Finite State Machines (FSM)

Lecturer: Guillaume Beslon
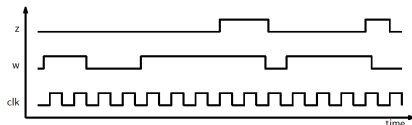### (Lecture adapted from Lionel Morel)

Computer Science and Information Technologies - INSA Lyon

Fall 2023

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 an example of a possible (short) 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
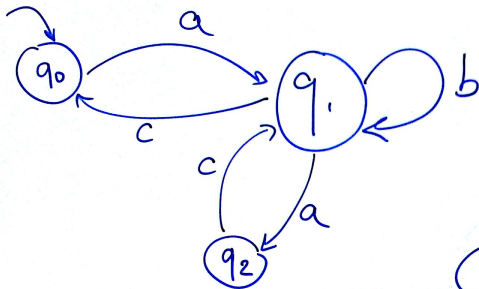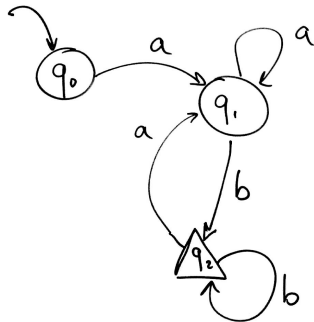

$$Q = \{ q_0, q_1, q_2 \}$$

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

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
  ⇒ **Mealy Machine**
- when the system lies in one state
  ⇒ **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)

$I \xrightarrow{} \boxed{\text{FSM } Mealy} \xrightarrow{O}$

$q_0 \xrightarrow{a} q_1$

$q_1 \xrightarrow{b/o} q_0$

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

$$O = \{o\}$$

"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.
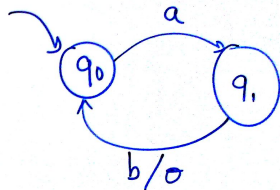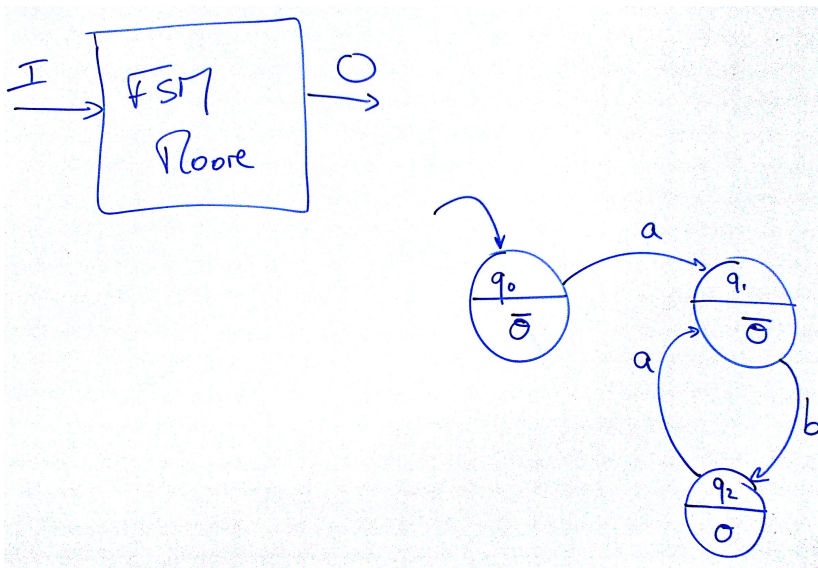- $F \subseteq Q \rightarrow O$ **is an output function.**

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
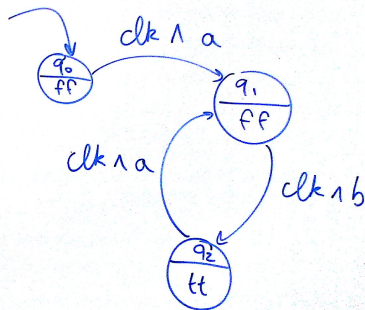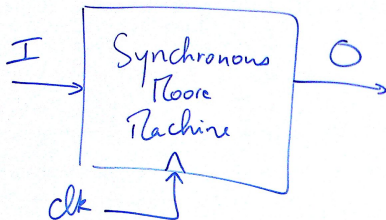
Synchronous Moore Machine

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

$\Rightarrow$ Diagrams are more readable.
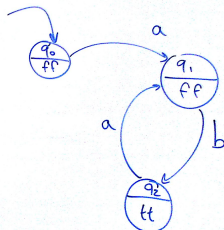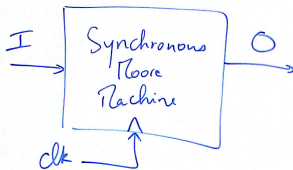
But **CLK** is always here, IMPLICITELY!!!

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.
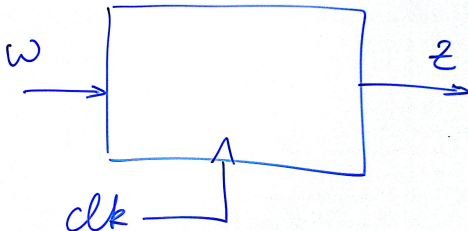
Blackboard

Running Example
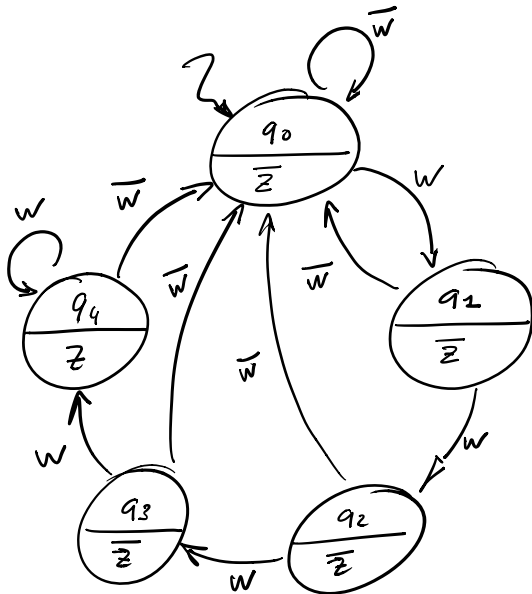
Reactivity and Determinism

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
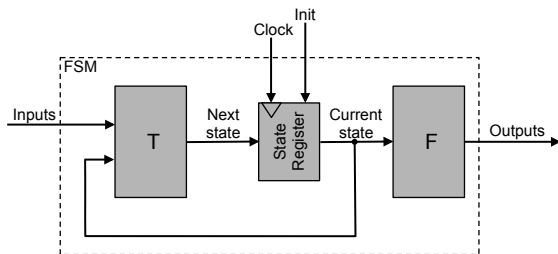
- ▶ **T** implements the **transition function** of the FSM
  → **T** is a combinatorial circuit
- ▶ The **State Register** store the current FSM state
  → **State Register** is a sequential circuit
- ▶ **F** implements the **output function** of the FSM
  → **F** is a combinatorial circuit
- ▶ Init enforces q_0 (initial state) in the state register
- ▶ **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
- ⇒ 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.

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

- 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
- 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...)

$\rightarrow$ In AC we will use logarithmic encoding with natural order...
$\rightarrow$ BUT ... in AO 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

FSM state	encoding ($s_4 s_3 s_2 s_1 s_0$)
q_0	00001
q_1	00010
q_2	00100
q_3	01000
q_4	10000

Logarithmic Encoding - Running Example

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

FSM state	encoding ($s_2 s_1 s_0$)
q_0	000
q_1	001
q_2	010
q_3	011
q_4	100

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 $j \in \{0, 1, 2\}$)

Back to implementation principle

Blackboard!

