CSCB58 - Lab 3

Latches, Flip-flops, and Registers

Learning Objectives

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

Prelab	/3
Part I (in-lab)	/2
Part II (in-lab)	/2
Clean work-space with all materials returned to their original state	/1
TOTAL	/8

Write your name, UTorID, and student ID:

Name: ____

Student ID: ____

UTorID: ____

Write your partner's name, UTorID, and student ID:

Partner name: ______________________________________

Partner student ID: __________________________________

Partner UTorID: _____________________________________

Note: 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. In Verilog, when we use `always @(*)` to create combinational logic, we sometimes create latches by mistake. These latches are created when the output of your always block does not get assigned a value.

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** get a new value when x is 0. In this instance a latch would be created. To "fix" this latch, w would need a default value. From now on, **you should check your compilation log in Quartus** and look out for warnings that latches have been created.

Part I

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*, i.e., clock transitions from 0 to 1. The Verilog code for a positive edge-triggered flip flop is shown below in Figure 1. This flip flop also has an active-low, synchronous reset, meaning that the reset only happens when `reset_n = 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_n == 1'b0)  /* when reset_n is 0 (note this is tested on
                                                every rising clock edge) */
        q <= 0;           // q is set to 0. Note these assignments use <=
    else                  // when reset_n is not 0
        q <= d;           // value of d passes through to output q
end
```

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

Starting with the circuit you built for Lab 2 Part III, build an ALU with the eight operations as shown in the pseudo-code in Figure 2. 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 3 shows the required connections.

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

always @(*) // declare always block
begin
  case (function) // start case statement
    0: $A + B$ using the adder from Lab 2 Part II
    1: $A + B$ using the Verilog '+' operator
    2: A XOR B in the lower four bits and A OR B in the upper four bits
    3: Output 1 (`8'b00000001`) if at least 1 of the 8 bits in the two inputs is 1 using a single OR operation
    4: Output 1 (`8'b00000001`) if all of the 8 bits in the two inputs are 1 using a single AND operation
    5: Left shift B by A bits using the Verliog left shift operator '<<'
    6: Right shift B by A bits (logical) using the Verliog right shift operator '>>'
    7: $A \times B$ using the Verilog multiplication operator '$\star$'
    default: ... // default case
  endcase
end

Figure 2: Pseudo-code for ALU.

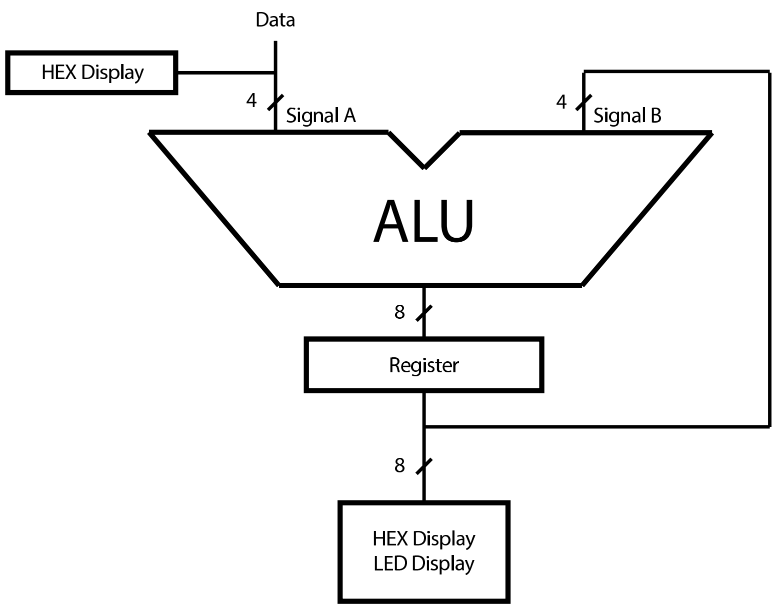

Figure 3: Simple ALU with register circuit for Part II.

Perform the following steps.

1. Create a Verilog module for the simple ALU with register. Use the code in Figure 1 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* in hexadecimal. *HEX4* and *HEX5* should display the least-significant and most-significant four bits of *Register* respectively, also in hexadecimal. **(PRELAB)**
2. Create a new Quartus II project for your circuit. Make sure to select the correct chip and import the pin assignments.
3. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit and show the TA.

Note: some boards have faulty buttons (for example, KEY_0 might constantly flip between 0 and 1 when pressed). If that happens, try to use other keys, or use a switch instead of a key.

Part II

In this part of the lab, you will create an 8-bit right-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 4 shows one part (one bit) of our right-shift register. It contains a positive edge-triggered flip-flop and several multiplexers. To accommodate 8-bits, you will use eight instances of the circuit in Figure 4 to design you right-shift register with optional arithmetic shift and parallel load shown in Figure 5.

