EXPLORING DIGITAL LOGIC

With Logisim-Evolution

GEORGE SELF

July 2018 – Edition 6.0

George Self: *Exploring Digital Logic,* With Logisim-Evolution

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 4.0 International" license.

BRIEF CONTENTS

List of Figures xii
List of Tables xv
Listings xviii

I THEORY
1 INTRODUCTION 3
2 FOUNDATIONS OF BINARY MATHEMATICS 11
3 BINARY MATHEMATICS OPERATIONS 35
4 BOOLEAN FUNCTIONS 63
5 BOOLEAN EXPRESSIONS 91
6 KARNAUGH MAPS 113
7 ADVANCED SIMPLIFYING METHODS 141

II PRACTICE
8 COMBINATIONAL ARITHMETIC CIRCUITS 157
9 COMBINATIONAL ENCODER CIRCUITS 173
10 SEQUENTIAL REGISTER CIRCUITS 195
11 SEQUENTIAL COUNTERS 207
12 SIMULATION 219
13 SIMULATION 223
14 SIMULATION 225

III APPENDIX
A BOOLEAN PROPERTIES AND FUNCTIONS 235
B IC REFERENCE 237

GLOSSARY 239

BIBLIOGRAPHY 241

CONTENTS

List of Figures xii
List of Tables xv
Listings xviii

I THEORY
1 INTRODUCTION 3
1.1 Preface 3
1.1.1 Introduction to the Study of Digital Logic 3
1.1.2 Introduction to the Author 3
1.1.3 Introduction to This Book 3
1.1.4 About the Creative Commons License 5
1.2 About Digital Logic 5
1.2.1 Introduction 5
1.2.2 A Brief Electronics Primer 6
1.3 Boolean Algebra 8
1.3.1 History 8
1.3.2 Boolean Equations 9
1.4 About This Book 10
2 FOUNDATIONS OF BINARY MATHEMATICS 11
2.1 Introduction to Number Systems 11
2.1.1 Background 11
2.1.2 Binary Mathematics 12
2.1.3 Systems Without Place Value 12
2.1.4 Systems With Place Value 13
2.1.5 Summary of Numeration Systems 16
2.1.6 Conventions 18
2.2 Converting Between Radices 19
2.2.1 Introduction 19
2.2.2 Expanded Positional Notation 19
2.2.3 Binary to Decimal 20
2.2.4 Binary to Octal 22
2.2.5 Binary to Hexadecimal 23
2.2.6 Octal to Decimal 24
2.2.7 Hexadecimal to Decimal 25
2.2.8 Decimal to Binary 26
2.2.9 Calculators 30
2.2.10 Practice Problems 30
2.3 Floating Point Numbers 31
3 BINARY MATHEMATICS OPERATIONS 35
3.1 Binary Addition 35
3.1.1 Overflow Error 36
3.1.2 Sample Binary Addition Problems 37

- 3.2 Binary Subtraction 37
  - 3.2.1 Simple Manual Subtraction 37
  - 3.2.2 Representing Negative Binary Numbers Using Sign-and-Magnitude 38
  - 3.2.3 Representing Negative Binary Numbers Using Signed Complements 39
  - 3.2.4 Subtracting Using the Diminished Radix Complement 42
  - 3.2.5 Subtracting Using the Radix Complement 44
  - 3.2.6 Overflow 45
- 3.3 Binary Multiplication 47
  - 3.3.1 Multiplying Unsigned Numbers 47
  - 3.3.2 Multiplying Signed Numbers 48
- 3.4 Binary Division 48
- 3.5 Bitwise Operations 49
- 3.6 Codes 49
  - 3.6.1 Introduction 49
  - 3.6.2 Computer Codes 50

4 BOOLEAN FUNCTIONS 63
- 4.1 Introduction to Boolean Functions 63
- 4.2 Primary Logic Operations 65
  - 4.2.1 AND 65
  - 4.2.2 OR 67
  - 4.2.3 NOT 68
- 4.3 Secondary Logic Functions 69
  - 4.3.1 NAND 69
  - 4.3.2 NOR 70
  - 4.3.3 XOR 71
  - 4.3.4 XNOR 72
  - 4.3.5 Buffer 73
- 4.4 Univariate Boolean Algebra Properties 74
  - 4.4.1 Introduction 74
  - 4.4.2 Identity 74
  - 4.4.3 Idempotence 75
  - 4.4.4 Annihilator 76
  - 4.4.5 Complement 77
  - 4.4.6 Involution 77
- 4.5 Multivariate Boolean Algebra Properties 78
  - 4.5.1 Introduction 78
  - 4.5.2 Commutative 78
  - 4.5.3 Associative 79
  - 4.5.4 Distributive 80
  - 4.5.5 Absorption 81
  - 4.5.6 Adjacency 82
- 4.6 DeMorgan's Theorem 83
  - 4.6.1 Introduction 83

- 4.6.2 Applying DeMorgan's Theorem 84
- 4.6.3 Simple Example 84
- 4.6.4 Incorrect Application of DeMorgan's Theorem 85
- 4.6.5 About Grouping 85
- 4.6.6 Summary 86
- 4.6.7 Example Problems 87
- 4.7 Boolean Functions 87
- 4.8 Functional Completeness 89

5 BOOLEAN EXPRESSIONS 91
- 5.1 Introduction 91
- 5.2 Creating Boolean Expressions 91
  - 5.2.1 Example 92
- 5.3 Minterms and Maxterms 94
  - 5.3.1 Introduction 94
  - 5.3.2 Sum Of Products (SOP) Defined 94
  - 5.3.3 Product of Sums (POS) Defined 94
  - 5.3.4 About Minterms 95
  - 5.3.5 About Maxterms 97
  - 5.3.6 Minterm and Maxterm Relationships 99
  - 5.3.7 Sum of Products Example 101
  - 5.3.8 Product of Sums Example 102
  - 5.3.9 Summary 103
- 5.4 Canonical Form 104
  - 5.4.1 Introduction 104
  - 5.4.2 Converting Terms Missing One Variable 106
  - 5.4.3 Converting Terms Missing Two Variables 106
  - 5.4.4 Summary 108
  - 5.4.5 Practice Problems 109
- 5.5 Simplification Using Algebraic Methods 109
  - 5.5.1 Introduction 109
  - 5.5.2 Starting From a Circuit 109
  - 5.5.3 Starting From a Boolean Equation 110
  - 5.5.4 Practice Problems 112

6 KARNAUGH MAPS 113
- 6.1 Introduction 113
- 6.2 Reading Karnaugh maps 114
- 6.3 Drawing Two-Variable Karnaugh maps 115
- 6.4 Drawing Three-Variable Karnaugh maps 116
  - 6.4.1 The Gray Code 117
- 6.5 Drawing Four-Variable Karnaugh maps 117
- 6.6 Simplifying Groups of Two 119
- 6.7 Simplifying Larger Groups 122
  - 6.7.1 Groups of 16 122
  - 6.7.2 Groups of Eight 123
  - 6.7.3 Groups of Four 123
  - 6.7.4 Groups of Two 125

6.8 Overlapping Groups 126
6.9 Wrapping Groups 127
6.10 Karnaugh maps for Five-Variable Inputs 128
6.11 “Don’t Care” Terms 130
6.12 Karnaugh map Simplification Summary 131
6.13 Practice Problems 132
6.14 Reed-Müller Logic 133
6.15 Introduction 133
6.16 Zero In First Cell 134
  6.16.1 Two-Variable Circuit 134
  6.16.2 Three-Variable Circuit 134
  6.16.3 Four-Variable Circuit 135
6.17 One In First Cell 138
7 ADVANCED SIMPLIFYING METHODS 141
7.1 Quine-McCluskey Simplification Method 141
  7.1.1 Introduction 141
  7.1.2 Example One 141
  7.1.3 Example Two 146
  7.1.4 Summary 152
  7.1.5 Practice Problems 152
7.2 Automated Tools 154
  7.2.1 Introduction 154

II PRACTICE

8 COMBINATIONAL ARITHMETIC CIRCUITS 157
8.1 Introduction 157
8.2 Adders and Subtractors 157
  8.2.1 Introduction 157
  8.2.2 Half Adder 157
  8.2.3 Full Adder 158
  8.2.4 Cascading Adders 160
  8.2.5 Half Subtractor 161
  8.2.6 Full Subtractor 163
  8.2.7 Cascading Subtractors 164
  8.2.8 Adder-Subtractor Circuit 165
  8.2.9 Integrated Circuits 167
8.3 Arithmetic Logic Units 167
8.4 Comparators 168
9 COMBINATIONAL ENCODER CIRCUITS 173
9.1 Introduction 173
9.2 Multiplexers/Demultiplexers 173
  9.2.1 Multiplexer 173
  9.2.2 Demultiplexer 175
  9.2.3 Minterm Generators 176
9.3 Encoders/Decoders 177
  9.3.1 Introduction 177

9.3.2 Ten-Line Priority 178
9.3.3 Seven-Segment Display 179
9.3.4 Function Generators 182
9.4 Error Detection 182
9.4.1 Introduction 182
9.4.2 Iterative Parity Checking 184
9.4.3 Hamming Code 186
9.4.4 Hamming Code Notes 192
9.4.5 Sample Problems 193
10 SEQUENTIAL REGISTER CIRCUITS 195
10.1 Introduction 195
10.2 Timing Diagrams 195
10.3 Flip-Flops 197
10.3.1 Introduction 197
10.3.2 SR Latch 197
10.3.3 Data (D) Flip-Flop 199
10.3.4 JK Flip-Flop 200
10.3.5 Toggle (T) Flip-Flop 202
10.3.6 Master-Slave Flip-Flops 202
10.4 Registers 203
10.4.1 Introduction 203
10.4.2 Registers As Memory 203
10.4.3 Shift Registers 204
11 SEQUENTIAL COUNTERS 207
11.1 Introduction 207
11.2 Counters 207
11.2.1 Introduction 207
11.2.2 Asynchronous Counters 207
11.2.3 Synchronous Counters 210
11.2.4 Ring Counters 212
11.2.5 Modulus Counters 214
11.2.6 Up-Down Counters 215
11.2.7 Frequency Divider 216
11.2.8 Counter Integrated Circuits (IC) 217
11.3 Memory 217
11.3.1 Read-Only Memory 217
11.3.2 Random Access Memory 218
12 SIMULATION 219
12.1 Introduction 219
12.2 Finite State Machines 219
12.2.1 Introduction 219
12.2.2 Moore Finite State Machine 220
12.2.3 Mealy Finite State Machine 220
12.2.4 Finite State Machine Tables 221
13 SIMULATION 223
13.1 Introduction 223

13.2 Elevator 223
14 SIMULATION 225
14.1 Introduction 225
14.2 Central Processing Unit 225
14.2.1 Introduction 225
14.2.2 CPU States 231

III APPENDIX

A BOOLEAN PROPERTIES AND FUNCTIONS 235
B IC REFERENCE 237
B.1 7400: Quad 2-Input NAND Gate 237
B.2 7402: Quad 2-Input NOR Gate 238
B.3 7404: Hex Inverter 238
B.4 7408: Quad 2-Input AND Gate 238
B.5 7410: Triple 3-Input NAND Gate 238
B.6 7411: Triple 3-Input AND Gate 238
B.7 7413: Dual 4-Input NAND Gate (Schmitt-trigger) 238
B.8 7414: Hex Inverter (Schmitt-trigger) 238
B.9 7418: Dual 4-Input NAND Gate (Schmitt-trigger Inputs) 238
B.10 7419: Hex Inverter (Schmitt-trigger Inputs) 238
B.11 7420: Dual 4-Input NAND Gate 238
B.12 7421: Dual 4-Input AND Gate 238
B.13 7424: Quad 2-Input NAND Gate (Schmitt-trigger) 238
B.14 7427: Triple 3-Input NOR Gate 238
B.15 7430: Single 8-Input NAND Gate 238
B.16 7432: Quad 2-Input OR Gate 238
B.17 7436: Quad 2-Input NOR Gate 238
B.18 7442: BCD to Decimal Decoder 238
B.19 7443: Excess-3 to Decimal Decoder 238
B.20 7444: Gray to Decimal Decoder 238
B.21 7447: BCD to 7-Segment Decoder 238
B.22 7451: Dual AND-OR-INVERT Gate 238
B.23 7454: Four-wide AND-OR-INVERT Gate 238
B.24 7458: Dual AND-OR Gate 238
B.25 7464: 4-2-3-2 AND-OR-INVERT Gate 238
B.26 7474: Dual D-Flipflops with Preset and Clear 238
B.27 7485: 4-Bit Magnitude Comparator 238
B.28 7486: Quad 2-Input XOR Gate 238
B.29 74125: Quad Bus Buffer, 3-State Outputs, Neg Enable 238
B.30 74165: 8-Bit Parallel-to-Serial Shift Register 238
B.31 74175: Quad D-Flipflop, Asynchronous Reset 238
B.32 74266: Quad 2-Input XNOR Gate 238
B.33 74273: Octal D-Flipflop with Clear 238
B.34 74283: 4-Bit Binary Full Adder 238
B.35 74377: Octal D-Flipflop with Enable 238

GLOSSARY 239

BIBLIOGRAPHY 241

LIST OF FIGURES

Figure 1.1 Simple Lamp Circuit 6
Figure 1.2 Transistor-Controlled Lamp Circuit 6
Figure 1.3 Simple OR Gate Using Transistors 7
Figure 1.4 OR Gate 8
Figure 3.1 Optical Disc 59
Figure 4.1 AND Gate 66
Figure 4.2 AND Gate Using IEEE Symbols 67
Figure 4.3 OR Gate 68
Figure 4.4 NOT Gate 69
Figure 4.5 NAND Gate 70
Figure 4.6 NOR Gate 71
Figure 4.7 XOR Gate 72
Figure 4.8 XNOR Gate 73
Figure 4.9 Buffer 73
Figure 4.10 OR Identity Element 75
Figure 4.11 AND Identity Element 75
Figure 4.12 Idempotence Property for OR Gate 75
Figure 4.13 Idempotence Property for AND Gate 76
Figure 4.14 Annihilator For OR Gate 76
Figure 4.15 Annihilator For AND Gate 76
Figure 4.16 OR Complement 77
Figure 4.17 AND Complement 77
Figure 4.18 Involution Property 78
Figure 4.19 Commutative Property for OR 79
Figure 4.20 Commutative Property for AND 79
Figure 4.21 Associative Property for OR 80
Figure 4.22 Associative Property for AND 80
Figure 4.23 Distributive Property for AND over OR 81
Figure 4.24 Distributive Property for OR over AND 81
Figure 4.25 Absorption Property (Version 1) 81
Figure 4.26 Absorption Property (Version 2) 82
Figure 4.27 Adjacency Property 83
Figure 4.28 DeMorgan's Theorem Defined 83
Figure 4.29 DeMorgan's Theorem Example 1 84
Figure 4.30 DeMorgan's Theorem Example 2 85
Figure 4.31 DeMorgan's Theorem Example 2 Simplified 86
Figure 4.32 Generic Function 87
Figure 4.33 XOR Derived From AND/OR/NOT 89
Figure 5.1 Logic Diagram From Switching Equation 93
Figure 5.2 Logic Diagram For SOP Example 94
Figure 5.3 Logic Diagram For POS Example 95

Figure 5.4 Example Circuit 110
Figure 5.5 Example Circuit With Gate Outputs 110
Figure 6.1 Simple Circuit For K-Map 114
Figure 6.2 Karnaugh map for Simple Circuit 114
Figure 6.3 Karnaugh map With Greek Letters 115
Figure 6.4 Karnaugh map For Two-Input Circuit 116
Figure 6.5 Karnaugh map for Three-Input Circuit 116
Figure 6.6 K-Map For Four Input Circuit 118
Figure 6.7 K-Map For Sigma Notation 118
Figure 6.8 K-Map For PI Notation 119
Figure 6.9 K-Map for Groups of Two: Ex 1 120
Figure 6.10 K-Map for Groups of Two: Ex 1, Solved 120
Figure 6.11 K-Map Solving Groups of Two: Example 2 122
Figure 6.12 K-Map Solving Groups of 8 123
Figure 6.13 K-Map Solving Groups of Four, Example 1 124
Figure 6.14 K-Map Solving Groups of Four, Example 2 124
Figure 6.15 K-Map Solving Groups of Four, Example 3 125
Figure 6.16 K-Map Solving Groups of Two, Example 1 125
Figure 6.17 K-Map Solving Groups of Two, Example 2 126
Figure 6.18 K-Map Overlapping Groups, Example 1 126
Figure 6.19 K-Map Overlapping Groups, Example 2 127
Figure 6.20 K-Map Wrapping Groups Example 1 128
Figure 6.21 K-Map Wrapping Groups Example 2 128
Figure 6.22 K-Map for Five Variables, Example 1 129
Figure 6.23 K-Map Solving for Five Variables, Example 2 130
Figure 6.24 K-Map With “Don’t Care” Terms, Example 1 131
Figure 6.25 K-Map With “Don’t Care” Terms, Example 2 131
Figure 6.26 Reed-Müller Two-Variable Example 133
Figure 6.27 Reed-Müller Three-Variable Example 1 134
Figure 6.28 Reed-Müller Three-Variable Example 2 135
Figure 6.29 Reed-Müller Four-Variable Example 1 135
Figure 6.30 Reed-Müller Four-Variable Example 2 136
Figure 6.31 Reed-Müller Four-Variable Example 3 136
Figure 6.32 Reed-Müller Four-Variable Example 4 137
Figure 6.33 Reed-Müller Four-Variable Example 5 137
Figure 6.34 Reed-Müller Four-Variable Example 6 138
Figure 6.35 Reed-Müller Four-Variable Example 7 138
Figure 6.36 Reed-Müller Four-Variable Example 8 139
Figure 8.1 Half-Adder 158
Figure 8.2 K-Map For The SUM Output 159
Figure 8.3 K-Map For The COut Output 159
Figure 8.4 Full Adder 160
Figure 8.5 4-Bit Adder 161
Figure 8.6 Half-Subtractor 162
Figure 8.7 K-Map For The Difference Output 163

Figure 8.8 K-Map For The BOut Output 164
Figure 8.9 Subtractor 164
Figure 8.10 4-Bit Subtractor 165
Figure 8.11 4-Bit Adder-Subtractor 166
Figure 8.12 One-Bit Comparator 168
Figure 8.13 K-Map For A > B 169
Figure 8.14 K-Map For A = B 170
Figure 8.15 K-Map For A = B 170
Figure 8.16 Two-Bit Comparator 171
Figure 9.1 Multiplexer Using Rotary Switches 174
Figure 9.2 Simple Mux 174
Figure 9.3 Simple Dmux 175
Figure 9.4 1-to-4 Dmux 176
Figure 9.5 1-to-4 Dmux As Minterm Generator 177
Figure 9.6 Three-line to 2-Bit Encoder 177
Figure 9.7 Four-Bit to 4-Line Decoder 178
Figure 9.8 Seven-Segment Display 179
Figure 9.9 7-Segment Decoder 181
Figure 9.10 Hex Decoder 181
Figure 9.11 Minterm Generator 182
Figure 9.12 Parity Generator 184
Figure 9.13 Hamming Distance 186
Figure 9.14 Generating Hamming Parity 190
Figure 9.15 Checking Hamming Parity 192
Figure 10.1 Example Timing Diagram 195
Figure 10.2 Example Propagation Delay 196
Figure 10.3 Capacitor Charge and Discharge 197
Figure 10.4 SR Latch Using NAND Gates 198
Figure 10.5 SR Latch Timing Diagram 199
Figure 10.6 SR Latch 199
Figure 10.7 D Flip-Flop Using SR Flip-Flop 200
Figure 10.8 D Flip-Flop 200
Figure 10.9 D Latch Timing Diagram 200
Figure 10.10 JK Flip-Flop 201
Figure 10.11 JK Flip-Flop Timing Diagram 201
Figure 10.12 T Flip-Flop 202
Figure 10.13 Toggle Flip-Flop Timing Diagram 202
Figure 10.14 Master-Slave Flip-Flop 203
Figure 10.15 4-Bit Register 204
Figure 10.16 Shift Register 205
Figure 11.1 Asynchronous 2-Bit Counter 208
Figure 11.2 Asynchronous 3-Bit Counter 208
Figure 11.3 Asynchronous 4-Bit Counter 209
Figure 11.4 4-Bit Asynchronous Counter Timing Diagram 209
Figure 11.5 Synchronous 2-Bit Counter 210
Figure 11.6 Synchronous 3-Bit Counter 210

Figure 11.7 Synchronous 4-Bit Up Counter 211
Figure 11.8 4-Bit Synchronous Counter Timing Diagram 211
Figure 11.9 Synchronous 4-Bit Down Counter 212
Figure 11.10 4-Bit Synchronous Down Counter Timing Diagram 212
Figure 11.11 4-Bit Ring Counter 213
Figure 11.12 4-Bit Ring Counter Timing Diagram 213
Figure 11.13 4-Bit Johnson Counter 214
Figure 11.14 4-Bit Johnson Counter Timing Diagram 214
Figure 11.15 Decade Counter 215
Figure 11.16 4-Bit Decade Counter Timing Diagram 215
Figure 11.17 Up-Down Counter 216
Figure 11.18 Synchronous 2-Bit Counter 216
Figure 11.19 Frequency Divider 217
Figure 12.1 Moore Vending Machine FSM 220
Figure 12.2 Mealy Vending Machine FSM 221
Figure 13.1 Elevator 223
Figure 14.1 Simple Data Flow Control Circuit 226
Figure 14.2 Simplified CPU Block Diagram 226
Figure 14.3 CPU State Diagram 231

LIST OF TABLES

Table 2.1 Roman Numerals 13
Table 2.2 Binary-Decimal Conversion 15
Table 2.3 Hexadecimal Numbers 16
Table 2.4 Counting To Twenty 17
Table 2.5 Expanding a Decimal Number 19
Table 2.6 Binary-Octal Conversion Examples 22
Table 2.7 Binary-Hexadecimal Conversion Examples 24
Table 2.8 Decimal to Binary 26
Table 2.9 Decimal to Octal 27
Table 2.10 Decimal to Hexadecimal 27
Table 2.11 Decimal to Binary Fraction 28
Table 2.12 Decimal to Binary Fraction Example 28
Table 2.13 Decimal to Long Binary Fraction 29
Table 2.14 Decimal to Binary Mixed Integer 29
Table 2.15 Decimal to Binary Mixed Fraction 30
Table 2.16 Practice Problems 31
Table 2.17 Floating Point Examples 33
Table 3.1 Addition Table 35
Table 3.2 Binary Addition Problems 37
Table 3.3 Binary Subtraction Problems 38

Table 3.4 Ones Complement 40
Table 3.5 Twos Complement 41
Table 3.6 Example Twos Complement 42
Table 3.7 Binary Multiplication Table 47
Table 3.8 ASCII Table 51
Table 3.9 ASCII Symbols 51
Table 3.10 ASCII Practice 52
Table 3.11 BCD Systems 53
Table 3.12 Nines Complement for 127 55
Table 3.13 BCD Practice 55
Table 3.14 Gray Codes 62
Table 4.1 Truth Table for AND 65
Table 4.2 Truth Table for OR 68
Table 4.3 Truth Table for NOT 69
Table 4.4 Truth Table for NAND Gate 70
Table 4.5 Truth Table for NOR 71
Table 4.6 Truth Table for XOR 71
Table 4.7 Truth Table for XNOR 72
Table 4.8 Truth Table for a Buffer 73
Table 4.9 Truth Table for Absorption Property 82
Table 4.10 Truth Table for Generic Circuit One 87
Table 4.11 Truth Table for Generic Circuit Two 88
Table 4.12 Boolean Function Six 88
Table 4.13 Boolean Functions 89
Table 5.1 Truth Table for Example 93
Table 5.2 Truth Table for First Minterm Example 95
Table 5.3 Truth Table for Second Minterm Example 96
Table 5.4 Truth Table for First Maxterm Example 97
Table 5.5 Truth Table for Second Maxterm Example 98
Table 5.6 Minterm and Maxterm Relationships 99
Table 5.7 Minterm-Maxterm Relationships 101
Table 5.8 Truth Table for SOP Example 102
Table 5.9 Truth Table for POS Example 103
Table 5.10 Canonical Example Truth Table 104
Table 5.11 Truth Table for Standard Form Equation 105
Table 5.12 Truth Table for Standard Form Equation 107
Table 5.13 Canonical Form Practice Problems 109
Table 5.14 Simplifying Boolean Expressions 112
Table 6.1 Circuit Simplification Methods 113
Table 6.2 Truth Table for Simple Circuit 114
Table 6.3 Truth Table with Greek Letters 115
Table 6.4 Truth Table for Two-Input Circuit 115
Table 6.5 Truth Table for Three-Input Circuit 116
Table 6.6 Truth Table for Four-Input Circuit 117
Table 6.7 Karnaugh maps Practice Problems 132
Table 6.8 Truth Table for Checkerboard Pattern 133

Table 7.1 Quine-McCluskey Ex 1: Minterm Table 142
Table 7.2 Quine-McCluskey Ex 1: Rearranged Table 142
Table 7.3 Quine-McCluskey Ex 1: Size 2 Implicants 143
Table 7.4 Quine-McCluskey Ex 1: Size 4 Implicants 144
Table 7.5 Quine-McCluskey Ex 1: Prime Implicants 144
Table 7.6 Quine-McCluskey Ex 1: 1st Iteration 145
Table 7.7 Quine-McCluskey Ex 1: 2nd Iteration 145
Table 7.8 Quine-McCluskey Ex 1: 3rd Iteration 146
Table 7.9 Quine-McCluskey Ex 2: Minterm Table 147
Table 7.10 Quine-McCluskey Ex 2: Rearranged Table 148
Table 7.11 Quine-McCluskey Ex 2: Size Two Implicants 149
Table 7.12 Quine-McCluskey Ex 2: Size 4 Implicants 150
Table 7.13 Quine-McCluskey Ex 2: Prime Implicants 151
Table 7.14 Quine-McCluskey Ex 2: 1st Iteration 151
Table 7.15 Quine-McCluskey Ex 2: 2nd Iteration 152
Table 7.16 Quine-McCluskey Practice Problems 153
Table 8.1 Truth Table for Half-Adder 158
Table 8.2 Truth Table for Full Adder 159
Table 8.3 Truth Table for Half-Subtractor 162
Table 8.4 Truth Table for Subtractor 163
Table 8.5 One-Bit Comparator Functions 168
Table 8.6 Truth Table for Two-Bit Comparator 169
Table 9.1 Truth Table for a Multiplexer 175
Table 9.2 Truth Table for a Demultiplexer 176
Table 9.3 Truth Table for Priority Encoder 179
Table 9.4 Truth Table for Seven-Segment Display 180
Table 9.5 Even Parity Examples 183
Table 9.6 Iterative Parity 185
Table 9.7 Iterative Parity With Error 185
Table 9.8 Hamming Parity Bits 187
Table 9.9 Hamming Parity Cover Table 187
Table 9.10 Hamming Example - Iteration 1 188
Table 9.11 Hamming Example - Iteration 2 188
Table 9.12 Hamming Example - Iteration 3 188
Table 9.13 Hamming Example - Iteration 4 188
Table 9.14 Hamming Example - Iteration 5 189
Table 9.15 Hamming Example - Iteration 6 189
Table 9.16 Hamming Parity Cover Table Reproduced 191
Table 9.17 Hamming Parity Examples 193
Table 9.18 Hamming Parity Errors 193
Table 10.1 Truth Table for SR Latch 198
Table 10.2 JK Flip-Flop Timing Table 201
Table 11.1 Counter IC's 217
Table 12.1 Crosswalk State Table 222
Table a.1 Univariate Properties 235
Table a.2 Multivariate Properties 236

Table a.3 Boolean Functions 236

LISTINGS

ACRONYMS

Part I

THEORY

DIGITAL LOGIC, as most computer science studies, depends on a foundation of theory. This part of the book concerns the theory of digital logic and includes binary mathematics, gate-level logic, Boolean algebra, and simplifying Boolean expressions. An understanding of these foundational conceptsis essential before attempting to design complex combinational and sequential logic circuits.

1

INTRODUCTION

1.1 PREFACE

1.1.1 *Introduction to the Study of Digital Logic*

Digital logic is the study of how electronic devices make decisions. It functions at the lowest level of computer operations: bits that can either be "on" or "off" and groups of bits that form "bytes" and "words" that control physical devices. The language of digital logic is Boolean algebra, which is a mathematical model used to describe the logical function of a circuit; and that model can then be used to design the most efficient device possible. Finally, various simple devices, such as adders and registers, can be combined into increasingly complex circuits designed to accomplish advanced decision-making tasks.

1.1.2 *Introduction to the Author*

I have worked with computers and computer controlled systems for more than 30 years. I took my first programming class in 1976; and, several years later, was introduced to digital logic while taking classes to learn how to repair computer systems. For many years, my profession was to work on computer systems, both as a repair technician and a programmer, where I used the principles of digital logic daily. I then began teaching digital logic classes at Cochise College and was able to share my enthusiasm for the subject with Computer Information Systems students. Over the years, I have continued my studies of digital logic in order to improve my understanding of the topic; I also enjoy building logic circuits on a simulator to solve interesting challenges. It is my goal to make digital logic understandable and to also ignite a lifelong passion for the subject in students.

1.1.3 *Introduction to This Book*

This book has two goals:

1. AUDIENCE. Many, perhaps most, digital logic books are designed for third or fourth year electronics engineering or computer science students and presume a background that includes advanced mathematics and various engineering classes. For example, it is possible to find a digital logic book that discusses topics like physically building circuits from discrete components and then calculating the heat rise of those circuits while operating at maximum capacity. This book,

though, was written for students in their second year of a Computer Information Systems program and makes no assumptions about prior mathematics and engineering classes.

2. Cost. Most digital logic books are priced at $150 (and up) but this book is published under a Creative Commons license and, though only a tiny drop in the proverbial textbook ocean, is hoped to keep the cost of books for at least one class as low as possible.

Following are the features for the various editions of this book:

1. 2012. Prior to 2012, handouts were given to students as they were needed during class; however, it was in this year that the numerous disparate documents were assembled into a cohesive book and printed by a professional printing company.

2. 2013. A number of complex circuits were added to the book, including a Hamming Code generator/checker, which is used for error detection, and a Central Processing Unit (CPU) using discrete logic gates.

3. 2014. New material on Mealy and Moore State Machines was included, but the major change was in the laboratory exercises where five Verilog labs were added to the ten gate-level simulation labs.

4. 2015. New information was added about adding/subtracting Binary Coded Decimal (BCD) numbers and representing floating point numbers in binary form; and all of the laboratory exercises were re-written in Verilog. Also, the book was reorganized and converted to LaTeXfor printing.

5. 2018. The labs were re-written using *Logisim-evolution* because students find that system easier to understand than iVerilog.

This book was written with LaTeX using TeXstudio. The source for this book is available at GITHUB, http://bit.ly/2w6qU2C, and anyone is welcomed to fork the book and develop their own version.

DISCLAIMER

I wrote, edited, illustrated, and published this book myself. While I did the best that I could, there are, no doubt, errors. I apologize in advance if anything presented here is factually erroneous; I'll correct those errors as soon as they are discovered. I'll also correct whatever typos I overlooked, despite TeXstudio's red squiggly underlines trying to tell me to check my work. –George Self

1.1.4 *About the Creative Commons License*

This book is being released under the Creative Commons 0 license, which is the same as public domain. That permits people to share, remix, or even rewrite the work so it can be used to help educate students wherever they are studying digital logic.

1.2 ABOUT DIGITAL LOGIC

1.2.1 *Introduction*

Digital logic is the study of how logic is used in digital devices to complete tasks in fields as diverse as communication, business, space exploration, and medicine (not to mention everyday life). This definition has two main components: logic and digital. *Logic* is the branch of philosophy that concerns making reasonable judgment based on sound principles of inference. It is a method of problem solving based on a linear, step-by-step procedure. *Digital* is a system of mathematics where only two possible values exist: *True* and *False* (usually represented by 1 and 0). While this approach may seem limited, it actually works quite nicely in computer circuits where *True* and *False* can be easily represented by the presence or absence of voltage.

Digital logic is not the same as programming logic, though there is some relationship between them. A programmer would use the constructs of logic within a high-level language, like Java or C++, to get a computer to complete some task. On the other hand, an engineer would use digital logic with hardware devices to build a machine, like an alarm clock or calculator, which executes some task. In broad strokes, then, programming logic concerns writing software while digital logic concerns building hardware.

Digital logic may be divided into two broad classes:

1. Combinational Logic, in which the outputs are determined solely by the input states at one particular moment with no memory of prior states. An example of a combinational circuit is a simple adder where the output is determined only by the values of two inputs.

2. Sequential Logic, in which the outputs depend on both current and prior inputs, so some sort of memory is necessary. An example of a sequential circuit is a counter where a sensed new event is added to some total contained in memory.

Both combinational and sequential logic are developed in this book, along with complex circuits that require both combinational and sequential logic.

1.2.2 *A Brief Electronics Primer*

Electricity is nothing more than the flow of electrons from point *A* to point *B*. Along the way, those electrons can be made to do work as they flow through various devices. Electronics is the science (and, occasionally, art) of using tiny quantities of electricity to do work. As an example, consider the circuit schematic illustrated in Figure 1.1:

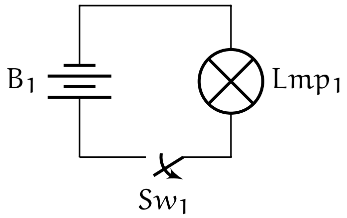


Figure 1.1: Simple Lamp Circuit

In this diagram, battery *B1* is connected to lamp *Lmp1* through switch *Sw1*. When the switch is closed, electrons will flow from the negative battery terminal, through the switch and lamp, and back to the positive battery terminal. As the electrons flow through the lamp's filament it will heat up and glow: light!

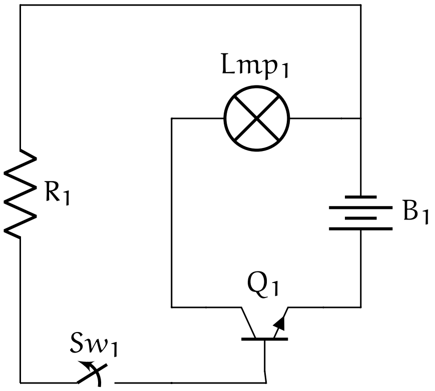


Figure 1.2: Transistor-Controlled Lamp Circuit

A slightly more complex circuit is illustrated in Figure 1.2. In this case, a transistor, *Q1*, has been added to the circuit. When switch *Sw1* is closed, a tiny current can flow from the battery, through the transistor's emitter (the connection with the arrow) to its base, through the switch, through a resistor *R1*, and back to the positive terminal of the battery. However, that small current turns on transistor *Q1* so a much larger current can also flow from the battery, through the collector port, to lamp *Lmp1*, and back to the battery. The final effect is the same for both circuits: close a switch and the lamp turns on. However, by using a transistor, the lamp can be controlled by applying any sort of positive voltage to the transistor's base; so the lamp could be controlled by a switch, as illustrated, or by the output of some other electronic process, like a photo-electric cell sensing that the room is dark.

Using various electronic components, like transistors and resistors, digital logic "gates" can be constructed, and these become the building blocks for complex logic circuits. Logic gates are more thoroughly covered in a later chapter, but one of the fundamental logic gates is an OR gate, and a simplified schematic diagram for an OR gate is in Figure 1.3. In this circuit, any voltage present at *Input A* or *Input B* will activate the transistors and develop voltage at the output.

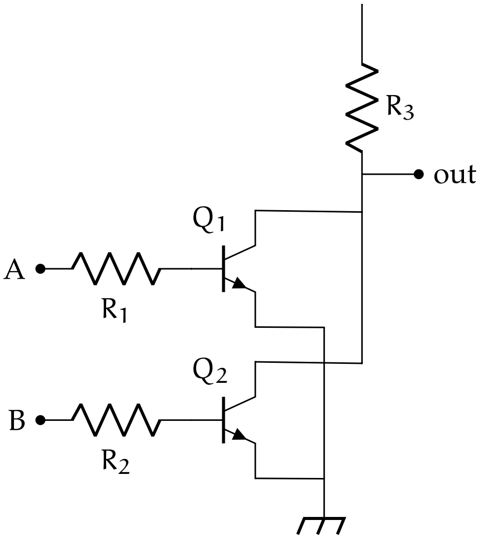


Figure 1.3: Simple OR Gate Using Transistors

Figure 1.4[1] shows a more complete OR gate created with what are called "N-Channel metal-oxide semiconductor" (or *nMOS*) transistors. The electronic functioning of this circuit is beyond the scope of this book (and is in the domain of electronics engineers), but the important point to keep in mind is that this book concerns building electronic circuits designed to accomplish a physical task rather than write a program to control a computer's processes.

Early electronic switches took the form of vacuum tubes, but those were replaced by transistors which were much more energy efficient and physically smaller. Eventually, entire transistorized circuits, like the OR gate illustrated in Figure 1.4, were miniaturized and placed in a single Integrated Circuit (IC), sometimes called a "chip," smaller than a postage stamp.

ICs make it possible to produce smaller, faster, and more powerful electronics devices. For example, the original ENIAC computer, built in 1946, occupied more than 1800 square feet of floor space and required 150KW of electricity, but by the 1980s integrated circuits in hand-held

---

[1] This schematic diagram was created by Ramón Jaramillo and found at http://www.texample.net/tikz/examples/or-gate/

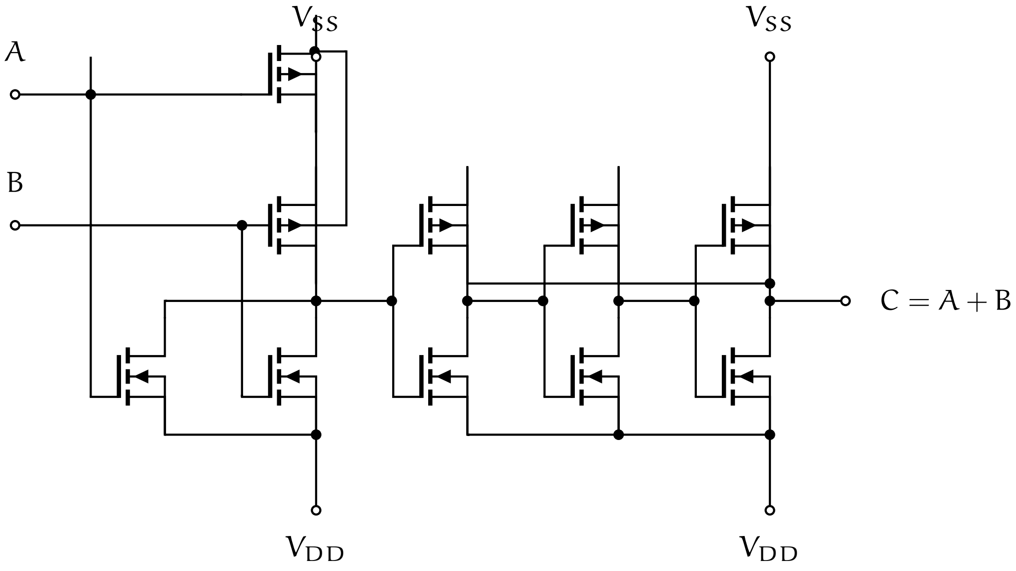


Figure 1.4: OR Gate

calculators were more powerful than that early computer. Integrated circuits are often divided into four classes:

1. Small-scale integration with fewer than 10 transistors
2. Medium-scale integration with 10-500 transistors
3. Large-scale integration with 500-20,000 transistors
4. Very large-scale integration with 20,000-1,000,000 transistors

Integrated circuits are designed by engineers who use software written specifically for that purpose. While the intent of this book is to afford students a foundation in digital logic, those who pursue a degree in electronics engineering, software engineering, or some related field, will need to study a digital logic language, like iVerilog.

