Lab 9

ALU with Input Register

Updated: March 30, 2020. Version: Vivado 2018.3

9.1 Introduction

In previous labs, you have developed modules to do math operations and display numbers. The eventual goal is to build a small calculator. In this lab, you will take the next step by creating an Arithmetic Logic Unit (ALU) capable of a few operations. In order to do mathematical operations with the ALU, we need two numbers. One of these will come from the switches. For the other, we will use a use a register to store a number.

9.2 Objectives

After completing this lab, you should be able to:

- Explain the difference between combinational and regular sequential logic
- Describe the operation of an SR latch, D latch, D flip-flop, and D register
- Describe the differences in Verilog procedural blocks for combinational versus sequential logic
- Import and modify modules from a previous project, and use them to design a modular system

9.3 Pre-Lab

1. Read the following pages:
   - SR-Latch, D-Latch
   - Flip-Flops
   - (Optional - provides high-level explanation) Introduction to Memory Circuits
2. Answer the following questions:
   (a) What inputs do all useful memory devices have?
   (b) What is the difference between a D latch and a D flip-flop?
3. Suppose you test the three memory units described in the links above with the waveforms shown in Figure 9.1. Print this page or re-draw the given signals on your own paper and draw the outputs (Q and QN). Use the NAND-based behavior for the SR latch.

9.4 Procedure

Now that we are getting into larger systems with more modules, it's crucial that you are able to visualize each module and the overall system in your mind. Good engineers are able to think both "top-down" and "bottom-up." We need the top-down approach to understand the system as a whole and what are the various pieces, but often we design from the bottom-up because that's how we get modular systems. Each sub-component can be designed and tested as its own little system, then incorporated into the next layer up. Figure 9.2 is a pictorial description of our overall (top-level) system. We'll design and test each piece, then combine them together in a top-level file.

9.4.1 Create and test 8-bit register

Create a new Verilog file named `register`. Complete the code given in Listing 9.1 to implement a D register with synchronous enable and asynchronous reset. Note that the parameter N specified the number of bits of the data input and output (D and Q).

