LOGISIM-EVOLUTION LAB MANUAL

GEORGE SELF

July 2019 – Edition 4.0

George Self: *Logisim-Evolution Lab Manual*

PREFACE

I have taught CIS 221, *Digital Logic*, for Cochise College since about 2003 and enjoy working with students on this topic. From the start, I wanted students to work with labs as part of our studies and actually design circuits to complement our theoretical instruction. As I evaluated circuit design software I had three criteria:

- **Open Educational Resource (OER)**. It is important to me that students use software that is available free of charge and is supported by the entire web community.
- **Platform**. While most of my students use a Windows-based system, some use Macintosh and it was important to me to use software that is available for both of those platforms. As a bonus, most OER software is also available for the Linux system, though I'm not aware of any of my students who are using Linux.
- **Simplicity**. I wanted to use software that was easy to master so students could spend their time understanding digital logic rather than learning the arcane structures of a simulation language.

I originally wrote a number of lab exercises using *Logisim*, but the creator of that software, Carl Burch, announced that he would quit developing it in 2014. Because it was published as an open source project, a group of Swiss institutes started with the *Logisim* software and developed a new version that integrated several new tools, like a chronogram, and released it under the name *Logisim-Evolution* .

It is my hope that students will find these labs instructive and the labs enhance their learning of digital logic. This lab manual is written with LaTeX and published under a Creative Commons Zero license with a goal that other instructors can modify it to meet their own needs. The source code can be found at my personal GITHUB page and I always welcome comments that will help me improve this manual.

—George Self

BRIEF CONTENTS

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ALU	Arithmetic Logic Unit
BCD	Binary Coded Decimal
CPU	Central Processing Unit
IC	Integrated Circuit
OER	Open Educational Resource
PLA	Programmable Logic Array
RAM	Random Access Memory
ROM	Read Only Memory
TTL	Transistor-Transistor Logic

Part I

INTRODUCTION TO LOGISIM-EVOLUTION

Logisim-Evolution is used to create and test simulations of digital circuits. This part of the lab manual includes only one lab designed to introduce *Logisim-Evolution* and teach the fundamentals of using this application.

1 INTRODUCTION TO LOGISIM-EVOLUTION

1.1 PURPOSE

This lab introduces the *Logisim-Evolution* logic simulator, which is used for all lab exercises in this manual.

1.2 PROCEDURE

1.2.1 *Installation*

Logisim-Evolution is a Java application, so a Java runtime environment will need to be installed before using the application. Many students who are taking a digital logic class already have a Java runtime on their computer and can skip this step, but those who do not will need to install the Java runtime. That process is not covered in this manual but information about installing the Java runtime environment is available at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. It can be confusing to know which version of Java to download but students working on the labs in this manual only need the runtime, called *JRE* on the website. Students who are also in programming classes will likely already have the runtime as part of the Java Developer's Kit (JDK). It can be tricky testing the Java installation since the Chrome, Firefox, and Edge browsers will not run Java apps, but students can open a command prompt and enter `java -version` to see what version of Java their computers are running, if any.

Logisim-Evolution (`https://github.com/reds-heig/logisim-evolution`) is available as a free download. Visit the website and about halfway down the page find a section named "Running logisim-evolution." Click the "here" link at the end of the first sentence in that section.

Since the *Logisim-Evolution* file is a Java application, it does not need to be installed like most software. To start *Logisim-Evolution* , double-click the *Logisim-Evolution* shortcut. That will start Java and then run the *Logisim-Evolution* application. Also, *Logisim-Evolution* will not need to be uninstalled when it is no longer needed since it is not actually installed, the *Logisim-Evolution* file can simply be deleted.

1.2.2 *Beginner's Tutorial*

Logisim-Evolution comes with a beginner's tutorial available in HELP -> TUTORIAL. That tutorial only takes a few minutes and introduces

students to the major components of the application. Students should complete that tutorial before starting this lab.

1.2.3 *Logisim-evolution Workspace*

Start *Logisim-Evolution* by double-clicking its icon. The initial *Logisim-Evolution* window will be similar to Figure 1.1.

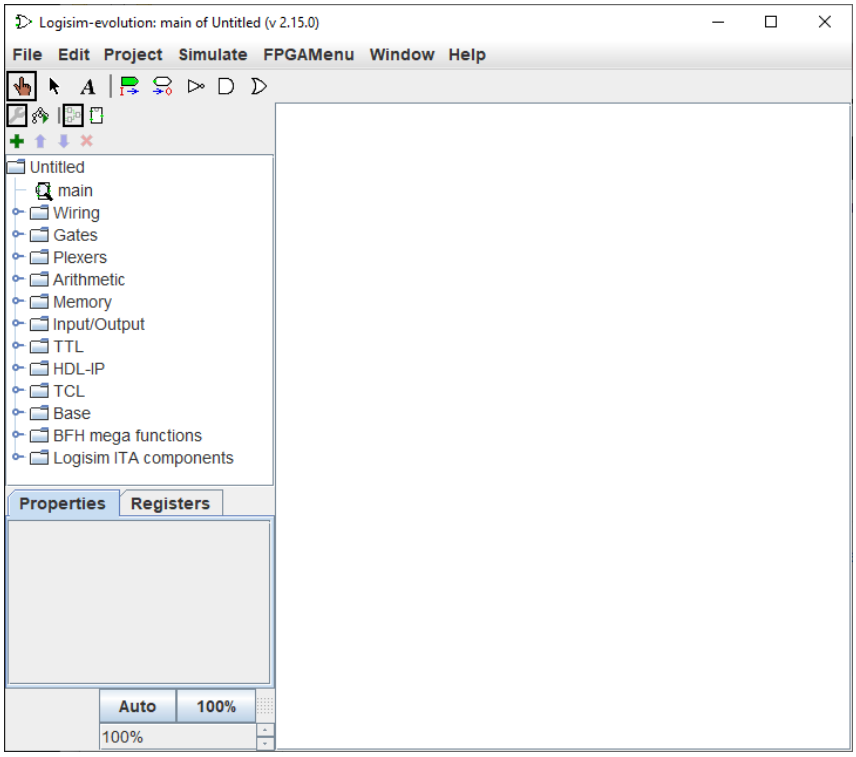

Figure 1.1: Logisim-evolution Initial Screen

The *Logisim-Evolution* space is divided into several areas. Along the top is a text menu that includes the types of selections found in most programs. For example, the "File" menu includes items like "Save" and "Exit." The "Edit" menu includes an "Undo" option that is useful. In later labs, the various options under "Project" and "Simulate" will be described and used. Items in the "FPGAMenu" are beyond the scope of this class and will not be used. Of particular importance at this point is "Library Reference" in the "Help" menu. It contains information about every logical device available in *Logisim-Evolution* and is very useful while using those components in new circuits.

Under the menu bar is the Toolbar, which is a row of eight buttons that are the most commonly used tools in *Logisim-Evolution* :

- **Pointing Finger**: Used to "poke" and change input values while the simulator is running.

- **Arrow**: Used to select components or wires in order to modify, move, or delete them.
- **A**: Activates the Text tool so text information can be added to the circuit.
- **Green Input Port**: Creates an input port for a circuit.
- **White Output Port**: Creates an output port for a circuit.
- **NOT Gate**: Creates a NOT gate.
- **AND Gate**: Creates an AND gate.
- **OR Gate**: Creates an OR gate.

The Explorer Pane is on the left side of the workspace and contains a folder list. The folders contain "libraries" of components organized in a logical manner. For example, the "Gates" folder contains various gates (AND, OR, XOR, etc.) that can be used in a circuit. The four icons across the top of the Explorer Pane are used for advanced operations and will be covered as they are needed.

The Properties panel on the lower left side of the screen is where the properties for any selected component can be read and set. For example, the number of inputs for an AND gate can be set to a specific number.

The drawing canvas is the largest part of the screen. It is where circuits are constructed and simulated.

1.2.4 *Simple Multiplexer*

Do not be concerned with the exact placement of components on the drawing canvas. They can be rearranged as the build progresses.

A multiplexer is used to select which of two or more inputs will be connected to a single output. For this lab, a simple two-input, one-bit multiplexer will be built. It is understood that students will not know the significance of a multiplexer at this point in the class, but the purpose of this lab is to use *Logisim-Evolution* to build a simple circuit and a multiplexer serves that purpose well.

Start by clicking the *And* button on the toolbar and placing two `AND` gates on the canvas. The canvas should resemble Figure 1.2

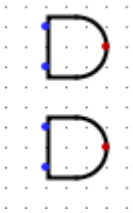

Figure 1.2: Two AND Gates

Click one of the AND gates to select it and observe the various properties available for that gate, as seen in Figure 1.3. The default values do not need to be changed for this circuit; however, all circuits in this manual use the "Narrow" gate size in order to make the circuit fit the screen better. The other properties will be explained as they are needed.

Properties | Registers

Selection: AND Gate

VHDL	Verilog
Facing	East
Data Bits	1
Gate Size	Narrow
Number Of Inputs	2
Output Value	0/1
Label	
Label Font	SansSerif Bold 16
Negate 1 (Top)	No
Negate 2 (Bottom)	No

Figure 1.3: AND Gate Properties

The outputs of the two `AND` gates need to be combined with an `OR` gate. Add an `OR` gate as illustrated in Figure 1.4.

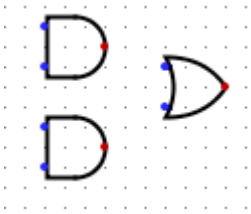

Figure 1.4: OR Gate Added to Circuit

The top input for the first `AND` gate needs two `NOT` gates (inverters) so the two `AND` gates can function as on/off switches. This is a rather common digital logic construct and when the circuit is complete it will become clear how the switching function works.

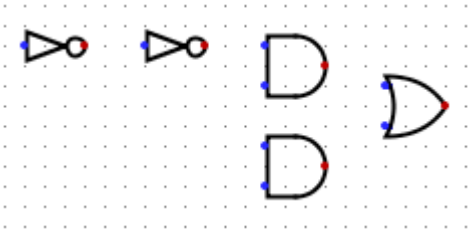

Figure 1.5: Two NOT Gates Added to Circuit

All inputs and outputs need to be added as in Figure 1.6. Note: inputs are square and outputs are round. The *Label* property for each input and output should be specified as in the figure. The pins are labeled according to their function in the circuit. Pin *Sel* carries a signal that selects which input to connect to the output, pins *In1* and *In2* are the two inputs, and pin *Out1* is the output. Note: output pins display a blue-colored X until they are actually wired to some device like the `OR` gate in the illustration.

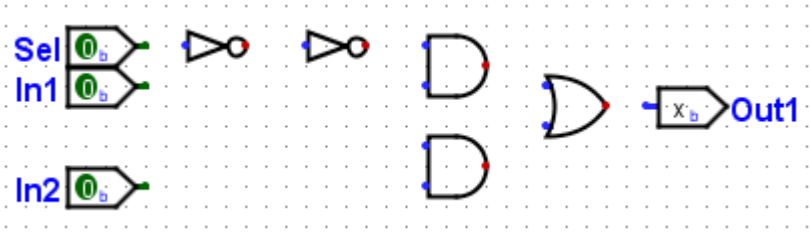

Figure 1.6: Inputs and Output Added

Finally, connect each device with a wire by clicking on the various ports and dragging a wire to the next port. To start the wire in the middle of the two NOT gates click the wire connecting those gates and drag downward. Wires will automatically "bend" one time but to get two bends, like between the output of an `AND` gate and the input of the `OR` gate, click-and-drag the wire from the output of the `AND` gate to a spot a short distance in front of that same gate, then release the mouse button and then immediately click again to start a new wire that will "bend" to the input of the `OR` gate. Only a little practice is needed to master this wiring technique.

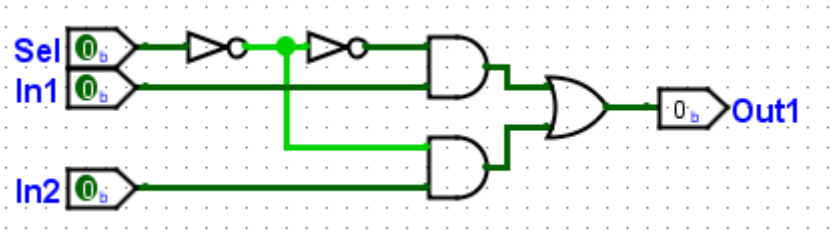

Figure 1.7: Circuit Wiring Added

To operate the circuit in a simulator, click the *Pointing Finger* and "poke" the various inputs. If it is working properly, when the *Sel* input is high then the value of *In2* should be transmitted to the output, but when *Sel* is low then the value of *In1* should be transmitted to the output. This circuit is used to select one of two inputs to be transmitted to the output.

1.2.5 *Identifying Information*

Before finishing, add standard identification information near the top left corner of the circuit using the text tool (the *A* button on the toolbar). That information should include the designer's name, the lab number and circuit name, and the date. Standard identification information for this lab would look like this:

```
George Self
Lab 01: 2-Way, 1-Bit multiplexer
February 13, 2018
```

The font properties in Figure 1.8 have been set to bold and a large size to make the text easier to read.

Note that *Logisim-evolution* will automatically center text in a new box, so text boxes will need to be aligned after they have been created. To align the text boxes, click the *Arrow* tool and use it to drag the boxes to their desired location. The completed circuit should look like Figure 1.8.

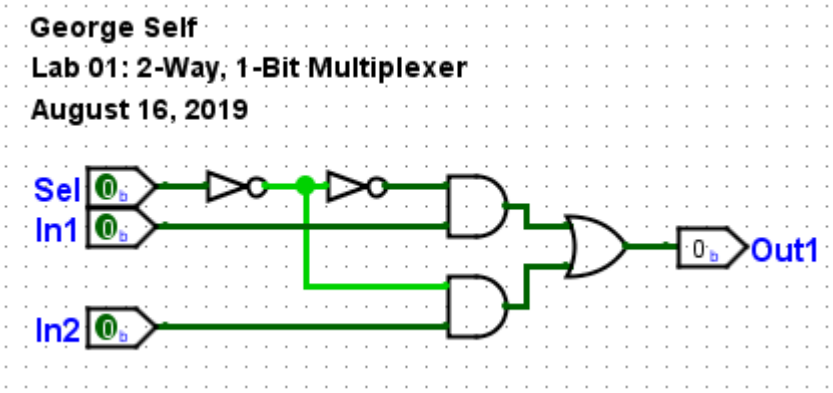

Figure 1.8: Simple multiplexer

1.3 DELIVERABLE

The purpose of this lab is to install and test the *Logisim-evolution* system and become comfortable creating a digital logic circuit.

To receive a grade for this lab, create the Simple Multiplexer as defined in this lab, be sure the standard identifying information is at the top left of the circuit, and then save the file with this name: `Lab01_Intro`. Submit that circuit file for grading.

Part II

THEORY

THEORY exercises are designed to provide practice with simple logic circuits in order to both develop skill with *Logisim-Evolution* and illustrate the foundations of digital logic theory.

2 ADDER

2.1 PURPOSE

This lab builds a full 1-bit adder, but the intent is to continue to familiarize students with *Logisim-Evolution* and how basic arithmetic functions can be completed using simple gate-level logic. Additionally, this lab develops an automated testing system that will be used to test future lab submissions.

2.2 PROCEDURE

2.2.1 *Half-Adder*

Open Logisim and start a new project. In *Logisim-Evolution* circuits can contain any number of sub-circuits. Subcircuits fill the same role in a physical circuit as a function or procedure fills in a software project. A new subcircuit can be added to a circuit by clicking PROJECT -> ADD CIRCUIT. Name the new circuit `Half_Adder`. Open the new subcircuit by double-clicking its name in the Explorer Pane.

Because this is a new subcircuit, the drawing canvas is blank. A half-adder is a circuit that will add two input bits. Because it is possible to add two bits and generate a carry, the half-adder must allow for a carry out bit. Figure 2.1 is the circuit diagram for a half-adder.

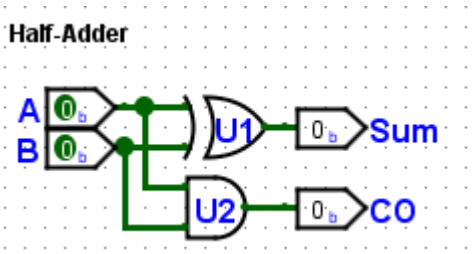

Figure 2.1: Half-Adder

This is fairly easy to build. Component `U1` is an `XOR` gate and `U2` is an `AND` gate. There are two inputs and two outputs and all of the devices should be connected as shown. To test the circuit, whenever input *A* and input *B* are different the *Sum* should be high and when input *A* and input *B* are both high the *CO* (Carry Out) bit should also be high.

2.2.2 *Full Adder*

A full adder takes two input bits and adds them, like a half-adder, but it also includes the logic necessary to input or output a carry bit so it can be cascaded with other adders. Figure 2.2 is the logic diagram for a full adder.

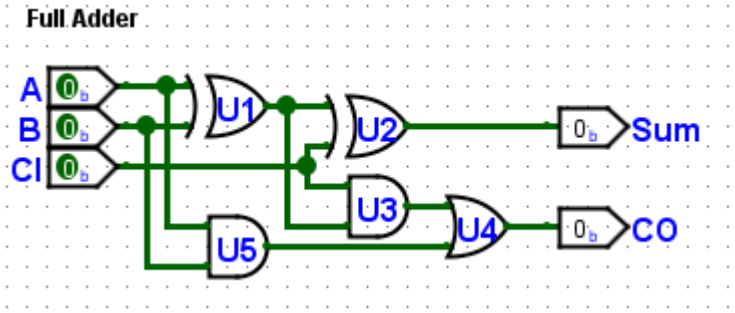

Figure 2.2: Full Adder

Create a new subcircuit named `Full_Adder` and wire all of the components as illustrated in Figure 2.2. Notice that `U1` and `U2` are `XOR` gates, `U3` and `U5` are `AND` gates, and `U4` is an `OR` gate. There are also three inputs and two outputs.

2.2.3 *Main Circuit*

In most *Logisim-Evolution* projects, the `main` circuit is used to provide a nice user interface for the project. Typically, various subcircuits are dropped onto the main circuit canvas and various inputs and outputs are wired to them. A user can then test the project without worrying about the details of the subcircuits.

For this project, open the `main` circuit by double-clicking its name in the Library panel. Click one time on the `Full_Adder` subcircuit then move the mouse over the drawing canvas. Notice that the mouse pointer has changed into a representation of the subcircuit. Drop that subcircuit anywhere on the drawing canvas and then wire the various inputs and outputs as shown in Figure 2.3.

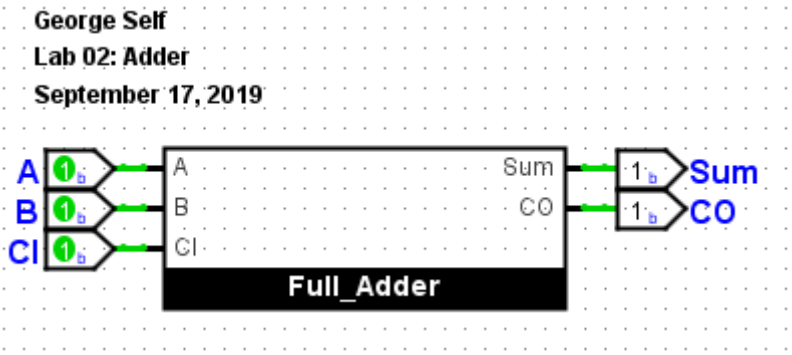

Figure 2.3: Full Adder

This circuit can be tested by using the *poke* tool and entering these values. After each line, check to see that the outputs are correct.

Inputs			Outputs	
A	B	CI	Sum	CO
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Table 2.1: Test Vector For Full Adder

2.2.4 *Automated Testing*

While it is possible to use the *poke* tool and check the outputs for various input combinations, as digital logic circuits become more complex it is important to automate the testing process so no input combinations are overlooked. *Logisim-Evolution* includes a SIMULATE -> TEST VECTOR feature that is used for automating circuit testing.

The first step in using automatic testing is to create a *Test Vector* file. This is a simple *.txt* file that can be created in any text processor, like *Notepad*. The format for a test vector is fairly simple.

Do not use a word processor to create the Test Vector since that would add unneeded codes for things like fonts and margins.

- Every line is a single test of the circuit, except the first line.
- The first line defines the various inputs and outputs being tested.
- Any line that starts with a hash mark (#) is a comment and is ignored.

Following is the test vector file used to test the `main` circuit.

```
1  # Test vector for Adder
2  CI A B Sum CO
3  0 0 0 0 0
4  0 0 1 1 0
5  0 1 0 1 0
6  0 1 1 0 1
7  1 0 0 1 0
8  1 0 1 0 1
9  1 1 0 0 1
10 1 1 1 1 1
```

Following is an explanation for the *Test vector for Adder* file.

LINE 1 This is just the title of the file. Because this line starts with a hash (#) it is a comment and will be ignored by *Logisim-Evolution* .

LINE 2 This line lists all of the inputs and outputs in the circuit under test. In this case, there are three inputs, *CI*, *A*, and *B*, along with two outputs, *Sum* and *CO*. *Logisim-Evolution* is able to determine whether the pin is an input or output from its properties.

LINE 3 This line contains the first test for the circuit. This line specifies that *Logisim-Evolution* make *CI*, *A*, and *B* equal to zero and then check to be certain that *Sum* and *CO* are also zero.

OTHER LINES All other lines set the three input bits and specify the expected response in the output bits.

To start a test, click SIMULATE -> TEST VECTOR. The window illustrated in Figure 2.4 opens.

Figure 2.4: Test Vector Window

Click the *Load Vector* button at the bottom of the window and load the test vector file. The test will automatically start and *Logisim-Evolution* will report the results, like in Figure 2.5.

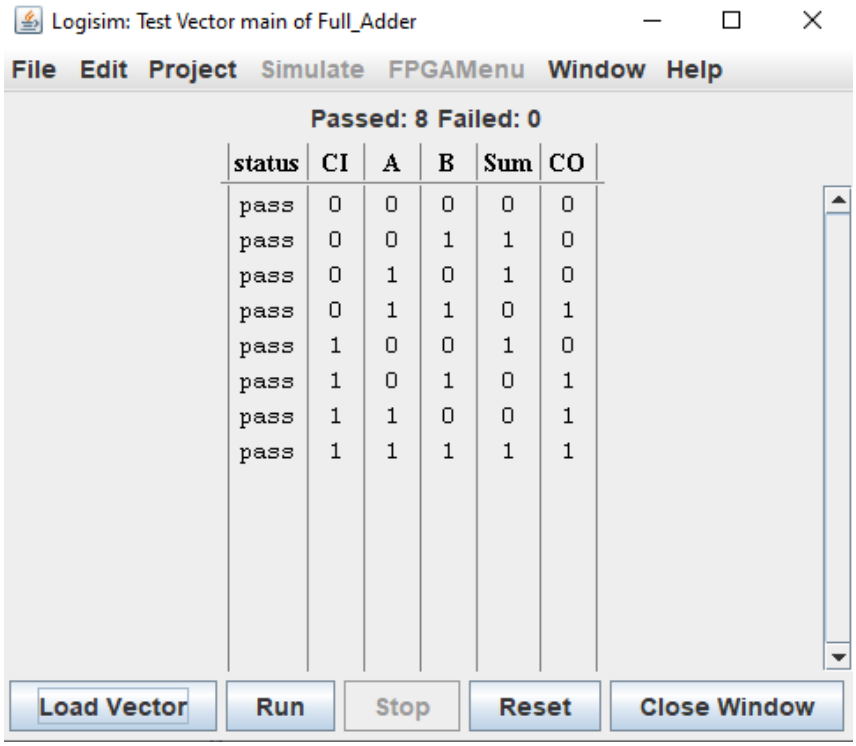

Figure 2.5: Test Completed

An error was intentionally added to the test vector file to generate a failed test.

The test indicates all 8 lines passed and zero failed so it could be reasonably concluded that the circuit is functioning properly. Figure 2.6 illustrates a failed test. The circuit designer would then need to troubleshoot to determine what went wrong with the circuit.

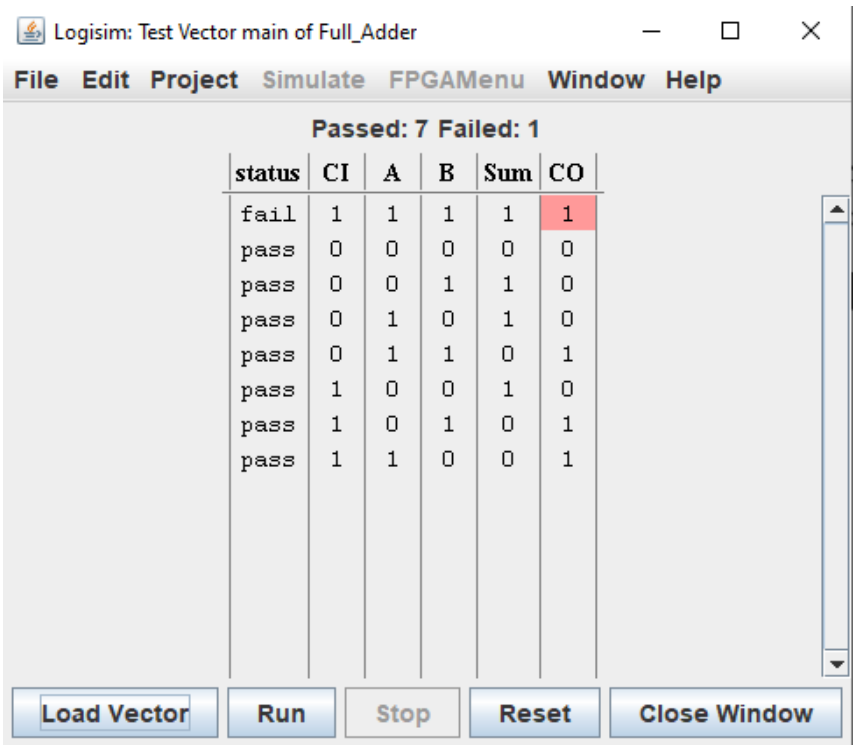

Figure 2.6: Test Failure

Note: the test vector files for all labs are made available to students so they can check their work prior to submitting them.

2.3 DELIVERABLE

To receive a grade for this lab, build both the half-adder and full adder and then add the full adder to the `main`. It is important to ensure the input and output pin names are the same as in the lab instructions since a test vector will be used to check the circuit. Be sure the standard identifying information is at the top left of the `main` circuit, similar to:

```
George Self
Lab 02: Adder
September 17, 2019
```

Save the file with this name: *`Lab02_Adder`* and submit that file for grading.

3

ARITHMETIC OPERATIONS

3.1 PURPOSE

This lab develops an arithmetic unit that includes eight different arithmetic functions using *Logisim-Evolution* library components. This device will eventually be used as part of the Arithmetic Logic Unit (ALU) in Lab 8. This device will have two inputs, labeled *A* and *B*, and will output the following calculations.

