LOGISIM-EVOLUTION LAB MANUAL

GEORGE SELF

July 2019 – Edition 4.0

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
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.

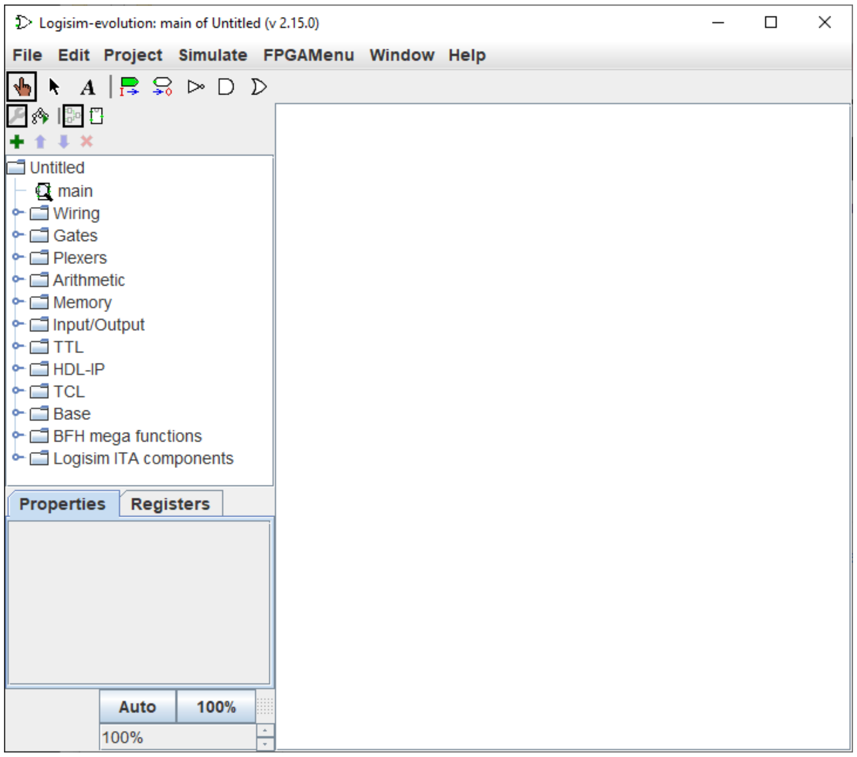


Figure 1.1: Logisim-evolution Initial Screen

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*

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.

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

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

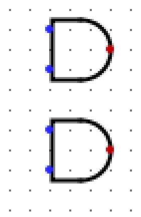
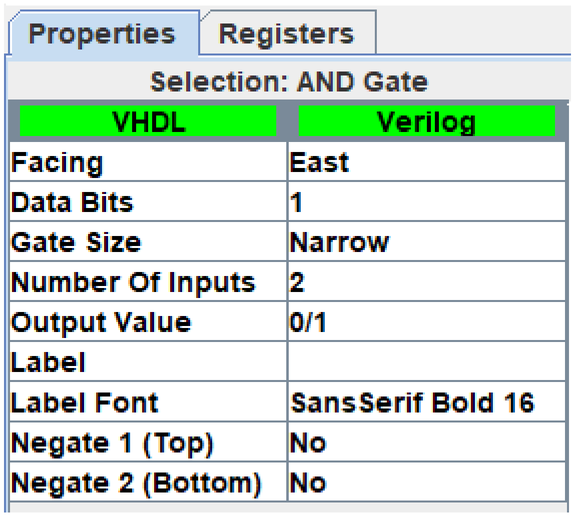


Figure 1.2: Two AND Gates

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.



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.

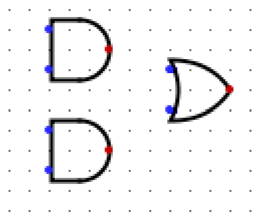


Figure 1.4: OR Gate Added to Circuit

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.

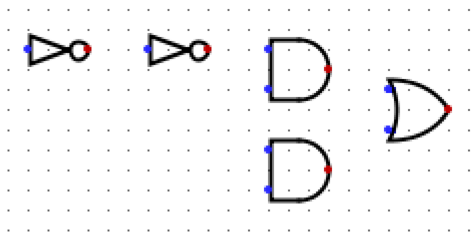


Figure 1.5: Two NOT Gates Added to Circuit

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.

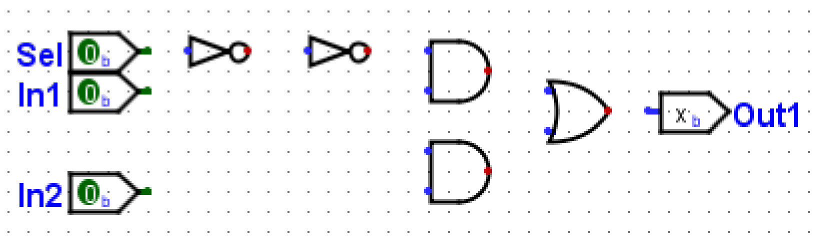


Figure 1.6: Inputs and Output Added

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.

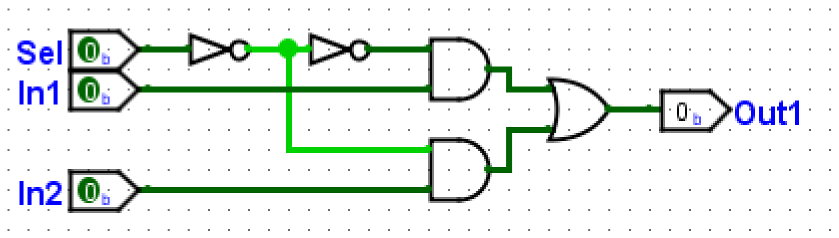


Figure 1.7: Circuit Wiring Added

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.

