Simulation Lab 2: 4-Bit Full Adder, Multiplexer, Decoder & Buffer

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

- Use Logisim.
- Use Karnaugh maps.
- Have completed Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit.

Equipment: Personal computer and Logisim.

Objectives: In this laboratory exercise, you will build and debug combinational logic subcircuits that perform arithmetic operations and data routing using Logisim.

Outcomes: When you have completed the tasks in this experiment you will be able to design, build, test, debug, and imbed in a subcircuit, the following components:

- A 1-bit full adder.
- A 4-bit full adder.
- A 2-to-1 multiplexer.
- A 4-bit, 2-to-1 multiplexer.
- A 1-to-2 decoder.
- A 2-to-4 decoder.
- A 4-to-16 decoder.
- A 4-bit buffer.

Introduction

In Simulation Lab 1, you developed the increment circuit (which you will use in the program-counter portion of the microprocessor design) and the two's-complement circuit (which you will use as part of the arithmetic and logic unit [ALU] of the microprocessor design.) In this laboratory exercise you will continue constructing modules that will eventually be used in assembling the microprocessor. Our concern in this laboratory exercise is with circuits that can perform binary addition (the adder) and with circuits that control the flow of data through our system (the multiplexer and decoder). You will eventually use the data-flow-control circuits you create in this lab exercise (a 4-bit 2-to-1 multiplexer, and a 4-to-16 decoder) to make the microprocessor self-capable of routing data to appropriate locations. The binary-addition circuitry you will create (which is a 4-bit full adder) will contribute another piece to the ALU. You will also build a 4-bit three-state buffer to control different signal sources sharing a common communication bus. Each circuit you build will be modularized by imbedding it in a subcircuit and these modules will be used in subsequent labs to create more complex circuits. Using modules to create more complex modules is a powerful strategy that we will use throughout these simulation laboratory exercises.

As we progress through these laboratory exercises, you will notice that the burden of circuit design will be shifted to you. Instead of giving you a circuit schematic and asking you to build, debug and modularize it, we will slowly begin to ask you to do more of the design work yourself. Any design work you are asked to do will be well within your capabilities and will be supported by the lecture material, and your innate ability to recognize patterns.

Task 2-1: Design a Full Adder Using *NAND/NAND* Logic

Using the full-adder truth table, Table 1, write down the **canonical SOP** expressions for the C_{out} and SUM functions of a full adder. Using these canonical SOP expressions, build, test and debug the circuits that realize the C_{out} and SUM functions using **only NAND/NAND logic** with Logisim. (Remember: you will need to design two circuits: one for the SUM function and one for the C_{out} function. Both functions should share the same A, B and C_{in} inputs. You need to transform the canonical SOP expressions to use NAND/NAND logic. Hint: deMorgan's law will help. When you need the complement of a variable, you may use inverters, i.e., NOT gates, rather than using NAND gates connected as inverters.) Record the results of your validation tests in the form of a truth table in your lab template.

Table 1. Full Adder Truth Table

A	B	C_{in}	C_{out}	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Task 2-2: Build, Debug and Test a 1-Bit Full Adder

Build, debug and test a 1-bit full-adder circuit. Use **NAND/NAND logic** to implement your **minimal SOP** form for **the C_{out} function** (**use Figure 1 only as a guide since you need to implement C_{out} using NAND/NAND logic**) and construct **the SUM function** using a **3-input XOR gate**. When you use XOR gate in Logisim, specify its "Multiple-Input Behavior" as "**When an odd number are on**". Test your circuit. Record the results of these tests (in the form of a truth table) in your lab template.

