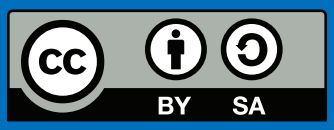
A Practical Introduction to Computer Architecture

Daniel Page ⟨dan@phoo.org⟩

git # 5ac601b @ 2018-07-13



FINITE STATE MACHINES (FSMS)

1 State machines: from simple to more complex control-paths

The topic of automaton, specifically **Finite State Machines (FSMs)**, has a very formal basis; basically they are models of computation, not too far from topics such as Turing Machines (TMs). Put another way, you can think of an FSM as a computer, albeit a simple one.

The control-path in Figure 2.37 is *very* simple: this is partly an artefact of the problem at hand of course, but masks the difficulty of dealing with more complicated problems. FSMs represent an attractive solution however, allowing us to reason about and implement more complicated, general-purpose control-paths.

1.1 A rough overview of FSM-related theory

Definition 0.1. *An FSM is a (theoretical) machine that can be in a finite set of* **states**. *The machine consumes input* **symbols** *from an* **alphabet** *(which defines which symbols are valid and so on) 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.*

Definition 0.2. *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-style** *FSM only uses entry actions, i.e., the output depends on the state only, while*
2. *a* **Mealy-style** *FSM only uses input actions, i.e., the output depends on the state and the input.*

An alternative classification relates to transition behaviour, where an FSM is deemed

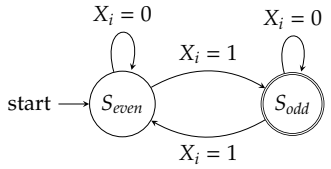
1. **deterministic** *if for each state there is always one transition for each possible input (i.e., we always know what the next state should be), or*
2. **non-deterministic** *if for each state there might be zero, one or more transitions for each possible input (i.e., we only know what the next state could be).*

Definition 0.3. *A given FSM can be defined via the following:*

1. *S, a finite set of states and a distinguished start state $s \in S$.*

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

(a) *A tabular description.*

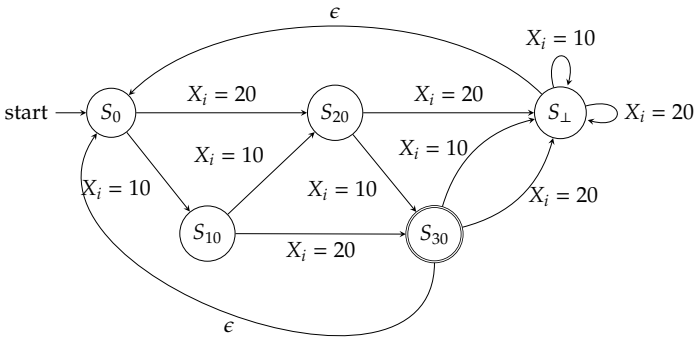


(b) *A diagrammatic description.*

Figure 1: *An example FSM to decide whether there is an odd number of* 1 *elements in some sequence* X*.*

Q	δ	
	Q'	
	$X_i = 10$	$X_i = 20$
S_0	S_{10}	S_{20}
S_{10}	S_{20}	S_{30}
S_{20}	S_{30}	$S_\perp$
S_{30}	$S_\perp$	$S_\perp$

(a) *A tabular description.*



(b) *A diagrammatic description.*

Figure 2: *An example FSM modelling a simple vending machine.*

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" (or null) input denoted* ϵ *allows a transition which can* always *occur.*
- *It is common to allow* δ *to be a partial function, i.e., a function which is not defined for all inputs.*
- *If the FSM is non-deterministic, then* δ *might instead give a set of possibilities that is sampled from.*

More simply, you can think of an FSM as a directed graph where moving between nodes (which represent each state) means consuming the input on the corresponding edge. Some examples should show that the fairly formal description above translates into a much more manageable reality.

