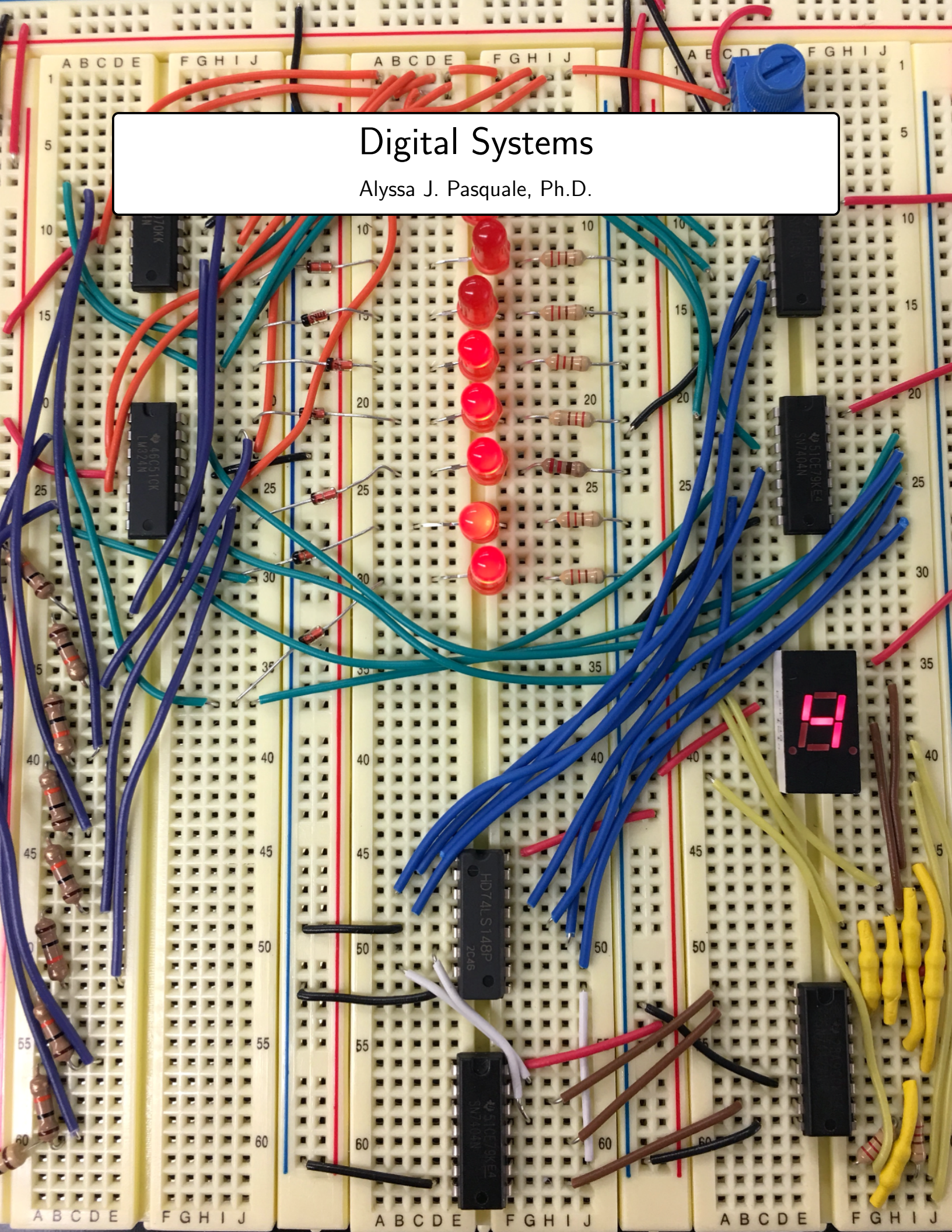

Digital Systems
Alyssa J. Pasquale, Ph.D.

Table of Contents

Author Note **9**

Special Thanks . . . 9

Resources . . . 9

References . . . 9

License and Attribution Information . . . 9

Changelog **11**

1 Number Systems and Binary Arithmetic **13**

1.1 Digital Signals . . . 13

1.2 Foundation of Digital Systems . . . 13

1.3 Number Systems . . . 14

1.4 Decimal (Base 10) . . . 15

1.5 Converting to Decimal . . . 15

1.6 Efficiency of Number Systems . . . 16

1.7 Converting From Decimal . . . 17

1.8 Converting to/from Binary, Octal, and Hexadecimal . . . 19

1.9 Constraints of Binary Numbers . . . 20

1.10 Representation of Non-Integers in Binary . . . 20

1.11 Negative Numbers in Binary . . . 22

1.12 Binary Addition . . . 23

1.13 Binary Subtraction . . . 26

1.14 Binary Multiplication . . . 27

1.15 Binary Codes . . . 30

1.15.1 Binary Coded Decimal (BCD) . . . 30

1.15.2 Weighted Binary Codes (BCD) . . . 30

1.15.3 Gray Code . . . 31

1.15.4 ASCII . . . 32

1.16 Example Problems . . . 33

2 Boolean Algebra **35**

2.1 Logic Gates . . . 35

2.1.1 Buffer . . . 35

2.1.2 Inverter . . . 36

2.1.3 AND . . . 36

2.1.4 OR . . . 37

2.2 Multi-Level Circuits and Boolean Expressions . . . 37
2.2.1 Converting a Circuit Diagram to a Boolean Expression . . . 37
2.2.2 Converting a Boolean Expression to a Circuit Diagram . . . 39
2.3 Truth Tables . . . 40
2.3.1 Converting a Boolean Expression to a Truth Table . . . 40
2.3.2 Converting a Circuit Diagram to a Truth Table . . . 41
2.3.3 Determining Function Equivalence . . . 42
2.4 Boolean Expression Forms . . . 42
2.4.1 Sum of Products . . . 43
2.4.2 Product of Sums . . . 45
2.4.3 Identifying SOP and POS Expressions . . . 47
2.5 Minimizing Boolean Expressions . . . 47
2.5.1 Converting a Circuit Diagram to a Minimum SOP or POS Expression . . . 49
2.5.2 Converting a Truth Table to a Minimum SOP or POS Expression . . . 49
2.5.3 Checking for Consensus Terms . . . 51
2.6 Converting POS to SOP . . . 52
2.7 Converting SOP to POS . . . 53
2.8 Exclusive OR and Equivalence . . . 54
2.8.1 Exclusive OR (XOR) . . . 54
2.8.2 Equivalence (XNOR) . . . 55
2.8.3 Simplifying XOR and XNOR Expressions . . . 56
2.9 Building Circuits . . . 57
2.9.1 TTL Logic . . . 57
2.9.2 7400 Series . . . 58
2.9.3 Input and Output Types . . . 58
2.9.4 Dual In-Line Package (DIP) . . . 59
2.9.5 DIP Switches . . . 60
2.9.6 Output Devices . . . 60
2.10 Example Problems . . . 63

3 Applications of Boolean Algebra 66
3.1 Open-Ended Design . . . 66
3.2 Written or Verbal Parameters . . . 67
3.3 Numeric Parameters . . . 67
3.4 Binary Adders . . . 69
3.4.1 Half Adder . . . 69
3.4.2 Full Adder . . . 69

3.4.3 Ripple Carry Adder . . . 70
3.5 Parity Checking . . . 71
3.5.1 Parity Generation . . . 73
3.5.2 Parity Detection . . . 73
3.5.3 Benefits and Drawbacks to Parity Checking . . . 74
3.6 Incompletely Specified Circuits . . . 74
3.7 Minterm and Maxterm Expressions . . . 77
3.7.1 Minterm Expressions . . . 77
3.7.2 Maxterm Expressions . . . 79
3.7.3 Incompletely Specified Minterm and Maxterm Expressions . . . 81
3.7.4 Converting Minterm and Maxterm Expressions . . . 82
3.7.5 Minterm and Maxterm Expansions . . . 83
3.8 Example Problems . . . 85

4 Karnaugh Maps 87
4.1 Three Variable K-Maps . . . 88
4.2 Four Variable K-maps . . . 91
4.3 Incompletely Specified K-Maps . . . 93
4.4 Implicants, Prime Implicants, and Essential Prime Implicants . . . 94
4.4.1 Implicants . . . 94
4.4.2 Prime Implicants . . . 95
4.4.3 Essential Prime Implicants . . . 95
4.5 Prime Implicant Tables . . . 96
4.6 Five Variable K-Maps . . . 99
4.7 Example Problems . . . 101

5 Quine-McCluskey Method 103
5.1 Example Problems . . . 110

6 NAND and NOR 111
6.1 NAND . . . 111
6.1.1 NAND-Only Circuits . . . 112
6.2 NOR . . . 115
6.2.1 NOR-Only Circuits . . . 116
6.3 Example Problems . . . 121

7 Circuit Optimization **122**

7.1 Two-Level Circuits: Least Time Implementation . . . 122

7.2 Finding Least Cost Implementation . . . 122

7.3 Optimization of Multiple Output Circuits . . . 125

7.3.1 Multiple Output Circuits Using Prime Implicant Tables . . . 129

7.3.2 Finding Multiple Output Implicants Using the Quine-McCluskey Method . . . 136

7.4 Example Problems . . . 147

8 Basic Semiconductor Physics **149**

8.1 P-N Junction . . . 151

8.2 Bipolar Junction Transistor (BJT) . . . 151

8.2.1 Resistor-Transistor Logic (RTL) . . . 152

8.2.2 Transistor-Transistor Logic (TTL) . . . 153

8.3 Metal Oxide Semiconductor Field Effect Transistor (MOSFET) . . . 153

9 Timing **156**

9.1 Timing Diagrams . . . 157

9.2 Static Hazards . . . 159

9.2.1 Identifying and Eliminating Static Hazards . . . 160

9.3 Dynamic Hazards . . . 161

9.4 Example Problems . . . 163

10 Logic Devices **166**

10.1 Buffers and Tri-State Devices . . . 166

10.1.1 Types of Tri-State Buffers . . . 166

10.1.2 Uses of Tri-State Buffers . . . 167

10.2 Multiplexers (MUX) . . . 169

10.2.1 2 to 1 MUX . . . 169

10.2.2 4 to 1 MUX . . . 171

10.2.3 Implementing Boolean Algebra Expressions with a MUX . . . 172

10.3 Sourcing vs. Sinking Current . . . 180

10.4 Demultiplexers (DEMUX) . . . 181

10.5 Using Multiplexers and Demultiplexers Together . . . 185

10.6 Decoders . . . 186

10.6.1 Implementing Boolean Algebra Expressions with a Decoder . . . 187

10.6.2 Expanding Decoders . . . 190

10.6.3 Creating Multiplexers out of Decoders . . . 192

10.6.4 Speciality Decoders . . . 193
10.7 Encoders . . . 194
10.7.1 Priority Encoders . . . 196
10.7.2 Uses of Priority Encoders . . . 197
10.8 Memory . . . 198
10.8.1 Volatile Memory . . . 198
10.8.2 Non-Volatile Memory . . . 199
10.8.3 Implementing Boolean Algebra with ROM . . . 200
10.9 Programmable Logic . . . 202
10.9.1 Programmable Array Logic (PAL) and Generic Array Logic (GAL) . . . 202
10.9.2 PAL/GAL Output Logic Macrocells . . . 204
10.10Example Problems . . . 206

11 Sequential Circuits **212**
11.1 Set-Reset (SR) Latch . . . 212
11.2 Gated SR Latch . . . 214
11.3 Gated Data (D) Latch . . . 215
11.4 D Flip-Flop . . . 215
11.5 Toggle (T) Flip-Flop . . . 217
11.6 SR Flip-Flop . . . 218
11.7 JK Flip-Flop . . . 219
11.8 Setup and Hold Considerations . . . 221
11.9 Converting Flip-Flop Types . . . 222
11.10Asynchronous Flip-Flop Inputs . . . 223
11.11Example Problems . . . 225

12 Registers and Counters **231**
12.1 Registers . . . 231
12.1.1 Serial In / Serial Out (SISO) . . . 231
12.1.2 Serial In / Parallel Out (SIPO) . . . 231
12.1.3 Parallel In / Serial Out (PISO) . . . 232
12.1.4 Parallel In / Parallel Out (PIPO) . . . 233
12.2 Using Registers to Add and Multiply . . . 233
12.2.1 Addition . . . 233
12.2.2 Multiplication . . . 235
12.3 Synchronous Counters . . . 238
12.3.1 Initialization Values . . . 242

12.4 Ripple Counters . . . 244
12.5 Ring Counters . . . 248
12.6 Johnson Counters . . . 249
12.7 Example Problems . . . 250

13 Finite-State Machines **252**
13.1 Moore Machines . . . 252
13.2 Mealy Machines . . . 256
13.3 Sequential Circuit Clock Frequency Constraints . . . 259
13.4 Sequence Detectors . . . 261
13.4.1 Overlapping Sliding Window . . . 261
13.4.2 Non-Overlapping Sliding Window . . . 265
13.4.3 Disjoint Window . . . 266
13.4.4 More Complicated Sequence Detectors . . . 268
13.5 Sequential Comparison Logic . . . 272
13.6 Finite-State Machine Derivation . . . 273
13.6.1 Multiple Outputs . . . 274
13.6.2 Multiple Inputs . . . 276
13.6.3 Incompletely Specified State Machines . . . 277
13.6.4 Alphanumeric State Diagrams and State Diagram Completeness . . . 278
13.7 Reduction of State Tables . . . 280
13.7.1 Visual Reduction of State Tables . . . 280
13.7.2 Implication Tables . . . 282
13.8 State Machine Equivalence . . . 284
13.9 State Assignment . . . 285
13.9.1 One-Hot State Assignment . . . 289
13.10Example Problems . . . 292

14 Solutions to Example Problems **302**
14.1 Chapter 1 . . . 302
14.2 Chapter 2 . . . 303
14.3 Chapter 3 . . . 305
14.4 Chapter 4 . . . 307
14.5 Chapter 5 . . . 313
14.6 Chapter 6 . . . 315
14.7 Chapter 7 . . . 316
14.8 Chapter 9 . . . 317

14.9 Chapter 10 318
14.10Chapter 11 326
14.11Chapter 12 333
14.12Chapter 13 337

Index **344**

Author Note

This book is an attempt to compile together my notes for Digital Systems (ENGIN-2213 at the College of DuPage). I want students to have a free resource that they can use instead of paying hundreds of dollars for a book that they may not read and does not align exactly with my course content.

I also want to include other resources that I've created for students, such as tutorial videos. In addition, I want to include sample circuits that students can build in their free time to add to their knowledge and enjoyment of the topic. Personally, I love building circuits, and I want my students to love it too!

That said, this book will always be a work in progress. I'm sure I will keep adding content to it over time. And I cannot guarantee that it is free from typos. I will, however, do my best to implement your feedback if you find any issues with the text. Feel free to e-mail me at pasqualea185@cod.edu with your notes.

Special Thanks

Special thanks go to all of my students who have contributed to this textbook by finding typos, suggesting new sections to include, and providing feedback on its layout and structure.

Resources

My YouTube Channel has a playlist for Digital Systems related videos. There are many videos with content that supplements the topics covered in this book.

References

Floyd, T. L. (2015). Digital Fundamentals (11th ed.). Pearson Education, Inc.

McCluskey, E. J. (1965). Introduction to the Theory of Switching Circuits. McGraw-Hill Book Company.

Roth, C. H., Jr, & Kinney, L. L. (2014). Fundamentals of Logic Design (7th ed.). Cengage Learning.

Wakerly, J. F. (2000). Digital Design: Principles & Practices (3rd ed.). Prentice Hall.

License and Attribution Information

NC – Only noncommercial uses of the work are permitted

SA – Adaptations must be shared under the same terms

The entirety of this work was created by Alyssa J. Pasquale, Ph.D. The cover photograph is by the author and is a breadboard analog to digital converter. All circuit diagrams and figures in this text were created by the author using LaTeX libraries.

Changelog

Date	Chapter(s)	Description of Change(s)
2020/11/16	all	Corrected minor typos
2020/11/22	1	Corrected mistakes in converting from base 4 to decimal example
2020/11/22	all	Changed figure and table numbers to correspond with the chapter
2021/01/20	12	Added a section on using registers to add and multiply
2021/01/20	13	Added a sub-section on one-hot state assignments
2021/02/15	all	Changed license to CC-BY-NC-SA
2021/03/28	note	Added a description of the CC-BY-NC-SA license in the author note
2021/03/28	all	Modified example box formatting
2021/03/28	3	Corrected Figure 3.2
2021/03/28	3	Corrected minimized expression in alarm clock example
2021/03/28	10	Added paragraph and figure about implementing sequential logic with ROM
2021/07/13	2	Moved videos links to the author note section
2021/07/14	all	Made minor formatting changes
2021/07/14	note	Added attribution information
2021/09/27	14	Corrected typo in binary addition example 2 solution
2021/10/07	10	Corrected typo in example in implementing Boolean algebra with a MUX
2021/10/07	all	Added link colors to make clickable elements more clear
2021/11/20	12	Corrected typo in figure of 3-bit ripple counter from T flip-flops
2021/12/09	12	Added a process flow for designing ripple counters
2021/12/09	2	Added a section on TTL chip input and output types
2021/12/13	14	Corrected typo in four-variable k-map example 3 solution
2021/12/13	14	Corrected typo in PI table example 3 solution
2021/12/13	14	Corrected typo in NAND-only circuits example 4 solution
2021/12/13	14	Corrected typo in least cost implementation example 5 solution
2021/12/13	14	Corrected typo in multiple output optimization example 2 solution
2021/12/13	14	Corrected typo in hazards example 5 solution
2022/04/07	10	Added a section on expanding decoders
2022/04/10	13	Corrected error in state machine equivalence question 5 circuit diagram
2022/04/11	all	Made minor formatting changes
2022/05/04	all	Made minor formatting changes
2022/05/11	all	Added chapter and section headings to top of each page
2022/05/16	11	Added a section on SR flip-flops
2022/05/16	11	Added a section on converting flip-flop types

2022/05/17	index	Added an index
2022/05/17	12	Added a section on ring counters
2022/05/17	12	Added a section on Johnson counters
2022/05/17	13	Added a section on sequential comparison logic
2022/09/30	note	Added a special thanks section
2022/09/30	12	Removed sentence redundancy at start of ripple counter section
2023/02/08	11	Added a section on flip-flop setup and hold time
2023/02/08	13	Added a section on sequential circuit clock frequency constraints
2023/03/02	note	Added a references section
2023/03/08	7	Added a section on multiple-output optimization with PI tables
2023/03/08	7	Added a section on multiple-output optimization with Quine-McCluskey
2023/04/14	9	Corrected typo in static 1 hazard example
2023/04/14	11	Corrected typo in table of SR flip-flop operation
2023/05/04	13	Corrected minor typo in figure caption
2023/05/30	all	Corrected minor typos
2024/02/07	13	Added end of chapter examples on state machine completeness
2024/02/20	14	Corrected typo in minterm expressions example 4 solution
2024/02/22	14	Corrected typo in XOR and XNOR circuits example 5 solution
2024/02/26	14	Corrected typo in maxterm expressions example 4 solution
2024/04/01	14	Corrected typo in timing diagrams example 5 solution
2024/04/01	14	Corrected typo in Quine-McCluskey example 4 solution
2024/04/02	10	Corrected typo related to Figure 10.5
2024/04/12	10	Fixed “4 to 1 DEMUX” to correctly state “1 to 4 DEMUX”
2024/04/19	12	Corrected minor grammatical typo
2024/04/22	14	Corrected typo in load input AND gate of Figure 14.48.
2024/09/24	2	Replaced minimum POS expressions example 5 question and solution
2024/09/30	5	Corrected typo in Quine-McCluskey with don’t cares example
2024/11/04	14	Corrected typo in prime implicant tables example 3 solution
2024/11/04	12 and 13	Corrected all references to flip-flop initialization values
2024/11/04	12	Corrected typo in the caption of Figure 12.14
2024/11/04	12	Corrected typo in the caption of Figure 12.15
2024/11/04	13	Added more information to the section on clock frequency constraints
2024/11/20	all	Corrected minor typos
2025/03/13	5	Corrected minor typo
2025/05/12	14	Corrected typo in circuit diagram shown in Figure 14.40

1 Number Systems and Binary Arithmetic

1.1 Digital Signals

This book is called digital systems. The important word to note is "digital." A digital signal is quantized; that is to say that it can only take on one of a certain number of particular values. This is opposed to analog signals, which can take on any one of a continuum of values.

A good example of a digital value would be to ask how many people are in a room, or how many marbles are in a jar. Because there cannot be half people or third-of-a-marbles, or negative people, the number of values that could be listed in these scenarios are defined: positive integers.

An analog value would be the collective weight of a roomful of people. There is no reason that our scale couldn't read 10,102.574 lbs, or 9,942.$\overline{3}$ lbs, for example. Any one of an infinite possibility of positive real numbers could define the weight of a roomful of people.

In electrical engineering classes, we are interested in digital vs. analog signals. A signal would be the voltage measured over a particular circuit element. A digital signal is one that can only take on particular values. Particularly, in this book, digital signals will only be 0 V or 5 V. There are no other possible values that are compatible with our digital electronic system. Figure 1.1 is an example of a digital signal.

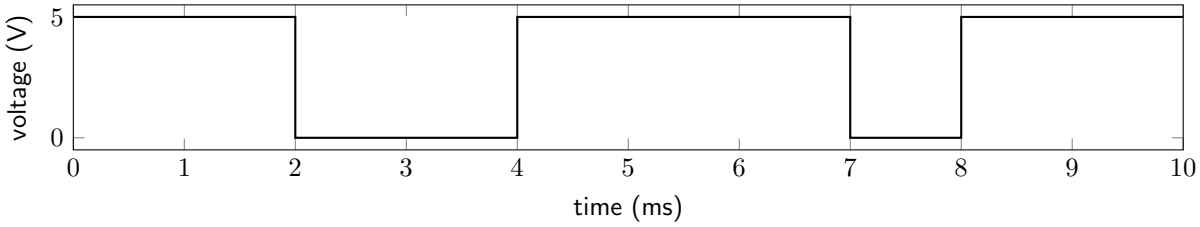


Figure 1.1: An example of a digital signal.

Digital systems are generally easier to design than analog systems and require much less math to analyze and understand.

As we will see later in this book, even digital signals are truly analog at heart. But for the most part, we can "hand-wave" away the analog nature of things in this course and focus on the digital.

1.2 Foundation of Digital Systems

To be able to design a digital circuit, we need to progress in stages.

First, we need to understand binary. As our signals can take on only one of two values, we use the binary number system to quantify information. Therefore, this book starts by teaching how to convert between number systems with the goal of being able to derive binary numbers.

Second, we need to understand Boolean algebra. If binary forms the "words" then Boolean algebra is the "language" that links them together. Boolean algebra is how we start to create logical systems, and we

will spend considerable effort into learning how to create Boolean expressions and how to minimize them so that they are easy to build.

Third, we will learn about logic gates. Logic gates form the building blocks of our digital circuits. From these simple building blocks we can create circuits from something very simple to something very complicated. Remember: your computer, at heart, is built from these simple building blocks!

1.3 Number Systems

Digital systems make use of the binary number system. Many students are nervous about learning a new number system. They are comfortable with and familiar with decimal (base 10) because it is so commonly used in our lives. So to provide you with some comfort, I'd like to point out how many number systems you are familiar with.

- **Base 10 (decimal)** – this is the standard number system that you use all the time in your math classes, to figure out how much money it will cost to buy several things at the store, or to do pretty much any arithmetic.
- **Base 12** – the month that I am currently writing this in is September. What month will it be ten months from now? How do we know that ten months after September comes July? There are twelve months in a year, so we are using a base 12 number system when we do this reasoning. If it is currently 5:04 p.m., what time will it be eight hours from now? Our 12-hour clock is also a base 12 system!
- **Base 24** – if you are familiar with the 24-hour clock, then you have used this number system to determine how many hours it will be, say, thirteen hours from now.
- **Base 7** – today is Thursday, so what day will it be four days from now? As there are seven days in a week, we have to use a base 7 number system any time we do this type of reasoning.

Hopefully you get the idea that you are already used to many different number systems, most of which relate to our strange time-keeping systems (365 days in a year – most of the time, twelve months in a year, twenty-eight/twenty-nine/thirty/thirty-one days in a month, seven days in a week, fifty-two weeks in a year, twenty-four hours in a day, sixty minutes in an hour, sixty seconds in a minute, etc).

Why is it that computers and digital signals are binary? Why not decimal, to make things simpler? It is difficult to make things take on three or more distinct values. The electrical switches (transistors) that are used only create two values: 0 V (represented as the binary number 0) and 5 V (represented as the binary number 1). In the past, punched cards were used to create computers going back to the 1800s when the Jacuqard loom was created. These punch cards have holes in particular locations. A hole can either be present or absent, which corresponds to a binary system.

A benefit of digital systems derives from their digital nature. Analog signals are very susceptible to noise, which is not so for digital signals. It is very easy to differentiate between 0 V and 5 V. It is not so easy to differentiate between (say) 3.1 V and 3.12 V in an analog system.

1.4 Decimal (Base 10)

To understand number systems in general, let's first review the decimal number system. Each numeral in a decimal number is actually related to a power of ten. Take a look at the example shown below in Figure 1.2. The number is 358.917, but each of the numerals has a meaning.

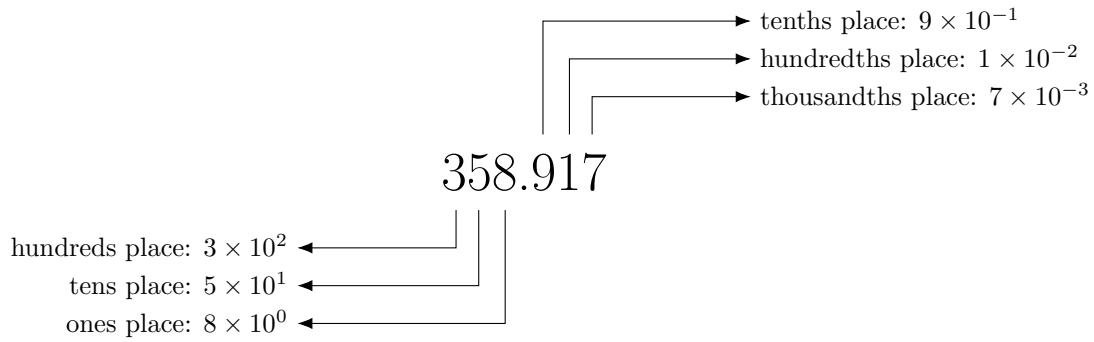


Figure 1.2: The base 10 system explained.

Each of the numerals is related to a power of ten. That power of ten gives the numeral its weight. The 3, for example, isn't just 3. It's 300, or 3×10^2. The 5 is actually 50, or 5×10^1.

To determine the power of ten that each number corresponds to, determine how far away from the decimal point it is. To the left, the powers of ten increase from zero upward. To the right, the powers of ten decrease from negative one downward.

In general, all numbers in different number systems are just numerals that can be raised to powers of their base. If the base is seven, then each numeral relates to powers of seven. In binary, each numeral relates to powers of two.

1.5 Converting to Decimal

This realization that all number systems are composed of raising numbers to powers of the base can be used to convert numbers from any number system to decimal. Following are some examples. The subscripts following the number indicate the number system (or base).

Example: Converting from base 4 to decimal

Convert 1312.01 from base 4 to decimal.

$$\begin{aligned}1312.01_4 &= (1 \times 4^3) + (3 \times 4^2) + (1 \times 4^1) + (2 \times 4^0) + (0 \times 4^{-1}) + (1 \times 4^{-2}) \\ &= 64 + 48 + 4 + 2 + 0.0625 \\ &= 118.0625_{10}\end{aligned}$$

Example: Converting from base 2 to decimal

Convert 11011.1101 from binary to decimal.

$$\begin{aligned}11011.1101_2 &= (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2}) \\ &\quad + (0 \times 2^{-3}) + (1 \times 2^{-4}) \\ &= 16 + 8 + 2 + 1 + 0.5 + 0.25 + 0.0625 \\ &= 27.8125_{10}\end{aligned}$$

Example: Converting from base 16 to decimal

Some number systems have bases that are greater than ten. In that case, we need to use different symbols to represent numbers that are larger than nine. The purpose of a number system is to represent each of the numerals in only one digit. For example, the number ten in decimal is not a single digit. It is two digits (a one and a zero). Hexadecimal, which can represent numerals from zero through fifteen in one digit, needs to "squeeze" the numbers larger than nine into one digit. After nine, the letters A–F are used instead of numbers.

$$\begin{aligned}4F30.A_{16} &= (4 \times 16^3) + (F \times 16^2) + (3 \times 16^1) + (0 \times 16^0) + (A \times 16^{-1}) \\ &= (4 \times 16^3) + (15 \times 16^2) + (3 \times 16^1) + (0 \times 16^0) + (10 \times 16^{-1}) \\ &= 16384 + 3840 + 48 + 0.625 \\ &= 20272.625_{10}\end{aligned}$$

1.6 Efficiency of Number Systems

If we look at numbers in various number systems, we can see that number systems with a large base are more efficient than number systems with a small base. In other words: we can pack more information into less space (fewer digits) using something like base 16 over using a number system like base 2 or base 3.

The numbers in table 1.1 are all identical, but the ones represented in number systems with a smaller base require more digits to express the number than the number systems with a larger base.

Base	Number
2	10000100000
3	1110010
4	100200
5	13211
6	4520
7	3036
8	2040
9	1403
10	1056
16	420

Table 1.1: The number 1056_{10} expressed in several different number systems.

1.7 Converting From Decimal

As noted, binary forms the basis for digital circuits, because there are only two possible values for the logic signals. We can picture binary values as numbers (0 or 1), as a signal value (LOW or HIGH), as the status of an electrical switch (which is ON or OFF), as a voltage (0 V or 5 V), or as notation that you would see on a circuit diagram (Vcc, GND). These are depicted in table 1.2.

Binary Value	Signal Value	Switch Status	Voltage	Circuit Notation
0	LOW	OFF (open)	0 V	GND
1	HIGH	ON (closed)	5 V	Vcc

Table 1.2: Binary values, signal value, switch status, voltages, and circuit notation in digital logic.

It is therefore important to learn how to convert numbers from decimal to binary. (You might be wondering what to do if you have a number in any other base and need to convert it to binary. Convert it to decimal, and then convert that to binary.)

Rather than multiplying by powers of the base, we now divide by the base and find remainders to obtain the solution. Let's start with a decimal number to understand the process before moving to binary.

To convert 358_{10} to our desired base (which is ten), we divide repeatedly by ten. The remainders provide the digits of our converted number.

Decimal – Decimal

$$
\begin{array}{rll}
10\,\overline{)\,358} & & \\
10\,\overline{)\,35} & \text{R: } 8 \rightarrow \text{LSB} & \\
10\,\overline{)\,3} & \text{R: } 5 & \\
0 & \text{R: } 3 \rightarrow \text{MSB} &
\end{array}
\qquad
\begin{array}{ccc}
\underline{3} & \underline{5} & \underline{8} \\
\text{MSB} & & \text{LSB}
\end{array}
$$

The first remainder that we get from our division is called the LSB, which stands for least significant bit. It gives the ones place, the digit that is closest to the decimal point all the way to the right.

The MSB is the most significant bit. It is the digit that is farthest left from the decimal point. It has the most weight of all of the digits in the number.

How do we know when we are done converting? We are done when the quotient of our division step is

zero. In fact, we could keep dividing, and obtain a lot more zeros to place to the left of our M.S.B. We know from algebra that placing zeros to the left of a number does not change its value.

Let's now divide 358_{10} by two to obtain a binary number. (Remember: we want to obtain a base 2 number, so we divide by the base, which in this case is two.) Just as before, the first remainder is the LSB and the last remainder is the MSB.

358_{10} – Binary

$2 \overline{)358}$	
$2 \overline{)179}$	R: 0 → LSB
$2 \overline{)89}$	R: 1
$2 \overline{)44}$	R: 1
$2 \overline{)22}$	R: 0
$2 \overline{)11}$	R: 0
$2 \overline{)5}$	R: 1
$2 \overline{)2}$	R: 1
$2 \overline{)1}$	R: 0
0	R: 1 → MSB

$$\underset{\text{MSB}}{\underline{1}} \quad \underline{0} \quad \underline{1} \quad \underline{1} \quad \underline{0} \quad \underline{0} \quad \underline{1} \quad \underline{1} \quad \underset{\text{LSB}}{\underline{0}}_2$$

This process of successive division can be used to convert numbers into any base, not just binary.

To convert a number to base 5, for example, just divide by five repeatedly and find the remainders. The first remainder will always be the LSB.

230_{10} – Base 5

$5 \overline{)230}$	
$5 \overline{)46}$	R: 0 → LSB
$5 \overline{)9}$	R: 1
$5 \overline{)1}$	R: 4
0	R: 1 → MSB

$$\underset{\text{MSB}}{\underline{1}} \quad \underline{4} \quad \underline{1} \quad \underset{\text{LSB}}{\underline{0}}_5$$

Note that this method only works for integer values. If a number has a fractional component, we need to use a different system to convert it, as will be discussed later.

Remember: the procedure for converting from a number that is not in base 10, to another number that is not in base 10, is to convert to decimal in between. For example, to convert 152_6 to base 9, you need to first convert to decimal.

$$\begin{aligned} 152_6 &= (1 \times 6^2) + (5 \times 6^1) + (2 \times 6^0) \\ &= 36 + 30 + 2 \\ &= 68_{10} \end{aligned}$$

Once we have converted to decimal, we now divide by nine to find the number in base 9. We see that $152_6 = 68_{10} = 75_9$.

68_{10} – Base 9

$9\overline{)68}$

$9\overline{)7}$ R: $5 \rightarrow$ LSB

$9\overline{)0}$ R: $7 \rightarrow$ MSB

1.8 Converting to/from Binary, Octal, and Hexadecimal

The only exception to the rule of going to decimal in between is when you are converting binary to/from octal or hexadecimal. This has to do with the fact that all of these number systems are powers of two. If you forget this fact, **you can always use decimal as your in-between number**, but it is faster not to.

Every hexadecimal digit corresponds to four bits of binary (called a nibble), as shown in table 1.3.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 1.3: Decimal, binary, and hexadecimal conversions.

Therefore, if you have a binary number, you can break it up into nibbles, starting with the four least significant bits (the bits closest to the decimal point, all the way to the right). Each of those nibbles will be represented by a single hexadecimal digit, as shown in Figure 1.3. In other words, $1101010001110001111_2 = 6A38F_{16}$.

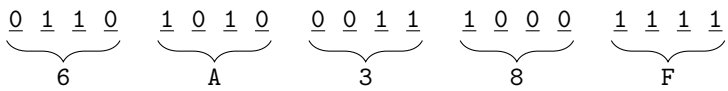


Figure 1.3: Converting binary to hexadecimal is as easy as splitting up the nibbles.

The reverse process is also true. A binary number can be generated by taking each hexadecimal digit and expanding it into the corresponding four binary bits.

$$FCA01_{16} = 1111\ 1100\ 1010\ 0000\ 0001_2$$

Binary can be converted to octal by splitting up the number into groups of three bits, as shown in Figure 1.4. In other words, $1101010001110001111_2 = 1521617_8$.

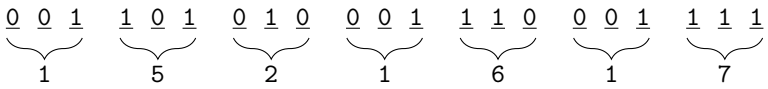


Figure 1.4: Converting binary to octal occurs in groups of three bits.

As with hexadecimal, the reverse conversion (octal to binary) occurs by taking the octal digit and expanding it into the corresponding three binary bits.

1.9 Constraints of Binary Numbers

Now that we have an understanding of how to convert numbers to binary, we need to understand the constraints of digital systems using binary. Digital systems are designed to work with a certain number of bits. For example, the Arduino microcontroller is an 8-bit system, so all of the data that it deals with come in groups of eight bits. Your computer is probably a 64-bit system, which is capable of dealing with much larger numbers. Either way, there is a constraint to the largest possible number that can be portrayed in a given number of bits. It will be important for us to understand this limitation whenever we are doing arithmetic with binary numbers.

Table 1.4 shows the maximum decimal number that can be represented, given a certain number of bits both with unsigned (only positive) binary numbers, and with signed (binary numbers can be either positive or negative). We will discuss how to create negative binary numbers later in this chapter. As you can see, the number of bits that we have relates to the largest possible number that we can represent in binary.

Bits	Decimal Range (Unsigned)	Decimal Range (Signed)
1	$0 \rightarrow 1$	N/A
2	$0 \rightarrow 3$	$-2 \rightarrow 1$
3	$0 \rightarrow 7$	$-4 \rightarrow 3$
4	$0 \rightarrow 15$	$-8 \rightarrow 7$
5	$0 \rightarrow 31$	$-16 \rightarrow 15$
6	$0 \rightarrow 63$	$-32 \rightarrow 31$
8	$0 \rightarrow 255$	$-128 \rightarrow 127$
16	$0 \rightarrow 65{,}535$	$-32{,}768 \rightarrow 32{,}767$
n	$0 \rightarrow 2^n - 1$	$-2^{n-1} \rightarrow 2^{n-1} - 1$

Table 1.4: The range of decimal numbers for signed and unsigned binary numbers of differing numbers of bits.

1.10 Representation of Non-Integers in Binary

Frequently, we need to express non-integer quantities. If we purchase something that is $14.25, how can we store that number in a computer? Binary numbers can only be expressed with two characters: zero and one. There is no decimal point in binary. Fractional numbers are expressed in computer systems using something

called floating-point math. However, floating-point conversions are beyond the scope of this book. We will use a different method to come up with fractional numbers in binary, for now using a decimal point and ignoring that it would not be allowed in a true computer or microcontroller system.

Converting a number that has both integer and fractional parts from decimal to binary should always occur in two steps. First, convert the integer part using the process above. Second, convert the fractional part using the process described below.

To derive the integer binary number from a decimal number, divide by two. To derive the fractional part, multiply by two. The integer part of the product represents the digit that should go after the decimal point. Continue to multiply the fractional parts by the base (two for binary) until you end up with zero.

Let's convert from decimal to decimal once more just to understand the process. Repeatedly multiply the fractional part of the number by the base (in this case, ten).

$$\begin{aligned} 0.25 \times 10 &= \mathbf{2}.5 \text{ (2 is the most significant fractional bit)} \\ &- 2.0 \\ &= 0.5 \\ 0.5 \times 10 &= \mathbf{5}.0 \text{ (5 is the next fractional bit)} \\ &- 5.0 \\ &= 0.0 \text{ (result is 0, you can stop)} \end{aligned}$$

Example: Non-integer binary number

Convert 0.25 from decimal to binary.

$$\begin{aligned} 0.25 \times 2 &= \mathbf{0}.5 \text{ (0 is the most significant fractional bit)} \\ &- 0.0 \\ &= 0.5 \\ 0.5 \times 2 &= \mathbf{1}.0 \text{ (1 is the next fractional bit)} \\ &- 1.0 \\ &= 0.0 \text{ (result is 0, you can stop)} \\ 0.25_{10} &= 0.01_2 \end{aligned}$$

Sometimes, numbers that have non-repeating fractional parts in decimal have repeating fractional parts in binary. Numbers can be strange when you convert to and from different number systems! When doing the fractional conversion, if you notice a number repeating itself, it means if you were to continue multiplying, you would be stuck in a loop forever, continuing to multiply. At the point where you reach the repeat number, simply draw the overbar over the repeating part to represent the repeating fraction, as you would

in decimal.

Example: Repeating fraction in binary

Convert 0.4_{10} to binary.

$$0.4 \times 2 = \mathbf{0}.8 - 0.0 = 0.8$$
$$0.8 \times 2 = \mathbf{1}.6 - 1.0 = 0.6$$
$$0.6 \times 2 = \mathbf{1}.2 - 1.0 = 0.2$$
$$0.2 \times 2 = \mathbf{0}.4 - 0.0 = 0.4$$

Note that the result is 0.4, which is the number we started with. Therefore,

$$0.4_{10} = 0.\overline{0110}_2$$

1.11 Negative Numbers in Binary

Just as fractional numbers are difficult to represent in binary (because there is no '.' included in a number system that only understands zero and one), it is also not possible to express negative numbers by sticking a negative sign '$-$' in front of them. We therefore need to find a number system that satisfies the following requirements.

1. All numbers (both positive and negative) only consist of zeros and ones.
2. We can add, subtract, multiply, and divide with the numbers (both positive and negative) and obtain meaningful results.

There are many ways to satisfy the first criterion, but very few ways to satisfy the second criterion. For example, we could take binary numbers and just stick a zero at the beginning to signify that the number is positive, or stick a one at the beginning to signify that the number is negative. This is known as a sign and magnitude number. For example, if we have a 5-bit binary system and wanted to define $+14_{10}$ and -14_{10}:

$$14_{10} = 01110_2$$
$$-14_{10} = 11110_2$$

To test to see if our number system is useful, we could add together both of these numbers and hope that we obtain zero. That would not be the case. That means that this system for representing signed (positive and negative) numbers is not useful.

The number system that satisfies both of the above criteria for a useful signed number system is called 2's complement. It enables us to quickly establish if a number is positive or negative, and also enables us to

carry out the arithmetic operations of addition, subtraction, multiplication, and division.

To generate a number in 2's complement, simply follow the steps below.

1. Start with an n-bit positive binary number (if you are working with eight bits, that means an 8-bit positive binary number).
2. If you want a positive number, you're done! Positive numbers are universal (meaning they are both "normal" binary and also 2's complement simultaneously).
3. To obtain a negative number, simply
   (a) invert all of the bits (zeros become ones and vice versa), and
   (b) add one to the result.

For example, let's represent the number 4_{10} as a 2's complement number using six bits.

$$4_{10} = 000100_2$$

Representing positive numbers in 2's complement is as easy as ensuring that we have a sufficient number of bits in the value. Let's represent -4_{10} as a 2's complement number using six bits.

$$\begin{aligned} -4_{10} &= \\ &= 000100 \text{ (start with +4 in 6-bit binary)} \\ &= 111011 \text{ (flip all of the bits)} \\ &= 111100 \text{ (add one)} \end{aligned}$$

1.12 Binary Addition

Being able to perform arithmetic in binary is essential to creating microcontrollers and computers. As seemingly simple as it is, addition is a critical building block in arithmetic and logic units (ALUs) in computers.

We start by considering a 1-bit binary adder (that is, the adder sums together two 1-bit binary numbers). Remembering from our primary school days, we know that we have a sum and also a carry term that might possibly result from our addition. A truth table helps us to understand the outputs that result when we add together two numbers. Table 1.5 shows the possible results for $F = S_1 + S_2$, with C_{OUT} representing the carry out term.

When adding numbers in binary, it is important to understand how many bits our system is. All of the numbers that we add together must be comprised of that many bits. (For example: if we have an 8-bit system, we must add together two 8-bit numbers. It is not OK to add together a 3-bit number to a 4-bit number in an 8-bit system. The math will not work out if there are negative numbers involved!)

S_1	S_2	F	C_{OUT}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1.5: Truth table for binary addition.

Just as with adding numbers together in decimal, start by taking two binary numbers, line them up together, and start from the rightmost column. Any carry out then gets cascaded into the next column. For this reason, it is important to understand how to add together three numbers: S_1, S_2, and C_{IN} (the carry in term from the previous column). Table 1.6 shows the truth table for addition with a carry in term.

S_1	S_2	C_{IN}	F	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1.6: Truth table for binary addition with a carry in term.

It is also important to understand whether or not the answer is valid. As seen in a previous section of this book, there is a limit to how large each binary number can be when expressed in a certain number of bits. When adding together two large numbers, it is possible to get an answer that is too large to represent in the number of bits allotted. This situation is called an overflow, and we need to understand how to identify when they occur. Overflows occur in addition when

- the addition of two positive binary numbers results in a negative sum, or
- the addition of two negative binary numbers results in a positive sum.

Using 2's complement, any carries that are generated by the leftmost column get ignored. It is a feature of the 2's complement number system that we can ignore any final carry out terms. This is very important to note, because a frequent misunderstanding of students is that a carry out term represents an overflow situation. That is not true. Overflow can only be checked by testing the above criteria. Carry out terms are ignored.

Now that we understand how to add, and how to identify an overflow situation, let's look at some examples. In each example, the numbers in gray at the top represent the carry terms from previous columns.

Example: Binary addition without overflow and no final carry out

In a 5-bit system, add 3_{10} to 5_{10}.

$$\begin{array}{cccccc} & & 1 & 1 & 1 & \\ & 0 & 0 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 & 0 & 0 \end{array}$$

Example: Binary addition without overflow and with a final carry out

In a 4-bit system, add 4_{10} to -2_{10} (Note that there is a carry out term from the leftmost column here that we will ignore. There is no overflow because adding a positive and a negative number cannot possibly result in an overflow!)

$$\begin{array}{ccccc} 1 & 1 & & & \\ & 0 & 1 & 0 & 0 \\ + & 1 & 1 & 1 & 0 \\ \hline & 0 & 0 & 1 & 0 \end{array}$$

Example: Binary addition with overflow and no final carry out

In a 4-bit system, add 4_{10} to 5_{10}. (Note that we would expect a positive result, but the sum is negative, which indicates an overflow.)

$$\begin{array}{ccccc} & 1 & & & \\ & 0 & 1 & 0 & 0 \\ + & 0 & 1 & 0 & 1 \\ \hline & \boxed{1} & 0 & 0 & 1 \end{array}$$

Example: Binary addition with overflow and a final carry out

In a 4-bit system, add -6_{10} to -3_{10}. (We would expect the result of this operation to be negative, but our sum is positive, indicating an overflow.)

$$\begin{array}{ccccc} 1 & & & & \\ & 1 & 0 & 1 & 0 \\ + & 1 & 1 & 0 & 1 \\ \hline & \boxed{0} & 1 & 1 & 1 \end{array}$$

1.13 Binary Subtraction

A similarly important basic operation to perform with binary numbers is subtraction. A truth table can be used to represent the operation $F = S_1 - S_2$. And just as with decimal subtraction, at times we need to borrow a digit from columns over to the left. That borrow is represented as the B column on the truth table shown in table 1.7.

S_1	S_2	F	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Table 1.7: Truth table for binary subtraction.

It is possible to subtract two binary numbers and obtain an invalid result (overflow). Overflows occur in subtraction when

- subtracting a negative number from a positive number results in a negative sum, or
- subtracting a positive number from a negative number results in a positive sum.

Because it is possible to subtract a negative number from a positive number, at times we need to borrow when we are in the left-most column of numbers. While it appears that there is nowhere to borrow from, a feature of 2's complement is that we can create a borrow (conjured out of thin air) and borrow from it. One of the examples, below, shows this seemingly magical borrowing process.

Example: Binary subtraction without overflow

In a 5-bit system, subtract 5_{10} from 14_{10}.

$$\begin{array}{rccccc} & & & & \color{gray}{0} & \color{gray}{10} \\ & 0 & 1 & 1 & \cancel{1} & \cancel{0} \\ - & 0 & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 & 0 & 1 \end{array}$$

Example: Binary subtraction without overflow

In a 4-bit system, subtract -3_{10} from -2_{10}

$$\begin{array}{rcccc} & & & \color{gray}{0} & \color{gray}{10} \\ & 1 & 1 & \cancel{1} & \cancel{0} \\ - & 1 & 1 & 0 & 1 \\ \hline & 0 & 0 & 0 & 1 \end{array}$$

Example: Binary subtraction with overflow

In a 4-bit system, subtract 4_{10} from -5_{10}. (Note that we would expect a negative number from this result, but we get a positive number indicating that there was an overflow.)

$$\begin{array}{rcccc} & {\scriptstyle 0} & {\scriptstyle 10} & & \\ & \cancel{1} & \cancel{0} & 1 & 1 \\ - & 0 & 1 & 0 & 0 \\ \hline & 0 & 1 & 1 & 1 \end{array}$$

Example: Binary subtraction with overflow

In a 5-bit system, subtract -6_{10} from 10_{10}.

$$\begin{array}{rccccc} & {\scriptstyle 0} & & & & \\ \cancel{1} & {\scriptstyle 10} & & & & \\ & \cancel{0} & 1 & 0 & 1 & 0 \\ - & 1 & 1 & 0 & 1 & 0 \\ \hline & 1 & 0 & 0 & 0 & 0 \end{array}$$

1.14 Binary Multiplication

Multiplication is the last type of arithmetic that we will discuss in this book. We will use a truth table to represent how to perform the multiplication process of $F = S_1 \times S_2$. Note that in multiplication, there are no borrows or carries shown on the truth table. (However, in multiplication, there are addition steps that need to be performed, so carries may still need to be used later in the process.) The truth table for binary multiplication is shown in table 1.8.

S_1	S_2	F
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.8: Truth table for binary multiplication.

Multiplication is a difficult process, both for humans and for computers. Particularly cumbersome is the need to be able to determine if an overflow occurs. Because it is not clear from doing regular multiplication if an overflow occurred, we need to do a special process in multiplication.

1. Start with an n-bit 2's complement number, using the process discussed earlier in this chapter to generate 2's complement.

2. Double the number of bits. These new bits (called overflow bits) will go to the left of the 2's complement number.
   (a) If the number is positive, these overflow bits will all be zero.
   (b) If the number is negative, these overflow bits will all be one.

It cannot be emphasized enough that the use of these overflow bits is mandatory. You simply cannot get an accurate result if you do not use them. Now that we have generated overflow bits, we can use them to check if there is an overflow. An overflow results if

- all of the overflow bits of the final result do not agree with each other, or
- the overflow bits all agree with each other, but they disagree with the sign bit of the non-overflow bits of the result.

At times, multiplying in binary can lead to a lot of addition steps. To minimize the number of addition steps, place the number with the fewest number of ones on the bottom row of your multiplication. (For every one in the binary number on the bottom row, you will have that many terms to add together.) Because addition is commutative, it doesn't matter if you do 2×-1 or -1×2. Secondly, if you do still have many addition steps, it can be difficult to deal with the number of carries that you will generate. In that case, add the first two numbers together, then add to that the third number, and so on. (In other words: add your addition steps in groups of two rather than trying to add them all at once.)

Example: Binary multiplication of positive numbers without overflow

In a 4-bit system, multiply 3_{10} and 2_{10}.

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \times & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}$$

Example: Binary multiplication of negative numbers without overflow

In a 4-bit system, multiply -3_{10} and -2_{10}.

$$\begin{array}{cccccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ \times & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ \hline & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ + & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}$$

Binary multiplication of positive numbers with overflow

In a 4-bit system, multiply 5_{10} and 3_{10}. (Note that even though the four overflow bits agree with each other, they indicate a positive number, whereas the sign bit of the result indicates a negative number. This is an overflow situation.)

$$\begin{array}{cccccccccc} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \times & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ + & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \times \\ \hline & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$

Binary multiplication of negative numbers with overflow

In a 4-bit system, multiply 5_{10} and -4_{10}. It will be much easier to place the negative number on the top row of our multiplication, because it will mean far fewer numbers to add together in the second step of our multiplication process.

$$\begin{array}{cccccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ \times & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ + & 1 & 1 & 1 & 1 & 0 & 0 & \times & \times \\ \hline & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \end{array}$$

1.15 Binary Codes

To conclude this chapter, we will learn about different types of binary codes that might be encountered when designing digital systems. Each binary code is used for particular applications.

1.15.1 Binary Coded Decimal (BCD)

Binary coded decimal (BCD) is used to represent decimal numbers in a binary format (hence the name). Each nibble of binary data is used to encode a single decimal digit. This is extremely useful when displaying numbers, such as the time, on displays. If the time is 9:52, it is much easier to send the nine, the five, and the two to their own separate displays rather than encode 952_{10} as a binary number (1110111000_2) and then decode that back into ones and tens places for both minutes and hours.

Instead, we would represent 9:52 in BCD. Because there are four decimal numbers in the time (the leading zero is usually left off when we write numbers in 12-hour time), we would need sixteen bits to represent the time in BCD, as shown in Figure 1.5.

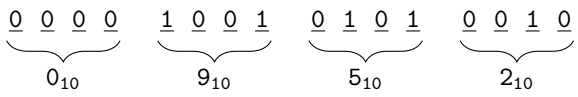


Figure 1.5: The decimal number 952 represented in BCD.

1.15.2 Weighted Binary Codes (BCD)

A weighted code just means that each digit in the binary number is assigned some type of weight. The "normal" binary code that we have already used is a weighted code. It is called 8-4-2-1 code in that those are the weights of each digit in a nibble (2^3-2^2-2^1-2^0).

Some other weighted codes were devised by mathematicians or engineers for specific purposes. These other codes can be judged to be complete or not (in other words: can you generate all decimal numbers from zero through nine in the weighted code to express numbers in BCD).

One complete weighted code is 6-3-1-1. Rather than the powers of two that we are accustomed to from 8-4-2-1 binary, the MSB of the nibble is multiplied by six. The next bit is multiplied by three, and the least significant two bits are multiplied by one. This can then be used as a BCD code to represent different decimal numbers. These 6-3-1-1 values are shown in table 1.9.

Decimal	6-3-1-1	Expansion			
0	0000				
1	0001				(1×1)
2	0011			(1×1)	(1×1)
3	0100		(1×3)		
4	0101		(1×3)		(1×1)
5	0111		(1×3)	(1×1)	(1×1)
6	1000	(1×6)			
7	1001	(1×6)			(1×1)
8	1011	(1×6)		(1×1)	(1×1)
9	1100	(1×6)	(1×3)		

Table 1.9: 6-3-1-1 weighted code.

The number 952_{10} is shown converted to 6-3-1-1 BCD in Figure 1.6.

$$\underbrace{0\ 0\ 0\ 0}_{0_{10}}\quad\underbrace{1\ 1\ 0\ 0}_{9_{10}}\quad\underbrace{0\ 1\ 1\ 1}_{5_{10}}\quad\underbrace{0\ 0\ 1\ 1}_{2_{10}}$$

Figure 1.6: The decimal number 952 represented in 6-3-1-1 BCD.

Another possible weighted code could be 6-2-1-1. Remember that the completeness of a binary code is determined by whether or not it is able to represent all decimal numbers from zero through nine. It is not possible to generate the number 5_{10} using this code. Therefore it is not a complete binary code.

1.15.3 Gray Code

Gray code represents a non-weighted code. It is useful in that from moving from any one binary number to the next, the number of bits that change in the process is only one. We will see why that is useful later in this book when we use Karnaugh maps to solve Boolean algebra expressions, and when we use a similar type of mapping to minimize expressions in sequential logic.

The way that we will use Gray code, each binary number has exactly the same representation in our 8-4-2-1 weighted code as before. The only difference is in the *order* that the numbers are arranged.

To generate the ordering of Gray code, we start with 1-bit Gray code (which is exactly the same as 1-bit binary). 0–1. Only one bit changes going from the first number to the second.

To generate 2-bit Gray code, start with the 1-bit Gray code. Think of it as existing on a mirrored surface, where it sees its reflection beneath. Then, take the top digits and place a zero in front. The digits "under the mirror" get a one in front. This is 2-bit Gray code, and this generation process is shown in Figure 1.7.

$$\begin{array}{c}0\\1\\\hline 1\\0\end{array}\quad\rightarrow\quad\begin{array}{cc}0&0\\0&1\\\hline 1&1\\1&0\end{array}$$

Figure 1.7: Generating 2-bit Gray code.

A similar process is used to generate 3-bit Gray code, and is shown in Figure 1.8.

0	0		0	0	0
0	1		0	0	1
1	1		0	1	1
1	0		0	1	0
1	0	→	1	1	0
1	1		1	1	1
0	1		1	0	1
0	0		1	0	0

Figure 1.8: Generating 3-bit Gray code.

1.15.4 ASCII

ASCII is a binary code that takes numbers, letters, and other characters such as punctuation and encodes them into binary numbers. When you type words into your computer, they need to be stored as binary numbers as your computer does not understand anything that is not a zero or a one. If you do an Internet search for "ASCII character table," you will see all of the encodings for each alphanumeric character.

1.16 Example Problems

Number System Conversions

1. Convert 413.1_5 to unsigned binary. Express the non-integer part exactly, indicating any repeat digits.
2. Express -120_{10} as an 8-bit 2's complement number.
3. Convert 27.77_{10} as an unsigned binary number, rounding the non-integer component to fit into three spaces.
4. Convert $AF013C_{16}$ to a base 4 number.
5. Convert 101101.11_2 to a base 7 number. Express the non-integer part exactly, indicating any repeat digits.

Binary Addition

1. In a 5-bit system, add 15_{10} and -4_{10}. If there is an overflow, indicate how you identified it.
2. In a 5-bit system, add -13_{10} and 5_{10}. If there is an overflow, indicate how you identified it.
3. In a 5-bit system, add -6_{10} and 8_{10}. If there is an overflow, indicate how you identified it.
4. In a 5-bit system, add 12_{10} and 10_{10}. If there is an overflow, indicate how you identified it.
5. In a 5-bit system, add -14_{10} and 7_{10}. If there is an overflow, indicate how you identified it.

Binary Subtraction

1. In a 5-bit system, subtract 10_{10} from -8_{10}. If there is an overflow, indicate how you identified it.
2. In a 4-bit system, subtract -1_{10} from 0_{10}. If there is an overflow, indicate how you identified it.
3. In a 6-bit system, subtract -7_{10} from -24_{10}. If there is an overflow, indicate how you identified it.
4. In a 4-bit system, subtract -5_{10} from 2_{10}. If there is an overflow, indicate how you identified it.
5. In a 4-bit system, subtract -7_{10} from 5_{10}. If there is an overflow, indicate how you identified it.

Binary Multiplication

1. In a 5-bit system, multiply 10_{10} and -3_{10}. If there is an overflow, indicate how you identified it.
2. In a 4-bit system, multiply 2_{10} and -4_{10}. If there is an overflow, indicate how you identified it.
3. In a 5-bit system, multiply 9_{10} and -1_{10}. If there is an overflow, indicate how you identifed it.
4. In a 5-bit system, multiply -8_{10} and 3_{10}. If there is an overflow, indicate how you identified it.
5. In a 6-bit system, multiply -6_{10} and 3_{10}. If there is an overflow, indicate how you identified it.

Binary Codes

1. Express 352_{10} as an 8-4-2-1 BCD number.
2. Express 352_{10} as a 6-3-1-1 BCD number.
3. Look up the ASCII encoding for lowercase q and express as a 7-bit binary number.
4. List the numbers in 4-bit Gray code, in order.
5. Is 2-4-2-1 a complete binary code? If so, list the encodings for numbers zero through nine. If not, which numbers cannot be encoded in this binary code?

2 Boolean Algebra

Boolean algebra gives us a method to take our ideas for a digital circuit and turn them into a reality. For now, we will create circuits that can be expressed in three forms. Any one of the forms is enough to completely identify the circuit.

- Circuit diagram – a schematic drawing of how the circuit looks, using logic gates as building blocks.
- Truth table – a table showing all of the different combinations of inputs and their corresponding outputs.
- Boolean algebra expression – an equation that acts as a shorthand for the circuit.

Each of these representations is useful in its own right. A circuit diagram is the most useful when you are building a circuit on a breadboard and need to know which logic gates to use. A Boolean expression takes up the least amount of space to write and is very concise while still containing all of the information needed to identify the circuit. A truth table can be a valuable way to understand how a circuit works before reducing it to an expression or drawing a circuit diagram.

2.1 Logic Gates

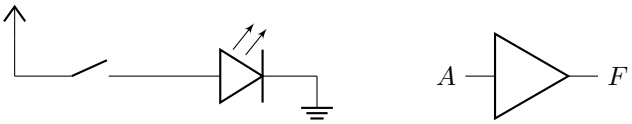
Logic gates are devices that carry out certain Boolean (logical) functions. They are represented by a pictorial digram, called a circuit symbol, that goes into a circuit diagram. The four basic, universal logic gates are described in this section. (Universal means that we can make any digital logic circuit using only these gates, regardless of how complicated the circuits are.)

Each logic gate will be expressed as a switch circuit (a circuit built out of switches), a circuit symbol, a truth table, and a Boolean expression. Recall from the previous chapter that binary is used in computation because of the ability to express the two possible states of our logic devices, as described in table 1.2.

For each of the following gates, the switch circuit can be interpreted as follows: each switch corresponds to an input. The status of the switch corresponds to the binary value. If a connection between 5 V and ground exists from one end of the LED to the other, then the LED would be on and the output of the logic gate is a one. If the connection between 5 V and ground is broken, the LED would be off and the output of the logic gate is a zero.

2.1.1 Buffer

A buffer takes an input signal and passes it through to the output unchanged. While this might seem like something we could do with a wire, without having to use an actual logic gate, an actual buffer in a circuit has the benefit of boosting the amount of current in the circuit, which maintains sufficient signal strength in a circuit with a lot of elements in it. A buffer is shown in Figure 2.1.



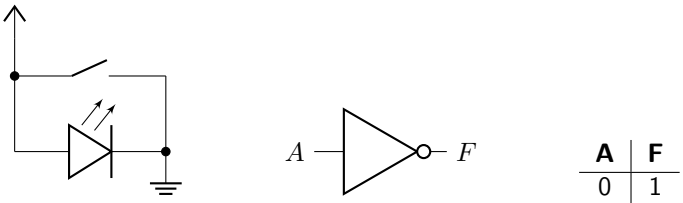
A	F
0	0
1	1

Figure 2.1: A buffer represented as a switch circuit, a circuit symbol, and a truth table.

2.1.2 Inverter

The inverter is also known as a NOT gate. It takes the complement (opposite) of the input and passes it through to the output. The small bubble indicates complement.

The Boolean expression for an inverter is $F = A'$. The $'$ symbol is called "prime" and indicates that a complement is occurring. The inverter is shown in Figure 2.2.



A	F
0	1
1	0

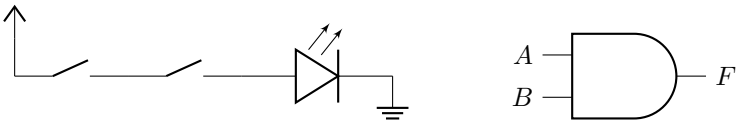
Figure 2.2: An inverter represented as a switch circuit, a circuit symbol, and a truth table.

Note in the switch circuit for the inverter, when the switch is closed (logical one), a short circuit is created, causing all of the current to flow through the wire including the switch, and none of the current to flow through the wire that contains the LED.

2.1.3 AND

An AND gate is a logic gate that has an output of one only when all of the inputs are one.

The Boolean expression for a 2-input AND gate is $F = AB$. While this may look similar to an arithmetic operation (multiplication), it is not multiplication. It is an AND operation. To determine when you should multiply and when you should perform this Boolean AND operation, look at the context. In this book, you will always be asked to "multiply" when you need to multiply. If you see two variables together in an expression such as $Z = XY$, then an AND operation is taking place. A 2-input AND gate is shown in Figure 2.3.



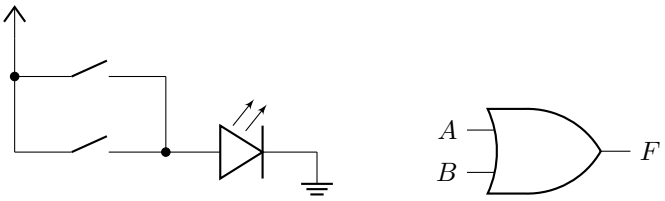
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2.3: An AND gate represented as a switch circuit, a circuit symbol, and a truth table.

2.1.4 OR

An OR gate is a logic gate that has an output of one any time one or more inputs is one.

The Boolean expression for a 2-input OR gate is $F = A + B$. This might look similar to addition, but it is NOT arithmetic. It is a Boolean (logical) operation. In this book, it will be very clear when you need to add two numbers together, as the word "add" will be used. Any time you see the + sign connecting two variable names together, you can be sure that it is an OR operation. A 2-input OR gate is depicted in Figure 2.4.



A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.4: An OR gate represented as a switch circuit, a circuit symbol, and a truth table.

2.2 Multi-Level Circuits and Boolean Expressions

More complicated circuits can be represented by stringing logic gates together. We will start by representing these circuits as circuit diagrams and Boolean algebra expressions, and then we will learn how to include truth tables in the mix.

It is important to point out that there are many different Boolean algebra expressions that correspond to any given circuit. However, we usually want to find a minimized form, as the minimization allows us to build the circuit with the fewest number of logic gates. We will start by considering Boolean expressions in general, and then we will learn how to reduce them to their minimum forms.

2.2.1 Converting a Circuit Diagram to a Boolean Expression

A circuit diagram is read roughly from left to right until an expression is created. Essentially, you find the output of each logic gate (using the expressions for each individual logic gate, given above) and then cascade those outputs into the next logic gates, and so on.

For now, the goal is to obtain a Boolean expression. Later on we will worry about making it a minimized expression.

There are three different ways to represent a complemented (inverted, primed) input to a logic gate. The preference used in this book is to use a bubble on any inverted input. However, there are two other ways to do this. These three ways are shown in Figure 2.5.

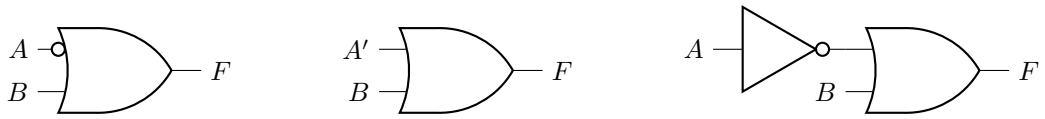
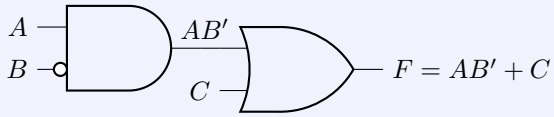


Figure 2.5: Using an OR gate to express the function $F = A' + B$, with the inversion of A represented in three different ways: a bubble, a prime symbol ($'$), and an input inverter.

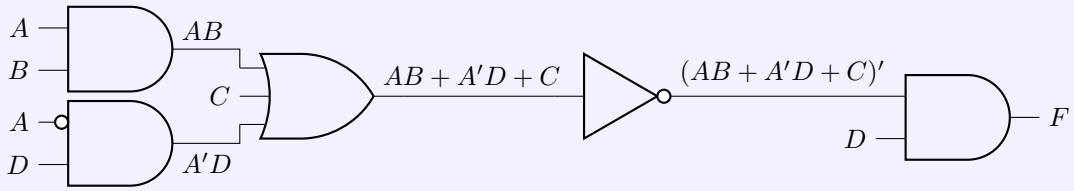
Be careful not to do any two of these at once. If you take the complement of a complement, you wind up with what you started with!

Example: Obtaining a Boolean expression from a circuit diagram



Starting from the left, we find that the output of the AND gate is AB'. That output is then fed into an OR gate along with C, leading to an output of $F = AB' + C$.

Example: Obtaining a Boolean expression from a circuit diagram



This example is slightly more complicated, but still uses the same methodology to determine the Boolean expression. The AND gates all the way to the left are analyzed, and the outputs of AB and $A'D$ are determined. Those outputs then go into the OR gate which has an output of $AB + A'D + C$, which feeds into an inverter $(AB + A'D + C)'$, and into an AND gate along with D to give the final expression of $F = D(AB + A'D + C)'$.

For now, we lack the ability to deal with inverters that occur at the output of any logic gate (whether it be an AND gate or an OR gate) other than to just put the preceding terms into parenthesis and stick a prime at the end. We will learn how to reduce inverted outputs once we get further into this chapter.

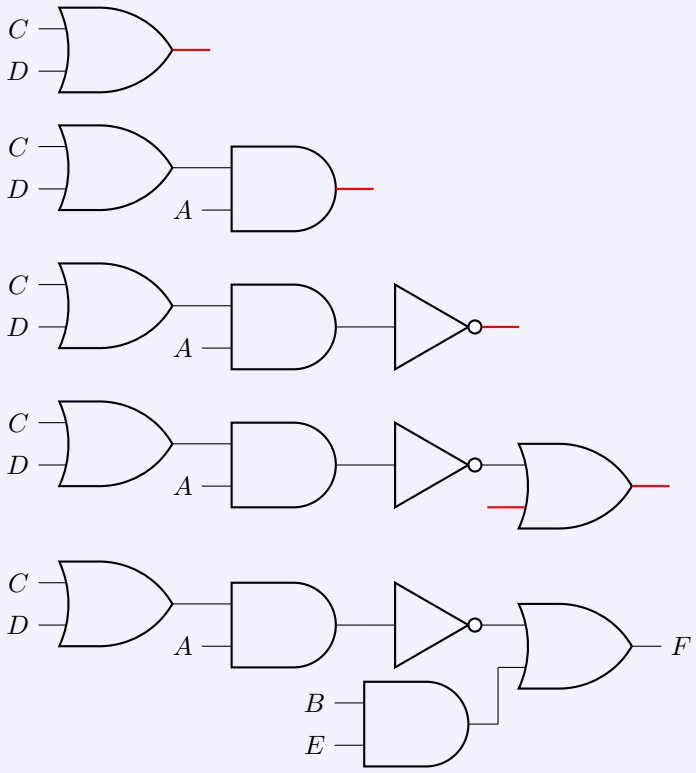
2.2.2 Converting a Boolean Expression to a Circuit Diagram

We will now do the reverse operation from what we did above. Now we would like to take a circuit diagram and obtain from it a Boolean expression. Rather than going from left to right, we are now going to go from "inside-out" to obtain our circuit diagram.

Identify the operation that occurs nested within the most parenthesis, and draw that logic gate with its inputs first. Then determine what type of logic gate that goes into, and draw that. Continue until you have created a logic gate for every term in the expression.

Example: Obtaining a circuit diagram from a Boolean expression

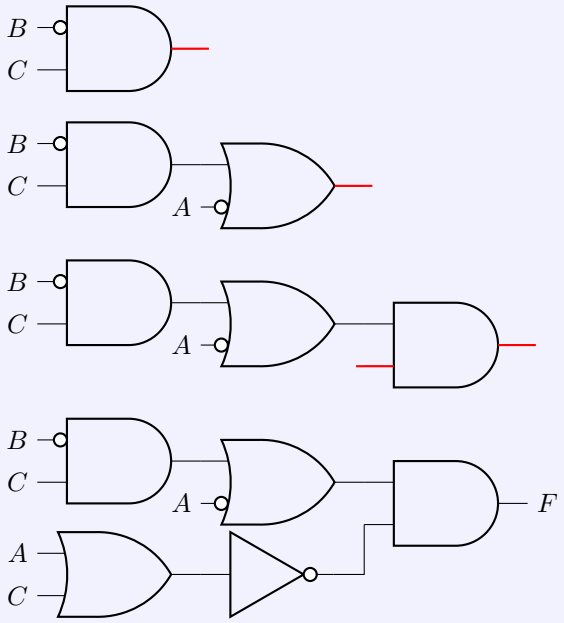
Draw a circuit diagram corresponding to $F(A, B, C, D, E) = [A(C + D)]' + BE$. Going from the inside-out, start with the OR gate with inputs C and D. That then feeds into an AND gate with another input of A. All of that feeds into a NOT gate, which is then one input into an OR gate. The OR gate has a second input of B and E. The red wires indicate an incomplete connection at each step.



Example: Obtaining a circuit diagram from a Boolean expression

Draw a circuit diagram corresponding to $F(A, B, C) = (A + C)'(A' + B'C)$. Start with an AND gate with inputs B' and C. This feeds into an OR gate along with A'. There is also an OR gate with A and C as inputs. This is complemented and sent to an AND gate with the previous statement. The

incomplete connections are depicted in red in the intermediate steps.



2.3 Truth Tables

Truth tables show all possible combinations of input values for a circuit and their corresponding outputs. You may also include other columns that show intermediate steps if it is helpful to do so.

Truth tables are important in that they can fully identify a circuit in a way that a Boolean expression cannot. Two expressions that have identical truth tables are said to be equivalent. They may have different Boolean expressions, but if the truth tables are the same, then the circuits are functionally identical.

To determine how many different possible combinations of inputs there are, we can identify the relationship between the number of inputs in a circuit and the number of rows in a truth table. When there are two inputs, there are four possible outputs (truth table rows). When there are three inputs, there are eight possible outputs. When there are four inputs, there are sixteen possible outputs. Therefore, if there are n inputs, there are 2^n rows on the truth table. Later on in this chapter we will learn a shorthand that makes identifying circuits easier than drawing out truth tables that may have many rows.

2.3.1 Converting a Boolean Expression to a Truth Table

To convert a Boolean expression to a truth table, first draw out all of the rows of the truth table and all of the input combinations. Analyze each of the input values and determine what the output(s) to each Boolean expression would be. If it helps to include intermediate steps in the truth table, then include those!

Example: Obtaining a truth table from a Boolean expression

Derive a truth table for the expression $F = AB' + C$.

A	B	C	B'	AB'	F
0	0	0	1	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	1	0	0	1

2.3.2 Converting a Circuit Diagram to a Truth Table

To convert a circuit diagram to a truth table, either first convert that circuit diagram to a Boolean expression and then follow the steps above, or simply determine what the output of every logic gate would be in each case and follow the signals through to the output.

Figure 2.6 depicts $F = AB + C'$ for all of the eight different possible combination of input values. Each of the HIGH signals are depicted with red wires, and each of the LOW signals are depicted with black wires. A truth table can then be created with all of this data.

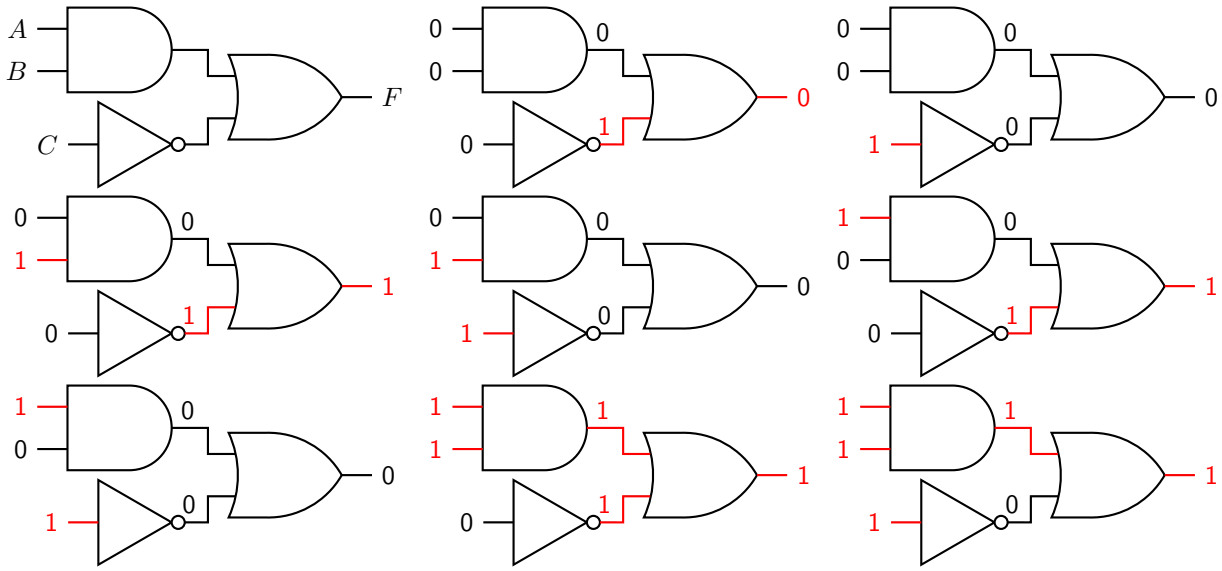


Figure 2.6: Deriving a truth table from $F = AB + C'$ by analyzing the circuit diagram for each combination of input values.

2.3.3 Determining Function Equivalence

As mentioned, a truth table can be used to identify equivalent circuits. If the original expression is $F = AB' + C$, we can see if either one of the following expressions is equivalent by creating truth tables.

- $F = (A + C)(A + B')$
- $F = (A + C)(B' + C)$

The truth table for $(A + C)(A + B')$ is shown in table 2.1, along with $AB' + C$. As you can see, the outputs are not identical for all of the different input combinations. Therefore the two expressions are not equivalent.

A	B	C	A+C	A+B'	(A+C)(A+B')	AB'+C
0	0	0	0	1	0	0
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	1	1	0	0	1
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	0
1	1	1	1	1	1	1

Table 2.1: Comparisons of the truth tables of $(A + C)(A + B')$ and $AB' + C$.

The truth table for the second expression, $(A + C)(B' + C)$ is shown in table 2.2 along with $AB' + C$. These two expressions are equivalent, because they have identical outputs for every possible combination of input values. The Boolean algebra expressions are different, but the truth tables are identical.

A	B	C	A+C	B'+C	(A+C)(B'+C)	AB'+C
0	0	0	0	1	0	0
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	1	1
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	1	0	0	0
1	1	1	1	1	1	1

Table 2.2: Comparisons of the truth tables of $(A + C)(B' + C)$ and $AB' + C$.

2.4 Boolean Expression Forms

Before we determine how to create Boolean expressions from a truth table, we need to learn about different ways to format those Boolean expressions. There are two main types of formats that can be used. It is important to understand that while there are many different ways to express Boolean expressions, these are two very formal types of expressions. They therefore come with particular rules.

2.4.1 Sum of Products

The first type of Boolean expression is called sum of products (SOP). Based on the logic symbols that we use to express AND and OR operations, sum of products implies that the expression consists of AND gates that output into an OR gate, as depicted in Figure 2.7.

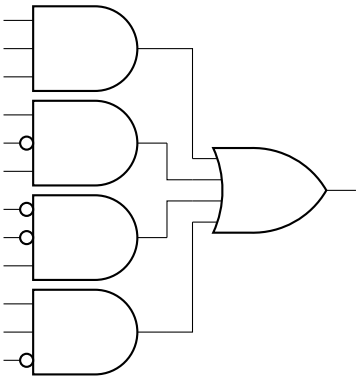


Figure 2.7: Template of an example sum of products (SOP) expression.

In an SOP expression, all of the truth table outputs that are one are analyzed. Any time there is an output of one, it implies that the OR gate in the circuit has at least one input that is equal to one. In order to generate a value of one from an AND gate, all of the inputs to that AND gate must be one. In this manner, we can look at every output on a truth table that is one and determine how the inputs must be entered into an AND gate to create an output of one. That is called the product expression. Each product expression is then OR'd together (summed) to obtain the final SOP expression.

The procedure for deriving an SOP expression from a truth table is therefore to

1. identify the rows in the truth table that have an output of one,
2. determine what product term would have to exist for the inputs in that row to give an output of one in an AND gate (this is called the product term for that row), and then
3. sum together all of the product terms to obtain the SOP expression.

We will use the truth table in table 2.3 as an example for deriving an SOP expression from a truth table.

A	B	C	F
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

Table 2.3: Example truth table for deriving an SOP expression.

The rows in the truth table that have an output of one are rows one, three, six, and seven. Let us identify what the outputs of AND gates would be for each of these rows. Then let us determine how we could reconfigure those inputs to obtain AND gate outputs of ones.

Figure 2.8 shows row one as is inputted into an AND gate. It would lead to an output of zero. If, however, A and B were primed, then the output of the AND gate would be a one. Therefore, the product term for row one is $A'B'C$.

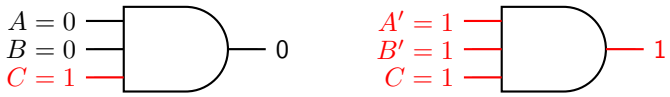


Figure 2.8: Row one (ABC=001) and how to generate an output of one from an AND gate given the inputs.

The product term for row three, shown in Figure 2.9 shows that the product term is $A'BC$.

Figure 2.9: Row three (ABC=011) and how to generate an output of one from an AND gate given the inputs.

The product term for row six, shown in Figure 2.10 shows that the product term is ABC'.

Figure 2.10: Row six (ABC=110) and how to generate an output of one from an AND gate given the inputs.

Row seven, as depicted in Figure 2.11 does not require any modification to obtain the product term ABC.

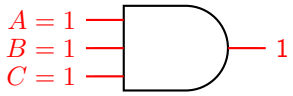


Figure 2.11: Row seven (ABC=111) and how to generate an output of one from an AND gate given the inputs.

The sum of products expression for the truth table given in table 2.3 is $F = A'B'C+A'BC+ABC'+ABC$. This is not a minimized expression, but it is an SOP expression nonetheless.

If you notice, creating a product term for each row requires either taking an input variable as is if it has a value of one, or taking the prime if it has a value of zero. Therefore row one, 001, becomes $A'B'C$, forcing the A and B to take on values of one.

2.4.2 Product of Sums

The second type of Boolean expression is called product of sums (POS). Based on the logic symbols that we use to express AND and OR operations, product of sums implies that the expression consists of OR gates that output into an AND gate, as depicted in Figure 2.12.

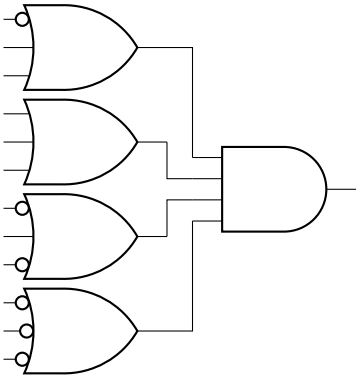


Figure 2.12: Template of an example product of sums (POS) expression.

In a POS expression, the idea is to analyze the outputs of the truth table for that expression that are equal to zero. Any time there is an output of zero, it implies that the AND gate in the circuit has at least one input that is equal to zero. In order to generate a value of zero from an OR gate, all of the inputs to that OR gate must be zero. In this manner, we can look at every output on a truth table that is zero and determine how the inputs must be entered into an OR gate to create an output of zero. That is called the sum expression. Each sum expression is then AND'd together to obtain the final POS expression.

The procedure for deriving an POS expression from a truth table is therefore to

1. identify the rows in the truth table that have an output of zero,
2. determine what sum term would have to exist for the inputs in that row to give an output of zero in an OR gate (this is called the sum term for that row), and then

3. AND together all of the sum terms to obtain the POS expression.

We will use the same truth table as above, table 2.3, as an example for deriving a POS expression from a truth table. The rows in the truth table that have an output of zero are rows zero, two, four, and five. Let us identify what the outputs of OR gates would be for each of these rows. Then let us determine how we could reconfigure those inputs to obtain OR gate outputs of zeros.

Figure 2.13 shows row zero as is inputted into an OR gate. It requires no change to obtain an output value of zero. Therefore, the sum term for row zero is $(A + B + C)$.

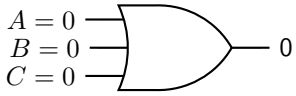


Figure 2.13: Row zero (ABC=000) in the truth table, and how to generate an output of zero from an OR gate given the inputs.

The sum term for row two, shown in Figure 2.14 shows that the product term for row two needs to be $(A + B' + C)$ in order to generate an output of zero on the OR gate.

Figure 2.14: Row two (ABC=010) in the truth table, and how to generate an output of zero from an OR gate given the inputs.

Row four requires variable A to be primed in order to obtain an output of zero on the OR gate, as shown in Figure 2.15. Its corresponding sum term is therefore $(A' + B + C)$.

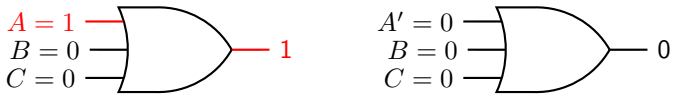


Figure 2.15: Row four (ABC=100) in the truth table, and how to generate an output of zero from an OR gate given the inputs.

Lastly, row five, as depicted in Figure 2.16 has a sum term of $(A' + B + C')$.

Figure 2.16: Row five (ABC=101) in the truth table, and how to generate an output of zero from an OR gate given the inputs.

Therefore, the product of sums expression for the truth table given in table 2.3 is $F = (A + B + C)(A + B' + C)(A' + B + C)(A' + B + C')$.

If you notice, creating a sum term for each row requires either taking an input variable as is if it has a

value of zero, or taking the prime if it has a value of one. Therefore row two, 010, becomes $(A + B' + C)$, forcing the B to take on values of zero.

2.4.3 Identifying SOP and POS Expressions

It is important to identify what is SOP and what is POS, as these formats will be used frequently in this book. Both of these formats are defined very formally. While many valid Boolean expressions may exist, only certain conditions of variables are considered SOP or POS.

Of the following expressions, both of them are equivalent, but only one is SOP.

- $F = A(B + C) + C'D' + B'D$ – this is not SOP because one of the terms has been factored out
- $F = AB + AC + C'D' + B'D$ – this is SOP

Of the following expressions, both are equivalent, but only one is POS.

- $Z = (A + D)(A + B')(B + E + F)$ – this is POS
- $Z = (A + B'D)(B + E + F)$ – this is not POS because one of the terms has been factored out

If it is not clear to you if an expression is in SOP or POS format, draw a circuit diagram. If there are more than just two levels of logic gates (AND leading into OR, or OR leading into AND), then it is not in SOP or POS format.

Expressions with only one term are both SOP and POS. For example, $F = AB$ is simultaneously in SOP and POS format. $F = A + B$ is similarly both SOP and POS.

2.5 Minimizing Boolean Expressions

When actually building a circuit, it is in our best interest to use as few logic gates as possible. For this reason, we want to find minimum Boolean expressions. First, we find our SOP or POS expression. Then, we can use the rules of Boolean algebra to minimize the expression. We will expend considerable effort to find minimized SOP and POS expressions.

The rules of Boolean algebra are given below. These rules can be applied to expressions to find a minimum expression.

Operations with 0 and 1:

$$X + 0 = X$$
$$X + 1 = 1$$
$$X \cdot 0 = 0$$
$$X \cdot 1 = X$$

Idempotent laws:

$$X + X = X$$
$$X \cdot X = X$$

Involution law:

$$(X')' = X$$

Complementary laws:

$$X + X' = 1$$
$$X \cdot X' = 0$$

Distributive laws:

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

Absorption theorem:

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

Commutative laws:

$$X + Y = Y + X$$
$$XY = YX$$

DeMorgan's laws:

$$(X + Y)' = X'Y'$$
$$(XY)' = X' + Y'$$

Elimination theorem:

$$X + X'Y = X + Y$$
$$X(X' + Y) = XY$$

Associative laws:

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

Uniting theorem:

$$XY + XY' = X$$
$$(X + Y)(X + Y') = X$$

Consensus theorem:

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

Many of these rules of Boolean algebra look different from the "normal" rules of "regular" algebra that you are used to. If you aren't sure how they are derived, then you can create a truth table for both sides of the equation and convince yourself that both sides of the equation are equivalent.

