- The topic of **Finite State Machines (FSMs)** has very formal underpinnings in automata theory ...
- ... basically they are a model of **computation**:
  - A FSM is a machine that can be in a finite set of states.
  - The machine consumes input symbols from an alphabet one at a time; symbols make the machine transition from one state to another according to a transition function.
  - When the input is exhausted, the machine halts; depending on the state it halts in, the machine is said to accept or reject the input.
  - The set of inputs accepted by the machine is termed the language accepted; this can be used to classify the machine itself.

Notes:

"Automata Theory in 10 minutes" (1)

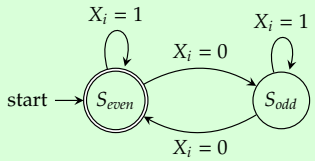
Question

Design an FSM that decides whether a binary sequence X has an even or odd number of 0 elements in it.

Algorithm (tabular)

Q	δ	
	Q'	
	$X_i = 0$	$X_i = 1$
S_{even}	S_{odd}	S_{even}
S_{odd}	S_{even}	S_{odd}

Algorithm (diagram)



- Note that:
  1. For the input $X = \langle 1, 0, 1, 1 \rangle$ the transitions are

$$\rightsquigarrow S_{even} \overset{X_0=1}{\rightsquigarrow} S_{even} \overset{X_1=0}{\rightsquigarrow} S_{odd} \overset{X_2=1}{\rightsquigarrow} S_{odd} \overset{X_3=1}{\rightsquigarrow} S_{odd}$$

     so the input is rejected, and has an odd number of 0 elements.
  2. For the input $X = \langle 1, 0, 1, 0 \rangle$ the transitions are

$$\rightsquigarrow S_{even} \overset{X_0=1}{\rightsquigarrow} S_{even} \overset{X_1=0}{\rightsquigarrow} S_{odd} \overset{X_2=1}{\rightsquigarrow} S_{odd} \overset{X_3=0}{\rightsquigarrow} S_{even}$$

     so the input is accepted, and has an even number of 0 elements.

Notes:

- Based on the fact that
  1. entry actions happen when entering a given state,
  2. exit actions happen when exiting a given state,
  3. input actions happen based on the state and any input received, and
  4. transition actions happen when a given transition between states is performed

  we can categorise an FSM based on output behaviour ...
  1. a **Moore** FSM only uses entry actions, i.e., the output depends on the state only, while
  2. a **Mealy** FSM only uses input actions, i.e., the output depends on the state and the input
- ... or on transition behaviour, where an FSM is deemed
  1. **deterministic** if for each state there is always one transition for each possible input (i.e., we always know what the next state should be), or
  2. **non-deterministic** if for each state there might be zero, one or more transitions for each possible input (i.e., we only know what the next state could be).

Notes:

"Automata Theory in 10 minutes" (3)

Definition

A **Finite State Machine (FSM)** is defined by the following:

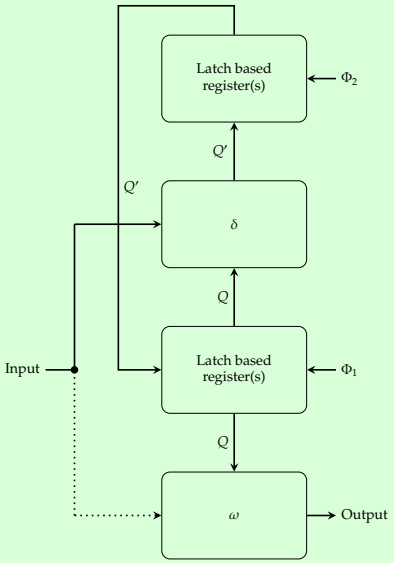
1. S, a finite set of **states** and a distinguished **start state** $s \in S$.
2. $A \subseteq S$, a finite set of **accepting states**.
3. An **input alphabet** Σ and **output alphabet** Γ.
4. A **transition function**
$$\delta : S \times \Sigma \rightarrow S.$$
5. An **output function**
$$\omega : S \rightarrow \Gamma$$
in the case of a Moore FSM, or
$$\omega : S \times \Sigma \rightarrow \Gamma$$
in the case of a Mealy FSM.

