# 7.8 Memory elements: Flip-flops, latches, and registers

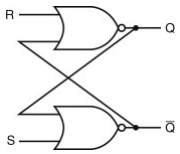
(Original section[1])

In this section and the next, we discuss the basic principles behind memory elements, starting with flip-flops and latches, moving on to register files, and finishing with memories. All memory elements store state: the output from any memory element depends both on the inputs and on the value that has been stored inside the memory element. Thus all logic blocks containing a memory element contain state and are sequential.

The simplest type of memory elements are *unclocked*; that is, they do not have any clock input. Although we only use clocked memory elements in this text, an unclocked latch is the simplest memory element, so let's look at this circuit first. The figure below shows an *S-R latch* (set-reset latch), built from a pair of NOR gates (OR gates with inverted outputs). The outputs $Q$ and $\overline{Q}$ represent the value of the stored state and its complement. When neither $S$ nor $R$ are asserted, the cross-coupled NOR gates act as inverters and store the previous values of $Q$ and $\overline{Q}$.

Figure 7.8.1: A pair of cross-coupled NOR gates can store an internal value (COD Figure A.8.1).

The value stored on the output $Q$ is recycled by inverting it to obtain $\overline{Q}$ and then inverting $\overline{Q}$ to obtain $Q$. If either $R$ or $\overline{Q}$ is asserted, $Q$ will be deasserted and vice versa.




For example, if the output, $Q$, is true, then the bottom inverter produces a false output (which is $\overline{Q}$), which becomes the input to the top inverter, which produces a true output, which is $Q$, and so on. If $S$ is asserted, then the output $Q$ will be asserted and $\overline{Q}$ will be deasserted, while if $R$ is asserted, then the output $\overline{Q}$ will be asserted and $Q$ will be deasserted. When $S$ and $R$ are both deasserted, the last values of $Q$ and $\overline{Q}$ will continue to be stored in the cross-coupled structure. Asserting $S$ and $R$ simultaneously can lead to incorrect operation: depending on how $S$ and $R$ are deasserted, the latch may oscillate or become metastable (this is described in more detail in COD Section A.11 (Timing methodologies)).

This cross-coupled structure is the basis for more complex memory elements that allow us to store data signals. These elements contain additional gates used to store signal values and to cause the state to be updated only in conjunction with a clock. The next section shows how these elements are built.

### Flip-flop and latches

*Flip-flops* and *latches* are the simplest memory elements. In both flip-flops and latches, the output is equal to the value of the stored state inside the element. Furthermore, unlike the S-R latch described above, all the latches and flip-flops we will use from this point on are clocked, which means that they have a clock input and the change of state is triggered by that clock. The difference between a flip-flop and a latch is the point at which the clock causes the state to actually change. In a clocked latch, the state is changed whenever the appropriate inputs change and the clock is asserted, whereas in a flip-flop, the state is changed only on a clock edge. Since throughout this text we use an edge-triggered timing methodology where state is only updated on clock edges, we need only use flip-flops. Flip-flops are often built from latches, so we start by describing the operation of a simple clocked latch and then discuss the operation of a flip-flop constructed from that latch.

***Flip-flop***: A memory element for which output is equal to the value of the stored state inside the element and for which the internal state is changed only on a clock edge.

***Latch***: A memory element in which the output is equal to the value of the stored state inside the element and the state is changed whenever the appropriate inputs change and the clock is asserted.

For computer applications, the function of both flip-flops and latches is to store a signal. A D latch or *D flip-flop* stores the value of its data input signal in the internal memory. Although there are many other types of latch and flip-flop, the D type is the only basic building block that we will need. A D latch has two inputs and two outputs. The inputs are the data value to be stored (called $D$) and a clock signal (called $C$) that indicates when the latch should read the value on the $D$ input and store it. The outputs are simply the value of the internal state ($Q$) and its complement ($\overline{Q}$). When the clock input $C$ is asserted, the latch is said to be open, and the value of the output ($Q$) becomes the value of the input $D$. When the clock input $C$ is deasserted, the latch is said to be closed, and the value of the output ($Q$) is whatever value was stored the last time the latch was open.

