ENEL260 CA5: Sequential Logic

Ciaran Moore

"Almost every object that computes is naturally viewed as a state machine"
– Leslie Lamport

Storing *state* (both programs, and intermediate results of programmed computations) is part of what makes computers so powerful and flexible. So how can we use stored state in digital logic circuits?

Although combinational logic can produce some useful outputs, much of what makes computers interesting is their ability to create complex behaviour. Behaviour arises from *state* and changes in state over time. Which brings us to *sequential logic* circuits (see Fig. 1).

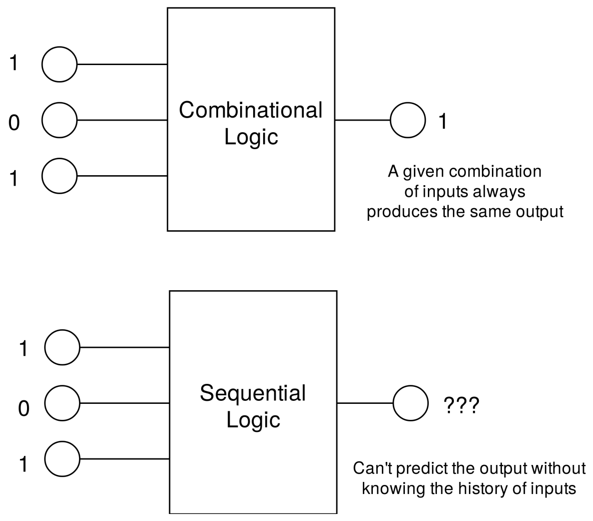
1
0
1
Combinational Logic
1
A given combination of inputs always produces the same output
1
0
1
Sequential Logic
???
Can't predict the output without knowing the history of inputs

Figure 1: Combinational logic and Sequential logic

Sequential logic circuits are characterised by feedback, and feedback provides the property of *memory*[1]. We previously looked at the edge-triggered D-type flip-flop, which stores a single bit, and a register built from an array of D flip-flops, which can store a sequence of bits. As we saw, the D flip-flop uses an explicit *clocking signal* to make the design of reliable circuits based on the flip-flop easier. The use of an explicit clock makes the D flip-flop a *synchronous*[2] sequential circuit.

[1] I.e., the ability to retain state.

[2] Meaning that changes in state throughout the circuit are synchronized by a common clock signal. The majority of digital logic is designed as synchronous circuits.

Typical synchronous circuits include:

- Registers
- Counters
- Timers
- State machines (including CPUs)

Registers

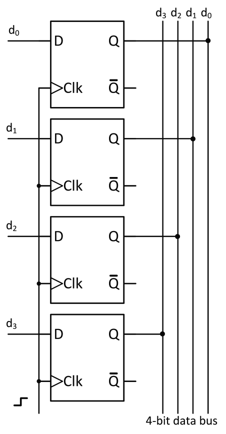

Figure 2: A 4-bit register built from D flip-flops

As we saw in a previous lecture, we can collect together a set of flip-flops, all clocked from a single clock waveform, to form a *register* that stores a group of bits. Fig. 2 shows how a register can be implemented using D-type flip-flops.

But a simple register isn't the only way that we can organise a group of flip-flops. Other configurations are possible, depending on how the inputs and outputs of the flip-flops are connected. Two important flip-flop configurations are the *shift register*, and the *counter*.

Shift Registers

A *shift register* can be created by connecting the output of one flip-flop to the input of the next. For example, the shift register in Fig. 3 is constructed from four D-type flip flops. This particular version is designed to shift bits towards the righthand output. The register can straightforwardly be extended to more bits. A bit input as p_i at the ith clock edge appears at the output as q_{i+4}. This circuit can therefore implement a fixed delay.