1.1.1 Example #1: even or odd number of 0 elements

Imagine we are tasked with designing an FSM that decides whether a binary sequence X has an odd number 1 elements in it (i.e., it computes the parity of X). The input alphabet in this case is

$$\Sigma = \{0, 1\}$$

since each X_i can either be 0 or 1. The FSM can clearly be in two states: having consumed the input so far, it can either have seen an even or odd number of 1 elements. Therefore we can say

$$S = \{S_{even}, S_{odd}\},$$

have $s = S_{even}$ as the starting state, and let $A = \{S_{odd}\}$ be the (singleton) set of accepting states. There is no output as such, so in this case both the and output alphabet Γ and output function ω are irrelevant.

Our final task is to define the transition function. Figure 1 includes a tabular and a diagrammatic description of the same thing. The tabular, truth table style description is easier to discuss. The idea is that it lists the current state (left-hand side), alongside the next state for each possible input (right-hand side). In words, the rows read as follows:

- if we are in state S_{even} and the input $X_i = 0$ then we stay in state S_{even},
- if we are in state S_{even} and the input $X_i = 1$ then we move to state S_{odd},
- if we are in state S_{odd} and the input $X_i = 0$ then we stay in state S_{odd}, and
- if we are in state S_{odd} and the input $X_i = 1$ then we move to state S_{even}.

The intuition is, for example and with a similar argument possible for the state S_{odd}, that if we are in state S_{even} (i.e., have seen an even number of 1 elements so far) and the next input is 1, then we have *now* seen an odd number of 1 elements so move to state S_{odd}. Conversely, if we are in state S_{even} (i.e., have seen an even number of 1 elements so far) and the next input is 0, then we have *still* seen an even number of 1 elements so stay in state S_{even}.

Consider some examples of the FSM in operation

1. For the input $X = \langle 1, 0, 1, 1 \rangle$ the transitions are

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

   meaning we start in state S_{even} then

   (a) move to S_{odd} since $X_0 = 1$,
   (b) stay in S_{odd} since $X_1 = 0$,
   (c) move to S_{even} since $X_2 = 1$, and finally
   (d) move to S_{odd} since $X_3 = 1$

   Since we finish in state S_{odd}, the input is accepted and hence we conclude it has an odd number of 1 elements.

2. For the input $X = \langle 0, 1, 1, 0 \rangle$ the transitions are

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

   meaning we start in state S_{even} then

   (a) stay in S_{even} since $X_0 = 0$,
   (b) move to S_{odd} since $X_1 = 1$,
   (c) move to S_{even} since $X_2 = 1$, and finally
   (d) stay in S_{even} since $X_3 = 0$.

   Since we finish in state S_{even}, the input is rejected and hence we conclude it has an even number of 1 elements.

1.1.2 Example #2: a vending machine

Imagine we are tasked with designing an FSM that controls a vending machine. The machine accepts tokens worth 10 or 20 units: when the total value of tokens entered reaches 30 units it delivers a chocolate bar but it does not give change. That is, the exact amount must be entered otherwise an error occurs, all tokens are ejected and we start afresh.

The design is clearly a little more complex this time. The input alphabet is basically just the tokens that the machine can accept, so we have

$$\Sigma = \{10, 20\}.$$

The set of states the machine can be in is easy to enumerate: it can either have accepted tokens totalling 0, 10, 20 or 30 units in it or be in the error state which we denote by $\perp$. Thus, we can say

$$S = \{S_\perp, S_0, S_{10}, S_{20}, S_{30}\}$$

and clearly set $s = S_0$ since initially the machine has accepted no tokens. There is one accepting state, which is when a total of 30 tokens has been accepted, so $A = \{S_{30}\}$. Since there is again no output, our final task is again to define the transition function. As before, Figure 2 outlines a tabular and diagrammatic description.

1. For the input $X = \langle 10, 20 \rangle$ the transitions are

   $$\rightsquigarrow S_0 \overset{X_0=10}{\rightsquigarrow} S_{10} \overset{X_1=20}{\rightsquigarrow} S_{30}$$

   meaning we start in state S_0 then

   (a) move to S_{10} since $X_0 = 10$, and finally
   (b) move to S_{30} since $X_3 = 30$.

   Since we finish in state S_{30}, the input is accepted and we get a chocolate bar as output!

