Chapter 6
Flip-Flop-Based Synchronous Networks

Abstract The flip-flops are the building blocks of all sequential networks. A regular structure made of flip-flop and combinational networks can implement any sequential circuit. In this chapter, the structures are not designed but either assembled in an intuitive fashion or taken from standard building blocks. The presentation of counters and registers introduces progressively the real full-featured components that are available for design. Sequential network analysis in the time domain is the important skill that is developed at the end of the chapter.

In this chapter, we will examine the most commonly used flip-flop-based networks that can be used as *functional standard blocks* to create more complex digital networks. We will analyze these networks *intuitively*, whereas in later chapters will deal with their systematic design. We will represent the networks through logical schematics that is in the form of a set of components, such as logical gates and flip-flops with their connections.

In Chap. 5, we examined simple sequential networks, which are generally formed by combinational networks with feedback. This was to gain an understanding of the structure and functionality of various types of flip-flops, the elementary memory cells. Theoretically, a more complex sequential network could be designed in the same way, as a set of combinational networks with feedback. This approach would, however, pose a number of problems especially on the design and testability level.

It is preferable to design a complex sequential network in a more structured way, by using flip-flops as the base *sequential elements*, connected by *purely combinational* networks. This makes it possible to clearly divide the memorization function, the typical quality of a sequential network (entrusted to the flip-flops), from the function that determines its logical behavior and evolution over time (entrusted to the combinational networks).

As seen in Chap. 5, the output of a sequential network is not only a function of the inputs at that moment but also of the values that they took on in the past.

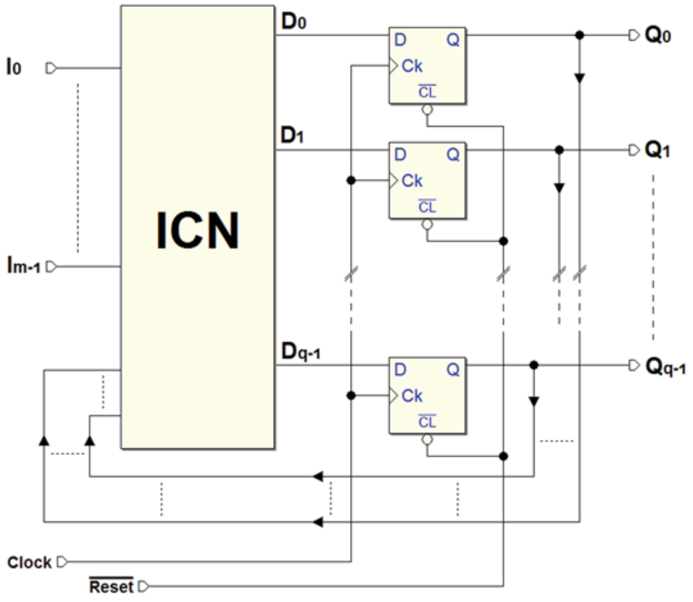
In a sequential network made up of flip-flops and combinational networks, the history of the network only leaves a trace on the values taken on by the flip-flop, since combinational networks, as such, have no memory.

The *set of values* memorized by the flip-flops is called the *state* of the networks. The outputs of a sequential network will thus be a function of the *inputs* and the *state*.

G. Donzellini et al., *Introduction to Digital Systems Design*,
https://doi.org/10.1007/978-3-319-92804-3_6

A *flip-flop-based synchronous network* is a network in which the flip-flops *share the same clock*, and the flip-flop asynchronous inputs (i.e., $\overline{Clear}$ and $\overline{Preset}$) are used *only for their initialization*. If these two conditions are not met, the network falls into the category of *asynchronous networks*.

Due to their regularity, synchronous networks have great advantages from the design and testability point of view, and it is possible to design the simplest structures without relying on formal synthesis methods. The networks examined in this chapter use D, E, and JK logical-type flip-flops with PET behavior.



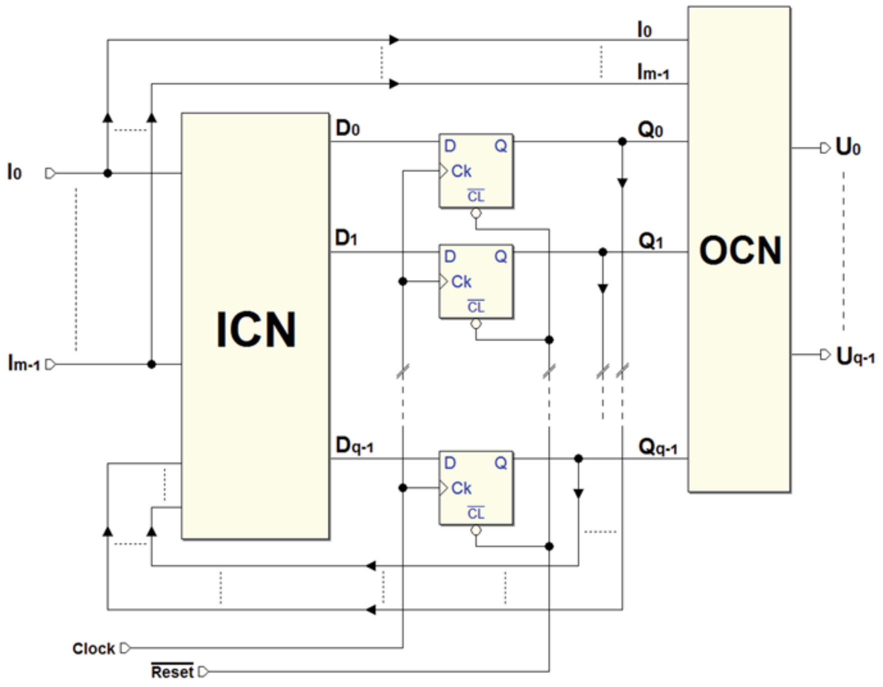
The figure above shows the basic schematic of a *flip-flop-based synchronous network* in a simplified form. The flip-flops share the clock signal. This guarantees one of the properties of synchronous sequential networks, the simultaneity[1] of the change in the flip-flop outputs. The active-low $\overline{Reset}$ makes it possible to initialize all the flip-flops.

The *D* inputs of the flip-flop are generated by the *Input Combinational Network* (ICN), which processes inputs $I_0..I_{m-1}$ coming from outside and the outputs $Q_0..Q_{q-1}$ of the flip-flops themselves.

At every *positive edge* of the *Clock*, the values of $Q_0..Q_{q-1}$ are replaced by those the ICN provides. It is therefore easy to understand how the combinational network shapes the sequential network's *behavior* through the *external inputs* and the *values memorized* in the flip-flops (the *state* of the network). This is why the ICN is called also the *next state combinational network*.

[1] Within the limits of the propagation delay dispersion of real components.

The general structure of a synchronous network is shown in the figure below:



This structure is different from the previous simplified one because of addition of the *Output Combinational Network* (OCN), which allows for greater flexibility in generating outputs. Each of the outputs $U_0..U_{p-1}$ can be generated as a combinational function of the state memorized in flip-flops $Q_0..Q_{q-1}$ and of the external inputs $I_0..I_{m-1}$.

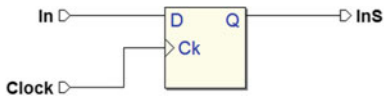
6.1 Synchronous and Asynchronous Signals

Since flip-flops share the same clock in a synchronous network, their outputs change in response to the edges of the clock; that is, they are *synchronous* with the clock. All the *network's outputs* and *internal* signals obtained through the flip-flop outputs' combinational networks maintain a fixed temporal relation with the clock even in presence of propagation delays.

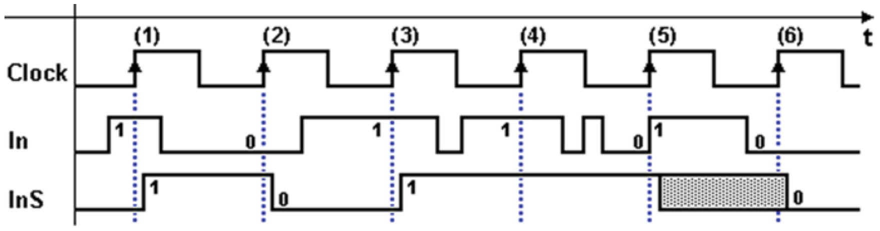
In principle, *input signals* have no relationship with the network's clock since they are generated by external systems. In general, an input signal is *asynchronous* unless it comes from another *synchronous network* that uses *the same clock*. Before using an *asynchronous signal*, it makes sense to *synchronize it* using an appropriate synchronization network.

6.1.1 Synchronizer

Let's now consider the network below, which is made up of a simple D-PET flip-flop that receives an *asynchronous* signal in the input.



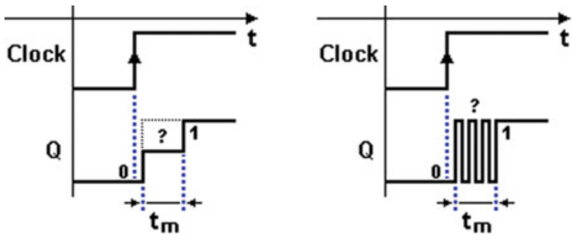
We want to examine this network as the *synchronizer* of the input signal. Output *InS* represents the *synchronized* version of the signal applied to input *In*. The timing diagram below shows the evolution of the input. Note that the fact that it is *asynchronous* has been highlighted.



On the rising edges of the *Clock*, from (1) to (4), output *InS* takes on the value of input *In* and keeps it until the next active edge. Any *intermediate variation* of the input is not detected, however, as we can see in the time intervals between edges (3) and (5).

On edges (1) to (4), we assume the *setup times* (**ts**) and *hold times* (**th**) of the flip-flop have been respected. On edge (5), however, they have been violated. The input makes an upward transition that is *simultaneous* to that of the *Clock*. Because of that, we cannot predict the output of the flip-flop, which is shown as *indeterminate* from the edge (5) and (6).

Under these anomalous conditions, a physical circuit may generate *invalid logical levels* or *oscillations* for a brief period.



Two examples of this phenomenon that is called *"metastability"* are reported in the figure above. On the left, output *Q* of the flip-flop is brought to an invalid logical level for time **tm** before stabilizing at value 1.

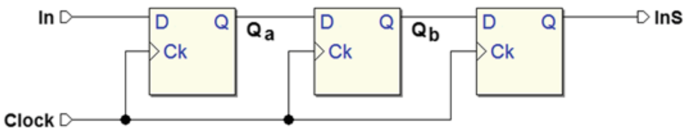
On the right, the output goes through a few oscillation cycles between the two levels before settling.

The logical network that reads these types of signal can produce an *error*. This is a *random* behavior in terms of the probability of it happening and for how long time **tm** lasts. These errors depend on the physical characteristics of the flip-flop and the frequency of the signals in play. The closer we get to the speed limits of the specific component in use, the greater the probability of metastable behavior becoming significant. The relationship is exponential.

In any case, errors due to metastability are very rare in well-designed flip-flops in non-critical conditions. The average time between two consecutive errors could be on the order of hundreds of years. Since the probability of an error in the digital circuit due to other causes (a circuit failure, electromagnetic disturbance, for example) is much higher, we can say that signal synchronization through this technique is generally reliable for many applications.

6.1.2 *Multistage Synchronization*

Under critical conditions or when higher security margins are requested, it is necessary to use a configuration that is more complex than the previous one and uses multiple synchronizer flip-flops connected in ripple fashion.

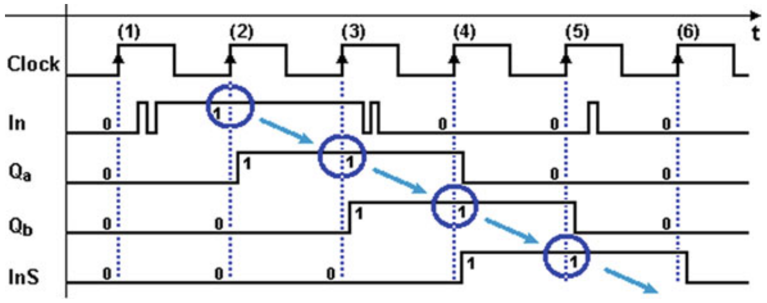


By extending the synchronization procedure this way, we can reduce the probability of error due to metastability to acceptably low limits. This means that any metastable behavior by the first flip-flop is filtered by the second and so forth. Aside from the question of metastability, it is interesting to consider the temporal relation between input *In* and output *InS* as shown in the figure further on.

At every rising edge of the *Clock*, the first flip-flop transfers the value of input *In* onto output Q_a, as with the previous synchronizer. Q_a thus reproduces the evolution of input signal *In* in clean, synchronized form.

Seeing that the first flip-flop generates the very output Q_a after the edge of the *Clock*, the second reads the new value of Q_a on the edge of the next *Clock*. Thus the timing path of Q_b is the same as that for Q_a, but delayed by *one Clock period.*

The next flip-flop echoes the process so output *InS* is still identical to Q_a, but delayed by another *Clock* cycle. On the next figure, we have highlighted the fact that the value generated by a flip-flop on edge (n) is read by the next flip-flop at edge $(n + 1)$.



Further on, we will often revisit the concept of *reading the value at the next Clock cycle* in the information exchange between two synchronous networks.

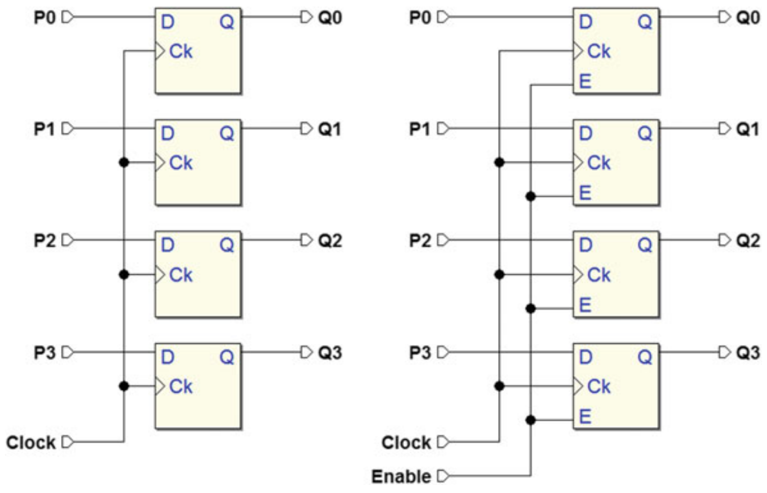
6.2 Registers

Registers are important logical structures used to memorize data. It is possible to *"write"* binary data on a register, keep it for a period of time, and *"read it"* as many times as necessary. From this perspective, the D flip-flop is also a *register*, since it can carry out the operations described on *one single bit*.

A register is usually made up of a number of flip-flops equal to the *number of bits* of the data that needs to be memorized. Registers allow for two ways to store and retrieve data: the *parallel* mode and the *serial* mode, as we will see in the next few sections.

6.2.1 Parallel Registers

The networks in the figure below are two examples of *parallel registers* that can memorize four bits of information.

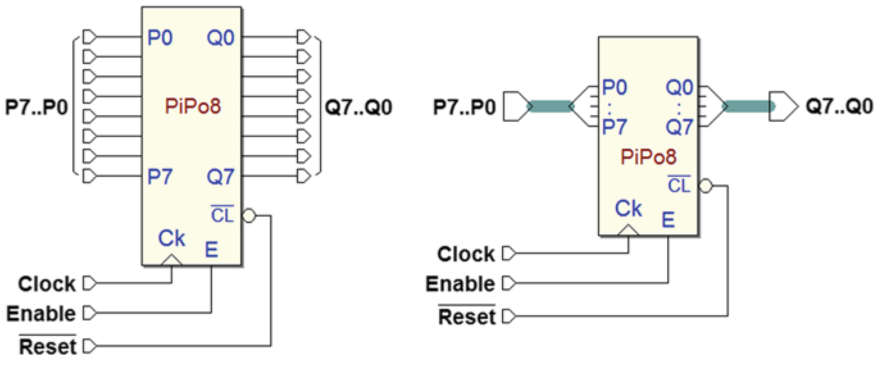


The register at the left uses D-type flip-flops while the one on the right uses the E type. These are *synchronous* networks since the flip-flops receive the same *Clock*. An older term for this type of register is "PIPO" (*Parallel Input–Parallel Output*). Normally, registers have an initialization input, which for simplicity's sake is not shown in this figure.

In the D-type version, input data $P3..P0$ is memorized *in parallel* in the flip-flops at a rising edge of the *Clock*. The data is maintained on outputs $Q3..Q0$ and remains available until the next writing. If there is no new writing, they remain in the register indefinitely as long as the network is in operation (i.e., its power supply is on).

In the E-type variant, we have the added opportunity to enable/disable the writing under the control of the *Enable* input. This is the most commonly used version in complex systems where, for example, there can be many registers and the one that will store the information needs to be *selected* each time.

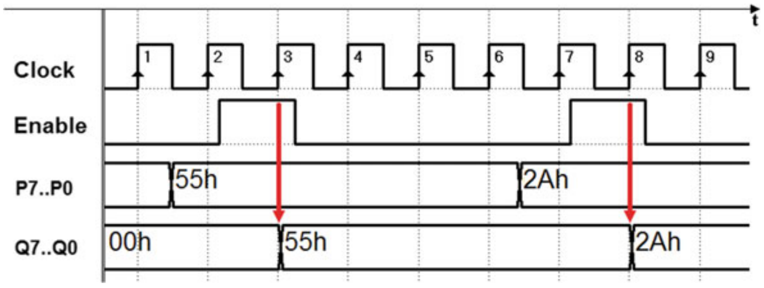
Below there are two examples of 8-bit synchronous parallel registers taken from the *Deeds* simulator library. These are two versions of the same component that differ by the way their connections are represented.



In the version on the left, the terminations are represented individually: the eight data inputs $P7...P0$, the eight outputs $Q7...Q0$, and the *Clock* inputs, the E enable, and the asynchronous initialization input $\overline{CL}$ (*Clear*). On the right, inputs $P7...P0$ and the eight outputs $Q7...Q0$ are shown in the form of *multi-wire connections*, in short, *bus type*.[2]

The figure below shows the typical operation sequence of an eight-bit register with *bus-type* connections and enabling. In the timing diagram, the hypothesis is that the register will initially contain 0 in all the flip-flops. As we can see, the graphic representation of the values is "*cumulative*"; that is, it represents all the bits in the register on one single track, a bar formed by two parallel lines that cross at the transitions of the signals. The value taken by the signals between one transition and another (in this case, represented in hexadecimals) is written inside the bar.

[2]Representing multi-wire connections as *bus* allows us to simplify the logical schematic and make it more readable, especially if complex.

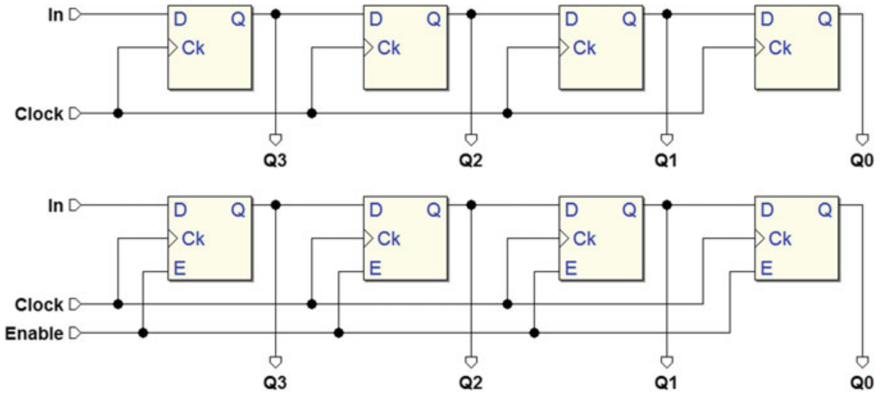


In our example, lines *P*7..*P*0 are set over time to values 01010101 (= 55h) and then 00101010 (= 2Ah). *Enable* is only driven to load the new data onto outputs *Q*7..*Q*0 in response to edges 3 and 8 of the *Clock*.

Parallel registers are widely used in digital systems where data is generally organized into "words" made of multiple bits and stored in registers. Note that in the structure of the *generic synchronous sequential network*, featured at the beginning of this chapter, the *state of the network* is memorized in a register.

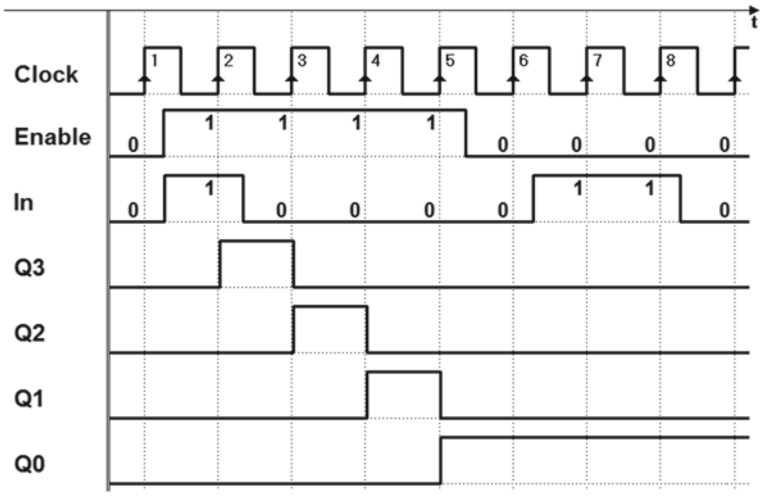
6.2.2 Shift Registers

The other mode of data input in registers is *serial*, where the bits that make up a word to memorize are presented to the input one at a time in succession. One register that allows for this is called *shift register* (SHR). It can be made of D, E, or JK logical-type flip-flops. We have already been introduced to this structure, with the D type, used as a synchronizer.



In the figure here above, we see two examples of four-bit shift registers: the first with a D-type flip-flop and the second, with the E type. We can see that the output of each flip-flop is connected to the input of the next while the *Clock* is shared by all, since this is a synchronous network. For simplicity's sake, the initialization network is omitted. The *"serial"* input of the register is *In*. The outputs in both cases are *Q*3..*Q*0; the *Enable* input is available only in the second version.

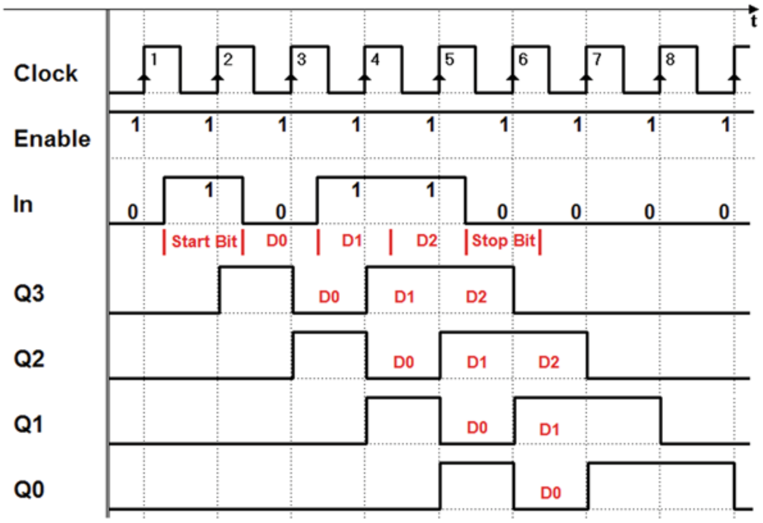
The timing diagram below is an example of how the type E register functions. The hypothesis is that outputs *Q*3...*Q*0 will initially be at 0 and that input *In* will be activated in correspondence with edge 2 of the *Clock*. Also, *Enable* is activated for four *Clock* cycles, from 2 to 5.



As with the synchronizer, the flip-flop farthest to the left transfers the value of input *In* onto output *Q*3 at every rising edge of the *Clock*. This means that output *Q*3 reproduces the evolution of input signal *In*, synchronized to the *Clock*. The second flip-flop reads the changes in *Q*3 on the next rising edge so *Q*2 is identical to *Q*3, but *delayed* by one *Clock* cycle. The same goes for all the other flip-flops.

From edge 6 on, *Enable* is read at 0. With no enabling, the register's content remains unchanged from edge 5 on, so the new activation of *In* is ignored.

Let's now look at another use of the shift register. Here, it is used as a base element of a *serial sequence receiver*. In the next figure, it is assumed that input *In* receives a sequence of bits with a pre-established format (protocol). In this case, the format requires the bits to be put into groups of five with a *"start bit"* at 1, three *"information"* bits (D0, D1, D2) and a *"stop bit"* at 0. Each of these bits has the same duration as a *Clock* cycle, as shown.

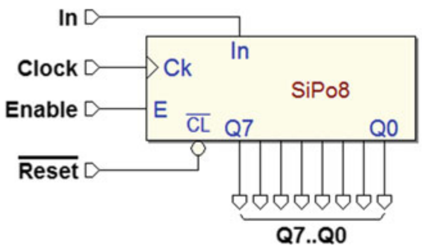


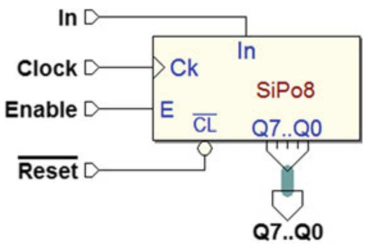
Assume that when line *IN* is in the *"idle"* state, that it is normally defined at 0, so the *start bit* at 1 signals the beginning of the sequence. The *stop bit* separates two consecutive groups with a 0.

In our example, the *start bit* is read by the first flip-flop on edge 2, so the information bits (D0 = 0, D1 = D2 = 1) are read by edges 3, 4, and 5, and the *stop bit*, by edge 6. When receiving the sequence, the register memorizes and shifts the bits received one by one. After edge 6, information bits D0, D1, and D2 are made available *in parallel* on outputs $Q0$, $Q1$, and $Q2$. In this simplified example, the shifting continues with each new *Clock* cycle, and so data is progressively lost.

Serial/parallel conversion is widely employed in digital telecommunication systems where serial communication is mainly used to cover great distances. It is also used in data processing systems when it is necessary to reduce the number of wires connecting the system's modules, to reduce cost or improve usability and practicality. USB (*Universal Serial Bus*) connections are an example of serial format communication. The parallel format provides quicker and more efficient data processing and is generally used in computational systems.