1. -1
2. $A - 1$
3. $A + B$
4. $A - B$
5. $AB - 1$
6. $AB' - 1$
7. $A + A$
8. $A + 1$

3.2 PROCEDURE

Start a new *Logisim-Evolution* project and create a subcircuit named `arithmetic` that will eventually contain the entire arithmetic unit. Begin the build by adding eight devices in the subcircuit as in Figure 3.1. *Notes: exact device placement is not important at this point since they can be repositioned as necessary; however, they should be in the correct order on the subcircuit. Also, all of the devices, inputs, and outputs need to be set for eight data bits in the properties panel.*

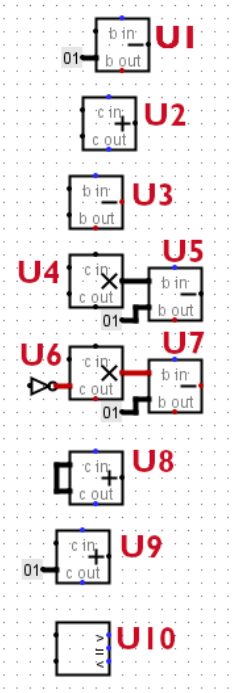

Figure 3.1: Placing the Arithmetic Components

Here are the devices in Figure 3.1. *Note: the device numbers were added to the illustration as an aid for the following discussion; they will not be present in the Logisim-Evolution subcircuit.*

- U1-Subtractor (*Arithmetic* library). This device subtracts the bottom input from the top input on its east side and sends the result to the output on its west side.
- Constant (*Wiring* library). Subtractor *U1* is intended to supply the *A-1* output and the "01" on its bottom input is a constant used in this calculation. It should be placed near the bottom input for subtractor *U1* and its properties set for Facing: East, Data Bits: 8, Value: 0x1 (this means hexadecimal 1).
- U2-Adder (*Arithmetic* library). This device adds the top and bottom inputs on its east side and sends the result to the output on its west side.
- U3-Subtractor (*Arithmetic* library). This device subtracts the bottom input from the top input on its east side and sends the result to the output on its west side.
- U4-Multiplier (*Arithmetic* library). This device multiplies the top and bottom inputs on its east side and sends the result to the output on its west side.
- U5-Subtractor (*Arithmetic* library). This device is connected to multiplier U4 output and is intended to subtract one from its product.

- Constant (*Wiring* library). Because subtractor U5 is designed to subtract one from the product of U4, the "01" on its bottom input is a constant. It should be placed near the bottom input and its properties set for Facing: East, Data Bits: 8, Value: 0x1 (this means hexadecimal 1).
- U6-Multiplier (*Arithmetic* library). This device multiplies the top and bottom inputs on its east side and sends the result to the output on its west side.
- Not Gate (*Gates* library). This is placed near the bottom input of multiplier U6 in order to negate that input.
- U7-Subtractor (*Arithmetic* library). This device is connected to multiplier U6 output and is intended to subtract one from its product.
- Constant (*Wiring* library). Because subtractor U7 is designed to subtract one from the product of U6, the "01" on its bottom input is a constant. It should be placed near the bottom input and its properties set for Facing: East, Data Bits: 8, Value: 0x1 (this means hexadecimal 1).
- U8-Adder (*Arithmetic* library). This device is designed to output the value of $A + A$ so the two inputs on its east side are tied together. The result is sent to the output on its west side.
- U9-Adder (*Arithmetic* library). This device adds the top and bottom inputs on its east side and sends the result to the output on its west side.
- Constant (*Wiring* library). Adder *U9* is intended to supply the *A+1* output and the "01" on its bottom input is a constant. It should be placed near the bottom input for adder *U9* and its properties set for Facing: East, Data Bits: 8, Value: 0x1 (this means hexadecimal 1).
- U10-Comparator (*Arithmetic* library). A comparator compares the two inputs on its east side. If the top input is greater than the bottom input then output ">" will go high. If the top and bottom inputs are equal then output "=" will go high. If the top input is less than the bottom input then output "<" will go high.

The next step is to place all of the inputs and outputs. Figure 3.2 shows where those items should go and the label for each item. Note: exact placement is not important since they can be repositioned. The *Radix* property for each of the inputs and outputs should be set to "Hexadecimal." The *Data Bits* property should be set as follows.

- CI: 1
- A: 8
- B: 8
- ArOut: 8
- CO: 1
- Cmp: 1

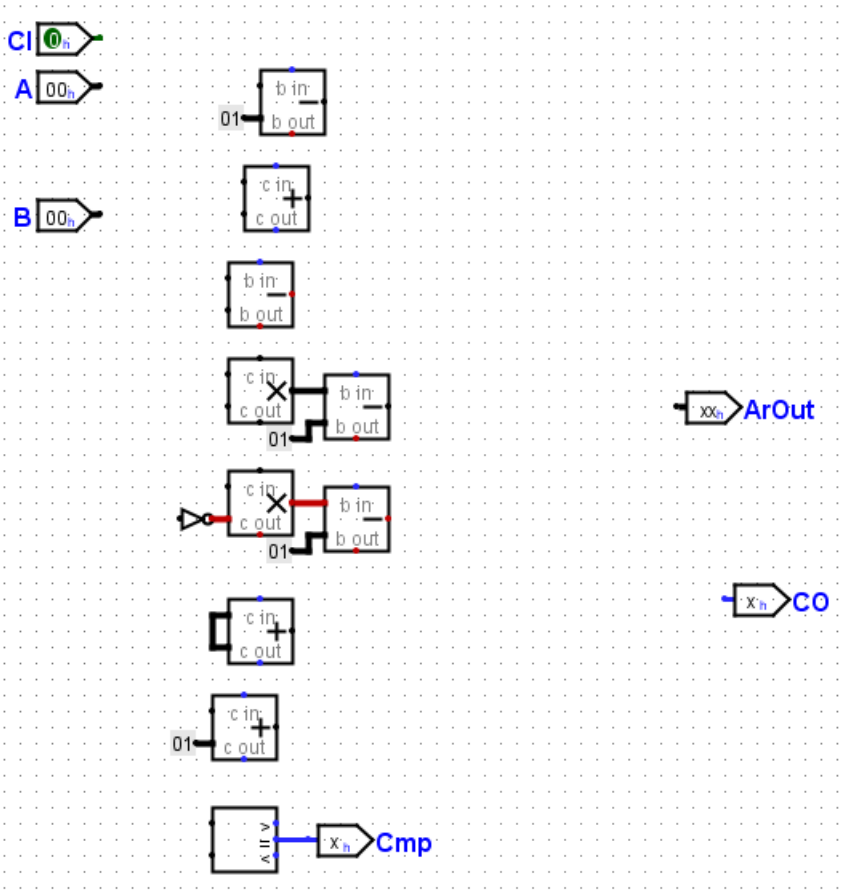

Figure 3.2: Arithmetic Inputs and Outputs

This circuit uses a Multiplexer (*Plexers* library) to select which device's output to connect to *ArOut*. A multiplexer is a digital logic workhorse that is found in many circuits, including Central Processing Units (CPUs). It is designed to switch a selected input to the output while ignoring all other input ports. In Figure 3.3, two multiplexers have been placed in the circuit. The top multiplexer will select one of eight inputs coming from the eight arithmetic devices to connect to *ArOut*. The bottom multiplexer will connect the carry out signal from the selected arithmetic device to the *CO* output. Also notice that the bottom multiplexer has a constant zero wired to input port zero.

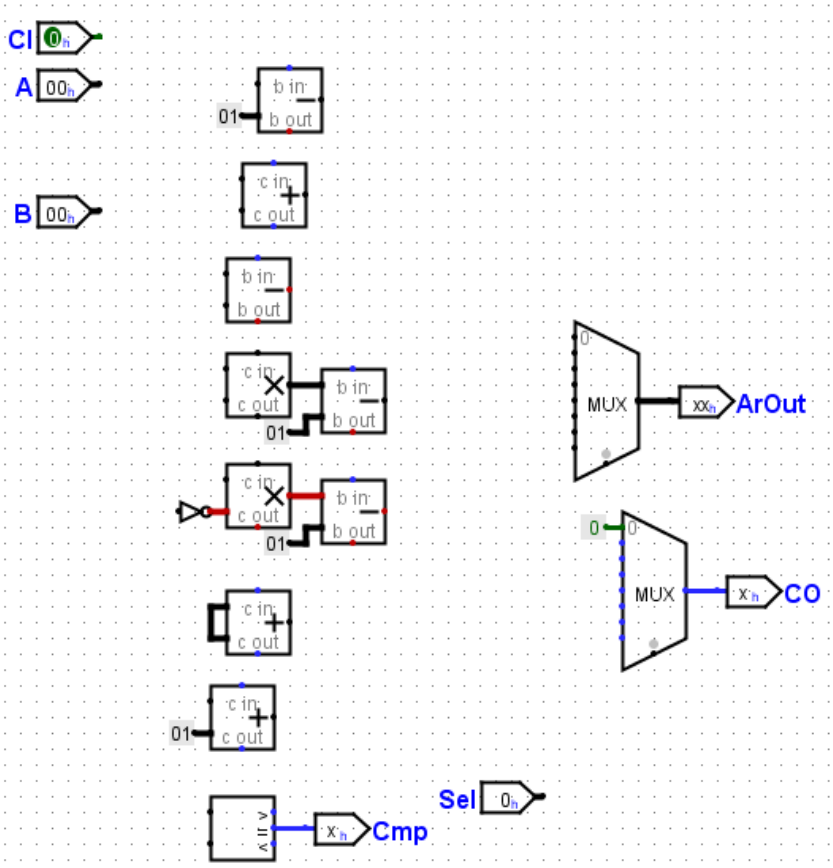

Figure 3.3: Placing the Multiplexers

The next step is to wire inputs *A* and *B* to each of the devices, as shown in Figure 3.4. Then the outputs of each device is wired to an appropriate port in the top multiplexer. This is not particularly challenging, but be careful to avoid crossed wires.

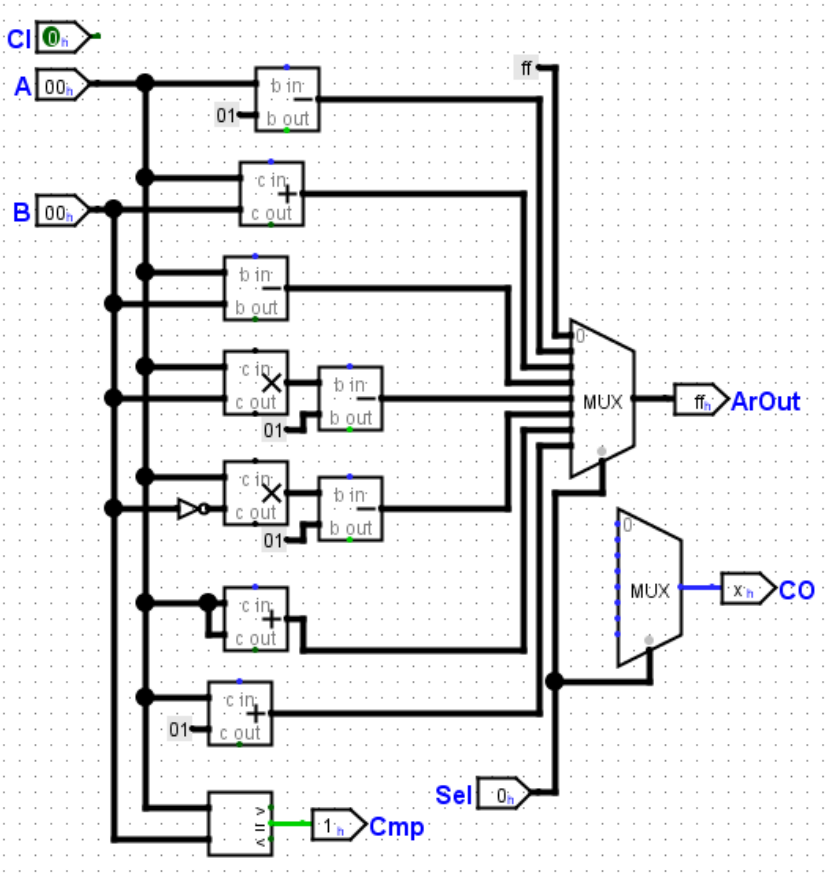

Figure 3.4: Wiring the Data Inputs and Outputs

Figure 3.4 shows the completed subcircuit with wires connecting *CI* to each device and then the devices wired to the appropriate port on the bottom multiplexer.

Notice that input zero on the top multiplexer is wired to a constant *ff*. That is the two's complement of negative one so that port will always transmit negative one, as in the specification sheet.

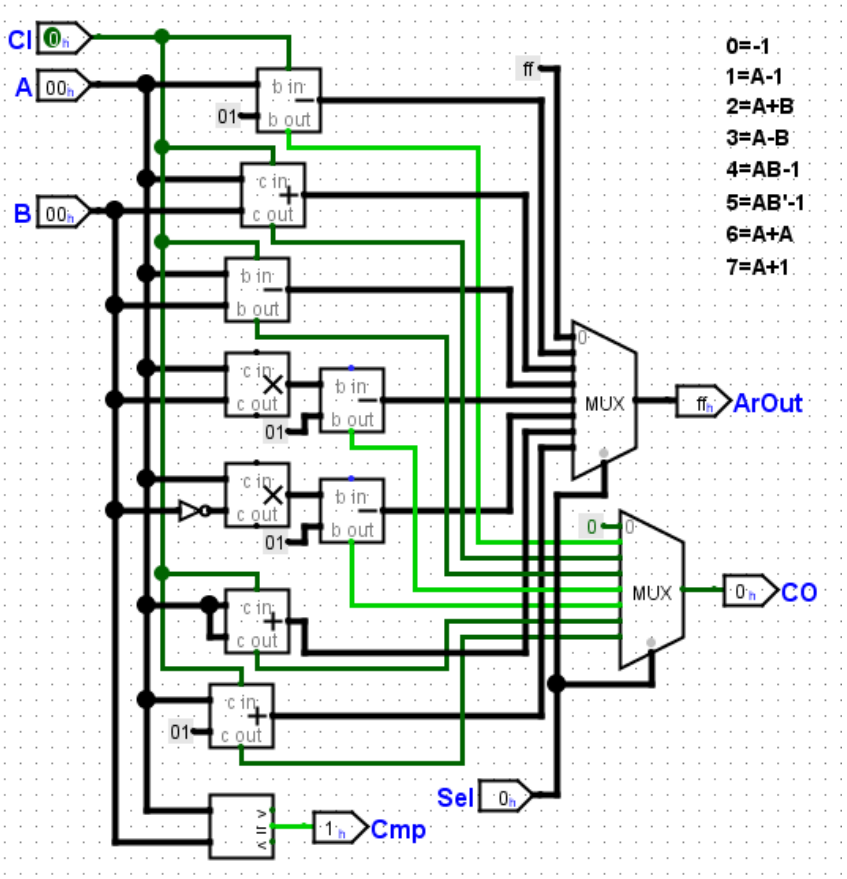

Figure 3.5: Arithmetic Final Circuit

Finally, the `main` circuit is completed by dropping the `arithmetic` subcircuit on the canvas and wiring an appropriate input or output port to each of the ports on the subcircuit. Figure 3.6 illustrates the main circuit.

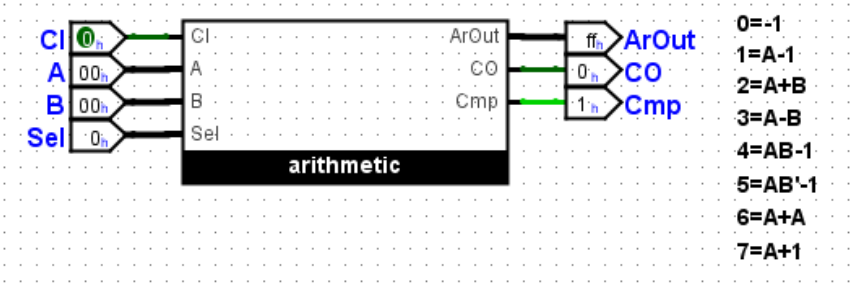

Figure 3.6: Arithmetic Main Circuit

This arithmetic device can be tested by entering numbers for the various inputs and checking to see if the output is correct. For example, if input *A* was set to 3 and input *B* were set to 4, then if *Sel* is set to 2, *ArOut* should be seven $(3+4)$. A test vector file has been provided for this lab so all of the arithmetic functions can be exercised.

As a last step, the `main` circuit must be renamed since this circuit will be reused in Lab 8. Click one time on the `main` circuit to activate it and in the properties panel change its label to `arith_main`.

3.3 DELIVERABLE

To receive a grade for this lab, complete the circuit. Be sure the standard identifying information is at the top left of the `arith_main` circuit, similar to:

```
George Self
Lab 03: Arithmetic Operations
September 17, 2019
```

Save the file with this name: *`Lab03_Arithmetic`* and submit that file for grading.

4 LOGIC OPERATIONS

4.1 PURPOSE

This lab develops a logic unit that includes eight different logic functions using *Logisim-Evolution* library components. This device will eventually be used as part of the Arithmetic Logic Unit (ALU) in Lab 8. This device will have two inputs, labeled *A* and *B*, and will output the following logic values.

1. AB
2. $(AB)'$
3. $A + B$
4. $(A + B)'$
5. AXorB
6. AB'
7. $A + B'$
8. A'

4.2 PROCEDURE

This circuit is very similar to the arithmetic circuit developed in Lab 3. However, the logic circuit is much simpler than the arithmetic circuit since there is not carry in/out bit and no comparator output.

To complete the lab, start a new circuit in *Logisim-Evolution* and create a subcircuit named `logic`. Place appropriate devices from the *Gates* library in the `logic` subcircuit such that it can produce the required output. Connect each of the devices to inputs *A* and *B* and then wire the outputs from each device to *LoOut* through a multiplexer. The exact design of the `logic` subcircuit is left to the student.

> Tip: the value *(AB)'* is a `NAND` gate and $(A + B)'$ is a `NOR` gate. To negate input *B* for AB' set the *Negate* property in the properties panel for the `AND` gate to "Yes." When that is done, the *B* input on the `AND` gate should include a small "negate" circle.

Drop the logic subcircuit on the main circuit and wire the various inputs and outputs, as shown in Figure 4.1.

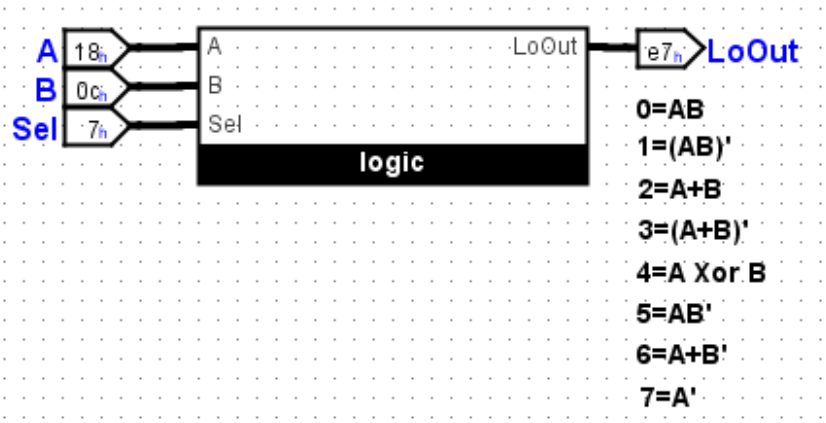

Figure 4.1: The Main Logic Circuit

The circuit can be tested by using the *poke* tool and entering various inputs and then checking to see that the output is correct. A test vector file has been provided for this lab so all of the logic functions can be exercised.

As a last step, the main circuit must be renamed since this circuit will be reused in Lab 8. Click one time on the main circuit to activate it and in the properties panel change its label to logic_main.

4.3 DELIVERABLE

To receive a grade for this lab, complete the circuit. Be sure the standard identifying information is at the top left of the logic_main circuit, similar to:

```
George Self
Lab 04: Logic Operations
September 17, 2019
```

Save the file with this name: *Lab04_Logic* and submit that file for grading.

5 BOOLEAN LOGIC

5.1 PURPOSE

The goal for this lab is to design circuits when given a Boolean expression. This is normally called "realizing" a circuit; that is, making a real circuit from a Boolean expression.

The *Logisim-Evolution* starter for this lab includes a `main` circuit and one subcircuit, named `Equation_1`. The starter subcircuit is used to practice creating a circuit from a Boolean expression and then a new subcircuit is added and a second Boolean expression is used to build that circuit.

5.2 PROCEDURE

5.2.1 *Subcircuit: Equation 1*

The starter circuit includes a subcircuit named `Equation_1`. Double-click that circuit in the Explorer Pane to activate it. The drawing canvas for this subcircuit is mostly blank except for a Boolean expression: $(A'BC') + (AB'C') + (ABC)$. Before starting to design a circuit, it is helpful to take a minute to analyze the expression.

- There are only three variables used in the entire expression: *A*, *B*, and *C*. Therefore, there would be three inputs into the circuit.
- There are three groups of variables and within each group the variables are joined with an `AND`. Therefore, the circuit must include three `AND` gates with three inputs for each gate.
- The three groups of variables are joined with an `OR`. Therefore, the circuit must include an `OR` gate with three inputs.
- While the expression does not name an output variable, it is reasonable to assume that the circuit would output a logic 1 or 0. Therefore, a one-bit output variable must be specified.

Start by placing three inputs and an output on the drawing canvas. Inputs are indicated by a green icon with *I->* on the tool bar above the drawing canvas. Click that tool and place three input pins named *In1A*, *In1B*, and *In1C* —that means "Input for Equation One, variable A" and so forth.

Outputs are indicated by a white icon with *->O* found on the tool bar above the drawing canvas. Click that tool and place an output named *Out1*. The circuit should look like Figure 5.1.

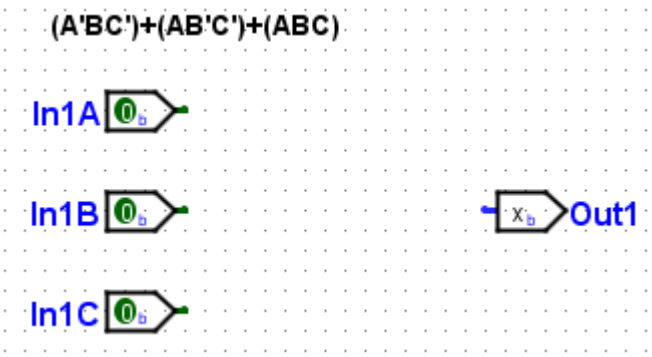

Figure 5.1: Equation 1 Inputs-Outputs

Next, the gates should be added. Place three `AND` gates on the circuit. Click each gate and in its properties panel set the *Number of Inputs* to 3.

Place an `OR` gate on the circuit. Click that gate and in its properties panel set the *Number of Inputs* to 3.

The circuit should look like Figure 5.2.

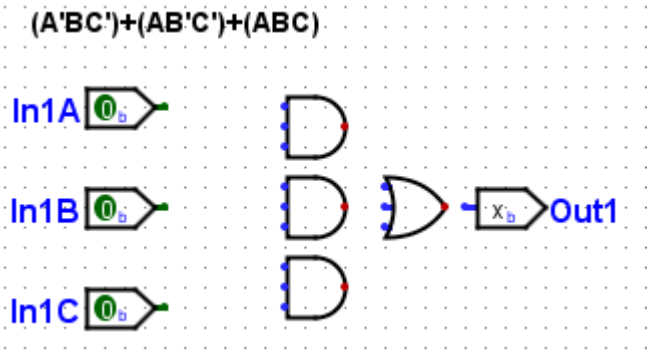

Figure 5.2: Equation 1 And-Or Gates

Next, the inputs for the `AND` gates should be set to match the Boolean expression. The top `AND` gate will match the first group of inputs, $(A'BC')$, so inputs A and C should be negated. To negate those two inputs, click the `AND` gate and in the properties panel set the *Negate* item for the top and bottom input to "Yes." When that is done, the two inputs on the `AND` gate should include a small "negate" circle.

In the same way, the middle and bottom input for the second `AND` gate should also be negated. The circuit should look like Figure 5.3.

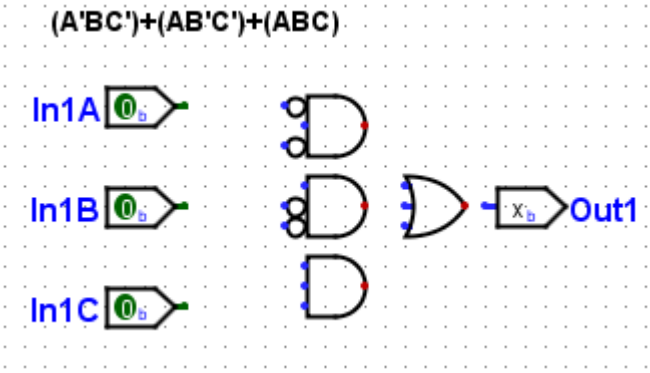

Figure 5.3: Equation 1 And Gate Inputs Set

Finally, connect all gates with wires, like Figure 5.4.

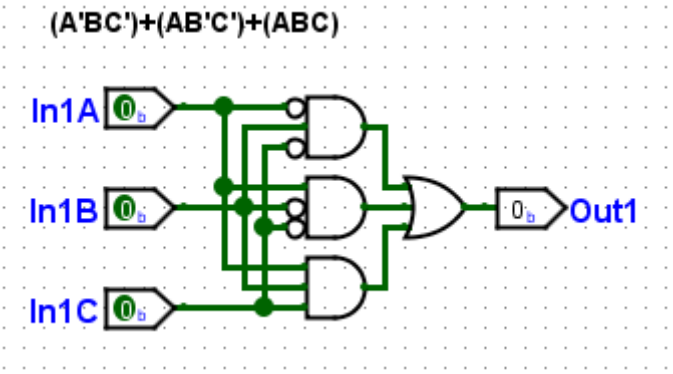

Figure 5.4: Equation 1 Circuit Completed

Test the circuit by selecting the *poke* tool in the tool bar (it looks like a pointing finger) and setting various combinations of 1 and 0 on the three inputs. The output pin should go high only when the inputs are set to $(A'BC')$, $(AB'C')$, or (ABC).

5.2.2 *Subcircuit: Equation 2*

A new subcircuit can be added to a circuit by clicking PROJECT -> ADD CIRCUIT. Name the new circuit `Equation_2`. Open the new subcircuit by double-clicking its name in the Explorer Pane.

Because this is a new subcircuit, the drawing canvas is blank. To start this subcircuit, write the equation for the circuit near the top of the drawing canvas by clicking the "A" button on the Toolbar and then clicking near the top of the drawing canvas and typing the following:

$$(A'B'CD') + (A'BCD) + (AB'CD') + (ABCD')$$

It will save time to take a few minutes and analyze the expression.

- There are only four variables used in the entire expression: *A*, *B*, *C*, and *D*. Therefore, there would be four inputs into the circuit.
- There are four groups of variables and within each group the variables are joined with an `AND`. Therefore, the circuit must include four `AND` gates with four inputs for each gate.
- The four groups of variables are joined with an `OR`. Therefore, the circuit must include an `OR` gate with four inputs.
- While the expression does not name an output variable, it is reasonable to assume that the circuit would output a logic 1 or 0. Therefore, a one-bit output variable must be specified.

Design the subcircuit using these names for the inputs: *In2A*, *In2B*, *In2C*, and *In2D*. Also include an output named *Out2*. Set the `AND` gates so the their inputs are negated properly and then wire the entire subcircuit. Finally, test the circuit to ensure the output goes high only when the four specified combinations of inputs are present.

5.2.3 *Main Circuit*

Make the `main` circuit active by double-clicking its name in the Explorer Panel. Click once on the `Equation_1` circuit and the cursor will change into an image of that circuit as it will appear on the drawing canvas. Click on the drawing canvas to drop that subcircuit. The circuit can later be moved by clicking it and dragging it to a new location. Wire the three inputs and output as shown in Figure 5.5. Notice that the input/output pins do not need to be named the same as in the subcircuit; for example, the output for `Equation_1` is labeled *Out1* but it is connected to an output pin labeled *True1*.

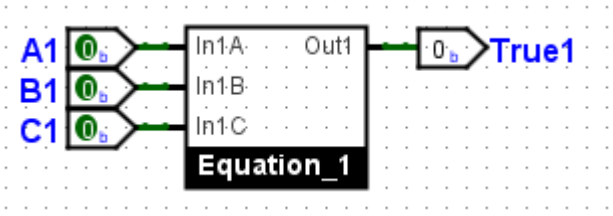

Figure 5.5: Main Circuit

Add the `Equation_2` circuit in the same way and wire four inputs and one output to that circuit. The inputs should be labeled *A2*, *B2*, *C2*, and *D2* and the output labeled *True2*.

5.3 DELIVERABLE

To receive a grade for this lab, complete the `main` circuit and both subcircuits. Be sure the standard identifying information is at the top left of the `main` circuit, similar to:

```
George Self
Lab 05: Boolean Equations
February 18, 2018
```

Save the file with this name: *`Lab05_Bool`* and submit that file for grading.