2. For the input $X = \langle 20, 20 \rangle$ the transitions are

   $$\rightsquigarrow S_0 \overset{X_0=20}{\rightsquigarrow} S_{20} \overset{X_1=20}{\rightsquigarrow} S_\perp$$

   meaning we start in state S_0 then

   (a) move to S_{20} since $X_0 = 20$, and finally
   (b) move to $S_\perp$ since $X_3 = 20$.

   Since we finish in state $S_\perp$, the error state, the input is rejected and the tokens are returned.

Note that the input marked ϵ is the empty input; that is, with no input we can move between the accepting or error states back into the start state thus resetting the machine. So for example, once we accept or reject the input we might assume the machine returns to state S_0.

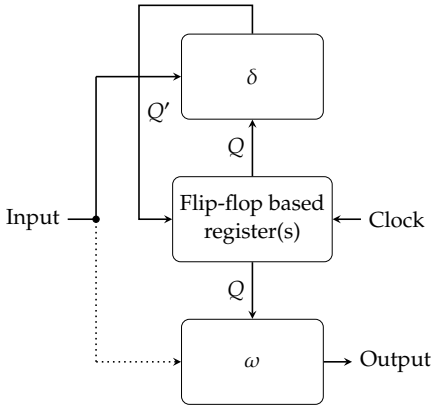
1.2 Practical implementation of FSMs in hardware

Based on the formal definition above, Figure 3 illustrates a general framework into which we can place concrete implementations of the component parts in a specific FSM. It is crucial to notice that when drawn as a diagram like this, we can have

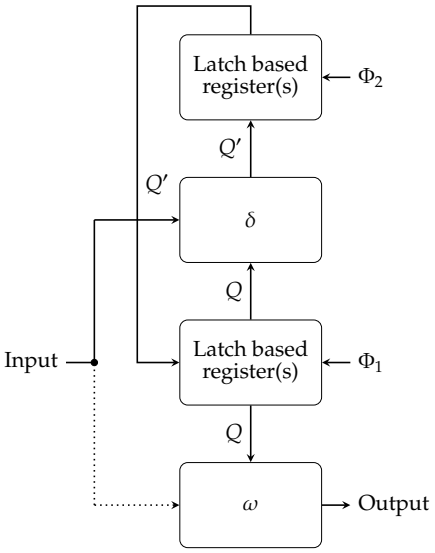
1. the state implemented by register (i.e., a group of latches or flip-flops), and
2. the δ and ω functions implemented using combinatorial logic only: they are functions of the current state and any input.

The behaviour of the framework is illustrated by Figure 4. The idea is that within a given current clock cycle

1. ω computes the output from the current state and input, and
2. δ computes the next state from the current state and input



(a) *Using a* 1*-phase clock.*



(b) *Using a* 2*-phase clock.*

Figure 3: *Two generic FSM frameworks (for different clocking strategies) into which one can place implementations of the state, δ (the transition function) and ω (the output function).*

such that the next state is latched by the positive clock edge marking the next clock cycle. So we have a period of computation in which ω and δ operate, then an update triggered by a positive clock edge which steps the FSM from the current state into the next state. What results is a series of steps, under control of the clock, each performing some computation. As such, it should be clear that the clock frequency determines how quickly computation occurs; it has to be fast enough to to satisfy the design goals, yet slow enough to cope with the critical path of a given step of computation. That is, the faster the clock oscillates the faster we step though the computation, but if it is too fast we cannot finish one step before the next one starts.

To summarise, this is a framework for a *computer* we can *build*: we know how each of the components function, and can reason about their behaviour from the transistor-level upward. To solve a concrete problem using the framework, we follow a (fairly) standard sequence of steps:

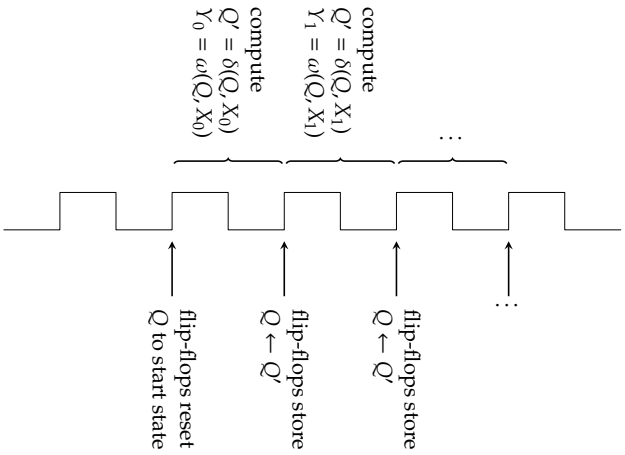
1. Count the number of states required, and give each state an abstract label.
2. Describe the state transition and output functions using a tabular or diagrammatic approach.
3. Decide how the states will be represented, i.e., assign concrete values to the abstract labels, and allocate a large enough 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.

