Chapter 8
The Finite State Machine as System Controller

Abstract The Finite State Machine can implement any algorithm, but it becomes over complicated when dealing with data. It is therefore convenient to include in digital systems other components that are more efficient to process and memorize data under the machines control. Such systems are called controller and datapath and described in this chapter. They optimize the sharing of duties between Finite State Machine and an external architecture. They are the first choice for the design of a wide variety of systems.

This chapter presents a collection of design examples of *synchronous digital systems* using an FSM as *system controller*.

8.1 Digital Systems

In previous chapters, we have seen that the FSM can model and implement various algorithms. A FSM describes a system in terms of the evolution of states and outputs as functions of inputs. From a theoretical perspective, the FSM can represent any discrete system that evolves over time moving from one state to another and has a finite number of inputs, outputs, and states.

Still, describing and designing a system in terms of *states*, while very general and versatile, can become overly complex and impracticable when the number of states becomes very large.

As we have seen with the shift register, the number of states in data management systems depends on all the possible configurations that the data can take on. For example, an 8-bit shift register will be described by an algorithm with at least 256 states. To describe a microprocessor in terms of states, one needs to only observe that a single 32-bit register (a microprocessor could contain many of them) would require 2^{32} states.

Regular digital structures for data management are available as blocks. They can be combinational (e.g., multiplexer, demultiplexer, encoders, decoders, and arithmetic circuits) or sequential (registers, counters, memories, and many others). Although the FSM can do arithmetic and logical operations, they can be done more effectively by dedicated devices.

G. Donzellini et al., *Introduction to Digital Systems Design*,
https://doi.org/10.1007/978-3-319-92804-3_8

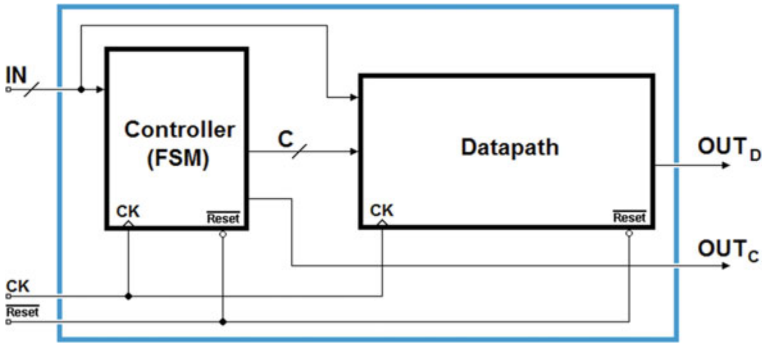
A digital system can be optimized by dividing its tasks between a *“controller”* and *“dedicated components.”*

The *controller* that we will design as FSM will manage the functioning of the system, while the *dedicated components* define its *“architecture,”* which interacts directly with the data.

The system’s *controller* is sometimes called *“sequencer”* or *“timing generator.”* The architecture is often called *“datapath.”* Hereinafter, we will use the terms *controller* and *datapath*.

8.2 Open Control Systems

In the simplest version of this system, the *controller* gives commands to the *datapath* without receiving *any feedback*, as we can see in the figure below. The system’s outputs can be generated by the *controller* or the *datapath*. Inputs can be dealt with by both subsystems.



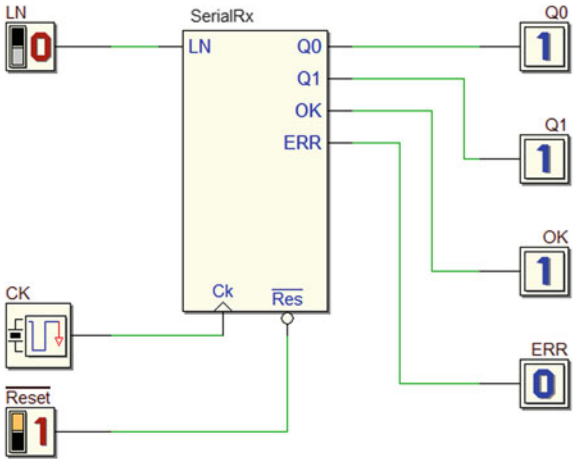
In the following, we will look at an example of this structure: a *2-bit serial receiver*, a version of the one based on the FSM alone that we have seen previously on p. 303.

The introduction of the *datapath* will simplify the FSM and make the system structure more easily *scalable*. In fact, in order to manage a serial signal containing *8 data bits*, a system based only on FSM would need at least *256 states*. Adding a *datapath* made up of as many flip-flops as serial bits would make the FSM only moderately more complex, when increasing the number of bits. In sum, the second approach raises the *complexity of the design linearly* along with the number of bits to manage, while the first approach raises it *exponentially*.

In further implementations of the serial receiver, we will show how an adequate *datapath* will make it possible to design a controller whose complexity is *independent* of the number of bits.

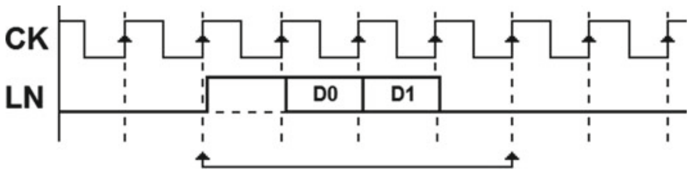
8.2.1 2-Bit Serial Receiver

Let's briefly summarize the specifications. We must design a *2-bit synchronous serial receiver*. The device must receive serial sequences on line *LN* and generate outputs *Q0*, *Q1*, *OK*, and *ERR*, as shown in the figure below. There, we reference the previous version with the FSM only. Note that in this figure the network's inputs and outputs appear as *active components*, as represented by *Deeds* during the simulation *by animation*.



The format of the sequence received is summarized here verbally and also in the timing diagram below:

- The *bit time* is the clock period *CK*.
- The sequence begins with a *start bit* at 1.
- The sequence continues with the *two* data bits *D0* and *D1* (in order).
- The sequence ends with a *stop bit* at 0.

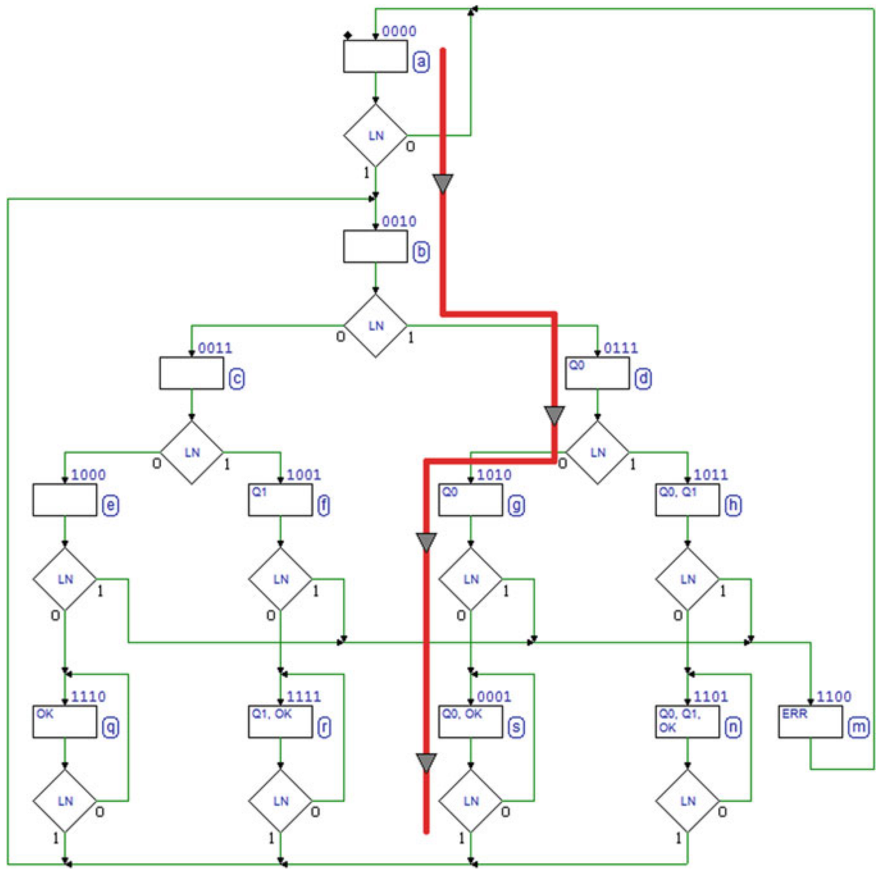


Notice that signal *LN* is synchronous with the clock *CK*, which is why we represent it with an approximate propagation delay after the clock rising edge. The system performs the following operations:

- Upon receiving a sequence, it makes the two data bits *D0* and *D1* available on outputs *Q0* and *Q1*.
- It checks that the *stop bit* is correctly at 0, and if so, it activates output *OK* and maintains it active until the next sequence arrives. If the stop bit is at 1, the receiver

only activates output *ERR* (Error) for the duration of one clock cycle and awaits a new sequence.

The figure shows the ASM of the previous version based on the FSM alone:

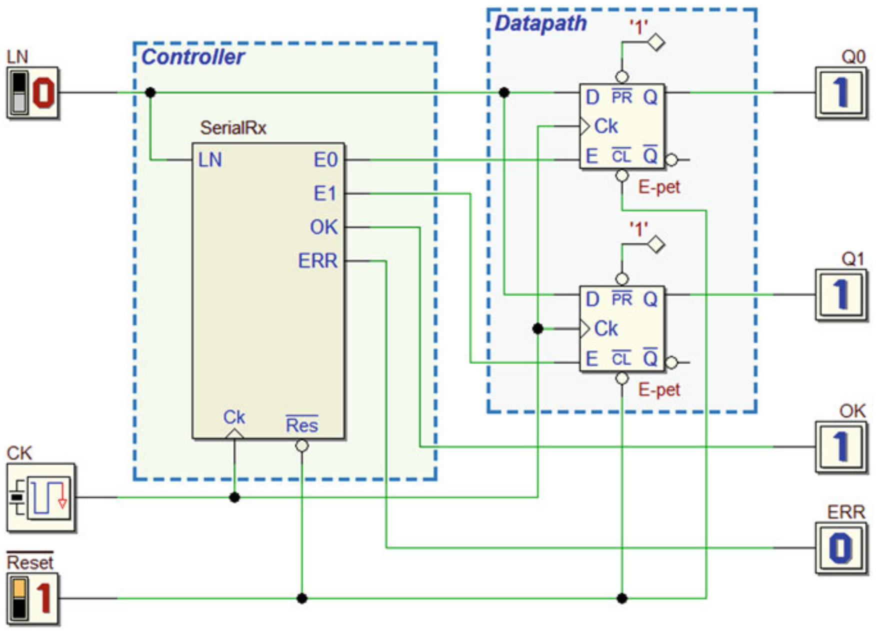


The path highlighted in red reminds us that in order for the FSM to *remember* which bits it received, it must *separate the paths*, since it cannot memorize the data in any other way. In the example, the FSM received a sequence where $D0 = 1$ and $D1 = 0$, followed by a regular stop bit. To keep the outputs active (here $Q0 = 1$ and $Q1 = 0$) until the next sequence, there must be a loop around the state that activates the outputs.

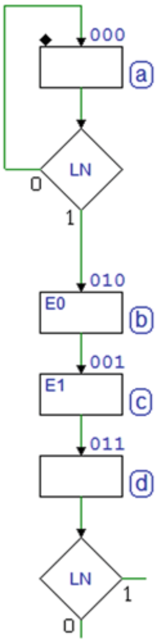
This requires a number of states that depend on all the possible data configurations. It is clear that extending the number of bits raises the complexity of the FSM (13 states in this example, which go up to 25 if the serial signal contains three bits and to 49 with four bits; 97 with five bits, etc.).

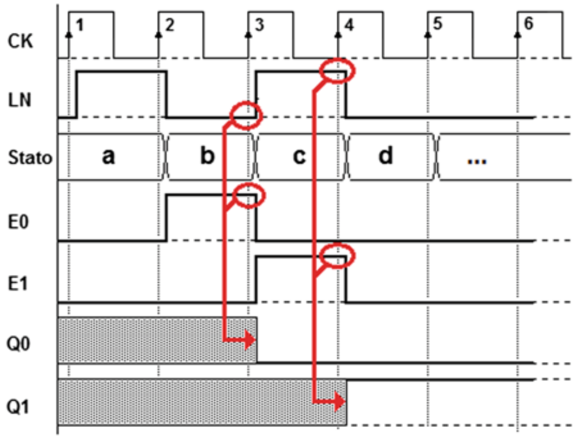
Thus, using this design structure is impractical for real-life cases (a typical serial signal contains eight bits).

The situation changes radically when we introduce a *datapath* that can memorize data, consisting in two E-PET flip-flops.



Notice the initialization input $\overline{Reset}$, which acts both on the *controller*, forcing the FSM into the *reset state*, and on the sequential components of the *datapath*. We will find this type of connection in all the upcoming designs excluding specific cases, which will be specially marked. Let's begin to draw the ASM diagram (see the figure below). The FSM waits for the *start bit* in state (a), as in the previous approach.





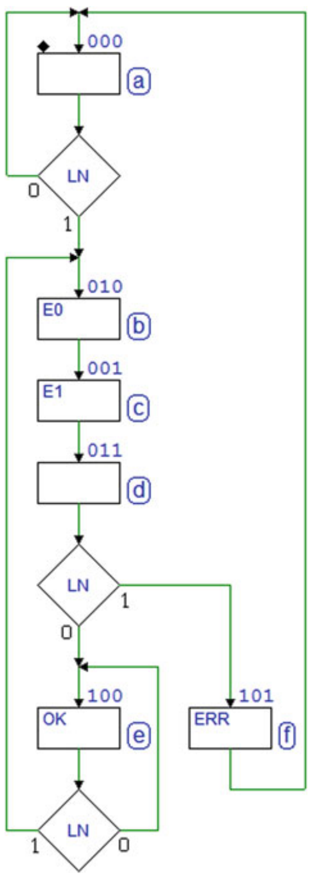
Still, as soon as the *start bit* is identified (edge 2 of the clock, see the figure above) rather than continuing to check the line cycle by cycle, this version *delegates* the job of memorizing the bits of the sequence to the flip-flops.

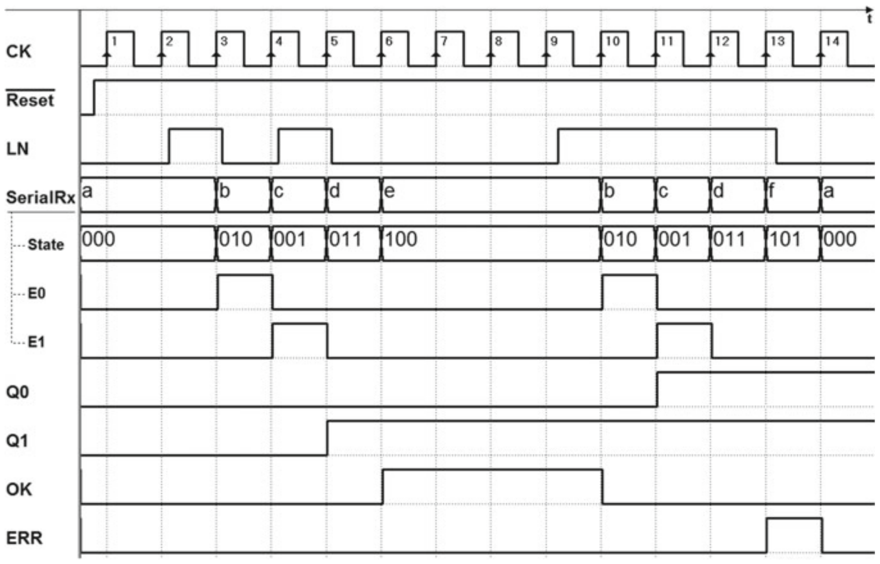
Input D of the flip-flops is actually connected directly to line LN.

The FSM only needs to *enable* the two flip-flops by activating lines $E0$ and $E1$ *at the right time*, that is in states (b) and (c), as suggested in the timing diagram seen in the previous figure. The highlighting shows that the value of the line is copied onto the corresponding flip-flop on the active edge of the clock in presence of enable $E0$ or $E1$. It is assumed that their previously memorized value is unknown and that the sequence received is $D0 = 0$ and $D1 = 1$.

In the clock cycle between edges 4 and 5, while the FSM is in state (d), the serial sequence presents the *stop bit*. Its value determines whether to generate *OK* (until the next sequence) or *ERR* for one cycle and then returning to await another sequence (see the complete ASM diagram on the right).

In the timing simulation below, the first sequence is correct, while the second presents an erroneous stop bit (due to line disturbances) causing ERR to be activated.



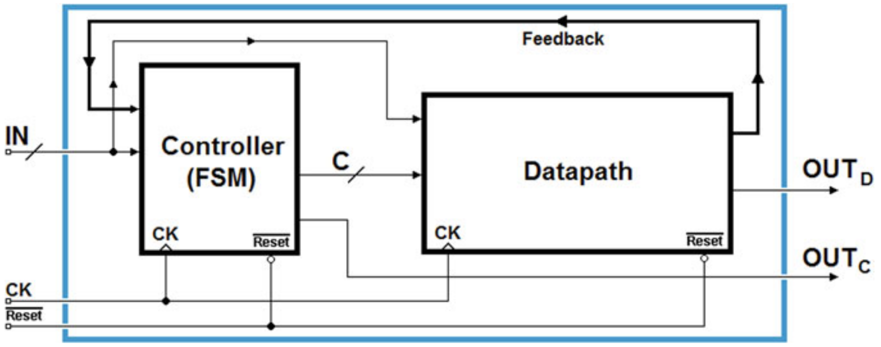


To extend the receiver to more bits, we only need to introduce more states like (b) and (c) into the ASM diagram and, obviously, change the *datapath* by adding the necessary flip-flops.

8.3 Feedback Control Systems

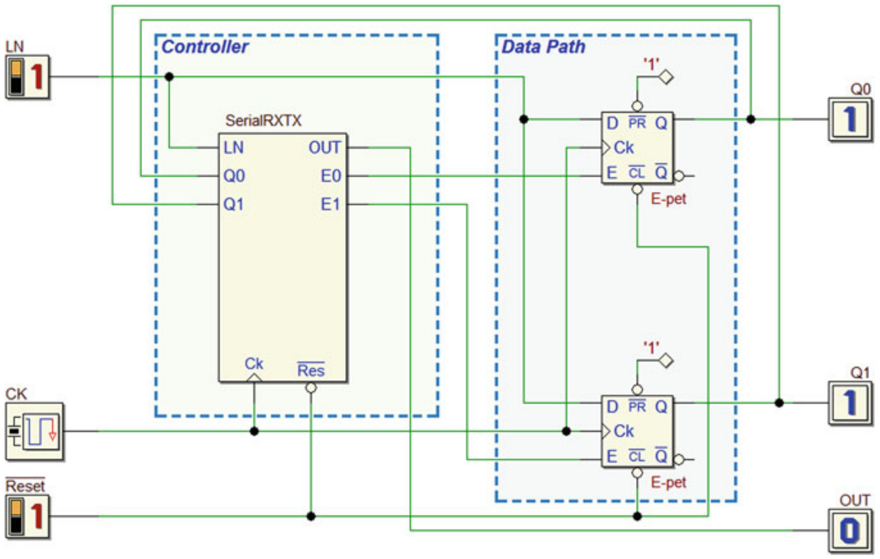
FSM generally needs to have *feedback* information from the *datapath*.

The *feedback* connection that carries this information as FSM inputs greatly increases the system's possibilities.



8.3.1 2-Bit Serial Receiver and Transmitter

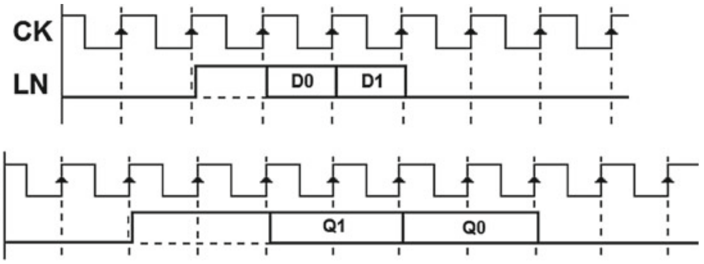
This design is an example of a system in which the FSM acquires signals generated by the *datapath* as inputs and uses them to re-transmit the serial data in another format.

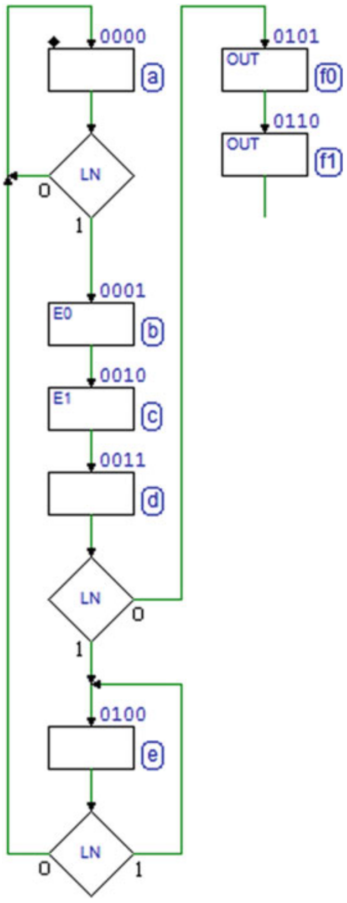


At first glance, the schematic resembles the previous example in that it uses two E-PET flip-flops to store serial data received through the *LN* line. This system, however, has an output *OUT* that re-transmits the serial signal in a different format.

The connection between flip-flops' outputs $Q0$ and $Q1$, and the FSM inputs with the same name allows the FSM to know the values of the bits received, which is necessary for re-transmission.

In *idle state*, lines *LN* and *OUT* are at 0. The device waits for a bit packet on line *LN* in the known format (*start bit*, *D*0, *D*1, *stop bit*).





When it is done receiving, the device transmits a similar sequence on OUT (*start bit*, $Q1$, $Q0$, *stop bit*) but with a bit time that is two times (two clock cycles) the one of the sequence received.

As we see in the figure above, this operation halves the *bit rate* (the number of bits transmitted in a second).

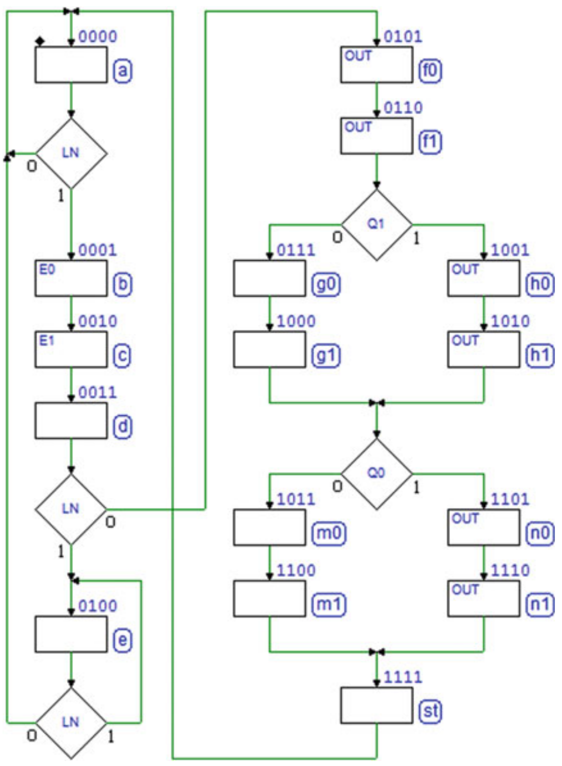
The transmission on OUT should only occur if the sequence received on *LN* is correct (the *stop bit* must be at 0). If it is not, the device generates no output on OUT, but rather *checks LN* to go back to zero before awaiting a new sequence.

Let's draw the first part of the ASM diagram (see the figure aside). The receiver saves the data on the flip-flops, as we have already seen in the previous example.

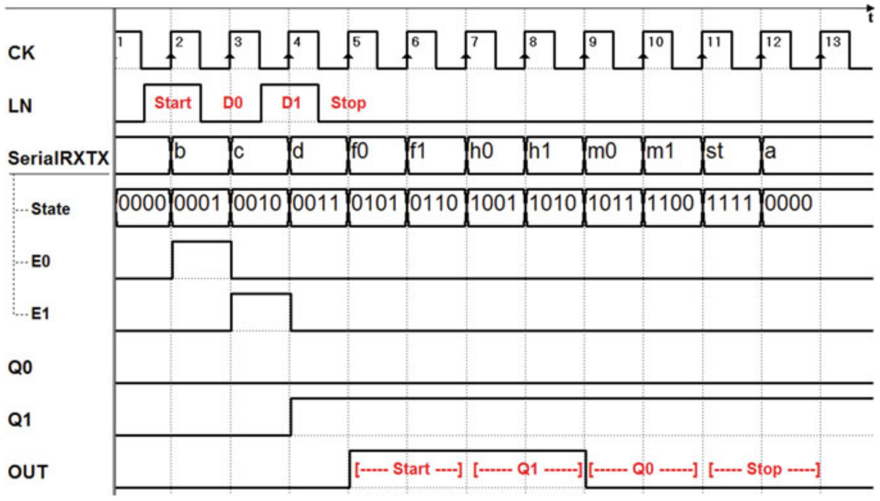
Notice that the FSM waits in state (e) until the line goes back to 0, if the *stop bit* is wrong. If, however, it is right as expected, the FSM begins to generate the serial signal on output OUT.

The OUT activation in the consecutive states (f0) and (f1) means that the *start bit* is transmitted on OUT, with the duration of two clock cycles.

To generate the serial sequence as defined, the FSM needs to know the values of the serial data received, now contained in the flip-flops. Thus, the FSM re-reads outputs $Q1$ and $Q0$ in order and, based on their value, generates pairs of states with or without the activation of the output OUT. Notice the inverted order of transmission as required (before $Q1$, then $Q0$). Finally, the FSM generates the *stop bit* in state (st). Since state (a), which has no output, occurs after (st), there is no need to duplicate (st).

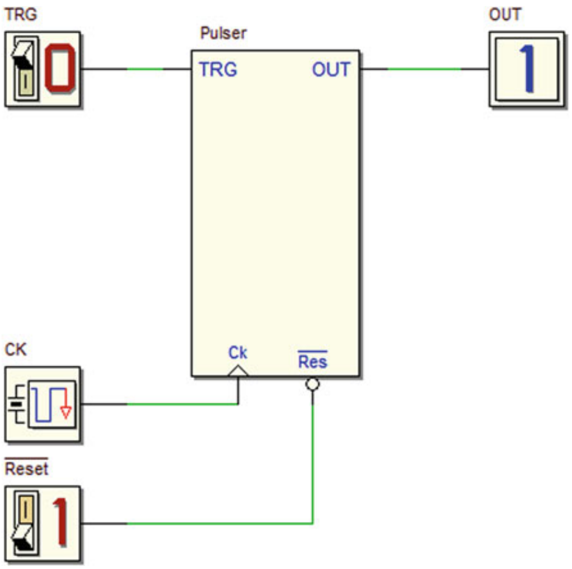


In the following timing diagram, the sequence received 1 – 0 – 1 – 0 produces sequence 11 – 11 – 00 – 00 in the output.

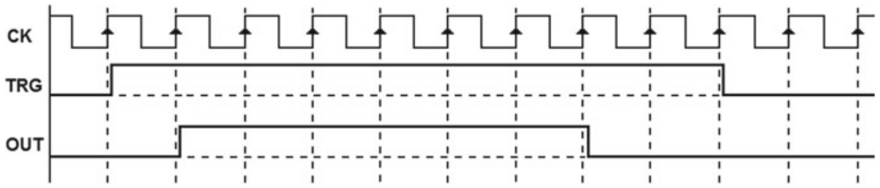


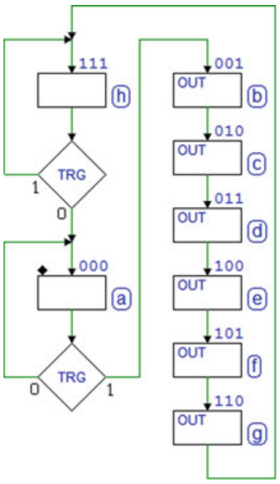
8.3.2 Pulse Generator

This design is another example of a system made in two different ways: using only the FSM and with the *controller - datapath* structure.
On detection of 0 → 1 transition on the input *TRG*, the *pulse generator* produces a *pulse* on output *OUT* (a high signal with a duration that is multiple of the clock cycle).
In this first version, we use only the FSM (see the figure aside).



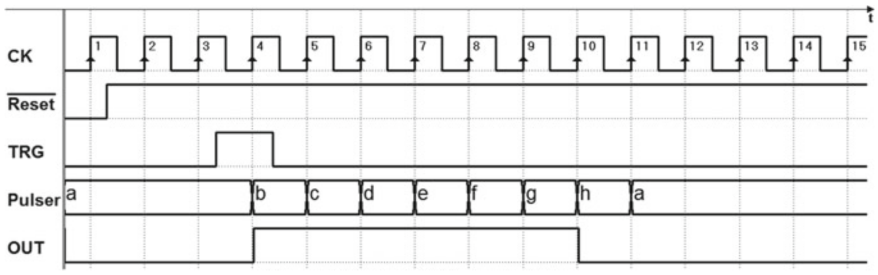
The pulse duration is fixed and lasts only six clock cycles, as seen in the timing track below, which shows an example of *TRG* command activation, too.



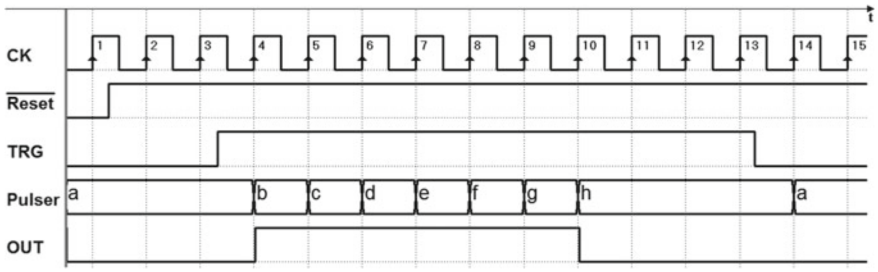


The ASM diagram does not pose any particular difficulty. In the *reset state* (a), the FSM waits for *TRG* to go to 1.
When this occurs, the FSM goes to state (b) and the following ones, generating $OUT = 1$. The pulse duration is equal to the number of consecutive states where *OUT* is active.
When pulse generation is finished, in (h) we wait for *TRG* to go back to 0.
The *wait state* (h) is necessary. If it were skipped, and if *TRG* were still at 1 when it went back to state (a), the FSM would immediately generate a new pulse.
The specifications, however, require this to happen on the *rising edge* of *TRG*.

Finally, this is the timing diagram produced by the simulator when input *TRG* duration is short.

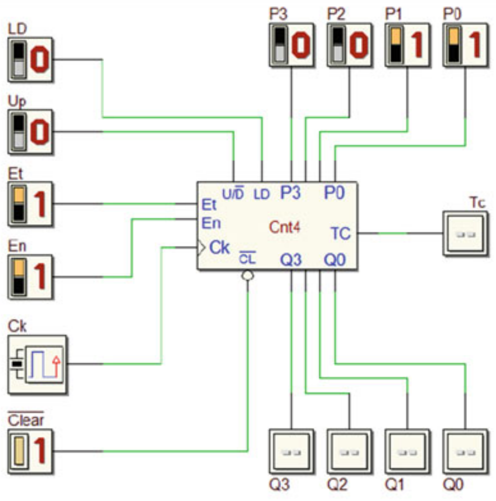


Below, with *TRG* very long, the FSM will wait until *TRG* it goes back to 0.



Let us now look at the second version based on the *controller–datapath* structure. We add to the system a “Cnt4” counter (see below).

Counter “Cnt4” (from the *Deeds* library)



When input $\overline{CL}$ is at 0, it *asynchronously* forces outputs $Q3..Q0$ to zero.

If $\overline{CL}$ is inactive, the other *synchronous* inputs control the counter: LD (*Load*), En (*Enable*), Et (*EnableTc*) and $U/\overline{D}$ (*Up/Down*).

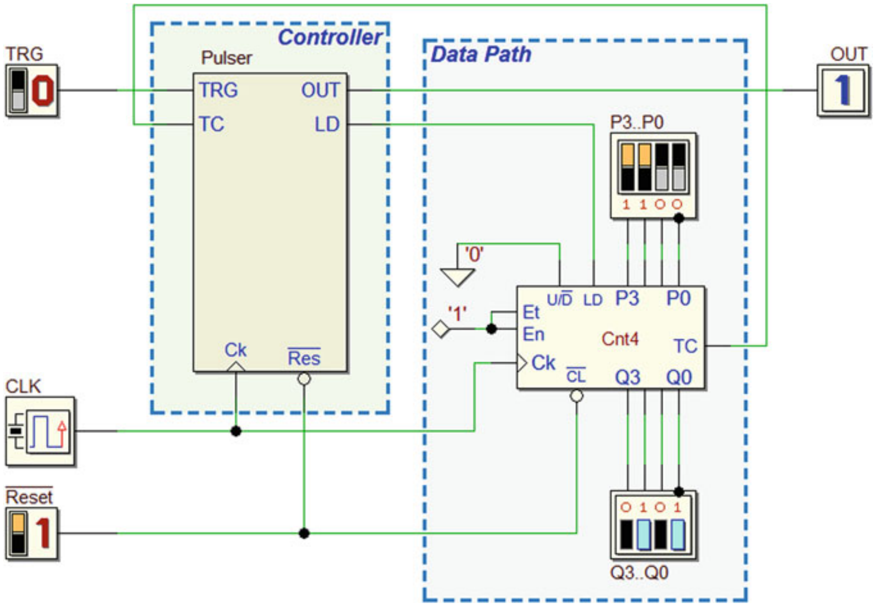
On the rising edge of the clock CK, the active-high input LD loads inputs $P3..P0$ onto outputs $Q3..Q0$.

When LD is not active, inputs En and Et enable the count on the rising edge of CK, if they are both at 1.

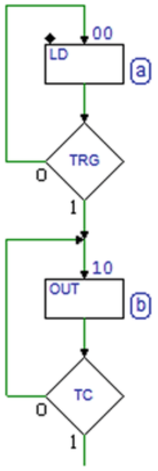
The count is *up* if $U/\overline{D} = 1$, otherwise it is *down*.
Output TC (*Terminal Count*) activates to 1 when $Et = 1$ and the counter's outputs reach the value 1111 (if the count is *up*) or the value 0000 (if the count is *down*).
TC always equals 0 if $Et = 0$.

Before continuing, it can be useful to get familiar with the behavior of the counter "Cnt4" by simulating a test circuit as the one of previous figure, available also in the *digital contents* of the book. We suggest to check the operation modes of up/down count, enable, load, and clear.

As we can see in the figure below, the *datapath* is made up of a *counter* that keeps trace of the number of clock cycles where *OUT* must remain active. This extension allows us to implement more flexible specifications, like the ability to program pulse duration.

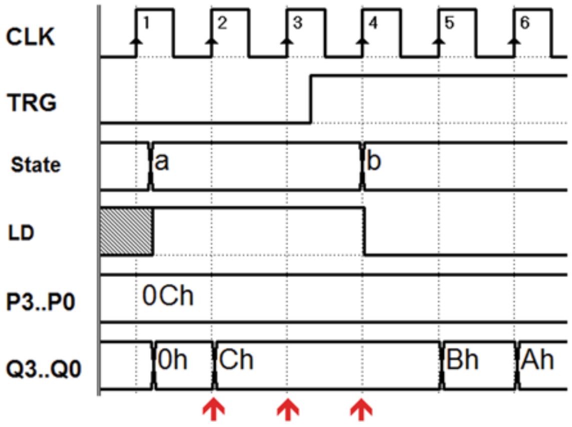


Counter is set to count *down* ($U/\overline{D}$=0) and always *enabled* (*EN*=*ET*=1). The FSM defines the start number of the count controlling the line *LD* and checks its end by reading *TC*. In the ASM diagram (see figure at right), we wait for command *TRG* in state (a).

