

---

# EXPLORING DIGITAL LOGIC WITH LOGISIM

George Self



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

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

[selfg@cochise.edu](mailto:selfg@cochise.edu)

---

## TABLE OF CONTENTS

Table of Contents.....	3
Foreword.....	7
Introduction to the Study of Digital Logic .....	7
Introduction to the Author.....	7
Introduction to This Book.....	7
Introduction to Logisim .....	8
About the Creative Commons License .....	8
1: About Digital Logic .....	9
1.1 Introduction.....	9
1.2 Boolean Algebra .....	10
2: Boolean Algebra.....	13
2.1 Introduction.....	13
2.2: Primary Logic Functions .....	15
2.3: Lab – Introduction to Logisim.....	20
2.4: Secondary Logic Functions .....	24
2.5: Lab – Logisim Subcircuits.....	28
2.6: Lab – Logisim Circuit Analyzer .....	33
2.7: Univariate Boolean Algebra Properties.....	39
2.8 Multivariate Properties .....	43
2.9: DeMorgan's Theorem.....	49
2.10: Boolean Functions.....	53
2.11: Boolean Functions and Properties Summary.....	55
3: Binary Mathematics .....	57
3.1: Introduction to Number Systems.....	57
3.2: Converting Between Bases.....	64
3.3: Binary Addition.....	73
3.4: Binary Subtraction.....	76
3.5: Codes.....	86
3.6: Lab – BCD Adder.....	93
3.7: Lab – 8-Bit Binary to BCD.....	96

4: Simplification of Boolean Expressions .....	99
4.1 Introduction.....	99
4.2 Creating Boolean Expressions .....	100
4.3: Minterms and Maxterms.....	102
4.4: Canonical Form.....	113
4.5: Simplification Using Algebraic Methods .....	118
4.6: Karnaugh Maps.....	122
4.7: Reed-Müller Logic.....	141
4.8: Quine-McCluskey Simplification Method.....	148
4.9: Automated Tools.....	157
5: Combinational Logic.....	167
5.1 Introduction.....	167
5.2: Adders and Subtractors.....	168
5.3: Lab – 8-Bit Adder .....	174
5.4: Lab – 8-Bit Subtractor.....	176
5.5: Comparators.....	179
5.6: Encoders/Decoders .....	181
5.7: Lab – Magic 8-Ball.....	192
5.8: Multiplexers/Demultiplexers .....	194
5.9: Lab – Adding Machine .....	197
5.10: Error Detection.....	204
5.11: Lab – Hamming Parity Check.....	210
6: Sequential Logic .....	213
6.1 Introduction.....	213
6.2 Timing Diagrams .....	214
6.3: Finite State Machines .....	216
6.4: Flip-Flops .....	220
6.5: Registers .....	224
6.6: Lab – Guessing Game .....	229
6.7: Counters .....	231
6.8: Lab – Timer .....	242
7: Applications.....	245

7.1 Introduction.....	245
7.2: Logic Arrays .....	246
7.3: Memory (ROM and RAM).....	249
7.4: Lab –RAM Lab.....	252
7.5 Lab – ROM Lab.....	254
7.6: Arithmetic Logic Unit (ALU).....	256
7.7: Lab – Arithmetic and Logic Unit .....	259
7.8: Central Processor Unit (CPU): Fundamentals .....	264
7.9: Central Processor Unit (CPU): Components.....	270
7.10: Central Processor Unit (CPU): Control .....	278
7.11: Central Processor Unit (CPU): Finalization.....	284
7.12: Lab – Vending Machine .....	285
7.13: Lab – Elevator.....	294
App A: Integrated Circuit Part Numbers .....	297
Integrated Circuit Part Numbers .....	297
Manufacturer Prefixes.....	297
Logic Subfamilies .....	298
App B: References.....	301
Books .....	301
Web Sites.....	301



---

## 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.***

I had two goals in mind when I wrote this book:

1. Level. I have over a dozen digital logic books in my personal library. Unfortunately, most of them are designed for third or fourth year electronics engineering or computer science students and presume a background that includes advanced mathematics and various engineering classes. For example, one of the books discusses topics like physically building circuits from discrete components and then calculating the heat rise of those circuits while operating at maximum capacity. The classes that I teach are for students in their second year of a Computer Information Systems program and the existing books are just plain at the wrong level.
2. Cost. Most digital logic books are priced at \$150 (and up). I have a personal believe that it is wrong to charge students that much for a book; especially given that they may have to purchase ten or more books during a semester, plus lab fees, tuition, and other expenses. Moreover, book publishers attempt to force students to purchase new books, rather than used, with such tricks as rapid edition changes, bundling materials, and online versions that "expire" at the end of the semester. This book is but a tiny drop in the proverbial textbook ocean; but I intend to do anything that I can to keep the cost of books for students in my classes as low as possible.

The book includes a number of lab exercises designed for the Logisim simulator, which is an excellent logic simulator that is available free of charge. Also, most of the circuit illustrations in this book were created with Logisim and then screen captured for this book.

*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

Logisim 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 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 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.



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/>.

---

# 1: ABOUT DIGITAL LOGIC

## 1.1 INTRODUCTION

Digital logic is the study of how logic is used in digital devices that do useful tasks in fields like communication, business, traffic control, space exploration, and medicine, along with millions of other uses in science and industry. This definition has two main components: logic and digital. "Logic" is the branch of philosophy that concerns making reasonable judgment based on sound principles of inference. It is a method of problem solving based on a linear, step-by-step procedure. "Digital" is a system of mathematics where only two possible values exist: true and false (or 1 and 0). While this approach may seem limited, it actually works quite nicely in computer circuits where true and false can be easily represented by the presence or absence of voltage, and simple digital expressions can be expanded to communicate very complex relationships and interactions among any number of individual conditions.

Digital logic is not the same as programming logic, though there is some relationship between them. A programmer would use the constructs of logic within a high-level language, like Java or C++, to get a computer to handle some task. On the other hand, an engineer would use digital logic with hardware devices to build a machine, like an alarm clock or calculator, which handles some task. In broad strokes, then, programming logic concerns writing software while digital logic concerns building hardware. Digital logic may be divided into two broad classes: combinational logic, in which the outputs are determined by both the logical function being performed and the input states at one particular moment; and sequential logic, in which the outputs depend on the prior states of outputs from other elements.

## 1.2 BOOLEAN ALGEBRA

### ARISTOTLE

The Greek philosopher Aristotle founded a system of logic based on only two types of propositions: true and false. His bivalent (two-mode) definition of truth led to the four foundational laws of logic: the Law of Identity (A is A); the Law of Non-contradiction (A is not non-A); the Law of the Excluded Middle (either A or non-A); and the Law of Rational Inference. These so-called Laws function within the scope of logic where a proposition is limited to one of two possible values, like "true" and "false"; but they do not apply in cases where propositions can hold other values.

The English mathematician George Boole (1815-1864) sought to give symbolic form to Aristotle's system of logic. Boole wrote a treatise on the subject in 1854, titled *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*, which codified several rules of relationship between mathematical quantities limited to one of two possible values: true or false, 1 or 0. His mathematical system became known as Boolean algebra.

All arithmetic operations performed with Boolean quantities have but one of two possible outcomes: 1 or 0. There is no such thing as "2" or "-1" or "1/2" in the Boolean world. It is a world in which all other possibilities are invalid by definition. As one might guess, this is not the kind of math needed to balance a checkbook; however, Claude Shannon of MIT recognized how Boolean algebra could be applied to on-and-off circuits, where all signals are characterized as either "high" (1) or "low" (0). His 1938 thesis, titled *A Symbolic Analysis of Relay and Switching Circuits*, put Boole's theoretical work to use in telephone switching, a system Boole never could have imagined, giving us a powerful mathematical tool for designing and analyzing digital circuits.

There are a number of similarities between Boolean algebra and real-number algebra. It is important to bear in mind that the system of numbers defining Boolean algebra is severely limited in scope; there can only be one of two possible values for any Boolean variable: 1 or 0. Consequently, the "Laws" of Boolean algebra often differ from the "Laws" of real-number algebra, making possible such statements as  $1 + 1 = 1$ , which would be considered absurd if using real-number algebra. Once the premise that all quantities in Boolean algebra are limited to 1 and 0 is comprehended, the "nonsense" of Boolean algebra disappears.

### BOOLEAN EQUATIONS

Boolean algebra is a mathematical system that defines a series of logical operations performed on a set of variables. The expression of a single logical function is called a Boolean Equation (sometimes referred to as a "Switching Equation"). Boolean equations follow basic rules and use fundamental logical symbols, and they are the foundation of working with digital circuits. Boolean equations include binary variables and Boolean functions.

Binary variables are like variables in regular algebra, except they can only have values of zero or one. Boolean algebra includes three primary logical functions: AND, OR, and NOT; and five secondary logical functions: NAND, NOR, XOR, and XNOR (sometimes called "equivalence"), and Buffer (sometimes called

"transfer"). A Boolean function is simply an electronic circuit that defines some relationship between input and output variables. Therefore, a Boolean equation takes the form of:

$$A * B = C$$

where A and B are binary input variables that are related to the binary output variable C by the function AND (noted by an asterisk).

In Boolean algebra, it is common to speak of "truth." This term does not mean the same as it would to a philosopher, though its use is based on Aristotelian philosophy where a statement was either true or false. In Boolean algebra, "true" commonly means "voltage present" (or "1") while "false" commonly means "voltage absent" (or "0"), and this can be applied to either input or output variables. It is common to create a "Truth Table" for a Boolean equation to indicate which combination of inputs should evaluate to a "true" output; and Truth Tables are used very frequently throughout this course.



---

## 2: BOOLEAN ALGEBRA

### 2.1 INTRODUCTION

Before starting a study of Boolean algebra, it is important to keep in mind that this mathematical system concerns electronic components that are capable of only two states: True and False (sometimes called "High-Low" or "1-0"). The Boolean algebra system is based on evaluating a series of "True-False" statements to determine the final output of a circuit.

For example, a Boolean expression could be created that would describe "IF the floor is dirty OR company is coming THEN I'll mop." (The things I do for visiting company!) These types of logic statements are often represented symbolically using the symbols 1 and 0, where 1 stands for "True" and 0 stands for "False." So, let "Floor is dirty" equal 1 and "Floor is not dirty" equal 0. Also, let "Company is coming" equal 1 and "Company is not coming" equal 0. Then, "Floor is dirty OR company is coming" can be symbolically represented by "1 OR 1". Within the discipline of Boolean algebra, common mathematical symbols are used to represent Boolean expressions; for example, Boolean OR is frequently represented by a mathematics plus sign, as shown below.

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 1 \end{aligned}$$

These look like addition problems, but they are not (as evidenced by the last line). It is essential to keep in mind that these are merely symbolic representations of True-False statements. The first three lines make perfect sense and look like elementary addition. The last line, though, violates the principles of addition for real numbers; but it is a perfectly valid Boolean expression. Remember, in the world of Boolean algebra, there are only two possible values for any quantity: 1 or 0; and that last line is actually saying "True OR True is True." To use the dirty floor example from above, "The floor is dirty" (True) OR "Company is coming" (True) SO "I will mop the floor" (True) is symbolized by:  $1 + 1 = 1$ . This could be expressed as "T OR T SO T"; but the convention is to use common mathematical symbols; thus " $1 + 1 = 1$ ".

Moreover, it does not matter how many or few terms are ORed together; if just one is True, then the output is True, as illustrated below:

$$\begin{aligned} 1 + 1 + 1 &= 1 \\ 1 + 1 + 0 + 1 + 1 &= 1 \\ 1 + 1 + 1 + 1 + 1 + 1 &= 1 \end{aligned}$$

Next, consider a very simple electronic sensor in an automobile: IF the headlights are on AND the driver's door is open THEN a buzzer will sound. In the same way that the plus sign is used to mathematically represent OR, a times sign is used to represent AND. Therefore, using common

mathematical symbols, this automobile alarm circuit would be represented by  $1 \times 1 = 1$ . The following list shows all possible states of the headlights and door:

$$\begin{array}{l} 0 \times 0 = 0 \\ 0 \times 1 = 0 \\ 1 \times 0 = 0 \\ 1 \times 1 = 1 \end{array}$$

The first row above shows "False (the lights are not on) AND False (the door is not open) results in False (do not sound an alarm)". For AND logic, the only time the output is True is when all inputs are also True; therefore:  $1 \times 1 \times 1 \times 1 \times 0 = 0$ . In this way, Boolean AND behaves somewhat like algebraic multiplication.

Within Boolean algebra's simple True-False system, there is no equivalent for subtraction or division, so those mathematical symbols are not used. Like real-number algebra, Boolean algebra uses alphabetical letters to denote variables; however, Boolean variables are always CAPITAL letters, never lower-case. Thus, a Boolean equation would look something like this:

$$A + B = Q$$

As Boolean expressions are realized (that is, turned into a real, or physical, circuit), the various operators become "gates." For example, the above equation would be realized using an OR gate with two inputs (labeled A and B) and one output (labeled Q).

## 2.2: PRIMARY LOGIC FUNCTIONS

### AND

An AND gate is a Boolean function that will output a logical one, or true, only if all of the inputs are true. As an example, consider this statement: "If I have 10 bucks AND there is a good movie at the cinema, then I will go see the movie." In this statement, "If I have 10 bucks" is one variable and "there is a good movie at the cinema" is another variable. If both of these inputs are true, then the output variable ("I will go see the movie") will also be true. However, if either of the two inputs is false, then the output will also be false (or, "I will not go see the movie"). Of course, if I want popcorn, I would need another ten spot, but that is not germane to this example. When written in an equation, the Boolean AND term is represented a number of different ways, depending on the author's desire. One method is to use the logic "AND" symbol:

$$A \wedge B = Q$$

One other method is to use the same symbols that are used for multiplication in traditional algebra; that is, by writing the variables next to each other, with parenthesis, or, sometimes, with an asterisk between them, like these examples:

$$AB = Q$$

$$(A)(B) = Q$$

$$A * B = Q$$

It is also possible to use other mathematical multiplication symbols, such as X or • (dot); however, these symbols are not very commonly used in digital logic equations. Logic AND is normally represented in equations by using an algebra multiplication symbol (since it is easy to type); however, if there is any chance for ambiguity, then the Logic AND symbol can be used to differentiate between multiplication and a logic AND function.

Following is the truth table for an AND gate.

Inputs		Output
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 1: TRUTH TABLE FOR AND GATE

All possible input combinations are normally found by counting in binary starting with all variables having a 0 value to all variables having a 1 value. Thus, in the above table, the inputs are 00, 01, 10, 11. Notice for the AND gate truth table that the output is false (that is, 0) until the last row: when both inputs are true then the output is true. Therefore, it could be said that one False input would turn off an AND Gate.



Because a single False input can turn an AND gate off, these types of gates are frequently used as a switch in a logic circuit. As a simple example, imagine an assembly line where there are four different safety sensors of some sort. The sensor outputs could be routed to a single 4-input AND gate; and then as long as all sensors are True, the assembly line motor will run. If, however, any one of those sensors begins to output a False due to some unsafe condition, then the AND gate would also output a False and the motor would stop.

Logic gates are realized (or created) in electronic circuits by using many transistors, resistors, and other components. These components are normally packaged into a single integrated circuit (IC) "chip," so the logic circuit designer does not need to know all of the details of the electronics in order to use the gate. In logic diagrams, gates are usually represented by various distinctive shapes. For example, an AND gate is represented by a shape that looks like a capital D. In the following diagram, the input variables A and B are wired to an AND gate, and the output from that gate goes to output Y.

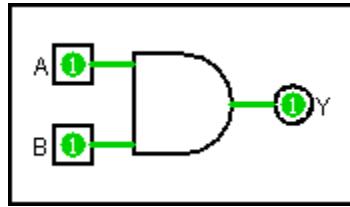


FIGURE 1: AND GATE

Notice that each input and output is named in order to make it easier to describe the circuit algebraically. As circuits become more complex, a simple logic gate may be replaced with an Integrated Circuit (IC), something like this:

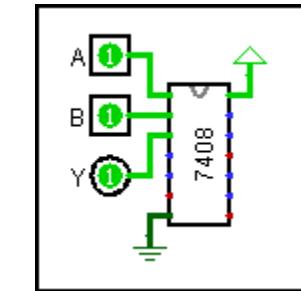


FIGURE 2: 74X08 QUAD AND GATE LOGIC SYMBOL

The above Integrated Circuit (IC) is a 74x08 Quad 2-input AND Gate, which means that there are 4 separate 2-input AND gates on one IC. Notice that only one of the four available AND gates is used in the illustration; and the inputs have been named A and B, with the output named Y. The pins for an IC are numbered from the top left counterclockwise to the top right. Pins 1 and 2 are the inputs for the first AND gate; and pin 3 is that gate's output. In the same way, pins 4, 5, and 6 are for the second AND gate. Pin 7, in the lower left corner, is the Ground pin. While Logisim does not require that pin to be connected, it is in this circuit for realism. Pins 8, 9, and 10 are for the third AND gate; and pins 11, 12, and 13 are for the forth AND gate. Pin 14, on the top right corner of the IC, is for power. Again, it has

been connected to a power source (represented by a small triangle) for a more realistic circuit, though Logisim does not require that connection.

There are two common sets of symbols used to represent the various elements in logic diagrams, and whichever is used is of little consequence since the logic is the same. Shaped symbols, as used above, are more common; but an AND gate can also be represented with the following symbol from the Institute of Electrical and Electronics Engineers (IEEE):

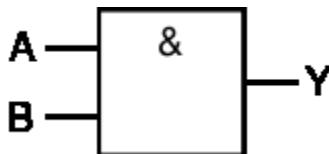


FIGURE 3: 2-INPUT AND GATE USING AN IEEE SYMBOL

As an example of an AND gate at work, consider an elevator: if the door is closed (logic 1) AND someone in the elevator presses a floor button (logic 1), THEN the elevator will move (logic 1). If both sensors (door and button) are input to an AND gate, then the elevator motor will not operate if either the door is open or no one has pressed a floor button.

## OR

An OR gate is a Boolean function that will output a logical one, or True, if any of the inputs are true. As an example, consider this statement: "If my dog needs a bath OR I am going swimming, then I will put on a bathing suit." In this statement, "if my dog needs a bath" is one input variable and "I am going swimming" is another input variable. If either of these is true, then the output variable, "I will put on a bathing suit," will also be true. However, if both of the inputs are false, then the output will also be false (or, "I will not put on a bathing suit"). If you think it odd that I would wear a bathing suit to bathe my dog, then you have obviously never attempted to complete that chore.

When written in an equation, the Boolean OR term is represented a number of different ways, depending on the author's desire. One method is to use the logic "OR" symbol:

$$A \vee B = Q$$

One other method is to use the "plus" sign that is used for addition in traditional algebra, like this example:

$$A + B = Q$$

For simplicity, the mathematical "plus" symbol is normally be used to indicate OR since that is easy to enter on a keyboard; however, if there is any chance for ambiguity, then the logic OR symbol is used to differentiate between addition and logic OR.

Following is the truth table for an OR gate.

Inputs		Output
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

FIGURE 4: TRUTH TABLE FOR OR GATE

Notice for the OR truth table that the output is true ("1") whenever at least one input is true. Therefore, it could be said that one True input would turn on an OR Gate. In the following diagram, the input variables A and B are wired to an OR gate, and the output from that gate goes to Y.

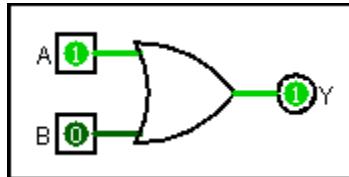


FIGURE 5: SIMPLE 2-INPUT OR GATE.

As an example of an OR gate at work, consider a traffic signal. Suppose an intersection is set up such that the main road is normally green; however, if a car pulls up on the crossroad, or if a pedestrian presses the "cross" button, then the main light is changed to red to stop traffic. This could be done with a simple OR gate. An automobile sensor on the crossroad would be one input and the pedestrian "cross" button would be the other input; the output of the OR gate connecting these two inputs would change the light to red.

## NOT

A NOT gate (or "inverter") is a Boolean function that inverts the input. That is, if the input is true, then the output will be false; or if the input is false, then the output will be true. When written in an equation, the Boolean NOT function is represented in many ways, though two are most popular. The older method is to use an overbar (that is, a line above) on a term, or group of terms, that are to be inverted in an equation:

$$A + \overline{B} = Q$$

The above is read "A OR B NOT = Q" (notice that when spoken, the word "not" follows the term that is inverted). A newer method of indicating NOT is to use the algebra "prime" character, a single quote. Thus, the above equation could be written like this:

$$A + B' = Q$$

The reason the second method of writing the NOT symbol is popular is because it is easier to type on a computer keyboard. There are many other ways some authors use to represent NOT in a formula, but none are considered standardized. For example, some authors use an exclamation point for NOT:  $A + !B = Q$ , others use a broken line:  $A + \neg B = Q$ , others use a backslash:  $A + \backslash B = Q$ , and still others use a tilde:  $A + \tilde{B} = Q$ .

$A + \sim B = Q$ . However, only the prime symbol and the overbar are consistently used to indicate NOT. Following is the truth table for a NOT gate:

Input	Output
0	1
1	0

FIGURE 6: TRUTH TABLE FOR NOT GATE

In a logic diagram, a NOT gate is represented by a small triangle with a circle. It is assumed that the input is on the flat part of the triangle and the output is from the circle. In the following diagram, the input variable A ("0") is inverted by a NOT gate and then sent to output Y ("1").

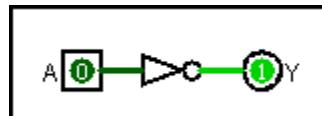


FIGURE 7: INVERTER LOGIC DIAGRAM.

## 2.3: LAB – INTRODUCTION TO LOGISIM

### PURPOSE

This lab introduces the Logisim Logic Simulator, which is used for all other lab exercises.

### INSTALLING

Logisim (<http://ozark.hendrix.edu/~burch/logisim/>) is available as a free download. It is a Java application, so Java will need to be installed before using Logisim. After clicking on the Logisim icon, it will automatically start Java and then the simulator. The program will not need to be uninstalled since it is not actually installed; the Logisim file can simply be deleted when it is no longer needed.

### BEGINNER'S TUTORIAL

Logisim 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.

### LOGISIM WORKSPACE

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

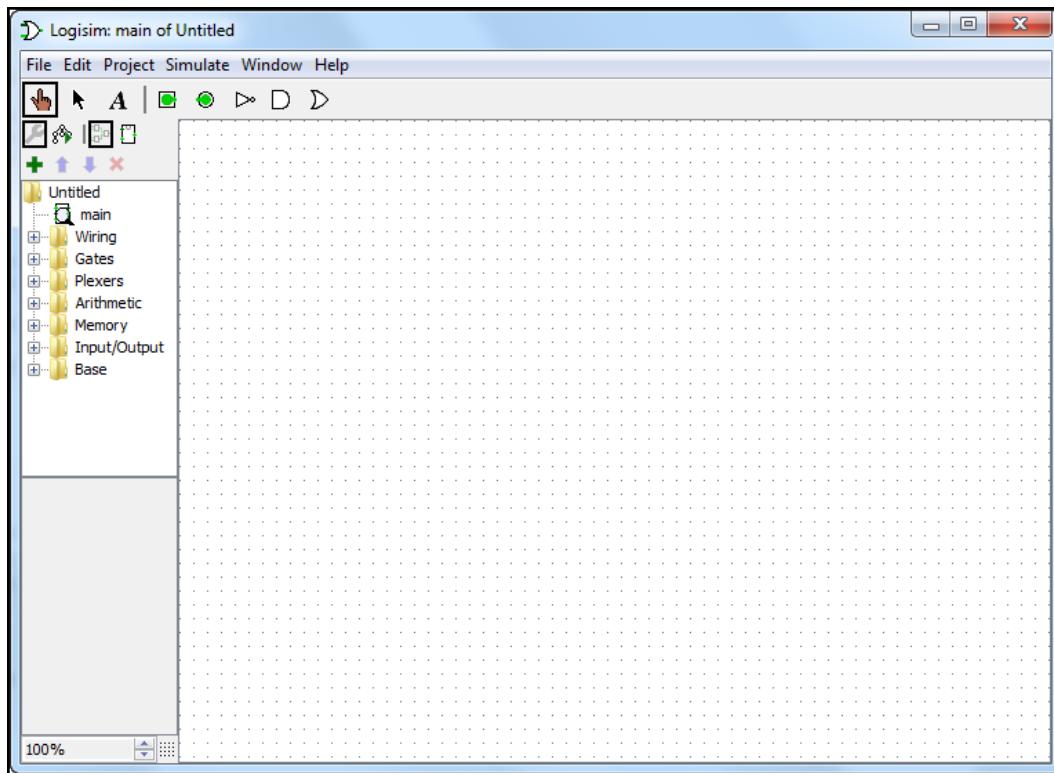


FIGURE 8: LOGISIM WORKSPACE

The Logisim 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 and is very useful while wiring components in new circuits.

Under the menu is the Toolbar, which is a row of eight icons that are the most commonly used tools in Logisim:

- **Pointing Finger:** Used to "poke" and change inputs 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 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:

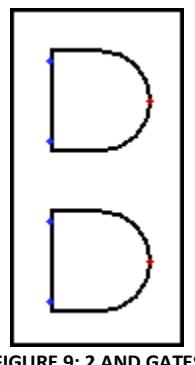


FIGURE 9: 2 AND GATES

Select each AND gate and set the "Number of Inputs" attribute in the attribute pane (lower left corner) to two. That will reduce the number of AND gate inputs from five to two. The other default attribute values do not need to be changed for this circuit.

Selection: AND Gate	
Facing	East
Data Bits	1
Gate Size	Medium
Number Of Inputs	2
Output Value	0/1
Label	
Label Font	SansSerif Plain 12
Negate 1 (Top)	No
Negate 2 (Bottom)	No

FIGURE 10: AND GATE ATTRIBUTES

The two AND gates need to be combined with an OR gate. Add an OR gate and set its "Number of Inputs" attribute to two:

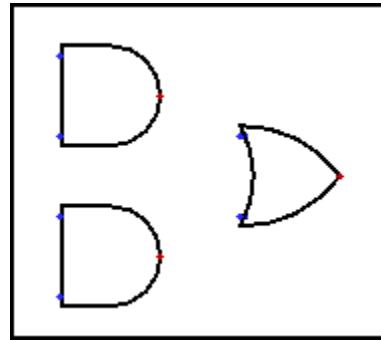


FIGURE 11: OR GATE ADDED

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

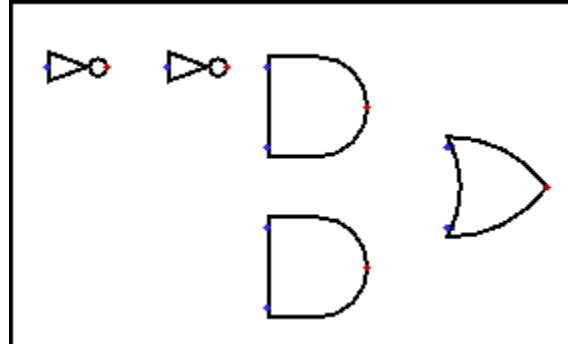


FIGURE 12: 2 NOT GATES ADDED

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.

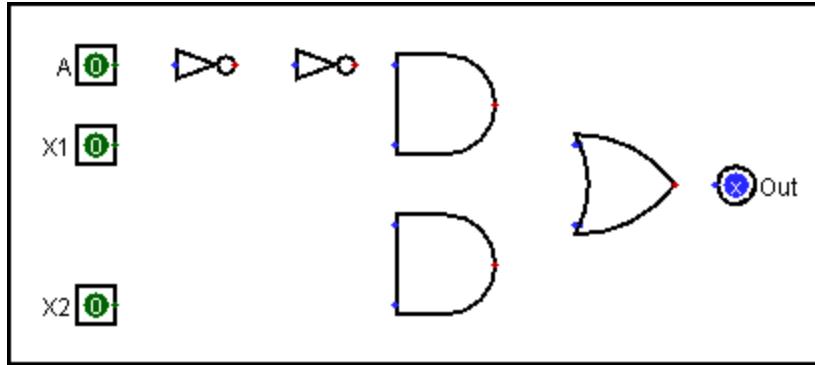


FIGURE 13: INPUTS AND OUTPUTS ADDED

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

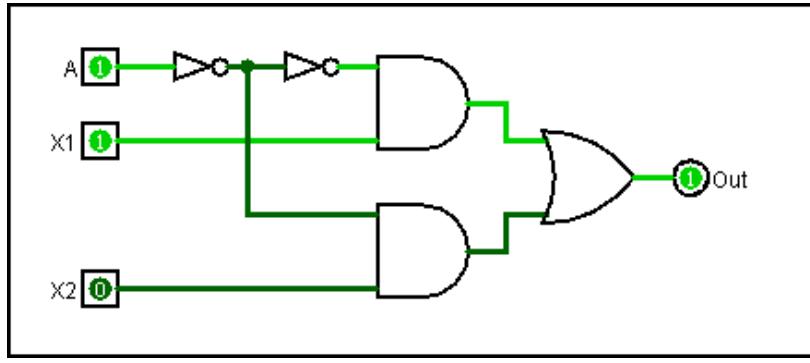


FIGURE 14: INPUTS AND OUTPUTS ADDED

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 X1 should be transmitted to the output; but when "A" is low then 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 2.3 - Introduction to Logisim  
May 4, 2013

Note that Logisim will automatically center text in a box, so all text boxes will need to be aligned.

#### CLEANUP

Be sure the standard identifying information is at the top left of the circuit and then save the file with this name: "Lab 2\_3 - Intro".

## 2.4: SECONDARY LOGIC FUNCTIONS

### NAND

A NAND gate is a Boolean function that is the opposite of an AND gate ("NOT AND"). It will output a logical zero, or false, only if all of the inputs are true.

Note: As a matter of interest, NAND and NOR gates are said to be universal. That is, given enough of either of these gates, the operation of any other gate or function (even complete complex circuits) can be mimicked. For example, it is possible to build an OR circuit using three interconnected NAND gates. In fact, some logic systems have been designed around nothing but either NAND or NOR gates; with all the necessary logic functions being derived from interconnected collections of these gates.

Following is the truth table for a NAND gate.

Inputs		Output
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

TABLE 2: TRUTH TABLE FOR NAND GATE

In the following diagram, the input variables A and B are wired to a NAND gate, and the output from that gate goes to output Y.

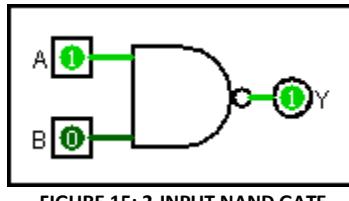


FIGURE 15: 2-INPUT NAND GATE

Tip: The logic diagram symbol for a NAND gate looks like an AND gate, but with a small circle on the output pin. A circle (usually called a "bubble") in a logic diagram always represents some sort of signal inversion, and it can appear at the inputs or outputs of nearly any logic gate. For example, the bubble on a NAND gate could be interpreted as "take whatever the output is for the AND gate, and then invert it." As a final note, bubbles are never found by themselves on a wire; they are always associated with a logic gate. To invert a signal on a wire, a NOT gate is used.

### NOR

A NOR gate is a Boolean function that is the opposite of an OR gate ("NOT OR"). It will output a logical one, or true, only if all of the inputs are false.

Following is the truth table for a NOR gate.

Inputs		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

TABLE 3: TRUTH TABLE FOR NOR GATE

In the following diagram, the input variables A and B are wired to a NOR gate, and the output from that gate goes to output Y.

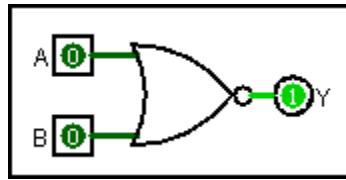


FIGURE 16: 2-INPUT NOR GATE

### XOR

An XOR ("Exclusive OR") gate is a Boolean function that will output a logical one, or true, only if the two inputs are different. This is useful for circuits that compare inputs; if the inputs are different, then the output will be true, otherwise it is false.

The XOR function is not often used in Boolean equations; but when necessary, it is represented by a plus sign (like the OR function) inside a circle. Here is an example:

$$A \oplus B = Q$$

Following is the truth table for an XOR gate.

Inputs		Output
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 4: TRUTH TABLE FOR XOR GATE

In the following diagram, the input variables A and B are wired to a XOR gate, and the output from that gate goes to output Y.

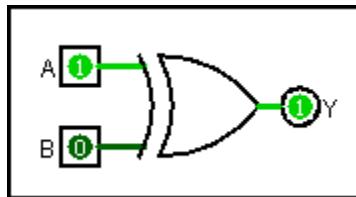


FIGURE 17: 2-INPUT XOR GATE



There is some debate about the proper behavior of an XOR gate that has more than two inputs. Some experts believe that an XOR gate should output a True if one, and only one, input is true. This would seem to be in keeping with the rules of digital logic developed by George Boole and other early logicians; and is the strict definition of XOR promulgated by the IEEE. Others believe, though, that an XOR gate should output a True if an odd number of inputs is True. This is more intuitive and is the behavior of a group of cascaded XOR gates; also, this type of XOR gate has a much greater practical application in circuits.

## XNOR

An XNOR gate (sometimes called an "equivalence" gate) is a Boolean function that will output a logical one, or true, only if the two inputs are the same.

Following is the truth table for an XNOR gate.

Inputs		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

TABLE 5: TRUTH TABLE FOR XNOR GATE

In the following diagram, the input variables A and B are wired to a XNOR gate, and the output from that gate goes to output Y.

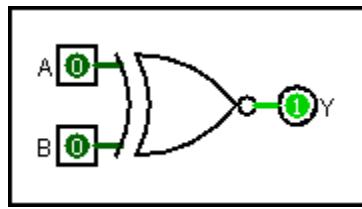


FIGURE 18: 2-INPUT XNOR GATE

## BUFFER

A buffer (sometimes called transfer) is a Boolean function that will transfer the input to the output without change. If the input is true, then the output will be true; if the input is false, then the output will be false. It may seem to be an odd function since the gate does not change anything; but it has an important use in a circuit. As logic circuits become more complex, the signal from input to output may become weak and no longer able to drive (or activate) additional gates. A buffer is used to boost (and stabilize) a logic level so it is more dependable. Cleaning up a signal is one other important function done by a buffer. When an electronic circuit interacts with the physical world (for example, when a user pushes a button), there is often a very brief period when the signal from that physical device waivers between high and low unpredictably. A buffer can smooth out that signal so it is a constant high or low

without a lot of spikes in between. There is no symbol to indicate a buffer in a Boolean equation since it does nothing to change the input.

Following is the truth table for buffer.

Input	Output
0	0
1	1

TABLE 6: TRUTH TABLE FOR A BUFFER

In the following diagram, the input variable A is transferred to output Y by a buffer. Notice that a buffer looks like a NOT gate, but lacks the bubble on the output.

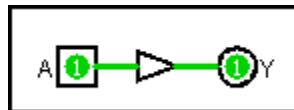


FIGURE 19: BUFFER

## 2.5: LAB – LOGISIM SUBCIRCUITS

### PURPOSE

In this lab a main circuit and one subcircuit are linked in order to further exercise the Logisim 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 "1" since it will be used to input a "1" into the circuit (like the number 1 on a calculator).

Next, add four more buttons and label them 2-5:

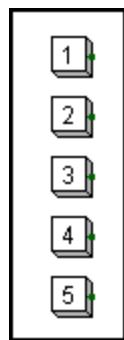


FIGURE 20: 5 BUTTONS

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 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 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.

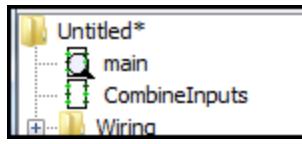


FIGURE 21: EXPLORER PANE

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 *In1*, *In2*, *In3*, *In4*, and *In5* (for "Input Pin 1-5").

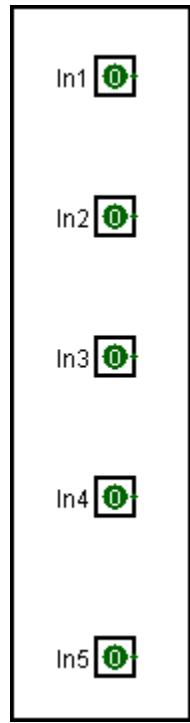


FIGURE 22: FIVE INPUTS

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.

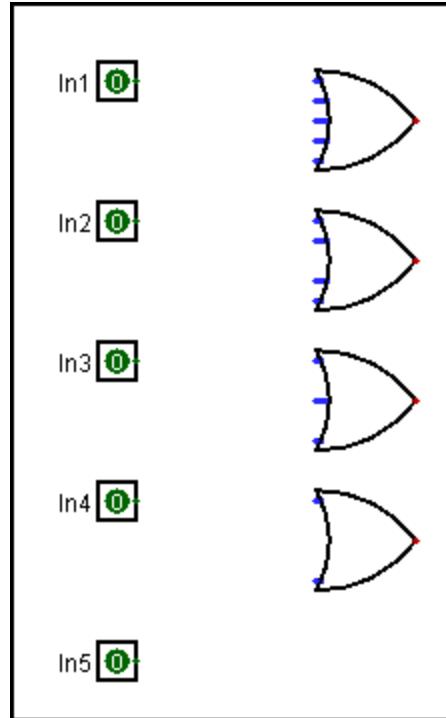


FIGURE 23: OR GATES ADDED

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

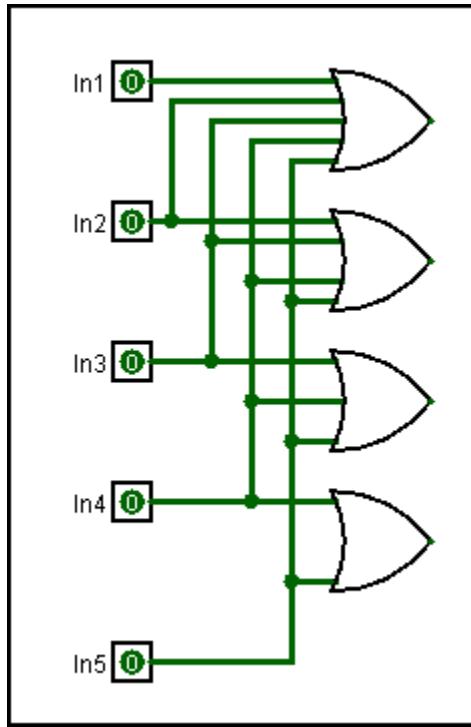


FIGURE 24: OR GATES WIRED

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*.

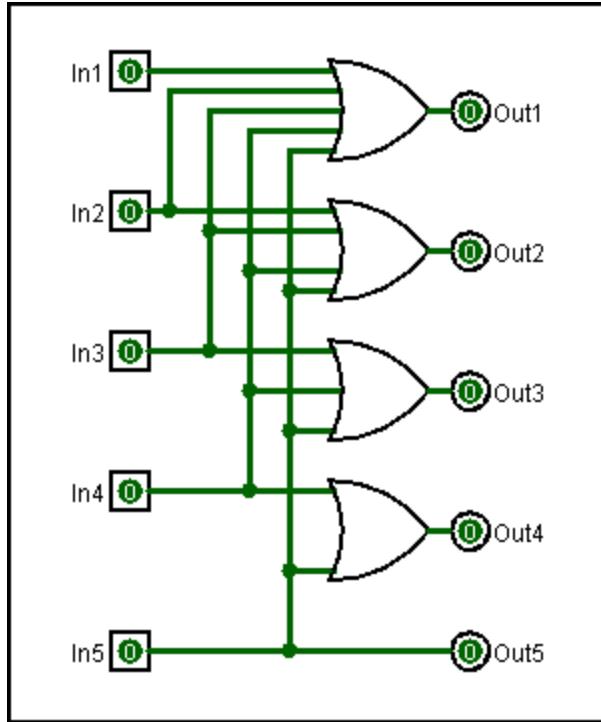


FIGURE 25: COMPLETED SUBCIRCUIT

**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 look like this:

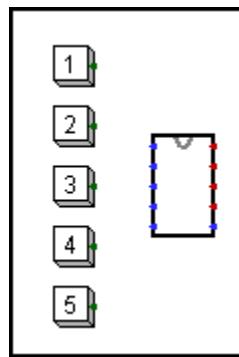


FIGURE 26: SUBCIRCUIT ADDED TO MAIN CIRCUIT

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:

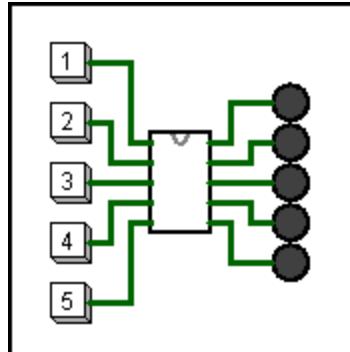


FIGURE 27: COMPLETED LAB 2

#### 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 2.5: Subcircuits", and today's date. Save the file as "Lab 2\_5 – SubCircuits."

## 2.6: LAB – LOGISIM CIRCUIT ANALYZER

### PURPOSE

In this lab the Logisim circuit analyzer will be presented.

### PROCEDURE

Open Logisim and start a new project.

Logisim includes a "circuit analyzer" that helps simplify and build circuits. Suppose a circuit is needed to satisfy this Boolean expression:

$$\sum(a,b,c) = (a'b'c') + (abc') + (abc)$$

Click *Project -> Analyze Circuit*. On the *Input* tab, enter a, b, and c as separate inputs.

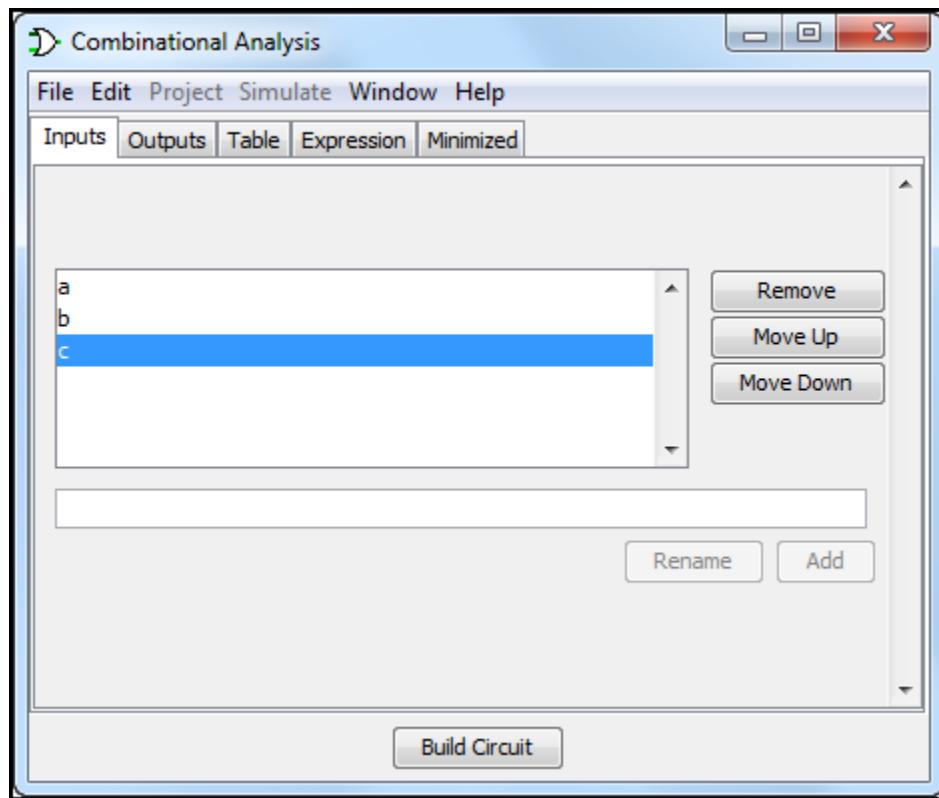


FIGURE 28: COMBINATIONAL ANALYSIS INPUTS

On the *Output* tab, enter X as the only output. (Since no specific output variable was found in the Boolean expression, X is as good as any).

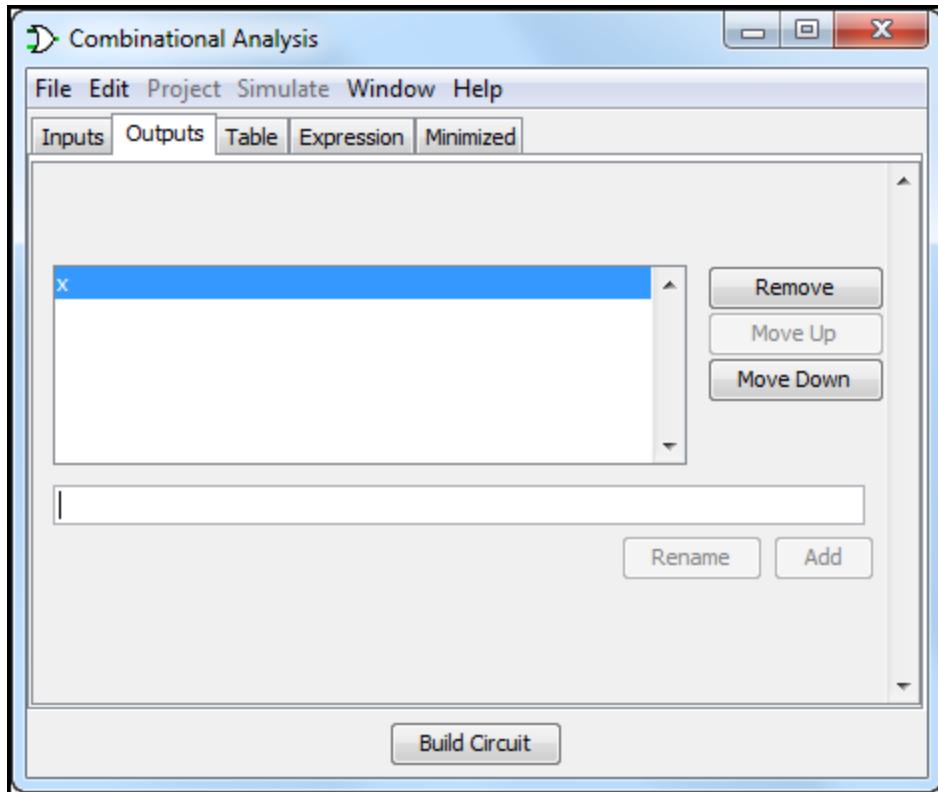


FIGURE 29: COMBINATIONAL ANALYSIS OUTPUTS

On the *Expression* tab, enter the canonical form of the equation. Logisim uses a tilde ("~") before an expression to mean "NOT." Terms ANDed together must be separated with a space.

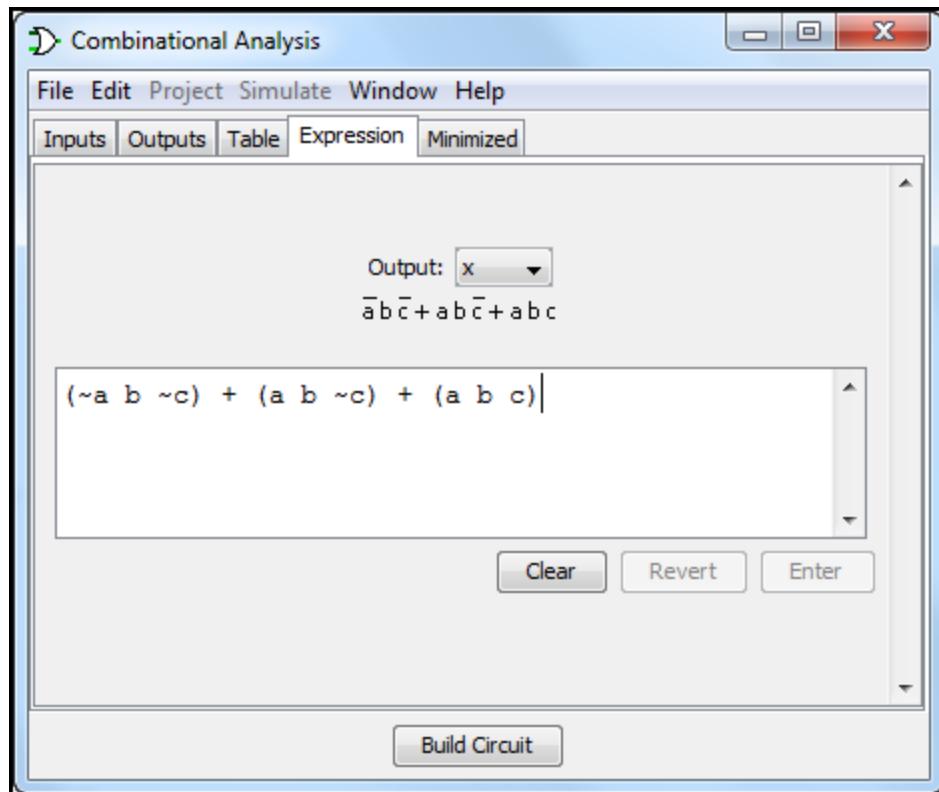


FIGURE 30: COMBINATIONAL ANALYSIS EXPRESSION

The truth table is now automatically generated, based upon the Boolean expression. If a truth table had been available with no Boolean expression, that could have been filled it in first and the expression would have been generated.

The screenshot shows the 'Combinational Analysis' window in Logisim. The menu bar includes File, Edit, Project, Simulate, Window, and Help. Below the menu is a tab bar with Inputs, Outputs, Table, Expression, and Minimized, where 'Table' is selected. The main area displays a truth table:

a	b	c	x
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

At the bottom center is a 'Build Circuit' button.

FIGURE 31: COMBINATIONAL ANALYSIS TRUTH TABLE

To build this circuit, click the "Build Circuit" button. Enter "main" for the Circuit Name and do not check either of the two option boxes. Click OK and permit LogiSim to replace the "main" circuit. The completed circuit should look like this:

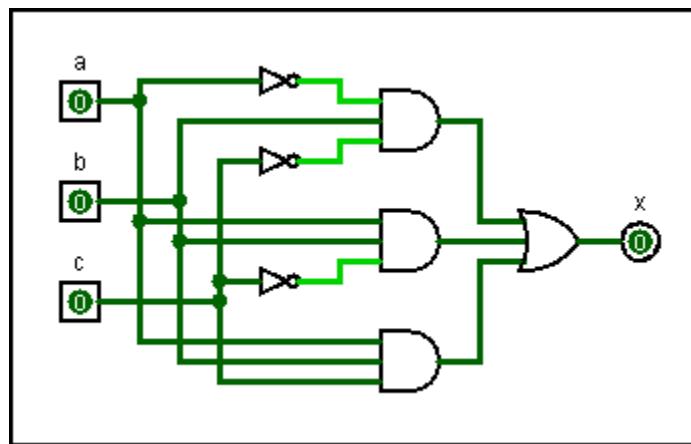


FIGURE 32: COMBINATIONAL ANALYSIS CIRCUIT 1

Open the analyzer again (*Project -> Analyze Circuit*). The circuit will automatically be minimized using a Karnaugh map. Click the Minimized tab to see it.

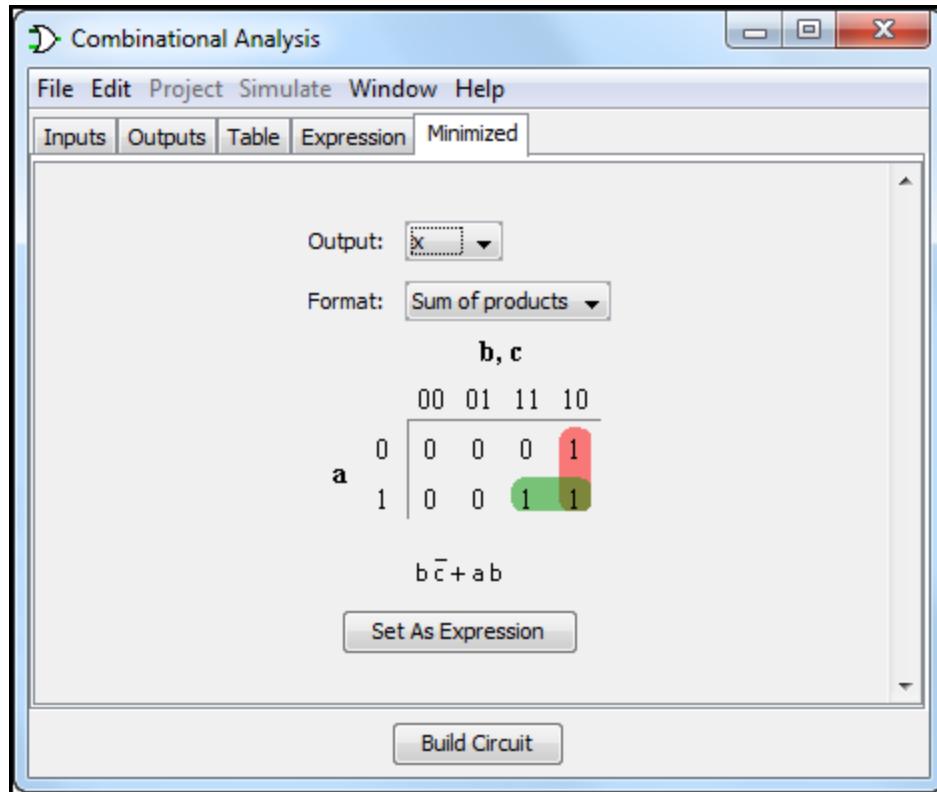


FIGURE 33: COMBINATIONAL ANALYSIS MINIMIZED

Click the *Set As Expression* button to change the original Boolean expression to its minimized form. Logisim will build the circuit that is set as the current expression when you click the *Build Circuit* button. To build the minimized circuit, click the *Build Circuit* button. Enter "Simplified" for the Circuit Name and do not check either of the two option boxes. Click OK. The completed circuit should look like this:

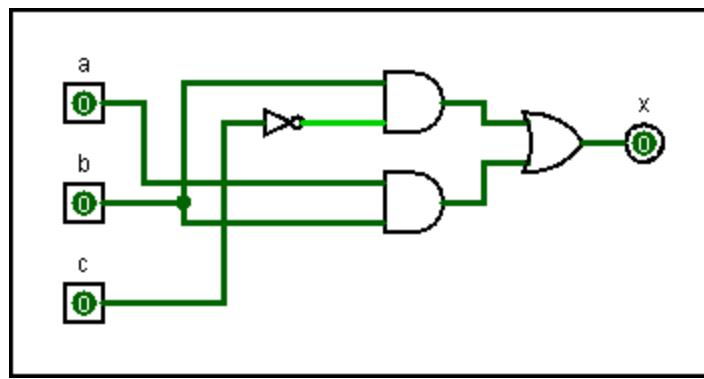


FIGURE 34: COMBINATIONAL ANALYSIS CIRCUIT 2

Do not hesitate to use the Circuit Analyzer tool in Logisim; it will save both time and trouble.

#### CHALLENGE

Create a sub-circuit named *Challenge*. Use the Circuit Analyzer to create the simplified version of the circuit described by this Boolean expression as the *Challenge* sub-circuit:

$$\sum(a,b,c) = (a'b'c') + (a'b'c) + (a'bc') + (abc)$$

**CLEANUP**

Rename the "main" circuit as "Analyzer." Be sure the standard identifying information block is at the top left of the "Analyzer" circuit: Name, "Lab 2.6: Circuit Analyzer", today's date. Save the file as "Lab 2\_6 – Circuit Analyzer."

## 2.7: UNIVARIATE BOOLEAN ALGEBRA PROPERTIES

### INTRODUCTION

Boolean Algebra, like real number algebra, includes several properties. This unit introduces the univariate Boolean properties, or those properties that involve only one input variable. These properties permit Boolean expressions to be simplified, and circuit designers are interested in simplifying circuits to reduce construction expense, power consumption, heat loss (wasted energy), and troubleshooting time.

### IDENTITY

In mathematics, an identity is an expression that is true for all possible values of its variables; that is, the answer to a problem stays the same regardless of how an input variable changes. As an example, the algebraic identity of  $x + 0 = x$  tells us that anything ( $x$ ) added to zero equals the original "anything," no matter what value that "anything" ( $x$ ) may be.

Combining any logic input and a logic zero with an OR gate yields the original input. Because OR is represented by a plus sign when written in a Boolean equation, and the identity element for OR is zero, the following equation is true:

$$A + 0 = A \quad (0.1)$$

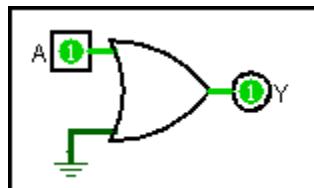


FIGURE 35: IDENTITY ELEMENT FOR OR GATE

The symbol with three small lines represents ground and is always considered to be a logic 0, or False. The output for this circuit will be the same as the input no matter whether that input is True or False (1 or 0). When  $A=1$ , the output will also be 1, and when  $A=0$ , the output will also be 0. Therefore, the identity element for OR is 0.

Combining anything and a logic one with an AND gate yields the original "anything." Because AND is represented by a multiplication sign when written in a Boolean equation, and the identity element for AND is one, the following equation is true:

$$A * 1 = A \quad (0.2)$$

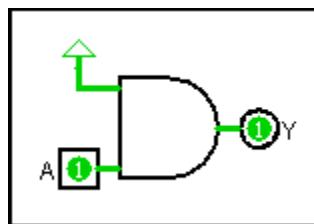


FIGURE 36: IDENTITY ELEMENT FOR AND GATE

The triangle symbol represents the circuit's power supply and is always considered to be logic 1, or True. The output for this circuit will be the same as the input no matter whether that input is High or Low (1 or 0). When A=1, the output will also be one; and when A=0, the output will also be zero. Therefore, the identity element for AND is one.

#### IDEMPOTENCE

If the two inputs of either an OR or AND gate are tied together, then the same signal will be applied to both inputs. This results in the output of either of those gates being the same as the input; and this is called the idempotence property. An electronic gate wired in this manner performs the same function as a buffer.

$$A + A = A \quad (0.3)$$

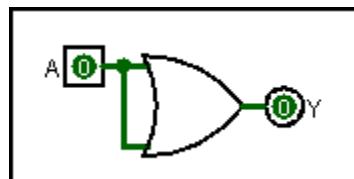


FIGURE 37: IDEMPOTENCE PROPERTY FOR OR

Tip: Be careful with this; it is different from real-number algebra, where the sum of two identical variables is twice the original variable's value ( $x + x = 2x$ ).

Following is the idempotence property for AND using both a mathematical equation and logic diagram:

$$A * A = A \quad (0.4)$$

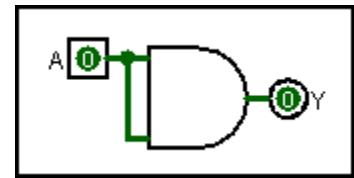


FIGURE 38: IDEMPOTENCE PROPERTY FOR AND

Tip: Be careful with this; it is different from real-number algebra, where the product of two identical variables is the square of the original variable's value ( $x * x = x^2$ ).

#### ANNIHILATOR

Combining any data and a logic one with an OR gate yields a constant output of one. This property is called the annihilator since the OR gate outputs a constant one; in other words, whatever other data were input are lost. Because OR is represented by a plus sign when written in a Boolean equation, and the annihilator for OR is one, the following equation is true:

$$A + 1 = 1 \quad (0.5)$$

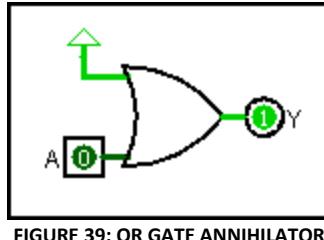


FIGURE 39: OR GATE ANNIHILATOR

The triangle symbol represents the circuit's power supply and is always considered to be a logic 1, or True. The output for this circuit will be True (or 1) no matter whether input A is True or False (1 or 0).

Combining any data and a logic zero with an AND gate yields a constant output of zero. This property is called the annihilator since the AND gate outputs a constant zero; in other words, whatever other data were input are lost. Because AND is represented by a multiplication sign when written in a Boolean equation, and the annihilator for AND is zero, the following equation is true:

$$A * 0 = 0 \quad (0.6)$$

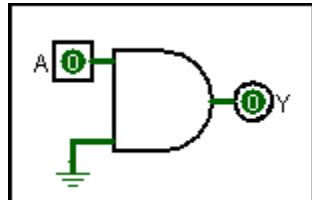


FIGURE 40: AND GATE ANNIHILATOR

The symbol with three small lines represents ground and is always considered to be a logic 0, or False. The output for this circuit will be False (or 0) no matter whether input A is True or False (1 or 0).

#### COMPLEMENT

In Boolean logic there are only two possible values for variables: 0 and 1. Since either a variable or its complement must be one, and since combining any data with one through an OR gate yields one (see the annihilation property), then the following equation is true:

$$A + A' = 1 \quad (0.7)$$

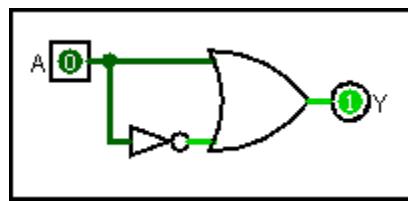


FIGURE 41: COMPLEMENT PROPERTY FOR OR

In the above circuit, the output (Y) will always equal 1, regardless of the value of input A.

In Boolean logic there are only two possible values for variables: 0 and 1. Since either a variable or its complement must be zero, and since combining any data with zero through an AND gate yields zero (see the annihilation law), then the following equation is true:

$$A * A' = 0 \quad (0.8)$$

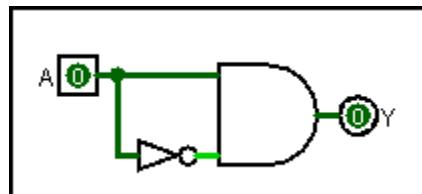


FIGURE 42: COMPLEMENT PROPERTY FOR AND

In the above circuit, the output (Y) will always equal 0, regardless of the value of input A.

#### INVOLUTION

Another law having to do with complementation is that of involution, also called a double complement. Complementing a Boolean variable two times (or any even number of times) results in the original Boolean value. Written mathematically:

$$(A')' = A \quad (0.9)$$

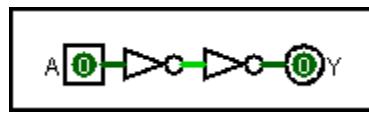


FIGURE 43: INVOLUTION PROPERTY

In the above circuit, the output (Y) will be the same as the input (A).

Aside: It takes the two NOT gates a short period of time to pass a signal from input to output; and this is known as "propagation delay." A designer occasionally needs to build an intentional signal delay into a circuit for some reason, and two (or any greater even number) NOT gates would be one option.

#### MiniLab 2.7

Select any one of the properties presented in this unit and build the circuit that is used to illustrate the property. After building the circuit, notice how the output matches what was predicted by the property. Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 2.7", and today's date. Then save the file with this name: "minilab\_2\_7".

## 2.8 MULTIVARIATE PROPERTIES

### INTRODUCTION

Boolean Algebra, like real number algebra, includes several properties. This unit introduces the multivariate Boolean properties, or those properties that involve more than one input variable. These properties permit Boolean expressions to be simplified, and circuit designers are interested in simplifying circuits to reduce construction expense, power consumption, heat loss (wasted energy), and troubleshooting time.

### COMMUTATIVE

In essence, the commutative property indicates that the order of the input variables can be reversed in either OR or AND gates without changing the truth of the expression. The following expresses this property algebraically:

$$\begin{aligned} A + B &= B + A \\ AB &= BA \end{aligned} \tag{0.10}$$

In the following circuits, the inputs are reversed for the two gates, but the outputs are the same. For example, A is entering the top input for the upper OR gate, but the bottom input for the lower gate; however, Y1 is equal to Y2.

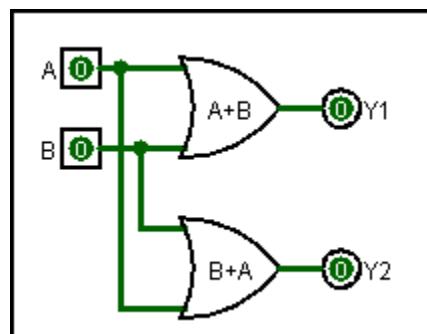


FIGURE 44: COMMUTATIVE PROPERTY FOR OR

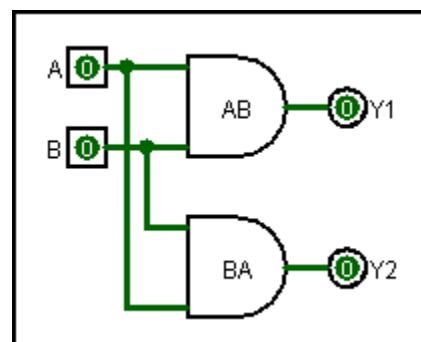


FIGURE 45: COMMUTATIVE PROPERTY FOR AND

## ASSOCIATIVE

This property indicates that groups of variables in an OR and AND gate can be associated in various ways without altering the truth of the equations. The following expresses this property algebraically:

$$\begin{aligned}(A+B)+C &= A+(B+C) \\ (AB)C &= A(BC)\end{aligned}\quad (0.11)$$

In the following circuits, notice that A and B are associated together in the first gate, and then C is associated with the output of that gate. Then, in the lower half of the circuit, B and C are associated together in the first gate, and then A is associated with the output of that gate. Since Y1 is equal to Y2 for any combination of inputs, it does not matter which of the two variables are associated together in a group of gates.

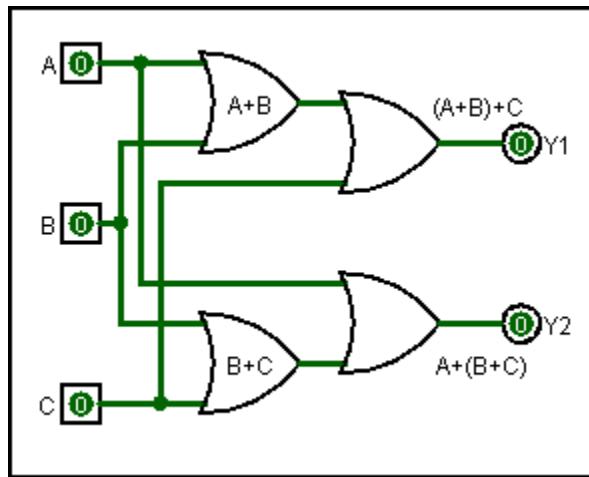


FIGURE 46: ASSOCIATIVE PROPERTY FOR OR

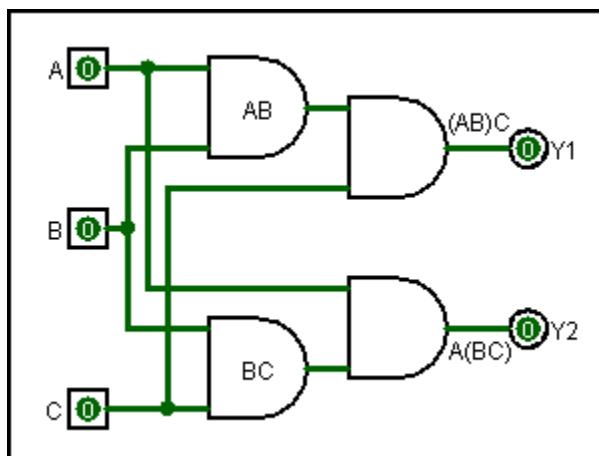


FIGURE 47: ASSOCIATIVE PROPERTY FOR AND

**DISTRIBUTIVE**

The distributive property of real number algebra permits certain variables to be "distributed" to other variables. This operation is frequently used to create groups of variables that can be easily simplified; thus, simplifying the entire expression. Boolean algebra also includes a distributive property, and that can be used to combine AND and OR gates in various ways that make it easier to simplify the circuit. The following expresses this property algebraically:

$$\begin{aligned} A(B+C) &= AB + AC \\ A + (BC) &= (A+B)(A+C) \end{aligned} \quad (0.12)$$

In the following circuits, notice that input A in the top circuit is distributed to inputs B and C in the bottom circuit. However, output Y1 is always equal to output Y2 regardless of how the inputs are set. These circuits show distributive of AND over OR and distributive of OR over AND.

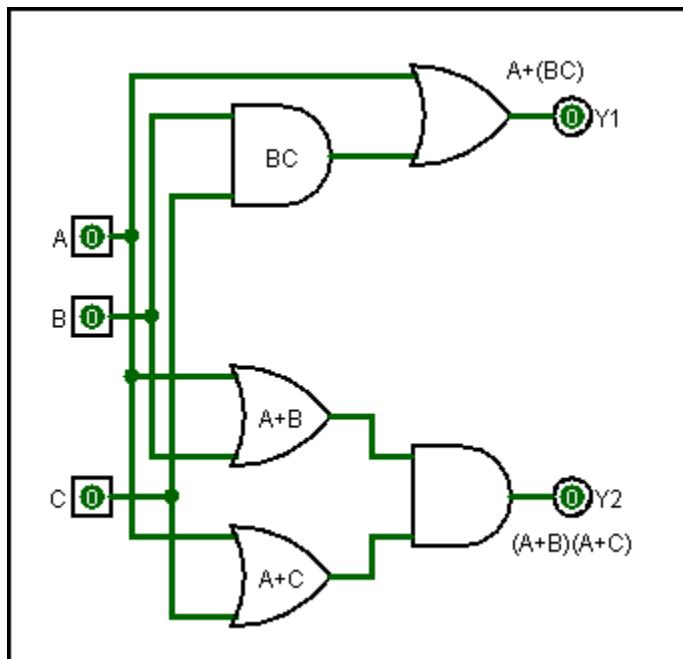


FIGURE 48: DISTRIBUTIVE PROPERTY OF AND OVER OR

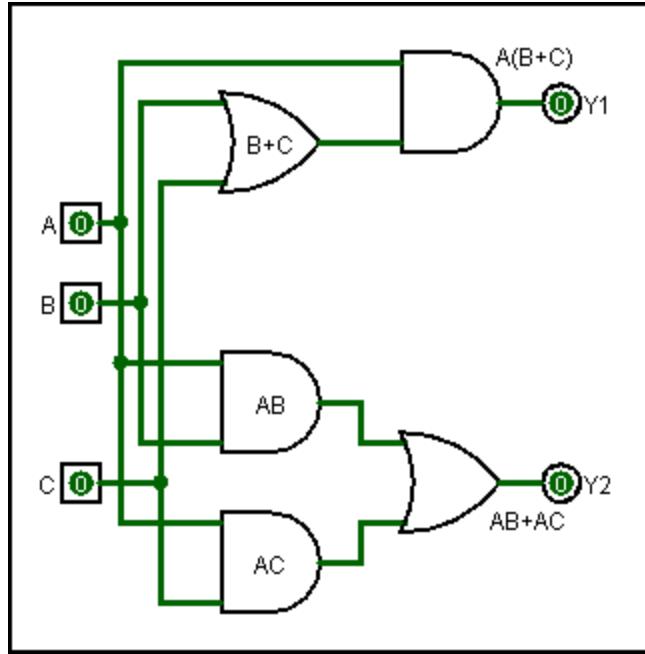


FIGURE 49: DISTRIBUTIVE PROPERTY OF OR OVER AND

### ABSORPTION

The absorption property is used to remove logic gates from a circuit if those gates have no effect on the output. In essence, a gate is "absorbed" if it is not needed. There are two different absorption properties:

$$\begin{aligned} A + AB &= A \\ A(A + B) &= A \end{aligned} \tag{0.13}$$

The best way to think about why these properties are true is to imagine a circuit that contains them. The first circuit below illustrates the top equation.

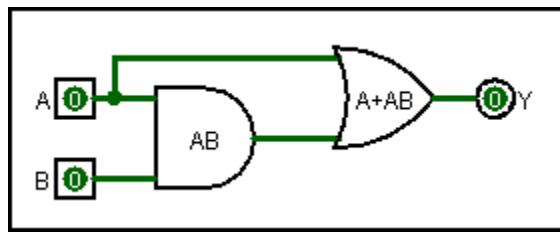


FIGURE 50: ABSORPTION PROPERTY (VERSION #1)

This is the truth table for that circuit:

Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	1
1	1	1

TABLE 7: ABSORPTION CIRCUIT TRUTH TABLE

Notice that the output, Y, is always the same as input A. This means that input B has no bearing on the output of the circuit; therefore, the circuit could be replaced by a piece of wire from input A to output Y. Another way to state that is to say that input B is absorbed by the circuit.

The following circuit illustrates the second absorption property. Like the first absorption circuit, a truth table would demonstrate that input B is absorbed by the circuit.

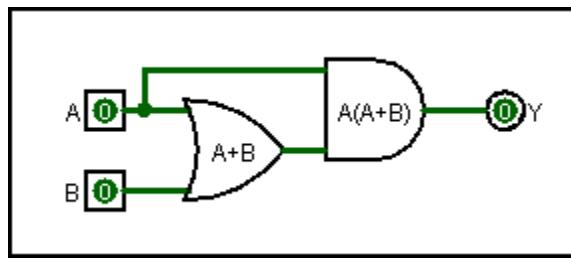


FIGURE 51: ABSORPTION PROPERTY (VERSION #2)

#### ADJACENCY

The adjacency property simplifies a circuit by removing unnecessary gates.

$$AB + AB' = A \quad (0.14)$$

This property can be proven by simple algebraic manipulation:

$AB + AB'$	Original Expression
$A(B + B')$	Distributive Property
$A1$	Complement Property
$A$	Identity Element

The following circuit illustrates the adjacency property. If this circuit were constructed it would be seen that the output, Y, is always the same as input A; therefore this entire circuit could be replaced by a single wire from input A to output Y.

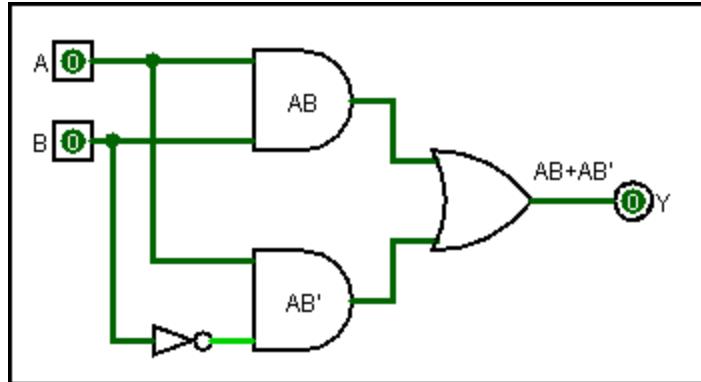


FIGURE 52: ADJACENCY PROPERTY

**MiniLab 2.8**

Select any one of the properties presented in this unit and build the circuit that is used to illustrate the property. After building the circuit, notice how the output matches what was predicted by the property. Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 2.8 ", and today's date. Then save the file with this name: "minilab\_2\_8".

## 2.9: DEMORGAN'S THEOREM

### INTRODUCTION

A mathematician named Augustus DeMorgan developed a pair of important theorems regarding the complementation of groups in Boolean algebra. DeMorgan found that an OR gate with all inputs inverted (a Negative-OR gate) behaves the same as a NAND gate with non-inverted inputs; and an AND gate with all inputs inverted (a Negative-AND gate) behaves the same as a NOR gate with non-inverted inputs. DeMorgan's theorem states that inverting the output of any gate is the same as using the opposite type of gate with inverted inputs. To put this in circuit terms, DeMorgan's theorem states that the NAND gate with normal inputs and the OR gate with inverted inputs in the following circuit are functionally equivalent.

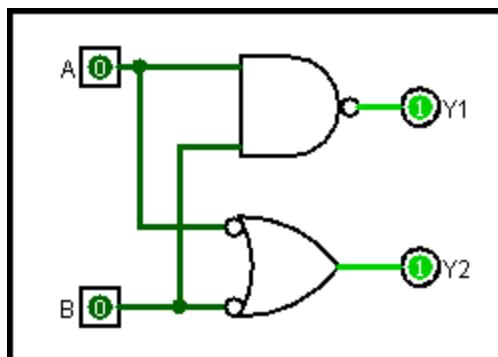


FIGURE 53: CIRCUIT EQUIVALENT OF DEMORGAN'S THEOREM

The NOT function is commonly represented as an apostrophe because it is easy to type, like:  $(AB)'$  for "A AND B NOT." However, it is easiest to work with DeMorgan's theorem if NOT is represented by an overbar rather than an apostrophe. Thus, you would write  $\overline{AB}$  rather than  $(AB)'$ . Remember that an overbar is a grouping symbol (like parenthesis) and it means that everything under that bar would be complemented.

