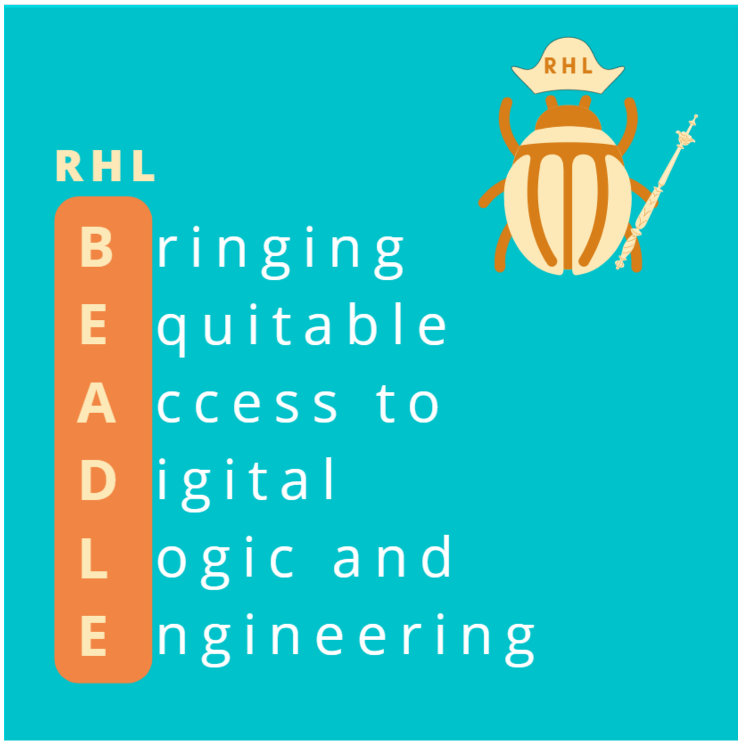
ELECTRICAL & COMPUTER ENGINEERING

The RHL-BEADLE Project

Version 1.0

An Introductory Digital Logic Curriculum Using Remotely Accessible FPGA Lab



RHL-BEADLE Team:

Authors:

Prof. Rania Hussein, University of Washington

Florence Atienza, University of Washington

Testers:

Justine Bailey, Intel Corporation

Cinthya Rosales, Intel Corporation

Kaylee Lam, University of Washington

Developers:

Pablo Orduna, LabsLand

Luis Rodriguez-Gil, LabsLand

ELECTRICAL & COMPUTER
ENGINEERING

Lab 0: GETTING STARTED

The University of Washington | The Remote Hub Lab | Last Revised: March 2022

Summary

This lab helps students install and become familiar with the technology tools they need to complete the Remote Hub Lab's introductory digital logic lab series. Material includes making an account on the LabsLand platform and downloading Quartus and ModelSim.

Table of Contents

About the LabsLand Platform

Description

LabsLand is an online platform where students around the world can access remote hardware laboratories. For the purposes of this introductory digital logic lab series from the Remote Hub Lab (RHLab), LabsLand provides virtual access to real remote hardware through the University of Washington's Electrical and Computer Engineering department.

For Instructors

1. Go to the following link: https://labsland.com/web/intel/community-colleges.
2. Follow the instruction prompts on your screen to create a group for your students.

For Students

1. Wait for a link from your instructor to join your LabsLand group. **DO NOT ACCESS THE LINK ABOVE, WHICH IS FOR INSTRUCTORS ONLY.**
2. When you are able to login using the link from your instructor, you should be able to see a dashboard which includes three LabsLand laboratories: 1) Digital Trainer, 2) Boole, and 3) Intel DE1_SoC.

Downloading Quartus and ModelSim

Getting the Quartus and ModelSim Software

The designs and simulations in this lab series will be done through Quartus and ModelSim software, respectively. We will use these tools beginning in Lab 4.

If you are using a Windows machine, you can install the software for free by following these steps:

1. Go to https://fpgasoftware.intel.com/?edition=lite.
2. Download the free web edition and install it. Note that you will have to register to be able to download the software.
    a. Make sure you Select Edition “Lite” and Select Release “17.0”.
    b. Download the 5.8 GB tar file under the “Combined files” tab.
    c. Extract the tar file using a program like 7-zip.
    d. Run the QuartusLiteSetup-17.0-windows.exe file.
    e. When it asks for the components to install, make sure you select each of these:
        - Quartus Prime Lite Edition (Free)
        - Devices: Cyclone V
        - ModelSim: Intel FPGA Starter Edition (Free)
3. When the software is done downloading, make sure to install the USB blaster driver.
4. Run Quartus next, and if asked about licensing just run the software (we use the free version, so no license required).

In the case that you use a Mac, you can install a virtual machine (which allows you to run Windows on a Mac) and proceed with the steps 1-4 above.

Creating Your First Quartus Project

Follow the instructions in this video tutorial to set up and create your first Quartus project and directory for future use in Lab 4: https://youtu.be/iLbmSTG7bpA.

Lab 1: INTRODUCTION TO GATES AND DIGITAL LOGIC

The University of Washington | The Remote Hub Lab | Last Revised: March 2022

Summary

The goal of this lab is to build a foundational understanding of basic logic operators and logic gates. The lab introduces the AND, OR, NOR, XOR, and NAND operators, their corresponding gates, and the functions of a MUX.

Table of Contents

Why Should We Study Digital Logic?

Electronics are powered by electricity and the ability of electricity to be conceptualized into two states: an “off” or an “on” state. Usually, a zero (0) represents the off state (also known as the “false” state), and a one (1) represents the on state (also known as the “true” state).

How can only two representations -- a binary representation -- lead to complex technology? How can electronic devices (phones, computers, televisions, etc.) translate zeros and ones to information and applications? Under their cases, the devices contain fundamental circuits which rely on the basic components of digital logic. As a result, studying these building blocks is a key step in understanding how these devices work.

Before analyzing the zeros and ones of digital logic, it is important to understand the concept of inputs vs. outputs. A simple way to visualize this is to first view the inner workings of electronics as a “mystery box” system. We will understand the details later in this curriculum. For now, the key points are the following:

- First, some information enters the box as inputs.
- Then, some transformation happens inside the mystery box system.
- Finally, the changed information exits the system as output.

This input/output (I/O) relationship is shown in Figure 1.

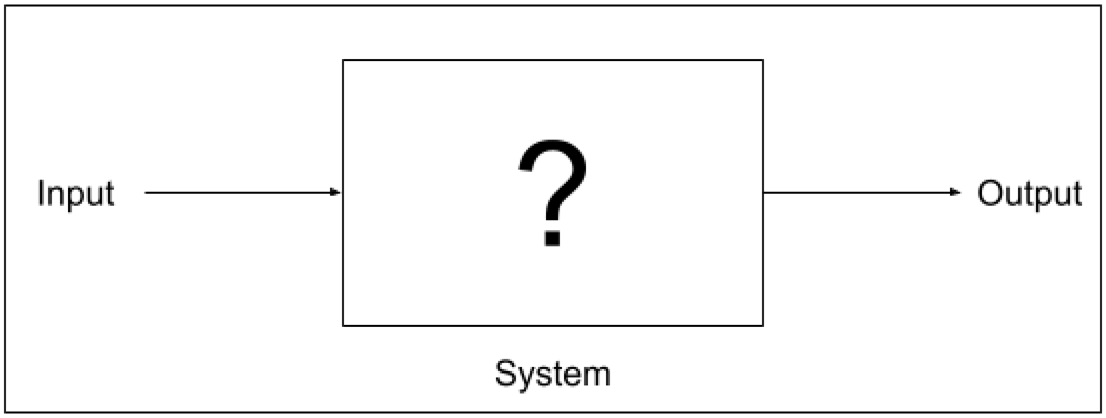


Figure 1: An Input/Output (I/O) System

Building on Figure 1 so the I/O components can be represented in binary format (zeros and ones), we can begin to analyze relationships between electronic components.

Logic Operators and Gates

Definitions of Basic Operators in Digital Logic

We will focus on five basic operators and one more special piece of hardware (MUX) defined as follows:

1. AND: For the output to be true, all inputs "AND-ed" together must be true.
2. OR: For the output to be true, at least one of the "OR-ed" inputs must be true.
3. NOR: For the output to be true, none of the "NOR-ed" inputs can be true.
4. XOR: For the output to be true, the number of true inputs "XOR-ed" together must be odd.
5. NAND: For the output to be true, the"NAND-ed" inputs cannot all be true.
6. MUX: A switch (control input) chooses between data inputs. Output is equal to the chosen input.

Logic Gates

Logic operators can be represented as logic gates. These gates are the hardware or physical components inside electronics. Figures 2-7 illustrate two-input representations of AND, OR, NOR, XOR, and NAND operators and a 3-input representation of the MUX. Accompanied with each figure are "equivalent notations" which preview Boolean Algebra expressions, a topic in "Lab 3: K-Maps and Boolean Algebra." (NOTE: The inverter or NOT gate (not shown) is implied by the bubble on NOR and NAND. It is equivalent to a "negation" denoted by the bar over a variable. The negation of 0 is 1; the negation of 1 is 0.)

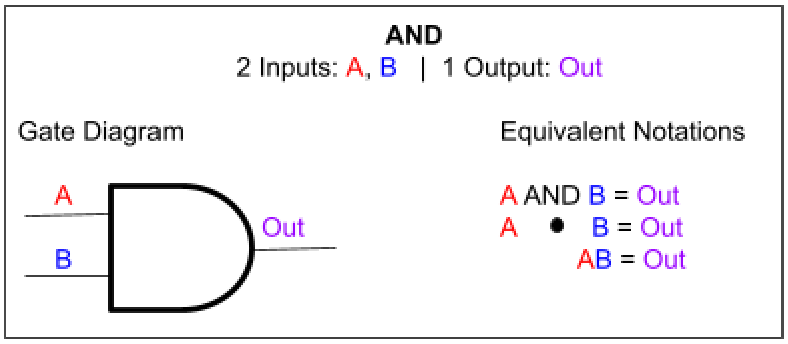


Figure 2: The "AND" Gate and Operator

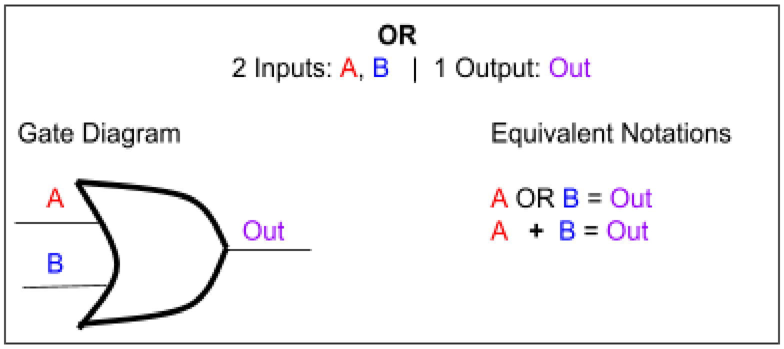
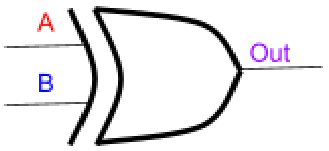


Figure 3: The "OR" Gate and Operator

XOR

2 Inputs: A, B | 1 Output: Out

Gate Diagram



Equivalent Notations

A XOR B = Out

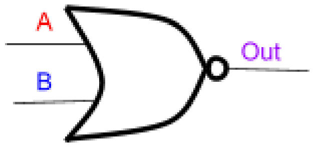
$A \oplus B = Out$

Figure 4: The “XOR” Gate and Operator

NOR

2 Inputs: A, B | 1 Output: Out

Gate Diagram



Equivalent Notations

A NOR B = Out

$\overline{A \text{ OR } B} = Out$

$\overline{A} \bullet \overline{B} = Out$

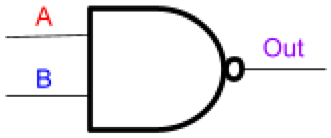
Where the bar over a variable X (ex: $\overline{X}$) means the negation of X.

Figure 5: The “NOR” Gate and Operator

NAND

2 Inputs: A, B | 1 Output: Out

Gate Diagram



Equivalent Notations

A NAND B = Out

$\overline{A \text{ AND } B} = Out$

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

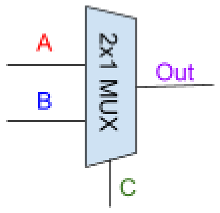
Where the bar over a variable X (ex: $\overline{X}$) means the negation of X.

Figure 6: The “NAND” Gate and Operator

MUX

3 Inputs: A, B, C | 1 Output: Out

Diagram



Equivalent Notations

A = input 1

B = input 2

C = control / select input

Out = output

A 2x1 mux acts like a switch. When C = 0, Out = A. When C = 1, Out = B.

Figure 7: The Multiplexor “MUX”

LabsLand Activity

Premise

The Digital Trainer Activity builds intuition for the AND, OR, NOR, XOR, and NAND operators and the MUX -- the basic building blocks of digital logic. By experimenting with inputs and outputs (I/O) on remote hardware through LabsLand, you will learn how to distinguish relationships between these operators.

Digital Trainer Activity

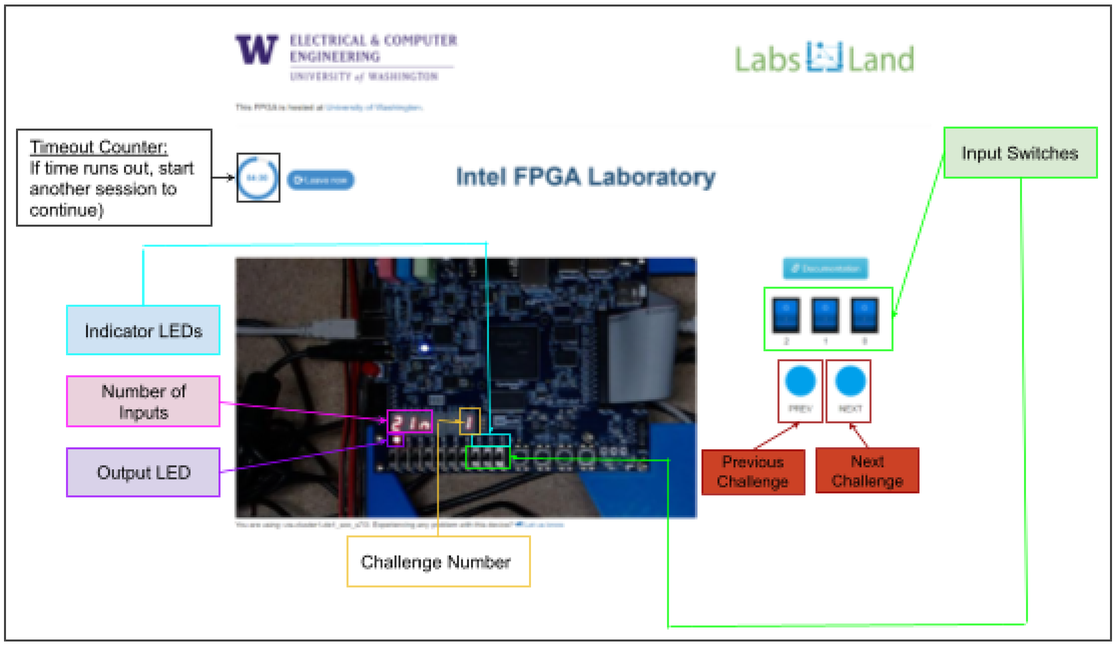


Figure 8: Digital Trainer Diagram

Instructions

1. Familiarize yourself with the LabsLand interface. (A diagram is shown in Figure 8.)
    a. The Field Programmable Gate Array (FPGA) board has 10 challenges pre-programmed.
    b. For each of the 10 challenges, notice that the HEX display (where the red digits and letters are on the FPGA) shows a label for the challenge number and the number of inputs in the challenge.
    c. Locate the blue input switches (Switch2, Switch1, and Switch0) on the right hand side of the screen.
    d. Locate the indicator LEDs (LED2, LED1, LED0) on the right hand side below the HEX display of the challenge number. These LEDs will be lit if their corresponding input switches (Switch2, Switch1, Switch0) are on; the LEDs will be off if their corresponding input switches are off.

e. Locate the output LED (LEDout) on the left hand side below the HEX display of the number of inputs. This LED will be on if an operator or mux applied to the system inputs results in a "true"; otherwise, the LED will be off.
f. Use the blue "Prev" and "Next" buttons to go to the previous or the next challenge respectively.

2. Using the definitions of basic operators, observe the output LEDs when the input switches are in different combinations of on and off. Note that a switch is "on" when it is flipped "up" and "off" when it is flipped "down."

3. Based on your observations, record your initial thoughts in "Digital Trainer Observations" and your guess for the corresponding logic function on display for each challenge in "Digital Trainer Hypotheses and Logic Gates."

Digital Trainer Observations

What do you notice about the output LED when the input switches are in different combinations of on and off? Which combinations result in the output LED being on vs. off? Notice that some challenges use 2 inputs, while others use 3 inputs (check the HEX display to find out the number of inputs in the challenge). Hints are given to help you start challenge 1.

- Challenge 1:
  - When Switch1 is false and Switch0 is false, LEDOut is _____________.
  - When Switch1 is false and Switch0 is true, LEDOut is _____________.
  - When Switch1 is true and Switch0 is false, LEDOut is _____________.
  - When Switch1 is true and Switch0 is true, LEDOut is _____________.

- Challenge 2:

- Challenge 3:

- Challenge 4:

- Challenge 5:

- Challenge 6:

- Challenge 7:

- Challenge 8:

- Challenge 9:

- Challenge 10:

Based on your observations, what operator do you suppose is on display for each challenge? Use Table 1 to record your guess for what logic function or operator is on display.

Table 1: Digital Trainer Hypotheses

Challenge	How many inputs?	What is the corresponding logic function or operator?				
1		AND	OR	NAND	NOR	XOR
2		AND	OR	NAND	NOR	XOR
3		AND	OR	NAND	NOR	XOR
4		AND	OR	NAND	NOR	XOR
5		AND	OR	NAND	NOR	XOR
6		AND	OR	NAND	NOR	MUX
7		AND	OR	NAND	NOR	MUX
8		AND	OR	NAND	NOR	MUX
9		AND	OR	NAND	NOR	MUX
10		AND	OR	NAND	NOR	MUX

Draw the corresponding logic gate or diagram of each of your guesses below.

Challenge 1:

Challenge 2:

Challenge 3:

Challenge 4:

Challenge 5:

Challenge 6:

Challenge 7:

Challenge 8:

Challenge 9:

Challenge 10:

Looking Ahead

Look back at Table 1 where you kept track of your hypotheses. Fill out Table 2 with your guess and a short description of why you chose that operator.

Table 2: Digital Trainer Reasoning

Challenge	Guess	Why did you choose this operator (1-2 sentences)?
1		
2		
3		
4		
5		

6		
7		
8		
9		
10		

These observations will help you in the following lab which introduces a graphic organizer called a “Truth Table”. The Truth Table is a mechanism for mapping I/O to zeros and ones which will help you determine if your guesses match the actual operator used in each of the 10 challenges.