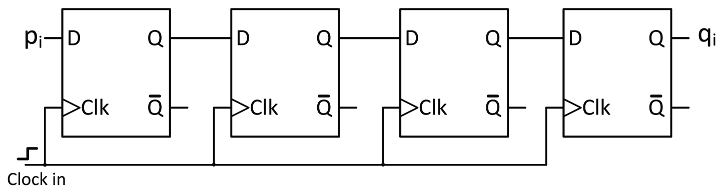

Figure 3: A 4-bit shift register constructed from edge-triggered D-type flip-flops.

Although the register in Fig. 3 takes its input one bit at a time (i.e., serially), more sophisticated shift register designs can be loaded in parallel.

Example: An 8-bit shift register that can be loaded in parallel allows a byte to be loaded in a single clock cycle. Once loaded into the register, the bits of the byte can be presented, one bit at a time, at the output[3].

For example, if the pattern 01101001 is loaded into the register, the individual bits 1, 0, 0, etc. can be shifted out and transmitted (assuming a 'little endian' order). At the receiving end, an identical shift register can be used to reassemble the pattern for storage and further processing.

---

[3] This is useful when data is transmitted across a serial line, such as a USB port, or an Ethernet connection.

In later lectures you'll encounter the "UART" (*Universal Asynchronous Receiver Transmitter*), which is one of the common microcomputer peripherals, and which provides a shift register capability for serial communications.

Counters

A *counter* can be created by introducing some kind of feedback between the output and input of a D-type flip-flop[4]. For a single-bit counter, simply using the inverted output ($\overline{Q}$) of the flip-flop as

[4] Alternative designs are possible by using J-K flip-flops.

the input[5] will cause the output to toggle between 0 and 1 on each clock cycle.

[5] You may recognize this idea from the second set of tutorial questions

---

Example: The single-bit counter can be extended to create a multi-bit counter, such as Fig. 4, by clocking each additional flip-flop using a clock signal derived from the previous flip-flop output (this is known as a "ripple counter").

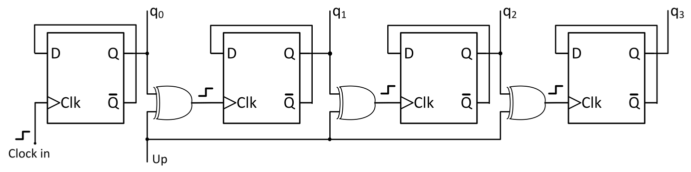

Figure 4: An example of a **ripple** 4-bit counter. This counter can either count up or down.

The counter output is $q_3\,q_2\,q_1\,q_0$ with MSB q_3. In this case, the counter is designed to be controllable for counting up or down, according to the '`Up`' control signal. Thus the second flip-flop is triggered by one of the changes to q_0, either a $1 \rightarrow 0$ transition for counting up or a $0 \rightarrow 1$ transition for counting down.

Remembering the truth table for an edge triggered D-type flip flop and for an XOR gate, convince yourself that the circuit in Fig. 4 generates the correct sequence.

The design in Fig. 4 is called a *ripple* counter, because of the way one flip-flop triggers the next in sequence. This means that the changes in the outputs are not *synchronous*; rather, it takes a finite time from the initiating $0 \rightarrow 1$ transition in '`Clock in`' until the outputs settle to their new values. Clearly, if the design was for a (say) 32-bit counter, this time could be quite significant. A synchronous design, such as Fig. 5, is a little more complicated than the ripple counter, but is usually considered to be superior.

Example: A *synchronous* counter must have all of the flip-flops clocked by the same signal. So we need to find a different way to update the value stored in each flip-flop than we used for the ripple counter. We can do this by thinking about what we want the *next* state[6] of each flip-flop to be, given the current state of all of the flip-flops.

[6] Since the state of a flip-flop output is just whatever the last input was, knowing the desired next state is equivalent to knowing what the flip-flop inputs need to be.

Let's look at a 3-bit counter with outputs $Q0, Q1$, and $Q2$, where the current state represents the binary count, $Q2Q1Q0$, at the present timestep. The next state should increment that count by 1 (and wraparound to 0 when the counter overflows). We can capture this relationship between current and next states as a truth table:

Here the subscript i represents the current timestep, and $i+1$ is the next timestep.