### APPLYING DEMORGAN'S THEOREM

Applying DeMorgan's theorem to a Boolean expression may be thought of in terms of "breaking the bar." When applying DeMorgan's theorem to a Boolean expression:

1. A complement bar is broken over a group of variables
2. The operation (AND or OR) directly underneath the broken bar changes
3. Pieces of the broken bar remain over the individual variables.

To illustrate:

$$\overline{A * B} \leftrightarrow \overline{A} + \overline{B}$$

This formula shows how a two-input NAND gate is "broken" to form an OR gate with two inverted inputs.

$$\overline{A + B} \leftrightarrow \overline{A} * \overline{B}$$

This formula shows how a two-input NOR gate is "broken" to form an AND gate with two complemented inputs.

#### SIMPLE EXAMPLE

When multiple "layers" of bars exist in an expression, only one bar is broken at a time, and the longest, or uppermost, bar is broken first. To illustrate, consider this circuit:

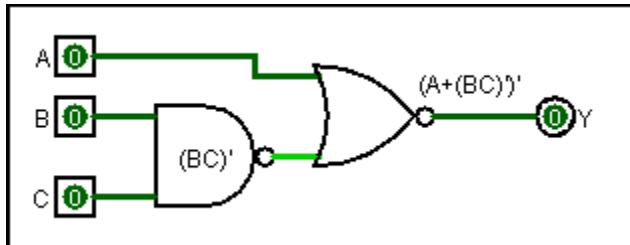


FIGURE 54: CIRCUIT TO SIMPLIFY WITH DEMORGAN'S THEOREM

By writing the output at each gate (as illustrated), it is easy to determine the Boolean expression for the circuit:

$$\overline{A + \overline{BC}}$$

To simplify the circuit, break the bar covering the entire expression (the "longest bar"), and then simplify the resulting expression.

$\overline{A + BC}$	Original Expression
$\overline{\overline{ABC}}$	"Break" the longer bar and change to AND
$\overline{\overline{ABC}}$	Involution Property

As a result, the original circuit is reduced to a three-input AND gate with one inverted input:

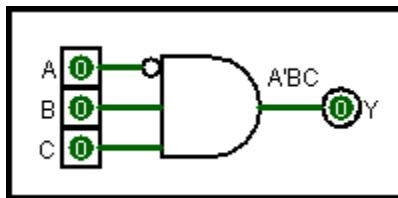


FIGURE 55: SIMPLIFIED CIRCUIT USING DEMORGAN'S THEOREM

#### INCORRECT APPLICATION OF DEMORGAN'S THEOREM

More than one bar is never broken in a single step, as illustrated by making that mistake with the previous problem:

$\overline{A + \overline{BC}}$	Original Expression
$\overline{\overline{AB} + \overline{C}}$	Improperly "breaking" two bars at one time
$\overline{\overline{AB} + C}$	Incorrect Solution

Thus, as tempting as it may be to take a shortcut and break more than one bar at a time, it often leads to an incorrect result. Also, while it is possible to properly reduce an expression by breaking the short bar first; more steps are usually required, so it is not recommended.

#### ABOUT GROUPING

An important, but easily neglected, aspect of DeMorgan's theorem concerns grouping. Since a long bar functions as a grouping symbol, the variables formerly grouped by a broken bar must remain grouped or else proper precedence (order of operation) will be lost. Therefore, after simplifying a large grouping of variables, place them in parentheses in order to keep the order of operation the same.

Consider the following circuit:

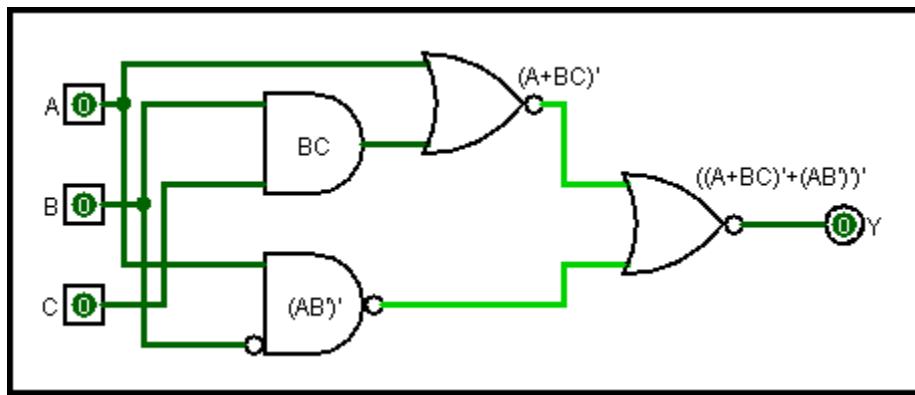


FIGURE 56: CIRCUIT USED TO ILLUSTRATE GROUPING WITH DEMORGAN'S THEOREM

As always, the first step in simplifying this circuit is to generate the Boolean expression, which is done by writing the sub-expression at the output of each gate, as illustrated.

$\overline{A + BC} + \overline{AB}$	Original Expression
$\overline{(A + BC)(AB)}$	Breaking the longest bar but keep grouping
$(A + BC)\overline{(AB)}$	Involution Property
$(A\overline{A}) + (\overline{B}\overline{C}\overline{A})$	Distribute $(A+BC)$
$(\overline{A}\overline{B}) + (\overline{B}\overline{C}\overline{A})$	Idempotence: $AA=A$
$(\overline{A}\overline{B}) + (0\overline{C}A)$	Complement: $BB'=0$
$(\overline{A}\overline{B}) + 0$	Annihilator: $A0=0$
$\overline{A}\overline{B}$	Identity Element: $A+0=A$

TABLE 8: SIMPLIFYING A CIRCUIT USING DEMORGAN'S THEOREM

The equivalent gate circuit for this much-simplified expression is as follows:

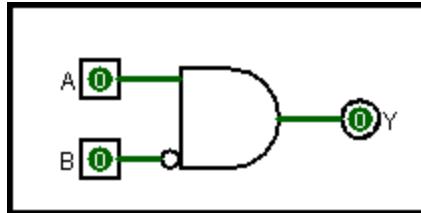


FIGURE 57: CIRCUIT SIMPLIFIED WITH DEMORGAN'S THEOREM

**SUMMARY**

Here are the important points to remember about DeMorgan's Theorem:

- It describes the equivalence between gates with inverted inputs and gates with inverted outputs.
- When "breaking" a complementation (or NOT) bar in a Boolean expression, the operation directly underneath the break (AND or OR) reverses and the broken bar pieces remain over the respective terms.
- It is normally easiest to approach a problem by breaking the longest (uppermost) bar before breaking any bars under it.
- Two complementation bars are never broken in one step.
- Complementation bars function as grouping symbols. Therefore, when a bar is broken, the terms underneath it must remain grouped. Parentheses may be placed around these grouped terms as a help to avoid changing precedence.

**EXAMPLE PROBLEMS**

The following examples use DeMorgan's Theorem to simplify a Boolean function.

	<b>Original Function</b>	<b>Simplified</b>
<b>1</b>	$(A+B)(\overline{ABC})(\overline{\overline{AC}})$	$\overline{ABC}$
<b>2</b>	$(AB+\overline{BC})+(\overline{BC}+\overline{AB})$	$\overline{BC}$
<b>3</b>	$(AB+\overline{BC})(AC+\overline{\overline{AC}})$	$\overline{A}+\overline{C}$

**MiniLab 2.9**

Build both the “Original Function” and the “Simplified” circuit for Example Problem One. After building both circuits, notice that the outputs are the same when the same inputs are applied. Be sure the standard lab identifying information is at the top left of the circuit: Name, “MiniLab 2.9”, and today’s date. Then save the file with this name: “minilab\_2\_9”.

## 2.10: BOOLEAN FUNCTIONS

Consider a generic two-input circuit with one output:

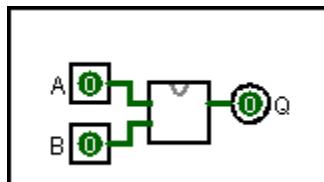


FIGURE 58: GENERIC 2-INPUT CIRCUIT

Without knowing anything about what is in the unlabeled box at the center of the circuit, there are a number of possible truth tables which could describe the circuit's output. Here are two possibilities:

Inputs		Output
A	B	Q
0	0	0
0	1	1
1	0	0
1	1	1

TRUTH TABLE 9: POTENTIAL OUTCOME 1

Inputs		Output
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

TRUTH TABLE 10: POTENTIAL OUTCOME 2

In fact, there are 16 possible truth tables for this one circuit. Each of those truth tables reflect a single potential "Function" of the circuit by setting various combinations of input/output. Therefore, any 2-input, 1-output circuit has 16 possible "functions." It is easiest to visualize all 16 combinations of inputs/outputs by using an odd-looking truth table. Consider only one of those 16 functions, the one shown in Truth Table 2 above:

	A	0	0	1	1	
	B	0	1	0	1	
F <sub>6</sub>		0	1	1	0	Exclusive-or (XOR): A $\oplus$ B. Also equivalent to A $\neq$ B; A or B, but not both.

TABLE 11: BOOLEAN FUNCTION 6

This line is for Function 6, or F6 (note that the pattern of the outputs is 0110). For this line, the output is true when (Input A is 0 and Input B is 1) OR when (Input A is 1 and Input B is 0). This is an XOR function, and the last column of the table verbally describes that function. Again, compare the Boolean Function 6

table with the table labeled Outcome 2 (above) in order to better understand the relationship between the two truth tables.

Following is the complete 16-Function table:

	<b>A</b>	0	0	1	1	
	<b>B</b>	0	1	0	1	
<b>F<sub>0</sub></b>	0	0	0	0	Zero or Clear. Always returns zero regardless of A and B input values (Annihilation)	
<b>F<sub>1</sub></b>	0	0	0	1	Logical AND = A*B. Returns A AND B.	
<b>F<sub>2</sub></b>	0	0	1	0	Inhibition = AB' (A, not B). Also equivalent to A>B or B<A.	
<b>F<sub>3</sub></b>	0	0	1	1	Copy A. Returns the value of A and ignores B's value. Sometimes called <i>Transfer</i> .	
<b>F<sub>4</sub></b>	0	1	0	0	Inhibition = BA' (B, not A). Also equivalent to B>A or A < B.	
<b>F<sub>5</sub></b>	0	1	0	1	Copy B. Returns the value of B and ignores A's value. Sometimes called <i>Transfer</i> .	
<b>F<sub>6</sub></b>	0	1	1	0	Exclusive-or (XOR): A⊕B. Also equivalent to A≠B; A or B, but not both.	
<b>F<sub>7</sub></b>	0	1	1	1	Logical OR = A+B. Returns A OR B.	
<b>F<sub>8</sub></b>	1	0	0	0	Logical NOR (NOT (A OR B)) = (A+B)'.	
<b>F<sub>9</sub></b>	1	0	0	1	Equivalence: (A = B). Also known as exclusive-NOR (not exclusive-or).	
<b>F<sub>10</sub></b>	1	0	1	0	NOT B. Returns B' and ignores A (B Complement).	
<b>F<sub>11</sub></b>	1	0	1	1	Implication, B implies A: A + B'. (if B then A). Also equivalent to B >= A.	
<b>F<sub>12</sub></b>	1	1	0	0	NOT A. Ignores B and returns A' (A Complement).	
<b>F<sub>13</sub></b>	1	1	0	1	Implication, A implies B: B + A' (if A then B). Also equivalent to A >= B.	
<b>F<sub>14</sub></b>	1	1	1	0	Logical NAND (NOT (A AND B)) = (A*B)'.	
<b>F<sub>15</sub></b>	1	1	1	1	One or Set. Always returns one regardless of A and B input values (Identity).	

TABLE 12: BOOLEAN FUNCTIONS

## 2.11: BOOLEAN FUNCTIONS AND PROPERTIES SUMMARY

### PRIMARY LOGIC FUNCTIONS

Inputs		Output
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 13: TRUTH TABLE FOR AND GATE

Inputs		Output
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 14: TRUTH TABLE FOR OR GATE

Input	Output
0	1
1	0

TABLE 15: TRUTH TABLE FOR NOT GATE

### SECONDARY LOGIC FUNCTIONS

Inputs		Output
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

TABLE 16: NAND GATE TRUTH TABLE

Inputs		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

TABLE 17: NOR GATE TRUTH TABLE

Inputs		Output
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 18: XOR GATE TRUTH TABLE

Inputs		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

TABLE 19: XNOR GATE TRUTH TABLE

Input	Output
0	0
1	1

TABLE 20: BUFFER TRUTH TABLE

### UNIVARIATE PROPERTIES

Property	OR	AND
Identity Elements	$A + 0 = A$	$1A = A$
Idempotence	$A + A = A$	$AA = A$
Annihilator	$A + 1 = 1$	$0A = A$
Complement	$A + A' = 1$	$AA' = 0$
Involution	$(A')' = A$	

TABLE 21: BOOLEAN UNIVARIATE PROPERTIES

## MULTIVARIATE PROPERTIES

Property	OR	AND
Commutative	$A + B = B + A$	$AB = BA$
Associative	$(A + B) + C = A + (B + C)$	$(AB)C = A(BC)$
Distributive	$A + (BC) = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption	$A + AB = A$	$A(A + B) = A$
DeMorgan	$\overline{A + B} = \overline{AB}$	$\overline{AB} = \overline{A} + \overline{B}$
Adjacency		$AB + AB' = A$

TABLE 22: BOOLEAN MULTIVARIATE PROPERTIES

## FUNCTIONS

	A	0	0	1	1	
	B	0	1	0	1	
<b>F<sub>0</sub></b>	0	0	0	0	Zero or Clear. Always returns zero regardless of A and B input values (Annihilation)	
<b>F<sub>1</sub></b>	0	0	0	1	Logical AND = $A * B$ . Returns A AND B.	
<b>F<sub>2</sub></b>	0	0	1	0	Inhibition = $AB'$ (A, not B). Also equivalent to $A > B$ or $B < A$ .	
<b>F<sub>3</sub></b>	0	0	1	1	Copy A. Returns the value of A and ignores B's value. Sometimes called <i>Transfer</i> .	
<b>F<sub>4</sub></b>	0	1	0	0	Inhibition = $BA'$ (B, not A). Also equivalent to $B > A$ or $A < B$ .	
<b>F<sub>5</sub></b>	0	1	0	1	Copy B. Returns the value of B and ignores A's value. Sometimes called <i>Transfer</i> .	
<b>F<sub>6</sub></b>	0	1	1	0	Exclusive-or (XOR): $A \oplus B$ . Also equivalent to $A \neq B$ ; A or B, but not both.	
<b>F<sub>7</sub></b>	0	1	1	1	Logical OR = $A + B$ . Returns A OR B.	
<b>F<sub>8</sub></b>	1	0	0	0	Logical NOR (NOT (A OR B)) = $(A + B)'$ .	
<b>F<sub>9</sub></b>	1	0	0	1	Equivalence: $(A = B)$ . Also known as exclusive-NOR (not exclusive-or).	
<b>F<sub>10</sub></b>	1	0	1	0	NOT B. Returns $B'$ and ignores A (B Complement).	
<b>F<sub>11</sub></b>	1	0	1	1	Implication, B implies A: $A + B'$ . (if B then A). Also equivalent to $B \geq A$ .	
<b>F<sub>12</sub></b>	1	1	0	0	NOT A. Ignores B and returns $A'$ (A Complement).	
<b>F<sub>13</sub></b>	1	1	0	1	Implication, A implies B: $B + A'$ (if A then B). Also equivalent to $A \geq B$ .	
<b>F<sub>14</sub></b>	1	1	1	0	Logical NAND (NOT (A AND B)) = $(A * B)'$ .	
<b>F<sub>15</sub></b>	1	1	1	1	One or Set. Always returns one regardless of A and B input values (Identity).	

TABLE 23: BOOLEAN FUNCTIONS

---

## 3: BINARY MATHEMATICS

### 3.1: INTRODUCTION TO NUMBER SYSTEMS

#### BACKGROUND

The expression of numerical quantities is often taken for granted, which is both a good and a bad thing in the study of electronics. It is good since the use and manipulation of numbers is familiar for many calculations used in analyzing electronic circuits. On the other hand, the particular system of notation that has been taught from grade school onward is not the system used internally in modern electronic computing devices; and learning any different system of notation requires some re-examination of deeply ingrained assumptions.

First, it is important to distinguish the difference between numbers and the symbols used to represent numbers. A number is a mathematical quantity, usually correlated in electronics to a physical quantity such as voltage, current, or resistance. There are many different types of numbers, for example:

- Whole Numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9 . . .
- Integers: -4, -3, -2, -1, 0, 1, 2, 3, 4 . . .
- Rational Numbers: -5.3, 0, 1/3, 6.7
- Irrational Numbers:  $\pi$  (approx. 3.1415927), e (approx. 2.718281828), square root of any prime number
- Real Numbers: (combination of all rational and irrational numbers)
- Complex Numbers:  $3 - j4$

Different types of numbers are used for different applications in electronics. As examples:

- Whole numbers work well for counting discrete objects, such as the number of resistors in a circuit.
- Integers are needed when negative equivalents of whole numbers are required, such as expressing a negative Direct Current (DC) voltage level.
- Irrational numbers are used to describe the charge/discharge cycle of electronic objects like capacitors.
- Real numbers, in either fractional or decimal form, are used to express the non-integer quantities of voltage, current, and resistance in DC circuits.
- Complex numbers, in either rectangular or polar form, must be used rather than real numbers to capture the dual essence of the magnitude and phase angle of the current and voltage in Alternating Current (AC) circuits.

There is a difference between the concept of a "number" as a measure of some quantity and the means used to express a number in spoken or written communication. A way to symbolically denote numbers had to be developed in order to use them to describe processes in the physical world, make scientific predictions, or balance a checkbook. The written symbol that represents some number, like how many

apples there are in a bin, is called a cipher; and in western mathematics, common ciphers include 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

### BINARY MATHEMATICS

Binary mathematics is a specialized branch of mathematics that concerns itself with a number system that contains only two ciphers: 0 and 1. It would seem to be very limiting to use only two ciphers; however, it is easy to create electronic devices that can differentiate between two voltage levels rather than the ten that would be needed for a decimal mathematics system.

### SYSTEMS WITHOUT PLACE VALUE

#### HASH MARKS

One of the earliest cipher systems was simply a hash mark to represent each quantity. For example, three apples could be represented like this: | | | . Often, five hash marks were "bundled" to aid in the counting of large quantities, so seven apples would be represented like this: + + + | | | .

#### ROMAN NUMERALS

The Romans devised a system that was a substantial improvement over hash marks, because it used a variety of ciphers to represent increasingly large quantities. The notation for 1 is the capital letter I. The notation for 5 is the capital letter V. Other ciphers possess increasing values:

Numeral	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

TABLE 24: ROMAN NUMERALS

If a cipher is accompanied by a second cipher of equal or lesser value to the immediate right of it, with no ciphers greater than that second cipher to the right, the second cipher's value is added to the total quantity. Thus, VIII symbolizes the number 8, and CLVII symbolizes the number 157. On the other hand, if a cipher is accompanied by another cipher of lesser value to the immediate left, that other cipher's value is subtracted from the first. Therefore, IV symbolizes the number 4 (V minus I), and CM symbolizes the number 900 (M minus C). The ending credit sequences for most motion pictures contain the date of production, often in Roman numerals. For the year 1987, it would read: MCMLXXXVII. To break this numeral down into its constituent parts, from left to right:

$$(M = 1000) + (CM = 900) + (LXXX = 80) + (VII = 7)$$

Large numbers are very difficult to denote with Roman numerals; and the left vs. right (or subtraction vs. addition) of values can be very confusing. Adding and subtracting two Roman numerals is also very

challenging, to say the least. Finally, one other major problem with this system is that there is no provision for representing the number zero or negative numbers, and both are very important concepts in mathematics. Roman culture, however, was more pragmatic with respect to mathematics than most, choosing only to develop their numeration system as far as it was necessary for use in daily life.

## SYSTEMS WITH PLACE VALUE

### DECIMAL NUMERATION

The Babylonians developed one of the most important ideas in numeration: they were the first to develop the concept of cipher position, or place value, in representing larger numbers. Instead of inventing new ciphers to represent larger numbers, as the Romans had done, they re-used the same ciphers, placing them in different positions from right to left to represent increasing values. This system also required a cipher that represents zero value, and the inclusion of zero in a numeric system was one of the most important inventions in all of mathematics (many would argue the single most important human invention, period). The decimal numeration system uses the concept of place value, with only ten ciphers (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) used in "weighted" positions to symbolize numbers.

Each cipher represents an integer quantity, and each place from right to left in the notation is a multiplying constant, or weight, for the integer quantity. For example, the decimal notation "1206" may be broken down into its constituent weight-products as such:

$$1206 = (1 \times 1000) + (2 \times 100) + (0 \times 10) + (6 \times 1)$$

Each cipher is called a "digit" in the decimal numeration system, and each weight, or place value, is ten times that of the one to the immediate right. So, there is a "ones" place, a "tens" place, a "hundreds" place, a "thousands" place, and so on, working from right to left.

The decimal numeration system uses ten ciphers, and place-weights that are multiples of ten. It is possible, though, to make a different numeration system using the same strategy, except with fewer or more ciphers.

### BINARY NUMERATION

The binary numeration system uses only two ciphers, and the weight for each place in a binary number is two times as much as the one to its right. Contrast this to the decimal numeration system that has ten different ciphers and the weight for each place is ten times the one to its right. The two ciphers for the binary system are "0" and "1," and these ciphers are arranged right-to-left in a binary number, each place doubling the weight of the previous place. The rightmost place is the "ones" place; and, moving to the left, is the "twos" place, the "fours" place, the "eights" place, the "sixteens" place, and so on. For example, the following binary number can be expressed, just like a decimal number, as a sum of each cipher value times its respective weight:

$$11010 = (1 \times 16) + (1 \times 8) + (0 \times 4) + (1 \times 2) + (0 \times 1)$$

The primary reason that the binary system is popular in modern electronics is because it is easy to represent the two cipher states (0 and 1) electronically; if no current is flowing in the circuit it

represents a binary zero, while flowing current represents a binary one. Binary numeration also lends itself to the storage and retrieval of numerical information: as examples, magnetic tapes have spots of iron oxide that are magnetized for a binary "1" or demagnetized for a binary "0" and optical disks have a laser-burned pit in the aluminum foil representing a binary "1" and an unburned spot representing a binary "0".

Digital numbers require so many bits to represent relatively small numbers, programming or analyzing electronic circuitry can be a tedious task. However, anyone working with digital devices must learn to quickly count in binary to at least 11111 (that is decimal 31). Any time spent practicing counting both up and down between 0 and 31 will be rewarded while studying binary mathematics, codes, and other digital logic topics. The following table will help:

Bin	Dec	Bin	Dec	Bin	Dec	Bin	Dec
0	0	1000	8	10000	16	11000	24
1	1	1001	9	10001	17	11001	25
10	2	1010	10	10010	18	11010	26
11	3	1011	11	10011	19	11011	27
100	4	1100	12	10100	20	11100	28
101	5	1101	13	10101	21	11101	29
110	6	1110	14	10110	22	11110	30
111	7	1111	15	10111	23	11111	31

TABLE 25: COUNTING TO 31 IN BINARY

#### OCTAL NUMERATION

The octal numeration system is a place-weighted system with a base of eight. Valid ciphers include the symbols 0, 1, 2, 3, 4, 5, 6, and 7 to make a total of eight. These ciphers are arranged right-to-left in an octal number, each place being eight times the weight of the previous place. For example, the following octal number can be expressed, just like a decimal number, as a sum of each cipher value times its respective weight:

$$4270 = (4 \times 512) + (2 \times 64) + (7 \times 8) + (0 \times 1)$$

#### HEXADECIMAL NUMERATION

The hexadecimal numeration system is a place-weighted system with a base of sixteen (the word "hexadecimal" is a combination of "hex" for 6 and "decimal" for 10). There needs to be ciphers for numbers greater than nine and English letters are used for those values. Here is a table showing the hexadecimal numbers up to decimal 15:

Hex	Dec	Hex	Dec
0	0	8	8
1	1	9	9
2	2	A	10
3	3	B	11
4	4	C	12
5	5	D	13
6	6	E	14
7	7	F	15

TABLE 26: HEXADECIMAL 0-F

Hexadecimal ciphers are arranged right-to-left, each place being 16 times the weight of the previous place. For example, the following hexadecimal number can be expressed, just like a decimal number, as a sum of each cipher value times its respective weight:

$$13A2 = (1 \times 4096) + (3 \times 256) + (A \times 16) + (2 \times 1)$$

!	How large of a number can be represented with a limited number of cipher positions? For example, if only four cipher positions are available, what is the largest number that can be represented in each of the numeration systems? With the crude hash-mark system, the number of places IS the largest number that can be represented, since one hash mark "place" is required for every integer step. For place-weighted systems, however, the answer is found by taking the number base of the numeration system (10 for decimal, 2 for binary) and raising that number to the power of the number of desired places. For example, in the decimal system, a 5-place number can represent $10^5$ , or 100,000, values from 0 to 99,999. Eight places in a binary numeration system, or $2^8$ , can represent 256 different values.
---	---

#### SUMMARY OF NUMERATION SYSTEMS

The following table counts from zero to twenty using several different numeration systems:

Text	Hash Marks	Roman	Dec	Bin	Oct	Hex
Zero	n/a	n/a	0	0	0	0
One		I	1	1	1	1
Two		II	2	10	2	2
Three		III	3	11	3	3
Four		IV	4	100	4	4
Five		V	5	101	5	5
Six		VI	6	110	6	6
Seven		VII	7	111	7	7
Eight		VIII	8	1000	10	8
Nine		IX	9	1001	11	9
Ten	+ + +	X	10	1010	12	A
Eleven	+ + +	XI	11	1011	13	B
Twelve	+ + +	XII	12	1100	14	C
Thirteen	+ + +	XIII	13	1101	15	D
Fourteen	+ + +	XIV	14	1110	16	E
Fifteen	+ + + + +	XV	15	1111	17	F
Sixteen	+ + + + +	XVI	16	10000	20	10
Seventeen	+ + + + +	XVII	17	10001	21	11
Eighteen	+ + + + +	XVIII	18	10010	22	12
Nineteen	+ + + + +	XIX	19	10011	23	13
Twenty	+ + + + + + +	XX	20	10100	24	14

! An interesting footnote for this topic concerns the first electronic digital computers, the ENIAC. The designers of the ENIAC chose to work with decimal numbers rather than binary in order to emulate a mechanical adding machine; unfortunately, this approach turned out to be counter-productive and required more circuitry (and maintenance nightmares) than if they had used binary numbers. "ENIAC contained 17,468 vacuum tubes, 7,200 crystal diodes, 1,500 relays, 70,000 resistors, 10,000 capacitors and around 5 million hand-soldered joints" (<http://en.wikipedia.org/wiki/Eniac>). Today, all digital devices use binary numbers for internal calculation and storage and convert those numbers to/from decimal only when necessary to interface with human operators.

## CONVENTIONS

Using different numeration systems can get confusing since many ciphers, like "1," are used in several different numeration systems. Therefore, the numeration system being used is typically indicated with a subscript following a number, like  $11010_2$  for a binary number or  $26_{10}$  for a decimal number. The subscripts are not mathematical operation symbols like superscripts, which are exponents; all they do is indicate the system of numeration being used. By convention, if no subscript is shown, then the number is assumed to be decimal.

One method used to represent hexadecimal numbers is the prefix "0x". This has been used for many years by programmers who work with any of the languages descended from C: C++, C#, Java, JavaScript, and certain shell scripts. Thus, "0x1A" would be the hexadecimal number 1A.

Another commonly used convention for hexadecimal numbers is to add an "h" (for "hexadecimal") after the number. This is used because it is easier to type that indicator than to use a subscript, especially on a machine like a computer keyboard, and is more intuitive than using a "0x" prefix. Thus,  $1A_{16}$  would be written 1Ah. In this case, the "h" only indicates that the number 1A is hexadecimal; it is not some sort of mathematical operator. In this course, all hexadecimal numbers will be indicated with a trailing "h" and subscripts will only be used when context does not make it clear whether the number is binary or some other system, like decimal.

As one final note, occasionally binary numbers are written with a "0b" prefix; thus "0b1010" would be  $1010_2$ . This is more of a programming standard and not often used in literature outside of programming; it will not be used in this book.

### 3.2: CONVERTING BETWEEN BASES

#### INTRODUCTION

The number of ciphers used by a number system (and therefore, the place-value multiplier for that system) is called the radix, or base, for the system. The binary system, with two ciphers (0 and 1), is "base two" numeration, and each position in a binary number is a binary digit (or "bit"). The decimal system, with 10 ciphers, is "base ten" numeration, and each position in a decimal number is a "digit." When working with various digital logic processes it is desirable to be able to convert between binary/octal/decimal/hexadecimal number systems.

#### EXPANDED POSITIONAL NOTATION

Expanded Positional Notation is a method of representing a number in such a way that each position is identified with both its cipher symbol and its place-value multiplier. For example, consider the decimal number 347:

$$347 = (3 \times 10^2) + (4 \times 10^1) + (7 \times 10^0)$$

The steps to use to expand a decimal number, like 347 are:

Step	Result
Count the number of digits in the number.	3 digits
Create a series of "( _ X _ )" connected by a "plus" sign so that you have one set for each of the digits in the original number.	$347 = (\_ \times \_) + (\_ \times \_) + (\_ \times \_)$
Fill in the digits of the original number on the left side of each set of parenthesis.	$347 = (3 \times \_) + (4 \times \_) + (7 \times \_)$
Fill in the radix (or base number) on the right side of each parenthesis.	$347 = (3 \times 10) + (4 \times 10) + (7 \times 10)$
Starting on the far right side of the expression, add a power for each of the base numbers. The powers start at 0 and increase to the left.	$347 = (3 \times 10^2) + (4 \times 10^1) + (7 \times 10^0)$

The above example is for a positive decimal integer; but a number in any base can also have a fractional part. In that case, the number's integer component is to the left of the radix point (or "decimal" point in the decimal system), while the fractional part is to the right of the radix point. For example, in the decimal number 139.25, "139" is the integer component while "25" is the fractional component. If a number includes a fractional component, then the expanded positional notation uses increasingly negative powers of the radix for numbers to the right of the radix point. Consider this binary example: 101.011<sub>2</sub>. The expanded positional notation for this number is:

$$101.011 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$$

When a number in expanded positional notation includes one or more negative radix powers, the radix point is assumed to be to the immediate right of the "zero" power term, but it is not actually written

into the notation. Expanded positional notation is useful in converting a number from one base to another.

#### BINARY TO DECIMAL

To convert a number in binary form to decimal, start by writing the binary number in expanded positional notation and then add all of the values. For example convert binary 1101 to decimal:

$$1101 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

$$1101 = 8 + 4 + 0 + 1$$

$$1101 = 13_{10}$$

Binary numbers with a fractional component are converted to decimal in exactly the same way, but the fractional parts use negative powers of two. Convert binary 10.11<sub>2</sub> to decimal:

$$10.11_2 = (1 \times 2^1) + (0 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2})$$

$$10.11_2 = 2 + 0 + \frac{1}{2} + \frac{1}{4}$$

$$10.11_2 = 2 + .5 + .25$$

$$10.11_2 = 2.75_{10}$$

Most technicians who work with digital circuits learn to quickly convert simple binary numbers to decimal in their heads. However, for longer numbers, it may be useful to write down the various place weights and add them up; something like a shortcut way of writing expanded positional notation. For example, convert the binary number 11001101 to decimal:

<b>Binary Number:</b>	1	1	0	0	1	1	0	1
- - - - - - - - -								
<b>(Read Down)</b>	1	6	3	1	8	4	2	1
	2	4	2	6				
				8				

Remember, a bit value of 1 in the original number means that the respective place weight gets added to the total value, while a bit value of 0 means that the respective place weight does not get added to the total value. Thus, using the table above this paragraph, the binary number 11001101 is converted to:

$$11001101_2 = 128 + 64 + 8 + 4 + 1$$

$$11001101_2 = 205_{10}$$

Note that the bit on the far right side of any binary number is the Least Significant Bit (LSB) because it has the least weight (the one's place). While the bit on the far left side is the Most Significant Bit (MSB) because it has the greatest weight (the one hundred twenty-eight's place in the example above).

**BINARY TO OCTAL**

The octal numeration system serves as a "shorthand" method of denoting a large binary number. Technicians find it easier to discuss a number like  $57_8$  rather than  $101111_2$ .

Because octal is a base eight system, and that is the same as  $2^3$ ; binary numbers can be converted to octal by simply grouping them into three's. As an example, convert  $101111_2$  to octal:

1	0	1		1	1	1
	5		7			

Thus,  $101111_2$  is equal to  $57_8$ .

If a binary integer cannot be grouped into an even grouping of three, it is padded on the left with 0's. For example, to convert  $11011101_2$  to octal, the most significant bit must be padded with a 0:

0	1	1		0	1		1	0	1
	3		3		5				

Thus,  $11011101_2$  is equal to  $335_8$ .

A binary fraction may need to be padded on the right with 0's in order to create even groups of three before it can be converted into octal. For example, convert  $0.1101101_2$  to octal:

0.	1	1	0		1	1	0		1	0	0
0.		6		6		4					

Thus,  $0.1101101_2$  is equal to  $664_8$ .

A binary mixed number may need to be padded on both the left and right with 0's in order to create even groups of three before it can be converted into octal. For example, convert  $10101.00101_2$  to octal:

0	1	0	1		0	0	1		0	1
2		5		.	1		2			

Thus,  $10101.00101_2$  is equal to  $25.12_8$ .

While it is easy to convert between binary and octal, the octal system is not frequently used in electronics texts since computers store and transmit binary numbers in groups of 16, 32, or 64 bits, which are multiples of four rather than three.

**BINARY TO HEXADECIMAL**

The hexadecimal numeration system serves as a "shorthand" method of denoting a large binary number. Technicians find it easier to discuss a number like  $2F_{16}$  rather than  $101111_2$ . Because hexadecimal is a base 16 system, and that is the same as  $2^4$ ; binary numbers can be converted to hexadecimal by simply grouping them into four's. As an example, convert  $10010111_2$  to hexadecimal:

1001	0111
9	7

Thus,  $10010111_2$  is equal to  $97h$ . (Note, this number is not pronounced "Ninety Seven." Since it is a hexadecimal number, and "ninety" is part of the decimal system, it is pronounced "Nine Seven Hex.")

A binary integer may need to be padded on the left with 0's in order to create even groups of four before it can be converted into hexadecimal. For example, convert  $1001010110_2$  to hexadecimal:

0010	0101	0110
2	5	6

Thus,  $1001010110_2$  is equal to  $256h$ .

A binary fraction may need to be padded on the right with 0's in order to create even groups of four before it can be converted into hexadecimal. For example, convert  $0.1001010110_2$  to hexadecimal:

0.1001	0101	1000
9	5	8

Thus,  $0.1001010110_2$  is equal to  $.958h$ .

A binary mixed number may need to be padded on both the left and right with 0's in order to create even groups of four before it can be converted into hexadecimal. For example, convert  $11101.10101_2$  to hexadecimal:

0001	1101.	1010	1000	
1	D	.	A	8

Thus,  $11101.10101_2$  is equal to  $1D.A8h$ .

#### OCTAL TO DECIMAL

The simplest way to convert an octal number to decimal is to write the octal number in expanded positional notation and then add each of the places. For example, to convert  $245_8$  to decimal:

$$\begin{aligned} 245_8 &= (2 \times 8^2) + (4 \times 8^1) + (5 \times 8^0) \\ 245_8 &= (2 \times 64) + (4 \times 8) + (5 \times 1) \\ 245_8 &= 128 + 32 + 5 \\ 245_8 &= 165_{10} \end{aligned}$$

If the octal number has a fractional component, then that part would be converted using negative powers of eight. As an example, convert octal  $25.71$  to decimal:

$$\begin{aligned} 25.71_8 &= (2 \times 8^1) + (5 \times 8^0) + (7 \times 8^{-1}) + (1 \times 8^{-2}) \\ 25.71_8 &= (2 \times 8) + (5 \times 1) + (7 \times .125) + (1 \times .015625) \\ 25.71_8 &= 21.890625 \end{aligned}$$

**HEXADECIMAL TO DECIMAL**

The simplest way to convert a hexadecimal number to decimal is to write the hexadecimal number in expanded positional notation and then add each of the places. For example, to convert 2A6h to decimal:

$$\begin{aligned}2A6_{16} &= (2 \times 16^2) + (A \times 16^1) + (6 \times 16^0) \\2A6_{16} &= (2 \times 256) + (10 \times 16) + (6 \times 1) \\2A6_{16} &= 512 + 160 + 6 \\2A6_{16} &= 678_{10}\end{aligned}$$

If the hexadecimal number has a fractional component, then that part would be converted using negative powers of 16. As an example, convert hexadecimal 1B.36 to decimal:

$$\begin{aligned}1B.36_{16} &= (1 \times 16^1) + (B \times 16^0) + (3 \times 16^{-1}) + (6 \times 16^{-2}) \\1B.36_{16} &= (1 \times 16) + (11 \times 1) + (3 \times \frac{1}{16}) + (6 \times \frac{1}{16^2}) \\1B.36_{16} &= (1 \times 16) + (11 \times 1) + (3 \times 0.0625) + (6 \times 0.00390625) \\1B.36_{16} &= 16 + 11 + 0.1875 + 0.0234375 \\1B.36_{16} &= 27.2109375\end{aligned}$$

**DECIMAL TO BINARY****INTEGERS**

Converting decimal to binary numbers is somewhat involved. A decimal integer is converted to another base by using repeated cycles of division. In the first cycle of division, the original decimal integer is divided by the base of the target numeration system (binary=2 octal=8, hex=16). Then, the whole-number portion of the quotient is divided by the base value again. This process continues until the quotient is less than 1. Finally, the binary, octal, or hexadecimal digits are determined by the "remainders" left over at each division step.



**Tip:** Once a decimal number is converted to binary, it can be easily converted to either octal or hexadecimal; so the easiest way to convert decimal to hexadecimal is to first convert it to binary and then convert that result to hexadecimal.

Note: Converting decimal fractions is a bit different and covered later in this module.

To convert  $87_{10}$  to binary, repeatedly divide 87 by 2 (the base for binary) until reaching zero. The following table shows those divisions. The number in column 1 is divided by 2 and the quotient is placed in column one on the next row with the remainder in column 2. For example, when 87 is divided by 2, the quotient is 43 with a remainder of 1. This division process is continued until the quotient is less than one.

Integer	Remainder
87	
43	1
21	1
10	1
5	0
2	1
1	0
0	1

When the division process is completed, the binary number is found by using the remainders, reading from the bottom to top. Thus  $87_{10}$  is  $1010111_2$ .

This repeat-division technique will also work for numeration systems other than binary. To convert a decimal integer to octal, for example, divide each line by 8; but follow the process as described above. As an example, to convert decimal 87 to octal:

Integer	Remainder
87	
10	7
1	2
0	1

RESULT:  $87_{10} = 127_8$

The same process can be used to convert a decimal integer to hexadecimal; except, of course, the divisor would be 16. Also, some of the remainders could be greater than 10, so these would have to be written as letters. For example, to convert decimal 678 to hexadecimal:

Integer	Remainder
678	
42	6
2	A
0	2

RESULT:  $678_{10} = 2A6h$

#### FRACTIONS

Converting decimal fractions to binary is a repeating operation similar to converting decimal integers, but each step repeats multiplication rather than division. To convert  $0.5625_{10}$  to binary, repeatedly multiply the fractional part of the number by 2 until the fractional part is zero (or whatever degree of precision is reached). The following table shows these multiplications. The number in column 2 (the fractional part of the factor) is multiplied by 2 and the integer part of the product is placed in column one on the next row with the fractional part in column 2. For example, when the top row, 0.5625, is multiplied by 2, the product is 1.125.

Integer	Fraction
	5625
1	125
0	25
0	5
1	0

When the multiplication process is completed, the binary number is found by using the integer parts, reading from the top to the bottom. Thus  $0.5625_{10}$  is  $0.1001_2$ .

Convert decimal 0.78125 to binary. Take the solution out to full precision (that is, the last multiplication yields a fractional part of zero).

Integer	Fraction
	78125
1	5625
1	125
0	25
0	5
1	0

Thus,  $0.78125_{10} = 0.11001_2$

Often, a decimal fraction will create a huge binary fraction. In that case, continue the multiplication until the desired number of binary places are achieved. As an example, convert decimal 0.1437 to binary (stop after 10 bits):

Integer	Fraction
	1437
0	2874
0	5748
1	1496
0	2992
0	5984
1	1968
0	3936
0	7872
1	5744
1	1488

Thus,  $0.1437_{10} = 0.0010010011_2$  (with 10 bits of precision). Note: to calculate this to full precision requires 6617 bits; thus, it is normally wise to specify the desired precision.

Converting decimal fractions to any other base would involve the same process, but remember that the base would be used as a multiplier. Thus, to convert decimal to hexadecimal multiply each line by 16 rather than 2.

**MIXED NUMBERS**

To convert a mixed decimal number (one that contains both an integer and fraction part) to binary, treat each component as a separate problem, and then combine the result. As an example, convert decimal  $375.125_{10}$  to binary.

First, convert  $375_{10}$  to binary:

Integer	Remainder
375	
187	1
93	1
46	1
23	0
11	1
5	1
2	1
1	0
0	1

Then, convert  $0.125_{10}$  to binary:

Integer	Fraction
	125
0	25
0	5
1	0

Then combine the two parts into a single answer:  $1\ 0111\ 0111.001_2$

A similar process could be used to convert decimal numbers into octal or hexadecimal, but using those base numbers instead of two, as in the example above.

**CALCULATORS**

For the most part, converting numbers between the various "computer" bases (binary, octal, or hexadecimal) is done with a calculator. Using a calculator is quick and error-free. However, for the sake of applying digital logic to a mathematical problem, it is essential to understand the theory behind converting bases. It will not be possible to construct a digital circuit where one step is "calculate the next answer on a hand-held calculator." Conversion circuits (like all circuits) need to be designed with simple gate logic, and an understanding of the theory behind the conversion process is important for that type of problem.



**Note:** Excel will convert between decimal/binary/octal/hexadecimal integers (including negative integers), but cannot handle fractions; however, the following website has a conversion tool that can convert between all bases (up to 62), both integer and fraction: <http://markknowsnothing.com/cgi-bin/baseconv.php>

### PRACTICE PROBLEMS

The following table lists several numbers in decimal, binary, octal, and hexadecimal form. To practice converting between numbers, select a number on any row and then convert it to the other bases.

Decimal	Binary	Octal	Hexadecimal
13.	1101.	15.	D.
30.	11110.	36.	1E.
175.	10101111.	257.	AF.
1872.	11101010000.	3520.	750.
0.0625	0.0001	0.04	0.1
0.15625	0.0011	0.12	0.28
0.90625	0.11101	0.72	0.E8
0.45703125	0.01110101	0.352	0.75
43.125	101011.001	53.1	2B.2
58.1875	111010.0011	72.14	3A.3
108.71875	1101100.10111	154.56	6C.B8
183.75	10110111.11	267.6	B7.C

TABLE 27: PRACTICE PROBLEMS

### 3.3: BINARY ADDITION

Adding binary numbers is a very simple task, and similar to the longhand addition of decimal numbers. As with decimal numbers, the bits are added one column at a time, from right to left. Unlike decimal addition, there is little to memorize in the way of an "Addition Table":

Inputs			Outputs	
Carry In	Augend	Addend	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

TABLE 28: BINARY ADDITION TABLE

Just as with decimal addition, two binary integers are added one column at a time, starting from the Least Significant Bit (the right-most bit in the integer):

$$\begin{array}{r}
 1001101 \\
 + 0010010 \\
 \hline
 1011111
 \end{array}$$

When the sum in one column includes a carry out, it is added to the next column to the left. Consider the following examples:

$$\begin{array}{r}
 11 \quad 1 \quad \text{---Carry bits} \\
 1001001 \\
 + 0011001 \\
 \hline
 1100010
 \end{array}$$

$$\begin{array}{r}
 11 \quad \text{---Carry bits} \\
 1000111 \\
 + 0010110 \\
 \hline
 1011101
 \end{array}$$

Binary numbers that include a fractional part are added just like binary integers; however the radix points must align so the augend and addend may need to be padded with zeros on either the left or the right. Here is an example:

$$\begin{array}{r}
 111\ 1 \quad \text{---Carry bits} \\
 1010.0100 \\
 + \underline{0011.1101} \\
 1110.0001
 \end{array}$$

### OVERFLOW ERROR

One problem circuit designers must accommodate is a carry out bit in the Most Significant Bit in the answer. Consider the following:

$$\begin{array}{r}
 11\ 11 \quad \text{---Carry bits} \\
 10101110 \\
 + \underline{11101101} \\
 110011011
 \end{array}$$

Suppose this calculation was done with a circuit that could only accommodate eight data bits. The augend and addend are both eight bits wide, so they are fine; however, the sum is nine bits wide due to that carry in the Most Significant Bit. In an eight-bit circuit, that carry bit would simply be dropped since there is not enough room to accommodate it. In decimal, the true answer to the problem above is  $174_{10} + 237_{10} = 411_{10}$ ; but if the Most Significant Bit is dropped, the answer becomes  $155_{10}$ , which is, of course, wrong. This type of error is called an *Overflow Error*, and a circuit designer must find a way to allow for overflow. One typical solution is to simply alert the user that there was an overflow error. For example, on a handheld calculator, often the display will change to something like -E- if there is an error of any sort, including overflow.

### SAMPLE PROBLEMS

The following table lists several binary addition problems that can be used for practice.

Augend	Addend	Sum
10110.	11101.	110011.
111010.	110011.	1101101.
1011.	111000.	1000011.
1101001.	11010.	10000011.
1010.111	1100.001	10111.000
101.01	1001.001	1110.011

TABLE 29: ADDITION PROBLEMS

### MiniLab 3.3

Logisim includes an 8-bit adder. Wire that adder to two 8-bit inputs and an 8-bit output, along with a 1-bit Carry In and 1-bit Carry Out. (See the illustration below.)

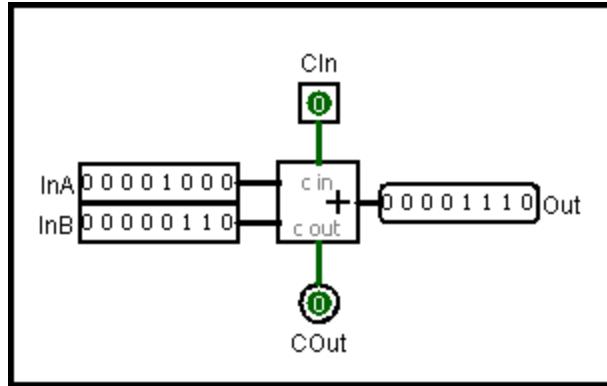


FIGURE 59: BINARY ADDITION

Enter several 8-bit numbers to check the function of this component. What happens if the input numbers cause an overflow? Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 3.3", and today's date. Then save the file with this name: "minilab\_3\_3".

### 3.4: BINARY SUBTRACTION

#### SIMPLE MANUAL SUBTRACTION

Subtracting binary numbers is similar to subtracting decimal numbers and uses the same process children learn in elementary school. The minuend and subtrahend are aligned on the radix point, and then columns are subtracted one at a time, starting with the least significant place and moving to the left. If the subtrahend is larger than the minuend for any one column, an amount is "borrowed" from the column to the immediate left. It may help to review the process with a simple decimal example:

$$\begin{array}{r} 54.3 \\ - 26.2 \\ \hline 28.1 \end{array}$$

In the above problem, 6 cannot be subtracted from 4, so 10 was borrowed from the 5, making the problem more like this:

$$\begin{array}{r} 45\ 14.3 \\ - 2\ 6.2 \\ \hline 2\ 8.1 \end{array}$$

Binary numbers can also be subtracted in the same way, but it is important to keep in mind that binary numbers have only two possible values: 0 and 1. Consider the following problem:

$$\begin{array}{r} 10.1 \\ - 01.0 \\ \hline 01.1 \end{array}$$

In this problem, the least significant column is 1 - 0, and that equals 0. The middle column, though, is 0 - 1, and that cannot be done. Therefore, one is borrowed from the most significant bit column, so the problem in middle column becomes 10 - 1. (Note: do not think of this as "ten minus one" - remember that this is binary so this problem is "one-zero minus one.") The middle column is 10 - 1 = 1. That makes the most significant column 0 - 0 = 0.

The radix point must be kept in alignment throughout the problem, so if one of the two operands has too few places, it should be padded on both the left and right to make both operands the same length. As an example, subtract: **101101.01** - **1110.1**:

$$\begin{array}{r} 101101.01 \\ - 001110.10 \\ \hline 1110.11 \end{array}$$

There is no difference between decimal and binary as far as the subtraction process is concerned. In each of the problems in this section the minuend is greater than the subtrahend, leading to a positive difference; however, if the minuend is less than the subtrahend, the result is a negative number; and negative numbers are developed in the next section of this module.

Here are some subtraction problems for practice:

Minuend	Subtrahend	Difference
1001011.	0111010.	10001.
100010.	010010.	10000.
101110110.	11001010.	10101100.
1110101.	111010.	111011.
11011010.1101	101101.1	10101101.0101
10101101.1	1101101.101	111111.111

TABLE 30: SUBTRACTION PROBLEMS

#### REPRESENTING NEGATIVE BINARY NUMBERS USING SIGN-AND-MAGNITUDE

Binary numbers, like decimal numbers, can be both positive and negative. While there are several methods of representing negative binary numbers; one of the simplest is using "sign-and-magnitude," which is essentially the same as placing a "-" in front of a decimal number. With the sign-and-magnitude system, the circuit designer simply designates the most significant bit as the "sign bit" and all others as the magnitude of the number. When the sign bit is 1 the number is negative, and when it is 0 the number is positive. Thus, decimal -5 would be written as  $1101_2$  in binary.

Unfortunately, despite the simplicity of the sign-and-magnitude approach, it is not very practical for arithmetic purposes. For instance, negative five (1101) cannot be added to any other binary number using standard addition technique since the sign bit would interfere. Errors can easily occur when bits are used for any purpose other than standard place-weighted values; for example, 1101 could be misinterpreted as the number 13 when in fact it is meant to represent -5. To keep things straight, the circuit designer must first decide how many bits are going to be used to represent the largest numbers in the circuit, and then be sure to not exceed that bit field length in arithmetic operations. For the above example, four bits would limit arithmetic operations to the representation of numbers from negative seven (1111) to positive seven (0111), and no more.

This system also has the quaint property of having two values for zero. If using three magnitude bits, these two numbers are both zero: 0000 (positive zero) and 1000 (negative zero).



The sign-and-magnitude system for representing negative numbers was used in early computers since it mimics real number arithmetic; but it required a number of logic gates (thus, electronic circuitry), so it was eventually replaced with a more efficient system.

#### REPRESENTING NEGATIVE BINARY NUMBERS USING SIGNED COMPLEMENTS

##### ABOUT COMPLEMENTATION

Before discussing negative binary numbers, it is important to understand the concept of complementation. The radix (or base) of any number system is the number of ciphers available for

counting; the decimal (or base-10) number system has ten ciphers (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) while the binary (or base-two) number system has two ciphers (0, 1).

#### DECIMAL

By definition, a number plus its complement equals the radix. This is called the "tens" complement for decimals since the radix of the decimal system is 10. Thus, the complement of decimal 8 is 2 since  $8 + 2 = 10$ , and 10 is the radix of the decimal system. The "diminished radix" (or  $\text{radix} - 1$ ) complement is called the "nines" complement in the decimal system. As an example, the nines complement of 7 is 2 because  $7 + 2 = 9$ , and 9 is the diminished radix of the decimal system.

To find the nines complement for a number larger than one place, the nines complement must be found for each place in the number. For example, to find the nines complement for decimal 538, find the nines complement for each of those three digits: 461. The easiest way to find the tens complement for a larger number is to first find the nines complement and then add one. For example, the tens complement of 283 is 717, which is calculated by finding the nines complement of 283 (716) and then adding one.

#### BINARY

Since the radix for a binary number is 102, the diminished radix is 12. The diminished radix complement is found by subtracting a binary number from  $1_2$ . This is normally called the "ones" complement. For example, the ones complement of 0 is found by  $1 - 0$ , or 1; and the ones complement of 1 is found by  $1 - 1$ , or 0. In essence, the ones complement is obtained by simply reversing (or "flipping") each bit in a binary number; so the ones complement of  $100101_2$  is  $011010_2$ .

The radix complement (or "twos complement") of a binary number is found by first calculating the ones complement and then adding one to that number. The ones complement for  $101101_2$  is  $010010_2$ , so the twos complement is  $010010_2 + 1_2$ , or  $010011_2$ .

#### SIGNED COMPLEMENTS

In circuits that use binary mathematics, a circuit designer can opt to use ones complement for negative numbers and designate the most significant bit as the sign bit; and, if so, the other bits are the magnitude of the number. With ones complement negative numbers, if the most significant bit is 0, then the number is positive and the magnitude of the number is determined by the remaining bits; but if the most significant bit is 1, then the number is negative and the magnitude of the number is determined by taking the ones complement of the number. Thus:  $0111 = +7$ , and  $1000 = -7$  (ones complement for 1000 is 0111). The following table may help to clarify this concept:

Decimal	Positive	Negative
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

TABLE 31: ONES COMPLEMENT NEGATIVE NUMBERS

In a four-bit binary number, any decimal number from -7 to +7 can be represented; but, notice that, like the sign-and-magnitude system, there are two values for 0, one positive and one negative. This requires extra circuitry to test for both values of zero after operations like subtraction.

To simplify circuit design, a designer can also opt to use twos complement negative numbers and designate the most significant bit as the sign bit; and, if so, the other bits are the magnitude of the number. To use twos complement numbers, if the most significant bit is 0, then the number is positive and the magnitude of the number is determined by the remaining bits; but if the most significant bit is 1, then the number is negative and the magnitude of the number is determined by taking the ones complement of each of the bits and then adding one (making the twos complement). Thus: 0111 = 7, and 1001 = -7 (ones complement of 1001 is 0110, and 0110 + 1 is 0111). The following table may help to clarify this concept:

Decimal	Positive	Negative
0	0000	10000
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001

TABLE 32: TWOS COMPLEMENT NEGATIVE NUMBERS

The twos complement removes that quirk of having two values for zero. The table above shows that zero is either 0000 or 10000; but since this is assumed to be a 4-bit circuit, that initial "1" is discarded, leaving 0000 for zero whether the number is positive or negative.

#### ABOUT CALCULATING THE TWOS COMPLEMENT

In the above section, the twos (or radix) complement is calculated by finding the ones complement of a number and then adding one. For machines, this is the most efficient method of calculating the twos complement; but there is a method that is much easier for humans to use to calculate the twos complement of a number. Start with the least significant bit (the right-most bit) and then read the

number from right to left. Look for the first "1," and then invert every bit to the left of that "1." As an example, the twos-complement for 101010 is formed by starting with the least significant bit (the 0 on the right), and working to the left, looking for the first 1, which is in the second place from the right. Then, every bit to the left is inverted, ending with: 010110.

Here are some examples:

Number	Twos Complement
0110100	1001100
11010	00110
001010	110110
1001011	0110101
111010111	000101001

TABLE 33: TWOS COMPLEMENTS

#### MiniLab 3.4A

Logisim includes a device that calculates the twos complement of an input number. It is called a "negator" and is found in the "Arithmetic" folder. It has a "-x" on its body when it is inserted into a circuit.

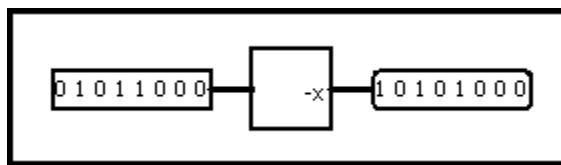


FIGURE 60: NEGATOR

Enter several 8-bit numbers to check the function of this component. Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 3.4A ", and today's date. Then save the file with this name: "minilab\_3\_4A".

#### SUBTRACTING USING THE DIMINISHED RADIX COMPLEMENT

##### DECIMAL

It is possible to subtract two decimal numbers using the 9s complement, as in the following example:

$$\begin{array}{r} 735 \\ - \underline{142} \end{array}$$

Calculate the 9s complement of the subtrahend: 857 (that is 9-1, 9-4, and 9-2). Then, add that 9s complement number to the original minuend:

$$\begin{array}{r} 735 \\ + \underline{857} \\ 1592 \end{array}$$

The initial "1" in the answer (the 1000s place) is dropped so the number of places in the answer is the same as for the two addends, leaving 592. Because the diminished radix is one less than the radix, one must be added to the answer; giving 593, which is the correct answer for 735-142.

Aside: For what it's worth, this is the method used by magicians who can subtract large numbers in their heads. While the explanation for complement subtraction is somewhat convoluted, the actual practice is fairly easy to master.

#### BINARY

The diminished radix complement (or ones complement) of a binary number is found by simply "flipping" each bit. Thus, the ones complement of 11010 is 00101. Just as in decimal, a binary number can be subtracted from another by adding the diminished radix complement of the subtrahend to the minuend, and then adding one to the answer. Here's an example:

$$\begin{array}{r} 101001 \\ - \underline{011011} \end{array}$$

Add the ones complement of the subtrahend:

$$\begin{array}{r} 101001 \\ + \underline{100100} \\ \hline 1001101 \end{array}$$

The most significant bit is discarded so the solution has the same number of bits as for the two addends. This leaves 001101. Adding one to that number leaves 1110. In decimal, the problem becomes 41-27=14.

Subtracting by adding the diminished radix of the subtrahend and then adding one may seem awkward for humans, but complementing and adding is a snap for digital circuits. In fact, many early mechanical calculators used a system of adding complements rather than having to turn gears backwards for subtraction. Because the diminished radix (or ones) complement of a binary number includes that awkward problem of having two representations for zero, this form of subtraction is not used in digital circuits; instead, the radix (or twos) complement is used.

#### SUBTRACTING USING THE RADIX COMPLEMENT

##### DECIMAL

The radix (or tens) complement of a decimal number is the nines complement plus one. Thus, the tens complement of 7 is 3; or (9-7+1) and the tens complement of 248 is 752 (that is 751 + 1). It is possible to subtract two decimal numbers using the tens complement, as in the following example:

$$\begin{array}{r} 735 \\ - \underline{142} \end{array}$$

Calculate the 10s complement of the subtrahend by finding the 9s complement and adding 1: 858 (that is 9-1, 9-4, and 9-2+1). Then, add that 10s complement number to the original minuend:

$$\begin{array}{r}
 735 \\
 + 858 \\
 \hline
 1593
 \end{array}$$

The initial "1" in the answer (the 1000s place) is dropped so the answer has the same number of decimal places as the addends, leaving 593, which is the correct answer for 735-142.

#### BINARY

To find the radix complement (or twos complement) of a binary number, each bit in the number is "flipped" (the ones complement) and then 1 is added to the result. Thus, the twos complement of 11010 is 00110 (or 00101+1). Just as in decimal, a binary number can be subtracted from another by adding the radix complement of the subtrahend to the minuend. Here's an example:

$$\begin{array}{r}
 101001 \\
 - \underline{011011} \\
 \hline
 \end{array}$$

Add the twos complement of the subtrahend:

$$\begin{array}{r}
 101001 \\
 + \underline{100101} \\
 \hline
 1001110
 \end{array}$$

The most significant bit is discarded so the solution has the same number of bits as for the two addends. This leaves 001110 (or decimal 14). Converting all of this to decimal, the original problem is 41-27=14.

#### OVERFLOW

One caveat with signed binary numbers is that of overflow, where the answer to an addition or subtraction problem exceeds the magnitude which can be represented with the allotted number of bits. Remember that the sign bit is defined as the most significant bit in the number. For example, with a six-bit number, five bits are used for magnitude, so there is a range from 00000 to 11111, or 0 to 31. If a sign bit is included, and using twos complement, numbers as high as 011111 (+31) or as low as 100000 (-32) are possible. However, an addition problem with two signed six-bit numbers that results in a sum greater than +31 or less than -32 will yield an incorrect answer. As an example, add 17 and 19 with signed six-bit numbers:

$$\begin{array}{r}
 010001 \quad (17) \\
 + \underline{010011} \quad (19) \\
 \hline
 100100
 \end{array}$$

The answer (100100), interpreted with the most significant bit as a sign, is equal to -28, not +36 as expected. Obviously, this is not correct. The problem lies in the restrictions of a six-bit number field. Since the true sum (36) exceeds the allowable limit for our designated bit field (five magnitude bits, or +31), it produces what is called an *overflow error*. Simply put, six places is not enough bits to represent the correct sum, so whatever sum is obtained will be incorrect. A similar error will occur if two negative

numbers are added together to produce a sum that is too low for a six-bit binary field. As an example, add -17 and -19:

$$\begin{array}{r} -17 = 101111 \\ -19 = 101101 \\ \hline 101111 & (-17) \\ + 101101 & (-19) \\ \hline 1011100 \end{array}$$

Solution as shown: 011100 = +28. (Remember that the most significant bit is dropped in order for the answer to have the same number of places as the two addends.)

The (incorrect) answer for this calculation is 28. The fact that the true sum of  $-17 + -19$  was too large to be properly represented with a five bit magnitude field is the cause of this difficulty.

Here is the same overflow problem again, but expanding the bit field to six magnitude bits plus a seventh sign bit. In the following example, both  $17+19$  and  $(-17)+(-19)$  are calculated to show that both can be solved using a seven-bit magnitude field rather than six-bits.

**Add 17 + 19**

$$\begin{array}{r} 0010001 & (17) \\ + 0010011 & (19) \\ \hline 0100100 & (36) \end{array}$$

**Add  $(-17) + (-19)$**

$$\begin{array}{r} 1101111 & (-17) \\ + 1101101 & (-19) \\ \hline 11011100 & (-36) \end{array}$$

The correct answer is only found by using bit fields sufficiently large to handle the magnitude of the sums.

#### ERROR DETECTION

Overflow errors in the above problems were detected by checking the problem in decimal form and then comparing the results with the binary answers calculated. For example, when adding  $+17$  and  $+19$ , the answer was supposed to be  $+36$ , so when the binary sum checked out to be  $-28$ , something had to be wrong. Although this is a valid way of detecting overflow errors, it is not very efficient, especially for computers. After all, the whole idea is to be able to reliably add binary numbers together and not have to double-check the result by adding the same numbers together in decimal form! This is especially true when building logic circuits to add binary quantities: the circuit has to be able to detect an overflow error without the supervision of a human who already knows the correct answer.

The simplest way to detect overflow errors is to check the sign of the sum and compare it against the signs of the numbers added. Obviously, two positive numbers added together will give a positive sum,

and two negative numbers added together will give a negative sum. With an overflow error, however, the sign of the sum is always opposite that of the two added numbers: +17 plus +19 gave -28, while -17 plus -19 gave +28. By checking the sign bits alone it becomes evident that something is wrong and an overflow error is detected.

In cases where the two addends have opposite signs, it is not possible to generate an overflow error. The reason for this is apparent when the nature of overflow is considered. Overflow occurs when the magnitude of a number exceeds the range allowed by the size of the bit field. If a positive number is added to a negative number, the sum will always be closer to zero than either of the two added numbers; or, its magnitude must be less than the magnitude of either original number, and so overflow is impossible.

### **MiniLab 3.4B**

Logisim includes an 8-bit subtractor. Wire that subtractor to two 8-bit inputs and an 8-bit output, along with a 1-bit Carry In and 1-bit Carry Out. (See the illustration below.)

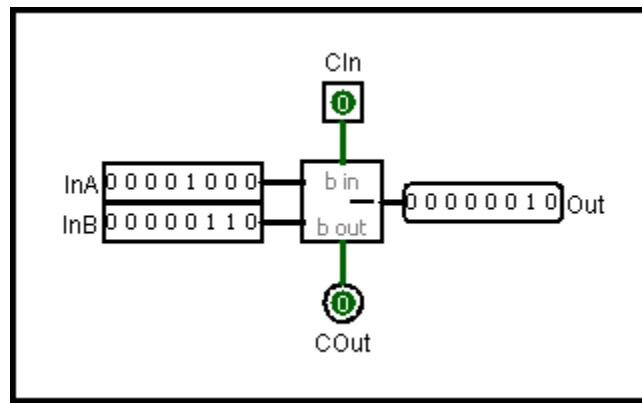


FIGURE 61: 8-BIT SUBTRACTOR MINILAB 3.4A

Enter several 8-bit numbers to check the function of this component. What happens if the output is a negative number? Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 3.4B ", and today's date. Then save the file with this name: "minilab\_3\_4B".

### **MiniLab 3.4C**

Modify the above circuit to include logic to handle negative differences (see the illustration below).

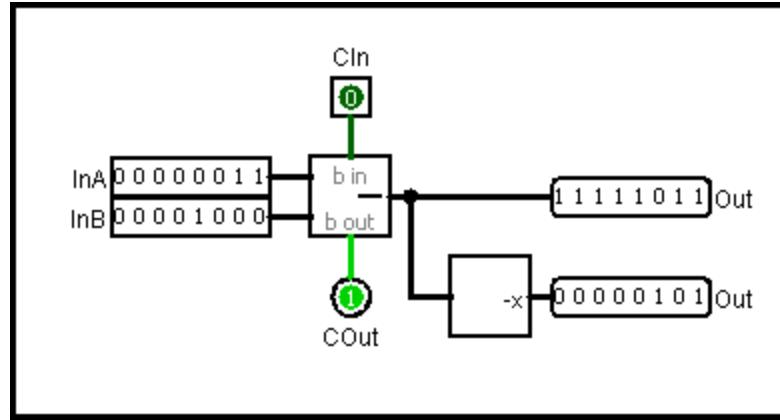


FIGURE 62: 8-BIT SUBTRACTOR MINILAB 3.4B

Enter several 8-bit numbers to check the function of this circuit. What happens if the output is a negative number (the illustration shows the problem  $3-8=-5$ )? Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 3.4C", and today's date. Then save the file with this name: "minilab\_3\_4C".

### 3.5: CODES

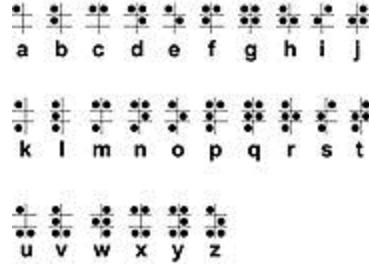
#### INTRODUCTION

Codes are nothing more than using one system of symbols to represent another system of symbols or information. Humans have used codes to encrypt secret information from ancient times. However, digital logic codes have nothing to do with secrets; rather, only with the efficient storage and retrieval of information.

#### MORSE CODE

As an example, Morse Code changes letters to sounds that can be easily transmitted over a radio or telegraph wire. Samuel Morse's code uses a series of dots and dashes to represent letters, so an operator at one end of the wire can use electromagnetic pulses to send a message to some receiver at a distant end. Most people are familiar with at least one phrase in Morse Code: SOS. Here is a short sentence in Morse:

- . - . - - - - . . . . . - - - .  
 C o d e s A r e F u n



#### BRAILLE ALPHABET

As one other example of a commonly-used code, in 1834 Louis Braille, at the age of 15, created a code of raised dots that enable blind people to read books. The illustration on the right shows the Braille alphabet.

#### COMPUTER CODES

The fact is, computers can only work with binary numbers; that is how information is stored in memory, how it is processed by the CPU, how it is transmitted over a network, and how it is manipulated in any of a hundred different ways. It all boils down to binary numbers. However, humans generally want a computer to work with words (such as email or a word processor), decimal numbers (such as a spreadsheet), or graphics (such as photos). All of that information must be encoded into binary numbers for the computer and then decoded back into understandable information for humans. Thus, binary numbers stored in a computer are often codes used to represent letters, programming steps, or other non-numeric information.

#### ASCII

Computers must be able to store and process letters, like those on this page. At first, it would seem easiest to create a "letter" code by simply making A = 1, B = 2, and so forth. Unfortunately, this simple code does not work for a number of reasons. However, the idea is on the right track and the code that is actually used for letters is similar to this simple example.

In the early 1960s, computer scientists came up with a code named *American Standard Code for Information Interchange*, or ASCII. The ASCII code is still among the most common ways to encode letters and other symbols for a computer. If the computer program knows that a particular spot in memory contains binary numbers that are actually ASCII-coded letters, it is a fairly easy job to convert

them to letters to display on a screen. For example, the word "Hello" in ASCII is: 048 065 06C 06C 06F

For simplicity, ASCII is usually represented by hexadecimal numbers, as in the "Hello" example above, rather than binary. The ASCII code has a predictable relationship between letters. For example, capital letters are exactly 20h higher in ASCII than their lower-case version. Thus, to change a letter from lower-case to upper-case, a programmer can simply add 20h to the ASCII code for the lower-case letter. This can be done in a single processing step by using what is known as a "bit-wise AND" on the bit representing 20h in the ASCII code's binary number.

Following is an ASCII chart using hexadecimal values. The most significant digit is read down in the left column and the least significant digit is read across the top row. For example, the letter "A" is 41h and the number "6" is 36h.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

TABLE 34: ASCII CHART

#### BINARY CODED DECIMAL (BCD)

It is often desirable to have numbers coded in such a way that they can be easily translated back and forth between decimal (which is easy for humans to manipulate) and binary (which is easy for computers to manipulate). The code used to represent decimal numbers in binary systems is called Binary Coded Decimal, or BCD. BCD is useful when working with decimal input (keypads or transducers) and output (displays).

There are, in general, two types of BCD systems: non-weighted and weighted. Non-weighted codes are special codes devised for a single purpose where there is no implied relationship between one value and the next. As an example, 1001 could mean 1 and 1100 could mean 2 in some device. The circuit designer would create whatever code meaning is desired for some specific application.

Weighted BCD is a more generalized system where each bit position is assigned a "weight," or value. These types of BCD systems are far more common and are found in all sorts of applications. Weighted BCD codes can be converted to decimal by adding the place value for each position, in exactly the same

way that Expanded Positional Notation is used for decimal and binary numbers. As an example, the weights for the Natural BCD system are 8-4-2-1. (These are the same weights used for binary numbers; thus the name "natural" for this system.) The code  $1001_{BCD}$  is converted to decimal like this:

$$\begin{aligned}1001_2 &= (1 \times 8) + (0 \times 4) + (0 \times 2) + (1 \times 1) \\1001_2 &= 9_{10}\end{aligned}$$

Because there are ten decimal ciphers (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9), it requires four bits to represent all decimal digits; so all BCD code systems are four bits wide. It is important to remember that BCD is a code system, not a number system; so the meaning of each combination of 4-bit codes is up to the designer and will not necessarily follow any sort of binary numbering sequence. In practice, only a few different BCD code systems are commonly used:

Decimal	8421 (Natural)	2421	Ex3	5421
0	0000	0000	0011	0000
1	0001	0001	0100	0001
2	0010	0010	0101	0010
3	0011	0011	0110	0011
4	0100	0100	0111	0100
5	0101	1011	1000	1000
6	0110	1100	1001	1001
7	0111	1101	1010	1010
8	1000	1110	1011	1011
9	1001	1111	1100	1100

TABLE 35: BINARY CODED DECIMAL

The name of each type of BCD code indicates the various place values. Thus, the "2421" BCD gives the most significant bit of the number a value of two, not eight as in the natural code. The Ex3 code (for "Excess 3") is the same as the natural code, but each value is increased by three (that is, three is added to the natural code).

In each of the BCD code systems in the table above, there are six unused 4-bit combinations; for example, in the "Natural" system the unused codes are: 1010, 1011, 1100, 1101, 1110, and 1111. Thus, any circuit designed to use BCD must include some sort of check to ensure that if unused binary values are accidentally input into a circuit it does not create an undefined outcome.

It is natural to wonder why there are so many different ways to code decimal numbers. Each of the BCD systems shown above have certain strengths and weaknesses; and a circuit designer would chose a specific system based upon those characteristics.

**Important:** It is important to remember that BCD is a code system, not a number system; so the meaning of each combination of 4-bit codes is up to the designer and will not necessarily follow any sort of binary numbering sequence.

**CONVERTING**

One thing that makes BCD so useful is the ease of converting from BCD to decimal. Each decimal digit is converted into a 4-bit BCD code, one at a time. Here is  $37_{10}$  in Natural BCD:

0011	0111
3	7

It is, generally, very easy to convert Natural BCD to decimal since the BCD codes are the same as binary numbers. Other BCD systems use different place values, and those require more thought to convert (though the process is the same). The place values for BCD systems other than Natural are indicated in the name of the system; so, for example, the 5421 system would interpret the number  $1001_{BCD5421}$  as:

$$\begin{aligned}1001_{bcd5421} &= (1 \times 5) + (0 \times 4) + (0 \times 2) + (1 \times 1) \\1001_{bcd5421} &= 6\end{aligned}$$

**SELF-COMPLEMENTING**

The Excess-3 code (called "Ex3" in the table) is self-complementing; that is, the 9's complement of any decimal number is found by complementing each bit in the Ex3 code. As an example:

**127<sub>10</sub> is 0100 0101 1010<sub>Ex3</sub>**  
**The ones complement of 0100 0101 1010<sub>Ex3</sub> is 1011 1010 0101<sub>Ex3</sub>**  
**1011 1010 0101<sub>Ex3</sub> is 872<sub>10</sub>, which is the 9s complement of 127<sub>10</sub>**

It is a powerful feature to be able to find the 9s complement of a decimal number by simply complementing each bit of its Ex3 BCD representation.

**REFLEXIVE**

Some BCD codes exhibit a reflexive property. Each of the upper five codes are complementary reflections of the lower five codes; that is,  $0111_{Ex3}$  (4) and  $1000_{Ex3}$  (5) are complements,  $0110_{Ex3}$  (3) and  $1001_{Ex3}$  (6) are complements, and so forth. The reflexive property for the 5421 code is different. Notice that the codes for 0-4 are the same as those for 5-9, except for the most significant bit (0 for the lower codes, 1 for the upper codes). Thus,  $0000_{5421}$  (0) is the same as  $1000_{5421}$  (5) except for the first bit,  $0001_{5421}$  (1) is the same as  $1001_{5421}$  (6) except for the first bit, and so forth.

**PRACTICE**

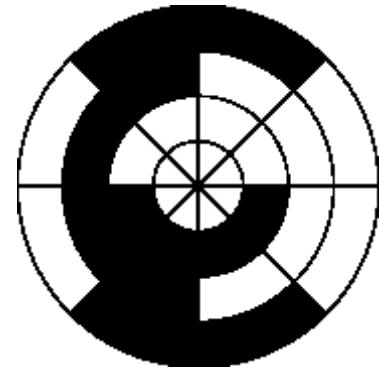
The following table shows several decimal numbers in various BCD systems. It can be used for practice in converting between these number systems.

Decimal	8421 (Natural)	2421	Ex3	5421
57	0101 1110	1011 1101	1000 1010	1000 1010
79	0111 1001	1101 1111	1010 1100	1010 1100
28	0010 1000	0010 1110	0101 1011	0010 1011
421	0100 0010 0001	0100 0010 0001	0111 0101 0100	0100 0010 0001
903	1001 0000 0011	1111 0000 0011	1100 0011 0110	1100 0000 0011

TABLE 36: SAMPLE BCD CONVERSION PROBLEMS

**GRAY CODE**

It is often desirable to use a wheel to encode digital input for a circuit. As an example, consider the tuning knob on a radio. The knob is attached to a shaft that has a small, clear disk which is etched with a code, similar to the illustration on the right. As the disk turns, the etched patterns open or block a light beam, which is then encoded into binary input.



One of the most challenging aspects of using a mechanical device to encode binary is ensuring that the input is stable. As the wheel turns past the light beam reader, if two of the etched areas change at the same time (thus, changing two bits at once), there is a high probability that the input will fluctuate between two or more values for a brief period of time. For example, imagine that the encoded circuit changes from 11 to 00 at one time. Since it is impossible to create a mechanical wheel precise enough to change those bits at exactly the same moment in time (remember that the light sensors will "see" an input several million times a second), as the bits change from 11 to 00 they will also change to 10 or 01 for a few microseconds. The entire change may form a pattern like 00-01-10-00-10-11. That instability would be enough to create havoc in a digital circuit.

The solution to the stability problem is to etch the disk with a code designed in such a way that only one bit changes at a time. The code used for that task is the Gray Code. Additionally, a Gray code is cyclic, so when it reaches its maximum value it can cycle back to its minimum value by changing only a single bit. In the above illustration, each of the concentric rings is for one bit in the 3-bit number. Notice that if the disk rotates past a fixed point, the black areas ("blocked light beam") change only one bit at a time; which is a characteristic of a Gray code pattern.

