Why encode?

- input and output alphabets are **symbols**
- Computer architecture deals with **bits**, not symbols

$\Rightarrow$ Need to **encode symbols into bit words.**

The final encoding...

- ... is often imposed by the hardware that will **use** the FSM
- ... is subject to **optimizations** that we will intentionally put aside here.

Input/Output Encoding: Running example (4-in-a-row)

The example is trivial:

- $I = \{w\}$, a unique boolean signal
- $O = \{z\}$, a unique boolean signal

② State encoding

Question: Given n states, how to choose a binary representation, ie choose an **injection from states to words of bits**.

- the binary representation is of size b, such that: $\lceil \log_2 n \rceil \leq b \leq n$
- any injection works!
- State encoding has a major influence on the complexity of functions **T** and **F** (some encodings lead to simpler implementations)
- But will not focus on optimizations here

② State encoding

Two main strategies

- Logarithmic encoding
  - States are associated with numbers
  - Numbers are encoded on $b = \lceil \log_2 n \rceil$ bits
  - All permutations are valid (but some may lead to simpler implementations)
  - Simpler solution: natural order (remember that we will not focus on optimizations...)
- One-hot-coding
  - n states encoded on n bits
  - One and only one bit is active at a time
  - The active bit indicates the state

$\rightarrow$ During AC labworks we will mainly use logarithmic encoding with natural order...
$\rightarrow$ BUT ... in AO labworks you will need One-Hot-Coding so be sure you understand it!

One-Hot Coding - Running Example

5 states $\Rightarrow$ state encoded on 5 bits (s_0 to s_4)

State	s_4	s_3	s_2	s_1	s_0
q_0	0	0	0	0	1
q_1	0	0	0	1	0
q_2	0	0	1	0	0
q_3	0	1	0	0	0
q_4	1	0	0	0	0

The property of this encoding ensures that there is always one and only one 1 in the encoding of the state. This property can be used to simplify the computation of the next state.

Logarithmic Encoding - Running Example

5 states $\Rightarrow$ state encoded on 3 bits ($\lceil \log_2 5 \rceil = 3$)

State	s_2	s_1	s_0
q_0	0	0	0
q_1	0	0	1
q_2	0	1	0
q_3	0	1	1
q_4	1	0	0

Warning: Be sure that you clearly distinguish states (q_i with $i \in \{0, 1, 2, 3, 4\}$) and state encoding bits (s_j with $j \in \{0, 1, 2\}$). Note that codes 101, 110 and 111 are impossible in this example (because there is no state q_5, q_6 and q_7 in the FSM).

Back to implementation principle

Blackboard!

3 Transition and Output Tables

State Transition Table:

- allows for the description of an FSM's graph as a table
- defines **T** as a function of the state encoding.

Output Table

- describes *state* $\rightarrow$ *output* function
- defines **F** as a function of the state encoding.

$\rightarrow$ Both tables can be directly constructed by "reading" the automaton ...

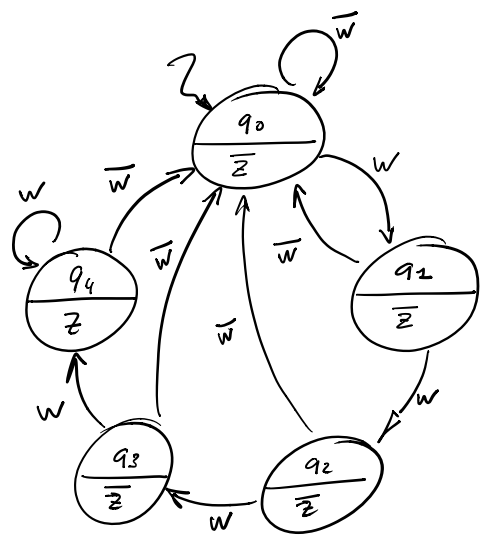
Transition and output tables: running example

Transition table

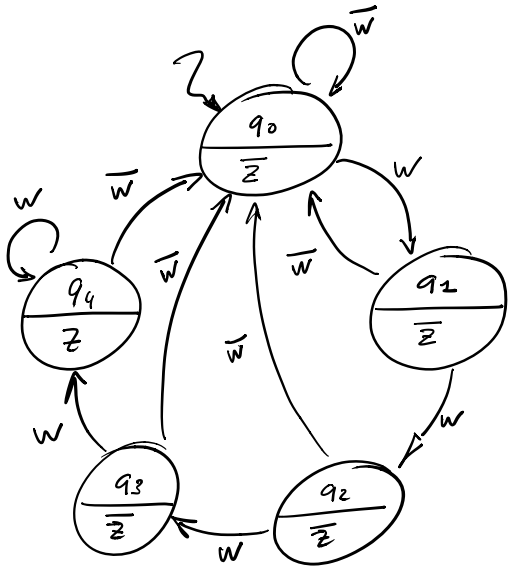
State	w	Next State
q_0	0	q_0
q_0	1	q_1
q_1	0	q_0
q_1	1	q_2
q_2	0	q_0
q_2	1	q_3
q_3	0	q_0
q_3	1	q_4
q_4	0	q_0
q_4	1	q_4

Output table

State	z
q_0	0
q_1	0
q_2	0
q_3	0
q_4	1



4 Include state encoding to get the truth tables



Truth table of the output function

State	z
q_0	0
q_1	0
q_2	0
q_3	0
q_4	1

$\Rightarrow$

s_2	s_1	s_0	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1

4 Include state encoding to get the truth tables

Truth table of the transition function

State	w	Next State
q_0	0	q_0
q_0	1	q_1
q_1	0	q_0
q_1	1	q_2
q_2	0	q_0
q_2	1	q_3
q_3	0	q_0
q_3	1	q_4
q_4	0	q_0
q_4	1	q_4

$\Rightarrow$

s_2	s_1	s_0	w	s'_2	s'_1	s'_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	0	1	1
0	1	1	0	0	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	0
1	0	0	1	1	0	0

5 Build the disjunctive normal forms #1

$\rightarrow$output function

s_2	s_1	s_0	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1

$$\Downarrow$$

$$z = s_2.\overline{s_1}.\overline{s_0}$$

5 Build the disjunctive normal forms #2

$\rightarrow$transition function

s_2	s_1	s_0	w	s'_2	s'_1	s'_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	0	1	1
0	1	1	0	0	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	0
1	0	0	1	1	0	0

$\Downarrow$

$$s'_0 = \overline{s_2}.\overline{s_1}.\overline{s_0}.w + \overline{s_2}.s_1.\overline{s_0}.w$$
$$s'_1 = \overline{s_2}.\overline{s_1}.s_0.w + \overline{s_2}.s_1.\overline{s_0}.w$$
$$s'_2 = \overline{s_2}.s_1.s_0.w + s_2.\overline{s_1}.\overline{s_0}.w$$

6 Translate boolean expression to gates and circuit...

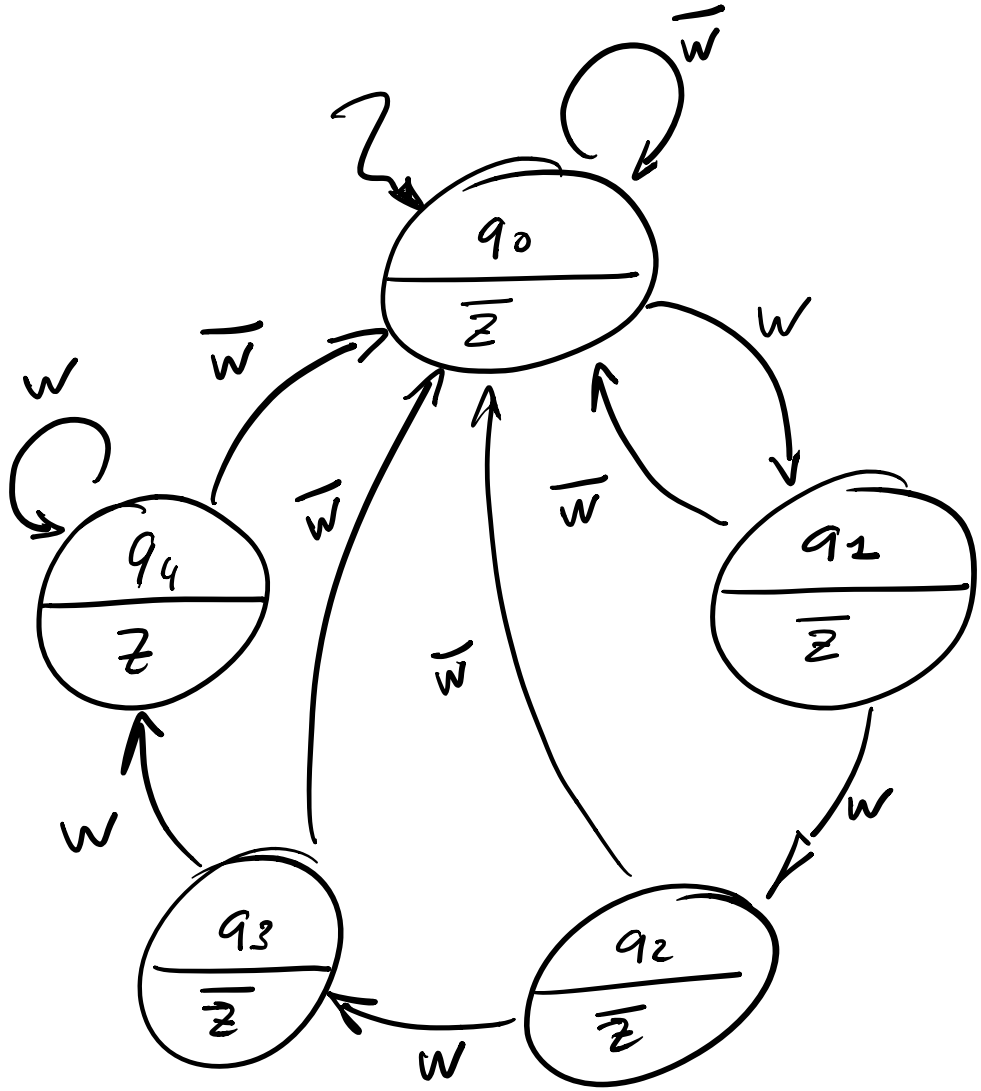
Demo time!

From the automaton to the circuit with the “one-hot-coding” state encoding

- When using “one-hot-coding”, it is not necessary to write the truth tables nor the logical equations!
- In “one-hot-coding”, each state is encoded in an independent one-bit register
- The graph can be directly encoded in the circuit
- But be careful when initializing the circuit...

$\Rightarrow$Example...

From the automaton to the circuit with the "one-hot-coding" state encoding



Demo time!