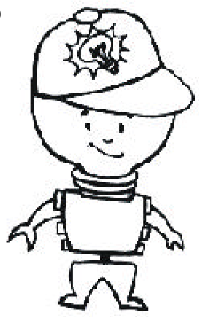
Digital Electronics & Chip Design



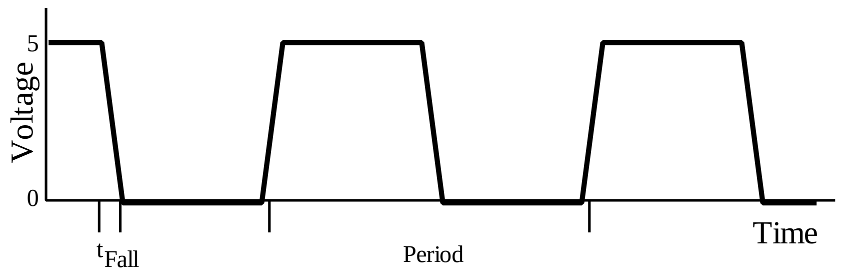
Lecture Notes V

In order to build sophisticated digital circuits, it is a good idea to have an established design methodology. Just as we restricted ourselves to only digital circuits from the field of all possible circuits at the beginning of this class, we may find it useful to restrict ourselves further to building a special type of digital circuit called a finite state machine (FSM). We will examine how clocks are generated to sequence digital logic through computations. Then we will learn what FSMs are and why they are so powerful for certain applications. Then we will study the shift register, the last of the 74xx series logic covered in this course.

Clocking

Clocking is both one of the most powerful and most subtle aspects of digital electronics. So far, we have made great use of its power in constructing flip-flops and counters. Indeed, virtually every digital electronics project of interesting nature requires clocking. However, clocking of complex circuits often requires careful consideration of the timing of the logic to prevent insidious errors. Fortunately, for this class, we will use clocks slow enough to avoid these problems. While most of the chips we use can easily run at 30 MHz (30 million cycles per second), breadboards contain sufficient capacitance and inductance to wreck havoc with most breadboarded circuits that run at greater than about 4 MHz.

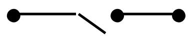
But the question remains: how does one generate a clock signal? To answer this, let us first consider the properties of a typical clock signal, as shown below:



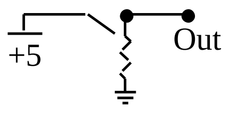
An ideal clock signal would be a square wave, alternating between a low voltage and high voltage (typically 0 and 5 volts for digital electronics applications). The length of time from positive edge to positive edge is called the period. The frequency is 1/period. For instance, if a wave had a period of 0.5 ms, it would have a frequency of 2 KHz. Unfortunately, the real world rears its ugly head; due to effects such as capacitance, it is impossible to ever produce an ideal square wave. Instead, there is a finite rise and fall time. When operating at extremely high clock frequencies, this can be a significant part of the total period, but we will be working at speeds low enough that we can ignore these secondary effects.

There are a multitude of methods to generate clocks. We will use two in this class. The easiest method is simply by toggling a switch or pressing a pushbutton manually. One can switch from high to low and back to high again. The low to high transition is the rising edge on which many devices trigger. However, this method has a serious disadvantage called switch bounce, which we will discuss next. The more sophisticated method of generating clock signals is with an oscillating circuit. Chips like the 555 timer, together with a resistor and capacitor, can be used to generate square waves with frequencies approximately equal to the product of the resistance and capacitance. By choosing appropriate values, we can produce clocks of frequencies from extremely slow (10^{-2} Hz, or slower) to far faster than human response (tens or hundreds of KHz). This method tends to break down at high frequencies due to stray capacitance and resistance, so for precise high-frequency circuits, one typically uses circuits built from tuned quartz crystals.