Versus a theoretical alternative, it is less common for a hardware-based FSM to have have an accepting states since we cannot usually halt the circuit (without turning it off); we might include **idle** or **error** states to cope. In addition, and although the framework does not show it, it is common to have a **reset** input that (re)initialises the FSM into the start state. For one thing, this avoids the need to turn the FSM off then on again to reset it!

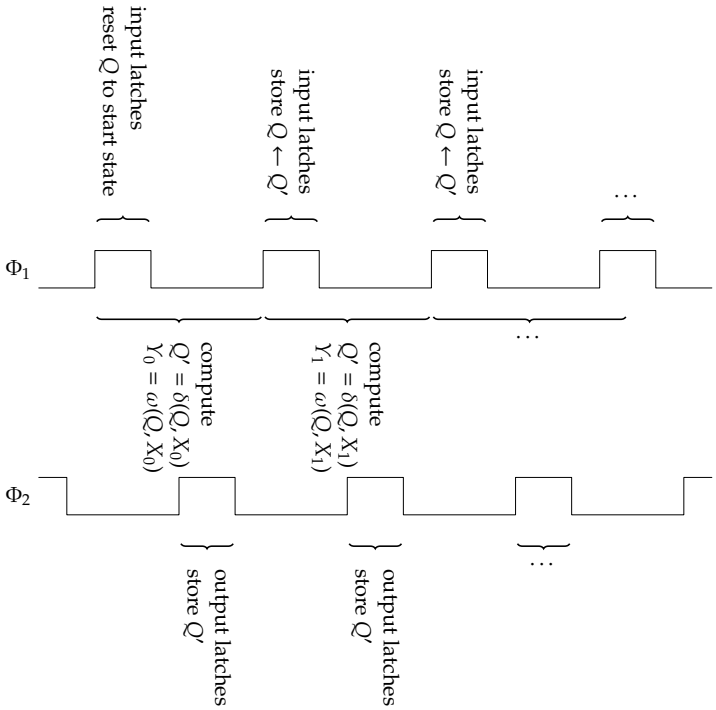
Example #1: an ascending modulo 6 counter Imagine we are tasked with designing an FSM that acts as a cyclic counter modulo n (rather than 2^n as before). If $n = 6$ for example, we want a component whose output r steps through values

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

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default). In this case it is clear the FSM can be in one of 6 states (since the counter value is is one of $0, 1, \ldots, 5$), which we label $S_0, S_1, \ldots, S_5$. Figure 5 includes tabular and diagrammatic descriptions of the transition function, both of which are a little dull: they simply move from one state to the next (with the ϵ meaning no input is required), cycling from S_5 back to S_0.



(a) *Using a* 1*-phase clock.*



(b) *Using a* 2*-phase clock.*

Figure 4: *Two illustrative waveforms (for different clocking strategies), outlining stages of computation within the associated FSM framework.*

An aside: binary versus one-hot encodings.

The fact that state assignment occurs quite late in the design of a given FSM is intentional: it allows us to optimise the representation based on what we do with it. So far, we have used a natural, binary encoding to represent the i-th of n states as a $(\lceil \log_2(n) \rceil)$-bit unsigned integer i. For example, if $n = 6$ we use

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

This is not the *only* option, however.

