**EE260 Lab 3 Sign-Off Sheet**

Your Name:

Lab Partner: Date Performed:

## Demonstrated correctly:

* + Pre-lab Complete (1) (2) (10 pts)

(Each student graded individually up to 10 pts)

* + Hex to 7-seg Decoder (10 pts)
  + Verilog 4-bit Adder /w hex display (15pts)
  + Verilog 4-bit ALU with 7-seg display (30 pts) Two's Comp module

4-bit AND/OR modules

B input multiplexer

negSign logic

difference multiplexer

Output multiplexer

*ALU total points*

* + ***BONUS***: Signed/Unsigned Overflow (**5pts**)
  + TA Questions: (1) (2) (5 pts)
  + Report (one per team) (30 pts)

(including all Verilog source code & test benches, constraint files and simulation screen shots)



**EE260: Introduction to Digital Design**

**Lab 3 – Implementing a 4-bit Four Function ALU**

# Objective:

Inside a computer's central processing unit (CPU) there is a sub-block called the arithmetic logic unit (ALU) which computes integer math and bit-wise logic functions. In this lab exercise you will implement a simple ALU capable of addition, subtraction,

bit-wise AND and bit-wise OR on 4-bit operations. As usual, you will implement this lab entirely in Verilog and program it to your FPGA. It may seem like you have a long time in this lab but remember the Thanksgiving holidays are in there so you should start right away. While no single part is particularly difficult, there are ***a lot*** of parts to this lab.

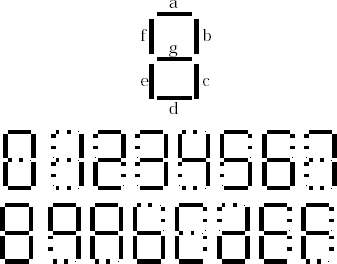
# Pre-lab Assignment:

This pre-lab assignment is to be completed before your lab session and must be

signed-off by the TA during your lab session. Pre-labs help you to become oriented to the problem before you enter the lab, help complete your design in advance and prevent wasting time in the lab. INCLUDE THE PRE­LABS FROM BOTH PARTNERS IN YOUR REPORT!!

1. READ the whole lab assignment!
2. Implement the 7-segment display

Seven segment displays are commonly used as alpha-numeric displays by logic and computer systems. A seven segment display is an arrangement of 7 LEDs (Figure 1) that can be used to show the hex digit for any number between 0000-1111b (think The Martian!) by illuminating combinations of these LEDs. In most cases, all LED’s in a seven segment display will have a common cathode. To illuminate an LED segment, you will assert a logic level on its input. For example, segments *a, b, c, d, e,* and *f* must be lit to display a 0 and only segments *b* and *c* are lit to display 1.

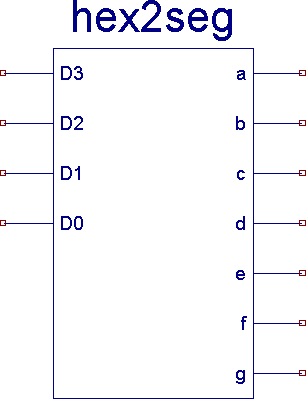


*Figure 1*

Finish implementing this truth table for a hexadecimal (0-Fh) to 7-segment display decoder shown in Table 1. This circuit block (figure 2) has four inputs and seven outputs. The four inputs D3 D2 D1 D0 can be any of the binary codes for 0-F hex. The output is the logic levels for the segments, *a, b, ... g,* which need to be lit to display the hex digit. *However*, the 7-segment displays on the Basys3 boards are "active low" which means *a* = 0 will turn on segment *a* and *a* = 1 will turn it off. The same is true for all segments. Applying 0 means the segment is on and 1 means it is off. For more information see pages 15­17 (8.1 Seven Segment Display) of the Basys3 Reference Manual.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Char** | **Inputs** | | | | **Outputs** | | | | | | |
|  | **D3** | **D2** | **D1** | **D0** | **a** | **b** | **c** | **d** | **e** | **f** | **g** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 |  |  |  |  |  |  |
| 3 | 0 | 0 | 1 | 1 | 0 |  |  |  |  |  |  |
| 4 | 0 | 1 | 0 | 0 | 1 |  |  |  |  |  |  |
| 5 | 0 | 1 | 0 | 1 | 0 |  |  |  |  |  |  |
| 6 | 0 | 1 | 1 | 0 | 0 |  |  |  |  |  |  |
| 7 | 0 | 1 | 1 | 1 | 0 |  |  |  |  |  |  |
| 8 | 1 | 0 | 0 | 0 | 0 |  |  |  |  |  |  |
| 9 | 1 | 0 | 0 | 1 | 0 |  |  |  |  |  |  |
| A | 1 | 0 | 1 | 0 | 0 |  |  |  |  |  |  |
| B | 1 | 0 | 1 | 1 | 1 |  |  |  |  |  |  |
| C | 1 | 1 | 0 | 0 | 0 |  |  |  |  |  |  |
| D | 1 | 1 | 0 | 1 | 1 |  |  |  |  |  |  |
| E | 1 | 1 | 1 | 0 | 0 |  |  |  |  |  |  |
| F | 1 | 1 | 1 | 1 | 0 |  |  |  |  |  |  |

*Table 1*



*Figure 2*

# Lab Assignment:

The first task in this lab is to create the hex to 7-segment decoder so we will be able to display the output of our 4-bit ALU.

