Simulation Lab 3: Arithmetic and Logic Unit

Prerequisites: Before beginning this laboratory experiment you must be able to:

- Use Logisim.
- Perform a 1's complement operation on a multi-bit operand.
- Have completed Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit.
- Have completed Simulation Lab 2: 4-Bit Full Adder, Multiplexer, Decoder & Buffer.

Equipment: Personal computer and Logisim.

Objectives: In this laboratory exercise you will learn how to construct an arithmetic and logic unit (ALU) using subcircuits you built with Logisim in Simulation Labs 1 and 2.

Outcomes: When you have completed the tasks in this experiment you will be able to:

- Build, test, debug, and imbed in a subcircuit, an elementary arithmetic and logic unit (ALU).
- Describe the arithmetic and logical operations of which the ALU is capable.
- Describe the input control line values that correspond to each arithmetic and logical operation of the ALU.
- Write a top-down description of a complex logic circuit.

Introduction

In Simulation Labs 1 and 2 you built combinational logic circuits of increasing complexity. In this laboratory exercise we will build upon these modules and combine them to form a more complex combinational logic subcircuit: the arithmetic and logic unit (ALU). The ALU in a microprocessor is the unit that performs all of the arithmetic operations (such as add, subtract, negate, etc.) and all of the logical operations (such as 1's complement, AND, OR, etc.) Because this is an introductory class, we will limit the number of operations our ALU will perform so that we can limit the complexity of the design. Conceptually, however, the ALU we will design will be no different from the ALU in the personal computer you use for performing these digital-logic simulations.

Our ALU, like all ALU's, will be a combinational logic circuit. It will have two data input ports (each of which will accept a 4-bit binary data values[1]) and it will have a carry input. These data values are known as operands. Our ALU will be capable of operating on either one operand (e.g., performing a 1's complement operation), or two operands (e.g., performing the sum, A+B) and will produce a 4-bit result (plus a possible carry). The ALU will be controlled by a set of input control signals. These input control signals will determine which operation the ALU will perform.

In this laboratory exercise you will construct the ALU. It will be left as an exercise at the end of this lab to create a table that lists the operations the ALU performs for each set of input control signals. As you work through this laboratory assignment, if you focus on how each subcircuit works, and how they interact, then you will find that constructing the table at the end is much easier. The modules we create will be capable of performing either logical or arithmetic operations. It may help you in organizing your understanding if you divide the ALU operations into these two types: logical and arithmetic.

Task 3-1: Build the NOT/NEG Circuit

The first module we build will be capable of performing two different operations on one operand. Which operation we perform will be determined by the control signals we apply to the module. In Simulation Lab 1, you designed the two's complementing circuit shown in Figure 1. This circuit is capable of performing only the 2's-complement

[1] We choose 4-bit (rather than 32-bit) operands because the 4-bit circuitry is much less tedious to construct and debug. This simplicity will allow you to spend your time understanding the concepts we introduce rather than tediously building 32-bit versions of the subcircuits we will need.

operation; it has no control inputs that allow us change the function that it performs. Starting with this circuit, you will modify it as shown in Figure 2.

