

Laboratory Exercise 4

Latches, Flip-flops, and Registers

Revision of October 12, 2023

The purpose of this exercise is to investigate the fundamental synchronous logic elements: latches, flip-flops, and registers.

For Part I you will use the *breadboard* to build and test a circuit. For Parts II and III of the lab you should begin by writing and testing Verilog code and simulating it with ModelSim. You should be prepared to show schematics, Verilog, and simulations to your TA, if requested. You must simulate your circuit with ModelSim using reasonable test vectors written in the format used in Lab 2 for the simulation files.

1 Part I

In this part, you will build a D-Latch (textbook Section 5.3). Specifically, a gated D-Latch shown in Figure 1. You must once again use the 7400 chips (as in Lab 1) and the protoboard (breadboard). Refer back to the Lab 1 handout for the specifications of the 7400 chips.

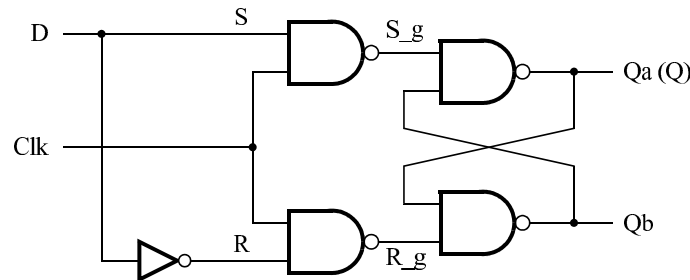


Figure 1: Circuit for a gated D latch.

1.1 What to Do

Perform the following steps:

1. *Prior to coming to lab*, draw a schematic of the gated D latch using 7400-series chips. Recall from Lab 1 what a gate-level schematic looks like.
2. Build the gated D latch using the chips and breadboard. Use switches to control the clock and D input. Use LEDs to make Qa and Qb visible. Don't forget to hook up the power and ground on all of your chips!
3. Study the behaviour of the latch for different D and Clk settings. Observe Q when Clk is set high and you change D several times. Then observe Q when Clk is set low and you change D several times. How do you set Q high? How do you set Q low?
4. What are all the cases you need to show that your D latch is working correctly?

Once your circuit works as expected, demo it to your TA for marking.

2 Part II

In this section, you must build an ALU which includes registers to store values. Registers are sequential storage elements, consisting of several D flip flops working in unison.

2.1 Introduction to D flip flops

In modern digital circuit design, latches are rarely used, and only in very special circumstances. The most common storage element today is the *edge-triggered D flip flop*. One way to build an edge-triggered D flip flop is to connect two D latches in series with the two D latches using opposite edges of the clock. This is called a leader-follower flip flop¹ (textbook Section 5.4.1). The output of the flip flop changes on a clock *edge*, unlike the latch, which changes according to the *level* of the clock. For a positive edge-triggered flip flop, the output changes when the clock edge *rises*. The code for a positive edge-triggered flip flop is shown in Listing 1 (textbook Section 5.13 and 5.14 have more examples).

```
module D_flip_flop(
    input wire clk,
    input wire reset_b,
    input wire d,
    output reg q
);
    always@(posedge clk)
    begin
        if (reset_b) q <= 1'b0;
        else q <= d;
    end
endmodule
```

Listing 1: Code for a 1-bit D-Flip flop with synchronous active-high reset.

There are several important things to note about the code shown in Listing 1

1. The flip-flop is created using a new definition inside **always** block. Verilog uses different arguments inside always blocks to spot whether the code is creating combinational (using **always@(*)**) or sequential (using **always@(posedge clk)**) logic.
2. For now, statements inside **always** blocks should use **<=** (non-blocking) instead of regular **=** (blocking) assignments. Using **=** inside **always** blocks can lead to your circuit behaving

¹In the past and in the textbook it was called master-slave flip flop, the industry is transitioning away from this terminology

in odd ways². Always double check that you are using the right type of assignment.