It is fairly easy to create a Gray code from scratch. Start by writing two bits, a 0 and 1:

```
0
1
```

Then, reflect those bits by writing them in reverse order underneath the original bits:

```
0
1
-----
1
0
```

Next, prefix the top half of the group with a 0 and the bottom half with a 1 to get a 2-bit Gray code.

```
00
01
11
10  (2-bit Gray Code)
```

Now, reflect all four values of a two-bit Gray code.

```
00  
01  
11  
10  
----  
10  
11  
01  
00
```

Next, prefix the top half of the group with a 0 and the bottom half with a 1 to get a 3-bit Gray code.

```
000  
001  
011  
010  
110  
110  
111  
101  
100 (3-bit Gray Code)
```

Now, reflect all eight values of a three-bit Gray code.

```
000  
001  
011  
010  
110  
111  
101  
100  
----  
100  
101  
111  
110  
010  
011  
001  
000
```

Finally, prefix the top half of the group with a 0 and the bottom half with a 1 to get a 4-bit Gray code.

```
0000  
0001  
0011
```

0010  
0110  
0111  
0101  
0100  
1100  
1101  
1111  
1110  
1010  
1011  
1001  
1000    **(4-bit Gray Code)**

The process of reflecting and prefixing can continue indefinitely to create a Gray code of any bit length. Of course, Gray code tables are also available in many different bit lengths.

2-Bit Code	3-Bit Code	4-Bit Code
00	000	0000
01	001	0001
11	011	0011
10	010	0010
	110	0110
	111	0111
	101	0101
	100	0100
		1100
		1101
		1111
		1110
		1010
		1011
		1001
		1000

TABLE 37: GRAY CODE TABLE

### 3.6: LAB – BCD ADDER

#### PURPOSE

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

#### PROCEDURE

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

Logisim 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:

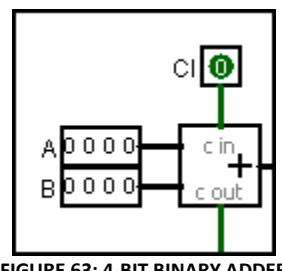


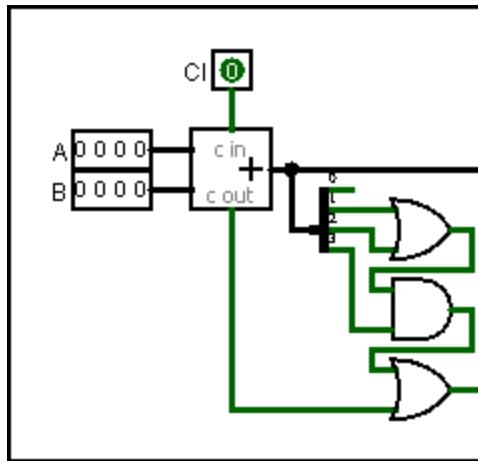
FIGURE 63: 4-BIT BINARY ADDER

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 adder 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  $10_{10}$  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 64: 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  $9_{\text{hex}}$  it must skip A, B, C, D, E, and F to reset back to  $0_{\text{hex}}$  for the next count. The simplest way to do this is add the value  $0110_2$  (hexadecimal 6) to any 4-bit binary number greater than  $1001_2$  (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- $5_{\text{hex}}$ . Thus, what follows  $09_{\text{hex}}$  is  $10_{\text{hex}}$ , not  $0A_{\text{hex}}$ . Here are two examples to help clarify this concept:

$$\begin{array}{r}
 1010 \text{ (Binary 10)} \\
 + 0110 \text{ (Binary 6)} \\
 \hline
 1\ 0000 \text{ (BCD 10)}
 \end{array}$$

$$\begin{array}{r}
 1110 \text{ (Binary 14)} \\
 + 0110 \text{ (Binary 6)} \\
 \hline
 1\ 0100 \text{ (BCD 14)}
 \end{array}$$

Once a binary number greater than  $1001_2$  (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.

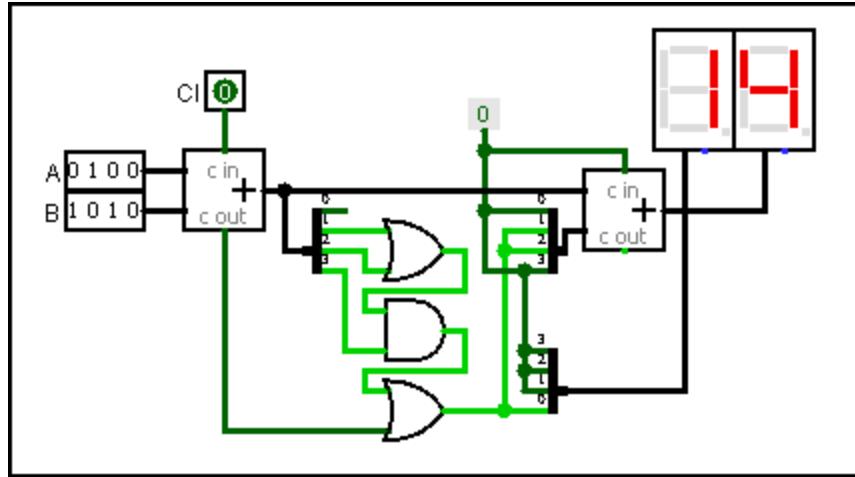


FIGURE 65: BCD ADDER

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 and the result goes to the hex display. If the result is greater than 9, then 6 is added to the number and that result is sent to the hex display.

*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 3.6: BCD Adder", and today's date. Save the file as *Lab 3\_6 – BCD Adder*.

### 3.7: LAB – 8-BIT BINARY TO BCD

#### PURPOSE

This lab creates an 8-bit Binary to BCD decoder.

#### PROCEDURE

This is a complex circuit that is based upon information found in the data sheet for IC 74x185 (see All Data Sheet: <http://www.alldatasheet.com/view.jsp?sSearchword=74185>).

#### INTERMEDIATE CIRCUIT

Using a new Logisim circuit, start by creating a new sub-circuit named *Intermediate*. That sub-circuit will contain the logic needed to convert a hexadecimal input into an intermediate BCD form that can later be used for a final BCD output.

For this sub-circuit, it is easiest to enter information into the Logisim Circuit and let the simulator build it. Open the analysis tool, specify A, B, C, D, and E as inputs; Y1, Y2, Y3, Y4, Y5, and Y6 as outputs. Both the Truth Table and Expressions are provided below, choose either method and enter that into the analyzer. Click the *Build Circuit* button to have Logisim build the circuit. Note: be certain the circuit is built in the *Intermediate* sub-circuit, not *main*.

#### TRUTH TABLE

Inputs	Outputs					
	Y1	Y2	Y3	Y4	Y5	Y6
00000	0	0	0	0	0	0
00001	1	0	0	1	1	0
00010	1	1	0	1	0	0
00011	0	0	1	0	0	1
00100	0	0	1	0	0	0
00101	0	0	0	0	0	1
00110	0	1	0	0	1	0
00111	1	1	0	1	0	1
01000	0	1	0	0	0	0
01001	1	1	0	1	1	0
01010	0	0	0	0	1	0
01011	1	0	0	1	0	1
01100	1	0	0	1	0	0
01101	0	1	0	0	0	1
01110	0	0	1	0	1	0
01111	0	0	0	0	1	1
10000	1	0	0	0	0	0
10001	0	1	0	1	1	0
10010	0	0	1	1	0	0
10011	0	0	0	1	0	1

1	0	1	0	0	0	0	1	0	0
1	0	1	0	1	1	0	0	0	1
1	0	1	1	0	1	1	0	0	1
1	0	1	1	1	0	0	1	1	0
1	1	0	0	0	1	1	0	0	0
1	1	0	0	1	0	0	1	1	0
1	1	0	1	0	1	0	1	0	1
1	1	1	0	0	1	0	1	0	0
1	1	1	0	1	1	1	0	0	1
1	1	1	1	0	0	0	1	1	0
1	1	1	1	1	1	0	0	0	1

TABLE 38: BIN TO BCD INTERMEDIATE CIRCUIT

## EXPRESSIONS

$$Y_1: \sim A \sim C \sim D E + \sim A \sim B \sim C D \sim E + \sim A \sim B C D E + \sim A B \sim C E + \sim A B C \sim D \sim E + A \sim C \sim D \sim E + A C \sim D E + A \sim B C D \sim E + A B \sim C \sim E + A B C E$$

$$Y_2: \sim A \sim B D \sim E + \sim A \sim B C D + \sim B C D \sim E + \sim A B \sim C \sim D + B \sim C \sim D \sim E + \sim A B \sim D E + A \sim B \sim C \sim D E + A B \sim C D E + A B C \sim D$$

$$Y_3: \sim A \sim B \sim C D E + \sim A \sim B C \sim D \sim E + \sim A B C D \sim E + A \sim B \sim C D \sim E + A \sim B C D E + A B \sim C \sim D E$$

$$Y_4: \sim C \sim D E + \sim B \sim C D \sim E + \sim B C D E + B \sim C E + B C \sim D \sim E + A \sim C E + A C \sim D \sim E + A B C \sim E$$

$$Y_5: \sim C \sim D E + C D \sim E + B D \sim E + B C D$$

$$Y_6: \sim B D E + C E + A D E$$

## MAIN CIRCUIT

Once the *Intermediate* sub-circuit is ready, create the following *main* circuit. Each of the ICs in this circuit is a copy of the *Intermediate* subcircuit.

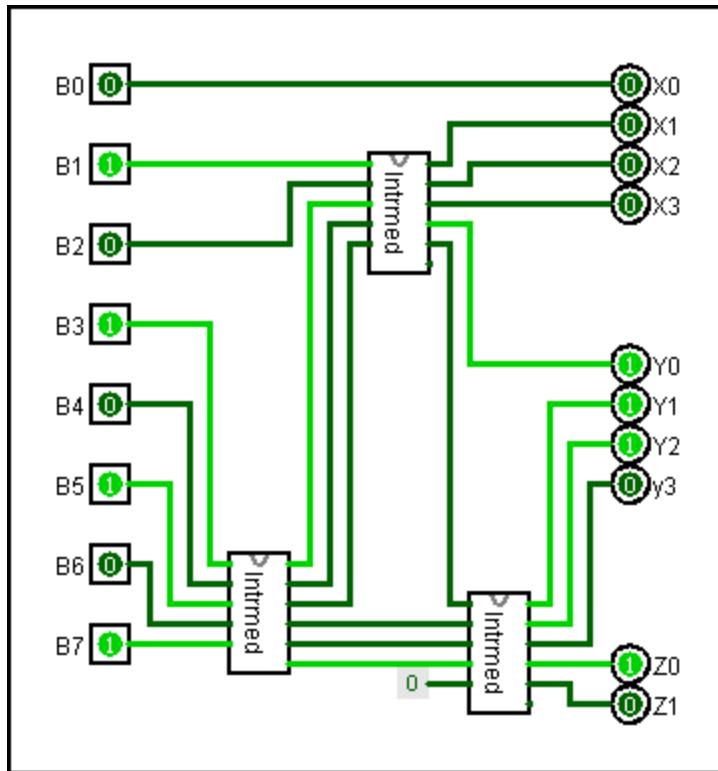


TABLE 39: 8-BIT BINARY TO BCD

The illustration shows an input of  $10101010_2$  being converted to  $01\ 0111\ 0000_{BCD}$ , or  $170_{10}$ . The most significant bit is at the bottom of each group.



**NOTE:** Be careful when constructing this circuit. The bottom output port (Y6) on the second and third *Intermediate* IC is not connected. That is hard to see in the illustration, but an important detail to note.

#### CHALLENGE

The output is presented in three BCD groups, but to make it easier to read, replace those groups with three 7-segment displays.

#### CLEANUP

Rename the *main* circuit to *Bin2BCD*. Be sure the standard identifying information block is at the top left of the *Bin2BCD* circuit: Name, "Lab 3.7: Binary to BCD", and today's date. Save the file as *Lab 3\_7 – Binary to BCD*.

## 4: SIMPLIFICATION OF BOOLEAN EXPRESSIONS

### 4.1 INTRODUCTION

Electronic circuits that do not require any memory devices (like flip-flops or registers) use what is called "Combinational Logic." These systems can be quite complex, but all outputs are determined solely on the status of the inputs to a series of logic gates. Combinational circuits can be reduced to a Boolean Algebra expression, though one that may be quite complex; and that expression is subject to the rules of simplification as covered in this unit.

In contrast, electronic circuits that require memory devices (like flip-flops or registers) use what is called "Sequential Logic." Those circuits often include feedback loops so that the final output is determined by both the inputs and the feedback. This makes sequential logic circuits much more complex than combinational, and the simplification of those circuits is covered in another unit.

Finally, many complex circuits include both combinational and sequential subcircuits. In that case, the combinational portion of the circuit is subject to simplification using the techniques in this unit, while the sequential portion would be simplified by techniques found in another unit.

## 4.2 CREATING BOOLEAN EXPRESSIONS

A circuit designer is often only given a written (or oral) description of a circuit and asked to build that device. It may well be that the designer will get some scribble on the back of a dinner napkin along with some verbal description of the desired output; and be expected to build a circuit to accomplish that task. Regardless of the request's form, the process that the designer follows, in general, is:

1. Problem Statement. The problem to be solved is written in a clear, concise statement. The better this statement is written, the easier each of the following steps will be; so time spent re-writing the problem statement will be worthwhile.
2. Construct Truth Table. Once the problem is clearly defined, the circuit designer constructs a truth table where all inputs/outputs are included. All possible input combinations that lead to a "true" output must be identified.
3. Write switching equation. When the truth table is completed, it is easy to create a switching equation; and that process is covered elsewhere in this unit.
4. Simplify switching equation. The switching equation should be simplified as much as possible. That process is covered elsewhere in this unit.
5. Draw logic diagram. A circuit can be easily constructed from the final switching equation.
6. Build circuit. If desired, a physical circuit can be built using the logic diagram.

### EXAMPLE

A machine is to be programmed to help pack shipping boxes for the ABC Novelty Company. They are running a promotion so if a customer purchases any of the following two items (but not all three), a free poster will be added to the purchase: joy buzzer, fake blood, itching powder. Design the logic needed to add the poster to appropriate orders.

1. Problem Statement. Actually, the problem is fairly well stated. A circuit is needed that will activate the "drop poster" machine when any two of three inputs (joy buzzer, fake blood, itching powder) is true.
2. Construct Truth Table. Let "J" be the Joy Buzzer "B" be the Fake Blood, and "P" be the Itching Powder; and let the truth table contain True (or 1) when any of those items are present in the shipping box. Let the output "D" be for "Drop Poster" and when it is true (or 1), then a poster will be dropped into the shipping box.

Inputs			Output
J	B	P	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

TABLE 40: EXAMPLE PROBLEM TRUTH TABLE

3. Write switching equation. This process is covered more thoroughly in another unit, but it may be easy to see how the following equation could be generated from the truth table.

$$D = BP + JP + JB$$

4. Simplify switching equation. The switching statement for this example problem is already as simple as possible, so no further simplification is needed.

5. Draw logic diagram. The following logic diagram was drawn from the switching equation:

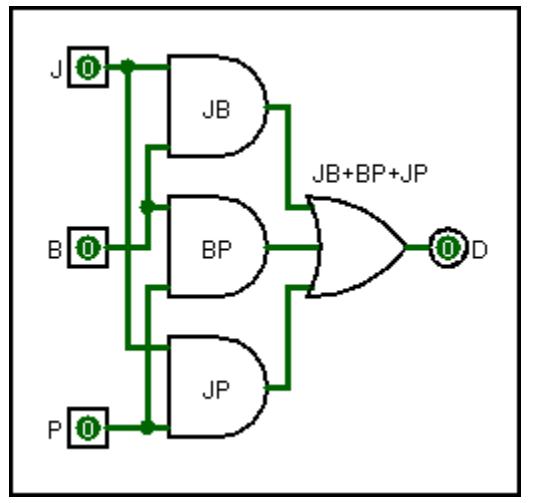


FIGURE 66: CIRCUIT GENERATED FROM SWITCHING EQUATION

6. This circuit could be realized with three AND gates and one 3-input OR gate.

### 4.3: MINTERMS AND MAXTERMS

#### INTRODUCTION

The solution to a Boolean equation is normally expressed in one of two formats: the sum of a group of products (Sum-Of-Products, or SOP) or the product of a group of sums (Product-of-Sums, or POS).

#### SUM OF PRODUCTS (SOP) DEFINED

An example of a SOP expression follows. Notice that the expression describes four inputs (A, B, C, D) and how those inputs are related. They are grouped into two 4-input AND gates and those two gates are combined through a 2-input OR gate.

$$Y = (A'BC'D) + (AB'CD)$$

Each of the two terms in this expression is called a "minterm." Minterms can be identified in a Boolean expression as a group of inputs joined by an AND gate; and then two or more minterms can be combined with an OR gate. The circuit would be realized like this:

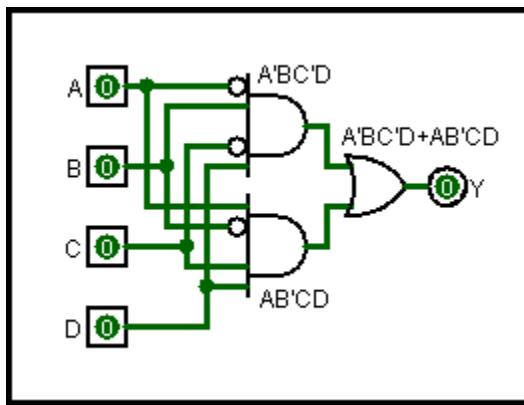


FIGURE 67: SAMPLE CIRCUIT ILLUSTRATING THE SUM-OF-PRODUCTS EXPRESSION

#### PRODUCT OF SUMS (POS) DEFINED

An example of a POS expression follows. Notice that the expression describes four inputs (A, B, C, D) and how those inputs are related. They are grouped into two 4-input OR gates and those two gates are combined through a 2-input AND gate.

$$Y = (A' + B + C' + D)(A + B' + C + D)$$

Each term in this expression is called a "maxterm." Maxterms can be identified in a Boolean expression as a group of inputs joined by an OR gate; and then two or more maxterms can be combined with an AND gate. The circuit would be realized like this:

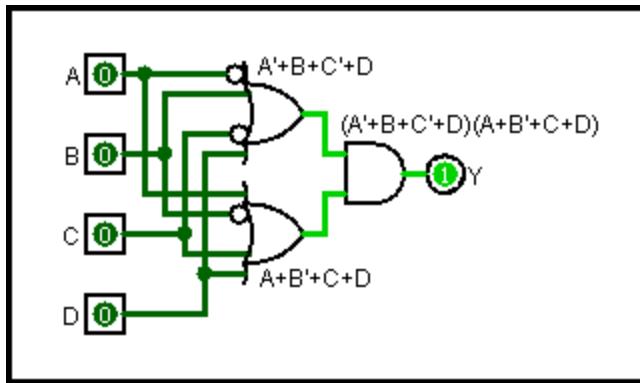


FIGURE 68: SAMPLE CIRCUIT ILLUSTRATING THE PRODUCT-OF-SUMS EXPRESSION

Both SOP's and POS's can be easily identified from a truth table, and they can then be further simplified using Boolean algebra methods; thus, they are very commonly used.

#### ABOUT MINTERMS

A term that contains all of the input variables in one row of a truth table joined with an AND gate is called a ***minterm***. Consider the following truth table for a circuit with three inputs (A, B, and C) and one output (Q):

Inputs			Output	
A	B	C	Q	m
0	0	0	0	0
0	0	1	0	1
0	1	0	0	2
0	1	1	1	3
1	0	0	0	4
1	0	1	1	5
1	1	0	0	6
1	1	1	0	7

TABLE 41: EXAMPLE MINTERMS

The circuit that is represented by this truth table would output a "True" in only two cases; when the inputs are A'BC or AB'C. The Boolean equation for this circuit is:

$$Q = (A'BC) + (AB'C)$$

The terms A'BC and AB'C are called minterms and they contain every combination of input variables that outputs a "True" when the inputs are joined by an AND gate. Minterms are most often used to describe circuits that have fewer "True" outputs than "False" (that is, there are fewer 1's than 0's in the output column). In the example above, there are only two "True" outputs with six "False" outputs, so minterms describe the circuit most efficiently.

Minterms are frequently abbreviated with a lower-case *m* along with a subscript that indicates the decimal value of the variables. For example, A'BC, the first of the "True" outputs in the truth table

above, have a binary value of 011, which is a decimal value of 3; thus, the minterm would be  $m_3$ . The other minterm in this equation is  $m_5$  since its binary value is 101, which equals decimal 5. It is possible to describe the entire circuit as minterms:  $m_3$  AND  $m_5$  joined by an OR gate.

As another example, consider the following truth table:

Inputs				Output	
A	B	C	D	Q	m
0	0	0	0	1	0
0	0	0	1	0	1
0	0	1	0	0	2
0	0	1	1	0	3
0	1	0	0	0	4
0	1	0	1	1	5
0	1	1	0	1	6
0	1	1	1	0	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	0	10
1	0	1	1	0	11
1	1	0	0	0	12
1	1	0	1	0	13
1	1	1	0	1	14
1	1	1	1	0	15

TABLE 42: EXAMPLE MINTERMS

The Boolean equation that includes the minterms for all "True" outputs is:

$$Q = A'B'C'D' + A'BC'D + A'BCD' + ABCD'$$

These would be minterms  $m_0$ ,  $m_5$ ,  $m_6$ , and  $m_{14}$ . There is a commonly used, more compact way to express this result:

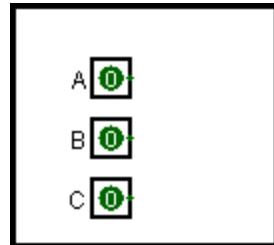
$$\sum(A,B,C,D) = \sum(0,5,6,14)$$

The above notation would indicate "For the function of inputs A, B, C, and D, the output is true for minterms 0, 5, 6, and 14." This format is called *Sigma Notation*, and it is easy to derive the full Boolean equation from it. Remember that  $m_0$  is 0000, or  $A'B'C'D'$ ,  $m_5$  is 0101, or  $A'BC'D$ ,  $m_6$  is 0110, or  $A'BCD'$ , and  $m_{14}$  is 1110, or  $ABCD'$ . The equation shown above can be quickly created.

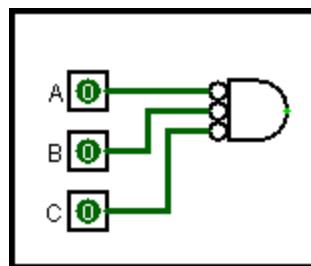
An equation that is described using minterms is often called the "Sum of Products" (or SOP) since each term is composed of inputs multiplied together, but the terms are summed. The "SOP" nature of a minterm solution is why the Greek letter *Sigma* is used to represent this Boolean expression.

**MiniLab 4.3A**

Creating a circuit from a minterm expression is fairly easy. As an example, consider this expression:  $\sum(A,B,C) = \sum(0,4,7)$ . There are three inputs, labeled A, B, and C.



The first term in the solution is minterm 0, or  $A'B'C'$ . Notice that all inputs are inverted to make this AND gate output a logic one if all four inputs are low.



Next, the AND gates for minterms 4 and 7 are added, along with a combining OR gate and an output port:

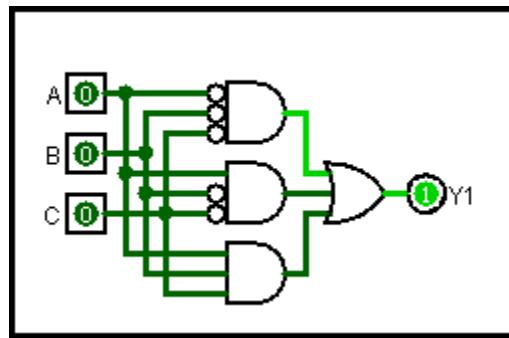


FIGURE 69: CIRCUIT FOR MINTERMS 0,4,7

Now, create a minterm generator for this expression:  $\sum(A,B,C) = \sum(1,2,5)$ . Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 4.3A", and today's date. Then save the file with this name: "minilab\_4\_3A".

### ABOUT MAXTERMS

A term that contains all of the input variables joined with an OR gate (or added together) for a FALSE output is called a **maxterm**. Consider the following truth table with three inputs (A, B, and C) and one output (Q):

Inputs			Output	
A	B	C	Q	M
0	0	0	1	0
0	0	1	1	1
0	1	0	0	2
0	1	1	1	3
1	0	0	1	4
1	0	1	1	5
1	1	0	0	6
1	1	1	1	7

TABLE 43: EXAMPLE MAXTERMS

The circuit that is represented by this truth table would output a "False" in only two cases. Since there are fewer "False" outputs than "True," it is easier to create a Boolean equation that would generate the "False" outputs rather than "True." Therefore, the equation for this circuit describes the complement of the output; or the "False" lines. Applying DeMorgan's Theorem to the inputs (in order to complement the output) changes all inputs to their complement and then combines them with an OR gate. Here is the Boolean equation for this circuit:

$$Q = (A + B' + C)(A' + B' + C)$$

The terms  $A + B' + C$  and  $A' + B' + C$  are called maxterms and they contain the complement of every input variable joined by an OR gate. Maxterms are most often used to describe circuits that have more "False" outputs than "True." In the example above, there are only two "False" outputs with six "True" outputs, so maxterms describe the circuit most efficiently.

Maxterms are frequently abbreviated with an upper-case M along with a subscript that indicates the decimal value of the variables. For example,  $A + B' + C$ , the first of the "False" outputs in the truth table above, has a binary value of 010, which is a decimal value of 2; thus, the maxterm would be  $M_2$ . This can be confusing, but remember that the *complements* of the inputs are used to form the expression. Thus,  $A + B' + C$  is 010, not 101. It may help to note the input values for the first "False" line in the truth table. The other maxterm in this equation is  $M_6$  since its binary value is 110, which equals decimal 6. It is possible to describe the entire circuit as a group of maxterms:  $M_2$  and  $M_6$ .

As another example, consider the following truth table:

Inputs				Output	
A	B	C	D	Q	M
0	0	0	0	1	0
0	0	0	1	1	1
0	0	1	0	1	2
0	0	1	1	0	3
0	1	0	0	1	4
0	1	0	1	1	5
0	1	1	0	1	6
0	1	1	1	1	7
1	0	0	0	1	8
1	0	0	1	0	9
1	0	1	0	1	10
1	0	1	1	1	11
1	1	0	0	0	12
1	1	0	1	1	13
1	1	1	0	1	14
1	1	1	1	0	15

TABLE 44: EXAMPLE MAXTERMS

The Boolean equation that includes the maxterms for all "False" outputs is:

$$Q = (A+B+C'+D')(A'+B+C+D')(A'+B'+C+D)(A'+B'+C'+D')$$

These would be maxterms M3, M9, M12, and M15. There is a commonly used, more compact way to express this result:

$$\prod(A, B, C, D) = \prod(3, 9, 12, 15)$$

The above notation would indicate "For the function of A, B, C, D, the output is False for maxterms 3, 9, 12, and 15." This format is called *Pi Notation*, and it is easy to derive the full Boolean equation from it. Remember that M3 is 0011, or  $(A+B+C'+D'$  – the complement of the inputs), M9 is 1001, or  $(A'+B+C+D')$ , M12 is 1100, or  $(A'+B'+C+D)$ , and M15 is 1111, or  $(A'+B'+C+D')$ . The above equation can be quickly created.

An equation that is described using maxterms is often called the "Product of Sums" (or POS) since each term is composed of inputs added together, but the terms are multiplied. The "POS" nature of a maxterm solution is why the Greek letter *Pi* is used to represent this Boolean expression.

### MiniLab 4.3B

Creating a circuit from a maxterm expression is not quite as intuitive as when using minterms.

As an example, consider this expression:  $\prod(A, B, C) = \prod(0, 4, 7)$ . There are three inputs, labeled

A, B, and C. Those inputs feed three OR gates; and the output from those gates goes to an AND gate:

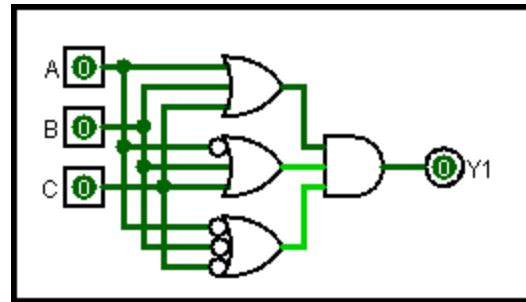


FIGURE 70: CIRCUIT FOR MAXTERMS 0,4,7

Now, create a minterm generator for this expression:  $\sum(A,B,C) = \prod(0,1,3)$ . Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 4.3B", and today's date. Then save the file with this name: "minilab\_4\_3B".

#### MINTERM AND MAXTERM RELATIONSHIPS

The minterms and maxterms of a circuit have three interesting relationships: equivalence, duality, and inverse. To define and understand these terms, consider the following truth table for a sample unspecified circuit:

Inputs			Outputs		Terms	
A	B	C	Q	Q'	minterm	Maxterm
0	0	0	0	1	$A'B'C' (m0)$	$A+B+C (M0)$
0	0	1	1	0	$A'B'C (m1)$	$A+B+C' (M1)$
0	1	0	0	1	$A'BC' (m2)$	$A+B'+C (M2)$
0	1	1	0	1	$A'BC (m3)$	$A+B'+C' (M3)$
1	0	0	0	1	$AB'C' (m4)$	$A'+B+C (M4)$
1	0	1	1	0	$AB'C (m5)$	$A'+B+C' (M5)$
1	1	0	0	1	$ABC' (m6)$	$A'+B'+C (M6)$
1	1	1	1	0	$ABC (m7)$	$A'+B'+C' (M7)$

TABLE 45: MINTERM AND MAXTERM FOR SAMPLE CIRCUIT

#### EQUIVALENCE

The minterms and the maxterms for a given circuit are considered equivalent ways to describe that circuit. For example, the circuit described by the above truth table could be defined using minterms:

$$Q = \sum(1, 5, 7)$$

However, that same circuit could also be defined using maxterms:

$$Q = \prod(0, 2, 3, 4, 6)$$

These two functions describe the same circuit and are, consequently, equivalent. The sigma function includes the terms 1, 5, and 7 while the pi function includes all other terms in the truth table (0, 2, 3, 4, and 6). To put it a slightly different way, the sigma function describes the truth table rows where  $Q = 1$  (minterms) while the pi function describes the rows in the same truth table where  $Q = 0$  (maxterms). Thus:

$$\sum(1, 5, 7) \equiv \prod(0, 2, 3, 4, 6)$$

#### DUALITY

Each row in the above truth table describes two terms that are considered duals. For example, the minterm  $m_5$  ( $AB'C$ ) and the maxterm  $M_5$  ( $A' + B + C'$ ) are duals. In dual terms, the outputs are complements of each other ( $Q$  vs.  $Q'$ ) and the input variables are also complements of each other; moreover, the inputs for the three minterms are combined with an AND, but the maxterms with an OR. The circuit that is described by the above truth table could be defined using minterms:

$$Q = \sum(1, 5, 7)$$

The dual of the circuit would be defined by using the maxterms for the same output rows. However, those rows are actually the complement of the circuit; thus:

$$Q' = \prod(1, 5, 7)$$

This leads to the conclusion that the complement of a sigma function is the pi function with the same inputs. These are considered dual functions:

$$\sum(1, 5, 7) \equiv (\prod(1, 5, 7))'$$

#### INVERSE

The complement of a function yields the opposite output. For example the following functions are inverses because one defines  $Q$  while the other defines  $Q'$  using only minterms of the same circuit (or truth table):

$$\begin{aligned} Q &= \sum(1, 5, 7) \\ Q' &= \sum(0, 2, 3, 4, 6) \end{aligned}$$

Dual functions are also considered inverses since they define  $Q$  and  $Q'$  for the same circuit (truth table).

#### SUMMARY

These three relationships are summarized in the following table. Imagine a circuit with two or more inputs and an output of  $Q$ , as in the “Minterm and Maxterm Summary” table above:

Q Minterms (Minterms where Q is 1)	Q' Minterms (Minterms where Q' is 1)
Q Maxterms (Maxterms where Q is 1)	Q' Maxterms (Maxterms where Q' is 1)

The adjacent items in a single column are equivalent (that is, Q Minterms are equivalent to Q Maxterms), items that are diagonal are duals (Q Minterms and Q' Maxterms are duals), and items that are adjacent in a single row are inverses (Q Minterms and Q' Minterms are inverses).

#### SUM OF PRODUCTS EXAMPLE

##### GIVEN

A gas pipeline into Tucson is fed by three pipelines: Phoenix, El Paso, and San Diego. Design a logic circuit that will control the pipeline by switching on or off the three feeders. If El Paso is open, then Phoenix and San Diego must both be either open or closed. If Phoenix is open, then both El Paso and San Diego must be closed.

##### TRUTH TABLE

When realizing a circuit from a verbal description, the best place to start is constructing a truth table. This will make the Boolean expression easy to write and then make the circuit easy to realize. For the pipeline problem, start by defining variables for the truth table:

- Tucson (the output): T
- Phoenix: P
- El Paso: E
- San Diego: S

Next, construct the truth table by identifying columns for each of the three input variables and then indicate the output for every possible input condition:

Inputs			Output	
P	E	S	T	m
0	0	0	0	0
0	0	1	0	1
0	1	0	1	2
0	1	1	0	3
1	0	0	1	4
1	0	1	0	5
1	1	0	0	6
1	1	1	1	7

TABLE 46: MIINTERM SAMPLE PROBLEM

**BOOLEAN EQUATION**

In a truth table, if there are fewer "True" outputs than "False," then it is easiest to construct a Sum-of-Products equation. In that case, a Boolean expression can be derived by creating the minterms for all "True" outputs and combining those minterms with OR gates.

$$T = (P'E'S') + (PE'S') + (PES)$$

This equation can also be expressed with sigma notation:

$$\sum(P, E, S) = \sum(2, 4, 7)$$

It may be possible to simplify the Boolean expression, but that process is covered in another unit.

**PRODUCT OF SUMS EXAMPLE****GIVEN**

A printer control circuit must be activated to print a red dot on every student application form except when the student is a male who lives in a Greek house and is over 21 years of age or a student who is female who lives in a Greek house and is under 21 years of age.

**TRUTH TABLE**

When realizing a circuit from a verbal description, the best place to start is constructing a truth table. This will make the Boolean expression easy to write and then make the circuit easy to realize. For the printer problem, start by defining variables for the truth table:

- Print Red Dot (the output): P
- Gender: G (will be true for male applicants)
- Age over 21: A (will be true when applicant's age is over 21)
- Residence: R (will be true when the residence is a Greek house)

Next, construct the truth table by identifying columns for each of the three input variables and then indicate the output for every possible input condition:

Inputs			Output	
G	A	R	P	M
0	0	0	1	0
0	0	1	0	1
0	1	0	1	2
0	1	1	1	3
1	0	0	1	4
1	0	1	1	5
1	1	0	1	6
1	1	1	0	7

TABLE 47: EXAMPLE MAXTERM PROBLEM

**BOOLEAN EQUATION**

In the truth table, if there are fewer "False" outputs than "True," then it is easiest to construct a Products-of-Sums equation. In that case, a Boolean expression can be derived by creating the maxterms for all "False" outputs and combining those maxterms with AND gates.

$$P = (G + A + R')(G' + A' + R')$$

This equation can also be expressed with pi notation:

$$\prod(G, A, R) = \prod(1, 7)$$

It may be possible to simplify the Boolean expression, but that process is covered in another unit.

**SUMMARY**

Sum-Of-Products, or SOP, Boolean expressions may be generated from truth tables quite easily, by determining which rows of the table have an output of "True," writing one minterm for each of those rows, and then summing all of the minterms. The resulting expression will lend itself well to implementation as a set of AND gates (products) feeding into a single OR gate (sum).

Product-Of-Sums, or POS, Boolean expressions may be generated from truth tables quite easily, by determining which rows of the table have an output of "False," writing one maxterm for each of those rows, and then multiplying all of the maxterms. The resulting expression will lend itself well to implementation as a set of OR gates (sums) feeding into a single AND gate (product).

## 4.4: CANONICAL FORM

### INTRODUCTION

The word "canonical" simply means "standard" and it is used throughout mathematics and science to denote some standard form for equations. In digital electronics, Boolean equations are considered to be in canonical form when each of the terms in the equation includes all of the possible inputs and those terms appear in the same order as in the truth table. The canonical form is important when simplifying a circuit.

For example, imagine the solution to a given problem generated the following truth table:

Inputs			Output	
A	B	C	Q	m
0	0	0	0	0
0	0	1	1	1
0	1	0	0	2
0	1	1	1	3
1	0	0	0	4
1	0	1	0	5
1	1	0	0	6
1	1	1	1	7

TABLE 48: TRUTH TABLE FOR EXAMPLE PROBLEM

The following minterm equation is derived from the truth table and is presented in canonical form. Notice that each term includes all possible inputs (A, B, and C), and that the terms are in the same order as they appear in the truth table.

$$Q = (A'B'C) + (A'BC) + (ABC)$$

Frequently, though, a Boolean equation is expressed in *standard* form, which is not the same as canonical form. Standard form means that some of the terms have been simplified and not all of the inputs will appear in all of the terms. For example, consider the following equation (which is in canonical form).

$$Q = A'B'CD' + A'B'CD + A'BCD' + A'BCD + AB'CD$$

When simplified, this equation reduces to the following, which is presented in standard form.

$$Q = A'C + B'CD$$

Notice that in standard form, the first term,  $A'C$ , does not include inputs B or D; but all inputs must be present in every term for an equation to be in canonical form. Many simplification techniques (such as Karnaugh Maps) require a Boolean equation to be in canonical form rather than standard form.

Building a truth table using the canonical form of the above equation is easy, 1s are placed in the output column for each of the five terms in the expression.

Inputs				Output	
A	B	C	D	Q	m
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	2
0	0	1	1	1	3
0	1	0	0	0	4
0	1	0	1	0	5
0	1	1	0	1	6
0	1	1	1	1	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	0	10
1	0	1	1	1	11
1	1	0	0	0	12
1	1	0	1	0	13
1	1	1	0	0	14
1	1	1	1	0	15

TABLE 49: TRUTH TABLE FOR SAMPLE PROBLEM

Consider, though, the simplified equation. In what row would  $B'CD$  be placed?  $B'CD$  is 011 (in binary), but since the "A" term is missing, is it considered a 0 or 1; would  $B'CD$  generate an output 1 for row 0011 or 1011? In fact, an output 1 would need to be placed in *both* of those rows to ensure that the truth table is complete.

#### CONVERTING STANDARD TERMS THAT ARE MISSING ONE VARIABLE

To change a standard Boolean expression that is missing one input term into a canonical Boolean expression, insert both True and False for the missing term into the original standard expression. As an example, consider  $B'CD$ . Since term A is missing, both A and  $A'$  must be included in the converted canonical expression. Notice how the equation below expands  $B'CD$  to include both possible values for A.

$$B'CD \rightarrow AB'CD + A'B'CD$$

A term that is missing one input variable will expand into two terms that include all variables. For example, in a system with four input variables (as above), any standard term with only three variables will expand to a canonical expression with two groups of four variables.

Expanding a standard term that is missing one variable can also be done with a truth table. To do that, fill in an output of 1 for every line where the expression is found while ignoring the missing variables. As an example, consider a truth table where  $B'CD$  is marked as true:

Inputs				Output	
A	B	C	D	Q	m
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	2
0	0	1	1	1	3
0	1	0	0	0	4
0	1	0	1	0	5
0	1	1	0	0	6
0	1	1	1	0	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	0	10
1	0	1	1	1	11
1	1	0	0	0	12
1	1	0	1	0	13
1	1	1	0	0	14
1	1	1	1	0	15

TABLE 50: TRUTH TABLE FOR B'CD

Notice that outputs for  $m_3$  and  $m_{11}$  are true because for both of those minterms  $B'CD$  is true (input A is ignored when marking the two minterms). Finally, those two minterms lead to the Boolean expression  $AB'CD + A'B'CD$ .

#### CONVERTING STANDARD TERMS MISSING TWO VARIABLES

It is easiest to expand a standard expression that is missing two terms by first inserting one of the missing variables and then inserting the other missing variable in two distinct steps. The process for inserting a single missing variable is the same as illustrated above. Consider the term  $A'C$  (in a four-variable system). It is missing both the B and D variables. To expand that term to its canonical form, first insert B and  $B'$ :

$$A'C \rightarrow A'BC + A'B'C$$

Then, insert D and  $D'$  into each of the terms formed in the previous step.

$$\begin{aligned} A'BC &\rightarrow A'BCD + A'BCD' \\ A'B'C &\rightarrow A'B'CD + A'B'CD' \end{aligned}$$

In the end,  $A'C$  expands into the following:

$$A'C \rightarrow A'BCD + A'BCD' + A'B'CD + A'B'CD'$$

In a four-variable system, any standard term with only two variables will expand to a canonical expression with four groups of four variables, as shown above.

Expanding a standard term that is missing two variables can also be done with a truth table. To do that, fill in an output of 1 for every line where the expression is found while ignoring the missing variables. As an example, consider a truth table where  $A'C$  is marked as true:

Inputs				Output	
A	B	C	D	Q	m
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	2
0	0	1	1	1	3
0	1	0	0	0	4
0	1	0	1	0	5
0	1	1	0	1	6
0	1	1	1	1	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	0	10
1	0	1	1	0	11
1	1	0	0	0	12
1	1	0	1	0	13
1	1	1	0	0	14
1	1	1	1	0	15

TABLE 51: TRUTH TABLE FOR  $A'C$ 

Notice that outputs for  $m_2$ ,  $m_3$ ,  $m_6$ , and  $m_7$  are true because for each of those minterms  $A'C$  is true (inputs B and D are ignored when marking the four minterms). Finally, those four minterms lead to the Boolean expression  $A'B'CD' + A'B'CD + A'BCD' + A'BCD$ .

## SUMMARY

This discussion started with this standard form equation:

$$Q = A'C + B'CD$$

After expanding both terms, the following Boolean equation is generated:

$$Q = A'BCD + A'BCD' + A'B'CD + A'B'CD' + AB'CD + A'B'CD$$

Notice, though, that the term  $A'B'CD$  appears two times, so one of those can be eliminated (Idempotence property), leaving the following equation:

$$Q = A'BCD + A'BCD' + A'B'CD + A'B'CD' + AB'CD$$

To put the equation in canonical form, which is important for simplification; all that remains is to rearrange the terms so they are in the same order as they would appear in a truth table, which results in the following equation:

$$A = A'B'CD' + A'B'CD + A'BCD' + A'BCD + AB'CD$$

## PRACTICE PROBLEMS

The following problems are presented as practice in creating a Canonical Form from a Standard Form.

<b>1</b>	Standard Form	$A'B + C + AB'$
	Canonical Form	$A'BC + A'BC' + A'B'C + AB'C + ABC + AB'C'$
<b>2</b>	Standard Form	$A(B' + C)$
	Canonical Form	$(A + B + C)(A + B + C')(A + B' + C)(A + B' + C')(A' + B' + C)$

TABLE 52: CANONICAL FORM EXERCISES

## 4.5: SIMPLIFICATION USING ALGEBRAIC METHODS

### INTRODUCTION

One method of simplifying a Boolean equation is to use common algebraic processes. It is possible to reduce an equation step-by-step using the various properties of Boolean algebra, in the same way that real-number equations can be simplified.

### STARTING FROM A CIRCUIT

Occasionally, the circuit designer is faced with an existing circuit and must attempt to simplify it. In that case, the first step is to find the Boolean equation for the circuit. In the circuit below, the "A," "B," and "C" input signals are assumed to be provided from switches, sensors, or perhaps other circuits. Where these signals originate is of no concern in the task of gate reduction.

### GENERATE THE BOOLEAN EQUATION

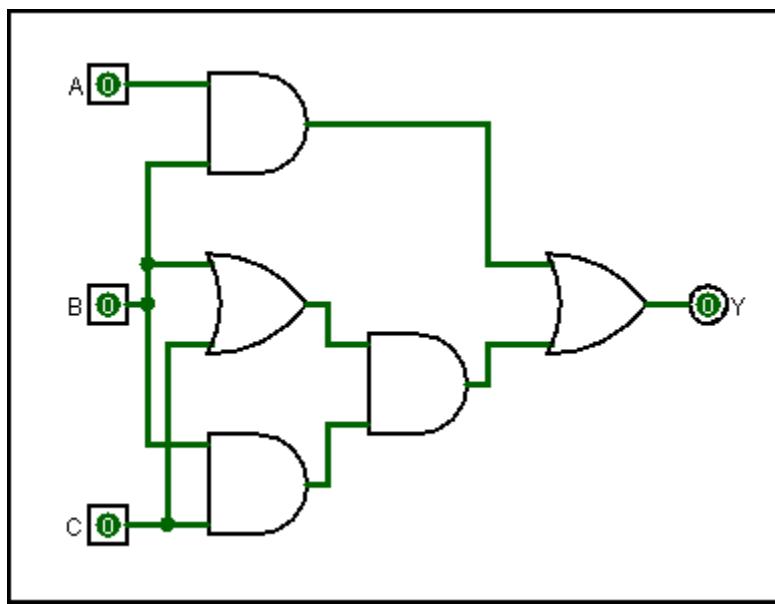


FIGURE 71: SAMPLE CIRCUIT TO SIMPLIFY

To generate the Boolean equation for a circuit, write the output of each gate as referenced by the input signals for that gate, starting with the gates that have the variable inputs (A, B, and C in the example). Working from the inputs to the final output, normally that would be left to right, the Boolean expression for each gate is noted. Finally, the output, Y, is determined.

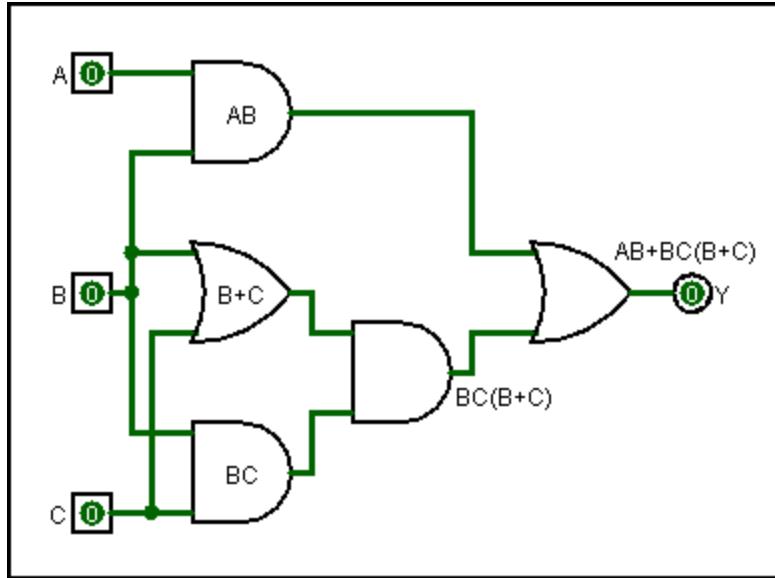


FIGURE 72: SAMPLE CIRCUIT WITH GATE OUTPUTS IDENTIFIED

$$Y = AB + BC(B+C)$$

#### SIMPLIFY THE EQUATION

Now that there is a Boolean equation for the circuit, apply the various properties of Boolean algebra to reduce the equation to its simplest form ("simplest" is defined as requiring the fewest gates to implement):

$AB + BC(B+C)$	Original Term
$AB + BBC + BCC$	Distribute BC
$AB + BC + BC$	Idempotence: $BB=B$ and $CC=C$
$AB + BC$	Idempotence: $BC+BC = BC$
$B(A+C)$	Factor

TABLE 53: SIMPLIFYING A BOOLEAN EQUATION

The final expression,  $B(A + C)$ , is much simpler than the original, yet performs the same function. To verify this, realize both circuits with a simulator and then check to see if the truth table is the same for both circuits.

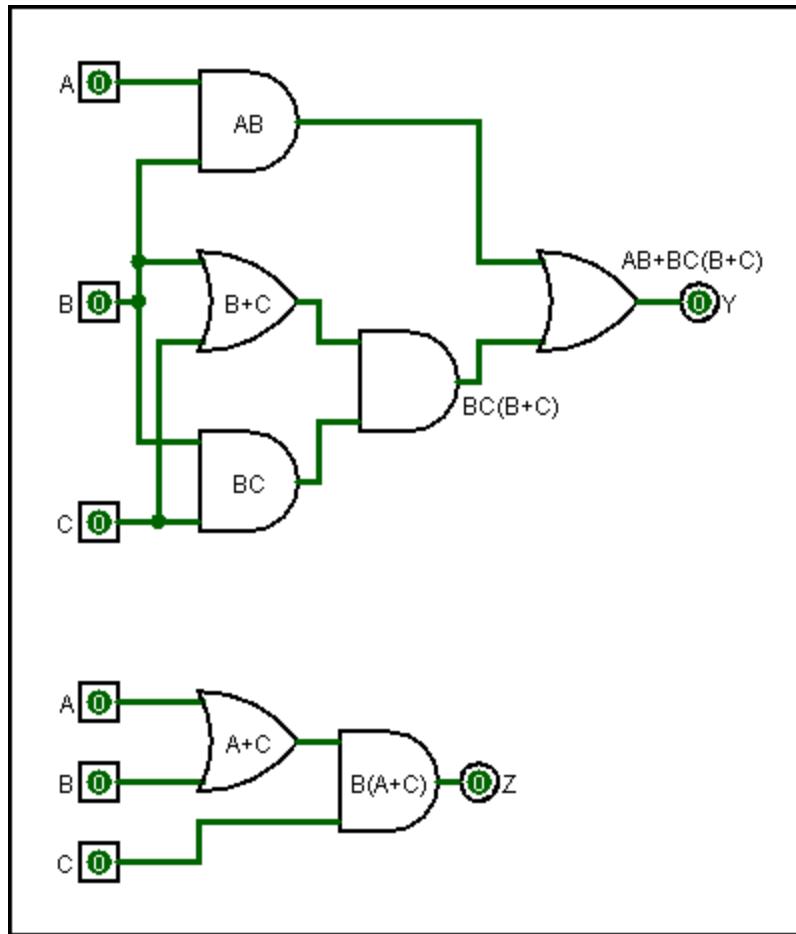


FIGURE 73: ORIGINAL AND SIMPLIFIED CIRCUITS

Obviously, the second circuit is much simpler than the original, having only two logic gates instead of five; yet the output at Y is the same as Z. Such component reduction results in higher operating speed (less gate propagation delay), less power consumption, less cost to manufacture, and greater reliability.

#### STARTING FROM A BOOLEAN EQUATION

If a logic circuit's function is expressed as a Boolean equation, then algebraic methods can be applied to reduce the number of logic gates, resulting in a circuit that performs the same function with fewer components. As an example, imagine that a circuit has this Boolean equation:

$$Y = A + AB$$

$A + AB$	Original Term
$A(1+B)$	Factor A
$A(1)$	Annihilation ( $1 + B = 1$ )
A	Identity $A1 = A$

TABLE 54: SIMPLIFYING A BOOLEAN EQUATION

The expression on the left has been reduced to simply "A" so it could be replaced by a wire from input A to output Y; thus The entire original circuit is redundant and can be replaced by a wire. The next circuit

simplification looks similar to one presented above, but is actually quite different and requires a more clever proof.

$$Y = A + A'B$$

$A + A'B$	Original Term
$A + AB + A'B$	Expand A to A + AB
$A + B(A + A')$	Factor
$A + B(1)$	Identity of A+A'
$A + B$	Simplify

TABLE 55: SIMPLIFYING A BOOLEAN EQUATION

Note how the Absorption Property ( $A + AB = A$ ) is used to "un-simplify" the first "A" term in step one, changing "A" into "A + AB". While this may seem like a backward step, it ultimately helped to reduce the expression to something simpler. Sometimes "backward" steps must be taken to achieve the most elegant solution. Knowing when to take such a step and when not to is part of the art of algebra.

As another example, simplify this product-of-sums expression equation:

$$Y = (A+B)(A+C)$$

$(A+B)(A+C)$	Original Term
$AA + AC + AB + BC$	Distribute A + B
$A + AC + AB + BC$	Idempotence: AA = A
$A + AB + BC$	Absorption: A + AC = A
$A + BC$	Absorption: A + AB = A

TABLE 56: SIMPLIFYING A BOOLEAN EQUATION

In each of the examples in this section, a Boolean expression was simplified, which led to a reduction in the number of gates needed and made the final circuit more economical to construct and reliable to operate.

#### EXAMPLES FOR PRACTICE

The following table shows a Boolean expression on the left and its simplified version on the right. This is provided for practice in simplifying expressions using algebraic methods.

Original Expression	Simplified
$A(A' + B)$	$AB$
$A + A'B$	$A + B$
$(A+B)(A+B')$	$A$
$AB + A'C + BC$	$AB + A'C$

TABLE 57: EXERCISES FOR SIMPLIFYING A BOOLEAN EQUATION

## 4.6: KARNAUGH MAPS

### INTRODUCTION

A Karnaugh map, like Boolean algebra, is a tool used to simplify a digital circuit. Keep in mind that "simplify" means reducing the number of gates and inputs; and as components are eliminated, not only does the cost go down, but the circuit becomes simpler, more stable, and more energy efficient.

!	Maurice Karnaugh, a telecommunications engineer, developed the Karnaugh Map at Bell Labs in 1953 while designing digital logic based telephone switching circuits.
---	--

In general, using Boolean Algebra is the easiest way to simplify a circuit involving one to three input variables. For four input variables, Boolean algebra becomes tedious and Karnaugh maps are both faster and easier (and are less prone to error). However, Karnaugh maps become rather complex with five input variables, and are generally too difficult to use above six variables (with five input variables, a Karnaugh map uses multiple dimensions that become very challenging to manipulate). For five or more input variables, circuit simplification should be done by Quine-McClusky methods or the use of Computer Aided Tools (CAT).

Variables	Boolean Algebra	Karnaugh Map	Quine-McClusky	CAT
1-2	X			
3	X	X		
4	X	X		
5-6		X	X	
7-8			X	X
>8				X

TABLE 58: CHOOSING A SIMPLIFICATION TOOL

In theory, any of the methods will work for any number of variables; however, as a practical matter, the guidelines presented in the above table work well. Normally, there is no need to resort to computer tools to simplify a simple equation involving two or three variables since it is much quickly to use either Boolean Algebra or Karnaugh Maps. However, for more complex input/output combinations, then computer tools become essential to both speed the process and improve accuracy.

### KARNAUGH MAPS, TRUTH TABLES, AND BOOLEAN EXPRESSIONS

Consider the following circuit:

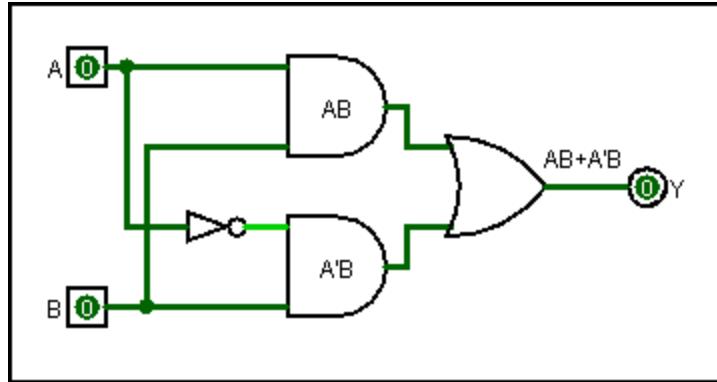


FIGURE 74: SIMPLE CIRCUIT FOR KARNAUGH MAP

$$Y = AB + A'B$$

EQUATION 1: BOOLEAN EQUATION FOR SIMPLE CIRCUIT

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	0
1	1	1

TABLE 59: TRUTH TABLE FOR SIMPLE CIRCUIT

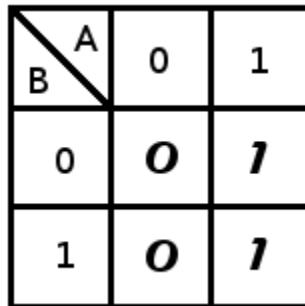


FIGURE 75: KARNAUGH MAP FOR SIMPLE CIRCUIT

Above are four different ways of representing the same thing: a 2-input digital logic function. First is the circuit diagram, followed by a Boolean equation, a truth table, and, finally, a Karnaugh map. Think of a Karnaugh map as simply a rearranged truth table; but simplifying a three or four input circuit using a Karnaugh map is much easier and more accurate than with either a truth table or Boolean equation.

#### DRAWING KARNAUGH MAPS FOR TWO INPUT VARIABLES

Following is a truth table and a Karnaugh map. Greek symbols were used as outputs to emphasize the relationship between these two representations of the same data.

A	B	Q
0	0	$\alpha$
0	1	$\beta$
1	0	$\gamma$
1	1	$\delta$

TABLE 60: TRUTH TABLE WITH GREEK LETTERS

	A	0	1
B			
0		$\alpha$	$\beta$
1		$\gamma$	$\delta$

FIGURE 76: KARNAUH MAP WITH GREEK LETTERS

On the Karnaugh map, all of the possible values for input "A" are listed across the top of the map and values for input "B" are listed down the left side. Thus, to find the output for A=0 and B=0, look for the cell where those two quantities intersect; which is output  $\alpha$  in the example above. You should be able to see how all four data squares on the Karnaugh map correspond to their equivalents in the Truth Table.

Here's another example:

A	B	Q
0	0	1
0	1	1
1	0	0
1	1	1

TABLE 61: TRUTH TABLE FOR 2-INPUT CIRCUIT

	A	0	1
B			
0	00	02	1
1	01	03	1

FIGURE 77: KARNAUH MAP FOR 2-INPUT CIRCUIT



**Note:** Karnaugh maps frequently include the minterm numbers in each cell to aid in properly placing variables. Also, zeros are usually not shown on a Karnaugh map in order to make simplification quicker and less prone to error.

### DRAWING KARNAUGH MAPS FOR THREE INPUT VARIABLES

Consider this Boolean equation:

$$Q = ABC' + AB'C + A'B'C$$

Following is the truth table and Karnaugh map for this equation:

Inputs			Output	minterm
A	B	C	Q	
0	0	0	0	0
0	0	1	1	1
0	1	0	0	2
0	1	1	0	3
1	0	0	0	4
1	0	1	1	5
1	1	0	1	6
1	1	1	0	7

TABLE 62: TRUTH TABLE FOR BOOLEAN EQUATION

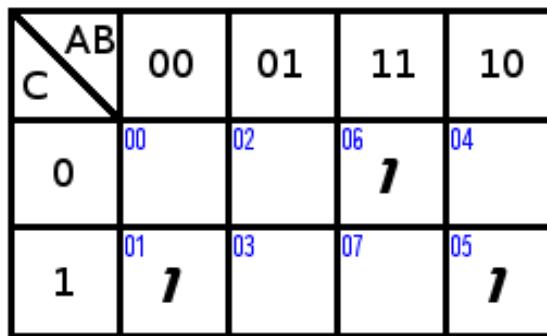


FIGURE 78: KARNAUGH MAP FOR SAMPLE BOOLEAN EQUATION

Notice in the Karnaugh map how all possible values for inputs A and B are listed across the top of the map, while input C is listed down the left side. Therefore, minterm 05, in the lower right corner of the Karnaugh map, is for A=1, B=0, and C=1, ( $AB'C$ ), one of the “True” terms in the original equation.

### THE GRAY CODE

You should also note that the values across the top of the Karnaugh map are not in binary order. Instead, those values are in "Gray Code" order. Gray code is essential for a Karnaugh map since the values for adjacent cells must change by only one bit. Constructing the Gray Code for three, four, and five variables is covered elsewhere in this course; however, for the Karnaugh maps used in this lesson, it is enough to know the 2-bit Gray code: 00, 01, 11, 10.

## DRAWING KARNAUGH MAPS FOR FOUR INPUT VARIABLES

Consider the following Boolean equation:

$$Q = ABCD' + AB'CD + A'B'CD$$

Following is the truth table and Karnaugh map for this equation:

Inputs				Output	minterm
A	B	C	D	Q	
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	2
0	0	1	1	1	3
0	1	0	0	0	4
0	1	0	1	0	5
0	1	1	0	0	6
0	1	1	1	0	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	0	10
1	0	1	1	1	11
1	1	0	0	0	12
1	1	0	1	0	13
1	1	1	0	1	14
1	1	1	1	0	15

TABLE 63: TRUTH TABLE FOR 4-INPUT CIRCUIT

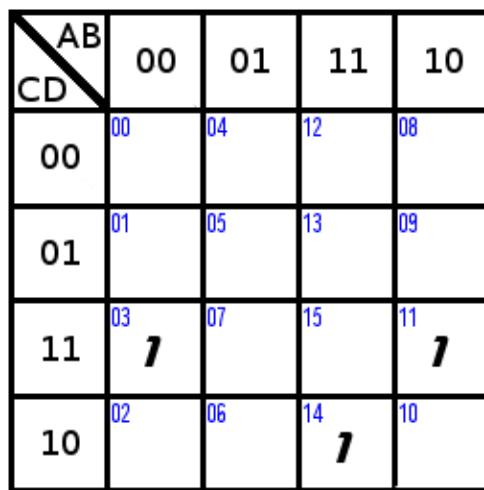


FIGURE 79: KARNAUGH MAP FOR 4-INPUT CIRCUIT

This Karnaugh map is similar to those for two and three variables, but the top row is for the A and B inputs while the left column is for the C and D inputs. Notice that the values in both the top row and left column use Gray Code sequencing rather than binary counting.

It is easy to place the ones on a Karnaugh map if sigma notation is available since the numbers following the sigma sign are the minterms. As an example, map this function:  $\sum(A,B,C,D) = \sum(0,1,2,4,5)$ :

AB CD	00	01	11	10
00	00 <b>1</b>	04 <b>1</b>	12 08	
01	01 <b>1</b>	05 <b>1</b>	13 09	
11	03 02	07 06	15 14	11 10
10	<b>1</b>			

FIGURE 80: MAP FOR SIGMA(0,1,2,4,5)

It is also possible to map values if the circuit is represented in pi notation; but remember that maxterms indicate where zeros are placed on the Karnaugh map and the simplified circuit would actually be the inverse of the needed circuit. As an example, map this function:  $\prod(A,B,C,D) = \prod(8,9,12,13)$ :

AB CD	00	01	11	10
00	00 01	04 05	12 13	08 09
01			<b>O</b> <b>O</b>	
11	03 02	07 06	15 14	11 10
10				

FIGURE 81: MAP FOR PI(8,9,12,13)

To simplify this map, the designer could place ones in all of the empty cells and then simplify the “ones” circuit using the techniques explained below. Alternatively, the designer could also simplify the map by combining the zeros as if they were ones, and then finding the DeMorgan inverse of that simplification.

As an example, the map above would simplify to  $AC'$  and the DeMorgan equivalent for that is  $A'+C$  ; which is the simplified circuit.

#### SIMPLIFYING 4-INPUT EQUATIONS (GROUPS OF 2)

To simplify a Boolean equation using a Karnaugh map, start by creating the Karnaugh map, indicating the input variable combinations that lead to a True output. Here is an example.

$$Q = A'B'C'D' + A'BC'D' + A'BCD + AB'CD' + AB'CD$$

This equation can also be represented with the following sigma notation:

$$\sum(A,B,C,D) = \sum(0,4,7,10,11)$$

		AB	00	01	11	10
		CD	00	04	12	08
		00	1	1		
		01	01	05	13	09
		11	03	07	15	11
		10	02	06	14	10

FIGURE 82: KARNAUGH MAP FOR SAMPLE 4-INPUT PROBLEM

Once the True outputs are indicated, circle any groups of 1's that are adjacent to each other, either horizontally or vertically (but not diagonally). Also, circle any 1's that are "left over" and are not adjacent to any other 1's.

AB \ CD	00	01	11	10
00	00 1	04 1	12	08
01	01	05	13	09
11	03	07 1	15	11 1
10	02	06	14	10 1

Notice the group in the top-left corner (minterms 00-04). This group includes the following two input variable combinations:  $A'B'C'D' + A'BC'D'$ . In this expression, the B and B' terms can be removed due to the Complement Property; so this group reduces to  $A'C'D'$ . To simplify this expression by simply inspecting the Karnaugh map, notice that the variable A is zero for both of these two minterms. Therefore, A' must be part of the final expression. In the same way, variables C and D are 0 for both terms; therefore, C'D' must be part of the final expression. Since variable B changes, it can be ignored when forming the simplified expression.

The group in the lower-right corner (minterms 10-11) includes the following two input variable combinations:  $AB'CD + AB'CD'$ . The D and D' terms can be removed due to the Complement Property; so this group simplifies to  $AB'C$ . Again, inspecting these two terms would reveal that the variables  $AB'C$  do not change between the two terms, so they must appear in the final expression.

Minterm 07, the lone term circled in column two, cannot be reduced since it is not adjacent to any other ones. Therefore, it must go into the simplified equation unchanged:  $A'BCD$ .

When finished, the original equation reduces to:

$$Q = A'C'D' + AB'C + A'BCD$$

Using a Karnaugh map, the circuit was simplified from four 4-input AND gates to two 3-input AND gates and one 4-input AND gate.

The various ones on a Karnaugh Map are called the **Implicants** of the solution. These are the algebraic products that are necessary to "imply" (or bring about) the final simplification of the circuit. When an implicant cannot be grouped with any others, or when two or more implicants are grouped together, they are called **Prime Implicants**. The three groups ( $A'C'D'$ ,  $AB'C$ , and  $A'BCD$ ) found by analyzing the Karnaugh Map above are the prime implicants for this equation. When prime implicants are a necessary part of the final simplified equation, and they are not subsumed by any other implicants, they are called **Essential Prime Implicants**. For the simple example given above, all of the prime implicants are

essential; however, more complex Karnaugh Maps may have numerous prime implicants that are subsumed by other implicants; thus, are not essential. There are examples of these types of maps below.

Here's a second example Karnaugh map.

$$Q = A'B'C'D + A'B'CD + A'BCD' + ABC'D + ABCD' + AB'C'D'$$

Or:

$$\sum(A,B,C,D) = \sum(1,3,6,8,13,14)$$

		AB	00	01	11	10
		CD	00	04	12	08
		00	00			1
		01	01	1	05	13
		11	03	1	07	15
		10	02	06	14	1

All groups of adjacent ones have been circled, so this circuit can be simplified by looking for groups of two. Starting with minterms 1 and 3,  $A'B'C'D + A'B'CD$  simplifies to  $A'B'D$ . Minterms 6 and 14 simplifies to  $BCD'$ . The other two circled minterms are not adjacent to any other ones, so they cannot be simplified. Each of the circled terms are prime implicants; and since they are not subsumed by any other implicants, they are Essential Prime Implicants. The simplified equation is:

$$Q = ABC'D + AB'C'D' + A'B'D + BCD'$$

#### SIMPLIFYING 4-INPUT EQUATIONS (LARGER GROUPS)

When simplifying Karnaugh maps, it is most efficient to find groups of 16, 8, 4, or 2 adjacent ones, in that order. In general, the larger the group, the simpler the expression becomes; so one large group is preferable to two smaller groups. However, remember that any group can only use ones that are adjacent along a horizontal or vertical line, not diagonal.

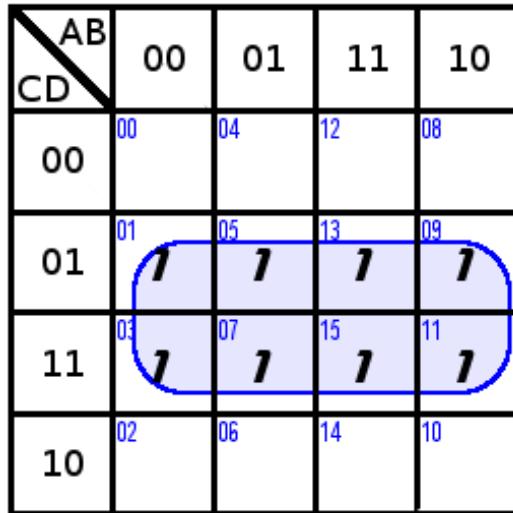
#### GROUPS OF 16

Groups of 16 reduce to a constant output of one. This is because if a circuit is built such that every possible combination of four inputs yields a True output, then the circuit is unnecessary and can be replaced by a wire. There is no example Karnaugh map posted here to illustrate a circuit like this; but if

every cell in a Karnaugh map contains a one, then the circuit is unnecessary. By the same token, any Karnaugh map that contains only zeros indicates that the circuit would never output a true condition; so the circuit is unnecessary.

#### GROUPS OF 8

Groups of 8 simplifies to a single output variable. Consider the following Karnaugh map:



The expression for row two is:  $A'B'C'D + A'BC'D + ABC'D + AB'C'D$ . The term  $C'D$  is constant in this group, while  $A$  and  $B$  change, so this one line would simplify to  $C'D$ . The expression for row three is:  $A'B'CD + A'BCD + ABCD + AB'CD$ . The term  $CD$  is constant in this group, so this one line would simplify to  $CD$ . Then, combining the two rows:  $C'D + CD$  simplifies the output to  $D$ . Since the only term in this group that does not change is  $D$ , then the final equation for the circuit is:

$$Q=D$$

This Karnaugh Map also provides a good example of prime implicants that are not essential. Consider row two of the map. That entire row is a Prime Implicant for this circuit since it is a group of four; however, that row was subsumed by the group of eight that was formed with the next row. Since every cell in the group of four is also present in the group of eight, then the group of four is not essential to the final circuit simplification. While this may seem to be rather obvious, it is important to remember that frequently implicants are formed that are not essential, and they can be ignored. This concept will come up again when using the Quine-McCluskey Simplification method, as presented in a later unit.

#### GROUPS OF 4

Groups of 4 can form as a single row, a single column, or a square. In any case, the four cells will simplify to a 2-variable expression. Consider the following examples:

AB CD	00	01	11	10
00	00	04	12	08
01	01	05	13	09
11	03	07	15	11
10	02	06	14	10

Since the A and B variables can be removed due to the Complement Property, the above circuit simplifies to:

$$Q = C'D$$

Here is another example:

AB CD	00	01	11	10
00	00	04	12	08
01	01	05	13	09
11	03	07	15	11
10	02	06	14	10

Since the C and D variables can be removed due to the Complement Property, the above circuit simplifies to:

$$Q = AB$$

AB \ CD	00	01	11	10
00	00	04	12	08
01	01	05	13	09
11	03	07	15	11
10	02	06	14	10

Since the B and C variables could be removed by using the Complement Property, the above circuit simplifies to:

$$Q = A'D$$

#### GROUPS OF 2

Groups of 2 will simplify to three variables. Here are two examples.

AB \ CD	00	01	11	10
00	00	04	12	08
01	01	05	13	09
11	03	07	15	11
10	02	06	14	10

Since the C is the only variable that can be removed due to the Complement Property, the above circuit simplifies to:

$$Q = AB'D$$

AB \ CD	00	01	11	10
00	00	04	12	08
01	01	05	13	09
11	03	07	15	11
10	02	06	14	10

Since the A is the only variable that can be removed due to the Complement Property, the above circuit simplifies to:

$$Q = BCD$$

#### OVERLAPPING GROUPS

Frequently, groups overlap to create numerous patterns on the Karnaugh map. Consider the following two examples.

AB \ CD	00	01	11	10
00	00	04	12	08
01	01	05	13	09
11	03	07	15	11
10	02	06	14	10

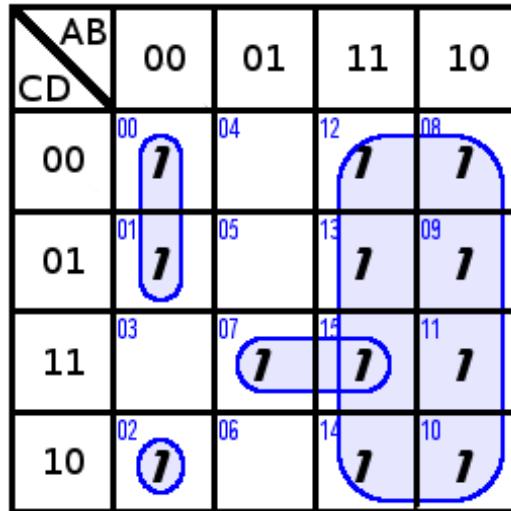
The one in cell A'BCD (minterm 7) can be grouped with either the horizontal or vertical group (or both). This creates the following three potential circuits:

- Group minterms 05-07 with a separate minterm (15):  $Q = A'BD + ABCD$
- Group minterms 07-15 with a separate minterm (05):  $Q = BCD + A'BC'D$
- Two groups of two minterms (05-07 and 07-15):  $Q = A'BD + BCD$

In general, it would be considered simpler to have two 3-input AND gates rather than one 3-input AND gate and one 4-input AND gate, so the last grouping option would be chosen. The designer always chooses whatever grouping yields the smallest number of gates and the smallest number of inputs per gate. The equation for the simplified circuit is:

$$Q = A'B'D + BCD$$

Here is a more complex example.



The circuit represented by this Karnaugh Map would simplify to:

$$Q = A + A'B'C' + BCD + A'B'CD'$$

#### WRAPPING GROUPS

A Karnaugh map also “wraps” around the edges (top/bottom and left/right), so groups can be formed around the borders. It is almost like the Karnaugh map is on some sort of weird sphere where every edge touches the edge across from it (but not diagonal corners). Here are two examples.

AB \ CD	00	01	11	10
00	00	04	12	08
01	01	1	13	09
11	03	07	15	11
10	02	06	14	10

The two ones on this map can be grouped "around the edge" to form this equation:

$$Q = B'C'D$$

Here is one other example of wrapping:

AB \ CD	00	01	11	10
00	1	04	12	08
01	01	05	13	09
11	03	07	15	11
10	02	06	14	10

The ones on the above map can be formed into a group of four and simplify into:

$$Q = B'D'$$

#### KARNAUGH MAPS FOR 5-VARIABLE INPUTS

It is possible to create a Karnaugh map for circuits with five input variables; however, the map must be simplified as if it were a 3-dimensional object so it is more complex than the maps described above. As an example, imagine that a circuit is defined by this equation:

$$\sum(A, B, C, D, D) = \sum(0, 9, 13, 16, 25, 29)$$

The Karnaugh map for this circuit needs five input variables:

		ABC	000	001	011	010	100	101	111	110
		DE	00	04	12	08	16	20	28	24
		00	1				1			
		01	01	05	13	1	1		1	1
		11	03	07	15	11	19	23	31	27
		10	02	06	14	10	18	22	30	26

FIGURE 83: 5-VARIABLE KARNAUH MAP

This map lists variables A, B, and C across the top row with D and E down the left column. Variables B and C are in Gray Code order, and variable A is zero on the left side of the map and one on the right. To simplify the circuit, the map must be turned into a three-dimensional item; so imagine that the map is cut off along the heavy line between minterms 08 and 16 (where variable A changes from zero to one) and then the right half slides under the left half in such a way that cell 16 ends up directly beneath cell 00. Those two cells then form a group of two since they both contain ones. That group would be  $A'B'C'D'E' + AB'C'D'E'$ . Since variable A and  $A'$  are both present in this expression, and none of the other variables change, it can be simplified to:  $B'C'D'E'$ . This process is exactly the same as for a two-dimension Karnaugh map, except that adjacent cells include those above or below each other.

Next, consider the group formed by minterms 09, 13, 25, and 29. The expression for that group is  $A'BC'D'E + A'BCD'E + ABC'D'E + ABCD'E$ . The variables A and C can be removed from the simplified expression by the Complement Property, leaving:  $BD'E$  for this group.

The final simplified expression for this circuit is:

$$(B'C'D'E') + (BD'E)$$

Here is a more complex example:

ABC \ DE		000	001	011	010	100	101	111	110
00	00 1	04	12	08	16	20	28	24 1	
01	01	05 1	13 1	09	17	21	29 1	25	
11	03	07	15 1	11	19	23	31 1	27 1	
10	02 1	06	14	10	18	22	30	26	

FIGURE 84: 5-VARIABLE KARNAUGH MAP

On the above map, the largest group would be minterms 13-15-29-31, so they should be combined first. Minterms 05-13, 27-31, and 00-02 would form two groups of two. Finally, even though Karnaugh maps wrap around the edges, minterm 00 will not group with 24 since they are on different layers (notice that two bits, A and B, change between those two minterms, so they are not adjacent); therefore, minterms 00 and 24 will not group with any other minterms. This is the simplified expression for the circuit:

$$(A'B'C'E') + (A'CD'E) + (BCE) + (ABDE) + (ABC'D'E')$$



It is possible to simplify a 6-input circuit with a Karnaugh map, but that becomes quite challenging since the map must be simplified in four dimensions.

