ECE2029: Introduction to Digital Circuit 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 completes integer math and bitwise logic functions. In this lab exercise you implement a simple ALU capable of implementing addition, subtraction, bitwise AND and bitwise OR on 4-bit operands. You will implement this lab completely in Verilog as it is easier than "wiring" a schematic for larger designs like this project. It may seem like you have a long time for this lab but remember the Thanksgiving holidays are in there you should start right away. 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 lab, help complete your design in advance and prevent wasting time in lab. INCLUDE THE PRE-LABS FROM BOTH PARTNERS IN YOUR REPORT!!

1) READ the whole lab assignment!

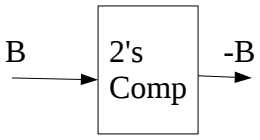
Two's Complement Circuit:

2) Design a 2's complement circuit. As we saw earlier in the term 2's complement encoding is an efficient way to encode negative numbers because it is "consistant" under addition. The four bit adder you implemented last lab works for the addition of both positive and negative numbers provided the numbers are encoded in 2's complement format. As you may remember from Mini-Exam #1 expressing negative numbers in 2's comp is not very intuitive to do by hand. While it is relatively easy to convert +6 to 0110b it is not as straigthforward to convert -6 to its 2's comp representation of 1010. Therefore our ALU will automatically convert the subtrahend B to 2's complement before doing subtraction.

$$C = A - B = A + (-B)$$

If B is a positive number in the range 0 to 2^{n-1} where n is the number of bits in the binary representation (in our case $n = 4$) then

$$-B = \text{not}(B) + 1.$$



Assuming B is a 4-bit value, $B_3\ B_2\ B_1\ B_0$, draw a circuit that generates the 2's complement representation of -B. You may use your four bit adder block from last week for adding the 1.

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 by illuminating combinations of these LEDs. In most cases all LED's in a seven segment display will have 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.

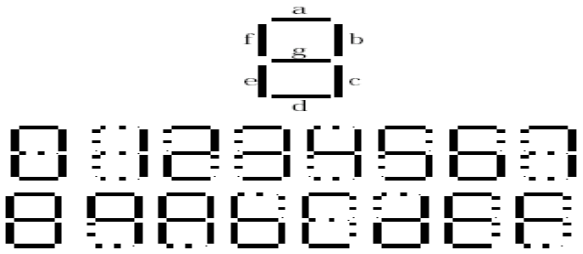


Figure 1

3) Finish implementing the this truth table for a hexidecimal (0-Fh) to 7-segment display decoder shown in Table 1. This circuit block (figure 2) has 4 inputs and 7 outputs. The 4 inputs D_3 D_2 D_1 D_0 can be any of the binary codes for 0-F hex. The output are the logic levels for the segments *a*, *b*, ... *g* that need to be lit to display the hex digit. *HOWEVER*, the 7-segment displays on the Nexys3 boards and "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 Section 2.6 pg 72 of your text book and pages 18 and 19 (Basic I/O and Seven Segment Display) of the Nexys3 Reference .

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

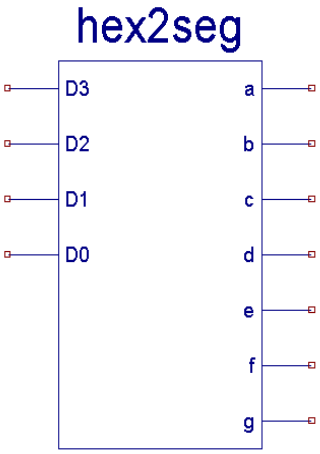


Figure 2

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 pre-lab. *Remember*, the 7-segment displays on the Nexys3 boards and "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 Section 2.6 pg 72 of your text book and pages 18 and 19 of the Nexys3 Reference.