A **one-hot encoding** represents the i-th of n states as a sequence X st. $X_i = 1$ and $X_j = 0$ for $j \neq i$. For example, if $n = 6$ again, then we use

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

meaning that for S_0, the 0-th bit is 1 and all others are 0. On one hand, and depending on n, this might mean we need more flip-flops to store the state (i.e., n instead of $\lceil \log_2(n) \rceil$). On the other hand, we potentially get two advantages, namely

1. transition between states is easier (we simply rotate any given encoding by the right distance to get another), and
2. switching behaviour (and hence power consumption) is reduced since only two bits toggle for any change (one from 1 to 0, and one from 0 to 1).

Clearly $2^3 = 8 > 6$, so we can represent the current state using a 3-bit integer $Q = \langle Q_0, Q_1, Q_2 \rangle$. That is,

$$\begin{array}{lcll} S_0 & \mapsto & \langle 0,0,0 \rangle & \equiv & 000_{(2)} \\ S_1 & \mapsto & \langle 1,0,0 \rangle & \equiv & 001_{(2)} \\ S_2 & \mapsto & \langle 0,1,0 \rangle & \equiv & 010_{(2)} \\ S_3 & \mapsto & \langle 1,1,0 \rangle & \equiv & 011_{(2)} \\ S_4 & \mapsto & \langle 0,0,1 \rangle & \equiv & 100_{(2)} \\ S_5 & \mapsto & \langle 1,0,1 \rangle & \equiv & 101_{(2)} \end{array}$$

To implement the FSM, all we need to do is derive Boolean equations for the transition function δ so it can compute the next state Q' from Q; with this FSM there is no input, so δ is a function of the current state. To do so, we first rewrite the tabular description of δ by replacing the abstract labels with concrete values. The result is a truth table, i.e.,

			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	**?**	**?**	**?**	**?**	**?**	**?**

which encodes the same information. For example, if the current state is $Q = \langle 0,0,0 \rangle$ (i.e., we are in state S_0) then the next state should be $Q' = \langle 1,0,0 \rangle$ (i.e., state S_1). Note that there are 2 unused states, namely $\langle 0,1,1 \rangle$

An aside: Moore vs. Mealy style FSMs.

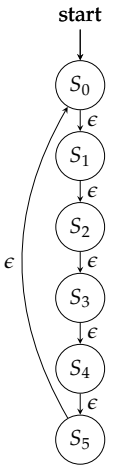
When written symbolically, the motivation for using either a Moore or Mealy style FSMs may be unclear. When the framework for *implementing* FSMs is taken into account, however, the issue should become more concrete:

- In a Moore FSM the output depends on the current state only, implying changes to any input are only relevant when the state is updated; you can think of this as meaning the inputs are only relevant in relation to the clock signal that triggers said update (i.e., they are only taken into account periodically, rather than continuously).
- In contrast, a Mealy FSM allows the output to depend on the current state *and* any input. ω is a combinatorial function, so this implies the output can change a) in relation to the clock signal as a result up an update to the state, and/or b) at *any* time as a result of changes to the input. You could think of this as meaning the FSM is more responsive, in the sense that although the state is updated at the same frequency (i.e., in relation to the same features of the clock) the output can continuously, and *instantaneously* change if/when the input changes.

Both are viable options, so it is not true that one is correct or incorrect. However, it is clearly important to understand the (subtle) difference so an informed choice can be made within some specific context.

	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

(a) *A tabular description.*

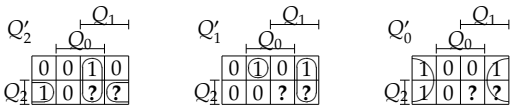


(b) *A diagrammatic description.*

Figure 5: *An example FSM modelling an ascending modulo 6 counter.*

and $\langle 1, 1, 1 \rangle$, which we include in the table: the next state in either of these cases does not matter since they are invalid, so the entries are don't care.

