Lab 3: Complete ALU

Summary:

Students will learn how to combine the components built from past labs into an ALU.

Learning Objectives:

- Explore the implementation of a simple ALU
- Gain experience running multi-bit simulations with *Virtuoso*

Resources and Supplies:

- Engineering PC with PuTTY and Xming
- *Lab3A.txt & Lab3B.txt**
- *Cadence Virtuoso Setup Guide**
- *Cadence Virtuoso Logic Gates Tutorial**

All documents* are available on the class website

Pre-lab Assignment:

Each student must complete his/her own pre-lab before coming to the lab. Pre-lab check-off sheets must be turned in to the TA before starting the lab assignment.

1. Read the **Background** sections of this lab.
2. Print the Pre-lab 3 Check-off Sheet near the end of this document and perform the required tasks. Show the completed sheet to the TA at the beginning of your lab.
3. Read through the entire **Laboratory Assignment** section so you know what to expect in the lab.

Background:

ALU Bit Slice:

Many digital integrated circuits are designed to process multiple bits in parallel. For example, an 8-bit ALU with two inputs bytes will execute an arithmetic/logic function on all 8 input pairs simultaneously. As you might expect, the circuitry for each bit is exactly the same (with some minor exceptions). In such a case, it is more efficient to design and optimize 1-bit circuit blocks and then combine them into word-sized functions, which could contain to as many bits as needed. As discussed in the Lab 2 Background, the 1-bit circuit block for an ALU is called a *bit slice*. Figure 1 shows an example of a 4-bit ALU composed of four bit slices. Notice the select signals (S_0, S_1) are shared by all bits so that they all perform the same task. The C_{out} bit from a lower-bit block is cascaded to the next bit block as its C_{in} input, and only the lowest bit (Bit 0) actually has an external input for C_{in}. The two 4-bit inputs A[3:0] and B[3:0] are paired bit-wise as inputs to each of the bit slices. Note: the B input to the ALU is not the same signal as the B input of the adder because the op-code will modify the B input of the adder (to B, B', or 0) regardless of the value in B. Each bit slice is composed of a logic block and an arithmetic block, and you designed a simple version of such blocks in Lab 2. For reasons beyond the scope of this class, for large-word ALUs (~16+ bits) this two-block architecture is not typically maintained, but it is functionally similar so this "simplified structure" is a good way to introduce ALU design.

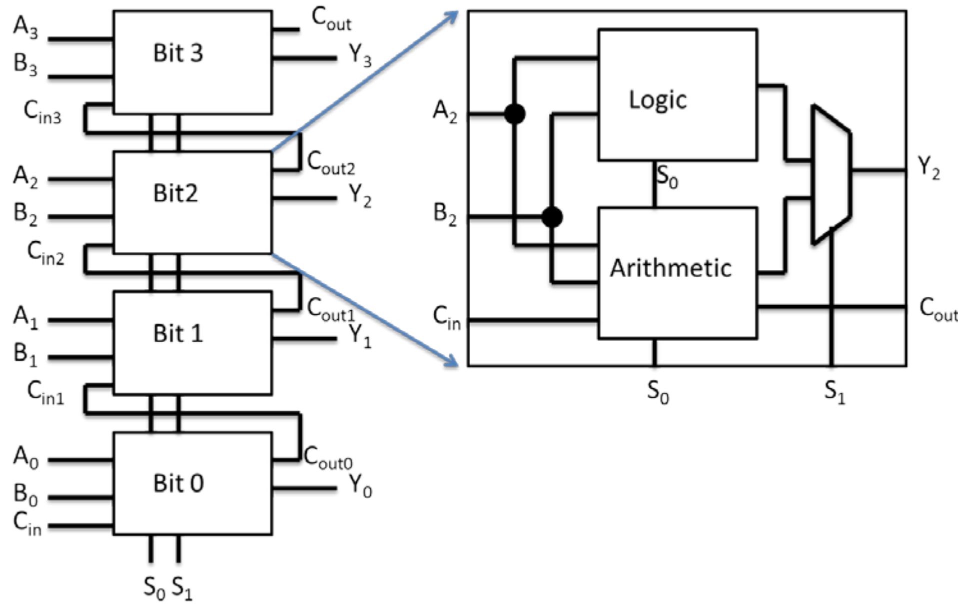


Figure 1: A generic 4-bit ALU organized into four bit slices each consisting of a logic block and an arithmetic block.