- Note that:
  - The FSM itself might be enough to solve a given problem, but it is common to control an associated data-path using the outputs.
  - A special "empty" input denoted ϵ allows a transition that can *always* occur.
  - It's common to allow δ to be a **partial function**, so it needn't be defined for all inputs.
  - If the FSM is non-deterministic, δ might instead give a *set* of possibilities that is randomly sampled from.

Notes:

FSMs in Hardware (1)

Algorithm



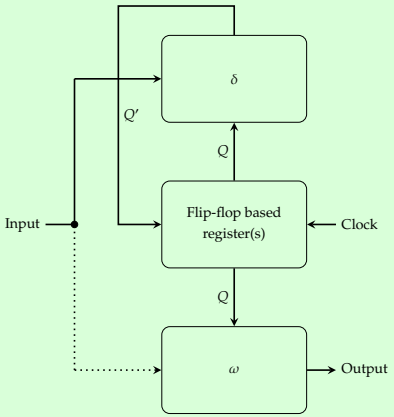
- Note that
  1. δ and ω are simply combinatorial logic,
  2. the state is retained in a register (i.e., a group of latches or flip-flops),
  3. within the current clock cycle
     - 3.1 ω computes the output from the current state and input, and
     - 3.2 δ computes the next state from the current state and input,
  4. the next state is latched by an appropriate feature (i.e., level or edge) in the clock

  i.e., this is a framework for a *computer* we can *build*!

Notes:

FSMs in Hardware (2)

Algorithm



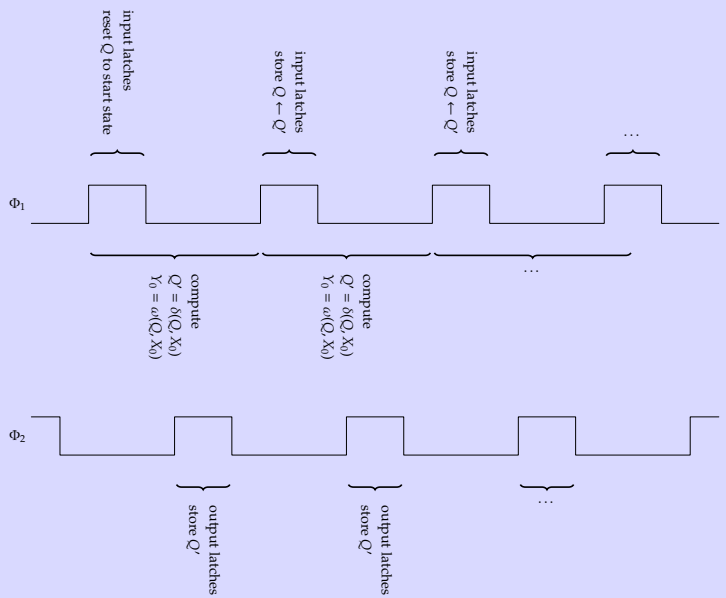
- Note that
  1. δ and ω are simply combinatorial logic,
  2. the state is retained in a register (i.e., a group of latches or flip-flops),
  3. within the current clock cycle
     - 3.1 ω computes the output from the current state and input, and
     - 3.2 δ computes the next state from the current state and input,
  4. the next state is latched by an appropriate feature (i.e., level or edge) in the clock

  i.e., this is a framework for a *computer* we can *build*!

Notes:

FSMs in Hardware (3)

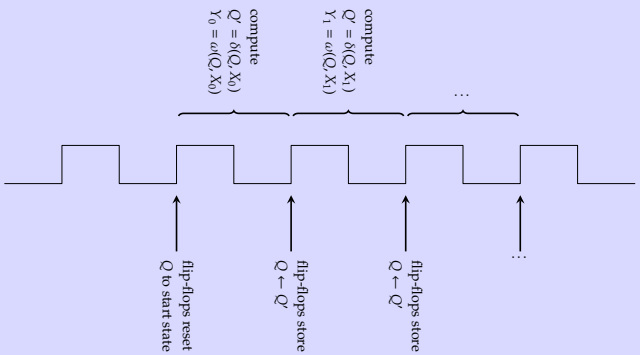
Example (given some sequence of input $X = \langle X_0, X_1, \ldots \rangle$ and output $Y = \langle Y_0, Y_1, \ldots \rangle$)



