

ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 3.1

Serialization and Finite State Machines

The third part of our class builds upon the basic combinational and sequential logic elements that we developed in the second part. After discussing a simple application of stored state to trade between area and performance, we introduce a powerful abstraction for formalizing and reasoning about digital systems, the Finite State Machine (FSM). General FSM models are broadly applicable in a range of engineering contexts, including not only hardware and software design but also the design of control systems and distributed systems. We limit our model so as to avoid circuit timing issues in your first exposure, but provide some amount of discussion as to how, when, and why you should eventually learn the more sophisticated models. Through development a range of FSM examples, we illustrate important design issues for these systems and motivate a couple of more advanced combinational logic devices that can be used as building blocks. Together with the idea of memory, another form of stored state, these elements form the basis for development of our first computer. At this point we return to the textbook, in which Chapters 4 and 5 provide a solid introduction to the von Neumann model of computing systems and the LC-3 (Little Computer, version 3) instruction set architecture. By the end of this part of the course, you will have seen an example of the boundary between hardware and software, and will be ready to write some instructions yourself.

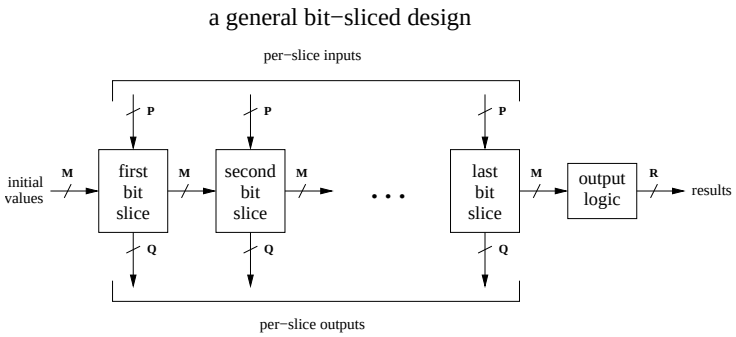
In this set of notes, we cover the first few parts of this material. We begin by describing the conversion of bit-sliced designs into serial designs, which store a single bit slice's output in flip-flops and then feed the outputs back into the bit slice in the next cycle. As a specific example, we use our bit-sliced comparator to discuss tradeoffs in area and performance. We introduce Finite State Machines and some of the tools used to design them, then develop a handful of simple counter designs. Before delving too deeply into FSM design issues, we spend a little time discussing other strategies for counter design and placing the material covered in our course in the broader context of digital system design. Remember that *sections marked with an asterisk are provided solely for your interest,* but you may need to learn this material in later classes.

Serialization: General Strategy

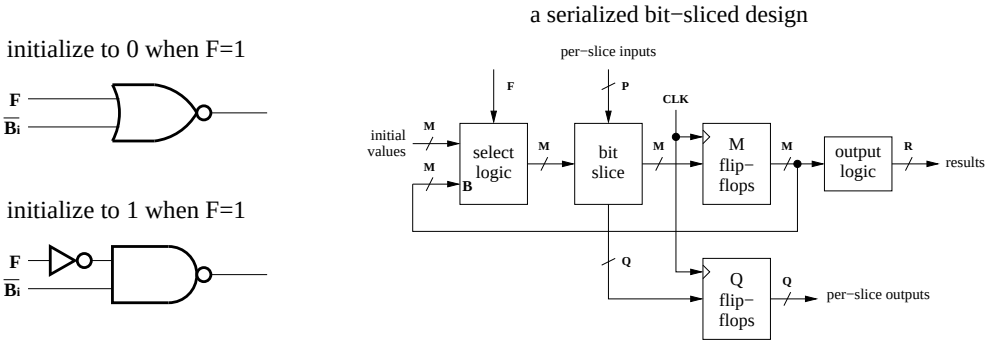
In previous notes, we discussed and illustrated the development of bit-sliced logic, in which one designs a logic block to handle one bit of a multi-bit operation, then replicates the bit slice logic to construct a design for the entire operation. We developed ripple carry adders in this way in Notes Set 2.3 and both unsigned and 2's complement comparators in Notes Set 2.4.

Another interesting design strategy is **serialization**: rather than replicating the bit slice, we can use flip-flops to store the bits passed from one bit slice to the next, then present the stored bits *to the same bit slice* in the next cycle. Thus, in a serial design, we only need one copy of the bit slice logic! The area needed for a serial design is usually much less than for a bit-sliced design, but such a design is also usually slower. After illustrating the general design strategy, we'll consider these tradeoffs more carefully in the context of a detailed example.