1.3 BOOLEAN ALGEBRA

1.3.1 *History*

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

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

All arithmetic operations performed with Boolean quantities have but one of two possible outcomes: 1 or 0. There is no such thing as 2 or -1 or $\frac{1}{3}$ in the Boolean world and numbers other than 1 and 0 are invalid by definition. Claude Shannon of MIT recognized how Boolean algebra could be applied to on-and-off circuits, where all signals are characterized as either *high* (1) or *low* (0), and his 1938 thesis, titled *A Symbolic Analysis of Relay and Switching Circuits*, put Boole's theoretical work to use in land line telephone switching, a system Boole never could have imagined.

While there are a number of similarities between Boolean algebra and real-number algebra, it is important to bear in mind that the system of numbers defining Boolean algebra is severely limited in scope: 1 and 0. Consequently, the "Laws" of Boolean algebra often differ from the "Laws" of real-number algebra, making possible such Boolean statements as $1 + 1 = 1$, which would be considered absurd for real-number algebra.

1.3.2 *Boolean Equations*

Boolean algebra is a mathematical system that defines a series of logical operations performed on a set of variables. The expression of a single logical function is a *Boolean Equation* that uses standardized symbols and rules. Boolean expressions are the foundation of digital circuits.

Sometimes Boolean Equations are called SWITCHING EQUATIONS

Binary variables used in Boolean algebra are like variables in regular algebra except that they can only have two values: one or zero. Boolean algebra includes three primary logical functions: AND, OR, and NOT; and five secondary logical functions: NAND, NOR, XOR, and XNOR, and Buffer. A Boolean equation defines an electronic circuit that provides a relationship between input and output variables and takes the form of:

XNOR is sometimes called EQUIVALENCE *and Buffer is sometimes called* TRANSFER.

$$C = A * B \tag{1.1}$$

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

In Boolean algebra, it is common to speak of *truth*. This term does not mean the same as it would to a philosopher, though its use is based on Aristotelian philosophy where a statement was either *True* or

False. In Boolean algebra as used in electronics, *True* commonly means "voltage present" (or "1") while *False* commonly means "voltage absent" (or "0"), and this can be applied to either input or output variables. It is common to create a *Truth Table* for a Boolean equation to indicate which combination of inputs should evaluate to a *True* output and Truth Tables are used very frequently throughout this book.

1.4 ABOUT THIS BOOK

This book is organized into two main parts:

1. THEORY. Chapters two through six cover the foundational theory of digital logic. Included are chapters on binary mathematics, Boolean algebra, and simplifying Boolean expressions using tools like Karnaugh maps and the Quine-McCluskey method.

2. PRACTICE. Chapters seven through nine expand the theory of digital logic into practical applications. Covered are combinational and sequential logic circuits, and then simulation of various physical devices, like elevators.

There is also an accompanying lab manual where *Logisim-evolution* is used to build digital logic circuits. By combining the theory presented in this book along with the practical application presented in the lab manual it is hoped that students gain a thorough understanding of digital logic.

By combining the theoretical background of binary mathematics and Boolean algebra with the practical application of building logic devices, digital logic becomes understandable and useful.

2

FOUNDATIONS OF BINARY MATHEMATICS

2.1 INTRODUCTION TO NUMBER SYSTEMS

2.1.1 *Background*

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

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

- Whole Numbers: $1, 2, 3, 4, 5, 6, 7, 8, 9...$
- Integers: $-4, -3, -2, -1, 0, 1, 2, 3, 4...$
- Rational Numbers: $-5.3, 0, \frac{1}{3}, 6.7$
- Irrational Numbers: π (approx. 3.1416), e (approx. 2.7183), and the square root of any prime number
- Real Numbers: (combination of all rational and irrational numbers)
- Complex Numbers: $3 - j4$

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

- Whole numbers work well for counting discrete objects, such as the number of resistors in a circuit.
- Integers are needed to express a negative voltage or current.
- Irrational numbers are used to describe the charge/discharge cycle of electronic objects like capacitors.

- Real numbers, in either fractional or decimal form, are used to express the non-integer quantities of voltage, current, and resistance in circuits.
- Complex numbers, in either rectangular or polar form, must be used rather than real numbers to capture the dual essence of the magnitude and phase angle of the current and voltage in alternating current circuits.

There is a difference between the concept of a "number" as a measure of some quantity and "number" as a means used to express that quantity in spoken or written communication. A way to symbolically denote numbers had to be developed in order to use them to describe processes in the physical world, make scientific predictions, or balance a checkbook. The written symbol that represents some number, like how many apples there are in a bin, is called a *cipher* and in western mathematics, the commonly-used ciphers are $0, 1, 2, 3, 4, 5, 6, 7, 8$, and 9.

2.1.2 *Binary Mathematics*

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

2.1.3 *Systems Without Place Value*

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

ROMAN NUMERALS. The Romans devised a system that was a substantial improvement over hash marks, because it used a variety of ciphers to represent increasingly large quantities. The notation for one is the capital letter I. The notation for 5 is the capital letter V. Other ciphers, as listed in Table 2.1, possess increasing values:

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

Table 2.1: Roman Numerals

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

$$(\mathrm{M} = 1000) + (\mathrm{CM} = 900) + (\mathrm{LXXX} = 80) + (\mathrm{VII} = 7)$$

Large numbers are very difficult to denote with Roman numerals; and the left vs. right (or subtraction vs. addition) of values can be very confusing. Adding and subtracting two Roman numerals is also very challenging, to say the least. Finally, one other major problem with this system is that there is no provision for representing the number zero or negative numbers, and both are very important concepts in mathematics. Roman culture, however, was more pragmatic with respect to mathematics than most, choosing only to develop their numeration system as far as it was necessary for use in daily life.

2.1.4 *Systems With Place Value*

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

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

$$1206 = (1X1000) + (2X100) + (0X10) + (6X1)$$

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

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

BINARY NUMERATION. The binary numeration system uses only two ciphers and the weight for each place in a binary number is two times as much as the place to its right. Contrast this to the decimal numeration system that has ten different ciphers and the weight for each place is ten times the place to its right. The two ciphers for the binary system are zero and one, and these ciphers are arranged right-to-left in a binary number, each place doubling the weight of the previous place. The rightmost place is the "ones" place; and, moving to the left, is the "twos" place, the "fours" place, the "eights" place, the "sixteens" place, and so forth. For example, the binary number 11010 can be expressed as a sum of each cipher value times its respective weight:

$$11010 = (1X16) + (1X8) + (0X4) + (1X2) + (0X1)$$

The primary reason that the binary system is popular in modern electronics is because it is easy to represent the two cipher states (zero and one) electronically; if no current is flowing in the circuit it represents a binary zero while flowing current represents a binary one. Binary numeration also lends itself to the storage and retrieval of numerical information: as examples, magnetic tapes have spots of iron oxide that are magnetized for a binary one or demagnetized for a binary zero and optical disks have a laser-burned pit in the aluminum substrate representing a binary one and an unburned spot representing a binary zero.

Digital numbers require so many bits to represent relatively small numbers that programming or analyzing electronic circuitry can be a tedious task. However, anyone working with digital devices soon learns to quickly count in binary to at least 11111 (that is decimal 31). Any time spent practicing counting both up and down between zero and 11111 will be rewarded while studying binary mathematics, codes, and other digital logic topics. Table 2.2 will help in memorizing binary numbers:

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

Table 2.2: Binary-Decimal Conversion

OCTAL NUMERATION. The octal numeration system is place-weighted with a base of eight. Valid ciphers include the symbols $0, 1, 2, 3, 4, 5, 6$, and 7. These ciphers are arranged right-to-left in an octal number, each place being eight times the weight of the previous place. For example, the octal number 4270 can be expressed, just like a decimal number, as a sum of each cipher value times its respective weight:

$$4270 = (4X512) + (2X64) + (7X8) + (0X1)$$

HEXADECIMAL NUMERATION. The hexadecimal numeration system is place-weighted with a base of sixteen. There needs to be ciphers for numbers greater than nine so English letters are used for those values. Table 2.3 lists hexadecimal numbers up to decimal 15:

The word "hexadecimal" is a combination of "hex" for six and "decimal" for ten

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

Table 2.3: Hexadecimal Numbers

Hexadecimal ciphers are arranged right-to-left, each place being 16 times the weight of the previous place. For example, the hexadecimal number 13A2 can be expressed, just like a decimal number, as a sum of each cipher value times its respective weight:

$$13A2 = (1X4096) + (3X256) + (AX16) + (2X1)$$

Maximum Number Size

It is important to know the largest number that can be represented with a given number of cipher positions. For example, if only four cipher positions are available then what is the largest number that can be represented in each of the numeration systems? With the crude hash-mark system, the number of places IS the largest number that can be represented, since one hash mark "place" is required for every integer step. For place-weighted systems, however, the answer is found by taking the number base of the numeration system (10 for decimal, 2 for binary) and raising that number to the power of the number of desired places. For example, in the decimal system, a five-place number can represent 10^5, or $100,000$, with values from zero to $99,999$. Eight places in a binary numeration system, or 2^8, can represent 256 different values, $0 - 255$.

2.1.5 *Summary of Numeration Systems*

Table 2.4 counts from zero to twenty using several different numeration systems:

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

Table 2.4: Counting To Twenty

Numbers for Computer Systems

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

---

[a] http://en.wikipedia.org/wiki/Eniac

2.1.6 *Conventions*

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

In this book, subscripts are normally used to make it clear whether the number is binary or some other system.

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

One other commonly used convention for hexadecimal numbers is to add an *h* (for *hexadecimal*) after the number. This is used because that is easier to enter with a keyboard than to use a subscript and is more intuitive than using a 0x prefix. Thus, $1A_{16}$ would be written 1Ah. In this case, the *h* only indicates that the number 1A is hexadecimal; it is not some sort of mathematical operator.

Occasionally binary numbers are written with a 0b prefix; thus 0b1010 would be 1010_2, but this is a programmer's convention not often found elsewhere.

2.2 CONVERTING BETWEEN RADICES

2.2.1 *Introduction*

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

The radix of a system is also commonly called its "base."

2.2.2 *Expanded Positional Notation*

Expanded Positional Notation is a method of representing a number in such a way that each position is identified with both its cipher symbol and its place-value multiplier. For example, consider the number 347_{10}:

$$347_{10} = (3X10^2) + (4X10^1) + (7X10^0) \tag{2.1}$$

The steps to use to expand a decimal number like 347 are found in Table 2.5.

Step	**Result**
Count the number of digits in the number.	Three Digits
Create a series of (_X_) connected by plus signs such that there is one set for each of the digits in the original number.	(_X_) + (_X_) + (_X_)
Fill in the digits of the original number on the left side of each set of parenthesis.	(3X_) + (4X_) + (7X_)
Fill in the radix (or base number) on the right side of each parenthesis.	(3X10) + (4X10) + (7X10)
Starting on the far right side of the expression, add an exponent (power) for each of the base numbers. The powers start at zero and increase to the left.	$(3X10^2) + (4X10^1) + (7X10^0)$

Table 2.5: Expanding a Decimal Number

Additional examples of expanded positional notation are:

$$2413_{10} = (2x10^3) + (4X10^2) + (1X10^1) + (3X10^0) \tag{2.2}$$

$$1052_8 = (1X8^3) + (0X8^2) + (5X8^1) + (2X8^0) \tag{2.3}$$

$$139_{16} = (1X16^2) + (3X16^1) + (9X16^0) \tag{2.4}$$

The above examples are for positive decimal integers; but a number with any radix can also have a fractional part. In that case, the number's integer component is to the left of the radix point (called the "decimal point" in the decimal system), while the fractional part is to the right of the radix point. For example, in the number 139.25_{10}, 139 is the integer component while 25 is the fractional component. If a number includes a fractional component, then the expanded positional notation uses increasingly negative powers of the radix for numbers to the right of the radix point. Consider this binary example: 101.011_2. The expanded positional notation for this number is:

$$101.011_2 = (1X2^2) + (0X2^1) + (1X2^0) + (0X2^{-1}) + (1X2^{-2}) + (1X2^{-3}) \tag{2.5}$$

Other examples are:

$$526.14_{10} = (5X10^2) + (2X10^1) + (6X10^0) + (1X10^{-1}) + (4X10^{-2}) \tag{2.6}$$

$$65.147_8 = (6X8^1) + (5X8^0) + (1X8^{-1}) + (4X8^{-2}) + (7X8^{-3}) \tag{2.7}$$

$$D5.3A_{16} = (13X16^1) + (5X16^0) + (3X16^{-1}) + (10X16^{-2}) \tag{2.8}$$

When a number in expanded positional notation includes one or more negative radix powers, the radix point is assumed to be to the immediate right of the "zero" exponent term, but it is not actually written into the notation. Expanded positional notation is useful in converting a number from one base to another.

2.2.3 *Binary to Decimal*

To convert a number in binary form to decimal, start by writing the binary number in expanded positional notation, calculate the values for each of the sets of parenthesis in decimal, and then add all of the values. For example, convert 1101_2 to decimal:

$$\begin{aligned} 1101_2 &= (1X2^3) + (1X2^2) + (0X2^1) + (1X2^0) \\ &= (8) + (4) + (0) + (1) \\ &= 13_{10} \end{aligned} \tag{2.9}$$

Binary numbers with a fractional component are converted to decimal in exactly the same way, but the fractional parts use negative powers of two. Convert binary 10.11_2 to decimal:

$$\begin{aligned} 10.11_2 &= (1X2^1) + (0X2^0) + (1X2^{-1}) + (1X2^{-2}) \\ &= (2) + (0) + (\frac{1}{2}) + (\frac{1}{4}) \\ &= 2 + .5 + .25 \\ &= 2.75_{10} \end{aligned} \tag{2.10}$$

Most technicians who work with digital circuits learn to quickly convert simple binary integers to decimal in their heads. However, for longer numbers, it may be useful to write down the various place weights and add them up; in other words, a shortcut way of writing expanded positional notation. For example, convert the binary number 11001101_2 to decimal:

```
Binary Number:  1 1 0 0 1 1 0 1
             - - - - - - - -
  (Read Down)  1 6 3 1 8 4 2 1
             2 4 2 6
             8
```

A bit value of one in the original number means that the respective place weight gets added to the total value, while a bit value of zero means that the respective place weight does not get added to the total value. Thus, using the example above this paragraph, the binary number 11001101_2 is converted to: $128 + 64 + 8 + 4 + 1$, or 205_{10}.

Naming Conventions

The bit on the right end of any binary number is the Least Significant Bit (LSB) because it has the least weight (the ones place) while the bit on the left end is the Most Significant Bit (MSB) because it has the greatest weight. Also, groups of bits are normally referred to as *words*, so engineers would speak of 16-bit or 32-bit words. As exceptions, an eight-bit group is commonly called a *byte* and a four-bit group is called a *nibble* (occasionally spelled *nybble*).

2.2.4 *Binary to Octal*

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

"Five Seven Octal" is not pronounced "Fifty Seven" since "fifty" is a decimal number.

Because octal is a base eight system, and eight is 2^3, binary numbers can be converted to octal by creating groups of three and then simplifying each group. As an example, convert 101111_2 to octal:

$$\begin{array}{cc} 101 & 111 \\ 5 & 7 \end{array}$$

Thus, 101111_2 is equal to 57_8.

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

$$\begin{array}{ccc} 011 & 011 & 101 \\ 3 & 3 & 5 \end{array}$$

Thus, 11011101_2 is equal to 335_8.

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

$$\begin{array}{ccccc} 0 & . & 110 & 110 & 100 \\ 0 & . & 6 & 6 & 4 \end{array}$$

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

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

$$\begin{array}{ccccc} 010 & 101 & . & 001 & 010 \\ 2 & 5 & . & 1 & 2 \end{array}$$

Thus, 10101.00101_2 is equal to 25.12_8.

Table 2.6 lists additional examples of binary/octal conversion:

Binary	Octal
100 101.011	45.3
1 100 010.1101	142.64
100 101 011.110 100 1	453.644
1 110 010 011 101.000 110 10	16 235.064
110 011 010 100 111.011 101	63 247.35

Table 2.6: Binary-Octal Conversion Examples

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

2.2.5 *Binary to Hexadecimal*

The hexadecimal numeration system serves as a "shorthand" method of denoting a large binary number. Technicians find it easier to discuss a number like $2F_{16}$ rather than 101111_2. Because hexadecimal is a base 16 system, and 16 is 2^4; binary numbers can be converted to hexadecimal by creating groups of four and then simplifying each group. As an example, convert 10010111_2 to hexadecimal:

$$\begin{array}{cc} 1001 & 0111 \\ 9 & 7 \end{array}$$

Thus, 10010111_2 is equal to 97_{16}.

"Nine Seven Hexadecimal," or, commonly, "Nine Seven Hex," is not pronounced "Ninety Seven" since "ninety" is a decimal number.

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

$$\begin{array}{ccc} 0010 & 0101 & 0110 \\ 2 & 5 & 6 \end{array}$$

Thus, 1001010110_2 is equal to 256_{16}.

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

$$\begin{array}{ccccc} 0 & . & 1001 & 0101 & 1000 \\ & . & 9 & 5 & 8 \end{array}$$

Thus, 0.1001010110_2 is equal to 0.958_{16}.

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

$$\begin{array}{ccccc} 0001 & 1101 & . & 1010 & 1000 \\ 1 & D & . & A & 8 \end{array}$$

Thus, 11101.10101_2 is equal to $1D.A8_{16}$.

Table 2.7 lists additional examples of binary/hexadecimal conversion:

Binary	Hexadecimal
100 101.011	25.6
1 100 010.1101	62.D
100 101 011.110 100 1	12B.D2
1 110 010 011 101.000 110 10	1C9D.1A
110 011 010 100 111.011 101	66A7.74

Table 2.7: Binary-Hexadecimal Conversion Examples

2.2.6 *Octal to Decimal*

The simplest way to convert an octal number to decimal is to write the octal number in expanded positional notation, calculate the values for each of the sets of parenthesis, and then add all of the values. For example, to convert 245_8 to decimal:

$$\begin{aligned} 245_8 &= (2X8^2) + (4X8^1) + (5X8^0) \\ &= (2X64) + (4X8) + (5X1) \\ &= (128) + (32) + (5) \\ &= 165_{10} \end{aligned} \tag{2.11}$$

If the octal number has a fractional component, then that part would be converted using negative powers of eight. As an example, convert 25.71_8 to decimal:

$$\begin{aligned} 25.71_8 &= (2X8^1) + (5X8^0) + (7X8^{-1}) + (1X8^{-2}) \\ &= (2X8) + (5X1) + (7X0.125) + (1X0.015625) \\ &= (16) + (5) + (0.875) + (0.015625) \\ &= 21.890625_{10} \end{aligned} \tag{2.12}$$

Other examples are:

$$\begin{aligned} 42.6_8 &= (4X8^1) + (2X8^0) + (6X8^{-1}) \\ &= (4X8) + (2X1) + (6X0.125) \\ &= (32) + (2) + (0.75) \\ &= 34.75_{10} \end{aligned} \tag{2.13}$$

$$\begin{aligned} 32.54_8 &= (3X8^1) + (2X8^0) + (5X8^{-1}) + (4X8^{-2}) \\ &= (3X8) + (2X1) + (5X0.125) + (4X0.015625) \\ &= (24) + (2) + (0.625) + (0.0625) \\ &= 26.6875_{10} \end{aligned} \tag{2.14}$$

$$\begin{aligned} 436.27_8 &= (4X8^2) + (3X8^1) + (6X8^0) + (2X8^{-1}) + (7X8^{-2}) \\ &= (4X64) + (3X8) + (6X1) + (2X0.125) + (7X0.015625) \\ &= (256) + (24) + (6) + (0.25) + (0.109375) \\ &= 286.359375_{10} \end{aligned} \tag{2.15}$$

2.2.7 *Hexadecimal to Decimal*

The simplest way to convert a hexadecimal number to decimal is to write the hexadecimal number in expanded positional notation, calculate the values for each of the sets of parenthesis, and then add all of the values. For example, to convert $2A6_{16}$ to decimal:

$$\begin{aligned} 2A6_{16} &= (2X16^2) + (AX16^1) + (6X16^0) \\ &= (2X256) + (10X16) + (6X1) \\ &= (512) + (160) + (6) \\ &= 678_{10} \end{aligned} \tag{2.16}$$

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

$$\begin{aligned} 1B.36_{16} &= (1X16^1) + (11X16^0) + (3X16^{-1}) + (6X16^{-2}) \\ &= (16) + (11) + (3X\frac{1}{16}) + (6X\frac{1}{256}) \\ &= 16 + 11 + 0.1875 + 0.0234375 \\ &= 27.2109375_{10} \end{aligned} \tag{2.17}$$

Other examples are:

$$\begin{aligned} A32.1C_{16} &= (AX16^2) + (3X16^1) + (2X16^0) + (1X16^{-1}) + (CX16^{-2}) \\ &= (10X256) + (3X16) + (2X1) + (1X\frac{1}{16}) + (12X\frac{1}{256}) \\ &= 2560 + 48 + 2 + 0.0625 + 0.046875 \\ &= 6300.109375_{10} \end{aligned} \tag{2.18}$$

$$\begin{aligned} 439.A_{16} &= (4X16^2) + (3X16^1) + (9X16^0) + (AX16^{-1}) \\ &= (4X256) + (3X16) + (9X1) + (10X\frac{1}{16}) \\ &= 1024 + 48 + 9 + 0.625 \\ &= 1081.625_{10} \end{aligned} \tag{2.19}$$

2.2.8 *Decimal to Binary*

2.2.8.1 *Integers*

Note: Converting decimal fractions is a bit different and is covered on page 27.

Converting decimal integers to binary (indeed, any other radix) involves repeated cycles of division. In the first cycle of division, the original decimal integer is divided by the base of the target numeration system (binary=2, octal=8, hex=16), and then the whole-number portion of the quotient is divided by the base value again. This process continues until the quotient is less than one. Finally, the binary, octal, or hexadecimal digits are determined by the "remainders" left over at each division step.

After a decimal number is converted to binary it can be easily converted to either octal or hexadecimal.

Table 2.8 shows how to convert 87_{10} to binary by repeatedly dividing 87 by 2 (the radix for binary) until reaching zero. The number in column one is divided by two and that quotient is placed on the next row in column one with the remainder in column two. For example, when 87 is divided by 2, the quotient is 43 with a remainder of one. This division process is continued until the quotient is less than one. When the division process is completed, the binary number is found by using the remainders, *reading from the bottom to top*. Thus 87_{10} is 1010111_2.

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

Table 2.8: Decimal to Binary

This repeat-division technique will also work for numeration systems other than binary. To convert a decimal integer to octal, for example, divide each line by 8; but follow the process as described above. As an example, Table 2.9 shows how to convert 87_{10} to 127_8.

Integer	Remainder
87	
10	7
1	2
0	1

Table 2.9: Decimal to Octal

The same process can be used to convert a decimal integer to hexadecimal; except, of course, the divisor would be 16. Also, some of the remainders could be greater than 10, so these are written as letters. For example, to convert 678_{10} to $2A6_{16}$ use the process illustrated in Table 2.10.

Integer	Remainder
678	
42	6
2	A
0	2

Table 2.10: Decimal to Hexadecimal

2.2.8.2 *Fractions*

Converting decimal fractions to binary is a repeating operation similar to converting decimal integers, but each step repeats multiplication rather than division. To convert 0.8215_{10} to binary, repeatedly multiply the fractional part of the number by two until the fractional part is zero (or whatever degree of precision is desired). As an example, in Table 2.11, the number in column two, 8215, is multiplied by two and the integer part of the product is placed in column one on the next row while the fractional part in column two. Keep in mind that the “Remainder” is a decimal fraction with an assumed leading decimal point. That process continues until the fractional part reaches zero.

Integer	Remainder
	8125
1	625
1	25
0	5
1	0

Table 2.11: Decimal to Binary Fraction

When the multiplication process is completed, the binary number is found by using the integer parts and *reading from the top to the bottom*. Thus 0.8125_{10} is 0.1101_2.

As another example, Table 2.12 converts 0.78125_{10} to 0.11001_2. The solution was carried out to full precision (that is, the last multiplication yielded a fractional part of zero).

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

Table 2.12: Decimal to Binary Fraction Example

Often, a decimal fraction will create a huge binary fraction. In that case, continue the multiplication until the desired number of binary places are achieved. As an example, in Table 2.13, the fraction 0.1437_{10} was converted to binary, but the process stopped after 10 bits.

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

Table 2.13: Decimal to Long Binary Fraction

Thus, $0.1437_{10} = 0.0010010011_2$ (with 10 bits of precision).

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

To calculate this to full precision requires 6617 bits; thus, it is normally wise to specify the desired precision.

2.2.8.3 *Mixed Numbers*

To convert a mixed decimal number (one that contains both an integer and fraction part) to binary, treat each component as a separate problem and then combine the result. As an example, Table 2.14 and Table 2.15 show how to convert 375.125_{10} to 101110111.001_2.

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

Table 2.14: Decimal to Binary Mixed Integer

Integer	Remainder
	125
0	25
0	5
1	0

Table 2.15: Decimal to Binary Mixed Fraction

A similar process could be used to convert decimal numbers into octal or hexadecimal, but those radix numbers would be used instead of two.

2.2.9 *Calculators*

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

Online Conversion Tool

Excel will convert between decimal/binary/octal/hexadecimal integers (including negative integers), but cannot handle fractions; however, the following website has a conversion tool that can convert between common bases, both integer and fraction: http://baseconvert.com/. An added benefit for this site is conversion with twos complement, which is how negative binary numbers are represented and is covered on page 40.

2.2.10 *Practice Problems*

Table 2.16 lists several numbers in decimal, binary, octal, and hexadecimal form. To practice converting between numbers, select a number on any row and then covert it to the other bases.

Decimal	Binary	Octal	Hexadecimal
13.0	1101.0	15.0	D.0
1872.0	11 101 010 000.0	3520.0	750.0
0.0625	0.0001	0.04	0.1
0.457 031 25	0.011 101 01	0.352	0.75
43.125	101 011.001	53.1	2B.2
108.718 75	1 101 100.101 11	154.56	6C.B8

Table 2.16: Practice Problems

2.3 FLOATING POINT NUMBERS

Numbers can take two forms: Fixed Point and Floating Point. A fixed point number generally has no fractional component and is used for integer operations (though it is possible to design a system with a fixed fractional width). On the other hand, a floating point number has a fractional component with a variable number of places.

Before considering how floating point numbers are stored in memory and manipulated, it is important to recall that any number can be represented using *scientific notation*. Thus, 123.45_{10} can be represented as $1.2345X10^{2}$ and 0.0012345_{10} can be represented as $1.2345X10^{-3}$. Numbers in scientific notation with one place to the left of the radix point, as illustrated in the previous sentence, are considered *normalized*. While most people are familiar with normalized decimal numbers, the same process can be used for any other base, including binary. Thus, 1101.101_{2} can be written as $1.101101X2^{3}$. Notice that for normalized binary numbers the radix is two rather than ten since binary is a radix two system, also there is only one bit to the left of the radix point.

IEEE Standard 754 defines Floating Point Numbers

By definition, floating point numbers are stored and manipulated in a computer using a 32-bit word (64 bits for "double precision" numbers). For this discussion, imagine that the number 10010011.0010_{2} is to be stored as a floating point number. That number would first be normalized to $1.00100110010X2^{7}$ and then placed into the floating point format:

x	xxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxx
sign	exponent	mantissa

- **Sign**: The sign field is a single bit that is zero for a positive number and one for a negative number. Since the example number is positive the sign bit is zero.
- **Exponent**: This is an eight-bit field containing the radix's exponent, or 7 in the example. However, the field must be able to contain both positive and negative exponents, so it is offset by

127. The exponent of the example, 7, is stored as $7 + 127$, or 134; therefore, the exponent field contains 10000110.

- Mantissa (sometimes called *significand*): This 23-bit field contains the number that is being stored, or 100100110010 in the example. While it is tempting to just place that entire number in the mantissa field, it is possible to squeeze one more bit of precision from this number with a simple adjustment. A normalized binary number will always have a one to the left of the radix point since in scientific notation a significant bit must appear in that position and one is the only possible significant bit in a binary number. Since the first bit of the stored number is assumed to be one it is dropped. Thus, the mantissa for the example number is 00100110010000000000000.

Here is the example floating point number (the spaces have been added for clarity):

$$10010011.0010 = 0\ 10000110\ 00100110010000000000000$$

A few floating point special cases have been defined:

- zero: The exponent and mantissa are both zero and it does not matter whether the sign bit is one or zero.
- infinity: The exponent is all ones and the mantissa is all zeros. The sign bit is used to represent either positive or negative infinity.
- NaN: The exponent is all ones and the mantissa has at least one one (it does not matter how many or where). NaN is returned as the result of an illegal operation, like an attempted division by zero.

Two specific problems may show up with floating point calculations:

- Overflow. If the result of a floating point operation creates a positive number that is greater than $(2 - 2^{-23})X2^{127}$ it is a positive overflow or a negative number less than $-(2 - 2^{-23})X2^{127}$ it is a negative overflow. These types of numbers cannot be contained in a 32-bit floating point number; however, the designer could opt to increase the circuit to 64-bit numbers (called "double precision" floating point) in order to work with these large numbers.
- Underflow. If the result of a floating point operation creates a positive number that is less than 2^{-149} it is a positive underflow or a negative number greater than -2^{-149} it is a negative underflow. These numbers are vanishingly small and are sometimes

simply rounded to zero. However, in certain applications, such as multiplication problems, even a tiny fraction is important and there are ways to use "denormalized numbers" that will sacrifice precision in order to permit smaller numbers. Of course, the circuit designer can always opt to use 64-bit ("double precision") floating point numbers which would permit negative exponents about twice as large as 32-bit numbers.

Table 2.17 contains a few example floating point numbers.

Decimal	Binary	Normalized	Floating Point
34.5	100010.1	$1.000101X2^{5}$	0 10000100 00010100000000000000000
324.75	101000100.11	$1.0100010011X2^{8}$	0 10000111 01000100010000111101100
-147.25	10010011.01	$1.001001101X2^{7}$	1 10000110 00100110100000000000000
0.0625	0.0001	$1.0X2^{-4}$	0 01111011 00000000000000000000000

Table 2.17: Floating Point Examples

3

BINARY MATHEMATICS OPERATIONS

3.1 BINARY ADDITION

Adding binary numbers is a simple task similar to the longhand addition of decimal numbers. As with decimal numbers, the bits are added one column at a time, from right to left. Unlike decimal addition, there is little to memorize in the way of an "Addition Table," as seen in Table 3.1

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

Table 3.1: Addition Table

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

```
 1001101
+0010010
 1011111
```

When the sum in one column includes a carry out, it is added to the next column to the left (again, like decimal addition). Consider the following examples:

```
  11  1  <--Carry Bits
 1001001
+0011001
 1100010
```

```
    11   <--Carry Bits
 1000111
+0010110
 1011101
```

The "ripple-carry" process is simple for humans to understand, but it causes a significant problem for designers of digital circuits. Consequently, ways were developed to carry a bit to the left in an electronic adder circuit and that is covered in Section 8.2, page 157.

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

```
  111 1    <--Carry Bits
 1010.0100
+0011.1101
 1110.0001
```

3.1.1 *Overflow Error*

One problem circuit designers must consider is a carry out bit in the MSB (left-most bit) in the answer. Consider the following:

```
 11 11    <--Carry Bits
 10101110
+11101101
110011011
```

This example illustrates a significant problem for circuit designers. Suppose the above calculation was done with a circuit that could only accommodate eight data bits. The augend and addend are both eight bits wide, so they are fine; however, the sum is nine bits wide due to the carry out in the MSB. In an eight-bit circuit (that is, a circuit where the devices and data lines can only accommodate eight bits of data), the carry out bit would be dropped since there is not enough room to accommodate it.

The result of a dropped bit cannot be ignored. The example problem above, when calculated in decimal, is $174_{10} + 237_{10} = 411_{10}$. If, though, the MSB carry out is dropped, then the answer becomes 155_{10}, which is, of course, incorrect. This type of error is called an *Overflow Error*, and a circuit designer must find a way to correct overflow. One typical solution is to simply alert the user that there was an overflow error. For example, on a handheld calculator, the display may change to something like -*E*- if there is an error of any sort, including overflow.

3.1.2 *Sample Binary Addition Problems*

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

Augend	Addend	Sum
10 110.0	11 101.0	110 011.0
111 010.0	110 011.0	1 101 101.0
1011.0	111 000.0	1 000 011.0
1 101 001.0	11 010.0	10 000 011.0
1010.111	1100.001	10 111.000
101.01	1001.001	1110.011

Table 3.2: Binary Addition Problems

3.2 BINARY SUBTRACTION

3.2.1 *Simple Manual Subtraction*

Subtracting binary numbers is similar to subtracting decimal numbers and uses the same process children learn in primary school. The minuend and subtrahend are aligned on the radix point, and then columns are subtracted one at a time, starting with the least significant place and moving to the left. If the subtrahend is larger than the minuend for any one column, an amount is "borrowed" from the column to the immediate left. Binary numbers are subtracted in the same way, but it is important to keep in mind that binary numbers have only two possible values: zero and one. Consider the following problem:

```
 10.1
-01.0
 01.1
```

In this problem, the LSB column is $1 - 0$, and that equals one. The middle column, though, is $0 - 1$, and one cannot be subtracted from zero. Therefore, one is borrowed from the most significant bit, so the problem in middle column becomes $10 - 1$. (Note: do not think of this as "ten minus one" - remember that this is binary so this problem is "one-zero minus one," or two minus one in decimal) The middle column is $10 - 1 = 1$, and the MSB column then becomes $0 - 0 = 0$.

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

```
 101101.01
-001110.10
  11110.11
```

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

Table 3.3 includes some subtraction problems for practice:

Minuend	Subtrahend	Difference
1 001 011.0	111 010.0	10 001.0
100 010.0	10 010.0	10 000.0
101 110 110.0	11 001 010.0	10 101 100.0
1 110 101.0	111 010.0	111 011.0
11 011 010.1101	101 101.1	10 101 101.0101
10 101 101.1	1 101 101.101	111 111.111

Table 3.3: Binary Subtraction Problems

3.2.2 *Representing Negative Binary Numbers Using Sign-and-Magnitude*

Sign-and-magnitude was used in early computers since it mimics real number arithmetic, but has been replaced by more efficient negative number systems in modern computers.

Binary numbers, like decimal numbers, can be both positive and negative. While there are several methods of representing negative binary numbers; one of the most intuitive is using *sign-and-magnitude*, which is essentially the same as placing a "–" in front of a decimal number. With the sign-and-magnitude system, the circuit designer simply designates the MSB as the *sign bit* and all others as the magnitude of the number. When the sign bit is one the number is negative, and when it is zero the number is positive. Thus, -5_{10} would be written as 1101_2.

Unfortunately, despite the simplicity of the sign-and-magnitude approach, it is not very practical for binary arithmetic, especially when done by a computer. For instance, negative five (1101_2) cannot be added to any other binary number using standard addition technique since the sign bit would interfere. As a general rule, errors can easily occur when bits are used for any purpose other than standard place-weighted values; for example, 1101_2 could be misinterpreted as the number 13_{10} when, in fact, it is meant to represent -5. To keep things straight, the circuit designer must first decide how many bits are going to be used to represent the largest numbers in the circuit, add one more bit for the sign, and then be sure to never exceed that bit field length in arithmetic operations. For the above example, three data bits plus a sign bit would limit arithmetic operations to numbers from negative seven (1111_2) to positive seven (0111_2), and no more.

This system also has the quaint property of having two values for zero. If using three magnitude bits, these two numbers are both zero: 0000_2 (positive zero) and 1000_2 (negative zero).

3.2.3 *Representing Negative Binary Numbers Using Signed Complements*

3.2.3.1 *About Complementation*

Before discussing negative binary numbers, it is important to understand the concept of complementation. To start, recall that the *radix* (or base) of any number system is the number of ciphers available for counting; the decimal (or base-ten) number system has ten ciphers $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$ while the binary (or base-two) number system has two ciphers $(0, 1)$. By definition, a number plus its complement equals the radix (this is frequently called the *radix complement*). For example, in the decimal system four is the radix complement of six since $4 + 6 = 10$. Another type of complement is the *diminished radix complement*, which is the complement of the radix minus one. For example, in the decimal system six is the diminished radix complement of three since $6 + 3 = 9$ and nine is equal to the radix minus one.

DECIMAL. In the decimal system the radix complement is usually called the *tens* complement since the radix of the decimal system is ten. Thus, the tens complement of eight is two since $8 + 2 = 10$. The diminished radix complement is called the *nines* complement in the decimal system. As an example, the nines complement of decimal eight is one since $8 + 1 = 9$ and nine is the diminished radix of the decimal system.

To find the nines complement for a number larger than one place, the nines complement must be found for each place in the number. For example, to find the nines complement for 538_{10}, find the nines complement for each of those three digits, or 461. The easiest way to find the tens complement for a large decimal number is to first find the nines complement and then add one. For example, the tens complement of 283 is 717, which is calculated by finding the nines complement, 716, and then adding one.

BINARY. Since the radix for a binary number is 10_2, (be careful! this is not ten, it is one-zero in binary) the diminished radix is 1_2. The diminished radix complement is normally called the *ones complement* and is obtained by reversing (or “flipping”) each bit in a binary number; so the ones complement of 1001011_2 is 0110100_2.

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

3.2.3.2 *Signed Complements*

In circuits that use binary mathematics, a circuit designer can opt to use ones complement for negative numbers and designate the most significant bit as the sign bit; and, if so, the other bits are the magnitude of the number. This is similar to the *sign-and-magnitude* system discussed on page 38. By definition, when using ones complement negative numbers, if the most significant bit is zero, then the number is positive and the magnitude of the number is determined by the remaining bits; but if the most significant bit is one, then the number is negative and the magnitude of the number is determined by calculating the ones complement of the number. Thus: $0111_2 = +7_{10}$, and $1000_2 = -7_{10}$ (the ones complement for 1000_2 is 0111_2). Table 3.4 may help to clarify this concept:

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

Table 3.4: Ones Complement

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

To simplify circuit design, a designer can opt to use twos complement negative numbers and designate the most significant bit as the sign bit so the other bits are the number's magnitude. To use twos complement numbers, if the most significant bit is zero, then the number is positive and the magnitude of the number is determined by the remaining bits; but if the most significant bit is one, then the number is negative and the magnitude of the number is determined by taking the twos complement of the number (that is, the ones complement plus one). Thus: $0111 = 7$, and $1001 = -7$ (the ones complement of

1001 is 0110, and $0110 + 1 = 0111$). Table 3.5 may help to clarify this concept:

Decimal	Positive	Negative
0	0000	10000
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8	N/A	1000

Table 3.5: Twos Complement

The twos complement removes that quirk of having two values for zero. Table 3.5 shows that zero is either 0000 or 10000; but since this is a four-bit number the initial one is discarded, leaving 0000 for zero whether the number is positive or negative. Also, 0000 is considered a positive number since the sign bit is zero. Finally, notice that 1000 is -8 (ones complement of 1000 is 0111, and $0111 + 1 = 1000$). This means that binary number systems that use a twos complement method of designating negative numbers will be asymmetrical; running, for example, from -8 to $+7$. A twos complement system still has the same number of positive and negative numbers, but zero is considered positive, not neutral.

