Lab Manual: Exploring Digital Logic with Logisim-Evolution

George Self

Creative Commons License This work is licensed under the Creative Commons Attribution 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by/3.0/.

First Edition: Summer, 2013  
Second Edition: Summer, 2014  
Third Edition: Summer, 2015

George Self  
Cochise College  
901 N. Colombo Ave  
Sierra Vista, AZ 85650

selfg@cochise.edu

# Table of Contents

Table of Contents 3

Foreword 5

Introduction to the Study of Digital Logic 5

Introduction to the Author 5

Introduction to This Book 5

Introduction to Logisim-Evolution 5

About the Creative Commons License 6

Lab 01 – Introduction to Logisim-Evolution 7

Lab 02 – Logisim-Evolution Subcircuits 13

Lab 03 – BCD Adder 19

Lab 04 – 8-Bit Adder 23

Lab 05 – 8-Bit Subtractor 25

Lab 06 – Magic 8-Ball 29

Lab 07 – Hamming Parity Check 31

Lab 08 – Timer 35

Lab 09 – RAM Lab 37

Lab 10 – ROM Lab 39

# 

# Foreword

## Introduction to the Study of Digital Logic

Digital logic is the study of how electronic devices make decisions. It functions at the lowest level of computer operations: bits that can either be "on" or "off" and groups of bits that form "bytes" and "words" that control physical devices. The language of digital logic is Boolean algebra, which is a mathematical model used to describe the logical function of a circuit; and that model can then be used to design the most efficient device possible. Finally, various devices, such as adders and registers, can be combined into increasingly complex circuits designed to accomplish advanced decision-making tasks.

## Introduction to the Author

I have worked with computers and computer controlled systems for more than 30 years. I took my first programming class in 1976; and, several years later, was introduced to digital logic while taking classes to learn how to repair computer systems. For many years, my profession was to work on computer systems, both as a repair technician and a programmer, where I used the principles of digital logic daily. I then began teaching digital logic classes at Cochise College and was able to share my enthusiasm for the subject with Computer Information Systems students. Over the years, I have continued my studies of digital logic in order to improve my understanding of the topic; I also enjoy building logic circuits on a simulator to solve interesting challenges. It is my goal to make digital logic understandable and to also ignite a lifelong passion for the subject in students.

## Introduction to This Book

First Edition published August, 2013. Second Edition published August, 2014. Third Edition published August, 2015.

The book is the lab manual to accompany Exploring Digital Logic with Logisim-Evolution. It includes a number of lab exercises designed for the Logisim-Evolution simulator, which is an excellent logic simulator that is available free of charge.

Disclaimer: I wrote, edited, illustrated, and published this book myself. While I did the best that I could, there are, no doubt, errors. I apologize in advance if anything presented here is factually wrong; I’ll correct that in future editions. I’ll also correct whatever typos I overlooked, despite Word’s red squiggly lines trying to tell me to double check my work.

## Introduction to Logisim-Evolution

Logisim-Evolution is a logic simulator that is used to put into practice the theories of mathematics and logic presented in this book, making those lessons easier to comprehend. Logisim-Evolution is a Java application that is available as a free download (as described in the first lab exercise). The simulator is easy to use and includes enough logic devices (like adders and registers) to cover all of the aspects of gate-level digital logic that are presented in this book.

## About the Creative Commons License

This book is being released under the Creative Commons Attribution license. This permits other people to share, remix, or even use the work commercially as long as they attribute my original contribution. Like most folks who use a Creative Commons license, I believe that information wants to be freed, and I’ll do whatever I can to aid in that process.

Creative Commons License This work is licensed under the Creative Commons Attribution 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by/3.0/.

# Lab 01 – Introduction to Logisim-Evolution

### Purpose

This lab introduces the Logisim-Evolution Logic Simulator, which is used for all gate-level lab exercises.

### Installing

Logisim-Evolution (https://github.com/reds-heig/logisim-evolution) is available as a free download. Find the download link at the end of this sentence: “You can find an already compiled stable version of the code here” in the *How to install logisim-evolution* section about halfway down the page.

Logisim-Evolution is a Java application, so Java will need to be installed before using Logisim-Evolution. Double-clicking the Logisim-Evolution file will automatically start Java and then the simulator. The program will not need to be uninstalled since it is not actually installed; the Logisim-Evolution file can simply be deleted when it is no longer needed.

### Beginner's Tutorial

Logisim-Evolution comes with a beginner’s tutorial available in the Help files. That tutorial only takes a few minutes and introduces users to the major components of Logisim-Evolution.