To summarise, we need to derive Boolean expressions for each of Q'_2, Q'_1 and Q'_0 in terms of Q_2, Q_1 and Q_0. This can be achieved by applying the Karnaugh map technique to get



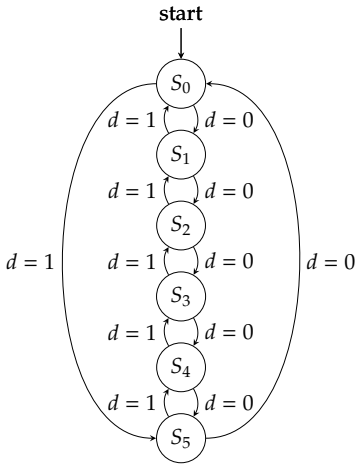
which produce

$$\begin{aligned}
Q'_2 &= (\quad\quad\quad\quad Q_1 \wedge \quad Q_0 \quad) \vee \\
&\quad\ (\quad Q_2 \wedge \quad\quad\quad \neg Q_0 \quad) \\
\\
Q'_1 &= (\ \neg Q_2 \wedge \neg Q_1 \wedge \quad Q_0 \quad) \vee \\
&\quad\ (\quad\quad\quad\ Q_1 \wedge \neg Q_0 \quad) \\
\\
Q'_0 &= (\quad\quad\quad\quad\quad\quad \neg Q_0 \quad)
\end{aligned}$$

Now we have enough to fill in the FSM framework: the state is simply a 3-bit register, δ is represented by

	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

(a) *A tabular description.*

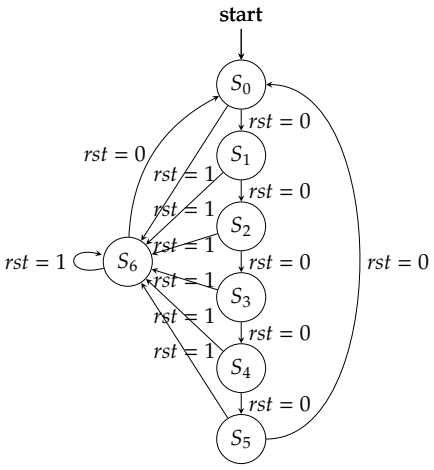


(b) *A diagrammatic description.*

Figure 6: *An example FSM modelling an ascending or descending modulo 6 counter.*

	δ		ω					
Q	Q'		M_g	M_a	M_r	A_g	A_a	A_r
	$rst=0$	$rst=1$						
S_0	S_1	S_6	1	0	0	0	0	1
S_1	S_2	S_6	0	1	0	0	0	1
S_2	S_3	S_6	0	0	1	0	1	0
S_3	S_4	S_6	0	0	1	1	0	0
S_4	S_5	S_6	0	0	1	0	1	0
S_5	S_0	S_6	0	1	0	0	0	1
S_6	S_0	S_6	0	0	1	0	0	1

(a) *A tabular description.*



(b) *A diagrammatic description.*

Figure 7: *An example FSM modelling a traffic light controller.*

circuit analogues of the expressions above. Note that tn this case, the output function ω is trivial: the counter output $r = Q$ due to our state assignment, so in a sense ω is just the identity function.

Example #2: an ascending or descending modulo 6 counter No imagine we need to upgrade the previous example: we are tasked with designing an FSM that again 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

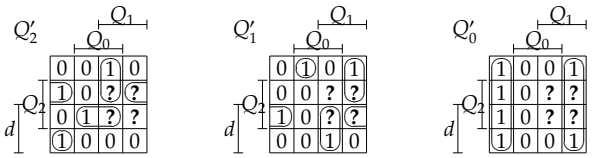
$$0, 5, 4, 3, 2, 1, 0, 5, \ldots$$

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).

The possible states are the same as before: we still have 6 states, labelled $S_0, S_1, \ldots S_6$. The difference is how transitions between states occur; this is illustrated by Figure 6, in which the new tabular and diagrammatic descriptions of the transition function are shown. Although it *looks* more complicated, we take exactly the same approach as before: we start by rewriting the tabular description of δ by replacing the abstract labels with concrete values to yield:

				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	**?**	**?**	**?**	**?**	**?**	**?**	**?**

The table is larger since we need to consider d as input as well as Q, but the process is the same: to compute δ, we just need a set of appropriate Boolean expressions. So next we translate the truth table into a set of Karnaugh maps



