***Carleton University***

***Department of Systems and Computer Engineering***

***SYSC 3006 Fall 2016***

***Computer Organization***

***Lab #2 (Version 3)***

---

**Prelab:** Your Prelab content should be done *before* the start of your scheduled lab session. Bring your Prelab with you in hard or soft copy form (i.e. print it or have a word/pdf document ready) for TA inspection during the lab session (i.e. no online submission).

*Note:* missing or forgetting to bring the Prelab can result in up to 40% mark loss.

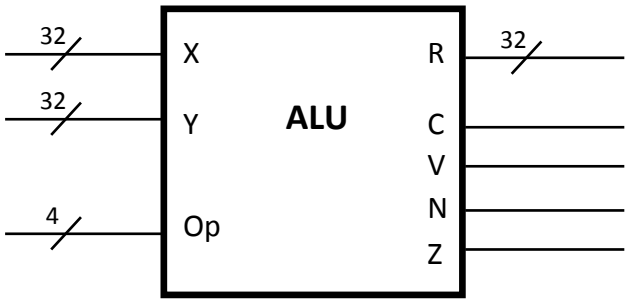
**Self-study:** These questions/tasks are there to help you check your own understanding of the content. They will not be marked.

In this lab you will:

- Design an ALU circuit similar to one that might be included in a processor
- Implement and test the design using Logisim.

**Read this entire document carefully before deciding what to do.**

The component-level (black box) abstraction of the ALU is shown here:



The ALU must satisfy the following block-level requirements. Additional design and implementation requirements are given later.

**The ALU Interface:**

| | |
|---|---|
| 2 x 32-bit Inputs: X and Y | inputs |
| 1 x 32-bit Output: R | output: R represents the result of applying an operation to X and Y |
| 1 x 4-bit Operation Selector: Op | input: represents the operation to be applied to X and Y to produce R |
| 4 x 1-bit Status Flags: C, V, N, Z | outputs: represent additional information about the result |

**ALU Operations:**

The table below encodes the Operations performed by the ALU. Not all of the Op encodings have been used (just to keep the size of the lab down). *Note:* iff means "if and only if"

| Operation | 4-bit Op | Result | C | V | N | Z |
|---|---|---|---|---|---|---|
| **NOP** | 0000 | R= don't care | don't care | don't care | don't care | don't care |
| **ADD** | 0001 | R = X + Y | Carry from msb | 1 iff signed overflow | 1 iff R < 0 as signed value | 1 iff R = 0 |
| **SUB*** | 0010 | R = X – Y | Borrow into msb | 1 iff signed overflow | 1 iff R < 0 as signed value | 1 iff R = 0 |
| **RY** | 0011 | R = Y | 0 | 0 | 1 iff R < 0 as signed value | 1 iff R = 0 |
| **NEG*** | 1110 | R = 2sComplement(Y) | 1 iff the addition of 1 (in the 2's complement operation) generated a carry | 1 iff signed overflow | 1 iff R < 0 as signed value | 1 iff R = 0 |
| **UMIN** | 1001 | R=minimum(X,Y) unsigned values | 0 | 0 | 1 iff R < 0 as signed value | 1 iff R = 0 |
| **SMIN** | 1010 | R=minimum(X,Y) signed values | 0 | 0 | 1 iff R < 0 as signed value | 1 iff R = 0 |

** The SUB and NEG circuits are optional; to be attempted only after finishing the other ops*

**Background:**

Review ELEC 2607 Lab 3 (Adder/Subtracter) to remind yourself about the conversion of decimal values to/from binary values, 2's complement representation of signed integers, sign extension, taking the 2's complement of a value, addition, carry and overflow. Some background information covered in SYSC 3006 class has been included at the end of this Lab description along with some self-study questions.

**Prelab:**

- Why can a single binary value represent both a signed integer value and an unsigned integer value? Use a representative example to help your explanation.

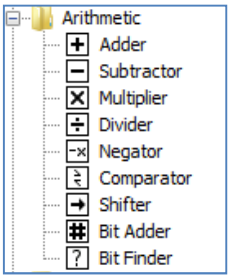
- For each component supplied in Logisim's Arithmetic Library, give a simple one-sentence summary (in your own words) of the function performed by the component.

- Why are separate UMIN and SMIN operations included? Give an example in which a pair of input values (X and Y) would result in different output values (R) for the functions.