Below are two examples of 8-bit shift registers from the library of *Deeds*:





The same device is shown with different connections. The one on the right uses the *bus* type for outputs $Q7..Q0$. On the component, we see the acronym “SIPO,” which stands for *Serial Input–Parallel Output.*

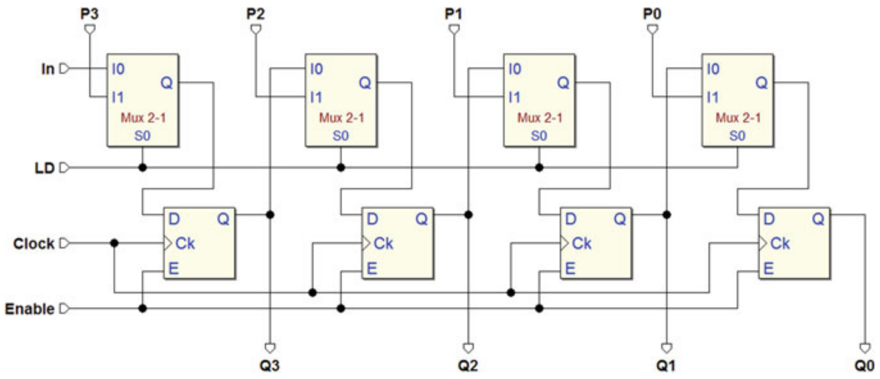
There are also “SISO” (*Serial Input - Serial Output*) registers whose internal structure is identical to that of the SIPO type. It is clear that, in principle, one can simply choose any of the outputs Q to obtain a serial output. SISO registers as such (those with *one single serial output)* are normally made with many, sometimes thousands of flip-flops.

They are used to obtain a delayed signal of as many clock cycles. Due to the large number of connections required, it is impractical to make the intermediate outputs externally available.

6.2.3 Shift Registers with Parallel Load

We have seen that registers with serial input and parallel outputs directly carry out serial–parallel conversion. To obtain the opposite conversion, from the *parallel* format to the *serial* format we must first load the data onto the register *in parallel*, and then make it shift *serially*.

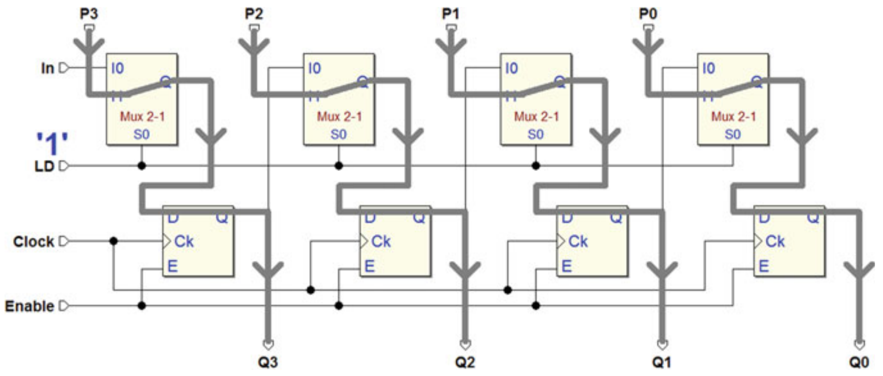
We can do this by combining the structures we have already seen in the parallel and shift registers, with the help of multiplexers. The figure below shows the schematic of a 4-bit shift register with parallel load (we have omitted the initialization network).



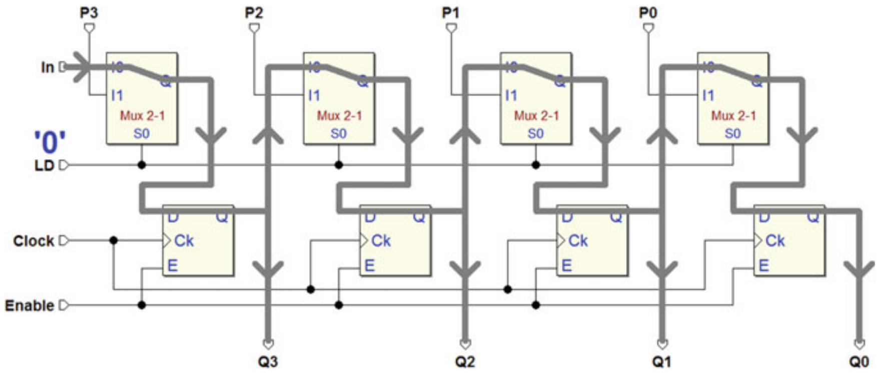
We see the four E-type flip-flops with their outputs $Q3..Q0$, as in the previous register types. Here, however, we have the serial input *In* and the parallel inputs $P3..P0$.

The new input *LD* (*“Load”*) controls the parallel load through the four multiplexers that allow to choose the data to send as input to the flip-flops.

As we can see in the next figure, when $LD = 1$, the selectors route the parallel inputs $P3..P0$ to the flip-flop. If $Enable = 1$, the parallel data in the input is loaded onto the flip-flops at the next rising edge of the *Clock*. The data appears on outputs $Q3..Q0$.



If $LD = 0$, the data is shifted, as we can see in the figure below. The serial input *In* is routed to the first flip-flop; its output is brought to the second and so on. Obviously, the shifting is carried out on the next rising edge of the *Clock*. Note that parallel load and serial shifting are mutually exclusive operations.

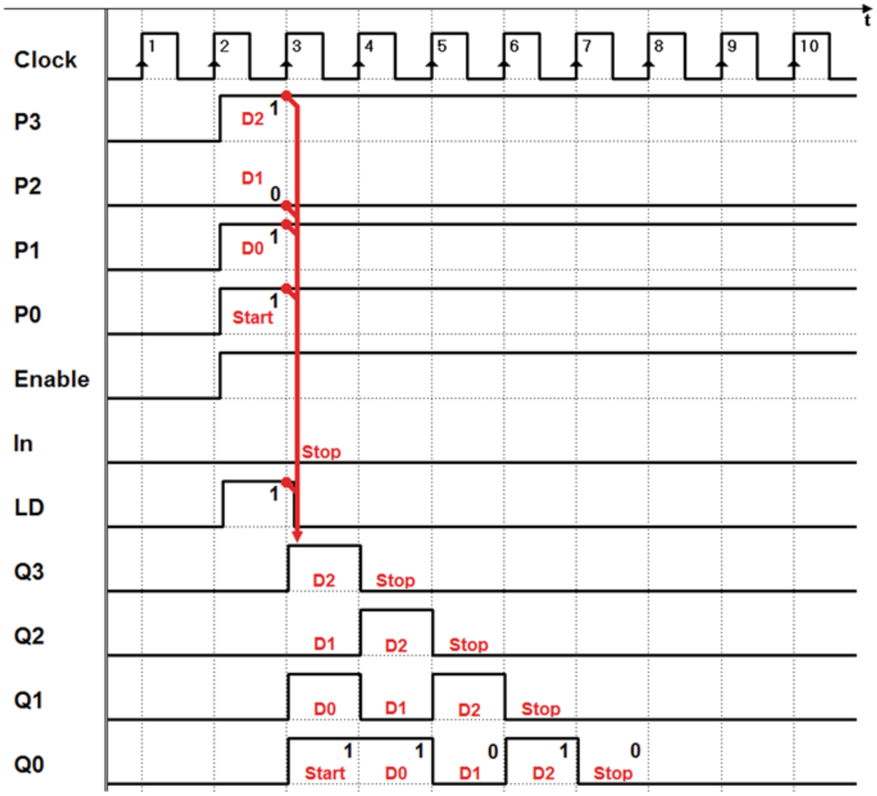


We have previously seen an example of *serial sequence receivers*. A shift register with parallel load, like the one we examined above, is perfectly suited to *transmit a serial sequence*.

In the timing diagram below, let's assume we want to transmit a bit sequence to output $Q0$ according to the serial receiver format defined in the example. Remember that the format requires a group of five bits: the *start bit* at 1, then three *information bits* D0, D1, and D2 followed by the *stop bit* at 0. Each bit has the same length as a *Clock* cycle.

We choose D0 = 1, D1 = 0, and D2 = 1. As seen in the next figure, we set these values on the register's parallel inputs: $P1 = \text{D0}$, $P2 = \text{D1}$, and $P3 = \text{D2}$.

Notice that *P*0 has been forced to 1 since it is the *start bit* to be transmitted first, while the serial input *In* is forced to 0, because we need to transmit the *stop bit* last.

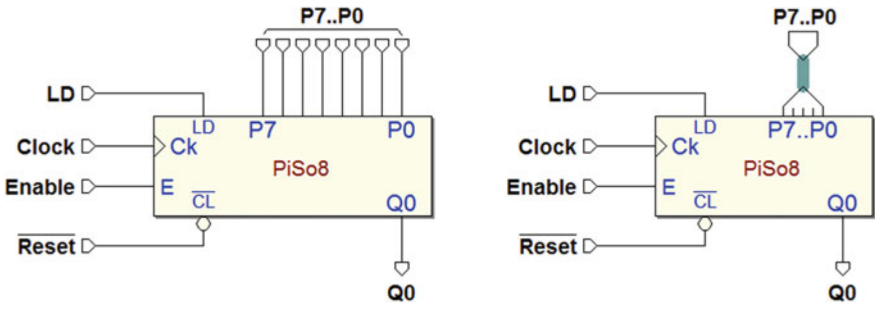


Together with the data, we enable the register by activating the *Enable* input. We bring *LD* to 1 for one *Clock* cycle so that the parallel load can be carried out on edge 3 of the *Clock*. The *start bit* is brought to output *Q*0 and is maintained for one *Clock* cycle.

Given that *LD* = 0, the register shifts right at edge 4 bringing the value of D0 to output *Q*0, which is maintained for one *Clock* cycle.

Meanwhile, note that serial input *In* inserts a 0 from the left, so the register's content progressively returns to zero. In the end, the *stop bit* is transmitted at 0. In this simplified example, the shifting continues on the following *Clock* cycles and the register continues to send 0 on line *Q*0.

In the next page are two examples of 8-bit shift registers ("PiSo8") with parallel load taken from the library of *Deeds*.



The only difference between these two components is in the connections. The one on the right uses the *bus type* for inputs *P*7..*P*0. These components belong to the "PISO" (*Parallel Input–Serial Output*) classification.

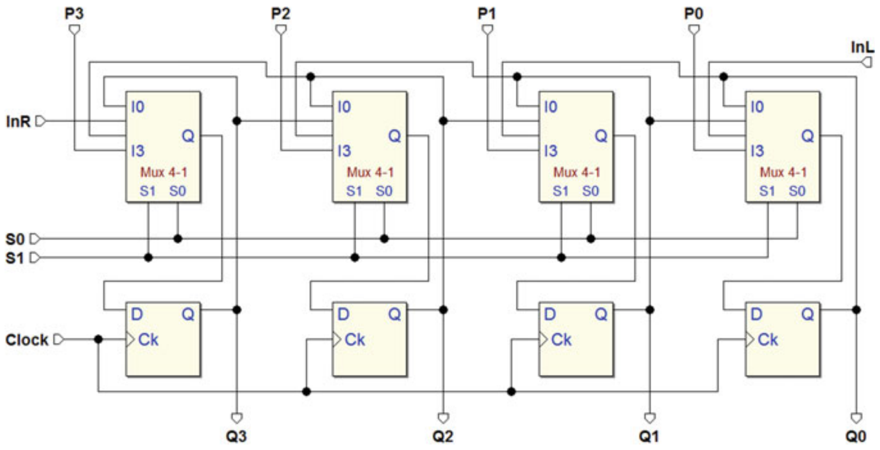
Note, however, that their only output is line *Q*0 and there are also a few small differences in the enabling logic. Here, input *E* controls the enabling of the shifting only. To do a parallel load of the data, one needs to activate *LD*.

6.2.4 Universal Shift Register

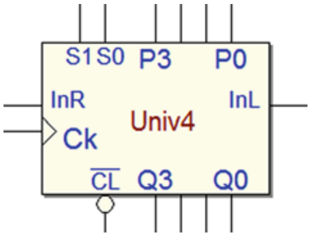
The registers shown so far make it possible to load and read data in serial or parallel format but shifting is only done in one direction (to the right, $Q3 \rightarrow Q0$). Other registers allow shifting in both directions.

The *universal register* allows *bidirectional shifting* and can be loaded and read in *serial* and *parallel* format.

Below, the schematic of a 4-bit universal register (the initialization network has been omitted also in this case).



For a summary of this register's terminations and their functions, let's consider the component "Univ4" shown below (from the *Deeds* library).



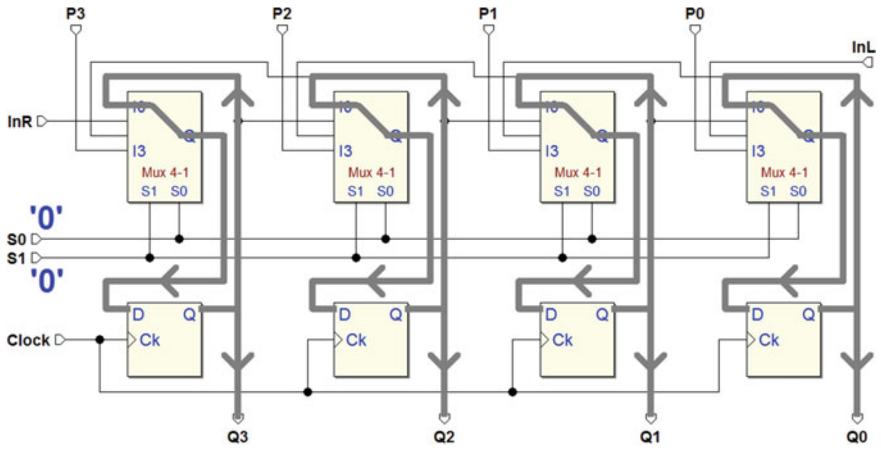
As in parallel registers, here there are inputs $P3..P0$ and outputs $Q3..Q0$. Obviously, clock (Ck) and clear ($\overline{CL}$) inputs are also present.

InR and *InL* are the serial inputs for *right* and *left shifting*, respectively.

Inputs $S1$ and $S0$ select component's operation according to the table below.

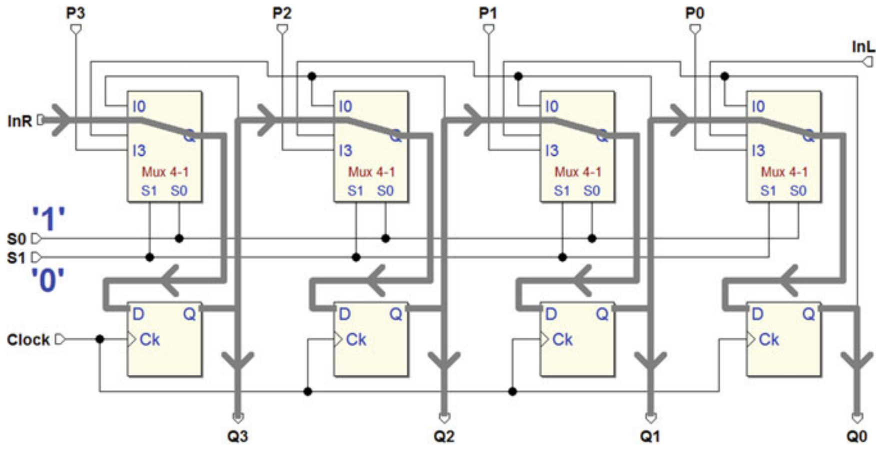
$S1$	$S0$	Function
0	0	Maintaining information
0	1	"Right" shifting
1	0	"Left" shifting
1	1	Parallel loading

The configuration ($S1 = 0$, $S0 = 0$) maintains the information because connects each input D with the output Q of the same flip-flop (see the figure).



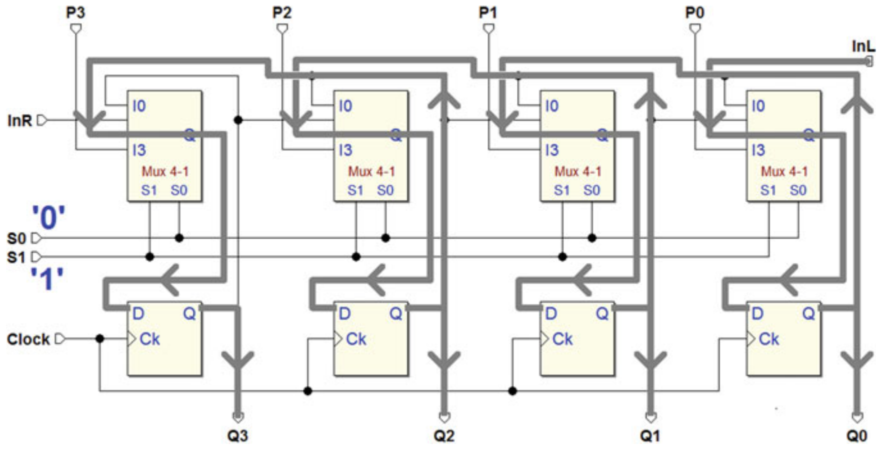
On the rising edge of the *Clock*, the register reloads the previous values on the flip-flops.

The combination ($S1 = 0$, $S0 = 1$) configures the register for *right shifting* (see the figure below).



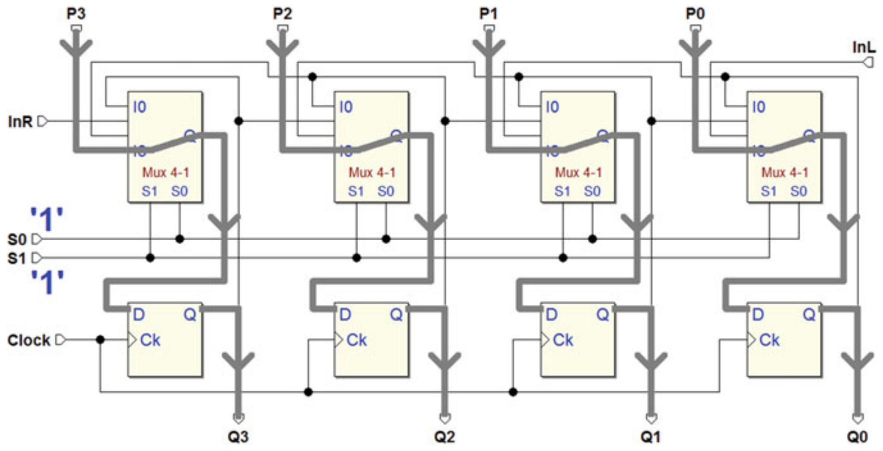
On the rising edge of the *Clock*, the data at $Q1$ is loaded onto the farthest right flip-flop and appears on $Q0$. The one on $Q2$ is transferred to $Q1$, and $Q3$ to $Q2$. *InR* is copied onto the first flip-flop and appears on $Q3$.

The next figure shows the *left shifting* operation. It is obtained through the combination ($S1 = 1$, $S0 = 0$).



Despite the seemingly complex paths in the figure, the routing here is analogous to that of the previous figure, only in the opposite direction. In this mode, the serial input is *InL*. On the rising edge of the *Clock*, the value of *InL* is loaded onto $Q0$, while the other outputs shift one stage to the left.

Finally, the configuration ($S1 = 1, S0 = 1$) routes the value of parallel load input P onto each corresponding input D. On the rising edge of the $Clock$, the register will carry out a parallel load as shown in the figure below.



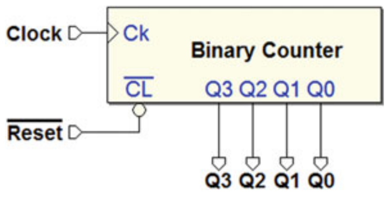
6.3 Counters

Another commonly used type of sequential network is the *counter*. This term indicates a network that generates a numerical sequence in a particular code (think, for example, of an increasing sequence made up of binary numbers represented by a certain number of bits). The network's *active edge* of the clock input causes the passage from one element of the sequence to the next. The counter is *synchronous* when the flip-flop network that creates it is *synchronous*.

6.3.1 Binary Counters

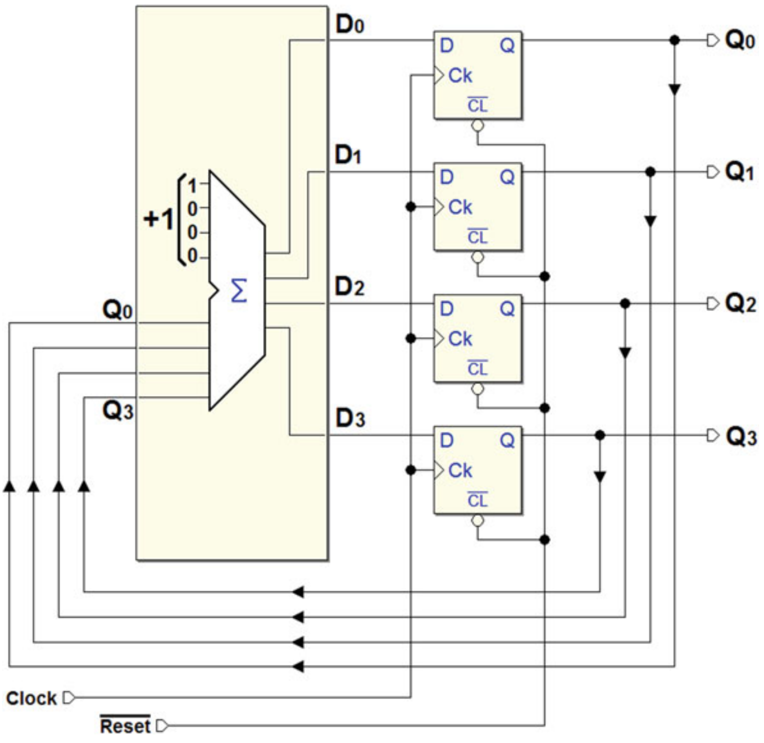
The following figure depicts an example of a *natural binary 4-bit counter*. The table on the right shows the outputs' *16 combinations*. This is an *increasing sequence*, so it is an *"up counter"*.

A *counting cycle* is made up of a sequence of 16 different configurations that can be generated: it is the case of a *"module 16"* counting. When it gets to the highest number "1111", the count continues cyclically at "0000". The rising edge of the *Clock* advances the count. Input $\overline{CL}$ ($\overline{Reset}$) makes it possible to initialize the count at the value "0000".



$Q3$	$Q2$	$Q1$	$Q0$
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

The internal structure of a counter is a synchronous sequential network like the one described at the start of this chapter. It is made up of a D-PET flip-flop parallel register and a combinational network that controls its behavior. In this counter, the function required to the combinational network is to increase the binary number on the outputs by 1, as shown in the figure below.



Intuitively, the count could be done by a full adder that adds the constant +1 to the number on the flip-flops' outputs at that time. The result is submitted to the flip-flops' inputs that will load the new number at the next rising edge of the *Clock*. Note that the carry from the fourth bit of the sum is ignored, since there are only four bits. This is how we get module 16 counting.

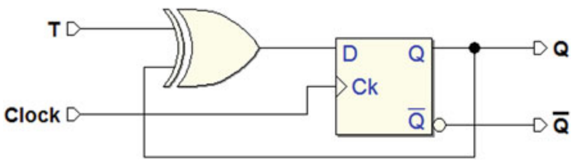
Let us now proceed more systematically and describe, in this truth table, the behavior that the combinational network should have.

$Q3$	$Q2$	$Q1$	$Q0$	$D3$	$D2$	$D1$	$D0$
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	0	0

The left side of the table shows the 16 possible combinations of flip-flop outputs $Q3..Q0$ and, on the right, the corresponding values of inputs $D3..D0$ that the combinational network must produce.

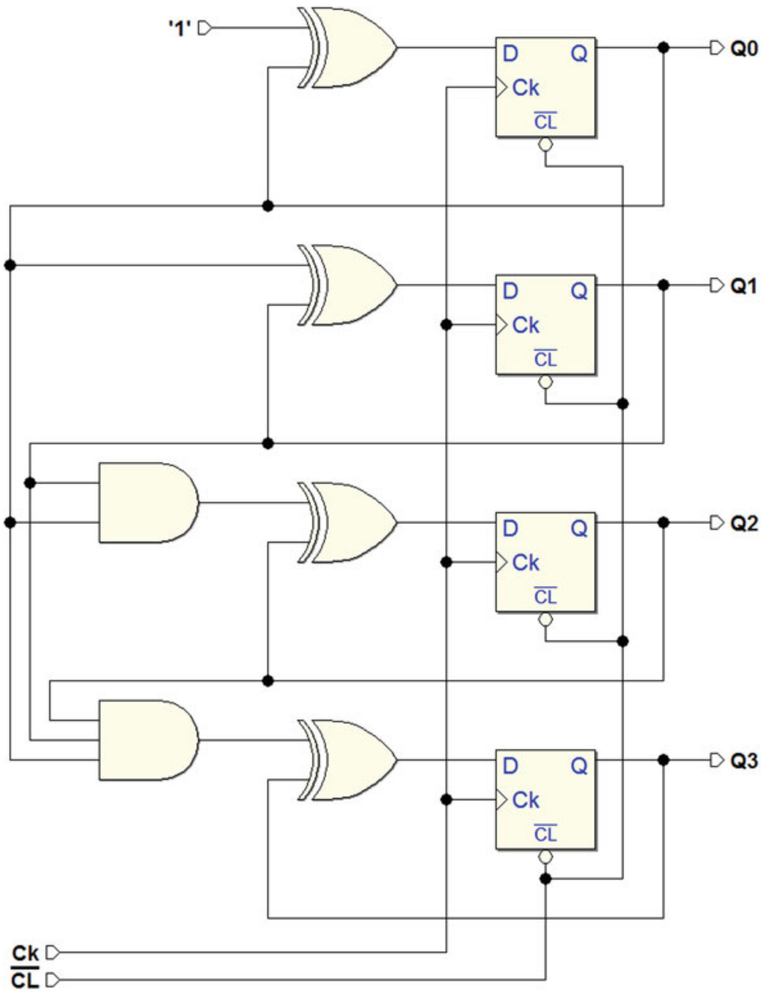
Since the rising edge of the *Clock* loads the value processed by the combinational network onto the flip-flops, the table actually links the *current state* of the network with the *next state*.

Remember that a D-PET-type flip-flop connected to an XOR through feedback reproduces the functioning of the T type, as we have seen before.



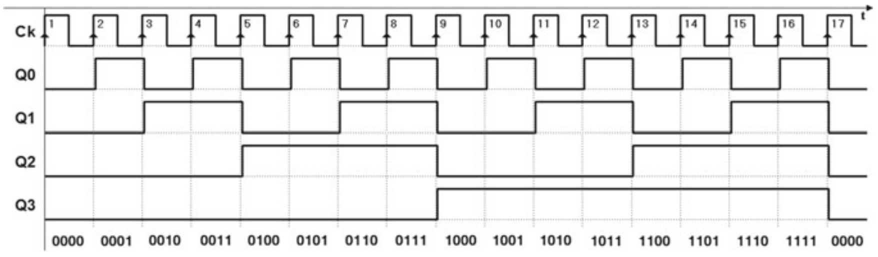
If we put $T = 0$ at the input of the XOR the flip-flop keeps the previously memorized value. If $T = 1$, then outputs are inverted.

With this in mind, we synthesize the combinational network described in the table above. Omitting all the intermediate steps, we derive the schematic of the adder based on D-PET flip-flops connected to form a T-type flip-flop.