It is important to name all inputs and outputs as specified in the lab since they are checked with a Test Vector file that depends on those names.

6 PROGRAMMABLE LOGIC ARRAY

6.1 PURPOSE

This lab explores using a Programmable Logic Array (PLA) to simplify circuits that are designed for Boolean operations. A PLA is an Integrated Circuit (IC) that contains an array of `AND` and `OR` gates that can be linked in whatever way the circuit designer needs. A single PLA can replace dozens of other gates and greatly simplify circuit design.

6.2 PROCEDURE

6.2.1 *Equation One*

$$(A'BC') + (AB'C') + (ABC) \tag{6.1}$$

In Lab 5 the circuit in Figure 6.1 was developed to realize a Boolean expression.

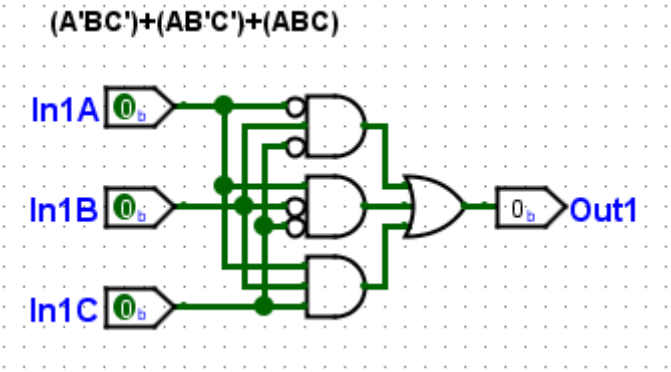

Figure 6.1: Boolean Expression Realized

This entire circuit can be realized in a single PLA saving the cost of redundant `NOT/AND/OR` gates and improving the reaction time for the circuit while reducing the heat it generates.

The *Logisim-Evolution* PLA component is a single box, as illustrated in 6.2. This PLA has a 3-bit input attached, for inputs *A*, *B*, and *C*, and a 1-bit output. The input is labeled *ABC1* to indicate this is the input for equation one.

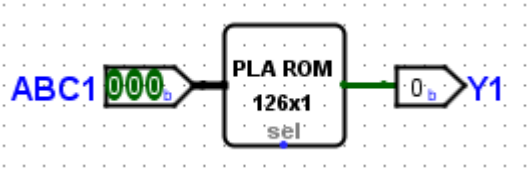

Figure 6.2: Equation One

Internally, a PLA contains a matrix of NOT/AND/OR gates that are connected by the circuit designer. Figure 6.3 illustrates the internal connections for the Equation One PLA.

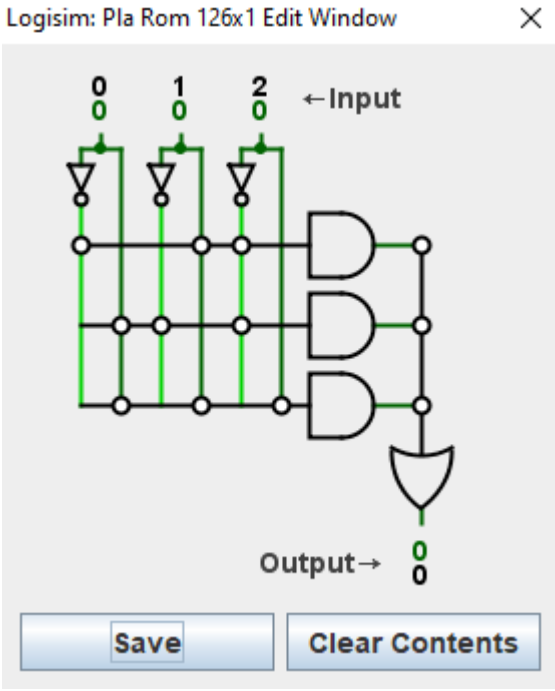

Figure 6.3: PLA Internal Connections

The three inputs are at the top of the PLA and are labeled input 0, 1, and 2. Also, the values of those inputs is indicated immediately under the input number. At this time, all three are inputting a zero.

Notice that all three inputs are split and a NOT gate is inserted in one of the two split lines. By doing this, both A and A' are available on PLA input zero.

To the right of the grid is a column of three AND gates. These gates look a bit odd since there is only one input for each gate. This, though, simply represents that the AND gate has a variable number of inputs, depending on how the designer wired the circuit. For example, the top row has three connections to the top AND gate so it is a 3-input gate. The gate sizing happens automatically within the PLA so the designer does not have to worry about it.

In the same way, there is a single OR gate near the bottom of the PLA. While the diagram indicates that the OR gate has a single input,

that gate actually has a variable input and will expand as necessary to support the circuit design. In this case, three separate rows are connected to the OR gate so it is a 3-input gate.

The very bottom of the PLA is the output. It is numbered as output zero and its current value is zero.

To wire the PLA device, the designer clicks on the wire intersections to place a connector (the small circle). Thus, row one of this device is connecting A', B, and C' to the top AND gate, which is then connected to the output OR gate.

Inspecting the connections for each row in this device should reveal that the top row is $(A'BC')$, the second row is $(AB'C')$ and the third row is (ABC), which are the three gates in the Boolean expression. The outputs from all three of those gates goes through an OR gate to the output.

Thus, this one PLA replaces all of the circuitry found in Figure 6.1.

Before moving on to the second equation, it will be helpful to take a look at the properties for a PLA.

VHDL	Verilog
Number Of Inputs	3
PlaANDAttr	3
PlaOutputsAttr	1
Label	
Label Font	SansSerif Bold 16
Label Visible	Yes
Contents	(click to edit)
ramSelAttr	Low Level

Figure 6.4: PLA Properties

The first property sets the number of inputs for the PLA, the second property, *PalANDAttr*, sets the number of AND gates and the third property, *PlaOutputsAttr*, sets the number of outputs. These three properties ensure that the PLA device can be used for many different applications. The three label properties are the same as found in most *Logisim-Evolution* components. Clicking the *Contents* property will open the PLA editor as illustrated in Figure 6.3[1]. Finally, the *ramSelAttr* determines whether the *Select* port, on the south edge of the PLA is enabled on a high or low signal. The enable port is not used in this lab.

1 Important! The *Logisim-Evolution* PLA seems to have a minor bug and the Contents editor will not always open. However, if the *ramSelAttr* is clicked so its drop-down list is visible then the Contents *(click to edit)* link will function properly. This is odd, but it will hopefully be corrected in a future iteration of *Logisim-Evolution* .

6.2.2 *Equation Two*

$$(BCD') + (A'B'C) + (A'BCD') + (ABCD) \tag{6.2}$$

The subcircuit for Equation Two is very similar to that for Equation One, but the PLA has different internal connections.

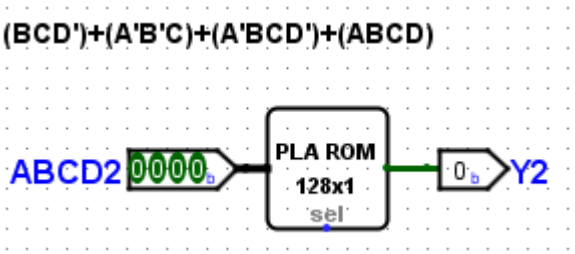

Figure 6.5: Equation Two

Notice that the input has four bits since the equation includes inputs *A*, *B*, *C*, and *D*. Also, the inputs and outputs include the number 2 to differentiate this subcircuit from the others.

When building this subcircuit be sure to use a PLA device from the library instead of copy/paste from the subcircuit for Equation One. The internal connections for Equation Two are illustrated in Figure 6.6.

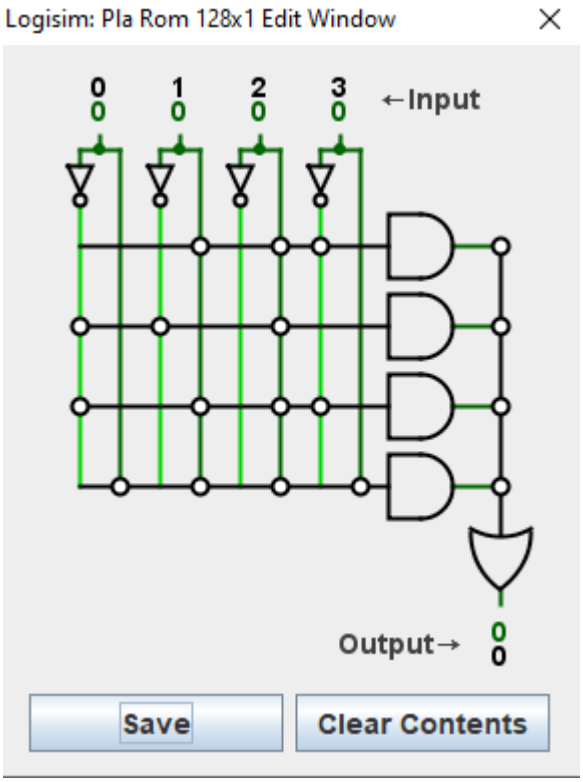

Figure 6.6: Connections for Equation Two

Notice that this equation includes two terms that have only three variables instead of four: (BCD′) and (A′B′C). It does not cause a problem when a term in incomplete, only the variables present in the term are connected and the missing variable is ignored.

6.2.3 *Equation Three*

$$(A'BC') + (AB'C') + (ABC) \qquad (6.3)$$
$$(A'B'CD') + (A'BCD) + (AB'CD') + (ABCD')$$

The subcircuit for Equation Three is very similar to that for Equation Two, but the PLA has different internal connections.

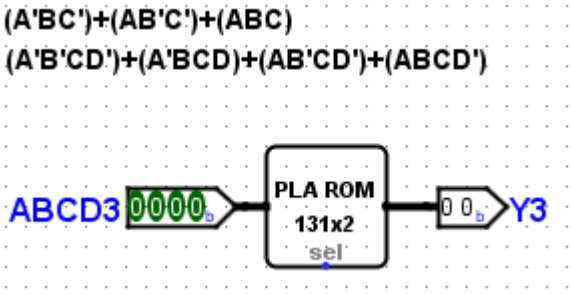

Figure 6.7: Equation Three

The major difference in Equation Three is that there are two outputs, so the PLA must have two outputs specified and each equation is connected to the correct output.

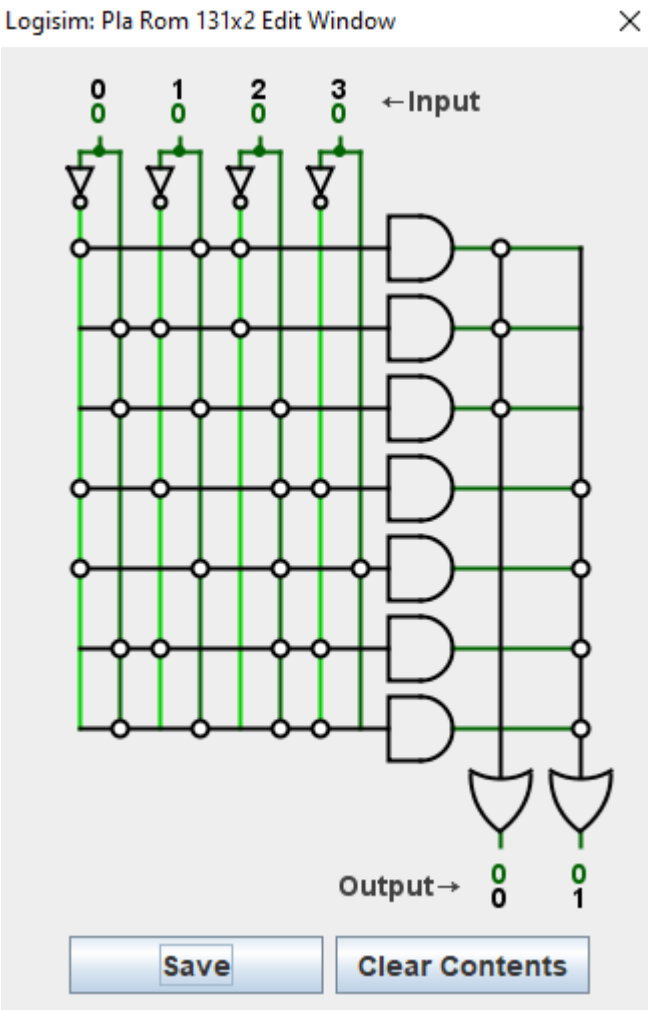

Figure 6.8: Connections for Equation Three

6.3 CHALLENGE

On the `main` circuit, wire a PLA device using the same procedure that was used for circuits one, two, and three. That PLA should display the correct outputs for equations 6.4.

$$(A'BC) + (ABC'D) + (A'B'CD) \tag{6.4}$$
$$(BCD') + (A'B'C) + (A'BCD') + (ABCD)$$

The main circuit can be tested with the test vector provided with the lab starter circuit.

6.4 DELIVERABLE

To receive a grade for this lab, complete the circuit. Be sure the standard identifying information is at the top left of the `main` circuit, similar to:

```
George Self
Lab 06: PLA
September 17, 2019
```

Save the file with this name: *Lab06_PLA* and submit that file for grading.

Part III

PRACTICE

Practice exercises are designed to familiarize students with many aspects of both combinational and sequential digital logic circuits. This section develops devices as varied as counters, encoders, and read-only memory. It also includes a rather complex

7

COUNTERS

Counters are perhaps the most commonly-used circuits in electronic devices. They are found in virtually all electronics systems, from the simplest embedded computers to massive mainframes. Counters are designed to cycle through a specific predefined sequence of binary numbers when an input pulse is applied. Typically, counters simply count up or down from given start and end numbers, but they can be designed to produce unique output patterns for special uses.

Counters, though, are used for more than simple counting. They can measure time so devices like alarm clocks and watches include counters. They are used as frequency dividers so a fast input frequency can be output at a slower rate. In devices with memory they are used to increment memory addresses as a program steps through some process. They can activate a series of subcircuits in sequence as part of a complex process. They are, in short, one of the most important workhorses of the digital logic world.

7.1 PURPOSE

This lab has two goals:

1. Develop several different common counters using *D* flip-flops. Because there are two main families of counters, asynchronous and synchronous, this lab includes examples of both.
2. Introduce the *Logisim-Evolution* chronogram feature that generates a timing diagram as a sequential circuit functions.

7.2 PROCEDURE

7.2.1 *Asynchronous Up Counter*

A counter is built from a series of flip-flops and where the output from each flip-flop is combined to create the counter output, trigger the next flip-flop, or both. Each flip-flop is considered a "stage" of the counter. A counter is triggered by a clock signal that is typically supplied by a timer with a regularly-recurring pattern of high/low levels, but it can also be triggered by an event of some sort, like the press of a button or the completion of a process.

One of the simplest counters is illustrated in Figure 7.1. This is an asynchronous four-stage up counter. A counter is is considered "asynchronous" if the input clock signal is applied to only the first

In all Counter circuits in this manual flip-flop U0 provides the Least Significant Bit to the output and U3 provides the Most Significant Bit.

stage and then that signal ripples through each flip-flop in turn. Thus, an asynchronous counter is frequently called a "ripple" counter.

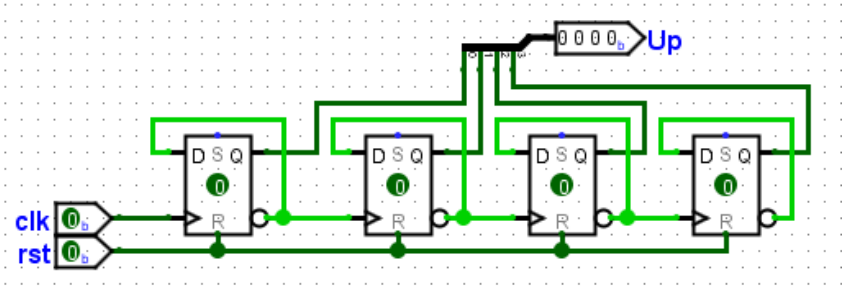

Figure 7.1: Asynchronous Up Counter

The following list describes the operation of the counter in Figure 7.1. Students should open the counter circuit with *Logisim-Evolution* then use the "poke" tool to set the clock high then low (one complete clock cycle) as they follow the description below.

RESET IS ACTIVATED All flip-flops are reset so Q is low and Q' is high.

TICK 1 $U0$ clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

TICK 2 $U0$ clocked: $Q0 \downarrow$ — $Q'0 \uparrow$
$U1$ clocked: $Q1 \uparrow$ — $Q'1 \downarrow$

TICK 3 $U0$ clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

TICK 4 $U0$ clocked: $Q0 \downarrow$ — $Q'0 \uparrow$
$U1$ clocked: $Q1 \downarrow$ — $Q'1 \uparrow$
$U2$ clocked: $Q2 \uparrow$ — $Q'2 \downarrow$

TICK 5 $U0$ clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

TICK 6 $U0$ clocked: $Q0 \downarrow$ — $Q'0 \uparrow$
$U1$ clocked: $Q1 \uparrow$ — $Q'1 \downarrow$

TICK 7 $U0$ clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

TICK 8 $U0$ clocked: $Q0 \downarrow$ — $Q'0 \uparrow$
$U1$ clocked: $Q1 \downarrow$ — $Q'1 \uparrow$
$U2$ clocked: $Q2 \downarrow$ — $Q'2 \uparrow$
$U3$ clocked: $Q3 \uparrow$ — $Q'3 \downarrow$

As the clock continues the counter would cycle through the binary values 1001 - 1111. The following table lists the *Up* counter output as indicated by the Q values at each tick listed above.

Tick	Output
Reset	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000

Table 7.1: Up Counter Output

7.2.2 *Asynchronous Down Counter*

The asynchronous down counter illustrated in Figure 7.2 is very similar to the up counter in Figure 7.1 except the stages are triggered from the Q output of the preceding stage rather than Q' and the *Reset* signal is applied to the flip-flop S input rather than R.

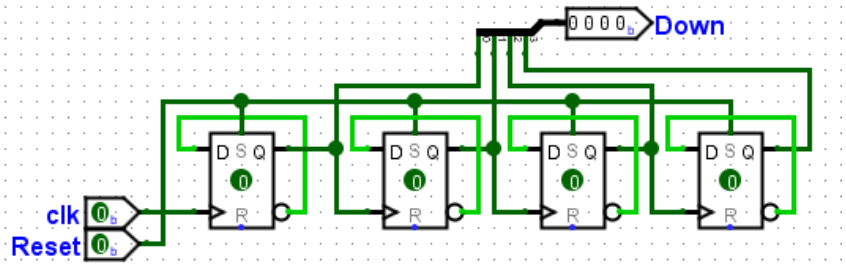

Figure 7.2: Asynchronous Down Counter

The following list describes the operation of the counter in Figure 7.2. Students should open the counter circuit with *Logisim-Evolution* then use the "poke" tool to set the clock high then low (one complete clock cycle) as they follow the description below.

RESET IS ACTIVATED All flip-flops are set so Q is high and Q' is low.

TICK 1 *Uo* clocked: $Qo \downarrow$ — $Q'o \uparrow$

TICK 2 *Uo* clocked: $Qo \uparrow$ — $Q'o \downarrow$

U1 clocked: $Q1 \downarrow$ — $Q'1 \uparrow$

TICK 3 *Uo* clocked: $Qo \downarrow$ — $Q'o \uparrow$

TICK 4 *Uo* clocked: $Qo \uparrow$ — $Q'o \downarrow$

U1 clocked: $Q1 \uparrow$ — $Q'1 \downarrow$

U2 clocked: $Q2 \downarrow$ — $Q'2 \uparrow$

TICK 5 *U0* clocked: $Q0 \downarrow$ — $Q'0 \uparrow$

TICK 6 *U0* clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

U1 clocked: $Q1 \downarrow$ — $Q'1 \uparrow$

TICK 7 *U0* clocked: $Q0 \downarrow$ — $Q'0 \uparrow$

TICK 8 *U0* clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

U1 clocked: $Q1 \uparrow$ — $Q'1 \downarrow$

U2 clocked: $Q2 \uparrow$ — $Q'2 \downarrow$

U3 clocked: $Q3 \downarrow$ — $Q'3 \uparrow$

As the clock continues the counter would cycle through the binary values 0110 - 0000. The following table lists the *Down* counter output as indicated by the *Q* values at each tick listed above.

Tick	Output
Reset	1111
1	1110
2	1101
3	1100
4	1011
5	1010
6	1001
7	1000
8	0111

Table 7.2: Down Counter Output

7.2.3 *Asynchronous Decade Counter*

Binary counters, like those considered in Figure 7.1 and Figure 7.2 are only able to count to a value that is a power of two but it is often necessary to build a counter that stops at some other value. These types of counters are called "mod" counters (short for "modulus") since they count up to a preset value then reset and start over, like modulus math. One of the most common mod counters is one that has ten states (it counts from zero to nine) and then resets, and that type of counter is generally referred to as a decade counter. Decade counters are found in any application that has to count in decimal for easy human interpretation.

The logic of a mod counter is to add an AND gate on the flip-flop outputs such that the output of the AND gate is high when the flip-flop outputs equal the mod number. For example, the AND gate for a decade counter would go high when the count reaches ten and that signal would immediately reset the counter back to zero.

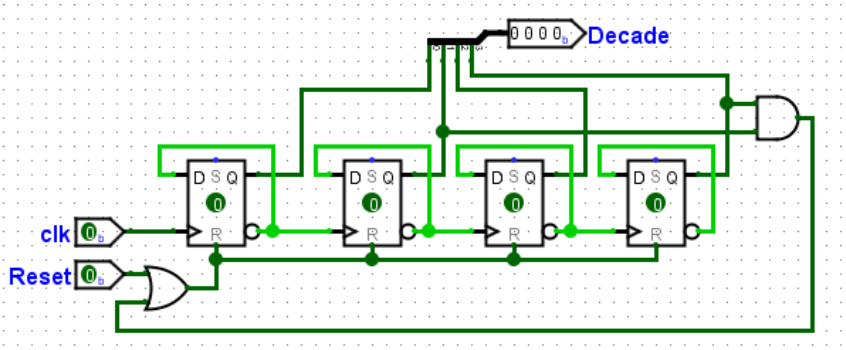

Figure 7.3: Asynchronous Decade Counter

The following list describes the operation of the counter in Figure 7.3:

RESET IS ACTIVATED All flip-flops are reset so Q is low and Q' is high.

TICK 1 $U0$ clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

TICK 2 $U0$ clocked: $Q0 \downarrow$ — $Q'0 \uparrow$
$U1$ clocked: $Q1 \uparrow$ — $Q'1 \downarrow$

TICK 3 $U0$ clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

TICK 4 $U0$ clocked: $Q0 \downarrow$ — $Q'0 \uparrow$
$U1$ clocked: $Q1 \downarrow$ — $Q'1 \uparrow$
$U2$ clocked: $Q2 \uparrow$ — $Q'2 \downarrow$

TICK 5 $U0$ clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

TICK 6 $U0$ clocked: $Q0 \downarrow$ — $Q'0 \uparrow$
$U1$ clocked: $Q1 \uparrow$ — $Q'1 \downarrow$

TICK 7 $U0$ clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

TICK 8 $U0$ clocked: $Q0 \downarrow$ — $Q'0 \uparrow$
$U1$ clocked: $Q1 \downarrow$ — $Q'1 \uparrow$
$U2$ clocked: $Q2 \downarrow$ — $Q'2 \uparrow$
$U3$ clocked: $Q3 \uparrow$ — $Q'3 \downarrow$

TICK 9 $U0$ clocked: $Q0 \uparrow$ — $Q'0 \downarrow$

TICK 10 *U1* clocked: $Qo \uparrow$ — $Q'o \downarrow$

> Both inputs for the `AND` gate are momentarily high and that sends a reset signal that causes all outputs to go low.

As the clock continues the counter would cycle through the binary values 0000 - 1001. The following table lists the *Decade* counter output as indicated by the *Q* values at each tick listed above.

Tick	Output
Reset	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	0000

Table 7.3: Decade Counter Output

7.2.4 *Synchronous Ring Counter*

In a ring counter the high bit is shifted through all of the bits one at a time. This counter is very useful in controlling subcircuits since the high bit in the counter can activate the next subcircuit in the sequence.

The ring counter presented here is also a synchronous circuit; that is, each clock pulse is applied to all of the flip-flops instead of just the first stage. The *Q* output from each flip-flop is used but *Q'* is not needed at all. Also, there is a feedback line from *U3* to the data input port of *Uo* so when the *Q* output of *U3* goes high that is made available to *Uo* and loop that value back through the circuit.

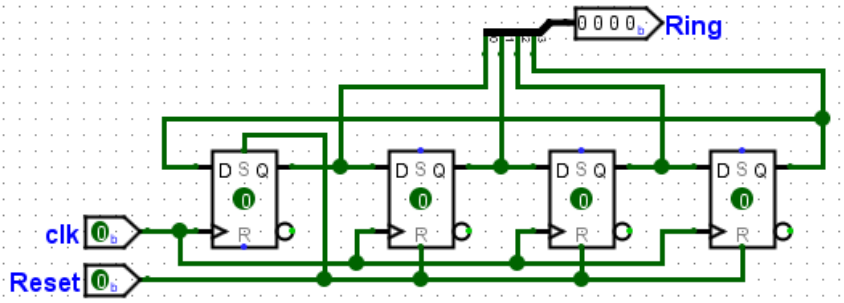

Figure 7.4: Synchronous Ring Counter

The following list describes the operation of the counter in Figure 7.4. Students should open the counter circuit with *Logisim-Evolution* then use the "poke" tool to set the clock high then low (one complete clock cycle) as they follow the description below.

RESET IS ACTIVATED *U0* is set and *U1*-*U3* are reset so the counter is seeded with a single high bit to shift.

TICK 1 $Q0 \downarrow$ — $Q1 \uparrow$

TICK 2 $Q1 \downarrow$ — $Q2 \uparrow$

TICK 3 $Q2 \downarrow$ — $Q3 \uparrow$

TICK 4 $Q3 \downarrow$ — $Q1 \uparrow$

As the clock continues the counter would cycle through the binary values 0001 - 1000. The following table lists the *ring* counter output as indicated by the *Q* values at each tick listed above.

Tick	Output
Reset	0001
1	0010
2	0100
3	1000
4	0001
5	0010
6	0100
7	1000
8	0001

Table 7.4: Ring Counter Output

7.2.5 *Synchronous Johnson Counter*

A Johnson Counter is similar to a ring counter in that a high bit value is shifted through the entire binary word. The difference is that the feedback loop comes from the Q' output of the last stage rather than the Q output. This type of counter is sometimes called a "twisted tail" counter since the Q' output is fedback.

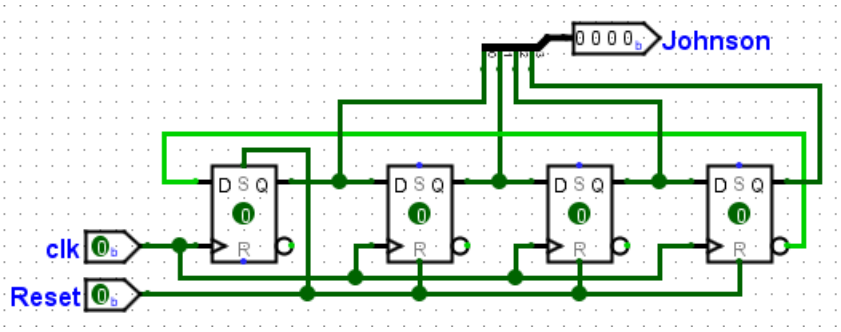

Figure 7.5: Synchronous Johnson Counter

The following list describes the operation of the counter in Figure 7.5. Students should open the counter circuit with *Logisim-Evolution* then use the "poke" tool to set the clock high then low (one complete clock cycle) as they follow the description below.

RESET IS ACTIVATED *U0* is set and *U1-U3* are reset so the counter is seeded with a single high bit to shift.