Some of the rules of Boolean algebra can be verified just by applying some of the other rules. For example, the second distributive law $X + YZ = (X + Y)(X + Z)$ (which students generally have a hard time believing, as it is so different from "regular" algebra), is derived below.

$$\begin{aligned} X + YZ &= (X + Y)(X + Z) \\ &= XX + XZ + XY + YZ \text{ (first distributive law)} \\ &= X + XZ + XY + YZ \text{ (idempotent law)} \\ &= X(1 + Z + Y) + YZ \text{ (first distributive law)} \\ &= X + YZ \text{ (operations with 1 and 0)} \end{aligned}$$

The procedure for minimizing Boolean algebra expressions is to apply any of the possible rules that are given above. Continue until none of the rules can be applied, then the expression is minimized. Now we are able to derive minimum SOP and POS expressions from circuit diagrams or from truth tables, closing the circle of circuit diagrams / Boolean expressions / truth tables.

2.5.1 Converting a Circuit Diagram to a Minimum SOP or POS Expression

There are always two options when confronted with a circuit diagram and asked to generate a minimum SOP or POS expression. The first option is to create a truth table. The second option is to generate a non-minimized Boolean expression and then minimize it using the rules of Boolean algebra.

Figure 2.17 shows a circuit diagram that is clearly not minimized.

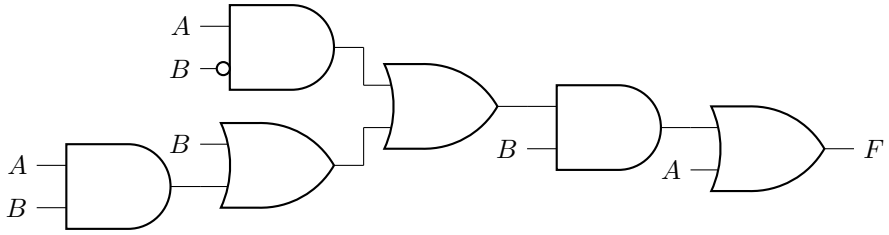


Figure 2.17: A non-minimum circuit diagram.

It is possible to derive an expression for F by using the procedure outlined above. Then, the rules of Boolean algebra are applied to obtain a minimum expression.

$$\begin{aligned}(AB' + (AB + B))B + A &= \\ (AB' + B)B + A &= \text{(absorption theorem)} \\ (A + B)B + A &= \text{(elimination theorem)} \\ AB + BB + A &= \\ AB + B + A &= \\ B + A &= \text{(absorption theorem)} \\ &= A + B\end{aligned}$$

2.5.2 Converting a Truth Table to a Minimum SOP or POS Expression

Deriving a minimum SOP or POS expression from a truth table is as straightforward as deriving an SOP or POS expression and then using the rules of Boolean algebra to obtain a minimum form. Uniting, elimination, and consensus are most commonly used. Start by finding uniting terms. Then look for elimination steps. Continue to look for more uniting and elimination steps until you cannot use those steps any more. If there are three or more terms, look for and eliminate any consensus terms.

We will use the truth table given in table 2.4 to derive a minimum SOP expression, starting by finding the SOP expression and reducing it down.

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

Table 2.4: Example truth table for deriving a minimum SOP expression.

The Boolean algebra steps are shown below.

$$\begin{aligned} F &= A'B'C' + A'B'C + A'BC + AB'C' + AB'C \\ &= A'B' + A'BC + AB' \text{ (uniting)} \\ &= A'(B' + BC) + AB' \\ &= A'(B' + C) + AB' \text{ (elimination)} \\ &= A'B' + A'C + AB' \\ &= B' + A'C \text{ (uniting)} \end{aligned}$$

A similar process is used for finding a minimum POS expression. The truth table given in table 2.5 will be used as an example.

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

Table 2.5: Example truth table for deriving a minimum POS expression.

The Boolean algebra steps are shown below.

$$\begin{aligned} F &= (A + B + C)(A + B + C')(A' + B + C)(A' + B + C')(A' + B' + C) \\ &= (A + B)(A' + B)(A' + B' + C) \text{ (uniting)} \\ &= (B)(A' + B' + C) \text{ (uniting)} \\ &= (B)(A' + C) \text{ (elimination)} \end{aligned}$$

2.5.3 Checking for Consensus Terms

Consensus is generally the most difficult Boolean algebra rule to use. However, it is very important in ensuring that an expression is minimized. Consensus terms are terms in a Boolean expression that do not add anything useful to the expression. They are completely redundant. (We will see in a later chapter that they somtimes have a use, but not for now.)

The best way that I have found to teach students how to check for consensus terms is to derive a consensus table. In a consensus table, every variable will be noted along with its complement. In an SOP expression, any terms that contain the variable or its complement will be compared and AND'd together. The result of that AND operation will determine what the consensus terms, if any, are. As an example, let's consider $F = A'B'C + A'CD + A'BD + BC'D + ABC'$, which cannot be reduced using uniting, elimination, or absorption steps. There are four terms in the expression, so we will need four rows in our consensus table, shown in table 2.6.

	Variable	Complement	Consensus Term(s)
AA'	BC'	BD, CD, B'C	BC'D, 0, 0
BB'	A'D, AC'	A'C	A'CD
CC'	A'B'	AB	0 (none)
DD'	A'B	0 (none)	0 (none)

Table 2.6: An example of a consensus table for an SOP expression.

In the first row of the consensus table, we identify the terms that contain an A and the terms that contain an A'. The terms that contain an A are ABC', we factor out the A and obtain BC' and stick it in the first cell of the table. We then identify the terms that contain an A'. These terms are $A'B'C$, $A'CD$, and $A'BD$. The A' is factored of each, and these terms are placed in the second cell of the table. We then AND together every combination of these terms.

$$(BC')(BD) = BC'D$$
$$(BC')(CD) = 0$$
$$(BC')(B'C) = 0$$

In this AND process, we have identified a consensus term: $BC'D$. We remove it from our expression and do not have to include it in the next rows of our table. Our newly simplified expression is $F = A'B'C + A'CD + A'BD + ABC'$. We are not done checking for consensus terms until the table is complete.

In the next row, we find terms $A'D$ and AC' as being associated with B. We find $A'C$ as being associated with B'. We then AND all of these combinations together and identify a second consensus term: $A'CD$.

$$(A'D)(A'C) = A'CD$$
$$(AC')(A'C) = 0$$

Our newly simplified expression is $F = A'B'C + A'BD + ABC'$. We are still not finished checking for consensus terms, however.

In the next row, we find term $A'B'$ associated with C, and AB associated with C'. ANDing these two terms together we obtain no consensus terms. In the final row, we see that there are no terms associated with D', and therefore there are no consensus terms created by that variable.

Our final minimum SOP expression is therefore $F = A'B'C + A'BD + ABC'$.

Let's do one more example with a consensus check for a POS expression. The process is much the same, but now ORing will occur rather than ANDing. Our initial expression is $F = (A'+C+D)(A'+B'+C)(B'+D')$. Because it has at least three terms, it is possible for there to be consensus terms in the expression. There are four variables, so there will be four rows in the consensus table, shown in table 2.7.

	Variable	**Complement**	**Consensus Term(s)**
AA'	1 (none)	C+D, B'+C	1 (none)
BB'	1 (none)	A'+C, D'	1 (none)
CC'	A'+D, A'+B'	1 (none)	1 (none)
DD'	A'+C	B'	A'+B'+C

Table 2.7: An example of a consensus table for a POS expression.

There are no terms in the expression that contain an A, therefore A does not contribute any consensus terms. This is similar to what we find with the variable B and with C'. However, both D and D' have terms associated with them. When we OR them together we identify $(A' + B' + C)$ as a consensus term. Our minimum POS expression is therefore $F = (A' + C + D)(B' + D')$.

2.6 Converting POS to SOP

At times, it is advantageous to convert an expression from POS to SOP, or vice versa. There are two options: one option is to create a truth table and re-derive the expression in the other format. The second option is to use a conversion method. Sometimes conversion is much simpler than creating a truth table, especially if there are more than four variables in the expression!

Converting from POS to SOP is by far the simpler direction to go. (We will cover SOP to POS in the next section.) Conversion from POS to SOP takes place using the following steps.

1. Ensure that the POS expression is minimized. If not, minimize it.
2. Look for second distributive terms. If possible, do as many second distributive steps as you can.
3. "Multiply" out all of the terms using the distributive law.
4. Simplify to minimum SOP using uniting, absorption, elimination, and consensus.

As an example, convert $F = (A' + C)(B + C')(C' + D)(B' + C + D')$ to SOP. First, verify that it is minimized. For practice, you should create a consensus table and convince yourself that there are no consensus terms.

The next step is to look for second distributive law steps. The second two terms can be combined with this law, as can the first and last terms. This reduces the number of "multiplication" steps that we have to do, and thus reduces our chances for making a mistake. Then we can multiply out the terms and reduce.

$$\begin{aligned} F &= (C' + BD)(C + (A')(B' + D')) \\ &= (C' + BD)(C + A'B' + A'D') \\ &= A'B'C' + A'C'D' + BCD \end{aligned}$$

Again, a consensus table should be created to relieve all doubts that there are no consensus terms. Once you have convinced yourself of that, you have found the minimum SOP expression.

Note how long the conversion process takes if you do not use second distributive law steps.

$$\begin{aligned} F &= (A' + C)(B + C')(C' + D)(B' + C + D') \\ &= (A'B + A'C' + BC)(C' + D)(B' + C + D') \\ &= (A'BC' + A'BD + A'C' + A'C'D + BCD)(B' + C + D') \\ &= (A'BD + A'C' + BCD)(B' + C + D') \text{ (absorption)} \\ &= A'BCD + A'B'C' + A'C'D' + BCD \\ &= A'B'C' + A'C'D' + BCD \text{ (absorption)} \end{aligned}$$

2.7 Converting SOP to POS

By far the more difficult direction is to convert SOP to POS. As always, you have the option to use a truth table rather than doing a conversion. But that is not always going to save time. The process for converting SOP to POS follows.

1. Ensure that the SOP expression is minimized. If not, minimize it.
2. Apply DeMorgan's law to obtain F'.
3. Reduce F' to a minimum SOP form.
4. Apply DeMorgan's law to obtain F.

As an example, let's convert $F = A'C' + C'D' + ACD + AB'C$ to POS. There is a consensus term ($AB'D'$), but it is not in our expression, so our expression meets the requirement in step one to find a

minimized SOP expression. We then use DeMorgan's law to find F' and reduce it to a minimum SOP form.

$$\begin{aligned} F' &= (A'C' + C'D' + ACD + AB'C)' \\ &= (A'C')'(C'D')'(ACD)'(AB'C)' \\ &= (A + C)(C + D)(A' + C' + D')(A' + B + C') \\ &= (C + AD)(A' + C' + BD') \\ &= A'C + BCD' + AC'D \end{aligned}$$

Now that we have found F' as a minimum SOP expression, we simply do one more DeMorgan's step to find F as a minimum POS expression.

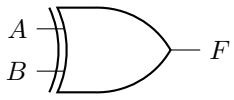
$$\begin{aligned} F &= (A'C + BCD' + AC'D)' \\ &= (A'C)'(BCD')'(AC'D)' \\ &= (A + C')(B' + C' + D)(A' + C + D') \end{aligned}$$

2.8 Exclusive OR and Equivalence

There are two more logic gates that will be introduced in this section. Note that these logic gates are not part of the SOP or POS family, but are still well-defined and often-used logic functions.

2.8.1 Exclusive OR (XOR)

A two-input exclusive OR (XOR) gate gives an output of one only when the two inputs are different from each other. This can be represented, as with all logic functions, as a circuit diagram, a truth table, and an expression ($F = A \oplus B$). The circuit diagram and truth tables are shown below in Figure 2.18.



A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.18: XOR portrayed as a circuit diagram and a truth table.

As mentioned, XOR is not a formal sum of products or product of sums expression. However it can be a convenient tool to use if you are not being restrained by the rigidity of the SOP and POS systems. It is, however, possible to obtain SOP or POS expressions by analyzing the truth table and using the methodology

explained earlier in this chapter.

$$F_{SOP} = AB' + A'B$$
$$F_{POS} = (A + B)(A' + B')$$

There are Boolean rules regarding the use of XOR expressions. They are not necessarily helpful in terms of minimizing expressions but in understanding how XOR works, especially if you need to deal with an XOR operation with a zero or one, or inverting an XOR gate, or figuring out how a 3-input XOR gate would work.

Operations with 0 and 1:

$$X \oplus 0 = X$$
$$X \oplus 1 = X'$$
$$X \oplus X = 0$$
$$X \oplus X' = 1$$

Distributive law:

$$X(Y \oplus Z) = XY \oplus XZ$$

DeMorgan's law:

$$(X \oplus Y)' = X \oplus Y' = X' \oplus Y = XY + X'Y'$$

Commutative law:

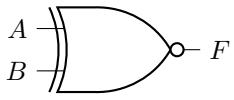
$$X \oplus Y = Y \oplus X$$

Associative law:

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z$$

2.8.2 Equivalence (XNOR)

A two-input equivalence (XNOR) gate gives an output of one when the two inputs are the same. The circuit diagram and truth table for the XNOR operation are shown in Figure 2.19, and the expression is $F = A \equiv B$. The $\equiv$ sign is called "equivalence."



A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

Figure 2.19: XNOR portrayed as a circuit diagram and a truth table.

Note the bubble on the output of what otherwise looks like an XOR gate. This implies that an XNOR is the complement of an XOR. XOR and XNOR are simply complements of each other.

$$(A \oplus B)' = A \equiv B$$

XNOR is also not a formal SOP or POS expression. However, we can use the truth table to derive SOP

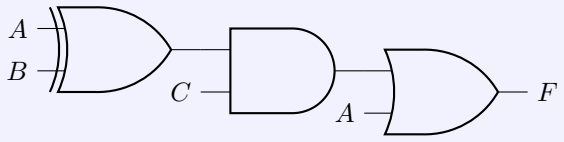
and POS expressions for XNOR.

$$F_{SOP} = AB + A'B'$$
$$F_{POS} = (A' + B)(A + B')$$

2.8.3 Simplifying XOR and XNOR Expressions

At times, we may want to simplify an expression that contains an XOR or XNOR operation. Or we may want to find the expression in minimum SOP or POS form. As with all things, we always have options in how we choose to do this. We can opt to create a truth table and derive a minimum expression from that. Or we can find a Boolean expression and use the rules of Boolean algebra, as well as the SOP/POS formats of XOR/XNOR, to derive a minimum expression.

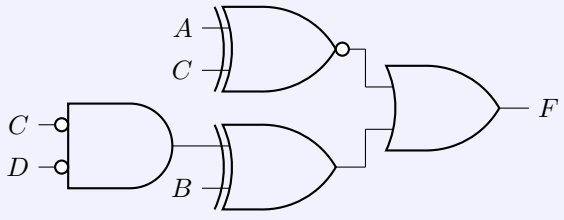
Example: Using a circuit diagram to simplify XOR/XNOR



The expression, which contains an XOR, will be reduced to minimum SOP by using the relationship $X \oplus Y = X'Y + XY'$ that we derived previously in this section.

$$\begin{aligned} F &= (A \oplus B)C + A \\ &= (AB' + A'B)C + A \\ &= AB'C + A'BC + A \\ &= A + A'BC \\ &= A + BC \end{aligned}$$

Example: Simplifying a more complicated XOR/XNOR circuit



The circuit has both an XOR and an XNOR gate. Using the logical relationships we can find a minimum SOP expression.

$$\begin{aligned} F &= (B \oplus C'D') + (A \equiv C) \\ &= (B'C'D' + B(C'D')') + (AC + A'C') \\ &= B'C'D' + B(C + D) + AC + A'C' \\ &= B'C'D' + BC + BD + AC + A'C' \end{aligned}$$

2.9 Building Circuits

Circuit diagrams, Boolean expressions, and truth tables define our circuits. To build circuits, we need to use actual, physical logic devices and connect them together in a meaningful way. Digital logic chips are connected together on a breadboard or printed circuit board (PCB).

2.9.1 TTL Logic

The types of logic chips featured in this book are part of the 7400 series of integrated circuits that was initially developed and manufactured by Texas Instruments. These devices are all compatible with each other and are based on TTL (transistor-transistor logic) that uses a particular type of transistor known as a bipolar junction transistor (BJT). The basic physics behind the BJT will be discussed in chapter 8. A second type of digital logic using CMOS architecture is also available. The exact voltages used for a HIGH and LOW signal are not generally compatible between the two types of devices.

TTL chips have particular advantages and disadvantages compared to CMOS chips, which are explained below.

- TTL chips are generally less expensive than CMOS chips.
- TTL chips are less susceptible to damage from electrostatic discharge than CMOS chips.
- TTL chips consume more power than CMOS chips.
- A LOW signal on a TTL input is registered when the voltage is between 0 V and 0.8 V.
- A HIGH signal on a TTL input is registered when the voltage is between 2 V and 5 V.

Because TTL chips, and the 7400 series, is so standardized, many companies manufacture chips that are virtually identical. As long as the part number is the same, any TTL logic chip can be replaced by one from another manufacturer with identical functionality of the circuit.

2.9.2 7400 Series

The 7400 series has to do with the numbering system for the digital logic chips. The chips are all numbered 74–××.

The 74 prefix indicates that the devices are part of the TTL series of logic devices. If the prefix is 54, it means that the device is military grade (which usually indicates that it is capable of operating at larger extremes of temperature). There may be letters in front of the 74, which just indicates the manufacturer, and can be ignored.

The last two, three, or four numbers (indicated by ××in the 74–××) identifies the exact part. A 7408 chip contains four AND gates, whereas a 7432 chip contains four OR gates. At times there may be a final letter stamped after the part number. This indicates the type of packaging used. (Common is N for DIP architecture, which will be explained below.)

There may be an infix of letters between the 74 and the part number ××. These letters indicate the exact architecture of the transistors that are built inside the chip. Two common infixes are listed below.

- LS – low-power Shottkey, these chips consume less power than chips that do not have the LS infix.
- HCT – high-speed CMOS / TTL compatable, these chips are able to interface between both types of logic families: TTL and CMOS.

2.9.3 Input and Output Types

The 7400 series of digital logic chips features some chips with special types of inputs and outputs.

Open-collector outputs are a particular output type where the output of the digital signal has no internal connection to Vcc. On its own, the signal output of the chip cannot be driven high without an external connection to high voltage through an external pull-up resistor. This can be connected to a signal that differs from the internal logic signal for Vcc, enabling connections to voltages higher than 5 V. This was particularly useful when output devices such as Nixie tubes and fluorescent displays were common and required connections to 50 V, 100 V, or more. Those voltages would damage the logic gate but were required for the operation of such displays. Open-collector outputs can also be used to interface chips of different logic families together so that they will work properly.

A schematic of an open-collector output is shown in Figure 2.20. On the left is a representation of the output circuitry of an open-collector chip. On the right is a schematic of how that could be connected to a power supply using an external pull-up resistor.

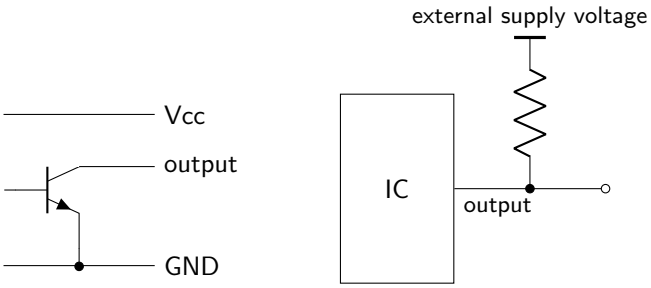


Figure 2.20: Open-collector output schematic (left) and connection to a supply voltage using an external pull-up resistor (right).

Tri-state outputs are a type of output signal that is capable of taking on one of three output values: logic HIGH, logic LOW, and what's known as a high-Z state (also known as a tri-state or third state). This tri-state is an electrical disconnection of the output signal. It is very useful when working with hardware where multiple chips need to be connected to a single output signal. They are discussed in more detail in section 10.1.

Schmitt trigger inputs are inputs that connect to a device known as a Schmitt trigger. A Schmitt trigger is a device that uses hysteresis. That is to say, a certain threshold of voltage must be reached before the output signal will change from HIGH to LOW or vice versa. Hysteresis is used to remove noise from a signal. Therefore, a Schmitt trigger device is used when there may be noise present in an input signal that should be filtered out. These devices are also useful when creating a switch debounce circuit. Logic chips that feature Schmitt trigger inputs have a circuit symbol similar to that shown in Figure 2.21, where the logic gate is overlaid with a symbol representing the hysteresis property.

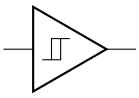


Figure 2.21: Circuit symbol of a buffer featuring a Schmitt trigger input.

2.9.4 Dual In-Line Package (DIP)

The dual in-line package (DIP) architecture is used in all of the devices that will be used as examples in this book. It is convenient to use because it fits into a standard breadboard without any need for soldering. A DIP chip has two (dual) sets of pins set in parallel (in-line) columns, as illustrated in Figure 2.22. Each of the pins in a DIP chip is unique, so it is important to ensure that the pins on each side of the chip are electrically isolated. This is accomplished by straddling the DIP chips across the trench that is featured in the center of a breadboard.

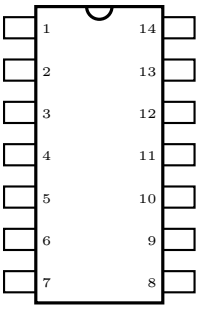


Figure 2.22: Illustration of a DIP chip with fourteen pins.

Each chip features a notch or a dot at the top. This indicates which way is "up" and helps to identify each of the pins on the logic chip. A datasheet or a pinout diagram is used to determine how to wire each of the chips to successfully connect them in a circuit.

2.9.5 DIP Switches

DIP switches are used to control several inputs without the need to move wires around on a breadboard, and also prevents the inconvenience of having to use multiple toggle switches or pushbuttons. It is simply a convenient package that contains multiple switches that can be turned on and off. DIP switches require connections to power and ground, and use a pull-down resistor to prevent short circuits.

2.9.6 Output Devices

Most digital logic devices have some type of output. Perhaps we just want to turn an LED on and off under different conditions. Or maybe we are interested in displaying different numbers on a display. For this reason, we need to connect the outputs of our circuits to an LED or a 7-segment display. (There are other outputs that can be used, such as LCD screens, but they are quite difficult to use without a microcontroller.)

LED stands for light-emitting diode, which is a semiconductor that emits light when current travels through in a particular direction (from anode to cathode). If the anode of an LED is connected to the output of a circuit, and the cathode is connected to ground, then when the output of the circuit is HIGH, current will flow and the LED will illuminate. When the output of the circuit is LOW, current will not flow and the LED will remain off. A current-limiting resistor is used to prevent an exess of current from damaging the LED. (They can and will melt if too much current passes through, causing them to heat up.)

A 7-segment display contains several LEDs inside of the package and is able to illuminate decimal numbers from zero through nine. A 7-segment display is depicted in Figure 2.23.

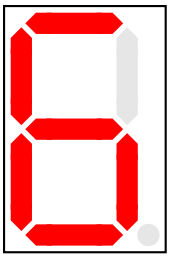


Figure 2.23: Illustration of a 7-segment display.

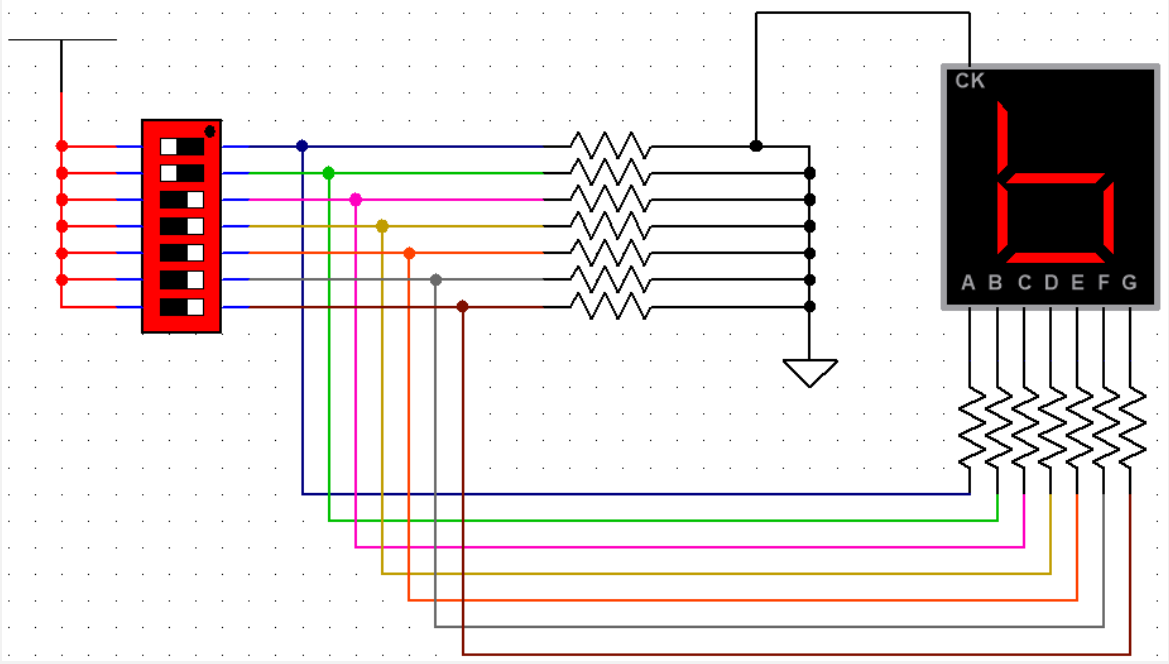
7-segment displays will play a role in several of our discussions in this book. They are particularly useful devices because they represent digital logic outputs in a number system that humans are familiar with: base 10. They are commonly used in clocks and consumer appliances such as microwaves, ovens, and DVD players.

Circuit Project: 7-Segment Display

A circuit project that you can try to become familiar with the 7-segment display, breadboards, and DIP switches is to create a circuit where each segment of the display is toggled on and off by the DIP switch. You will need:

- 1– Breadboard
- 1– 5 V power supply or battery pack
- 1– DIP switch
- 14– 220 Ω resistors
- 1– 7-segment display

The display used for the schematic included here is a common cathode display which means that the common pin gets connected to ground and a voltage of 5 V will illuminate each segment. (A common anode display will work in the opposite manner; 5 V is connected to the common pin, and 0 V applied to a segment will cause it to illuminate.)



2.10 Example Problems

Circuit Diagrams, Truth Tables, and Boolean Expressions

1. Draw $F = A'B + A'C' + AB'C$ as a circuit diagram.
2. Determine if F and G are equivalent. Explain your reasoning. $F = AC + A'C' + BC + A'B$. $G = (A' + B + C)(A' + B' + C)(A + B + C')$
3. Determine if F and G are equivalent. Explain your reasoning. $F = A \oplus BC$. $G = A'BC + AB' + AC'$
4. Create a truth table for $F = B(A' + C) + AC$.
5. Express the circuit diagram given in Figure 2.24 as a Boolean expression.

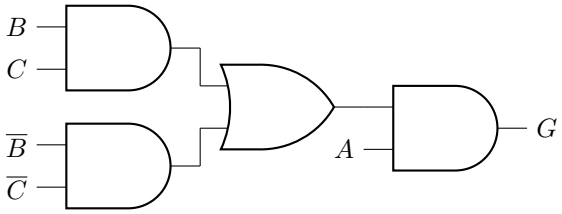


Figure 2.24: Circuit to be expressed as a Boolean expression.

Minimum Sum of Products Expressions

1. Express the truth table given in table 2.8 as a minimum SOP expression.

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

Table 2.8: Truth table to be expressed as minimum SOP.

2. Convert $F = (C + D)(C' + D')(A' + B + D')$ to a minimum SOP expression.
3. Convert $F = (B' + C' + D)(A' + B + C')(C + D)(A + B + C)$ to a minimum SOP expression.
4. Convert $F = (A + B + C)(B' + C + D)(A' + B + D')(A' + C' + D)$ to a minimum SOP expression.
5. Express $F = A'BC'D' + A'BC'D + ABC'D$ as a minimum SOP expression.

Minimum Product of Sums Expressions

1. Express the truth table given in table 2.9 as a minimum POS expression.

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

Table 2.9: Truth table to be expressed as minimum POS.

2. Convert $F = BC' + A'B' + CD' + AD$ to a minimum POS expression.

3. Convert $F = BC + C'D' + A'B'C'$ to a minimum POS expression.

4. Express the circuit diagram given in Figure 2.25 as a minimum POS expression.

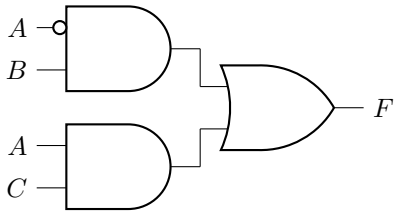


Figure 2.25: Circuit to be expressed as minimum POS.

5. Express $F = AB'D + A'BC' + A'C'E' + AB'C$ as a minimum POS expression.

XOR and XNOR Circuits

1. Express the circuit diagram given in Figure 2.26 as a minimum POS expression.

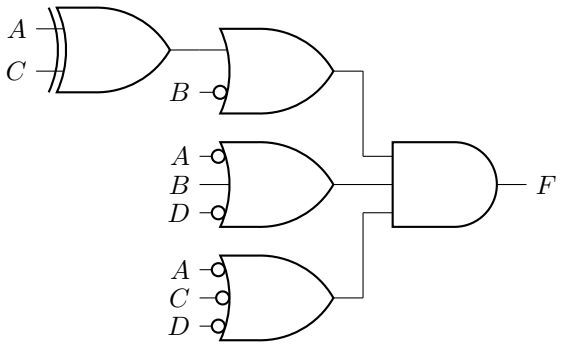


Figure 2.26: XOR circuit to be expressed as minimum POS.

2. Create a truth table corresponding to the circuit diagram given in Figure 2.27.

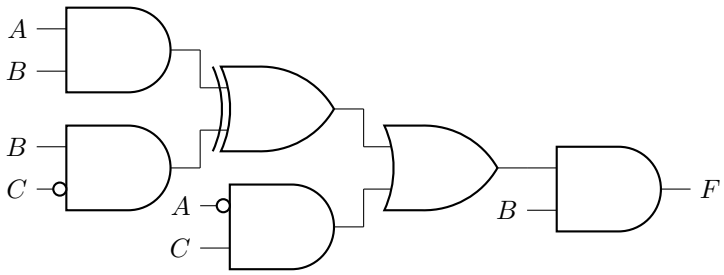


Figure 2.27: XOR circuit to be expressed as a truth table.

3. Express F as a minimum POS expression. $F = (A \equiv B')(CD \oplus B') + ABCD$

4. Express F as a minimum SOP expression. $F = (A \equiv D)(CD \oplus B) + ABCD$

5. Express F as a minimum POS expression. $F = (W \equiv Z)(YZ \oplus X)$

3 Applications of Boolean Algebra

Now that we understand Boolean algebra expressions, we can use them to realize useful logic functions. There are three types of problems we'll learn to solve. The first involves open-ended design. The second involves translating written parameters to Boolean algebra. The third involves using a truth table to implement a problem with numeric constraints.

The first step in all cases is to define all of the variables involved in the process. What are the output variables? Give them names, and define what a zero and a one correspond to. What are all of the input variables? Give them names, and define what a zero and a one correspond to.

3.1 Open-Ended Design

In open-ended design, you are asked to design something. The exact parameters are up to you. Think about all of the possible inputs and outputs that should play a role in the project. Define them clearly and include exactly what a zero and a one should mean. Because Boolean values of zero and one imply logical opposites, the definitions of the zero and the one should be logical opposites of each other. For example: the logical opposite of hot is not cold; the logical opposite of hot is not hot.

Example: Car alarm

Let's say you're tasked with designing a car alarm. There are so many different ways that you could go about designing this. I am only going to provide you with one possible interpretation. The output of the car alarm is an alarm. Let's call it A. A value of 0 means the alarm will not go off. A value of 1 means the alarm will go off. There could be many different inputs. Here are a few possibilities.

- Force (F) – 0 means that not a lot of force was applied to the car; 1 means that a lot of force was applied.
- Key (K) – 0 means that the key was not used to unlock the doors; 1 means that the key was used to unlock the doors.
- Passenger (P) – 0 means that there is nobody in the car; 1 means that there is somebody in the car.
- Window (W) – 0 means that the window is closed; 1 means that the window is open.
- Key Fob Button (B) – 0 means that the alarm button on the key fob was not pressed; 1 means that the alarm button on the key fob was pressed.

Again, while there are many possible interpretations of this design problem, one possible Boolean

expression could be what follows.

$$A = F + K'WP' + B$$

In other words, the alarm will go off if too much force is applied to the car, or if the key is not used to unlock the door while nobody is inside and the window is open, or if the alarm button is pressed on the key fob.

3.2 Written or Verbal Parameters

In these types of questions, you are given written or verbal parameters. Read them carefully, paying attention to words such as **and**, **or** and **not**. The same procedure still applies. You must consider all of the input and output variables and what zero and one correspond to for each variable.

Example: Laughing at a joke

You should laugh at a joke if it is funny and not offensive to others, or if it was told by your professor (regardless of if it was funny, but only if it's not offensive to others). The output variable here is whether or not your should laugh. We'll call it L. When it is 0, you should not laugh. When it is 1, it means that you should laugh. The input values are given below.

- Funny (F) – 0 means that the joke was not funny; 1 means that the joke was funny.
- Offensive (O) – 0 means that the joke was not offensive; 1 means that the joke was offensive.
- Professor (P) – 0 means that the joke was told by somebody who is not your professor; 1 means that the joke was told by your professor.

The Boolean expression in this case is given below.

$$L = FO' + PO'$$

3.3 Numeric Parameters

In these types of design problems, you are given numeric parameters. The different values of the numbers result in different output values, depending on how the question is formatted. In this case, the input is usually a binary number (signed or unsigned, depending on how the question is asked). The output, however, still needs to be defined as far as what a zero and one means.

Example: Refrigerator warning

The temperature sensor on a refrigerator gives a 4-bit 2's complement number as a value corresponding to the temperature in degrees Celsius. If the temperature is greater than or equal to 3 °C, then an overheat warning light needs to go on. If the temperature is less than −2 °C, then a freeze warning light needs to go on. First, both of the output variables need to be defined.

- Overheat Warning (OW) – 0 means that the overheat warning will not go on; 1 means that the overheat warning will go on.
- Freeze Warning (FW) – 0 means that the freeze warning will not go on; 1 means that the freeze warning will go on.

The outputs are best considered by using a truth table. It is no longer sufficient to "reason through" to come up with a Boolean expression. The truth table is given below.

T (°C)	A	B	C	D	OW	FW
0	0	0	0	0	0	0
1	0	0	0	1	0	0
2	0	0	1	0	0	0
3	0	0	1	1	1	0
4	0	1	0	0	1	0
5	0	1	0	1	1	0
6	0	1	1	0	1	0
7	0	1	1	1	1	0
−8	1	0	0	0	0	1
−7	1	0	0	1	0	1
−6	1	0	1	0	0	1
−5	1	0	1	1	0	1
−4	1	1	0	0	0	1
−3	1	1	0	1	0	1
−2	1	1	1	0	0	0
−1	1	1	1	1	0	0

You may use SOP or POS to implement the function. Minimum SOP expressions for each OW and FW are given below.

$$OW = A'CD + A'B$$

$$FW = AB' + AC'$$

3.4 Binary Adders

Binary adders are a particularly useful application of Boolean algebra. They form the basis for computer arithmetic and logic units. While adders can be built that can sum together very large numbers, the building block of an adder is quite simple. There are two types of adders: full adders and half adders.

3.4.1 Half Adder

A half adder takes two binary input bits (A and B) and outputs a sum (S) and carry out (C_{OUT}) term. The truth table for a half adder is given in table 1.5 in chapter 1. We can use this truth table to derive Boolean algebra expressions for both the sum and the carry out terms.

$$S = A \oplus B$$
$$C_{OUT} = AB$$

We can therefore build a half adder circuit using an XOR gate and an AND gate. This circuit is shown in Figure 3.1.

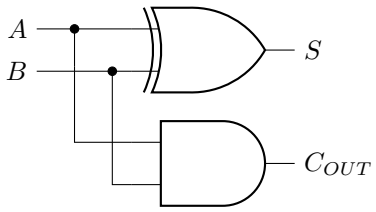


Figure 3.1: Half adder circuit diagram.

3.4.2 Full Adder

As we saw in the chapter about addition, a half adder is not sufficient when we need to add binary numbers that contain more than one bit. What if there's a carry from a previous column? In that case, we need to consider a full adder which includes a third input variable, C_{IN}. The full adder truth table is table 1.6 in chapter 1. Now that we understand Boolean algebra, we can derive an expression for the sum (S).

$$\begin{aligned} S &= A'B'C_{IN} + A'BC'_{IN} + AB'C'_{IN} + ABC_{IN} \\ &= A'(B'C_{IN} + BC'_{IN}) + A(B'C'_{IN} + BC_{IN}) \\ &= A'(B \oplus C_{IN}) + A(B \equiv C_{IN}) \\ &= A'(B \oplus C_{IN}) + A(B \oplus C_{IN})' \\ &= A \oplus B \oplus C_{IN} \end{aligned}$$

We can also derive an expression for the carry out (C_{OUT}).

$$\begin{aligned} C_{OUT} &= A'BC_{IN} + AB'C_{IN} + ABC'_{IN} + ABC_{IN} \\ &= A'BC_{IN} + AB'C_{IN} + AB \\ &= A'BC_{IN} + A(B'C_{IN} + B) \\ &= A'BC_{IN} + A(C_{IN} + B) \\ &= A'BC_{IN} + AC_{IN} + AB \\ &= C_{IN}(A'B + A) + AB \\ &= C_{IN}(B + A) + AB \\ &= BC_{IN} + AC_{IN} + AB \end{aligned}$$

We can now build a full adder using two XOR gates, three AND gates and an OR gate. This circuit diagram is given in Figure 3.2.

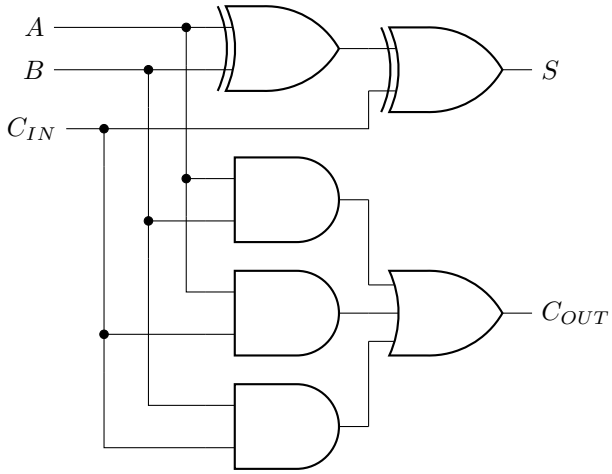


Figure 3.2: Circuit diagram for a full adder.

3.4.3 Ripple Carry Adder

What happens when we want to add binary numbers that contain more than one bit? We cascade 1-bit adders together to create an adder that can add together binary numbers with more than one bit. This type of adder is called a ripple carry adder, as the carry bit ripples from one bit to another. It performs addition much the same way that we do when we use pencil and paper. Start at the LSB and continue with the addition steps until the MSB has been reached. A 4-bit ripple carry adder is depicted schematically in Figure 3.3. The adder performs the arithmetic shown in table 3.1 is performed out.

A serious drawback to ripple carry adders is the time it takes to add numbers together. Each bit adds one at a time (just as we do with pencil and paper), and the second bit cannot be added until the first is

					C_{IN}
	+	A_3	A_2	A_1	A_0
	+	B_3	B_2	B_1	B_0
=	C_4	S_3	S_2	S_1	S_0

Table 3.1: Addition performed in a ripple carry adder.

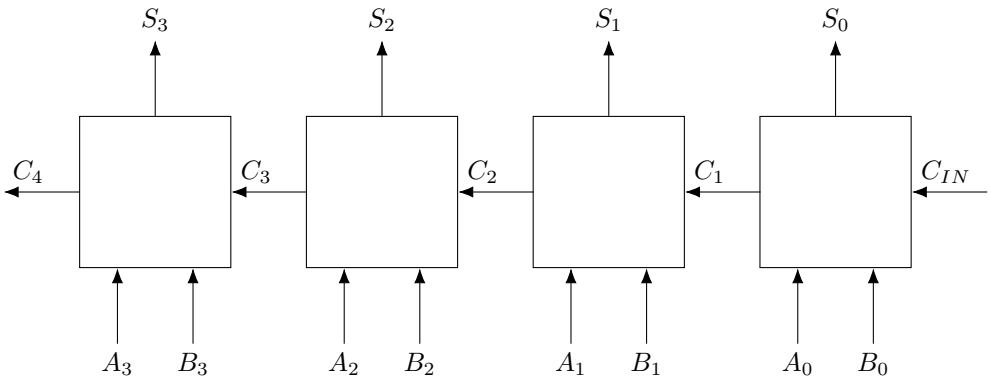


Figure 3.3: 4-bit ripple carry adder.

completed. The third bit cannot be added until the second is completed, and so on. Therefore, the more bits that are being added, the longer it takes to obtain a completed sum. This long time delay occurs because the carry bits need to ripple from one bit to another. More complicated logic circuits have been created to sum numbers together more quickly at the expense of taking up more space on a chip.

3.5 Parity Checking

Another useful application of Boolean algebra is in parity checking. Parity refers to the number of ones in a variable. If there are an even number of ones, then the variable is said to have even parity. If there are an odd number of ones, then the variable is said to have odd parity. For example, $A'B'C$ corresponds to 001, which has a single one, and therefore odd parity.

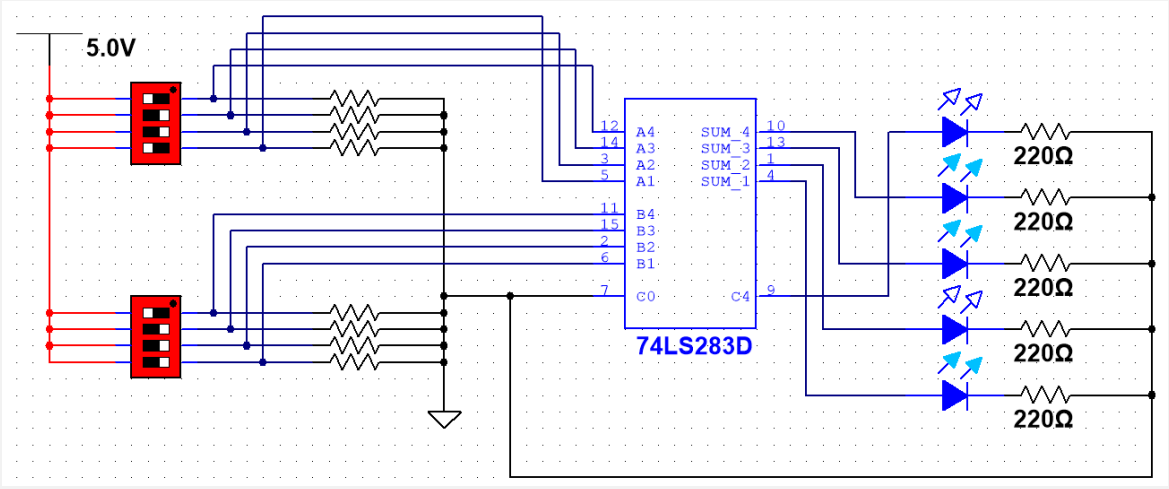
Parity checking is a simple way to check for errors in transmitting a signal from point A to point B. A message of a certain bit length is generated. Before it is sent, a parity bit is added to either the beginning or the end of the message. Both the transmitting and receiving parties have agreed on which parity (odd or even) will be expected beforehand.

Let's say for that you and a friend have decided on a 3-bit messaging system, shown in table 3.2.

Circuit Project: 4-Bit Unsigned Adder

This circuit project uses an adder chip to add two unsigned 4-bit binary numbers together, and display the 5-bit binary result on LEDs. Later in this book we'll learn how to use something called a decoder to turn the result into a decimal number to display onto a 7-segment display.

- 1– Breadboard
- 1– 5 V power supply or battery pack
- 1– DIP switch (8 switches, or 2 DIP switches with 4 switches each)
- 13– 220 Ω resistors
- 1– 74LS283 4-bit adder chip
- 5– LEDs



A	B	C	Message
0	0	0	Come over at 7 tonight
0	0	1	Don't come over tonight
0	1	0	Bring nachos when you visit
0	1	1	How are you doing?
1	0	0	I'm doing great
1	0	1	I am not doing great
1	1	0	Have you received my message?
1	1	1	I have received your message

Table 3.2: Truth table corresponding to your 3-bit messaging system.

Before you send your message, you want to include a parity bit at the end of the message. This would include a fourth bit to be tacked on to the end of your message. Depending on if you choose even or odd parity, you will need different hardware to do this. The parity bits that you would need to use in each case are given in table 3.3.

A	B	C	Even	Odd
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

Table 3.3: Truth table corresponding to the parity bits needed for your 3-bit messaging system.

3.5.1 Parity Generation

If you look carefully, you may notice the XOR configuration related to your even parity bit. To generate an even parity bit, you would therefore need to wire up the hardware of $PB_{EVEN} = A \oplus B \oplus C$. If you want an odd parity bit, you would need to wire up the hardware using XNOR. The expression would be $PB_{ODD} = A \equiv B \equiv C$.

Let's say that you and your friend have decided on even parity. You wish to send the message "bring nachos when you visit." The full signal that you send is therefore 0101 (the first three bits contain the message, and the parity bit is at the end of the message).

3.5.2 Parity Detection

If your friend successfully receives the transmission, then they will test the parity. To test for even parity, XNOR will be used. To test for odd parity, XOR will be used. Because you decided on even parity, your friend will send your signal through XNOR hardware:

$$\begin{aligned} A \equiv B \equiv C \equiv D &= 0 \equiv 1 \equiv 0 \equiv 1 \\ &= 0 \equiv 0 \\ &= 1 \end{aligned}$$

This one indicates that the received message has even parity. Your friend then inspects the first three bits of the message and buys nachos.

Let's assume that there was electrical noise introduced somewhere in the transmission process. Instead of receiving 0101, your friend received the message 0111. When your friend uses their hardware to check the

parity of the message, they will use their XNOR hardware:

$$\begin{aligned} A \equiv B \equiv C \equiv D &= 0 \equiv 1 \equiv 1 \equiv 1 \\ &= 0 \equiv 1 \\ &= 0 \end{aligned}$$

This zero indicates that the message had odd parity. Your friend immediately knows that there was an issue with the message transmission process, and sends you a text message instead to see what you want.

3.5.3 Benefits and Drawbacks to Parity Checking

Parity checking is a very simple way to check for transmission errors. It only requires one extra bit added to your message, and requires minimal hardware on the generation and detection ends. It is therefore a "cheap and easy" way to check for transmission errors.

However, parity checking has a serious drawback. If an odd number of bits are flipped in the transmission process (due to electrical noise or other reasons), then the incorrect transmission will be indicated. However, if an even number of bits are flipped in the transmission process, the detection will generate a false positive transmission. Therefore, parity checking is capable of creating false positive situations (although it is incapable of creating a false negative situation).

3.6 Incompletely Specified Circuits

At times, it is possible that there are inputs (or combinations of inputs) to a circuit that will never show up under any circumstance. In that case, we do not care what the output should be in those cases. The output could be a zero or it could be a one without impacting the functionality of the circuit. We call these outputs don't cares and they are represented on a truth table with an $\times$.

Finding a minimized expression when there are don't care terms requires a slightly different protocol.

1. Use all of the zero terms to find a minimum POS expression **or** use all of the one terms to find a minimum SOP expression. Do not consider the don't care terms.
2. Inspect each of the don't care terms individually and see if it will help to minimize the expression further by removing a variable from a term, or removing a term. If so, include it in your expression. If not, do not include it in your expression.

Do not just include the don't care terms without considering whether or not they will help to reduce the expression. If you include them and they don't help to minimize the expression, then you no longer have a minimum SOP or POS expression!

We will see that incompletely specified circuits result frequently from using 4-bit binary to represent BCD characters on 7-segment displays. There are sixteen possible binary values that can be represented in

four bits (zero through fifteen), but only ten decimal values (zero through nine) will ever be displayed. The other six outputs would therefore be don't care values.

Example: Alarm clock

Consider a real-time clock. It is a chip that keeps track of the time, day of the week, month, day, and year. You wish to configure an alarm that goes off on work nights. The day of the week is given as a 3-bit binary number. There are only seven days of the week, and there are eight 3-bit binary numbers, so one of the 3-bit numbers is unused.

Day of Week	**A**	**B**	**C**	**ALRM**
–	0	0	0	×
Sun	0	0	1	1
Mon	0	1	0	1
Tue	0	1	1	1
Wed	1	0	0	1
Thu	1	0	1	1
Fri	1	1	0	0
Sat	1	1	1	0

We would like to find a minimum SOP expression. The first step is to consider only the terms that have an output of one on the truth table.

$$\begin{aligned} ALRM &= A'B'C + A'BC' + A'BC + AB'C' + AB'C \\ &= A'B'C + A'B + AB' \\ &= A'B + A'C + AB' \end{aligned}$$

This is as minimum as we can make the expression under the circumstances. We can then consider the don't care term, $A'B'C'$, and determine if it will help minimize the expression further.

$$\begin{aligned} ALRM &= A'B + A'C + AB' + A'B'C' \\ &= A'(B + C + B'C') + AB' \\ &= A'(B + C + C') + AB' \\ &= A' + AB' \\ &= A' + B' \end{aligned}$$

The don't care does indeed minimize the expression further, so we will include it in the implementation of this alarm project.

Example: Deriving SOP and POS expressions for an incompletely specified circuit

Derive both minimum SOP and minimum POS expressions from the truth table given below.

A	B	C	F
0	0	0	1
0	0	1	×
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	×
1	1	1	1

To find minimum SOP, first minimize only the rows of the truth table that have outputs of one.

$$\begin{aligned} F &= A'B'C' + A'BC + ABC \\ &= A'B'C' + BC \end{aligned}$$

Now consider each of the don't care terms individually and see if they help. The first don't care term corresponds to $A'B'C$.

$$\begin{aligned} F &= A'B'C' + BC + A'B'C \\ &= A'B' + BC \end{aligned}$$

This minimizes the expression by removing a variable. Next, consider the don't care term corresponding to ABC'.

$$\begin{aligned} F &= A'B' + BC + ABC' \\ &= A'B' + B(C + AC') \\ &= A'B' + B(C + A) \\ &= A'B' + BC + AB \end{aligned}$$

This does **not** reduce the expression, so it will not be included. $F_{SOP} = A'B' + BC$.

To find minimum POS, first consider only the rows that have an output of zero.

$$\begin{aligned} F &= (A + B' + C)(A' + B + C)(A' + B + C') \\ &= (A + B' + C)(A' + B) \end{aligned}$$

Consider the don't care term corresponding to $(A + B + C')$.

$$\begin{aligned} F &= (A + B' + C)(A' + B)(A + B + C') \\ &= (A + B' + C)[B + A'(A + C')] \\ &= (A + B' + C)(B + A'C') \\ &= (A + B' + C)(A' + B)(B + C') \end{aligned}$$

This does **not** minimize the expression, so it will not be used. Next, consider the don't care term $(A' + B' + C)$.

$$\begin{aligned} F &= (A + B' + C)(A' + B)(A' + B' + C) \\ &= (B' + C)(A' + B) \end{aligned}$$

That term reduces the expression by eliminating a variable, so it will be used. $F_{POS} = (B' + C)(A' + B)$.

3.7 Minterm and Maxterm Expressions

Minterm and maxterm expressions are simply a means to write possibly complicated Boolean expressions in one simple equation, rather than using a truth table (which can get very large when more than four input variables are used), and without using a Boolean algebra expression. They are simply a shorthand that represents another way of discussing a logic function. (We have already discussed truth tables, circuit diagrams, and Boolean algebra expressions. We can now add minterm and maxterm expressions to the mix.)

3.7.1 Minterm Expressions

A minterm is a product expression where each variable appears only once in either true or complemented form, but not both. In the last chapter, we called these product expressions, as they represent the logical AND operation.

Minterms correspond to truth table rows and we use them in sum of products (or sum of minterm) expressions. We can therefore identify the minterms that correspond to every row of a truth table, and use the ones that pertain to the expression that we are interested in. All of the minterms of a 3-variable truth table are shown in table 3.4.

Row	A	B	C	Minterm
0	0	0	0	$A'B'C' = m_0$
1	0	0	1	$A'B'C = m_1$
2	0	1	0	$A'BC' = m_2$
3	0	1	1	$A'BC = m_3$
4	1	0	0	$AB'C' = m_4$
5	1	0	1	$AB'C = m_5$
6	1	1	0	$ABC' = m_6$
7	1	1	1	$ABC = m_7$

Table 3.4: 3-input variable minterms.

This allows us to use a convenient shorthand for identifying a Boolean expression. For example, the truth table given in table 3.5 includes minterms m_1, m_2, m_3, m_5, and m_7.

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

Table 3.5: Example truth table for determining a minterm and maxterm expression.

We can represent this as the sum of those minterms.

$$F(A, B, C) = \Sigma m(1, 2, 3, 5, 7)$$

To derive a minimum SOP expression for this sum of minterms, we simply convert each minterm to its Boolean expression and use the rules of Boolean algebra to simplify.

$$\begin{aligned} F(A, B, C) &= m_1 + m_2 + m_3 + m_5 + m_7 \\ &= A'B'C + A'BC' + A'BC + AB'C + ABC \\ &= A'B'C + A'B + AC \\ &= A'(B'C + B) + AC \\ &= A'(C + B) + AC \\ &= A'B + A'C + AC \\ &= A'B + C \end{aligned}$$

Example: Creating a minimum expression from a minterm expression

Find a minimum SOP expression.

$$\begin{aligned}
F(A,B,C) &= \Sigma m(1,2,4,6) \\
&= m_1 + m_2 + m_4 + m_6 \\
&= A'B'C + A'BC' + AB'C' + ABC' \\
&= A'B'C + A'BC' + AC' \\
&= A'B'C + C'(A'B + A) \\
&= A'B'C + C'(B + A) \\
&= A'B'C + BC' + AC'
\end{aligned}$$

It is also possible to express a minterm expression for functions with more than 3 variables. In fact, minterm expressions make it much easier to define a function than a truth table, because rather than writing out 2^n rows, just one simple expression can be written.

$$\begin{aligned}
F(A,B,C,D) &= \Sigma m(3,5,7,12,13) \\
&= m_3 + m_5 + m_7 + m_12 + m_13 \\
&= A'B'CD + A'BC'D + A'BCD + ABC'D' + ABC'D \\
&= A'B'CD + A'BD + ABC' \\
&= A'D(B'C + B) + ABC' \\
&= A'D(C + B) + ABC' \\
&= A'CD + A'BD + ABC'
\end{aligned}$$

3.7.2 Maxterm Expressions

A maxterm is a sum expression where each variable appears only once in either true or complemented form, but not both. In the last chapter, we called these sum expressions, as they represent the logical OR operation.

Maxterms correspond to truth table rows and we use them in product of sums (or product of maxterm) expressions. We can therefore identify the maxterms that correspond to every row of a truth table, and use the ones that pertain to the expression that we are interested in. All of the maxterms of a 3-variable truth table are shown in table 3.6.

Row	A	B	C	Maxterm
0	0	0	0	$(A+B+C)=M_0$
1	0	0	1	$(A+B+C')=M_1$
2	0	1	0	$(A+B'+C)=M_2$
3	0	1	1	$(A+B'+C')=M_3$
4	1	0	0	$(A'+B+C)=M_4$
5	1	0	1	$(A'+B+C')=M_5$
6	1	1	0	$(A'+B'+C)=M_6$
7	1	1	1	$(A'+B'+C')=M_7$

Table 3.6: 3-input variable maxterms.

Just as with minterm expressions, we can use shorthand to identify our Boolean expression. The truth table shown earlier in table 3.5 includes maxterms M_0, M_4, and M_6. Our shorthand for identifying this Boolean expression is therefore the product of these maxterms.

$$F(A,B,C)=\Pi M(0,4,6)$$

To derive a minimum POS expression for this product of maxterms, we simply convert each maxterm to its Boolean expression and use the rules of Boolean algebra to simplify.

$$\begin{aligned}F(A,B,C)&=M_0M_4M_6\\&=(A+B+C)(A'+B+C)(A'+B'+C)\\&=(B+C)(A'+B'+C)\\&=C+B(A'+B')\\&=C+A'B\\&=(A'+C)(B+C)\end{aligned}$$

Example: Creating a minimum expression from a maxterm expression

Find a minimum POS expression.

$$\begin{aligned}F(A,B,C)&=\Pi M(0,3,5,7)\\&=M_0M_3M_5M_7\\&=(A+B+C)(A+B'+C')(A'+B+C')(A'+B'+C')\\&=(A+B+C)(A+B'+C')(A'+C')\\&=(A+B+C)[C'+A'(A+B')]\\&=(A+B+C)(C'+A'B')\\&=(A+B+C)(A'+C')(B'+C')\end{aligned}$$

Similarly with minterm expressions, we can use maxterm expressions to represent functions of more than three variables. An example with a four variable Boolean expression is given below.

$$\begin{aligned}
F(A,B,C,D) &= \Pi M(1,3,5,10,11,12,15) \\
&= M_1 M_3 M_5 M_{10} M_{11} M_{12} M_{15} \\
&= (A+B+C+D')(A+B+C'+D')(A+B'+C+D')(A'+B+C'+D) \\
&\quad (A'+B+C'+D')(A'+B'+C+D)(A'+B'+C'+D') \\
&= (A+B+D')(A+B'+C+D')(A'+B+C')(A'+B'+C+D)(A'+B'+C'+D') \\
&= [A+D'+B(B'+C)](A'+B'+C+D)[A'+C'+B(B'+D')] \\
&= (A+D'+BC)(A'+B'+C+D)(A'+C'+BD') \\
&= (A+B+D')(A+C+D')(A'+B'+C+D)(A'+B+C')(A'+C'+D')
\end{aligned}$$

3.7.3 Incompletely Specified Minterm and Maxterm Expressions

It is possible to portray minterm and maxterm expressions that are incompletely specified (contain don't care terms). In those cases, a minterm expression will include a Σd term with the don't care terms, and a maxterm expression will include a ΠD term with the don't care terms.

The procedure for deriving a minimum SOP or minimum POS expression is identical to that given earlier in this chapter. The only difference is that we are writing down the expression differently than before.

Given the expression

$$F(A,B,C,D) = \Sigma m(0,1,3,8,10,15) + \Sigma d(2,9,14),$$

start by simplifying the minterms. Then individually inspect each of the don't care terms to see if they help to further reduce the expression.

$$\begin{aligned}
F(A,B,C,D) &= m_0 + m_1 + m_3 + m_8 + m_{10} + m_{15} \\
&= A'B'C'D' + A'B'C'D + A'B'CD + AB'C'D' + AB'CD' + ABCD \\
&= A'B'C' + A'B'CD + AB'D' + ABCD \\
&= A'B'(C' + CD) + AB'D' + ABCD \\
&= A'B'(C' + D) + AB'D' + ABCD \\
&= A'B'C' + A'B'D + AB'D' + ABCD
\end{aligned}$$

First, consider don't care d_2, which is $A'B'CD'$.

$$\begin{aligned} F(A,B,C,D) &= A'B'C' + A'B'D + AB'D' + ABCD + A'B'CD' \\ &= A'B'(C' + D + CD') + AB'D' + ABCD \\ &= A'B'(C' + D + C) + AB'D' + ABCD \\ &= A'B' + AB'D' + ABCD \\ &= B'(A' + AD') + ABCD \\ &= B'(A' + D') + ABCD \\ &= A'B' + B'D' + ABCD \end{aligned}$$

This is a great improvement over the previous expression. Next, consider don't care d_9 which is $AB'C'D$.

$$\begin{aligned} F(A,B,C,D) &= A'B' + B'D' + ABCD + AB'C'D \\ &= B'(A' + AC'D + D') + ABCD \\ &= B'(A' + C'D + D') + ABCD \\ &= B'(A' + C' + D') + ABCD \\ &= A'B' + B'C' + B'D' + ABCD \end{aligned}$$

This introduces a new term into the expression, and therefore should not be used. The last don't care term is d_{14}, or $ABCD'$.

$$\begin{aligned} F(A,B,C,D) &= A'B' + B'D' + ABCD + ABCD' \\ &= A'B' + B'D' + ABC \end{aligned}$$

The final minimum SOP expression is therefore $F(A,B,C,D) = A'B' + B'D' + ABC$.

3.7.4 Converting Minterm and Maxterm Expressions

If we have a minterm expression and want to identify the maxterm expression for that function, we just need to identify the terms that would have an output of zero. This means that all of the numbers between zero and $2^n - 1$ that aren't minterms or don't cares will be maxterms. As an example, let's consider the minterm expression from the previous section's example:

$$F(A,B,C,D) = \Sigma m(0,1,3,8,10,15) + \Sigma d(2,9,14).$$

The terms that are not accounted for as minterms or don't cares are four, five, six, seven, eleven, twelve,

and thirteen. These are therefore maxterms. Don't care terms apply to both the minterm and maxterm expressions. Therefore,

$$F(A,B,C,D) = \Pi M(4,5,6,7,11,12,13)\Pi D(2,9,14).$$

The same process applies in reverse to find a minterm expression from a maxterm expression.

$$\begin{aligned} F(A,B,C,D) &= \Pi M(3,5,8)\Pi D(1,7,9) \\ &= \Sigma m(0,2,4,6,10,11,12,13,14,15) + \Sigma d(1,7,9) \end{aligned}$$

3.7.5 Minterm and Maxterm Expansions

If we start with a truth table, it is straightforward to derive the minterm and maxterm expressions. However, if we start with a Boolean algebra expression or a circuit diagram, it may not be clear how to obtain a minterm or maxterm expression. We can expand the Boolean expression until we get what's called a minterm expansion or a maxterm expansion. (Why would this be useful? Because if we can simplify the expression, it can be easier to start with minterms or maxterms to use a k-map as we'll do in the next chapter in this book.)

The expansion process consists of doing uniting steps in reverse. Essentially, we want every term in our Boolean expression to have every input variable accounted for.

As an example, let's consider $F(A,B,C) = A'C + A'B + BC$. If we want to know which minterms are associated, we could create a truth table. Instead we will expand each term. Whichever of the three variables is missing from each term will be expanded on using the complementary law ($A + A' = 1$). Any repeat terms will be eliminated using the idempotent law ($A + A = A$).

$$\begin{aligned} F(A,B,C) &= A'C + A'B + BC \\ &= A'(B+B')C + A'B(C+C') + (A+A')BC \\ &= A'BC + A'B'C + A'BC + A'BC' + ABC + A'BC \\ &= A'BC + A'B'C + A'BC' + ABC \\ &= m_3 + m_1 + m_2 + m_7 \\ &= \Sigma m(1,2,3,7) \end{aligned}$$

A similar process occurs in a maxterm expansion. The complementary law that is used now is $AA' = 0$, and the idempotent law is $AA = A$. The following example is a function of four variables, which means that

the expansion may need to be repeated a few times until every variable is accounted for.

$$
\begin{aligned}
F(A,B,C,D) &= (A+D')(A'+B+C) \\
&= (A+BB'+D')(A'+B+C+DD') \\
&= (A+B+D')(A+B'+D')(A'+B+C+D)(A'+B+C+D') \\
&= (A+B+CC'+D')(A+B'+CC'+D')(A'+B+C+D)(A'+B+C+D') \\
&= (A+B+C+D')(A+B+C'+D')(A+B'+C+D')(A+B'+C'+D') \\
&\quad (A'+B+C+D)(A'+B+C+D') \\
&= M_1M_3M_5M_7M_8M_9 \\
&= \Pi M(1,3,5,7,8,9)
\end{aligned}
$$

3.8 Example Problems

Design Applications

1. You receive a 5-bit signed binary number $ABCDE$ corresponding to the temperature of a freezer in degrees Celsius. If the temperature rises above -14 °C, the compressor should turn on. If the temperature rises above -5 °C, a warning light should turn on. Define both output variables and express them as minimum SOP functions.

2. You receive two 2-bit numbers designated as AB and CD. If $AB > CD$, a red LED should turn on with all other LEDs off. If $AB < CD$, a green LED should turn on with all other LEDs off. If $AB = CD$, a blue LED should turn on with all other LEDs off. Derive minimum SOP expressions to control each LED. Assume that a value of zero corresponds to an LED that is off, and a value of one corresponds to an LED that is on.

3. Create a programmable logic gate with inputs A and B, control bits X and Y, and output F. If $X = Y = 0$, then $F = AB$. If $X = Y = 1$, then $F = A + B$. If $X = 1$ and $Y = 0$, then $F = 0$. Any other combination of X and Y will never occur. Derive a minimum SOP expression for F.

4. The motor of a vending machine must run if enough money has been inserted into the machine and if the selection button has been pressed. Change needs to come out of the vending machine if too much money has been inserted and the selection button has been pressed. Create a minimum SOP or POS expression for each output in terms of the provided inputs. Ensure that each variable is well-defined.

5. Use half-adders to create a 2-bit unsigned binary multiplier. Draw a schematic diagram of the circuit.

Minterm Expressions

1. Find a minimum Boolean expression (either SOP or POS) for $F(A, B, C) = \Sigma m(0, 1, 5, 6)$.

2. Find a minimum SOP expression for $F(A, B, C, D) = \Sigma m(0, 8, 9, 13, 15) + \Sigma d(4, 7, 11)$. Indicate which (if any) of the don't care terms you used to minimize the expression.

3. Find a minimum SOP expression for $F(A, B, C) = \Sigma m(0, 1, 3, 4, 5)$.

4. Find a minimum SOP expression for $F(A, B, C, D) = \Sigma m(0, 4, 7, 13) + \Sigma d(2, 5, 10, 15)$. Indicate which (if any) of the don't care terms you used to minimize the expression.

5. Find a minimum POS expression for $F(A, B, C, D) = \Sigma m(5, 7, 10, 13) + \Sigma d(0, 11, 15)$. Indicate which (if any) of the don't care terms you used to minimize the expression.

Maxterm Expressions

1. Find a minimum SOP expression for $F(A, B, C) = \Pi M(2, 3, 5, 7)$
2. Find a minimum POS expression for $F(A, B, C) = \Pi M(2, 3, 5)\Pi D(0, 7)$. Indicate which (if any) of the don't care terms you use to minimize the expression.
3. Find a minimum POS expression for $F(A, B, C, D) = \Pi M(1, 2, 14)\Pi D(0, 3, 8, 15)$. Indicate which (if any) of the don't care terms you use to minimize the expression.
4. Find a minimum POS expression for $F(A, B, C, D) = \Pi M(2, 3, 6, 8, 9, 13)\Pi D(1, 10, 12)$. Indicate which (if any) of the don't care terms you use to minimize the expression.
5. Find a minimum SOP expression for $F(A, B, C, D) = \Pi M(1, 2, 3, 4, 6, 8, 9, 12, 14)\Pi D(0, 11, 15)$. Indicate which (if any) of the don't care terms you use to minimize the expression.

Minterm and Maxterm Expansions

1. Express $F = (A' + C)D + A'BCD$ as a minterm expansion of four variables.
2. Express $F = AB' + B'D + A'BC$ as a minterm expansion of four variables.
3. Express $F = A(B' + C)(C' + D')$ as a maxterm expansion of four variables.
4. Express $F = (B \oplus C)(A + D)$ as a minterm expansion of four variables.
5. Express $F = (A + BD')(C + AD')$ as a maxterm expansion of four variables.

4 Karnaugh Maps

We will now introduce Karnaugh maps (k-maps) which are used to graphically minimize SOP and POS expressions. They eliminate the difficulty of minimizing expressions using Boolean algebra. Data from a truth table (or minterm/maxterm expression) is placed into a map, and then loops are created around adjacent ones (for SOP) or zeros (for POS). Those loops use the uniting theory to create minimized expressions.

The loops that are created on a k-map must have a size that is a power of two by a power of two. That is, a loop can be 1×1, 1×2, 2×1, 2×2, etc. Each loop that is created should be as large as possible, as long as it abides by the rule of being a power of two by a power of two ($2^n \times 2^m$).

The feature of a k-map that allows power of two loops to create minimized Boolean expressions is the fact that Gray code is used to ensure that any two cells on a k-map are adjacent to each other. When moving from one cell to another on a k-map only one variable changes from a zero to one or from a one to zero.

This adjacency applies not only to cells that are directly next to each other, but also to cells that loop from top to bottom or side to side. Figure 4.1 shows the binary values that are associated with the input variables in each cell of a three variable k-map. As you can see, cell 0 (000) is adjacent to both cell 1 (001) and cell 4 (100), but also to cell 2 (010) which is at the "bottom" of the k-map.

	A	
BC	**0**	**1**
00	000	100
01	001	101
11	011	111
10	010	110

Figure 4.1: Three variable k-map showing the binary values of each cell.

The algorithm for solving a k-map follows.

- Find a one (for SOP) or zero (for POS) that can only be looped one way, and loop it in the largest possible loop that is a power of two by a power of two.
- Repeat with remaining ones (for SOP) or zeros (for POS) until all ones or zeros have been looped. Overlap loops as required to make the largest possible loops.
- Unless otherwise specified, do not include redundant loops, as these just make the equation longer than necessary.

4.1 Three Variable K-Maps

As stated, three variable k-maps use Gray code along one side of the map to ensure that any two cells that are adjacent to each other (including wrapping from top to bottom) only differ by one variable. Figure 4.1 above shows the binary values that correspond to each cell. Figure 4.2 shows the decimal values that correspond to each cell. This helps us to determine how to fill values in to the k-map.

	A	
BC	**0**	**1**
00	0	4
01	1	5
11	3	7
10	2	6

Figure 4.2: 3 variable k-map showing the decimal values of each cell.

The best way to understand how to use a k-map is to do some examples.

Find the minimum SOP expression for

$$F(A, B, C) = \Sigma m(1, 3, 4, 5, 7).$$

Start by putting a one in the corresponding cell for each minterm. An $\times$ would go into any don't care cell (which does not apply to this example). All of the remaining cells get filled in with a zero. The filled out k-map is shown in Figure 4.3.

	A	
BC	**0**	**1**
00	0	1
01	1	1
11	1	1
10	0	0

Figure 4.3: Filled in k-map for $F(A, B, C) = \Sigma m(1, 3, 4, 5, 7)$.

The next step is to find ones (for SOP, as instructed) that can only be looped in one way. The best example is minterm m_4, which is adjacent to only m_5 and no other minterm. Loop it in the biggest possible loop that is a power of two by a power of two. m_4 and m_5 would therefore be looped together. Let's look at the Boolean algebra that would result from combining the two minterms.

$$\begin{aligned} m_4 + m_5 &= AB'C' + AB'C \\ &= AB' \end{aligned}$$

The uniting step removes a variable (in this case C) from the expression. Many students are tempted to include minterm m_7, however, that loop, being of size 1×3 is not a power of two by a power of two.

$$\begin{aligned} m_4 + m_5 + m_7 &= AB'C' + AB'C + ABC \\ &= AB' + ABC \end{aligned}$$

This does NOT allow us to do two uniting steps, so it is not allowed as a loop. Our k-map, with minterms m_4 and m_5 looped together is shown in Figure 4.4.

	A	
BC	**0**	**1**
00	0	1
01	1	1
11	1	1
10	0	0

Figure 4.4: Filled in k-map for $F(A, B, C) = \Sigma m(1, 3, 4, 5, 7)$ with minterms m_4 and m_5 looped.

There are three remaining minterms: m_1, m_3, and m_7. These can all be looped together in a 2×2 loop that includes m_5. It is important to include m_5 in the loop so that it can be as large as possible. Otherwise, our expression would not be minimized. Let's look at the Boolean algebra that results in creating this loop. (Again, the point of a k-map is to avoid having to use Boolean algebra; this derivation is included so you can see why and how a k-map works.)

$$\begin{aligned} m_1 + m_3 + m_5 + m_7 &= A'B'C + A'BC + AB'C + ABC \\ &= A'C + AC \\ &= C \end{aligned}$$

The completed, fully looped k-map is shown in Figure 4.5.

	A	
BC	**0**	**1**
00	0	1
01	1	1
11	1	1
10	0	0

Figure 4.5: Completed k-map for $F(A, B, C) = \Sigma m(1, 3, 4, 5, 7)$.

4.2 Four Variable K-maps

Four variable k-maps use Gray code in both the rows and the columns to assure adjacency between cells. Not only do cells wrap from bottom to top, but also from side to side. For example, cell 0 (0000) is adjacent to cell 1 (0001), cell 4 (0100), cell 8 (1000), and cell 2 (0010). The decimal values that correspond to each cell in a four variable k-map are shown in Figure 4.6.

	AB			
CD	**00**	**01**	**11**	**10**
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

Figure 4.6: Four variable k-map showing the decimal values of each cell.

As an example, find the minimum POS expression for

$$F(A, B, C, D) = \Pi M(1, 5, 6, 7, 9, 12, 13, 14).$$

First, place a zero in all of the cells corresponding to the maxterms. There are no don't care terms in this

expression, so all of the other cells must be filled in with a one. The completed k-map is shown in Figure 4.7.

	AB			
CD	**00**	**01**	**11**	**10**
00	1	1	0	1
01	0	0	0	0
11	1	0	1	1
10	1	0	0	1

Figure 4.7: Filled in k-map for $F(A, B, C, D) = \Pi M(1, 5, 6, 7, 9, 12, 13, 14)$.

Maxterm M_1 can only be looped in one way, so loop it in as big a loop as possible. This loop is 4×1 and covers maxterms M_1, M_5, M_9, and M_{13}. Note that the variables A and B change for all values along this loop. However, C is always zero and D is always one, so the corresponding sum term for this loop is $(C + D')$. The Boolean algebra procedure that we would have used on this loop follows.

$$\begin{aligned} M_1 M_5 M_9 M_{13} &= (A + B + C + D')(A + B' + C + D')(A' + B' + C + D')(A' + B + C + D') \\ &= (A + C + D')(A' + C + D') \\ &= (C + D') \end{aligned}$$

Maxterms M_6, M_7, M_{12}, and M_{14} remain unlooped. None of these can be looped in only one way. However, after inspecting the k-map carefully, the fewest number of loops to cover these terms would mean looping M_6 with M_7, and M_{12} with M_{14}.

The loop that covers M_6 and M_7 has a value of $A = 0$ along the entire loop. Similarly, $B = 1$, $C = 1$, and D changes. This gives a sum term of $(A + B' + C')$. The Boolean algebra showing this follows.

$$\begin{aligned} M_6 M_7 &= (A + B' + C' + D')(A + B' + C' + D) \\ &= (A + B' + C') \end{aligned}$$

The loop that covers M_{12} and M_{14} has a value of $A = 1$, $B = 1$, and $D = 0$ everywhere along the loop, while C changes. This gives a sum term of $(A' + B' + D)$.

The completed, fully-looped k-map is shown in Figure 4.8.

CD \ AB	00	01	11	10
00	1	1	0	1
01	0	0	0	0
11	1	0	1	1
10	1	0	0	1

Figure 4.8: Completed k-map for $F(A, B, C, D) = \Pi M(1, 5, 6, 7, 9, 12, 13, 14)$.

4.3 Incompletely Specified K-Maps

The procedure for a k-map with don't cares is much the same as a regular k-map, except that don't care terms should only be looped if they allow you to create a larger loop. Do not make loops that cover only don't care terms, because they are not necessary to include and just create extra logic gates and inputs.

The k-map in Figure 4.9 shows the encoding for segment e on a 7-segment display. Binary values corresponding to ten through fifteen are don't cares because they do not exist as a BCD value.

CD \ AB	00	01	11	10	
00	1	0	×	1	$(B' + C)$, $B'D'$
01	0	0	×	0	(D')
11	0	0	×	×	
10	1	1	×	×	CD'

Figure 4.9: K-map for segment e of a 7-segment display.

The SOP loops are shown in thick black lines. Including don't care term d_{10} enables us to create a 2×2 loop corresponding to $B'D'$. Including don't care terms d_{10} and d_{14} enables us to create a 4×1 loop

corresponding to CD'.

The POS loops are shown in dashed black lines. Including don't care terms D_{12} and D_{13} enable us to make the large loop corresponding to $(B' + C)$. Including don't care terms D_{11}, D_{13}, and D_{15} enables us to make a 4×2 loop corresponding to D'.

4.4 Implicants, Prime Implicants, and Essential Prime Implicants

At this point, we know how to create loops on a k-map. However, it is not always clear which loops to make to lead to a minimum expression. Before we can learn a tool to enable us to find minimum expressions, we need to understand some properties of k-map loops.

4.4.1 Implicants

In an SOP expression, an implicant is a one or any group of ones that can be looped, regardless of the size. In a POS expression, an implicant is a zero or any group of zeros that can be looped, regardless of the size.

The k-map shown in Figure 4.10 has seventeen implicants that are looped.

	AB			
CD	**00**	**01**	**11**	**10**
00	1	0	1	1
01	0	0	1	1
11	1	0	0	0
10	1	1	0	0

Figure 4.10: Implicants on an example k-map.

Specifically, this k-map has

- eight loops that correspond to minterms (1×1 loops),
- eight loops that are 1×2 or 2×1, and
- one loop that is 2×2.

Including every single implicant in our expression would clearly not be minimized by any means. We will continue expanding our definitions.

4.4.2 Prime Implicants

A prime implicant is any implicant that cannot be combined with another implicant to eliminate a variable. In other words, a prime implicant implies that we have made the largest possible loop.

Our k-map from above can now be looped with just prime implicants, shown in Figure 4.11. There are only five prime implicants, which include

- four loops that are 1×2 or 2×1, and
- one loop that is 2×2.

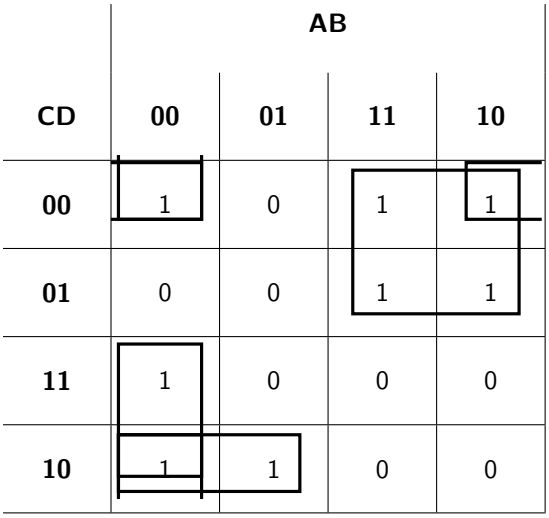


Figure 4.11: Prime implicants on an example k-map.

The fact that there are five prime implicants (or PIs) is much better than the seventeen implicants. However, if you are observant, you may note that not all of them are truly necessary for a minimum SOP expression.

A minimum Boolean expression (whether SOP or POS) contains the minimum number of prime implicants that cover all minterms (for SOP) or maxterms (for POS) on the k-map.

4.4.3 Essential Prime Implicants

A prime implicant that is the only one that covers a specific minterm or maxterm is known as an essential prime implicant. It **must** be included in the final Boolean expression.

In the above example, minterms m_3, m_6, m_9, m_{12}, and m_{13} are only covered by a single loop. That means that the loops that cover those minterms are essential prime implicants (or EPIs).

- $A'B'C$ is an EPI because it is the only loop that covers minterm m_3.

- $A'CD'$ is an EPI because it is the only loop that covers minterm m_6.
- AC' is an EPI because it is the only loop that covers minterms m_9, m_{12}, and m_{13}.

These three terms **must** be included in our final expression. The only minterm left, once these three loops have been included, is minterm m_0. There are two loops that cover m_0, so there are two correct, equivalent, minimum SOP expressions for the k-map.

$$\begin{aligned} F(A,B,C,D) &= A'B'C + A'CD' + AC' + B'C'D' \\ &= A'B'C + A'CD' + AC' + A'B'D' \end{aligned}$$

When we discussed function equivalence earlier in this textbook, it was mentioned that a truth table is the only method for determining function equivalence. As the above example demonstrates, it is possible to have equivalent functions that have different Boolean algebra expressions. Comparing truth tables (or minterm expansions) is the only way to determine that the expressions are indeed equivalent.

At this point, how do we know which of the PIs should be included in a minimum expression? How do we know when we have a truly minimum SOP or POS expression? We can use a prime implicant table.

4.5 Prime Implicant Tables

A prime implicant table shows a listing of all of the PIs in an expression. The columns in the table list all of the minterms (for SOP) or maxterms (for POS). The don't care terms are not included.

Let's consider a PI table for

$$F(A,B,C,D) = \Sigma m(0,4,5,7,11,14,15) + \Sigma d(1,10,12).$$

Each prime implicant is listed, and an $\times$ is placed in each column whose minterm is included in the PI. This PI table is given in table 4.1.

PI	0	4	5	7	11	14	15
A'C'	×	×	×				
AC					×	×	×
BCD				×			×
A'BD			×	×			
BC'D'		×					
ABD'						×	

Table 4.1: PI table for $F(A,B,C,D) = \Sigma m(0,4,5,7,11,14,15) + \Sigma d(1,10,12)$.

The first step is to identify the columns that only have a single $\times$ placed in them. These indicate the minterms that are only covered by a single loop. The columns in this particular case correspond to minterms m_0 and m_{11}. The PIs covering those minterms ($A'C'$ and AC) are therefore essential.

Essential PI $A'C'$ takes care of minterms m_0, m_4, and m_5. Therefore we can stop considering those columns. They have been taken care of. Essential PI AC takes care of minterms m_{11}, m_{14}, and m_{15}. We no longer have to consider those columns, as they have been taken care of.

The only remaining columns is m_7. Therefore, we only need pick **one** of the PIs that covers that minterm. There are two valid minimum SOP expressions, based on the selection of the PI that covers m_7.

$$\begin{aligned} F(A,B,C,D) &= A'C' + AC + BCD \\ &= A'C' + AC + A'BD \end{aligned}$$

We can similarly make a PI table for the maxterm (POS) expression. That PI table is given in table 4.2.

PI	2	3	6	8	9	13
A'+C				×	×	×
A+C'+D			×			
A+B+D'		×				
B+C+D'					×	
A+B+C'	×	×				
B+C'+D	×					
A'+B+D				×		

Table 4.2: PI table for $F(A,B,C,D) = \Pi M(2,3,6,8,9,13)\Pi D(1,10,12)$.

The first step is to identify maxterms (columns) that only have a single ×. Those columns correspond to maxterms M_6 and M_{13}. Therefore $(A'+C)$ and $(A+C'+D)$ are essential PIs. After those two EPIs are considered, the only two maxterms that are left to be looped are M_2 and M_3. The only PI that covers all of these maxterms is $(A+B+C')$, therefore that PI will lead to a minimum POS expression.

$$F(A,B,C,D) = (A'+C)(A+C'+D)(A+B+C')$$

At times, it can be difficult to determine what all of the prime implicants of an expression are. For example, consider the expression

$$F(A,B,C,D) = \Sigma m(0,1,3,4,5,9,11,12) + \Sigma d(2,6,10,13,14,15).$$

It is very simple to find the POS prime implicants. However, there are eleven prime implicants corresponding to the SOP terms. We will learn how to find all of the prime implicants using the Quine-McCluskey method in the next chapter. However, the use of a prime implicant table is very important to find a minimum expression once you have found all of the PIs. The k-map for this expression is shown in Figure 4.12.

	AB			
CD	**00**	**01**	**11**	**10**
00	1	1	1	0
01	1	1	×	1
11	1	0	×	1
10	×	×	×	×

Figure 4.12: K-map with many prime implicants.

The prime implicant table is given in table 4.3. All eleven of the SOP prime implicants are shown. In addition to placing an × in each column, the number of minterms covered by each term is indicated in the last column. It is clear that none of these prime implicants is essential, making the expression difficult to minimize.

PI	**0**	**1**	**3**	**4**	**5**	**9**	**11**	**12**	
0-1-2-3 (A'B')	×	×	×						3
0-1-4-5 (A'C')	×	×		×	×				4
0-2-4-6 (A'D')	×			×					2
1-3-9-11 (B'D)		×	×			×	×		4
1-5-9-13 (C'D)		×			×	×			3
2-3-10-11 (B'C)			×				×		2
4-5-12-13 (BC')				×	×			×	3
4-6-12-14 (BD')				×				×	2
9-11-13-15 (AD)						×	×		2
10-11-14-15 (AC)							×		1
12-13-14-15 (AB)								×	1

Table 4.3: Prime implicant table for the eleven PIs.

It is important to start by selecting the PIs that will cover the most minterms. Start by choosing either $B'D$ or $A'C'$. (In this case, it doesn't matter which.) If you choose $A'C'$, you no longer need to consider minterms m_0, m_1, m_4, and m_5. Reconsider the remaining PIs and how many of the remaining minterms that they will cover. This is shown in table 4.4. Any of the PIs that do not cover any of the remaining minterms have been removed from the list for brevity's sake.

PI	3	9	11	12	
0-1-2-3 (A'B')	×				1
1-3-9-11 (B'D)	×	×	×		3
1-5-9-13 (C'D)		×			1
2-3-10-11 (B'C)	×		×		2
4-5-12-13 (BC')				×	1
4-6-12-14 (BD')				×	1
9-11-13-15 (AD)		×	×		2
10-11-14-15 (AC)			×		1
12-13-14-15 (AB)				×	1

Table 4.4: Minimized prime implicant table once $A'C'$ has been chosen.