Notes:

FSMs in Hardware (4)

Example (given some sequence of input $X = \langle X_0, X_1, \ldots \rangle$ and output $Y = \langle Y_0, Y_1, \ldots \rangle$)



Notes:

FSMs in Hardware (5)

- To use the framework to solve a concrete problem, we follow a (fairly) standard sequence of steps:

Algorithm

1. Count the number of states required, and give each state an abstract label.
2. Describe the state transition and output functions using a tabular or diagrammatic approach.
3. Decide how the states will be represented, i.e., assign concrete values to the abstract labels, and allocate a large enough register to hold the state.
4. Express the functions δ and ω as (optimised) Boolean expressions, i.e., combinatorial logic.
5. Place the registers and combinatorial logic into the framework.

- Note that:
  - In hardware, it isn't common to have accepting states since we can't "halt"; we might include **idle** or **error** states to cope.
  - The framework doesn't show it, but in hardware it is common to have a **reset** input that (re)initialises the FSM into the start state.

Notes:

FSMs in Hardware (6) – a "modulo 6 ascending counter"

Question

Design an FSM that acts as a cyclic counter modulo n (rather than 2^n as before). If $n = 6$ for example, we want a component whose output r steps through values

$$0, 1, 2, 3, 4, 5, 0, 1, \ldots,$$

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

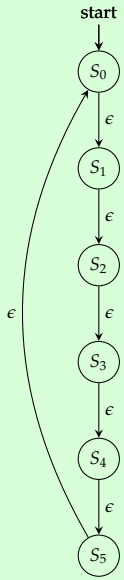
Notes:

FSMs in Hardware (7) – a "modulo 6 ascending counter"

Algorithm (tabular)

	δ	ω
Q	Q'	r
S_0	S_1	0
S_1	S_2	1
S_2	S_3	2
S_3	S_4	3
S_4	S_5	4
S_5	S_0	5

Algorithm (diagram)



Notes:

FSMs in Hardware (8) – a "modulo 6 ascending counter"

- There are 6 states representing the integers $0, 1, \ldots, 5$; we've given them the abstract labels $S_0, S_1, \ldots, S_5$.
- Since $2^3 = 8 > 6$, we can assign a concrete 3-bit value

$$\begin{aligned} S_0 &\mapsto \langle 0,0,0 \rangle \\ S_1 &\mapsto \langle 1,0,0 \rangle \\ S_2 &\mapsto \langle 0,1,0 \rangle \\ S_3 &\mapsto \langle 1,1,0 \rangle \\ S_4 &\mapsto \langle 0,0,1 \rangle \\ S_5 &\mapsto \langle 1,0,1 \rangle \end{aligned}$$

to each abstract label; this basically means we can talk about

1. $Q = \langle Q_0, Q_1, Q_2 \rangle$ as being the current state, and
2. $Q' = \langle Q'_0, Q'_1, Q'_2 \rangle$ as being the next state.

Notes:

FSMs in Hardware (9) – a "modulo 6 ascending counter"

Algorithm (truth table)

Rewriting the abstract labels yields the following concrete truth table:

			δ			ω		
Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0	r_2	r_1	r_0
0	0	0	0	0	1	0	0	0
0	0	1	0	1	0	0	0	1
0	1	0	0	1	1	0	1	0
0	1	1	1	0	0	0	1	1
1	0	0	1	0	1	1	0	0
1	0	1	0	0	0	1	0	1
1	1	0	?	?	?	?	?	?
1	1	1	?	?	?	?	?	?

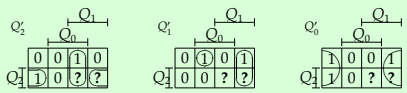
Note that our state assignment means $r = Q$, so ω is basically just the identity function for that output.