The regular structure of the network allows us to study its functioning even on an intuitive level, with the help of the timing simulation below.

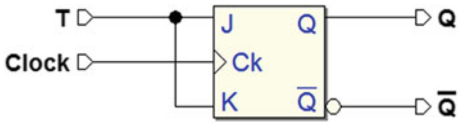
Output $Q0$ changes values at each active edge of the *Clock* since the corresponding flip-flop always receives the inverse of $Q0$ in the input. With $Q1$, however, the inversion condition is only true when $Q0$ is 1 at the XOR, whereas if $Q0 = 0$, $Q1$ keeps its previous value.



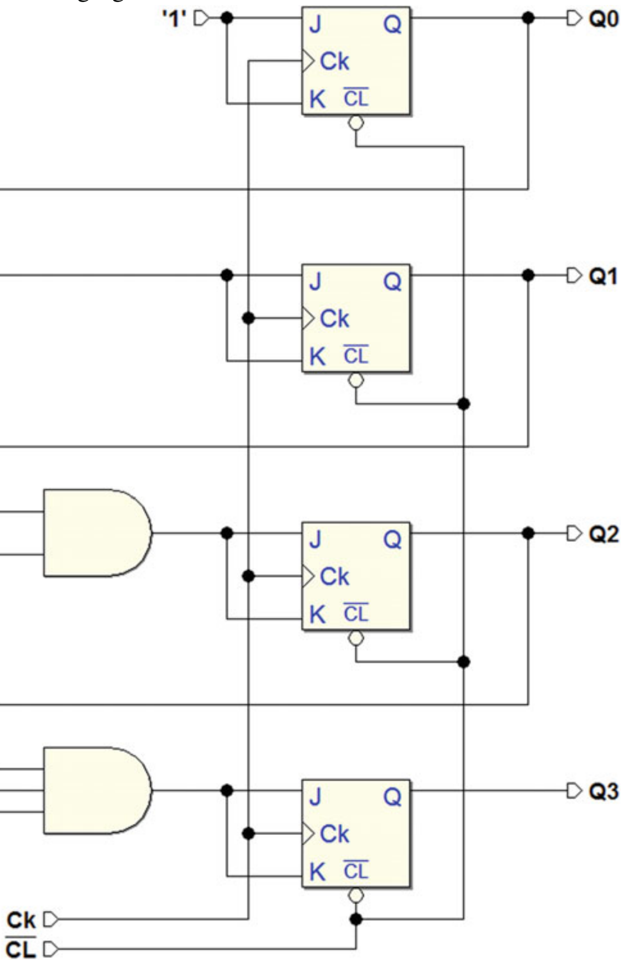
Likewise, $Q2$ changes when outputs $Q0$ and $Q1$ are both at 1, due to the two-input AND gate. Finally, $Q3$ toggles only when $Q0$, $Q1$, and $Q2$ are all at 1, due to the three-input AND.

The simulation shows that an output changes when all the lesser significant outputs are high. This observation allows us to intuitively extend the binary counter to any number of bits.

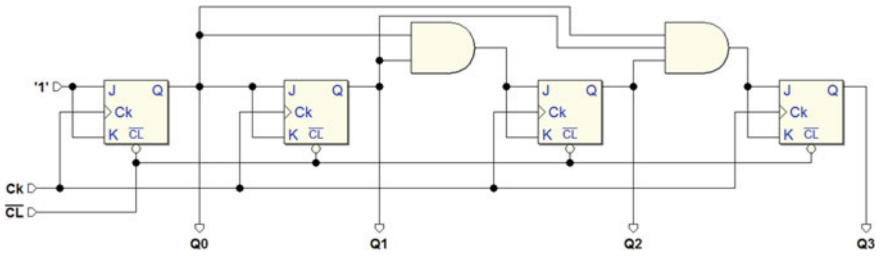
Let us now look at the inversion function, which can be obtained through the JK-PET flip-flop (this is implicit in this logical type and it is enough to connect the inputs J and K together):



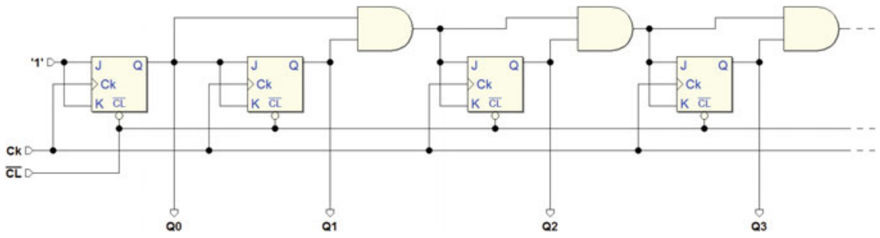
By substituting the D flip-flops with as many JKs, we simplify the previous schematic as seen in the following figure:



We can also draw the same network placing the flip-flops horizontally, as in the figure below. The advantage is a more intuitive view of the flip-flop inputs' *"cascade"* driving.

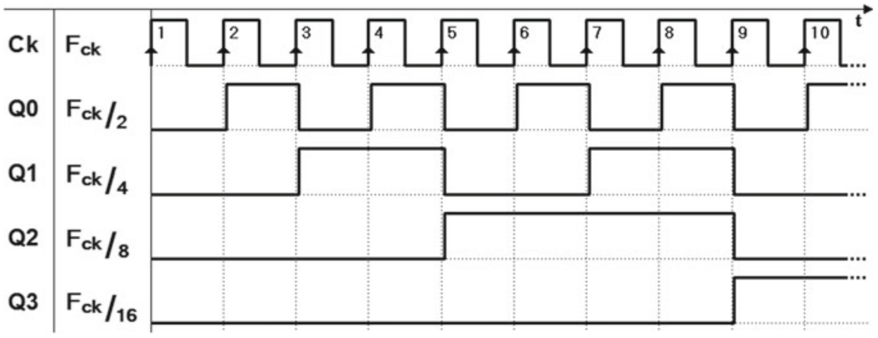


The network simplifies even more if we take AND's associative property into consideration. It allows us to use simple 2-input ANDs to arrive at the structure shown in the figure below:



Nevertheless, when the number of bits increases in this type of structure, the maximum operating frequency declines since the number of levels in the combinational network rises linearly along with the number of flip-flops.

Let's go back to the binary counter's timing simulation. An interval of this is seen below:



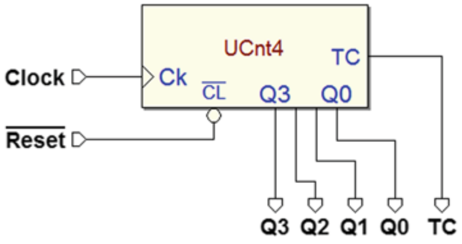
If we consider the relation between the signals' timing evolution (the so-called *waveforms*), we see that the *period* of $Q0$ is twice that of the clock Ck. Likewise, the period of $Q1$ is twice that of $Q0$ and four times the clock.

If *Fck* is the frequency of the clock signal, then *Fck/2* is the frequency of the $Q0$ signal, *Fck/4* the frequency of $Q1$ and so on. In our example, the waveforms of the outputs are also *symmetrical*; that is, the lengths of their high and low intervals are identical.

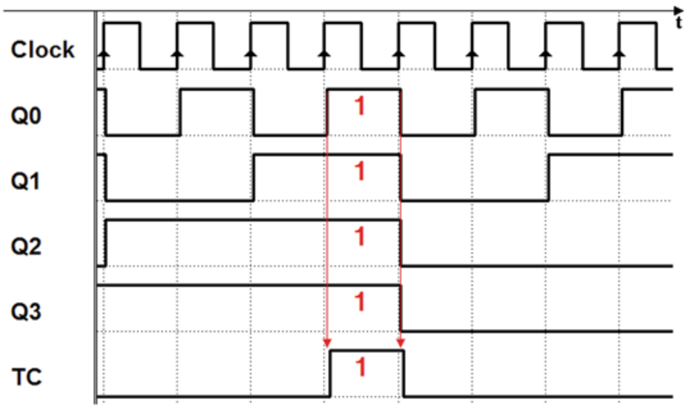
A counter therefore can be used as a *"frequency divider"*, a network that provides periodic signals derived from the clock with frequencies equal to that of the clock divided by a power of 2.

The following figure shows the "UCnt4," an example of a *synchronous, binary, 4-bit up counter* taken from the *Deeds* simulator library. This counter is functionally identical to the one described above with an added *TC* (*"Terminal Count"*) output.

TC activates when the number generated by the counter reaches the highest value, according to the simple *combinational function* $TC = Q3 \cdot Q2 \cdot Q1 \cdot Q0$.

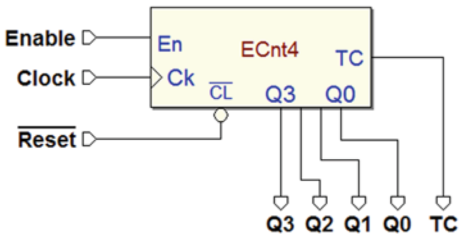


The figure below shows an example of the component's timing simulation that highlights the activation of *TC*.



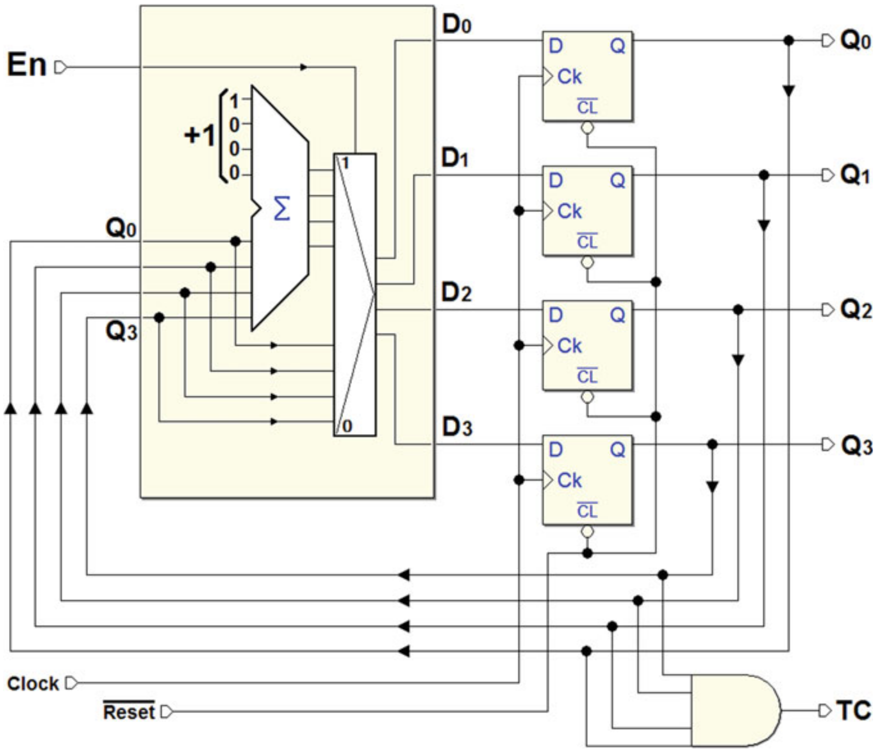
6.3.2 Counters with Enabling

The figure in the next page shows a counter that is similar to the previous one. It has an extra *enable* input *En* that controls the counting function.



The count is *enabled* only if *En* is at 1. When the count is disabled, the counter's outputs do not change despite the edges of the clock. The counter here is the "ECnt4" (*"up counter with enable 4 bits"*), taken from the library of *Deeds*.

Let's examine how enabling functions in principle. The following figure describes the function of a binary up counter 4 bits *with enabling*:



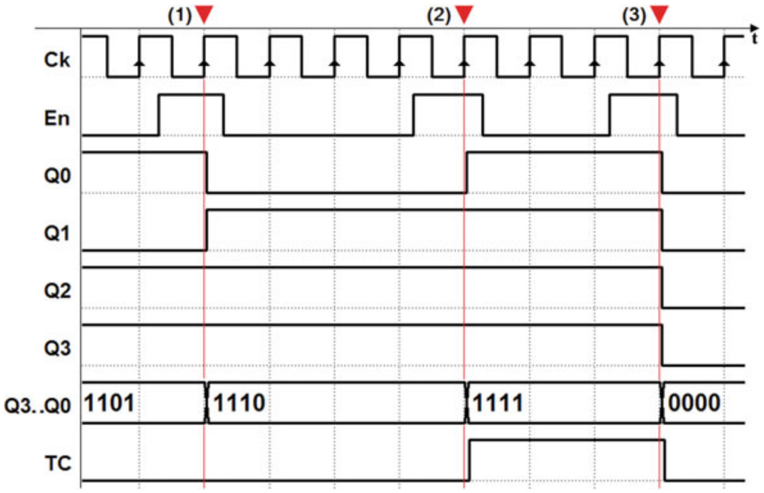
Compared to the last structure we saw, this one has a multiplexer in front of the inputs of the flip-flop that is controlled by *En*. When *En* is high, the multiplexer connects the output of the adder with the flip-flops, which brings us back to the normal count as seen previously.

When *En* is low, the multiplexer feeds back the outputs into the inputs of the flip-flop. Thus, at each edge of the clock, the previous values are confirmed (and the count halts). The *TC* is generated in function of the value taken on by the flip-flops. If the value “1111” is on the outputs *Q*3..*Q*0, it activates.

The timing diagram depicted in the next figure shows an example of the functionality of a counter with enabling. The record starts with *En* set low and the count halted (say, at the value of “1101”). The count only proceeds on the rising edges of the clock if $En = 1$. In this simulation, it is activated three times for the duration of one clock cycle.

At the first (1) activation of *En*, the count gets to the value “1110” and stops. The value will increment (“1111”) at the next (2) activation of *En*. Note that the entire time the counter has this combination of outputs, the output *TC* is activated, an indication that the counter is at its *“terminal”* value.

Then, the counter moves up one increment at the last (3) activation of *En*: its outputs go from “1111” to “0000”, because the count is cyclical and uses only 4 bits.

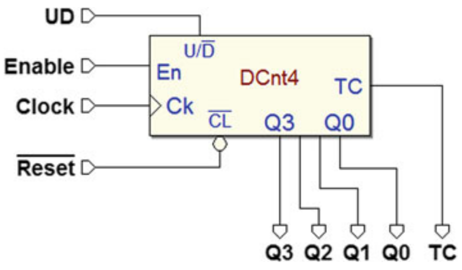


Note that $TC = 1$ signals also that the count on the next active edge of the clock will go from the maximum value to zero. In the next section, we will see that this will be useful in connecting multiple *“cascading”* counters in order to get a count with a higher number of bits.

In sum, counters with enabling make it possible to *count the number of times* input *En* is activated in response to the active edge of the clock, all while keeping the operations rigorously synchronous.

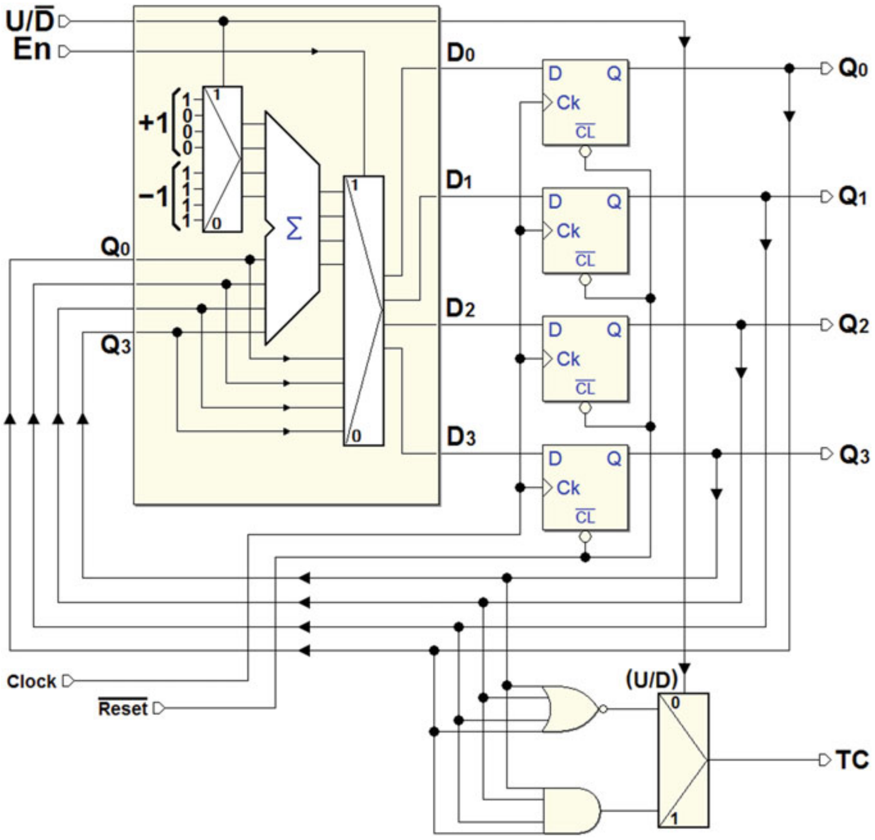
6.3.3 Up/Down Counters

Up/down counters allow to count *up* or *down*. For example, take the “DCnt4” (*“up/down counter with enable 4 bits”*) from the *Deeds* library:



Here, the *up/down counter* has the added input $U/\overline{D}$ that sets the count direction.

The up/down counter in the figure below repeats the structure of the counters seen previously but with one main difference: the content of the register can be increased or decreased.

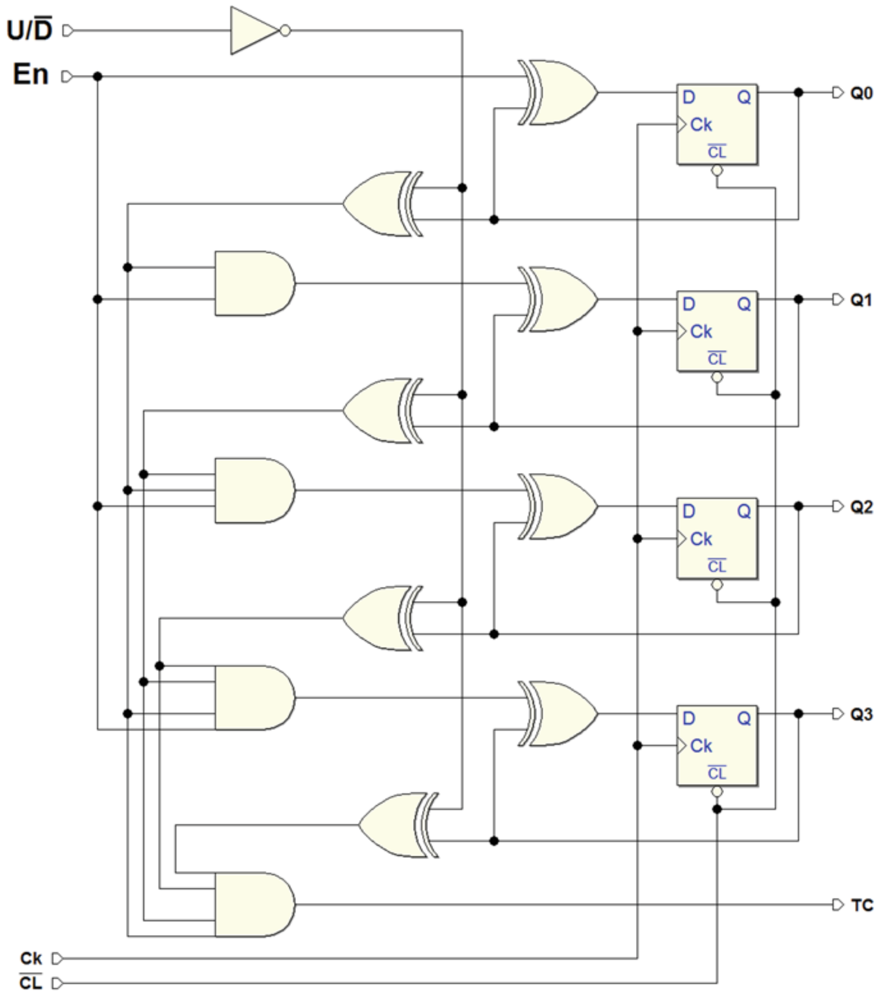


We can do this by presenting the constants +1 or −1 (represented in *two's complement code*) to the adder through the multiplexer seen in the upper left-hand corner of the figure. The choice of +1 or −1 is based on the value of the input direction control $U/\overline{D}$.

$U/\overline{D}$ controls also the multiplexer in the lower right-hand side of the figure that generates the *Terminal Count* coherently with the count direction. When counting up, TC is activated when the highest number “1111” is reached. When $U/\overline{D} = 0$, the counting is down, *TC* is set to 1 if we have reached the lowest number “0000”.

This block description finds a possible circuital synthesis in the following figure. The network is very similar in structure to that of the *up counter without enable*. Keeping the similarities in mind, let’s try to interpret the elements of this new network from an intuitive perspective.

The D-PET flip-flops are connected as T type, but four XORs have been added in the feedback loop.

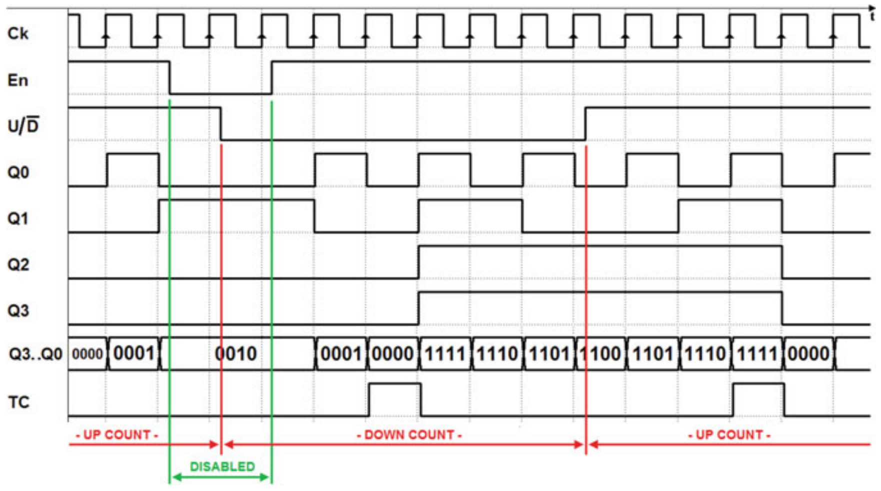


With the input $U/\overline{D}$, we can invert, or not, the value taken from the outputs of the flip-flops. If $U/\overline{D} = 1$, these XORs *do not invert*, and the network works like an up counter. If $U/\overline{D} = 0$, the network works like a down counter.

Input *En* controls all the flip-flops. If $En = 0$, all of them are forced to recharge their own value (at each active edge of the clock), so the outputs do not change, and the count does not change. If $En = 1$, however, everything works as if that input were not there and the count is enabled.

Independently of *En*, output *TC* is generated by an AND gate. AND's four inputs come from the outputs of the (direct or negated) flip-flops in function of input $U/\overline{D}$. *TC* will be activated when the outputs of the flip-flops get to "1111" if the count is *up* (or to "0000", if it is *down*).

The figure below shows an example of the timing simulation of the *up/down counter with enable*:



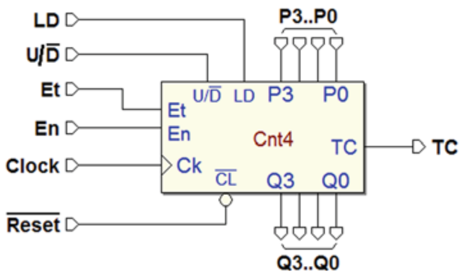
In the first part of the timing diagram, the counter is enabled ($En = 1$) and counts up ($U/\overline{D} = 1$). Then for two clock cycles, the counter is disabled ($En = 0$) and the count stops, staying at the last number it had got to, "0010". In the meantime, the counter is asked to count down ($U/\overline{D} = 0$).

When it is enabled again, ($En = 1$), it starts counting down. When it reaches "0000", *TC* is activated and it starts counting from "1111" until it gets down to "1100", at which point we order it to count up again ($U/\overline{D} = 1$). When it gets to "1111", *TC* is activated again and starts from "0000" (and so on).

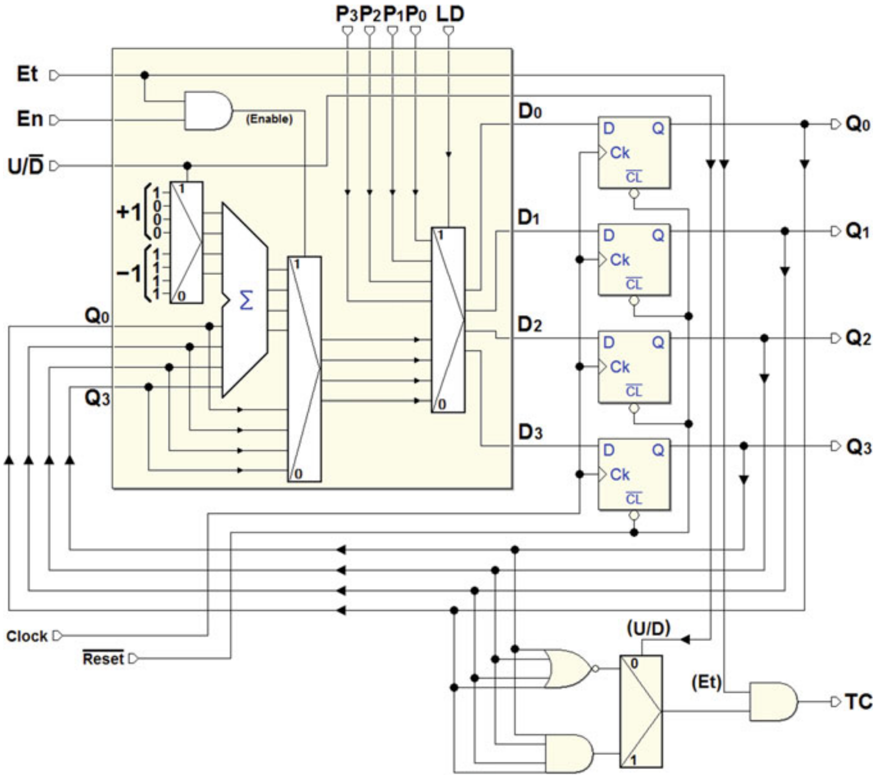
6.3.4 "Universal" Counters

The most complete counter is the *"universal"* counter. It adds the possibility to *preset* the number the counter contains, like in parallel registers. The structure presented improves enabling and *TC* output, too.

In the figure next page, we see an example of a universal counter, the "Cnt4" (*"counter 4 bits"*), from the *Deeds* library.



The figure below shows a universal counter block by block. The counter has the *preset* inputs $P3..P0$, in the same number as the outputs $Q3..Q0$, and the *load* command input LD.