Now $B'D$ is the most effective PI, as it wipes out three more minterms, m_3, m_9, and m_{11}. We no longer need to consider those minterms. The remaining PIs are shown in table 4.5.

PI	12
4-5-12-13 (BC')	×
4-6-12-14 (BD')	×
12-13-14-15 (AB)	×

Table 4.5: Minimized prime implicant table once $A'C'$ and $B'D$ have been chosen.

The only minterm left is m_{12}. Therefore you have a choice of three PIs that cover that term. There are three valid minimum SOP expressions corresponding to this function.

$$\begin{aligned} F(A,B,C,D) &= A'C' + B'D + BC' \\ &= A'C' + B'D + BD' \\ &= A'C' + B'D + AB \end{aligned}$$

A PI table made this otherwise overwhelming k-map much more straightforward to minimize.

4.6 Five Variable K-Maps

It is possible to use a k-map to solve five variable expressions. It is even possible to do six variable expressions, but those will not be covered in this book. For expressions with more than six variables, a k-map is no longer going to be a convenient method to solve for a minimum expression: Quine-McCluskey (explained in the next chapter) should be used instead.

The difficulty with a five variable k-map is with how adjacent cells manifest themselves. In a three and four variable k-map, it was rather straightforward to see how two cells could be adjacent to each other. The introduction of another "dimension" in our k-map creates even more adjacencies. For this reason, I like to draw a thick line down the center of my k-map and treat it like a mirror. In this textbook the mirror line will be indicated with a double line. Any cell that sees its reflection in that mirror line is adjacent to that reflection.

A five variable k-map is shown in Figure 4.13 with the decimal values given in each cell. Cell 4 (00100) is adjacent to cell 0 (00000), cell 5 (00101), cell 6 (00110), and cell 12 (01100) as discussed in a four variable k-map. But it is now also adjacent to cell 20 (10100) as that cell is the "mirror image" of cell 4 (00100).

	ABC							
DE	**000**	**001**	**011**	**010**	**110**	**111**	**101**	**100**
00	0	4	12	8	24	28	20	16
01	1	5	13	9	25	29	21	17
11	3	7	15	11	27	31	23	19
10	2	6	14	10	26	30	22	18

Figure 4.13: Five variable k-map showing the decimal values of each cell.

Example: Solving minimum SOP with a five variable k-map

Use a k-map to find a minimum SOP expression for

$$F(A,B,C,D,E) = \Sigma m(4,6,13,20,21,22,23,28,29,30,31) + \Sigma d(15).$$

The k-map with loops are shown below. Note the loops that see their "reflection" in the mirror line.

	ABC							
DE	**000**	**001**	**011**	**010**	**110**	**111**	**101**	**100**
00	0	1	0	0	0	1	1	0
01	0	0	1	0	0	1	1	0
11	0	0	×	0	0	1	1	0
10	0	1	0	0	0	1	1	0

$B'CE'$

BCE

AC

The minimum expression is $F(A,B,C,D,E) = B'CE' + BCE + AC$.

4.7 Example Problems

Three Variable K-Maps

1. Use a k-map to find a minimum SOP expression for

$$F(A, B, C) = \Sigma m(0, 1, 3, 4, 6).$$

2. Create a k-map for the expression $F(A, B, C) = AB + AB' + BC$ and use it to reduce F to minimum SOP.

3. Identify all of the implicants contained in $F(A, B, C) = A \oplus B + B'C'$.

4. Use a k-map of $F(A, B, C) = AB + BC'$ to identify the minterms.

5. $F(A, B, C) = A'C + AB'$ and $G(A, B, C) = (A' + C')(B' + C)(A + C)$ are equivalent expressions. Use a k-map to determine which term(s) must be don't cares.

Four Variable K-Maps

1. Use a k-map to find a minimum SOP expression for

$$F(A, B, C, D) = \Pi M(0, 3, 4, 8, 9, 10, 14).$$

2. Use a k-map to find a minimum SOP expression for

$$F(A, B, C, D) = \Sigma m(2, 3, 6, 7, 8, 11) + \Sigma d(0, 10, 12, 13).$$

3. Use a k-map to find a minimum POS expression for

$$F(A, B, C, D) = \Sigma m(1, 3, 4, 5, 6, 12, 14, 15).$$

4. A sensor is capable of determining whether or not a car is speeding (driving faster than the speed limit) or driving dangerously (driving 10 m.p.h. or more above the speed limit). The sensor receives the codes given in table 4.6, where AB corresponds to the speed limit and CD corresponds to the speed of the vehicle. Use a k-map to solve for F (as either minimum SOP or minimum POS), which indicates if the car is speeding.

AB	Speed Limit	CD	Car's Speed
00	45 m.p.h.	00	$<$ 45 m.p.h.
01	55 m.p.h.	01	46–55 m.p.h.
10	65 m.p.h.	10	56–65 m.p.h.
11	unused	11	66–75 m.p.h.

Table 4.6: Speed codes used for four variable k-maps questions 4 and 5.

5. Continuing on the above question, use a k-map to solve for G (as either minimum SOP or minimum POS), which indicates if the car is driving dangerously.

Prime Implicant Tables

1. Use a PI table to find a minimum SOP expression for

$$F(A, B, C, D) = \Sigma m(2, 5, 6, 7, 10, 14).$$

2. Use a PI table to find a minimum POS expression for

$$F(A, B, C, D) = \Pi M(0, 1, 2, 3, 6, 9, 14).$$

3. Use a PI table to find a minimum SOP expression for

$$F(A, B, C, D, E) = \Sigma m(5, 10, 11, 15, 26, 30) + \Sigma d(7, 13, 31).$$

4. Use a PI table to find a minimum SOP expression for

$$F(A, B, C, D) = \Sigma m(2, 4, 5, 10, 11, 13, 14, 15) + \Sigma d(1, 8).$$

5. Identify the essential PIs by using a PI table for

$$F(A, B, C, D, E) = \Pi M(1, 2, 3, 4, 5, 12, 13, 15, 16, 17, 20, 28, 29)\Pi D(6, 14, 18, 19, 22, 30).$$

5 Quine-McCluskey Method

The advantage of k-maps, as we saw, was creating visual patterns of ones and zeros that enable us to effectively do uniting steps without actually doing any Boolean algebra. However, electrical engineers do not get paid to work at a desk with a pen and paper and solve k-maps. Computers solve for minimum expressions. Computers are not as good as humans at spotting visual patterns, they are better at following step-by-step instructions (called algorithms).

The Quine-McCluskey method is an algorithm that can be used to minimize Boolean expressions without having to use human pattern-spotting skills. When done correctly, it is also a foolproof method of identifying every implicant and prime implicant in an expression.

In this book, we will only cover SOP expressions with Quine-McCluskey. However, it is possible to carry out the same steps using maxterms and zeros (instead of minterms and ones) to solve for a POS expression.

The Quine-McCluskey procedure follows.

1. Start by creating a number of columns equal to the number of input variables.
2. In the first column, write down all of the minterms ordered in groups by the number of ones contained in each minterm.
3. Starting with the first group, compare each minterm with all of the minterms in the succeeding group. If they differ by one bit, they go into the next column and get a checkmark.
4. Repeat step three until all of the groups in column one have been compared with each other.
5. Move to the next column. Repeat steps three and four until you either run out of columns, or no more terms can be combined in such a manner.
6. Eliminate repeat terms.
7. Any term that does not have a checkmark next to it is a prime implicant.
8. Use a prime implicant table to obtain a minimum SOP expression.

Because the best way to learn Quine-McCluskey is to practice, this chapter will consist completely of different and diverse examples of the Quine-McCluskey method.

Example: Three-variable Quine-McCluskey

Use the Quine-McCluskey method to find a minimum SOP expression for

$$F(A, B, C) = \Sigma m(1, 3, 4, 5, 6, 7).$$

The first step is to create a table with three columns. Then, in column one, we write down all of the

minterms in groups ordered by the number of ones in each term.

Column 1		Column 2	Column 3
one	1. 001		
	4. 100		
two	3. 011		
	5. 101		
	6. 110		
three	7. 111		

We compare every term in group one with every term in group two. For example minterms m_1 and m_3 differ by only one variable. In comparing 001 and 011 we see that the second bit changes. This becomes $0-1$ which goes into column two. Minterms m_1 and m_3 get checkmarks. m_1 is then compared with both m_5 and m_6. Next, m_4 is compared with every minterm in group two (m_3, m_5, and m_6). Once that process is complete, groups two and three in column one are compared.

Column 1		Column 2	Column 3
one	1. 001 ✓	1-3: 0−1	
	4. 100 ✓	1-5: −01	
two	3. 011 ✓	4-5: 10−	
	5. 101 ✓	4-6: 1−0	
	6. 110 ✓	3-7: −11	
three	7. 111 ✓	5-7: 1−1	
		6-7: 11−	

Now we can compare the terms in column two groups one and two with each other. Every term in group one gets compared with every term in group two. If they differ by only one bit, they get checked and placed in column three.

Column 1		Column 2	Column 3
one	1. 001 ✓	1-3: 0−1 ✓	1-3-5-7: −−1
	4. 100 ✓	1-5: −01 ✓	~~1-5-3-7~~
two	3. 011 ✓	4-5: 10− ✓	4-5-6-7: 1−−
	5. 101 ✓	4-6: 1−0 ✓	~~4-6-5-7~~
	6. 110 ✓	3-7: −11 ✓	
three	7. 111 ✓	5-7: 1−1 ✓	
		6-7: 11− ✓	

At this point, there are no more columns to compare. We can eliminate the repeat terms (1-5-3-7

and 4-6-5-7 are repeats) and analyze the terms that have no checkmarks (which are PIs). There are two PIs that have been identified in this method.

- 1-3-5-7: $--1$ which means that A changed, B changed, and C was always one. This product term is therefore C.
- 4-5-6-7: $1--$ which means that A was always one, B changed, and C changed. This product term is therefore A.

We can create a PI table for these two PIs, but it is pretty clear that both are essential (C is the only PI that covers m_1 and m_3, and A is the only PI that covers m_4 and m_6), so a PI table will not be shown.

$$F(A, B, C) = A + C$$

Example: Quine-McCluskey with don't cares

Find a minimum SOP expression for $F(A, B, C, D) = \Sigma m(3, 7, 9, 11, 13) + \Sigma d(1, 10, 15)$. When there are don't care terms, treat them as minterms when going through each of the columns. This helps to combine and minimize terms. The first step is to create four columns. Place all of the minterms and don't cares into column one, ordered by the number of ones in the term.

Column 1		Column 2	Column 3	Column 4
one	1. 0001			
two	3. 0011			
	9. 1001			
	10. 1010			
three	7. 0111			
	11. 1011			
	13. 1101			
four	15. 1111			

Compare all of the minterms in column one group one with all of the minterms in column one group two. Then all of the terms in group two are compared with the terms in group three. Then all of the terms in group three are compared with all of the terms in group four.

Column 1		Column 2		Column 3	Column 4
one	1. 0001 ✓	1-3:	00−1		
two	3. 0011 ✓	1-9:	−001		
	9. 1001 ✓	3-7:	0−11		
	10. 1010 ✓	3-11:	−011		
three	7. 0111 ✓	9-11:	10−1		
	11. 1011 ✓	9-13:	1−01		
	13. 1101 ✓	10-11:	101−		
four	15. 1111 ✓	7-15:	−111		
		11-15:	1−11		
		13-15:	11−1		

Do the same for all of the groups in column two. Notice that $m_{10} - m_{11}(AB'C)$ cannot be combined with another term in group two, making it a prime implicant.

Column 1		Column 2		Column 3		Column 4
one	1. 0001 ✓	1-3:	00−1 ✓	1-3-9-11:	−0−1	
two	3. 0011 ✓	1-9:	−001 ✓	~~1-9-3-11~~		
	9. 1001 ✓	3-7:	0−11 ✓	3-7-11-15:	−−11	
	10. 1010 ✓	3-11:	−011 ✓	~~3-11-7-15~~		
three	7. 0111 ✓	9-11:	10−1 ✓	9-11-13-15:	1−−1	
	11. 1011 ✓	9-13:	1−01 ✓	~~9-13-11-15~~		
	13. 1101 ✓	10-11:	101−			
four	15. 1111 ✓	7-15:	−111 ✓			
		11-15:	1−11 ✓			
		13-15:	11−1 ✓			

None of the terms in column three differ by one bit with the terms in the other group. Nothing moves to column four. All of the terms that do not have checkmarks are PIs. Use a prime implicant table to find a minimum expression.

PI	3	7	9	11	13
10-11 (AB'C)				×	
1-3-9-11 (B'D)	×		×	×	
3-7-11-15 (CD)	×	×		×	
9-11-13-15 (AD)			×	×	×

Terms CD and AD are essential prime implicants. They cover all minterms, so no additional

prime implicants are required in the expression.

$$F(A, B, C, D) = CD + AD$$

Example: Prime implicants in every column

Find a minimum SOP expression corresponding to $F(A, B, C, D) = \Sigma m(0, 3, 7, 11, 12, 14, 15)$. Start by creating four columns. Then, order all of the minterms in column one. Note that there aren't any minterms in group one!

Column 1		**Column 2**	**Column 3**	**Column 4**
zero	0. 0000			
one	–			
two	3. 0011			
	12. 1100			
three	7. 0111			
	11. 1011			
	14. 1110			
four	15. 1111			

When comparing terms in each successive group, minterm m_0 cannot be compared. It therefore does not get a checkmark which indicates that it is a prime implicant. Continue to compare terms in groups two and three, and three and four.

Column 1		**Column 2**		**Column 3**	**Column 4**
zero	0. 0000	3-7:	0−11		
one	–	3-11:	−011		
two	3. 0011 ✓	12-14:	11−0		
	12. 1100 ✓	7-15:	−111		
three	7. 0111 ✓	11-15:	1−11		
	11. 1011 ✓	14-15:	111−		
	14. 1110 ✓				
four	15. 1111 ✓				

Comparisons between groups one and two in column two are conducted. A couple of the terms are not checked, and are therefore PIs. Column three only has one group and ends the Quine-McCluskey process.

Column 1		Column 2		Column 3		Column 4
zero	0. 0000	3-7:	0−11 ✓	3-7-11-15:	−−11	
one	–	3-11:	−011 ✓	~~3-11-7-15~~		
two	3. 0011 ✓	12-14:	11−0			
	12. 1100 ✓	7-15:	−111 ✓			
three	7. 0111 ✓	11-15:	1−11 ✓			
	11. 1011 ✓	14-15:	111−			
	14. 1110 ✓					
four	15. 1111 ✓					

There are four terms without checkmarks, which means that there are four PIs. Use a PI table to determine which terms to include in the minimum SOP expression.

PI	0	3	7	11	12	14	15
0 (A'B'C'D')	×						
12-14 (ABD')					×	×	
14-15 (ABC)						×	×
3-7-11-15 (CD)		×	×	×			×

The three essential prime implicants cover all of the minterms. Therefore the term ABC is redundant and unnecessary.

$$F(A,B,C,D) = A'B'C'D' + ABD' + CD$$

Example: Five variable Quine-McCluskey

Five variable Quine-McCluskey is exactly the same process, just with possibly more steps. Find a minimum SOP expression for $F(A,B,C,D,E) = \Sigma m(4,13,22,23,29,30,31) + \Sigma d(15,28)$. In this example, only the final form of the Quine-McCluskey process is shown, once all of the comparisons have been made.

Column 1		Column 2		Column 3		Column 4
one	4. 00100	13-15:	011−1 ✓	13-15-29-31:	−11−1	
two	–	13-29:	−1101 ✓	~~13-29-15-31~~		
three	13. 01101 ✓	22-23:	1011− ✓	22-23-30-31	1−11−	
	22. 10110 ✓	22-30:	1−110 ✓	~~22-30-23-31~~		
	28. 11100 ✓	28-29:	1110−✓	28-29-30-31:	111−−	
four	15. 01111 ✓	28-30:	111−0 ✓	~~28-30-29-31~~		
	23. 10111 ✓	15-31:	−1111 ✓			
	29. 11101 ✓	23-31:	1−111 ✓			
	30. 11110 ✓	29-31:	111−1✓			
five	31. 11111 ✓	30-31:	1111− ✓			

A PI table is then used to determine which of the PIs to include in the minimum expression.

PI	4	13	22	23	29	30	31
4 (A'B'CD'E')	×						
13-15-29-31 (BCE)		×			×		×
22-23-30-31 (ACD)			×	×		×	×
28-29-30-31 (ABC)					×	×	×

The three essential PIs cover all of the minterms, therefore the last PI (ABC) does not need to be included in the expression.

$$F(A, B, C, D, E) = A'B'CD'E' + BCE + ACD$$

5.1 Example Problems

1. Use the Quine-McCluskey method to find a minimum SOP expression for

$$F(A, B, C) = \Sigma m(1, 3, 4, 5).$$

2. Use the Quine-McCluskey method to find a minimum SOP expression for

$$F(A, B, C, D) = \Sigma m(0, 2, 8, 9, 10, 11).$$

3. Use the Quine-McCluskey method to find a minimum SOP expression for

$$F(A, B, C, D, E) = \Sigma m(0, 7, 21, 23) + \Sigma d(6, 14).$$

4. Use the Quine-McCluskey method to find a minimum SOP expression for

$$F(A, B, C, D, E) = \Sigma m(0, 1, 9, 11, 16, 17, 31) + \Sigma d(13, 15, 27).$$

5. Use the Quine-McCluskey method to find a minimum SOP expression for

$$Z(A, B, C, D, E, F, G) = \Sigma m(1, 3, 5, 7, 44, 45, 60, 61, 65, 67, 69, 71, 96).$$

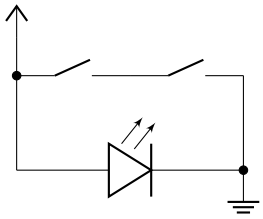
6 NAND and NOR

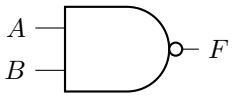
We will now introduce two new logic gates: NAND and NOR. These gates are useful because they are universal. In other words, you can make any digital logic circuit from only NAND gates, or from only NOR gates.

It is interesting to note that the Apollo Guidance Computer, which was one of the first integrated-circuit based computers built in the 1960s, used entirely 3-input NOR gates. There were approximately 2800 integrated circuits in the completed computer, which was capable of sending humans to the moon and back while keeping them alive in the vacuum of space. That's an impressive feat for a simple logic gate!

6.1 NAND

A NAND gate is a logic gate that has an output of zero only when all of the inputs are one. It is the logical complement of an AND gate, which means that for the same inputs, AND and NAND gates will have opposite outputs. A 2-input NAND gate is shown in Figure 6.1. (Note the bubble on the output of the gate, which, as discussed, represents the logical complement operation.)





A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

Figure 6.1: A NAND gate represented as a switch circuit, a circuit diagram, and a truth table.

It is important to note that, due to the DeMorgan's law, a NAND gate can be represented in two different forms.

$$\begin{aligned} A \text{ NAND } B &= (AB)' \\ &= A' + B' \end{aligned}$$

This means that a NAND gate can be depicted in three ways: a NAND gate, an AND gate with an inverted output, or an OR gate with inverted inputs. These three NAND symbols are shown in Figure 6.2.

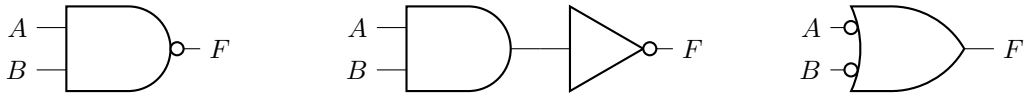


Figure 6.2: A NAND gate represented as a NAND gate, as an AND gate with an inverted output, and as an OR gate with inverted inputs.

6.1.1 NAND-Only Circuits

It is possible to create a NAND-only circuit by either starting with an SOP expression and using the double prime method, or by starting with any other type of expression using the double prime or bubble method. (The double prime method is less efficient when the expression is not SOP.)

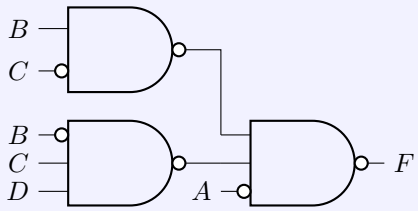
The double prime method consists of taking the complement of the function twice (the involution law tells us that this does not change the function at all). The inner prime is then distributed. At this point, the circuit becomes NAND-only.

Example: Using the double-prime method to create a NAND-only circuit

Create a NAND-only expression of $F = A + BC' + B'CD$.

$$\begin{aligned} F &= A + BC' + B'CD \\ &= [(A + BC' + B'CD)']' \\ &= [(A)'(BC')'(B'CD)']' \end{aligned}$$

At this point, all of the statements are NAND statements. The circuit diagram is shown below.



Note that if you were to continue distributing the primes and continuing to use the DeMorgan's law, you would start with an AND-OR (SOP circuit), obtain a NAND-NAND circuit, then an OR-NAND circuit, a NOR-OR circuit, and then finish back where you started with an AND-OR (SOP) circuit.

The double-prime method can be used on a factored SOP expression, but requires more than one step in the process. Double prime the expression, and distribute the inner prime. Any terms that still contain an OR operation need to be double primed, with the inner prime distributed until the expression is NAND-only.

Example: Using the double-prime method to create a NAND-only circuit with a factored SOP expression

Create a NAND-only circuit of $F = AB' + B(C + DE) + B'C'E$.

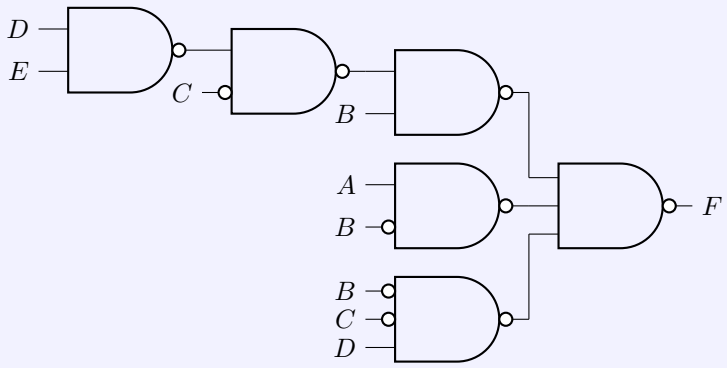
$$\begin{aligned} F &= AB' + B(C + DE) + B'C'E \\ &= [(AB' + B(C + DE) + B'C'E)']' \\ &= [(AB')'(B(C + DE))'(B'C'E)']' \end{aligned}$$

There is still an OR operation in this expression, so the term $C + DE$ itself needs to be double primed, with only the inner prime being distributed.

$$[(C + DE)']' = [(C)'(DE)']'$$

This is now NAND-only, and can get plugged back in to the original expression, making everything NAND-only.

$$\begin{aligned} F &= [(AB')'(B(C + DE))'(B'C'E)']' \\ &= [(AB')'(B[(C)'(DE)')]')'(B'C'E)']' \end{aligned}$$



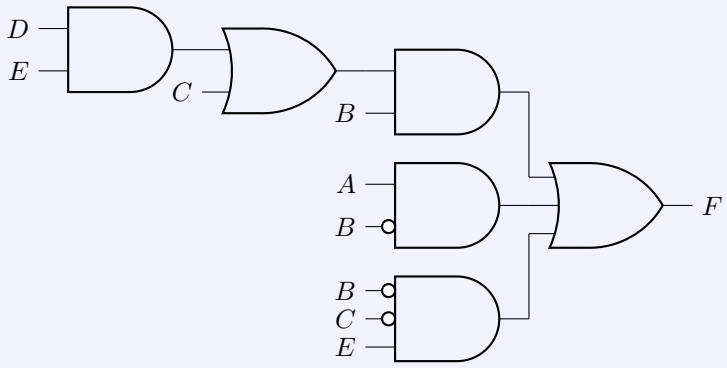
The double-prime method can be used for POS or other circuits, but is much more complicated, as double-priming a POS expression leads to a NOR-only circuit (which will be discussed in the next section). Instead, the bubble method is a more effective approach. The bubble method is a graphical method of creating NAND and NOR circuits, and can be used in lieu of the double prime method for any type of circuit.

First we need to know how to modify AND and OR gates to create a NAND gate. From Figure 6.2 we saw that a NAND gate can be created by priming the output of an AND gate, or by priming the inputs of an OR gate.

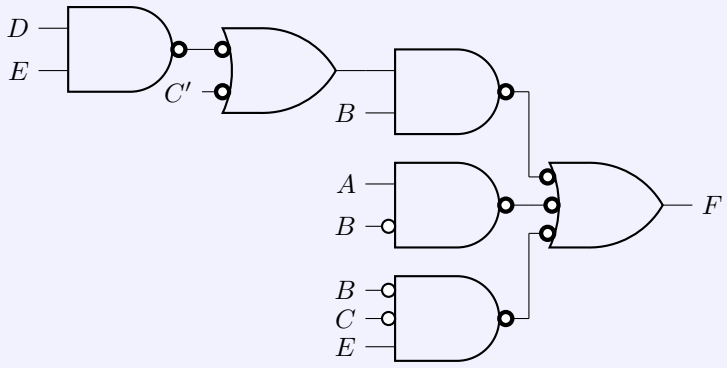
Therefore, start with any AND/OR/NOT circuit. Start all the way on the left-hand side and inspect each logic gate. If the logic gate is an AND gate, bubble the output. A second bubble needs to be placed at the other end of the wire in order to cancel out the first bubble (due to the involution law). Any time an OR gate is encountered, bubble the inputs. Again, every bubble will need to be cancelled out with a second bubble due to the involution law.

Example: Using the bubble method to create a NAND-only circuit with a factored SOP expression

Use the bubble method to find a NAND-only circuit of $F = AB' + B(C + DE) + B'C'E$.



To create a NAND-only circuit, bubble the output of every AND gate and cancel that out with a second bubble somewhere else in that path. Bubble all inputs to every OR gate and cancel that out with a second bubble somewhere else in the path. (Note that in this case, it means bubbling the input to the OR gate from $DE + C$ and then priming the C term.) If there is already a bubble on an OR gate input, then we don't have to do anything with that input.

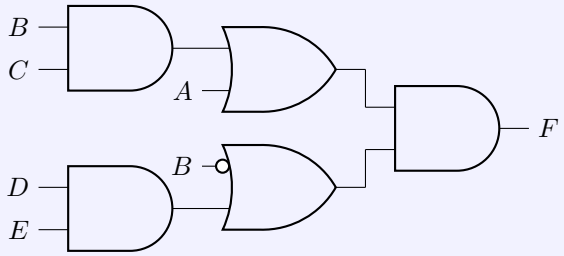


At this point, every logic gate has been converted into a NAND gate. If we were to re-draw it, it would look identical to the NAND-only circuit in the previous example.

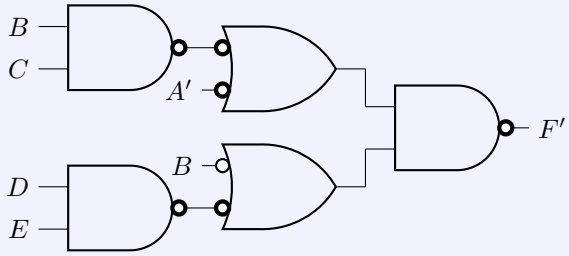
The bubble method is better suited to POS circuits due to the fact that an AND gate forms the output gate in a POS expression. When the output of that AND gate is primed, a second prime must be included somewhere along the same path to cancel it out per the involution law. Because there is nothing else to prime, we must include an output inverter.

Example: Using the bubble method to create a NAND-only circuit with a POS expression

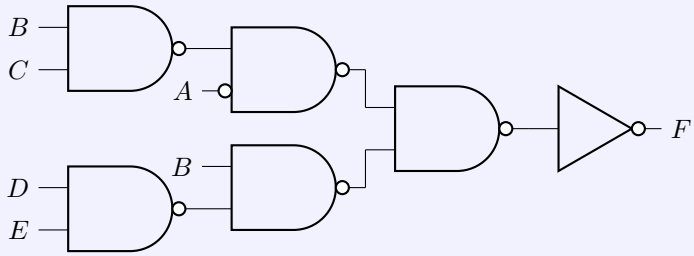
Create a NAND-only circuit for $F = (A + BC)(B' + DE)$. Start by drawing the circuit using AND gates, OR gates, and inverters.



The bubble method is exactly the same regardless of the type of circuit that is started with. To create a NAND gate, bubble all AND gate outputs and include a second bubble elsewhere in the same path. Bubble all OR gate inputs and include a second bubble elsewhere in the same path.



Re-draw the circuit using only NAND gates.



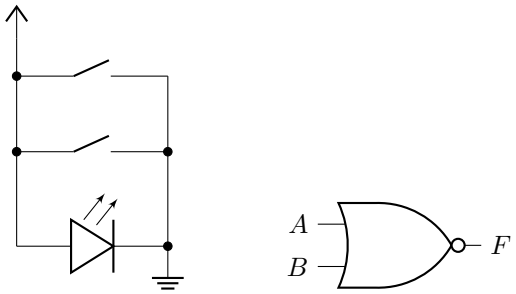
Now that this is a NAND-only circuit (with the exception of the output inverter), we can more easily determine an expression.

$$F = (\{[(A')(BC)']'[(B)(DE)']'\}')'$$

6.2 NOR

A NOR gate is a logic gate that has an output of zero any time one or more inputs is one. It is the logical complement of an OR gate. A 2-input NOR gate is shown in Figure 6.3.

Similarly as with NAND gates, it is important to note that, due to the DeMorgan's law, a NOR gate can



A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Figure 6.3: A NOR gate represented as a switch circuit, a circuit diagram, and a truth table.

be represented in two different forms.

$$\begin{aligned} A \text{ NOR } B &= (A+B)' \\ &= A'B' \end{aligned}$$

This means that a NOR gate can be depicted in three ways: a NOR gate, an OR gate with an inverted output, or an AND gate with inverted inputs. These three NOR symbols are shown in Figure 6.4.

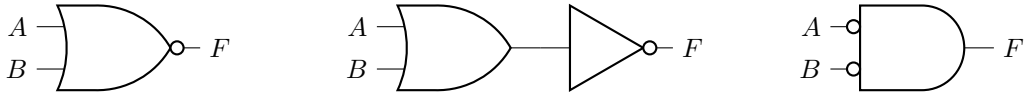


Figure 6.4: A NOR gate represented as a NOR gate, as an OR gate with an inverted output, and as an AND gate with inverted inputs.

6.2.1 NOR-Only Circuits

It is possible to create a NOR-only circuit by either starting with a POS expression and using the double prime method, or by starting with any other type of expression using the double prime or bubble method. (The double prime method is less efficient when the expression is not POS.)

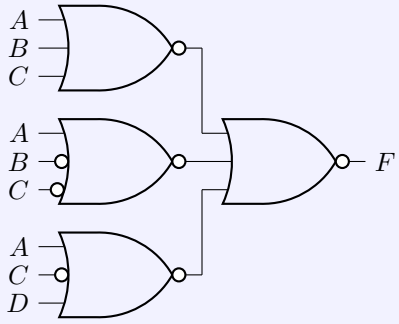
The double prime method is exactly the same as it was with creating a NAND-only circuit.

Example: Using the double-prime method to create a NOR-only circuit

Create a NOR-only expression for $F = (A+B+C)(A+B'+C')(A+C'+D)$.

$$\begin{aligned} F &= (A+B+C)(A+B'+C')(A+C'+D) \\ &= \{[(A+B+C)(A+B'+C')(A+C'+D)]'\}' \\ &= [(A+B+C)' + (A+B'+C')' + (A+C'+D)']' \end{aligned}$$

At this point, every one of the statements is a NOR statement. The circuit diagram is shown below.



Note that if you were to continue distributing the primes and continuing to use the DeMorgan's law, you would start with an OR-AND (POS circuit), obtain a NOR-NOR circuit, then an AND-NOR circuit, a NAND-AND circuit, and then finish back where you started with an OR-AND (POS) circuit.

Just as we saw with NAND gates, the double-prime method can be used on a factored POS expression, but requires more than one step in the process. Double prime the expression, and distribute the inner prime. Any terms that still contain an AND operation need to be double primed, with the inner prime distributed until the expression is NOR-only.

Example: Using the double-prime method to create a NOR-only circuit with a factored POS expression

Create a NOR-only circuit of $F = (A' + C')(A + B' + CD')$.

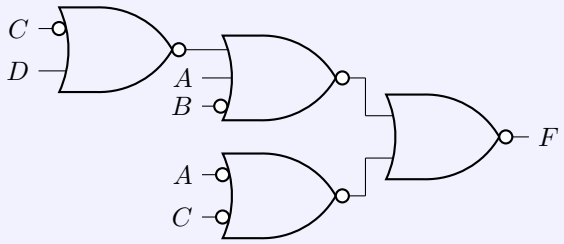
$$\begin{aligned} F &= (A' + C')(A + B' + CD') \\ &= \{[(A' + C')(A + B' + CD')]'\}' \\ &= [(A' + C')' + (A + B' + CD')']' \end{aligned}$$

There is still an AND operation in this expression, so the term CD' itself needs to be double primed, with only the inner prime being distributed.

$$[(CD')']' = (C' + D)'$$

This is now NOR-only, and can get plugged back in to the original expression.

$$\begin{aligned} F &= [(A' + C')' + (A + B' + CD')']' \\ &= [(A' + C')' + (A + B' + (C' + D)')']' \end{aligned}$$

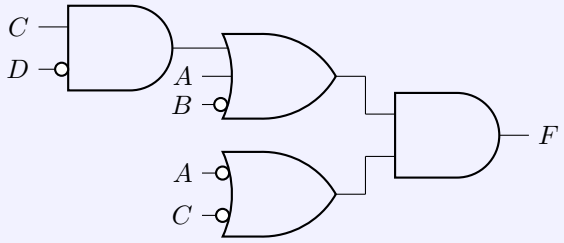


As mentioned, the double-prime method can be used for SOP or other circuits, but is much more complicated, as double-priming a SOP expression leads to a NAND-only circuit. Instead, the bubble method is a more effective approach. The bubble method works in exactly the same way as with NAND gates, except that here we are interested in how to modify AND and OR gates to create NOR gates. As we saw in Figure 6.4 a NOR gate can be created by priming the output of an OR gate, or by priming the inputs of an AND gate.

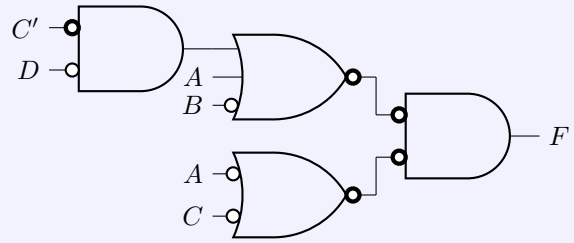
Therefore, start with any AND/OR/NOT circuit. Start all the way on the left-hand side and inspect each logic gate. If the logic gate is an OR gate, bubble the output. A second bubble needs to be placed at the other end of the wire in order to cancel out the first bubble (due to the involution law). Any time an AND gate is encountered, bubble the inputs. Again, every bubble will need to be cancelled out with a second bubble due to the involution law.

Example: Using the bubble method to create a NOR-only circuit with a factored POS expression

Use the bubble method to create a NOR-only circuit of $F = (A' + C')(A + B' + CD')$.



To create a NOR-only circuit, bubble the output of every OR gate and cancel that out with a second bubble somewhere else in that path. Bubble all inputs to every AND gate and cancel that out with a second bubble somewhere else in the path. (Note that in this case, it means bubbling the input to the AND gate from CD' and then priming the C term; the D' term is already primed.) If there is already a bubble on an AND gate input, then we don't have to do anything with that input.

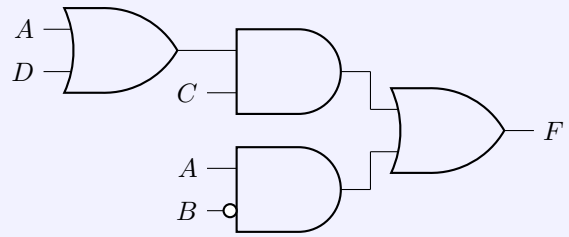


At this point, every logic gate has been converted into a NOR gate. If we were to re-draw it, it would look identical to the NOR-only circuit in the previous example.

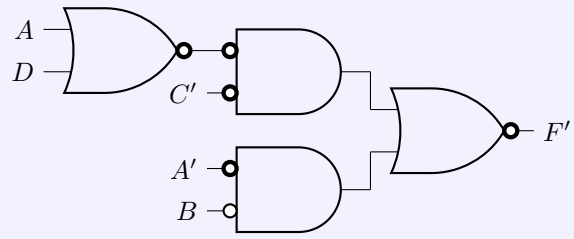
The bubble method is better suited to SOP circuits due to the fact that an OR gate forms the output gate in an SOP expression. When the output of that OR gate is primed, a second prime must be included somewhere along the same path to cancel it out per the involution law. Because there is nothing else to prime, we must include an output inverter.

Example: Using the bubble method to create a NOR-only circuit with an SOP expression

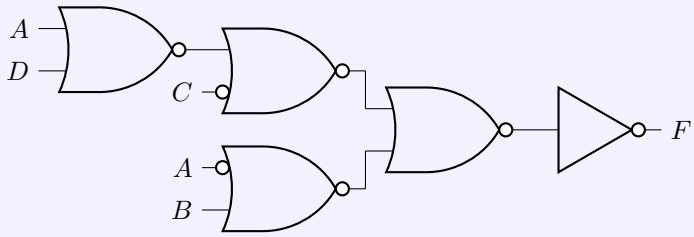
Create a NOR-only circuit for $F = AB' + C(A + D)$. Start by drawing the circuit using AND gates, OR gates, and inverters.



The bubble method is exactly the same regardless of the type of circuit that is started with. To create a NOR gate, bubble all OR gate outputs and include a second bubble elsewhere in the same path. Bubble all AND gate inputs and include a second bubble elsewhere in the same path.



Re-draw this circuit using only NOR gates.



Now that this is a NOR-only circuit (with the exception of the output inverter), we can more easily determine an expression.

$$F = \{([C' + (A + D)']' + [A' + B]')'\}'$$

6.3 Example Problems

NAND-Only Circuits

1. Express $F(A, B, C, D) = \Sigma m(1, 3, 4, 5, 7, 13)$ as a NAND-only circuit.
2. Express $F(A, B, C, D) = BD' + A'C'D + A'B'D + ABC$ as a NAND-only circuit.
3. Express $F(A, B, C, D) = A'B'(C' + D) + AB(C + D)$ as a NAND-only circuit.
4. Express $F(A, B, C, D, E) = A + B'C + B(D + EC')$ as a NAND-only circuit.
5. Express $F(A, B, C, D) = (A + B)(A' + B' + D)(C + D')$ as a NAND-only circuit.

NOR-Only Circuits

1. Express $F(A, B, C) = (A + B' + C)(A' + C')$ as a NOR-only circuit.
2. Express $F(A, B, C, D) = (A + BC)(B + C' + D')$ as a NOR-only circuit.
3. Express $F(A, B, C, D, E) = (A + B'C'D')(B + C + DE)$ as a NOR-only circuit.
4. Express $F(A, B, C) = A'B' + AB$ as a NOR-only circuit.
5. Express $F(A, B, C, D) = A'B'(C' + D) + AB(C + D)$ as a NOR-only circuit.

7 Circuit Optimization

Now that we have a solid grasp of how to create Boolean expressions for a given design problem, and how to minimize them to a reduced form, we will learn how to optimize the circuits for different outcomes. For now, the two things we may choose to optimize would be time and cost.

First: a note about inverters. We will see later in this class that there are certain types of logic devices that give us access to signals and their complements for no extra penalty in time or cost. Therefore, we will not consider inverters to be penalizing factors in our discussion of timing and cost in optimizing circuits.

7.1 Two-Level Circuits: Least Time Implementation

So far, we have mostly considered two-level circuits. These include AND-OR (SOP) circuits and OR-AND (POS) circuits. The term "two-level" means that the maximum number of logic gates that any signal could travel through is two. We have seen that using minimization strategies such as k-maps and Quine-McCluskey, it is relatively straightforward to create minimum two-level circuits. Two-level circuits also have the advantage of being the fastest type of circuit, as adding on any more levels will cause the signals to incur more of a delay from input to output.

We will see in chapter 9 (timing) that two-level circuits will also have an advantage (if they are designed accordingly) of not allowing any hazards (or signal glitches) to occur on the output.

7.2 Finding Least Cost Implementation

Sometimes, a two-level circuit is also the cheapest implementation of a circuit. Other times, it is not. To determine what a low-cost circuit means, we need to understand what causes a circuit to cost money. Those things are the number of logic gates in a circuit, and the number of inputs that are used on each gate.

As of October 15, 2020, Digikey sold the following logic gates from Texas Instruments in a DIP format for the following prices, leading to the individual logic gate costs that follow.

- 74LS00 – Quad 2-input NAND gate – \$0.64 per chip – \$0.16 per gate
- 74LS10 – Triple 3-input NAND gate – \$0.79 per chip – \$0.26 per gate
- 74LS13 – Dual 4-input NAND gate – \$0.88 per chip – \$0.44 per gate
- 74LS30 – Single 8-input NAND gate – \$0.98 per chip – \$0.98 per gate

Therefore, from a cost perspective, we want to minimize the number or logic gates and inputs as much as possible. The first step is finding minimum SOP and POS expressions, as those are the lowest-cost two-level implementations of a given circuit.

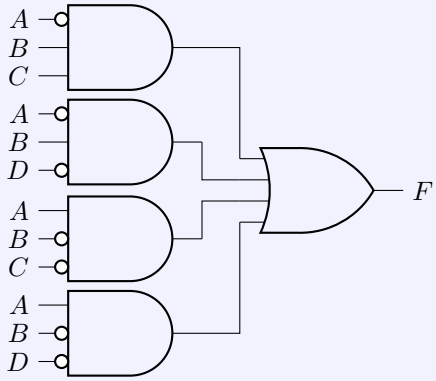
Next is looking at a factored SOP and factored POS implementation, and seeing if those lead to a lower-cost circuit. The process for finding the lowest cost circuit is therefore to create each of the following circuits, and compare the gate/input cost for each: SOP, factored SOP, POS, and factored POS.

Example: Find the lowest-cost circuit

Find the lowest-cost circuit that corresponds to $F(A, B, C, D) = \Sigma m(4, 6, 7, 8, 9, 10)$. First: using any technique (Boolean algebra, k-map, Quine-McCluskey), find an SOP expression.

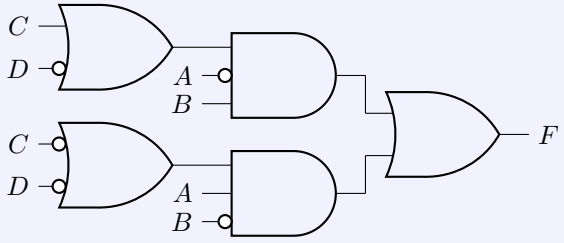
$$F_{SOP}(A, B, C, D) = A'BC + A'BD' + AB'C' + AB'D'$$

Count the number of gates and inputs. There are five gates and sixteen inputs. Using the logic gate costs from above, this circuit would cost \$1.48.



Factor the SOP expression and see if that leads to a cost reduction. When factoring, attempt to factor out the maximum number of terms possible from each expression. There are now five gates but only twelve inputs. If we use the gate costs quoted earlier, this circuit would only cost \$1.00.

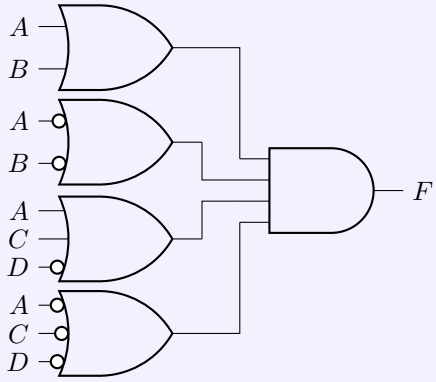
$$F(A, B, C, D) = A'B(C + D') + AB'(C' + D')$$



Check both POS and factored POS to see if the cost can be further reduced. A POS implementation requires five gates and fourteen inputs. It would cost \$1.28. While it is better than SOP, it is

still not better than a factored SOP expression.

$$F_{POS}(A,B,C,D) = (A+B)(A'+B')(A+C+D')(A'+C'+D')$$



Factoring the POS expression would not be helpful in this example.

If the goal is to create a NAND-only or NOR-only lowest-cost circuit, the process is basically the same. The first part is to determine which of the four types of circuits has the fewest gates and inputs. Then it can be converted into a NAND-only or NOR-only circuit. Consideration may be taken into whether or not an output inverter would be present on the output if you have the flexibility to decide between NAND-only or NOR-only or an SOP/factored SOP or POS/factored POS implementation.

Example: Find the lowest-cost NAND-only circuit

Find the lowest cost NAND-only circuit that corresponds to $F(A,B,C,D) = \Sigma m(0,2,3,6,8,9,12,13)$. First find the SOP expression.

$$F_{SOP}(A,B,C,D) = AC' + A'B'D' + A'B'C + A'CD'$$

There are five gates and fifteen inputs. This circuit would cost \$1.38 using the prices quoted above. The next step is to factor the SOP expression.

$$F(A,B,C,D) = AC' + A'B'(D'+C) + A'CD'$$

This implementation requires five gates and thirteen inputs and would cost \$1.10. Next, find a POS expression.

$$F_{POS}(A,B,C,D) = (A'+C')(A+C+D')(A+B'+C)(A+B'+D')$$

This implementation requires five gates and fifteen inputs, and would cost the same amount as an SOP implementation. Finally, find a factored POS circuit.

$$F(A,B,C,D) = (A'+C')(A+C+D')(A+B'+CD')$$

This implementation requires five gates and thirteen inputs, and has the same cost as the factored SOP circuit. The lowest-cost circuit is a tie between factored SOP and factored POS. However, because we specifically want a NAND-only circuit, consider that the factored POS expression would require an output inverter, which would add to the cost of the circuit. Therefore, the factored SOP circuit is the preferred lowest-cost NAND-only circuit.

$$F(A,B,C,D) = [(AC')'(A'CD')'(A'B'(DC')')']'$$

7.3 Optimization of Multiple Output Circuits

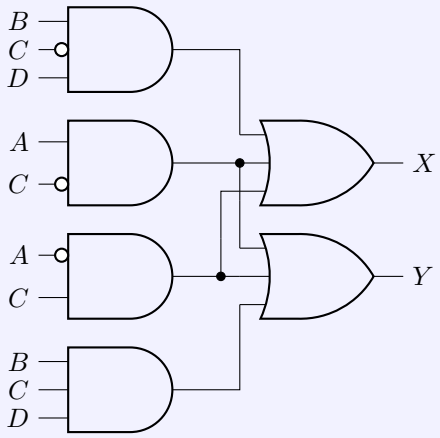
It can be difficult to design circuits when we have multiple outputs, or functions. If possible, we want to share logic gates in order to use the fewest number of logic gates and inputs.

Example: Optimizing functions that are already optimized

In this example, we find that the minimum SOP expressions have terms that can be shared between the two outputs without having to do any specific optimization steps.

$$\begin{aligned} X(A,B,C,D) &= \Sigma m(2,3,5,6,7,8,9,12,13) \\ &= AC' + A'C + BC'D \\ Y(A,B,C,D) &= \Sigma m(2,3,6,7,8,9,12,13,15) \\ &= AC' + A'C + ABD \end{aligned}$$

It would require eight logic gates and twenty inputs to build X and Y individually. However, AC' and $A'C$ are shared terms so X and Y can be implemented together.



This optimized circuit only requires six gates and sixteen inputs. Because we shared the terms that were the same between each function, we were able to save two logic gates and four inputs, creating a circuit that is easier to build and costs less money.

Not all circuits have minimum expressions that lend themselves so easily to sharing gates, however. In those cases, the procedure for optimizing two outputs simultaneously follows.

1. Identify the minterms that are NOT shared between the two functions.
2. The unshared minterms must be looped in prime implicants. Loop these first. During this looping process, attempt to maximize the number of shared minterms that remain.
3. Any minterms that remain are shared. Loop these, not necessarily in prime implicants, to create shared functions.

Example: Optimizing functions that are not already optimized

Find the lowest-cost implementation of the following functions.

$$X(A,B,C,D) = \Sigma m(1,3,5,6,7,13,15)$$
$$Y(A,B,C,D) = \Sigma m(1,3,6,9,11,13,15)$$

It can be helpful to create k-maps for each expression. The k-map for X is shown below. Minterms m_5 and m_7 are unshared and are shown in red.

CD \ AB	00	01	11	10
00	0	0	0	0
01	1	1	1	0
11	1	1	1	0
10	0	1	0	0

The k-map for Y is shown below. Minterms m_9 and m_{11} are unshared and are shown in red.

CD \ AB	00	01	11	10
00	0	0	0	0
01	1	0	1	1
11	1	0	1	1
10	0	1	0	0

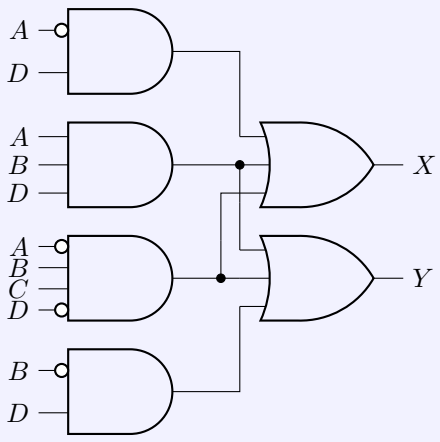
Starting with X, we see that there are two options for looping m_5 and m_7 in prime implicants. Either we could loop $m_1 - m_3 - m_5 - m_7$ or we could loop $m_5 - m_7 - m_{13} - m_{15}$. In this case it does not matter which is chosen. We will start by looping $m_1 - m_3 - m_5 - m_7$ and will show what happens with the second scenario later in this example. We can re-draw the k-map for Y with m_1 and m_3 drawn in blue to show which minterms have been used up in X and can no longer be shared with Y.

CD \ AB	00	01	11	10
00	0	0	0	0
01	1	0	1	1
11	1	0	1	1
10	0	1	0	0

We need to loop the unshared minterms in Y with a prime implicant. The only prime implicant that makes sense is $m_1 - m_3 - m_9 - m_{11}$ as this maximizes the number of shared minterms that remain.

The remaining minterms are m_6, m_{13} and m_{15}. These can be looped in implicants: m_6 by itself and $m_{13} - m15$. The expressions for X and Y therefore become

$$X(A,B,C,D) = A'D + ABD + A'BCD'$$
$$Y(A,B,C,D) = B'D + ABD + A'BCD'$$



This implementation uses six gates and seventeen inputs, which saves two gates and four inputs from just using minimum SOP implementations. At this point you may be wondering what would have happened if we had chosen to loop $m_5 - m_7 - m_{13} - m_{15}$ in X. In that case, minterms m_{13} and m_{15} would no longer be shared with Y. The prime implicant in Y would then be $m_9 - m_{11} - m_{13} - m_{15}$.

The two expressions are given below.

$$X(A,B,C,D) = BD + A'B'D + A'BCD'$$
$$Y(A,B,C,D) = AD + A'B'D + A'BCD'$$

Either way, six gates and seventeen inputs are used. Both options are the lowest-cost implementations of X and Y.

7.3.1 Multiple Output Circuits Using Prime Implicant Tables

It is possible to derive a lowest-cost implementation of multiple functions by using a prime implicant table. The following procedure is used.

1. Identify the minterms (or maxterms, if POS is desired) of each function.
2. Find all prime implicants of each function.
3. Find all prime implicants of each of the product functions (or sum functions, if POS is desired).
4. Create a prime implicant table including PI's of **all** functions and product/sum functions.
   (a) Identify the essential prime implicants of each function and include them in the corresponding function output(s).
   (b) Optimally assign the remaining prime implicants.

In this method, it is not sufficient to simply find the prime implicants of each function, it is also necessary to find the prime implicants of the product functions (if using SOP) or sum functions (if using POS). For example, if two functions $X(A,B,C,D)$ and $Y(A,B,C,D)$ are defined, the product function is as follows.

$$XY(A,B,C,D) = X(A,B,C,D)Y(A,B,C,D)$$

If three functions $X(A,B,C,D)$, $Y(A,B,C,D)$, and $Z(A,B,C,D)$ are defined, the product functions are as follows.

$$XY(A,B,C,D) = X(A,B,C,D)Y(A,B,C,D)$$
$$XZ(A,B,C,D) = X(A,B,C,D)Z(A,B,C,D)$$
$$YZ(A,B,C,D) = Y(A,B,C,D)Z(A,B,C,D)$$
$$XYZ(A,B,C,D) = X(A,B,C,D)Y(A,B,C,D)Z(A,B,C,D)$$

When minterm expressions are joined together in a product function, only the shared minterms are included in the output.

Example: Product functions of three functions

Find all product functions of each of the following functions.

$$X(A,B,C,D) = \Sigma m(1,5,13,15)$$
$$Y(A,B,C,D) = \Sigma m(0,2,5,8,10,13,15)$$
$$Z(A,B,C,D) = \Sigma m(0,1,2,8,10,15)$$

The product function of $X(A,B,C,D)$ and $Y(A,B,C,D)$ contains only their shared minterms: 5, 13, and 15.

$$XY(A,B,C,D) = \Sigma m(5,13,15)$$

The product function of $X(A,B,C,D)$ and $Z(A,B,C,D)$ contains only their shared minterms: 1, and 15.

$$XZ(A,B,C,D) = \Sigma m(1,15)$$

The product function of $Y(A,B,C,D)$ and $Z(A,B,C,D)$ contains only their shared minterms: 0, 2, 8, 10, and 15.

$$YZ(A,B,C,D) = \Sigma m(0,2,8,10,15)$$

The product function of all three functions contains only minterms shared between all three functions. In this case, only minterm 15 is shared.

$$XYZ(A,B,C,D) = \Sigma m(15)$$

Alternatively, if two functions $X(A,B,C,D)$ and $Y(A,B,C,D)$ are defined, the sum function is as follows.

$$X + Y(A,B,C,D) = X(A,B,C,D) + Y(A,B,C,D)$$

If three functions $X(A,B,C,D)$, $Y(A,B,C,D)$, and $Z(A,B,C,D)$ are defined, the sum functions are as

follows.

$$X + Y(A, B, C, D) = X(A, B, C, D) + Y(A, B, C, D)$$
$$X + Z(A, B, C, D) = X(A, B, C, D) + Z(A, B, C, D)$$
$$Y + Z(A, B, C, D) = Y(A, B, C, D) + Z(A, B, C, D)$$
$$X + Y + Z(A, B, C, D) = X(A, B, C, D) + Y(A, B, C, D) + Z(A, B, C, D)$$

When maxterm expressions are joined together in a sum function, only the shared maxterms are included in the output.

Example: Sum functions of three functions

Find all sum functions of each of the following functions.

$$X(A, B, C, D) = \Pi M(0, 2, 3, 4, 6, 9, 10, 11)$$
$$Y(A, B, C, D) = \Pi M(2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 15)$$
$$Z(A, B, C, D) = \Pi M(1, 4, 5, 7, 12, 13, 15)$$

The sum function of $X(A, B, C, D)$ and $Y(A, B, C, D)$ contains only their shared maxterms: 2, 3, 4, 6, 9, 10, and 11.

$$X + Y(A, B, C, D) = \Pi M(2, 3, 4, 6, 9, 10, 11)$$

The sum function of $X(A, B, C, D)$ and $Z(A, B, C, D)$ contains only their shared maxterm: 4.

$$X + Z(A, B, C, D) = \Pi M(4)$$

The sum function of $Y(A, B, C, D)$ and $Z(A, B, C, D)$ contains only their shared maxterms: 4, 7, 12, 13, and 15.

$$Y + Z(A, B, C, D) = \Pi M(4, 7, 12, 13, 15)$$

The sum function of all three functions contains only maxterms shared between all three functions. In this case, only maxterm 4 is shared.

$$X + Y + Z(A, B, C, D) = \Pi M(4)$$

A prime implicant table of two or more functions will contain all minterms/maxterms of each function separately, and contain every prime implicant (those of the functions themselves, as well as those of the

product/sum functions). Each prime implicant included within a function will be indicated with an $\times$ in a function's minterm/maxterm columns.

Example: Multiple output SOP implementation of three functions

Find the lowest-cost SOP implementation of the following functions.

$$X(A,B,C,D) = \Sigma m(1,5,13,15)$$
$$Y(A,B,C,D) = \Sigma m(0,2,5,8,10,13,15)$$
$$Z(A,B,C,D) = \Sigma m(0,1,2,8,10,15)$$

The first step is to identify the prime implicants of each function.

PIs of X	**PIs of Y**	**PIs of Z**
$A'C'D$ (1–5)	$B'D'$ (0–2–8–10)	$B'D'$ (0–2–8–10)
$BC'D$ (5–13)	$BC'D$ (5–13)	$A'B'C'$ (0–1)
ABD (13–15)	ABD (13–15)	$ABCD$ (15)

Then, find all product functions.

$$XY(A,B,C,D) = \Sigma m(5,13,15)$$
$$XZ(A,B,C,D) = \Sigma m(1,15)$$
$$YZ(A,B,C,D) = \Sigma m(0,2,8,10,15)$$
$$XYZ(A,B,C,D) = \Sigma m(15)$$

Identify the prime implicants of each product function.

PIs of XY	**PIs of XZ**	**PIs of YZ**	**PIs of XYZ**
$BC'D$ (5–13)	$A'B'C'D$ (1)	$B'D'$ (0–2–8–10)	$ABCD$ (15)
ABD (13–15)	$ABCD$ (15)	$ABCD$ (15)	

Create a PI table including each of the three functions. Redundant PIs do not need to be included more than once. (For example, $ABCD$ is only included once on the PI table, even though it is a prime implicant of Z, XZ, YZ, and XYZ.)

	X				Y							Z					
PI	**1**	**5**	**13**	**15**	**0**	**2**	**5**	**8**	**10**	**13**	**15**	**0**	**1**	**2**	**8**	**10**	**15**
1–5	×	×															
5–13		×	×				×			×							
13–15			×	×						×	×						
0–2–8–10					×	×		×	×			×		×	×	×	
0–1												×	×				
1	×												×				
15				×							×						×

The functions $Y(A, B, C, D)$ and $Z(A, B, C, D)$ have minterms only covered by a single prime implicant. These point to the essential prime implicants of the functions. 0–2–8–10 ($B'D'$) must be included in functions Y and Z. 5–13 ($BC'D$) must be included in Y, and, although it is not essential in X, will be included in X as well. 15 ($ABCD$) is essential in Z, and will be included in all three functions.

The remaining PIs and uncovered minterms can then be analyzed to optimally choose the remaining (non-essential) PIs.

	X	Y	Z
PI	**1**	**–**	**1**
1–5 (A'C'D)	×		
13–15 (ABD)			
0–1 (A'B'C')			×
1 (A'B'C'D)	×		×

The most optimal choice of PIs remaining is 1 ($A'B'C'D$), as that is the only PI that can be used in both functions X and Z. The final expressions are as follows.

$$
\begin{aligned}
X(A, B, C, D) &= \quad BC'D + ABCD + A'B'C'D \\
Y(A, B, C, D) &= B'D' + BC'D + ABCD \\
Z(A, B, C, D) &= B'D' \quad + ABCD + A'B'C'D
\end{aligned}
$$

This optimized implementation uses four AND gates and three OR gates, which saves two AND gates over the individually optimized expressions (solving each function as minimum SOP individually).

Example: Multiple output POS implementation of three functions

Find the lowest-cost POS implementation of the following functions.

$$X(A,B,C,D) = \Pi M(1,5,7,8,12,13,14,15)$$
$$Y(A,B,C,D) = \Pi M(0,1,5,8,14)$$
$$Z(A,B,C,D) = \Pi M(0,2,3,6,8,9,10,11,14)$$

The first step is to identify the prime implicants of each function.

PIs of X	**PIs of Y**	**PIs of Z**
$A'+B'$ (12–13–14–15)	$B+C+D$ (0–8)	$A'+B$ (8–9–10–11)
$B'+D'$ (5–7–13–15)	$A+B+C$ (0–1)	$C'+D$ (2–6–10–14)
$A+C+D'$ (1–5)	$A+C+D'$ (1–5)	$B+C'$ (2–3–10–11)
$A'+C+D$ (8–12)	$A'+B'+C'+D$ (14)	$B+D$ (0–2–8–10)

Then, find all sum functions.

$$X+Y(A,B,C,D) = \Pi M(1,5,8,14)$$
$$X+Z(A,B,C,D) = \Pi M(8,14)$$
$$Y+Z(A,B,C,D) = \Pi M(0,8,14)$$
$$X+Y+Z(A,B,C,D) = \Pi M(8,14)$$

Identify the prime implicants of each sum function.

PIs of X + Y	**PIs of X + Z**	**PIs of Y + Z**
$A+C+D'$ (1–5)	$A'+B+C+D$ (8)	$B+C+D$ (0–8)
$A'+B+C+D$ (8)	$A'+B'+C'+D$ (14)	$A'+B'+C'+D$ (14)
$A'+B'+C'+D$ (14)		

PIs of X + Y + Z
$A'+B+C+D$ (8)
$A'+B'+C'+D$ (14)

Create a PI table including each of the three functions. Redundant PIs do not need to be included more than once.

	X								Y				
PI	**1**	**5**	**7**	**8**	**12**	**13**	**14**	**15**	**0**	**1**	**5**	**8**	**14**
12–13–14–15					×	×	×	×					
5–7–13–15		×	×			×		×					
1–5	×	×								×	×		
8–12				×	×								
0–8									×			×	
0–1									×	×			
14							×						×
8–9–10–11													
2–6–10–14													
2–3–10–11													
0–2–8–10													
8				×								×	

	Z								
PI	**0**	**2**	**3**	**6**	**8**	**9**	**10**	**11**	**14**
12–13–14–15									
5–7–13–15									
1–5									
8–12									
0–8	×				×				
0–1									
14									×
8–9–10–11					×	×	×	×	
2–6–10–14		×		×			×		×
2–3–10–11		×	×				×	×	
0–2–8–10	×	×			×		×		
8					×				

All three functions have maxterms only covered by a single prime implicant. These point to the essential prime implicants of the functions. 5–7–13–15 ($B' + D'$) is an EPI of X. 1–5 ($A + C + D'$) is an EPI of X and Y. 14 ($A' + B' + C' + D$) is an EPI of Y and will be included in X and Y. 2–3–10–11 ($B + C'$) is an EPI of Z. 2–6–10–14 ($C' + D$) is an EPI of Z. 8–9–10–11 ($A' + B$) is an EPI of Z.

The remaining PIs and uncovered maxterms can then be analyzed to optimally chose the remaining

(non-essential) PIs.

	X		Y		Z
PI	**8**	**12**	**0**	**8**	**0**
12–13–14–15		×			
8–12	×	×			
0–8			×	×	×
0–1			×		
14					
0–2–8–10					×
8	×			×	

The most optimal choice of PIs remaining is 0–8 $(B+C+D)$ for Y and Z, and 8–12 $(A'+C+D)$ for X. The final expressions are as follows.

$$X = (A+C+D')(B'+D')(A'+B'+C'+D) \qquad (A'+C+D)$$

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

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

This optimized implementation uses eight OR gates and three AND gates, which saves three OR gates over the individually optimized expressions (solving each function as minimum POS individually).

7.3.2 Finding Multiple Output Implicants Using the Quine-McCluskey Method

Finding prime implicants of all functions and product (or sum) functions can be difficult at times. The Quine-McCluskey method is well-suited to finding all prime implicants of functions and product functions. (This textbook does not explicitly cover Quine-McCluskey for POS expressions, although it can be done using the same methodology as SOP.) The Quine-McCluskey method is used with a few differences from the single-function method.

For each column, include a section with each function. For each implicant included in that column, put a 0 under each function that is **not** covered by the implicant. Put a – under each function that **is** covered by the implicant. When comparing from group to group to the next column, AND together the 0s to determine where to place 0's and –'s in the next column. If a new 0 gets included when combining implicants into the next column, any implicant with 0's not in those new places will **not** get a checkmark. Any implicants with 0's in all function columns can be ignored when combining implicants into the next column.

Example: Comparing implicants in groups into the next column

Compare implicants in column 1, groups 1 and 2, into column 2.

$$X(A,B,C,D) = \Sigma m(3,4,6,9)$$
$$Y(A,B,C,D) = \Sigma m(2,4,8)$$
$$Z(A,B,C,D) = \Sigma m(3,6,10)$$

Fill out column 1. Place a 0 in any function where the minterm does not appear, and a – in any function where the minterm does appear.

Column 1		X	Y	Z	Column 2	X	Y	Z
one	2. 0010	0	–	0				
	4. 0100	–	–	0				
	8. 1000	0	–	0				
two	3. 0011	–	0	–				
	6. 0110	–	0	–				
	9. 1001	–	0	0				
	10. 1010	–	–	0				

Compare minterms in group one to minterms in group two to move into the next column. 2–3 and 2–6 are not an implicant of any function (0's in all three function columns) and can be ignored. When 2–10 are combined, 2 receives a checkmark because there are no new 0's moving from 2 to 2–10. However, 10 does not receive a checkmark as there is a new 0 going from 10 to 2–10. When 4–6 are combined, neither 4 nor 6 receives a checkmark. 8–9 is not an implicant of any function and can be ignored. When 8–10 are combined, 8 receives a checkmark and 10 does not.

Column 1		X	Y	Z	Column 2	X	Y	Z
one	2. 0010 ✓	0	–	0	~~2–3. 001–~~	0	0	0
	4. 0100	–	–	0	~~2–6. 0–10~~	0	0	0
	8. 1000 ✓	0	–	0	2–10. –010	0	–	0
two	3. 0011	–	0	–	4–6. 01–0	–	0	0
	6. 0110	–	0	–	~~8–9. 100–~~	0	0	0
	9. 1001	–	0	0	8–10. 10–0	0	–	0
	10. 1010	–	–	0				

There are 8 prime implicants.

PIs	Included in X?	Included in Y?	Included in Z?
$A'BC'D'$ (4)	yes	yes	no
$A'B'CD$ (3)	yes	no	yes
$A'BCD'$ (6)	yes	no	yes
$AB'C'D$ (9)	yes	no	no
$AB'CD'$ (10)	yes	yes	no
$B'CD'$ (2–10)	no	yes	no
$A'BD'$ (4–6)	yes	no	no
$AB'D'$ (8–10)	no	yes	no

After all groups have been compared, any implicant without a checkmark is a prime implicant (either of a function, or of a product function). The functions and product functions of each PI are indicated by the presence of a – in the section with each function. After identifying all of the PIs, a prime implicant table can be created to find the optimum implementation of each function, as described in the preceding section.

Example: Multiple-output Quine-McCluskey

Find the lowest-cost SOP implementation of the following functions.

$$X(A,B,C,D) = \Sigma m(0,1,2,5,7,8,10,12,13,14,15)$$
$$Y(A,B,C,D) = \Sigma m(0,1,3,5,6,7,9,13,14,15)$$
$$Z(A,B,C,D) = \Sigma m(0,1,2,4,5,8,9,12,13,14,15)$$

Perform the Quine-McCluskey method.

Column 1		X	Y	Z	Column 2	X	Y	Z	Column 3	X	Y	Z
zero	0. 0000	–	–	–								
one	1. 0001	–	–	–								
	2. 0010	–	0	–								
	4. 0100	0	0	–								
	8. 1000	–	0	–								
two	3. 0011	0	–	0								
	5. 0101	–	–	–								
	6. 0110	0	–	0								
	9. 1001	0	–	–								
	10. 1010	–	0	0								
	12. 1100	–	0	–								
three	7. 0111	–	–	0								
	13. 1101	–	–	–								
	14. 1110	–	–	–								
four	15. 1111	–	–	–								

Compare column one minterms between groups to populate column two. Note that a few implicants are not included in any function and can be ignored moving forward. There are no prime implicants in column one.

Column 1		X	Y	Z	Column 2	X	Y	Z	Column 3	X	Y	Z
zero	0. 0000 ✓	–	–	–	0–1. 000–	–	–	–				
one	1. 0001 ✓	–	–	–	0–2. 00–0	–	0	–				
	2. 0010 ✓	–	0	–	0–4. 0–00	0	0	–				
	4. 0100 ✓	0	0	–	0–8. –000	–	0	–				
	8. 1000 ✓	–	0	–	1–3. 00–1	0	–	0				
two	3. 0011 ✓	0	–	0	1–5. 0–01	–	–	–				
	5. 0101 ✓	–	–	–	1–9. –001	0	–	–				
	6. 0110 ✓	0	–	0	~~2–3~~	0	0	0				
	9. 1001 ✓	0	–	–	~~2–6~~	0	0	0				
	10. 1010 ✓	–	0	0	2–10. –010	–	0	0				
	12. 1100 ✓	–	0	–	4–5. 010–	0	0	–				
three	7. 0111 ✓	–	–	0	~~4–6~~	0	0	0				
	13. 1101 ✓	–	–	–	4–12. –100	0	0	–				

	14. 1110 ✓	–	–	–	8–9. 100–	0	0	–
four	15. 1111 ✓	–	–	–	8–10. 10–0	–	0	0
					8–12. 1–00	–	0	–
					3–7. 0–11	0	–	0
					5–7. 01–1	–	–	0
					5–13. –101	–	–	–
					6–7. 011–	0	–	0
					6–14. –110	0	–	0
					9–13. 1–01	0	–	–
					10–14. 1–10	–	0	0
					12–13. 110–	–	0	–
					12–14. 11–0	–	0	–
					7–15. –111	–	–	0
					13–15. 11–1	–	–	–
					14–15. 111–	–	–	–

Compare column two implicants between groups to populate column three. All ignored implicants from the previous step have been removed from this table. Note that a few implicants are not included in any function and can be ignored moving forward. (There are also some redundant implicants created in column 3 that are crossed out.) There are seven prime implicants in column two.

Column 1		X	Y	Z	Column 2	X	Y	Z	Column 3	X	Y	Z
zero	0. 0000 ✓	–	–	–	0–1. 000–	–	–	–	0–1–4–5. 0–0–	0	0	–
one	1. 0001 ✓	–	–	–	0–2. 00–0	–	0	–	0–1–8–9. –00–	0	0	–
	2. 0010 ✓	–	0	–	0–4. 0–00 ✓	0	0	–	~~0–2–1–3~~	0	0	0
	4. 0100 ✓	0	0	–	0–8. –000	–	0	–	0–2–8–10. –0–0	–	0	0
	8. 1000 ✓	–	0	–	1–3. 00–1 ✓	0	–	0	~~0–4–1–5~~			
two	3. 0011 ✓	0	–	0	1–5. 0–01	–	–	–	0–4–8–12. – –00	0	0	–
	5. 0101 ✓	–	–	–	1–9. –001 ✓	0	–	–	~~0–8–1–9~~			
	6. 0110 ✓	0	–	0	2–10. –010 ✓	–	0	0	~~0–8–2–10~~			
	9. 1001 ✓	0	–	–	4–5. 010– ✓	0	0	–	~~0–8–4–12~~			
	10. 1010 ✓	–	0	0	4–12. –100 ✓	0	0	–	1–3–5–7. 0– –1	0	–	0
	12. 1100 ✓	–	0	–	8–9. 100– ✓	0	0	–	~~1–5–3–7~~			
three	7. 0111 ✓	–	–	0	8–10. 10–0 ✓	–	0	0	1–5–9–13. – –01	0	–	–
	13. 1101 ✓	–	–	–	8–12. 1–00	–	0	–	~~1–9–5–13~~			
	14. 1110 ✓	–	–	–	3–7. 0–11 ✓	0	–	0	~~2–10–6–14~~	0	0	0
four	15. 1111 ✓	–	–	–	5–7. 01–1 ✓	–	–	0	~~4–5–6–7~~	0	0	0
					5–13. –101 ✓	–	–	–	4–5–12–13. –10–	0	0	–
					6–7. 011– ✓	0	–	0	~~4–12–5–13~~			
					6–14. –110 ✓	0	–	0	~~4–12–6–14~~	0	0	0
					9–13. 1–01 ✓	0	–	–	8–9–12–13. 1–0–	0	0	–
					10–14. 1–10 ✓	–	0	0	8–10–12–14. 1– –0	–	0	0
					12–13. 110– ✓	–	0	–	~~8–12–9–13~~			
					12–14. 11–0 ✓	–	0	–	~~8–12–10–14~~			
					7–15. –111 ✓	–	–	0	5–7–13–15. –1–1	–	–	0
					13–15. 11–1	–	–	–	~~5–13–7–15~~			
					14–15. 111–	–	–	–	6–7–14–15. –11–	0	–	0
									~~6–14–7–15~~			
									12–13–14–15. 11– –	–	0	–
									~~12–14–13–15~~			

Compare column three implicants between groups to populate column four. (Note: column 1 is no longer shown to save space.) All ignored implicants from the previous step have been removed from this table. Note that a few implicants are not included in any function and can be ignored moving forward. (There are also some redundant implicants created in column 4 that are crossed out.) There are seven prime implicants in column three.

Column 2	X	Y	Z
0–1. 000–	–	–	–
0–2. 00–0	–	0	–
0–4. 0–00 ✓	0	0	–
0–8. –000	–	0	–
1–3. 00–1 ✓	0	–	0
1–5. 0–01	–	–	–
1–9. –001 ✓	0	–	–
2–10. –010 ✓	–	0	0
4–5. 010– ✓	0	0	–
4–12. –100 ✓	0	0	–
8–9. 100– ✓	0	0	–
8–10. 10–0 ✓	–	0	0
8–12. 1–00	–	0	–
3–7. 0–11 ✓	0	–	0
5–7. 01–1 ✓	–	–	0
5–13. –101 ✓	–	–	–
6–7. 011– ✓	0	–	0
6–14. –110 ✓	0	–	0
9–13. 1–01 ✓	0	–	–
10–14. 1–10 ✓	–	0	0
12–13. 110– ✓	–	0	–
12–14. 11–0 ✓	–	0	–
7–15. –111 ✓	–	–	0
13–15. 11–1	–	–	–
14–15. 111–	–	–	–

Column 3	X	Y	Z
0–1–4–5. 0–0– ✓	0	0	–
0–1–8–9. –00– ✓	0	0	–
0–2–8–10. –0–0	–	0	0
0–4–8–12. – –00 ✓	0	0	–
1–3–5–7. 0– –1	0	–	0
1–5–9–13. – –01	0	–	–
~~4–5–12–13. –10– ✓~~	0	0	–
~~8–9–12–13. 1–0– ✓~~	0	0	–
8–10–12–14. 1– –0	–	0	0
5–7–13–15. –1–1	–	–	0
6–7–14–15. –11–	0	–	0
12–13–14–15. 11– –	–	0	–

Column 4	X	Y	Z
0–1–4–5–			
8–9–12–13	0	0	–
~~0–1–8–9–~~			
~~4–5–12–13~~			
~~0–4–8–12–~~			
~~1–5–9–13~~			
~~4–5–12–13–~~			
~~6–7–14–15~~	0	0	0

There are fifteen prime implicants in total. Use a prime implicant table to determine the optimum combination of PIs for each function.

	X										
PI	**0**	**1**	**2**	**5**	**7**	**8**	**10**	**12**	**13**	**14**	**15**
0–1	×	×									
0–2	×		×								
0–8	×					×					
1–5		×		×							
8–12						×		×			
13–15									×		×
14–15										×	×
0–2–8–10	×		×			×	×				
1–3–5–7											
1–5–9–13											
8–10–12–14						×	×	×		×	
5–7–13–15				×	×				×		×
6–7–14–15											
12–13–14–15								×	×	×	×
0–1–4–5–											
8–9–12–13											

	Y									
PI	**0**	**1**	**3**	**5**	**6**	**7**	**9**	**13**	**14**	**15**
0–1	×	×								
0–2										
0–8										
1–5		×		×						
8–12										
13–15								×		×
14–15									×	×
0–2–8–10										
1–3–5–7		×	×	×		×				
1–5–9–13										
8–10–12–14										
5–7–13–15				×		×		×		×
6–7–14–15					×	×			×	×
12–13–14–15										
0–1–4–5–										
8–9–12–13										

	Z										
PI	**0**	**1**	**2**	**4**	**5**	**8**	**9**	**12**	**13**	**14**	**15**
0–1	×	×									
0–2	×		×								
0–8	×					×					
1–5		×			×						
8–12						×		×			
13–15									×		×
14–15										×	×
0–2–8–10											
1–3–5–7											
1–5–9–13		×			×		×		×		
8–10–12–14											
5–7–13–15											
6–7–14–15											
12–13–14–15								×	×	×	×
0–1–4–5–	×	×		×	×	×	×	×	×		
8–9–12–13											

All three functions have minterms only covered by a single prime implicant. These point to the essential prime implicants of the functions. 0–1 is an EPI in Y and will also be included in X (while it can be included in Z, an EPI of Z makes this term redundant). 0–2 is an EPI of Z (while it can be included in X, there is a more efficient PI that can be looped in X instead). 1–3–5–7 is an EPI of Y. 1–5–9–13 is an EPI of Y (it will not be included in Z as an EPI of Z will make it redundant). 5–7–13–15 is an EPI of X and will also be included in Y. 6–7–14–15 is an EPI of Y. 0–1–4–5–8–9–12–13 is an EPI of Z.

The remaining PIs and uncovered minterms can then be analyzed to optimally chose the remaining (non-essential) PIs.

	X					Z	
PI	**2**	**8**	**10**	**12**	**14**	**14**	**15**
0–8		×					
1–5							
8–12		×		×			
13–15							×
14–15					×	×	×
0–2–8–10	×	×	×				
8–10–12–14		×	×	×	×		
12–13–14–15				×	×	×	×

At this point, the most efficient selection of remaining PI's is 12–13–14–15 for both X and Z. Then 0–2–8–10 will complete X. The final expressions are as follows.

$$X(A,B,C,D) = BD + A'B'C' + AB + B'D'$$
$$Y(A,B,C,D) = BD + A'D + BC + C'D + A'B'C'$$
$$Z(A,B,C,D) = + C' + A'B'D' + AB$$

This optimized implementation uses nine AND gates and three OR gates, which saves one AND gate over the individually optimized expressions (solving each function as minimum SOP individually, and sharing gates when possible).

7.4 Example Problems

Least Cost Implementation

1. Find the lowest cost implementation of

$$F(A, B, C, D) = \Sigma m(2, 3, 4, 6, 7, 10, 11, 12, 14).$$

2. Find the lowest cost implementation of

$$F(A, B, C, D) = \Pi M(2, 5, 6, 9, 13, 14).$$

3. Find the lowest cost implementation of

$$F(A, B, C, D, E) = B'C'D' + B'C'E + A'BCE + A'BCD.$$

4. Find the lowest cost implementation of

$$F(A, B, C, D) = \Sigma m(0, 2, 5, 6, 7, 8, 9, 10, 11, 14).$$

5. Find the lowest cost implementation of

$$F(A, B, C, D) = \Sigma m(2, 5, 6, 7, 10, 14).$$

Multiple Output Optimization

1. Find the optimized implementation of

$$X(A, B, C, D) = \Sigma m(2, 3, 4, 6, 7, 10, 12)$$
$$Y(A, B, C, D) = \Sigma m(4, 6, 7, 10, 12, 14, 15).$$

   How many gates and/or inputs do you save by implementing the circuits together rather than using minimum SOP?

2. Find the optimized implementation of

$$X(A, B, C, D) = \Sigma m(0, 2, 7, 10) + \Sigma d(12, 15)$$
$$Y(A, B, C, D) = \Sigma m(2, 7, 8) + \Sigma d(0, 5, 13).$$

   How many gates and/or inputs do you save by implementing the circuits together rather than using minimum SOP?

3. Find the optimized implementation of

$$X(A, B, C, D) = \Sigma m(0, 1, 2, 9, 15)$$
$$Y(A, B, C, D) = \Sigma m(0, 2, 8, 12, 15).$$

   How many gates and/or inputs do you save by implementing the circuits together rather than using minimum SOP?

4. Find the optimized implementation of

$$X(A, B, C, D) = \Sigma m(3, 7, 9, 14) + \Sigma d(1, 4, 6, 11)$$
$$Y(A, B, C, D) = \Sigma m(6, 7, 12) + \Sigma d(3, 14).$$

   How many gates and/or inputs do you save by implementing the circuits together rather than using minimum SOP?

5. Find the optimized implementation of

$$X(A, B, C, D) = \Sigma m(2, 3, 7, 9, 10, 11, 13)$$
$$Y(A, B, C, D) = \Sigma m(1, 5, 7, 9, 13, 14, 15).$$

   How many gates and/or inputs do you save by implementing the circuits together rather than using minimum SOP?

8 Basic Semiconductor Physics

A basic overview of semiconductor physics can be helpful in understanding why issues from timing come about in digital logic circuits. It also helps in the understanding of how transistors are created, leading to the TTL and CMOS devices that are used in the creation of digital logic circuits. This book is by no means a comprehensive look into semiconductor physics, but just explains general concepts.

Semiconductors are materials that are not pure electrical insulators or conductors, but lie somewhere in between. They typically have four valence electrons. It is important to recall the difference between core and valence electrons. Core electrons are very tightly bound to the nucleus of an atom. They require very large amounts of energy to create shared bonds or to move freely throughout a material. Valence electrons are more weakly bound to the nucleus, and can be more easily excited to form bonds or move throughout a material.

Valence electrons require a certain amount of energy to conduct (move freely throughout a material). In insulators, valence electrons require a lot of energy to conduct. In metals, valence electrons conduct without the addition of extra energy. In a semiconductor, conduction requires a little bit of energy to conduct.

Silicon, the most ubiquitous semiconductor, and the material used in TTL and CMOS devices, has fourteen electrons. Ten of these are tightly bound to the nucleus and four are valence electrons. This allows silicon to form a crystalline lattice where one of each of the four valence electrons is shared with neighbors, creating a stable, happy situation for all atoms. The diagram shown in Figure 8.1 is a simplified version of a two-dimensional crystal lattice. Each of the circles corresponds to a silicon atom, and each line corresponds to a valence electron.

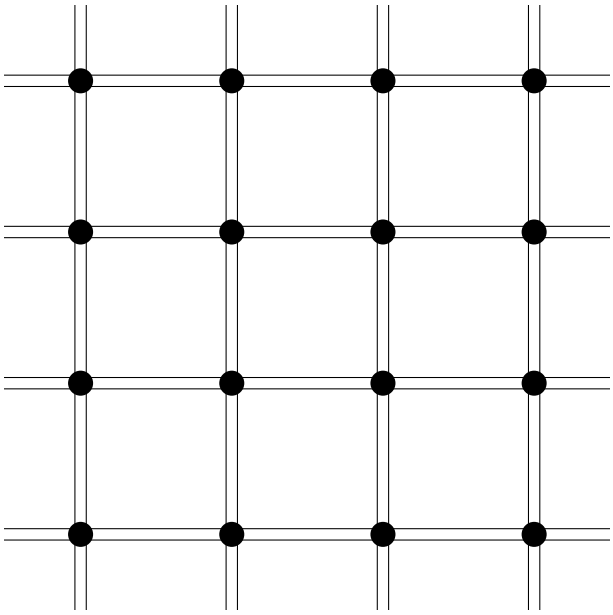


Figure 8.1: A simplified version of a two-dimensional crystal lattice.

On its own, a crystal lattice such as this where every atom interacts with eight valence electrons is not

interesting. Stable, happy situations for the atom means that there is no motivation (so to speak) for the electrons to do anything interesting. When the stable situation is disrupted, interesting things start to happen.

The doping process disrupts the crystal lattice's happy equilibrium. Doping comes in two types: p-type doping and n-type doping. In n-type doping, donor atoms introduce excess electrons into the crystal lattice. For example, phosphorus is used with silicon because it has five valence electrons to silicon's four. When a phosphorus atom replaces a silicon atom, it introduces an extra electron into the crystal. In p-type doping, acceptor atoms introduce "holes" (an absence of electrons) into the crystal lattice. Boron is typically used with silicon because it has three valence electrons. When a boron atom replaces a silicon atom, it introduces a hole into the crystal. Both of these situations, corresponding to n-type and p-type silicon, are shown in Figure 8.2.

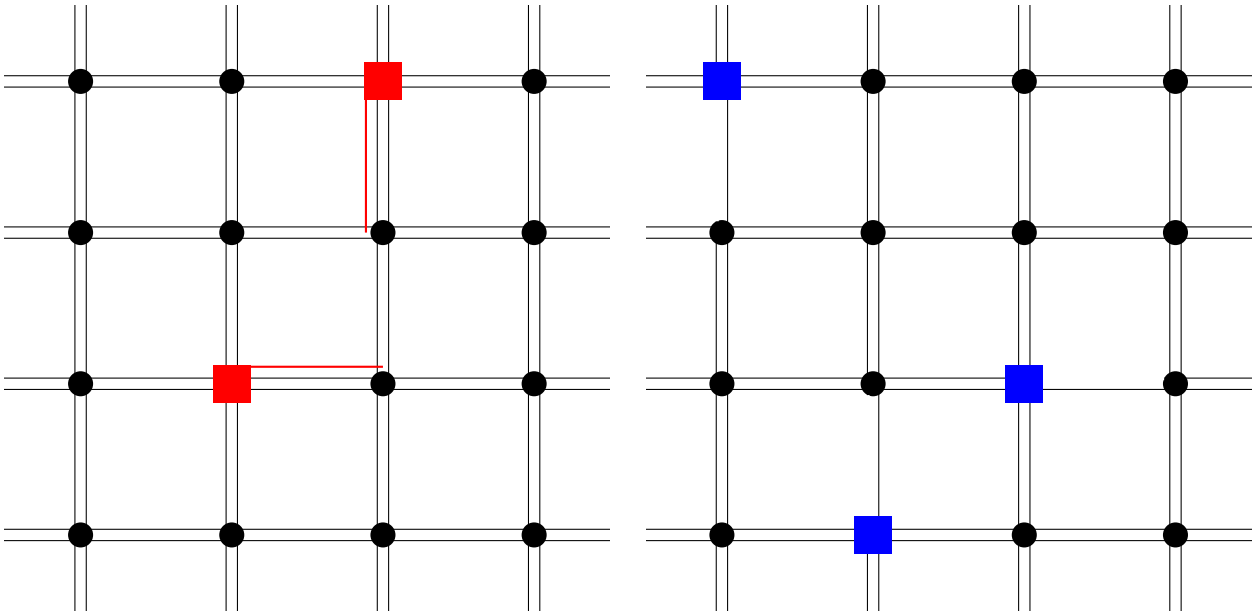


Figure 8.2: A simplified version of a two-dimensional silicon lattice with donor atoms (red) contributing extra electrons, and with acceptor atoms (blue) contributing extra holes.

