THE UNIVERSITY OF ALABAMA IN HUNTSVILLE
Department of Electrical and Computer Engineering

CPE 324 Advanced Logic Design Laboratory

Laboratory Assignment #3
Basic Arithmetic Logic Unit

(12% of Final Grade)

Purpose

The purpose of this laboratory is to give each student the opportunity to examine the lower-level issues involved with sequential logic in a synchronous circuit and with interfacing it to a mechanical device such as a simple button. The specific issues being exposed include the key bounce problem and the introduction of a system clock rate and its maximum achievable frequency. This laboratory experience also provides students with the opportunity to get accustomed to Verilog hardware description language design entry. Extensive example code is provided, yet the system will not function without additional coding by the student. It also requires that students be able to integrate their own design elements into a larger system that is composed of additional intellectual property modules.

Design Problem

The specific problem is to develop a sequential design that will perform a variety of arithmetic and logical operations on a pair of 8-bit inputs. This experiment will be carried out on the DE2-115 or DE10-Lite development board (according to the student's preference), and will feature the use of the switches (SW[9:0]), pushbuttons (KEY[1:0]), and 7-segment LED displays (HEX[5:0]). An operand is loaded into an 8-bit register OPREG by pressing the KEY[0] button. A 3-bit OPCODE value is selected by pressing the KEY[1] button; the opcode value increments once per button press. This will require debouncing to prevent a single button press from resulting in several (unpredictable) increments. When neither button is pressed, the values in SW[7:0] will form a second operand in conjunction with OPREG. The HEX LED display must show the hex value for OPREG for a duration of 3 seconds any time KEY[0] is pressed, it must show the opcode value for 3 seconds any time KEY[1] is pressed, otherwise it will show the 4-digit hexadecimal value of the ALU output.

The following table shows the required opcodes for the ALU:

Table 1: ALU Opcode Table

OPCODE	Operation
0: ADD	RESULT = OPREG + SW_IN
1: SUBTRACT	RESULT = OPREG – SW_IN
2: XOR	RESULT = OPREG ^ SW_IN
3: AND	RESULT = OPREG & SW_IN
4: OR	RESULT = OPREG | SW_IN
5: MULTIPLY	RESULT = OPREG & SW_IN
6: SHIFT LEFT	RESULT = OPREG << SW_IN
7: SHIFT RIGHT	RESULT = OPREG >> SW_IN

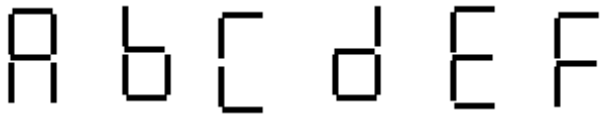
Background

The Arithmetic Logic Unit (ALU) is a key component within the execution unit of a CPU. The execution unit sends computational commands in the form of operation codes (opcodes) to the ALU

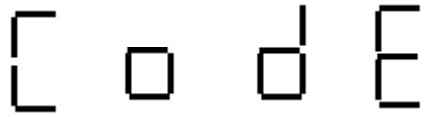
for processing. Other circuitry, outside the scope of this lab, in the execution unit is responsible for maintaining the program counter and executing Jump commands (and comparisons for Jump commands). The complexity of each listed OPCODE is not equivalent to each other; for example, an 8-bit bitwise OR operation (OPCODE==4) is much less complex than an 8-by-8 MULTIPLY operation (OPCODE==5) with its 15-bit output. The frequency at which the ALU can operate depends upon the amount of combinational logic of the most complex OPCODE, plus any multiplexing logic for selecting that particular result.

The momentary pushbuttons KEY[1] and KEY[0] are connected through a Schmidtt Trigger hysteresis to pullup resistors, and shunt to ground when pressed. The mechanical action of pushing each button typically creates a glitchy signal that experiences several edges on the signal when sampled with a clock above a few kHz. This suppression, also known as "debouncing" the button input, can be accomplished in an analog circuit (the Schmidtt Trigger is a well known example) or in a digital circuit. In this lab, we will AND the KEY[1] button with SW[9] to demonstrate debouncing with a digital circuit.