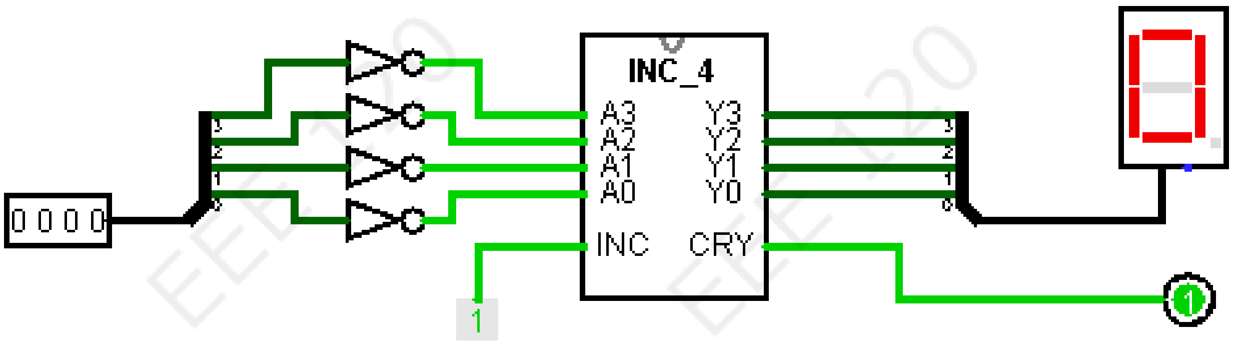

Figure 1. Two's complement circuit

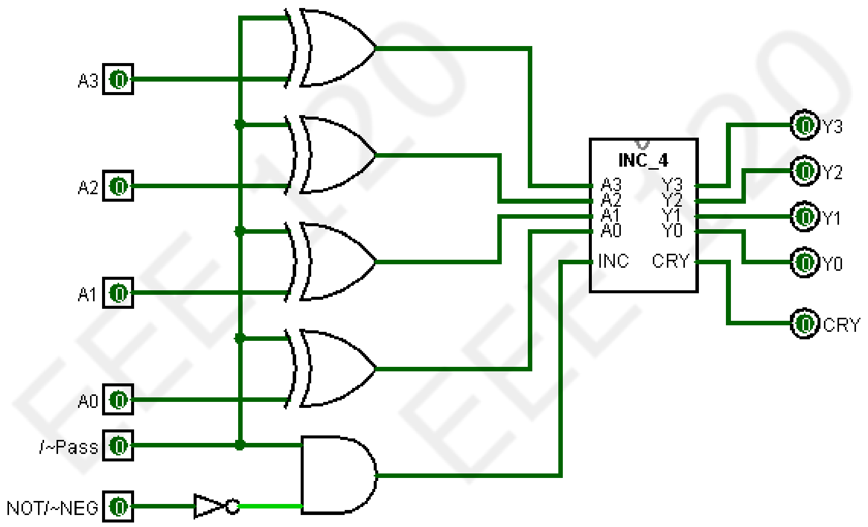

Figure 2. NOT/NEG circuit

The modifications shown in Figure 2 allow this circuit to do three things: it can perform the **2's complement operation** (i.e., negate arithmetically), perform the **1's complement** (i.e., logically complement each bit, also known as performing the NOT of each bit) or allow the input argument to **pass through** unscathed. In Figure 2, two control signals have been added to affect these controls: the ***/~Pass*** and the ***NOT/~NEG***. The notation chosen for these signals communicates two things: which operation the signal controls, and whether the signal is high or low when active. The '/~' notation means that the signal is low (0) when active. If the '/~' is not present, the signal is high when active. (If you understand the notation we are using, in many cases you will be able to determine the functions being performed by each circuit simply by inspecting the notation.)

Justify to yourself that the */~Pass* and *NOT/~NEG* signals work together to produce the operations shown in Table 1.

Table 1. NOT/NEG function definition table

/~Pass	NOT/~NEG	Function
0	0	Pass through
0	1	Pass through
1	0	Two's complement

		(Arithmetic negate)
1	1	One’s complement

Build and test the circuit shown in Figure 2. Each of the circuits we design in this lab will play a key role in your microprocessor; so it is important to select your tests to ensure that each of these circuits work correctly. Briefly mention in your lab template how you chose to test this circuit. It is not necessary to describe all of the input and output values you tested; instead, for each function (e.g., two’s complement, etc.) give one or two examples of input/output pairs you generated. After you are convinced that the circuit is working properly, imbed it in a subcircuit using the notation of Figure 3.

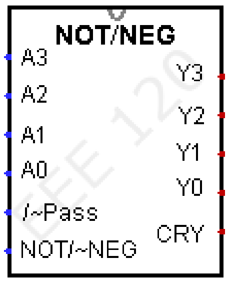

Figure 3. Subcircuit symbol for NOT/NEG circuit

Task 3-2: Build and Test the AND/ADD Circuit

The NOT/NEG circuit is a very simple version of an ALU, so simple that few would call it an ALU. Yet, it can do either an arithmetic operation, the NEGATE, or a logical operation, the NOT of 4 bits. It can also pass data through without operating on it. To justify calling our circuit an ALU, we will want to add more functions than just the three listed in Table 1. In particular, we would like to be able to perform an ADD and an AND of two 4-bit numbers, and we would like to retain the pass-through capability. In your previous laboratory exercises, you constructed all of the subcircuits we will need to build this circuit. Let's build this circuit in two stages; first let's build the AND/ADD circuit; then let's modify the circuit to gain the pass through capability.