ALU Function Selection:

The ALU is responsible for executing all of the arithmetic and logic functions for a CPU. The number of functions that an ALU can execute is determined by the number of functions implemented within the logic and arithmetic blocks of each bit slice. Operation codes, or op-codes, are the binary commands sent to the ALU to make it perform a specific operation (i.e., execute a specific function). Although you may think of op-codes as a software feature, the bits of the op-code are tied to specific hardware controls, like the select lines in the Figure 1 example. For large/complex ALUs, the op-codes may be an encoded form of the selection lines, but functionally we can consider the op-code bits as hardware selection signals. In fact, for simple function ALUs, like the one you will design in this lab, the op-code bits will be precisely the selection signals that specify what function the ALU will implement. The more functions an ALU can perform, the larger the op-code must be. Often an op-code can be optimized (use fewer bits) by careful organization of ALU selection signals so that multiple inputs can share the same op-code bit. Note that the op-code is responsible for setting all inputs that define the *function* of the ALU. Consider the example 4-bit ALU in Figure 1 and the design of your Lab 2 logic and arithmetic blocks that use 4:1 MUX circuits. The data inputs, A and B, are independent of the ALU function, but all of the MUX select signals and the C_{in} signal (the lowest-bit C_{in}; see Figure 1) are external inputs that must be controlled by the op-code in order to set the desired ALU function.

In this lab you will implement an 8 bit ALU with 8 functions using your logic and arithmetic blocks from Lab 2. You may recall that your logic block had 2 selection signals, and your arithmetic block had 2 selection signals plus a C_{in} signal. To select between logic and arithmetic, you would need an additional selection signal for a total of 6 command input signals. Combined, these circuit blocks and their 6 selection signals implement 8 functions. However, the optimal op-code for 8 functions could be composed of only 3 bits (2^3=8). Resolving this discrepancy is the subject of your pre-lab assignment

8-bit ALU Operations:

In learning logic functions, you have probably always seen results operated on single bits. However, the 8-bit ALU will perform operations on two 8-bit bytes. For logic functions, two-byte operations will be performed ***bit-wise***, meaning each ith bit of one byte, A_i, will be operated on with the ith bit of the other byte, B_i. For example:

A[7:0] OR B[7:0] → (A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0) OR (B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0)
would generate an 8-bit result Y[7:0] = (Y_7 Y_6 Y_5 Y_4 Y_3 Y_2 Y_1 Y_0) such that
$Y_7 = A_7 + B_7$, $Y_6 = A_6 + B_6$, $Y_5 = A_5 + B_5$, etc., where "+" refers to the bit-wise logic OR.

This example also clearly defines the multi-bit notation A[n-1:0] for an n-bit binary value used throughout this lab assignment.

Notice that the Bit Slice and ALU will have a Cout output. Like all digital outputs, Cout will have a value for all input combinations. However, logic functions do not generate a Cout. If you notice form Figure 1, Cout is always generated by the arithmetic function, even when Y is selected to come from the logic block. In practice, additional logic would be incorporated to ensure logic functions do not change the Carry flag status register, but for this simplified design we will omit this step and simply ignore the Cout bit for all logic functions.

In Lab 2, you observed several arithmetic functions for two single-bit inputs. However, the 8-bit ALU will act on two 8-bit bytes. Let's clarify the 8-bit operations for functions implemented by your arithmetic block designed in Lab 2.