Current			Next		
$Q2_i$	$Q1_i$	$Q0_i$	$Q2_{i+1}$	$Q1_{i+1}$	$Q0_{i+1}$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Aside: We're looking here at binary counters. It's also possible to make a *decade counter* that moves from 1001 (9) to 0000 (0) in the upwards counting sequence. The sequence is known as binary coded decimal (BCD).

Inspecting this table, we can see that the next $Q0$ is 1 whenever the current $Q0$ is 0, and vice versa. That is:

$$Q0_{i+1} = \overline{Q0}_i.$$

Determining the inputs for the other two flip-flops is a little more complicated.

Looking at the $Q1_{i+1}$ column, we can see that the 1's and 0's come in groups of two. What characterises each group? Looking at the current state, we can see that whenever $Q0_i$ and $Q1_i$ are different, $Q1_{i+1}$ is 1. And whenever $Q0_i$ and $Q1_i$ are the same, $Q1_{i+1}$ is 0. In other words:

$$Q1_{i+1} = Q0_i \oplus Q1_i.$$

For $Q2$, the 1's come in groups of four. Looking at the first 4 rows (i.e., when $\overline{Q2}_i$ is true) we can see that

$$Q2_{i+1} = \overline{Q2}_i Q1_i Q0_i,$$

while the last 4 rows (when $Q2_i$ is true) gives us

$$Q2_{i+1} = Q2_i \overline{Q1_i Q0_i}.$$

Combining these two expressions, we get

$$Q2_{i+1} = \overline{Q2}_i Q1_i Q0_i + Q2_i \overline{Q1_i Q0_i} = Q2_i \oplus (Q1_i Q0_i).$$

Fig. 5 shows an implementation of the next-state logic using logic gates.

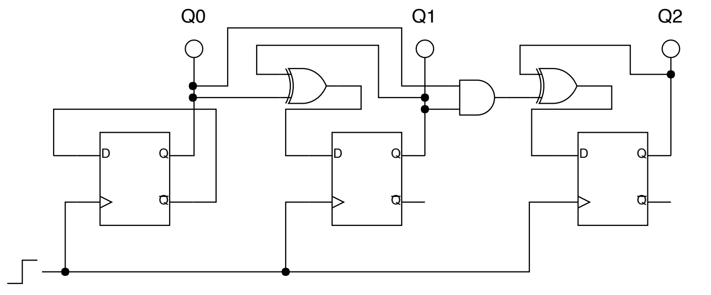

Figure 5: An example of a **synchronous** 3-bit counter

Timers

Timers are special counters that are provided as peripherals on microcontrollers. Timers are used for setting up regular internal events, for measuring the timing of external events, and for generating waveforms. For example, the AVR ATmega32U2 microcontroller has several timers. To quote from the ATmega32U2 datasheet, the microcontroller has:

```
∗ One 8-bit Timer/Counter with Separate Prescaler and Compare Mode
    (two 8-bit PWM channels)
∗ One 16-bit Timer/Counter with Separate Prescaler, Compare and Capture Mode
    (three 8-bit PWM channels)
```

We'll look at timers in more detail in a later lecture, but for now:

- The *'prescaler'* allows the timer to be clocked at a rate lower than the microcontroller itself
- The *'compare'* mode allows the count to be compared to a number pre-loaded into a special register
- The *'capture'* mode allows the instant of an external event to be timed
- *'PWM channels'* refers to generating timer-based digital waveforms

---

Example: Imagine that you want an LED to flash on and off at a particular frequency. Here are two possible ways to accomplish that task:

- *Method 1:* Write a C function that's designed to perform a short calculation a pre-selected number of times[7]; i.e., the function generates a time delay. Implement a `while`-loop that uses an I/O instruction to turn the LED on (via an I/O port), calls the delay function, turns the LED off, calls the delay function, and repeats. The microcontroller is thus directly and continuously working to keep the LED flashing.
- *Method 2:* Set up one of the timers to count up from 0 at a specific clock rate until it reaches a particular maximum number, N, then down until it reaches 0; the sequence then repeats (without the microcontroller's intervention). The count is constantly compared with the number $N/2$; if count $\geq N/2$, an output connected to the LED is set to 1 to turn the LED on, and if count $< N/2$, the output is 0 and the LED off. By initially choosing the clock rate and N, we can precisely set the cycling rate for the LED. The program executing on the microcontroller simply configures the timer and sets it to run.