The 7-segment LED displays will be used for demonstrating functionality. Each is connected as 8 discrete LED signals (including the decimal point, which we will not use), and will demonstrate a useful level of abstraction between the 4-bit hex digit value and the 7-bit LED drive value. The desired LED configuration for the values from 4'hA through 4'hF are as follows:



Additionally, we will display rEg and CoDE momentarily during button presses on the leftmost LEDs, which require the addition of 3 codes ("r", "g", and "o") beyond the 16 hexadecimal values from 0-F:



According to the subsequent diagrams, the HEXN[7:0] signals are encoded as follows (0 per bit turns the LED on, 1 turns it off):

Table 2: Hex Digit Encodings

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

The values for r, g, and o are left for the student to fill in according to the following diagrams (remember, a 0 in the bit position corresponds with turning the LED on).

The final result of the operations may be optionally displayed as a 4-digit hexadecimal number or as a 5-digit decimal number, depending on the state of SW[8].

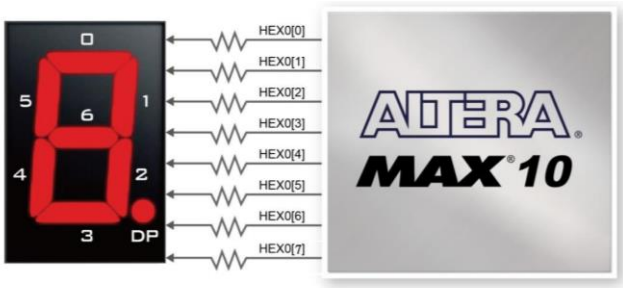


Figure 3-17 Connections between the 7-segment display HEX0 and the MAX 10 FPGA

Table 3-6 Pin Assignment of 7-segment Displays

Signal Name	FPGA Pin No.	Description	I/O Standard
HEX00	PIN_C14	Seven Segment Digit 0[0]	3.3-V LVTTL
HEX01	PIN_E15	Seven Segment Digit 0[1]	3.3-V LVTTL
HEX02	PIN_C15	Seven Segment Digit 0[2]	3.3-V LVTTL
HEX03	PIN_C16	Seven Segment Digit 0[3]	3.3-V LVTTL
HEX04	PIN_E16	Seven Segment Digit 0[4]	3.3-V LVTTL
HEX05	PIN_D17	Seven Segment Digit 0[5]	3.3-V LVTTL
HEX06	PIN_C17	Seven Segment Digit 0[6]	3.3-V LVTTL
HEX07	PIN_D15	Seven Segment Digit 0[7], DP	3.3-V LVTTL
HEX10	PIN_C18	Seven Segment Digit 1[0]	3.3-V LVTTL
HEX11	PIN_D18	Seven Segment Digit 1[1]	3.3-V LVTTL
HEX12	PIN_E18	Seven Segment Digit 1[2]	3.3-V LVTTL
HEX13	PIN_B16	Seven Segment Digit 1[3]	3.3-V LVTTL
HEX14	PIN_A17	Seven Segment Digit 1[4]	3.3-V LVTTL
HEX15	PIN_A18	Seven Segment Digit 1[5]	3.3-V LVTTL
HEX16	PIN_B17	Seven Segment Digit 1[6]	3.3-V LVTTL
HEX17	PIN_A16	Seven Segment Digit 1[7] , DP	3.3-V LVTTL
HEX20	PIN_B20	Seven Segment Digit 2[0]	3.3-V LVTTL
HEX21	PIN_A20	Seven Segment Digit 2[1]	3.3-V LVTTL
HEX22	PIN_B19	Seven Segment Digit 2[2]	3.3-V LVTTL
HEX23	PIN_A21	Seven Segment Digit 2[3]	3.3-V LVTTL
HEX24	PIN_B21	Seven Segment Digit 2[4]	3.3-V LVTTL
HEX25	PIN_C22	Seven Segment Digit 2[5]	3.3-V LVTTL
HEX26	PIN_B22	Seven Segment Digit 2[6]	3.3-V LVTTL
HEX27	PIN_A19	Seven Segment Digit 2[7] , DP	3.3-V LVTTL
HEX30	PIN_F21	Seven Segment Digit 3[0]	3.3-V LVTTL
HEX31	PIN_E22	Seven Segment Digit 3[1]	3.3-V LVTTL
HEX32	PIN_E21	Seven Segment Digit 3[2]	3.3-V LVTTL
HEX33	PIN_C19	Seven Segment Digit 3[3]	3.3-V LVTTL
HEX34	PIN_C20	Seven Segment Digit 3[4]	3.3-V LVTTL
HEX35	PIN_D19	Seven Segment Digit 3[5]	3.3-V LVTTL
HEX36	PIN_E17	Seven Segment Digit 3[6]	3.3-V LVTTL
HEX37	PIN_D22	Seven Segment Digit 3[7] , DP	3.3-V LVTTL
HEX40	PIN_F18	Seven Segment Digit 4[0]	3.3-V LVTTL
HEX41	PIN_E20	Seven Segment Digit 4[1]	3.3-V LVTTL
HEX42	PIN_E19	Seven Segment Digit 4[2]	3.3-V LVTTL
HEX43	PIN_J18	Seven Segment Digit 4[3]	3.3-V LVTTL
HEX44	PIN_H19	Seven Segment Digit 4[4]	3.3-V LVTTL
HEX45	PIN_F19	Seven Segment Digit 4[5]	3.3-V LVTTL
HEX46	PIN_F20	Seven Segment Digit 4[6]	3.3-V LVTTL
HEX47	PIN_F17	Seven Segment Digit 4[7] , DP	3.3-V LVTTL
HEX50	PIN_J20	Seven Segment Digit 5[0]	3.3-V LVTTL
HEX51	PIN_K20	Seven Segment Digit 5[1]	3.3-V LVTTL
HEX52	PIN_L18	Seven Segment Digit 5[2]	3.3-V LVTTL
HEX53	PIN_N18	Seven Segment Digit 5[3]	3.3-V LVTTL
HEX54	PIN_M20	Seven Segment Digit 5[4]	3.3-V LVTTL
HEX55	PIN_N19	Seven Segment Digit 5[5]	3.3-V LVTTL
HEX56	PIN_N20	Seven Segment Digit 5[6]	3.3-V LVTTL
HEX57	PIN_L19	Seven Segment Digit 5[7] , DP	3.3-V LVTTL