While it may be simple to picture electrons conducting through a material, it is also possible for holes to move through a material. In this manner, doping causes it to be easier for conduction to occur. Rather than requiring external energy to be applied to the material, it is generally possible for just the heat of room temperature surroundings to cause electrons and holes to move about. Not many will conduct, but a few will gain sufficient energy from heat to do so. However, by themselves, the motion of electrons or holes through a doped semiconductor material will be random, causing no net electrical current through the material.

8.1 P-N Junction

Introducing together a p-type material and an n-type material creates what is called a p-n junction. This junction creates a very important type of semiconductor device called a diode. When voltage is applied to a p-n junction, depending on how it is oriented, it can cause the holes and electrons to move. When voltage is applied in such a manner that high potential is introduced to the p-type side, and low potential to the n-type side, the holes are repelled away from the p-type side to the n-type side, and the electrons are repelled away from the n-type side to the p-type side. This creates current from p-type to n-type as electrons travel in the opposite direction.

When voltage is applied in the opposite manner, electrons are attracted to the high potential applied to the n-type side, and holes are attracted to the low potential applied to the p-type side, and no current will flow. Both of these situations are depicted schematically in Figure 8.3.

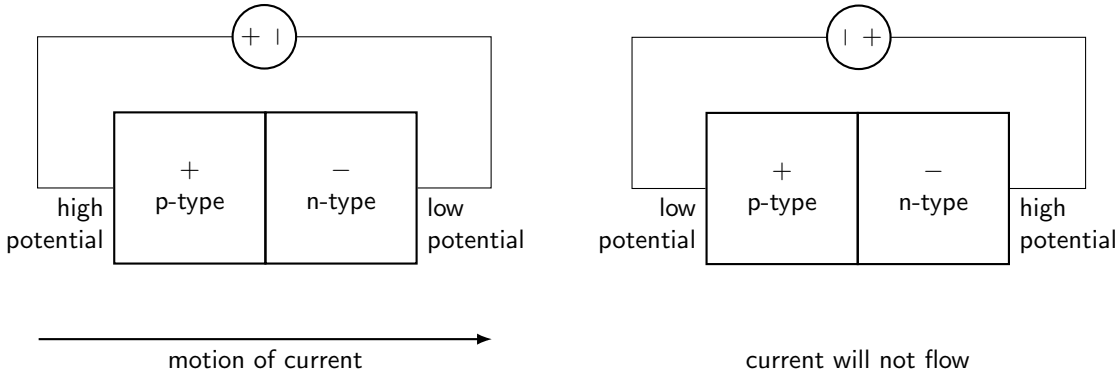


Figure 8.3: A p-n junction with voltage applied in both orientations.

This explains why diodes (including light-emitting diodes) need to be inserted into a circuit in a particular direction. Current will only flow in one direction. Diodes therefore act as one way valves in an electric circuit. A light-emitting diode is created when electrons and holes recombine in the semiconductor material to create photons. This does not naturally occur in all semiconductors (notably, it does not occur in silicon), but can be utilized in some materials to generate light sources with many beneficial properties.

8.2 Bipolar Junction Transistor (BJT)

A bipolar junction transistor (BJT) is fabricated by creating two p-n junctions. An NPN BJT has two n-type regions separated by a thin p-type region. A PNP BJT has two p-type regions separated by a thin n-type region. An NPN BJT is depicted schematically in Figure 8.4.

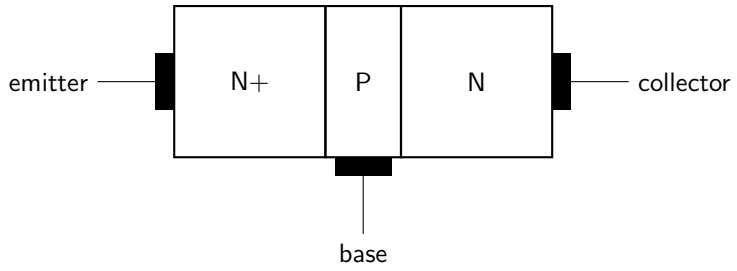


Figure 8.4: An NPN bipolar junction transistor.

The circuit symbol for an NPN BJT is shown in Figure 8.5. The arrow indicates the direction that current usually flows when the device is in a mode called forward active. In that case, current flows from collector to emitter, which might seem strange considering the names of the pins. However, remember that current flows in the direction opposite to electrons. In the forward active mode, electrons go from the emitter (which emits electrons) to the collector (which collects electrons).

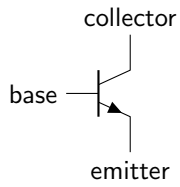


Figure 8.5: Circuit symbol for an NPN BJT, including labels of each terminal.

Different voltages can be supplied to the emitter, base, and collector. In fact, there are four different modes that a BJT can be in based on the comparison of the voltages at each of those points.

8.2.1 Resistor-Transistor Logic (RTL)

Resistor-transistor logic (RTL) is the easiest type of BJT-based logic circuits to understand. It is not used often these days due to the relatively large amounts of power consumed by the device. The transistors in RTL operate in the forward-active mode, in which current flows from collector to emitter if high potential is applied at the base. In this manner, the voltage at the base acts like an electrical switch. Current flows through the transistor if the base voltage is 5 V, and does not if the base voltage is 0 V.

In earlier chapters, we discussed how we could create different types of logic gates out of switches. This is essentially what happens in RTL. Series combinations of transistors are used in AND and NAND gates. Parallel combinations of transistors are used in OR and NOR gates.

A simple RTL inverter is shown in Figure 8.6. If the base voltage is 0 V, the transistor does not allow current to flow, and the output is tied to 5 V. If the base voltage is 5 V, current flows through the transistor and the output is pulled to 0 V.

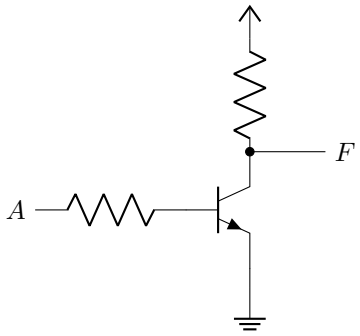


Figure 8.6: RTL inverter.

8.2.2 Transistor-Transistor Logic (TTL)

Transistor-transistor logic (TTL) uses more transistors but consumes much less power than RTL, making it very widely used (for example, in the 7400 series of logic gates we've discussed at length in this book). However, they are not necessarily as clear to understand as RTL devices. Inputs are applied to the emitter, rather than the base, of the BJT. When the emitter voltage is HIGH, it puts the transistor into reverse active mode (which causes current to flow from the emitter to the collector, which is the opposite of forward active mode). When the emitter voltage is LOW, current does not flow through the transistor.

A very simplified schematic of a TTL inverter is shown in Figure 8.7. Note that most actual modern TTL devices are more complicated with extra transistors, diodes, and other power saving and speed boosting properties included. However, AND and NAND gates still include transistors in series, with OR and NOR gates using transistors in parallel.

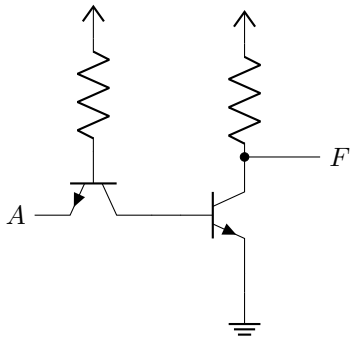


Figure 8.7: TTL inverter.

8.3 Metal Oxide Semiconductor Field Effect Transistor (MOSFET)

A field effect transistor operates by creating an electric field that builds up charges in a particular area in the device. The first three words: metal oxide semiconductor correspond to the materials that a MOSFET are made out of. A schematic of an n-type MOSFET is shown in Figure 8.8.

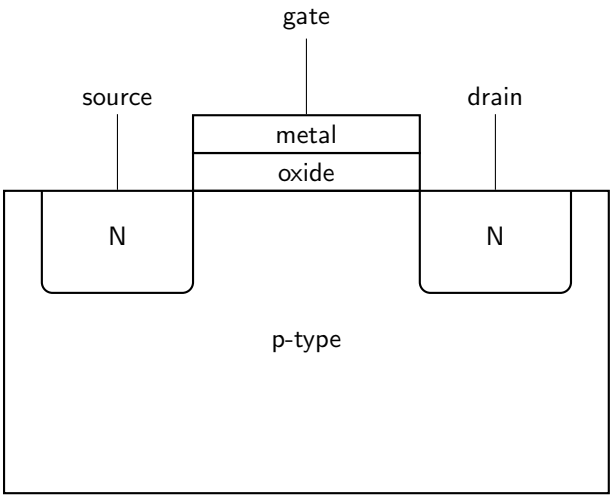


Figure 8.8: An NMOS field effect transistor.

If the gate is connected to high potential, positive charges build up on the metal, causing the electrons in the p-type semiconductor to build up between the n-type source and drain regions. This creates a channel that allows current to flow between the drain and the source. Otherwise, if the gate is connected to low potential, current does not flow between the drain and the source.

A p-type MOSFET is created by using n-type silicon with a p-type source and drain. Low potential on the gate will cause a channel to form (switch is ON), and high potential will lead to a switch is OFF scenario.

Using both NMOS and PMOS transistors together leads to a technology called complementary metal oxide semiconductor: CMOS. This is the technology that generally competes with TTL. It dissipates much less power than TTL devices because current does not flow in through the gate (unlike in TTL where current does flow through the base), so it is used almost exclusively in modern computers and microcontrollers.

The circuit symbols for NMOS and PMOS transistors are shown in Figure 8.9. Note the bubble on the gate of the PMOS transistor, indicating the opposite properties of the NMOS transistor.

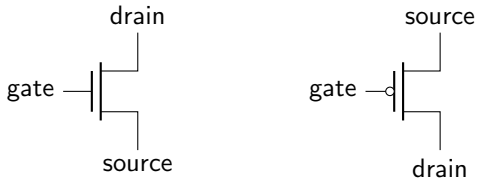


Figure 8.9: NMOS transistor (no bubble on the gate) and PMOS transistor (bubble on the gate).

A simple CMOS inverter is shown in Figure 8.10. The lack of resistors is a good hint that CMOS technology is completely voltage controlled rather than current controlled (as is the case with BJTs and TTL devices). Ideally, if no current flows, no power is consumed. However, because the layers of metal, oxide and semiconductor form a type of capacitor, it takes time for charge to accumulate into the channel. Therefore CMOS devices can be slower than TTL devices.

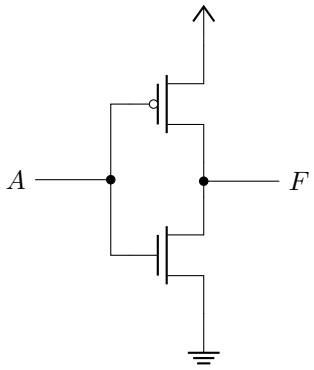


Figure 8.10: A simple CMOS inverter.

Regardless of the type of transistor (BJT or MOSFET), the fact that electrons need to physically move through a device causes delays between the input and output of a logic gate. This causes important ramifications that will be discussed in the next chapter.

9 Timing

As discussed in the previous chapter, logic gates are not as simple inside as they appear from their logic symbol. In fact, there are resistors, transistors, and diodes inside of each TTL device. Electrons require time to move through each circuit element. The time it takes for a signal to propagate from the input to the output of a logic device is known as the propagation delay. There are two types of propagation delays: t_{PLH} which is the amount of time it takes for an output signal to transition from LOW to HIGH, and t_{PHL} which is the amount of time it takes for an output signal to transition from HIGH to LOW. A timing diagram showing input and output signals for an inverter, including both propagation delays, is shown in Figure 9.1.

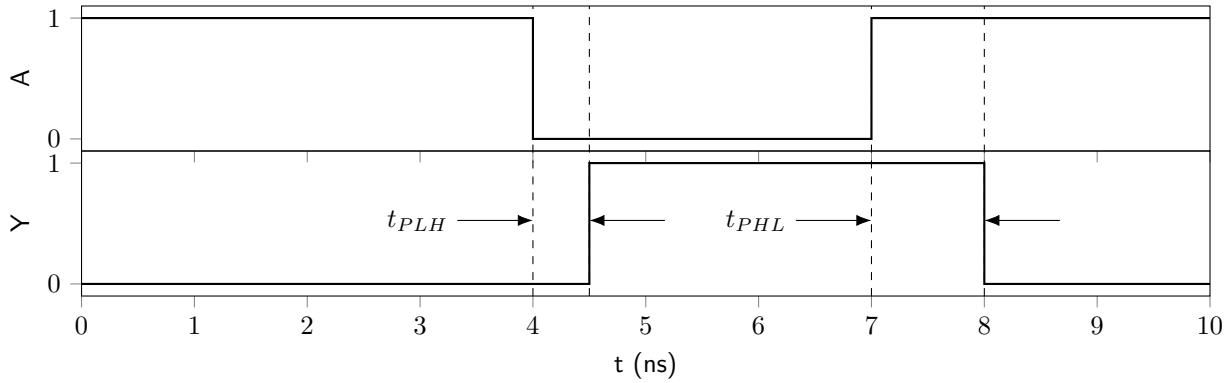


Figure 9.1: Input and output signals for an inverter, including propagation delays.

Logic chip datasheets list propagation delays. The 7404 inverter has a t_{PLH} between 12–22 ns, and t_{PHL} between 8–15 ns. The exact duration of the propagation delay depends on the temperature, the number of devices connected to the output of the logic gate, and other external factors. However, it gives us an idea of a typical propagation delay for that type of logic gate.

In addition to logic gates, wires also introduce delay into a circuit. This delay is approximately 1 ns per 15 cm of wire. We will treat all wire delays as zero in this book, but if very long wires or cabling is used it is clear that it may be cause for concern about additional delays.

The delay in digital devices causes the voltage to increase gradually from 0 V to 5 V. The transition from a logic LOW to a logic HIGH takes time; this is the analog nature of digital devices. Figure 9.2 shows how the voltage signal changes with time in the real world (left figure), and how an idealized version of the voltage change is visualized (right figure).

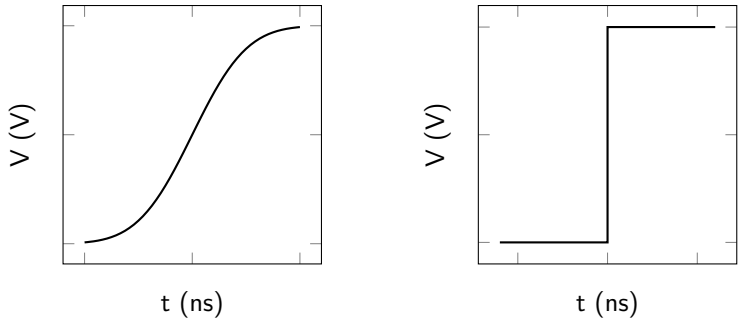


Figure 9.2: Real voltage transitions (left) are gradual, while ideal voltage transitions (right) are instantaneous.

This means that there is a period of time where the logic output of the function is undefined. In this book, we will "hand-wave" this undefined logic levels and simply show instantaneous, idealized transitions in each timing diagram.

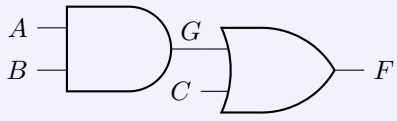
9.1 Timing Diagrams

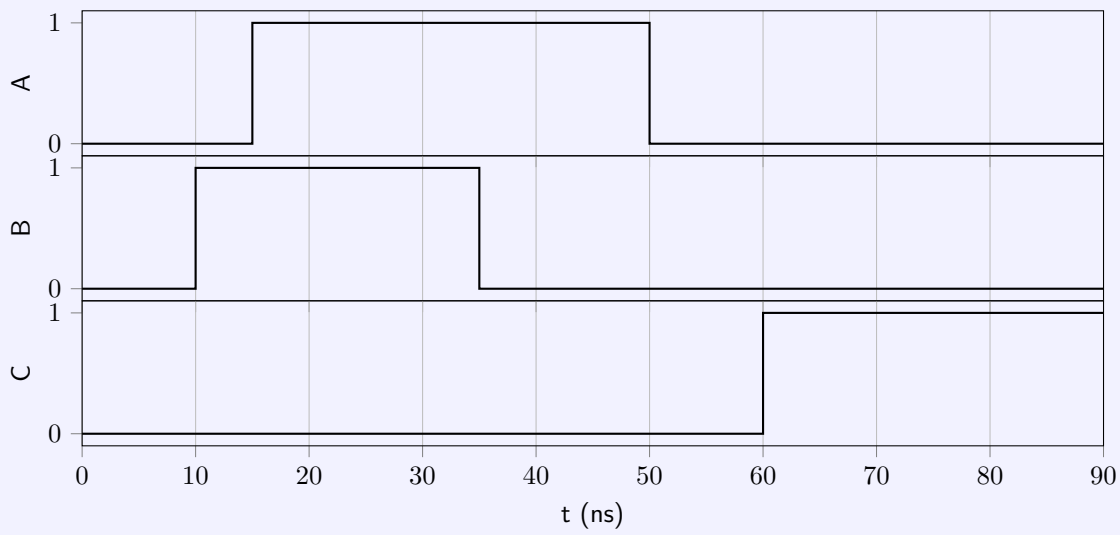
Timing diagrams are used to enable us to better understand the relationship between input and output signals in a circuit. Each signal path is carefully analyzed and each delay is included. The best way to do this is by starting from the far left-hand side of the circuit diagram and analyzing signals until the output is reached.

In this book, we will assume that all signals have been well-defined for a sufficiently long period of time such that all outputs are well-defined at a time of 0 ns.

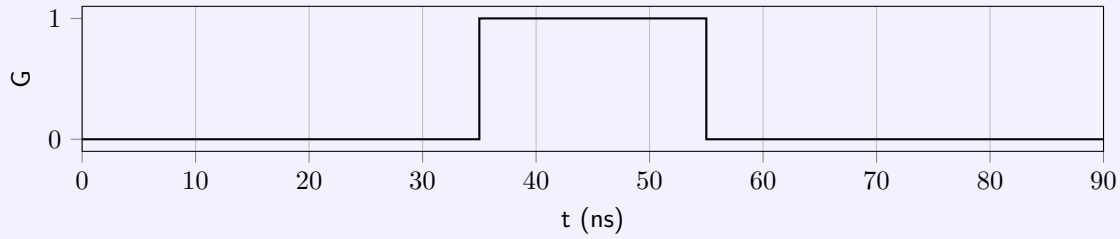
Example: Simple two-level circuit timing diagram

Assuming that $t_{PLH} = t_{PHL} = 20$ ns delay on each logic gate, draw the output signals G and F for the following circuit, given inputs A, B, and C.

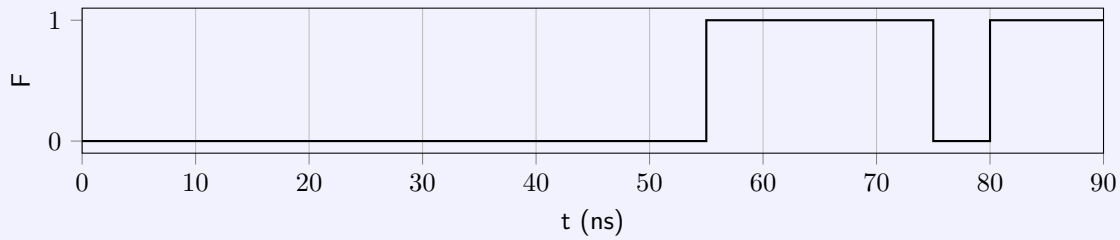




Start by determining the signal for G, which is equal to AB. In an ideal world, the signal would start out as zero and transition from zero to one at 15 ns. Due to the delay, the transition will not occur until 35 ns. The signal would then transition from one to zero ideally at 35 ns. Due to the delay, this will actually occur at 55 ns. The signal for G is graphed below.



Next, the signal F is determined. In this case it is equal to $G + C$. Note that we must include the delays from the AND gate, so do not start by looking at $AB + C$, but look at $G + C$ which already includes the AND gate delays. F would start at zero and transition from zero to one ideally at 35 ns. Due to delay this won't occur until 55 ns. The transition from one to zero which would ideally occur at 55 ns will not occur until 75 ns. Another transition from zero to one (from signal C going HIGH) would ideally occur at 60 ns but will occur at 80 ns due to the delay. The signal for F is graphed below.



9.2 Static Hazards

Hazards occur in a circuit when the output has a glitch due to gate delays that would not be present if all gates were infinitely fast. There are two types of hazards: static hazards and dynamic hazards.

Static hazards occur when the output should be a steady signal (either HIGH or LOW) but has a temporary glitch. There are two types of static hazards.

- Static 1 hazard – occurs when the output should be HIGH and has a temporary LOW glitch, they occur only in SOP circuits.
- Static 0 hazard – occurs when the output should be LOW and has a temporary HIGH glitch, they occur only in POS circuits.

Both types of static hazard are shown in Figure 9.3.

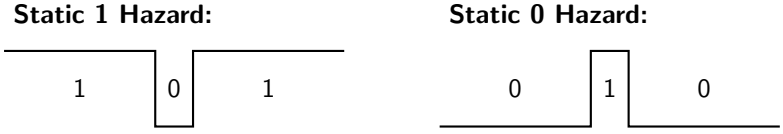
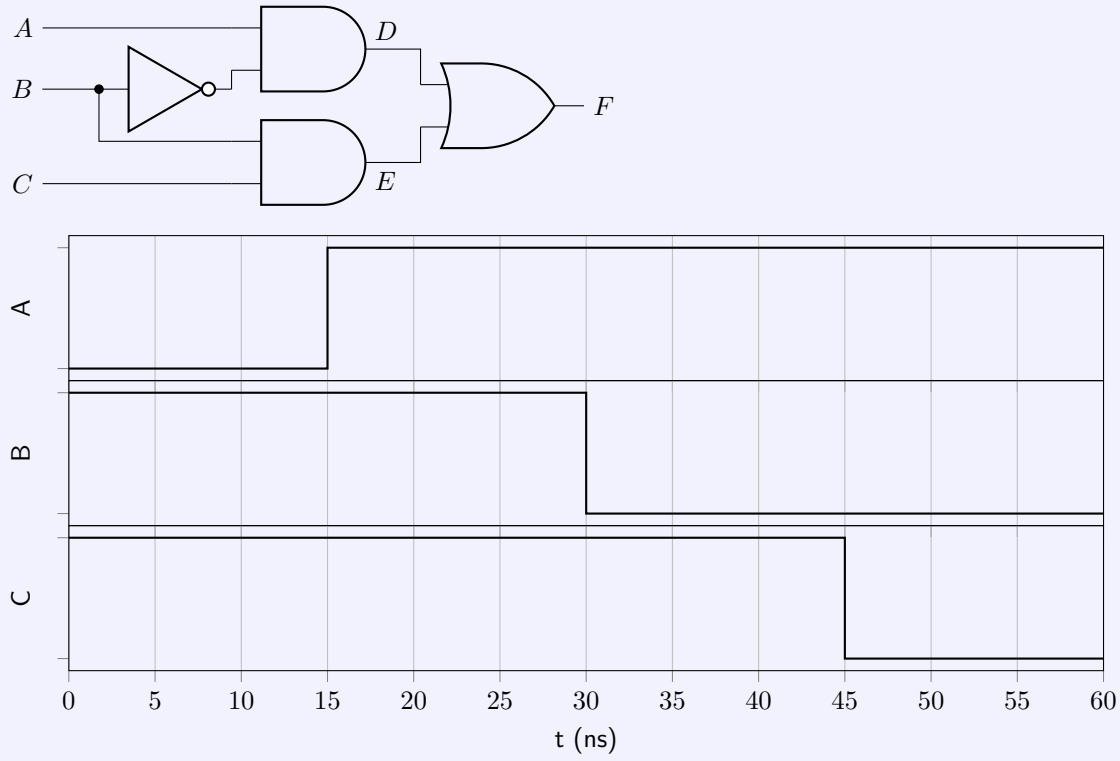


Figure 9.3: Static 1 and static 0 hazards.

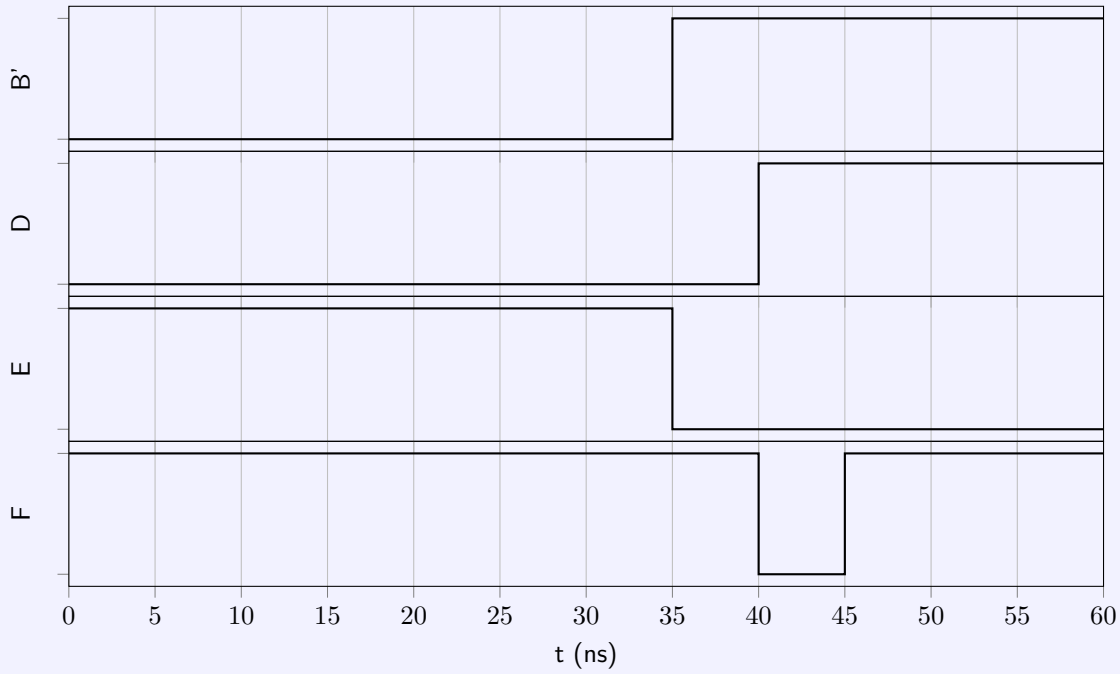
Example: Static 1 hazard

Create a timing diagram for $F = AB' + BC$. Each logic gate (including the inverter) has a 5 ns delay.



Determine what the output *should* be for the duration of the timing diagram. The inputs ABC go from 011–111–101–100. The output should be HIGH the entire time.

Create a timing diagram for each output signal. B', D, E, and F are all shown below on one graph.



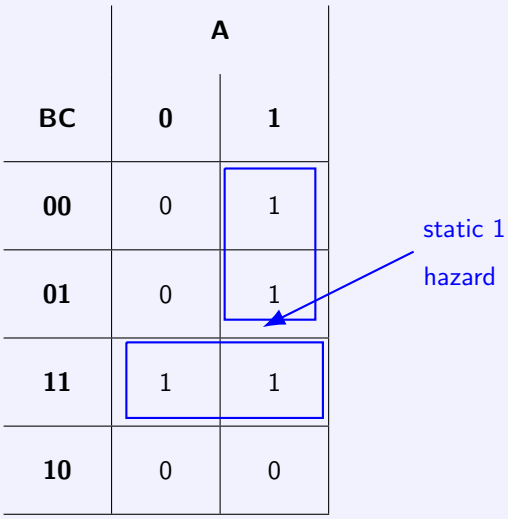
Analyzing the output signal, there is a glitch to zero from 40–45 ns. This glitch corresponds to the transition from 111–101 on the input. This is a static 1 hazard.

9.2.1 Identifying and Eliminating Static Hazards

We can use a k-map to identify where static hazards occur in a circuit. Hazards occur if there is a transition between two minterms or two maxterms that aren't covered by a prime implicant.

Example: Using a k-map to identify static hazards

We will use the previous example to find the location of a static hazard using a k-map. The minterms of $F = AB' + BC$ are m_3, m_4, m_5, and m_7. The two prime implicants are AB' covering $m_4 - m_5$ and BC covering $m_3 - m_7$. The hazard occurs between minterms m_7 and m_5, which is not covered by a prime implicant.



To eliminate static hazards, every prime implicant must be included in an expression. This ensures that there are no gaps between loops of minterms or maxterms. This keeps the signal either HIGH (in SOP expressions) or LOW (in POS expressions) during signal transitions that would otherwise cause an output glitch.

9.3 Dynamic Hazards

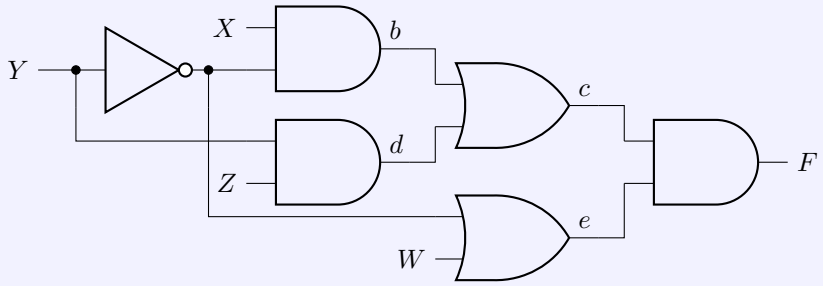
Dynamic hazards occur when an output oscillates briefly while changing value from zero to one or from one to zero. Examples of dynamic hazards are shown in Figure 9.4. These types of hazards often occur in larger logic circuits where there are many routes to the output from the input.

Figure 9.4: Dynamic hazards

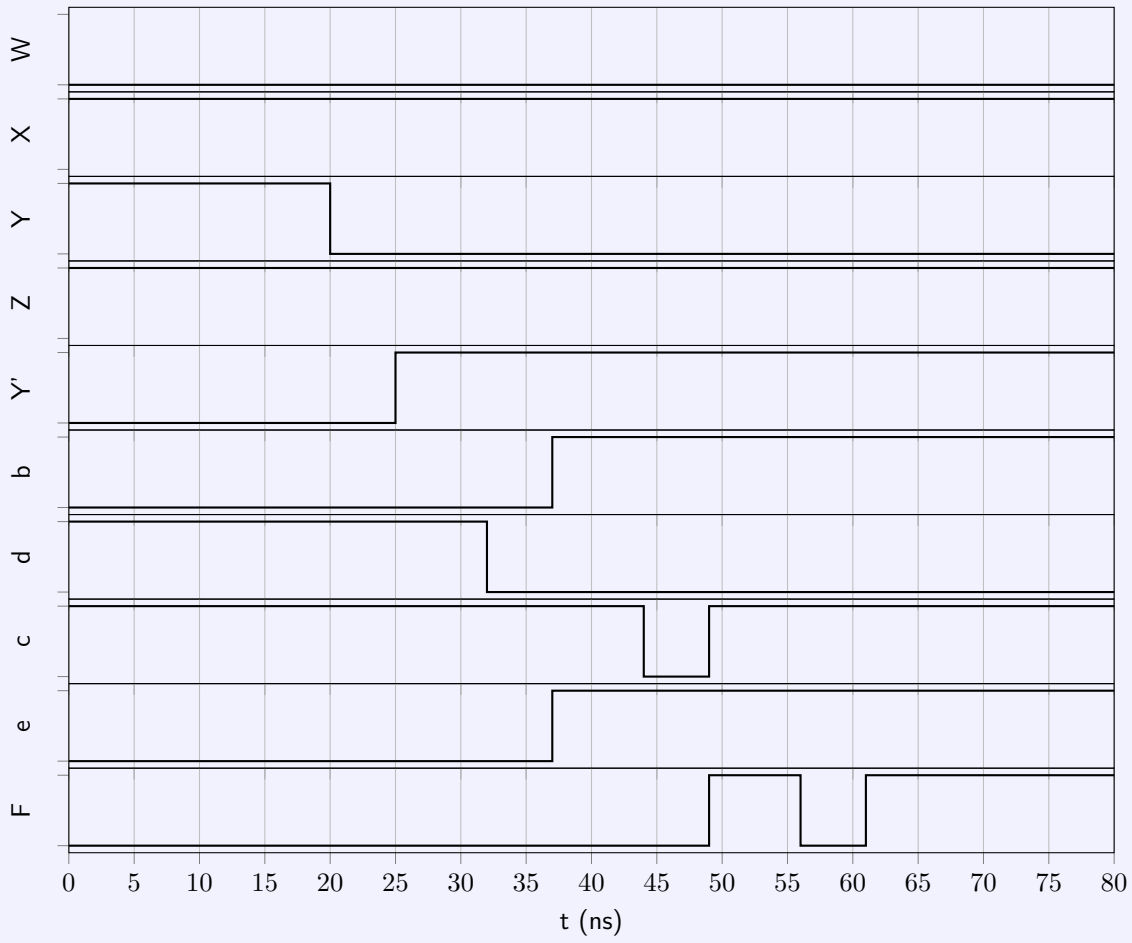
Dynamic hazards can be difficult to diagnose. The good news is that a circuit that has no static hazards will also have no dynamic hazards. Therefore a two-level hazard-free circuit is not only the fastest implementation of a circuit, but it also is guaranteed to have no hazards of any kind as long as only one input changes at a time.

Example: Circuit with a dynamic hazard

Each inverter has a 5 ns delay, and all other logic gates have a 12 ns delay.



The Boolean expression is $F(W, X, Y, Z) = (WY')(XY' + YZ)$. As the input goes from 0111–0101 at 20 ns, the output ideally would transition from zero to one. The timing diagram demonstrates that instead of a simple transition, a dynamic hazard occurs instead.



9.4 Example Problems

Timing Diagrams

1. Draw a timing diagram for $Z(A, B, C) = (A'B + B'C)$. Inverters have a propagation delay of 2 ns and AND and OR gates have a propagation delay of 5 ns. The input signals are shown in Figure 9.5.

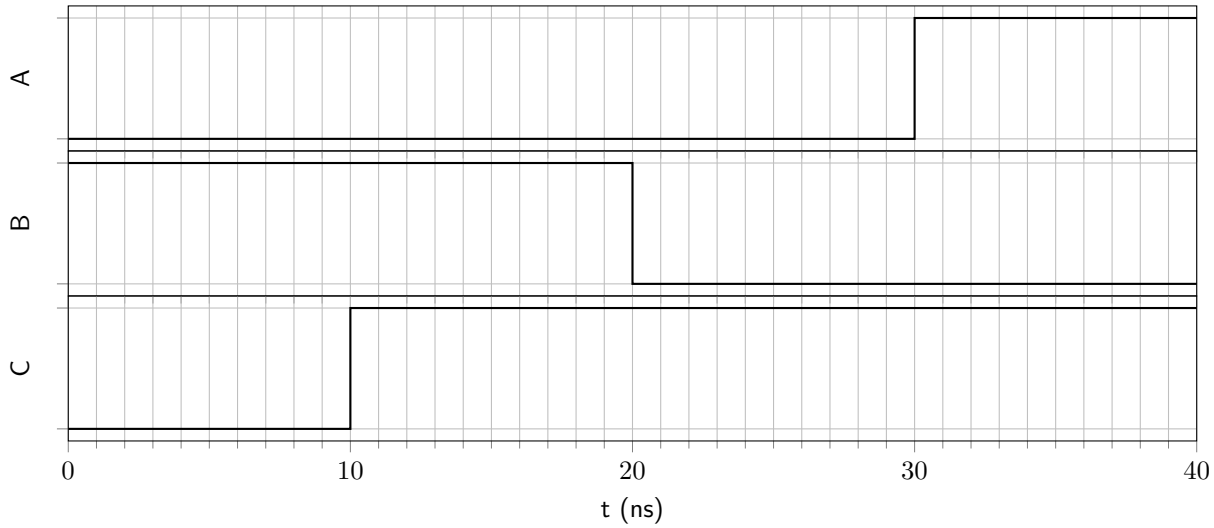


Figure 9.5: Input signals for timing diagram questions 1 and 2.

2. Draw a timing diagram for $F(A, B, C) = (B \oplus C)(A + C)$. AND gates have a delay of 2 ns, OR gates have a delay of 3 ns, and XOR gates have a delay of 5 ns. The input signals are shown in Figure 9.5.

3. Draw a timing diagram for $F(A, B, C, D) = (B' + D')(A + B + C)$. Inverters have a propagation delay of 2 ns and AND and OR gates have a propagation delay of 5 ns. The input signals are shown in Figure 9.6.

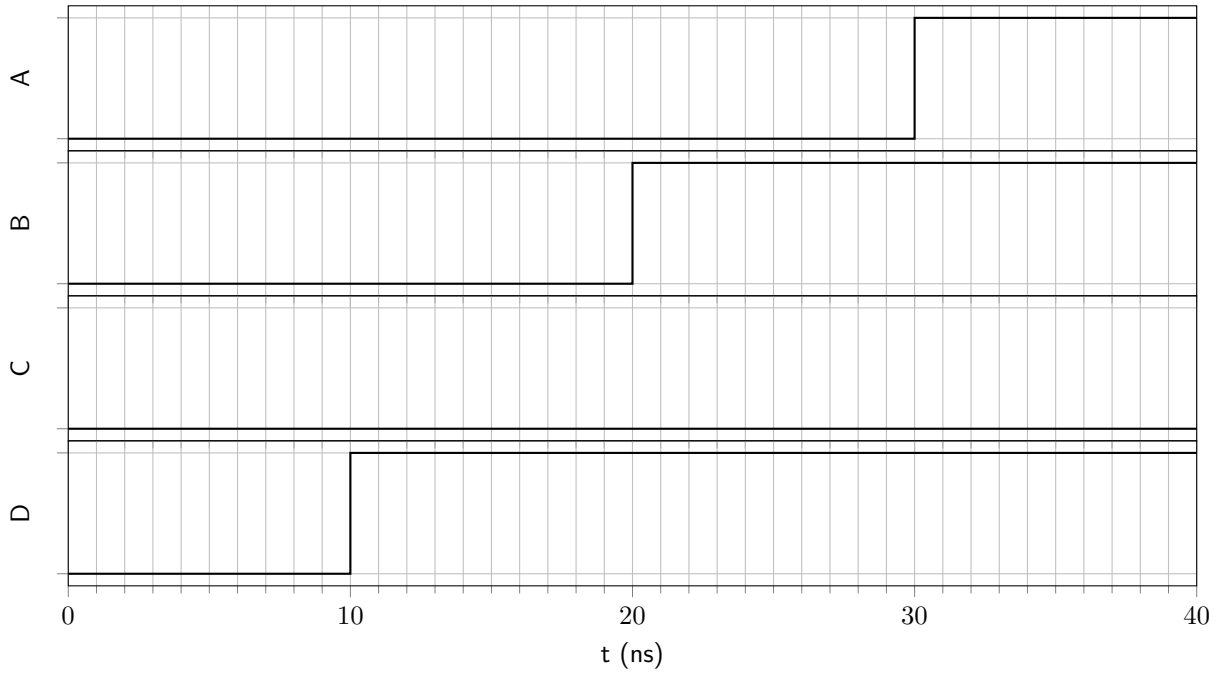


Figure 9.6: Input signals for timing diagram questions 3 and 4.

4. Draw a timing diagram for $F(A, B, C, D) = A'C'D + A'BC'$. All logic gates have a propagation delay of 5 ns. The input signals are shown in Figure 9.6.

5. Look up the datasheet for the 74LS139A 2 to 4 decoder. Although we haven't discussed decoders yet, you will be able to interpret the logic diagram which only uses inverters and NAND gates. Assuming that every logic gate has a delay of 10 ns, draw the output for $1Y1$ giving the input signals shown in Figure 9.7.

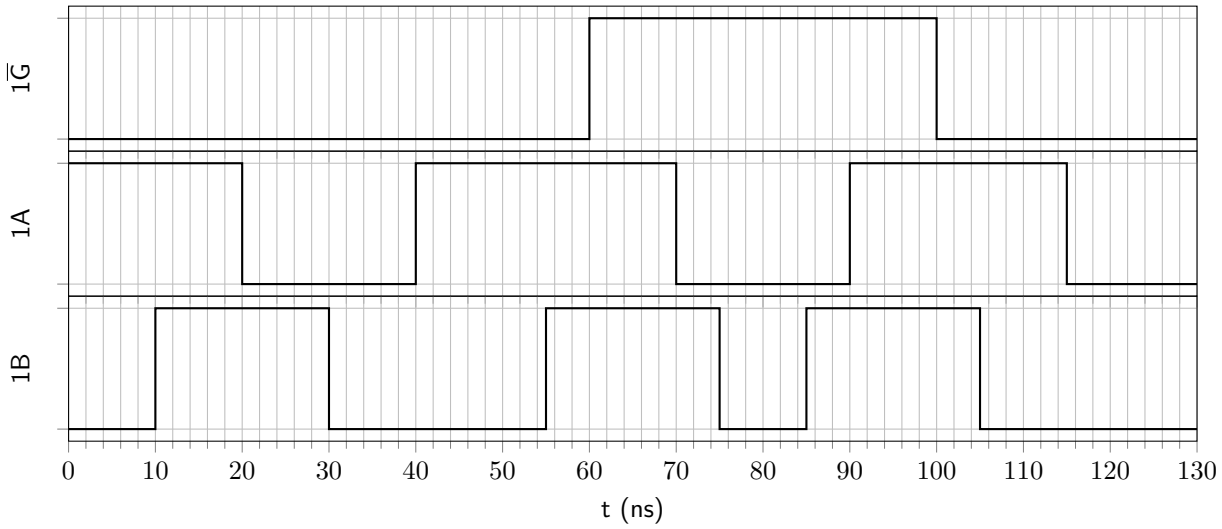


Figure 9.7: Input signals for timing diagram question 5.

Hazards

1. Find a hazard-free SOP expression for

$$F(A, B, C) = \Sigma m(2, 3, 5, 7).$$

2. Find a hazard-free SOP expression for

$$F(A, B, C, D, E) = \Sigma m(5, 12, 21, 23, 24, 28, 29, 31) + \Sigma d(6, 7, 14, 15).$$

3. Find a NAND-only hazard-free expression for

$$F(A, B, C, D) = \Sigma m(0, 1, 3, 4, 6, 11) + \Sigma d(8, 12, 13).$$

4. Find a NOR-only hazard-free expression for $F(A, B, C, D) = A'B'D' + AC'D' + ABCD$.

5. Find a NOR-only hazard-free expression for

$$F(A, B, C, D) = \Sigma m(6, 7, 12) + \Sigma d(3, 14).$$

10 Logic Devices

Now that we have a good understanding of combinational logic, we can start learning about logic devices. Logic devices are made out of AND/OR/NOT gates, but are capable of performing useful or complicated functions in a single package.

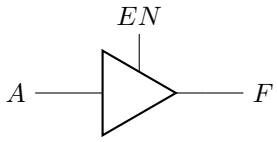
10.1 Buffers and Tri-State Devices

Buffers are logic devices that transfer voltage from input to output. Usually, they are capable of increasing the amount of current that is sourced or sunk by a device. This can be particularly useful if a logic gate output has been connected to many logic gate inputs; in these situations the amount of current provided by the output of the logic gate may not be enough to source all of the logic gates that it's connected to.

Non-inverting buffers simply increase the current without changing the logical value of the input. (In other words, if the input is LOW, the output is LOW. If the input is HIGH, the output is HIGH.) Inverting buffers increase the current capability while also changing the output to make it the logical complement of the input. (HIGH becomes LOW and vice versa.)

A tri-state buffer is a very special type of buffer that provides a third option. We've seen that a LOW signal is connected to 0 V or ground. A HIGH signal is connected to 5 V. These are physical electrical connections to a certain point in a circuit. The "third" state (tri-state) is called a high impedance, or high-Z mode. This tri-state means that the output is electrically isolated from the input. The output would register as a float on a logic probe, which is very different in an important way from a LOW or HIGH signal. In the high-Z mode, it is almost like the component is not connected to any part of the circuit.

The circuit symbol and truth table for a tri-state buffer is shown in Figure 10.1. Note the use of the letter Z on the truth table. This indicates the high-Z mode.



EN	A	F
0	0	Z
0	1	Z
1	0	0
1	1	1

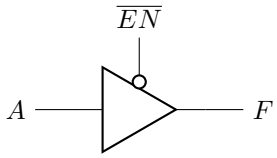
Figure 10.1: A tri-state buffer represented as a circuit diagram and a truth table.

10.1.1 Types of Tri-State Buffers

There are different types of tri-state buffers. the types refer to how the enable is activated (with active HIGH a one enables the buffer; with active LOW a zero enables the buffer) and how the output is configured (active HIGH output means a one on the input will be a one on the output; active LOW output means a one on the input becomes a zero on the output).

The type of tri-state buffer described above in Figure 10.1 is an active HIGH enable, active HIGH output device.

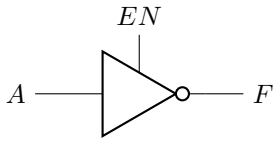
A tri-state buffer that requires an enable signal of zero in order to function as a buffer, and an enable signal of one to put the output into the high-Z mode would be called an active LOW enable, active HIGH output buffer. It is depicted in Figure 10.2.



EN	A	F
0	0	0
0	1	1
1	0	Z
1	1	Z

Figure 10.2: An active LOW enable, active HIGH output tri-state buffer.

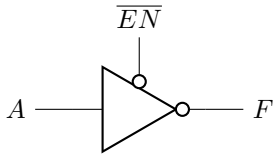
An inverting tri-state buffer that requires an enable signal of one in order to function as a buffer would be called an active HIGH enable, active LOW output device. It is depicted in Figure 10.3.



EN	A	F
0	0	Z
0	1	Z
1	0	1
1	1	0

Figure 10.3: An active HIGH enable, active LOW output tri-state buffer.

Finally, an inverting tri-state buffer that requires an enable signal of zero in order to function as a buffer would be called an active LOW enable, active LOW output device. It is depicted in Figure 10.4.



EN	A	F
0	0	1
0	1	0
1	0	Z
1	1	Z

Figure 10.4: An active LOW enable, active LOW output tri-state buffer.

10.1.2 Uses of Tri-State Buffers

Tri-state buffers have many uses in digital computing. Their major advantage comes in the fact that outputs of tri-state buffers can be connected together, which is not allowed with most other logic gates. Consider what would happen if the outputs of two AND gates were connected together as shown in Figure 10.5. The

upper AND gate has an output of zero, and the lower AND gate has an output of one. The two competing outputs creates a short circuit.

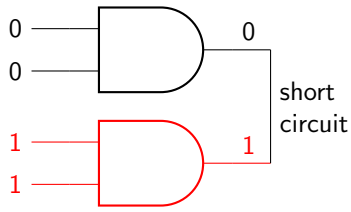


Figure 10.5: A short circuit can be created when the output of two logic gates are connected together.

In the case of tri-state buffers, as long as only one buffer is enabled at a time, the output of many of them can be connected together. Only the enabled buffer is electrically connected to the output. The other buffers are completely isolated from the circuit, almost as if they are not there.

Tri-state buffers are therefore frequently used in computing buses. An electrical bus is simply a connection of wires used to send a signal from place to place in a computer. A data bus could be used to route data from several different places in a computer to memory, a hard drive, or other places. Because we only want one of the data sources to be supplying data to the bus at any one time, access to write to the data bus is therefore controlled by tri-state devices, as shown in Figure 10.6.

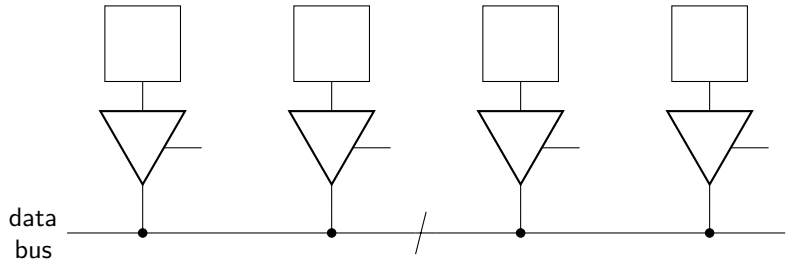


Figure 10.6: Four devices are connected to a single data bus. As long as only one is enabled at a time, there is no chance of short circuits on the output.

An electrical bus is depicted on a circuit diagram as a line with a slash through it. A number next to the slash indicates how many wires are included in the bus.

Tri-state buffers are also used to create bi-directional pins. A bi-directional pin means it can be used as an input or an output, depending on the configuration. This is very useful in microcontrollers. The popular ATmega328P, which is the microcontroller included in the Arduino Uno, has twenty pins that can be used in either direction: input or output. If you have a project that needs two input pins and fourteen outputs, you can use the ATmega328P. If you need ten inputs and ten outputs, you can use the ATmega328P. If you need fifteen inputs and one output, you can use the ATmega328P. The ability to use a pin as an input or an output leads to maximum flexibility in design.

The bi-directional control of the pins is depicted in Figure 10.7. Note that if the tri-state buffer is enabled (EN = 1), then data goes from the microcontroller to the pin, which means that the pin is an output pin.

Otherwise, if the tri-state buffer is not enabled (EN = 0), then the pin is an input pin. Data can only go from the pin to the microcontroller, and not in the other direction.

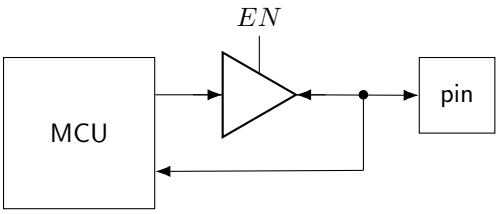


Figure 10.7: Schematic of a bi-directional data pin such as that used in a microcontroller (MCU).

10.2 Multiplexers (MUX)

A multiplexer acts as a switch with a control variable or variables controlling which input passes through to the output. Generally speaking, a MUX uses n control bits to select which one of 2^n inputs goes through to the one output pin. All of the other inputs are ignored.

There are a few purposes of multiplexers. One is to allow several signals to share one set of infrastructure. For example, if I wanted to start a phone company, and had sixteen subscribers, I wouldn't want to pay to install sixteen wires to control every individual customer's phone calls. I would want to install one wire and use it for all of the phone calls. By quickly switching between each phone conversation along one wire, that can be achieved. Multiplexers are used frequently therefore to lower cost any time there is a need to share signals along one set of infrastructure.

The second major purpose of multiplexers is to implement Boolean algebra expressions, and that will be discussed in detail later in this section.

10.2.1 2 to 1 MUX

A 2 to 1 MUX is used to select between two inputs. There are two input signals, a control signal, and one output. Just as with all devices, a MUX can be described as a circuit diagram, as a truth table, and as an equation. In this case, because a MUX is such a common device, it has its own symbol, a trapezoid. We will use the trapezoid symbol to discuss a MUX, but we can also see how a MUX is built with AND gates, OR gates, and inverters.

The circuit diagram for a 2 to 1 MUX is shown in Figure 10.8. The data inputs are I_0 and I_1, the control input is A, and the output is Z.

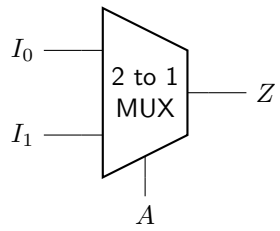


Figure 10.8: Circuit diagram of a 2 to 1 multiplexer.

If the control bit A is zero, then input signal I_0 is sent to the output. Otherwise, if A is one, I_1 is sent to the output. We can use this description of the operation of the 2 to 1 MUX to derive its Boolean algebra equation.

$$Z = A'I_0 + AI_1$$

Note that the decimal value of the control bit (in this case, limited to zero and one) tells us which of the inputs (zero or one) gets sent to the output.

It is also possible to describe the operation of this device using a truth table. The truth table for a 2 to 1 MUX is given in table 10.1.

A	I_0	I_1	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 10.1: Truth table for the operation of a 2 to 1 MUX

The expression and truth table enable us to determine how to build a MUX using just AND gates, OR gates, and inverters. This circuit diagram is shown in Figure 10.9.

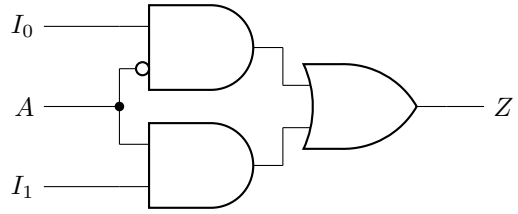


Figure 10.9: The construction of a 2 to 1 multiplexer using only AND gates, OR gates, and inverters.

To cement our understanding of 2 to 1 MUXes, let's look at a timing diagram, shown below in Figure 10.10. We will ignore all of the internal gate delays and just consider the basic operation of the device.

Any time the control bit A is 0, the input I_0 is sent to the output. Any time the control bit A is 1, the input I_1 is sent to the output.

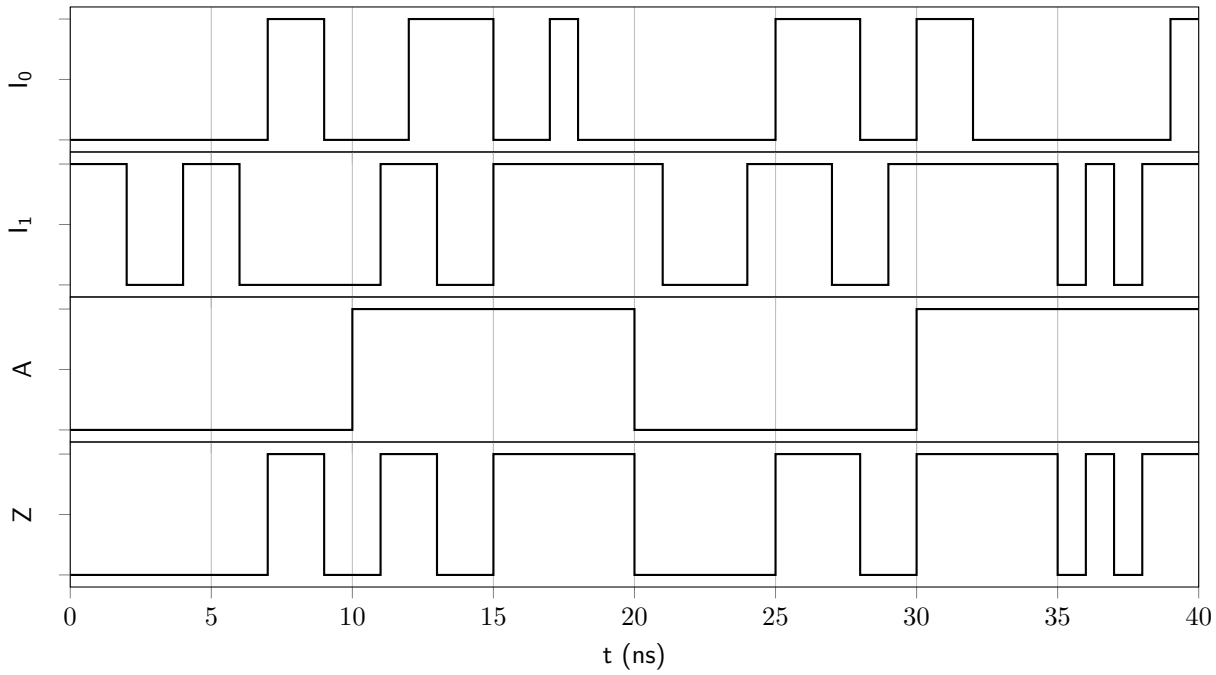


Figure 10.10: Timing diagram for a 2 to 1 MUX.

10.2.2 4 to 1 MUX

In a 4 to 1 multiplexer, there are now four data inputs (I_0, I_1, I_2, and I_3), two control inputs (A and B) to determine which of the data inputs is selected, and an output (Z). The decimal value of the control bits AB tells us which of the inputs (zero, one, two, or three) is sent to the output. The circuit symbol for a 4 to 1 MUX is shown in Figure 10.11.

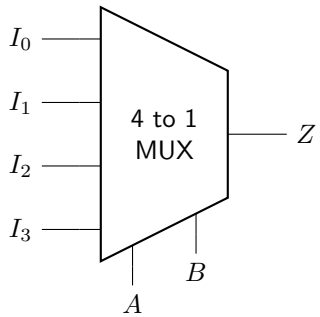


Figure 10.11: Circuit symbol of a 4 to 1 multiplexer.

The choice of control bits changes the connections internal to the MUX to determine which input changes the output. This is depicted schematically in Figure 10.12.

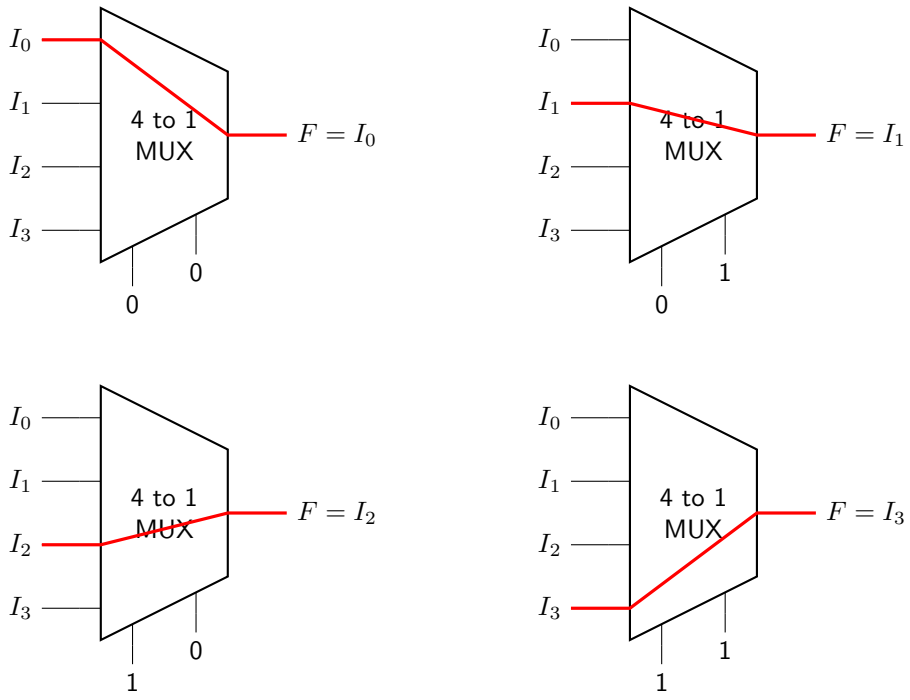


Figure 10.12: Connections between input pins and the output in a 4 to 1 MUX depend on the value of the control bits.

The Boolean expression that describes how the MUX operates follows.

$$Z = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$$

From the expression, it can be seen how a 4 to 1 MUX could be built using only AND gates, OR gates, and inverters. However, it is also possible to build a 4 to 1 MUX using 2 to 1 MUXes. This property of scaling up multiplexers makes them a very useful tool. The schematic for connecting 2 to 1 MUXes together to create a 4 to 1 MUX is shown in Figure 10.13.

Notice that the MUX closest to the output is controlled by the most significant control bit, and the MUXes farthest from the output are controlled by the least significant control bit. This would also be how we could connect together, say, 4 to 1 MUXes to create a 16 to 1 MUX.

10.2.3 Implementing Boolean Algebra Expressions with a MUX

Any Boolean expression that is a function of n variables can be implemented with an n to 1 MUX. In order to implement a Boolean expression out of a MUX, we exploit the format of the MUX equation, which is given below.

$$F = X'Y'I_0 + X'YI_1 + XY'I_2 + XYI_3$$

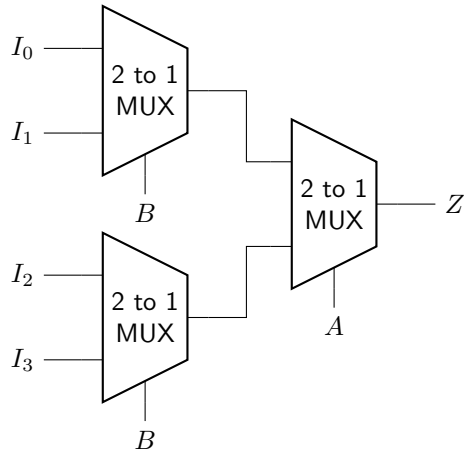


Figure 10.13: Creating a 4 to 1 multiplexer from 2 to 1 multiplexers.

The goal is to determine which control bits (indicated as XY in the above MUX equation) would lead to the fewest number of external logic gates connected to the MUX. We determine how many logic gates would be required by looking at the forms of I_0, I_1, I_2, and I_3. There are six different possible combinations of control bits that can be chosen from four input variables A, B, C, and D.

For each of the different control bits, we check to see which four squares in a k-map are unchanged for each of the values of 00, 01, 10, and 11 of the control bits.

If A and B are chosen as the control bits, then the four quadrants of the k-map are distributed as shown in Figure 10.14. Because A and B are constant along every column, the four quadrants are the same as the four columns.

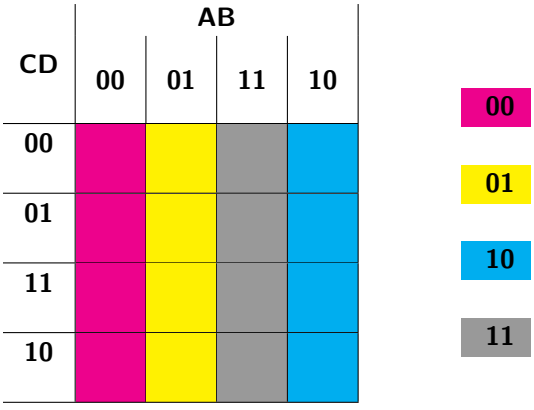


Figure 10.14: The four quadrants of cells to be looped when the control bits are A and B.

If A and C are chosen as control bits, then the four quadrants of the k-map are distributed as shown in Figure 10.15.

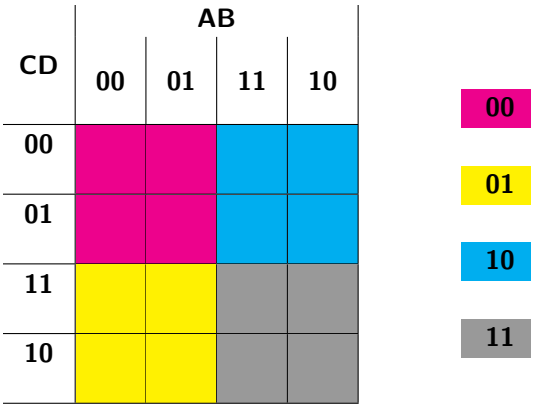


Figure 10.15: The four quadrants of cells to be looped when the control bits are A and C.

If A and D are chosen as control bits, then the four quadrants of the k-map are distributed as shown in Figure 10.16.

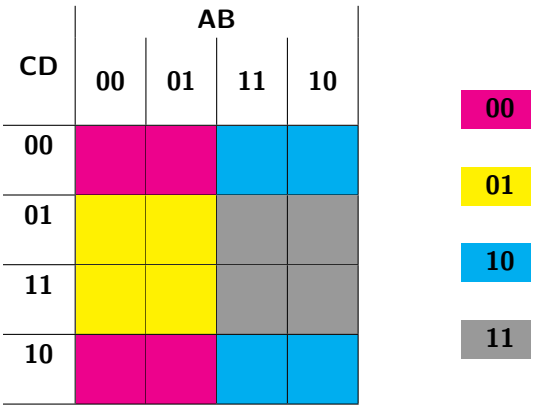


Figure 10.16: The four quadrants of cells to be looped when the control bits are A and D.

If B and C are chosen as control bits, then the four quadrants of the k-map are distributed as shown in Figure 10.17.

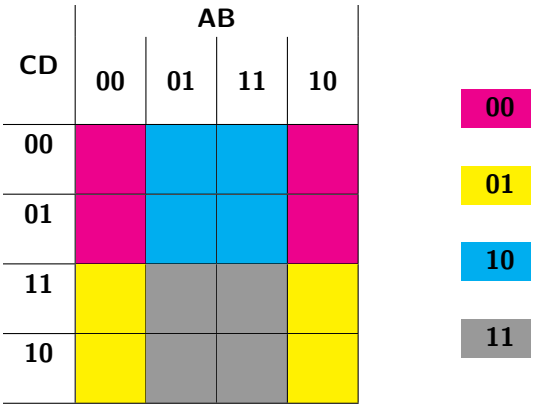


Figure 10.17: The four quadrants of cells to be looped when the control bits are B and C.

If B and D are chosen as control bits, then the four quadrants of the k-map are distributed as shown in Figure 10.18.

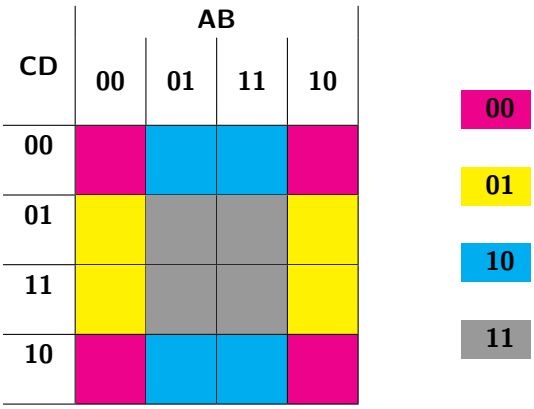


Figure 10.18: The four quadrants of cells to be looped when the control bits are B and D.

If C and D are chosen as the control bits, then the four quadrants of the k-map are distributed as shown in Figure 10.19. Because C and D are constant along every row, the four quadrants are the same as the four rows.

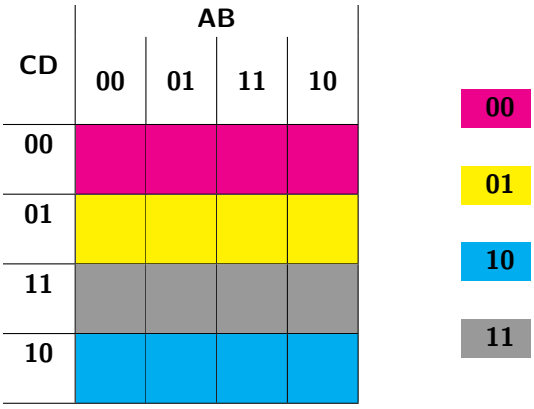
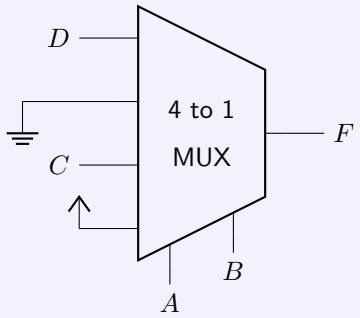


Figure 10.19: The four quadrants of cells to be looped when the control bits are C and D.

Example: Implementing a Boolean expression with a 4 to 1 MUX

Implement the function $F(A, B, C, D) = \Sigma m(1, 3, 10, 11, 12, 13, 14, 15)$ using a MUX. If we were to find a minimum SOP implentation of the function, it would be $A'B'D + AB + AC$. This would require four logic gates and ten inputs to implement. We will instead see how it would work if we implemented it using a MUX. We will analyze each of the six control bit options and see which possibility or possibilities are the best.

If we choose A and B as control bits, we analyze each of the four quadrants where the values of A and B do not change. Then, inside each of those quadrants, we must make the largest possible loops that contain minterms. In this case, where A and B are both 0, we can loop minterms m_1 and m_3, leading to $A'B'D$. Factoring out an $A'B'$ we get D. In the next column where A is 0 and B is 1, we do not loop anything. The expression is $A'B(0)$. The column farthest to the right (due to Gray code) is where A is 1 and B is zero. We would loop minterms m_{10} and m_{11}. That gives us the expression $AB'(C)$. We would loop the entire quadrant, giving us the expression $AB(1)$. This would require zero external logic gates, as shown below.

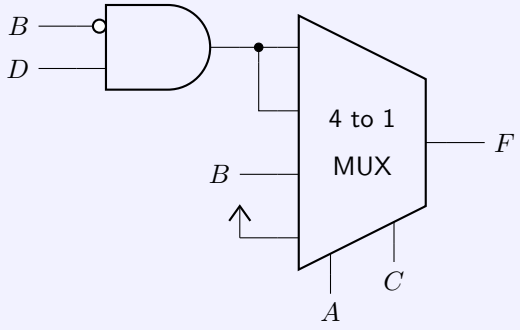


In this case, we have found an optimum implementation of F using a 4 to 1 MUX. But so that we can understand how to do this process with each set of control bits, we will continue the exercise.

If we choose A and C as control bits, we look in the four quadrants to find our MUX equation.

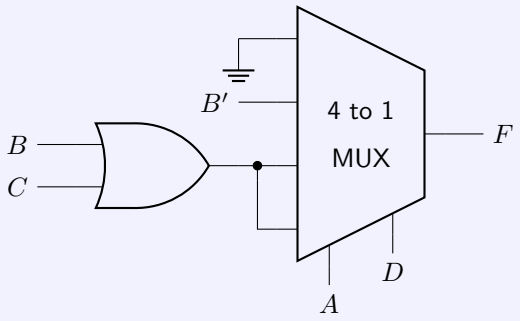
$$F = A'C'(B'D) + A'C(B'D) + AC'(B) + AC(1)$$

This would require one external logic gate to AND together B' and D. (Remember, we still do not consider inverters to count as logic gates.)



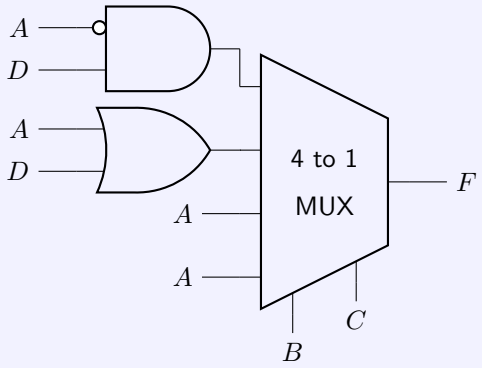
The next selection of control bits, A and D, would lead to the following MUX equation, which would require one external logic gate.

$$F = A'D'(0) + A'D(B') + AD'(B + C) + AD(B + C)$$



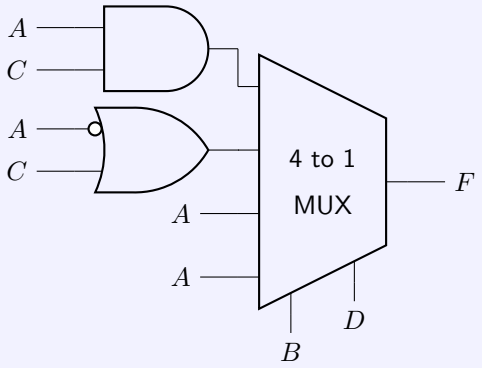
The selection of B and C as control bits would lead to two external gates, indicating that it is a particularly poor choice of control bits for the function F.

$$F = B'C'(A'D) + B'C(A + D) + BC'(A) + BC(A)$$



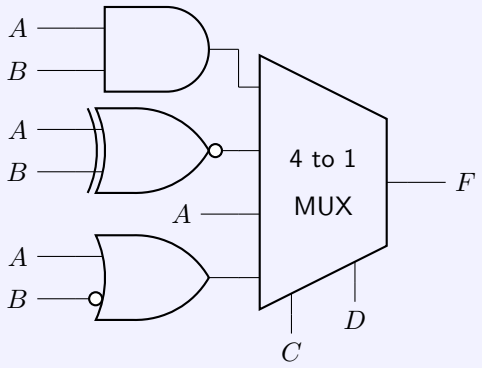
Using B and D as control bits would also lead to two external gates.

$$F = B'D'(AC) + B'D(A' + C) + BD'(A) + BD(A)$$



The last selection of control bits would be C and D. In this particular example, it would be the worst selection, as it would lead to three external gates being required to implement the Boolean function with a MUX.

$$F = C'D'(AB) + C'D(A \equiv B) + CD'(A) + CD(A + B')$$



Sometimes it can be simple to look at the k-map and identify which quadrants the largest loops live in,

and from there to identify the best choice of control bits. Other times it is not as simple. In those cases, look at each quadrant, and see what types of loops exist in each quadrant. The types of loops that you would have to make indicate the logic gates that would be required in the MUX implementation. These are explained below.

- ZERO minterms – no logic gate (connection to ground)
- ONE 1×1 loop – one AND gate required
- ONE 1×2 or 2×1 loop – no logic gate (connection directly to one of the inputs)
- ONE 2×2 or 1×4 loop – no logic gate (connection to Vcc)
- TWO 1×1 loops – one XOR or XNOR gate required
- TWO 1×2 or 2×1 loops – one OR gate required

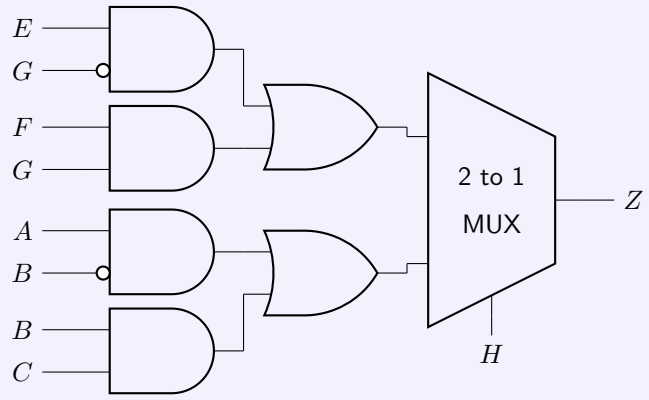
Sometimes a Boolean algebra expression is a good candidate for implementation with a MUX, although it has more than 4 variables, and cannot be implemented with a 4 to 1 MUX. In these cases, it is useful to determine how it could be implemented with multiple 2 to 1 MUXes instead. Identify a variable that exists in each term in either true or complemented form, and that would be the control bit on the most significant MUX. Then repeat the process for subsequent MUXes.

Example: Implementing a Boolean function of many variables

Use multiplexers and a minimum number of external gates to realize the function given below.

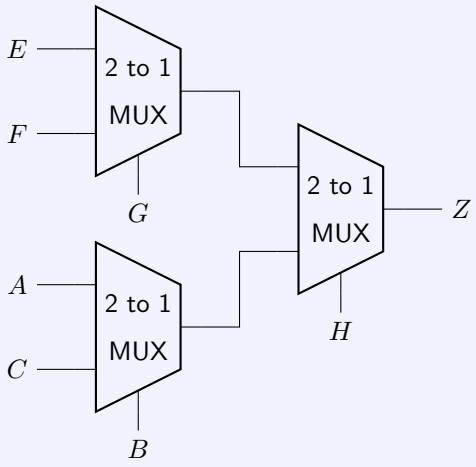
$$Z = AB'H' + BCH' + EG'H + FGH$$

In this case, the variable H exists in either true or complemented form in each term, so it will be used as the control bit on a 2 to 1 MUX.



This does not appear to be much of an improvement over just building this with an SOP implementation! Fortunately, we can look for variables that exist in true and complemented form in the

remaining MUX inputs. Input I_0 on the 2 to 1 MUX is connected to $EG' + FG$. G would therefore make a good candidate as a control bit on another 2 to 1 MUX. Input I_1 on the 2 to 1 MUX is connected to $AB' + BC$. Therefore B would make a good candidate for a control bit.



10.3 Sourcing vs. Sinking Current

Before we continue our discussion of logic devices, it is important to discuss the concept of sourcing and sinking current.

In active HIGH output devices, a one is the ON state and a zero is the OFF state. In these active HIGH output cases, an LED can be connected between the output of the digital logic and ground, as shown in Figure 10.20. When the output is a one, current flows and the LED turns on. When the output is a zero, current does not flow and the LED remains off. In this case, current must be sourced by the digital logic in order to turn the LED on.

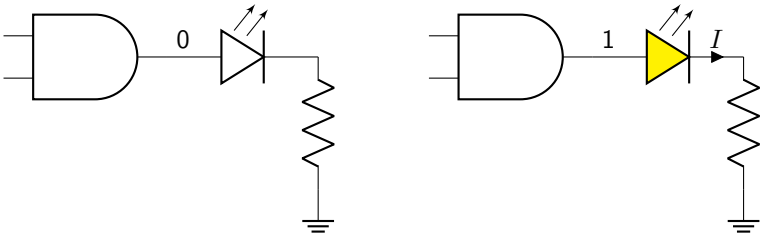


Figure 10.20: A digital logic gate sourcing the current to illuminate an LED.

In active LOW output devices, a zero is the ON state and a one is the OFF state. In this case, an LED can be connected between the output of the digital logic and Vcc, as shown in Figure 10.21. When the output is a zero, current flows and the LED turns on. When the output is a one, current does not flow and the LED remains off. In this case, current is sourced by the power supply and is sunk through the digital logic.

Sometimes, digital logic devices are not capable of sourcing very much current. In those cases, an active

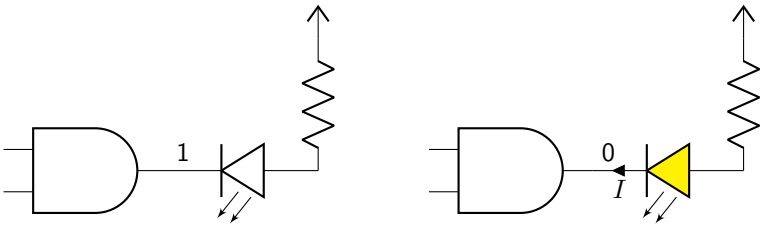


Figure 10.21: A digital logic gate sinking the current to illuminate an LED.

LOW output configuration would be much preferred, in which case the power supply can directly source the current required to turn the LED on. Digital logic devices are generally much better suited to sinking current than sourcing it.

Because of this characteristic of digital logic devices, we will notice that many devices such as demultiplexers and decoders have active LOW output configurations.

10.4 Demultiplexers (DEMUX)

A demultiplexer (DEMUX) is the exact opposite of a multiplexer (MUX). It routes one input signal to n different outputs. The output that is controlled by the input depends on the value of the control bits. There are n control bits required for 2^n output signals.

The circuit symbol for a 1 to 4 DEMUX is shown in Figure 10.22. The input is I, the control bit inputs are A and B, and the four outputs are Z_0, Z_1, Z_2, and Z_3.

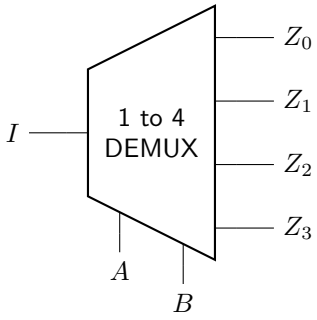


Figure 10.22: Circuit symbol of a 1 to 4 demultiplexer.

The choice of control bits changes the connections internal to the DEMUX to determine which output receives the input signal. This is depicted schematically in Figure 10.23.

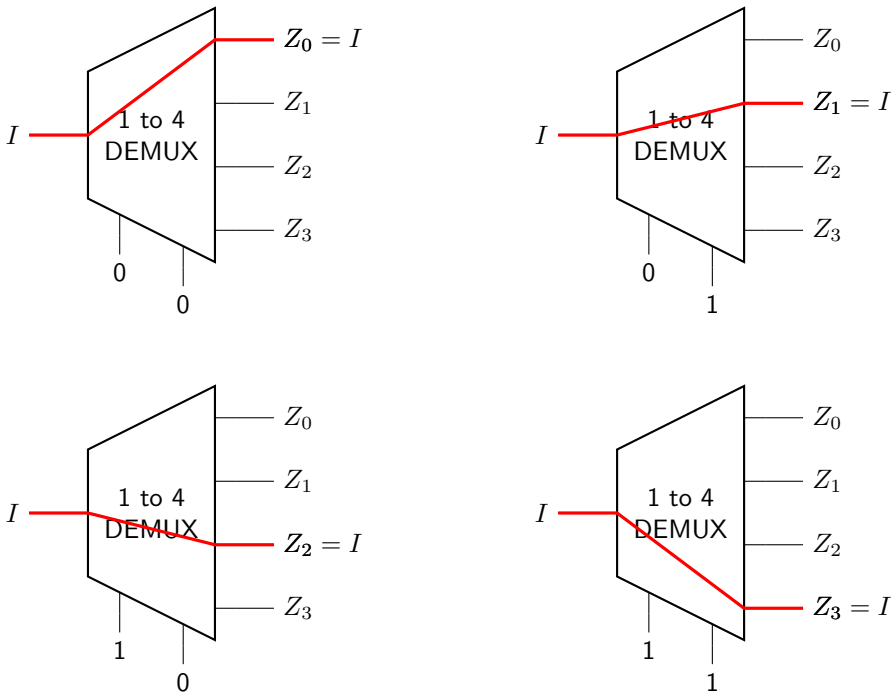


Figure 10.23: Connections between input pin and the output pins in a 1 to 4 DEMUX depend on the value of the control bits.

We can define the operation of a DEMUX using Boolean algebra expressions or a truth table. In this case, each output is defined by its own Boolean algebra expression, given below.

$$Z_0 = A'B'(I)$$
$$Z_1 = A'B(I)$$
$$Z_2 = AB'(I)$$
$$Z_3 = AB(I)$$

The truth table is given in table 10.2.

A	B	I	Z_0	Z_1	Z_2	Z_3
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	0	0	0	1

Table 10.2: Truth table for a 1 to 4 demultiplexer.

If we so desired, we could build a DEMUX out of AND gates, OR gates, and inverters. Such a circuit

diagram is given in Figure 10.24.

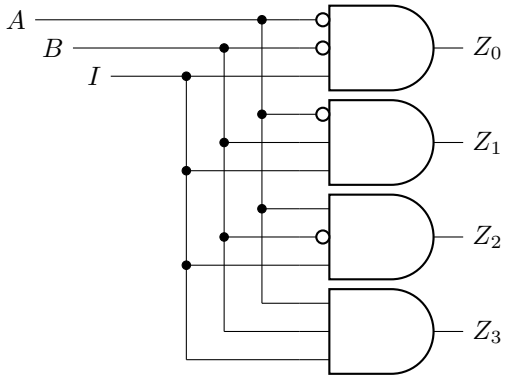


Figure 10.24: A 1 to 4 demultiplexer created from AND, OR, and NOT gates.

Circuit Project: Multiple 7-Segment Displays

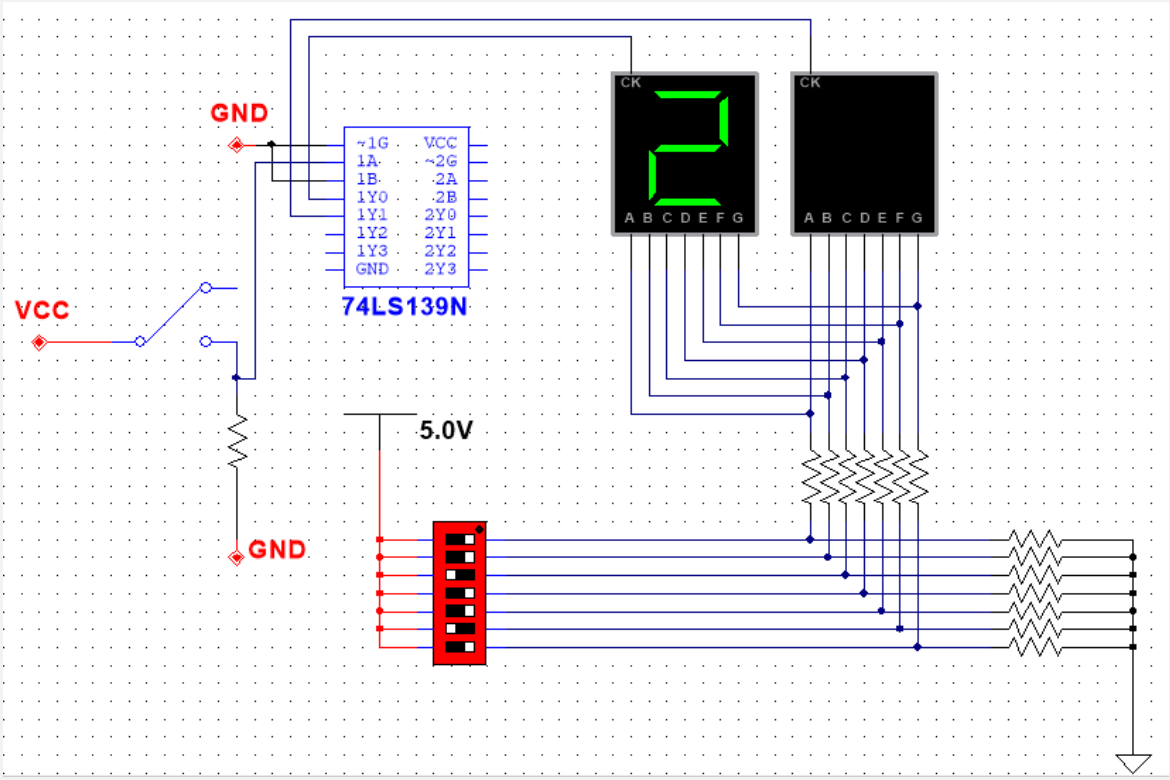
A circuit project that you can try to become familiar with demultiplexers is to use one to connect together two or more 7-segment displays. This particular schematic shows two displays connected together, but the DEMUX chip used is capable of connecting together up to four displays at once.