Figure 4 shows the schematic of the AND/ADD circuit. (This figure uses the Logisim bus concept and notation[2]. If you are not familiar with this notation, watch the Logisim tutorial videos.) This circuit accepts two 4-bit operands, A and B, as input, which are fed to the 4-bit full-adder subcircuit (FA_4 found in your library) and to the four 2-input AND gates. The 4-bit output of the AND gates and adder is fed into a 2-to-1 4-bit mux (MUX_4 found in your library). This mux is used to select which result (ADD or AND) is routed to the output. (This is a typical example of how multiplexers are used in digital systems.) To control which result is routed to the output, the mux selector input is controlled by the ***AND/~ADD*** signal. The selector signal notation is consistent with the notation described earlier: when the *AND/~ADD* signal is low, the (arithmetic) addition result is routed to the output. When the *AND/~ADD* signal is high, the (logical) AND result is routed to the output.

To add the data pass-through capability to the circuit of Figure 4, we add a 4-bit 2-to-1 mux as the output stage as shown in Figure 5. One input to the output-stage mux is the AND/ADD-circuit output, the other is the A operand. The selector input of the output stage mux is controlled by the ***/~Pass*** signal. When */~Pass* = 0, the A operand appears at the output unscathed. When */~Pass* = 1, the output function, either ADD or AND, depends on the value of the *AND/~ADD* signal appears at the output. The operations performed by the AND/ADD circuit is summarized in Table 2.

[2] Use of a bus simplifies the construction and presentation of the circuit schematic by eliminating the need to provide a unique route for each signal line.