3. This flip flop has an **active-high, synchronous reset**, meaning that the reset only happens when *reset_b* = 1 on the rising clock edge.
4. If *q* is declared as **reg** *q*, then you get a single flip flop. If *q* is declared as **reg[7:0]** *q*, then you get eight parallel flip flops, which is called an *8-bit register*. Of course, *d* should have the same width as *q*.

NOTE: When creating a flip flop, the **if conditions** inside the **always** block are checked sequentially, same as in software, so think about the ordering carefully. In order for the circuit to behave as we expect, you should always check the reset condition first inside your **always** block.

²The `<=` operator means all actions in the **always** block should happen at the same time. `=` implies ordering meaning the next line can not happen until the previous line completes if the input of the next line depends on the previous line output.

2.2 Designing a sequential ALU

Similar to Lab 3, you must now build an ALU with the four operations shown in Listing 2. *Note* that the operations here are not the same as in Lab 3. The output of the ALU is to be stored in an 8-bit *register* and the four least-significant bits of the register output are connected to the *B* input of the ALU. The 8-bit register should have an **active-high synchronous** reset. Figure 2 shows a high-level schematic of the ALU you must build.

```
always@(*)
begin
    case (Function)
    0: A + B the '+' operator.
    1: A * B using the '*' operator.
    2: Left shift B by A bits using the shift operator.
    3: Hold current value in the Register, i.e., the register
        value does not change.
    default: 0
    endcase
end
```

Listing 2: Pseudo-code for ALU

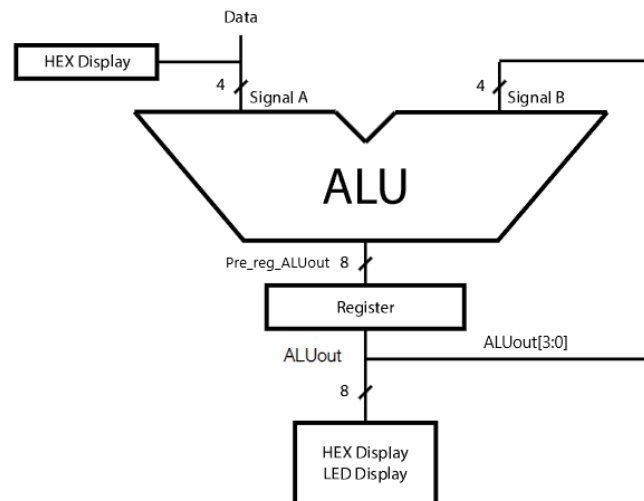


Figure 2: Simple ALU with register circuit for Part II.

2.3 What to Do

The top-level module of your design should have the following signature declaration:

```
module part2(Clock, Reset_b, Data, Function, ALUout);
```

1. *Prior to coming to the lab* write code that corresponds to the schematic on Figure 2. Your Verilog code should use the same names for the wires and instances as shown in your schematic. Use the code in Listing 1 as the model for your register code.
2. *Prior to coming to the lab* Simulate your ALU with ModelSim to satisfy yourself that your circuit is working. Be prepared to justify that your test cases are enough to give confidence that your circuit is working. Each group member should prepare at least 1 unique edge case with a justification. When you are satisfied with your simulations, you can submit to the Automarker.

2.4 Running on the FPGA

If you wish to run the code on the FPGA, you can use the port mapping shown in Table 1. This however will not be part of the demo.

NOTE 1: When using Quartus, avoid module names such as *dff*. This is a reserved Quartus keyword and modules with this name will not be synthesized. This is reported as a *warning* in Quartus but it's easy to miss.

Note 2: All mechanical switches, such as a push/toggle button, will often make contact several times due the electrical contacts bouncing. This happens quickly in human time, but not in electrical time. With a bouncing switch you can observe multiple high-frequency toggles making it difficult to create single clock edges. If you run into bounce problems with KEY_0 for your clock you can try to use KEY_1 instead to see if that is better.

module Port Name	Direction	DE1-SoC Pin Name
Clock	Input	KEY[0]
Reset_b	Input	KEY[1]
Data	Input	SW[3:0] shown on HEX0
Function	Input	SW[9:8] shown on HEX1
ALUout	Output	shown on LEDR[7:0] and also on HEX4 and HEX5

Table 1: Module port mapping to DE1-SoC/DE10-Lite pin names

3 Part III

A basic element in designing sequential logic is the **shift register** (textbook Section 5.8). When bits are shifted in a register, it means that the bits are copied to the next flip flop on the left or the right. For example, to shift the bits left, each flip flop loads the value of the flip flop to its right when the clock edge occurs. In a normal shift register, the right-most register is loaded with an input value or with a fixed value such as 0.