Programmers reading this book may have wondered why the maximum/minimum values for various types of variables is asymmetrical.

One other quirk about the twos complement system is that the decimal value of the binary number can be quickly calculated by assuming the sign bit has a negative place value and all other places are added to it. For example, in the negative number 1010_2, if the sign bit is assumed to be worth -8 and the other places are added to that, the result is $-8 + 2$, or -6; and -6 is the value of 1010_2 in a twos complement system.

All modern computer systems use radix (or twos) complements to represent negative numbers.

3.2.3.3 *About Calculating the Twos Complement*

In the above section, the twos (or radix) complement is calculated by finding the ones complement of a number and then adding one. For machines, this is the most efficient method of calculating the twos complement; but there is a method that is much easier for humans to use to find the twos complement of a number. Start with the LSB (the right-most bit) and then read the number from right to left. Look for the first one and then invert every bit to the left of that one. As an example, the twos complement for $1010\underline{10}_2$ is formed by starting with the least significant bit (the zero on the right), and working to the left, looking for the first one, which is in the second place from the right. Then, every bit to the left of that one is inverted, ending with: $0101\underline{10}_2$ (the two LSB are underlined to show that they are the same in both the original and twos complement number).

Table 3.6 displays a few examples:

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

Table 3.6: Example Twos Complement

3.2.4 *Subtracting Using the Diminished Radix Complement*

When thinking about subtraction, it is helpful to remember that $A - B$ is the same as $A + (-B)$. Computers can find the complement of a particular number and add it to another number much faster and easier than attempting to create separate subtraction circuits. Therefore, subtraction is normally carried out by adding the complement of the subtrahend to the minuend.

3.2.4.1 *Decimal*

This method is commonly used by stage performers who can subtract large numbers in their heads. While it seems somewhat convoluted, it is fairly easy to master.

It is possible to subtract two decimal numbers by adding the nines complement, as in the following example:

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

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

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

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

3.2.4.2 *Binary*

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

```
 101001
-011011
```

Add the ones complement of the subtrahend:

```
 101001
+100100
1001101
```

The most significant bit is discarded so the solution has the same number of bits as for the two addends. This leaves 001101_2 and adding one to that number (because the diminished radix is one less than the radix) leaves 1110_2. In decimal, the problem is $41 - 27 = 14$.

Often, diminished radix subtraction circuits are created such that they use *end around* carry bits. In this case, the most significant bit is carried around and added to the final sum. If that bit is one, then that increases the final answer by one, and the answer is a positive number. If, though, the most significant bit is zero, then there is no end around carry so the answer is negative and must be complemented to find the true value. Either way, the correct answer is found.

Here is an example:

```
 0110  (6)
-0010  (2)
```

Solution:

```
 0110  (6)
+1101  (-2 in ones complement)
10011
    1  (End-around carry the MSB)
=0100  (4)
```

Answer: 4 (since there was an end-around carry the solution is a positive number). Here is a second example:

```
 0010  (2)
-0110  (6)
```

Solution:

```
 0010  (2)
+1001  (-6 in ones complement)
 1011  (No end-around carry, so ones complement)
=0100  (-4: no end-around carry so negative answer)
```

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

3.2.5 *Subtracting Using the Radix Complement*

3.2.5.1 *Decimal*

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

```
 735
-142
```

Calculate the tens complement of the subtrahend, 142, by finding the nines complement for each digit and then adding one to the complete number: 858 (that is $9-1$, $9-4$, and $9-2+1$). Then, add that tens complement number to the original minuend:

```
 735
+858
1593
```

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

3.2.5.2 *Binary*

To find the radix (or twos) complement of a binary number, each bit in the number is "flipped" (making the ones complement) and then one is added to the result. Thus, the twos complement of 11010_2 is 00110_2 (or $(00101_2)+1_2$). Just as in decimal, a binary number can be subtracted from another by adding the radix complement of the subtrahend to the minuend. Here's an example:

```
 101001
-011011
```

Add the twos complement of the subtrahend:

```
 101001
+011011
1001110
```

The most significant bit is discarded so the solution has the same number of bits as for the two addends. This leaves 001110_2 (or 14_{10}). Converting all of this to decimal, the original problem is $41 - 27 = 14$.

Here are two worked out examples:

Calculate $0110_2 - 0010_2$ (or $6_{10} - 2_{10}$):

```
 0110  (6)
-0010  (2)
```

Solution:

```
 0110  (6)
+1110  (-2 in twos complement)
10100  (Discard the MSB, the answer is 4)
```

Calculate $0010_2 - 0110_2$ (or $2_{10} - 6_{10}$)

```
 0010  (2)
-0110  (6)
```

Solution:

```
 0010  (2)
+1010  (-6 in twos complement)
 1100
 0100  (Twos complement of the sum, -4)
```

3.2.6 *Overflow*

One caveat with signed binary numbers is that of overflow, where the answer to an addition or subtraction problem exceeds the magnitude which can be represented with the allotted number of bits. Remember that the sign bit is defined as the most significant bit in the number. For example, with a six-bit number, five bits are used for magnitude, so there is a range from 00000_2 to 11111_2, or 0_{10} to 31_{10}. If a sign bit is included, and using twos complement, numbers as high as 011111_2 ($+31_{10}$) or as low as 100000_2 (-32_{10}) are possible. However, an addition problem with two signed six-bit numbers that results in a sum greater than $+31_{10}$ or less than -32_{10} will yield an incorrect answer. As an example, add 17_{10} and 19_{10} with signed six-bit numbers:

```
 010001  (17)
+010011  (19)
 100100
```

The answer (100100_2), interpreted with the most significant bit as a sign, is equal to -28_{10}, not $+36_{10}$ as expected. Obviously, this is not correct. The problem lies in the restrictions of a six-bit number field. Since the true sum (36) exceeds the allowable limit for our designated bit field (five magnitude bits, or $+31$), it produces what is called an overflow error. Simply put, six places is not large enough to represent the correct sum if the MSB is being used as a sign bit, so whatever sum is obtained will be incorrect. A similar error will occur if two negative numbers are added together to produce a sum that is too small for a six-bit binary field. As an example, add -17_{10} and -19_{10}:

```
 -17 = 101111
 -19 = 101101

 101111  (-17)
+101101  (-19)
1011100
```

The solution as shown: 011100_2 = $+28_{10}$. (Remember that the most significant bit is dropped in order for the answer to have the same number of places as the two addends.) The calculated (incorrect) answer for this addition problem is 28 because true sum of $-17 + -19$ was too small to be properly represented with a five bit magnitude field.

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

Add 17 + 19:

```
 0010001  (17)
+0010011  (19)
 0100100  (36)
```

Add (-17) + (-19):

```
 -17 = 1101111
 -19 = 1101101

 1101111  (-17)
+1101101  (-19)
11011100  (-36)
```

The correct answer is only found by using bit fields sufficiently large to handle the magnitude and sign bits in the sum.

3.2.6.1 *Error Detection*

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

The simplest way to detect overflow errors is to check the sign of the sum and compare it to the signs of the addends. Obviously, two positive numbers added together will give a positive sum and two negative numbers added together will give a negative sum. With an overflow error, however, the sign of the sum is always opposite that of the two addends: $(+17) + (+19) = -28$ and $(-17) + (-19) = +28$. By checking the sign bits an overflow error can be detected.

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

3.3 BINARY MULTIPLICATION

3.3.1 *Multiplying Unsigned Numbers*

Multiplying binary numbers is very similar to multiplying decimal numbers. There are only four entries in the Binary Multiplication Table:

0X0 = 0
0X1 = 0
1X0 = 0
1X1 = 1

Table 3.7: Binary Multiplication Table

To multiply two binary numbers, work through the multiplier one number at a time (right-to-left) and if that number is one, then shift left and copy the multiplicand as a partial product; if that number is zero, then shift left but do not copy the multiplicand (zeros can be used as placeholders if desired). When the multiplying is completed add all partial products. This sounds much more complicated than it actually is in practice and is the same process that is used to multiply two decimal numbers. Here is an example problem.

```
    1011  (11)
  X 1101  (13)
    1011
   0000
  1011
 1011
10001111  (143)
```

3.3.2 *Multiplying Signed Numbers*

The simplest method used to multiply two numbers where one or both are negative is to use the same technique that is used for decimal numbers: multiply the two numbers and then determine the sign from the signs of the original numbers: if those signs are the same then the result is positive, if they are different then the result is negative. Multiplication by zero is a special case where the result is always zero.

The multiplication method discussed above works fine for paper-and-pencil; but is not appropriate for designing binary circuits. Unfortunately, the mathematics for binary multiplication using an algorithm that can become an electronic circuit is beyond the scope of this book. Fortunately, though, IC already exist that carry out multiplication of both signed and floating-point numbers, so a circuit designer can use a pre-designed circuit and not worry about the complexity of the multiplication process.

3.4 BINARY DIVISION

Binary division is accomplished by repeated subtraction and a right shift function; the reverse of multiplication. The actual process is rather convoluted and complex and is not covered in this book. Fortunately, though, IC already exist that carry out division of both signed and floating-point numbers, so a circuit designer can use a pre-designed circuit and not worry about the complexity of the division process.

3.5 BITWISE OPERATIONS

It is sometimes desirable to find the value of a given bit in a byte. For example, if the LSB is zero then the number is even, but if it is one then the number is odd. To determine the "evenness" of a number, a bitwise mask is multiplied with the original number. As an example, imagine that it is desired to know if 1001010_2 is even, then:

```
     1001010  <- Original Number
BitX 0000001  <- "Evenness" Mask
     0000000
```

The bits are multiplied one position at a time, from left-to-right. Any time a zero appears in the mask that bit position in the product will be zero since any number multiplied by zero yields zero. When a one appears in the mask, then the bit in the original number will be copied to the solution. In the given example, the zero in the least significant bit of the top number is multiplied with one and the result is zero. If that LSB in the top number had been one then the LSB in the result would have also been one. Therefore, an "even" original number would yield a result of all zeros while an odd number would yield a one.

3.6 CODES

3.6.1 *Introduction*

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

3.6.1.1 *Morse Code*

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

```
-.-. --- -.. . ...   .- .-. .   ..-. ..- -.
 c    o   d  e  s    a   r  e    f    u  n
```

3.6.1.2 *Braille Alphabet*

As one other example of a commonly-used code, in 1834 Louis Braille, at the age of 15, created a code of raised dots that enable blind people to read books. For those interested in this code, the Braille alphabet can be found at http://braillebug.afb.org/braille_print.asp.

3.6.2 *Computer Codes*

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

3.6.2.1 *ASCII*

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

In the early 1960s, computer scientists came up with a code they named ASCII and this is still among the most common ways to encode letters and other symbols for a computer. If the computer program knows that a particular spot in memory contains binary numbers that are actually ASCII-coded letters, it is a fairly easy job to convert those codes to letters for a screen display. For simplicity, ASCII is usually represented by hexadecimal numbers rather than binary. For example, the word *Hello* in ASCII is: 048 065 06C 06C 06F.

ASCII code also has a predictable relationship between letters. For example, capital letters are exactly 20_{16} higher in ASCII than their lower-case version. Thus, to change a letter from lower-case to upper-case, a programmer can add 20_{16} to the ASCII code for the lower-case letter. This can be done in a single processing step by using what is known as a *bit-wise AND* on the bit representing 20_{16} in the ASCII code's binary number.

An ASCII chart using hexadecimal values is presented in Table 3.8. The most significant digit is read across the top row and the least significant digit is read down the left column. For example, the letter *A* is 41_{16} and the number *6* is 36_{16}.

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

Table 3.8: ASCII Table

ASCII 20_{16} is a space character used to separate words in a message and the first two columns of ASCII codes (where the high-order nibble are zero and one) were codes essential for teletype machines, which were common from the 1920s until the 1970s. The meanings of a few of those special codes are given in Table 3.9.

Teletype operators from decades past tell stories of sending 25 or more 07_{16} codes (ring the bell) to a receiving terminal just to irritate another operator in the middle of the night.

NUL	All Zeros (a “null” byte)
SOH	Start of Header
STX	Start of Text
ETX	End of Text
EOT	End of Transmission
ENQ	Enquire (is the remote station on?)
ACK	Acknowledge (the station is on)
BEL	Ring the terminal bell (get the operator’s attention)

Table 3.9: ASCII Symbols

Table 3.10 contains a few phrases in both plain text and ASCII for practice.

Plain Text	ASCII
codes are fun	63 6f 64 65 73 20 61 72 65 20 66 75 6e
This is ASCII	54 68 69 73 20 69 73 20 41 53 43 49 49
365.25 days	33 36 35 2e 32 35 20 64 61 79 73
It's a gr8 day!	49 74 27 73 20 61 20 67 72 38 20 64 61 79 21

Table 3.10: ASCII Practice

While the ASCII code is the most commonly used text representation, it is certainly not the only way to encode words. Another popular code is EBCDIC (pronounced like "Eb See Deck"), which was invented by IBM in 1963 and has been used in most of their computers ever since.

Since the early 2000's, computer programs have begun to use Unicode character sets, which are similar to ASCII but multiple bytes are combined to expand the number of characters available for non-English languages like Cyrillic.

3.6.2.2 *Binary Coded Decimal (BCD)*

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

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

Weighted BCD is a more generalized system where each bit position is assigned a "weight," or value. These types of BCD systems are far more common than non-weighted and are found in all sorts of applications. Weighted BCD codes can be converted to decimal by adding the place value for each position in exactly the same way that Expanded Positional Notation is used for to covert between decimal and binary numbers. As an example, the weights for the Natural BCD system are $8-4-2-1$. (These are the same weights used for binary numbers; thus the name "natural" for this system.) The code 1001_{BCD} is converted to decimal like this:

$$\begin{aligned}1001_{BCD} &= (1X8) + (0X4) + (0X2) + (1X1) \\ &= (8) + (0) + (0) + (1) \\ &= 9_{10}\end{aligned} \tag{3.1}$$

Because there are ten decimal ciphers $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$, it requires four bits to represent all decimal digits; so most BCD code systems are four bits wide. In practice, only a few different weighted BCD code systems are commonly used and the most common are shown in Table 3.11.

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

Table 3.11: BCD Systems

Remember that BCD is a code system, not a number system; so the meaning of each combination of four-bit codes is up to the designer and will not necessarily follow any sort of binary numbering sequence.

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

In each of the BCD code systems in Table 3.11 there are six unused four-bit combinations; for example, in the *Natural* system the unused codes are: $1010, 1011, 1100, 1101, 1110$, and 1111. Thus, any circuit designed to use BCD must include some sort of check to ensure that if unused binary values are accidentally input into a circuit it does not create an undefined outcome.

Normally, two BCD codes, each of which are four bits wide, are packed into an eight-bit byte in order to reduce wasted computer memory. Thus, the packed BCD 01110010 contains two BCD numbers: 72. In fact, a single 32-bit word, which is common in many computers, can contain 8 BCD codes. It is a trivial matter for software to either pack or unpack BCD codes from a longer word.

It is natural to wonder why there are so many different ways to code decimal numbers. Each of the BCD systems shown in Table 3.11

has certain strengths and weaknesses and a circuit designer would choose a specific system based upon those characteristics.

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

```
0011 0111
  3    7
```

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

$$\begin{aligned} 1001_{BCD5421} &= (1X5) + (0X4) + (0X2) + (1X1) \\ &= (5) + (0) + (0) + (1) \\ &= 6_{10} \end{aligned} \tag{3.2}$$

Converting from decimal to BCD is also a rather simple process. Each decimal digit is converted to a four-bit BCD equivalent. In the case of Natural BCD the four-bit code is the binary equivalent to the decimal number, other weighted BCD codes would be converted with a similar process.

```
  2    4    5
0010 0100 0101
```

To convert binary to BCD is no trivial exercise and is best done with an automated process. The normal method used is called the *Shift Left and Add Three* algorithm (or, frequently, *Double-Dabble*). The process involves a number of steps where the binary number is shifted left and occasionally three is added to the resulting shift. Wikipedia (https://en.wikipedia.org/wiki/Double_dabble) has a good explanation of this process, along with some examples.

Converting BCD to any other system (like hexadecimal) is most easily done by first converting to binary and then to whatever base is desired. Unfortunately, converting BCD to binary is not as simple as concatenating two BCD numbers; for example, 01000001 is 41 in BCD, but those two BCD numbers concatenated, 01000001, is 65 in binary. One way to approach this type of problem is to use the reverse of the *Double-Dabble* process: *Shift Right and Subtract Three*. As in converting binary to BCD, this is most easily handled by an automated process.

SELF-COMPLEMENTING. The Excess-3 code (called *Ex3* in the table) is self-complementing; that is, the nines complement of any decimal number is found by complementing each bit in the Ex3 code. As an example, find the nines complement for 127_{10}:

1	127_{10}	Original Number
2	$0100\ 0101\ 1010_{Ex3}$	Convert 127 to Excess 3
3	$1011\ 1010\ 0101_{Ex3}$	Ones Complement
4	872_{10}	Convert to Decimal

Table 3.12: Nines Complement for 127

Thus, 872_{10}, is the nines complement of 127_{10}. It is a powerful feature to be able to find the nines complement of a decimal number by simply complementing each bit of its Ex3 BCD representation.

REFLEXIVE. Some BCD codes exhibit a reflexive property where each of the upper five codes are complementary reflections of the lower five codes. For example, 0111_{Ex3} (4) and 1000_{Ex3} (5) are complements, 0110_{Ex3} (3) and 1001_{Ex3} (6) are complements, and so forth. The reflexive property for the 5421 code is different. Notice that the codes for zero through four are the same as those for five through nine, except for the MSB (zero for the lower codes, one for the upper codes). Thus, 0000_{5421} (zero) is the same as 1000_{5421} (five) except for the first bit, 0001_{5421} (one) is the same as 1001_{5421} (six) except for the first bit, and so forth. Studying Table 3.11 should reveal the various reflexive patterns found in these codes.

PRACTICE. Table 3.13 shows several decimal numbers in various BCD systems which can be used for practice in converting between these number systems.

Dec	8421	2421	Ex3	5421
57	0101 1110	10111 1101	1000 1010	1000 1010
79	0111 1001	1101 1111	1010 1100	1010 1100
28	0010 1000	0010 1110	0101 1011	0010 1011
421	0100 0010 0001	0100 0010 0001	0111 0101 0100	0100 0010 0001
903	1001 0000 0011	1111 0000 0011	1100 0011 0110	1100 0000 0011

Table 3.13: BCD Practice

ADDING BCD NUMBERS. BCD numbers can be added in either of two ways. Probably the simplest is to convert the BCD numbers to binary, add them as binary numbers, and then convert the sum back to BCD. However, it is possible to add two BCD numbers without converting. When two BCD numbers are added such that the result is less than ten, then the addition is the same as for binary numbers:

```
  0101  (5)
 +0010  (2)
  0111  (7)
```

However, four-bit binary numbers greater than 1001_2 (that is: 9_{10}) are invalid BCD codes, so adding two BCD numbers where the result is greater than nine requires a bit more effort:

```
  0111  (7)
 +0101  (5)
  1100  (12 -- not valid in BCD)
```

When the sum is greater than nine, then six must be added to that result since there are six invalid binary codes in BCD.

```
  1100  (12 -- from previous addition)
 +0110  (6)
1 0010  (12 in BCD)
```

When adding two-digit BCD numbers, start with the LSN, the rightmost nibble, then add the nibbles with carry bits from the right. Here are some examples to help clarify this concept:

```
  0101 0010  (52)
 +0011 0110  (36)
  1000 1000  (88 in BCD)

  0101 0010  (52)
 +0101 0110  (56)
  1010 1000  (1010, MSN, invalid BCD code)
 +0110 0000  (Add 6 to invalid code)
1 0000 1000  (108 in BCD)

  0101 0101  (55)
 +0101 0110  (56)
  1010 1011  (both nibbles invalid BCD code)
 +0000 0110  (Add 6 to LSN)
  1011 0001  (1 carried over to MSN)
 +0110 0000  (Add 6 to MSN)
1 0001 0001  (111 in BSD)
```

NEGATIVE NUMBERS. BCD codes do not have any way to store negative numbers, so a sign nibble must be used. One approach to this problem is to use a sign-and-magnitude value where a sign nibble is prefixed onto the BCD value. By convention, a sign nibble of 0000 makes the BCD number positive while 1001 makes it negative. Thus, the BCD number 000000100111 is 27, but 100100100111 is -27.

A more mathematically rigorous, and useful, method of indicating negative BCD numbers is use the tens complement of the BCD number since BCD is a code for decimal numbers, exactly like the twos complement is used for binary numbers. It may be useful to review Section 3.2.3.1 on page 39 for information about the tens complement. In the *Natural BCD* system the tens complement is found by adding one to the nines complement, which is found by subtracting each digit of the original BCD number from nine. Here are some examples to clarify this concept:

```
0111 (7 in BCD)
0010 (2, the 9's complement of 7 since 9-7=2)
0011 (3, the 10's complement of 7, or 2+1)

0010 0100 (24 in BCD)
0111 0101 (75, the 9's complement of 24)
0111 0110 (76, the 10's complement of 24)
```

Also, some BCD code systems are designed to easily create the tens complement of a number. For example, in the 2421 BCD system the tens complement is found by nothing more than inverting the MSB. Thus, three is the tens complement of seven and in the 2421 BCD system $0011_{BCD2421}$ is the tens complement of $1101_{BCD2421}$, a difference of only the MSB. Therefore, designers creating circuits that must work with negative BCD numbers may opt to use the 2421 BCD system.

SUBTRACTING BCD NUMBERS. A BCD number can be subtracted from another by changing it to a negative number and adding. Just like in decimal, $5 - 2$ is the same as $5 + (-2)$. Either a nines or tens complement can be used to change a BCD number to its negative, but for this book, the tens complement will be used. If there is a carry-out bit then it can be ignored and the result is positive, but if there is no carry-out bit then answer is negative so the magnitude must be found by finding the tens complement of the calculated sum. Compare this process with subtracting regular binary numbers. Here are a few examples:

```
7 - 3 = 4

    0111  (7 in BCD)
   +0111  (add the 10's complement of 3)
    1110  (invalid BCD code)
   +0110  (add 6 to invalid BCD code)
  1 0100  (4 - drop the carry bit)
```

```
7 - 9 = -2

    0111  (7 in BCD)
   +0001  (10's complement of 9)
    1000  (valid BCD code)
    0010  (10's complement)
```

```
32 - 15 = 17

    0011 0010  (32 in BCD)
   +1000 0101  (10's complement of 15)
    1011 0111  (MSB is invalid BCD code)
   +0110 0000  (add 6 to MSB)
  1 0001 0111  (17, drop the carry bit)
```

```
427 - 640 = -213

    0100 0010 0111  (427 in BCD)
   +0011 0110 0000  (10's complement of 640)
    0111 1000 0111  (no invalid BCD code)
    0010 0001 0011  (10's complement)
```

```
369 - 532 = -163

    0011 0110 1001  (369 in BCD)
   +0100 0110 1000  (10's complement of 532)
    0111 1100 0001  (two invalid BCD codes)
   +0000 0110 0110  (add 6 to invalid codes)
    1000 0011 0111  (sum)
    0001 0110 0011  (10's complement)
```

Here are some notes on the last example: adding the LSN yields $1001 + 1000 = 10001$. The initial one is ignored, but this is an invalid BCD code so this byte needs to be corrected by adding six to it. Then the result of that addition includes an understood carry into the next nibble after the correction is applied. In the same way, the middle nibble was corrected: $1100 + 0110 = 10011$ but the initial one in this answer is carried to the MSN and added there.

3.6.2.3 *Gray Code*

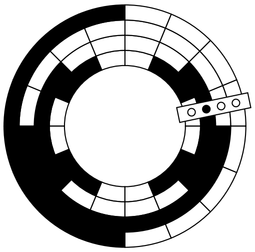


Figure 3.1: Optical Disc

It is often desirable to use a wheel to encode digital input for a circuit. As an example, consider the tuning knob on a radio. The knob is attached to a shaft that has a small, clear disk which is etched with a code, similar to Figure 3.1. As the disk turns, the etched patterns pass or block a laser beam from reaching an optical sensor, and that pass/block pattern is encoded into binary input.

One of the most challenging aspects of using a mechanical device to encode binary is ensuring that the input is stable. As the wheel turns past the light beam, if two of the etched areas change at the same time (thus, changing two bits at once), it is certain that the input will fluctuate between those two values for a tiny, but significant, period of time. For example, imagine that the encoded circuit changes from 1111 to 0000 at one time. Since it is impossible to create a mechanical wheel precise enough to change those bits at exactly the same moment in time (remember that the light sensors will "see" an input several million times a second), as the bits change from 1111 to 0000 they may also change to 1000 or 0100 or any of dozens of other possible combinations for a few microseconds. The entire change may form a pattern like $1111 - 0110 - 0010 - 0000$ and that instability would be enough to create havoc in a digital circuit.

The solution to the stability problem is to etch the disk with a code designed in such a way that only one bit changes at a time. The code used for that task is the Gray code. Additionally, a Gray code is cyclic, so when it reaches its maximum value it can cycle back to its minimum value by changing only a single bit. In Figure 3.1, each of the concentric rings encodes one bit in a four-bit number. Imagine that the disk is rotating past the fixed laser beam reader —the black areas ("blocked light beam") change only one bit at a time, which is characteristic of a Gray code pattern.

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

```
  0
  1
```

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

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

Next, prefix the top half of the group with a zero and the bottom half with a one to get a two-bit Gray code.

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

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

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

Next, prefix the top half of the group with a zero and the bottom half with a one to get a three-bit Gray code.

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

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

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

```
011
001
000
```

Finally, prefix the top half of the group with a zero and the bottom half with a one to get a four-bit Gray code.

```
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000  (four-bit Gray code)
```

The process of reflecting and prefixing can continue indefinitely to create a Gray code of any desired bit length. Of course, Gray code tables are also available in many different bit lengths. Table 3.14 contains a two-bit, three-bit, and four-bit Gray code:

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

Table 3.14: Gray Codes

4

BOOLEAN FUNCTIONS

4.1 INTRODUCTION TO BOOLEAN FUNCTIONS

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

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

$$\begin{aligned}0+0&=0\\0+1&=1\\1+0&=1\\1+1&=1\end{aligned}\tag{4.1}$$

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

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

$$\begin{aligned} 1+1+1&=1 \\ 1+0+0+0+0&=1 \\ 1+1+1+1+1+1&=1 \end{aligned} \tag{4.2}$$

Next, consider a very simple electronic sensor in an automobile: IF the headlights are on AND the driver's door is open THEN a buzzer will sound. In the same way that the plus sign is used to mathematically represent OR , a times sign is used to represent AND . Therefore, using common mathematical symbols, this automobile alarm circuit would be represented by $1X1 = 1$. The following list shows all possible states of the headlights and door:

$$\begin{aligned} 0X0 &= 0 \\ 0X1 &= 0 \\ 1X0 &= 0 \\ 1X1 &= 1 \end{aligned} \tag{4.3}$$

The first row above shows "*False* (the lights are not on) AND *False* (the door is not open) results in *False* (the alarm does not sound)". For AND logic, the only time the output is *True* is when all inputs are also *True*; therefore: $1X1X1X1X0 = 0$. In this way, Boolean AND behaves somewhat like algebraic multiplication.

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

$$A + B = Y \tag{4.4}$$

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

Boolean algebra includes three primary and four secondary logic operations (plus a Buffer, which has no logic value), six univariate, and six multivariate properties. All of these will be explored in this chapter.

4.2 PRIMARY LOGIC OPERATIONS

4.2.1 *AND*

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

$$A \wedge B = Y \tag{4.5}$$

One other method is to use the same symbols that are used for multiplication in traditional algebra; that is, by writing the variables next to each other, with parenthesis, or, sometimes, with an asterisk between them, as in Equation 4.6.

$$\begin{aligned} AB &= Y \\ (A)(B) &= Y \\ A * B &= Y \end{aligned} \tag{4.6}$$

The multiplication symbols X and • (dot) are not commonly used in digital logic equations.

Logic AND is normally represented in equations by using an algebra multiplication symbol since it is easy to type; however, if there is any chance for ambiguity, then the Logic AND symbol ($\wedge$) can be used to differentiate between multiplication and a logic AND function.

Following is the truth table for the AND operator.

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

Table 4.1: Truth Table for AND

A truth table is used to record all possible inputs and the output for each combination of inputs. For example, in the first line of Table 4.1, if input A is 0 and input B is 0 then the output, Y, will be 0. All possible input combinations are normally formed in a truth table by counting in binary starting with all variables having a 0 value to all variables having a 1 value. Thus, in Table 4.1, the inputs are 00, 01, 10, 11. Notice that the output for the AND operator is *False* (that is, 0) until the last row, when both inputs are *True*. Therefore, it could be said that just one *False* input would inactivate a physical AND gate. For that reason, an AND operation is sometimes called an *inhibitor*.

AND Gate Switches

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

Logic gates are realized (or created) in electronic circuits by using transistors, resistors, and other components. These components are normally packaged into a single IC "chip," so the logic circuit designer does not need to know all of the details of the electronics in order to use the gate. In logic diagrams, an AND gate is represented by a shape that looks like a capital D. In Figure 4.1, the input variables A and B are wired to an AND gate and the output from that gate goes to Y.

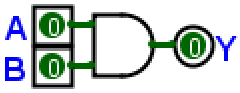


Figure 4.1: AND Gate

Notice that each input and output is named in order to make it easier to describe the circuit algebraically. In reality, AND gates are not packaged or sold one at a time; rather, several would be placed on a single IC, like the *7408 Quad AND Gate*. The designer would design the circuit card to use whichever of the four gates are needed while leaving the unused gates unconnected.

There are two common sets of symbols used to represent the various elements in logic diagrams, and whichever is used is of little consequence since the logic is the same. Shaped symbols, as used in

Figure 4.1, are more common in the United States; but the IEEE has its own symbols which are sometimes used, especially in Europe. Figure 4.2 illustrates the circuit in Figure 4.1 using IEEE symbols:

In this book, only shaped symbols will be used.

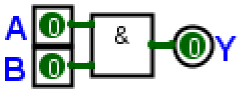


Figure 4.2: AND Gate Using IEEE Symbols

As an example of an AND gate at work, consider an elevator: if the door is closed (logic 1) AND someone in the elevator car presses a floor button (logic 1), THEN the elevator will move (logic 1). If both sensors (door and button) are input to an AND gate, then the elevator motor will only operate if the door is closed AND someone presses a floor button.

4.2.2 *OR*

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

When written in an equation, the Boolean OR term is represented a number of different ways. One method is to use the logic OR symbol, as found in Equation 4.7.

$$A \vee B = Y \tag{4.7}$$

A more common method is to use the *plus* sign that is used for addition in traditional algebra, as in Equation 4.8.

$$A + B = Y \tag{4.8}$$

For simplicity, the mathematical *plus* symbol is normally used to indicate OR in printed material since it is easy to enter with a keyboard; however, if there is any chance for ambiguity, then the logic OR symbol ($\vee$) is used to differentiate between addition and logic OR.

Table 4.2 is the truth table for an OR operation.

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

Table 4.2: Truth Table for OR

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

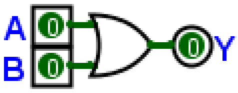


Figure 4.3: OR Gate

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

4.2.3 *NOT*

NOT (or *inverter*) is a Boolean operation that inverts the input. That is, if the input is *True* then the output will be *False* or if the input is *False* then the output will be *True*. When written in an equation, the Boolean NOT operator is represented in many ways, though two are most popular. The older method is to overline (that is, a line above) a term, or group of terms, that are to be inverted, as in Equation 4.9.

Equation 4.9 is read A OR B NOT = Q *(notice that when spoken, the word* not *follows the term that is inverted).*

$$A + \overline{B} = Y \tag{4.9}$$

Another method of indicating NOT is to use the algebra *prime* indicator, an apostrophe, as in Equation 4.10.

$$A + B' = Y \tag{4.10}$$

The reason that NOT is most commonly indicated with an apostrophe is because that is easier to enter on a computer keyboard. There are many other ways authors use to represent NOT in a formula, but none are considered standardized. For example, some authors use an exclamation point: $A+!B = Q$, others use a broken line: $A + \neg B = Q$, others use a backslash: $A + \backslash B = Q$, and still others use a tilde: $A+ \sim B = Q$. However, only the apostrophe and overline are consistently used to indicate NOT. Table 4.3 is the truth table for NOT:

Input	Output
0	1
1	0

Table 4.3: Truth Table for NOT

In a logic diagram, NOT is represented by a small triangle with a "bubble" on the output. In Figure 4.4, the input variable A is inverted by a NOT gate and then sent to output Y.

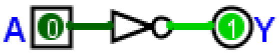


Figure 4.4: NOT Gate

4.3 SECONDARY LOGIC FUNCTIONS

4.3.1 *NAND*

NAND is a Boolean operation that outputs the opposite of AND, that is, NOT AND; thus, it will output a logic *False* only if all of the inputs are *True*. The NAND operation is not often used in Boolean equations, but when necessary it is represented by a vertical line. Equation 4.11 shows a NAND operation.

$$A|B = Y \tag{4.11}$$

Table 4.4 is the Truth Table for a NAND gate.

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

Table 4.4: Truth Table for NAND Gate

In Figure 4.5, the input variables A and B are wired to a NAND gate, and the output from that gate goes to Y.

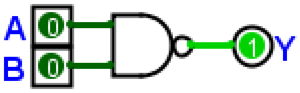


Figure 4.5: NAND Gate

Inverting bubbles are never found by themselves on a wire; they are always associated with either the inputs or output of a logic gate. To invert a signal on a wire, a NOT gate is used.

The logic diagram symbol for a NAND gate looks like an AND gate, but with a small bubble on the output port. A bubble in a logic diagram always represents some sort of signal inversion, and it can appear at the inputs or outputs of nearly any logic gate. For example, the bubble on a NAND gate could be interpreted as "take whatever the output would be generated by an AND gate—then invert it."

4.3.2 *NOR*

NOR is a Boolean operation that is the opposite of OR, that is, NOT OR; thus, it will output a logic *True* only if all of the inputs are *False*. The NOR operation is not often used in Boolean equations, but when necessary it is represented by a downward-pointing arrow. Equation 4.12 shows a NOR operation.

$$A \downarrow B = Y \tag{4.12}$$

Table 4.5 is the truth table for NOR .

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

Table 4.5: Truth Table for NOR

In Figure 4.6, the input variables A and B are wired to a NOR gate, and the output from that gate goes to Y.

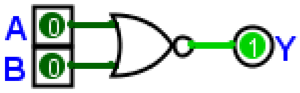


Figure 4.6: NOR Gate

4.3.3 *XOR*

XOR (*Exclusive OR*) is a Boolean operation that outputs a logical one, or *True*, only if the two inputs are different. This is useful for circuits that compare inputs; if they are different then the output is *True*, otherwise it is *False*. Because of this, an XOR gate is sometimes referred to as a *Difference Gate*. The XOR operation is not often used in Boolean equations, but when necessary it is represented by a plus sign (like the OR function) inside a circle. Equation 4.13 shows an XOR operation.

$$A \oplus B = Y \tag{4.13}$$

Table 4.6 is the truth table for an XOR gate.

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

Table 4.6: Truth Table for XOR

In Figure 4.7, the input variables A and B are wired to an XOR gate, and the output from that gate goes to Y.

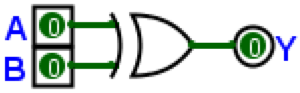


Figure 4.7: XOR Gate

There is some debate about the proper behavior of an XOR gate that has more than two inputs. Some experts believe that an XOR gate should output a *True* if one, and only one, input is *True* regardless of the number of inputs. This would seem to be in keeping with the rules of digital logic developed by George Boole and other early logisticians and is the strict definition of XOR promulgated by the IEEE. This is also the behavior of the XOR gate found in *Logisim-evolution*, the digital logic simulator used in the lab manual accompanying this text. Others believe, though, that an XOR gate should output a *True* if an odd number of inputs is *True*. in *Logisim-evolution* this type of behavior is found in a device called a "parity gate" and is covered in more detail elsewhere in this book.

4.3.4 *XNOR*

XNOR is a Boolean operation that will output a logical one, or *True*, only if the two inputs are the same; thus, an XNOR gate is often referred to as an *Equivalence Gate*. The XNOR operation is not often used in Boolean equations, but when necessary it is represented by a dot inside a circle. Equation 4.14 shows an XNOR operation.

$$A \odot B = Y \tag{4.14}$$

Table 4.7 is the truth table for XNOR .

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

Table 4.7: Truth Table for XNOR

In Figure 4.8, the input variables A and B are wired to an XNOR gate, and the output from that gate goes to Y.

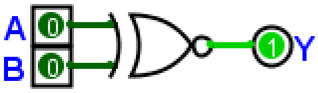


Figure 4.8: XNOR Gate

4.3.5 *Buffer*

A buffer (sometimes called *Transfer*) is a Boolean operation that transfers the input to the output without change. If the input is *True*, then the output will be *True* and if the input is *False*, then the output will be *False*. It may seem to be an odd function since this operation does not change anything, but it has an important use in a circuit. As logic circuits become more complex, the signal from input to output may become weak and no longer able to drive (or activate) additional gates. A buffer is used to boost (and stabilize) a logic level so it is more dependable. Another important function for a buffer is to clean up an input signal. As an example, when an electronic circuit interacts with the physical world (such as a user pushing a button), there is often a very brief period when the signal from that physical device waivers between high and low unpredictably. A buffer can smooth out that signal so it is a constant high or low without voltage spikes in between.

Table 4.8 is the truth table for buffer.

Input	Output
0	0
1	1

Table 4.8: Truth Table for a Buffer

Buffers are rarely used in schematic diagrams since they do not actually change a signal; however, Figure 4.9, illustrates a buffer.

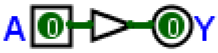


Figure 4.9: Buffer

4.4 UNIVARIATE BOOLEAN ALGEBRA PROPERTIES

4.4.1 *Introduction*

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

4.4.2 *Identity*

In mathematics, an identity is an equality where the left and right members are the same regardless of the values of the variables present. As an example, Equation 4.15 is an identity since the two members are identical regardless of the value of α:

$$\frac{\alpha}{2} = 0.5\alpha \tag{4.15}$$

An *Identity Element* is a special member of a set such that when that element is used in a binary operation the other element in that operation is not changed. This is sometimes called the *Neutral Element* since it has no effect on binary operations. As an example, in Equation 4.16 the two members of the equation are always identical. Therefore, zero is the identity element for addition since anything added to zero remains unchanged.

$$a + 0 = a \tag{4.16}$$

In a logic circuit, combining any logic input with a logic zero through an OR gate yields the original input. Logic zero, then, is considered the OR identity element because it causes the input of the gate to be copied to the output unchanged. Because OR is represented by a plus sign when written in a Boolean equation, and the identity element for OR is zero, Equation 4.17 is *True*.

$$A + 0 = A \tag{4.17}$$

The bottom input to the OR gate in 4.10 is a constant logic zero, or *False*. The output for this circuit, Y, will be the same as input A; therefore, the identity element for OR is zero.