Reflection and Observations

Lab 1 introduced you to logic gates and operators. Reflect on things you found interesting and/or challenging in the space below.

What questions do you still have, if any?

Lab 2: BINARY NUMBERS AND TRUTH TABLES

The University of Washington | The Remote Hub Lab | Last Revised: March 2022

Summary

This lab introduces Truth Tables as graphic organizers for digital logic analysis. In this lab you will learn about binary numbers, truth table practice, and an application of truth tables.

Table of Contents

Building on Lab 1

In the previous lab, we covered digital logic operators and gates including AND, OR, NOR, XOR, NAND, and MUX. If you recall, inputs and outputs can be represented as zeros and ones. This lab will allow you to check your intuition about the Digital Trainer's 10 Challenges with zeros and ones organized by a Truth Table.

OPTIONAL: If it is helpful for you, copy your guesses from Lab 1's Table 1 to this lab's Table 1 for easy access. Otherwise, have your answers from Lab 1 readily available to you throughout this lab.

Table 1: Digital Trainer Hypotheses

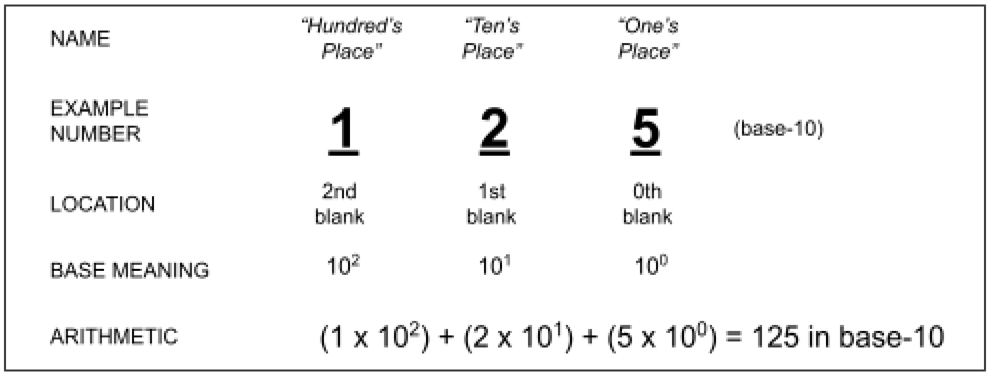
Challenge	How many inputs?	What is the corresponding logic function or operator?				
1		AND	OR	NAND	NOR	XOR
2		AND	OR	NAND	NOR	XOR
3		AND	OR	NAND	NOR	XOR
4		AND	OR	NAND	NOR	XOR
5		AND	OR	NAND	NOR	XOR
6		AND	OR	NAND	NOR	MUX
7		AND	OR	NAND	NOR	MUX
8		AND	OR	NAND	NOR	MUX
9		AND	OR	NAND	NOR	MUX
10		AND	OR	NAND	NOR	MUX

Binary Numbers

Premise

Before we introduce Truth Tables, we need a basic understanding of binary numbers. Numbers are binary if they use "base-2" arithmetic. This means every digit in a number can only be a 0 or a 1; there are only two possibilities. In regular arithmetic, every digit in a number can be a 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. These 10 possibilities for each digit mean we use "base-10" arithmetic in everyday life. Electronics, however, are much more simplistic, so a one-digit number like 8 in base-10 is equivalent to 1000 in binary. How did this arise?

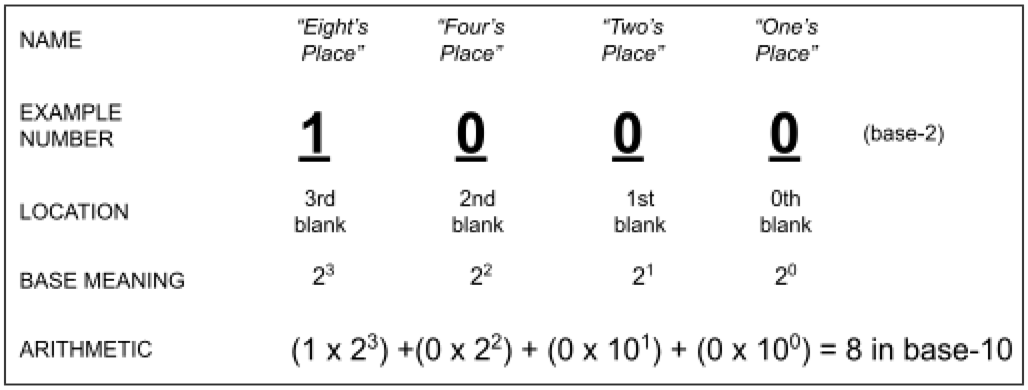
Let's start with base-10 numbers. Base-10 numbers are made up of digits. The rightmost digit is the "ones" place. Moving left, the place increases by a factor of 10. Take 125 for example.



NAME	*"Hundred's Place"*	*"Ten's Place"*	*"One's Place"*	
EXAMPLE NUMBER	**1**	**2**	**5**	(base-10)
LOCATION	2nd blank	1st blank	0th blank	
BASE MEANING	10^2	10^1	10^0	
ARITHMETIC	$(1 \times 10^2) + (2 \times 10^1) + (5 \times 10^0) = 125$ in base-10			

Figure 1: Base-10 Math

Figure 1 shows the arithmetic value of 125 is preserved if we view 125 as having three sub-locations (0th blank, 1st blank, and 2nd blank). Knowing 125 is base-10, we can break down the number by its one's, ten's, and hundred's places. We can think of binary numbers in the same way. Take 1000 in Figure 2, and note how each digit can only be zero or one.



NAME	*"Eight's Place"*	*"Four's Place"*	*"Two's Place"*	*"One's Place"*	
EXAMPLE NUMBER	**1**	**0**	**0**	**0**	(base-2)
LOCATION	3rd blank	2nd blank	1st blank	0th blank	
BASE MEANING	2^3	2^2	2^1	2^0	
ARITHMETIC	$(1 \times 2^3) + (0 \times 2^2) + (0 \times 10^1) + (0 \times 10^0) = 8$ in base-10				

Figure 2: Binary (Base-2) Math

We can see that base-10 and binary numbers can be understood similarly. Binary digits are incremented by powers of two, whereas base-10 digits are incremented by powers of ten.

What You Need To Know

For the purposes of this lab, you should be able to recognize and write out small binary numbers equivalent to 0-15 in base-10. To get you started, Table 2 lists the binary equivalents of the numbers 0-9.

Table 2: Base-10 and Base-2 Numbers

BASE-10	BASE-2 (BINARY)
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

Challenge Questions

To test your understanding, find the binary equivalents of the numbers 11, 12, and 15. Hint: find the largest powers of 2which fit in the number.

- 11 =
- 12 =
- 15 =

Find the base-10 equivalents of these numbers. Hint: each digit corresponds to a power of 2.

- 1010 =
- 1101 =
- 1110 =

What is a Truth Table?

Truth Tables are graphic organizers for keeping track of patterns between the inputs and outputs. Remember that "1" represents "on" and "0" represents "off."

The general form is a T-chart with inputs on the left and outputs on the right. The labels are located at the top of the T-chart, specifying the name of input and output signals. A blank Truth Table with two input signals, A and B, and one output signal C is depicted in Figure 3.

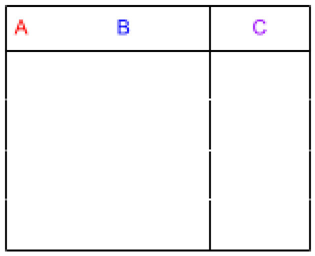


Figure 3: A Blank Truth Table

The information in a Truth Table represents whether the signal was true/on (shown by a 1) or false/off (shown by a 0). Given x inputs, there are 2^x entries of outputs in the Truth Table. The Truth Table inputs correspond to binary numbers, with 0 at the top and (2^x - 1) at the bottom. Figure 4 shows a color-labeled diagram of each column and row in a Truth Table with sample data.

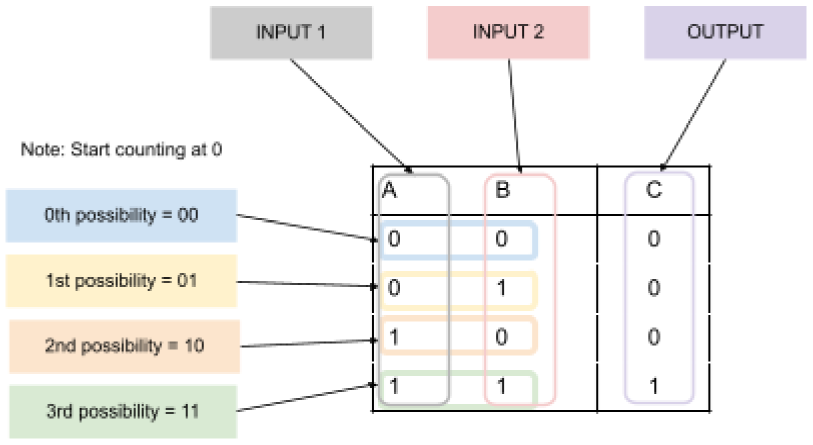


Figure 4: A Truth Table with Data

Notice that, because we have two inputs, there are 2^2 = 4 possible combinations of zeros and ones. Counting in binary, that means the left side of the Truth Table goes from 00 to 11 (0 to 3 in base-10). The right side represents the result of each possibility. In this case, the output was only true when both input signals were on. All other cases -- A and B both off (00), A off and B on (01), A on and B off (10) -- resulted in false outputs.

Application

Figures 5a-b have 10 blank Truth Tables, 1 for each of the 10 challenges you observed in Lab 1. **For the best learning experience, do not look at Figure 6 until you complete Figures 5a-b.**

- For each row (an input combination), record the output as a 1 (for on) or 0 (for off).
- Use the output information to guess/verify what the corresponding operator is: AND, OR, NAND, NOR, XOR, and MUX.
- Refer back to the Digital Trainer activity in LabsLand as needed. Also, refer to the definitions for these operators in Lab 1. Your guess can be the same or different from your guess in Lab 1.
- Note that some of the Truth Tables have the input signals filled in as hints; for the others, you must complete all the columns.

Challenge 1
Operator:

SW1	SW0	LEDout
0	0	
0	1	
1	0	
1	1	

Challenge 2
Operator:

SW1	SW0	LEDout
0	0	
0	1	
1	0	
1	1	

Challenge 3
Operator:

SW1	SW0	LEDout

Challenge 4
Operator:

SW1	SW0	LEDout

Figure 5a: Digital Trainer Blank Truth Tables

Challenge 5
Operator:

SW1	SW0	LEDout

Challenge 6
Operator:

SW2	SW1	SW0	LEDout
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Challenge 7
Operator:

SW2	SW1	SW0	LEDout
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Challenge 8
Operator:

SW2	SW1	SW0	LEDout

Challenge 9
Operator:

SW2	SW1	SW0	LEDout

Challenge 10
Operator:

SW2	SW1	SW0	LEDout

Figure 5b: Digital Trainer Blank Truth Tables

Recognizing Patterns

Figure 4 illustrates 10 Truth Tables for the 10 challenges you observed in LabsLand. For each Truth Table, the corresponding operator for the challenge is given.

Challenge 1: NOR

SW1	SW0	LEDout
0	0	1
0	1	0
1	0	0
1	1	0

Challenge 2: AND

SW1	SW0	LEDout
0	0	0
0	1	0
1	0	0
1	1	1

Challenge 3: XOR

SW1	SW0	LEDout
0	0	0
0	1	1
1	0	1
1	1	0

Challenge 4: NAND

SW1	SW0	LEDout
0	0	1
0	1	1
1	0	1
1	1	0

Challenge 5: OR

SW1	SW0	LEDout
0	0	0
0	1	1
1	0	1
1	1	1

Challenge 6: MUX

SW2	SW1	SW0	LEDout
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Challenge 7: OR

SW2	SW1	SW0	LEDout
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Challenge 8: NAND

SW2	SW1	SW0	LEDout
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Challenge 9: NOR

SW2	SW1	SW0	LEDout
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Challenge 10: AND

SW2	SW1	SW0	LEDout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Figure 6: Digital Trainer Truth Tables

Compare your Truth Tables in Figures 5a-5b the Truth Tables in Figure 6. If your guess matches, compare your initial reasoning (before learning about Truth Tables) to the Truth Table process. If the tables do not match, revisit LabsLand and double check the system behavior. Describe what you observe in Table 3.

Table 3: Truth Table Reasoning

Challenge	Guess	Did your guess match the operator? Why or why not? (1-2 sentences)
1		
2		
3		
4		
5		

6		
7		
8		
9		
10		

The reflection and Truth Tables you completed will be helpful in our transition to Boolean Algebra and K-Maps in the following lab. Understanding Truth Tables as graphic organizers will help you derive and represent relationships from observed I/O.

Reflection and Observations

Lab 2 introduced you to Truth Tables and binary numbers. Reflect on the new topics and the things you found interesting and/or challenging in the space below.

What questions do you still have, if any?

ELECTRICAL & COMPUTER ENGINEERING

Lab 3: BOOLEAN ALGEBRA AND K-MAPS

The University of Washington | The Remote Hub Lab | Last Revised: March 2022

Summary

This lab introduces the concept of Karnaugh Maps (K-Maps), a graphic organizer method which converts Truth Tables into Boolean algebra equations or expressions. In this lab you will learn about Boolean algebra, K-Maps, and practice with an application on LabsLand.

Table of Contents

What is Boolean Algebra?

Boolean Algebra Background

Operators are the building blocks of digital logic because they relate input signals to the appropriate output signal(s). Recall from Lab 1 that examples of these operators or hardware are AND, OR, NOR, XOR, NAND, and MUX. While the “word” form of these operators is helpful for understanding their meaning, there is a simpler way to articulate the relationship between signals. To express all the information in a Truth Table in a concise way, Boolean algebra uses regular arithmetic symbols like ‘+’ to represent OR and ‘•’ to represent AND

Lab 1 provided a summary of operators and gate diagrams, depicted here by Figure 1.

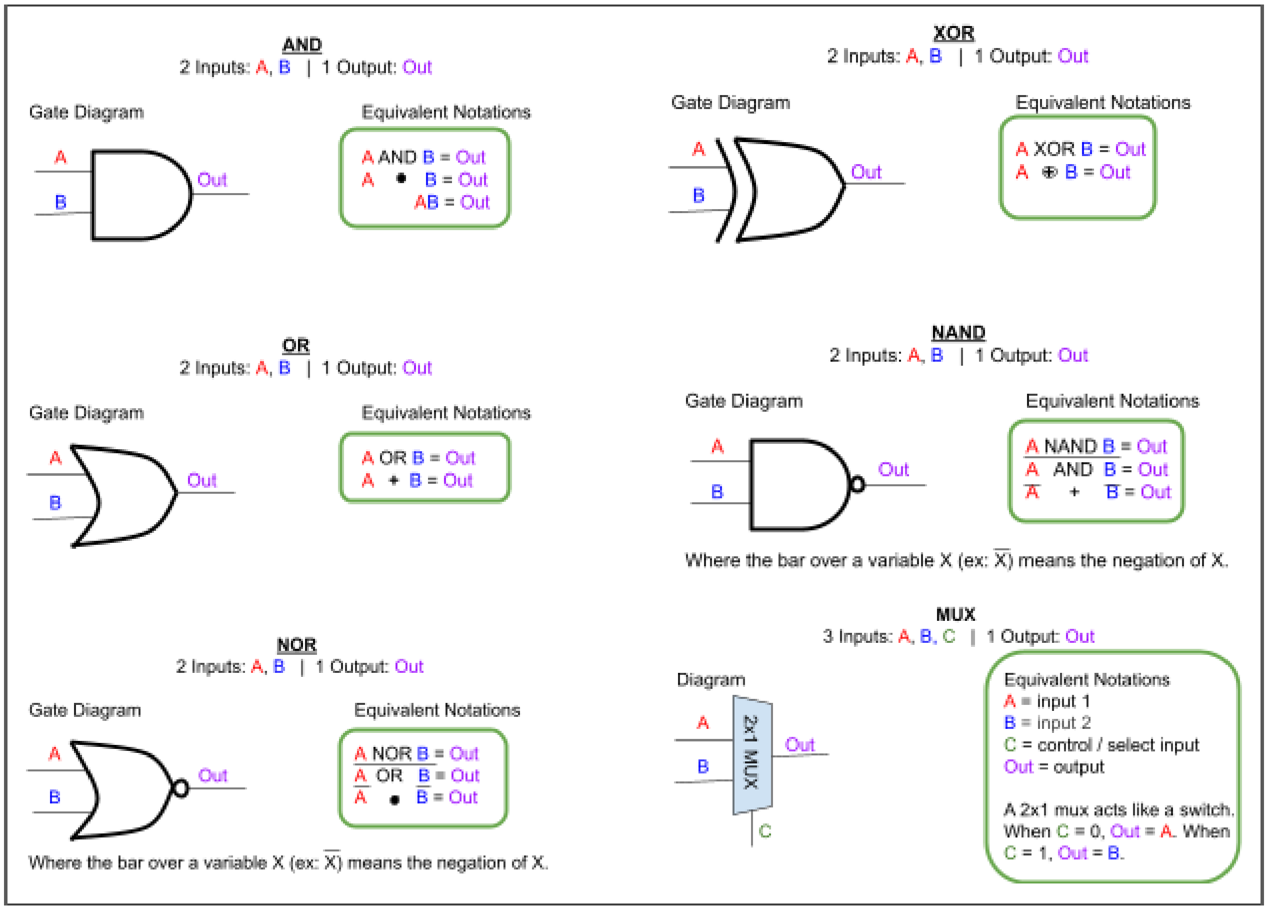

AND
2 Inputs: A, B | 1 Output: Out
Gate Diagram
Equivalent Notations
A AND B = Out
A • B = Out
AB = Out
XOR
2 Inputs: A, B | 1 Output: Out
A XOR B = Out
A ⊕ B = Out
OR
2 Inputs: A, B | 1 Output: Out
A OR B = Out
A + B = Out
NAND
2 Inputs: A, B | 1 Output: Out
$\overline{A \text{ NAND } B}$ = Out
$\overline{A \text{ AND } B}$ = Out
$\overline{A}$ + $\overline{B}$ = Out
Where the bar over a variable X (ex: $\overline{X}$) means the negation of X.
NOR
2 Inputs: A, B | 1 Output: Out
$\overline{A \text{ NOR } B}$ = Out
$\overline{A \text{ OR } B}$ = Out
$\overline{A}$ • $\overline{B}$ = Out
Where the bar over a variable X (ex: $\overline{X}$) means the negation of X.
MUX
3 Inputs: A, B, C | 1 Output: Out
Diagram
2x1 MUX
A = input 1
B = input 2
C = control / select input
Out = output
A 2x1 mux acts like a switch. When C = 0, Out = A. When C = 1, Out = B.

Figure 1: Operators and Gate Diagrams

The right-hand columns labeled “Equivalent Notations” and boxed in **green** are examples of Boolean Algebra expressions. Recall that these gates are physical components in electronic circuits, so it is ideal that we minimize the complexity of gates in our operations. In particular, Boolean Algebra can help us reduce the number of inputs to the gates and the number of gates.