A more advanced version of the shift register is the **rotating register**, which feeds the value shifted out back to the input of the shift register, thus ‘rotating’ the value. The only way to change the value in a rotating register is to use the ‘parallel load’ input to set all the D-flip flops at the same time. In this part of the lab you must design a 4-bit rotating register with parallel load. Each bit of this rotating register consists of a positive edge-triggered D flip flop and several multiplexers as shown in Figure 3a. The mux closest to the flip flop supports parallel load using the `loadn` input. The second mux controls the direction of rotation. The D flip flops should have an **active-high synchronous** reset.

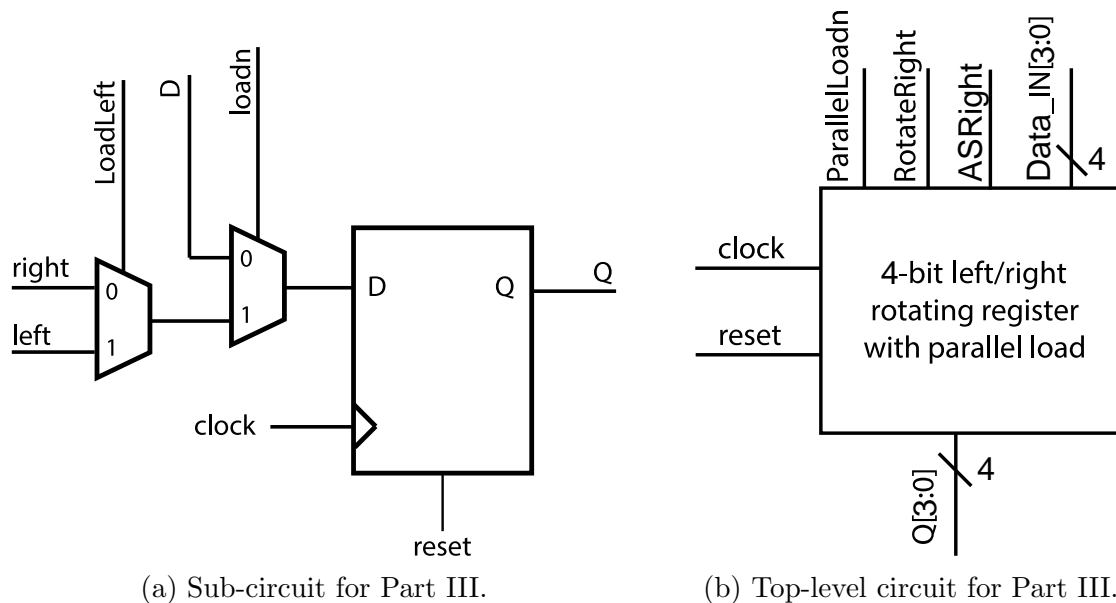


Figure 3: Rotating register for Part III.

The top-level circuit is shown in Figure 3b and consists of 4 instances of the circuit shown in Figure 3a. Let’s take a look at the function of each of the inputs shown in Figure 3b.

1. *Data_IN* connects to the *D* input of the 4 registers.
2. *ParallelLoadn* controls loading a value into all 4 registers.
3. The *LoadLeft* input of all four instances of the circuit in Figure 3a should be tied to the

single rotating register input *RotateRight* because when you want to rotate the bits right, you have to load the bit to the left.

4. *ASRight* is a special input to perform an *arithmetic shift right* on the register. Arithmetic shift is done to support shifting of *signed* numbers (textbook Section 3.3). You will learn about signed numbers in class in the future. For now, we will explain what you have to do to implement this functionality below.