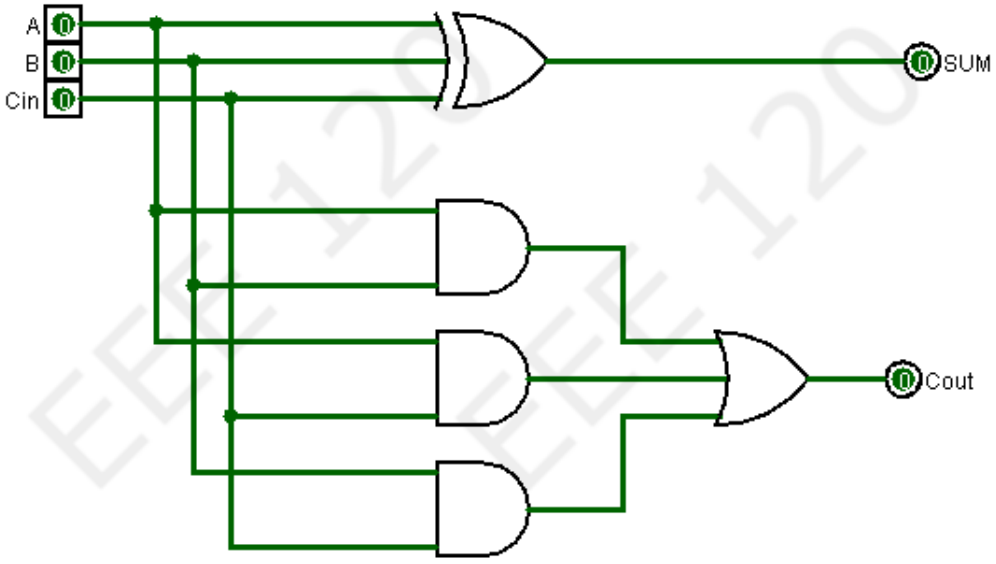


Figure 1. SOP implementation of a 1-bit full adder

Imbed your 1-bit full adder in a subcircuit (see Figure 2), label the subcircuit FA_1.

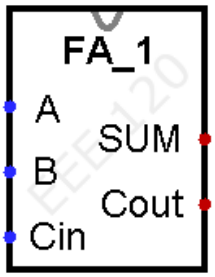


Figure 2. Subcircuit symbol for a 1-bit full adder

Task 2-3: Design, Build and Test a 4-Bit Full Adder

Using Figure 3 (2-bit full adder) as a guide, design a 4-bit full adder. The 4-bit full adder should accept two 4-bit numbers and a carry as input, and give one 4-bit sum and a 1-bit carry as output. Build, test and debug the 4-bit full adder. For naming inputs and outputs, see Figure 4 for reference. Test the circuit using a **multi-bit input pin** and a **hex digit display**. Include the test results in the form of a truth table in your lab template. **DO NOT** test all input combinations. It is left to you to decide what constitutes a sufficient set of tests to make it likely that the 4-bit full-adder circuit is operating correctly. (One set of test inputs is not enough.) Justify in your lab template why successful completion of your tests proves beyond a reasonable doubt that your circuit is operating correctly. Once you are satisfied that the circuit is working correctly, imbed the 4-bit adder in a subcircuit, label it "FA_4", as shown in Figure 4.

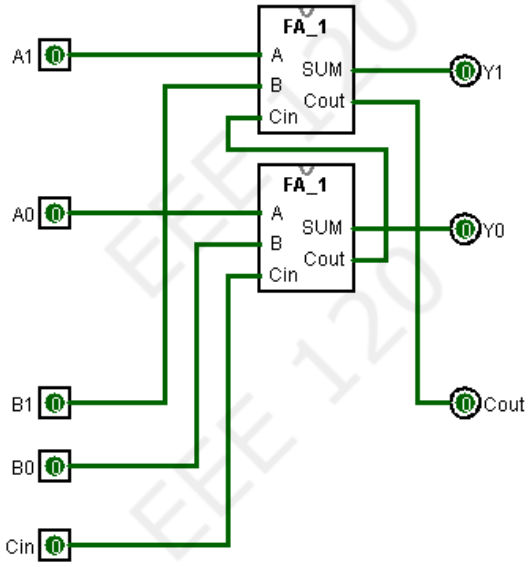


Figure 3. 2-bit full adder

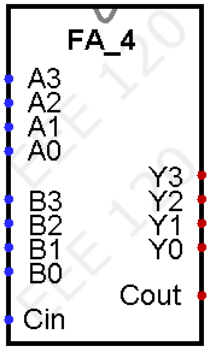
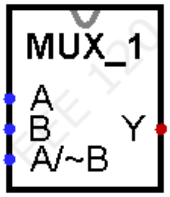


Figure 4. Subcircuit symbol for a 4-bit full adder

Task 2-4: Design, Build and Test a MUX Using NAND/NAND Logic

Notation: We will use the following naming convention to represent a signal that is active when it is low, i.e., an active low signal: ***/~***. Using this notation, the active low signal *Arith* would be represented by */~Arith*. Where a signal performs one operation when high (operation Y) and another operation when it is low (Z) we will label the signal *Y/~Z*. Using this notation, the complement of an active low signal is represented as: *(/~Arith)'* .

When we discuss the architecture of the microprocessor in later laboratory experiments, we will find that hardware is needed that allows the executing program to select the route along which data flows. One component that we will need to perform this data routing function is the multiplexer, or MUX. A MUX is a device that can be controlled to route one of its many input signals to its sole output. A truth table and symbol for the 1-bit 2-to-1 MUX you will build is shown in Figure 5. The control/select input, 'A/~B', indicates that the output is identical to the A input when the select signal is high (1) and identical to the B input when the select signal is low (0).