TICK 1 $Q1 \uparrow$

TICK 2 $Q2 \uparrow$

TICK 3 $Q3 \uparrow$

TICK 4 $Q0 \downarrow$

TICK 5 $Q1 \downarrow$

TICK 6 $Q2 \downarrow$

TICK 7 $Q3 \downarrow$

TICK 8 $Q0 \uparrow$

As the clock continues the counter would cycle through the binary values 0000 - 1111. The following table lists the *Johnson* counter output as indicated by the Q values at each tick listed above.

Tick	Output
Reset	0001
1	0011
2	0111
3	1111
4	1110
5	1100
6	1000
7	0000
8	0001

Table 7.5: Johnson Counter Output

7.2.6 *Main*

The `main` circuit provides a human interface to try out each of the counters by dropping them in place of the *Up* counter.

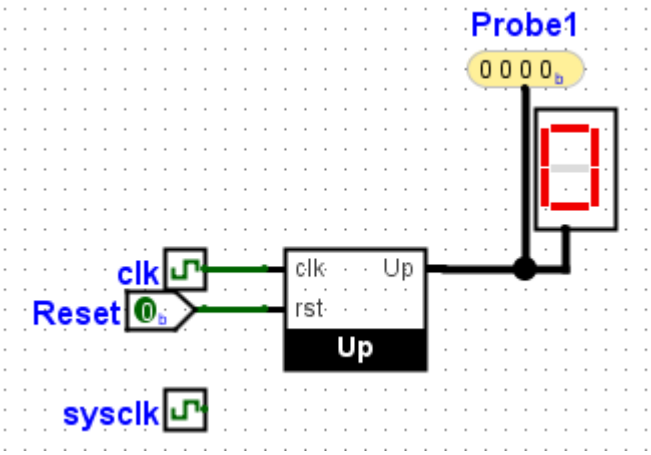

Figure 7.6: Main Circuit

Notice that there are two clocks in the `main` circuit. *Clk* is linked to the counter being tested and is used within the counter circuit to advance the count. *Sysclk* is used by the *Logisim-Evolution* chronogram as described in the next section of this document.

7.2.7 *Chronogram*

Logisim-Evolution can generate a timing diagram, called a *chronogram*, for a sequential circuit. That is a representation of the various signals in a circuit and how those signals change over time. Figure 7.7 is the timing diagram for an Up counter.

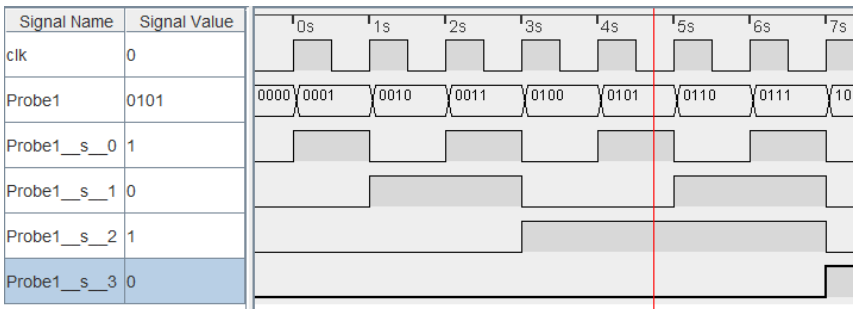

Figure 7.7: Timing Diagram for Up Counter

At the top of Figure 7.7 is a scale that indicates the number of seconds that the counter has been operating. The first trace is the input clk signal. The clock goes high at the start of each second and then goes low at the half-second mark. Under the clock is the "Probe1" signal. Because that is a four-bit number *Logisim-Evolution* displays the number, but under that number is a breakout of the four bits that make up that number. Thus, at time zero "Probe1" is 0001 and "Probe1_s_0" (that stands for "Probe 1, Signal 0") is high while the other bits are low. The *Logisim-Evolution chronogram* includes a cursor indicated by a red line (found just before the five second tick in Figure 7.7) that can be placed anywhere along the diagram. The cursor sets the values of each signal in the area on the left edge of the diagram, so the cursor in Figure 7.7 is pointing to a spot where the *clk* is low, *Probe1* is at 0101, and so forth.

Follow the next steps to use the chronogram. Notes: the chronogram will only check subcircuits that are found on the `main` subcircuit. Therefore, in order to create a timing diagram all subcircuits need to be combined on `main`. The labs completed in this manual have been designed to use the `main` subcircuit as the human interface so the chronogram feature will work well with these circuits.

1. In the `main` subcircuit, add a "sampling clock" labeled *sysclk* (this name is important, do not change it to something else). The sampling clock is only used by the *chronogram* and will not show up in the timing diagram. It should not be connected to any other components and can be placed anywhere on `main`. Set the properties for *sysclk* to a 1 Tick high duration and a 1 Tick low duration (this is the default).

2. Add a circuit master clock labeled *clk*. This is the clock that will be used to trigger all components in the circuit. Set the properties for *clk* to a 4 Tick high duration and a 4 Tick low duration.

3. Set Simulate -> Tick Frequency to 4 Hertz. This will simulate a clock that ticks once per second, as in Figure 7.7. While the

actual tick frequency can be changed later to "speed up" the circuit, a one-second tick is useful for learning how the *chronogram* works.

4. Click SIMULATE -> CHRONOGRAM to set up the *chronogram*. Figure 7.8 illustrates the initial setup screen for the *chronogram*.

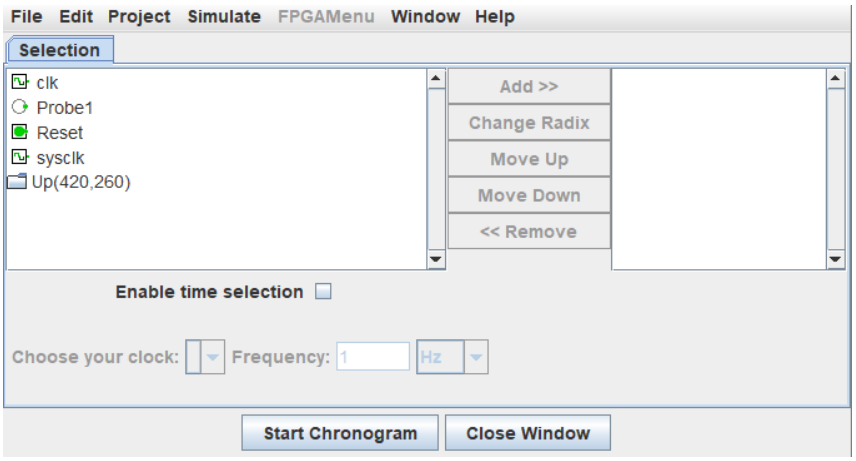

Figure 7.8: Set Up Chronogram

5. Click *sysclk* in the left panel and then click *Add* » to add that signal to the *chronogram*. The "-2" following the *sysclk* name in the right panel indicates that it is a binary signal. It is probably best to add the *sysclk* signal first so it is not overlooked.

NOTE: sysclk must be added to the chronogram or it will not sample the circuit; however, the sysclk signal will not actually show up in the timing diagram.

6. Click *clk* in the left panel and then click *Add* » to add that signal to the *chronogram*.

7. Click *Probe1* in the left panel and then click *Add* » to add that signal to the *chronogram*.

8. Click "Enable time selection" and chose *clk* as the clock with a frequency of 1 Hertz.

9. The *chronogram* setup should look like Figure 7.9.

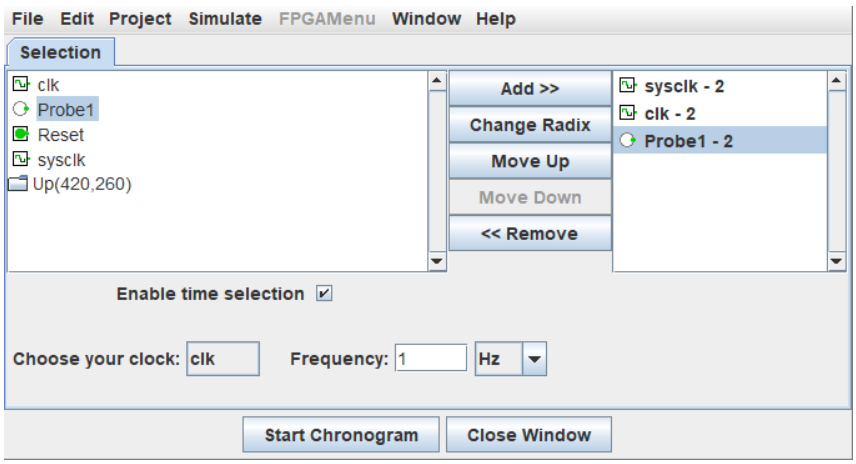

Figure 7.9: Chronogram Ready

10. Click *Start Chronogram* and the screen illustrated in Figure 7.10 pops up.

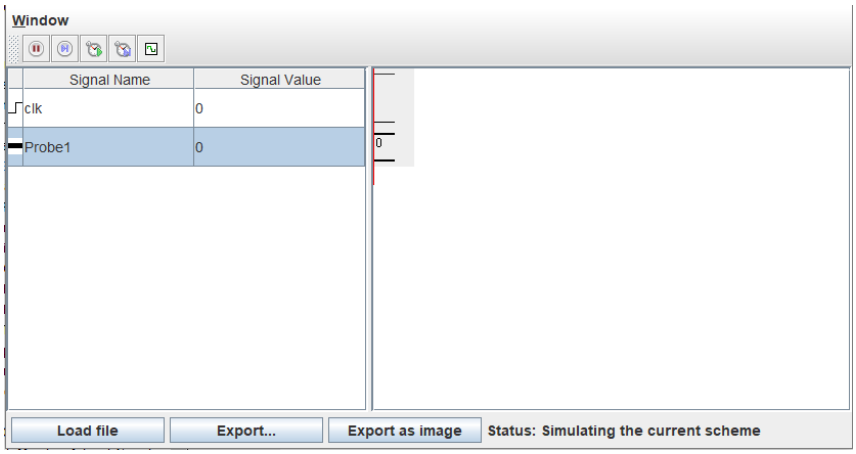

Figure 7.10: Chronogram Starting

11. Right-click on the *Probe1* signal and set the format for binary. The format can be set for any radix but to match this lab binary numbers should be specified.

12. Right-click on the *Probe1* signal and enable *Expand* to see all four signals that create *Probe1*.

13. At this point, the chronogram should look like Figure 7.11.

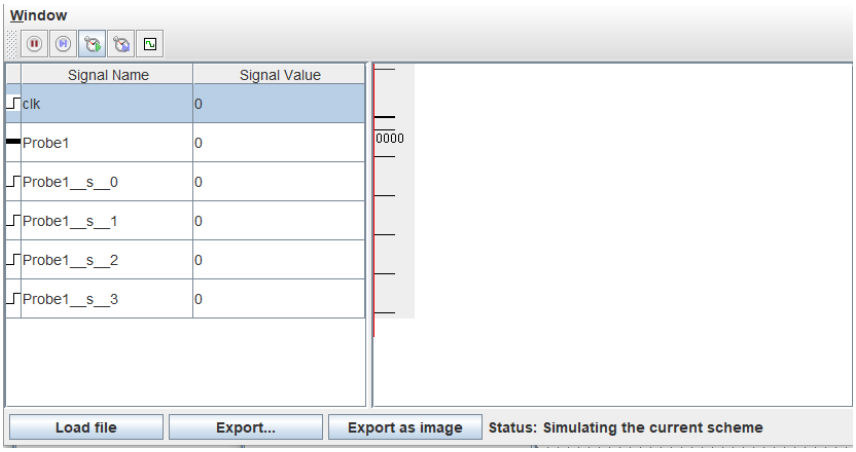

Figure 7.11: Chronogram At Zero Time

14. The *chronogram* has five buttons that control the simulator.

Figure 7.12: Chronogram Controls

- Button One: Start/Stop the simulation.
- Button Two: Simulate one step.
- Button Three: Start/Stop *sysclk*. This will "turn on" the chronogram and begin creating a timing diagram.
- Button Four: Step one *sysclk* tick. This will tick the *sysclk* one time. Since this lab set up the *sysclk* for four ticks per second this button would need to be clicked four times to extend the timing diagram one second.
- Button Five: Step one *clk* tick. This extends the timing diagram by one complete clock tick, or one second in this circuit.

15. Click button three to start the *chronogram* and watch the timing diagram unfold. After a few seconds click that button a second time to stop the *chronogram*.

16. The following can be done once the timing diagram is complete.

- Click on the timing diagram to set the cursor (indicated by a red line). Once the cursor is set the values for each signal at the cursor's location are printed next to the signal's label on the left edge of the timing diagram.
- Hover the mouse over the timing diagram and roll the mouse wheel to zoom the timing diagram appearance.

- Click "Export" to save the timing diagram signal levels in a text file. That file can later be loaded to reevaluate the timing diagram.
- Click "Export as image" to save the timing diagram as a PNG file.

7.3 CHALLENGE

This lab includes several different timers. Place all of them on a single subcircuit named `Universal` that includes an output mux so a user can select the type of counter output desired. Place the `Universal` circuit on `main` and wire appropriate inputs and outputs.

Set up the chronogram for the ring counter and create a ten-second timing diagram for that counter. Save the timing diagram as a PNG image named "RingCounter."

7.4 DELIVERABLE

To receive a grade for this lab, complete the Challenge. Be sure the standard identifying information is at the top left of the `main` circuit:

```
George Self
Lab 07: Counters
March 17, 2018
```

Save the circuit with this name: *Lab07_counter* and submit that along with *RingCounter.PNG* for grading.

8

ARITHMETIC LOGIC UNIT (ALU)

8.1 PURPOSE

In this lab you will build an Arithmetic Logic Unit (ALU). An ALU is an important digital logic device used to perform all sorts of arithmetic and logic functions in a circuit. The commercial 74181 ALU has two four-bit data inputs along with a one-bit mode (M) and a four-bit select input. Depending on those settings, the device will complete one of the functions listed in Table 8.1.

Select	Logic (M=1)	Arithmetic (M=0)
0000	A'	A
0001	(A + B)'	A + B
0010	A'B	A + B'
0011	Logical 0	minus 1 (2's Comp)
0100	(AB)'	A + AB'
0101	B'	(A + B) plus AB'
0110	A XOR B	A minus B minus 1
0111	AB'	AB' minus 1
1000	A' + B	A plus AB
1001	(A XOR B)'	A plus B
1010	B	(A + B') plus AB
1011	AB	AB minus 1
1100	Logical 1	A plus A
1101	A + B'	(A + B) plus A
1110	A + B	(A + B') plus A
1111	A	A minus 1

Table 8.1: Function Table for 74181 ALU

Notes: in the "Arithmetic" column, the + sign indicates logic *OR* while the words *plus* and *minus* indicate arithmetic add and subtract operations. The value of *A plus A* is the same as shifting the bits left to the next most significant position.

The ALU built in this lab is not as complex as an 74181 IC, however it demonstrates the basic functions of an ALU.

8.2 PROCEDURE

Load the ALU starter circuit in *Logisim-evolution*.

8.2.1 *main*

The `main` circuit does nothing more than provide a human-friendly interface for the rest of the ALU. That interface include two eight-bit inputs (labeled *A* and *B*), a three-bit select, a one-bit mode, a carry-in and carry-out bit (so the ALU could be chained to another), a *compare* output (TRUE if the two inputs are equal), and a eight-bit output (labeled *ALU_Out*). In operation, numbers are entered at *A* and *B*, the mode and select are set, and then the result is read on *ALU_Out*.

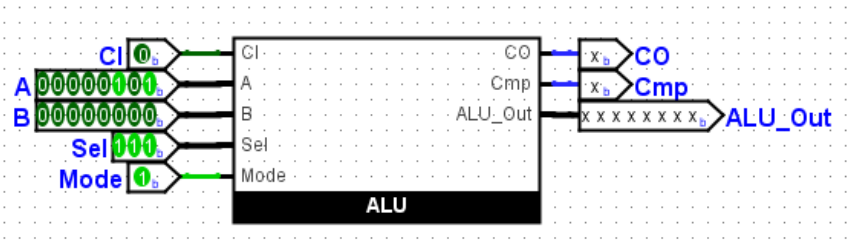

Figure 8.1: ALU main

8.2.2 *ALU*

The `ALU` subcircuit is designed to contain the logic that routes *A*, *B*, and *Sel* to two other subcircuits, `arith\_main` or `logic\_main`. It then uses a multiplexer to route the output of one of those subcircuits to an output port depending on the setting of the *Mode* bit. Note that the inputs are sent to both subcircuits but only the output specified by the *Mode* is returned to the user. This type of logic is also used in Labs 3 and 4.

The Arithmetic and Logic subcircuits that were built in Labs 3 and 4 will be reused for this lab. This is the way that circuit designers can reuse their work to build more complex circuits without having to reinvent the proverbial wheel for every project. To reload those old labs, click PROJECT -> LOAD LIBRARY -> LOGISIM-EVOLUTION LIBRARY[1]. Find Lab 3, Arithmetic, and click *Open*. That lab is now available as a library in the library list on the left side of the *Logisim-Evolution* workspace. Follow the same procedure to also load Lab 4, Logic, as a library.

Open the Arithmetic library and drop the `arithmetic` subcircuit in the `ALU` subcircuit. This process is exactly like dropping a device from the Wiring library and should not be difficult to figure out. In the

1 In order to load labs for reuse it is important to store the project files in the same folder.

same way, drop the `logic` subcircuit from the Logic library onto the `ALU` subcircuit.

At this point, the ALU subcircuit should resemble Figure 8.2.

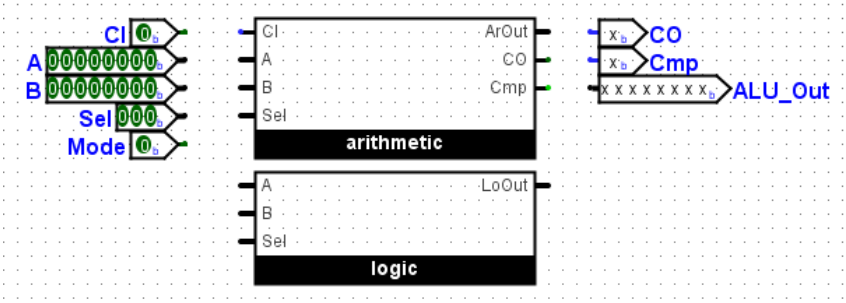

Figure 8.2: ALU Subcircuit

8.2.3 *Challenge*

Continue to wire the `ALU` subcircuit. Note: the inputs and outputs that were provided will need to be repositioned in order to complete this build.

- Wire the *CI* input to the *CI* port on the *arithmetic* device
- Wire inputs A and B to ports *A* and *B* on both devices.
- Wire the *Sel* input to the *Sel* ports on both devices.
- The *arithmetic* *CO* and *Cmp* ports should be wired to the *CO* and *Cmp* outputs
- Add an 8-bit, 2-input multiplexer. The inputs should be wired to the output ports on both devices. The multiplexer select bit should be wired to the *Mode* input. Finally, the multiplexer output should be wired to the *ALU_Out* output.

8.2.4 *Testing the Circuit*

The ALU can be tested from the `main` circuit. Several values can be entered on *A* and *B* and then various arithmetic and logic operations selected. The outputs for each check should be accurate. A test vector file has been provided for this lab so all of the ALU's functions can be exercised.

8.3 DELIVERABLE

To receive a grade for this lab, complete the Challenge. Be sure the standard identifying information is at the top left of the *main* circuit, similar to this:

```
George Self
Lab 08: ALU
February 18, 2018
```

Save the file with this name: *Lab08_ALU* and submit that file for grading.

9 PRIORITY ENCODER

9.1 PURPOSE

Often a circuit will receive data from several sources at one time and there must be a way to prioritize those inputs. This circuit creates a simple priority encoder for nine different inputs. This is a fairly simple circuit but is best explained by building and "playing around" with it rather than attempting to understand a printed text; thus, the explanation for this lab is somewhat limited.

9.2 PROCEDURE

Start *Logisim-Evolution* and create a subcircuit named `Encoder`. Open that subcircuit and place 12 `AND` gates as illustrated in Figure 9.1.

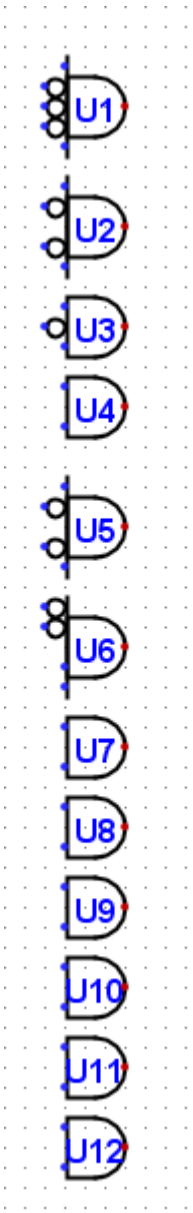

Figure 9.1: AND Gates

The gates have one data bit and these properties:

- **U1**: Five inputs, numbers two, three, and four negated.
- **U2**: Four inputs, numbers two and three negated.
- **U3**: Three inputs, number two negated.
- **U4**: Two inputs, none negated.
- **U5**: Four inputs, numbers two and three negated.
- **U6**: Four inputs, numbers one and two negated.
- **U7-U12**: Two inputs, none negated.

Many of the output signals need to be combined with `OR` gates and those should be added next, as in Figure 9.2. Note: U16 is a `NOR` (*Gates* library) gate.

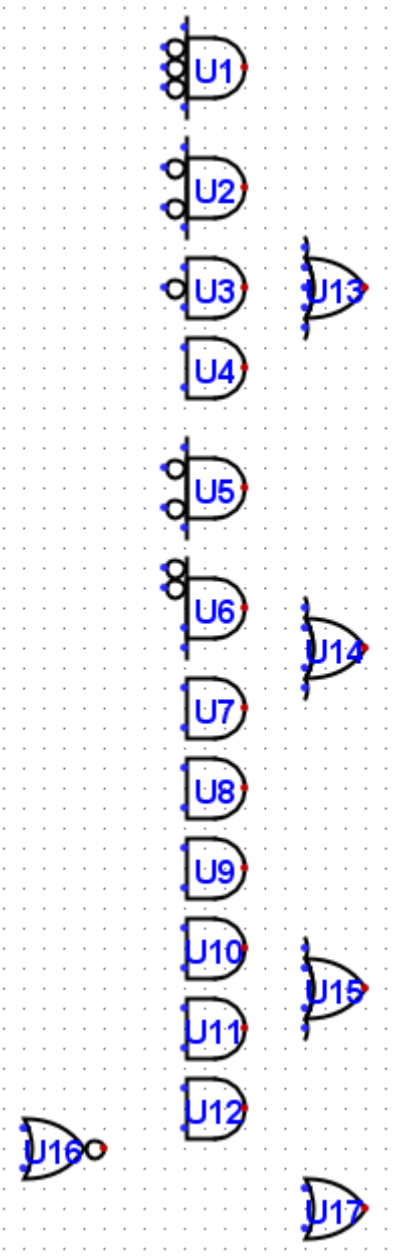

Figure 9.2: OR Gates Added

This encoder is designed to prioritize nine input lines so nine inputs must be added, as illustrated in Figure 9.3.

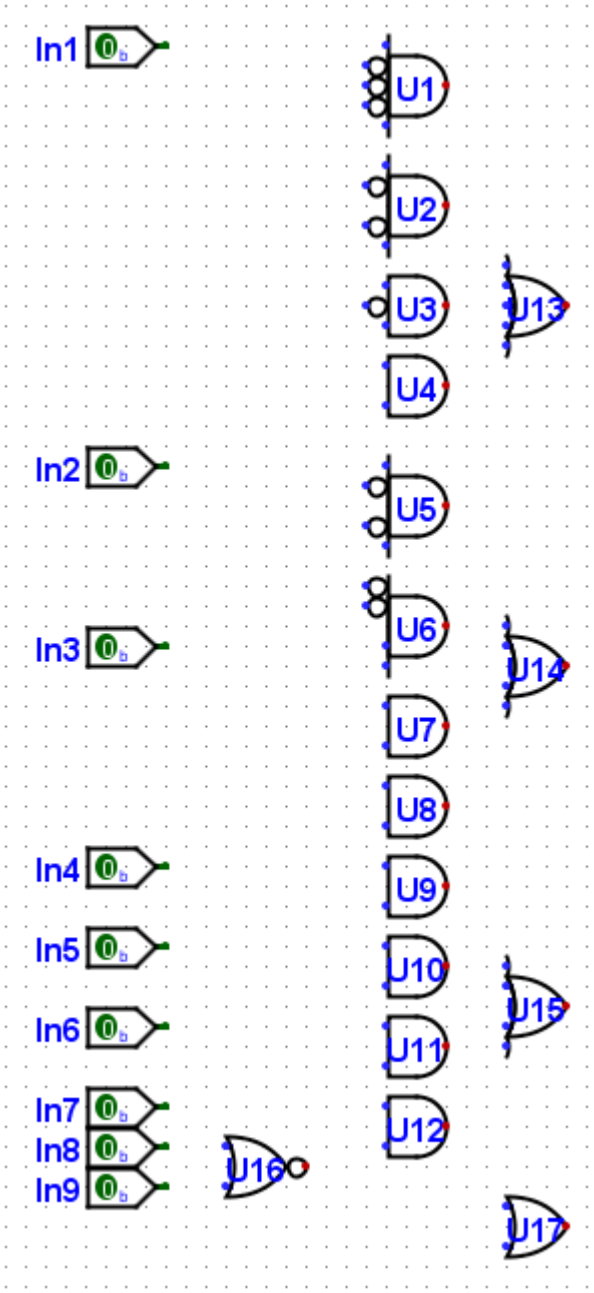

Figure 9.3: Inputs Added

Wiring this circuit is the most challenging part of the build. As illustrated in Figure 9.4, the inputs are wired to several different `AND` gates.

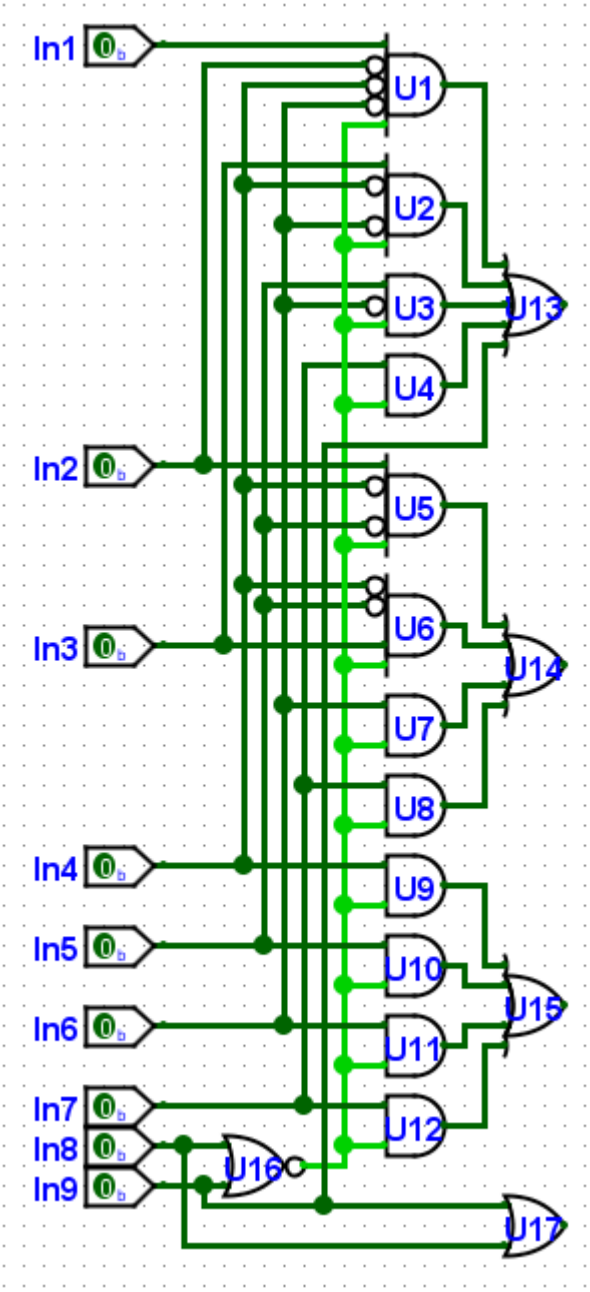

Figure 9.4: Wiring the Encoder

Finally, four output ports are added, as illustrated in Figure 9.5.

