COMS12200 lab. worksheet: week #6

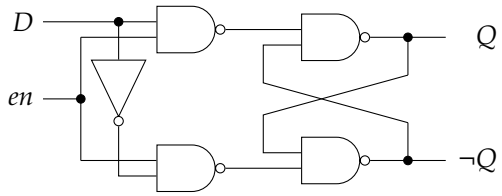
Although some questions have a written solutions below, for others it makes more sense to experiment in a hands-on manner: the Logisim project

http://www.cs.bris.ac.uk/home/page/teaching/material/arch_new/sheet/lab-6.s.circ

supports such cases.

Note that the example solutions for this worksheet start to make a *real* effort to tame implementation complexity; this is achieved by first decomposing components into sub-components where appropriate, then use of more advanced features of Logisim. A good example is the tunnel component. This acts like a "magic" wire-less connection between two points, without the need to connect them with an actual wire: any number of tunnel components placed on the canvas are connected iff. they share the same label. For large designs this removes the problem of having to route long wires (potentially avoiding other components), meaning a more organised and easily understandable result.

S1. A NAND-based D-type latch design is as follows:



Notice the internal SR-type latch is still evident, but now implemented via cross-coupled NAND (rather than NOR) gates. One result of this change is that the storage and meta-stable states are swapped over.

Imagine we use S' and R' to denote the internal latch inputs. With the NOR-based version, if $S' = R' = 1$ then we *must* have $Q = \neg Q = 0$: given $Q = R \overline{\vee} \neg Q$ and $\neg Q = S \overline{\vee} Q$, this represents the only possibility but is also invalid in that $\neg Q$ should be the inverse of Q. As such, the latch behaviour when $S' = R' = 1$ cannot be determined, and we say it is in a meta-stable state. We have a similar situation with the NAND-based version, in that if $S' = R' = 0$ then we *must* have $Q = \neg Q = 1$ which is again invalid.

Since the NAND-based latch is now in storage mode when $S' = R' = 1$, we need to alter how it is enabled. With the NOR-based version we previously did this by setting

$$\begin{array}{rcl} S' & = & S \wedge en \\ R' & = & R \wedge en \end{array}$$

The idea was that since $t \wedge 0 = 0$ and $t \wedge 1 = t$ for any t, en gated (or conditionally turned off) S and R: if $en = 0$ then $S' = R' = 0$ irrespective of S and R so the internal latch behaves in storage mode, but if $en = 1$ then $S' = S$ and $R' = R$ so the internal latch behaves as normal. The NAND-based version needs $S' = R' = 1$ when $en = 0$, so we alter the design st.

$$\begin{array}{rcl} S' & = & S \overline{\wedge} en \\ R' & = & R \overline{\wedge} en \end{array}$$

Now we have $t \overline{\wedge} 0 = 1$ and $t \overline{\wedge} 1 = \neg t$ for any t, so en still acts to gate S and R (albeit with S and R swapped due the NOT).

Implementation of this design in Logisim is trivial, and not *too* much harder on the NAND board: the design is already close to being written in NAND-only form, so can be fairly directly implemented as

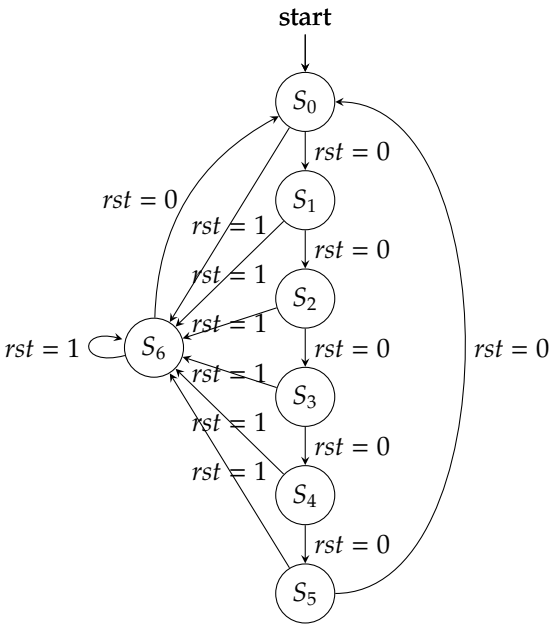
$$\begin{array}{rcccl} t_0 & = & D & \overline{\wedge} & D \\ t_1 & = & D & \overline{\wedge} & en \\ t_2 & = & t_0 & \overline{\wedge} & en \\ t_3 & = & t_4 & \overline{\wedge} & t_1 \\ t_4 & = & t_3 & \overline{\wedge} & t_2 \end{array}$$

