Intro. to Computer Architecture

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

January 9, 2018

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

- Recap:

Definition

A (deterministic) **Finite State Machine (FSM)** is a tuple

$$C = (S, s, A, \Sigma, \Gamma, \delta, \omega)$$

including

1. S, a finite set of **states** that includes a **start state** $s \in S$,
2. $A \subseteq S$, a finite set of **accepting states**,
3. an **input alphabet** Σ and an **output alphabet** Γ,
4. a **transition function**

$$\delta : S \times \Sigma \rightarrow S$$

and

5. an **output function**

$$\omega : S \rightarrow \Gamma$$

in the case of a **Moore FSM**, or

$$\omega : S \times \Sigma \rightarrow \Gamma$$

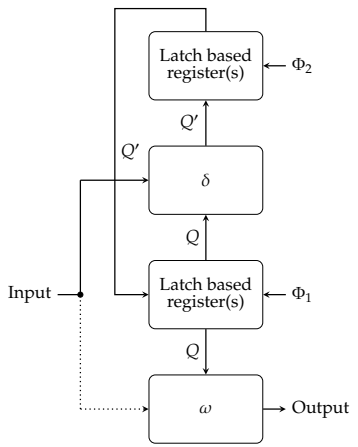
in the case of a **Mealy FSM**,

noting an **empty** input denoted ϵ allows a transition that can *always* occur.

Notes:

FSMs in Hardware (1)

Algorithm (latch version)

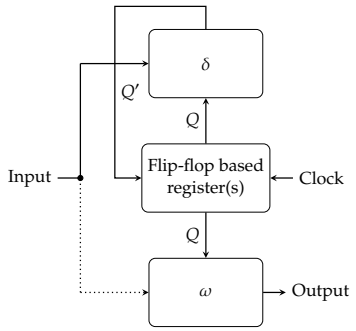


- Note that
  1. the state is retained in a register (i.e., a group of latches, resp. flip-flops),
  2. δ and ω are simply combinatorial logic,
  3. within the current clock cycle
     - 3.1 ω computes the output from the current state and input, and
     - 3.2 δ computes the next state from the current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

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

Notes:

Algorithm (flip-flop version)

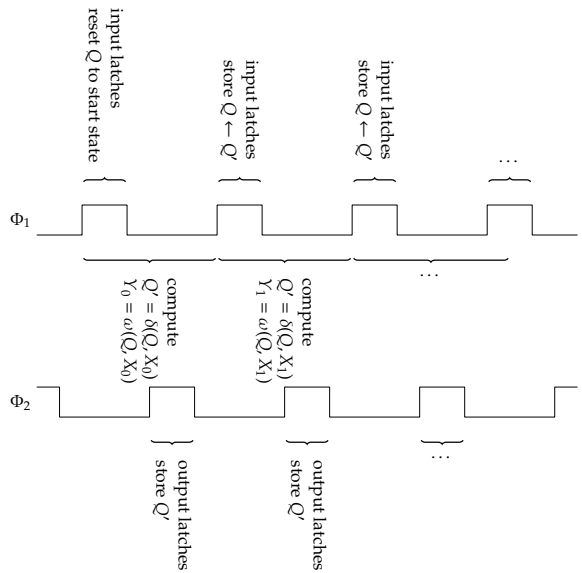


- Note that
  1. the state is retained in a register (i.e., a group of latches, resp. flip-flops),
  2. δ and ω are simply combinatorial logic,
  3. within the current clock cycle
     - 3.1 ω computes the output from the current state and input, and
     - 3.2 δ computes the next state from the current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

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

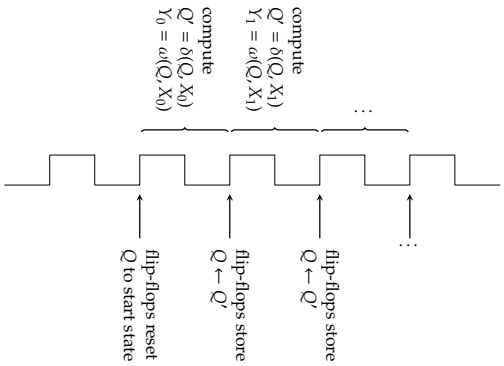
Notes:

Example (latch version: given input $X = \langle X_0, X_1, \ldots \rangle$ and output $Y = \langle Y_0, Y_1, \ldots \rangle$)