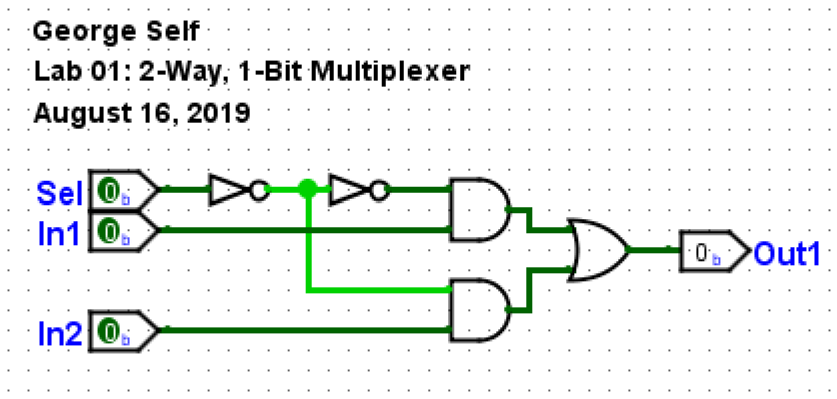


Figure 1.8: Simple multiplexer

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_Mux21` (that stands for multiplexer, 2-way, 1-bit). 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.

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

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.

2 VENDING MACHINE

2.1 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.

2.2 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.

2.2.1 *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 *Prod01* and *Prod02* 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.

2.2.2 *Subcircuit Descriptions*

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

2.2.2.1 *main*

The main circuit is the interface between a human customer and the simulator, as shown in Figure 2.1.

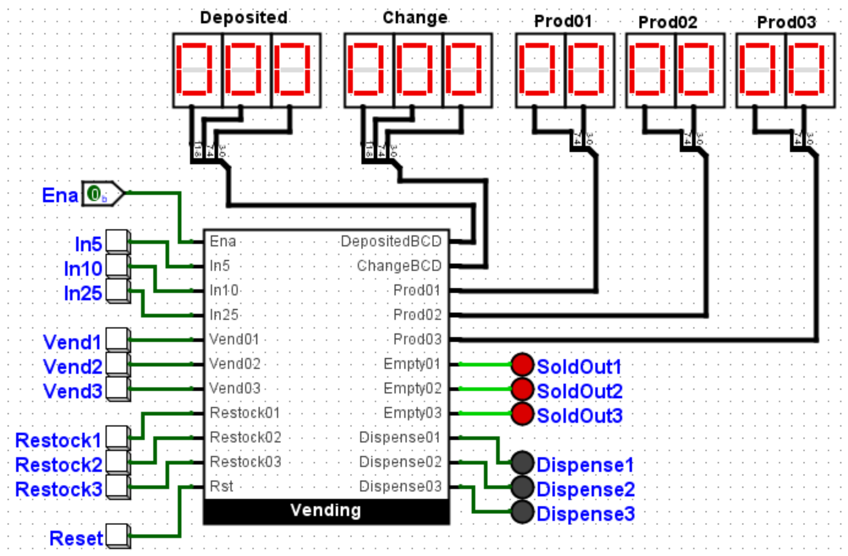


Figure 2.1: Vending Machine Main Circuit

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.

2.2.2.2 *Activator*

The Activator subcircuit receives a signal from the Bank subcircuit that indicates how much money has been collected. The Activator returns the Binary Coded Decimal (BCD) Total and Change values and sets a signal to activate the Dispenser subcircuit once 75 cents has been deposited. Figure 2.2 illustrates the Activator subcircuit.

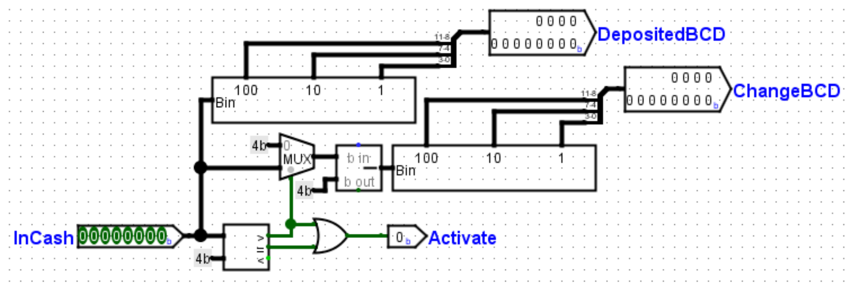


Figure 2.2: Activator Subcircuit

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.

2.2.2.3 *Bank*

The Bank subcircuit keeps a running total of the amount deposited and sends that total to the Activator subcircuit. Figure 2.3 illustrates the Bank subcircuit.

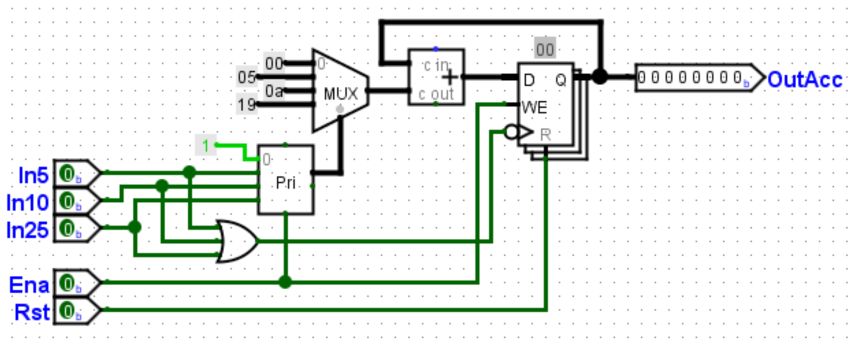


Figure 2.3: Bank Subcircuit

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.

