5 Regular (Category 1) State Machines

5.1 Introduction

We know that, from a hardware perspective, state machines can be classified into two types, based on their *input connections*, as follows.

1) *Moore machines:* The input, if it exists, is connected only to the logic block that computes the next state.
2) *Mealy machines:* The input is connected to both logic blocks, that is, for the next state and for the actual output.

In Section 3.6 we introduced a new classification, also from a hardware point of view, based on the *transition types* and *nature of the outputs*, as follows (see figure 5.1).

1) *Regular (category 1) state machines:* This category, illustrated in figure 5.1a and studied in chapters 5 to 7, consists of machines with only untimed transitions and outputs that do not depend on previous (past) output values.
2) *Timed (category 2) state machines:* This category, illustrated in figure 5.1b and studied in chapters 8 to 10, consists of machines with one or more transitions that depend on time (so they can have all four transition types: conditional, timed, conditional-timed, and unconditional). However, all outputs are still independent from previous (past) output values.
3) *Recursive (category 3) state machines:* This category is illustrated in figure 5.1c and studied in chapters 11 to 13. It can have all four types of transitions, but one or more outputs depend on previous (past) output values. Recall that the outputs are produced by the FSM's *combinational* logic block, so the current output values are "forgotten" after the machine leaves that state; consequently, to implement a recursive (recurrent) machine, some sort of extra memory is needed.

As seen in this and in upcoming chapters, the classifications mentioned above (no other classification is needed) will immensely ease the design of hardware-based

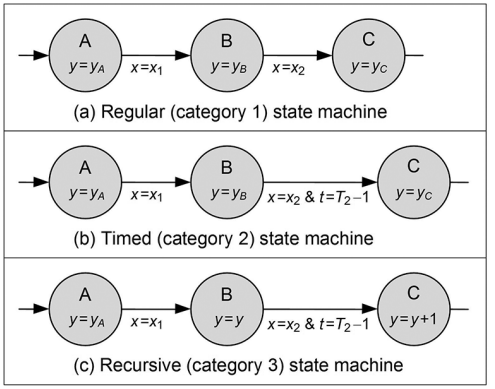


Figure 5.1
State machine categories (from a hardware perspective).

state machines. The two fundamental decisions before starting a design are then the following:

1) Decide the state machine category (regular, timed, or recursive).
2) Next, decide the state machine type (Moore or Mealy).

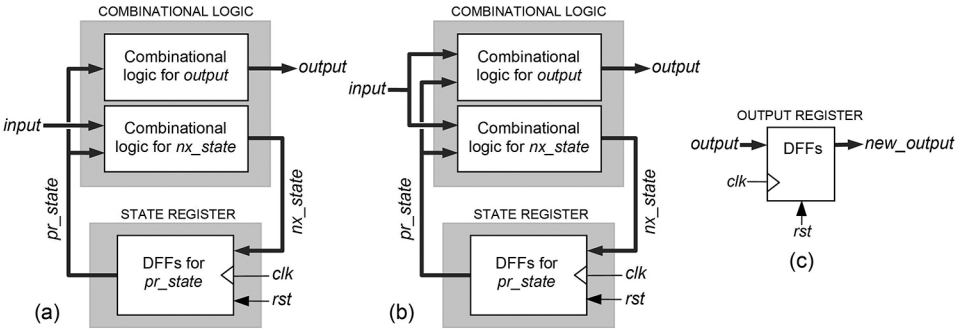
It is important to recall, however, that regardless of the machine category and type, the state transition diagram must fulfill three fundamental requisites (seen in section 1.3):

1) It must include all possible system states.
2) All state transition conditions must be specified (unless a transition is unconditional) and must be truly complementary.
3) The list of outputs must be exactly the same in all states (standard architecture).

5.2 Architectures for Regular (Category 1) Machines

The architectures for category 1 machines are summarized in figure 5.2. These representations follow the style of figures 3.1b,d, but the style of figures 3.1a,c could be used equivalently. The output register (figure 5.2c) is optional. The four possible constructions, listed in figure 5.2d, are summarized below.

Regular Moore machine (figure 5.2a): In this case, the input (if it exists) is connected only to the logic block for the next state. Consequently, the output depends only on the state in which the machine is (in other words, for each state, the output value in unique), resulting a synchronous behavior (see details in section 3.5). Because modern designs are generally synchronous, this implementation is preferred whenever the application permits.



(d)

Construction	Resulting category 1 circuit	Behavior
(a) only	Pure Moore machine (preferred)	Order-1 synchronous
(b) only	Pure Mealy machine	Asynchronous
(a) + (c)	Out-registered (pipelined) Moore machine	Order-2 synchronous
(b) + (c)	Out-registered (pipelined) Mealy machine	Order-1 synchronous

Figure 5.2
Regular (category 1) state machine architectures for (a) Moore and (b) Mealy types. (c) Optional output register. (d) Resulting circuits.

Regular Mealy machine (figure 5.2b): In this case, the input is connected to both logic blocks, so it can affect the output directly, resulting an asynchronous behavior. Therefore, the machine can have more than one output value for the same state (section 3.5).

Out-registered (pipelined) Moore machine: This consists of connecting the register of figure 5.2c to the output of the Moore machine of figure 5.2a. As seen in sections 2.5 and 2.6, two fundamental reasons for doing so are glitch removal and pipelined construction. As a result, the final circuit's output will be delayed with respect to the original machine's output by either one clock period (if the same clock edge is employed in the state register and in the output register) or by one-half of a clock period (if different clock edges are used). Note that the resulting circuit is order-2 synchronous because the original Moore machine was already a registered circuit (in other words, the input–output transfer occurs after two clock edges—see details in section 3.5). If in a given application this extra register is needed but its consequent extra delay is not acceptable, the next alternative can be used.

Out-registered (pipelined) Mealy machine: This consists of connecting the register of figure 5.2c to the output of the Mealy machine of figure 5.2b. The reasons for doing so are the same as for Moore machines. The resulting circuit is order-1 synchronous because the original Mealy machine is asynchronous. Consequently, the overall

behavior (with the output register included) is similar to that of a pure Moore machine (without the output register—see details in section 3.5).

5.3 Number of Flip-Flops

In general, and particularly in large designs, it is difficult to estimate the number of logic gates that will be needed to implement the desired solution. However, it is always possible to determine, and *exactly*, the number of flip-flops.

In the case of sequential circuits implemented as category 1 state machines, there are two demands for DFFs, as follows (see state-encoding options in section 3.7).

1) For the state register (see *nx_state* and *pr_state* in figure 5.2a, which are the state memory flip-flops' input and output, respectively; below, M_{FSM} is the number of states):
   For sequential or Gray encoding: $N_{FSM} = \lceil \log_2 M_{FSM} \rceil$. Example: $M_{FSM} = 25 \rightarrow N_{FSM} = 5$.
   For Johnson encoding: $N_{FSM} = \lceil M_{FSM}/2 \rceil$. Example: $M_{FSM} = 25 \rightarrow N_{FSM} = 13$.
   For one-hot encoding: $N_{FSM} = M_{FSM}$. Example: $M_{FSM} = 25 \rightarrow N_{FSM} = 25$.
2) For the output register (figure 5.2c, optional, with b_{output} bits):
   $N_{output} = b_{output}$. Example: $b_{output} = 16 \rightarrow N_{output} = 16$.