***D flip-flop***: A flip-flop with on data input that stores the value of that input signal in the internal memory when the clock edge occurs.

The figure below shows how a D latch can be implemented with two additional gates added to the cross-coupled NOR gates. Since when the latch is open the value of $Q$ changes as $D$ changes, this structure is sometimes called a *transparent latch*. COD Figure A.8.3 (Operation of a D latch …) shows how this D latch works, assuming that the output $Q$ is initially false and that $D$ changes first.

Figure 7.8.2: A D latch implemented with NOR gates (COD Figure A.8.2).

A NOR gate acts as an inverter if the other input is 0. Thus, the cross-coupled pair of NOR gates acts to store the state value unless the clock input, $C$, is asserted, in which case the value of input $D$ replaces the value of $Q$ and is stored. The value of input $D$ must be stable when the clock signal $C$ changes from asserted to deasserted.

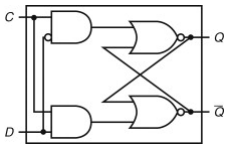
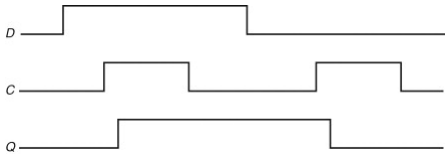



Figure 7.8.3: Operation of a D latch, assuming the output is initially deasserted (COD Figure A.8.3).

When the clock, *C*, is asserted, the latch is open and the *Q* output immediately assumes the value of the *D* input.




As mentioned earlier, we use flip-flops as the basic building block, rather than latches. Flip-flops are not transparent: their outputs change *only* on the clock edge. A flip-flop can be built so that it triggers on either the rising (positive) or falling (negative) clock edge; for our designs we can use either type. The figure below shows how a falling-edge D flip-flop is constructed from a pair of D latches. In a D flip-flop, the output is stored when the clock edge occurs. COD Figure A.8.5 (Operation of a D flip-flop with a falling-edge trigger …) shows how this flip-flop operates.

Figure 7.8.4: A D flip-flop with a falling-edge trigger (COD Figure A.8.4).

The first latch, called the master, is open and follows the input *D* when the clock input, *C*, is asserted. When the clock input, *C*, falls, the first latch is closed, but the second latch, called the slave, is open and gets its input from the output of the master latch.

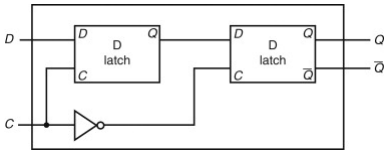
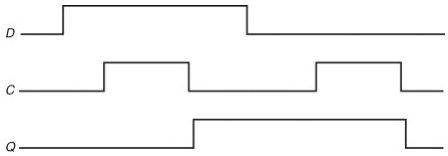



Figure 7.8.5: Operation of a D flip-flop with a falling-edge trigger, assuming the output is initially deasserted (COD Figure A.8.5).

When the clock input (*C*) changes from asserted to deasserted, the *Q* output stores the value of the *D* input. Compare this behavior to that of the clocked *D* latch shown in COD Figure A.8.3 (Operation of a D latch …). In a clocked latch, the stored value and the output, *Q*, both change whenever *C* is high, as opposed to only when *C* transitions.