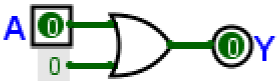


Figure 4.10: OR Identity Element

In the same way, combining any logic input with a logic one through an AND gate yields the original input. Logic one, then, is considered the AND identity element because it causes the input of the gate to be copied to the output unchanged. Because AND is represented by a multiplication sign when written in a Boolean equation, and the identity element for AND is one, Equation 4.18 is *True*.

$$A * 1 = A \tag{4.18}$$

The bottom input to the AND gate in 4.11 is a constant logic one, or *True*. The output for this circuit, Y, will be the same as input A; therefore, the identity element for AND is one.

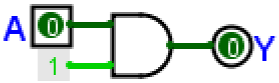


Figure 4.11: AND Identity Element

4.4.3 *Idempotence*

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

Remember that in Boolean expressions a plus sign represents an OR *gate, not mathematical addition.*

$$A + A = A \tag{4.19}$$

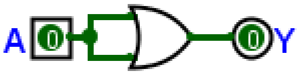


Figure 4.12: Idempotence Property for OR Gate

Figure 4.12 illustrates the idempotence property for an AND gate.

Remember that in a Boolean expression a multiplication sign represents an AND gate, not mathematical multiplying.

$$A * A = A \tag{4.20}$$

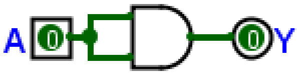


Figure 4.13: Idempotence Property for AND Gate

4.4.4 *Annihilator*

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

$$A + 1 = 1 \tag{4.21}$$

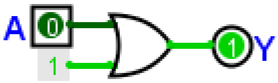


Figure 4.14: Annihilator For OR Gate

The bottom input for the OR gate in Figure 4.14 is a constant logic one, or *True*. The output for this circuit will be *True* (or 1) no matter whether input A is *True* or *False* (1 or 0).

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

$$A * 0 = 0 \tag{4.22}$$

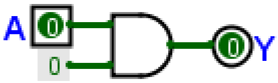


Figure 4.15: Annihilator For AND Gate

The bottom input for the AND gate in Figure 4.15 is a constant logic zero, or *False*. The output for this circuit will be *False* (or 0) no matter whether input A is *True* or *False* (1 or 0).

4.4.5 *Complement*

In Boolean logic there are only two possible values for variables: 0 and 1. Since either a variable or its complement must be one, and since combining any data with one through an OR gate yields one (see the Annihilator in Equation 4.21), then the following is true:

$$A + A' = 1 \tag{4.23}$$

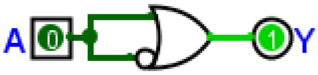


Figure 4.16: OR Complement

In Figure 4.16, the output (Y) will always equal one, regardless of the value of input A. This leads to the general property that when a variable and its complement are combined through an OR gate the output will always be one.

In the same way, since either a variable or its complement must be zero, and since combining any data with zero through an AND gate yields zero (see the Annihilator in Equation 4.22), then the following is true:

$$A * A' = 0 \tag{4.24}$$

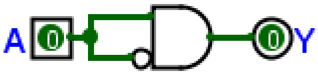


Figure 4.17: AND Complement

4.4.6 *Involution*

Another law having to do with complementation is that of Involution. Complementing a Boolean variable two times (or any even number of times) results in the original Boolean value.

The Involution Property is sometimes called the "Double Complement" Property.

$$(A')' = A \tag{4.25}$$

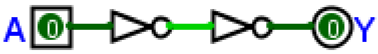


Figure 4.18: Involution Property

In the circuit illustrated in Figure 4.18, the output (Y) will always be the same as the input (A).

Propagation Delay

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

4.5 MULTIVARIATE BOOLEAN ALGEBRA PROPERTIES

4.5.1 *Introduction*

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

4.5.2 *Commutative*

The examples here show only two variables, but this property is true for any number of variables.

In essence, the commutative property indicates that the order of the input variables can be reversed in either OR or AND gates without changing the truth of the expression. Equation 4.26 expresses this property algebraically.

$$\begin{aligned} A + B &= B + A \\ A * B &= B * A \end{aligned} \tag{4.26}$$

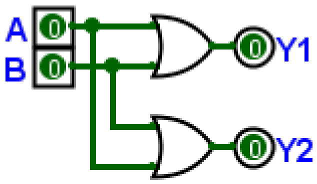


Figure 4.19: Commutative Property for OR

XOR and XNOR are also commutative; but for only two variables, not three or more.

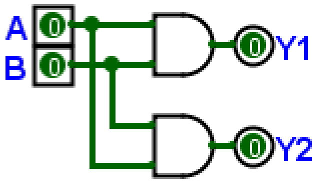


Figure 4.20: Commutative Property for AND

In Figures 4.19 and 4.20 the inputs are reversed for the two gates, but the outputs are the same. For example, A is entering the top input for the upper OR gate, but the bottom input for the lower gate; however, Y1 is always equal to Y2.

4.5.3 *Associative*

This property indicates that groups of variables in an OR or AND gate can be associated in various ways without altering the truth of the equations. Equation 4.27 expresses this property algebraically:

The examples here show only three variables, but this property is true for any number of variables.

$$\begin{aligned}(A + B) + C &= A + (B + C) \\ (A * B) * C &= A * (B * C)\end{aligned} \tag{4.27}$$

XOR and XNOR are also associative; but for only two variables, not three or more.

In the circuits in Figure 4.21 and 4.22 , notice that A and B are associated together in the first gate, and then C is associated with the output of that gate. Then, in the lower half of the circuit, B and C are associated together in the first gate, and then A is associated with the output of that gate. Since Y1 is always equal to Y2 for any combination of inputs, it does not matter which of the two variables are associated together in a group of gates.

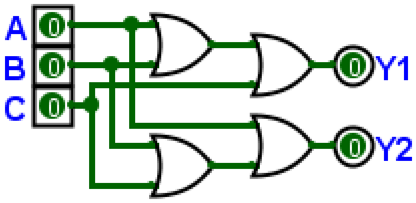


Figure 4.21: Associative Property for OR

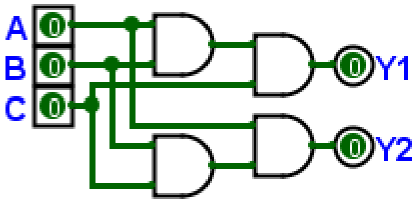


Figure 4.22: Associative Property for AND

4.5.4 *Distributive*

The distributive property of real number algebra permits certain variables to be "distributed" to other variables. This operation is frequently used to create groups of variables that can be simplified; thus, simplifying the entire expression. Boolean algebra also includes a distributive property, and that can be used to combine OR or AND gates in various ways that make it easier to simplify the circuit. Equation 4.28 expresses this property algebraically:

$$\begin{aligned} A(B+C) &= AB + AC \\ A + (BC) &= (A+B)(A+C) \end{aligned} \tag{4.28}$$

In the circuits illustrated in Figures 4.23 and 4.24, notice that input A in the top half of the circuit is distributed to inputs B and C in the bottom half. However, output Y1 is always equal to output Y2 regardless of how the inputs are set. These two circuits illustrate Distributive of AND over OR and Distributive of OR over AND .

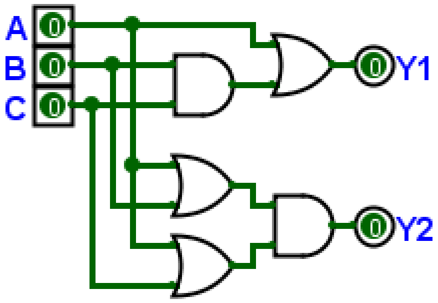


Figure 4.23: Distributive Property for AND over OR

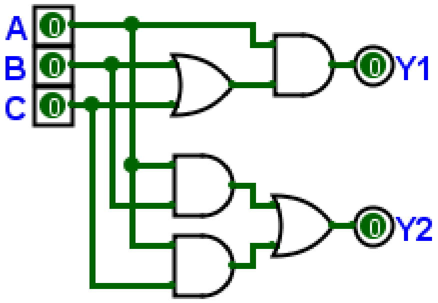


Figure 4.24: Distributive Property for OR over AND

4.5.5 *Absorption*

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

$$\begin{aligned} A + (AB) &= A \\ A(A + B) &= A \end{aligned} \tag{4.29}$$

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

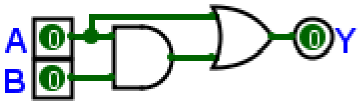


Figure 4.25: Absorption Property (Version 1)

Table 4.9 is the truth table for the circuit in Figure 4.25.

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

Table 4.9: Truth Table for Absorption Property

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

The circuit illustrated in Figure 4.26 is the second version of the Absorption Property. Like the first Absorption Property circuit, a truth table would demonstrate that input B is absorbed by the circuit.

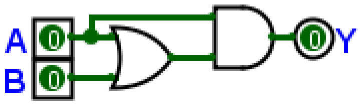


Figure 4.26: Absorption Property (Version 2)

4.5.6 *Adjacency*

The adjacency property simplifies a circuit by removing unnecessary gates.

$$AB + AB' = A \tag{4.30}$$

This property can be proven by simple algebraic manipulation:

$$\begin{array}{rr} AB + AB' & \text{Original Expression} \\ A(B + B') & \text{Distributive Property} \\ A1 & \text{Complement Property} \\ A & \text{Identity Element} \end{array} \tag{4.31}$$

The circuit in Figure 4.27 illustrates the adjacency property. If this circuit were constructed it would be seen that the output, Y, is always the same as input A; therefore, this entire circuit could be replaced by a single wire from input A to output Y.

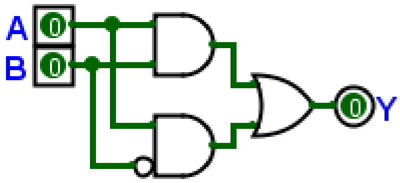


Figure 4.27: Adjacency Property

4.6 DEMORGAN'S THEOREM

4.6.1 *Introduction*

A mathematician named Augustus DeMorgan developed a pair of important theorems regarding the complementation of groups in Boolean algebra. DeMorgan found that an OR gate with all inputs inverted (a Negative-OR gate) behaves the same as a NAND gate with non-inverted inputs; and an AND gate with all inputs inverted (a Negative-AND gate) behaves the same as a NOR gate with non-inverted inputs. DeMorgan's theorem states that inverting the output of any gate is the same as using the opposite type of gate with inverted inputs. Figure 4.28 illustrates this in circuit terms: the NAND gate with normal inputs and the OR gate with inverted inputs are functionally equivalent; that is, Y1 will always equal Y2, regardless of the values of input A or B.

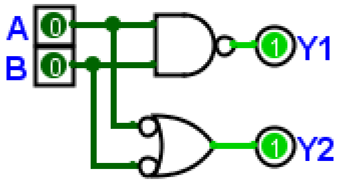


Figure 4.28: DeMorgan's Theorem Defined

The NOT function is commonly represented in an equation as an apostrophe because it is easy to enter with a keyboard, like: $(AB)'$ for *A AND B NOT*. However, it is easiest to work with DeMorgan's theorem if NOT is represented by an overline rather than an apostrophe, so it would be written as $\overline{AB}$ rather than $(AB)'$. Remember that an overline is a grouping symbol (like parenthesis) and it means that everything under that bar would first be combined (using an AND or OR gate) and then the output of the combination would be complemented.

4.6.2 *Applying DeMorgan's Theorem*

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

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

To illustrate:

$$\overline{A * B} \leftrightarrow \overline{A} + \overline{B} \tag{4.32}$$

$$\overline{A + B} \leftrightarrow \overline{A} * \overline{B} \tag{4.33}$$

Equation 4.32 shows how a two-input NAND gate is "broken" to form an OR gate with two inverted inputs and equation 4.33 shows how a two-input NOR gate is "broken" to form an AND gate with two complemented inputs.

4.6.3 *Simple Example*

When multiple "layers" of bars exist in an expression, only one bar is broken at a time, and the longest, or uppermost, bar is broken first. As an example, consider the circuit in Figure 4.29:

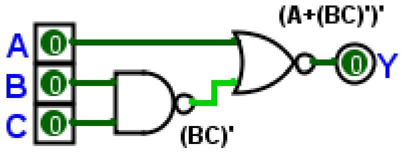


Figure 4.29: DeMorgan's Theorem Example 1

By writing the output at each gate (as illustrated in Figure 4.29), it is easy to determine the Boolean expression for the circuit. Note: all circuit diagrams in this book are generated with *Logisim-evolution* and the text tool in that software does not permit drawing overbars. Therefore, the circuit diagram will use the apostrophe method of indicating NOT but overbars will be used in the text.

$$\overline{A + \overline{BC}} \tag{4.34}$$

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

$$\begin{aligned} &\overline{A + \overline{BC}} && \text{Original Expression} \\ &\overline{A}\,\overline{\overline{BC}} && \text{"Break" the longer bar} \\ &\overline{A}BC && \text{Involution Property} \end{aligned} \tag{4.35}$$

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

4.6.4 *Incorrect Application of DeMorgan's Theorem*

More than one bar is never broken in a single step, as illustrated in Equation 4.36:

$$\begin{aligned} &\overline{A + \overline{BC}} && \text{Original Expression} \\ &\overline{A\overline{B}} + \overline{\overline{C}} && \text{Improperly Breaking Two Bars} \\ &\overline{A}B + C && \text{Incorrect Solution} \end{aligned} \tag{4.36}$$

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

4.6.5 *About Grouping*

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

Consider the circuit in Figure 4.30.

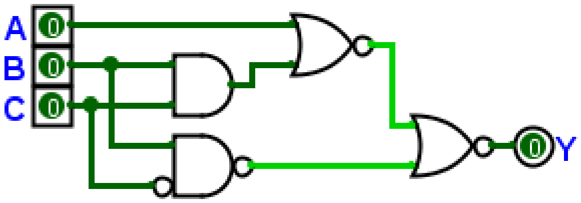


Figure 4.30: DeMorgan's Theorem Example 2

As always, the first step in simplifying this circuit is to generate the Boolean expression for the circuit, which is done by writing the sub-expression at the output of each gate. That results in Expression 4.37, which is then simplified.

$$\begin{aligned}
\overline{\overline{A+BC}+\overline{A\overline{B}}} &\quad \text{Original Expression} \qquad (4.37)\\
(\overline{\overline{A+BC}})(\overline{\overline{A\overline{B}}}) &\quad \text{Breaking the Longest Bar}\\
(A+BC)(A\overline{B}) &\quad \text{Involution}\\
(AA\overline{B})(BCA\overline{B}) &\quad \text{Distribute } A\overline{B} \text{ to } (A+BC)\\
(A\overline{B})+(BCA\overline{B}) &\quad \text{Idempotence: } AA=A\\
(A\overline{B})+(0CA)) &\quad \text{Complement: } B\overline{B}=0\\
(A\overline{B})+0 &\quad \text{Annihilator: } 0CA=0\\
A\overline{B} &\quad \text{Identity: } A+0=A
\end{aligned}$$

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

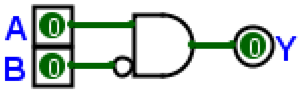


Figure 4.31: DeMorgan's Theorem Example 2 Simplified

4.6.6 *Summary*

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

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

4.6.7 *Example Problems*

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

	Original Expression	Simplified
1	$(\overline{A+B})(\overline{ABC})(\overline{\overline{A}C})$	$\overline{A}\,\overline{B}\,\overline{C}$
2	$\overline{(AB+\overline{B}C)+(B\overline{C}+\overline{A}B)}$	$\overline{B}\,\overline{C}$
3	$(AB+\overline{B}C)(AC+\overline{A}\,\overline{C})$	$\overline{A}+\overline{C}$

4.7 BOOLEAN FUNCTIONS

Consider Figure 4.32, which is a generic circuit with two inputs and one output.

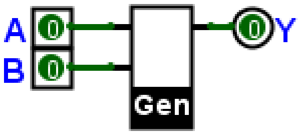


Figure 4.32: Generic Function

Without knowing anything about what is in the unlabeled box at the center of the circuit, there are a number of possible truth tables which could describe the circuit's output. Two possibilities are shown in Truth Table 4.10 and Truth Table 4.11.

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

Table 4.10: Truth Table for Generic Circuit One

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

Table 4.11: Truth Table for Generic Circuit Two

In fact, there are 16 possible truth tables for this circuit. Each of those truth tables reflect a single potential function of the circuit by setting various combinations of input/output. Therefore, any two-input, one-output circuit has 16 possible functions. It is easiest to visualize all 16 combinations of inputs/outputs by using an odd-looking truth table. Consider only one of those 16 functions, the one for the generic circuit described by Truth Table 4.11. That function is also found in the Boolean Functions Table 4.13 and one row from that table is reproduced in Table 4.12.

A	0	0	1	1	
B	0	1	0	1	
F_6	0	1	1	0	Exclusive Or (XOR): $A \oplus B$

Table 4.12: Boolean Function Six

The line shown in Table 4.12 is for *Function 6*, or F_6 (note that the pattern of the outputs is 0110, which is binary 6). Inputs A and B are listed at the top of the table. For example, the highlighted column of the table shows that when A is zero and B is one the output is one. Therefore, on the line that defines F_6, the output is *True* when $[(A = 0 \text{ AND } B = 1) \text{ OR } (A = 1 \text{ AND } B = 0)]$ This is an XOR function, and the last column of the table verbally describes that function.

Table 4.13 is the complete Boolean Function table.

A	0	0	1	1	
B	0	1	0	1	
F_0	0	0	0	0	Zero or Clear. Always zero (Annihilation)
F_1	0	0	0	1	Logical AND: $A * B$
F_2	0	0	1	0	Inhibition: AB' or $A > B$
F_3	0	0	1	1	Transfer A to Output, Ignore B
F_4	0	1	0	0	Inhibition: $A'B$ or $B > A$
F_5	0	1	0	1	Transfer B to Output, Ignore A
F_6	0	1	1	0	Difference, XOR: $A \oplus B$
F_7	0	1	1	1	Logical OR: $A + B$
F_8	1	0	0	0	Logical NOR: $(A + B)'$
F_9	1	0	0	1	Equivalence, XNOR: $(A = B)'$
F_{10}	1	0	1	0	Not B and ignore A, B Complement
F_{11}	1	0	1	1	Implication, $A + B'$, $B >= A$
F_{12}	1	1	0	0	Not A and ignore B, A Complement
F_{13}	1	1	0	1	Implication, $A' + B$, $A >= B$
F_{14}	1	1	1	0	Logical NAND: $(A * B)'$
F_{15}	1	1	1	1	One or Set. Always one (Identity)

Table 4.13: Boolean Functions

4.8 FUNCTIONAL COMPLETENESS

A set of Boolean operations is said to be *functionally complete* if every possible Boolean function can be derived from that set. The Primary Logic Operations (page **??**) are functionally complete since the Secondary Logic Functions (page 69) can be derived from them. As an example, Equation 4.38 and Figure 4.33 shows how an XOR function can be derived from only AND, OR, and NOT gates.

$$((A * B)' * (A + B)) = A \oplus B \tag{4.38}$$

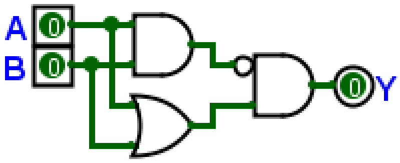


Figure 4.33: XOR Derived From AND/OR/NOT

While the Primary Operations are functionally complete, it is possible to define other functionally complete sets of operations. For example, using DeMorgan's Theorem, the set of {AND, NOT} is also functionally complete since the OR operation can be defined as $(A'B')'$. In fact, both {NAND} and {NOR} operations are functionally complete by themselves. As an example, the NOT operation can be derived using only NAND gates: $(A|A)$. Because all Boolean functions can be derived from either NAND or NOR operations, these are sometimes considered *universal* operations and it is a common challenge for students to create some complex Boolean function using only one of these two types of operations.

5

BOOLEAN EXPRESSIONS

5.1 INTRODUCTION

Electronic circuits that do not require any memory devices (like flip-flops or registers) are created using what is called "Combinational Logic." These systems can be quite complex, but all outputs are determined solely by input signals that are processed through a series of logic gates. Combinational circuits can be reduced to a Boolean Algebra expression, though it may be quite complex; and that expression can be simplified using methods developed in this chapter and Chapter 6, Practice Problems, page 113, and Chapter 7, One In First Cell, page 141. Combinational circuits are covered in Chapter 8, Introduction, page 157.

In contrast, electronic circuits that require memory devices (like flip-flops or registers) use what is called "Sequential Logic." Those circuits often include feedback loops, so the final output is determined by input signals plus the feedback loops that are processed through a series of logic gates. This makes sequential logic circuits much more complex than combinational and the simplification of those circuits is covered in Chapter 9, Comparators, page 173.

Finally, most complex circuits include both combinational and sequential sub-circuits. In that case, the various sub-circuits would be independently simplified using appropriate methods. Several examples of these types of circuits are analyzed in Chapter 14, Elevator, page 225.

5.2 CREATING BOOLEAN EXPRESSIONS

A circuit designer is often only given a written (or oral) description of a circuit and then asked to build that device. Too often, the designer may receive notes scribbled on the back of a dinner napkin, along with some verbal description of the desired output, and be expected to build a circuit to accomplish that task. Regardless of the form of the request, the process that the designer follows, in general, is:

1. Write The Problem Statement. The problem to be solved is written in a clear, concise statement. The better this statement is written the easier each of the following steps will be, so time spent polishing the problem statement is worthwhile.

2. Construct A Truth Table. Once the problem is clearly defined, the circuit designer constructs a truth table where all input-

s/outputs are included. It is essential that all possible input combinations that lead to a *True* output are identified.

3. Write A Boolean Expression. When the truth table is completed, it is easy to create a Boolean expression from that table as covered in Example 5.2.1.
4. Simplify the Boolean Expression. The expression should be simplified as much as possible, and that process is covered in this chapter, Chapter 6, Practice Problems, page 113, and Chapter 7, One In First Cell, page 141.
5. Draw The Logic Diagram. The logic diagram for a circuit is constructed from the simplified Boolean expression.
6. Build The Circuit. If desired, a physical circuit can be built using the logic diagram.

5.2.1 *Example*

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

1. Problem Statement. The problem is already fairly well stated. A circuit is needed that will activate the "drop poster" machine when any two of three inputs (joy buzzer, fake blood, itching powder), but not all three, are *True*.
2. Truth Table. Let J be the Joy Buzzer, B be the Fake Blood, and P be the Itching Powder; and let the truth table inputs be *True* (or 1) when any of those items are present in the shipping box. Let the output D be for "Drop Poster" and when it is *True* (or 1) then a poster will be dropped into the shipping box. The Truth table is illustrated in Table 5.1.

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

Table 5.1: Truth Table for Example

3. Write Boolean Expression. According to the Truth Table, the poster will be dropped into the shipping box in only three cases (when output D is *True*). Equation 5.1 was generated from the truth table.

$$BP + JP + JB = D \tag{5.1}$$

4. Simplify Boolean Expression. The Boolean expression for this problem is already as simple as possible so no further simplification is needed.

5. Draw Logic Diagram. Figure 5.1 was drawn from the switching equation.

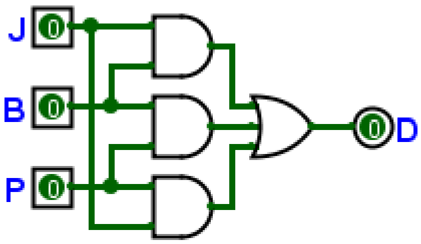


Figure 5.1: Logic Diagram From Switching Equation

6. Build The Circuit. This circuit could be built (or "realized") with three AND gates and one 3-input OR gate.

5.3 MINTERMS AND MAXTERMS

5.3.1 *Introduction*

The solution to a Boolean equation is normally expressed in one of two formats: SOP or POS.

5.3.2 *Sum Of Products (SOP) Defined*

Equation 5.2 is an example of a SOP expression. Notice that the expression describes four inputs (A, B, C, D) that are combined through two AND gates and then the output of those AND gates are combined through an OR gate.

$$(A'BC'D) + (AB'CD) = Y \tag{5.2}$$

Each of the two terms in this expression is a *minterm*. Minterms can be identified in a Boolean expression as a group of inputs joined by an AND gate and then two or more minterms are combined with an OR gate. The circuit illustrated in Figure 5.2 would realize Equation 5.2.

Notice the inverting bubble on three of the AND gate inputs.

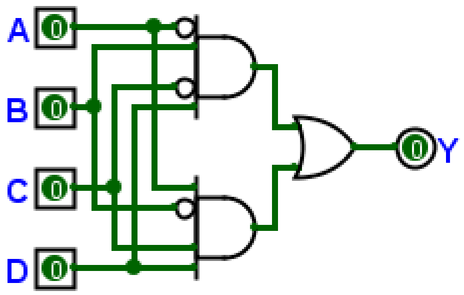


Figure 5.2: Logic Diagram For SOP Example

5.3.3 *Product of Sums (POS) Defined*

Equation 5.3 is an example of a POS expression. Notice that the expression describes four inputs (A, B, C, D) that are combined through two OR gates and then the output of those OR gates are combined through an AND gate.

$$(A' + B + C +' D)(A + B' + C + D) = Y \tag{5.3}$$

Each term in this expression is called a *maxterm*. Maxterms can be identified in a Boolean expression as a group of inputs joined by an OR gate; and then two or more maxterms are combined with an AND gate. The circuit illustrated in Figure 5.3 would realize Equation 5.3.

Notice the inverting bubble on three of the OR gate inputs.

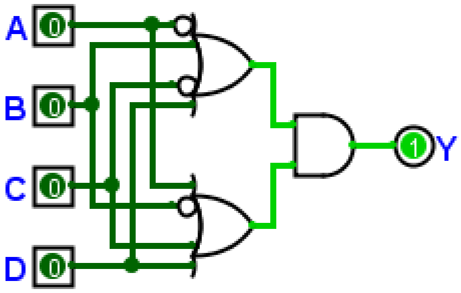


Figure 5.3: Logic Diagram For POS Example

5.3.4 *About Minterms*

A term that contains all of the input variables in one row of a truth table joined with an AND gate is called a minterm. Consider truth table 5.2 which is for a circuit with three inputs (A, B, and C) and one output (Q).

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

Table 5.2: Truth Table for First Minterm Example

The circuit that is represented by this truth table would output a *True* in only two cases, when the inputs are $A'BC$ or $AB'C$. Equation 5.4 describes this circuit.

$$(A'BC) + (AB'C) = Q \tag{5.4}$$

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

Minterms are frequently abbreviated with a lower-case m along with a subscript that indicates the decimal value of the variables. For example, $A'BC$, the first of the *True* outputs in the truth table above, has a binary value of 011, which is a decimal value of 3; thus, the minterm is m_3. The other minterm in this equation is m_5 since its binary value is 101, which equals decimal 5. It is possible to verbally describe the entire circuit as: m_3 OR m_5. For convenience, each of the minterm numbers are indicated in the last column of the truth table.

As another example, consider truth table 5.3.

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

Table 5.3: Truth Table for Second Minterm Example

Equation 5.5 describes this circuit because these are the rows where the output is *True*.

$$(A'B'C'D') + (A'BC'D) + (A'BCD') + (ABCD') = Q \tag{5.5}$$

These would be minterms m_0, m_5, m_6, and m_{14}. Equation 5.6 shows a commonly used, more compact way to express this result.

$$\int(A, B, C, D) = \sum(0, 5, 6, 14) \tag{5.6}$$

Equation 5.6 would read: "For the function of inputs A, B, C, and D, the output is *True* for minterms 0, 5, 6, and 14 when they are combined with an OR gate." This format is called *Sigma Notation*, and it is easy to derive the full Boolean equation from it by remembering that m_0 is 0000, or $A'B'C'D'$; m_5 is 0101, or $A'BC'D$; m_6 is 0110, or $A'BCD'$; and m_{14} is 1110, or $ABCD'$. Therefore, the Boolean equation can be quickly created from the Sigma Notation.

A logic equation that is created using minterms is often called the *Sum of Products* (or *SOP*) since each term is composed of inputs ANDed together ("products") and the terms are then joined by OR gates ("sums").

5.3.5 *About Maxterms*

A term that contains all of the input variables joined with an OR gate ("added together") for a *False* output is called a *maxterm*. Consider Truth Table 5.4, which has three inputs (A, B, and C) and one output (Q):

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

Table 5.4: Truth Table for First Maxterm Example

The circuit that is represented by this truth table would output a *False* in only two cases. Since there are fewer *False* outputs than *True*, it is easier to create a Boolean equation that would generate the *False* outputs. Because the equation describes the *False* outputs, each term is built by *complementing* the inputs for each of the *False* output lines. After the output groups are defined, they are joined with an AND gate. Equation 5.7 is the Boolean equation for Truth Table 5.4.

$$(A + B' + C)(A' + B' + C) = Q \tag{5.7}$$

The terms $A + B' + C$ and $A' + B' + C$ are called $maxterms$, and they contain the complement of the input variables for each of the

False output lines. Maxterms are most often used to describe circuits that have fewer *False* outputs than *True*. In Truth Table 5.4, there are only two *False* outputs with six *True* outputs, so maxterms describe the circuit most efficiently.

Maxterms are frequently abbreviated with an upper-case M along with a subscript that indicates the decimal value of the complements of the variables. For example, the complement of $A + B' + C$, the first of the *False* outputs in Truth Table 5.4, is 010, which is a decimal value of 2; thus, the maxterm would be M_2. This can be confusing, but remember that the *complements* of the inputs are used to form the expression. Thus, $A + B' + C$ is 010, not 101. The other maxterm in this equation is M_6 since the binary value of its complement is 110, which equals decimal 6. It is possible to describe the entire circuit as a the product of two groups of maxterms: M_2 and M_6.

As another example, consider Truth Table 5.5.

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

Table 5.5: Truth Table for Second Maxterm Example

Equation 5.8 describes this circuit.

$$(A + B + C' + D')(A' + B + C + D')(A' + B' + C + D) = Q \tag{5.8}$$

These would be maxterms M_3, M_9, and M_{12}. Equation 5.9 shows a commonly used, more compact way to express this result.

$$\int(A,B,C,D) = \prod(3,9,12) \tag{5.9}$$

Equation 5.9 would read: "For the function of A, B, C, D, the output is *False* for maxterms 3, 9, and 12 when they are combined with an AND gate." This format is called *Pi Notation*, and it is easy to derive the Boolean equation from it. Remember that M_3 is 0011, or $A + B + C' + D'$ (the complement of the inputs), M_9 is 1001, or $A' + B + C + D'$, and M_{12} is 1100, or $A' + B' + C + D$. The original equation can be quickly created from the Pi Notation.

A logic equation that is created using maxterms is often called the *Product of Sums* (or *POS*) since each term is composed of inputs OR ed together ("sums") and the terms are then joined by AND gates ("products").

5.3.6 *Minterm and Maxterm Relationships*

The minterms and maxterms of a circuit have three interesting relationships: equivalence, duality, and inverse. To define and understand these terms, consider Truth Table 5.6 for some unspecified "black box" circuit:

Inputs			Outputs		Terms	
A	B	C	Q	Q'	minterm	Maxterm
0	0	0	0	1	$A'B'C'$ (m_0)	$A+B+C$ (M_0)
0	0	1	1	0	$A'B'C$ (m_1)	$A+B+C'$ (M_1)
0	1	0	0	1	$A'BC'$ (m_2)	$A+B'+C$ (M_2)
0	1	1	0	1	$A'BC$ (m_3)	$A+B'+C'$ (M_3)
1	0	0	0	1	$AB'C'$ (m_4)	$A'+B+C$ (M_4)
1	0	1	1	0	$AB'C$ (m_5)	$A'+B+C'$ (M_5)
1	1	0	0	1	ABC' (m_6)	$A'+B'+C$ (M_6)
1	1	1	1	0	ABC (m_7)	$A'+B'+C'$ (M_7)

Table 5.6: Minterm and Maxterm Relationships

5.3.6.1 *Equivalence*

The minterms and the maxterms for a given circuit are considered equivalent ways to describe that circuit. For example, the circuit described by Truth Table 5.6 could be defined using minterms (Equation 5.10).

$$\int(A,B,C) = \sum(1,5,7) \tag{5.10}$$

However, that same circuit could also be defined using maxterms (Equation 5.11).

$$\int(A,B,C) = \prod(0,2,3,4,6) \tag{5.11}$$

These two functions describe the same circuit and are, consequently, equivalent. The *Sigma Function* includes the terms 1, 5, and 7 while the *Pi Function* includes all other terms in the truth table (0, 2, 3, 4, and 6). To put it a slightly different way, the *Sigma Function* describes the truth table rows where Q = 1 (minterms) while the *Pi Function* describes the rows in the same truth table where Q = 0 (maxterms). Therefore, Equation 5.12 can be derived.

$$\sum(1,5,7) \equiv \prod(0,2,3,4,6) \tag{5.12}$$

5.3.6.2 *Duality*

Each row in Truth Table 5.6 describes two terms that are considered duals. For example, minterm m_5 $(AB'C)$ and maxterm M_5 $(A' + B + C')$ are duals. Terms that are duals are complements of each other (Q vs. Q′) and the input variables are also complements of each other; moreover, the inputs for the three minterms are combined with an AND while the maxterms are combined with an OR. The output of the circuit described by Truth Table 5.6 could be defined using minterms (Equation 5.13).

$$Q = \sum(1,5,7) \tag{5.13}$$

The dual of the circuit would be defined by using the maxterms for the same output rows. However, those rows are the *complement* of the circuit (Equation 5.14).

$$Q' = \prod(1,5,7) \tag{5.14}$$

This leads to the conclusion that the complement of a *Sigma Function* is the *Pi Function* with the same inputs, as in Equation 5.15 (the overline was used to emphasize the the fact that the *PI Function* is complemented).

$$\sum(1,5,7) = \overline{\prod(1,5,7)} \tag{5.15}$$

5.3.6.3 *Inverse*

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

$$Q = \sum(1,5,7) \tag{5.16}$$

$$Q' = \sum(0,2,3,4,6) \tag{5.17}$$

5.3.6.4 *Summary*

These three relationships are summarized in the following table. Imagine a circuit with two or more inputs and an output of Q. Table 5.7 summarizes the various relationships in the Truth Table for that circuit.

Minterms where Q is 1	Minterms where Q' is 1
Maxterms where Q is 1	Maxterms where Q' is 1

Table 5.7: Minterm-Maxterm Relationships

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

5.3.7 *Sum of Products Example*

5.3.7.1 *Given*

A "vote-counter" machine is designed to turn on a light if any two or more of three inputs are *True*. Create a circuit to realize this machine.

5.3.7.2 *Truth Table*

When realizing a circuit from a verbal description, the best place to start is constructing a truth table. This will make the Boolean expression easy to write and then make the circuit easy to realize. For the "vote-counter" problem, start by defining variables for the truth table: Inputs A, B, C and Output Q.

Next, construct the truth table by identifying columns for each of the three input variables and then indicate the output for every possible input condition (Table 5.8).

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

Table 5.8: Truth Table for SOP Example

5.3.7.3 *Boolean Equation*

In a truth table, if there are fewer *True* outputs than *False*, then it is easiest to construct a Sum-of-Products equation. In that case, a Boolean expression can be derived by creating the minterms for all *True* outputs and combining those minterms with OR gates. Equation 5.18 is the *Sigma Function* of this circuit.

$$\int(A, B, C) = \sum(3, 5, 6, 7) \tag{5.18}$$

It may be possible to simplify the Boolean expression, but that process is covered elsewhere in this book.

At this point, a circuit could be created with four 3-input AND gates combined into one 4-input OR gate.

5.3.8 *Product of Sums Example*

5.3.8.1 *Given*

A local supply company is designing a machine to sort packages for shipping. All packages go to the post office *except* packages going to the local ZIP code containing chemicals (they are shipped by courier) and packages going to a distant ZIP code containing only perishables (they are shipped via air freight).

5.3.8.2 *Truth Table*

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

- Ship via post office (the Output): O is *True* if ship by Post Office
- Zip Code: Z is *True* if the zip code is local
- Chemicals: C is *True* if the package contains chemicals
- Perishable: P is *True* if the package contains perishables

Next, construct the truth table by identifying columns for each of the three input variables and then indicate the output for every possible input condition (Table 5.8).

Inputs			Outputs	
Z	C	P	O	M
0	0	0	1	0
0	0	1	0	1
0	1	0	1	2
0	1	1	0	3
1	0	0	1	4
1	0	1	1	5
1	1	0	0	6
1	1	1	0	7

Table 5.9: Truth Table for POS Example

5.3.8.3 *Boolean Equation*

In the truth table, if there are fewer *False* outputs than *True* then it is easiest to construct a Products-of-Sums equation. In that case, a Boolean expression can be derived by creating the maxterms for all *False* outputs and combining the complement those maxterms with AND gates. Equation 5.19 is the *Pi Expression* of this circuit.

$$\int(Z,C,P) = \prod(1,3,6,7) \tag{5.19}$$

At this point, a circuit could be created with four 3-input OR gates combined into one 4-input AND gate.

It may be possible to simplify the Boolean expression, but that process is covered elsewhere in this book.

5.3.9 *Summary*

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

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

5.4 CANONICAL FORM

5.4.1 *Introduction*

The word "canonical" simply means "standard" and it is used throughout mathematics and science to denote some standard form for equations. In digital electronics, Boolean equations are considered to be in canonical form when each of the terms in the equation includes all of the possible inputs and those terms appear in the same order as in the truth table. Using the canonical form is important when simplifying a Boolean equation. For example, imagine the solution to a given problem generated table 5.10.

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

Table 5.10: Canonical Example Truth Table

Minterm equation 5.20 is derived from the truth table and is presented in canonical form. Notice that each term includes all possible inputs (A, B, and C), and that the terms are in the same order as they appear in the truth table.

$$(A'B'C) + (A'BC) + (ABC) = Q \tag{5.20}$$

Frequently, though, a Boolean equation is expressed in standard form, which is not the same as canonical form. Standard form means that some of the terms have been simplified and not all of the inputs will appear in all of the terms. For example, consider Equation 5.21, which is the solution for a 4-input circuit.

$$(A'C) + (B'CD) = Q \tag{5.21}$$

This equation is in standard form so the first term, $A'C$, does not include inputs B or D and the second term, $B'CD$, does not include input A. However, all inputs must be present in every term for an equation to be in canonical form.

Building a truth table for an equation in standard form raises an important question. Consider the truth table for Equation 5.21.