Both these approaches will cause the LED to flash. The main difference is that adding extra processing to the program in Method 1 will require the delay function to be re-calibrated, and is unlikely to ever achieve a precise flashing rate, while Method 2 allows the controlling program to do other processing without affecting the LED flashing rate.

---

[7] Performing the short calculation consumes a small amount of time. Repeating the calculation consumes multiples of that small amount of time.

State Machines

FLIP-FLOPS ALLOW US TO STORE A "STATE", encoded as one or more bits. The counters that we looked at earlier are a simple example of this idea. In a counter, the state is the current value of the count, and the next state is just the next value in the count. But we can generalize this concept to arbitrary states (i.e., representing something other than a count) and arbitrary state transitions (i.e., not just incrementing the state) that can depend on both the current state and on input values. This generalized idea of states and state transitions is known as a *Finite State Machine*[8] (FSM).

An FSM consists of:

[8] In Computer Science, the term Finite State Automaton (FSA) is often used.

- A set of *states* representing whatever it is that you might want to remember about a system. It's common to specify an *initial state* as well.
- A set of *inputs*.
- A set of *transitions* that define how to get a *next state* based on the *current state* and the *input*.
- A set of *outputs*, and some definition of when those outputs occur (either associated with a state or with a transition).

State machines are often depicted as *state transition diagrams* (often simply 'state diagram'), like the example in Fig. 6 which shows the state transitions for a 2-button stopwatch. The diagram shows the states of the system (the bubbles), and the new state the system will move to when a particular input occurs (the transition arrow, labeled with the input and pointing at the new state)[9]. The initial state and final state (if any) in the state diagram are represented using special circular symbols.

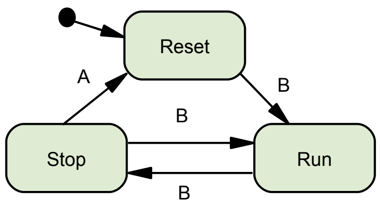

Figure 6: Example of a state machine

[9] It's sometimes implicit that an input with no outgoing transition from a state results in a "self-loop" transition, which means the system stays in the same state.

Example: One advantage of drawing a state transition diagram is that it lets us ask questions about the state machine. For example, Fig. 6 doesn't seem to allow a return to the *Reset* state to occur from the *Run* state. Does that make sense? Whether it does or not may depend on other design considerations. But by drawing the diagram we can communicate the design to others, and prompt these kinds of questions.

Example: A state transition diagram for a garage door opener is shown in Fig. 7. The transitions from state-to-state occur *when a clock edge occurs*, if the stated condition is met.

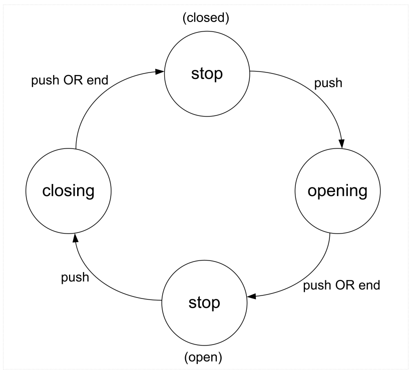

Figure 7: A state (transition) diagram for a garage door opener.

We can implement an FSM by using flip-flops to store state, and using combinational logic to determine the next state and generate outputs. The number of possible states the FSM can be in depends on the number of flip-flops. A collection N flip-flops, each able to store a single bit, can represent 2^N different binary sequences,

The choice of how to encode states as binary numbers is often influenced by the amount of combinational logic that would be required to define the transition from one state to the next. In some cases, designers will use more than the minimum number of flip-flops in order to gain flexibility in assigning codes to states.

which places an upper bound on the number of states that can be stored.