### Logisim-Evolution Workspace

Start Logisim-Evolution by double-clicking its icon. The initial Logisim-Evolution window will be similar to this:

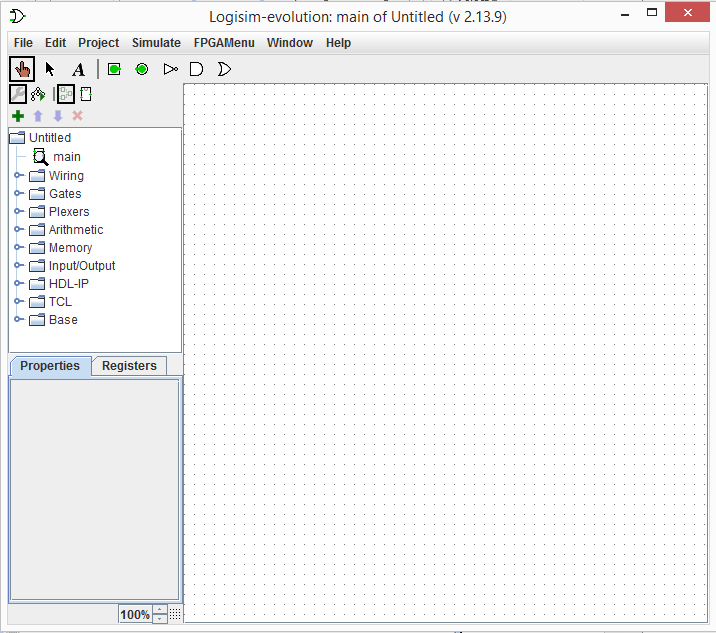


Figure 1: Logisim-Evolution Workspace

The Logisim-Evolution space is divided into several areas. Along the top is a text menu that includes the types of selections expected in any program. For example, the "File" menu includes items like "Save" and "Exit." In later labs, the various options under "Project" and "Simulate" will be described. Of particular importance at this point is a Help menu item named "Library Reference" that contains information about every logical device available in Logisim-Evolution and is very useful while wiring components in new circuits.

Under the menu 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.
* **Square Port**: Creates an input port for a circuit.
* **Round 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 with four icons along its top edge. 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 when they are needed.

The Properties panel on the lower left side of the screen is where the properties for any selected component can be changed. For example, the number of inputs for an AND gate can be specified.

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

### Simple Multiplexor

A multiplexor selects a single output from two or more inputs. For this lab, a simple one-bit multiplexor will be built.

Start by clicking the "AND" button on the toolbar and placing two AND gates on the canvas:

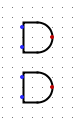


Figure 2: 2 AND Gates

Select one AND gate and observe the various properties available for that gate. The default values do not need to be changed for this circuit; however, all illustrations 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.

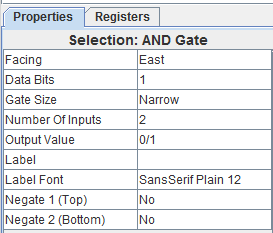


Figure 3: AND Gate Attributes

The two AND gates need to be combined with an OR gate. Add an OR gate as illustrated below:

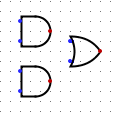


Figure 4: OR Gate Added

The top input for the first AND gate needs two NOT gates (inverters) to change the controlling signal:

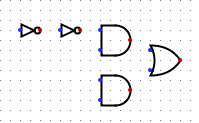


Figure 5: 2 NOT Gates Added

All inputs and outputs need to be added as illustrated below. Note: inputs are square and outputs are round. The "Label" attribute for each input and output should be specified as in the illustration. Note: output pins show an "x" until they are actually wired to some device like the OR gate in the illustration.

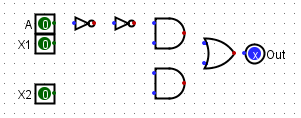


Figure 6: Inputs and Outpus Added

Finally, connect each device with a wire by clicking on the various ports and stretching a wire to the next port.

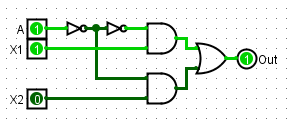


Figure 7: Inputs and Outpus Added

The above illustration also shows the circuit in operation. By clicking the "Pointing Finger" and "poking" the various inputs the circuit function can be tested. If it is working properly, when the "A" (for "Address") input is high then the value of X1 should be transmitted to the output; but when "A" is low then the value of X2 should be transmitted to the output.

### 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 - Introduction to Logisim-Evolution

July 15, 2015

Note that Logisim-Evolution will automatically center text in a box, so all 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 this:

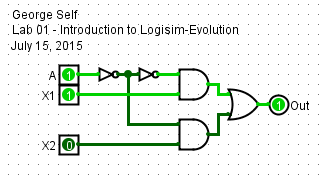


Figure 8: Completed Lab 1

### Cleanup

Be sure the standard identifying information is at the top left of the circuit and then save the file with this name: Lab 01 - Intro.

# Lab 02 – Logisim-Evolution Subcircuits

### Purpose

In this lab a main circuit and one subcircuit are linked in order to further explore the Logisim-Evolution simulator.

### Procedure

#### Main Circuit

Place a Button (found in the *Input/Output* library) on a blank canvas. While that button is highlighted (it will have a drag handle on each of its four corners), change its "Label" attribute to "In\_1" since it will be used to input a "1" into the circuit (like the number 1 on a calculator). Then change the location of the label to “West.”

Next, add four more buttons and label them In\_2 – In\_5:

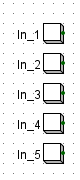


Figure 9: 5 Buttons

A button can be tested by adding an LED (in the *Input/Output* library) to the circuit just to the right of a button and connecting the button directly to the LED. Then, whenever the button is clicked with the poke tool (the "pointing finger") the LED should light. When finished testing the button's function, delete the LED and wire.