Notes:

FSMs in Hardware (10) – a "modulo 6 ascending counter"

Circuit (δ)

Translating the truth table into a set of Karnaugh maps



yields the following Boolean expressions:

$$Q'_2 = (\ Q_1 \wedge Q_0\ ) \vee (\ Q_2 \wedge \neg Q_0\ )$$

$$Q'_1 = (\ \neg Q_2 \wedge \neg Q_1 \wedge Q_0\ ) \vee (\ Q_1 \wedge \neg Q_0\ )$$

$$Q'_0 = (\ \neg Q_0\ )$$

Notes:

- The fact we do state assignment late on in the process is intentional; it allows us to optimise the representation based on what we do with it.
  1. A **binary encoding** represents the i-th of n states as a $(\lceil \log_2(n) \rceil)$-bit unsigned integer i.
  2. A **one-hot encoding** is where for state i, a valid code word X has $X_i = 1$ and $X_j = 0$ for $j \neq i$, e.g., for $n = 6$

$$\begin{array}{lcl} S_0 & \mapsto & \langle 1,0,0,0,0,0 \rangle \\ S_1 & \mapsto & \langle 0,1,0,0,0,0 \rangle \\ S_2 & \mapsto & \langle 0,0,1,0,0,0 \rangle \\ S_3 & \mapsto & \langle 0,0,0,1,0,0 \rangle \\ S_4 & \mapsto & \langle 0,0,0,0,1,0 \rangle \\ S_5 & \mapsto & \langle 0,0,0,0,0,1 \rangle \end{array}$$

  noting that
  - we have a larger state (i.e., n bits instead of $\lceil \log_2(n) \rceil$), but
  - transition between states is easier, and
  - switching behaviour (and hence power consumption) is reduced.

Notes:

FSMs in Hardware (12) – a "modulo 6 ascending/descending counter with alert"

Question

Design an FSM that acts as a cyclic counter modulo n, but whose direction can also be controlled. If $n = 6$ for example, we want a component whose output r steps through values

$$0, 1, 2, 3, 4, 5, 0, 1, \ldots$$

or

$$0, 5, 4, 3, 2, 1, 0, 5, \ldots$$

depending on some input d, *plus* has an output f to signal when the cycle occurs (i.e., when the current value is last or first in the sequence, depending on d).

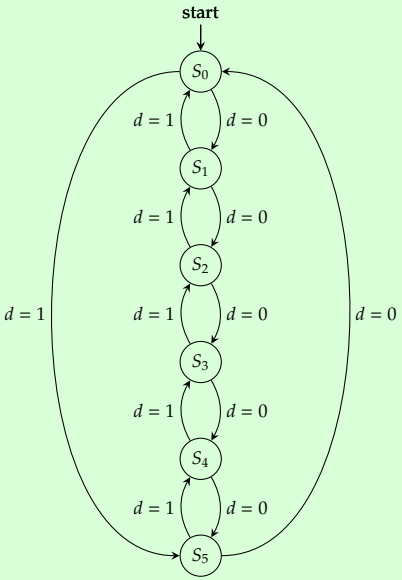
Notes:

FSMs in Hardware (13) – a "modulo 6 ascending/descending counter with alert"

Algorithm (tabular)

	δ		ω		
Q	Q'		r	f	
	$d = 0$	$d = 1$		$d = 0$	$d = 1$
S_0	S_1	S_5	0	0	1
S_1	S_2	S_0	1	0	0
S_2	S_3	S_1	2	0	0
S_3	S_4	S_2	3	0	0
S_4	S_5	S_3	4	0	0
S_5	S_0	S_4	5	1	0

Algorithm (diagram)



Notes:

FSMs in Hardware (14) – a "modulo 6 ascending/descending counter with alert"

Algorithm (truth table)

Rewriting the abstract labels yields the following concrete truth table:

				δ			ω			
d	Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0	r_2	r_1	r_0	f
0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	0	1	0
0	0	1	0	0	1	1	0	1	0	0
0	0	1	1	1	0	0	0	1	1	0
0	1	0	0	1	0	1	1	0	0	0
0	1	0	1	0	0	0	1	0	1	1
0	1	1	0	**?**	**?**	**?**	**?**	**?**	**?**	**?**
0	1	1	1	**?**	**?**	**?**	**?**	**?**	**?**	**?**
1	0	0	0	1	0	1	0	0	0	1
1	0	0	1	0	0	0	0	0	1	0
1	0	1	0	0	0	1	0	1	0	0
1	0	1	1	0	1	0	0	1	1	0
1	1	0	0	0	1	1	1	0	0	0
1	1	0	1	1	0	0	1	0	1	0
1	1	1	0	**?**	**?**	**?**	**?**	**?**	**?**	**?**
1	1	1	1	**?**	**?**	**?**	**?**	**?**	**?**	**?**

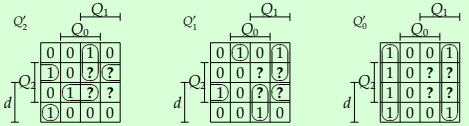
Note that our state assignment means $r = Q$, so ω is basically just the identity function for that output.