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

Table 5.11: Truth Table for Standard Form Equation

In what row would $B'CD$, the second term in Equation 5.21, be placed? $B'CD$ is 011 (in binary), but since the A term is missing would it be a 0 or 1; in other words, would $B'CD$ generate an output of 1 for row 0011 (m_3) or 1011 (m_{11})? (The output for these two rows are marked with a question mark in Table 5.11.) In fact, the output for *both* of these rows must be considered *True* in order to ensure that all possible combinations of input are covered. Thus, the final equation for this circuit must include at least these two terms:

$(A'B'CD) + (AB'CD)$. In the same way, the term $A'C$ means that the output is *True* for m_2, m_3, m_6, and m_7 since input $A'C$ is *True* and any minterm that contains those two value is also considered *True*. Thus, the final equation for this circuit must include at least these four terms: $(A'B'CD') + (A'B'CD) + (A'BCD') + (A'BCD)$.

5.4.2 *Converting Terms Missing One Variable*

To change a standard Boolean expression that is missing one input term into a canonical Boolean expression, insert both *True* and *False* for the missing term into the original standard expression. As an example, consider the term $B'CD$. Since term A is missing, both A and A' must be included in the converted canonical expression. Equation 5.22 proves that $B'CD$ can be expanded to include both possible values for A by using the Adjacency Property (page 82).

$$(B'CD) \rightarrow (AB'CD) + (A'B'CD) \tag{5.22}$$

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

Expanding a standard term that is missing one variable can also be done with a truth table. To do that, fill in an output of 1 for every line where the *True* inputs are found while ignoring all missing variables. As an example, consider Truth Table 5.11 where the outputs for m_3 and $m_1 1$ are marked with a question mark. However, the output for both of these lines should be marked as *True* because $B'CD$ is *True* (input A is ignored). Then, those two minterms lead to the Boolean expression $AB'CD + A'B'CD$.

5.4.3 *Converting Terms Missing Two Variables*

It is easiest to expand a standard expression that is missing two terms by first inserting one of the missing variables and then inserting the other missing variable in two distinct steps. The process for inserting a single missing variable is found in Section 5.4.2. Consider the term $A'C$ in a four-variable system. It is missing both the B and D variables. To expand that term to its canonical form, start by inserting either of the two missing variables. For example, Equation 5.23 illustrates entering B and B' into the expression.

$$(A'C) \rightarrow (A'BC) + (A'B'C) \tag{5.23}$$

Then, Equation 5.24 illustrates inserting D and D' into the expression.

$$\begin{aligned} (A'BC) &\rightarrow (A'BCD) + (A'BCD') \\ (A'B'C) &\rightarrow (A'B'CD) + (A'B'CD') \end{aligned} \tag{5.24}$$

In the end, $A'C$ expands to Equation 5.25:

$$\begin{aligned} (A'C) \rightarrow &(A'BCD) + (A'BCD') \\ &+ (A'B'CD) + (A'B'CD') \end{aligned} \tag{5.25}$$

Thus, in a four-variable system, any standard term with only two variables will expand to a canonical expression with four groups of four variables.

Expanding a standard term that is missing two variables can also be done with a truth table. To do that, fill in an output of 1 for every line where the *True* inputs are found while ignoring all missing variables. As an example, consider a Table 5.12, where $A'C$ is marked as *True*:

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

Table 5.12: Truth Table for Standard Form Equation

Notice that outputs for m_2, m_3, m_6, and m_7 are *True* because for each of those minterms $A'C$ is *True* (inputs B and D are ignored).

Then, those four minterms lead to the Boolean expression $A'B'CD' + A'B'CD + A'BCD' + A'BCD$.

5.4.4 *Summary*

This discussion started with Equation 5.26, which is in standard form.

$$(A'C) + (B'CD) = Q \tag{5.26}$$

After expanding both terms, Equation 5.27 is generated.

$$\begin{aligned} &(A'BCD) + (A'BCD') + (A'B'CD) \\ +&(A'B'CD') + (AB'CD) + (A'B'CD) = Q \end{aligned} \tag{5.27}$$

Notice, though, that the term $A'B'CD$ appears two times, so one of those can be eliminated by the Idempotence property (page 75), leaving Equation 5.28.

$$\begin{aligned} &(A'BCD) + (A'BCD') \\ +&(A'B'CD') + (AB'CD) + (A'B'CD) = Q \end{aligned} \tag{5.28}$$

To put the equation in canonical form, which is important for simplification; all that remains is to rearrange the terms so they are in the same order as they would appear in a truth table, which results in Equation 5.29.

$$\begin{aligned} &(A'B'CD') + (A'B'CD) \\ +&(A'BCD') + (A'BCD) + (AB'CD) = Q \end{aligned} \tag{5.29}$$

5.4.5 *Practice Problems*

Parenthesis were not used in order to save space; however, the variable groups are evident.

1	Standard (A,B,C)	$A'B+C+AB'$
	Cannonical	$A'B'C+A'BC'+A'BC+AB'C'+AB'C+ABC$
2	Standard (A,B,C,D)	$A'BC+B'D$
	Cannonical	$A'B'C'D'+A'B'CD'+A'BCD'+A'BCD+AB'C'D'+AB'CD'$
3	Standard (A,B,C,D)	$A'+D$
	Cannonical	$A'B'C'D'+A'B'C'D+A'B'CD'+A'B'CD+A'BC'D'+A'BC'D+A'BCD'+A'BCD+AB'C'D+AB'CD+ABC'D+ABCD$
4	Standard (A,B,C)	$A(B'+C)$
	Cannonical	$AB'C'+AB'C+ABC$

Table 5.13: Canonical Form Practice Problems

5.5 SIMPLIFICATION USING ALGEBRAIC METHODS

5.5.1 *Introduction*

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

5.5.2 *Starting From a Circuit*

Occasionally, the circuit designer is faced with an existing circuit and must attempt to simplify it. In that case, the first step is to find the Boolean equation for the circuit and then simplify that equation.

5.5.2.1 *Generate a Boolean Equation*

In the circuit illustrated in Figure 5.4, the A, B, and C input signals are assumed to be provided from switches, sensors, or perhaps other sub-circuits. Where these signals originate is of no concern in the task of gate reduction.

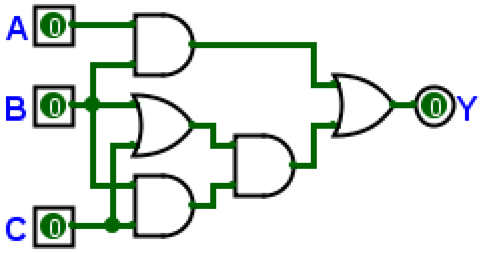


Figure 5.4: Example Circuit

To generate the Boolean equation for a circuit, write the output of each gate as determined by the input signals and type of gate, working from the inputs to the final output. Figure 5.5 illustrates the result of this process.

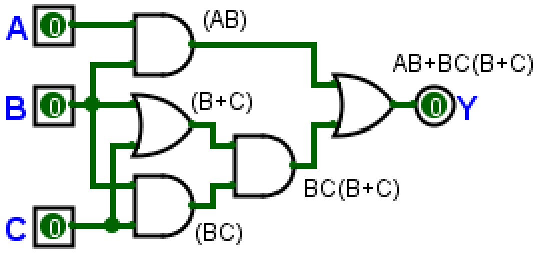


Figure 5.5: Example Circuit With Gate Outputs

This process leads to Equation 5.30.

$$AB + BC(B + C) = Y \tag{5.30}$$

5.5.3 *Starting From a Boolean Equation*

If a logic circuit's function is expressed as a Boolean equation, then algebraic methods can be applied to reduce the number of logic gates, resulting in a circuit that performs the same function with fewer components. As an example, Equation 5.31 simplifies the circuit found in Figure **??**.

$$\begin{aligned} AB + BC(B + C) &\quad \text{Original Expression} \\ AB + BBC + BCC &\quad \text{Distribute BC} \\ AB + BC + BC &\quad \text{Idempotence: BB=B and CC=C} \\ AB + BC &\quad \text{Idempotence: BC+BC=BC} \\ B(A + C) &\quad \text{Factor} \end{aligned} \tag{5.31}$$

The final expression, $B(A+C)$, requires only two gates, and is much simpler than the original, yet performs the same function. Such component reduction results in higher operating speed (less gate propagation delay), less power consumption, less cost to manufacture, and greater reliability.

As a second example, consider Equation 5.32.

$$A + AB = Y \tag{5.32}$$

This is simplified below.

$$\begin{array}{rr} A + AB & \text{Original Expression} \\ A(1+B) & \text{Factor} \\ A(1) & \text{Annihilation (1+B=1)} \\ A & \text{Identity A1=A} \end{array} \tag{5.33}$$

The original expression, $A + AB$ has been reduced to A so the original circuit could be replaced by a wire directly from input A to output Y. Equation 5.34 looks similar to Equation 5.32, but is quite different and requires a more clever simplification.

$$A + A'B = Y \tag{5.34}$$

This is simplified below.

$$\begin{array}{rr} A + A'B & \text{Original Expression} \\ A + AB + A'B & \text{Expand A to A+AB (Absorption)} \\ A + B(A + A') & \text{Factor B out of the last two terms} \\ A + B(1) & \text{Complement Property} \\ A + B & \text{Identityt: B(1)=B} \end{array} \tag{5.35}$$

Note how the Absorption Property ($A + AB = A$) is used to "un-simplify" the first A term, changing A into $A + AB$. While this may seem like a backward step, it ultimately helped to reduce the expression to something simpler. Sometimes "backward" steps must be taken to achieve the most elegant solution. Knowing when to take such a step is part of the art of algebra.

As another example, simplify this POS expression equation:

$$(A+B)(A+C) = Y \tag{5.36}$$

This is simplified below.

$$\begin{aligned}
(A+B)(A+C) &\qquad \text{Original Expression} \\
AA+AC+AB+BC &\qquad \text{Distribute A+B} \\
A+AC+AB+BC &\qquad \text{Idempotence: AA=A} \\
A+AB+BC &\qquad \text{Absorption: A+AC=A} \\
A+BC &\qquad \text{Absorption: A+AB=A}
\end{aligned} \tag{5.37}$$

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

5.5.4 *Practice Problems*

Table 5.14 shows a Boolean expression on the left and its simplified version on the right. This is provided for practice in simplifying expressions using algebraic methods.

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

Table 5.14: Simplifying Boolean Expressions

6

KARNAUGH MAPS

6.1 INTRODUCTION

Maurice Karnaugh developed this process at Bell Labs in 1953 while designing switching circuits for landline telephones.

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

In general, using Boolean Algebra is the easiest way to simplify a circuit involving one to three input variables. For four input variables, Boolean algebra becomes tedious and Karnaugh maps are both faster and easier (and are less prone to error). However, Karnaugh maps become rather complex with five input variables and are generally too difficult to use above six variables (with more than four input variables, a Karnaugh map uses multiple dimensions that become very challenging to manipulate). For five or more input variables, circuit simplification should be done by Quine-McClusky methods (page **??**) or the use of CAT (page 154).

In theory, any of the methods will work for any number of variables; however, as a practical matter, the guidelines presented in Table 6.1 work well. Normally, there is no need to resort to CAT to simplify a simple equation involving two or three variables since it is much quickly to use either Boolean Algebra or Karnaugh maps. However, for more complex input/output combinations, then CAT become essential to both speed the process and improve accuracy.

Variables	Algebra	K-Map	Quine-McClusky	CAT
1-2	X			
3	X	X		
4	X	X		
5-6		X	X	
7-8			X	X
¿8				X

Table 6.1: Circuit Simplification Methods

6.2 READING KARNAUGH MAPS

Following are four different ways to represent the same thing, a two-input digital logic function.

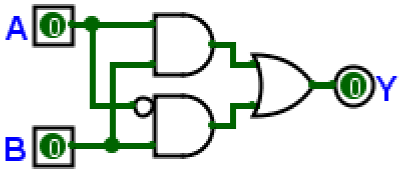


Figure 6.1: Simple Circuit For K-Map

$$AB + A'B = Y \tag{6.1}$$

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

Table 6.2: Truth Table for Simple Circuit

B \ A	0	1
0	0 $_{00}$	0 $_{02}$
1	1 $_{01}$	1 $_{03}$

Figure 6.2: Karnaugh map for Simple Circuit

Karnaugh maps frequently include the minterm numbers in each cell to aid in placing variables.

First is a circuit diagram, followed by its Boolean equation, truth table, and, finally, Karnaugh map. Think of a Karnaugh map as simply a rearranged truth table; but simplifying a three or four input circuit using a Karnaugh map is much easier and more accurate than with either a truth table or Boolean equation.

6.3 DRAWING TWO-VARIABLE KARNAUGH MAPS

Truth Table 6.3 and Karnaugh map 6.3 illustrate the relationship between these two representations of the same circuit using Greek symbols.

Inputs		Output
A	B	Y
0	0	α
0	1	β
1	0	γ
1	1	δ

Table 6.3: Truth Table with Greek Letters

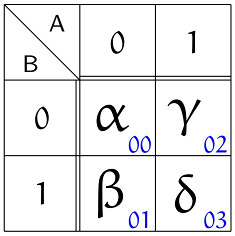


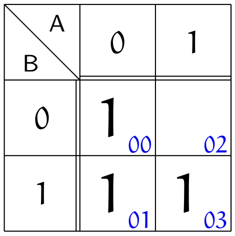
Figure 6.3: Karnaugh map With Greek Letters

On the Karnaugh map, all of the possible values for input A are listed across the top of the map and values for input B are listed down the left side. Thus, to find the output for $A = 0$; $B = 0$, look for the cell where those two quantities intersect; which is output α in the example. It should be clear how all four data squares on the Karnaugh map correspond to their equivalent rows (the minterms) in the Truth Table.

Truth Table 6.4 and Karnaugh map 6.4 illustrate another example of this relationship.

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

Table 6.4: Truth Table for Two-Input Circuit



B \ A	0	1
0	1 $_{00}$	$_{02}$
1	1 $_{01}$	1 $_{03}$

Figure 6.4: Karnaugh map For Two-Input Circuit

Karnaugh maps usually do not include zeros to decrease the chance for error.

6.4 DRAWING THREE-VARIABLE KARNAUGH MAPS

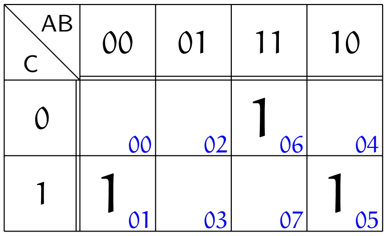
Consider Equation 6.2.

$$ABC' + AB'C + A'B'C = Y \tag{6.2}$$

Table 6.5 and Karnaugh map 6.5 represent this equation.

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

Table 6.5: Truth Table for Three-Input Circuit



C \ AB	00	01	11	10
0	$_{00}$	$_{02}$	1 $_{06}$	$_{04}$
1	1 $_{01}$	$_{03}$	$_{07}$	1 $_{05}$

Figure 6.5: Karnaugh map for Three-Input Circuit

In Karnaugh map 6.5 all possible values for inputs A and B are listed across the top of the map while input C is listed down the left side. Therefore, minterm m_{05}, in the lower right corner of the

Karnaugh map, is for $A = 1$; $B = 0$; $C = 1$, ($AB'C$), one of the *True* terms in the original equation.

6.4.1 *The Gray Code*

It should be noted that the values across the top of the Karnaugh map are not in binary order. Instead, those values are in “Gray Code” order. Gray code is essential for a Karnaugh map since the values for adjacent cells must change by only one bit. Constructing the Gray Code for three, four, and five variables is covered on page 59; however, for the Karnaugh maps used in this chapter, it is enough to know the two-bit Gray code: 00, 01, 11, 10.

6.5 DRAWING FOUR-VARIABLE KARNAUGH MAPS

Consider Equation 6.3.

$$ABCD' + AB'CD + A'B'CD = Y \tag{6.3}$$

Table 6.6 and the Karnaugh map in Figure 6.6 illustrate this equation.

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

Table 6.6: Truth Table for Four-Input Circuit

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

Figure 6.6: K-Map For Four Input Circuit

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

It is easy to indicate the minterms that are *True* on a Karnaugh map if Sigma Notation is available since the numbers following the Sigma sign are the minterms. As an example, Equation 6.4 creates the Karnaugh map in Figure 6.7.

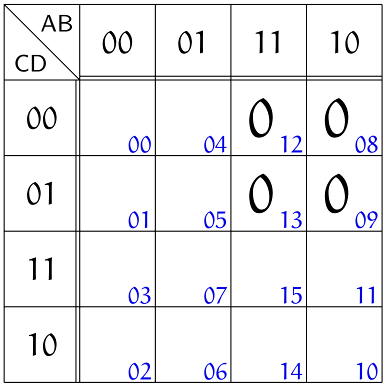
$$\int (A,B,C,D) = \sum (0,1,2,4,5) \tag{6.4}$$

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

Figure 6.7: K-Map For Sigma Notation

It is also possible to map values if the circuit is represented in Pi notation; but remember that maxterms indicate where zeros are placed on the Karnaugh map and the simplified circuit would actually be the inverse of the needed circuit. As an example, Equation 6.5 creates the Karnaugh map at Figure 6.8.

$$\int (A,B,C,D) = \prod (8,9,12,13) \tag{6.5}$$



AB CD	00	01	11	10
00	00	04	0 12	0 08
01	01	05	0 13	0 09
11	03	07	15	11
10	02	06	14	10

Figure 6.8: K-Map For PI Notation

To simplify this map, the designer could place ones in all of the empty cells and then simplify the "ones" circuit using the techniques explained below. Alternatively, the designer could also simplify the map by combining the zeros as if they were ones, and then finding the DeMorgan inverse (page 83) of that simplification. As an example, the maxterm Karnaugh map above would simplify to $\overline{AC'}$, and the DeMorgan equivalent for that is $A' + C$, which is the minterm version of the simplified circuit.

6.6 SIMPLIFYING GROUPS OF TWO

To simplify a Boolean equation using a Karnaugh map, start by creating the Karnaugh map, indicating the input variable combinations that lead to a *True* output for the circuit. Equations 6.6 and 6.7 are for the same circuit and the Karnaugh map in Figure 6.9 was built from these equations.

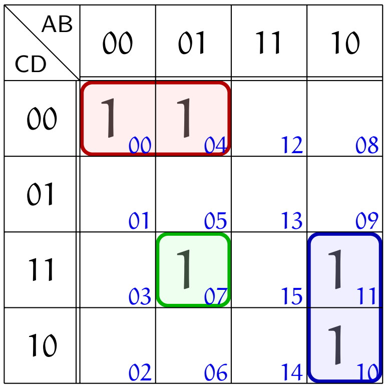
$$A'B'C'D' + A'BC'D' + A'BCD + AB'CD' + AB'CD \tag{6.6}$$

$$\int(A,B,C,D) = \sum(0,4,7,10,11) \tag{6.7}$$

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

Figure 6.9: K-Map for Groups of Two: Ex 1

Once the *True* outputs are indicated on the Karnaugh map, mark any groups of ones that are adjacent to each other, either horizontally or vertically (but not diagonally). Also, mark any ones that are "left over" and are not adjacent to any other ones, as illustrated in the Karnaugh map in Figure 6.16.



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

Figure 6.10: K-Map for Groups of Two: Ex 1, Solved

Notice the group in the top-left corner (minterms 00 and 04). This group includes the following two input combinations: $A'B'C'D' + A'BC'D'$. In this expression, the B and B' terms can be removed by the Complement Property (page 77); so this group reduces to $A'C'D'$. To simplify this expression by inspecting the Karnaugh map, notice that the variable A is zero for both of these minterms; therefore, A' must be part of the final expression. In the same way, variables C and D are zero for both terms; therefore, $C'D'$ must be part of the final expression. Since variable B changes it can be ignored when forming the simplified expression.

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

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

When finished, the original equation reduces to Equation 6.8.

$$A'C'D' + AB'C + A'BCD \tag{6.8}$$

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

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

A second example is illustrated in Equation 6.9 and the Karnaugh map in Figure 6.17.

$$\begin{aligned} &A'B'C'D + A'B'CD + A'BCD' + \\ &ABC'D + ABCD' + AB'C'D' = Y \end{aligned} \tag{6.9}$$

$$\int (A, B, C, D) = \sum (1, 3, 6, 8, 13, 14) \tag{6.10}$$

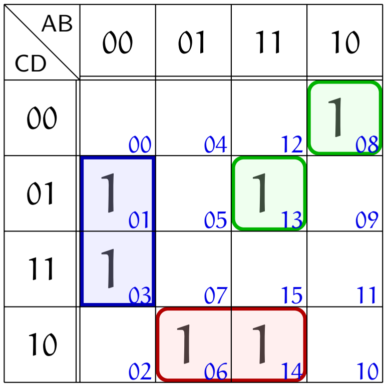


Figure 6.11: K-Map Solving Groups of Two: Example 2

All groups of adjacent ones have been marked, so this circuit can be simplified by looking for groups of two. Starting with minterms 01 and 03, $A'B'C'D + A'B'CD$ simplifies to $A'B'D$. Minterms 06 and 14 simplifies to BCD'. The other two marked minterms are not adjacent to any others, so they cannot be simplified. Each of the marked terms are prime implicants; and since they are not subsumed by any other implicants, they are essential prime implicants. Equation 6.17 is the simplified solution.

$$ABC'D + AB'C'D' + A'B'D + BCD' = Y \tag{6.11}$$

6.7 SIMPLIFYING LARGER GROUPS

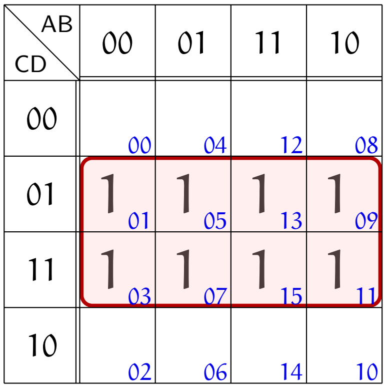
When simplifying Karnaugh maps, it is most efficient to find groups of 16, 8, 4, and 2 adjacent ones, in that order. In general, the larger the group, the simpler the expression becomes; so one large group is preferable to two smaller groups. However, remember that any group can only use ones that are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

6.7.1 *Groups of 16*

Groups of 16 reduce to a constant output of one. This is because if a circuit is built such that every possible combination of four inputs yields a *True* output, then the circuit is unnecessary and can be replaced by a wire. There is no example Karnaugh map posted here to illustrate a circuit like this because if every cell in a Karnaugh map contains a one, then the circuit is unnecessary. By the same token, any Karnaugh map that contains only zeros indicates that the circuit would never output a *True* condition so the circuit is unnecessary.

6.7.2 *Groups of Eight*

Groups of eight simplifies to a single output variable. Consider the Karnaugh map in Figure 6.12.



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

Figure 6.12: K-Map Solving Groups of 8

The expression for row two is: $A'B'C'D + A'BC'D + ABC'D + AB'C'D$. The term $C'D$ is constant in this group, while A and B change, so this one line would simplify to $C'D$. The expression for row three is: $A'B'CD + A'BCD + ABCD + AB'CD$. The term CD is constant in this group, so this one line would simplify to CD. Then, if the two rows are combined: $C'D + CD$, C and C' are dropped by the complement property and the circuit simplifies to D. To put this another way, since the only term in this group of eight that never changes is D, then Equation 6.12 is the simplified solution.

$$D = Y \tag{6.12}$$

This Karnaugh map also provides a good example of prime implicants that are not essential. Consider row two of the map. Minterms 01 and 05 form a Prime Implicant for this circuit since it is a group of two; however, that group was subsumed by the group of eight that was formed with the next row. Since every cell in the group of two is also present in the group of eight, then the group of two is not essential to the final circuit simplification. While this may seem to be rather obvious, it is important to remember that frequently implicants are formed that are not essential and they can be ignored. This concept will come up again when using the Quine-McCluskey Simplification method on page **??**.

6.7.3 *Groups of Four*

Groups of four can form as a single row, a single column, or a square. In any case, the four cells will simplify to a two-variable expression. Consider the Karnaugh map in Figure 6.13

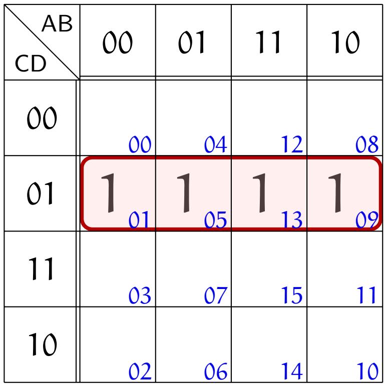

AB
CD
00
01
11
10
00
01
11
10
00
04
12
08
1
01
1
05
1
13
1
09
03
07
15
11
02
06
14
10

Figure 6.13: K-Map Solving Groups of Four, Example 1

Since the A and B variables can be removed due to the Complement Property, Equation 6.13 shows the simplified solution.

$$C'D = Y \tag{6.13}$$

The Karnaugh map in Figure 6.14 is a second example.

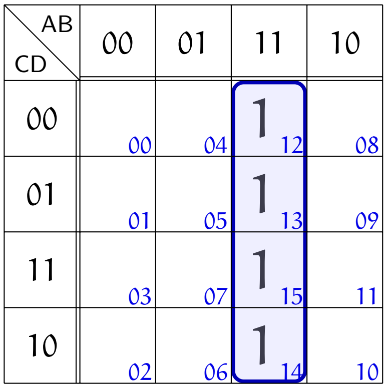
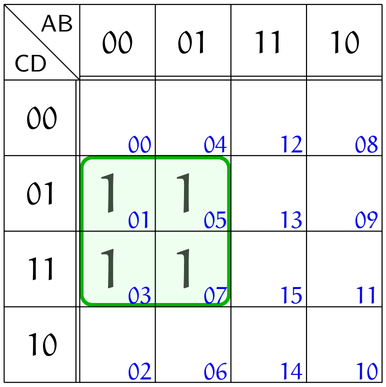

AB
CD
00
01
11
10
00
01
11
10
00
04
1
12
08
01
05
1
13
09
03
07
1
15
11
02
06
1
14
10

Figure 6.14: K-Map Solving Groups of Four, Example 2

Since the C and D variables can be removed due to the Complement Property, Equation 6.14 shows the simplified circuit.

$$AB = Y \tag{6.14}$$

The Karnaugh map in Figure 6.15 is an example of a group of four that forms a square.



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

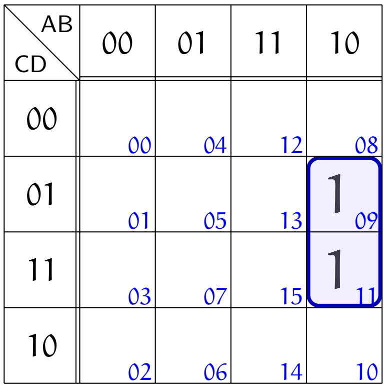
Figure 6.15: K-Map Solving Groups of Four, Example 3

Since the B and C variables can be removed due to the Complement Property, Equation 6.15 shows the simplified circuit.

$$A'D = Y \tag{6.15}$$

6.7.4 *Groups of Two*

Groups of two will simplify to a three-variable expression. The Karnaugh map in Figure 6.16 is one example of a group of two.



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

Figure 6.16: K-Map Solving Groups of Two, Example 1

Since C is the only variable that can be removed due to the Complement Property, the above circuit simplifies to Equation 6.16.

$$AB'D = Y \tag{6.16}$$

As a second example, consider the Karnaugh map in Figure 6.17.

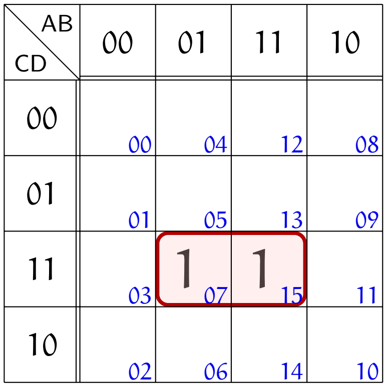

AB
CD
00
01
11
10
00
01
11
10
00
04
12
08
01
05
13
09
1
1
03
07
15
11
02
06
14
10

Figure 6.17: K-Map Solving Groups of Two, Example 2

Equation 6.17 is the simplified equation for the Karnaugh map in 6.17.

$$BCD = Y \tag{6.17}$$

6.8 OVERLAPPING GROUPS

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

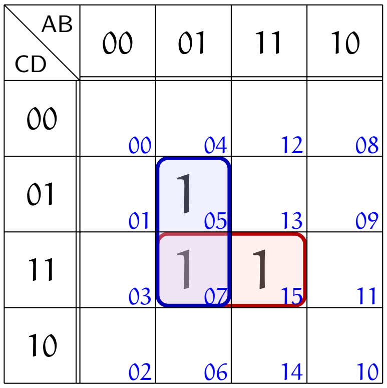

AB
CD
00
01
11
10
00
01
11
10
00
04
12
08
1
01
05
13
09
1
1
03
07
15
11
02
06
14
10

Figure 6.18: K-Map Overlapping Groups, Example 1

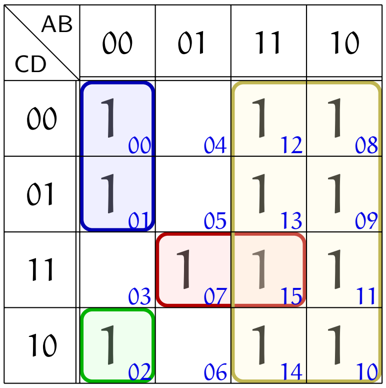
The one in cell $A'BCD$ (minterm 07) can be grouped with either the horizontal or vertical group (or both). This creates the following three potential simplified circuits:

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

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

$$A'BD + BCD = Y \tag{6.18}$$

Karnaugh map 6.19 is a more complex example.



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

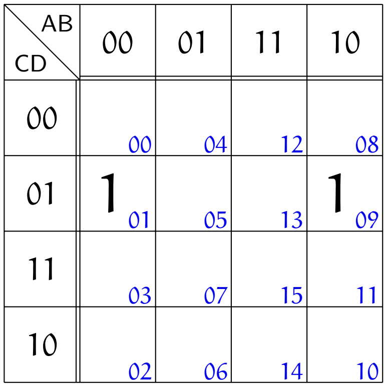
Figure 6.19: K-Map Overlapping Groups, Example 2

The circuit represented by this Karnaugh map 6.19 would simplify to:

$$A + A'B'C' + BCD + A'B'CD' = Y \tag{6.19}$$

6.9 WRAPPING GROUPS

A Karnaugh map also "wraps" around the edges (top/bottom and left/right), so groups can be formed around the borders. It is almost like the Karnaugh map is on some sort of weird sphere where every edge touches the edge across from it (but not diagonal corners). The Karnaugh map in Figure 6.20 is an example.



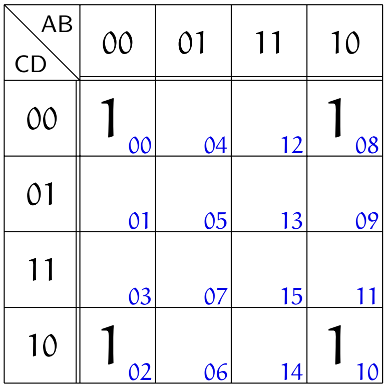
CD \ AB	00	01	11	10
00	00	04	12	08
01	1 01	05	13	1 09
11	03	07	15	11
10	02	06	14	10

Figure 6.20: K-Map Wrapping Groups Example 1

The two ones on this map can be grouped “around the edge” to form Equation 6.20.

$$B'C'D = Y \tag{6.20}$$

The Karnaugh map in Figure 6.21 is another example of wrapping.



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

Figure 6.21: K-Map Wrapping Groups Example 2

The ones on the above map can be formed into a group of four and simplify into Equation 6.21.

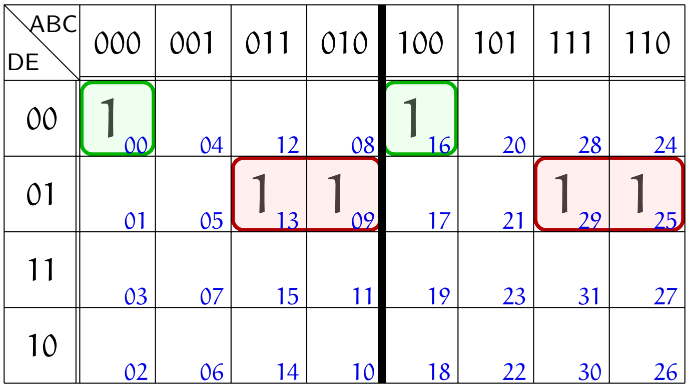
$$B'D' = Y \tag{6.21}$$

6.10 KARNAUGH MAPS FOR FIVE-VARIABLE INPUTS

It is possible to create a Karnaugh map for circuits with five input variables; however, the map must be simplified as a three-dimensional object so it is more complex than the maps described above. As an example, imagine a circuit that is defined by Equation 6.22.

$$\int(A,B,C,D,E) = \sum(0,9,13,16,25,29) \tag{6.22}$$

The Karnaugh map in Figure 6.22 would be used to simplify that circuit.



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

Figure 6.22: K-Map for Five Variables, Example 1

This map lists variables A, B, and C across the top row with D and E down the left column. Variables B and C are in Gray Code order, while variable A is zero on the left side of the map and one on the right. To simplify the circuit, the map must be imagined to be a three-dimensional item; so the map would be cut along the heavy line between minterms 08 and 16 (where variables ABC change from 010 to 100) and then the right half would slide under the left half in such a way that minterm 16 ends up directly under minterm 00.

By arranging the map in three dimensions there is only one bit different between minterms 00 and 16: bit A changes from zero to one while the bits B and C remain at zero. Those two cells then form a group of two and would be $A'B'C'D'E' + AB'C'D'E'$. Since variable A and A' are both present in this expression, and none of the other variables change, it can be simplified to: $B'C'D'E'$. This process is exactly the same as for a two-dimension Karnaugh map, except that adjacent cells may include those above or below each other (though diagonals are still not simplified).

Next, consider the group formed by minterms 09, 13, 25, and 29. The expression for that group is $(A'BC'D'E + A'BCD'E + ABC'D'E + ABCD'E)$. The variables A and C can be removed from the simplified expression by the Complement Property, leaving: $BD'E$ for this group. Equation 6.23 is the final simplified expression for this circuit.

$$(B'C'D'E') + (BD'E) = Y \tag{6.23}$$

The Karnaugh map in Figure 6.23 is a more complex example.

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

Figure 6.23: K-Map Solving for Five Variables, Example 2

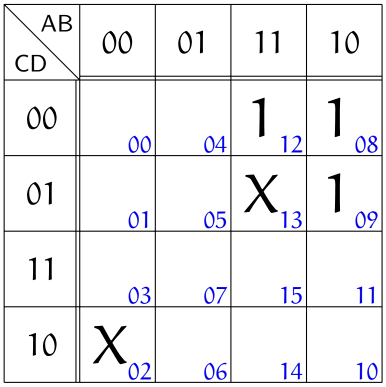
On the map in Figure 6.23, the largest group would be minterms 13-15-29-31, so they should be combined first. Minterms 05-13, 27-31, and 00-02 would form groups of two. Finally, even though Karnaugh maps wrap around the edges, minterm 00 will not group with minterm 24 since they are on different layers (notice that two bits, A and B, change between those two minterms, so they are not adjacent); therefore, minterms 00 and 24 will not group with any other minterms. Equation 6.24 is the simplified expression for this circuit.

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

$$(A'B'C'E') + (A'CD'E) + (BCD) + (ABDE) + (ABC'D'E') = Y \tag{6.24}$$

6.11 "DON'T CARE" TERMS

Occasionally, a circuit designer will run across a situation where the output for a particular minterm makes no difference in the circuit; so that minterm is considered "don't care;" that is, it can be either one or zero without having any effect on the entire circuit. As an example, consider a circuit that it designed to work with BCD (page 52) values. In that system, minterms 10-15 do not exist, so they would be considered "don't care." On a Karnaugh map, "don't care" terms are indicated using several different methods, but the two most common are a dash or an "x". When simplifying a Karnaugh map that contains "don't care" values, the designer can choose to consider those values as either zero or one, whichever makes simplifying the map easier. Consider the following Karnaugh map:

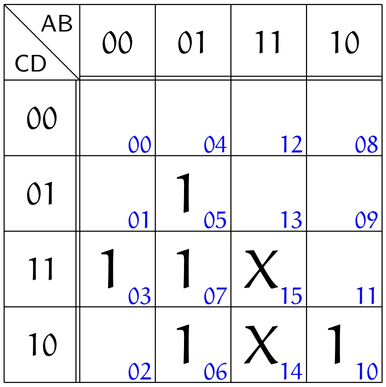


AB / CD	00	01	11	10
00	$_{00}$	$_{04}$	1_{12}	1_{08}
01	$_{01}$	$_{05}$	X_{13}	1_{09}
11	$_{03}$	$_{07}$	$_{15}$	$_{11}$
10	X_{02}	$_{06}$	$_{14}$	$_{10}$

Figure 6.24: K-Map With "Don't Care" Terms, Example 1

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

The Karnaugh map in Figure 6.25 is another example circuit with "don't care" terms:



AB / CD	00	01	11	10
00	$_{00}$	$_{04}$	$_{12}$	$_{08}$
01	$_{01}$	1_{05}	$_{13}$	$_{09}$
11	1_{03}	1_{07}	X_{15}	$_{11}$
10	$_{02}$	1_{06}	X_{14}	1_{10}

Figure 6.25: K-Map With "Don't Care" Terms, Example 2

This circuit would simplify to Equation 6.25.

$$ACD' + BC + A'BC + A'CD = Y \tag{6.25}$$

6.12 KARNAUGH MAP SIMPLIFICATION SUMMARY

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

1. Create the map and plot all ones from the truth table output.

2. Circle all groups of 16. These will reduce to a constant output of one and the entire circuit is unnecessary.
3. Circle all groups of eight. These will reduce to a one-variable expression.
4. Circle all groups of four. These will reduce to a two-variable expression. The ones can be either horizontal, vertical, or in a square.
5. Circle all groups of two. These will reduce to a three-variable expression. The ones can be either horizontal or vertical.
6. Circle all ones that are not in any other group. These do not reduce and will result in a four-variable expression.
7. All ones must be circled at least one time.
8. Groups can overlap.
9. If ones are in more than one group, they can be considered part of either, or both, groups.
10. Groups can wrap around the edges to the other edge of the map.
11. "Don't Care" terms can be considered either one or zero, whichever makes the map simpler.

6.13 PRACTICE PROBLEMS

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

Table 6.7: Karnaugh maps Practice Problems

6.14 REED-MÜLLER LOGIC

6.15 INTRODUCTION

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

Consider the Truth Table 6.8.

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

Table 6.8: Truth Table for Checkerboard Pattern

This pattern is easy to recognize as an XOR gate. The Karnaugh map for Truth Table 6.8 is in Figure 6.26 (the zeros have been omitted and the cells with ones have been shaded to emphasize the checkerboard pattern):

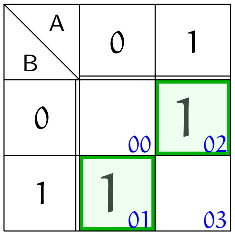


Figure 6.26: Reed-Müller Two-Variable Example

Equation 6.26 describes this circuit.

$$AB' + A'B = Y \tag{6.26}$$

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

6.16 ZERO IN FIRST CELL

Karnaugh maps with a zero in the first cell (that is, in a four-variable map, $A'B'C'D'$ is *False*) are simplified in a slightly different manner than those with a one in that cell. This section describes the technique used for maps with a zero in the first cell and Section 6.17 (page 138) describes the technique for maps with a one in the first cell.