#### “DON’T CARE” TERMS

Occasionally, a circuit designer will run across a situation where the output for a particular minterm makes no difference in the circuit; so that minterm is considered “don’t care;” that is, it can be either one or zero without having any effect on the entire circuit. As an example, consider a circuit that is designed to work with Binary Coded Decimal values. In that system, minterms 10-15 do not exist, so they would be considered “don’t care.” On a Karnaugh map, “don’t care” terms are indicated using several different methods, but the two most common are a dash or an “x”. When simplifying a Karnaugh map that contains don’t care values, the designer can choose to consider those values as either one or zero, whichever makes simplifying the map easier. Consider the following Karnaugh map:

AB \ CD	00	01	11	10
00	00	04	12	08
01	01	05	13	09
11	03	07	15	11
10	02	06	14	10

FIGURE 85: KARNAUGH MAP WITH DON'T CARE TERMS

On this map, minterm 13 is “don’t care.” Since it could form a group of 4 with minterms 08, 09, and 12, and since groups of 4 are preferable to two groups of 2, then minterm 13 should be considered a “true” and grouped with the other three minterms. However, minterm 02 does not group with anything, so it should be considered a “false” and it would then be removed from the simplified expression altogether. This Karnaugh map would simplify to  $AC'$ .

Here is another example circuit with “don’t care” terms:

AB \ CD	00	01	11	10
00	00	04	12	08
01	01	05	13	09
11	1	1	-	
10	02	06	14	10

FIGURE 86: KARNAUGH MAP WITH DON'T CARE TERMS

This circuit would simplify to:

$$(ACD') + (BC) + (A'BD) + (A'CD)$$

**KARNAUGH MAP SIMPLIFICATION SUMMARY**

Here are the rules for simplifying a Boolean equation using a 4-variable Karnaugh map:

1. Create the map and plot all 1's from the truth table output.
2. Circle all groups of 16. These will reduce to a constant output of 1 and the entire circuit is unnecessary.
3. Circle all groups of 8. These will reduce to one variable.
4. Circle all groups of 4. These will reduce to two variables. The ones can be either horizontal, vertical, or in a square.
5. Circle all groups of 2. These will reduce to three variables. The ones can be either horizontal or vertical.
6. Circle all 1's that are not in any other group. These do not reduce and will be a 4-variable expression.
7. All ones must be circled at least one time.
8. Groups can overlap.
9. If ones are in more than one group, they can be considered part of either group.
10. Groups can wrap around the edges to the other side of the map.

**PRACTICE PROBLEMS**

The following problems are presented as practice for using a Karnaugh Map to simplify a Boolean expression.

<b>1</b>	Expression (A,B)	$A'B + AB' + AB$
	Simplified	$A + B$
<b>2</b>	Expression (A,B,C)	$A'BC + AB'C' + ABC' + ABC$
	Simplified	$BC + AC'$
<b>3</b>	Expression (A,B,C)	$A'C + A'B + AB'C + BC$
	Simplified	$C + A'B$
<b>4</b>	Expression (A,B,C,D)	$A'B'C' + B'CD' + A'BCD' + AB'C'$
	Simplified	$B'D' + B'C' + A'CD'$
<b>5</b>	Expression	$\int(A, B, C, D) = \sum(0, 1, 6, 7, 12, 13)$
	Simplified	$A'B'C' + ABC' + A'BC$
<b>6</b>	Expression	$\int(A, B, C, D) = \prod(0, 2, 4, 10)$
	Simplified	$D + BC + AC'$

TABLE 64: KARNAUGH MAP EXERCISES

## 4.7: REED-MÜLLER LOGIC

### INTRODUCTION

Irving Reed and D.E. Müller are noted for inventing various codes that self-correct transmission errors in the field of digital communications. However, they also formulated ways of simplifying digital logic expressions that do not easily yield to traditional methods, such as a Karnaugh Map where the ones form a checkerboard pattern.

Consider the following truth table:

Inputs		Output
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 65: TRUTH TABLE FOR CHECKERBOARD PATTERN

This pattern is easy to recognize as an XOR gate. The Karnaugh Map for the above Truth Table looks like this (the zeros have been omitted in the data cells and the cells with ones have been shaded to emphasize the checkerboard pattern):

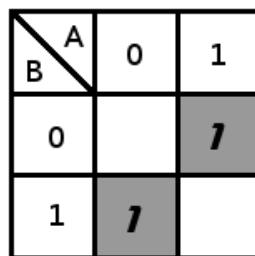


FIGURE 87: KARNAUGH MAP SHOWING CHECKERBOARD PATTERN

The equation for this circuit would be:

$$Q = AB' + A'B$$

This equation cannot be simplified using common Karnaugh Map simplification techniques since none of the ones are adjacent vertically or horizontally. However, whenever a Karnaugh Map displays a checkerboard pattern, the circuit described can be simplified using XOR or XNOR gates.

### MAPS WITH ZERO IN THE FIRST CELL

Karnaugh Maps with a zero in the first cell (that is, in a four-variable map,  $A'B'C'D'$  is False) are simplified in a slightly different manner than those with a one in that cell. This section describes the technique used for maps with a zero in the first cell, while the next section describes the technique for maps with a one in the first cell.

With a zero in the first cell, the equation for the Karnaugh Map is generated by:

1. Grouping the ones into horizontal, vertical, or square groups if possible
2. Identifying the variable in each group that is True and does not change
3. Combining those groups with an XOR gate

#### TWO-VARIABLE CIRCUIT

In the Karnaugh Map presented in the Introduction above, it is not possible to group the ones. Since both A and B are True in at least one cell, the equation for that circuit is:

$$Q = A \oplus B$$

#### THREE-VARIABLE CIRCUIT

Here is a Karnaugh Map for a circuit containing three variable inputs:

		AB	00	01	11	10	
		C	0		1		1
		0					
		1	1			1	
		0					
		1	1	1			

FIGURE 88: KARNAUGH MAP SHOWING CHECKERBOARD PATTERN

In this Karnaugh Map, it is not possible to group the ones. Since A, B, and C are all True in at least one cell, the equation for the circuit is:

$$Q = A \oplus B \oplus C$$

Following is a more interesting 3-variable Karnaugh Map:

		AB	00	01	11	10	
		C	0			1	1
		0					
		1	1	1			
		0					
		1	1	1			

FIGURE 89: KARNAUGH MAP SHOWING CHECKERBOARD PATTERN

In this Karnaugh Map, the ones form two groups in a checkerboard pattern. If this map were to be simplified using common techniques, the equation would be:

$$Q = AC' + A'C$$

If realized, this circuit would contain two 2-input AND gates joined by a 2-input OR gate. However, this equation can be simplified using the Reed-Müller technique. For group in the upper right corner, A is a constant one; and, for the group in the lower left corner, C is a constant one. The equation becomes:

$$Q = A \oplus C$$

If realized, this circuit would contain nothing more than a 2-input XOR gate; and is much simpler than the first attempt.

#### FOUR-VARIABLE CIRCUIT

Here is a Karnaugh Map for a circuit containing four variable inputs:

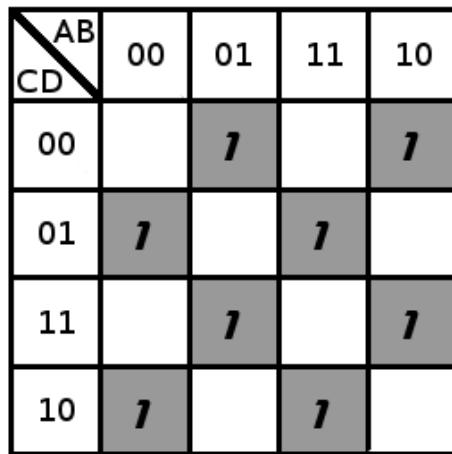


FIGURE 90: KARNAUH MAP SHOWING CHECKERBOARD PATTERN

In this Karnaugh Map, it is not possible to group the ones. Since A, B, C, and D are all True in at least one cell, the equation for the circuit is:

$$Q = A \oplus B \oplus C \oplus D$$

Following are more interesting 4-variable Karnaugh Maps with groups of ones:

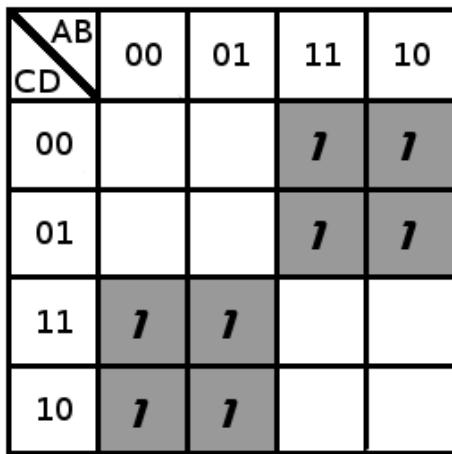


FIGURE 91: KARNAUH MAP SHOWING CHECKERBOARD PATTERN

In this Karnaugh Map, the group of ones in the upper right corner has a constant one for A and the group in the lower left corner has a constant one for C, so the equation for this circuit is:

$$Q = A \oplus C$$

		AB	00	01	11	10
		CD				
		00		1		1
		01		1		1
		11		1		1
		10		1		1

FIGURE 92: KARNAUUGH MAP SHOWING CHECKERBOARD PATTERN

In this Karnaugh Map, the group of ones in the first shaded column has a constant one for B and in the second shaded column have a constant one for A, so the equation for this circuit is:

$$Q = A \oplus B$$

		AB	00	01	11	10
		CD				
		00		1	1	
		01	1			1
		11	1			1
		10		1	1	

FIGURE 93: KARNAUUGH MAP SHOWING CHECKERBOARD PATTERN

Keep in mind that groups can wrap around the edges in a Karnaugh Map. The group of ones in columns 1 and 4 combine and has a constant one for D. The group of ones in rows 1 and 4 combine and has a constant one for B, so the equation for this circuit is:

$$Q = B \oplus D$$

It is interesting that all of the above examples used XOR gates to combine the constant True variable found in groups of ones; however, it would yield the same result if XOR gates combined the constant

False variable found groups of ones. The designer could choose either, but must be consistent. For example, consider this Karnaugh Map:

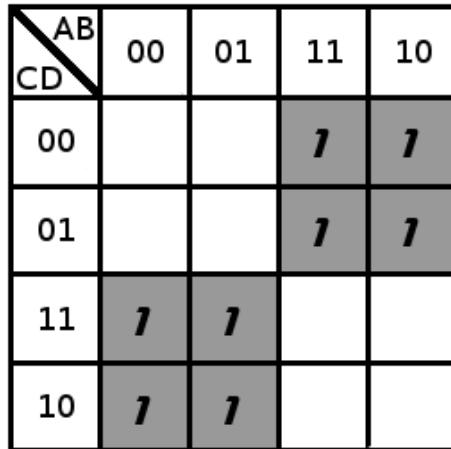


FIGURE 94: KARNAUGH MAP SHOWING CHECKERBOARD PATTERN

When this Karnaugh Map was simplified earlier, the constant True for the group in the upper right corner is A and in the lower left corner is C; the constant False in the lower left corner is A and in the upper right corner is C. Whichever way the designer chooses gives this equation:

$$Q = A \oplus C$$

To avoid confusion when simplifying these Karnaugh Maps, it is probably best to always look for the constant True variable.

### **MiniLab 4.7**

Create a circuit to meet the requirements of the Karnaugh Map in Figure 94 (above). Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 4.7", and today's date. Then save the file with this name: "minilab\_4\_7".

### **MAPS WITH ONE IN THE FIRST CELL**

Karnaugh Maps with a one in the first Cell (that is, in a four-variable map,  $A'B'C'D'$  is True) are simplified in a slightly different manner than those with a zero in that cell. When a one is present in the first cell, two of the input variables must be combined with an XNOR gate (though it does not matter which two are combined). To simplify these circuits, use the same technique presented above; but then select any two of the variable and change the gate from XOR to XNOR. Following are some examples.

AB \ CD	00	01	11	10
00	1		1	
01		1		1
11	1		1	
10		1		1

FIGURE 95: KARNAUGH MAP SHOWING CHECKERBOARD PATTERN

In the above Karnaugh Map, it is not possible to group the ones. Since A, B, C, and D are all True in at least one cell, they would all appear in the final equation; however, since there is a one in the first cell, then two of the variables must be combined with an XNOR gate. Here is one possible solution:

$$Q = (A \oplus B)' \oplus C \oplus D$$

AB \ CD	00	01	11	10
00	1	1	1	1
01				
11	1	1	1	1
10				

FIGURE 96: KARNAUGH MAP SHOWING CHECKERBOARD PATTERN

Remember that it does not matter if constant zeros or ones are used to simplify any given group, and when there is a 1 in the top left square it is usually easiest to look for constant zeros for that group (since that square is for input 0000). Row one of this map has a constant zero for C and D, and row three has a constant one for C and D. Since there is a one in the first cell, then the two variables must be combined with an XNOR gate:

$$Q = (C \oplus D)'$$

		AB 00	01	11	10
		CD 00	1	1	
		01	1	1	
		11		1	1
		10			1

FIGURE 97: KARNAUGH MAP SHOWING CHECKERBOARD PATTERN

The upper left corner of this map has a constant zero for A and C, and the lower right corner has a constant one for A and C. Since there is a one in the first cell, then the two variables must be combined with an XNOR gate:

$$Q = (A \oplus C)'$$

## 4.8: QUINE-MCCLUSKEY SIMPLIFICATION METHOD

### INTRODUCTION

When a Boolean equation involves five or more variables it becomes very difficult to solve using standard algebra techniques or Karnaugh maps; however, the Quine–McCluskey algorithm can be used to solve these types of Boolean equations. This method was developed by W.V. Quine and Edward J. McCluskey and is sometimes referred to as the *method of prime implicants* or the *tabulation method*.

The Quine-McCluskey method is based upon a simple Boolean algebra principle: if two different expressions differ by only a single variable and its complement, then those two expressions can be combined:

$ABC + ABC'$	Original Term
$AB + AB'$	Complement
$AB$	Idempotence

TABLE 66: SIMPLIFYING A BOOLEAN EXPRESSION

Thus:

$$ABC + ABC' = AB$$

### EXAMPLE 1

#### STEP 1: CREATE THE IMPLICANTS

Given the following Sigma representation of a Boolean equation:

$$\sum(A,B,C,D) = \sum(0,1,2,5,6,7,9,10,11,14)$$

The following table shows only the input variables for the True minterm values.

Minterm	A	B	C	D
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
14	1	1	1	0

TABLE 67: MINTERM TABLE

To simplify this equation, the minterms that evaluate to True (as listed above) are first placed in a minterm table so that they form sections that are easy to combine. Each section contains only the

minterms that have the same number of 1's. Thus, the first section contains all minterms with zero 1's, the second section contains the minterms with one 1, and so forth.

Number of 1's	Minterm	Binary
0	0	0000
1	1	0001
	2	0010
2	5	0101
	6	0110
	9	1001
	10	1010
	7	0111
3	11	1011
	14	1110

TABLE 68: REARRANGED MINTERM TABLE

Start combining minterms with other minterms to create *Size 2 Implicants* (called that since each implicant combines two minterms), but only those terms that vary by a single binary digit can be combined. When two minterms are combined, the binary digit that is different between the minterms is replaced by a dash, indicating that the digit doesn't matter. For example, 0000 and 0001 can be combined to form 000-. The table is modified to add a *Size 2 Implicant* column that indicates all of the combined terms. Note that every minterm must be compared to every other minterm so all possible implicants are formed. This is easier than it sounds, though, since terms in section 1 must be compared only with section 2, then those in section 2 are compared with section 3, and so forth, since each section differs from the next by a single binary digit. The *Size 2 Implicant* column contains the combined binary form along with the numbers of the minterms used to create that implicant. It is also important to mark all minterms that are used to create the *Size 2 Implicants* since allowance must be made for any not combined. Therefore, in the following table, as a minterm is used it is also marked with an asterisk.

Number of 1's	Minterm	Binary	Size 2 Implicants
0	0*	0000	000-(0,1)
1	1*	0001	00-0 (0,2)
	2*	0010	0-01 (1,5)
2	5*	0101	-001 (1,9)
	6*	0110	0-10 (2,6)
	9*	1001	-010 (2,10)
	10*	1010	01-1 (5,7)
			011- (6,7)
3	7*	0111	-110 (6,14)
	11*	1011	10-1 (9,11)
	14*	1110	101- (10,11)

TABLE 69: SIZE 2 IMPLICANTS

All of the *Size 2 Implicants* can now be combined to form *Size 4 Implicants* (those that combine a total of four minterms). Again, it is essential to only combine those with only a single binary digit difference. For this step, the dash can be considered the same as a single binary digit, as long as it is in the same place for both implicants. Thus, -010 and -110 can be combined to --10, but -010 and 0-00 cannot be combined since the dash is in different places in those numbers. It helps to match up the dashes first and then look at the binary digits. Again, as the various size-2 implicants are used they are marked; but notice that a single size-4 implicant actually combines four size-2 implicants.

Number of 1's	Minterm	Binary	Size 2 Implicants	Size 4 Implicants
0	0*	0000	000-(0,1)	--10 (2,10,6,14)
1	1*	0001	00-0 (0,2)	
	2*	0010	0-01 (1,5)	
2	5*	0101	-001 (1,9)	
	6*	0110	0-10 (2,6)*	
	9*	1001	-010 (2,10)*	
	10*	1010	01-1 (5,7)	
3	7*	0111	011- (6,7)	
	11*	1011	-110 (6,14)*	
	14*	1110	10-1 (9,11)	
			101- (10,11)	
			1-10 (10,14)*	

TABLE 70: SIZE 4 IMPlicants

None of the terms can be combined any further. All of the minterms or implicants that are not marked are *Prime Implicants*. In the table above, for example, 000- is a *Prime Implicant*. The *Prime Implicants* will be placed in a chart and further processed in the next step.

#### STEP 2: THE PRIME IMPLICANT TABLE

A *Prime Implicant Table* can now be constructed. The prime implicants are listed down the left side of the table, the decimal equivalent of the minterms goes across the top, and the Boolean representation of the prime implicants is listed down the right side of the table.

	0	1	2	5	6	7	9	10	11	14	
000- (0,1)	x	x									A'B'C'
00-0 (0,2)	x		x								A'B'D'
0-01 (1,5)		x	x								A'C'D
-001 (1,9)	x				x						B'C'D
01-1 (5,7)			x	x							A'BD
011- (6,7)				x	x						A'BC
10-1 (9,11)					x		x	x			AB'D
101- (10,11)						x	x	x	x		AB'C
--10 (2,10,6,14)	x	x		x	x	x	x	x	x		CD'

TABLE 71: PRIME IMPLICANT TABLE

An x marks the intersection where each minterm (on the top row) is used to form one of the prime implicants (on the left column). Thus, minterm 0 (or "0000") is used to form the prime implicant 000- (0,1) in row 1 and 00-0 (0,2) in row 2.

The *Essential Prime Implicants* can be found by looking for columns that contain only one "x". The column for minterm 14 has only one x, in the last row, 101- (10,11); thus, it is an *Essential Prime Implicant*. That term in the right column for the last row, CD', must appear in the simplified equation. However, that term also covers the columns for 2, 6, and 10; so they can be removed from the table. The Prime Implicant table, then, can be simplified.

	0	1	5	7	9	11	
000- (0,1)	x	x					A'B'C'
00-0 (0,2)	x						A'B'D'
0-01 (1,5)		x	x				A'C'D
-001 (1,9)		x		x			B'C'D
01-1 (5,7)			x	x			A'BD
011- (6,7)				x			A'BC
10-1 (9,11)					x	x	AB'D
101- (10,11)						x	AB'C

TABLE 72: PRIME IMPLICANT TABLE

Finally, the various rows can be combined in any order the designer desires. For example, if row 10-1 (9,11), is selected as a required implicant in the solution, then minterms 9 and 11 are accounted for in the final equation, which means that all x's in those columns can be removed. When that is done, then, rows 101- (10,11) and 10-1 (9,11) no longer have any marks in the table, and they can be removed.

	0	1	5	7	
000- (0,1)	x	x			A'B'C'
00-0 (0,2)	x				A'B'D'
0-01 (1,5)		x	x		A'C'D
-001 (1,9)		x			B'C'D
01-1 (5,7)			x	x	A'BD
011- (6,7)				x	A'BC

TABLE 73: PRIME IMPLICANT TABLE

The designer next decided to select 01-1 (5,7), A'BD, as a required implicant. That will include minterms 5 and 7, and those columns may be removed along with rows 01-1 (5,7), A'BD, and 011- (6,7), A'BC.

	0	1	
000- (0,1)	x	x	A'B'C'
00-0 (0,2)	x		A'B'D'
0-01 (1,5)		x	A'C'D
-001 (1,9)		x	B'C'D

TABLE 74: PRIME IMPLICANT TABLE

The last two minterms (0 and 1) can be covered by the implicant 000- (0,1), and that also eliminates the last three rows in the chart.

The original Boolean expression, then, has been simplified from:

$$Q = A'B'C'D' + A'B'C'D + A'B'CD' + A'BC'D + A'BCD' + A'BCD + AB'C'D + AB'CD' + AB'CD + ABCD'$$

To:

$$Q = A'B'C' + A'BD + AB'D + CD'$$

## EXAMPLE 2

### STEP 1: CREATE THE IMPLICANTS

Given the following Sigma representation of a Boolean equation:

$$\sum(A,B,C,D,E,F) = \sum(0,1,8,9,12,13,14,15,32,33,37,39,48,56)$$

The following table shows only the input variables for the True minterm values.

Minterm	A	B	C	D	E	F
0	0	0	0	0	0	0
1	0	0	0	0	0	1
8	0	0	1	0	0	0
9	0	0	1	0	0	1
12	0	0	1	1	0	0
13	0	0	1	1	0	1
14	0	0	1	1	1	0
15	0	0	1	1	1	1
32	1	0	0	0	0	0
33	1	0	0	0	0	1
37	1	0	0	1	0	1
39	1	0	0	1	1	1
48	1	1	0	0	0	0
56	1	1	1	0	0	0

TABLE 75: MINTERM TABLE

To simplify this equation, the minterms that evaluate to True (as listed above) are first placed in a minterm table so that they form sections that are easy to combine. Each section contains only the minterms that have the same number of 1's. Thus, the first section contains all minterms with zero 1's, the second section contains the minterms with one 1, then two 1's, and so forth.

Number of 1's	Minterm	Binary
0	0	000000
1	1	000001
	8	001000
	32	100000
2	9	001001
	12	001100
	33	100001
	48	110000
3	13	001101
	14	001110
	37	100101
	56	111000
4	15	001111
	39	100111

TABLE 76: REARRANGED MINTERM TABLE

Start combining minterms with other minterms to create *Size 2 Implicants*.

Number of 1's	Minterm	Binary	Size 2 Implicants
0	0*	000000	00000-(0,1)
1	1*	000001	-00000 (0,32)
	8*	001000	00-000 (0,8)
	32*	100000	-00001 (1,33)
	9*	001001	00-001 (1,9)
2	12*	001100	10000- (32,33)
	33*	100001	1-0000 (32,48)
	48*	110000	00100- (8,9)
	13*	001101	001-00 (8,12)
3	14*	001110	100-01 (33,37)
	37*	100101	001-01 (9,13)
	56*	111000	00110- (12,13)
	15*	001111	0011-0 (12,14)
4	39*	100111	11-000 (48,56)
			1001-1 (37,39)
			0011-1 (13,15)
			00111- (14,15)

TABLE 77: SIZE 2 IMPLICANTS

All of the *Size 2 Implicants* can now be combined to form *Size 4 Implicants*.

Number of 1's	Minterm	Binary	Size 2 Implicants	Size 4 Implicants
0	0*	000000	00000-(0,1)*	-0000- (0,1,32,33)
1	1*	000001	-00000 (0,32)*	00-00- (0,1,8,9)
	8*	001000	00-000 (0,8)*	
	32*	100000		001-0- (8,9,12,13)
2	9*	001001	-00001 (1,33)*	
	12*	001100	00-001 (1,9)*	0011-- (12,13,14,15)
	33*	100001	10000- (32,33)*	
	48*	110000	1-0000 (32,48)	
3	13*	001101	00100- (8,9)*	
	14*	001110	001-00 (8,12)*	
	37*	100101	100-01 (33,37)	
	56*	111000	001-01 (9,13)*	
4	15*	001111	00110- (12,13)*	
			0011-0 (12,14)*	
	39*	100111	11-000 (48,56)	
			1001-1 (37,39)	
			0011-1 (13,15)*	
			00111- (14,15)*	

TABLE 78: SIZE 4 IMPLICANTS

None of the terms can be combined any further. All of the minterms or implicants that are not marked with an asterisk are *Prime Implicants*. In the table above, for example, 1-0000 is a *Prime Implicant*. The *Prime Implicants* will be placed in a chart and further processed in the next step.

#### STEP 2: THE PRIME IMPlicant TABLE

A *Prime Implicant Table* can now be constructed. The prime implicants are listed down the left side of the table, the decimal equivalent of the minterms goes across the top, and the Boolean representation of the prime implicants is listed down the right side of the table.

	0	1	8	9	12	13	14	15	32	33	37	39	48	56	
11-000 (48,56)												x	x		ABD'E'F'
00-00- (0,1,8,9)	x	x	x	x											A'B'D'E'
1001-1 (37,39)										x	x				AB'C'DF
1-0000 (32,48)							x					x			AC'D'E'F'
0011-- (12,13,14,15)				x	x	x	x								A'B'CD
-0000- (0,1,32,33)	x	x						x	x						B'C'D'E'
001-0- (8,9,12,13)		x	x	x	x										A'B'CE'
100-01 (33,37)								x	x						AB'C'E'F

TABLE 79: PRIME IMPlicant TABLE

In the above chart, there are four columns that contain only one mark: 14, 15, 39, and 56. The rows that intersect the columns at that mark are *Essential Prime Implicants*, and their Boolean Expressions must appear in the final equation. Therefore, the final equation will contain, at a minimum: A'B'CD (row 5,

covers minterms 14 and 15),  $AB'C'DF$  (row 3, covers minterm 39), and  $ABD'E'F'$  (row 1, covers minterm 56). Since those expressions are in the final equation, the rows that contain those expressions can be removed from the chart in order to make further analysis less confusing.

Also, because the rows with *Essential Prime Implicants* are contained in the final equation, other minterms marked by those rows are covered and need no further consideration. For example, minterm 48 is covered by row 1 (used for minterm 56), so column 48 can be removed from the table. In a similar fashion, columns 12, 13, and 37 are covered by other minterms, so they can be removed from the table.

	0	1	8	9	32	33	
00-00- (0,1,8,9)	x	x	x	x			$A'B'D'E'$
1-0000 (32,48)					x		$AC'D'E'F'$
-0000- (0,1,32,33)	x	x			x	x	$B'C'D'E'$
001-0- (8,9,12,13)		x	x				$A'B'CE'$
100-01 (33,37)					x		$AB'C'E'F$

TABLE 80: PRIME IMPLICANT TABLE

The circuit designer can select the next term to include in the final equation from any of the five rows still remaining in the chart; however, the first term (00-00-, or  $A'B'D'E'$ ) would eliminate four columns, so that will be a logical next choice. When that term is selected for the final equation, then row one, 00-00-, can be removed from the chart; and columns 0, 1, 8, and 9 can be removed since those minterms are covered.

The minterms marked for row 001-0- (8,9,12,13) are also covered, so this row can be removed.

	32	33	
1-0000 (32,48)	x		$AC'D'E'F'$
-0000- (0,1,32,33)	x	x	$B'C'D'E'$
100-01 (33,37)		x	$AB'C'E'F$

TABLE 81: PRIME IMPLICANT TABLE

For the next simplification, row -0000- is selected since that would also cover the minterms that are marked for all remaining rows. Therefore, the expression  $B'C'D'E'$  will become part of the final equation.

When the analysis is completed, the original requirement, which contained 14 minterms, is simplified into this equation which contains only 5 terms:

$$Q = ABD'E'F' + A'B'D'E' + AB'C'DF + A'B'CD + B'C'D'E'$$

#### SUMMARY

While the Quine–McCluskey method is useful for large Boolean expressions containing multiple inputs, it is also tedious and prone to error when done by hand. Also, there are some Boolean expressions (called Cyclic and Semi-Cyclic Primes) that do not reduce using this method. Finally, both Karnaugh maps and Quine-McCluskey methods become very complex when more than one output is required of a

circuit. Fortunately, many software solutions are available to simplify Boolean Expressions using advanced mathematical techniques.

#### PRACTICE PROBLEMS

The following problems are presented as practice for using the Quine-McClusky method to simplify a Boolean expression. Note: designers can select different Prime Implicants so the simplified expression could vary from what is presented below.

<b>1</b>	Expression	$\sum(A,B,C,D) = \sum(0,1,2,5,6,7,9,10,11,14)$
	Simplified	$A'B'C' + A'BD + AB'D + CD'$
<b>2</b>	Expression	$\sum(A,B,C,D) = \sum(0,1,2,3,6,7,8,9,14,15)$
	Simplified	$A'C + BC + B'C'$
<b>3</b>	Expression	$\sum(A,B,C,D) = \sum(1,5,7,8,9,10,11,13,15)$
	Simplified	$C'D + AB' + BD$
<b>4</b>	Expression	$\sum(A,B,C,D,E) = \sum(0,4,8,9,10,11,12,13,14,15,16,20,24,28)$
	Simplified	$A'B + D'E'$

## 4.9: AUTOMATED TOOLS

### INTRODUCTION

There are numerous automated tools available to aid in simplifying complex Boolean equations. Many of the tools are quite expensive and intended for professionals working full time in systems design; but others are inexpensive, or even free of charge, and are more than adequate for student use. This topic introduces one such free tool: Karma.

### KARMA

#### INTRODUCTION

Karma (for KARnaugh MAp simplifier) is a free Java-based tool designed to help simplify Boolean expressions. Both an online and downloaded version of Karma is available. The application can be found at: <http://www.inf.ufrgs.br/logics/docman/karma/>

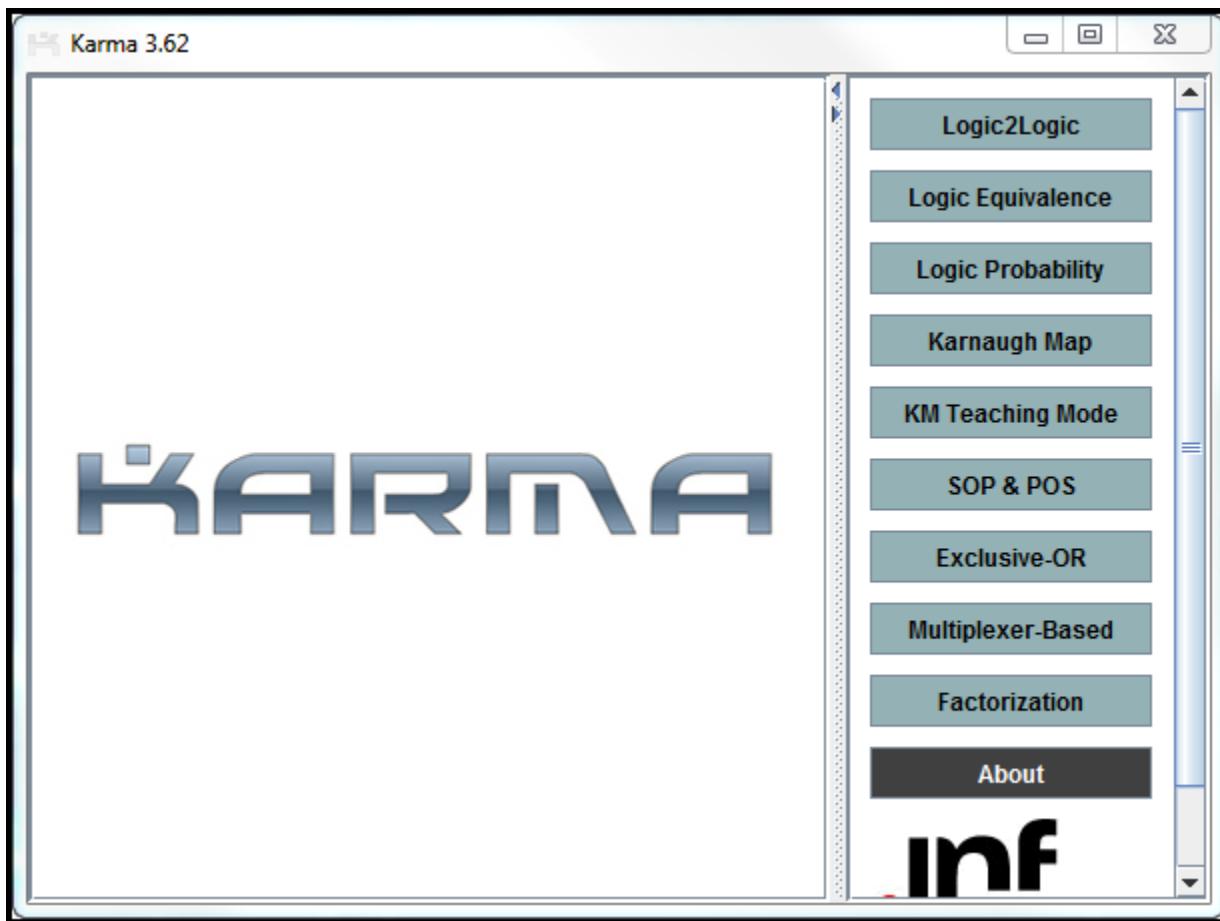


FIGURE 98: KARMA INITIAL SCREEN

The right side of the screen contains a row of tools available in Karma and the main part of the screen is a canvas where most of the work is done. The following tools are available:

- Logic2Logic. Converts between two different logical representations of data; for example, a Truth Table can be converted to Boolean expressions.
- Logic Equivalence. Compares two functions and determines if they are equivalent; for example, a truth table can be compared with a SOP expression to see if they are the same.
- Logic Probability. Calculates the probability of any one outcome for a given Boolean expression.
- Karnaugh Map. Analyzes a Karnaugh map and returns the Minimized Expression.
- KM Teaching Mode. Provides drill and practice with Karnaugh maps; for example, finding adjacent minterms on a 6-variable map.
- SOP and POS. Finds the SOP and POS expressions for a given function.
- Exclusive-OR. Uses XOR gates to simplify an expression.
- Multiplexer-Based. Realizes a function using multiplexers.
- Factorization. Factors Boolean expressions.
- About. Information about Karma.

For this lesson, only the Karnaugh Map analyzer will be used, and the initial screen for that function is below.

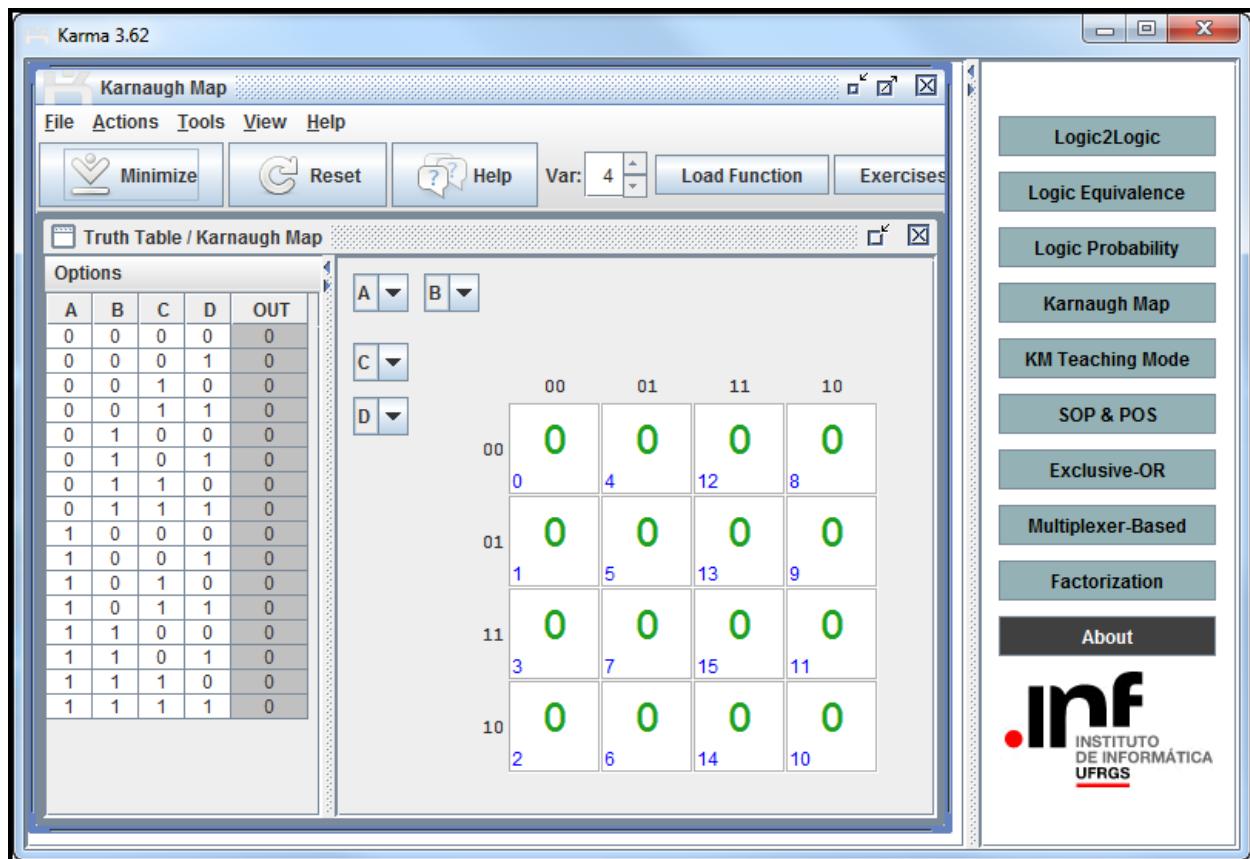


FIGURE 99: KARNAUGH MAP IN KARMA

**DATA ENTRY**

When using Karma, the first step is to input some sort of information about the circuit to be analyzed. That information can be entered in several different formats, but the most common for this class would be either a truth table or a Boolean expression.

To enter the initial data, click the *Load Function* button at the top of the canvas.

By default, the Load Function screen opens with a blank screen. In the lower left corner of the Load Function window, the Source Format for the input data can be selected. There is a template available for each of the different source formats; and that template can be used to help with data entry. The best way to work with Karma is to click the “Templates” button and select the data format being used. In the following illustration, the “Expression 1” template was selected:

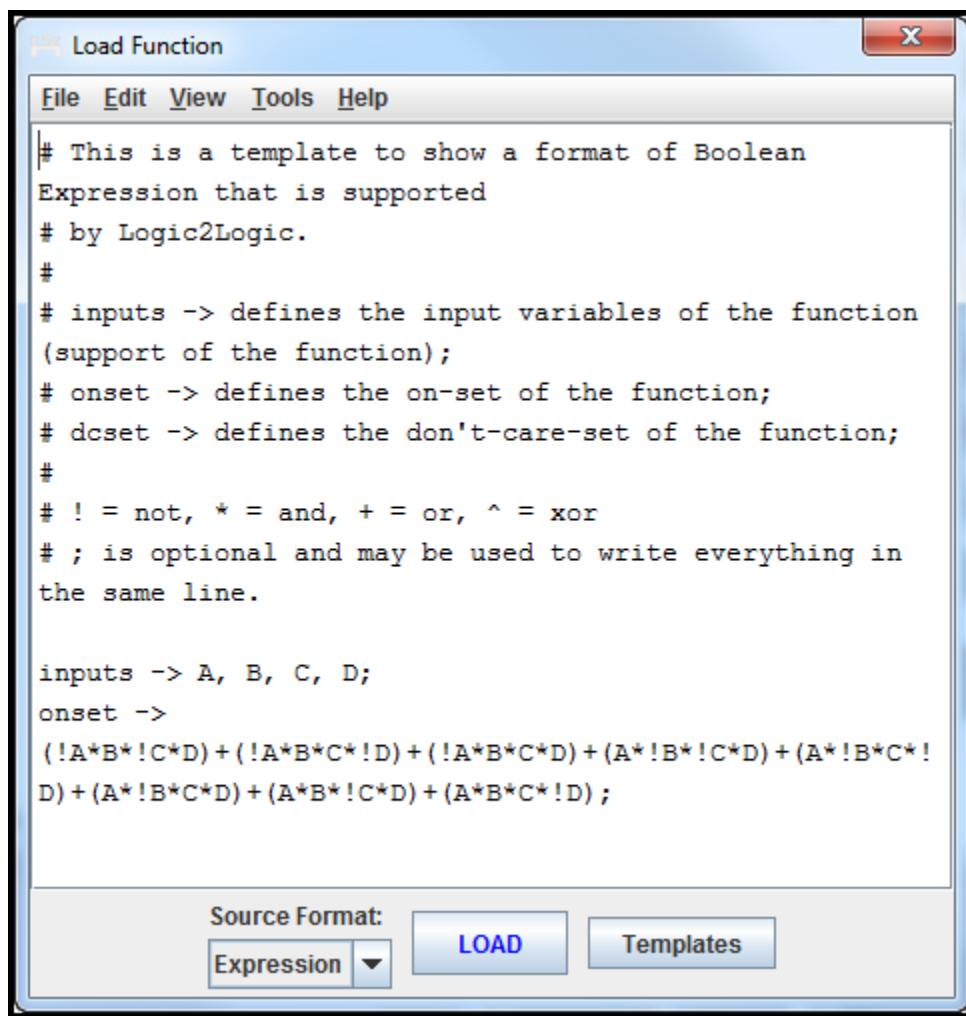


FIGURE 100: EXPRESSION ONE LOADED INTO KARMA

The designer would replace the “inputs” and “onset” lines with information for the circuit being simplified. Once the source data are entered into this window, click the “Load” button at the bottom of the window to load the data into Karma.

**DATA SOURCE FORMATS**

Karma works with input data in any of six different formats: Boolean Expression, Truth Table, Integer, Minterms, Berkeley Logic Interchange Format ("BLIF"), and Binary Decision Diagram (BDD). BLIF and BDD are programming tools that are beyond the scope of this lesson and will not be covered.

**EXPRESSION**

Boolean expressions can be defined in Karma using this type of format:

```
#Sample Expression
(!x1*x2*x4) + (!x1*x2*x3) + (x1*x4*x5) + (x1*x3*x4)
```

Notes:

- Any line that starts with a hash mark ("#") is a comment and will be ignored by Karma.
- "Not" is indicated by a leading exclamation mark. Thus "!x1" is the same as X1'.
- All operations are explicit. In real-number algebra the phrase "AB" is understood to be "A\*B". However, in Karma, since variable names can be more than one character long, all operations must be explicitly stated. AND is indicated by an asterisk and OR is indicated by a plus sign.
- No space is left between operations.

**TRUTH TABLE**

A truth table can be defined in Karma using the following format:

```
#Sample Truth Table
inputs -> X, Y, Z
000 : 1
001 : 1
010 : 0
011 : 0
100 : 0
101 : 1
110 : 0
111 : 1
```

Notes:

- Any line that starts with a hash mark ("#") is a comment and will be ignored by Karma.
- The various inputs are named before they are used. In the example, there are three inputs: X, Y, and Z.
- Each row in the truth table is shown, along with the output required. So, in the example above, an input of 000 should yield an output of 1.
- An output of "-" is permitted and means "don't care".

**INTEGER**

In Karma, an integer can be used to define the outputs of the truth table, so it is "shorthand" for an entire truth table input. Here is the example of the "integer" type input:

```
#Sample Integer Input
inputs -> A, B, C, D
onset -> E81A base 16
```

Notes:

- Any line that starts with a hash mark ("#") is a comment and will be ignored by Karma.
- Input variables are defined first. In this example, there are four inputs: A, B, C, and D.
- The "onset" line indicates what combinations of inputs should yield a True on a truth table. In the example, the number E81A is a hexadecimal number that is written like this in binary:

```
1110 1000 0001 1010
E     8      1      A
```

The least significant bit of the binary number, 0 in this example, corresponds to the output of the first row in the truth table; thus, it is false. Each bit to the left of the least significant bit corresponds to the next row, counting from 0000 to 1111. Here is the truth table generated by the hexadecimal integer E81A:

Inputs				Output
A	B	C	D	Q
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

TABLE 82: TRUTH TABLE

The "Output" column contains the binary integer 1110 1000 0001 1010 (or E81Ah) from bottom to top.

TERMS

Data input can be defined by using the minterms for the Boolean expression. Here is an example minterm input:

```
#Sample Minterms
inputs -> A, B, C, D
onset -> 0, 1, 2, 3, 5, 10
```

Notes:

- Any line that starts with a hash mark ("#") is a comment and will be ignored by Karma.
- The inputs, W, X, Y, and Z, are defined first.
- The "onset" line indicates the minterms that yield a "true" output.
- This is similar to a SOP Sigma expression, and the digits in that expression could be directly entered on the onset line. For example, the onset line above would have been generated from this Sigma expression:

$$\sum(A,B,C,D) = \sum(0,1,2,3,5,10)$$

#### TRUTH TABLE AND KARNAUGH MAP INPUT

While Karma will accept a number of different input methods, as described above, one of the easiest to use is the Truth Table, and the related Karnaugh Map, and these are displayed by default the Karnaugh Map function is selected. The value of any of the cells in the Out column in the Truth Table, or cells in the Karnaugh Map, can be cycled through 0, 1, and "don't care" (indicated by a dash) on each click of the mouse in the cell. The Truth Table and Karnaugh Map are synchronized as cells are clicked. The number of input variables can be adjusted by changing the *Var* setting at the top of the screen. Also, the placement of those variables on the Karnaugh Map can be adjusted as desired.

## SOLUTION

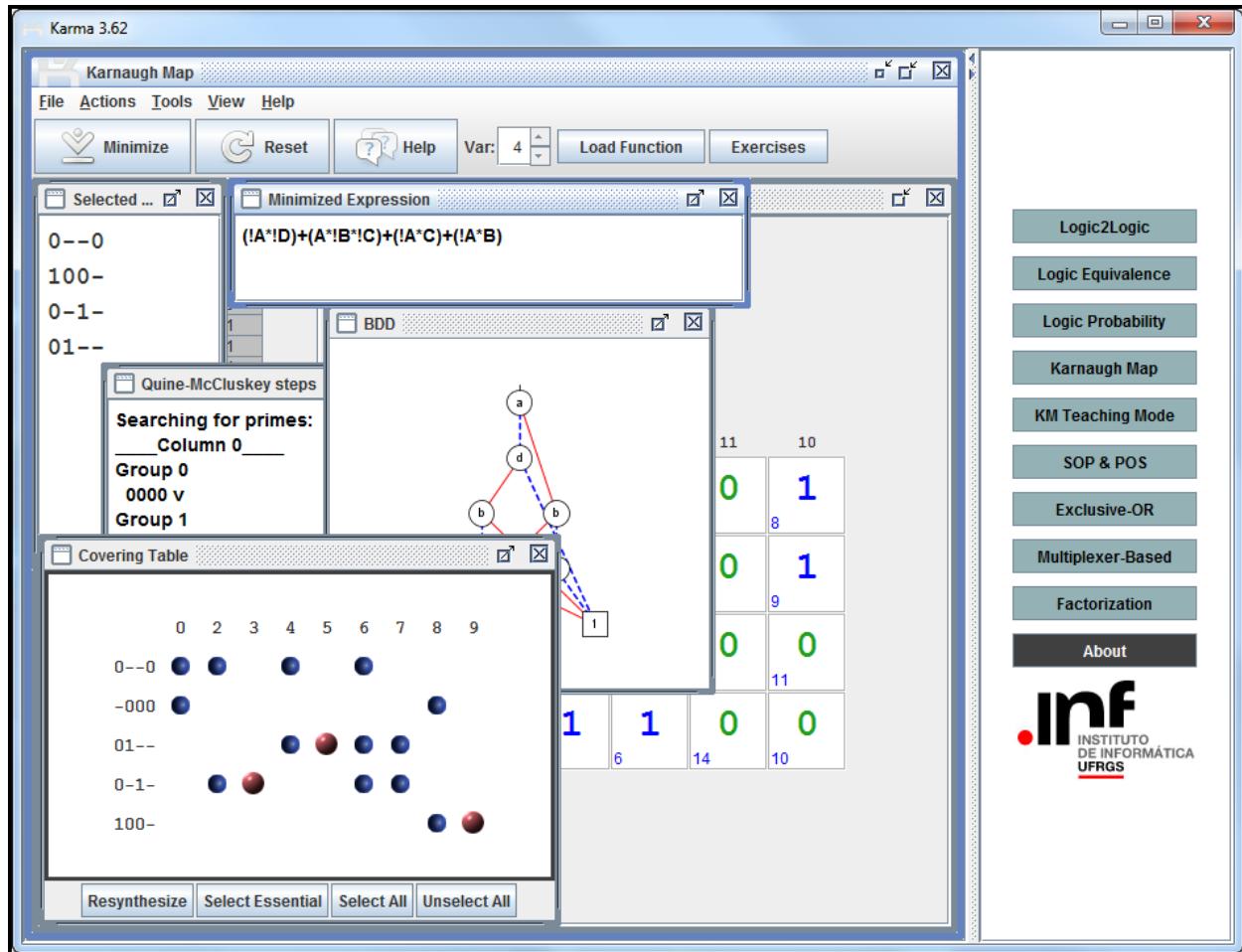


FIGURE 101: KARMA SOLUTION

To simplify the Karnaugh Map, click the Minimize button. A number of windows will pop up (illustrated in the image above), each showing the circuit simplification in a slightly different way. Note: this Boolean expression was entered to generate the following illustrations:  $A'C + A'B + AB'C' + B'C'D'$ . Karma minimized the expression and returned:

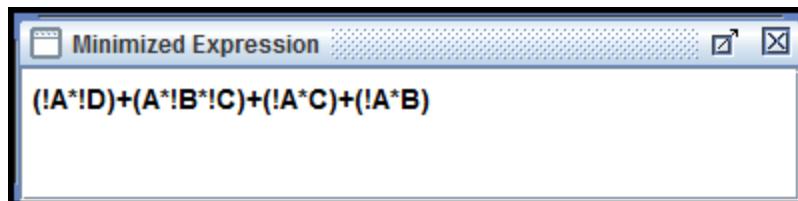


FIGURE 102: MINIMIZED EXPRESSION

In this solution, a NOT term is identified by a leading exclamation point; thus, the minimized expression is:  $A'D' + AB'C' + A'C + A'B$ .



Karma groups the variables for clarity, but the grouping symbols (parenthesis) are not needed since the groups are obvious when the expression is written in a normal Boolean form.

### BDDEIRO

The BDDeiro window is a form of *Binary Decision Diagram (BDD)*. The decision tree for the minimized expression is represented graphically in this window.

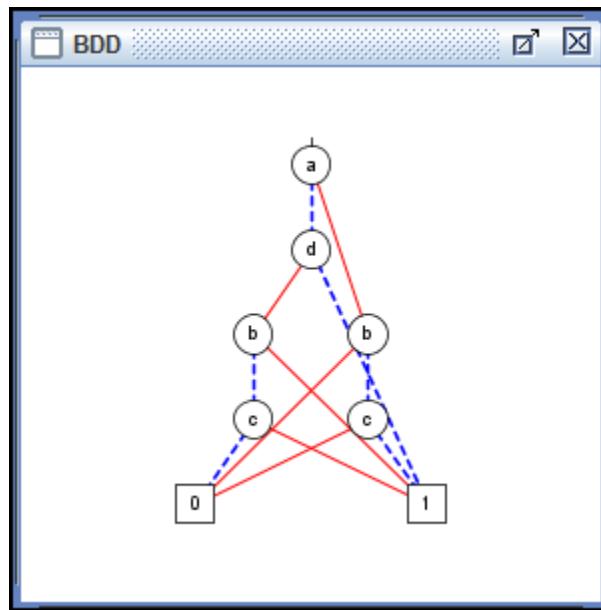


FIGURE 103: BDDEIRO MAP IN KARMA

The top node represents input  $a$ , which can either be true or false. If false, then follow the blue dotted line down to node  $d$ , which can also be either true or false. If  $d$  is false, follow the blue dotted line down to the  $1$  output. This would mean that one True output is  $A'D'$ , which can be verified as one of the outputs in the minimized expression.

Starting at the top again, if  $a$  is true, then follow the solid red line down to  $b$ . If that is true, then follow the red solid line down to the  $0$  output. The expression  $AB$  is false and does not appear in the minimized solution. In a similar way, all four True outputs, and three false outputs, can be traced from the top to bottom of the diagram.

### QUINE-MCCLUSKEY

Karma includes complete Quine-McCluskey solution data. Several tables display the various implicants and show how they are derived.

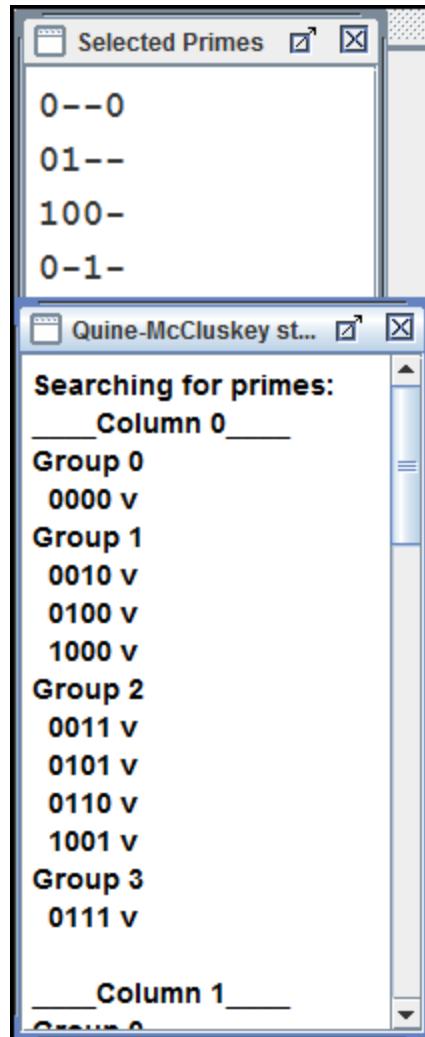


FIGURE 104: QUINE-MCCLUSKEY SOLUTION

Karma also displays the Covering Table for a Quine-McCluskey solution.

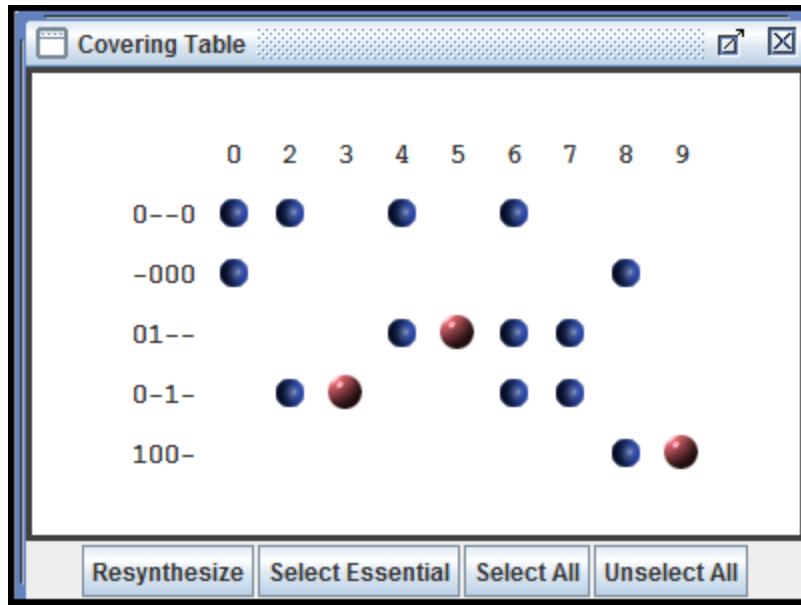


FIGURE 105: COVERING TABLE

Each of the minterms (down the left column) can be turned on or off by clicking on it. The smaller blue balls in the table indicate prime implicants and the larger red balls (if any) indicate essential prime implicants. Because this table is interactive, various different solutions can be attempted by clicking some of the colored markers to achieve the best possible simplification.

#### PRACTICE PROBLEMS

The following problems are presented as practice for using Karma to simplify a Boolean expression.

Note: designers can select different Prime Implicants so the simplified expression could vary from what is presented below.

<b>1</b>	Expression	$\sum(A,B,C,D) = \sum(5,6,7,9,10,11,13,14)$
	Simplified	$BC'D + A'BC + ACD' + AB'D$
<b>2</b>	Expression	$A'BC'D + A'BCD' + A'BCD + AB'C'D + AB'CD' + AB'CD + ABC'D + ABCD'$
	Simplified	$BC'D + A'BC + ACD' + AB'D$
<b>3</b>	Expression	4-Variable Karnaugh Map where cells 5,6,7,9,10 are True and 13,14 are Don't Care
	Simplified	$BC'D + AC'D + A'BC + ACD'$
<b>4</b>	Expression	$\sum(A,B,C,D,E) = \sum(0,3,4,12,13,14,15,24,25,28,29,30)$
	Simplified	$ABD' + A'B'C'DE + BCE' + A'BC + A'B'D'E'$

## 5: COMBINATIONAL LOGIC

### 5.1 INTRODUCTION

Electronic circuits that do not require any memory devices (like flip-flops or registers) use what is called "Combinational Logic." These systems can be quite complex, but all outputs are determined solely on the actions of a series of logic gates to a given set of inputs. Another characteristic of combinational circuits is the lack of feedback. An output state is determined almost instantly from the inputs with no feedback loops. Combinational circuits can be reduced to a Boolean Algebra expression, though one that may be quite complex. Combinational circuits include many useful applications, including adders, subtractors, multipliers, dividers, encoders, decoders, display drivers, and keyboard encoders.

This unit develops combinational logic circuits.

## 5.2: ADDERS AND SUBTRACTORS

### INTRODUCTION

In the Binary Mathematics unit, the concept of adding two binary numbers was developed; and the process of adding binary numbers can be easily implemented in hardware. If two one-bit numbers are added they will produce a sum and an optional carry; and the hardware that performs this is called a half adder. If two one-bit numbers along with a third input (a carry-in from another adder) are added, they will produce a sum and an optional carry; and the hardware that performs this is function called a full adder.

### HALF ADDER

Following is the truth table for a half adder:

Inputs		Outputs	
A	B	Sum	Carry Out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

TABLE 83: TRUTH TABLE FOR HALF ADDER

The Sum column in this truth table matches the Exclusive Or (XOR) function, so that is the easiest way to create a half-adder. Following is an XOR gate being used as a 1-bit half adder; but the Carry Out bit has not been added to the circuit yet.

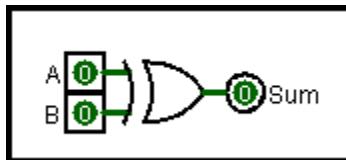


FIGURE 106: SIMPLE HALF ADDER USING XOR GATE

A half adder will generate a Carry Out bit if both inputs are high (or  $1+1=10_2$ ), so the half adder circuit should be modified to provide that carry bit. Notice in the following circuit, if A and B are both high, then the Sum bit will be low, but the Carry Out bit will be high.

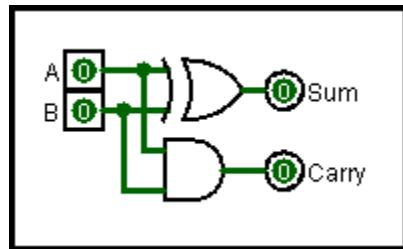


FIGURE 107: SIMPLE HALF ADDER WITH A CARRY OUT BIT

The half adder circuit is now complete and provides all of the outputs required by the truth table.

**FULL ADDER**

A full adder circuit sums two one-bit numbers along with a carry-in bit and produces a sum with a carry-out bit. The truth table for a full adder follows:

Inputs			Outputs	
A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

TABLE 84: TRUTH TABLE FOR FULL ADDER

The outputs from this truth table lead to these two Boolean equations:

$$\text{Sum} = X \oplus Y \oplus C_I$$

$$\text{CarryOut} = XY + X(C_I) + Y(C_I)$$

A full adder is realized like this:

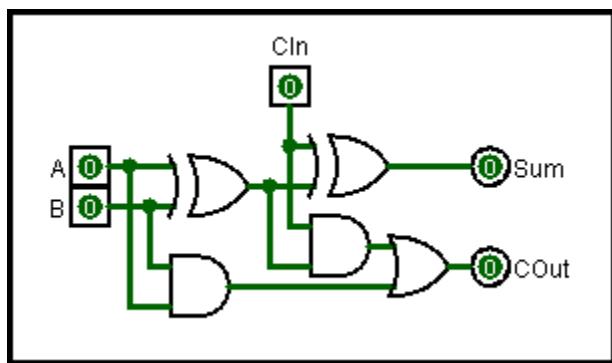


FIGURE 108: 1-BIT FULL ADDER INCLUDING CARRY IN AND CARRY OUT BITS

The XOR gate combining inputs A and B (on the left side of the circuit) is a simple half adder. The other XOR gate combines the Carry In bit with the Sum from the half adder. The two AND gates at the bottom right of the circuit control the Carry Out bit. The first of those gates will be active if there is a high from A XOR B and a carry in bit is present, the second if both A and B are high. This circuit meets the requirements of the truth table and forms a full adder.

**CASCADING ADDERS**

Full adders can be cascaded to create an adder of any desired size. For example, the following circuit is a 4-Bit Adder:

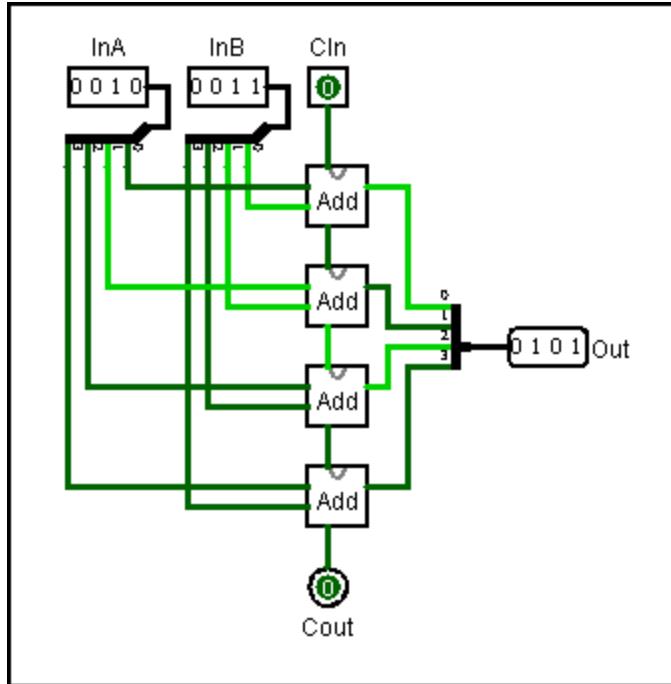


FIGURE 109: 4-BIT ADDER

The four "Add" circuits are each a one-bit full adder as developed above. To cascade them, the Carry Out bit from one stage is connected to the Carry In bit of the next. The bits from two four-bit numbers, "InA" and "InB," are connected to the inputs of each stage and the outputs of those stages are connected to an output pin. The circuit illustrates adding  $0010 + 0011 = 0101$ .

#### ADDER INTEGRATED CIRCUITS

The adder developed in this unit adds two 1-bit inputs (plus an optional Carry In) and produces a Sum along with a Carry Out. Then, four full adders were wired together to create a 4-bit adder. While this circuit is easy enough to create with discrete gates, integrated circuits implementing adders are easy to find and use. The following illustration shows a 7483 4-Bit Adder:

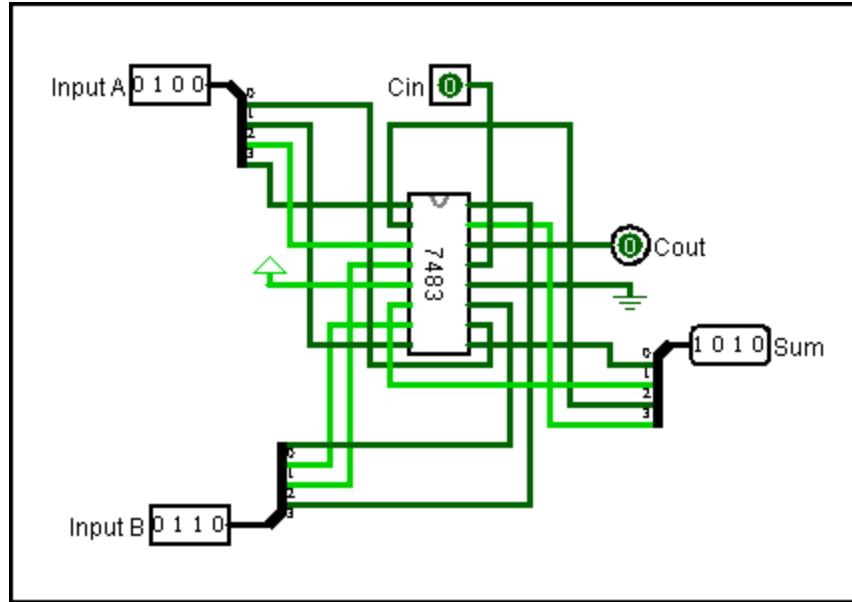


FIGURE 110: 7483 4-BIT ADDER IC

Two different 4-bit inputs are provided to the adder (*Input A* and *Input B*), along with a Carry In (*Cin*) bit. The adder outputs a 4-bit Sum along with a Carry Out (*Cout*) bit. The wiring for this IC seems convoluted, but the pins for the IC are not in the order that may be expected. Moving counterclockwise from the top left of the IC, the pins are: A4, Sum3, A3, B3, Vcc, Sum2, B2, A2, Sum1, A1, B1, Gnd, Cin, Cout, Sum4, B4. The illustration shows that 0100 (on Input A) plus 0110 (on Input B) is 1010 (on Sum). The Carry In and Carry Out bits are not used. The Vcc (power) and Ground pins are required for a physical IC; though the simulator would have permitted these two pins to remain disconnected. Two or more 7483 ICs could be cascaded to handle sums greater than 4-bits, as illustrated below:

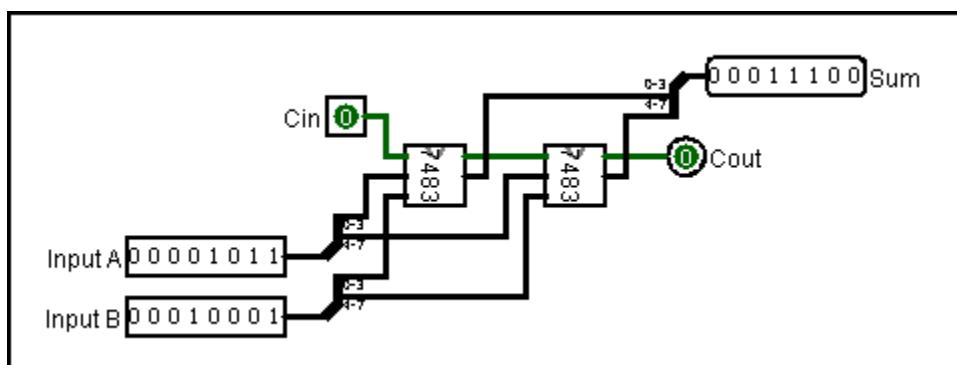


FIGURE 111: CASCADING 7483 4-BIT ADDERS TO CREATE 8-BIT ADDER

In the illustrated circuit, the low order bits (0-3) from Input A and Input B are sent to the first 7483 and the high order bits (4-7) are sent to the second 7483. The Carry Out bit from the first is connected to the Carry In bit of the second. The sum produced by the first 7483 is used as the low order bits in the 8-bit Sum, while the second 7483 is used for the high order bits in the Sum.

## SUBTRACTORS

The concept of subtraction in a digital circuit is simple:

$$A - B = A + (-B)$$

Thus, to subtract one binary number from another, change the subtrahend (the second number) into its two's complement and then add that to the first number. Finding the two's complement of a binary number is easy; invert all bits and then add one, which is more thoroughly covered in the Binary Subtraction unit. The Boolean equation for creating a subtractor would be:

$$A - B = A + (B') + 1$$

The following illustrates a one-bit subtractor:

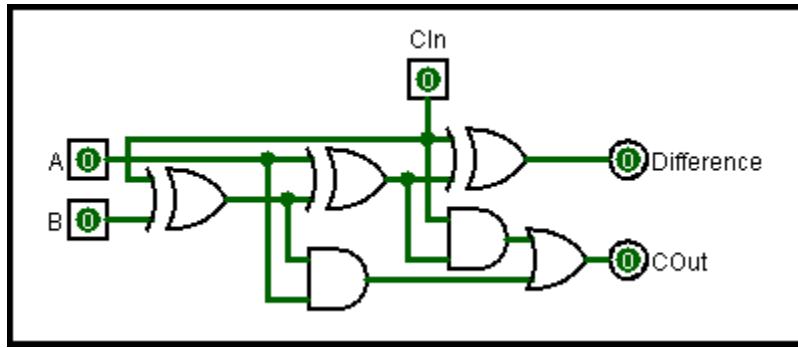


FIGURE 112: ONE-BIT SUBTRACTOR

The above circuit is the same as a one-bit adder, but with an additional XOR gate on the  $B$  input. To subtract with an adder, first add one to the input by making the  $Cin$  (or *Carry In*) high. That bit is also going to be used as a control to change the circuit from adder to subtractor by using an interesting trick. Rather than simply inverting input  $B$  with a NOT gate, an XOR is used. The  $Cin$  bit is fed to one of the XOR's inputs and  $B$  is fed to the other. When  $Cin$  is low, then the top XOR input is low and the XOR output would be the same as input  $B$ ; in this case, the XOR gate functions like a simple buffer. When  $Cin$  is high, then the XOR gate outputs the inverse of  $B$ ; in this case, the XOR gate is functioning like a NOT gate.

Therefore, with  $Cin$  low then the circuit is an adder, but with  $Cin$  high then the circuit is a subtractor. In the following illustration, four one-bit adders are cascaded to create a four-bit adder/subtractor.

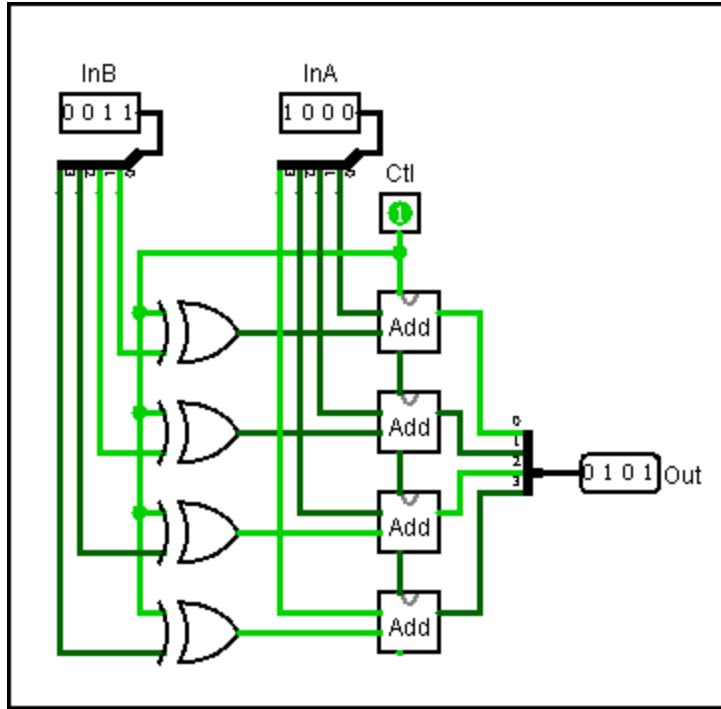


FIGURE 113: FOUR-BIT SUBTRACTOR

In this circuit, *Ctl* (which is actually *Cin* for the first adder) is linked to one of the XOR inputs on each bit of input *B*. That *Cin* input is functioning as a controller, changing this circuit from an adder to a subtractor. Note that the problem  $1000 - 0011 = 0101$  (or  $8 - 3 = 5$ ) is displayed in the circuit.

Just like adders, subtractors are available as Integrated Circuit (ICs). The 74X283 is a combination adder/subtractor; and the subtractor developed in this unit should make it easy to understand why an adder/subtractor would be so easy to create.

### 5.3: LAB – 8-BIT ADDER

#### PURPOSE

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

#### PROCEDURE

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

#### 1-BIT ADDER SUBCIRCUIT

Create a new subcircuit named *1-BitAdder* and add the components in the diagram on the left to the subcircuit:

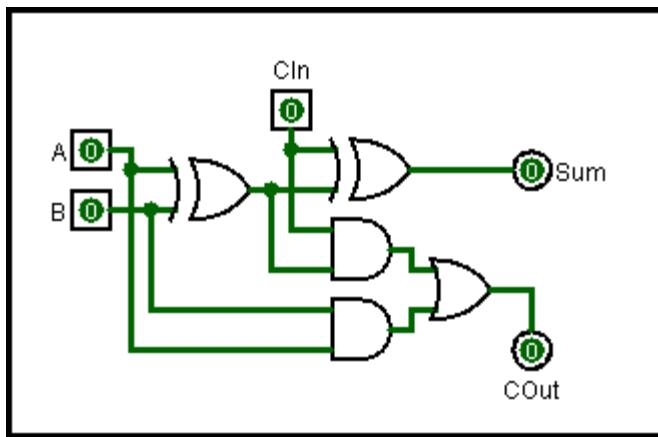


FIGURE 115: 1-BIT ADDER

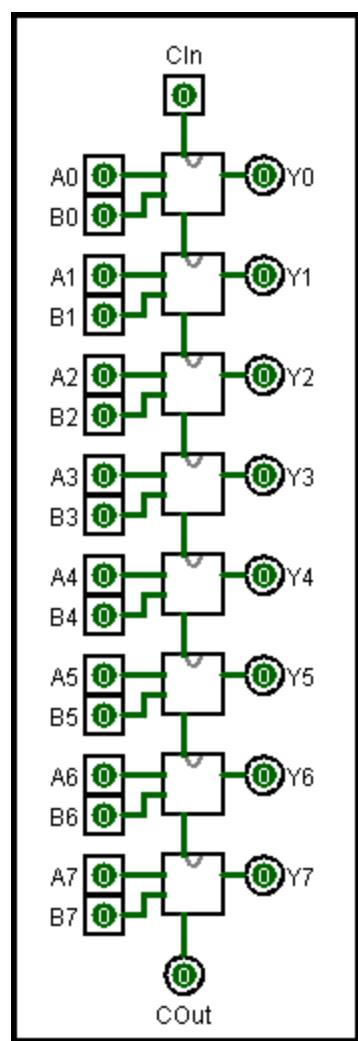


FIGURE 114: 8-BITADDER

#### 8-BIT ADDER SUBCIRCUIT

Create a new subcircuit named *8-BitAdder* and add the components found in the diagram on the right. Each of the small boxes in the center of the diagram is a *1-BitAdder* 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, A and 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  $101_2 + 011_2 = 1000_2$ .