Recall the general bit-sliced design approach, as illustrated to the right. Some number of copies of the logic for a single bit slice are connected in sequence. Each bit slice accepts P bits of operand input and produces Q bits of external output. Adjacent bit slices receive an additional M bits of information from the previous bit slice and pass along M bits to the next bit slice, generally using some representation chosen by the designer. The first bit slice is initialized by passing in constant values, and some calculation may be performed on the final bit slice's results to produce R bits more external output.



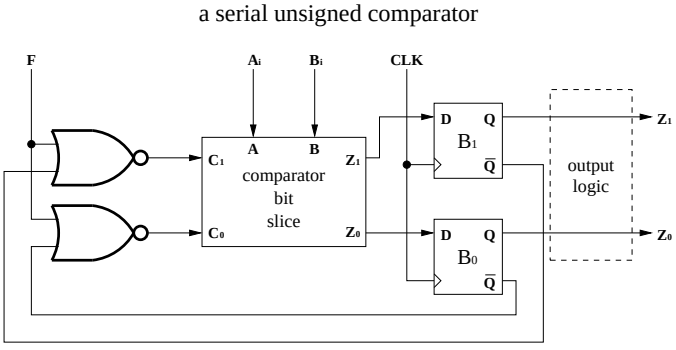
We can transform this bit-sliced design to a serial design with a single copy of the bit slice logic, $M + Q$ flip-flops, and M gates (and sometimes an inverter). The strategy is illustrated on the right below. A single copy of the bit slice operates on one set of P external input bits and produces one set of Q external output bits each clock cycle. In the design shown, these output bits are available during the next cycle, after they have been stored in the flip-flops. The M bits to be passed to the "next" bit slice are also stored in flip-flops, and in the next cycle are provided back to the same physical bit slice as inputs. The first cycle of a multi-cycle operation must be handled slightly differently, so we add selection logic and an control signal, F. For the first cycle, we apply $F = 1$, and the initial values are passed into the bit slice. For all other bits, we apply $F = 0$, and the values stored in the flip-flops are returned to the bit slice's inputs. After all bits have passed through the bit slice—after N cycles for an N-bit design—the final M bits are stored in the flip-flops, and the results are calculated by the output logic.



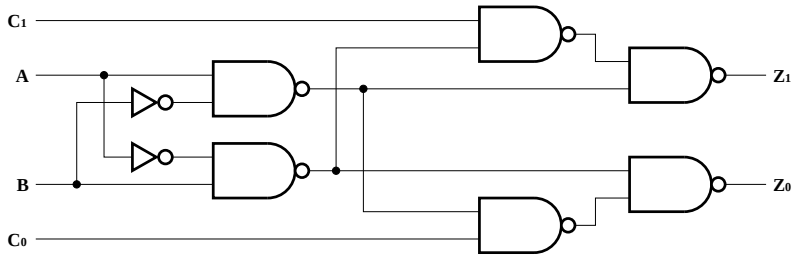
The selection logic merits explanation. Given that the original design initialized the bits to constant values (0s or 1s), we need only simple logic for selection. The two drawings on the left above illustrate how $\overline{B_i}$, the complemented flip-flop output for a bit i, can be combined with the first-cycle signal F to produce an appropriate input for the bit slice. Selection thus requires one extra gate for each of the M inputs, and we need an inverter for F if any of the initial values is 1.

Serialization: Comparator Example

We now apply the general strategy to a specific example, our bit-sliced unsigned comparator from Notes Set 2.4. The result is shown to the right. In terms of the general model, the single comparator bit slice accepts $P = 2$ bits of inputs each cycle, in this case a single bit from each of the two numbers being compared, presented to the bit slice in increasing order of significance. The bit slice produces no external output other than the final result ($Q = 0$). Two bits ($M = 2$) are produced each cycle by the bit slice and stored into flip flops B_1 and B_0. These bits represent the relationship between the two numbers compared so far (including only the bit already seen by the comparator bit slice). On the first cycle, when the least significant bits of A and B are being fed into the bit slice, we set $F = 1$, which forces the C_1 and C_0 inputs of the bit slice to 0 independent of the values stored in the flip-flops. In all other cycles, $F = 0$, and the NOR gates set $C_1 = B_1$ and $C_0 = B_0$. Finally, after N cycles for an N-bit comparison, the output logic—in this case simply wires, as shown in the dashed box—places the result of the comparison on the Z_1 and Z_0 outputs ($R = 2$ in the general model). The result is encoded in the representation defined for constructing the bit slice (see Notes Set 2.4, but the encoding does not matter here).



a comparator bit slice (optimized, NAND/NOR)



How does the serial design compare with the bit-sliced design? As an estimate of area, let's count gates. Our optimized design is replicated to the right for convenience. Each bit slice requires six 2-input gates and two inverters. Assume that each flip-flop requires eight 2-input gates and two inverters, so the serial design overall requires 24 gates and six inverters to handle any number of bits. Thus, for any number of bits $N \geq 4$, the serial design is smaller than the bit-sliced design, and the benefit grows with N.