Switch bounce is the bane of manual clocking with cheap switches. A schematic of a single-pole single-throw (SPST) switch is shown below:

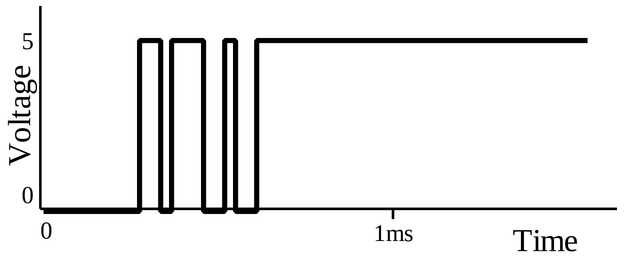


When the switch is open, there is no connection between the two ends. When the switch is closed, a piece of metal connects the two ends and allows current to flow. To produce five volts when the switch is closed and 0 volts when it is open, we can build a circuit as shown below:



When the switch is open, the resistor pulls the output down to ground. When the switch closes, the direct connection to 5 volts overrides the resistor and brings the output to 5 volts.

Unfortunately, when one makes contact, the switch does not simply close and stay closed. Instead, the metal strip in the switch tends to bounce a bit, perhaps repeatedly opening and closing several times over a period on the order of milliseconds. Instead of getting a single clock edge, one gets many short pulses, perhaps resembling that shown below shown below:

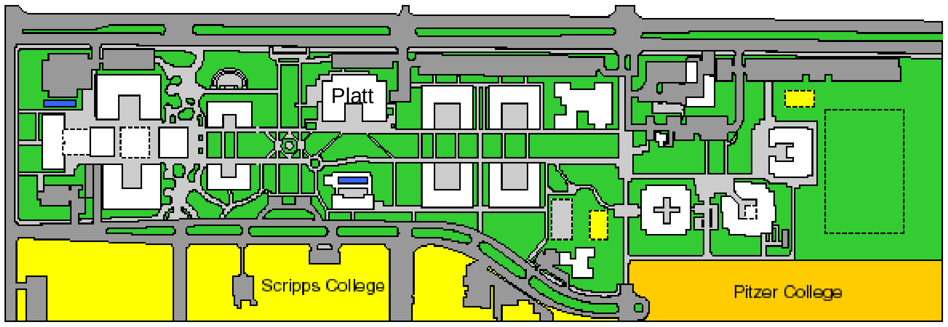


If the clock were simply controlling a D flip-flop with the D input manually connected to 0 or 5 volts, the multiple rising edges would cause no harm. However, if the clock were controlling a counter, the sequence of four edges would cause the counter to increment by four, a clearly undesirable effect.

Sometimes we are lucky enough to get switches that do not severely bounce; however, it is a bad idea to rely on a SPST switch in circumstances where bounce is a problem. There are many ways around this problem. In this class, using a clock signal generated from a 555 timer is probably easiest. One could use single pole double throw (SPDT) switches that physically connect the output to power on one side and ground on the other without the resistor. More elaborate tricks involve using timers or RC circuits and "Schmidtt triggers;" generally, however, using an electronic oscillator is easiest.

The Traffic Light Controller Problem

A concrete example might be a good lead-in to our discussion of finite state machines. During lunch time, there has been excessive congestion in front of Platt Dining Hall. Students headed north and south bound in front of Platt have nearly collided with unicyclists headed east or west reading books or juggling swords. No serious injuries have yet occurred, but parents have been calling the President demanding action. He has directed Physical Plant to install a system of traffic lights at the intersection controlling east/west traffic and north/south traffic.[1] Lights A control east/west traffic while lights B control north/south traffic.