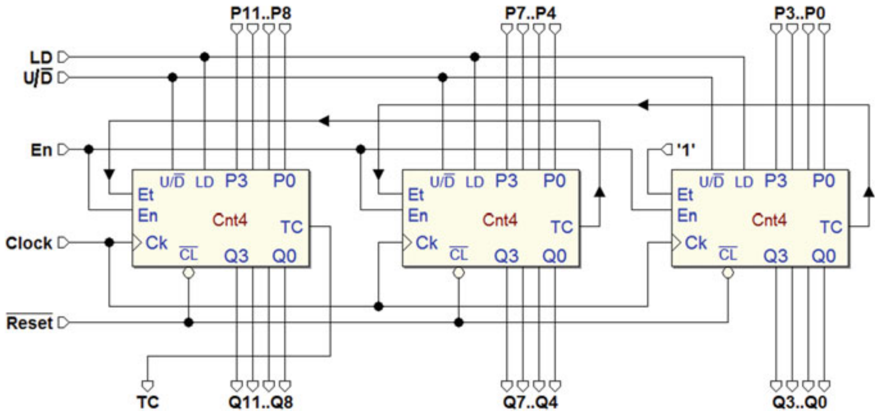
Here, a multiplexer controlled by input LD has been added to the structure of the counters seen previously. If LD is at 1, the number set on the inputs $P3..P0$ is routed to the D of the flip-flops, and it will be loaded into the counter on the next rising edge of the clock. If $LD = 0$, however, the counter works like the previous types. For example, the functionality of the direction input $U/\overline{D}$ is the same.

There are now two enable inputs: *En* (*"Enable"*) and *Et* (*"Enable Terminal Count"*). Both *En* and *Et* must be active to enable the count. Since *Et* enables the generation of output *TC*, it is used separately from *En* with multiple cascading counters, as we will see further on.

Remember that in all the counters seen previously, *TC* function could not be disabled and its value depended only on the direction of the count and the output values.

Counter Extension

The synchronous structure of a universal counter lends itself to the *extension of the number of bits* by using multiple interconnected *"cascading"* devices. For example, we see in the figure below that a 12-bit counter has been obtained through three 4-bit counters.



As we can see, all three devices share the clock signal, so the entire structure is *synchronous*. Inputs $\overline{Reset}$ and $U/\overline{D}$ are also shared, so the three counters will be initialized together and will always count in the same direction.

The enable input *En* is shared as well as the load command *LD*. The counter on the far right-hand side of the figure is used for the less significant bits ($Q3..Q0$) and the farthest left, for the most significant ($Q11..Q8$).

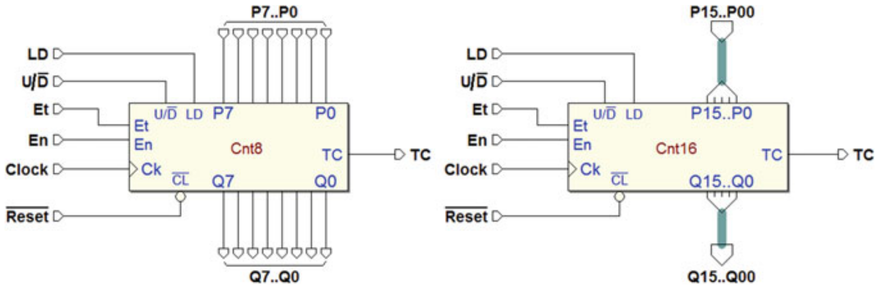
Now let's assume that the *En* command is set to 1 and the direction is up. The counter that generates $Q3..Q0$ will be enabled since its input *Et* is at 1. So that the middle counter, which produces $Q7..Q4$, counts only when it should, we connect its input *Et* to the *TC* of the counter that generates $Q3..Q0$.

This way, *TC* is used like a *"carry"*: when the farthest right counter gets to the maximum number, it instructs the middle one to count up one unit by enabling it with $Et = 1$.

There is an analogous connection between the *TC* of the middle counter and the farthest left *Et* input. The *TC*, which is produced by the farthest left counter, will only be active when *all three* counters reach "1111".

This structure can be extended to more bits, but we must remember that every counter we add increases the overall propagation delay of the combinational network, which propagates the *Et* enable signals through the *TC* of the various components.

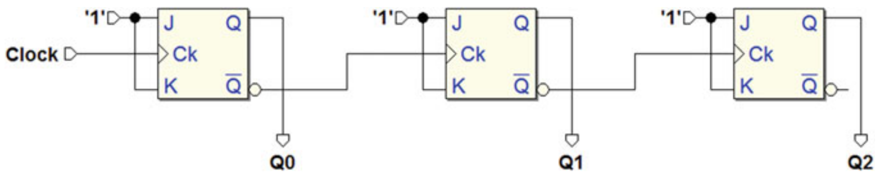
In this book, we have used 4-bit counters for simplicity's sake. Obviously, in CAD system libraries, we find components of any size, like the 8- and 16-bit counters in the figures below.



The 8-bit counter on the left is shown in its normal version. The 16-bit counter on the right is shown in the *bus-type* connections' version.

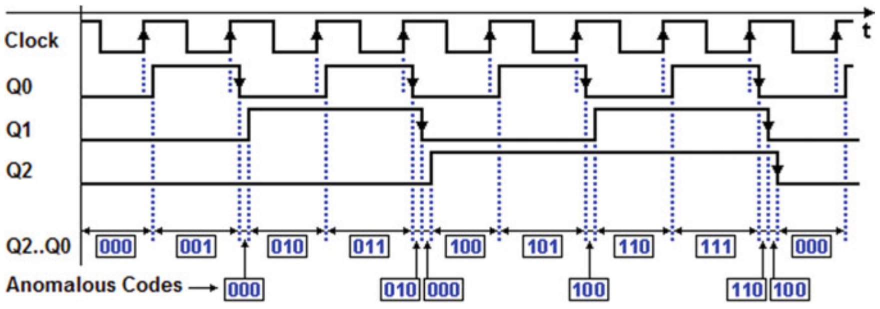
6.3.5 Asynchronous Counters

A counter is *asynchronous* when the flip-flops it is made of *do not all share* the same clock. The network in the figure below represents an asynchronous, binary up counter where all JK flip-flops but the first use the $\overline{Q}$ of the one before it as a clock signal.



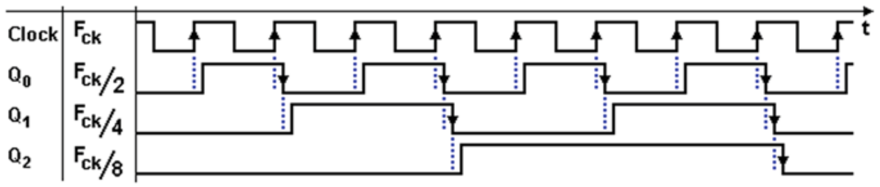
Every flip-flop in this network (active on the rising edge of its *own Ck* input) changes state when the output *Q* of the previous flip-flop switches *from one to zero*. Because of this behavior, this is called a *"ripple counter"*, a term that recalls a wave-like propagation.

The evolution of the outputs is described in the figure below where the propagation times of the flip-flops have been highlighted (in approximate terms). Note that the delay between the *Clock* at the input of the counter and any given output grows *proportionally* to the position (weight) of that output.



If we ignore the delays, this counter follows an *up binary sequence*, but the asynchronous commutation of the outputs generates anomalous codes that alter this sequence, as shown in the figure. These codes have a short duration (the same as the propagation time of the flip-flop), but they can create problems in a network that processes its outputs, for example a decoding network. This is why asynchronous counters are used only in special cases.

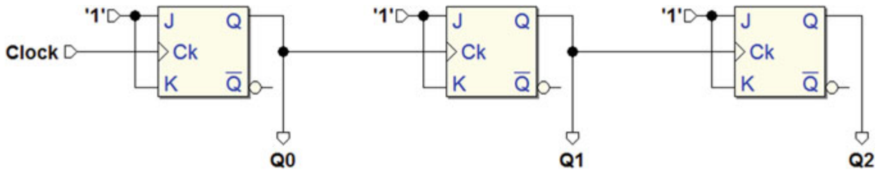
As we know, a counter can be considered a *frequency divider*, when we consider the timing relation between the waveform of the *Clock* and that of any of the outputs. An asynchronous counter can be used to good advantage to this purpose. As we see in the figure below, the frequency of outputs $Q0$ is $1/2$ of that of the *Clock* and outputs $Q1$ and $Q2$ provide a frequency signal of $1/4$ and $1/8$, respectively.



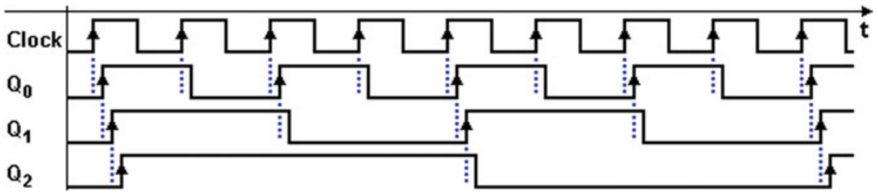
In counters used as frequency dividers, the asynchronous outputs do not pose a problem since the signals generated are often used independently of each other. Also, the simplicity of the asynchronous counter versus the synchronous one is a great advantage for very high *"division ratios"*.

Frequency dividers are used in telecommunications devices where they generate signals whose frequency is a submultiple of that of a *Clock* generator, which works at a higher frequency.

To be thorough, below, we show an asynchronous *down* counter. In this network, the input *Ck* of each flip-flop except the first is connected to the output Q of the one before it.



In the timing diagram below, we see that the flip-flop's change in outputs in the backward count occurs when the one before it makes a transition *from zero to one*.



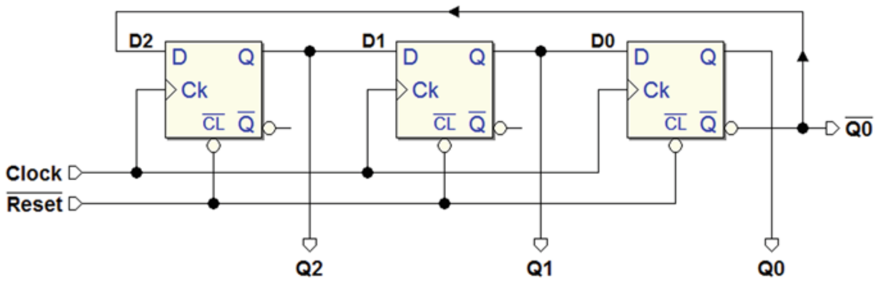
6.4 Network Analysis Examples

One essential step in beginning to design digital systems is to understand the behavior of a given sequential network. The timing analysis of a network provides familiarity with some general aspects of the interaction between the combinational and sequential components of a logical circuit. The familiarity with the *low-level* behavior of sequential networks is an important step for the designer who is mindful of the workflow from the project specifications through to the final product.

Next, we will show a series of examples of analyses through the *timing diagrams* of simple sequential networks that have a given logical schematic associated with suitable input signal sequences. We will carry out a functional network test; that is, we will study the evolution on time of the outputs as a function of the inputs.

6.4.1 Example 1

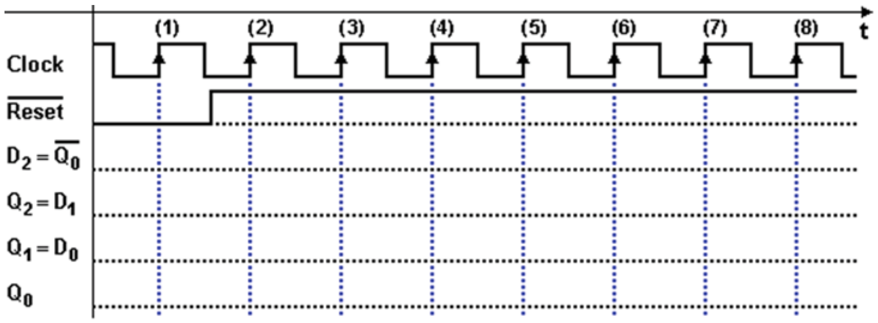
In this section, our goal is to analyze the function of the network of flip-flops depicted in the figure below. A simple observation of the schematic gives useful indications on how to analyze it. This is a synchronous network made of D-PET flip-flops with an asynchronous initialization input $\overline{Reset}$ (which acts on the inputs $\overline{Clear}$ of the flip-flops). The network generates the three outputs $Q2$, $Q1$, and $Q0$.



The structure of the network is also easy to identify: a shift register where the serial input $D2$ is connected to the negated output $\overline{Q0}$ of the last flip-flop. It is not necessary to identify the specific structure to analyze it. The procedures shown here work for any synchronous network of flip-flops.

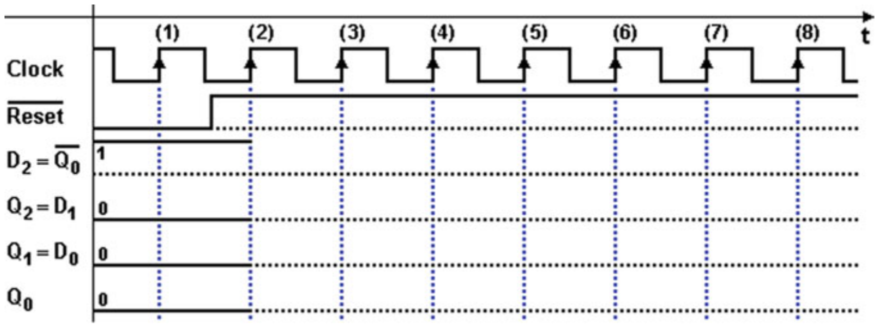
To start, we must first have a *timing diagram* where we can sketch the evolution of *Clock*, and the inputs and outputs of the network. In the case at hand, we will insert signal $D2$ ($= \overline{Q0}$) in the diagram for ease of examination.

The next figure shows the track of the diagram to construct. Here, the *Clock* and the initialization signal $\overline{Reset}$ have been defined. We suppose $\overline{Reset}$ active at the beginning of the diagram and then deactivated in the interval between the edges (1) and (2) of the *Clock*.



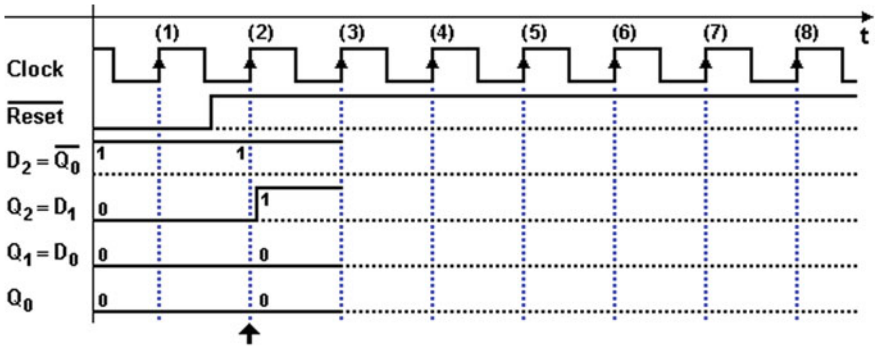
As long as input $\overline{Reset}$ is kept active (low), outputs Q of the flip-flops are *forced* to zero and edge (1) of the *Clock* cannot provoke changes. Note that we must draw $D2$ with the value of 1 in this initialization phase since it is connected to the negated output $Q0$.

Remember that deactivating $\overline{Reset}$ does not change the state of the network and the signals remain unchanged until the next active edge of the *Clock* (2).



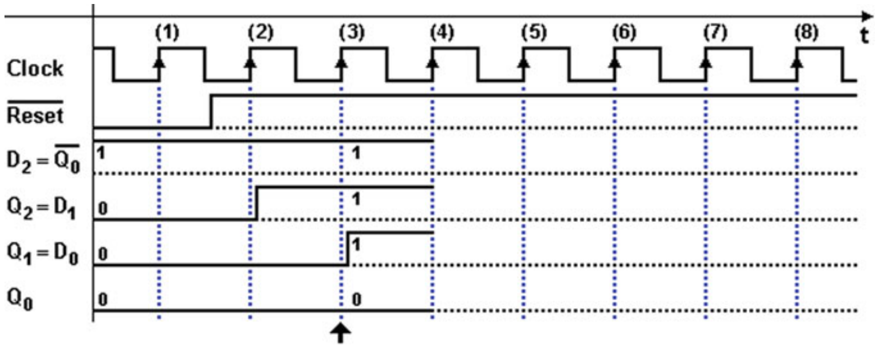
At every active edge of the *Clock*, the D-PET flip-flops transfer the logical value that is on their own input D at that moment onto their output Q.

On edge (2) of the figure above, inputs $D2$, $D1$, and $D0$ of the flip-flop are 1, 0, and 0, respectively. Therefore, let's draw the outputs of the flip-flops after edge (2) the figure in the next page.

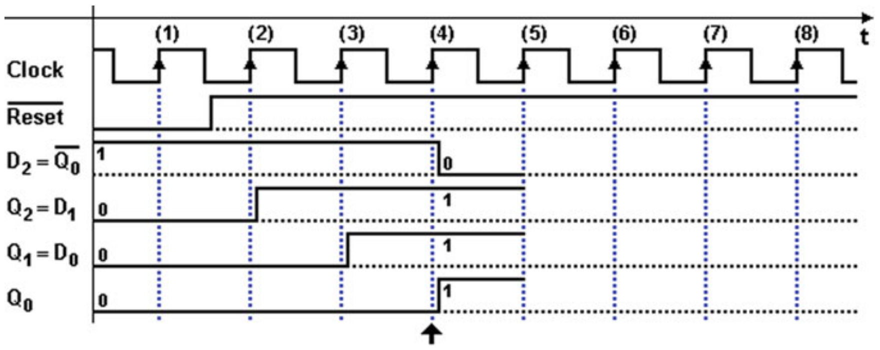


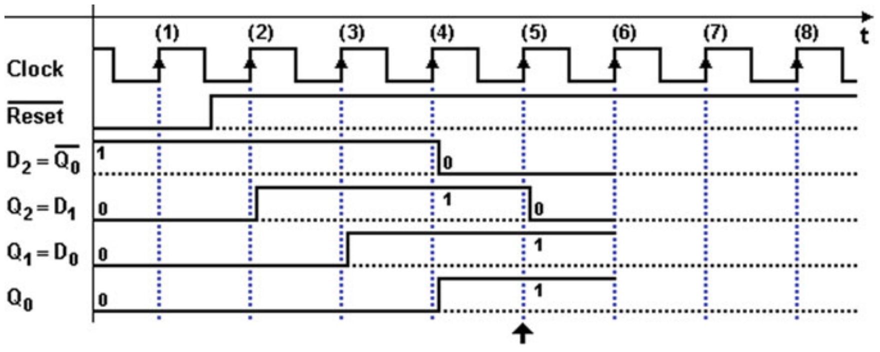
In the previous figure, we have chosen to highlight the propagation delay between the edge of the *Clock* and the change of output $Q2$. Note that the new value taken by $Q2$ (and thus by $D1$) will be acquired by flip-flop $Q1$ on the next edge (3).

Up until edge (3), the situation remains the same, given that flip-flops change their state only on the edge of the *Clock*. On edge (3), the values on inputs D are transferred to outputs Q, in the same way as on edge (2). In the figure below, we see the situation after edge (3). Note that there is a delay in the activation of $Q1$.

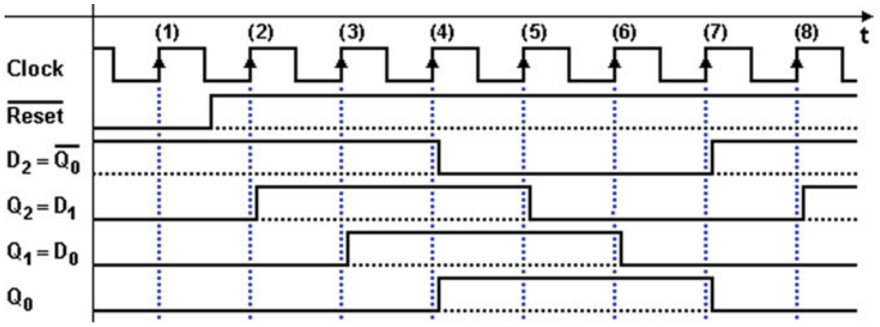


In the two following diagrams, we continue drawing the diagram in relation to edge (4) and (5) by applying the same criteria.





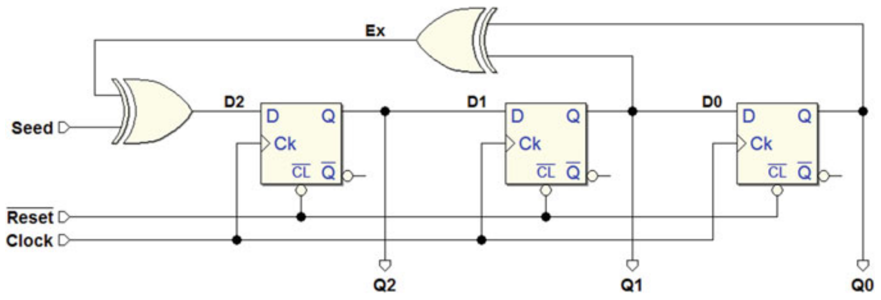
Finally, in the figure below, we see the complete timing diagram. The figure shows the typical behavior of a *shift register*.



The network we have examined is called a 3-bit *"Johnson Counter"*. It can be made with a shift register with any number of bits by connecting the negated output of the last flip-flop to the input of the first. It has the advantage of a simple structure and the corresponding disadvantage of a counting code which is different from pure binary (but can be easily decoded).

6.4.2 Example 2

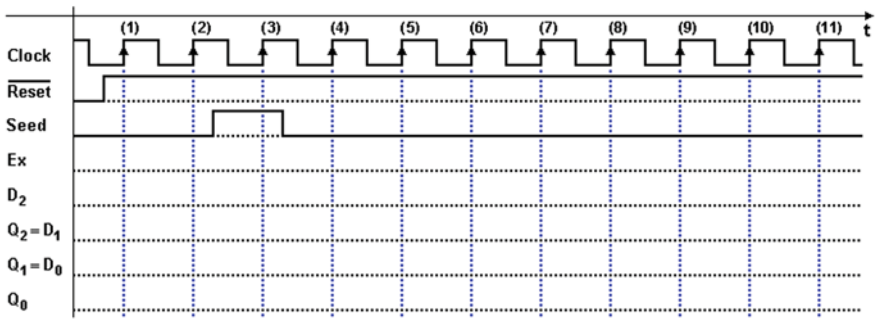
The network in the following figure is made up of three D-PET flip-flops with shared *Clock* and $\overline{Reset}$ signals. It is easy to find the base structure of the shift register but the input of the farthest left flip-flop, $D2$, is obtained by an XOR tree that processes the outputs $Q1$ and $Q0$, and the input *Seed*.



We know that, at every active edge of the *Clock*, the flip-flops transfer the logical value on input D at that moment onto output Q. The network analysis consists in evaluating inputs $D2$, $D1$ and $D0$ in relation to those active edges. Based on the figure above, we can write the Boolean expressions. Note that only the first one describes a network with logical gates, while the others are simple connections.

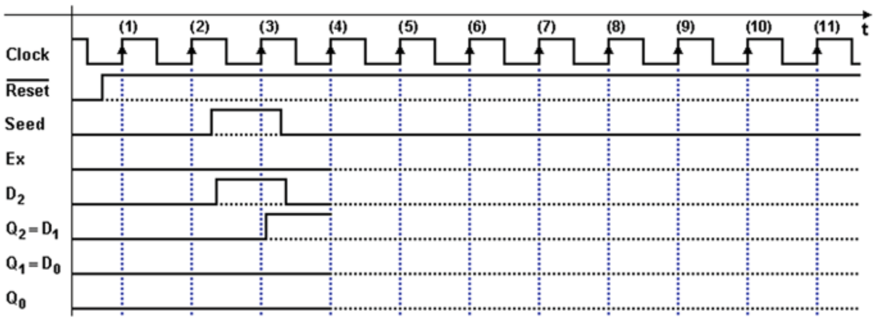
$$D2 = Seed \oplus Q1 \oplus Q0; \qquad D1 = Q2; \qquad D0 = Q1$$

These three expressions provide the values of inputs $D2$, $D1$ and $D0$ as function of the values of input Seed and flip-flop outputs $Q2$, $Q1$, and $Q0$ (the current state of the network). Once these are loaded on the flip-flop, they will constitute the *next state* of the network. Here too, we trace the behavior of the network on a timing diagram seen in the figure below. We prepare traces for all the network's inputs and outputs, the flip-flops' inputs and even the intermediate signal *Ex* for ease of analysis.



The next figure shows the timing analysis up until edge (4). The $\overline{Reset}$ signal, which is active before edge (1) of the *Clock*, sets the flip-flop's outputs to 0. It is easy to understand why the *Seed* activation is necessary to make the network evolve. In its absence, the network would remain in the situation set by $\overline{Reset}$ indefinitely.

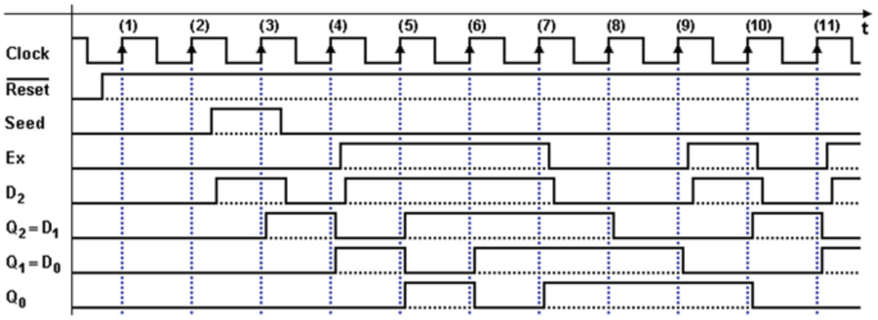
Inputs $D1$ and $D0$ are also at 0, since they are connected to outputs $Q2$ and $Q1$. $D2$ is at 0, as we can see from the Boolean expression since the external input *Seed* is set to 0. Therefore, edge (1) of the *Clock* does not change its outputs, which remain at 0. The same goes for edge (2).



In the initial setting of the diagram, we assumed that the input *Seed* is activated for the duration of the *Clock* cycle between edges (2) and (3). The immediate result is that $D2$ is activated: on edge (3), the output $Q2$ switches to 1, while the other flip-flops do not change their values ($D1 = D0 = 0$).

After edge (3), the input *Seed* returns to 0 and remains to this value until the end of the diagram, so the evaluation of $D2$ is simplified, since it now depends only on the variations of outputs $Q1$ and $Q0$.

To continue the analysis, after *Seed* is brought to zero, the networks that generate $D2$, $D1$, and $D0$ produce the values 0, 1, and 0, respectively. These are transferred onto the flip-flop outputs at edge (4). With the same method, let's continue the analysis until we complete the diagram shown here:

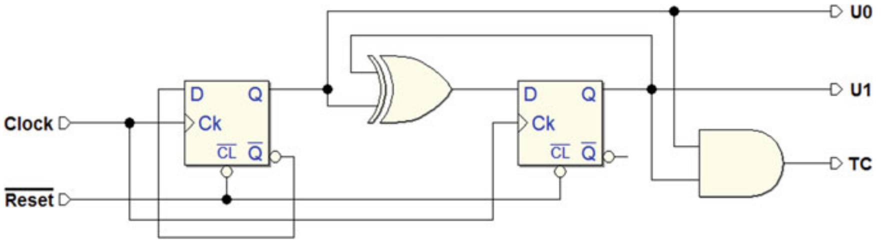


We can make some general comments about the complete diagram. All the flip-flop outputs commute at the active edges of the *Clock* with a delay that is equal to their propagation time. Signals *Ex* and $D2$ show an added delay due to the combinational network that generates them. As we can see in the diagram, signal $D2$ can change asynchronously with respect to the *Clock* between edges (2) and (4) because it is dependent on external input *Seed*.

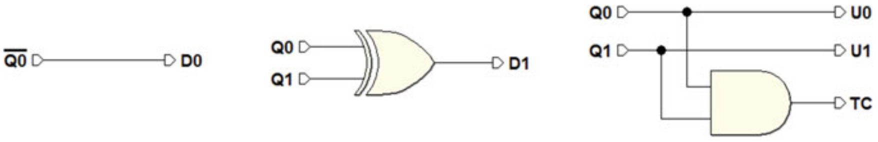
The network we have analyzed is a simplified example of a *pseudo-random number generator*. The number generated in this case is made up of outputs $Q2$, $Q1$, and $Q0$. A pseudo-random number generator is normally created with a high number of flip-flops (ex. 32) because the sequence generated is only *apparently* random; it actually repeats cyclically.

6.4.3 Example 3

The network in this example uses two D-PET flip-flops with shared *Clock* and $\overline{Reset}$ signals and without command inputs. Outputs $U0$ and $U1$ are taken directly from outputs $Q0$ and $Q1$ of the flip-flops while TC is obtained through a logical gate.



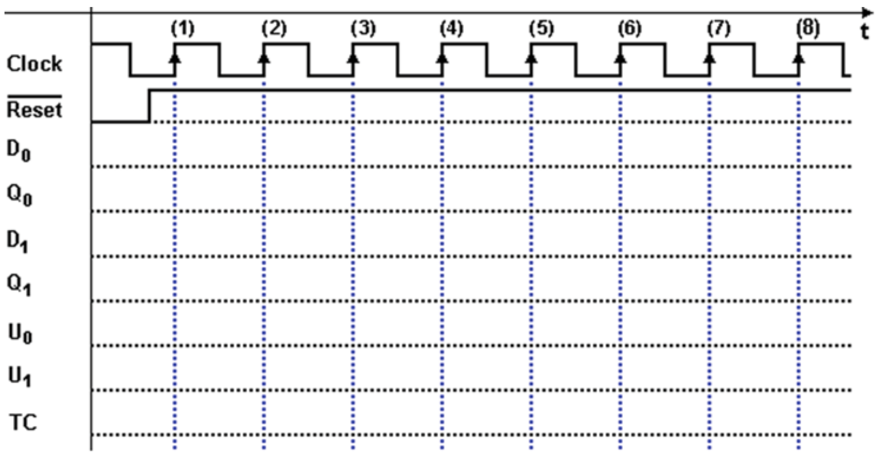
Let's apply the same analytical process as before. We evaluate $D0$ and $D1$ at the active edge of the *Clock* since at that time they will be loaded onto the flip-flops. It may be useful to *"separate"* the combinational networks that produce $D0$, $D1$ and the outputs $U0$, $U2$, and TC from the overall schematic. Let's re-draw them apart, in the form of a circuit and as Boolean expressions.



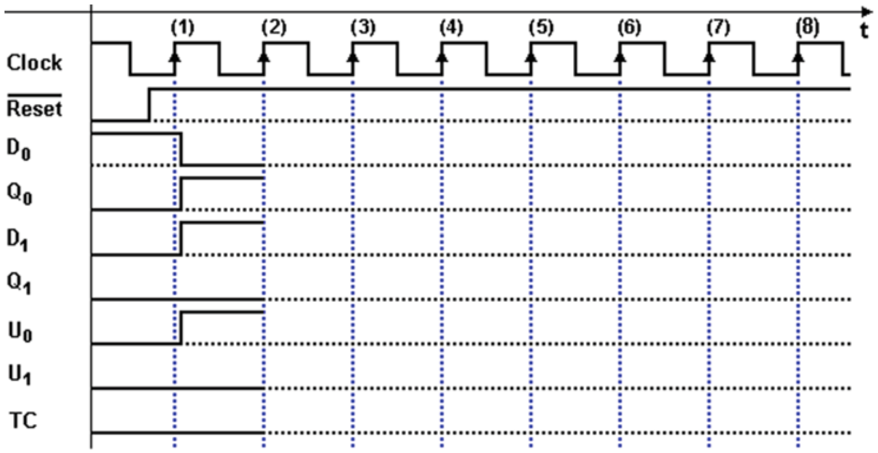
$$D0 = \overline{Q0}; \quad D1 = Q0 \oplus Q1;$$

$$U0 = Q0; \quad U1 = Q1; \quad TC = Q0 \cdot Q1$$

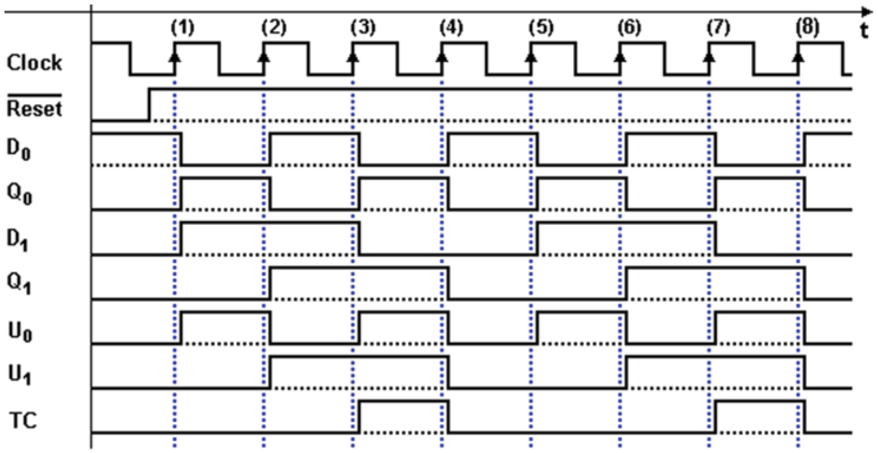
By using the schematics or expressions just described, we can trace the timing diagram. Aside from the *Clock*, let's trace the signal $\overline{Reset}$ in the diagram, as active from the beginning and deactivated just before edge (1) of the *Clock*.



The first step is to consider $\overline{Reset}$, which forces the flip-flops to 0 at the beginning and then is removed. The state of the network changes at *Clock* edge (1) when the two flip-flops take on the values of $D0$ and $D1$.



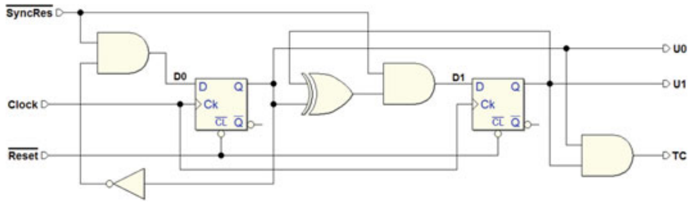
Now, we suggest that for practice the readers continue the analysis on their own, following the criteria suggested so far. The timing diagram will look like the following figure, where we see the *cyclical* quality of the sequence, which repeats every four edges of the *Clock*.



This device behaves as a *counter*; the values taken on by outputs $U1$ (MSB) and $U0$ (LSB) follow a *module 4 binary natural up count*. *TC* signals when the outputs have reached their highest value.

6.4.4 Example 4

The network in the figure below is very similar to the previous one. It uses two D-PET flip-flops and has the same outputs $U0$, $U1$, and TC, but it has an added input $\overline{SyncRes}$ ("*Synchronous Reset*") and the logic associated with it. The *Clock* and $\overline{Reset}$ network are identical to the previous case.

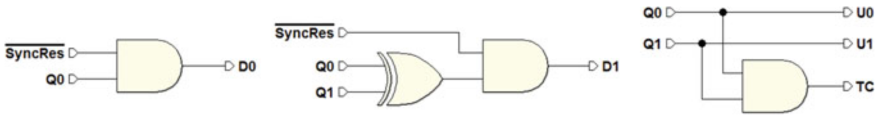


A couple of intuitive points can help the analysis of this network.

If the input $\overline{SyncRes}$ is 1, the two ANDs conditioned by this signal transmit the value of their other input to their respective outputs and the network is functionally identical to the one in the previous exercise.

If $\overline{SyncRes}$ equals 0, the outputs of both the ANDs are at 0, setting $D0$ and $D1$ to 0 and so the two flip-flops are brought to zero *synchronously*.

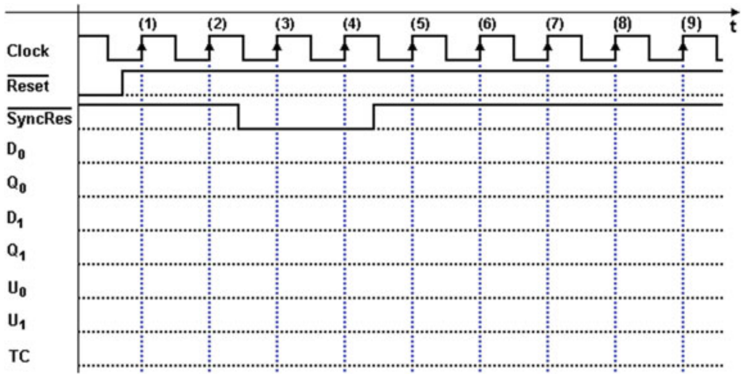
Let's derive the combinational networks that produce $D0$ and $D1$, and/or the expressions. The $U0$, $U1$, and TC network is identical to that of the previous exercise.



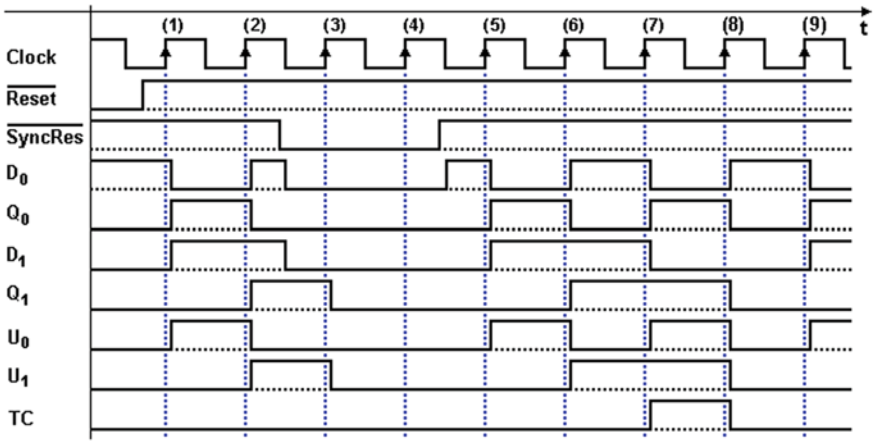
$$D0 = \overline{SyncRes} \cdot \overline{Q0}; \quad D1 = \overline{SyncRes} \cdot (Q0 \oplus Q1);$$

$$U0 = Q0; \quad U1 = Q1; \quad TC = Q0 \cdot Q1;$$

Let's set up the timing diagram as in the figure below, with the *Clock* and $\overline{Reset}$ signals set as in the previous exercise. Let's assume that the $\overline{SyncRes}$ command is activated for two Clock cycles as drawn here:

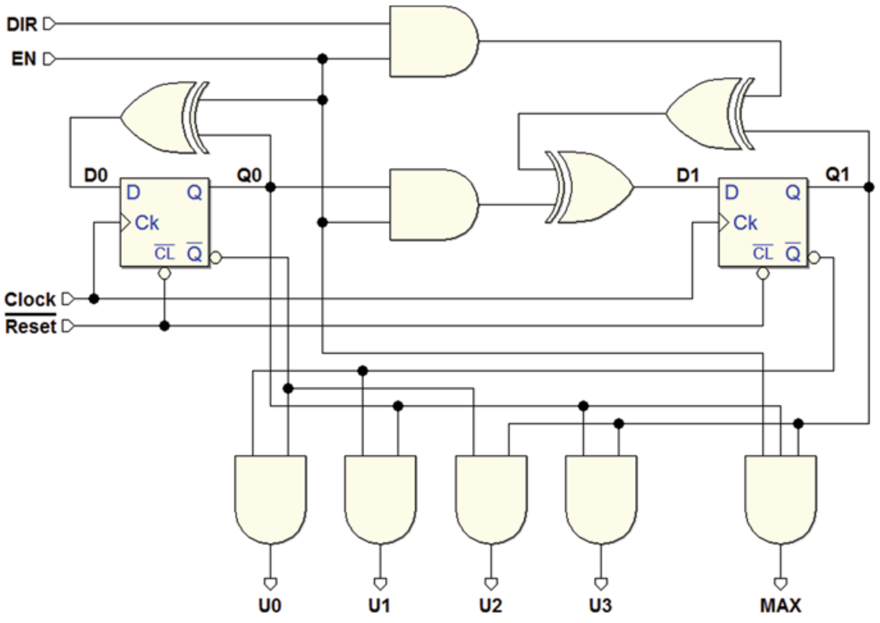


Now, it is the reader's job to complete the timing layout, which will turn out to be like the figure below. Notice that signals $D0$ and $D1$ respond immediately (except for propagation times) to $\overline{SyncRes}$ command variations.

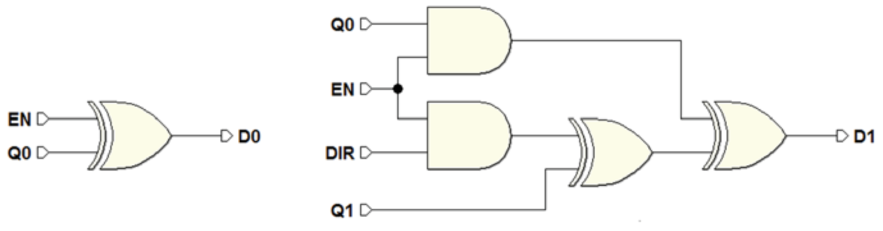


6.4.5 Example 5

The figure below shows a network that uses two D-PET flip-flops that share the same *Clock*. Both flip-flops are connected to the asynchronous initialization input $\overline{Reset}$. The two inputs are *EN* and *DIR*. The outputs generated by the networks are $U0$, $U1$, $U2$, $U3$, and *MAX*.



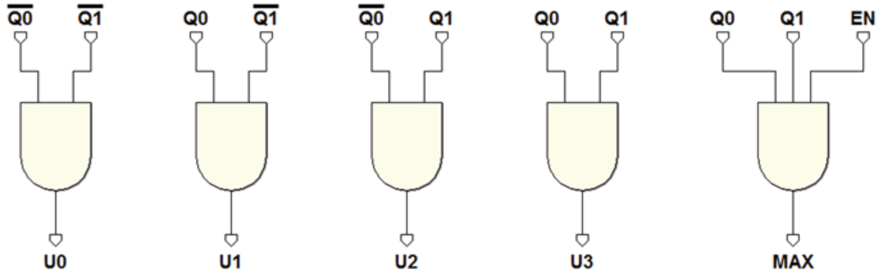
As before, the fundamental step to analyze the network is to evaluate $D0$ and $D1$ at the active edges of the *Clock*. Let's separate the combinational networks that produce $D0$ and $D1$ from the full schematics and describe them in terms of Boolean expression as well.



$$D0 = EN \oplus Q0; \qquad D1 = (EN \cdot Q0) \oplus (Q1 \oplus (EN \cdot DIR))$$

These networks combine the values of inputs *EN* and *DIR* with the values of outputs $Q0$ and $Q1$ of the flip-flops (the *"state"* of the network), and they produce a new value for $D0$ and $D1$. On the active edge of the *Clock*, these values will be loaded and will constitute the *"next state"* of the network.

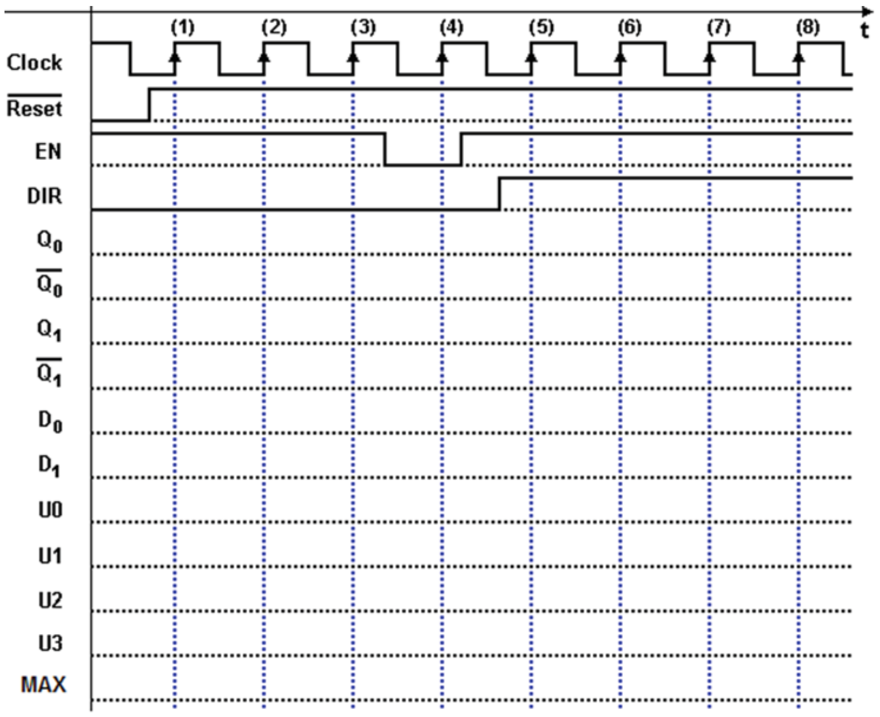
The network's outputs $U0$, $U1$, $U2$, $U3$, and *MAX* are combinational functions of the flip-flops' outputs and of the input *EN*, as shown below, both as a network schematic and in terms of Boolean expressions.



$$U0 = \overline{Q0} \cdot \overline{Q1}; \quad U1 = Q0 \cdot \overline{Q1}; \quad U2 = \overline{Q0} \cdot Q1; \quad U3 = Q0 \cdot Q1;$$

$$MAX = Q0 \cdot Q1 \cdot EN$$

For the timing analysis, inputs *EN* and *DIR* should be set in a way to avoid an unrepresentative timing diagram. We have chosen to include the flip-flops' inputs and outputs in the diagram to make the analysis easier.

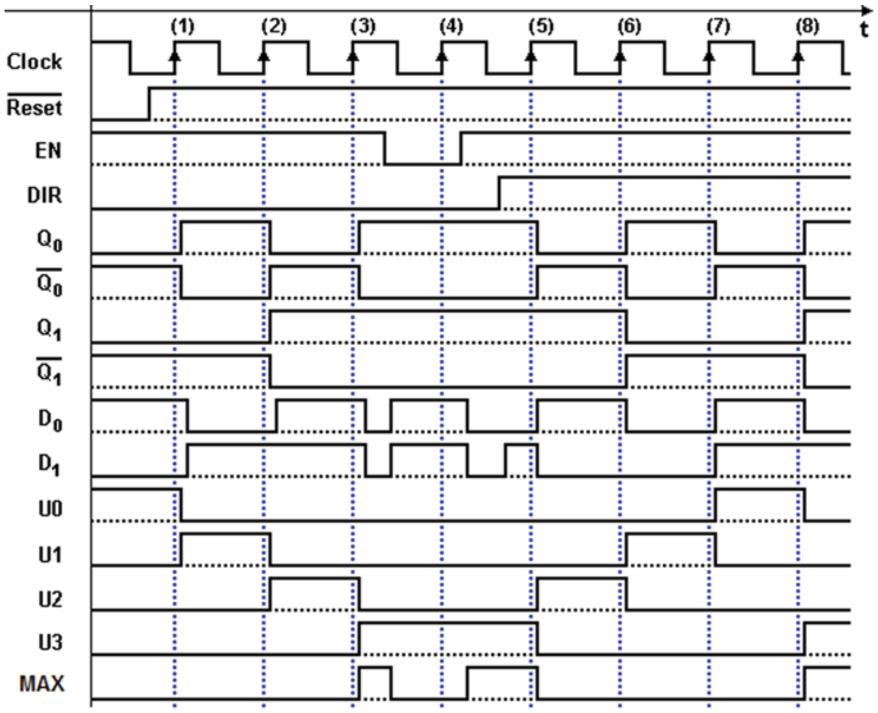


The same criteria of analysis we have seen before are appropriate to analyze this network. We suggest that the reader carries out the analyses personally.

Below we provide some advice on how to proceed:

1. We should focus initially on the evolution of the state of the network and then afterwards on the generation of outputs. So, we should first trace the evolution of signals $Q0$ and $Q1$ (direct and negated), and $D0$ and $D1$.
2. Even if we assume, as usual, that the propagation delays are short with respect to the *Clock* period, it is very useful to represent them in the diagram, albeit in a qualitative way.
3. Note that after the activation of $\overline{Reset}$, the network starts to evolve with edge (1) of the *Clock*.
4. After each active edge of the *Clock*, we must recalculate the values of $D0$ and $D1$, which will be loaded onto the flip-flops at the next active edge.
5. Between edges (3) and (4), *EN* changes. As a result, $D0$ and $D1$ immediately follow this change and the values of $D0$ and $D1$, which are transferred onto $Q0$ and $Q1$ are sampled by the flip-flops at edge (4).
6. Likewise, between edges (4) and (5), both *EN* and *DIR* change.
7. When the analysis of the next state is finished, we trace the outputs $U0$, $U1$, $U2$, and $U3$. These outputs can only change on the active edges of the *Clock*. This does not apply to *MAX*, which depends also on an input.

The figure below reports the results of the analysis. We see in the timing diagram that the flip-flops' outputs $Q0$ and $Q1$ follow a binary up counting sequence in the first few cycles after $\overline{Reset}$ is activated.



When input *EN* is at 0, the network loads the preexisting values onto the two flip-flops. On edge (4), $Q0$ and $Q1$ do not change. As on edge (5), the count changes direction since the value on input *DIR* changes.

The circuit, evaluated on outputs $Q0$ and $Q1$, behaves like an *up/down binary counter*. Input *EN* enables the count (if $EN = 1$), while *DIR* sets the direction (*down* if $DIR = 1$).

As we see in the timing diagram, outputs $U0$, $U1$, $U2$, and $U3$ are activated by combinations $00, 01, 10,$ and 11 of $Q1$ and $Q0$, respectively. Output *MAX* decodes the same combination of $U3$, but it is enabled only if $EN = 1$. Thus, the output is brought to 0 asynchronously following the evolution of *EN* between edges (3) and (5).

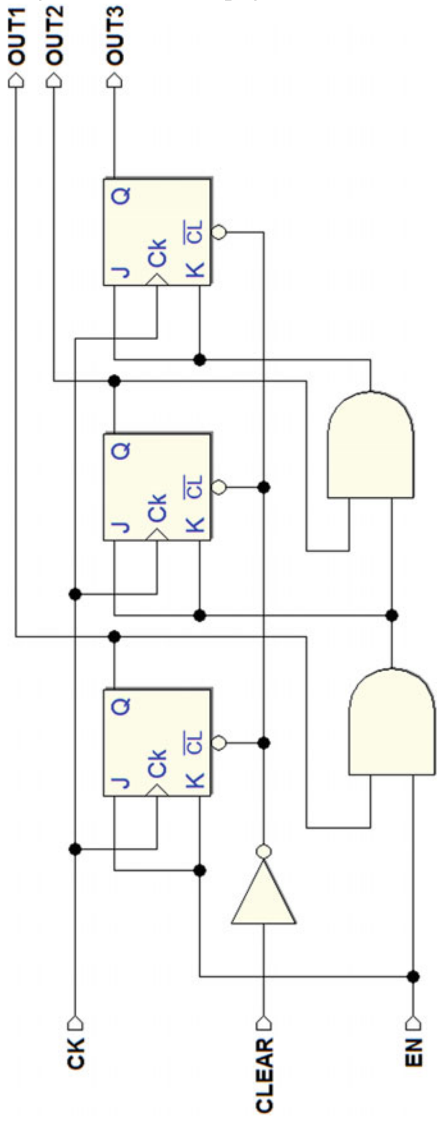
Finally, note that $D0$ and $D1$ evolve asynchronously between edges (3) and (5), since they follow the changes of inputs *EN* and *DIR*. What matters, however, is that their values should be stable at the moment they are read (i.e., on the rising edge of the *Clock*).

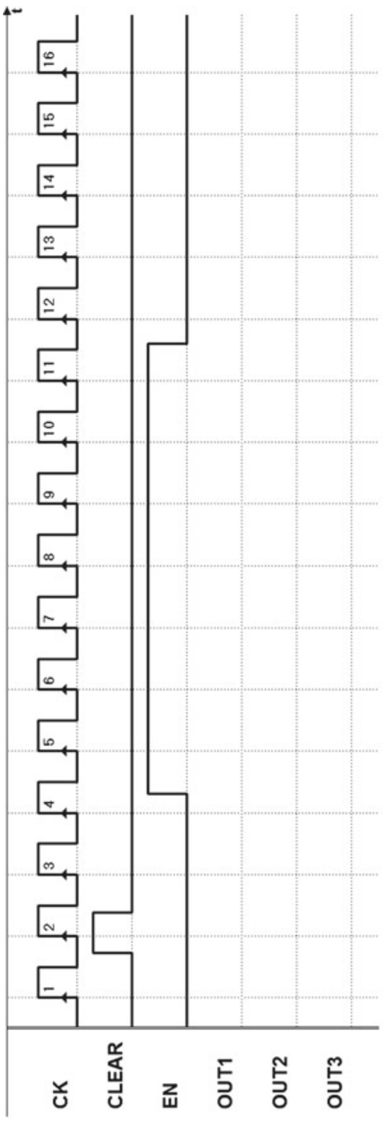
6.5 Exercises

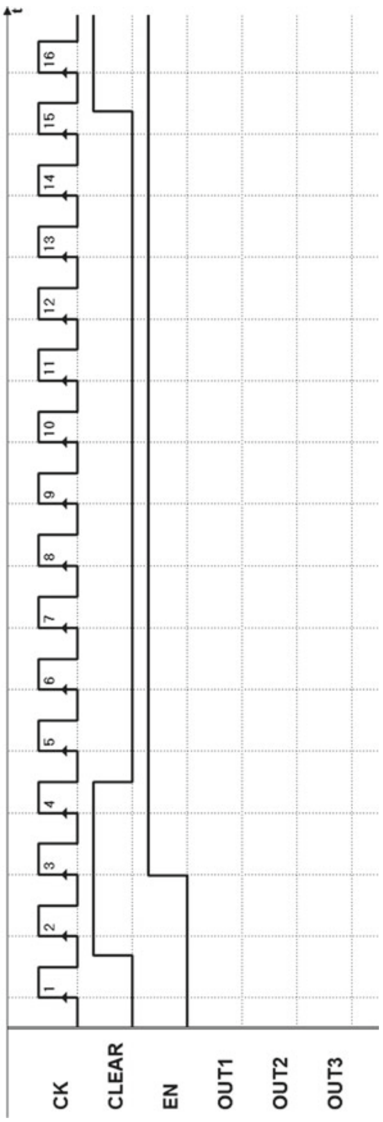
Analyze the following synchronous sequential networks by completing the timing diagrams on the side page. The templates are also available on the simulator Web site, as *PDF* files, on the *digital contents* pages.

It is advisable to complete the layouts on paper *without* the help of *Deeds*, using it only to check the solutions. The Web site also provides the Deeds files of the networks proposed.

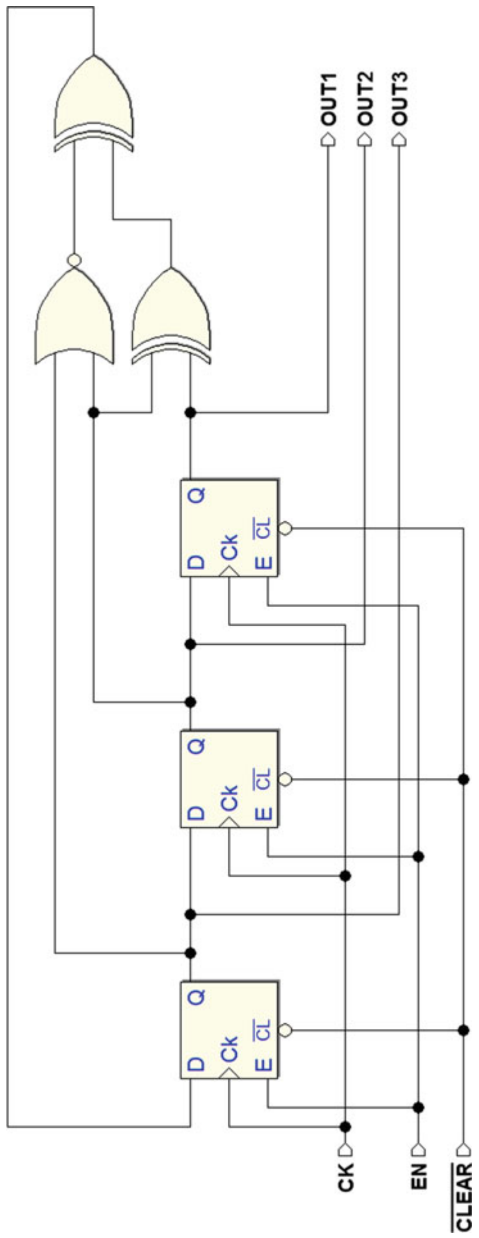
1. Exercise 1—(timing diagrams in the next page)

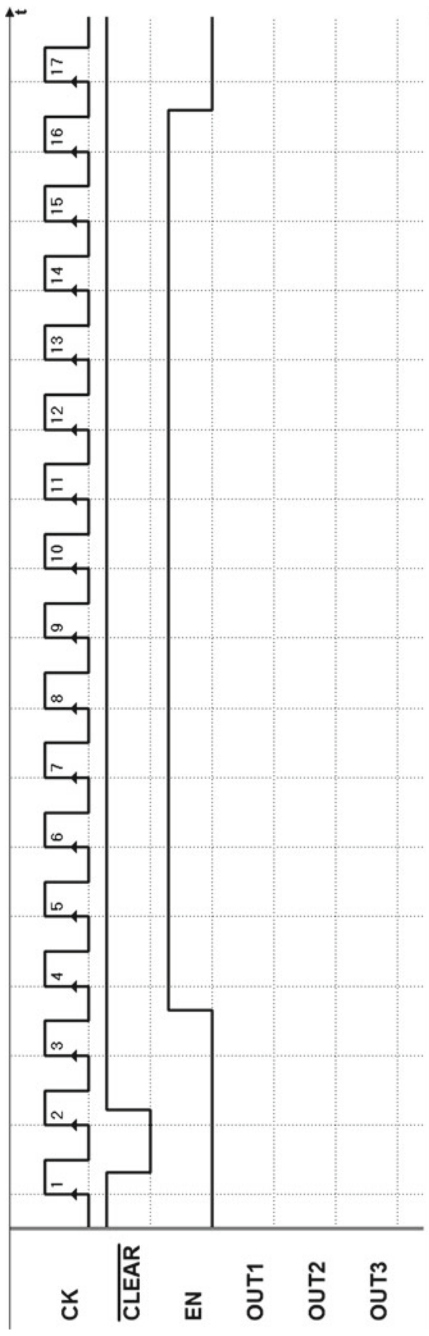


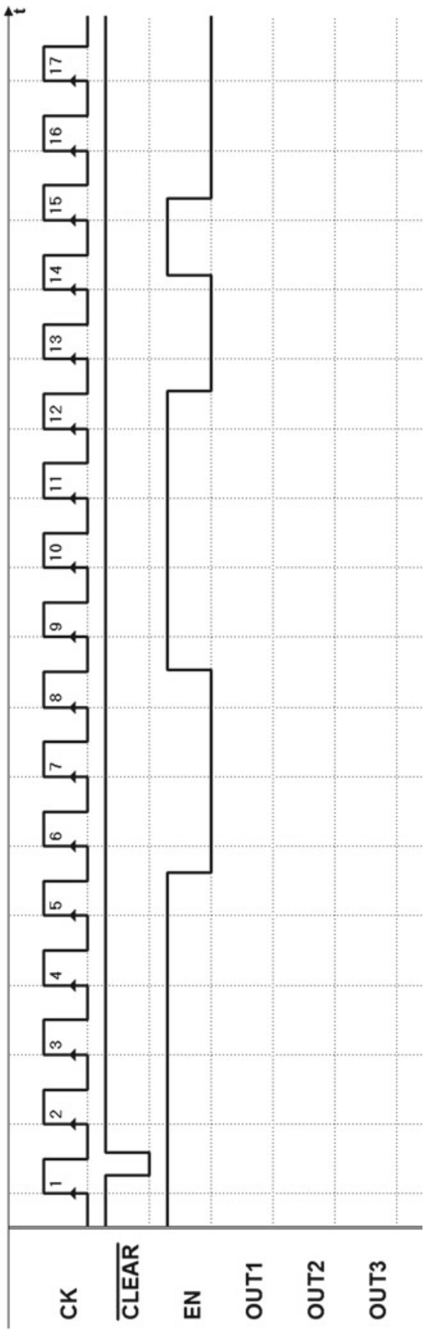

t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
CK
CLEAR
EN
OUT1
OUT2
OUT3


t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
CK
CLEAR
EN
OUT1
OUT2
OUT3

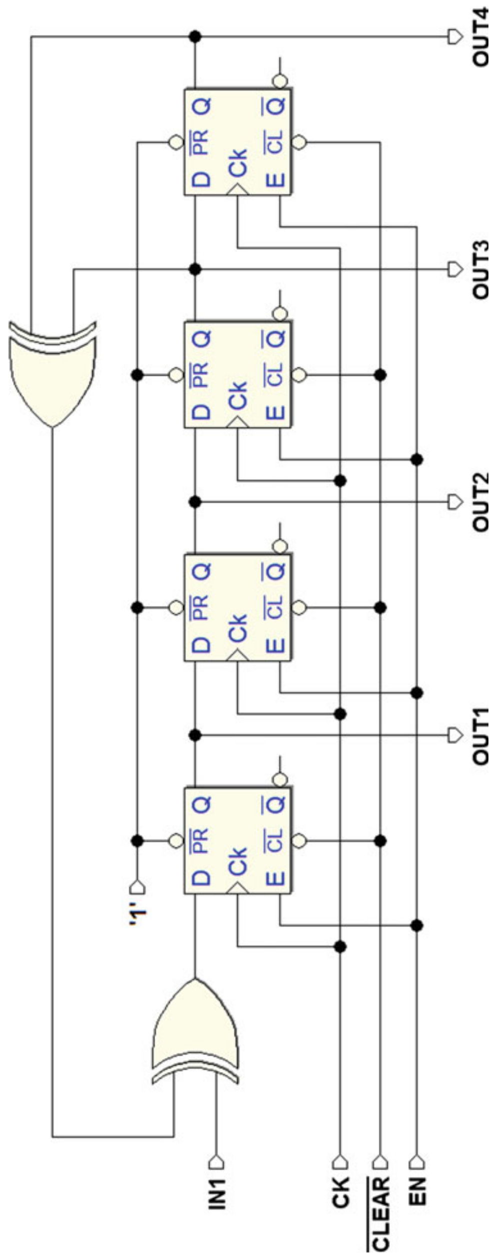
2. Exercise 2—(timing diagrams in the next page)

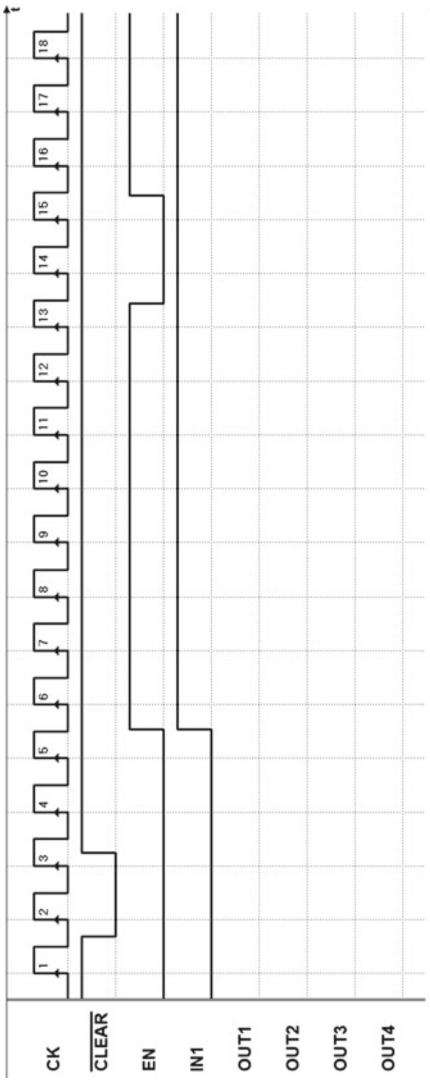


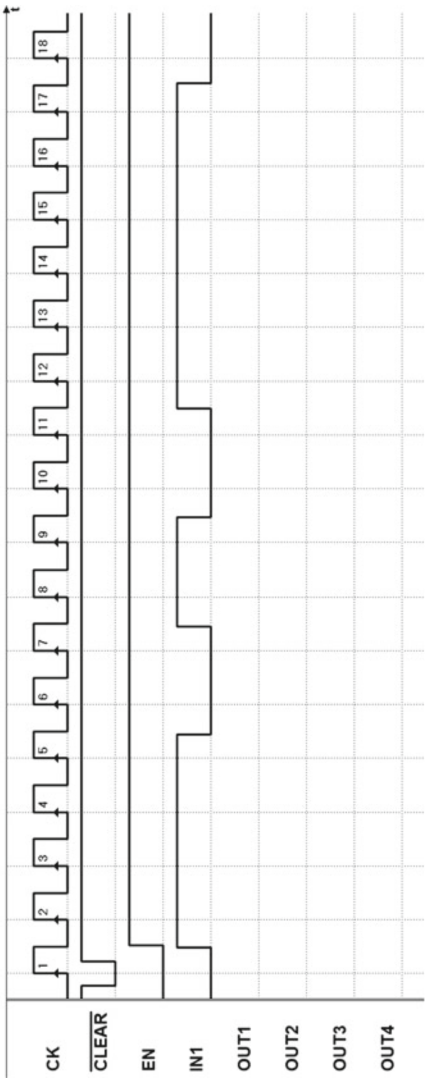

CK
CLEAR
EN
OUT1
OUT2
OUT3
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17


CK
CLEAR
EN
OUT1
OUT2
OUT3
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

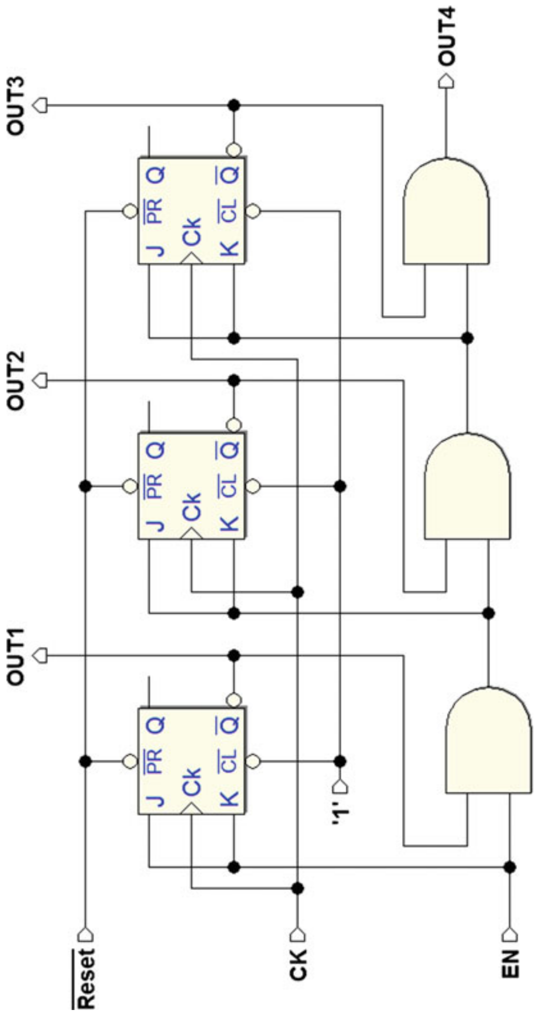
3. Exercise 3—(timing diagrams in the next page)

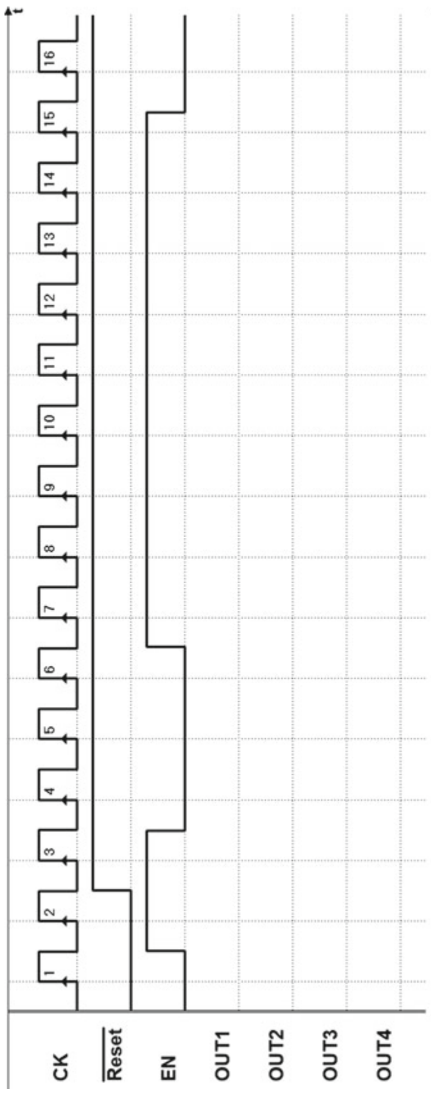


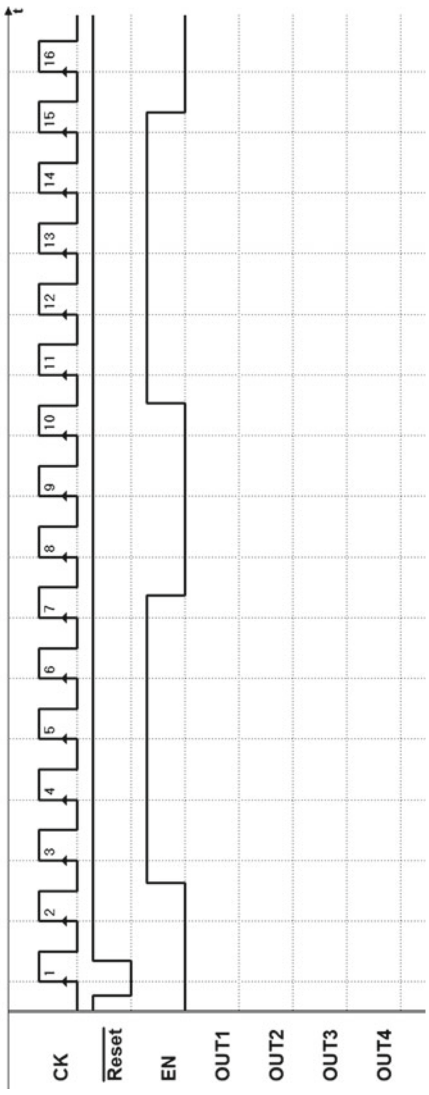

CK
CLEAR
EN
IN1
OUT1
OUT2
OUT3
OUT4
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18


CK
CLEAR
EN
IN1
OUT1
OUT2
OUT3
OUT4
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

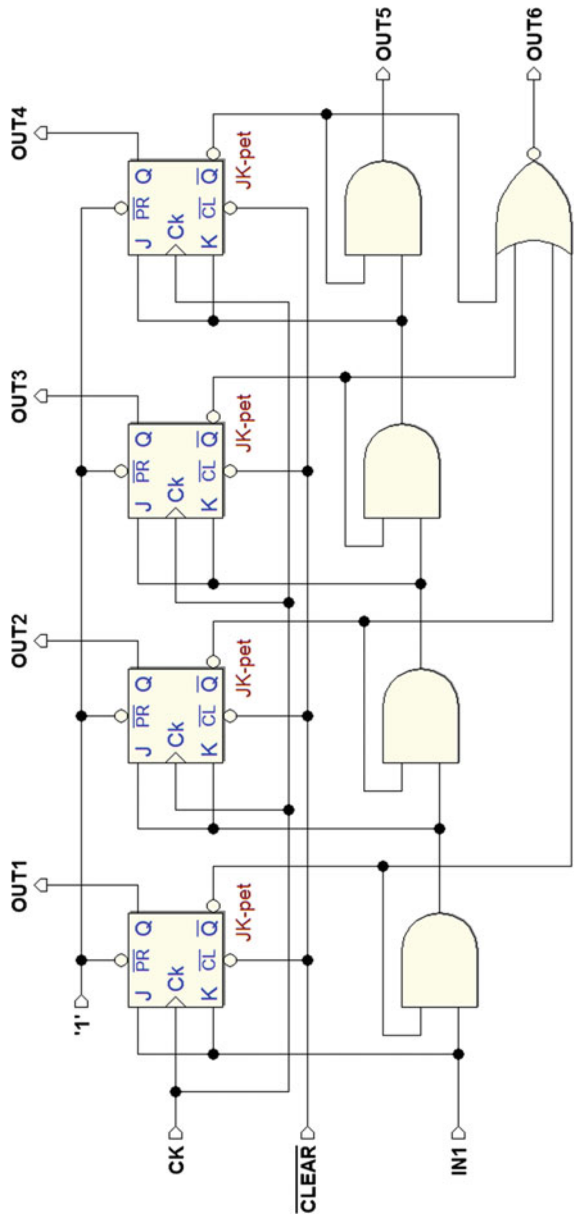
4. Exercise 4—(timing diagrams in the next page)

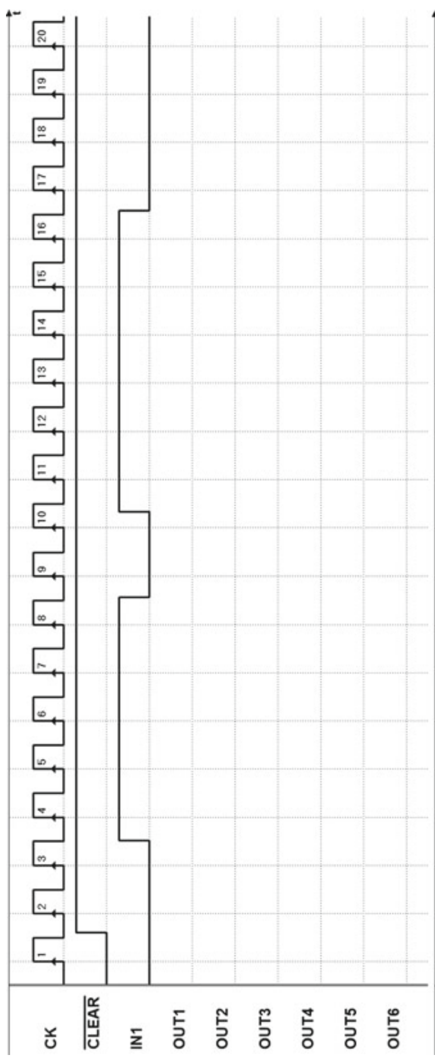


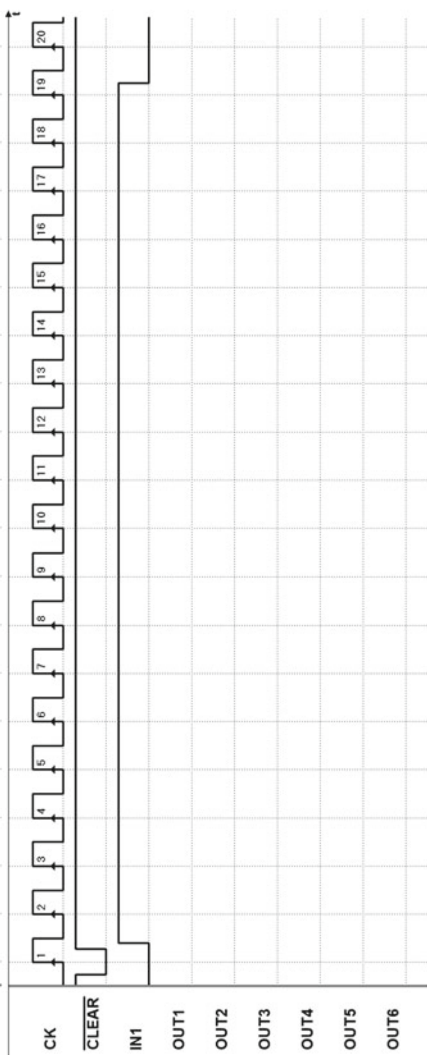

t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
CK
$\overline{\text{Reset}}$
EN
OUT1
OUT2
OUT3
OUT4


t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
CK
$\overline{\text{Reset}}$
EN
OUT1
OUT2
OUT3
OUT4

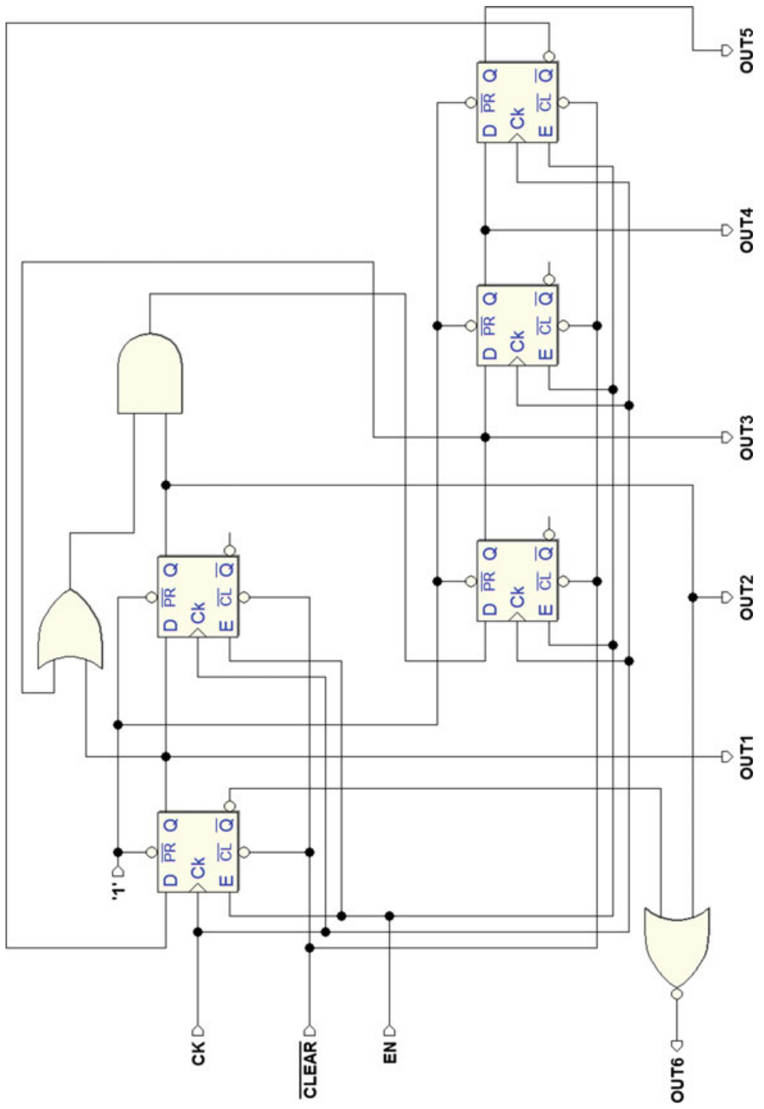
5. Exercise 5—(timing diagrams in the next page)