Let's take a look at some examples of how your design should work:

1. When *ParallelLoadn* = 0, the value on *Data_IN* is stored in the flip-flops on the next positive clock edge (i.e., parallel load behaviour).
2. When *ParallelLoadn* = 1, *RotateRight* = 1 and *ASRight* = 0 the bits of the register rotate to the right on each positive clock edge (notice the bits rotate to the right with wrap around):

$Q_7 Q_6 Q_5 Q_4 Q_3 Q_2 Q_1 Q_0$
 $Q_0 Q_7 Q_6 Q_5 Q_4 Q_3 Q_2 Q_1$
 $Q_1 Q_0 Q_7 Q_6 Q_5 Q_4 Q_3 Q_2$
 \dots

3. When *ParallelLoadn* = 1, *RotateRight* = 1 and *ASRight* = 1 the bits of the register shift to the right on each positive clock edge but the most significant bit is copied. This is called an *Arithmetic shift right*:

$Q_7 Q_6 Q_5 Q_4 Q_3 Q_2 Q_1 Q_0$
 $Q_7 Q_7 Q_6 Q_5 Q_4 Q_3 Q_2 Q_1$
 $Q_7 Q_7 Q_7 Q_6 Q_5 Q_4 Q_3 Q_2$
 \dots

4. When *ParallelLoadn* = 1 and *RotateRight* = 0, the bits of the register rotate to the left on each positive clock edge. *ASRight* is ignored:

$Q_7 Q_6 Q_5 Q_4 Q_3 Q_2 Q_1 Q_0$
 $Q_6 Q_5 Q_4 Q_3 Q_2 Q_1 Q_0 Q_7$
 $Q_5 Q_4 Q_3 Q_2 Q_1 Q_0 Q_7 Q_6$
 \dots

3.1 What to Do

The top-level module of your design should have the following signature declaration:

```
module part3(clock, reset, ParallelLoadn, RotateRight, ASRight, Data_IN, Q);
```

1. Draw a schematic for the 4-bit rotating register with parallel load. Your schematic should contain four instances of the sub-circuit in Figure 3a and all the wiring required to implement the desired behaviour. Label the signals on your schematic with the same names you will use in your Verilog code.
2. Using the code in Listing 1, write a module to implement the circuit shown in Figure 3a.
3. Now, write a module that instantiates four instances of the previous module which implements Figure 3a.
4. Simulate your rotating register with ModelSim to satisfy yourself that your circuit is working. In your simulation, you should perform the reset operation first. Then, clock the register for several cycles to demonstrate rotation in the left and right directions. Be prepared to justify that your test cases are enough to give confidence that your circuit is working.
5. When you are satisfied with your simulations, you can submit to the Automarker.

NOTE: If you do not perform a reset first, your simulation will not work! Try simulating without doing reset first and see what happens. Can you explain the results?

3.2 Running on the FPGA.

If you wish to run your code on the FPGA, you can use the port mapping shown in Table 2. This is not part of the demo but helps with the debug process.

module Port Name	Direction	DE1-SoC Pin Name
clock	Input	KEY[0]
reset	Input	KEY[1]
Data_IN	Input	SW[3:0] shown on HEX0
ParallelLoadn	Input	SW[9]
RotateRight	Input	SW[8]
ASRight	Input	SW[7]
Q	Output	LEDR[3:0] and HEX4

Table 2: Module port mapping to DE1-SoC/DE10-Lite pin names

4 Submission

When submitting to the Automarker make sure you have modules declared as shown below as the Automarker will be looking for modules with these exact signatures.

4.1 Part I

Part 1 is demonstrated in the lab and does not need to be submitted to the automarker.

4.2 Part II

For Part II, you need to submit a file named `part2.v` with the following module in it:

```
1. module part2(Clock, Reset_b, Data, Function, ALUout);
```

4.3 Part III

For Part III, you need to submit a file named `part3.v` with the following module in it:

```
1. module part3(clock, reset, ParallelLoadn, RotateRight, ASRight, Data_IN, Q);
```