2.2.2.4 *Dispenser*

The `Dispenser` subcircuit dispenses the three products available in the machine. Figure 2.4 illustrates the `Dispenser` subcircuit.

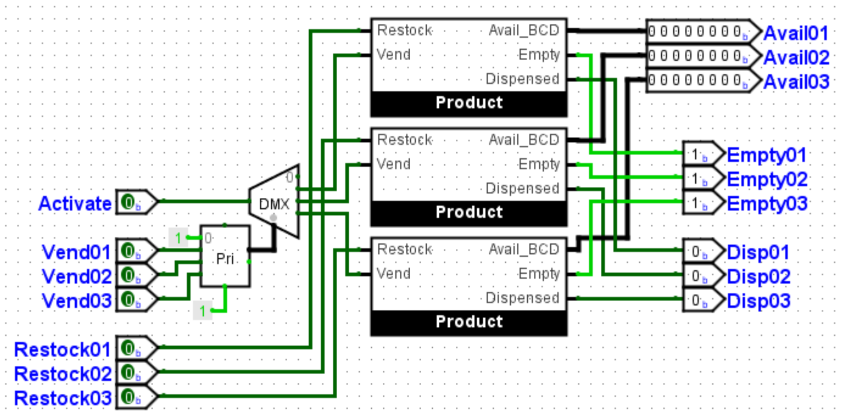


Figure 2.4: Dispenser Subcircuit

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.

2.2.2.5 *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.

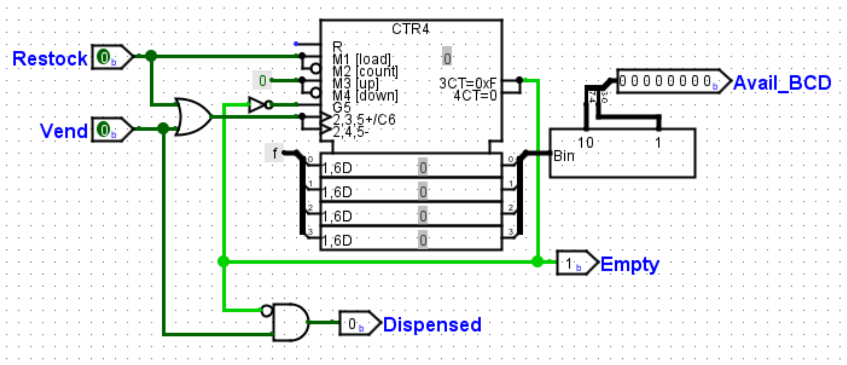


Figure 2.5: Product Subcircuit

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=0xF* 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.

2.2.2.6 *Vending*

The `Vending` subcircuit consolidates the other subcircuits into an Integrated Circuit (IC) that is used in the `main` circuit. Figure 2.6 illustrates the `Vending` subcircuit.

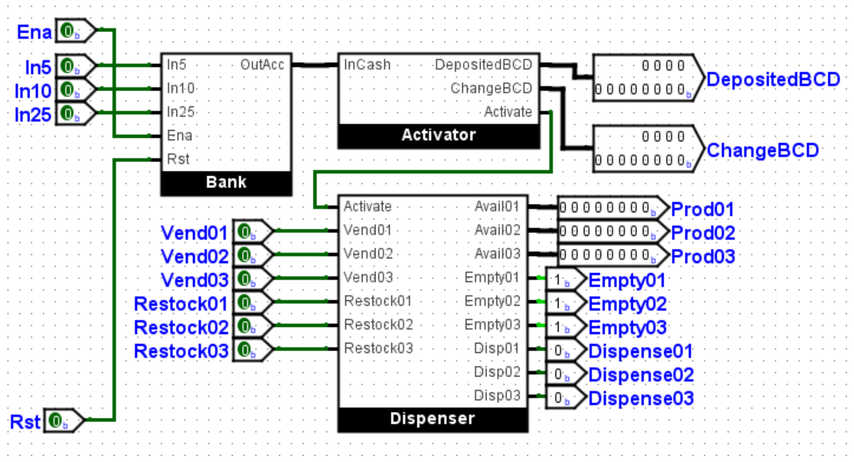


Figure 2.6: Vending Subcircuit

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

2.3 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.