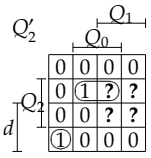
and finally produce

$$\begin{aligned}
Q'_2 = (\ & \neg d & & & \wedge & Q_1 & \wedge & Q_0 &)\ \vee \\
(\ & \neg d & \wedge & Q_2 & & & \wedge & \neg Q_0 &)\ \vee \\
(\ & d & \wedge & Q_2 & & & \wedge & Q_0 &)\ \vee \\
(\ & d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &)
\end{aligned}$$

$$\begin{aligned}
Q'_1 = (\ & \neg d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & Q_0 &)\ \vee \\
(\ & \neg d & & & \wedge & Q_1 & \wedge & \neg Q_0 &)\ \vee \\
(\ & d & \wedge & Q_2 & & & \wedge & \neg Q_0 &)\ \vee \\
(\ & d & & & \wedge & Q_1 & \wedge & Q_0 &)
\end{aligned}$$

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

This time however, we need to deal with ω more carefully: we can still generate the counter output trivially as $r = Q$, but also need to compute f somehow. This is straight-forward of course, because using the truth table we can write



and finally produce

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

which completes the design in the sense we have now specified all components of the framework.

Example #3: a traffic light controller Imagine we are tasked with designing a traffic light controller for two roads (a main road and an access road) that intersect. The requirements are to

1. stop cars crashing into each other, so the behaviour should see
   (a) green on main road and red on access road, then
   (b) amber on main road and red on access road, then
   (c) red on main road and amber on access road, then
   (d) red on main road and green on access road, then
   (e) red on main road and amber on access road, then
   (f) amber on main road and red on access road,

   and then cycle, and
2. allow an emergency stop button to force red on both main and access roads while pushed, then reset the system into an initial start state when released.

First we need to take stock of the problem itself: there is basically one input (the emergency stop button, denoted *rst*) and six outputs (namely the traffic light values, denoted M_g, M_a and M_r for the main road and A_g, A_a and A_r for the access road). Next we try to develop a precise description of the FSM behaviour. We need 7 states in total: $S_0, S_1, \ldots, S_5$ represent steps in the normal traffic light sequence, and S_6 is an extra emergency stop state. Figure 7 shows both tabular and diagrammatic descriptions of the transition function; in essence, it is similar to the counter example (in the sense that it cycles from S_0 through to S_5 and back again) provided $rst = 0$, but if $rst = 1$ in *any* state then we move to the S_6. As an aside however, it is important to see this description represents *one* solution among several derived from what is (by design) an imprecise question. Put another way, we have already made several choices. On example is the decision to use a separate emergency stop state, and have the FSM enter this as the next state of *any* current state provided $rst = 1$; the red lights are both forced on by virtue of being in the emergency stop state, rather than by *rst* per se. Another valid approach might be to have ω depend on *rst* as well (rather than just Q, so it turns from a Moore-based into a Mealy-based FSM) and forcing the red lights on as soon as $rst = 1$ and irrespective of what state the FSM is in. In some ways this is arguably more attractive, in the sense that the emergency stop is instant: we no longer need to wait for the next clock cycle when the next state is latched. Likewise, we have opted to make the first state listed in the question (i.e., green on the main road and red on the access road) the initial state; since the sequence is cyclic this choice seems a little arbitrary, so other choices (plus what state the FSM restarts in after an emergency stop) might also seem reasonable.

Given our various choices however, we next follow standard practice by translating the description into an implementation. Since $2^3 = 8 > 7$ we can represent the current and next states via 3-bit integers $Q = \langle Q_0, Q_1, Q_2 \rangle$ and $Q' = \langle Q_0', Q_1', Q_2' \rangle$. where

$$\begin{array}{lcll} S_0 & \mapsto & \langle 0,0,0 \rangle & \equiv 000_{(2)} \\ S_1 & \mapsto & \langle 1,0,0 \rangle & \equiv 001_{(2)} \\ S_2 & \mapsto & \langle 0,1,0 \rangle & \equiv 010_{(2)} \\ S_3 & \mapsto & \langle 1,1,0 \rangle & \equiv 011_{(2)} \\ S_4 & \mapsto & \langle 0,0,1 \rangle & \equiv 100_{(2)} \\ S_5 & \mapsto & \langle 1,0,1 \rangle & \equiv 101_{(2)} \\ S_6 & \mapsto & \langle 0,1,1 \rangle & \equiv 110_{(2)} \end{array}$$

and we have one unused state (namely $\langle 1,1,1 \rangle$). As such, both input and output registers will be comprised of three 1-bit storage components, in this case D-type latches. Now we have a concrete value for each abstract