Hence, the total is $N_{total} = N_{FSM} + N_{output}$. In the examples that follow, as well as in the actual designs with VHDL and SystemVerilog, the number of flip-flops will be often examined.

5.4 Examples of Regular (Category 1) Machines

A series of regular FSMs are presented next. Several of these examples are designed later using VHDL (chapter 6) and SystemVerilog (chapter 7).

5.4.1 Small Counters

Counters are well-known circuits easily designed without the FSM approach using VHDL or SystemVerilog. Moreover, a counter might have thousands of states, rendering it impractical for representation as a regular state machine. Nevertheless, for designing counters without the help of any EDA tool (as done in sections 3.3 and 3.4), the FSM model can be very helpful, particularly if the counter is not too big and has several control inputs such as enable and up-down. Moreover, the implementation of such counters can be very illustrative of the FSM approach. For these reasons, an example is included in this section.

A 1-to-5 counter with enable and up-down controls is presented in figure 5.3 (just to practice, equivalent detailed and simplified representations are shown—recall figure 1.4). The circuit counts if *ena* = '1', or stops (and holds its last output value) otherwise. If *up* = '1', the circuit counts from 1 to 5, restarting then automatically from 1; oth-

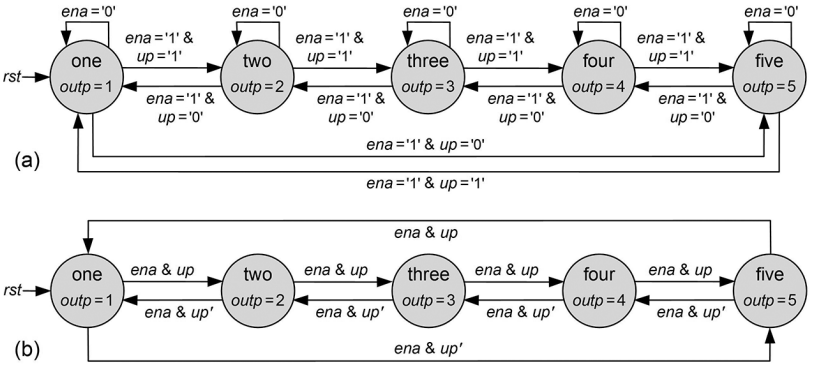


Figure 5.3
Detailed (a) and simplified (b) representations for a 1-to-5 counter with enable and up-down controls.

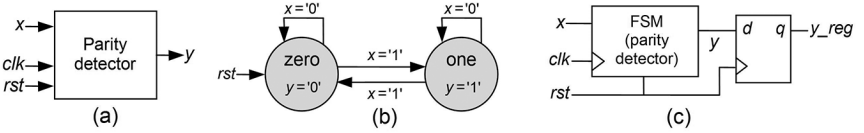


Figure 5.4
Parity detector. (a) Circuit ports. (b) State transition diagram. (c) Hardware block diagram.

erwise, it counts from 5 down to 1, restarting then automatically from 5. Because counters are inherently synchronous, the Moore model is the natural choice for their implementations.

Because this machine has $M_{FSM} = 5$ states, and the optional output register is generally not needed in counters, the number of flip-flops required to implement it (see section 5.3) is $N_{FSM} = 3$ if sequential, Gray, or Johnson encoding is used, or 5 for one-hot encoding.

VHDL and SystemVerilog implementations for this counter are presented in sections 6.6 and 7.5, respectively.

5.4.2 Parity Detector

This example concerns a circuit that detects the parity of a serial data stream. As depicted in figure 5.4a, x is the serial data input, and y is the circuit's response. The output must be y = '1' when the number of '1's in x is odd.

A basic solution for the case when a reset pulse is applied before every calculation starts is presented in figure 5.4b. In this case the parity value is the value of y after the last bit has been presented to the circuit (before a new reset pulse is applied). Note

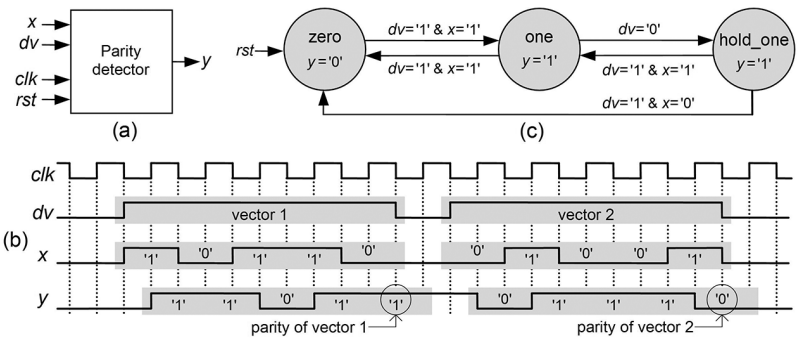


Figure 5.5
Another parity detector. (a) Circuit ports. (b) Illustrative time behavior. (c) State transition diagram.

the arrangement in figure 5.4c, based on the material seen in section 3.11; when the reset pulse goes up (which subsequently resets the FSM), it causes the value of *y* to be stored in the auxiliary register, producing *y_reg*, which stays stable (constant) until a new calculation is completed (i.e., a new reset pulse occurs).

A slightly different parity detection problem is depicted in figure 5.5, which has to be reset only at power-up (thus a more usual situation). A data-valid (*dv*) bit indicates the extension of the data vector whose parity must be calculated (when *dv* goes up, a new vector begins, finishing when *dv* returns to zero). It is assumed that after a calculation (data stream) is completed, the machine must keep displaying the final parity value until a new vector is presented, as depicted in the illustrative timing diagram of figure 5.5b, which shows two vectors of size 5 bits each, with final parity *y* = '1' for vector 1 and *y* = '0' for vector 2.

A Moore machine that complies with these specifications is presented in figure 5.5c (note that in this example *dv* and *x* are updated at the negative clock edge). Because of *dv*, this machine does not need to be reset before a new calculation starts. Indeed, depending on the encoding scheme (sequential or Gray, for example), this circuit might not need a reset signal at all because deadlock cannot occur (the unused codeword will converge back to one of the machines' states) and *dv* will cause the computations to be correct even if the initial state is arbitrary (see exercise 3.11).

5.4.3 Basic One-Shot Circuit

One-shot circuits are circuits that, when triggered, generate a single voltage or current pulse, possibly with a fixed time duration. This section discusses the particular case in which the time duration of the output is exactly one clock period. In this example it will be considered that the input lasts at least one clock period; generic cases are studied in sections 8.11.8 to 8.11.10, which deal specifically with triggered circuits.