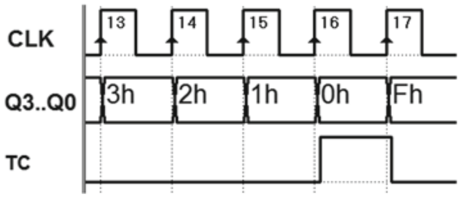


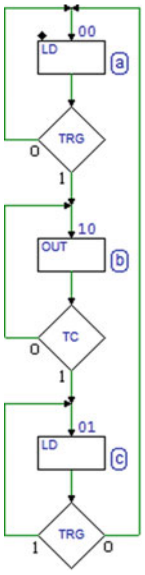
In the waiting state (a) we activate *LD* to force the counter to load the number set on the switches (lines *P*3..*P*0).
Note that activating *LD* in (a) we freeze the count, thus reducing also the network's energy consumption (a real circuit uses a certain amount of energy each time lines change levels).
Exiting state (a), the counter will start to count. However, given that command *LD* is synchronous, the counter does not start yet in the transition between (a) and (b), since *LD* is still active on that rising edge of the clock.

In the figure below, the red arrows indicate clock edges 2, 3, and 4. These are the instants when the number *P*3..*P*0 is loaded onto the counter, including edge 4, when the FSM moves to state (b). This means that the count will effectively begin only on edge 5.



State (b) repeats until the counter signals the end of the count. At each edge of the clock, the counter decreases its value until it *gets to zero* and activates *TC* as in the timing diagram below:





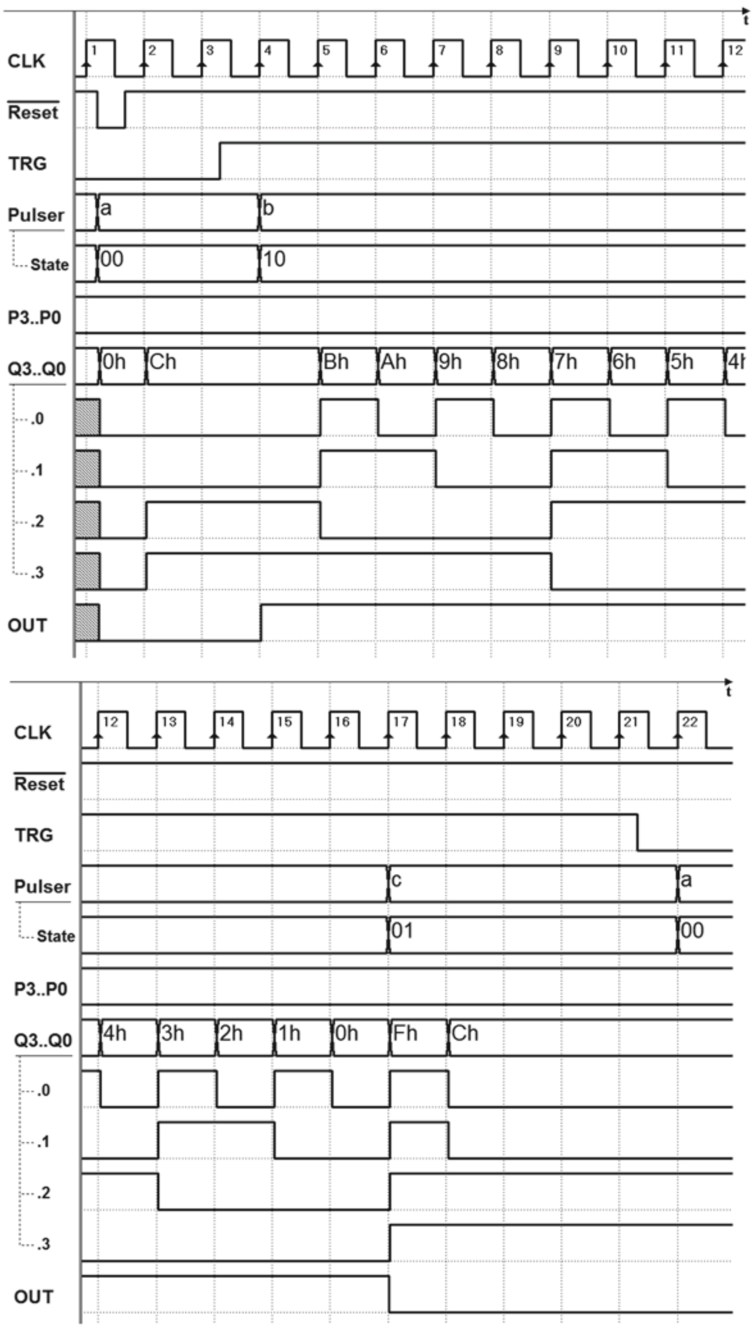
As a result, the FSM leaves state (b). Notice a detail that in this case poses no problem: edge 17 of the clock allows the FSM to leave state (b) but since the counter is enabled, the down count continues (from 0000 to 1111).

We finish the ASM diagram by adding state (c) where output *OUT* is no longer active and waits for *TRG* to go back to zero if, it is still at 1, as in the previous version of the generator. In state (c), we also activate *LD* to freeze the count.

Finally, we carry out the complete system simulation in *Deeds*. An analysis of its behavior (see the timing diagram on the next page) will allow us to answer detailed questions such as the following:

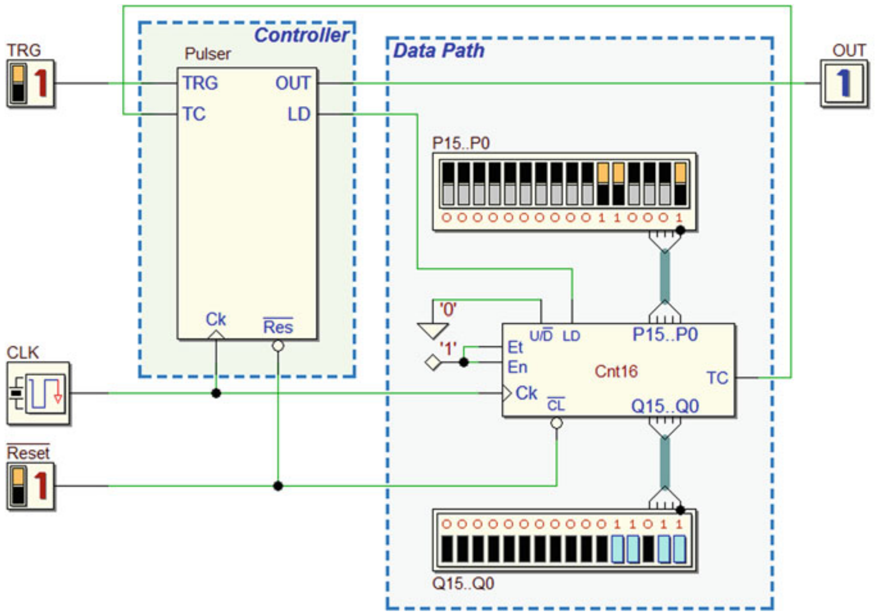
How many times state (b) has been repeated?

How many clock cycles have occurred since the counter was loaded?



As the diagram shows, OUT lasts 13 clock cycles, in response to a 12 set on counter inputs $P3..P0$ (the 0 is counted as well).

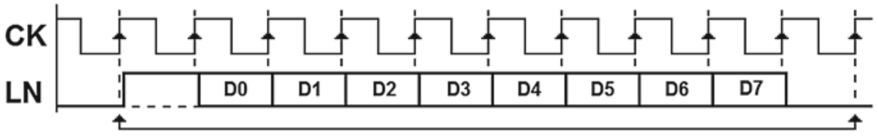
Finally, notice that the system can be easily modified to generate a longer pulse by simply using a counter with the right number of bits, without modifing the FSM. The system shown below uses a 16-bit counter (the "Cnt16" from the *Deeds* library):



This FSM is exactly the same, but the new circuit can generate pulses with durations of up to $2^{16} = 65536$ clock cycles.

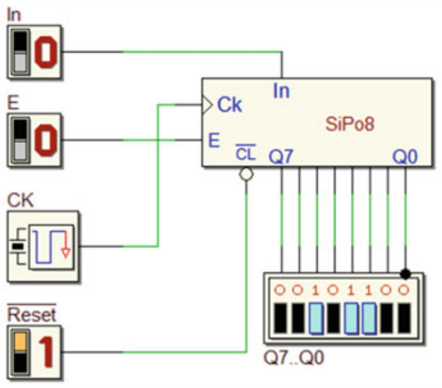
8.3.3 8-Bit Serial Receiver

The two projects here introduce two variations of the serial receiver we have seen before: to memorize bits, they use a *shift register* rather than flip-flops. The serial sequence has the same protocol as what was used for the 2-data-bit receiver, except that it contains eight, from D0 to D7. The specifications remain the same regarding the activation of *OK* or *ERR* when the sequence has been received.



Let's now look at the first of these two new versions of the receiver. The *datapath* is made up of only a shift register, the "SiPo8" component of the *Deeds* library (see the figure in the next page).

Shift register "SiPo8" (from the *Deeds* library)

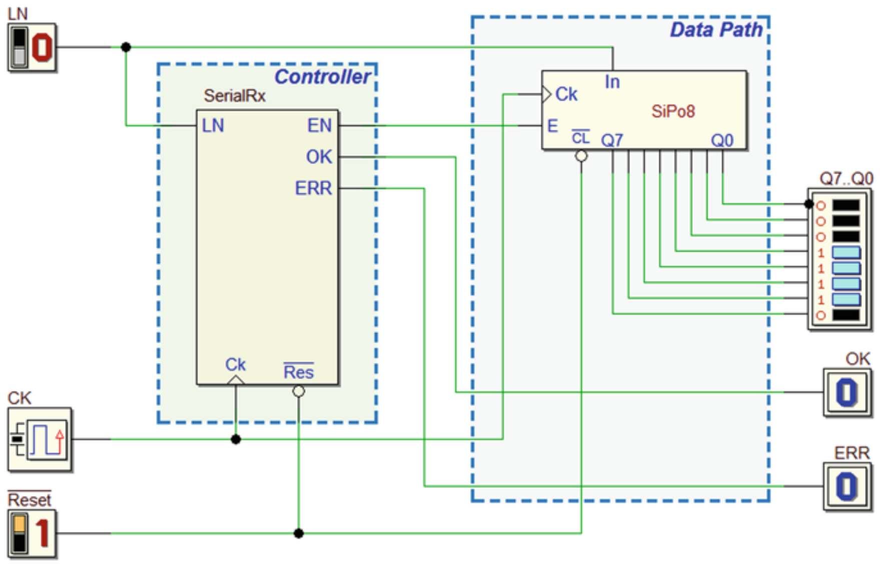


8-bit shift register with serial input In and parallel outputs $Q7..Q0$.

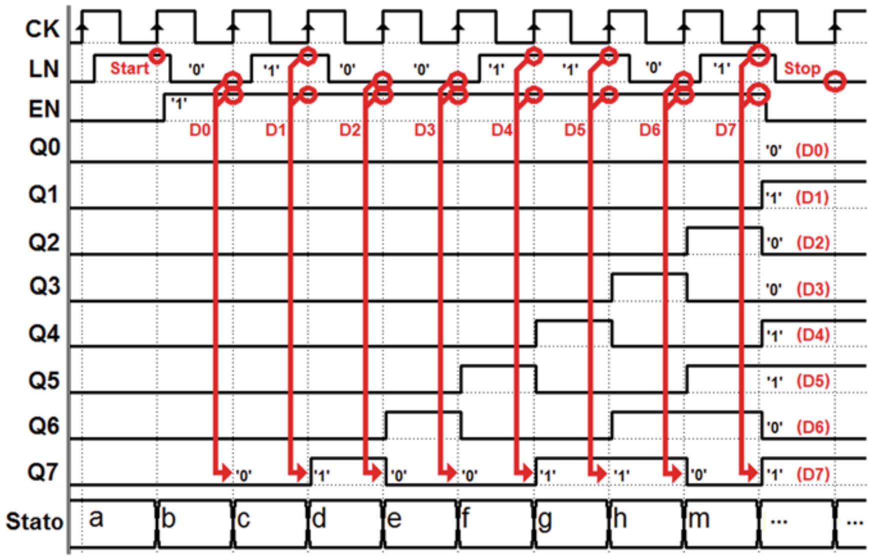
If input E (*enable*) is active, on the rising edge of the clock CK data moves one step in the direction $In \rightarrow Q7 \rightarrow .. \rightarrow Q0$.
If $E = 0$, the outputs remain unchanged.

The clear input $\overline{CL}$ is active-low and *asynchronous*. When at 0, it forces outputs $Q7..Q0$ to zero.

As we can see in the figure below, input *IN* of the *shift register* is directly driven by line *LN* with no mediation by the FSM. The shift enable *E* of the register is controlled by line *EN* of the FSM. Resetting the system initializes the FSM and the register.

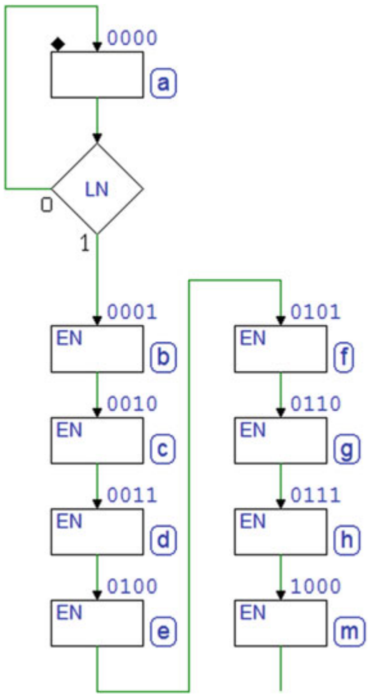


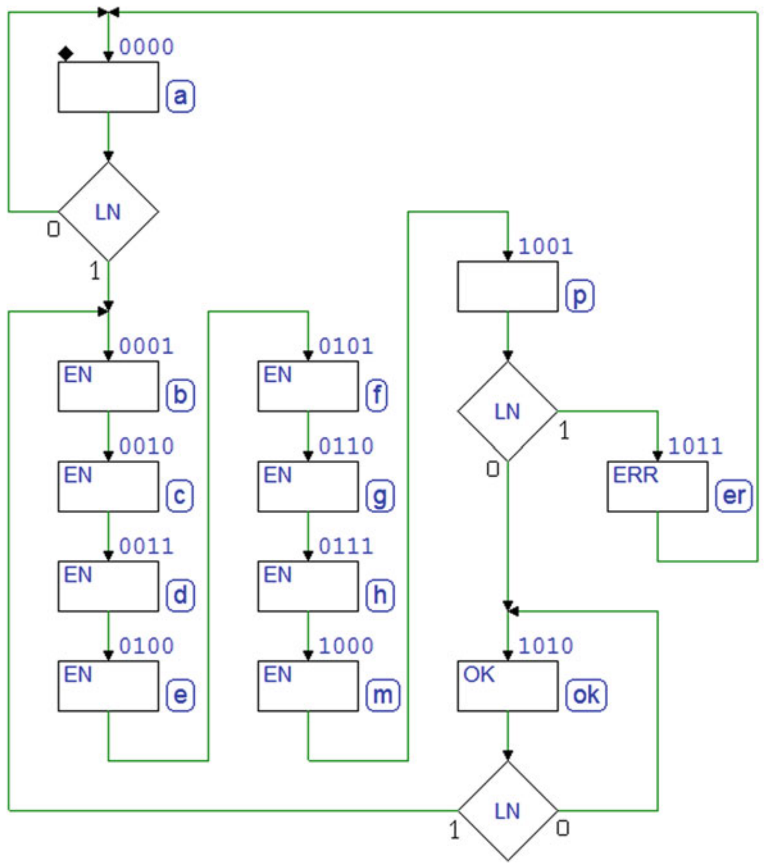
To correctly design the FSM, one must have a clear understanding of the timing of the signals in play. Before drawing the ASM diagram, it is often useful to make an outline of signal evolution over time. The figure below gives an example that assumes a serial sequence that transports bits $D0..D7 = 01001101$, in this order.



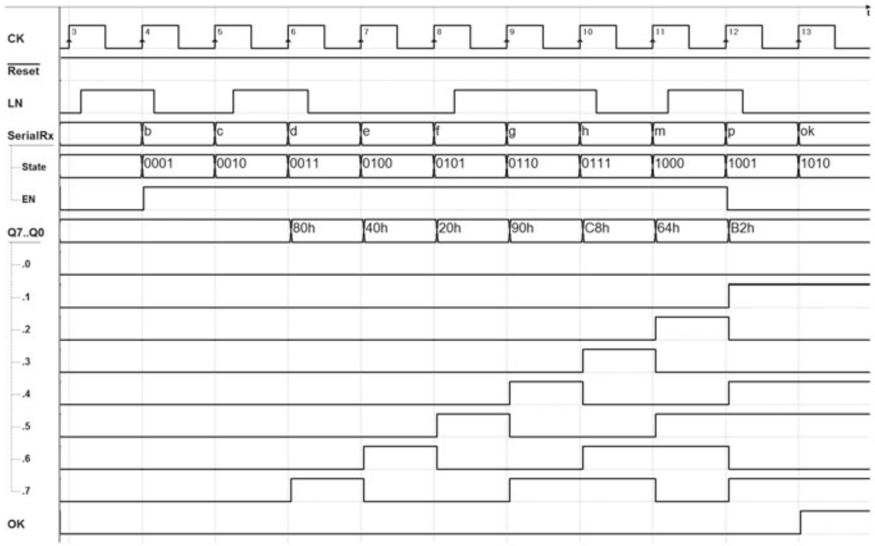
We must load the bits onto the shift register one by one.

As we see in the timing diagram above, we must activate *EN* as soon as the start bit is identified, and keep it active for eight clock cycles.
The bits enter into $Q7$ and they shift toward $Q0$, at every clock edge.
When we deactivate *EN* after the data bits on the line are terminated, all the received bits will be stored in the outputs $Q0..Q7$ of the register, available *in parallel*.
Thus, after waiting for the start bit in state (a), we will activate *EN* from the following state (b) (see the ASM diagram).
To keep it active for eight cycles, we must insert a total of eight states where *EN* is active (b)..(m).
Finally, we check the *stop bit* and generate *OK* or *ERR* according to specifications as in previous exercises, thus completing the ASM diagram (see the figure in the next page).



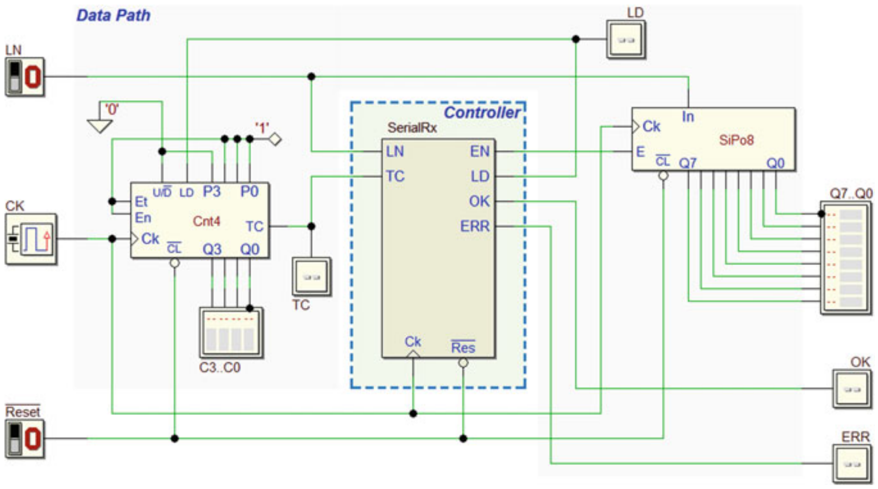


Now, with a complete timing simulation, we can verify that the system functions correctly.

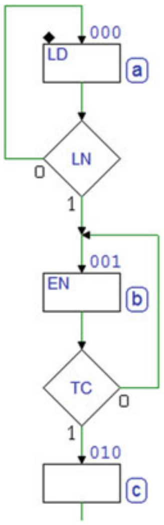


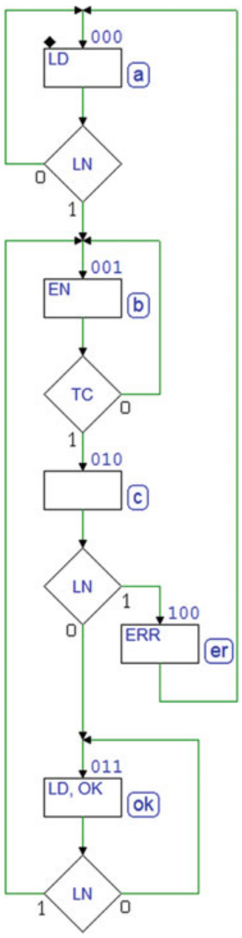
The version of receiver just examined uses an external device to memorize the serial data but still uses the FSM to count the *number of bits*. Obviously, it is possible to delegate this function to the *datapath* by using a counter set for the down count from 7 to 0.

The counter is a "Cnt4," used in a previous example (see the *Pulse Generator* on p. 372). The new system schematic now contains two elements in the *datapath*, a register and a counter, and optimizes task distribution between the *datapath* and the *controller*.



The beginning of the ASM diagram is the same as the previous version (see the figure at the right). In the idle state (a) the machine waits for the start bit in line *LN* and initializes and freeze the count through line *LD*.
When the start bit is detected, the FSM moves to state (b) where deactivates *LD* and activates register input *EN*, as in the previous case. Here, the difference is that it uses a single state rather than introducing as many states as there are bits to insert in the register.
The counter, in any case, is enabled ($En = Et = 1$). As soon as *LD* is deactivated, the count will decrease at each following rising edge of the clock, starting from 0111 (the value set on the inputs $P3..P0$, see the schematic).
In state (b) data bits are memorized one by one onto the register and counted by the counter, while the machine waits for *TC* to be activated.

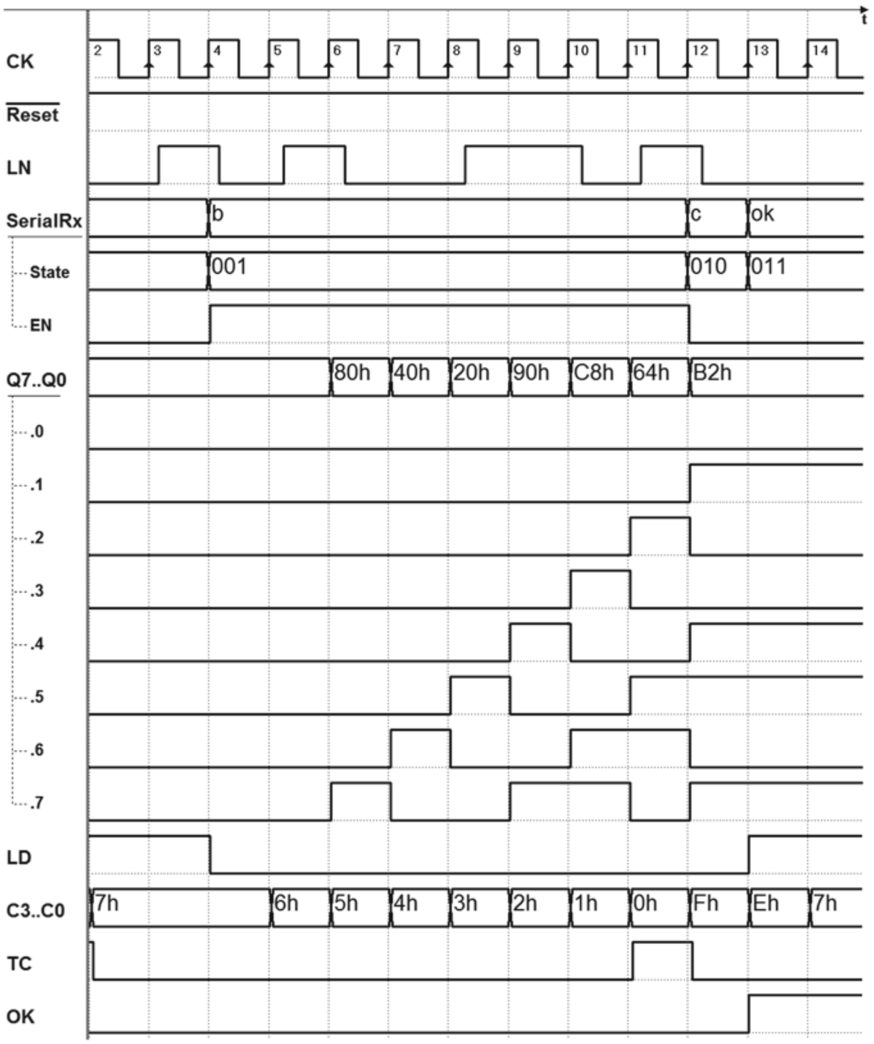




When the count gets to 0000, the counter activates *TC*, so the FSM leaves the cycle at state (b) and goes to (c) at the right time to check the stop bit of the sequence.
The state sequence that follows is almost identical to the one of the previous example, except that *LD* is activated in state (ok) as well as in (a).
Thus we have created a remarkably simple, easily analyzable FSM.
Notice that the *very same algorithm* can check serial sequences with a definable number of bits without changing the *datapath's* components. This is done by changing only the constant value applied to inputs $P3..P0$ of the counter.

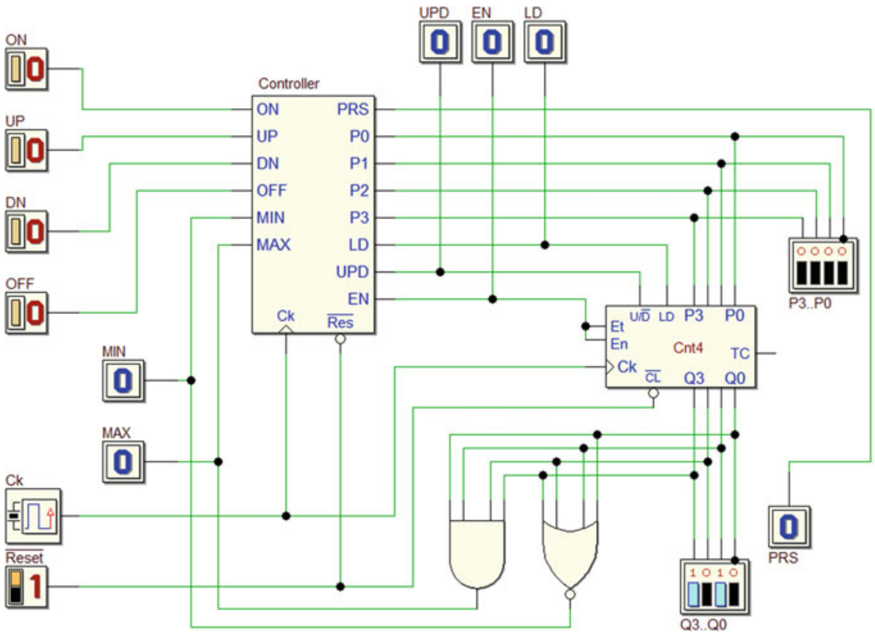
The timing simulation of the complete design now allows us to check every aspect of the relation between the evolution of states and the behavior of the register and the counter (see the figure below). Specifically, we can observe the following details.

- The counter decrements from 7 to 0, starting from clock edge 5 (that is *at the end* of the cycle where the FSM is in state (b) for the first time).
- The machine stays in state (b) until *TC* is read at 1 (on edge 12).
- The counter, after getting to zero (0000), continues to count cyclically (1111, 1110...) until the FSM activates *LD* in state (ok), so it is re-loaded to 0111 on edge 14.



8.3.4 Light Dimmer

The system in the figure is a *light dimmer* that controls the intensity of a light bulb by means of four switches (*ON*, *UP*, *DN* e *OFF*).



The circuit generates the output lines $Q3..Q0$ that encode 16 possible values of light intensity, starting from 0000 (fully off) to 1111 (fully on). Note that the light bulb will be driven by a device that converts the code value $Q3..Q0$ in an analog quantity, which is fed to the light bulb. This device is not part of our design, and it is not represented in the schematic.

The *switches* generate a *low* level in idle state and *high* while they are pressed. They should be considered *ideal*, that is having no electromechanical contact *bouncing*. *Pressing* and then *releasing* each switch determine the system's behavior, as follows.

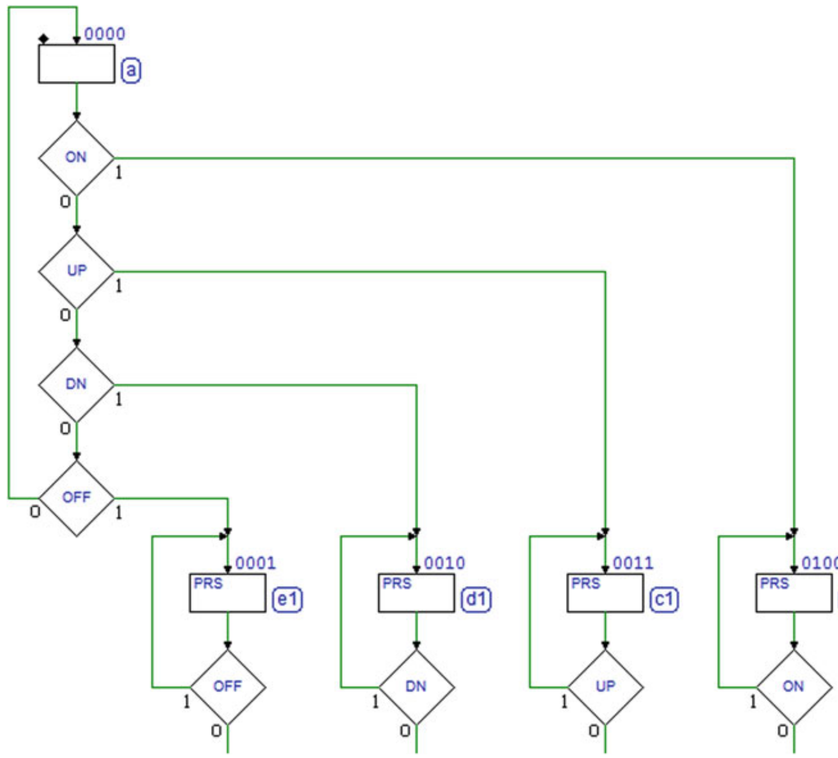
$ON \rightarrow$ *Turns on* the lamp to the maximum$(Q3..Q0 = 1111)$.
$UP \rightarrow$ *Increases* light intensity by one unit.
$DN \rightarrow$ *Decreases* light intensity by one unit.
$OFF \rightarrow$ *Turns off* the lamp$(Q3..Q0 = 0000)$.

Output *PRS* serves as a signaling device and is active while a switch is pressed. The output number $Q3..Q0$ must not be incremented if it has reached the *maximum* value or decremented if it has reached the *minimum* value.

This system uses the counter "Cnt4" more completely than in previous examples (see *Pulse Generator* on p. 372 or *8-bit Serial Receiver* on p. 377), in that here it must be *loaded*, *enabled*, and set for the *up or down* count.

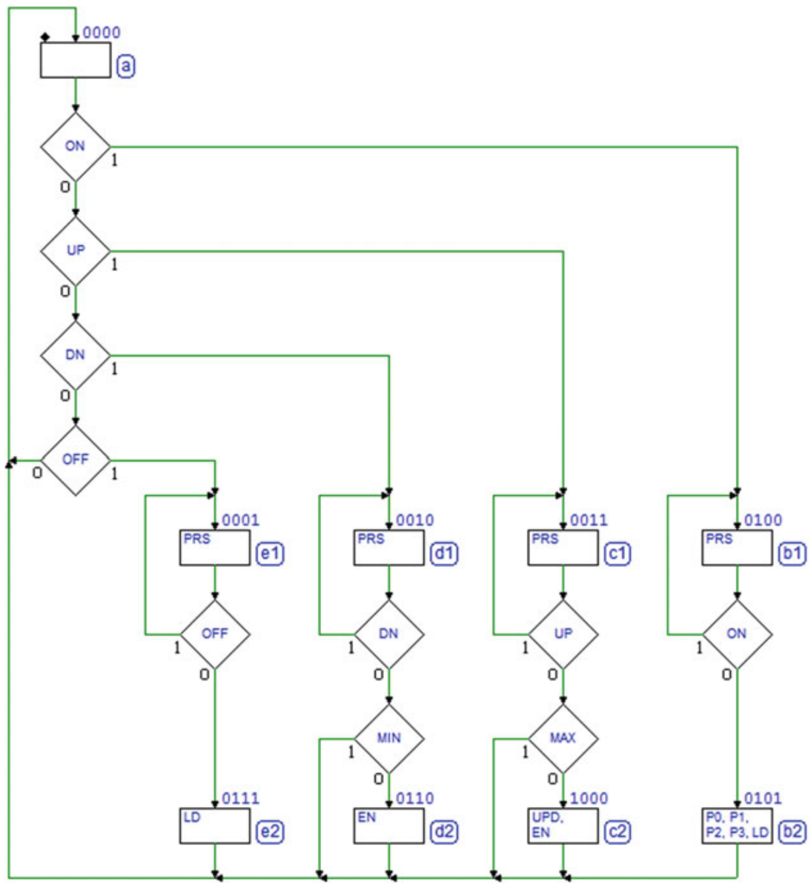
The two logical gates have the counter outputs $Q3..Q0$ as inputs: the AND activates *MAX* when the count gets to the maximum, while the NOR activates *MIN* when it gets to the minimum.

As for reading the switches, they should be given the same considerations as in the *Push-button Handling* example on p. 306. The structure of the first part of the ASM diagram is the same, the simultaneous control of the pressure on the push-buttons and the wait states that follow.



The diagram branches off into four separate paths corresponding to push-buttons *OFF*, *DN*, *UP*, and *ON*. Based on specifications, output *PRS* is active in states (e1), (d1), (c1), and (b1), where the FSM is when a push-button has been pressed and it is waiting for it to be released.

Let's fill in the four paths of the diagram keeping in mind how it must work when the push-buttons are released. For now, let's focus on the paths related to the release of *OFF* and *ON*. The figure below shows that in states (e2) and (b2), we send the counter the command to load the data present on counter inputs $P3..P0$ by activating *LD*.



To achieve this, the FSM sets $P3..P0 = 0000$ in (e2) while it assigns $P3..P0 = 1111$ in (b2). So, in (e2), the counter will be loaded to the minimum value (light off) and in (b2) to the maximum (light fully on).

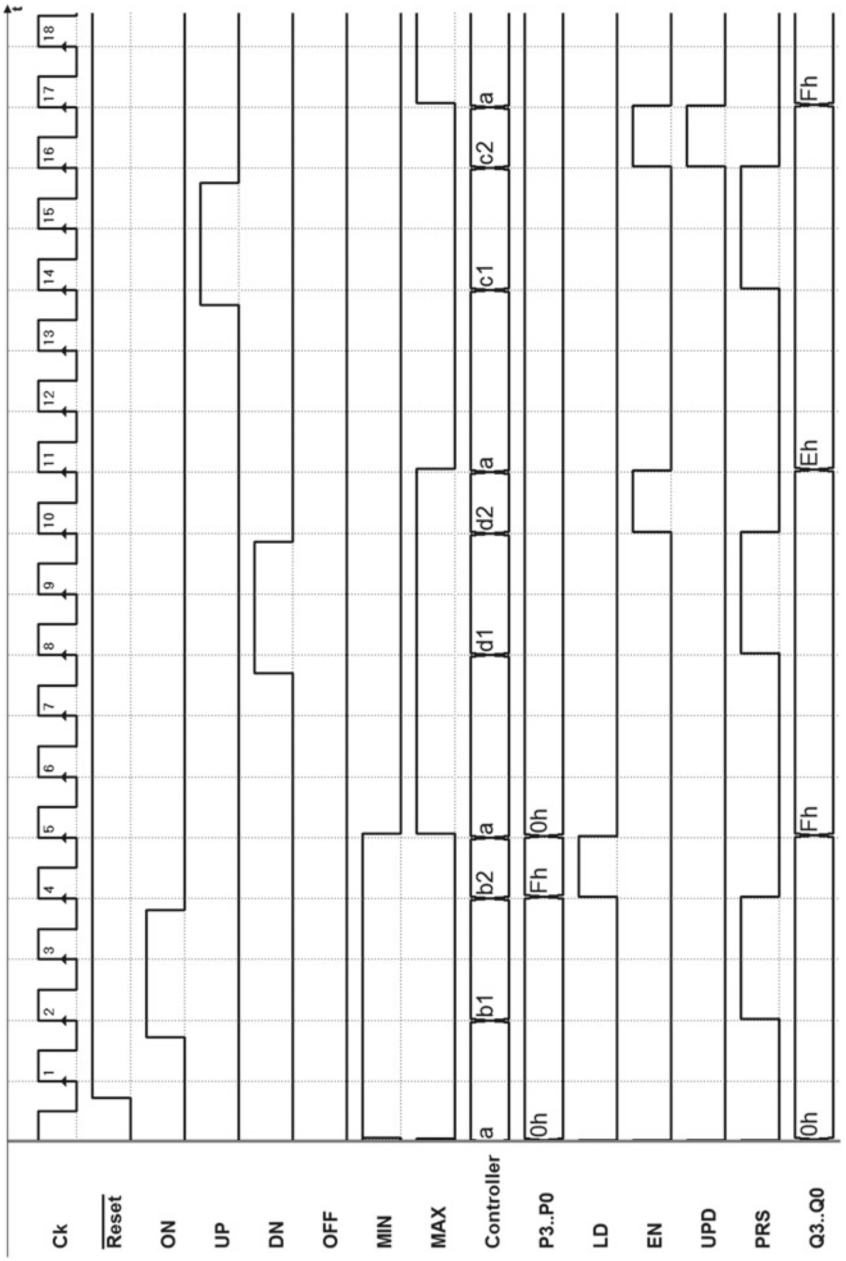
Notice that lines $P3..P0$ and *LD* are set at the same time in the same state. Given that *LD* is synchronous, the actual loading takes place at the edge of the clock that makes the FSM leave the state and return to (a).

