CSC258 - Lab 4

Latches, Flip-flops, and Registers

Fall 2018

1 Learning Objectives

The purpose of this lab is to investigate the fundamental synchronous logic elements: latches, flip-flops, and registers. In this lab you will build a gated D-Latch with the 7400 logic chips and write Verilog to create a registered ALU and a shift register.

2 Marking Scheme

Each lab is worth 4% of your final grade, but you will be graded out of 8 marks for this lab, as follows.

- Prelab + Simulations: 3 marks
- Part I (in-lab): 1 mark
- Part II (in-lab): 2 mark
- Part III (in-lab): 2 marks

3 Preparation Before the Lab

You are required to complete the prelab for Part I of the lab as you would have prepared for Lab 1. Parts II and III of the lab require you to write and test your Verilog code. Include your schematics, Verilog, and simulations (where applicable) for Parts I to III in the prelab. You must simulate your circuits 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 TAs after you tested them yourselves.

4 Latches in Verilog

In modern digital circuit design, latches are rarely used, and only in very special circumstances. On FPGAs especially, we seldom use latches except in very specific designs. Most of the time, if we create a latch in Verilog in a design for an FPGA, we have created the latch in error. To explore the behaviour of latches, in this lab you will create a latch using only the 7400 logic chips.

In Verilog, when we use `always @(*)` to create combinational logic, we sometimes inadvertently create latches. For instance, this happens when you use the equal (=) sign to set an output value, but don't specify what the output should be for all possible combinations of input values. The circuit assumes it has to retain a past value in those cases, creating a latch.

For example in the following code:

```
reg w;
always @(*)
begin
    if (x)              // when x is true (i.e., logic 1)
        w = 1;          // w gets set to 1
end
```

w does **not** have a value specified when x is 0. In this instance a latch would be created. To fix this code and create a proper combinational circuit, w should have a value specified for all possible values of inputs, meaning that the truth table for the logic function is fully specified. From now on, *you should check your compilation log in Quartus* and look out for warnings that latches have been created. An example warning message may look like this:

Warning (10240): Verilog HDL Always Construct warning at mymodule.v(9): inferring latch(es) for variable "w", which holds its previous value in one or more paths through the always construct

The warning message clearly identifies that identifier *w* caused the latch to be created. It also points to the file (mymodule.v) and line number (9) where the offending always block resides. In your code, the module name, identifier name, and line number will be different, of course. Note that this is a warning message (as opposed to an error or critical warning), so make sure that your filters in the Messages window are *not* set to hide warning messages. Message filtering can be achieved by clicking on the red circle (for error messages), yellow triangle (for warnings), and purple triangle (for critical warnings). The Messages window is usually situated at the bottom of the Quartus window. If you closed this window by mistake, you can enable it again by selecting View > Utility Windows > Messages.

5 Shift Operators in Verilog

Verilog has the following two logic shift operators: (a) Logic Right Shift operator (`>>`) and (b) Logic Left Shift operator (`<<`). Doing `(A >> N)` shifts A by N bits to the right; the N most significant bits of the resulting vector are filled-in with zeros. Doing `(A << N)` shifts A by N bits to the left; the N least significant bits of the resulting vector are filled-in with zeros. As with any combinational circuit A and N are inputs to the circuit and the shifted result is the output.

Here is a Verilog snippet that uses these two operators:

```
wire [2:0] a, b;
wire c;

assign c = 1'b1;
assign a = (3'b011 >> 1'b1); // a is 3'b011 shifted right one bit
                             // result: a = 3'b001
assign b = a << c; // b is a shifted left one bit
                   // result: b = 3'b010
```

6 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) on the protoboard (breadboard). Refer back to the Lab 1 handout for 7400 chips schematics, specifications, and pin-out.