Rules to Remember

Figures 2-3 list some important Boolean algebra identities and laws, along with an explanation for how to think about the relationship. Recall that a variable with a bar over it means the "negation" or the "opposite" of the variable. For example, $\overline{X}$ means 'not X', so if X is true, $\overline{X}$ is false. Conversely, if X is false, $\overline{X}$ is true. Key identities that may be new to you are highlighted in green. Notice that many identities and laws are similar to the rules of regular algebra.

"OR" (+) Identities

⟶ *When is (X OR 0) true?* **Ans**: 0 can never be true. Answer is only when **X** is true.

$$X + 0 = X$$

⟶ *When is (X OR True) true?* **Ans**: 1 is always true no matter X. Answer is **1** (all the time).

$$X + 1 = 1$$

⟶ *When is (X OR X) true?* **Ans**: We have the same input signal. Answer is only when **X** is true.

$$X + X = X$$

⟶ *When is (X OR $\overline{X}$) true?* **Ans**: X and not X are the only possibilities, so at least one of them is true all the time, so the answer is **1**.

$$X + \overline{X} = 1$$

"AND" (•) Identities

⟶ *When is (X AND True) true?* **Ans**: 1 is always true, so the answer depends only on **X**.

$$X \bullet 1 = X$$

⟶ *When is (X AND False) true?* **Ans**: **0** is always false, no matter X.

$$X \bullet 0 = 0$$

⟶ *When is (X AND X) true?* **Ans**: Same input signal, so answer is only when **X** is true.

$$X \bullet X = XX = X$$

⟶ *When is (X OR $\overline{X}$) true?* **Ans**: X and not X are the only possibilities of X, so both cannot be true at the same time. The answer is **0**.

$$X \bullet \overline{X} = X\overline{X} = 0$$

Figure 2: Boolean Algebra Identities

Laws

Commutative Laws:

→ *Comparing when X OR Y is true is the same as comparing when Y OR X is true.*

$$X + Y = Y + X$$

→ *Comparing when X AND Y is true is the same as comparing when Y AND X is true.*

$$XY = YX$$

Associative Laws:

→ *Set A = (Y OR Z) = (Y+Z). Set B = (X OR Y) = (X+Y). (X OR A) is the same as (B OR Z).*

$$X + (Y + Z) = (X + Y) + Z$$

→ *Set A = YZ. Set B = XY. (X AND A) is the same as (B AND Z).*

$$X(YZ) = (XY)Z$$

DeMorgan's Laws:

→ *Negating an entire operation = negating each signal and changing to the opposite operator.*

$$\overline{(X + Y)} = \overline{X} \bullet \overline{Y} \qquad \overline{(X \bullet Y)} = \overline{X} + \overline{Y}$$

Absorption Laws:

- $X + XY = X(1 + Y) = X(1) = X$

$$X + XY = X$$

- $XY + X\overline{Y} = X(Y + \overline{Y}) = X(1) = X$

$$XY + \overline{X}Y = X$$

- $X + \overline{X}Y = X + XY + \overline{X}Y = X + Y(X + \overline{X}) = X + Y(1) = X + Y$

$$X + \overline{X}Y = X + Y$$

- $X(X + Y) = XX + XY = X + XY = X(1 + Y) = X(1) = X.$

$$X(X + Y) = X$$

- $(X + Y)(X + \overline{Y}) = XX + X\overline{Y} + XY + Y\overline{Y} = X + X\overline{Y} + XY + 0 = X + XY = X$

$$(X + Y)(X + \overline{Y}) = X$$

- $(X)(\overline{X} + Y) = X\overline{X} + XY = 0 + XY = XY$

$$X(\overline{X} + Y) = XY$$

Distributive Laws:

→ *The AND operator distributes over OR'ed quantities.*

$$X(Y + Z) = XY + XZ$$

→ *(X+Y)(X+Z) = XX + XZ + YX + YZ = X + XZ + XY + YZ = X(1+Z+Y) + YZ = X + YZ*

$$X + YZ = (X + Y)(X + Z)$$

Figure 3: Boolean Algebra Laws

What is the simplest form of the expression $(A + C)(AD + A\overline{D}) + AC$? Use the identities and laws in Figures 2-3. (To see examples, refer to the "Absorption Laws" in Figure 3.)

What is a K-Map?

Karnaugh Map Background

Karnaugh Maps, or K-Maps, are another way of simplifying Boolean expressions. Using a grid-type chart, K-Maps allow you to arrive at a simplified Boolean expression by eliminating redundant variables. The size of the chart depends on the number of inputs, as shown in Figure 4. To understand how K-Maps work, look at the 2-input, 3-input, and 4-input cases in Figure 4.

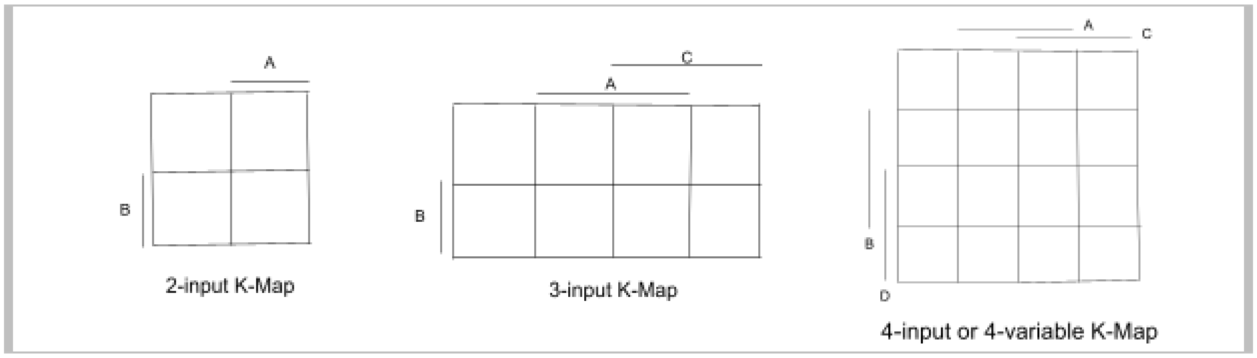


Figure 4: Blank K-Maps

All these K-Map types involve smaller grid squares. Each square represents the output value for an input combination (and thus a row on a Truth Table). For example, imagine a 2-input K-Map. It has labels for its variables written on the perimeter of the chart and marked with a bar for the relevant columns and rows. These labels designate when the inputs are true. Take the K-Map with sample data in Figure 5, assuming the inputs are called ‘A’ and ‘B’.

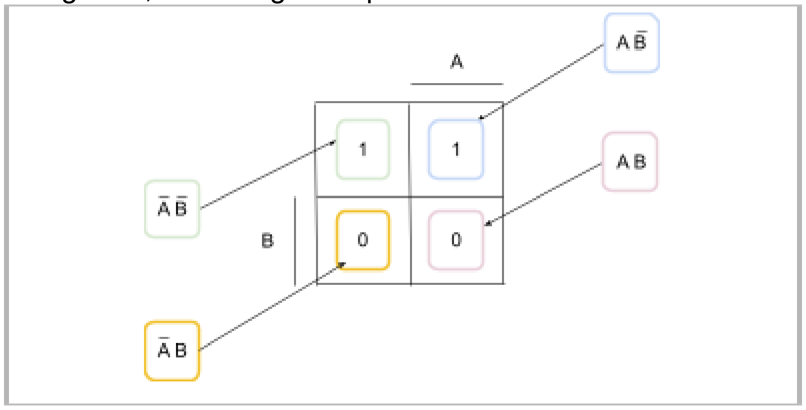


Figure 5: K-Map with Sample Data

Notice the labels for two inputs, A and B, mark a column and a row respectively. This means the squares under the column labeled A denote when A is true. The left-hand column signifies when A is false. Moreover, the squares in the row aligned with the label B designate when B is true, and the top row designates when B is false. If the output is ‘true’ when inputs A and B are on, then the corresponding box for ‘AB’ will contain a 1. In this K-Map, AB is false, so the pink square in the bottom right corner has a 0.

The 3-input and 4-input cases follow the same reasoning. Because each square in a K-Map represents the output of an input combination, we can derive a Truth Table from a K-Map and vice versa. For accuracy, you need to “fill in” a K-Map in a certain order; that’s where binary numbers come in. Observe the numbers in the bottom left of the grid squares in each K-Map in Figure 6; they correspond to specific rows (and thus input combinations) on a Truth Table.

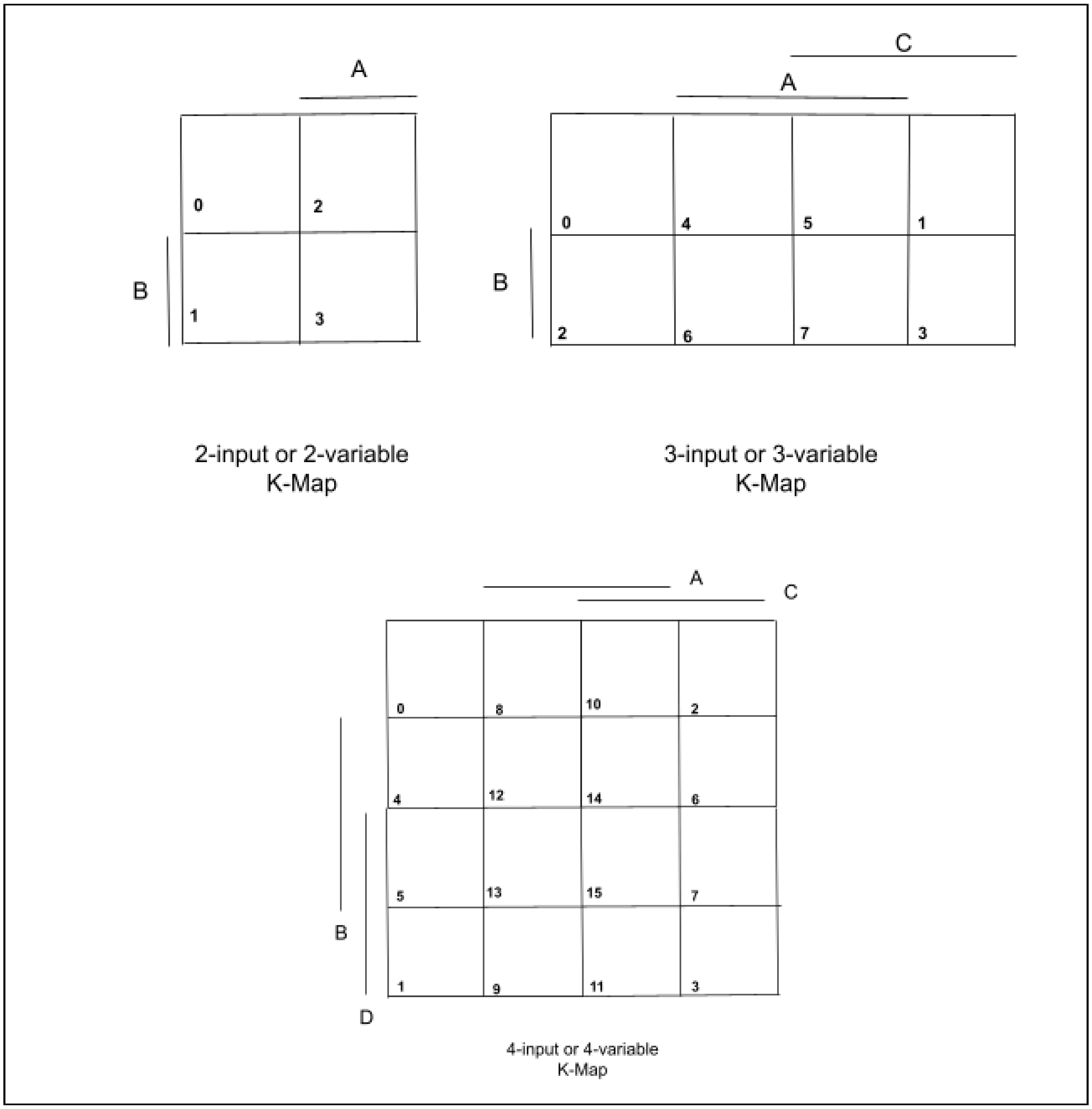
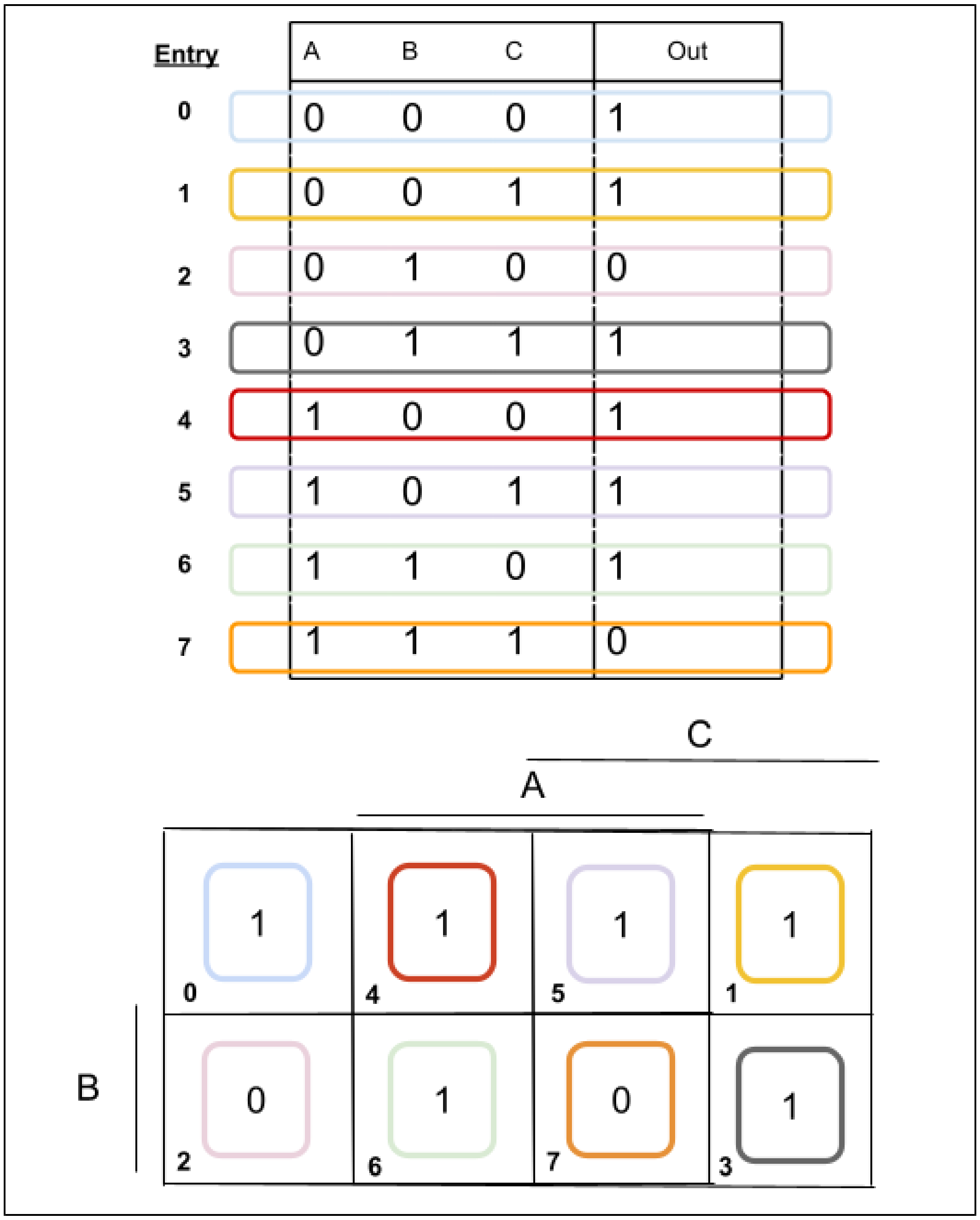

A
B
0
2
1
3
2-input or 2-variable K-Map
C
A
B
0
4
5
1
2
6
7
3
3-input or 3-variable K-Map
A
C
B
D
0
8
10
2
4
12
14
6
5
13
15
7
1
9
11
3
4-input or 4-variable K-Map

Figure 6: Input Combinations Mapped to K-Map Grid Squares

As an example, let's look at the Truth Table and corresponding K-Map in Figure 7.



Entry	A	B	C	Out
0	0	0	0	1
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 7: Truth Table to K-Map Conversion

Notice how the 3-inputs map to labels around the perimeter of the K-Map. Moreover, notice how the nth entry of the Truth Table corresponds to the grid square label in the bottom left corner. This illustrates how, when translating a Truth Table to a K-Map, you should carefully transfer the outputs to the appropriate grid square. This process also applies to translating a K-Map to a Truth Table.

Simplifying K-Maps

Now that we know how to understand and write K-Maps from Truth Tables and vice versa, how can we simplify the Boolean algebra? With 2-input, 3-input, and 4-input K-Maps, the trick is to look for groups of adjacent 1s in quantities of one, two, four, eight, or sixteen and group them together. The goal is to maximize the number of 1s in a group, but minimize the number of groups we make. Only the squares encompassed by a group will be included in the expression.

A 2-input example is shown in Figure 8. Notice that the output contains the expression that best describes the group of 1s. In this case, we need an expression for the top row. We notice that the blue group is the region not included by B. As a result, the output of this system is $\overline{B}$ according to the K-Map.

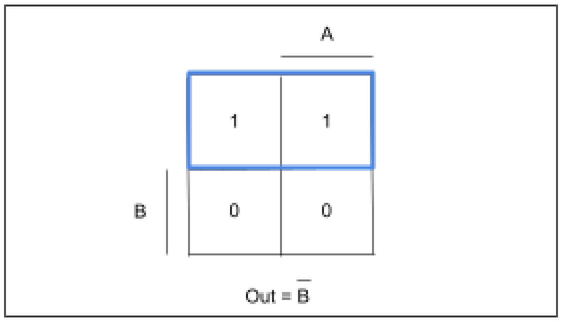


Figure 8: Forming Groups on K-Maps (2-Input Example)

Figure 9 depicts another example with a 3-input K-Map. Note that the simplest expressions may have a group of four which can "go off the grid." This can happen from right-to-left (as shown in Figure 9) or top-to-bottom. Because the bottom left corner 1 and top right corner 1 were already covered by the group of four, we only need to cover the two 1s highlighted in pink. But, we notice that there are adjacent 1s to these lone, uncovered pink 1s. It is important to note that our goal is to simplify, so we would rather have 2-variable terms rather than 3-variable terms. As a result, we allow overlap so lone, uncovered 1s can join a group of two, four, eight, or sixteen.