ADD: Notice from the function tables in Lab 2 and the Lab 3 Pre-lab that you will be implementing an ADD function with Cin = 0. This means your ALU will not support an input carry, for example coming from a previous instruction. You may have observed that the 68HC12 instruction set provides both *without-carry* (e.g., ADDA) and *with-carry* (e.g., ADCA) arithmetic instructions. To keep your ALU simple (and permit you to optimize to only 3 function select signals), you will only implement the *without-carry* arithmetic. Thus, for your ALU, the 8-bit ADD operation is simply the binary addition of A[7:0] with B[7:0], including carries from all lower order bits. This can be expressed as:

$$\begin{array}{r} A_7\ A_6\ A_5\ A_4\ A_3\ A_2\ A_1\ A_0 \\ B_7\ B_6\ B_5\ B_4\ B_3\ B_2\ B_1\ B_0 \\ \underline{+\quad C_6\ C_5\ C_4\ C_3\ C_2\ C_1\ C_0\ 0} \\ C_{out}\ |\ Y_7\ Y_6\ Y_5\ Y_4\ Y_3\ Y_2\ Y_1\ Y_0 \end{array}$$

where C_i is the lower-order carry out and the lowest carry (Cin) = 0 by the op code selects.

SUBT: Subtraction is functionally the same as ADD, but here the Cin is set high and B is inverted so that you will implement A[7:0] + B'[7:0] + 1. As you hopefully fully understand by now, B'[7:0] + 1 is simply the 2's complement of B[7:0] such that
Y[7:0] = A[7:0] - B[7:0] = A[7:0] + B'[7:0] + 1

Also, note that Cout in your circuit is simply the Cout of the full adder and not the carry flag. Thus, you should not expect it to be inverted for subtract functions like the C flag.

INC: The increment function simply sets all B inputs to 0 and Cin to 1 so that
Y[7:0] = A[7:0] + 1, where "+" refers to binary addition; adds $01 not $FF (eight 1's)

PASS A: To pass A to the output, the B inputs and Cin are set to zero. Thus
Y[7:0] = A[7:0], which in a bit-wise notation says $Y_7 = A_7$, $Y_6 = A_6$, $Y_5 = A_5$, etc.,

Laboratory Assignment:

This lab consists of two parts. A check-off sheet is included at the end of this lab document to record your design work and results. After you successfully finish each part of the lab, show the TA your results and ask him/her to sign the check-off sheet.

- Print the check off sheet.

Part 1: Bit Slice

Design

1. The design stage was completed in the pre-lab assignment. Before proceeding, make sure you have shown your op-code and bit slice sketch to the TA and confirmed everything is correct. If anything is wrong, correct your op-code and bit slice design and then continue.

   In the table under Part 1 of the Lab 2 Check-off Sheet, fill in the correct function defined by the op-code select bits S[2:0] to match your results from pre-lab. The function (e.g., XOR) should be written in the first column labeled “Function”.

   Copy the function list into the “Function” column in the check-off sheet table for Part 2. The order of the select bits is the same so your function list should be repeated.

   Also, because Cout is only used for arithmetic functions, we do not care what the value of Cout is for all *logic* functions (AND, OR, XOR, INV). To represent this, write an ‘X’ (don’t care) into the Cout column for all 4 logic functions of the Part 1 & 2 tables. For the Part 2 table, you would place the ‘X’ before the vertical line, ‘|’, separating Cout from the 8-bit output Y[7:0].

Implementation in Cadence Virtuoso

2. Start Cadence *Virtuoso* in your ECE331/virtuoso directory. Refer to the *Cadence Virtuoso Setup Guide* and Cadence Virtuoso Logic Gates Tutorial if you run into problems.
3. Select your project library and create a new schematic cellview for the Bit Slice. Name the circuit **ALUbitslice**.
4. Create the ALUbitslice schematic by implementing the design sketched in the pre-lab assignment. Add the necessary instances (cells) and add wires to connect your circuit properly.
5. Add I/O pins for all external inputs and outputs. Make sure the inputs are input pins and the output is an output pin. Use the signal names defined in the pre-lab (**A, B, Cin, S2, S1, S0**, **Y, Cout**).
6. Check-and-Save the cellview and fix any errors or warnings.
7. Create a symbol for the ALUbitslice schematic (Create->from cellview…->symbol). A rectangular shape is fine. Generally, inputs are placed on the left and outputs on the right. You can move the select signals to the bottom or leave them on the left but separate them from the A, B & Cin inputs. The Figure 1 inset may be helpful.
8. Check and Save the symbol, then exit the symbol tool.