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

```
// F = X̄Y + XZ + Z̄ 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 if then else statements is 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 7-segment decoder) without solving all the logic expressions.

In our 7-segment decoder the indiviual segments *a-g* will be stored in a 7-bit variable called `segs[6:0]`. In general, to set the value of a multibit 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 on the last condition you close all the parentheses. There are 3 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.

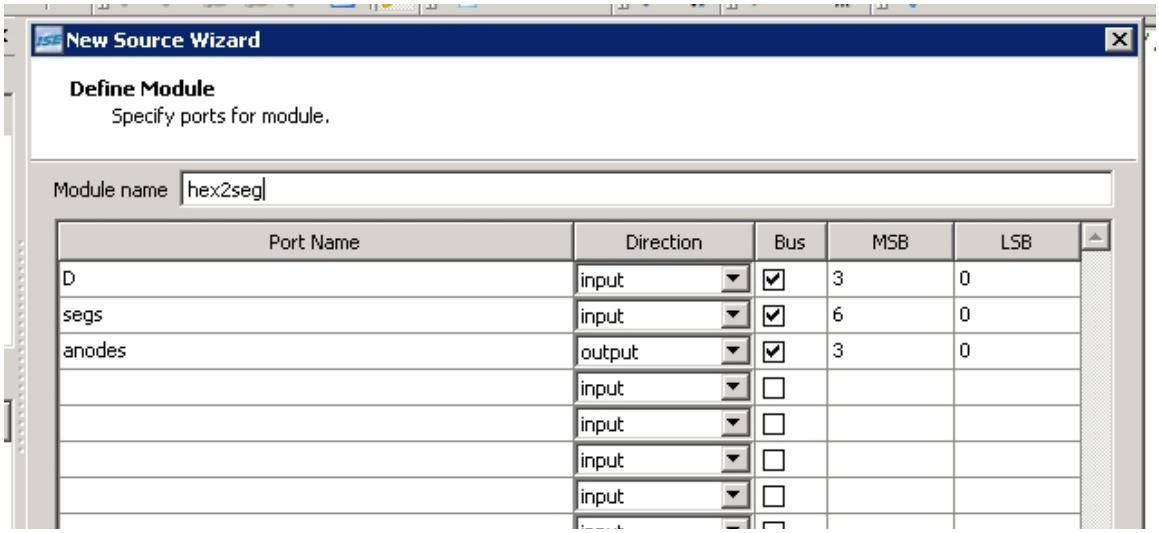
```
assign segs = (D==4'b0000) ? 7'b0000001 : (
                (D==4'b0001) ? 7'b1001111 : (

                          . . .

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

2. Create a new HDL project in Xilinx (i.e. Top Level Source set to HDL) and add a new Verilog module called `hex2seg.v`. The input should be a 4-bit bus called `D` and there should be 2 output, a 7-bit bus called `segs` and a 4-bit output called `anodes`.

3. Enter the Verilog conditional assignment statement to implement the 7-segment decode. Also, assign the 4 display *anodes* to be 1110 (i.e. 3 anodes off only one anode on). Verify your HDL code and save.

```
assign  anodes = 4'b1110;
```

4. Add a Veriog test fixture to your project and assert all possible inputs to your 7-seg decoder. *Be sure to include a screen capture of your test bench results in your report.*



5. Click on your hex2seg.v file and add the following ucf file with SW3-SW0 connected to *D* and *segs* connected to the 7-segment display and *anodes* connected to AN3, AN2, AN1 and AN0.

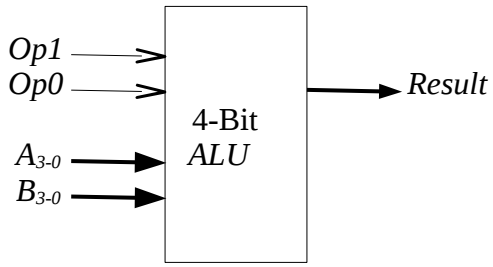
```
#Example contents of hex2seg.ucf
NET "D(0)"  LOC = "T10"  ;
NET "D(1)"  LOC = "T9"  ;
NET "D(2)"  LOC = "V9"  ;
NET "D(3)"  LOC = "M8"  ;
# NOTE: segs(6) is a and segs(0) is g
NET "segs(0)" LOC = "L14" ;
NET "segs(1)" LOC = "N14" ;
NET "segs(2)" LOC = "M14" ;
NET "segs(3)" LOC = "U18" ;
NET "segs(4)" LOC = "U17" ;
NET "segs(5)" LOC = "T18" ;
NET "segs(6)" LOC = "T17" ;
NET "anodes(0)" LOC = "N16" ;
NET "anodes(1)" LOC = "N15" ;
NET "anodes(2)" LOC = "P18" ;
NET "anodes(3)" LOC = "P17" ;
```

6. You are ready to generate the bit file but remember to first set the start up clock to JTAG CLOCK to avoid the annoying error box in Adept. Right click on Generate Programming File and then select Startup Options on the left. Switch the FPGA Startup Clock option to JTAG Clock and click Apply.

7. Now generate the bit file, download your design and test by setting the value to be displayed on SW3-SW0. Save your project. You will reuse this 7-seg decoder in future labs! ***Show the TA for sign-off***.