2.4 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 05: Vending Machine
February 16, 2018
```

Save the file with this name: *`Lab05_Vend`* and submit that file for grading.

3 PROCESSOR

3.1 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.

3.1.1 *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.

3.2 PROCEDURE

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

3.2.1 *Arithmetic-Logic Unit*

This processor starts with a simple Arithmetic Logic Unit (ALU), as in Figure 3.1.

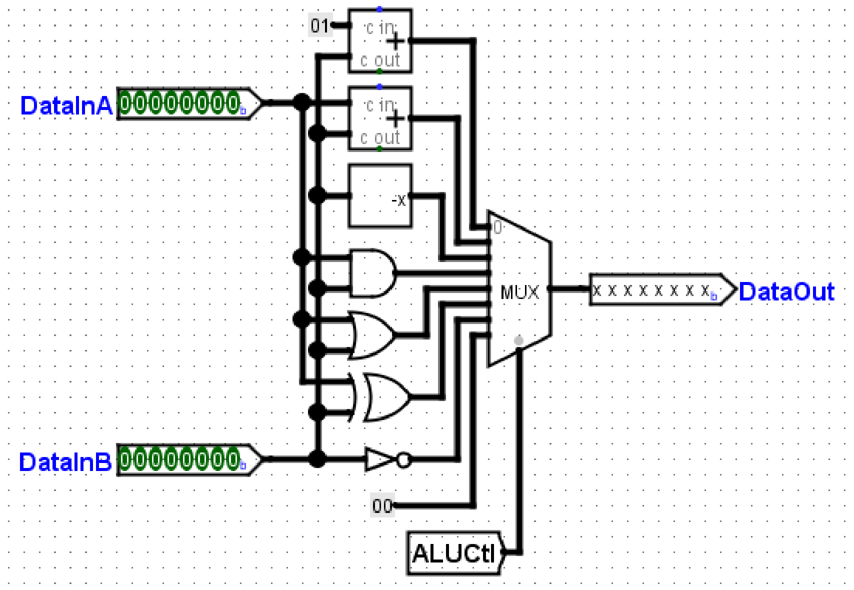


Figure 3.1: Simple ALU

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 3.2 shows only the left side of the ALU.

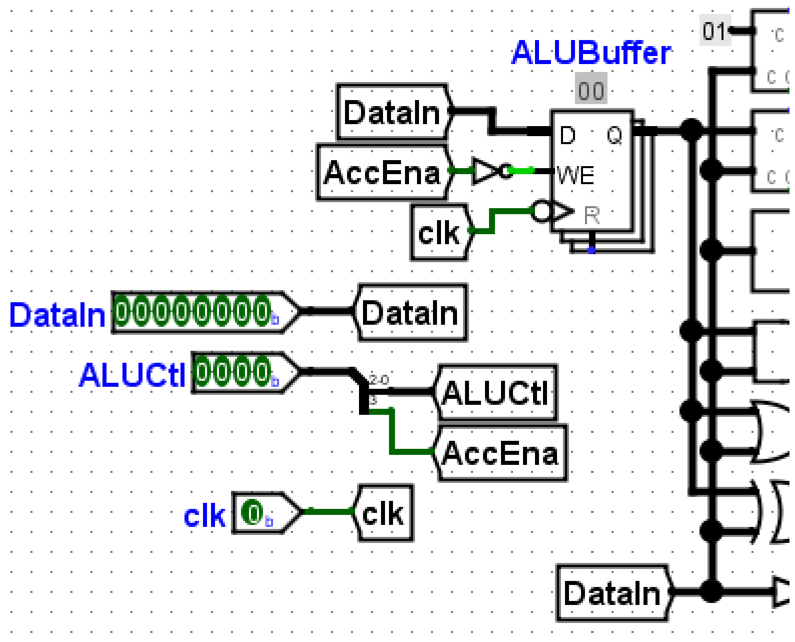


Figure 3.2: Left Side of ALU

Figure 3.3 shows the right side of the ALU.

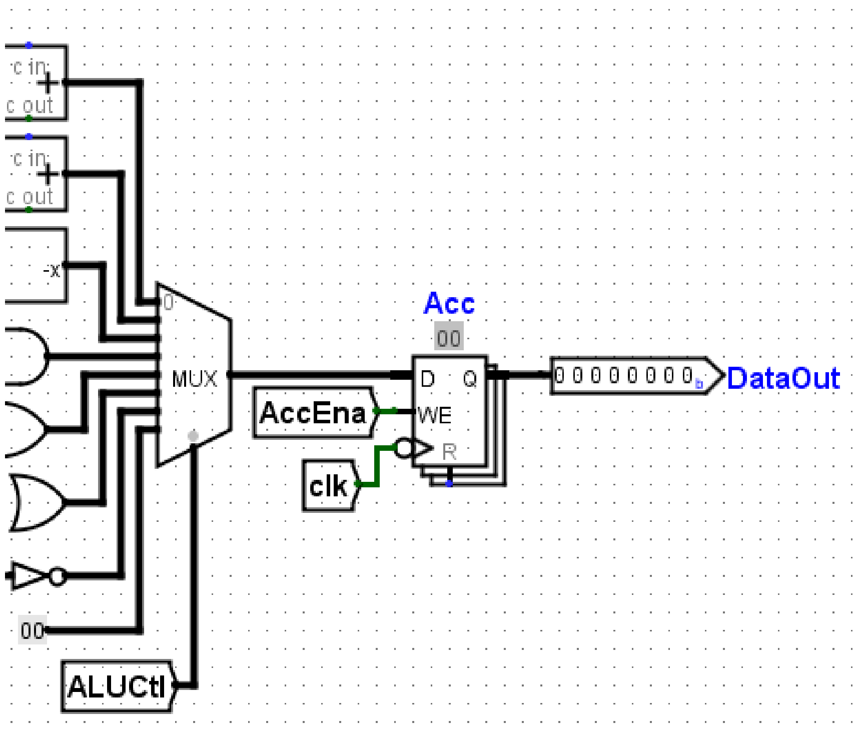


Figure 3.3: Full ALU

The simple ALU functions are found in the center of Figure 3.2. 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.

3.2.2 *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 3.4 illustrates the `GenReg` subcircuit.

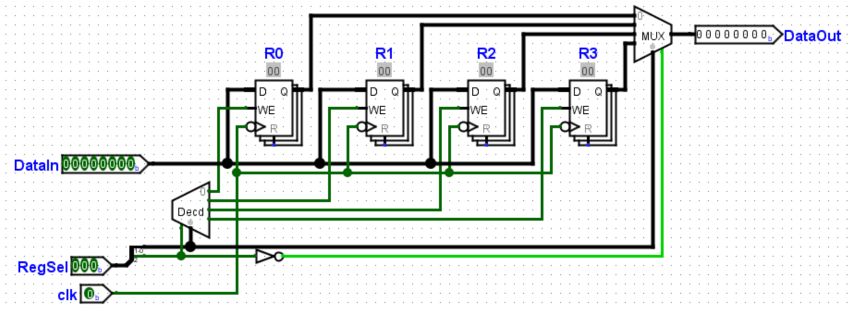


Figure 3.4: General Registers

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.

3.2.3 *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 3.5 illustrates the `Control` subcircuit.

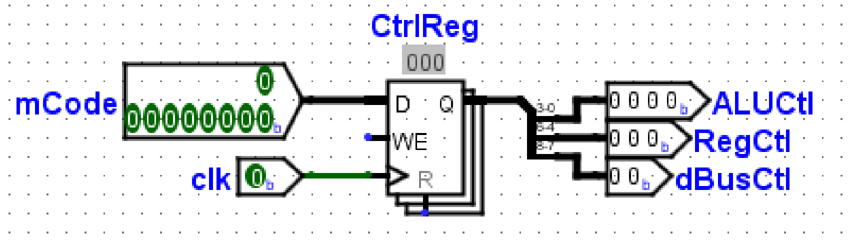


Figure 3.5: Control Subcircuit

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.

3.2.4 *Main*

The main circuit ties the three subcircuits together with three control busses and one data bus. Figure 3.6 illustrates the main circuit.

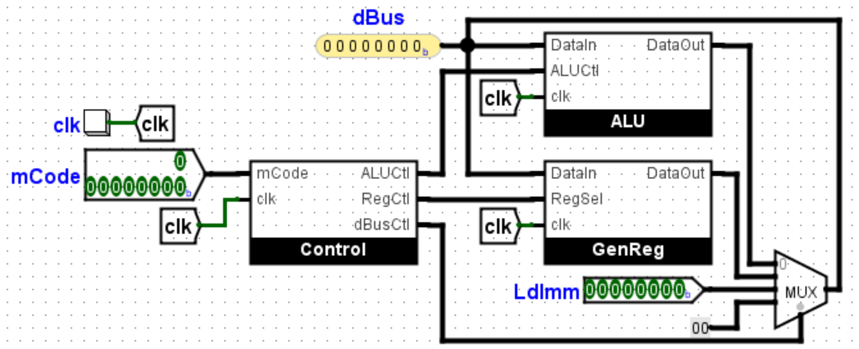


Figure 3.6: Main Circuit

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.

3.2.5 *Testing the Circuit*

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

3.2.5.1 *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 3.1: R0 <- LdImm

3.2.5.2 *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 3.2: R1 <- LdImm

3.2.5.3 *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 3.3: ALU <- LdImm

3.2.5.4 *Increment R0*

Use the LdImm function to initialize R0.

Incrementing the value in R0 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, *R0* will contain the original value of the *R0*+1.

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

Table 3.4: R0 <- Inc(R0)

3.2.5.5 *Add R0 And R1, Store In R0*

Use the LdImm function to initialize R0 and R1.

Adding the values of *R0* and *R1* and storing the result in *R0* 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 *R0* and *R1* will be stored in *R0*.

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 3.5: R0 <- R0 + R1

3.2.5.6 *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 3.6: R0 <- R0 - R1

3.2.5.7 *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 3.7: R1 <- R0

3.2.5.8 *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 3.8: R0 <-> R1

3.3 ABOUT PROGRAMMING LANGUAGES

The codes that were input for the last example (swap *R0* 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 example, "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 Read Only Memory (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.

3.4 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 *R0* back into *R0* (*R0* <- ~*R0*). The subtraction example will help with this function.

4. Store the bitwise NOT of the value in *R0* back into *R0* (*R0* <- *R0'*).

3.5 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 11: Processor
April 5, 2018
```