#### SubCircuit: CombineInputs

Logisim-Evolution permits designers to work with a main circuit and any number of subcircuits. Students who have studied programming languages are familiar with "subroutines" or "classes" that can be designed and built one time and then reused many times whenever they are needed. Logisim-Evolution permits that same type of modular design by using subcircuits.

To create the *CombineInputs* subcircuit, click *Project -> Add Circuit* in the menubar. Name the new circuit *CombineInputs.* Notice that now two circuits are available (look at the list of circuits in Explorer Pane near the top left part of the screen): *main* and *CombineInputs.* To open either circuit, double-click its name in the Explorer Pane. The active circuit will have a small magnifying glass over its icon, like *main* in the illustration below.

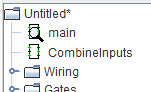


Figure 10: Explorer Pane

To create *CombineInputs,* start by double-clicking that circuit in the Explorer Pane. The screen should change to a blank canvas since nothing has been added to that circuit.

Place five inputs (that is the square tool in the toolbar). Using the attributes panel for each input, label them *In\_1,* *In\_2,* *In\_3,* *In\_4,* and *In\_5* (for "Input Pin 1" and so forth). Note: exact placement for these pins is not critical since they can be moved later if they are not in the optimum spot.

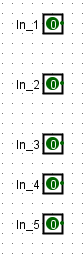


Figure 11: Five Inputs

Place 4 OR gates in the circuit to the right of the 5 input pins. In the diagram below, notice that the top OR gate has 5 inputs, the second OR gate has 4 inputs, the third OR gate has 3 inputs, and the fourth OR gate has 2 inputs. The *Number of Inputs* for each gate can be set in the attributes panel for that gate. The exact placement of these inputs and OR gates is not important since they can be easily moved later.

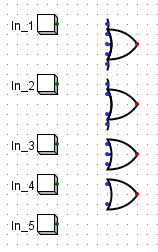


Figure 12: OR Gates Added

Wire the inputs to the various OR gates using the following illustration as a guide.

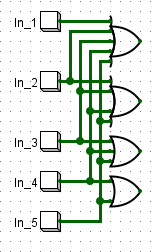


Figure 13: OR Gates Wired

Place 5 outputs (that is the round tool in the toolbar) and wire them to the circuit as shown in the following illustration. The outputs should be named *Out1,* *Out2,* *Out3,* *Out4,* and *Out5.*

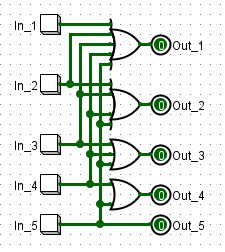


Figure 14: Completed Subcircuit

#### Finish the Main Circuit

Make the main circuit active by double-clicking its name in the Explorer Panel.

You can insert the subcircuit you just created into the main circuit by clicking one time on the subcircuit's name in the Explorer Panel and then clicking on the canvas to drop a copy of it there. When a circuit is used like that, it will have the appearance of an integrated circuit (a small rectangle) with the various inputs on the left and the outputs on the right. Your main circuit should now look like this:

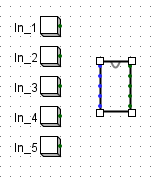


Figure 15: Subcircuit Added to Main Circuit

Place 5 LEDs (in the *Input/Output* library) to the right of the *CombineInputs* subcircuit and then wire all of the buttons and LEDs to the *CombineInputs* subcircuit:

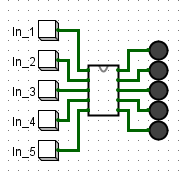


Figure 16: Completed Lab 2

### Testing the Circuit

Click on the "poke" tool and click on any of the numbered buttons. If the circuit is correct, each button will light that same number of LEDs. For example, poking the "3" button will light three LEDs.

### Renaming Circuits

It is possible to rename circuits in order to make them easier to find and use in later projects. To rename a circuit, click on that circuit in the Explorer Pane and enter its new name in the Attributes Pane. For this project, rename *main* to *SubCircuits.*

### Cleanup

Be sure the standard identifying information block is at the top left of the *SubCircuits* circuit: Name, "Lab 02 - Subcircuits", and today's date. Save the file as Lab 02 – SubCircuits.

# Lab 03 – BCD Adder

### Purpose

In this lab you will build an adder for two 4-bit BCD numbers.

### Procedure

Start a new circuit in Logisim-Evolution. Rename the *main* circuit to *BCD\_Adder.*

Logisim-Evolution contains a built-in adder that can be set to add anything from 1 to 32 bits, including the carry bits. Open the arithmetic folder and click one time on the adder, place it in the circuit, and then set the bit width in the attributes panel to 4.

The adder is made so the first number to be added is input on the top left and the second on the bottom left. The carry in bit is input on the top, and the carry out bit is output on the bottom. The sum of the two numbers appears at the node on the right. Create the following circuit:

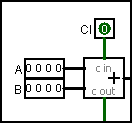


Figure 17: 4-Bit Binary Adder

Test the circuit by poking the various input bits and observing the output and CO bits.

### Values Greater Than Nine

This adder works fine as long as the sum is not greater than 9 (remember, this circuit is being designed to work for BCD, which does not have any values greater than 9). However, if 5 and 5 must be added, then the circuit would fail since there is no number 1010 in BCD. It is important to add a bit of logic to handle overflow; that is, any sum that is greater than 9.