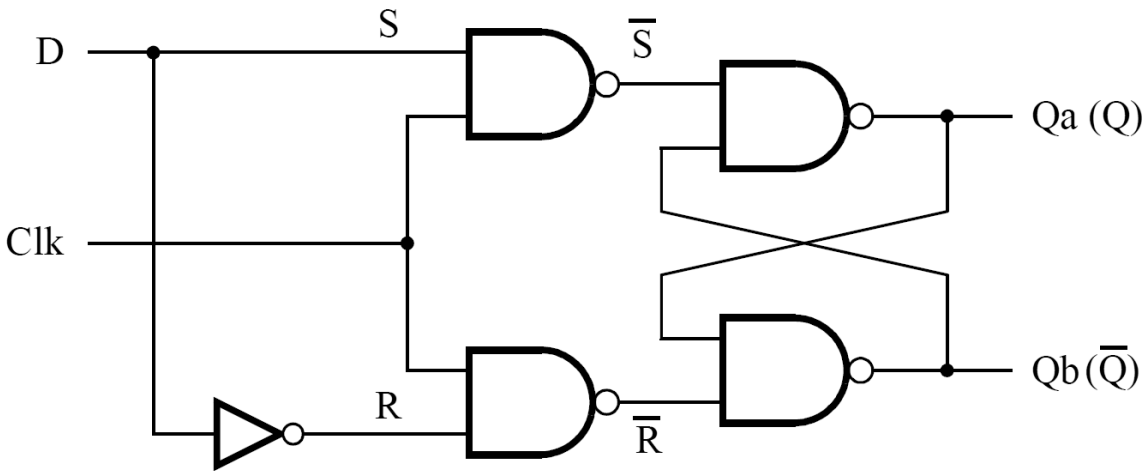

Figure 1: Circuit for a gated D latch.

Perform the following steps:

1. In your pre-lab, draw a schematic of the gated D latch using interconnected 7400-series chips. For this exercise you are allowed to use NAND gates. Recall from Lab 1 what a gate-level schematic looks like. **(PRELAB)**
2. Build the gated D latch using the chips on the protoboard. Use switches to control the clock (Clk) and D input. Use lights (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 clock (Clk) settings.
4. Are there any input combinations of Clk and D that should NOT be the first you test? Explain this in your prelab and list them if applicable. **(PRELAB)**
5. Demonstrate your latch implementation to the TA after you have tested it. **(IN-LAB)**

7 Part II

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, such that the two D latches use opposite levels of the clock for gating the latch. 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*, i.e., when clock transitions from 0 to 1. 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_n* is 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*.

[1]For a negative edge-triggered flip-flop, substitute the `posedge` keyword with `negedge`.

```
always @(posedge clock)        // Triggered every time clock rises
                               // Note that clock is not a keyword
begin
    if (reset_n == 1'b0)       // When reset_n is 0
                               // Note this is tested on every rising clock edge
        q <= 0;                // Set q to 0.
                               // Note that the assignment uses <= instead of =
    else                       // When reset_n is not 0
        q <= d;                // Store the value of d in q
end
```

Figure 2: Verilog for a positive edge-triggered flip-flop with active-low, synchronous reset[1].

Note that all assignment statements in the aforementioned code use <= instead of =. From now on you should use the assignment operator = only for combinational circuits, and the assignment operator <= for sequential circuits. Sequential circuits are circuits where the output relies not just on the current combination of inputs, but also on the prior state of the circuit (*i.e.*, its prior input sequence). Combinational circuits are described using `assign` statements or `always @(*)` blocks, while sequential circuits are described using `always @(posedge...)` and `always @(negedge...)` blocks. Note that you can place multiple statements inside of an `if-else` statement if you enclose such statements inside of a `begin-end` block.

Starting with the circuit you built for Lab 3 Part III, build an ALU that supports the eight operations 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 to be connected to the *B* input of the ALU. Figure 4 shows the required connections.

```
always @(*)          // Declare always block
begin
  case (ALU_function)//Start of the case statement
    0: ...           // Make the output equal to A+1, using the adder
                     // circuit from Part II of Lab 3.
    1: ...           // A + B using the adder from Part II of Lab 3
    2: ...           // A + B using the Verilog '+' operator
    3: ...           // A XOR B in the lower four bits, A OR B in the upper four bits
    4: ...           // Output 1 (8'b00000001) if any of the 8 bits in
                     // either A or B are high, and 0 (8'b00000000)
                     // if all the bits are low (use a reduction OR operator)
    5: ...           // Left shift B by A bits
    6: ...           // Right shift B by A bits (logical right shift)
    7: ...           // A × B using the Verilog * operator
    default: ...     // Default case (if needed)
  endcase
end
```

Figure 3: Pseudo-code for ALU.

Perform the following steps.

1. Create a Verilog module for the simple ALU with register, with the following specifications: **(PRELAB)**
   - Use the code in Figure 2 as the model for your register code.
   - Connect the *Data (A)* input of your ALU to switches SW_{3-0}.