Let’s look at the diagram in relation to the decrement and increment of the number (push-buttons *DN* and UP). The decrement of the counter takes place in state (d2) and the increment in state (c2), after the *MIN* and *MAX* check, because the number must not be changed if it is respectively at the *minimum* or *maximum* value.

In these states, *EN* must be activated to enable the count and UPD to determine its direction. In state (d2), UPD is set at 0 (*to count down*), while in (c2) it is set at 1 (*to count up*).

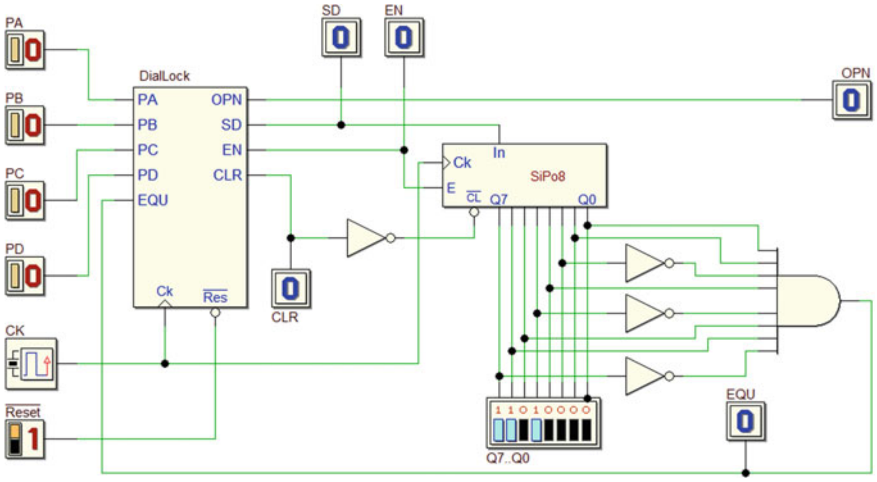
Note that the count occurs on the positive edge of the clock; therefore, the increment or decrement will take place *on exit* from states (d2) and (c2) where it is enabled, at the same time when the machine goes back to (a).

Below a timing simulation of the system. Note the loading, increments, and decrements discussed above.



8.3.5 Combination Lock

This system commands the opening of a door with an electric lock through a *combination* the user inserts. The lock is commanded by output *OPN* and opens when the input push-buttons *PA*, *PB*, *PC*, and *PD* are pressed in a specific sequence.



The *datapath* is made up of an “SiPo8” shift register (seen in *8-bit Serial Receiver* on p. 377) and a simple combinational logic that reads its outputs $Q7..Q0$ and activates the FSM’s input *EQU* when they take on the value 01101011_2 (the internal code of the *combination*).

Register inputs *In* and *E* are driven by FSM outputs *SD* (*serial data*) and *EN*, respectively. Notice that, unlike the previous cases, $\overline{Reset}$ is applied only on the FSM, while the register’s $\overline{CL}$ command is generated by the FSM’s output *CLR*.

When the user *presses* the push-button and then *releases* it, the system inserts a 2-bit code in the shift register.

$$
\begin{aligned}
PA &\rightarrow \text{Loads code } 00.\\
PB &\rightarrow \text{Loads code } 01 \text{ (first 1, then 0).}\\
PC &\rightarrow \text{Loads code } 10 \text{ (first 0, then 1).}\\
PD &\rightarrow \text{Loads code } 11.
\end{aligned}
$$

The sequence of push-buttons (the combination) we have chosen to open the lock is:

$$PD - PC - PC - PB\ .$$

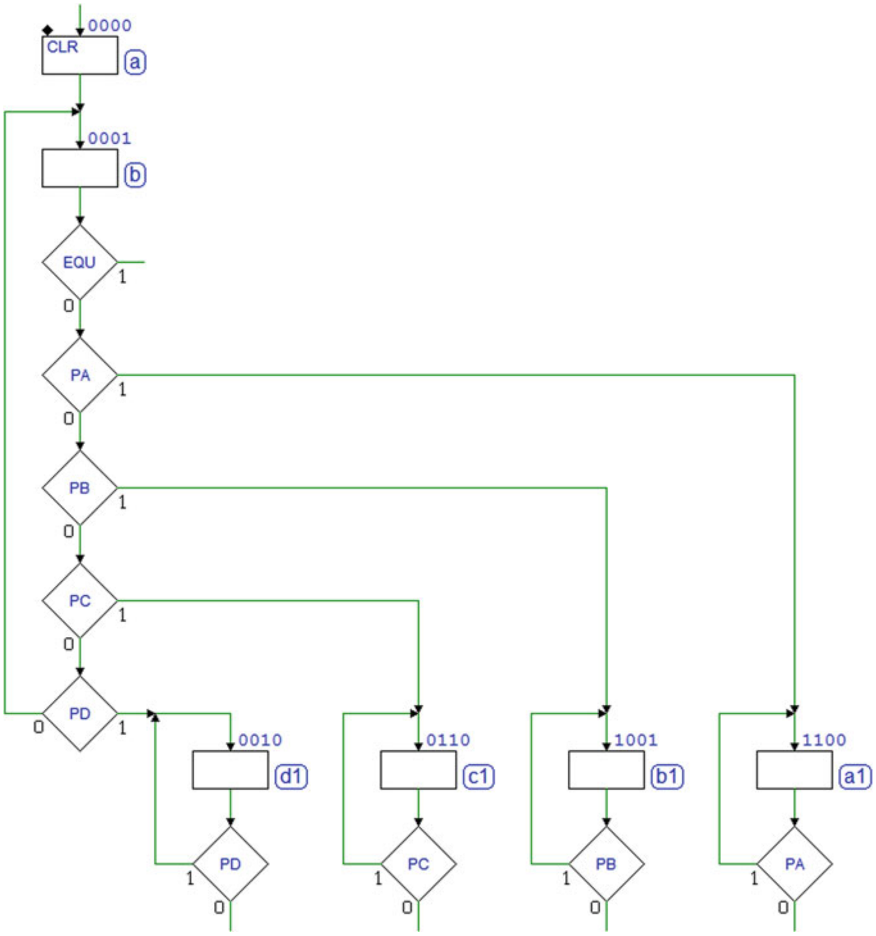
When the data in the register is equal to that produced by this input sequence (01101011_2), the FSM activates output *OPN* for one clock cycle and then clears the data in the register.

Once the lock is open, it will be mechanically locked again when the door is closed. We assume that:

- Pressing a push-button activates the corresponding line to 1 at the FSM input, while releasing it puts it back to 0.
- Two push-buttons are never pressed at the same time.
- Enough time passes between releasing a push-button and pressing the next one so that the system concludes its operations.

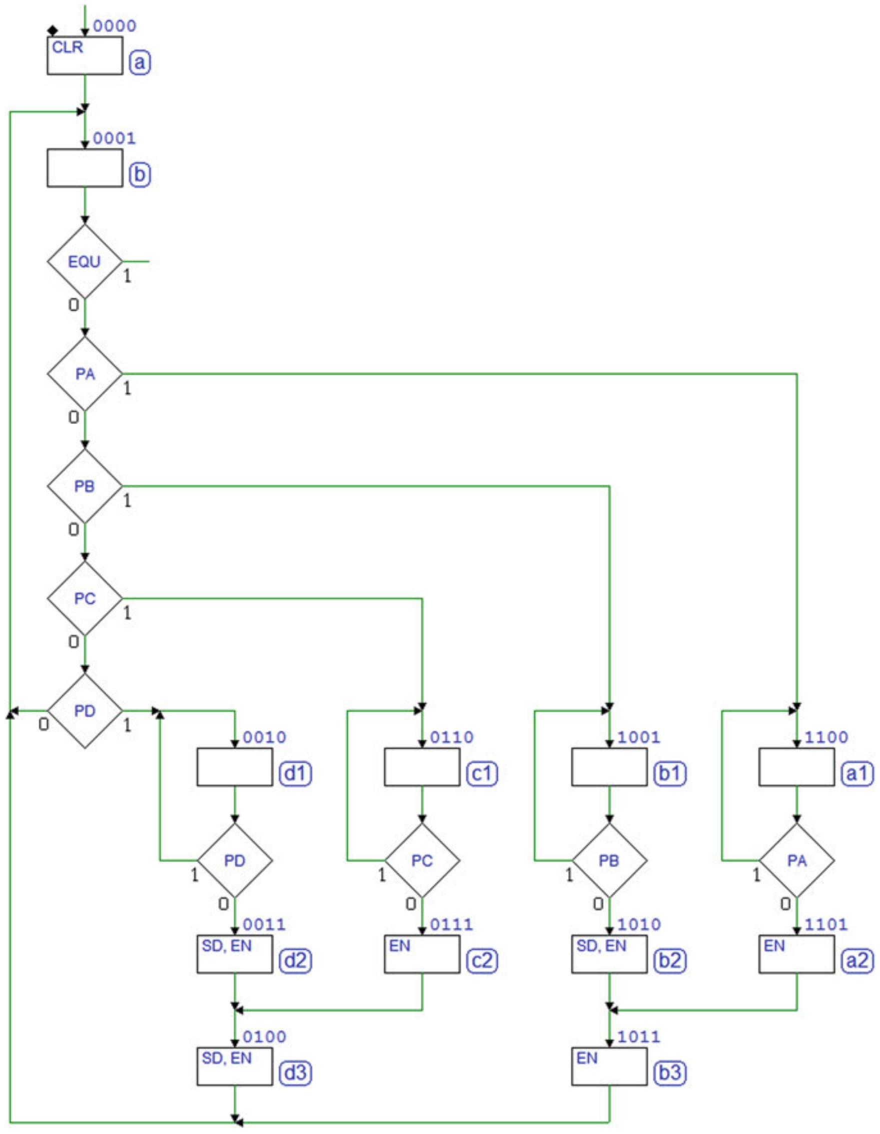
As we can see in the figure below, the section of the ASM diagram that handles the push-buttons was previously introduced (as in *Push-buttons Handling* on P. 306 and in the *Light Dimmer* on p. 384).

Notice that here, when the system is reset, the FSM goes to state (a), which *precedes* the push-button control cycle. In (a), *CLR* is active and clears the register (for reasons that will become clear later).



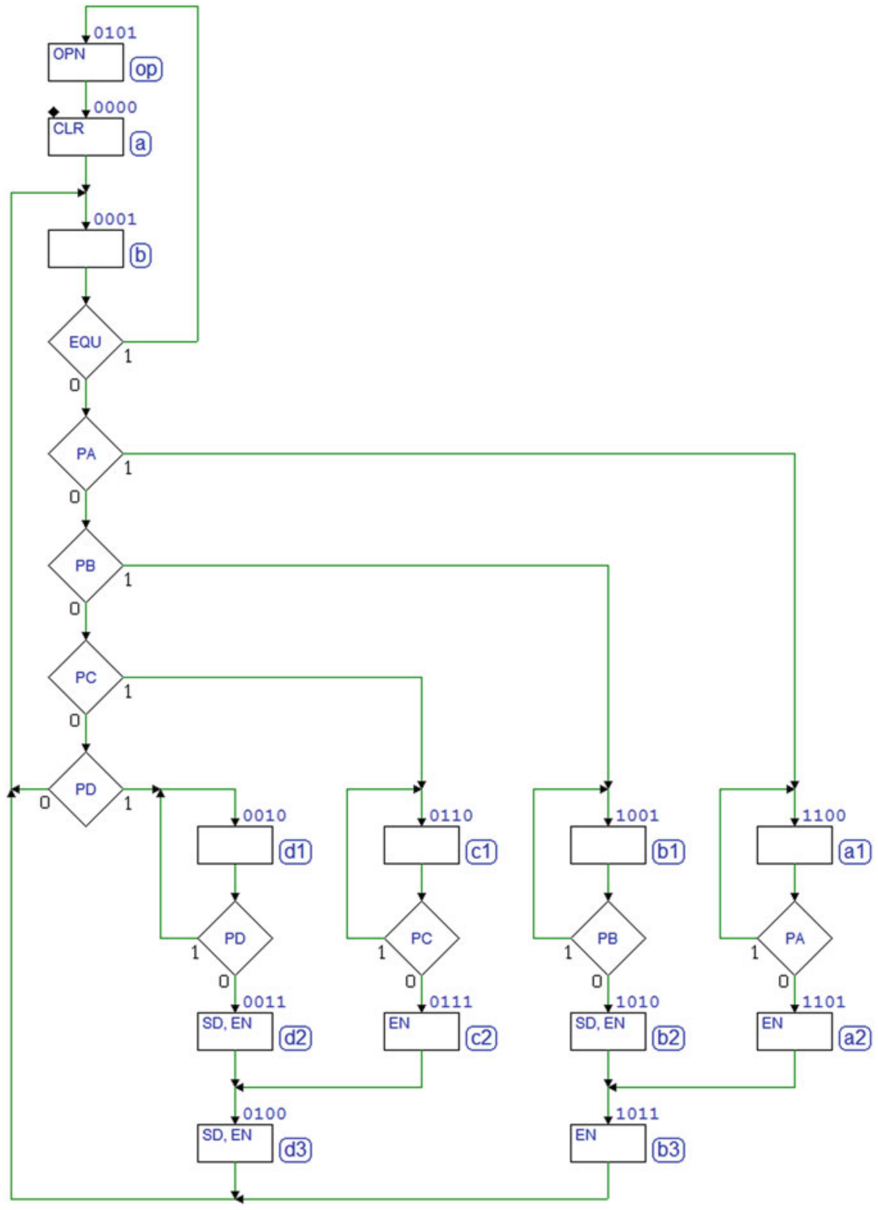
After the waiting for the release of the push-buttons, the states that follow will *load onto the register* the two bits corresponding to the pressed push-button.

The figure below shows an almost complete ASM diagram. States (d2) and (d3) have been inserted along the path related to the release of push-button *PD*. They load the pair of bits 11 onto the register by activating *EN* and *SD* for two clock cycles.



Likewise, we load a 0 onto the register in state (c2). Then, we take advantage of the existing state (d3) to load the next 1. We do the same in states (b2), (b3), and (a2). Here, we see another example of how we can use the same state many times along different paths.

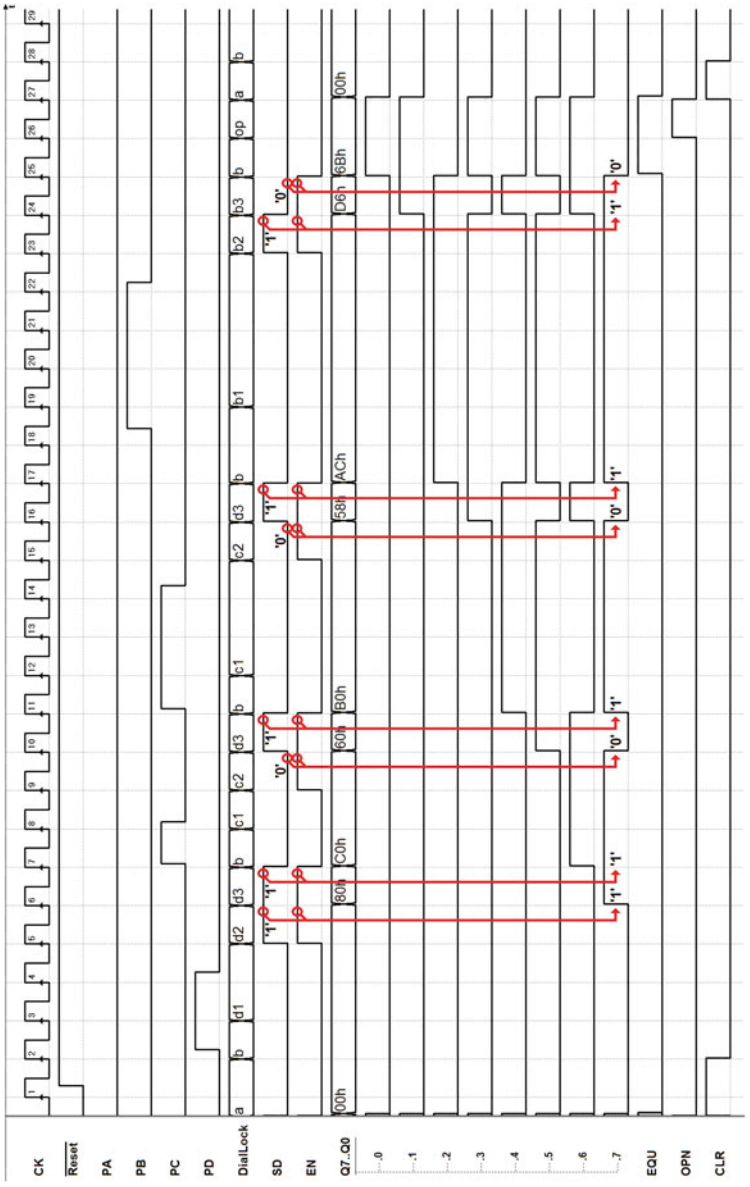
Now, let's finish the diagram according to the specifications regarding opening the lock (see below).



Notice that the check on *EQU* is inserted in the same cycle that checks the push-buttons; this is for simplicity's sake. When *EQU* is 1, the FSM moves to state (op) activating *OPN* for one cycle, as per specifications, and then goes to state (a), which clears the register. Without this step, once the right combination was set, *OPN* would stay active forever.

Finally, notice that (a) is also set as a *reset state*. This is because the register is not cleared directly by the system reset, as previously explained. Rather, it needs to be cleared before entering the loop waiting for the push-button.

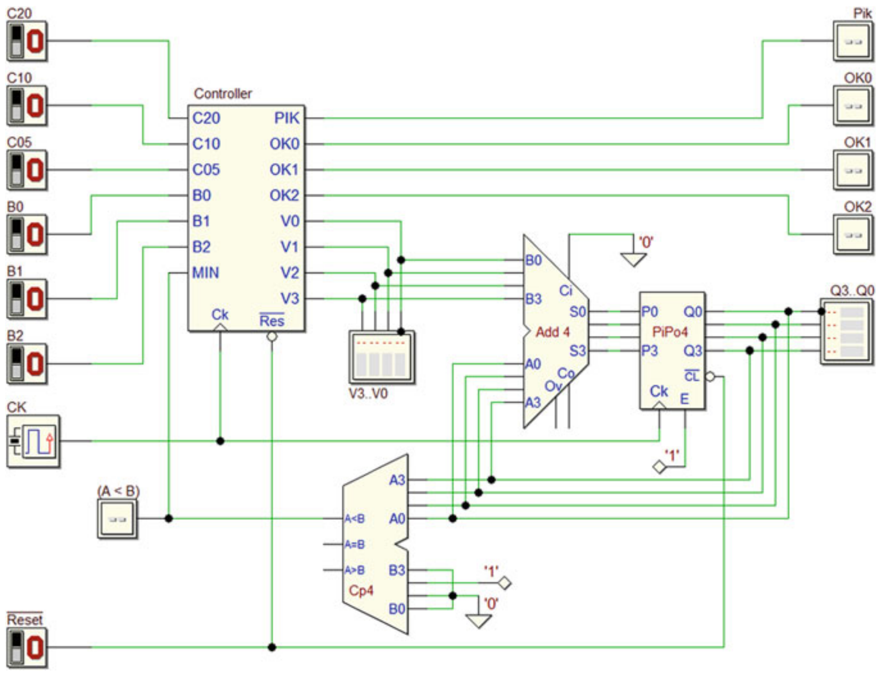
The timing simulation below highlights the FSM outputs *SD* and *EN* in relation to the values loaded on the shift register.



8.3.6 Automatic Drink Dispenser

We are designing a digital system to manage a (simplified) automatic drink dispenser. It is made up of a synchronous FSM plus a *datapath* made of a *4-bit arithmetic circuit* that includes an “Add4” *adder*, a “PiPo4” *parallel register*, and a “Cp4” *magnitude comparator*.

Both the FSM and the register are initialized by the system $\overline{Reset}$.



The dispenser offers *three types* of drinks that all cost *20 cents*. The dispenser waits for a user to insert the money to pay for the drink. Inputs $C05$, $C10$, and $C20$ show when a 5, 10, or 20 cent coin is inserted, respectively. When this happens, they produce a high *pulse* on the corresponding line for *one clock cycle* (when idle, the lines are low).

The system calculates the total value of the coins inserted and memorizes it in the “PiPo4” register. By convention, a unit of $Q3..Q0$ corresponds to *5 cents*: for example, $Q3..Q0 = 0100_2 = 4_{10}$ corresponds to $(4 \cdot 5) = 20$ cents.

The register’s input is connected to the output of the “Add4” adder. One of the adder’s inputs is connected to the register’s output, while the other is directly provided by the FSM through lines $V3..V0$. As a result, instant by instant, the adder generates the sum of the number on the register and the number provided by the FSM.

The FSM adds the *value* of the *inserted coin* to the register, presenting it to the adder for one clock cycle. Given that the register’s enable input E is always active, it will be necessary for the FSM to keep lines $V3..V0$ at zero for the rest of the time to keep the calculated total unchanged.

The comparator compares the total $Q3..Q0$ accumulated on the register with the price of the drink ($= 0100_2 = 4 = 20$ cents, established on inputs $B3..B0$ of the comparator). Of its three outputs $A < B$, $A = B$, $A > B$, only one $A < B$ is read by the FSM at input *MIN (“minor than”)*.

When *at least* 20 cents are inserted, the dispenser activates *PIK* to ask the user to choose the drink by pressing one of the three push-buttons $B0$, $B1$, or $B2$ (*PIK* lights up the push-buttons). Push-buttons $B0$, $B1$, and $B2$ are normally at the logical value 0 and go to 1 when they are pressed.

When one of the buttons is pressed, the FSM deactivates *PIK* and activates one of the outputs *OK0*, *OK1*, or *OK2* (corresponding to the drink the user picks) for one clock cycle, thus commanding the dispensing of the drink.

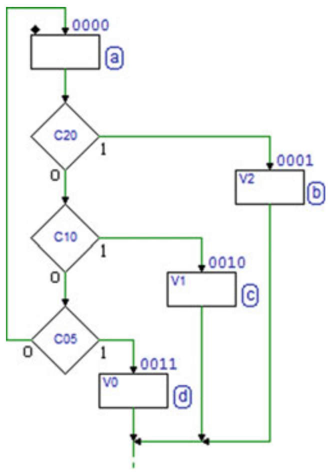
Finally, the dispenser *collects* the cost of the drink but *credits* the next user for any amount inserted *over* the 20 cent cost of the drink. To do this, the FSM must subtract 20 cents from the value stored in the register, and does so by generating *two’s complement* of the code of the value of the drink on lines $V3..V0$ for one clock cycle. After this operation, the register will contain the credit to be used by the next customer.

Further specification: assume all the input signals to be synchronous with the clock, that it is impossible to insert more than one coin at a time and that enough time passes between inserting one coin and the next that the system can function properly.

Let’s start drawing the ASM diagram (see aside). We wait in (a) for a coin to be inserted.

Depending on the coin’s value, it activates $V2$, $V1$ or $V0$ for one clock cycle. Thus, it provides the binary numbers 0100_2, 0010_2 or 0001_2 (corresponding to 20, 10 o 5 cents) to the adder.

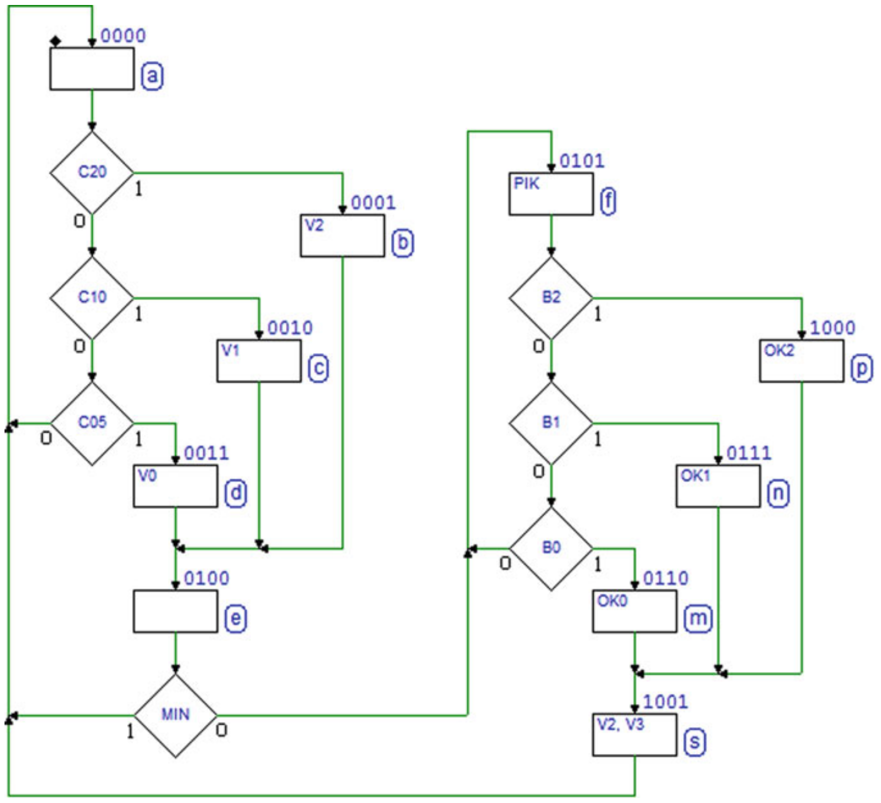
In one of states (b), (c) or (d), the adder generates the sum of the number $V3..V0$ and the number on the register.



The register memorizes the sum on the *next positive edge* of the clock, so it is necessary to add to the ASM state (e) to wait for one clock cycle before checking *MIN* (see the complete ASM diagram below).

We go back to (a) if the money inserted is lower than the cost of the drink. When the sum reaches or exceeds the 20 cent cost, *MIN* goes to zero and the FSM to state (f) where it activates *PIK* to ask the user to choose a drink and waits for him/her to press one of the push-buttons $B0$, $B1$ or $B2$.

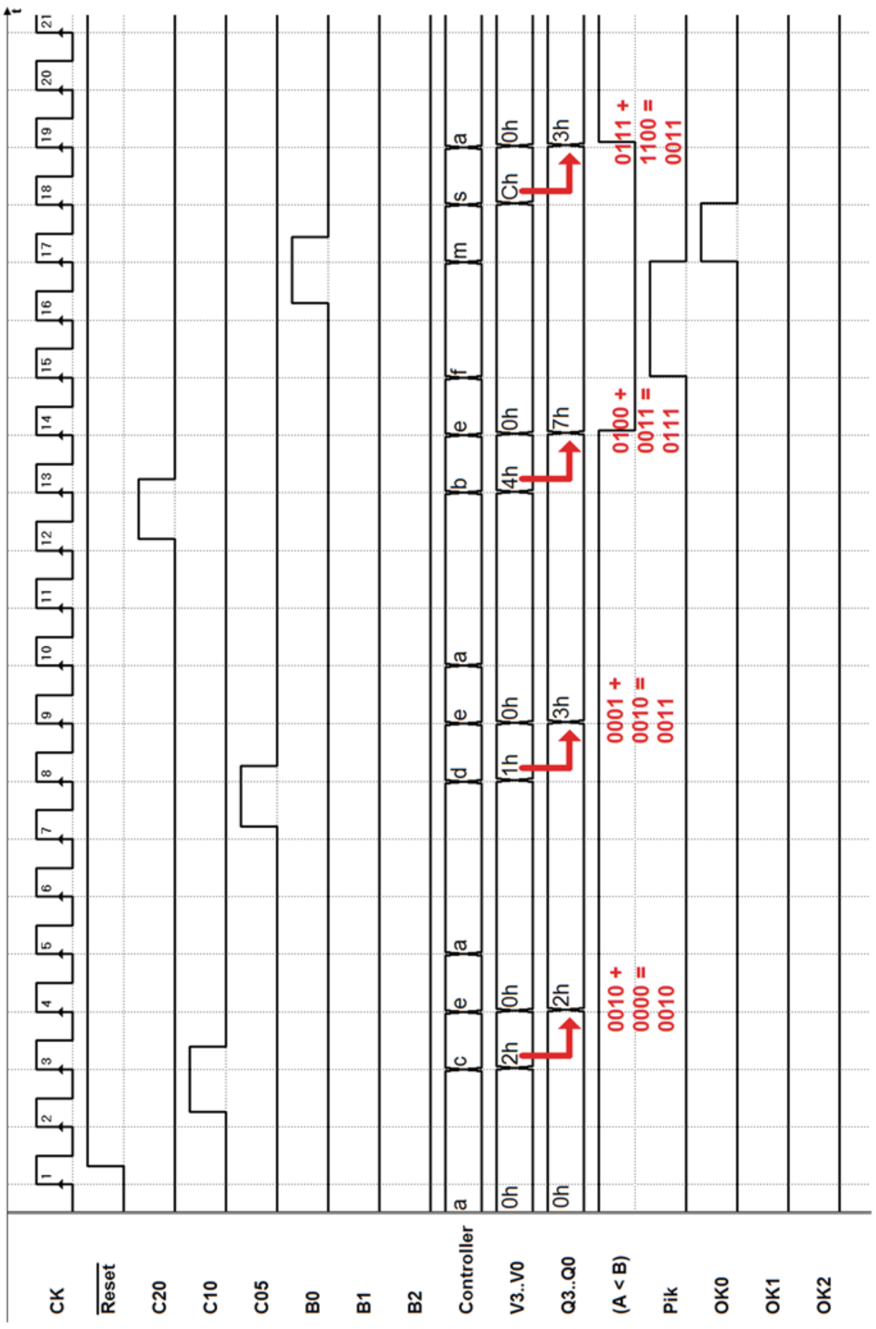
Depending on which button is pressed, $OK2$, $OK1$, or $OK0$ is activated for one clock cycle to command the corresponding drink to be dispensed.



Finally, to close the diagram, we make it so that any residual credit is available for the next user. To subtract 20 cents from the register, the FSM sets $V3..V0 = 1100_2$ in state (s) (two's complement of the code for 20 cents).

The system's timing simulation in the next figure highlights the adding operation of the value set by the FSM on $V3..V0$ and the value on the register at that moment.

Notice the last addition where the cost of the drink is subtracted from the total (35 cents $= 7_{10} = 7_h = 0111_2$) by adding two's complement to it (-20 cents $= -4_{10} = C_h = 1100_2$).


CK
Reset
C20
C10
C05
B0
B1
B2
Controller
V3..V0
Q3..Q0
(A < B)
Pik
OK0
OK1
OK2
t
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a c e a d e a b e f m s a
0h 2h 0h 1h 0h 4h 0h Ch 0h
0h 2h 3h 7h 3h
0010 + 0000 = 0010
0001 + 0010 = 0011
0100 + 0011 = 0111
0111 + 1100 = 0011

8.3.7 Programmable Square Wave Generator

We anticipate that here the *network structure* will not be provided as in previous examples but must be designed according to specifications.

We are designing a synchronous (*controller–datapath*) digital system that generates a two-level periodic signal (hereinafter called *"square wave"*). The system must have a group of eight input lines, which we will call *TH*, a second group of input lines *TL* and a single output line *SW*. Upon activation, the system generates a square wave on *SW* without waiting for any command. The high part of the square wave lasts for as many clock cycles *CK* as those set by $(TH + 1)$; the low part has the same duration as the number set by $(TL + 1)$. The generation of *SW* begins with the high part of the waveform.

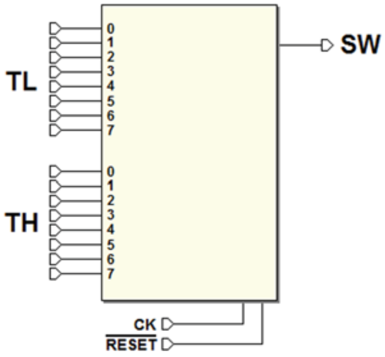
The steps to take to carry out a design of this type are:

(a) Define the system as a functional block, highlighting inputs and outputs.
(b) Choose the components to use in the datapath.
(c) Design the datapath with the connections between the controller and the datapath components and the inputs and outputs of the whole system.
(d) Draw the ASM diagram of the FSM that describes the controller.
(e) Perform a timing simulation of the whole system by choosing input sequences that help demonstrate the system's functionality set.

Step (a)
It is fundamental in this phase to correctly identify the system's inputs and outputs based on the given specifications. Here is a summary of what was defined in the text:

1. *Inputs*:
   (a) *TL* (8 bit) sets the duration of the low part of the periodic signal in pure binary.
   (b) *TH* (8 bit) sets the duration of the high part.
2. *Outputs*:
   (a) *SW* (1 bit) periodically commutes between the logical value 0 and 1.

The system, represented as a single functional block, is as follows:

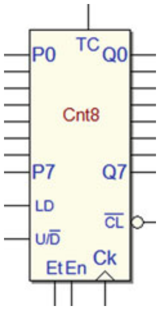


Step (b)

To choose the datapath components, we must know which functions are required to create the system. Remember that the controller must be used as such, not as a register, a counter, a comparator or to memorize and/or process numerical data sets, for example.

To create the final, complete system, the controller must be the *"director"* of combinational, arithmetic, or memorization components. In this example, the system must acquire *TL* and *TH* and use them to measure the number of clock cycles where *SW* must remain low or high.

The best component for this job is an *8-bit universal counter* like the "Cnt8" from the *Deeds* library. We have already encountered a smaller version of it (for example in the *Pulse Generator* on p. 372).
We can pre-load it with a number and then, with the *count down* enabled, we can wait for *TC* (*Terminal Count*).
So, according to the specifications, the components needed are two universal counters of this type, one for *TL* and one for *TH*.



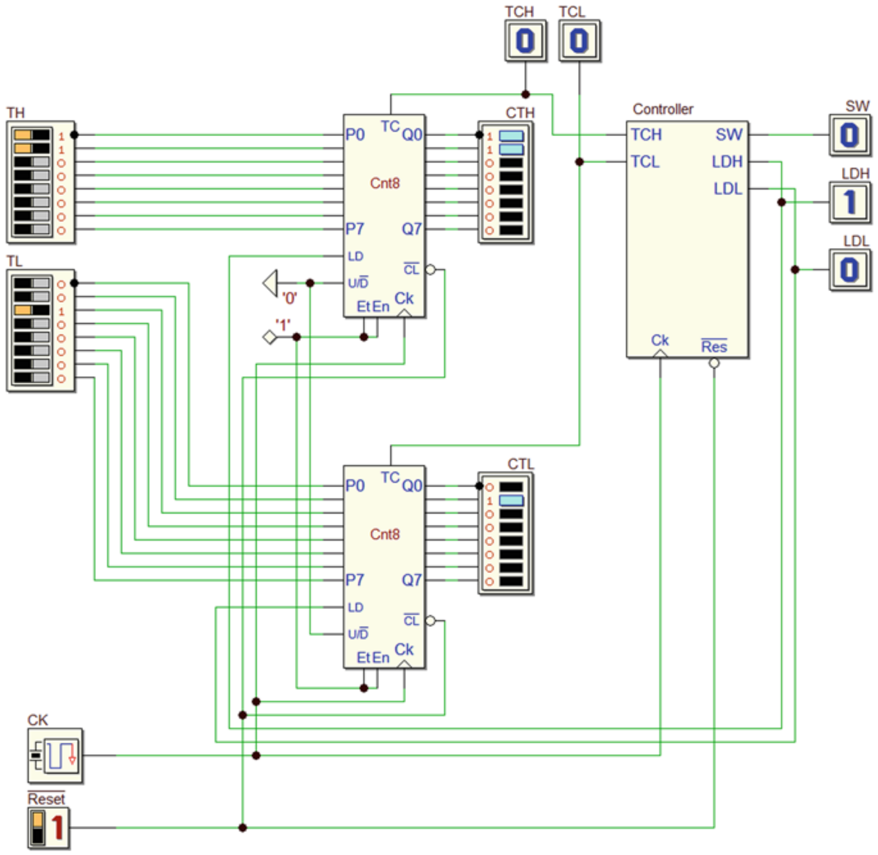
Step (c)

This step is deeply connected to step (b). As explained above, two counters enabled for counting down are needed. They will be loaded at the right time by the controller (with the values of *TL* and *TH*). The *TCs* will be evaluated by the controller to check if the set time has passed.