The first step is to detect sums greater than 9. To do that, the 4-bit output bus is split into four single bits. Then, two OR gates and one AND gate is used to detect any of these bit patterns: 101x, 110x, 111x (that would be the three high-order bits for these four numbers: 1010, 1011, 1100, 1101, 1110, 1111). Finally, the Carry Out bit is OR'd with these three bits to produce a single bit that is high whenever the result of the adder is greater than 9.

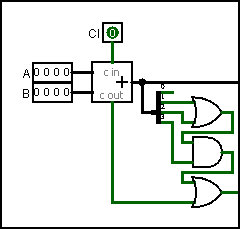


Figure 18: Detecting Numbers Greater Than 9

If the sum from the adder is greater than 9, then some work must be done to properly display the "one's" place. While counting in decimal, when the count changes from 09 to 10, the one's place is reset and starts counting from zero. When working with BCD, though, once the count reaches 916 it must skip A, B, C, D, E, and F to reset back to 016 for the next count. The simplest way to do this is add the value 01102 (hexadecimal 6) to any 4-bit binary number greater than 10012 (hexadecimal 9). The result is a 5-bit binary number where the first bit is 1 and the other four bits are the numbers 0-516. Thus, what follows 0916 is 1016, not 0A16. Here are two examples to help clarify this concept:

   1010 (Binary 10)  
+  0110 (Binary 6)  
 1 0000 (BCD 10)

   1110 (Binary 14)  
+  0110 (Binary 6)  
 1 0100 (BCD 14)

Once a binary number greater than 10012 (or decimal 9) is detected, a high is generated at the output of the last OR gate. That value is then combined with a constant low in a splitter to form the number 0110 (or decimal 6) for the input of a second adder. That same high is also combined with three lows to create the binary number 0001, which is used to drive the most significant digit of the output. That is, the ten's place will display a 1 rather than a 0.

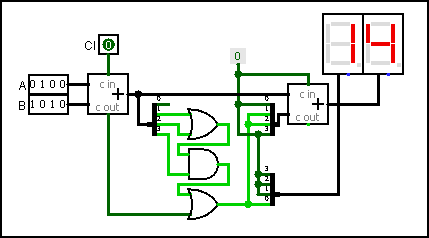


Figure 19: BCD Adder

In summary, when two BCD numbers are added, if the result is 9 or less, then that value is sent to a second adder, where 0 is added to it; but if the result is greater than 9, then 6 is added to the number.

The output from both the adder and the splitter are 4-bit numbers that can drive a hex display. Attach two hex displays, as in the above illustration, to make the output of the circuit easy to read.

*Note*: This circuit does not check to see if BCD numbers were input. It is assumed that there would have been a check elsewhere in the device to ensure that only BCD numbers are input to this circuit. However, by using a simple 4-bit input port, it is certainly possible to input numbers like 1010, which do not exist in BCD. Therefore, when testing this circuit, input only BCD numbers.

### Cleanup

Be sure the standard identifying information block is at the top left of the *BCD Adder* circuit: Name, "Lab 03: BCD Adder", and today's date. Save the file as *Lab 03 – BCD Adder*.

# Lab 04 – 8-Bit Adder

### Purpose

In this lab you will build an adder for two 8-bit numbers.

### Procedure

Start a new circuit in Logisim-Evolution. Rename the main circuit to Adder*.*

#### 1-Bit Adder SubCircuit

Create a new subcircuit named Adder\_1Bit and add the components in the diagram on the left to the subcircuit:

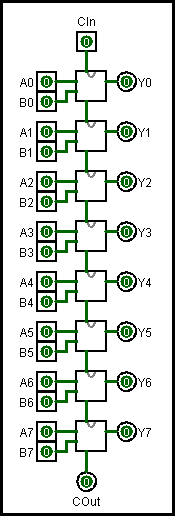


Figure 21: 8-BitAdder

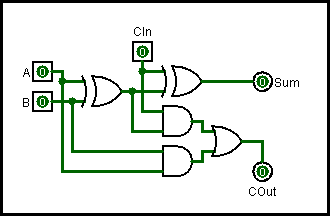


Figure 22: 1-Bit Adder

#### 8-Bit Adder SubCircuit

Create a new subcircuit named Adder\_8Bit and add the components found in the diagram on the right. Each of the small boxes in the center of the diagram is an Adder\_1Bit circuit as created in the previous step. Notice that bit 0 from both input A and B are fed into the first adder. In the same way, bits 1-7 from both input A and B are fed into the other adders. The Carry In (CIn) bit is input at the top and the Carry Out (COut) bit is output at the bottom. The COut bit from each stage is wired to the CIn bit of the next stage. Finally, the outputs from each stage are labeled Y0-Y7.

#### Main (Adder) Circuit

The only thing remaining is to create the Main circuit. In the following illustration, an 8-Bit Adder subcircuit (created in step two above) was placed in the circuit. Two 8-bit inputs, *In\_A* and *In\_B*, are wired to splitters and the bits from those inputs are fed to the appropriate input ports in the 8-Bit Adder subcircuit. The output bits are gathered through a splitter into an 8-bit bus and then sent to the Sum port. The inputs and outputs are aligned so they form something that looks like an addition problem. Notice that the image shows 1012+0112=10002.