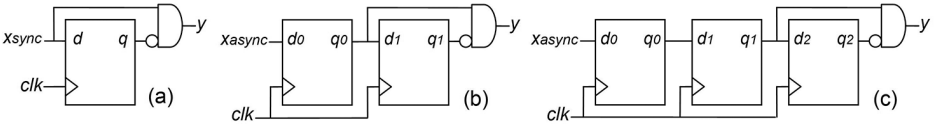


Figure 5.6
Trivial one-shot circuits. (a) Basic version, for synchronous input only. (b) Preceded by a synchronizing DFF, so the input can be asynchronous. (c) With a two-stage synchronizer.

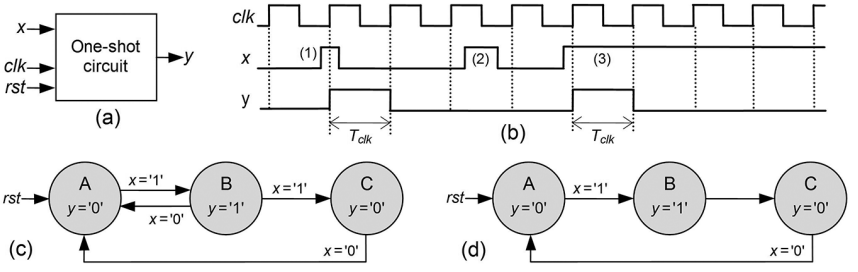


Figure 5.7
One-shot state machine. (a) Circuit ports. (b) Example of expected behavior. (c) State transition diagram. (d) An inferior solution (exercise 5.5).

In fact, a one-shot circuit (not employing the FSM approach) was already seen in chapter 2 (figure 2.10), with its schematic repeated in figure 5.6a. This option, however, is fine only if the triggering input (x) is synchronous; otherwise, the output pulse could last less than T_{clk}. For it to work with asynchronous inputs, another DFF is needed, as shown in figure 5.6b. A version with a full synchronizer (section 2.3) is shown in figure 5.6c.

The general operating principle is illustrated in figure 5.7. The circuit ports are shown in figure 5.7a, where x is the triggering input and y is the one-shot output. An illustrative timing diagram is presented in figure 5.7b, with x having an arbitrary duration and y lasting exactly one clock period. Pulse 1 lasts less than T_{clk} but happened to fall under a positive clock edge, so it was detected. This is obviously not guaranteed to happen, as illustrated for pulse 2. Only if the duration is T_{clk} or longer, as for pulse 3, is the triggering of y guaranteed. Note that x and y are uncorrelated (mutually asynchronous) if x and *clk* are uncorrelated.

A solution using a regular (category 1) Moore machine is presented in figure 5.7c. Note that it stays in state B during only one clock period; because y = '1' occurs only in that state, the desired pulse results. An inferior solution is presented in figure 5.7d (see exercise 5.5).

As a final comment, let us consider the circuit of figure 5.6b, which is a kind of optimized synchronous version of the one-shot circuit. Because the solution in figure 5.7c is also synchronous (all Moore machines are), would you expect the circuit that implements this state machine to be equal or at least similar to that of figure 5.6b? (See exercise 5.5.)

5.4.4 Temperature Controller

Figure 5.8a shows a circuit diagram for a temperature controller of an air conditioning system. In the upper branch, the room temperature is sensed by some type of temperature sensor and converted to digital format by the ADC (analog-to-digital converter), producing the signal T_{room}. In the lower branch, the user, by means of two pushbuttons (*up*, *dn*), selects the reference (desired) temperature, producing the signal T_{ref}. Depending on the values of these two signals, the controller core decides whether to heat the room (h = '1'), to cool it (c = '1'), or to stay in the idle state.

Because mechanical switches are subject to bounces before they finally settle in the proper position, the pushbuttons must be debounced. However, debouncers are timed circuits, thus requiring a timed (category 2) machine to be implemented. Such machines are seen in chapter 8, so for now let us just consider that the proper value is produced for T_{ref} (the design of this block is treated in section 8.11.4). For example, T_{ref} could be selected in the 60°F to 90°F range with an initial value (on power-up, defined by the reset signal) of 73°F, if degrees Fahrenheit are used, or in the 15°C to 30°C range with a default value of 23°C, if degrees centigrade are employed instead.

An important addition to the system is depicted in figure 5.8b, which consists of a display accessed by means of a multiplexer. The display shows the room temperature while the selection pushbutton (*sel*, with no need for debouncing, not shown in the figure) is at rest (*sel* = '0') or the reference temperature while it is pressed (*sel* = '1').

A state machine for the controller core, using the Moore approach, is depicted in figure 5.8c. ΔT represents the system hysteresis, which is generally a fixed circuit

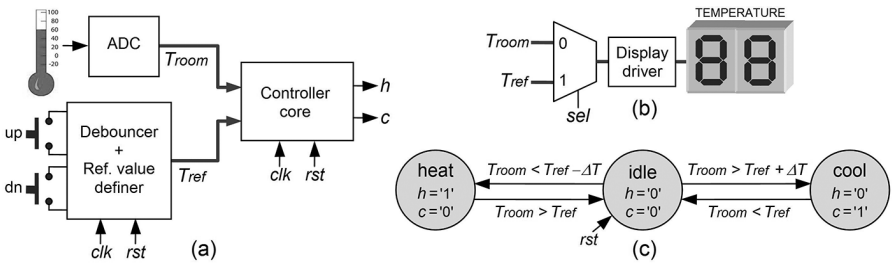


Figure 5.8

Temperature controller. (a) Overall circuit diagram. (b) Display driver. (c) State machine for the controller core block.

parameter. For example, if ΔT = 1°F, the room temperature will be kept within T_{ref} ± 1°F. By comparing T_{room} to T_{ref} and taking into account the hysteresis, the machine will be able to produce the proper values for *h* and *c*.

Finally, note that the inputs from the pushbuttons are asynchronous with respect to the system clock, which could, in principle, cause metastability (see section 2.3). This, however, is prevented here by the debouncer (section 8.11.3).

5.4.5 Garage Door Controller

This example presents a garage door controller that operates as follows. If the door is completely closed or completely open and the remote is activated, the motor is turned on in the direction to open or close it, respectively. If the door is opening or closing and the remote is activated, the door stops. If the remote is activated again, the motor is turned on to move the door in the opposite direction.

The circuit ports are depicted in figure 5.9a, where *remt* (command from the remote control), *sen1* (door-open sensor), and *sen2* (door-closed sensor) are the inputs (plus the conventional *clk* and *rst* signals), and *ctr* (control) is the output. Note that *ctr* has two bits; *ctr*(1) turns the motor on ('1') or off ('0'), whereas *ctr*(0) defines its direction, opening ('0') or closing ('1') the door (thus the value of the latter does not matter when the former is '0').

