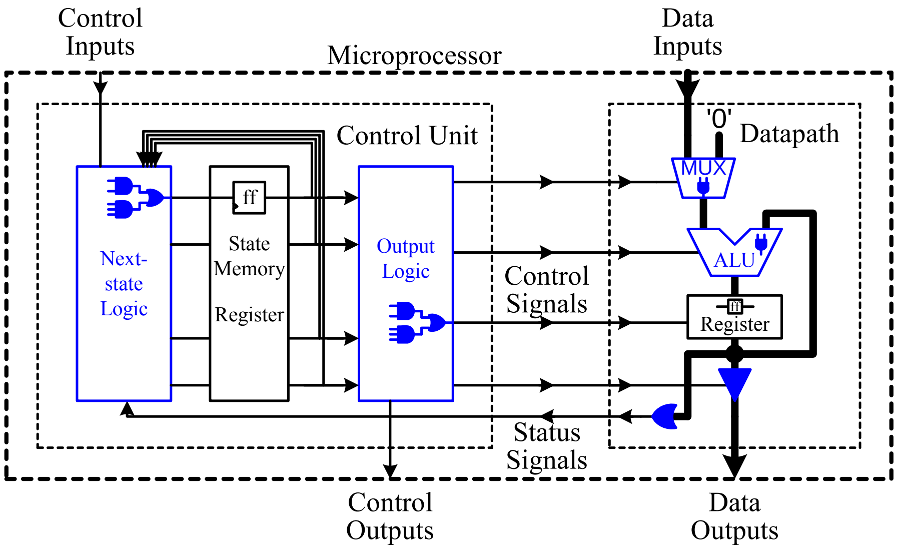
Combinational Logic Trainer

Lab Manual



Contents

1 Digital Logic Circuits

1.1 Introduction to Microprocessors

Whether you like it or not, microprocessors (also known as microcontrollers) control many aspects of our lives today – either directly or indirectly. In the morning, a microcontroller inside your alarm clock wakes you up, and another microcontroller adjusts the temperature in your coffee pot and alerts you when your coffee is ready. When you turn on the TV for the morning news, it is a microcontroller that controls the operation of the TV such as adjusting the volume and changing the channel. A microcontroller opens your garage door, and another inside your car releases your anti-lock break when you drive your car out. At the traffic light, a microcontroller senses the flow of traffic and turns on (hopefully) the green light for you when you reach the intersection. You stop by a gas station and a microcontroller reads and accepts your credit card, and let you pump your gas. When you walk up to your office building, a sensor senses your presence and informs a microcontroller to open the glass door for you. You press button eight inside the elevator, and a microprocessor controls the elevator to take you up to the 8th floor. During lunch break, you stop by a gift shop to buy a musical birthday card for your love one and find out that the birthday song is being generated by a microprocessor that looks like a dried-up pressed-down piece of gum inside the card. Well, I can continue on with this list of things that are controlled by microprocessors, but I think you got the idea and I better stop before you get bored. Oh, one last example, do you know that at the heart of your computer (either a PC or a Mac) is a microprocessor too? That's right the Intel Duo Core® CPU inside a PC is a general-purpose microprocessor.

So you see, microprocessors are at the heart of all "smart" devices, whether they be electronic devices or otherwise, and their smartness comes as a direct result of the decisions and controls that microprocessors make. In this three part award-winning series on microprocessor design training kits, you will learn how to design and actually implement real working custom microprocessors. Designing and building microprocessors may sound very complicated, but don't let that scare you, because it is not really all that difficult to understand the basic principles of how microprocessors are designed. After you have learned the materials presented in these labs, you will have the basic knowledge of how microprocessors are designed, and be able to design and implement your very own custom microcontrollers!

There are generally two types of microprocessors: **general-purpose microprocessors** and **dedicated microprocessors**. General-purpose microprocessors, such as the Intel Pentium® CPU, can perform different tasks under the control of software instructions. General-purpose microprocessors are used in all personal computers.

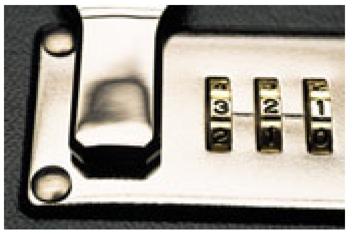
Dedicated microprocessors, also known as **application-specific integrated circuits** (**ASIC**s), on the other hand, are designed to perform just one specific task. For example, inside your cell phone, there is a dedicated microprocessor that controls its entire operation. The embedded microprocessor inside the cell phone does nothing else but controls the operation of the phone. Dedicated microprocessors are therefore usually much smaller, and not as complex as general-purpose microprocessors. Although the small dedicated microprocessors are not as powerful as the general-purpose microprocessors, they are being sold and used in a lot more places than the powerful general-purpose microprocessors that are used in personal computers.

The electronic circuitry inside a microprocessor is called a **digital logic circuit** or just **digital circuit**. As opposed to an **analog circuit** where the main components consist of discrete resistors, capacitors, inductors, and transistors, the main components in a digital circuit consist of the **AND**, **OR**, and **NOT** logic gates. From these three basic logic gates, the most powerful computer can be made. Furthermore, these logic gates are built using transistors—the fundamental building blocks for all digital logic circuits. Transistors are just electronic binary switches that can be turned on or off. The two binary values, 0 and 1, are used to represent the on and off states of a transistor. So instead of having to deal with different voltages and currents as in analog circuits, digital circuits only deal with the two abstract values of 0 and 1. Hence, it is usually easier to design digital circuits than analog circuits.

Every digital logic circuit is categorized as either a **combinational circuit** or a **sequential circuit**. A microprocessor circuit is composed of many different combinational circuits and many different sequential circuits as shown in the drawing on the front cover of this manual. In part I of this three part series on microprocessor design training kits you will learn how to design combinational circuits. In part II you will learn how to design sequential circuits. And finally in part III you will learn how to put these different combinational and sequential circuits together to make a real working microprocessor.

1.2 Combinational and Sequential Circuit Analogy

A simple analogy of the difference between a combinational circuit and a sequential circuit is the combination lock that we are familiar with. There are actually two different types of combination locks as shown in Figure 1. For the lock in Figure 1 (a), you just turn the three number dials in any order you like to the correct number and the lock will open. For the lock in Figure 1 (b), you also have three numbers that you need to turn to, but you need to turn to these three numbers in the correct sequence. If you turn to these three numbers in the wrong sequence the lock will not open even if you have the numbers correct. The lock in (a) is like a combinational circuit where the order in which the inputs are entered into the circuit does not matter, whereas, a sequential circuit is like the lock in (b) where the sequence of the inputs does matter.



(a)

(b)

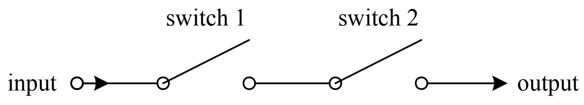
Figure 1: Two types of combination locks: (a) the order in which you enter the numbers does not matter, (b) the order in which you enter the numbers does matter.

So, a combinational circuit is one where the output of the circuit (like opening the lock) is dependent only on the current inputs to the circuit, but not dependent on the order in which these inputs are entered. One example of a combinational circuit is the adder circuit for adding two numbers. The adder takes two numbers for its inputs. With these two input numbers, it evaluates the sum of these two numbers and outputs the result. It doesn't matter which input number you enter first as long as you enter both numbers and the adder will output the sum.

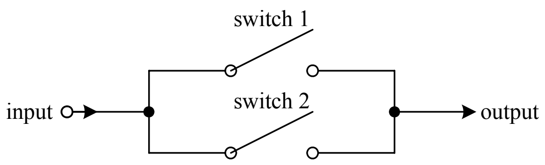
Examples of combinational circuits used inside a microprocessor circuit include adders, multiplexers, decoders, arithmetic and logic unit (ALU), and comparators. Some of these combinational components are highlighted in blue in the drawing on the front cover of this manual. In this courseware you will learn about these and many other combinational circuits.

1.3 Basic Logic Gates

All digital circuits are implemented with logic gates. The three basic logic gates are the AND gate, the OR gate and the NOT gate. These logic gates functions similar to binary switches and the only signals that they deal with are 0's and 1's. An analogy for the operation of the AND gate is like connecting two binary switches together in series as shown in Figure 2 (a). In order for a signal to go from the input to the output, both switches have to be turned on. On the other hand the OR gate is like connecting two binary switches together in parallel as shown in Figure 2 (b). In order for a signal to go from the input to the output, either one or both of the switches needs to be turned on.



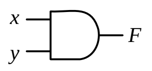
(a)



(b)

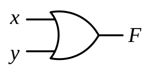
Figure 2: The connection of two binary switches: (a) in series; (b) in parallel.

Using the convention of a 1 meaning “on” and a 0 meaning “off”[1], the AND gate will output a 1 when both inputs have a 1 (i.e., both switches are on); otherwise the output will be a 0. Whereas, the OR gate will output a 1 when either one or both of its inputs has a 1 (i.e., either one or both of the switches is on); otherwise the output will be a 0. The NOT gate simply inverts the value at its input. So if the input is a 0, the output will be a 1, and vice versa. The logic symbols for the AND, OR and NOT gates together with their functional truth tables are shown in Figure 3. In these truth tables, *x* and *y* are the inputs to the gate and *F* is the output from the gate.



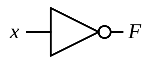
x	*y*	*F*
0	0	0
0	1	0
1	0	0
1	1	1

(a)



x	*y*	*F*
0	0	0
0	1	1
1	0	1
1	1	1

(b)

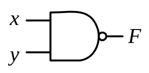


X	*F*
0	1
1	0

(c)

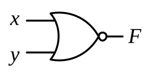
Figure 3: The logic symbols and functional truth tables for: (a) AND gate; (b) OR gate; (c) NOT gate. In these truth tables, *x* and *y* are the inputs and *F* is the output.

Any digital circuits, no matter how complex they may be, can be built using these three basic gates. However, there are several other gates derived from the AND, OR and NOT gates that are also used frequently in digital circuits. They are the NAND gate, NOR gate, XOR gate and XNOR gate. Their logic symbols and functional truth tables are shown in Figure 4.



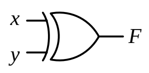
x	*y*	*F*
0	0	1
0	1	1
1	0	1
1	1	0

(a)



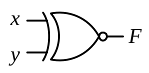
x	*y*	*F*
0	0	1
0	1	0
1	0	0
1	1	0

(b)



x	*y*	*F*
0	0	0
0	1	1
1	0	1
1	1	0

(c)

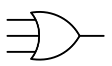


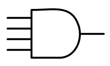
x	*y*	*F*
0	0	1
0	1	0
1	0	0
1	1	1

(d)

Figure 4: The logic symbols and functional truth tables for: (a) NAND gate; (b) NOR gate; (c) XOR gate; (d) XNOR gate. In these truth tables, *x* and *y* are the inputs and *F* is the output.

Furthermore, in addition to having just two inputs for all of the above mentioned gates (except for the NOT gate), they can have more than two inputs. Theoretically, there is no upper limit to the maximum number of inputs to these gates. In practice, however, they have only 2-, 3-, 4-, 6- and 8- inputs. Regardless of the number of inputs they have, they always have only one output. Figure 5 shows the symbols for some of these gates.


(a)


(b)

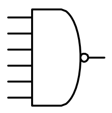

(c)

Figure 5: The logic symbols for: (a) 3-input OR gate; (b) 4-input AND gate; (c) 6-input NAND gate.

[1] This familiar convention is referred to as *active-high*. In practice, the *active-low* convention is sometimes used where a 1 means “off” and a 0 means “on”.

1.4 Digital Circuits

A digital circuit is just the connections of a bunch of these logic gates together. There are certain rules, however, that one must follow in making these connections. The output of a gate is always connected to the input of one or more gates unless it is a primary output. However, the outputs from two or more gates cannot be connected to the same input of a gate. Primary inputs from the external world are always connected to the inputs of various gates. The outputs that are not connected to the inputs of other gates are the primary outputs from the circuit to the external world. Figure 6 shows some very simple digital circuits.

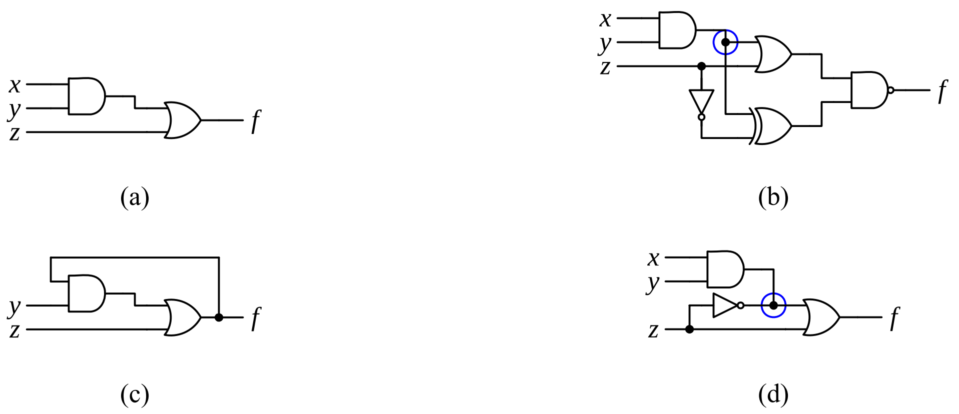


Figure 6: Sample digital circuits: (a) combinational circuit; (b) combinational circuit with one output connected to two inputs; (c) sequential circuit; (d) invalid circuit with two outputs connected to the same input.

In Figure 6 (a), the circuit has three primary inputs, *x*, *y* and *z*, from the external world, i.e., the user supplies either a 0 or a 1 to each of these three inputs. *x* and *y* are connected to the inputs of the AND gate, and *z* is connected to one input of the OR gate. The output of the AND gate is connected to the second input of the OR gate. Finally the primary output of the circuit is *f* which is from the output of the OR gate.

The circuit in Figure 6 (b) shows an example of where the output of one gate is connected to the inputs of two different gates. Like the circuit in Figure 6 (a) this circuit also has three inputs, *x*, *y* and *z*, and one output *f*. However, in this circuit, the output of the AND gate is connected to both the OR gate and the XOR gate. This connection point is circled in blue.

The circuit in Figure 6 (c) is very similar to the circuit in Figure 6 (a). The one main difference, besides having only two inputs, is that the output from the OR gate is connected back to the input of the AND gate. Keep this very important point in mind!

Finally, the circuit in Figure 6 (d) will produce errors because both the AND gate output and the NOT gate output are connected to the same input of the OR gate. This invalid connection point is circled in blue. Think about what happens if the AND gate outputs a 1 and the NOT gate outputs a 0? Since a 1 is like Vcc, and 0 is like ground, by connecting them together, you are creating a short circuit between power and ground!

1.5 Identifying Combinational Circuits

Before learning to design combinational circuits, we should be able to determine whether a given digital circuit is a combinational circuit or not. And if it is a combinational circuit, then we want to be able to describe its operation.

Looking back at the three simple digital circuits in Figure 6, remember the main difference between circuits (a) and (b), and circuit (c)? To reiterate, for both of the circuits in Figure 6 (a) and (b), the signals flow in one general direction from the primary inputs on the left side to the primary output on the right side. However, in Figure 6 (c), not only is the output from the AND gate connected to the input of the OR gate, but the output from the OR gate is connected back to the input of the AND gate, thus, creating a feedback loop. Another way to identify this feedback loop is that the output of a gate is connected back to one of its own input either directly or indirectly via other gates. The first two circuits are combinational circuits because of the absence of a feedback loop. The circuit in Figure 6 (c), because of the feedback loop, makes it a sequential circuit. Be careful, however, that in larger sequential circuits, the feedback loop can go through many gates, and not just two as shown in the example in Figure 6 (c). There might be one or more feedback loops within a circuit, but as long as there is at least one loop, the circuit is a sequential circuit. This is the only distinction between a combinational circuit and a sequential circuit. Hence, it is very easy to tell whether a given digital circuit is a combinational circuit or not.

1.6 Analysis of Combinational Circuits

Analyzing a circuit means to determine its functional operation. So in analyzing a combinational circuit, we are given a combinational circuit and we want to find out how it operates. A truth table (which is simply a two-dimensional array) is used to formally describe the functional operation of a combinational circuit. You have already seen some sample truth tables in Figure 3 and Figure 4. So basically what we want to do is to derive the truth table for a given combinational circuit.

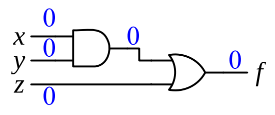
As an example, let us analyze the combinational circuit in Figure 6 (a). The first step in the analysis process is to set up the truth table for it. First, we list all of the primary inputs found in the circuit, one input per column, followed by all of the primary outputs found in the circuit, again one output per column. These columns are labeled with the names of the inputs and outputs. Since the circuit has three input variables, *x*, *y* and *z*, and one output variable, *f*, therefore we get a table having four columns as shown in Figure 7 (a). Second, we enumerate all possible combinations of 0's and 1's for all of the input variables. For three variables, we will have $2^3 = 8$ different combinations. In general, for a circuit with *n* input variables, there will be 2^n combinations going from 0 to $2^n - 1$. We will insert a row for each combination into the table. Figure 7 (b) shows the eight rows with the eight sets of input values in order.

x	*y*	*z*	*f*

(a)

x	*y*	*z*	*f*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(b)



(c)

x	*y*	*z*	*f*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

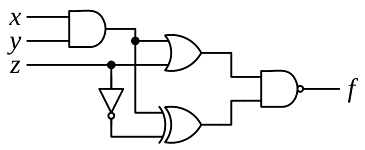
(d)

Figure 7: Deriving a truth table for a combinational circuit: (a) Step one–create a column for each input and output; (b) Step two–enumerate all possible combinations of 0's and 1's for the inputs; (c) Step three–for each set of input values, trace through the circuit to determine the output value. Circuit is annotated with the first set of input values; (d) the completed truth table.

The third and final step is to fill in the values for the output column(s). For each row in the table (that is, for each set of input values), we need to determine what the output value is. This is done by first substituting each set of input values into the circuit's primary inputs. Knowing the primary input values, we can determine for each gate in the circuit what its output ought to be starting from the primary inputs and tracing through the circuit to the final output. For example, using *xyz* = 000 (i.e., *x* = 0, *y* = 0 and *z* = 0), the two inputs to the AND gate are both 0. Hence the output of the AND gate will be 0, since from the AND gate truth table, 0 AND 0 is 0. This 0 from the output of the AND gate goes to one input of the OR gate, and the other input to the OR gate is also a 0 (from *z* = 0). Looking at the OR gate truth table, we get 0 OR 0 equals a 0. Hence, the output of the OR gate at *f*, which is also the primary output for the circuit is 0. Therefore, for the set of input values *xyz* = 000, the circuit output *f* is 0. This tracing process is shown in the annotated circuit in Figure 7 (c).

Continuing on for the next set of input values where *xyz* = 001, the output of the AND gate is again 0. However, with *z* being a 1 going into the OR gate, the output of the OR gate will be a 1. Therefore, *f* is a 1 for this second set of inputs. Repeat this tracing process for the rest of the combinations and you will have a completed truth table for the circuit. To help speed up the process, notice that when input *z* is a 1, the output of the OR gate will always be a 1 regardless of the other inputs. Therefore, when *z* is a 1, it doesn't matter what *x* and *y* are, *f* will always be a 1. By reasoning this way, you can more quickly determine many of the output values.

As an exercise, you may want to derive the truth table for the circuit in Figure 6 (b), and compare your results with the answer shown in Figure 8. For convenience, the circuit is repeated here.



x	*y*	*z*	*f*
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

Figure 8: The circuit from Figure 6 (b) and the truth table for it.

2 Labs

The following labs will teach you how to design and implement combinational circuits. Many of these circuits are standard components used in microprocessor circuits. They will be used in our Microprocessor Trainer where you will actually design and implement your very own custom real working microprocessor!

2.1 Lab 1: Basic Gates, Lights, Action!

Purpose

In this lab you will first learn how to implement combinational logic circuits using the Combinational Logic Trainer by connecting the basic logic gates and I/Os correctly from a given circuit. You will use the trainer to confirm the operations of the AND, OR, NOT and XOR gates. Finally, you will learn to derive the truth table for any combinational circuits.

Introduction

The AND, OR and NOT gates are the basic building blocks for building any digital logic circuits. The Combinational Logic Trainer that you have contains all of the necessary tools for you to implement almost any combinational logic circuit. The logic gates and I/O's are pre-mounted for easy wiring of the circuit. The layout of the trainer is shown in Figure 9. All logic gate inputs are connected to one wire connection point, and all logic gate outputs have multiple wire connection points. To connect from the output of a logic gate to the input of another logic gate, simply use a hook-up wire to connect between the two wire connection points. For example, switch SW7 has

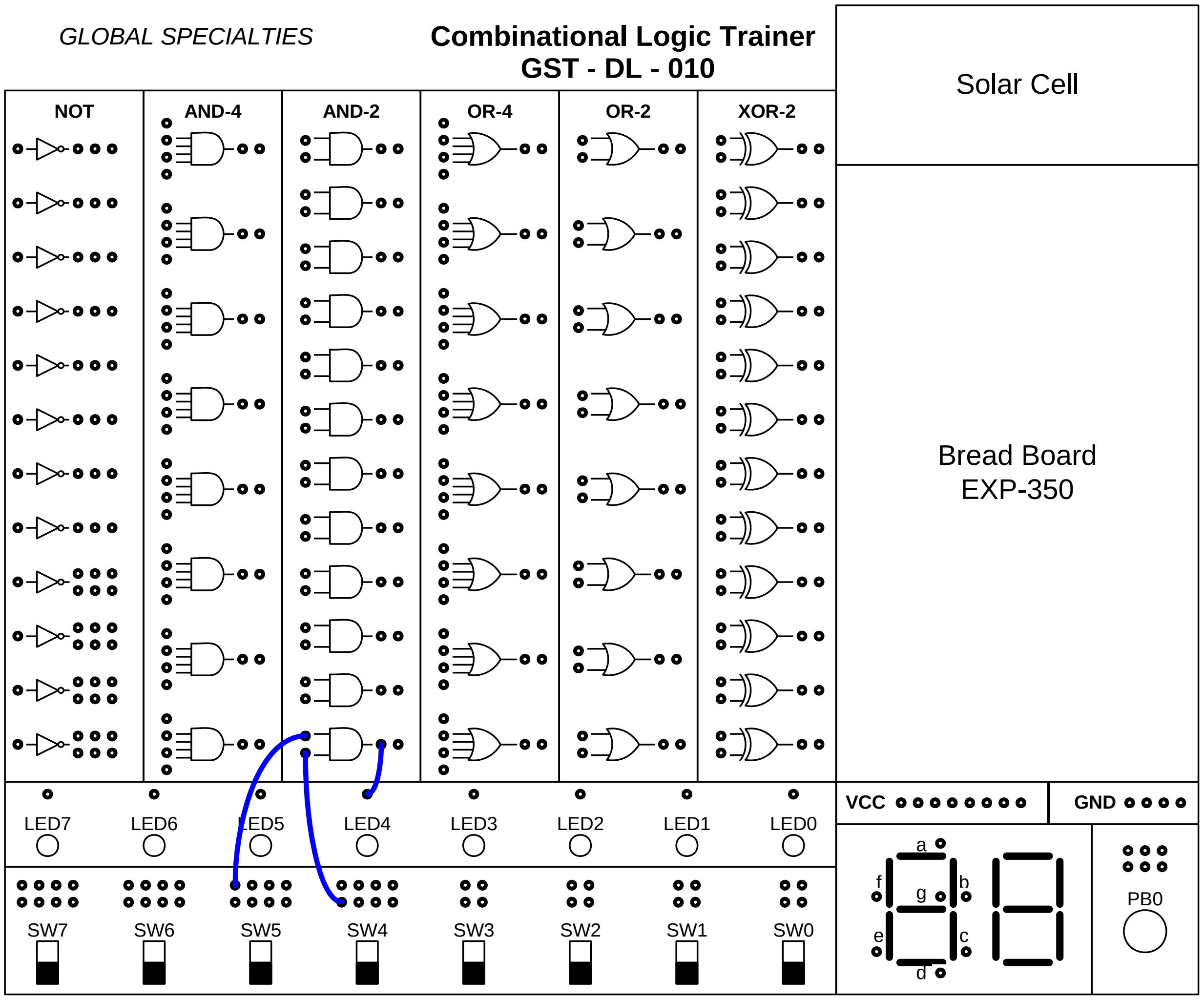


Figure 9: Combinational Logic Trainer layout. All of the logic gates and I/O's are pre-mounted with wire connection points connected to them.

eight common wire connection points, so to use SW7 one can connect a wire to any one of these eight connection points.

All of the eight LEDs are active high, which means that a logic one will turn the light on, and a logic zero will turn the light off. The push button PB0 is also active high, so pressing the button will produce a logic one. All of the eight switches are configured so that when the switch is in the up position the output is a logic one, and when the switch is in the down position the output is a logic zero.

Experiments

1. The three thick blue lines in Figure 9 show three wires connected from SW4 and SW5 to the inputs of a two-input AND gate, and the output of the AND gate is connected to LED4. Using three pieces of wire, make these same connections now on your trainer. Slide the two switches up and down and record the output on LED4 for all combinations of the input switches in the blank truth table provided at the end of this lab. Verify that they match the AND logic truth table shown in Figure 3 (a).

x	*y*	*F*
0	0	0
0	1	0
1	0	0
1	1	1

(a)

x	*y*	*F*
0	0	0
0	1	1
1	0	1
1	1	1

(b)

x	*F*
0	1
1	0

(c)

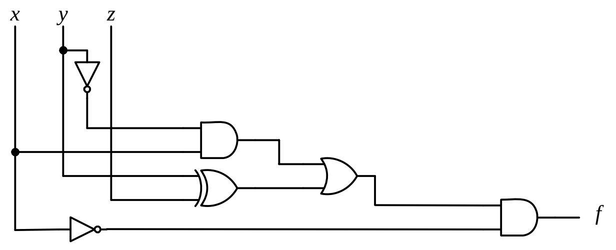
x	*y*	*F*
0	0	0
0	1	1
1	0	1
1	1	0

(d)

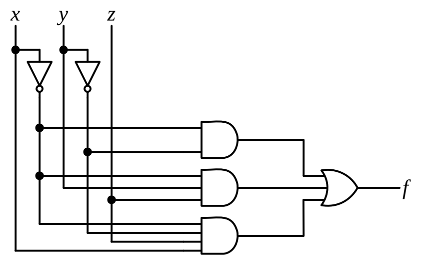
Figure 10: Truth tables for the four basic logical operators: (a) AND; (b) OR; (c) NOT; (d) XOR.

2. After you have confirmed that the AND gate do work according to the truth table, repeat the experiment with the 2-input OR gate, the NOT gate, and the 2-input XOR gate. Slide the two switches up and down and record the LED output for all combinations of the input switches in the blank truth tables provided. Verify that they match the corresponding truth tables in Figure 3 and Figure 4.
3. Repeat the experiment with the 4-input AND gate and the 4-input OR gate. Record your results in the blank truth tables provided.
4. Design and implement a circuit that operates exactly like a 6-input AND gate. You will need to use one 4-input AND gate and two 2-input AND gates. Determine its operation by recording the LED output for all combinations of the input switches in a truth table.
5. Design and implement a circuit that operates exactly like a 6-input OR gate. You will need to use one 4-input OR gate and two 2-input OR gates. Determine its operation by recording the LED output for all combinations of the input switches in a truth table.
6. Another frequently use gate is the NAND gate. This gate operates exactly like an AND gate with its output connected to a NOT gate. Implement this circuit by using a 2-input AND gate followed by a NOT gate. Connect the two inputs of the AND gate to two switches, the output of the AND gate to the input of the NOT gate, and finally the output of the NOT gate to an LED. Determine its operation by recording the LED output for all combinations of the input switches in the blank truth table provided. Verify that it matches the truth table in Figure 4 (a).
7. Another frequently use gate is the NOR gate. This gate operates exactly like an OR gate with its output connected to a NOT gate. Implement this circuit by using a 2-input OR gate followed by a NOT gate. Connect the two inputs of the OR gate to two switches, the output of the OR gate to the input of the NOT gate, and finally the output of the NOT gate to an LED. Determine its operation by recording the LED output for all combinations of the input switches in the blank truth table provided. Verify that it matches the truth table in Figure 4 (b).
8. Implement the circuit in Figure 6 (a) and verify that it operates according to the truth table in Figure 7 (d).
9. Implement the circuit in Figure 6 (b) and verify that it operates according to the truth table in Figure 8.

10. Implement the following circuit and determine its operation by filling out the truth table for it. Connect the three inputs, *x*, *y* and *z*, to three switches, and connect the output *f* to an LED.

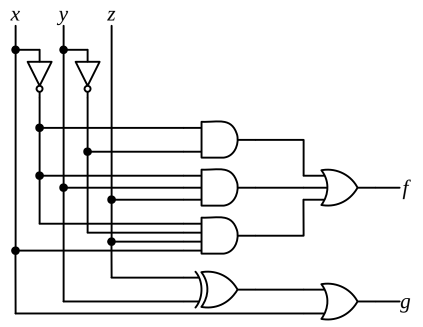


11. Implement the following circuit and determine its operation by filling out the truth table for it. Notice that this circuit uses a 3-input AND gate and a 3-input OR gate, both of which the trainer does not have. Instead, you can use a 4-input AND gate and a 4-input OR gate respectively. In order for the 4-input AND gate to work like a 3-input AND gate, connect the extra input to VCC. In order for the 4-input OR gate to work like a 3-input OR gate, connect the extra input to GND. Connect the three inputs, *x*, *y*, and *z*, to three switches, and connect the output *f* to an LED.



12. For experiment 11, what happens if you connect the extra input of the 4-input AND gate to GND instead of VCC? What happens if you connect the extra input of the 4-input OR gate to VCC instead of GND?

13. Implement the following circuit and determine its operation by filling out the truth table for it.



14. Implement the circuit in Figure 6 (d) and determine its operation by deriving the truth table for it. Is there anything wrong with the results?

Truth Tables for Reporting Results

x	*y*	*F*
0	0	
0	1	
1	0	
1	1	

2-input AND gate

x	*y*	*F*
0	0	
0	1	
1	0	
1	1	

2-input OR gate

x	*F*
0	
1	

NOT gate

x	*y*	*F*
0	0	
0	1	
1	0	
1	1	

2-input XOR gate

w	*x*	*y*	*z*	*F*
0	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	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	

4-input AND gate

w	*x*	*y*	*z*	*F*
0	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	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	

4-input OR gate

x	*y*	*F*
0	0	
0	1	
1	0	
1	1	

2-input NAND gate

x	*y*	*F*
0	0	
0	1	
1	0	
1	1	

2-input NOR gate

x	*y*	*z*	*f*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Experiment 8

x	*y*	*z*	*f*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Experiment 9

x	*y*	*z*	*f*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Experiment 10

x	*y*	*z*	*f*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Experiment 11

x	*y*	*z*	*f*	*g*
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Experiment 13

2.2 Lab 2: Designing Combinational Circuits

Purpose

In this lab you will learn how to design combinational circuits from any given truth table and implement them on the trainer.

Introduction

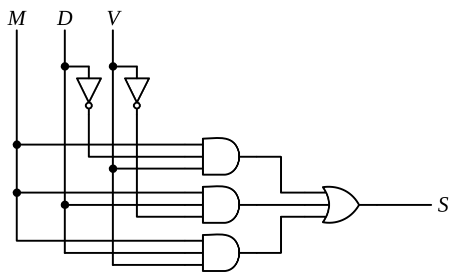
As you saw in Lab 1, the operation of any combinational circuit can be described formally by a truth table. Moreover, any given truth table can be implemented with a digital circuit. In fact, any truth table can be implemented with one or more different but functionally equivalent digital circuit. A digital circuit that implements a truth table is always a combinational circuit. The design process begins with an informal description of the circuit that you want. This informal description is translated into a precise and formal description of the circuit in the form of a truth table. Given a truth table, you can easily construct a combinational circuit for it. Hence, the circuit will operate according to the specifications in the truth table.

As an example, we begin with an informal description of a car security system that we would like to implement. This simple car security system consists of a master switch (*M*) for turning on and off the system, a door switch (*D*) for detecting whether the car door is open or close, a vibration sensor (*V*) for detecting movements of the car, and a siren (*S*) for sounding the alarm. Given these input and output parameters, I assume that you are at least slightly acquainted with the functional operations of a security system so that you can precisely describe its operation using a truth table.

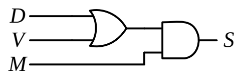
First, you layout your truth table with the column labels being the inputs and output signals. Next, using binary numbers, you enumerate all possible input values; creating one row per value. You should have the table shown in Figure 11 (a) but without the values in the *S* column. Interpreting a 1 being "on" and a 0 being "off", determine for each row (i.e., each combination of input values) what the corresponding output ought to be (i.e., whether the siren *S* should be turned on or off). For example, in the first row where *M*=0, *D*=0, and *V*=0, this means that the system is off, so regardless of the state of the door switch or the vibration sensor, the siren should be off. Hence the output *S* is 0 for this row. In the row where *M*=1, *D*=0, and *V*=0, this means that the system is on, but since the door is closed (*D*=0) and there is no vibration (*V*=0), the siren should also be off. Hence the output *S* for this row is also 0. Continuing with this reasoning, you should be able to complete the table and obtain the complete truth table as shown in Figure 11 (a).

M	*D*	*V*	*S*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)



(c)

Figure 11: Car security system: (a) truth table; (b) circuit diagram derived from the truth table; (c) simplified circuit diagram.

Obtaining a circuit from a truth table is fairly straight forward. For each row where the output (*S*) is a 1, you use the AND operation to AND the input values together. Where the input value is a 1, you connect that input directly to an input of the AND gate. Where the input value is a 0, you connect the negated input (through the NOT gate) to an input of the AND gate. Finally, the outputs of all the AND gates are connected to the inputs of an OR gate to produce the primary output signal. Hence from the truth table in Figure 11 (a), you should obtain the circuit in Figure 11 (b). This circuit will operate exactly according to the truth table.

In practice, we usually want to make the circuit as small as possible but still operates exactly the same. Using logical reasoning and looking at the truth table in Figure 11 (a) more carefully, you might notice that the siren should be turned on (S=1) only when the master switch is on (M=1), and either the door is opened (D=1) or there is vibration (V=1). In other words, you want S to be 1 only when M=1 and either D=1 or V=1 (or both D and V are 1's). Writing this out as a Boolean equation, we get

$$S = M \text{ AND } (D \text{ OR } V)$$

In textbooks, you might see this equation[2] written as

$$S = M\,(D + V)$$

This equation gives rise to the simplified circuit shown in Figure 11 (c). The two circuits in Figure 11 are functionally equivalent, i.e., they satisfy the same truth table.

Experiments

1. Implement the two circuits in Figure 11 (b) and (c), and verify that both of them operate according to the truth table in Figure 11 (a). Connect the three inputs, M, D, and V, to three switches, and connect the output S to an LED.

2. Design and implement the circuit for the 2-input XOR gate without using the XOR gate.

3. Design and implement the circuit for the following truth table where x, y, and z are the inputs, and f is the output. See Experiment 5 in Lab 1 on how to construct a 6-input OR gate.

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

4. Design and implement the circuit for the following truth table where x, y, and z are the inputs, and f is the output.

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

[2] For an in-depth discussion on Boolean equations, refer to the book "Digital Logic and Microprocessor Design with VHDL" by E. Hwang.

5. Notice that the outputs for Experiment 4 are just the inverse of those from Experiment 3, i.e., all the 0's and 1's are flipped. In other words, if you implement a circuit by selecting the rows where the output *f* is a 0 instead of a 1, you will end up with the inverse of *f*. Also notice that the circuit for Experiment 4 is much smaller than the circuit for Experiment 3. So a better way to implement the truth table for Experiment 3 is to start with the circuit for Experiment 4, and simply add a NOT gate to the output to invert the result. Implement this new circuit and verify that it operates exactly the same as the circuit from Experiment 3, i.e., both circuits produce the same truth table.

6. Design and implement the circuit for the following truth table where x, y, and b_{in} are the inputs, and b_{out} and d are the outputs.

x	y	b_{in}	b_{out}	d
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

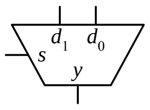
2.3 Lab 3: Multiplexers

Purpose

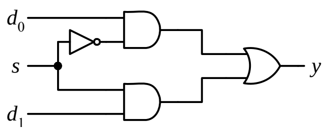
In this lab you will learn about multiplexers. A multiplexer is used to channel data from multiple sources to one destination. You will design multiplexer circuits and implement them on the trainer.

Introduction

Multiplexers, also known as mux, are a frequently used component in a digital circuit. They are used to pass data from multiple sources to one destination. An analogy of its operation is like a railroad switch where two rail tracks are merged into one track. Depending on the switch setting, trains from either one of the two tracks are directed onto the one track. The logic symbol for a 2-to-1 mux is shown in Figure 12 (a). It has two inputs labeled d_0 and d_1, and one output labeled y. Instead of passing trains, data is passed from either input d_0 or d_1 to the output y depending on the select line s. If the value of s is 0 then the data from input d_0 is passed to the output, and if the value of s is 1 then the data from input d_1 is passed to the output. The simplified circuit for the 2-to-1 mux is shown in Figure 12 (b). The truth table and the equation[3] are shown in Figure 12 (c) and (d).



(a)



(b)

s	d_1	d_0	y
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

(c)

$$\begin{aligned} y &= s'd_1'd_0 + s'd_1d_0 + sd_1d_0' + sd_1d_0 \\ &= s'd_0(d_1' + d_1) + sd_1(d_0' + d_0) \\ &= s'd_0 + sd_1 \end{aligned}$$

(d)

Figure 12: A 2-to-1 multiplexer: (a) logic symbol; (b) circuit; (c) truth table; (d) equation.

Experiments

1. Implement the simplified 2-to-1 mux circuit as shown in Figure 12 (b), and confirm that it operates according to the truth table shown in Figure 12 (c). Connect the three inputs, s, d_1, and d_0, to three switches, and connect the output y to an LED.

2. Design the 2-to-1 mux circuit based on the truth table shown in Figure 12 (c) and without any simplifications. Implement your circuit and confirm that it operates according to the truth table.

[3] For an in-depth discussion on how to simplify Boolean equations and circuits, refer to the book "Digital Logic and Microprocessor Design with VHDL" by E. Hwang.

3. A larger size mux is a 4-to-1 mux where there are four data inputs, d_3, d_2, d_1, and d_0. In order to select one of the four data inputs, two select lines, s_1, and s_0, are needed. When s_1s_0 = 00, data from d_0 is passed to the output; when s_1s_0 = 01, data from d_1 is passed to the output; when s_1s_0 = 10, data from d_2 is passed to the output; and When s_1s_0 = 11, data from d_3 is passed to the output. Design a 4-to-1 mux circuit. Implement your circuit and confirm that it works correctly.

4. Design and implement an 8-to-1 multiplexer. There should be 8 data input lines, 3 select lines and 1 output line. Since there are only 8 switches on the trainer, you can connect the three select lines directly to either VCC (for 1) or GND (for 0).

2.4 Lab 4: Decoders

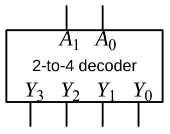
Purpose

In this lab you will learn about decoders. A decoder is used to select one among several devices or memory locations. You will design decoder and encoder circuits and implement them on the trainer.

Introduction

A decoder is another frequently used component in a digital circuit. The function of the decoder is to select one thing among several things. For example, if you have an array of 16 memory locations and you want to read from one particular location, a decoder will be used to select which one of the 16 memory locations that you want to access. The logic symbol for a 2-to-4 decoder is shown in Figure 13 (a).

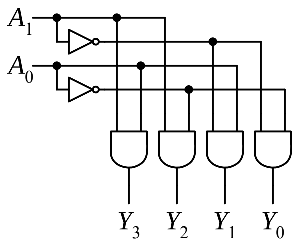
As implied by the name, 2-to-4 decoder, this component has two input lines, A_1A_0, which are the two address select lines for selecting one of the four output lines, $Y_3Y_2Y_1Y_0$. When the address lines A_1A_0=00, the output line Y_0 will be selected; when the address lines A_1A_0=01, the output line Y_1 will be selected; when the address lines A_1A_0=10, the output line Y_2 will be selected; and when the address lines A_1A_0=11, the output line Y_3 will be selected. The selected output line will have a 1 value, while all of the remaining output lines will have 0's. Given this description, you should now be able to derive the truth table and the circuit for a 2-to-4 decoder. The truth table and the circuit are shown in Figure 13 (b) and (c).



(a)

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

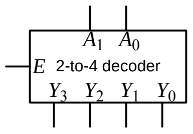
(b)



(c)

Figure 13: A 2-to-4 decoder: (a) logic symbol; (b) truth table; (c) circuit.

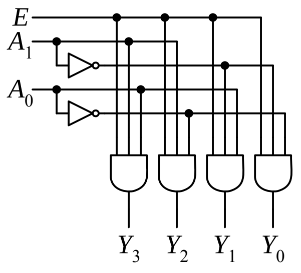
A variation of the 2-to-4 decoder has an extra enable (E) input line. When E is a 1, this decoder works exactly like the original. However, when the circuit is disabled with E set to 0, then it doesn't matter what the address input lines are, all of the output lines will be a 0. The logic symbol, truth table, and the circuit for the 2-to-4 decoder with enable are shown in Figure 14.



(a)

E	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	×	×	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

(b)



(c)

Figure 14: A 2-to-4 decoder with enable: (a) logic symbol; (b) truth table; (c) circuit.

In addition to the 2-to-4 decoder, there are other sizes of decoders, namely, the 1-to-2 decoder and the 3-to-8 decoder. Both of these can have the variation of either having or not having the enable input line. The 3-to-8 decoder will have three address lines and eight output lines.

Experiments

1. Implement the 2-to-4 decoder circuit as shown in Figure 13 (c), and verify that it works according to the truth table shown in Figure 13 (b). Connect the three inputs, E, A_1, and A_0, to three switches, and connect the outputs Y_3, Y_2, Y_1, and Y_0, to four LEDs.

2. Implement the 2-to-4 decoder with enable circuit as shown in Figure 14 (c), and verify that it works according to the truth table shown in Figure 14 (b).

3. Design and implement the 3-to-8 decoder.

4. Design and implement the 3-to-8 decoder with enable.

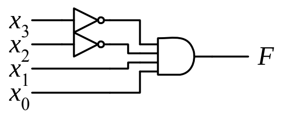
2.5 Lab 5: Comparators

Purpose

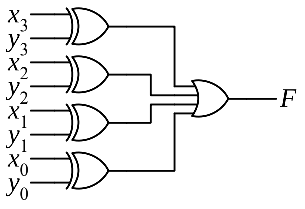
In this lab you will learn about comparator circuits. Comparators are for comparing between two numbers to see if one number is less than, equal to, or greater than a second number. You will implement different comparator circuits and verify their operations.

Introduction

A comparator will output a 1 if the logical condition that it is testing for is true, and outputs a 0 if the condition is false. The simplest comparator is to compare whether a value is equal or not equal to a constant. The use of an AND gate is all that is needed for the equality comparator with a constant. For example, the circuit in Figure 15 (a) tests whether a 4-bit variable x is equal to the constant 3 or not. Since 3 in binary is 0011, therefore, with x_3 and x_2 inverted, the AND gate will output a 1 when x is equal to 0011. For all numbers other than 3, it will output a 0.



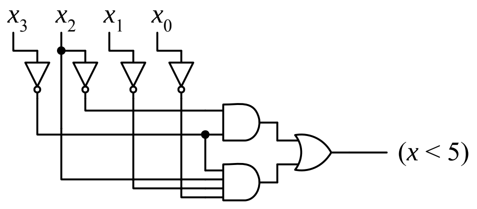
(a)



(b)

x_3	x_2	x_1	x_0	$(x < 5)$
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
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

$(x < 5) = x_3'x_2' + x_3'x_2x_1'x_0'$



(c)

Figure 15: Simple 4-bit comparators for: (a) $x = 3$; (b) $x \neq y$; (c) $x < 5$.

The XOR gate can be used for comparing inequality between two variables. Recall that the XOR gate outputs a 1 when its two input values are different. Hence, we can use one XOR gate for comparing each bit pair of the two operands. A 4-bit inequality comparator is shown in Figure 15 (b). Four XOR gates are used, with each one comparing the same position bit from the two operands. The outputs of the XOR gates are ORed together so that if any bit pair is different then the two operands are different, and the resulting output is a 1.

For the greater-than or less-than relationships, we can construct a truth table and build the circuit from it. For example, to compare whether a 4-bit value x is less than five, we get the truth table shown in Figure 15 (c). The first five rows have a 1 output since their decimal values are equal to 0 to 4 respectively. The remaining rows from 5 to 15 will all have a 0 output. The resulting simplified circuit[4] is shown in Figure 15 (c).

[4] For an in-depth discussion on how to simplify Boolean equations and circuits, refer to the book "Digital Logic and Microprocessor Design with VHDL" by E. Hwang.

Experiments

1. Design and implement a 4-bit comparator circuit that tests for the condition $x = 7$. Verify that it operates correctly.

2. Design and implement a 4-bit comparator circuit that tests for the condition $x \neq 7$. Verify that it operates correctly.

3. Design and implement a 4-bit comparator circuit that tests for the condition $x = y$. Verify that it operates correctly.

4. Design and implement a 4-bit comparator circuit that tests for the condition ($x = 7$) AND ($x = 12$). Verify that it operates correctly.

5. Design and implement a 4-bit comparator circuit that tests for the condition ($x = 7$) OR ($x = 12$). Verify that it operates correctly.

6. Design and implement a 4-bit comparator circuit that tests for the condition $x \leq 7$. Verify that it operates correctly.

7. Design and implement a 4-bit comparator circuit that tests for the condition $x \geq 7$. Verify that it operates correctly.

8. Design and implement a 4-bit comparator circuit that tests for the condition ($x = 3$) AND ($x \geq 7$). Verify that it operates correctly.

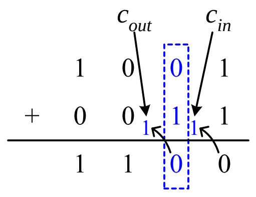
2.6 Lab 6: Full Adder

Purpose

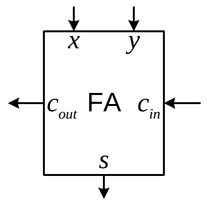
In this lab you will learn about the full adder (FA) circuit. The full adder circuit is for adding two 1-bit binary numbers with carry-in and carry-out. You will implement the circuit and verify its operations.

Introduction

The full adder circuit adds a 1-bit binary number with a second 1-bit number to produce a sum and a carry-out bit. The circuit also has a carry-in bit that allows it to be connected in series with other FA circuits. Consider the following addition of two 4-bit binary numbers



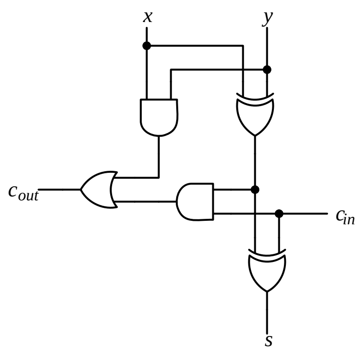
The full adder circuit is designed to only add one bit slice of the 4-bit number, for example, the bit slice that is highlighted in blue. In adding this bit slice, in addition to the two input operands, there is also a carry-in bit from the previous bit slice. The result of the addition produces a sum bit and a carry-out bit for a carry to the next bit slice. The carry-in and carry-out bits allow the FA to pass bits from one bit position to the next bit position when several FAs are connected together in series. So in total, there are three input bits, first input operand (x), second input operand (y) and the carry-in (c_{in}), and two output bits, carry-out (c_{out}) and the sum (s). The logic symbol, truth table and the simplified circuit are shown in Figure 16 (a), (b) and (c) respectively.



(a)

x	y	c_{in}	c_{out}	s
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

(b)



(c)

Figure 16: The full adder (FA): (a) logic symbol; (b) truth table; (c) simplified circuit.

Experiments

1. Implement the circuit shown in Figure 16 (c) and verify that it operates according to the truth table.

2. Notice that the full adder circuit shown in Figure 16 (c) is much simplified. If you construct the circuit by picking out the 1's rows in the output columns of the truth table, you will end up with a much larger circuit. Derive this circuit, then implement and verify that it operates according to the truth table.

3. What should the value for c_{in} be if all you want is to add the values of the two operands x and y?

4. Derive the truth table for a full subtractor circuit following the same way that the full adder was constructed. You will have three inputs, *x*, *y* and b_{in} (for borrow in), and two outputs, b_{out} (for borrow out) and *d* (for difference). The circuit performs the subtraction $x - y$ to give *d*. A 1 for b_{in} denotes that there is a borrow from the bit on the right side, and a 1 for b_{out} denotes that this current bit needs to borrow from the bit on the left. So in essence, you are doing $x - y - b_{in}$ to give *d*, and if you need to borrow then set b_{out} to a 1, otherwise b_{out} is a 0. Keep in mind that you are working with binary numbers, so when you borrow, you get a 2 and not a 10. The first two rows of the truth table are shown next.

x	y	b_{in}	b_{out}	d
0	0	0	0	0
0	0	1	1	1
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

For the second row, you have 0 – 0 – 1, and *x* being a 0 is not enough, so you borrow by setting b_{out} to a 1. When you borrow, you get a 2, so *x* now has a 2; and 2 – 0 – 1 = 1, so *d* is also a 1. Try to complete the truth table on your own before looking for the answers. The complete truth table is found in Experiment 6 in Lab 2.

5. Implement the circuit from the truth table in Experiment 4, and verify that it operates correctly.

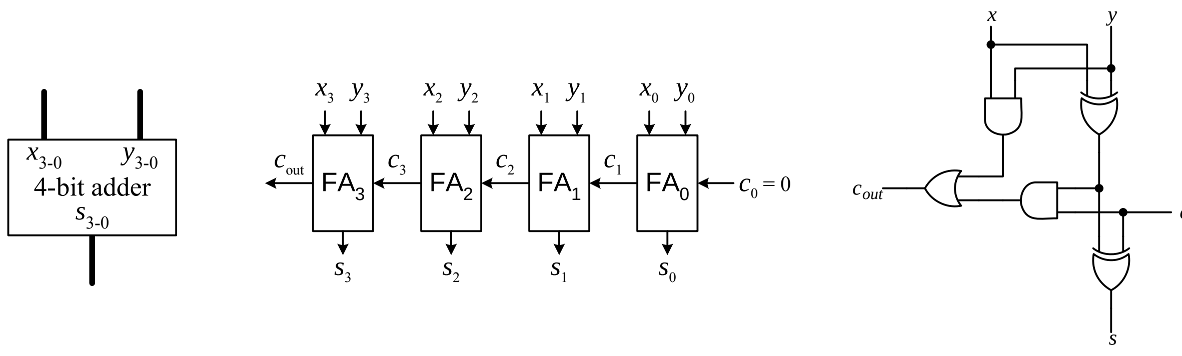
2.7 Lab 7: 4-bit Adder

Purpose

In this lab you will learn about the 4-bit adder circuit. This 4-bit adder circuit is for adding two 4-bit binary numbers producing a 4-bit sum and a carry-out. You will implement the circuit and verify its operations.

Introduction

There are different ways to design a 4-bit adder circuit, but one very easy to understand way is to simply connect four full adder circuits together in series as shown in Figure 17 (b). Each box with the label FA is the full adder circuit from Figure 16 (c), and repeated here in Figure 17 (c) for easy reference. The carry-out (c_{out}) signal from each FA is connected to the carry-in (c_{in}) signal of the next FA on the left side. The first carry-in signal (c_0) is connected to 0, and the final carry-out (c_{out}) signal is the carry signal for the complete 4-bit adder circuit. The two 4-bit operands are x and y, and the resulting 4-bit sum from the adder is s. The subscripts for x, y, and s denote the bit position of the 4-bit number, e.g., x_0 is bit zero or the first bit of operand x, and s_3 is bit three or the fourth bit of s. This adder circuit is called a ripple-carry adder because of how the carry signal ripples through the chain of FAs[5].



(a) (b) (c)

Figure 17: The 4-bit adder: (a) logic symbol; (b) circuit; (c) the FA circuit from Figure 16 (c).

Experiments

1. Implement the circuit shown in Figure 17 (b). You will need to have four FA circuits with their c_{in}'s and c_{out}'s connected together. Connect the 4-bit operand x_3, x_2, x_1 and x_0 to switches SW7 to SW4. Similarly connect the 4-bit operand y_3, y_2, y_1 and y_0 to switches SW3 to SW0. Connect the 4-bit sum s_3, s_2, s_1 and s_0 to LED3 to LED0. Connect the carry-out signal (c_{out}) to LED7. Verify that your circuit does add two 4-bit binary numbers correctly. The following table lists some sample numbers that you might want to try. The numbers in parenthesis are the corresponding decimal numbers. Note that to be absolutely certain that your circuit works correctly, you need to test all possible input combinations.

Input		Output		Observed Result
Operand *x*	**Operand *y***	**Carry-out c_{out}**	**Sum *s***	
0010 (2)	0011 (3)	0	0101 (5)	
0101 (5)	0100 (4)	0	1001 (9)	
0110 (6)	1001 (9)	0	1111 (15)	
0111 (7)	1100 (12)	1	0011 (19)	
1001 (9)	1101 (13)	1	0110 (22)	
1111 (15)	1111 (15)	1	1110 (30)	

[5] The operation of the ripple-carry adder is relatively slow. A faster adder circuit is called the carry-lookahead adder.

2. What happens if the initial carry-in signal c_0 is connected to a 1 instead of a 0?
3. How would you expand this 4-bit adder circuit to be an 8-bit adder circuit?

2.8 *Lab 8: 4-bit Adder/Subtractor*

Purpose

In this lab you will learn about a 4-bit adder/subtractor circuit. This 4-bit adder/subtractor circuit is for both adding and subtracting two 4-bit binary numbers producing either a 4-bit sum or difference. You will implement the circuit and verify its operations.

Introduction

It turns out that with very minimal modifications to the 4-bit adder circuit from Figure 17 (b), you can get it to subtract numbers as well. This simple and elegant solution is a direct result of how negative numbers are represented inside computers.

Negative Numbers

Binary numbers can be interpreted as either signed or unsigned. Unsigned numbers include only positive numbers and zero, whereas signed numbers include positive, negative, and zero. Given a binary number such as 01101001_2, the computer does not know whether it is a signed or unsigned number. It is up to you, the designer, to decide how you want to interpret it. If you say that this binary number represents an unsigned number, then the decimal value of this number would be

$$
\begin{aligned}
&01101001_2\\
&= (0 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)\\
&= (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^3) + (1 \times 2^0)\\
&= 64 + 32 + 8 + 1\\
&= 105 \text{ in decimal.}
\end{aligned}
$$

(Notice that the power x used in 2^x denotes the position of the bit in the binary number. The first bit of the binary number starting from the right hand side is position 0; the second bit to the left is position 1; etc.)

If you say that this same binary number 01101001_2 represents a signed number, then the decimal value of this number would be evaluated differently. For signed numbers, the most significant bit (MSB) which is the left-most bit, tells whether the number is positive or negative. If the most significant bit is a 0, then the number is positive, and the value of this positive signed number is obtained exactly as for unsigned numbers. So if we interpret 01101001_2 as a signed number, we would get the same decimal value 105.

However, if the most significant bit of a signed number is a 1, then the number is negative, and we use what is referred to as the *two's complement* method to determine its value. The 2's complement[6] is a method for representing negative or signed numbers, and it involves three steps to determine the value. In step 1, you flip all the 1 bits in the binary number to 0's and all the 0 bits to 1's. In step 2, you add a 1 to the result obtained from step 1. Finally, for step 3, interpret the binary number obtained in step 2 as an unsigned number, and determine its value. The negative of this resulting value is the value of the original negative signed number.

For example, if we say that the binary number 11101001_2 represents a signed number, then we would have to do the following to determine its value. First we note that the MSB is a 1 so we know that it is a negative number. To determine the value of a negative number, we perform the three steps for the 2's complement process.

Starting with this number:	11101001
Step 1, flip the bits:	00010110
Step 2, add a one to the number:	00010111
Step 3, determine the value:	$(1 \times 2^4) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

[6] Sometimes two's complement is also written as 2's complement.

$$= 16 + 4 + 2 + 1$$
$$= 23$$

Therefore, the value for the signed number 11101001_2 is –23.

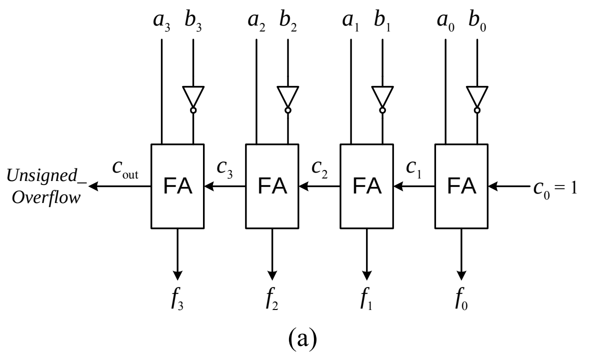
To find the 2's complement binary representation of a negative number, we start with the positive binary representation and then perform only the first two steps in the process. For example, to find the 2's complement binary representation for –35, we start with the binary representation for +35.

Starting with +35:	00100011
Step 1, flip the bits:	11011100
Step 2, add a one to the number:	11011101

Therefore, 11011101_2 is the 2's complement representation for –35. You can verify that this is indeed correct by performing the three step process again.

Subtractor Circuit

Now that we understand how negative numbers are represented, we are ready to design the subtractor circuit. We know from algebra that subtracting a positive number is the same as adding the negative of the number. So using this fact, we can use the 4-bit adder to do subtraction simply by changing the second number to its negative equivalent. In other words, we simply perform the first two steps ("flip the bits" and "add a 1") in the 2's complement process to convert the second number to its negative value, and then pass this negative number to the adder circuit. The interesting trick is that these two steps can be easily done at two different places in the circuit. First, we can flip the bits by using a NOT gate for every bit of the second number. Second, remember that setting c_0 to a 1 in the FA adds an extra one to the sum, and since the addition of a 1 can occur anytime during the addition process, therefore we can add the 1 simply by setting c_0 to a 1 instead of the original 0. Hence, we obtain the subtractor circuit shown in Figure 18 (a).



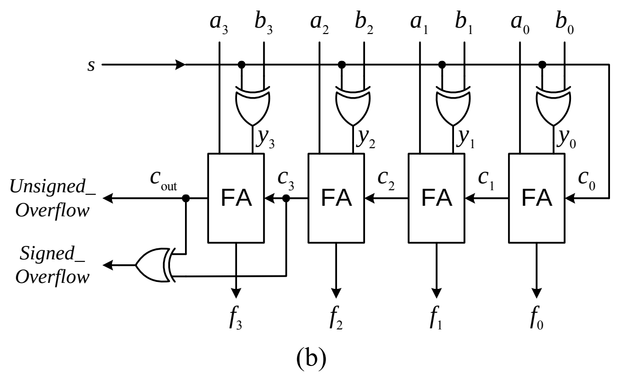


Figure 18: (a) The 4-bit subtractor circuit; (b) the 4-bit adder/subtractor combination circuit.

Adder/Subtractor Circuit

One enhancement that we can make to the subtractor circuit is to replace the NOT gates with 2-input XOR gates, and connect the second input to all of the XOR gate to a common select signal s, as shown in Figure 18 (b). The select signal is also connected to c_0. With this change, the circuit can now perform both addition and subtraction depending on the select signal s. When s is 0, c_0 will also be 0, and with one input of the XOR gate being a 0, the bit at the second input will pass unchanged to the output (refer to the XOR gate truth table to see that this is so). Therefore, the circuit will perform an addition when s is a 0. When s is 1, c_0 will be 1, and the XOR gate will flip the bit of its second input. Therefore, the circuit will perform a subtraction when s is a 1. In Figure 18 (b), there is an extra XOR gate connected to c_3 and c_{out}. The output from this XOR gate indicates whether there is an overflow or not for signed numbers. Whereas, the c_{out} signal alone indicates an overflow for unsigned numbers. Notice that to add more bits to the adder/subtractor circuit, we simply have to daisy chain more of the FA bit slices.

Experiments

1. Implement the circuit shown in Figure 18 (a). You will need to have four FA circuits with their c_{in}'s and c_{out}'s connected together. Connect the 4-bit operand *a* to switches SW7 to SW4. Connect the 4-bit operand *b* to switches SW3 to SW0. Connect the 4-bit result *f* to LED3 to LED0. Connect the carry-out signal (c_{out}) to LED7. Verify that your circuit does subtract two 4-bit binary numbers correctly. The following table lists some sample numbers that you might want to try. The numbers in parenthesis are the corresponding decimal numbers.

Input		**Output**		**Observed Result**
Operand *x*	**Operand *y***	**Carry-out c_{out}**	**Difference *f***	
0111 (7)	0010 (2)	0	0101 (5)	
0101 (5)	0100 (4)	0	1001 (9)	
0110 (6)	1001 (9)	0	1111 (15)	
0111 (7)	1100 (12)	1	0011 (19)	
1001 (9)	1101 (13)	1	0110 (22)	
1111 (15)	1111 (15)	1	1110 (30)	

2. Implement the circuit shown in Figure 18 (b). You may ignore the *Signed_Overflow* XOR gate for now. Verify that your circuit does either add or subtract two 4-bit binary numbers correctly. The following table lists some sample numbers that you might want to try. The numbers in parenthesis are the corresponding decimal numbers.

Input		**Output**		**Observed Result**
Operand *x*	**Operand *y***	**Carry-out c_{out}**	**Result *f***	
0010 (2)	0011 (3)	0	0101 (5)	
0101 (5)	0100 (4)	0	1001 (9)	
0110 (6)	1001 (9)	0	1111 (15)	
0111 (7)	1100 (12)	1	0011 (19)	
1001 (9)	1101 (13)	1	0110 (22)	
1111 (15)	1111 (15)	1	1110 (30)	

2.9 Lab 9: 2-bit Arithmetic and Logic Unit (ALU)

Purpose

In this lab you will learn about the Arithmetic and Logic Unit (ALU). This is the main component inside the microprocessor for performing simple logical and arithmetic operations. The logical operations are like those of the basic logic gates such as the AND, OR and NOT. The arithmetic operations are just simple additions and subtractions. More complex arithmetic operations such as multiply and divide are done in other dedicated components. You will design a 2-bit ALU, implement the circuit, and verify its operations.

Introduction

Like with most circuits, there are many ways of designing the ALU circuit. One method is to do something similar to what we did with the adder/subtractor circuit in Lab 8 by starting out with the basic layout of connecting several full adders (FA) together in series. And just like for the adder/subtractor circuit, we will modify the two operand inputs to the FA appropriately so that the FAs will produce the correct results.

For doing additions and subtractions, we will modify the second (y) operand to the FAs doing something similar to the XOR gate that was used in the adder/subtractor circuit, but with some slight modifications. We will label this sub-circuit black box the AE for Arithmetic Extender.

For doing logical operations, we need to do something a little more extensive. After all, you remember that the FAs can only add numbers, and you saw in Lab 8 how you could use the FAs to both add and subtract numbers. However, the FAs cannot do logical operations, so what we will need to do is to have another sub-circuit to perform the actual logical operations and then pass the result of the logical operations through the FAs to the output. So for logical operations, we do not want the FAs to modify the number that is being passed through its first (x) operand, and the way to do this is to have the FAs add a 0 to the number. We will label this sub-circuit black box the LE for Logic Extender.

Furthermore, depending on the operation that we want to perform, the initial carry-in signal, c_0, has to be set appropriately. In the design, this is done by the Carry Extender, CE, black box.

Select lines are needed to tell the ALU which one of several operations to perform. If we want the ALU to be able to perform four operations, we will need two select lines, s_1 and s_0, because two bits will give four different combinations (00, 01, 10 and 11). Since the outputs of the LE, AE and CE are dependent on which operation we want to perform, therefore, the select lines are also inputs to these three boxes.

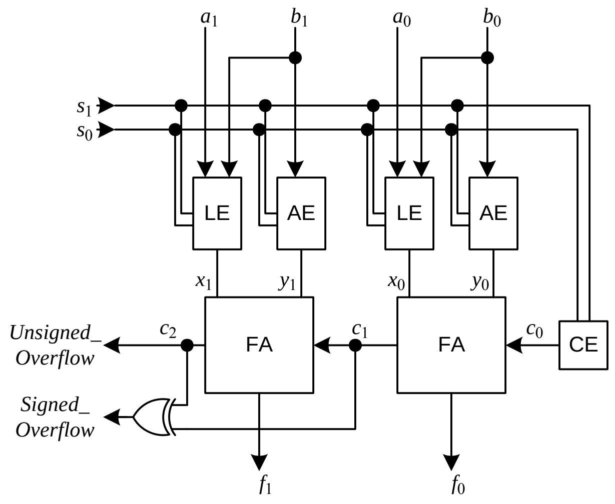


Figure 19: The 2-bit ALU circuit.

The overall circuit for the 2-bit ALU is shown in Figure 19. If you compare this circuit with the adder/subtractor circuit in Figure 18 (b), you will see that they are very similar except for the three added black boxes, LE, AE and CE. And just like with the adder/subtractor circuit, to add more bits to the ALU you simply have to add more of the FA bit slices.

What we now need to do is to come up with the three circuits for the LE, AE, and the CE. The constructions of these circuits, of course, will be dependent on what operations we want to implement, and the way to do it is the same as designing any other combinational circuits. As an example, we will design an ALU that can perform four operations: addition, subtraction, AND, and OR. Following the steps for designing a combinational circuit as outlined in Lab 2, we will first come up with the truth table for each of these three circuits, and then from the truth tables, we can derive the circuits for them.

We start with the operational specifications for our ALU as shown in Figure 20 (a). Note that the assignment of which operation is assigned to which selection combination is arbitrary. So for the combination s_1s_0 = 00, we will perform the addition of $A + B$. And just like for the adder/subtractor circuit, we want the first operand to the FA to be A and the second operand to the FA to be B, thus, the LE will output a_i (the i^{th} bit of A) and the AE will output b_i (the i^{th} bit of B). Recall from the adder/subtractor circuit that for addition, we want the initial carry-in c_0 to be a 0, hence CE will output a 0.

For the combination s_1s_0 = 01, we will perform the subtraction of $A – B$. Recall from the adder/subtractor circuit that for subtraction, we need to invert the second operand B and then add a 1 through the initial carry-in c_0. Hence the AE will output b_i' and the CE will output a 1.

s_1	s_0	Operation Name	Operation	x_i (LE)	y_i (AE)	c_0 (CE)
0	0	Addition	$A + B$	a_i	b_i	0
0	1	Subtraction	$A – B$	a_i	b_i'	1
1	0	Logical AND	A AND B	a_i AND b_i	0	0
1	1	Logical OR	A OR B	a_i OR b_i	0	0

(a)

Op	s_1	s_0	a_i	b_i	x_i (LE)
Add	0	0	0	0	0
	0	0	0	1	0
	0	0	1	0	1
	0	0	1	1	1
Subtract	0	1	0	0	0
	0	1	0	1	0
	0	1	1	0	1
	0	1	1	1	1
AND	1	0	0	0	0
	1	0	0	1	0
	1	0	1	0	0
	1	0	1	1	1
OR	1	1	0	0	0
	1	1	0	1	1
	1	1	1	0	1
	1	1	1	1	1

(b)

Op	s_1	s_0	b_i	y_i (AE)
Add	0	0	0	0
	0	0	1	1
Sub	0	1	0	1
	0	1	1	0
AND	1	0	0	0
	1	0	1	0
OR	1	1	0	0
	1	1	1	0

(c)

s_1	s_0	c_0 (CE)
0	0	0
0	1	1
1	0	0
1	1	0

(d)

Figure 20: ALU operations: (a) function table; (b) LE truth table; (c) AE truth table; (d) CE truth table.

For the two logical operations, the actual operation will be performed in the LE, so the LE will output the result of the respective logical operation. So for the combination $s_1s_0 = 10$ where the ALU will perform the logical AND operation, the LE will output the result of a_i AND b_i. Similarly, for the combination $s_1s_0 = 11$ where the ALU will perform the logical OR operation, the LE will output the result of a_i OR b_i. For all logical operations, we do not want the FAs to add anything, so the AE, which outputs to the second operand of the FA, should output a 0. Similarly, the CE should output a 0.

So from the above analysis, we are able to come up with the three truth tables for the LE, AE and CE as shown in Figure 20 (b), (c) and (d) respectively. Given these three truth tables for the LE, AE and CE, you should now be able to derive their respective circuits. Try to first work it out

2.10 Lab 10: BCD to 7-segment LED Decoder

Purpose

In this lab you will learn how to construct a Binary-Coded-Decimal (BCD) to 7-segment LED display decoder circuit. This circuit converts a 4-bit binary number to cause a 7-segment LED display to show the corresponding decimal digit. You will implement the circuit and verify its operation.