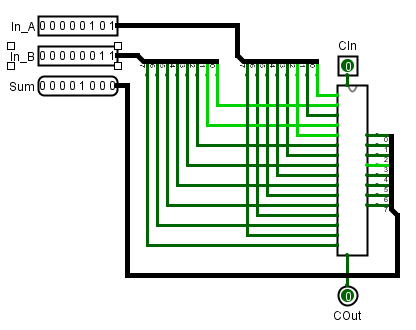


Figure 23: Main (Adder) Circuit

### Cleanup

Be sure the standard identifying information block is at the top left of the *Adder* circuit: Name, "Lab 04: 8-Bit Adder", and today's date. Save the file as *Lab 04 – Adder*.

# Lab 05 – 8-Bit Subtractor

### Purpose

In this lab you will build a subtractor for two 8-bit numbers.

### Procedure

Start a new circuit in Logisim-Evolution. Rename the main circuit to Subtractor*.*

#### 1-Bit Adder SubCircuit

Create a new subcircuit named Adder\_1Bit and add the components in the diagram below to the subcircuit:

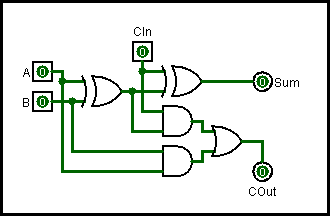


Figure 24: 1-Bit Adder

#### 8-Bit Subtractor SubCircuit

Create a new subcircuit named Subtractor\_8Bit and add the components found in the diagram below. Each of the small boxes just to the right of the center of the diagram is an Adder\_1Bit circuit as created in the previous step. The 8 bits of input A are fed into the top of each stage. The bits from input B, though, are fed through an XOR gate into the bottom of each stage. The other input to the XOR gate comes from the Cin input, as explained in the textbook. The COut bit from each stage is wired to the CIn bit of the next stage. Finally, the outputs from each stage are labeled Y0-Y7.

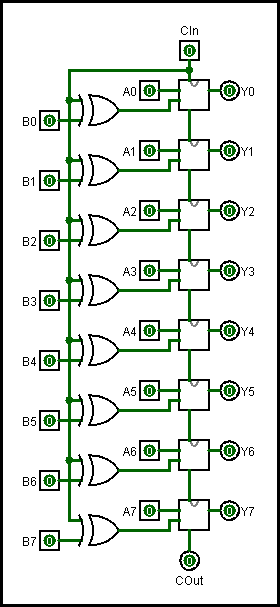


Figure 25: 8-BitSubtractor

#### Main (Adder) Circuit

The only thing remaining is to create the Main circuit. In the following illustration, an 8-Bit Subtractor subcircuit (created in step two above) was placed in the circuit. Two 8-bit inputs, *In\_A* and *In\_B*, are wired to splitters and the bits from those inputs are fed to the appropriate input ports in the 8-Bit Subtractor subcircuit. The output bits are gathered through a splitter into an 8-bit bus and then sent to the Difference port. The inputs and outputs are aligned so they form something that looks like a subtraction problem. Notice that the image shows 10002-1012=112 (remember that Cin must be active in order to subtract, otherwise the circuit is an 8-bit adder).

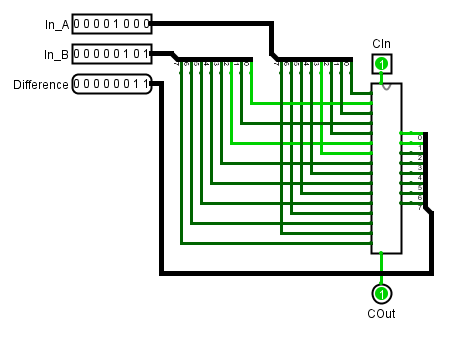


Figure 26: Main (Adder) Circuit

### Cleanup

Be sure the standard identifying information block is at the top left of the *Subtractor* circuit: Name, "Lab 05: 8-Bit Subtractor", and today's date. Save the file as *Lab 05 – Subtractor*.

# Lab 06 – Magic 8-Ball

### Purpose

This lab creates a circuit that simulates the old "Magic 8-Ball" toy. That toy was a small plastic sphere made to look like a billiards 8-ball. If someone "asked it a question" and then turned the ball upside down the answer would appear like magic in a small window on the bottom of the ball. This circuit is of little value, but is a fun sequential circuit that is quick and easy to build.

### Procedure

Place a counter (Memory Library) on the circuit drawing pad. In the counter's properties panel, set the number of Data Bits to 4 and the Maximum Value to 0x9. Make sure the "Action on Overflow" property is set for "wrap around." Wire a button (Input/Output Library) to the *R* port (that is for "Reset") on the west side of the counter and label that button "Reset". Wire a constant "1" to the *M3 [up]/M4 [down]* port on the west side of the counter. Wire one clock (Wiring Library) to both of the clock inputs on the counter (those are on the west side and are marked with a triangle). Finally, wire a constant 4-bit "0" (Wiring Library) to each of the input bits near the bottom of the west side of the counter.