A preliminary state diagram is shown in figure 5.9b. The transition control signals are *remt*, *sen1*, and *sen2*. Note that this machine complies with all three requisites of

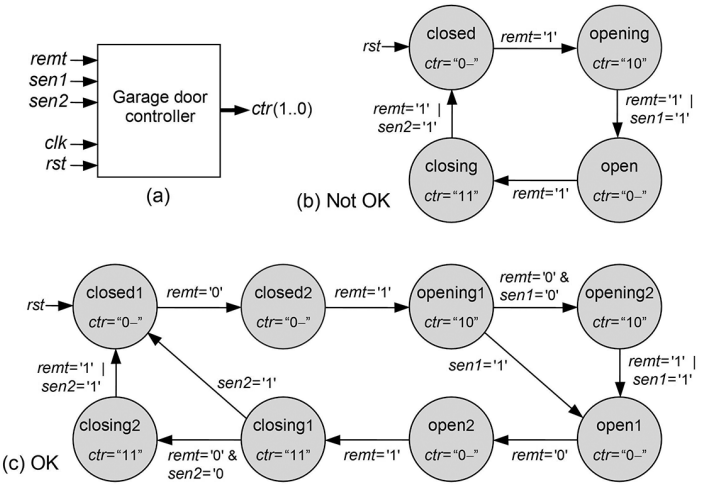


Figure 5.9

Garage door controller. (a) Circuit ports. (b) Bad solution (with state-bypass). (c) Good solution.

section 1.3. However, it exhibits a major problem, which is state bypass (see section 4.2.4). For example, if the door is closed and a long (lasting several clock cycles) *remt* = '1' command is received, the machine goes around the entire loop. Of course, if a one-shot circuit (section 5.4.3) is used to reduce the duration of *remt* to a single clock period, then this machine is fine.

A corrected diagram is presented in figure 5.9c, containing additional states that wait for *remt* to return to zero before proceeding, thus eliminating the state-bypass problem. This is a Moore machine because there is no reason to employ an asynchronous solution in this kind of application. Glitches at the output are not a problem here, so the optional output register is not needed.

A good practice in this kind of application is to include debouncers for the signals coming from the remote control and from the sensors, which not only eliminate the need for synchronizers but also prevent short input glitches (due to lightning or the switching of large electric currents, for example) from activating the machine (in this case, it has to be a full debouncer, like that in section 8.11.3, for example).

Because the machine of figure 5.9c has $M_{FSM} = 8$ states, the required number of DFFs is $N_{FSM} = 3$ if sequential or Gray encoding is used, 4 for Johnson, or 8 for one-hot.

VHDL and SystemVerilog implementations for this garage door controller are presented in sections 6.7 and 7.6, respectively.

5.4.6 Vending Machine Controller

This example deals with a controller for a vending machine. It is assumed that it sells candy bars for the single price of $0.40, accepting nickel, dime, and quarter coins.

The circuit ports are depicted in figure 5.10a. The inputs *nickel_in*, *dime_in*, and *quarter_in* are generated by the coin collector, informing the type of coin that was deposited by the customer. The inputs *nickel_out* and *dime_out* are generated by the coin dispenser mechanism, informing the type of coin that was returned to the customer. The last nonoperational input is *candy_out*, produced by the candy dispenser mechanism, informing that a candy was delivered to the customer. The outputs *disp_nickel* and *disp_dime* tell the coin dispenser mechanism that a nickel or a dime must be returned to the customer, while the output *disp_candy* tells the candy bar dispenser mechanism that a candy bar must be delivered to the customer.

A corresponding Moore machine is presented in figure 5.10b. To simplify the notation, numbers were used instead of names (see other examples of equivalent state diagram representations in section 1.4). The state names correspond to the accumulated amount (*credit*). The transition conditions refer to the last coin entered, with negative values indicating change returned to the customer. In the coin-return operations it was opted to deliver the largest coins possible. After the machine reaches the state 40 (thick circle), the only way to return to the initial state is by receiving a

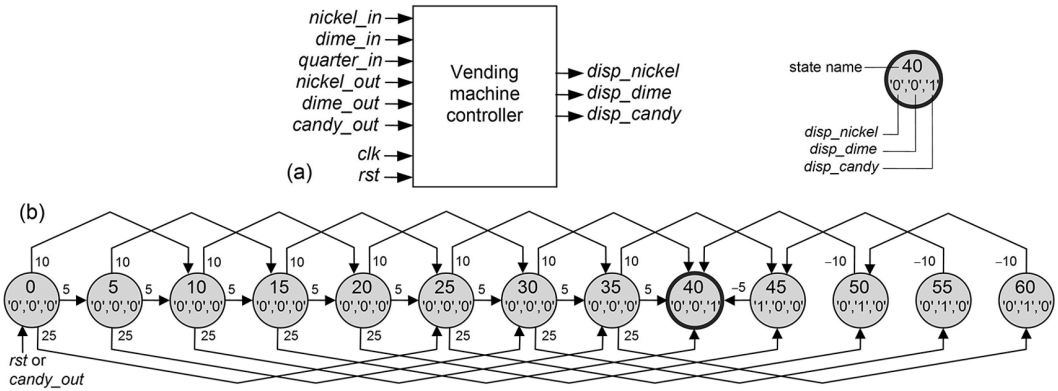


Figure 5.10
Controller for a vending machine that sells candy bars for $0.40, accepting nickels, dimes, and quarters. (a) Circuit ports. (b) Corresponding Moore machine (state-bypass prevention not included).

candy_out = '1' command from the candy-delivering mechanism confirming that a candy bar was dispensed or a reset pulse.

Note that the machine of figure 5.10b is subject to state bypass (section 4.2.4) if the inputs last longer than one clock period (which is generally the case in this kind of application), so wait states (or a flag or one-shot conversion) must be added (exercise 5.11).

Because glitches are definitely not acceptable in this application, the optional output register should be used here. In regard to the inputs, we can assume that they are produced by other circuits that process the actual inputs and hence operate with the same clock as our state machine, dispensing with the use of debouncers and/or synchronizers (although they might be needed at the inputs of preceding circuits).

If we assume that all control inputs to this machine last exactly one clock period (due to one-shot circuits, for example), so state bypass cannot occur and additional states are not needed, the number of DFFs required to build it (with M_{FSM} = 13 states) is N_{FSM} = 4 if sequential or Gray encoding is used, 7 for Johnson, or 13 for one-hot, plus N_{output} = 3 for the output register.

5.4.7 Datapath Control for an Accumulator

Before we examine this example, a review of section 3.13 is suggested.

In this example we assume that the datapath of figure 3.22a must operate as an add-and-accumulate circuit (ACC), accumulating in register A four consecutive values

of *inpB*. The data-valid bit (*dv*), when asserted (during just one clock period), will again be responsible for starting the computations, after which the resulting value must remain displayed at *ALUout* until another pulse occurs in *dv*. In summary, the operations are: 0 + B → A, A + B → A, A + B → A, and A + B → A.