1. Double check the truth table you generated for pre­lab. *Remember*, the

7-segment displays on the Basys3 boards are "active low" which means *a* = 0 will turn on segment *a* and *a* = 1 will turn it off. The same is true for all segments. Applying 0 means the segment is on and applying 1 means it is off. For more information see pages 15­17 of the [Basys3 Reference](https://www.digilentinc.com/Data/Products/BASYS3/Basys3_rm.pdf).

Verilog is a very capable HDL. In addition to being able to make combinational logic assignments based on boolean expressions like

// F = X'Y + XZ + Z'Y

assign F = ((~X)&Y) | (X&Z) | ((~Z)&Y);

it can create higher level conditional statements. The syntax of a Verilog conditional statement is like the if then else statements in software programming languages. The syntax is

assign val = (condition) ? true\_value : false\_value

If the condition evaluates to 1 then the val=true\_value, otherwise

val=false\_value.

Ex: Consider a Boolean expression where F =1 when inputs A=0, B=1 and C=0 or when A=1, B=1 and C=1. In a Verilog conditional statement, this becomes

assign F = ((~A)&B&(~C)) | (A&B&C)) ? 1 : 0;

The real power of Conditional Statements is that they can be nested and the variables can be buses. This allows the designer to *describe* a truth table with multiple inputs and outputs (like the hex to the 7-segment decoder) without solving all the logic expressions.

In our 7-segment decoder, the individual segments *a-g* will be stored in a 7-bit variable called segs[6:0]. In general, to set the value of a multi-bit variable in Verilog, the syntax is #bits 'b bitpattern;

Ex: //Assign a 7-bit value to segs assign segs = 7'b0000001;

Nested conditional statements are an especially efficient way to implement decoders, encoders, and multiplexers. Below is the implementation of a generic 2-to-4 decoder as a Verilog module.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***A[1]*** | ***A[0]*** | ***Y[3]*** | ***Y[2]*** | ***Y[1]*** | ***Y[0]*** |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

module decode2to4(input [1:0] A, output [3:0] Y);

assign Y = (A == 2'b00) ? 4'b0001 : (

(A == 2'b01) ? 4'b0010 : (

(A == 2'b10) ? 4'b0100 : 4'b1000));

endmodule

This code is interpreted as

If A == 00 then Y = 0001 else If A == 01 then Y = 0010 else

If A == 10 then Y = 0100 else Y=1000

It is important that in the last condition you close all the parentheses. There are three lines in this conditional statement thus there are two ) to close.

Below is the start (and end) of the conditional assignment for the 7-seg decoder.

assign segs = (D==4'b0000) ? 7'b0000001 : (

(D==4'b0001) ? 7'b1001111 : (

. . .

(D==4'b1110) ? 7'b0110000 : 7'b0111000))))))))))))));

See the class notes from Verilog lecture (Lecture 10) for more examples.

1. Create a new RTL project for Lab 3 in Vivado and add a new source for design called

hex2seg.v. The input should be a 4-bit bus called D, and there should be two outputs, a 7-bit bus called segs, and a 4-bit output called anodes.

## BE SURE TO FILL IN THE COMMENT BLOCK FOR EACH VERILOG MODULE IN YOUR DESIGN!

1. Enter the Verilog conditional assignment statement to implement the 7-segment decoder you designed for pre-lab. Also, assign the four display *anodes* to be 1110 (i.e., three anodes off only one anode on). Verify your HDL code and save.

assign anodes = 4'b1110;

1. Add a constraint file to your project and assign inputs D[3:0] to SW3 to SW0 and connect outputs segs[6:0] to CA thru CG with segs[6] = CA and segs[0] = CG. Also connect anodes[3:0] to AN3 to AN0 respectively (See page figure 16 in the [Basys3 Reference](https://www.digilentinc.com/Data/Products/BASYS3/Basys3_rm.pdf)).
2. Add a Verilog test fixture to your project and assert all possible inputs to your 7-seg decoder. Run Synthesis and then run a Behavioral Simulation. *Be sure to include a screen capture of your test bench results in your report.*
3. Run Implementation and perform a Post Implementation Timing simulation. *Be sure to include a screen capture of your test bench results in your report.*
4. Now generate the bit file, download your design and test it by setting the value to be displayed on SW3-SW0. Save your project. You will reuse this 7-seg decoder in future labs! ***Show you hex-to-7-segment decoder to the TA for sign-off***.