Now, it is important to know how to connect the two counters' inputs and outputs to the controller and the system. Specifically:

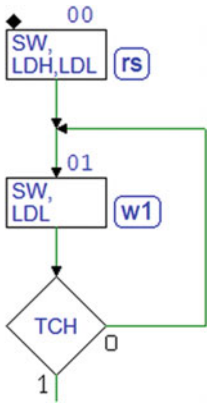
- The inputs *P*7..*P*0 of the two counters must be connected to the *TL* and *TH* input groups, as explained before.
- The counters must count down so let's set input $U/D = 0$.
- The input *LD* of the counters must be commanded by the controller so that they are loaded at the right time. It is possible to keep Et and En active all the time so that the counters work always and *LD* is the only necessary control. Then, the controller will be simply waiting for *TC*.
- The output *SW* can be driven directly by the controller.
- The schematic includes signals as *LD* and *TC* of the two counters, *LDL* and *TCL* for the counter connected to *TL*, *LDH* and *TCH* for the counter of *TH* and the output *SW*.
- It is convenient to add a few auxiliary outputs into the system schematic, just to visualize the *relevant signals* and *counter outputs*, during the simulation.
- Notice that the clock *CK* and the system $\overline{Reset}$ must be shared by the counters and the controller.

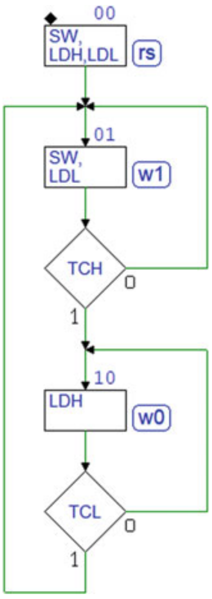
After these evaluations, the final schematic is shown here:



Step (d)
Once points (a), (b), and (c) are defined, point (d) should not pose particular difficulties in that the controller should implement the logic specified in describing the datapath design.

Notice that when the system's $\overline{Reset}$ is activated, the two counters go to zero. When $\overline{Reset}$ is released, if *LDH* and *LDL* are not active, the counters will start to count. It can be useful to activate them in the *reset state* (rs), as shown at the right.
In this way, we start pre-loading the counters with the *TL* and *TH* values. To satisfy the specifications, the system needs to start generating *SW* = 1, so we activate also *SW* in the reset state.
We no longer activate *LDH* in the next state (w1) in order to leave the counter free to count, and we wait for its terminal count *TCH*.

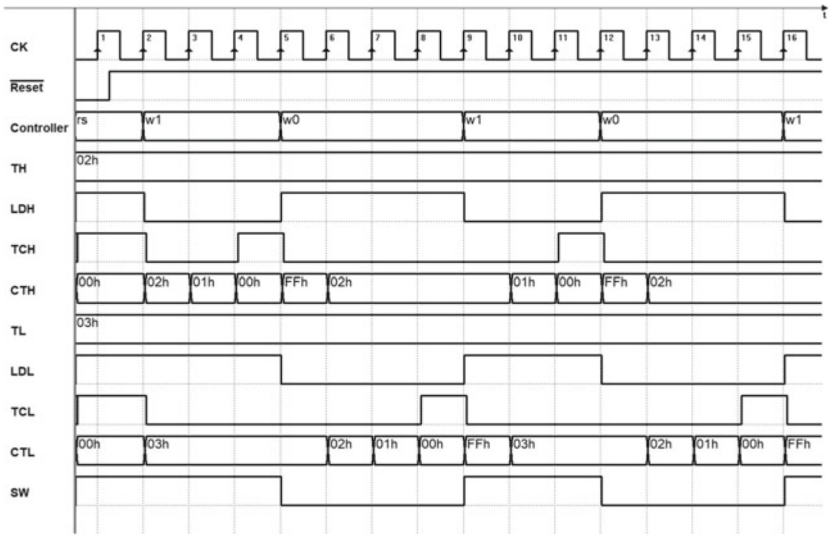




This way, we keep output *SW* at 1 for *(TH* + 1) clock cycles (+1 because we count also the zero).
To be ready to start the count for the low part of *SW*, we maintain pre-loaded in (w1) the related counter, activating *LDL*.
When *TCH* signals that count has ended, it is time to change the output *SW* value to 0, so we introduce another state (w0).
In this new state, the counter of the low part is free to count (*LDL* is 0) and the FSM stands by for the terminal count *TCL*.
Then, we finalize the ASM diagram closing the loop to state (w1), on the activation of *TCL*.
The output sequence on *SW* will be repeated indefinitely.

Step (e)

To test the system, we define a timing trace that includes the activation of reset and the setting of *TL* and *TH* (in the example, they are set to 3_{10} and 2_{10}, respectively). The output square wave on *SW* is low for four clock cycles and high for three.



As we can see in the simulation, the first part of the signal on output *SW* is wrong (it is kept high for a longer time than needed). Starting from the subsequent cycles, however, the system produces the square wave that was set by the inputs *TH* and *TL*.

8.3.8 Christmas Light Systems

Design a system for Christmas lights that controls three strips of red, green, and blue lights. Each strip should light up in sequence for a time controllable by a 4-bit number. The *network structure* must be designed according to the specifications above.

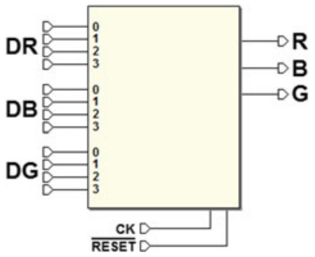
Here too, we should read the project specifications carefully and follow the same five steps (a), (b), (c), (d), and (e) suggested on p. 397.

Step (a)

In step (a), we must define clearly inputs and outputs of the system. Reading the specifications above, we understand that the machine must have 12 inputs and three outputs that we identify as follows.

1. *Inputs*:
   (a) *DR*, four bits, lighting time of the red light (in binary);
   (b) *DB*, four bits, lighting time of the blue light;
   (c) *DG*, four bits, lighting time of the green light.
2. *Outputs*:
   (a) *R* (one bit), red light control (on for *DR* clock cycles);
   (b) *B* (one bit), blue light control (on for *DB* clock cycles);
   (c) *G* (one bit), green light control (on for *DG* clock cycles).

The system, as a single functional block, will appear as follows:

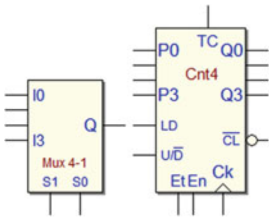


Step (b)

The system acquires *DR*, *DB*, and *DG* to use them to measure the number of clock cycles the output signals *R*, *B*, *G* should be kept active for. We could use three counters, as in the previous example.

Another possible architecture is to use a single 4-bit counter that will be pre-loaded to the correct value by multiplexers connected to system inputs *DR*, *DB* and *DG*.

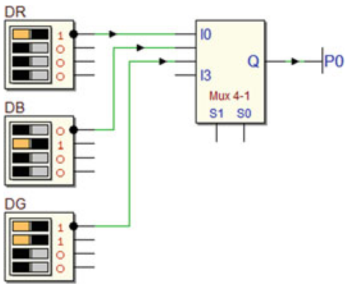
We need a counter "Cnt4" (see on p. 372) and four multiplexer "Mux4-1" (p. 51), shown on the right.



Step (c)
Following the previous analysis, we should decide how to connect the elements of the system:

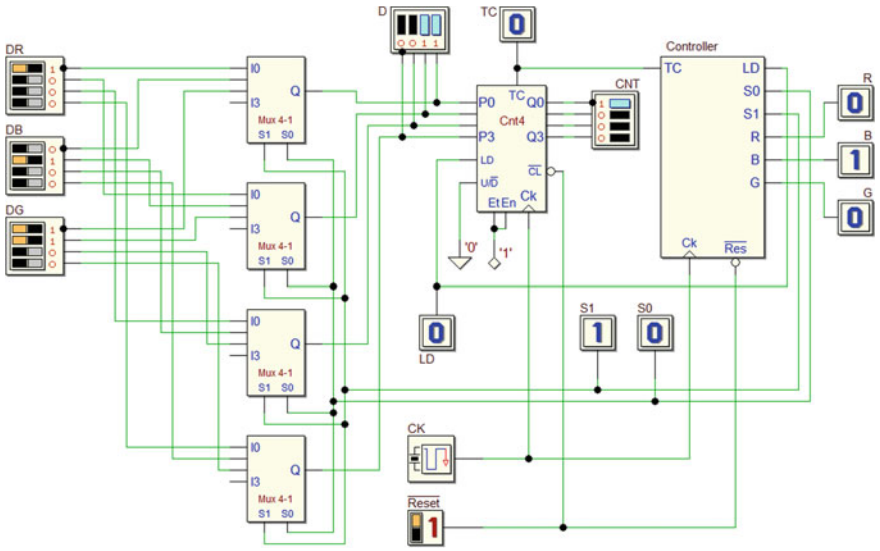
- The counter's inputs $P3..P0$ must be connected to the three groups of inputs *DR*, *DB*, and *DG* by the multiplexers.
- We use four multiplexers, one for each bit of *DR*, *DB*, and *DG*, as described in the following:

The bits in position 0 of *DR*, *DB* and *DG* will be connected to the inputs $I0$, $I1$ and $I2$ of the first multiplexer, respectively, as shown in the figure. The same will be done for the other bits, through the other multiplexers. This way, the outputs Q of the four multiplexers will copy *DR*, *DB* or *DG*, based on how their selection inputs $S1$, $S0$ are set (at 00, 01 and 10, respectively).



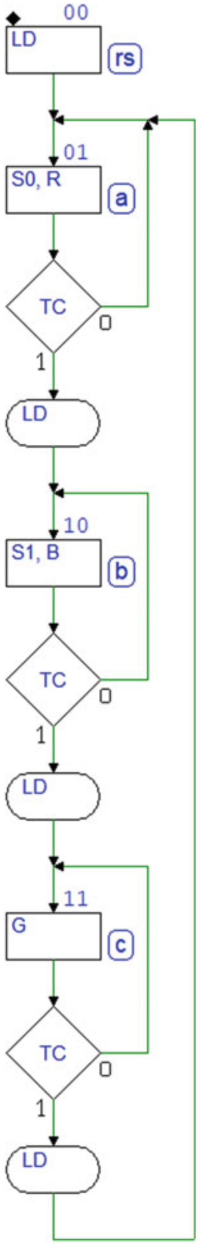
- The counter must count down, so U/D is set at 0. We set also $Et = En = 1$, so that we control the counter using the input *LD* only.
- Let's take *DR* as an example: the controller sets the selection lines $S1$, $S0$ at 00 and then activates *LD*. The counter generates *TC* after $DR + 1$ clock cycles (since the count also includes zero).
- The controller generates directly the outputs *R*, *B*, and *G*.
- Clock *CK* and $\overline{Reset}$ are common to the counter and the controller, as discussed in the previous example.

Below is the final schematic of the system.



Step (d)

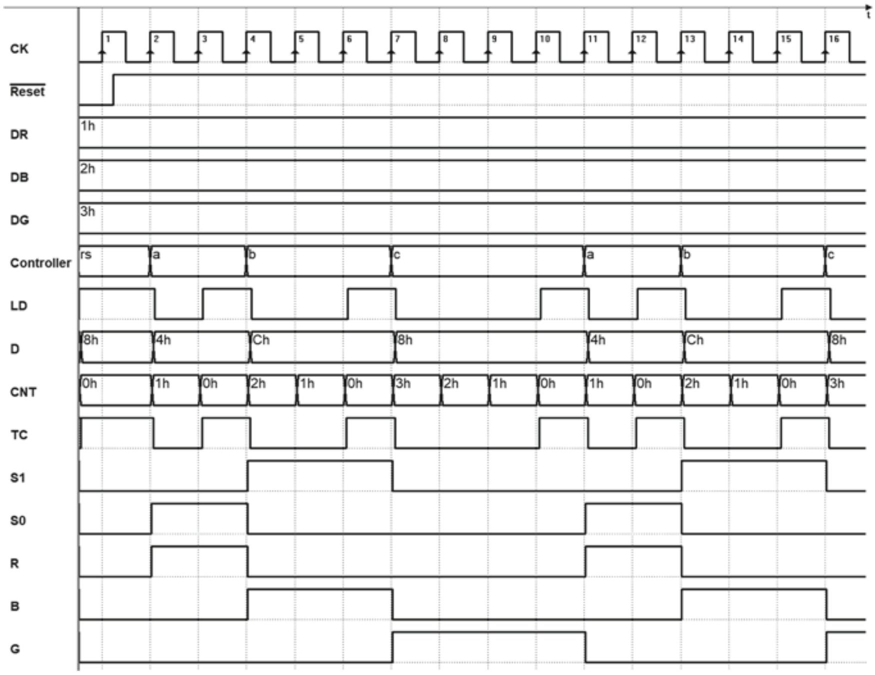
As we can see in the figure, in the *reset state* (rs), we activate only the counter's input *LD*. This means in the same state (rs) we are setting $S1$, $S0 = 00$ on the multiplexers, so that *DR* is routed to the counter pre-load inputs $P3..P0$.



Note that the *reset state* (rs) is not included in the main algorithm loop (it is used only to initialize the counter when the FSM starts working).
On entering the following state (a), the counter will be loaded with the number on *DR* (the lighting time of the red light). In state (a) we activate output *R*, and wait for *TC* from the counter.
The counter's input *LD* is activated in state (a) only when the counter reaches zero. When this happens, on entering the following state (b) the counter is loaded again, but this time with the number *DB*. In fact, in state (a) we have prepared $S1S0 = 01$ (declaring only $S0$ in the block), to route the multiplexers accordingly.
State (b) is similar in its operation to state (a), but now *B* is active and we set the multiplexers to present *DG* to the counter's pre-load inputs.
In the same way, in state (c) we activate *G* and set the multiplexers to re-load again the counter with the number on *DR*.
The FSM loops back to state (a) when the lighting time of the green light has elapsed.

Step (e)

To simulate the complete system (see the timing diagram on the next page), we set inputs $DR = 1$, $DB = 2$ and $DG = 3$ in addition to the usual initial reset sequence. Values have been chosen to obtain a reasonably short graphic.
Notice how the outputs related to lights *R*, *B* and *G* activate in a cyclical sequence for 2, 3 and 4 clock cycles, respectively.


CK
Reset
DR
DB
DG
Controller
LD
D
CNT
TC
S1
S0
R
B
G
1h
2h
3h
rs
a
b
c
8h
4h
Ch
0h
1h
2h
3h
t

8.4 Design Exercises

8.4.1 Design of the Controller of a Given Datapath

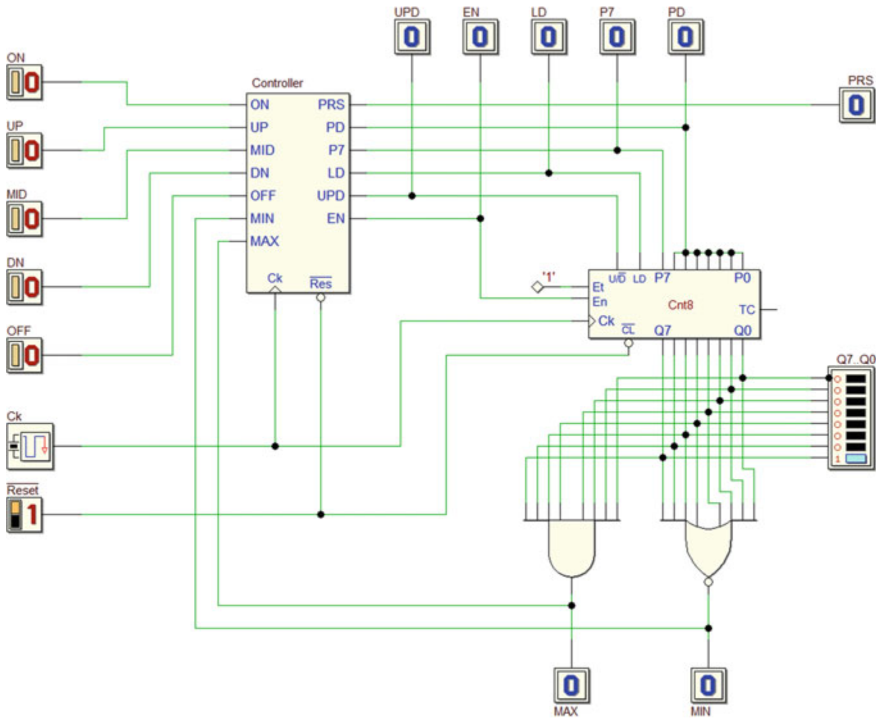
For each of the following exercises of digital systems design, the schematic of a complete *controller–datapath* architecture is supplied.

You should define the controller's ASM diagram according to the specifications and complete the timing diagram (on the opposite page). Remember to indicate the state of the FSM at each clock cycle. The synthesis of the FSM is not requested.

For each exercise, the Web site offers:

(a) A trace of the FSM to design where state variables, inputs, and outputs are pre-defined.
(b) A PDF file with a suggested timing diagram template, to fill in on paper without using the simulator.
(c) The network schematic where you can insert your FSM to check its behavior through the timing simulation.

Exercise 1 The system shown in the figure below contains the controller, a "Cnt8" counter and two logical gates.



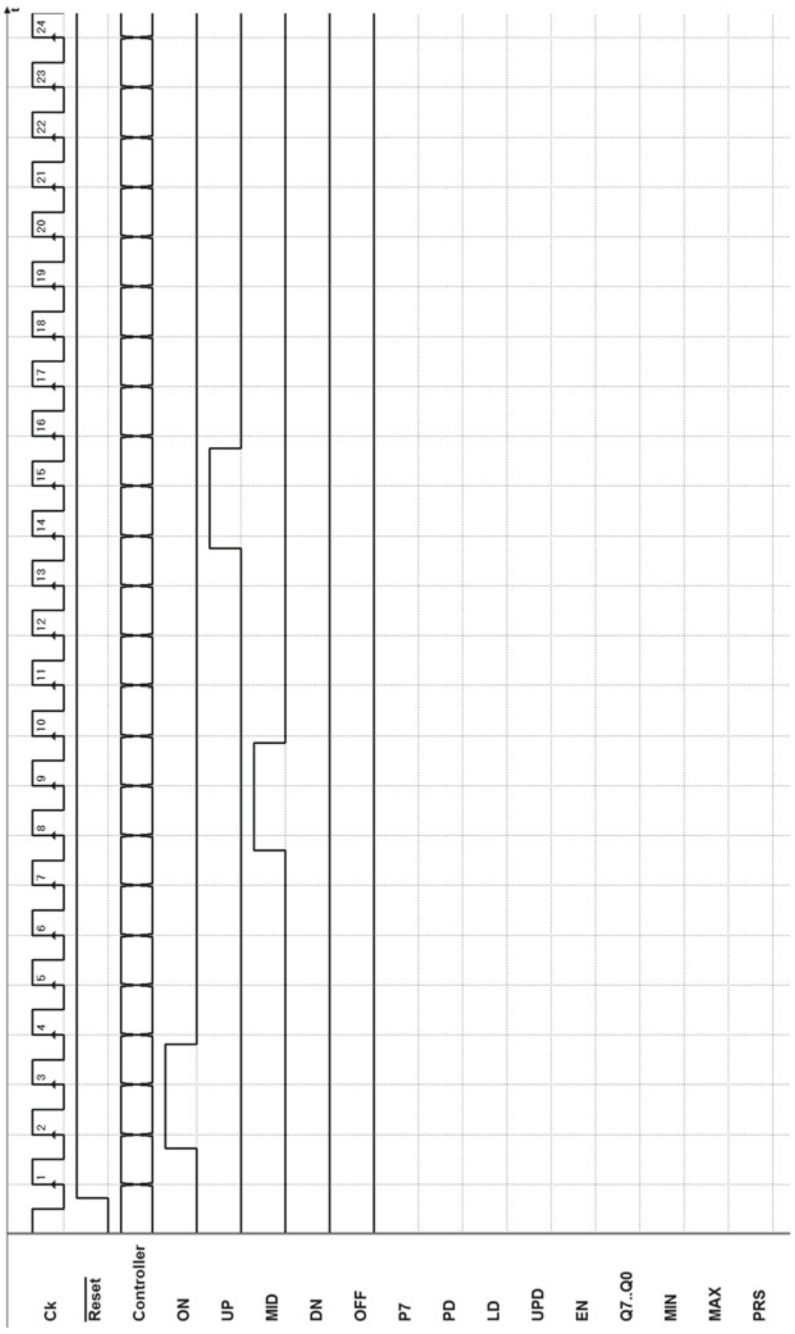
The system implements a *binary number generator*, controlled by five push-buttons *ON*, *UP*, *MID*, *DN*, and *OFF*. Output $Q7..Q0$ is taken directly from the counter.

Pressing and *then releasing* each push-button determine the system's behavior. Push-buttons' functions are:

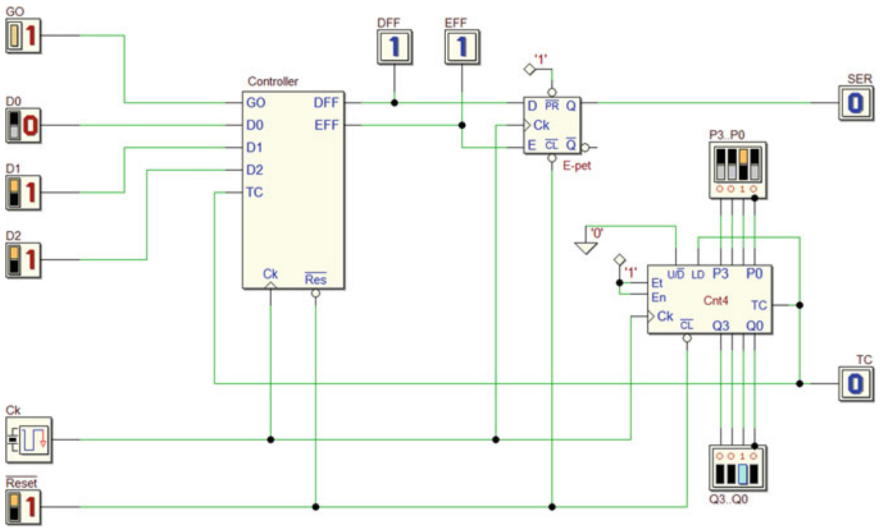
ON	$\rightarrow$ Sets number$Q7..Q0$at the highest value.
UP	$\rightarrow$ Increments the number by one unit.
MID	$\rightarrow$ Sets number$Q7..Q0$to the intermediate value$(= 10000000_2)$;
DN	$\rightarrow$ Decrements the number by one unit.
OFF	$\rightarrow$ Sets number$Q7..Q0$to the lowest value.

In the idle state, the push-buttons are at 0, while pressed they are at 1.

The controller activates output *PRS* when one of the push-buttons is pressed. Number $Q7..Q0$ does not have to be increased or decreased if it has reached the highest or lowest value, respectively.


Ck
Reset
Controller
ON
UP
MID
DN
OFF
P7
PD
LD
UPD
EN
Q7..Q0
MIN
MAX
PRS
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24

Exercise 2 The system shown in the figure below is made up of a controller, a "Cnt4" counter and an E-PET flip-flop.



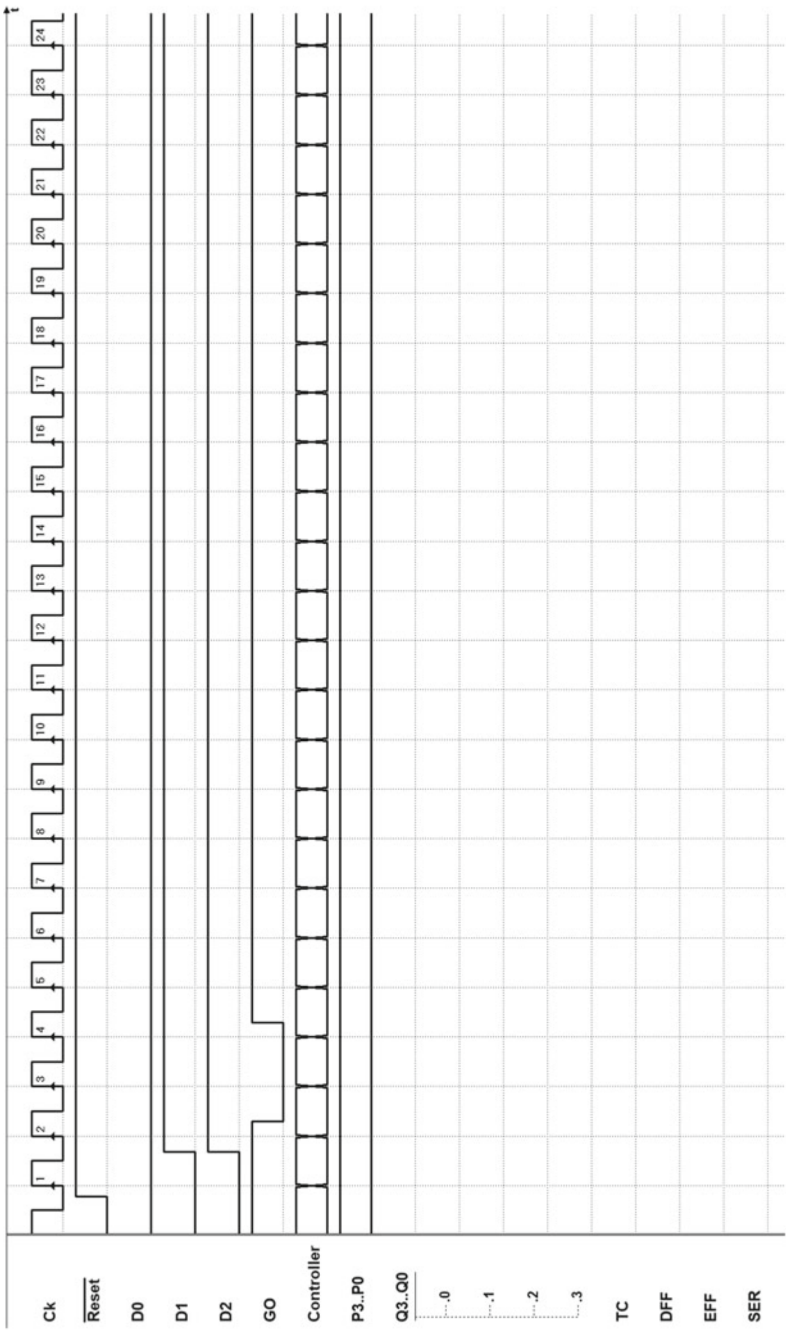
The system, a *serial transmitter*, must generate on output *SER* a 5-bit packet with one start bit at 1, three data bits (order: $D0$, $D1$, and $D2$), and a stop bit at 0.

Duration N of each bit of the packet (the bit time) is P time the clock period CK. P is the number set at the counter's inputs $P3..P0$.

To carry out the system's timing analysis, assign $P = 2$.

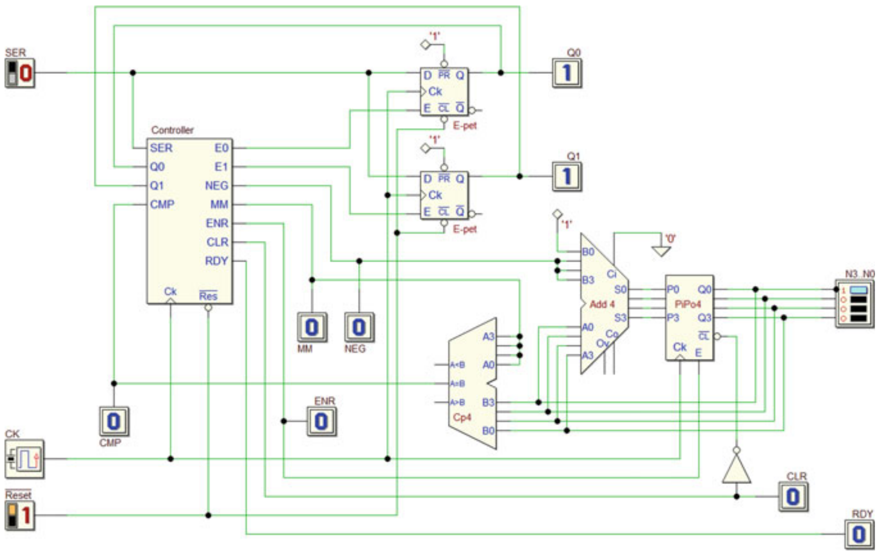
The data to transmit are available on the controller's inputs $D0$, $D1$, and $D2$. A falling edge on input *GO* starts the packet's generation.

Determine the relation between N and the number P that depends on your own specific solution.


t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
Ck
Reset
D0
D1
D2
GO
Controller
P3..P0
Q3..Q0
.0
.1
.2
.3
TC
DFF
EFF
SER

Exercise 3

The system shown in the figure below includes a controller that receives data from a serial line and manages a network made of two E-PET flip-flops, a register, and a few arithmetic circuits.



On *SER*, the controller receives a 4-bit pack with one start bit at 1, two data bits (order: *D*0 and *D*1), and a stop bit at 0. The duration of the bit time is one clock period; the two data bits codify an operation that the system carries out on outputs *N*3..*N*0.

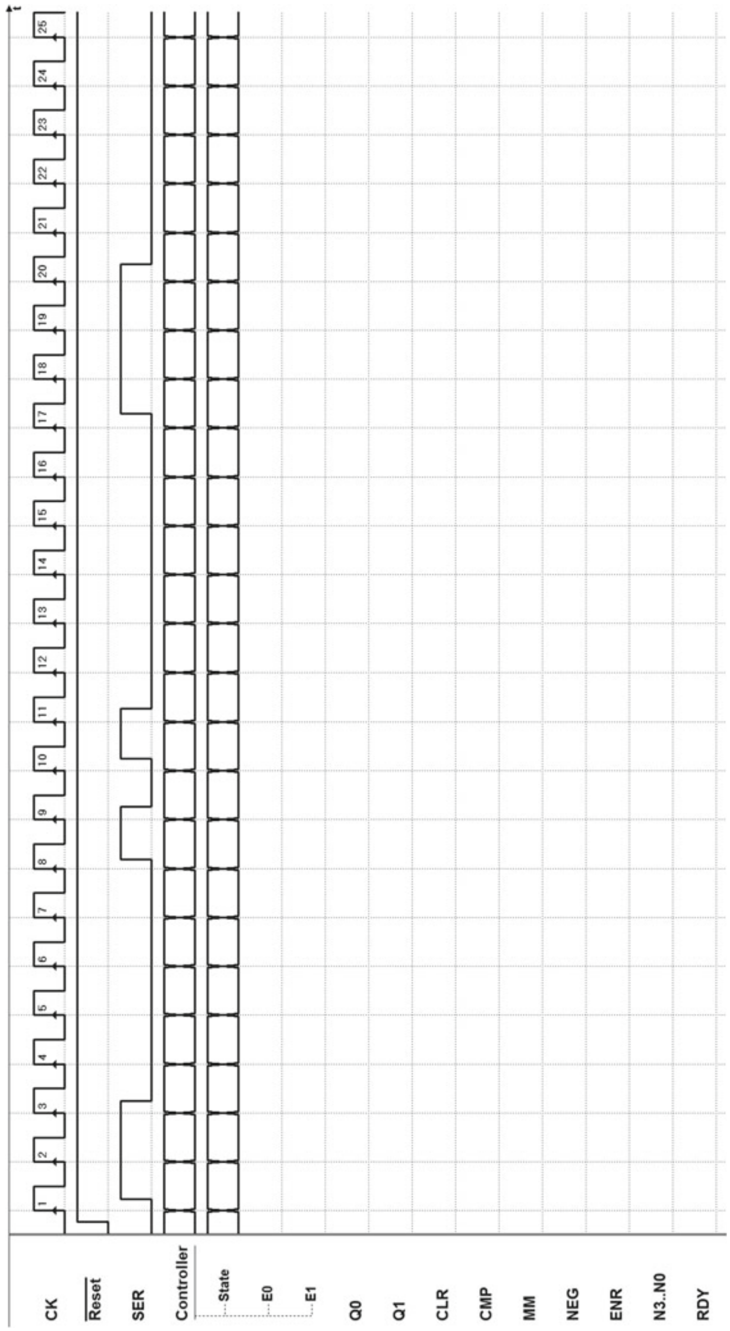
If an incorrect stop bit (= 1) is received, no operation is carried out and the system waits for *SER* to go back to zero before waiting for the next packet. When a correct one (stop bit = 0) is received, the system carries out the following operations according to the value of *D*1 and *D*0.

*D*1	*D*0	Operation
0	0	No operation (NOP)
0	1	Clear output *N*3..*N*0
1	0	Increment output *N*3..*N*0 by one
1	1	Decrement output *N*3..*N*0 by one

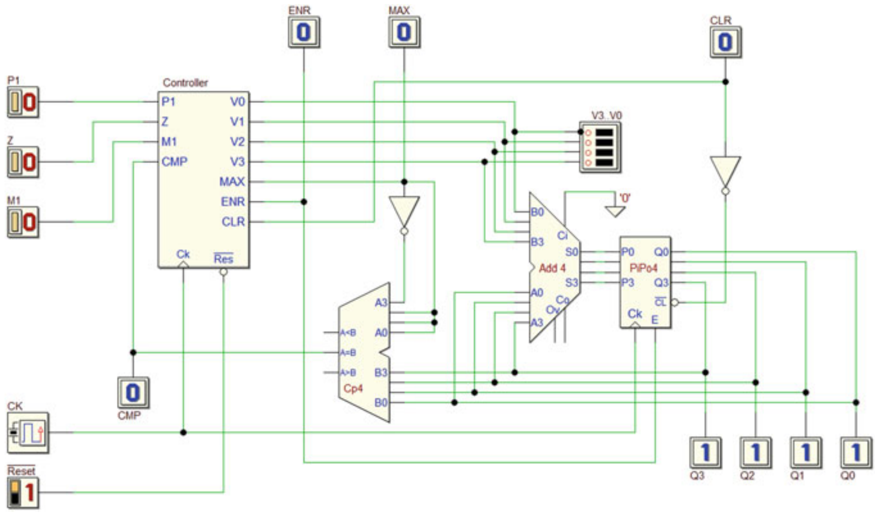
Output *N*3..*N*0 does not increase when it has reached the highest value nor does it decrease when it has reached the lowest. Output *RDY* is activated for one clock cycle when any command (except NOP) is received.

Notice that the FSM memorizes bits *D*1 and *D*0 on flip-flops *Q*1 and *Q*0, so that their values can be reused by the FSM itself.

The FSM output *NEG* selects two different values on the input *B*3..*B*0 of the adder. The FSM line *MM* allows to choose two different numbers to be compared with the output number *N*3..*N*0.


t
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
CK
Reset
SER
Controller
State
E0
E1
Q0
Q1
CLR
CMP
MM
NEG
ENR
N3..N0
RDY

Exercise 4 The system shown in the figure below is composed by of a controller, a register, and a few arithmetic circuits.



The system is a binary number generator controlled by three push-buttons $P1$, Z, and $M1$. Output $Q3..Q0$ is taken from the register "PiPo8." The binary number in the output is signed (two's complement code).