Implementing the 4-Bit ALU

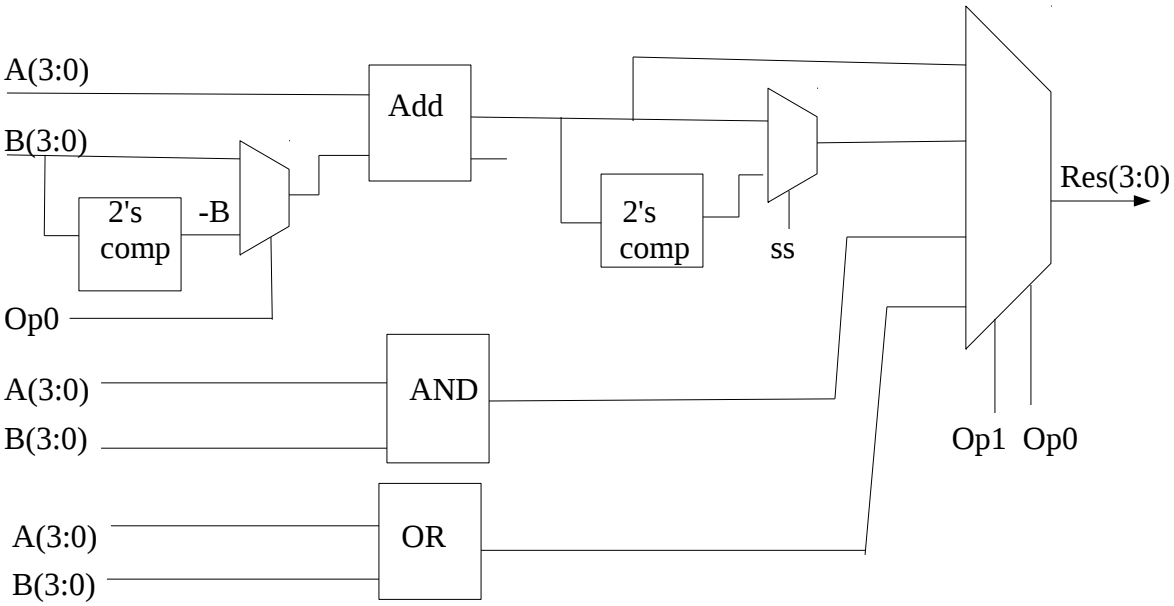
Below is a block diagram of our ALU. The inputs to the ALU are two 4-bit operands (A and B) and 2 single bit inputs Op1 Op0 that select the operation to be performs. The output is 4-bit *Result.* The "op codes" for our ALU are listed in Table1.



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 1

We will use a divide and conquer approach to this lab and implement each operation in its own module. Then, we'll combine those modules and the 7-segment display into the final design. Each operation module is *its own 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. Actually we will start with a bit of *code reuse* and import the the 4-bit adder you created in your last lab.



8. 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 connect *Overflow* to just an internal "wire" for the moment).

9. Under the Project Tab, select Add Source and add your full adder and four-bit adder Verilog code from last lab (i.e. *fullAdder.v* and *fourAdd.v*).

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 make 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 Overflow=((~Sum[3])&A[3]&B[3])|(Sum[3]&(~A[3])&(~B[3]));
```

10. *A*dd the following instantiations to the *add4hex.v* module and save.

```
wire Overflow;

fourAdd U1(A,B,Sum,Overflow);
hex2seg U2(Sum,segs7,anodes);
```

11. Add the ucf file below to add4hex.v then generate the bit file, download and test. 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***.

```
#Example contents of add4hex.ucf
NET "A(0)"  LOC = "T10"  ;
NET "A(1)"  LOC = "T9"  ;
NET "A(2)"  LOC = "V9"  ;
NET "A(3)"  LOC = "M8"  ;
NET "B(0)"  LOC = "N8"  ;
NET "B(1)"  LOC = "U8"  ;
NET "B(2)"  LOC = "V8"  ;
NET "B(3)"  LOC = "T5"  ;
NET "Sum(3)"  LOC = "V15"  ;
NET "Sum(2)"  LOC = "U15"  ;
NET "Sum(1)"  LOC = "V16"  ;
NET "Sum(0)"  LOC = "U16"  ;
NET "segs7(6)" LOC = "T17" ;
NET "segs7(5)" LOC = "T18" ;
NET "segs7(4)" LOC = "U17" ;
NET "segs7(3)" LOC = "U18" ;
NET "segs7(2)" LOC = "M14" ;
NET "segs7(1)" LOC = "N14" ;
NET "segs7(0)" LOC = "L14" ;
NET "anodes(0)" LOC = "N16" ;
NET "anodes(1)" LOC = "N15" ;
NET "anodes(2)" LOC = "P18" ;
NET "anodes(3)" LOC = "P17" ;
```

12. Add a new Verilog file to you project called *comp2.v*. The module will have one four bit input *B* and a single 4-bit output *negB*. In Verilog, 2's complement can be implemented in one line using your 4-bit adder. Declare the overflow output at a wire.