Recall that in a datapath-based design the FSM is not responsible for implementing the whole computation but just the *control unit* (shown on the left in figure 3.22a), which controls the datapath. In other words, the FSM must produce the signals *selA* (selects the data source for register A), *wrA* and *wrB* (enable writing into registers A and B), and *ALUop* (produces the ALU opcode, defining its operations, according to the table in figure 3.22b).

An illustrative timing diagram (similar to what was done in figure 3.22c) for an FSM that controls this datapath such that the desired accumulator results is presented in figure 5.11a. Note that the computations take five steps (called *start*, *acc1*, *acc2*, *acc3*, and *acc4*), after which the control unit (FSM) returns to the *idle* state (so the machine has six states). The corresponding state transition diagram, which is a direct translation of the timing diagram (compare the values in the timing diagram against those in the state transition diagram), is exhibited in figure 5.11b. Observe that this control unit is indeed a category 1 machine.

Because this machine has $M_{FSM} = 6$ states, and the optional output register is generally not needed in control units, the number of flip-flops required to implement it (see section 5.3) is $N_{FSM} = 3$ if sequential, Gray, or Johnson encoding is used or 6 for one-hot.

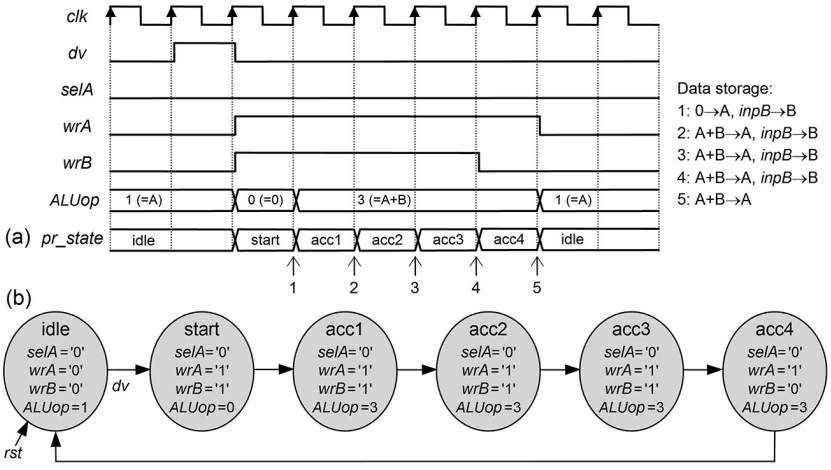


Figure 5.11
(a) Illustrative timing diagram for the datapath of figure 3.22a operating as an accumulator. (b) Corresponding Moore machine.

5.4.8 Datapath Control for a Greatest Common Divisor Calculator

Before we examine this example, a review of section 3.13 is suggested. Particular attention should be paid to comment number 4 at the end of that section, which is helpful here.

This section shows another example of a datapath-based circuit. The datapath must compute the GCD (greatest common divisor) between two integers. The corresponding algorithm is shown in figure 5.12; the largest value is substituted with the difference between it and the other value until the values become equal, which is then declared to be the GCD. A corresponding flowchart is also included in figure 5.12. As in the previous example, a *dv* bit, when asserted (during one clock period), must start the computations.

The datapath to be used in this example is depicted in figure 5.13a. The ALU's opcode table is shown in figure 5.13b. The ALU has also an auxiliary output (*sign*) that indicates whether its output (*ALUout*) is zero ("00"), positive ("01"), or negative ("10"), as listed in figure 5.13c.

As shown, the datapath's control signals are *selA* and *selB* (select the data sources for registers A and B), *wrA* and *wrB* (enable writing into registers A and B), and *ALUop* (produces the ALU opcode, defining its operations, according to the table in figure 5.13b). The control unit (FSM) is responsible for generating all control signals.

An illustrative timing diagram for an FSM that controls this datapath such that the desired computations occur is presented in figure 5.13d. Dashed lines indicate "don't care" values. Because *inpA* = 9 and *inpB* = 15 were adopted, the following computations are expected: Iteration 1, 9 → A, 15 → B; Iteration 2, B > A, then 15 – 9 = 6 → B; Iteration 3, A > B, then 9 – 6 = 3 → A; Iteration 4, B > A, so 6 – 3 = 3 → B. Because A = B, GCD = A = 3.

Observe in figure 5.13d that the time slots are identified as *idle* (waiting for a *dv* bit), *load* (*inpA* and *inpB* are stored in A and B), *writeA* (*ALUout* is stored in A), and

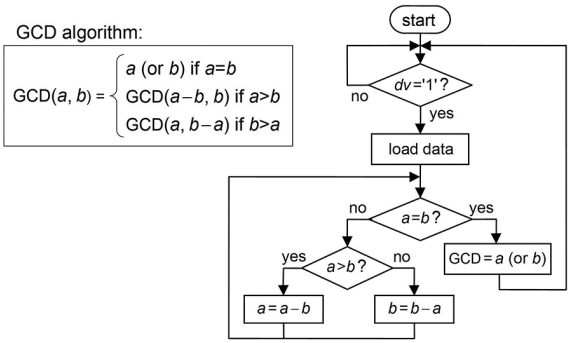
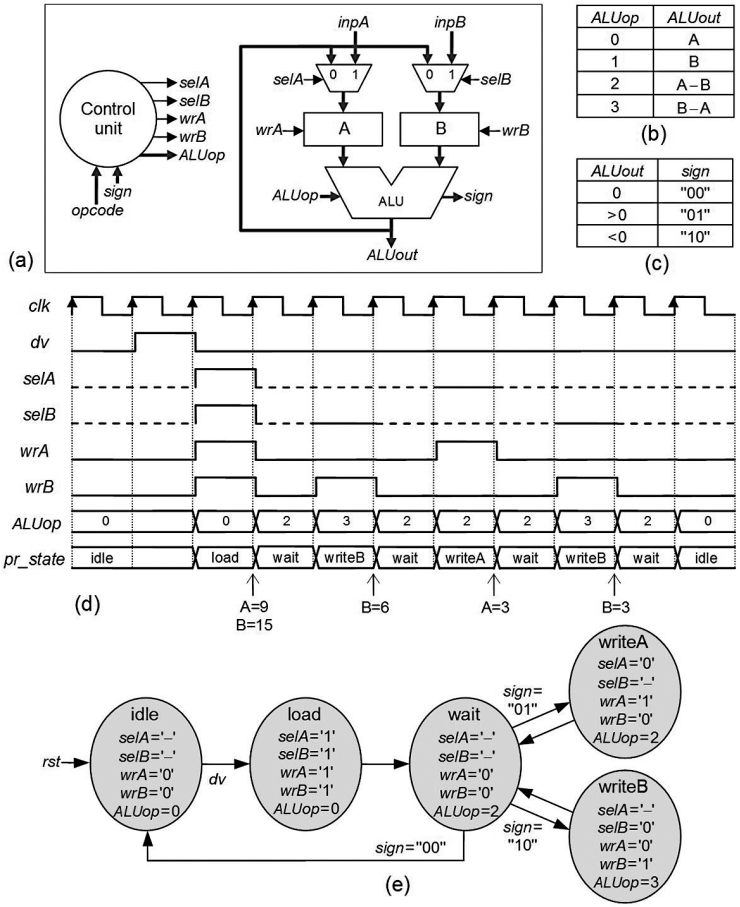


Figure 5.12
GCD algorithm and flowchart.