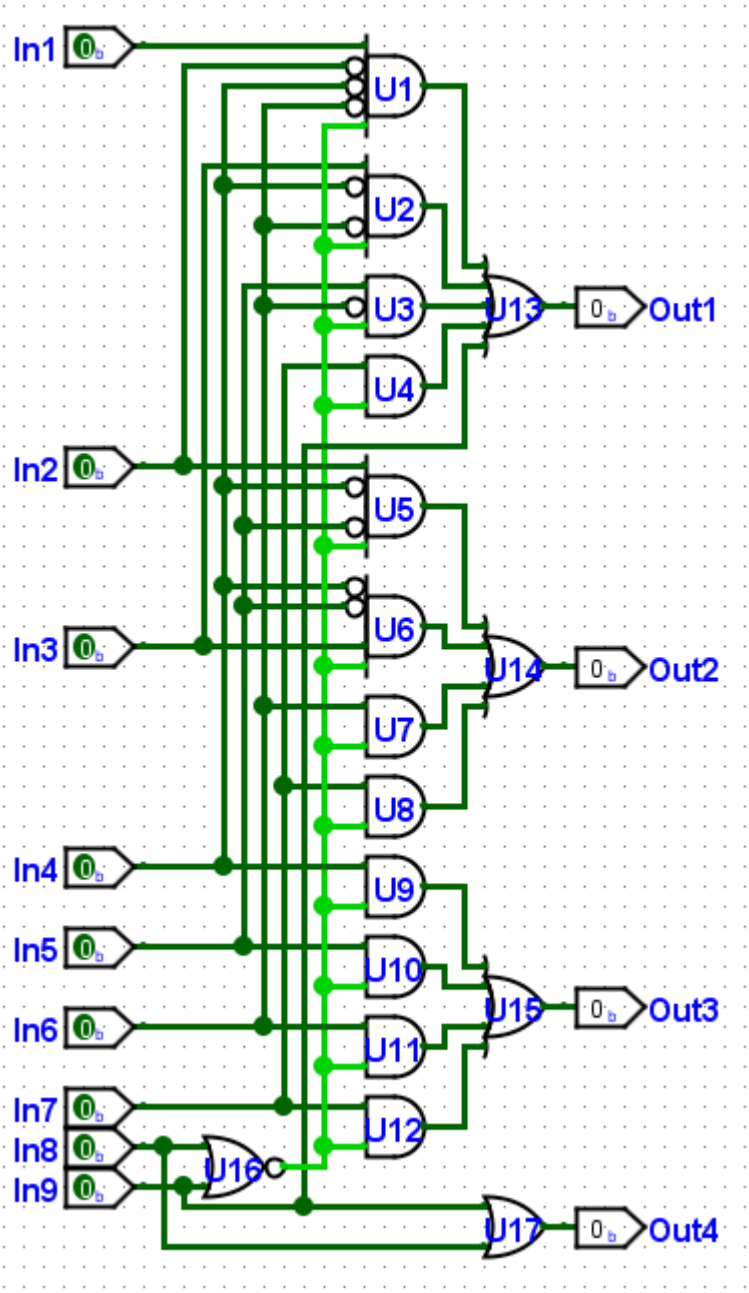

Figure 9.5: Nine-line Priority Encoder

This circuit is designed to output a Binary Coded Decimal (BCD) number, so no further conversion is needed to be able to read the highest priority input line. At this point, the circuit is complete and the *poke* tool can be used to change the inputs and observe how that high input bit drives the outputs.

To finish the project, open the `main` circuit and drop the `Encoder` on the drawing canvas. Add nine inputs and label them *In1* through *In9*. Place a four-bit output labeled *PriOut* and wire the four outputs through a splitter to that output port. To make it easier to read the BCD number, connect a Hex Digit Display (*Input/Output* library) to

the four-bit bus between the splitter and output port. The completed `main` circuit is illustrated in Figure 9.6.

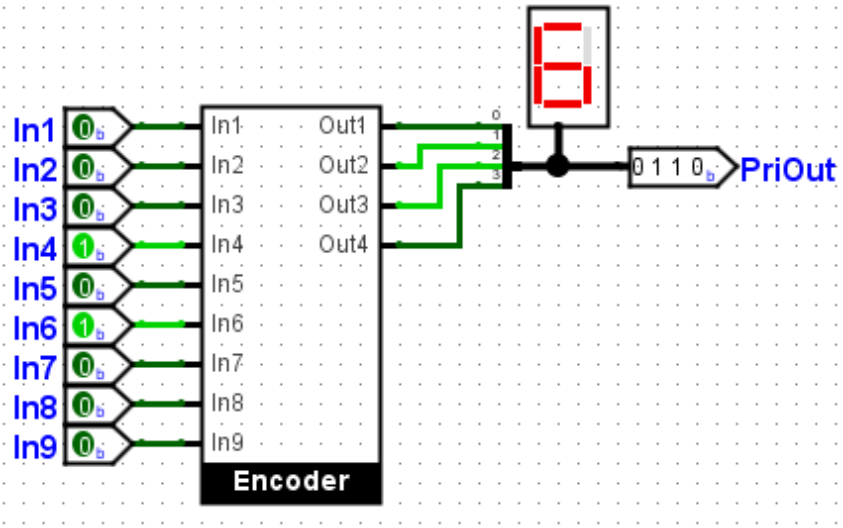

Figure 9.6: Main Circuit

In Figure 9.6, notice that two inputs are selected, *In4* and *In6*. Since *In6* is a higher priority (it is a larger number), the output is set for six and *In4* is ignored.

9.2.1 *Testing the Circuit*

The circuit is now complete. It should be tested by entering various combinations of inputs and observing that the output always displays the highest numbered input.

9.3 DELIVERABLE

To receive a grade for this lab, create the Nine-line Priority Encoder circuit as defined in this lab. Be sure the standard identifying information is at the top left of the circuit, similar to this:

```
George Self
Lab 09: Nine-line Priority Encoder
February 18, 2018
```

Save the file with this name: *`Lab09_Encoder`* and submit that file for grading.

10 TIMER

10.1 PURPOSE

A timer is used to time events. This lab creates a timer where the minimum and maximum counts can be set and counts both up and down. The timer assumes an input clock pulse at 1 Hz (or 60 pulses per minute) but for testing, the clock can be set to any value.

10.2 PROCEDURE

The lab starter circuit includes several versions of the timer as an illustration of the thought process used to develop the final product.

- **Timer_V1**. This is little more than a test of the Counter (*Memory* library) component. The various inputs were wired so both the *Load* and *Up* input pins could be tested. Instead of a clock pulse, a Button (*Input/Output* library) was used for better control over the device. A Bin2BCD (*BFH mega functions* library) device was used for easier interpretation of the output.
- **Timer_V2**. The first circuit was expanded such that both the minimum and maximum counts could be specified. Note that the multiplexer (*Plexers* library) selects whether the minimum or maximum number is loaded depending on whether the count is Up or Down.
- **Timer_V3**. This is the version of the timer that will be completed for this lab.

10.2.1 *Timer_V3*

Complete the circuit to match Figure 10.1.

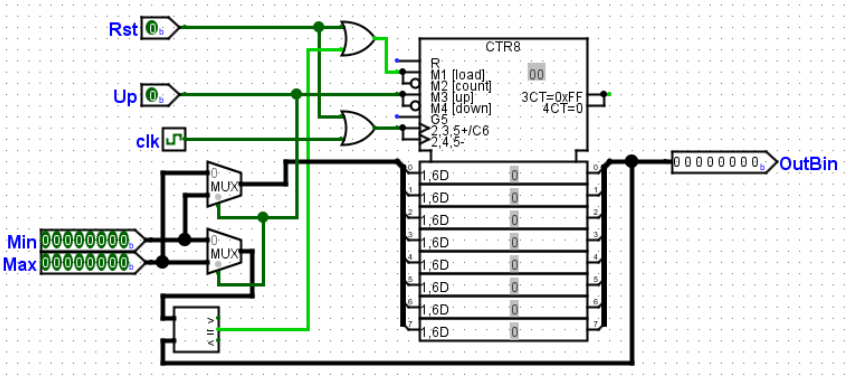

Figure 10.1: Completed Timer

In the timer circuit, the key is the comparator in the lower left corner. That device compares the binary output of the counter to either the minimum or maximum requested value and if they are equal the comparator sends a reset signal to start the count over.

There are two multiplexers with a subtle, but important, difference. The Maximum input value is wired to the top input of the top multiplexer but the bottom input of the bottom multiplexer. The result is the when the count is "Up" the Minimum input is loaded into the counter but the Maximum input is used in the compare, so the counter starts at the minimum and counts up to the maximum. The opposite is true for a "Down" count.

Finally, the BCD output is combined by a splitter (*Wiring* library) into a 12-bit bus for transmission.

10.2.2 *Testing the Circuit*

The `Timer_V3` subcircuit should be added to the `main` circuit and wired as in Figure 10.2.

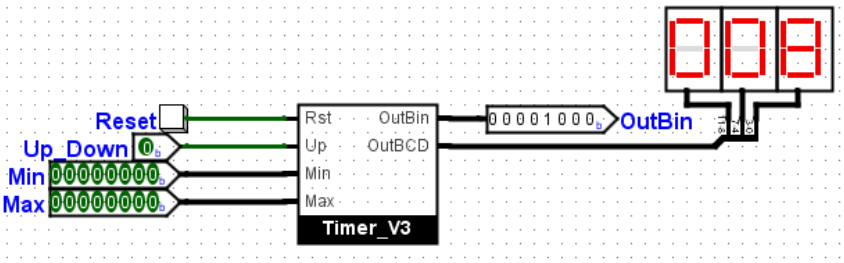

Figure 10.2: Timer Main Circuit

To test the circuit:

1. Enter binary four for a minimum value and eight for a maximum value. (Actually, any values can be entered but four and eight are enough to test the circuit.)

2. Poke *Up_Down* to change its value to one so the circuit counts up.
3. Poke the Reset button and observe that the BCD out changes to 004.
4. Activate the clock SIMULATE -> TICKS ENABLED and observe that it counts up from four to eight and then resets to four. If the speed of the timer is not reasonable then the SIMULATE -> TICK FREQUENCY can be adjusted.
5. Poke *Up_Down* to change the count to down and observe that the timer now counts from eight to four and resets.

10.3 CHALLENGE

As designed, the output of this circuit is an integer count. If it were set for counting seconds then the count of seconds would increase from 59 to 60 then 61 rather than going 0:59, 1:00, 1:01 as expected. Rewrite the `Timer_V3` subcircuit so the output is two BCD numbers: minutes and seconds. As a hint, the Divider (*Arithmetic* library) device products an integer ("modulus") division along with a remainder. It should help to divide the count by 60, use the whole number as "minutes" and the remainder as the "seconds."

10.4 DELIVERABLE

To receive a grade for this lab, complete the Challenge. Be sure the standard identifying information is at the top left of the `main` circuit, similar to:

```
George Self
Lab 10: Timer
March 1, 2018
```

Save the file with this name: *`Lab10_Timer`* and submit that file for grading.

11

ROM

11.1 PURPOSE

This lab introduces students to Read Only Memory (ROM) and builds a fun application: The *Magic 8-Ball*. This was a toy that was developed in the 1950s and was popular throughout the 1960s. It was a small plastic sphere with the markings of an 8-ball. If the user "asked it a question" and then turned the toy upside down the answer would magically appear in a small window on the bottom of the ball.

11.2 PROCEDURE

Start a new *Logisim-Evolution* project and create a subcircuit named `Magic_8_Ball`. Open that circuit and place a ROM (*Memory* library) device near the center of the drawing canvas. Set the ROM properties for an *Address Bit Width* of 12 and a *Data Bit Width* of 8[1].

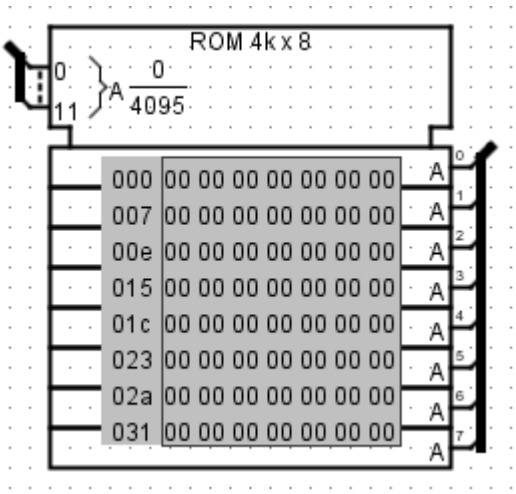

Figure 11.1: Placing ROM

A ROM stores data that is accessed by setting an address on the inputs at the top left of the device and then reading the contents of that address on the 8-bit bus on the right side of the device. By attaching a counter to the ROM address port several consecutive addresses can be "stepped through" to output a message. Attach a Counter (*Mem-*

1 The provided starter circuit already contains the `Magic_8_Ball` subcircuit along with two devices needed in the early part of the build.

ory library) with 12 Data Bits to the address port of the ROM, as in Figure 11.2.

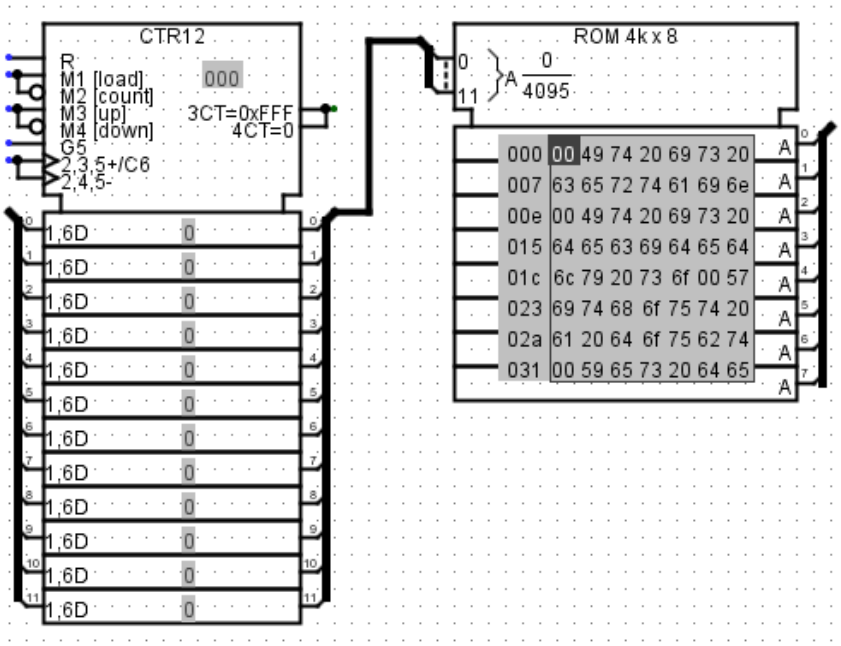

Figure 11.2: ROM With Counter

According to Wikipedia[2], the Magic 8-Ball featured 20 sayings:

```
 1 001 It is certain
 2 00f It is decidedly so
 3 022 Without a doubt
 4 032 Yes definitely
 5 041 You may rely on it
 6 054 As I see it yes
 7 064 Most likely
 8 070 Outlook good
 9 07d Yes
10 081 Signs point to yes
11 094 Reply hazy try again
12 0a9 Ask again later
13 0b8 Better not tell you now
14 0d1 Cannot predict now
15 0e4 Concentrate and ask again
16 0fe Do not count on it
17 111 My reply is no
18 120 My sources say no
19 132 Outlook not so good
20 146 Very doubtful
```

2 https://en.wikipedia.org/wiki/Magic_8-Ball

The *Magic 8-Ball* simulator built in this lab uses those same 20 saying. In the above chart, each saying is numbered and the start point in ROM (using hexadecimal notation) for each saying is also noted. Thus, saying one starts on ROM byte 001, saying two starts on ROM byte 00f, saying three starts on ROM byte 022, and so forth.

The content of the ROM device must be loaded before it can be used and that content is provided in `Lab09_ROM.txt` accompanying this lab. To load the ROM device, click it one time and then click the "(click to edit)" link in its properties panel. In the ROM editor window that pops up, click the "open" button and navigate to the ROM memory file. Click "close window" to load the ROM device and make it ready for service[3].

The start point for each saying, as indicated on the above table, is stored in a Constant (*Wiring* library) then a Mux (*Plexers* library) with five select bits is used to transmit a message start location to the counter so it can be read from the ROM device. Figure 11.3 illustrates the circuit at this point[4].

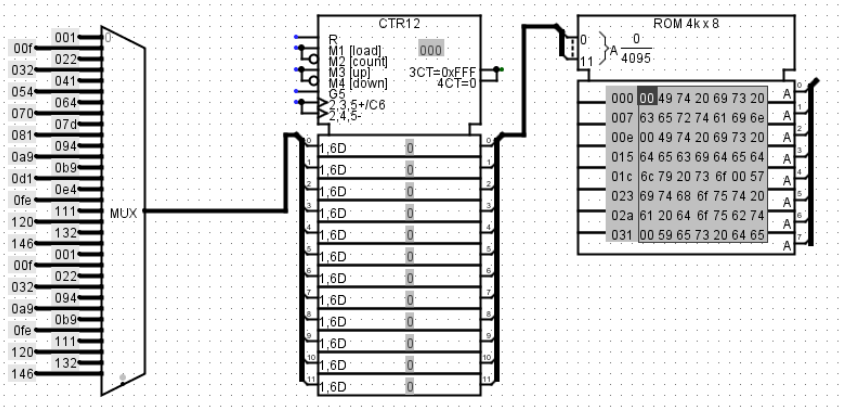

Figure 11.3: ROM Filter Mux

A five-bit Random Generator (*Memory* library) is used to select a random message. Figure 11.4 illustrates the placement of the random generator.

3 The ROM device provided with the starter circuit is pre-loaded so it will not be necessary to load it again. However, this information is left here for students who may want to load the ROM for practice.

4 The multiplexer provided with the starter circuit already has the various constants attached. Students who wish to do so can create their own multiplexer by using the start addresses in the "Sayings" listing above.

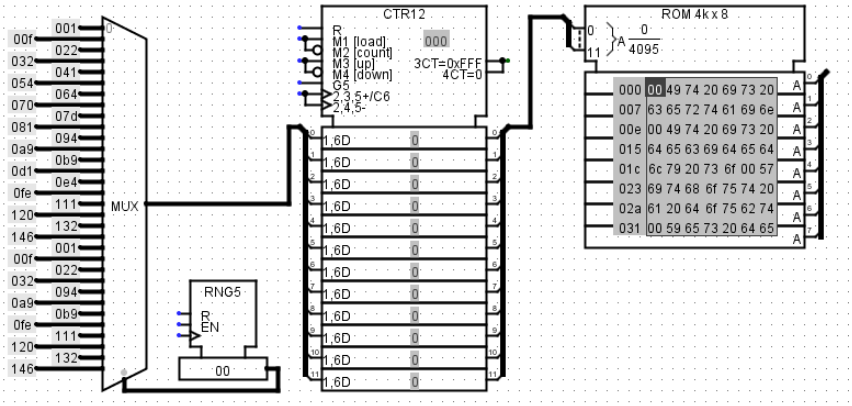

Figure 11.4: Random Generator Added

To complete the circuit, a few odds-and-ends were added. Figure 11.5 shows the completed circuit, but details from that figure are used below to describe how to complete the circuit.

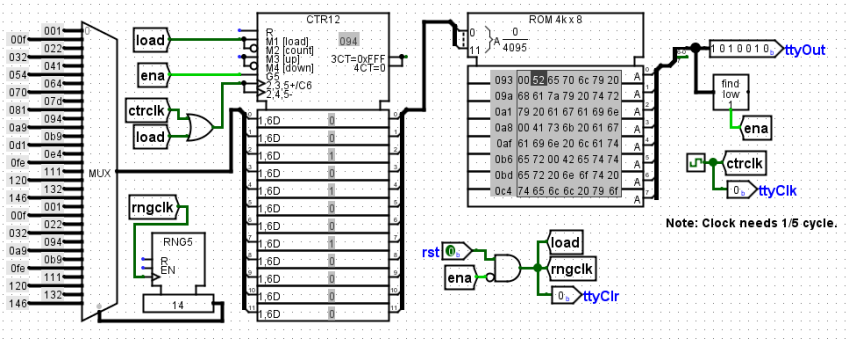

Figure 11.5: Completed Magic 8-Ball Circuit

To set up the counter, four signals are needed. These are all from tunnels (*Wiring* library) connected to other spots on the circuit. (See Figure 11.6.)

- **load**. The load signal goes high when the counter should be loaded with a new number from the multiplexer. The number loaded is the location in ROM for the start of a message. Notice that the load signal is used on two pins. The top pin places the counter in load mode while the bottom pin uses the load signal as a clock pulse.
- **ena**. The enable signal turns the counter on/off. When enable is high then the counter functions normally and when it is low the counter is disabled.
- **ctrclk**.The counter clock provides the clock signal for the counter.

Connect the random number generator as follows. (See Figure 11.6.)

- The clock input pin is connected to a “rngclk” tunnel.

- The generator output is wired to the select port of the multiplexer.

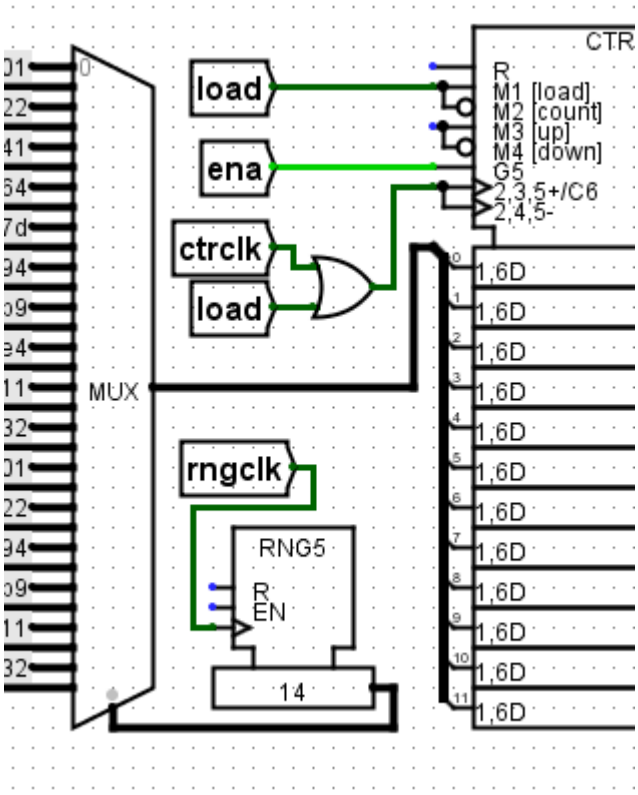

Figure 11.6: Counter Inputs

The counter control signals are generated and distributed from a small group located under the ROM device. The purpose of this tiny group is to transmit a high signal through the AND gate when the reset pin goes high while enable is low. This generates the signals needed to select a new random message and put the starting address of that message in the counter. (See Figure 11.7.)

- **rst**. The reset pin is an external signal that originates from the `main` circuit.
- **ena'**. Enable Not originates from the ROM output group.
- **load**. This signal is used to load a message starting address into the counter. When it goes high it activates the "load" function and also becomes a single clock pulse for the counter.
- **rngclk**. The random number generator clock signal activates that device so it generates a random number. That number is then used to select a single line from the multiplexer so a message starting address can be loaded.

- **ttyClr**. This sends a high signal to the TTY "clear" pin on the `main` circuit. That signal is used to clear the TTY device.

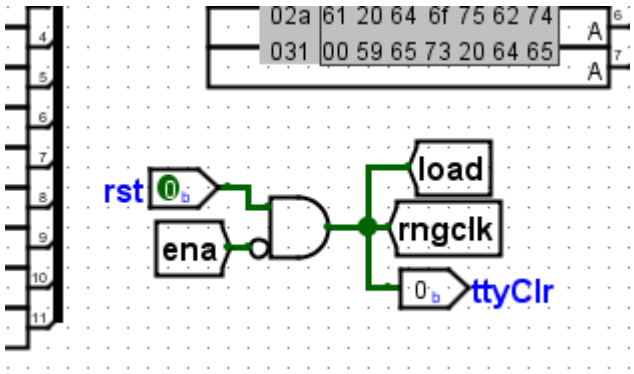

Figure 11.7: Counter Control Generation and Distribution

There are two functions found at the ROM device output. (See Figure 11.8.)

- The output of the ROM device is connected to the *ttyOut* pin in order to drive the teletype device on the `main` circuit.[5]
- The Bit Finder (*Arithmetic* library) attached to the output of the ROM device is used to find the lowest-order *one* in the ROM byte output. If the ROM byte includes at least one *one* then the south port of the finder is high. If the ROM output is all zeros then the Finder output it goes low and that is used as the *ena* signal for the counter and random number generator. When the enable signal is low it also permits a *rst* signal (generated on the `main` circuit when the user "asks another question") to create a new answer.
- Near the output of the ROM device a clock signal is split to two outputs. One is the *ctrclk* tunnel that is used by the counter and the other is the *ttyClk* pin, which is used on the `main` circuit to clock the teletype device. **It is important to note that the clock properties are set for a 1 tick high duration and 5 ticks low duration (a 1/5 clock).**

5 Note that at the output of the ROM device is a splitter. ASCII letters are only seven bits wide so this splitter passes bits 0-6 to the *ttyOut* port but bit 7 (the most significant bit) is simply discarded. The provided starter circuit includes the splitter.

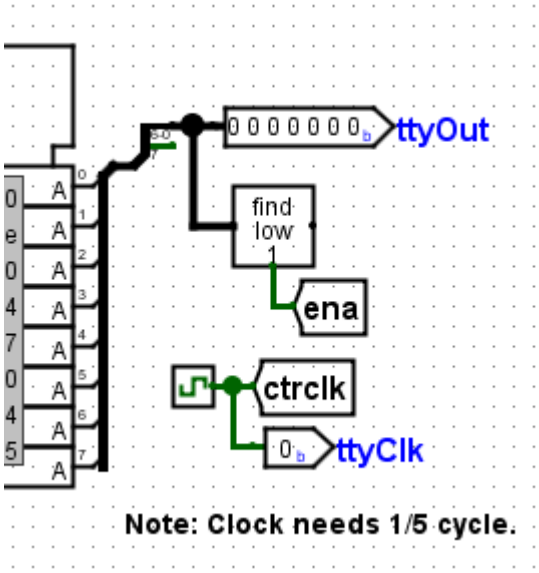

Figure 11.8: ROM Output

The only remaining step is to create the `main` circuit. As in all labs in this manual, the `main` circuit does nothing more than provide a user interface for the *Magic 8-Ball* Circuit. Figure 11.9 illustrates the `main` circuit.