Place a register (Memory Library) to the right of the counter and specify 4 Data Bits for the register. Wire the counter's data output to the Data (D) input of the register. Wire a pushbutton (Input/Output Library) to the clock input (bottom of the west edge, marked with a triangle) on the register and label that button “Answer.”

Place a decoder (Plexers library) to the right of the register and set the "Select Bits" to 4. Wire the register's output (Q) to the decoder's "Select" port (on the south edge).

Finally, place 10 LEDs to the right of the decoder. Wire one LED to each of the top 10 output ports on the decoder (the bottom 6 output ports are not used). The LEDs should be labeled: Yes; No; Maybe; Cannot Tell; Try Again Later; It Is Certain; It Is Doubtful; Unknown; Hazy Try Again; Absolutely.

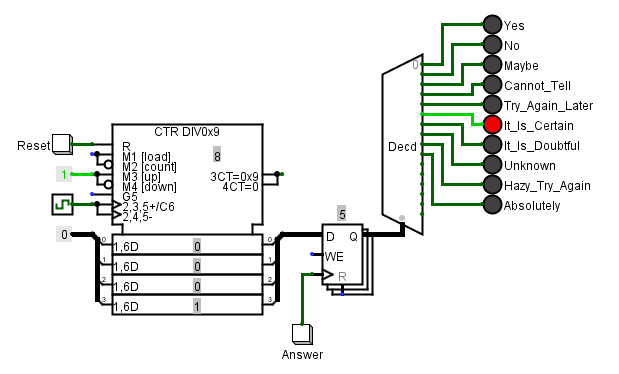


Figure 27: Electronic Magic 8-Ball

To test the circuit, start the clock using a fairly slow tick frequency (maybe about 4 Hz) and click the "Answer" button. It should light one of the LEDs, matching the value in the counter and giving you an answer to whatever question was asked. In the above illustration the counter has moved forward to 8, but the “Answer” button was clicked when the counter was at 5, so that is the answer displayed. Next, speed up the clock's frequency so the answers are more random and let it run. Ask a question (like "Am I smart?") and click the "Answer" button.

### Cleanup

Rename the *main* circuit to *8Ball*. Be sure the standard identifying information block is at the top left of the *8Ball* circuit: Name, "Lab 06: Magic 8-Ball", and today's date. Save the file as *Lab 06 – 8Ball*.

# Lab 07 – Hamming Parity Check

### Purpose

This circuit created in this lab inputs a 16-bit number that includes Hamming Parity bits. The circuit checks the input number and then outputs a zero if all bits are correct or the bit number if one is not correct.

### Procedure

This circuit design is largely left to the student; however, the following circuit is included as a hint for one way to proceed. This circuit takes an 11-bit number for input and intersperses parity bits as needed to create a 16-bit number with Hamming parity. Also, the overall parity bit (bit 16) is calculated and added to the Hamming Code

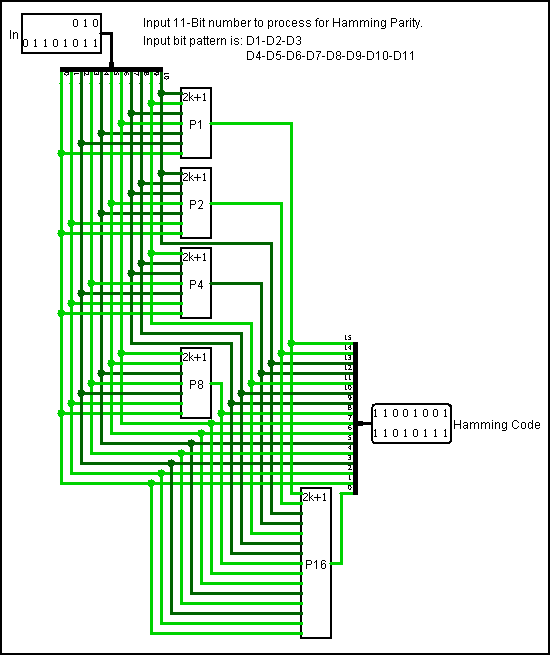


Figure 37: Create Hamming Parity Code

For the lab, it would seem to be a good idea to feed the 16 input data bits into appropriate parity checkers, and then feed the output of each parity checker to one input of an XOR gate which has the parity bit from the original input number feeding the second input gate. If the parity generated and the parity present are different, then the output of the XOR gate would be high and could be used to help indicate the bad bit.

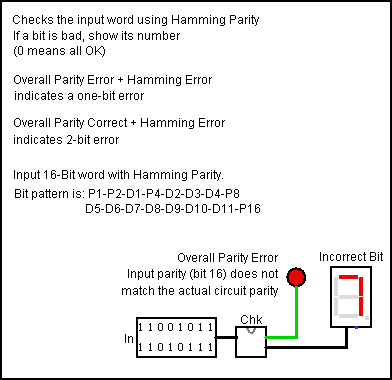


Figure 38: Checking Hamming Parity

The above figure shows the Hamming parity checker at work. Bit 7 is bad in the 16-bit input number and should be complemented to correct the number. The actual parity checking is taking place in the subcircuit named “Chk.”