Save the Processor circuit in a file with this name: *Lab11_Processor*. Complete the code required in the Challenge and store that in a text file with the name *Lab11_Code.txt*. Submit both files for grading.

4 ELEVATOR

4.1 PURPOSE

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

4.2 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 4.1 is one student's concept from an earlier class.

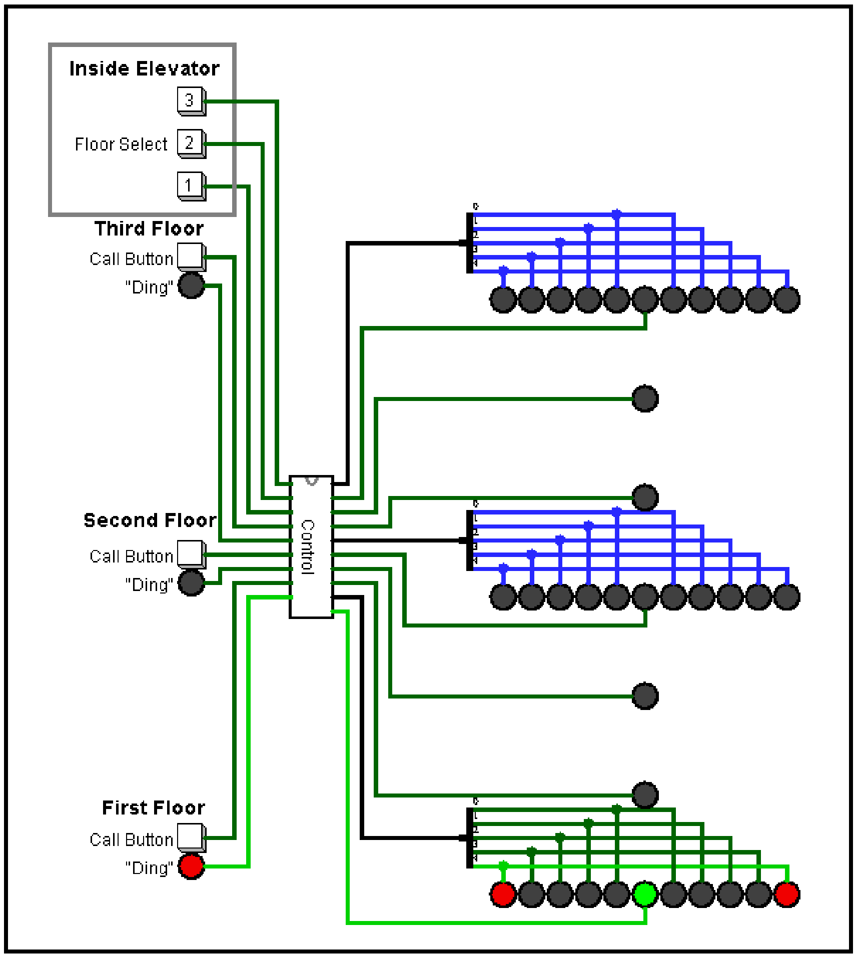


Figure 4.1: Example Elevator Simulator

4.3 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.