Notes:

FSMs in Hardware (15) – a "modulo 6 ascending/descending counter with alert"

Circuit (δ)

Translating the truth table into a set of Karnaugh maps



yields the following Boolean expressions:

$$
\begin{aligned}
Q_2' = (\ & \neg d & & & \wedge\ & Q_1 & \wedge\ & Q_0 &)\ \vee \\
(\ & \neg d & \wedge\ & Q_2 & & & \wedge\ & \neg Q_0 &)\ \vee \\
(\ & d & \wedge\ & Q_2 & & & \wedge\ & Q_0 &)\ \vee \\
(\ & d & \wedge\ & \neg Q_2 & \wedge\ & \neg Q_1 & \wedge\ & \neg Q_0 &)
\end{aligned}
$$

$$
\begin{aligned}
Q_1' = (\ & \neg d & \wedge\ & \neg Q_2 & \wedge\ & \neg Q_1 & \wedge\ & Q_0 &)\ \vee \\
(\ & \neg d & & & \wedge\ & Q_1 & \wedge\ & \neg Q_0 &)\ \vee \\
(\ & d & \wedge\ & Q_2 & & & \wedge\ & \neg Q_0 &)\ \vee \\
(\ & d & & & \wedge\ & Q_1 & \wedge\ & Q_0 &)
\end{aligned}
$$

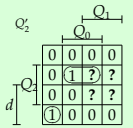
$$
\begin{aligned}
Q_0' = (\ & & & & & & & \neg Q_0 &)
\end{aligned}
$$

Notes:

FSMs in Hardware (16) – a "modulo 6 ascending/descending counter with alert"

Circuit (ω)

Translating the truth table into a set of Karnaugh maps



yields the following Boolean expressions:

$$
\begin{aligned}
f = (\ & \neg d & \wedge\ & Q_2 & & & \wedge\ & Q_0 &)\ \vee \\
(\ & d & \wedge\ & \neg Q_2 & \wedge\ & \neg Q_1 & \wedge\ & \neg Q_0 &)
\end{aligned}
$$

Notes:

Conclusions

- Take away points:
  1. We've linked together theory and practice: FSMs are abstract computational models, but we've used them to solve concrete problems.
  2. We've *only* used concepts in digital logic that we know how to construct right from the transistor-level; there is no "magic" going on behind the scenes.
  3. Clearly the examples are limited, but a fundamentally similar framework can be used for more complex computational machines.

Notes:

References and Further Reading

[1] Wikipedia: Clock signal.
http://en.wikipedia.org/wiki/Clock_signal.

[2] Wikipedia: Finite State machine (FSM).
http://en.wikipedia.org/wiki/Finite-state_machine.

[3] Wikipedia: Flip-flop.
http://en.wikipedia.org/wiki/Flip-flop_(electronics).

[4] Wikipedia: Latch.
http://en.wikipedia.org/wiki/Latch_(electronics).

[5] D. Page.
Chapter 2: Basics of digital logic.
In *A Practical Introduction to Computer Architecture*. Springer-Verlag, 1st edition, 2009.

[6] W. Stallings.
Chapter 11: Digital logic.
In *Computer Organisation and Architecture*. Prentice-Hall, 9th edition, 2013.

[7] A.S. Tanenbaum.
Section 3.1: Gates and Boolean algebra.
In *Structured Computer Organisation* [9].

Notes:

References and Further Reading

[8] A.S. Tanenbaum.
Section 3.2: Basic digital logic circuits.
In *Structured Computer Organisation* [9].

[9] A.S. Tanenbaum.
Structured Computer Organisation.
Prentice-Hall, 6th edition, 2012.

Notes: