Cover Page
CPE 324-02: Advanced Logic Design Laboratory

Lab 3

Basic Arithmetic Logic Unit

Submitted by: Nolan Anderson

Date of Experiment: 02/23/21

Report Deadline: 02/23/21

1. Introduction

1.1 - What is to be studied, what is the purpose, and how is this purpose accomplished?

Lab 3 introduces an ALU implementation, basic debounce code, and interfacing the 7-segment display on the DE10 Lite board. To do this, we will first introduce how to interface and initialize the 7-segment display, see section 2.1.1. Continually in this section, we will show how to select the correct output for the display. Section 2.1.2 covers the debouncing portion of the code. Lastly, section 2.1.3 will describe the implementation of the ALU implemented for this project. Overall, since most of the code was provided, the general purpose of this lab is centered on interfacing with the 7-segment display. The implementation of these can be found in the following section, 2.2.

2. Experiment Description

2.1 Theory, analysis, and purpose

2.1.1 The 7-segment Display

The DE10 Lite 7-Segment display is very straightforward, but accuracy is very important. As seen in figure one, we can select specific bits (0 is on, 1 is off) to display our output. We need to provide the pins with a hexadecimal value for each character to output our value. For instance, if we wanted to display r, that would be 1010 1111 in binary, and AF in hexadecimal. Providing the input array with AF will leave only bar 4 and 6 on, and the rest off. The implementation for this can be seen in section 2.2.1.

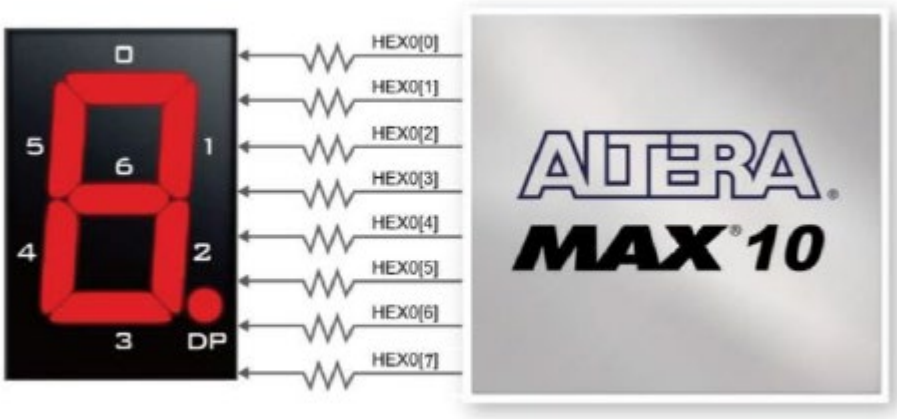


Figure 1: 7-Segment display, inputs, and corresponding output.

2.1.2 Debouncing

Adding debouncing to code is very important when implementing buttons and or switches. If you neglect to add debouncing, there is a high possibility you will receive noise when interfacing with the switches. This noise comes from the input taking its time to switch from low to high or high to low, see figure 2. This noise can severely impact the functionality of

your program. There are multiple ways of implementing this debouncing, but the most common one is to introduce some delay after a physical change so that the signal has time to settle into its true state. This is very important for our code as there are many button presses and switches, and without debouncing it could get confusing. The implementation of this can be seen in section 2.2.2.

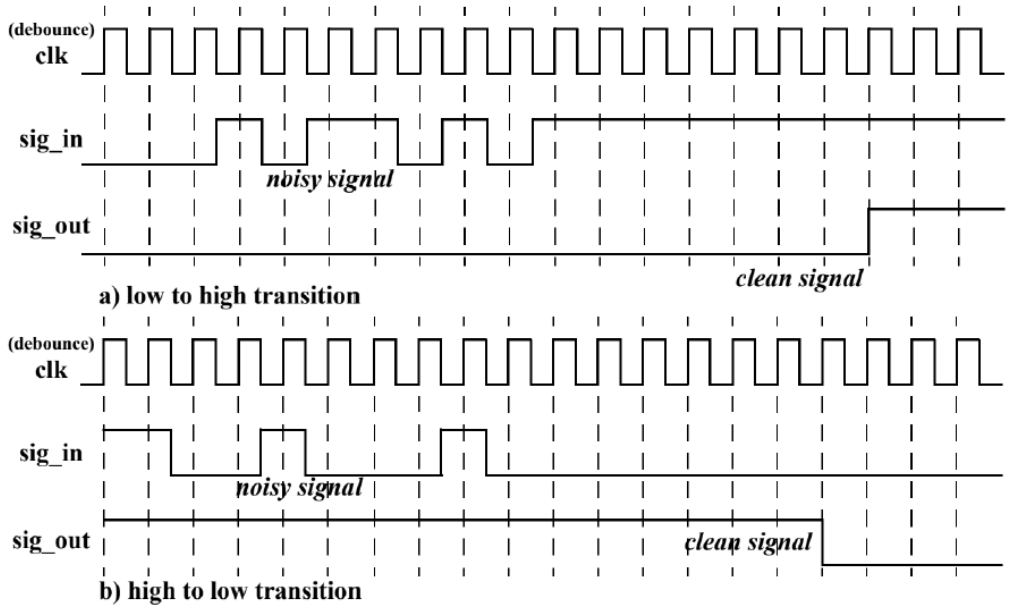


Figure 2: Debouncing importance, shown in waveforms.

2.1.3 Arithmetic Logic Unit

Arithmetic logic units (ALU) perform operations on differing input values and output a result of said operation. In this case, we are using Verilog to implement our ALU and consists of 8 total functions, see section 2.2.3. In our case, the ALU is called by our top-level file DE10_LITE_Goldon_Top.v and our inputs come from the switches selected and the operation register. A basic ALU can be seen in figure 3.

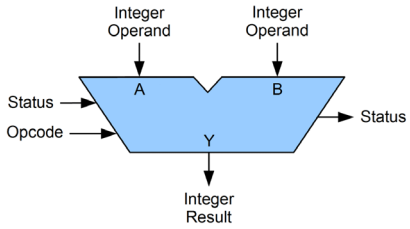


Figure 3: Basic ALU diagram

2.1.5 Analysis and Purpose

The purpose of this lab is to teach us about the different states of a FSM, ALU's, Debouncing, and the 7-segment display on the DE10 Lite. Continually, we need to be able to develop all of the items previously mentioned in Verilog.

2.2 Design and implementation procedure

2.2.1 7-Segment display implementation

This section will briefly describe the implementation of the 7-segment display values. Figure 4 provides the table as shown in the lab assignment. Figure 5 implements this graph using a case statement with InVal acting as the argument. 5'd0 (case 0) represents the input state, and 8'hc0 (output 0) shows the output state.

InVal[4:0]	Out Char	OutVal[7:0]	InVal[4:0]	Out Char	OutVal[7:0]
5'd0	0	8'hC0	**5'd10**	A	8'hC8
5'd1	1	8'hF9	**5'd11**	b	8'h83
5'd2	2	8'hA4	**5'd12**	C	8'hC6
5'd3	3	8'hB0	**5'd13**	d	8'hA1
5'd4	4	8'h99	**5'd14**	E	8'h86
5'd5	5	8'h92	**5'd15**	F	8'h8E
5'd6	6	8'h82	**5'd16**	r	???
5'd7	7	8'hF8	**5'd17**	g	???
5'd8	8	8'h80	**5'd18**	o	???
5'd9	9	8'h90	**5'd19-5'd31**	(blank)	8'hFF

Figure 4: Input / Output values for 7-Segment Display.

```
1  module  hex_driver (
2    input       [4:0] InVal,
3    output reg [7:0] OutVal
4  );
5
6  always @(InVal)
7  begin
8    case (InVal)
9      5'd0 : OutVal = 8'hC0;   // 0
10     5'd1 : OutVal = 8'hF9;   // 1
11     5'd2 : OutVal = 8'hA4;   // 2
12     5'd3 : OutVal = 8'hB0;   // 3
13     5'd4 : OutVal = 8'h99;   // 4
14     5'd5 : OutVal = 8'h92;   // 5
15     5'd6 : OutVal = 8'h82;   // 6
16     5'd7 : OutVal = 8'hF8;   // 7
17     5'd8 : OutVal = 8'h80;   // 8
18     5'd9 : OutVal = 8'h90;   // 9
19     5'd10 : OutVal = 8'h88;  // A
20     5'd11 : OutVal = 8'h83;  // b
21     5'd12 : OutVal = 8'hC6;  // C
22     5'd13 : OutVal = 8'hA1;  // d
23     5'd14 : OutVal = 8'h86;  // E
24     5'd15 : OutVal = 8'h8E;  // F
25     5'd16 : OutVal = 8'hAF;  // r
26     5'd17 : OutVal = 8'h90;  // g
27     5'd18 : OutVal = 8'hA3;  // o
28     default : OutVal = 8'hFF; // blank
29   endcase
30 end
31
32 endmodule
33
34
```

Figure 5: Figure 4 implemented in Verilog

2.2.2 Debounce implementation

As mentioned in section 2.1.2, debouncing is critically important for our code and this section will cover the implementation for this project. This debounce implementation initially sets the counter to a hex value of 0 and 3 if statements follow. If the counter is equal to the local parameter CNT, the state essentially resets. If the sig_in is equal to the state, we compare the counter to 0. The last statement is where the actual debouncing happens. We add 1 to the counter until it hits the CNT and then the signal is sent, see line 37.

```
module debounce #(
  parameter [15:0] DWELL_CNT
 ) (
  input   clk,
  input   sig_in,
  output  sig_out
);

    reg state;
    reg [15:0] counter;
    localparam[15:0] CNT = DWELL_CNT - 1;


 initial
begin
    state <= sig_in;
    counter <= 'h0;      // Initialize counter to 0
 end

 always @(posedge clk)
begin

    if((counter & CNT) == CNT) begin
       counter <= 'h0;
       state <= sig_in;
    end

    else if(sig_in == state) begin
       counter <= 'h0;
    end

    else begin
       counter <= counter + 'h1;
    end

 end
 assign sig_out = state;
 endmodule

```

Figure 6: Debouncing implemented

2.2.3 ALU implementation

The ALU portion of this assignment consists of a case statement in the ALU Verilog file. The case statement depends on the selected opcode, and the opResult operates on operandA (the Operation Register) and operandB (the selected switches). The output of the ALU can be seen without pressing any additional buttons on the device. Figure 7 shows the code implemented. The ALU is called directly from the DE10_Lite_Golden_Top Verilog file and is the main operation of the code.

```
module alu (
   input    clk,
   input  [7:0]  operandA,
   input  [7:0]  operandB,
   input  [2:0]  opCode,
   output reg [15:0] opResult
);

   always @(posedge clk)
   begin
   case (opCode[2:0])
    3'd0  : opResult = operandA + operandB;       // OPCODE 0, ADD.
    3'd1  : opResult = operandA - operandB;       // OPCODE 1, SUB.
    3'd2  : opResult = operandA ^ operandB;       // OPCODE 2, XOR.
    3'd3  : opResult = operandA & operandB;       // OPCODE 3, AND.
    3'd4  : opResult = operandA | operandB;       // OPCODE 4, OR.
    3'd5  : opResult = operandA * operandB;       // OPCODE 5, MUL.
    3'd6  : opResult = operandA << operandB;      // OPCODE 6, SL.
    3'd7  : opResult = operandA >> operandB;      // OPCODE 7, SR.
    default : opResult = operandA + operandB;     // Defualt case.
   endcase
   end
 endmodule


```

Figure 7: ALU Verilog implementation

2.2.4 Theoretical and Simulated values

In this section, I will show theoretical values for the 8 different operations shown in Figure 7 and provide the output I received from the board. Figure 8 shows this in detail:

Question	A (right)	B (left)	Theoretical	Simulated
1 (ADD)	0000 0000 [0 0]	0000 1111 [0 F]	0000 1111 [0 F]	0000 1111 [0 F]
2 (SUB)	0000 0011 [0 3]	0000 0001 [0 1]	0000 0010 [0 2]	0000 0010 [0 2]
3 (XOR)	1111 0000 [F 0]	0000 0000 [0 0]	1111 0000 [F 0]	1111 0000 [F 0]
4 (AND)	0000 0000 [0 0]	1111 1111 [F F]	0000 0000 [0 0]	0000 0000 [0 0]
5 (OR)	1111 1111 [F F]	0000 0000 [0 0]	1111 1111 [F F]	1111 1111 [F F]
6 (MUL)	0000 0011 [0 3]	0000 0101 [0 5]	0000 1111 [0 F]	0000 1111 [0 F]
7 (SL)	0000 1111 [0 F]	0000 0010 [0 2]	0011 1100 [3 C]	0011 1100 [3 C]
8 (SR)	0000 1111 [0 F]	0000 0010 [0 2]	0000 0011 [0 3]	0000 0011 [0 3]

Figure 8: Simulated and theoretical Inputs and Outputs

3. Demonstration

3.1 Video Link:

Folder:
https://drive.google.com/drive/folders/16QoZFWimvjmE4P-8_F79kciyIY5xSQq?usp=sharing
Video:
https://drive.google.com/file/d/183duoTTSMuOzvEW3QTE4-nE9HJzIn2xA/view?usp=sharing
Also provided in the google drive file you provided us.

4. Experimental Results

4.1 Observations:

Observing the actual operations is not all that important as they are built in to Verilog (see section 2.2.3). However, when observing the 7-segment display we can see that the values implemented in section figure 5 provide the correct values. If figure 5 had incorrect values, the math behind the "screen" would be correct but our output would be incorrect. Next, we can see that the debouncing implementation works as expected and we do not receive any erroneous signals or errors when interfacing with the buttons and switches. Lastly, the ALU works as expected. The results are consistent to the opcode provided, and the outputs are consistent to theoretical values.

4.2 Post lab questions:

4.2.1 Using the Quartus GUI, report the following results after finishing all the steps:

a. FPGA component utilization (use the Flow Summary tab):
i. How many "Total logic elements" are in use? These are the FPGA's lookup tables (LUTs)
421 / 49,760
ii. How many "Total registers" are in use? These are the FPGA's sequential elements

138

iii. How many pins are in use?

97 / 360

iv. Did Quartus use a dedicated multiplier in the ALU (for OPCODE 5)? How can you tell?

Yes. If you look at the compilation report it shows that the "Embedded Multiplier 9-bit elements is 1/288."

v. Assuming we don't need to add more pins, how many copies of this design would fit in this device? (Hint: just look at the total number of Logic Elements listed in your device)

Just 3 copies, 97 * 3 = 291.

b. FPGA timing closure

i. Hopefully this design was able to meet timing at a 50 MHz system clock. What does the TimeQuest tool say was the fastest it can be clocked? To check this, use the Slow 1200 mV 85C model, and it can be seen in the Quartus Table of Contents or alternatively in the TimeQuest Tool (Tools > TimeQuest Timing Analyzer), and try to get the Fmax summary.

139.02MHz.

ii. Is Fmax better or worse for Slow 1200 mV 0C (here the only difference is the simulated temperature of operation)?

When switching to 0C, the Fmax = 153.78 MHz, whereas 85C the Fmax operates at 139.02MHz. Fmax seems to run quacking under the 0C model.

iii. What's the critical path? This requires running TimeQuest Timing Analyzer, selecting Report Setup Summary. Right-click the clock name and hit "Report Timing". Then accept all the defaults and run. The 10 worst-case timing paths are reported.

	Property	Value
1	From Node	display_driver:hex_leds|decState[0]
2	To Node	display_driver:hex_leds|decDigits[3][3]
3	Launch Clock	MAX10_CLK1_50
4	Latch Clock	MAX10_CLK1_50
5	Data Arrival Time	10.388
6	Data Required Time	23.195
7	Slack	12.807

← Critical path

1. The worst-case slack (which can be negative, indicating failed timing!) is listed first.

12.807

2. List the source and destination node and the reported slack.

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	12.807	display_driver:hex_leds|decState[0]	display_driver:hex_leds|decDigits[3][3]	MAX10_CLK1_50	MAX10_CLK1_50	20.000	-0.061	7.124

3. Go into the Verilog code and see if you can locate the source and destination nodes and trace the path the signal takes. (TimeQuest also shows this progression in the "Data Arrival Path".) Comment on what part of the circuit this… does it make sense that's the worst-case path?

The following line comes up multiple times when looking at the longest slack:

decDigits[decState] <= decDigits[decState]+4'd1; // ...and add it to the decimal digit that we're building

When reaching the upper end of the data, such as FF, the code will naturally slow down as it has to compensate to allow for the larger numbers. This is part of the last if else statement in the always @(posedge clk), which means that this is the absolute last thing that gets checked by the clock statement. Overall, it makes sense that this is the worst-case path as it is one of the if not the last case the code checks for when running the clock sequence.

4.2.2 Take a look at the display_driver.v code (provided for the student in this assignment). There are two finite state machines (FSMs) in the always @(posedge clk) process. Both of these FSMs use a major state signal (displayState, decState), but also have sub-registers that must meet conditions for the major state variables to change.

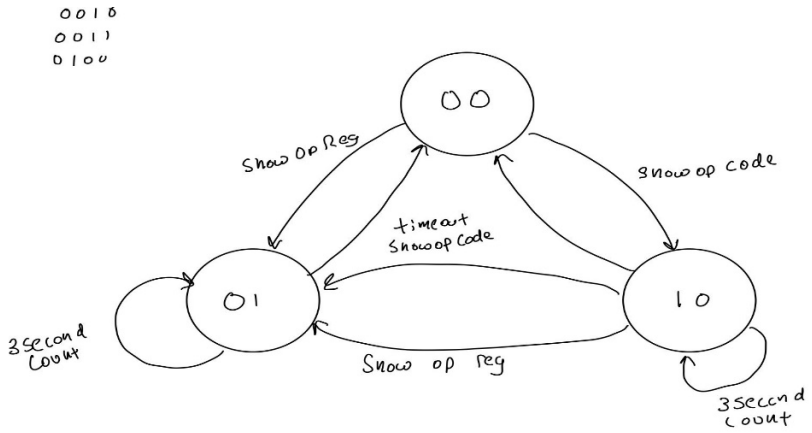
a. Pertaining to the FSM that involves the displayState[1:0] signal… how many states exist altogether in this FSM? Ignore states that are not reached through normal signal value propagation (i.e. don't consider displayState[1:0]==2'd3 because it is not a valid destination state). Hint – it is finite, but the number of states is much larger than 3.

6007

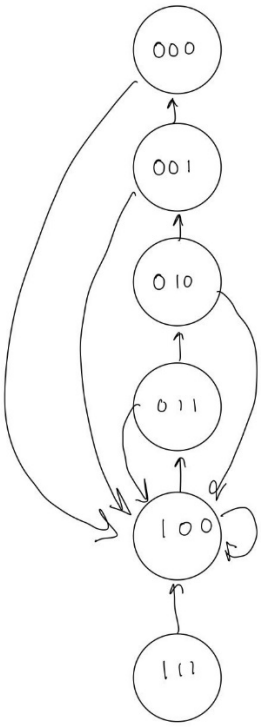
b. Pertaining to the FSM that involves the decState[2:0] signal… how many states can the signal decState occupy (again, assuming normal/legal state transitions)? How many states can decDigits[4] occupy? How many for

decDigits[4] – 2 – depends on 111 and 100.
decDigits[3] – 2 – depends on 111 and 011.
decDigits[2] – 2 - depends on 111 and 010.
decDigits[1] – 2 - depends on 111 and 001.
decDigits[0] – 2 - depends on 111 and 000.
decState – 6 states.

c. Draw state graphs for these that only concern the major state signals (displayState and decState). Instead of drawing in discrete inputs, put into words the conditions that cause the signals to change state on the graph



Part a:



Part b:

d. Are these Mealy or Moore state machines?

Mealy machines depend on the present state as well as the present input. Moore only depends on the present state. These are mealy machines. If you look at part a, the output depends on the current state (3 second countdown)_as well as the input (the show opreg / show opcode). The second FSM is a moore machine, however. It only depends on the current input.

4.2.3 For the ALU, is the multiplier operation signed or unsigned? If you use "reg signed" or "reg unsigned" for the result output and for the multiplier inputs, does Quartus change the output behavior from simply using "reg"?

The multiplication operand (*) in Verilog performs unsigned arithmetic. If you were to mix signed and unsigned numbers, it would default to an unsigned nature. So technically yes, it would change the output behavior because your result would be unsigned no matter what.

4.2.4 Comment about the benefit of creating the ***hex_driver*** module and replicating it 6 times instead of writing the same code 6 times within the ***display_driver*** module.

Beyond having less lines of code, replicating hex_driver frees up space and efficiency. If you were to throw hex_driver into display_driver 6 times you would most likely end up using more resources and pins. Continually, you would have to go to each separate replication of hex_driver each time you were to call to an input. This would take a lot more time and could introduce some unwanted delay into your program. Replicating hex_driver is just better overall as well. Each time you want to display a new character you just provide hex_driver with the input and it sends back an output. It's very similar to if you wrote C++ code without using functions. If you need code to do the same thing repeatedly, just send it to another function and let it perform those operations. Overall, writing hex_driver 6 times into display_driver can be summed up as poor programming practice.

5. Conclusions

5.1 - Results and lessons learned:

I learned a lot from this lab. Most importantly was understanding how to interface with the 7-segment display and case statements in Verilog. The code was straightforward to implement, however the theory and post-lab questions took some time and consideration. I would go as far to say that the post-lab questions are really an extension of the theory as there was not a whole lot to cover for this lab. A similar statement can be said about the code implementation. Overall, I would say that while the implementation and theory behind this lab was not all that complicated, the questions certainly made me think and using the board gave me a good review of shift left and shift right operations.

6. Appendix

6.1

I am not providing any detail for the appendix for this lab. The sections above cover the material well enough.