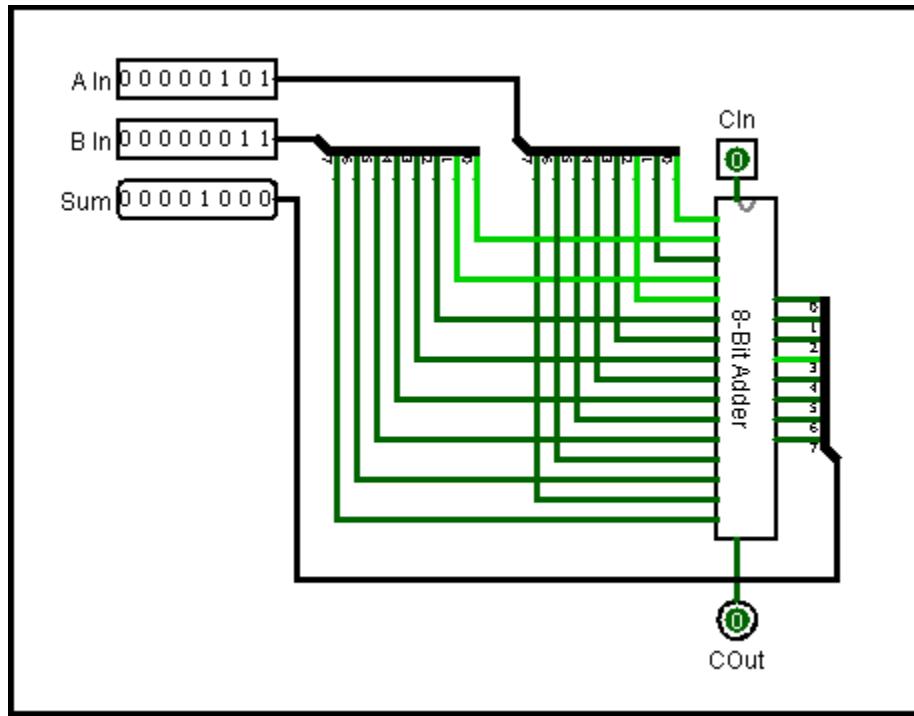


FIGURE 116: MAIN (ADDER) CIRCUIT

#### CLEANUP

Be sure the standard identifying information block is at the top left of the *Adder* circuit: Name, "Lab 5.3: 8-Bit Adder", and today's date. Save the file as *Lab 5\_3 – Adder*.

## 5.4: LAB – 8-BIT SUBTRACTOR

## PURPOSE

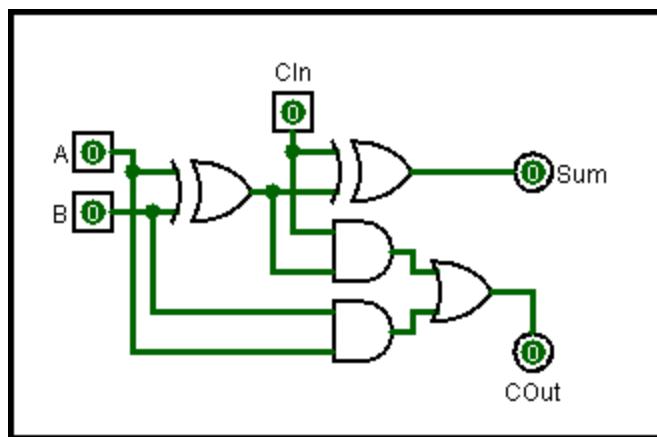
In this lab you will build a subtractor for two 8-bit numbers.

## PROCEDURE

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

## 1-BIT ADDER SUBCIRCUIT

Create a new subcircuit named *1-BitAdder* and add the components in the diagram below to the subcircuit:



**FIGURE 117: 1-BIT ADDER**

## 8-BIT SUBTRACTOR SUBCIRCUIT

Create a new subcircuit named *8-BitSubtractor* 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 a *1-BitAdder* 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 earlier in this book. 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.

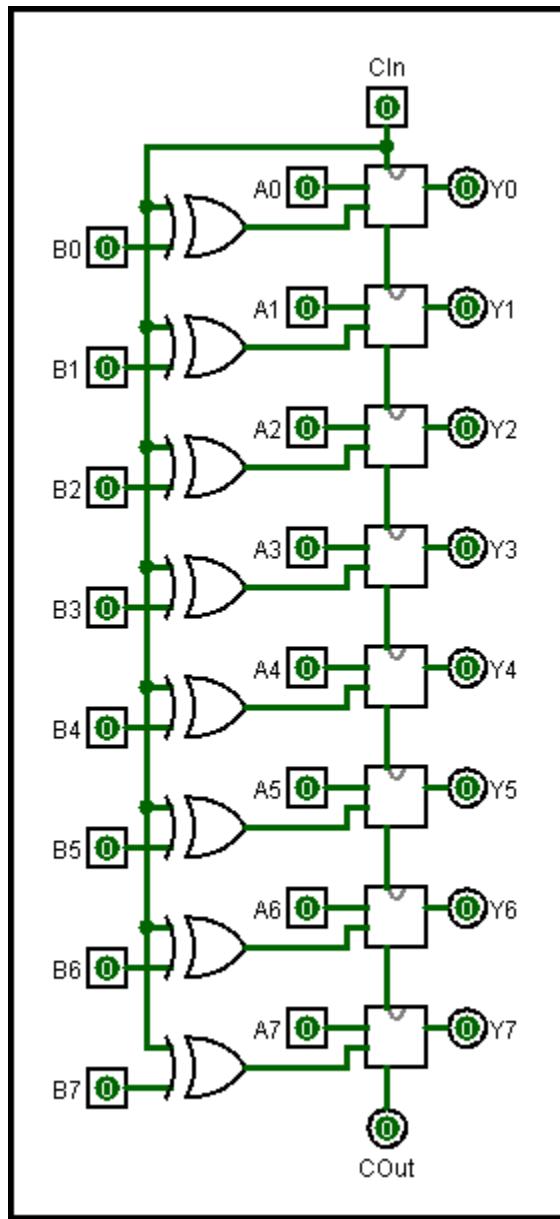


FIGURE 118: 8-BIT SUBTRACTOR

**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, A and 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  $1000_2 - 101_2 = 11_2$ .

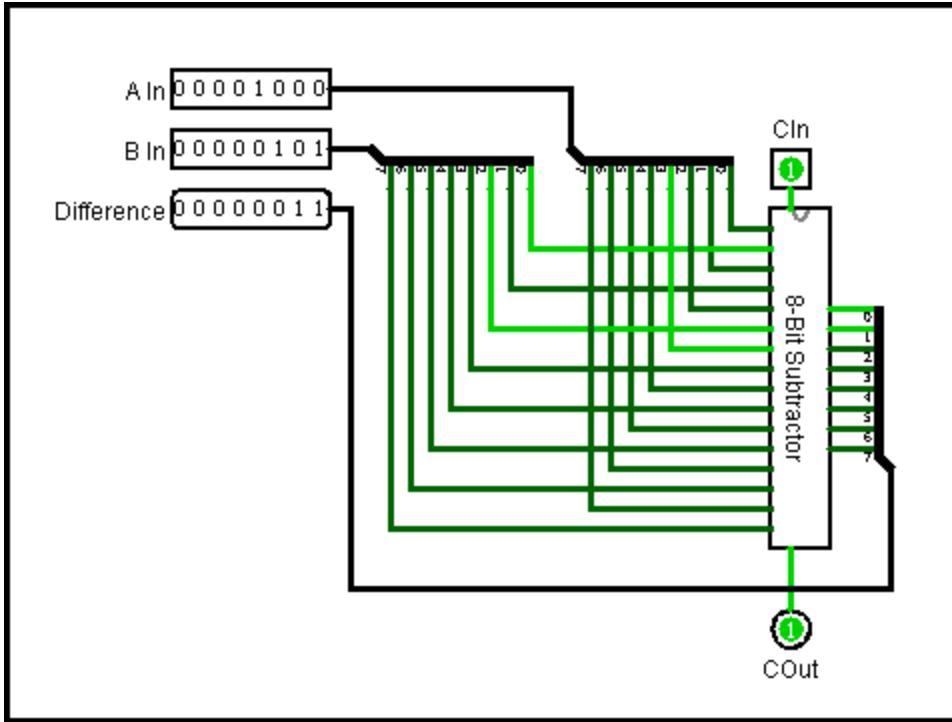


FIGURE 119: MAIN (ADDER) CIRCUIT

**CLEANUP**

Be sure the standard identifying information block is at the top left of the *Subtractor* circuit: Name, "Lab 5.4: 8-Bit Subtractor", and today's date. Save the file as *Lab 5\_4 – Subtractor*.

## 5.5: COMPARATORS

A comparator takes two binary inputs (A and B) and compares them. One of three outputs is generated:  $A = B$ ,  $A > B$ , or  $A < B$ .

A one-bit comparator satisfies these three Boolean equations:

$A = B$	$(A \oplus B)'$
$A > B$	$AB'$
$A < B$	$A'B$

TABLE 85: ONE BIT COMPARATOR

The following circuit satisfies the requirements for a one-bit comparator:

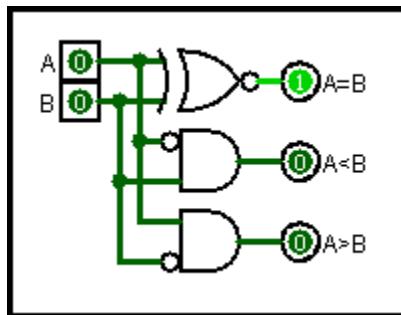


FIGURE 120: ONE-BIT COMPARATOR

It is possible to cascade one-bit comparators to compare larger numbers; however, as the number of input bits increases the comparator's logic becomes more complex.

### USING ARITHMETIC AND LOGIC UNIT (ALU) FLAGS

In complex applications that include an Arithmetic and Logic Unit (ALU), the function of a comparator can be created by using the “flags” (or indicators) generated by a subtraction problem. ALU circuits always include two flags of interest for this situation: Zero (set to “true” if the result is zero) and Negative (set to “true” if the result is negative). The following will clarify how a circuit designer can use an ALU output as a comparator. Assume that the ALU is set up to subtract input B from input A (that is,  $A - B$ ):

Input Condition	Zero Flag	Negative Flag
$A=B$	1	0
$A>B$	0	0
$A<B$	0	1

### 74F85 INTEGRATED CIRCUIT

As with most circuits, it is easier to use an IC to compare two inputs rather than create something from discrete gates. Following is the functional logic diagram for the 74F85 4-bit comparator. NOTE: this diagram is intended to show the logic function of the 74F85, it is not an accurate representation of the pinout of a physical IC.

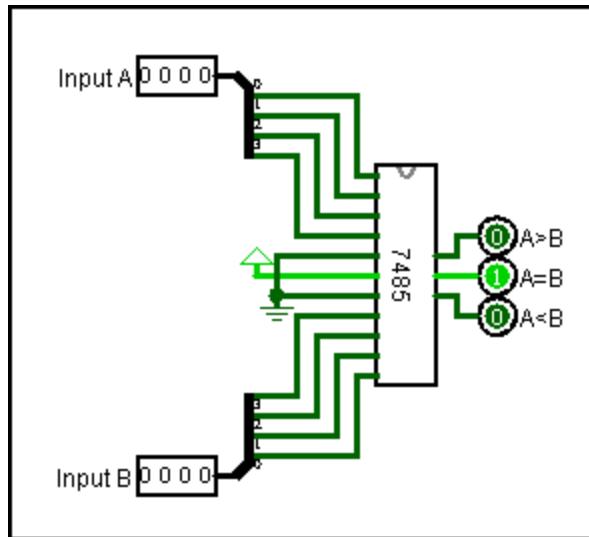


FIGURE 121: 74F85 4-BIT COMPARATOR

This device takes two 4-bit inputs and compares them. The appropriate output goes high while the others stay low. For example, if  $A = B$  then output  $A = B$  will be high while  $A < B$  and  $A > B$  will be low, as illustrated. Additionally, this IC can input the result of a previous comparison (the three input terminals in the center are labeled  $A < B$ ,  $A = B$ , and  $A > B$ ) so these ICs can be cascaded to create a comparator for binary numbers greater than 4-bits. NOTE: The 74F85 IC is designed in such a way that a constant high (or 1) must be present on the  $A = B$  input in order for the circuit to function; the two other inputs have been tied to ground (a constant 0) in order to keep them from floating.

To create an 8-bit comparator, two 74F85 ICs are cascaded. The four low order bits are fed into the first IC and the four high order bits are fed into the second IC:

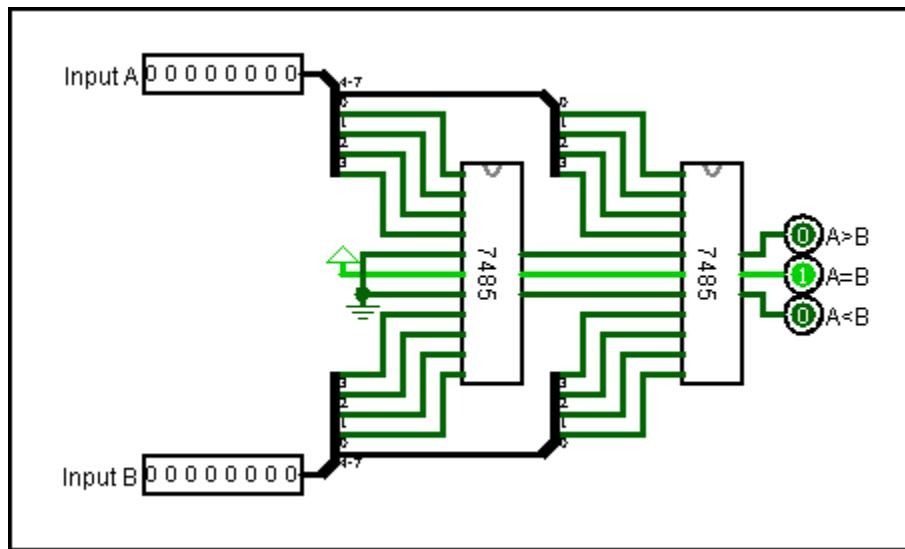


FIGURE 122: 74F85 4-BIT COMPARATORS CASCADED

## 5.6: ENCODERS/DECODERS

### INTRODUCTION

Encoders and decoders are used to convert some sort of coded data into a different code. For example, it may be necessary to convert some code created by a keyboard into ASCII for use in a word processor. The difference between an encoder and a decoder is the number of inputs and outputs: Encoders have more inputs than outputs, while decoders have more outputs than inputs.

As an introduction to encoders, consider the following circuit, which is designed to encode a single line (maybe the output of a push button on a control panel) into binary for further processing by the computer:

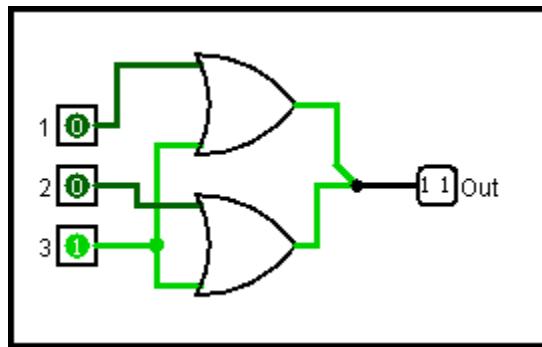


FIGURE 123: SIMPLE 4-LINE TO 2-LINE ENCODER

When Button 3 is pushed (as illustrated), both top and bottom OR gates are activated, which produces an output of 11. The next part of the circuit would then interpret that binary number in some specific way.

As an introduction to decoders, consider the following circuit, which is designed to decode a binary number and drive a single output line high. A circuit like this may be used to light a warning LED if some binary code was generated somewhere in a circuit.

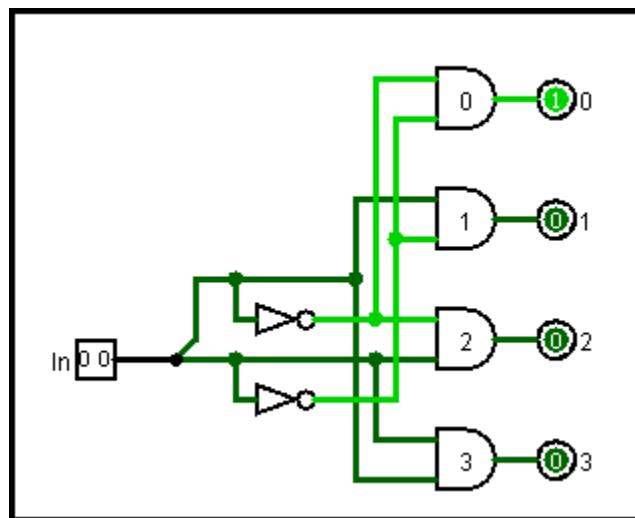


FIGURE 124: SIMPLE 2-LINE TO 4-LINE DECODER

When a binary code of 00 appears on the input lines (as illustrated), output 0 goes high, which may be used to generate a signal of some sort on an operator's console.

Both encoders and decoders are quite common and are used in many electronic devices. However, it is not very common to build these circuits out of discrete components (like in the circuits above). Rather, inexpensive integrated circuits are available for most encoder/decoder operations and these are much easier, and more reliable, to use.

#### 10-LINE TO BINARY ENCODER

Consider a 10-key keypad containing the numbers 0-9, like one that could be used for input from a numeric keypad to some device. A key press would need to be encoded to a binary number for further processing. That job is handled by a 10-line to 4-line encoder (that is, 10 input lines, like from a keypad, are encoded to 4 output lines, or 4-bit binary). Illustrated below is a 10- to 4-line encoder using gate-level logic.

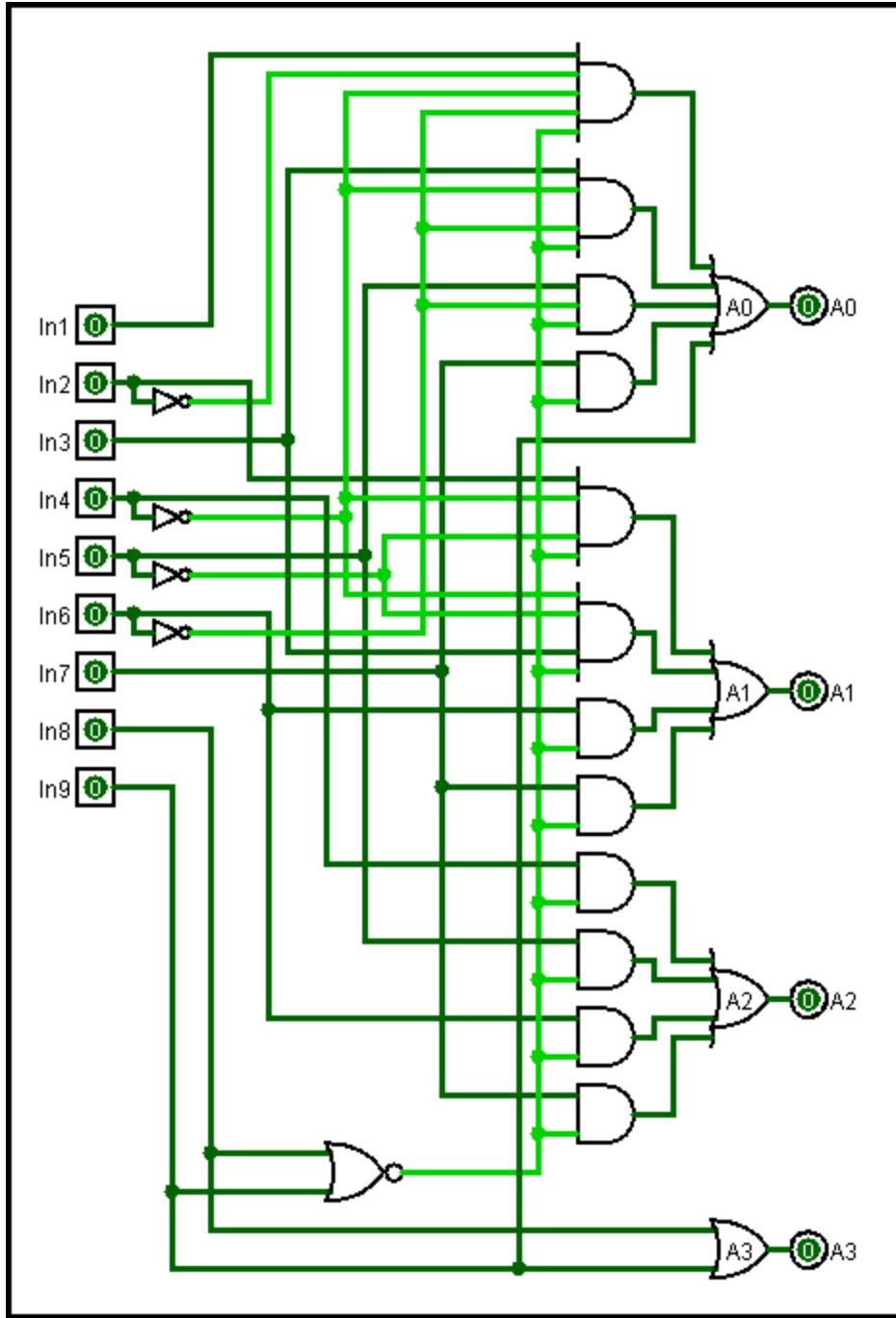


FIGURE 125: 10-LINE TO 4-LINE PRIORITY ENCODER

The above circuit is based on the Phillips 74HC147 IC, which is a 10-line to 4-Line priority encoder. Since it is a "priority" encoder, if more than one line goes high at the same time, the one with the highest number would "win"; that is, line 8 is a higher priority than line 7, so line 7 would have no effect on the circuit if line 8 were active. Following is a truth table for the encoder illustrated above:

Input									Output			
1	2	3	4	5	6	7	8	9	A3	A2	A1	A0
-	-	-	-	-	-	-	-	H	1	0	0	1
-	-	-	-	-	-	-	-	H	1	0	0	0
-	-	-	-	-	-	-	-	H	0	1	1	1
-	-	-	-	-	-	-	-	H	0	1	1	0
-	-	-	-	-	-	-	-	H	0	1	0	1
-	-	-	-	-	-	-	-	H	0	1	0	0
-	-	-	-	-	-	-	-	H	0	0	1	1
-	-	-	-	-	-	-	-	H	0	0	1	0
-	-	-	-	-	-	-	-	H	0	0	0	1
-	-	-	-	-	-	-	-	-	0	0	0	0

TABLE 86: PRIORITY ENCODER TRUTH TABLE

Notes: 1) For clarity, Low inputs were left blank. 2) Dashes indicate "don't care" states. That is, for example, if input 5 is high, then it does not matter if inputs 0-4 are high or low. 3) If all input lines are low, the output is assumed to be 0000, so the IC does not specifically check for that input condition.

#### BCD To 7-SEGMENT DECODER



FIGURE 126: SEVEN SEGMENT DISPLAY-GALLERY

A 7-segment display<sup>1</sup> is commonly used for decimal or hexadecimal numbers. Each segment of a single digit can be turned on or off to produce a decimal or hexadecimal number. The following illustrates a single 7-segment number with each segment labeled<sup>2</sup> to indicate the input pin responsible for that segment:

<sup>1</sup> 7-Segment Display image supplied by Oliver Wolters and found at Wikimedia Commons: [http://upload.wikimedia.org/wikipedia/commons/2/2c/Seven\\_segment\\_display-gallery.png](http://upload.wikimedia.org/wikipedia/commons/2/2c/Seven_segment_display-gallery.png). It is used under the terms of the GNU Free Documentation License, Version 1.2.

<sup>2</sup> 7-Segment Display with Labels supplied by “h2g2bob” and found at Wikimedia Commons: [http://commons.wikimedia.org/wiki/File:7\\_segment\\_display\\_labeled.svg](http://commons.wikimedia.org/wiki/File:7_segment_display_labeled.svg). It is used under the terms of the GNU Free Documentation License, Version 1.2.

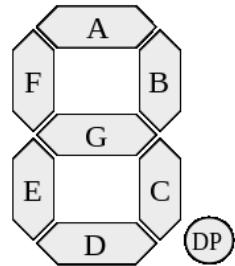


FIGURE 127: 7-SEGMENT DISPLAY WITH LABELS

The various segments light to form digits. For example, the number "1" is formed when b and c are lit, and "5" by lighting a, f, g, c, and d. There is usually a decimal point on 7-segment displays; but, for simplicity, that has been ignored in this module.

The 7448 Decoder takes a 4-bit binary input and produces an output designed to drive a 7-segment display. The following diagram shows a simplified 7448; the circuit does not include either input or output Ripple Blanking (which blanks leading zeros in a multiple-digit display), but does include all other 7448 functions.

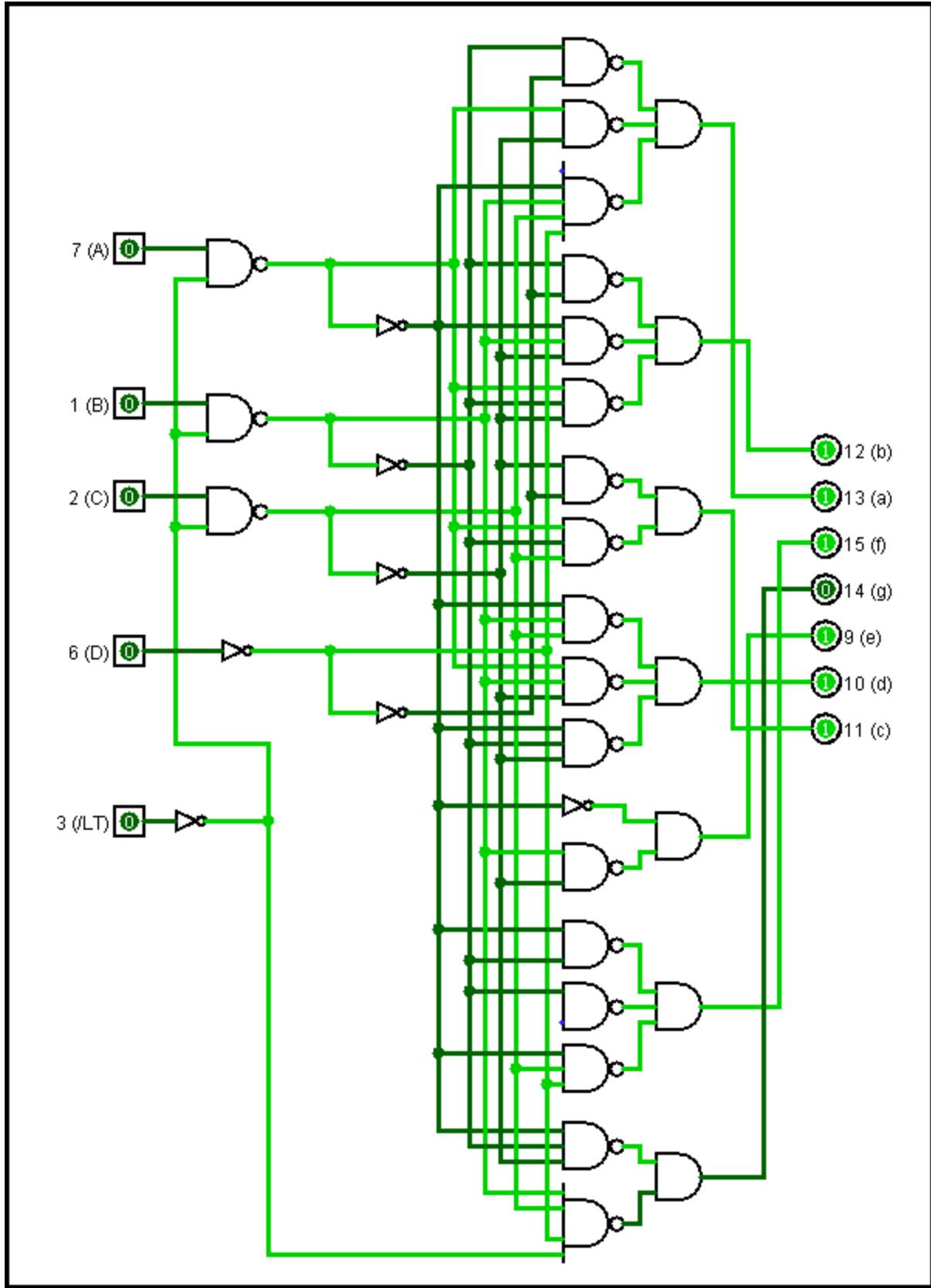


FIGURE 128: BCD TO 7-SEGMENT DECODER

The 4-bit hexadecimal number is input on pins A-D at the top left of the diagram (NOTE: pin numbers for the 7448 are indicated in this circuit; for example, input "A" is made on pin 7 of the IC). The output pins (a-g) are driven high to activate the appropriate segments on a 7-segment display. As shown, the input number is 0000, and the output lights segments A-F, which is 0 on the display. This circuit also includes a "Light Test" ("LT"), and when that is active then all LED segments will be activated.

Following is the truth table for the 7448 IC. The "Low" outputs have been left blank to make it easier to see which segments are lit for each input number. Also, keep in mind that the 7448 is designed for only BCD input; therefore, the output for numbers greater than 9 is just a senseless pattern on the display. Also, note that for input 15 (all 1's), the output is to blank the display.

Hex	Input	a	b	c	d	e	f	g
0	0000	H	H	H	H	H	H	
1	0001	H	H					
2	0010	H	H		H	H		
3	0011	H	H	H				H
4	0100	H	H			H	H	
5	0101	H		H	H	H		
6	0110		H	H	H	H	H	
7	0111	H	H	H				
8	1000	H	H	H	H	H	H	
9	1001	H	H	H		H	H	
A	1010			H	H	H		
B	1011			H	H			
C	1100	H				H	H	
D	1101	H			H		H	
E	1110			H	H	H	H	
F	1111							

FIGURE 129: 7448 TRUTH TABLE

### **MiniLab 5.6**

Logisim includes a 7-Segment Display. Create a simple circuit like the one illustrated below and explore how this device works by activating and deactivating various segments. NOTE: use the 7-Segment Display component, not the Hex Display component. Be sure the standard lab identifying information is at the top left of the circuit: Name, "MiniLab 5.6", and today's date. Then save the file with this name: "minilab\_5\_6".

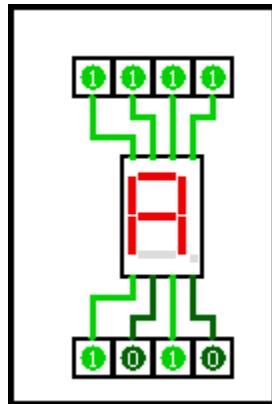


FIGURE 130: 7-SEGMENT DISPLAY

#### DECODERS AS FUNCTION GENERATORS

Decoders provide an easy way to create a minterm function generator circuit. The Motorola 74LS138 integrated circuit is a 3-of-8 decoder, which means that an input of three bits (a 3-bit number) is decoded into one of eight output lines. Imagine that a circuit is needed with this minterm output:

$$\sum(A, B, C) = \sum(0, 2, 7)$$

Whatever this circuit is designed to do, it should activate an output only when the input is 000, 010, or 111. The 3-to-8 circuit could then be wired like this:

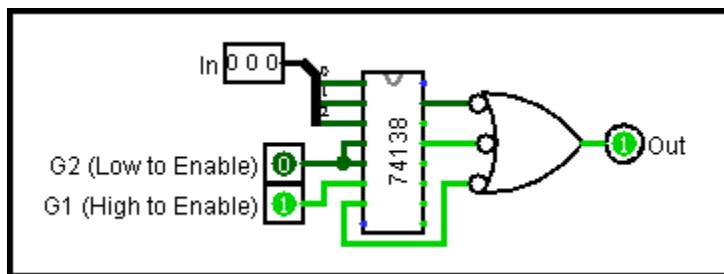


FIGURE 131: FUNCTION GENERATOR USING A 74138 DECODER

The 74138 IC has three input pins at the top left corner. The next two pins down are G2A and G2B; they are tied together in the circuit and must both be low for the circuit to function. They are useful when this IC is cascaded with several others to provide an expanded capacity. The G1 pin is an "enable" pin and must be high for the decoder to operate. All of the pins on the right side of the IC, plus the pin in the lower left corner, are outputs, numbered from 0 to 6 on the right side with output 7 on the left side. These are "active low" outputs; that is, when the output is "True" the 74138 actually outputs a low rather than high on that pin. For that reason, the inputs of the OR gate are inverted (notice the bubble on the inputs); that way, when a "low" is output from the 74138 it is felt as a "high" in the OR gate..

The OR gate is only active when the input to the IC is equal to 000 (as illustrated), 010, or 111 since those are the only pins connected to the OR gate; any other input would be ignored. Of course, the output would be used to activate some other circuit, perhaps to turn on a cooling fan, change a traffic light, or some other function. This makes the circuit behave like a function generator.

### CASCADING DECODERS

Decoders are available in a wide variety of sizes; however, the number of input/output lines can quickly outstrip the physical size of the integrated circuit. For example, a decoder designed to handle 6 input variables would require 6 input lines and 32 output lines; and this would be much greater than the 14 pins commonly found in a dual inline (IC) package. For that reason, decoders are often cascaded in order to get the required number of input and output lines at the lowest cost. Consider the following function:

$$\sum(A,B,C,D) = \sum(0,5,10)$$

This function requires a 4-to-16 decoder. For practical purposes, designers often only use 3-to-8 decoder ICs, so two of these would need to be cascaded to provide the required number of output ports. The best place to start with this problem is the truth table:

Input				Output
A	B	C	D	Q
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Notice that the most significant bit of the input is low for the first eight rows and high for the next eight. This input, then, can be used to activate either the first (or low order) or second (or high order) decoder. Here is the circuit using a 74LS138 integrated circuit introduced above:

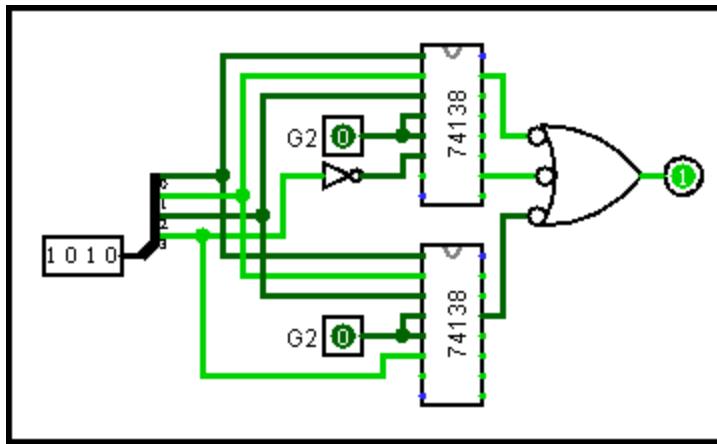


FIGURE 132: CASCADING DECODERS

The trickiest part of this circuit is the cascade. The four bits of the input number are split so that the high order bit is used to activate either the top or bottom decoder. When that bit is high, as illustrated, the bottom decoder is activated (the input line under pin G2 is the enable pin), but the inverter in the line going to the top decoder will inactivate that device; the opposite happens when the high order bit is low, the top decoder is activated while the bottom one is inactivated.

The truth table for this circuit indicates that the first "True" input combination is 0000. Since input A (the high order bit in the selector) is low, then the top decoder is enabled due to the inverter in the control line and the bottom decoder is disabled. The other three inputs are also low, so output 0 is activated on the top decoder, and the output from the OR gate goes high. In the same way, input 0101 activates output 5 on the top decoder, which causes the output to go high. Finally, input 1010 activates output 3 on the bottom decoder and the output goes high (this is the state of the circuit in the illustration).

#### EXAMPLE PROBLEM

Given: Build a circuit with three inputs (labeled A, B, and C) and one output (labeled Q). The output, Q, should be true when A, B, and C are all true; when C is true but A and B are false; and when A is true but B and C are false. Here is the truth table for this problem:

Input			Out
A	B	C	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

This truth table leads to this Canonical Equation:

$$Q = A'B'C + AB'C' + ABC$$

Minterm Function expressed as in Sigma notation:

$$\sum(A, B, C) = \sum(1, 4, 7)$$

Realized Circuit:

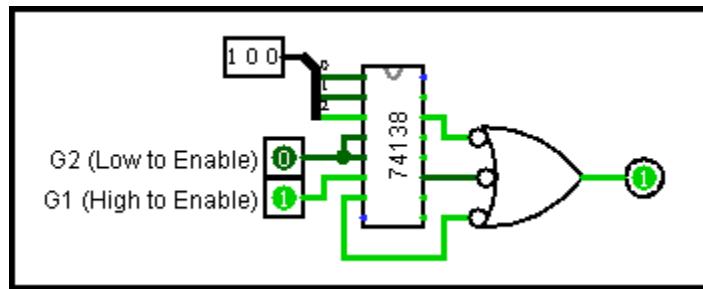


FIGURE 133: MINTERM GENERATOR FOR EXAMPLE PROBLEM

## 5.7: LAB – 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. Wire a clock to the clock input of the counter (it is on the bottom edge and is marked with a triangle). The top input port on the west side of the counter should have a constant 0 and the bottom input port on the west side of the counter should have a constant 1. Finally, wire a button to the "Clear" port on the east side of the bottom edge. When pressed, this button will reset the counter back to 0.

Place a register (memory library) to the right of the counter and specify 4 Data Bits for the register. Wire the counter's Q output to the Data (D) input of the register. Wire a button to the clock input (west side of the south edge, marked with a triangle) on the register.

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.

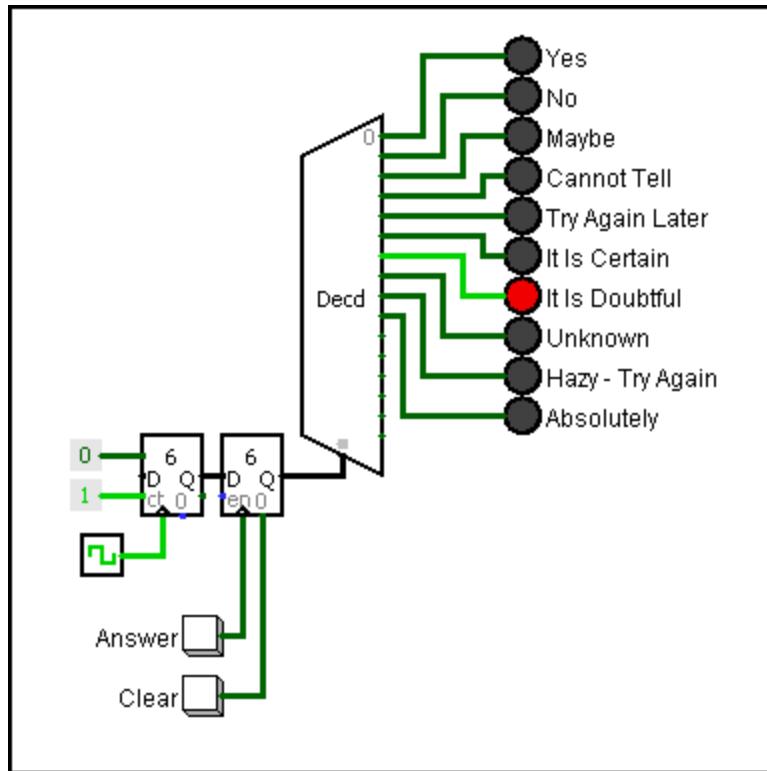


FIGURE 134: ELECTRONIC MAGIC 8-BALL

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 at random, giving you an answer to whatever question was asked. Next, speed up the clock's frequency and let it run. Ask a question (like "Am I smart or what?") 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 5.7: Magic 8-Ball", and today's date. Save the file as *Lab 5\_7 – 8Ball*.

## 5.8: MULTIPLEXERS/DEMULITPLEXERS

### INTRODUCTION

A multiplexer (often called "mux") is used to connect one of several input lines to a single output line. Thus, it selects which input can pass to the output. This function is similar to a rotary switch where several potential inputs to the switch can be sent to a single output. A demultiplexer (often called "demux") is a multiplexer in reverse, so a single input can be fed to any of several outputs. The following simplified diagram illustrates a multiplexing system.

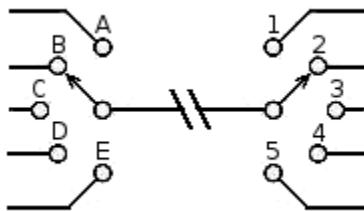


FIGURE 135: MULTIPLEXER CONCEPT USING ROTARY SWITCHES

Here is the schematic design of a multiplexer built with discrete components. There are four data lines (In0-In3) and two control lines (Sel0-Sel1). When  $10_2$  is input on the control, then whatever data is coming in on In1 will be passed to the output port. In this way, the four different inputs can share a single output line.

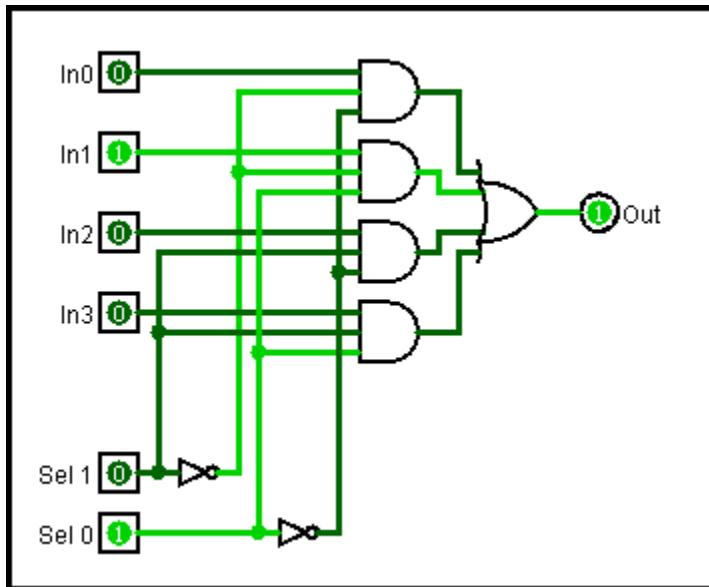


FIGURE 136: MULTIPLEXER CONCEPT USING DISCRETE COMPONENTS

### MULTIPLEXER IN TRANSMISSION SYSTEM

In practice, a simple data transmission system can be easily built using a mux/demux pair. This makes it possible, for example, for a single copper wire to carry multiple telephone conversations by rapidly switching between the various inputs/output pairs (of course, timing is critical in this type of

application). This would be appropriate for a small application, like the security system for a single building. Consider the following circuit:

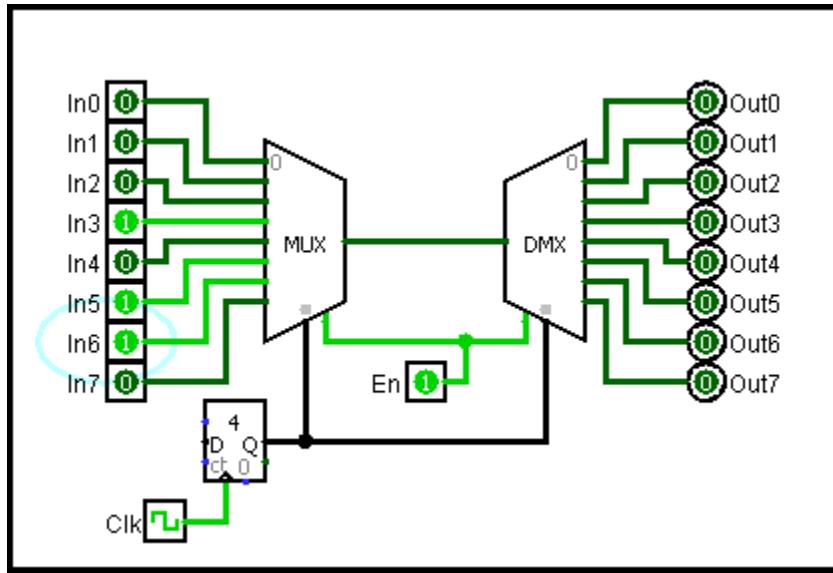


FIGURE 137: MULTIPLEXER-DEMULTIPLEXER TRANSMISSION SYSTEM

Imagine the multiplexer on the left has eight different door or window sensors sending signals. Those signals are multiplexed onto a single wire and sent to a guard station where they are demultiplexed into eight warning lights on a panel. A timing signal is sent from a ring counter to the mux/demux pair so they are synchronized, and an Enable switch turns the system on. When any one input line goes high (someone has opened a door, for example), then that signal will turn on a light at the guard station. Because only one data wire and three control wires are used, the result is a savings over the eight wires needed to connect the system without the mux/demux pair. Moreover, the one data and three control wires would be able to easily synchronize many more sensors/lights, so the system becomes more economical as it grows.

#### MULTIPLEXERS AS MINTERM GENERATORS

Multiplexers are similar in design to encoders, except a multiplexer includes a "select" function. Thus, a multiplexer can be used as a minterm generator. Consider this Boolean function:

$$\sum(A,B,C,D) = \sum(1,3,4,11,12,13,14,15)$$

A multiplexer can be wired to create a minterm generator that will only activate an output signal when any of the inputs indicated above are active:

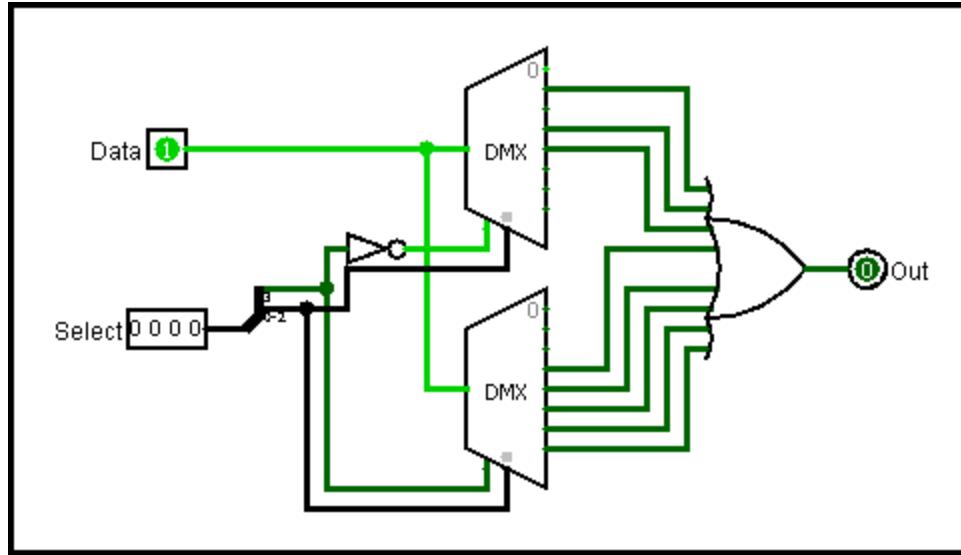


FIGURE 138: MULTIPLEXER AS MINTERM GENERATOR

Because there are four signal lines (A, B, C, D), two 2-bit multiplexers are cascaded in a way similar to that for decoders. The high order select bit is used to activate either the top or bottom multiplexer while the other three bits are used to select the specific output line on that multiplexer. When the select bit is set for  $15_{10}$ , then the bottom multiplexer would be activated and line 7 on that device would be activated. This is an easy way to create a minterm generator for a Boolean function.

## 5.9: LAB – ADDING MACHINE

### PURPOSE

This lab builds a simple adding machine that will add two different 2-digit numbers and display the results in decimal.

### PROCEDURE

This is a reasonably complex circuit. While the idea is simple, add two numbers, the execution is quite involved.

### KEYPAD

Start by placing 9 buttons in a row on the left side of the workspace. These will be the 9 digit inputs, so label them 0 through 9, with 0 at the top. The outputs of these 9 buttons should be fed into a Priority Encoder (Plexers library). Specify 4 select bits for the encoder. Note that some of the encoder's inputs are not used; these are for the digits a-f. Connect a constant High to the enable pin of the encoder.

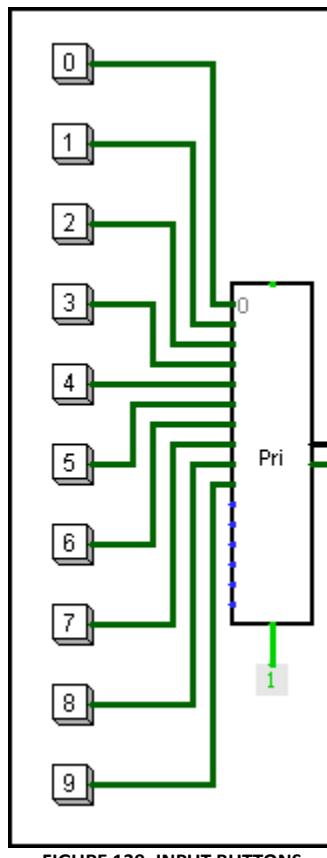


FIGURE 139: INPUT BUTTONS

The output of the priority encoder goes to the input of a Shift Register (Memory library). Specify that the register has 4 Data Bits, 2 Stages, and the Parallel Load is "Yes." This shift register is set up to contain 2 4-bit numbers. It will take the output of the priority encoder, which is a 4-bit BCD number, and shift that

number to the right in the register. The register will constantly contain the last two numbers clicked on the numeric keypad. The easiest way to see how a shift register works is to start the simulator, click a few of the number buttons, and observe the contents of the register.

The "Group Select" pin on the Priority Encoder will pulse high when any of the number buttons are pressed (that is, there is some input to the encoder). Wire that pin to the clock input on the shift register.

Finally, wire the output on the east end of the south edge to a hex display and the output for the next pin west to a second display. This will help troubleshoot the circuit. The output pin farthest east is the most significant byte of the 2-digit number input. At this point, the circuit should look like this:

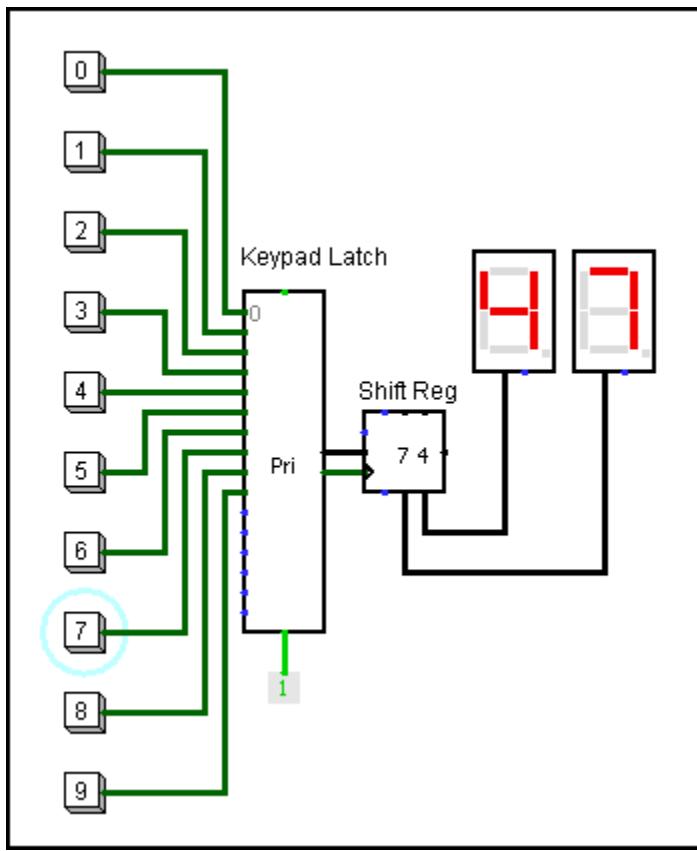


FIGURE 140: SHIFT REGISTER ADDED TO CIRCUIT

Notice that a few labels have been added to the circuit. This is to help with troubleshooting since the final circuit will be rather complex. To test the circuit, poke the buttons and notice that the output will display the last 2 digits entered. Do not be concerned with clearing the display; that will come in a later step.

#### BCD TO 12-BIT BINARY

Next the 4-bit BCD numbers need to be converted to a 12-bit binary code. This must be done since it is very complicated to add BCD codes with a digital circuit; however, adding binary numbers is a snap. It is far easier to convert the BCD numbers stored in the shift register to a 12-bit binary number than to try

to work with BCD in the circuit. Later, the result of adding 2 12-bit binary numbers will be converted back to BCD to display to the user.

BCD is nothing more than a 4-bit binary number between 0000 and 1001 (which is decimal 0-9). Therefore, to convert this BCD to a 12-bit binary is as simple as making bits 4-11 of the 12-bit number equal to zero. The following circuit shows how this is done:

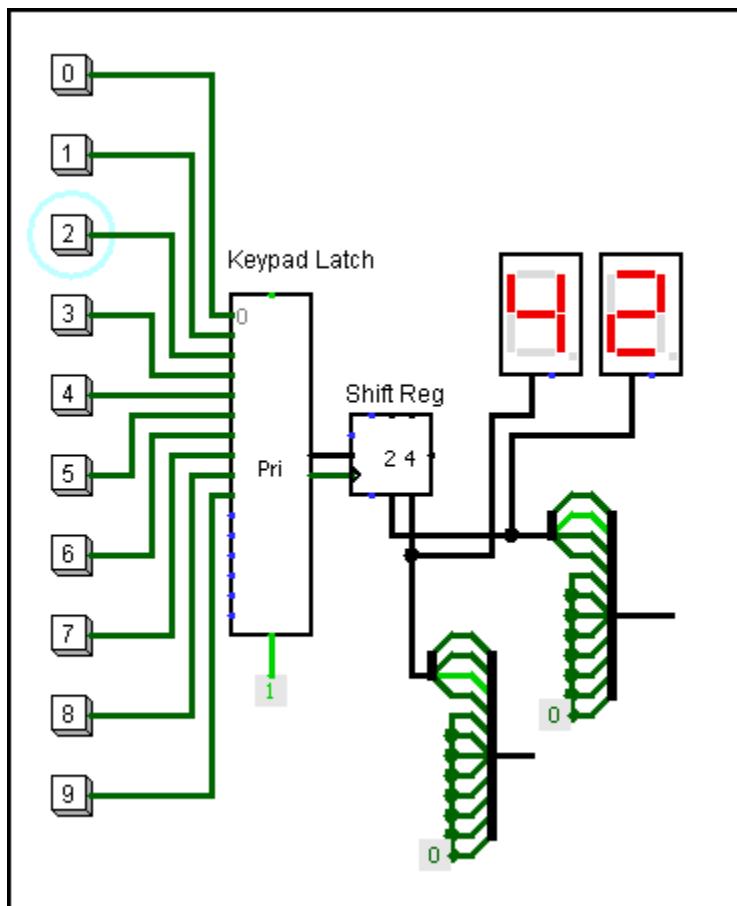
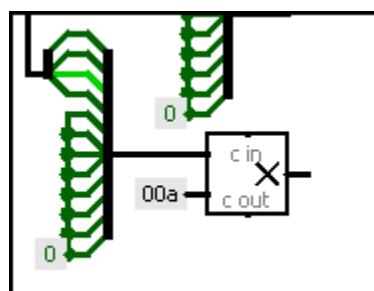


FIGURE 141: CONVERTING BCD TO BINARY

It is important to keep in mind that the most significant byte of the register is the one on the right (the east output on the south edge). This actually represents the 10s digit of the decimal input. Therefore, it must be multiplied by decimal 10 before it is added to the least significant byte. To do that, a multiplier is added to the circuit and a constant of A (decimal 10) is added to that byte:



**FIGURE 142: MULTIPLYING THE MOST SIGNIFICANT BYTE BY 10**

## ADDING TWO 12-BIT NUMBERS

The two 12-bit numbers are now added to create a single 12-bit number that represents the low-order byte plus the (high-order byte X 10):

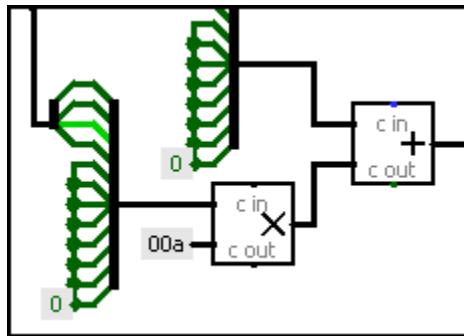


FIGURE 143: 2 12-BIT NUMBERS ADDED

## REGISTER STORAGE

Each of the 2 2-digit numbers must be stored in a register so they can be added. Once the 12-bit number is created, it is sent to one of two registers for storage. To do that, the output of a dmux is toggled from one register to another. The 12-bit version of the first 2-digit number entered goes to one of the registers and the 12-bit version of the second 2-digit number goes to the other. Here is the part of the circuit that does that storage:

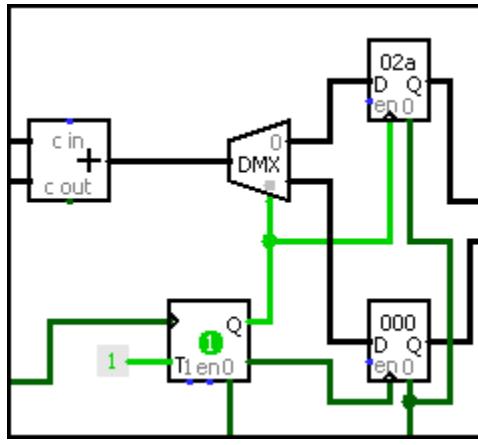


FIGURE 144: REGISTER STORAGE OF NUMBERS

Starting with the adder at the top left of the above illustration, the 12-bit number goes to a demultiplexor. If the select bit (on the south edge of the demultiplexor) is high, then that number is stored in the top of the two registers. If the select bit is low, then the number is stored in the bottom of the two registers. The select bit is controlled by a T flipflop. A constant high on the T input pin means that the outputs will change on every clock pulse. The "clock" in this case is connected to the "+" button. Thus, every time the user clicks "+" the demultiplexor will change and whatever number is at the input of the demultiplexor will be transferred to either the top or bottom register. Also notice that the Q and Q' outputs of the T flip-flop connect to the clock pin of the registers. This means that not only does the flip-flop change the demultiplexor setting, but it also strobes the appropriate register so it stores whatever 12-bit number the demultiplexor has sent it.

**HEX DISPLAY**

The only remaining part is to actually add the two 12-bit numbers in the registers and show the result on Hex Digit Displays:

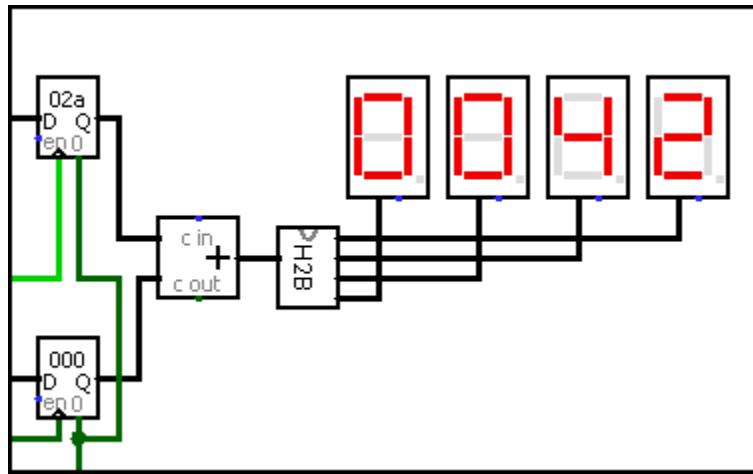


FIGURE 145: HEX DISPLAY OUTPUT

**12-BIT BINARY TO BCD**

The Q output from the register is fed into a 12-bit adder. The output of that adder is fed into a 12-bit binary-to-BCD circuit in order to drive the four Hex Digit Displays. The theory behind the conversion from 12-bit binary to BCD is beyond the scope of this lab; however, the circuit is based upon information found in the data sheet for IC 74x85.

Create a new sub-circuit named Bin2BCDInter. That sub-circuit will contain the logic needed to convert a binary input into an intermediate BCD form that can later be used for a final BCD output. It is easiest to use the Logisim Circuit Analyzer and let Logisim build the circuit automatically. Open the analyzer and specify A, B, C, D, and E as inputs; Y1, Y2, Y3, Y4, Y5, and Y6 as outputs; and then copy/paste the following expressions. Finally, have Logisim build the intermediate circuit automatically.

$$\text{Y1: } \begin{aligned} &\sim A \sim C \sim D \sim E + \sim A \sim B \sim C D \sim E + \sim A \sim B C D E + \sim A B \sim C E + \sim A B C \\ &\sim D \sim E + A \sim C \sim D \sim E + A C \sim D E + A \sim B C D \sim E + A B \sim C \sim E + A B C E \end{aligned}$$

$$\text{Y2: } \begin{aligned} &\sim A \sim B D \sim E + \sim A \sim B C D + \sim B C D \sim E + \sim A B \sim C \sim D + B \sim C \sim D \sim E + \\ &\sim A B \sim D E + A \sim B \sim C \sim D E + A B \sim C D E + A B C \sim D \end{aligned}$$

$$\text{Y3: } \begin{aligned} &\sim A \sim B \sim C D E + \sim A \sim B C \sim D \sim E + \sim A B C D \sim E + A \sim B \sim C D \sim E + A \\ &\sim B C D E + A B \sim C \sim D E \end{aligned}$$

$$\text{Y4: } \begin{aligned} &\sim C \sim D E + \sim B \sim C D \sim E + \sim B C D E + B \sim C E + B C \sim D \sim E + A \sim C E \\ &+ A C \sim D \sim E + A B C \sim E \end{aligned}$$

$$\text{Y5: } \sim C \sim D E + C D \sim E + B D \sim E + B C D$$

$$\text{Y6: } \sim B D E + C E + A D E$$

Once the Bin2BCDInter Circuit is ready, create the following new circuit named Bin2BCD. Each of the boxes labelled *Intrmed* in the circuit is an instance of the Bin2BCDInter Circuit created above. The input pin is 12-bits. Run the input through a 12-bit splitter and then into the various Bin2BCDInter circuits as shown.

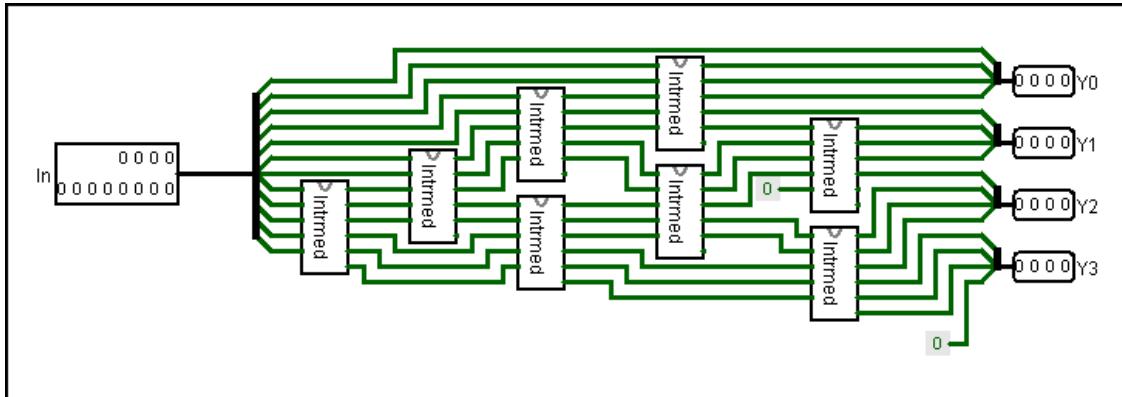


FIGURE 146: BINARY-TO-BCD CIRCUIT

The final output of this subcircuit is a group of four 4-bit output pins, labeled Y0-Y3. These will carry the BCD signal to each of the 4 Hex displays on the main circuit.

#### COMPLETE CIRCUIT

Following is the complete adding machine main circuit:

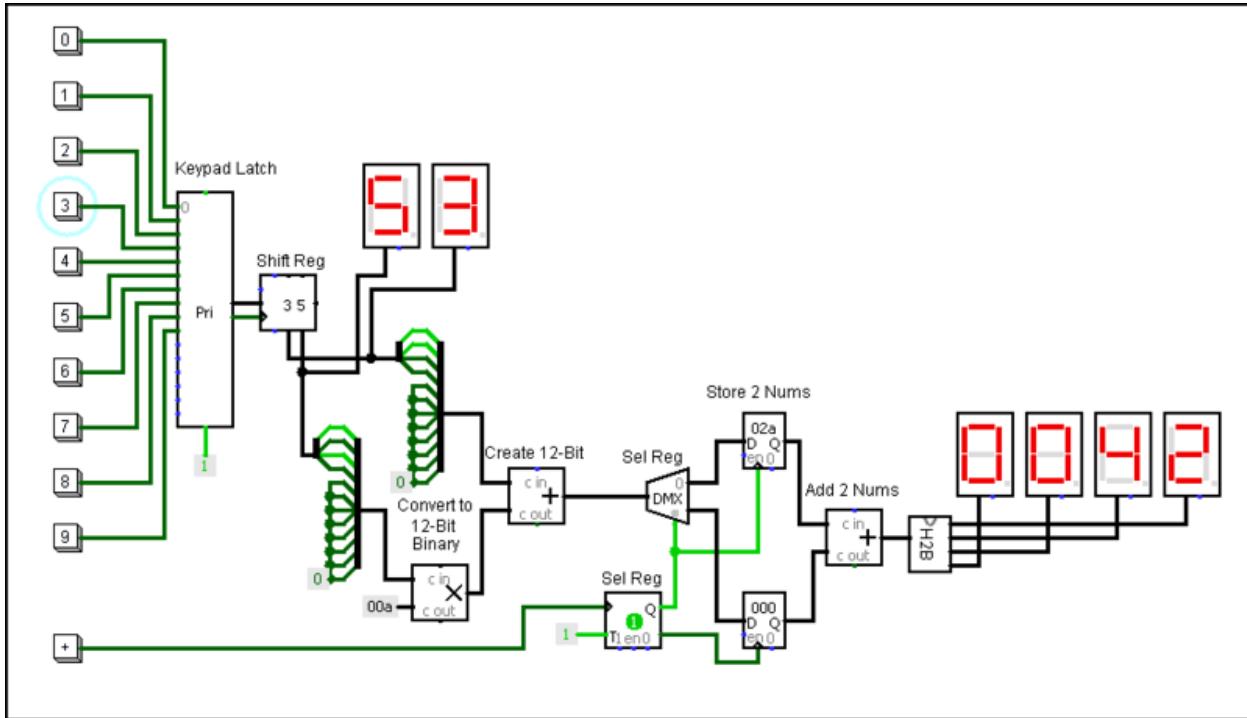


FIGURE 147: ADDING MACHINE

**CHALLENGE**

The Adding Machine is nearly complete. However, it still needs a reset so all of the various registers can be cleared. Add a button labeled "Reset" below the "+" button. Wire that button to the "Clear" port of the T flip-flop that controls the register select demultiplexor. Also connect the reset signal to the "Clear" ports for both of the 12-bit registers. Finally, the reset signal should be routed to the "Clear" pin of the Shift Register. It may be somewhat confusing to the user to keep the input number in the Shift Register while a new number is being entered. Therefore, also find a way to clear the Shift Register every time the "+" button is clicked.

**CLEANUP**

Rename the *main* circuit to *Add*. Be sure the standard identifying information block is at the top left of the *Add* circuit: Name, "Lab 5.9: Adding Machine", and today's date. Save the file as *Lab 5\_9 – Adding Machine*.

## 5.10: ERROR DETECTION

### INTRODUCTION

Whenever a byte (or any other group of bits) is transmitted or stored, there is always the possibility that one or more bits will be accidentally complemented. Consider these two binary numbers:

```
0110 1010 0011 1010
0110 1010 0111 1010
```

They differ by only one bit (look in group #3). If the first number is what is supposed to be in some memory location, but the second number is what is actually in that location, then there is obviously a problem. There could be any number of reasons why a bit would be wrong, but the most common is some sort of error that creeps in from somewhere while the byte is being transmitted between two stores, like a hard drive and memory in one computer or between computers located at two different locations. It is desirable to be able to detect that a byte contains a bad bit and, ideally, even know which bit is wrong so it can be corrected.

Parity is a common method used to check data for errors and it can be used to check data that has been transmitted, held in memory, or stored on a hard drive. The concept of parity is fairly simple: A bit (called the "parity bit") is added to each data byte and that extra bit is either set or reset in order to make the bit-count of that byte contain an even or odd number of 1s. For example, consider this binary number:

**1101**

There are three 1's in this number, which is an odd number of 1's. If "odd" parity is being used in this circuit, then the parity bit would be 0 and there would still be an odd number of 1's in the number. However, if the circuit is using even parity, then the parity bit would be set to 1 in order to have 4 1's in the number, which is an even number of 1's. Following is the above number with both even and odd parity bits (those parity bits are in the least significant position and are separated from the original number by a space for clarity):

**1101 0 (Odd)**  
**1101 1 (Even)**

Following table shows several examples that may help to clarify this concept. In each case, a parity bit is used to make the data byte even parity (spaces were left in the data byte for clarity).

Data Byte	Parity Bit
0000 0000	0
0000 0001	1
0000 0011	0
0000 0100	1
1111 1110	1
1111 1111	0

TABLE 87: PARITY BIT DEMO

Parity is a simple concept and is the foundation for one of the most basic methods of error checking. As an example, if some byte is transmitted using odd parity, but the data arrives with an even number of bits, then one of the bits was changed during transmission.

#### ITERATIVE PARITY CHECKING

One of the problems with using parity for error detection is that there is no way to know which of the bits is wrong. For example, imagine an 8-bit system is using even parity and receives this data and parity bit:

**1001 1110 PARITY: 0**

There is something wrong with the byte. It is using even parity but has an odd number of 1's in the byte. It is impossible to know which bit changed during transmission. In fact, it may be that the byte is correct but the parity bit itself changed (a "false error"). It would be nice if the parity error detector would not only indicate that there was an error, but could also determine which bit changed so it could be corrected.

One method of error correction is what is called Iterative Parity Checking. Imagine that a series of 8-bit bytes were being transmitted. Each byte would have a parity bit attached; however, there would also be a parity byte that contains a parity bit for each bit in the preceding five bytes. It is easiest to understand this by using a table (even parity is being used):

Byte	Data Bits	Parity Bit
1	00000000	0
2	10110000	1
3	10110011	1
4	11101010	1
5	01000000	1
P	10101001	0

TABLE 88: ITERATIVE PARITY

In this table, transmitted Byte 1 is 0000 0000. Since the system is set for even parity, and it is assumed that a byte with all zeros is even, then the parity bit is 0. Each of the five bytes has a parity bit that is properly set such that each byte (with the parity bit) includes an even number of bits. However, after every group of five bytes a "parity byte" is transmitted so that each column of five bits also has a parity check; and that parity bit is found in row "P" on the table. Thus, the parity bit at the bottom of the first column is one since that column has three other ones. As a final check, the parity byte itself also has a parity bit added.

Here is the same table again, but Bit 0, the least significant bit, in Word 1 has been changed from a 0 to a 1 (that number is in bold font):

Byte	Data Bits	Parity Bit
1	00000001	0
2	10110000	1
3	10110011	1
4	11101010	1
5	01000000	1
P	10101001	0

TABLE 89: ITERATIVE PARITY WITH ERROR

Now, the parity for Byte 1 is wrong, and the parity for Bit 0 in the parity byte is wrong; therefore, Bit 0 in Byte 1 needs to be changed. If the parity bit for a row is wrong, but no column parity bits are wrong, or a column is wrong but no rows are wrong, then the parity bit itself was changed. This is one simple way to not only detect data errors, but correct those errors.

The weakness with iterative parity checking is that it is restricted to only single-bit errors. If more than one bit is changed in a group then the system fails. This, though, is a general weakness for most parity checking schemes.

#### HAMMING CODE

Richard Hamming worked at Bell labs in the 1940s and he devised a way to not only detect that a transmitted word had changed, but exactly which bit had changed by interspersing parity bits within the word. Hamming first defined the “distance” between any two words as the number of bits that were different between them. As an example, the two binary numbers 1010 and 1010 has a distance of 0 between them since there are no different bits. In the same way, 1010 and 1011 has a distance of one since one bit is different. This concept is called the **Hamming Distance** in honor of his work.

Hamming decided to make bits in positions that are even powers of two parity bits while everything else is data. Thus, parity bits are at bit 1, 2, 4, 8, 16, 32, etc. Moreover, each parity bit was set or not based on a group of overlapping data bits; designed in such a way that every data bit is covered by at least two different parity bits. Consider this table:

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Purpose	P1	P2	d1	P4	d2	d3	d4	P8	d5	d6	d7	d8	d9	d10	d11	P16
P1 Cover	X		X		X		X		X		X		X		X	
P2 Cover		X	X			X	X			X	X			X	X	
P4 Cover				X	X	X	X				X	X	X	X	X	
P8 Cover								X	X	X	X	X	X	X	X	
P16 Cover																X

TABLE 90: HAMMING PARITY CHECKING



**NOTE:** for the purposes of working with Hamming Codes, bits are normally counted from left to right starting at 1 rather than 0. This is different from binary number bit counts, which count right to left and starts at 0; but it makes Hamming codes easier to understand.

In a system that uses Hamming codes, bits 1, 2, 4, 8, and 16 are parity bits and all others are data bits. It is easy to remember the parity bit positions since they are powers of two. Since some of the bits are parity bits rather than data bits, a system using Hamming codes can only contain 11 data bits in a 16-bit word. This would considerably limit the system, so the designer may elect to change the bus width to 24 bits so a full 16 bits of data would be available.

As an example of a Hamming code, imagine this 11-bit binary number needed to be transmitted: 011 0110 1001 in a system using even parity. Keep in mind that the first position in this binary number is considered "d1" even though it is the most significant bit of the number. This number could be placed in the data bit positions of the Hamming table:

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Purpose	P1	P2	d1	P4	d2	d3	d4	P8	d5	d6	d7	d8	d9	d10	d11	P16
		0		1	1	0		1	1	0	1	0	0	1		

Bit one, "P1", is designed to generate even parity for the data bits in positions 3, 5, 7, 9, 11, 13, and 15. The following table shows the bits that would be considered for P1 (under the cells with an "X"), and it becomes clear that bit P1 must be 1 for even parity of the other bits.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Purpose	P1	P2	d1	P4	d2	d3	d4	P8	d5	d6	d7	d8	d9	d10	d11	P16
P1 Cover	X		X		X		X		X	X	X	X	X		X	
	1	0	1	1	0		1	1	0	1	0	0	1			

Bit two, "P2", is designed to generate even parity for the data bits in positions 3, 6, 7, 10, 11, 14, and 15. The following table shows the bits that would be considered for P2 (under the cells with an "X"), and it becomes clear that bit P2 must be 1 for even parity of the other bits.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Purpose	P1	P2	d1	P4	d2	d3	d4	P8	d5	d6	d7	d8	d9	d10	d11	P16
P2 Cover		X	X			X	X			X	X			X	X	
	1	1	0		1	1	0		1	1	0	1	0	0	1	

Bit four, "P4", is designed to generate even parity for the data bits in positions 5, 6, 7, 12, 13, 14, and 15. The following table shows the bits that would be considered for P4 (under the cells with an "X"), and it becomes clear that bit P4 must be 0 for even parity of the other bits.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Purpose	P1	P2	d1	P4	d2	d3	d4	P8	d5	d6	d7	d8	d9	d10	d11	P16
P4 Cover				X	X	X	X				X	X	X	X	X	
	1	1	0	0	1	1	0		1	1	0	1	0	0	1	

Bit eight, "P8", is designed to generate even parity for the data bits in positions 9, 10, 11, 12, 13, 14, and 15. The following table shows the bits that would be considered for P8 (under the cells with an "X"), and it becomes clear that bit P8 must be 0 for even parity of the other bits.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Purpose	P1	P2	d1	P4	d2	d3	d4	P8	d5	d6	d7	d8	d9	d10	d11	P16
P8 Cover								X	X	X	X	X	X	X	X	
	1	1	0	0	1	1	0	0	1	1	0	1	0	0	0	1

Bit 16, "P16", would normally check parity on bits to its right. However, since it is the last bit, then it should be 0 (if it were 1 there would be odd parity for that one bit).

The final 16-bit word, including Hamming codes, is: **1100 1100 1101 0010**

Hamming codes can correct errors that have only 1 bit of distance (that is, only one bit is bad); but can detect errors that are 2 bits in distance. It is easy, though, to increase the detection to 3-bit distance. In the above example, bit 16 would normally be used as parity for bits 17-31; however, since this word is only 16 bits wide, then parity is not needed beyond bit 16. That means that bit 16 is unused. If it were changed to indicate simple even parity for the other 15 bits, then the system could reliably detect errors that are 3-bit distance.

When a group of bits is received, the parity bits can be easily stripped out and the other bits checked; in fact, Error Detection and Correction (EDIC) integrated circuits are readily available to do just that job. As an example, imagine that bit 14 (third bit in the last group of four) was changed in the word created above: **1100 1100 1101 0110**. With bit 14 bad, parity bits 2, 4, and 8 will be wrong, along with bit 16. By adding 2+4+8 (the Hamming bits), bit 14 is determined to be bad, and it can be changed.

If the error affects only one Hamming bit, then the error is in the parity bit itself and the data would be correct. As an example, consider another variation of the number created above: **1100 1101 1101 0010**. Bit 8, P8, has been changed from 0 to 1. Now, parity is bad for the P8 group, but no other group; thus, bit P8 must be bad and can be changed.