What about performance? In Notes Set 2.4, we counted gate delays for our bit-sliced design. The path from A or B to the outputs is four gate delays, but the C to Z paths are all two gate delays. Overall, then, the bit-sliced design requires $2N + 2$ gate delays for N bits. What about the serial design?

The performance of the serial design is likely to be much worse for three reasons. First, all paths in the design matter, not just the paths from bit slice to bit slice. None of the inputs can be assumed to be available before the start of the clock cycle, so we must consider all paths from input to output. Second, we must also count gate delays for the selection logic as well as the gates embedded in the flip-flops. Finally, the result of these calculations may not matter, since the clock speed may well be limited by other logic elsewhere in the system. If we want a common clock for all of our logic, the clock must not go faster than the slowest element in the entire system, or some of our logic will not work properly.

What is the longest path through our serial comparator? Let's assume that the path through a flip-flop is eight gate delays, with four on each side of the clock's rising edge. The inputs A and B are likely to be driven by flip-flops elsewhere in our system, so we conservatively count four gate delays to A and B and five gate delays to C_1 and C_0 (the extra one comes from the selection logic). The A and B paths thus dominate inside the bit slice, adding four more gate delays to the outputs Z_1 and Z_0. Finally, we add the last four gate delays to flip the first latch in the flip-flops for a total of 12 gate delays. If we assume that our serial comparator limits the clock frequency (that is, if everything else in the system can use a faster clock), we take 12 gate delays per cycle, or $12N$ gate delays to compare two N-bit numbers.

You might also notice that adding support for 2's complement is no longer free. We need extra logic to swap the A and B inputs in the cycle corresponding to the sign bits of A and B. In other cycles, they must remain in the usual order. This extra logic is not complex, but adds further delay to the paths.

The bit-sliced and serial designs represent two extreme points in a broad space of design possibilities. Optimization of the entire N-bit logic function (for any metric) represents a third extreme. As an engineer, you should realize that you can design systems anywhere in between these points as well. At the end of Notes Set 2.4, for example, we showed a design for a logic slice that compares two bits at a time. In general, we can optimize logic for any number of bits and then apply multiple copies of the resulting logic in space (a generalization of the bit-sliced approach), or in time (a generalization of the serialization approach), or in a combination of the two. Sometimes these tradeoffs may happen at a higher level. As mentioned in Notes Set 2.3, computer software uses the carry out of an adder to perform addition of larger groups of bits (over multiple clock cycles) than is supported by the processor's adder hardware. In computer system design, engineers often design hardware elements that are general enough to support this kind of extension in software.

As a concrete example of the possible tradeoffs, consider a serial comparator design based on the 2-bit slice variant. This approach leads to a serial design with 24 gates and 10 inverters, which is not much larger than our earlier serial design. In terms of gate delays, however, the new design is identical, meaning that we finish a comparison in half the time. More realistic area and timing metrics show slightly more difference between the two designs. These differences can dominate the results if we blindly scale the idea to handle more bits without thinking carefully about the design. Neither many-input gates nor gates driving many outputs work well in practice.

Finite State Machines

A **finite state machine** (or **FSM**) is a model for understanding the behavior of a system by describing the system as occupying one of a finite set of states, moving between these states in response to external inputs, and producing external outputs. In any given state, a particular input may cause the FSM to move to another state; this combination is called a **transition rule**. An FSM comprises five parts: a finite set of states, a set of possible inputs, a set of possible outputs, a set of transition rules, and methods for calculating outputs.

When an FSM is implemented as a digital system, all states must be represented as patterns using a fixed number of bits, all inputs must be translated into bits, and all outputs must be translated into bits. For a digital FSM, transition rules must be **complete**; in other words, given any state of the FSM, and any pattern of input bits, a transition must be defined from that state to another state (transitions from a state to itself, called **self-loops**, are acceptable). And, of course, calculation of outputs for a digital FSM reduces to Boolean logic expressions. In this class, we focus on clocked synchronous FSM implementations, in which the FSM's internal state bits are stored in flip-flops.

In this section, we introduce the tools used to describe, develop, and analyze implementations of FSMs with digital logic. In the next few weeks, we will show you how an FSM can serve as the central control logic in a computer. At the same time, we will illustrate connections between FSMs and software and will make some connections with other areas of interest in ECE, such as the design and analysis of digital control systems.