ALUop	ALUout
0	A
1	B
2	A−B
3	B−A

ALUout	sign
0	"00"
>0	"01"
<0	"10"

Figure 5.13
(a) Datapath and control unit for a GCD calculator. (b) ALU's opcode table. (c) ALU's sign table. (d) Illustrative timing diagram, for *inpA* = 9 and *inpB* = 15. (e) Corresponding state machine.

writeB (*ALUout* is stored in B). Observe also the presence of a *wait* time slot after every data storage, which is needed for the data to be effectively ready for comparison before an actual comparison occurs (recall comment 4 of section 3.13).

A corresponding state transition diagram is presented in figure 5.13e, which is a direct translation of the timing diagram (compare the values in the plots against those in the state transition diagram). Note that after each write-enabling state (*load*, *writeA*, and *writeB*) the machine goes unconditionally to the *wait* state. In the *idle* state, *wrA* = *wrB* = '0', so nothing can be written into the registers, and because *ALUop* = 0, the output is *ALUout* = A, so the computed GCD value is kept unchanged until *dv* is asserted again.

VHDL and SystemVerilog implementations for this control unit are presented in sections 6.8 and 7.7, respectively.

5.4.9 Generic Sequence Detector

This is another interesting example from a conceptual point of view. Say that we want to design a signature detector that searches for the string "*abc*" in a sequential data stream, examining one character at a time (a character here represents a bit vector with any number of bits). So this is exactly the same problem presented in the very first state transition diagram of the book (figure 1.3, repeated in figure 5.14a). In this example it was assumed that $a \neq b \neq c$, so this machine works well. But let us consider now a completely generic situation, in which *a*, *b*, and *c* are *programmable*, so we can no longer assume that they are all different. Will this machine still work?

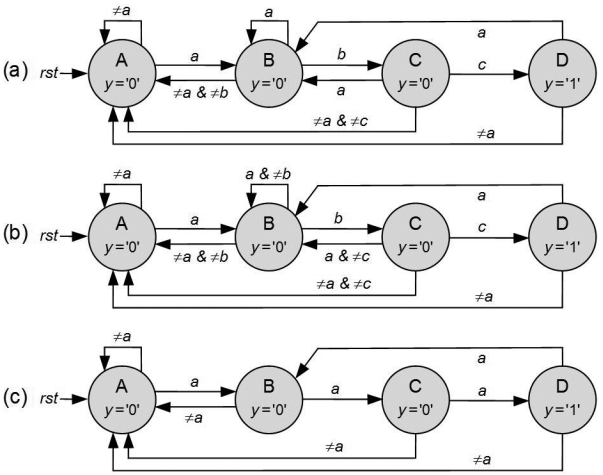


Figure 5.14

Generic string detection. (a) Nongeneric case (requires $a \neq b \neq c$). (b) Completely generic implementation due to the inclusion of priorities in the transition conditions. (c) Example for the case of $a = b = c$.

To answer this question, let us assume that $a = b$, so b can be replaced with a in figure 5.14a. Consequently, state B (for example) has the following transition conditions: a in the BB transition; a also in the BC transition; and $\neq a$ & $\neq b = \neq a$ in the BA transition. This shows that state B is now *overspecified* because both BB and BC transitions are governed by the same condition (a). Therefore, this machine is not fine for generic values of a, b, and c.

The new question then is "How do we fix overspecifications?" We do it in the way explained in section 1.5, that is, with the establishment of *priorities*. This is done in figure 5.14b. For state B, the BC transition must have priority over the BB transition, so the transition condition in the former remains just b, while that in the latter becomes a & $\neq b$. Likewise, for state C, the CD transition must have priority over the CB transition; thus, the transition condition in the former remains c, whereas that in the latter becomes a & $\neq c$.

As an example, figure 5.14c shows the extreme case in which $a = b = c$. Then $\neq a$ & $\neq b = \neq a$, $\neq a$ & $\neq c = \neq a$, a & $\neq b$ = null (so the BB transition disappears), and a & $\neq c$ = null (the CB transition also disappears).

The only restriction of this generic string detector is that it detects only nonoverlapping strings.

5.4.10 Transparent Circuits

We close this chapter with the description of a special (although uncommon) type of circuit for FSMs, which consists of sequential circuits that are required to be "transparent" (i.e., the output must "see" the input; in other words, if the input changes, so should the output). If implemented using an FSM, the circuit must provide outputs that are capable of changing when the input changes, even if the machine remains in the same state.

As an example, consider the case in figure 5.15a, with inputs a and b and output y. The output must be $y = a$ during one clock period, $y = a{\cdot}b$ during the next period, and finally $y = b$ during the third clock cycle, with this sequence repeated indefinitely. Corresponding Moore and Mealy diagrams are included in figures 5.15b,c. Note that because the machine must go to the next state at every clock cycle, its transitions are unconditional.

Because in this case the output depends solely on the machine's state, a Moore machine seems to be the natural choice. However, because the output must change when the input changes, a Mealy machine, being asynchronous, would be recommended. In fact, both are fine.

In the Moore case the transparency problem is circumvented by associating the machine with switches such as the multiplexer in figure 5.15d, in which case the machine plays just the role of mux selector (in this example, the resulting machine is clearly just a 0-to-2 counter), so even though the machine is not transparent, the

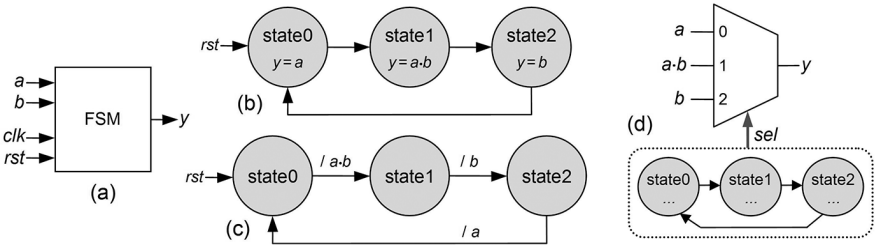


Figure 5.15
A "transparent" circuit. (a) Circuit ports. (b) Moore and (c) Mealy state transition diagrams. (d) Typical implementation based on the Moore model.

overall circuit is (this is typically what a VHDL/SystemVerilog compiler would do). In the Mealy case the implementation is straightforward, but the output will be one clock cycle ahead of the desired sequence (compare figures 5.15b and 5.15c).