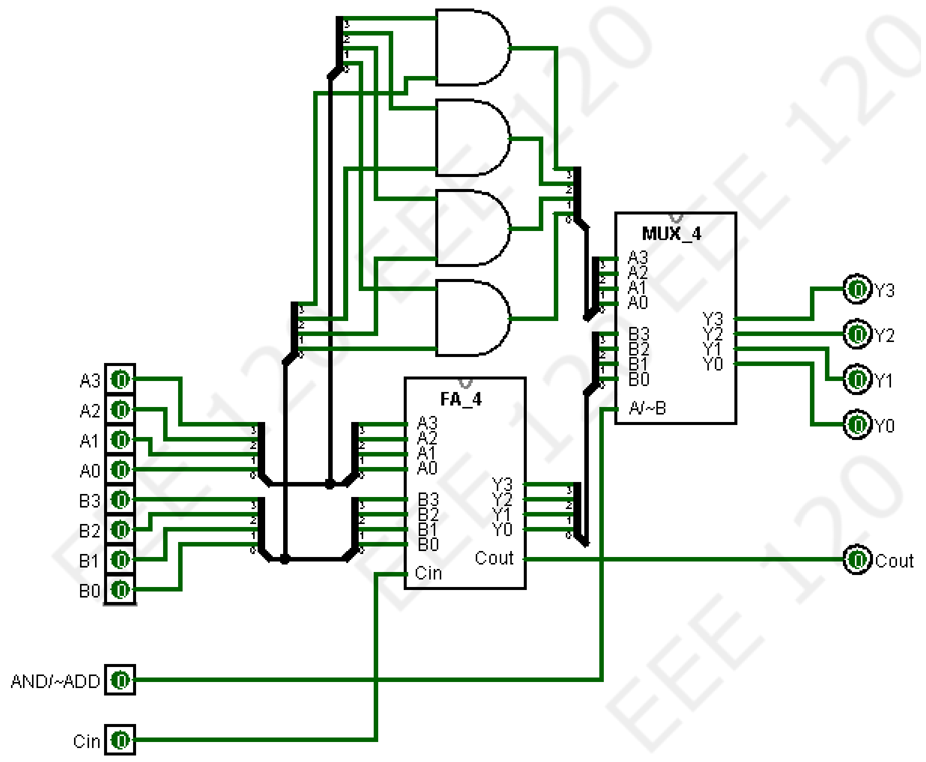

Figure 4. AND/ADD circuit

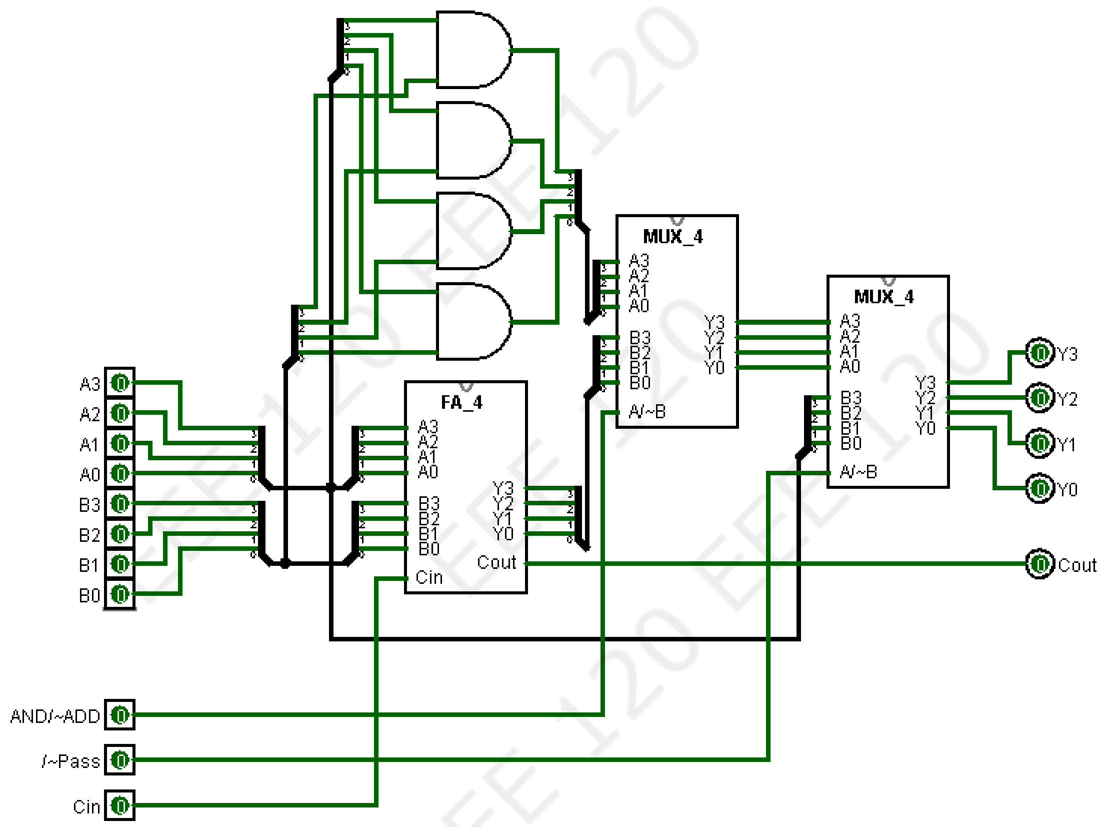

Figure 5. AND/ADD circuit with pass-through

Table 2. AND/ADD (with pass-through) function definition table

/~Pass	AND/~ADD	Function
0	0	Pass through
0	1	Pass through
1	0	ADD
1	1	AND

Using the subcircuits contained in your library, build and test the AND/ADD circuit with pass-through shown in Figure 5. Create a subcircuit for this circuit using the symbol in Figure 6. Briefly mention the procedure you used to test it. Justify why the results of the tests you chose make it likely that your circuit functions correctly.

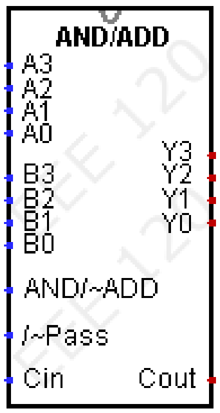

Figure 6. Subcircuit symbol for AND/ADD circuit

Task 3-3: Build and Test the ALU Circuit

The last stage in building the ALU is to combine the NOT/NEG and the AND/ADD subcircuits as shown in Figure 7 to create the complete ALU.