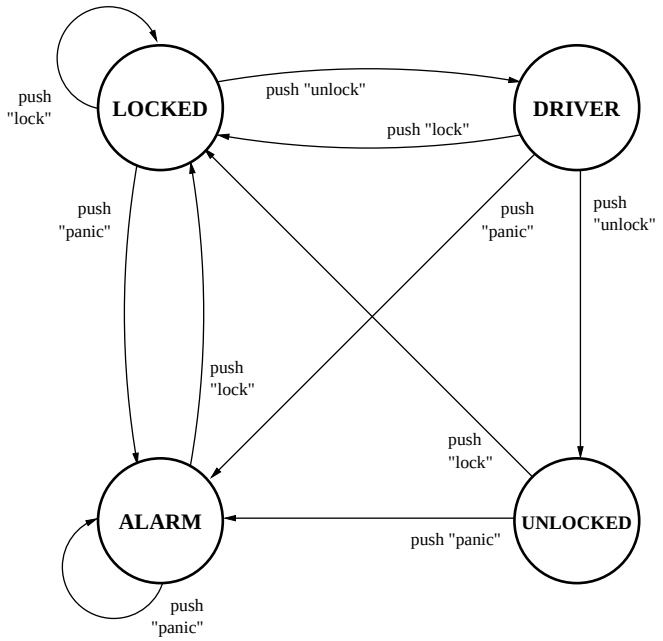
The table below gives a **list of abstract states** for a typical keyless entry system for a car. In this case, we have merely named the states rather than specifying the bit patterns to be used for each state—for this reason, we refer to them as abstract states. The description of the states in the first column is an optional element often included in the early design stages for an FSM, when identifying the states needed for the design. A list may also include the outputs for each state. Again, in the list below, we have specified these outputs abstractly. By including outputs for each state, we implicitly assume that outputs depend only on the state of the FSM. We discuss this assumption in more detail later in these notes (see "Machine Models"), but will make the assumption throughout our class.

meaning	state	driver's door	other doors	alarm on
vehicle locked	LOCKED	locked	locked	no
driver door unlocked	DRIVER	unlocked	locked	no
all doors unlocked	UNLOCKED	unlocked	unlocked	no
alarm sounding	ALARM	locked	locked	yes

Another tool used with FSMs is the **next-state table** (sometimes called a **state transition table**, or just a **state table**), which maps the current state and input combination into the next state of the FSM. The abstract variant shown below outlines desired behavior at a high level, and is often ambiguous, incomplete, and even inconsistent. For example, what happens if a user pushes two buttons? What happens if they push unlock while the alarm is sounding? These questions should eventually be considered. However, we can already start to see the intended use of the design: starting from a locked car, a user can push "unlock" once to gain entry to the driver's seat, or push "unlock" twice to open the car fully for passengers. To lock the car, a user can push the "lock" button at any time. And, if a user needs help, pressing the "panic" button sets off an alarm.

state	action/input	next state
LOCKED	push "unlock"	DRIVER
DRIVER	push "unlock"	UNLOCKED
(any)	push "lock"	LOCKED
(any)	push "panic"	ALARM

A **state transition diagram** (or **transition diagram**, or **state diagram**), as shown to the right, illustrates the contents of the next-state table graphically, with each state drawn in a circle, and arcs between states labeled with the input combinations that cause these transitions from one state to another.



Putting the FSM design into this graphical form does not solve the problems with the abstract model. The questions that we asked in regard to the next-state table remain unanswered.

Implementing an FSM using digital logic requires that we translate the design into bits, eliminate any ambiguity, and complete the specification. How many internal bits should we use? What are the possible input values, and how are their meanings represented in bits? What are the possible output values, and how are their meanings represented in bits? We will consider these questions for several examples in the coming weeks.

For now, we simply define answers for our example design, the keyless entry system. Given four states, we need at least $\lceil \log_2(4) \rceil = 2$ bits of internal state, which we store in two flip-flops and call S_1S_0. The table below lists input and output signals and defines their meaning.

outputs	D	driver door; 1 means unlocked
	R	other doors (Remaining doors); 1 means unlocked
	A	alarm; 1 means alarm is sounding
inputs	U	unlock button; 1 means it has been pressed
	L	lock button; 1 means it has been pressed
	P	panic button; 1 means it has been pressed