- 1– Breadboard
- 1– 5 V power supply or battery pack
- 1– DIP switch
- 1– toggle switch
- 15– 220 Ω resistors
- 2 to 4– 7-segment displays
- 1– 74LS139 dual 2-line to 4-line decoders/demultiplexers chip

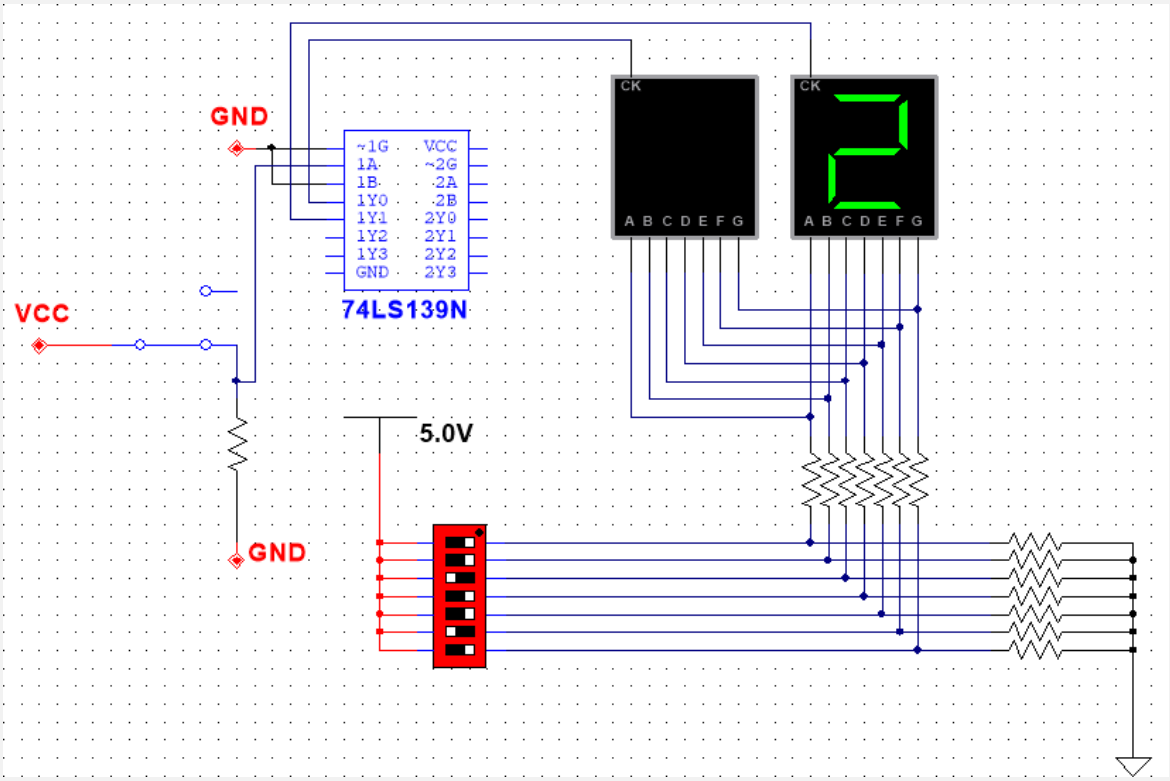
The major benefit of this configuration of displays is in the amount of wiring, resistors, and other components needed to connect them together. Only one signal (in this case coming from the DIP switch) needs to be supplied to the displays, and only one resistor is needed per segment as only one display is illuminated at a time. The toggle switch selects between which output is activated on the DEMUX, and those outputs are connected to each common cathode pin on the displays. The two images shown depict each individual display being controlled by the DEMUX.

It is important to note that the control bits on the 74LS139 chip use the opposite notation that we use in this book. Generally my preference is to consider A as the most significant bit. However, many digital logic chips use A as the least significant bit. Keep that in mind when inspecting the

following schematics. In addition, the outputs of the 74LS139 chip are active LOW output. Therefore an input of 0 is used to switch between an activated ON/OFF signal on each output.



When the toggle switch is in the up position, the output signal is LOW, which configures the control bits BA as 00. This sends the input signal (LOW) to output $1Y0$ (pin 4 on the 74LS139 chip). As this is an active-LOW output demux, the other outputs are all HIGH. Because a common cathode display requires a LOW signal on the common pin in order to work, only the display connected to $1Y0$ is illuminated. The other one is not.



When the toggle switch is in the down position, the output signal is HIGH, which configures the control bits BA as 01. This sends the input signal (still LOW) to output $1Y1$ (pin 5 on the 74LS139 chip). All of the other output pins are HIGH. In this case, only the display connected to $1Y1$ is illuminated.

If you wish to connect together more than two (and up to four) displays using the 74LS139 chip, a second toggle switch (or a DIP switch) can be used to select between each of the four outputs on the demux chip.

Note that many multi-segment displays are multiplexed together in this fashion, so demultiplexing is a common way to write to each of the display digits. This is more easily done using a microcontroller, but the basic technique is exactly the same as we would use with digital logic chips.

10.5 Using Multiplexers and Demultiplexers Together

It is possible to use a n to 1 MUX with a 1 to n DEMUX to route signals from one place to another using only one set of infrastructure. This is commonly used in telecommunications industries. An example schematic of how this would be achieved is shown in Figure 10.25.

If the switching between customer signals occurs very rapidly, no interruption in the signal is perceived on the receiving end by the customer.

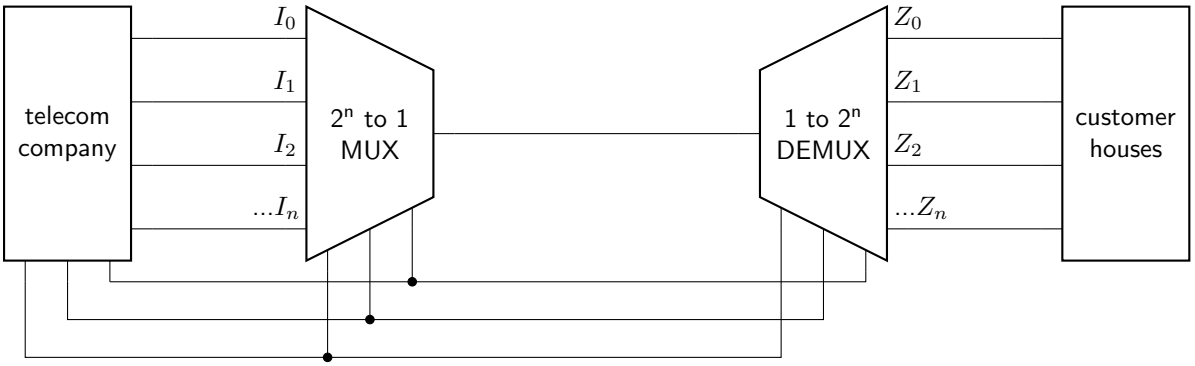


Figure 10.25: Use of a MUX and a DEMUX together to send multiple signals through one wire.

10.6 Decoders

A decoder takes n inputs and accordingly sets the value of 2^n outputs. In an active HIGH output decoder, all outputs will be zero except for the one output corresponding to the decimal value of the input. The circuit symbol for a 3 to 8 decoder is shown in Figure 10.26. Its truth table is given in table 10.3.

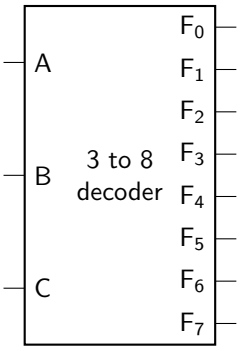


Figure 10.26: Circuit symbol of an active HIGH output 3 to 8 decoder.

A	B	C	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Table 10.3: Truth table of an active HIGH output 3 to 8 decoder.

Note how if the inputs are 011, which is the decimal number three, then output F_3 is activated with the other outputs deactivated. The fact that only one output is activated at a time makes a decoder a very useful logic device. They can be used to implement Boolean functions as well as to address individual elements in memory (which will be discussed later in this chapter in the section about memory) or to create large

multiplexer elements.

10.6.1 Implementing Boolean Algebra Expressions with a Decoder

Decoders are very useful in realizing Boolean functions because, in the case of an active HIGH output decoder, each output corresponds to a minterm. As shown in Figure 10.27, each output is asserted HIGH when the corresponding minterm is available on the input of the decoder.

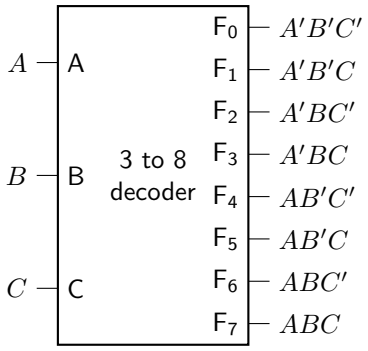


Figure 10.27: The outputs of a 3 to 8 decoder correspond to three variable minterms.

If the decoder has an active LOW output configuration, then each output corresponds to a maxterm. That is, each output is asserted LOW when the corresponding maxterm is available on the input of the decoder. This is depicted schematically in Figure 10.28.

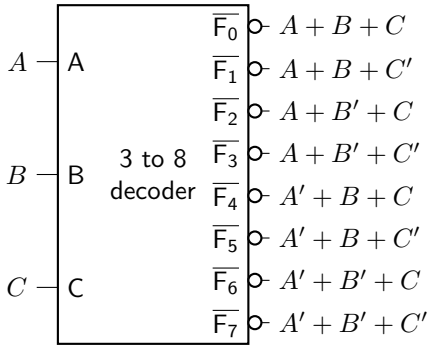


Figure 10.28: The outputs of a 3 to 8 decoder correspond to three variable maxterms.

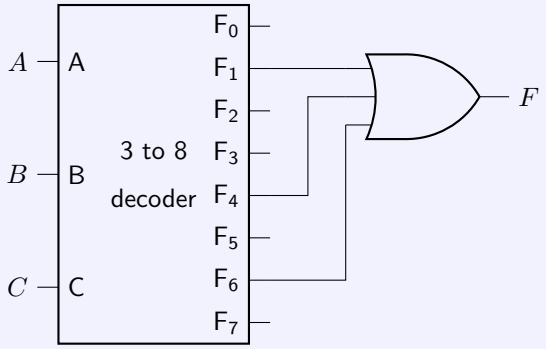
The best way to optimize this process is to make the following determinations.

1. Determine if the expression has fewer minterms or maxterms. Choose to implement the function with whichever has the fewest terms.
2. If there are fewer minterms
   - and the decoder has active HIGH outputs: implement the function with an OR gate, or
   - the decoder has active LOW outputs: implement the function with a NAND gate (which is equivalent to an OR gate with inverted inputs).

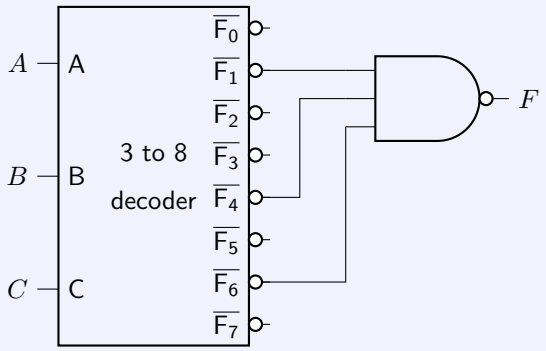
3. If there are fewer maxterms
   - and the decoder has active LOW outputs: implement the function with an AND gate, or
   - the decoder has active HIGH outputs: implement the function with a NOR gate (which is equivalent to an AND gate with inverted inputs).

Example: Implementing a function of three variables with a 3 to 8 decoder

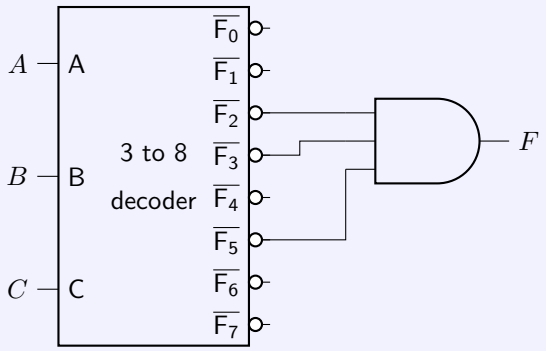
Implement the function $F(A, B, C) = \Sigma m(1, 4, 6) + \Sigma d(0, 7)$ using a 3 to 8 decoder. Note that there are equal numbers of minterms and maxterms, so either a minterm or maxterm implementation would work equally well. The first schematic shows an implementation of minterms using an active HIGH output decoder. In that case, the minterms are connected to an OR gate.



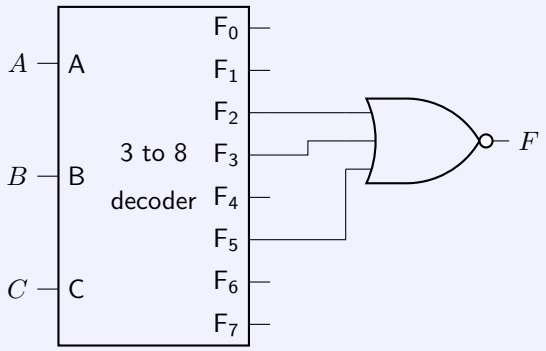
If minterms are implemented with an active LOW output decoder, then the minterms are connected through a NAND gate.



If the maxterms are implemented with an active LOW output decoder, then the maxterm outputs are connected through an AND gate.

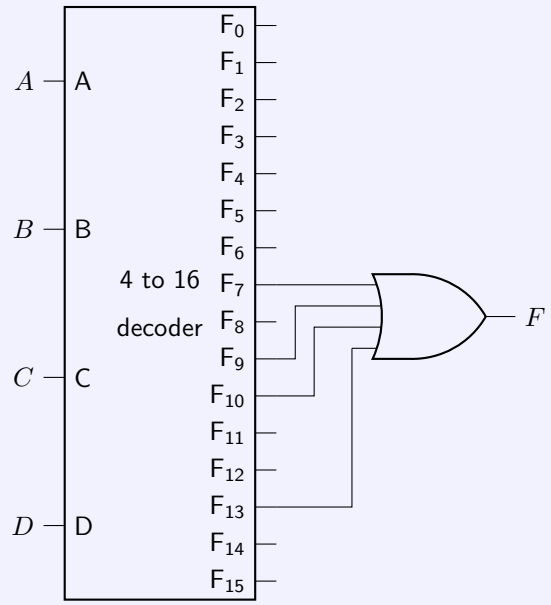


Lastly, if the maxterms are implemented with an active HIGH output decoder, then the maxterm outputs are connected through a NOR gate.

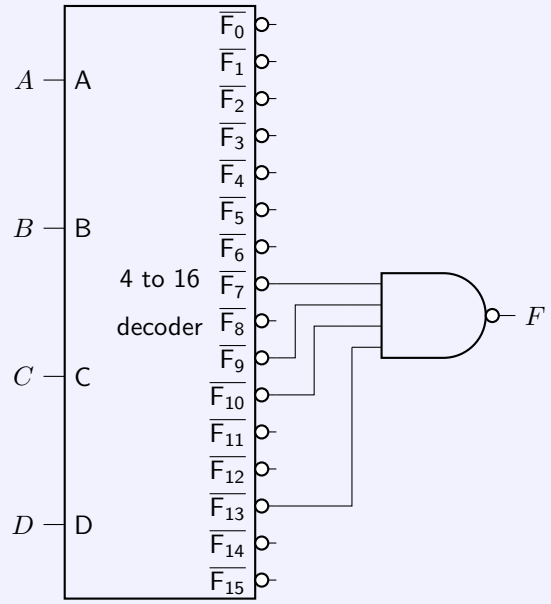


Example: Implementing a function of four variables with a 4 to 16 decoder

Implement the function $F(A, B, C, D) = \Sigma m(7, 9, 10, 14) + \Sigma d(0, 2, 4)$ using a 4 to 16 decoder. In this case, there are far fewer minterms than maxterms, so a minterm implementation would be optimal. If the decoder has active HIGH outputs, then an OR gate is used to connect each minterm.



If the decoder outputs are active LOW, then a NAND gate is used to connect the minterms together.



10.6.2 Expanding Decoders

Just as many-input MUXes can be created by connecting together MUXes with fewer inputs, decoders can also be expanded. For example, if a 3 to 8 decoder is required, but only 2 to 4 decoders are available, they can be connected in such a way that the inputs and outputs are expanded.

Two identical n to 2^n decoders can be connected together with an inverter to generate one $n+1$ to 2^{n+1} decoder. This is done by using the enable bit to effectively create one more input. The decoder that will be

enabled when the enable input is LOW will contain the least-significant outputs. The decoder enabled when the input is HIGH will contain the most-significant outputs. Two 2 to 4 decoders expanded to a single 3 to 8 decoder is shown in Figure 10.29.

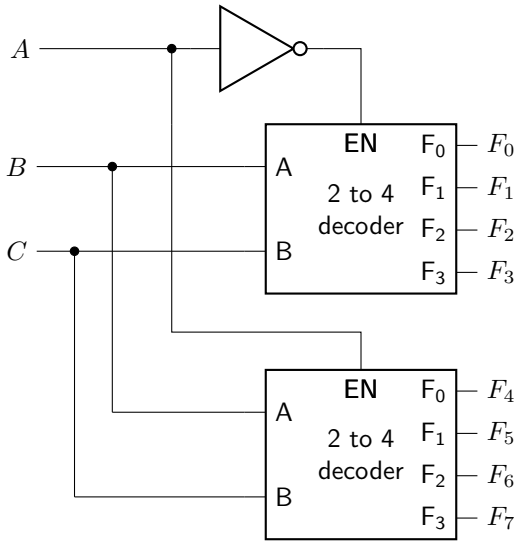


Figure 10.29: Two 2 to 4 decoders are expanded to create one 3 to 8 decoder.

Another possibility is to connect one n to 2^n decoder with 2^n identical m to 2^m decoders to create one $(n+m)$ to 2^{n+m} decoder. The outputs of the single n to 2^n decoder are used to enable each of the m to 2^m decoders. One 2 to 4 decoder and four 3 to 8 decoders are expanded to a single 5 to 32 decoder in Figure 10.30.

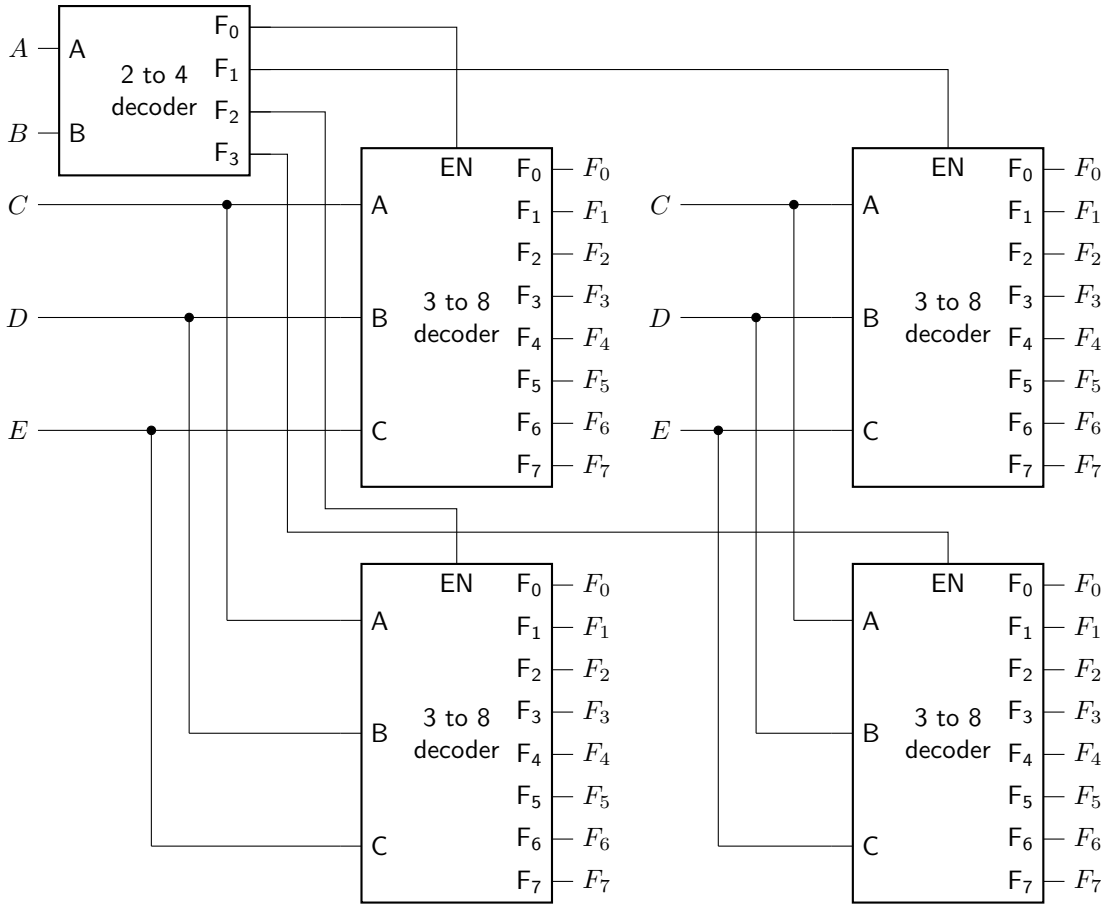


Figure 10.30: One 2 to 4 decoder and four 3 to 8 decoders are expanded to create one 5 to 32 decoder. (Not shown are connections between both C inputs, both D inputs, and both E inputs.)

10.6.3 Creating Multiplexers out of Decoders

A MUX can be implemented using a decoder and tri-state buffers. Because of the fact that only one output is asserted at a time, we know that only one tri-state buffer will be enabled and therefore electrically connected to the output, ensuring that there is no tug-of-war occurring between the outputs of each tri-state buffer. A schematic of an 8 to 1 MUX created in such a manner is given in Figure 10.31. In the schematic, the data inputs are I_0 through I_7, the control inputs are A, B, and C, and the output is F.

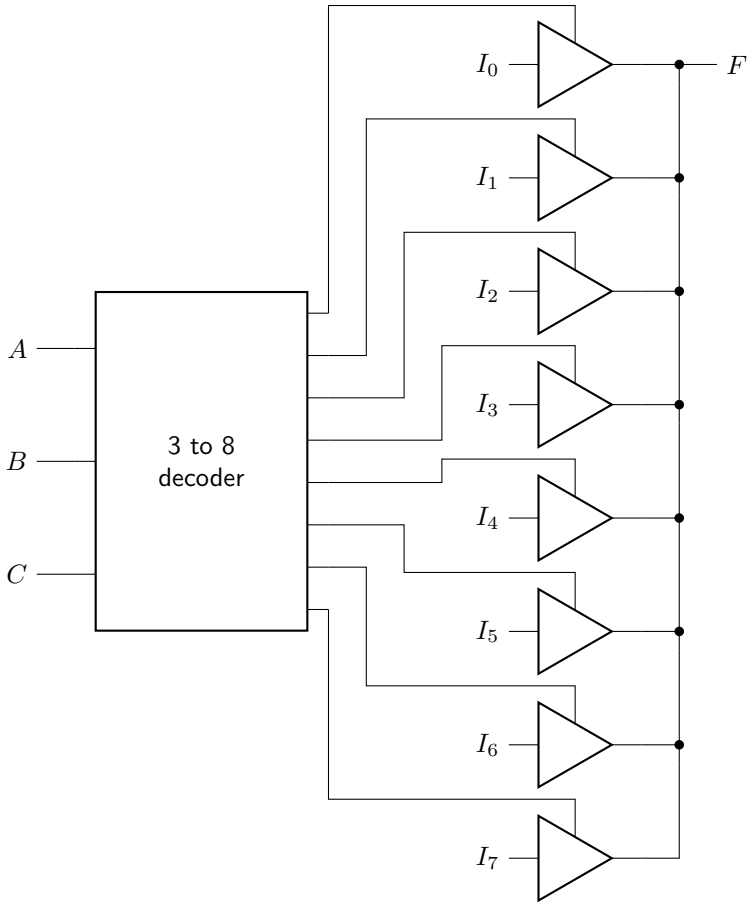


Figure 10.31: An 8 to 1 MUX created from a 3 to 8 decoder and tri-state buffers.

10.6.4 Speciality Decoders

There are speciality decoders that exist to perform specific useful operations. For example, there are BCD to 7-segment decoders that take a 4-bit BCD input and convert it to the proper 7-bit output that is required to create that digit on a 7-segment decoder.

The 74LS48 digital logic chip is an active HIGH output decoder that is used with common cathode 7-segment displays to create numerals from zero through nine. The truth table of this chip is shown in table 10.4. The inputs are $DCBA$, where D is the most significant bit. The outputs are for segments a through g. Only the BCD inputs are shown.

D	C	B	A	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	0	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1

Table 10.4: Truth table of the 74LS48 BCD to 7-segment decoder chip.

Each of these inputs causes the segments to light up as shown in Figure 10.32.

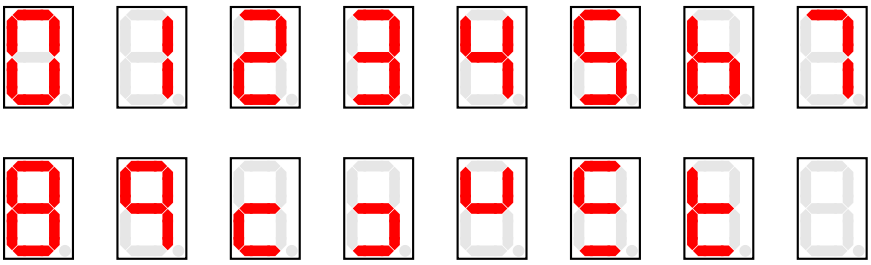


Figure 10.32: Numeral displays created by the 74LS48 BCD to 7-segment decoder chip.

10.7 Encoders

Encoders are the exact opposite of decoders. There are 2^n inputs and n outputs. In an encoder, whichever input number is asserted causes the output to take on the binary value of that input. For example, if I_4 is asserted, the output would be 100.

A circuit symbol for an 8 to 3 encoder is shown in Figure 10.33.

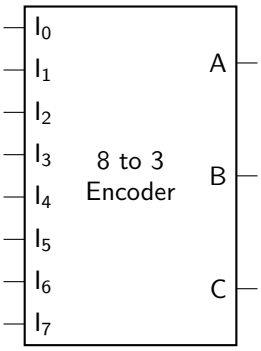


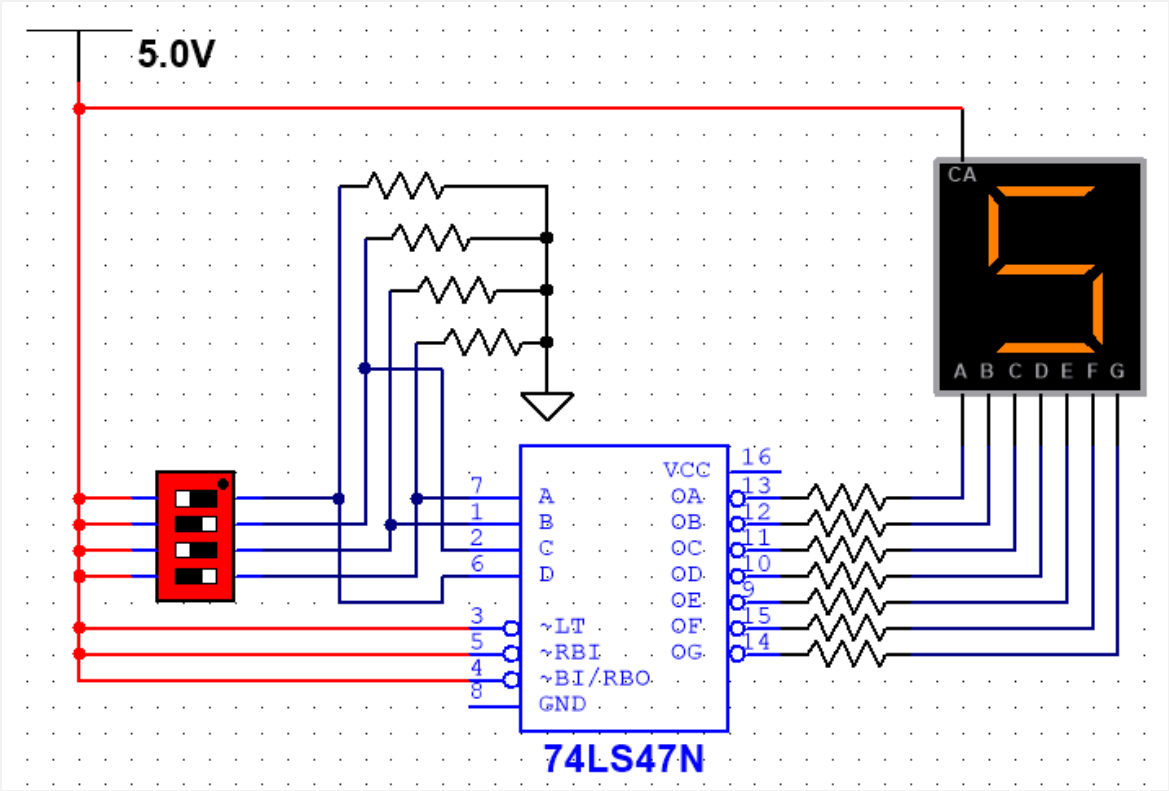
Figure 10.33: Circuit symbol of an 8 to 3 encoder.

Circuit Project: Using a BCD to 7-Segment Decoder

This project uses the 74LS47 BCD to 7-segment decoder chip to illuminate a 7-segment display.

- 1– Breadboard
- 1– 5 V power supply or battery pack
- 1– DIP switch
- 1– toggle switch
- 11– 220 Ω resistors
- 2– 7-segment display, common anode
- 1– 74LS47 BCD to seven-segment decoder/driver chip

The 7-segment display will now automatically light up based on the BCD value present on the input pins. This enables us to work with BCD values rather than having to decode every segment. Note that the 74LS47 chip uses D as the MSB, rather than A as is the preference used in this book. In addition, the outputs are active LOW. This means that a signal of zero turns on the segments, which means that a common anode display must be used.



Encoders by themselves cause a couple of issues. First, what distinguishes between having I_0 asserted and having none of the inputs asserted? In either case, the output pins would all read zero. To differentiate between these two cases, another output pin is included that is asserted when any one of the inputs is asserted. (This enables us to distinguish between I_0 asserted and nothing asserted, as shown in Figure 10.34.)

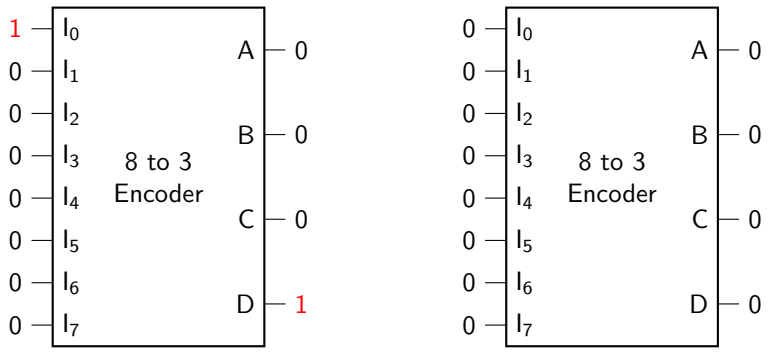


Figure 10.34: Schematic of 8 to 3 encoders, showing the purpose of the data output pin D.

The next issue with an encoder is the question of what happens when more than one input is asserted simultaneously. If both I_1 and I_4 are asserted, will the output be 001 or 100? To solve this problem, we need to introduce the concept of a priority encoder.

10.7.1 Priority Encoders

A priority encoder is an encoder where one of the inputs takes priority over all others. The priority input could be the largest (I_7 in an 8 to 3 encoder) or the smallest (I_0). In an 8 to 3 priority encoder, if I_7 has priority, this means that the largest input that is asserted will determine the output of the device. All other asserted inputs with smaller values will be ignored. This is depicted schematically in Figure 10.35. In this 8 to 3 priority encoder, input I_7 has priority, which means the largest asserted input will dictate the binary output value.

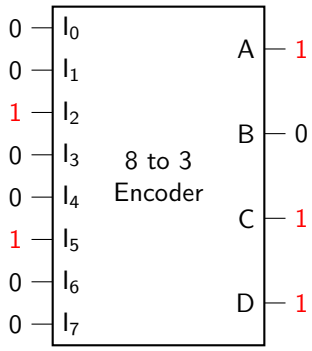


Figure 10.35: Schematic of an 8 to 3 priority encoder with multiple asserted inputs.

The truth table for an 8 to 3 priority is given in table 10.5. The input with priority is I_7.

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	A	B	C	D
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
×	1	0	0	0	0	0	0	0	0	1	1
×	×	1	0	0	0	0	0	0	1	0	1
×	×	×	1	0	0	0	0	0	1	1	1
×	×	×	×	1	0	0	0	1	0	0	1
×	×	×	×	×	1	0	0	1	0	1	1
×	×	×	×	×	×	1	0	1	1	0	1
×	×	×	×	×	×	×	1	1	1	1	1

Table 10.5: Truth table of an 8 to 3 priority encoder.

10.7.2 Uses of Priority Encoders

Priority encoders have several uses. However, most of their uses correspond to their ability to turn a single asserted input into a binary output. A keypad encoder is used to create a binary output corresponding to the button that is pressed on a numeric keypad. The 74C922 logic chip is capable of encoding a keypad with up to sixteen pins.

Scaling up from the keypad encoder would be an entire alphanumeric keyboard. Your computer interprets a button press and converts it using an encoder to an ASCII character, which is then saved in memory.

Encoders can also be used in analog to digital converters. Generally, an analog to digital converter generates digital outputs that are not necessarily binary. A priority encoder can take the digital output and convert it to binary, which is necessary for performing calculations or comparisons in a computer or microcontroller environment. The cover of this textbook is in fact a 3-bit analog to digital converter that uses the 74LS148 priority encoder to generate a binary output.

The schematic shown in Figure 10.36 shows a 3-bit analog to digital converter. Each of the components on the left is a comparator. In a comparator, the output is a one if the input voltage is higher than some threshold. Those threshold voltages are shown inside each comparator component. (Analog comparators such as these are outside the scope of this book.) The output of these comparators are digital but not yet binary. They are fed into an 8 to 3 priority encoder, where the highest voltage comparator has priority. The binary output is then sent to a 7-segment decoder and to a 7-segment display.

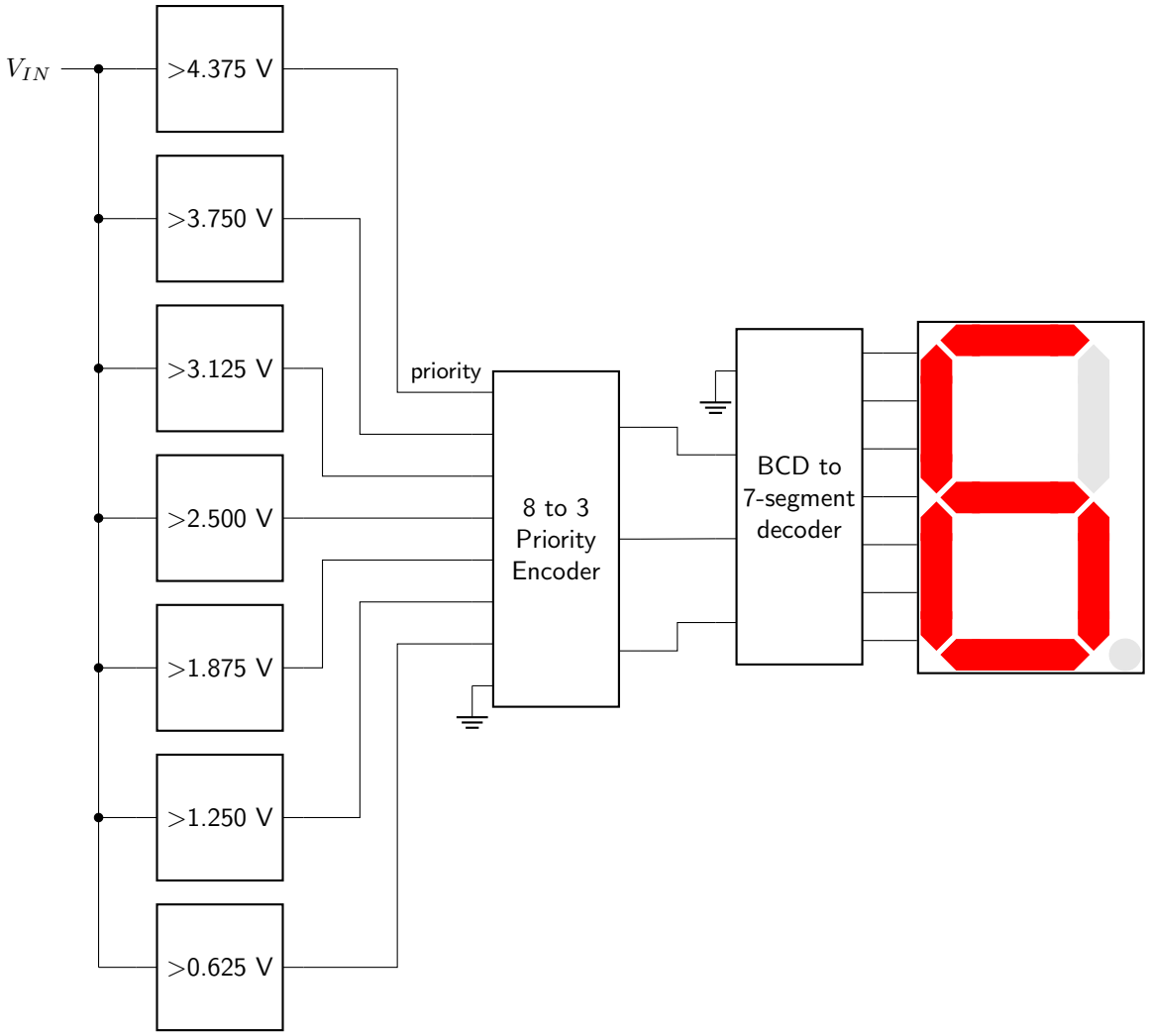


Figure 10.36: Schematic of a 3-bit analog to digital converter created from an encoder and decoder.

10.8 Memory

Memory is a device that stores information for immediate or later use. There are two major types of memory: volatile and non-volatile. This refers to whether or not data persists in the memory device when power is removed.

10.8.1 Volatile Memory

Volatile memory, commonly referred to as RAM (random access memory) in colloquial use, is a type of memory that does not persist when power is removed from the memory cell. That is, volatile memory can only store information when it is supplied with power.

In computing, RAM is typically used to store data memory. Your computer will move a program executable file (for a word processing software, or a game) into RAM and run it from there, as RAM is typically a fast access type of memory. The two types of RAM commonly found in computing and microcontrollers

are SRAM and DRAM.

SRAM stands for static RAM, and consists of flip-flop type storage elements (that we will discuss in the next chapter in this book). SRAM tends to be faster than DRAM but takes up much more physical space. The microcontroller used in the popular Arduino Uno uses SRAM for data memory.

DRAM stands for dynamic RAM, and consists of a capacitor that temporarily stores data. Because it takes some time for capacitors to charge up and to discharge, DRAM is usually slower than SRAM, but takes up much less space. Of additional concern with DRAM is the fact that capacitors eventually discharge by themselves, and need to be rewritten on occasion in order to keep the data from erasing itself. DRAM is used in most consumer laptops and computers as it is much smaller and can therefore be included up to several gigabytes of storage.

10.8.2 Non-Volatile Memory

Non-volatile memory refers to types of memory where data can be stored even after power has been removed from the device. There are very many types of non-volatile memory, many of which are no longer in use today. Magnetic storage was used for a long period of time both in floppy discs and computer hard drives. CDs and DVDs are a type of optical storage (rather than magnetic or semiconductor based) that were commonly used in the late 20th and early 21st century to store data. These days, most non-volatile memory has moved from optical and magnetic storage to semiconductor storage, as it is much faster to read and write.

ROM is the most commonly used type of non-volatile memory as of the writing of this textbook. ROM stands for read-only memory. It is somewhat of a misnomer, in that ROM can very much be erased and rewritten these days. However, ROM generally requires much more effort to erase and rewrite than it does to read.

The first type of ROM that was created is referred to as mask ROM. This was created using the same photolithography process in which your computer processors are created. In this process, successive masks are used to pattern different layers in semiconductor materials. In this manner, ones and zeros were physically stored into semiconductors using connections to power and ground. There are some mask ROM chips that could be custom programmed by Texas Instruments using punched cards in the 7400 series of TTL logic devices. This was useful for storing information in a permanent manner, but after the data was written, there was no way to rewrite the chip. Therefore it was not the most efficient manner of creating a prototype device.

The next type of ROM that was developed is called PROM, or programmable ROM. PROM is created with connections that can be broken as needed. Every output in the memory starts as a one, and as needed an irreversible vaporization process can occur by sending a lot of current into particular areas in the chip to remove connections to power and thereby create zeros. PROM is also known as one-time programmable (OTP) EPROM, and can still be purchased today from manufacturers such as Microchip (formerly Atmel).

PROM is a very useful type of non-volatile data storage once a prototype has been created and the memory is known to be functioning properly.

Erasable PROM (EPROM) has the benefit of being erasable and reprogrammable. The first types of EPROM chips created are UV erasable. There is a quartz window that exposes the semiconductor material underneath. Upon exposure to high intensity UV radiation for a long enough period of time, all of the connections will reset to a value of one. EPROM is very useful in creating a prototype to ensure that things work before moving to OTP ROM. However, it requires physical removal from a circuit to be erased and reprogrammed. UV erasable PROM is shown in Figure 10.37. You can see the quartz window with the wire interconnects connecting to each of the chip's pins.

Figure 10.37: UV erasable programmable ROM. (Photograph by the author.)

Electrically erasable PROM (EEPROM) is a very commonly used type of non-volatile memory used today. It can be reprogrammed electrically, rather than with UV radiation. EEPROM and flash memory are almost interchangeable. This type of memory is used in most modern computers, laptops, and USB drives.

10.8.3 Implementing Boolean Algebra with ROM

One of the major benefits of ROM is that it acts like a truth table. This enables us to store zeros and ones at different locations in the ROM in order to implement Boolean functions without having to do any minimization techniques (such as k-maps or Quine-McCluskey).

Each ROM chip contains multiple pieces of information, called words. The size of the word is generally eight bits or sixteen bits in ROM such as that shown above in Figure 10.37, but probably thirty-two bits or sixty-four bits in your computer. This word length tells us how many bits of information can be dealt with by a computer processor at any given time. In terms of implementing a Boolean function, having 8-bit words means that eight Boolean functions can be implemented on the chip simultaneously. A ROM with 16-bit words can implement sixteen Boolean functions simultaneously.

We say that ROM is $2^n \times m$-bit, where m is the word length, and 2^n refers to the number of words that

can be stored on the chip. The 27C64 chip shown in Figure 10.37 is an 8K $\times$ 8 chip, as it is capable of storing 8,192 words of 8-bit length. If there are 2^n words stored on a ROM chip, we require n address pins to access each of those pieces of information. For this reason, a decoder is an integral part of a memory chip; the decoder enables us to use n bits of input to individually select between (address) 2^n pieces of data inside of the device, and output only that piece of data. A schematic of a 4 $\times$ 8 ROM using a 2 to 4 decoder to address each of the memory elements is shown in Figure 10.38.

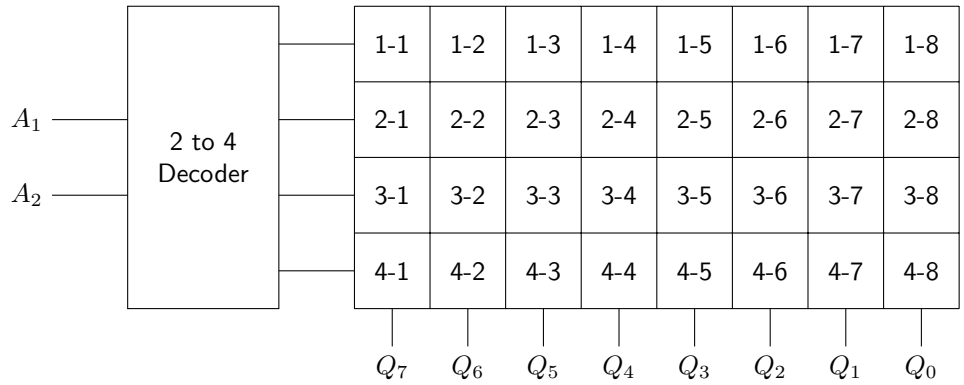


Figure 10.38: A 4 $\times$ 8 ROM chip. A decoder is used to address each of the individual words.

Going back to the truth table analogy, a ROM enables us to have a 2^n row truth table, able to implement m functions of n input variables. This makes it an unbelievably useful device. Rather than having to optimize many Boolean algebra functions and implement them with digital hardware, a ROM chip can be programmed that implements up to m Boolean algebra functions with only one single chip.

Sequential circuits (which will be described later in this textbook) can be implemented with a ROM chip and flip-flops. Each of the next-state equations are connected to flip-flop outputs; those outputs are then fed back to the ROM as inputs. The output equations can also be programmed in to the ROM. This makes ROMs a very versatile piece of hardware. A sequential circuit using a ROM chip is depicted in Figure 10.39.

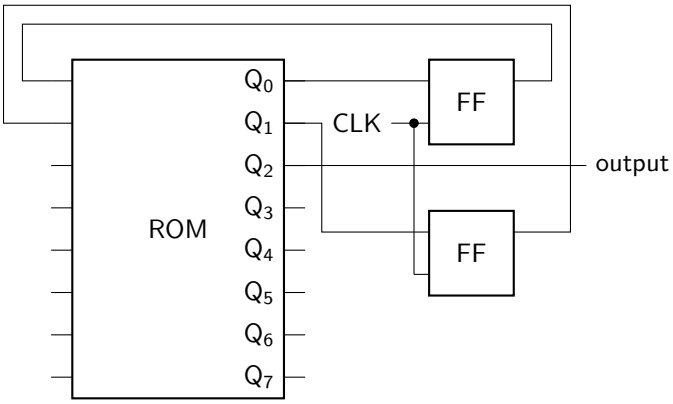


Figure 10.39: Implementation of a sequential circuit using a ROM chip and flip-flops (labeled FF in this schematic). The signal labeled CLK depicts a clock input to each flip-flop.

Programming an EPROM chip usually requires an external programming device. The TL866II is a commonly used programmer capable of programming many different types of ROM and microcontroller chips. A programmer is a very useful component of any hobbyists toolkit.

10.9 Programmable Logic

Programmable logic is another way to implement Boolean algebra functions without having to build them using digital logic chips. There are simple programmable logic devices (SPLDs) and complex programmable logic devices (CPLDs). The scope of this book is unfortunately not large enough to include CPLDs, so we will instead focus on SPLDs.

SPLDs are simply digital integrated circuits that can be programmed to provide a variety of logic functions. Technically, ROM is a type of SPLD, as it is capable of carrying out Boolean algebra functions. In this section of the book we will discuss PAL and GAL SPLDs.

10.9.1 Programmable Array Logic (PAL) and Generic Array Logic (GAL)

Programmable array logic, or PAL, is an integrated circuit chip where there are multiple programmable AND gates connected to a fixed OR gate. They are one-time programmable with a fuse generating connections between input signals and the AND gates.

Generic array logic, or GAL, is essentially just PAL that can be reprogrammed. In a way, it is similar to the difference between PROM (PAL) and EPROM (GAL). Most GAL can be electrically reprogrammed. The advantage of GAL is that it is possible to reprogram the output in case of mistakes, to improve functionality, or to change how the device works.

The programmable AND array is usually specified by its size $m \times n$, where m is the number of AND gates included in the chip, and n is the number of connections that each AND gate is capable of making. Connections between the input signals and the AND gates are created when the device is programmed, and creates the Boolean functions that are to be implemented on the output. An example of a 3×4 programmable AND array is shown in Figure 10.40.

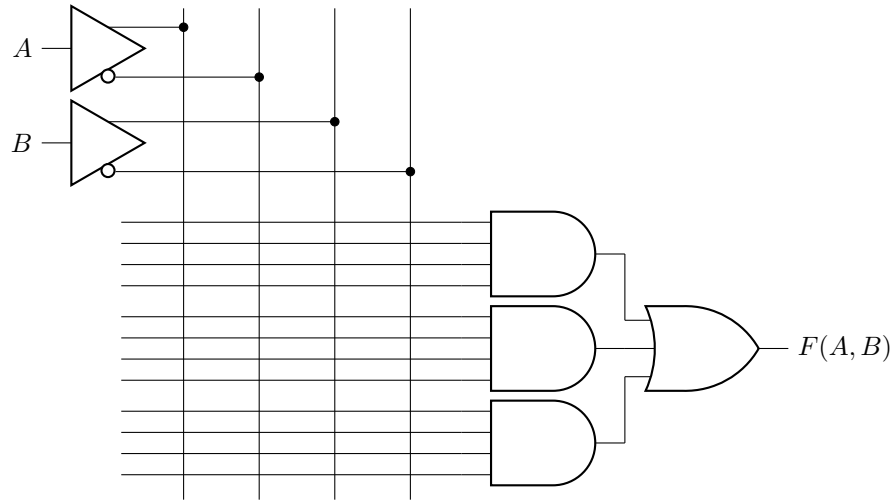


Figure 10.40: A 3 × 4 programmable AND array in a PAL/GAL.

Note that each input variable is complemented inside of the GAL/PAL, so external inverters are not required in their use. (This is one of the reasons that we did not consider inverters when we determined the lowest cost circuits earlier in this book!)

A simplified PAL/GAL diagram removes all of the input pins from each AND gate, and just shows it as a bus. If the bus has a size of eight, for example, that means up to eight connections can be made on the inputs to the AND gate. A simplified PAL/GAL digram with a 4 × 6 AND array is shown in Figure 10.41.

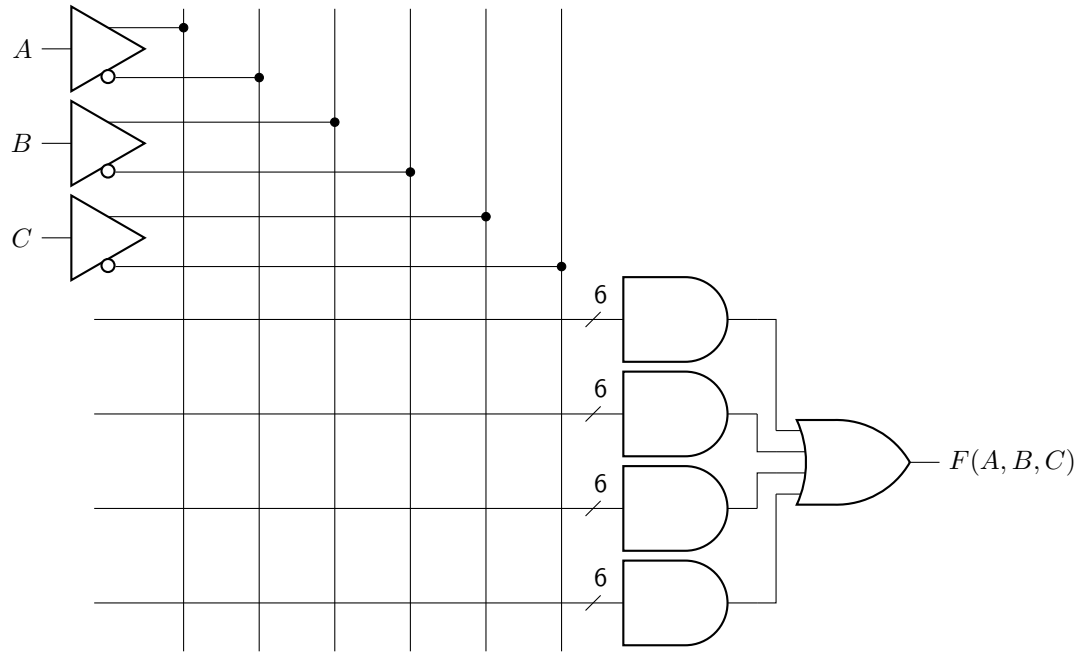


Figure 10.41: A 4 × 6 programmable AND array in a simplified PAL/GAL diagram.

Connections between inputs and the AND gates on a PAL/GAL diagram are indicated with an ×. The simplified diagram shown in Figure 10.42 shows how $F(A, B, C) = AB' + B'C' + A'BC$ could be implemented

in a 4×6 AND array.

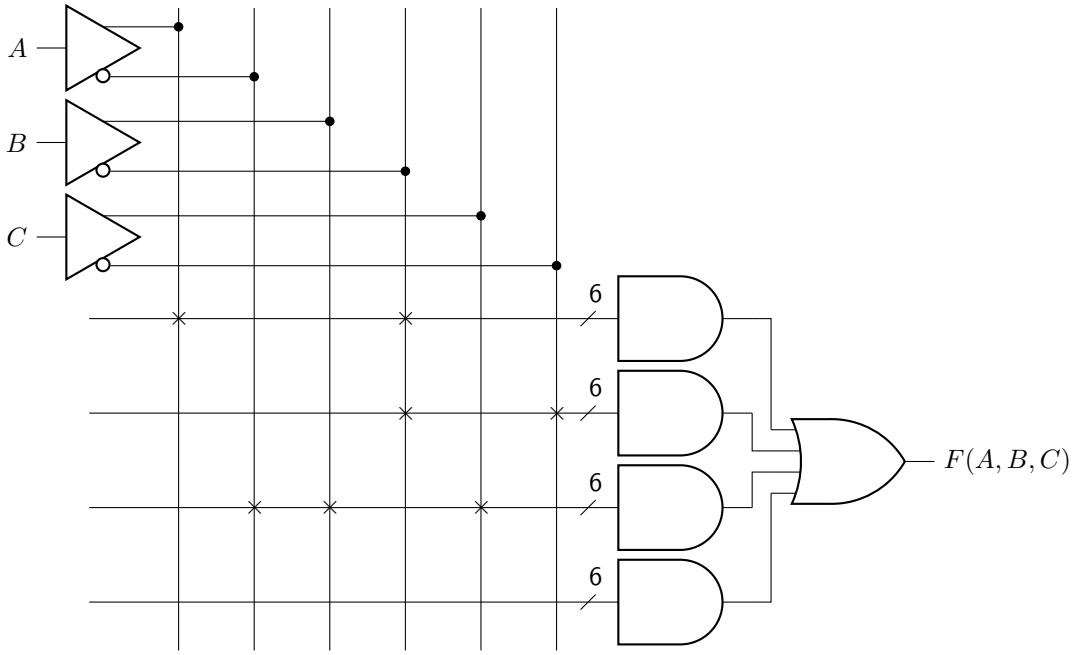


Figure 10.42: Implementation of $F(A, B, C) = AB' + B'C' + A'BC$ in a 4×6 programmable AND array.

PAL/GAL devices were very popular in the 1980s and 1990s, and are no longer widely used thanks to the ubiquity of inexpensive microcontrollers. However, some PAL/GAL devices continue to be manufactured today. These devices have a given number of input/output pins, say, sixteen. Of those pins, all can be used as input pins, but only a certain number (usually about half) can be used as output pins. In other words, there are complementing buffers and connections to the programmable AND gates available on every pin, but only some of the pins are connected to the output logic.

10.9.2 PAL/GAL Output Logic Macrocells

The simplified PAL/GAL diagrams shown so far show only fixed OR logic on the outputs. However, most PAL/GAL output logic is much more versatile than that. The output logic exists to allow us to make the outputs either active HIGH or active LOW, and can make the output combinational or synchronous. The exact options vary depending on the particular device that is used.

Combinational output logic uses an XOR gate to allow us to change the output logic from active HIGH to active LOW, if needed. An output enable can tri-state each of the output pins. This output enable is also deactivated if the output is instead used as an input pin (making it a bi-directional pin, as was discussed earlier in this chapter). A simplified example of a combinational output logic macrocell is shown in Figure 10.43.

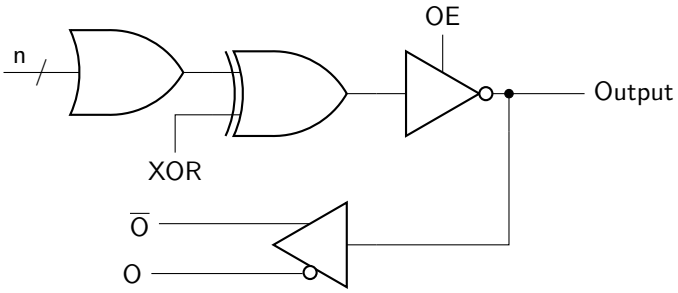


Figure 10.43: Combinational output logic macrocell.

Synchronous output logic uses a D flip-flop (which we will discuss in the next chapter in this book) to enable us to create synchronous devices with a PAL/GAL. There is generally a dedicated input pin that is used as the clock signal to synchronize each of the flip-flops. The XOR gate to create active HIGH and active LOW output conditions is still available. An example synchronous output logic macrocell is shown in Figure 10.44.

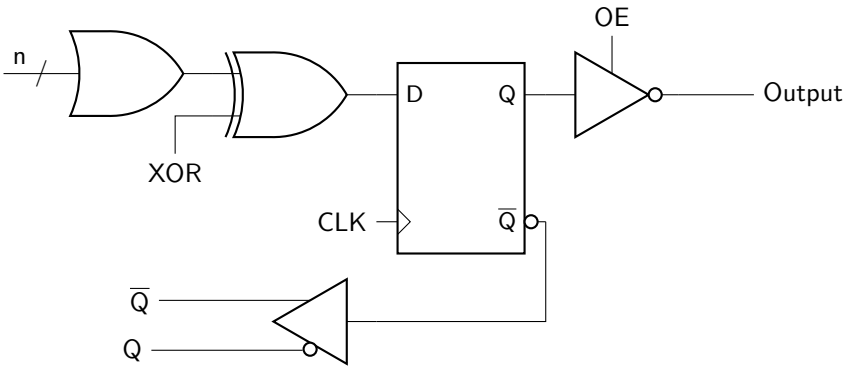


Figure 10.44: Synchronous output logic macrocell.

10.10 Example Problems

Multiplexers

1. Design a 3 to 1 MUX using only 2 to 1 multiplexers.

2. Implement the following expression with a 4 to 1 MUX and a minimum number of external gates. $F(A, B, C, D) = \Sigma m(0, 2, 7, 8, 10, 13, 14) + \Sigma d(3, 9, 12)$.

3. You see a MUX circuit wired up as shown in Figure 10.45. Find a MUX implementation of this circuit that requires a minimum number of external gates.

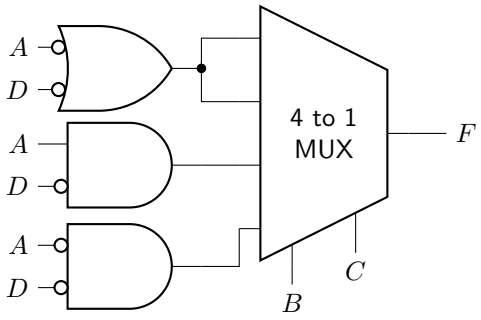


Figure 10.45: Circuit schematic corresponding to multiplexers question 3.

4. Draw a timing diagram for F from a 4 to 1 MUX, given the input signals (I_0–I_3) and control bit signals (A and B) shown in Figure 10.46. Ignore all gate delays.

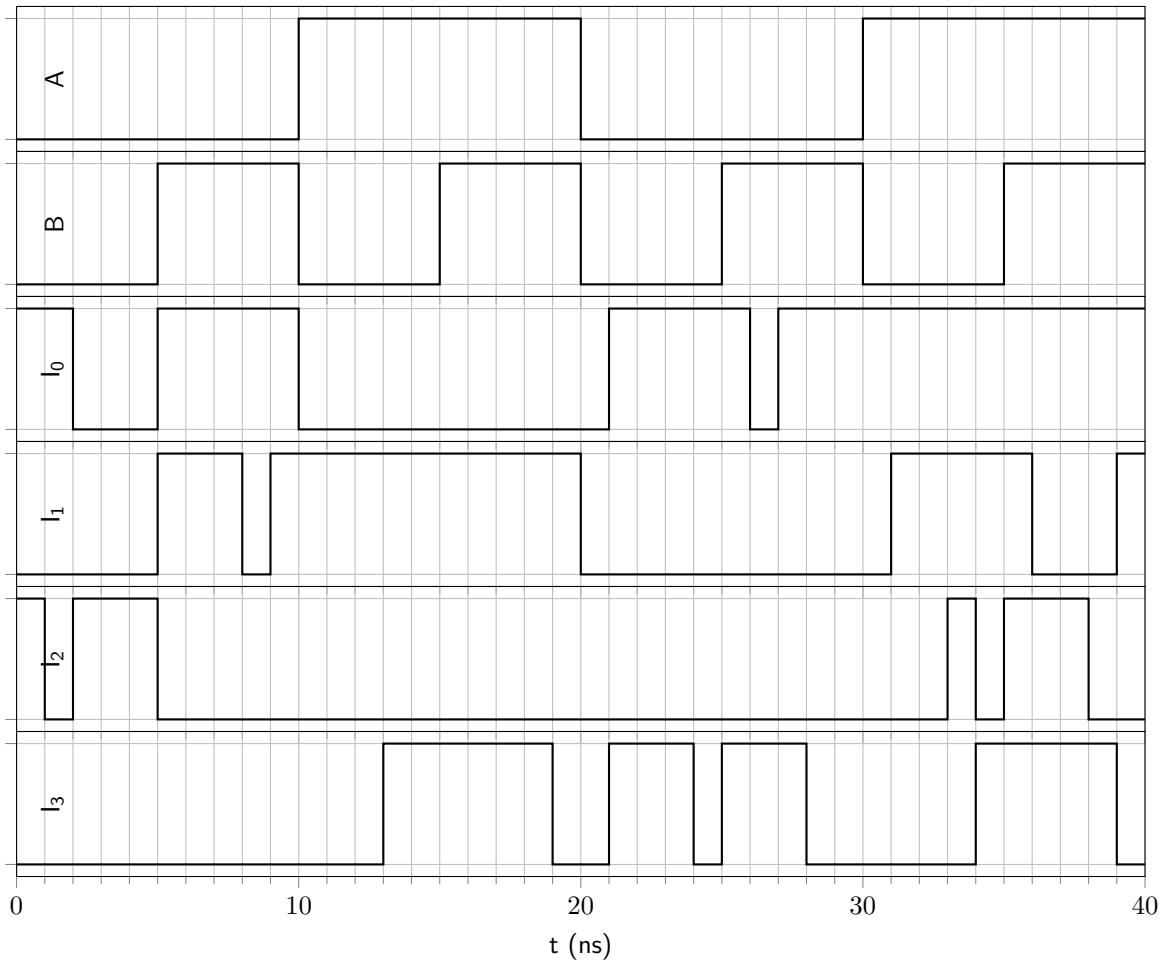


Figure 10.46: Timing diagram corresponding to multiplexers question 4.

5. You receive two 2-bit numbers designated as AB and CD. If $AB \geq CD$, an LED should turn on. The output is active HIGH. The output of this function, F, will therefore be 1 if the LED should be on. Implement this using a 4 to 1 MUX and a minimum number of external gates.

Demultiplexers

1. Create a timing diagram for each output of an active HIGH output 1 to 4 DEMUX given the input I and control bits A and B as shown in Figure 10.47.

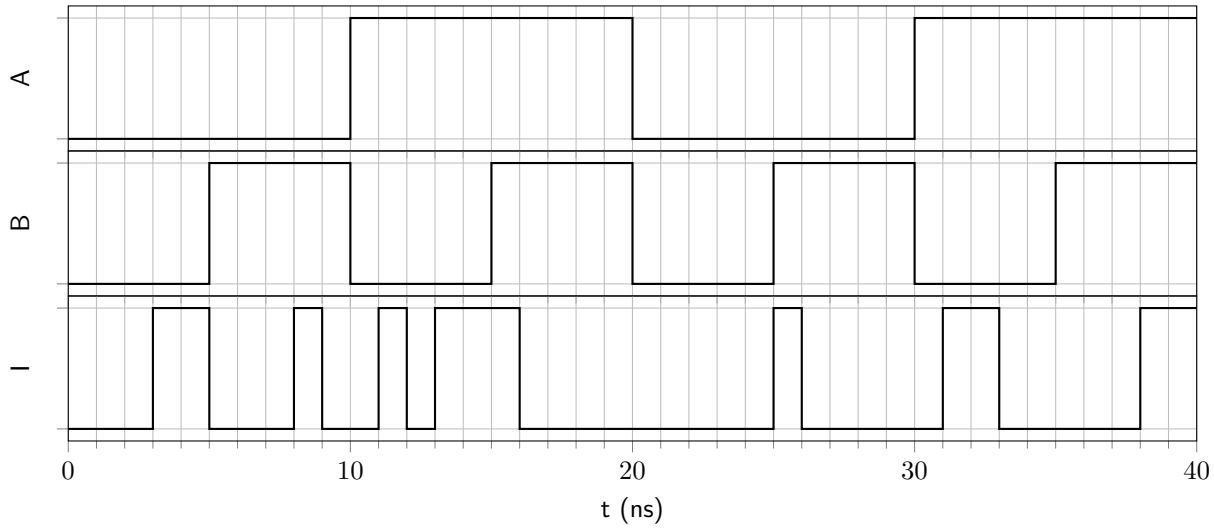


Figure 10.47: Timing diagram corresponding to demultiplexers question 1.

2. Design a 1 to 4 DEMUX using only 1 to 2 demultiplexers.
3. How many control bits would be required to implement a 1 to 24 DEMUX?
4. Derive expressions for each output of a 1 to 6 active HIGH output DEMUX. The control bits are A, B, and C.
5. Draw a schematic of a 1 to 4 active HIGH output DEMUX created from an active LOW enable, active LOW output 3 to 8 decoder.

Decoders

1. Implement the following function using an active HIGH output 4 to 16 decoder and a minimum number of external gates and inputs. $F(A, B, C, D) = \Sigma m(1, 3, 5, 7, 12, 14) + \Sigma d(4, 8, 11, 13)$
2. Implement the following function using an active LOW output 4 to 16 decoder and a minimum number of external gates and inputs. $F(A, B, C, D) = \Pi M(0, 4, 6, 10, 13, 14, 15)\Pi D(1, 7, 8, 12)$

3. You notice a 3 to 8 decoder wired up as shown in Figure 10.48. Label each of the outputs from F_0 to F_7 in the correct order based on the values of the control bits A, B, and C.

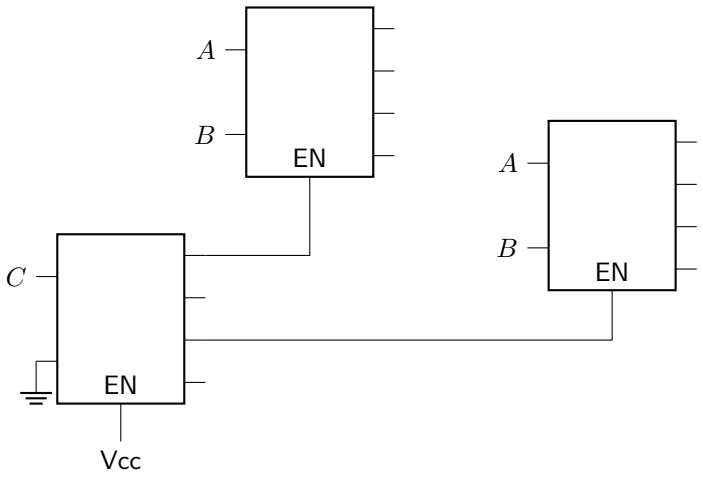


Figure 10.48: Decoder schematic corresponding to question 3.

4. Assume that you only have access to 74LS00 chips, which contain 2-input NAND gates. Use a 3 to 8 active LOW output decoder and a minimum number of 2-input NAND gates to implement $F(A, B, C) = \Sigma m(0, 2, 3) + \Sigma d(1)$

5. Design a speciality decoder that takes 3-bit binary numbers 0–5 and creates active HIGH outputs for a 7-segment display. What are the equations for each of the segments? Use an SOP implementation.

Encoders

1. An 8 to 3 priority encoder (I_0 has priority) has both I_3 and I_6 asserted on the inputs. What will the binary value of the output be?

2. Derive Boolean algebra expressions for the A, B, and $DATA$ outputs for a 4 to 2 active HIGH input, active HIGH output priority encoder where I_3 has priority.

3. Derive Boolean algebra expressions for the A, B, and $DATA$ outputs for a 4 to 2 active HIGH input, active HIGH output priority encoder where I_0 has priority.

4. The 74LS148 chip is an active LOW input, active LOW output, 8 to 3 priority encoder. What are the outputs $A_2A_1A_0$ when the inputs I_0–I_7 are 11010011? Assume that the chip is properly enabled. Consult the data sheet if you need help.

5. The 74LS147 chip is an active LOW input, active LOW output, 10 to 4 (BCD) priority encoder. What are the outputs $A_3A_2A_1A_0$ when the inputs I_0–I_9 are 1000000111? Assume that the chip is properly enabled. Consult the data sheet if you need help.

Memory

1. How many bytes of capacity does a ROM with 18 address pins and 8-bit words have?
2. How many bytes of capacity does a ROM with 13 address pins and 16-bit words have?
3. How many address pins would be required to address all of the words in a 65,536 × 8 ROM?
4. Use a 2 to 4 decoder and four 256 × 8 ROM chips to create a 1024 × 8 ROM chip. Assume the decoder is active LOW output, and the ROM chips are active LOW enable.
5. What would be the capacity in bytes of a ROM created from a 4 to 16 decoder and sixteen 1024 × 8 ROM chips?

Programmable Logic

1. Determine the Boolean expression given from the PAL/GAL diagram shown in Figure 10.49.

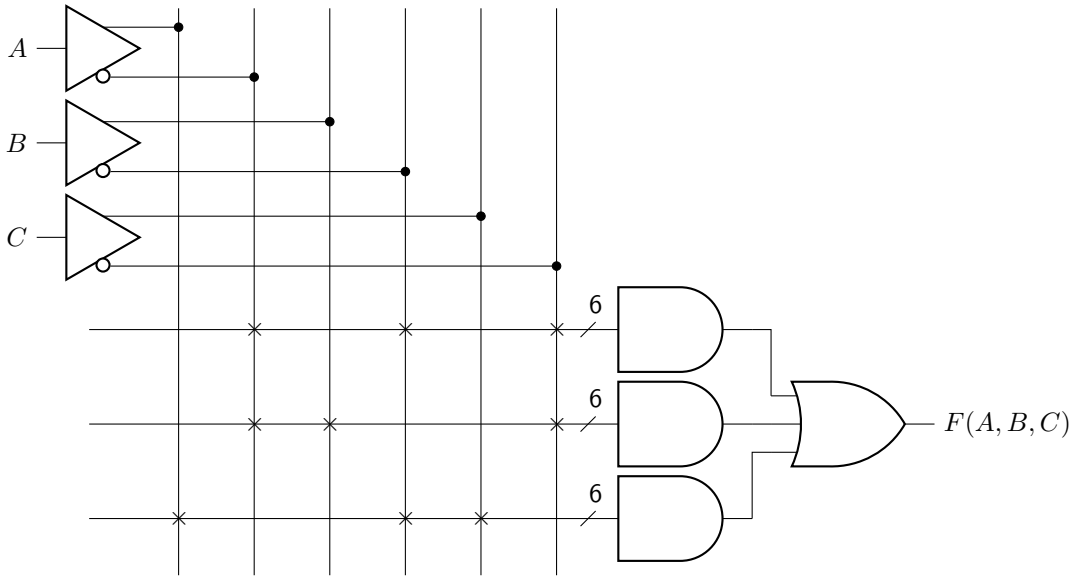


Figure 10.49: Simplified PAL/GAL diagram for question 1.

2. Draw a simplified PAL/GAL diagram to implement the expression $F(A, B, C) = AB'C + A'BC' + ABC$.
3. To create active LOW outputs, what should the input to the XOR gate in an output logic macrocell be?

4. The programmable logic device shown in Figure 10.50 has both programmable AND and programmable OR gates. Determine the output of the expression given in the circuit diagram.

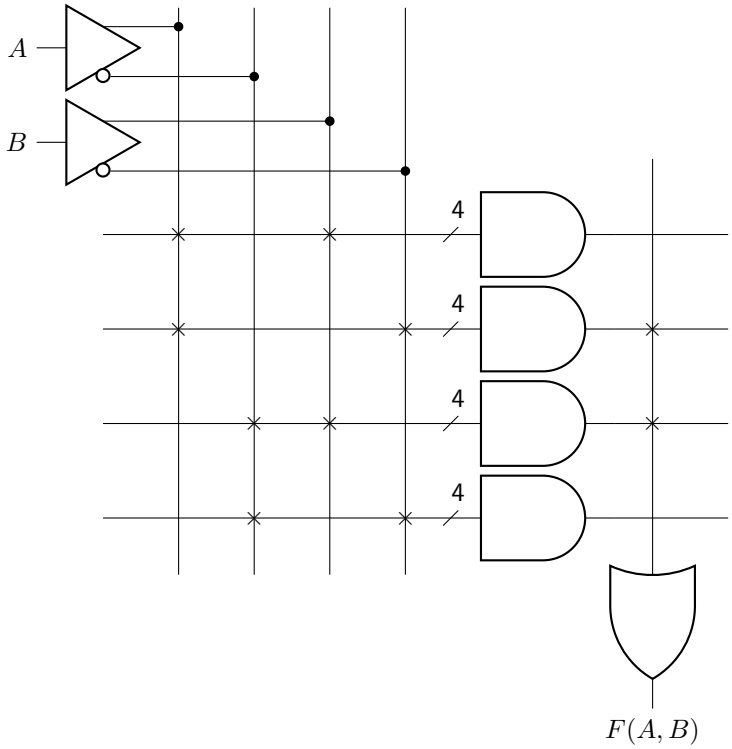


Figure 10.50: Simplified PAL/GAL diagram for question 4.

5. A sensor on a car tire sends a 5-bit binary signal ($ABCDE$) that represents the tire pressure in PSI. The output L (low pressure) should be 1 if the pressure is less than 19 PSI. The output P (puncture) should be 1 if the pressure is less than 3 PSI. Implement P and L using a simplified PAL/GAL diagram.

11 Sequential Circuits

The introduction of sequential circuits opens up a whole new world of digital circuits. The simple act of connecting the output of our logic chips to the input gives our circuits memory, enabling them to carry out a sequence of events, rather than simply giving a static output for a set of input signals.

A simple, but not terribly useful, sequential circuit is the OR latch. As shown in the schematic in Figure 11.1, one of the inputs is connected to a pushbutton, while the other input comes from the output of the OR gate. When power is initially supplied to the circuit, the output of the OR gate is zero. If the pushbutton is not pressed, then the output of the OR gate will remain zero. Once the pushbutton is pressed, the output of the OR gate will become one. At this point, with the pushbutton in its not pressed state, the output will remain a one until power is again removed from the circuit.

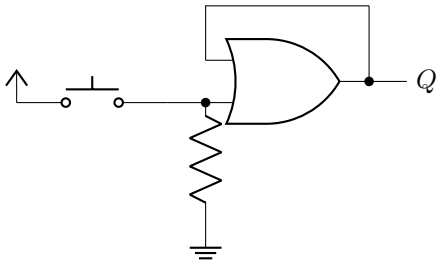


Figure 11.1: An OR latch.

While this may be subtle, the fact is that an OR gate always had one possible output when the inputs were in a certain state. This OR latch has two states: the output can be zero or one with the pushbutton not pressed. The new output depends on what the previous value of the output was!

To describe the operation of our OR latch, we can define what happens to the output when the input has a particular value. When the input is zero (the pushbutton is not pressed), the output keeps its old value. We call this a hold or latch situation. When the input is one (the pushbutton is pressed), the output becomes a one. We call this a set.

A latch is a digital circuit that has more than one output state and where the output changes immediately in response to a change in input. Because the output responds immediately to changes in the input values, we say that latches are asynchronous. We will learn about two important types of latches in the sections below.

11.1 Set-Reset (*SR*) Latch

A set-reset (SR) latch has four different possible output outcomes depending on the values of the inputs, S and R. A schematic of an SR latch is shown in Figure 11.2.

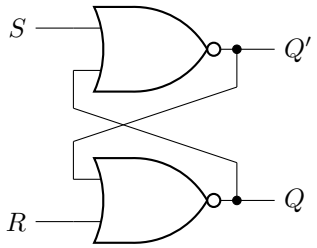


Figure 11.2: An SR latch.

Note that an SR latch gives us access to both the output Q and its complement Q'. This was another reason why we did not count inverters when we discussed the cost of a circuit. When we use latches and flip-flops, we get access to complemented values at no extra cost!

When S and R are both equal to zero, the output Q will retain its previous value. That is, if Q had initially been a zero, it will continue to be a zero. If Q had initially been a one, it will continue to be a one. Therefore an SR latch holds when the inputs are 00.

When S is zero and R is one, it does not matter what the previous value of Q is, it will become zero. Therefore an SR latch resets (becomes zero) when the inputs are 01.

When S is one and R is zero, it does not matter what the previous value of Q is, it will become one. Therefore an SR latch sets when the inputs are 10.

Finally, when S and R are both one, both Q and Q' become zero. This doesn't make logical sense. In addition, it leads to a possible unstable situation if the inputs go from 11 to 00 simultaneously. If one of the NOR gates had a slightly different propagation delay than the other, then the output could become either a zero or a one. This unstable situation is called a race condition (the outputs race each other to the inputs), and because it leads to an unpredictable latch operation, is a forbidden state. We therefore want to ensure that both S and R are not both one at the same time.

The operation of an SR latch is summarized in table 11.1. Q denotes the initial value of the output, and Q^+ denotes the next value of the output.

S	R	Q	Q^+	
0	0	0	0	hold
0	0	1	1	
0	1	0	0	reset
0	1	1	0	
1	0	0	1	set
1	0	1	1	
1	1	0	–	not allowed
1	1	1	–	

Table 11.1: Operation of an SR latch.

Simplified circuit symbols can be used for latches. The symbol for an SR latch is shown in Figure 11.3.

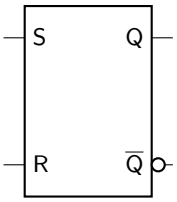


Figure 11.3: Circuit symbol for an SR latch.

11.2 Gated *SR* Latch

Latches without an enable bit are said to be transparent. That is, the output "sees" and immediately reacts to the inputs.

Gated latches are simply latches with an enable bit. When the enable bit is activated, the latch operates normally. When the enable bit is deactivated, the latch output holds the previous value, regardless of the other input values. When the enable is deactivated, the latch is said to be opaque, because the output no longer "sees" and reacts to the inputs.

An active HIGH enable gated SR latch can therefore be created from a normal SR latch and two additional AND gates, as shown in Figure 11.4. The circuit symbol for a gated SR latch is also shown in the same figure.

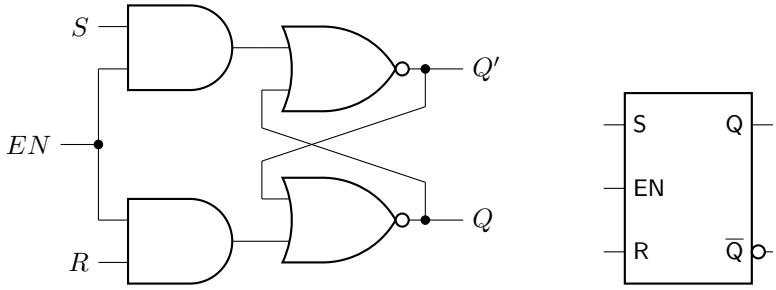


Figure 11.4: An active HIGH enable gated SR latch created from NOR and AND gates; and the circuit symbol for that same gated SR latch.

The ability to create a gated SR latch helps to remove some of the difficulty with otherwise transparent latches. If the enable bit is to be activated over an interval of time, and we want to ensure that the output only changes once during that time interval, regardless of how the inputs S and R change, we can cascade two gated SR latches together into something called a primary-secondary SR flip-flop. This ensures that the output only changes once any time the enable bit is activated, although it requires a change from the enable bit from zero to one in order to cascade the output from the primary latch to the output. A schematic of a primary-secondary SR flip-flop is shown in Figure 11.5.

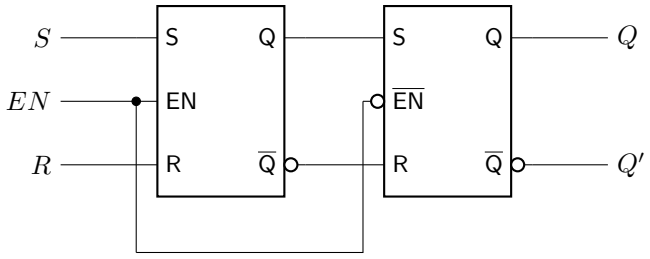


Figure 11.5: A primary-secondary SR flip-flop.

11.3 Gated Data (D) Latch

Similarly to a gated SR latch, we can create a gated data (D) latch. Recall the problem with the SR latch when both inputs are one. The D latch removes this possibility by ensuring that S and R always have different values, leading only to reset and set outputs. A gated D latch is created from an SR latch and the addition of two AND gates. Or, a gated D latch can be created from a gated SR latch and an inverter. Both of these possibilities are shown in Figure 11.6.

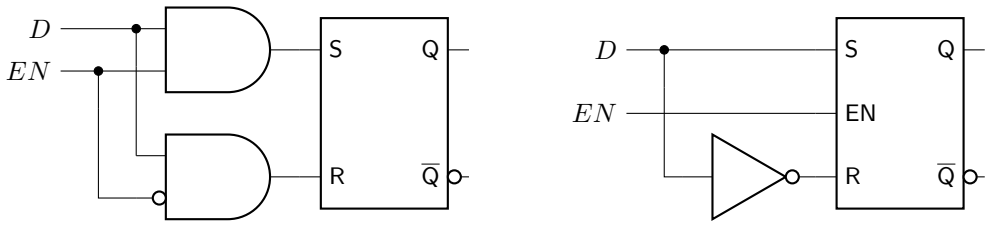


Figure 11.6: Two ways to create an active HIGH enable gated D latch.

11.4 D Flip-Flop

As noted in the brief discussion of an SR flip-flop above, it is possible to create a latch that only changes in response to a changing enable signal. This removes the asynchronous aspect (transparency) of the latch. A flip-flop is a synchronous device that is capable of having different output conditions. Rather than using a level enable signal, a clock (oscillating signal that goes between zero and one) signal is used instead. An edge-triggered flip-flop has the output state change only when a clock signal transitions from zero to one or from one to zero.

A clock transition from zero to one is called a rising edge, and a flip-flop whose output changes on this condition is called a rising-edge triggered flip-flop. A clock transition from one to zero is called a falling edge, and a flip-flop whose output changes on this condition is called a falling-edge (or negative-edge) triggered flip-flop.

A D flip-flop acts much like a D latch, but rather than have an enable signal, it is clocked. The circuit symbol for a rising-edge triggered D flip-flop is shown in Figure 11.7.

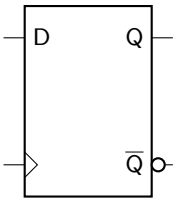


Figure 11.7: Circuit symbol for a rising-edge triggered D flip-flop.

The operation of a rising-edge triggered D flip-flop is defined in table 11.2.

CLK	D	Q^+
0	×	hold
1	×	hold
↑	0	reset
↑	1	set

Table 11.2: Operation of a rising-edge triggered D flip-flop.

If a flip-flop is falling-edge triggered, it will have a bubble on the clock input, much as we have seen with active LOW signals.

We can determine a characteristic equation for a D flip-flop that defines what happens to the output when the clock triggers the flip-flop. We need to understand the relationship between Q (the previous output value), D (the input value), and Q^+ (the next output value after the triggering event). The truth table that we will work with is given in table 11.3.

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

Table 11.3: Truth table for a D flip-flop.

Using Boolean algebra, it is easy to find an expression for Q^+.

$$\begin{aligned} Q^+ &= DQ' + DQ \\ &= D \end{aligned}$$

It is similarly simple to find the inverse of this characteristic equation for the D flip-flop. In other words, we may ask ourselves what value of D would be required to cause Q^+ to transition between two particular values. This information becomes vitally important in the next two chapters of this book. These transitions are shown in table 11.4. Note that any time we want the new output to be a zero, the input of a D flip-flop should be zero. Any time we want the new output to be a one, the input of a D flip-flop should be a one. This makes the D flip-flop a very easy flip-flop to work with, as we will see later in this book.

$Q \to Q^+$	D
$0 \to 0$	0
$0 \to 1$	1
$1 \to 0$	0
$1 \to 1$	1

Table 11.4: Transitions of a D flip-flop.

11.5 Toggle (T) Flip-Flop

A toggle flip-flop is capable of an action that we have not yet seen in a sequential circuit: toggling the output. The circuit symbol for a rising-edge triggered T flip-flop is shown in Figure 11.8, and the operating conditions are given in table 11.5.

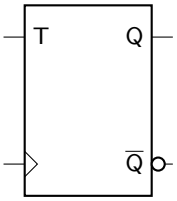


Figure 11.8: Circuit symbol for a rising-edge triggered T flip-flop.

CLK	T	Q^+
0	$\times$	hold
1	$\times$	hold
$\uparrow$	0	hold
$\uparrow$	1	toggle

Table 11.5: Operation of a rising-edge triggered T flip-flop.

We can determine a characteristic equation for a T flip-flop that defines what happens to the output when the clock triggers the flip-flop. The truth table that explains the relationship between T, Q, and Q^+ is given in table 11.6.

T	Q	Q^+
0	0	0
0	1	1
1	0	1
1	1	0

Table 11.6: Truth table for a T flip-flop.

Using Boolean algebra, it is easy to find an expression for Q^+.

$$\begin{aligned} Q^+ &= TQ' + T'Q \\ &= T \oplus Q \end{aligned}$$

By analyzing the truth table shown in table 11.6, we can find the transitions for a T flip-flop. Note any time we want the output to stay the same, the value of T should be zero. Any time we want the output to change (whether it goes from zero to one or one to zero) the value of T should be 1. The transitions for a T flip-flop are shown in table 11.7.