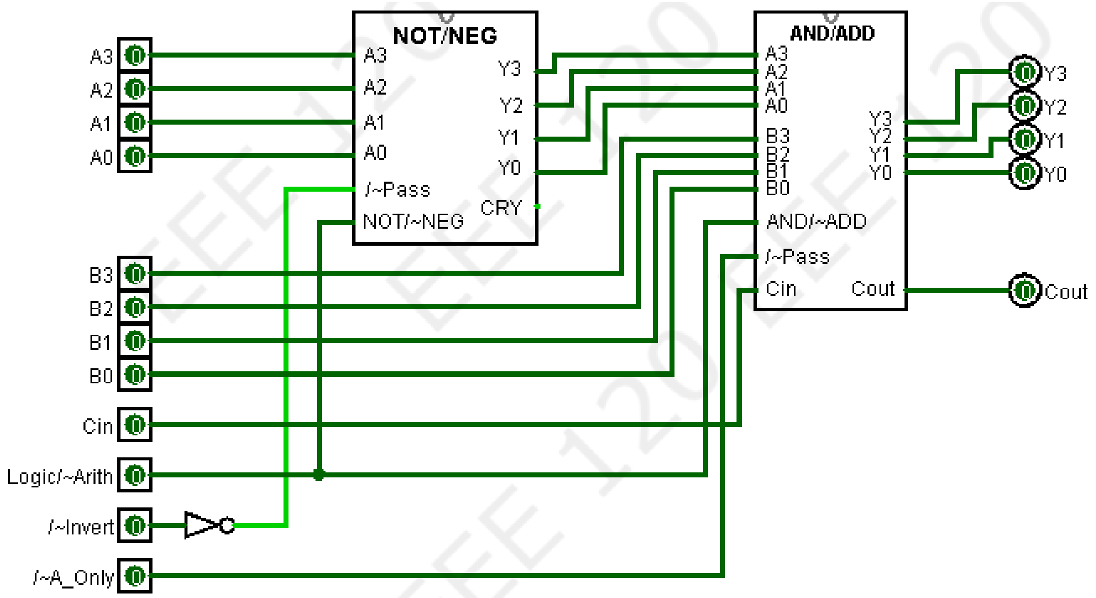

Figure 7. The complete ALU

This ALU contains three input control signals:

- The ***/~Invert*** signal, when active, indicates that some type of inversion, either a one's or two's complement is taking place.
- The ***/~A_Only*** signal, when active, indicates that only the A operand is being acted upon by the ALU.
- The ***Logic/~Arith*** signal operates so that when *Logic/~Arith* = 0, an arithmetic operation is performed. When *Logic/~Arith* = 1, a logical operation is performed by the ALU.

Once this notation is understood, you may be able to fill out the function definition table in Table 3 for this circuit even without reference to Figure 7. Consider the example filled-in in Table 3:

- */~A_Only* = 1 implies that the A and B operands are combined in some fashion.
- */~Invert* = 1 implies that A is not inverted (neither a one's nor a two's complement is performed.)
- *Logic/~Arith* = 0 implies the operation is arithmetic.

The only two-operand, noninverting arithmetic operation of which our ALU is capable is the sum, A + B.

Table 3. ALU function definition table

/~A_Only	Logic/~Arith	/~Invert	Function
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	Arithmetic Sum, A+B
1	1	0	
1	1	1	

Build the complete ALU circuit shown in Figure 7. Complete the ALU function definition table (Table 3) and include it in your lab template. Test the ALU and then imbed the ALU in a subcircuit using the notation shown in Figure 8. Describe in your lab template briefly how you tested the circuit. The ALU you have saved in your library is the one around which we will construct a microprocessor.

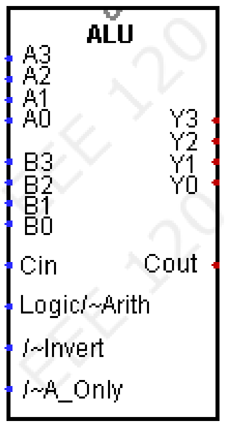

Figure 8. Subcircuit symbol for ALU circuit

The design methodology we have used to construct the ALU has followed the same pattern we established early on in these laboratory exercises: create subcircuits from complex circuits, then use these subcircuits to create more complex subcircuits, and so on to ever greater levels of complexity. This made designing the ALU easy. Imagine how difficult it would be to design an ALU using only the symbols for NAND gates and to manage the resulting schematic diagram!

In spite of the complexity of laying out large complex circuits at the gate level, many ALU's (or at least parts thereof) are designed just that way today. Rather than use a hierarchical design approach, a truth table is constructed of the complex function to be implemented and sophisticated software programs, using techniques like Karnaugh methods, find the optimum design. (Optimization must balance many competing criteria. Speed, as measured by propagation delay time, is one of these.) Whether or not hierarchical techniques are used in designing a component depends on a number of factors including the type of component to be designed, the need for optimization, and the availability of the necessary software to manage the optimization. In this laboratory manual, we will continue to use hierarchical design techniques because they are easier to use. Also, the resultant designs are easier maintain and easier to understand by ourselves and by others who may work with them. When we come to a component that may benefit from using more sophisticated design techniques, we'll point that out.

Task 3-4: Backup Your Files

Take a moment and back up your circuit files.