Demonstration

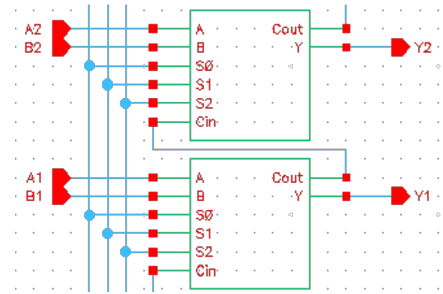
9. For any given set of inputs, your ALUbitslice circuit should generate known output values. Based on your design, fill in the expected output columns (**Y, Cout**) of the partial function table in the check-off sheet where A=1, B=0, and $C_{in}=S_1$. Although this does not show all possible input combinations, it provides a sample of outputs from each logic function.
10. In *Virtuoso*, run a simulation of your ALUbitslice schematic using the Lab3A.txt stimulus file available from the class website. Note, you must assign logic functions to op-code bits and use the I/O signal names as defined in the pre-lab or this stimulus file will not work correctly. This file should simulate the input combinations in the check-off sheet table by cycling through op-codes from 000 to 111. Inputs A and B are set to the DC values shown on the check-off sheet table; you may want to omit them from your list of waveforms to plot to focus only on variable signals.
11. Observe the simulation waveforms and identify the output value for each of the input combinations on the check-off sheet function table. For each function that matches your expected output values, place a checkmark in the last column of the table. If you find any errors, search for the cause and fix it.
12. Print the simulation output waveform to include in your lab report. Be sure all signals are clearly identifiable and the plot is clearly labeled and titled.
13. Show the TA your ALUbitslice schematic, symbol, and simulation results along with the function table in the check-off sheet. Ask the TA to sign your check-off sheet for Part 1.

Part 2: 8-bit ALU

Design & Implementation in Cadence Virtuoso

With a correctly functioning bit slice, you can now construct an 8-bit ALU. To do this you will use 8 bit slices working in parallel, with some minor adjustments to form the 8-bit ripple carry adder. Rather than sketching the design first, this time we'll just do all the design in *Virtuoso*.

1. With *Virtuoso* running from your class directory, create a new schematic cellview called **8bitALU**.
2. Instantiate 8 ALUbitslice symbols into the schematic and stack them similar to Figure 1. Choose one of the bit slices to be your lowest order ALU bit, like the Bit 0 identified in Figure 1. It could be on bottom, or top, or left or right depending on how you organized the pins in your ALUbitslide symbol.
3. Wire the Cout from the lowest order bit slice to Cin of the next bit. Then wire Cout of that bit to Cin of the next bit, and so on until all the Couts are connected to Cins in series. You should see now why it's called a ripple carry adder, since the carry bit ripples through the bits in series. The only unconnected carry signals should be Cin of the lowest order bit and Cout of the highest order bit.
4. Wire all of the op-code select bits to each bit slice. To keep your wiring neat, you may want to create bus-style wiring, where each select line is formed by a



continuous wire across one side (or top or bottom) of your schematic and then you tap off of this wire to make each bit slice connection. This can be much easier to view than random point-to-point wiring. See image for an example.

5. Wire the *Cin* input from the lowest order bit to the appropriate op-code signal. Notice that now Cin is not an independent input but rather is determined by the op-code (and the function it specifies).
6. Add input pins for each of the op-code select signals (**S0, S1, S2**) and connect them to the proper signal wires on your schematic.
7. Add input pins for the A and B inputs and name them **A0, A1… A7** and **B0, B1… B7**. Wire these inputs to the bit slices, with A0 and B0 wired to the lowest order bit slice, and so on up to the 7th bit slice.
8. Add output pins for the Y outputs and name them **Y0, Y1… Y7**. Wire these outputs to the appropriate bit slice.
9. Finally, add an output pin for the highest bit's **Cout** and wire it to the highest order bit slice.