### Cleanup

Rename the *main* circuit to *Hamming*. Be sure the standard identifying information block is at the top left of the *Hamming* circuit: Name, "Lab 7: Check Parity", and today's date. Save the file as *Lab 07 – Hamming*.

# Lab 08 – Timer

### Purpose

Create a simple timer that counts clock "ticks" either up or down.

### Procedure

The core of this circuit is two counters and a "D" Flip-Flop. Create the following circuit:

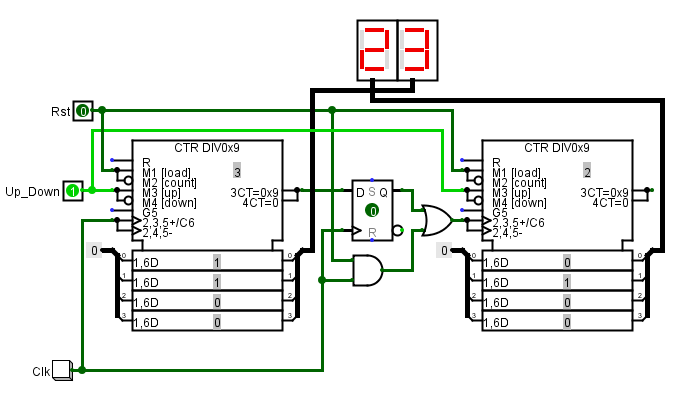


Figure 40: 2 Digit Up/Down Counter

Set the counters for 4 data bits, a maximum count of 0x9, and they should wrap around on overflow. That will set each counter to count from 0 to 9 and then start back at 0.

There are only three control signals on this circuit.

* When "Up\_Down" is high the circuit counts up; otherwise it counts down.
* "Rst" resets the count to 0. However, the counters will only “read” that signal on a clock pulse, so Rst must be set high and then the clock pulsed.
* The "Clk" button provides a clock signal

When counting up, the first counter increments by one on each clock tick. When it reaches 9, the overflow port goes high. That high becomes "Data" input into the D Flip-Flop, and on the next clock tick the output of that Flip-Flop is set high. That output is used as the clock for the next stage and increments that counter. When the first counter returns to zero, the overflow goes low so the "Data" input for the D Flip-Flop is now low. The next clock tick then resets the Flip-Flop and it is set for the next overflow (or, the next time the first counter reaches 9).

The explanation seems somewhat convoluted, but building and observing the circuit clarifies the explanation.

### Challenge

As created, the circuit is a two-digit decade counter; it counts either up or down between 00 and 99. However, it is not exactly right for a minutes-and-seconds timer.

Expand the circuit to three digits (a "minutes" and two "seconds" digits). The least significant digit should increase to 9, but the next digit should only increase to 5 (since "seconds" only go up to :59).

Finally, replace the "Clk" button with a clock component.

### Cleanup

Rename the *main* circuit to *Timer*. Be sure the standard identifying information block is at the top left of the *Timer* circuit: Name, "Lab 08: Timer", and today's date. Save the file as *Lab 08 – Timer*.

# Lab 09 – RAM Lab

### Purpose

Random Access Memory (RAM) is the most complex component in the Logisim-Evolution simulator; it requires a number of different control signals and uses input/output ports (that is, the same port is both input and output). It is used to hold volatile data which is data that will vanish as soon as the circuit's power is turned off. This lab is intended to create a simple circuit to exercise RAM by storing ASCII codes in RAM and then reading those codes out to a teletype display.

### Procedure

This is the RAM Lab circuit:

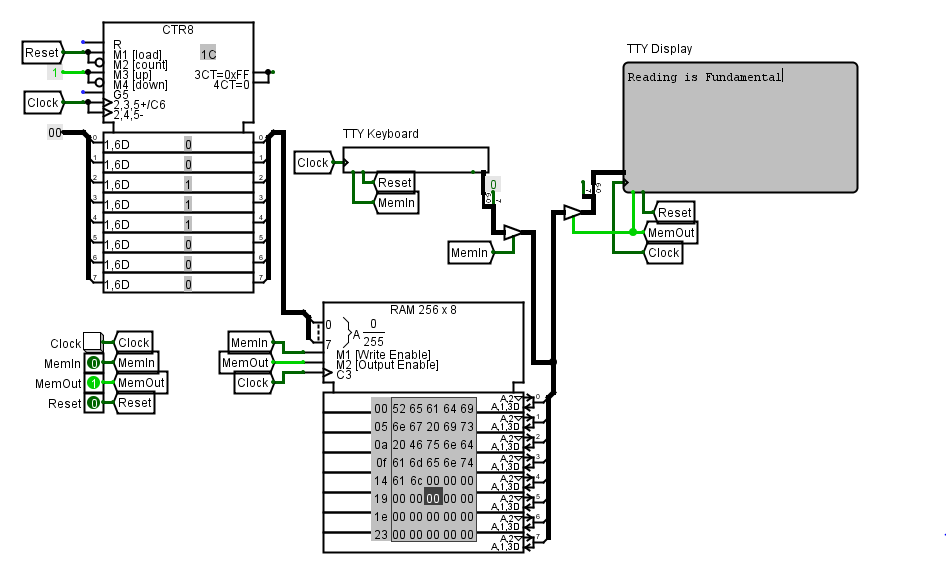


Figure 41: RAM Lab

The RAM component is the box at the center bottom of the circuit. It is set up so both the address and data are 8-bits wide. That means that it can store 8-bit bytes in up to 256 locations. The way that RAM works is that an address is placed on the Address port and then this is where the data will be stored or read. If the *Write Enable* input control is high, then RAM will store the data in memory and if *Output Enable* is high then RAM will place whatever data is in that memory location on the data port. These two control pins should never both be high at the same time or it will cause an input/output port conflict while making both control signals low effectively disables the RAM device.