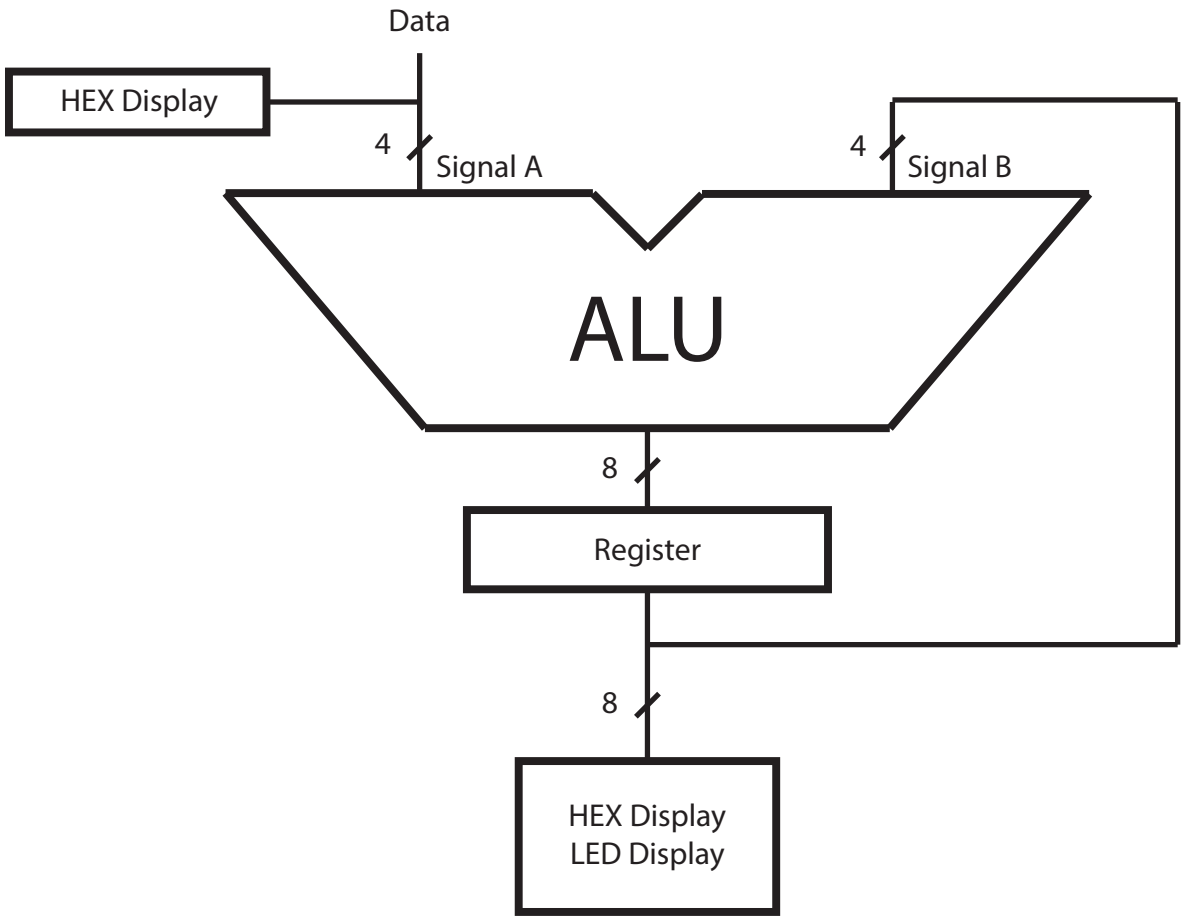

Figure 4: Simple ALU with register circuit for Part II.

- Connect KEY_0 to the Clock input for the register, SW_9 to *reset_n* and use SW_{7-5} for the ALU_function inputs.
- Display the ALU outputs on $LEDR_{7-0}$; have *HEX0* display the value of *Data (A)* in hexadecimal and set *HEX1*, *HEX2* and *HEX3* to display nothing (all segments off).
- *HEX4* and *HEX5* should display the least-significant and most-significant four bits of *Register* (ALU outputs) respectively, also in hexadecimal.