where $Q \equiv t_3$ and $\neg Q \equiv t_4$.

S3. 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). We *know* that Figure 2 acts as a starting point, in the sense that we just need to fill in each part (e.g., δ) with problem-specific implementations.

Q	δ		ω					
	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 1: *An FSM for the traffic light controller.*

So 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 1 shows both a tabular and diagrammatic description 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

$$\begin{array}{lcl} S_0 & \mapsto & \langle 0,0,0 \rangle \\ S_1 & \mapsto & \langle 1,0,0 \rangle \\ S_2 & \mapsto & \langle 0,1,0 \rangle \\ S_3 & \mapsto & \langle 1,1,0 \rangle \\ S_4 & \mapsto & \langle 0,0,1 \rangle \\ S_5 & \mapsto & \langle 1,0,1 \rangle \\ S_6 & \mapsto & \langle 0,1,1 \rangle \end{array}$$

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

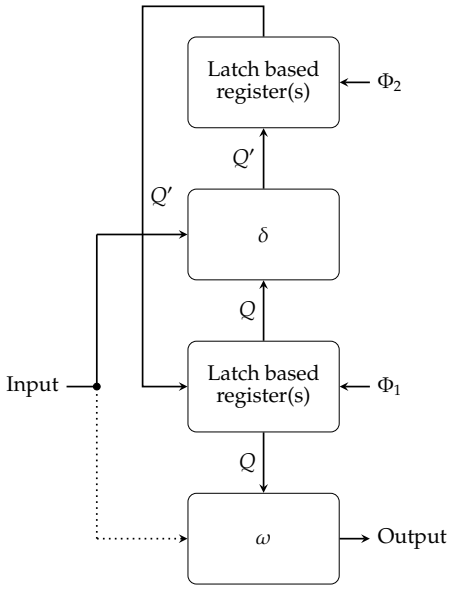


Figure 2: *A generic FSM framework (for a 2-phase clocking strategy) into which one can place implementations of the state, δ (the transition function) and ω (the output function).*

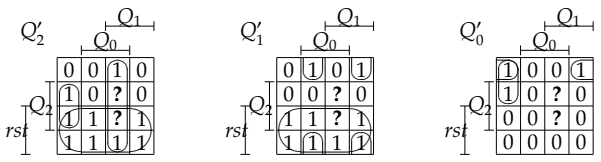
state label, we can expand the tabular description of the FSM into a (lengthy) truth table:

				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.

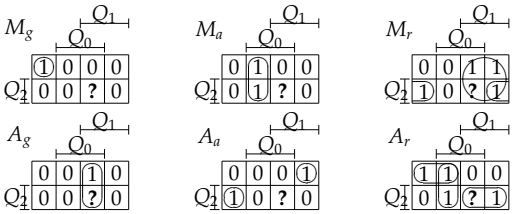
So we just need to derive each expression. For δ, the Karnaugh maps



can be used to produce

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

Likewise for ω, we find



can be used to produce

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

The final step is to implement everything in Logisim. Although this requires some effort, conceptually it just means reproducing each of expressions for δ and ω in terms of logic gates (the example solution uses two sub-components to make this easier to manage), and then using them within an implementation of Figure 2 (which of course requires instantiating the input and output registers).