

CS61c: State Elements: Circuits That Remember

J. Wawrzynek

October 12, 2007

Reading: P&H, Appendix B

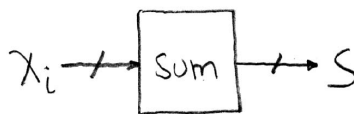
1 Introduction

Last time we saw that synchronous systems are composed of two basic types of circuits, combination logic circuits, and state elements. Combination logic circuits produce outputs based purely on their input signals. They are used for a wide variety of functions. State elements, on the other hand, have a small number of very specific uses. These are circuits that remember their input signal values. Last lecture we saw an example of state element, a register used to implement one of the general purpose registers for the MIPS.

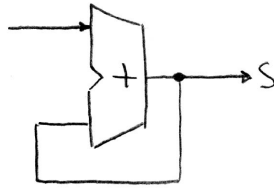
2 Accumulator Example

State elements are also used to **control the flow of signals** between combination logic circuits. This example should convince you that there are places where state elements are necessary for correct circuit function.

Consider the design of a circuit whose job is to form the sum of a list of integers, $X_0, X_1, X_2, \dots, X_{n-1}$. Assume that some other circuit applies the numbers from the list one at a time—one per clock cycle. Below is a graphical depiction of an abstraction of our circuit used to form the sum—we'll call it "sum". The X values are applied, one per cycle, and after n cycles the sum is present at the output, S .

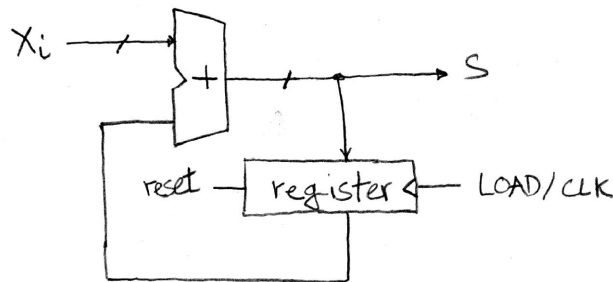

$$\begin{aligned} S &= 0; && \text{for } i \text{ from } 0 \text{ to } n-1 \\ S &= S + X_i; \end{aligned}$$

What should we put inside the sum block to achieve the desired function? Obviously the circuit involves an adder. Also, on each step we need to take the current sum and pass it back to the adder so that it can add another X value to it. Here's an idea on how to do this:



Let's examine the operation of this circuit in detail to see if it does the job. Assume that S begins at 0. We then apply X_0 . After a short delay (the adder propagation delay, τ_{add}) S will change to X_0 . Then after another τ_{add} of delay, S will change to $X_0 + X_0$, then after another τ_{add} of delay, S will change to $X_0 + X_0 + X_0$, etc. Because τ_{add} is typically less than the clock period, all of these adds of X_0 will happen before X_1 is applied. Obviously this is not correct operation. The circuit is out of control. We need some way to control the computation one step at a time.

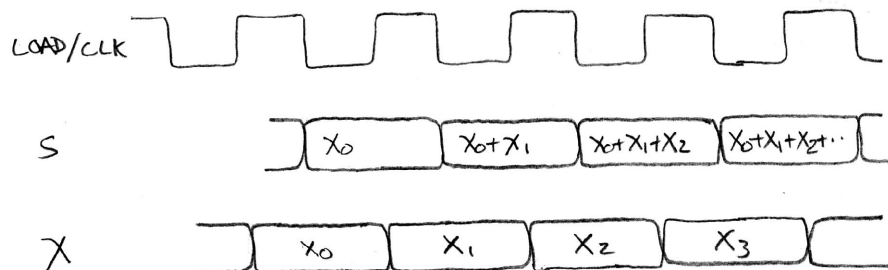
The way to control the computation is to put a register in the feedback path (the connection from the output of the adder to its input), as shown below.



The register holds the current value of S while the next one is being formed. After we are happy with the next S , we load it into the register and apply the new X value, then wait for a new S to appear at the output of the adder. The process is repeated for all n X values.

In the circuit diagram we used a another input signal to the register, labeled "reset". This is a signal that can be used to clear the register value, and thus gives us a way to initialize the circuit. Reset signals are a common feature of register circuits.