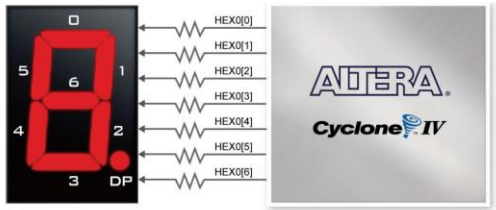


Figure 4-10 Connections between the 7-segment display HEX0 and Cyclone IV E FPGA

Table 4-4 Pin Assignments for 7-segment Displays

Signal Name	FPGA Pin No.	Description	I/O Standard
HEX0[0]	PIN_G18	Seven Segment Digit 0[0]	2.5V
HEX0[1]	PIN_F22	Seven Segment Digit 0[1]	2.5V
HEX0[2]	PIN_E17	Seven Segment Digit 0[2]	2.5V
HEX0[3]	PIN_L26	Seven Segment Digit 0[3]	
HEX0[4]	PIN_L25	Seven Segment Digit 0[4]	
HEX0[5]	PIN_J22	Seven Segment Digit 0[5]	
HEX0[6]	PIN_H22	Seven Segment Digit 0[6]	
HEX1[0]	PIN_M24	Seven Segment Digit 1[0]	
HEX1[1]	PIN_Y22	Seven Segment Digit 1[1]	
HEX1[2]	PIN_W21	Seven Segment Digit 1[2]	
HEX1[3]	PIN_W22	Seven Segment Digit 1[3]	
HEX1[4]	PIN_W25	Seven Segment Digit 1[4]	
HEX1[5]	PIN_U23	Seven Segment Digit 1[5]	
HEX1[6]	PIN_U24	Seven Segment Digit 1[6]	
HEX2[0]	PIN_AA25	Seven Segment Digit 2[0]	
HEX2[1]	PIN_AA26	Seven Segment Digit 2[1]	
HEX2[2]	PIN_Y25	Seven Segment Digit 2[2]	
HEX2[3]	PIN_W26	Seven Segment Digit 2[3]	
HEX2[4]	PIN_Y26	Seven Segment Digit 2[4]	
HEX2[5]	PIN_W27	Seven Segment Digit 2[5]	
HEX2[6]	PIN_W28	Seven Segment Digit 2[6]	
HEX3[0]	PIN_V21	Seven Segment Digit 3[0]	
HEX3[1]	PIN_U21	Seven Segment Digit 3[1]	
HEX3[2]	PIN_AB20	Seven Segment Digit 3[2]	
HEX3[3]	PIN_AA21	Seven Segment Digit 3[3]	
HEX3[4]	PIN_AD24	Seven Segment Digit 3[4]	
HEX3[5]	PIN_AF23	Seven Segment Digit 3[5]	
HEX3[6]	PIN_Y19	Seven Segment Digit 3[6]	
HEX4[0]	PIN_AB19	Seven Segment Digit 4[0]	
HEX4[1]	PIN_AA19	Seven Segment Digit 4[1]	
HEX4[2]	PIN_AG21	Seven Segment Digit 4[2]	
HEX4[3]	PIN_AH21	Seven Segment Digit 4[3]	
HEX4[4]	PIN_AE19	Seven Segment Digit 4[4]	
HEX4[5]	PIN_AF19	Seven Segment Digit 4[5]	
HEX4[6]	PIN_AE18	Seven Segment Digit 4[6]	
HEX5[0]	PIN_AD18	Seven Segment Digit 5[0]	
HEX5[1]	PIN_AC18	Seven Segment Digit 5[1]	
HEX5[2]	PIN_AB18	Seven Segment Digit 5[2]	
HEX5[3]	PIN_AH19	Seven Segment Digit 5[3]	
HEX5[4]	PIN_AG19	Seven Segment Digit 5[4]	
HEX5[5]	PIN_AF18	Seven Segment Digit 5[5]	
HEX5[6]	PIN_AH18	Seven Segment Digit 5[6]	
HEX6[0]	PIN_AA17	Seven Segment Digit 6[0]	
HEX6[1]	PIN_AB16	Seven Segment Digit 6[1]	
HEX6[2]	PIN_AA16	Seven Segment Digit 6[2]	
HEX6[3]	PIN_AB17	Seven Segment Digit 6[3]	
HEX6[4]	PIN_AB15	Seven Segment Digit 6[4]	
HEX6[5]	PIN_AA15	Seven Segment Digit 6[5]	

Note: The student in this lab is given some amount of Verilog code that has been developed and tested, then intentionally and selectively deleted. The code that remains is to be used as a starting point and shows a variety of different coding styles. **Please read through all the existing code and fill in all the sections with //TODO comments.**

Top Level Design

The top-level of the design is shown in Figure 1. It is composed of five main modules. These instance names are dbc0, dbc1, u_alu, u_timer, and hex_leds. These names are arbitrary – sometimes they are named similarly to the module's name, sometimes with a prefix or suffix. Again, this is intentionally inconsistent to show the student a variety of styles. The top-level file also includes some sequential logic and combinational logic in support of the lower-level modules. This comprises some asynchronous clock crossing registers (more about this in a future lecture), the OPCODE and OPREG registers that feed towards the ALU, and edge detectors on the output of the debouncer circuits.

The SW_in_meta (***meta***) module performs a simple double-register synchronizer to stabilize the SW[9:0] switches (more on this in future lectures).