5.4.11 LCD, I²C, and SPI Interfaces

Three special additional design examples are presented in chapter 14, consisting of circuits for interfacing with alphanumeric LCD displays and for implementing I²C or SPI serial interfaces. Depending on the application, any of the three FSM categories might be needed in these circuits; for instance, in the LCD driver example of section 14.1, a category 1 FSM is employed, whereas in the I²C and SPI serial interfaces of sections 14.2 and 14.3, categories 2 and 3 are used.

5.5 Exercises

Exercise 5.1: Machine Category and Number of Flip-Flops

a) Why are the state machines in figures 5.3, 5.9c, and 5.13e (among others) said to be of category 1?

b) How many DFFs are needed to implement each of these FSMs using (*i*) sequential encoding, (*ii*) Gray encoding, or (*iii*) one-hot encoding?

Exercise 5.2: Metastability and Synchronizer

a) Solve exercise 2.2 if not done yet.

b) Consider now the garage door controller of figure 5.9. (*i*) Which inputs are asynchronous? (*ii*) If no debouncing circuits (which are synchronous) are adopted for the asynchronous inputs, are synchronizers indispensable in this application?

Exercise 5.3: Need for Reset

a) Solve exercise 3.10 if not done yet.

b) Solve exercise 3.11 if not done yet.

Exercise 5.4: Truly Complementary Transition Conditions

In section 1.5 the importance of having the state transition diagram neither under- nor overspecified was discussed. What happens if, in the garage door controller of figure 5.9c, the condition *sen1* = '0' is removed from the *opening1-opening2* transition, or the condition *sen2* = '0' is removed from the *closing1-closing2* transition?

Exercise 5.5: One-Shot Circuits Analysis

a) It is said in section 5.4.3 that the solution in figure 5.7d is inferior to that in figure 5.7c. Why? (Suggestion: fill in the last two plots of figure 5.16 and you will see the answer.)

b) Is reset indispensable in these two solutions?

c) In order to answer the question posed at the end of section 5.4.3, solve exercise 3.3 if not done yet.

Exercise 5.6: Two-Signal-Triggered One-Shot Circuit

Figure 5.17 shows an illustrative timing diagram for a one-shot circuit that is not triggered by a single signal but rather by a pair of signals. The triggering condition is the following: the one-shot pulse (in *y*) must be generated if the control signal *x* lasts at least as long as the *dv* pulse (this is obviously checked only at positive clock transitions). Note in the figure that only the first pulse of *x* fulfills this requirement, so the one-clock-period pulse in *y* has to be produced only in that case. Draw the state transition diagram for a state machine capable of implementing this circuit.

Exercise 5.7: Arbiter

Arbiters are used to manage access to shared resources. An example is depicted in figure 5.18, which shows three peripherals (P1 to P3) that use a common bus

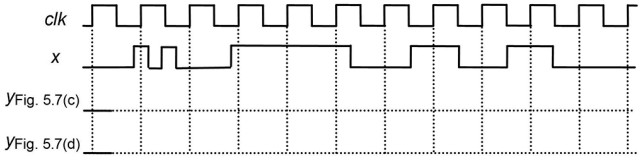


Figure 5.16

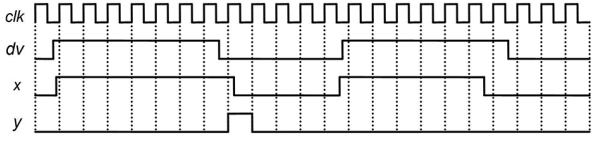


Figure 5.17

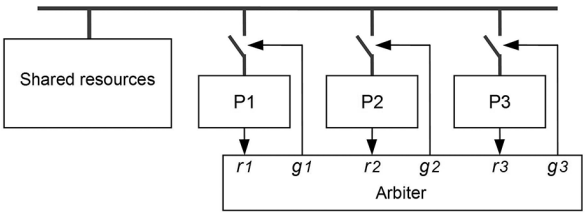


Figure 5.18

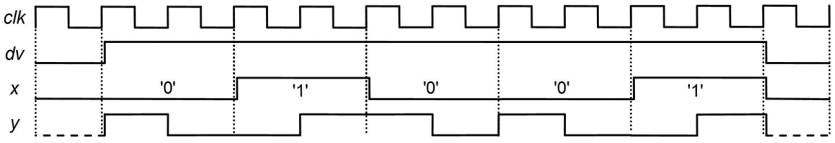


Figure 5.19

to access common resources. Obviously, only one of them can use the bus at a time; for example, if P1 wants to use the bus, it issues a request (r_1 = '1') to the arbiter, which grants (g_1 = '1') access only if the bus is idle at that moment. If multiple requests are received by the arbiter, access is granted based on preestablished priorities. Assuming that the priorities are P1 > P2 > P3, draw a state transition diagram for a machine capable of implementing this arbiter. The machine's input and output are the vectors $r = r_1r_2r_3$ and $g = g_1g_2g_3$, respectively (besides clock and reset, of course).

Exercise 5.8: Manchester Encoder

An IEEE Manchester encoder produces a low-to-high transition when the input is '1' or a high-to-low transition when it is '0', as illustrated in figure 5.19 for the sequence "01001". Note that each input value lasts two clock periods. Observe also the presence of a *dv* bit, which defines the extent of the vector to be encoded (dashed lines in *y* indicate "don't care" values). To be more realistic, *dv* is produced at the same time that the first valid bit is presented; additionally, a small propagation delay is included between clock transitions and corresponding responses. Assume that the machine too must operate at the positive clock edge.

a) Draw a state transition diagram for a Moore machine capable of implementing this encoder.

b) Redraw the illustrative timing diagram of figure 5.19 for your Moore machine, including in it a plot for *pr_state*. Does the Moore circuit behave exactly as in figure 5.19, or is *y* one clock cycle delayed?

c) Redo the design, this time employing a Mealy machine.

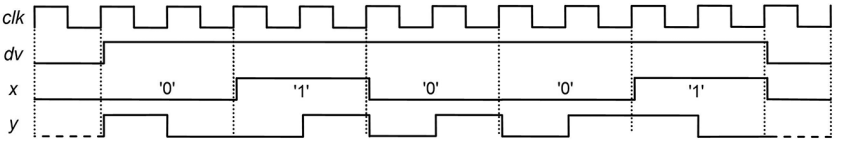


Figure 5.20

d) Repeat part b now for your Mealy solution.
e) Say that we want the output to be completely clean. Are any of the solutions above guaranteed to be glitch-free? If not, how can glitches be removed? What happens then with the time response?

Exercise 5.9: Differential Manchester Encoder

Figure 5.20 illustrates the operation of a differential Manchester encoder for the sequence "01001". Note that the shape of the output pulse remains unchanged when the input is '0' but gets inverted when it is '1'. For example, if the last pulse was a '1'-to-'0' pulse, the next pulse must be '1'-to-'0' if the input is '0' or '0'-to-'1' if it is '1'. Observe the presence of a *dv* bit, which defines the extent of the vector to be encoded (dashed lines in *y* indicate "don't care" values). To be more realistic, *dv* is produced at the same time that the first valid bit is presented; additionally, a small propagation delay has been included between the clock transitions and the corresponding responses. Assume that the machine too must operate at the positive clock edge.

a) Draw a state transition diagram for a Moore machine capable of implementing this encoder.
b) Redraw the illustrative timing diagram of figure 5.20 for your solution, including in it a plot for *pr_state*. Does the Moore circuit behave exactly as in figure 5.20, or is *y* one clock cycle delayed?