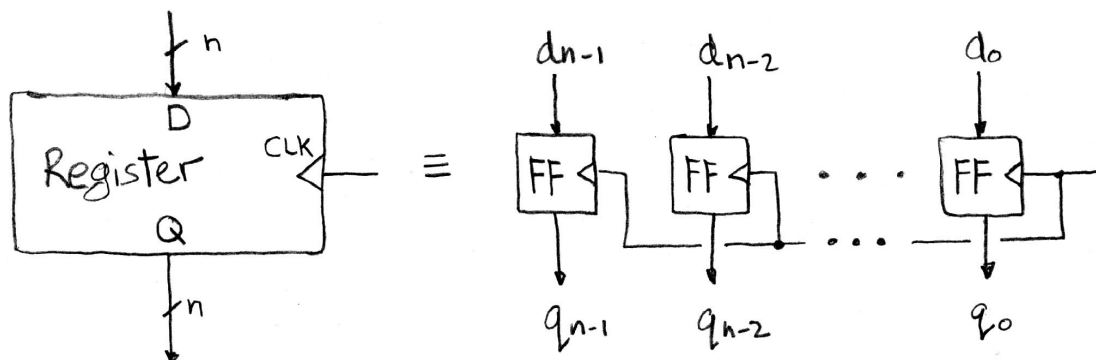
The waveforms demonstrating the operation of the accumulator circuit are shown below. Results are generated nicely one at a time. In this example, the register is used to hold up the transfer of data from one place to another in the circuit.



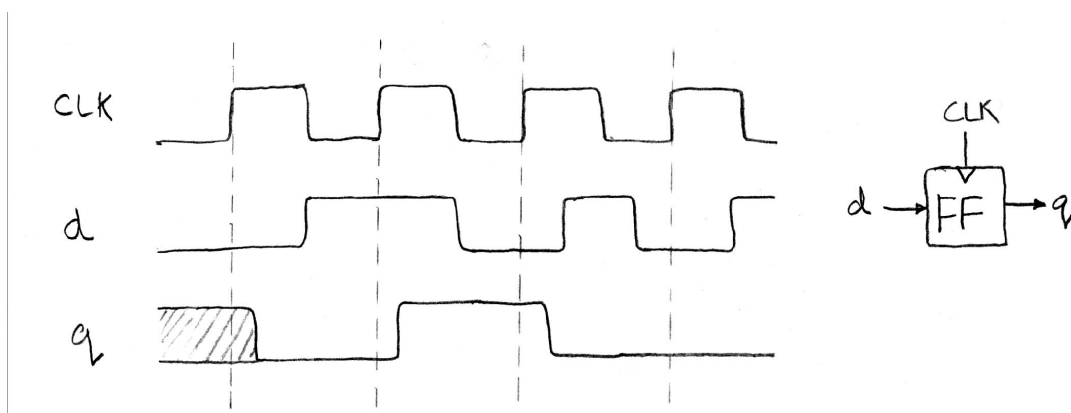
Later we will discuss the detailed timing of the operations in the accumulator circuit, but first we need to look at the detailed operation of the register.


3 Register Details: Flip-flops

What's inside a register? As illustrated in the figure below, an n -bit wide register is nothing other than **n instances of a simpler circuit**, called a *flip-flop* (FF). A flip-flop is 1-bit wide register. Its name comes from the fact that when in operation it flips (and flops) between holding a 0 or a 1. Notice that the CLK (or LOAD) signal is sent to the CLK (or LOAD) input of all n flip-flops, and that each is responsible for storing one bit of the n -bit data stored by the register. The convention for registers is that the input is label "D", or "d" in the case of a single bit, and the output is label "Q", or "q" for a single bit.



Here we are going to talk about the most common type of flip-flop, called an **edge-triggered d-type flip-flop**. Internally each flip-flop comprises around 10 transistors. The operation of the flip-flop is illustrated below. The illustration depicts the operation of one of two types of edge-triggered d-type flip-flops, a *positive edge-triggered* one (*negative edge-triggered* is the other). For the most part we will stick with positive edge flip-flops this semester.