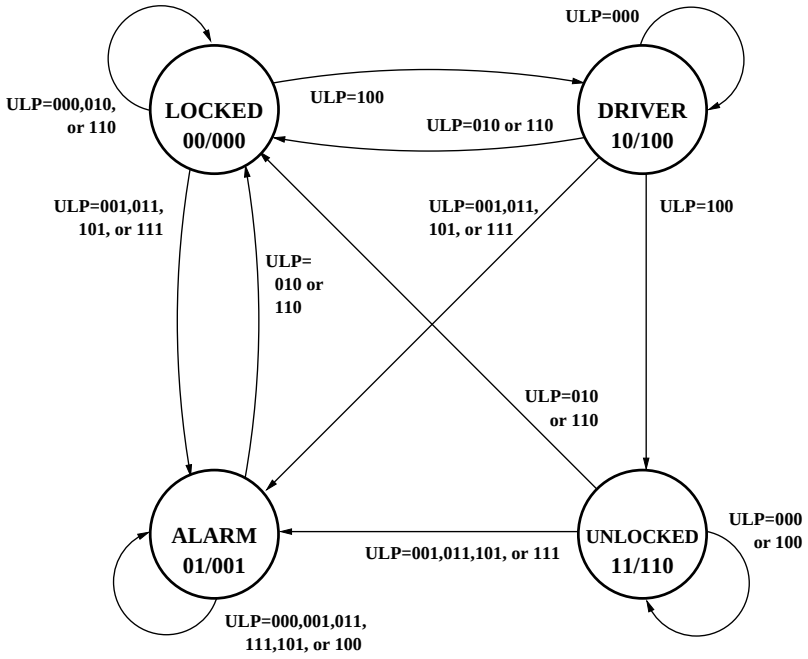
We can now choose a representation for our states and rewrite the list of states, using bits both for the states and for the outputs. We also include the meaning of each state for clarity in our example. Note that we can choose the internal representation in any way. Here we have matched the D and R outputs when possible to simplify the output logic needed for the implementation. The order of states in the list is not particularly important, but should be chosen for convenience and clarity (including transcribing bits into to K-maps, for example).

meaning	state	S_1S_0	driver's door D	other doors R	alarm on A
vehicle locked	LOCKED	00	0	0	0
driver door unlocked	DRIVER	10	1	0	0
all doors unlocked	UNLOCKED	11	1	1	0
alarm sounding	ALARM	01	0	0	1

We can also rewrite the next-state table in terms of bits. We use Gray code order on both axes, as these orders make it more convenient to use K-maps. The values represented in this table are the next FSM state given the current state S_1S_0 and the inputs U, L, and P. Our symbols for the next-state bits are S_1^+ and S_0^+. The "+" superscript is a common way of expressing the next value in a discrete series, here induced by the use of clocked synchronous logic in implementing the FSM. In other words, S_1^+ is the value of S_1 in the next clock cycle, and S_1^+ in an FSM implemented as a digital system is a Boolean expression based on the current state and the inputs. For our example problem, we want to be able to write down expressions for $S_1^+(S_1, S_0, U, L, P)$ and $S_1^+(S_1, S_0, U, L, P)$, as well as expressions for the output logic $U(S_1, S_0)$, $L(S_1, S_0)$, and $P(S_1, S_0)$.

current state	ULP							
S_1S_0	000	001	011	010	110	111	101	100
00	00	01	01	00	00	01	01	10
01	01	01	01	00	00	01	01	01
11	11	01	01	00	00	01	01	11
10	10	01	01	00	00	01	01	11

In the process of writing out the next-state table, we have made decisions for all of the questions that we asked earlier regarding the abstract state table. These decisions are also reflected in the complete state transition diagram shown to the right. The states have been extended with state bits and output bits, as S_1S_0/DRA. You should recognize that we can also leave some questions unanswered by placing x's (don't cares) into our table. However, you should also understand at this point that any implementation will produce bits, not x's, so we must be careful not to allow arbitrary choices unless any of the choices allowed is indeed acceptable for our FSM's purpose. We will discuss this process and the considerations necessary as we cover more FSM design examples.

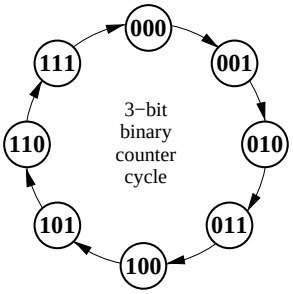


We have deliberately omitted calculation of expressions for the next-state variables S_1^+ and S_0^+, and for the outputs U, L, and P. We expect that you are able to do so from the detailed state table above, and may assign such an exercise as part of your homework.

Synchronous Counters

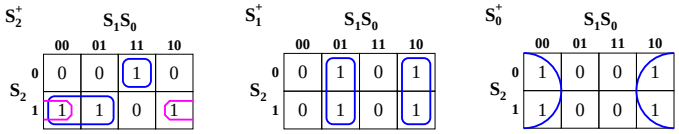
A **counter** is a clocked sequential circuit with a state diagram consisting of a single logical cycle. Not all counters are synchronous. In other words, not all flip-flops in a counter are required to use the same clock signal. A counter in which all flip-flops do utilize the same clock signal is called a **synchronous counter**. Except for a brief introduction to other types of counters in the next section, our class focuses entirely on clocked synchronous designs, including counters.