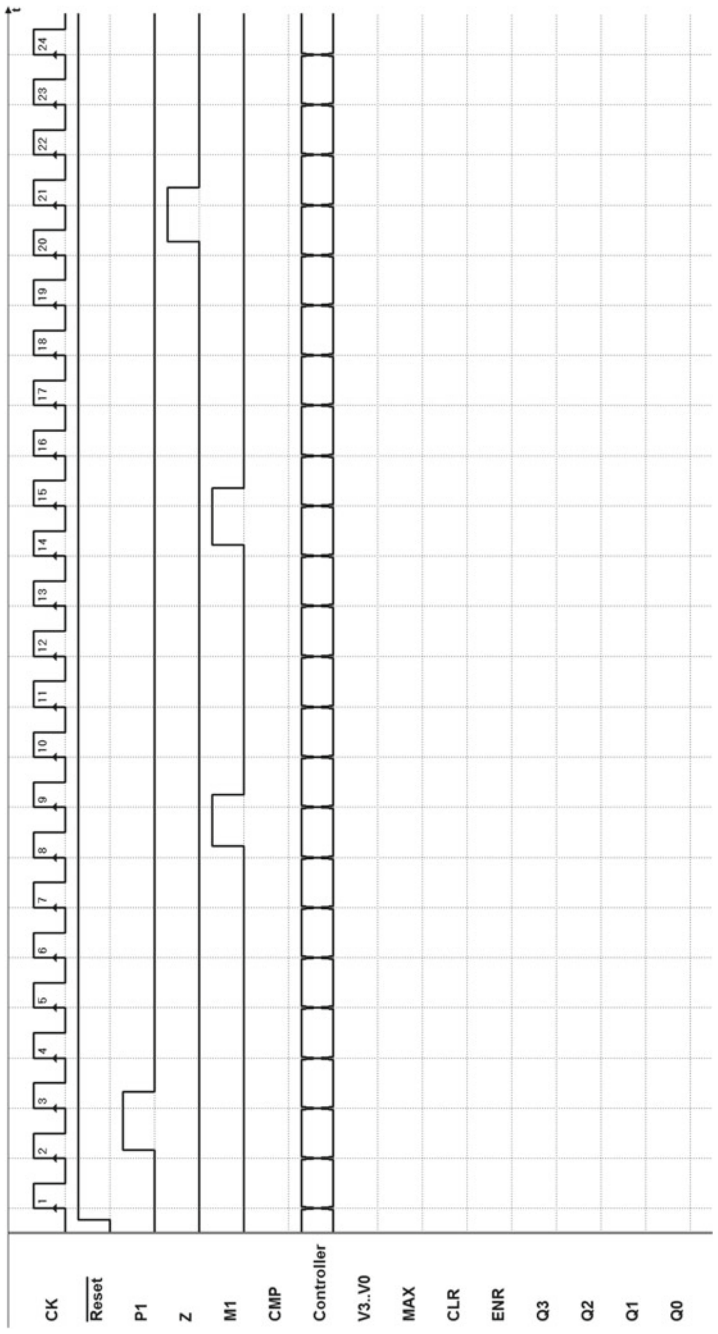
In the idle state, the push-buttons are at 0; while pressed, they are at 1. The push-buttons are assumed to be ideal with no mechanical bounce.

Pressing and then releasing each push-button determine the system's behavior. Push-buttons' functions are defined as follows:

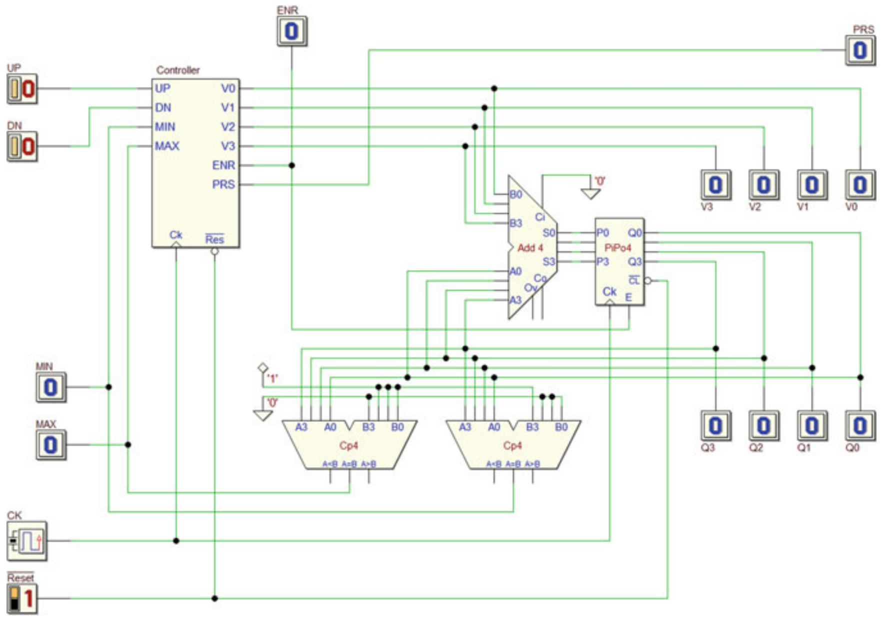
P1	$\rightarrow$ Increments the number $Q3..Q0$ by one.
Z	$\rightarrow$ Clears the number $Q3..Q0$.
M1	$\rightarrow$ Decrements the number $Q3..Q0$ by one.

The number $Q3..Q0$ must not be incremented if it is at the maximum positive value nor decremented if it is at the minimum negative value. When the system is initialized, the number $Q3..Q0$ must be cleared.

Notice that the FSM outputs $V3..V0$ set the values to add on the adder inputs, while output *MAX* allows to compare the number $Q3..Q0$ with the maximum and minimum values.


CK
Reset
P1
Z
M1
CMP
Controller
V3..V0
MAX
CLR
ENR
Q3
Q2
Q1
Q0
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24

Exercise 5 Consider the digital system shown in the figure below. It is made up of a controller, a register, and a few arithmetic circuits.



The system implements a *binary number generator* controlled by two push-buttons *UP* and *DN*. Output *Q*3..*Q*0 is taken from the parallel register "PiPo4." The output binary number is signed (two's complement code).

In idle state, the push-buttons are at 0 and at 1 while they are pressed. The push-buttons are assumed ideal with no mechanical bounce.

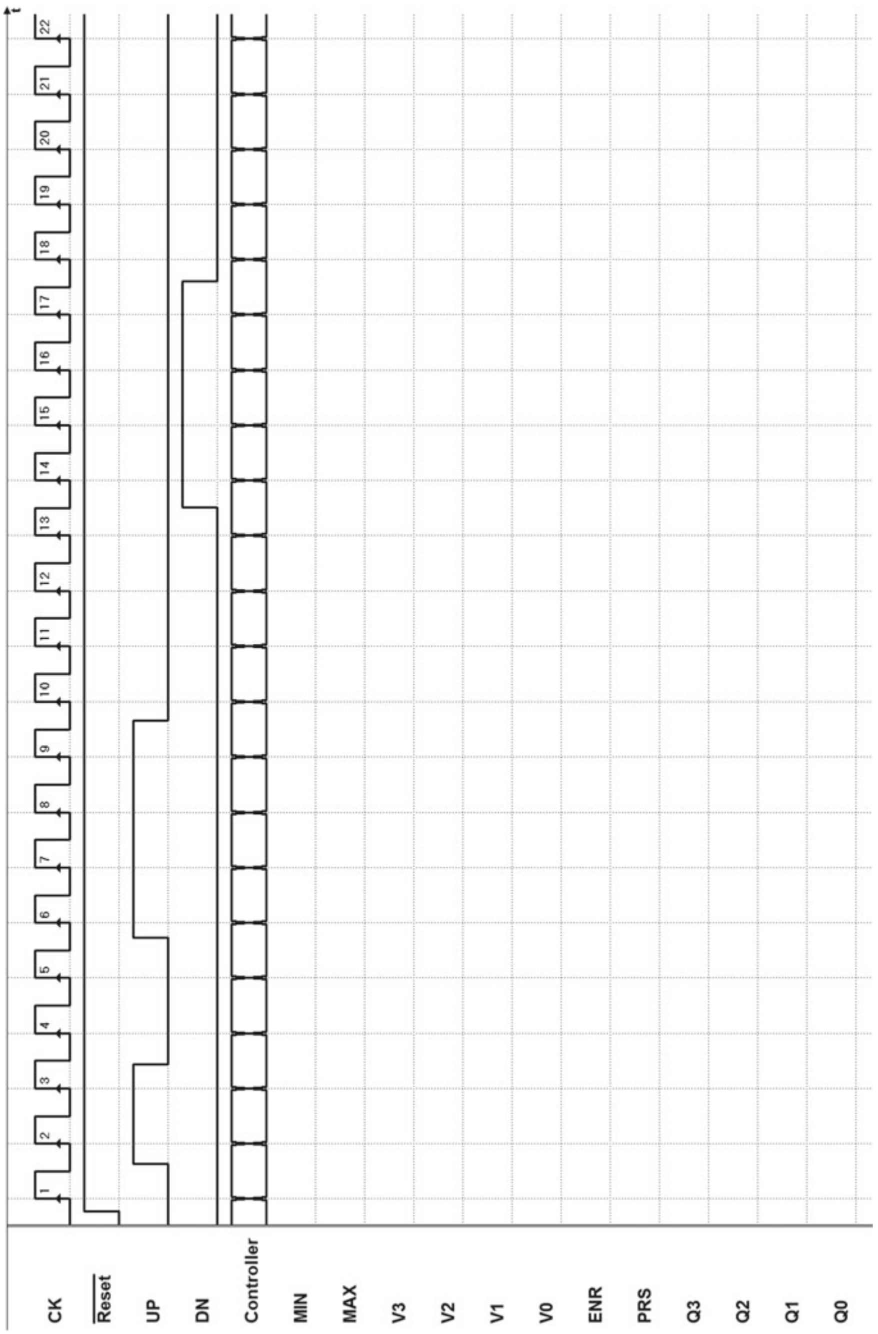
When the push-buttons are pressed and then released, the system carries out the following functions:

UP button $\rightarrow$ increments the number *Q*3..*Q*0 by one.
DN button $\rightarrow$ decrements the number *Q*3..*Q*0 by one.

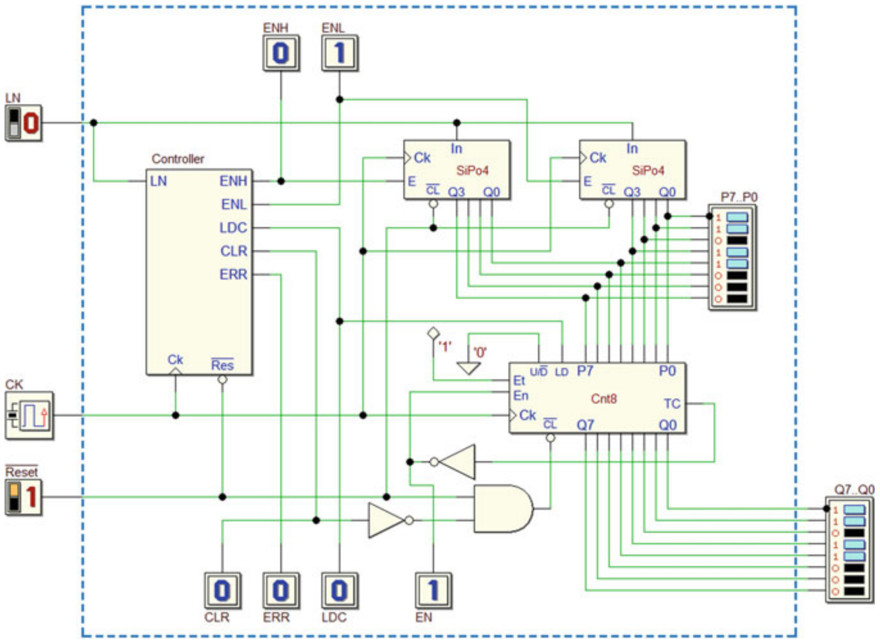
The system ignores the pressing of the push-buttons if its time duration is shorter than 3 clock cycles.

The two comparators receive the value of output *Q*3..*Q*0 on inputs *A*3..*A*0, and they compare it with the maximum and minimum values set on *B*3..*B*0. The number *Q*3..*Q*0 must not be incremented if it is at the maximum (positive) value or decremented, if it is at the minimum (negative) value.

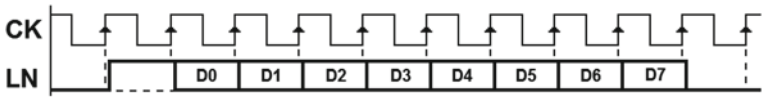
The system activates output *PRS* for the time a push-button is pressed. Assume that the two push-buttons can never be pressed at the same time.


t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
CK
Reset
UP
DN
Controller
MIN
MAX
V3
V2
V1
V0
ENR
PRS
Q3
Q2
Q1
Q0

Exercise 6
The system shown in the figure is made up of a controller, a “SiPo4” shift register, an 8-bit counter “Cnt8,” and a few logical gates.

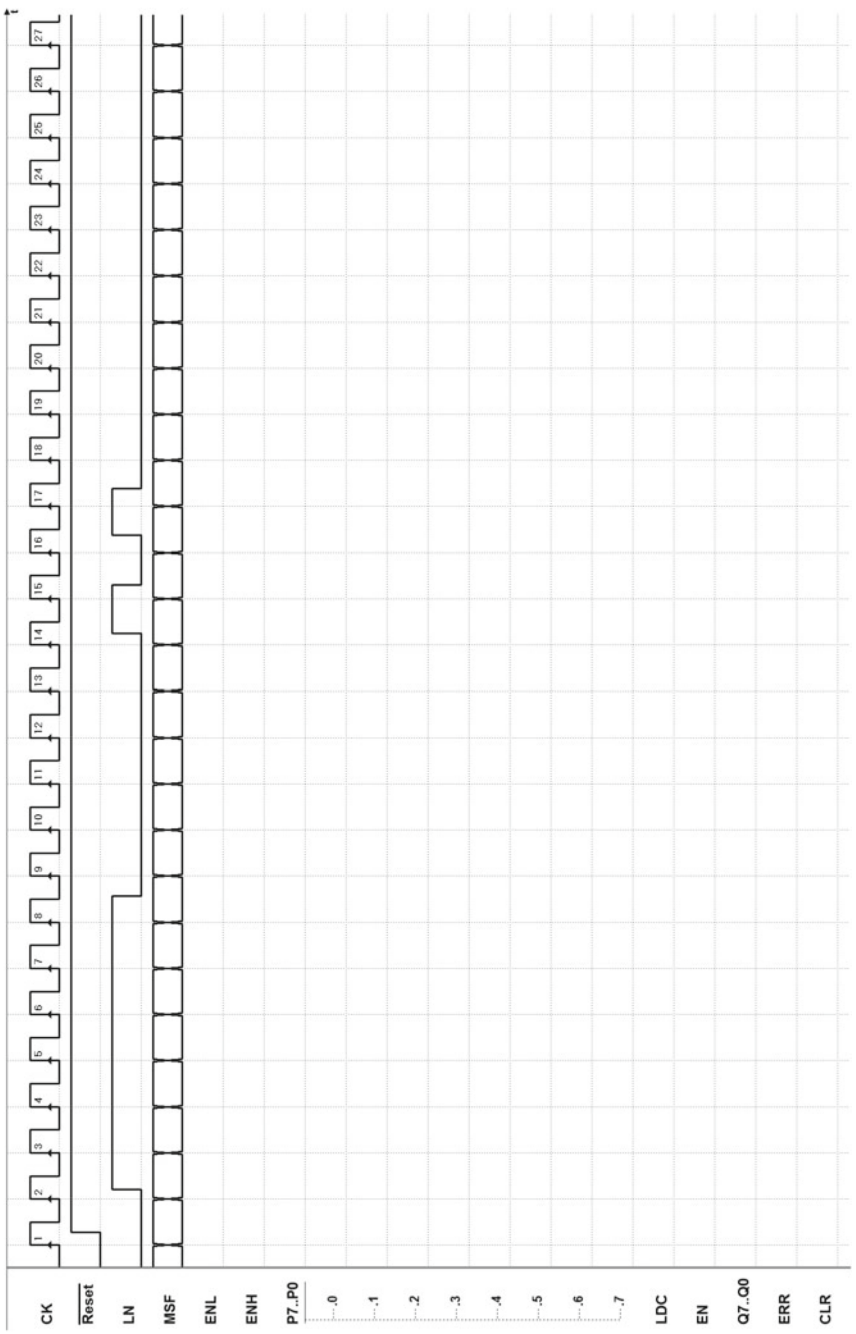


On input *LN*, the system receives a standard serial signal made up of a *start bit*, eight *data bits*, and a *stop bit* (see below).



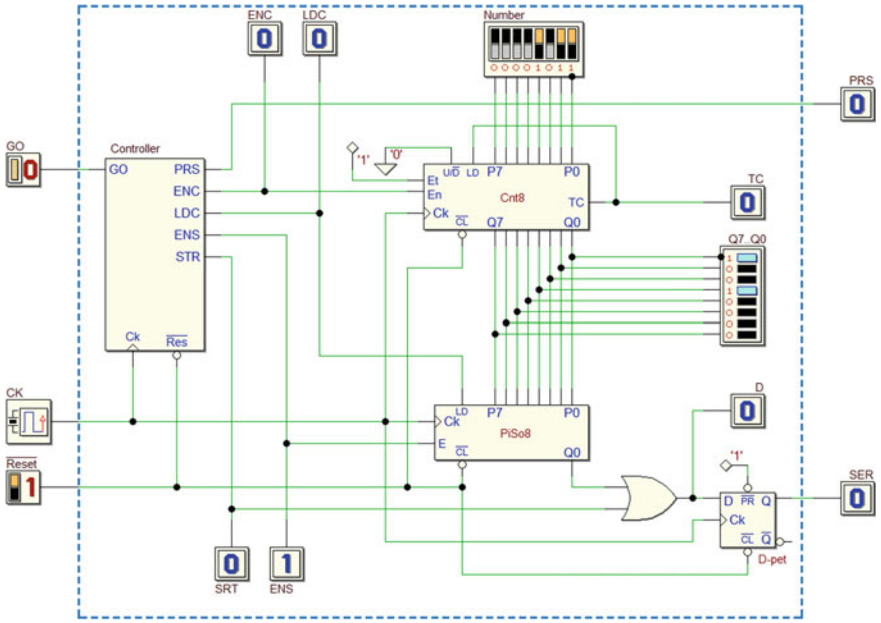
The controller commands the shifting of the serial data into the two registers with the order that we can infer from the schematic (the low part $D3..D0$ on the register at the right of the schematic, the high part $D7..D4$ on the register at the left).

The controller checks the *stop bit* at the end of the serial sequence. If correct, it loads the counter with the byte received through the lines $P7..P0$. If, however, the stop bit is not valid, the counter is cleared and the controller signals *ERR*, keeping it active until *LN* returns to zero.

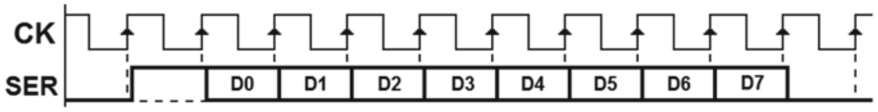

CK
Reset
LN
MSF
ENL
ENH
P7..P0
.0
.1
.2
.3
.4
.5
.6
.7
LDC
EN
Q7..Q0
ERR
CLR
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Exercise 7

Consider the digital system shown in the figure below. It is made up of a controller, an 8-bit counter "Cnt8," a "PiSo8" shift register, a D-PET flip-flop, and a few logical gates.



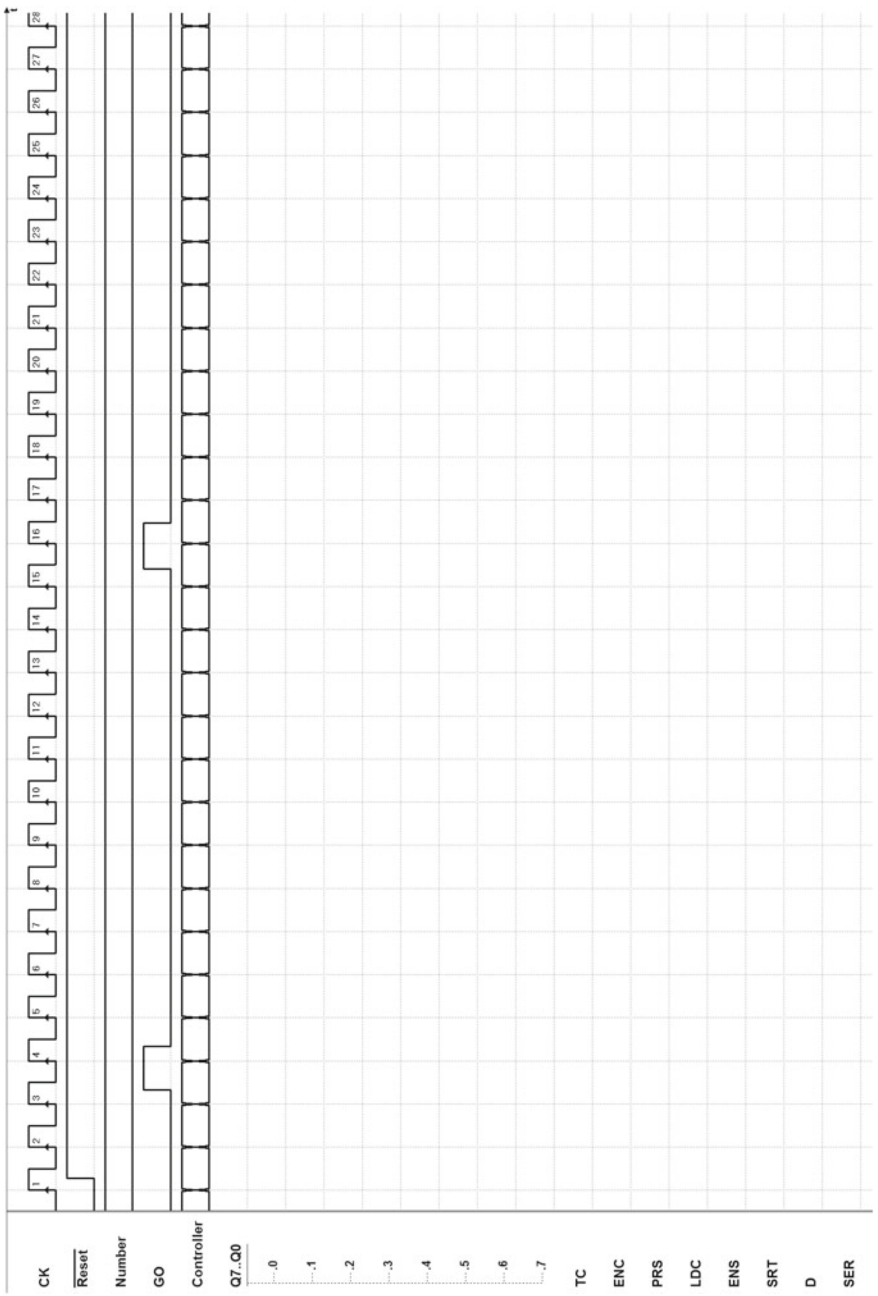
When push-button *GO* is released, the system transmits on *SER* a serial signal made up of a start bit, eight data bits, and a stop bit.



The eight data bits to transmit are taken from the counter's outputs. When push-button *GO* is not pressed ($GO = 0$), the counter is enabled through line *ENC*. For the rest of the time, the count is disabled. When push-button is pressed ($GO = 1$), the FSM activates line *PRS*.

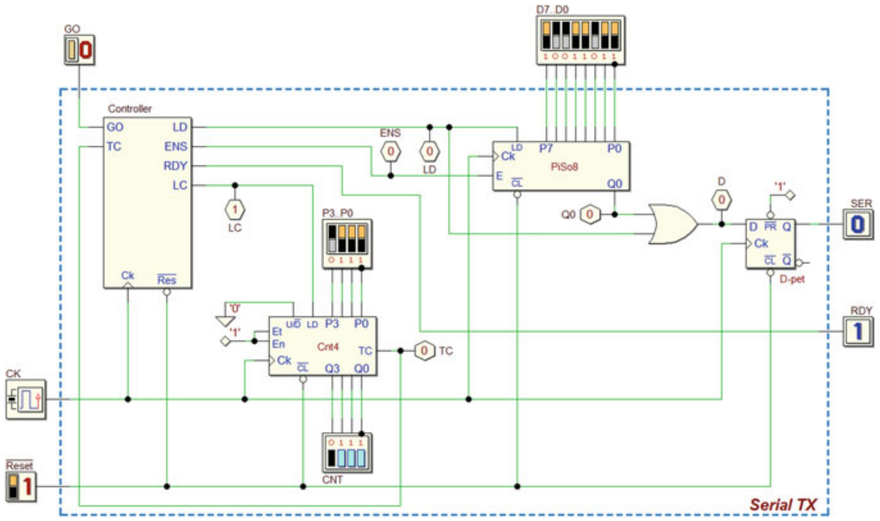
When the push-button is released, the controller starts transmission by loading the counter's outputs onto the shift register through line *LDC*. Notice that line *STR* makes it possible to generate the start bit for one clock cycle and that serial output *SER* is generated by the flip-flop.

The controller commands serialization through line *ENS*, which enables the shift register. When the generation of *SER* is over, the system checks push-button *GO* again. For the purposes of the timing analysis, assume that *Number* (see schematic) equals 0Bh.

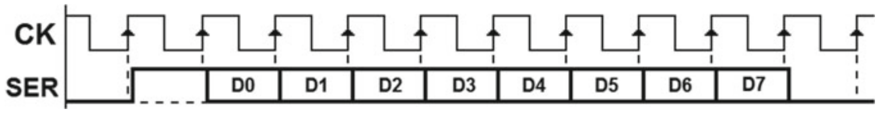

t
CK
Reset
Number
GO
Controller
Q7..Q0
.0
.1
.2
.3
.4
.5
.6
.7
TC
ENC
PRS
LDC
ENS
SRT
D
SER
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

Exercise 8

The digital system shown in the figure below is made up of a controller, a counter, and a shift register.



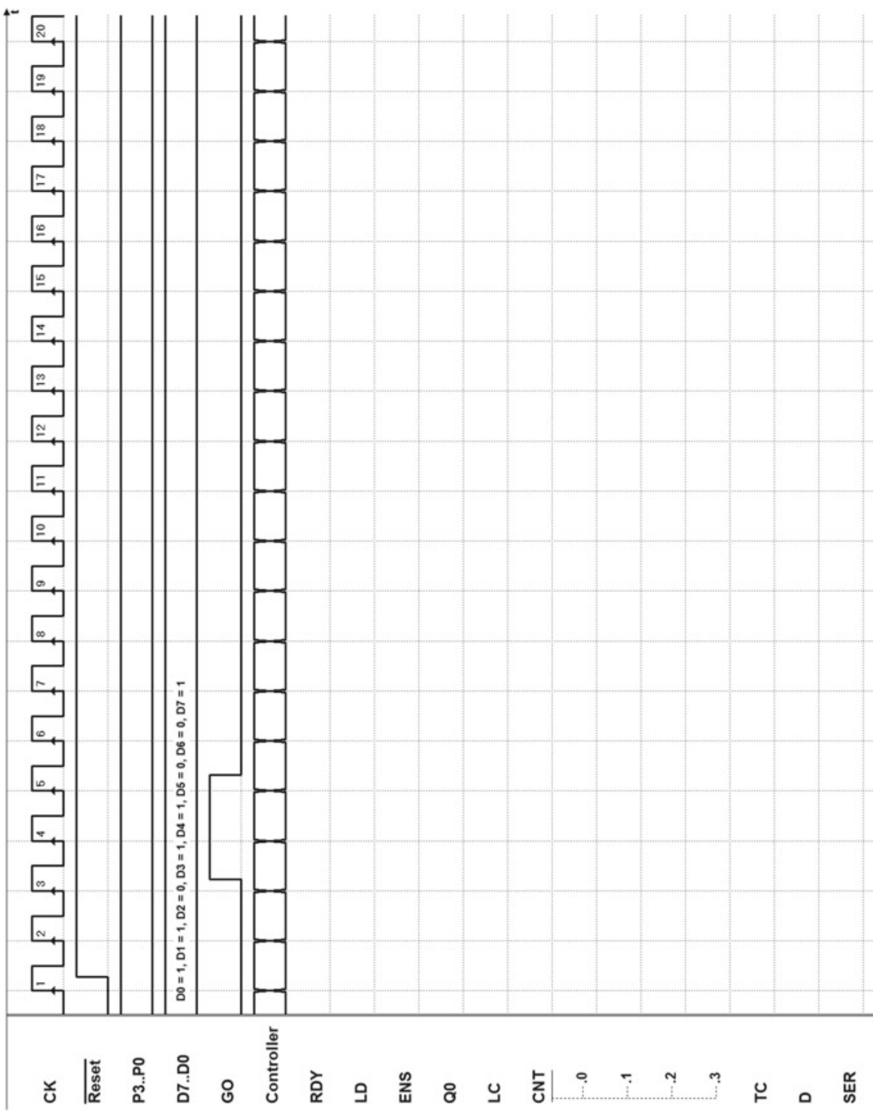
The system is a synchronous serial transmitter whose *bit time* is equal to the clock period. Transmission is launched on the rising edge of input *GO*. The packet transmitted on *SER* is made up of a *start bit* at 1, the eight *data bits* *D*0..*D*7 (in that order), and a *stop bit* at 0.



Output *RDY* (ready) is activated when the system does not transmit but waits for the launch command.

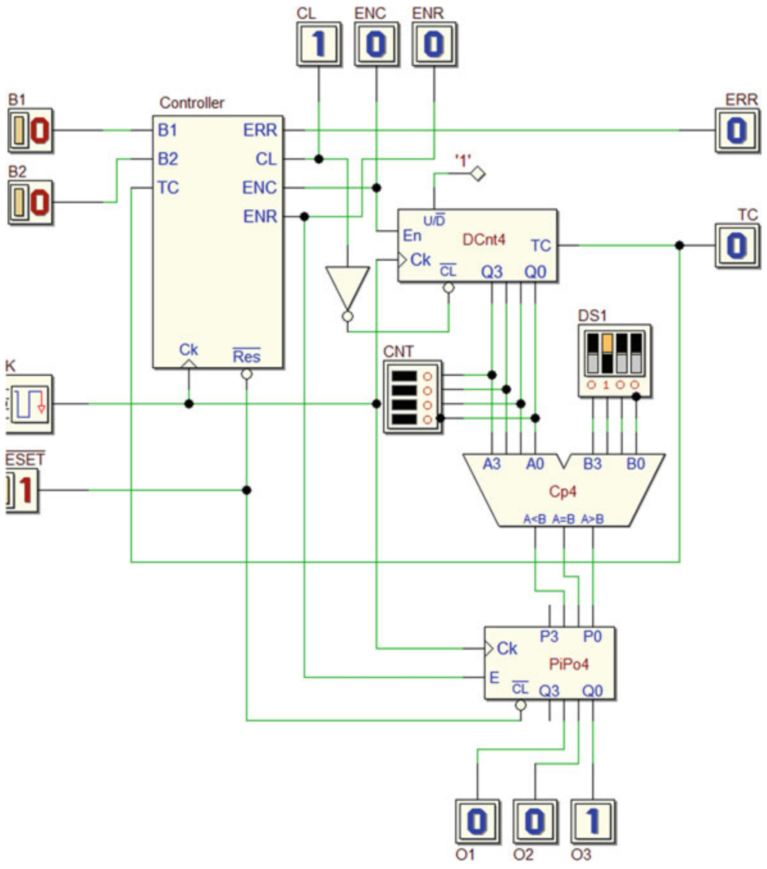
The counter is used by the controller to count the number of transmitted bits in order to end operations after the packet is transmitted.

Define the number *P*3..*P*0 required by your specific project (in the figure, 0111_2 is just an indication).



t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
CK
Reset
P3..P0
D7..D0
D0 = 1, D1 = 1, D2 = 0, D3 = 1, D4 = 1, D5 = 0, D6 = 0, D7 = 1
GO
Controller
RDY
LD
ENS
Q0
LC
CNT
.0
.1
.2
.3
TC
D
SER

Exercise 9 The system shown in the figure below is made up of a controller, a counter, a comparator, and a parallel register.

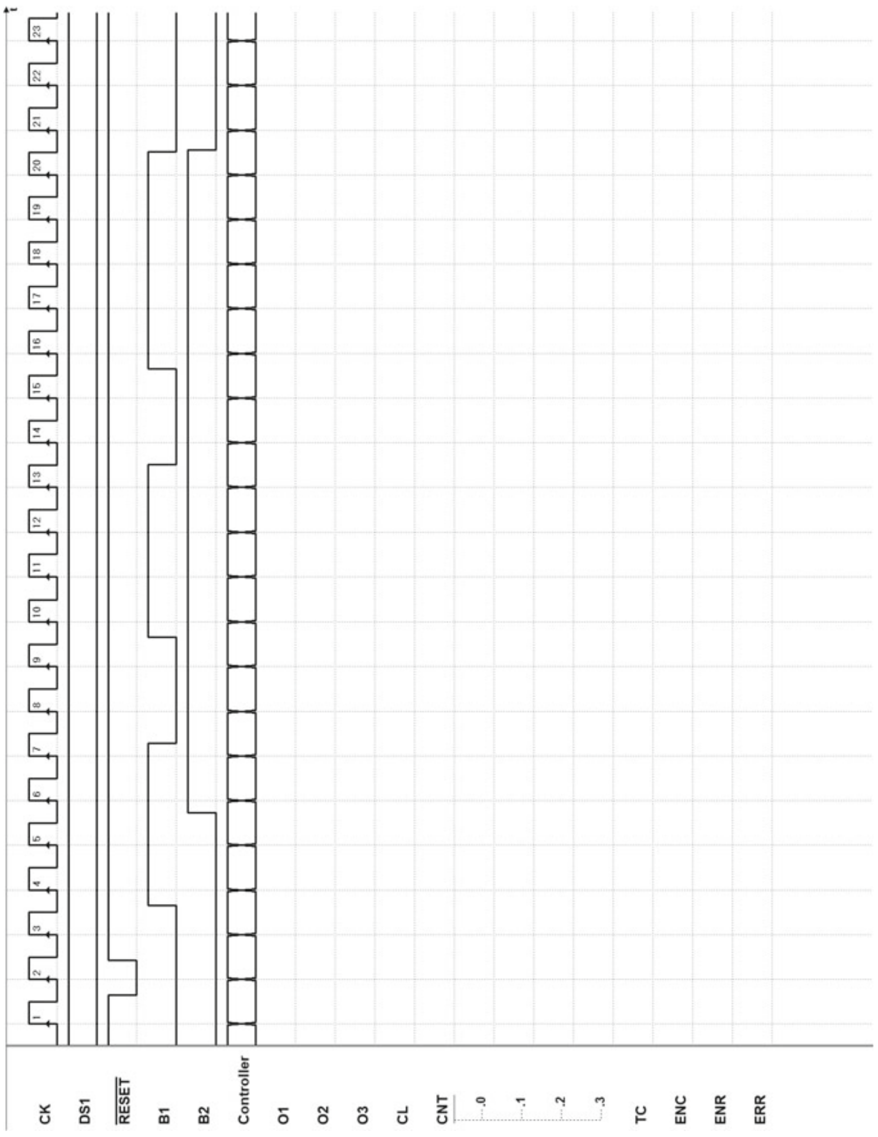


The system's inputs are push-buttons *B1* and *B2* (when pressed are at 1). The FSM checks how long they have been pressed at the same time.

Specifically:

- It activates output $O1$ if $B1$ and $B2$ are pressed at the same time for less than four clock cycles.
- It activates output $O2$ if $B1$ and $B2$ are pressed at the same time for exactly four clock cycles.
- It activates $O3$ if $B1$ and $B2$ are pressed at the same time for more than four clock cycles.

If $B1$ and $B2$ are pressed for more than 16 clock cycles, the system activates *ERR* until the two push-buttons are both released.


t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
CK
DS1
RESET
B1
B2
Controller
O1
O2
O3
CL
CNT
.0
.1
.2
.3
TC
ENC
ENR
ERR

8.4.2 Design of a Controller–Datapath System

To carry out a design of this type, as suggested on p. 397, it is advisable to follow these steps:

(a) Define the system as a functional block, highlighting inputs and outputs.
(b) Choose the components to use in the datapath.
(c) Design the datapath with the connections between the controller and the datapath components and the inputs and outputs of the whole system.
(d) Draw the ASM diagram of the FSM that describes the controller.
(e) Perform a timing simulation of the whole system by choosing input sequences that help demonstrate the system's functionality set.