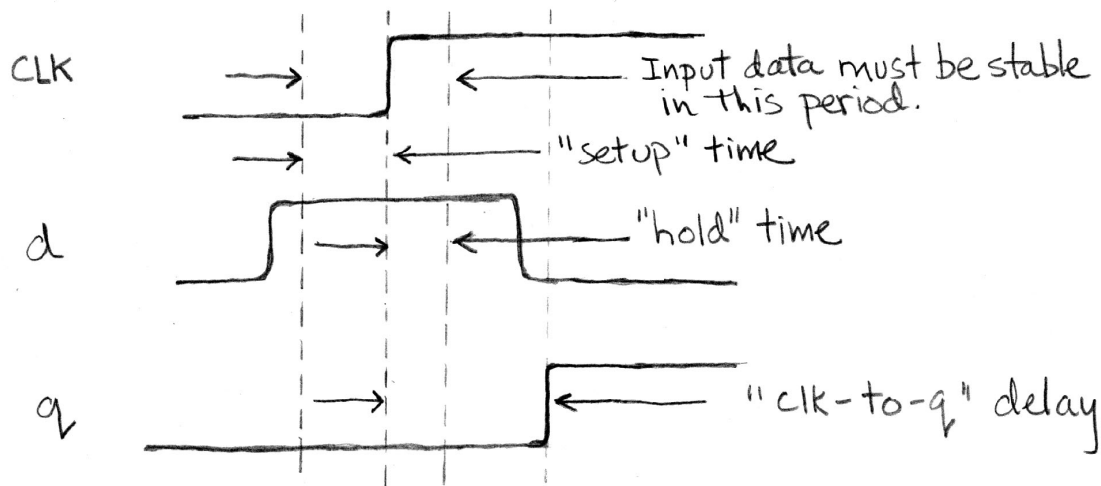


On each positive clock edge , the q output is changed to the current value of the d input.

**On the rising edge of the clock, the input d is sampled and transferred to the output.
At all other times, the input d is ignored.**

You should verify this for yourself by studying the waveforms. You should see that the **only time the output changes is right after the rising edge of the clock**. The input d can go up and down many times within a cycle, the only thing that is important, however, is its value right at the rising edge of the clock. You may also notice times when the flip-flop output does not change in response to the rising edge of the clock. This will happen only when the input d and the output q are already the same value.

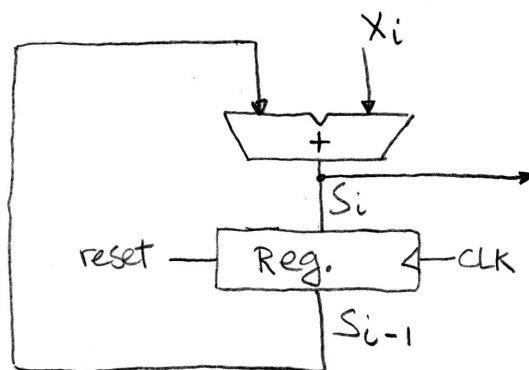
The detailed flip-flop timing is shown below. Like combinational logic circuits, flip-flops cannot change their outputs instantaneously. Also, **time is needed to transfer inputs internally**. Therefore, the input d must be stable for a short amount of time before the rising edge of the clock and remain stable for a short amount of time after the edge. The combination of these two (the former called the *setup time*, and the latter called the *hold time*) create a time window when the input d cannot change. If it changes in this window, the flip-flop will not reliably capture the new data input. Also, once the flip-flop captures the new data input in response to the clock edge, it takes a small amount of time to transfer the new value to the output. This delay is called the **clk-to-q delay**.