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

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

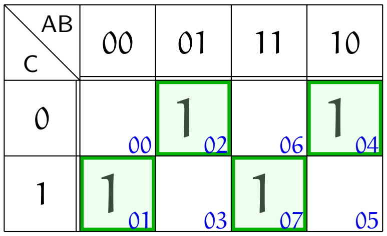
6.16.1 *Two-Variable Circuit*

In Karnaugh map 6.26, it is not possible to group the ones. Since both A and B are *True* in at least one cell, the equation for that circuit is:

$$A \oplus B = Y \tag{6.27}$$

6.16.2 *Three-Variable Circuit*

The Karnaugh map in Figure 6.27 is for a circuit containing three inputs.



AB / C	00	01	11	10
0	00	1 02	06	1 04
1	1 01	03	1 07	05

Figure 6.27: Reed-Müller Three-Variable Example 1

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

$$A \oplus B \oplus C = Y \tag{6.28}$$

The Karnaugh map in Figure 6.28 is more interesting.

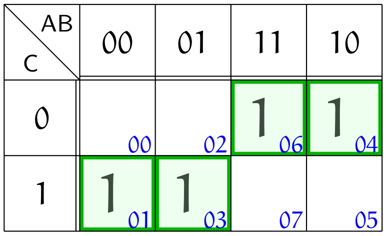

AB
C
00
01
11
10
0
1
1
1
1
1
00
02
06
04
01
03
07
05

Figure 6.28: Reed-Müller Three-Variable Example 2

In this Karnaugh map, the ones form two groups in a checkerboard pattern. If this map were to be simplified using common techniques it would form Equation 6.29.

$$AC' + A'C = Y \tag{6.29}$$

If realized, this circuit would contain two two-input AND gates joined by a two-input OR gate. However, this equation can be simplified using the Reed-Müller technique. For the group in the upper right corner A is a constant one and for the group in the lower left corner C is a constant one. The equation simplifies to 6.30.

$$A \oplus C = Y \tag{6.30}$$

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

6.16.3 *Four-Variable Circuit*

The Karnaugh map in Figure 6.29 is for a circuit containing four variable inputs.

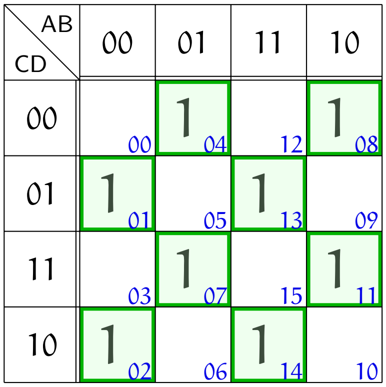
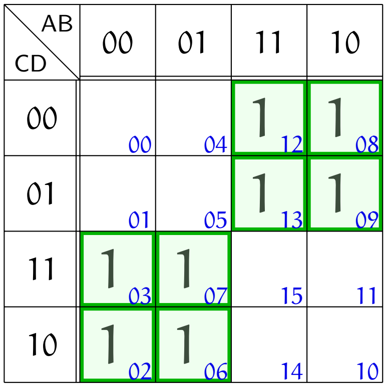

AB
CD
00
01
11
10
00
01
11
10
1
1
1
1
1
1
1
1
00
04
12
08
01
05
13
09
03
07
15
11
02
06
14
10

Figure 6.29: Reed-Müller Four-Variable Example 1

In this Karnaugh map, it is not possible to group the ones. Since A, B, C, and D are all *True* in at least one cell, Equation 6.31 would describe the circuit.

$$A \oplus B \oplus C \oplus D = Y \tag{6.31}$$

Following are more interesting four-variable Karnaugh maps with groups of ones:

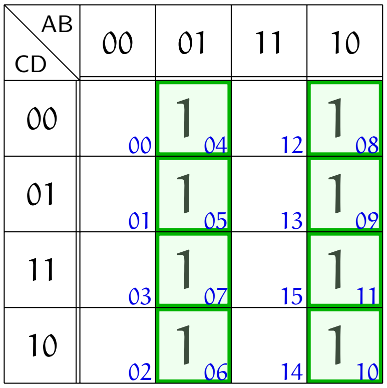


CD \ AB	00	01	11	10
00	00	04	1 12	1 08
01	01	05	1 13	1 09
11	1 03	1 07	15	11
10	1 02	1 06	14	10

Figure 6.30: Reed-Müller Four-Variable Example 2

In the Karnaugh map in Figure 6.30, the group of ones in the upper right corner has a constant one for A and the group in the lower left corner has a constant one for C, so Equation 6.32 would describe the circuit.

$$A \oplus C = Y \tag{6.32}$$

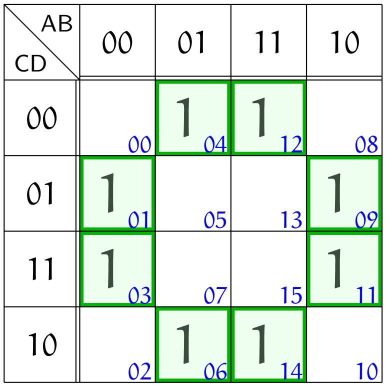


CD \ AB	00	01	11	10
00	00	1 04	12	1 08
01	01	1 05	13	1 09
11	03	1 07	15	1 11
10	02	1 06	14	1 10

Figure 6.31: Reed-Müller Four-Variable Example 3

In the Karnaugh map in Figure 6.31, the group of ones in the first shaded column has a constant one for B and in the second shaded column have a constant one for A, so Equation 6.33 describes this circuit.

$$A \oplus B = Y \tag{6.33}$$



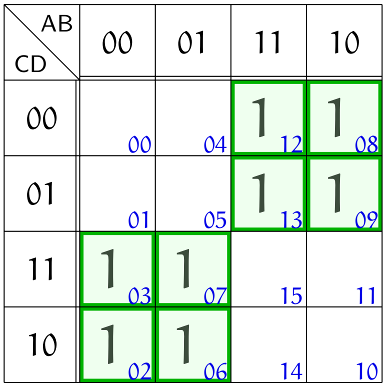
AB / CD	00	01	11	10
00	00	1 04	1 12	08
01	1 01	05	13	1 09
11	1 03	07	15	1 11
10	02	1 06	1 14	10

Figure 6.32: Reed-Müller Four-Variable Example 4

Keep in mind that groups can wrap around the edges in a Karnaugh map. The group of ones in columns 1 and 4 combine and has a constant one for D. The group of ones in rows 1 and 4 combine and has a constant one for B, so Equation 6.34 describes this circuit.

$$B \oplus D = Y \tag{6.34}$$

It is interesting that all of the above examples used XOR gates to combine the constant *True* variable found in groups of ones; however, it would yield the same result if XOR gates combined the constant *False* variable found in groups of ones. The designer could choose either, but must be consistent. For example, consider the Karnaugh map in Figure 6.33.



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

Figure 6.33: Reed-Müller Four-Variable Example 5

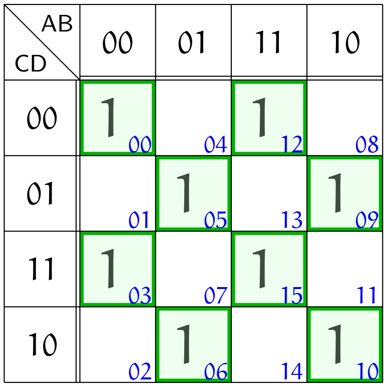
When this Karnaugh map was simplified earlier (Figure 6.30), the constant *True* for the group in the upper right corner, A, and lower left corner, C, was used; however, the constant *False* in the lower left corner, A, and upper right corner, C, would give the same result. Either way yields Equation 6.35.

To avoid confusion when simplifying these maps it is probably best to always use the constant True *terms.*

$$A \oplus C = Y \tag{6.35}$$

6.17 ONE IN FIRST CELL

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

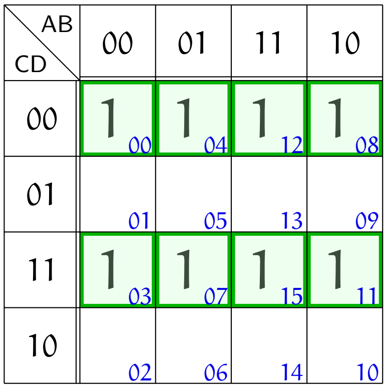


AB / CD	00	01	11	10
00	1 00	04	1 12	08
01	01	1 05	13	1 09
11	1 03	07	1 15	11
10	02	1 06	14	1 10

Figure 6.34: Reed-Müller Four-Variable Example 6

In the Karnaugh map in Figure 6.34 it is not possible to group the ones. Since A, B, C, and D are all *True* in at least one cell, they would all appear in the final equation; however, since there is a one in the first cell, then two of the terms must be combined with an XNOR gate. Here is one possible solution:

$$A \odot B \oplus C \oplus D = Y \tag{6.36}$$

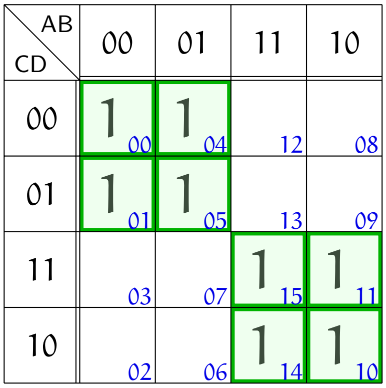


AB / CD	00	01	11	10
00	1 00	1 04	1 12	1 08
01	01	05	13	09
11	1 03	1 07	1 15	1 11
10	02	06	14	10

Figure 6.35: Reed-Müller Four-Variable Example 7

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

$$C \odot D = Y \tag{6.37}$$



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

Figure 6.36: Reed-Müller Four-Variable Example 8

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

$$A \odot C = Y \tag{6.38}$$

7

ADVANCED SIMPLIFYING METHODS

7.1 QUINE-MCCLUSKEY SIMPLIFICATION METHOD

7.1.1 *Introduction*

This method was developed by W.V. Quine and Edward J. McCluskey and is sometimes called the method of prime implicants.

When a Boolean equation involves five or more variables it becomes very difficult to solve using standard algebra techniques or Karnaugh maps; however, the Quine-McCluskey algorithm can be used to solve these types of Boolean equations.

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

$$ABC + ABC' = AB \tag{7.1}$$

The Quine-McCluskey method looks for expressions that differ by only a single variable and combines them. Then it looks at the combined expressions to find those that differ by a single variable and combines them. The process continues until there are no expressions remaining to be combined.

7.1.2 *Example One*

7.1.2.1 *Step 1: Create the Implicants*

Equation 7.2 is the Sigma representation of a Boolean equation.

$$\int(A, B, C, D) = \sum(0, 1, 2, 5, 6, 7, 9, 10, 11, 14) \tag{7.2}$$

Truth Table 7.1 shows the input variables for the *True* minterm values.

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

Table 7.1: Quine-McCluskey Ex 1: Minterm Table

To simplify this equation, the minterms that evaluate to *True* (as listed above) are first placed in a minterm table so that they form sections that are easy to combine. Each section contains only the minterms that have the same number of ones. Thus, the first section contains all minterms with zero ones, the second section contains the minterms with one one, and so forth. Truth Table 7.2 shows the minterms rearranged appropriately.

Number of 1's	Minterm	Binary
0	0	0000
1	1	0001
	2	0010
2	5	0101
	6	0110
	9	1001
	10	1010
3	7	0111
	11	1011
	14	1110

Table 7.2: Quine-McCluskey Ex 1: Rearranged Table

Start combining minterms with other minterms to create Size Two Implicants (called that since each implicant combines two minterms), but only those terms that vary by a single binary digit can be combined. When two minterms are combined, the binary digit that is different between the minterms is replaced by a dash, indicating that the digit does not matter. For example, 0000 and 0001 can be combined to form 000−. The table is modified to add a Size Two Implicant column that indicates all of the combined terms. Note that every minterm must be compared to every other minterm so all possible implicants are formed. This is easier than it sounds, though, since terms in section one must be compared only with section two, then those in section two are compared with section three, and so forth, since each section differs from the next by a single binary digit. The Size Two Implicant column contains the combined binary form along with the numbers of the minterms used to create that implicant. It is also important to mark all minterms that are used to create the Size Two Implicants since allowance must be made for any not combined. Therefore, in the following table, as a minterm is used it is also struck through. Table 7.3 shows the Size Two Implicants that were found.

1's	Mntrm	Bin	Size 2
0	~~0~~	0000	000- (0,1)
1	~~1~~	0001	00-0 (0,2)
	~~2~~	0010	0-01 (1,5)
2	~~5~~	0101	-001 (1,9)
	~~6~~	0110	0-10 (2,6)
	~~9~~	1001	-010 (2,10)
	~~10~~	1010	01-1 (5,7)
3	~~7~~	0111	011- (6,7)
	~~11~~	1011	-110 (6,14)
	~~14~~	1110	10-1 (9,11)
			101- (10,11)
			1-10 (10,14)

Table 7.3: Quine-McCluskey Ex 1: Size 2 Implicants

All of the Size Two Implicants can now be combined to form Size Four Implicants (those that combine a total of four minterms). Again, it is essential to only combine those with only a single binary digit difference. For this step, the dash can be considered the same as a single binary digit, as long as it is in the same place for both implicants. Thus, −010 and −110 can be combined to −−10, but −010 and 0−00 cannot be combined since the dash is in different places in those numbers. It helps to match up the dashes first and then look at the binary digits. Again, as the various size-two implicants are used they are marked; but notice that a single size-four implicant actually combines four size-two implicants. Table 7.4 shows the Size Four Implicants.

1's	Mntrm	Bin	Size 2	Size 4
0	~~0~~	0000	000- (0,1)	–10 (2,10,6,14)
1	~~1~~	0001	00-0 (0,2)	
	~~2~~	0010	0-01 (1,5)	
2	~~5~~	0101	-001 (1,9)	
	~~6~~	0110	~~0-10 (2,6)~~	
	~~9~~	1001	~~-010 (2,10)~~	
	~~10~~	1010	01-1 (5,7)	
3	~~7~~	0111	011- (6,7)	
	~~11~~	1011	~~-110 (6,14)~~	
	~~14~~	1110	10-1 (9,11)	
			101- (10,11)	
			~~1-10 (10,14)~~	

Table 7.4: Quine-McCluskey Ex 1: Size 4 Implicants

None of the terms can be combined any further. All of the minterms or implicants that are not marked are *Prime Implicants*. In the table above, for example, the Size Two Implicant 000– is a Prime Implicant. The Prime Implicants will be placed in a chart and further processed in the next step.

7.1.2.2 *Step 2: The Prime Implicant Table*

A *Prime Implicant Table* can now be constructed, as in Table 7.5. The prime implicants are listed down the left side of the table, the decimal equivalent of the minterms goes across the top, and the Boolean representation of the prime implicants is listed down the right side of the table.

	0	1	2	5	6	7	9	10	11	14	
000 – (0, 1)	X	X									A′B′C′
00 – 0 (0, 2)	X		X								A′B′D′
0 – 01 (1, 5)		X		X							A′C′D
–001 (1, 9)		X					X				B′C′D
01 – 1 (5, 7)				X		X					A′BD
011 – (6, 7)					X	X					A′BC
10 – 1 (9, 11)							X		X		AB′D
101 – (10, 11)								X	X		AB′C
– – 10 (2, 10, 6, 14)			X		X			X		X	CD′

Table 7.5: Quine-McCluskey Ex 1: Prime Implicants

An *X* marks the intersection where each minterm (on the top row) is used to form one of the prime implicants (in the left column). Thus, minterm 0 (or 0000) is used to form the prime implicant $000-(0,1)$ in row one and $00-0(0,2)$ in row two.

The Essential Prime Implicants can be found by looking for columns that contain only one *X*. The column for minterm 14 has only one *X*, in the last row, $--10(2,10,6,14)$; thus, it is an Essential Prime Implicant. That means that the term in the right column for the last row, CD', must appear in the final simplified equation. However, that term also covers the columns for 2, 6, and 10; so they can be removed from the table. The Prime Implicant table is then simplified to 7.6.

	0	1	5	7	9	11	
$000-\ (0,1)$	X	X					$A'B'C'$
$00-0\ (0,2)$	X						$A'B'D'$
$0-01\ (1,5)$		X	X				$A'C'D$
$-001\ (1,9)$		X			X		$B'C'D$
$01-1\ (5,7)$			X	X			$A'BD$
$011-\ (6,7)$				X			$A'BC$
$10-1\ (9,11)$					X	X	$AB'D$
$101-\ (10,11)$						X	$AB'C$

Table 7.6: Quine-McCluskey Ex 1: 1st Iteration

The various rows can now be combined in any order the designer desires. For example, if row $10-1(9,11)$, is selected as a required implicant in the solution, then minterms 9 and 11 are accounted for in the final equation, which means that all *X* marked in those columns can be removed. When that is done, then, rows $101-(10,11)$ and $10-1(9,11)$ no longer have any marks in the table, and they can be removed. Table 7.7 shows the last iteration of this solution.

	0	1	5	7	
$000-\ (0,1)$	X	X			$A'B'C'$
$00-0\ (0,2)$	X				$A'B'D'$
$0-01\ (1,5)$		X	X		$A'C'D$
$-001\ (1,9)$		X			$B'C'D$
$01-1\ (5,7)$			X	X	$A'BD$
$011-\ (6,7)$				X	$A'BC$

Table 7.7: Quine-McCluskey Ex 1: 2nd Iteration

The designer next decided to select $01-1(5,7)$, $A'BD$, as a required implicant. That will include minterms 5 and 7, and those columns may be removed along with rows $01-1(5,7)$, $A'BD$, and $011-(6,7)$, $A'BC$, as shown in Table 7.8.

	0	1	
$000-$ $(0,1)$	X	X	$A'B'C'$
$00-0$ $(0,2)$	X		$A'B'D'$
$0-01$ $(1,5)$		X	$A'C'D$
-001 $(1,9)$		X	$B'C'D$

Table 7.8: Quine-McCluskey Ex 1: 3rd Iteration

The last two minterms (0 and 1) can be covered by the implicant $000-(0,1)$, and that also eliminates the last three rows in the chart.

The original Boolean expression, then, has been simplified from ten minterms to Equation 7.3.

$$A'B'C' + A'BD + AB'D + CD' = Y \tag{7.3}$$

7.1.3 *Example Two*

7.1.3.1 *Step 1: Create the Implicants*

Given Equation 7.4, which is a Sigma representation of a Boolean equation.

$$\int(A,B,C,D,E,F) = \sum(0,1,8,9,12,13,14,15,32,33,37,39,48,56) \tag{7.4}$$

Truth Table 7.9 shows the *True* minterm values.

Minterm	A	B	C	D	E	F
0	0	0	0	0	0	0
1	0	0	0	0	0	1
8	0	0	1	0	0	0
9	0	0	1	0	0	1
12	0	0	1	1	0	0
13	0	0	1	1	0	1
14	0	0	1	1	1	0
15	0	0	1	1	1	1
32	1	0	0	0	0	0
33	1	0	0	0	0	1
37	1	0	0	1	0	1
39	1	0	0	1	1	1
48	1	1	0	0	0	0
56	1	1	1	0	0	0

Table 7.9: Quine-McCluskey Ex 2: Minterm Table

To simplify this equation, the minterms that evaluate to *True* are placed in a minterm table so that they form sections that are easy to combine. Each section contains only the minterms that have the same number of ones. Thus, the first section contains all minterms with zero ones, the second section contains the minterms with one one, and so forth. Table 7.10 shows the rearranged truth table.

Number of 1's	Minterm	Binary
0	0	000000
1	1	000001
	8	001000
	32	100000
2	9	001001
	12	001100
	33	100001
	48	110000
3	13	001101
	14	001110
	37	100101
	56	111000
4	15	001111
	39	100111

Table 7.10: Quine-McCluskey Ex 2: Rearranged Table

Start combining minterms with other minterms to create Size Two Implicants, as in Table 7.11.

1's	Mntrm	Bin	Size 2
0	~~0~~	000000	00000- (0,1)
1	~~1~~	000001	-000000 (0,32)
	~~8~~	001000	00-000 (0,8)
	~~32~~	100000	-00001 (1,33)
2	~~9~~	001001	00-001 (1,9)
	~~12~~	001100	10000- (32,33)
	~~33~~	100001	1-0000 (32,48)
	~~48~~	110000	00100- (8,9)
3	~~13~~	001101	001-00 (8,12)
	~~14~~	001110	100-01 (33,37)
	~~37~~	100101	001-01 (9,13)
	~~56~~	111000	00110- (12,13)
4	~~15~~	001111	0011-0 (12,14)
	~~39~~	100111	11-000 (48,56)
			1001-1 (37,39)
			0011-1 (13,15)
			00111- (14,15)

Table 7.11: Quine-McCluskey Ex 2: Size Two Implicants

All of the Size Two Implicants can now be combined to form Size Four Implicants, as in Table 7.12.

1's	Mntrm	Bin	Size 2	Size 4
0	~~0~~	000000	~~00000- (0,1)~~	-0000- (0,1,32,33)
1	~~1~~	000001	~~-000000 (0,32)~~	00-00- (0,1,8,9)
	~~8~~	001000	~~00-000 (0,8)~~	001-0- (8,9,12,13)
	~~32~~	100000	~~-00001 (1,33)~~	0011– (12,13,14,15)
2	~~9~~	001001	~~00-001 (1,9)~~	
	~~12~~	001100	~~10000- (32,33)~~	
	~~33~~	100001	1-0000 (32,48)	
	~~48~~	110000	~~00100- (8,9)~~	
3	~~13~~	001101	~~001-00 (8,12)~~	
	~~14~~	001110	100-01 (33,37)	
	~~37~~	100101	~~001-01 (9,13)~~	
	~~56~~	111000	~~00110- (12,13)~~	
4	~~15~~	001111	~~0011-0 (12,14)~~	
	~~39~~	100111	11-000 (48,56)	
			1001-1 (37,39)	
			~~0011-1 (13,15)~~	
			~~00111- (14,15)~~	

Table 7.12: Quine-McCluskey Ex 2: Size 4 Implicants

None of the terms can be combined any further. All of the minterms or implicants that are not struck through are *Prime Implicants*. In the table above, for example, $1 - 0000$ is a Prime Implicant. The Prime Implicants are next placed in a table and further processed.

7.1.3.2 *Step 2: The Prime Implicant Table*

A *Prime Implicant Table* can now be constructed, as in Table 7.13. The prime implicants are listed down the left side of the table, the decimal equivalent of the minterms goes across the top, and the Boolean representation of the prime implicants is listed down the right side of the table.

	0	1	8	9	12	13	14	15	32	33	37	39	48	56	
11 − 000 (48, 56)													X	X	ABD′D′F′
00 − 00 − (0, 1, 8, 9)	X	X	X	X											A′B′D′E′
1001 − 1 (37, 39)											X	X			AB′C′DF
1 − 0000 (32, 48)									X				X		AC′D′E′F′
0011 − − (12, 13, 14, 15)					X	X	X	X							A′B′CD
−0000 − (0, 1, 32, 33)	X	X							X	X					B′C′D′E′
001 − 0 − (8, 9, 12, 13)			X	X	X	X									A′B′CE′
100 − 01 (33, 37)										X	X				AB′C′E′F

Table 7.13: Quine-McCluskey Ex 2: Prime Implicants

In the above table, there are four columns that contain only one *X*: 14, 15, 39, and 56. The rows that intersect the columns at that mark are *Essential Prime Inplicants*, and their Boolean Expressions must appear in the final equation. Therefore, the final equation will contain, at a minimum: A′B′CD (row 5, covers minterms 14 and 15), AB′C′DF (row 3, covers minterm 39), and ABD′E′F′ (row 1, covers minterm 56). Since those expressions are in the final equation, the rows that contain those expressions can be removed from the chart in order to make further analysis less confusing.

Also, because the rows with Essential Prime Implicants are contained in the final equation, other minterms marked by those rows are covered and need no further consideration. For example, minterm 48 is covered by row one (used for minterm 56), so column 48 can be removed from the table. In a similar fashion, columns 12, 13, and 37 are covered by other minterms, so they can be removed from the table. Table 7.14 shows the next iteration of this process.

	0	1	8	9	32	33	
00 − 00 − (0, 1, 8, 9)	X	X	X	X			A′B′D′E′
1 − 0000 (32, 48)					X		AC′D′E′F′
−0000 − (0, 1, 32, 33)	X	X			X	X	B′C′D′E′
001 − 0 − (8, 9, 12, 13)			X	X			A′B′CE′
100 − 01 (33, 37)						X	AB′C′E′F

Table 7.14: Quine-McCluskey Ex 2: 1st Iteration

The circuit designer can select the next term to include in the final equation from any of the five rows still remaining in the chart; however, the first term (00 − 00−, or A′B′D′E′) would eliminate four columns, so that would be a logical next choice. When that term is selected for the final equation, then row one, 00 − 00−, can be removed from the chart; and columns 0, 1, 8, and 9 can be removed since those minterms are covered.

The minterms marked for row $001-0-(8,9,12,13)$ are also covered, so this row can be removed. Table 7.15 shows the next iteration.

	32	33	
$1-0000\ (32,48)$	X		$AC'D'E'F'$
$-0000-\ (0,1,32,33)$	X	X	$B'C'D'E'$
$100-01\ (33,37)$		X	$AB'C'E'F$

Table 7.15: Quine-McCluskey Ex 2: 2nd Iteration

For the next simplification, row $-0000-$ is selected since that would also cover the minterms that are marked for all remaining rows. Thus, the expression $B'C'D'E'$ will become part of the final equation.

When the analysis is completed, the original equation (7.4), which contained 14 minterms, is simplified into Equation 7.5, which contains only five terms.

$$ABD'E'F' + A'B'D'E' + AB'C'DF + A'B'CD + B'C'D'E' = Y \tag{7.5}$$

7.1.4 *Summary*

While the Quine–McCluskey method is useful for large Boolean expressions containing multiple inputs, it is also tedious and prone to error when done by hand. Also, there are some Boolean expressions (called "Cyclic" and "Semi-Cyclic" Primes) that do not reduce using this method. Finally, both Karnaugh maps and Quine-McCluskey methods become very complex when more than one output is required of a circuit. Fortunately, many automated tools are available to simplify Boolean expressions using advanced mathematical techniques.

7.1.5 *Practice Problems*

The following problems are presented as practice for using the Quine-McClusky method to simplify a Boolean expression. Note: designers can select different Prime Implicants so the simplified expression could vary from what is presented below.

1	Expression	$\int(A,B,C,D) = \sum(0,1,2,5,6,7,9,10,11,14)$
	Simplified	$A'B'C' + A'BD + AB'D + CD'$
2	Exression	$\int(A,B,C,D) = \sum(0,1,2,3,6,7,8,9,14,15)$
	Simplified	$A'C + BC + B'C'$
3	Exression	$\int(A,B,C,D) = \sum(1,5,7,8,9,10,11,13,15)$
	Simplified	$C'D + AB' + BD$
3	Exression	$\int(A,B,C,D,E) = \sum(0,4,8,9,10,11,12,13,14,15,16,20,24,28)$
	Simplified	$A'B + D'E'$

Table 7.16: Quine-McCluskey Practice Problems

7.2 AUTOMATED TOOLS

7.2.1 *Introduction*

There are numerous automated tools available to aid in simplifying complex Boolean equations. Many of the tools are quite expensive and intended for professionals working full time in large companies; but others are inexpensive, or even free of charge, and are more than adequate for student use. One free tool, a Java application named KARMA, can be downloaded from http://goo.gl/8Lmx5v and installed on a local computer. It helps to simplify complex Boolean expressions using both Karnaugh Maps and Quine-McCluskey methods. KARMA is a good program with numerous benefits but is not further covered here.

One online site of interest is http://www.32x8.com/. This site permits visitors to set up a truth table with two to eight variables and then will create a simplified Boolean circuit for that truth table.

Part II

PRACTICE

Once the foundations of digital logic are mastered it is time to consider creating circuits that accomplish some practical task. This part of the book begins with the simplest of combinational logic circuits and progresses to sequential logic circuits and, finally, to complex circuits that combine both combinational and sequential elements.

8

COMBINATIONAL ARITHMETIC CIRCUITS

8.1 INTRODUCTION

Electronic circuits that do not require any memory devices (like flip-flops or registers) use what is called "Combinational Logic." These systems can be quite complex, but all outputs are determined solely by the actions of a series of logic gates on a given set of inputs. Another characteristic of combinational circuits is the lack of feedback. An output state is determined almost instantly from the inputs with no feedback loops. Combinational circuits can be reduced to a Boolean Algebra expression, though one that may be quite complex. Combinational circuits include many useful applications, including adders, subtractors, multipliers, dividers, encoders, decoders, display drivers, and keyboard encoders.

This chapter develops combinational logic circuits.[1]

8.2 ADDERS AND SUBTRACTORS

8.2.1 *Introduction*

In the Binary Mathematics chapter, the concept of adding two binary numbers was developed and the process of adding binary numbers can be easily implemented in hardware. If two one-bit numbers are added, they will produce a sum and an optional carry-out bit and the circuit that performs this is called a "half adder." If two one-bit numbers along with a carry-in bit from another stage are added they will produce a sum and an optional carry-out, and the circuit that performs this is function called a "full adder."

Subtractors are similar to adders but they must be able to signal a "borrow" bit rather than send a carry-out bit. Like adders, subtractors are developed from half-subtractor circuits. This unit develops both adders and subtractors.

8.2.2 *Half Adder*

Following is the truth table for a half adder.

---

[1] All circuits found in this chapter are available as *Logisim-evolution* files in the course materials.

Inputs		Output	
A	B	Sum	COut
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 8.1: Truth Table for Half-Adder

The *Sum* column in this truth table is the same pattern as an XOR) so the easiest way to create a half adder is to use an XOR gate. However, if both input bits are high a half adder will also generate a carry out (*COut*) bit, so the half adder circuit should be designed to provide that carry out. The circuit in Figure 8.1 meets those requirements. In this circuit, *A* and *B* are connected to XOR gate U1 and that is connected to output *Sum*. *A* and *B* are also connected to AND gate U2 and that is connected to carry-out bit *COut*.

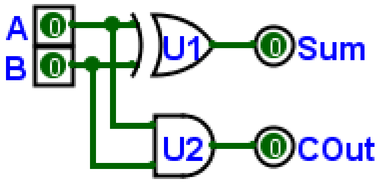


Figure 8.1: Half-Adder

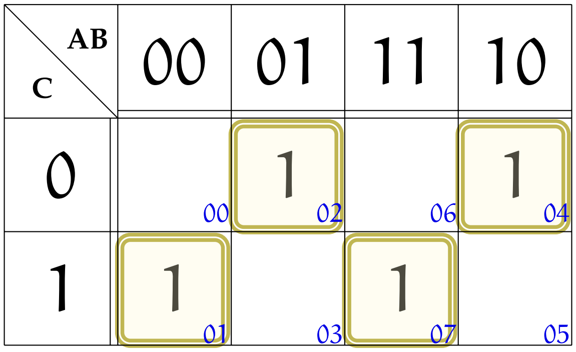
8.2.3 *Full Adder*

A full adder sums two one-bit numbers along with a carry-in bit and produces a sum with a carry-out bit. Truth table 8.2 defines a full adder.

Inputs			Output	
A	B	CIn	Sum	COut
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

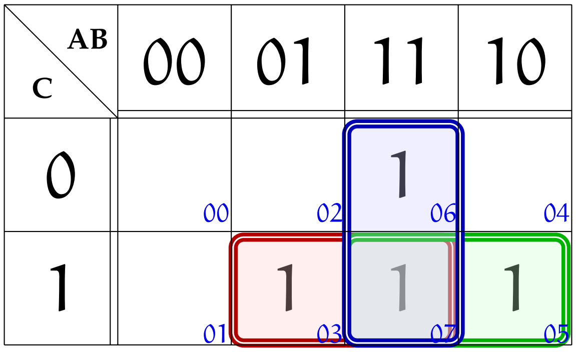
Table 8.2: Truth Table for Full Adder

Following are Karnaugh Maps for both outputs.



AB C	00	01	11	10
0	00	1 02	06	1 04
1	1 01	03	1 07	05

Figure 8.2: K-Map For The SUM Output



AB C	00	01	11	10
0	00	02	1 06	04
1	01	1 03	1 07	1 05

Figure 8.3: K-Map For The COut Output

Karnaugh map 8.2 is a Reed-Muller pattern that is typical of an XOR gate. Karnaugh map 8.3 can be reduced to three Boolean expressions. The full adder circuit is, therefore, defined by the following Boolean equations.

$$\begin{aligned} A \oplus B \oplus CIn &= Sum \\ (A * B) + (A * CIn) + (B * CIn) &= COut \end{aligned} \tag{8.1}$$

Figure 8.4 is a full adder. In essence, this circuit combines two half-adders such that U1 and U2 are one half-adder that sums *A* and *B* while U3 and U4 are the other half-adder that sums the output of the first half-adder and *CIn*.

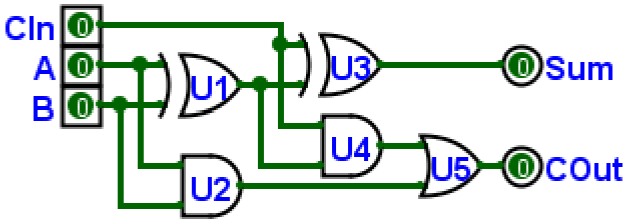


Figure 8.4: Full Adder

8.2.4 *Cascading Adders*

The full adder developed above will only add two one-bit numbers along with an optional carry-in bit; however, those adders can be cascaded such that an adder of any bit width can be easily created. Figure 8.5 shows a four-bit adder created by cascading four one-bit adders.

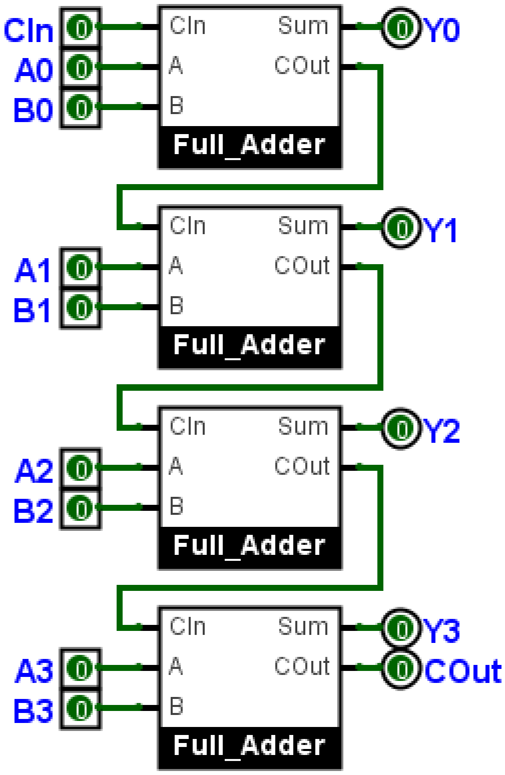


Figure 8.5: 4-Bit Adder

This circuit would add two four-bit inputs, *A* and *B*. Stage zero, at the top of the stack, adds bit zero from both inputs and then outputs bit zero of the sum, *Y0*, along with a carry-out bit. The carry-out bit from stage zero is wired directly into the stage one's carry-in port. That adder then adds bit one from both inputs along with the carry-in bit to create bit one of the sum, *Y1*, along with a carry-out bit. This process continues until all for bits have been added. In the end, outputs *Y0* - *Y3* are combined to create a four-bit sum. If there is a carry-out bit from the last stage it could be used as the carry-in bit for another device or could be used to signal an overflow error.

8.2.5 *Half Subtractor*

To understand a binary subtraction circuit, it is helpful begin with subtraction in a base-10 system.

$$\begin{array}{r} 83 \\ -\underline{65} \\ 18 \end{array}$$

Since 5 cannot be subtracted from 3 (the least significant digits), 10 must be borrowed from 8. This is simple elementary-school arithmetic but the principle is important for base-2 subtraction. There are only four possible one-bit subtraction problems.

$$\begin{array}{rrrr} 0 & 1 & 1 & 10 \\ -\underline{0} & -\underline{0} & -\underline{1} & -\underline{1} \\ 0 & 1 & 0 & 1 \end{array}$$

The first three examples above need no explanation, but the fourth only makes sense when it is understood that it is impossible to subtract 1 from 0 so 10_2 was borrowed from the next most significant bit position. The problems above were used to generate the following half-subtractor truth table.

Inputs		Outputs	
A	B	Diff	BOut
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Table 8.3: Truth Table for Half-Subtractor

Diff is the difference of *A* minus *B*. *BOut* ("Borrow Out") is a signal that a borrow is necessary from the next most significant bit position when *B* is greater than *A*. The following Boolean equations define the calculations needed for a half-subtractor.

$$\begin{aligned} A \oplus B &= \mathrm{Diff} \\ A' * B &= \mathrm{BOut} \end{aligned} \tag{8.2}$$

The pattern for *Diff* is the same as an XOR gate so using an XOR gate is the easiest way to generate the difference. *BOut* is only high when *A* is low and *B* is high so a simple AND gate with one inverted input can be used to generate *BOut*. The circuit in figure 8.6 realizes a half-subtractor.

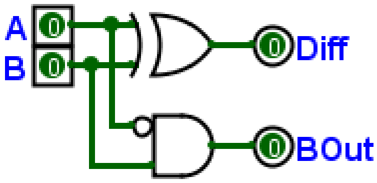


Figure 8.6: Half-Subtractor

8.2.6 *Full Subtractor*

A full subtractor produces a difference and borrow-out signal, just like a half-subtractor, but also includes a borrow-in signal so they can be cascaded to create a subtractor of any desired bit width.

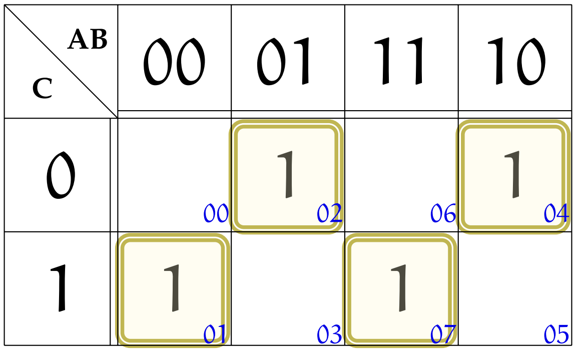
Truth table 8.4 is for a full subtractor.

Inputs			Output	
A	B	BIn	Diff	BOut
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Table 8.4: Truth Table for Subtractor

Diff is the difference of *A* minus *B* minus *BIn*. *BOut* ("Borrow Out") is a signal that a borrow is necessary from the next most significant bit position when *A* is less than *B* plus *BIn*.

Following are Karnaugh Maps for both outputs.