Introduction

BCD is the name given for the 4-bit binary representation of the ten decimal digits. Only the first ten combinations of the 4-bit binary number (from 0000 to 1001) are used as shown in the first two columns (Inputs and Decimal Digit) of the table in Figure 21. For each one of these ten binary combinations, the corresponding decimal digit is to be shown on the 7-segment LED display as shown in the Display column of the table in Figure 21.

Each LED in the 7-segment display has a letter name (from *a* to *g*) given to it as shown in the first row and last seven columns of the table in Figure 21. Hence, to display the decimal digit 0, we want segments *a*, *b*, *c*, *d*, *e* and *f* to be turned on, while segment *g* is turned off. Similarly, to display the decimal digit 1, we want only segments *b* and *c* to be turned on, while the remaining segments are turned off. Continuing on in this fashion, we obtain the rest of the truth table for the seven individual segments as shown in the last seven columns of the table in Figure 21.

The remaining six binary combinations (from 1010 to 1111) are not used in the BCD to 7-segment decoder, therefore, it does not matter what those values are. However, one can do a complete 4-bit to 7-segment hexadecimal decoder in which all of the 16 combinations of the 4-bit binary number are decoded to a hexadecimal digit. See Experiment 4 for the complete 4-bit to 7-segment hexadecimal LED display decoder.

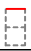
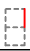
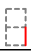
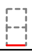
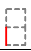
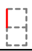
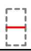
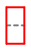
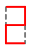
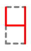
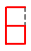
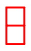
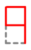
Inputs				Decimal Digit	Display	*a*	*b*	*c*	*d*	*e*	*f*	*g*
i_3	i_2	i_1	i_0									
0	0	0	0	0		1	1	1	1	1	1	0
0	0	0	1	1		0	1	1	0	0	0	0
0	0	1	0	2		1	1	0	1	1	0	1
0	0	1	1	3		1	1	1	1	0	0	1
0	1	0	0	4		0	1	1	0	0	1	1
0	1	0	1	5		1	0	1	1	0	1	1
0	1	1	0	6		1	0	1	1	1	1	1
0	1	1	1	7		1	1	1	0	0	0	0
1	0	0	0	8		1	1	1	1	1	1	1
1	0	0	1	9		1	1	1	0	0	1	1
Rest of the Combinations						×	×	×	×	×	×	×

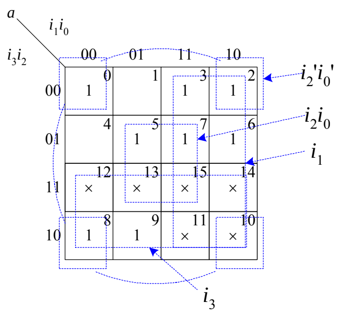
Figure 21: Design and truth table for the BCD to 7-segment LED decoder. The ×'s denote that the value can be either a 0 or a 1.

Having completed the truth table for the decoder, we can continue with the design by drawing the circuit for each of the seven segments. As discussed in Lab 2, given any truth table, we can produce a circuit for it by simply

ANDing the inputs of a row for which the output of that row is a 1, and then ORing the outputs of all the AND gates together. Looking at the column in the truth table for segment *a*, we note that there are eight 1's. Hence we obtain the following Boolean equation.

$$a = i_3'i_2'i_1'i_0' + i_3'i_2'i_1i_0' + i_3'i_2'i_1i_0 + i_3'i_2i_1'i_0 + i_3'i_2i_1i_0' + i_3'i_2i_1i_0 + i_3i_2'i_1'i_0' + i_3i_2'i_1'i_0$$

To implement the circuit for segment *a* based on this equation would require three NOT gates, eight 4-input AND gates and one 8-input OR gate. However, we can get a much smaller circuit by simplifying the equation. There are different methods for simplifying combinational circuits. One combinational circuit simplification method is the use of K-maps[7]. The K-map for simplifying the Boolean equation for segment *a* is shown next.



A K-map is a two dimensional array with the columns and rows labeled with all possible combinations of the binary value for the input variables i_3, i_2, i_1, i_0. Note that the sequence of these binary combinations is always 00, 01, 11, and 10. The eight 1's from the truth table are entered into the corresponding squares in the K-map. For example, in the truth table, there is a 1 for the input combination $i_3i_2i_1i_0 = 0000$. Therefore, a 1 is placed in the square that intersects column 00 and row 00. The ×'s are also placed in the K-map. Next, all adjacent 1's and optional ×'s in the K-map are grouped together to form larger rectangles. Each × can either be considered as a 1 if it helps to make a larger rectangle, or it can be considered as a 0 and be ignored. The number of 1's and ×'s in each of these rectangles has to be a power of 2 (i.e., 1, 2, 4, 8 or 16). This is sort of like a puzzle where you want to have as few rectangles as possible, and each rectangle as large as possible. Each rectangle results in one AND term, and these AND terms are again ORed together to produce the final simplified Boolean equation. From evaluating the above K-map for segment *a*, we obtain the following simpler equation for segment *a*

$$a = i_3 + i_1 + i_2'i_0' + i_2i_0 = i_3 + i_1 + (i_2 \odot i_0)$$

Proceeding in a similar manner, we get the following remaining six simplified equations

$$\begin{aligned}
b &= i_2' + (i_1 \odot i_0)\\
c &= i_2 + i_1' + i_0\\
d &= i_1i_0' + i_2'i_0' + i_2'i_1 + i_2i_1'i_0\\
e &= i_1i_0' + i_2'i_0'\\
f &= i_3 + i_2i_1' + i_2i_0' + i_1'i_0'\\
g &= i_3 + (i_2 \oplus i_1) + i_1i_0'
\end{aligned}$$

Based on these seven simplified equations, we obtain the circuit as shown in Figure 22.

[7] Refer to the book "Digital Logic and Microprocessor Design with VHDL" by E. Hwang for a detail discussion on how to use K-maps to simplify combinational circuits.

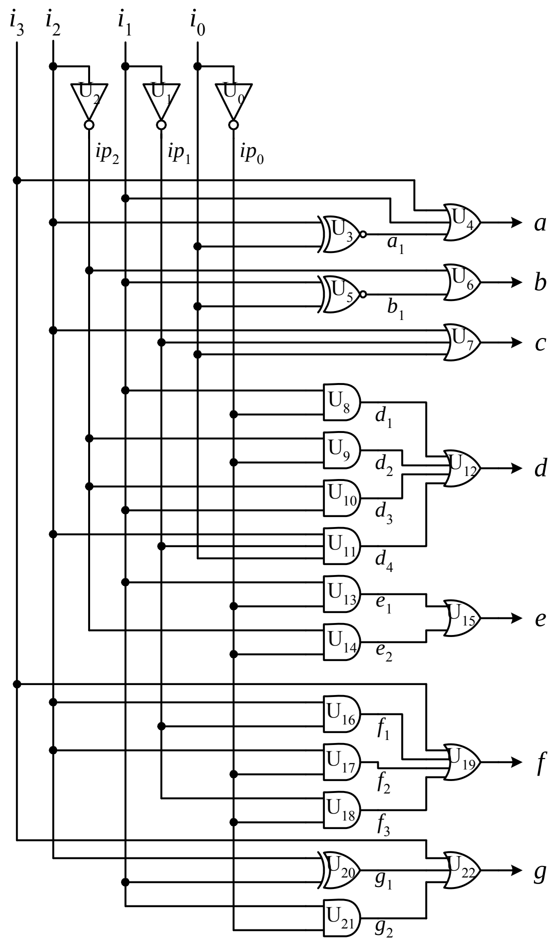


Figure 22: Reduced circuit for the BCD to 7-segment LED decoder.

Experiments

1. Implement the BCD to 7-segment LED display decoder circuit as shown in Figure 22. Connect the four inputs i_3, i_2, i_1, i_0 to four switches, and connect the seven outputs *a*, *b*, *c*, *d*, *e*, *f*, *g* to the 7-segment LED display. Verify that it operates correctly according to the table in Figure 21.

2. Implement the BCD to 7-segment LED display decoder circuit based directly from the original truth table shown in Figure 21 and without doing any simplifications. Verify that it operates correctly according to the table in Figure 21.

3. Recall from Experiment 5 of Lab 2 that you can get the same functional circuit by selecting the rows in the truth table where the output is a 0 instead of a 1 and then inverting the final output. By doing this, you might get a smaller circuit. Use this method to design the circuit for the truth table in Figure 21. Do you get a smaller circuit than the one in Figure 22?

4. Instead of the BCD to 7-segment decoder where only ten of the 16 combinations are used, we can build a complete 4-bit to 7-segment hexadecimal LED display decoder where we decode all of the 16 combinations of the 4-bit binary number. Figure 23 shows the initial design table. Complete the design of this 4-bit to 7-segment hexadecimal LED display decoder circuit and implement it.

Inputs				Decimal Digit	Display	*a*	*b*	*c*	*d*	*e*	*f*	*g*
i_3	i_2	i_1	i_0									
0	0	0	0	0		1	1	1	1	1	1	0
0	0	0	1	1		0	1	1	0	0	0	0
0	0	1	0	2		1	1	0	1	1	0	1
0	0	1	1	3		1	1	1	1	0	0	1
0	1	0	0	4		0	1	1	0	0	1	1
0	1	0	1	5		1	0	1	1	0	1	1
0	1	1	0	6		1	0	1	1	1	1	1
0	1	1	1	7		1	1	1	0	0	0	0
1	0	0	0	8		1	1	1	1	1	1	1
1	0	0	1	9		1	1	1	0	0	1	1
1	0	1	0	A		1	1	1	0	1	1	1
1	0	1	1	B		0	0	1	1	1	1	1
1	1	0	0	C		1	0	0	1	1	1	0
1	1	0	1	D		0	1	1	1	1	0	1
1	1	1	0	E		1	0	0	1	1	1	1
1	1	1	1	F		1	0	0	0	1	1	1

Figure 23: Design of the 4-bit to 7-segment hexadecimal LED display decoder.