Here is a Verilog description of a module for a rising-edge D flip-flop, assuming that C is the clock input and D is the data input:

```
module DFF(clock, D, Q, Qbar);
   input clock, D;
   output reg Q;            // Q is a reg since it is assigned in an always block

   output Qbar;
   assign Qbar = ~ Q;       // Qbar is always just the inverse of Q

   always @(posedge clock)  // perform actions whenever the clock rises
      Q = D;
endmodule
```

Because the *D* input is sampled on the clock edge, it must be valid for a period of time immediately before and immediately after the clock edge. The minimum time that the input must be valid before the clock edge is called the *setup time*; the minimum time during which it must be valid after the clock edge is called the *hold time*. Thus the inputs to any flip-flop (or anything built using flip-flops) must be valid during a window that begins at time $t_{setup}$ before the clock edge and ends at $t_{hold}$ after the clock edge, as shown in the figure below. COD Section A.11 (Timing methodologies) talks about clocking and timing constraints, including the propagation delay through a flip-flop, in more detail.

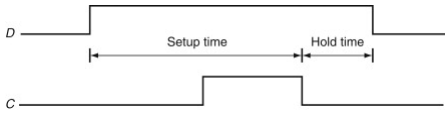
***Setup time***: The minimum time that the input to a memory device must be valid before the clock edge.

***Hold time***: The minimum time during which the input must be valid after the clock edge.

We can use an array of D flip-flops to build a register that can hold a multibit datum, such as a byte or word. We used registers throughout our datapaths in COD Chapter 4 (The Processor).

Figure 7.8.6: Setup and hold time requirements for a D flip-flop with a falling-edge trigger (COD Figure A.8.6).

The input must be stable for a period of time before the clock edge, as well as after the clock edge. The minimum time the signal must be stable before the clock edge is called the setup time, while the minimum time the signal must be stable after the clock edge is called the hold time. Failure to meet these minimum requirements can result in a situation where the output of the flip-flop may not be predictable, as described in COD Section A.11 (Timing methodologies). Hold times are usually either 0 or very small and thus not a cause of worry.




**Register files**

One structure that is central to our datapath is a *register file*. A register file consists of a set of registers that can be read and written by supplying a register number to be accessed. A register file can be implemented with a decoder for each read or write port and an array of registers built from D flip-flops. Because reading a register does not change any state, we need only supply a register number as an input, and the only output will be the data contained in that register. For writing a register we will need three inputs: a register number, the data to write, and a clock that controls the writing into the register. In COD Chapter 4 (The Processor), we used a register file that has two read ports and one write port. This register file is drawn as shown in the figure below. The read ports can be implemented with a pair of multiplexors, each of which is as wide as the number of bits in each register of the register file. COD Figure A.8.8 (The implementation of two read ports for a register file ...) shows the implementation of two register read ports for a 64-bit-wide register file.

Figure 7.8.7: A register file with two read ports and one write port has five inputs and two outputs (COD Figure A.8.7).

The control input Write is shown in color.

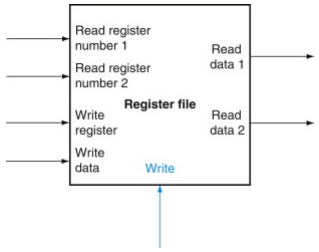
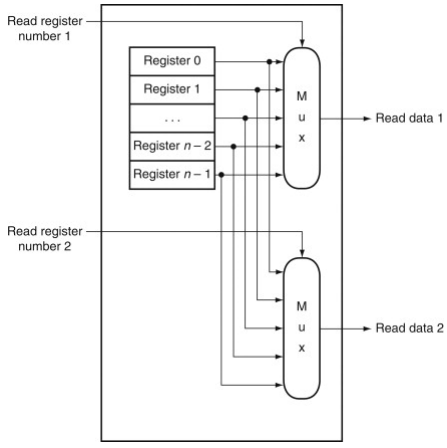



Figure 7.8.8: The implementation of two read ports for a register file with n registers can be done with a pair of n-to-1 multiplexors, each 64 bits wide (COD Figure A.8.8).