To use the 16<sup>th</sup> bit, imagine that two bits are changed in the original word, bits 3 and 5: **1110 0100 1101 0010**. Hamming bits 1, 2, and 4 will be wrong; but adding those numbers together would indicate that bit 7 was in error, which is not correct. Using the overall parity bit (bit 16) could help with this since it indicates whether there were an even or odd number of error bits. The following table will help interpret a number with both Hamming and overall parity bits:

<b>Hamming Bits</b>	<b>Overall Parity Bit</b>	<b>Interpretation</b>
Incorrect	Incorrect	1-Bit error occurred. Correct the bit indicated by the sum of the Hamming bit numbers.
Incorrect	Correct	2-Bit error occurred and it cannot be corrected.
Correct	Incorrect	Error in the overall parity bit.
Correct	Correct	No errors in word.

As a note of interest, if three bits are in error, the Hamming bits will indicate an error but the overall parity bit will be correct, which is the same as for a one-bit error; this is why the system is limited to detecting 2-bit errors. This system is called SECDED (“Single Error Correction, Double Error Detection”) and is commonly used in memory systems.

#### SAMPLE PROBLEMS

Here are a few 11-bit input numbers and the corresponding 16-bit output that includes Hamming parity and an overall parity in bit 16.

<b>11-Bit Input</b>	<b>16-Bit Output (With Hamming Parity)</b>
111 1101 0101	1111 1110 1010 1010
100 1011 0111	1010 0011 0110 1110
110 1001 0001	1011 1010 0010 0010
111 1011 0011	1111 1110 0110 0110
111 1110 0000	0011 1110 1100 0001
101 1110 1110	0111 0111 1101 1101
110 0001 1110	0110 1000 0011 1101
110 0011 1001	0011 1000 0111 0011

Here are a few 16-bit numbers that include Hamming parity. In each case one bit is wrong; find it.

<b>16-Bit Number (With Hamming Parity)</b>	<b>Incorrect Bit</b>
0011 1110 0001 1010	Bit 13
0110 0100 0001 1110	Bit 7
1011 1111 1010 0110	Bit 9
0110 0110 0011 0010	Bit 8
1110 0010 0001 1111	Bit 7
1010 1001 1011 0110	Bit 16

Finally, the following 16-bit numbers have two errors:

<b>16-Bit Number (With Hamming Parity)</b>	<b>Incorrect Bits</b>
1001 1101 1100 1110	Bits 4 and 5
0001 1111 1100 0111	Bits 4 and 14

## 5.11: LAB – 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

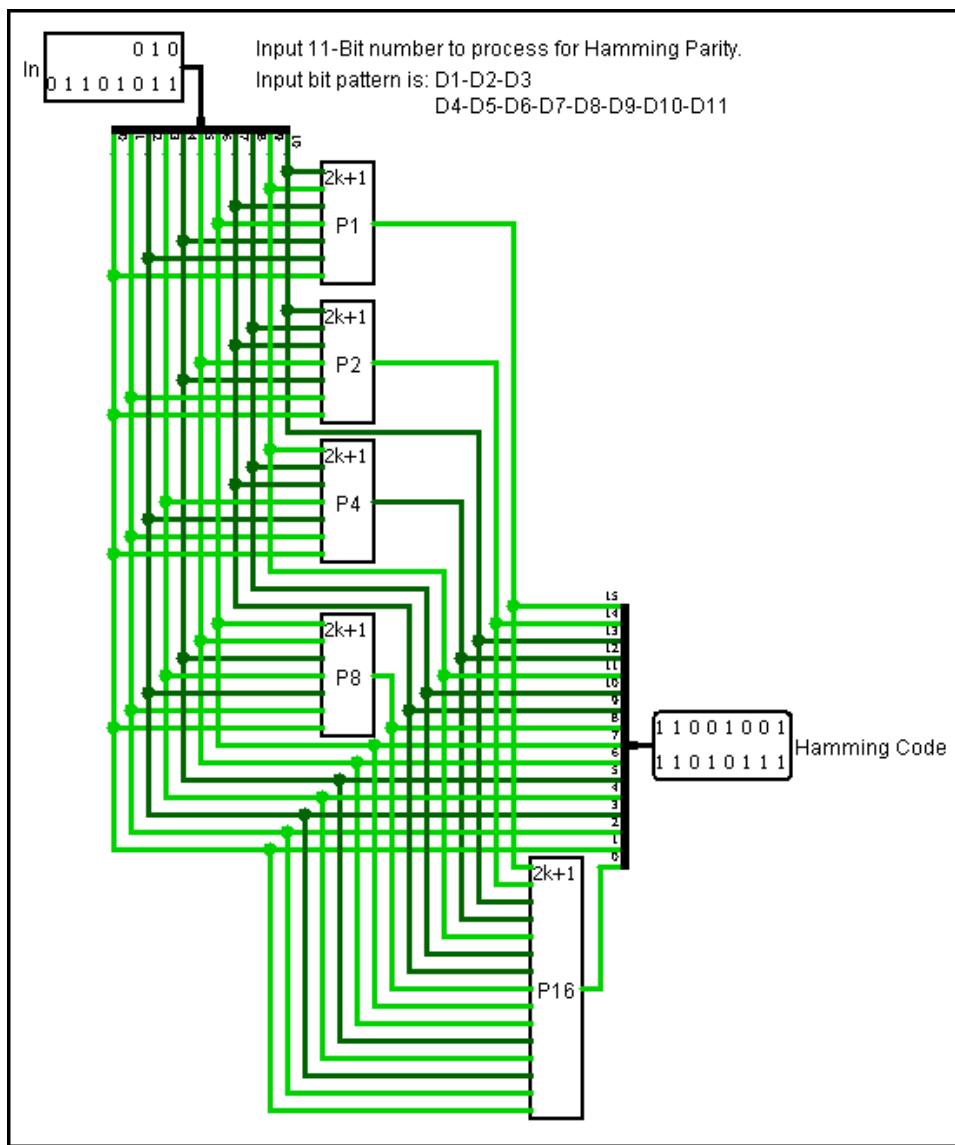


FIGURE 148: CREATE HAMMING PARITY CODE

For the lab, it would seem to be a good idea to feed the 16 input data bits into their proper parity checker, 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.

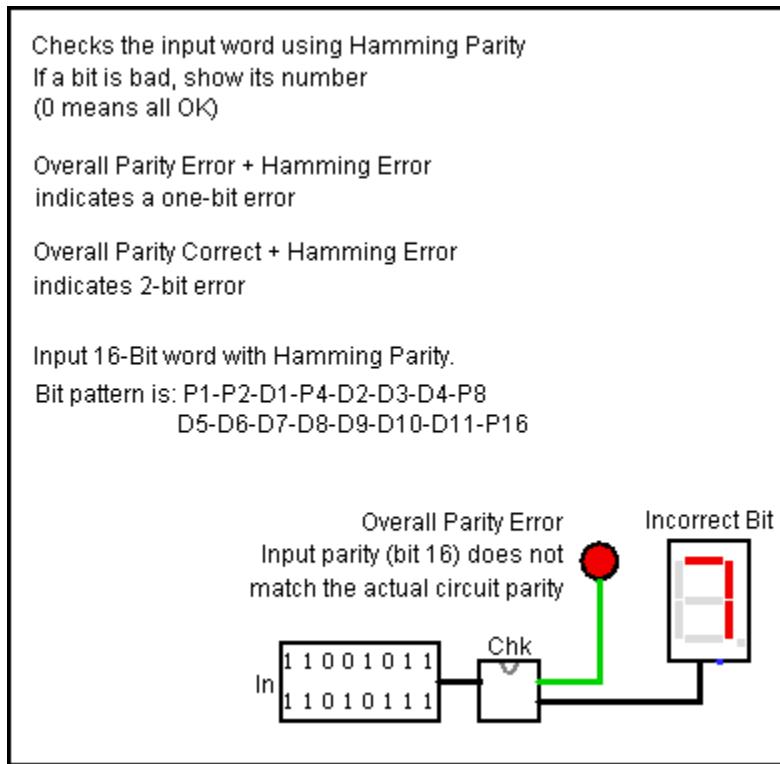


FIGURE 149: CHECKING HAMMING PARITY

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 5.11: Check Parity", and today's date. Save the file as *Lab 5\_11 – Hamming*.



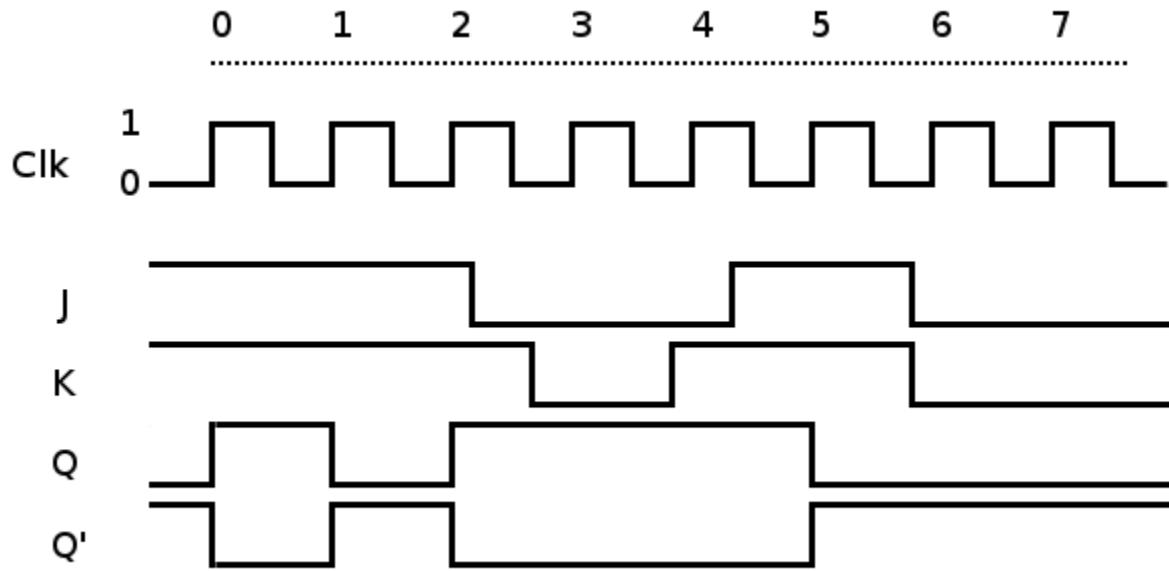
## 6: SEQUENTIAL LOGIC

### 6.1 INTRODUCTION

Electronic circuits that require memory devices (like flip-flops or registers) and feedback loops are designed using is called "Sequential Logic." The final output of the circuit is determined by both the inputs and the feedback; and the states of the various logic gates may change rapidly as the output from one stage is fed back to various input points. This makes sequential logic circuits much more dynamic, and complex, than combinational circuits, and their analysis normally requires tools like state maps. Because sequential circuits are able to latch and hold output states, useful applications like flip-flops, registers, counters, and memory are possible.

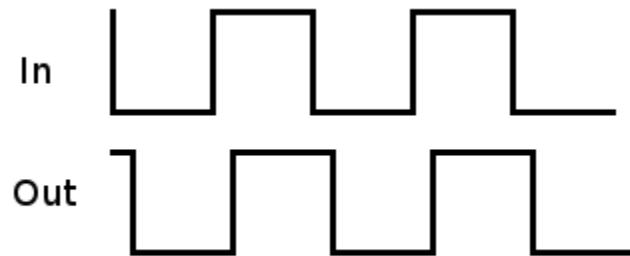
## 6.2 TIMING DIAGRAMS

Unlike combinational logic circuits, timing is essential in sequential circuits. Normally, a device will have a "clock" generating a square wave that is used to time various activities throughout the device.

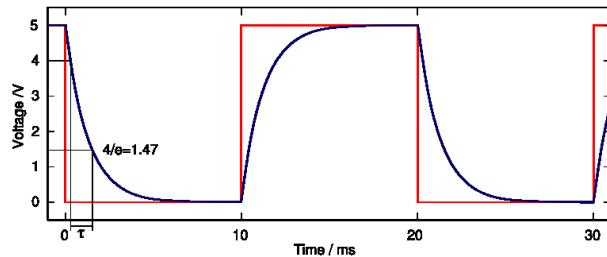


A square wave generated by a clock is seen at the top of the above diagram. At each "tick" of time, the clock changes from low to high. Timing for a JK Flip-flop is seen below the clock. Notice that when both J and K are high, and the clock changes from low to high, the output Q changes (if low, it goes high; or if high, it goes low). For example, at "tick 2" both J and K are high, so Q changes from low to high. Q' is always the opposite of Q; and the timing diagram makes that clear.

Whenever the input for any device changes; it takes a tiny, but measurable, amount of time for the output to change. This is known as "propagation delay." In the timing diagram below, notice that the "out" line follows the "in" line, but a tiny amount of time later.



It is also true that a square wave is not exactly square. Due to the presence of capacitors and inductors in circuits, a square wave will actually build up and decay over time rather than instantly change. The image below shows a typical charge/discharge cycle for a capacitor and the resulting deformation of a square wave<sup>3</sup>. It should be kept in mind, though, that the times involved for capacitance charge/discharge are quite small (measured in nanoseconds); so for all but the most sensitive of applications, square waves can be assumed to be truly square.



For the circuits presented in this course, ideal conditions are assumed; thus, all outputs react instantly and no devices have any propagation delay. This is done so it is possible to focus on the digital logic used rather than the engineering behind the devices.

---

<sup>3</sup> Image found at WikiMedia ([http://commons.wikimedia.org/wiki/File:Capacitor\\_Square\\_wave\\_charge-discharge.svg](http://commons.wikimedia.org/wiki/File:Capacitor_Square_wave_charge-discharge.svg)). The originator, “inductiveload,” released the image into the public domain.

### 6.3: FINITE STATE MACHINES

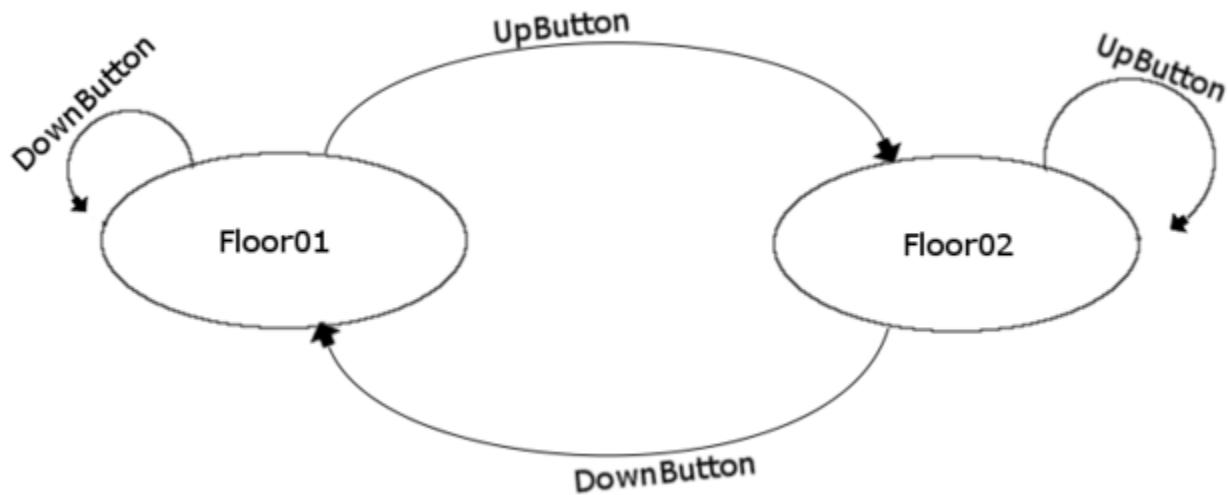
#### INTRODUCTION

Sequential logic circuits are dynamic and the combined inputs and outputs of the circuit at any given stable moment is called a "state." Over time, the circuit changes states as triggering events take place. As an example, a traffic signal may be green at some point, but then it would change to red based on a timer, a pedestrian pressing a "cross" button, or an automobile activating a surface sensor on a cross street. The current state of this system would be "green" and the triggering events are the timer, pedestrian cross button, and traffic surface sensor.

The mathematical model of a sequential circuit is often called a *Finite State Machine*, or simply *State Machine*. A Finite State Machine is an abstract model of a sequential circuit where each state of the circuit is indicated along with the triggering events that cause state transitions. The behavior of many devices can be represented by a Finite State Machine model, including traffic signals, elevators, vending machines, and robotic devices. Finite State Machines are an analysis tool that helps simplify sequential circuits.

#### EXAMPLE ONE: ELEVATOR

Imagine an elevator in a two-story building. That elevator has two states, Floor01 and Floor02, and the two possible triggering events are UpButton and DownButton for the two buttons. The following state diagram illustrates this elevator system.



On the state diagram, the states are represented by circles (or ovals). In the above diagram, "Floor01" and "Floor02" are the two states. When the elevator is in either of those two states (that is, it is on a floor awaiting a button press), the system is quiescent and awaiting the next triggering event. The Lines with arrows indicate the various triggers and what happens with each trigger. Notice that the "UpButton" trigger will change the state from "Floor01" to "Floor02," which indicates that someone pushed the Up button while the elevator was on Floor 1, so it moved to Floor 2. In the same way, the

"DownButton" trigger changes the elevator's state from "Floor02" to "Floor01." Also notice that if the "DownButton" event is triggered while the elevator state is "Floor01" then the state does not change (the arrow loops back to the same state). In the same way, the "UpButton" does not change the elevator's state if it is already on "Floor02"

A Finite State Machine can also be represented by a table that is similar to a truth table. For the elevator, assume that "Floor01" is "0" and "Floor02" is "1"; also, "DownButton" is "0" and "UpButton" is "1." This is the state table for the elevator:

Current State	Trigger	Next State
0	0	0
0	1	1
1	0	0
1	1	1

TABLE 91: ELEVATOR STATE TABLE

To read this table, if the Current State is 0 (that is "Floor01") and a trigger of 1 ("UpButton") occurs, then the elevator's next state is "1"; that is, it will move to Floor 2. This table makes it possible to determine the next state for every possible current state and trigger; and a logic circuit can be created to realize this Finite State Machine.

#### EXAMPLE TWO: PEDESTRIAN CROSSWALK

Imagine a pedestrian crosswalk in the middle of a block, similar to that illustrated on the right. That crosswalk has a traffic signal to stop traffic and a "cross/don't cross" light for pedestrians. It also has a button that pedestrians can press to make the light change so it is safe to cross.



This system has several states. The traffic signal can be Red, Yellow, or Green; and the pedestrian signal can be Walk or Don't Walk. Each of these signals can be either 0 (for "Off") or 1 (for "On"). Also, there are two triggers for the circuit; a push button (the "Cross" button a pedestrian presses) and a timer (so the red traffic light will eventually change back to green). If the button is assumed to be "1" when it is pressed but "0" when not pressed; and the timer is assumed to be "1" when a certain time interval has expired but "0" otherwise, then the following state table can be created:

Current State					Triggers		Next State				
R	Y	G	W	D	Btn	Tmr	R	Y	G	W	D
1	0	0	1	0	1	0	0	0	0	1	0
2	0	0	1	0	1	1	0	0	1	0	0
3	0	1	0	0	1	X	0	1	0	0	1
4	1	0	0	1	0	X	1	0	0	1	0

The various states are "R" (for "Red"), "Y" (for "Yellow"), and "G" (for "Green") traffic lights; and "W" (for "Walk") and "D" (for "Don't Walk") pedestrian lights. The "Btn" (for "Button") and "Tmr" (for "Timer") signals can, potentially, change the state of the system.

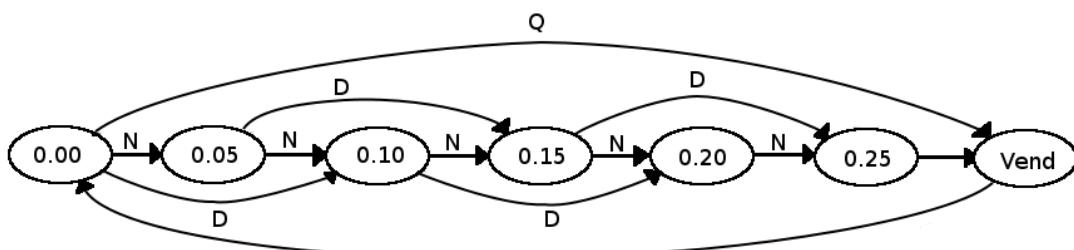
Row one on this table shows that the traffic light is Green and the pedestrian light is Don't Walk. If the button is not pressed (it is 0), then the next state is still a Green traffic light and Don't Walk pedestrian light; in other words, the system is static. In row 2, the button was pressed ("Btn" is 1); notice that the traffic light changes state to Yellow, but the pedestrian light is still Don't Walk. In row 3, the current state is a Yellow traffic light with Don't Walk pedestrian light (in other words, the "Next State" from row 2), the "X" for the button means it doesn't matter if it is pressed or not, and the next state is a Red traffic light and Walk pedestrian light. In row 4, the timer expires (it changes to "1" to signal the end of the timing cycle), and the traffic light changes back to Green while the pedestrian light changes to Don't Walk.

This table would be extended to cover all possible combinations of input, but it must be kept in mind that Red, Yellow, and Green can never all be "1" at one time, nor could Walk and Don't Walk. Therefore, the designer must intentionally define the inputs rather than use a simple binary count from 00000 to 11111. Also, there are certain combinations that should never happen, like a Green traffic light and a Green pedestrian light at the same time; so those would have to be carefully avoided.

A designer could create a circuit that would meet all of the requirements from the state table, and then realize that circuit to actually build a traffic light system.

#### EXAMPLE THREE: VENDING MACHINE

Imagine a vending machine that accepts nickels, dimes, and quarters and vends a product when 25 cents has been deposited. To keep this example simple, assume that the machine does not return change.

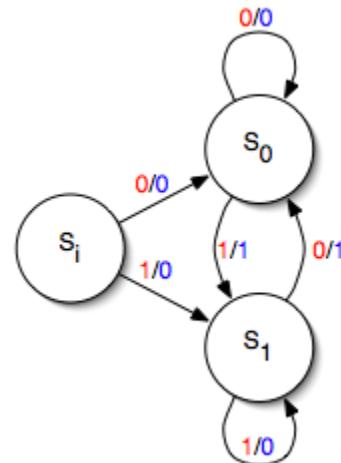


The above drawing shows the various states for the coin acceptor of the vending machine. Starting at the left, the machine holds \$0.00. If a nickel ("N") is deposited, the state changes to \$0.05; if a dime ("D") is deposited the state changes to \$0.10; and if a quarter is deposited ("Q"), the state changes to \$0.25. Each new coin deposited moves the state to the right by some amount. When \$0.25 is available, the state changes to "Vend" and a product is dispensed; then the state changes back to \$0.00 and a new cycle is ready to start.

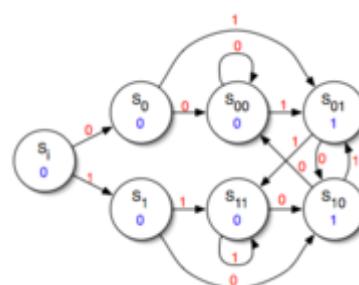
### MEALY AND MOORE MACHINES

There are two fundamental state machine models: Mealy and Moore. These two models are generalizations of a state machine and differ only in the way the outputs are generated. A Mealy machine generates an output as a function of the current state and the inputs while a Moore machine generates an output as a function of only the current state.

The Mealy machine is named after George H. Mealy, who presented the concept in a 1955 paper, "A Method for Synthesizing Sequential Circuits." The Mealy machine permits only one possible transition for any given combination of inputs and states and a designer may choose to use a Mealy machine to reduce the possible number of states for the system. The diagram on the right shows a Mealy machine. Each of the transitions shows the value of the input and the value of the output. Thus, moving from state  $S_i$  (the initial state) to  $S_0$  an input of 0 to the machine generates an output of 0 and moves the machine to state  $S_0$ . In the same way, an input of 1 generates an output of 0 and moves the machine to state  $S_1$ . This diagram illustrates an exclusive-or of the two most-recent input values; thus, the machine implements an edge detector, outputting a one every time the input flips and a zero otherwise.



The Moore machine is named after Edward F. Moore, who presented the concept in a 1956 paper, "Gedanken-experiments on Sequential Machines." The Moore machine uses only entry actions, so its output depends only on the state rather than the inputs. The Moore machine is typically simpler than a Mealy machine and modeling hardware systems is normally best done using a Moore model. The diagram on the right is a Moore machine that shows an edge detector using an XOR gate. The numbers between the states are inputs and the numbers within each state are the outputs.



## 6.4: FLIP-FLOPS

### INTRODUCTION

Flip-flops are digital circuits that can maintain an electronic state even after the initial signal is removed. They are, then, the simplest of memory devices. Flip-flops are often called latches since they are able to "latch" some electronic state. In general, devices are called flip-flops when a clock pulse is needed to set the output state and they are called latches when the output is constantly reacting to the input.

Commonly, flip-flops are used for clocked devices, like counters, while latches are used for storage.

### SR LATCH

One of the simplest of the flip-flops is the SR (for "Set-Reset") latch. (Note, this latch is often also called an RS latch.) Following is a logic diagram for one potential configuration of an SR latch using NAND gates:

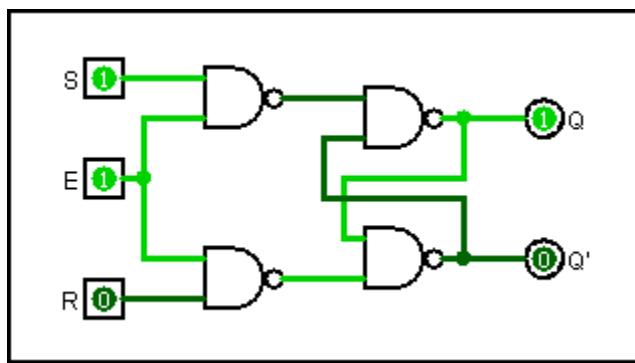


FIGURE 150: SR LATCH

SR Truth Table				
E	S	R	Q	Q'
0	0	0	Last Q	Last Q'
0	0	1	Last Q	Last Q'
0	1	0	Last Q	Last Q'
0	1	1	Last Q	Last Q'
1	0	0	Last Q	Last Q'
1	0	1	0	1
1	1	0	1	0
1	1	1	No Allowed	Not Allowed

In the logic diagram, input "E" is an enable and must be high for the latch to work; when it is low then the output state remains constant regardless of how inputs S or R change. When the latch is enabled, if  $S=R=0$  then the values of Q and Q' remain fixed at their last value; or, the circuit is "remembering" how they were previously set. When the input to S goes high (as illustrated), then Q goes high (the latch's output, Q, is "set"). When the input to R goes high, then Q' goes high (the latch's output, Q', is "reset"). Finally, it is important that in this latch inputs R and S cannot both be high at the same time or the circuit becomes unstable; thus, input 11 must be avoided. Normally, there is an additional bit of circuit before that shown above to ensure S and R will always be different (an XOR gate could do that job).

### DATA (D) LATCH

A Data Latch (or D Latch) is formed when the inputs for an SR latch are tied together through an inverter. This is the diagram for this type of circuit built with an SR latch using the Logisim system:

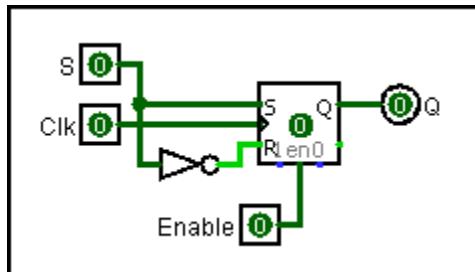


FIGURE 151: DATA LATCH USING SR LATCH

However, a data latch is available in Logisim (see below). If the input, "Data," is high, then Q goes high on the next clock pulse. If Data is low, then Q goes low on the next clock pulse. In other words, Q will always mirror the value of Data when the clock pulses. Notice that there is an enable pin to turn this latch on and off and there are also "1" and "0" pins to preset the output as desired.

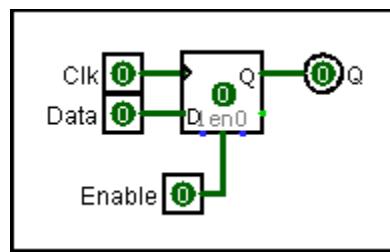


FIGURE 152: DATA LATCH

### JK FLIP-FLOP

The simplest flip-flop is the JK flip-flop, which is the "workhorse" of the flip-flop family. The JK Flip-Flop looks like this in a circuit:

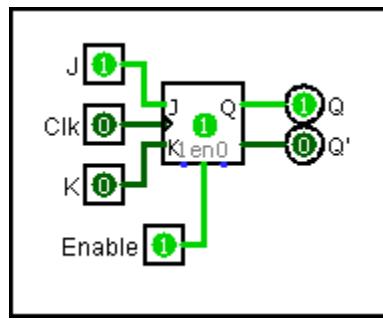
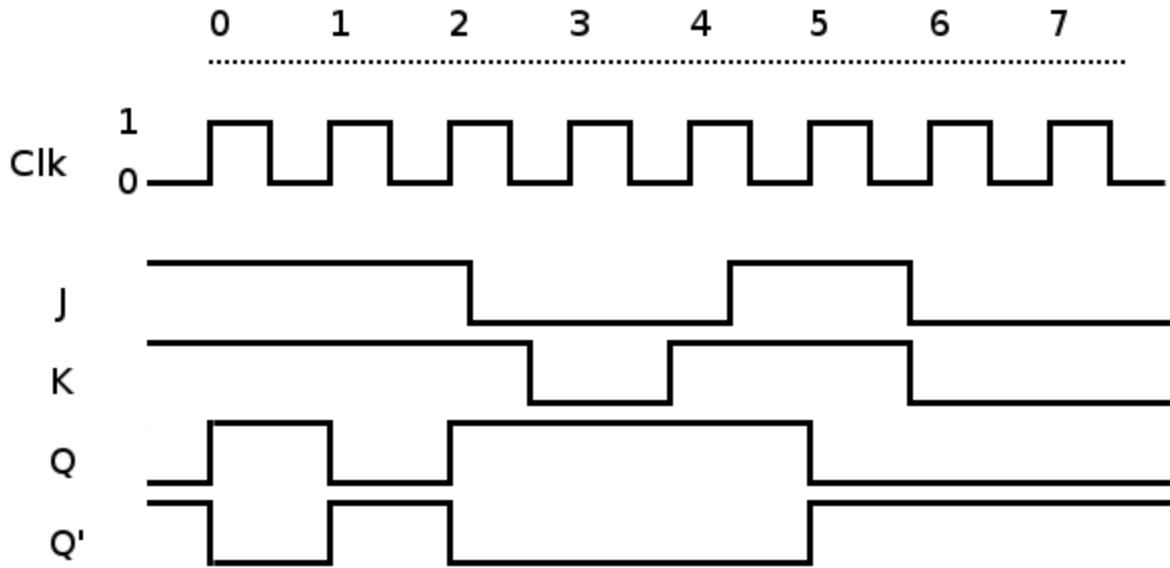


FIGURE 153: JK FLIP-FLOP

Internally, a JK flip-flop is similar to an RS latch. However, the outputs to the circuit (Q and Q') are connected to the inputs (J and K) in such a way that the unstable input condition of the RS latch is corrected. If both inputs, J and K, are high, then the outputs are simply reversed (they are "flip-flopped") on the next clock pulse. This toggle feature makes the JK Flip-Flop extremely useful in many logic circuits; some of which are illustrated elsewhere in this course.



The above timing diagram was used in the introduction to this unit. A JK Flip-flop is designed to activate not on a high or low level on the clock; but on a change in level, which is called "edge triggered."

The timing diagram is read from left to right. Notice that when both J and K are high, then Q changes on every positive-going clock pulse (ticks 0, 1, 2, and 5); otherwise, Q does not change. Also, note that Q and Q' are mirror images of each other.

#### TOGGLE (T) FLIP-FLOP

If the J and K inputs to a JK flip-flop are tied together, then the output will toggle on every clock pulse. This is often referred to as a Toggle Flip-Flop (or T-Type Flip-Flop). T-Type Flip-Flops are not usually found as separate ICs since they are so easily created using a JK Flip-Flop.

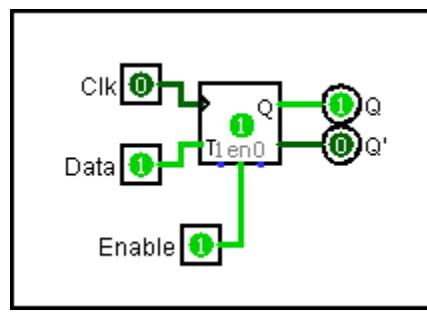


FIGURE 154: T FLIP-FLOP

In a T Flip-Flop, if data is present on the T input (that is, it is high), then the Q output toggles on every clock pulse. This is why this type of flip-flop is called a "T" type, for "Toggle."

**MASTER-SLAVE FLIP-FLOPS**

Master-Slave Flip-Flops are nothing more than two flip-flops connected in a cascade and operating from the same clock pulse. These flip-flops tend to stabilize an input circuit and are used where the inputs may have voltage glitches (such as from a push button). Following is the logic diagram for two JK Flip-Flops set up as a Master-Slave Flip-Flop:

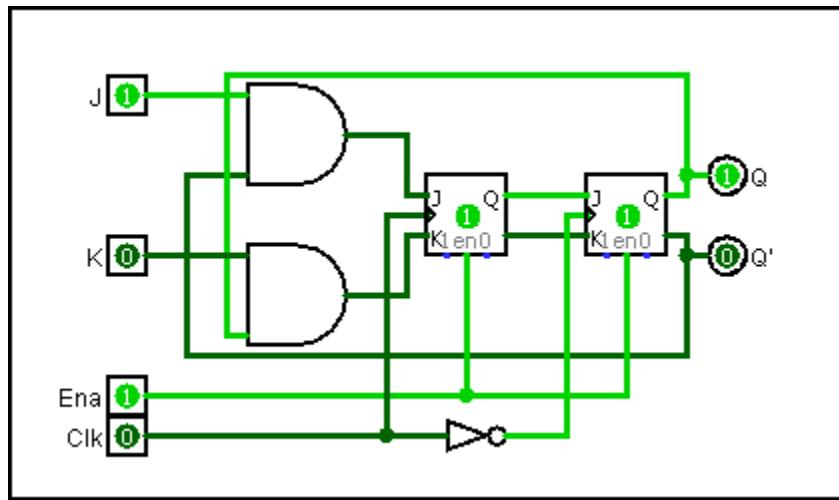


FIGURE 155: TWO JK FLIP-FLOPS IN MASTER-SLAVE CONFIGURATION

Often, only one IC is needed to provide the two flip-flops for a circuit because two JK flip-flops are frequently found on a single IC. Thus, the output pins for one flip-flop would be connected directly to the input pins for the second flip-flop. By combining two flip-flops into a single IC package, circuit design can be simplified and fewer components need to be purchased.

## 6.5: REGISTERS

### INTRODUCTION

A register is a simple memory device that is composed of a series of flip-flops wired together so that they share a common clock pulse. Registers come in various sizes and types and are often used as "scratch pads" for devices. For example, a register can hold a number entered on a keypad until the calculator is ready do something with that number or a register can hold a byte of data coming from a hard drive until the CPU is ready to move that data to memory.

### REGISTERS AS MEMORY

Internally, a register is constructed from D flip-flops like this:

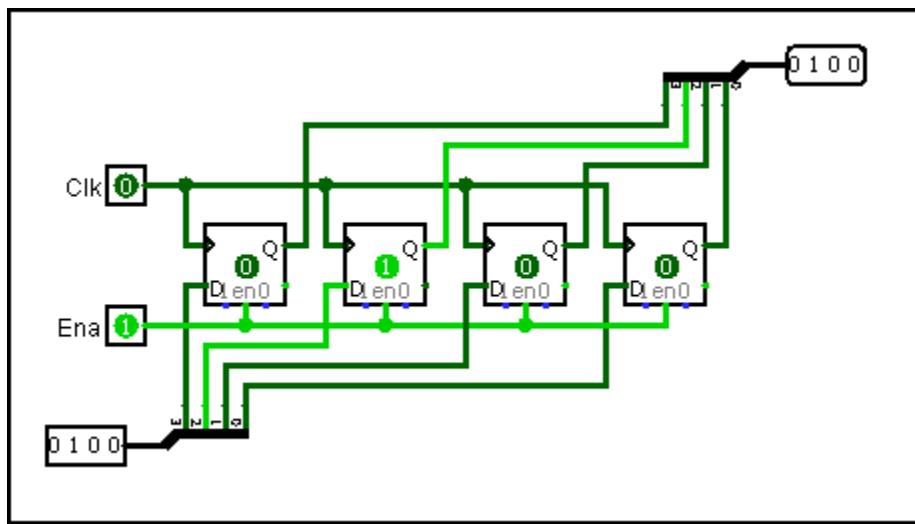


FIGURE 156: 4-BIT REGISTER BUILT FROM FLIP-FLOPS

On each clock pulse, data are moved from the input port, in the lower left corner, into the register; one bit goes into each of the four flip-flops. Because each flip-flop constantly outputs whatever it contains, the output port, in the top right corner, displays the contents of the register. By using the enable/disable feature, a register can load data by turning enable on for one clock pulse, and then continue to output its contents regardless of what happens to the clock or the inputs if the enable is turned off. Thus, this 4-bit register can "remember" some number until it is enabled a second time and something new loaded; it is a memory device.

### 74x273 8-BIT REGISTER

The 74x273 8-bit register loads and stores a single 8-bit number:

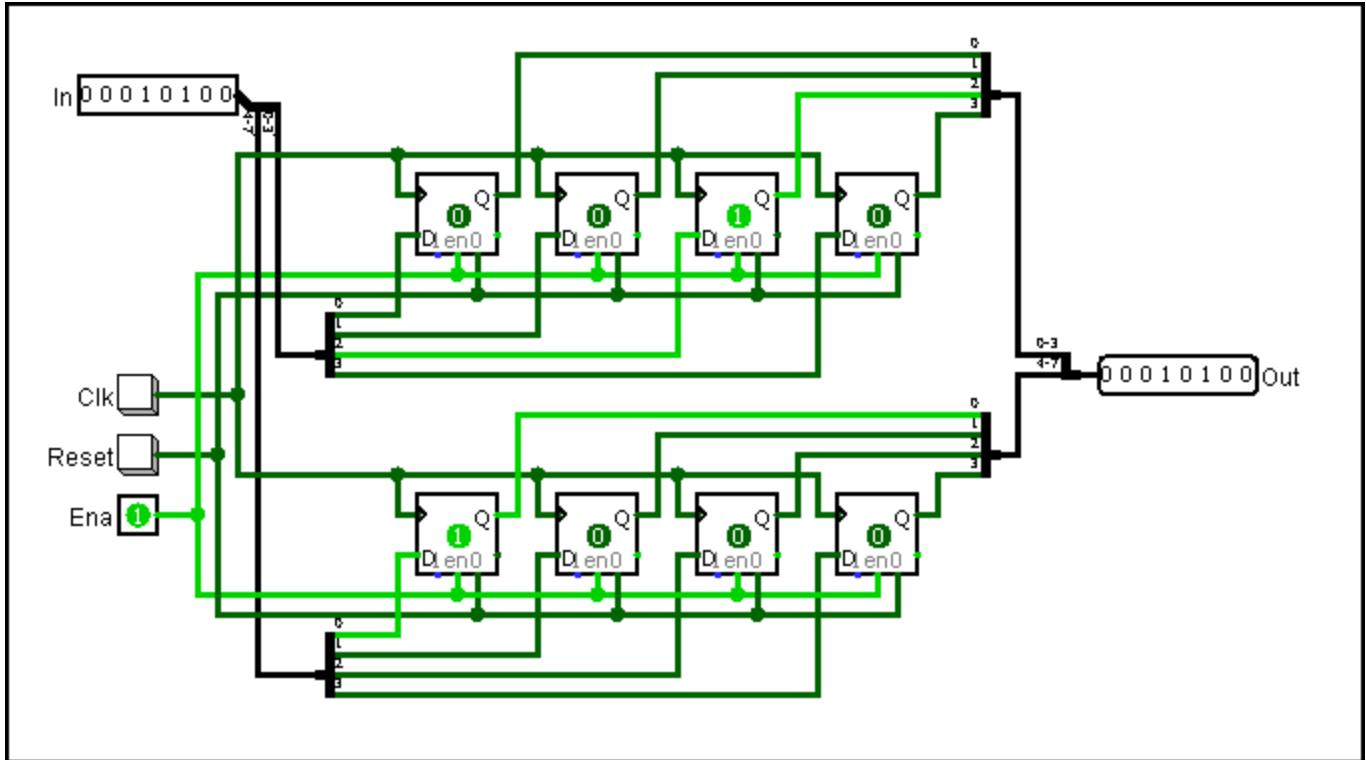


FIGURE 157: 74X278 8-BIT REGISTER

The internal logic of this IC is similar to the 4-bit register shown above. The IC is enabled and then the clock is pulsed to load one bit of the number on the input port into each of 8 D flip-flops. That number is immediately seen at the output port. If the enable is made low, no further changes at the input will be loaded into the register and it will continue to output its current contents (00010100 in the above example). The output of this register will not change again until after it is enabled and a new value loaded on a clock pulse.

#### LOGISIM REGISTER

Logisim includes a register device that can be set up to contain an arbitrary number of bits. The following illustration shows an 8-bit register:

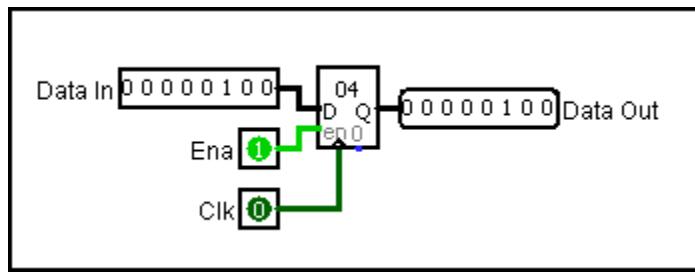


FIGURE 158: LOGISIM 8-BIT REGISTER

In this circuit, the register has an 8-bit input. If the register is enabled, the next clock pulse will load whatever is in the input port into the register. The register can be set to load on a rising or falling edge or a high or low level. For convenience, the register also displays its current contents as a hexadecimal

number since a register will normally be embedded deep in a circuit and its contents may not be easy to otherwise determine.

### SHIFT REGISTERS

Registers have an important function in changing a stream of data from serial to parallel or parallel to serial; a function that is called "shifting" data. For example, data entering a computer from a network or USB port is in serial form; that is, one bit at a time is streamed into or out of the computer. However, data inside the computer are always moved in parallel; that is, all of the bits are placed on a bus and moved through the system simultaneously. Changing data from serial to parallel or vice-verse is an essential function and enables a computer to communicate over a serial device.

74x299 UNIVERSAL SHIFT REGISTER

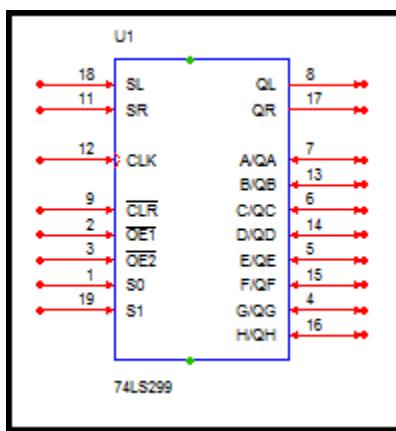


FIGURE 159: 74X299 UNIVERSAL SHIFT REGISTER PINOUT

The above illustration is the pinout diagram for a 74x299 Universal Shift Register (this particular version is the 74LS299). This device is able to take data in, hold it, and then shift it out. Moreover, the device is able to work with data in either serial or parallel form, that is, serial in/serial out, serial in/parallel out, parallel in/serial out, or parallel in/parallel out; thus, it is a universal shift register. In this register, pins 1 and 19 (S0 and S1) are the Serial Mode pins and determine whether the data in the register will shift left or right or hold. This IC has a bidirectional parallel port that can both input and output a single byte at a time; and Pins 2 and 3 (OE1 and OE2) determine that port's behavior.

Pin 11 (SR) accepts serial data for a right-shift while pin 18 (SL) accepts serial data for left shift. Pins 8 (QL) and 17 (QR) are the serial data outputs. Pins 7, 13, 6, 14, 5, 15, 4, 16 are the parallel data port and its function is determined by the setting on pins 2 and 3. Pin 12 is the clock and the other pins are used for enabling and resetting the IC.

In practice, a serial signal (perhaps from a USB device) would appear at pin 11, and then the IC would be put into a "Right Shift" mode. That serial stream would be accepted one bit at a time on subsequent clock pulses, and shifted right (most-to-least significant bit) onto the parallel pins. After eight clock pulses, the IC's mode would change to "hold" (using pins 1 and 19), the parallel port would be changed to output (using pins 2 and 3), and the byte would then be made available on the parallel output pins.

When a byte needs to be sent from the computer's bus to the USB device, it would appear on the parallel pins, shifted into the IC on one clock pulse, and then shifted to pin 8 one bit at a time.

#### 74x299 UNIVERSAL SHIFT REGISTER IN LOGISIM

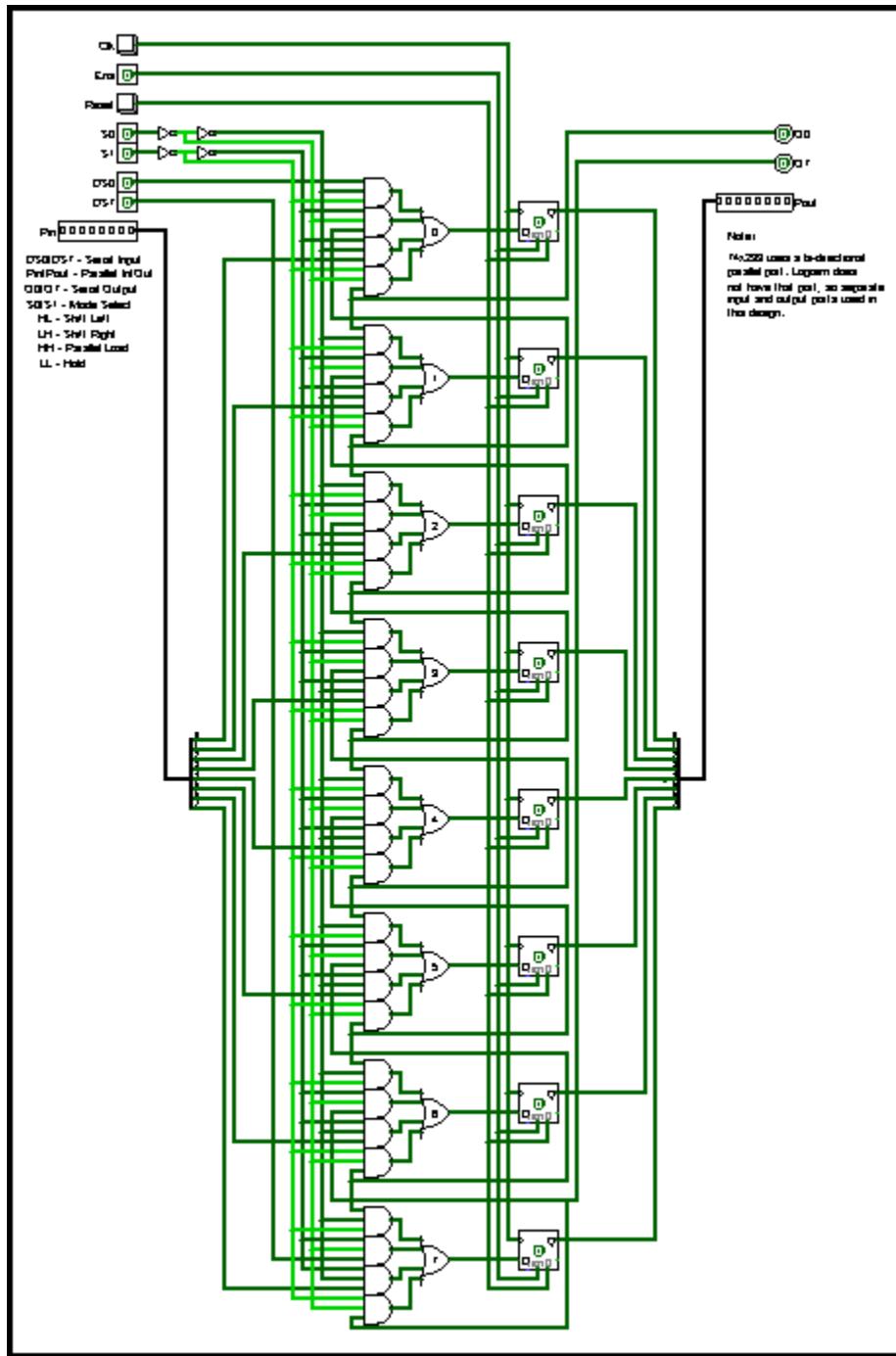


FIGURE 160: 74X299 UNIVERSAL SHIFT REGISTER IN LOGISIM

Logisim does not have a built-in 74x299 circuit; however it is not difficult to create that circuit from the logic diagram included in the 74x299 spec sheet. The illustration above is the logic diagram for a 74x299

circuit built in Logisim (it is a rather large circuit so the scaled down image is somewhat fuzzy). There are three important differences in the Logisim and true versions of this IC:

1. Logisim does not have a bidirectional port available, so the parallel input and output ports are different.
2. Because there is no bidirectional parallel port, controls OE1 and OE2 are unnecessary, and have been eliminated from the circuit.
3. The 74x299 IC uses a negative logic level for the "CLR" function, but the Logisim version was built with positive logic (though a simple inverter could be used to change that if desired).

## 6.6: LAB – GUESSING GAME

### PURPOSE

A register is one of the smallest units of memory. While a simple flip-flop can store a single bit, a register stores an entire byte of information and is very useful in many applications. Internally, then, an 8-bit register is little more than eight 1-bit flip-flops working in parallel to provide storage for an 8-bit byte.

In this lab, two registers will be used to store numeric inputs that are used in a game.

### PROCEDURE

A certain number-guessing game generates a random number and then a player then attempts to guess that number. After each guess, the game will display a message to let the player know whether the guess was correct, too high, or too low. In this lab, only 4-bit numbers are used, so all guesses must be between 0 and F (hexadecimal).

Create the following circuit:

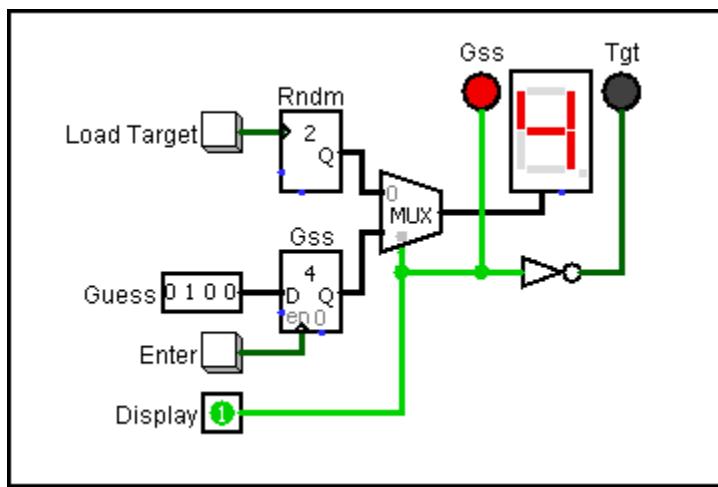


FIGURE 161: GUESSING GAME

The *Load Target* button in the top left corner of the circuit is connected to a 4-bit random number generator named *Rndm* (that component is found in the *Memory* library). A random number generator is a specialized form of register that will output a random number every time a clock pulse is input. That random number (which is the "target") is fed to input 0 of a multiplexor.

Below the target portion of circuit is the 4-bit *Guess* input port. It is connected to a 4-bit D register where the user's guess is held when the "Enter" button is clicked. The output of the *Gss* register is fed to input 1 of a multiplexor.

The multiplexor is controlled by the 1-bit *Display* input port. When that port is set to 1, the hex display will show the contents of the Guess register and the *Gss* LED will be active. When the *Display* port is at 0, the hex display will show the contents of the random number generator and the *Tgt* LED will be active.

**CHALLENGE**

This guessing game is not complete without a circuit to compare the *guess* with the *target* and let the player know if the guess is too high, too low, or correct. There is a comparator available in the Arithmetic library. Place a comparator in the circuit and wire the target and guess numbers to it. Use the output of the comparator to turn on one of three LEDs: "Too High," "Too Low," or "Correct." If desired, the LED colors can be modified to better emphasize the guess outcome.

Also, remove the *Enter* button connected to the *Gss* register and replace it with the clock from the base library. While about any frequency would work, 16 or 32 HZ seems to work best for this circuit.

**CLEANUP**

Rename the *main* circuit to *Guess*. Be sure the standard identifying information block is at the top left of the *Guess* circuit: Name, "Lab 6.6: Guessing Game", and today's date. Save the file as *Lab 6\_6 – Guessing Game*.

## 6.7: COUNTERS

### INTRODUCTION

Counters are a common sequential circuit. Counters are designed to count input pulses (normally from a clock), and then activate some output when a specific count is reached. Counters are commonly used as timers; thus, all digital clocks, microwave ovens, and other digital timing displays use some sort of counting circuit. However, counters can be used in many diverse applications. For example, a speed gauge is a counter. By attaching some sort of sensor to a rotating shaft, the count of the number of revolutions for 60 seconds gives the RPM for the shaft. Counters are also commonly used as frequency dividers. If a high frequency is applied to the input of a counter, but only the "tens" count is output, then the input frequency will be divided by ten. Counters can also be used to control sequential circuits or processes; each count can activate or deactivate some part of a circuit that controls a sequential process.

### ASYNCHRONOUS COUNTERS

One of the simplest counters possible is an asynchronous 2-bit counter. This can be built with two T Flip-Flops in sequence, as shown below:

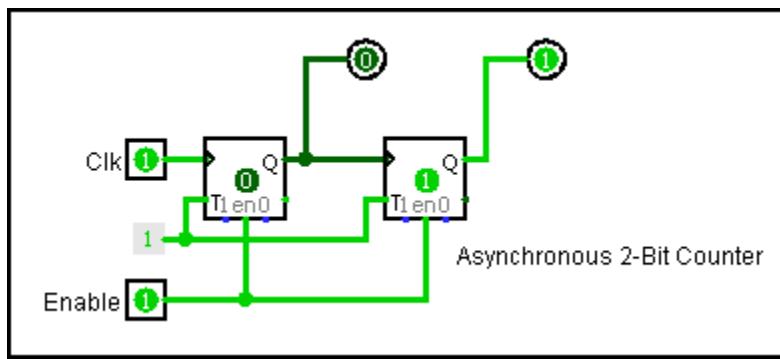


FIGURE 162: ASYNCHRONOUS 2-BIT COUNTER

Since the T inputs on both flip-flops are tied high, every time a positive-going pulse hits the clock input, the flip-flop's outputs will change. Q0 (the output of the first flip-flop) is wired to the clock input of the second flip-flop, triggering it to change; this is why this type of circuit is frequently called a "ripple" counter since the input clock pulse must ripple through all of the flip-flops.

In the above circuit, assume both flip-flops start with the output low (or "00" out). This can be set by applying a high input on the CLR pins (labeled "0" on at the lower right corner of the flip-flop, it is not connected in the illustration). On the first clock pulse, Q0 will go high, and that will send a high signal to the clock input of the second flip-flop, which also activates the output (Q1) on that device since the flip-flop changes on any change from low to high on the clock input. At this point, both Q0 and Q1 are high (or "11" out). On the next clock pulse, Q0 will go low, and that will send a negative change to the clock input on the second flip-flop, but that device will not change since it only flip-flops when the clock goes from low to high. At this point, the output is 01. On the next clock pulse, Q0 will go high and Q1 will go low (or "10"). The next clock pulse changes Q0 to low and Q1 does not change (or "00"). Then the cycle repeats. This simple circuit counts 00, 11, 10, 01. (Note, output Q0 is the low-order bit.) This counter is

counting backwards, but it is a trivial exercise to add the function needed to reverse that count. The function of this circuit, as all digital logic circuits, is best understood when the circuit is created on a simulator so the various inputs can be manipulated and the outputs observed.

An asynchronous 3-bit counter looks much like the asynchronous 2-bit counter, except a third stage is added:

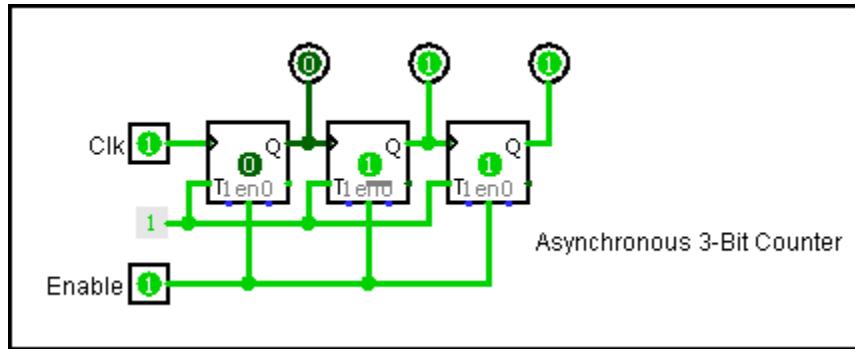


FIGURE 163: ASYNCHRONOUS 3-BIT COUNTER

In the above diagram, the ripple of the clock pulse from one flip-flop to the next is more evident than in the 2-bit counter. However, the overall operation of this counter is very similar to the 2-bit counter.

More stages can be added so the number of outputs can be greater than three. Asynchronous (or ripple) counters are very easy to build and require very few parts. Unfortunately, they suffer from two rather important flaws: propagation delay and glitches.

As the clock pulse ripples through the various flip-flops, it is slightly delayed by each due to the simple physical switching of the circuitry within the flip-flop. Propagation delay cannot be prevented and as the number of stages increases the delay becomes more pronounced. At some point, one clock pulse will still be winding its way through all of the flip-flops when the next clock pulse hits; and this makes the counter unstable.

The other issue is glitches. If a 3-bit counter is needed (as illustrated above), there will be a very brief moment while the clock pulse ripples through the flip-flops that the output will be wrong. For example, the circuit should go from 111 to 000, but it will actually go from 111 to 110 then 100 then 000 as the "low" ripples through the flip-flops. These glitches are very short, but they may be enough to introduce errors into a circuit.

#### SYNCHRONOUS COUNTERS

Both problems with ripple counters can be corrected with a synchronous counter, where the same clock pulse is applied to every flip-flop at one time. Here is the logic diagram for a synchronous 2-bit counter using JK Flip-Flops rather than T Flip-Flops:

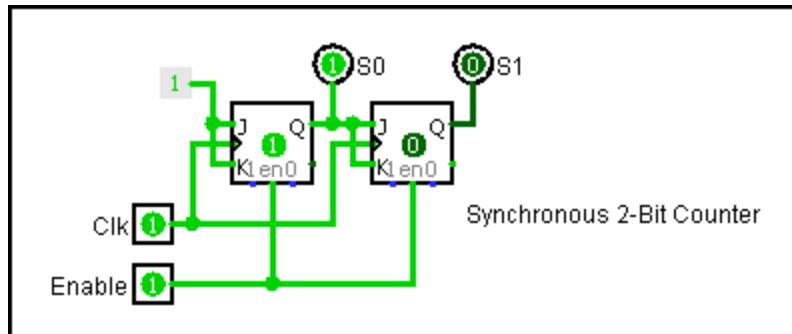


FIGURE 164: SYNCHRONOUS 2-BIT COUNTER

Notice in this circuit, the clock is applied to both flip-flops and control is exercised by applying the output of one stage to both J and K inputs of the next stage, which effectively enables/disables that stage. When the output of one stage is high, for example, then the next clock pulse will make the next stage change states.

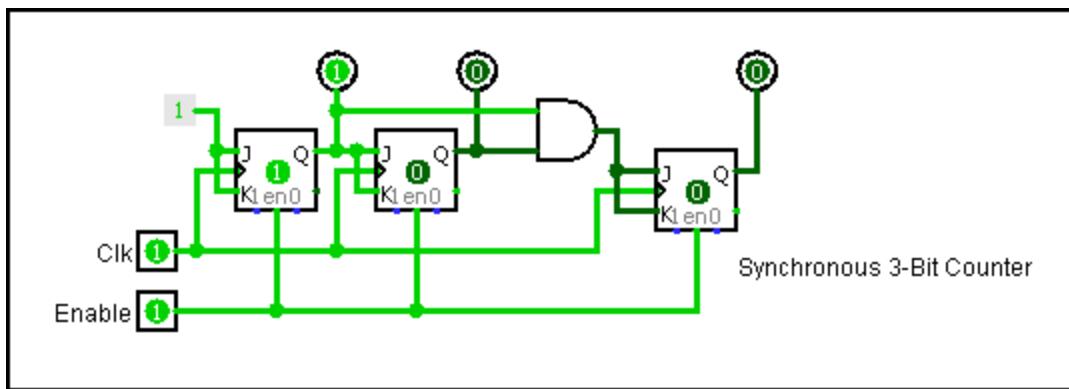


FIGURE 165: SYNCHRONOUS 3-BIT COUNTER

A 3-bit synchronous counter applies the clock pulse to each flip-flop. Notice, though, that the output from the first two stages must be ANDed so the last stage will only change when both of those earlier stages are high. This circuit would then count properly from 000 to 111.

In general, synchronous counters become more complex as the number of stages increases since it must include logic for every stage to determine when that stage should activate. This complexity results in greater power usage (every additional gate requires power) and heat generation; however, synchronous counters do not have the problems found in asynchronous counters.

The Logisim simulator includes a built-in counter device that is very easy to use in a circuit. The following illustration shows a counter with a high enable and a clock. As the clock changes from low to high and back to low (simulating a square wave), the counter will count. Notice that the output of the counter, 101, is also indicated on the counter's face. This is a handy way for a designer to know the contents of a counter at a glance. Logisim counters have a number of properties that need to be considered, such as whether the counter will increase on a low-to-high clock or some other part of the square wave input.

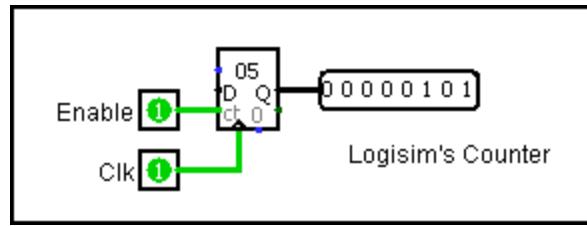


FIGURE 166: LOGISIM COUNTER

### RING COUNTERS

A ring counter is a special kind of counter where only one output is active at a time. As an example, a 4-bit ring counter may output this type of pattern:

1000 – 0100 – 0010 – 0001

Notice the high output cycles through each of the positions and then recycles. Ring counters are useful as controllers for processes where one process must follow another in a sequential manner. Each of the outputs from the ring counter can be used to activate a different part of the overall process; thus ensuring the process runs in proper sequence.

A ring counter is easily made from a series of D Flip-Flops, as in the following illustration.

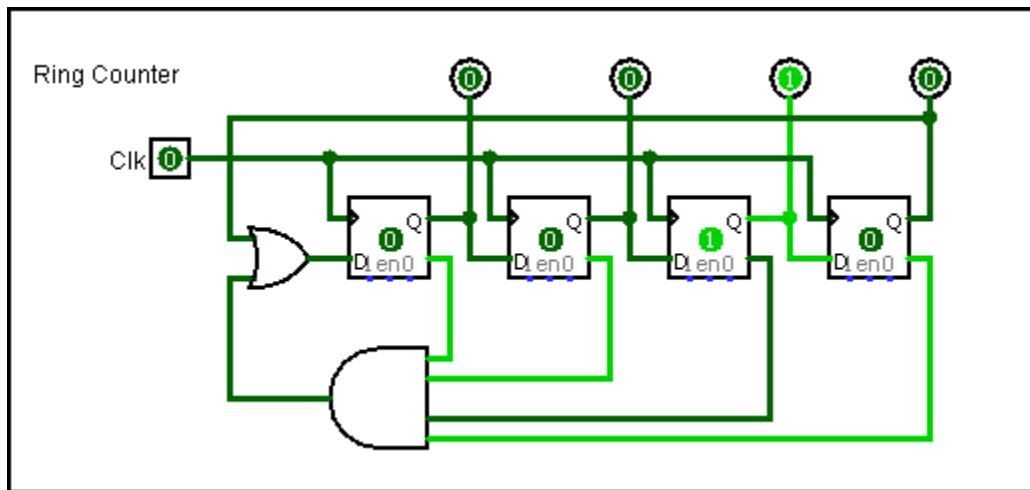


FIGURE 167: 4-BIT RING COUNTER

As this counter operates (a clock pulse is present), each of the four outputs will become active for exactly one clock pulse, and then the next will become active.

A ring counter must be seeded with a starting value or it may start with more than one flip-flop high, which would foul the count. In the circuit above, when the flip-flops are initially cleared on power-up, Q' will be active on all of them. That activates the 4-input AND gate, which then feeds a High into the first flip-flop. As soon as that flip-flop activates on the first clock pulse, Q' will go low on the first flip-flop and then as long as the circuit functions at least one Q' will be low, which means the only time a new high appears on the input of the first flip-flop is when Q on the last flip-flop is active. The 4-input AND gate is

an initialization gate that is only needed when the circuit is first activated to get the count started. As in all sequential circuits, this is most easily envisioned by building the circuit and activating the clock pulse.

It is possible to also build a ring counter by using a decoder. In the following circuit, a simple counter is used to control the output of the decoder, so only one decoder output is active at a time. The controlling counter can be set to loop after any given count, so the decoder can active any number of lines before cycling back to zero.

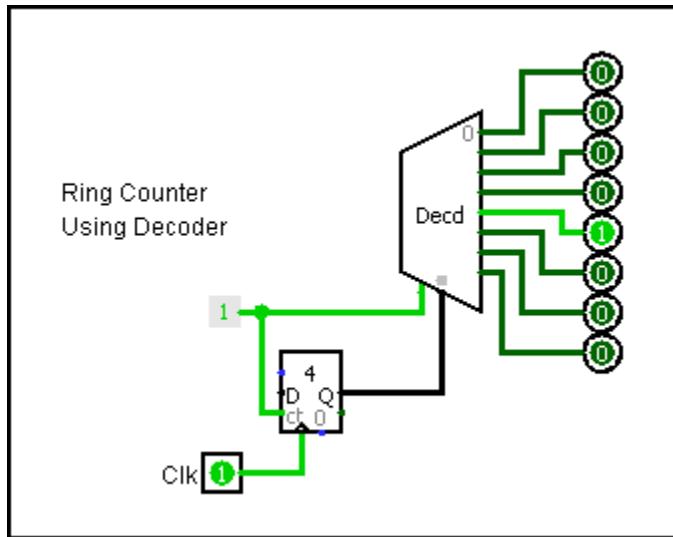


FIGURE 168: RING COUNTER FROM DECODER

#### APPLICATIONS

Counters can be used in a number of important applications. One of the most evident is a timer circuit, such as found in any system that displays a clock; but there are other practical uses for counters that may not be immediately obvious.

Note that all of the counters developed in this section are synchronous. While it is possible to build counter circuits using either asynchronous or synchronous techniques, synchronous counters are more common and have fewer problems. Since there is no compelling reason to develop these circuits with both types of counters, only synchronous counters will be used here; though, of course, the designer could opt for either in an application.

#### UP/DOWN COUNTER

Counters can be designed to count either up or down. Following is a 3-bit up counter:

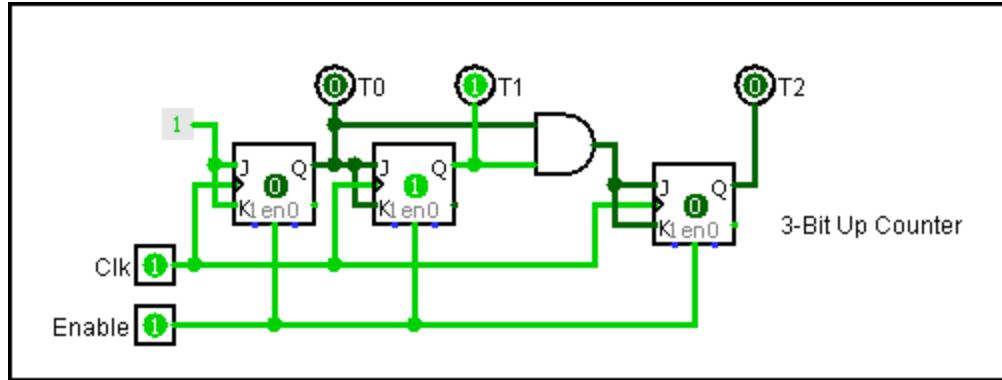


FIGURE 169: 3-BIT UP COUNTER

The output sequence for this counter follows (notice that the first stage in the above illustration, T0, is the low order bit):

Clk	T2	T1	T0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

TABLE 92: 3-BIT UP COUNTER SEQUENCE

The same circuit can also be used to create a 3-bit down-counter; however, though Q is used as the output, Q' drives subsequent stages:

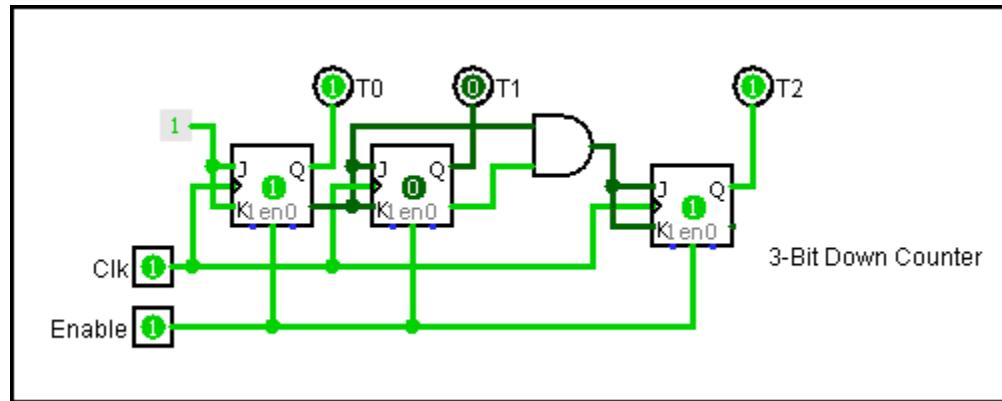


FIGURE 170: 3-BIT DOWN COUNTER

The output sequence for this counter follows (notice that the first stage in the above illustration, T0, is the low order bit):

Clk	T2	T1	T0
0	1	1	1
1	1	1	0
2	1	0	1
3	1	0	0
4	0	1	1
5	0	1	0
6	0	0	1
7	0	0	0

TABLE 93: 3-BIT DOWNCOUNTER SEQUENCE

Using similar techniques, an up or down counter with any number of stages can be built. It is also possible to build a more versatile counter that can count either up or down (the designer's choice) by adding a few more AND and OR gates so either the Q or Q' output can be sent to following stages.

#### COUNTERS FOR ANY INTEGER

All of the counters presented in this book have counted up to (or down from) a specific power of two. For example, a 2-bit counter can count 0-3 ( $2^2$  states) and a 3-bit counter can count 0-7 ( $2^3$  states). In general, any additional stage (or flip-flop) added to a counter will increase the states that counter can generate by a power of two. Thus, four stages will count 0-15 ( $2^4$  states) and five stages will count 0-31 ( $2^5$  states).

However, a designer will frequently need a counter that does not cycle on a specific power of two; for example, a decade counter needs to count 0-9. A synchronous counter that recycles before all possible states have been reached can be built using a common counter with additional logic.

Consider the following diagram of a Mod-3 counter (that is, a counter with only three states that counts 0-1-2 and then recycles).

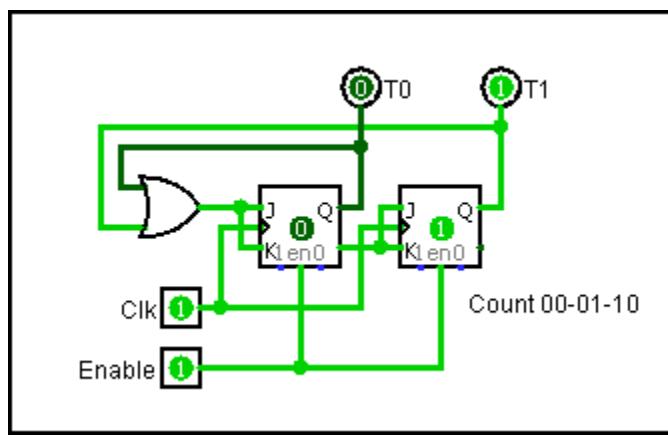


FIGURE 171: MOD-3 COUNTER

In this circuit, T1 is the least significant bit. Note how this circuit counts up to 10, but the next clock pulse returns it to 00 rather than 11. The count sequence is 00, 01, 10, 00....

A Truth Table can be constructed for the Mod-3 counter:

Clk	T0	T1
0	0	0
1	0	1
2	1	0

TABLE 94: MOD 3 COUNTER SEQUENCE

A mod-10 (or decimal) counter is very commonly needed in circuits. The logic diagram for a mod-10 counter follows:

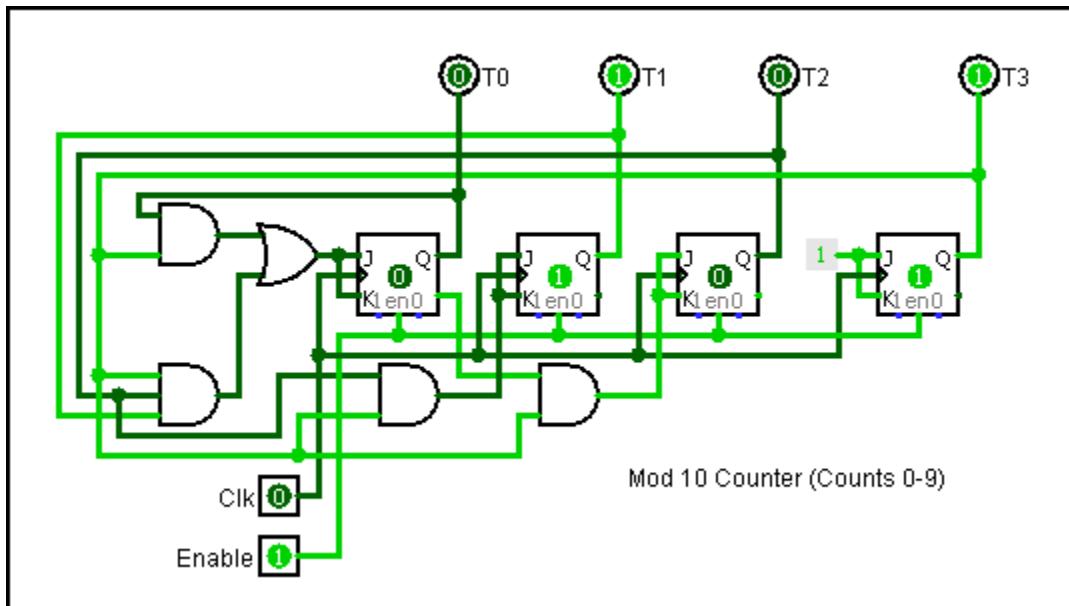


FIGURE 172: MOD-10 COUNTER

In a mod-10 counter, each stage is driven by outputs of other stages such that values 1010, 1100, 1101, 1110, and 1111 are not possible. In the above diagram, the last stage (T3) is the least significant bit and the first (T0) is the most significant bit. The best way to see how this circuit counts is to actually build the circuit in a logic simulator and click the clock input.

It is possible to create a counter that will start and stop on any binary number. Moreover, it is possible to create a counter that will skip numbers or count in some other odd sequence. Each of these modifications would require some sort of logic gate network to support the flip-flop counting function.