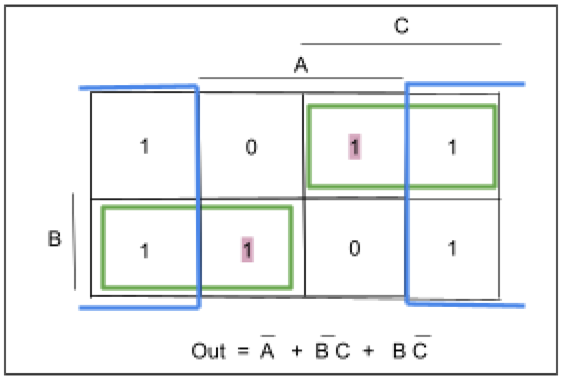
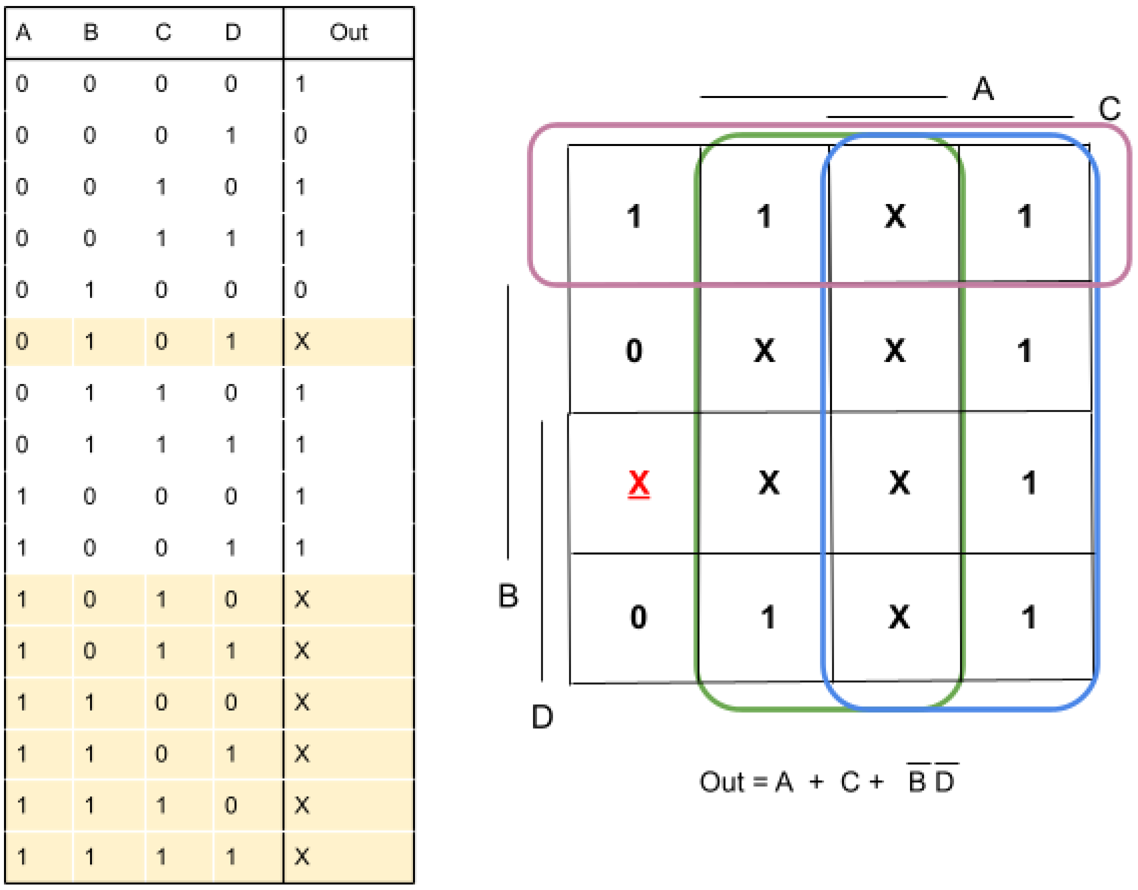


Figure 9: Forming Groups on K-Maps (3-Input Example)

The result is the sum of all possible groups. In this case, we found three groups so there are three terms added together - the "OR"-ing of all possibilities.

Sometimes, there are Truth Tables (and by extension, K-Maps) with input combinations which are not relevant to the system. For example, assume we have a 4-input Truth Table which has defined behavior for all input combinations except for 5, 10, 11, 12, 13, 14, and 15. In this case, we should put X's in the output column(s) of these input combinations which are irrelevant to the system. One of these X's is called a "Don't Care." This Truth Table example and how to simplify its K-Map is shown in Figure 10.



A	B	C	D	Out
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

$$Out = A + C + \overline{B}\,\overline{D}$$

Figure 10: Truth Table with Don't Cares and K-Map Simplification

In Figure 10's Truth Table, notice that we "don't care" about the input combinations which are highlighted in yellow, so the output column is marked with X's. These Don't Cares can be treated as '1s' or '0s' if it helps us maximize the number of 1s in a group, but minimize the number of groups we make. This simplification is seen in Figure 10's K-Map. All Don't Cares in the middle two columns can help create non-redundant groups of eight 1s. Consequently, all Don't Cares are assumed to be '1' except for the Don't Care in red underlined.

Summary of K-Map Rules

STEP 1: Draw the K-Map.

1. *Draw a 2x2 square grid (2-input system), a 4x2 square grid (3-input system), or a 4x4 square grid (4-input system).*
2. *Fill in the smaller squares with the outputs from the Truth Table.*

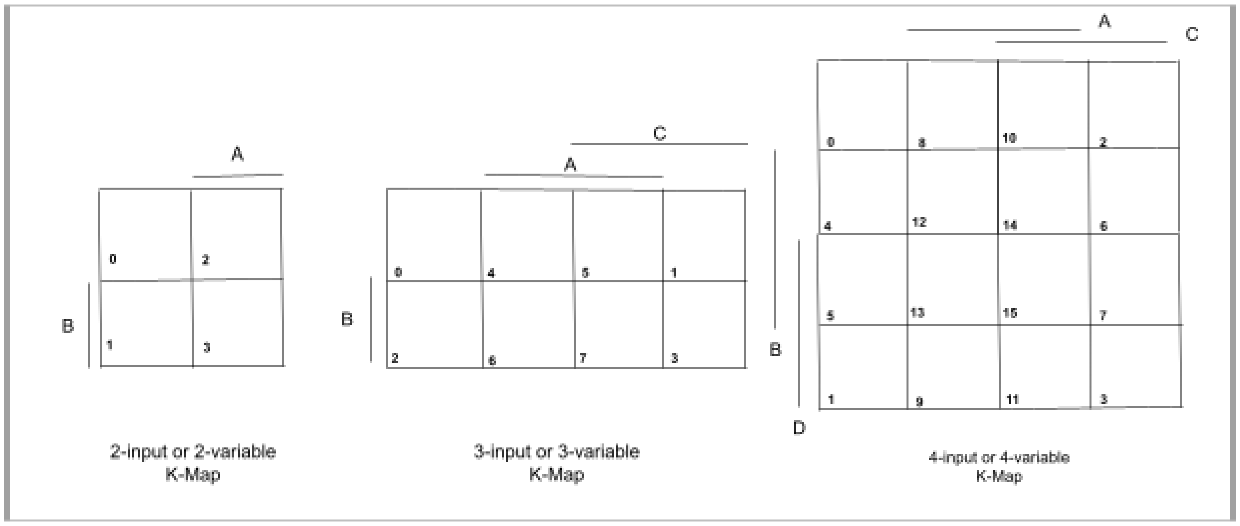

A
B
0
2
1
3
2-input or 2-variable K-Map
C
A
B
0
4
5
1
2
6
7
3
3-input or 3-variable K-Map
A
C
B
D
0
8
10
2
4
12
14
6
5
13
15
7
1
9
11
3
4-input or 4-variable K-Map

Figure 11: Input Combinations Mapped to K-Map Grid Squares

STEP 2: Form groups according to the following rules:

1. *The goal is to form groups of adjacent 1s.*
2. *Groups must take a square or rectangular shape (no diagonals).*
3. *Groups of 1s can be a power of two in size: one, two, four, eight, or sixteen.*
4. *Maximize the number of 1s in each group, and minimize the number of groups you make. (This may involve Don't Care's.)*

A group is "maximized" when it includes all possible adjacent 1s. The example in Figure 12 shows a maximized group of four vs. two non-maximized pairs. The one on the left is more correct because it maximizes the number of 1s in a group (there are four 1s) and minimizes the number of groups (there is only one group instead of two). Note that going "off the grid" or off the sides of the grid is legal!

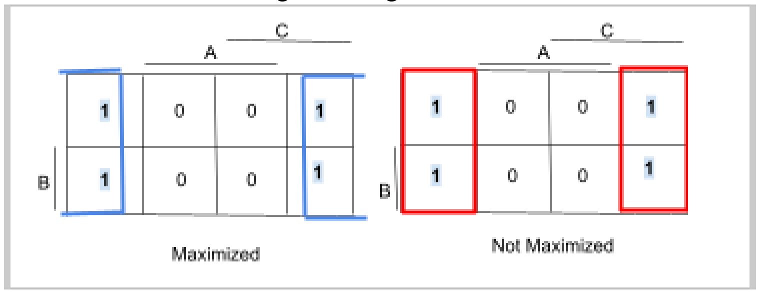


Figure 12: Maximized vs. Non-Maximized K-Maps

5. *Groups can overlap, but groups cannot be fully redundant; only make groups if there is a 1 that is uncovered/ungrouped.*

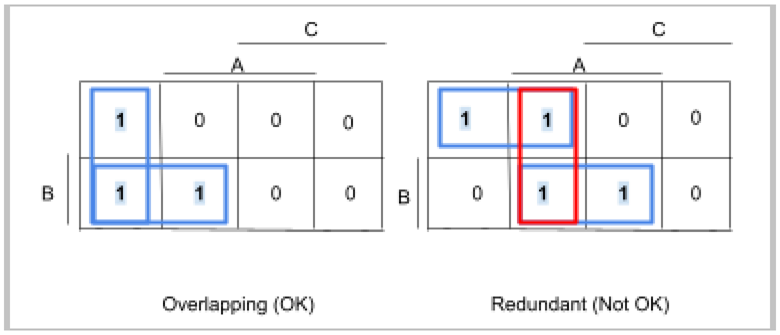


Figure 13: Overlapping vs. Redundant K-Map Groups

6. *You are "done" finding groups when every 1 is part of a maximized group.*

   In Figure 13, notice that, after finding a pair of ones in the left K-Map (say the pair in the bottom row), there is still a 1 above without a group. Remember that we want to maximize the number of ones in a group and minimize the number of groups, so we would rather have two groups of two 1s rather than one pair and a single 1. As a result, we group the remaining 1 (in the $\overline{A}\,\overline{B}\,\overline{C}$ position) with the 1 below it. So, we get two groups which overlap.

<u>STEP 3: Translate into a Boolean expression.</u>

1. *Name the groups you found using the labels on the K-Map perimeter.*

   In the 2-input K-Map, notice that the label 'A' is aligned with the right column and the label 'B' is aligned with the bottom row. Similarly, in the 3-input K-Map, notice that the label 'A' is aligned with the middle four-square region, 'C' is aligned with the right four-square region, and 'B' is aligned with the bottom row.

   When naming a group, locate which region(s) contain it and which region(s) do not contain it. For example, look at the pair of ones in the bottom row of the left K-Map in Figure 13. Notice that this group is not contained by C but is contained by B. However, one element of this group is contained by A but the other element is not contained by A. Therefore, we only need B and C to describe the group, and we can name it $B\overline{C}$ because it is NOT in C and it is in B. Following similar reasoning, the vertical pair in the leftmost column is not contained by A, is not contained by C, and is split over B. Therefore, this group is called $\overline{A}\,\overline{C}$.

2. *Add the terms together to arrive at the simplified result.*

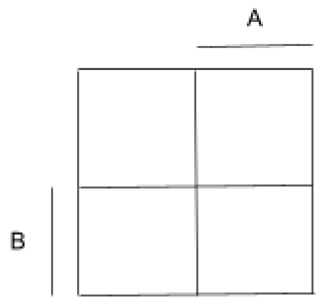
   This works because you are essentially "OR-ing" each possible group together. The simplified result for the left (correct) K-Map in Figure 13 is $Out = B\overline{C} + \overline{A}\,\overline{C}$.

Practice: Truth Table to K-Map to Boolean Expression

Using the given Truth Tables, draw the corresponding K-Maps and determine the Boolean expression.

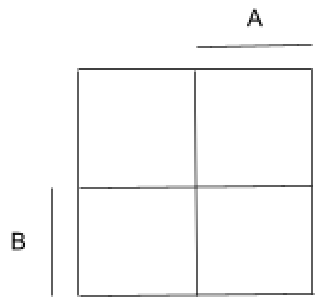
2-input K-Maps

A	B	Out
0	0	0
0	1	0
1	0	1
1	1	0



Out = ____________________

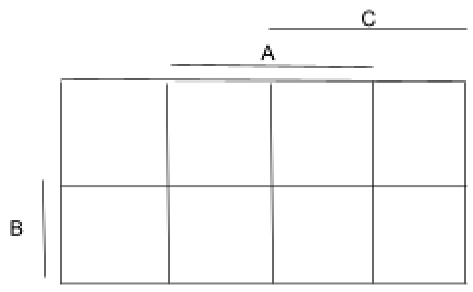
A	B	C
0	0	1
0	1	1
1	0	0
1	1	0



Out = ____________________

3-input K-Maps

A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Out = ____________________

K-Maps with the Digital Trainer

For each challenge corresponding to the Digital Trainer Activity, do the following:

1. Fill in the Truth Table. You can use your answers from Lab 1 or Lab 2.
2. Use the Truth Table to make a K-Map.
3. Use the K-Map to arrive at a Boolean Expression/Equation.
4. Check that the Boolean Equation you arrive at matches the correct operator you found in Lab 1. Remember that there are multiple equivalent representations of the same Boolean relationship. For example, $\overline{(X + Y)} = \overline{X} \bullet \overline{Y}$ and $\overline{(X \bullet Y)} = \overline{X} + \overline{Y}$.

Table 1: Digital Trainer Challenge 1

SW1	SW0	LEDout

Boolean Equation: ______________________________

Table 2: Digital Trainer Challenge 2

SW1	SW0	LEDout

Boolean Equation: ______________________________

Table 3: Digital Trainer Challenge 3

SW1	SW0	LEDout

Boolean Equation: ______________________________

Table 4: Digital Trainer Challenge 4

SW1 SW0	LEDout

Boolean Equation: ______________________

Table 5: Digital Trainer Challenge 5

SW1 SW0	LEDout

Boolean Equation: ______________________

Table 6: Digital Trainer Challenge 6

SW2 SW1 SW0	LEDout

Boolean Equation: ______________________

Table 7: Digital Trainer Challenge 7

SW2	SW1	SW0	LEDout

Boolean Equation: ____________________________

Table 8: Digital Trainer Challenge 8

SW2	SW1	SW0	LEDout

Boolean Equation: ____________________________

Table 9: Digital Trainer Challenge 9

SW2	SW1	SW0	LEDout

Boolean Equation: ___________________________

Table 10: Digital Trainer Challenge 10

SW2	SW1	SW0	LEDout

Boolean Equation: ___________________________

Exercise on LabsLand

Use the Boole-Web feature on LabsLand to see the whole I/O → Truth Table → K-Map process, and verify your answers to Challenges 1-10 from the Digital Trainer Activity. Follow the process detailed in this section to access Boole-Web, and repeat steps 4-10 for each of the challenges.

1. On your LabsLand dashboard, navigate to the "Boole" feature, and click on "Access this lab."

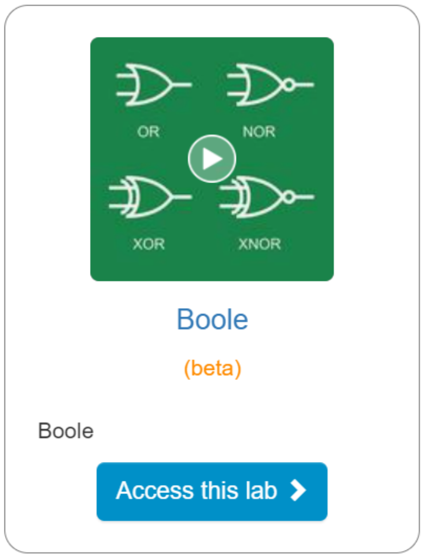


Figure 14: Boole Feature

2. Click on "Access" under "Boole IDE."

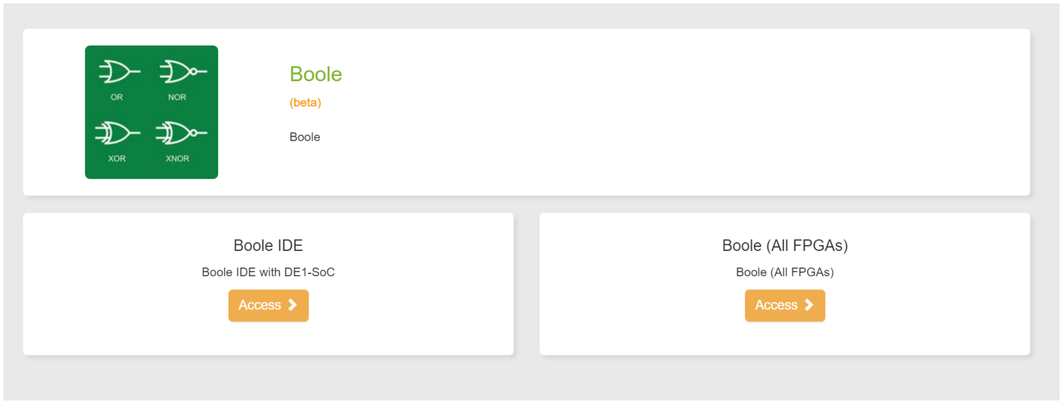
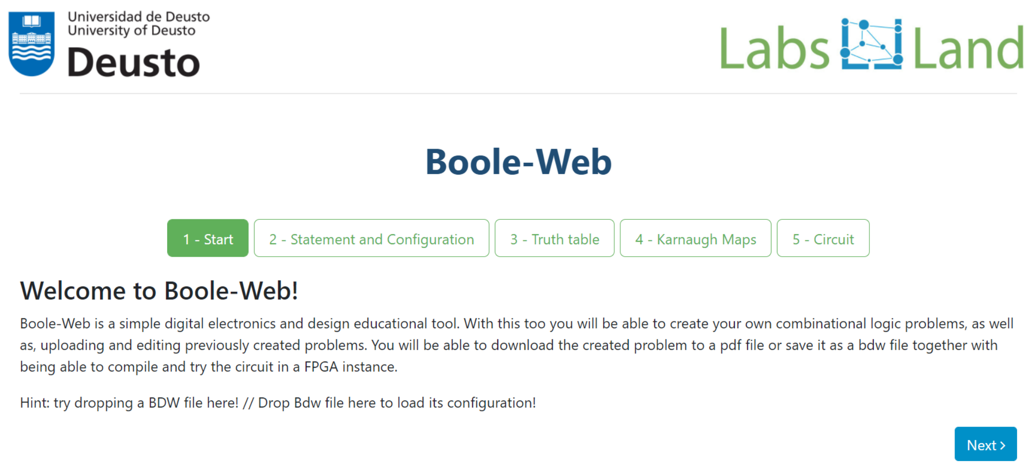


Figure 15: Boole IDE

3. You will see the Boole-Web welcome page as shown in Figure 16. Read the intro, and press "Next." You should now see the screen shown in Figure 17.



Universidad de Deusto
University of Deusto
Deusto

Labs Land

Boole-Web

1 - Start | 2 - Statement and Configuration | 3 - Truth table | 4 - Karnaugh Maps | 5 - Circuit

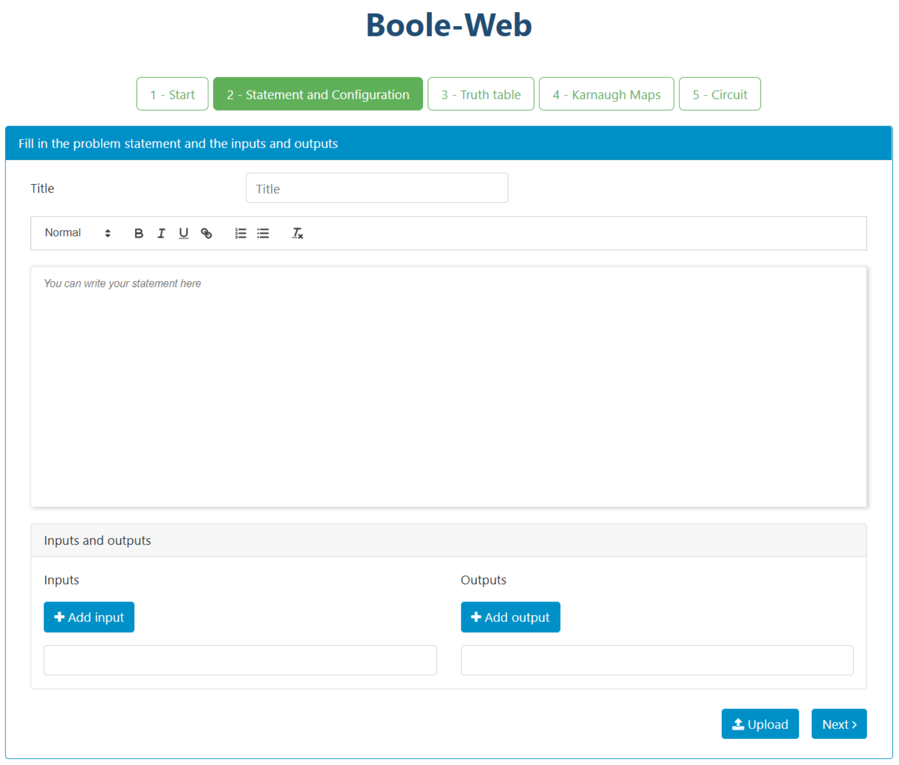
Welcome to Boole-Web!

Boole-Web is a simple digital electronics and design educational tool. With this too you will be able to create your own combinational logic problems, as well as, uploading and editing previously created problems. You will be able to download the created problem to a pdf file or save it as a bdw file together with being able to compile and try the circuit in a FPGA instance.

Hint: try dropping a BDW file here! // Drop Bdw file here to load its configuration!

Next ›

Figure 16: Boole-Web Welcome Page



Boole-Web

1 - Start | 2 - Statement and Configuration | 3 - Truth table | 4 - Karnaugh Maps | 5 - Circuit

Fill in the problem statement and the inputs and outputs

Title: Title

You can write your statement here

Inputs and outputs

Inputs: Add input

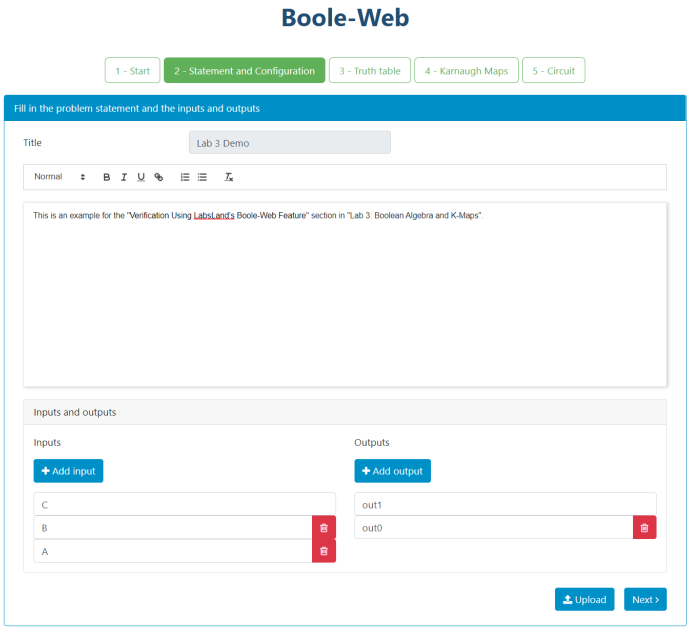
Outputs: Add output

Upload | Next ›

Figure 17: Blank Statement and Configuration Page

4. Add a project title and description in the Statement and Configuration Page (Figure 17). This is for documentation purposes.

5. Add the inputs and outputs of your system. For example, given a 3-input system with inputs 'A', 'B', and 'C' and 2 outputs 'Out1' and 'Out0', click 'Add input' twice and 'Add output' once. Enter the appropriate input and output names in each blank. Ensure that the topmost name is the rightmost variable in your TruthTable; in this case, that is input C and output out1. Press 'Next' when finished, as depicted in Figure 18.



Fill in the problem statement and the inputs and outputs

Title: Lab 3 Demo

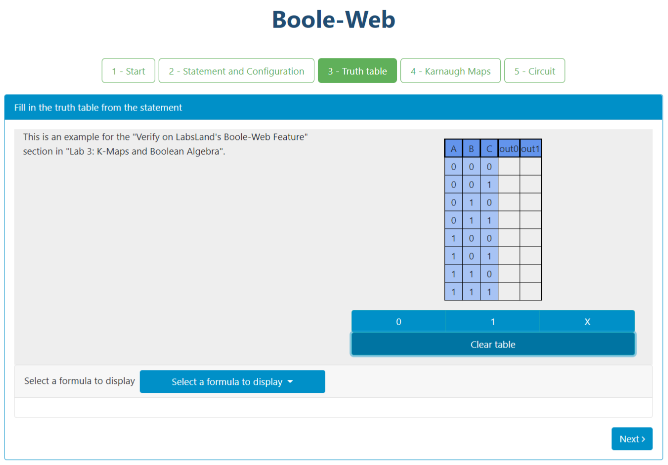
This is an example for the "Verification Using LabsLand's Boole-Web Feature" section in "Lab 3: Boolean Algebra and K-Maps".

Inputs and outputs

Inputs	Outputs
C	out1
B	out0
A	

Figure 18: Statement and Configuration Page with Data

6. You should now see your project description in the top left with a K-Map missing its outputs in the top right. Note how the rows of the K-Map have its inputs auto-filled.



Fill in the truth table from the statement

This is an example for the "Verify on LabsLand's Boole-Web Feature" section in "Lab 3: K-Maps and Boolean Algebra".

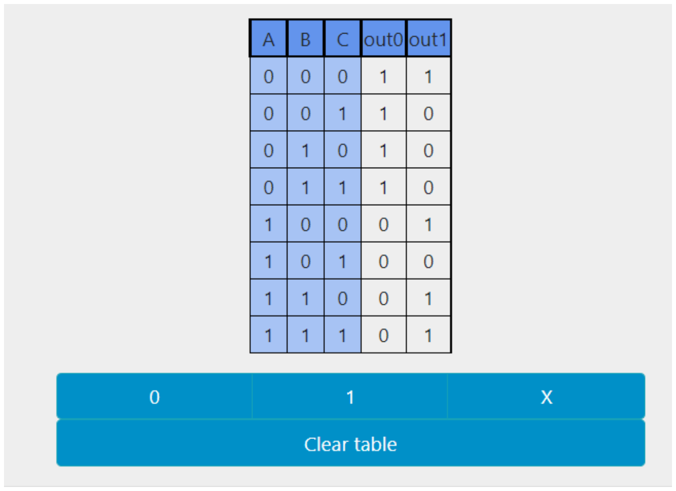
A	B	C	out0	out1
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Select a formula to display

Figure 19: Statement and Configuration Page with Data

7. Fill in out0 and out1 according to your observed I/O behaviors. notice the shortcuts below the Truth Table: 'Clear Table', '0' (fill all remaining output rows with 0) , '1' (fill all remaining output rows with 1) , 'X' (fill all remaining output rows with X which means "Don't Care" - something we'll learn about in Lab 5).

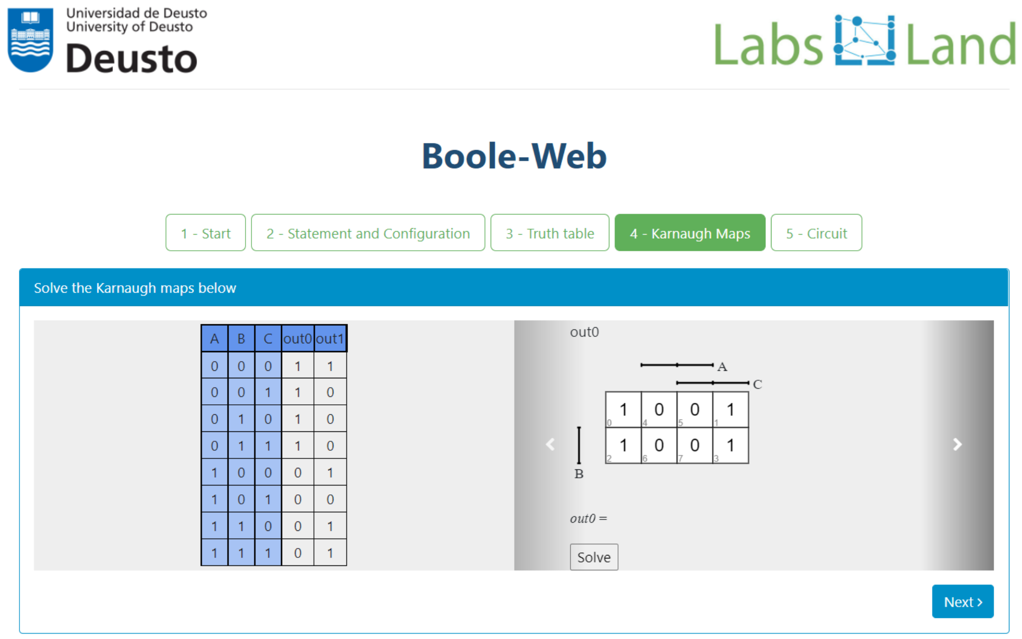
   A Truth Table may look like the example in Figure 20. Press 'Next' when finished.



A	B	C	out0	out1
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

Figure 20: Boole-Web Truth Table with Data

8. You should see a page as shown in Figure 21 with the K-Maps drawn for each of your outputs. Press the left and right arrows highlighted in gray to scroll through the K-Maps.



Solve the Karnaugh maps below

A	B	C	out0	out1
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

out0

1	0	0	1
1	0	0	1

Figure 21: Boole-Web K-Map Page

9. Press ‘Solve’ to see the resulting Boolean Expressions from each K-Map. Boole-Web solves the grouping and simplification for you!

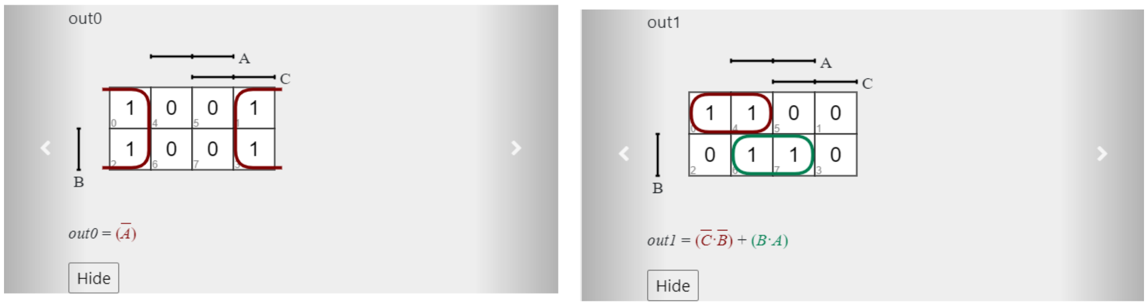
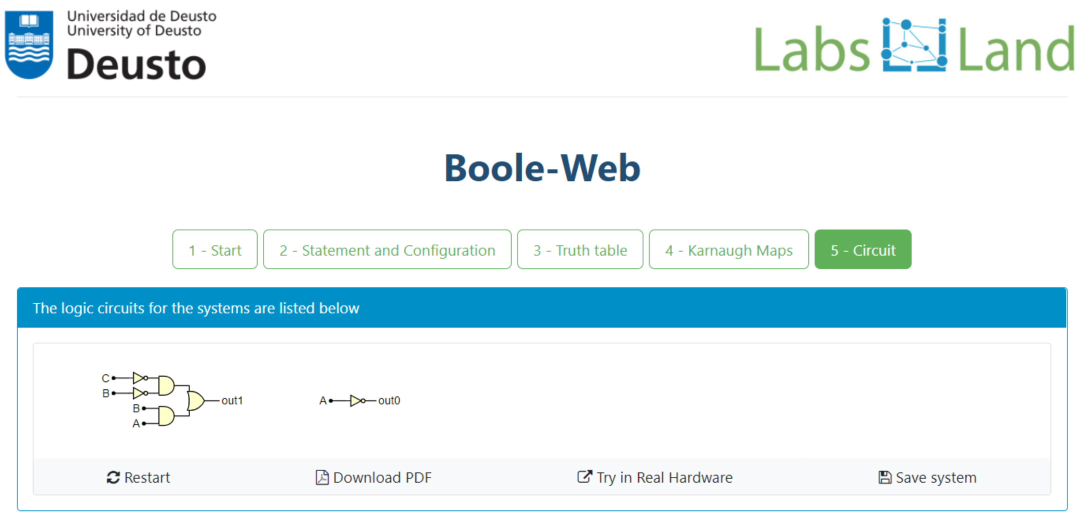


Figure 22: Solved K-Maps in Boole-Web

10. To see the schematic of gates which are built according to the Boolean expressions found by Boole-Web, press “Next.” You should see the screen as shown in Figure 22 with a combination of logic gates for your system.



The logic circuits for the systems are listed below

Figure 23: Circuit Schematics in Boole-Web

11. Repeat steps 4-10 for each of the Digital Trainer challenges from Labs 1 and 2. Verify that the resulting K-Maps and Boolean equations you found in Tables 1-10 are equivalent to those computed by Boole-Web. Use the “Recognizing Patterns” section in Lab 3 to write down your observations.

12. OPTIONAL: Note that Boole-Web has many features which are not covered by this Lab but you are free to explore. If you’re curious, under tab “3-Truth table, look at the options under “Select a formula to display.” What do you notice for each option? In tab “5-Circuit’, check out “Try in Real Hardware.” Can you connect signals to I/O on an FPGA to observe real-world behavior? Also, take a look at the “Download PDF” and “Save system” features for easy access outside of LabsLand.

Recognizing Patterns

Compare your K-Maps and Boolean equations in Tables 1-10 to the K-Maps and Boolean Equations computed by Boole-Web. If your guess matches, compare your initial reasoning to Boole-Web's process. If the K-Maps or equations do not match, revisit the LabsLand Digital Trainer and double check the system behavior. Describe what you observe.

Table 11: K-Map Reasonings

Challenge	Operator Guess	Did your guess match Boole-Web? Why or why not? (1-2 sentences)
1		
2		
3		
4		
5		

6		
7		
8		
9		
10		

Looking Ahead

This lab wraps up the fundamental topics covered in this lab series for understanding digital logic. Now that you know about Boolean Algebra and K-Maps, we can look into applying it in real systems. The following lab will shift to introducing computer tools which will help us visualize the digital logic concepts we've been discussing. Future labs will call on all the foundational knowledge (Boolean Algebra, K-Maps, Truth Tables, Binary Numbers, Gates, and Operators) we've built so far.

Reflection and Observations

Lab 3 introduced you to K-Maps and Boolean algebra. Reflect on the new topics and the things you found interesting and/or challenging in the space below.

What questions do you still have, if any?

ELECTRICAL & COMPUTER ENGINEERING

Lab 4: FPGA INTERACTIONS AND SYSTEM VERILOG

The University of Washington | The Remote Hub Lab | Last Revised: March 2022

Summary

Building on the background knowledge gained from the previous labs, Lab 4 introduces the Hardware Description Language (HDL) SystemVerilog. In this lab, you will learn about 1) Quartus for SystemVerilog synthesis, 2) ModelSim for testing and simulation, and 3) Writing code to Field Programmable Gate Arrays (FPGAs).

Table of Contents

What is a Field Programmable Gate Array (FPGA)?

FPGAs are hardware chips with programmable logic cells -- electrical components which a designer can use to customize and configure circuits for specific purposes. Digital circuits typically consist of a large amount of logic components which necessitates a way to reconfigure circuits on scale, and this is why FPGAs are used.

To program an FPGA, we need to use a Hardware Description Language (HDL). This lab series uses an HDL called SystemVerilog, but note that there are other HDLs such as VHDL. After writing and synthesizing the code, it is good practice to simulate the output. This allows designers to check if the code results in what they expect before sending it to the FPGA. Sending a design to an FPGA means converting the HDL code into bitstreams (0s and 1s) that are understandable by the machine. This workflow is demonstrated in Figure 1.