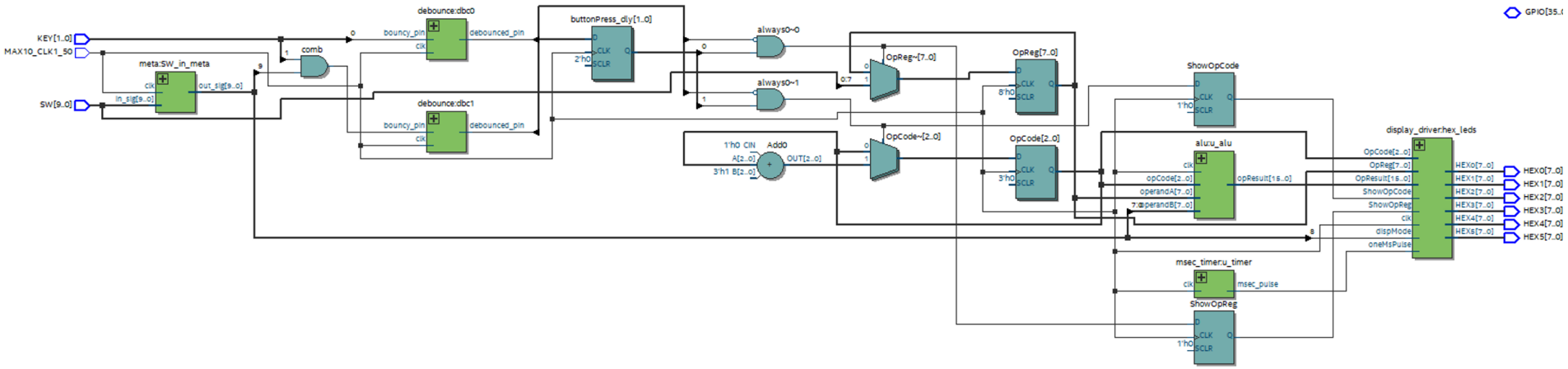


Figure 1: Top-level View of the ALU Test Design

Modules dbc0 and dbc1 (***debounce***) are in effect a glitch filtering module that is designed to produce clean high to-low and low-to-high transitions during initial switch transition periods. This clean signal is then used to trigger either loading data from SW[7:0] into the OPREG[7:0] register (C1) and displaying "rEg xx" for 3 seconds, or for incrementing the OPCODE[3:0] register (C2) and displaying "CoDE x" for 3 seconds. Note that KEY[1:0] utilize an analog debounce circuit, and that OPREG won't really show bouncy behavior, but we are feeding KEY[0] through the debouncer regardless. KEY[1] is AND-ed with SW[9] in the top-level so that KEY[1] can properly demonstrate the need for debouncing.

Module u_timer (***msec_timer***) generates a 1-millisecond pulse for the purposes of the 3-second display interval. The parameter FREQ_KHZ is a 16-bit value for the speed, in kHz, of the clk input. For this project, we will directly use the 50 MHz clock provided to the FPGA from its board (DE2 or DE10-Lite), so the value we use is 50000. The output is a single-bit **pulse** (i.e. active for only one clock cycle) that is high once every millisecond.

Module u_alu (***alu***) contains the arithmetic logic unit. The ALU selects from the table of 8 functions (see Table 1) according to the OpCode input. The operandA input is tied in the top-level to OPREG, and the operand input is tied to the switches (after the metastability registers). The 16-bit result must be passed through a set of flip-flop registers (clocked via the clk input) prior to being output from this module.

Module hex_leds (***display_driver***) accepts the debounced trigger edge-detected pulses, the millisecond timer pulses, the values of OPREG[7:0], SW[7:0], and RESULT[15:0] in order to produce the hexadecimal display output. This module contains 6 submodules of type ***hex_driver*** to encode the desired display values to the hex LED pins (as shown in Table 2). An FSM internal to ***display_driver*** is used to produce the 5-bit values for each digit. C4 must be implemented in Verilog.

Laboratory Assignment:

This laboratory assignment is composed of three phases, with each phase culminating with the current state of the design being validated by prototyping it on the Terasic DE2-115 or DE10-Lite platform. Students must successfully complete and demonstrate the valid functionality of a given phase to their design to the lab instructor before proceeding to the next phase, though this can be done via video recording for online/remote students.