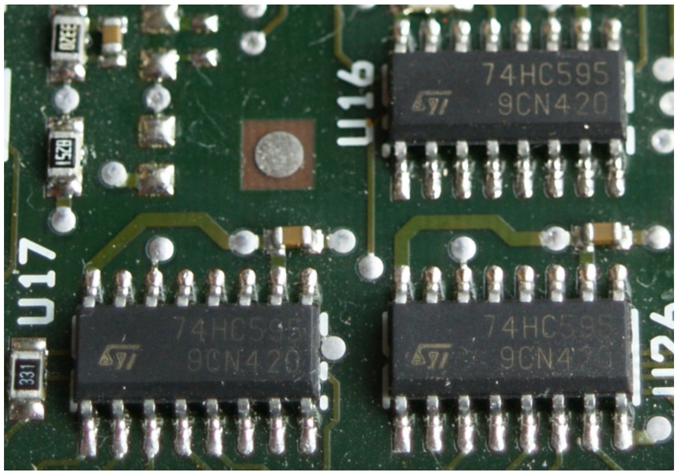


Figure A.1: Three Surface-Mounted Integrated Circuits

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.

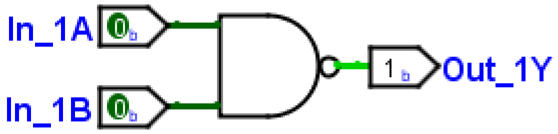


Figure A.2: 7400: Single NAND Gate Circuit

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.

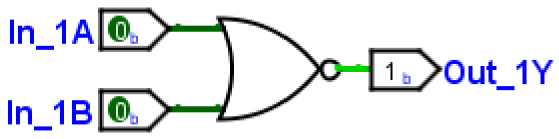


Figure A.3: 7402: Single NOR Gate Circuit

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.

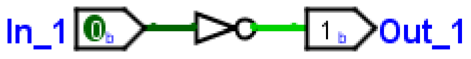


Figure A.4: 7404: Single Inverter Circuit

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.

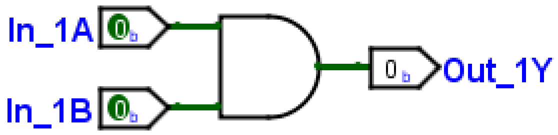


Figure A.5: 7408: Single AND Gate Circuit

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.

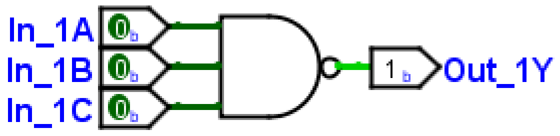


Figure A.6: 7410: Single 3-Input NAND Gate Circuit

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.

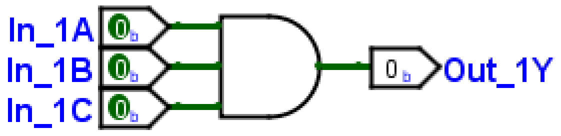


Figure A.7: 7411: Single 3-Input AND Gate Circuit

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.

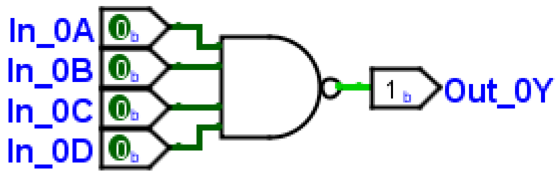


Figure A.8: 7413: Single 4-Input NAND Gate Circuit

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.

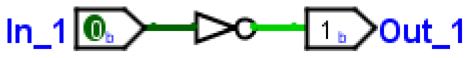


Figure A.9: 7414: Single Inverter Circuit

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.

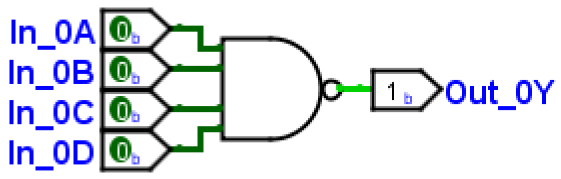


Figure A.10: 7418: Single 4-Input NAND Gate Circuit

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.

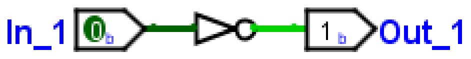


Figure A.11: 7419: Single Inverter Circuit

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.

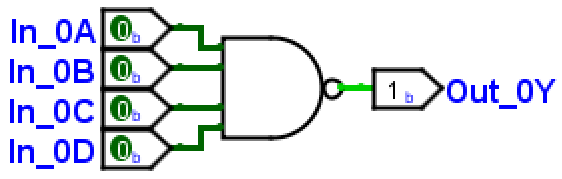


Figure A.12: 7420: Single 4-Input NAND Gate Circuit

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.

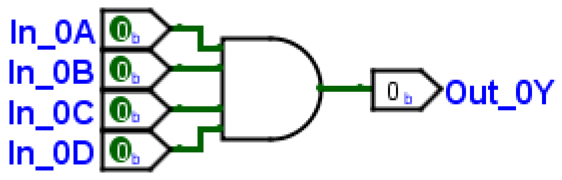


Figure A.13: 7421: Single 4-Input AND Gate Circuit

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.