$\mathbf{Q \rightarrow Q^+}$	**T**
$0 \rightarrow 0$	0
$0 \rightarrow 1$	1
$1 \rightarrow 0$	1
$1 \rightarrow 1$	0

Table 11.7: Transitions of a T flip-flop.

11.6 *SR* Flip-Flop

An SR flip-flop acts much like an SR latch, but is a synchronous device. The S input will set the output while the R input will reset the output. Both S and R should not be set HIGH simultaneously. The circuit symbol for a rising-edge triggered SR flip-flop is shown in Figure 11.9, and the operation is described in table 11.8.

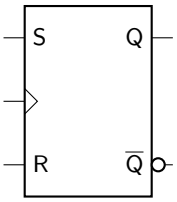


Figure 11.9: Circuit symbol for a rising-edge triggered SR flip-flop.

CLK	**S**	**R**	$\mathbf{Q^+}$
0	$\times$	$\times$	hold
1	$\times$	$\times$	hold
$\uparrow$	0	0	hold
$\uparrow$	0	1	reset
$\uparrow$	1	0	set
$\uparrow$	1	1	not allowed

Table 11.8: Operation of a rising-edge triggered SR flip-flop.

We can find a characteristic equation for the SR flip-flop by analyzing its truth table, given in table 11.9.

S	R	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	×	–

Table 11.9: Truth table for an SR flip-flop.

Using Boolean algebra, we can find an expression for Q^+.

$$Q^+ = SR' + R'Q$$

Finding the transitions for an SR flip-flop are a bit more complicated than with D and T flip-flops. Going from zero to zero, we notice that either a hold or reset situation must be present on the inputs. This requires S to be zero, but we don't care what the R input is. (If $SR = 00$ we get a hold, and if $SR = 01$ we get a reset.)

Going from zero to one requires a set operation where S is one and R is zero. Going from one to zero requires a reset operation (S is zero and R is one). Finally, going from one to one requires either a hold or a set, therefore R must be zero and we don't care what S is. All of these transitions are given in table 11.10.

Q → Q^+	S	R
0 → 0	0	×
0 → 1	1	0
1 → 0	0	1
1 → 1	×	0

Table 11.10: Transitions of an SR flip-flop.

11.7 *JK* Flip-Flop

A JK flip-flop is named after Jack Kilby, the inventor of the integrated circuit. (Therefore, the inputs J and K don't really stand for anything the way that S, R, D, and T did in previous devices.)

A JK flip-flop combines the utility of both D and T flip-flops. It is capable of holding, setting, resetting, and toggling the output based on the values of J and K. The circuit symbol for a rising-edge triggered JK flip-flop is shown in Figure 11.10, and the operation is described in table 11.11.

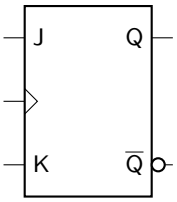


Figure 11.10: Circuit symbol for a rising-edge triggered JK flip-flop.

CLK	J	K	Q^+
0	×	×	hold
1	×	×	hold
↑	0	0	hold
↑	0	1	reset
↑	1	0	set
↑	1	1	toggle

Table 11.11: Operation of a rising-edge triggered JK flip-flop.

Just as we did with the other two flip-flops, we can find a characteristic equation for the JK flip-flop by analyzing its truth table, given in table 11.12.

J	K	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 11.12: Truth table for a JK flip-flop.

Using Boolean algebra, we can find an expression for Q^+.

$$\begin{aligned} Q^+ &= J'K'Q + JK'Q' + JK'Q + JKQ' \\ &= J'K'Q + JK' + JKQ' \\ &= K'Q + JK' + JQ' \\ &= K'Q + JQ' \end{aligned}$$

Finding the transitions for a JK flip-flop are a bit more complicated than with D and T flip-flops. Going from zero to zero, we notice that either a hold or reset situation must be present on the inputs. This requires J to be zero, but we don't care what the K input is. (If $JK = 00$ we get a hold, and if $JK = 01$ we get a reset.)

Going from zero to one, we either need a set or a toggle. J must be one and we don't care what K is. Going from one to zero, we either need a reset or a toggle. In this case, K must be one and we don't care what the value of J is. Finally, going from one to one, we either need a hold or a set, therefore K must be zero and we don't care what J is. All of these transitions are given in table 11.13.

$\mathbf{Q \rightarrow Q^+}$	**J**	**K**
$0 \rightarrow 0$	0	$\times$
$0 \rightarrow 1$	1	$\times$
$1 \rightarrow 0$	$\times$	1
$1 \rightarrow 1$	$\times$	0

Table 11.13: Transitions of a JK flip-flop.

11.8 Setup and Hold Considerations

Whenever an edge-triggered circuit element is used, it is important to verify that the data input will not cause any conflicts with the clock signal. If a flip-flop receives a signal that changes at the exact instant a rising (or falling) edge occurs, what value will be clocked through to the output? For example, given the timing diagram shown in Figure 11.11 corresponding to a rising-edge triggered D flip-flop, what value will clock to the output at a time of 5 ns? A zero? Or a one? It is unclear.

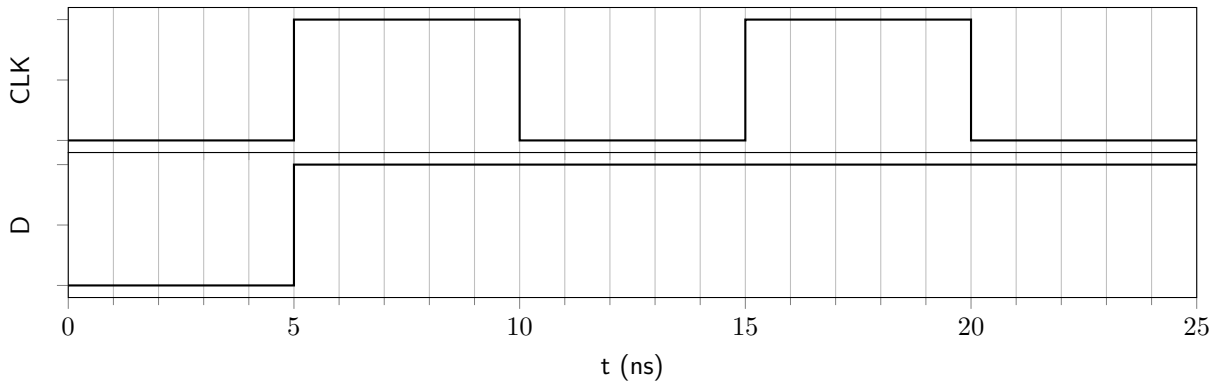


Figure 11.11: Input signals on a D flip-flop, leading to potential setup or hold problems.

A flip-flop's setup time (t_{su}) refers to the minimum amount of time that must be value on a flip-flop input must be constant prior to a triggering clock edge. A flip-flop's hold time (t_h) refers to the minimum amount of time that must be value on a flip-flop input must be constant after a triggering clock edge. Figure 11.12 depicts setup and hold time of a rising-edge triggered flip-flop.

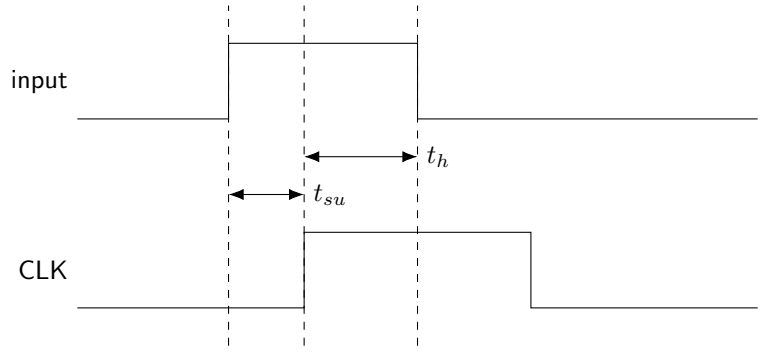


Figure 11.12: Input signals on a rising-edge triggered flip-flop, indicating the setup and hold times.

When using a flip-flop in a synchronous circuit, it is important to understand the setup and hold time values. These can be determined by reading the datasheet for the flip-flop in use. Be sure that any circuit is designed such that input signals will not change outside of setup and hold times.

11.9 Converting Flip-Flop Types

There are four major types of flip-flops (SR, D, T, and JK). However, it may be possible that you need to generate the functionality of one flip-flop using another. (Say, you have a bunch of D flip-flop chips but wish to use a T flip-flop.) It is possible to convert one type of flip-flop to another by deriving Boolean expressions for each of the flip-flop inputs on the chips that you have based on the transitions of the flip-flop that you want to create.

Converting one type of flip-flop to another can be accomplished by creating a transition table. The columns on the left define the flip-flop that you want, and the values of Q and Q^+ are based on the desired flip-flop functionality. The column(s) on the right define the flip-flop that you have, and are based on the known transitions of the flip-flop. This is outlined in table 11.14.

Flip-flop you want			**Flip-flop you have**
Input(s)	Q	Q^+	Input(s)
based on how the flip-flop should function			based on the transitions of the flip-flop you have

Table 11.14: Outline of creating a transition table to convert one type of flip-flop to another.

Example: Converting an SR flip-flop into a T flip-flop

Convert an SR flip-flop into a T flip-flop. The transition table below has the functionality of the desired flip-flop (T) on the left-hand side. The right-hand side uses the transitions of an SR flip-flop given in table 11.10 to generate the desired output response.

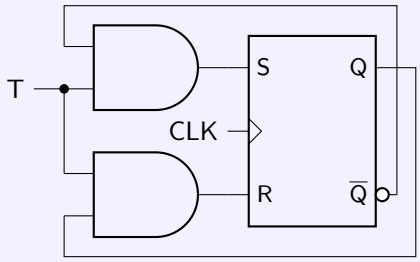
What you want			**What you have**	
T	**Q**	**Q⁺**	**S**	**R**
0	0	0	0	×
0	1	1	×	0
1	0	1	1	0
1	1	0	0	1

At this point, Boolean expressions can be derived for each input of the flip-flop you have, S and R.

$$S = TQ'$$

$$R = TQ$$

Draw a circuit diagram corresponding to the converted flip-flop.



11.10 Asynchronous Flip-Flop Inputs

All of the inputs we have discussed so far to each flip-flop (D, T, SR, and JK) are synchronous. That is, the output only changes in response to these variables when a clock has a triggering event (rising edge or falling edge). There are times when it is convenient or even necessary to have an asynchronous input that allows us to either set or reset the output of the flip-flop immediately, without having to wait for a clock trigger. This can be useful especially in timing situations when a reset back to zero is necessary. This is something that might be desirable when a button is pressed (for example, pushing a button to reset a synchronous circuit); in the case of ripple counters, which are discussed in section 12.4; in the use of one-hot state assignments, which are discussed in section 13.9.1; or possibly in other scenarios.

Many flip-flops contain asynchronous inputs called preset (sets the output to one) and clear (resets the output to zero). Usually, they are active LOW inputs. When these inputs are disabled, the flip-flop will act

like a normal flip-flop. When activated, the output will either be zero (if clear is enabled) or one (if preset is enabled). (You may wonder what happens if both preset and clear are asserted simultaneously; if you look at a datasheet for a specific flip-flop it will usually tell you what the outputs will be in that scenario. Usually asserting both asynchronous inputs simultaneously is not desirable.)

A circuit symbol of each type of flip-flop with active LOW preset and clear inputs is shown in Figure 11.13.

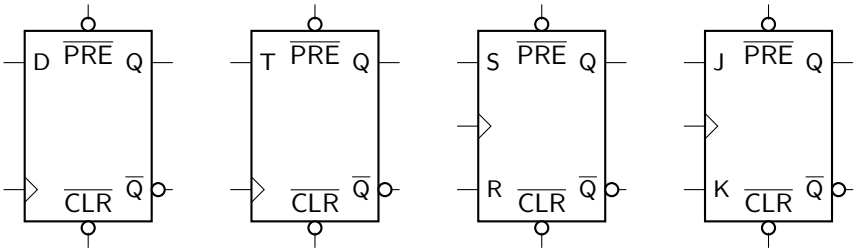


Figure 11.13: Circuit symbols for flip-flops with active LOW preset and clear inputs.

11.11 Example Problems

SR Latch

1. Create a timing diagram for an SR latch, given the inputs shown in Figure 11.14. The initial value of Q is 1. Ignore any propagation delays.

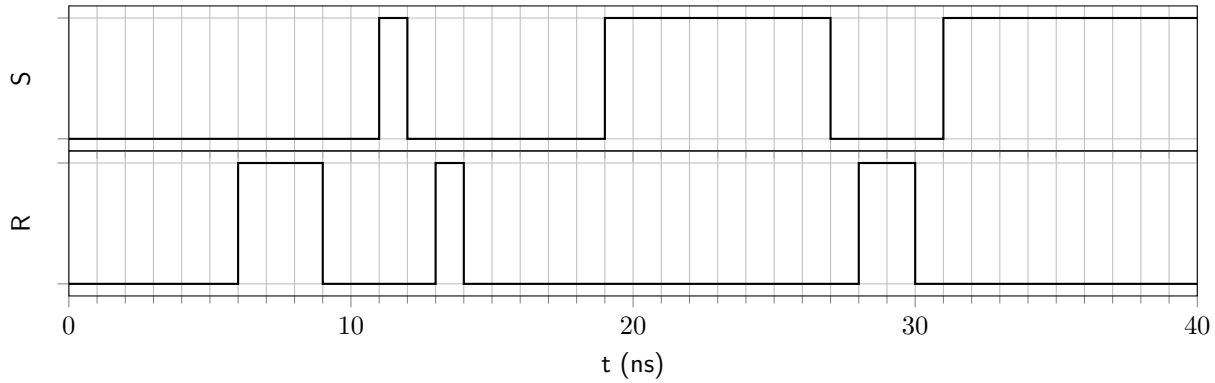


Figure 11.14: Input signals for SR latch question 1.

2. Design an SR latch using only a 4 to 1. MUX. Design it so that the 11 state leads to a hold situation.

3. The latch showed in Figure 11.15 is capable of holding, setting, and resetting. However it does not do so with the same input combinations as the SR latch defined in this book. What input combinations lead to a hold, set, and reset condition on the output?

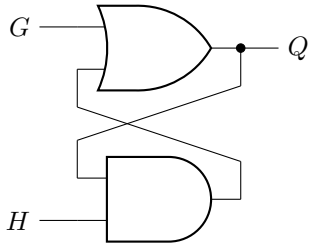


Figure 11.15: Latch circuit diagram corresponding to question 3.

4. A NAND gate SR latch is built as shown in Figure 11.16. Which of the outputs is Q, and which is Q'?

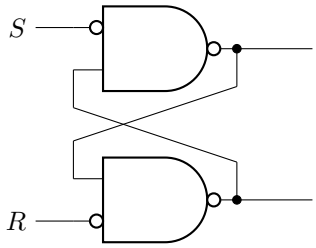


Figure 11.16: NAND SR latch circuit diagram corresponding to question 4.

5. Design an SR latch using one AND gate, one OR gate, and one inverter. (Note that this latch will have one output, Q, and no Q' output.)

D Latch

1. Create a timing diagram for an active LOW enabled gated D latch, given the inputs shown in Figure 11.17. The initial value of Q is 0. Ignore any propagation delays.

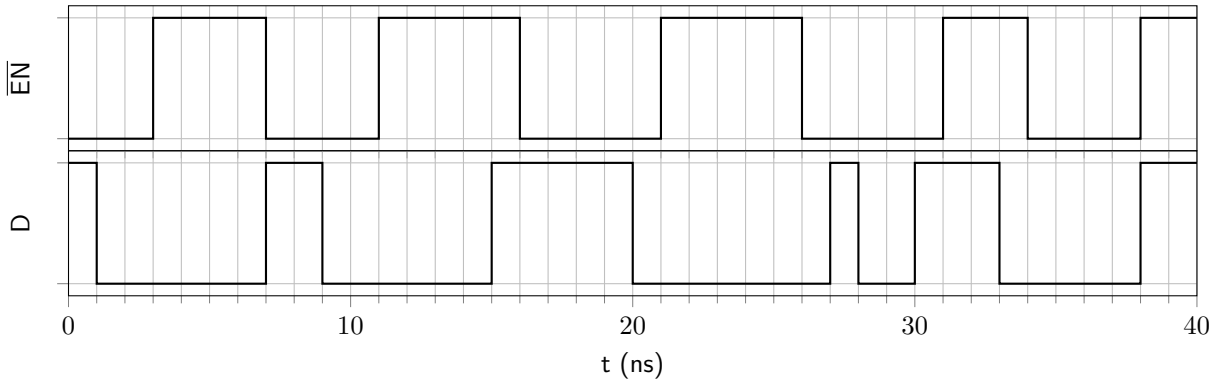


Figure 11.17: Input signals for D latch question 1.

2. Design a D latch out of NAND gates.

3. Design a primary-secondary D flip-flop.

4. Take a look at the datasheet for the 74LS75 (4-bit bistable latches). Which of the pins on the DIP chip correspond to the enable pins? Are they active HIGH enable or active LOW enable?

5. Take a look at the datasheet for the 74116 (dual 4-bit latches with clear). How many enable pins are there per latch, and are they active HIGH enable or active LOW enable?

D Flip-Flop

1. On the next rising edge of the clock, what will the output value Q be for the circuit diagram given in Figure 11.18?

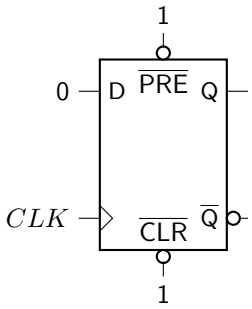


Figure 11.18: Circuit diagram corresponding to question 1.

2. On the next rising edge of the clock, what will the output value Q be for the circuit diagram given in Figure 11.19?

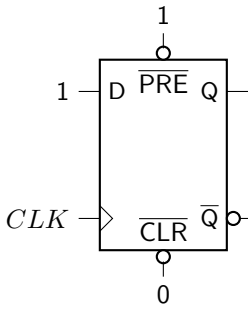


Figure 11.19: Circuit diagram corresponding to question 2.

3. Create a toggle flip-flop using a D flip-flop and as many external logic gates as you need.

4. Use a 2 to 1 MUX and two D flip-flops to create a D flip-flop that updates on both rising and falling edges.

5. A scan flip-flop is built from a D flip-flop and has two extra inputs. When the TE (test enable) input is zero, the flip-flop acts as a normal D flip-flop. When the TE input is one, the flip-flop uses TI (test input) rather than D to update the output. Design a scan flip-flop.

T Flip-Flop

1. Create a timing diagram for a rising-edge triggered T flip-flop, given the inputs shown in Figure 11.20. Assume that any asynchronous inputs are not asserted. The initial value of Q is zero. Ignore any propagation delays.

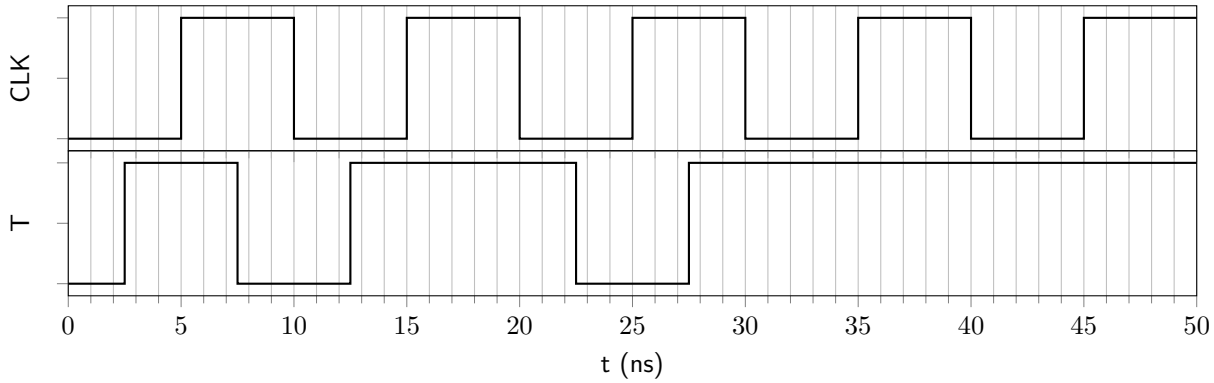


Figure 11.20: Input signals for T flip-flop question 1.

2. Create a timing diagram for a falling-edge triggered T flip-flop, given the inputs shown in Figure 11.21. Assume that any asynchronous inputs are not asserted. The initial value of Q is one. Ignore any propagation delays.

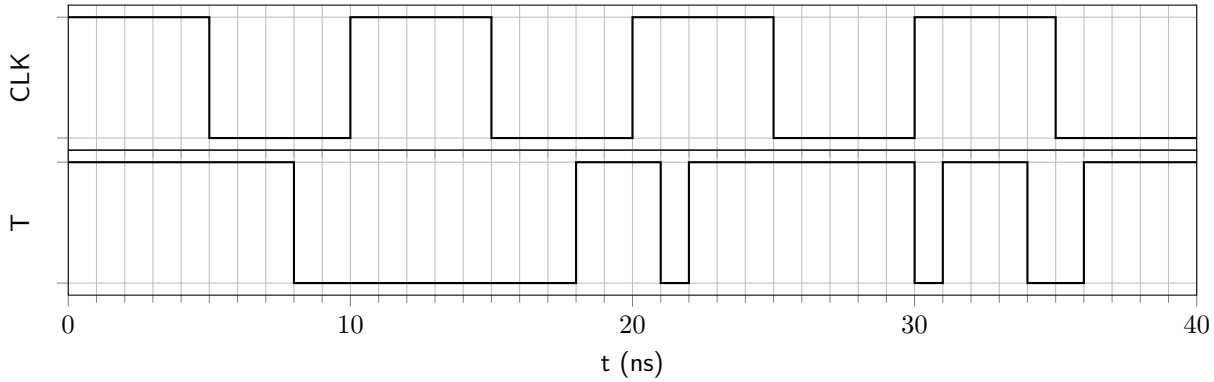


Figure 11.21: Input signals for T flip-flop question 2.

3. Create a JK flip-flop using a T flip-flop and as many external logic gates as you need.

4. Create an SR flip-flop using a T flip-flop and as many external logic gates as you need. Force the flip-flop to hold if S and R are both 1 at the same time.

5. Create a T flip-flop with active LOW asynchronous preset and clear pins. Assume that you only have a JK flip-flop (without any asynchronous input pins) and AND/OR/NOT gates. You can safely assume that both the preset and clear inputs will not both be asserted simultaneously.

JK Flip-Flop

1. Create a timing diagram for a falling-edge triggered JK flip-flop, given the inputs shown in Figure 11.22. Assume that any asynchronous inputs are not asserted. The initial value of Q is one. Ignore any propagation delays.

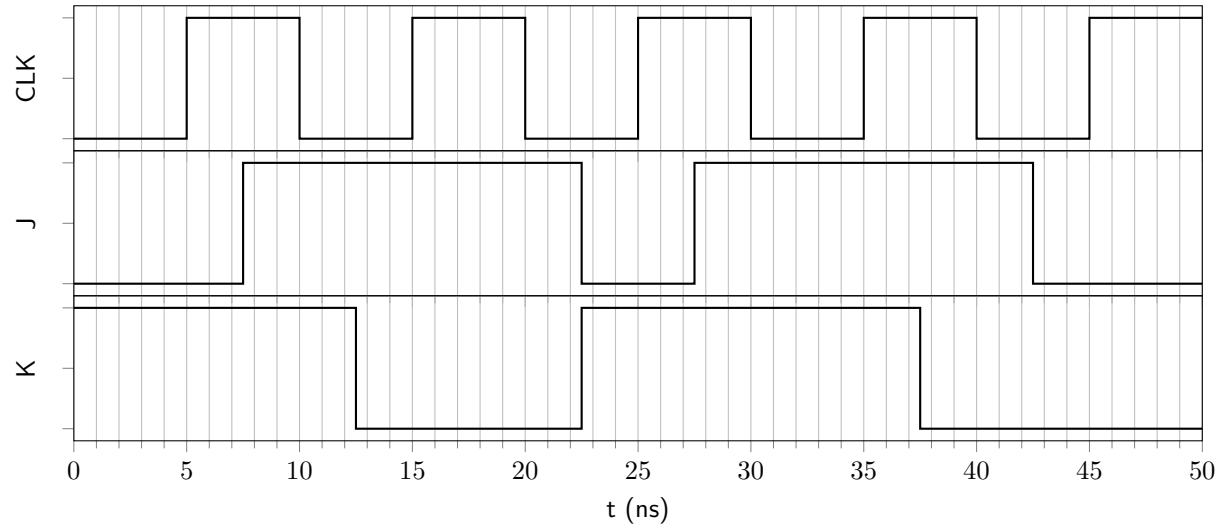


Figure 11.22: Input signals for JK flip-flop question 1.

2. Create a timing diagram for a rising-edge triggered JK flip-flop, given the inputs shown in Figure 11.23. Assume that any asynchronous inputs are not asserted. The initial value of Q is one. Ignore any propagation delays.

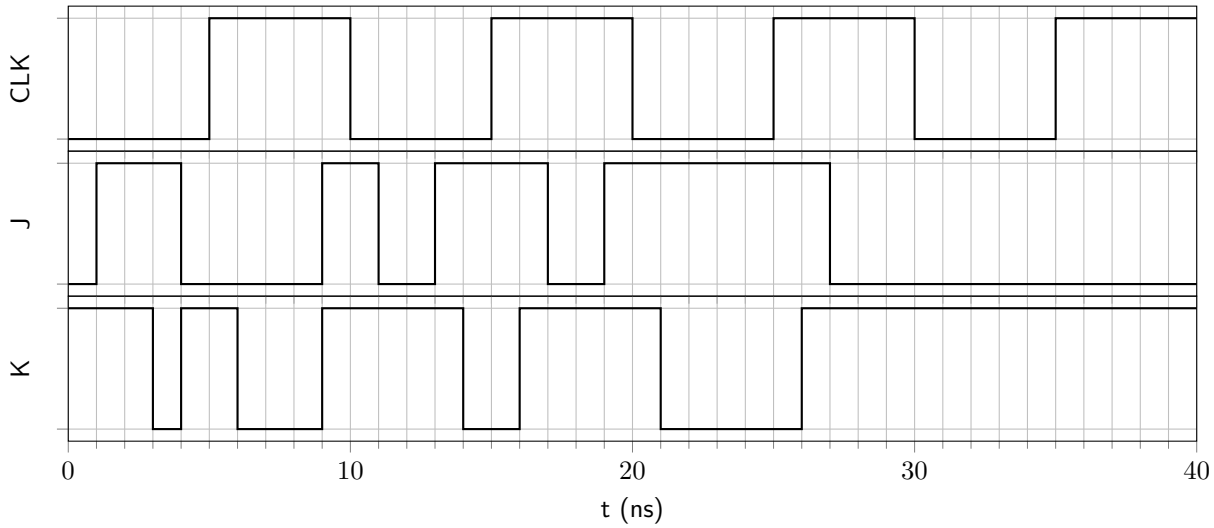


Figure 11.23: Input signals for JK flip-flop question 2.

3. Design a primary-secondary JK flip-flop.

4. Design a JK flip-flop from a D flip-flop and as many external gates as you need.

5. You wish to design a new type of flip-flop, an AB flip-flop, that has the characteristic of 00 (reset), 01 (set), 10 (hold) and 11 (toggle). Create an AB flip-flop using a JK flip-flop and as many external gates as you need.

12 Registers and Counters

This chapter will focus on two very important devices that can be created with flip-flops: registers and counters.

12.1 Registers

A register is a sequential logic circuit that is capable of storing, loading, and shifting data. It is made up of flip-flops, usually D flip-flops due to their simplicity. The storage capacity of a register pertains to how many bits of binary data it can store. Modern computers generally have register sizes of thirty-two bits or sixty-four bits. Many microcontrollers, such as the one used in the popular Arduino Uno, have register sizes of eight bits.

The ability to store data and move it from one place to another using a serial communication protocol (sending multiple bits of data over one wire rather than multiple wires) makes registers supremely important in modern computing.

There are four types of registers, each described below.

12.1.1 Serial In / Serial Out (SISO)

In a serial in / serial out (SISO) register, binary data is input into the first flip-flop, and then is shifted through subsequent flip-flops on successive clock ticks. The input data can take on different values in order to send a particular variable or message through from input to output. The circuit diagram of a 4-bit SISO register is shown in Figure 12.1.

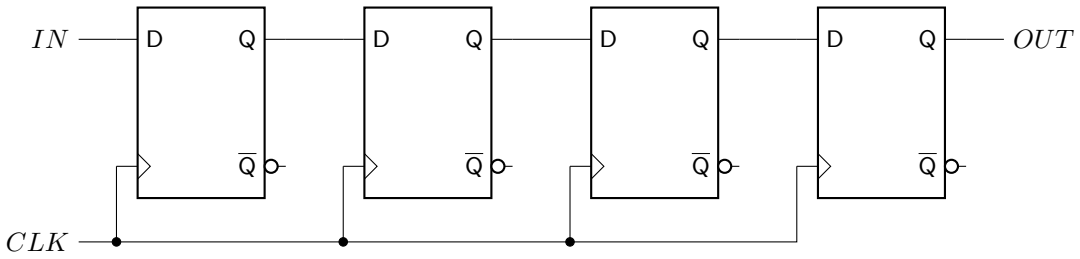


Figure 12.1: A serial in / serial out (SISO) register.

A purpose of SISO registers is as a data buffer, temporarily storing data as it passes through a digital circuit.

12.1.2 Serial In / Parallel Out (SIPO)

A serial in / parallel out (SIPO) register takes a serial data stream and allows each bit to be accessed individually at the output of each flip-flop. The data is shifted from one flip-flop to another upon a clock signal (rising or falling edge). A schematic of a 4-bit SIPO register is shown in Figure 12.2.

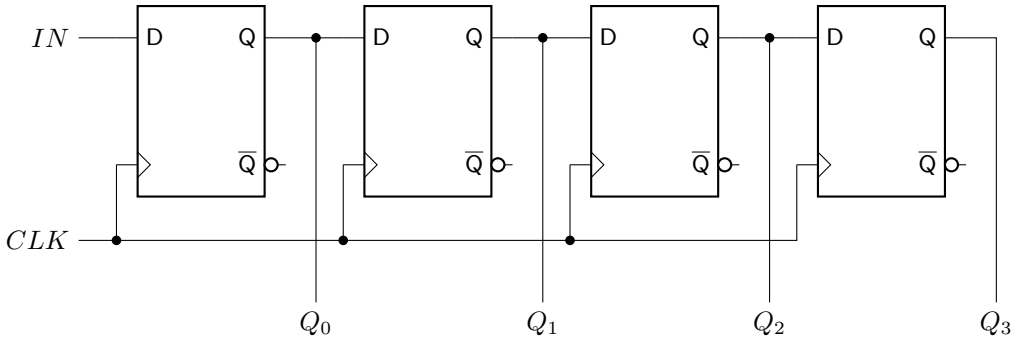


Figure 12.2: A serial in / parallel out (SIPO) register.

SIPO registers are very useful when working with a microcontroller (that has limited input/output pins) and a device with many inputs, such as a 7-segment display. With just a data stream and a clock signal, a 7-segment display can be written to using only two input/output pins. The drawback is that a serial protocol must be used. (Discussion of serial protocols is outside of the scope of this book.)

12.1.3 Parallel In / Serial Out (PISO)

A parallel in / serial out (PISO) register is in some ways the "opposite" of a SIPO register. Data can be loaded into the register, and then shifted through to the output at each clock tick. Whether or not the data is loaded or shifted depends on a control signal indicating either a load or shift. A full circuit diagram for a 4-bit PISO register would take a lot of space, so only 2-bits are shown in the circuit diagram in Figure 12.3. However, more flip-flops can be added to create a PISO register with a larger storage capacity.

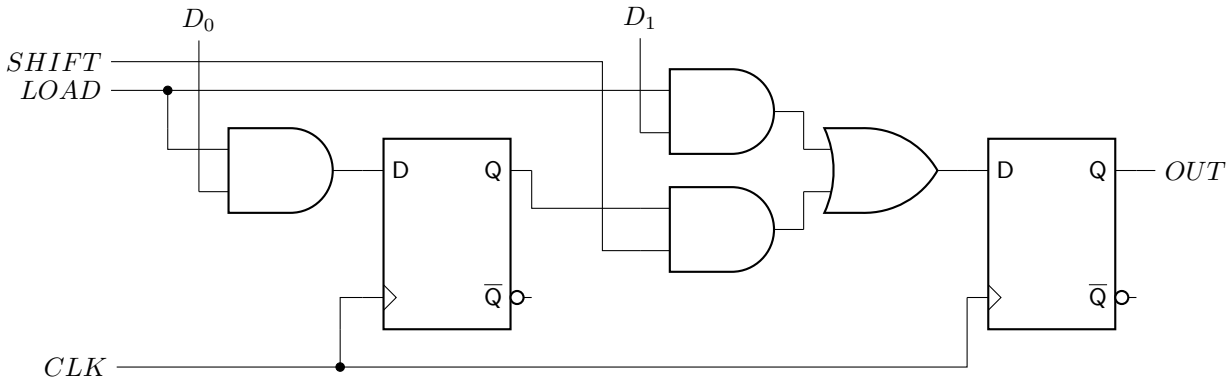


Figure 12.3: A parallel in / serial out (PISO) register.

Just as SIPO registers are useful in handling devices with multiple inputs, PISO registers are useful in handling devices with multiple outputs, such as a DIP switch. After interfacing with a PISO register, only one data pin and one clock pin need be used on a microcontroller to access all of the data from a DIP switch.

12.1.4 Parallel In / Parallel Out (PIPO)

A parallel in / parallel out (PIPO) register allows for simultaneous access to all input and output signals. The schematic shown in Figure 12.4 is a very simple 4-bit PIPO register. However, control logic can be included on the input to each flip-flop to enable data to shift as well as just load.

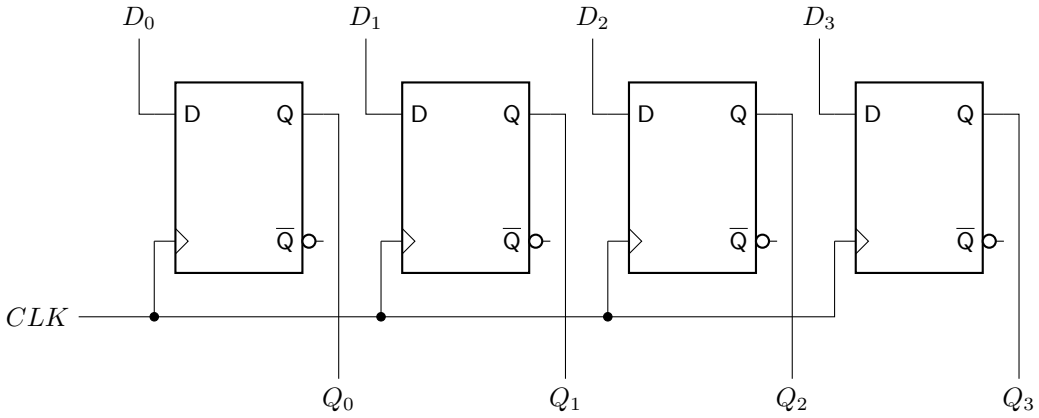


Figure 12.4: A parallel in / parallel out (PIPO) register.

PIPO registers are great for temporary data storage. A PIPO register that is capable of just loading and holding is useful in something like an arithmetic and logic unit. Data is temporarily stored, some type of operation (such as addition or subtraction) is carried out, and the result is stored into a different register. The output can then be written onto a data bus.

A PIPO register capable of shifting in one or both directions is also able to act as a PISO register or a SIPO register, giving it a very diverse set of functionality.

12.2 Using Registers to Add and Multiply

While registers are an important aspect of communication protocols, they also help to reduce the footprint (size) of computing devices used to add and multiply.

12.2.1 Addition

As discussed in chapter 3, combinational logic can be used to create half and full adders. A full adder by itself is only capable of summing together two numbers.

If three numbers need to be added together, then two full adders must be used. If four numbers need to be added together, then three full adders need to be used. These two possibilities are shown in Figure 12.5.

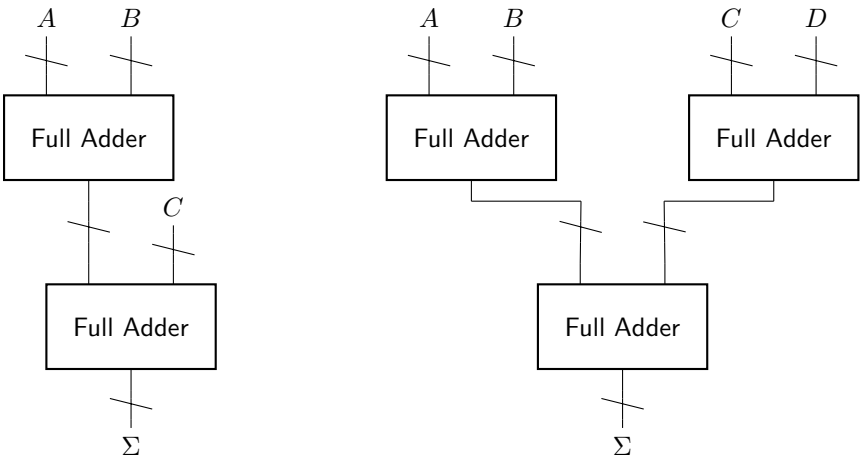


Figure 12.5: Adding three or four numbers together using a combinational approach.

Using combinational logic like this to implement an adder has several drawbacks. First, the number of m-bit full adders required is $n-1$, where n is how many numbers are to be added together. This quickly becomes untenable if many numbers need to be added together. Second, the combinational approach is not flexible. In other words: you cannot use a setup that can add two numbers together to add four numbers together. The adder would have to be rebuilt. This lack of flexibility makes the use of combinational logic a poor choice for computation.

Using sequential logic (including registers in the adding hardware) is a much better choice as far as scalability and flexibility. In general, the design of the hardware will limit the number of bits (which is also a limitation of combinational logic), but the hardware can add together as many numbers as needed, in sequence.

A block diagram for the approach to adding numbers together sequentially is shown in Figure 12.6.

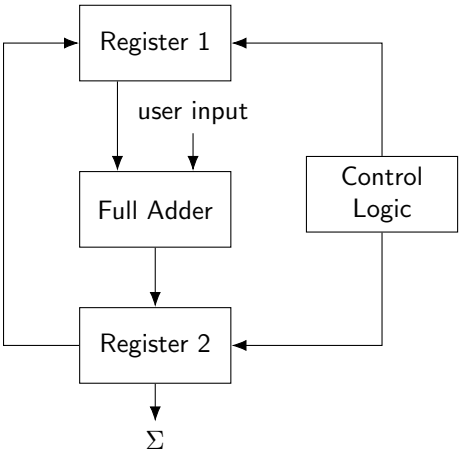


Figure 12.6: Adding multiple numbers together using a sequential logic approach.

The sequential addition works as follows.

1. Register 1 holds while register 2 loads (register 1 contains A, register 2 contains $A + B$).
2. Register 1 loads while register 2 holds (register 1 and 2 both contain $A + B$).
3. Both registers hold (register 1 and 2 both contain $A + B$).

There are a couple of drawbacks to using sequential logic to add. The first drawback is that it will take at least three clock cycles to accomplish the steps outlined above. This will very likely be longer than the propagation delay required to add numbers together using the combinational approach. However, the tradeoff between time and space generally favors the sequential approach. Second, the sequential approach requires control logic to enable the registers to hold or load at the appropriate time.

12.2.2 Multiplication

This section applies to unsigned multiplication only. Multiplication can also be tackled using combinational or sequential logic. Multiplying two 2-bit numbers together, shown below, requires one adder and four AND gates. This hardware is capable of generating numbers as large as 9 (3×3).

			A	B
		$\times$	C	D
			AD	BD
+	AC		BC	0
	AC		AD+BC	BD

Multiplying two 3-bit numbers together, shown below, requires four adders and nine AND gates. This hardware is capable of generating numbers as large as 49 (7×7).

				A	B	C
			$\times$	D	E	F
				AF	BF	CF
		AE		BE	CE	0
+	AD	BD		CD	0	0
	AD	AE+BD		AF+BE+CD	BF+CE	CF

It requires $(n-1)^2$ adders and n^2 AND gates to multiply together two n-bit numbers. This is not scalable; it becomes too large too fast to implement in a realistic way.

By analyzying the multiplication process, it can be seen that multiplication consists of shifting and AND/adding steps. By iterating this process, sequential logic can be used to much more easily multiply together two n-bit numbers. $n - 1$ shifts are required to multiply two n-bit numbers together. While this takes more time than a combinational approach, the small footprint makes it a feasable approach to the problem.

A block diagram for a 4-bit sequential multiplier is shown in Figure 12.7. The two numbers to be multiplied are $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$. In general, $2n$ bits of storage are needed to multiply together two n-bit numbers.

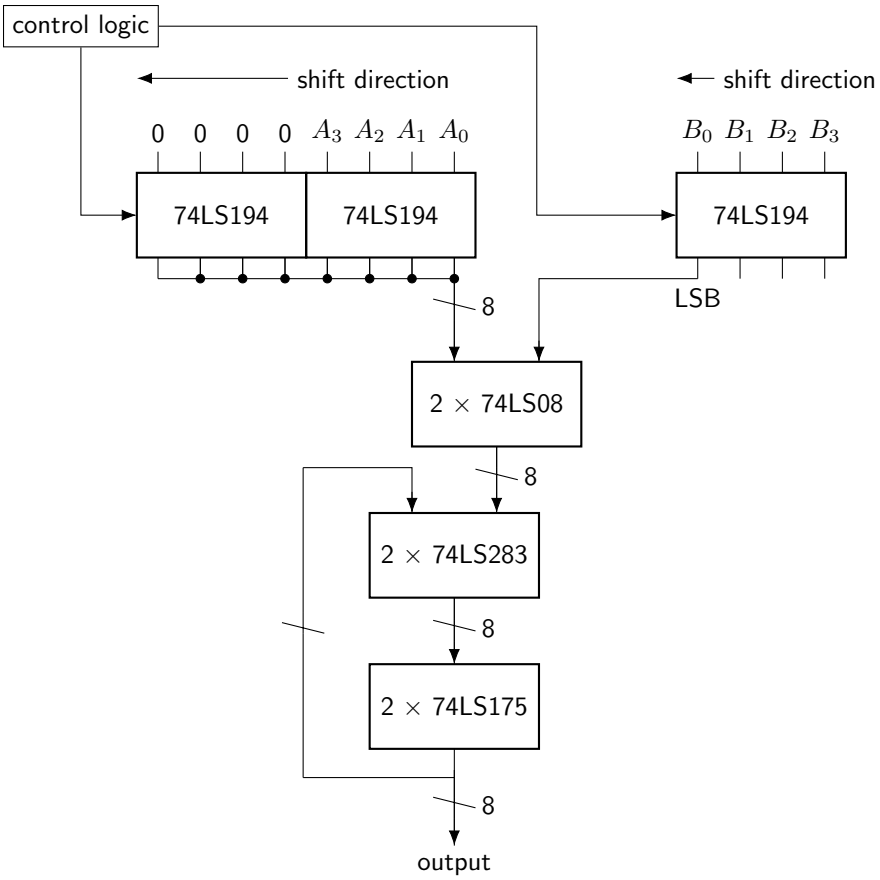


Figure 12.7: Multiplying two 4-bit numbers together using a sequential logic approach.

The control logic for this 4-bit multiplier works as follows.

1. All registers (the two 74LS194 containing A, the 74LS194 containing B, and the two 74LS175 containing the output) load data.
2. The two registers containing A and B shift one bit (the shift direction is shown in the block diagram), the register containing the output loads.
3. The two registers containing A and B shift one bit (the shift direction is shown in the block diagram), the register containing the output loads.
4. The two registers containing A and B shift one bit (the shift direction is shown in the block diagram), the register containing the output loads.
5. All registers hold data until the process is restarted.

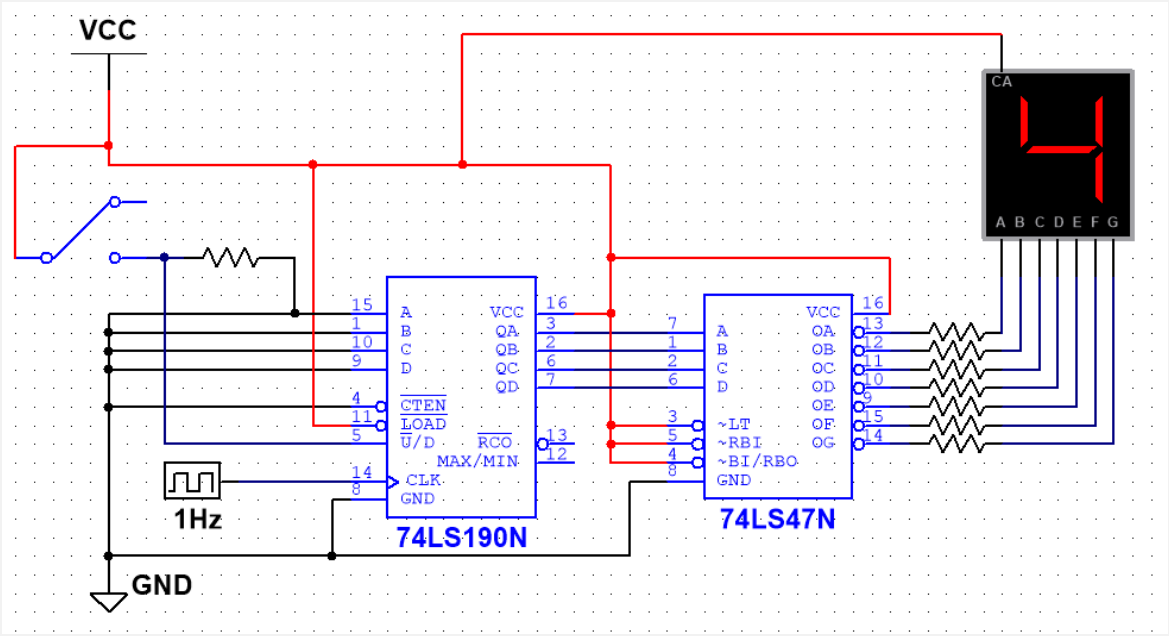
Note that steps 2–4 are identical; they are carrying out the shift and AND steps. The 74LS175 register carries out the addition step. The control logic creates an additional layer of difficulty to a synchronous approach, but can be scaled up much more easily than a combinational approach.

Circuit Project: Using the 74LS190 to Create an Up/Down BCD Counter

This project uses the 74LS190 BCD counter chip to create an up/down counter with the output on a 7-segment display.

- 1– Breadboard
- 1– 5 V power supply or battery pack
- 1– 1 Hz clock source
- 1– toggle switch
- 8– 220 Ω resistors
- 1– 7-segment display, common anode
- 1– 74LS47 BCD to seven-segment decoder/driver chip
- 1– 74LS190 Synchronous up/down counters with down/up mode control chip

The 74LS190 chip is a synchronous BCD counter. Given a clock source, it will automatically update each output value every time the clock ticks. The toggle switch in this circuit controls the direction of the count. When the toggle switch is in the up position, the signal sent to pin five is LOW and the counter counts up. When the toggle switch is in the down position, the signal sent to pin five is HIGH and the counter counts down.



12.3 Synchronous Counters

Counters are useful devices for accounting for how much time has passed. They can be used to display or keep track of time (such as with a clock), or they can be used to trigger events that occur at regular, predefined intervals of times. Counters are extremely useful components of microcontrollers that allow timed events to occur, or to create output waves that can control motors, for example.

A synchronous counter is simply a counter composed of two or more flip-flops, where every flip-flop shares a common clock signal. Synchronous counters can count up, down, or out-of-order, and can include user inputs (for example, count up when a toggle switch is in one direction, or count down when a toggle switch is in the other direction). Synchronous counters can be created using any type of flip-flop: D, T, JK, or SR.

To understand how a counter will function, we first create a state diagram. A state diagram simply shows all of the possible values that the flip-flops can take on. Arrows connect each of these states and indicate what the value will be after a clock trigger. A state diagram is not specific to the type of flip-flop being used. An example state diagram is shown in Figure 12.8.

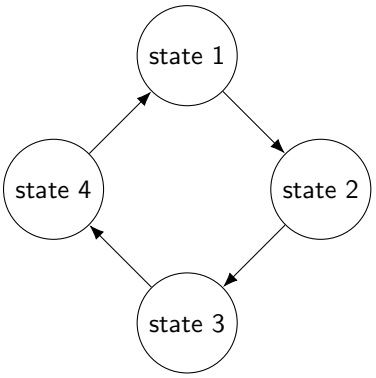


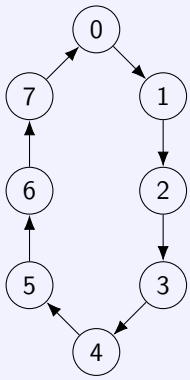
Figure 12.8: Example state digram.

Analyzing the state diagram enables us to make a transition table. This transition table shows the current values of the flip-flop, what the next values should be, and what input values need to be present on the flip-flop input in order to create that flip-flop transition (hence our focus on flip-flop transitions in the previous chapter).

After creating the transition table, Boolean expressions for each flip-flop input can be derived using any of the circuit minimization techniques that we have learned. This allows us to create a circuit diagram to build the counter.

Example: Creating a 3-bit counter with T flip-flops

The state diagram for a 3-bit counter is shown below. Note that it counts upward from zero to seven, and then wraps back to zero again.



The transition table, shown below, shows the current state (which is the binary value of the number in the counter), the next state (which, in this example, is one number higher than the current number, unless the current state is seven), and the input value on the flip-flop required to go from the current state value to the next state value. As an example, when the current state is 100 (4), the next state will be 101 (5). In order to have a hold on flip-flop A (the MSB stays one in the transition), a hold on flip-flop B (the next bit stays zero in the transition), and a toggle on flip-flop C (the LSB switches from zero to one), the flip-flop values must be 001.

Current State			**Next State**			**Flip-Flop Value**		
A	**B**	**C**	A^+	B^+	C^+	T_A	T_B	T_C
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

At this point, Boolean expressions can be derived for each of the flip-flop values based on the current state variables A, B, and C.

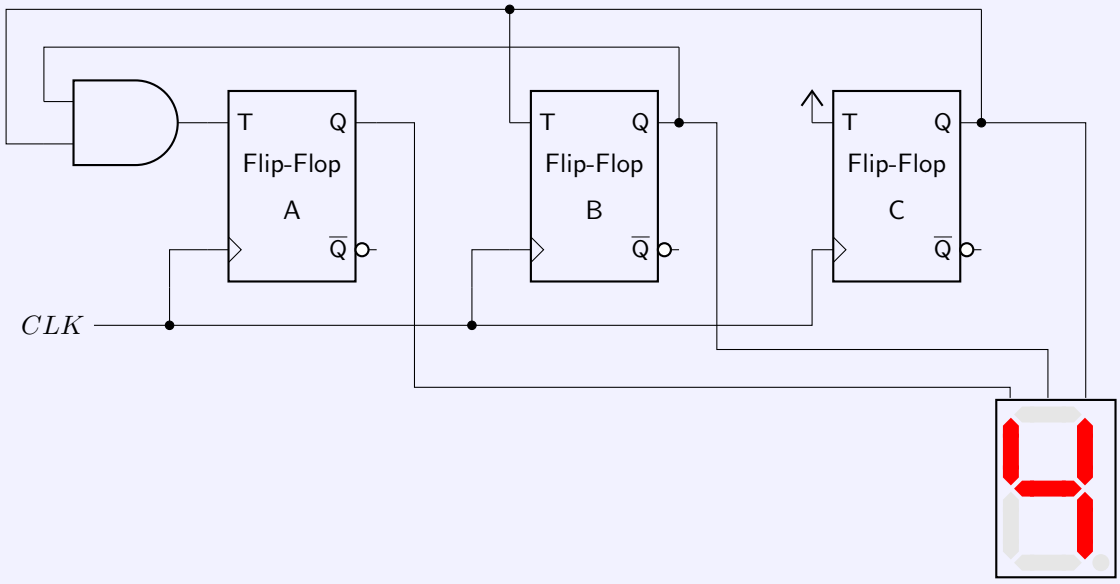
$$T_A = BC$$

$$T_B = C$$

$$T_C = 1$$

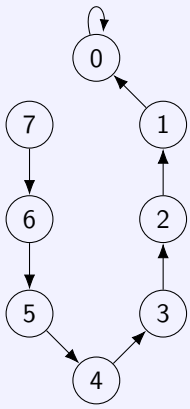
Now we can draw a circuit diagram corresponding to the completed counter. Notice that there

are no external inputs anywhere (pushbuttons, toggle switches, DIP switches). When the circuit is powered on, it will change its output values consistent with the state diagram.



Example: Creating a 3-bit countdown timer with D flip-flops

The state diagram for this example is shown below. We will now create a count-down timer that stops when it reaches zero. As circuit designers, we need to carefully consider how all conditions will affect our circuits. If our countdown timer initially turns on in the zero state, it will stay there indefinitely. We need to think of a manner to get all of the flip-flops to have an output value of one. Fortunately, we can connect a pushbutton to the preset pins on each flip-flop to accomplish this. Therefore, when a pushbutton is pressed, each flip-flop will take on an output value of one, and then continue the countdown to zero where they will remain until the button is pressed again.



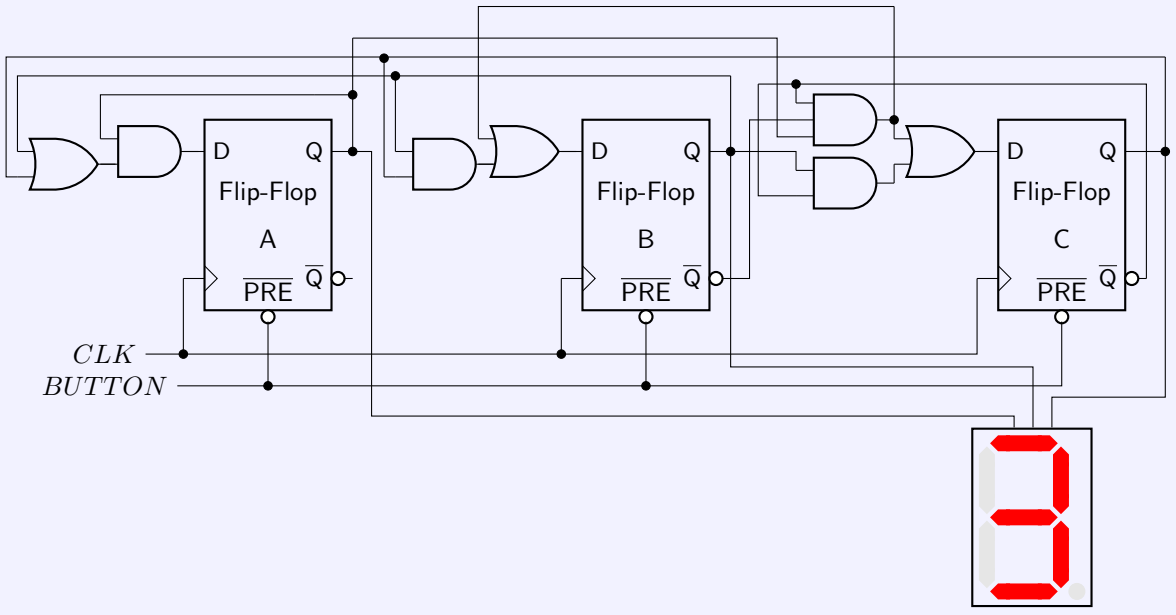
When creating a transition table for a D flip-flop, we note that the next value of Q is always equal

to the input value D. Therefore we do not need to create separate columns for the flip-flop values in the transition table when working with D flip-flops.

Current State			Next State		
A	**B**	**C**	**A^+**	**B^+**	**C^+**
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	1
0	1	1	0	1	0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

Now we can use our preferred method of Boolean algebra minimization to derive expressions for each flip-flop input.

$$D_A = A(B + C)$$
$$D_B = BC + AB'C'$$
$$D_C = BC' + AB'C'$$



An important type of counter is called a BCD counter or a decade counter. These counters count from zero through nine (either up or down). Four bits are required to count as high as nine. We know from our study of binary numbers that four bits are capable of representing numbers as high as fifteen. Because values

ten through fifteen are unused, they are generally set to don't care values in the transition table.

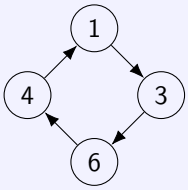
It is also possible to create counters that count out of order, and not in any logical sequence. The size of the counter in that case will refer to the largest number that needs to be expressed in binary. Any states that aren't represented in the state diagram can be don't care terms (with an important caveat, discussed in the next section!).

12.3.1 Initialization Values

It is important to understand that flip-flops may initialize with arbitrary values when a circuit is powered on for the first time. Therefore, it is important to be sure to designate a reset state (typically where all flip-flops have a value of zero) that can be asynchronously entered into by using the flip-flop clear pins. If zero is selected as the reset state, but is not an expected state in the state diagram, it's possible that don't care terms were used to represent the next state and flip-flop values in the design of a sequential circuit. If zero isn't a designed state, and don't care terms were used to designate the flip-flop values used when the flip-flop is in state zero, it is possible that unpredictable results can ensue after asynchronously entering into this state. The following example will demonstrate this.

Example: Out of order counter with and without initialization issues

Design an out of order counter with the following state diagram using T flip-flops. Designate all of the states that don't show up in the state diagram as don't cares.



The transition table is given below. Note that even though there are only four states, three flip-flops are required to represent numbers as high as six.

Current State			Next State			Flip-Flop Value		
A	B	C	A^+	B^+	C^+	T_A	T_B	T_C
0	0	0	×	×	×	×	×	×
0	0	1	0	1	1	0	1	0
0	1	0	×	×	×	×	×	×
0	1	1	1	1	0	1	0	1
1	0	0	0	0	1	1	0	1
1	0	1	×	×	×	×	×	×
1	1	0	1	0	0	0	1	0
1	1	1	×	×	×	×	×	×

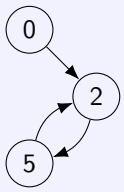
The flip-flop equations can be derived using any Boolean algebra minimization technique.

$$T_A = A' \oplus B$$

$$T_B = A \oplus B$$

$$T_C = A' \oplus B$$

Let's analyze what happens when this counter is turned on for the first time. Each flip-flop takes on a value of zero. If we plug 000 into each flip-flop equation, we get 010, which means flip-flop A will hold, flip-flop B will toggle, and flip-flop C will hold. That means the transition will go from 000–010. In state 2 (010), the flip-flops will toggle/hold/toggle, giving a transition from 010–101. In state 5 (101), the flip-flops will once again toggle/hold/toggle, giving a transition from 101–010. This will repeat indefinitely. The actual state diagram for this circuit is shown below.



To rectify this situation, we cannot have state zero take on don't care values. Instead, we will make the output go to state one after state zero. The new state table is shown below.

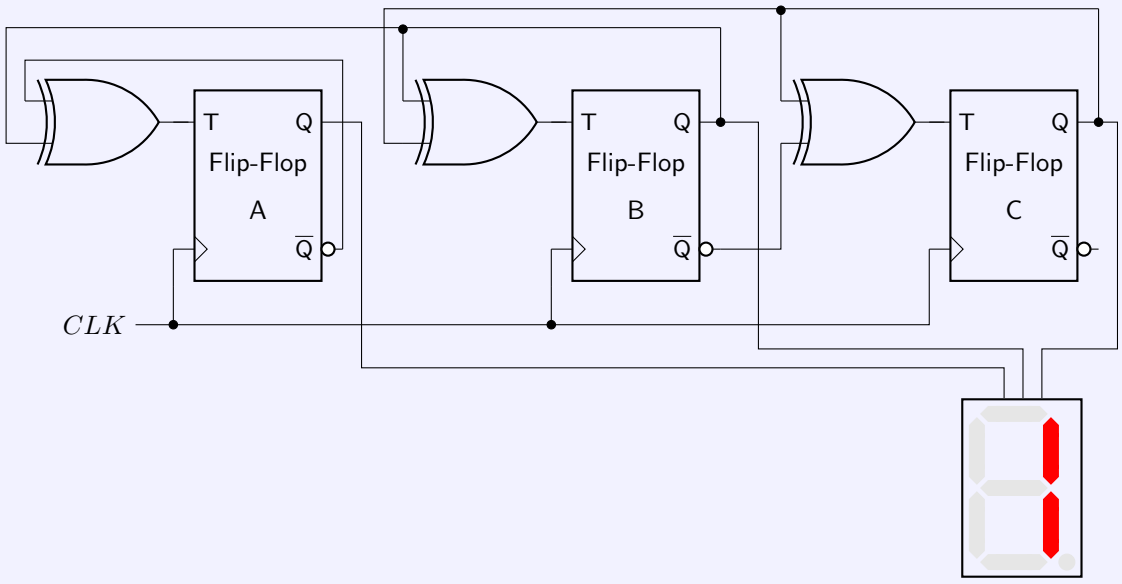
Current State			Next State			Flip-Flop Value		
A	B	C	A^+	B^+	C^+	T_A	T_B	T_C
0	0	0	0	0	1	0	0	1
0	0	1	0	1	1	0	1	0
0	1	0	×	×	×	×	×	×
0	1	1	1	1	0	1	0	1
1	0	0	0	0	1	1	0	1
1	0	1	×	×	×	×	×	×
1	1	0	1	0	0	0	1	0
1	1	1	×	×	×	×	×	×

The new flip-flop equations can be derived.

$$T_A = A' \oplus B$$

$$T_B = B \oplus C$$

$$T_C = B' \oplus C$$



12.4 Ripple Counters

A ripple counter is sometimes referred to as an asynchronous counter. I find the term "asynchronous" to be somewhat misleading. The difference between a ripple counter and a synchronous counter is that with a synchronous counter, every flip-flop shares a clock signal. With a ripple counter, one flip-flop receives an external clock signal and the other flip-flops receive clock signals from the output of other flip-flops (the output of one flip-flop "ripples" into the clock input of the next).

Ripple counters function under the operation of toggling every time the clock ticks. In this manner, whichever type of flip-flop is used, the input is connected to whatever signal will create a toggle situation. This acts to divide the frequency in half. A 3-bit ripple counter created with T flip-flops is shown in Figure 12.9.

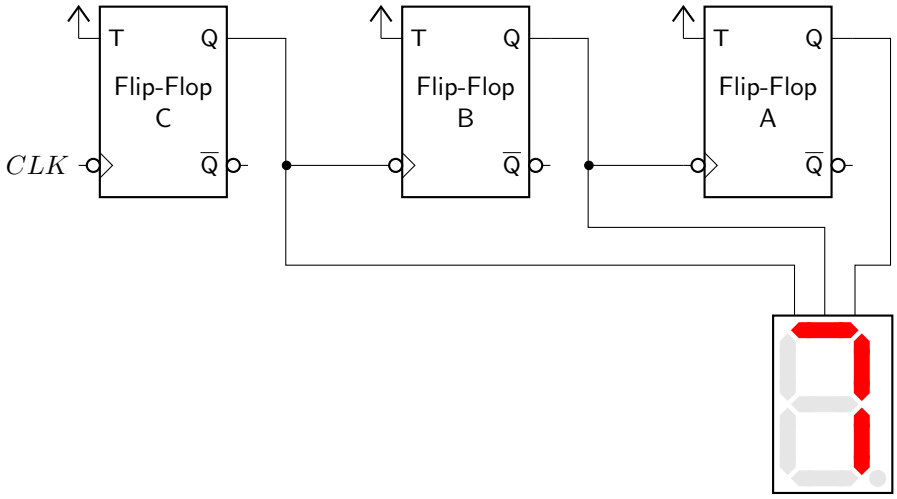


Figure 12.9: A 3-bit ripple counter created from T flip-flops.

Note that each flip-flop is falling-edge triggered (if they are rising-edge triggered, the counter will count down instead of up). The flip-flop connected to the external clock signal (labeled as C in Figure 12.9) is the least significant bit. The timing diagram (ignoring propagation delays) is shown in Figure 12.10.

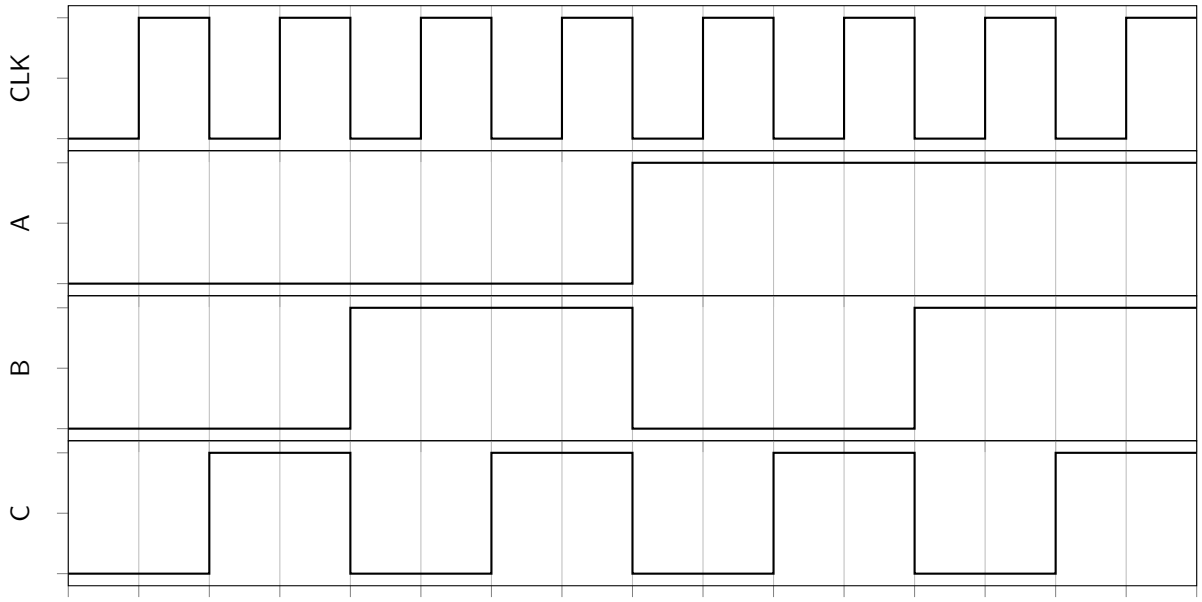


Figure 12.10: Timining diagram for a 3-bit ripple counter.

In this manner, as many flip-flops as desired can be connected together to create larger bit counters.

An issue that arises, however, with stringing together many flip-flops in a ripple counter, is that the gate delay becomes worse with each bit in the sequence. If there are n stages, the accumulated delay is nt_p, where t_p is the gate delay of each stage. This can lead to a situation where the LSB and MSB are no longer synchronized. In that case, a synchronous counter may be a better option.

As mentioned, ripple counters can be created with any type of flip-flop; the input is simply connected in a manner to make the output toggle. The inputs required for each type of flip-flop is shown in Figure 12.11.

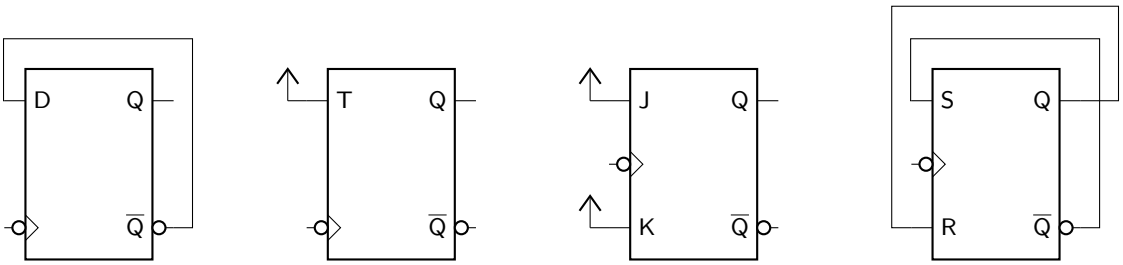


Figure 12.11: Input values for each type of flip-flop to create a toggling output.

While it is not possible to create a ripple counter that counts out of order, it is possible to create a ripple counter that does not count to a power of two minus one. This requires the use of the asynchronous clear pins.

A decade counter, for example, will need to trigger an asynchronous clear after it has completed the sequence zero through nine. Ten, the number after nine, will therefore trigger the clear. That will happen when the flip-flop values are 1010. Because most asynchronous clear pins are active LOW, that would lead to the following expression.

$$CLEAR = (AB'CD')'$$

The schematic for a ripple decade counter created from JK flip-flops is shown in Figure 12.12 (the 7-segment display is not shown).

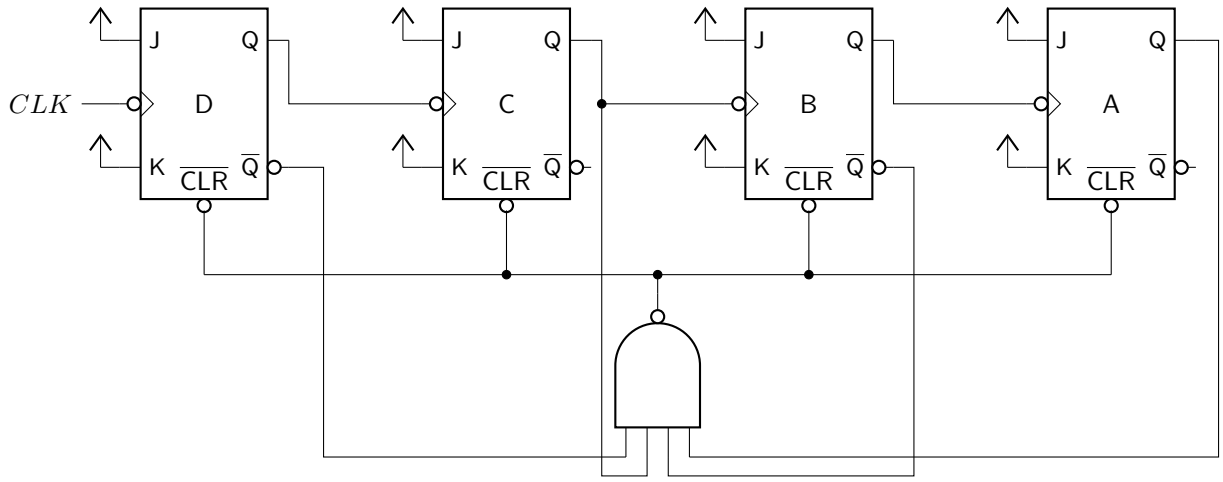


Figure 12.12: Decade ripple counter designed from JK flip-flops.

The process flow for designing a ripple counter follows.

1. Determine the number of flip-flops required. This will be equal to the number of binary bits required to represent the largest number in the count sequence.

2. Connect the output of each flip-flop into the clock input of the next flip-flop. The LSB will have its clock input connected directly to the clock signal.

3. Connect the input of each flip-flop to make it toggle, as shown in Figure 12.11.

4. Connect the preset and clear pins as needed.

   (a) If the counter counts from zero to $2^n - 1$, where n is the number of flip-flops, then the preset and clear pins should be de-asserted.

   (b) If the counter counts from zero to a different number than $2^n - 1$, then the clear pins should be connected to a signal that will cause them to be asserted when the output of the flip-flops is one higher than the maximum count number. Preset pins should be de-asserted.

   (c) If the counter counts from a non-zero value to $2^n - 1$, the preset and clear pins should be connected such that the non-zero initialization value will be asynchronously asserted on the outputs when the output of each flip-flop is zero.

   (d) If the counter counts from a non-zero value to a different number than $2^n - 1$, then the preset and clear pins should be connected such that the non-zero initialization value will be asynchronously asserted on the outputs when the output of the flip-flops is one higher than the maximum count number.

5. (Optional) Test the design in simulation software before building on a breadboard.

12.5 Ring Counters

Ring counters propagate a single logic HIGH value through a sequence of flip-flops. Ring counters are also known as one-hot counters as only one single flip-flop has a HIGH value at a time. The output of each flip-flop in a 4-bit ring counter during eight clock cycles is shown in table 12.1.

Clock Cycle	Flip-Flop Outputs A	B	C	D
1	**1**	0	0	0
2	0	**1**	0	0
3	0	0	**1**	0
4	0	0	0	**1**
5	**1**	0	0	0
6	0	**1**	0	0
7	0	0	**1**	0
8	0	0	0	**1**

Table 12.1: Flip-flop outputs in a 4-bit ring counter through eight subsequent clock cycles.

Ring counters are relatively simple to design, they do not require the derivation of next-state equations that is needed with synchronous counters, or the sometimes complicated preset/clear logic that is needed with ripple counters. However, because there is no state in a one-hot sequential circuit that corresponds to all zeros, it may be required to use the asynchronous preset and clear pins to initialize the ring counter. A schematic of a ring counter using an active-LOW initialization signal (labeled $\overline{\text{RST}}$) is shown in Figure 12.13.

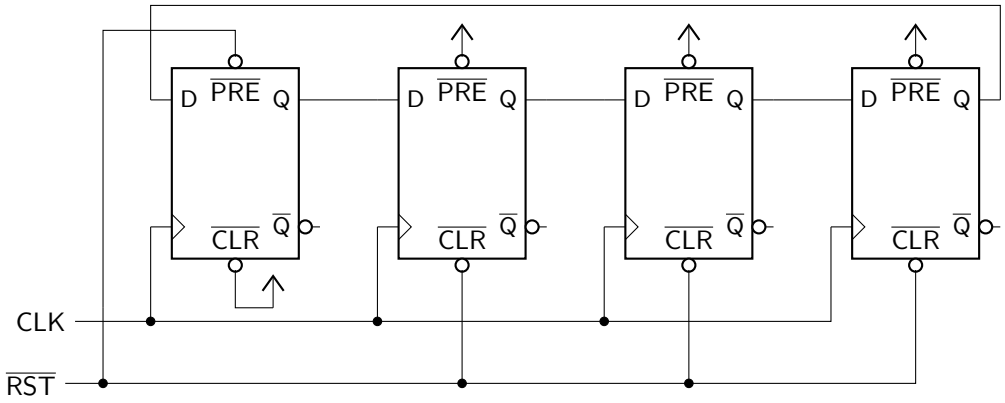


Figure 12.13: Circuit diagram of a 4-bit ring counter using an asynchronous initialization signal.

It is also possible to build a ring counter that is able to initialize without the use of asynchronous flip-flop inputs. This type of ring counter is also able to recover from any possible glitches that would cause it to be situated in a locked-up state (all flip-flops have an output of zero). This type of ring counter feeds the primed output of each flip-flop into an AND gate into the input of the first flip-flop. A schematic of this type of ring counter is shown in Figure 12.14.

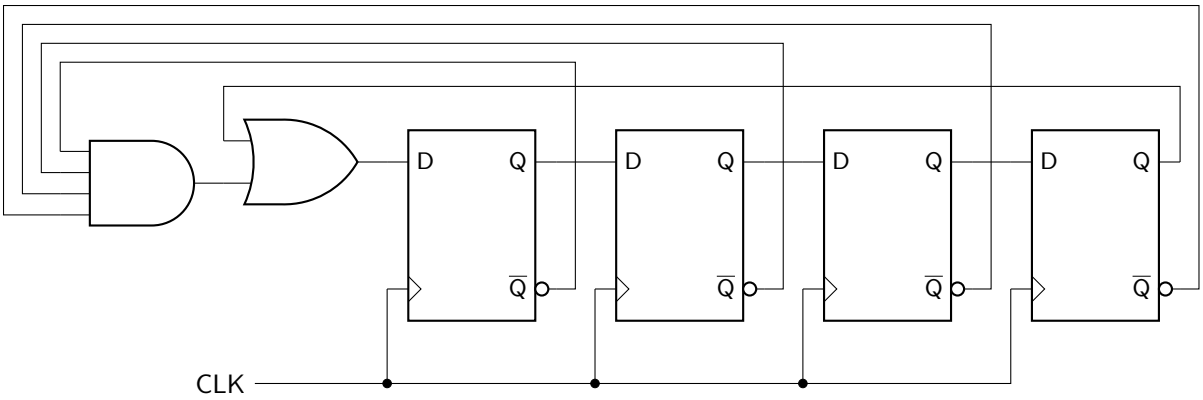


Figure 12.14: Circuit diagram of a 4-bit ring counter using a synchronous initialization signal.

12.6 Johnson Counters

Instead of propagating through a single logic HIGH value, Johnson counters propagate through an increasing sequence of HIGH values surrounded by LOW values. The output of each flip-flop in a 4-bit Johnson counter during nine clock cycles is shown in table 12.2.

Clock Cycle	Flip-Flop Outputs A	B	C	D
1	0	0	0	0
2	**1**	0	0	0
3	**1**	**1**	0	0
4	**1**	**1**	**1**	0
5	**1**	**1**	**1**	**1**
6	0	**1**	**1**	**1**
7	0	0	**1**	**1**
8	0	0	0	**1**
9	0	0	0	0

Table 12.2: Flip-flop outputs in a 4-bit Johnson counter through nine subsequent clock cycles.

Johnson counters are as simple to design as ring counters, and do not require the use of asynchronous inputs or special initialization logic. A schematic of a Johnson counter is shown in Figure 12.15.

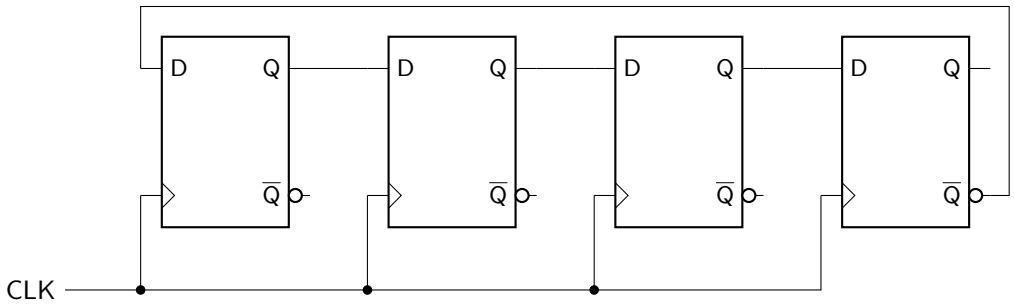


Figure 12.15: Circuit diagram of a 4-bit Johnson counter.

12.7 Example Problems

Registers

1. Design the control logic on the input of a bi-directional shift register. The control inputs are S_1S_0. The control inputs correspond to 00 (hold), 01 (shift left), 10 (shift right), 11 (load).

2. Design a 4-bit SISO register using SR flip-flops.

3. How long does it take the first bit of data to clock through an 8-bit SISO register if the propagation delay on each flip-flop is 25 ns?

4. Draw a diagram of how you could connect together two 74LS194 4-bit bidirectional shift registers to create an 8-bit bidirectional shift register.

5. The sequence 1011 is applied to the input of a 4-bit serial shift register that is initially cleared. What is the state of the register initially and after the first six clock pulses? Assume the input signal becomes zero once the sequence has been shifted in to the first flip-flop.

Synchronous Counters

1. Design a counter corresponding to the state diagram shown in Figure 12.16 out of D flip-flops.

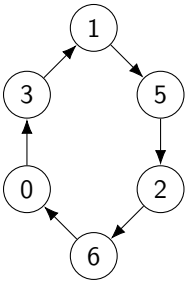


Figure 12.16: State diagram for the synchronous counter to be designed in questions 1 and 2.

2. Design a counter corresponding to the state diagram shown in Figure 12.16 out of T flip-flops.

3. Design a counter corresponding to the state diagram shown in Figure 12.17 out of T flip-flops.

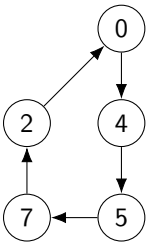


Figure 12.17: State diagram for the synchronous counter to be designed in question 3.

4. Determine the actual state diagram for a synchronous counter corresponding to the state diagram shown in Figure 12.18 assuming that state 000 uses don't care values.

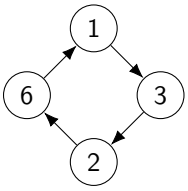


Figure 12.18: State diagram for the synchronous counter to be designed in questions 4 and 5.

5. Design a counter corresponding to the state diagram shown in Figure 12.18, but have state 000 transition to state 110.

Ripple Counters

1. Design a 4-bit ripple counter with D flip-flops.

2. Design a 3-bit ripple down counter with T flip-flops.

3. Design a 4-bit ripple counter that has the state diagram shown in Figure 12.19 using JK flip-flops with active LOW asynchronous preset and clear inputs.

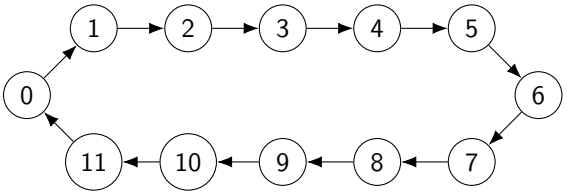


Figure 12.19: State diagram for the ripple counter to be designed in question 3.

4. Design a 4-bit ripple counter that has the state diagram shown in Figure 12.20 using T flip-flops with active LOW asynchronous preset and clear inputs.

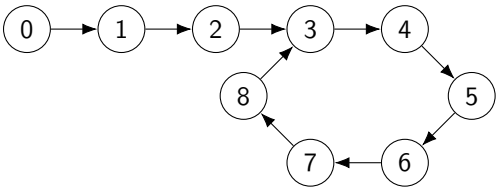


Figure 12.20: State diagram for the ripple counter to be designed in question 4.

5. Draw the waveforms for an entire count cycle for a 3-bit ripple counter that counts from 0–5. Each flip-flop has a delay of 20 ns, and the NAND gate logic that controls the asynchronous clear has a delay of 10 ns. Each flip-flop initially has a value of zero and the clock period is 40 ns.

13 Finite-State Machines

A discussion of finite-state machines will conclude our brief exploration of digital systems in this book. A finite-state machine is a sequential circuit with a finite number of states that occur in a prescribed manner. In fact, the counters discussed in the last chapter are all finite-state machines. That is, they all have a certain number of states, and our state diagram shows us how the circuit moves from one state to the next on a clock trigger. The major difference between a counter and a finite-state machine in general is that with a counter, the flip-flop outputs correspond to the state values. That is not necessarily true in a generic finite-state machine. With finite-state machines in general, we do not usually care what values each flip-flop takes on at any given moment. In addition, while counters are generally free of user inputs (with a few exceptions), finite-state machines can be heavily input-driven.

The two types of finite-state machines are Moore machines and Mealy machines. We will use them to create all manner of finite-state machines, including sequence detectors.

Regardless of the type of finite-state machine being designed, the process is the same.

1. Define all necessary states and their outputs.
2. Create a state diagram (this step can be skipped if desired).
3. Create a state table.
4. Decide which type of flip-flop to use
5. Create binary state assignments for each state.
6. Create a transition table.
7. Derive Boolean algebra expressions for each flip-flop input (called next-state expressions).
8. Derive Boolean algebra expressions for each output signal.

13.1 Moore Machines

A Moore machine is a finite-state machine with synchronous outputs. That is, each state has well-defined values for the output signal(s) and the output signal(s) can only change when a clock triggers a state change. Moore machines can be easier to design than Mealy machines, but generally require more states to implement. A block diagram describing this operation of a Moore machine is shown in Figure 13.1.

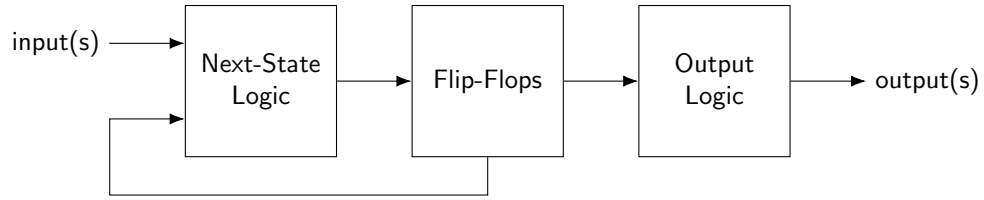


Figure 13.1: Block diagram describing the operation of a Moore machine.

State diagrams are slightly more complicated in Moore machines than they were in counter circuits. Because each state has an output value, the circle with each state has the state name on top and the output value(s) on the bottom. Arrows transitioning between states now depend on the value of input value(s) and there can be multiple arrows extending out of each state as a result of this. An example state diagram for a Moore machine is shown in Figure 13.2. The example state diagram only has one input, so there will be a transition for each possible value of one input, which is to say that two transitions must be defined.

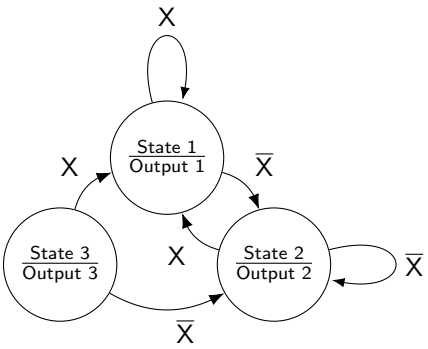


Figure 13.2: Example state diagram for a Moore machine.

A state table is a new concept in this chapter. We did not use them with counters because they were not needed. A state table simply takes the information from a state diagram and puts it into tabular form. Note that alphanumeric state designations are used; no binary values are necessarily present except for possibly to define output values. The state table is also nonspecific to the type of flip-flop that is to be used in the circuit. The state table that corresponds to the diagram given in Figure 13.2 is shown in table 13.1.

Current State	**Next State**		**Output Value(s)**
	X=0	**X=1**	
State 1	State 2	State 1	Output 1
State 2	State 2	State 1	Output 2
State 3	State 2	State 1	Output 3

Table 13.1: Example state table for a Moore machine.

After the state diagram and state table have been derived, the choice of flip-flop can be made and state assignments created for each state. n flip-flops are required to implement a finite-state machine that has 2^n states. Each state assignment will therefore be n bits long. State assignments are necessary in order to

give binary values to each state. It is not possible to conduct Boolean algebra with generic alphanumeric characters (not to mention, the names given to each state can take on literally any value, making them meaningless in the context of actual logic design).