Exercise 1

Design a *serial transmitter* of the measure of a *pulse duration*, by using the *controller–datapath* structure.

The system waits for line *IM* to go to 1 and then begins to count how many clock cycles the signal *IM* stays at 1.

At the first clock edge after *IM* goes back to zero, the system sends a 6-bit packet on the *PKG* channel. It consists of a *start bit* at 1, *four data bits* that represent the pulse duration in terms of clock cycles (an error of $+/-1$ is tolerated), and a *stop bit* at 0.

If the pulse lasts longer than 15 clock cycles, the machine activates output *ERR* until *IM* goes back to zero. It sends no data pack but waits for the next pulse.

(Design tips on p. 427)

Exercise 2

Design a *kitchen timer* by using the *controller–datapath* structure.

When push-button *ST* (at 1 when pressed) is released, the device waits for the number of clock cycles externally set on inputs *P*7..*P*0 (eight bits) and then activates an output *BEP* for *three cycles*.

The clock's frequency is 1*Hz*, and the time must be controlled between 2 and 256 seconds. The actual time can differ by a maximum of one second from the defined time.

(Design tips on p. 428)

Exercise 3

Design a *serial–parallel converter* by using the *controller–datapath* architecture.

The system has an input *SER* and outputs *D*0..*D*5 and *RDY*. It waits for the serial data in the format {1, *D*0, *D*1 .. *D*5, 0} and transfers it into parallel format on the six outputs *D*0..*D*5. If the *stop bit* check is positive, it activates *RDY* and keeps it until it receives new serial data.

If the *stop bit* is incorrect, the system simply goes back to waiting for new serial data.

(Design tips on p. 429)

Exercise 4

Design a *serial–serial repeater* by using the *controller–datapath* structure.

On its input *IN*, the system receives a serial packet in the format {1, *B*0, *B*1, *P*, 0} where *P* represents the *parity* (XOR function) of *B*0 and *B*1.

The system acquires *B*0 and *B*1 on two JK-PET flip-flops and uses an XOR gate to check that the parity of *B*0 and *B*1, which are saved on the flip-flops, corresponds to bit *P* of the serial packet, which was transmitted. If it does correspond and the stop bit is correct, it transmits the pack {1, *B*0, *B*1, 0} to output *OUT*.

If the parity check is negative or the stop bit is incorrect, there is no transmission and the system goes back to waiting for another input packet.

(Design tips on p. 430)

Exercise 5

Design a *parallel–serial converter* by using the *controller–datapath* architecture.

The system has five inputs (*GO* and *D*0..*D*3) and three outputs (*SER*, *BSY* and *RDY*). When *GO* appears, the system generates a serial packet made up of: {1, *D*0, *D*1, *D*2, *D*3, and 0}.

Signals *D*0..*D*3 are only guaranteed to have the correct value when *GO* is received. *GO* lasts for one clock period.

The system activates *BSY* during the transmission of the packet and generates a pulse with a duration of one clock cycle on *RDY* just after the packet is transmitted.

(Design tips on p. 431)

Exercise 6

Design a digital system that works as an *accumulator* that generates, on the output, the sum (4 bits) of the current number and the one at the input.

The system has five inputs:

*D*2..*D*0	The data the accumulator must add (3 bits with no sign).
NEG	If active, it commands to change the sign of the number on D2..D0 before adding it.
GO	If active, it commands the execution of the sum.

The system has six outputs:

*Q*3..*Q*0	The content of the accumulator.
OVF	To be activated in case of *Overflow*.
CO	To be activated in case of *Carry*.

(Design tips on p. 432)

Exercise 7

Design a *controller for a microwave oven* by using the *controller–datapath* structure.

The system has two 1-bit inputs and one 8-bit input.
Inputs:

GO	Starts cooking for time *TCC*
TCC	8-bit input to set cooking time
OPN	Active while the door is open

Outputs:

COK	When active, the oven heats
BEL	Signals the end of cooking time and lasts one clock cycle

The user sets the cooking time and presses the push-button. The signal *GO*, which is active for one clock cycle, starts the cooking for time *TCC*.

If the door is open, the oven does not work; when it is closed, it begins to heat. If the door is opened before the end of the set cooking time, it stops cooking and starts again when the door is closed.

(Design tips on p. 433)

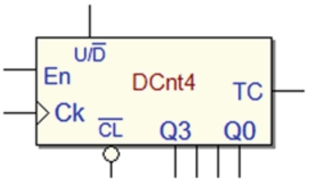
8.5 Design Tips

8.5.1 Design of a Controller–Datapath System

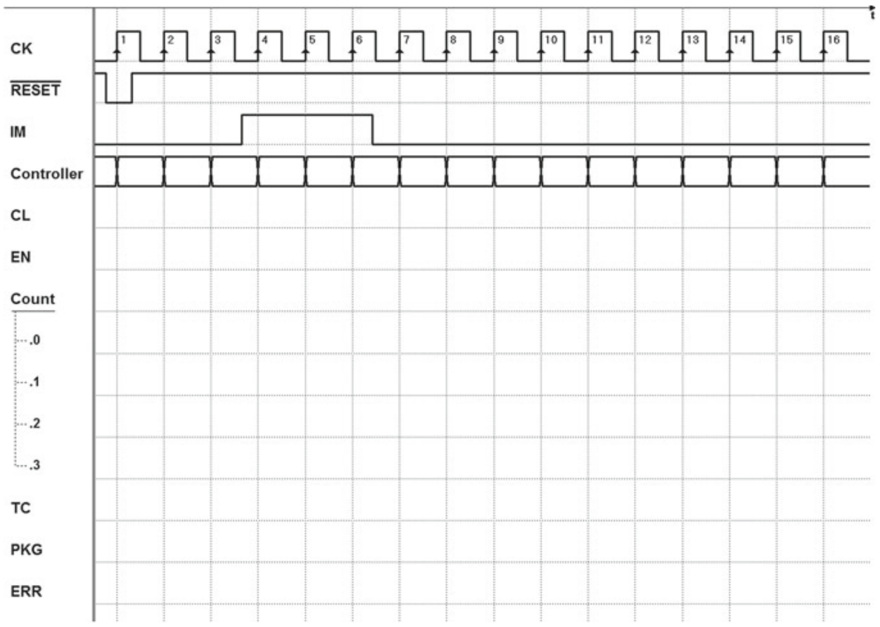
Here, you will find some hints to help you complete the design exercises starting on p. 424.

Exercise 1

We recommend using the counter “DCnt4” and logical gates of your choice.

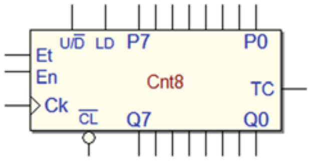


We recommend defining a timing trace like this one below:

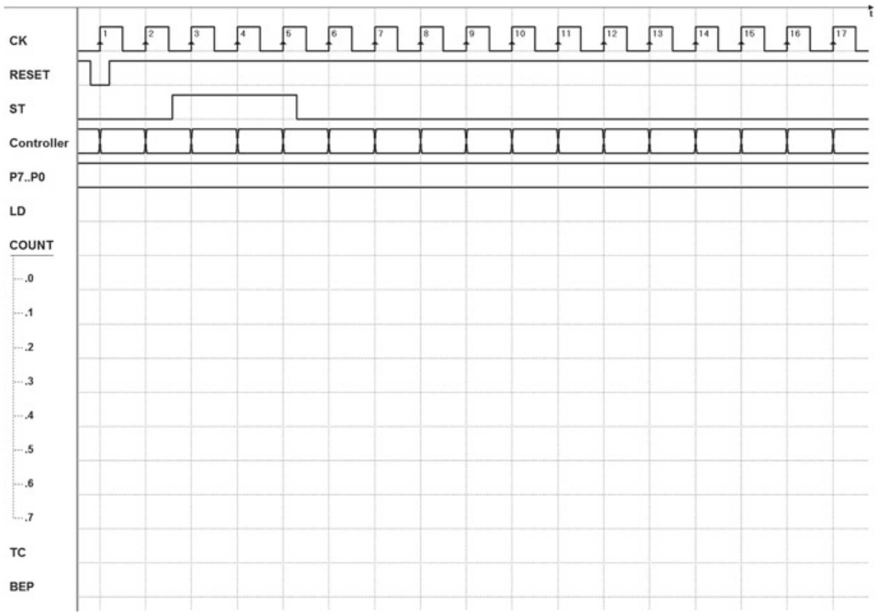


Exercise 2

You may use the counter "DCnt8" and any logical gate.

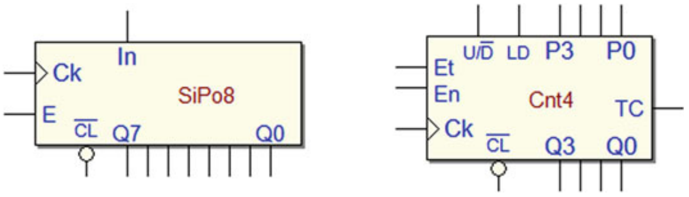


We recommend defining a timing trace like this one below:

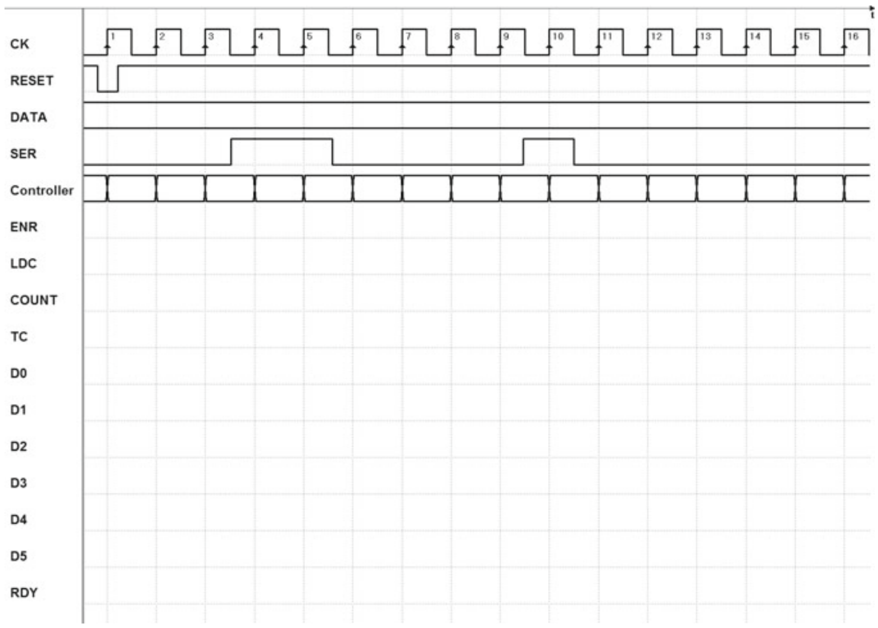


Exercise 3

We suggest using the components “Sipo8” and “Cnt4” as well as any logical gate.

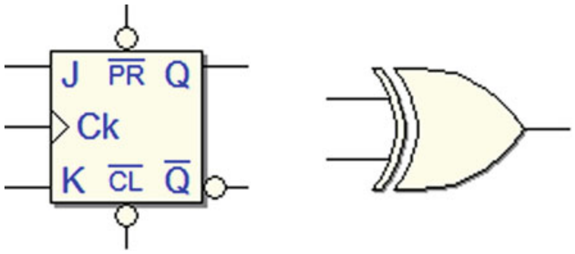


We recommend defining a timing trace like this one below:

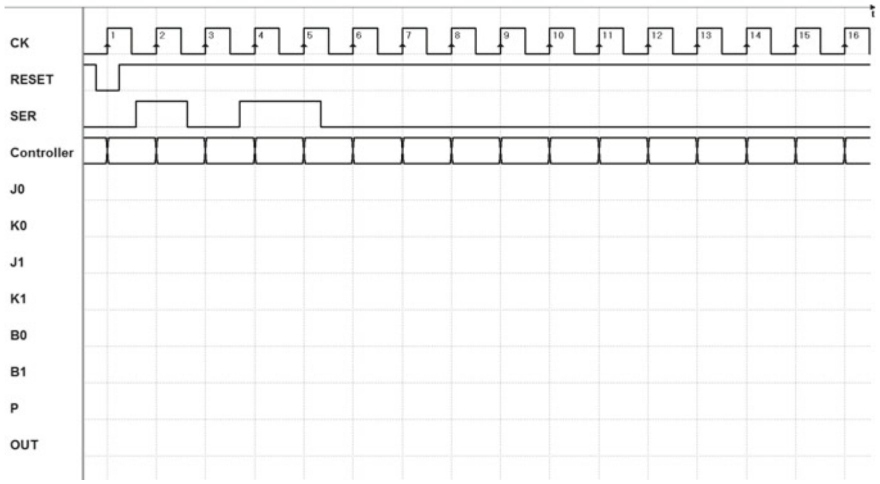


Exercise 4

We suggest using two JK flip-flops and an XOR gate for the datapath.

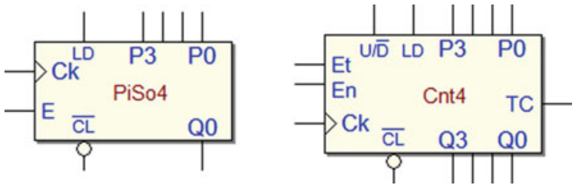


We recommend defining a timing trace like this one below:

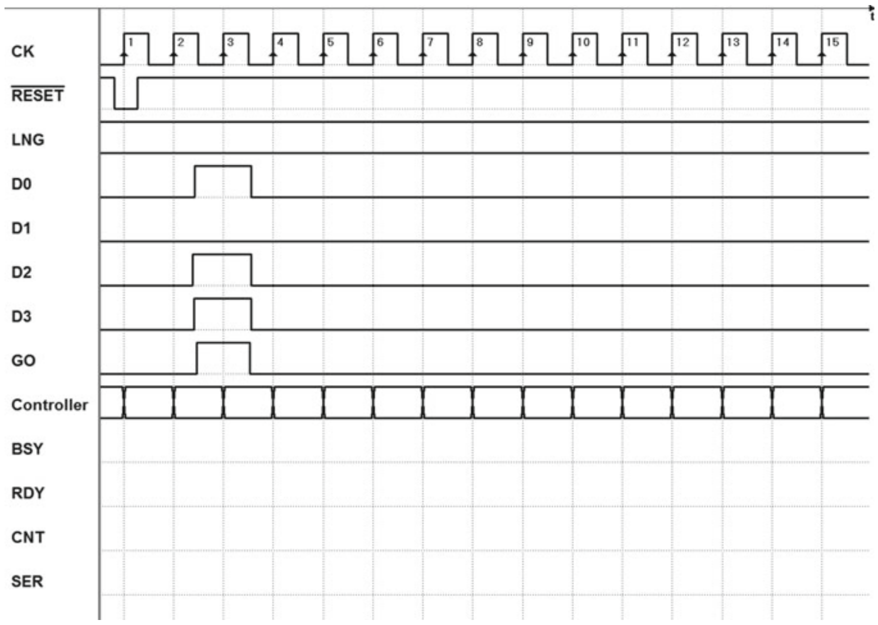


Exercise 5

For the datapath, we recommend using the components shown below (keep in mind that we can also design the system without a counter).

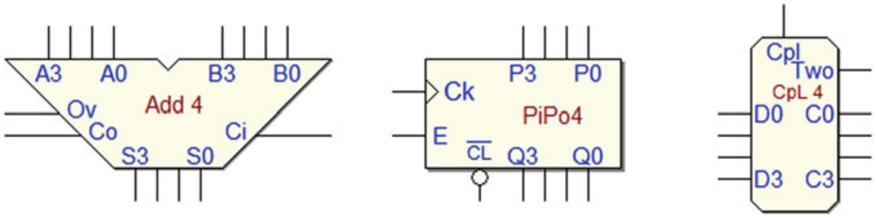


We recommend defining a timing trace like this one below:

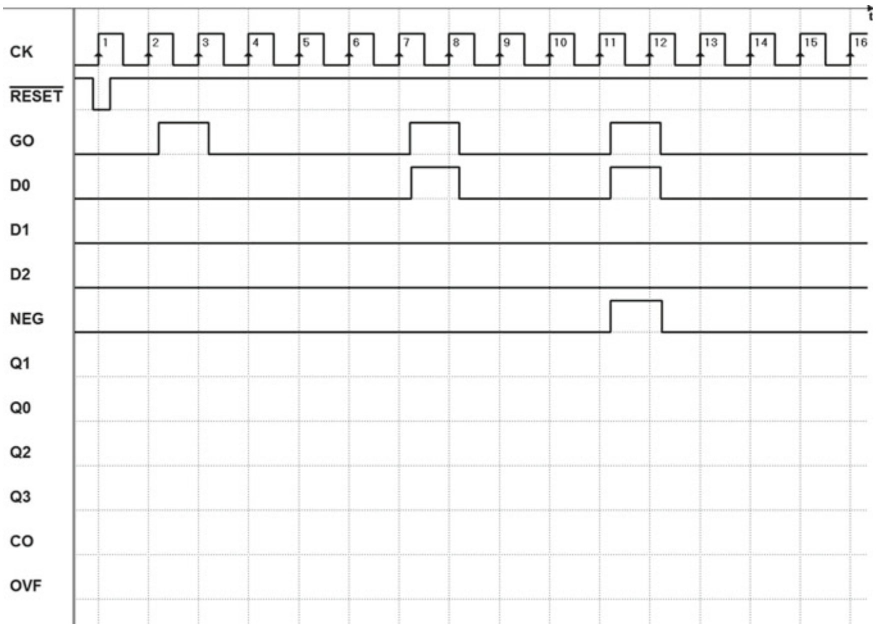


Exercise 6

We recommend using the components shown below:

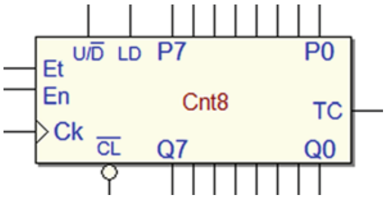


We recommend defining a timing trace like this one below:

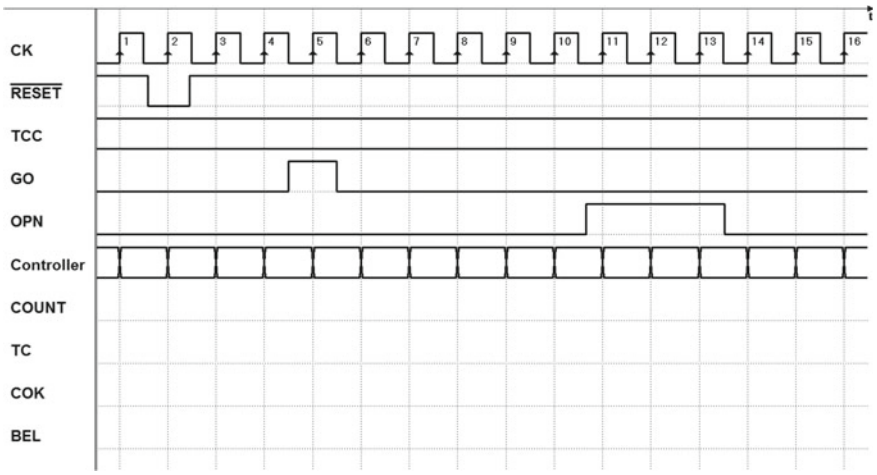


Exercise 7

For the datapath, we recommend using the component shown below:



We recommend defining a timing trace like this one below:

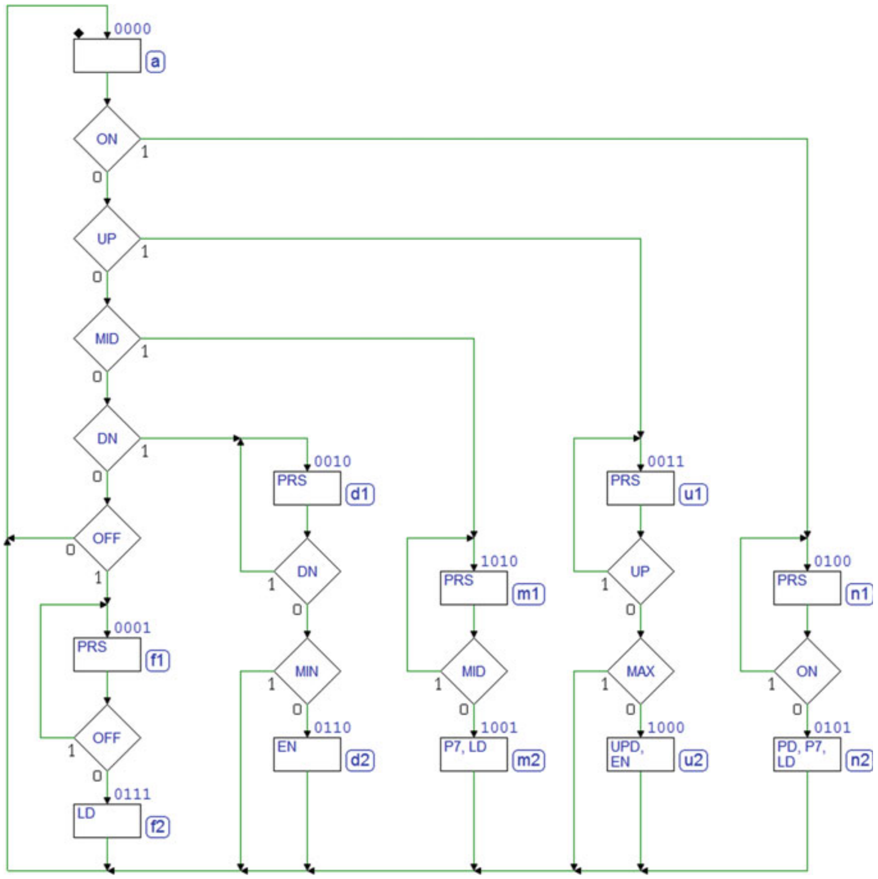


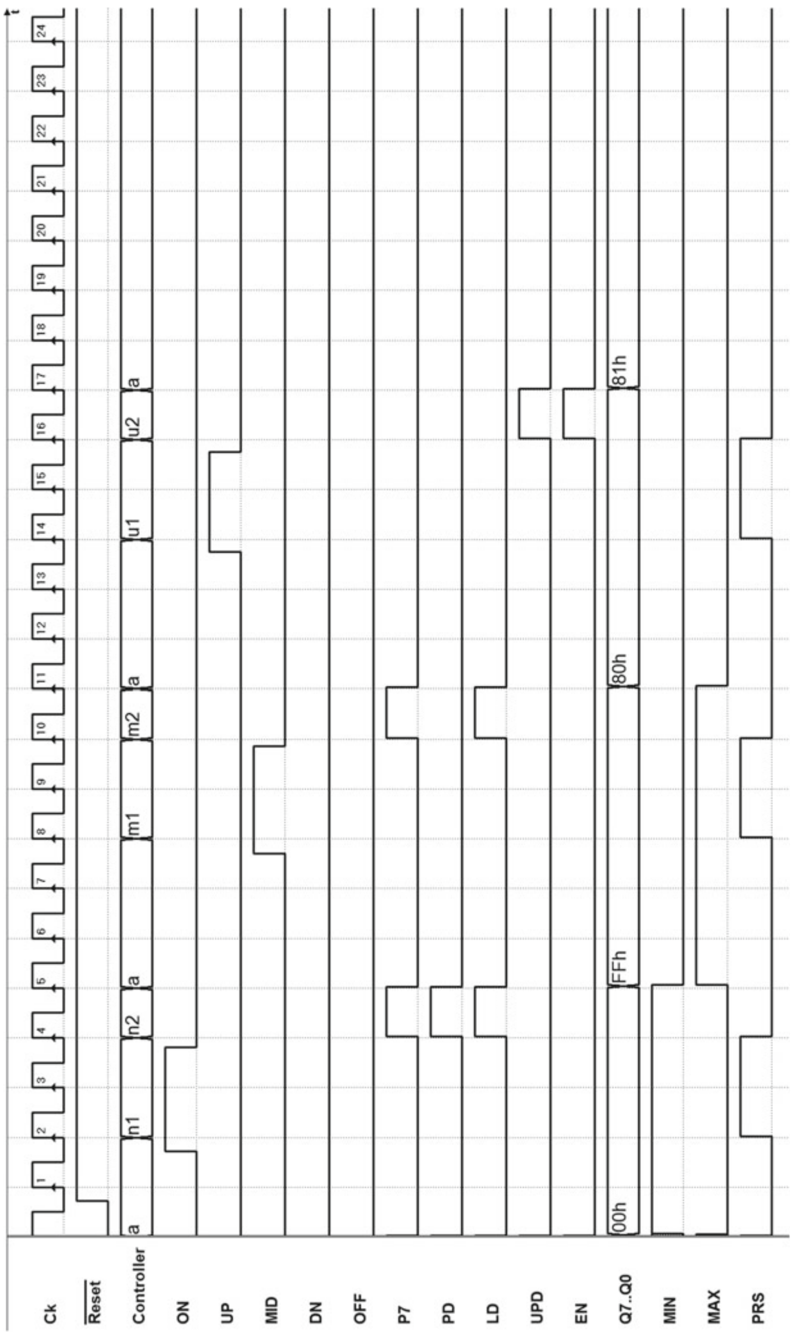
8.6 Solutions

8.6.1 Design of the Controller of a Given Datapath

We can download the MSF files shown here and their complete circuit schematics from the Web site of *Deeds*. It will be useful to analyze the solutions by using the timing simulation.

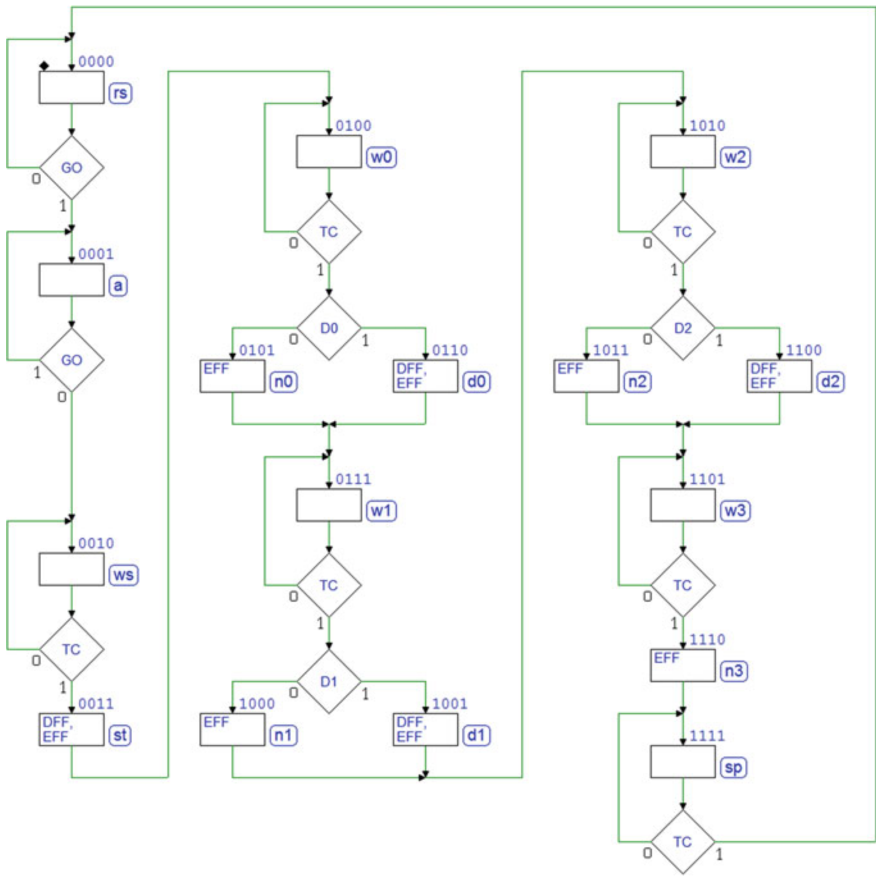
Solution of Exercise 1:

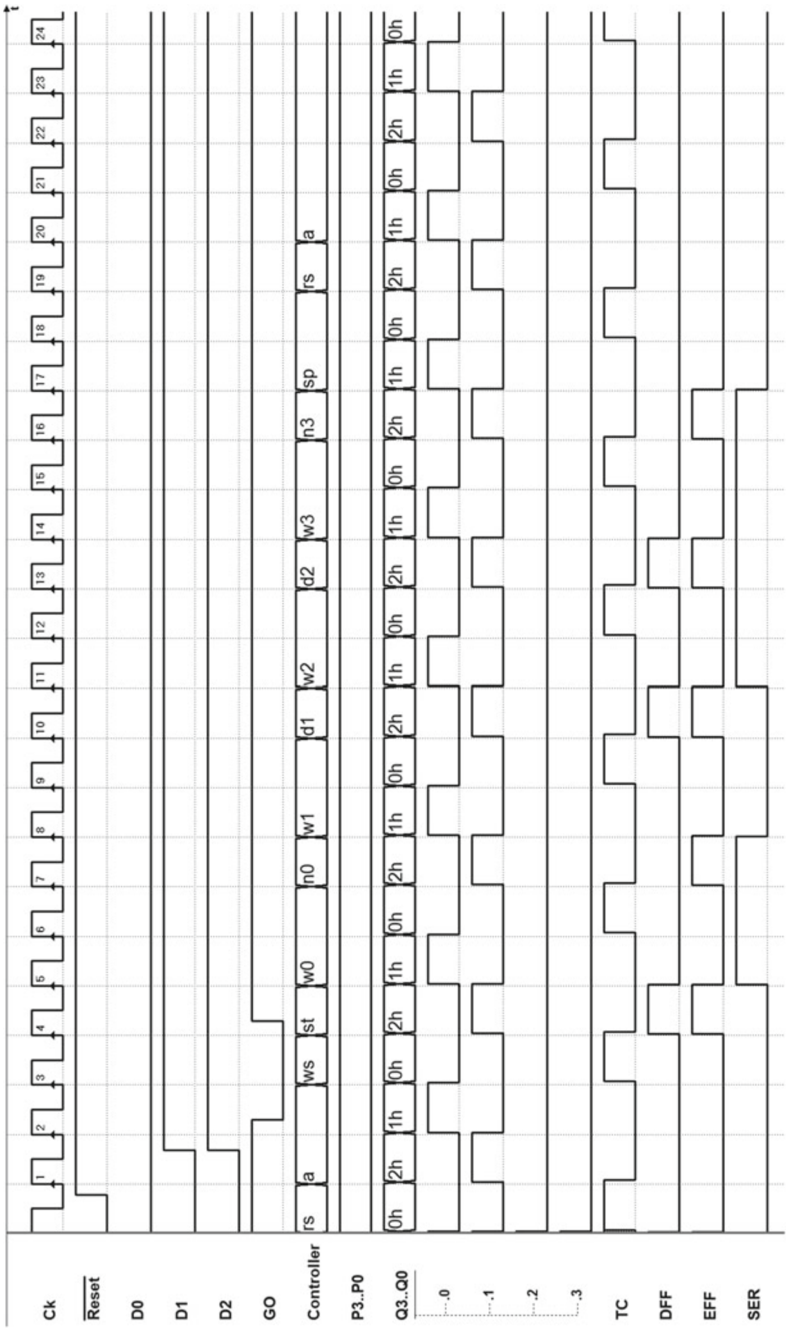




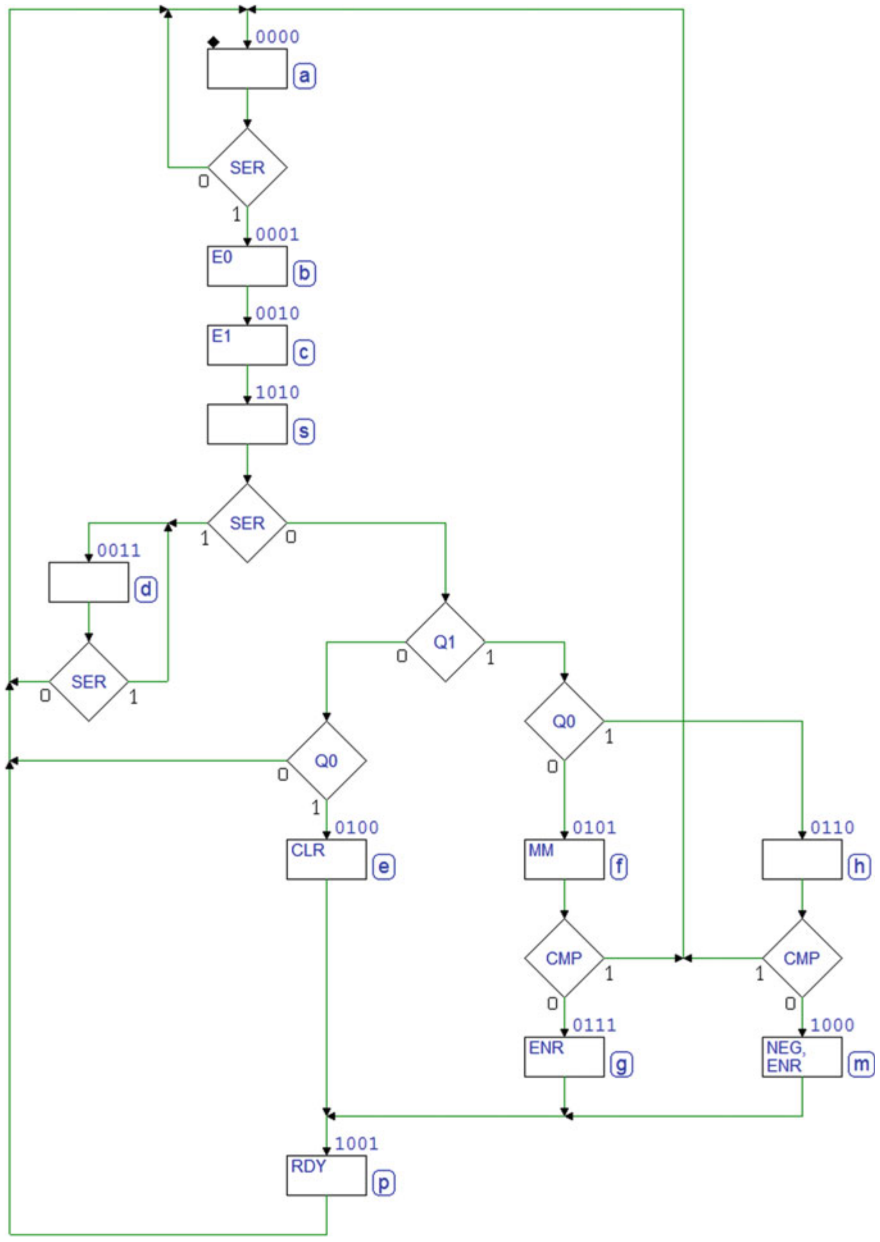
Ck
Reset
Controller
ON
UP
MID
DN
OFF
P7
PD
LD
UPD
EN
Q7..Q0
MIN
MAX
PRS
t
a
n1
n2
a
m1
m2
a
u1
u2
a
00h
FFh
80h
81h