# Implementing the 4-Bit ALU

Below is a block diagram of our ALU. The inputs to the ALU are two 4-bit operands (A and B) and two single bit inputs Op1 Op0, which select the operation to be performed. The output is 4-bit Result*.* The “opcodes” for our ALU are listed in Table1.

*Op1*



4-Bit

*ALU*

*Op0 Result*



*A3-0 B3-0*

|  |  |  |
| --- | --- | --- |
| ***Op1*** | ***Op0*** | ***Operation*** |
| 0 | 0 | Addition: Result = A + B |
| 0 | 1 | Subtraction: Result = A - B |
| 1 | 0 | Bitwise AND: Result = A & B |
| 1 | 1 | Bitwise OR: Result = A | B |

## Table 2

We will use a divide and conquer approach to this lab and implement each operation in a separate module. Then, we'll combine those modules and the 7-segment display into the final design. Each operation module is *a separate combinational circuit,* and all will exist in parallel. Remember we're implementing Hardware not writing Software. The logic within the ALU mostly involves multiplexing of inputs and outputs. We will start with a bit of *code reuse* and import the 4-bit adder and the 2's comp module you created in your last lab.

A(3:0)

B(3:0)

-B

Op0

2's

comp

Add

Res(3:0)

ss

Op1 Op0

OR

AND

2's comp

A(3:0)

B(3:0)

A(3:0)

B(3:0)

1. Add a new Verilog module to your project called *add4hex.v*. Make it the Top Module by right-clicking on it and choosing “Set as Top Module.” It should have two 4 bit inputs *A* and *B,* and it should also have three outputs, 4-bit Result, 7-bit *segs7* and 4-bit *anodes* (we will just connect the *OF\_S* and *C\_MSB* outputs of *fourAdd* to an internal “wire” for the moment )*.*
2. Under the Project Manager, select Add Sources and add your full adder and four-bit adder Verilog code from last lab (i.e., *fullAdder.v* and *fourAdd.v*). These source files are stored in a directory called something like

"proj\_name/proj\_name.srcs/sources\_1/new".

Below is this directory structure for my lab two project. Note that I run Linux, but you will see a similar directory structure in Windows Explorer for your lab two project directory.

-----------------------------------------

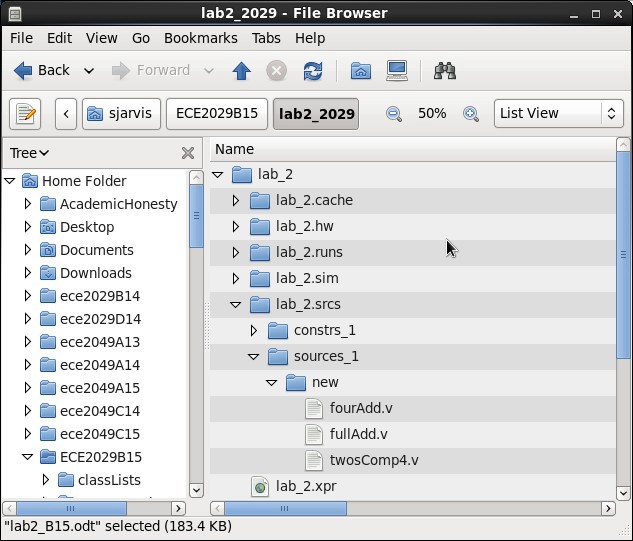
9a. *ALTERNATIVELY*: If your 4-bit adder from last week is not working, you can implement a new adder module in Verilog. Do not add *fullAdder.v* and *fourAdd.v* to your project. Rather add a new Verilog module to your project called *fourAdd.v* and make the following assignment inside your *fourAdd.v* module and save.

You *do NOT* need to do both step 9 and 9a. Choose one method only!

assign Sum = A + B;

assign OF\_S=((~Sum[3])&A[3]&B[3])|(Sum[3]&(~A[3])&(~B[3])); assign C\_MSB=Cout[3];

*------------------------------------------*



1. Make sure *add4Hex.v* is the Top Module and add the following instantiations to the

*add4hex.v* and save.

wire of\_s, c\_msb;

fourAdd U1(A,B,Sum,of\_s,c\_msb); hex2seg U2(Sum,segs7,anodes);

1. Remove (but don't delete) the constraints file for *hex2seg.v* then add a constraint file to *add4hex* and assign input A[3:0] to SW3 to SW0 and input B[3:0] to SW7 to SW4. Connect output Sum[3:0] to LEDs LD3 to LD0, respectively. These LEDs will help you debug your hex display. Connect segs7[6:0] to CA thru CG with segs[6] = CA and segs[0] = CG. Also connect anodes[3:0] to AN3 to AN0 respectively (See page figure 16 in the [Basys3 Reference](https://www.digilentinc.com/Data/Products/BASYS3/Basys3_rm.pdf)).
2. Synthesize, implement and generate the bit file. Download and test your design. The sum of the values entered on the switches for A and B should now be displayed on the

7-segment display as well as on the LEDs. ***Show the TA for sign-off***.