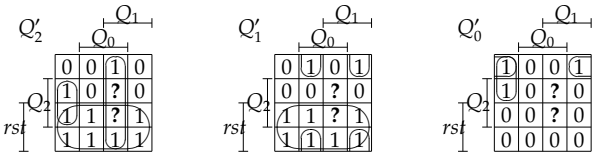
state label, we can expand the tabular description of the FSM into a (lengthy) truth table:

				δ			ω					
rst	Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0	M_g	M_a	M_r	A_g	A_a	A_r
0	0	0	0	0	0	1	1	0	0	0	0	1
0	0	0	1	0	1	0	0	1	0	0	0	1
0	0	1	0	0	1	1	0	0	1	0	1	0
0	0	1	1	1	0	0	0	0	1	1	0	0
0	1	0	0	1	0	1	0	0	1	0	1	0
0	1	0	1	0	0	0	0	1	0	0	0	1
0	1	1	0	0	0	0	0	0	1	0	0	1
0	1	1	1	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**
1	0	0	0	1	1	0	1	0	0	0	0	1
1	0	0	1	1	1	0	0	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1	0	1	0
1	0	1	1	1	1	0	0	0	1	1	0	0
1	1	0	0	1	1	0	0	0	1	0	1	0
1	1	0	1	1	1	0	0	1	0	0	0	1
1	1	1	0	1	1	0	0	0	1	0	0	1
1	1	1	1	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**

Although this *looks* intimidating, the point is that

- the transition function δ is just three Boolean expressions, one for each Q'_i, using rst, Q_2, Q_1 and Q_0 as input,
- the output function ω is just six Boolean expressions, one for each M_i and A_j, using rst, Q_2, Q_1 and Q_0 as input.

So we just need to derive each expression. For δ, the Karnaugh maps



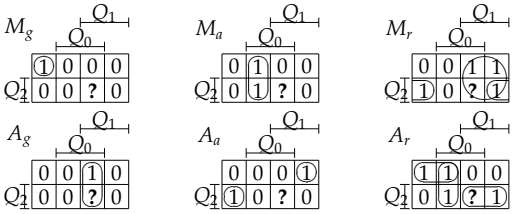
can be used to produce

$$
\begin{aligned}
Q'_2 = (\ & rst & & & & & & &) \vee \\
(\ & & & Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \vee \\
(\ & & & & & Q_1 & \wedge & Q_0 &)
\end{aligned}
$$

$$
\begin{aligned}
Q'_1 = (\ & rst & & & & & & &) \vee \\
(\ & & & \neg Q_2 & \wedge & \neg Q_1 & \wedge & Q_0 &) \vee \\
(\ & & & \neg Q_2 & \wedge & Q_1 & \wedge & \neg Q_0 &)
\end{aligned}
$$

$$
\begin{aligned}
Q'_0 = (\ & \neg rst & & & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \vee \\
(\ & \neg rst & \wedge & \neg Q_2 & & & \wedge & \neg Q_0 &)
\end{aligned}
$$

Likewise for ω, we find



can be used to produce

$$
\begin{aligned}
M_g = (\ & \neg Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \\
M_a = (\ & & & \neg Q_1 & \wedge & Q_0 &) \\
M_r = (\ & & & Q_1 & & &) \vee \\
(\ & Q_2 & & & \wedge & \neg Q_0 &)
\end{aligned}
$$

$$
\begin{aligned}
A_g = (\ & & & Q_1 & \wedge & Q_0 &) \\
A_a = (\ & \neg Q_2 & \wedge & Q_1 & \wedge & \neg Q_0 &) \vee \\
(\ & Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \\
A_r = (\ & \neg Q_2 & \wedge & \neg Q_1 & & &) \vee \\
(\ & & & \neg Q_1 & \wedge & Q_0 &) \vee \\
(\ & Q_2 & \wedge & Q_1 & & &)
\end{aligned}
$$

As before, these expressions can be used to fill in the FSM framework to yield a resulting design for the controller.