- For each operation, propose test cases (i.e. X and Y input values) that can be used to test the ALU to demonstrate that it complies with the requirements of the operation given above. Each test case should expose different important aspects of the required functionality. There should be enough test cases to expose all important aspects of the required functionality. Describe each test in terms of any initial conditions, the inputs, the expected outputs, the particular aspects of the requirements that are tested, and why the test validates the aspects of the requirements being tested.

**Don't include this in your PreLab document**, but do it to prepare for the lab:

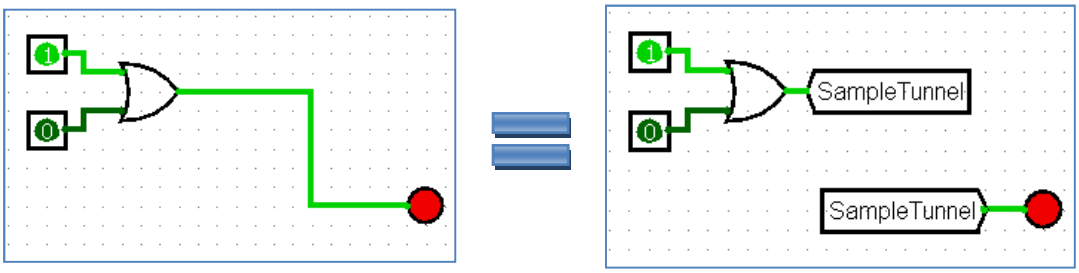
Think about how each of Logisim's Arithmetic components might be helpful in the implementation of the ALU operations. Come to the lab with design ideas for the implementation of each operation. Also think about how to design the *control circuitry* to integrate the individual operations into a single ALU circuit (HINT: the class lecture slides have a hint on this! ☺).




**In the Lab:**

Implement the required ALU. Use 32-bit registers for the inputs (X and Y). Use a 4-bit register as the operation selector (Op). Use a 32-bit probe to monitor the R output. Use individual 1-bit probes to monitor the Status Flag outputs.

The ALU implementation may result in many wires criss-crossing the circuit. Use Logisim Tunnels (in the Wiring Library) to simplify the circuit diagram. From the Logisim documentation: "A tunnel acts like a wire that binds points together, but unlike a wire the connection is not explicitly drawn. This is helpful when you need to connect points far apart in the circuit and a network of wires would make the circuit much more ugly." Always use appropriate names for the tunnels.




a) Direct wiring b) Tunnels

When testing, put appropriate values in the input registers.

Simulate and Log the test cases that you proposed in your PreLab (unless you think that other test cases might be more appropriate ☺). Log the inputs and outputs of the circuit.

Demonstrate your solution to the TA before the end of the scheduled lab session. The TAs have a few test cases to try☺ If your solution is not complete at the end of the lab session, demonstrate what you have working.

**NOTE:** The possibility that you might not get the entire circuit working in the lab should influence how you build your circuit! You don't want to waste a lot of time building up "many" components only to find that you had a fundamental misconception that resulted in having to trash all your work and redesign the entire circuit (sound familiar?). Instead, have an implementation plan that allows you to build and verify your design incrementally.

Good Luck! ☺

**Background Review and Self-Study Questions**

When talking about numerical values, this course will use terminology that is a little closer to that used in software and a little less like that used in the ELEC 2607 notes. A value from the set {0, 1, 2, 3, …} will be called an unsigned integer value, and a value from the set {…, -3, -2, -1, 0, 1, 2, 3, …} will be called a signed integer value. NOTE: 0 (zero) is not positive … it is neither positive nor negative … it is the reference point on a number line from which only positive values {1, 2, 3, …} are to the right and only negative values {…, -3, -2, -1} are to the left. When talking about signed integer values, the subset {0, 1, 2, 3, …} is called the non-negative numbers. The term "negative" is not used when talking about unsigned integer values (except in this sentence ☺).

This review note uses 8-bit binary values to represent numbers, which results in a total of $2^8 = 256$ different binary values. The concepts explored can be applied in general to n-bit values (such as the 32-bit values used in the Lab).

Each binary value can be interpreted two different ways: (1) as representing an unsigned integer value, or (2) as representing a signed integer value.