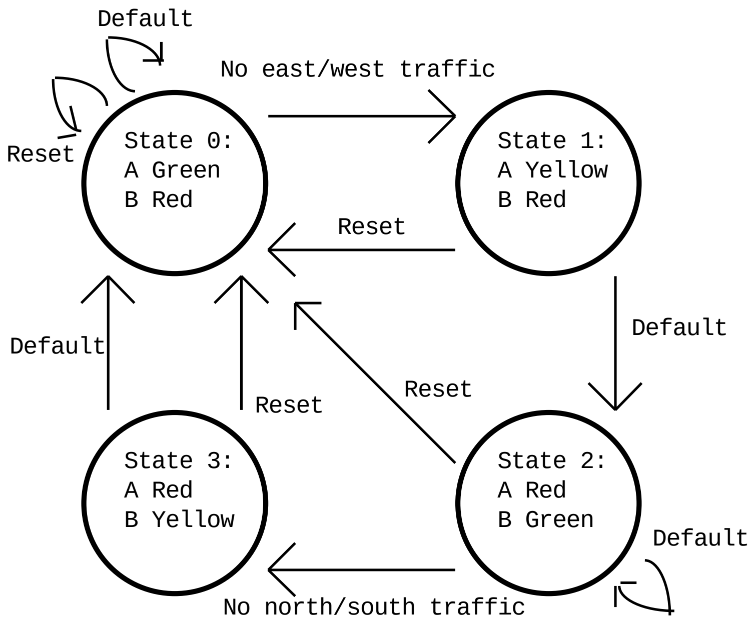


[1]Unfortunately, this system is doomed to fail because nobody in Southern California obeys traffic lights.

The traffic controller checks the state of traffic every 5 seconds and obeys the following rules:

> When reset, set light A green and light B red.
> If A was green:
> > If there is east/west traffic, A remains green.
> > If there is not east/west traffic, A turns yellow.
>
> If A was yellow, A turns red and B turns green.
> If B was green:
> > If there is north/south traffic, B remains green.
> > If there is not north/south traffic, B turns yellow.
>
> If B was yellow, B turns red and A turns green.

Let us design a controller for this light system. A good first step is to draw a "state transition diagram" showing the various states that the system might be in, what to do in each state, and when to change states. We use circles to indicate states and lines with arrows to represent transitions. The following picture shows one possible state transition diagram:



From this picture, we see that the system has four states, three inputs (Reset, east/west traffic, north/south traffic), and two outputs (light A and light B). Let us call the state S, the inputs R, E, and N, respectively, and the outputs A and B. Now, we can write a table to summarize these state transitions. An x in the table means "Don't care."

State	**Input**			**New State**	**Output**	
	R	**E**	**N**		**A**	**B**
x	Yes	x	x	0	Green	Red
0	No	Yes	x	0	Green	Red
0	No	No	x	1	Orange	Red

1	No	x	x	2	Red	Green
2	No	x	Yes	2	Red	Green
2	No	x	No	3	Orange	Green
3	No	x	x	0	Green	Red

In order to build this controller from digital electronics, we must translate these numbers and words into binary codes. A reasonable coding would be as follows:

State:	0 = 00, 1 = 01, 2 = 10, 3 = 11
Inputs:	No = 0, Yes = 1
Outputs:	Green = 00, Yellow = 01, Red = 11

With this coding, we require three bits of input (R, E, and N), two bits of state (for 0-3), and four bits of output (two bits each for A and B). Let us rewrite the previous table using these encodings (S_0' and S_1' represent the new state):

S_1	S_0	R	E	N	S_1'	S_0'	A_1	A_0	B_1	B_0
x	x	1	x	x	0	0	0	0	1	1
0	0	0	1	x	0	0	0	0	1	1
0	0	0	0	x	0	1	0	1	1	1
0	1	0	x	x	1	0	1	1	0	0
1	0	0	x	1	1	0	1	1	0	0
1	0	0	x	0	1	1	1	1	0	1
1	1	0	x	x	0	0	0	0	1	1

Now we have reduced our problem to a truth table. Perhaps the easiest way to implement the circuit is to write each of the new state and output variables as Boolean functions of the inputs and old state.