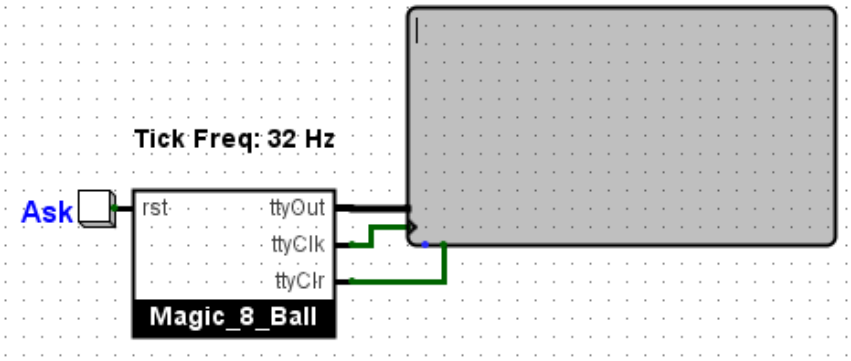

Figure 11.9: Magic 8-Ball Main Circuit

11.2.1 *Testing the Circuit*

Before the circuit can be tested the ROM device must be loaded. The ROM was loaded earlier in the lab but in case it does not have any content (it is filled with zeros), then load it with *`Lab09_ROM.txt`*, which was provided with the lab. To load the ROM device, click it one time and then click the "(click to edit)" link in its properties panel. In the ROM editor window that pops up, click the "open" button and find the ROM memory file. Click "close window" to load the ROM device and make it ready for service.

The circuit should be tested by enabling the simulator clock at a frequency of 32 Hertz. Every time the *Ask* button is pressed a new random message will be displayed on the teletype screen[6].

11.3 DELIVERABLE

To receive a grade for this lab, build this circuit. Be sure the standard identifying information is at the top left of the `main` circuit, similar to:

```
George Self
Lab 11: ROM
September 13, 2019
```

Save the file with this name: *Lab11_ROM* and submit that file for grading.

6 Due to the way this circuit is constructed one out of six button presses will fail and no message will be displayed. The failures are random events so the circuit may fail several times in a row but then not fail for the next 20 or more presses. Students may want to investigate this bug but that is not required.

Part IV

SIMULATION

SIMULATION is the most complex topic covered in this lab manual. Included in this manual are a vending machine simulator, a simple processor, designed to teach the foundations of a Central Processing Unit, and an elevator simulator, designed to be a capstone project.

12

VENDING MACHINE

12.1 PURPOSE

One of the important benefits of working with *Logisim-Evolution* is being able to simulate real-world circuits before they are physically built. This lab simulates a vending machine that meets these requirements:

1. The customer can input the following coins: 5-cent, 10-cent, 25-cent.
2. When 75 cents is input, the machine will activate the dispenser and permit the customer to select a product.
3. When at least 75 cents is input no more coins will be accepted.
4. Change will be returned to the customer if more than 75 cents is deposited.
5. A reset button will return the customer's money.
6. When a product is dispensed, 75 cents will be added to the machine's "Total Money Collected" register.
7. No product is dispensed if less than 75 cents is deposited.
8. The current number of items available for each product is stored in a counter.
9. When a service technician restocks the machine the item count for each product is set to 15, which is the maximum number of items that can be stocked.
10. If the number of products available is zero for any one product the machine will light a "sold out" light and no action will be taken if that product is selected.

This circuit uses only combinational logic and is an example of a reasonably complex system.

12.2 PROCEDURE

The starter circuit for this lab is almost complete, but three of the requirements have not been met.

- Requirement three is that the coin input will stop once 75 cents is reached but this is not working so customers can continue depositing coins into the machine.
- When a product is dispensed, the coins deposited and change returned is not reset back to zero. This means that a customer could deposit 75 cents and then keep selecting products until the machine is empty.
- Requirement six is that the machine totals all of the money collected but that is not functional.

12.2.1 *Testing the Circuit*

To test the circuit:

1. Ensure simulation is enabled at SIMULATE -> SIMULATION ENABLED.
2. Poke the *Ena* input pin to enable the vending machine simulator.
3. Notice that the *SoldOut1*, *SoldOut2*, and *SoldOut3* LEDs are lit, indicating that those products are sold out.
4. Restock products by poking the *Restock1* and *Restock2* buttons. For this test, do not poke *Restock3* to keep that product empty. As a product is restocked the "SoldOut" LED for that product goes out and the *Prodo1* and *Prodo2* counts change to 15.
5. Poke the *In5*, *In10*, and *In25* buttons to deposit coins. The total deposited is displayed and any amount over 75 cents is shown as change. Notice that the deposit circuit is not disabled after 75 cents is reached so customers can continue depositing coins.
6. Once at least 75 cents is deposited, poke *Vend1* to vend that product. When the button is poked the *Dispense1* LED momentarily lights to indicate that a product was sold. The number of items available for that product decreases. Notice that once a product is dispensed the amount of money deposited is not reset and the machine can dispense additional products without additional money being deposited.
7. Poke *Vend3* and notice that nothing happens since that product is sold out.
8. Poke *Reset* to reset the amount of money deposited.

12.2.2 *Subcircuit Descriptions*

This simulator contains five subcircuits in addition to the `main` circuit and this section describes all of those components.

12.2.2.1 *main*

The `main` circuit is the interface between a human customer and the simulator, as shown in Figure 12.1.

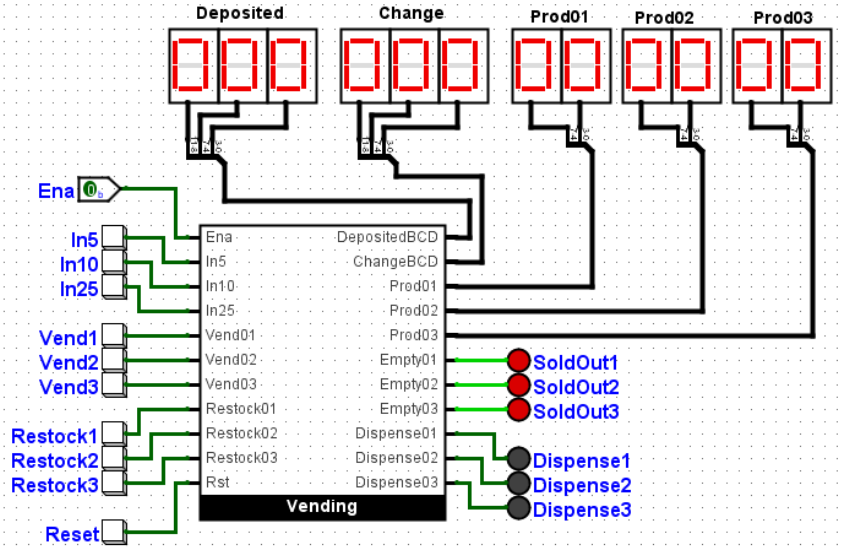

Figure 12.1: Vending Machine Main Circuit

The `main` circuit includes the following components.

- Numeric displays for the amount deposited, the change returned, and the number of items available for each of three products.
- An *Ena* (*Enable*) input so a technician can disable the machine for servicing.
- Buttons to simulate depositing coins, vending products, and restocking the machine.
- LEDs to indicate when products are sold out and dispensed.

12.2.2.2 *Activator*

The `Activator` subcircuit receives a signal from the `Bank` subcircuit that indicates how much money has been collected. The `Activator` returns the BCD Total and Change values and sets a signal to activate the `Dispenser` subcircuit once 75 cents has been deposited. Figure 12.2 illustrates the `Activator` subcircuit.

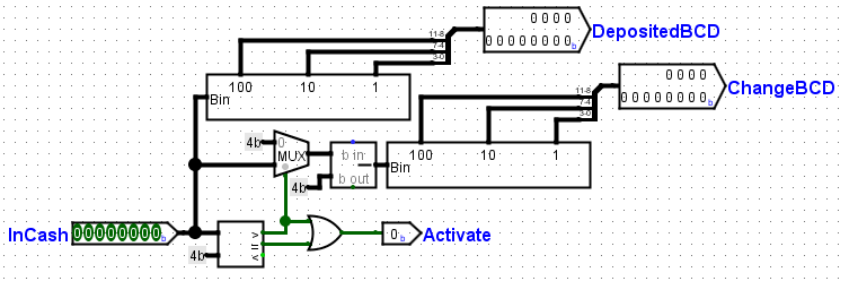

Figure 12.2: Activator Subcircuit

The `Activator` subcircuit has only one input, *InCash*. That input is connected to the `Bank` subcircuit output and contains the total amount of cash deposited. That input is connected to a Bin2BCD (*BFH mega functions* library) device and is then output as a BCD number on the *DepositedBCD* output pin.

The *InCash* input is also sent to a comparator where the amount is compared to 75. If the amount in the bank is equal to or greater than 75 then the `Activate` output goes high.

Finally, the *InCash* input is sent to a mux that outputs 75 until the comparator indicates that more than 75 is in the bank, then the mux passes the *InCash* amount to a subtractor where 75 is subtracted from it and the result sent to the *ChangeBCD* output.

12.2.2.3 *Bank*

The `Bank` subcircuit keeps a running total of the amount deposited and sends that total to the `Activator` subcircuit. Figure 12.3 illustrates the `Bank` subcircuit.

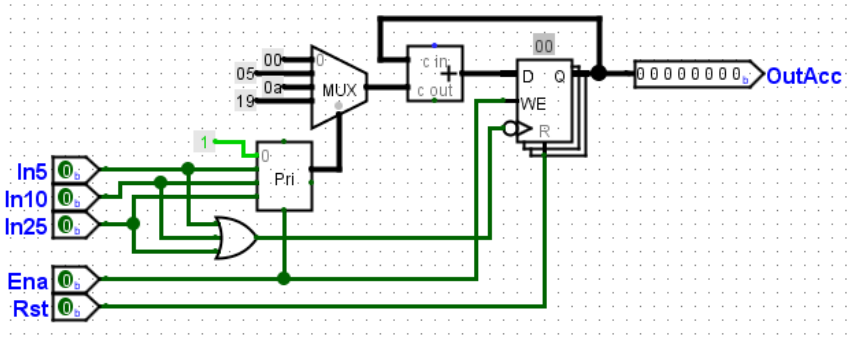

Figure 12.3: Bank Subcircuit

The `Bank` subcircuit has five inputs. *In5*, *In10*, and *In25* indicate the value of the coin dropped into the machine. When high, the *Ena* input enables the `Bank`. When high, the *Rst* input resets the total to zero.

The `Bank` subcircuit has only one output, *OutAcc*, that makes the total cash accumulated available to the `Activator` subcircuit.

For this description, imagine that a 5-cent coin is deposited. *In5* goes high which changes the output of the priority encoder from zero

to one. That output is sent to a mux control where the number five, on mux input one, is passed to an adder. The output of the adder is sent to a register where it is remembered. The output of the register is sent to the *OutAcc* pin but is also looped back to the adder so each new coin is added to the previous total. Thus, the register keeps a running total of the money deposited.

The final logic function in this subcircuit is a three-input `OR` gate where each of the coin input pins are sent to the clock input of the register. As coins are dropped into the machine the register is clocked in order to capture each new deposit. It is important to note that *the register is set to activate on a falling edge* in order to give the input signal enough time to propagate through the priority encoder, mux, and adder.

12.2.2.4 *Dispenser*

The `Dispenser` subcircuit dispenses the three products available in the machine. Figure 12.4 illustrates the `Dispenser` subcircuit.

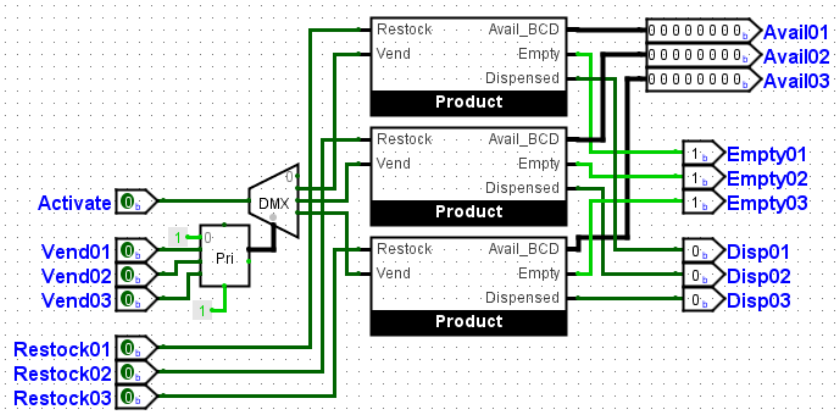

Figure 12.4: Dispenser Subcircuit

The Dispenser subcircuit has seven inputs and nine outputs.
Inputs:

- **Activate**. A high input on this pin permits a product to be dispensed. This signal is generated in the `Activator` subcircuit.
- **Vend**. These inputs cause one of three products to be dispensed.
- **Restock**. This resets the product count to 15, simulating a service technician restocking the machine.

Outputs:

- **Avail**. This is an 8-bit number (not BCD) that shows how many items each of the products have available for sale.
- **Empty**. This pin goes high when any product is sold out.

- **Disp**. This pin goes high when an item is dispensed.

Overall, this is a rather simple subcircuit. When one of the *Vend* inputs goes high the priority encoder sends the number for that input to the demux control port. Thus, if a customer selects product one then the priority encoder transmits a one to the demux.

The demux will transmit the value present on the *Activate* input to one of three `Product` subcircuits. When *Activate* is low then a zero is transmitted to the `Product` subcircuit which effectively disables the dispenser function. However, if *Activate* is high then a one is transmitted to one of the `Product` subcircuits and that will cause a product to be dispensed.

12.2.2.5 *Product*

The `Product` subcircuit keeps count of the number of items available for a product. There are two inputs and three outputs.

Inputs:

- **Restock**. This resets the count of the item to 15. It is designed to simulate a service technician restocking the machine.
- **Vend**. When this goes high a single item is dispensed.

Outputs:

- **AvailBCD**. This is a count, in BCD, of the number of items available for sale.
- **Empty**. This goes high when there are no items available for sale.
- **Dispensed**. This goes high when an item is dispensed. It represents an item physically dropping out of the machine for the customer to retrieve.

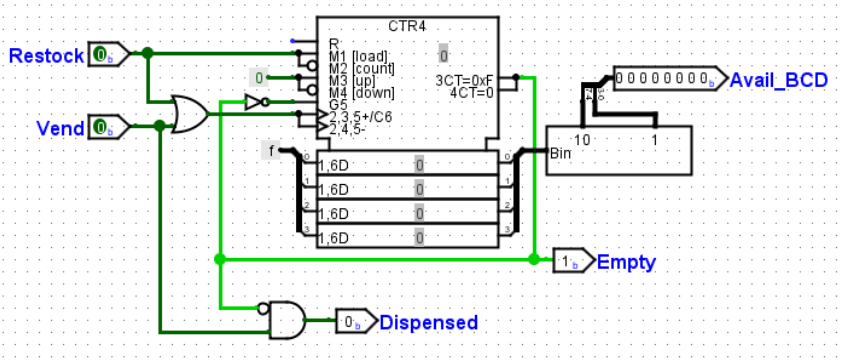

Figure 12.5: Product Subcircuit

This subcircuit is nothing more than a counter with a few controlling signals. The counter has a constant zero input on the *M3* port. That sets the counter to decrement the count on each clock pulse.

The *Restock* input is wired to the counter's reset port and a high input will reset the counter to 15. Note, the counter's properties are pre-set for a maximum count of 15.

The *Vend* input is wired to the counter's clock port so when an item is sold the count will decrease. This input is also wired to the *Dispensed* output to indicate that an item was sold.

The counter has two outputs. The *3CT=oxF* output goes high when the count reaches zero (the item is sold out). That signal is used to disable the counter so no further sales are made. The second counter output is the count it contains and that is wired to a Bin2BCD (*BFH mega functions* library) device. The output of that device is sent to the *AvailBCD* port for other subcircuits to use.

12.2.2.6 *Vending*

The `Vending` subcircuit consolidates the other subcircuits into an IC that is used in the `main` circuit. Figure 12.6 illustrates the `Vending` subcircuit.

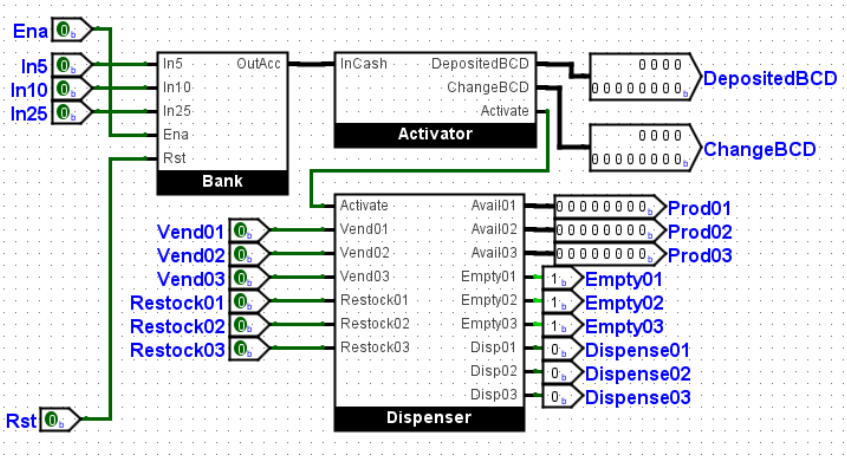

Figure 12.6: Vending Subcircuit

No further explanation is given for this subcircuit since it only wires the other subcircuits together and introduces no new logic.

12.3 CHALLENGE

The Vending Machine simulator has three vital flaws that must be corrected.

- Requirement three is that the coin input will stop once 75 cents is reached but this is not working so customers can continue depositing coins into the machine.

- When a product is dispensed, the coins deposited and change returned is not reset back to zero. This means that a customer could deposit 75 cents and then keep selecting products until the machine is empty.
- Requirement six is that the machine totals all of the money collected but that is not functional.

12.4 DELIVERABLE

To receive a grade for this lab, correct all three flaws identified in the Challenge. Be sure the standard identifying information is at the top left of the *main* circuit, similar to:

```
George Self
Lab 12: Vending Machine
February 16, 2018
```

Save the file with this name: *Lab12_Vend* and submit that file for grading.

13 PROCESSOR

13.1 PURPOSE

A Central Processing Unit (CPU) is arguably one of the most important digital logic devices. CPUs are found in all computers and many other embedded logic devices. They are versatile circuits that can be used to control many processes and peripheral devices. The purpose of this lab is to lay the foundation of CPU operation.

13.1.1 *A Definition*

When asked to define "CPU" many students offer poetic definitions like "it is the brain of the computer." This may be somewhat artistic but is not very helpful in defining CPU for digital logic purposes. Here is a much better definition:

> A Central Processing Unit (CPU) is a hardware device that is designed to translate binary codes stored in software into signals that control hardware. Thus, a CPU is the interface between software and hardware.

The purpose of this lab is to demonstrate how binary codes can be used to manipulate hardware devices, like registers and adders, to move data through a circuit and accomplish a purpose. While the circuit developed in this lab is not a practical start for a CPU is does serve as an introduction to the concept of hardware manipulation by software codes.

13.2 PROCEDURE

This processor contains only three subcircuits connected by several bus lines and each of the three subcircuits are reasonably simple to understand.

13.2.1 *Arithmetic-Logic Unit*

This processor starts with a simple ALU, as in Figure 13.1.

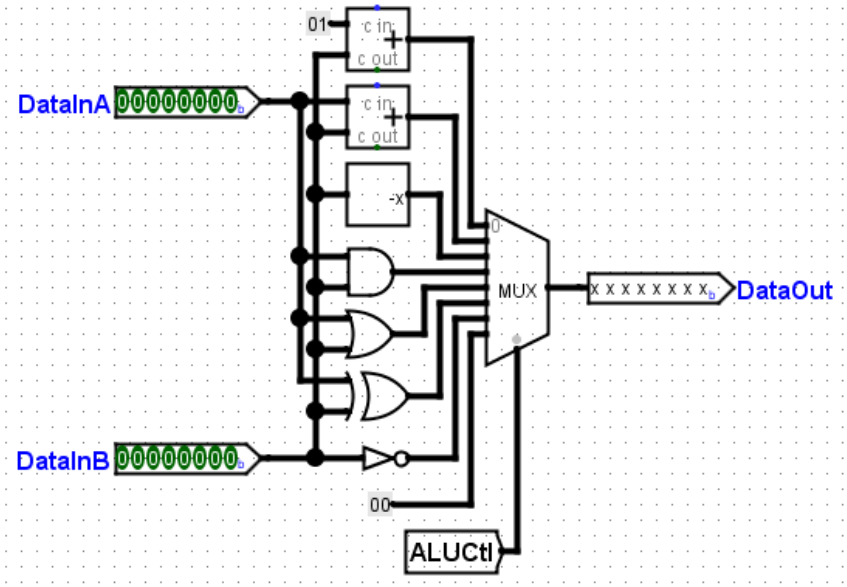

Figure 13.1: Simple ALU

To be sure, this ALU is not very complex but uses the same principles developed in Lab **??**, **??**. It contains only three arithmetic functions, increment, add, and negate; four logic functions, `AND`, `OR`, `XOR`, `NOT`; and one constant zero output. There are two data input ports but note that some of the functions only use the lower input, and one output port. The multiplexer determines which of the functions will be connected to the output and that is controlled by a signal named *ALUCtl*.

The ALU is then expanded somewhat to make it usable in a CPU. For simplicity, Figure 13.2 shows only the left side of the ALU.

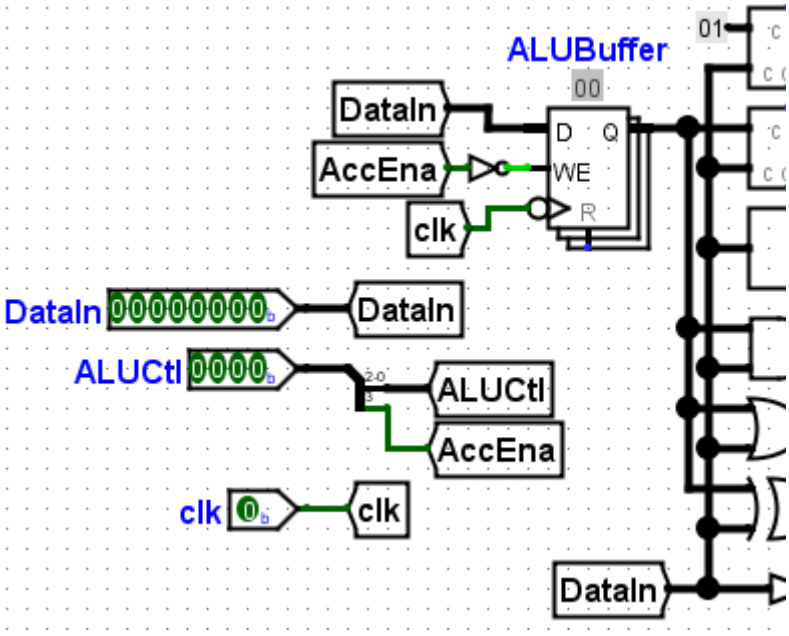

Figure 13.2: Left Side of ALU

Figure 13.3 shows the right side of the ALU.

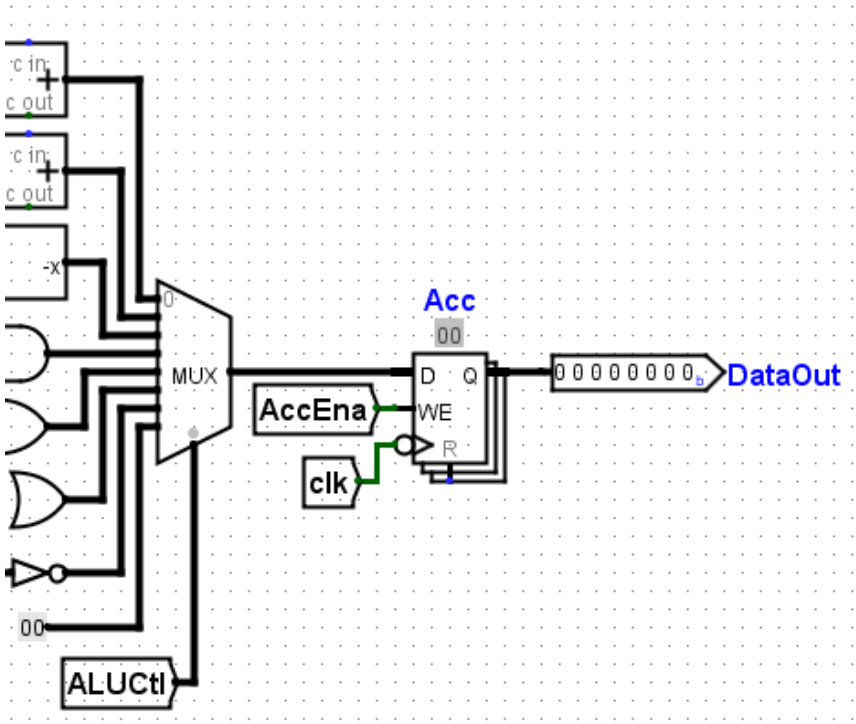

Figure 13.3: Full ALU

The simple ALU functions are found in the center of Figure 13.2. However, what started as *DataInA* has been replaced by a register

named *ALUBuffer*.[1] The *ALUBuffer's* inputs are from Tunnels (*Wiring* library) because those inputs are used in more than one location in the subcircuit.[2]

The ALU output is routed through a register named *Acc*, for *Accumulator*, which is the commonly-used name for the ALU output in a CPU circuit.

On the left side of the subcircuit are the three input ports. *DataIn* is an eight-bit number that is sent to both the *ALUBuffer* and the lower *DataIn* bus. The *ALUCtl* signal is split into two components. Bits 0-2 are sent to the multiplexer to select which of the eight functions will be output. Bit 3 of the *ALUCtl* signal is sent to the *AccEna* tunnel and when that is high the *Acc* register will be enabled but when that signal is low then the *ALUBuffer* register will be enabled. Finally, the clock input is sent to both registers.

13.2.2 *General Registers*

A CPU must have several general registers available to hold data temporarily while an instruction is being carried out. For example, it may be necessary to hold the *Acc* output until it is needed in a later step so that value can be stored in a register and then recovered when needed.

The processor circuit being built in this lab has four general registers. Figure 13.4 illustrates the `GenReg` subcircuit.

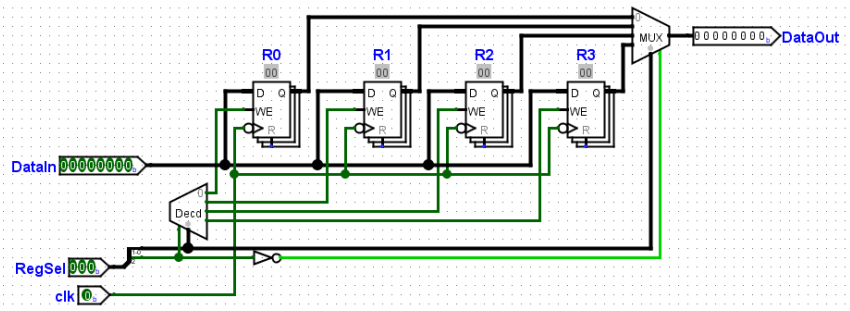

Figure 13.4: General Registers

The `GenReg` subcircuit does not require any novel digital logic concepts. Starting on the left side of the circuit:

- *DataIn* is connected to the data bus and is the main input port for the registers. Note that *DataIn* is connected to the *Data* port on all four registers.

1 IMPORTANT NOTE: All registers in this Processor circuit are triggered on the Falling Edge of the clock. The reason for this will become evident when the circuit is tested.

2 Tunnels are used extensively in this circuit to simplify the diagrams and aid in tracing signals.

- The register that actually stores the input data is determined by the Decoder (*Plexers* library) in the lower left corner of the subcircuit. The two low-order bits from the *RegSel* signal activate one of the output lines from the Decoder and that line is tied to the Write Enable port of the register. On the next clock pulse that register will lock in the data present on the *DataIn* port.
- The outputs from all of the registers are wired to a Multiplexer (*Plexers* library). The select bits from the Decoder that are used to select the storage register are also used to select the register output line which is, in turn, wired to the *DataOut* port.
- The high-order bit from the *RegSel* control signal is used to determine if data are stored to or read from a register. When that bit is high the decoder is active and will select a storage register but when that bit is low the output multiplexer will be activated and send a register's stored value to the output port.

13.2.3 *Control*

The `Control` subcircuit in this device is very simple and could, in all actuality, be eliminated. However, in a true CPU the `Control` subcircuit is rather complex and critical to the operation of the circuit so a `Control` subcircuit is included in this lab as an example. Figure 13.5 illustrates the `Control` subcircuit.