2. Simulate your circuit with ModelSim, ensuring the output waveforms are correct for each of your test cases. Choose test cases that make you feel confident about your ALU's correctness, in preparation for you in-lab demo. Your prelab should include at least one test for each ALU operation, though more test cases per operation is advisable. Make sure to include a few selected screenshots of these cases when you hand in your prelab. **(PRELAB)**
3. Create a new Quartus Prime project for your circuit. Make sure it is stored in your `W:\` drive. Do not forget to select the correct FPGA device (5CSEMA5F31C6) and import the pin assignments.
4. Compile the project.
5. Download the compiled circuit into the FPGA chip and test the functionality of the circuit. When it is working as it should, show it to TAs. **(IN-LAB)**

8 Part III

In this part of the lab, you will create an 8-bit shift-register that has an optional arithmetic shift. A shift register is a collection of flip-flops that move values sequentially between each other on each clock edge. Figure 5 shows one bit of this shift-register. It contains a positive edge-triggered flip-flop and two multiplexers. To create an 8-bits shift-register, you will use eight instances of the circuit in Figure 5 to design your 8-bit shift-register with optional arithmetic shift and parallel load as shown in Figure 6.

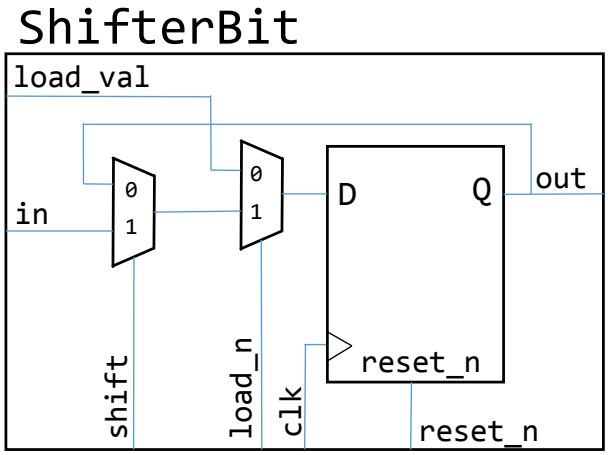

Figure 5: Single-bit shift-register

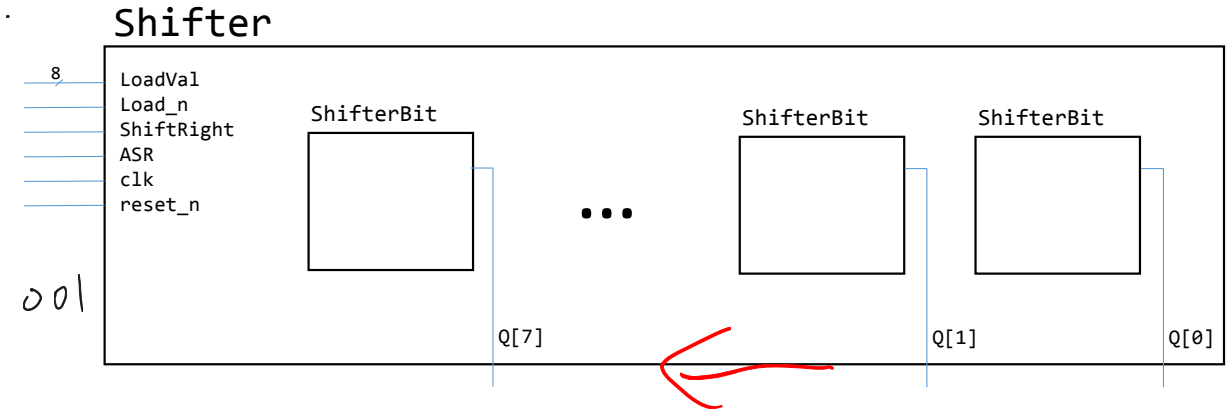

Figure 6: 8-bit shift-register of Part III. **None** of internal connections are shown here.

When bits are shifted in this register, it means that the bits are copied to the next flip-flop on the right. For example, to shift the bits right, each flip-flop loads the value of the flip-flop to its left when the clock edge occurs. In the right-shift, the flip-flop at the left end of the register has no left neighbour. One option is to load a zero, but what if the value in the register is signed? In this case we should perform *sign-extension*. When we perform the sign-extension, this shift operation is called an *Arithmetic Shift Right* (ASR).