When the wiring is completed, you should have an 8-bit ALU with *19 inputs and 9 outputs*.

10. Check-and-Save the 8bitALU schematic cellview and fix any errors or warnings.
11. Create a symbol for the 8bitALU organizing all of the pins in an orderly manner. Check and Save the symbol, then exit the symbol tool.

Demonstration

12. For any given set of inputs, your 8bitALU circuit should generate known output values. Based on your design, fill in the expected output columns (**Cout | Y**) of the partial function table in the check-off sheet where Y is an 8-bit value and Cout is shown as the 8th (highest) bit. Separate Cout from Y with a "|" or a "," to avoid confusion. Notice that this table uses a single 8-bit value for A (00001111) and B (01000110) but has a row for each of the op-code combinations (i.e., for each ALU function). When you are done, you may want to check your results with another student to decrease the chance that a small mistake will cause problems later (but don't just copy someone else's table!!).
14. In *Virtuoso*, run a simulation of your 8bitALU schematic using the Lab3B.txt stimulus file available from the class website. Note, you must use the 8-bit ALU I/O signal names defined above or this stimulus file will not work correctly. This file should simulate the input combinations in the check-off sheet table by cycling through op-codes from 000 to 111. Inputs A[7:0] and B[7:0] are set to the DC values shown on the check-off sheet table; you may want to omit them from your list of waveforms to plot and focus only on variable signals.
13. Observe the simulation waveforms and identify the output value for each of the input combinations on the check-off sheet function table. For each function that matches

your expected output values, place a checkmark in the last column of the table. If you find any errors, search for the cause and fix it.

14. Print the simulation output waveform to include in your lab report. Be sure all signals are clearly identifiable and the plot is clearly labeled and titled.
15. Show the TA your 8bitALU schematic, symbol, and simulation results along with the function table in the check-off sheet. Ask the TA to sign your check-off sheet for Part 2.

Wrap Up

Clean up your lab bench before you exit the lab.

Remember, your Lab 3 report will be due in lab next week. Each student must submit his/her own lab report. Include your check-off sheet and waveform printouts with your report. You may want to review the Discussion Points below and talk to the TA about anything that is not clear to you.

Discussion Points

As explained in the *Lab Report Guide*, you should address these discussion points in a designated section of your report.

1. In the demonstration for Part 1, we only observed 8 input combinations. To test all possible input combinations of the bit slice, how many unique settings would need to be tested. Discuss how you answer this problem; do not just state a number.
2. In the demonstration for Part 2, we only observed 8 input combinations. To test all possible input combinations of the 8-bit ALU, how many unique settings would need to be tested. Discuss how you answer this problem; do not just state a number.
3. Consider how the ripple carry adder works and discuss the disadvantage(s) of this structure for large-word (e.g., 32 bits) ALUs.
4. Based on your experience in this lab where you were able to implement an 8-function ALU with only 3 op-code bits ($8 = 2^3$), do you think it would always be possible to implement a *K*-function ALU with only *n* op-code bits (where K = 2n)? Include some explanation/thoughts with your answer.
5. When running the simulation you should have noticed sharp spikes or glitches in output signals when they were not supposed to be changing. If you study these closely, they should all occur when some input(s) changes. What is the source of these glitches? What feature of a logic gate determines the maximum width a glitch can be before it can become a significant problem?

PRE-LAB 3

Due: At the beginning of lab.

Student Name: ____________________________ Lab. Section (time): __________

Refer to the Background sections of Labs 2 and 3 for supporting information.

Observe the function table below that shows all of the ALU functions implemented in Lab 2 along with the logic and arithmetic selection bits. An "L" has been added to the logic selects (e.g., LS_1) and an "A" has been added to the arithmetic selects to reduce confusion. The table also includes an "LAS_2" selection bit that is necessary to choose between logic and arithmetic functions (like the S_1 bit in Figure 1). Thus, you will notice that the logic selects are only defined when $LAS_2 = 0$, and arithmetic selects are only defined when $LAS_2 = 1$.