1. The AND and OR modules are very simple to implement. In fact, you don't logically have to implement them as separate modules in Verilog but to follow the block diagram above we will. Add a new Verilog module to your project called *and4bits.v.* The module will have two 4-bit inputs *A* and *B* and a 4-bit output *andAB*. The module will have a single assignment. Save your AND module.

assign andAB = A & B;

1. Now, add another Verilog module called *or4bits.v* and similarly implement a 4-bit OR. The vertical | is OR in Verilog. Save your OR module.
2. Finally, add your four bit 2's complement module *twosComp4.v* from last weeks lab to the project.
3. We now have all the functional blocks needed to implement our ALU, but we need to consider the output multiplexing for display. The ADD, AND and OR operands will all always produce 4-bit unsigned results. However, SUBTRACT could produce a negative result which would be encoded in 2's complement. For the subtraction results to display properly, we need to detect when subtraction (and subtraction only) has resulted in a negative number (i.e., summ[3]=1), and determine the magnitude of the result. Thus we want to display the difference (equal to the output of the adder) when positive but to display the magnitude and a – sign when the subtraction result is negative. This can be implemented with multiplexers. See the code framework below.
4. To implement our ALU add a new Verilog module to your project called *ALU4bit.v* and make it the Top Module. This file will need the following three inputs: 4-bit *A* and *B*, and 2-bit o*p*. It should also have the following outputs: 4-bit *Result*, single bit *negligent*, 7-bit *segs7* and 4-bit *anodes*. Complete the code below to implement a four operation ALU.

// wires are not inputs or outputs but hold intermediate

// values within a circuit

// (each of these are 4 wire buses)

wire [3:0] summ, diff, my4and, my4or, Bneg, Barg, magg; wire ovf,c\_msb; // these are single bit wires

// encode -B in 2's comp twosComp4 U1(B,Bneg);

// multiplexer to create adder B argument (either B or -B)

// depending on whether op = 01 (subtract) assign Barg = (op=='b01) ? Bneg : B;

// single adder circuit handles addition and subtraction fourAdd U2(A,Barg,summ,ovf,c\_msb);

// Finish instantiating 4-bit AND and 4-bit OR and4bits U3(A,B,my4and);

...

// If result of subtraction is negative we'll want to

// display magnitude with a – sign. Implement logic that

// sets negSign = 1 if subtraction produced a negative

// result otherwise negSign = 0 assign negSign = ...

// Do 2's comp again to find magnitude

// (we will only use magg if negSign=1) twosComp4 U5(summ,magg);

// Now implement multiplexer logic that assigns diff=summ

// when the results of subtraction was positive and assigns

// diff=magg when the results are negative assign diff = ...

// Implement the final output mux assign Result =(op=='b00) ? summ :(

((op=='b01)) ? diff:(

. . . )));

// Display results to 7-seg display hex2seg U6(Result,segs7,anodes);

1. Remove (but don't delete) your *add4hex* constraint file and add a new constraint file to work with your ALU and save it as *alu4bit.xdc*. Operands *A* and *B* should be entered on the slide switches, and *op[1]* and *op[0]* should be connected to push buttons L and C. We only have a single 7-seg display working at the moment so we will display the *negative sign by lighting LD7*. Next lab we'll get the all four 7-seg displays working! Connect segs7[6:0] to CA thru CG and connect anodes[3:0] to AN3 to AN0.
2. Save your project. Run Synthesis and Run Implementation. Generate a programming file and test your four functions ALU. ***Show the TA for sign-off***.

# BONUS (5 pts): Overflow for signed and unsigned operations.

So far we've ignored the signed overflow and carry out of the MSB outputs of the four-bit adder. Recall that the SIGNED overflow circuit assumed the operands were encoded in 2's complement and that if the sign bit (MSB) of the result was different that the sign bits of the operands that Overflow had occurred. In our ALU the ADD operation (Op = 00) will be assumed to be *unsigned* meaning there is no sign bit. All 4 bits are used to convey magnitude. Overflow occurs in UNSIGNED addition when there is a Carry Out of the MSB. However, the SUBTRACT operation (Op=01) is still signed and AND and OR have no overflow at all. Edit your Verilog code to properly indicate using a single LED (e.g., LD15) whether Overflow for unsigned ADD or signed SUBTRACT has occurred.