4 Accumulator Revisited

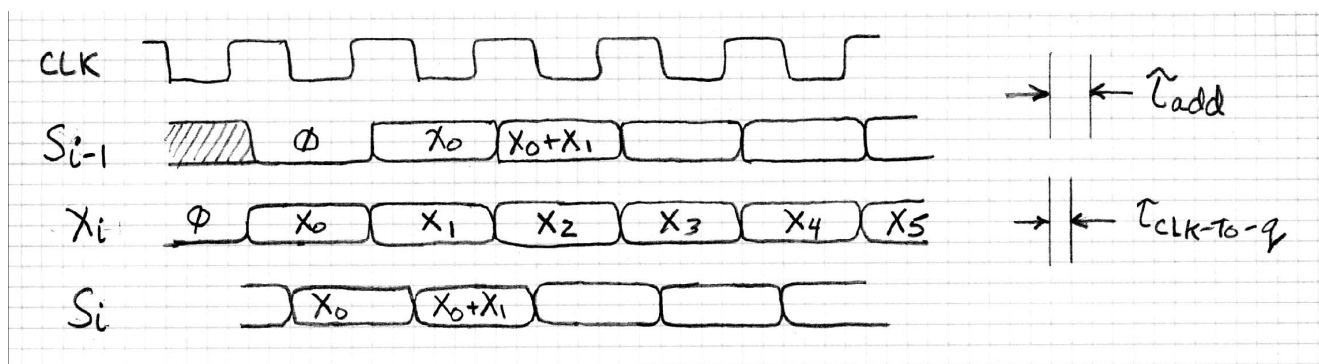
Now we can go back and look at the detailed timing of the waveforms associated with the accumulator example presented earlier. First a word about the register reset. We added a special input to the register called "reset" which is used to clear the register so that it holds all 0's. The most common type of reset input is called a **synchronous reset**. If the reset input signal has the value 1 on the rising edge of the clock signal, then the register is cleared, regardless of the value of the input D . In other words, the reset signal has priority over the data input and forces the register to all 0's.

Here again is the accumulator circuit.



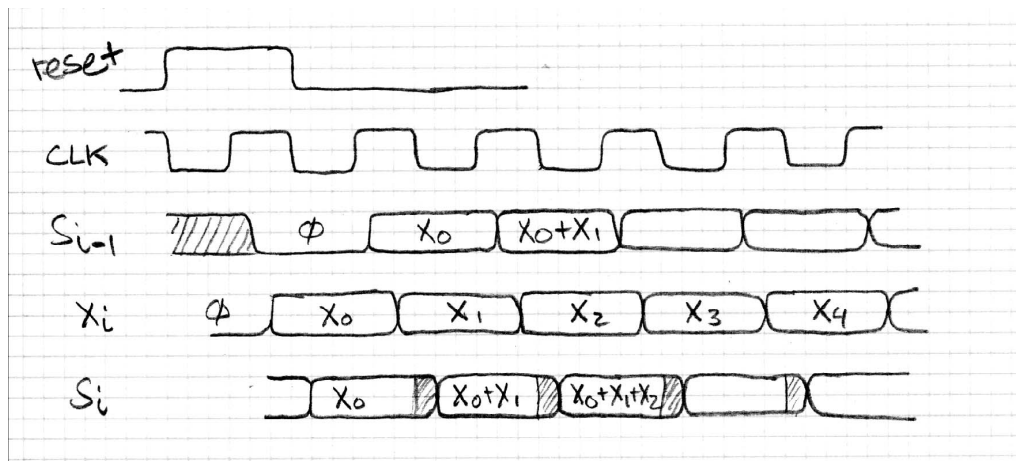
The output of the circuit is labeled S_i , and the output of the register is labeled S_{i-1} to remind us that the register delays the signal for 1 cycle. So if the output of the circuit is holding the result of the i^{th} iteration, then the register holds the result of the $i^{\text{th}} - 1$ iteration.

Below is the detailed waveforms for a few iterations.



Start by looking at the timing of the change on the output of the register S_{i-1} . This follows the positive-edge of the clock after a small delay (the clk-to-q time of the flip-flops used to implement the register). We assume that the input X is applied at precisely the same time. The two values move through the adder together and after a small delay (the adder propagation delay τ_{add}) a new result appears at the output of the adder, S_i . Then all is quiet until the rising edge of the clock. At that time the output value is transferred to the register and the whole process repeats.

In practice X may not necessarily arrive at the same time as the feedback value, S_{i-1} . The waveforms below show X arriving a little bit later than S_{i-1} .

