Laboratory Exercise 4

Latches, Flip-flops, and Registers
October 9, 2016

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

Preparation Before the Lab

Review the instructions in Lab 2 about preparations.

Prepare for Part I by drawing the schematic. Prepare for Parts II and III of the lab by writing and testing Verilog code and compiling it with Quartus. Show your schematics, Verilog, and simulations for Parts II to III to the teaching assistants. You must simulate your circuit with ModelSim (using reasonable test vectors using ModelSim scripts).

In-lab Work

You are required to implement and test all of Parts I to III of the lab. You need to demonstrate all parts to the teaching assistants.

Part I

Figure 1 shows the circuit for a gated D latch. In this part, you will build the gated D latch using 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.

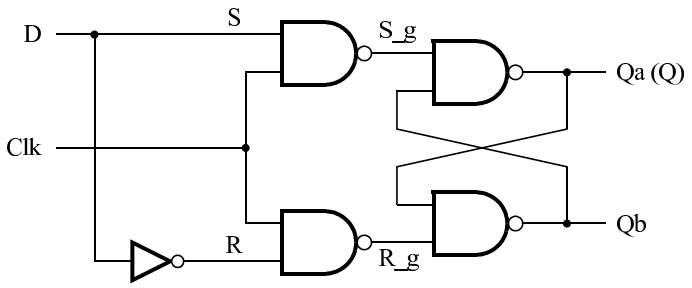

Figure 1: Circuit for a gated D latch.

Perform the following steps:

1. In your lab book, draw a schematic of the gated D latch using interconnected 7400-series chips. Recall from Lab 1 what a gate-level schematic looks like.
2. Build the gated D latch using the chips and protoboard. Use switches to control the clock and D input. Use lights 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 clock settings.

4. Demonstrate your latch implementation to the TA.

Part II

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 master-slave flip flop. The output of the master-slave 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 Verilog code for a positive edge-triggered flip flop is shown in Figure 2. This flip flop also has an active-low, synchronous reset, meaning that the reset only happens when *Reset_b* = 0 on the rising clock edge. 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*.

```
always @(posedge Clock)        // triggered every time clock rises
begin
   if (Reset_b == 1'b0)        // when Reset_b is 0 (note this is tested on every rising clock edge)
      q <= 0;                  // q is set to 0. Note that the assignment uses <=
   else                        // when Reset_b is not 0
      q <= d;                  // value of d passes through to output q
end
```

Figure 2: Verilog for a positive edge-triggered flip flop with active-low, synchronous reset.

Starting with the circuit you built for Lab 3 Part III build an ALU with the eight operations as shown in the pseudo-code in Figure 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. Figure 4 shows the required connections.

```
always @(*)          // declare always block
begin
   case (function)   // start case statement
      0: A + B using the adder from Part II of this Lab
      1: A + B using the Verilog '+' operator
      2: A OR B in the lower four bits and A XOR B in the upper four bits
      3: Output 8'b10000001 if at least 1 of the 8 bits in the two inputs is 1 using a single OR operation
      4: Output 8'b01111110 if all of the 8 bits in the two inputs are 1 using a single AND operation
      5: Left shift B by A bits
      6: A × B using the Verilog '*' operator
      7: Hold current value in the Register
      default: ...  // default case
   endcase
end
```

Figure 3: Pseudo-code for ALU.

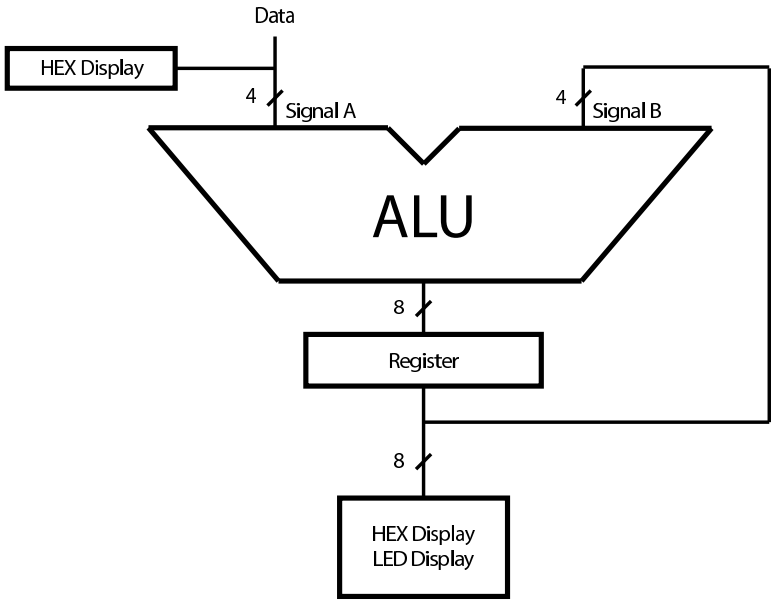

Figure 4: Simple ALU with register circuit for Part II.

Perform the following steps.

1. Draw a schematic showing your code structure with all wires, inputs and outputs labeled. Be prepared to explain it to the TA as part of your preparation.
2. Create a Verilog module for the simple ALU with register. Use the code in Figure 2 as the model for your register code. Connect the *Data* input to switches SW_{3-0}. Connect KEY_0 to the Clock input for the register, SW_9 to *Reset_b* and use KEY_{3-1} for the ALU function inputs. Display the outputs on $LEDR_{7-0}$; have *HEX0* display the value of *Data* and set *HEX1*, *HEX2* and *HEX3* to 0. *HEX4* and *HEX5* should display the least-significant and most-significant four bits of *Register* respectively.
3. Create a new Quartus project for your circuit.
4. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must show this to the TA as part of your preparation.
5. Compile the project.
6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit.