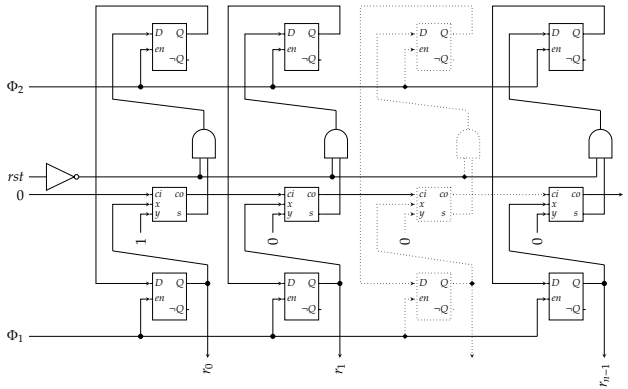
Notes:

Example (flip-flop version: given input $X = \langle X_0, X_1, \ldots \rangle$ and output $Y = \langle Y_0, Y_1, \ldots \rangle$)



Notes:

Circuit (latch version)



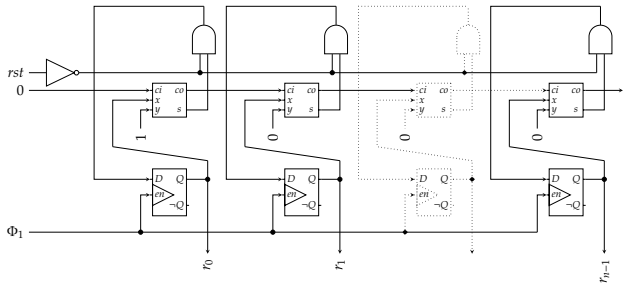
- This *should* sound familar:
  - 2^n states, labelled S_0 through S_{2^n-1}; state S_i represented as (unsigned) n-bit integer i,
  - the start state is $s = S_0$ and there are no accepting states (so $A = \emptyset$),
  - the δ function is

$$Q' \leftarrow \delta(Q, rst) = \begin{cases} Q + 1 \pmod{2^n} & \text{if } rst = 0 \\ 0 & \text{if } rst = 1 \end{cases}$$

  - the ω function is $r \leftarrow \omega(Q) = Q$.

Notes:

Circuit (flip-flop version)



- This *should* sound familar:
  - 2^n states, labelled S_0 through S_{2^n-1}; state S_i represented as (unsigned) n-bit integer i,
  - the start state is $s = S_0$ and there are no accepting states (so $A = \emptyset$),
  - the δ function is

$$Q' \leftarrow \delta(Q, rst) = \begin{cases} Q + 1 \pmod{2^n} & \text{if } rst = 0 \\ 0 & \text{if } rst = 1 \end{cases}$$

  - the ω function is $r \leftarrow \omega(Q) = Q$.

Notes:

FSMs in Hardware (7)

- 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. Perform state assignment, i.e., decide how concrete values will represent the abstract labels, allocating appropriate register(s) 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.

noting that it's common to

- include a **reset** input that (re)initialises the FSM into the start state,
- replace the accepting state(s) with an **idle** or **error** state since "halting" doesn't make sense in hardware, and
- use the FSM to control an associated data-path using the outputs, rather than (necessarily) solve some problem outright.

Notes:

FSMs in Hardware (8)

- 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, e.g.,

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

  2. A **one-hot encoding** represents the i-th of n states as a sequence X st. $X_i = 1$ and $X_j = 0$ for $j \neq i$, e.g.,

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

     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:

Examples (1)
A modulo 6 ascending counter

Notes:

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

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

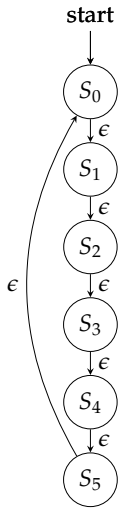
with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

Examples (2)
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:

Examples (3)
A modulo 6 ascending counter

- To implement the design via the framework, we need a state assignment step:
  - There are 6 abstract labels

$$\begin{array}{lcl} S_0 & \mapsto & 0 \\ S_1 & \mapsto & 1 \\ S_2 & \mapsto & 2 \\ S_3 & \mapsto & 3 \\ S_4 & \mapsto & 4 \\ S_5 & \mapsto & 5 \end{array}$$

    representing the integers $0, 1, \ldots, 5$.
  - Since $2^3 = 8 > 6$, we can represent them using 6 concrete values, namely

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

    and capture

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

    in a 3-bit register (i.e., via 3 latches or flip-flops).