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 → output* function
- ▶ defines *F* as a function of the state encoding.

→ Both tables can be directly constructed by "reading" the automaton ...

Transition table: running example

Transition

State	input	next state
q_0	$\overline{w}$	q_0
q_0	w	q_1
q_1	$\overline{w}$	q_0
q_1	w	q_2
q_2	$\overline{w}$	q_0
q_2	w	q_3
q_3	$\overline{w}$	q_0
q_3	w	q_4
q_4	$\overline{w}$	q_0
q_4	w	q_4

Output

State	output
q_0	$\overline{z}$
q_1	$\overline{z}$
q_2	$\overline{z}$
q_3	$\overline{z}$
q_4	z
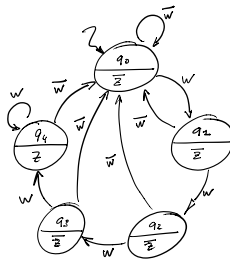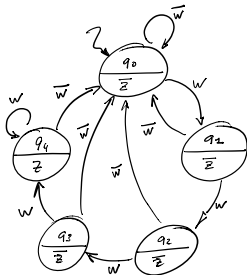
Output function with state encoding

State	s_2	s_1	s_0	output
q_0	0	0	0	$\overline{z}$
q_1	0	0	1	$\overline{z}$
q_2	0	1	0	$\overline{z}$
q_3	0	1	1	$\overline{z}$
q_4	1	0	0	z

Transition function with state encoding

State	s_2	s_1	s_0	input	next state	s_2'	s_1'	s_0'
q_0	0	0	0	$\overline{w}$	q_0	0	0	0
q_0	0	0	0	w	q_1	0	0	1
q_1	0	0	1	$\overline{w}$	q_0	0	0	0
q_1	0	0	1	w	q_2	0	1	0
q_2	0	1	0	$\overline{w}$	q_0	0	0	0
q_2	0	1	0	w	q_3	0	1	1
q_3	0	1	1	$\overline{w}$	q_0	0	0	0
q_3	0	1	1	w	q_4	1	0	0
q_4	1	0	0	$\overline{w}$	q_0	0	0	0
q_4	1	0	0	w	q_4	1	0	0

⑤ Build the disjunctive normal forms #1
→output function

State	s_2	s_1	s_0	output
q_0	0	0	0	$\overline{z}$
q_1	0	0	1	$\overline{z}$
q_2	0	1	0	$\overline{z}$
q_3	0	1	1	$\overline{z}$
q_4	1	0	0	z

$$\Downarrow$$

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

⑤ Build the disjunctive normal forms #2
→transition function

State	s_2	s_1	s_0	input	next state	s_2'	s_1'	s_0'
q_0	0	0	0	$\overline{w}$	q_0	0	0	0
q_0	0	0	0	w	q_1	0	0	1
q_1	0	0	1	$\overline{w}$	q_0	0	0	0
q_1	0	0	1	w	q_2	0	1	0
q_2	0	1	0	$\overline{w}$	q_0	0	0	0
q_2	0	1	0	w	q_3	0	1	1
q_3	0	1	1	$\overline{w}$	q_0	0	0	0
q_3	0	1	1	w	q_4	1	0	0
q_4	1	0	0	$\overline{w}$	q_0	0	0	0
q_4	1	0	0	w	q_4	1	0	0

$$\Downarrow$$

$$s_0' = \overline{s_2}.\overline{s_1}.\overline{s_0}.w + \overline{s_2}.s_1.\overline{s_0}.w$$
$$s_1' = \overline{s_2}.\overline{s_1}.s_0.w + \overline{s_2}.s_1.\overline{s_0}.w$$
$$s_2' = \overline{s_2}.s_1.s_0.w + s_2.\overline{s_1}.\overline{s_0}.w$$

Demo time!