The design of synchronous counter circuits is a fairly straightforward exercise given the desired cycle of output patterns. The task can be more complex if the internal state bits are allowed to differ from the output bits, so for now we assume that output Z_i is equal to internal bit S_i. Note that distinction between internal states and outputs is necessary if any output pattern appears more than once in the desired cycle.



The cycle of states shown to the right corresponds to the states of a 3-bit binary counter. The numbers in the states represent both internal state bits $S_2S_1S_0$ and output bits $Z_2Z_1Z_0$. We transcribe this diagram into the next-state table shown on the left below. We then write out K-maps for the next state bits S_2^+, S_1^+, and S_0^+, as shown to the right, and use the K-maps to find expressions for these variables in terms of the current state.

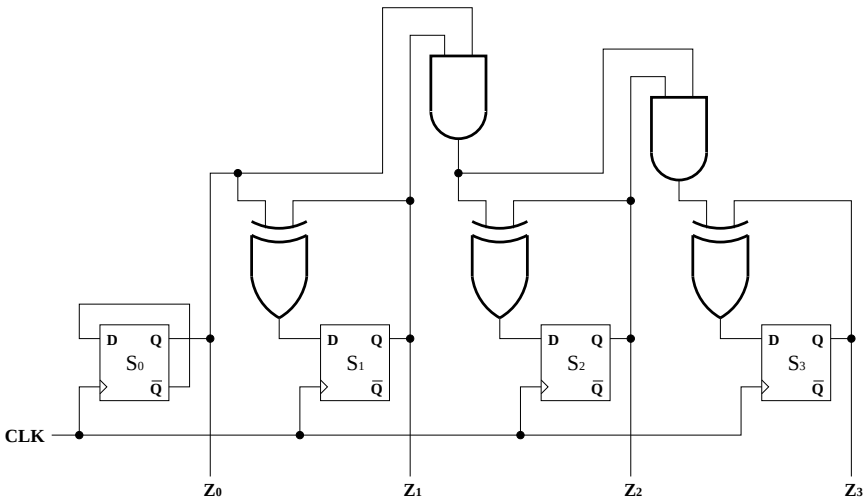
S_2	S_1	S_0	S_2^+	S_1^+	S_0^+
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0



$$S_2^+ = \bar{S}_2S_1S_0 + S_2\bar{S}_1 + S_2\bar{S}_0 = S_2 \oplus (S_1S_0)$$
$$S_1^+ = S_1\bar{S}_0 + \bar{S}_1S_0 = S_1 \oplus S_0$$
$$S_0^+ = \bar{S}_0 = S_0 \oplus 1$$

The first form of the expression for each next-state variable is taken directly from the corresponding K-map. We have rewritten each expression to make the emerging pattern more obvious. We can also derive the pattern intuitively by asking the following: given a binary counter in state $S_{N-1}S_{N-2}\ldots S_{j+1}S_jS_{j-1}\ldots S_1S_0$, when does S_j change in the subsequent state? The answer, of course, is that S_j changes when all of the bits below S_j are 1. Otherwise, S_j remains the same in the next state. We thus write $S_j^+ = S_j \oplus (S_{j-1}\ldots S_1S_0)$ and implement the counter as shown below for a 4-bit design. Note that the usual order of output bits along the bottom is reversed in the figure, with the most significant bit at the right rather than the left.

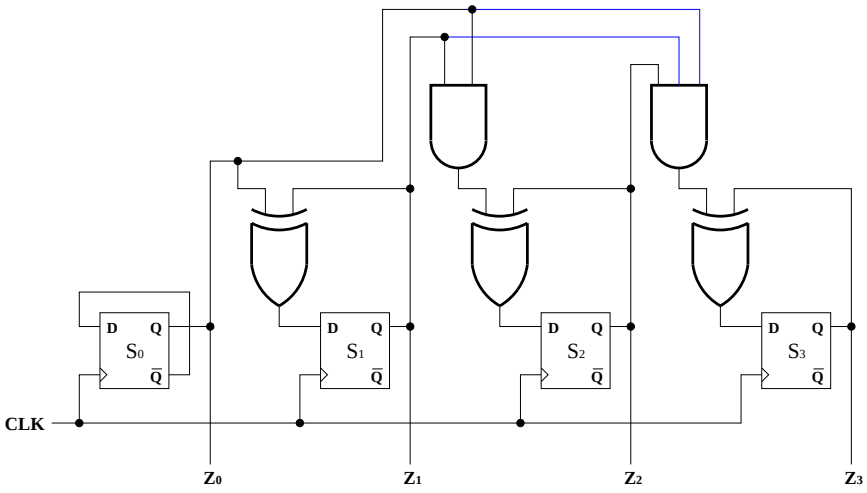
a 4−bit synchronous binary counter with serial gating