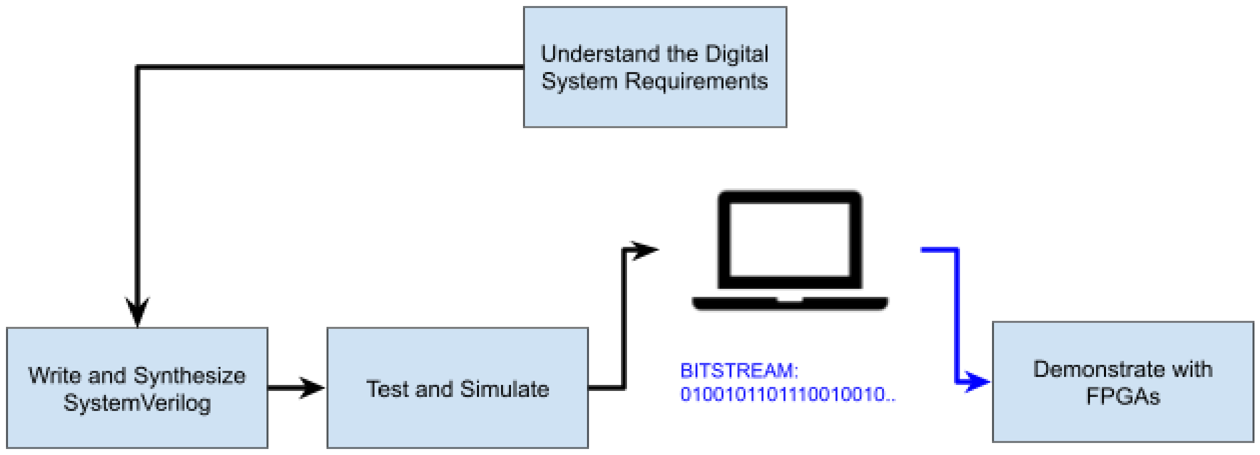

Understand the Digital System Requirements
Write and Synthesize SystemVerilog
Test and Simulate
BITSTREAM: 0100101101110010010..
Demonstrate with FPGAs

Figure 1: Understanding → SystemVerilog → Synthesis → Simulation→ FPGA Workflow

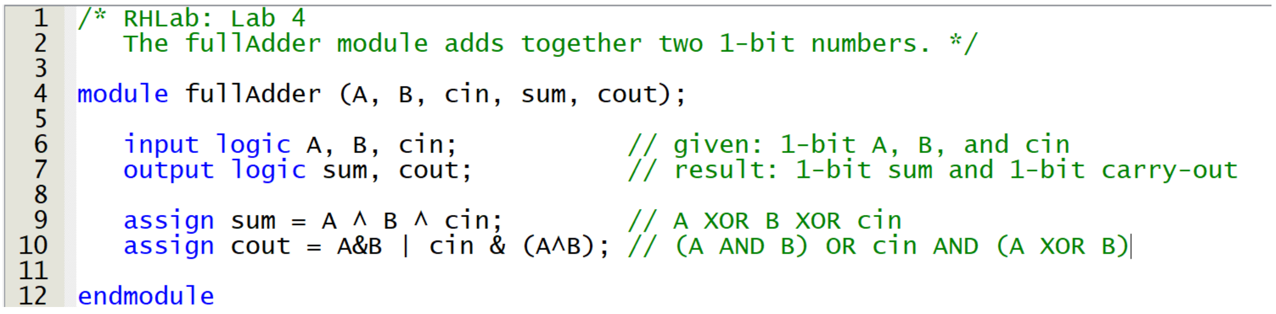
In this lab, we'll walk through each step of the workflow as depicted in Figure 1. As in the Digital Trainer Activity, we'll use LabsLand to access physical FPGAs remotely. We'll also use ModelSim for simulation and Quartus Prime Software to write and synthesize code. But what code should we write and how do we write it? To answer this question, let's look at the basics of SystemVerilog.

Implementing Digital Logic Using a Hardware Description Language

Description

In Labs 0-3, you gained skills to define various digital systems on paper. How do we do this on a computer? That's where a Hardware Description Language (HDL) like SystemVerilog comes in. Creating a digital design involves three steps: Describing the design using HDL with a software like Quartus, verifying the design with a tool like ModelSim, and then implementing the verified design on an FPGA. In this section, we will introduce the major SystemVerilog Syntax through a full adder example which will suffice for the needs of this lab. Interested readers can also find more resources about SystemVerilog on the Internet.

A full adder performs bitwise addition using two bits (A, B) and a carry-in value (cin), resulting in sum and carry-out (cout) outputs. Figure 2 shows an example of a SystemVerilog program which creates a full adder unit.



```
 1  /* RHLab: Lab 4
 2     The fullAdder module adds together two 1-bit numbers. */
 3
 4  module fullAdder (A, B, cin, sum, cout);
 5
 6     input logic A, B, cin;          // given: 1-bit A, B, and cin
 7     output logic sum, cout;         // result: 1-bit sum and 1-bit carry-out
 8
 9     assign sum = A ^ B ^ cin;       // A XOR B XOR cin
10     assign cout = A&B | cin & (A^B); // (A AND B) OR cin AND (A XOR B)
11
12  endmodule
```

Figure 2: Full Adder Module

SystemVerilog Syntax and Main Components

general notes

- Notice how most lines in SystemVerilog except for "endmodule" end with a semicolon (**;**).
- Notice the indents and white space left between lines for readability.

modules

- Think of this like the "container" for your code. A module begins with the starting line syntax "module [name()]". A module ends with the line "endmodule."
- See lines 4 and 12 in Figure 2.

input and output logic

- These lines are akin to declaring your variables. You list your input signals after the syntax “input logic” , and you list your output signals after the syntax “output logic” respectively.
- See lines 6 and 7 in Figure 2.

assign statements

- assign statements define the variables’ meaning with Boolean expressions.
- See Figure 2’s lines 9-10 which use assign statements as supposed to lines 6-7’s variable declarations.

comments

- Leave comments in your code for readability. Use one double forward slash at the beginning of in-line comments (//) and a single forward slash and star at the beginning and end of multi-line comments (/*...*/).
- In Figure 2, see lines 6-10 for in-line comments and lines 1-2 for multi-line comments.

Defining Logic functions with Bitwise operators

- Bitwise operators are used to define logic functions such as the sum and cout equations in lines 9 and 10 of figure 2. Table 1 summarizes the bitwise operators used in SystemVerilog

***Table 1**: Bitwise Operators*

Operator	Syntax Symbol	Boolean Expression → SystemVerilog Example
AND	&	$A\ AND\ B \rightarrow A\ \&\ B$
OR	|	$A\ OR\ B \rightarrow A \mid B$
NOT	~	$\overline{A} \rightarrow \sim A$
NOR	(combination)	$A\ NOR\ B \rightarrow \sim(A \mid B)$
NAND	(combination)	$A\ NAND\ B \rightarrow \sim(A\ \&\ B)$
XOR	^	$A\ XOR\ B \rightarrow A\ \hat{}\ B$

Note: Do not use ‘+’ for the OR operator, and do not use ‘x’ for the AND operator in SystemVerilog code. They mean different things in SystemVerilog.

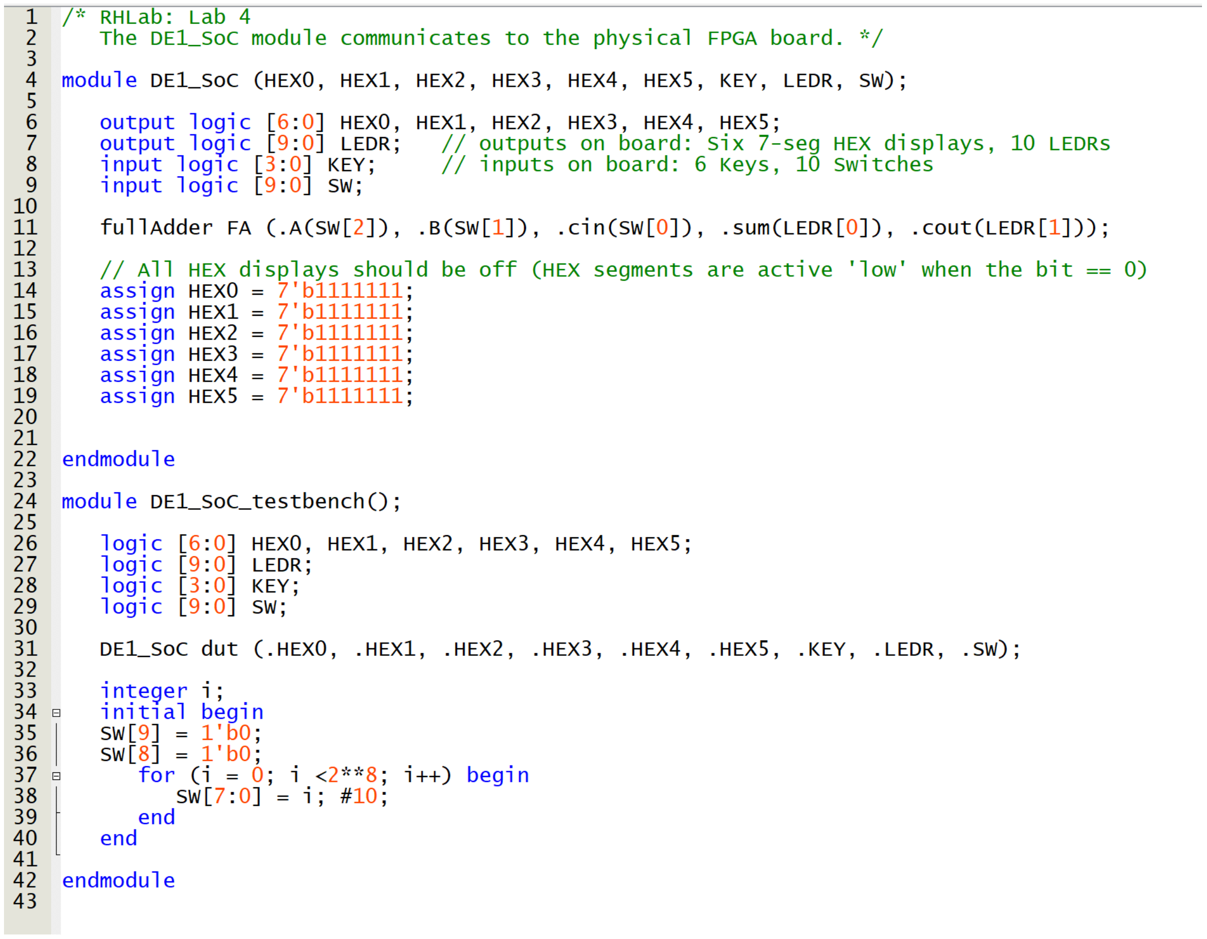
Quartus, ModelSim, and LabsLand Activity

Now that you have some background on Quartus, ModelSim, and LabsLand, we can jump into a practice exercise. The following steps will provide a walkthrough of the Understanding → SystemVerilog → Synthesis → Simulation → FPGA workflow using the full adder example.

1. **Understand the system requirements.**
   a) Before coding, we need to understand the system we will be implementing.
   b) One way to accomplish this first step is on paper (as we've seen in Labs 0-3). Another way is to draw a schematic diagram of the system.
   c) To help you understand the full adder:
      i) Open the project skeleton you created in Lab 0.
      ii) Follow this video tutorial to create a schematic diagram of a full adder's components and logic gates: https://youtu.be/qn6ggwxpDjQ?t=86.
      iii) Note that the video skips to timestamp 1:26 since you already created a project skeleton.

2. **Practice writing SystemVerilog in Quartus, synthesizing your system in Quartus, and simulating your system in ModelSim.**
   a) Follow the instructions in this video tutorial to design and simulate the full adder in Quartus and ModelSim: https://www.youtube.com/watch?v=BcvclrqZ2fc.
   b) This tutorial will help you implement the SystemVerilog code in Figure 2 in addition to writing a verification code to simulate the design in ModelSim.
   c) Note that it is intentional that the code in figure 2 is not "copy-and-paste-able" to Quartus; you must type out the syntax of the code to gain familiarity with the (HDL).

3. **Prepare your code for the LabsLand FPGA.**
   a) Follow the instructions in this video tutorial to create and simulate a SystemVerilog module called "DE1_SoC" for your FPGA: https://www.youtube.com/watch?v=mnZt2iNNfp4.
   b) Refer to Figure 3 to view the DE1_SoC code for both the design module and testbench.
   c) Note that this tutorial helps you make another module which instantiates the full adder system so we can interact with FPGA inputs and outputs. This hierarchical structure is a common feature of HDL. Because we are making another module, we must synthesize and simulate at this higher level as well.

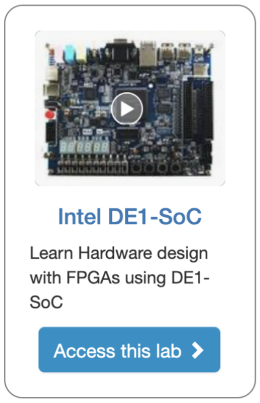


```
/* RHLab: Lab 4
   The DE1_SoC module communicates to the physical FPGA board. */

module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);

   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0] LEDR;    // outputs on board: Six 7-seg HEX displays, 10 LEDRs
   input logic [3:0] KEY;      // inputs on board: 6 Keys, 10 Switches
   input logic [9:0] SW;

   fullAdder FA (.A(SW[2]), .B(SW[1]), .cin(SW[0]), .sum(LEDR[0]), .cout(LEDR[1]));

   // All HEX displays should be off (HEX segments are active 'low' when the bit == 0)
   assign HEX0 = 7'b1111111;
   assign HEX1 = 7'b1111111;
   assign HEX2 = 7'b1111111;
   assign HEX3 = 7'b1111111;
   assign HEX4 = 7'b1111111;
   assign HEX5 = 7'b1111111;


endmodule

module DE1_SoC_testbench();

   logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic [9:0] LEDR;
   logic [3:0] KEY;
   logic [9:0] SW;

   DE1_SoC dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW);

   integer i;
   initial begin
   SW[9] = 1'b0;
   SW[8] = 1'b0;
      for (i = 0; i <2**8; i++) begin
         SW[7:0] = i; #10;
      end
   end

endmodule
```

Figure 3: DE1_SoC Module and Testbench

4. Demonstrate the digital system on an FPGA.

a) Login to LabsLand, navigate to your main dashboard with the RHLab activities, and choose “Intel DE1-SoC”. Click the “Access this Lab” button.

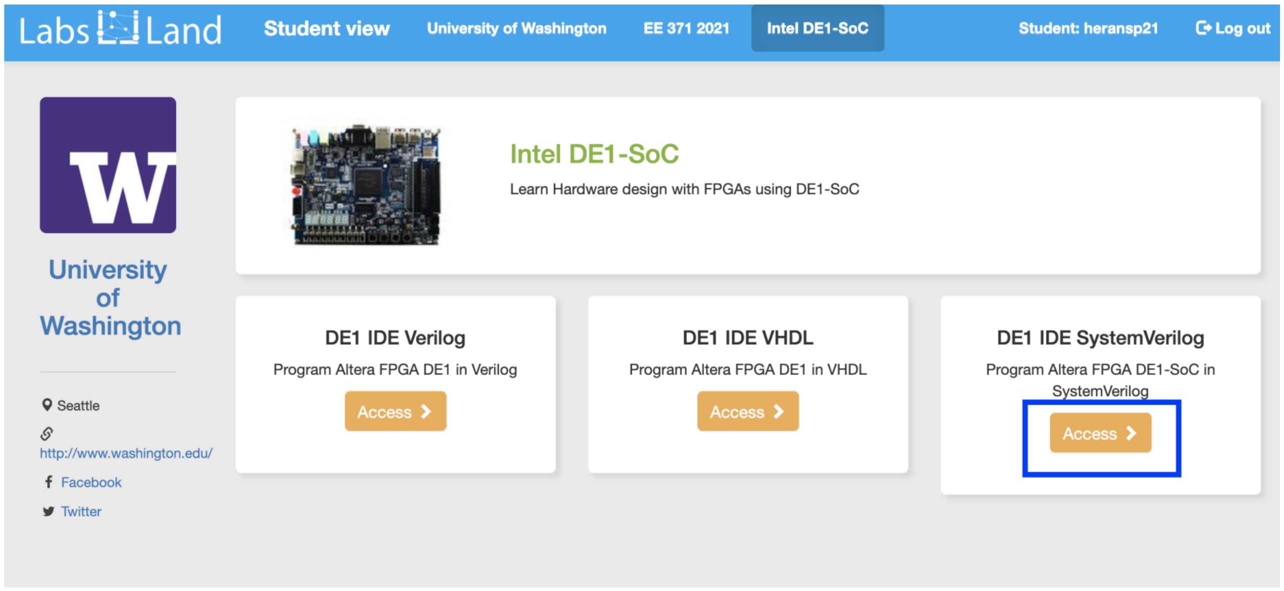


Intel DE1-SoC

Learn Hardware design with FPGAs using DE1-SoC

Figure 4: Intel DE1-SoC Activity

b) Locate “DE1 IDE SystemVerilog” and click the “Access” button below it. You will be directed to a new page called “SystemVerilog IDE for DE1-Soc”.



Intel DE1-SoC

Learn Hardware design with FPGAs using DE1-SoC

DE1 IDE Verilog

Program Altera FPGA DE1 in Verilog

DE1 IDE VHDL

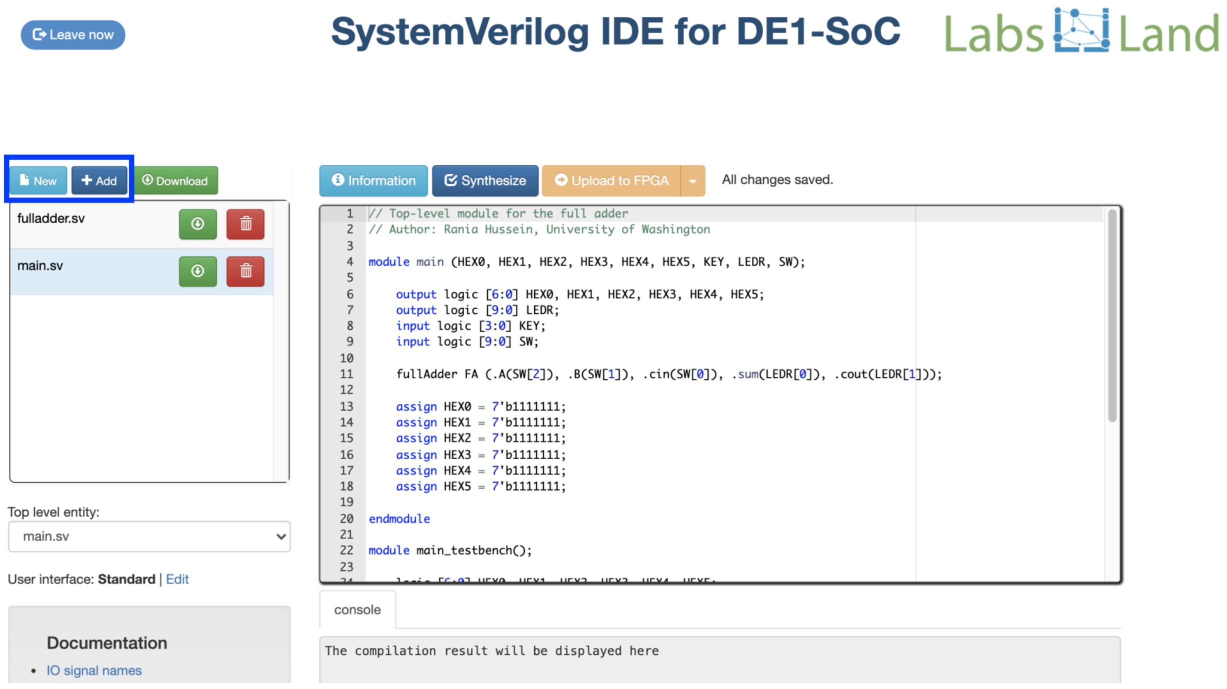
Program Altera FPGA DE1 in VHDL

DE1 IDE SystemVerilog

Program Altera FPGA DE1-SoC in SystemVerilog

Figure 5: DE1 SystemVerilog IDE

c) In the following page, select the “Add” button to import the top-module “DE1-SoC.sv” and file “full_adder.sv” that you created earlier into LabsLand.