$$S_1' = \overline{S_1}\, S_0\overline{R} + S_1\overline{S_0}\,\overline{R}\, N + S_1\overline{S0}\,\overline{R}\,\overline{N}$$

$$= \overline{S_1}\, S_0\overline{R} + S_1\overline{S_0}\,\overline{R}$$

$$= \overline{R}\, [S_1 \text{ H } S_0]$$

$$S_0' = \overline{S_1}\,\overline{S_0}\,\overline{R}\,\overline{E} + S_1\overline{S_0}\,\overline{R}\,\overline{N}$$

$$= \overline{R}\, [\overline{S_1}\,\overline{S_0}\,\overline{E} + S_1\overline{S_0}\,\overline{N}\,]$$

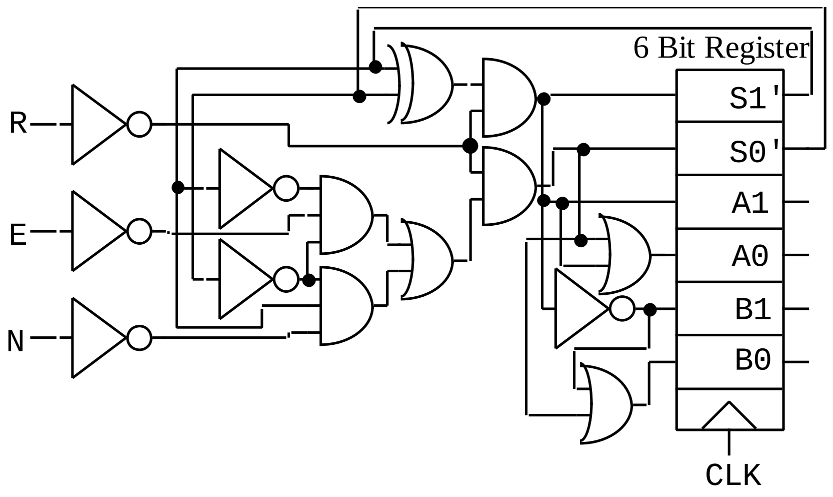
$$A_1 = S_1'$$

$$A_0 = S_0' + S_1'$$

$$B_1 = \overline{S_1}\ '$$

$B_0 \quad = B_1 + S_0'$

Notice that in some cases, we wrote variables in sum of products form and simplified if possible; in other cases, we were able to more easily express one output as a simple function of other outputs. These equations should allows us to easily produce a working circuit:



Our circuit is complete. Thanks to the diligent effort of Physical Plant, generations of future Harvey Mudd nerds will be able to safely cross in front of Platt!

Finite State Machines

The example we have just studied clearly makes use of some powerful ideas. While the problem is simple enough that a bright engineer could design a solution merely by inspection, the principles scale to yield a simple, methodical approach to solve arbitrarily large problems of this nature. Let us contemplate the steps we just took and try to refine them into a straightforward design methodology.

The circuit we just constructed is an example of a finite state machine. A finite state machine, in its most abstract form, is a device that accepts a set of inputs, maintains a current state, and, based on the inputs and the current state, produces outputs and a new state when it encounters a clock edge. This is a general enough capability to solve a wide range of problems, yet is restrictive enough that we can write a straightforward algorithm for designing such devices.

The steps of designing a FSM are as follows:

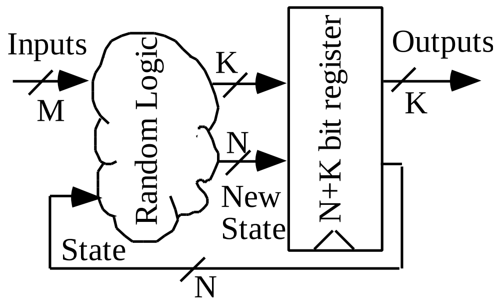
- State problem precisely
- Draw state transition diagram
- Determine number of inputs, states, and outputs
- Construct table listing outputs and new state in terms of inputs and old state
- Create binary coding for inputs, states, and outputs