Part III

Figure 5 shows a positive edge-triggered flip-flop with several multiplexers. In this part of the lab, you will use eight instances of the circuit in Figure 5 to design a left/right 8-bit rotating register with parallel load shown in Figure 6.

A rotating register uses the concept of *shifting* bits in the register. 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. The term *rotating* comes from how the bits at the ends of the register are handled. In the left-shift example, the flip flop at the right end of the register has no right neighbour. One option is to load a zero, but for *rotation* we load the value of the flip flop at the left end of the register. The behaviour is as if the register were really a ring because the left and right ends are connected.

The *LoadLeft* input of all eight instances of the circuit in Figure 5 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. The *loadn* input of all eight instances should be tied to the single rotating register input *ParallelLoadn*. The *clock* input of all eight instances should be tied to the single rotating register input *clock*. Create an 8-bit-wide rotating register input *DATA_IN*, whose individual wires $DATA\_IN_7$ to $DATA\_IN_0$ are tied to the *D* input of each instance of the circuit in Figure 5. Likewise, create an 8-bit-wide rotating register output Q, whose individual wires Q_7 to Q_0 are tied to the Q output of each instance of the circuit in Figure 5.

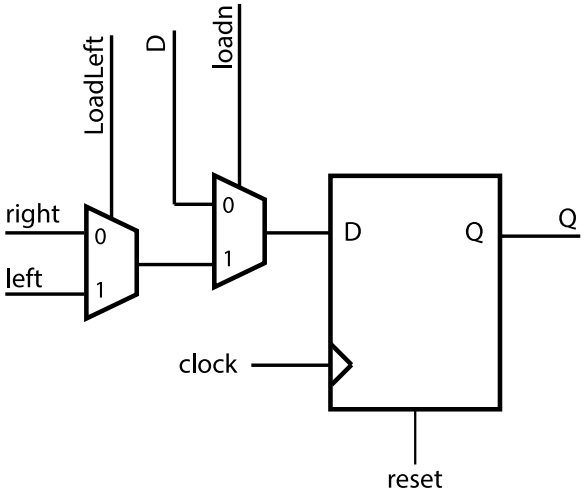

Figure 5: Sub-circuit for Part III.

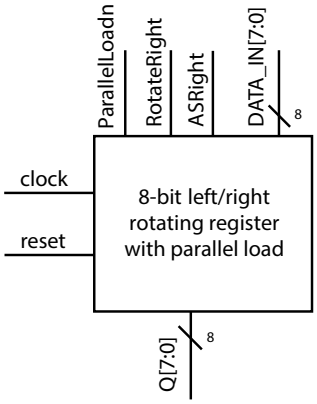

Figure 6: Top-level circuit for Part III.

The remaining connections between the eight instances of the circuit in Figure 5 should realize the following behaviour:

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):

   $Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$
   $Q_0Q_7Q_6Q_5Q_4Q_3Q_2Q_1$
   $Q_1Q_0Q_7Q_6Q_5Q_4Q_3Q_2$
   ...

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

   $Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$
   $Q_7Q_7Q_6Q_5Q_4Q_3Q_2Q_1$
   $Q_7Q_7Q_7Q_6Q_5Q_4Q_3Q_2$
   ...

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

   $Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$
   $Q_6Q_5Q_4Q_3Q_2Q_1Q_0Q_7$
   $Q_5Q_4Q_3Q_2Q_1Q_0Q_7Q_6$
   ...

Do the following steps:

1. Draw a schematic for the 8-bit rotating register with parallel load. Your schematic should contain eight instances of the sub-circuit in Figure 5 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. Starting with the code in Figure 2 for a flop flop, modify it to have an *active-high* synchronous reset. Combine this new flip flop with instances of the *mux2to1* module from Lab 2 to build the sub-circuit shown in Figure 5. To get you started, Figure 7, is a sample of hierarchical code showing the D flip flop with one of the 2-to-1 multiplexers connected to it.

```
mux2to1 M1(                 //instantiates 2nd multiplexer
    .x(rotatedata)          //output from left most multiplexer
    .y(data_D)              //data D coming in
    .s(parallel_loadn)      //selects input D or rotate
    .m(datato_dff)          //outputs to flip flop
);

flipflop F0(                //instantiates flip flop
    .d(datato_dff)          //input to flip flop
    .q(out_Q)               //output from flip flop
    .clock(clock)           //clock signal
    .reset(reset)           //synchronous active high reset
);
```

Figure 7: Part of the code for the sub-circuit in Figure 5.

3. Create a new Quartus project.

4. Write a Verilog module for the rotating register with parallel load that instantiates eight instances of your Verilog module for Figure 5. This Verilog module should match with the schematic in your lab book. Use SW_{7-0} as the inputs $DATA\_IN_{7-0}$ and SW_9 as a synchronous active high reset. Use KEY_1 as the *ParallelLoadn* input, KEY_2 as the *RotateRight* input and KEY_3 as the *ASRight* input. Use KEY_0 as the clock, but **read the important note below about switch bouncing.** The outputs Q_{7-0} should be displayed on the LEDs ($LEDR_{7-0}$).

5. Include the Verilog code in your project.

6. Compile your Verilog code and simulate the design with ModelSim. 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. **(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?)**

7. Download your circuit on the DE1-SoC board.

8. Test the functionality of your rotating register.

Note: If you run into bounce problems with KEY_0 for your clock you are welcome to try using any of the keys. 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.