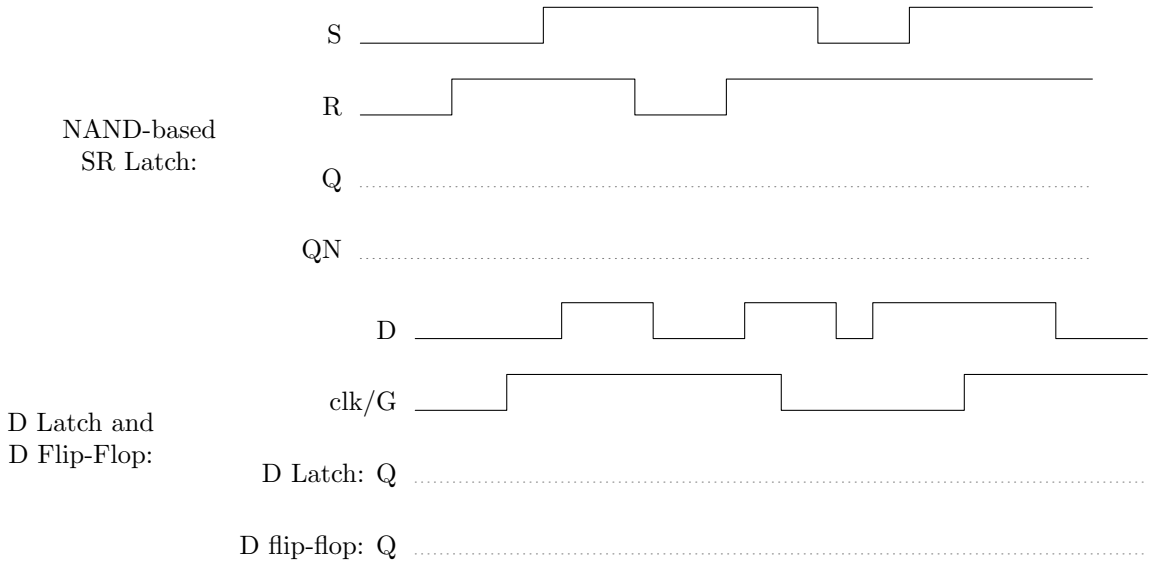

Figure 9.1: Test waveforms for pre-lab

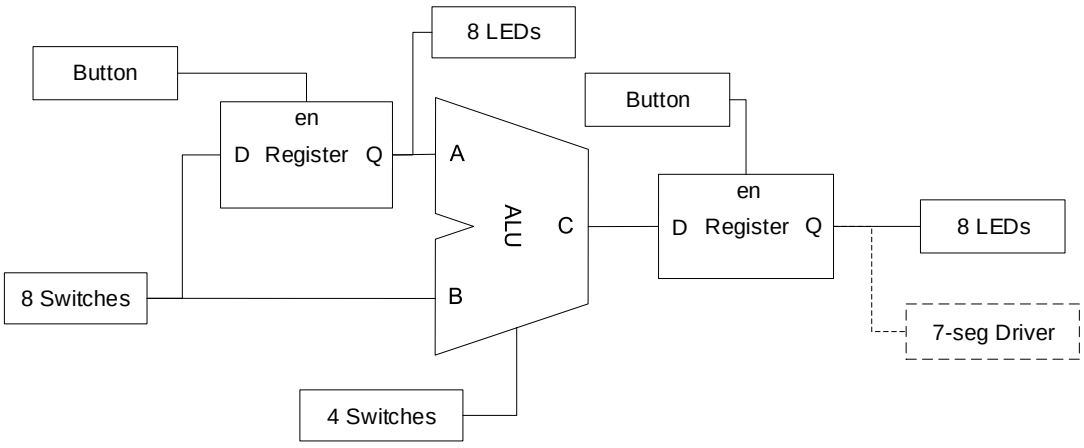

Figure 9.2: Conceptual block diagram showing registers, ALU, and output to 7-seg display

Listing 9.1: Register module

```
module register #(parameter N=1)
   (
   input clk, rst, en,
   input [N-1:0] D,
   output reg [N-1:0] Q
   );

   always @(posedge clk, posedge rst)
   begin
      if (rst==1)
         Q <= /*???*/ ;
      else if (en==1)
         Q <= /*???*/ ;
   end

// Notes:
// - Reset is asynchronous, so this
//   block needs to execute when rst
//   goes high.
// - We want enable to be synchronous
//   (i.e. only happens on rising
//   edge of clk), so it is left out
//   of "sensitivity" list.

endmodule
```

Create a new test bench for your register. Because the register has a clock input, we now need to add a clock to the test bench. This is actually easy. The `always` block in Listing 9.2 runs continuously. Thus, every 5 ns, the *clk* signal gets inverted, creating a constant period (10 ns, or 100 MHz) clock signal. The `initial` block is only executed once. By putting `clk = 0` there, it means the *clk* signal will start at 0, then oscillate according to the `always` block.

Listing 9.2: Register test bench

```
module register_test();

   reg [3:0] D;
   reg clk, en, rst;
   wire [3:0] Q;

   register #(.N(4)) r(.D(D), .clk(clk),
        .en(en), .rst(rst), .Q(Q) );

   // clock runs continuously
   always begin
      clk = ~clk; #5;
   end

   // this block only runs once
   initial begin
      clk=0; en=0; rst=0; D=4'h0; #7;
      rst = 1; #3;    // reset
      D = 4'hA; en = 1; rst = 0; #10;
      D = 4'h3;    #2;
      en = 0;      #5;
      en = 1;      #3;
      D = 4'h0;    #2;
      en = 0;      #10;
      en = 1;      #2;
      D = 4'h6;    #11;
      $finish;
   end
endmodule
```

Table 9.1 provides the sequence of test cases specified in Listing 9.2. "0→1" represents a rising edge (transition from 0 to 1), while "1→0" represents a falling edge. Using your understanding of how a register should work, fill in the rest of the output (Q) row. Then copy Listing 9.2 into your test bench and run a simulation. Compare your expected results with your simulation. If they don't match, figure out why and/or ask your instructor.