- Rewrite table using binary coding
- Write Boolean equations for the table
- Implement equations using digital building blocks

The first step is to clearly state the problem without ambiguity. Given the problem description, we draw a state transition diagram with circles representing states, words describing the output in the given state, and lines representing transitions between states. From this diagram, we can construct a table describing the outputs and new state in terms of various input and old state conditions. The x is a powerful symbol indicating "don't care." Next, we translate our names and states into binary codes. If there are 2^N possible states, we need N bits to represent the current state and new state. With these codes, we can write a new table strictly using 1's, 0's, and x's. From this table, it is relatively easy to read off Boolean equations in sum of products form. Sometimes this will suffice; other times, it makes sense to try to optimize, attempting to minimize gate count or maximize speed. In the example above, we tried to minimize gate count to simplify the appearance of the solution and make it more legible.

Implementing the logic can be done in many ways. In our class, we will use the discrete logic gates on 74xx series chips. In other cases, we may have specialized components such as programmable logic arrays (PLAs) at our disposal which allow extremely simple implementation of sum of products functions; in these cases, we may find it possible to implement the entire FSM on a single chip. Virtually all real FSMs of interesting complexity are now constructed using various forms of programmable logic; however, due to the special hardware required, we will not study programmable logic any further in this class.[2]

No matter what technology is used, a general FSM with M bits of input, N bits of state, and K bits of output has the form shown below:



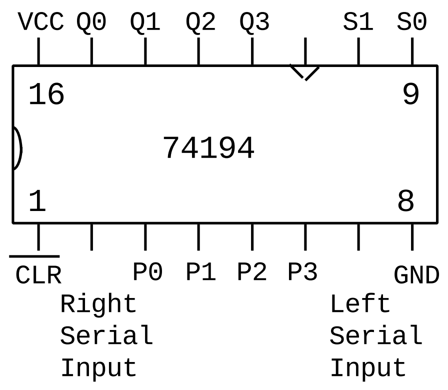
The FSM contains a section of combinational logic that computes the outputs and the new state as a function of the inputs and current state. On each clock edge, the result is latched by the register. The new state becomes the current state and new inputs are loaded into the FSM. Compare this abstract diagram to the schematic of the traffic control FSM we designed earlier.

[2] Anyhow, you need something to look forward to in E114 and E157!

Shift Registers

To wrap up our study of 74xx series logic components, we will examine the shift register. The shift register is an extension of a simple register, capable of shifting bits to the left or right as well as merely storing them. This is useful for extracting a single bit of information from a larger field, or for multiplying or dividing by 2 in binary.

The 74194, a typical shift register, is shown below. S1 and S0 are the two control bits. On each rising clock edge, the shift register performs an operation determined by the values of S1 and S0. The value stored in the register is output on the data lines Q0 through Q3. When both S1 and S0 are a logical 1, the shift register is loaded with the four parallel inputs P0 through P3. When S1 is 1 and S0 is 0, the register shifts its current contents to the left (Q3 toward Q0). Q0 is lost and Q3 is loaded from the Left Serial Input line. When S1 is 0 and S0 is 1, the register shifts its current contents to the right (Q0 toward Q3). Q3 is lost and Q0 is loaded from the Right Serial Input line. When both S0 and S1 are 0, the shift register does nothing on a clock edge.



Other 74xx Series Logic

Before departing from the realm of 74xx series logic, we will take a brief overview of other useful chips in the product line. There are a variety of arithmetic chips, including adders, comparators (that determine which of two inputs is greater in magnitude), and arithmetic logic units (ALUs). There are a number of multiplexor and demultiplexor chips which can choose one input or raise one output from a set of 2^N. There are various 4, 6, and 8 bit registers, some of which are designed to drive high-capacitance loads such as computer busses. And there are specialized circuits like parity generators and successive approximation registers that are dedicated to very specific tasks.

When designing your own projects, you may find the functional index in the front of most databooks a very useful reference to find what chips may already be available to solve your problem.