The calculation of the left inputs to the XOR gates in the counter shown above is performed with a series of two-input AND gates. Each of these gates AND's another flip-flop value into the product. This approach, called **serial gating**, implies that an N-bit counter requires more than $N-2$ gate delays to settle into the next state. An alternative approach, called **parallel gating**, calculates each input independently with a

single logic gate, as shown below. The blue inputs to the AND gate for S_3 highlight the difference from the previous figure (note that the two approaches differ only for bits S_3 and above). With parallel gating, the **fan-in** of the gates (the number of inputs) and the **fan-out** of the flip-flop outputs (number of other gates into which an output feeds) grow with the size of the counter. In practice, large counters use a combination of these two approaches.

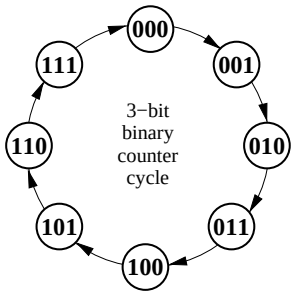
a 4−bit synchronous binary counter with parallel gating



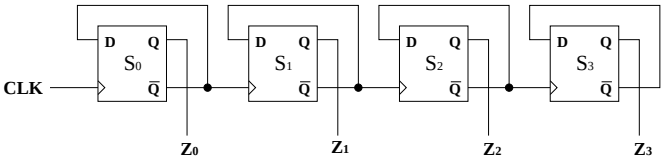
Ripple Counters

A second class of counter drives some of its flip-flops with a clock signal and feeds flip-flop outputs into the clock inputs of its remaining flip-flops, possibly through additional logic. Such a counter is called a **ripple counter**, because the effect of a clock edge ripples through the flip-flops. The delay inherent to the ripple effect, along with the complexity of ensuring that timing issues do not render the design unreliable, are the major drawbacks of ripple counters. Compared with synchronous counters, however, ripple counters consume less energy, and are sometimes used for devices with restricted energy supplies.

General ripple counters can be tricky because of timing issues, but certain types are easy. Consider the design of binary ripple counter. The state diagram for a 3-bit binary counter is replicated to the right. Looking at the states, notice that the least-significant bit alternates with each state, while higher bits flip whenever the next smaller bit (to the right) transitions from one to zero. To take advantage of these properties, we use positive edge-triggered D flip-flops with their complemented ($\bar{Q}$) outputs wired back to their inputs. The clock input is fed only into the first flip-flop, and the complemented output of each flip-flop is also connected to the clock of the next.



a 4−bit binary ripple counter



An implementation of a 4-bit binary ripple counter appears to the right. The order of bits in the figure matches the order used for our synchronous binary counters: least significant on the left, most significant on the right. As you can see from the figure, the technique generalizes to arbitrarily large binary ripple counters, but the time required for the outputs to settle after a clock edge scales with the number of flip-flops in the counter. On the other hand, an average of only two flip-flops see each clock edge ($1 + 1/2 + 1/4 + \ldots$), which reduces the power requirements.[1]

[1] Recall that flip-flops record the clock state internally. The logical activity required to record such state consumes energy.

Beginning with the state 0000, at the rising clock edge, the left (S_0) flip-flop toggles to 1. The second (S_1) flip-flop sees this change as a falling clock edge and does nothing, leaving the counter in state 0001. When the next rising clock edge arrives, the left flip-flop toggles back to 0, which the second flip-flop sees as a rising clock edge, causing it to toggle to 1. The third (S_2) flip-flop sees the second flip-flop's change as a falling edge and does nothing, and the state settles as 0010. We leave verification of the remainder of the cycle as an exercise.

Timing Issues*

Ripple counters are a form of a more general strategy known as clock gating.[2] **Clock gating** uses logic to control the visibility of a clock signal to flip-flops (or latches). Historically, digital system designers rarely used clock gating techniques because of the complexity introduced for the circuit designers, who must ensure that clock edges are delivered with little skew along a dynamically changing set of paths to flip-flops. Today, however, the power benefits of hiding the clock signal from flip-flops have made clock gating an attractive strategy. Nevertheless, digital logic designers and computer architects still almost never use clock gating strategies directly. In most of the industry, CAD tools insert logic for clock gating automatically. A handful of companies (such as Intel and Apple/Samsung) design custom circuits rather than relying on CAD tools to synthesize hardware designs from standard libraries of elements. In these companies, clock gating is used widely by the circuit design teams, and some input is occasionally necessary from the higher-level designers.