9.4.2 Create and test 8-bit ALU

Previously, you created an adder by explicitly defining a gate-level circuit (i.e. using AND, XOR, etc.) that implemented the desired logic function (adding two, 2-bit numbers with a carry). You then connected two of these in a "ripple adder" configuration. This works, but a) the timing is rather slow because of the ripple effect and b) it is not the most efficient way to write HDL code. HDL is designed so that you can specify the function that you want a component to perform and let the synthesis tools (within Vivado) determine how to implement that function at the gate level.

It's also worth noting that while we will be dealing with sequential circuits and timing in this lab, the ALU is a combinational circuit. This means it is constantly performing the specified calculation and producing an output.

Create a new Verilog file and name it `alu`. Some code is provided for you in Listing 9.3. *in0* and *in1* are the operands (numbers that you want to operate on). *op* chooses between the different operations that the ALU can perform.

Table 9.1: *register* expected results table

Time (ns):	0-5	5-10	10-15	15-20	20-25	25-30	30-35	35-40	40-45	45-50	50-55
D (hex)	0	0	A	A	3	3	0	0	0→6	6	6
clk	0	1	0	1	0	1	0	1	0	1	0
en	0	0	1	1	1→0	0→1	1→0	0	0→1	1	1
rst	0	0→1	0	0	0	0	0	0	0	0	0
Q (hex)	X	X→0	?...								

Table 9.2: *alu* expected results table skeleton

Time (ns):	0-10	10-20	20-30	30-40	40-50	50-60
in0						
in1						
op						
out						

Listing 9.3: ALU module

```
module alu  #(parameter N=8)
   (
   output reg[N-1:0] out,
   input [N-1:0] in0,
   input [N-1:0] in1,
   input [3:0] op
   );

   // Local parameters
   parameter ADD=0;
   parameter SUB=1;
   parameter AND=2;
   parameter OR=3;
   parameter XOR=4;

   always @*
   begin
      case(op)
         ADD: out = in0 + in1;
         // add the remaining commands
         default: out = in0;
      endcase
   end

endmodule
```

Verilog actually allows you to write statements such as "out = in0 + in1" to implement addition of multi-bit (or bus) signals. Hopefully, you realize how much easier this is than writing your own gate-level design from scratch! That's the beauty of HDL. The synthesis tools will take this behavioral description and turn it into a logic circuit on the FPGA. (The downside is that it may not operate as fast or efficient as doing your own design. Thus, when working under strict timing constraints, designers are sometimes forced to optimize lower-level circuits.)

Create a test bench for your ALU. Choose values for *in0* and *in1* that provide unique results for each of the operations (e.g. OR and XOR yield the same result for some bit combinations). Have the *op* signal move through all five operations to show all of them producing the correct results. As with the register, fill out an expected results table and compare it with your simulation. Your table might look something like Table 9.2. Verify that your ALU is operating correctly. Check with your instructor if you are unsure. (Remember: You can right-click and change the radix to hexadecimal.)

9.4.3 Top-level file

Create a new Verilog file named `top_lab9`. It is always helpful to draw a picture, so the schematic is provided in Figure 9.3. Define all of the inputs and outputs shown, instantiate the necessary modules and connect them as shown.

If you have a Basys3 board available, use the On-Board Testing procedure. If not, use the Top-Level Simulation procedure.

9.4.4 On-Board Testing

In addition to the `switches.xdc` constraint file used in previous labs, you will need the `btnC.xdc`, `btnD.xdc`, `btnU.xdc`, `clock.xdc` and `led.xdc` constraint files that are also in Support Docs in Canvas.

Implement your design on the Basys3 board and verify its operation using the Operation list. Take a picture of the board at each step, ensuring that both the LEDs and the switches are shown.

Operation

Here is how your board should operate:

The ALU input register is connected to LEDs 7-0. Set switches 7-0 to hexadecimal 14 and then press the Down button. The LEDs should light up to match your switches.