Example: If a drone control system has an FSM that keeps track of *three* possible states (fully functional, one component down, system failure), then how many flip-flops will the FSM require to store the state?

A *single flip-flop* can only represent $2^1 = 2$ possible values (0 or 1), so that won't be adequate to encode 3 states.

Two flip-flops can represent $2^2 = 4$ possible values (00, 01, 10, or 11), which is more than enough. In fact, since we have 4 values but only 3 states, we can choose which of the 3 values to use to encode the states. Which 3 values we choose may depend on other design considerations.

You may recognize this example from the first set of tutorial questions.

Fig. 8 and Fig. 9 show two slightly different ways to use flip-flops and combinational logic to build an FSM that updates its state on each clock cycle, based on the inputs and current state.

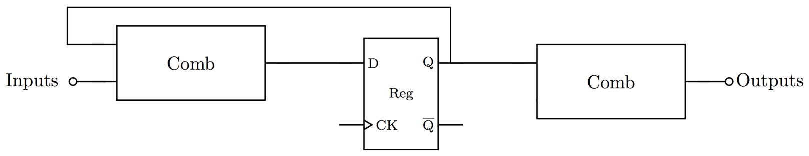

Figure 8: Moore machine

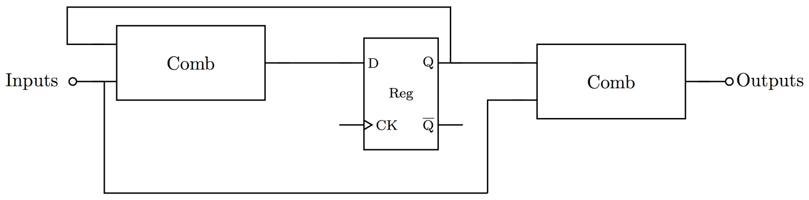

Figure 9: Mealy machine

The difference in form of the two state machine models in Fig. 8 and Fig. 9 is subtle, but important in terms of speed of operation and stability:

- For the *Moore* FSM, the outputs at any time are entirely determined *only* by the state
- For the *Mealy* FSM, the outputs are determined by the state *and* the current inputs.

Producing outputs based on both input *and* state can, in some cases, reduce the number of states required to obtain the desired behaviour. However, if the outputs are allowed to transition whenever the inputs change then the outputs may no longer be synchronized with the clock.

FSMs are very common in custom digital logic. More importantly, the heart of a microprocessor is essentially just an FSM. We'll look at FSM design in more detail in later lectures.

Aside: There's a good case to be made that all "computation" can be described in terms of state machines. See `https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Computation-and-State-Machines.pdf`

Interestingly, all computer programs can also be expressed as FSMs – although doing so can be very time consuming and difficult, hence high-level languages like "C" or Python are more commonly used describe what the FSM should do. The compiler or interpreter translates the high-level description into something closer to an FSM.

Looking Deeper: State Machines and Behaviour The classical way to describe *dynamic systems*[10], at least since the time of Newton, is to use one or more differential equations. Maybe you remember differential equations from your calculus classes. Or maybe you've suppressed the memory. Anyway, the basic concept of a differential equation is to describe the *rate of change* of one or more quantities as a function of the current value of those quantities. Your suppressed memories of calculus class may include equations like:

$$\frac{dy}{dt} = 0.5y$$

The differential equation is a concept that works great for the kind of systems Newton was interested in (continuously variable quantities in physical systems), but isn't quite so meaningful for describing computational behaviours (which mostly involve discretely varying inputs and outputs). And yet...

Differential equations are really just another kind of (synchronous) state machine:

- The value of y is the current state
- The infinitesimal interval of time dt is something like the clock signal that triggers a state transition
- The "next state" is equal to $y + (0.5ydt)$

---

[10] That is, systems that have *behaviour*.