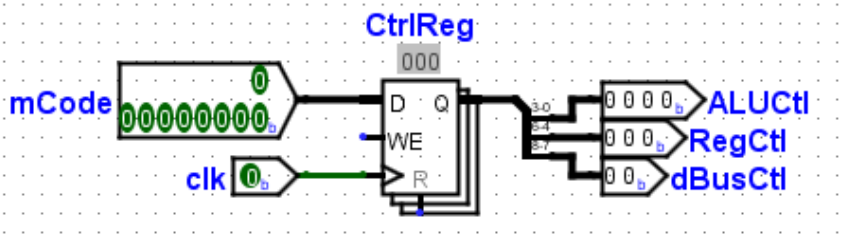

Figure 13.5: Control Subcircuit

The `Control` subcircuit includes a nine-bit input named *mCode* (for "Microcode"). That input is latched by a register[3] and the output of that register is split into three components.

BITS 0-3 These are the ALU control bits and they are sent to the `ALU` subcircuit.

BITS 4-6 These are the register control bits and are sent to that subcircuit.

3 Note, as an exception to the other registers in the Processor circuit, the register in the control subcircuit must be set to trigger on the leading edge of the clock rather than the falling edge.

BITS 7-8 These are the *dBus* ("Data Bus") control bits. The data bus is found in the `main` circuit and carries the data to each of the subcircuits. The dBus control is just a multiplexer that controls which subcircuit's output has control of the data bus.

13.2.4 *Main*

The `main` circuit ties the three subcircuits together with three control busses and one data bus. Figure 13.6 illustrates the `main` circuit.

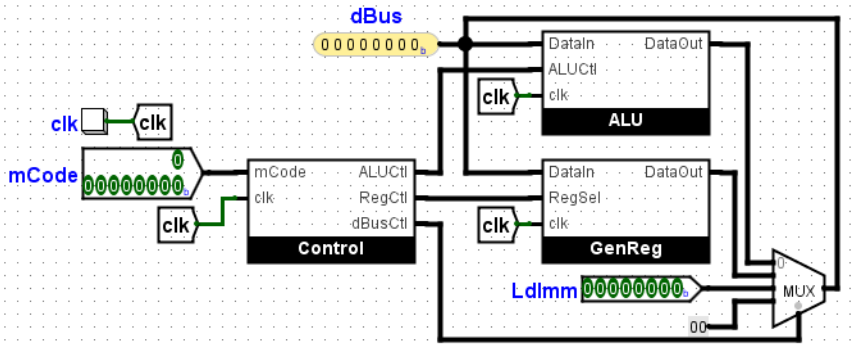

Figure 13.6: Main Circuit

There are no novel digital logic functions used in this circuit. The first input is *mCode* which is the microcode used to control the flow of data in the dBus ("data bus"). the other input, *LdImm* ("Load Immediate") can contain an eight-bit number that is to be loaded into one of the registers for processing. In a full CPU that input would be wired to a Random Access Memory (RAM) device.

13.2.5 *Testing the Circuit*

The circuit should be tested by inputting these signals and observing the output.

13.2.5.1 *Copy LdImm To R0*

Enter some value in the *LdImm* input port, set the *mCode* input to 101000000 (the first three values in the table below), and then pulse the *clk*. When completed, the *dBus* and *R0* should both contain the value of the *LdImm* port.

dBus	Reg	ALU	dBus	Notes
10	100	0000	LdImm	R0 <- LdImm

Table 13.1: R0 <- LdImm

13.2.5.2 *Copy LdImm To R1*

Enter some value in the *LdImm* input port, set the *mCode* input to 101010000 (the first three values in the table below), and then pulse the *clk*. When completed, the *dBus* and *R1* should both contain the value of the *LdImm* port.

dBus	Reg	ALU	dBus	Notes
10	101	0000	LdImm	R1 <- LdImm

Table 13.2: R1 <- LdImm

13.2.5.3 *Copy LdImm To ALUbuf*

Enter some value in the *LdImm* input port, set the *mCode* input to 100000000 (the first three values in the table below), and then pulse the *clk*. When completed, the *dBus* and *ALUbuf* should both contain the value of the *LdImm* port.

dBus	Reg	ALU	dBus	Notes
10	000	0000	LdImm	ALU <- LdImm

Table 13.3: ALU <- LdImm

13.2.5.4 *Increment Ro*

Use the LdImm function to initialize Ro.

Incrementing the value in Ro requires two steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, *Ro* will contain the original value of the *Ro*+1.

dBus	Reg	ALU	dBus	Notes
01	000	1000	R0	Acc <- R0+1
00	100	0000	Acc	R0 <- Acc

Table 13.4: Ro <- Inc(Ro)

13.2.5.5 *Add Ro And R1, Store In Ro*

Use the LdImm function to initialize Ro and R1.

Adding the values of *Ro* and *R1* and storing the result in *Ro* requires three steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, the sum of the original values of *Ro* and *R1* will be stored in *Ro*.

dBus	Reg	ALU	dBus	Notes
01	001	0001	R1	ALU <- R1
01	000	1001	R0	Acc <- R0 + R1
00	100	0001	Acc	R0 <- Acc

Table 13.5: R0 <- R0 + R1

13.2.5.6 *Subtract R1 From R0, Store In R0*

Use the LdImm function to initialize R0 and R1.

Subtracting the value of *R1* from *R0* and storing the result in *R0* requires four steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, the difference of the original values of *R0* and *R1* will be stored in *R0*.

dBus	Reg	ALU	dBus	Notes
01	000	0010	R0	ALUbuf <- R0
01	001	1010	R1	Acc <- ~R1
00	100	1001	R0-R1	dBus <- Acc
00	100	0111	dBus+1	R0 <- R0 - R1

Table 13.6: R0 <- R0 - R1

13.2.5.7 *Copy R0 to R1*

Use the LdImm function to initialize R0.

Copying the value of *R0* to *R1* requires four steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, the value of *R0* will be stored in *R1*.

dBus	Reg	ALU	dBus	Notes
00	000	1111	0	dBus <- 0
00	000	0100	0	ALU <- dBus
01	000	1100	Acc	Acc <- ALU OR R0
00	101	0111	Acc	R1 <- Acc

Table 13.7: R1 <- R0

13.2.5.8 *Swap R0 And R1*

Use the LdImm function to initialize R0 and R1.

Swapping the values of *R0* and *R1* requires 12 steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, the values of *R0* and *R1* will exchanged.

dBus	Reg	ALU	dBus	Notes
00	000	1111	0	dBus <- 0 (Move R0 to R2)
00	000	0100	0	ALU <- dBus
01	000	1100	Acc	Acc <- ALU OR R0
00	110	0111	Acc	R2 <- Acc
00	000	1111	0	dBus <- 0 (Move R1 to R0)
00	000	0100	0	ALU <- dBus
01	001	1100	Acc	Acc <- ALU OR R1
00	100	0111	Acc	R0 <- Acc
00	000	1111	0	dBus <- 0 (Move R2 to R1)
00	000	0100	0	ALU <- dBus
01	010	1100	Acc	Acc <- ALU OR R2
00	101	0111	Acc	R1 <- Acc

Table 13.8: Ro <-> R1

13.3 ABOUT PROGRAMMING LANGUAGES

The codes that were input for the last example (swap *Ro* and *R1*) would create the following program.

```
000001111
000000100
010001100
001100111
000001111
000000100
010011100
001000111
000001111
000000100
010101100
001010111
```

This group of instructions would be considered "CPU Microcode," which is a very highly specialized form of programming. It is the code that is built into a CPU circuit and it determines what gates, registers, and other devices are active for each step of the code. When Intel, AMD, Motorola, or other manufacturers create a new CPU, one of their main challenges is creating the microcode that will, for exam-

ple, "add the contents of register one to the contents of register two and store the result in register zero." The microcode must be able to activate and deactivate various devices within the CPU so data appear on the appropriate bus at the right time in order to achieve the objective. Normally, microcode steps must be executed over several clock cycles in order to do a single job. For example, in one clock cycle the contents of register one may be placed on the data bus, the next clock cycle will load that data into the ALU register, and so forth until the entire process is complete.

Microcode is usually stored in ROM that is built into the CPU. This is typically called "firmware" since it is a string of ones and zeros, like software, but it cannot be changed, like hardware.

It is important to keep in mind the difference between instructions contained in a software program (like Word) and those contained in microcode. A single instruction in software is interpreted and executed by the CPU using, perhaps, dozens of microcode steps. As an example, the software may want to move a single byte from RAM to the video card. The CPU may process that instruction by first moving the byte from RAM to register one and then moving it from there to the video card's input register and then activating the video card input function. Those moves may require several clock cycles as various multiplexers and other devices are activated in the correct sequence to move the data to its destination.

A software program, like Word, is nothing more than a series of ones and zeros, organized into groups, commonly 64 in modern computers. Each group of bits forms a single "word" of information; or a single instruction which would then be used by the CPU to trigger a microcode sequence. When viewed at the level of ones and zeros, a software program is said to be in "machine code," and could look something like the following (note, only the first 32 bits of each word are shown).

```
10010100101100101001101011001010
01101001101011000111101011101011
00011011110010000111010111100101
```

If a programmer could master machine code, then those programs would be as concise and efficient as possible since they would be written in machine code the CPU can execute directly. Of course, as it is easy to imagine, no one actually writes machine code due to its complexity.

The next level higher than machine code is called "Assembly" code. Assembly uses easy-to-remember abbreviations to represent the various CPU instructions available; and it looks something like this:

```
INP
STA FIRST
INP
STA SECOND
LDA FIRST
SUB SECOND
OUT
HLT
FIRST DAT
SECOND DAT
```

Once the program has been written in Assembly, it must be "assembled" into machine code before it can be executed. An assembler is a fairly simply program that converts a file containing assembly codes into machine codes that can be executed by the CPU.

Many programming languages have been developed that are considered "higher" than Assembly; for example, C++, Java, and Visual Basic. These languages tend to be easy to master and can enable a programmer to quickly create very complex programs. Programs written in each of these languages must be compiled, or changed into machine code, before they can be executed. Here is an example Java program:

```
public class HelloWorldExample{
  public static void main(String args[]){
  System.out.println("Hello World !");
  }
}
```

In the end, while there are dozens of different programming languages, they are all designed to be reduced into a series of machine codes which the CPU can then execute.

13.4 CHALLENGE

Using the examples in the "Testing the Circuit" section, create the microcode necessary to carry out these functions:

1. Store the value contained in *LdImm* in *R2* (*R2* <- *LdImm*). (Assume that *LdImm* is pre-loaded with the value to store.)
2. Store the value contained in *LdImm* in *R3* (*R3* <- *LdImm*). (Assume that *LdImm* is pre-loaded with the value to store.)
3. Store the 2s complement of the value in *Ro* back into *Ro* (*Ro* <- *~Ro*). The subtraction example will help with this function.
4. Store the bitwise NOT of the value in *Ro* back into *Ro* (*Ro* <- *Ro'*).

13.5 DELIVERABLE

To receive a grade for this lab, build the Processor circuit and then complete the Challenge. Be sure the standard identifying information is at the top left of the Processor `main` circuit, similar to:

```
George Self
Lab 13: Processor
April 5, 2018
```

Save the Processor circuit in a file with this name: *Lab13_Processor*. Complete the code required in the Challenge and store that in a text file with the name *Lab13_Code.txt*. Submit both files for grading.

14

ELEVATOR

14.1 PURPOSE

This final lab is used as a capstone digital logic project.

14.2 CHALLENGE

For this lab, build a circuit that simulates an elevator. This lab does not include step-by-step directions; instead, this document only specifies the requirement and students are on their own to design and build the circuit.

Here are the specifications:

1. The elevator should be in a 3-story building and stop on each floor.
2. There should be a call button on each floor so a guest can request the elevator. When a guest presses the call button, if the elevator is not busy, then it should proceed to the requested floor. If the elevator is busy, it should return to the called floor as soon as it finishes the current trip.
3. The elevator car must have a button for each floor (for this lab, ignore buttons like "Open Door"). When one of the buttons is pressed, the elevator will move to the requested floor. If the elevator is already on the requested floor (for example, some guest on the second floor presses the "Floor 2" button), then the elevator will do nothing.
4. The simulator must have some way to indicate where the elevator is located (its current floor). That could be done with a numeric display (a 7-segment display) or with some sort of light system (an LED on each floor that will light up when the elevator is present). There may be other ways to indicate the elevator's location, so creativity is encouraged.
5. The simulator must have some way to indicate the "door open" and "door close" process. For example, a row of LEDs could light in sequence to show the door opening and a few seconds later closing again.

Figure 14.1 is one student's concept from an earlier class.

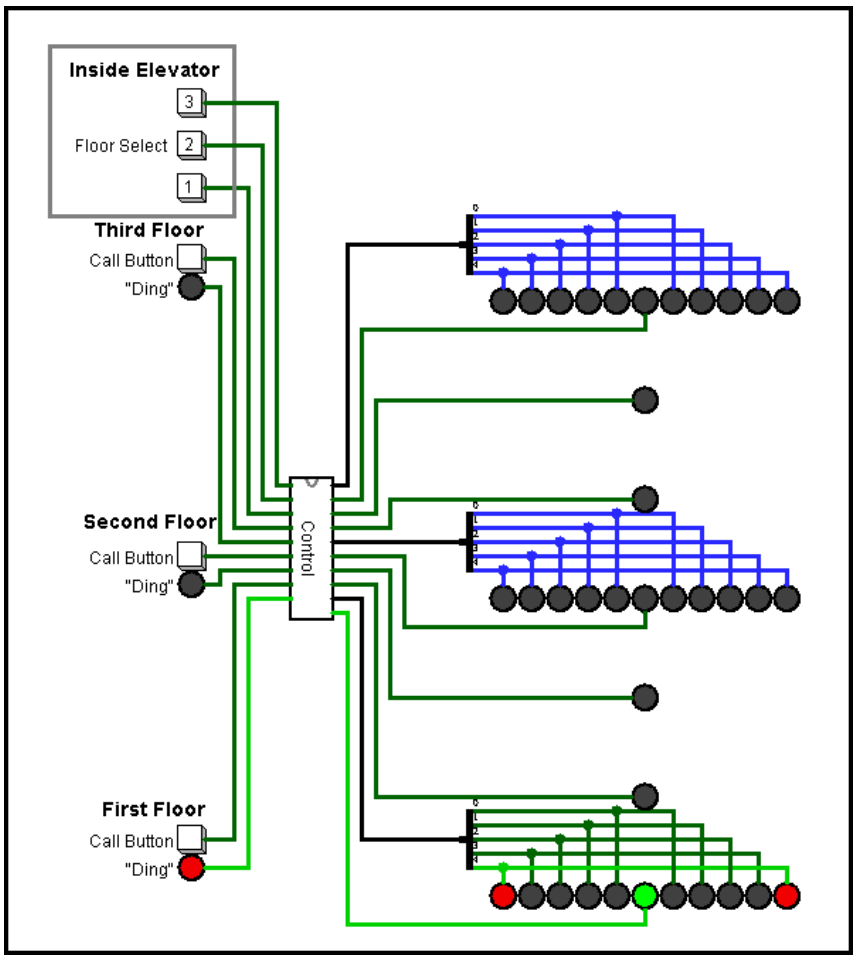

Figure 14.1: Example Elevator Simulator

14.3 DELIVERABLE

To receive a grade for this lab, complete the elevator simulator. Be sure the standard identifying information is at the top left of the `main` circuit:

```
George Self
Lab 14: Elevator
April 30, 2018
```

Save the file with this name: *Lab14_elevator* and submit that file for grading.

Part V

APPENDIX

A

TTL REFERENCE

Logisim-Evolution includes a number of Transistor-Transistor Logic (TTL) ICs. These are pre-packaged digital logic circuits that perform specific, well-defined functions. There are, literally, hundreds of TTL ICs available for purchase from electronics warehouses but *Logisim-Evolution* includes only 35 of the most commonly-used devices. Figure A.1 shows three surface-mounted ICs on a circuit board.

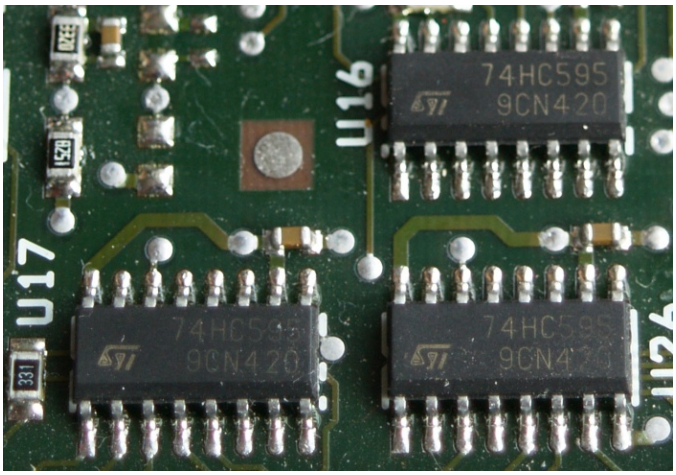

Figure A.1: Three Surface-Mounted Integrated Circuits

A.1 7400: QUAD 2-INPUT NAND GATE

This device contains four independent 2-input NAND gates. Figure A.2 is a logic diagram of one of the four circuits.

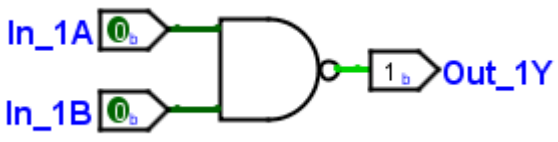

Figure A.2: 7400: Single NAND Gate Circuit

The 7400 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.1.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Output: 3	Out 1Y
Input: 4	In 2A
Input: 5	In 2B
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Output: 11	Out 4Y
Input: 12	In 4A
Input: 13	In 4B

Table A.1: Pinout For 7400

A.2 7402: QUAD 2-INPUT NOR GATE

This device contains four independent 2-input NOR gates. Figure A.3 is a logic diagram of one of the four circuits.

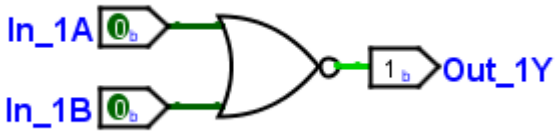

Figure A.3: 7402: Single NOR Gate Circuit

The 7402 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.2.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Output: 3	Out 1Y
Input: 4	In 2A
Input: 5	In 2B
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Output: 11	Out 4Y
Input: 12	In 4A
Input: 13	In 4B

Table A.2: Pinout For 7402

A.3 7404: HEX INVERTER

This device contains six independent inverters. Figure A.4 is a logic diagram of one of the six circuits.

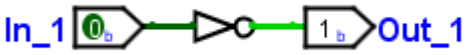

Figure A.4: 7404: Single Inverter Circuit

The 7404 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.3.

Logisim Label	Function
Input: 1	In 1
Output: 2	Out 1
Input: 3	In 2
Output: 4	Out 2
Input: 5	In 3
Output: 6	Out 3
Output: 8	Out 4
Input: 9	In 4
Output: 10	Out 5
Input: 11	In 5
Output: 12	Out 6
Input: 13	In 6

Table A.3: Pinout For 7404

A.4 7408: QUAD 2-INPUT AND GATE

This device contains four independent 2-input AND gates. Figure A.5 is a logic diagram of one of the four circuits.

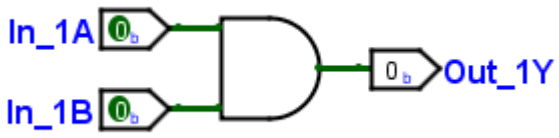

Figure A.5: 7408: Single AND Gate Circuit

The 7408 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.4.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Output: 3	Out 1Y
Input: 4	In 2A
Input: 5	In 2B
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Output: 11	Out 4Y
Input: 12	In 4A
Input: 13	In 4B

Table A.4: Pinout For 7408

A.5 7410: TRIPLE 3-INPUT NAND GATE

This device contains three independent 3-input NAND gates. Figure A.6 is a logic diagram of one of the three circuits.

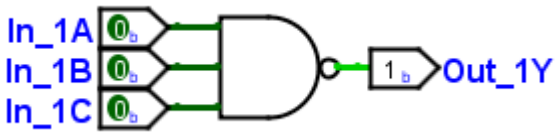

Figure A.6: 7410: Single 3-Input NAND Gate Circuit

The 7410 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.5.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Input: 3	In 2A
Input: 4	In 2B
Input: 5	In 2C
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Input: 11	In 3C
Output: 12	Out 1Y
Input: 13	In 1C

Table A.5: Pinout For 7410

A.6 7411: TRIPLE 3-INPUT AND GATE

This device contains three independent 3-input AND gates. Figure A.7 is a logic diagram of one of the three circuits.

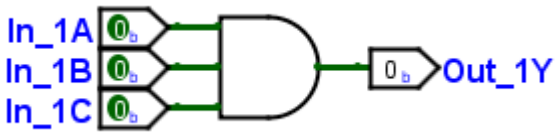

Figure A.7: 7411: Single 3-Input AND Gate Circuit

The 7411 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.6.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Input: 3	In 2A
Input: 4	In 2B
Input: 5	In 2C
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Input: 11	In 3C
Output: 12	Out 1Y
Input: 13	In 1C

Table A.6: Pinout For 7411

A.7 7413: DUAL 4-INPUT NAND GATE (SCHMITT-TRIGGER)

This device contains two independent 4-input NAND gates. Schmitt-triggers are a special type of device that are used to filter out spurious noise on a circuit. They are designed to change from low-to-high or high-to-low only when the input voltage reaches a preset level but not if the voltage randomly fluctuates without crossing the set-points. This device is essentially the same as the 7418. Figure A.8 is a logic diagram of one of the two circuits.

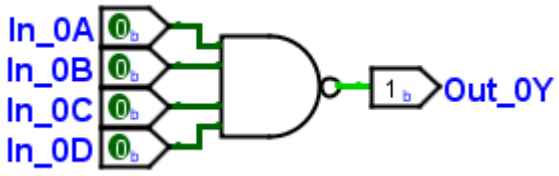

Figure A.8: 7413: Single 4-Input NAND Gate Circuit

The 7413 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.7.

Logisim Label	Function
Input: 1	In A0
Input: 2	In B0
Pin 3: NC	Not Connected
Input: 4	In C0
Input: 5	In D0
Output: 6	Out Y0
Output: 8	Out Y1
Input: 9	In D1
Input: 10	In C1
Pin 11: NC	Not Connected
Input: 12	In B1
Input: 13	In A1

Table A.7: Pinout For 7413

A.8 7414: HEX INVERTER (SCHMITT-TRIGGER)

This device contains six independent inverters. Schmitt-triggers are a special type of device that are used to filter out spurious noise on a circuit. They are designed to change from low-to-high or high-to-low only when the input voltage reaches a preset level but not if the voltage randomly fluctuates without crossing the set-points. This device is essentially the same as the 7419. Figure A.9 is a logic diagram of one of the six circuits.

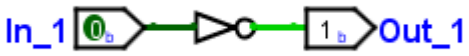

Figure A.9: 7414: Single Inverter Circuit

The 7414 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.8.

Logisim Label	Function
Input: 1	In 1
Output: 2	Out 1
Input: 3	In 2
Output: 4	Out 2
Input: 5	In 3
Output: 6	Out 3
Output: 8	Out 4
Input: 9	In 4
Output: 10	Out 5
Input: 11	In 5
Output: 12	Out 6
Input: 13	In 6

Table A.8: Pinout For 7414

A.9 7418: DUAL 4-INPUT NAND GATE (SCHMITT-TRIGGER INPUTS)

This device contains two independent 4-input NAND gates. Schmitt-triggers are a special type of device that are used to filter out spurious noise on a circuit. They are designed to change from low-to-high or high-to-low only when the input voltage reaches a preset level but not if the voltage randomly fluctuates without crossing the set-points. This device is essentially the same as the 7413. Figure A.10 is a logic diagram of one of the two circuits.

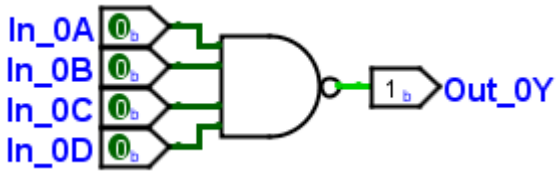

Figure A.10: 7418: Single 4-Input NAND Gate Circuit

The 7418 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.9.

Logisim Label	Function
Input: 1	In A0
Input: 2	In B0
Pin 3 NC	Not Connected
Input: 4	In C0
Input: 5	In D0
Output: 6	Out Y0
Output: 8	Out Y1
Input: 9	In D1
Input: 10	In C1
Pin 11 NC	Not Connected
Input: 12	In B1
Input: 13	In A1

Table A.9: Pinout For 7418

A.10 7419: HEX INVERTER (SCHMITT-TRIGGER)

This device contains six independent inverters. Schmitt-triggers are a special type of device that are used to filter out spurious noise on a circuit. They are designed to change from low-to-high or high-to-low only when the input voltage reaches a preset level but not if the voltage randomly fluctuates without crossing the set-points. This device is essentially the same as the 7414. Figure A.11 is a logic diagram of one of the six circuits.

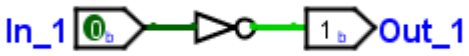

Figure A.11: 7419: Single Inverter Circuit

The 7419 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.10.

Logisim Label	Function
Input: 1	In 1
Output: 2	Out 1
Input: 3	In 2
Output: 4	Out 2
Input: 5	In 3
Output: 6	Out 3
Output: 8	Out 4
Input: 9	In 4
Output: 10	Out 5
Input: 11	In 5
Output: 12	Out 6
Input: 13	In 6

Table A.10: Pinout For 7419

A.11 7420: DUAL 4-INPUT NAND GATE

This device contains two independent 4-input NAND gates. Figure A.12 is a logic diagram of one of the two circuits.

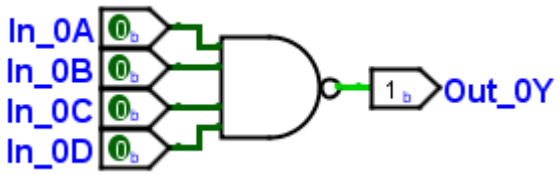

Figure A.12: 7420: Single 4-Input NAND Gate Circuit

The 7420 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.11.

Logisim Label	Function
Input: 1	In A0
Input: 2	In B0
Pin 3 NC	Not Connected
Input: 4	In C0
Input: 5	In D0
Output: 6	Out Y0
Output: 8	Out Y1
Input: 9	In D1
Input: 10	In C1
Pin 11 NC	Not Connected
Input: 12	In B1
Input: 13	In A1

Table A.11: Pinout For 7420

A.12 7421: DUAL 4-INPUT AND GATE

This device contains two independent 4-input AND gates. Figure A.13 is a logic diagram of one of the two circuits.

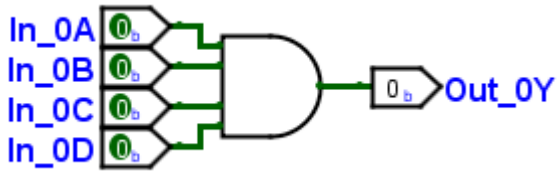

Figure A.13: 7421: Single 4-Input AND Gate Circuit

The 7421 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.12.

Logisim Label	Function
Input: 1	In A0
Input: 2	In B0
Pin 3 NC	Not Connected
Input: 4	In C0
Input: 5	In D0
Output: 6	Out Y0
Output: 8	Out Y1
Input: 9	In D1
Input: 10	In C1
Pin 11 NC	Not Connected
Input: 12	In B1
Input: 13	In A1

Table A.12: Pinout For 7421

A.13 7424: QUAD 2-INPUT NAND GATE (SCHMITT-TRIGGER)

This device contains four independent 2-input NAND gates. Schmitt-triggers are a special type of device that are used to filter out spurious noise on a circuit. They are designed to change from low-to-high or high-to-low only when the input voltage reaches a preset level but not if the voltage randomly fluctuates without crossing the set-points. This device is essentially the same as the 7400. Figure A.14 is a logic diagram of one of the four circuits.