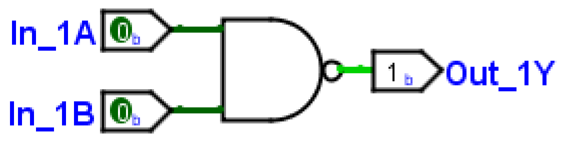


Figure A.14: 7424: Single NAND Gate Circuit

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.

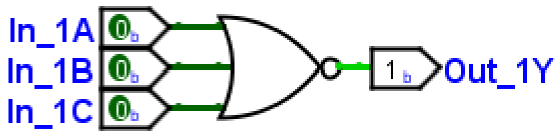


Figure A.15: 7411: Single 3-Input NOR Gate Circuit

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.

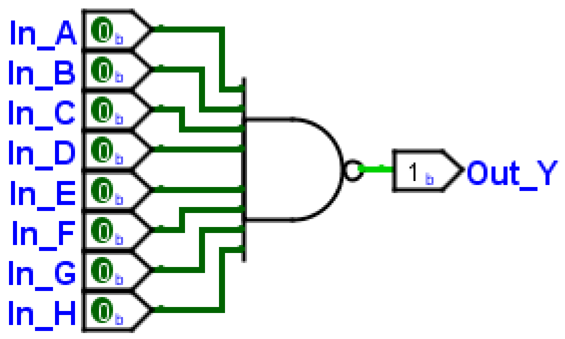


Figure A.16: 7430: Single 8-Input NAND Gate

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.

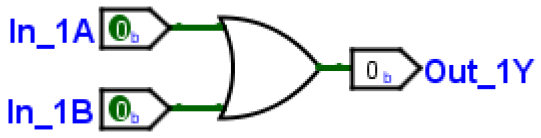


Figure A.17: 7432: Single OR Gate Circuit

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.

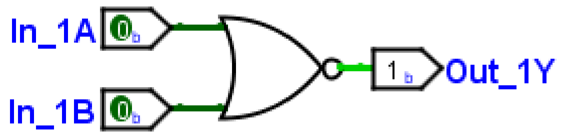


Figure A.18: 7436: Single NOR Gate Circuit

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.

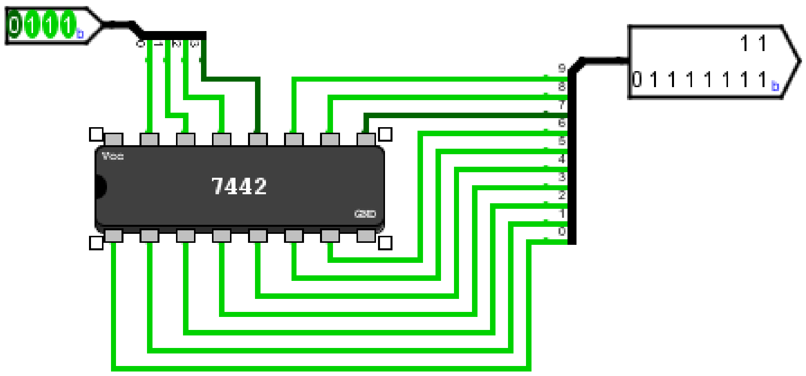


Figure A.19: 7442: BCD to Decimal Decoder

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.

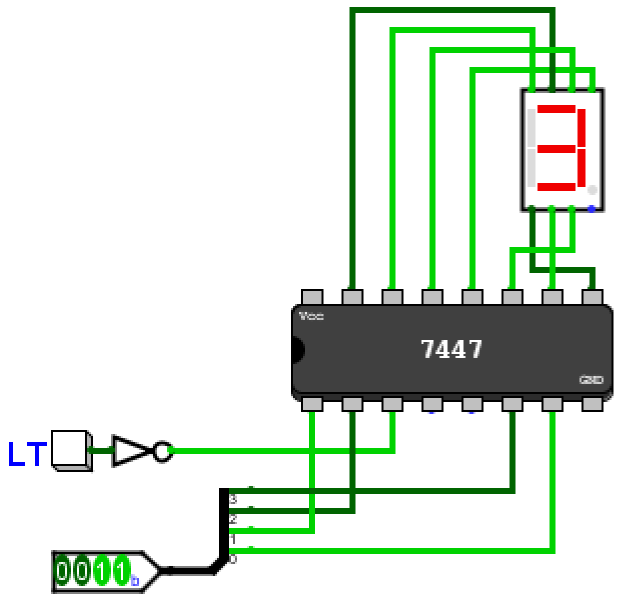


Figure A.20: 7447: BCD to 7-Segment Decoder

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.

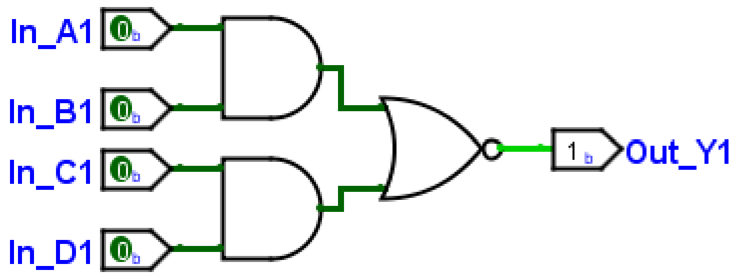


Figure A.21: 7451: Single AND-OR-INVERT Gate Circuit

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.