The state assignments are largely arbitrary, but some important considerations must be made. Because flip-flops may initialize with arbitrary values, it is important to designate a reset state that can be asynchronously entered into using the flip-flop clear pins. This means the reset state should have a state assignment corresponding to all zeros. (It is possible to use a different state assignment for the reset state if preset and clear pins are used. However, while clear pins are frequently included in flip-flop chips, preset pins are not always available.) Other guidelines for state assignments will be discussed later on in this chapter.

Transition tables are derived in exactly the same way that they were designed with synchronous counter circuits. You can think of transition tables as a "translation" from written English to binary, as they contain no new information but simply portray prior information in a different way. If anything other than D flip-flops are used, state transitions are used to determine the flip-flop values. Transition tables are used to determine Boolean expressions for each flip-flop input (called next state expressions) and output expressions. Output expressions in Moore machines will only be a function of the states, and will not contain any input variables, due to the synchronous nature of Moore machines.

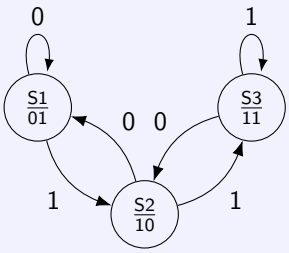
Example: Simple elevator circuit

Design the circuitry for an elevator in a three story building. The elevator can be on any one of the three floors. Each of the floors is a state. There is one input signal X. When X is one it means that the elevator up button has been pressed. When X is zero it means that the elevator down button has been pressed. The output of the finite-state machine is the binary value of the floor number, so that it can be displayed on a 7-segment display.

The first step in designing a finite-state machine is to define each of the states and their output values.

- S_1 – First floor, output: 01
- S_2 – Second floor, output: 10
- S_3 – Third floor, output: 11

The next step is to draw a state diagram.



Note that it is a design choice to keep the elevator on the first floor if the down button is pressed, and to keep the elevator on the third floor if the up button is pressed.

Now that there is a state diagram visually displaying transitions between each state and their outputs, a state table can be derived. Note that it contains no new information; it simply lays out the information in tabular rather than graphical form.

Current State	Next State		Output
	X=0	X=1	YZ
S1	S1	S2	01
S2	S1	S3	10
S3	S2	S3	11

Choosing to design this circuit with D flip-flops, we will choose the following state assignments.

- S_1 – 00 (this will be the default power-on or reset state)
- S_2 – 10
- S_3 – 11

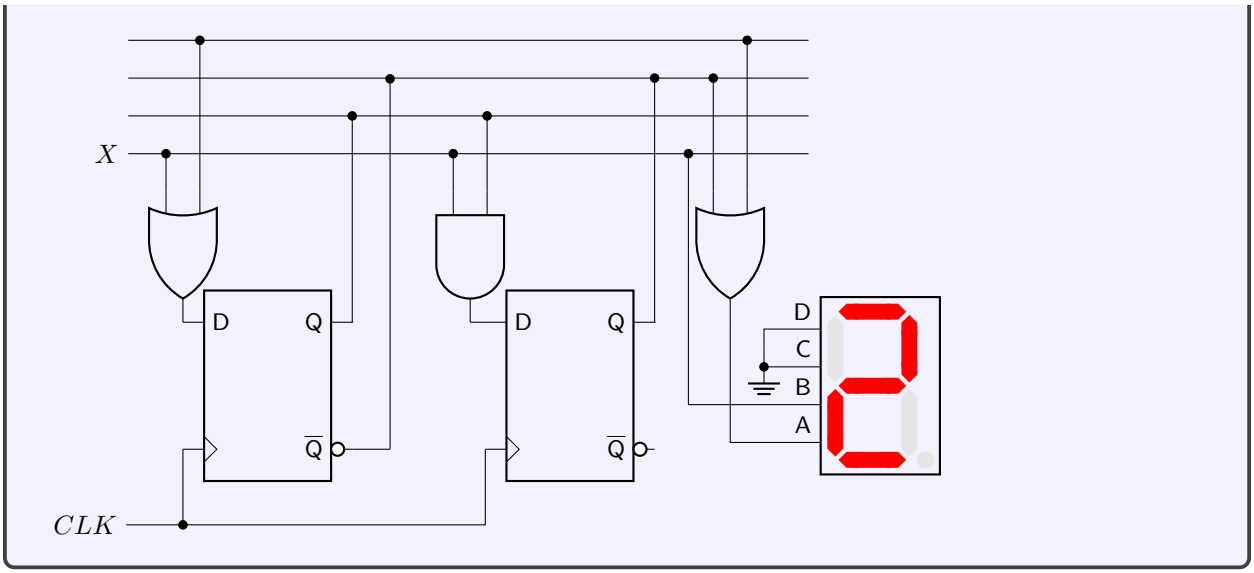
Now a transition table can be created.

Current State	Next State		Output
	X=0	X=1	YZ
00	00	10	01
01	$\times\times$	$\times\times$	$\times\times$
10	00	11	10
11	10	11	11

Using any Boolean algebra minimization, it is possible to find expressions for the input of each flip-flop as well as each output.

$$D_A = X + B$$
$$D_B = XA$$
$$Y = A$$
$$Z = A' + B$$

The circuit diagram for the completed finite-state machine is shown below.



13.2 Mealy Machines

A Mealy machine is a finite-state machine with asynchronous outputs. That is, the output(s) are not defined completely by the state that the circuit is in, but also by the value of the input at any given moment. Changing the input can immediately change the value of the output variable(s). Mealy machines generally have fewer states than otherwise identical Moore machines. Some types of outputs are better suited for Moore machines, and some are better suited for Mealy machines. A block diagram describing the operation of Mealy machines is shown in Figure 13.3.

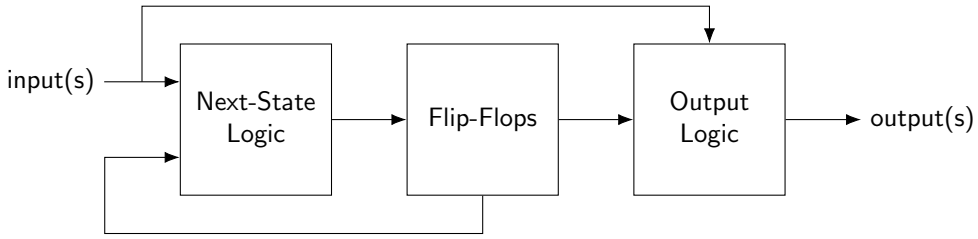


Figure 13.3: Block diagram describing the operation of a Mealy machine.

State diagrams differ in a Mealy machine in that the outputs are not defined with the states, but on the transitions between states. Each transition arrow will now contain the input value(s) on top and the output value(s) on the bottom of the part that looks like a fraction. An example Mealy machine state diagram is shown in Figure 13.4.

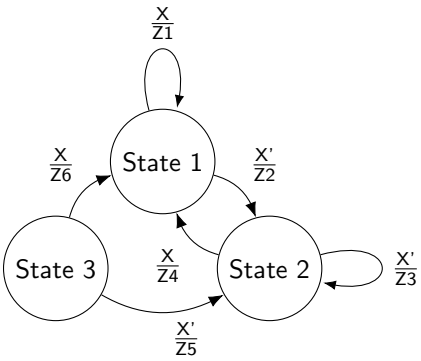


Figure 13.4: Example state diagram for a Mealy machine.

State tables still contain exactly the same information as state diagrams. In the case of a Mealy machine, because the output is a function of the state and the input(s), there needs to be more than one column describing the outputs. The state table that corresponds to the diagram in Figure 13.4 is given in table 13.2.

Current State	Next State		Output Value(s)	
	X=0	X=1	X=0	X=1
State 1	State 2	State 1	Z2	Z1
State 2	State 2	State 1	Z3	Z4
State 3	State 2	State 1	Z5	Z6

Table 13.2: Example state table for a Mealy machine.

After the state table is completed, the rest of the process of solving for the expressions is identical to that of a Moore machine. A flip-flop type is chosen, state assignments are determined, a transition table derived, and expressions are derived using a minimization technique.

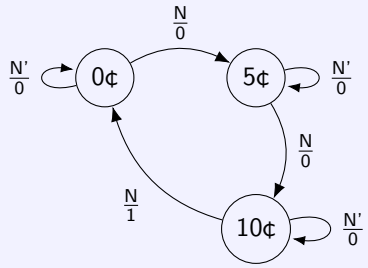
Example: Simple vending machine circuit

We will design the circuitry for a very simple vending machine. Every item in the vending machine costs 15¢, and the vending machine only accepts nickels. The input variable N is zero when no nickel has been inserted, and is one when a nickel has been inserted. The output variable V is zero when insufficient funds have been inserted, and V is one when enough funds have been inserted to vend the snack out of the machine.

Because the outputs change on transitions between states, it is only necessary to have three states. (A 15¢ state is not necessary, because as soon as the last nickel is inserted, the vending machine will go back to its reset mode.)

- 0¢ – No money has been inserted (this will be the reset state)
- 5¢ – Five cents have been inserted
- 10¢ – Ten cents have been inserted

We can now create a state diagram. Note that alphanumeric notation is used on the input values (N and N' instead of 1 and 0). Either method can be used to define inputs in state diagrams.



The state table simply takes the information from the state diagram and puts it into tabular form.

Current State	Next State		Output (V)	
	N=0	N=1	N=0	N=1
0¢	0¢	5¢	0	0
5¢	5¢	10¢	0	0
10¢	10¢	0¢	0	1

We will use T flip-flops to design this vending machine circuit. The state assignments are mostly arbitrary except for our choice of the 00 state. It would make the most sense for the vending machine to power on into the 0¢ state.

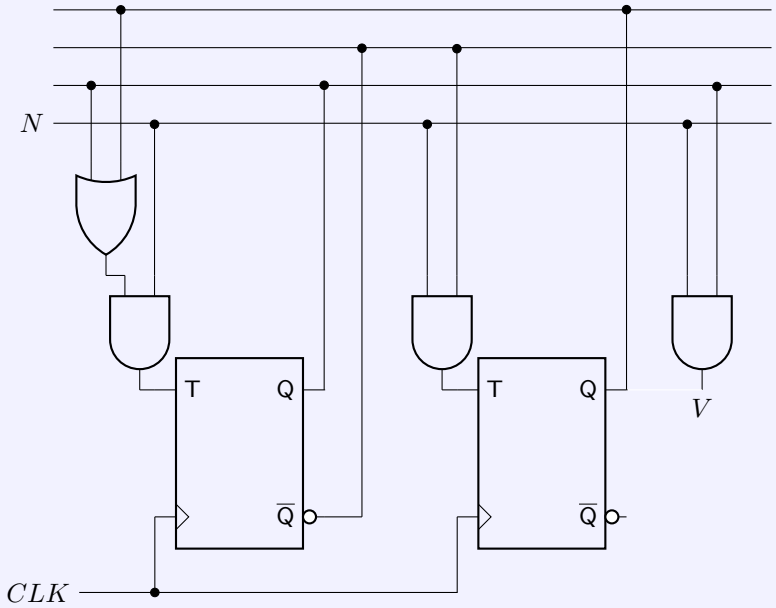
- 0¢ – 00
- 5¢ – 01
- 10¢ – 10

We can put this all together into a transition table.

Current State	Next State		Flip-Flop Values		Output (V)	
	N=0	N=1	N=0	N=1	N=0	N=1
00	00	01	00	01	0	0
01	01	10	00	11	0	0
10	10	00	00	10	0	1
11	××	××	××	××	×	×

The expressions for each flip-flop and output are given below.

$$T_A = N(A + B)$$
$$T_B = NA'$$
$$V = NA$$



13.3 Sequential Circuit Clock Frequency Constraints

Frequency describes the number of rising edges (or falling edges) of a clock signal that occur every second. It is defined by the symbol f and has a unit of Hz (hertz). Period, on the other hand, describes the amount of time that elapses between subsequent rising edges (or falling edges) in a clock signal. Period is defined by the symbol T and has a unit of s (seconds). Frequency and period are reciprocal values. That is,

$$f = \frac{1}{T}.$$

It is also important to define each of the possible limiting delays that occur in a synchronous circuit. These are defined in Table 13.3.

Symbol	Name	Description
t_p	Propagation delay	The amount of time it takes the output of a flip-flop to become valid after the clock ticks.
t_c	Combinational delay	The amount of time it takes the input of a flip-flop to become valid due to any combinational logic delay.
t_{su}	Setup time	The amount of time a flip-flop input must be stable before a triggering clock edge.
t_h	Hold time	The amount of time a flip-flop input must be stable after a triggering clock edge.
t_{sk}	Clock skew	The amount of delay between flip-flop clock inputs (described below).

Table 13.3: Delays that exist in a synchronous circuit.

Clock skew refers to the situation where not all flip-flops in a sequential circuit receive an update-triggering signal at the clock input at the same time. This could be substantial if there are big differences in wire length from one flip-flop clock input to another. While clock skew depends on the exact circuit layout, a datasheet can be used to determine setup time, hold time, propagation delay, and combinational delay for each circuit.

If the minimum clock period is determined for a circuit (the fastest amount of time that can elapse between subsequent triggering edges), the largest allowable frequency can be calculated by taking the reciprocal of the minimum period. It is important to ensure that all circuit elements in a synchronous circuit have a suitable amount of time to generate valid signals every clock cycle. For example, given the diagram shown in Figure 13.5, we can discuss the issues that can occur if FF 1 and FF 2 are clocked too quickly.

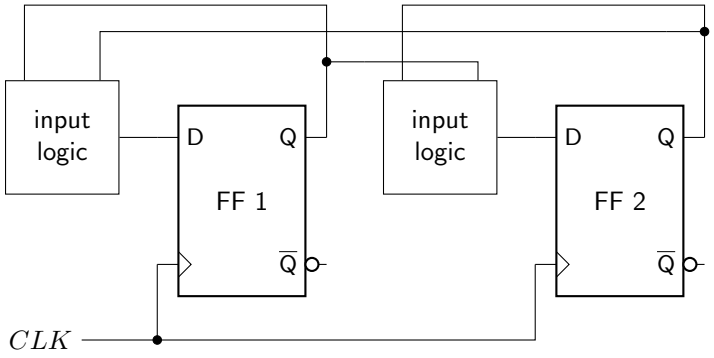


Figure 13.5: Schematic of a finite-state machine consisting of two flip-flops.

A setup violation occurs if one of the flip-flops (for example, FF 1, as depicted in Figure 13.5) clocks,

causing the input of the other flip-flop (FF 2, which has combinational logic potentially including the output value of FF 1 as an input) to be unstable before FF 2 clocks on the next clock cycle. Essentially, the clock is ticking too quickly for input values to become stable on the next cycle. This problem can be addressed by ensuring the clock speed is not too fast. In order to prevent setup violations,

$$t_{min} = t_p + t_c + t_{su} + t_{sk}$$

must be satisfied. This will limit the fastest allowable clock frequency.

A hold violation occurs if FF 1 (depicted in Figure 13.5) clocks first and the input of FF 2 (which has combinational logic potentially including the output value of FF 1 as an input) changes before the previous data can be clocked. This problem cannot be addressed by using a slower clock frequency. In order to prevent hold violations,

$$t_p + t_c \geq t_{sk} + t_h$$

must be satisfied. Note that a hold violation also applies if FF 2 clocks prior to FF 1 on the same clock cycle.

13.4 Sequence Detectors

A sequence detector is capable of detecting one or more strings of binary numbers. Usually, when the output is detected, the output becomes a one, otherwise the output is zero. (If we want to design active LOW logic, the output would be a one if not detected, or a zero if detected.) There are three types of sequence detectors, each of which has its own application.

13.4.1 Overlapping Sliding Window

Sliding window detectors have to do with the fact that a "window" of the same bit length as the sequence slides along looking for the sequence. If the sequence shows up in that window, then the output will become a one. If the window shows binary numbers that do not correspond to the sequence, then the output will be a zero. The graphic shown in Figure 13.6 depicts a 4-bit sliding window.

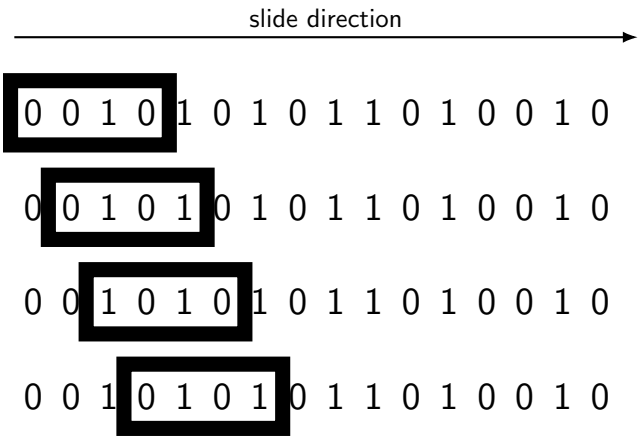


Figure 13.6: Graphic depiction of how a 4-bit window slides along a series of numbers to look for a pattern.

An overlapping sliding window detector indicates that a sequence that overlaps with another sequence will lead to an output of one. The same sequence used in Figure 13.6 would therefore lead to the sequences highlighted in red squares as being detected, shown in Figure 13.7. Output values are also shown at each step in the sequence.

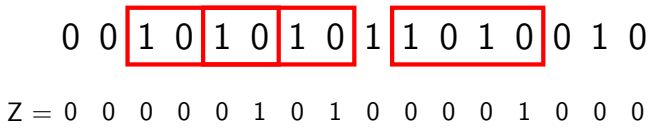


Figure 13.7: Detected sequences of 1010 inside the test pattern.

Generally, with sliding window detectors, states are defined for the reset state (nothing has been detected yet), and incrementally for each set of values being detected. For example, if detecting 1010, we would have the following states for a Moore machine:

- reset (output = 0)
- 1 has been detected (output = 0)
- 10 has been detected (output = 0)
- 101 has been detected (output = 0)
- 1010 has been detected (output = 1)

With Mealy machines, one less state is required to implement a sequence detector because of the fact that outputs are changed on transitions and are not connected to states. A Mealy machine detecting 1010 would therefore have the following states:

- reset

- 1 has been detected
- 10 has been detected
- 101 has been detected

When deriving state diagrams, it can be simplest to start at the reset state and note that the input values must be equal to the sequence values in order to move to the next state. These are the easiest transitions to determine in the state diagram. Figure 13.8 shows these state transitions for detecting 1010 for both a Moore machine and a Mealy machine.

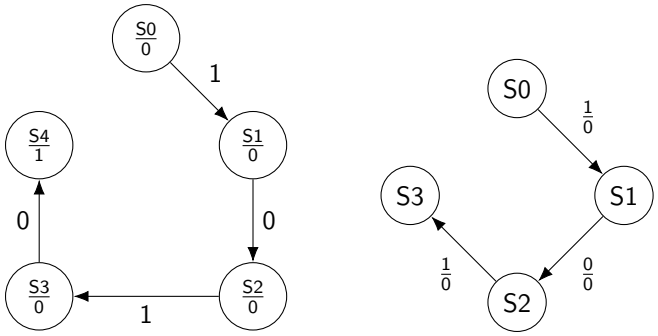


Figure 13.8: Simple sequence transitions for a Moore machine (left) and Mealy machine (right) detecting 1010.

The state diagrams shown in Figure 13.8 are not complete. There must be transitions corresponding to all valid input values. In this case, it means there must be two transitions defined for each state. The status of the sequence in each state must be appraised, and based on the input value, if there is a state that corresponds to the sequence that would arise, the state machine should go to that state.

For the Moore machine, the following transitions are derived, leading to the state diagram shown in Figure 13.9.

- reset (S_0) – 0 would lead to no sequence that is defined as a state: stay in the reset state
- 1 detected (S_1) – 1 would lead to a sequence of 11: because 11010 would lead to a sequence detection, the circuit should remain in state S_1
- 10 detected (S_2) – 0 would lead to a sequence of 100 which does not correspond to any defined state: go back to reset
- 101 detected (S_3) – 1 would lead to a sequence of 1011 which does not completely reset the sequence but should bring the circuit back to the 1 state which is S_1
- 1010 detected (S_4)
  - 0 would lead to 10100, which completely resets the sequence: go back to reset
  - 1 would lead to 10101, which corresponds to a 101: go to state S_3

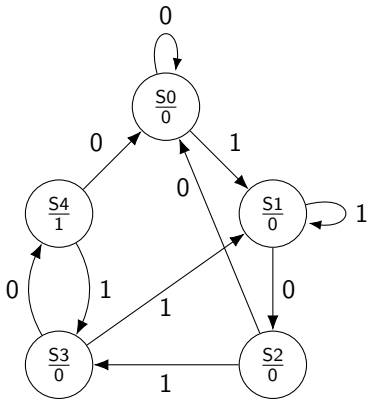


Figure 13.9: State diagram of the overlapping sliding window Moore machine detector (sequence 1010).

At this point, it is helpful to refer to the test sequence that we derived above and follow it along using the state diagram to check that the state diagram is correct. This step is very important in helping to understand the circuit better. Note that the test sequence was cleverly crafted to include overlapping sequences, non-overlapping sequences, and not-quite sequences to test what would happen in each of these situations.

For the Mealy machine, the following transitions are derived, leading to the state diagram shown in Figure 13.10.

- reset (S_0) – 0 would lead to no sequence that is defined as a state: stay in the reset state
- 1 detected (S_1) – 1 would lead to a sequence of 11: because 11010 would lead to a sequence detection, the circuit should remain in state S_1
- 10 detected (S_2) – 0 would lead to a sequence of 100 which does not correspond to any defined state: go back to reset
- 101 detected (S_3)
  - 0 would lead to 1010, which is both the correct sequence (leading to an output of 1) and also is halfway to a second 1010 sequence as the last two bits, 10, are part of the sequence: go to state S_2
  - 1 would lead to 1011, which brings the sequence back to a 1 state: go to state S_1

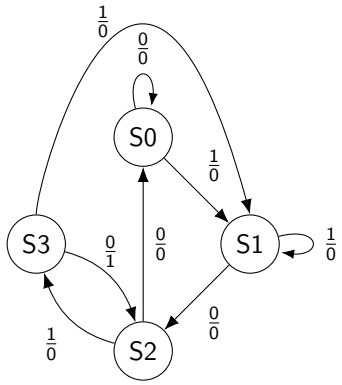


Figure 13.10: State diagram of the overlapping sliding window Mealy machine detector (sequence 1010).

13.4.2 Non-Overlapping Sliding Window

With a non-overlapping detector, sequences cannot overlap with each other. This means we need to be vigilant when determining the transitions between states on the state diagrams. Assume we will still detect the sequence 1010. The same sequence used in Figure 13.6 would therefore lead to the sequences highlighted in red squares as being detected, shown in Figure 13.11. The output values are also shown.

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

Z = 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0

Figure 13.11: Detected sequences of 1010 inside the test pattern (overlapping sequences are no longer allowed).

The same states as defined with an overlapping detector can be used for both Moore machines and Mealy machines, and the partial state diagrams shown in Figure 13.8 are still valid. The remaining transitions must be more carefully derived.

For the Moore machine, the only transition that changes is from state S_4 when a one is detected. Because we cannot allow the overlap of 1010 with the next one, the circuit must transition to S_1. the following transitions are derived, leading to the state diagram shown in Figure 13.12.

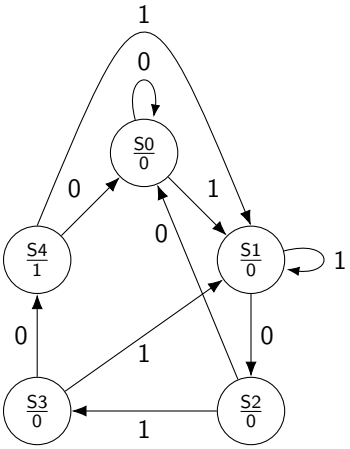


Figure 13.12: State diagram of the non-overlapping sliding window Moore machine detector (sequence 1010).

For the Mealy machine, the only transition that changes is when a zero is detected after the S_3 state. We do not want to allow it to overlap, so the circuit must go back to the reset state instead. The non-overlapping state diagram is shown in Figure 13.13.

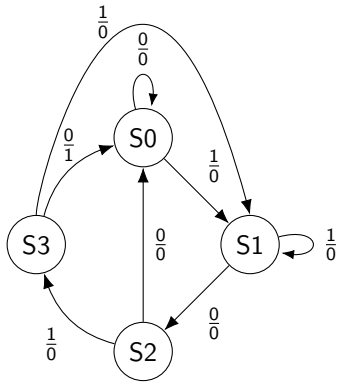


Figure 13.13: State diagram of the non-overlapping sliding window Mealy machine detector (sequence 1010).

13.4.3 Disjoint Window

In a disjoint window detector, rather than the "window" sliding along the input sequence, the window moves over in increments of n bits, where n is the length of the sequence that is being detected. The same test sequence from before, looking for 1010, would result in the graphic shown in Figure 13.14. The output values are shown as well.

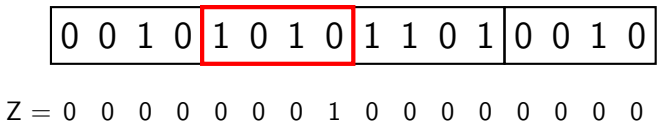


Figure 13.14: Detected sequences inside of a disjoint window detector.

When designing a disjoint window detector, there is no worry about overlaps. The only thing to consider is whether or not the sequence is correct so far or not. There is a "good" path corresponding to the sequence being detected, and a "bad" path corresponding to the sequence not being detected. The only way to remain on the "good" path is to continue receiving the correct inputs in the correct order. Any time the wrong input is received, the circuit transitions over to the bad path. To detect 1010, the Moore machine state diagram is shown in Figure 13.15. Notice that every time the state changes, the same level of the state diagram is reached. This insures that the window remains disjoint rather than sliding.

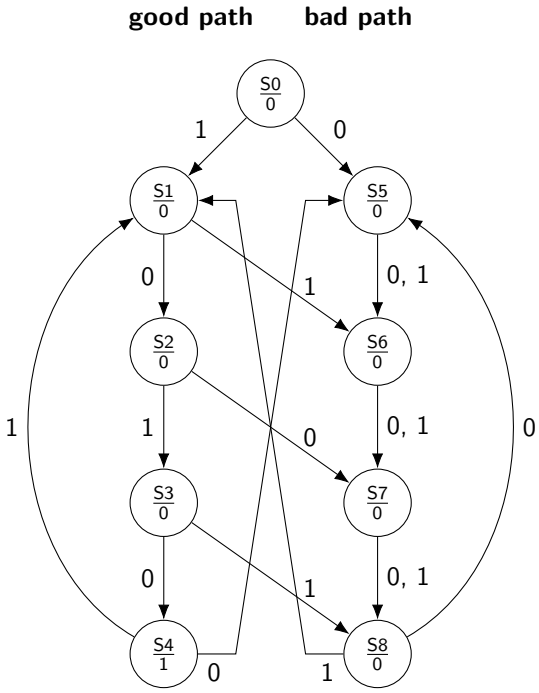


Figure 13.15: State diagram of the disjoint window Moore machine detector (sequence 1010).

The Mealy machine diagram, shown in Figure 13.16 is much the same as the Moore machine but with fewer states, as is the case with Mealy machines. Otherwise the sequencing of states is very similar.

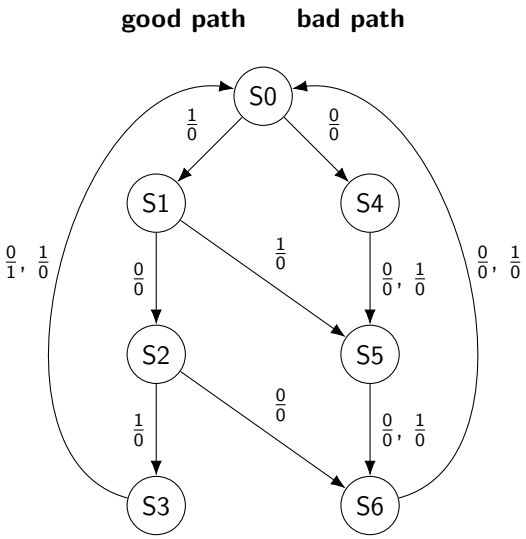


Figure 13.16: State diagram of the disjoint window Mealy machine detector (sequence 1010).

13.4.4 More Complicated Sequence Detectors

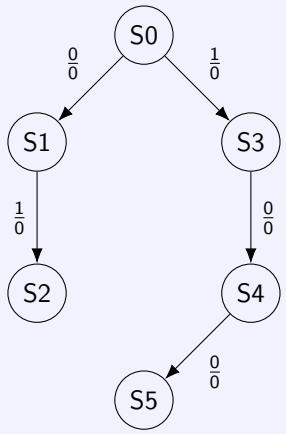
It is possible to design state machines that are capable of detecting multiple sequences. It is also possible to design a state machine that is designed to detect a sequence only once and ignore subsequent detections. Or we could create a state machine that, upon detection of a sequence, then looks for a second sequence. We will look at examples of each of these.

Example: Detect two different sequences

This state machine will be a Mealy machine capable of detecting two independent sequences: 010 and 1001. Overlapping is allowed. First we define each state. Note that the first three states help to define the first sequence (010), and the last three states along with the reset state help to define the second sequence (1001).

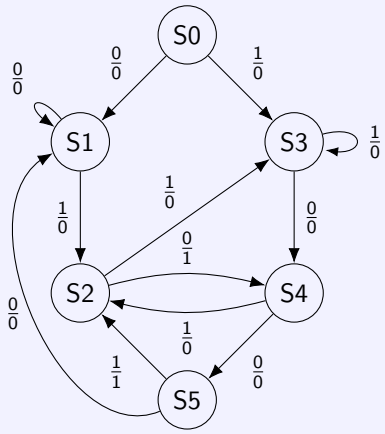
- reset
- 0 has been detected
- 01 has been detected
- 1 has been detected
- 10 has been detected
- 100 has been detected

Each of the simple transitions can now be placed onto the state diagram.



Now all of the subsequent transitions can be thought out.

- 0 has been detected (S_1) – 0 would lead to the possibility of a 010 detection still (0010 for example), so the circuit should remain in state S_1
- 01 has been detected (S_2) – if a 0 is detected the output should be 1, and we should go to S_4 because the most recently entered digits are 10; if a 1 is detected the circuit should to go S_3 because 1 is the most recent useful sequence
- 1 has been detected (S_3) –if a 1 is detected, the possibility of 1001 is still possible, so the circuit should remain in state S_3
- 10 has been detected (S_4) – if a 1 is detected, the most recent useful sequence is 01, so the circuit will go to state S_2
- 100 has been detected (S_5) – if a 0 is detected the circuit should go to state S_1; if a 1 is detected the output should be 1 and the circuit should go to S_2 because the most recent useful sequence is 01

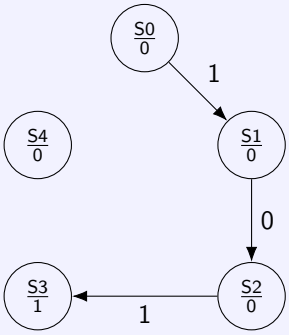


Example: Detect a sequence once and never again

Design a Moore machine that has an output of one if 101 is detected for the first time. After that, the output will remain zero.

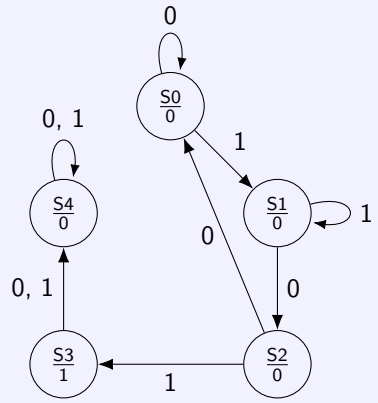
We first define each of the states, and then determine the simple transitions.

- reset (output = 0)
- 101 not yet detected, 1 detected (output = 0)
- 101 not yet detected, 10 detected (output = 0)
- 101 detected for the first time (output = 1)
- any input value once 101 has been detected (output = 0)



Now the other transitions can be analyzed, determined, and added to the state diagram.

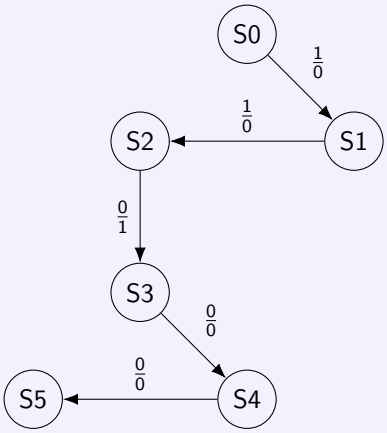
- reset (S_0) – 0 would lead to no sequence that is defined as a state: stay in the reset state
- 101 not yet detected, 1 detected (S_1) – 1 would lead to the possibility of 1101, so the circuit should stay in state S_1
- 101 not yet detected, 10 detected (S_2) – 0 would lead to a reset of the sequence: go to S_0
- 101 detected for the first time (S_3) – regardless of the input value received, the circuit should go to S_4 to prevent any other sequences from being detected
- any input values once 101 has been detected (S_4) – regardless of input value, the circuit needs to stay in this state to prevent any further detections



Example: Detect a sequence once then detect a second sequence multiple times

Design a Mealy machine that detects 100 once, then after that detects 001 multiple times. No overlap between the detection of 100 and 001 is allowed.

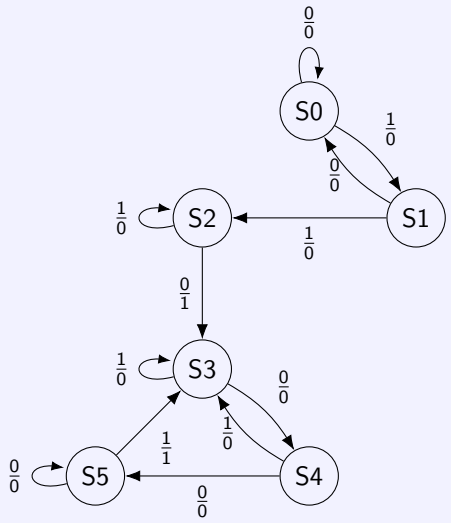
- reset
- 110 not yet detected, 1 detected
- 110 not yet detected, 11 detected
- 110 has been detected, no progress towards the second sequence
- 110 has been detected, 0 detected
- 110 has been detected, 00 detected



Analyze the other transitions.

- reset (S_0) – if 0 is detected, no progress is being made, stay in the reset state

- 110 not yet detected, 1 detected (S_1) – if 0 is detected, all progress toward the sequence is lost, go back to reset
- 110 not yet detected, 11 detected (S_2) – if 1 is detected, it's still possible to get to 110 (1110 or 11110 for example), so the circuit should stay put in S_2
- 110 has been detected, no progress toward second goal (S_3) – if 1 is detected, no progress is made toward the second goal, so the circuit should stay in S_3
- 110 detected, 0 detected (S_4) – progress toward the second sequence has been ruined so the circuit should return to S_3
- 110 detected, 00 detected (S_5) – if a 0 is detected, it is still possible to detect 001 (0001 or 00001 for example), so the circuit should stay in S_5; if a 1 is detected, the output should be 1 as the sequence was detected, and the circuit should return to S_3



13.5 Sequential Comparison Logic

As discussed in section 12.2, using registers and sequential logic can substantially decrease the footprint and hardware required to implement addition and multiplication. It is also possible to use sequential logic to generate a finite-state machine to compare two binary numbers.

There are commercially available comparator chips that can be purchased (notably, the 74LS85 is a 4-bit magnitude comparator), all of which have a limit to the number of bits that can be compared. These chips generally have separate outputs that are asserted if input A is greater than input B, if input A is less than input B, or if the two inputs are equal. It is possible to cascade multiple chips together to increase the bit-width, however, this creates propagation delay between stages and increases the footprint of the overall

circuit.

Generating an expression using an SOP or POS implementation quickly becomes unwieldy. When two n-bit numbers are compared, there are

- 2^n minterms to realize the output expression $A = B$,
- $2^{2n-1} - 2^{n-1}$ minterms to realize the output expression $A > B$, and
- $2^{2n-1} - 2^{n-1}$ minterms to realize the output expression $A < B$.

Comparing two n-bit binary numbers sequentially is a relatively straightforward process using a finite-state machine and a shift register. The MSB of each number is compared together, followed by the next significant bit, and so on, iterating for n clock cycles (or stopping once one binary number is found to be greater than the other). Each clock cycle, the shift register shifts so that the next bit is input into the finite-state machine input circuitry. The sequential approach requires more time to execute but uses a smaller footprint and allows for the comparison of inputs of as many bits as needed without any change in the register control logic.

Given two binary numbers A and B, the state diagram in Figure 13.17 can be used to generate control logic for a sequential magnitude comparator. The inputs of the Moore machine are A_i and B_i (the i^{th} bit of each binary number), and the outputs are $A > B$, $A = B$, and $A < B$. The states are defined as follows.

- S0 – A and B are equal.
- S1 – A is greater than B.
- S2 – A is less than B.

Once the comparator finds that one number is larger than the other, no more bits need be checked.

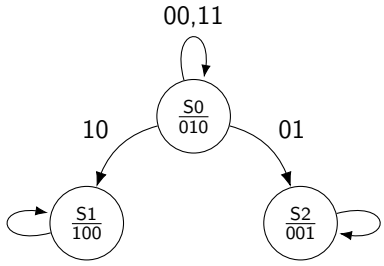


Figure 13.17: State diagram for comparing the magnitude of two n-bit binary numbers.

13.6 Finite-State Machine Derivation

Now that we have discussed the fundamentals of Mealy and Moore machine, as well as how to create sequence detectors, we can look at some more complicated finite-state machine scenarios. Once we can deal with the following situations, any finite-state machine can be designed.

13.6.1 Multiple Outputs

Technically, the Moore machine elevator example had two outputs as the output displayed the floor number which required two bits to specify. We will do one more example to demonstrate the use of multiple outputs.

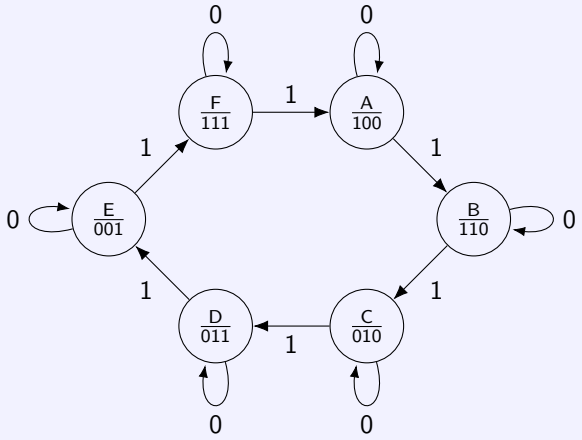
When there are multiple outputs, it is important to be specific about the order in which the outputs are specified on the state diagram.

Example: Changing the color of an RGB LED every time a button is pressed

This finite-state machine will progress through a sequence of different colors every time a button is pressed. This makes for an interesting example, as the button press can serve as the clock signal on the flip-flops as well as the input value. Because each LED color corresponds to a state, this must be designed as a Moore machine. The sequence of colors will be red → yellow → green → cyan → blue → white → back to red again.

Each of the outputs in the state diagram is ordered as shown below. That is, when there are three numbers, the first would be the output of the red anode of the LED (F_R), the second would be the output of the green anode of the LED (F_G), and the third would be the output of the blue anode of the LED (F_B).

The full state diagram is relatively simple to derive, as no button press ($X = 0$) leads to no change in color, and a button press ($X = 1$) leads to a change in color.



The state table is not derived in any different manner than how they were derived in previous examples.

Current State	Next State		Outputs		
	X=0	X=1	Fr	Fg	Fb
A	A	B	1	0	0
B	B	C	1	1	0
C	C	D	0	1	0
D	D	E	0	1	1
E	E	F	0	0	1
F	F	A	1	1	1

Use T flip-flops to design this circuit. The state assignments are mostly arbitrary except for our choice of the 000 state, which will correspond to state A.

- A – 000
- B – 001
- C – 010
- D – 011
- E – 100
- F – 101

Put this all together into a transition table.

Current State	Next State		Flip-Flop Values		Outputs		
	X=0	X=1	X=0	X=1	Fr	Fg	Fb
000	000	001	000	001	1	0	0
001	001	010	000	011	1	1	0
010	010	011	000	111	0	1	0
011	011	100	000	001	0	1	1
100	100	101	000	001	0	0	1
101	101	000	000	101	1	1	1
110	× × ×	× × ×	× × ×	× × ×	×	×	×
111	× × ×	× × ×	× × ×	× × ×	×	×	×

The Boolean expressions for the flip-flops and outputs can be derived from k-maps, Quine-

McCluskey, or Boolean algebra.

$$T_A = XC(A + B)$$
$$T_B = XA'C$$
$$T_C = X$$
$$F_R = B'(A' + C)$$
$$F_G = B + C$$
$$F_B = A + BC$$

13.6.2 Multiple Inputs

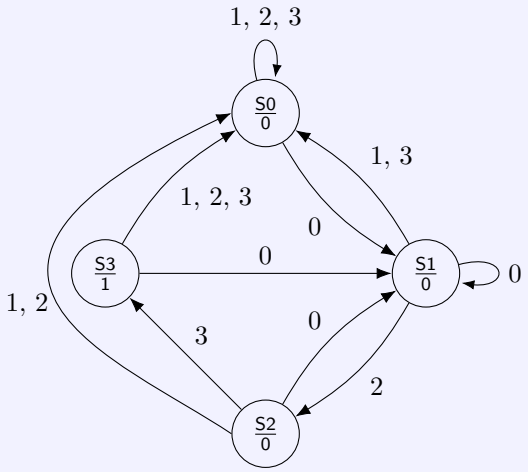
So far, each of the examples we have analyzed have only had a single input. However it is possible to have more than one input. In the case of multiple inputs, more next-state columns are required in a state table. For every n inputs, there will be 2^n next state columns on the state table.

When using multiple inputs, it is very important to ensure that transitions corresponding to each possible input combination accounted for. (It is much easier to account for each transition when there is only one input value!) A method for checking for the completeness of state diagrams are described in a subsequent section of this chapter.

Example: Sequence detector capable of detecting 2-bit binary numbers

Design a Moore machine that detects the sequence 0–2–3 (00–10–11 in binary). Each of the states is described below.

- reset (output = 0)
- 0 detected (output = 0)
- 0–2 detected (output = 0)
- 0–2–3 detected (output = 1)



Current State	Next State				Output
	XY=00	XY=01	XY=10	XY=11	Z
S0	S1	S0	S0	S0	0
S1	S1	S0	S2	S0	0
S2	S1	S0	S0	S3	0
S3	S1	S0	S0	S0	1

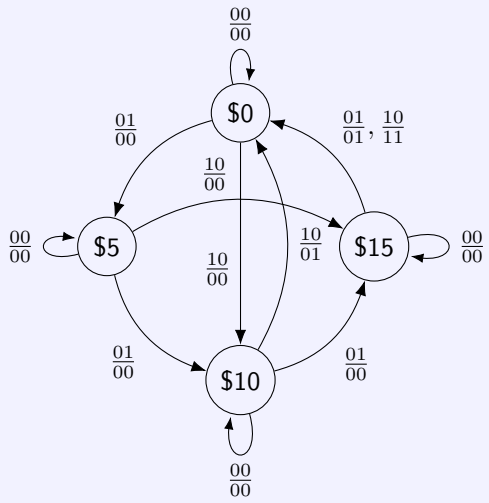
13.6.3 Incompletely Specified State Machines

There are different ways in which a finite-state machine can be incompletely specified. We have already seen what happens when we have to use n flip-flops and fewer than 2^n states; in those situations the state assignments that are unused become don't cares.

Another way in which a finite-state machine can be incompletely specified is in the input values. It's possible that there may be some binary values corresponding to the input variables that are not accounted for. In those situations, the next states, outputs, and flip-flop values will be don't cares.

Example: Slightly more complicated vending machine

Design a Mealy machine vending machine that has a candy bar that costs \$20. Change is given ($G = 1$) when more than \$20 has been inserted, otherwise no change is given. The candy bar is vended ($V = 1$) when at least \$20 has been inserted into the vending machine, otherwise no candy bar is vended ($V = 0$). The output is shown on the state diagram in the form GV. The input is in the form XY where 00 indicates no money has been inserted, 01 indicates that a \$5 bill has been inserted, 10 indicates that a \$10 bill has been inserted, and 11 will never happen. Each state corresponds to how much money has been inserted into the vending machine.



Current State	Next State				Outputs (GV)			
	XY=00	XY=01	XY=10	XY=11	XY=00	XY=01	XY=10	XY=11
$0	$0	$5	$10	×	00	00	00	××
$5	$5	$10	$15	×	00	00	00	××
$10	$10	$15	$0	×	00	00	01	××
$15	$15	$0	$0	×	00	01	11	××

13.6.4 Alphanumeric State Diagrams and State Diagram Completeness

Sometimes, it can be simpler to use alphanumeric notation for input variables rather than to define binary numbers for each transition on a state diagram.

Let's say we have a state machine that corresponds to three positions that a remote controlled vehicle can be in. The vehicle moves forward when the forward button (F) is pressed and moved backward when the reverse button (R) is pressed. A simplistic version of the state diagram is shown in Figure 13.18.

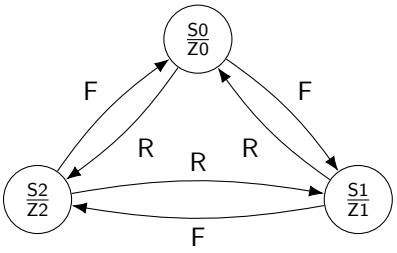


Figure 13.18: Simplistic state diagram corresponding to a vehicle moving in forward or reverse through each position.

However, this state diagram has some problems as currently defined. What happens if both inputs F and R are zero simultaneously? That transition is not defined on the state diagram. What if both inputs F and R are one simultaneously? That transition is over-defined on the state diagram, as the diagram shows

movement in both directions, which shouldn't be allowed. Priority should be given to one direction. As circuit designers, we can choose to design our circuit however we wish to overcome these issues. For the sake of showing a completed example, we will make the vehicle stay in its current state if both inputs are zero simultaneously. We will also give priority to the F input, so F must be zero for reverse to occur. The updated state diagram is shown in Figure 13.19.

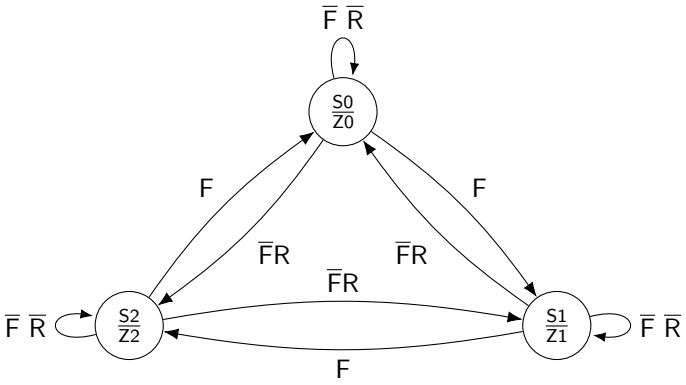


Figure 13.19: Updated state diagram corresponding to vehicle motion.

It can be difficult, and at times, unwieldy, to simply visually inspect the values of each transition to determine if the states are complete or not. In that case, we can use an algorithm to check for state diagram completeness.

To ensure that there are a sufficient number of transitions defined at each state, take the logical OR of every transition coming out of each state. The result should be one. If the result is anything other than one, then there is at least one combination of input values not accounted for.

Using the example given in Figure 13.18, we can take the logical OR of each transition coming out of state S_0.

$$F + R$$

Because this does not reduce to one, there are states missing. We can now see how the newly defined S_0 from Figure 13.19 holds up to scrutiny.

$$\begin{aligned} F'R' + F + F'R &= \\ F' + F &= \\ &= 1 \end{aligned}$$

To ensure that there aren't too many transitions defined at each state, take the logical AND of every combination of transitions at each state. Each of these AND operations should result in zero.

We can analyze the transitions from S_0 from Figure 13.18.

$$FR$$

This implies that there are redundancies in the transitions. We saw that this corresponded to the issue that arises when both F and R are one at the same time.

This problem is resolved in Figure 13.19, as shown.

$$(F)(F'R') = 0$$
$$(F)(F'R) = 0$$
$$(F'R')(F'R) = 0$$

13.7 Reduction of State Tables

We saw how to ensure that a sufficient number of transitions (but not too many!) are defined at each state. But how can we ensure that we have the correct number of states defined in our finite-state machine in the first place? What happens if we have over-defined some states in our state machine that are otherwise redundant? There are two methods that we can use to reduce state tables to eliminate redundant states.

First, it is important to understand what defines redundant states. Two states are said to be equivalent if they have the same output values (given the same input values, if it's a Mealy machine), and if the next states for each input value are equivalent.

13.7.1 Visual Reduction of State Tables

While this method is not foolproof (as we will see in the next section about implication tables), it is a good start to help in the process of eliminating redundant states. Let's consider the crafting of a disjoint window Mealy machine that has an output of 1 when either sequence 0101 or 1001 is detected. If every possible input combination is analyzed, table 13.4 shows the resulting state table.

Input Sequence	Current State	Next State		Output	
		X=0	X=1	X=0	X=1
reset	A	B	C	0	0
0	B	D	E	0	0
1	C	F	G	0	0
00	D	H	I	0	0
01	E	J	K	0	0
10	F	L	M	0	0
11	G	N	P	0	0
000	H	A	A	0	0
001	I	A	A	0	0
010	J	A	A	0	1
011	K	A	A	0	0
100	L	A	A	0	1
101	M	A	A	0	0
110	N	A	A	0	0
111	P	A	A	0	0

Table 13.4: Overdesigned state table for a disjoint window Mealy machine.

It is pretty clear from visually inspecting this state table that many states are equivalent. For example, H, I, K, M, N, and P all have A as the next state for both values of X, and the same output for both values of X. Therefore, they are equivalent. It is also clear that J and L are equivalent. This newly reduced state table is shown in table 13.5.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	H	0	0
E	J	H	0	0
F	J	H	0	0
G	H	H	0	0
H	A	A	0	0
J	A	A	0	1

Table 13.5: Overdesigned state table after a first round of visual state reduction.

Now that the first round of visual inspection has been completed, it is clear that $D \equiv G$ and $E \equiv F$. The final, reduced, state table is shown in table 13.6.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	B	C	0	0
B	D	E	0	0
C	E	D	0	0
D	H	H	0	0
E	J	H	0	0
H	A	A	0	0
J	A	A	0	1

Table 13.6: Final version of the overdesigned state table.

13.7.2 Implication Tables

The visual reduction method is recommended as the first step in reducing state tables, because it is simple and effective. However, it does not always find equivalent states. After completing a visual inspection of equivalent states, an implication table should be used to confirm that there are no further equivalent states.

The state table shown in tab 13.7 does not have any obviously identical states. This doesn't mean that there aren't any equivalent states, however!

Current State	Next State		Output
	X=0	X=1	
A	D	C	0
B	F	H	0
C	E	D	1
D	A	E	0
E	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1

Table 13.7: State table with no obvious visual matches for equivalent states.

An implication table is used to compare each state with every other state. The steps for creating and using an implication table are given below.

1. Construct a chart that contains a square for each pair (i, j) of states.
2. Compare pairs of rows on the state table.
   - If Z is different for (i, j), place an $\times$ in the corresponding box to indicate that $i \neq j$.
   - If Z is the same, place the state names in the square.
   - If the next states and outputs are the same, place a checkmark to indicate that $i \equiv j$.
3. Go through square by square. If $i - j$ is an $\times$, place an $\times$ in that square.
4. If you made any $\times$'s in step 3, repeat until no more $\times$ are placed.
5. For each square that doesn't have an $\times$, $i \equiv j$.

The implication table corresponding to the state table in table 13.7 is shown in table 13.8.

	A	B	C	D	E	F	G
B	D-F C-H						
C	×	×					
D	C-E	F-A H-E	×				
E	×	×	D-A	×			
F	×	×	E-F D-B	×	C-F A-B		
G	D-B C-H	F-B	×	A-B E-H	×	×	
H	×	×	E-C D-G	×	A-G	F-C B-G	×

Table 13.8: Implication table for the state table given in table 13.7.

The implication table, after this process has been carried out, is shown in table 13.9. The red ×'s correspond to the first pass of step 3. The blue ×'s correspond to the second pass of step 3.

	A	B	C	D	E	F	G
B	D-F C-H						
C	×	×					
D	C-E	F-A H-E	×				
E	×	×	D-A	×			
F	×	×	E-F D-B	×	C-F A-B		
G	D-B C-H	F-B	×	A-B E-H	×	×	
H	×	×	E-C D-G	×	A-G	F-C B-G	×

Table 13.9: Completed implication table for the state table given in table 13.7.

From the completed implication table, it can be concluded that $A \equiv D$ and $C \equiv E$, by looking at the boxes that do not contain an ×. The reduced state table is shown in table 13.10.

Current State	Next State X=0	Next State X=1	Output
A	A	C	0
B	F	H	0
C	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1

Table 13.10: Reduced state table after using an implication table.

13.8 State Machine Equivalence

Just as individual states can be deemed equivalent by using an implication table, entire state machines can be checked for equivalence by using an implication table. Two sequential circuits are equivalent if for every state S_i on one circuit, there is an equivalent state T_j in the other.

Let us consider the two finite-state machines as described in the state tables in table 13.11.

Current State	Next State X=0	Next State X=1	Output X=0	Output X=1
A	B	A	0	0
B	C	D	0	1
C	A	C	0	1
D	C	B	0	0

Current State	Next State X=0	Next State X=1	Output X=0	Output X=1
S0	S3	S1	0	1
S1	S3	S0	0	0
S2	S0	S2	0	0
S3	S2	S3	0	1

Table 13.11: Two finite-state machine state tables.

An implication table is made in much the same manner as with the types used to determine state equivalence; the major difference is that the implication table is square shaped to compare every state in the first state machine with every state in the second state machine. When determining state equivalence, the implication table is not square shaped because each state does not have to be compared with itself.

The implication table for the finite-state machines defined in table 13.11 are shown in table 13.12.

	S0	S1	S2	S3
A	×	B-S3 A-S0	B-S0 A-S2	×
B	C-S3 D-S1	×	×	C-S2 D-S3
C	A-S3 C-S1	×	×	A-S2 C-S3
D	×	C-S3 B-S0	C-S0 B-S2	×

Table 13.12: Implication table comparing the state tables shown in table 13.11.

Just as before, every square will be compared, and an × placed if the states inside the squares cannot be equivalent. After going through one round of comparisons, the implication table is shown in table 13.13. Because for each state in circuit one there is an equivalent state in circuit two, we can conclude that the two circuits are equivalent. $B \equiv S_0$, $D \equiv S_1$, $A \equiv S_2$, and $C \equiv S_3$.

	S0	S1	S2	S3
A	×	~~B-S3 A-S0~~	B-S0 A-S2	×
B	C-S3 D-S1	×	×	~~C-S2 D-S3~~
C	~~A-S3 C-S1~~	×	×	A-S2 C-S3
D	×	C-S3 B-S0	~~C-S0 B-S2~~	×

Table 13.13: Completed implication table comparing the state tables shown in table 13.11.

13.9 State Assignment

Up until right now, our state assignments have been mostly arbitrary (except for the reset state which is given all zeros). When there are only a few states, finding the state assignments that lead to the lowest cost (fewest gates and inputs) can be a simple trial-and-error process. However, the number of distinct state assignments increases non-linearly with the number of states in a finite-state machine. The number of flip-flops required to implement a circuit based on the number of states, and the number of distinct state

assignments that can be made, are shown in table 13.14.

States	Flip-Flops	Distinct Assignments
2	1	1
3	2	3
4	2	3
5	3	140
6	3	420
7	3	840
8	3	840
9	4	10,810,800
...	...	...
16	4	$\approx 5.5 \times 10^{10}$

Table 13.14: Number of flip-flops required, and number of distinct state assignments possible, given the number of states in a finite-state machine.

The first step in ensuring the most optimized expressions is to make sure that the state table is fully reduced (in other words: there are no redundant states) by using an implication table.

Once there are at least five states, it is not possible to use a trial-and-error approach to finding the assignments that lead to the most minimized Boolean expressions for the flip-flops and output(s). The following guidelines for state assignment can be used to help lead to optimized state assignments. Note that these guidelines only apply when using D flip-flops.

1. States that have the same next state for a given input (columns) should be adjacent.
2. States that are the next states of the same state (rows) should be adjacent.
3. States that have the same output for a given input (columns) should be adjacent.

Assignments are said to be adjacent if they differ by only one variable. We saw earlier in this book how this applied to adjacent squares in a k-map. Therefore k-maps can be used to place each state in the assignment process.

The state table shown in table 13.15 cannot be reduced with an implication table. Because it has seven states, it would take 840 trial and error takes to find the best state assignments leading to the most minimized expressions.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	G	B	0	1
B	C	E	0	0
C	A	D	1	0
D	G	B	1	1
E	B	F	0	0
F	C	A	0	1
G	A	E	1	1

Table 13.15: Example state table for a Mealy machine.

If a straight binary assignment (i.e. A=000, B=001, etc.) is used to assign states, the expressions would use ninteen logic gates and sixty inputs. The transition table is shown in table 13.16.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
000	110	001	0	1
001	010	100	0	0
010	000	011	1	0
011	110	001	1	1
100	001	101	0	0
101	010	000	0	1
110	000	100	1	1
111	× × ×	× × ×	×	×

Table 13.16: Transition table (using a straight binary assignment) for the state table shown in table 13.15.

The corresponding expressions for the flip-flop inputs and the output are shown below.

$$A^+ = X'A'B'C' + XA'B'C + X'BC + XAC'$$
$$B^+ = X'A'B' + XA'BC' + X'C$$
$$C^+ = AB'C' + XA'C' + XA'B$$
$$Z = XA'B'C' + X'B + BC + XAC + AB$$

Instead, we will use the guidelines for state assignment to try to obtain a better optimization of expressions. Guideline one means that we must examine the next state columns. Because states A and D both have G as the next state, A and D should be adjacent. Because B and F both have C as the next state, B and F should be adjacent, and so on. All of the states that should be adjacent based on guideline one are listed below.

- A, D ($\times$ 2)
- C, G
- B, F

- B, G

Guideline two means that we must examine the next states for each row in a state table. Because G and B are the next states of A, G and B should be adjacent. Because C and E are the next states of B, C and E should be adjacent. All of the guideline two states are shown below.

- B, G ($\times$ 2)
- C, E
- A, D
- B, F
- A, C
- A, E

Guideline three means that we must examine the output columns. All states that share a given output value for an input value should be adjacent, as shown below.

- A, B, E, F
- C, D, G
- A, D, F, G
- B, C, E

It's not possible to satisfy all of the recommendations from the guidelines. Guideline one has the most importance, followed by guideline two and guideline three. Any states that show up multiple times to be adjacent in any guideline should take the most weight.

A more optimized set of state assignments than the straight binary assignment are given below.

- $A = 000$
- $B = 111$
- $C = 001$
- $D = 100$
- $E = 010$
- $F = 011$
- $G = 101$

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
000	101	111	0	1
001	000	100	1	0
010	111	011	0	0
011	001	000	0	1
100	101	111	1	1
101	000	010	1	1
110	× × ×	× × ×	×	×
111	001	010	0	0

Table 13.17: More optimized transition table for the state table given in table 13.15.

The updated transition table is shown in table 13.17. This selection of state assignments leads to the use of fifteen logic gates and thirty-nine inputs, which is a big improvement over the binary assignment.

The expressions for the flip-flops and output are shown below.

$$A^+ = XA'B' + X'C' + AC'$$
$$B^+ = XC' + XA + BC'$$
$$C^+ = C' + X'B$$
$$Z = XB'C' + X'B'C + XA'BC + AB'$$

13.9.1 One-Hot State Assignment

The previous part of this section emphasized finding state assignments that lead to a minimized implementation of a finite-state machine (using as few flip-flops as possible, with expressions using a minimum number of logic gates and inputs). Programmable logic devices such as FPGAs and PAL/GALs contain a flip-flop on every output. It doesn't necessarily make sense to reduce the number of flip-flops when using those types of devices.

One-hot state assignment is a beneficial type of assignment to use when the number of flip-flops does not need to be minimized. (Note that it is still beneficial to reduce the state table to a minimum number of states!) There will be a flip-flop for every state; one flip-flop will be "hot" (have an output of one) at a time. An example state table is shown in table 13.18. Note that there are four states, so there will be four flip-flops.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
S_0	S_0	S_1	0	0
S_1	S_2	S_0	0	1
S_2	S_2	S_3	1	0
S_3	S_1	S_0	0	1

Table 13.18: Example state table for one-hot state assignment.

The state assignments will be

$$S_0(A, B, C, D) = 1000,$$
$$S_1(A, B, C, D) = 0100,$$
$$S_2(A, B, C, D) = 0010,$$
$$S_3(A, B, C, D) = 0001.$$

Because only one flip-flop will have a value of one at a time, the state equations (using D flip-flops) are simple to derive.

Flip-flop A (which is active when the circuit is in state S_0) needs to be activated when $X = 0$ and the circuit is in S_0; when $X = 1$ and the circuit is in S_1; or when $X = 1$ and the circuit is in S_3. This leads to the following expression for A^+.

$$A^+ = X'A + XB + XD$$

Note that each term corresponds to the flip-flop that is active in the current state as well as the value of X to get to the next state. Because only one flip-flop is active at a time, all of the other flip-flop values can be ignored.

Flip-flop B (which is active when the circuit is in state S_1) needs to be activated when $X = 1$ and the circuit is in state S_0; or when $X = 0$ and the circuit is in state S_3. This leads to the following expression for B^+.

$$B^+ = X'D + XA$$

Flip-flop C (which is active when the circuit is in state S_2) needs to be activated when $X = 0$ and the circuit is in state S_1; or when $X = 0$ and the circuit is in state S_2. This leads to the following expression for C^+.

$$C^+ = X'B + X'C$$

Flip-flop D (which is active when the circuit is in state S_3) needs to be activated when $X = 1$ and the circuit is in state S_2. This leads to the following expression for D^+.

$$D^+ = XC$$

Finally, the output expression can be derived for Z. The output is one when $X = 1$ and the circuit is in state S_1; when $X = 0$ and the circuit is in state S_2; or when $X = 1$ and the circuit is in state S_3. This leads

to the following expression for Z.

$$Z = XB + X'C + XD$$

There may be an issue with one-hot state assignment. As has been discussed in this textbook, it is typical to define a reset state that has a state assignment of all zeros. None of the states in the above example would therefore be a good candidate for a reset state. If preset pins are available on the flip-flops, then this challenge can be overcome by presetting the flip-flop corresponding to the reset state (in other words, the reset state will be, for example, 1000, and can be entered into by presetting flip-flop A after powering up the circuit). If there are no preset pins available on the flip-flops, then the expressions need to be modified.

Using the above example, let's say that state S_0 (corresponding to flip-flop A) is the reset state. To obtain expressions that lead to a state assignment of all zeros for state S_0, every term that has an A in it needs to be complemented. This would lead to the following expressions. The state assignments would become $S_0(A, B, C, D)$: 0000, $S_1(A, B, C, D)$: 1100, $S_2(A, B, C, D)$: 1010, and $S_3(A, B, C, D)$: 1001.

$$\begin{aligned} A^+ &= X'A' + XB + XD \\ B^+ &= X'D + XA' \\ C^+ &= X'B + X'C \\ D^+ &= XC \\ Z &= XB + X'C + XD \end{aligned}$$

13.10 Example Problems

Moore Machines

1. Derive a state table for the state diagram shown in Figure 13.20.

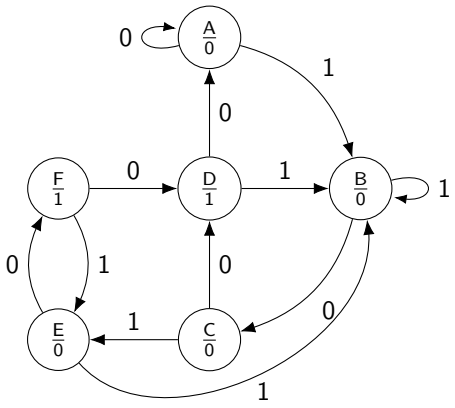


Figure 13.20: State diagram corresponding to Moore machine question 1.

2. Given the state table shown in table 13.19 and the following state assignments, derive minimum SOP expressions for each flip-flop and output using D flip-flops. $A = 000$, $B = 001$, $C = 101$, $D = 010$, $E = 011$, $F = 100$.

Current State	Next State X=0	Next State X=1	Output
A	A	B	0
B	C	B	0
C	D	E	0
D	A	B	1
E	F	B	0
F	D	E	1

Table 13.19: State table corresponding to Moore machine question 2.

3. Given the circuit diagram in Figure 13.21, and the state assignments ($A = 00$, $B = 01$, $C = 10$, $D = 11$) derive a state diagram for the finite-state machine.

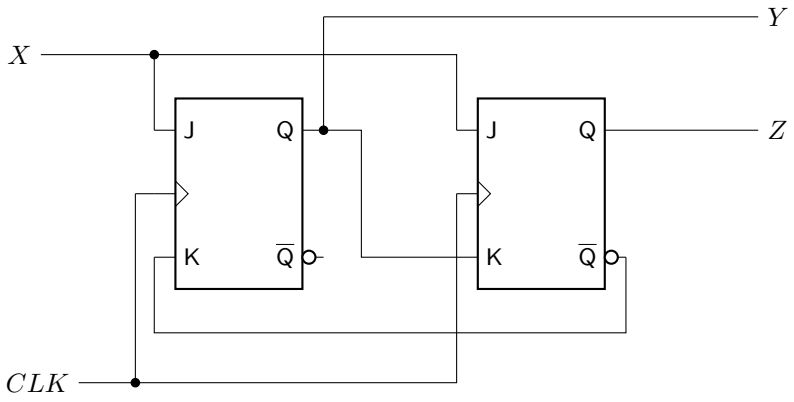


Figure 13.21: Moore machine finite-state machine circuit diagram.

4. Given the state table shown in table 13.20 and the state assignments ($A = 000$, $B = 001$, $C = 010$, $D = 100$), derive minimum SOP expressions for each flip-flop and output using D flip-flops.

Current State	Next State X=0	Next State X=1	Output
A	B	C	0
B	D	A	1
C	A	D	0
D	D	B	0

Table 13.20: State table corresponding to Moore machine question 4.

5. Design a Moore machine that either counts up if $X = 0$ or counts out of order (0–4–5–1–2–6–7–3–0...) if $X = 1$. The state assignment corresponds to the binary number of the state. The output is an even parity bit. Use D flip-flops.

Mealy Machines

1. Derive a state table for the state diagram shown in Figure 13.22.

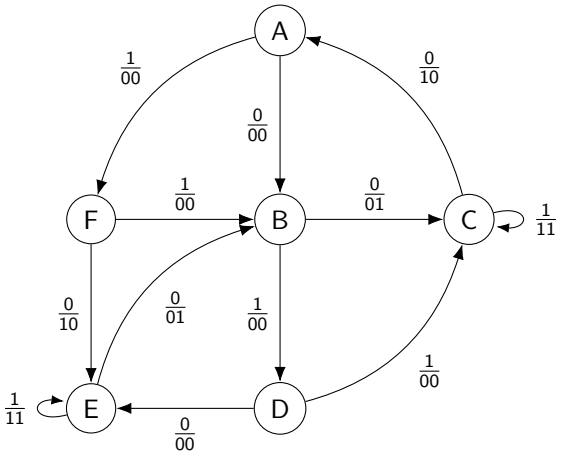


Figure 13.22: State diagram corresponding to Mealy machine question 1.

2. Derive the state table for a Mealy machine that generates an even parity bit for the three preceding bits of input. Define all of the states.

3. Given the circuit diagram in Figure 13.23 and the state assignments ($A = 00$, $B = 01$, $C = 10$, $D = 11$) derive a state table for the finite-state machine.

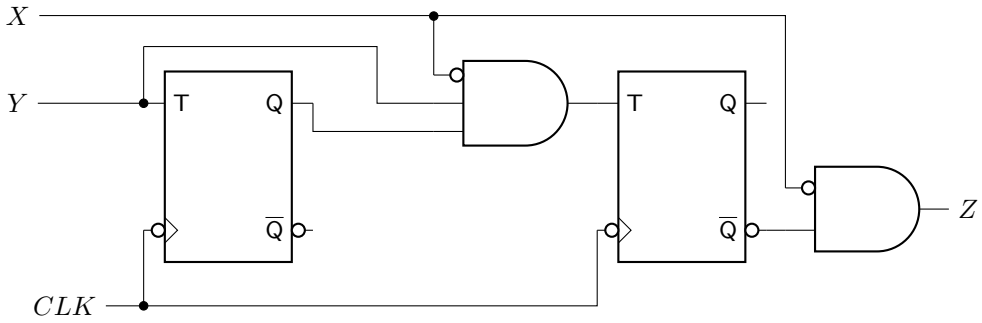


Figure 13.23: Mealy machine finite-state machine circuit diagram.

4. Given the state table in table 13.21 and the state assignments ($A = 000$, $B = 001$, $C = 011$, $D = 010$, $E = 100$, $F = 101$), derive expressions for the inputs to each JK flip-flop and the output Z.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	D	C	0	0
B	E	A	0	1
C	F	B	1	0
D	A	F	1	1
E	C	E	0	0
F	B	D	0	1

Table 13.21: State table corresponding to Mealy machine question 4.

5. Given the state diagram in Figure 13.24 and the state assignments ($A = 000$, $B = 001$, $C = 010$, $D = 100$, $E = 111$) derive expressions for the inputs to each D flip-flop and the output Q.

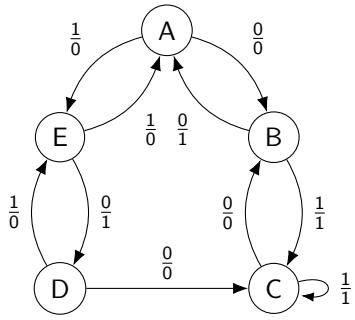


Figure 13.24: State diagram corresponding to Mealy machine question 5.

Sequence Detectors

1. Derive a state diagram for a disjoint window Mealy machine that detects the sequence 1101.

2. Derive a state table for a non-overlapping sliding window Moore machine that detects the sequence 101101. Define each state that is required to implement the state machine.

3. Derive a state table for an overlapping sliding window Mealy machine that detects the sequence 10101. Define each state that is required to implement the state machine.

4. Derive a state diagram for a non-overlapping Moore machine that has an output of one when two consecutive ones are detected. Define each state that is required to implement the state machine.

5. Derive a state table for a Mealy machine that has an output of one when the sequence 01 is detected. After 01 has been detected, 00 must be detected to enable another detection of 01. (The 00 can overlap with the 01.) Define each state that is required to implement the state machine.

State Machine Completeness

1. Consider the state diagram shown in Figure 13.25. Determine if any states have too many transitions, too few transitions, both problems, or no problems.

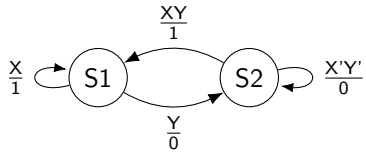


Figure 13.25: State diagram corresponding to state machine completeness question 1.

2. Consider the state diagram shown in Figure 13.26. Determine if any states have too many transitions, too few transitions, both problems, or no problems.

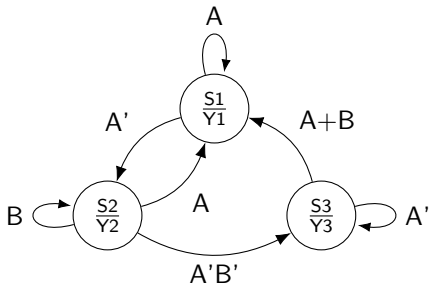


Figure 13.26: State diagram corresponding to state machine completeness question 2.

3. Consider the state diagram shown in Figure 13.27. Determine if any states have too many transitions, too few transitions, both problems, or no problems.

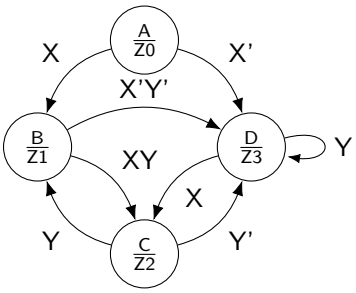


Figure 13.27: State diagram corresponding to state machine completeness question 3.

4. Consider the state diagram shown in Figure 13.28. Determine if any states have too many transitions, too few transitions, both problems, or no problems.

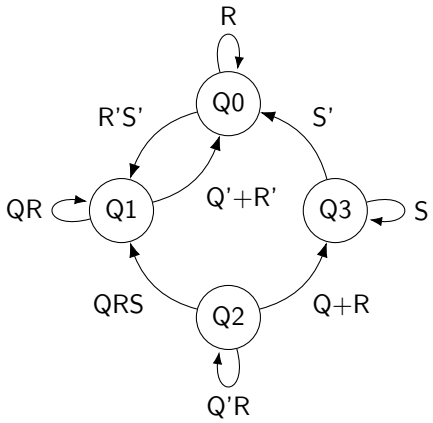


Figure 13.28: State diagram corresponding to state machine completeness question 4.

5. Consider the state diagram shown in Figure 13.29. Determine if any states have too many transitions, too few transitions, both problems, or no problems.

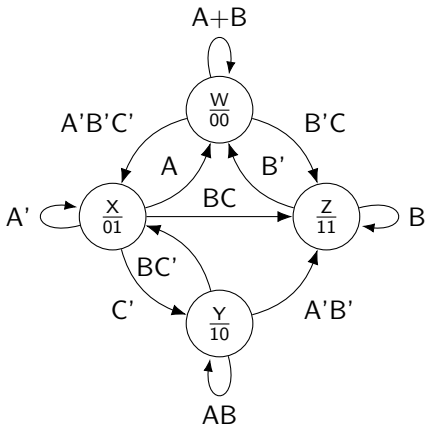


Figure 13.29: State diagram corresponding to state machine completeness question 5.

State Table Reduction

1. Reduce the state machine in Figure 13.30 to a minimum number of states.

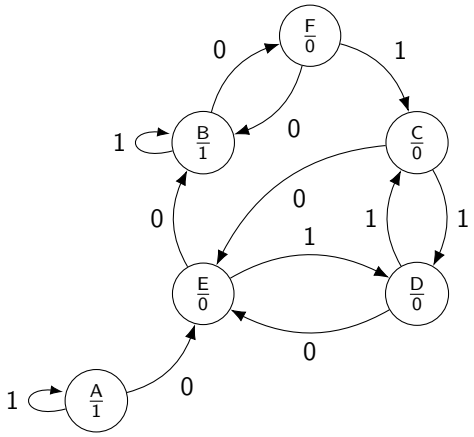


Figure 13.30: State diagram corresponding to state table reduction question 1.

2. Reduce the state table in table 13.22 to a minimum number of states.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	F	B	0	0
B	E	A	0	1
C	H	G	0	1
D	H	D	1	0
E	B	F	1	1
F	G	B	0	0
G	A	C	0	0
H	C	A	1	1

Table 13.22: State table corresponding to state table reduction question 2.

3. Reduce the state table in table 13.23 to a minimum number of states.

Current State	Next State		Output
	X=0	X=1	
a	e	e	1
b	c	e	1
c	i	h	0
d	h	a	1
e	i	f	0
f	e	g	0
g	h	b	1
h	c	d	0
i	f	b	1
j	f	a	0

Table 13.23: State table corresponding to state table reduction question 3.

4. Reduce the state table in table 13.24 to a minimum number of states.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	B	C	0	1
B	C	A	0	1
C	D	B	1	0
D	C	A	0	1

Table 13.24: State table corresponding to state table reduction question 4.

5. Reduce the state table in table 13.25 to a minimum number of states.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	E	D	0	0
B	A	F	1	0
C	C	A	0	1
D	B	A	0	0
E	D	C	1	0
F	C	D	0	1
G	H	G	1	1
H	C	B	1	1

Table 13.25: State table corresponding to state table reduction question 5.

State Machine Equivalence

1. Using the state tables given in table 13.26, determine if the two state machines are equivalent.

Current State	Next State X=0	Next State X=1	Output
S0	S3	S1	0
S1	S0	S1	0
S2	S0	S2	1
S3	S0	S3	1

Current State	Next State X=0	Next State X=1	Output
A	E	A	1
B	F	B	1
C	E	D	0
D	E	C	0
E	B	D	0
F	B	C	0

Table 13.26: State tables for state machine equivalence question 1.

2. Using the state tables given in table 13.27, determine if the two state machines are equivalent.

Current State	Next State X=0	Next State X=1	Outputs
A	B	D	00
B	D	B	01
C	C	B	11
D	A	C	10

Current State	Next State X=0	Next State X=1	Outputs
S0	S2	S4	00
S1	S1	S2	11
S2	S4	S5	01
S3	S5	S4	00
S4	S3	S1	10
S5	S4	S2	01

Table 13.27: State tables for state machine equivalence question 2.

3. Using the state tables given in table 13.28, determine if the two state machines are equivalent.

Current State	Next State X=0	Next State X=1	Output
A	A	B	0
B	C	D	0
C	B	C	1
D	D	A	0

Current State	Next State X=0	Next State X=1	Output X=0	Output X=1
S0	S0	S1	0	0
S1	S2	S3	0	0
S2	S1	S2	1	1
S3	S3	S0	0	0

Table 13.28: State tables for state machine equivalence question 3.

4. Using the transition tables given in table 13.29, determine if the two state machines are equivalent.

Current State	Next State X=0	Next State X=1	Outputs
000	001	011	00
001	010	011	01
010	000	111	10
011	111	000	01
100	× × ×	× × ×	××
101	× × ×	× × ×	××
110	× × ×	× × ×	××
111	011	001	11

Current State	Next State X=0	Next State X=1	Outputs
000	101	011	00
001	× × ×	× × ×	××
010	× × ×	× × ×	××
011	111	000	01
100	000	111	10
101	100	011	11
110	× × ×	× × ×	××
111	011	101	00

Table 13.29: Transition tables for state machine equivalence question 4.

5. Using the circuit diagrams given in Figure 13.31, determine if the two state machines are equivalent.

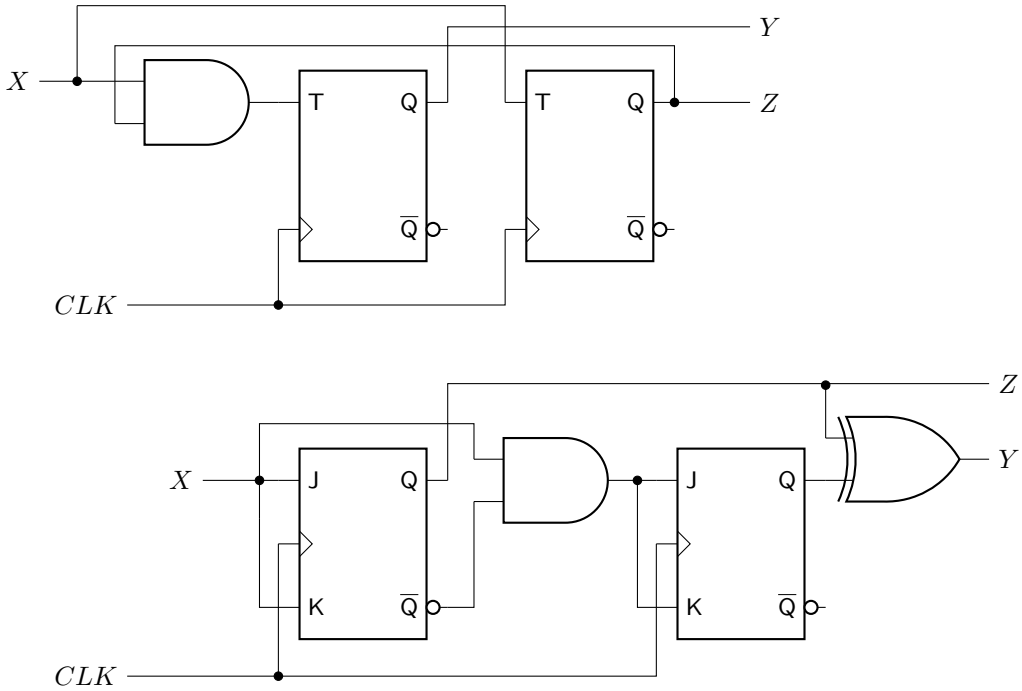


Figure 13.31: Circuit diagrams for state machine equivalence question 5.

14 Solutions to Example Problems

14.1 Chapter 1

Number System Conversions

1. $1101100.\overline{0011}$
2. 10001000
3. 11011.110
4. 223300010330
5. $63.\overline{51}$

Binary Addition

1. 01011
2. 11000
3. 00010
4. Overflow: adding two positive numbers yielded a negative result
5. 11001

Binary Subtraction

1. Overflow: negative minus positive should be negative, but the solution is positive
2. 0001
3. 101111
4. 0111
5. Overflow: positive minus negative should be positive, but the solution is negative

Binary Multiplication

1. Overflow: overflow bits indicate a negative solution, but the sign bit of the result indicates a positive answer
2. 1111 1000
3. 11111 10111

4. Overflow: overflow bits indicate a negative solution, but the sign bit of the result indicates a positive answer

5. 111111 101110

Binary Codes

1. 0011 0101 0010

2. 0100 0111 0011

3. 1110001

4. 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

5. 2-4-2-1 is a complete binary code. The numeral encodings for 0–9 in order are: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1110, 1111

14.2 Chapter 2

Circuit Diagrams, Truth Tables, and Boolean Expressions

1. The circuit diagram for question 1 is shown in Figure 14.1.

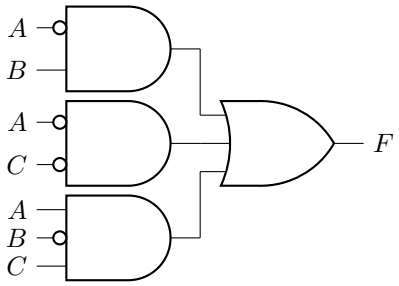


Figure 14.1: Circuit diagram corresponding to question 1.

2. F and G are equivalent. You can use a truth table to show that they are equivalent, or you can reduce both expressions to minimum SOP and verify that they are identical when reduced.

3. F and G are equivalent. You can use a truth table to show that they are equivalent, or you can reduce both expressions to minimum SOP and verify that they are identical when reduced.

4. The answer is shown in table 14.1

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

Table 14.1: Truth table answer for Circuit Diagrams, Truth Tables, and Boolean Expressions question 4.

5. $G = A(BC + B'C')$

Minimum Sum of Products Expressions

1. $F = A'B + A'C' + AB'C$
2. $F = CD' + A'C'D + BC'D$
3. $F = BD + AC'D + A'B'C$
4. $F = A'C + BD + AB'C'D'$
5. $F = A'BC' + BC'D$

Minimum Product of Sums Expressions

1. $F = (B + C)(A' + B')$
2. $F = (A' + B + C + D)(A + B' + C' + D')$
3. $F = (B + C')(A' + B + D')(B' + C + D')$
4. $F = (A' + C)(A + B)$
5. $F = (A + C')(A + B + E')(A' + B')(A' + C + D)$

XOR and XNOR Circuits

1. $F = (A + B' + C)(A' + B + D')(A' + B' + C')$

2. The answer is given in table 14.2

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

Table 14.2: Truth table answer for XOR and XNOR circuits question 2.

3. $F = (A + B)(B' + C)(B' + D)(B + C' + D')$

4. $F = ACD + ABD + A'BD'$

5. $F = (W + X)(W + Z')(W' + Z)(X + Y)(W' + X' + Y')$

14.3 Chapter 3

Design Applications

1. Warning light $W = 0$ (OFF) or 1 (ON). $W = A' + BC$. Compressor $C = 0$ (OFF) or 1 (ON). $C = A' + B + C + DE$.

2. $F_{RED} = BC'D' + AC' + ABD'$. $F_{GREEN} = B'C'D + A'C + AB'D$. $F_{BLUE} = A'B'C'D' + A'BC'D + AB'CD' + ABCD$.

3. $F = X'Y'A + XYB + XYA$.

4. Motor $M = 0$ (OFF) or 1 (ON). Give change $G = 0$ (don't give change) or 1 (give change). Enough money $E = 0$ (not enough money was inserted) or 1 (enough money was inserted). Too much money $T = 0$ (not too much money was inserted) or 1 (too much money was inserted). Selection button $B = 0$ (not pressed) or 1 (pressed). $G = TB$, $M = EB$ or $M = B(T + E)$.

5. This answer names the two 2-bit unsigned binary inputs as AB and CD. The result, which is a 4-bit number $F_3F_2F_1F_0$ (where F_3 is the MSB and F_0 is the LSB) is shown in Figure 14.2.

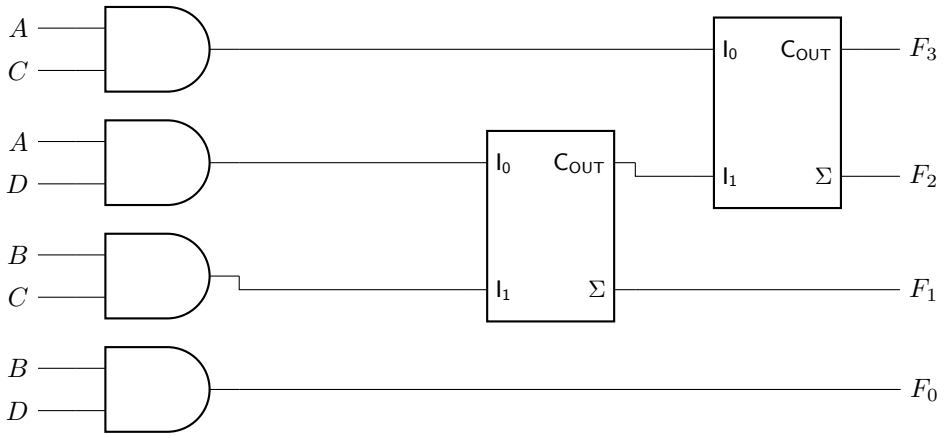


Figure 14.2: Schematic of a 2-bit unsigned binary multiplier using half adders.