In the Shifter module, create an 8-bit-wide register input *LoadVal*, whose individual wires are connected to `load_val` input of ShifterBit instances. Likewise, create an 8-bit-wide output *Q*, which is the output of ShifterBit instances. The *shift* input of all eight instances of the circuit in Figure 5 should be connected to the single input *ShiftRight*. The same thing applies to *load_n*, *clock (clk)*, and *reset_n*.

The *in* input of seven instances should be connected to the *out* port of the instance to its left. For the leftmost ShifterBit, you should design a circuit that will perform sign-extension when the signal *ASR* is high (arithmetic right shift) or load zeros if *ASR* is low (logic right shift). This special circuit is not shown in Figure 6.

Here is an example of the circuit operation:

1. When *Load_n = 0*, the value on *LoadVal* is stored in the flip-flops on the next positive clock edge (*i.e.*, parallel load behaviour).
2. When *Load_n = 1*, *ShiftRight = 1* and *ASR = 0*, the bits of the register shift to the right on each positive clock edge. Assuming that the initial value in the flip-flops at cycle 0 is A, with bits A_7 through A_0, the values in the two subsequent cycles would be:

	$Q[7]$	$Q[6]$	$Q[5]$	$Q[4]$	$Q[3]$	$Q[2]$	$Q[1]$	$Q[0]$
Cycle 0:	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
Cycle 1:	0	A_7	A_6	A_5	A_4	A_3	A_2	A_1
Cycle 2:	0	0	A_7	A_6	A_5	A_4	A_3	A_2
...								

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

	$Q[7]$	$Q[6]$	$Q[5]$	$Q[4]$	$Q[3]$	$Q[2]$	$Q[1]$	$Q[0]$
Cycle 0:	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
Cycle 1:	A_7	A_7	A_6	A_5	A_4	A_3	A_2	A_1
Cycle 2:	A_7	A_7	A_7	A_6	A_5	A_4	A_3	A_2
...								

Do the following steps:

1. What is the behaviour of the 8-bit shift register shown in Figure 6 when *Load_n = 1* and *ShiftRight = 0*? Briefly explain in your prelab. **(PRELAB)**
2. Draw a schematic for the 8-bit shift register shown in Figure 6 including the necessary connections. Your schematic should contain eight instances of the one-bit shifter shown 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. **(PRELAB)**
3. Starting with the code in Figure 2 for a positive edge-triggered D flip-flop, use this D flip-flop with instances of the *mux2to1* module from Lab 2 to build the one-bit shifter shown in Figure 5. To get you started, Figure 7 is a code snippet that shows how the D flip-flop will be connected to one of the 2-to-1 multiplexers. **(PRELAB)**

```
mux2to1 M1(                    // instantiates 2nd multiplexer
    .x(load_val),              // the parallel load value
    .y(data_from_other_mux),
    .s(load_n),
    .m(data_to_dff)            // outputs to flip-flop
);

flipflop F0(                   // instantiates flip-flop
    .d(data_to_dff),           // input to flip-flop
    .q(out),                   // output from flip-flop
    .clock(clk),               // clock signal
    .reset_n(reset_n)          // synchronous active low reset
);
```

Figure 7: Helper code snippet to implement one-bit shifter shown in Figure 5.

4. Write a Verilog module for the shift register that instantiates eight instances of one-bit shift register that you created in previous step. This Verilog module should match with the schematic in your prelab. Use SW_{7-0} as the inputs $LoadVal_{7-0}$ and SW_9 as a synchronous active low reset (reset_n).

Use KEY_1 as the *Load_n* input, KEY_2 as the *ShiftRight* input and KEY_3 as the *ASR* input. Use KEY_0 as the clock (clk). The outputs Q_{7-0} should be displayed on $LEDR_{7-0}$. **(PRELAB)**

5. Compile your Verilog code and simulate the design with ModelSim. In your simulation, you should perform the reset operation on the first clock cycle, then do a parallel load of your register on the next cycle. Finally, clock the register for several cycles to demonstrate both types of shifts. *(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?)* Include one (or a few) screenshot of simulation output in your prelab. **(PRELAB)**

6. Create a new Quartus Prime project for your circuit. Make sure it is stored in your `W:\` drive. And do not forget to select the correct FPGA device (5CSEMA5F31C6), as well as importing the pin assignments.

7. Compile the project and download your circuit on the DE1-SoC board. Then test the functionality of your shift register and show it to your TAs when you finished testing. **(IN-LAB)**