A/~B	A	B	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 5. Subcircuit symbol and truth table for a 1-bit 2-to-1 MUX

Using the Karnaugh Map methods you learned in lecture, design, build and test a **NAND/NAND** implementation of the 1-bit 2-to-1 MUX. You will want to base your design on the minimal SOP form of the equation that defines the MUX output. (When you need the complement of a variable or a function, you may use inverters, i.e., NOT gates, rather than using NAND gates connected as inverters.) Test the circuit and record the results as a truth table in your lab template. Once you are convinced that the circuit is working correctly, imbed it in a subcircuit, label it "MUX_1" as shown in Figure 5.

Task 2-5: Build a 2-Input 4-Bit Multiplexer

Because our microprocessor operates on 4-bit numbers, it will be necessary to construct a **4-bit, 2-to-1 MUX**. The 4-bit MUX should use a single control/select line to select one of two 4-bit numbers and the selected 4-bit number should appear on the output of the MUX. A 2-bit 2-to-1 MUX is shown in Figure 6. Expand on this figure to design

your 4-bit MUX. Build and test the 4-bit MUX. Once you are satisfied that it is working correctly, imbed it in a subcircuit, and label the subcircuit “MUX_4” (Figure 7).

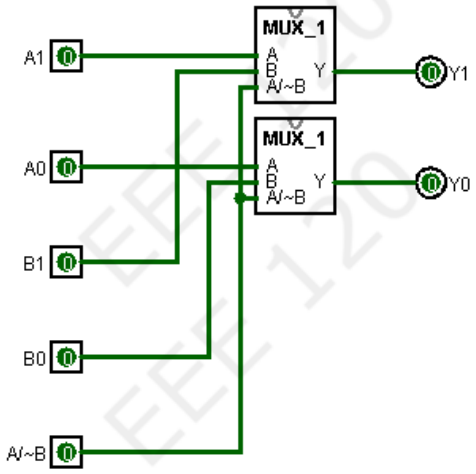


Figure 6. 2-bit 2-to-1 MUX

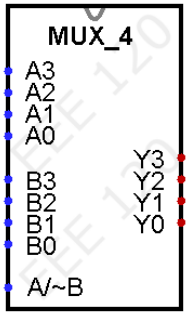
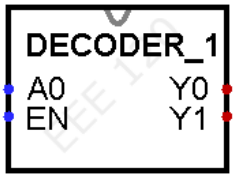


Figure 7. Subcircuit symbol for a 4-bit 2-to-1 MUX

Task 2-6: Build and Test a 1-to-2 Decoder Using NAND/NAND Logic

The other piece of hardware our microprocessor will need is a decoder, and it will be used to control access to memory. The truth table and subcircuit symbol for the 1-bit 1-to-2 decoder you will build is shown in Figure 8. When the EN input is low, the decoder is disabled, so both Y0 and Y1 output zeroes; when the EN input is high, the decoder decodes the binary number A0 to a decimal number, and the Y output with subscript corresponding to that decimal number will output a one, with the rest of Y’s outputting zeroes.



EN	A0	Y0	Y1
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

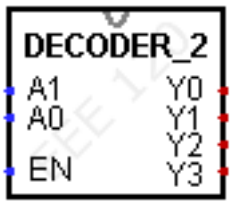
Figure 8. Subcircuit symbol and truth table for a 1-bit 1-to-2 decoder

Build, and test the 1-bit, 1-to-2 decoder using only **NAND/NAND** logic. You will want to base your design on the minimal SOP form of the equations that define the decoder outputs Y0 and Y1. (When you need the complement

of a variable or a function, you may use inverters, i.e., NOT gates, rather than using NAND gates connected as inverters.) Test the circuit and record the results as a truth table in your lab template. Once you are convinced that the circuit is working correctly, imbed it in a subcircuit, label it "DECODER_1" as shown in Figure 8.

Task 2-7: Build and Test a 2-to-4 Decoder Using NAND/NAND Logic

The microprocessor you will build will need a larger decoder than that constructed above. Let's start by building a 2-to-4 decoder. A 2-to-4 decoder has the function definition table shown in Figure 9. Just like a 1-to-2 decoder, when the EN input is low, the decoder is disabled, so all Y's output zeroes; when the EN input is high, the decoder decodes the binary number (A1A0) to a decimal number, and the Y output with subscript corresponding to that decimal number will output a one, with the rest of Y's outputting zeroes.



A1	A0	Y0	Y1	Y2	Y3
0	0	EN	0	0	0
0	1	0	EN	0	0
1	0	0	0	EN	0
1	1	0	0	0	EN

Figure 9. Subcircuit symbol and function definition table for a 1-bit 2-to-4 decoder

Build, and test the 1-bit, 2-to-4 decoder using only **NAND/NAND** logic. (When you need the complement of a variable or a function, you may use inverters, i.e., NOT gates, rather than using NAND gates connected as inverters.) Imbed your circuit in a subcircuit labeled "DECODER_2" as shown in Figure 9.