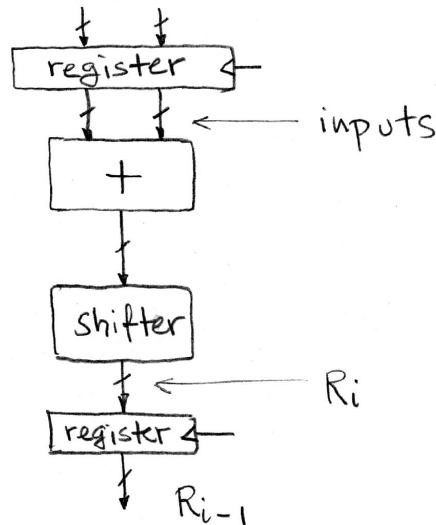


Therefore on each cycle there is a small time period where the adder has inconsistent inputs. For instance, when the register first captures X_0 , for a small time period the X input still has X_0 , therefore the adder begins to compute $X_0 + X_0$! However, this erroneous calculation is quickly aborted when the X input changes to X_1 . Unfortunately, the aborted computation will probably make it through the adder, creating a sort of instability at the output. However, the instability in S_i has no effect on S_{i-1} , as it captures its value from S_i before it goes bad. This sort of arrival mismatch and subsequent output instability is common in many circuits. In properly designed circuits, the instability never happens around the rising-edge of the clock and therefore gets ignored by the registers and down-stream circuitry.

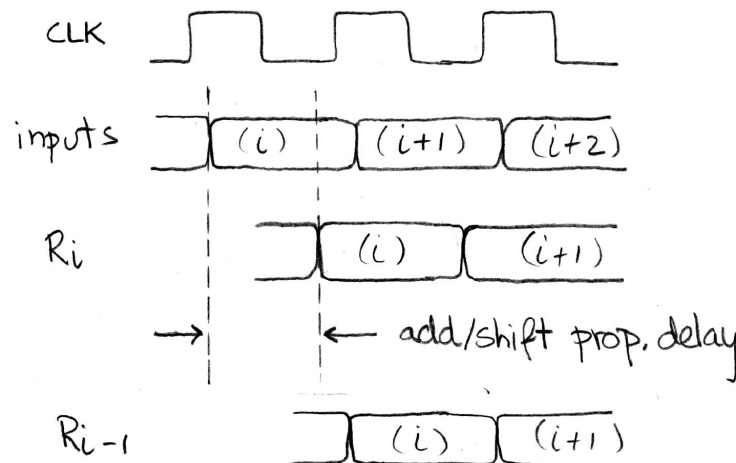
5 Pipelineing—Adding Registers to Improve Performance

In the previous section we saw an example of where a register was needed to ensure correct circuit behavior. In this section we shall see how registers can be used to increase the achievable clock frequency and thus improve the performance of the circuit.

Suppose we had the need to cascade two combinational logic circuits, **an adder and a shifter**. The idea of this circuit is that when input values arrive, they are added together, and then shifted by some amount (we won't worry about by how much right now). You could imagine that this circuit composition is part of a processor, perhaps in the floating-point unit. This circuit is shown below:

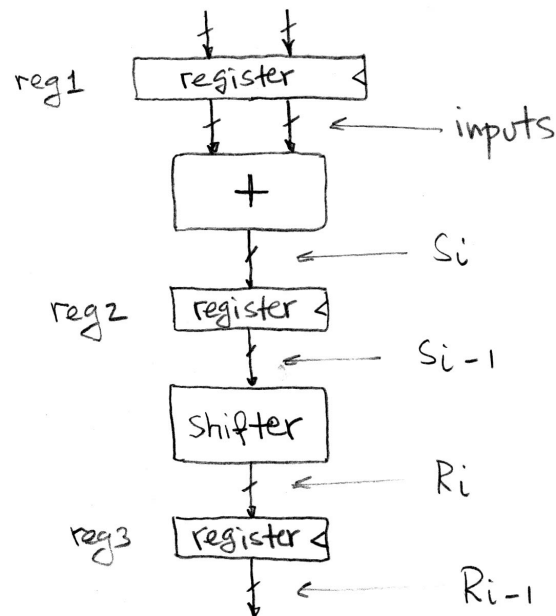


We assume that the input values come from a register (combined as one register in the drawing) and the output goes into a register. On each clock cycle we capture a new pair of input values in the input register and simultaneously capture the previous result in the output register. The waveforms showing the detailed timing of the operation of this circuit is shown below. Note that there is a delay of one clock cycle from input (output of the input register) to output (output of the output register, R_{i-1}).