Notes:

Algorithm (truth table)

Rewriting the abstract labels yields the following concrete truth table

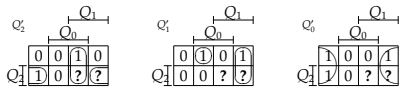
			δ			ω		
Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0	r_2	r_1	r_0
0	0	0	0	0	1	0	0	0
0	0	1	0	1	0	0	0	1
0	1	0	0	1	1	0	1	0
0	1	1	1	0	0	0	1	1
1	0	0	1	0	1	1	0	0
1	0	1	0	0	0	1	0	1
1	1	0	?	?	?	?	?	?
1	1	1	?	?	?	?	?	?

noting that our state assignment means $r = Q$, st. $r = \omega(Q) = Q$ is basically just the identity function (so we'll ignore it).

Notes:

Circuit (δ)

Translating the truth table into a set of Karnaugh maps



yields the following Boolean expressions for δ:

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

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

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

Notes:

Question

Design an FSM that controls traffic lights at the intersection of two roads (a main road and an access road); it should

1. stop cars crashing into each other, so the behaviour should see
   - green on main road and red on access road, then
   - amber on main road and red on access road, then
   - red on main road and amber on access road, then
   - red on main road and green on access road, then
   - red on main road and amber on access road, then
   - amber on main road and red on access road,

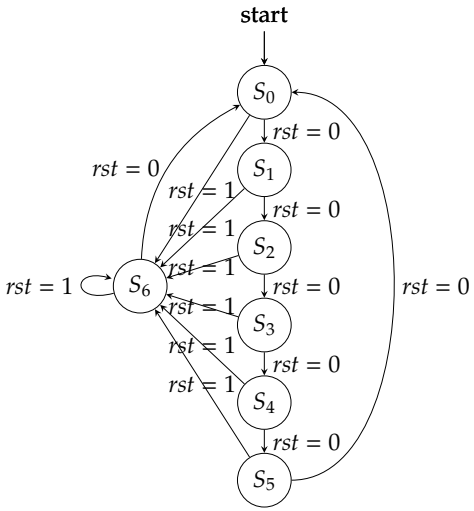
   and then cycle, and
2. allow an emergency stop button to force red on both main and access roads while pushed, then reset the system into an initial start state when released.

Notes:

Algorithm (tabular)

	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

Algorithm (diagram)



Notes:

Algorithm (truth table)

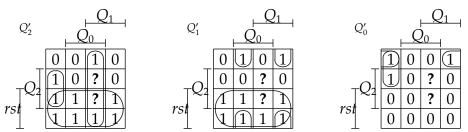
Rewriting the abstract labels yields the following concrete truth table:

				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	?	?	?	?	?	?	?	?	?

Notes:

Circuit (δ)

Translating the truth table into a set of Karnaugh maps



yields the following Boolean expressions for δ:

$$
\begin{array}{rcl}
Q'_2 &=& (\; rst \;) \lor \\
&& (\; Q_2 \land \neg Q_1 \land \neg Q_0 \;) \lor \\
&& (\; Q_1 \land Q_0 \;)
\end{array}
$$

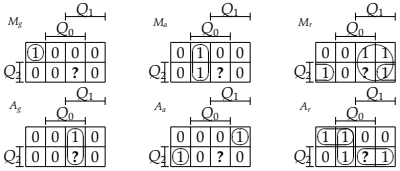
$$
\begin{array}{rcl}
Q'_1 &=& (\; rst \;) \lor \\
&& (\; \neg Q_2 \land \neg Q_1 \land Q_0 \;) \lor \\
&& (\; \neg Q_2 \land Q_1 \land \neg Q_0 \;)
\end{array}
$$

$$
\begin{array}{rcl}
Q'_0 &=& (\; \neg rst \land \neg Q_1 \land \neg Q_0 \;) \lor \\
&& (\; \neg rst \land \neg Q_2 \land \neg Q_0 \;)
\end{array}
$$

Notes:

Circuit (ω)

Translating the truth table into a set of Karnaugh maps



yields the following Boolean expressions for ω:

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

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:

Additional Reading

- *Wikipedia: Finite State Machine (FSM)*. . URL: http://en.wikipedia.org/wiki/Finite-state_machine.
- D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009.
- M. Sipster. "Chapter 1: Regular languages". In: *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006.

Notes:

References

[1] *Wikipedia: Finite State Machine (FSM)*. URL: http://en.wikipedia.org/wiki/Finite-state_machine (see p. 45).

[2] D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009 (see p. 45).

[3] M. Sipster. "Chapter 1: Regular languages". In: *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006 (see p. 45).

Notes: