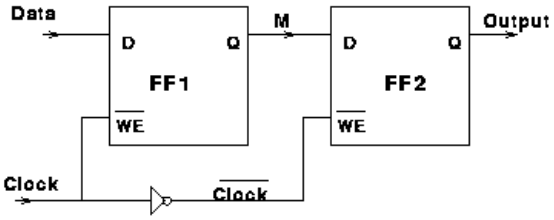
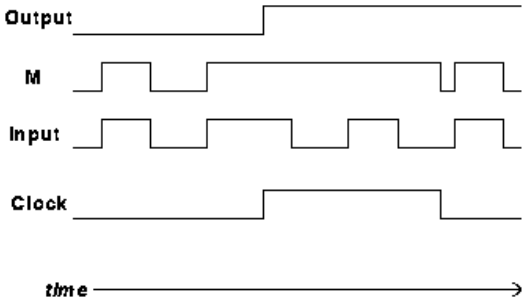
Edge-Triggered D-type Flip-flop

The transparent D-type flip-flop is written during the **period of time** that the write control is active. However there is a demand in many circuits for a storage device (*flip-flop* or *latch* - these terms are usually interchangeable), in which the writing of a value occurs at an **instance in time**. Such a device can be built in a number of ways, one of which uses 2 transparent D-type flip-flops. This form will be presented here and the circuit is shown in the figure.





The 2 transparent flip-flops, *FF1* & *FF2*, are linked in series with the output of *FF1* being fed into the *D* input of *FF2*. The flip-flops operate in a complementary fashion such that when *FF1* is being written, *FF2* is in storage mode, and vice versa. This is achieved by *FF1* being fed with the input signal *Clock*, while *FF2* is fed with its inverse. The circuit is called an *edge-triggered D-type flip-flop*, as the value on the *D* input of *FF1* (the circuit's data input) is stored in the circuit, and output on the Q of FF2, on the 0→1 transition of Clock. This transition is called the *rising* edge, sometimes represented on a circuit diagram by the symbol ↑. The timing diagram shows the response of the circuit to example input signals. It is assumed in the diagram that the output is 0 at the beginning. This diagram should help in understanding the circuit operation. The circuit operates in the following way:-

Clock at 0:

- *FF1* is enabled and is written with the value on its *D* input. Any change on *D* changes the stored value and the output value on its *Q* output. See trace *M* in the timing diagram.
- *FF2* is in storage mode, and outputs the value stored when last enabled (when *Clock* was 1).

Clock 0→1 & Clock at 1:

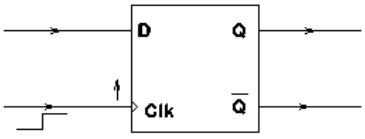
- *FF1* goes into storage mode, storing the value on *D* at the time of the transition. This value is output to *FF2* which goes into writing mode to store and output this value. *M* no longer changes as the *Input* changes, so that the value of FF2 remains constant.

Clock 1→0:

- *FF2* goes into storage mode holding the value it has been receiving from FF1 since the *0→1* edge of *Clock*. *FF1* goes into writing mode, storing the current value on its D input. *M* again follows the *Input*, while the *Output* holds the value stored at the *Clock* rising edge.

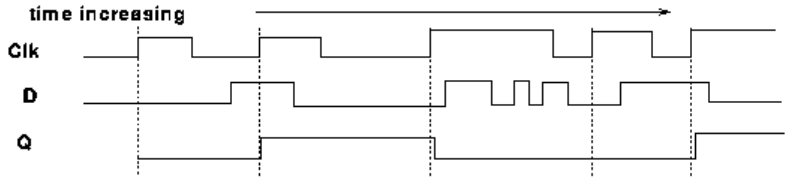
It can be seen that the output from the circuit (*Q* of *FF2*) only changes on the *0→1* edge of *Clock*, and that the output value is always the value on the circuits data input at the time of this edge. Thus, the circuit stores the input value at the instance that the edge occurs. There are more technical details in the box.

In practice, there is a short time period over which storage occurs, but this is not a problem since this period can be made much shorter than the time for changes in the stored value to propagate through the circuit. However, if the value on *D* is changing at the time of the edge, it is not possible to determine the stored value. The propagation delay through the inverter on the *Clock* signal must be shorter than propagation times through the flip-flops, otherwise the circuit has problems at the *1→0* transition.



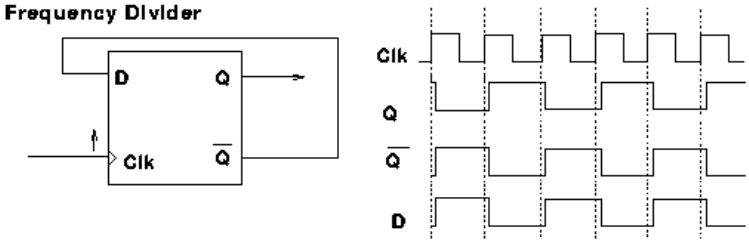
The symbol for an edge-triggered D-type flip-flip is shown to the left. Note the triangle on the Clock input to denote an edge-triggered device. A timing diagram with more clock edges is shown on the following page.

The vertical dotted lines mark the time of the edge. The value on *D* at this time appears on *Q* a short time later due to propagation delays through the circuit.



A Simple Example: the Frequency Divider

Edge-triggered devices are extremely useful and can be used for operations where a single transparent D-type flip-flop would fail. A simple circuit with 1 edge-triggered flip-flop is shown in the figure with



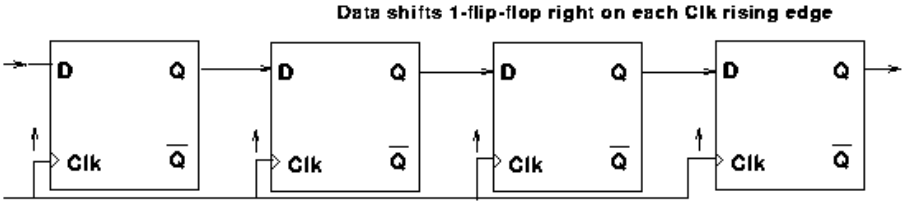
an example timing diagram to the right.

The inverse of the stored value is fed back into the device input, so that on each rising edge of *Clk* the stored value changes. In the timing diagram it can be seen that the value on *D* is the same as on the inverse output of the device. At the rising edge of *Clk*, the flip-flop takes on the value of *D* and *Q* takes on this value after a short time delay due to propagation delays through the flip-flop. At the time that *Q* changes the inverse output and *D* change. It can be seen that the frequency of the output on *Q* is half the frequency of the input on *Clk*: the circuit is a clock divider. Long chains of these circuits can be put together to reduce a clock frequency.

It should be noted that the edge-triggered device cannot be replaced by a single transparent D-type flip-flop, because on the input clock going low, the flip-flop would be enabled for writing while there is a low on the input and during this period the output would oscillate as a change on the output feeds back to change the stored value which changes the feedback value, and so on. A device that stores just on the edge of the input is required.

Delay/Shift Circuit

Another useful circuit which requires edge-triggered devices is shown to the right.



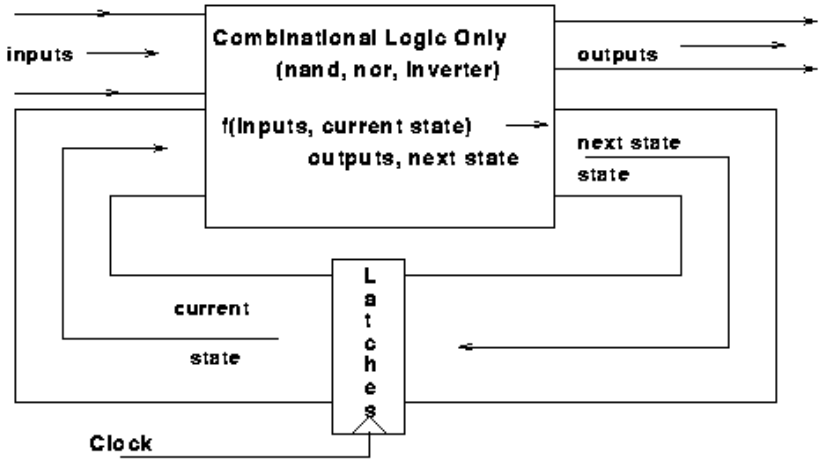
On the rising edge of *Clk*, each flip-flop stores the output of the flip-flop immediately to its left, while the leftmost flip-flop stores the input value. This circuit can be used immediately as a delay circuit: the input value appears on the rightmost output after a delay of 4 rising edges. It could also form the basis of a shift register, since the circuit shifts data values; extra logic would be needed in this case. A number of different types of shift register are used in computer systems:-

- *serial-in/parallel-out shift register*: in terms of the figure above, 4 bits would be shifted in from the input into different flip-flops using 4 rising edges on *Clk* and then all 4-bits would be read out in parallel as a single 4-bit data item.
- *parallel-in/serial-out shift register*: in terms of the figure above, each flip-flop would be loaded with a different bit from a 4-bit data value, and the device would be *clocked* to shift each bit out of the end *serially* one bit at a time.
- *parallel-in/parallel-out shift register*: these enable a data value to be shifted one or more binary places. A data item would be loaded in parallel, shifted and the modified value read out in parallel. Bits shifted out of one end can be shifted in at the other or new bits can be shifted in.

Finite State Machines

Edge-triggered flip-flops play a key role in a very important digital circuit, the *Finite State Machine*. Finite State Machines (FSMs) are important because they allow for a sequence of operations to be performed with a controllable interval for each operation, and for a choice to be made of the next operation to be performed under the control of input signals. Thus FSMs allow control circuits of great complexity to be built.

The digital FSM is a circuit with feedback signals, and as in many circuits where there are feedback signals, the outputs of the circuit are a function not only of the circuit's inputs, but also of the *internal state* of the circuit. The state of an FSM is easily identifiable as the information that forms the *state* of the FSM is stored in edge-triggered flip-flops. A generalised circuit for a FSM is shown below.

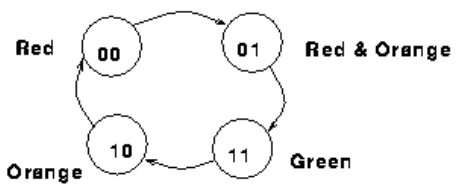


The circuit has 2 major components a set of edge-triggered flip-flops, labelled *latches* in the figure, and a block of combinational logic (there is no feedback in this block). This logic takes the input signals and the output values of the latches and generates a set of output values from the circuit and a set of new inputs to the latches. The outputs from the latches (their stored values) are called the *current state* of the circuit, while the inputs to the latches carry the *next state* of the circuit: the next state becomes the current state when the latches are written by the rising edge on the *Clock* input.

Both the output values and the next state values are functions of the inputs and the current state. The maximum number of states of the circuit (not all may be used or *reached*) is $2^{\text{no of latches}}$, i.e. with 8 latches there are 256 states, with 20 latches, 1M states. Since the next state is defined by the combinational logic, any state, i.e. a particular set of 1s and 0s in the latches, can be reached from any other state. Thus, in an 8 latch FSM, state 10100000 could be followed by 00011000 or by 11101000, depending on the design. It is possible to make the next state dependent on one or more input signals: state 10100000 might be followed by state 00011000 if input *I* is 1, but by state 11101000, if *I* is 0. The maximum number of choices from a state is $2^{\text{no of inputs}}$, i.e. with no inputs, there is only one possible *successor* state, with one input, two possible *successors*, with 2 inputs 4, etc. Of course the number of successor states can never be greater than the maximum number of states.

From a particular state, the FSM will step through a sequence of successor states controlled by the inputs to the device and the states themselves. The sequencing is controlled by the clock. The outputs from the FSM are controlled by the inputs and the state sequence.

A very simple FSM can be designed to produce the basic traffic light controller that goes through the light sequence:
Red - Red & Orange - Green - Orange - Red



The figure to the right shows the state diagram of the traffic light controller. The states of the controller are the circles; the transitions between states are marked by the arrows. It can be seen that there is only one successor state for each state: there is no choice of successor in this FSM.

This lack of choice means that no inputs are required. To turn this *state diagram* into a digital circuit requires a number of stages:-

State Assignment

Unique binary patterns, or labels, must be allocated to each state. The number of bits required is log_2*(number of states).* Thus, 2 bits are needed for 4 states, 3 bits for 5,6,7 and 8 states, etc. The pattern allocation can be purely arbitrary, but some allocations give simpler solutions. The number of state bits determine the number of latches in the FSM. A set of state assignments for the traffic light controller is shown on the state diagram above.

Generation of Boolean functions for combination logic

The functions mapping the inputs and current state to the next state and the output must be produced. These are generated from the state diagram and the state assignments. One way to do this is to build truth tables and then use Karnaugh maps to produce a function for each output from the combinational logic.

For the traffic light controller, the truth table is shown in the figure.

Here, the current state bits are named *cs1 & cs0*, while the next state bits are *ns1 & ns0*.
The outputs to control the lights are named *Red, Orange & Green*.

← INPUTS →	← OUTPUTS →			
Current State (cs1, cs0)	Next State (ns1, ns0)	Red	Orange	Green
00	01	1	0	0
01	11	1	1	0
11	10	0	0	1
10	00	0	1	0

Taking the first line of the table, the current state is 00(*RED)*, and only the *Red* output is active turning on just the red light, while the next state bits show that the successor state will be state 01(*Red & Orange*). The various outputs only depend on *cs1 & cs0*.

The Karnaugh map for *ns1* is to the right, along with the Boolean function derived from it. It can easily be seen that *ns1* is just *cs0*: the least significant bit of the state assignment becomes the most significant bit on the next *state transition.*

ns1 = cs0	cs0 = 0	cs0 = 1
cs1 = 0	0	1
cs1 = 1	0	1

The Karnaugh map for *ns0* is to the right, along with the Boolean function derived from it. It can easily be seen that *ns0* is just the inverse of *cs1*.

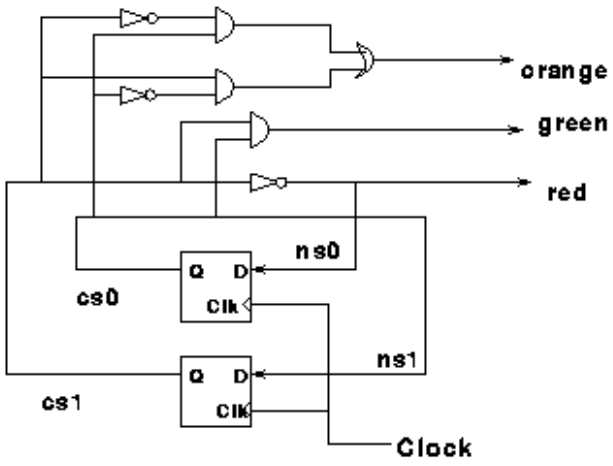
$ns0 = \overline{cs1}$	cs0 = 0	cs0 = 1
cs1 = 0	1	1
cs1 = 1	0	0

The Boolean functions can be derived in the same way to give the following:-

$$Red = \overline{cs1} \qquad Orange = \overline{cs1}.cs0 + cs1.\overline{cs0} \qquad Green = cs1.cs0$$

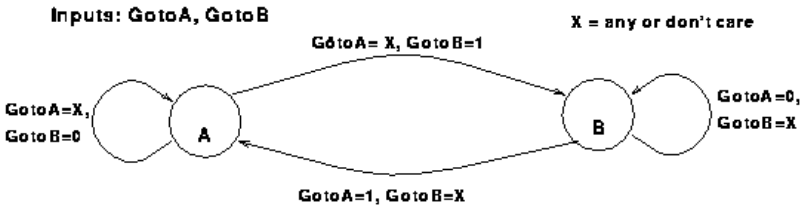
Circuit Layout:

It can be seen that *ns0* and *Red* are identical, so that they can be merged to reduce the number of outputs in the final circuit. Everything has now been done and the circuit can be drawn out. It is shown to the left.



Only one thing remains: the choice and generation of the clock frequency. The latter controls the interval between state transitions, i.e. the time spent in each state and how long a light is on.

A State Machine with Conditional Branching



To the right is a state diagram for a state machine with 2 states and 2 transitions from each state. Separate inputs control the choice of transition from each state, and the value of the input signals that cause a transition to be *taken* are marked against each transition.
Input *GotoB* determines the choice at *A*: if it is *0*, the machine stays in state *A*, if it is *1*, the state of the machine becomes *B*. Input *GotoA* has no effect while the machine in state A: this is shown by the fact that the *don't care (X)* value is shown against *GotoA* for both transitions from *A*. Input *GotoB* controls the transitions from state *B* in a similar fashion.

With 2 states, there is only need for 1 state bit. Let us label ithe state bit ***Q***, so that state *A* is assigned *Q =0*, B is *Q = 1*. After state assignment the truth table is simply derived from the state diagram as shown to the right.

current state	Current Inputs		Next state	
Q	GotoA	GotoB	next Q	
0	0	0	0	Stay In A
0	0	1	1	move to B
0	1	1	1	move to B
0	1	0	0	stay In A
1	0	0	1	stay In B
1	0	1	1	stay In B
1	1	1	0	move to A
1	1	0	0	move to A

The Karnaugh map, which shows how *next Q*, usually denoted *Q'*, is dependent on *Q, GotoA* and *GotoB*. It is trivially derived and not shown, but from it the Boolean function for *Q'* can be found:-

$$Q' = \overline{Q}.GotoB + Q.\overline{GotoA}$$

The circuit for the state machine has just one edge-triggered flip-flop and has logic to implement the Boolean function for the input to this flip-flop (*Q'*).
Although a simple state machine, its operation can be identified with the basic Fetch-Execute operation of a CPU. Taking state *A* as the *Fetch* state, and state *B* has the *Execute* state, then operation could be interpreted thus:-

- in the *Fetch* state, the *Q* signal is *0* and the *0* value enables logic elsewhere to read a word from memory.
- signal *GotoB* is set to *1* if all the instruction has been read in after this read; *GotoB* is set to *0* if further reads from memory are required to get the complete instruction. The number of words to read in is usually found by analysis of the first word of the instruction.
- when the state machine is clocked, it goes into the *Execute* state only when all the instruction is available, *GotoB = 1*; otherwise it stays in the *Fetch* state.
- in the *Execute* state, the *Q* signal is *1* and the *1* value enables logic elsewhere to execute the instruction.
- the machine stays in the *Execute* state until the execute logic indicates that execution is complete by setting *GotoA* to *1*, so that the machine moves back to the *Fetch* state.

An example of a more complex state diagram is shown right. The FSM arbitrates between 2 requests, *ReqA & ReqB*, for access to a shared resource and outputs 2 signals to grant access to the resource to a request.
The transitions are marked with the input values and the output values with a '/' separating the 2 sets.