	L/A select	Logic Block		Arithmetic Block			Bit Slice (op-code bits)		
Function	**LAS_2**	**LS_1**	**LS_0**	**AS_1**	**AS_0**	**C_{in}**	**S_2**	**S_1**	**S_0**
INV A	0	0	0						
XOR	0	0	1						
OR	0	1	0						
AND	0	1	1						
ADD	1			0	0	0			
PASS A	1			0	1	0			
SUBT A-B	1			1	0	1			
INC A	1			1	1	1			

<u>Pre-lab Tasks</u>

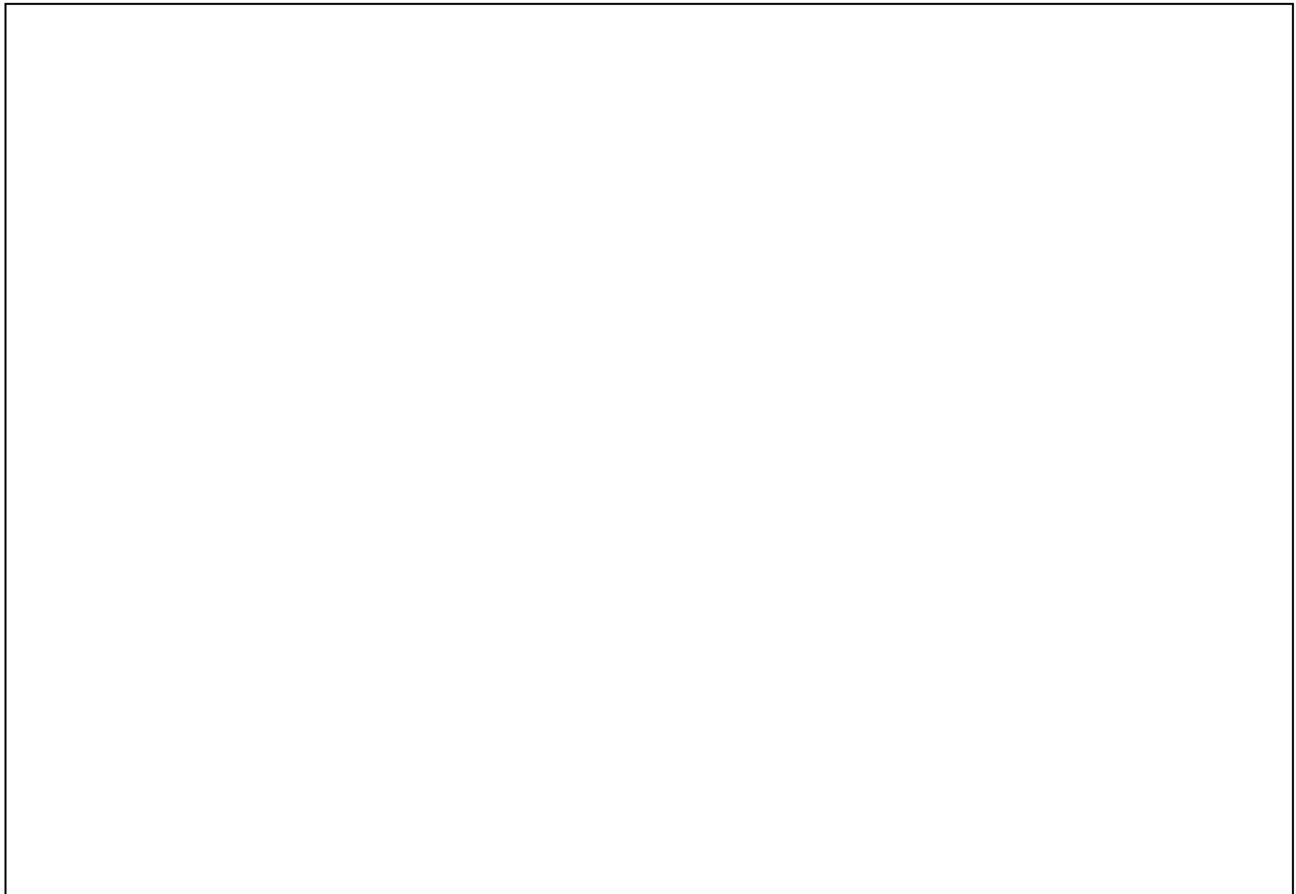
1. Check your Lab 2 circuits to confirm they were designed to implement exactly the functions and select combinations shown in the table above. If not, modify your circuits so that they do.
2. Recognizing that the logic select are don't cares (x) during arithmetic functions and vice versa, fill in the open spots of logic and arithmetic selects in the table above with don't cares (x). Then, study the table above and consider how the 6 physical select signals can be implemented with only 3 bit-slice control bits (op-code bits). Can some be shared? Can some be implemented a logic combinations of others?
3. Define your bit slice op-code by assigning values to S[2:0] in the table above for each ALU function and defining each of the physical inputs below as a logic expression of the op-code bits. For example, a good choice is to set $LAS_2 = S_2$. Although this may be a bit confusing at first, it should actually be quite straightforward. The real "hard part" has already been done for you when Lab 2 defined the logic and arithmetic select/function combinations in a way that can easily be simplified now.