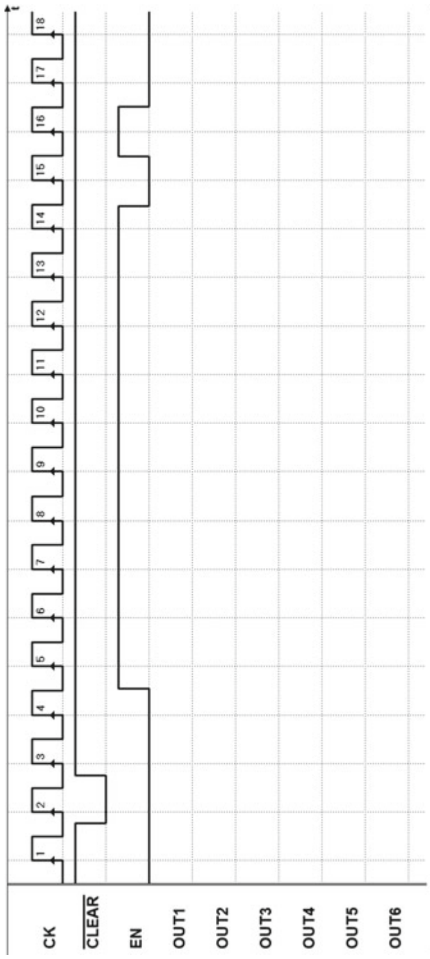


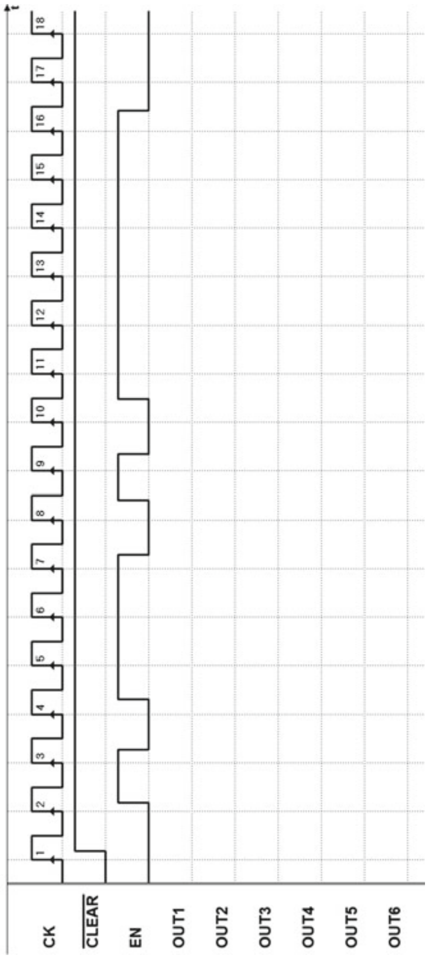

t
CK
CLEAR
IN1
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20


t
CK
CLEAR
IN1
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

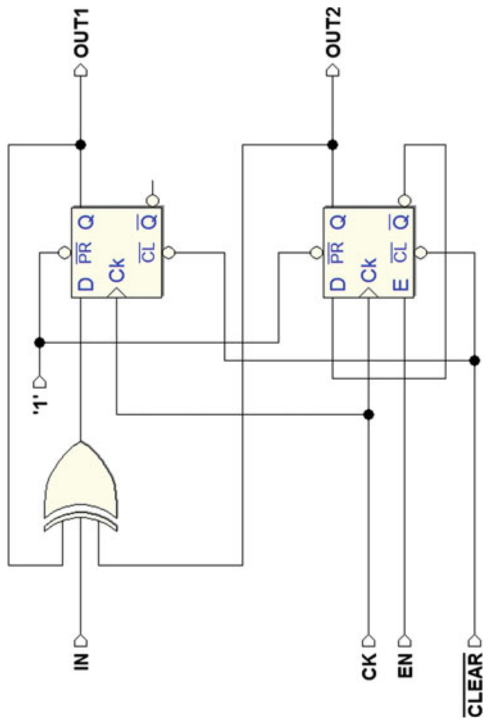
6. Exercise 6—(timing diagrams in the next page)

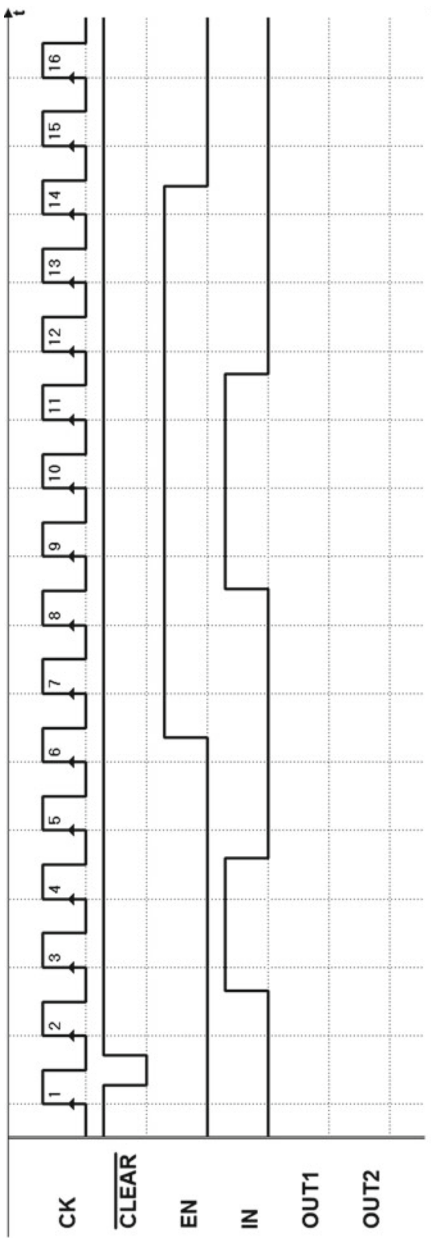


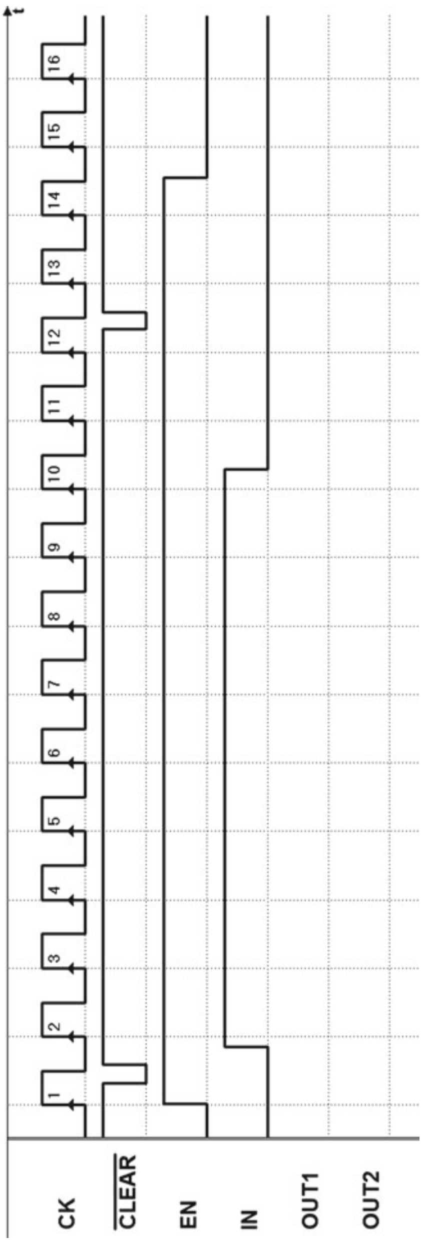

CK
CLEAR
EN
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18


CK
CLEAR
EN
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

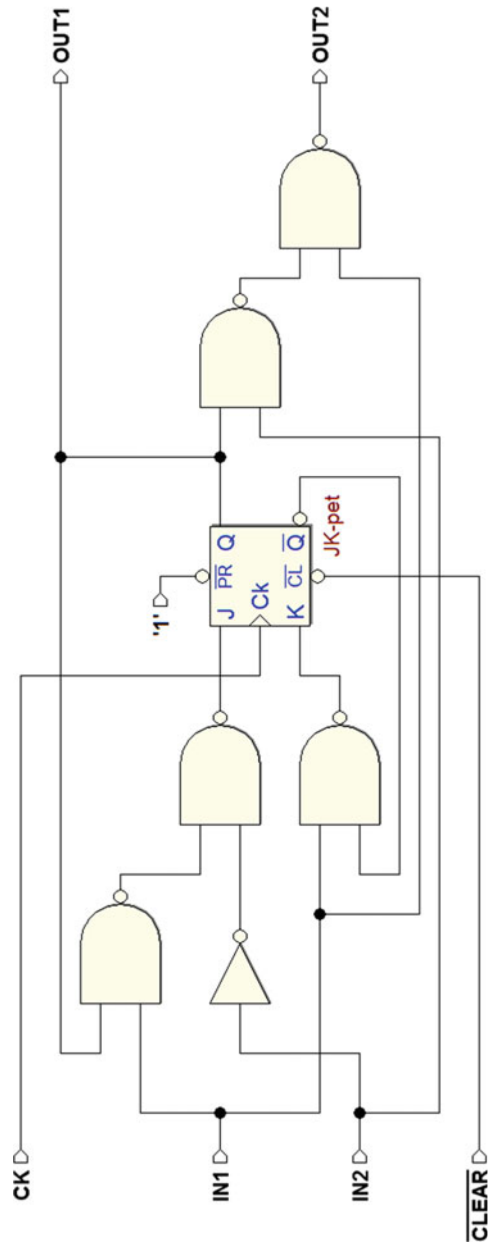
7. Exercise 7—(timing diagrams in the next page)

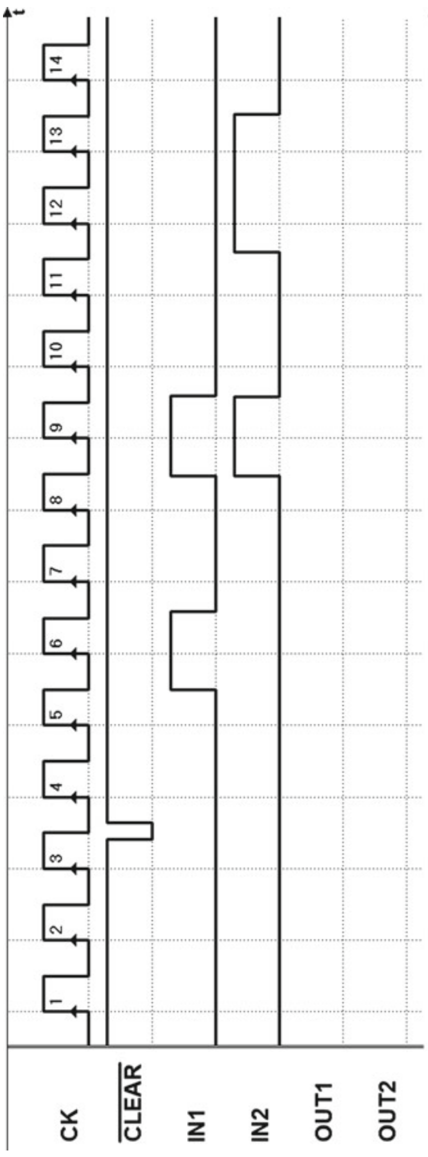


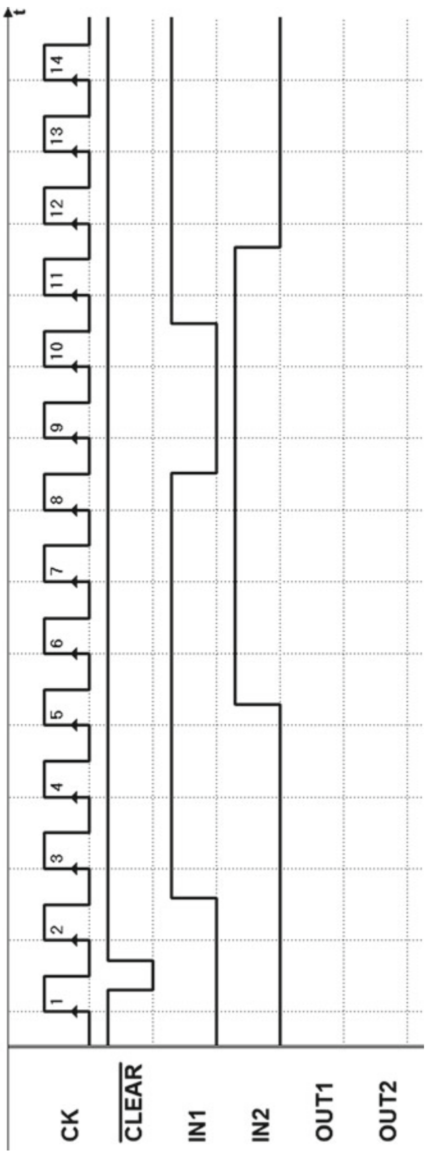

t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
CK
CLEAR
EN
IN
OUT1
OUT2


t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
CK
CLEAR
EN
IN
OUT1
OUT2

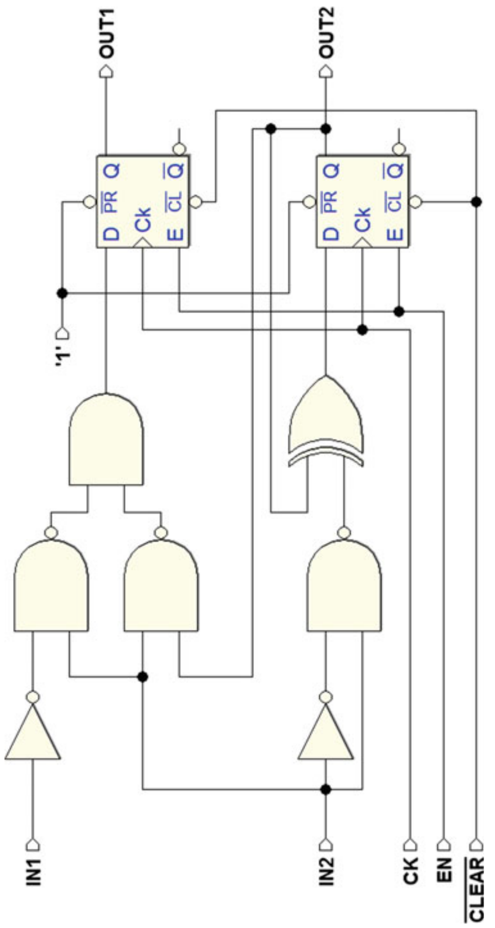
8. Exercise 8—(timing diagrams in the next page)

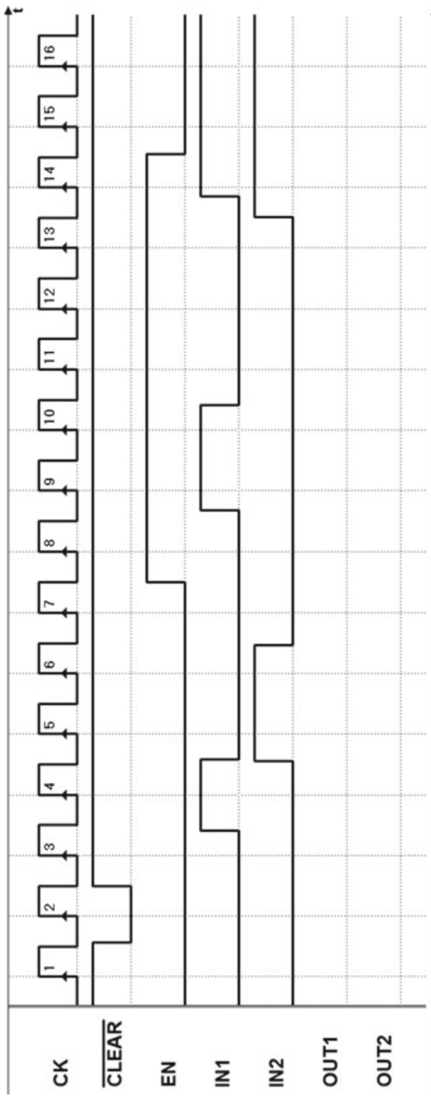


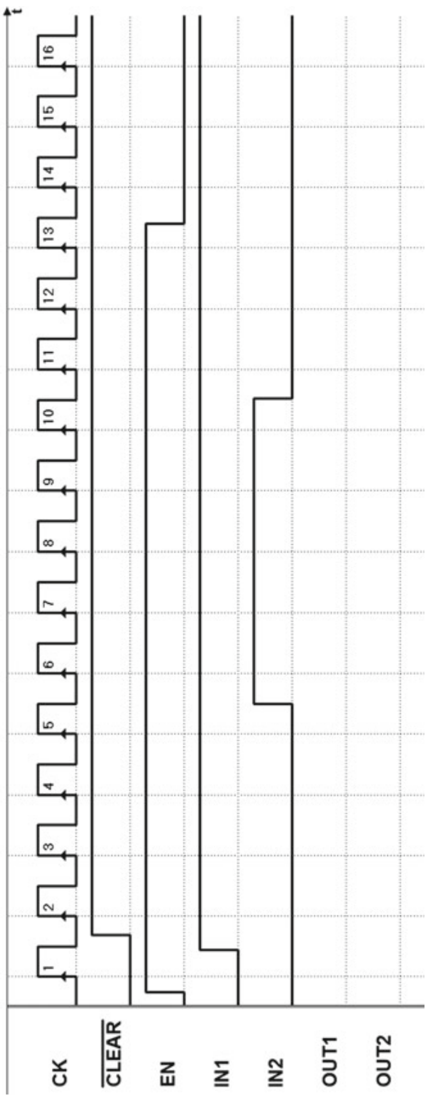

t
CK
$\overline{\text{CLEAR}}$
IN1
IN2
OUT1
OUT2
1
2
3
4
5
6
7
8
9
10
11
12
13
14


t
CK
$\overline{\text{CLEAR}}$
IN1
IN2
OUT1
OUT2
1
2
3
4
5
6
7
8
9
10
11
12
13
14

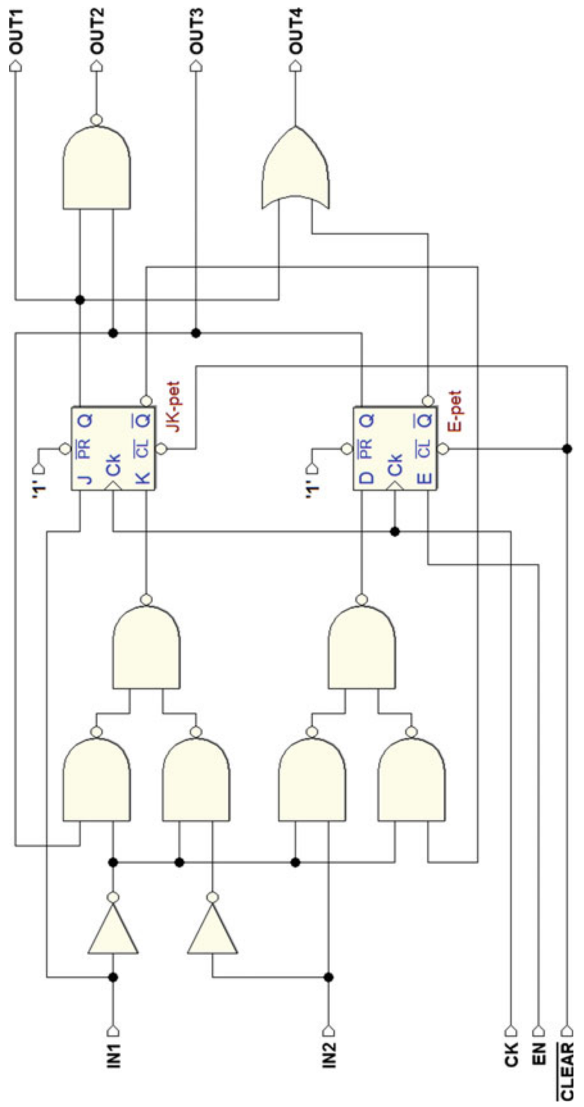
9. Exercise 9—(timing diagrams in the next page)

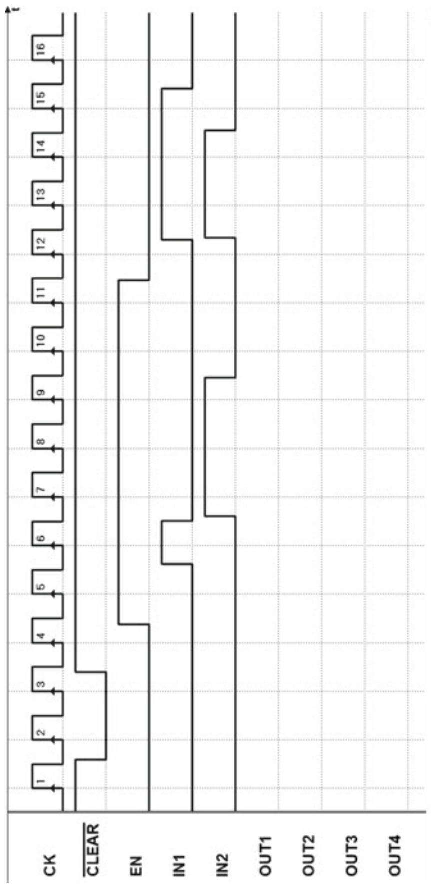


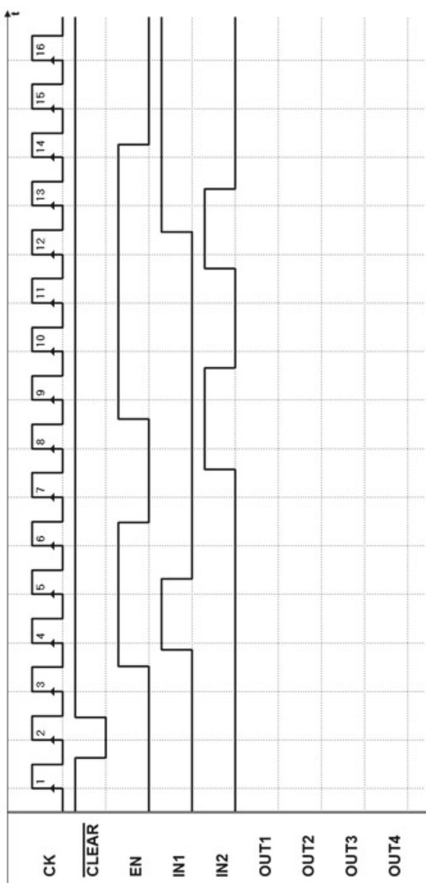

t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
CK
CLEAR
EN
IN1
IN2
OUT1
OUT2


t
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
CK
CLEAR
EN
IN1
IN2
OUT1
OUT2

10. Exercises 10—(timing diagrams in the next page)



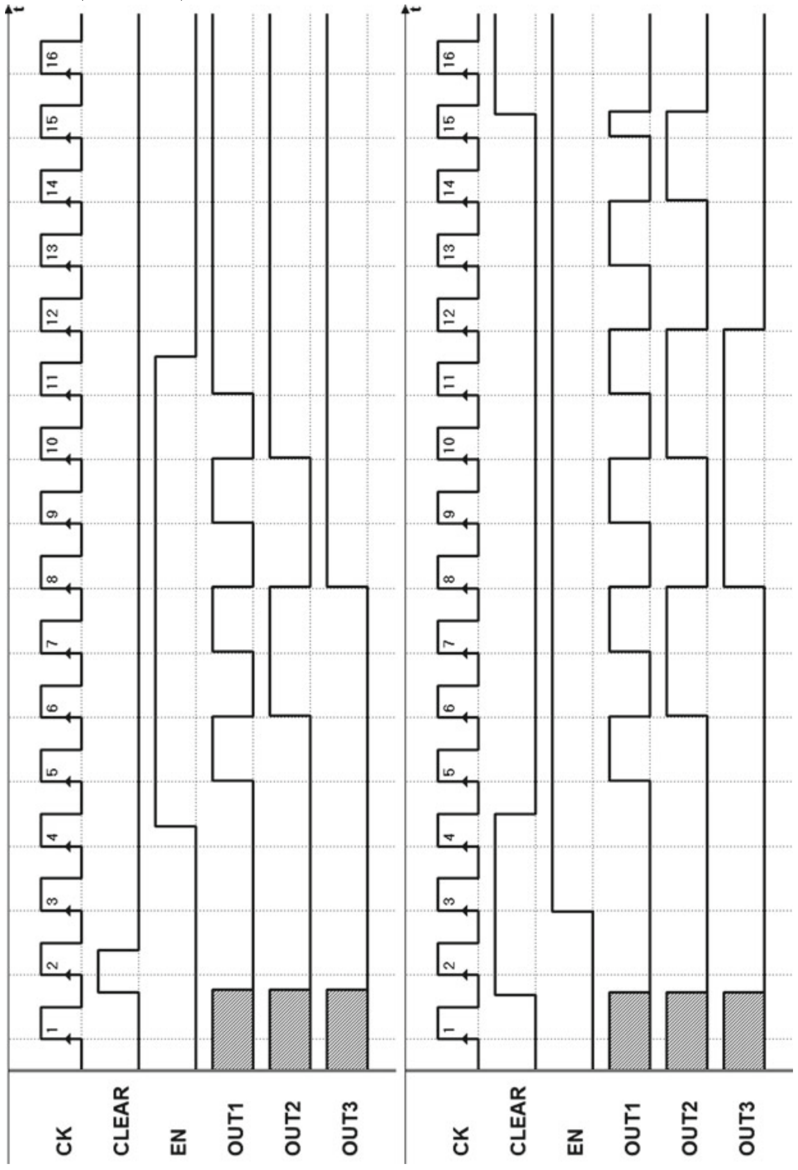

t
CK
CLEAR
EN
IN1
IN2
OUT1
OUT2
OUT3
OUT4
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16


t
CK
CLEAR
EN
IN1
IN2
OUT1
OUT2
OUT3
OUT4
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

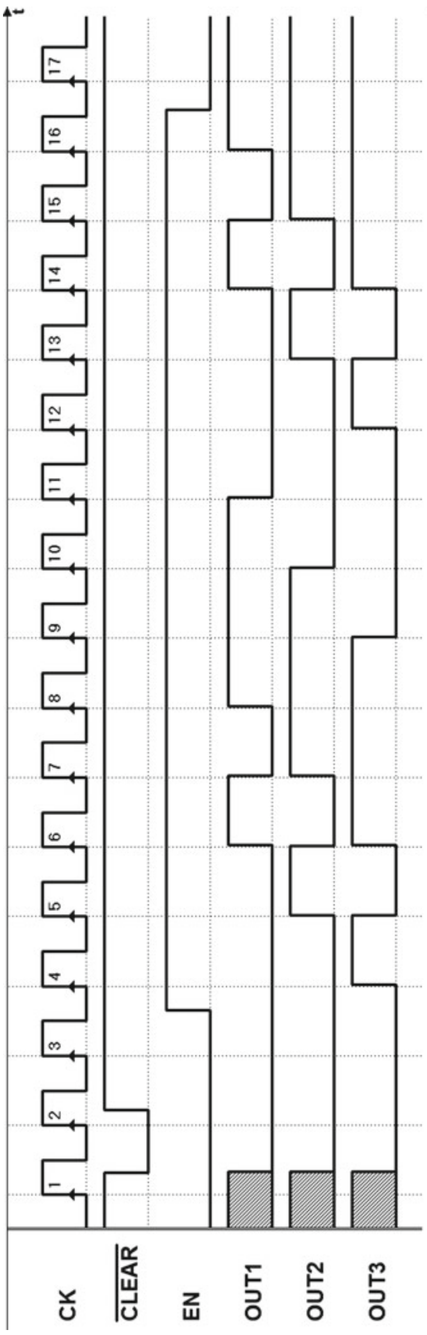
6.6 Solutions

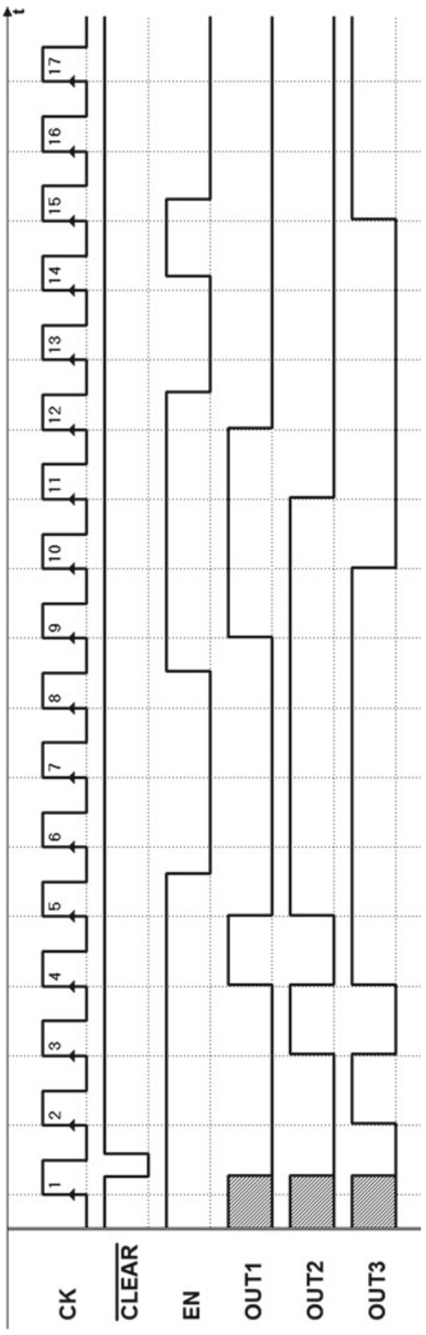
The timing diagrams reported here were obtained through the *Deeds* timing simulator. The files of the networks assigned here are available on the *Deeds* site and on the *digital contents* pages of the book, so the solutions can be checked on the simulator as well.