Preparation: Use the New Project Wizard as from Lab 2 for either the DE10-Lite or DE2-115 board. Open the file <project name>.sdc in your project folder, and add (or verify or overwrite an existing constraint, if it exists):

DE10-Lite:

```
create_clock -name {MAX10_CLK1_50} -period 20.0 -waveform {0.000 10.000}
[get_ports {MAX10_CLK1_50}]
```

DE2-115:

```
create_clock -name {CLOCK_50} -period 20.0 -waveform {0.000 10.000} [get_ports
{CLOCK_50}]
```

Phase I: Creation of the *display_driver* module and connection to hex LEDs. In this phase, students are to develop the hex digit display and its submodule (individual LED encoder ***hex_driver***). The ***hex_driver*** module is combinational only and is recommended to use a Verilog case statement to transcode the values in the InVal column of Table 2 to each corresponding OutVal value. Note that the OutVal encoding for special non-hexidecimal characters r, o, and g are left to the student to determine. After ***hex_driver*** is coded, continue to work on the //TODO sections in the display_driver.v Verilog file. Upon completion, open the top-level file and verify the ***display_driver*** component instance is connected to the HEX0-HEX5 outputs to the top-level outputs, and that

SW[8] is attached to the dispMode input. Set the OpResult input to 16'h1ECE (this is done in the top-level by default), then compile and load it onto the board for demonstration.

Phase II: Creation of the *debounce* module In this phase, students are to develop the basic glitch filtering circuit that reduces the effect of mechanical conductor vibrations. Whenever the two electrical conducting contacts that are present in a mechanical switch are placed in contact with one another they tend to bounce back and forth causing the electrical connection between them to be made and broken several times before the connection becomes stable. The same is generally true when the two contacts are separated from one another. A debounce circuit is one that filters this activity in a manner that only a single logic transition will be present. Often this clean output is important to prevent multiple triggering of electronic devices.