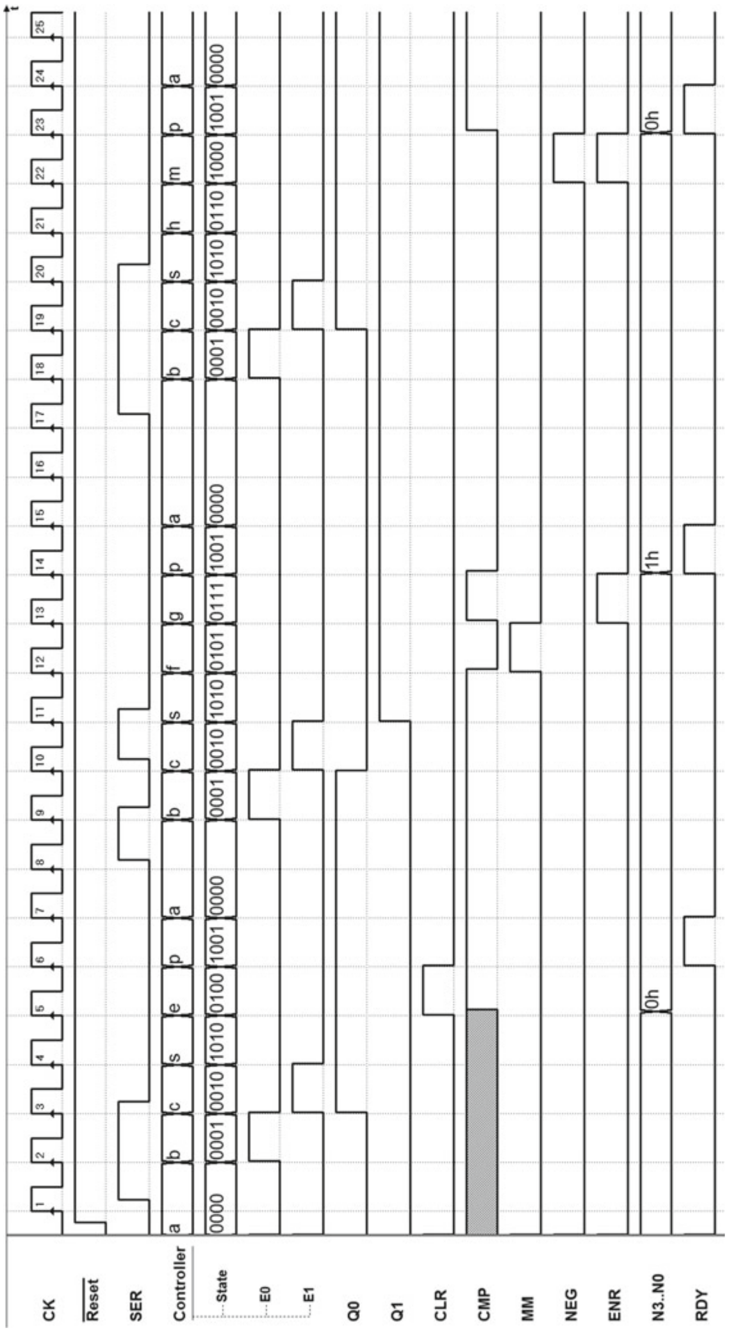
Solution of Exercise 2:



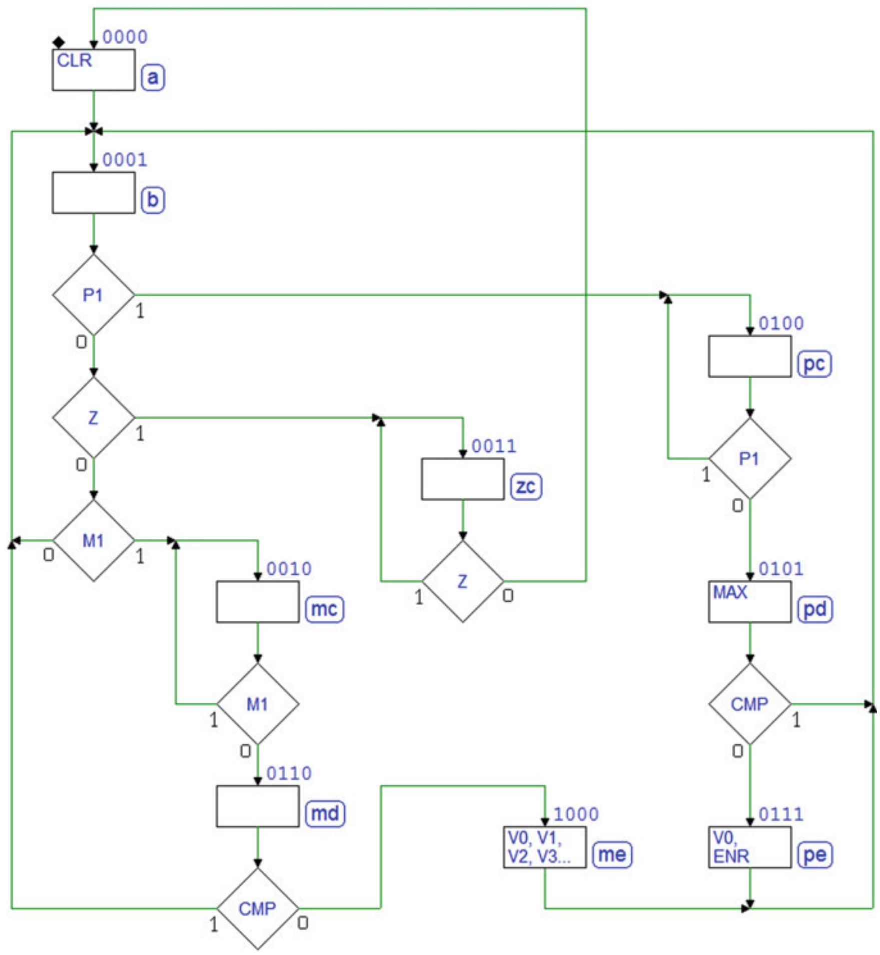

Ck
Reset
D0
D1
D2
GO
Controller
P3..P0
Q3..Q0
.0
.1
.2
.3
TC
DFF
EFF
SER
t
rs
a
ws
st
w0
n0
w1
d1
w2
d2
w3
n3
sp
rs
a

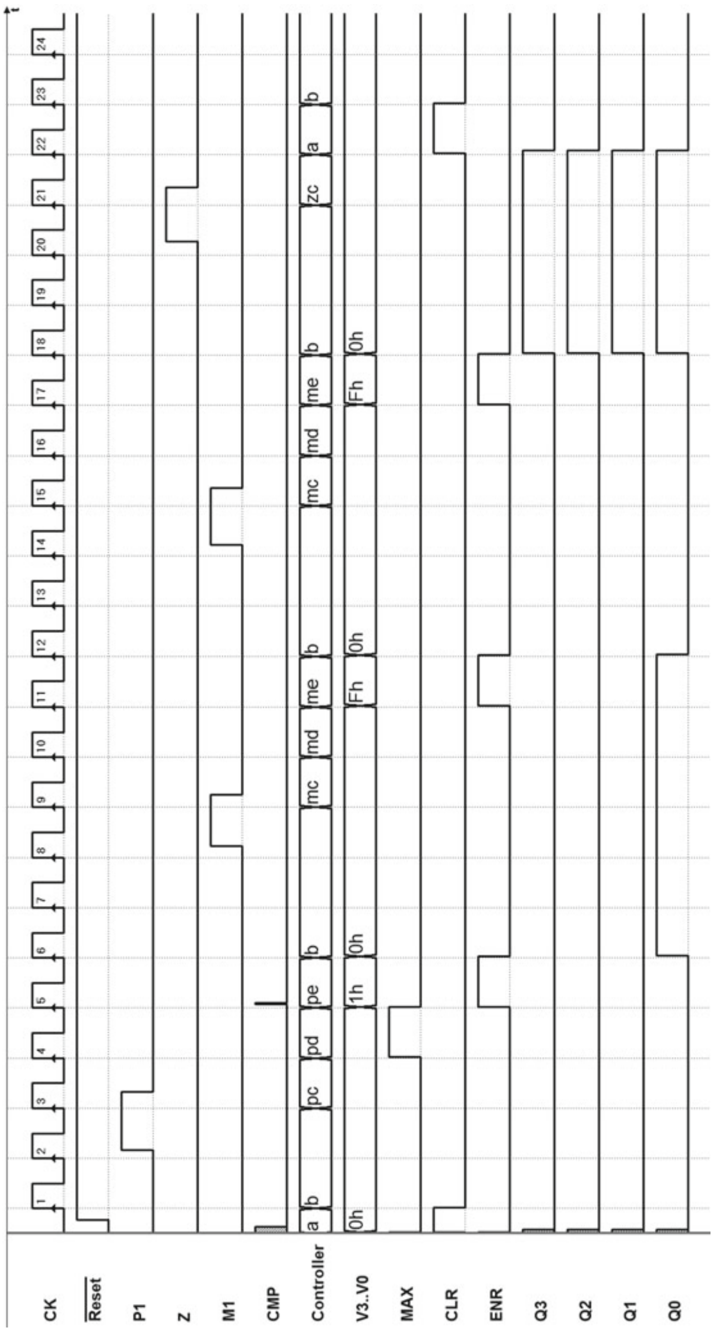
Solution of Exercise 3:



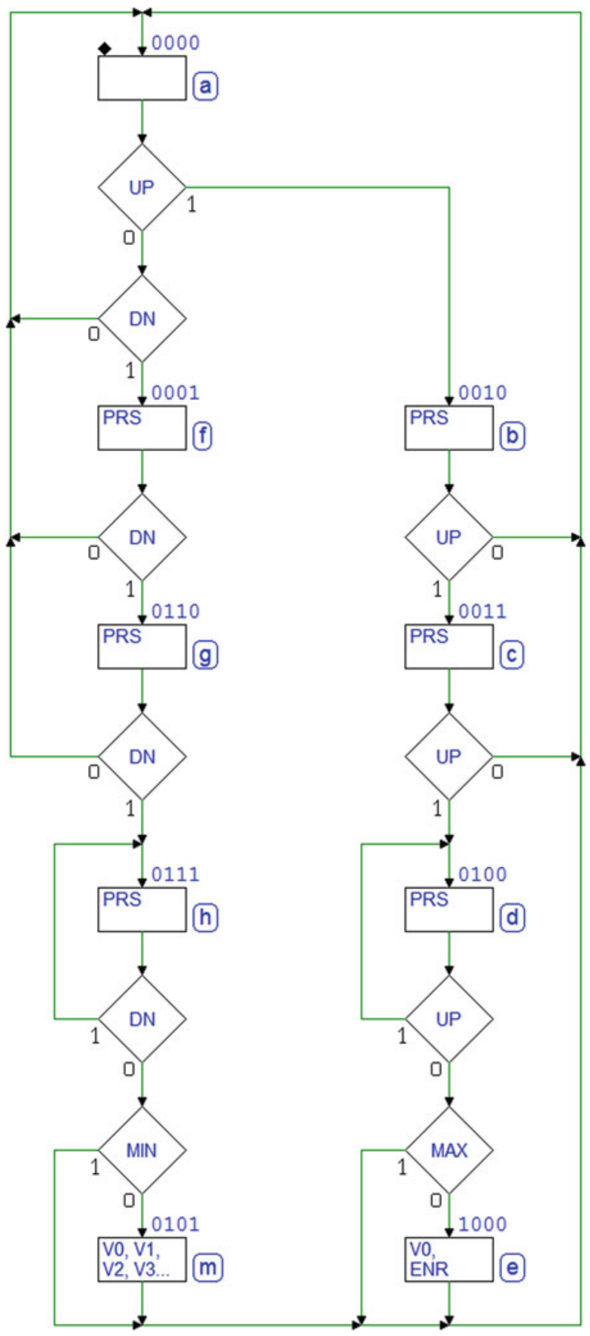

CK
Reset
SER
Controller
State
E0
E1
Q0
Q1
CLR
CMP
MM
NEG
ENR
N3..N0
RDY
t
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
a b c s e p a b c s f g p a b c s h m p a
0000 0001 0010 1010 0100 1001 0000 0001 0010 1010 0101 0111 1001 0000 0001 0010 1010 0110 1000 1001 0000
0h 1h 0h

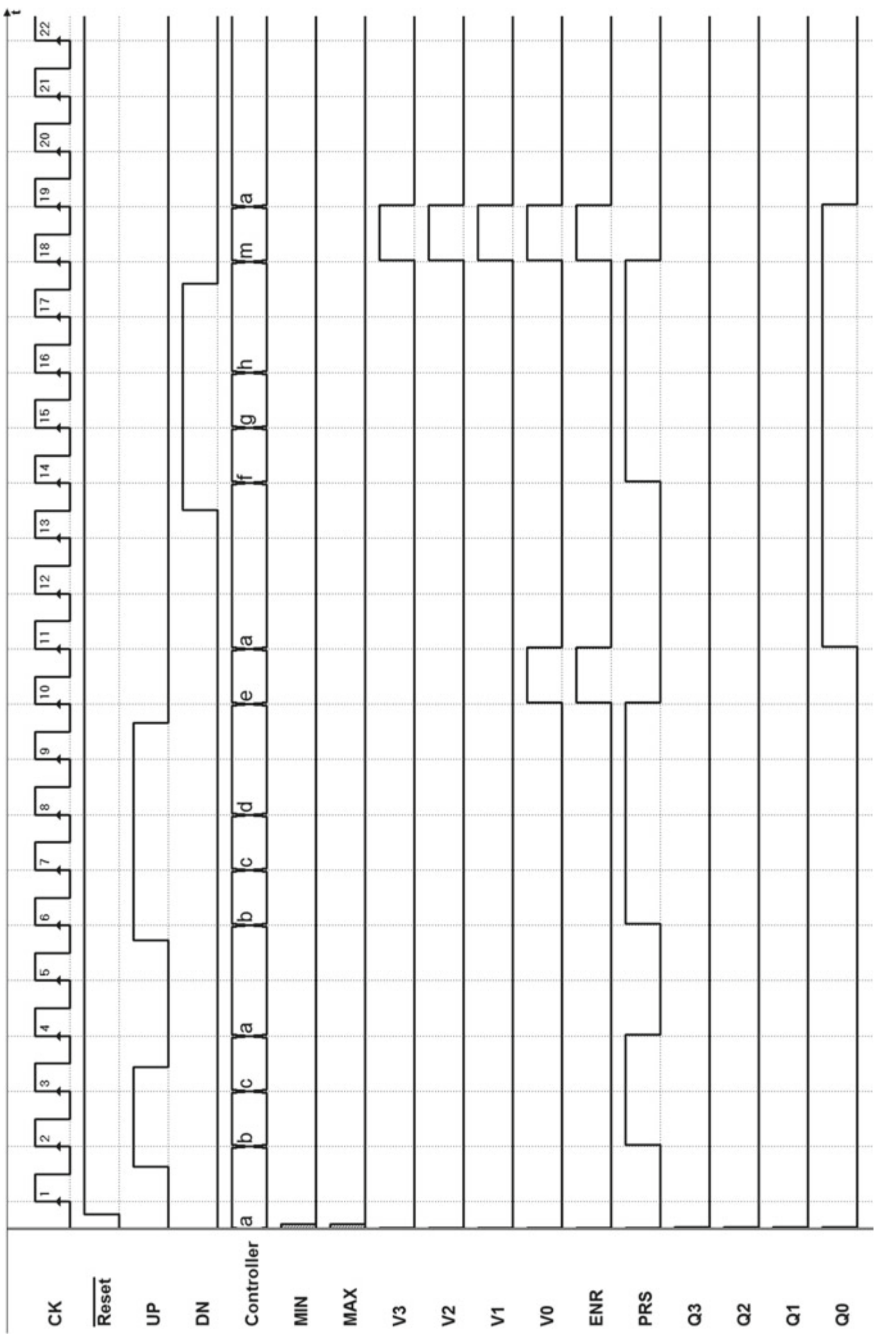
Solution of Exercise 4:



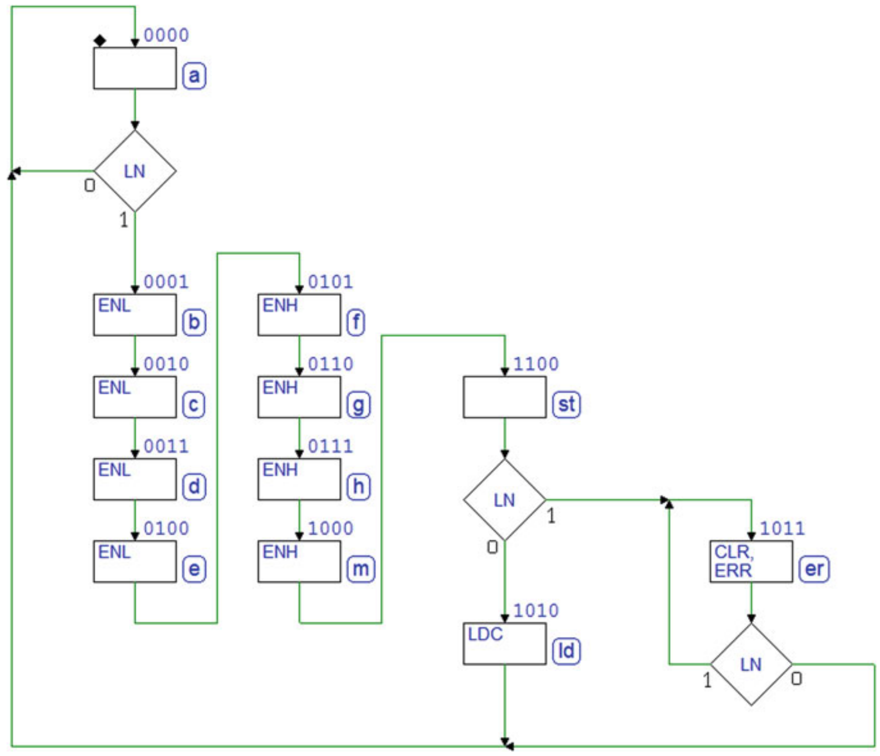

CK
Reset
P1
Z
M1
CMP
Controller
V3..V0
MAX
CLR
ENR
Q3
Q2
Q1
Q0
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
a
b
pc
pd
pe
b
mc
md
me
b
mc
md
me
b
zc
a
b
0h
1h
0h
Fh
0h
Fh
0h

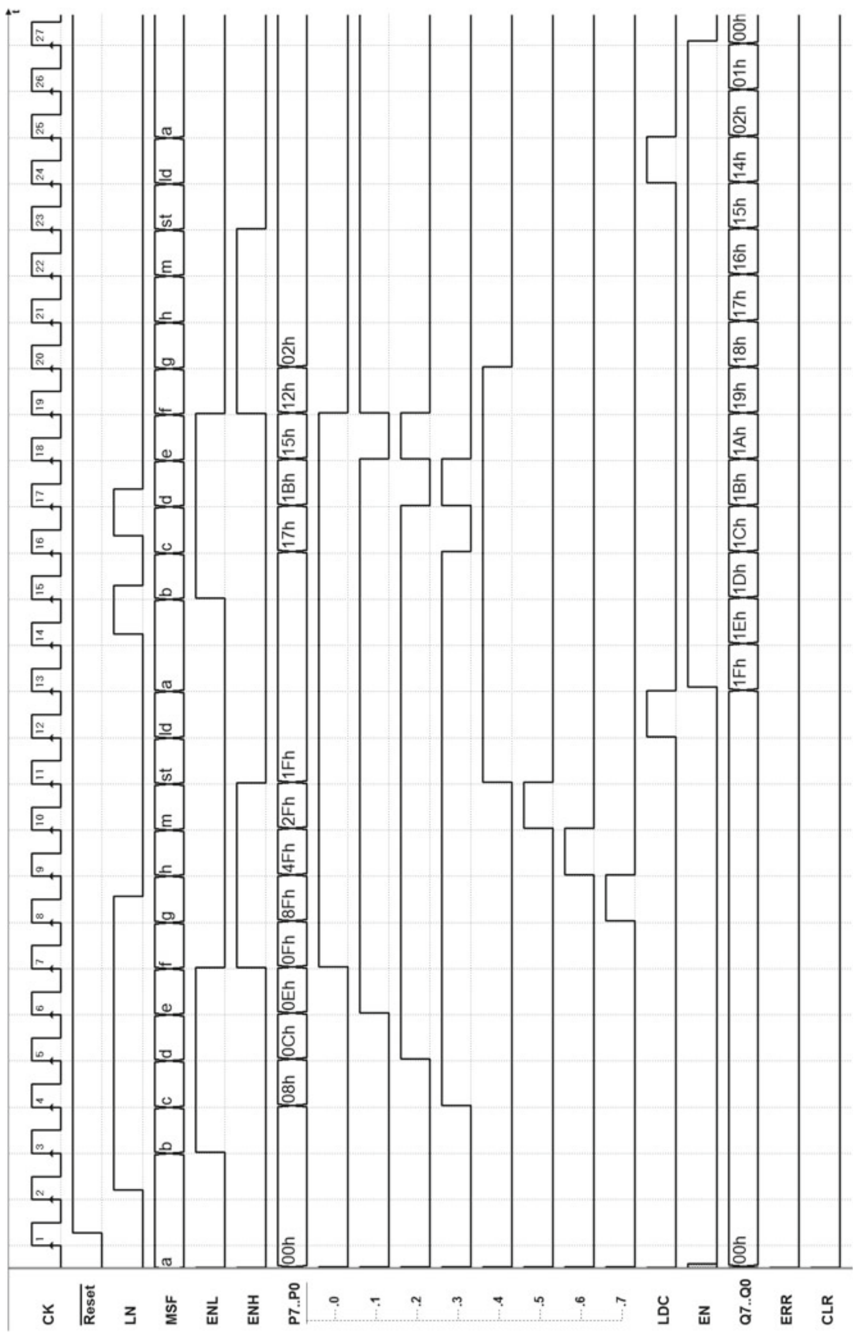
Solution of Exercise 5:



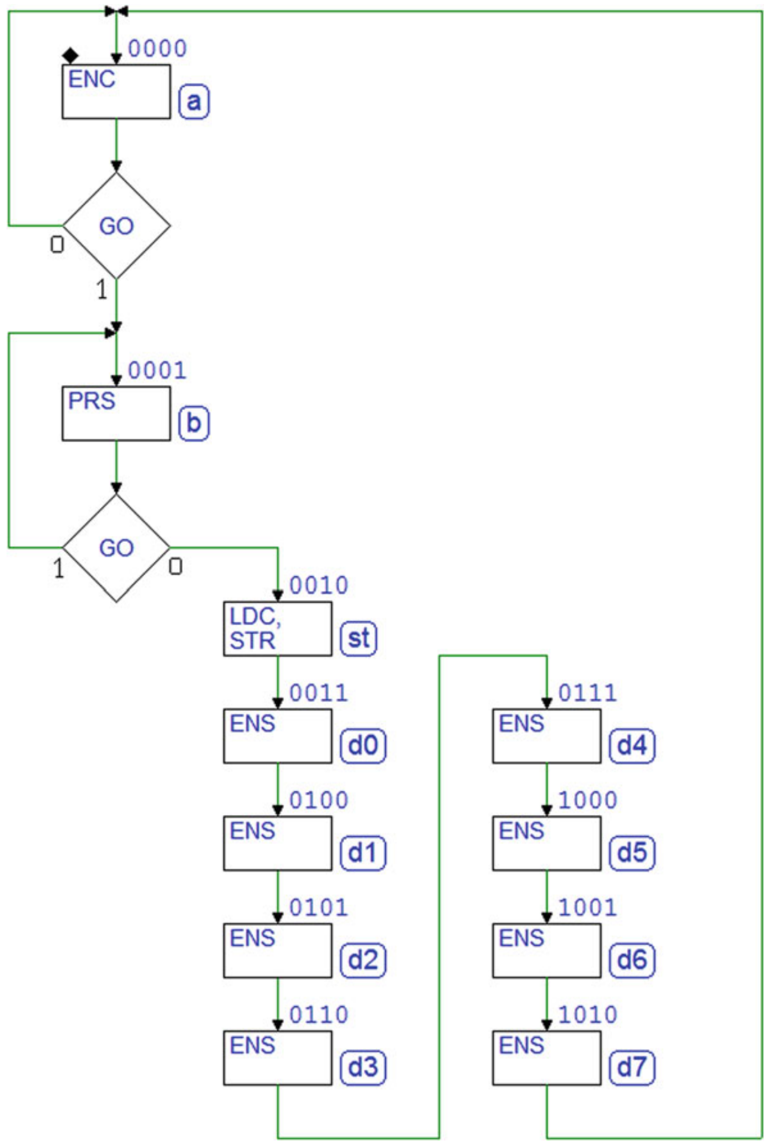

t
CK
Reset
UP
DN
Controller
MIN
MAX
V3
V2
V1
V0
ENR
PRS
Q3
Q2
Q1
Q0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
a b c a b c d e a f g h m a

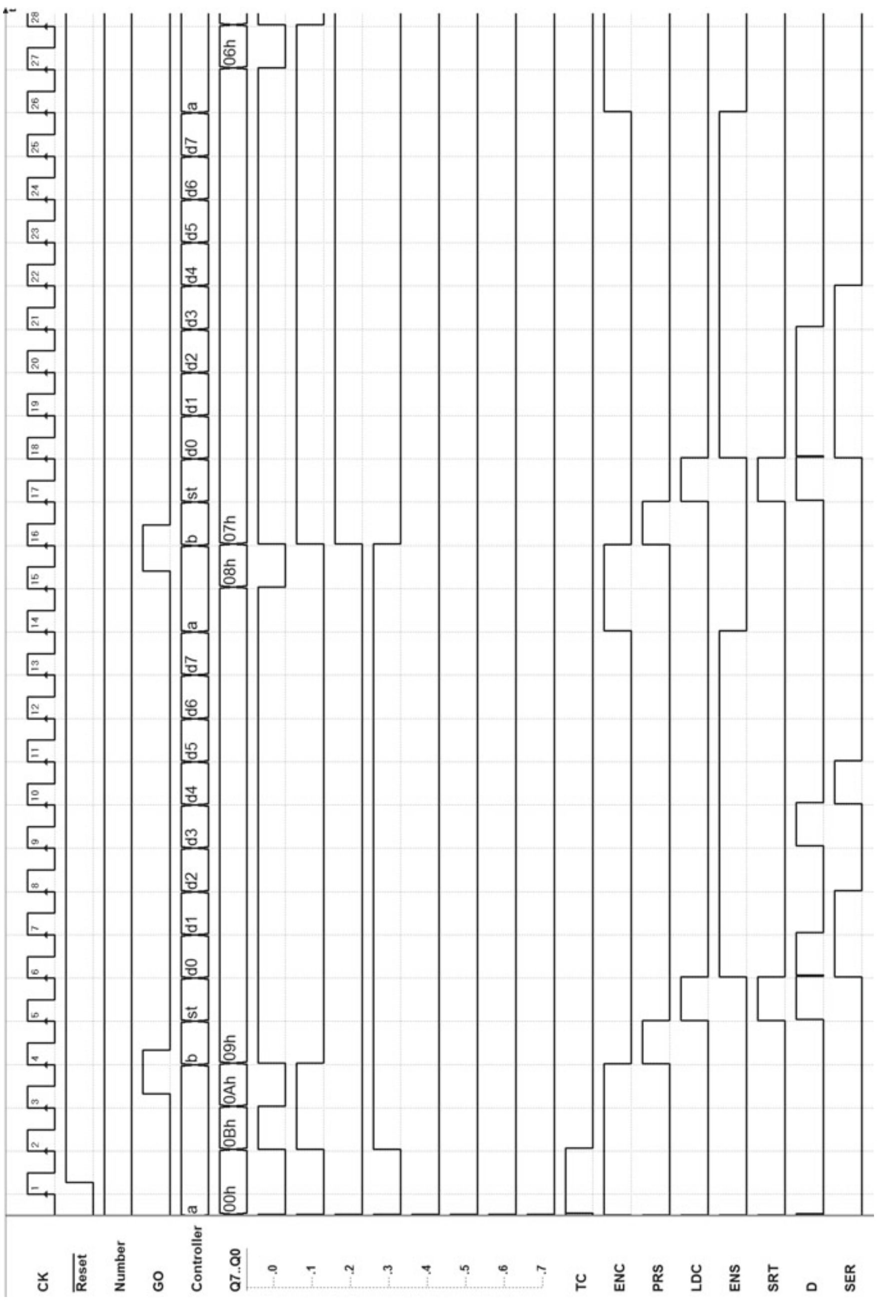
Solution of Exercise 6:



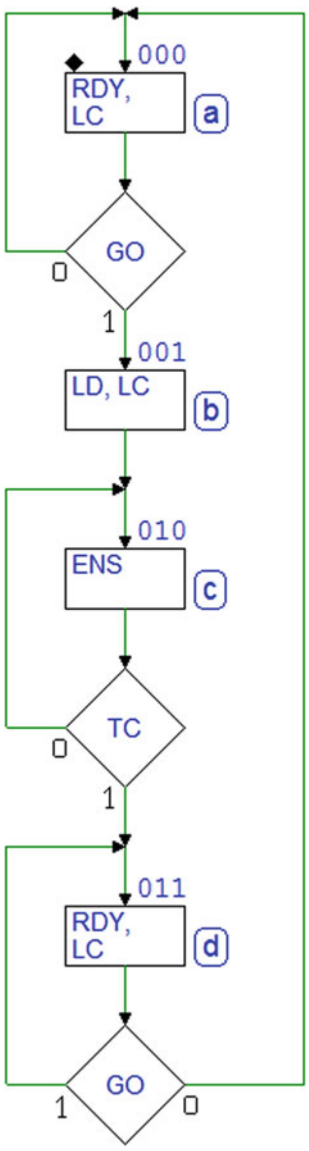

CK
Reset
LN
MSF
ENL
ENH
P7..P0
.0
.1
.2
.3
.4
.5
.6
.7
LDC
EN
Q7..Q0
ERR
CLR
t
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
a b c d e f g h m st ld a b c d e f g h m st ld a
00h 08h 0Ch 0Eh 0Fh 8Fh 4Fh 2Fh 1Fh 17h 1Bh 15h 12h 02h
00h 1Fh 1Eh 1Dh 1Ch 1Bh 1Ah 19h 18h 17h 16h 15h 14h 02h 01h 00h

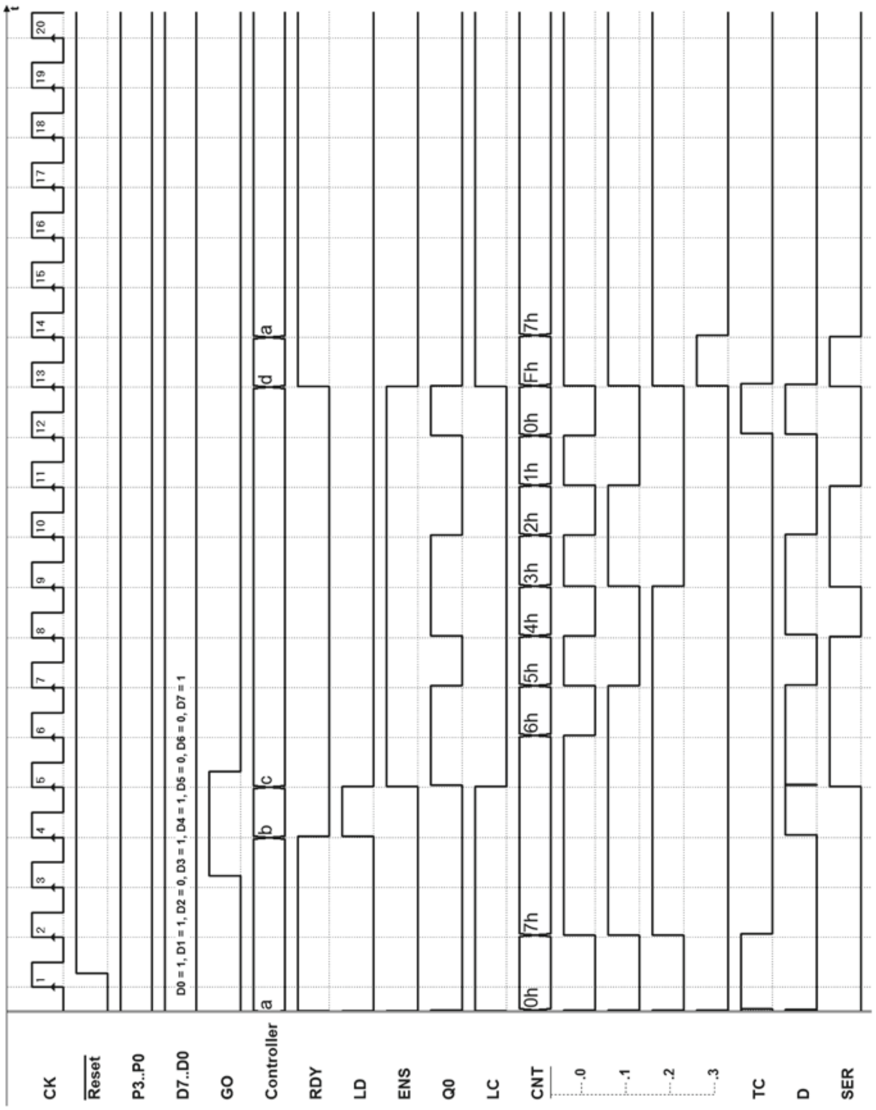
Solution of Exercise 7:




CK
Reset
Number
GO
Controller
Q7..Q0
.0
.1
.2
.3
.4
.5
.6
.7
TC
ENC
PRS
LDC
ENS
SRT
D
SER
a
b
st
d0
d1
d2
d3
d4
d5
d6
d7
a
b
st
d0
d1
d2
d3
d4
d5
d6
d7
a
00h
0Bh
0Ah
09h
08h
07h
06h
t

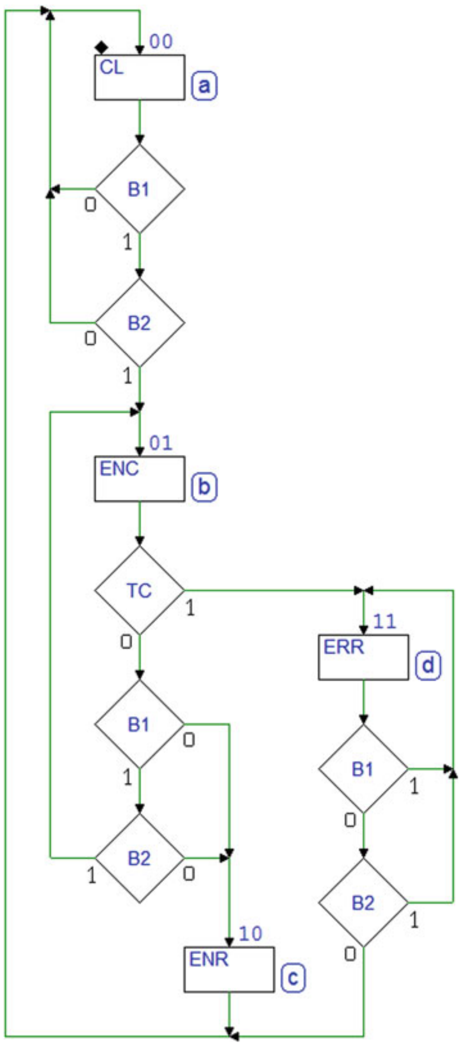
Solution of Exercise 8:

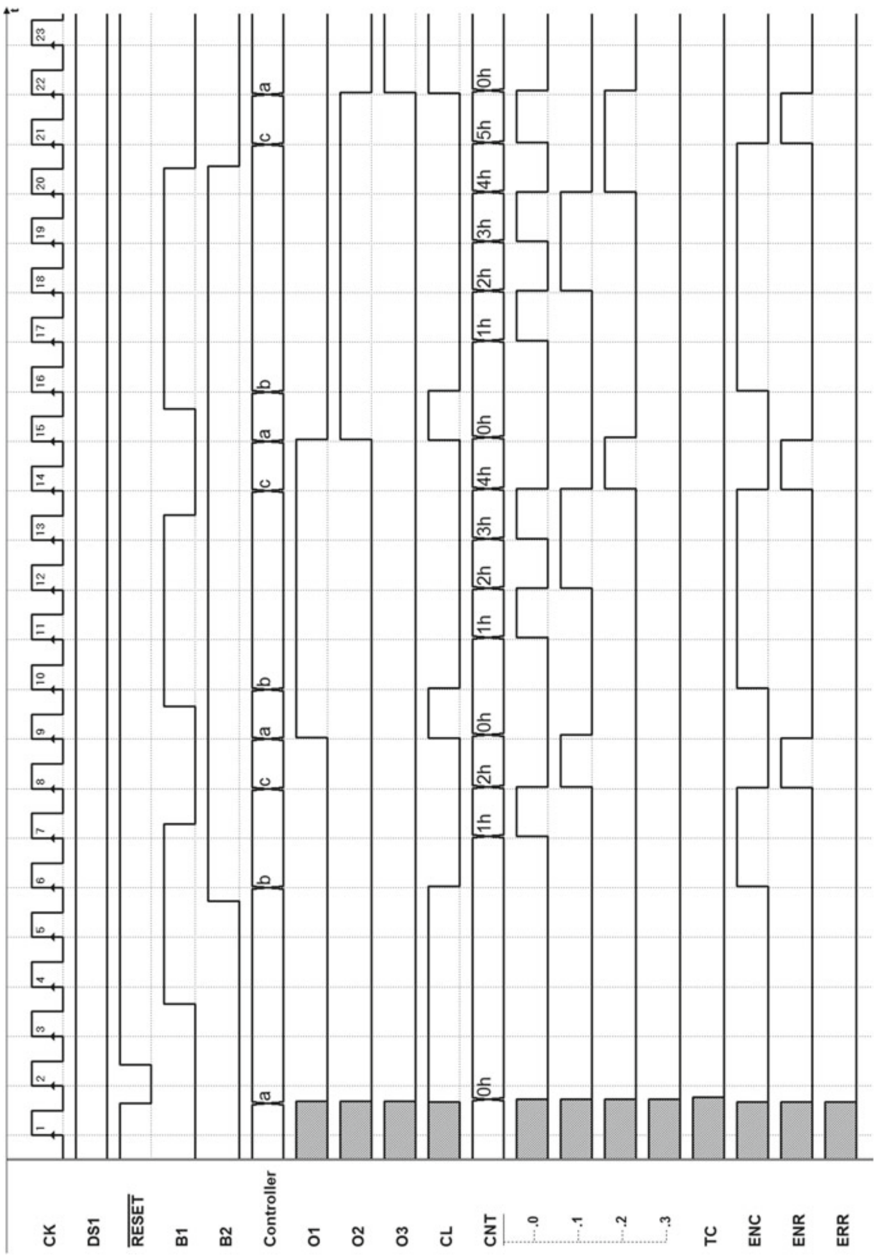




CK
Reset
P3..P0
D7..D0
GO
Controller
RDY
LD
ENS
Q0
LC
CNT
.0
.1
.2
.3
TC
D
SER
t
D0 = 1, D1 = 1, D2 = 0, D3 = 1, D4 = 1, D5 = 0, D6 = 0, D7 = 1
a
b
c
d
a
0h
7h
6h
5h
4h
3h
2h
1h
0h
Fh
7h

Solution of Exercise 9:



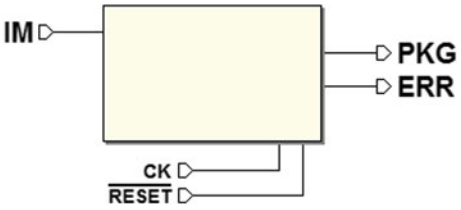

CK
DS1
RESET
B1
B2
Controller
O1
O2
O3
CL
CNT
.0
.1
.2
.3
TC
ENC
ENR
ERR
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
a
b
c
a
b
c
a
b
c
a
0h
1h
2h
0h
1h
2h
3h
4h
0h
1h
2h
3h
4h
5h
0h

8.6.2 Design of a Controller–Datapath System

The Web site of *Deeds* has all the files corresponding to the figures shown here (circuit schematics and FSMs) so that the solutions can be checked by simulation.