1. Exercise 1 (solutions)

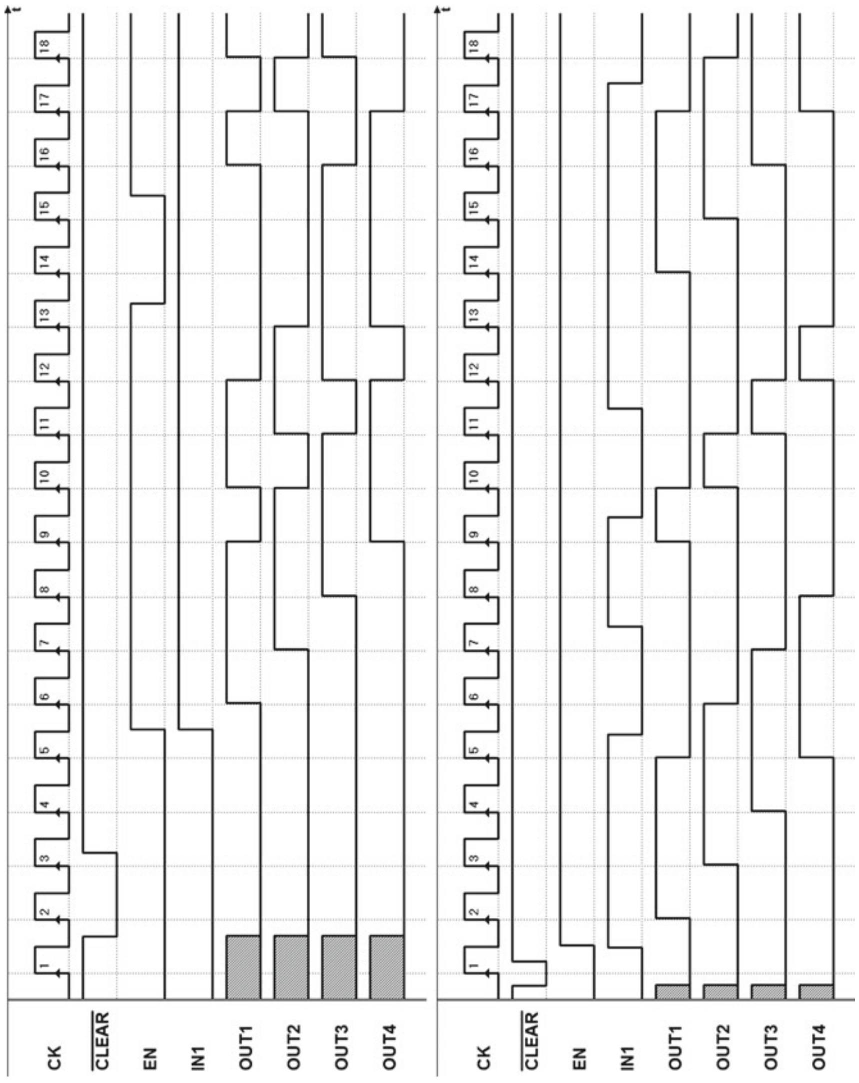


2. Exercise 2 (solutions)

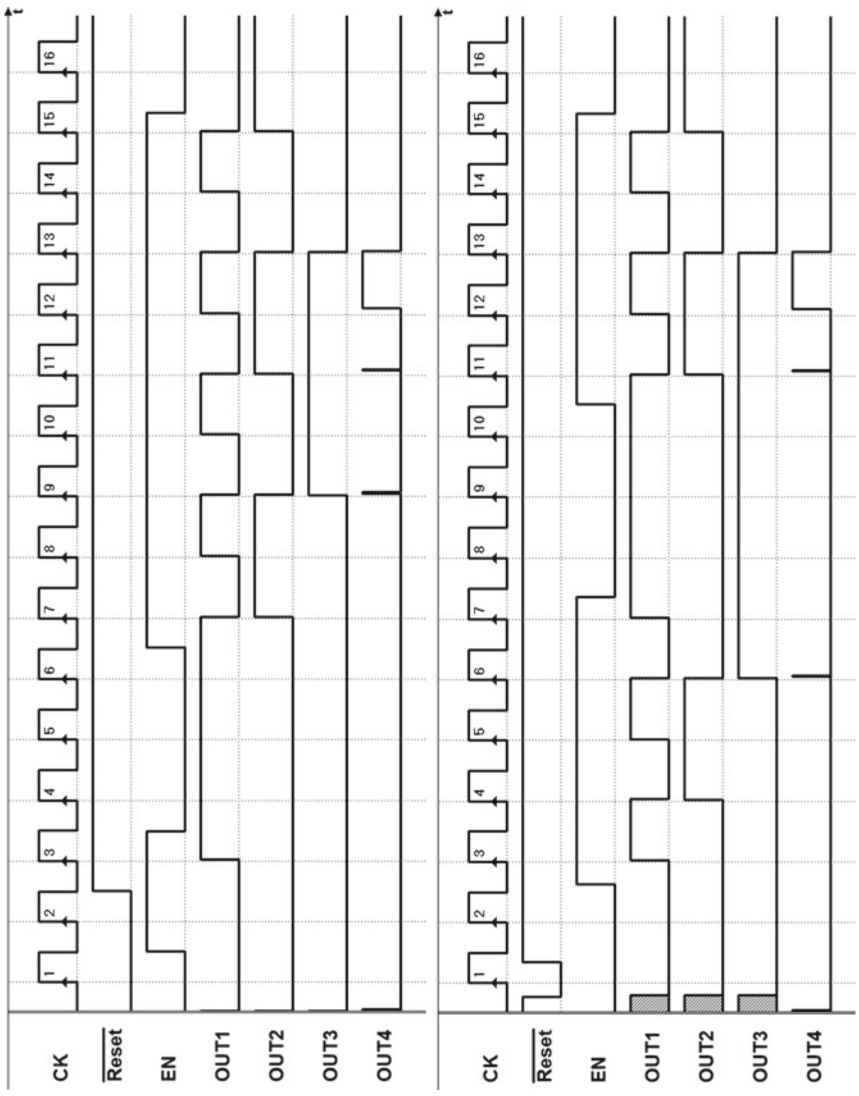




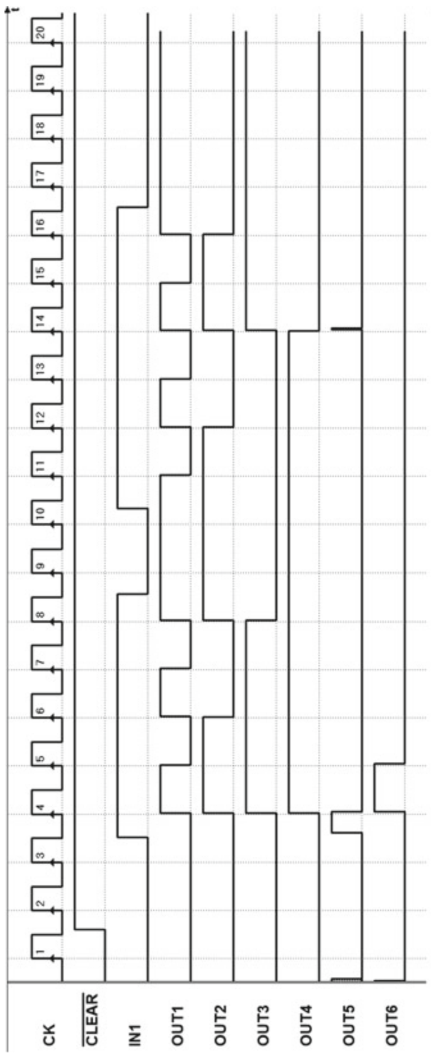
3. Exercise 3 (solutions)

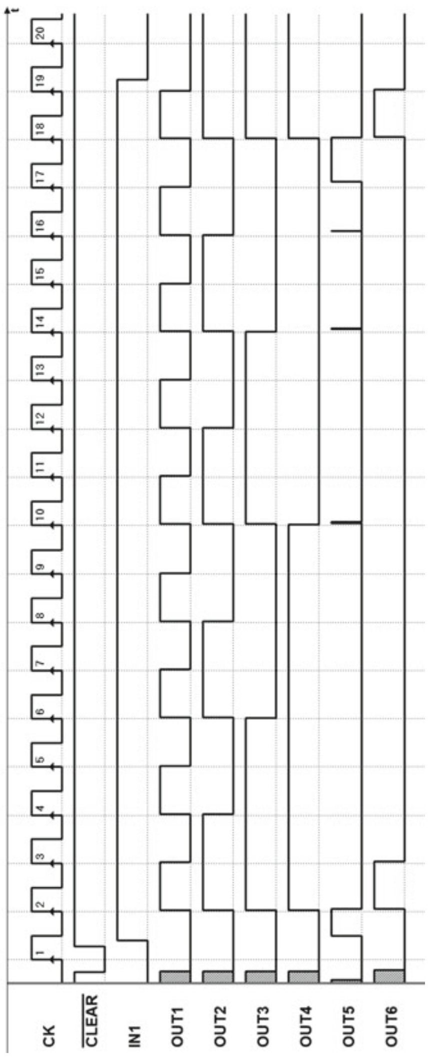


4. Exercise 4 (solutions)

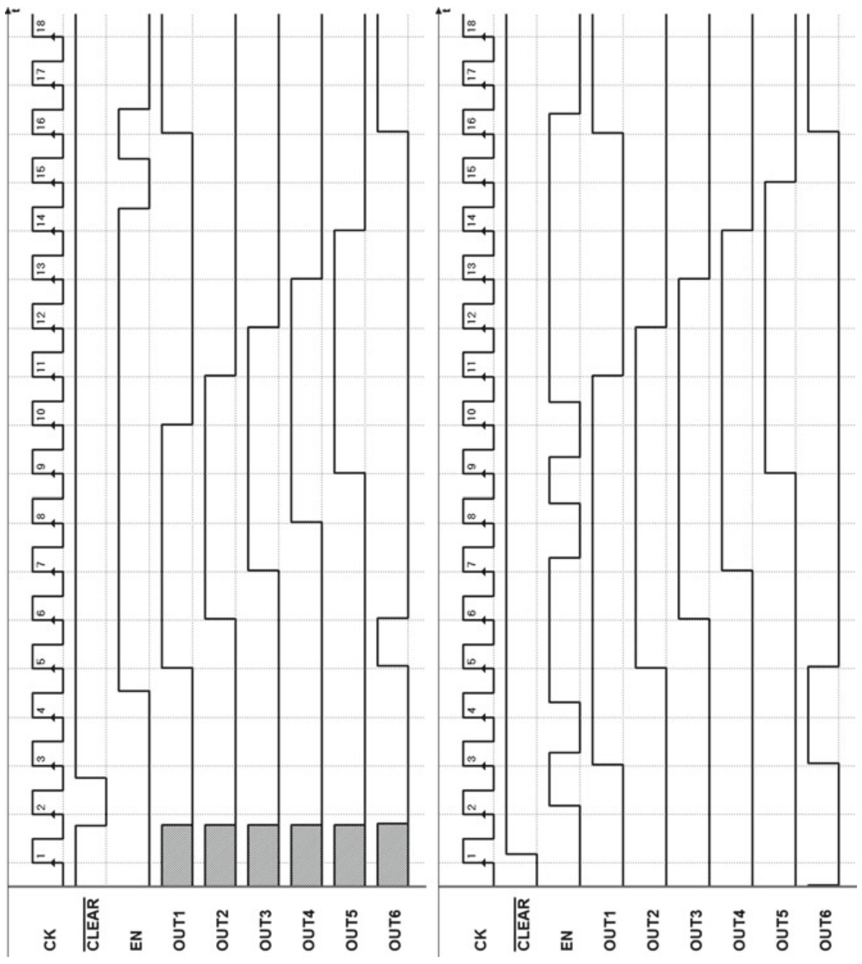


5. Exercise 5 (solutions)

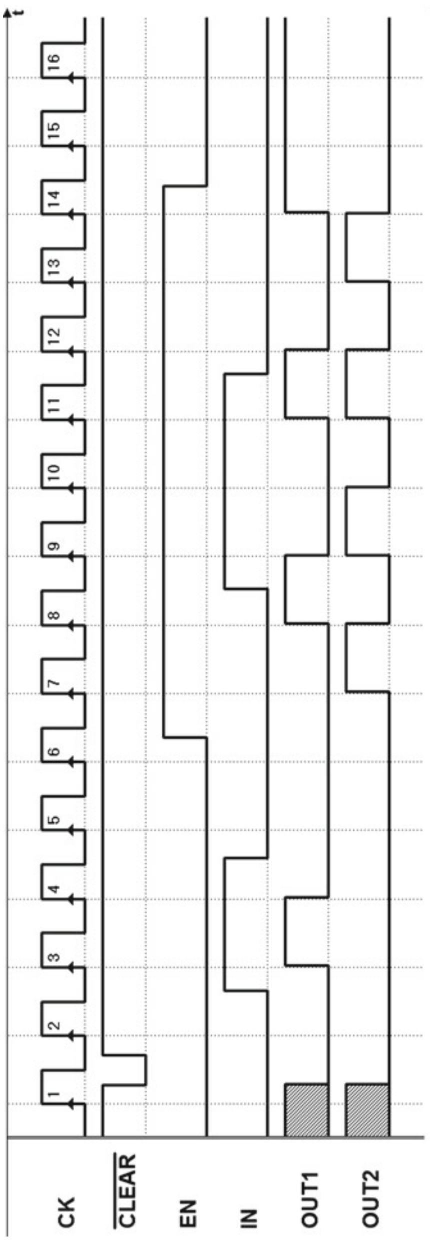


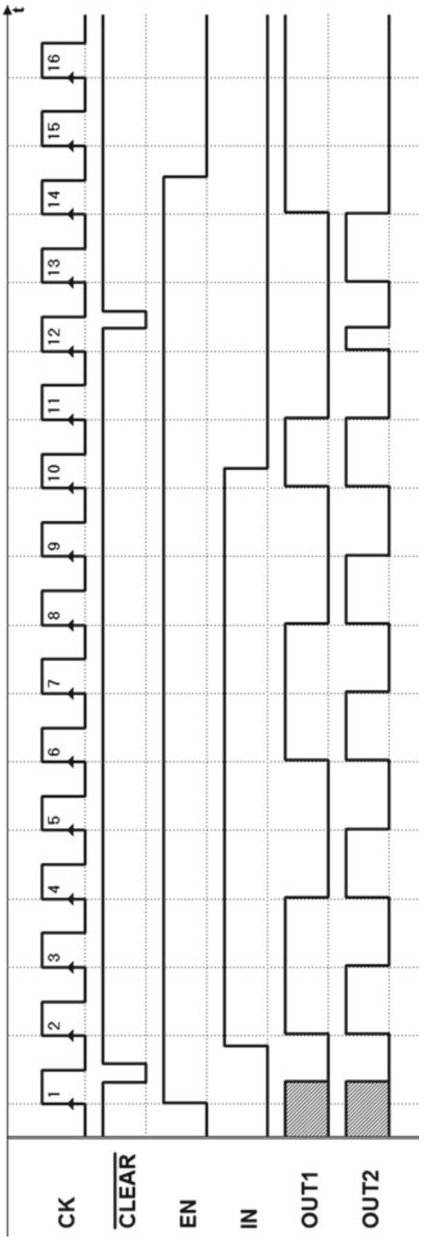


6. Exercise 6 (solutions)

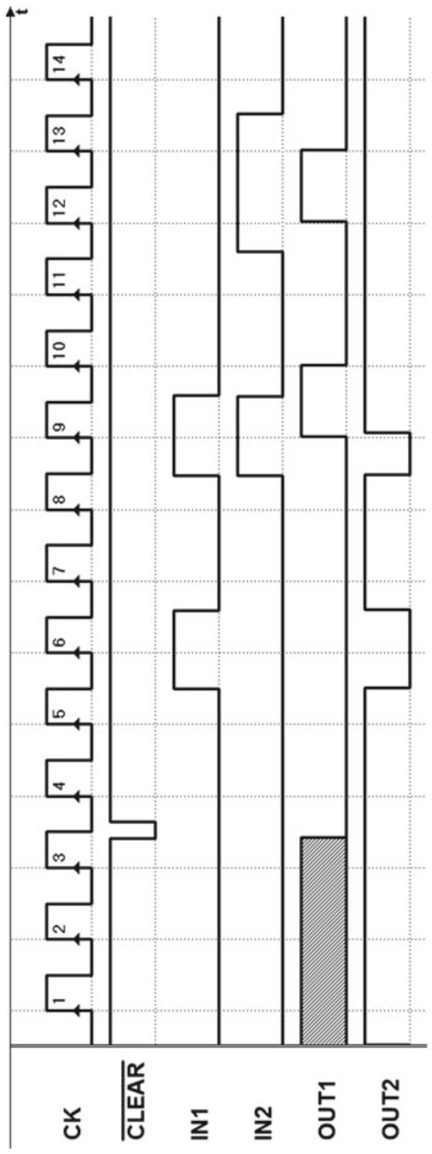


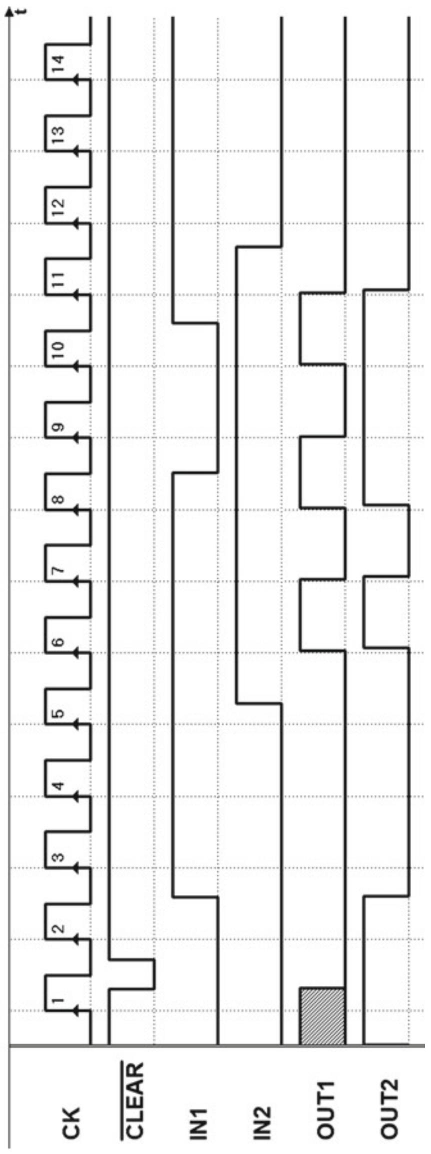
7. Exercise 7 (solutions)



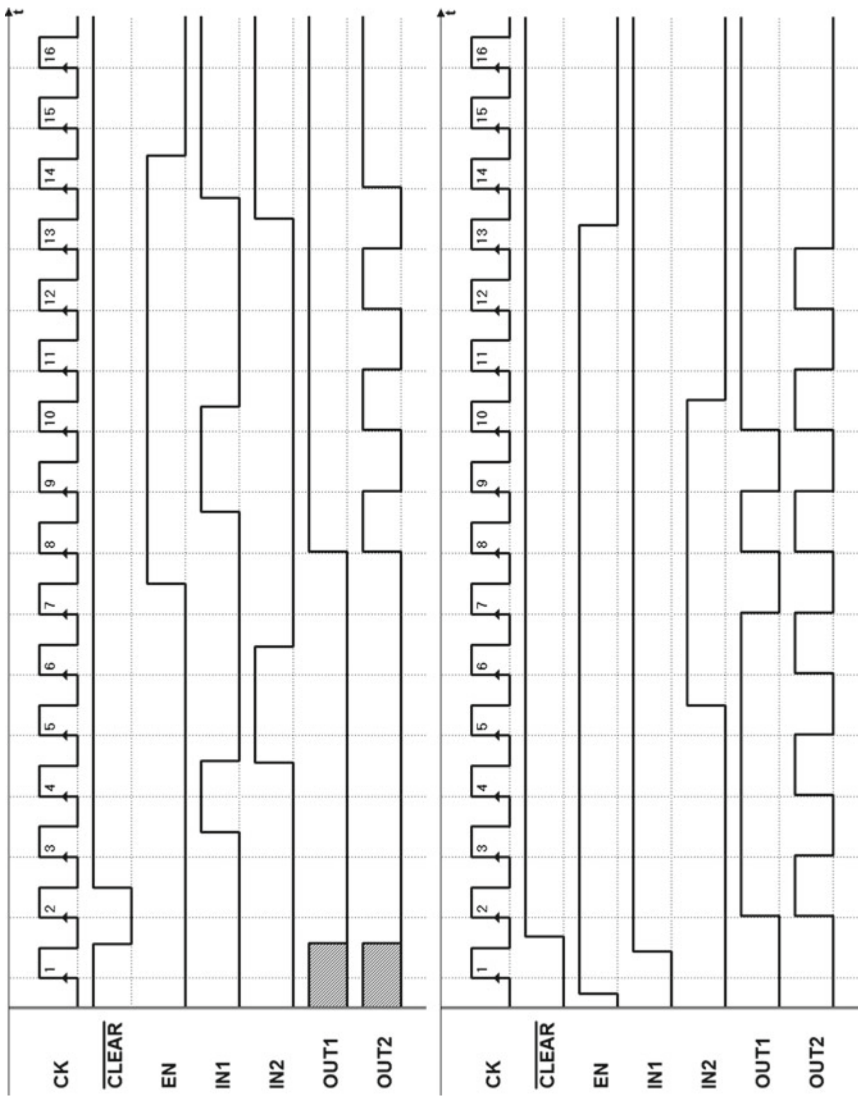


8. Exercise 8 (solutions)

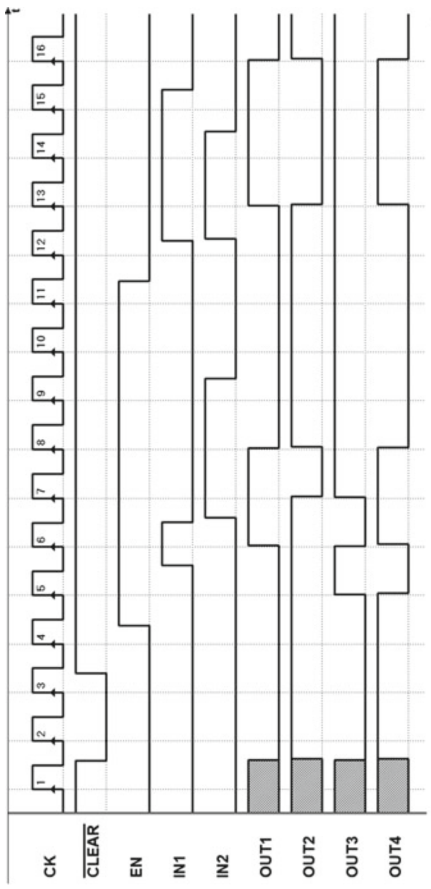


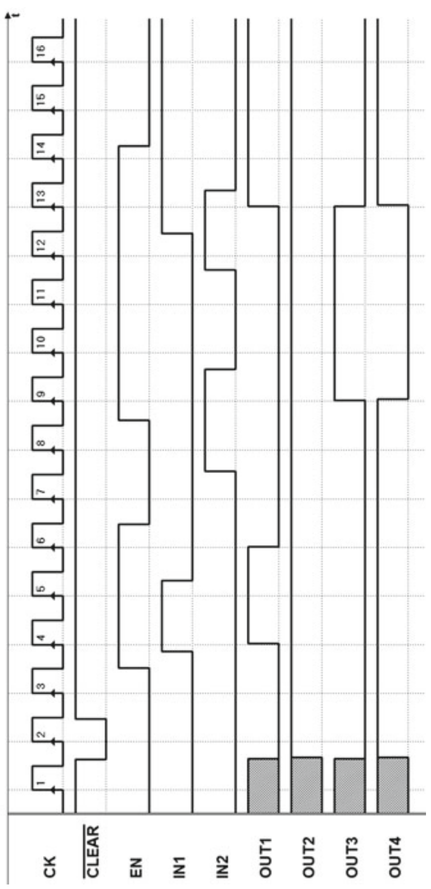


9. Exercise 9 (solutions)



10. Exercise 10 (solutions)


t
CK
CLEAR
EN
IN1
IN2
OUT1
OUT2
OUT3
OUT4
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16


t
CK
CLEAR
EN
IN1
IN2
OUT1
OUT2
OUT3
OUT4
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16