Exercise 5.10: Time-Ordered "111" Detector

Draw the state transition diagram for an FSM that detects the sequence *abc* = "111" under the constraint that it must be time ordered; that is, *a* = '1' must occur (and hold), then *b* = '1' must also occur (and hold), and finally, *c* = '1' must happen. The circuit ports are shown in figure 5.21a. The circuit operation is illustrated in figure 5.21b, where *x* = '1' occurs when *abc* = "111", but in a time-ordered fashion.

Exercise 5.11: Vending Machine

It was seen that the vending machine controller of figure 5.10b must be improved to avoid state bypass. Present a solution for this problem. Is it better to include wait

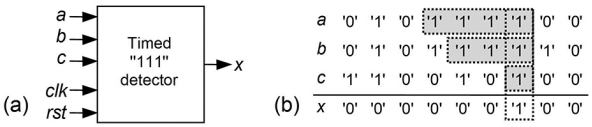


Figure 5.21

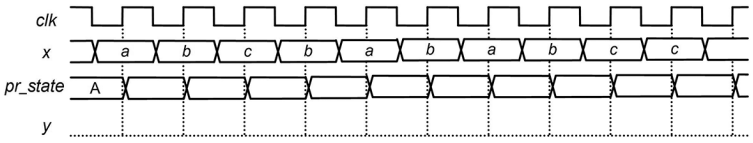


Figure 5.22

states or a flag or to convert the inputs into one-shot signals with one-clock-period duration?

Exercise 5.12: Time Behavior of a String Detector

Consider the Moore-type state machine of figure 5.14a, which detects the sequence "*abc*" for the case of $a \neq b \neq c$, where x and y represent the input and output, respectively.

a) Complete the timing diagram of figure 5.22 for the given values of x. Note that a little propagation delay was included between the clock transitions and the respective changes in the present state; do the same for y.

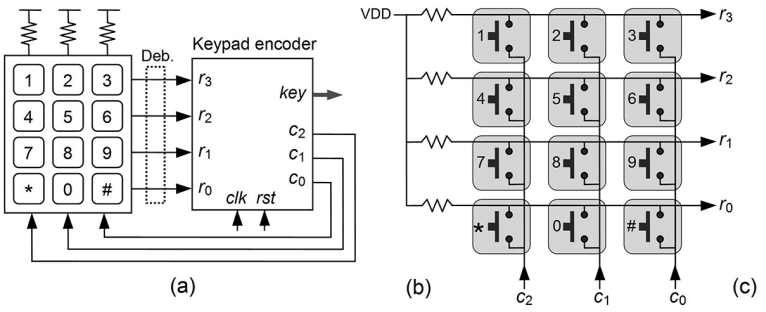
b) Does the output go up immediately when the sequence "*abc*" occurs or only at the next (positive) clock edge? Is this result as you expected? (Recall that Moore machines are fully synchronous.)

Exercise 5.13: Generic Overlapping String Detector

We saw in section 5.4.9 a generic approach for the implementation of nonoverlapping string detectors. In that case, if the sequence to be detected were "*aba*", for example, the response to the serial bit stream "*abababab…*" would be "00100010001…", whereas here, because overlaps must be allowed, it should be "0010101…". Can you find a generic solution (with or without a state machine) for this case?

Exercise 5.14: Keypad Encoder

Figure 5.23a shows a 12-key keypad for which we need to design an encoder (and possibly also a debouncer—debouncers are discussed in chapter 8). The actual push-button connections can be seen in figure 5.23b, where $r(3:0)$ and $c(2:0)$ represent the keypad's rows and columns, respectively. Note that because of the pull-up resistors,



c(2:0)	r(3:0)	key(3:0)
011	0111	0001 (1)
	1011	0100 (4)
	1101	0111 (7)
	1110	1010 (*)
	1111	1111 (n)
101	0111	0010 (2)
	1011	0101 (5)
	1101	1000 (8)
	1110	0000 (0)
	1111	1111 (n)
110	0111	0011 (3)
	1011	0110 (6)
	1101	1001 (9)
	1110	1011 (#)
	1111	1111 (n)

Figure 5.23

the rows' voltages are all high when no switch is pressed. The keypad encoder must connect the bottom of one column at a time to ground ('0'), then read the resulting row values, converting them into the respective codeword, as listed in figure 5.23c (n stands for "none"); for example, if c = "011", which means that the leftmost column is being inspected, and the reading is r = "1011", then we know that pushbutton 4 is pressed. Present a solution for this encoder. (A possible solution for the debouncer is treated in exercise 8.11.)

Exercise 5.15: Datapath Controller for a Largest-Value Detector

Say that you are given the datapath of figure 5.13a, with *inpB* monitoring a serial data stream, of which the largest value must be determined (placed at the ALU output, *ALUout*). The monitoring should start when a *dv* bit is asserted, ending when *dv* returns to zero.

a) Develop a state transition diagram (as in figure 5.13e) for an FSM capable of implementing the corresponding control unit. Include in it "nop" (no operation) states if necessary to have the number of clock cycles be the same in all iterations.
b) Present an illustrative timing diagram for your machine (as in figure 5.13d), assuming that the values presented to the circuit (while dv = '1') are $5 \rightarrow 8 \rightarrow 4 \rightarrow 0$. (If you prefer, do part b before part a.)

Exercise 5.16: Datapath Controller for a Square Root Calculator

To calculate $z = (x^2 + y^2)^{1/2}$, where x, y, and z are unsigned integers, the expression $z = \max(a - a/8 + b/2, a)$ can be used, where $a = \max(x, y)$ and $b = \min(x, y)$. Recall that to divide an integer by 8 or by 2 all that is needed is to shift it to the right three positions or one position, respectively. Make the adjustments that you find necessary in the datapath of figure 5.13a (for example, include a shift-right option in one of the existing registers or in a new register at the ALU output), then devise a state machine that computes the square root above using that datapath.

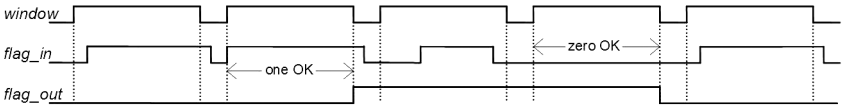


Figure 5.24

Exercise 5.17: Flag Monitor

Develop an FSM for a circuit that monitors a flag such that, if the flag remains constant within a given time window, the output copies the measured (constant) flag value. This is illustrated in figure 5.24; if *flag_in* has no transitions at all while *window* is high, then *flag_out* gets the value of *flag_in*; otherwise, it keeps the same value that it had when the time window started.