LAS2 = ___<u>S2</u>___, LS1 = __________, LS0 = ___________

AS1 = __________, AS0 = ___________, Cin = ___________

4. In the space below, sketch a schematic for a 1-bit bit slice composed of your LogicBlk, ArithBlk and 2:1 MUX from pervious labs. It should look something like the inset in Figure 1 but with more detail on the inputs. The LogicBlk and ArithBlk should show labels for all the inputs and outputs that your Virtuoso symbols for those blocks have. Open your Virtuoso files if you need to check. Include pins for the bit slice inputs (**A, B, Cin, S2, S1, S0**) and outputs (**Y, Cout**).

   Note: if you optimized your op-code correctly, you should have expressed Cin as a function of S[2:0] and thus you might wonder why you need Cin as an input to the bit slice. You want Cin as an input because you will need to cascade Cout from one bit to Cin of the next, as shown in Figure 1. Only the first/lowest (Bit 0) Cin will be assigned to the op-code bits. Thus, if you include Cin as an input, you can use the same bit slice schematic for all bits of the ALU and adjust for the different connections when constructing the overall ALU.



Sketch of Single Bit Slice Schematic

At the beginning of lab, ask the TA to check this Pre-lab 3 Check-off Sheet to ensure you have the correct op-code and bit slice plan. Fix any mistakes and ask the TA to sign below to indicate you have completed this assignment. Since you may need this sheet for the in-lab assignment, keep this sheet after the TA signs it and then attach it to your lab report.

TA sign off Initial___________

LAB 3 CHECK-OFF SHEET

Student Name: ___________________________ **Lab. Section (time):** __________

Part 1: Bit Slice

Step 1 & Step 9. Partial ALU Bit Slice function table & simulation results

Function	**S_2**	**S_1, C_{in}**	**S_0**	**A**	**B**	**Y**	**Cout**	**√**
	0	0	0	1	0			
	0	0	1	1	0			
	0	1	0	1	0			
	0	1	1	1	0			
	1	0	0	1	0			
	1	0	1	1	0			
	1	1	0	1	0			
	1	1	1	1	0			

Part 1: TA sign off Initial____________

Part 2: 8-bit ALU

Step 1 & Step 12. Partial ALU function table & simulation results

Function	**S_2**	**S_1**	**S_0**	**A[7:0]**	**B[7:0]**	**Cout | Y[7:0]**	**√**
	0	0	0	00001111	01000110	|	
	0	0	1	00001111	01000110	|	
	0	1	0	00001111	01000110	|	
	0	1	1	00001111	01000110	|	
	1	0	0	00001111	01000110	|	
	1	0	1	00001111	01000110	|	
	1	1	0	00001111	01000110	|	
	1	1	1	00001111	01000110	|	

Part 2: TA sign off Initial____________