Finite State Machines (FSM)

Lecturer: Guillaume Beslon

Computer Science and Information Technologies - INSA Lyon

Fall 2021
Previously, on AC ...

Two ways of representing a sequential behavior:

- time-diagrams, called **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*
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”).
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

\[ Q = \{ q_0, q_1, q_2 \} \]

\[ I = \{ a, b \} \]
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
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.
- \(F \subseteq Q \rightarrow O\) is an output function.
Moore — Graphical Example

FSM
Moore

\[ q_0 \]
\[ \overline{0} \]

\[ q_1 \]
\[ \overline{0} \]

\[ q_2 \]
\[ \overline{0} \]

\( a \)
\( b \)
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

⇒ 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 $\text{Succ}_i \in Q$ the set of all successors of state $s_i \in Q$.

**Reactive:** $\sum_{s_j \in \text{Succ}_i} c_{ij} = 1$

→ in any given state, for any input value, the next state is known (i.e., the component is never blocked).

**Deterministic:** $\forall (s_j, s_k) \in \text{Succ}_i \times \text{Succ}_i$, with $j \neq k$, $c_{ij} \cdot c_{ik} = 0$

→ 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
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
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
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
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
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...)

→ In AC we will only use logarithmic encoding with natural order...
One-Hot Coding - Running Example

5 states ⇒ state encoded on 5 bits

<table>
<thead>
<tr>
<th>FSM state</th>
<th>encoding ( (s_4 s_3 s_2 s_1 s_0) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>00001</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>00010</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>00100</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>01000</td>
</tr>
<tr>
<td>( q_4 )</td>
<td>10000</td>
</tr>
</tbody>
</table>
Logarithmic Encoding - Running Example

5 states ⇒ state encoded on 3 bits (⌈log₂5⌉ = 3)

<table>
<thead>
<tr>
<th>FSM state</th>
<th>encoding (s₂s₁s₀)</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>000</td>
</tr>
<tr>
<td>q₁</td>
<td>001</td>
</tr>
<tr>
<td>q₂</td>
<td>010</td>
</tr>
<tr>
<td>q₃</td>
<td>011</td>
</tr>
<tr>
<td>q₄</td>
<td>100</td>
</tr>
</tbody>
</table>

Warning: Be sure that you clearly distinguish states (qᵢ with i ∈ {0, 1, 2, 3, 4}) and state encoding bits (sⱼ with j ∈ {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 $state \rightarrow output$ 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

<table>
<thead>
<tr>
<th>State</th>
<th>input</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_0$</td>
<td>$w$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$w$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$w$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$w$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$w$</td>
<td>$q_4$</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>State</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$\overline{z}$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$\overline{z}$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$\overline{z}$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$\overline{z}$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$z$</td>
</tr>
</tbody>
</table>
Include state encoding into the truth tables

Output function with state encoding

<table>
<thead>
<tr>
<th>State</th>
<th>$s_2$</th>
<th>$s_1$</th>
<th>$s_0$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\overline{z}$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\overline{z}$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\overline{z}$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\overline{z}$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$z$</td>
</tr>
</tbody>
</table>
Include state encoding into the truth tables

Transition function with state encoding

<table>
<thead>
<tr>
<th>State</th>
<th>$s_2$</th>
<th>$s_1$</th>
<th>$s_0$</th>
<th>input</th>
<th>next state</th>
<th>$s'_2$</th>
<th>$s'_1$</th>
<th>$s'_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$w$</td>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$w$</td>
<td>$q_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$w$</td>
<td>$q_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$q_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$w$</td>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$w$</td>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Build the disjunctive normal forms #1 → output function

<table>
<thead>
<tr>
<th>State</th>
<th>$s_2$</th>
<th>$s_1$</th>
<th>$s_0$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\bar{z}$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\bar{z}$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\bar{z}$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\bar{z}$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$z$</td>
</tr>
</tbody>
</table>

$\Downarrow$

$$z = s_2.s_1.s_0$$
## Build the disjunctive normal forms #2

→ transition function

<table>
<thead>
<tr>
<th>State</th>
<th>$s_2$</th>
<th>$s_1$</th>
<th>$s_0$</th>
<th>input</th>
<th>next state</th>
<th>$s'_2$</th>
<th>$s'_1$</th>
<th>$s'_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$w$</td>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$w$</td>
<td>$q_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$w$</td>
<td>$q_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$q_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$w$</td>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\overline{w}$</td>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$w$</td>
<td>$q_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s'_0 = \overline{s_2}.s_1.\overline{s_0}.w + \overline{s_2}.s_1.\overline{s_0}.w \]
\[ s'_1 = s_2.s_1.s_0.w + s_2.s_1.s_0.w \]
\[ s'_2 = \overline{s_2}.s_1.s_0.w + s_2.\overline{s_1}.\overline{s_0}.w \]
Demo time!