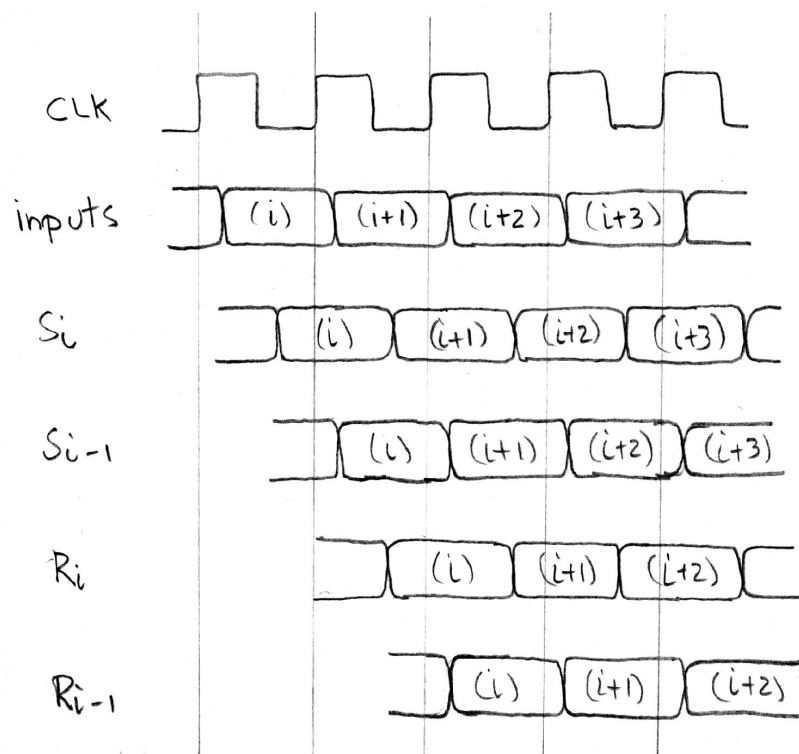


Notice that the maximum clock frequency (minimum clock period) is **limited by the propagation delay of the add/shift operation**. If we try to make the clock period too short, then the add/shift logic would not have sufficient time to generate its output and the output register would capture an incorrect value.

If we felt that the clock period for the correctly functioning circuit was too long we could choose to **split up the add/shift operation into two cycles**. We could do the add operation on the first cycle and the shift operation on the second cycle. This idea can be implemented by introducing a new register between the two blocks, as shown below:



On each cycle, data moves from the output of reg1 through the adder to the input of reg2, or moves from the output of reg2 through the shifter to the input of reg3. Since no longer does the data need to move through both blocks on each cycle, the clock frequency can be increased. See the waveforms below:



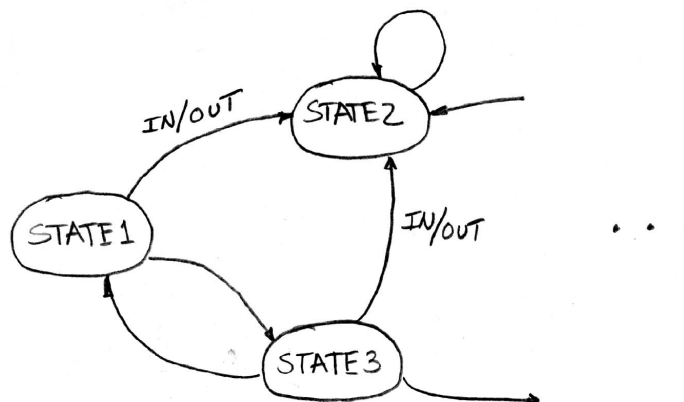
An interesting thing about this new circuit is that after the data moves through the adder and gets captured in reg2, on the next cycle when the data moves into the shifter, **new data can simultaneously move into the adder**, because it is free. Therefore new data values can be fed into the circuit on each clock cycle.