C \ AB	00	01	11	10
0	00	1 02	06	1 04
1	1 01	03	1 07	05

Figure 8.7: K-Map For The Difference Output

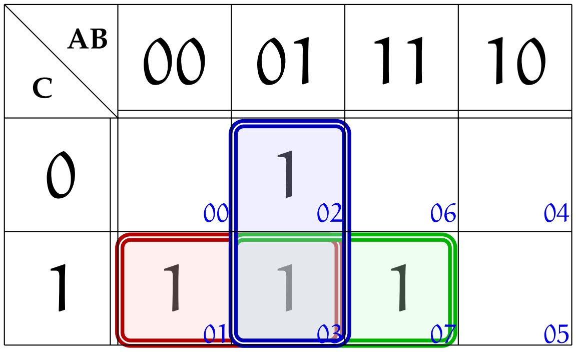


Figure 8.8: K-Map For The BOut Output

Karnaugh map 8.7 is a Reed-Muller pattern that is typical of an XOR gate. Karnaugh map 8.8 can be reduced to three Boolean expressions. The full subtactor circuit is, therefore, defined by the following Boolean equations.

$$\begin{aligned} A \oplus B \oplus BIn &= Diff \qquad (8.3)\\ A'B + (A' * BIn) + (B * BIn) &= BOut \end{aligned}$$

The circuit in figure 8.9 realizes a subtractor.

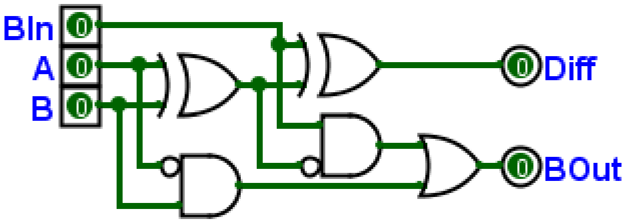


Figure 8.9: Subtractor

8.2.7 *Cascading Subtractors*

The full subtractor developed above will only subtract two one-bit numbers along with an optional borrow bit; however, those subtractors can be cascaded such that a subtractor of any bit width can be easily created. Figure 8.10 shows a four-bit subtractor created by cascading four one-bit subtractors.

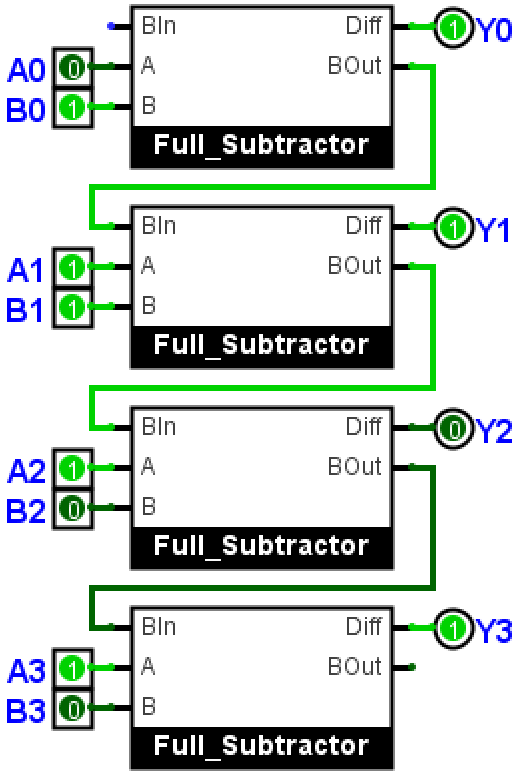


Figure 8.10: 4-Bit Subtractor

This circuit would subtract a four bit number B from A. The subtractor is set up to solve $1110_2 - 0011_2 = 1011_2$. Stage zero, at the top of the stack, subtracts bit zero of input B from bit zero of input A and then outputs bit zero of the difference, $Y0$, along with a borrow-out bit. The borrow-out bit from stage zero is wired directly into the stage one's borrow-in port. That stage then subtracts bit one of input B from bit one of input A along with the borrow-in bit to create bit one of the sum, $Y1$, along with a borrow-out bit. This process continues until all for bits have been subtracted. In the end, outputs $Y0$ - $Y3$ are combined to create a four-bit difference. The borrow-out bit of the last stage is not connected to anything but it could be used as the borrow-in bit for another device.

8.2.8 *Adder-Subtractor Circuit*

It is remarkably easy to create a device that both adds and subtracts based on a single-bit control signal. Figure 8.11 is a 4-bit adder that was modified to become both an adder and subtractor. The circuit has been set up with this problem: $0101_2 - 0011_2 = 0010_2$.

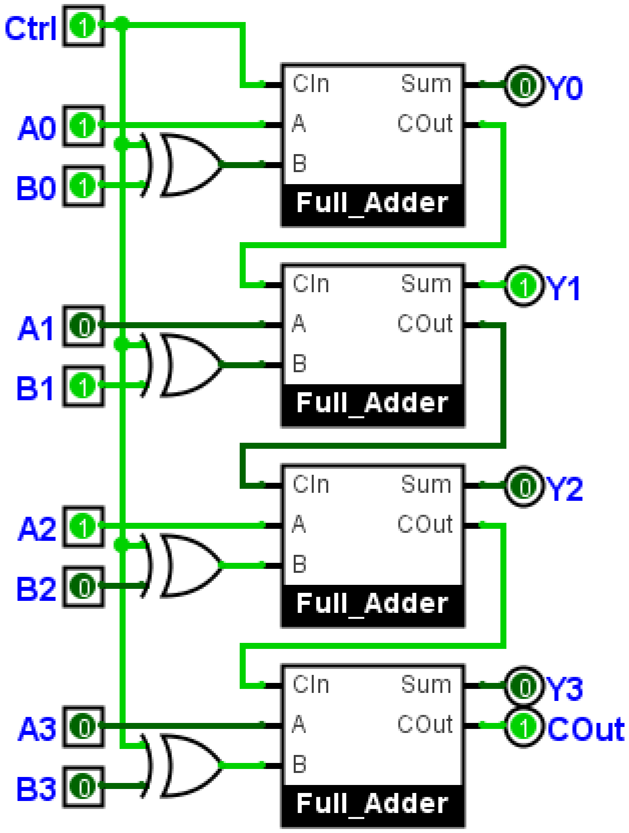


Figure 8.11: 4-Bit Adder-Subtractor

To change an adder to an adder-subtractor makes use of the binary mathematics concept of subtracting by adding the twos complement (see Section 3.2.5.2 on page 44). The "trick" is to use the XOR gates on input *B* to convert that input to its complement then the adder will subtract *B* from *A* instead of add.

To create the twos complement of a binary number each of the bits are complemented and then one is added to the result (again, this process is described in Section 3.2.5.2). Each of the *B* input bits are wired through one input of an XOR gate. The other input of that gate is a *Ctrl* ("Control") bit. When *Ctrl* is low then each of the *B* inputs are transmitted through an XOR gate without change and the adder works as an adder. When *Ctrl* is high then each of the *B* inputs are complemented by an XOR gate such that the ones complement is created. However, *Ctrl* is also wired to the *CIn* input of the first stage which has the effect of adding one to the result and turn input *B* into a twos complement number. Now the adder will subtract input *B* from input *A*.

In the end, the designer only needs to set *Ctrl* to zero to make the circuit add or one to make the circuit subtract.

8.2.9 *Integrated Circuits*

In practice, circuit designers rarely build adders or subtractors. There are many different types of manufactured low-cost adders, subtractors, and adder/subtractor combinations available and designers usually find it easiest to use one of those circuits rather than re-invent the proverbial wheel. A quick look at Wikipedia[2] found this list of adders:

- 7480, gated full adder
- 7482, two-bit binary full adder
- 7483, four-bit binary full adder
- 74183, dual carry-save full adder
- 74283, four-bit binary full adder
- 74385, quad four-bit adder/subtractor
- 74456, BCD adder

In addition to adder circuits, designers can also opt to use an ALU IC.

8.3 ARITHMETIC LOGIC UNITS

An ALU is a specialized IC that performs all arithmetic and logic functions needed in a device. Most ALU will carry out dozens of different functions like the following few examples from a 74181 ALU (assume that the ALU has two inputs, A and B, and one output, F):

- F = NOT A
- F = A NAND B
- F = (NOT A) OR B
- F = B
- F = (NOT A) AND B
- F = A − 1
- F = A − B
- F = AB − 1
- F = −1

---

[2] https://www.wikipedia.com/en/List_of_7400_series_integrated_circuits

ALU are very important in many devices, in fact, they are at the core of a CPU. Because they are readily available at low cost, most designers will use a commercially-produced ALU in a project rather than try to create their own.

A quick look at Wikipedia[3] found this list of ALU:

- 74181, four-bit arithmetic logic unit and function generator
- 74381, four-bit arithmetic logic unit/function generator with generate and propagate outputs
- 74382, four-bit arithmetic logic unit/function generator with ripple carry and overflow outputs
- 74881, Arithmetic logic unit

8.4 COMPARATORS

A comparator compares two binary numbers, *A* and *B*. One of three outputs is generated by the comparison: $A = B$, $A > B$, $A < B$. A one-bit comparator uses a combination of AND gates, NOT gates, and an XNOR gate to generate a *True* output for each of the three comparisons:

$A = B$	$(A \odot B)'$
$A > B$	AB'
$A < B$	$A'B$

Table 8.5: One-Bit Comparator Functions

Figure 8.12 is the logic diagram for a one-bit comparator.

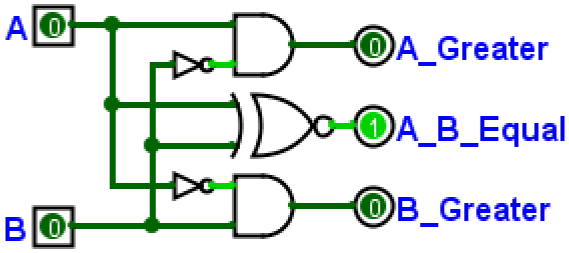


Figure 8.12: One-Bit Comparator

To compare numbers larger than one bit requires a more involved analysis of the problem. First, a truth table is developed for every possible combination of two 2-bit numbers, *A* and *B*.

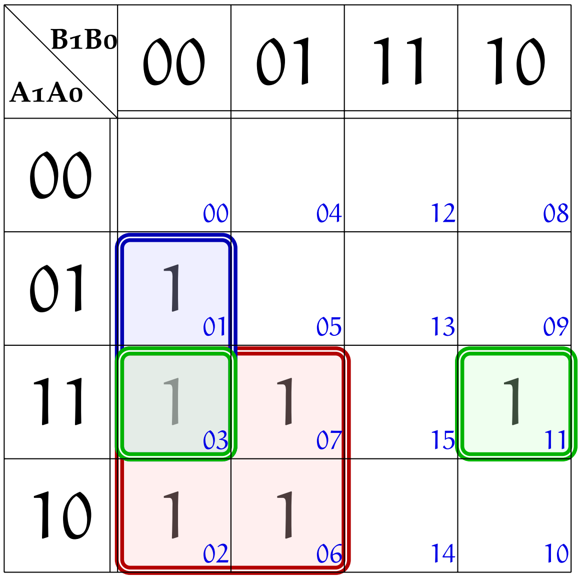
---

[3] https://www.wikipedia.com/en/List_of_7400_series_integrated_circuits

Inputs				Outputs		
A1	A0	B1	B0	$A < B$	$A = B$	$A > B$
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

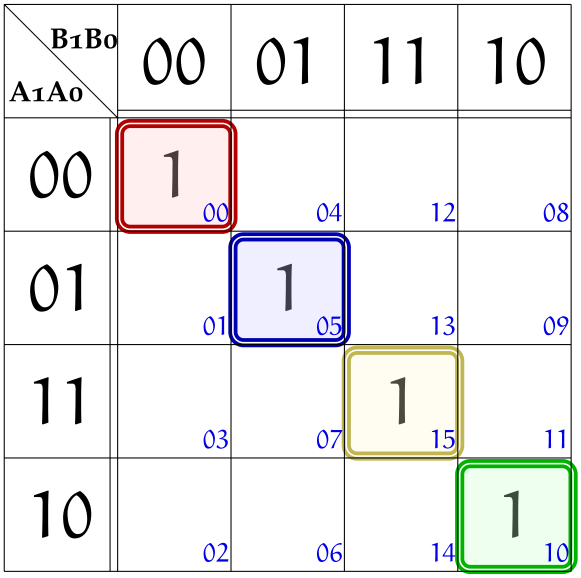
Table 8.6: Truth Table for Two-Bit Comparator

Next, Karnaugh Maps are developed for each of the three outputs.



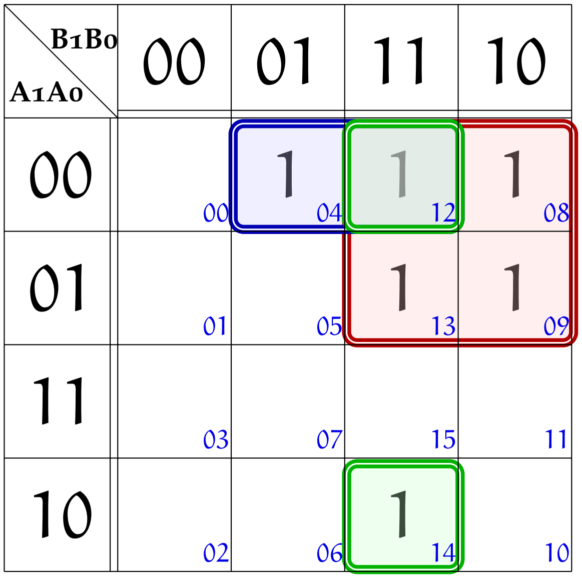
B1B0 A1A0	00	01	11	10
00	00	04	12	08
01	1 01	05	13	09
11	1 03	1 07	15	1 11
10	1 02	1 06	14	10

Figure 8.13: K-Map For $A > B$



B1B0 A1A0	00	01	11	10
00	1 00	04	12	08
01	01	1 05	13	09
11	03	07	1 15	11
10	02	06	14	1 10

Figure 8.14: K-Map For $A = B$



B1B0 A1A0	00	01	11	10
00	00	1 04	1 12	1 08
01	01	05	1 13	1 09
11	03	07	15	11
10	02	06	1 14	10

Figure 8.15: K-Map For $A = B$

Given the above K-Maps, the following Boolean Equations can be derived.

$$\begin{aligned} A < B &: A1'B1 + A0'B1B0 + A1'A0'B0 \\ A = B &: (A0 \odot B0)(A1 \odot B1) \\ A > B &: A1B1' + A0B1'B0' + A1A0B0' \end{aligned} \tag{8.4}$$

The above Boolean expressions can be used to create the circuit in Figure 8.16.

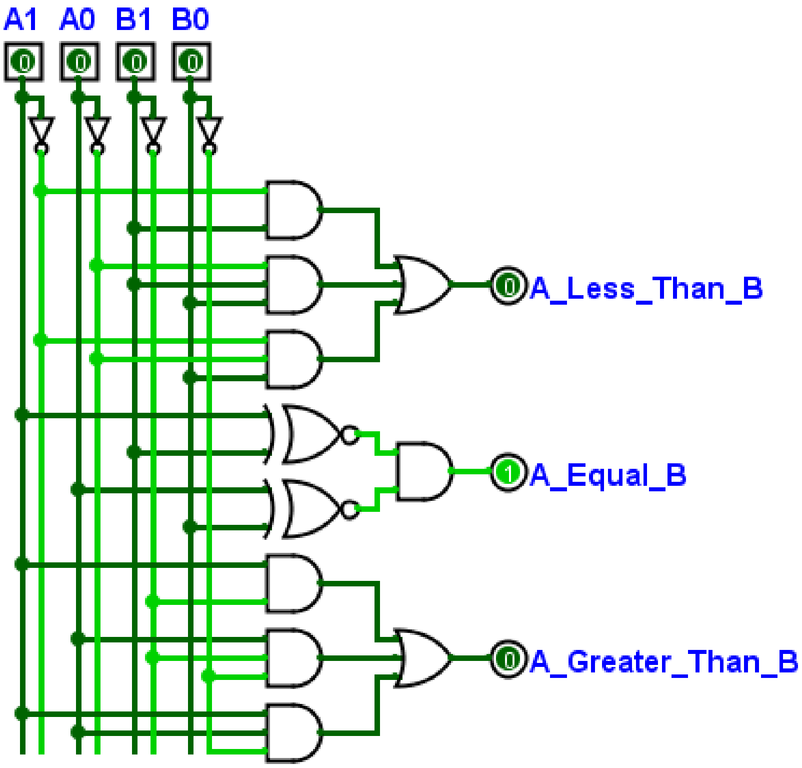

A1
A0
B1
B0
0
0
0
0
0
A_Less_Than_B
1
A_Equal_B
0
A_Greater_Than_B

Figure 8.16: Two-Bit Comparator

9

COMBINATIONAL ENCODER CIRCUITS

9.1 INTRODUCTION

Electronic circuits that do not require any memory devices (like flip-flops or registers) use what is called "Combinational Logic." These systems can be quite complex, but all outputs are determined solely by the actions of a series of logic gates on a given set of inputs. Another characteristic of combinational circuits is the lack of feedback. An output state is determined almost instantly from the inputs with no feedback loops. Combinational circuits can be reduced to a Boolean Algebra expression, though one that may be quite complex. Combinational circuits include many useful applications, including adders, subtractors, multipliers, dividers, encoders, decoders, display drivers, and keyboard encoders.

This chapter develops combinational logic circuits.[1]

9.2 MULTIPLEXERS/DEMULTIPLEXERS

9.2.1 *Multiplexer*

A multiplexer is usually called a "mux" and a demultiplexer is called a "dmux."

A multiplexer is used to connect one of several input lines to a single output line. Thus, it selects which input to pass to the output. This function is similar to a rotary switch where several potential inputs to the switch can be sent to a single output. A demultiplexer is a multiplexer in reverse, so a single input can be routed to any of several outputs. While mux/dmux circuits were originally built for transmission systems (like using a single copper wire to carry several different telephone calls simultaneously), today they are used as "decision-makers" in virtually every digital logic system and are, therefore, one of the most important devices for circuit designers.

To help clarify this concept, Figure 9.1 is a simple schematic diagram that shows two rotary switches set up as a mux/dmux pair. As the switches are set in the diagram, a signal would travel from `INPUT B` to `OUTPUT 2` through a connecting wire.

---

[1] All circuits found in this chapter are available as *Logisim-evolution* files in the course materials.

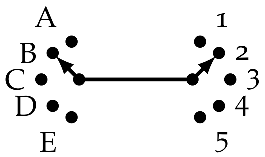


Figure 9.1: Multiplexer Using Rotary Switches

Imagine that the switches could somehow be synchronized so they rotated among the setting together; that is, `INPUT A` would always connect to `OUTPUT 1` and so forth. That would mean a single wire could carry five different signals. For example, imagine that the inputs were connected to five different intrusion sensors in a building and the five outputs were connected to lamps on a guard's console in a remote building. If something triggered sensor A then as soon as the mux/dmux pair rotated to that position it would light lamp one on the console. Carrying all of these signals on a single wire saves a lot of expense. Of course, a true alarm system would be more complex than this, but this example is only designed to illustrate how a mux/dmux pair works in a transmission system.

[2]Figure 9.2 is the logic diagram for a simple one-bit two-to-one multiplexer. In this circuit, an input is applied to input ports A and B. Port *Sel* is the selector and if that signal is zero then Port A will be routed to output Y; if, though, *Sel* is one then Port B will be routed to Y.

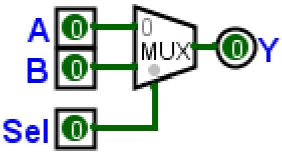


Figure 9.2: Simple Mux

Truth Table 9.1, below, is for a multiplexer:

---

[2] In this book, input data ports are labeled with a single letter from the beginning of the alphabet, output data ports are labeled with a single letter from the end of the alphabet, and other signals use descriptive labels.

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

Table 9.1: Truth Table for a Multiplexer

In this multiplexer, the data input ports are only a single bit wide; however, in a normal circuit those ports could be a full 32-bit or 64-bit word and the selected word would be passed to the output port. Moreover, a multiplexer can have more than two input ports so a very versatile switch can be built to handle switching full words from one of eight or even sixteen different inputs. Because of its ability to channel a selected data stream to a single bus line from many different sub-circuits, the multiplexer is one of the workhorses for digital logic circuits and is frequently found in complex devices like CPU.

9.2.2 *Demultiplexer*

A demultiplexer is the functional opposite of a multiplexer: a single input is routed to one of several potential outputs. Figure 9.3 is the logic diagram for a one-bit one-to-two demultiplexer. In this circuit, an input is applied to input port A. Port *Sel* is a control signal and if that signal is zero then input A will be routed to output Y, but if the control signal is one then input A will be routed to output Z.

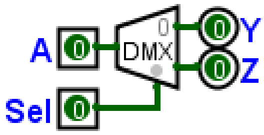


Figure 9.3: Simple Dmux

Truth Table 9.2, below, is for a demultiplexer.

Inputs		Outputs	
A	Sel	Y	Z
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

Table 9.2: Truth Table for a Demultiplexer

In this demultiplexer the data input port is only a single bit wide; however, in a normal circuit that port could be a full 32-bit or 64-bit word and that entire word would be passed to the selected output port. Moreover, a demultiplexer can have more than two outputs so a very versatile switch can be built to handle switching full words to one of eight or even sixteen different outputs. Because of its ability to switch a data stream to different sub-circuits, the demultiplexer is one of the workhorses for digital logic circuits and is frequently found in complex devices like CPU.

9.2.3 *Minterm Generators*

Demultiplexers can be combined with an OR gate and be used as a minterm generator. Consider the circuit for this two-variable equation.

$$\int(A,B) = \sum(1,2) \tag{9.1}$$

Since there are two input variables, *A* and *B*, the dmux needs to have two select bits, one for each variable, and that would generate four potential dmux outputs *W*, *X*, *Y*, and *Z*. This circuit could be constructed using a four-output dmux with a two-bit control signal.

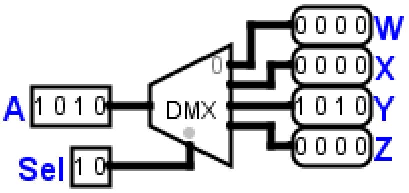


Figure 9.4: 1-to-4 Dmux

In Figure 9.4, a four-bit input (*A*) is routed to one of four output ports: *W*, *X*, *Y*, or *Z*, depending on the setting of the select, *Sel*. Figure 9.4 shows the data input of 1010 being routed to *Y* by a select of 10.

However, the equation specifies that the only outputs that would be used are when *Sel* is 01 or 10. Thus, output ports *X* and *Y* must be sent to an OR gate and the other two outputs ignored. The output of the OR gate would only activate when *Sel* is set to 01 or 10, as shown in Figure 9.5.

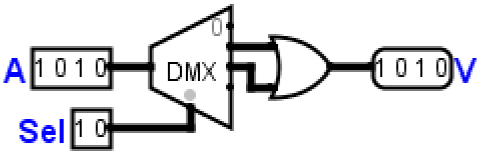


Figure 9.5: 1-to-4 Dmux As Minterm Generator

9.3 ENCODERS/DECODERS

9.3.1 *Introduction*

Encoders and decoders are used to convert some sort of coded data into a different code. For example, it may be necessary to convert the code created by a keyboard into ASCII for use in a word processor. By definition, the difference between an encoder and a decoder is the number of inputs and outputs: Encoders have more inputs than outputs, while decoders have more outputs than inputs.

As an introduction to encoders, consider Figure 9.6, which is designed to encode three single line inputs (maybe three different push buttons on a control panel) into a binary number for further processing by the computer. In this circuit, the junction between the two OR gates and the output (Y) is a *joiner* that combines two bit streams into a single bus. Physically, two wires (one from *U1* and one from *U2*) would be spliced together into a single cable (a *bus*) that contains two strands. The figure shows that when input *C* is active the output of the encoder is 11.

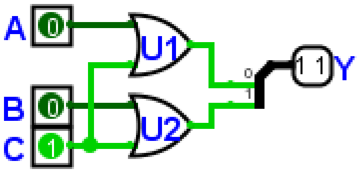


Figure 9.6: Three-line to 2-Bit Encoder

As an introduction to decoders, consider Figure 9.7, which is designed to decode a two-bit binary input and drive a single output line high. A circuit like this may be used to light an LED used as a warning on a console if a particular binary code is generated elsewhere in a circuit. In this circuit, the input is a two-bit number (10 in the illustration), but those two bits are separated through a splitter and each is applied to one of the inputs of a series of four AND gates. Imagine that the MSB was placed on the wire on the left of the grid and wired to the bottom input of each of the AND gates. If A = 10, then AND gate three would activate and output *X* would go high.

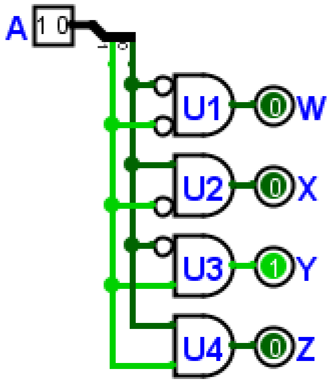


Figure 9.7: Four-Bit to 4-Line Decoder

Both encoders and decoders are quite common and are used in many electronic devices. However, it is not very common to build these circuits out of discrete components (like in the circuits above). Rather, inexpensive integrated circuits are available for most encoder/decoder operations and these are much easier, and more reliable, to use.

9.3.2 *Ten-Line Priority*

This encoder is sometimes called Ten-Line to Four-Line.

Consider a ten-key keypad containing the numbers zero through nine, like a keypad that could be used for numeric input from some hand-held device. In order to be useful, a key press would need to be encoded to a binary number for further processing by a logic circuit.

The keypad outputs a nine-bit number such that a single bit goes high to indicate which key was. For example, when key number two is pressed, 0_0000_0010 is output from the device. A priority encoder would accept that nine-bit number and output a binary number that could be used in computer circuits. Truth Table 9.3 is for the Priority Encoder that meets the specification. The device is called a "priority" encoder since it will respond to only the highest value key press. For example, if someone pressed the three and five keys simultaneously the encoder would ignore the three key press and transmit 0101, or binary five.

Dashes are normally used to indicate "don't care" on a Truth Table.

Inputs									Outputs			
9	8	7	6	5	4	3	2	1	Y1	Y2	Y3	Y4
0	0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	1	-	0	0	1	0
0	0	0	0	0	0	1	-	-	0	0	1	1
0	0	0	0	0	1	-	-	-	0	1	0	0
0	0	0	0	1	-	-	-	-	0	1	0	1
0	0	0	1	-	-	-	-	-	0	1	1	0
0	0	1	-	-	-	-	-	-	0	1	1	1
0	1	-	-	-	-	-	-	-	1	0	0	0
1	-	-	-	-	-	-	-	-	1	0	0	1

Table 9.3: Truth Table for Priority Encoder

This circuit can be realized by using a grid input and routing the various lines to an appropriate *AND* gate. This is one of the circuits built in the lab manual that accompanies this text.

9.3.3 *Seven-Segment Display*

A seven-segment display is commonly used in calculators and other devices to show hexadecimal numbers. To create the numeric shapes, various segments are activated while others remain off, so binary numbers must be decoded to turn on the various segments for any given combination of inputs. A seven-segment display has eight input ports and, when high, each of those ports will activate one segment of the display.

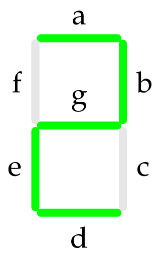


Figure 9.8: Seven-Segment Display

Usually an eighth "segment" is available —a decimal point in the lower right corner.

In Figure 9.8 the seven segments are labeled and it shows that the number "2" for example, is made by activating segments a, b, g, e, and d. Table 9.4 shows the various segments that must be activated for each of the 16 possible input values.

Hex	—	Binary				—	Display						
	—	**3**	**2**	**1**	**0**	—	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	—	0	0	0	0	—	1	1	1	1	1	1	0
1	—	0	0	0	1	—	0	1	1	0	0	0	0
2	—	**0**	**0**	**1**	**0**	—	**1**	**1**	**0**	**1**	**1**	**0**	**1**
3	—	0	0	1	1	—	1	1	1	1	0	0	1
4	—	0	1	0	0	—	0	1	1	0	0	1	1
5	—	0	1	0	1	—	1	0	1	1	0	1	1
6	—	0	1	1	0	—	1	0	1	1	1	1	1
7	—	0	1	1	1	—	1	1	1	0	0	0	0
8	—	1	0	0	0	—	1	1	1	1	1	1	1
9	—	1	0	0	1	—	1	1	1	1	0	1	1
A	—	1	0	1	0	—	1	1	1	0	1	1	1
B	—	1	0	1	1	—	0	0	1	1	1	1	1
C	—	1	1	0	0	—	1	0	0	1	1	1	0
D	—	1	1	0	1	—	0	1	1	1	1	0	1
E	—	1	1	1	0	—	1	0	0	1	1	1	1
F	—	1	1	1	1	—	1	0	0	0	1	1	1

Table 9.4: Truth Table for Seven-Segment Display

Notice that to display the number "2" (in bold font), segments a, b, d, e, and g must be activated.

A decoder circuit, as in Figure 9.9, uses a demultiplexer to activate a one-bit line based on the value of the binary input. Note: to save space, two parallel decoders are used in this circuit.

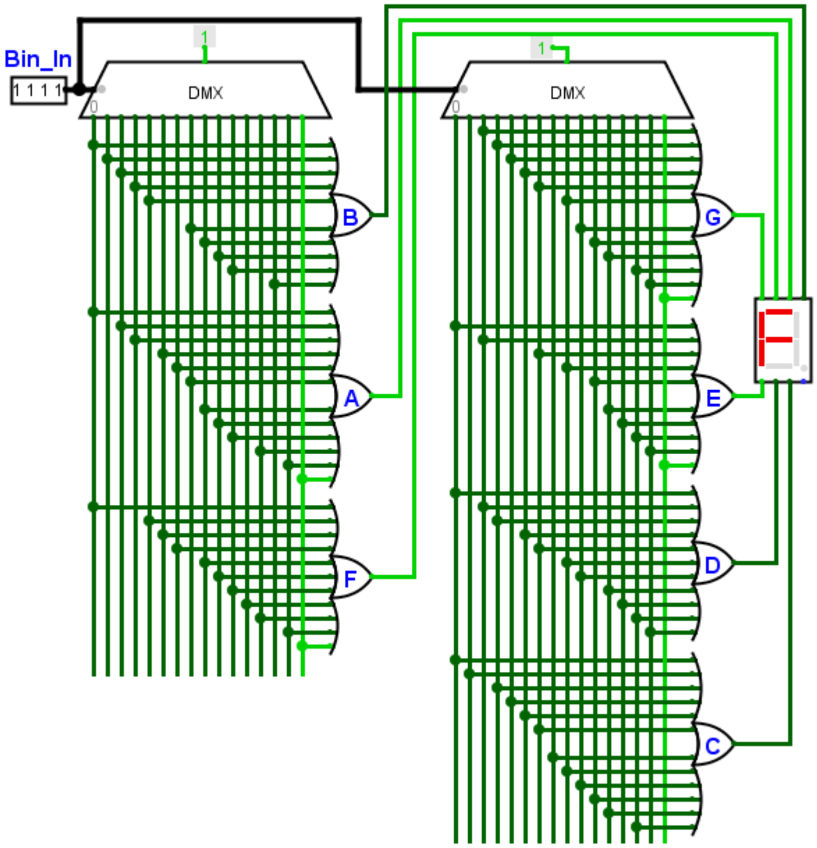


Figure 9.9: 7-Segment Decoder

Figure 9.9 shows an input of 1111_2 so the last line on each demultiplexer is activated. Those lines are used to activate the necessary inputs on the seven-segment display to create an output of “F”.

A Hex Digit Display has a single port that accepts a four-bit binary number and that number is decoded into a digital display. Figure 9.10 shows a hex digit display.

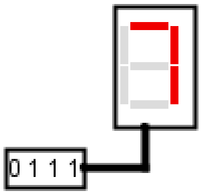


Figure 9.10: Hex Decoder

The *Logisim-evolution* simulator used in this class includes both a *7-Segment Display* and a *Hex Digit Display*. One of the strengths of using a seven-segment display rather than a hex digit display is that the circuit designer has total control over which segments are displayed. It is common, for example, to activate each of the outer segments in a rapid sequence to give the illusion of a rotating circle. As opposed to a seven-segment display, a hex digit display is very simple to wire and use, as Figures 9.9 and 9.10 make clear. Both of these two types of displays are available on the market and a designer would chose whichever type meets the needs.

9.3.4 *Function Generators*

Decoders provide an easy way to create a circuit when given a minterm function. Imagine that a circuit is needed for the function defined in Equation 9.2.

$$\int(A, B, C) = \sum(0, 2, 7) \tag{9.2}$$

Whatever this circuit is designed to do, it should activate an output only when the input is zero, two, or seven. The circuit in Figure 9.11 illustrates a simple minterm generator using a demultiplexer and an OR gate. When input A is zero, two, or seven then output Y will go high, otherwise it is low.

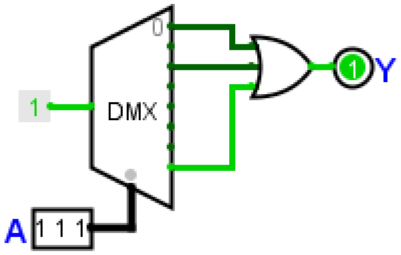


Figure 9.11: Minterm Generator

9.4 ERROR DETECTION

9.4.1 *Introduction*

Whenever a byte (or any other group of bits) is transmitted or stored, there is always the possibility that one or more bits will be accidentally complemented. Consider these two binary numbers:

0110 1010 0011 1010
0110 1010 0111 1010

They differ by only one bit (notice Group Three). If the top number is what is supposed to be in a memory location but the bottom number is what is actually there then this would, obviously, create a problem. There could be any number of reasons why a bit would be wrong, but the most common is some sort of error that creeps in while the byte is being transmitted between two stores, like between a USB drive and memory or between two computers sharing a network. It is desirable to be able to detect that a byte contains a bad bit and, ideally, even know which bit is wrong so it can be corrected.

Parity is a common method used to check data for errors and it can be used to check data that has been transmitted, held in memory, or stored on a hard drive. The concept of parity is fairly simple: A bit (called the *parity bit*) is added to each data byte and that extra bit is either set to zero or one in order to make the bit-count of that byte contain an even or odd number of ones. For example, consider this binary number:

1101

There are three ones in this number, which is an odd number. If odd parity is being used in this circuit, then the parity bit would be zero so there would be an odd number of ones in the number. However, if the circuit is using even parity, then the parity bit would be set to one in order to have four ones in the number, which is an even number. Following is the above number with both even and odd parity bits (those parity bits are in the least significant position and are separated from the original number by a space for clarity):

1101 0 (Odd Parity)
1101 1 (Even Parity)

Table 9.5 shows several examples that may help to clarify this concept. In each case, a parity bit is used to make the data byte even parity (spaces were left in the data byte for clarity).

Data Byte	Parity Bit
0000 0000	0
0000 0001	1
0000 0011	0
0000 0100	1
1111 1110	1
1111 1111	0

Table 9.5: Even Parity Examples

Generating a parity bit can be done with a series of cascading *XOR* gates but *Logisim-evolution* had two parity gates, one that outputs high when the inputs have an odd number of ones and the other when there are an even number of ones. Figure 9.12 illustrates using an odd parity gate (labeled "2K+1"). In this circuit, if input *A* has an odd number of ones, as illustrated, then the parity generator will output a one to indicate input A has an odd number of ones. That parity bit is added as the most significant bit to output Y. Since output Y will always have an even number of bits this is an even parity circuit.

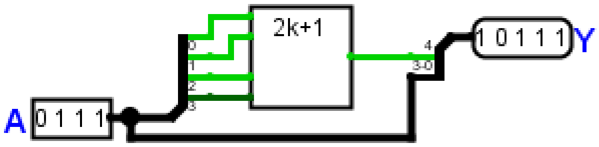


Figure 9.12: Parity Generator

Parity is a simple concept and is the foundation for one of the most basic methods of error checking. As an example, if some byte is transmitted using even parity but the data arrives with an odd number of ones then one of the bits was changed during transmission.

9.4.2 *Iterative Parity Checking*

One of the problems with using parity for error detection is that while it may be known that *something* is wrong, there is no way to know which of the bits is wrong. For example, imagine an eight-bit system is using even parity and receives this data and parity bit:

1001 1110 PARITY: 0

There is something wrong with the byte. It is indicating even parity but has an odd number of ones in the byte. It is impossible to know which bit changed during transmission. In fact, it may be that the byte is correct but the parity bit itself changed (a *false error*). It would be nice if the parity error detector would not only indicate that there was an error, but could also determine which bit changed so it could be corrected.

One method of error correction is what is known as *Iterative Parity Checking*. Imagine that a series of eight-bit bytes were being transmitted. Each byte would have a parity bit attached; however, there would also be a parity byte that contains a parity bit for each bit in the preceding five bytes. It is easiest to understand this by using a table (even parity is being used):

Byte	Data								Parity
1	0	0	0	0	0	0	0	0	0
2	1	0	1	1	0	0	0	0	1
3	1	0	1	1	0	0	1	1	1
4	1	1	1	0	1	0	1	0	1
5	0	1	0	0	0	0	0	0	1
P	1	0	1	0	1	0	0	1	0

Table 9.6: Iterative Parity

In Table 9.6, Byte one is 0000 0000. Since the system is set for even parity, and it is assumed that a byte with all zeros is even, then the parity bit is zero. Each of the five bytes has a parity bit that is properly set such that each byte (with the parity bit) includes an even number of bits. Then, after a group of five bytes a *parity byte* is inserted into the data stream so that each column of five bits also has a parity check; and that parity bit is found in row P on the table. Thus, the parity bit at the bottom of the first column is one since that column has three other ones. As a final check, the parity byte itself also has a parity bit added.

Table 9.7 is the same as Table 9.6, but Bit Zero, the least significant bit, in Byte One has been changed from a zero to a one (that number is highlighted).

Byte	Data								Parity
1	0	0	0	0	0	0	0	1	0
2	1	0	1	1	0	0	0	0	1
3	1	0	1	1	0	0	1	1	1
4	1	1	1	0	1	0	1	0	1
5	0	1	0	0	0	0	0	0	1
P	1	0	1	0	1	0	0	1	0

Table 9.7: Iterative Parity With Error

In Table 9.7 the parity for Byte One is wrong, and the parity for Bit Zero in the parity byte is wrong; therefore, Bit Zero in Byte One needs to be changed. If the parity bit for a row is wrong, but no column parity bits are wrong, or a column is wrong but no rows are wrong, then the parity bit itself is incorrect. This is one simple way to not only detect data errors, but correct those errors.

There are two weaknesses with iterative parity checking. First, it is restricted to only single-bit errors. If more than one bit is changed in a group then the system fails. This, though, is a general weakness for most parity checking schemes. The second weakness is that a parity byte must be generated and transmitted for every few data bytes (five in the example). This increases the transmission time dramatically and normally makes the system unacceptably slow.

9.4.3 *Hamming Code*

9.4.3.1 *Introduction*

Richard Hamming worked at Bell labs in the 1940s and he devised a way to not only detect that a transmitted byte had changed, but exactly which bit had changed by interspersing parity bits within the data itself. Hamming first defined the "distance" between any two binary words as the number of bits that were different between them. As an example, the two binary numbers 1010 and 1010 has a distance of zero between them since there are no different bits, but 1010 and 1011 has a distance of one since one bit is different. This concept is called the *Hamming Distance* in honor of his work.

The circuit illustrated in Figure 9.13 calculates the Hamming distance between two four-bit numbers. In the illustration, 0100 and 1101 are compared and two bits difference in those two numbers is reported.