#### FREQUENCY DIVIDER

Often, a designer creates a system that may need various clock frequencies throughout its subsystems. In this case, it is desirable to have only one main pulse generator, but divide the frequency from that generator so other frequencies are available where they are needed. As a simple example of a divide-by-four device, consider the two-stage synchronous counter described above:

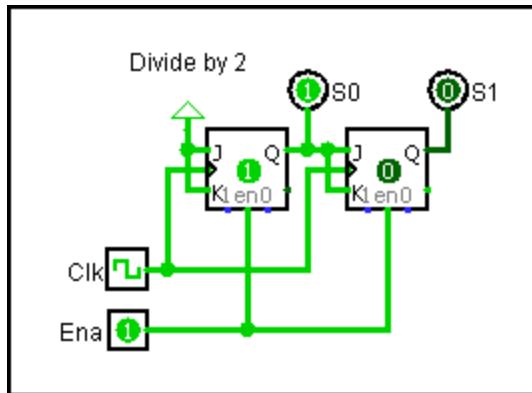
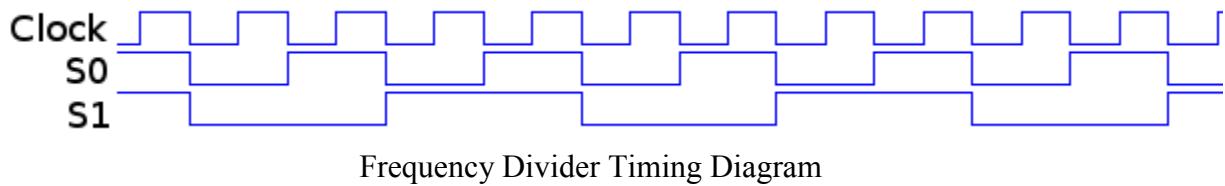


FIGURE 173: 2-BIT FREQUENCY DIVIDER

Following is the timing diagram for the clock input and both outputs:



Notice that after each negative-going clock pulse, S0 changes states. Thus, the frequency of S0 is one half that of the input clock. In addition, the output of the second stage (S1) changes only half as often as the first stage, S0. Thus, the frequency of S1 is equal to Clk/4. This is a divide-by-four frequency divider.

By adding more stages to the circuit, the designer can easily obtain a divide-by-eight, divide-by-sixteen, and other powers of two frequency divider circuits. The output from a modulus counter can also be used as an input if a designer needs some other divisor (like a divide-by-ten). For example, the most significant bit of a mod-10 counter will cycle on/off after every 10 clock pulses. That output could be used to create a divide-by-ten frequency divider. Finally, frequency dividers can be cascaded to provide large divisors. For example, a divide-by-six could cascade with a divide-by-ten to create a divide-by-sixty divider.

#### FREQUENCY COUNTER

A frequency counter is nothing more than a regular binary up-counter, but the clock pulse is provided by the frequency source being tested. The frequency counter is turned on/off using a precise quartz-controlled timer (maybe, for example, it will be on for only one second); and while it is on, it will count the test frequency since that is the "clock" driving the counter. These are fairly simple circuits to build, so the schematic is not presented here.

#### COUNTER INTEGRATED CIRCUITS (IC)

In practice, most designers do not build counter circuits since there are so many types available already packaged in an integrated circuit. When a counter is needed, a designer can simply select an appropriate counter from all of those available on the market. Here are just a few as examples of what is available:

IC Number	Function
74x68	dual 4 bit decade counters
74x69	dual 4 bit binary counters
74x90	decade counter (separate divide-by-2 and divide-by-5 sections)
74x143	decade counter/latch/decoder/7-segment driver
74x163	synchronous 4-bit binary counter with synchronous clear
74x168	synchronous 4-bit up/down decade counter
74x177	presetable binary counter/latch
74x291	4-bit universal shift register, binary up/down counter, synchronous

FIGURE 174: VARIOUS COUNTER ICS

As an example, the 74x163 is a 4-bit binary counter. Here is the pinout for that IC:

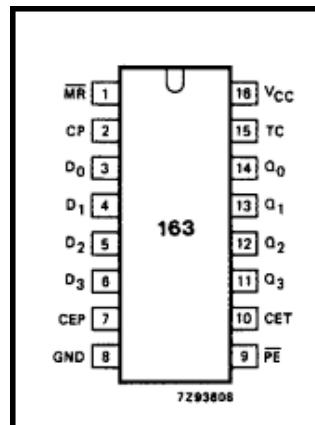


FIGURE 175: 74X163 PINOUT

The designer can choose to use either a positive or negative edge clock and pull the various outputs as needed. This one IC could replace the 3-stage synchronous counter illustrated above by connecting outputs Q0-Q2 and simply ignoring Q3. Also, the counter would have to be cleared after a count of 111 by connecting Q3 to the CLR input; so when Q3 goes high the IC would be reset back to 0. In fact, this counter could be made to count up to any 4-bit number (like 1010) and then reset by using a 4-input AND gate with inputs drawn from appropriate positive and negative values for Q0-Q3 and its output connected to the IC's CLR.

Logisim includes a 74x163 counter and following is a circuit built with that counter:

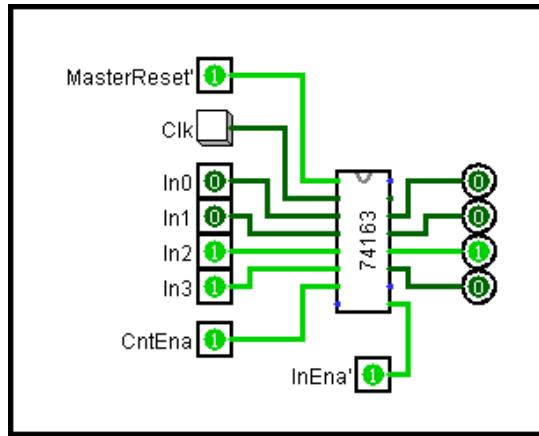


FIGURE 176: 74X163 IN LOGISIM

In this circuit, a push button is being used as a clock pulse to make the circuit operation simple. The Master Reset is an active low input (that is,  $\text{MasterReset}'$ ), so it needs to be high for the circuit to operate, and then sent low momentarily to reset the output pins to zero. The four input pins ( $\text{In}0$ - $\text{In}3$ ) can be used to pre-set the output so the counter can start at any 4-bit number.  $\text{CntEna}$  must be high for the counter to work, and when  $\text{InEna}'$  goes low then the input pins ( $\text{In}0$ - $\text{in}3$ ) are used to set the output pins.

## 6.8: LAB – 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:

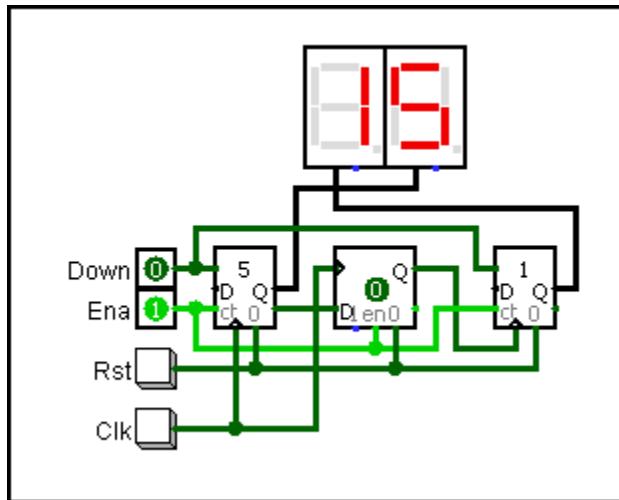


FIGURE 177: 2 DIGIT UP/DOWN COUNTER

Logisim counters have a number of important settings. For this circuit, 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 four control signals on this circuit.

- When "Down" is high the circuit counts down; otherwise it counts up.
- "Ena" enables the circuit.
- The "Rst" button resets the count to zero.
- The "Clk" button is 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 and that becomes "Data" for the D Flip-Flop. The next clock tick then resets the Flip-Flop and the circuit is set for the next overflow.

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 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 6.8: Timer", and today's date. Save the file as *Lab 6\_8 – Timer*.



---

## 7: APPLICATIONS

### 7.1 INTRODUCTION

Digital logic applications are systems of components designed to do a specific job. In this chapter certain applications are developed, leading up to a Central Processing Unit. This is the culminating chapter of the book since a Central Processing Unit is, arguably, the most important digital logic application in use. All of the applications presented in this chapter are built on the simple gates and Boolean algebra presented in the beginning chapters of the book.

## 7.2: LOGIC ARRAYS

### PROGRAMMABLE LOGIC ARRAY (PLA)

A Programmable Logic Array (PLA) is a circuit that contains a generic group of AND and OR gates wired in such a way that all possible minterm outputs can be created by a designer in the field. Using a PLA saves a designer considerable time in creating a matrix of AND/OR gates and also greatly simplifies the manufacturing of circuit boards that require such a matrix.

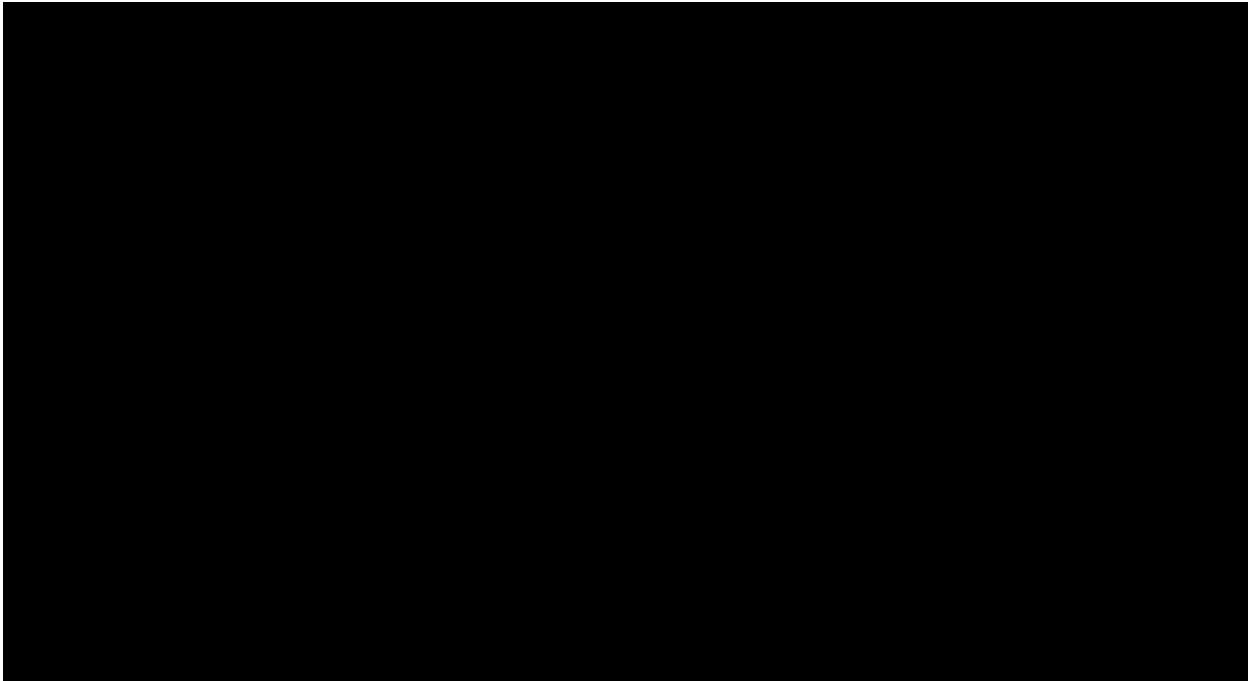


FIGURE 178: SIMPLIFIED PROGRAMMABLE LOGIC DEVICE

<sup>4</sup>The image above is a representation of a very simple PLA with only two inputs in the top left corner of the circuit. Within the PLA, those two inputs are all inverted and then both the original value and the inverted value are presented to the inputs of two AND gates. The outputs for the two AND gates are all presented to the input of an OR gate, which then feeds the output on the right side of the diagram.

The symbol that looks like a lazy "S" is a fusible link. There is a way to "blow out" any of those fuses by applying an appropriate signal to the PLA (however, that particular process is beyond the scope of this book). By selectively leaving and removing the fusible links, a designer can shape the overall function of the PLA to match any given circuit requirement. Thus, a generic PLA can work in many different applications.

---

<sup>4</sup> This image was found at Wikimedia:  
[http://en.wikipedia.org/wiki/File:Programmable\\_Logic\\_Device.svg](http://en.wikipedia.org/wiki/File:Programmable_Logic_Device.svg). It was released into the public domain by the original author, “DnetSVG.”

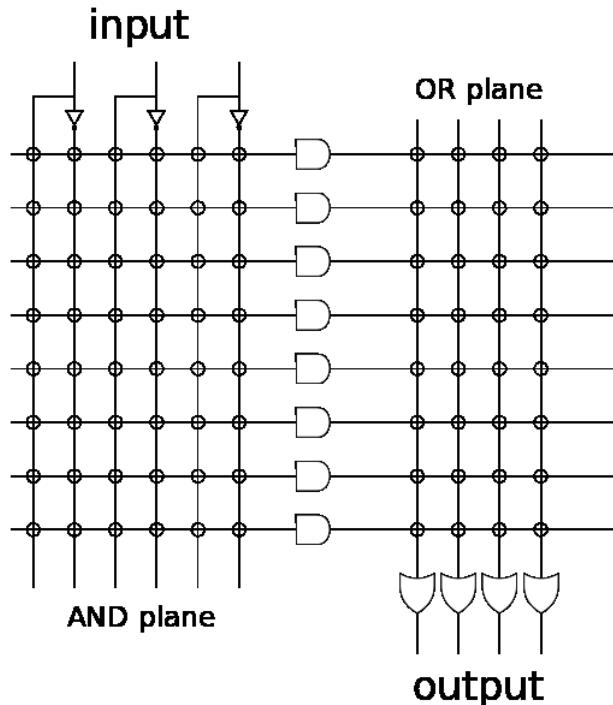


FIGURE 179: PROGRAMMABLE LOGIC ARRAY

<sup>5</sup>The illustration above is a more detailed example of a PLA at work. The three inputs are on the top left side of the circuit. Both the original signal and its inverted value are fed into a large AND plane. The designer would indicate which of the inputs would be fed to a specific AND gate by darkening the circle at the intersection of the two. On the right side of the circuit is the OR plane. Again, the designer would darken the circles at the various intersections to indicate which AND gate output are fed into specific OR gates. The circuit's output is at the bottom right corner.

Notice that the AND and OR gates are illustrated with a single input line. That is for convenience. In reality, each of those gates would have an input line from every possible input signal; however, using only a single input line to represent all of those individual lines makes the diagram much cleaner and easier to understand.

The circuits illustrated above are greatly simplified to improve understanding; a true PLA is much more complex. For example, the Signetics 82S153A PLA contains 18 inputs, 42 AND gates and 10 OR gates.

#### PROGRAMMABLE ARRAY LOGIC (PAL)

Programmable Array Logic (PAL) ICs are generic minterm processors, like PLAs above. However, the difference is that in a PAL, the AND gates are pre-wired to specific OR gates; that is, the OR gates are

<sup>5</sup> This image found at Wikimedia:

[http://commons.wikimedia.org/wiki/File:Programmable\\_logic\\_array\\_%28schematic\\_drawing\\_example%29.svg](http://commons.wikimedia.org/wiki/File:Programmable_logic_array_%28schematic_drawing_example%29.svg). It was created by "Ilia Kr" and released under the Creative Commons Attribution-Share Alike 3.0 Unported license.

designed in such a way that their wiring cannot be changed by the designer in the field. Thus, all of the customization is done by manipulating the inputs to the AND gates. While this seems to restrict the designer, it actually makes the circuit easier to design since the OR gates do not need to be considered.

As an example, consider the following diagram which was snipped from the specification sheet for a PAL 16R8, Programmable Array Logic:

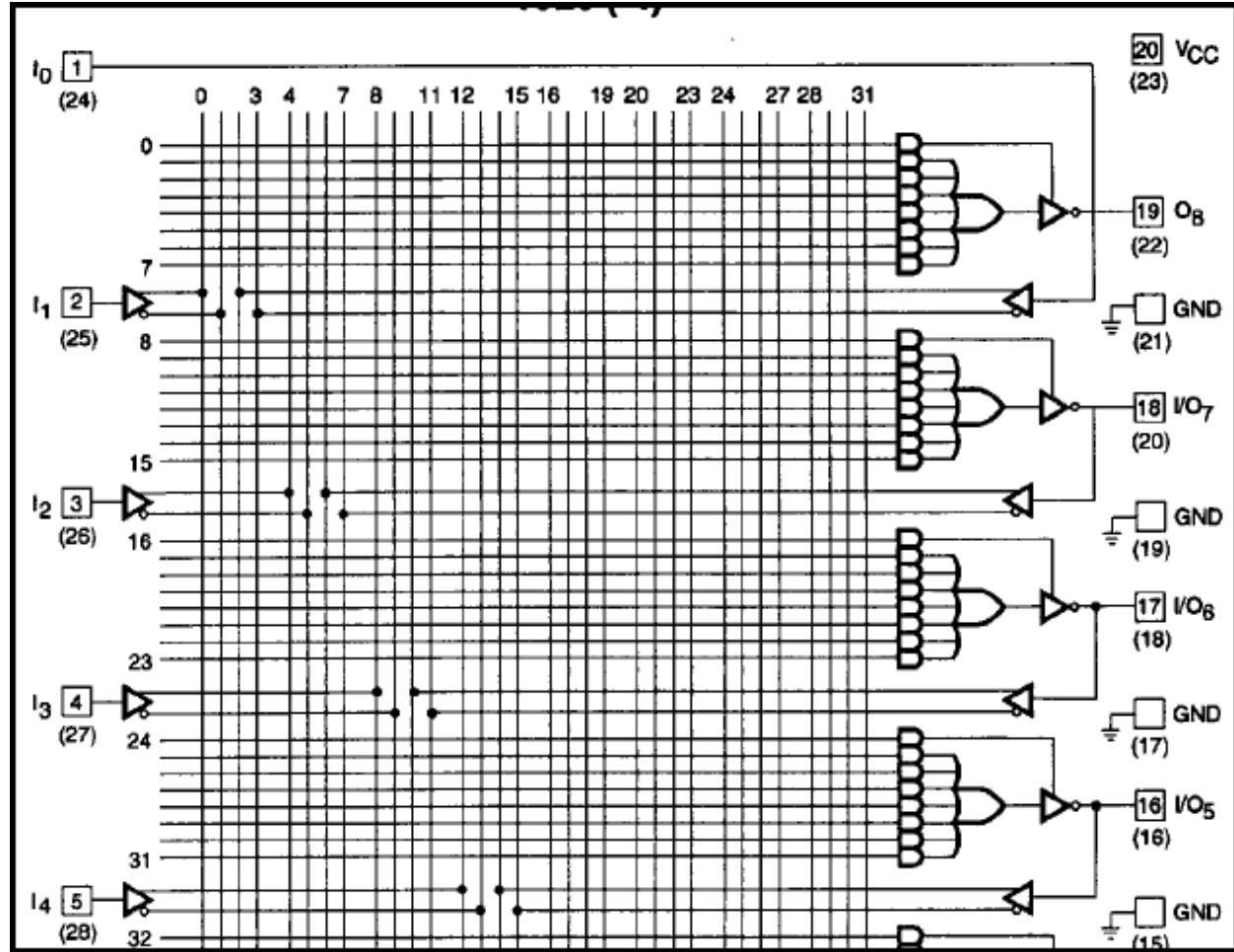


FIGURE 180: PAL 16R8 (EXCERPT)

Input 1 (I<sub>1</sub> on Pin 2) is linked to internal line 0, so I<sub>1</sub> can be fed to the top array of AND gates:

Notice that the second AND gate in this set (horizontal line 1) can take the input from any of the internal lines; so it is, potentially, a 32-input AND gate. The output from that gate is then fed through the OR gate and eventually out to OUTPUT 8 on Pin 19.

The input from I<sub>1</sub> (Pin 2) is also inverted and sent to internal line 1, so the negative of I<sub>1</sub> can be used in any of the AND gates in the grid.

While this circuit seems very complex, it is rather simple for the designer to "blow out" all unused links and create a custom circuit for some application in the field.

### 7.3: MEMORY (ROM AND RAM)

#### ROM

Read Only Memory (ROM) is an Integrated Circuit (IC) that contains tens-of-millions of registers (memory locations) to store information. Typically, ROM stores microcode (like bootup routines) and computer settings that do not change. However, the contents of a special kind of ROM (like EPROM or EEPROM) can be altered by the computer under special circumstances so it can be used to store changed information. An example of where that type of ROM would be used is in a computer's BIOS (the bootup operating system), where the user can alter certain "persistent" computer specifications, like the device boot order. One major difference between Read Only Memory and Random Access Memory (RAM) is ROM's ability to hold data when the computer is powered off.

Logisim includes a ROM memory component, as demonstrated in the following circuit:

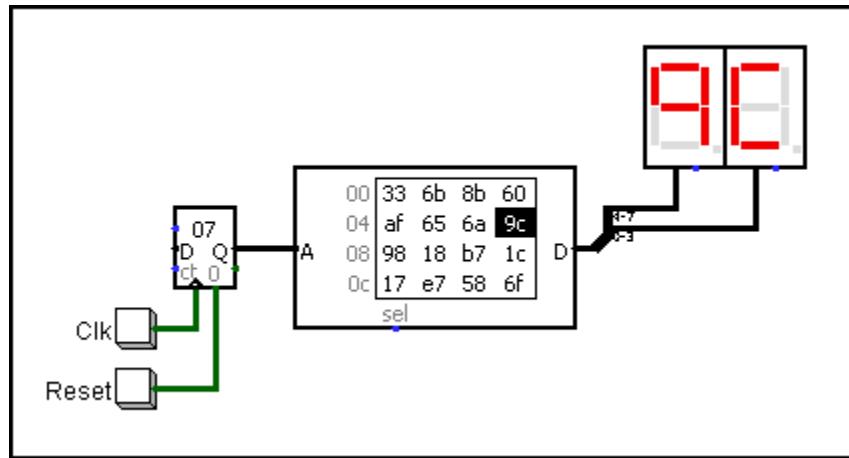


FIGURE 181: ROM IC IN LOGISIM

The ROM component is the large rectangle in the center of the circuit. Logisim helpfully displays the contents of each byte of memory in the component. The address to be accessed is entered on the "A" pin on the left, and the data at that location appears on the "D" pin on the right. In the illustration, address 07 is being entered by the counter on the left, and the contents of that location, 9C, is being made available on the data pin on the right. The counter can be incremented by one with each click of the "Clk" button, and it can be reset to 00 by a click on the "Reset" button.

In a Logisim circuit, the contents of the ROM component can be loaded from a memory file before the simulator is started; and then information contained in ROM can be accessed as needed. As an example of where this would be useful, consider a Central Processing Unit (CPU). Among other things, a CPU is required to move data between internal registers and external devices (like a monitor or hard drive). There are certain codes needed to turn on or off switches in data paths in order to move the bytes to their intended destination (something like switching tracks in a complex rail yard to move a train to its destination). The codes needed to manipulate those electronic switches could be contained in a series of bytes in ROM, and then those bytes can be "played" to open and close data paths for specific functions, like moving a byte from a register to the hard drive for storage.

## RAM

Random Access Memory (RAM) is an IC that contains tens-of-millions of registers (memory locations) to store information. The RAM IC is designed to quickly store data found at its input and then look-up and return stored data requested by the circuit. In operation, RAM contains both program code and user input (for example, the Word program along with whatever document the user is working on).

Logisim includes a RAM memory component, as demonstrated in the following circuit:

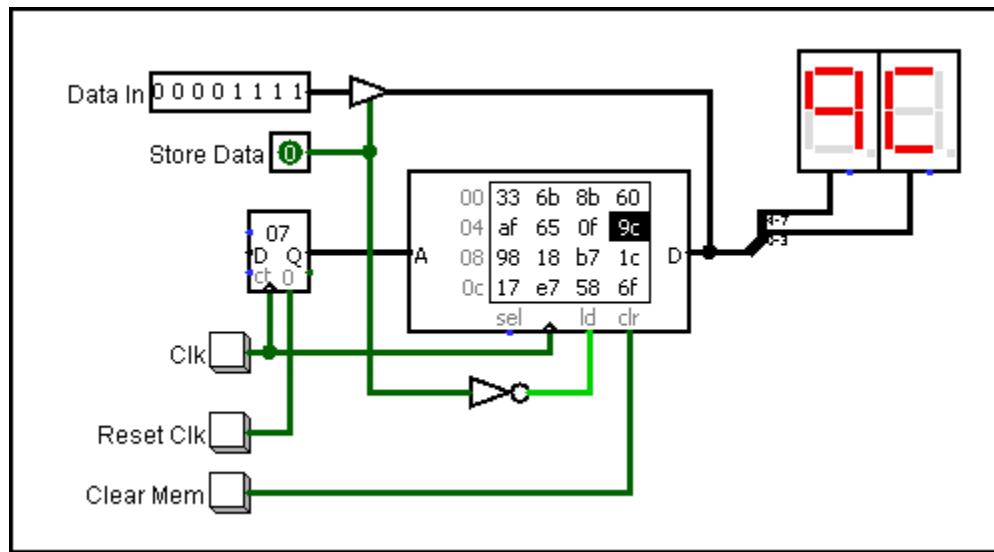


FIGURE 182: RAM IC IN LOGISIM

At first glance, the RAM component is similar to the ROM component. It is the large rectangle in the center of the circuit. Logisim helpfully displays the contents of each byte of memory in the component. The address to be accessed is entered on the "A" pin on the left, and the data at that location appears on the "D" pin on the right. In the illustration, address 07 is being entered by the counter on the left, and the contents of that location, 9C, is being made available on the data pin on the right. The counter can be incremented by one with each click of the "Clk" button, and it can be reset to 00 by a click on the "Reset Clk" button.

The RAM component, though, is more complex since the data port (on the right) is bidirectional, it can both write a byte into RAM or read a byte out of RAM. The "Data In" port on the top left corner contains "0F<sub>h</sub>" that is desired to be stored in RAM for some reason. The "Store Data" bit is turned on, which simultaneously activates the control buffer to put that value on the data bus and inactivates the "Id" port on the RAM, which switches the RAM's data port to bring "0F" into the RAM IC rather than read what is already at the selected address. After the "Store Data" bit is turned on, the next clock pulse will store "0F" on the data bus in the address contained in the address counter; and then increment that counter to point to the next memory location. In the illustration, notice that while 9C is at the current address, the memory location immediately preceding the current one contains 0F, which was loaded on the previous clock pulse.

In a Logisim circuit, the contents of the RAM component can be loaded from a memory file before the simulator is started; and then information contained in RAM can be changed or used as needed.

## 7.4: LAB –RAM LAB

### PURPOSE

Random Access Memory (RAM) is the most complex component in the Logisim simulator; it includes three different configurations and a number of different control signals. It is used to hold volatile data; that 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:

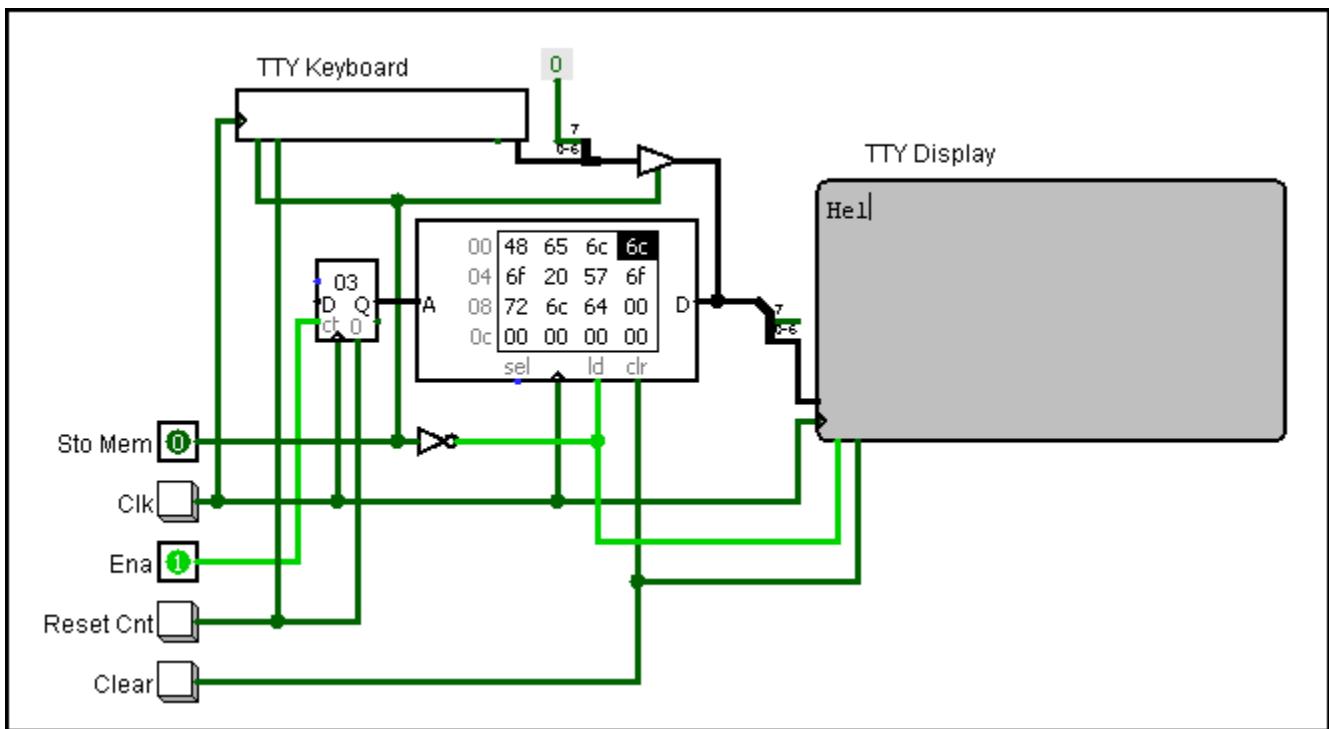


FIGURE 183: RAM LAB

The RAM component is the box with the numbers in the middle of the circuit. It is set up for an address bit width of eight and a data bit width of eight. That means that it can store 8-bit bytes in up to 256 locations. It is also configured for one synchronous load/store port. The way that RAM works is that an address is presented to the Address input port; this is the address in RAM that will load or store data. If the *ld* input control is high, then RAM will load data out to the data bus; but if *ld* is low then RAM will store whatever data appears on the data port.

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 whatever ASCII characters are stored in RAM will be sent to the display.

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

1. Set *Ena* to one to enable the circuit
2. Set *Sto Mem* to 1 so data at the *D* port will be stored in memory
3. Click the TTY Keyboard to activate it
4. Type a message; it should show up in the Keyboard import
5. Click the *Clk* button several times to store the contents of the keyboard in RAM
6. Reset *Sto Mem* (put it to 0) so data will be loaded out of RAM to the *D* port
7. Click *Reset Cnt* to reset the address counter to 0, this starts back to the beginning of RAM
8. Click the *Clk* button several times to load the contents of RAM into 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 7.4: RAM", and today's date. Save the file as *Lab 7\_4 – RAM*.

## 7.5 LAB – ROM LAB

### PURPOSE

Read Only Memory (ROM) is used to hold data that must persist; that is, data that 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:

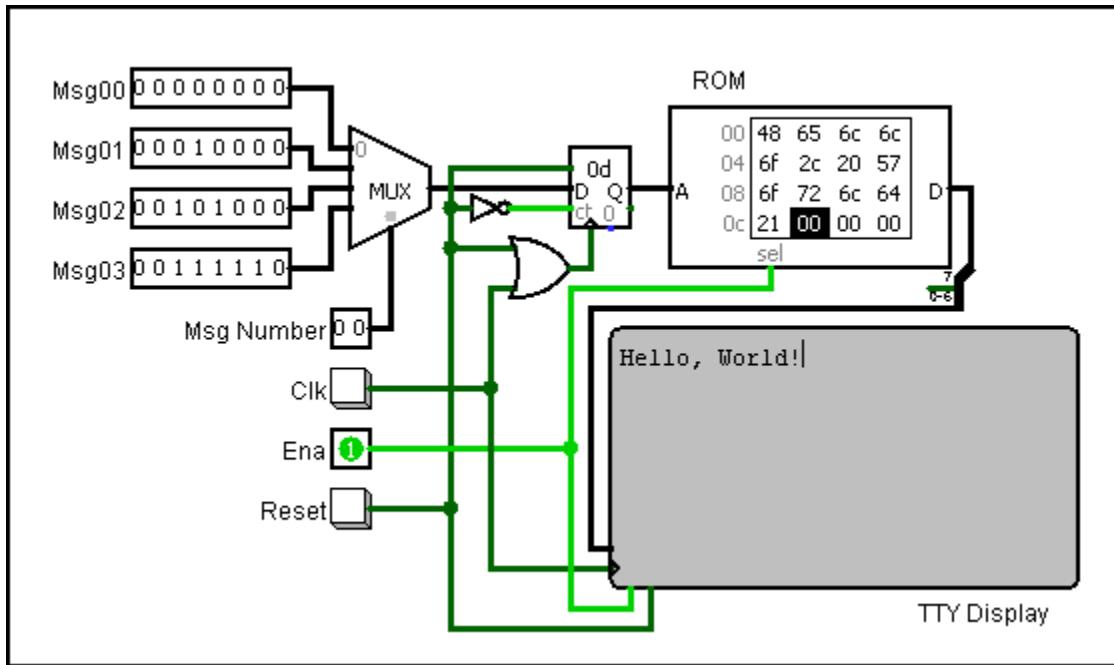


FIGURE 184: ROM LAB

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 then load the memory image
2. Enter the four message addresses in *Msg00*, *Msg01*, *Msg02*, and *Msg03*
3. Set *Ena* to one to enable the circuit
4. Set the desired message number
5. Click *Reset* to place address location in counter
6. Click the *C/k* button several times to load the contents of RAM into 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 7.5: ROM", and today's date. Save the file as *Lab 7\_5 – ROM*.

## 7.6: ARITHMETIC LOGIC UNIT (ALU)

### INTRODUCTION

An Arithmetic Logic Unit (ALU) is designed to provide basic mathematics and logical functions for a system. An ALU will provide these types of functions: add (with or without carry), subtract (with or without borrow), transfer the input to the output, increment the input by 1, decrement the input by 1, complement, AND, NAND, OR, NOR, and XOR.

### 74x181 ALU

Following is the logic diagram for a 74x181 ALU:

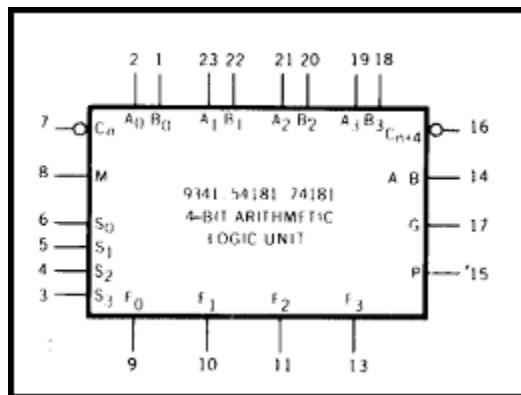


FIGURE 185: 74X181 ALU

In the 74x181 ALU, two 4-bit numbers can be input at A<sub>0</sub>-A<sub>3</sub> and B<sub>0</sub>-B<sub>3</sub> (on the top of the IC). Additionally, a carry in bit can be applied to the C<sub>n</sub>' input. The M input determines the mode of the circuit: when true, the circuit performs logic functions (like AND or OR); when false, the circuit performs arithmetic functions (like A+B or A-B). The function of the ALU is controlled by S<sub>0</sub>-S<sub>3</sub>, as described below. The output of the circuit appears on Q<sub>0</sub>-Q<sub>3</sub> (on the bottom of the IC), along with a carry out (C<sub>n+4</sub>) bit. Finally, if the function selected is to compare the two inputs, then output A=B will be true if the two inputs are equal. The G (Carry Generate) and P (Carry Propagate) outputs are used to cascade multiple ALUs.

Following is a partial truth table for the 74x181 ALU. Note, for simplicity, a second mode setting involving the carry bits is missing from this truth table; thus, there are many more functions available than implied by the table:

Input				Output	
S0	S1	S2	S3	Logic (M=H)	Arith (M=L)
L	L	L	L	A'	A
H	L	L	L	(A+B)'	A + B
L	H	L	L	A'B	A + B'
H	H	L	L	Logical 0	minus 1
L	L	H	L	AB'	A plus AB'
H	L	H	L	B'	(A+B) plus AB'
L	H	H	L	A XOR B	A minus B minus 1
H	H	H	L	AB'	AB' minus 1
L	L	L	H	A'+B	A plus AB
H	L	L	H	(A XOR B)'	A plus B
L	H	L	H	B	(A + B') plus AB
H	H	L	H	AB	AB minus 1
L	L	H	H	Logical 1	A plus A'
H	L	H	H	A+B'	(A + B) plus A
L	H	H	H	A+B	(A + B') plus A
H	H	H	H	A	A minus 1

TABLE 95: 74X181 ALU FUNCTION TABLE

An ALU is extremely versatile and can produce many different types of output from two input numbers. Using an ALU rather than coding with discrete gates saves the designer a great deal of time and effort.

#### ALU IN LOGISIM

There is no built-in ALU in Logisim; however, a serviceable ALU can be constructed rather easily using the components that are available. The following circuit illustrates the beginning of an ALU in Logisim:

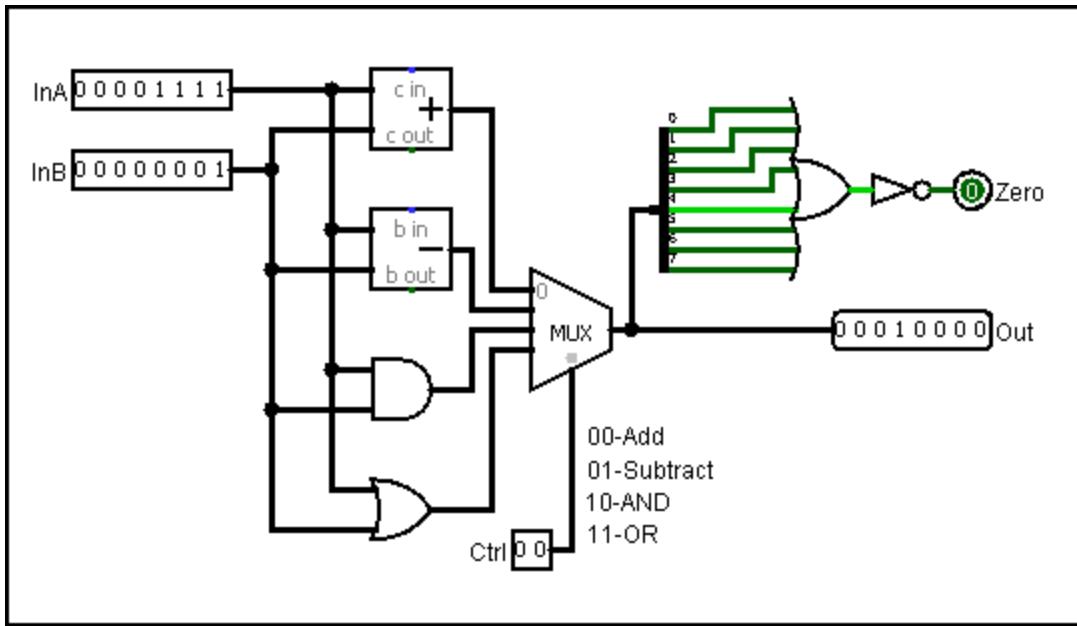


FIGURE 186: ALU CONSTRUCTED IN LOGISIM

This 8-Bit ALU includes two inputs ("InA" and "InB") that are fed into four devices that provide both arithmetic (adder and subtracter) and logic (AND and OR) functions. While all four devices receive input data, the output from only one of these devices is selected by a multiplexer using the "Ctrl" signal; and that data are sent to the output ("Out") port. In the illustration, the Ctrl is 00, which selects the Add function. 00001111 ("InA") is added to 00000001 ("InB") and the output is 00010000. Additionally, most CPUs provide various single-bit outputs called "flags" to indicate some output value. Commonly, a "zero" flag is activated if the output of an operation is zero. That is easily provided by sending all of the output bits to an OR gate and then inverting the output of that gate. If all bits are low, the OR gate would output a low and the inverter would then output a high, which indicates that the original operation's output was zero.

Of course, a much more complex ALU is possible by expanding this basic concept.

## 7.7: LAB – ARITHMETIC AND LOGIC UNIT

### PURPOSE

An Arithmetic and Logic Unit (ALU) is an essential component for many devices. The ALU performs dozens of operations which can be used by the logic requirements of the circuit. In this lab a simple ALU will be created using components that are built into Logisim.

### PROCEDURE

The ALU is divided into a main circuit and two sub-circuits: Arithmetic and Logic. Most of the work of the device is done in the two sub-circuits, while the main circuit supplies the various control signals needed.

*NOTE:* This is an 8-bit ALU. All components should be set to process eight data bits.

### ARITHMETIC

The arithmetic sub-circuit is designed to process the following:

- Add two 8-bit numbers
- Subtract two 8-bit numbers
- Multiply two 8-bit numbers
- Divide one 8-bit number by a second 8-bit number
- Provide the one's complement of an 8-bit number
- Provide the two's complement of an 8-bit number
- Increment by one an 8-bit number

Additionally, the arithmetic sub-circuit needs to provide certain *flags* (control signals) that indicate:

- Overflow, if in an addition operation the sign bit is different from the addends
- Negative, if the result of an operation is a negative number
- Carry, if the result of an operation includes a carry out

Following is the logic diagram of the Arithmetic sub-circuit:

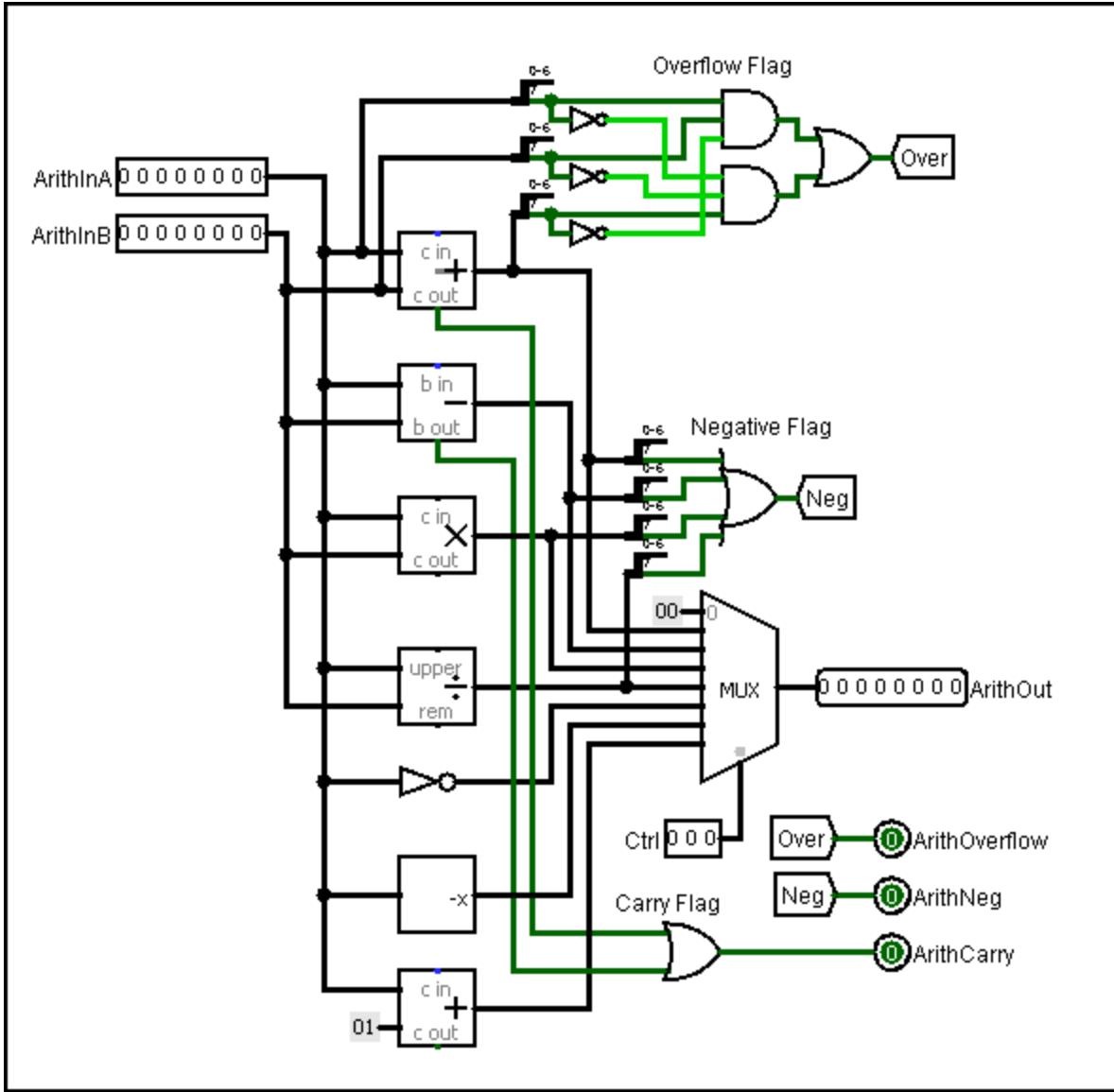


FIGURE 187: ALU ARITHMETIC SUB-CIRCUIT

The circuit is fairly simple to build. Two 8-bit inputs are fed to a series of various arithmetic components: add, subtract, multiply, and divide. Additionally, one input, *ArithInA* is sent to a NOT gate (which provides the one's complement), and a two's complement (identified as “-x” on the component). Finally, *ArithInA* is sent to one input of an adder while the other input has a constant 1; thus, incrementing *ArithInA* by one.

All component outputs are sent to an 8-to-1 multiplexor. Input 0 to the multiplexor is tied to a constant low since it is not used. The output of the multiplexor is controlled by a 3-bit control signal.

Signal	Output
000	Not Used
001	A+B
010	A-B
011	A*B
100	A/B
101	One's complement of A
110	Two's complement of A
111	Increment A

TABLE 96: ARITHMETIC CONTROL SIGNAL

The three flags are created by using various bits from the component outputs:

- Overflow: check bit 7 (the most significant bit, which is used as a sign bit). In an addition operation, if the two input bits are the same, and they are different from the output bit, then turn on the overflow flag.
- Negative: if bit 7 (the most significant bit, which is used as the sign bit) is high, then turn on the negative flag.
- Carry: if either the adder or subtractor indicates a "carry out" then turn on the carry flag.

The last note of interest concerns the use of Logisim "tunnels" in this circuit. A tunnel is an invisible wire that connects two or more points. They are used to keep a wire from snaking around a circuit and becoming a distraction. For example, the *Over* tunnel connects the output of the overflow flag circuit with the *Overflow* output port.

## LOGIC

The logic sub-circuit is designed to process the following:

- AND two 8-bit numbers
- OR two 8-bit numbers
- XOR two 8-bit numbers
- Shift Left one 8-bit number by the number of places indicated in a 3-bit number
- Shift Right one 8-bit number by the number of places indicated in a 3-bit number
- Rotate Left one 8-bit number by the number of places indicated in a 3-bit number
- Rotate Right one 8-bit number by the number of places indicated in a 3-bit number

Following is the logic diagram of the Logic sub-circuit:

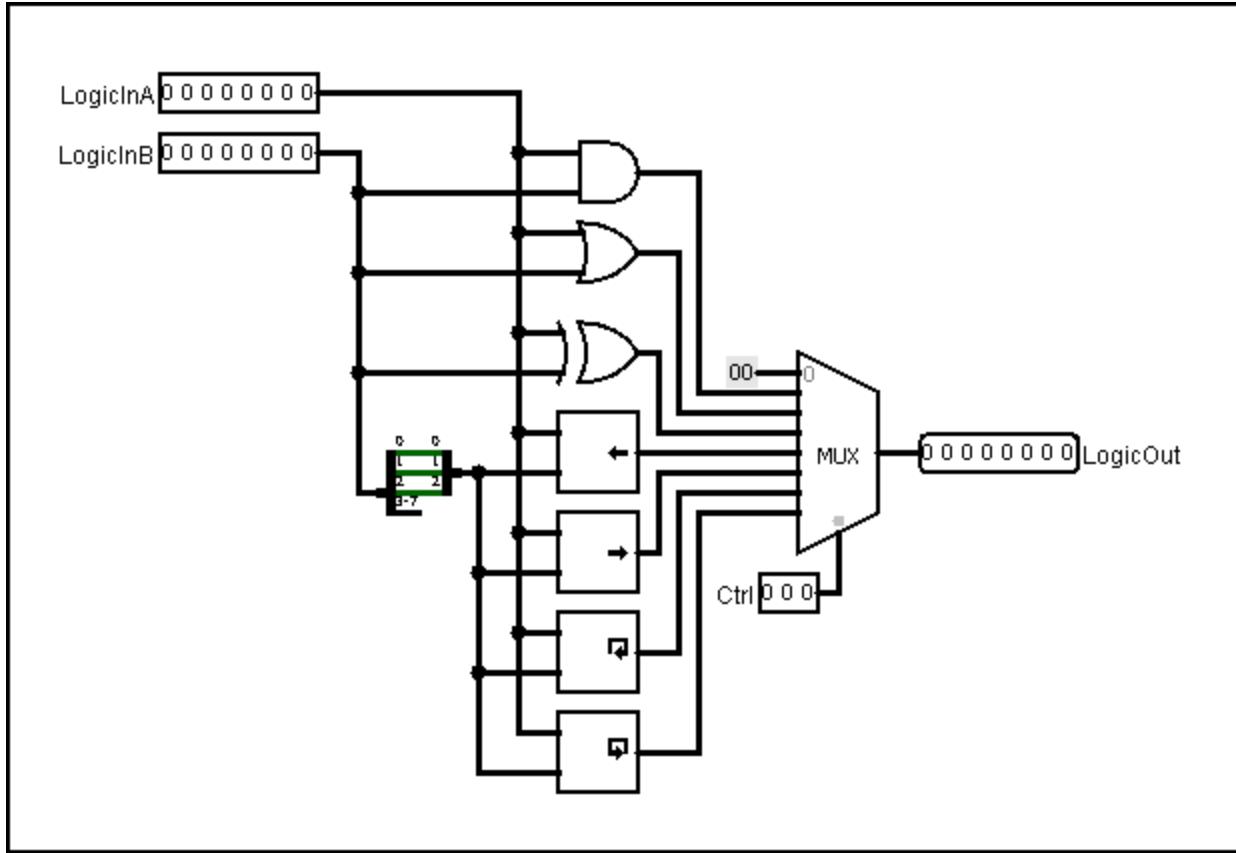


FIGURE 188: ALU LOGIC SUB-CIRCUIT

The circuit is fairly simple to build. Two 8-bit inputs are fed to a series of various logic components: AND, OR, XOR, Shift Left, Shift Right, Rotate Left, and Rotate Right.

All component outputs are sent to an 8-to-1 multiplexor. Input 0 to the multiplexor is tied to a constant low since it is not used. The output of the multiplexor is controlled by a 3-bit control signal.

Signal	Output
000	Not Used
001	A AND B
010	A OR B
011	A XOR B
100	Shift A left B times
101	Shift A right B times
110	Rotate A left B times
111	Rotate A right B times

TABLE 97: LOGIC CONTROL SIGNAL

#### CONTROL

The control circuit is designed to process two 8-bit numbers according to a control signal. Following is the logic diagram of the Control Circuit:

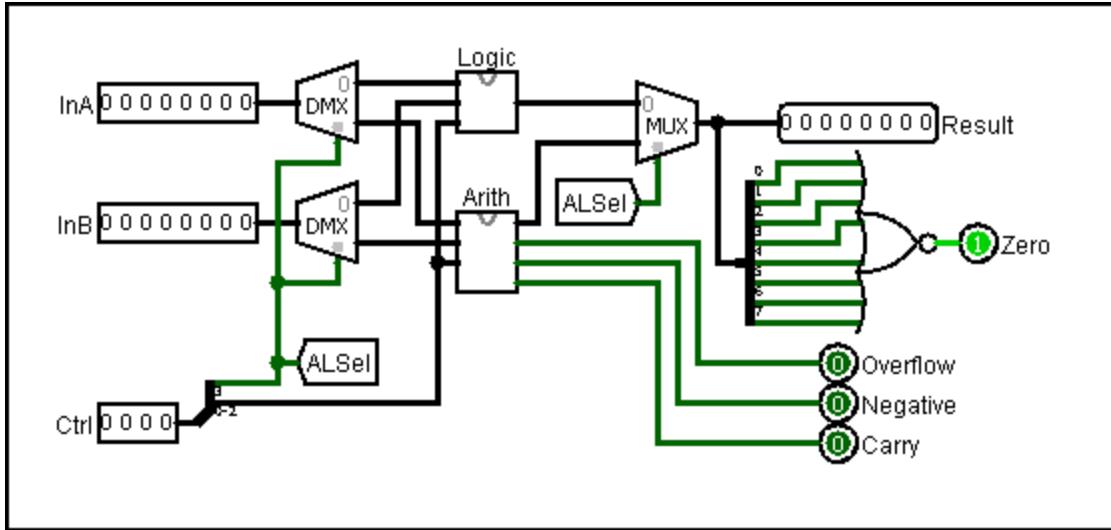


FIGURE 189: ALU CONTROL CIRCUIT

The two 8-bit numbers and 4-bit control signal is input on the left side of the circuit. Bit 3 (the most significant bit) of the control signal determines if the operation will be arithmetic (when high) or logic (when low), and is used to appropriately route the input numbers using two demultiplexors and one multiplexor. The only other logic component in this circuit is an 8-input NOR gate which outputs a high if the result of the specified operation is zero (that is, all 8 bits are low); this is the *Zero* flag.

#### CHALLENGE

This ALU does not have a function to compare the two input numbers to see if *InA* is greater than, equal to, or less than *InB*. However, that function is easily derived by calculating *InA* minus *InB* and then checking the various output flags. Connect appropriate logic gates to the various flags to create three new outputs: "A>B", "A=B", and "A<B."

#### CLEANUP

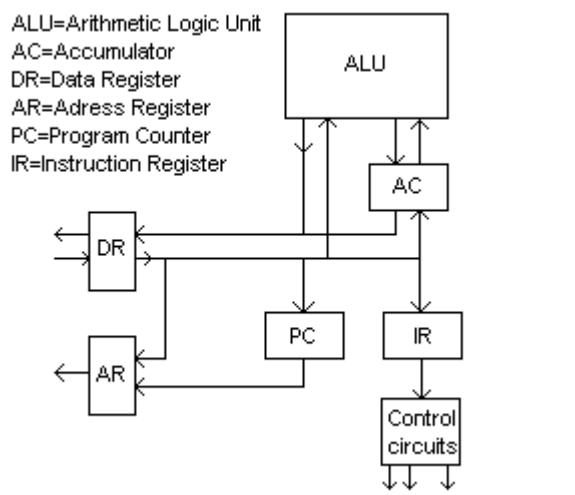
Rename the *main* circuit to *ALU*. Be sure the standard identifying information block is at the top left of the *ALU* circuit: Name, "Lab 7.7: ALU", and today's date. Save the file as *Lab 7\_7 – ALU*.

## 7.8: CENTRAL PROCESSOR UNIT (CPU): FUNDAMENTALS

### INTRODUCTION

<sup>6</sup>The Central Processing Unit (CPU) is the core of any computer, tablet, phone, or other computer-like device. While many definitions of CPU have been offered, many quite poetic (like the "heart" or "brains" of a computer), probably the best definition is that the CPU is the intersection of software and hardware. The CPU contains circuitry that converts the ones and zeros that are stored in memory into controlling signals for hardware devices. The CPU interprets the bytes contained in a software program and then retrieves data from the hard drive, sends pictures to the monitor, and permits users to complete intellectual work. In its simplest form, a CPU does nothing more than fetch a list of instructions from memory and then execute those instructions one at a time.

A CPU contains only a few components that are tied together with three different bus lines. The following block diagram shows the main components of a CPU:



7

FIGURE 190: CPU SIMPLE BLOCK DIAGRAM

Here is what each of these components do (from the top of the diagram). The *Arithmetic Logic Unit (ALU)* is responsible for all data manipulation, such as adding two numbers. The ALU sends the results of its work to a register called the *Accumulator*. Most CPUs will have a number of *Data Registers* that are used to temporarily store information while it is being processed. The *Address Register* contains an address in memory that is important for the current operation; perhaps, for example, the address of the next instruction to be executed. The *Program Counter* keeps track of what line in a program is the next to be executed. The *Instruction Register* contains the instruction that is currently being executed.

<sup>6</sup> Much of the material in this lab was adapted from a similar lab written by Dr. Lawlor for TKGate, another logic simulator. Here is the URL for that lab:

[http://www.cs.uaf.edu/2008/fall/cs441/lecture/09\\_09\\_cpu\\_construction.html](http://www.cs.uaf.edu/2008/fall/cs441/lecture/09_09_cpu_construction.html)

<sup>7</sup> This diagram was found at [http://commons.wikimedia.org/wiki/File:Cpu\\_adv.png](http://commons.wikimedia.org/wiki/File:Cpu_adv.png) and was released into the public domain by its author, identified only as Knoppson.

There are a number of control and timing circuits necessary for the operation of a CPU. Those circuits are designed to turn on and off various data paths within the CPU so data can be manipulated and sent to places like the ALU and memory. A specialized coding system (called *microcode*) is used to activate multiplexers and control buffers in order to move the data. The microcode used for any given CPU is specific for that device and the creation of that code is one of the most important challenges facing a CPU designer.

The best way to learn how CPU actually works is to build one. To be sure, the CPU that will be built in this class is very simple and wouldn't actually be used for any serious computer application. In fact, the device built here is simpler than even CPUs from the 1960s; however, there is enough detail in this project to show how digital logic plays a role within a CPU.

#### ITERATION 1

This CPU starts with a simple adder:

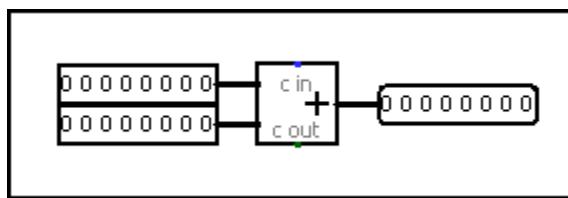


FIGURE 191: SIMPLE 8-BIT ADDER CIRCUIT

Notice that this is an 8-bit adder; and the two input ports, along with the output port, are also eight bits. This would mean that the bus lines connecting the inputs, the adder, and the output are also eight bits wide. This would be the start of an 8-bit CPU since the bus lines and devices are all 8-bits wide.

Expand the circuit so that both an adder and multiplier are being fed from the same two inputs:

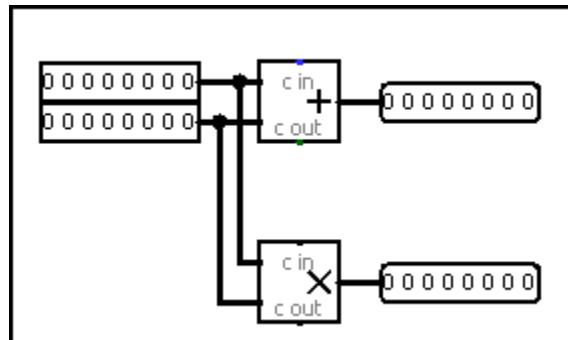


FIGURE 192: ADDER AND MULTIPLIER SHARING INPUTS

An output bus is now added so that the results of adding or multiplying two numbers will be available in a single output port:

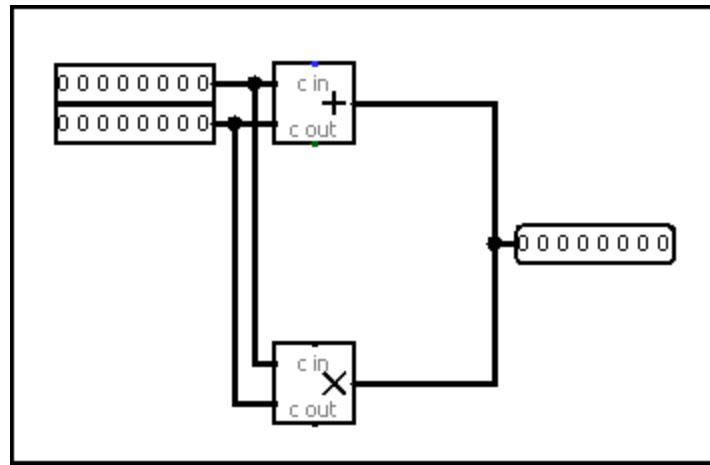


FIGURE 193: ADDER AND MULTIPLIER SHARING INPUTS AND OUTPUT

This would be fine, except that a single output bus is being fed by two different devices: an adder and the multiplier. This configuration is simply not possible. Any given bit on that output bus (for example, bit 7) may potentially be held high by one device and simultaneously low by the other, which would lead to a very unstable output. In fact, if this circuit is built in Logisim as presented, the output bus will turn red to indicate an error condition.

To compensate for that error, a *controlled buffer* is inserted after both the adder and the multiplier:

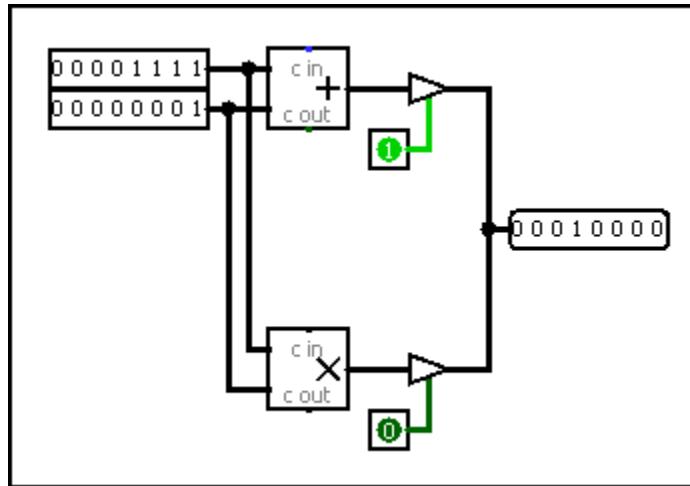


FIGURE 194: BUFFERS CONTROLLING THE OUTPUT BUS

A controlled buffer, sometimes called a tri-state device, is nothing more than a way to isolate a bus from a sub circuit. A controlled buffer, like any buffer, passes the input signal to the output; however, the buffer also includes a control input. If the buffer is turned on, that is, there is a high present at the control input pin, then the buffer will work as expected. If, though, the buffer is off, that is, there is a low on the control input pin, then the buffer appears as an open to the circuit (an open is a "broken wire" between the device and the bus). Technically, a low control voltage places a tri-state device in what is called a "high impedance" state; however, for our purposes, it is adequate to simply say the buffer appears as an open.

In the above illustration, the buffer on the adder is activated, that is, there's a high present on its control input; so the adder's output is moved onto the bus and can be seen at the output port. At the same time, the control for the multiplier must be low so that device is effectively removed from the circuit.

### ITERATION 2

This iteration of the CPU is a major revision, making the circuit more nearly functional. The revised circuit, illustrated below, still includes the adder and multiplier and their control buffers in the top right corner (now labeled as the Arithmetic Logic Unit, ALU). Three registers have been added (Reg 0, Reg 1, Reg 2), along with a control buffer for each register's output. The input for the registers comes from the ALU Bus. An 8-bit port labeled "ALU Bus" displays the contents of that bus for convenience.

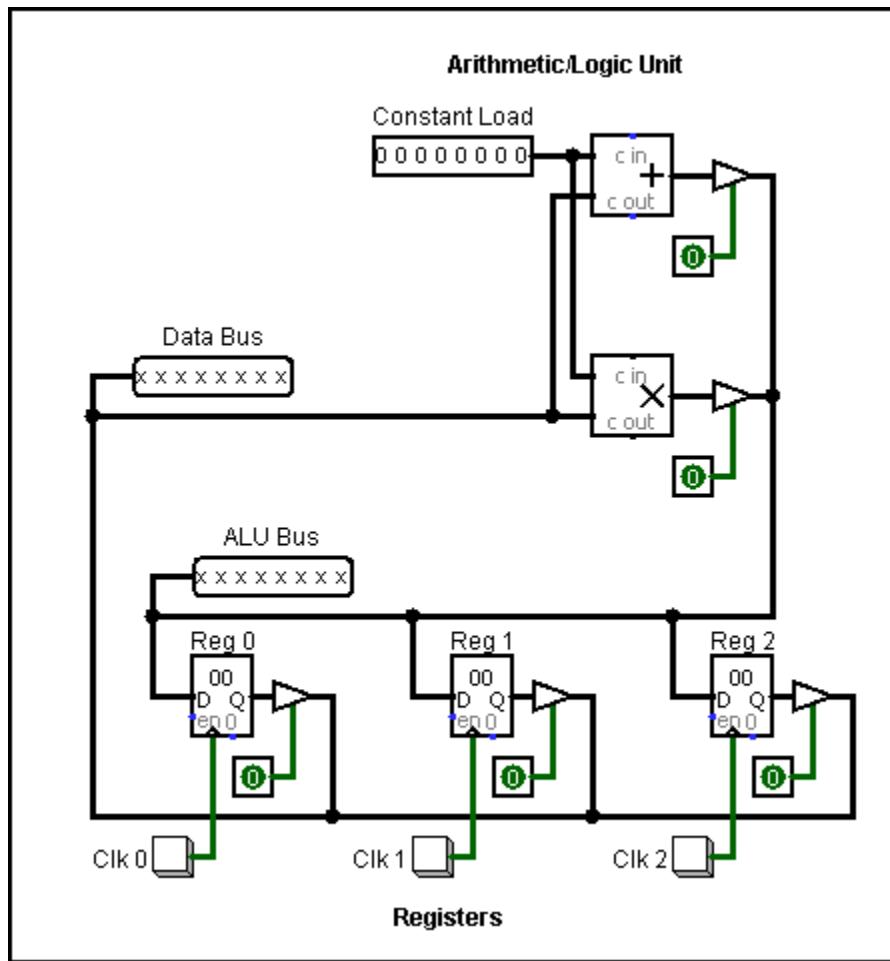


FIGURE 195: CPU FRAMEWORK

One input for the ALU is labeled "Constant Load." The other input is from the Data Bus, which contains the output from the registers. The data bus includes an 8-bit port labeled "Data Bus" that is only used to peek at the contents of the bus for troubleshooting.

A clock pulse has also been introduced to the circuit; however, each of the three registers are independently activated by the clock. In a true circuit, of course, the clock pulse would go to all of the

registers at once. Exactly which register gets the pulse would be controlled by a controlled buffer or other similar device.

### ITERATION 3

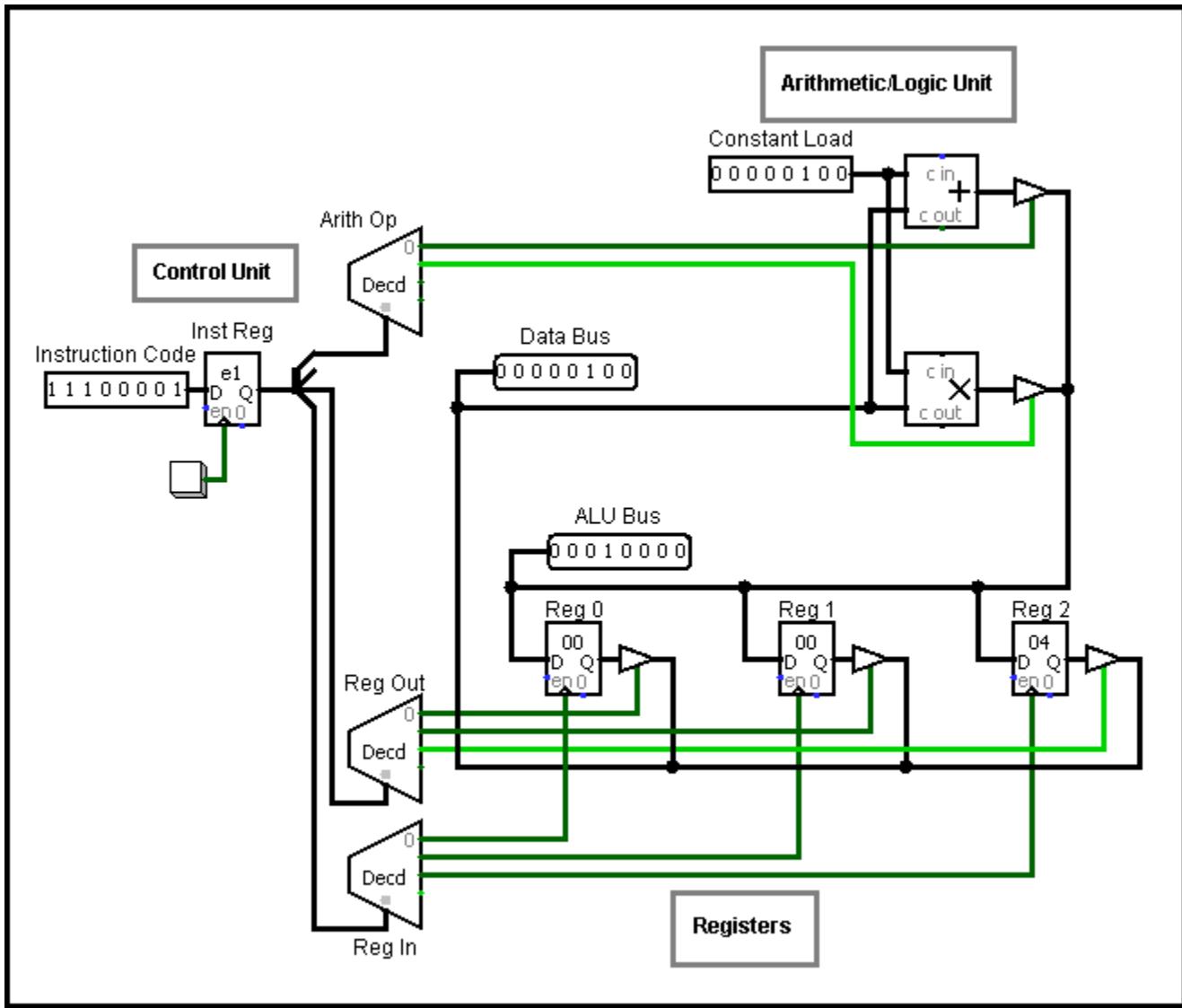


FIGURE 196: SIMPLE CPU

A Control Unit has been added to turn on/off the various buffers and registers so data flows appropriately through the circuit. Three decoders (labeled *Decd* in the diagram) have been added to channel the control signals where they need to go. Each decoder has two select bits, which gives them four outputs. The first decoder controls the ALU Bus and is labeled "Arith Op" since it determines which arithmetic operation will be present on the bus. The second decoder controls which of the three registers is actively connected to the data bus; it is labeled "Reg Out." The third decoder, labeled "Reg In", is connected to each register's clock input and it controls which register is used to store whatever is on the data bus

An 8-bit input port, labeled "Instruction Code," is fed into an Instruction Register and that code is used to control the three decoders (and the entire function of this circuit). The output of the Instruction Register is split four ways (the LSB is 0, the MSB is 7):

- Bits 0-1, ALU: 00=Add, 01=Multiply
- Bits 2-3: not used
- Bits 4-5, Reg Out to Data Bus: 00=Reg 0, 01=Reg 1, 10=Reg 2
- Bits 6-7, Reg In: 00=Reg 0, 01=Reg 1, 10=Reg 2

If 11100001 is entered in the Instruction Code port (as shown in the illustration) and then sent to the Instruction Register by activating its clock:

- 01 (bits 0-1): the Multiply control buffer will be active, so anything in that device will appear on the ALU bus. In the illustration, 100 (from the Constant Load) and 100 (from the Data Bus) are multiplied and 00010000 is placed on the ALU bus.
- 00 (bits 2-3): not used.
- 10 (bits 4-5): activate the control buffer for Reg 2, placing its contents, 100, on the data bus.
- 11 (bits 6-7): do not activate any of the register clocks, so nothing will be stored in a register.

By inputting various codes in the Instruction Register, a designer can cause data to be manipulated (by the ALU) and then stored in one or more registers. While this circuit is too simple to be of much value (for example, there is no external connection to memory, a monitor, or other devices), it does at least demonstrate the fundamentals of CPU operation.

## 7.9: CENTRAL PROCESSOR UNIT (CPU): COMPONENTS

### INTRODUCTION

A Central Processing Unit (CPU) includes a number of components that pass signals to each other and are tightly bound into a single device. In this module, the various components are described and their function in the overall circuit is developed. It is important to note that the CPU being discussed is a simple 8-bit device built for this class; it is not a practical device for use in some application. Therefore, many of the circuits presented are not as efficient as possible; however, they are intended to be used for teaching, and they serve that purpose well.

### GENERAL REGISTERS

CPUs require a number of registers that can be used for temporary storage of binary values ("scratch pads"). As an example, ASCII codes moving from memory to the screen are temporarily stored in a register while the data bus is busy with other activities (like updating the Program Counter). Actual CPUs may contain scores of registers, but the simple CPU being used for this class only has four:

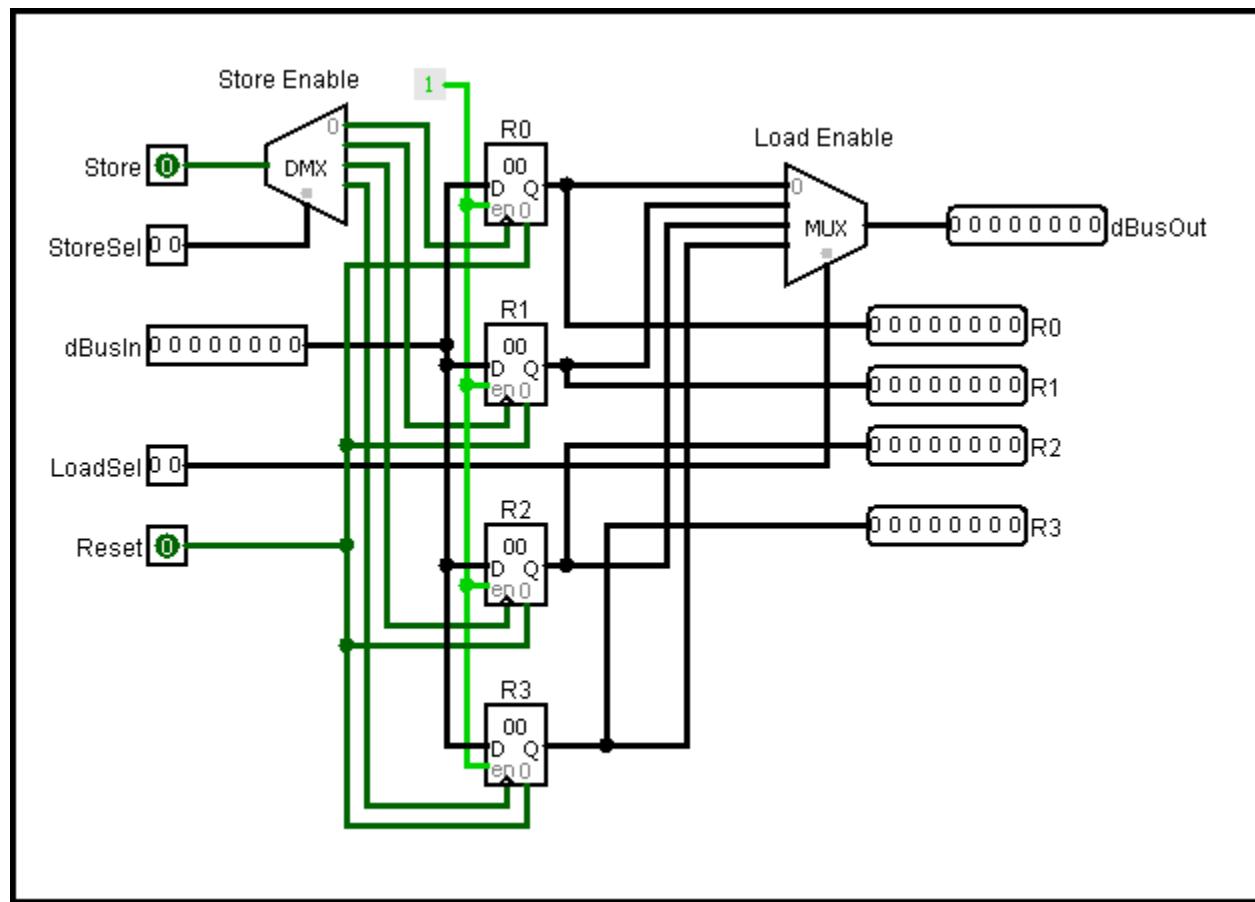


FIGURE 197: REGISTER BANK FOR LOGISIM CPU

In the center of this circuit are four registers. These registers are set up to store information on the falling edge of a clock pulse. That way, the register can be enabled through the demux on the leading

edge of a clock pulse and by the time that signal has propagated through the demux the register will be ready to actually activate when the clock pulse falls back to zero.