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 arithmetic shift right, instead of loading a zero to the left-most bit, we replicate its old value: if it was 1, we load a 1. If it was 0, we load a 0. (The "regular" right shift where we load a zero is called *logical shift right*.)

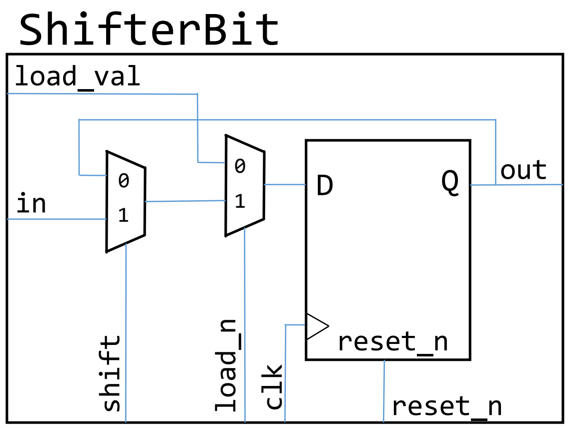

Figure 4: Sub-circuit for Part III.

In the Shifter module, create an 8-bit-wide register input `LoadVal`, whose individual wires (bits) are tied to each `load_val` input of each ShifterBit instance. Likewise, create an 8-bit-wide output `Q`, whose individual wires stem from each *out* port of each ShifterBit instance. The `shift` input of all eight instances of the circuit in Figure 4 should be tied to the single input `ShiftRight`. The `load_n` input of all eight instances should be tied to the input `Load_n`. This allows an 8-bit value to be loaded into all eight flip-flops on the same clock cycle. The `clk` input of all eight instances should be tied to the single input `clk`. Likewise for `reset_n`. The `in` input of all eight instances, should be connected to the `out` port of the instance to its left, because when you want to shift the bits right, you have to load the bit to the left - except for the leftmost ShifterBit instance. In this special case, you should design a circuit that will perform sign-extension when the signal `ASR` is high (arithmetic right shift) and will load zeros if `ASR` is low (logic right shift).

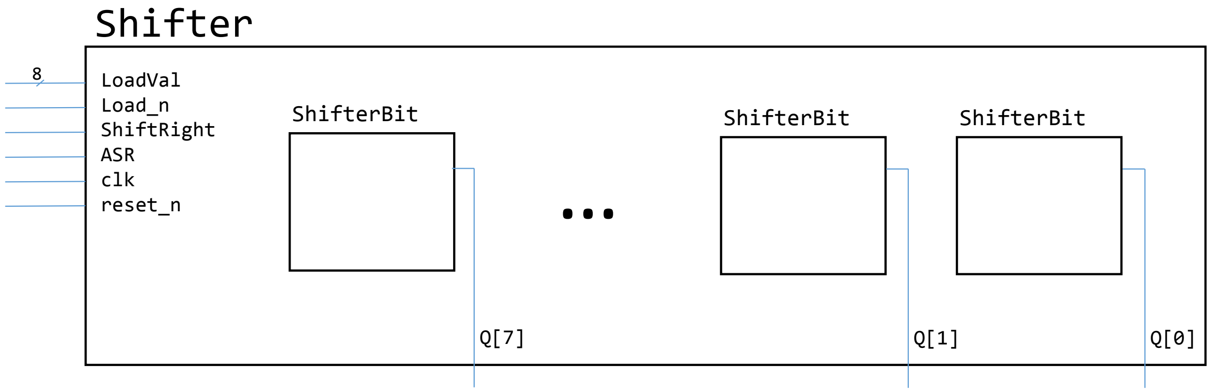

Figure 5: Shifter circuit for Part III. All required internal connections are **not** shown.

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:

	Bit_7	Bit_6	Bit_5	Bit_4	Bit_3	Bit_2	Bit_1	Bit_0
Cycle 0:	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2	Q_1	Q_0
Cycle 1:	0	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2	Q_1
Cycle 2:	0	0	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2
Cycle 3:	0	0	0	Q_7	Q_6	Q_5	Q_4	Q_3
...								

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*:

	Bit_7	Bit_6	Bit_5	Bit_4	Bit_3	Bit_2	Bit_1	Bit_0
Cycle 0:	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2	Q_1	Q_0
Cycle 1:	Q_7	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2	Q_1
Cycle 2:	Q_7	Q_7	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2
Cycle 3:	Q_7	Q_7	Q_7	Q_7	Q_6	Q_5	Q_4	Q_3
...								

Do the following steps:

1. What is the behaviour of the 8-bit shift register shown in Figure 5 when `Load_n` = 1 and `ShiftRight` = 0? Briefly explain. **(PRELAB)**
2. Draw a schematic for the 8-bit shift register shown in Figure 5 including the necessary connections. Your schematic should contain eight instances of the sub-circuit in Figure 4 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 1 for a flip flop, use this D flip flop with instances of the *mux2to1* module from Lab 1 to build the ShifterBit sub-circuit shown in Figure 4. To get you started, Figure 6 is a sample of hierarchical code showing the D flip flop with one of the 2-to-1 multiplexers connected to it. **(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 6: Part of the code for the sub-circuit in Figure 4.

4. Create a new Quartus II project and write a Verilog module for the shift register that instantiates eight instances of your Verilog module for Figure 4. This Verilog module should match with the schematic in your lab book. Use SW_{7-0} as the inputs `LoadVal`$_{7-0}$, and SW_9 as a synchronous active low reset. 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, but **read the important note below about switch bouncing.** The outputs Q_{7-0} should be displayed on the LEDs ($LEDR_{7-0}$).
5. Download your circuit on the DE2 board, and demonstrate its functionality to your TA.

Note: You may run into bounce problems using KEY_0 for your clock, (you are welcome to try using any of the keys if you find they preform better). 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.