(As described in class and in ELEC2607:) Under the unsigned integer interpretation, the bits in a binary value are given a weight according to their position in the value, for example:

$1101_2$ is $1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$ or $1\times8 + 1\times4 + 0\times2 + 1\times1 = 8 + 4 + 0 + 1 = 13_{10}$ (decimal)

**[Self-Study Questions]**

**Q1: What unsigned integer value does the binary value 1111 $1001_2$ represent?**

**Q2: What is the largest unsigned integer value that can be represented by an 8-bit binary value?**

**2's Complement Encoding:** Under signed integer interpretation, binary values use 2's complement encoding. In this representation, approximately half of the possible binary values represent positive numbers and the remaining values represent negative numbers. The actual representation requires careful consideration. First, we know that there are 256 values, so it might be tempting to assume that the encoding scheme can represent 128 positive values and 128 negative values. Careful! Recall that 0 is neither positive nor negative, but it is still necessary to have a representation for 0! The binary value consisting of all 0's is used to represent the signed integer 0. This leaves 255 remaining values to represent positive and negative values. The 2's complement scheme uses the remaining binary values with the most significant bit (msb) = 0 to represent positive values. Since 0 also has its msb = 0 (but is not representing a positive value), then this leaves 127 remaining values to represent positive signed integers. As a result of using binary values that have their msb = 0 to represent zero and positive values, all values with the msb = 1 represent negative numbers. A nice by product of 2's complement encoding is that the binary values representing non-negative values corresponds exactly with the unsigned integer encodings of the same values.

**Q3: What range of signed integer values can be represented by 8-bit binary values under 2's complement interpretation?**

Recall from the ELEC 2607 notes that taking the 2's complement of a binary value representing the signed value X gives the binary value representing the signed value –X. This was used to allow you to create a "subtractor" out of an adder in the 2607 lab using the following:

$$Y - X = Y + (-X)$$

If a circuit first converts X to –X (by taking the 2's complement of X) then an adder could also be used to subtract! And of course, this was the "secret sauce" in ELEC 2607 ☺

(As described in ELEC 2607:) The way to take the 2's complement of X is to:

1. Invert all of the bits in X
2. Add 1 to X (and ignore any carry out the msb)

**Q4: What is the 2's complement of 1111 $1001_2$ ?**

**Q5: What signed integer value is represented by 1111 $1001_2$ ?** (Now look at your answer to Q2 … the same binary value has 2 different interpretations!)

Taking the 2's complement of X = 1000 $0000_2$ is a special case, in that it does not result in the value representing –X.

**Q6: Why doesn't taking the 2's complement of X = 1000 $0000_2$ result in the value representing –X? (In your answer give the value obtained by taking the 2's complement of X, explain why it is not –X, and explain the anomaly in terms of the "2's Complement Encoding" paragraph above.)**

The addition of binary values can be carried out using ordinary rules for addition (as described in ELEC 2607). The real bonus of using 2's complement encoding for signed integers is that applying the same addition rules gives consistent results for signed integers! (I.E. Adding binary values representing the signed integers X and Y gives a binary value that represents the signed integer of the sum X + Y).

**Q7: What is the (8-bit) signed integer result of adding X = 1111 $1111_2$ + Y = 0000 $0001_2$ ? (Validate your answer by giving the signed integers represented by both X and Y, the 8-bit binary value of the sum, and the signed integer represented by the sum.)**

When adding binary values, it is possible to end up with a carry out of the msb (as described in the ELEC 2607 lab). When the values being added represent unsigned integers, this carry is useful for indicating when an addition has arrived at a value that is too large to represent in the number of bits provided in the result.

**Q8: Show an addition of two unsigned integers represented using 8-bit values that results in a carry out of the msb. Explain what value the 9-bit answer represents. (Hint: might some of your work in Q7 be useful here too?)**

When adding values representing signed integers, the carry out of the msb is not very useful for deciding whether a result can be represented in the number of bits provided in the result.

**Q9: Use your answer in Q7 to argue that a carry out of the msb when adding 8-bit values representing signed integers does not necessarily mean that the answer cannot be represented in 8-bit answer.**

**Q10: What is the (8-bit) binary result of adding X = 0111 $1111_2$ + Y = 0000 $0001_2$ ? What are the signed integer values represented by X, Y and the sum? Explain why the addition didn't arrive at the correct signed integer value.**

(As described in ELEC 2607) The cases where the addition of values representing signed integers results in a value that cannot be represented in the number of bits associated with the result is called overflow.

**Q11: Suppose that Z = X + Y and that X, Y and Z are signed integers. Describe a simple test that can be performed on the binary values representing X, Y and Z to decide if overflow occurs when doing the addition.**