Minterm Expressions

1. $F_{SOP}(A,B,C) = A'B' + B'C + ABC'$, $F_{POS}(A,B,C) = (A+B')(B'+C')(A'+B+C)$
2. $F(A,B,C,D) = AD + B'C'D'$. Don't care term d_{11} was used to minimize the expression.
3. $F(A,B,C) = B' + A'C$.
4. $F(A,B,C,D) = A'C'D' + BD$. Don't care terms d_5 and d_{15} were used to minimize the expression.
5. $F(A,B,C,D) = (B'+D)(A+B)(B+C)$. Don't care term D_0 was used to minimize the expression.

Maxterm Expressions

1. $F(A,B,C) = A'B' + AC'$
2. $F(A,B,C) = (A+B')(A'+C')$. Don't care term D_7 was used to minimize the expression.
3. $F(A,B,C,D) = (A+B)(A'+B'+C')$. Don't care terms D_0, D_3, and D_{15} were used to minimize the expression.
4. There are two possible solutions. Solution one is $F(A,B,C,D) = (A'+C)(A+C'+D)(A+B+C')$, which uses don't care term D_{12} to minimize the expression. Solution two is $F(A,B,C,D) = (A'+C)(A+C'+D)(A+B+D')$, which uses don't care terms D_1 and D_{12} to minimize the expression.
5. $F(A,B,C,D) = BD + AB'C$. Don't care terms d_{11} and d_{15} were used to minimize the expression.

Minterm and Maxterm Expansions

1. $F(A, B, C, D) = \Sigma m(1, 3, 5, 7, 11, 15)$
2. $F(A, B, C, D) = \Sigma m(1, 3, 6, 7, 8, 9, 10, 11)$
3. $F(A, B, C, D) = \Pi M(0, 1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 15)$
4. $F(A, B, C, D) = \Sigma m(3, 5, 10, 11, 12, 13)$
5. $F(A, B, C, D) = \Pi M(0, 1, 2, 3, 4, 5, 7, 9, 13)$

14.4 Chapter 4

Three Variable K-Maps

1. $F(A, B, C) = A'C + AC' + A'B'$ (looping m_0 with m_1), or $F(A, B, C) = A'C + AC' + B'C'$ (looping m_0 with m_4). The k-map is shown in Figure 14.3.

	A	
BC	**0**	**1**
00	1	1
01	1	0
11	1	0
10	0	1

Figure 14.3: K-map corresponding to three variable k-map question 1.

2. $F(A, B, C) = A + BC$. The k-map is shown in Figure 14.4.

	A	
BC	**0**	**1**
00	0	1
01	0	1
11	1	1
10	0	1

Figure 14.4: K-map corresponding to three variable k-map question 2.

3. Minterms: m_0, m_2, m_3, m_4, m_5. Groups of 2: $m_0 - m_2$, $m_0 - m_4$, $m_2 - m_3$, $m_4 - m_5$.

4. $F(A, B, C) = \Sigma m(2, 6, 7)$. The k-map is shown in Figure 14.5.

	A	
BC	**0**	**1**
00	0	0
01	0	0
11	0	1
10	1	1

Figure 14.5: K-map corresponding to three variable k-map question 4.

5. The k-maps are shown in Figure 14.6. Note that term $AB'C$ cannot be both a zero and a one, making it a don't care term.

	A	
BC	**0**	**1**
00	0	1
01	1	1
11	1	0
10	0	0

	A	
BC	**0**	**1**
00	0	1
01	1	0
11	1	0
10	0	0

Figure 14.6: K-maps corresponding to three variable k-map question 5. The k-map on the left corresponds to $F = A'C + AB'$, and the k-map on the right corresponds to $G = (A' + C')(B' + C)(A + C)$.

Four Variable K-Maps

1. $F(A, B, C, D) = ABC' + A'C'D + BD + ACD + A'CD'$. The k-map is shown in Figure 14.7.

	AB			
CD	**00**	**01**	**11**	**10**
00	0	0	1	0
01	1	1	1	0
11	0	1	1	1
10	1	1	0	0

Figure 14.7: K-map corresponding to four variable k-maps question 1.

2. $F(A,B,C,D) = B'D' + A'C + B'C$. The k-map is shown in Figure 14.8.

	AB			
CD	**00**	**01**	**11**	**10**
00	×	0	×	1
01	0	0	×	0
11	1	1	0	1
10	1	1	0	×

Figure 14.8: K-map corresponding to four variable k-maps question 2.

3. $F(A,B,C,D) = (A+B'+C'+D')(A'+C+D')(B+D)(A'+B)$. The k-map is shown in Figure 14.9.

	AB			
CD	**00**	**01**	**11**	**10**
00	0	1	1	0
01	1	1	0	0
11	1	0	1	0
10	0	1	1	0

Figure 14.9: K-map corresponding to four variable k-maps question 3.

4. $F_{SOP}(A,B,C,D) = A'C + A'B'D + CD$, $F_{POS}(A,B,C,D) = (C+D)(A'+D)(B'+C)(A'+C)$. The k-map is shown in Figure 14.10.

	AB			
CD	**00**	**01**	**11**	**10**
00	0	0	×	0
01	1	0	×	0
11	1	1	×	1
10	1	1	×	0

Figure 14.10: K-map corresponding to four variable k-maps question 4.

5. $G_{SOP}(A,B,C,D) = A'CD + A'B'C$, $G_{POS} = (A')(C)(B'+D)$. The k-map is shown in Figure 14.11.

	AB			
CD	**00**	**01**	**11**	**10**
00	0	0	×	0
01	0	0	×	0
11	1	1	×	0
10	1	0	×	0

Figure 14.11: K-map corresponding to four variable k-maps question 5.

Prime Implicant Tables

1. $F(A,B,C,D) = CD' + A'BD$. The PI table is given in table 14.3.

PI	2	5	6	7	10	14
A'BD		×		×		
A'BC			×	×		
CD'	×		×		×	×

Table 14.3: PI table for question 1.

2. $F(A,B,C,D) = (B + C + D')(B' + C' + D)(A + B)$. The PI table is given in table 14.4.

PI	0	1	2	3	6	9	14
B+C+D'		×				×	
A+C'+D			×		×		
B'+C'+D					×		×
A+B	×	×	×	×			

Table 14.4: PI table for question 2.

3. $F(A,B,C,D,E) = A'CE + ABDE' + A'BC'D$. The PI table is given in table 14.5.

PI	5	10	11	15	26	30
A'CE	×			×		
A'BC'D		×	×			
BC'DE'		×			×	
A'BDE			×	×		
ABDE'					×	×
BCDE				×		
ABCD						×

Table 14.5: PI table for question 3.

4. The two minimum SOP expressions for $F(A,B,C,D)$ are $B'CD' + A'BC' + AC + BC'D$ and $B'CD' + A'BC' + AC + ABD$. The PI table is given in table 14.6.

PI	2	4	5	10	11	13	14	15
A'C'D			×					
B'CD'	×			×				
A'BC'		×	×					
AB'D'				×				
BC'D			×			×		
ABD						×		×
AC				×	×		×	×

Table 14.6: PI table for question 4.

5. The essential PIs are $(A + B' + C')$ and $(B' + C' + D)$. The PI table is given in table 14.7.

PI	1	2	3	4	5	12	13	15	16	17	20	28	29
A+B+D+E'	×				×								
B+C+E'	×		×							×			
B+C+D'		×	×										
B+D'+E		×											
A+C'+D				×	×	×	×						
A'+B+C									×	×			
A'+B+E									×		×		
A+B'+C'						×	×	×					
B'+C'+D						×	×					×	×
C'+E				×		×					×	×	

Table 14.7: PI table for question 5.

14.5 Chapter 5

1. $F(A, B, C) = A'C + AB'$. The Quine-McCluskey columns are shown in table 14.8.

Column 1		Column 2		Column 3
one	1. 001 ✓	1-3:	0−1	
	4. 100 ✓	1-5:	−01	
two	3. 011 ✓	4-5:	10−	
	5. 101 ✓			

Table 14.8: Quine-McCluskey columns for question 1.

2. $F(A, B, C, D) = B'D' + AB'$. The Quine-McCluskey columns are shown in table 14.9.

Column 1		Column 2		Column 3		Column 4
zero	0. 0000 ✓	0-2:	00−0 ✓	0-2-8-10:	−0−0	
one	2. 0010 ✓	0-8:	−000 ✓	~~0-8-2-10~~		
	8. 1000 ✓	2-10:	−010 ✓	8-9-10-11	10−−	
two	9. 1001 ✓	8-9:	100− ✓	~~8-10-9-11~~		
	10. 1010 ✓	8-10:	10−0 ✓			
three	11. 1011 ✓	9-11:	10−1 ✓			
		10-11:	101− ✓			

Table 14.9: Quine-McCluskey columns for question 2.

3. $F(A,B,C,D,E) = A'B'C'D'E' + AB'CE + A'B'CD$ or $F(A,B,C,D,E) = A'B'C'D'E' + AB'CE + B'CDE$. The Quine-McCluskey columns are shown in table 14.10.

Column 1		Column 2		Column 3	Column 4
zero	0. 00000	6-7:	0011−		
one	–	6-14:	0−110		
two	6. 00110 ✓	7-23:	−0111		
three	7. 00111 ✓	21-23:	101−1		
	14. 01110 ✓				
	21. 10101 ✓				
four	23. 10111 ✓				

Table 14.10: Quine-McCluskey columns for question 3.

4. $F(A,B,C,D,E) = B'C'D' + BDE + A'BE$. The Quine-McCluskey columns are shown in table 14.11.

Column 1		Column 2		Column 3		Column 4
zero	0. 00000 ✓	0-1:	0000− ✓	0-1-16-17:	−000−	
one	1. 00001 ✓	0-16:	−0000 ✓	~~0-16-1-17~~		
	16. 10000 ✓	1-9:	0−001	–		
two	9. 01001 ✓	1-17:	−0001 ✓	9-11-13-15:	01−−1	
	17. 10001 ✓	16-17:	1000− ✓	~~9-13-11-15~~		
three	11. 01011 ✓	9-11:	010−1 ✓	11-15-27-31:	−1−11	
	13. 01101 ✓	9-13:	01−01 ✓	~~11-27-15-31~~		
four	15. 01111 ✓	11-15:	01−11 ✓			
	27. 11011 ✓	11-27:	−1011 ✓			
five	31. 11111 ✓	13-15:	011−1 ✓			
		15-31:	−1111 ✓			
		27-31:	11−11 ✓			

Table 14.11: Quine-McCluskey columns for question 4.

5. $Z(A,B,C,D,E,F,G) = ABC'D'E'F'G' + A'BDEF' + B'C'D'G$. The Quine-McCluskey columns are shown in table 14.12.

Column 1		Column 2		Column 3		Column 4	
one	0. 0000001 ✓	1-3:	00000−1 ✓	1-3-5-7:	0000−−1 ✓	1-3-5-7-	−000−−1
two	3. 0000011 ✓	1-5:	0000−01 ✓	1-3-65-67:	−0000−1 ✓	65-67-69-71	
	5. 0000101 ✓	1-65:	−000001 ✓	~~1-5-3-7~~		~~1-3-65-67-~~	
	65. 1000001 ✓	3-7:	0000−11 ✓	1-5-65-69:	−000−01 ✓	~~5-7-69-71~~	
	96. 1100000	3-67:	−000011 ✓	~~1-65-3-67~~		~~1-5-65-69~~	
three	7. 0000111 ✓	5-7:	00001−1 ✓	~~1-65-5-69~~		~~3-7-67-71~~	
	44. 0101100 ✓	5-69:	−000101 ✓	3-7-67-71:	−000−11 ✓		
	67. 1000011 ✓	65-67:	10000−1 ✓	~~3-67-7-71~~			
	69. 1000101 ✓	65-69:	1000−01 ✓	5-7-69-71:	−0001−1 ✓		
four	45. 0101101 ✓	7-71:	−000111 ✓	~~5-69-7-71~~			
	60. 0111100 ✓	44-45:	010110− ✓	65-67-69-71:	1000−−1 ✓		
	71. 1000111 ✓	44-60:	01−1100 ✓	~~65-69-67-71~~			
five	61. 0111101	67-71:	1000−11 ✓	44-45-60-61:	01−110−		
		69-71:	10001−1 ✓	~~44-60-45-61~~			
		45-61:	01−1101 ✓				
		60-61:	011110− ✓				

Table 14.12: Quine-McCluskey columns for question 5.

14.6 Chapter 6

NAND-Only Circuits

1. $F(A,B,C,D) = [(A'BC')'(BC'D)'(A'D)']'$
2. $F(A,B,C,D) = [(BD')'(A'C'D)'(A'B'D)'(ABC)']'$
3. $F(A,B,C,D) = \{[A'B'(CD')']'[AB(C'D')']'\}'$
4. $F(A,B,C,D,E) = \{A'(B'C)'(B[D'(EC')']')'\}'$
5. $F(A,B,C,D) = \{[(A'B')'(ABD')'(C'D)']'\}'$

NOR-Only Circuits

1. $F(A,B,C) = [(A + B' + C)' + (A' + C')']'$
2. $F(A,B,C,D) = \{[(B' + C')' + A]' + [B + C' + D']'\}'$
3. $F(A,B,C,D,E) = \{[A + (B + C + D)']' + [B + C + (D' + E')']'\}'$
4. $F(A,B) = \{[(A' + B')' + (A + B)']'\}'$
5. $F(A,B,C,D) = \{[(C' + D)' + A + B]' + [(C + D)' + A' + B']'\}'$

14.7 Chapter 7

Least Cost Implementation

1. Factored SOP is the lowest cost with four gates and eight inputs. $F(A,B,C,D) = C(A'+B') + BD'$
2. Factored POS is the lowest cost with five gates and twelve inputs. $F(A,B,C,D) = (C'+D+AB')(C+D'+A'B')$
3. Factored SOP is the lowest cost with five gates and thirteen inputs. $F(A,B,C,D,E) = B'C'(D'+E) + A'BC(D+E)$
4. POS is the lowest cost with four gates and twelve inputs. $F(A,B,C,D) = (A+B+D')(A'+B'+D')(B+C'+D')$
5. SOP is the lowest cost with three gates and seven inputs. $F(A,B,C,D) = A'BD + CD'$

Multiple Output Optimization

1. $X(A,B,C,D) = A'C + BC'D' + AB'CD'$, $Y(A,B,C,D) = BC + BC'D' + AB'CD'$. This implementation saves two gates and four inputs over using a minimum SOP implementation.
2. $X(A,B,C,D) = B'CD' + A'B'D' + A'BCD$, $Y(A,B,C,D) = B'C'D' + A'B'D' + A'BCD$. This implementation saves one gate and two inputs over using a minimum SOP implementation.
3. $X(A,B,C,D) = B'C'D + A'B'D' + ABCD$, $Y(A,B,C,D) = AC'D' + A'B'D' + ABCD$. This implementation is exactly the same as the minimum SOP implementation.
4. $X(A,B,C,D) = B'D + BCD' + A'BC$, $Y(A,B,C,D) = ABD' + A'BC$. This implementation is exactly the same as the minimum SOP implementation.
5. $X(A,B,C,D) = B'C + AC'D + A'BCD$, $Y(A,B,C,D) = C'D + ABC + A'BCD$. This implementation saves one gate and one input over using a minimum SOP implementation.

14.8 Chapter 9

Timing Diagrams

1. The output $Z(A, B, C)$ is shown in Figure 14.12. Note the static 1 hazard from 30–32 ns.

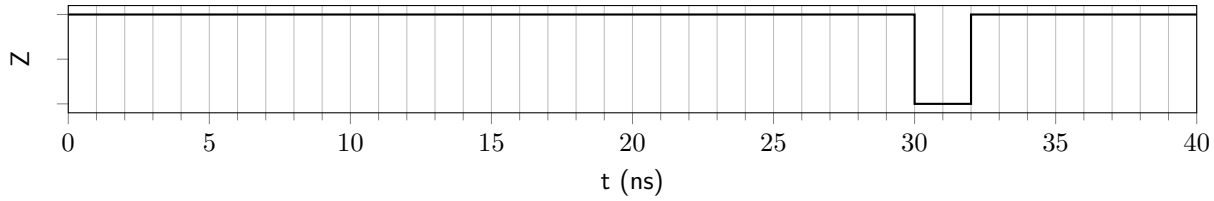


Figure 14.12: Timing diagram for question 1.

2. The output $F(A, B, C)$ is shown in Figure 14.13.

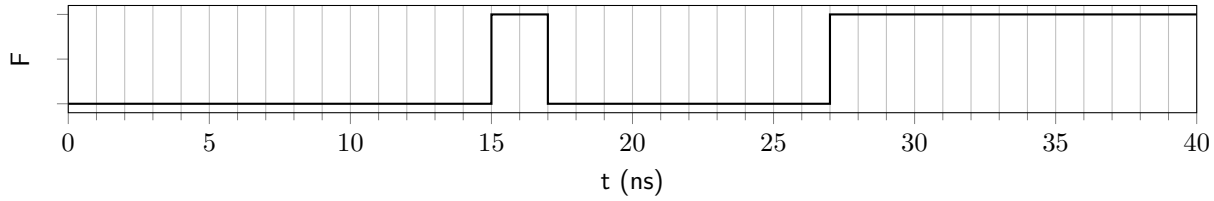


Figure 14.13: Timing diagram for question 2.

3. The output $F(A, B, C, D)$ is shown in Figure 14.14. Note the static 0 hazard from 30–32 ns.

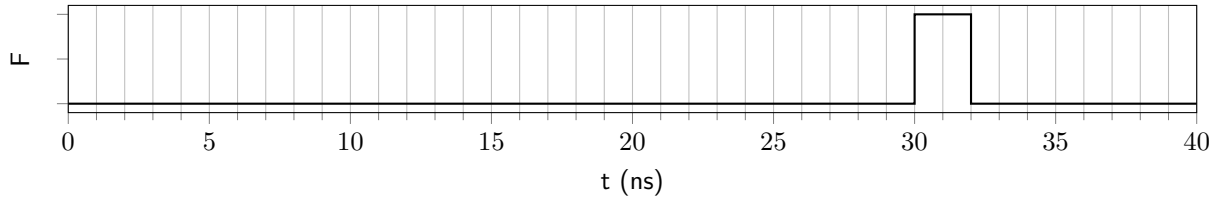


Figure 14.14: Timing diagram for question 3.

4. The output $F(A, B, C, D)$ is shown in Figure 14.15.

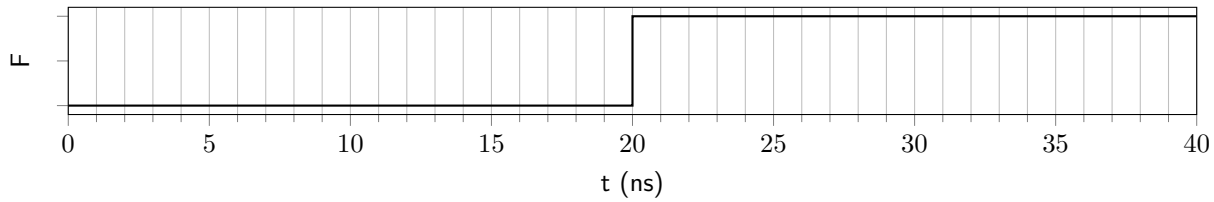


Figure 14.15: Timing diagram for question 4.

5. The output $1Y1$ is shown in Figure 14.16.

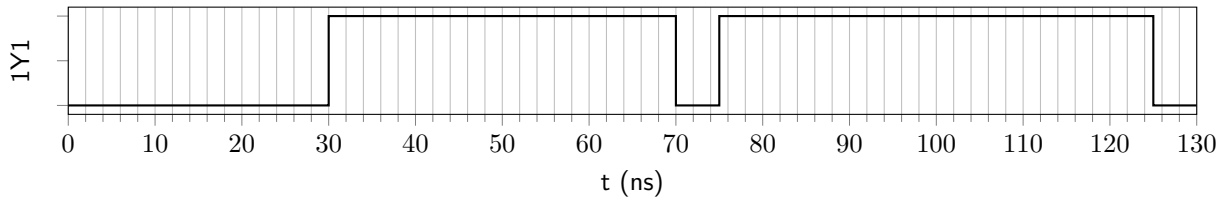


Figure 14.16: Timing diagram for question 5.

Hazards

1. $F(A, B, C) = AC + BC + A'B$
2. $F(A, B, C, D, E) = A'BCE' + BCD'E' + ABD'E' + ABCD' + B'CE + CDE + ACE$
3. $F(A, B, C, D) = [(A'B'C')'(A'B'D)'(A'BD')'(B'CD)'(C'D')']'$
4. $F(A, B, C, D) = [(B' + C' + D)' + (A' + B + C')' + (A' + C' + D)' + (A + D')' + (B + D')' + (C + D')' + (A + B')']'$
5. $F(A, B, C, D) = [(A + C)' + (C + D')' + (A' + D')' + (A' + C')' + B']'$

14.9 Chapter 10

Multiplexers

1. The schematic is shown in Figure 14.17.

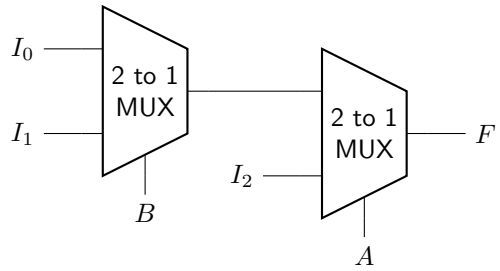


Figure 14.17: Circuit schematic corresponding to multiplexers question 1.

2. $F(A, B, C, D) = A'D'(B') + A'D(C) + AD'(1) + AD(C')$
3. $F(A, B, C, D) = B'D'(1) + B'D(A') + BD'(A \oplus C) + BD(0)$

4. The timing diagram for F is shown in Figure 14.18.

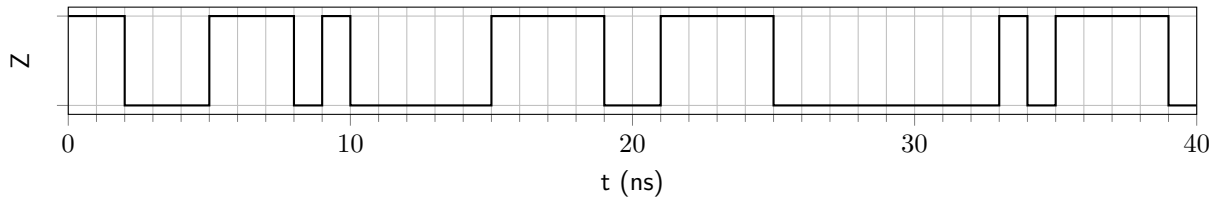


Figure 14.18: Timing diagram for question 4.

5. $F(A, B, C, D) = A'C'(D' + B) + A'C(0) + AC'(1) + AC(D' + B)$

Demultiplexers

1. The output timing diagrams are shown in Figure 14.19.

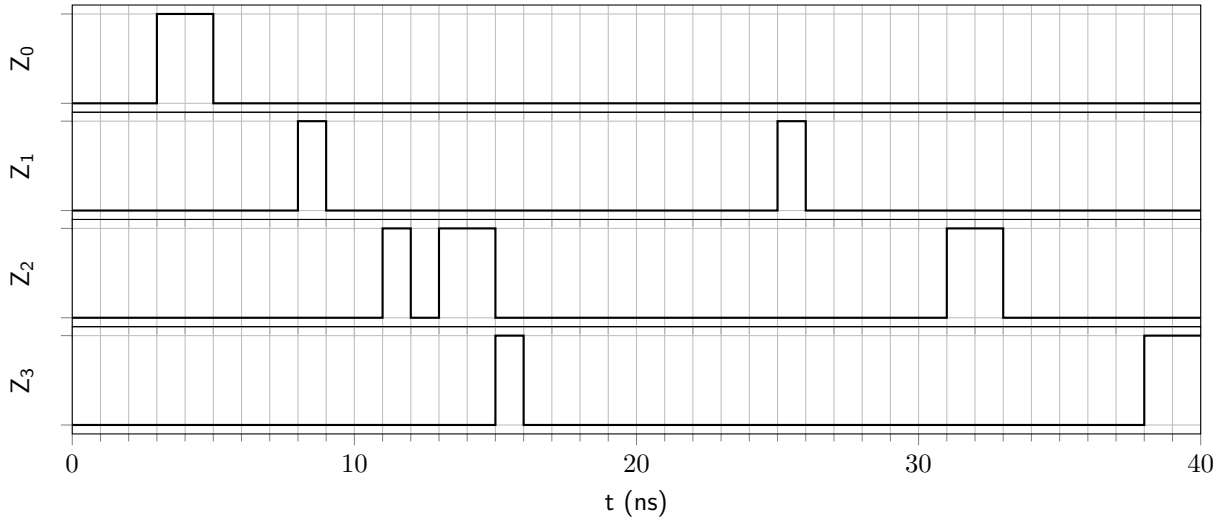


Figure 14.19: Timing diagram corresponding to demultiplexers question 1.

2. The schematic is shown in Figure 14.20.

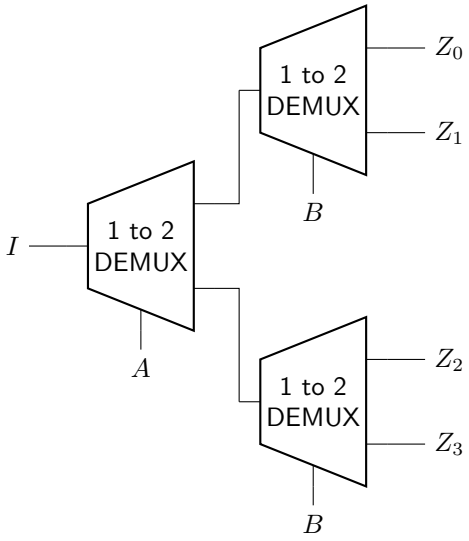


Figure 14.20: Circuit schematic corresponding to demultiplexers question 2.

3. Five control bits would be required.

4. $Z_0 = A'B'C'I$, $Z_1 = A'B'CI$, $Z_2 = A'BC'I$, $Z_3 = A'BCI$, $Z_4 = AB'C'I$, and $Z_5 = AB'CI$

5. The schematic is shown in Figure 14.21.

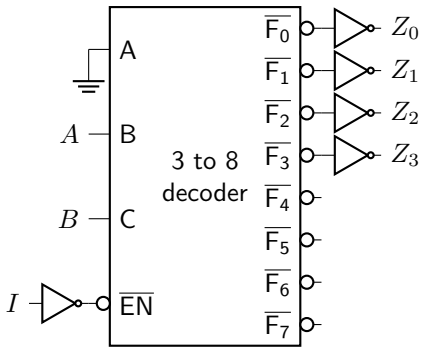


Figure 14.21: Circuit diagram corresponding DEMUX question 5.

Decoders

1. There are an equal number of minterms and maxterms, so either implementation would work. The circuit diagram is shown in Figure 14.22.

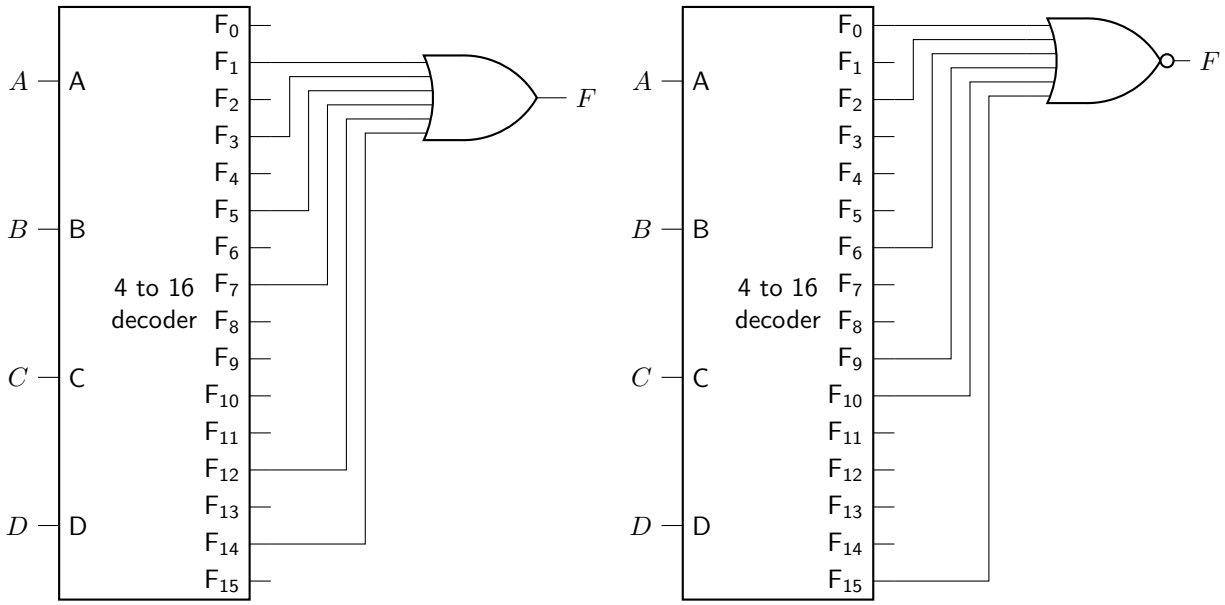


Figure 14.22: Circuit diagrams corresponding to decoders answer 1.

2. There are fewer minterms, so that will be the most efficient implementation of F. The circuit diagram is shown in Figure 14.23.

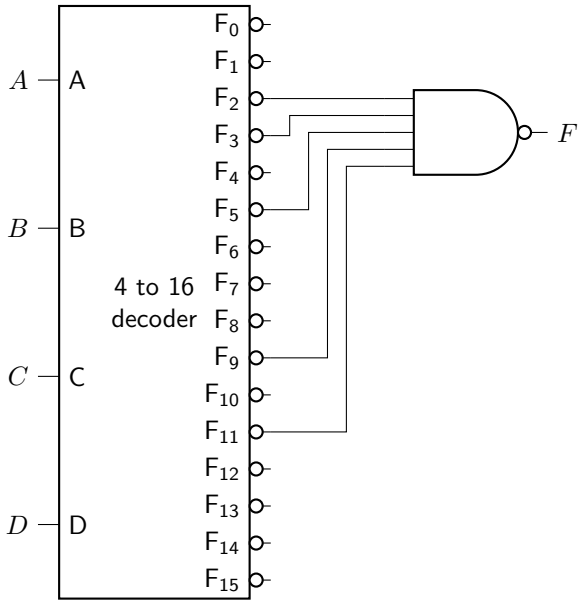


Figure 14.23: Circuit diagram corresponding to decoders answer 2.

3. The correctly labeled schematic of the 3 to 8 decoder is shown in Figure 14.24.

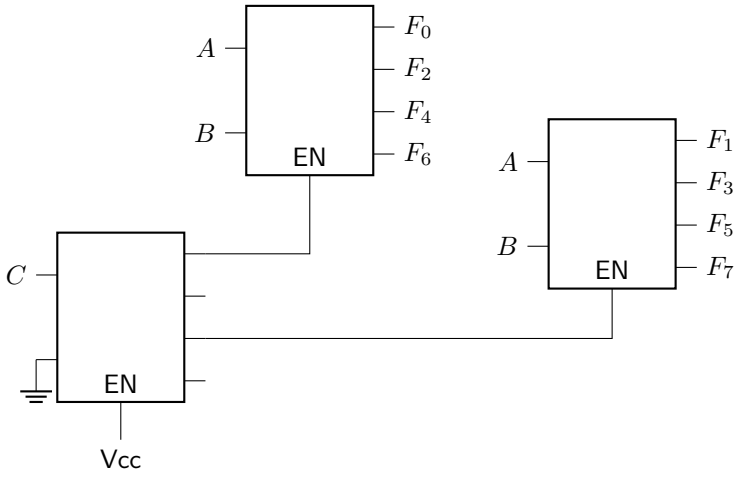


Figure 14.24: Decoder schematic corresponding to question 3.

4. Because there are fewer minterms, that leads to the minimum implementation. The corresponding circuit diagram is shown in Figure 14.25.

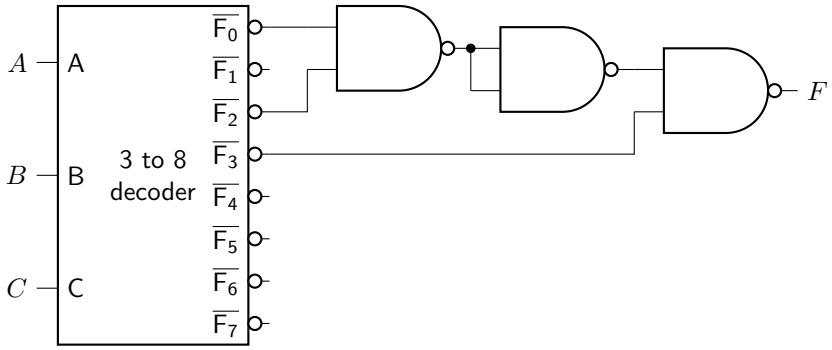


Figure 14.25: Circuit diagram corresponding to decoders answer 4.

5. The segment output expressions in minimum SOP form are $F_a = B + AC + A'C'$, $F_b = A' + C'$, $F_c = B' + C$, $F_d = B + AC + A'C'$, $F_e = A'C'$, $F_f = A + B'C'$, and $F_g = A + B$.

Encoders

1. 011
2. $A = I_2 + I_3$, $B = I_3 + I_1 I_2'$, $DATA = I_0 + I_1 + I_2 + I_3$
3. $A = I_0' I_1' I_3 + I_0' I_1' I_2$, $B = I_0' I_1 + I_0' I_2' I_3$, $DATA = I_0 + I_1 + I_2 + I_3$
4. $A_2 A_1 A_0 = 010$
5. $A_3 A_2 A_1 A_0 = 1001$

Memory

1. $2^{18} = 262{,}144$ bytes
2. $2 \times 2^{13} = 16{,}384$ bytes
3. Sixteen address bits are required

4. The schematic is shown in Figure 14.26.

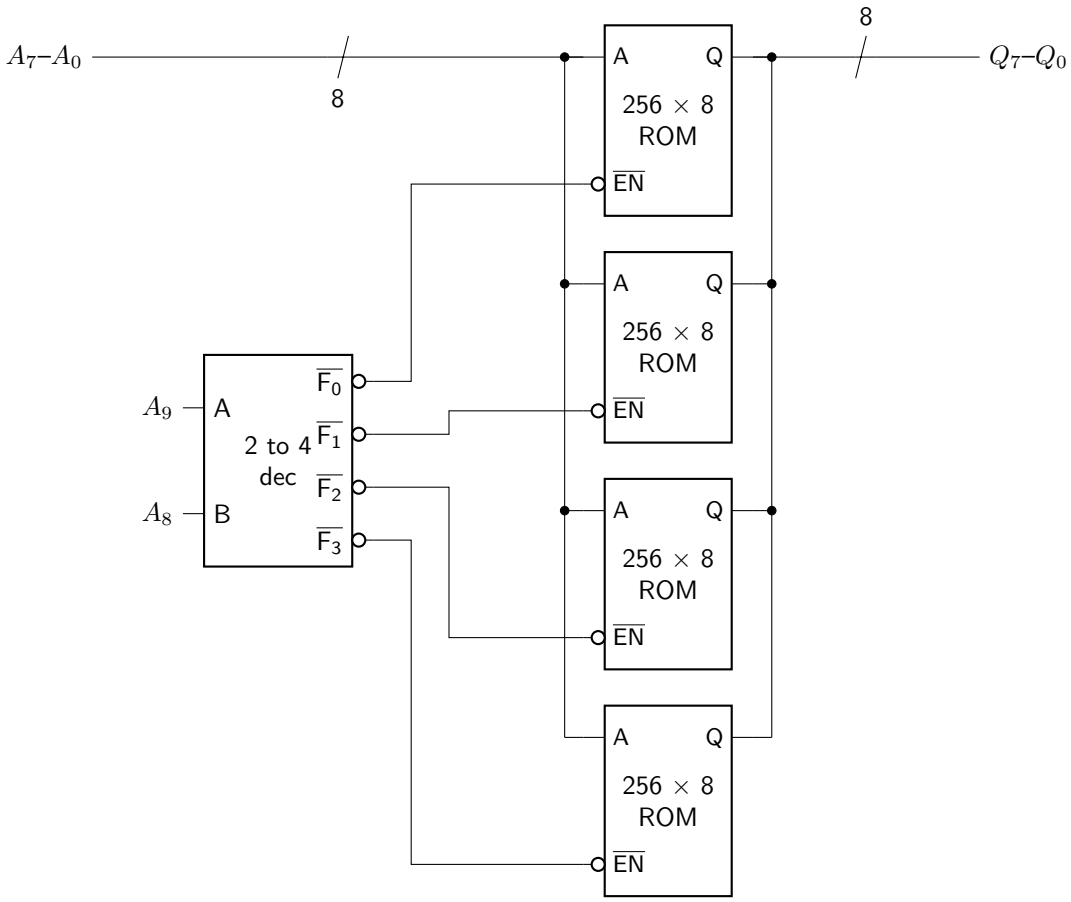


Figure 14.26: Extending 256 × 8 ROM to create 1024 × 8 ROM.

5. $16 \times 1024 = 16{,}384$ bytes

Programmable Logic

1. $F(A, B, C) = A'B'C' + A'BC' + AB'C$

2. The PAL/GAL digram is shown in Figure 14.27.

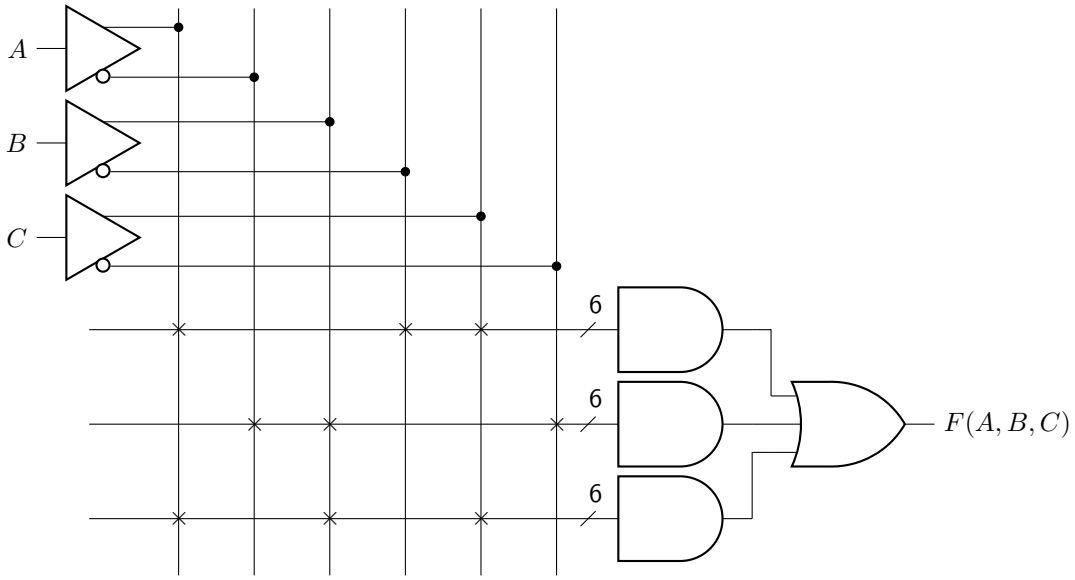


Figure 14.27: Simplified PAL/GAL diagram for question 2.

3. The input to the XOR gate should be one. This will toggle output values from zero to one and vice versa.

4. $F(A, B) = A'B + AB'$

5. $P(A,B,C,D,E) = A'B'C'D' + A'B'C'E'$ and $L(A,B,C,D,E) = A' + B'C'D' + B'C'E'$. The PAL/-GAL is shown in Figure 14.28.

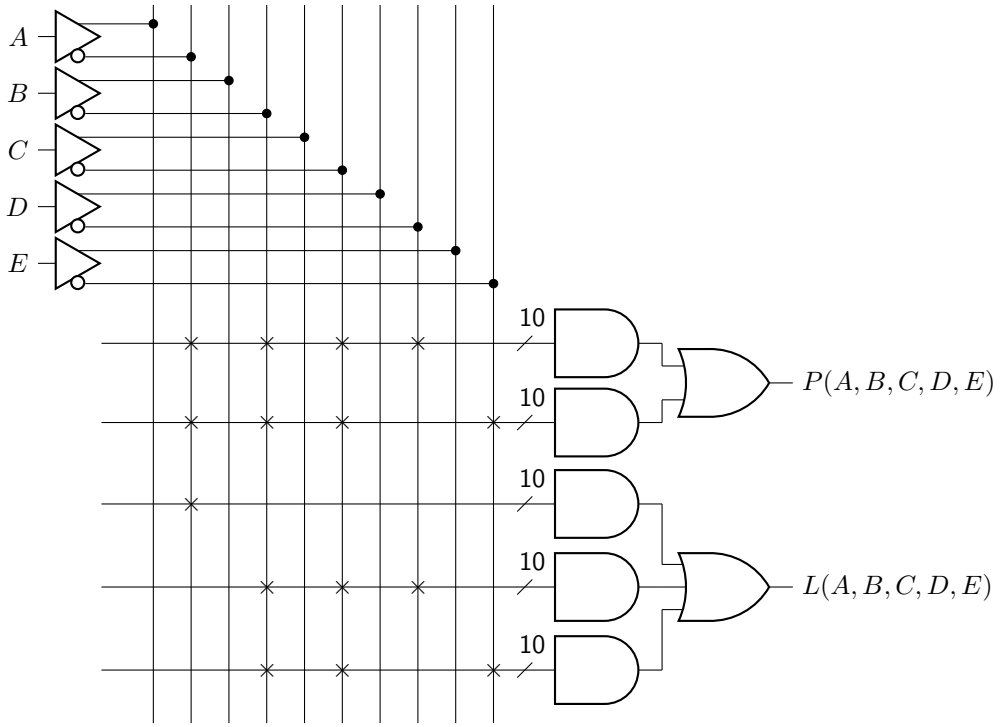


Figure 14.28: Simplified PAL/GAL diagram for question 5.

14.10 Chapter 11

SR Latch

1. The output timing diagram is shown in Figure 14.29.

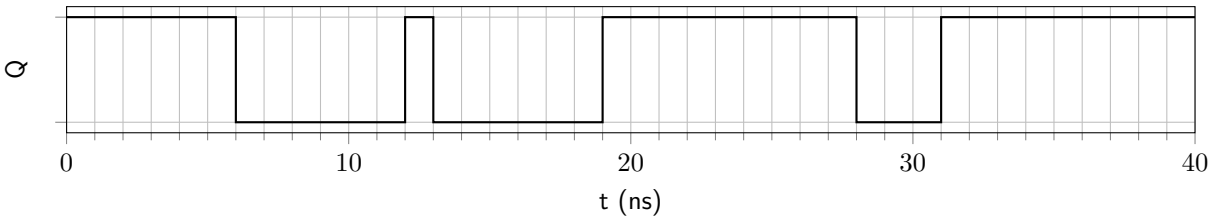


Figure 14.29: Timing diagram corresponding to SR latch question 1.

2. The schematic corresponding to this question is in Figure 14.30.

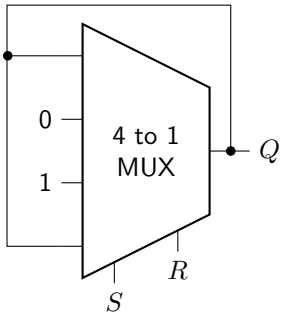


Figure 14.30: Circuit schematic corresponding to SR latch question 2.

3. The latch will reset when G and $H = 0$. The latch will hold when $G = 0$ and $H = 1$. Any time G is 1, the latch will set.

4. The Q output is the output of the NAND gate with the S' input. The Q' output is the output of the NAND gate with the R' input.

5. The circuit diagram is shown in Figure 14.31.

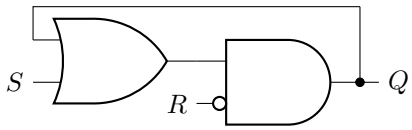


Figure 14.31: AND/OR SR latch circuit diagram corresponding to question 5.

D Latch

1. The timing diagram is given in Figure 14.32.

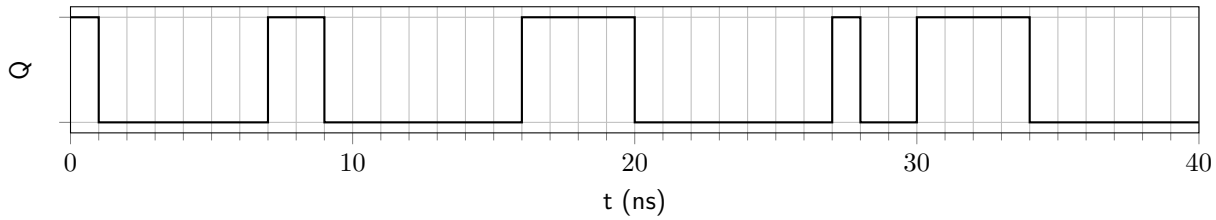


Figure 14.32: Input signals for D latch question 1.

2. The NAND gate D latch is shown in Figure 14.33.

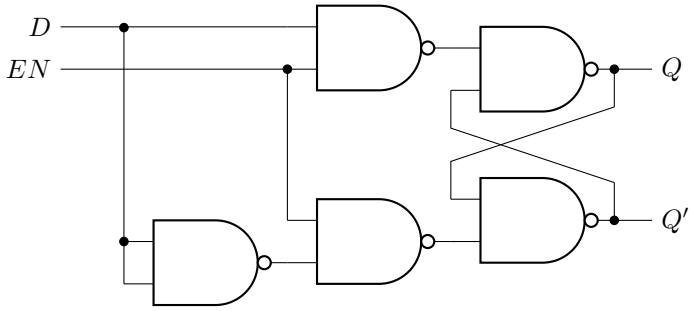


Figure 14.33: NAND gate D latch corresponding to question 2.

3. The circuit diagram for the primary-secondary D flip-flop is shown in Figure 14.34.

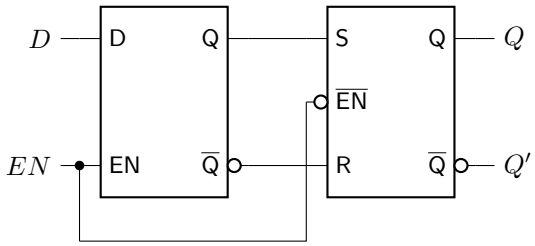


Figure 14.34: Primary-secondary D flip-flop corresponding to question 3.

4. Pins four and thirteen are the enable pins. They are active HIGH enable.

5. There are two enable pins per latch and both are active LOW enable.

D Flip-Flop

1. On the next rising edge, $Q = 0$.

2. On the next rising edge, $Q = 0$.

3. The circuit diagram for a T flip-flop created from a D flip-flop is given in Figure 14.35.

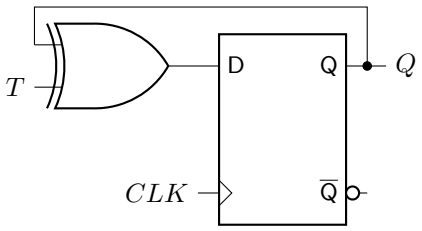


Figure 14.35: A T flip-flop designed from a D flip-flop.

4. The circuit diagram is given in Figure 14.36.

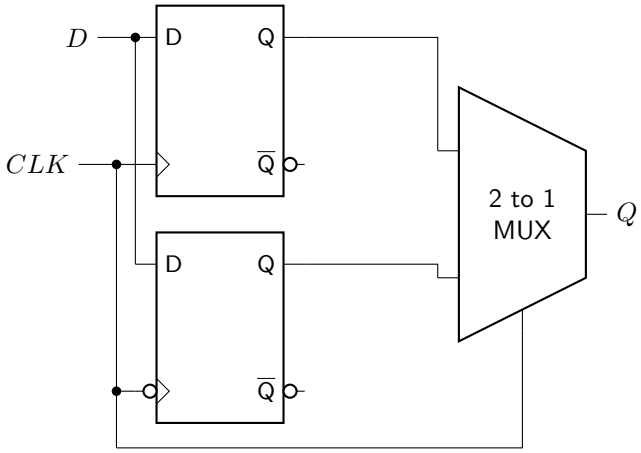


Figure 14.36: D flip-flop that updates Q on both rising and falling edges.

5. The circuit diagram for the scan flip-flop is given in Figure 14.37.

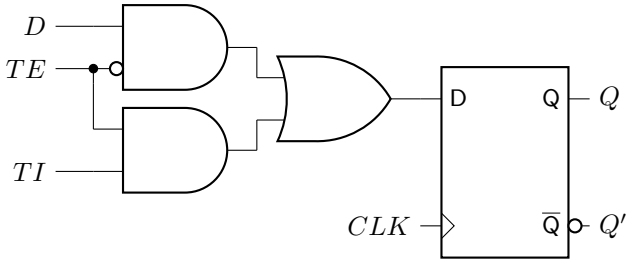


Figure 14.37: Scan flip-flop circuit diagram.

T Flip-Flop

1. The timing digram is shown in Figure 14.38.

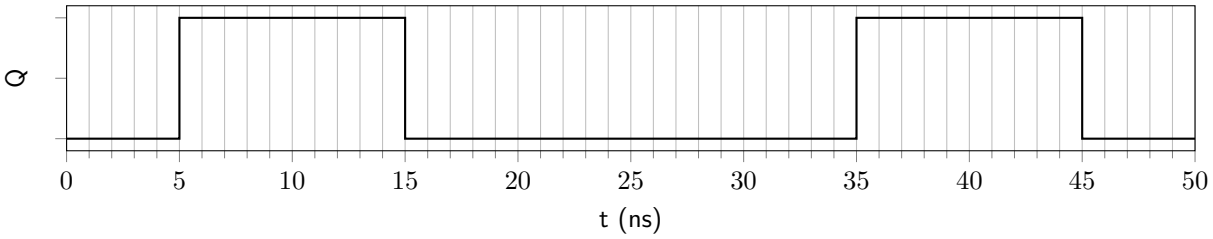


Figure 14.38: Output signal for T flip-flop question 1.

2. The timing diagram is shown in Figure 14.39.

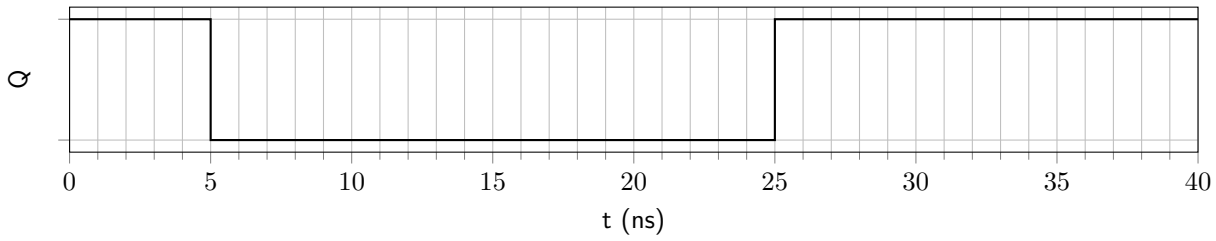


Figure 14.39: Output signal for T flip-flop question 2.

3. The circuit diagram for a JK flip-flop created from a T flip-flop is given in Figure 14.40.

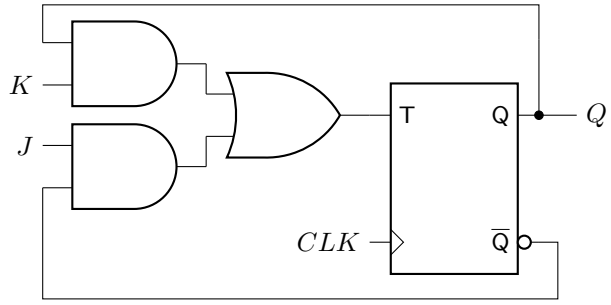


Figure 14.40: A JK flip-flop designed from a T flip-flop.

4. The circuit diagram for an SR flip-flop created from a T flip-flop is given in Figure 14.41.

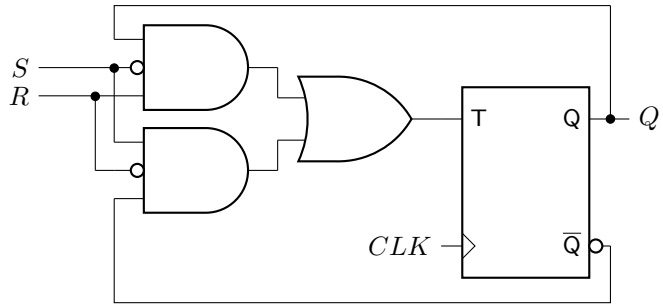


Figure 14.41: An SR flip-flop designed from a T flip-flop.

5. The circuit diagram for a T flip-flop with active LOW asynchronous preset and clear inputs is given in Figure 14.42.

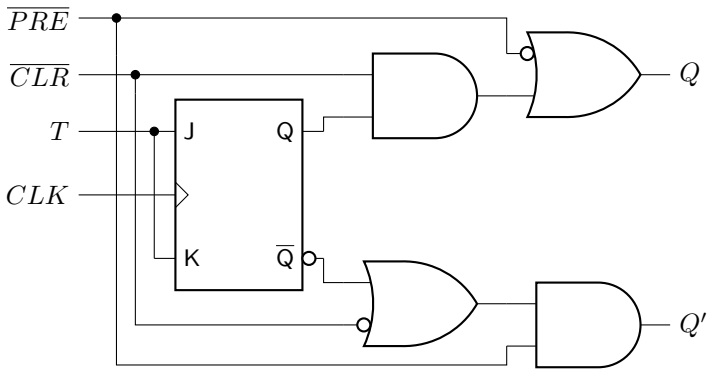


Figure 14.42: A T flip-flop with active LOW asynchronous preset and clear using a JK flip-flop.

JK Flip-Flop

1. The output timing diagram is shown in Figure 14.43.

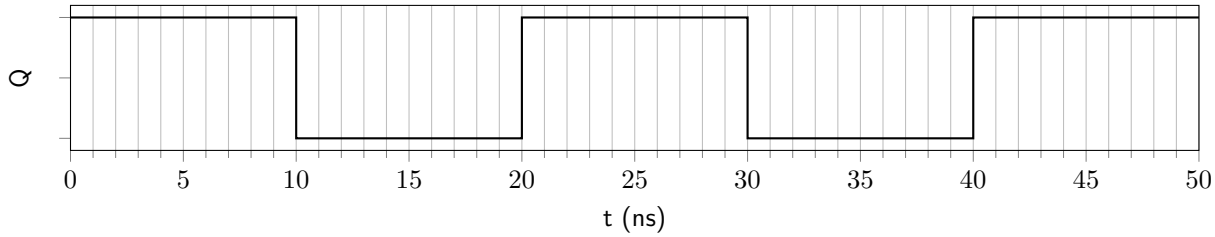


Figure 14.43: Output signal for JK flip-flop question 1.

2. The output timing diagram is shown in Figure 14.44.

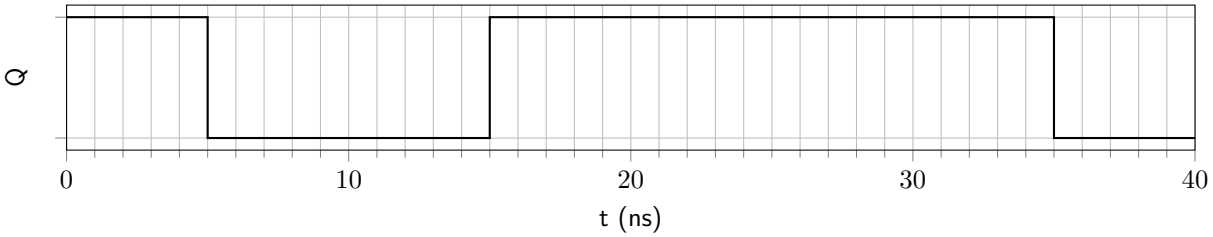


Figure 14.44: Output signal for JK flip-flop question 2.

3. The circuit diagram for a primary-secondary JK flip-flop is shown in Figure 14.45.

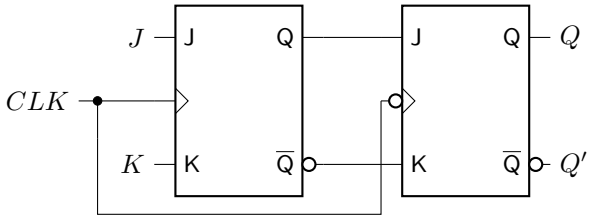


Figure 14.45: Primary-secondary JK flip-flop.

4. The circuit diagram for a JK flip-flop created from a D flip-flop is given in Figure 14.46.

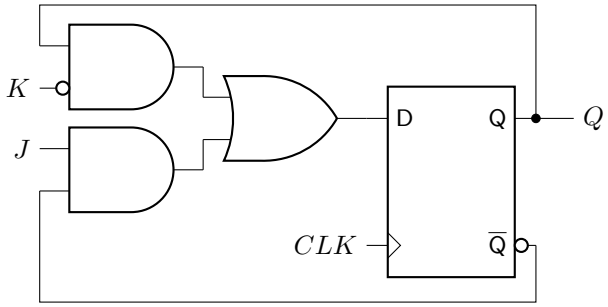


Figure 14.46: A JK flip-flop created with a D flip-flop.

5. The circuit diagram for an AB flip-flop created from a JK flip-flop is given in Figure 14.47.

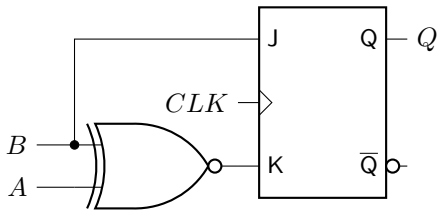


Figure 14.47: An AB flip-flop created with a JK flip-flop.

14.11 Chapter 12

Registers

1. The circuit diagram for the bidirectional shift register is given in Figure 14.48.

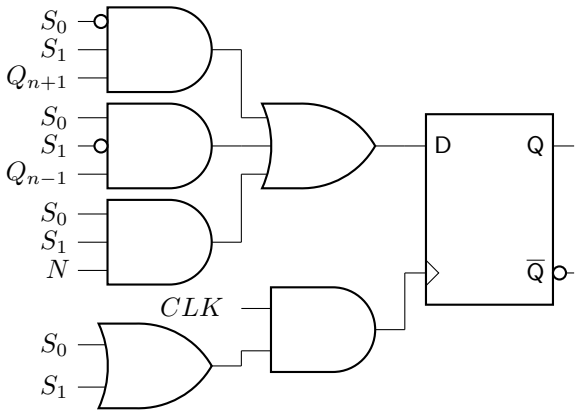


Figure 14.48: Control logic for a bi-directional shift register.

2. The SR flip-flop SISO register is shown in Figure 14.49.

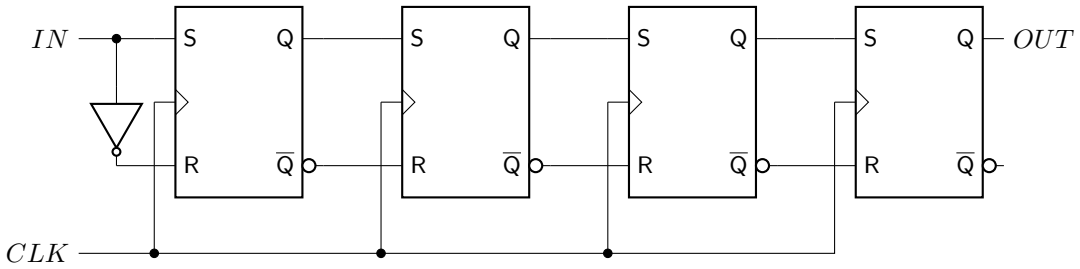


Figure 14.49: 4-bit SISO register designed from SR flip-flops.

3. 200 ns

4. The 8-bit bidirectional shift register is shown in Figure 14.50.

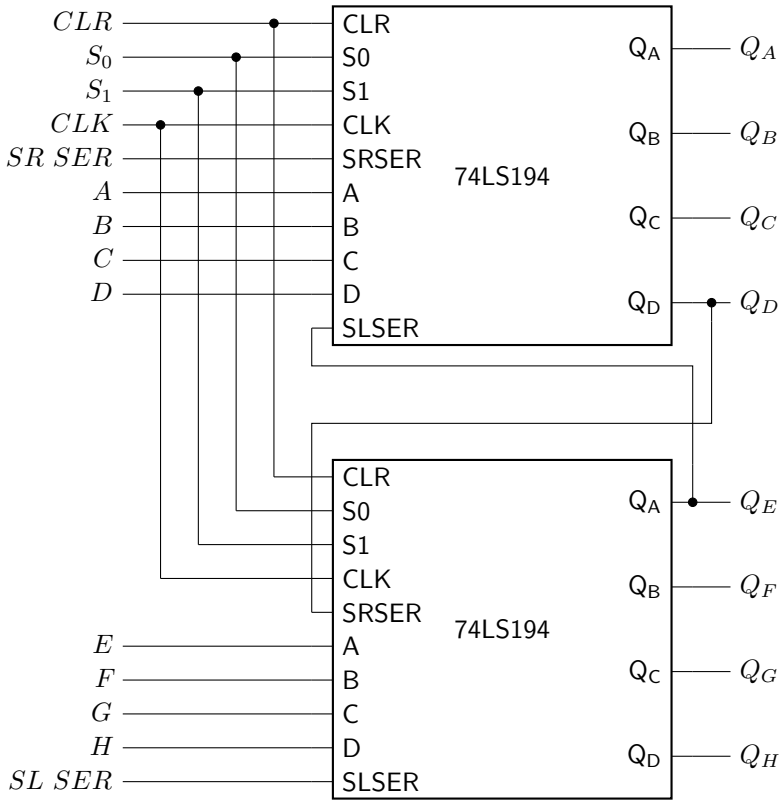


Figure 14.50: Two 74LS194 bidirectional shift registers daisy-chained together to create an 8-bit bidirectional shift register.

5. Initial value: 0000, then 1000, 1100, 0110, 1011, 0101, 0010

Synchronous Counters

1. $D_A = A'(B \oplus C)$, $D_B = A' \oplus C$, $D_C = A'(B' + C)$

2. $T_A = B \oplus C$, $T_B = B' \oplus C$, $T_C = B'C' + AC$

3. $T_A = A' \oplus B$, $T_B = B \oplus C$, $T_C = A(B + C')$

4. The actual state diagram for the counter designed using don't cares on state 000 is shown in Figure 14.51.

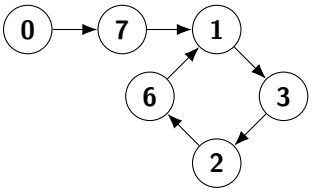


Figure 14.51: State diagram for the synchronous counter designed in question 4.

5. $D_A = A'C'$, $D_B = A'$, $D_C = A + B'C$

Ripple Counters

1. The schematic for the 4-bit ripple counter designed from D flip-flops is shown in Figure 14.52.

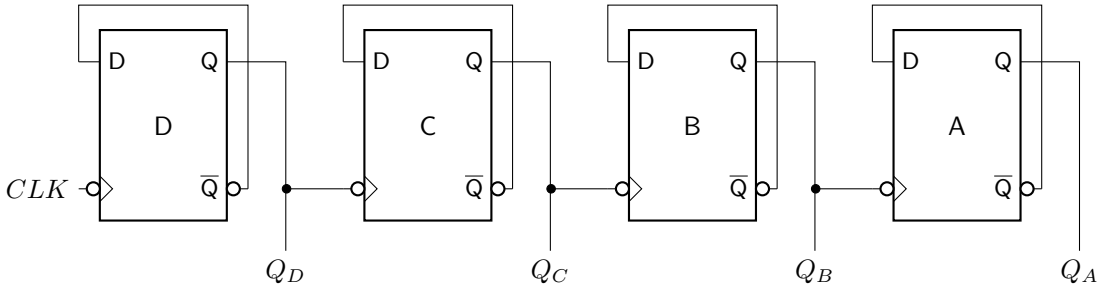


Figure 14.52: 4-bit ripple counter designed from D flip-flops.

2. The schematic for the 3-bit ripple down counter designed from T flip-flops is shown in Figure 14.53.

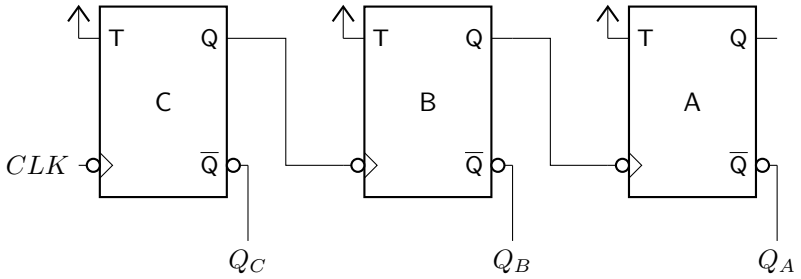


Figure 14.53: 3-bit ripple down counter designed from T flip-flops.

3. The circuit diagram for the 4-bit ripple counter is shown in Figure 14.54.

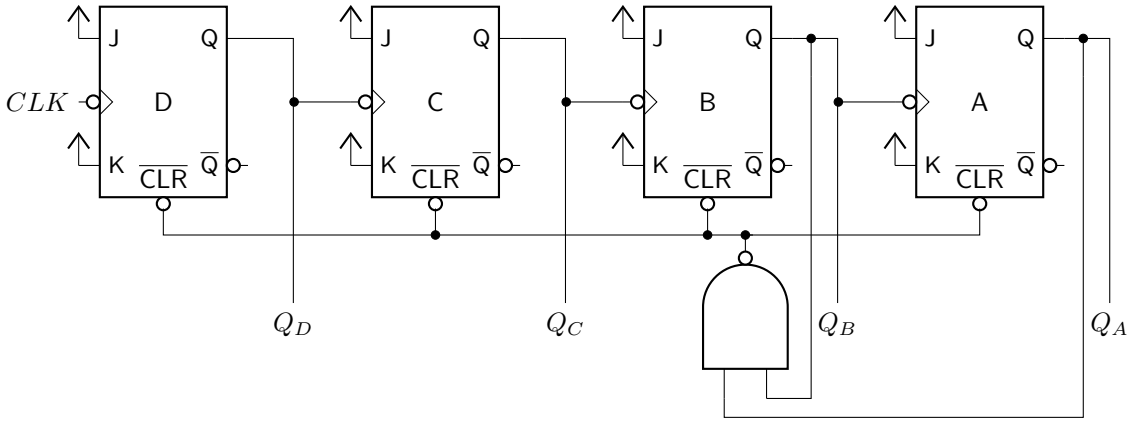


Figure 14.54: Ripple counter that counts from 0–11 designed from JK flip-flops.

4. The circuit diagram for the 4-bit ripple counter is shown in Figure 14.55.

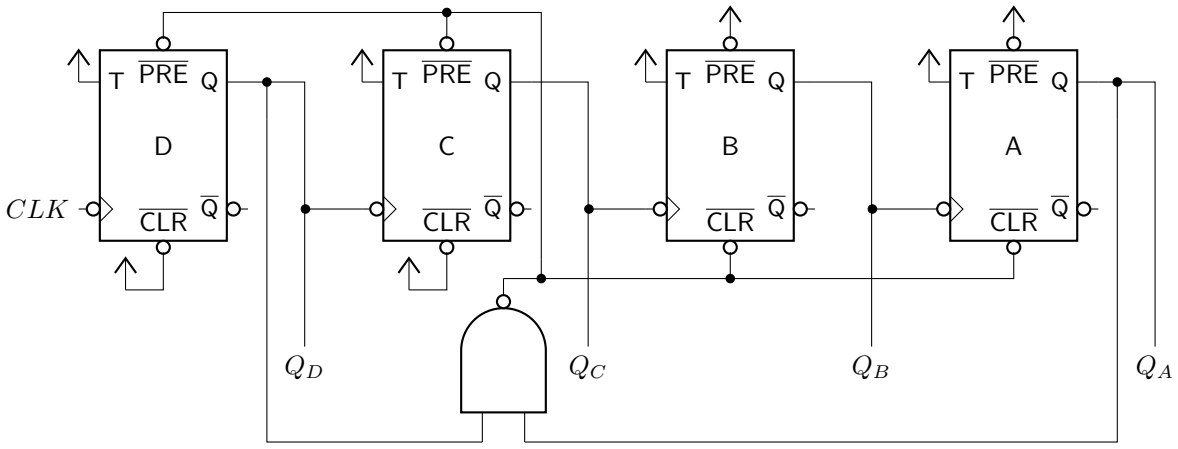


Figure 14.55: Ripple counter corresponding to question 4.

5. The timing diagrams are shown in Figure 14.56. Notice the output glitch between 260–270 ns corresponding to the delay on the NAND gate causing the asynchronous clear.

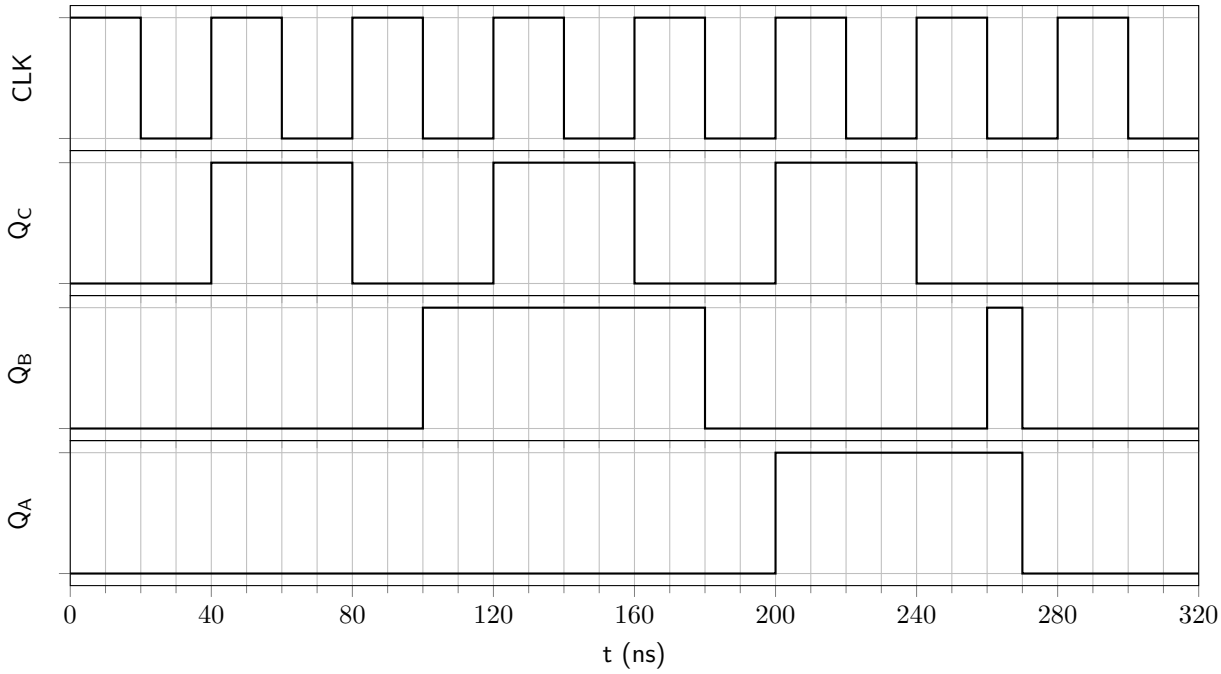


Figure 14.56: Output signals for 3-bit 0–5 counter.

14.12 Chapter 13

Moore Machines

1. The state table is shown in table 14.13.

Current State	Next State X=0	Next State X=1	Output
A	A	B	0
B	C	B	0
C	D	E	0
D	A	B	1
E	F	B	0
F	D	E	1

Table 14.13: State table for Moore machine question 1.

2. $A^+ = X'A'C$, $B^+ = A$, $C^+ = X + A'B'C$, $Z = BC' + AC'$

3. The state diagram corresponding to the circuit diagram is shown in Figure 14.57.

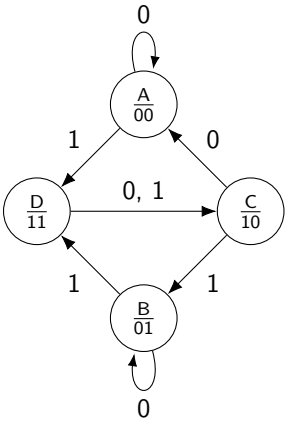


Figure 14.57: State diagram of the Moore machine corresponding to question 3.

4. $J_A = X'C + XB$, $K_A = X$, $J_B = XA'C'$, $K_B = 1$, $J_C = X'A'B' + XA$, $K_C = 1$, $Z = C$

5. $D_A = XB'C + XBC' + X'A'BC + X'AB' + X'AC'$, $D_B = X'B'C + X'BC' + XA$, $D_C = X'C' + XB$, $EVEN = A \oplus B \oplus C$

Mealy Machines

1. The state table is shown in table 14.14.

Current State	Next State		Outputs (YZ)	
	X=0	X=1	X=0	X=1
A	B	F	00	00
B	C	D	01	00
C	A	C	10	11
D	E	C	00	00
E	B	E	01	11
F	E	B	10	00

Table 14.14: State table corresponding to Mealy machine question 1.

2. The state table for the Mealy machine even parity bit generator is shown in table 14.15. The state definitions are S_0 reset; A 0; B 1; C 00; D 01; E 10; F 11.

Current State	Next State		Outputs (YZ)	
	X=0	X=1	X=0	X=1
S0	A	B	0	0
A	C	D	0	0
B	E	F	0	0
C	C	D	0	1
D	E	F	1	0
E	C	D	1	0
F	E	F	0	1

Table 14.15: State table corresponding to Mealy machine question 2.

3. The state table corresponding to the circuit diagram is shown in table 14.16.

Current State	Next State				Output (Z)			
	XY=00	XY=01	XY=10	XY=11	XY=00	XY=01	XY=10	XY=11
A	A	C	A	C	1	1	0	0
B	B	D	B	D	0	0	0	0
C	C	B	C	A	1	1	0	0
D	D	A	D	B	0	0	0	0

Table 14.16: State diagram of the Mealy machine in question 3.

4. $J_A = X'C + XBC'$, $K_A = X' + C$, $J_B = A'C' + X'C' + XAC$, $K_B = 1$, $J_C = XA' + X'A$, $K_C = A'B' + XB'$, $Z = XB'C + X'B + BC'$

5. $D_A = SB'C' + S'BC$, $D_B = AB' + SA'$, $D_C = S'A'C' + SB'C'$, $Q = A'C + SA'B + S'C$

Sequence Detectors

1. The state diagram for the Mealy machine disjoint window detector (sequence: 1101) is shown in Figure 14.58.

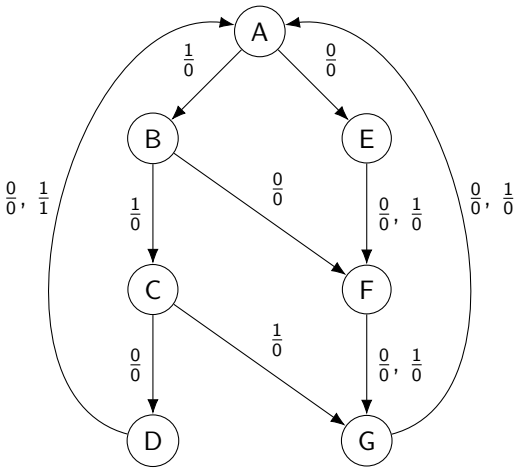


Figure 14.58: State diagram of the disjoint window Mealy machine detector (sequence 1101).

2. The state table for the Moore machine non-overlapping sliding window detector (sequence: 101101) is shown in table 14.17. The states are S_0 reset ($Z = 0$); S_1 1 ($Z = 0$); S_2 10 ($Z = 0$); S_3 101 ($Z = 0$); S_4 1011 ($Z = 0$); S_5 10110 ($Z = 0$); S_6 101101 ($Z = 1$).

Current State	Next State		Output
	X=0	X=1	
S0	S0	S1	0
S1	S2	S1	0
S2	S0	S3	0
S3	S2	S4	0
S4	S5	S1	0
S5	S0	S6	0
S6	S0	S1	1

Table 14.17: State table of the non-overlapping sliding window Moore machine detector (sequence 101101).

3. The state table for the Mealy machine overlapping sliding window detector (sequence: 10101) is shown in table 14.18. The states are A reset; B 1; C 10; D 101; E 1010.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	A	B	0	0
B	C	B	0	0
C	A	D	0	0
D	E	B	0	0
E	A	D	0	1

Table 14.18: State table of the overlapping sliding window Mealy machine detector (sequence 10101).

4. The state diagram for the Moore machine sequential 1 detector is shown in Figure 14.59. S_0 reset ($Z = 0$), S_1 1 ($Z = 0$), S_2 11 ($Z = 1$).

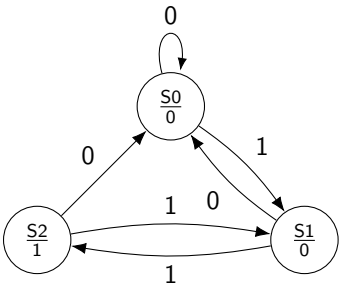


Figure 14.59: State diagram of the Moore machine sequence detector corresponding to question 4.

5. The state table for the Mealy machine sequence detector is shown in table 14.19. S_0 reset; S_1 0; S_2 01; S_3 0 after 01.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
S0	S1	S0	0	0
S1	S1	S2	0	1
S2	S3	S2	0	0
S3	S1	S2	0	0

Table 14.19: State table of the Mealy machine sequence detector from question 5.

State Machine Completeness

1. State S1 is over-defined for one transition and under-defined for one transition. State S2 is under-defined for two transitions.

2. State S1 has no problems. State S2 is over-defined for one transition. State S3 is under-defined for one transition.

3. States A and C have no problems. State B is under-defined for two transitions. State D is under-defined for one transition and over-defined for one transition.

4. States Q1 and Q3 have no problems. State Q0 is under-defined for two transitions. State Q2 is under-defined for two transitions and over-defined for two transitions.

5. State Z has no problems. State W is over-defined for one transition. State X is over-defined for six transitions. State Y is under-defined for three transitions and over-defined for one transition.

State Table Reduction

1. The minimized state table is shown in table 14.20. $A \equiv B$, $C \equiv D$, and $E \equiv F$.

Current State	Next State		Output
	X=0	X=1	
A	E	A	1
C	E	C	0
E	A	C	0

Table 14.20: State table for state table reduction question 1.

2. The minimized state table is shown in table 14.21. $A \equiv F \equiv G$, $B \equiv C$, and $E \equiv H$.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	A	B	0	0
B	E	A	0	1
D	E	D	1	0
E	B	A	1	1

Table 14.21: State table for state table reduction question 2.

3. The minimized state table is shown in table 14.22. $a \equiv b$, $c \equiv e$, $d \equiv g \equiv i$, $f \equiv h$.

Current State	Next State		Output
	X=0	X=1	
a	c	c	1
c	d	f	0
d	f	a	1
f	c	d	0
j	f	a	0

Table 14.22: State table for state table reduction question 3.

4. The minimized state table is shown in table 14.23. $B \equiv D$.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	B	C	0	1
B	C	A	0	1
C	B	B	1	0

Table 14.23: State table for state table reduction question 4.

5. The minimized state table is shown in table 14.24. $A \equiv D$, $B \equiv E$, and $C \equiv F$.

Current State	Next State		Output	
	X=0	X=1	X=0	X=1
A	B	A	0	0
B	A	C	1	0
C	C	A	0	1
G	H	G	1	1
H	C	B	1	1

Table 14.24: State table for state table reduction question 5.

State Machine Equivalence

1. The two state machines are equivalent. $S_0 \equiv E \equiv F$, $S_1 \equiv C \equiv D$, and $S_2 \equiv S_3 \equiv A \equiv B$.
2. The two state machines are equivalent. $A \equiv S_0 \equiv S_3$, $B \equiv S_2 \equiv S_5$, $C \equiv S_1$, and $D \equiv S_4$.
3. The two state machines cannot be equivalent; one of them is a Moore machine and the other is a Mealy machine.
4. The two state machines are not equivalent.

5. The two state machines are equivalent. The state diagram corresponding to the state machines is shown in Figure 14.60.

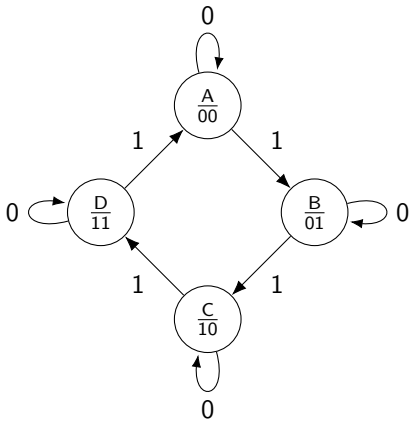


Figure 14.60: State diagram of both state machines in question 5.

Index

AND gate, 36
arithmetic
    binary addition, 23, 233
    binary multiplication, 27, 235
    binary subtraction, 26
    overflow, 20, 24, 26, 28

binary, 14, 20
    addition, 23, 233
    ASCII, 32
    binary coded decimal (BCD), 30, 193
    conversion, 19
    full adder, 69, 233
    gray code, 31
    half adder, 69, 233
    multiplication, 27, 235
    non-integer numbers, 20
    parity, 71
    ripple carry adder, 70
    signed numbers, 22
    subtraction, 26
    two's complement, 22
    weighted binary codes, 30
Boolean algebra, 35, 66
    consensus theorem, 51
    decoder, 187
    don't care, 74, 81
    equivalent expressions, 42
    essential prime implicants, 95
    expressions, 37, 39, 40
    implicants, 94
    maxterms, 79, 82, 83
    minimum expressions, 47, 87, 103, 122, 172, 187, 200, 280, 285
    minterms, 77, 82, 83
    multiplexers, 172
    prime implicant table, 96, 103
    prime implicants, 95, 160
    product of sums (POS), 45, 47
    ROM, 200
    sum of products (SOP), 43, 47
bubble method, 112, 116
buffer gate, 35, 166

circuit diagram, 37, 39, 41
clock, 259
    clock skew, 260
    hold violation, 261
    setup violation, 260
counter
    Johnson counter, 249
    ring counter, 248
    ripple counter, 244
    synchronous counter, 238

D flip-flop, 215
decimal, 14, 15
    conversion, 15, 17
decoder, 186
    converting to multiplexers, 192
    expanding, 190
    speciality decoder, 193
demultiplexer (DEMUX), 181, 185
diode, 151
don't cares, 74, 81, 277
    Karnaugh maps, 93
double prime method, 112, 116
dynamic hazard, 161

encoder, 194

priority encoder, 196

finite-state machine (FSM)
circuit equivalence, 284
don't cares, 277
implication table, 282
Mealy machine, 256
Moore machine, 252
multiple inputs, 276
multiple outputs, 274
sequence detector, 261
state assignment, 252, 285, 289
state diagram, 252, 278
state table, 252, 280
transition table, 252
flip-flop
asynchronous inputs, 223
converting flip-flops, 222
D flip-flop, 215
hold time, 221, 260
JK flip-flop, 219
primary-secondary, 214
setup time, 221, 260
SR flip-flop, 214, 218
T flip-flop, 217

gated D latch, 215
gated SR latch, 214
gray code, 31

hexadecimal
conversion, 19

JK flip-flop, 219

Karnaugh maps, 87, 160
don't cares, 93
five variables, 99
four variables, 91
three variables, 88

latch
gated D latch, 215
gated SR latch, 214
SR latch, 212
logic gates, 35
7400 series, 58
active HIGH, 166, 180, 187
active LOW, 166, 180, 187
AND, 36
buffer, 35, 166
delay, 156, 159, 161
dual in-line package (DIP), 59
inverter, 36
NAND, 111
NOR, 115
OR, 37
programmable, 202
resistor-transistor logic (RTL), 152
TTL, 57, 153
XNOR, 55
XOR, 54

maxterms, 79, 82, 83, 187
memory, 198
non-volatile, 199
volatile, 198
minterms, 77, 82, 83, 187
multiplexer (MUX), 169, 185
2 to 1 MUX, 169
4 to 1 MUX, 171

NAND gate, 111
NOR gate, 115
NOT gate, 36
number systems, 14
binary, 14, 20

decimal, 14, 15

octal
conversion, 19
optimization
least cost, 122
least time, 122
multiple outputs, 125
OR gate, 37

product of sums (POS), 45, 47, 122
converting to SOP, 52
minimum POS, 49
programmable logic, 202
generic array logic (GAL), 202
output logic macrocells, 204
programmable array logic (PAL), 202

Quine-McCluskey method, 103

register
parallel in / parallel out (PIPO), 233
parallel in / serial out (PISO), 232
serial in / parallel out (SIPO), 231
serial in / serial out (SISO), 231

semiconductors, 149
bipolar junction transistor (BJT), 151
metal oxide semiconductor field effect transistor (MOSFET), 153
P-N junction, 151
sequence detector
complicated sequence detectors, 268
disjoint window, 266
non-overlapping sliding window, 265
overlapping sliding window, 261
sequential circuits, 212
addition, 233
circuit equivalence, 284
comparison, 272
counter, 238, 244
finite-state machine (FSM), 252
flip-flop, 215, 217–219, 222
gated latch, 214, 215
latch, 212
multiplication, 235
registers, 231
reset state, 242
SR flip-flop, 214, 218
SR latch, 212
static hazard, 159, 160
static one hazard, 159, 160
static zero hazard, 159, 160
sum of products (SOP), 43, 47, 122
converting to POS, 53
minimum SOP, 49

T flip-flop, 217
timing diagrams, 157, 159, 161
transistor
bipolar junction transistor (BJT), 151
metal oxide semiconductor field effect transistor (MOSFET), 153
tri-state, 166
truth table, 40–42, 49, 68, 200
two's complement, 22

XNOR gate, 55
XOR gate, 54