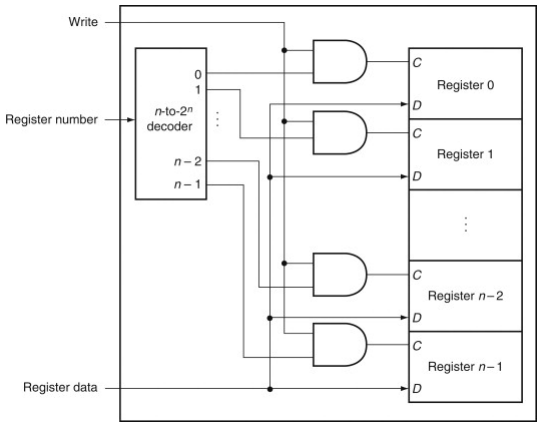
The register read number signal is used as the multiplexor selector signal. The figure below shows how the write port is implemented.




Implementing the write port is slightly more complex, since we can only change the contents of the designated register. We can do this by using a decoder to generate a signal that can be used to determine which register to write. The figure below shows how to implement the write port for a register file. It is important to remember that the flip-flop changes state only on the clock edge. In COD Chapter 4 (The Processor), we hooked up write signals for the register file explicitly and assumed the clock shown in the figure below is attached implicitly.

Figure 7.8.9: The write port for a register file is implemented with a decoder that is used with the write signal to generate the C input to the registers (COD Figure A.8.9).

All three inputs (the register number, the data, and the write signal) will have setup and hold-time constraints that ensure that the correct data is written into the register file.




What happens if the same register is read and written during a clock cycle? Because the write of the register file occurs on the clock edge, the register will be valid during the time it is read, as we saw earlier in COD Figure A.7.2 (The inputs to a combinational logic block come from a state element …). The value returned will be the value written in an earlier clock cycle. If we want a read to return the value currently being written, additional logic in the register file or outside of it is needed. COD Chapter 4 (The Processor) makes extensive use of such logic.

### Specifying sequential logic in Verilog

To specify sequential logic in Verilog, we must understand how to generate a clock, how to describe when a value is written into a register, and how to specify sequential control. Let us start by specifying a clock. A clock is not a predefined object in Verilog; instead, we generate a clock by using the Verilog notation `#n` before a statement; this causes a delay of `n` simulation time steps before the execution of the statement. In most Verilog simulators, it is also possible to generate a clock as an external input, allowing the user to specify at simulation time the number of clock cycles during which to run a simulation.

The code in the figure below implements a simple clock that is high or low for one simulation unit and then switches state. We use the delay capability and blocking assignment to implement the clock.

Figure 7.8.10: A specification of a clock (COD Figure A.8.10).

```
reg clock; // clock is a register
always
#1 clock = 1; #1 clock = 0;
```

Next, we must be able to specify the operation of an edge-triggered register. In Verilog, this is done by using the sensitivity list on an `always` block and specifying as a trigger either the positive or negative edge of a binary variable with the notation `posedge` or `negedge`, respectively. Hence, the following Verilog code causes register `A` to be written with the value b at the positive edge clock:

Throughout this chapter and the Verilog sections of COD Chapter 4 (The Processor), we will assume a positive edge-triggered design. The figure below shows a Verilog specification of a LEGv8 register file that assumes two reads and one write, with only the write being clocked.

Figure 7.8.11: A LEGv8 register file written in behavioral Verilog (COD Figure A.8.11).

This register file writes on the rising clock edge.

```
reg [63:0] A;
wire [63:0] b;
```

**PARTICIPATION ACTIVITY** 7.8.1: Check yourself: Register file implementation.

Refer to the Verilog register file implementation in the figure above.

1) The output ports corresponding to the registers being read are assigned using a continuous assignment, but the register being written is assigned in an always block. Which of the following is the reason?
   - ○ There is no special reason. Writing the register in an always block was simply convenient.
   - ○ Because Data1 and Data2 are output ports and WriteData is an input port.
   - ○ Reading is a combinational event, while writing is a sequential event.

(*1) This section is in original form.

**Provide feedback on this section**