Top level entity: main.sv

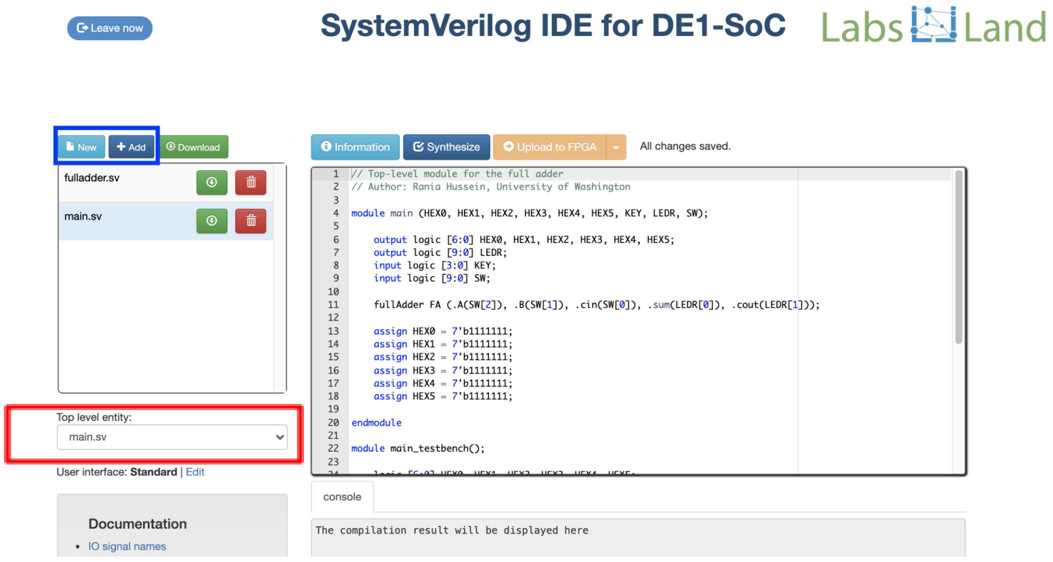
User interface: **Standard** | Edit

```
// Top-level module for the full adder
// Author: Rania Hussein, University of Washington

module main (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);

    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    output logic [9:0] LEDR;
    input logic [3:0] KEY;
    input logic [9:0] SW;

    fullAdder FA (.A(SW[2]), .B(SW[1]), .cin(SW[0]), .sum(LEDR[0]), .cout(LEDR[1]));

    assign HEX0 = 7'b1111111;
    assign HEX1 = 7'b1111111;
    assign HEX2 = 7'b1111111;
    assign HEX3 = 7'b1111111;
    assign HEX4 = 7'b1111111;
    assign HEX5 = 7'b1111111;

endmodule

module main_testbench();

```

```
The compilation result will be displayed here
```

Figure 6: Add Modules into LabsLand

d) Choose the top-module using the dropdown menu under “Top level entity” (boxed in red in Figure 7). Make sure you select “DE1-SoC.sv” as the top-module.



Top level entity: main.sv

User interface: Standard | Edit

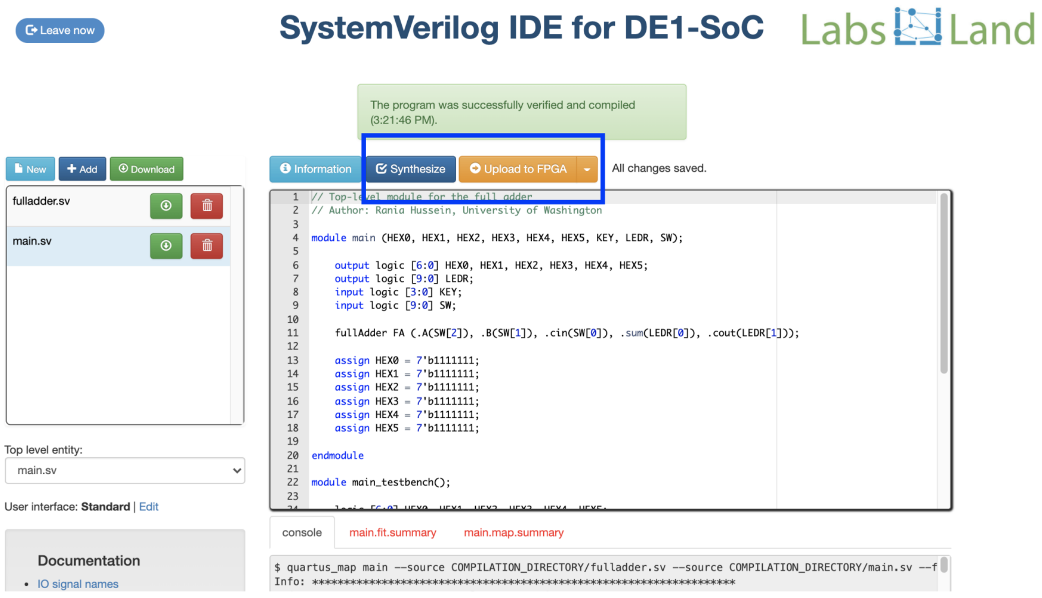
```
// Top-level module for the full adder
// Author: Rania Hussein, University of Washington

module main (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);

    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    output logic [9:0] LEDR;
    input logic [3:0] KEY;
    input logic [9:0] SW;

    fullAdder FA (.A(SW[2]), .B(SW[1]), .cin(SW[0]), .sum(LEDR[0]), .cout(LEDR[1]));

    assign HEX0 = 7'b1111111;
    assign HEX1 = 7'b1111111;
    assign HEX2 = 7'b1111111;
    assign HEX3 = 7'b1111111;
    assign HEX4 = 7'b1111111;
    assign HEX5 = 7'b1111111;

endmodule

module main_testbench();
```

```
The compilation result will be displayed here
```

Figure 7: Setting the Top Level Module

(Alternatively, you may also create new files by clicking “New” and copy the provided full adder example under “Examples” found in the bottom left corner of the interface into the corresponding new files. The top-module is named “main.sv” in this case, so make sure to adjust the settings accordingly.)

e) You will then be able to synthesize the code using the button “Synthesize”. Once the synthesis is complete and succeeds without errors, you can click on “Upload to FPGA” to load your design onto an FPGA.



The program was successfully verified and compiled (3:21:46 PM).

Top level entity: main.sv

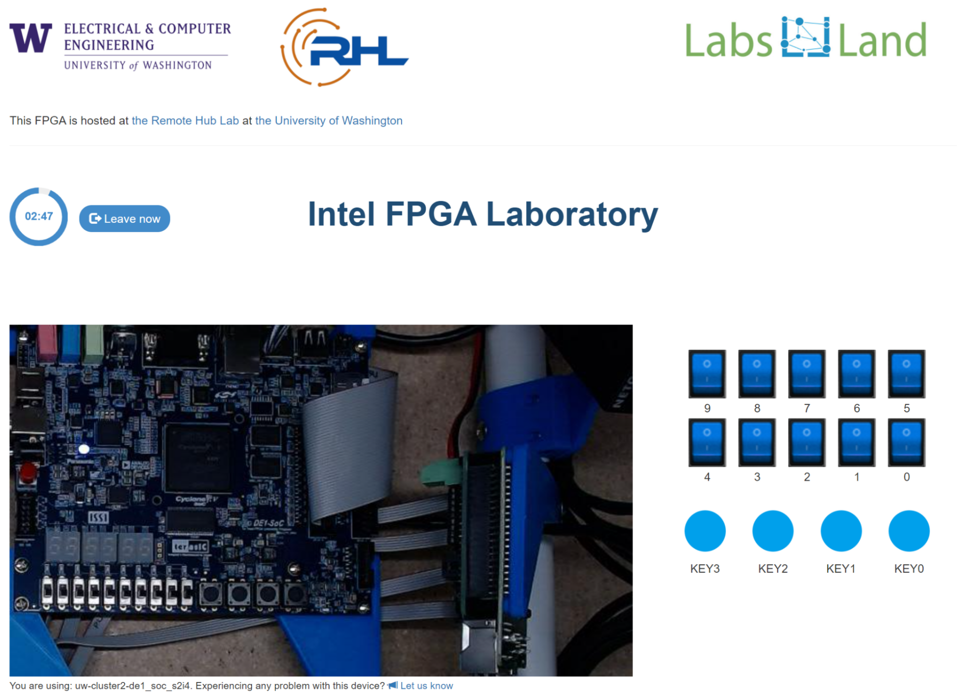
User interface: Standard | Edit

```
// Top-level module for the full adder
// Author: Rania Hussein, University of Washington

module main (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);

    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    output logic [9:0] LEDR;
    input logic [3:0] KEY;
    input logic [9:0] SW;

    fullAdder FA (.A(SW[2]), .B(SW[1]), .cin(SW[0]), .sum(LEDR[0]), .cout(LEDR[1]));

    assign HEX0 = 7'b1111111;
    assign HEX1 = 7'b1111111;
    assign HEX2 = 7'b1111111;
    assign HEX3 = 7'b1111111;
    assign HEX4 = 7'b1111111;
    assign HEX5 = 7'b1111111;

endmodule

module main_testbench();
```

```
$ quartus_map main --source COMPILATION_DIRECTORY/fulladder.sv --source COMPILATION_DIRECTORY/main.sv --f
Info: *******************************************************************
```

Figure 8: Synthesizing on LabsLand

f) After connecting to the remote FPGA, you will see a webpage like Figure 9's.



This FPGA is hosted at the Remote Hub Lab at the University of Washington

Intel FPGA Laboratory

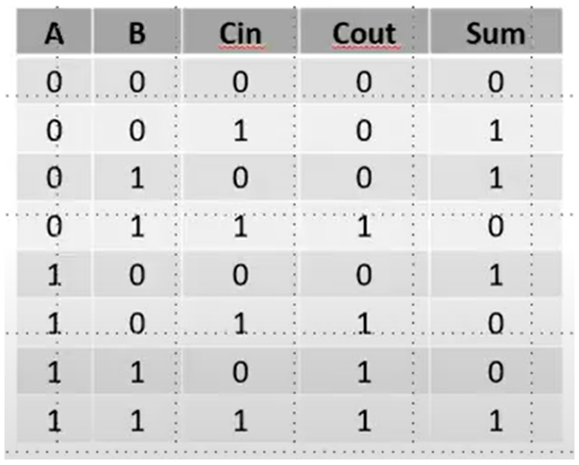
You are using: uw-cluster2-de1_soc_s2i4. Experiencing any problem with this device? Let us know

Figure 9: An FPGA on LabsLand

The right part of the page shows the buttons and keys of the FPGA. You can click on the buttons and keys accordingly as inputs. It is important to note that 'KEYS' need to be held down, as they do not function like switches.

g) Toggle Switch2 (A), Switch1 (B), and Switch0 (Cin) to test different input combinations. Check that your system in LabsLand outputs signals using LEDR1 (Cout) and LEDR0 (Sum) which are consistent with the Truth Table in Table 2.

Table 2: Full Adder Truth Table



A	B	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Reflection and Observations

Write down your initial thoughts about the Understanding → SystemVerilog → Synthesis → Simulation → FPGA workflow. What did you observe throughout the process?

What did you notice about the SystemVerilog code that you'd like to learn more about?

What questions do you have, if any?

ELECTRICAL & COMPUTER ENGINEERING

Lab 5: SEQUENTIAL LOGIC AND FINITE STATE MACHINES

The University of Washington | The Remote Hub Lab | Last Revised: March 2022

Summary

This lab introduces sequential logic and Finite State Machines (FSMs) both in theory and in SystemVerilog. The lab walks through a full system design problem, analyzing a problem statement and demonstrating behaviors on FPGAs using SystemVerilog.

Table of Contents

What is a Finite State Machine (FSM)?

Assume we have a digital system which is an automatic door with two states: open and closed. If there is movement near the door (input is true), we want the system to be in the “open” state. When there is no movement nearby (input is false), we want the output to be “closed.” We can describe this system visually with a “state diagram”, as in Figure 1.

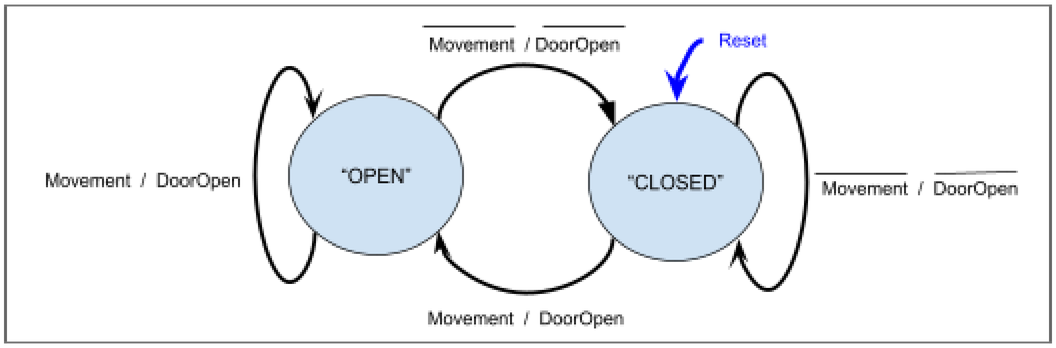

$\overline{Movement}$ / $\overline{DoorOpen}$
Reset
Movement / DoorOpen
"OPEN"
"CLOSED"
$\overline{Movement}$ / $\overline{DoorOpen}$
Movement / DoorOpen

***Figure 1**: Automatic Door State Diagram*

This example depicts the Finite State Machine (FSM): a state diagram with a finite number of states (two in this case). There are four important things to consider in the FSM:

1. **There is a Reset signal which indicates where our system should start after a reboot.**

2. **Each state has an arrow for every input possibility.**
   - In Figure 1, the arrows between states describe when changes occur. Because there is only one significant input signal (Movement), there are only two possibilities in each state: ON (Movement is true) and OFF (Movement is false).
   - As a result, there are two arrows which originate from the “open” state and two arrows which originate from the “closed” state.

3. **Every arrow is labeled with an Input/Output (I/O) behavior.**
   - In Figure 1, the arrow starting from the “open” state pointing back to itself demonstrates the following behavior: when there is movement, the door stays open (when Movement is true, the output DoorOpen is true).
   - The other arrow starting from the “open” state going to the “closed state” demonstrates the other behavior: when there is no movement, the door closes (when Movement is false, the output DoorOpen is false and we change states).

4. **Because we define there to be only one input (Movement) and one output (DoorOpen), the I/O pattern can be expressed with 1-bit binary. Because there are only two states, we can also assign distinct encodings to the states in binary.**
   - Figure 2 is a simpler (and more common) way to draw an equivalent FSM.

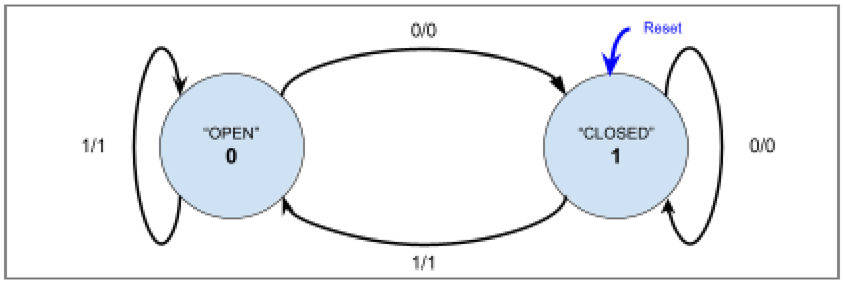


***Figure 2**: Simpler Automatic Door State Diagram*

How does a system remember the information it needs to transition between states? In Labs 0-4, we focused on combinational logic which is good for expressing output behaviors as one Boolean expression. Recall the "mystery box" model from Lab 1, provided in Figure 3.

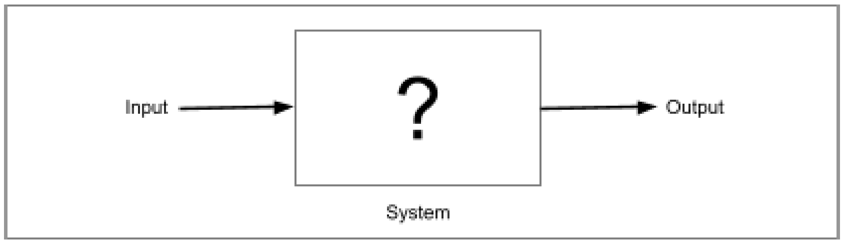


***Figure 3**: Input/Output System*

However, if we want to define a signal that has memory of what happened before a change, then we want to understand sequential logic. In this case, we have a new model in Figure 4.

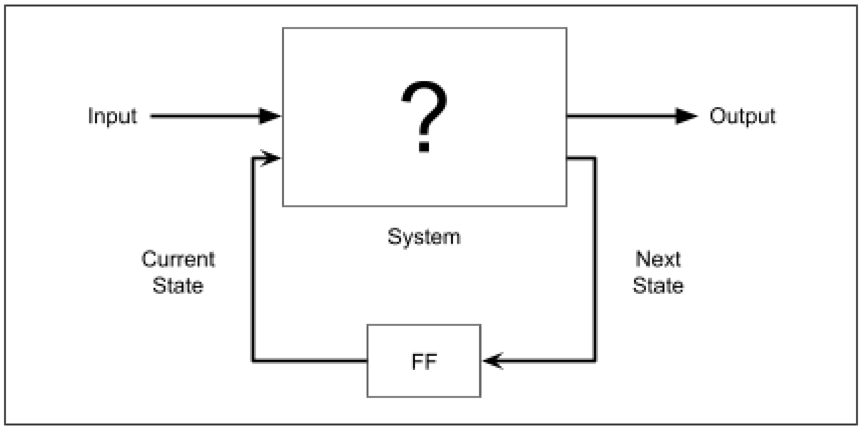


***Figure 4**: Input/Output System with a Flip Flop for Feedback*

Notice how there is a new box called "FF" (representing a flip-flop) which allows the system to remember states even as time progresses. To express this added information with the Truth Table, we must add a Present State input column and a Next State output column in addition to the normal I/O signals. A State Table for the automatic door FSM is provided in Table 1.

***Table 1**: Automatic Door State Table*

PresentState	Movement	DoorOpen	NextState
0	0	0	1
0	1	1	0
1	0	0	1
1	1	1	0

Table 1 is a direct translation of the state diagram in Figure 2. First, note that our "Truth Table" is now called a "State Table." Also, notice that there is an input column "Movement" and an output column "DoorOpen" for our input and output signals respectively. The "Present State" input column refers to our state encoding: 0 for "Open" and 1 for "Closed." So what does each row in Table 1 mean?

```
   // Abbreviations: PS = Present State; NS = Next State
1. When the door is open              →    // PS = Open = 0
   and there is no movement,          →    // Movement = 0
   the door should close and          →    // DoorOpen = 0
   go to the "Closed" state.          →    // NS= Closed = 1

2. When the door is open              →    // PS = Open = 0
   and there is movement,             →    // Movement = 1
   the door should be open            →    // DoorOpen = 1
   and stay in the "Open" state.      →    // NS= Open = 0

3. When the door is closed            →    // PS = Closed = 1
   and there is no movement,          →    // Movement = 0
   the door should be closed          →    // DoorOpen = 0
   and stay in the "Closed" state.→        // NS = Closed = 1

4. When the door is closed            →    // PS = Closed = 1
   and there is movement,             →    // Movement = 1
   the door should open               →    // DoorOpen = 1
   and go to the "Open" state.        →    // NS = Open = 0
```

Creating K-Maps for the DoorOpen and NextState signals is left as an exercise for the reader. However, it is important to note that this process will allow you to arrive at Boolean expressions for the output signals, and these expressions are programmable in SystemVerilog code.