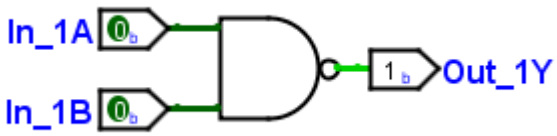

Figure A.14: 7424: Single NAND Gate Circuit

The 7424 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.13.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Output: 3	Out 1Y
Input: 4	In 2A
Input: 5	In 2B
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Output: 11	Out 4Y
Input: 12	In 4A
Input: 13	In 4B

Table A.13: Pinout For 7424

A.14 7427: TRIPLE 3-INPUT NOR GATE

This device contains three independent 3-input NOR gates. Figure A.15 is a logic diagram of one of the three circuits.

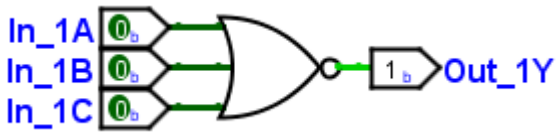

Figure A.15: 7411: Single 3-Input NOR Gate Circuit

The 7427 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.14.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Input: 3	In 2A
Input: 4	In 2B
Input: 5	In 2C
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Input: 11	In 3C
Output: 12	Out 1Y
Input: 13	In 1C

Table A.14: Pinout For 7427

A.15 7430: SINGLE 8-INPUT NAND GATE

This device contains a single 8-input NAND gate. The logic for this gate is $Y = \overline{A \cdot B \cdot C \cdot D \cdot E \cdot F \cdot G \cdot H}$. Figure A.16 is a logic diagram of the circuit.

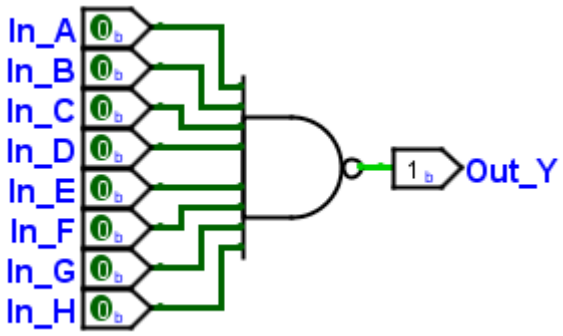

Figure A.16: 7430: Single 8-Input NAND Gate

The 7430 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.15.

Logisim Label	Function
Input: 1	In A
Input: 2	In B
Input: 3	In C
Input: 4	In D
Input: 5	In E
Input: 6	In F
Output: 8	Out Y
Pin 9: NC	Not Connected
Pin 10: NC	Not Connected
Input: 11	In G
Input: 12	In H
Pin 13: NC	Not Connected

Table A.15: Pinout For 7430

A.16 7432: QUAD 2-INPUT OR GATE

This device contains four independent 2-input OR gates. Figure A.17 is a logic diagram of one of the four circuits.

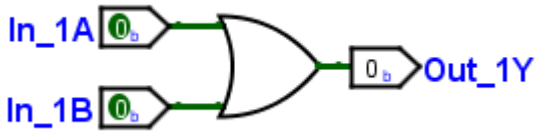

Figure A.17: 7432: Single OR Gate Circuit

The 7432 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.16.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Output: 3	Out 1Y
Input: 4	In 2A
Input: 5	In 2B
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Output: 11	Out 4Y
Input: 12	In 4A
Input: 13	In 4B

Table A.16: Pinout For 7432

A.17 7436: QUAD 2-INPUT NOR GATE

This device contains four independent 2-input NOR gates. This device is essentially the same as the 7402. Figure A.18 is a logic diagram of one of the four circuits.

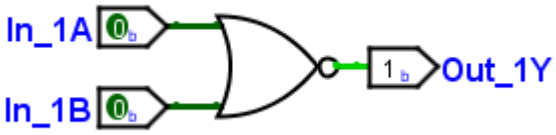

Figure A.18: 7436: Single NOR Gate Circuit

The 7436 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.17.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Output: 3	Out 1Y
Input: 4	In 2A
Input: 5	In 2B
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Output: 11	Out 4Y
Input: 12	In 4A
Input: 13	In 4B

Table A.17: Pinout For 7436

A.18 7442: BCD TO DECIMAL DECODER

This device takes a BDC input and deactivates a single line corresponding to the input number. It is often called a "One-Of-Ten" decoder. As an example, if 0111_{BCD} is input then line 7-of-10 will go low while all other outputs will remain high. Figure A.19 illustrates a 7442 IC in a very simple circuit.

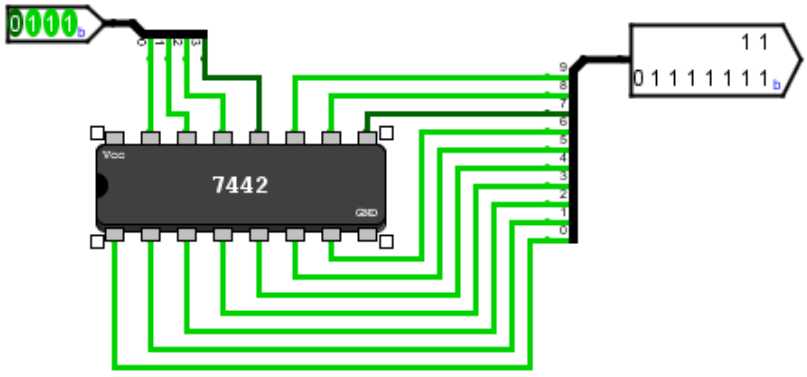

Figure A.19: 7442: BCD to Decimal Decoder

Table A.18 is the truth table for this device. Any BCD input greater than 1001 is ignored and all outputs will be high for those inputs.

Inputs				Output									
A	B	C	D	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0

Table A.18: Truth Table For The 7442 Circuit

The 7442 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.19.

Logisim Label	Function
Output 1: O0	Out 0
Output 2: O1	Out 1
Output 3: O2	Out 2
Output 4: O3	Out 3
Output 5: O4	Out 4
Output 6: O5	Out 5
Output 7: O6	Out 6
Output 8: O7	Out 7
Output 10: O8	Out 8
Output 11: O9	Out 9
Input 12: D	In D
Input 13: C	In C
Input 14: B	In B
Input 15: A	In A

Table A.19: Pinout For 7442

A.19 7443: EXCESS-3 TO DECIMAL DECODER

This device takes an Excess-3 input and deactivates a single line corresponding to the input number. It is often called a "One-Of-Ten"

decoder. As an example, if 0011_{Ex3} is input then line 0-of-10 will go low while all other outputs will remain high. This is wired in exactly the same way as the 7442 IC illustrated in Figure A.19.

Table A.20 is the truth table for this device. Any input numbers other than those found in the truth table are ignored and all outputs will be high for those inputs.

Inputs				Output									
A	B	C	D	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	1	1	1	1	1	1	1	1
0	1	0	0	1	0	1	1	1	1	1	1	1	1
0	1	0	1	1	1	0	1	1	1	1	1	1	1
0	1	1	0	1	1	1	0	1	1	1	1	1	1
0	1	1	1	1	1	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1	1	1	1	0	1	1
1	0	1	1	1	1	1	1	1	1	1	1	0	1
1	1	0	0	1	1	1	1	1	1	1	1	1	0

Table A.20: Truth Table For The 7443 Circuit

The 7443 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.21.

Logisim Label	Function
Output 1: O0	Out 0
Output 2: O1	Out 1
Output 3: O2	Out 2
Output 4: O3	Out 3
Output 5: O4	Out 4
Output 6: O5	Out 5
Output 7: O6	Out 6
Output 8: O7	Out 7
Output 10: O8	Out 8
Output 11: O9	Out 9
Input 12: D	In D
Input 13: C	In C
Input 14: B	In B
Input 15: A	In A

Table A.21: Pinout For 7443

A.20 7444: GRAY TO DECIMAL DECODER

This device takes a Gray Excess Code, which is a combination of Gray and Excess-3 Codes, input and deactivates a single line corresponding to the input number. It is often called a "One-Of-Ten" decoder. As an example, if 1100_{GrayEx3} is input then line 5-of-10 will go low while all other outputs will remain high. This is wired in exactly the same way as the 7442 IC illustrated in Figure A.19.

Table A.22 is the truth table for this device. Any input numbers other than those found in the truth table are ignored and all outputs will be high for those inputs.

Inputs				Output									
A	B	C	D	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	1	1	1	1	1	1	1	1
0	1	1	0	1	0	1	1	1	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1	1	1	1
0	1	0	1	1	1	1	0	1	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1	1	1
1	1	0	0	1	1	1	1	1	0	1	1	1	1
1	1	0	1	1	1	1	1	1	1	0	1	1	1
1	1	1	1	1	1	1	1	1	1	1	0	1	1
1	1	1	0	1	1	1	1	1	1	1	1	0	1
1	0	1	0	1	1	1	1	1	1	1	1	1	0

Table A.22: Truth Table For The 7444 Circuit

The 7443 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.23.

Logisim Label	Function
Output 1: O0	Out 0
Output 2: O1	Out 1
Output 3: O2	Out 2
Output 4: O3	Out 3
Output 5: O4	Out 4
Output 6: O5	Out 5
Output 7: O6	Out 6
Output 8: O7	Out 7
Output 10: O8	Out 8
Output 11: O9	Out 9
Input 12: D	In D
Input 13: C	In C
Input 14: B	In B
Input 15: A	In A

Table A.23: Pinout For 7444

A.21 7447: BCD TO 7-SEGMENT DECODER

This device takes a BCD Code input and activates a combination of outputs such that a 7-segment display will correctly indicate the input number. Figure A.20 illustrates a 7447 IC in a very simple circuit.

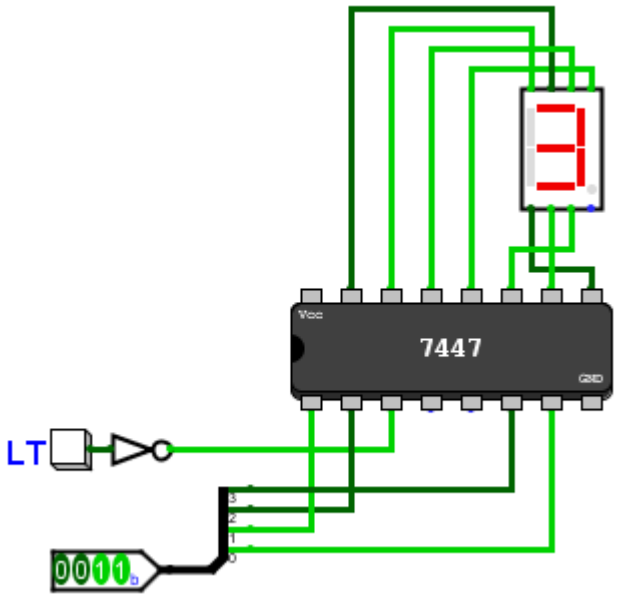

Figure A.20: 7447: BCD to 7-Segment Decoder

Table A.24 is the truth table for this device.

Inputs				Output						
A	**B**	**C**	**D**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1
1	1	0	0	1	0	0	1	1	1	0
1	1	0	1	0	1	1	1	1	0	1
1	1	1	0	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	1	1	1

Table A.24: Truth Table For The 7447 Circuit

The 7447 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.25.

Logisim Label	Function
Input 1: B	B
Input 2: C	C
Input 3: LT	LT
Input 4: BI	BI
Input 5: RBI	RBI
Input 6: D	D
Input 7: A	A
Output 8: e	e
Output 10: d	d
Output 11: c	c
Output 12: b	b
Output 13: a	a
Output 14: g	g
Output 15: f	f

Table A.25: Pinout For 7447

A.22 7451: DUAL AND-OR-INVERT GATE

This device contains two independent AND-OR-INVERT gates. Figure A.21 is a logic diagram of one of the two circuits.

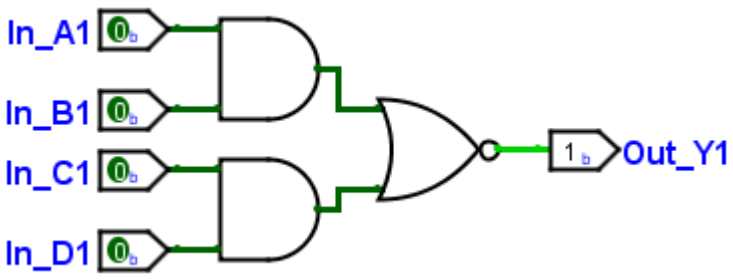

Figure A.21: 7451: Single AND-OR-INVERT Gate Circuit

The 7451 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.26.

Logisim Label	Function
Input 1: A1	In A1
Input 2: A2	In A2
Input 3: B2	In B2
Input 4: C2	In C2
Input 5: D2	In D2
Output 6: Y2	Out Y2
Output 8: Y1	Out Y1
Input 9: C1	In C1
Input 10: D1	In D1
Pin 11: NC	Not Connected
Pin 12: NC	Not Connected
Input 13: B1	In B1

Table A.26: Pinout For 7451

A.23 7454: FOUR WIDE AND-OR-INVERT GATE

This device contains a single four-wide AND-OR-INVERT gate. Figure A.22 is a logic diagram of the circuit.

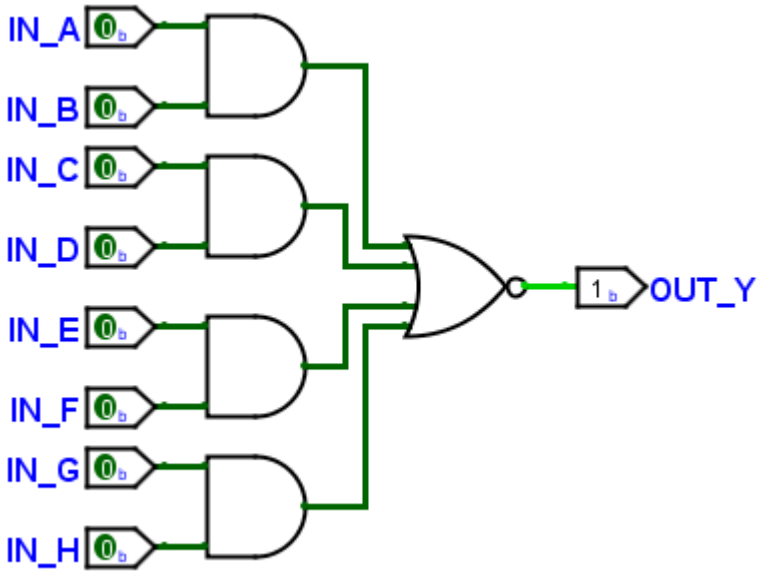

Figure A.22: 7454: Four Wide AND-OR-INVERT Gate Circuit

The 7454 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.27.

Logisim Label	Function
Input 1: A	In A
Input 2: C	In C
Input 3: D	In D
Input 4: E	In E
Input 5: F	In F
Pin 6: NC	Not Connected
Output 8: Y	Out Y
Input 9: G	In G
Input 10: H	In H
Pin 11: NC	Not Connected
Pin 12: NC	Not Connected
Input 13: B	In B

Table A.27: Pinout For 7454

A.24 7458: DUAL AND-OR GATE

This device contains a two AND-OR gates. One has three-input AND gates and the other has two-input AND gates. Figure A.23 is a logic diagram of the circuit.

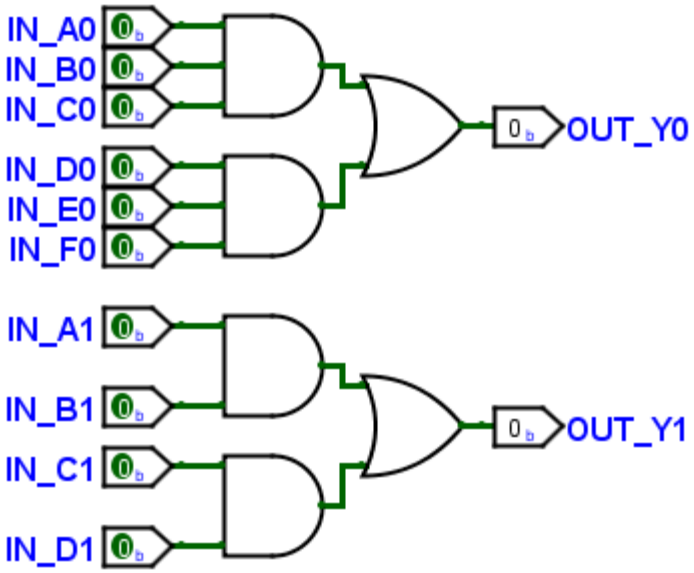

Figure A.23: 7458: Dual AND-OR Gate Circuit

The 7458 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.28.

Logisim Label	Function
Input 1: A0	In A0
Input 2: A1	In A1
Input 3: B1	In B1
Input 4: C1	In C1
Input 5: D1	In D1
Output 6: Y1	Out Y1
Output 8: Y0	Out Y0
Input 9: D0	In D0
Input 10: E0	In E0
Input 11: F0	In F0
Input 12: B0	In B0
Input 13: C0	In C0

Table A.28: Pinout For 7458

A.25 7464: 4-2-3-2 AND-OR-INVERT GATE

This device contains four AND gates of different input sizes that feed a NOR gate. Figure A.24 is a logic diagram of the circuit.

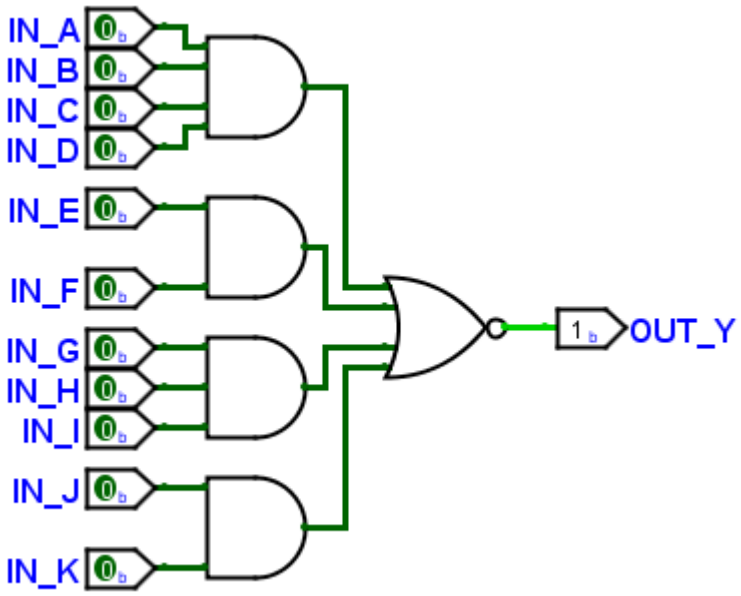

Figure A.24: 7464: 4-2-3-2 AND-OR-INVERT Gate Circuit

The 7464 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.29.

Logisim Label	Function
Input 1: A	In A
Input 2: E	In E
Input 3: F	In F
Input 4: G	In G
Input 5: H	In H
Input 6: I	In I
Output 8: Y	Out Y
Input 9: J	In J
Input 10: K	In K
Input 11: B	In B
Input 12: C	In C
Input 13: D	In D

Table A.29: Pinout For 7464

A.26 7474: DUAL D-FLIPFLOPS WITH PRESET AND CLEAR

This device contains two D-Flipflops, each with its own preset and clear. The 7474 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.30.

Logisim Label	Function
Input 1: nCLR1	On low, clear FF1
Input 2: D1	FF1 data input
Input 3: CLK1	FF1 clock
Input 4: nPRE1	On low, set FF1
Output 5: Q1	FF1 Q-out
Output 6: nQ1	FF1 Q-not-out
Output 8: nQ2	FF2 Q-not-out
Output 9: Q2	FF2 Q-out
Input 10: nPRE2	On low, set FF2
Input 11: CLK2	FF2 clock
Input 12: D2	FF2 data input
Input 13: nCLR2	On low, clear FF2

Table A.30: Pinout For 7474

A.27 7485: 4-BIT MAGNITUDE COMPARATOR

This device compares two 4-bit numbers and outputs one of three values: $A > B$, $A = B$, or $A < B$. It is also designed to be cascaded by including an input port for each of the three values. The 7485 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.31.

Logisim Label	Function
Input 1: B3	Bit B3
Input 2: A<B	Value from prior stage
Input 3: A=B	Value from prior stage
Input 4: A>B	Value from prior stage
Output 5: A>B	High if A>B
Output 6: A=B	High if A=B
Output 7: A<B	High if A<B
Input 9: B0	Bit B0
Input 10: A0	Bit A0
Input 11: B1	Bit B1
Input 12: A1	Bit A1
Input 13: A2	Bit A2
Input 14: B2	Bit B2
Input 15: A3	Bit A3

Table A.31: Pinout For 7485

A.28 7486: QUAD 2-INPUT XOR GATE

This device contains four independent 2-input XOR gates. Figure A.25 is a logic diagram of one of the four circuits.

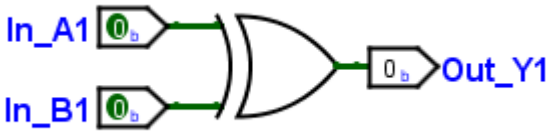

Figure A.25: 7486: Single XOR Gate Circuit

The 7486 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.32.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Output: 3	Out 1Y
Input: 4	In 2A
Input: 5	In 2B
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Output: 11	Out 4Y
Input: 12	In 4A
Input: 13	In 4B

Table A.32: Pinout For 7486

A.29 74125: QUAD BUS BUFFER, 3-STATE GATE

This device contains four independent buffers. When each is enabled with a low on the enable line then the input is passed to the output, when not enabled then the output floats. Figure A.26 is a logic diagram of one of the four circuits.

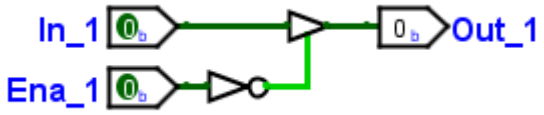

Figure A.26: 74125: Single Buffer Circuit

The 74125 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.33.

Logisim Label	Function
Input: 1	nEna 1
Input: 2	In 1
Output: 3	Out 1
Input: 4	nEna 2
Input: 5	In 2
Output: 6	Out 2
Output: 8	Out 3
Input: 9	In 3
Input: 10	nEna 3
Output: 11	Out 4
Input: 12	In 4
Input: 13	nEna 4

Table A.33: Pinout For 74125

A.30 74165: 8-BIT PARALLEL-TO-SERIAL SHIFT REGISTER

This device can accept data in either parallel or serial form and shift it out in serial form. The 74165 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.34.

Logisim Label	Function
Input 1: Shift/Load	Load when low, shift when high
Input 2: Clock	Clock
Input 3: P4	Input bit 4
Input 4: P5	Input bit 5
Input 5: P6	Input bit 6
Input 6: P7	Input bit 7
Output 7: Q7n	Complement of serial out
Output 9: Q7	Serial out
Input 10: Serial Input	Serial data in
Input 11: P0	Input bit 0
Input 12: P1	Input bit 1
Input 13: P2	Input bit 2
Input 14: P3	Input bit 3
Input 15: Clock Inhibit	Clock inhibit

Table A.34: Pinout For 74165

A.31 74175: QUAD D-FLIPFLOPS WITH SYNC RESET

This device contains four D-Flipflops with a single clock and master reset. The 74175 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.35.

Logisim Label	Function
Input 1: nCLR	On low, clear all FF
Output 2: Q1	FF1 Q-out
Output 3: nQ1	FF1 Q-not-out
Input 4: D1	FF1 data input
Input 5: D2	FF2 data input
Output 6: nQ2	FF2 Q-not-out
Output 7: Q2	FF2 Q-out
Input 9: CLK	Clock for all FF
Output 10: Q3	FF3 Q-out
Output 11: nQ3	FF3 Q-not-out
Input 12: D3	FF3 data input
Input 13: D4	FF4 data input
Output 14: nQ4	FF4 Q-not-out
Output 15: Q4	FF4 Q-out

Table A.35: Pinout For 74175

A.32 74266: QUAD 2-INPUT XNOR GATE

This device contains four independent 2-input XNOR gates. Figure A.27 is a logic diagram of one of the four circuits.

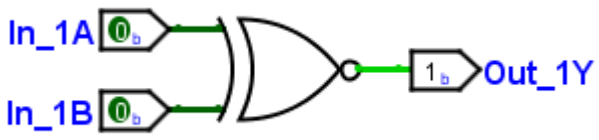

Figure A.27: 74266: Single XNOR Gate Circuit

The 74266 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.36.

Logisim Label	Function
Input: 1	In 1A
Input: 2	In 1B
Output: 3	Out 1Y
Input: 4	In 2A
Input: 5	In 2B
Output: 6	Out 2Y
Output: 8	Out 3Y
Input: 9	In 3A
Input: 10	In 3B
Output: 11	Out 4Y
Input: 12	In 4A
Input: 13	In 4B

Table A.36: Pinout For 74266

A.33 74273: OCTAL D-FLIPFLOP WITH CLEAR

This device contains a single 8-bit D-Flipflop with a single clock and master clear. The 74273 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.37.

Logisim Label	Function
Input 1: nCLR	On low, clear the FF
Output 2: Q1	data bit 1 output
Input 3: D1	data bit 1 input
Input 4: D2	data bit 2 input
Output 5: Q2	data bit 2 output
Output 6: Q3	data bit 3 output
Input 7: D3	data bit 3 input
Input 8: D4	data bit 4 input
Output 9: Q4	data bit 4 output
Input 11: CLK	Clock
Output 12: Q5	data bit 5 output
Input 13: D5	data bit 5 input
Input 14: D6	data bit 6 input
Output 15: Q6	data bit 6 output
Output 16: Q7	data bit 7 output
Input 17: D7	data bit 7 input
Input 18: D8	data bit 8 input
Output 19: Q8	data bit 8 output

Table A.37: Pinout For 74273

A.34 74283: 4-BIT BINARY FULL ADDER

This device contains a 4-bit adder with carry-in and carry-out bits. The 74283 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.38.

Logisim Label	Function
Output 1: ∑2	Sum, bit 2
Input 2: B2	Operand B, bit 2
Input 3: A2	Operand A, bit 2
Output 4: ∑1	Sum, bit 1
Input 5: A1	Operand A, bit 1
Input 6: B1	Operand B, bit 1
Input 7: CIN	Carry in bit
Output 9: C4	Carry out bit
Output 10: ∑4	Sum, bit 4
Input 11: B4	Operand B, bit 4
Input 12: A4	Operand A, bit 4
Output 13: ∑3	Sum, bit 3
Input 14: A3	Operand A, bit 3
Input 15: B3	Operand B, bit 3

Table A.38: Pinout For 74283

A.35 74377: OCTAL D-FLIPFLOP WITH ENABLE

This device contains a single 8-bit D-Flipflop with a single clock and enable. The 74377 device in *Logisim-Evolution* uses the wiring connections indicated in Table A.39.

Logisim Label	Function
Input 1: nCLKen	On low, enable the clock
Output 2: Q1	data bit 1 output
Input 3: D1	data bit 1 input
Input 4: D2	data bit 2 input
Output 5: Q2	data bit 2 output
Output 6: Q3	data bit 3 output
Input 7: D3	data bit 3 input
Input 8: D4	data bit 4 input
Output 9: Q4	data bit 4 output
Input 11: CLK	Clock
Output 12: Q5	data bit 5 output
Input 13: D5	data bit 5 input
Input 14: D6	data bit 6 input
Output 15: Q6	data bit 6 output
Output 16: Q7	data bit 7 output
Input 17: D7	data bit 7 input
Input 18: D8	data bit 8 input
Output 19: Q8	data bit 8 output

Table A.39: Pinout For 74377

COLOPHON

This book was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of September 29, 2019 (`classicthesis` Edition 4.0).

Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL*) are used. The "typewriter" text is typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)