```
wire  ovflw;

fourAdd U1((~B),'b0001,negB,ovflw);
```

13. Add a Verilog test fixture and associate it with *comp2.v*. Assert all possible inputs to your 2's complement module. Remember, if the sign bit $B_3 = 1$ then the result of the 2's comp circuit will be the magnitude of the negative value. That is, when you input

`0110b = +6 decimal  negB = 1010 = -6 in 2's comp`

but when you input `1010 = -6 in 2's comp then negB = 0110 = |-6|.`
Be sure to take a screen capture of your 2's complement test fixture for your report.

14. 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;
```

15. 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.

16. We now have all the operations needed to implement our ALU but we need to consider the output multiplexing and 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. In order 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 to equal the output of the adder when positive but to display the magnitude and a – sign when the subtraction is negative. This can be implemented with multiplexers. See the code framework below.

17. Now, 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 3 inputs: 4-bit *A* and *B*, and 2-bit o*p*. It should also have the following outputs: 4-bit *Result*, single bit *negSign*, 7-bit *segs7* and 4-bit *anodes*. Complete the code below to implement a 4 operation ALU.

```
// wires are not inputs or outputs but hold intermediate
// values within a circuit
wire  [3:0] summ, diff, my4and, my4or, Bneg, Barg, magg;
wire  ovf;

// encode -B in 2's comp
comp2 U1(B,Bneg);

// multiplexer to create adders B argument (B or -B)
assign Barg = (op=='b01) ? Bneg : B;

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

// 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)
comp2 U5(summ,magg);


// Now implement multiplexer logic that assigns diff=summ
// when the results of subtraction was positive and assigns
// diff=magg when the reults 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);
```

18. Modify your *add2hex.ucf* file to work with your ALU and save it as *alu2hex.ucf*. Operands *A* and *B* should be entered on the slide switches and *op[1]* and *op[0]* should be connected to push buttons 1 and 0. We only have a single 7-seg display working at the moment so we will display the *negative sign by lighting LED*7. Next lab we'll get the all four 7-seg displays working! Add *alu2hex.ucf* to your project and associate it with your ALU module. Remeber you can only have one ucf file associated with your top module.

19. Save your project. Generate a programming file and test your 4 function ALU. ***Show the TA for sign-off***.

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

So far we've ignored the overflow output of the four bit adder. Recall that the Overflow as implemented last lab was a SIGNED overflow. That is, the 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 is still signed and AND and OR have no overflow at all. Edit your Verilog code to properly indicate using a single LED Overflow for unsigned ADD or signed SUBTRACT.

ECE2029 Lab 3 Sign-Off Sheet

Make sure lab instructor/TA initials and dates each part. Attach this sheet and the Report Grading Rubric to your team's lab report!

Both partners MUST be present at sign-off!

Your Name: ECE BOX #:

Lab Partner: Date Performed:

Demonstrated correctly:

- •Pre-lab Complete (1)_____ (2)_____ (10 pts)
  (Each student graded individually upto 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 schematic, Verilog & test bench and test screen shots)

Lab 3 – Implementing a 4-Bit ALU

Review Item	***Comments***	***Points***	***(max)***
1) Prelabs from each student complete and thoughtful			(5)
2) Introduction effectively presents the objectives and purpose of the lab. Methodology gives enough details to allow for replication of procedure.			(5)
3) Discussion opens with an effective statement on the goals of the lab, backs up statement with reference to appropriate findings, provides sufficient and logical explanation for the statement, addresses other issues pertinent to lab.			(5)
4) Results opens with effective statement of overall findings, presents visuals clearly and accurately, presents findings clearly and with sufficient support. You MUST include screen shots of the test bench results for each part of the lab. Conclusion convincingly describes what has been learned in the lab.			(5)
5) Other: References are included. Tables and figures are formatted. Grammar and spelling are correct Report is written clearly and to the point.			(10)
Overall, the team... • has successfully demonstrated what the lab was designed to teach • demonstrates clear and thoughtful scientific inquiry • has accurately measured and analyzed data for lab findings			
Total:		_____	(30)