Of course, now there will be a two clock cycle delay from the insertion of a set of data into the circuit until when it appears at the output. But, the new clock period is shorter, so in absolute time, the delay from data insertion until output, is not really much worse. More importantly, because of the transformation, and the new higher clock rate, results will be **generated at a higher rate** (more outputs per second). If your figure of merit is results/time then this is a good transformation. If you are more interested in the latency (or delay) for any one set of numbers, then the transformation hurts a bit.

6 Finite State Machines

We assume that you have already been introduced to the theory of finite state machines (FSMs). These show up in many areas of computer science as they are a useful abstraction for many types of computing tasks. We use them often in hardware design. One example of where they would show up in a processor design would be in association with a hardware controlled cache. A FSM would be used to control the set of actions associated with servicing loads and stores from the processor. It would go through a series of “states” as it responds to requests from the processor, at times needing to access main memory and put new data into the cache. You can imagine describing the cache behavior as an algorithm or program. In hardware, that algorithm would be implemented with a FSM.

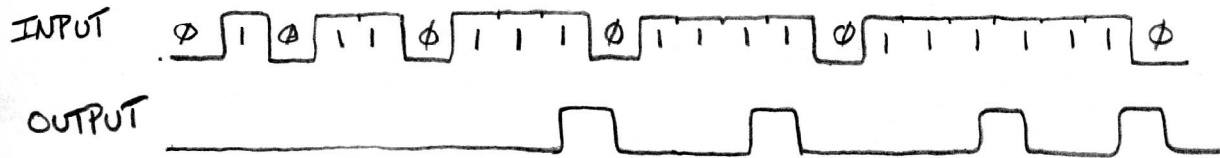
A finite state machine has a set of inputs, a set of outputs, and a finite collection of states. At any point in time, the machine is in one of the states. Each state can have one or more arcs that takes the machine to a new state, when certain input values appear. Self-loops are allowed. Output values are also associated with arcs. When the machine leaves a state it can change the output values. States are given unique names and, by convention, arcs are labeled with their associated input and output values.



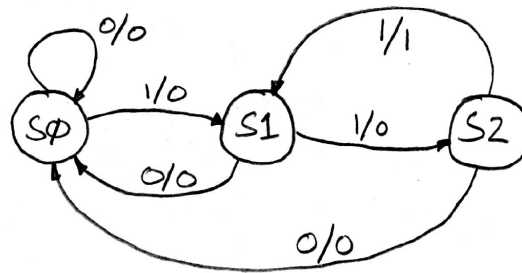
Given combinational logic circuits and registers, FSM, can be implemented in hardware.

As an example we will consider a simple circuit that is used to detect the occurrence of three 1's in a row in an input sequence of bits. Single bit values are applied one per cycle to the input of the FSM

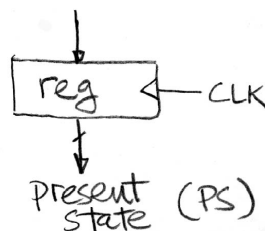
as shown below. On the cycle when the third of a string of three 1's appear at the input of the FSM, the output changes to a 1 for one cycle, then returns to 0 where it stays until another group of three.



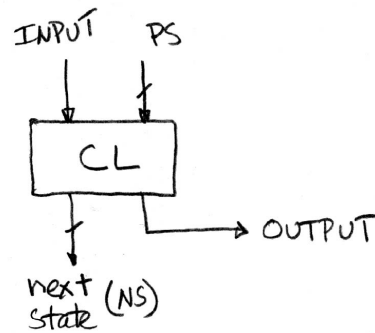
We begin the design of the FSM by drawing the state diagram. The S0 state is the state the machine is in when it begins counting 1s. S1 means the machine has seen one 1 and S2 means the machine has just seen the second 1. In S2 if the machine sees another 1 then the output is changed to a 1 and the machine goes back to S0. If in S2 the machine sees a 0, then the output remains at 0 and the machine returns back to S0.