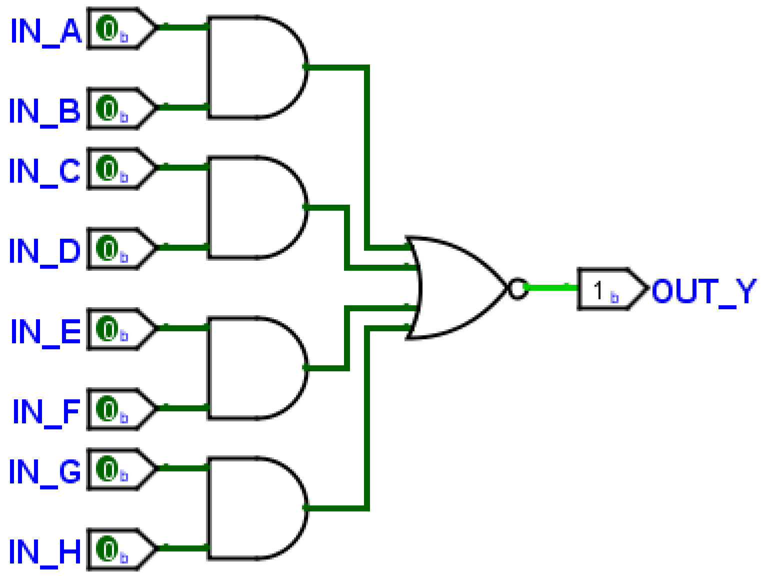


Figure A.22: 7454: Four Wide AND-OR-INVERT Gate Circuit

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.

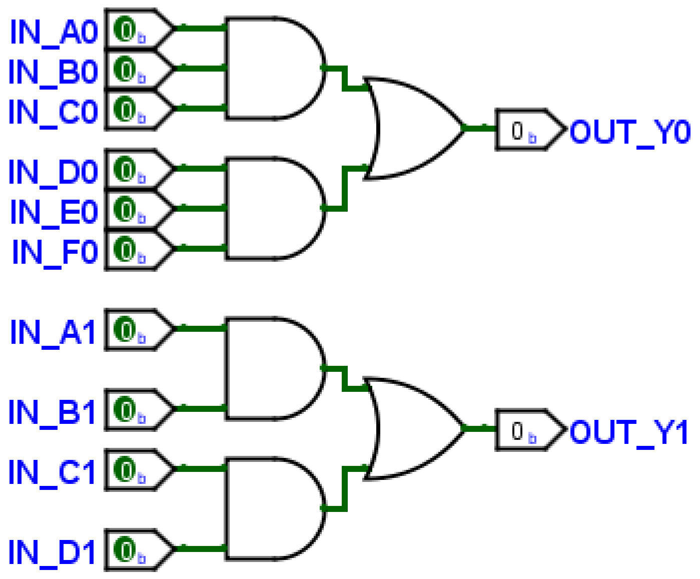


Figure A.23: 7458: Dual AND-OR Gate Circuit

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.

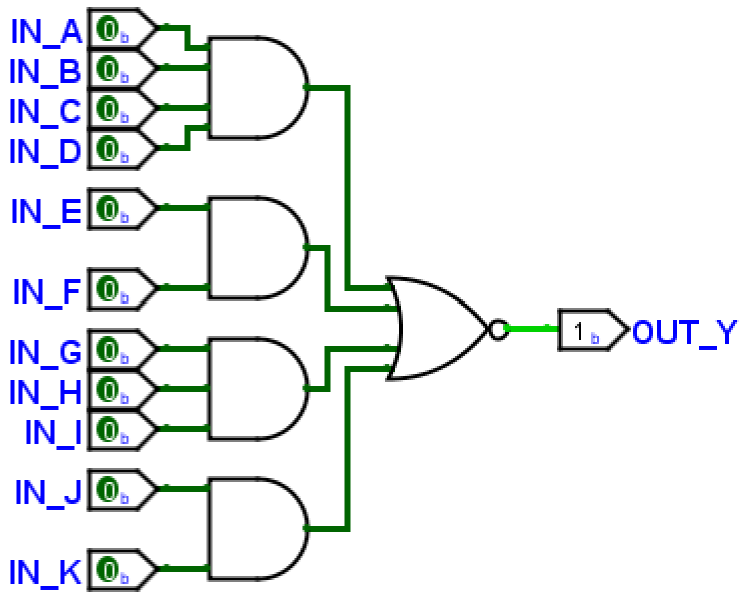


Figure A.24: 7464: 4-2-3-2 AND-OR-INVERT Gate Circuit

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.

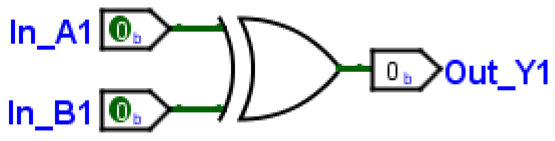


Figure A.25: 7486: Single XOR Gate Circuit

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.

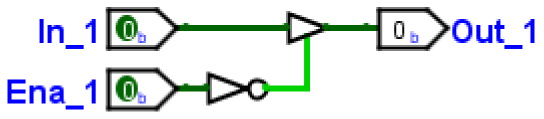


Figure A.26: 74125: Single Buffer Circuit

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.

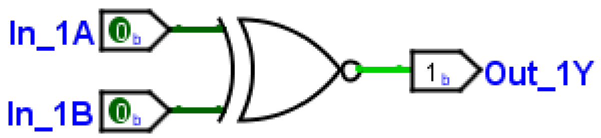


Figure A.27: 74266: Single XNOR Gate Circuit

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: $\sum 2$	Sum, bit 2
Input 2: B2	Operand B, bit 2
Input 3: A2	Operand A, bit 2
Output 4: $\sum 1$	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: $\sum 4$	Sum, bit 4
Input 11: B4	Operand B, bit 4
Input 12: A4	Operand A, bit 4
Output 13: $\sum 3$	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 17, 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.)