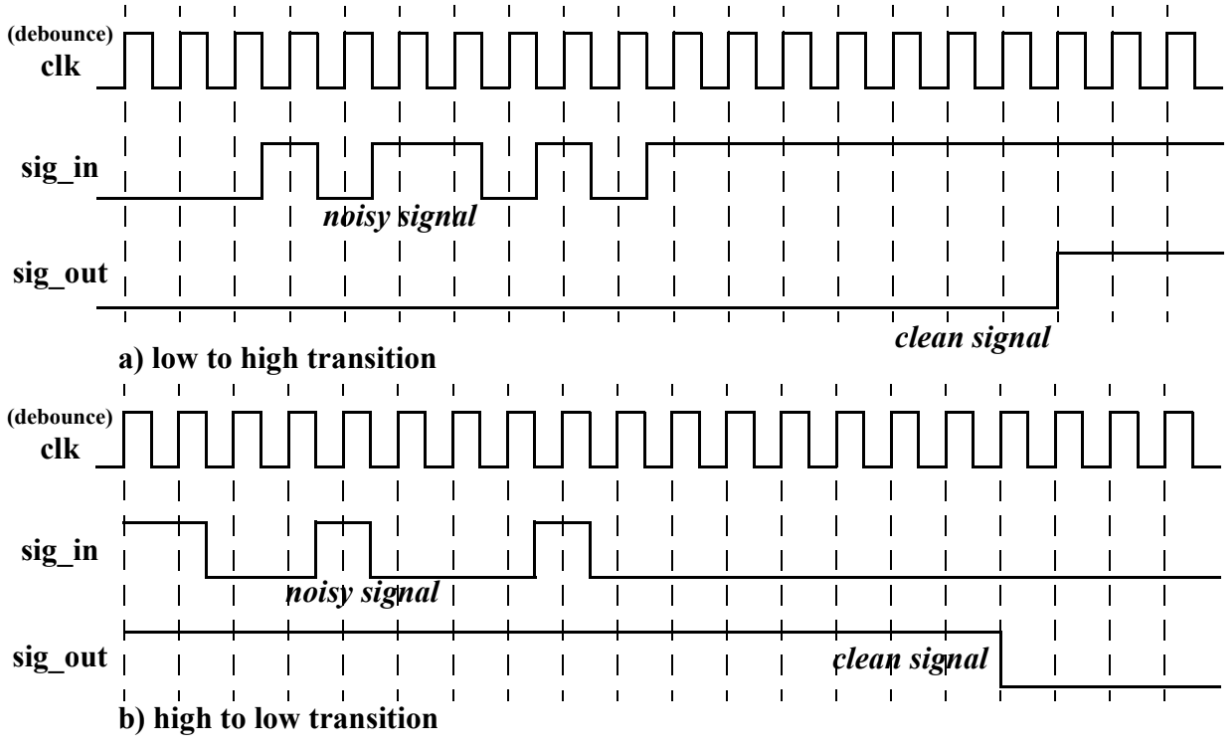


Figure 2: Example Waveforms for *debounce* Module

There are several ways the effects of switch bounce can be removed. One way is to introduce a delay period after an initial transition from one logic level to another during which time the input is no longer monitored. This has the disadvantage that responsiveness can be compromised if the period is too large and that noise can cause false triggers. The manner that switch bounce is to be filtered in this laboratory assignment involves the idea of creating a sequential circuit that is clocked at a rate that is likely to record the multiple triggerings that are associated with switch bounce. The idea is that this debounce circuit will remember a set of values from the past and only pass on a transition from low-to-high or high-to-low when the input being monitored has been stable at its new logic level for that set number of clock cycles.

In this phase of the assignment, students are to develop a design with the following module declaration:

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

This module must include a 1-bit register that maintains the current state of the debounced pin (i.e. the sig_out output that is not prone to rapid fluctuations). Whenever this state **does not match** the sig_in input, a 16-bit register that counts from 0 up to DWELL_CNT increments by one. Whenever the debounced state **matches** the input pin, the 16-bit register is cleared. If the 16-bit counter register reaches DWELL_CNT, then the state inverts to the new value on sig_in (i.e. the debounced state is inverted from its previous state) and the 16-bit counter resets to zero. It may help in this process to look at the msec_timer.v module that was provided in the template for an example of an up-counter that terminates at a specified constant value.

Once the ***debounce*** module is finished, it must be instantiated twice in the top-level file, once for the KEY[0] input (which is redundant) and once for the combined (SW[9] & KEY[1]) pair of inputs. The output of each of these debouncers is tied to the wire buttonPress[0] and buttonPress[1], respectively. Each of those signals goes through an edge detect circuit, with a falling edge of buttonPress[0] causing OpReg[7:0] to load along with a pulse on the showOpReg signal, and with a