More aggressive gating strategies are also used in modern designs, but these usually require more time to transition between the on and off states and can be more difficult to get right automatically (with the tools), hence hardware designers may need to provide high-level information about their designs. A flip-flop that does not see any change in its clock input still has connections to high voltage and ground, and thus allows a small amount of **leakage current**. In contrast, with **power gating**, the voltage difference is removed, and the circuit uses no power at all. Power gating can be tricky—as you know, for example, when you turn the power on, you need to make sure that each latch settles into a stable state. Latches may need to be initialized to guarantee that they settle, which requires time after the power is restored.

If you want a deeper understanding of gating issues, take ECE482, Digital Integrated Circuit Design, or ECE527, System-on-a-Chip Design.

Machine Models

Before we dive fully into FSM design, we must point out that we have placed a somewhat artificial restriction on the types of FSMs that we use in our course. Historically, this restriction was given a name, and machines of the type that we have discussed are called Moore machines. However, outside of introductory classes, almost no one cares about this name, nor about the name for the more general model used almost universally in hardware design, Mealy machines.

What is the difference? In a **Moore machine**, outputs depend only on the internal state bits of the FSM (the values stored in the flip-flops). In a **Mealy machine**, outputs may be expressed as functions both of internal state and FSM inputs. As we illustrate shortly, the benefit of using input signals to calculate outputs (the Mealy machine model) is that input bits effectively serve as additional system state, which means that the number of internal state bits can be reduced. The disadvantage of including input signals in the expressions for output signals is that timing characteristics of input signals may not be known, whereas an FSM designer may want to guarantee certain timing characteristics for output signals.

In practice, when such timing guarantees are needed, the designer simply adds state to the FSM to accommodate the need, and the problem is solved. The coin-counting FSM that we designed for our class' lab assignments, for example, required that we use a Moore machine model to avoid sending the servo controlling the coin's path an output pulse that was too short to enforce the FSM's decision about which way to send the coin. By adding more states to the FSM, we were able to hold the servo in place, as desired.

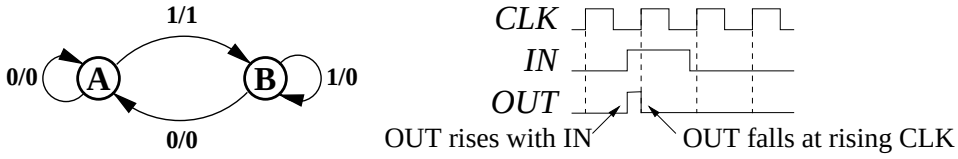
[2]Fall 2012 students: This part may seem a little redundant, but we're going to remove the earlier mention of clock gating in future semesters.

Why are we protecting you from the model used in practice? First, timing issues add complexity to a topic that is complex enough for an introductory course. And, second, most software FSMs are Moore machines, so the abstraction is a useful one in that context, too.

In many design contexts, the timing issues implied by a Mealy model can be relatively simple to manage. When working in a single clock domain, all of the input signals come from flip-flops in the same domain, and are thus stable for most of the clock cycle. Only rarely does one need to keep additional state to improve timing characteristics in these contexts. In contrast, when interacting across clock domains, more care is sometimes needed to ensure correct behavior.

We now illustrate the state reduction benefit of the Mealy machine model with a simple example, an FSM that recognizes the pattern of a 0 followed by a 1 on a single input and outputs a 1 when it observes the pattern. As already mentioned, Mealy machines often require fewer flip-flops. Intuitively, the number of combinations of states and inputs is greater than the number of combinations of states alone, and allowing a function to depend on inputs reduces the number of internal states needed.

A Mealy implementation of the FSM appears on the left below, and an example timing diagram illustrating the FSM's behavior is shown on the right. The machine shown below occupies state A when the last bit seen was a 0, and state B when the last bit seen was a 1. Notice that the transition arcs in the state diagram are labeled with two values instead of one. Since outputs can depend on input values as well as state, transitions in a Mealy machine are labeled with input/output combinations, while states are labeled only with their internal bits (or just their names, as shown below). Labeling states with outputs does not make sense for a Mealy machine, since outputs may vary with inputs. Notice that the outputs indicated on any given transition hold only until that transition is taken (at the rising clock edge), as is apparent in the timing diagram. When inputs are asynchronous, that is, not driven by the same clock signal, output pulses from a Mealy machine can be arbitrarily short, which can lead to problems.



For a Moore machine, we must create a special state in which the output is high. Doing so requires that we split state B into two states, a state C in which the last two bits seen were 01, and a state D in which the last two bits seen were 11. Only state C generates output 1. State D also becomes the starting state for the new state machine. The state diagram on the left below illustrates the changes, using the transition diagram style that we introduced earlier to represent Moore machines. Notice in the associated timing diagram that the output pulse lasts a full clock cycle.