The ALU output register is connected to LEDs 15-8. After doing the above, turn off switches 7-0 and then press the Up button. The LEDs 15-8 should now also display hexadecimal 14.

You actually used the ALU without realizing it. You had the op switches (11-8) set to 0, which is Add. You added the value you to stored (Up button) with zero (all switches down).

Pressing btnC clears the LEDs.

Now again store hexadecimal 14 in the input register using the Down button, then set switches 7-0 to hexadecimal 7A and press the Up button. The value shown by LEDs 15-8 should be the sum of the value stored in the register and the value on the switches.

If that works, then try the other operations (switches 11-8).

9.4.5 Top-Level Simulation

The SystemVerilog file `basys3_lab9.sv` is provided in the Lab 9 folder in Canvas. It is a top-level simulation source file that will instantiate and simulate your `top_lab9` module. Add `basys3_lab9.sv` to your simulation source files, set it as you simulation top level, and run the simulation. Compare your simulation waveforms to the Operation list in the On-Board Testing procedure to verify proper operation of your design. Note that to make the simulation run properly, an extra step is included as the first step, pressing btnC, that is not included in the Operation list.

9.5 Deliverables

Submit a report containing the following:

1. Two expected results tables (register and ALU)
2. Two simulation waveforms (register and ALU)
3. Picture of Basys3 board for each step in Operation list for On-Board Testing only
4. Simulation waveforms for `basys3_lab9` for Top-Level Simulation only
5. Verilog source files for register, ALU and top_lab9
6. Verilog test bench files for register and ALU

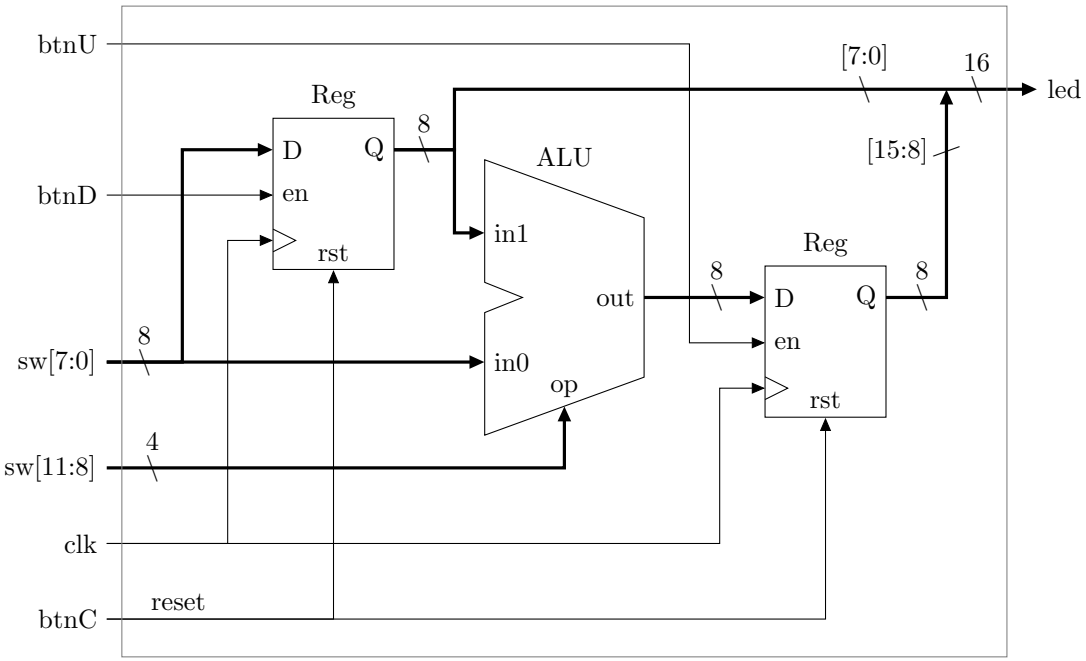

Figure 9.3: Schematic for top level