falling edge of buttonPress[1] causing OpCode[2:0] to increment along with a pulse on the showOpCode signal. Part of this code is done to show that the negative edge detect is performed by delaying the signal through a flip-flop to compare the current value of the signal with the value from one cycle ago. Thus we consistently use one global clock and avoid the misconception that a falling edge on buttonPress[] should be used as a clock input for a negative edge triggered flip-flop.

These 4 signals (OpReg, OpCode, showOpReg, showOpCode) are passed into the ***display_driver*** module on the corresponding inputs. At this point, the student must demonstrate the impact of the debounce circuit by first setting DWELL_CNT to 16'h0001. After compiling and loading onto the target, repeatedly flipping SW[9] back and forth should occasionally cause the CoDE # output to skip numbers instead of incrementing by 1 as expected. After demonstrating this, the student is encouraged to test a few values of DWELL_CNT to determine a suitable setting that eliminates skipping numbers. With a suitable value of DWELL_CNT set for both instances of ***debounce***, the next phase can commence.

Phase III: Creation of the *alu* module using Verilog HDL

To implement the ALU, students are required to implement all 8 OpCodes listed in Table 1 in a single always @(posedge clk) procedural block. Furthermore, students are required to use Verilog operators discussed in lectures (as listed in the Operators column in Table 1) to perform the arithmetic or bitwise logical functions. Note that the 8-bit inputs are to be treated as unsigned integers, which simply means that the 8-bit inputs do not need their MSbit to be interpreted as a sign bit and extended (replicated) to all bits from [15:8] to produce 16-bit results.

Once this has been coded, recompile and experiment with the board. Note the use of SW[8] to produce the result in decimal form (when low) versus hexidecimal form (when high). Demonstrate to your TA how to load OpReg with KEY[1], how to increment OpCode with KEY[0] as well as SW[9]. Then produce examples of each OpCode producing 3 different results where the two operands are unique in each of the 3 trials.

Post-Lab Questions:

Answer the following questions and include your answers in the appropriate section of the final laboratory report.