This circuit is set up with a Keyboard input port and a Teletype display port. When operating, whatever is on the keyboard port will be stored in RAM and then that same message will be sent to the display.

Create the circuit as shown and then take these steps to exercise RAM:

1. Set Reset to 1 and all other controls to 0.
2. Tick the clock several times to reset the circuit.
3. Set Reset to 0 and MemIn to 1.
4. Click the TTY Keyboard and enter a text message.
5. Tick the clock once per letter in the message to load it into RAM.
6. Set MemIn to 0 and Reset to 1.
7. Tick the clock one time only.
8. Set Reset to 0 and MemOut to 1.
9. Tick the clock once per letter to load the TTY display.

About the only unusual feature of this circuit is the odd splitter coming out of the TTY Keyboard and another going into the TTY Display. ASCII codes are seven bits wide, so in order to store them in an 8-bit RAM, bit 7, the most significant bit, had to be set to 0. By the same token, bit 7, the most significant bit, had to be stripped from the ASCII code before it was sent to the TTY Display.

### Cleanup

Rename the *main* circuit to *RAM*. Be sure the standard identifying information block is at the top left of the *RAM* circuit: Name, "Lab 09: RAM", and today's date. Save the file as *Lab 09 – RAM*.

# Lab 10 – ROM Lab

### Purpose

Read Only Memory (ROM) is used to hold data that must persist; that is, data that will will be available after the circuit's power has been recycled. This lab is intended to create a simple circuit to exercise ROM by storing ASCII codes and then reading those codes out to a teletype display.

### Procedure

This is the ROM Lab circuit:

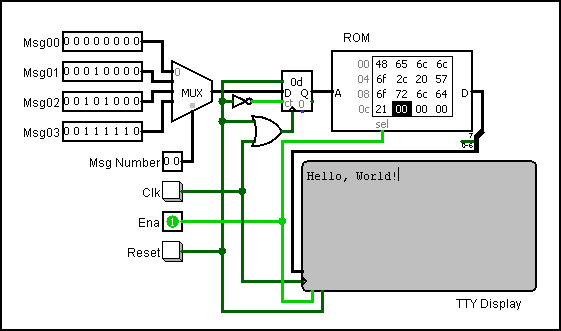


Figure 42: ROM Lab

The ROM component is identified in the top left corner of the circuit (it contains numbers). The ROM uses an 8-bit address and 8-bit data width. When an address is present on the *A* input then whatever data are at that address is made available on the *D* (for "Data") port. Note that this component does not need a clock, the data contained at the specified address are constantly present on the *D* port.

At the top left corner of this circuit is a 4-to-1 multiplexor. This permits the user to enter up to four predefined addresses and easily switch between them. The addresses are labeled *Msg00* to *Msg03* since this lab generates simple ASCII messages from ROM.

On the bottom right corner of this circuit is a TTY Display that will display the ASCII message that is contained in ROM.

The circuitry used in this lab is fairly standard and should be easy to understand. The only unusual feature is the odd splitter off the ROM *D* port. ASCII codes are seven bits wide but are stored in an 8-bit RAM; so the most significant bit, bit 7, had to be stripped from the ASCII code before it was sent to the TTY Display.

Load the ROM memory by right-clicking on the device and selecting "Edit" from the popup menu. Copy/paste the following into the ROM:

48 65 6c 6c 6f 2c 20 57  
6f 72 6c 64 21 00 00 00  
44 69 67 69 74 61 6c 20  
6c 6f 67 69 63 20 69 73  
20 66 75 6e 2e 00 00 00  
54 68 69 73 20 69 73 20  
61 20 52 4f 4d 20 74 65  
73 74 2e 00 00 00 57 68  
61 74 27 73 20 74 68 65  
20 4d 53 42 20 66 6f 72  
20 74 68 69 73 20 6c 65  
74 74 65 72 3f 00 00 00

Enter the following addresses into the *Msg* inputs:

Msg00: 0000 0000  
Msg01: 0001 0000  
Msg10: 0010 1000  
Msg11: 0011 1110

Finally, complete these steps to exercise ROM:

1. Right-click on the ROM component and click “edit” to load the memory.
2. Enter the four message addresses in *Msg00*, *Msg01*, *Msg02*, and *Msg03*.
3. Set Reset to 1.
4. Enter the desired message number on the Mux Select input port.
5. Tick the clock one time.
6. Set Reset to 0.
7. Tick the clock enough times to send the entire message to the TTY Display.

### Cleanup

Rename the *main* circuit to *ROM*. Be sure the standard identifying information block is at the top left of the *ROM* circuit: Name, "Lab 10: ROM", and today's date. Save the file as *Lab10–ROM*.