FSM SystemVerilog Structure

An FSM in SystemVerilog has several new components in addition to those in Lab 4's "SystemVerilog Syntax and Main Components" section. Figure 5 shows an example SystemVerilog module for our automatic door FSM. Figure 6 and Figure 7 provide the testbench and waveform respectively.

clock and reset

- The clock and reset input signals are needed for sequential logic and flip-flops.
- In the automatic door module (Figure 5), these signals are declared as ports (line 4 and 6) and used in the always_ff block (lines 25-30).
- In the automatic door testbench (Figure 6), lines 44-48 setup the clock, and lines 51-66 simulate the reset and clock signals to test system behavior.

enumerate states

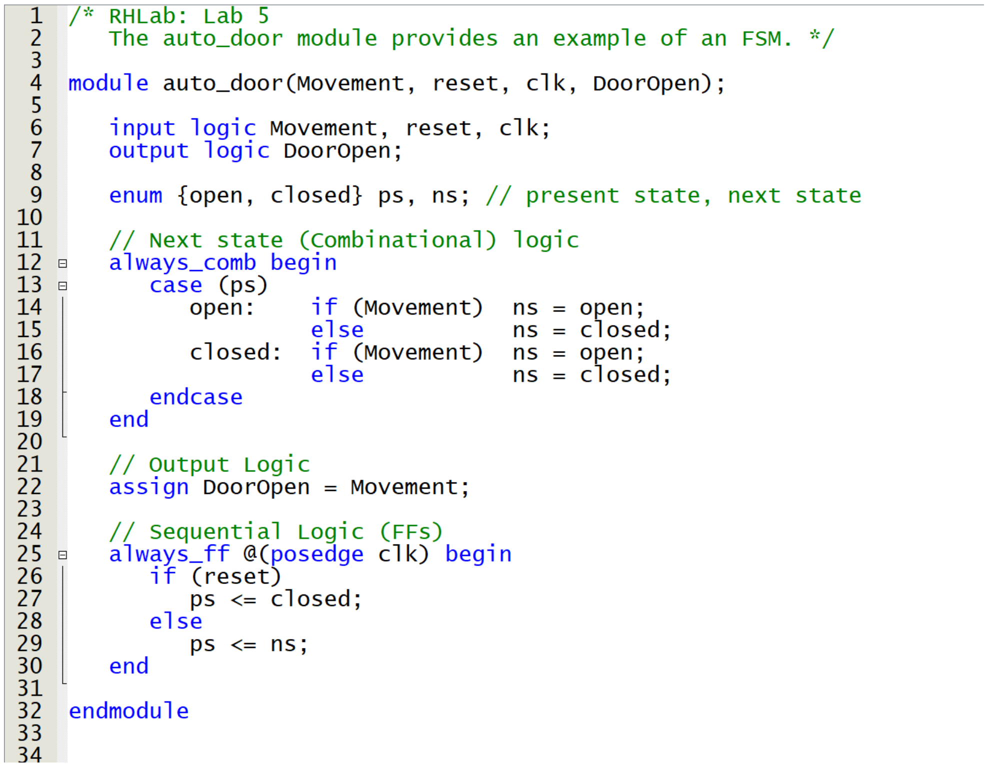
- This enumeration lists the states of your FSM.
- Line 9 in Figure 5 is an example of this enumeration. Notice the special syntax.

always_comb block

- This block encapsulates your next state logic which is combinational. As a result, this block relies on the present state and next state cases you defined in your FSM.
- Figure 5's lines 12-19 is an example of the always_comb block. Note that our convention is to include "case" and "endcase" syntax (lines 13 and 18).

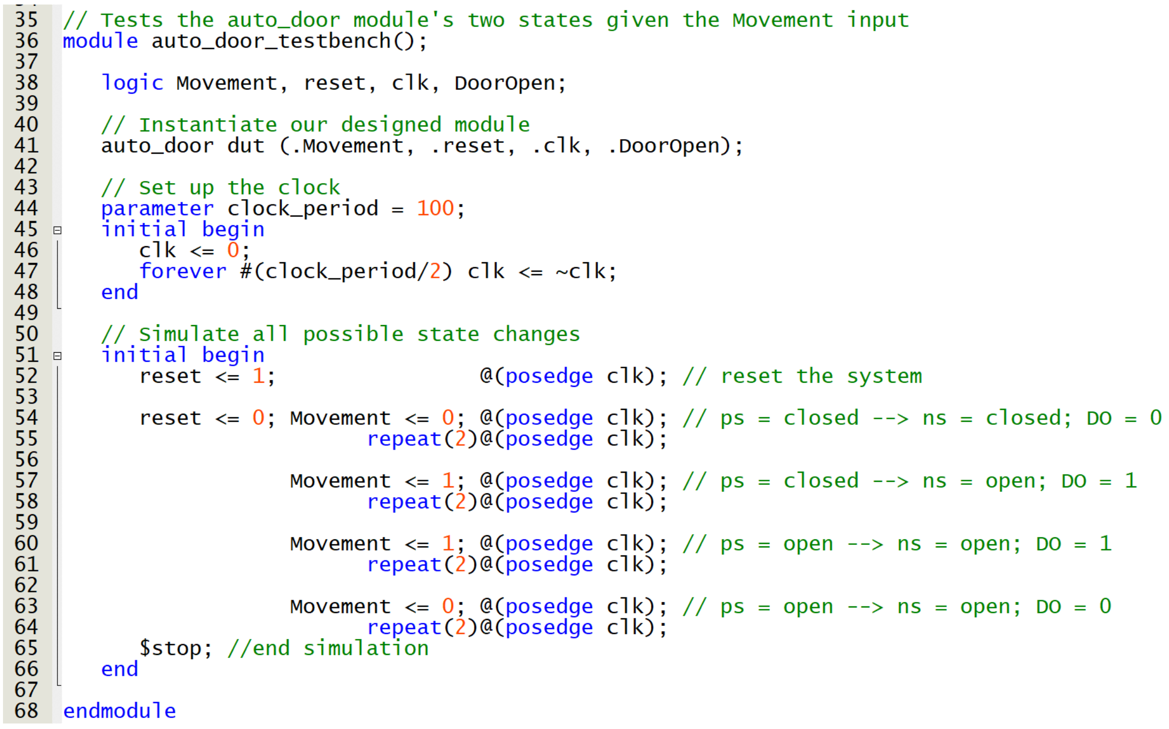
always_ff block

- This block encapsulates your sequential logic. The mechanics of this block are out of the scope of this lab series, but know that it drives the movement between present states and next states every clock cycle.
- See lines 25-30 in Figure 5.



```
/* RHLab: Lab 5
   The auto_door module provides an example of an FSM. */

module auto_door(Movement, reset, clk, DoorOpen);

   input logic Movement, reset, clk;
   output logic DoorOpen;

   enum {open, closed} ps, ns; // present state, next state

   // Next state (Combinational) logic
   always_comb begin
      case (ps)
         open:     if (Movement)  ns = open;
                   else           ns = closed;
         closed:   if (Movement)  ns = open;
                   else           ns = closed;
      endcase
   end

   // Output Logic
   assign DoorOpen = Movement;

   // Sequential Logic (FFs)
   always_ff @(posedge clk) begin
      if (reset)
         ps <= closed;
      else
         ps <= ns;
   end

endmodule

```

Figure 5*: Automatic Door Example FSM - SystemVerilog Module*



```
// Tests the auto_door module's two states given the Movement input
module auto_door_testbench();

   logic Movement, reset, clk, DoorOpen;

   // Instantiate our designed module
   auto_door dut (.Movement, .reset, .clk, .DoorOpen);

   // Set up the clock
   parameter clock_period = 100;
   initial begin
      clk <= 0;
      forever #(clock_period/2) clk <= ~clk;
   end

   // Simulate all possible state changes
   initial begin
      reset <= 1;                 @(posedge clk); // reset the system

      reset <= 0; Movement <= 0; @(posedge clk); // ps = closed --> ns = closed; DO = 0
                        repeat(2)@(posedge clk);

                  Movement <= 1; @(posedge clk); // ps = closed --> ns = open; DO = 1
                        repeat(2)@(posedge clk);

                  Movement <= 1; @(posedge clk); // ps = open --> ns = open; DO = 1
                        repeat(2)@(posedge clk);

                  Movement <= 0; @(posedge clk); // ps = open --> ns = open; DO = 0
                        repeat(2)@(posedge clk);
      $stop; //end simulation
   end

endmodule
```

Figure 6*: Automatic Door Example FSM - SystemVerilog Testbench*

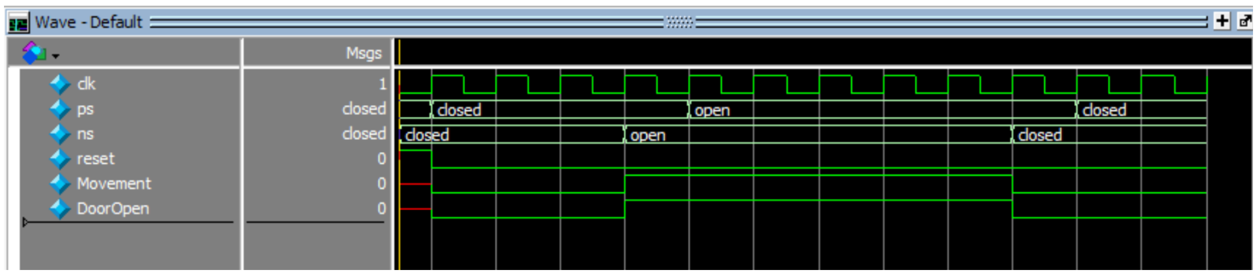


***Figure 7**: Automatic Door Example FSM - ModelSim Waveform*

Now that we can implement FSMs in SystemVerilog, let's try a guided design problem so we can put all that we've learned in Labs 0-5 together.

Problem Description

Using what you have learned in Labs 0-5, build a string recognizer which outputs a true only when the string “101” is observed. Note that a string is a sequence of characters, and our system receives a string of bits “001101010000111…”.

To solve this problem, there are five recommended steps:

1. Design an FSM to understand system behavior.
2. Describe inputs and outputs with a State Table.
3. Determine Boolean expressions for output signals using K-Maps.
4. Develop the system with SystemVerilog (Quartus) and simulate it (ModelSim).
5. Demonstrate the system on an FPGA (LabsLand).

The following sections walk you through this process.

Design a Finite State Machine to Understand System Behavior

Draw an FSM diagram corresponding to a string recognizer of ‘101’. Some hints are provided below.

- How many states do you need? What will you call them (words and binary)?
- How many inputs are there to the system? What will you call them? (Hint: think about the input signal AND present state inputs.)
- How many outputs are there to the system? What will you call them? (Hint: think about the output signal AND next state outputs.)

Describe Inputs and Outputs with a State Table

Fill in Table 2 to create a State Table for the inputs and outputs of your FSM.

***Table 2**: String Recognizer State Table*

Determine Boolean Expressions for Output Signals Using K-Maps

With the information from the State Table, use K-Maps to derive Boolean expressions for the outputs.

Verify your findings using the Boole-Web feature on LabsLand. Does the circuit schematic look like what you expected? Why or why not?

Develop and Simulate the System with SystemVerilog

Use the Boolean expressions you derived from the previous section to write and test SystemVerilog code for the string recognizer.

Quartus

Include screenshots of your code with your finished report. Use the following space to describe the syntax you used and why.

ModelSim

After simulating, draw the simulation curve you see on ModelSim. Make sure to specify your input and output signals.

Use the following space to describe why the simulation curve you see demonstrates the behavior we want the string recognizer to have.

Demonstrate the System on an FPGA

Send your code to an FPGA in LabsLand, mapping your input and output signals to the FPGA I/O (switches and LEDs). Describe what you observe in the space below.

How do you know the behavior you observed is that of a string recognizer for “101”?

Reflection and Observations

If you completed all the steps in this lab, you've just created your first digital system! Reflect on the process, things you found interesting, things you found challenging, etc. in the space below.

What did you observe about Quartus and ModelSim? Did you find the tools to be helpful?

What questions do you still have about sequential logic, FSMs, and SystemVerilog, if any?

ELECTRICAL & COMPUTER ENGINEERING

Lab 6: SMART HOME PROJECT

The University of Washington | The Remote Hub Lab | Last Revised: March 2022

Summary

This lab is the culmination of the Intro to Digital Logic Lab series. In this lab, you will have the opportunity to apply what you've learned in a self-guided project to gain experience with creating working digital systems on an FPGA.

Table of Contents

Putting it All Together

Labs 0-5 have covered many different topics, and connecting these topics together will help solidify your understanding of digital logic. This lab consists of two tasks which require you to apply all that you have learned in this introductory lab series.

Remember the five recommended steps (from Lab 5) when approaching any digital logic problem statement:

1. Design an FSM to understand system behavior.
2. Describe inputs and outputs with a State Table.
3. Determine Boolean expressions for output signals using K-Maps.
4. Develop the system with SystemVerilog (Quartus) and simulate it (ModelSim).
5. Demonstrate the system on an FPGA (LabsLand).

The Premise

In the era of high-speed data and the Internet of Things, there is a growing demand to leverage new technologies in smart homes.

In this lab, you are tasked to help create a smart home by developing the following:

1. A better automatic door system using motion detection, and
2. A security system using camera-based motion detection.

Task 1

Recall the automatic door system covered in Lab 5. That system was simplistic because it assumed that doors open and close instantaneously. However movement usually triggers a closed door to go to an "opening" state and then an "open" state. Likewise, a lack of movement usually would trigger an open door to go to a "closing" state and then to a "closed" state.

Implement and demonstrate this improved automatic door system, assuming it also checks for movement every 5 seconds. The door can only transition when a timer transmits a signal that 5 seconds has elapsed. For example, an open door should transition to the "closing" state and then to the "closed" state after 10 seconds of inactivity (that is, no movement is detected and the timer has transmitted the elapsed signal twice in a row).

Requirements

- Leverage Boole-Web as a verification tool. Include screenshots of the circuit schematic in your report.
- Include screenshots of your SystemVerilog code on Quartus and your waveform on Modelsim in your report.
- Demonstrate this modified automatic door system on LabsLand. (Remember to use switches and LEDs as your inputs and outputs respectively. It is left up to you to choose which switches and LEDs to use.)
- Reflect: How can you tell the FPGA is demonstrating the right behavior for this system?

Hints

- Consider the "5-seconds-elapsed" signal. Should it be an input or an output signal?
- Consider the FSM for the improved automatic door system. How many states should there be in total?
- Consider the output signal DoorOpen. What happens when movement is detected in the "opening" or "closing" state?

Task 2

The smart home we are creating has security cameras strategically placed throughout the residence in both indoor and outdoor locations. These cameras are capable of detecting motion and identifying whether that motion originated from a human. When the homeowners are on vacation, they want a security system with specific requirements.

Requirements

- If the outdoor cameras detect human motion, they will do the following:
    a. Record footage for 5 minutes.
    b. Stop recording after 5 minutes have elapsed.
- If the indoor cameras detect human motion, they will do the following:
    a. Notify the homeowner by visually displaying the word "siren" on an app.
    b. Record footage until the alarm is turned off in the app.
- Assume there are two main factors which will cause the cameras to detect non-human motion: cars and animals. We want to avoid any alarms for non-human motion.

Transferrable Requirements

The same requirements from Task 1 also apply to Task 2.

- Leverage Boole-Web as a verification tool. Include screenshots of the circuit schematic in your report.
- Include screenshots of your SystemVerilog code on Quartus and your waveform on Modelsim in your report.
- Demonstrate this security system on LabsLand.
- Reflect: How can you tell the FPGA is demonstrating the right behavior for this system?

Hints

- Remember that the security camera system is independent of Task 1's automatic doors; each system can use different timers.
- Use the 7-segment displays on the DE1_SoC board to display the word "siren."
    - The 7-segment display connections are shown in Figure 1. The segments are "active low" which means that a value of 0 is needed to turn ON any segment.
    - For example, to make HEX5 display the letter 'A', all the segments need to be turned ON except segment 3 (HEX5[3]).
    - Accordingly, to display the letter 'A' on HEX5, the following statement can be used in SystemVerilog:

```
assign HEX5 = 7'b0001000;      //displays 'A' where all segments except 3 are turned on.
```

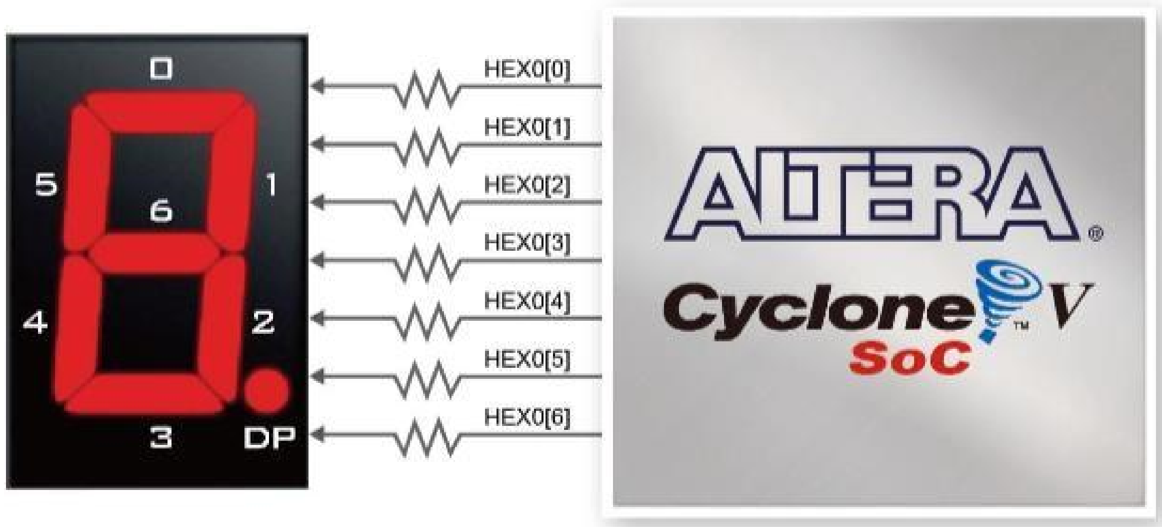


Figure 1: HEX0 connections on the DE1_SoC board
(Source: DE1_SoC Datasheet)

Reflection and Learnings

Congratulations! You have just completed your first digital systems which you created on your own. Take a moment to reflect on all that you have learned in this lab series. What is your takeaway from each of the following topics? Write down what stood out to you, or something you still wonder about.

- Operators
- Logic Gates
- Truth Tables
- Binary Numbers
- Boolean Algebra
- Karnaugh Maps (K-Maps)
- Finite State Machines (FSMs)
- System Verilog
- Field Programmable Gate Arrays (FPGAs)
- Quartus
- ModelSim
- LabsLand

Lastly, now that you have created your first digital system independently, what do you think about digital logic as a whole? Write down your thoughts below.

Thank you for completing this Intro to Digital Logic Lab Series. You're well on your way in the electrical and computer engineering field!