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)
        ii. How many "Total registers" are in use? These are the FPGA's sequential elements
        iii. How many pins are in use?
        iv. Did Quartus use a dedicated multiplier in the ALU (for OPCODE 5)? How can you tell?
        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)
    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.
        ii. Is Fmax better or worse for Slow 1200 mV 0C (here the only difference is the simulated temperature of operation)?
        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.
            1. The worst-case slack (which can be negative, indicating failed timing!) is listed first.
            2. List the source and destination node and the reported slack.
            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?
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.
    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[3], decDigits[2], decDigits[1], decDigits[0]?
    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 graphs.
    d. Are these Mealy or Moore state machines?
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"?
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.

DE2-115 Pin Connection Information.

Table 2: Keyboard Row/Column Connections

Keypad Pin Number	Design Signal Name (design inputs)	DE2-115 Pin Location	Expansion Cable Wire	Keypad Pin Number	Design Signal Name (design outputs)	DE2-115 Pin Location	Expansion Cable Wire
1	COL[0]	PIN_AC19	blue/green0	5	ROW[0]	PIN_AF24	blue/green4
2	COL[1]	PIN_AF16	blue/green1	6	ROW[1]	PIN_AE21	blue/green5
3	COL[2]	PIN_AD19	blue/green2	7	ROW[2]	PIN_AF25	blue/green6
4	COL[3]	PIN_AF15	blue/green3	8	ROW[3]	PIN_AC22	blue/green7

Table 3: DE2-115 Pin Locations for 7-Segment Hexadecimal LED Displays HEX0 -- HEX7

Design Signal Name	Direction	DE2-115 Pin Location	Design Signal Name	Direction	DE2-115 Pin Location
HEX0[6]	Output	PIN_H22	HEX4[6]	Output	PIN_AE18
HEX0[5]	Output	PIN_J22	HEX4[5]	Output	PIN_AF19
HEX0[4]	Output	PIN_L25	HEX4[4]	Output	PIN_AE19
HEX0[3]	Output	PIN_L26	HEX4[3]	Output	PIN_AH21
HEX0[2]	Output	PIN_E17	HEX4[2]	Output	PIN_AG21
HEX0[1]	Output	PIN_F22	HEX4[1]	Output	PIN_AA19
HEX0[0]	Output	PIN_G18	HEX4[0]	Output	PIN_AB19
HEX1[6]	Output	PIN_U24	HEX5[6]	Output	PIN_AH18
HEX1[5]	Output	PIN_U23	HEX5[5]	Output	PIN_AF18
HEX1[4]	Output	PIN_W25	HEX5[4]	Output	PIN_AG19
HEX1[3]	Output	PIN_W22	HEX5[3]	Output	PIN_AH19
HEX1[2]	Output	PIN_W21	HEX5[2]	Output	PIN_AB18
HEX1[1]	Output	PIN_Y22	HEX5[1]	Output	PIN_AC18
HEX1[0]	Output	PIN_M24	HEX5[0]	Output	PIN_AD18

HEX2[6]	Output	PIN_W28	HEX6[6]	Output	PIN_AC17
HEX2[5]	Output	PIN_W27	HEX6[5]	Output	PIN_AA15
HEX2[4]	Output	PIN_Y26	HEX6[4]	Output	PIN_AB15
HEX2[3]	Output	PIN_W26	HEX6[3]	Output	PIN_AB17
HEX2[2]	Output	PIN_Y25	HEX6[2]	Output	PIN_AA16
HEX2[1]	Output	PIN_AA26	HEX6[1]	Output	PIN_AB16
HEX2[0]	Output	PIN_AA25	HEX6[0]	Output	PIN_AA17
HEX3[6]	Output	PIN_Y19	HEX7[6]	Output	PIN_AA14
HEX3[5]	Output	PIN_AF23	HEX7[5]	Output	PIN_AG18
HEX3[4]	Output	PIN_AD24	HEX7[4]	Output	PIN_AF17
HEX3[3]	Output	PIN_AA21	HEX7[3]	Output	PIN_AH17
HEX3[2]	Output	PIN_AB20	HEX7[2]	Output	PIN_AG17
HEX3[1]	Output	PIN_U21	HEX7[1]	Output	PIN_AE17
HEX3[0]	Output	PIN_V21	HEX7[0]	Output	PIN_AD17

System Clock: Design Signal Name: **CLOCK_50** -- **Input** -- Pin Location: **PIN_Y2**