Solution of Exercise 1

The system's inputs and outputs:



The network schematic (controller + datapath):

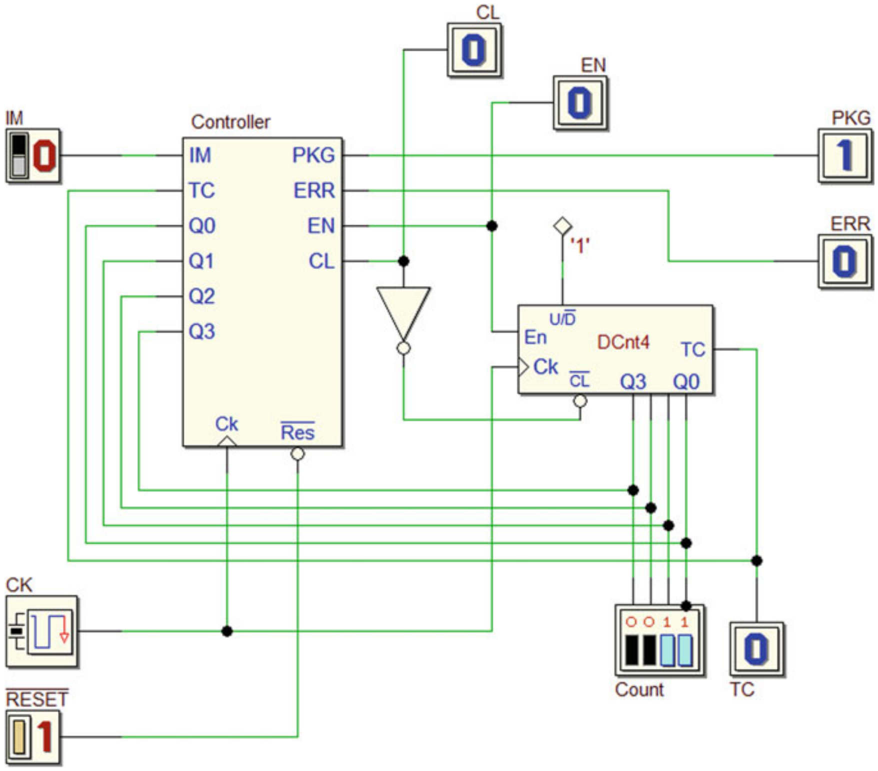
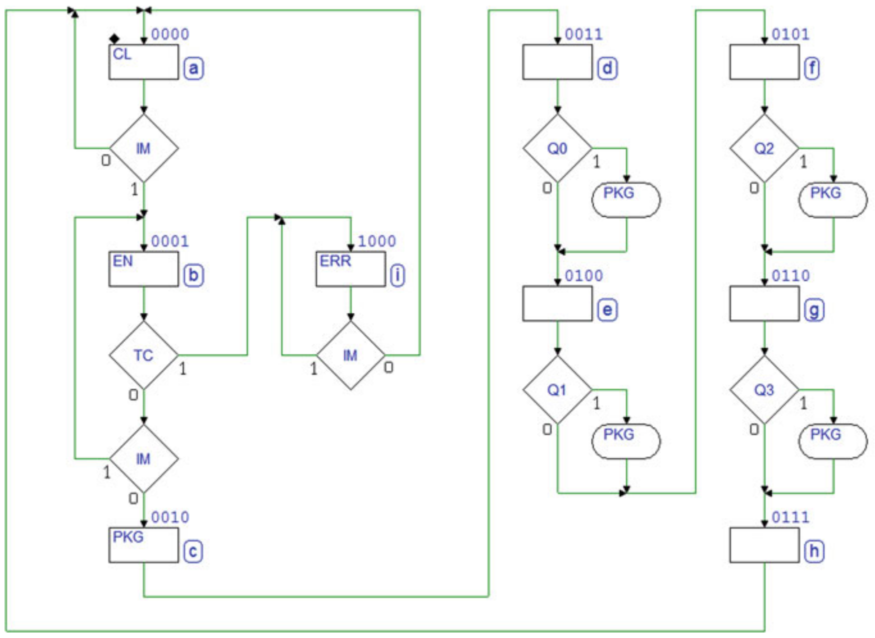
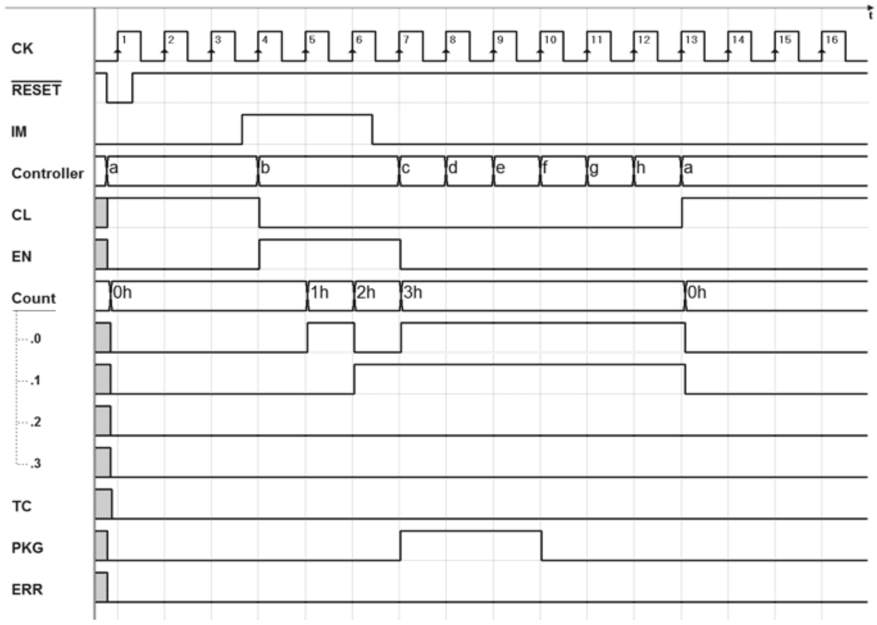


Diagram of the states of the controller:

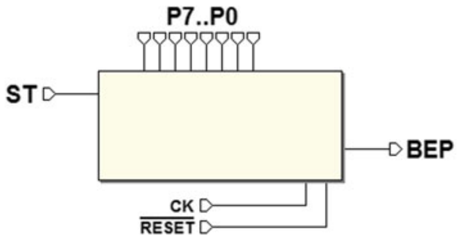


Timing simulation:



Solution of Exercise 2

The system's inputs and outputs:



The network schematic (controller + datapath):

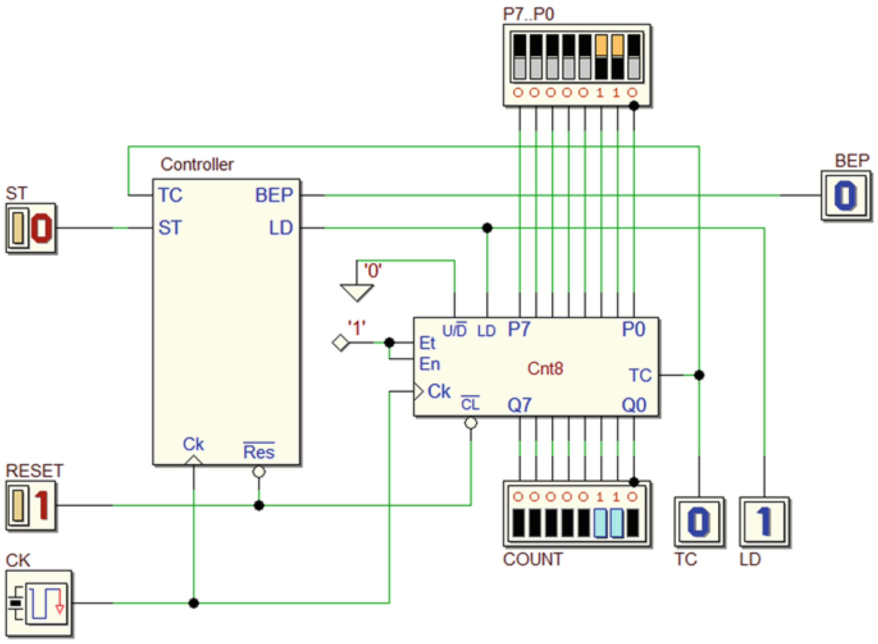
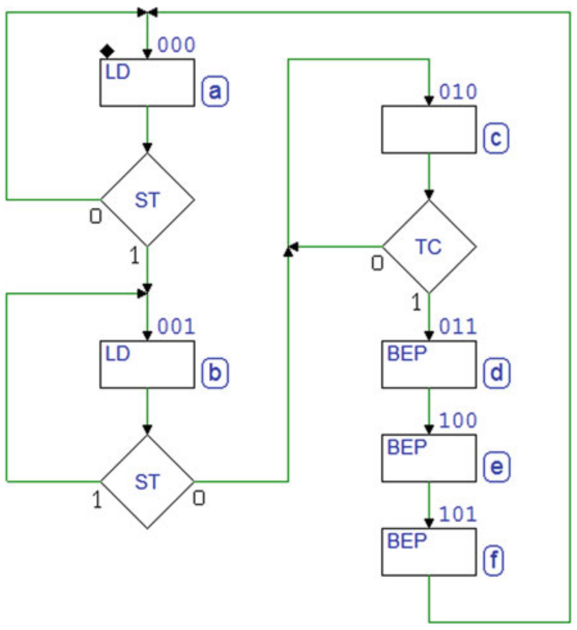
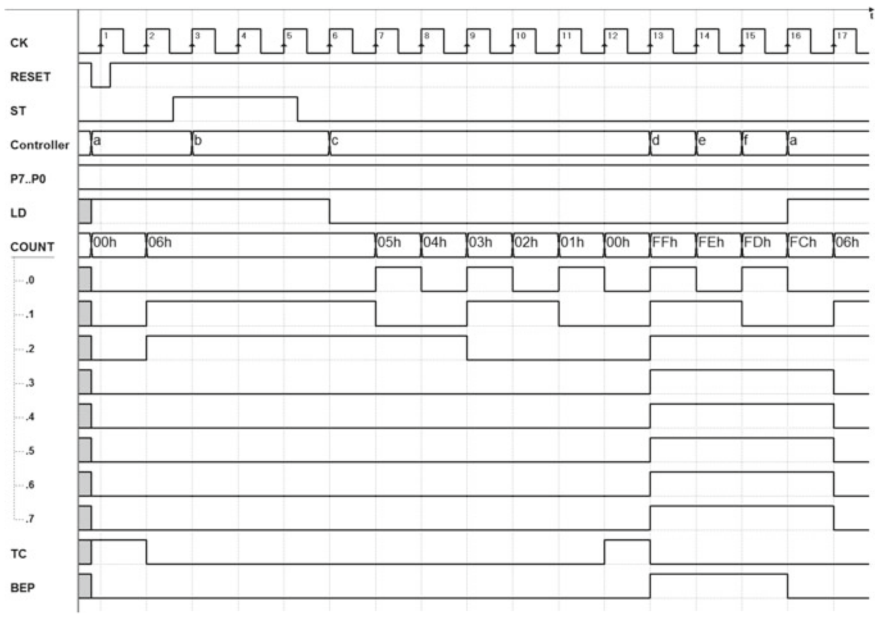


Diagram of the states of the controller:

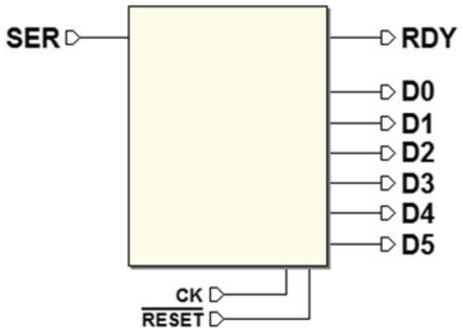


Timing simulation:



Solution of Exercise 3

The system's inputs and outputs:



The network schematic (controller + datapath):

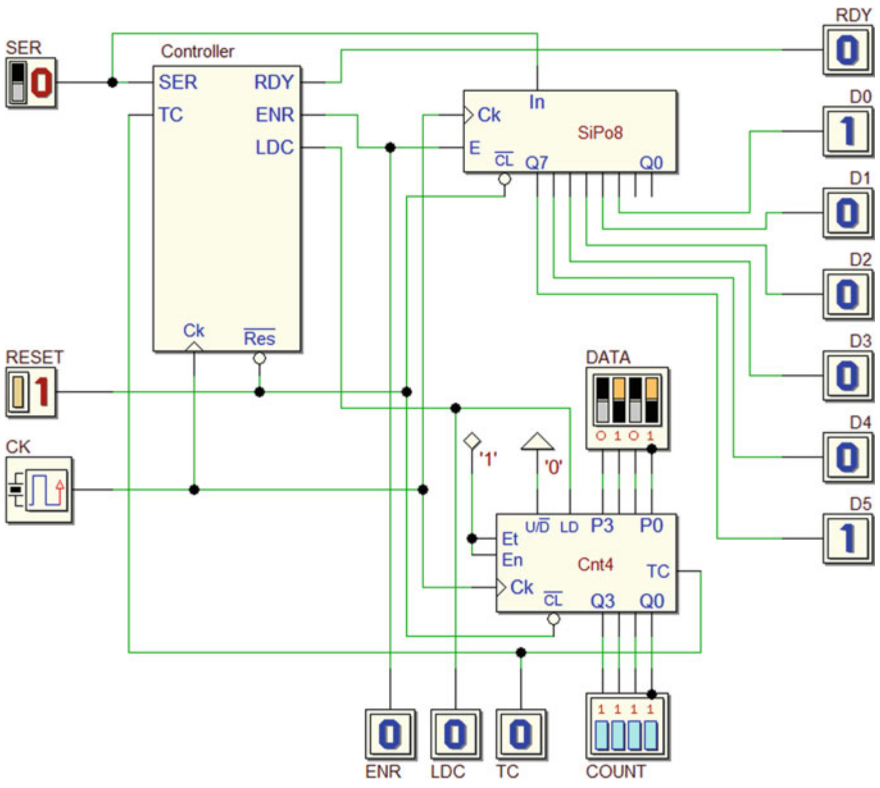
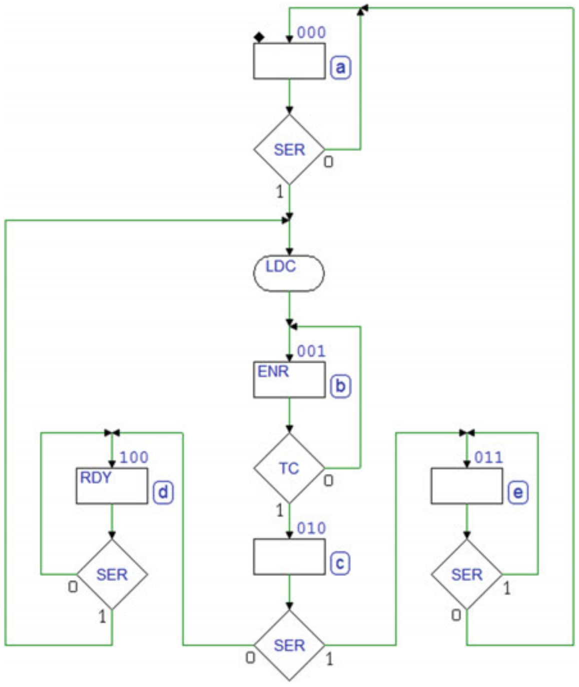
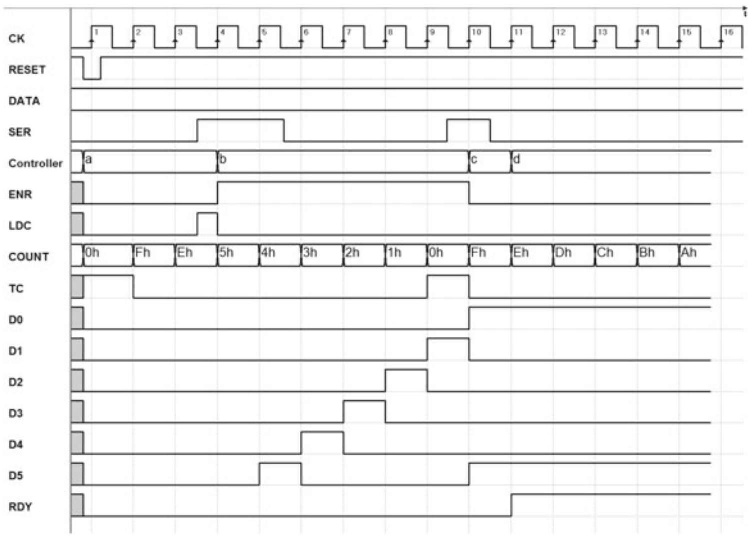


Diagram of the states of the controller:

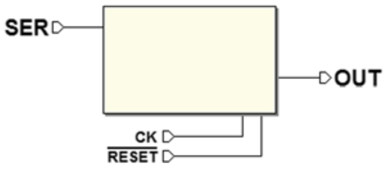


Timing simulation:



Solution of Exercise 4

The system's inputs and outputs:



The network schematic (controller + datapath):

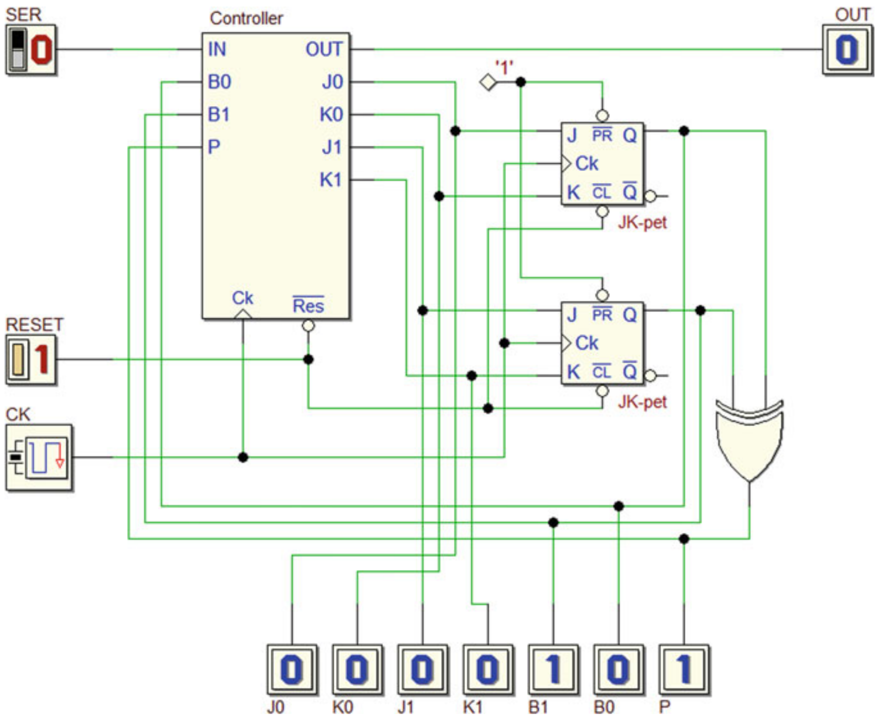
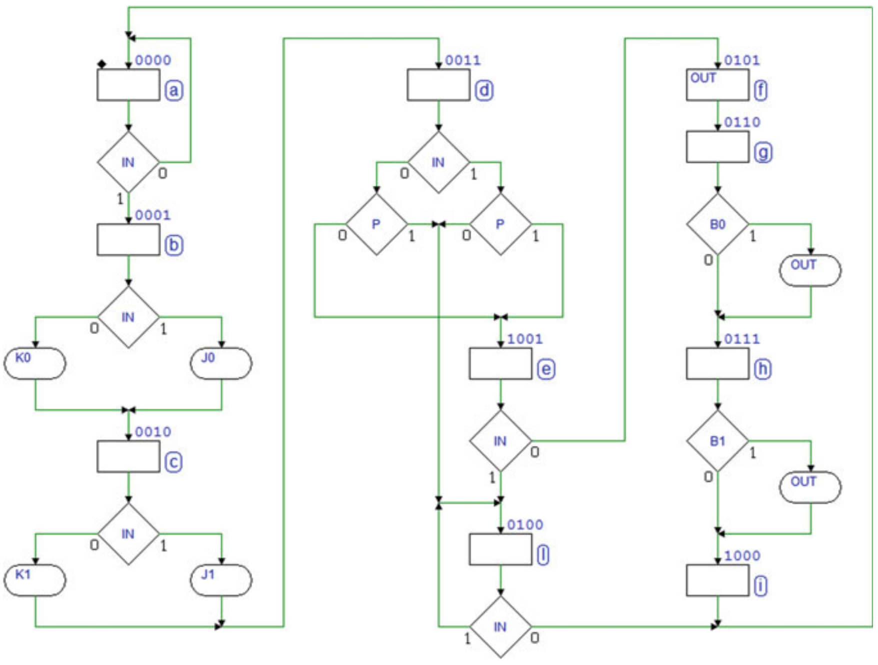
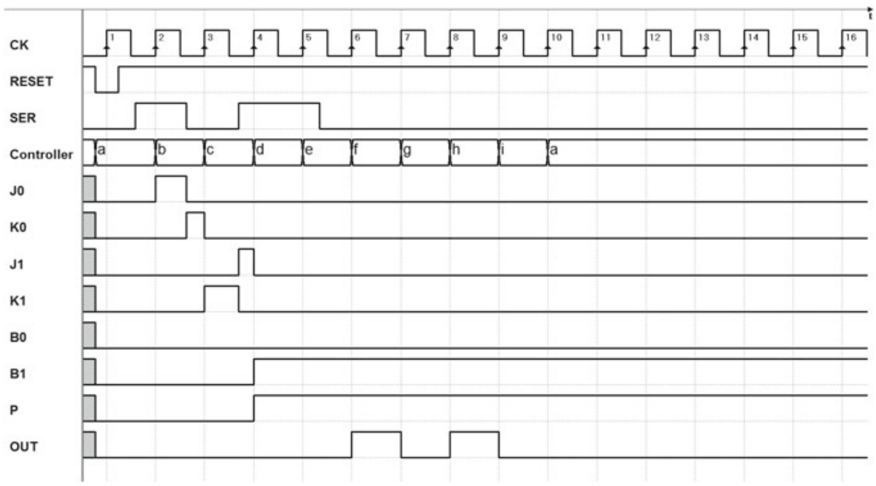


Diagram of the states of the controller:

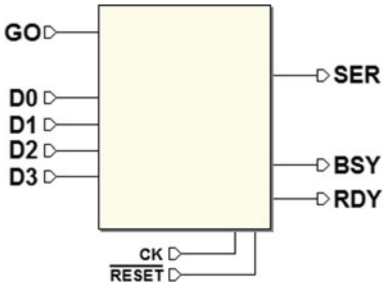


Timing simulation:



Solution of Exercise 5

The system's inputs and outputs:



The network schematic (controller + datapath):

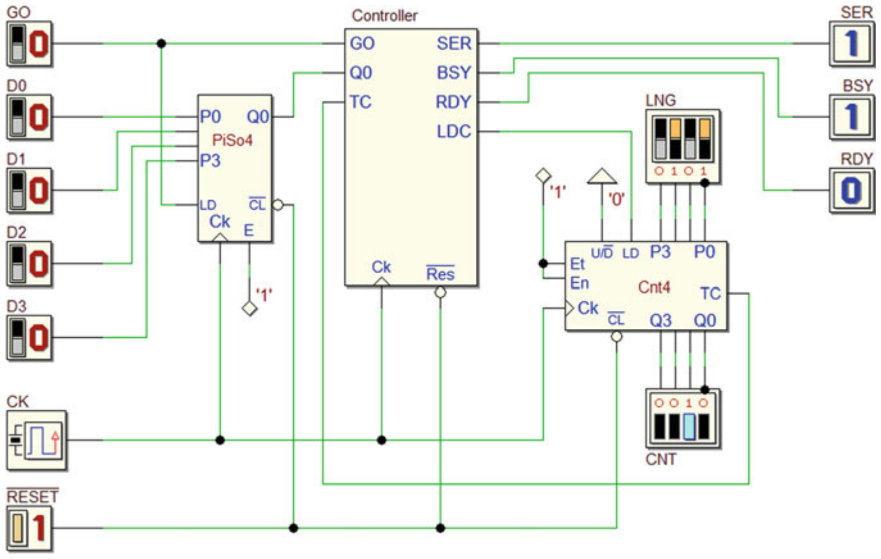
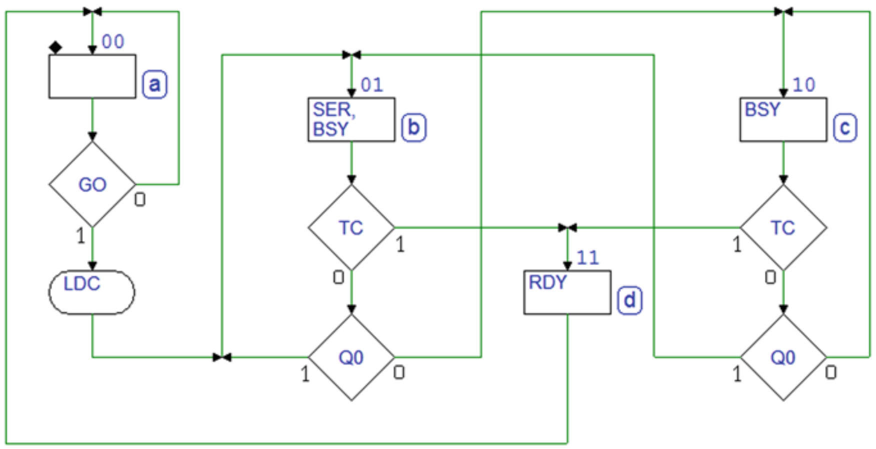
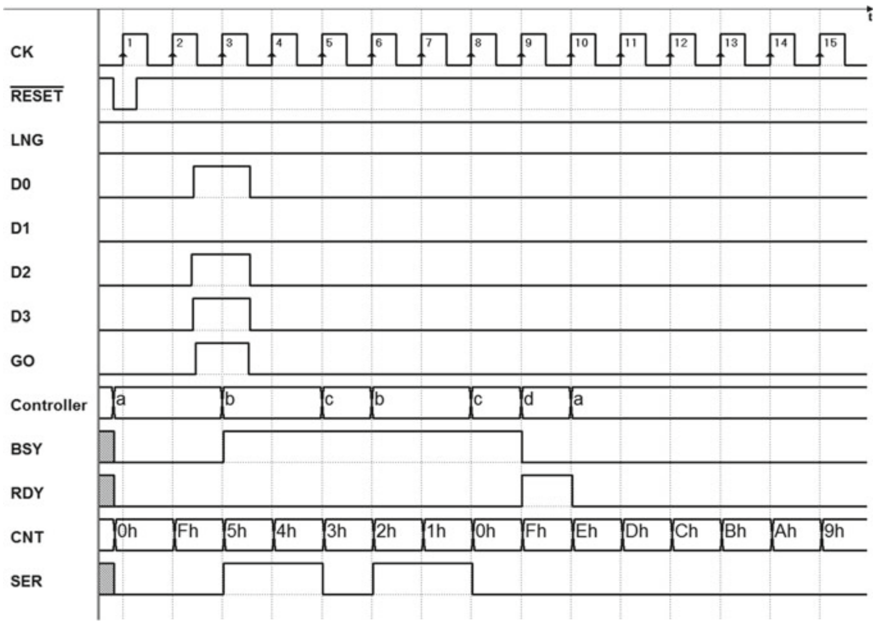


Diagram of the states of the controller:

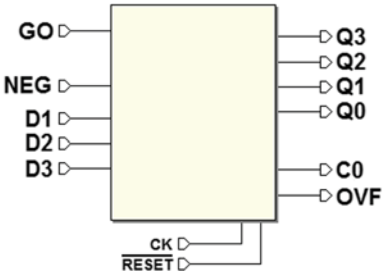


Timing simulation:

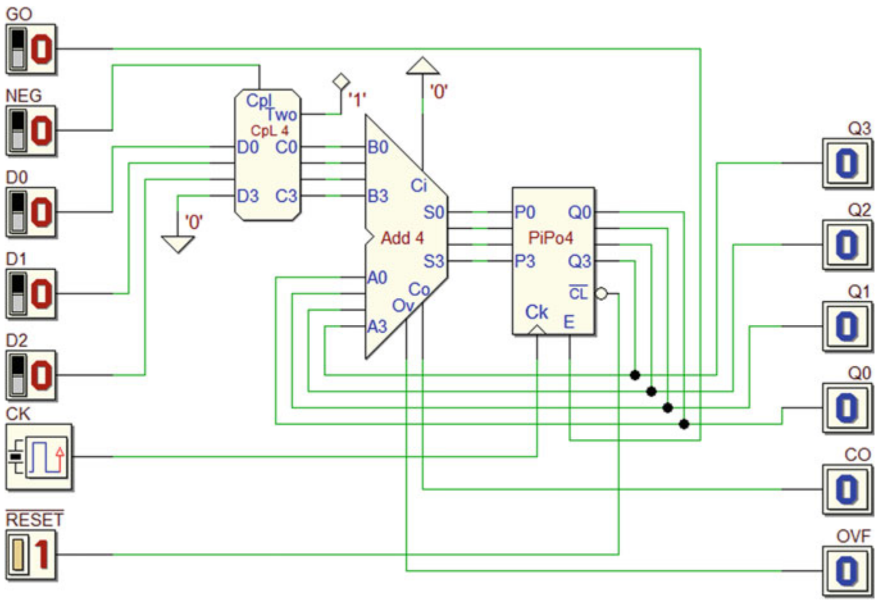


Solution of Exercise 6

The system's inputs and outputs:

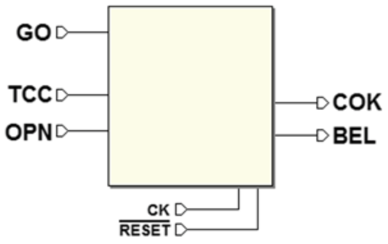


The network schematic (datapath only):



Solution of Exercise 7

The system's inputs and outputs:



The network schematic (controller + datapath):

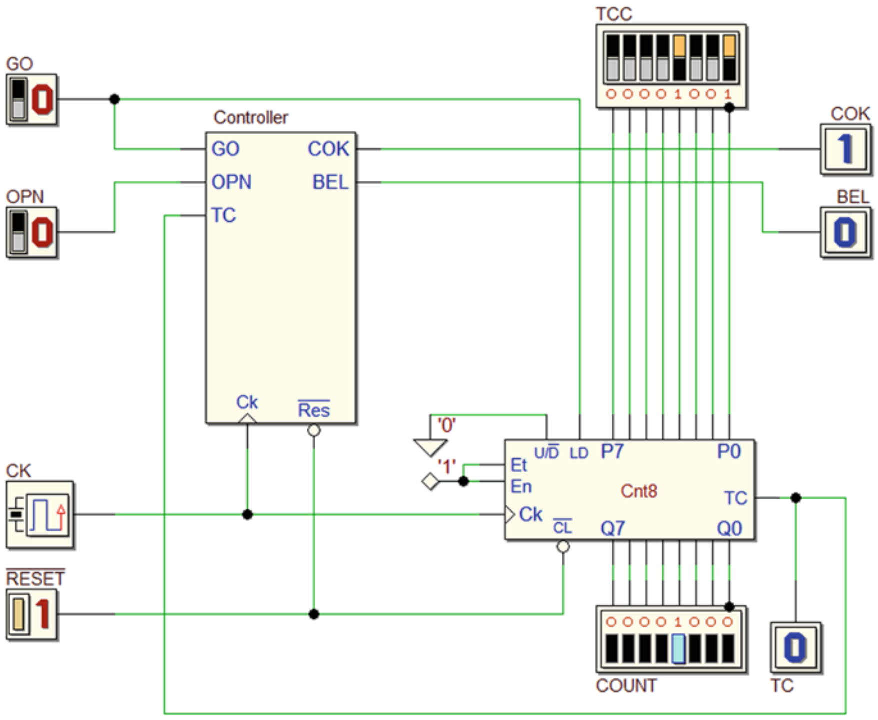
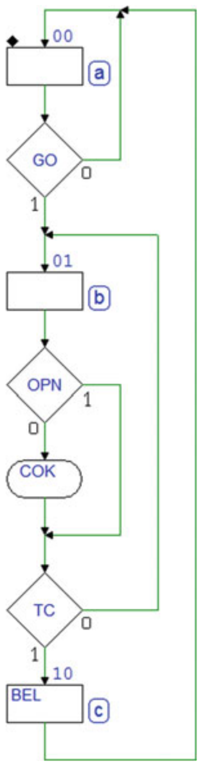
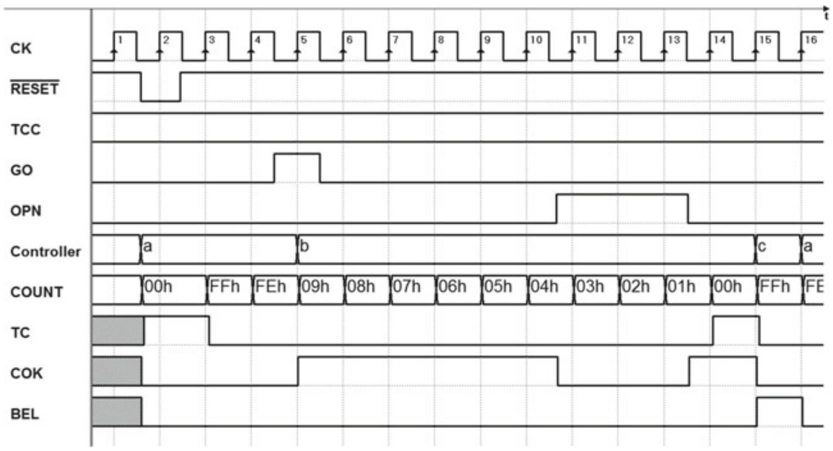


Diagram of the states of the controller:



Timing simulation:


CK
RESET
TCC
GO
OPN
Controller
COUNT
TC
COK
BEL
a
b
c
a
00h
FFh
FEh
09h
08h
07h
06h
05h
04h
03h
02h
01h
00h
FFh
FE