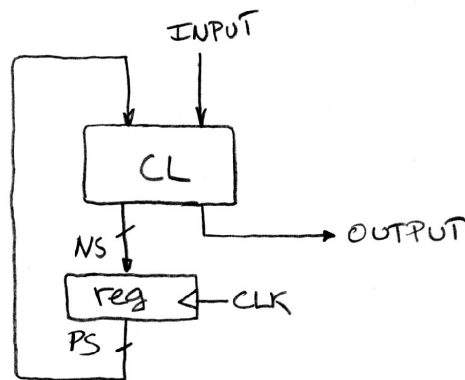
In the hardware implementation of a FSM, we assume that the state transitions are **controlled by the clock**; on each clock cycle the machine checks the inputs and transitions to a new state and produces a new output. Therefore we need a register to hold the current state of the machine. We will assign a unique bit pattern for each of the states in the state diagram. The register will keep track of which state the machine is currently in by holding the bit pattern corresponding to a particular state.



Now we need a circuit that implements a function that maps the input and present state to the next state and output. Because this circuit doesn't need to remember anything on its own, we can use a combinational logic circuit for this task.



Given that we want to force a state transition on each clock cycle (remember inputs bits are being applied one per clock cycle), we will put the combinational logic block in a loop with the state register as shown below:



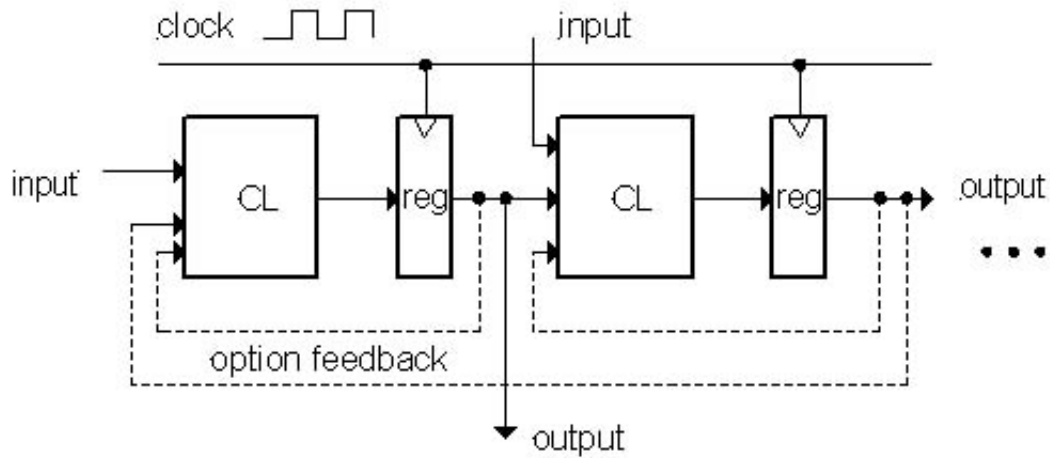
On each clock cycle the CL block looks at the input value and the present state and produces the bit pattern for the next state and generates an output of 0 or 1. On the rising edge of the clock the new state value gets captured by the register and the process starts again.

We didn't discuss the details of what's inside the CL block. We will discuss this in the next lecture, but for now you can consider the CL block as a function that maps its input to its output as shown in the table below. In this mapping we use the bit patterns 00, 01, and 10, for S0, S1, and S2, respectively. If the circuit sees the pattern on the left at its input, it produces the pattern on the right at its output.

| PS | INPUT | NS | OUTPUT |
|----|-------|----|--------|
| 00 | 0 | 00 | 0 |
| 00 | 1 | 01 | 0 |
| 01 | 0 | 00 | 0 |
| 01 | 1 | 10 | 0 |
| 10 | 0 | 00 | 0 |
| 10 | 1 | 00 | 1 |

7 A General Model for Synchronous Systems

By now you should have gotten the idea that any synchronous digital hardware system can be put together as a collection of combinational logic blocks separated by registers. This is true, and the illustration below is a good model of these systems.



However, usually we think of block of memory, such as the general purpose register file, the memory used to implement the caches, and main memory as special cases. We will consider memory blocks later.