Task 2-8: Design, Build & Test a 4-to-16 Decoder Using 2-to-4 Decoders

An alternate scheme for building the 1-bit 2-to-4 decoder using primitive logic gates as in Task 2-7 is to build the decoder using the 1-bit 1-to-2 decoders built in Task 2-6. Justify to yourself that the circuit of Figure 10 will function as a 1-bit 2-to-4 decoder. Also prove to yourself that when the EN input line is inactive (i.e., 0), all output lines will be inactive.

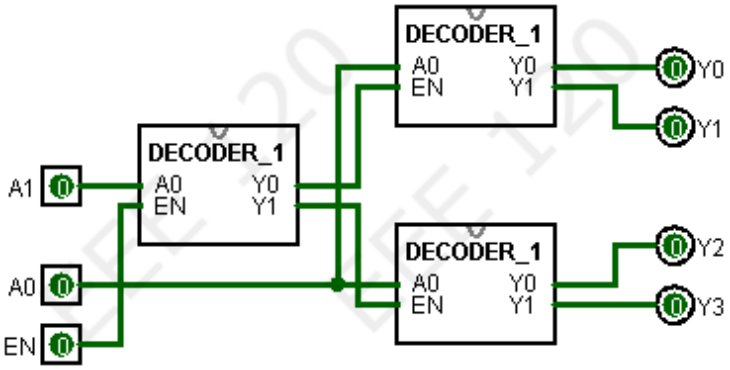


Figure 10. Schematic for a 2-to-4 decoder constructed from 1-to-2 decoders

Using the technique illustrated in Figure 10, design, build, and test a 1-bit 4-to-16 decoder using only the 1-bit 2-to-4 decoder subcircuits you constructed in Task 2-7. Imbed your circuit in a subcircuit labeled "DECODER_4" (see Figure 11 for the subcircuit symbol to use).

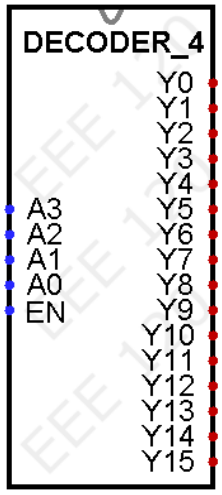


Figure 11. Subcircuit symbol for a 1-bit 4-to-16 decoder

Task 2-9: Build a 4-Bit Buffer

A buffer is a circuit with a data and a control input. The control input controls whether the data input signal is allowed to propagate to buffer's output. The buffer circuit uses the three-state device described in lecture. The three-state buffer in Logisim (under Gates-> Controlled Buffer) has a single active high enable as shown in Figure 12. When EN = 0, the output Y is in high impedance state; when EN = 1, the output Y is the same as input A.

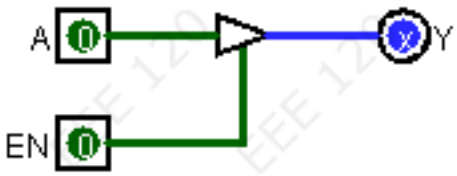


Figure 12. Tri-state buffer in Logisim

In our microprocessor, we will need to have more than two sources share a common communication bus. If the outputs of many three-state buffers are wired together and all but one are in the high-impedance state, then the active buffer will control the value measured at the output; however, if the outputs do not 'take turns' properly, (i.e., if more than one of the three-state buffers is active) the potential exists for a data conflict. Therefore, it is necessary when using three-state devices to be sure that all outputs but one are in the high-impedance state. In our microprocessor, we will use buffer circuits to allow eight 4-bit memory locations to share a common communication bus. To allow these memory circuits to share a common 4-bit communication bus, we will need to construct a 4-bit buffer circuit made of four three-state devices that share a common enable signal.

For our microprocessor design we will need a 4-bit buffer with two active high enables; that is, both of the enables must be high to enable the four three-state outputs. You can design such a control scheme by using a 2-input AND gate and routing its output to the active high enable inputs of each of four three-state devices. Design and test this circuit, enclose it in a subcircuit (see Figure 13 for the subcircuit symbol to use). Mention briefly in your lab template how you tested your circuit and comment on why you believe that your tests are a reliable indicator that your circuit is operating correctly.

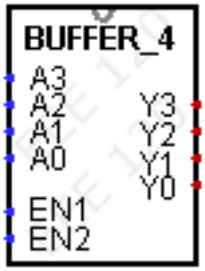


Figure 13. Subcircuit symbol for a 4-bit buffer circuit

Task 2-10: Backup Your Files

Take a moment and back up your circuit files.