The "Store Enable" demultiplexer on the left side of the circuit determines which of the four registers are enabled for a store operation (so a value at dBusIn can be stored in that register). That demux is controlled by a two-bit input, StoreSel, so any of the four registers can be selected. When selected, a one-bit "Store" signal is applied to the enable port of the selected register.

The "Load Enable" demultiplexer on the right side of the circuit determines which of the four registers will be output to dBusOut.

Finally, the outputs labeled R0-R3 on the right side of the screen are only used as a teaching tool for the CPU's dashboard.

### RANDOM ACCESS MEMORY (RAM)

The program that is being executed is stored in Random Access Memory (RAM), and the CPU must be able to fetch it from there. Additionally, most programs store information in RAM that will be used as the program executes (for example, a document being written with a word processor would be stored in RAM). Finally, in many cases, certain hardware devices (like USB ports) are mapped as a memory location so the CPU treats them like they are an extension of RAM. RAM is physically located outside the CPU and is not technically a part of this device; however, it is so integral to the operation of a CPU it is included in this simulation.

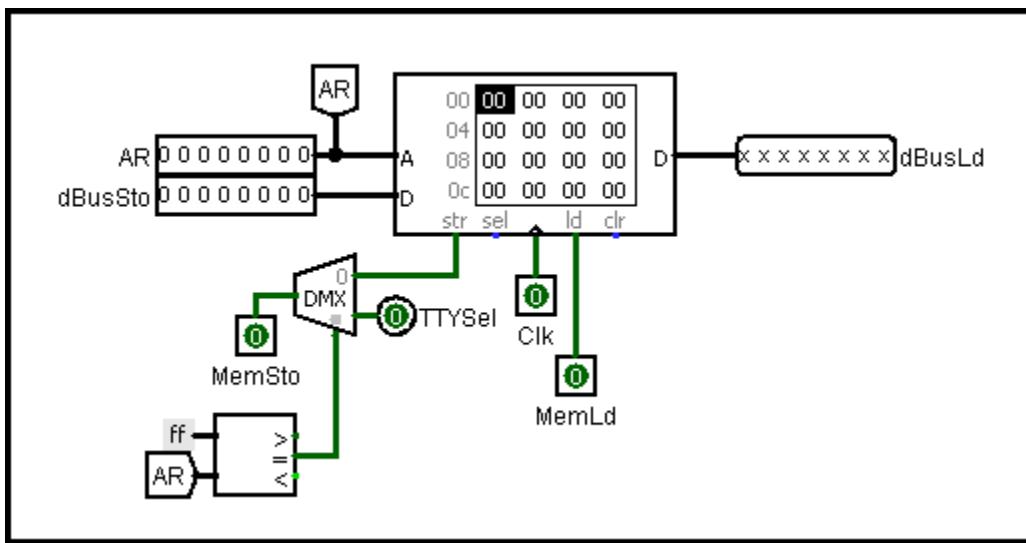


FIGURE 198: MEMORY FOR LOGISIM CPU

The RAM component consists of the RAM Integrated Circuit, which is the large box near the top center of this illustration. Logisim helpfully displays the RAM contents on the box, and highlights the currently selected address, to aid in troubleshooting a circuit.

On the left side of the RAM IC is an 8-bit address port where the current address is selected. The data port on the left ("dBusSto") is used to bring data from the data bus into the RAM IC for storage. The data

port on the right ("dBusLd") is used to load data from RAM onto the data bus. In practice, an address shows up on the address port and some data to be stored shows up on the dBusSto port, the "STR" pin (on the bottom left of the RAM IC) goes high, and on the next clock pulse that data are stored.

Whenever the "LD" pin (on the bottom of the RAM IC) goes high, then the data at the address on the address port are loaded out to the dBusLd port.

At the bottom of the illustrated circuit is a comparator and dMux. That sub-circuit compares the incoming address with the binary number 11111111 (or "FF" hex). If that is a match, then the incoming data are sent to the TTY display instead of being stored in RAM. By doing this, the TTY device is mapped as if it were memory location 111111 so it is easier to work with in the CPU's microcode.

Note: the simulated RAM IC in Logisim behaves in a slightly different way than a true RAM IC. Most RAM ICs have only a single data port that functions as both an input and an output port. To make it easier to understand the operation of RAM in this CPU circuit, separate input/output ports are used.

#### PROGRAM COUNTER

The Program Counter (PC) in a CPU is a simple register that contains the memory address for the next instruction to be executed. As an example, if instructions in memory locations 00, 01, and 02 have already been executed, then the PC will contain "03," which is the next address location with some sort of instruction to execute. In general, the PC is incremented at the end of every instruction execution so the next instruction can be fetched from memory.

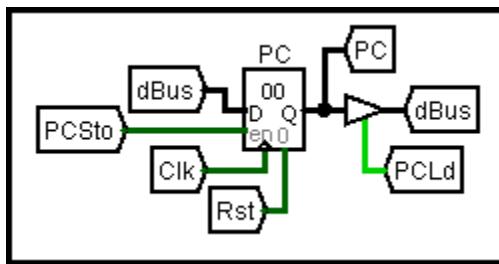


FIGURE 199: PROGRAM COUNTER FOR LOGISIM CPU

The simple PC illustrated above shows a connection to the Data Bus for both incoming and outgoing ports. This way, the PC can be loaded with a specific memory location in the event of a "JMP" instruction (where the program "jumps" to a new memory location and begins to execute there). The output of the PC is normally loaded into the Address Register so memory can deliver the instruction at that location.

#### ARITHMETIC LOGIC UNIT (ALU)

The Arithmetic Logic Unit (ALU) handles all arithmetic and logic functions in the CPU. The ALU component has been broken down into three parts: Logic Functions, Arithmetic Functions, and the ALU Control.

#### LOGIC FUNCTIONS

The Logic circuit is very simple:

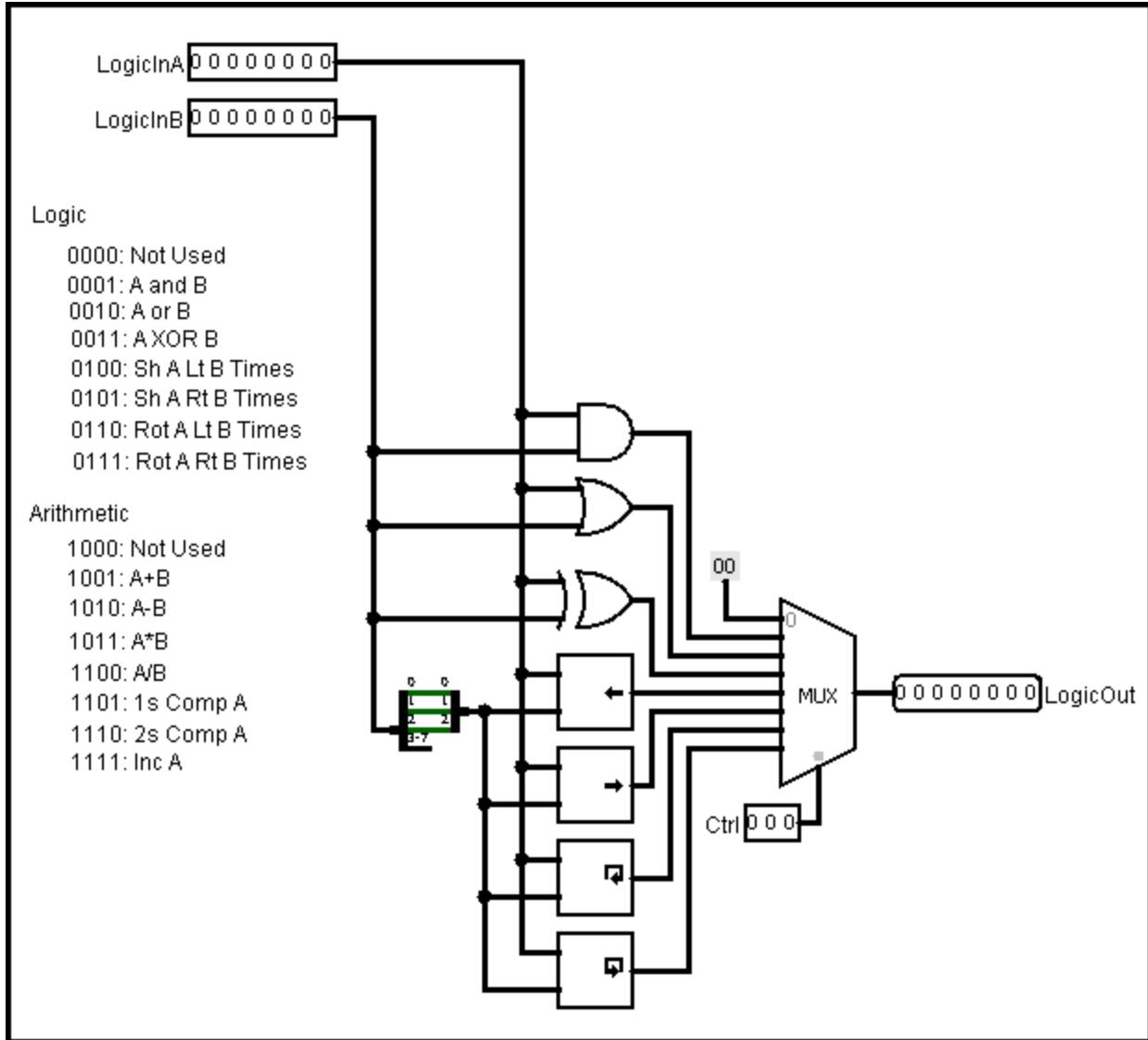


FIGURE 200: LOGIC PART OF THE ARITHMETIC LOGIC UNIT (ALU) FOR LOGISIM CPU

Input A and B are sent to seven different logic devices (which are built into Logisim). The output of the circuit is determined by an 8-to-1 multiplexor. Note that input 0 on the mux is tied to a constant low so that function is never used.

## ARITHMETIC FUNCTIONS

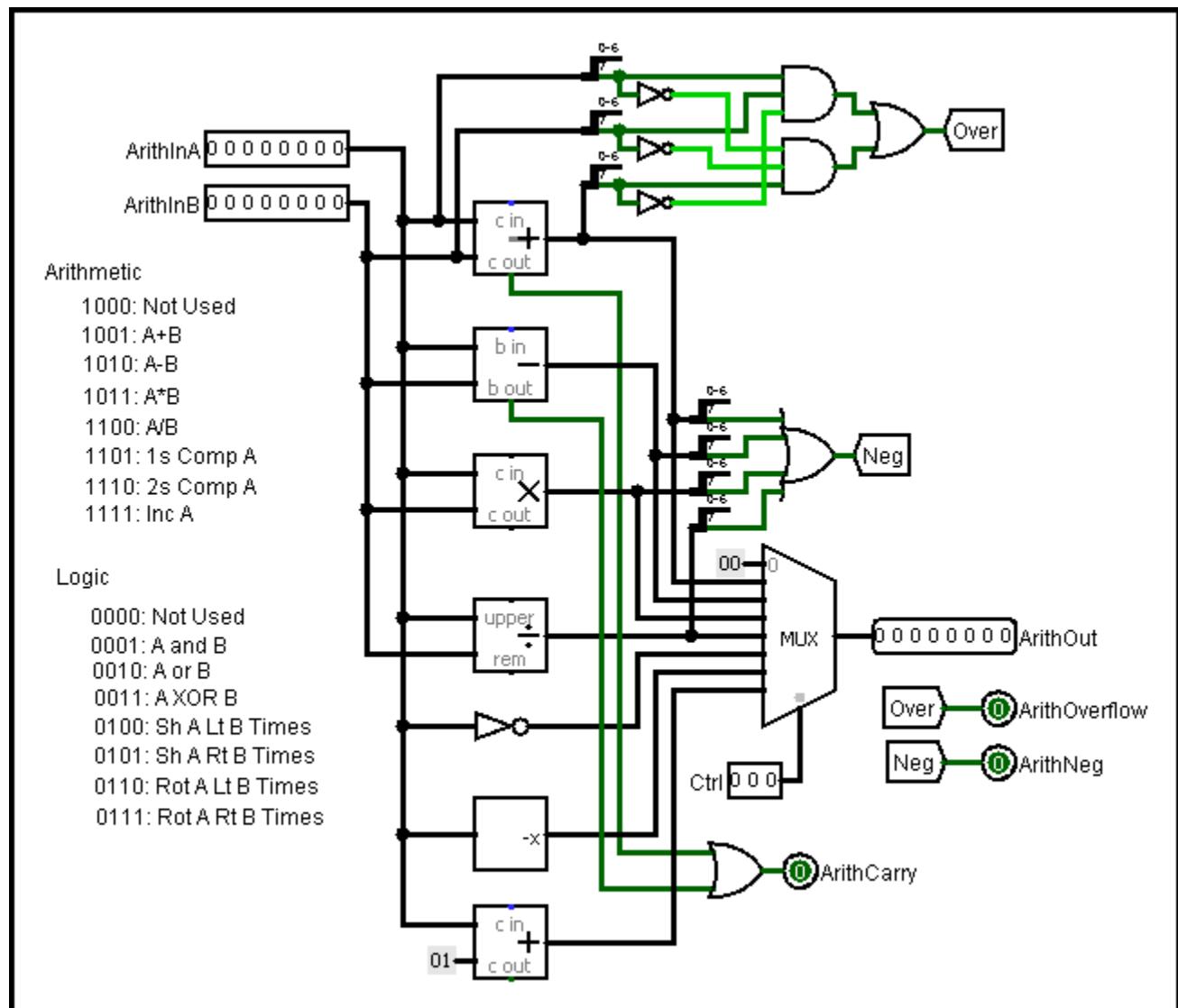


FIGURE 201: ARITHMETIC PART OF THE ARITHMETIC LOGIC UNIT (ALU) FOR LOGISIM CPU

The Arithmetic part of the ALU is very similar to the logic part, but there are a few additional outputs. The Overflow output is at the top right of the circuit. This checks the sign bit (bit 7, the most significant bit) for both inputs and the output. If the input bits are the same, but they are different from the output bit, then an overflow error has occurred and the "Over" flag will indicate that.

Just above the output multiplexor is an OR gate that checks the most significant bit of the output byte and if that bit is high then it generates a Negative flag.

Finally, near the right bottom of the circuit is a "Carry" bit that will go high if either the add or subtract function generates a carry.

Each of these flags are used in the CPU control component. As an example of how these flags can be used, imagine that an addition problem involving negative numbers (like  $-5 + -6$ ) is sent to this circuit.

The “Neg” flag will be set and the “overflow” flag will not be set; which would indicate a problem with a negative answer.

### ALU CONTROL

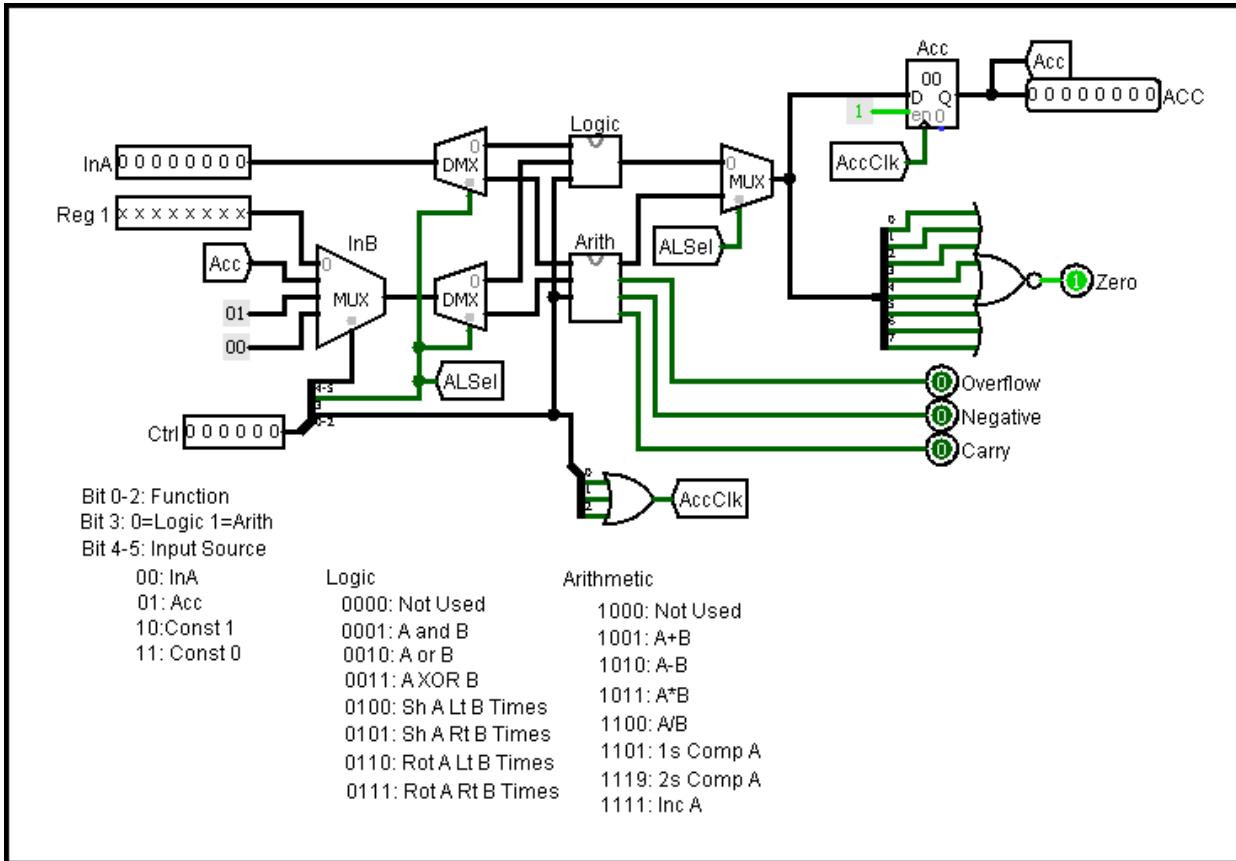


FIGURE 202: ARITHMETIC LOGIC UNIT (ALU) FOR LOGISIM CPU

There are two inputs at the left top of the ALU: InA is connected to the data bus and Reg 1 is connected to Register 01. In general, to manipulate two numbers (like add two numbers), one is placed in Register 1 and the other is placed on the data bus, and then the ALU is activated to "Add".

The four-input multiplexor labeled "InB" determines "Input B" for the arithmetic and logic functions. That input can originate at General Register 01 (as described in the previous paragraph), the Accumulator (which is the ALU output from a previous operation), a constant 1, or a constant 0. By selecting one of these inputs, the CPU designer can modify what values are being manipulated by the ALU. As a brief example, imagine that some number were placed on the data bus (thus, it is present at InA), and InB is set for a constant 01 input; then, if the ALU is set to add these numbers, the output would be InA incremented by one.

The two demux's located to the right of the inputs control whether the input values are sent to the arithmetic or logic sub-circuits.

At the top right of the circuit is the Accumulator, an 8-bit register. The Accumulator is designed to capture the output of the arithmetic or logic sub-circuits and hold that output for the data bus.

Notice that near the bottom center of the circuit is a three-input OR gate that creates a signal named "AccClk" (for "Accumulator Clock"). When any of the control lines go high, that OR gate activates and sends a high to the accumulator clock port. Since the accumulator activates on the falling edge of the clock, nothing happens until the control lines all go low and the AccClk signal goes low. This gives the arithmetic and logic circuits a chance to complete whatever operation was requested, and then their output is captured by the Accumulator when that clock signal goes low.

The ALU also passes the Overflow, Negative, and Carry flags from the arithmetic sub-circuits. Finally, the ALU creates a "Zero" flag if the Accumulator holds a value of zero. This is done by using an 8-input NOR gate connected to the eight bits of the accumulator. If any of those bits goes high then the NOR gate outputs a low; but if all eight inputs are low then the NOR gate outputs a high.

The ALU works through a 6-bit control signal. Bits 0-2 (the low order bits) determine the ALU's function; for example, arithmetic 001 is "Add" while logic 001 is "AND". Bit 3 determines if the arithmetic or logic sub-circuits are activated; for example, 0 indicates a logic function while 1 indicates an arithmetic function. Bits 4-5 (the high order bits) determine which of the "InB" inputs will be used. All of the possible control signals are listed on the various ALU figures above.

As an example, the Control 101001 chooses a constant one on InB ("10...."), the arithmetic sub-circuit (the "...1..." in bit 3), and an ADD function ("...001"). This setting would add one ("increment") to whatever value was present on input port A, and then put the result into the Accumulator.

### ADDRESS REGISTER (AR)

The Address Register (AR) is designed to hold the address of interest for the RAM memory. As an example, if address 00010001 needed to be loaded onto the data bus for some reason, the Address Register would contain 00010001. The value in the Address Register constantly changes as a program executes since the address of the currently executing instruction must be processed.

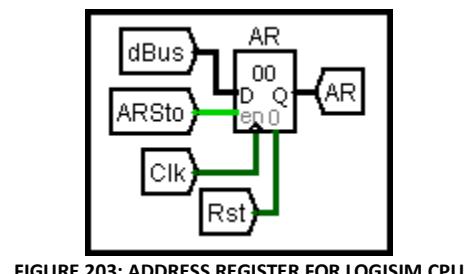


FIGURE 203: ADDRESS REGISTER FOR LOGISIM CPU

The Address Register is nothing more than simple register where the input is from the data bus and the output is to the RAM memory sub-circuit. Like all registers in this CPU, it is triggered by a falling clock edge.

### PERIPHERALS

This simulation includes only one peripheral: a TTY (for “Teletype”) screen. The TTY device accepts a 7-bit ASCII code from the data bus. Since the data bus is an 8-bit bus, the high order bit is discarded at the TTY input.

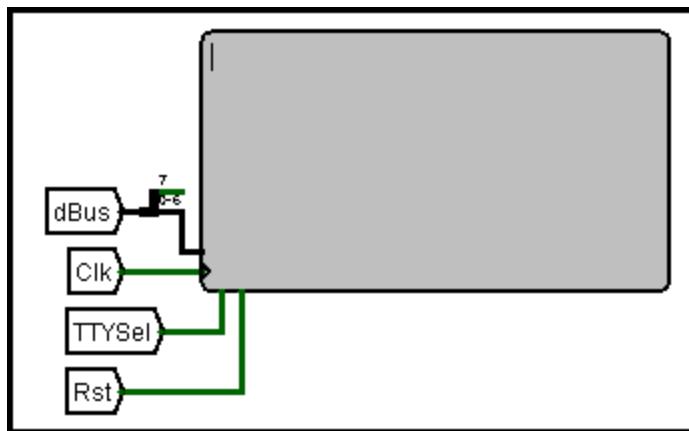


FIGURE 204: MEMORY-MAPPED PERIPHERAL (TTY) FOR LOGISIM CPU

The TTY device is "Memory Mapped" in this CPU; that is, anything stored in RAM address 0xFF (or 11111111) is sent to the TTY rather than memory. While there is only one peripheral in this simulation, it would be possible to have any number of peripherals, each mapped to a different RAM memory location.

## 7.10: CENTRAL PROCESSOR UNIT (CPU): CONTROL

### INTRODUCTION

A Central Processing Unit (CPU) is designed to use the bits in a binary word to turn on or off "gates" (like control buffers or demultiplexors) in the circuit. By doing that, data can be moved around the CPU and out to the various devices that may be present in the computer (like a hard drive or monitor). The binary numbers that are used to activate specific data paths are called *microcode* and they are an essential part of the CPU.

As an example, the following codes could be used to manipulate data in the simple CPU built in the Central Processor Unit (CPU): Fundamentals module:

- 11001111: Place the contents of Reg 0 on the data bus.
- 11001100: Place the Adder output on the ALU bus.
- 10000000: Store the contents of the ALU bus into Reg 2.
- 11100001: Place the contents of Reg 2 on the data bus, place the multiplier output on the ALU bus.
- 01100001: Store the contents of the ALU bus into Reg 1, place the contents of Reg 1 on the data bus.

Microcode is a very highly specialized form of programming. It is the code that is built into a CPU and it determines what gates, registers, and other devices are active for any specific program step. Each CPU would have a microcode built specifically for the architecture of that device. 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 1 to the contents of Register 2 and put the result in Register 0." The microcode must be able to activate and deactivate various controlled switches within the CPU so data appear on the appropriate bus at the right time in order to achieve the objective. Normally, microcode routines must be executed over a number of clock cycles in order to do a single job. For example, in one clock cycle the contents of Register 1 may be placed on the data bus, the next clock cycle will load that data into the ALU input register, and so forth until the entire process is complete. Microcode is normally contained in a ROM built into the CPU (though hard-wired versions are possible for simple CPUs). Microcode contained in ROM is typically called "firmware" since it is a string of ones and zeros, like software; but it cannot be changed, like hardware.

### LOGISIM 8-BIT CPU MICROCODE

The 8-bit Logisim CPU developed in this course uses a large number of control signals, and those are contained in a 28-bit microcode. Here is a breakdown of a single microcode instruction (these are listed in most significant to least significant bit order since that reads in a more natural left-to-right direction).

- Next Address (8 Bits): The next ROM address to execute. This permits the control circuit to step through a series of microcodes in order to complete a single instruction. Normally, the "Next Address" in the last step is 00000000, which loops back to the start of the microcode.

- IR Store (1 Bit): Activates the Store function for the Instruction Register. Instructions that are fetched from RAM are stored in the Instruction Register and then processed by the Control Circuit.
- AR Store (1 Bit): Activates the Store function for the Address Register. The address stored in this register determines the RAM location for memory store or load functions.
- Mem Store (1 Bit): Activates the Store function for the RAM. When active, whatever is on the data bus is stored in RAM at the address in the Address Register.
- Mem Load (1 Bit): Activates the Load function for the RAM. When active, whatever is stored in RAM at the address in the Address Register is loaded onto the data bus.
- PC Store (1 Bit): Activates the Store function for the Program Counter. This keeps track of the program being executed and contains the memory address for the next program instruction to process.
- PC Load (1 Bit): Activates the Load function for the Program Counter. The address of the next instruction to process is placed on the data bus.
- Gen Reg Load (1 Bit): Activates the Load function for the General Registers. The contents of one specific register are placed on the data bus.
- Gen Reg Store (1 Bit): Activates the Store function for the General Registers. The contents of the data bus are placed in one specified register.
- Gen Reg Ld Select (2 Bits): Selects which register will load data onto the data bus.
- Gen Reg Sto Select (2 Bits): Selects which register will store data from the data bus.
- ALU Control (6 Bits): Controls the function of the ALU. This is more fully defined in the module about the ALU.
- ALU Load (1 Bit): Loads the contents of the ALU Accumulator onto the data bus.
- Addr Select (1 Bit): Determines the address to be used for the next ROM step.

A single microcode step may look something like this (spaces were added for clarity):

00010101 0 0 0 1 0 0 0 1 00 10 000000 0 0

This code specifies 00010101 as the next microcode step to execute, load a RAM memory location onto the data bus, store the contents of the data bus in a register, select register 10 for that store operation. In essence, this one microcode copies a byte from memory to general register 03.

The first thing microcode needs to do is to fetch an instruction from RAM and then execute whatever that instruction requires. The following table contains the "Fetch" microcode. (Note: spaces have been added in the microcode to aid in interpretation. The actual code in ROM is a 28-bit word with no spaces.)

Addr	Code	Notes
00	00000001 0 1 0 0 0 1 0 0 00 00 000000 0 0	Store the Prog Cntr in the Addr Reg
01	00000010 1 0 0 1 0 0 0 0 00 00 000000 0 0	Store Mem Contents in Inst Reg
02	00000011 0 0 0 0 0 1 0 0 00 00 101001 0 0	Copy PC to ALU
03	00000100 0 0 0 0 1 0 0 0 00 00 000000 1 0	Store ALU Accumulator in PC
04	00000000 0 0 0 0 0 0 0 0 00 00 000000 0 1	Use "Next Addr" from currnt instr

TABLE 98: MICROCODE "FETCH"

Here is what each line does:

0. Load the number that is in the Program Counter (which contains the next RAM location to process) to the data bus. Store that number in the Address Register. Also, the first eight bits is the next microcode address to execute; and that number is 0000 0001.
1. Mem Load is high, so the content of the RAM address that was placed in the Addr Reg in the previous step is now loaded onto the data bus. That data contains the microcode for the next step to execute. At the same time that the data from RAM is being placed on the data bus, the Instruction Register is activated to store the microcode for later execution.
2. The PC Load bit is high, so the contents of the Program Counter Register is copied to the data bus. The ALU Control code is 101001, which is to input a constant 01 in InB and then add InA and InB; this effectively increments the contents of the program counter by 1.
3. Load the ALU Accumulator onto the data bus and then store the data bus in the Program Counter. Steps 02-03 increments the Program Counter by 1 so the next instruction can be fetched.
4. Within the Control Circuit, the next microcode step is either drawn from the first eight bits of the current microcode step; or a new address can be entered. The instruction codes are designed to actually be the microcode address for that function, so the instruction fetched from RAM can be stored in the microcode address register and it would cause the control circuit to jump to the step that handled that instruction. When "Address Select" is high, then the microcode counter jumps to the address contained in the fetched instruction.

#### CONTROL CIRCUIT

One of the main components of a CPU is the Control Circuit, which fetches a single instruction from RAM and then executes that instruction. In very broad brush strokes, this is how a Control Circuit works:

1. locates the next byte of the program being executed (this is a RAM address located in the Program Counter). That byte is an instruction for the CPU to execute, like "Move the contents of RAM location 1000 to the video card."

2. fetches that instruction; that is, it copies the instruction from RAM to the Instruction Register.
3. executes the instruction by executing all of the necessary microcode steps, in the proper order.
4. fetches and executes the program's next instruction, then the next, and so forth until the program ends.

It is important to keep in mind the difference between the instructions of a program and the steps contained in microcode. A single program instruction is interpreted and executed by the Control Circuit, using, perhaps, dozens of microcode steps. As an example, the program may have a single instruction to move a byte from RAM to the video card. The Control Circuit may process that instruction by first moving the byte to Register 01 and then moving it from there to the video card's input register. Those moves may require several clock cycles as various controlling devices and registers are activated in the correct sequence to move the data down the correct bus to its destination.

In summary, the Control Circuit controls all of the registers, buffers, buses, and other devices in the CPU in order to execute the instructions requested by the software program.

#### LOGISIM 8-BIT CPU CONTROL CIRCUIT

This is the Control Circuit for the 8-bit Logisim CPU being developed by this module.

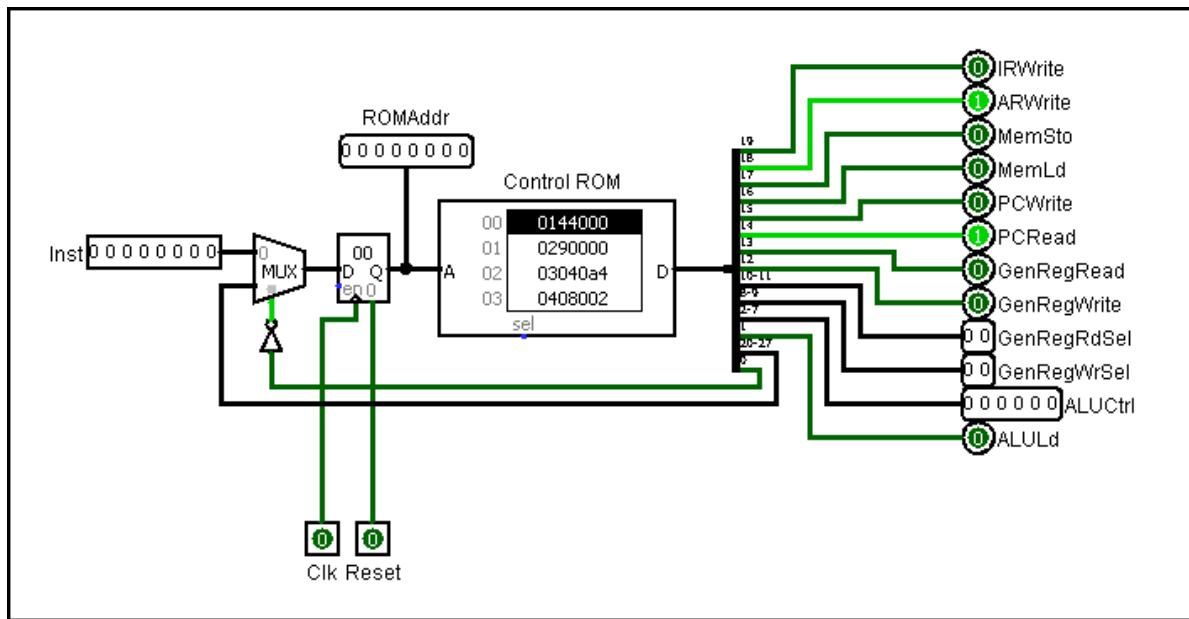


FIGURE 205: CONTROL CIRCUIT FOR LOGISIM CPU

An instruction's "OpCode" (the numeric code that stands for a given instruction) appears in the "Inst" input on the left side of the screen. The multiplexor then determines which opcode will be loaded into the ROM Address register. Normally, it would be the opcode found on the bottom leg, and that comes from the first 8 bits of the currently executing microcode step. By using this process, a program can move through a series of microcode steps, each step linked to the next by the first 8 bits in that microcode. However, at the end of any given microcode sequence, "Fetch" is called from microcode by entering the address 00000000 as the next instruction, and then bit 0 (the low order bit) is set high at

the end of the Fetch, which stores the microcode address from the instruction into the ROM Address register.

The Control ROM contains the microcode steps for various defined instructions. The large bank of outputs on the right side is linked to various gates and other controls in the CPU.

#### ASSEMBLY LANGUAGE

A computer program, as contained in software, is nothing more than a series of ones and zeros, organized into groups the size of the data bus (typically 16 or 32 bits). Each group of bits is a single instruction. When viewed at the level of ones and zeros, a program is said to be in "machine code," and would look something like this for the 8-bit CPU being developed in this class:

```
00101110  
00000110  
00010111
```

If a programmer could master machine code, then those programs would be as concise and efficient as possible since they would be written in the machine code the CPU can execute. 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 (called "mnemonics") to represent the available CPU instructions (or "opcodes") available. An assembly language program for the 8-bit CPU being developed in this class looks something like this:

```
LDI AA  
X01  
LDI 55  
ADD  
HLT
```

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 simple program that does little more than convert a file containing assembly code mnemonics into CPU instructions that can be executed by the CPU.

The Assembly program written above would compile into this:

```
00000110  
10101010  
00001011  
00000110  
01010101  
00101110  
00000101
```

It is important to keep in mind that assembly code is CPU specific, code written for an Intel 8088 CPU cannot be used on any other type of computer.

#### PROGRAMMING LANGUAGES

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; they can also normally be used by many different CPUs. 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 to a series of instructions which the CPU can execute.

### 7.11: CENTRAL PROCESSOR UNIT (CPU): FINALIZATION

An 8-Bit Logisim CPU has been developed in three other units: Fundamentals, Components, and Control; and in this unit the various sub-circuits are combined on a single "Dashboard" display.

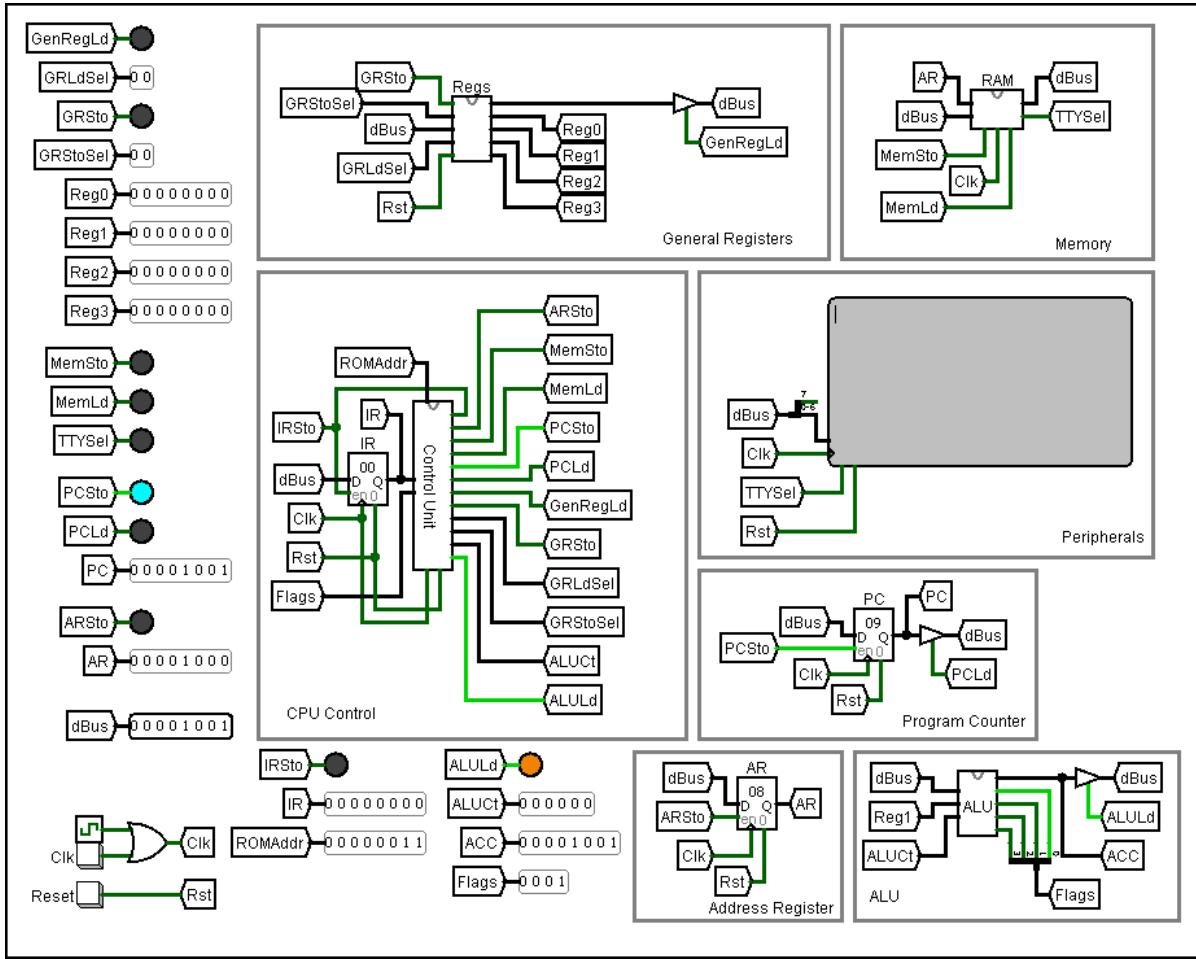


FIGURE 206: 8-BIT LOGISIM CPU

The various components are evident across the dashboard, and all are connected by Logisim "Tunnels" that carry the various signals from the Control Circuit. The CPU's clock is in the lower left corner and it is designed to transmit either a clock signal from Logisim or use a single-step push button. There is also a reset in the lower left corner so the CPU can be placed back into a beginning state.

To operate the CPU, a program is loaded into RAM (in the RAM component), and then it is executed by either starting the Logisim clock or by single-stepping through the program by using the push button. Keep in mind that every step of microcode must be executed individually, so a single program instruction may require a number of clock ticks.

## 7.12: LAB – VENDING MACHINE

### PURPOSE

The purpose of this lab is to build a soft drink vending machine simulator. Here are the specifications:

1. The user can input the following coins: 5 cents, 10 cents, 25 cents
2. When 75 cents is input, the machine will activate the soft drink vendor
3. When 75 cents is input, no more coins will be accepted
4. Change will be returned to the user if more than 75 cents was deposited
5. A reset button will return the user's money
6. When a soft drink is dispensed, the 75 cents deposited will be added to the machine's bank
7. No soft drink is dispensed if less than 75 cents is deposited
8. The current number of soft drinks available will be stored in a counter
9. If the number of soft drinks available is zero for any one brand the machine will light a "sold out" light if that brand is selected, but take no other action.

### PROCEDURE

#### COIN SORTER

The coin sorter accepts any of the following three coins: 5 cents, 10 cents, and 25 cents. The amount of the coin is sent to the Bank Circuit.

1. Create a sub-circuit of the main circuit. Name the sub-circuit *CoinSorter*.
2. The Coin Sorter should have an East-facing MUX with 8 data bits and 2 select bits.
3. The MUX select bits come from the output of a Priority Encoder (*Plexers* library). The encoder should have 2 select bits.
4. Three Priority Encoder inputs on pins 0-2 should be linked to input pins labeled "5c", "10c", and "25c". A fourth input pin should be linked to the Priority Encoder's Enable pin (on the south).
5. The MUX has 3 8-bit inputs on pins 0-2. These should be tied to an 8-to-1 Splitter (*Base* library).
6. Input pin 0 on the MUX should be tied to a splitter with the inputs 00000101.
7. Input pin 1 on the MUX should be tied to a splitter with the inputs 00001010.
8. Input pin 2 on the MUX should be tied to a splitter with the inputs 00011001.
9. The output of the MUX should be sent to an 8-bit output pin labeled "AmtOut".

Following is a diagram of the complete *CoinSorter* sub-circuit.

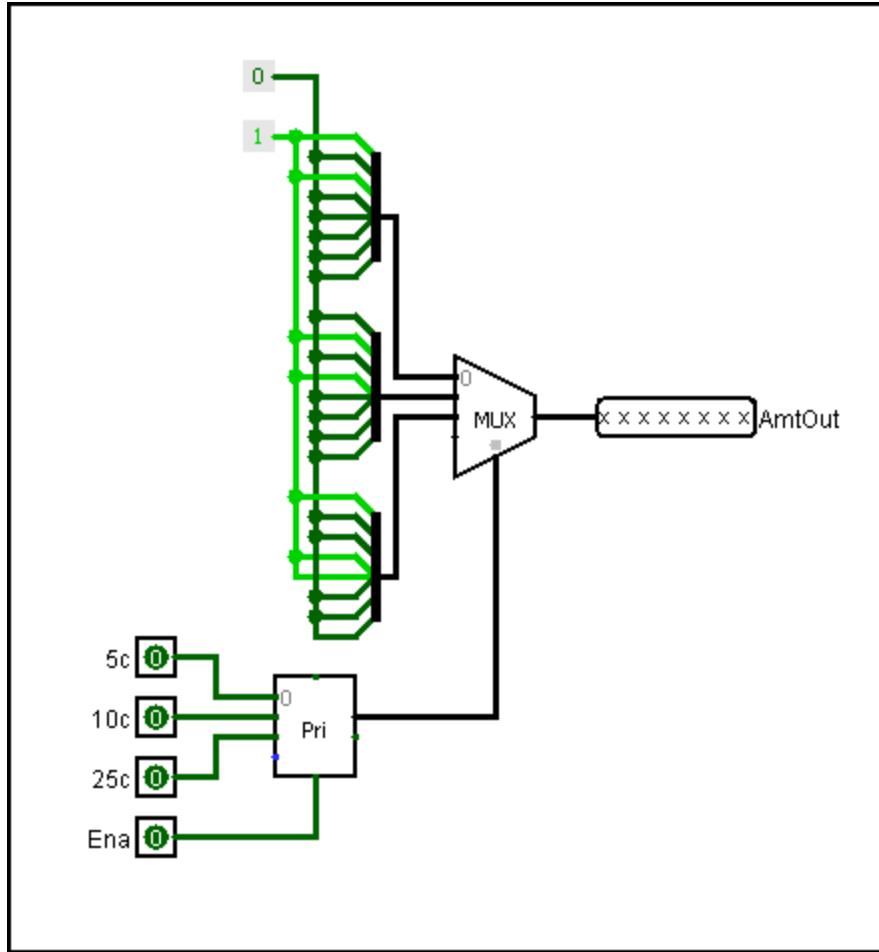


FIGURE 207: COIN SORTER

Place an instance of the *CoinSorter* sub-circuit on the left side of the main circuit.

Wire the *CoinSorter* inputs to buttons and label those buttons 5, 10, and 25. The main circuit should look like this:

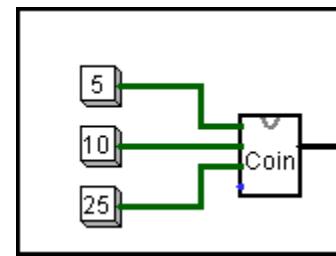


FIGURE 208: MAIN CIRCUIT WITH COIN SORTER

#### BINARY TO BCD

The vending machine will use binary for all mathematics applications (adding the amount of money collected, for example). However, in order to display the results of mathematics operations for people to easily read, the binary numbers will be converted to Binary Coded Decimal (BCD). An 8-Bit Binary to

BCD had already been developed in another lab. It includes both a main and one sub-circuit. Both of those should be copied into this vending machine lab.

#### BANK

The Bank Circuit is used to tally the amount of money deposited in the Coin Sorter. This circuit also sends an *Activate* signal to permit product to be dispensed when 75 cents has been deposited. The Bank Circuit also calculates the change due if someone deposits more than 75 cents. This is the most complex sub-circuit in the entire project.

#### TOTALING DEPOSITS

Start by creating the *bank* sub-circuit.

1. Place an 8-bit input pin and label it *CoinIn*.
2. Wire *CoinIn* to one leg of an adder.
3. Wire the output of the adder to the D input of an 8-bit register. ***Important Note: Set the Trigger attribute for this register to Falling Edge.***
4. Wire a constant 1 input to the enable port of the register.
5. Wire an input pin labeled "Clk" to the clock port of the register.
6. Loop a wire from the register output back to the second leg of the adder placed in Step 2.
7. Also wire the output of the register to a Bin2BCD circuit created earlier in this lab. The three outputs of the Bin2BCD circuit should be sent to three 4-bit ports labeled *DepOut0*, *DepOut1*, and *DepOut2*.
8. Following is the circuit you should have at this point.

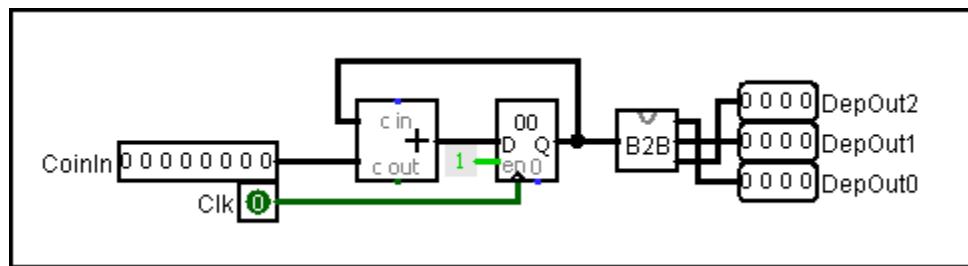


FIGURE 209: BANK VERSION 1

#### SEND DATA TO ACTIVATE DISPENSER

Send a signal that will activate the product dispenser once 75 cents has been deposited.

Wire the output of the bank's register to an 8-bit port named *AccOut*. In another sub-circuit the number from the register will be compared to 75 cents and if it is greater, then it will activate the product dispenser.

#### CALCULATE CHANGE

The basic concept for making change is that the total amount deposited will be subtracted from  $75_{10}$ . If the difference is zero then no money will be refunded. If the difference is greater than zero, then that difference will be refunded.

Place a subtractor under the existing circuit but in a position so that one input is taken from the register's output. The other input of the subtractor should have 75 permanently wired using a splitter and constant 1's and 0's.

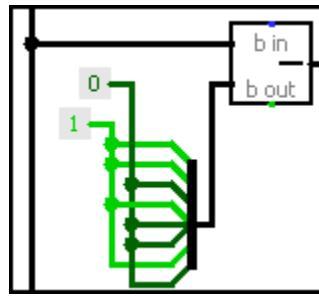


FIGURE 210: CHANGE CALCULATING SUBTRACTOR

The output of the subtractor should be wired to one of the inputs of a comparator. The other comparator input should be wired to a constant 0.

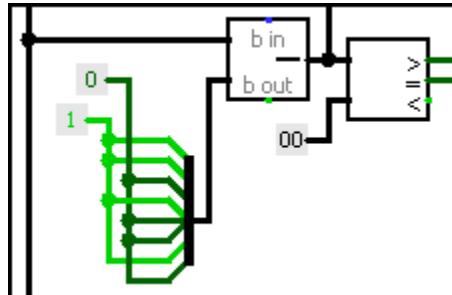


FIGURE 211: COMPARATOR USED FOR CALCULATING CHANGE

The subtractor output should also be wired to a Bin2BCD circuit that will display the amount of change returned. The output of the Bin2BCD circuit should be sent to three different 8-bit ports labeled *ChOut0*, *ChOut1*, and *ChOut2*.

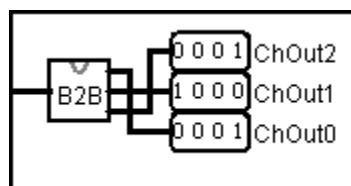


FIGURE 212: CREATING BCD OUTPUT FOR CHANGE

The change circuit now functions, but it should only display change when the machine vends a product. There also needs to be some way to reset the machine customers change their minds and want their money back. For the moment, leave this sub-circuit and create the vend circuits; then this circuit will be completed later.

#### ACTIVATE DISPENSER

The Activate Dispenser circuit sends a signal to activate the dispenser so customers can select a product. This is a very simple circuit that compares the amount of money deposited to 75 and enables an *Activate* signal when a proper amount has been deposited.

Wire an 8-bit input pin labeled *In* to a comparator. Wire the second comparator input to a constant 75 by using a splitter. If the value of *In* is greater than or equal to 75, then send a 1 to a 1-bit output pin called *Activate*.

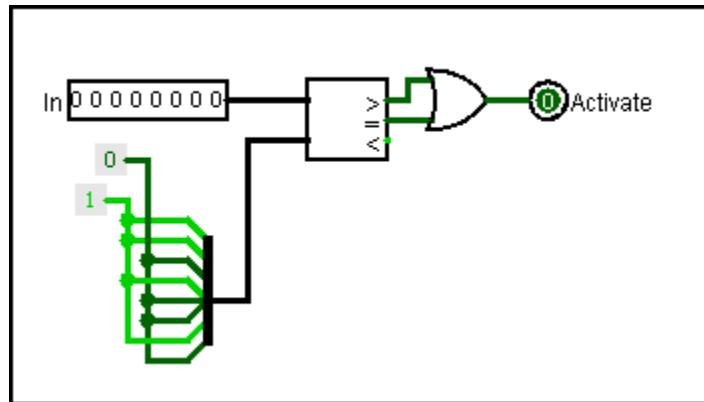


FIGURE 213: ACTIVATE DISPENSER

#### PRODUCT DISPENSER

The product dispenser circuit has three functions:

1. dispense a product when the *Activate* signal is present.
2. keep count of the product remaining and stay inactive when the product is sold out.
3. send a signal to the bank when a product is dispensed so change can be returned to customers.

This is the product dispenser circuit:

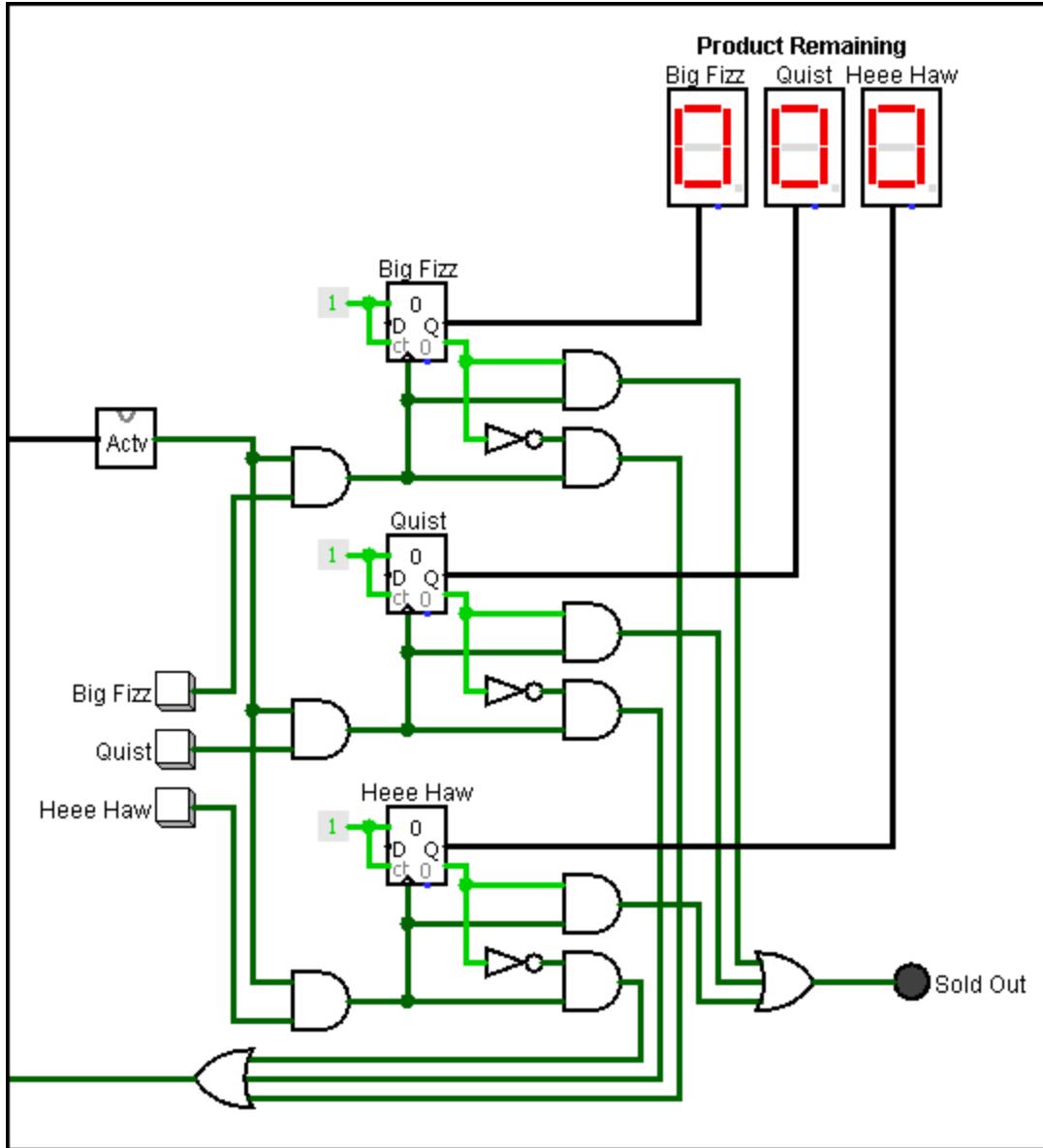


FIGURE 214: PRODUCT DISPENSER

To understand what is going on in this circuit, it is easiest to look at only one product since "Big Fizz," "Quist," and "Heeee Haw" use identical circuits. For this lab, consider only the "Big Fizz" product.

1. The *Activate* signal goes into one leg of an AND gate. The other leg is wired to the "Big Fizz" button on the machine. When there is an *Activate* signal (that is, enough money was deposited), and the customer presses the "Big Fizz" button, then that AND gate is activated.
2. The output of the *Activate* AND gate first clocks the "Big Fizz" counter. That counter is set to subtract one on every clock pulse. Output Q of that counter is sent to a Hex Display so service technicians can see how many cans of product are still in the machine.
3. The output of the *Activate* AND gate also goes to two other AND gates. The top gate controls the *Sold Out* circuit. When that AND gate receives a *Carry* signal from the counter (this indicates

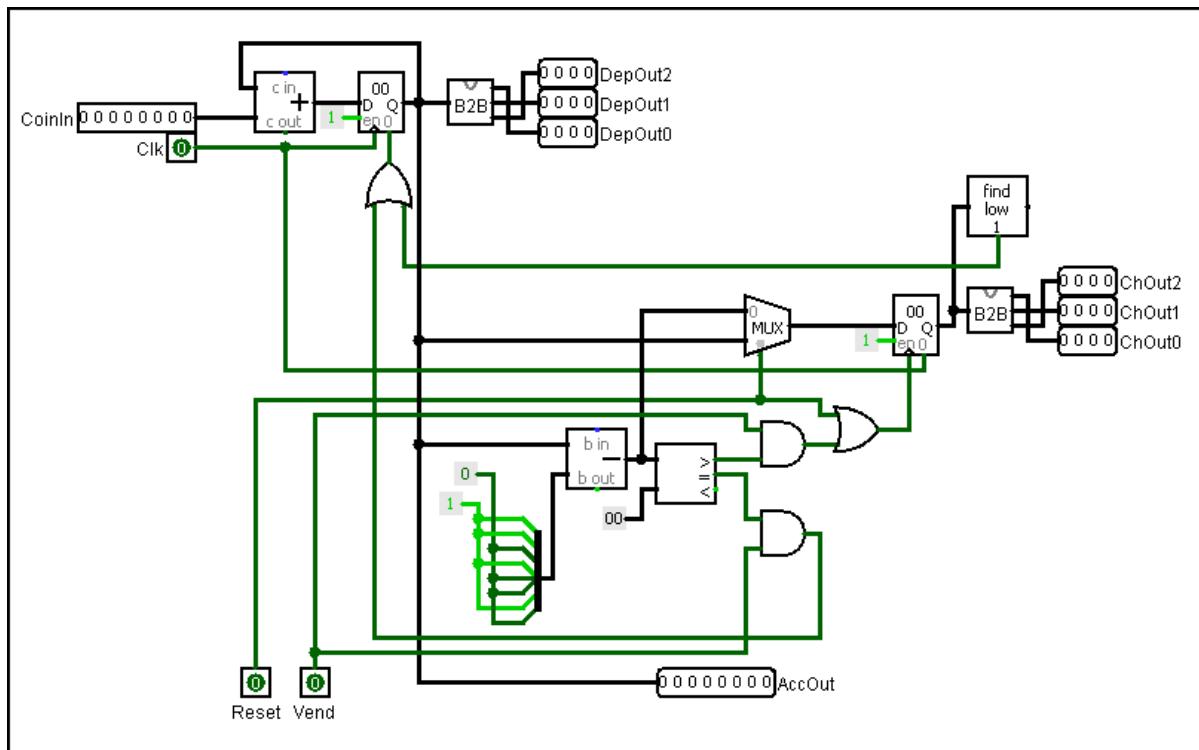
the counter has reached 0) and the customer presses the "Big Fizz" button, it will activate the "Sold Out" LED.

4. The output of the Activate AND gate also goes to two other AND gates. The bottom gate generates a signal that is sent through an OR gate back to the bank. That signal returns change to the customer and resets the circuit so another product can be purchased. The bottom AND gate also receives a signal from the "Big Fizz" counter. When the counter reaches zero, the *Carry* output goes high, but that signal is inverted so the bottom AND gate is disabled. Thus, when there is no product, no signal ever goes to the bank to complete the transaction.

## BANK REVISITED

Now that the vending machine includes the logic needed to dispense product, the bank circuit needs to be completed. What is remaining is a way to actually dispense change and then reset the machine for the next purchase. Also, logic needs to be added so if customers change their minds, all deposited money can be returned.

When the bank circuit was created earlier in this lab, the change was simply constantly displayed. However, change should only be returned to the customer if a purchase was made. The *product purchased* signal from the dispenser circuit will let the bank know that a product was purchased and the transaction needs to be completed. *Important Note:* when completing the *Bank* circuit as illustrated below, the *Trigger* attribute for both registers must be set for *Falling Edge*.



**FIGURE 215: COMPLETE BANK CIRCUIT**

1. The bank has two new inputs located in the lower left corner of the circuit. *Reset* will simply be connected to a button in the main circuit so customers can ask for their money back.

2. The *Vend* input is connected to the signal from the dispenser that indicates product was actually dispensed. That signal is used to activate two AND gates.
  - One AND gate indicates that the amount of money deposited was equal to 75 cents, and the output of that gate is used to reset the bank's register to 0 so another product can be dispensed.
  - The second AND gate is used to activate the change circuit so the correct change can be returned and then a reset signal sent to the bank's register.
3. The Multiplexer is used to determine how much money will be returned to the customer. If a product is vended, then only the change is returned. If the customer pressed the *Reset* button, then the entire amount deposited is returned.
4. The *Find Low 1* device looks for the lowest-order 1 in an 8-bit number. The amount returned is fed into that device. If any “one” is found in that 8-bit number (that is, the amount returned is not equal to 0), then a signal is generated that is used to reset the bank's register.

#### FINAL STEPS

Most of the logic and various circuits are now present. All that remains is to add a couple of other signals and *prettyify* the main circuit.

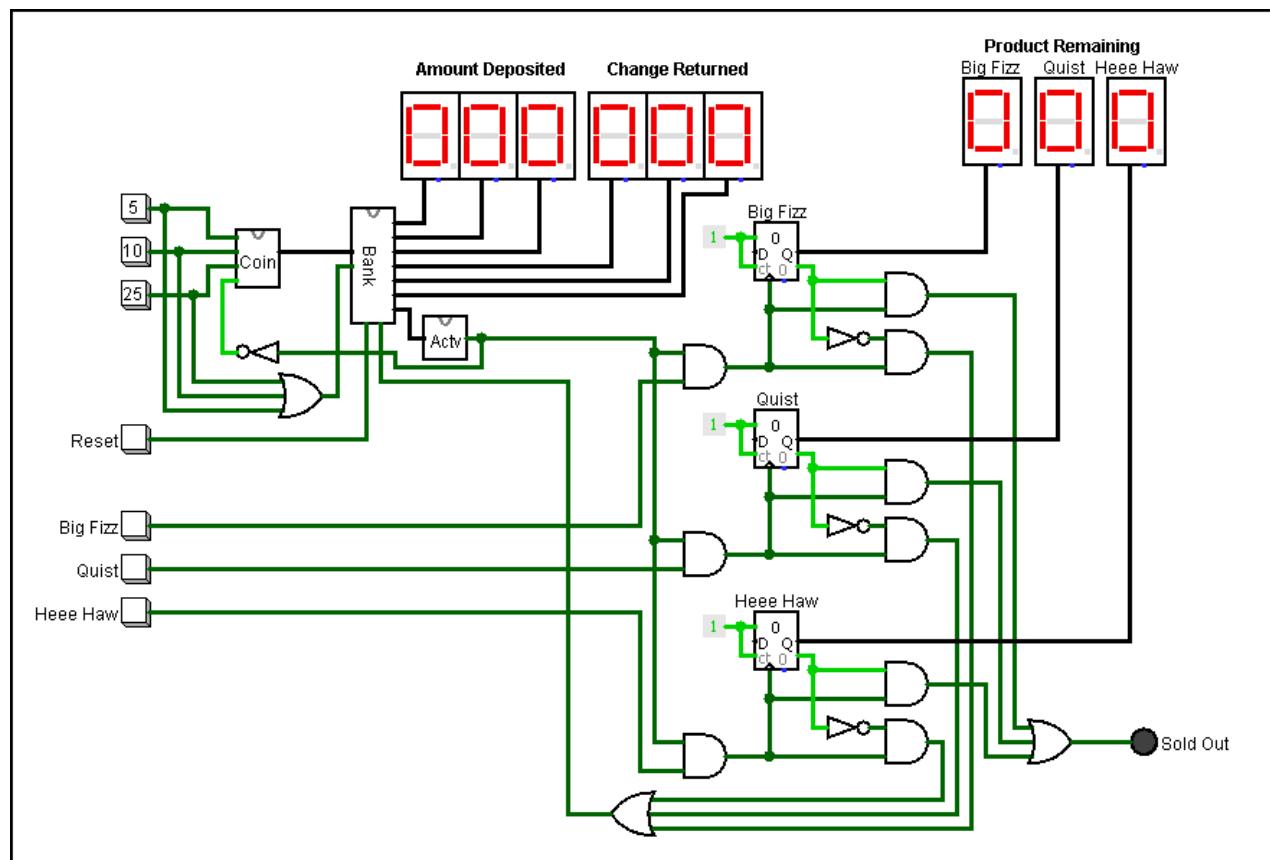


FIGURE 216: VENDING MACHINE MAIN CIRCUIT

1. Run an inverse of the *Activate Dispenser* signal back to the coin sorter. That should be fed to the enable pin of the Priority Encoder in the coin sorter. By doing this, when 75 cents has been deposited, the coin sorter will be inactivated so no more money can be deposited.
2. Each of the three coin buttons is wired to a three-input OR gate and that is wired to the *Clk* input of the bank. By doing that, every time a coin is deposited, the bank will activate and count that coin.
3. Everything else is set up so the inputs are on the left and the outputs are on the right.

#### CHALLENGE

Add a "Total Money Collected" display so a service technician can tell how much money was deposited. This circuit should only count the money collected, not the change returned. There should also be a way to reset that total so the service technician can start the count a zero after servicing the machine.

#### CLEANUP

Rename the *main* circuit to *Vend*. Be sure the standard identifying information block is at the top left of the *Vend* circuit: Name, "Lab 7.12: Vending Machine", and today's date. Save the file as *Lab 7\_12 – Vending Machine*.

### 7.13: LAB – ELEVATOR

#### PURPOSE

In this lab you will build a circuit that will simulate an elevator.

#### PROCEDURE

This lab is different from all of the others we have done in this class. Rather than having step-by-step directions, this document will simply specify the requirements and you will be on your own to design the circuit.

Build an elevator simulator.

1. Your 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 floor 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).
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.

As an example, here is one student’s version of an elevator simulator from a previous semester:

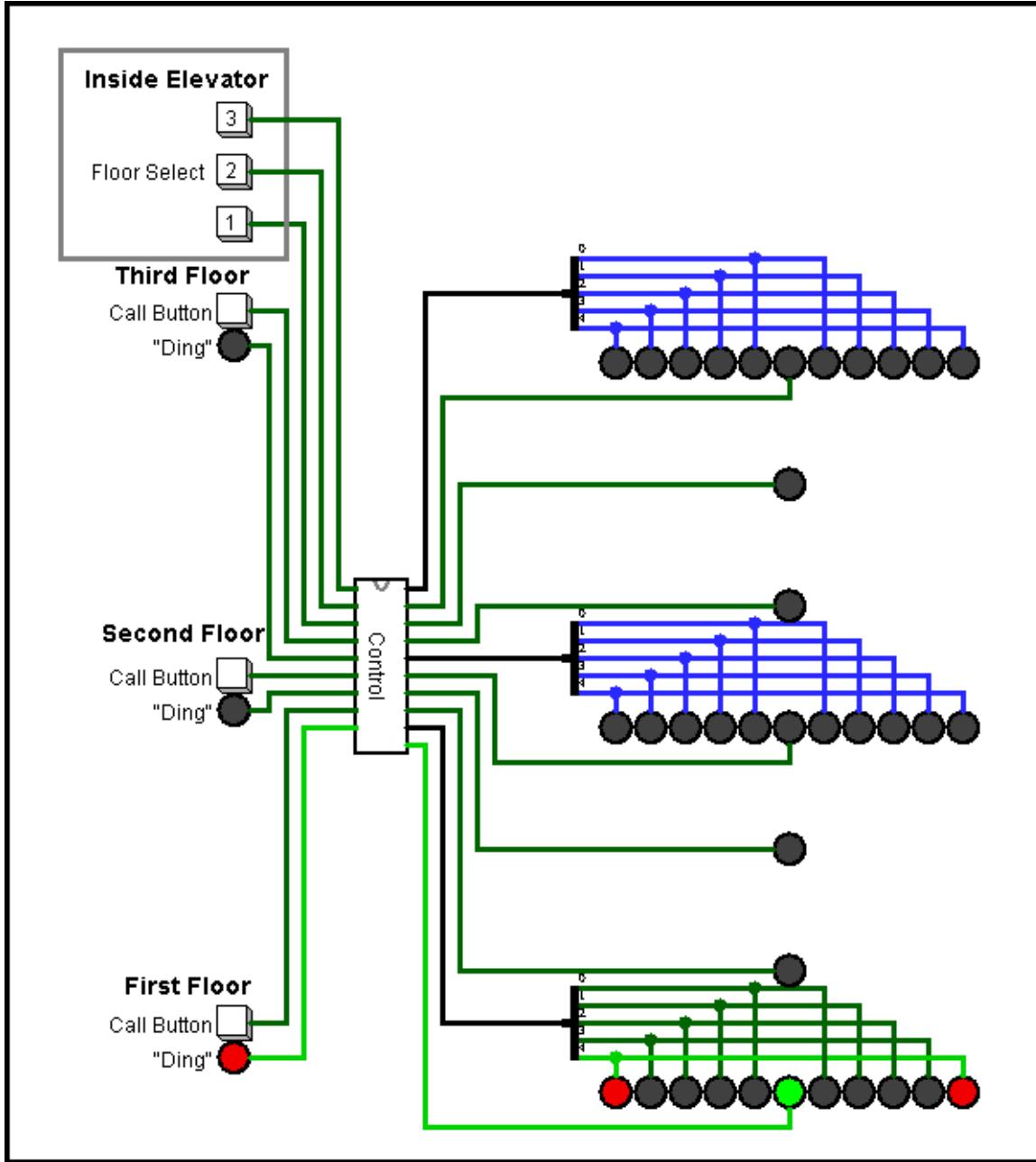


FIGURE 217: ELEVATOR CIRCUIT

#### CLEANUP

Rename the *main* circuit to *Elevator*. Be sure the standard identifying information block is at the top left of the *Elevator* circuit: Name, "Lab 7.13: Elevator", and today's date. Save the file as *Lab 7\_13 – Elevator*.



---

## APP A: INTEGRATED CIRCUIT PART NUMBERS

### INTEGRATED CIRCUIT PART NUMBERS

<sup>8</sup>The part numbers for 7400 series logic devices often use the following naming convention, though specifics vary between manufacturers.

1. First, although sometimes omitted, a two or three letter prefix which indicates the manufacturer of the device (e.g. SN for Texas Instruments, DM for National Semiconductor) although these codes are no longer closely associated with a single manufacturer, for example Fairchild Semiconductor manufactures parts with MM and DM prefixes, and none.
2. A two-figure secondary prefix, of which the two most common are "74", indicating a commercial temperature range device and "54", indicating an extended (military) temperature range
3. Up to four letters describing the logic subfamily, as listed below (e.g. "LS" or "HCT"). Frequently, an "x" is used to indicate a generic part that could be one of several specific versions. As an example, "74x299" could mean 74LS299 or 74VHC299.
4. Two or more digits assigned for each device, e.g. 00 for a quad 2-input NAND gate. There are hundreds of different devices in each family.
5. Additional suffix letters and numbers may be attached to indicate the package type, quality grade, or other information, but this varies widely by manufacturer.

For example SN74ALS245N means this is a device probably made by Texas Instruments (SN), it is a commercial temperature range TTL device (74), it is a member of the "advanced low-power Schottky" family (ALS), and it is a bi-directional eight-bit buffer (245) in a plastic through-hole DIP package (N).

In this Digital Logic course, integrated circuits are identified in the simplest way possible, using a nomenclature similar to "74x245" to represent any sort of bi-directional eight-bit buffer device.

### MANUFACTURER PREFIXES

Here are some of the more common manufacturer Prefixes:

- AD - Analog Devices
- Am - AMD
- Cx - Cyrix
- DM - National Semiconductor
- HD - Hitachi
- L1,L - LSI
- Max - Maxim
- MC - Motorola

---

<sup>8</sup> Information in this paragraph found at  
[http://en.wikipedia.org/wiki/7400\\_series#Part\\_numbering\\_scheme](http://en.wikipedia.org/wiki/7400_series#Part_numbering_scheme)

- MM - Fairchild
- Nx - NexGen
- SCN - Signetics
- SN,TX,TMS - Texas instruments
- SY - Synertek
- T - TI
- T - Toshiba
- WD - Western Design Center
- Z - Zilog

## LOGIC SUBFAMILIES

- ABT - Advanced BiCMOS, TTL-compatible input thresholds, faster than ACT and BCT
- AC - Advanced CMOS, performance generally between S and F
- AHC - Advanced High-Speed CMOS, three times as fast as HC
- ALS - Advanced Low Power Schottky
- ALVC - Low voltage - 1.65 to 3.3 V
- AS - Advanced Schottky
- AUC - Low voltage - 0.8 to 2.7 V
- BCT - BiCMOS, TTL-compatible input thresholds, used for buffers
- C - CMOS 4–15 V operation
- F - Fast (faster than normal Schottky, similar to AS)
- FC - Fast CMOS, performance similar to F
- H - High speed (still produced but generally superseded by the S-series, used in 1970s era computers)
- HC - High speed CMOS, similar performance to LS
- HCT - High speed, compatible logic levels to bipolar parts
- L - Low power (compared to the original TTL logic family), very slow
- LCX - CMOS with 3 V supply and 5 V tolerant inputs
- LS - Low Power Schottky
- LVC - Low voltage – 1.65 to 3.3 V and 5 V tolerant inputs
- LVQ - Low voltage - 3.3 V
- LVX - Low voltage - 3.3 V with 5 V tolerant inputs
- No Letter (obsolete) - the "standard TTL" logic family had no letters between the "74" and the specific part number, like "7400".
- S - Schottky (obsolete)
- VHC - Very High Speed CMOS - 'S' performance in CMOS technology and power
- X - Indicates a generic part, for example 74x299.

#### 4000 SERIES CMOS

This family of logic ICs is numbered from 4000 onwards and they usually have a B at the end of the number, like "4001B", which refers to an improved design introduced some years ago. They use CMOS circuitry which means they use very little power and can tolerate a wide range of power supply voltages (3 to 15V) making them ideal for battery powered projects. Other than starting with "40," these devices use the same numbering scheme as for the "74" series.



---

## APP B: REFERENCES

### BOOKS

Gregg, John. *Ones and Zeros*, 1<sup>st</sup> ed. New York: IEEE Press, 1998. Print. 0-7803-3426-4

Holdsworth, Brian. *Digital Logic Design*. 4th ed. Oxford: Newnes, 2006. Print. 978-0750645829

Langholz, Gideon, Abraham Kandel, and Joe Mott. *Foundations of Digital Logic Design*. 1st ed. Singapore: World Scientific Publishing Co, 1998. Print. 981-02-2110-5

Mano, Morris. *Digital Design with an Introduction to the Verilog HDL*. 5th ed. Upper Saddle River: Pearson, 2007. Print. 978-0-13-277420-8

Rafiquazzaman, Mohamed. *Fundamentals of Digital Logic and Design*. 5th ed. Hoboken: John Wiley & Sons, Inc, 2005. Print. 0-471-72784-9

Saha, A, and N Manna. *Digital Principles and Logic Design*. 1st ed. Hingham, MA: Infinity Science Press LLC, 2007. Print. 978-1-934015-03-2

Yarbrough, John. *Digital Logic Applications and Design*. 1st ed. Boston: PWS Publishing Company, 1997. Print. 0-314-06675-6

### WEB SITES

*Digital Electronics*. This is a Wikibook that includes basic information about digital electronics. The information is fairly limited and includes only summary and brief reference material.

([http://en.wikibooks.org/wiki/Digital\\_Electronics](http://en.wikibooks.org/wiki/Digital_Electronics))

*Digital Circuits*. This is a Wikibook that is designed as a reference for digital circuits. Unfortunately, the material is somewhat limited in scope and may serve as a quick reference rather than a tutorial.

([http://en.wikibooks.org/wiki/Digital\\_Circuits](http://en.wikibooks.org/wiki/Digital_Circuits))

*Digital Logic*. This is an extensive site that includes electronics diagrams and other information about electronic circuits. It was written as a resource for people who enjoy building electronic devices as a hobby. (<http://www.play-hookey.com/>)

*Digital Lessons*. This is an excellent text covering digital logic. The author explains various digital logic topics in a clear manor that is easy to understand.

(<http://www.openbookproject.net//electricCircuits/Digital/index.html>)

*Mark Knows Nothing*. This page has an excellent calculator for converting between bases and permits fractional numbers. It is quick and easy to use and will help when converting between decimal/binary/hex. (<http://markknowsnnothing.com/cgi-bin/baseconv.php>)

*Yellowpipe Encrypter/Decrypter.* This page is a tool that will encode/decode between a number of different code systems. It is especially useful for ASCII<->Hex code exchange for this class.  
(<http://www.yellowpipe.com/yis/tools/encrypter/index.php>)