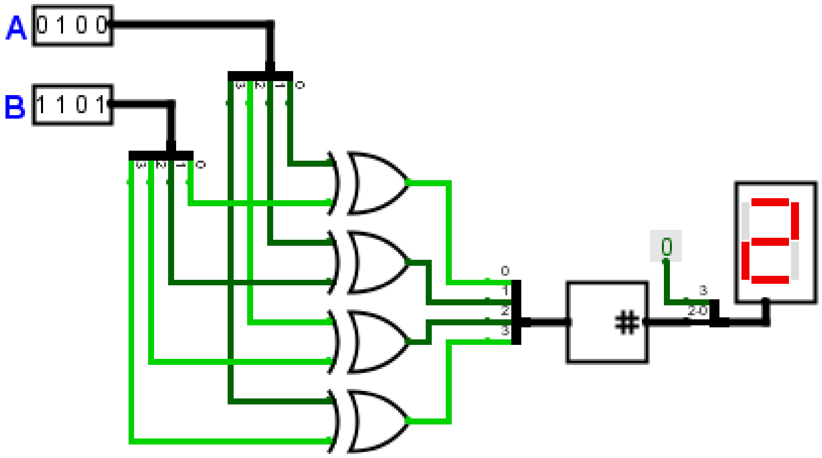


Figure 9.13: Hamming Distance

The four bits for input *A* and input *B* are wired to four XOR gates then the output of those gates is wired to a *Bit Adder* device. The XOR gates will output a one if the two input bits are different then the bit adder will total how many ones are present at its input. The output of the bit adder is a three-bit number but to make it easier to read that number it is wired to a hex digit display. Since that display needs a four-bit input a constant zero is wired to the most significant bit of the input of the hex digit display.

9.4.3.2 *Generating Hamming Code*

Hamming parity is designed so a parity bit is generated for various combinations of bits within a byte in such a way that every data bit is linked to at least three different parity bits. This system can then determine not only that the parity is wrong but which bit is wrong. The cost of a Hamming system is that it adds five parity bits to an eight-bit byte to create a 13-bit word. Consider the bits for the 13-bit word in Table 9.8.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	0	0	0	0	0	0	0	0	0	0	0

Table 9.8: Hamming Parity Bits

The bits numbered P_0 to P_4 are Hamming parity bits and the bits numbered d_0 to d_7 are the data bits. The Hamming parity bits are interspersed with the data but they occur in positions zero, one, three, and seven (counting right to left). The following chart shows the data bits that are used to create each parity bit:

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	X	0	X	0	X	0	X	0	X	0	P
0	0	X	X	0	0	X	X	0	0	X	P	0
0	X	0	0	0	0	X	X	X	P	0	0	0
0	X	X	X	X	P	0	0	0	0	0	0	0
P	X	0	X	X	0	0	X	X	0	X	0	0

Table 9.9: Hamming Parity Cover Table

From Table 9.9, line one shows the data bits that are used to set parity bit zero (P_0). If data bits d0, d1, d3, d4, and d6 are all one then P_0 would be one (even parity is assumed). The data bits needed to create the Hamming parity bit are marked in all five lines. A note is necessary about parity bit P_4. In order to detect transmission errors that are two bits large (that is, two bits were flipped), each data bit needs to be covered by three parity bits. Parity bit P_4 is designed to provide the third parity bit for any data bits that have only two others. For example, look down the column containing data bit d_0 and notice that it has only two parity bits (P_0 and P_1) before P_4. By adding P_4 to the circuit that data bit gets a third parity bit.

As an example of a Hamming code, imagine that this byte needed to be transmitted: 0110 1001. This number could be placed in the data bit positions of the Hamming table.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	0	1	0	0

Table 9.10: Hamming Example - Iteration 1

Bit zero, P0, is designed to generate even parity for data bits d_0, d_1, d_3, d_4, and d_6. Since there are three ones in that group, then P_0 must be one. That has been filled in below (for convenience, the Hamming parity bit pattern for P_0 is included in the last row of the table).

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	0	1	0	1
0	0	X	0	X	0	X	0	X	0	X	0	P

Table 9.11: Hamming Example - Iteration 2

Bit one, P_1, is designed to generate even parity for data bits d_0, d_2, d_3, d_5, and d_6. Since there are four ones in that group, then P_1 must be zero. That has been filled in below.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	0	1	0	1
0	0	X	X	0	0	X	X	0	0	X	P	0

Table 9.12: Hamming Example - Iteration 3

Bit three, P_2, is designed to generate even parity for data bits d_1, d_2, d_3, and d_7. Since there is one one in that group, then P_2 must be one. That has been filled in below.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	1	1	0	1
0	X	0	0	0	0	X	X	X	P	0	0	0

Table 9.13: Hamming Example - Iteration 4

Bit seven, P_3, is designed to generate even parity for data bits d_4, d_5, d_6, and d_7. Since there are two ones in that group, then P_3 must be zero. That has been filled in below.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	1	1	0	1
0	X	X	X	X	P	0	0	0	0	0	0	0

Table 9.14: Hamming Example - Iteration 5

Bit eight, P_4, is designed to generate even parity for data bits d_0, d_1, d_2, d_4, d_5, and d_7. Since there are two ones in that group, then P_4 must be zero. That has been filled in below.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	1	1	0	1
P	X	0	X	X	0	0	X	X	0	X	0	0

Table 9.15: Hamming Example - Iteration 6

When including Hamming parity, the byte 0110 1001 is converted to: 0 0110 0100 1101.

In Figure 9.14, a 11-bit input, A, is used to create a 16-bit word that includes Hamming parity bits. In the illustration, input 010 0111 0111 is converted to 1010 0100 1110 1111.

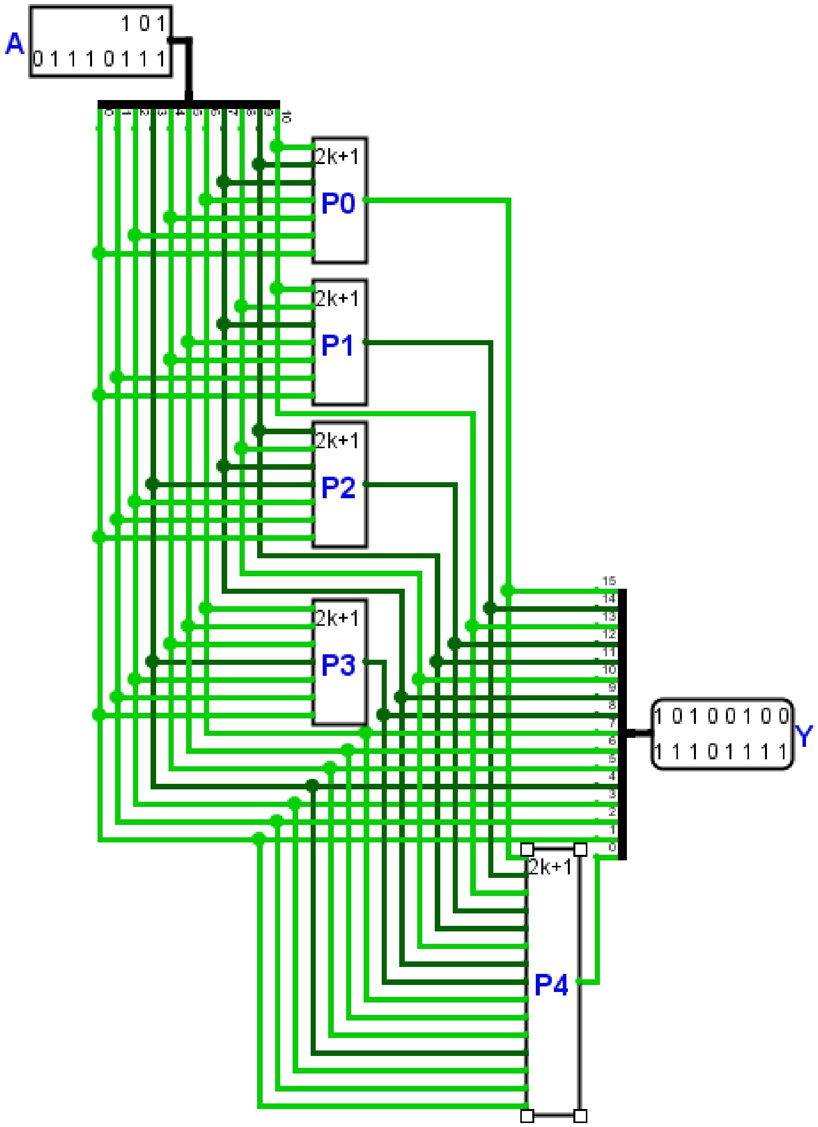


Figure 9.14: Generating Hamming Parity

The process used by the circuit in Figure 9.14 is to wire each of the input bits to various parity generators and then combine the outputs of those parity generators, along with the original bits, into a single 16-bit word. While the circuit has a lot of wired connections the concept is fairly simple. P0 calculates the parity for input bits $0, 1, 3, 4, 6, 8, 10$. That is then wired to the least significant bit of output Y.

9.4.3.3 *Checking Hamming Code*

To check the accuracy of the data bits in a word that contains Hamming parity bits the following general process is used:

1. Calculate the Hamming Parity Bit for each of the bit groups exactly like when the parity was first calculated.

2. Compare the calculated Hamming Parity bits with the parity bits found in the original binary word.
3. If the parity bits match then there is no error. If the parity bits do not match then the bad bit can be corrected by using the pattern of parity bits that do not match.

As an example, imagine that bit eight (last bit in the first group of four) in the Hamming code created above was changed from zero to one: 0 0111 0100 1101 (this is bit d_4). Table 9.16 shows that Hamming Bits P_0, P_3, and P_4 would now be incorrect since d_4 is used to create those parity bits.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	X	0	X	0	X	0	X	0	X	0	P
0	0	X	X	0	0	X	X	0	0	X	P	0
0	X	0	0	0	0	X	X	X	P	0	0	0
0	X	X	X	X	P	0	0	0	0	0	0	0
P	X	0	X	X	0	0	X	X	0	X	0	0

Table 9.16: Hamming Parity Cover Table Reproduced

Since the only data bit that uses these three parity bits is d_4 then that one bit can be inverted to correct the data in the eight-bit byte.

The circuit illusted in Figure 9.15 realizes a Hamming parity check. Notice that the input is the same 16-bit word generated in the circuit in Figure 9.14 except bit eight (the last bit on the top row of the input) has been complemented. The circuit reports that bit eight is in err so it would not only alert an operator that something is wrong with this data but it would also be able to automatically correct the wrong bit.

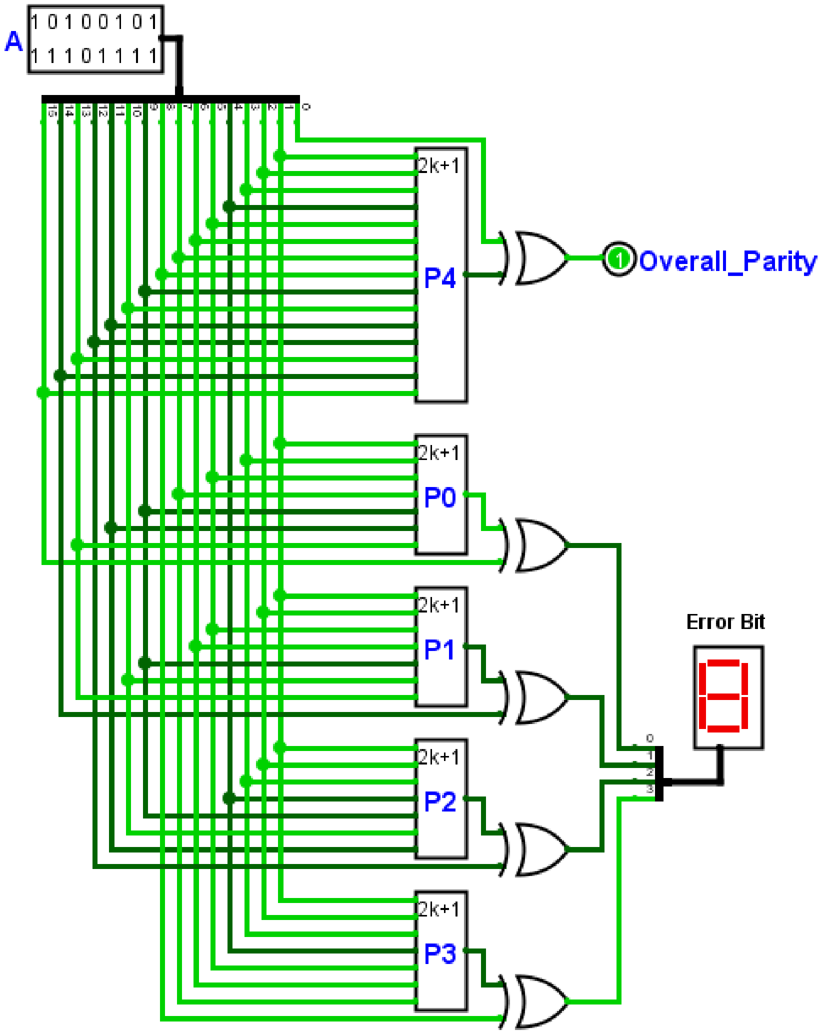


Figure 9.15: Checking Hamming Parity

9.4.4 *Hamming Code Notes*

- When a binary word that includes Hamming parity is checked to verify the accuracy of the data bits using three overlapping parity bits, as developed in this book, one-bit errors can be corrected and two-bit errors can be detected. This type of system is often called SECDED and is commonly used in computer memories to ensure data integrity.

- While it seems wasteful to add five Hamming bits to an eight-bit byte (a 62.5% increase in length), the number of bits needed for longer words does not continue to increase at that rate. Hamming bits are added to a binary word in multiples of powers of two. For example, to cover a 32-bit word only seven Hamming bits are needed, an increase of only about 22.%; and to cover a 256-

bit word only 10 Hamming bits are needed, an increase of just under 4%.

- This lesson counts bits from right-to-left and considers the first position as bit zero, which matches with the bit counting pattern used throughout the book. However, many authors and online resources count Hamming bits from left-to-right and consider the left-most position as bit one because that is a natural way to count.

9.4.5 *Sample Problems*

The following problems are provided for practice.

8-Bit Byte	With Hamming
11001010	1110011011001
10001111	1100001110111
01101101	0011011100111
11100010	0111000010000
10011011	1100111011100

Table 9.17: Hamming Parity Examples

Hamming With Error	Error Bit
0110101011001	1
1100000100110	3
0000100001100	5
1110111011010	9
1110010100110	12

Table 9.18: Hamming Parity Errors

10

SEQUENTIAL REGISTER CIRCUITS

10.1 INTRODUCTION

Electronic circuits that require memory devices (like flip-flops or registers) and feedback loops are designed using is called “Sequential Logic.” The final output of the circuit is determined by both the inputs and internal circuit feedback loops; and the states of the various logic gates may change rapidly as the output from one stage is fed back to various input points. This makes sequential logic circuits much more dynamic, and complex, than combinational circuits, and their analysis normally requires tools like timing diagrams and state maps. Because sequential circuits are able to latch and hold output states, useful applications like flip-flops, registers, counters, and memory are possible.

10.2 TIMING DIAGRAMS

Unlike combinational logic circuits, timing is essential in sequential circuits. Normally, a device will include a *clock* IC that generates a square wave that is used to control the sequencing of various activities throughout the device. In order to design and troubleshoot these types of circuits, a timing diagram is developed that shows the relationship between the various input, intermediate, and output signals. Figure 10.1 is an example timing diagram.

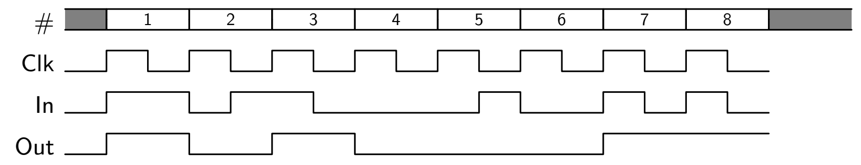


Figure 10.1: Example Timing Diagram

Figure 10.1 is a timing diagram for a device that has three signals, a clock, an Input, and an Output.

Signals in a timing diagram can be described as low//high, 0//1, on//off, or any other number of ways.

- #: The first line of the timing diagram is only a counter that indicates the number of times the system clock has cycled from Low to High and back again. For example, the first time the clock changes from Low to High is the beginning of the first cycle. This counter is only used to facilitate a discussion about the circuit timing.

- CLK: A clock signal regularly varies between Low and High in a predictable pattern, normally such that the amount of Low time is the same as the High time. In a circuit, a clock pulse is frequently generated by applying voltage to a crystal since the oscillation cycle of a crystal is well known and extremely stable. In Figure 10.1 the clock cycle is said have a 50% duty cycle, that is, half-low and half-high. The exact length of a single cycle would be measured in micro- or nano-seconds but for the purposes of this book all that matters is the relationship between the various signals, not the specific length of a clock cycle.
- IN: The input is Low until Cycle #1, then goes High for one cycle. It then toggles between High and Low at irregular intervals.
- OUT: The output goes High at the beginning of cycle #1. It then follows the input, but only at the start of a clock cycle. Notice that the input goes high halfway through cycle #2 but the output does not go high until the start of cycle #3. Most devices are manufactured to be *edge triggered* and will only change their output at either the positive or negative edge of the clock cycle (this example is positive edge triggered). Thus, notice that no matter when the input toggles the output only changes when the clock transitions from Low to High.

Given the timing diagram in Figure 10.1 it would be relatively easy to build a circuit to match the timing requirements. It would have a single input and output port and the output would match the input on the positive edge of a clock cycle.

Ideal circuits with no propagation delay are assumed in this book in order to focus on digital logic rather than engineering.

Whenever the input for any device changes it takes a tiny, but measurable, amount of time for the output to change since the various transistors, capacitors, and other electronic elements must reach saturation and begin to conduct current. This lag in response is known as *propagation delay* and that is important for an engineer to consider when building a circuit. It is possible to have a circuit that is so complex that the output has still not changed from a previous input signal before a new input signal arrives and starts the process over. In Figure 10.2 notice that the output goes High when the input goes Low, but only after a tiny propagation delay.

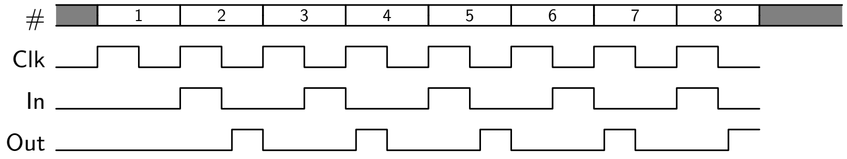


Figure 10.2: Example Propagation Delay

It is also true that a square wave is not exactly square. Due to the presence of capacitors and inductors in circuits, a square wave will actually build up and decay over time rather than instantly change. Figure 10.3 shows a typical charge/discharge cycle for a capacitance circuit and the resulting deformation of a square wave. It should be kept in mind, though, that the times involved for capacitance charge/discharge are quite small (measured in nanoseconds or smaller); so for all but the most sensitive of applications, square waves are assumed to be truly square.

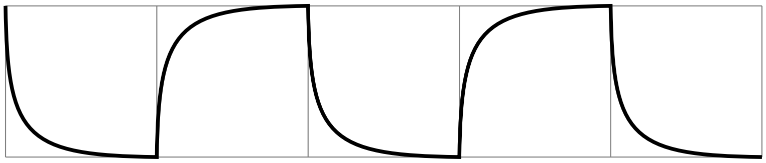


Figure 10.3: Capacitor Charge and Discharge

All devices are manufactured with certain tolerance levels built-in such that when the voltage gets "close" to the required level then the device will react, thus mitigating the effect of the deformed square wave. Also, for applications where speed is critical, such as space exploration or medical, high-speed devices are available that both sharpen the edges of the square wave and reduce propagation delay significantly.

10.3 FLIP-FLOPS

10.3.1 *Introduction*

Flip-flops are digital circuits that can maintain an electronic state even after the initial signal is removed. They are, then, the simplest of memory devices. Flip-flops are often called *latches* since they are able to "latch" and hold some electronic state. In general, devices are called flip-flops when sequential logic is used and the output only changes on a clock pulse and they are called latches when combinational logic is used and the output constantly reacts to the input. Commonly, flip-flops are used for clocked devices, like counters, while latches are used for storage. However, these terms are often considered synonymous and are used interchangeably.

10.3.2 *SR Latch*

One of the simplest of the flip-flops is the SR (for *Set-Reset*) latch. Figure 10.4 illustrates a logic diagram for an *SR latch* built with NAND gates.

This latch is often also called an RS Latch.

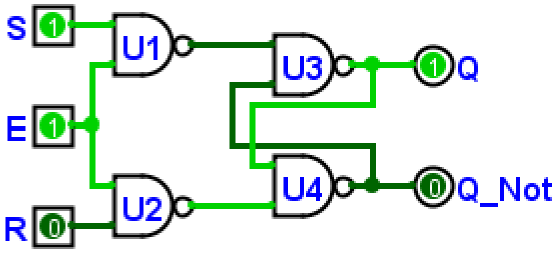


Figure 10.4: SR Latch Using NAND Gates

Table 10.2 is the truth table for an *SR Latch*. Notice that unlike truth tables used earlier in this book, some of the outputs are listed as "Last Q" since they do not change from the previous state.

Inputs			Outputs	
E	S	R	Q	Q'
0	0	0	Last Q	Last Q'
0	0	1	Last Q	Last Q'
0	1	0	Last Q	Last Q'
0	1	1	Last Q	Last Q'
1	0	0	Last Q	Last Q'
1	0	1	0	1
1	1	0	1	0
1	1	1	Not Allowed	Not Allowed

Table 10.1: Truth Table for SR Latch

In Figure 10.4, input E is an enable and must be high for the latch to work; when it is low then the output state remains constant regardless of how inputs S or R change. When the latch is enabled, if $S = R = 0$ then the values of Q and Q' remain fixed at their last value; or, the circuit is "remembering" how they were previously set. When input S goes high, then Q goes high (the latch's output, Q, is "set"). When input R goes high, then Q' goes high (the latch's output, Q, is "reset"). Finally, it is important that in this latch inputs R and S cannot both be high when the latch is enabled or the circuit becomes unstable and output Q will oscillate between high and low as fast as the NAND gates can change; thus, input 111_2 must be avoided. Normally, there is an additional bit of circuit prior to the inputs to ensure S and R will always be different (an XOR gate could do that job).

An SR Latch may or may not include a clock input; if not, then the outputs change immediately when any of the inputs change, like in a combinational circuit. If the designer needs a clocked type of memory device, which is routine, then the typical choice would be a

JK Flip-Flop, covered later. Figure 10.5 is a timing diagram for the SR Latch in Listing 10.4.

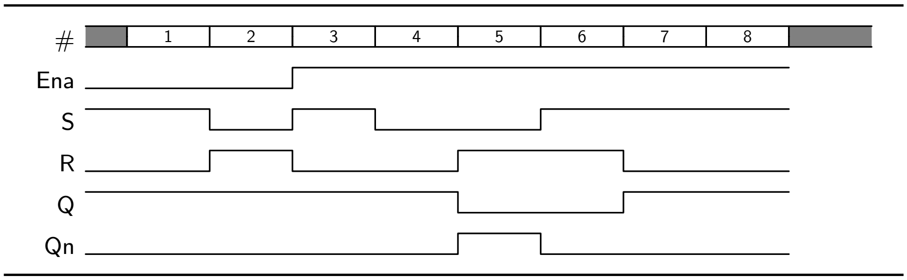


Figure 10.5: SR Latch Timing Diagram

At the start of Figure 10.5 *Enable* is low and there is no change in Q and Q′ until frame 3, when *Enable* goes high. At that point, S is high and R is low so Q is high and Q′ is low. At frame 4 both S and R are low so there is no change in Q or Q′. At frame 5 R goes high so Q goes low and Q′ goes high. In frame 6 both S and R are high so both Q and Q′ go low. Finally, in frame 7 S stays high and R goes low so Q goes high and Q′ stays low.

Logisim-evolution includes an *S-R Flip-Flop* device, as illustrated in Figure 10.6.

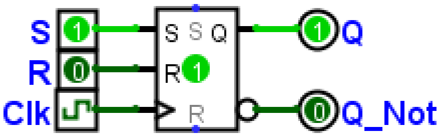


Figure 10.6: SR Latch

The *S-R Flip-Flop* has *S* and *R* input ports and *Q* and *Q′* output ports. Also notice that there is no *Enable* input but there is a *Clock* input. Because this is a clocked device the output would only change on the edge of a clock pulse and that makes the device a flip-flop rather than a latch. As shown, *S* is high and the clock has pulsed so *Q* is high, or the flip-flop is "set." Also notice that on the top and bottom of the device there is an *S* and *R* input port that are not connected. These are "preset" inputs that let the designer hard-set output *Q* at either one or zero, which is useful during a power-up routine. Since this device has no *Enable* input it is possible to use the *R* preset port as a type of enable. If a high is present on the *R* preset then output *Q* will go low and stay there until the *R* preset returns to a low state.

On a logic diagram a clock input is indicated with a triangle symbol.

10.3.3 *Data (D) Flip-Flop*

A Data Flip-Flop (or *D Flip-Flop*) is formed when the inputs for an *SR Flip-Flop* are tied together through an inverter (which also means that

S and *R* cannot be high at the same time, which corrects the potential problem with two high inputs in an *SR Flip-Flop*). Figure 10.7 illustrates an *SR Flip-Flop* being used as a *D Flip-Flop* in *Logisim-evolution* .

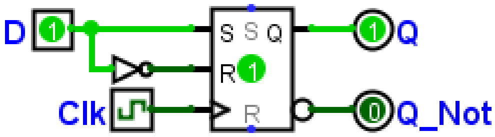


Figure 10.7: D Flip-Flop Using SR Flip-Flop

In Figure 10.7, the *D* input (for "Data") is latched and held on each clock cycle. Even though Figure 10.7 shows the inverter external to the latch circuit, in reality, a *D Flip-Flop* device bundles everything into a single package with only *D* and clock inputs and *Q* and *Q′* outputs, as in Figure 10.8. Like the *RS Flip-Flop*, *D Flip-Flops* also have "preset" inputs on the top and bottom that lets the designer hard-set output *Q* at either one or zero, which is useful during a power-up routine.

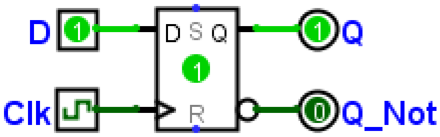


Figure 10.8: D Flip-Flop

Figure 10.9 is the timing diagram for a Data Flip-Flop.

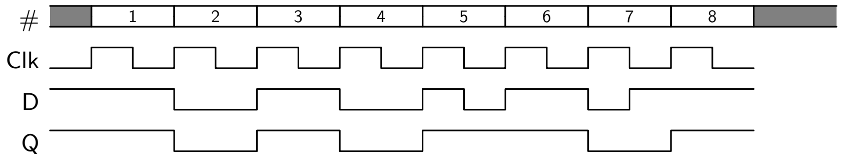


Figure 10.9: D Latch Timing Diagram

In Figure 10.9 it is evident that output *Q* follows input *D* but only on a positive clock edge. The latch "remembers" the value of *D* until the next clock pulse no matter how it changes between pulses.

10.3.4 *JK Flip-Flop*

The *JK flip-flop* is the "workhorse" of the flip-flop family. Figure 10.10 is the logic diagram for a *JK flip-flop*.

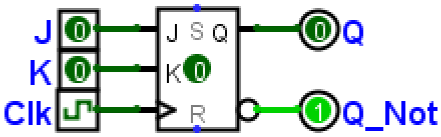


Figure 10.10: JK Flip-Flop

Internally, a *JK flip-flop* is similar to an *RS Latch*. However, the outputs to the circuit (Q and Q') are connected to the inputs (J and K) in such a way that the unstable input condition (both R and S high) of the *RS Latch* is corrected. If both inputs, J and K, are high then the outputs are toggled (they are "flip-flopped") on the next clock pulse. This toggle feature makes the *JK flip-flop* extremely useful in many logic circuits.

Figure 10.11 is the timing diagram for a *JK flip-flop*.

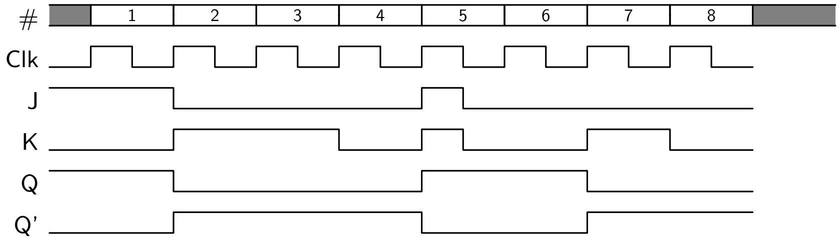


Figure 10.11: JK Flip-Flop Timing Diagram

Table 10.2 summarizes timing diagram 10.11. Notice that on clock pulse five both J and K are high so Q toggles. Also notice that Q' is not indicated in the table since it is always just the complement of Q.

Clock Pulse	J	K	Q
1	H	L	H
2	L	H	L
3	L	H	L
4	L	L	L
5	H	H	H
6	L	L	H
7	L	H	L
8	L	L	L

Table 10.2: JK Flip-Flop Timing Table

10.3.5 *Toggle (T) Flip-Flop*

If the J and K inputs to a *JK Flip-Flop* are tied together, then when the input is high the output will toggle on every clock pulse but when the input is Low then the output remains in the previous state. This is often referred to as a *Toggle Flip-Flop* (or *T Flip-Flop*). *T Flip-Flops* are not usually found in circuits as separate IC since they are so easily created by soldering together the inputs of a standard *JK Flip-Flop*. Figure 10.12 is a *T Flip-Flop* in *Logisim-evolution* .

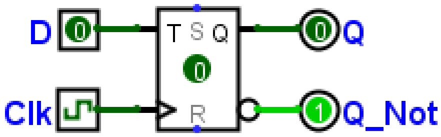


Figure 10.12: T Flip-Flop

Figure 10.13 is the timing diagram for a *T flip-flop*.

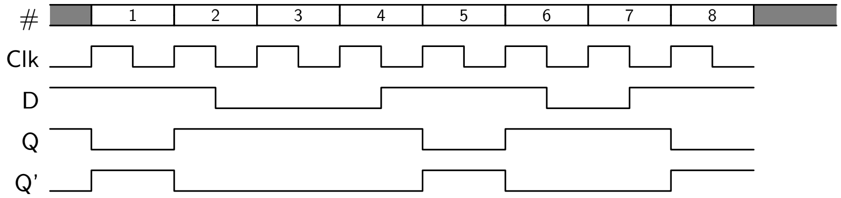


Figure 10.13: Toggle Flip-Flop Timing Diagram

In Figure 10.13 when *D*, the data input, is high then *Q* toggles on the positive edge of every clock cycle.

10.3.6 *Master-Slave Flip-Flops*

Master-Slave Flip-Flops are two flip-flops connected in a cascade and operating from the same clock pulse. These flip-flops tend to stabilize an input circuit and are used where the inputs may have voltage glitches (such as from a push button). Figure 10.14 is the logic diagram for two *JK flip-flops* set up as a Master-Slave Flip-Flop.

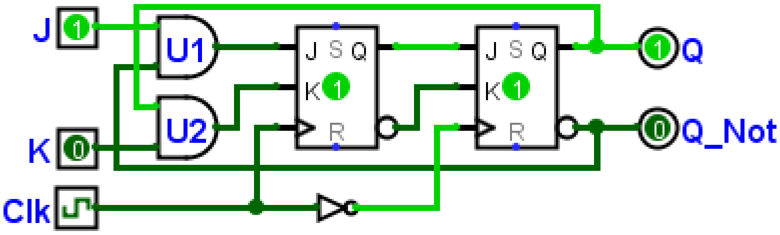


Figure 10.14: Master-Slave Flip-Flop

Because of the NOT gate on the clock signal these two flip-flops will activate on opposite clock pulses, which means the flip-flop one will enable first and read the JK inputs, then flip-flop two will enable and read the output of flip-flop one. The result is that any glitches on the input signals will tend to be eliminated.

Often, only one physical IC is needed to provide the two flip-flops for a master-slave circuit because dual *JK flip-flops* are frequently found on a single IC. Thus, the output pins for one flip-flop could be connected directly to the input pins for the second flip-flop on the same IC. By combining two flip-flops into a single IC package, circuit design can be simplified and fewer components need to be purchased and mounted on a circuit board.

10.4 REGISTERS

10.4.1 *Introduction*

A register is a simple memory device that is composed of a series of flip-flops wired together such that they share a common clock pulse. Registers come in various sizes and types and are often used as "scratch pads" for devices. For example, a register can hold a number entered on a keypad until the calculation circuit is ready do something with that number or a register can hold a byte of data coming from a hard drive until the CPU is ready to move that data someplace else.

10.4.2 *Registers As Memory*

Internally, a register is constructed from *D Flip-Flops* as illustrated in Figure 10.15.

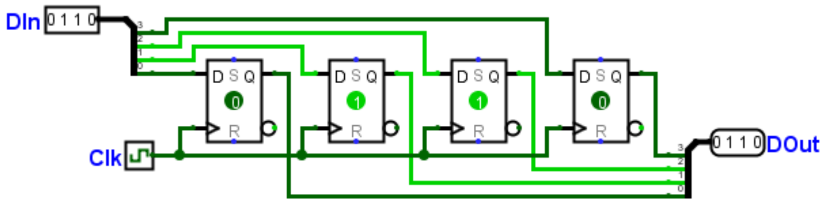


Figure 10.15: 4-Bit Register

Data are moved from the input port, in the upper left corner, into the register; one bit goes into each of the four *D Flip-Flops*. Because each latch constantly outputs whatever it contains, the output port, in the lower right corner, combines the four data bits and displays the contents of the register. Thus, a four-bit register can "remember" some number until it is needed, it is a four-bit memory device.

10.4.2.1 *74x273 Eight-Bit Register*

In reality, designers do not build memory from independent flip-flops, as shown in Figure 10.15. Instead, they use a memory IC that contains the amount of memory needed. In general, purchasing a memory IC is cheaper and more reliable than attempting to build a memory device.

One such memory device is a 74x273 eight-bit register. This device is designed to load and store a single eight-bit number, but other memory devices are much larger, including RAM that can store and retrieve millions of eight-bit bytes.

10.4.3 *Shift Registers*

Registers have an important function in changing a stream of data from serial to parallel or parallel to serial; a function that is called "shifting" data. For example, data entering a computer from a network or USB port is in serial form; that is, one bit at a time is streamed into or out of the computer. However, data inside the computer are always moved in parallel; that is, all of the bits in a word are placed on a bus and moved through the system simultaneously. Changing data from serial to parallel or vice-verse is an essential function and enables a computer to communicate over a serial device.

Figure 10.16 illustrates a four-bit parallel-in/serial-out shift register. The four-bit data into the register is placed on *D0-D3*, the *Shift_Write* bit is set to 1 and the clock is pulsed to write the data into the register. Then the *Shift_Write* bit is set to 0 and on each clock pulse the four bits are shifted right to the *SOUT* (for "serial out") port.

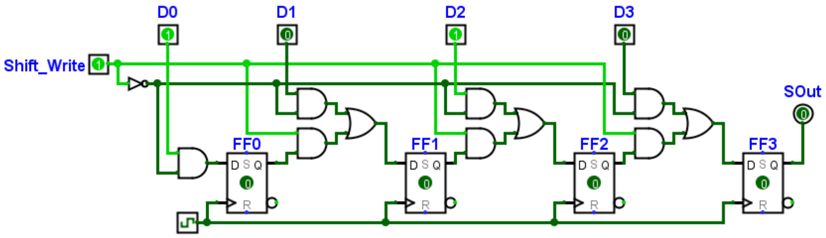

D0
D1
D2
D3
Shift_Write
SOut
FF0
FF1
FF2
FF3
D S Q
R

Figure 10.16: Shift Register

It is possible to also build other shift register configurations: serial-in/serial-out, parallel-in/parallel-out, and serial-in/parallel-out. However, universal shift registers are commonly used since they can be easily configured to work with data in either serial or parallel formats.

11

SEQUENTIAL COUNTERS

11.1 INTRODUCTION

Electronic circuits that require memory devices (like flip-flops or registers) and feedback loops are designed using is called “Sequential Logic.” The final output of the circuit is determined by both the inputs and internal circuit feedback loops; and the states of the various logic gates may change rapidly as the output from one stage is fed back to various input points. This makes sequential logic circuits much more dynamic, and complex, than combinational circuits, and their analysis normally requires tools like timing diagrams and state maps. Because sequential circuits are able to latch and hold output states, useful applications like flip-flops, registers, counters, and memory are possible.

11.2 COUNTERS

11.2.1 *Introduction*

Counters are a type of sequential circuit designed to count input pulses (normally from a clock) and then activate some output when a specific count is reached. Counters are commonly used as timers; thus, all digital clocks, microwave ovens, and other digital timing displays use some sort of counting circuit. However, counters can be used in many diverse applications. For example, a speed gauge is a counter. By attaching some sort of sensor to a rotating shaft and counting the number of revolutions for 60 seconds the RPM is determined. Counters are also commonly used as frequency dividers. If a high frequency is applied to the input of a counter, but only the “tens” count is output, then the input frequency will be divided by ten. As one final example, counters can also be used to control sequential circuits or processes; each count can either activate or deactivate some part of a circuit that controls one of the sequential processes.

11.2.2 *Asynchronous Counters*

One of the simplest counters possible is an asynchronous two-bit counter. This can be built with a two *JK flip-flops* in sequence, as shown in Figure 11.1.

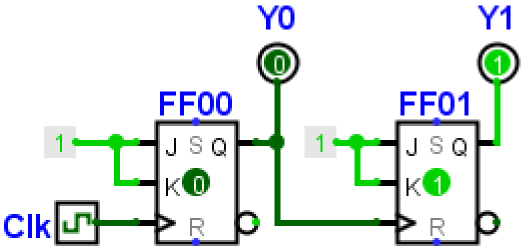


Figure 11.1: Asynchronous 2-Bit Counter

The J-K inputs on both flip-flops are tied high and the clock is wired into *FF00*. On every positive-going clock pulse the *Q* output for *FF00* toggles and that is wired to the clock input of *FF01*, toggling that output. This type of circuit is frequently called a "ripple" counter since the clock pulse must ripple through all of the flip-flops.

In Figure 11.1, assume both flip-flops start with the output low (or 00 out). On the first clock pulse, *Q_FF00* will go high, and that will send a high signal to the clock input of *FF01* which activates *Q_FF01*. At this point, the *Q* outputs for both flip-flops are high (or 11 out). On the next clock pulse, *Q_FF00* will go low, but *Q_FF01* will not change since it only toggles when the clock goes from low to high. At this point, the output is 01. On the next clock pulse, *Q_FF00* will go high and *Q_FF01* will toggle low: 10. The next clock pulse toggles *Q_FF00* to low but *Q_FF01* does not change: 00. Then the cycle repeats. This simple circuit counts: 00, 11, 10, 01. (Note, *Q_FF00* is the low-order bit.) This counter is counting backwards, but it is a trivial exercise to add the functionality needed to reverse that count.

An asynchronous three-bit counter looks much like the asynchronous two-bit counter, except that a third stage is added.

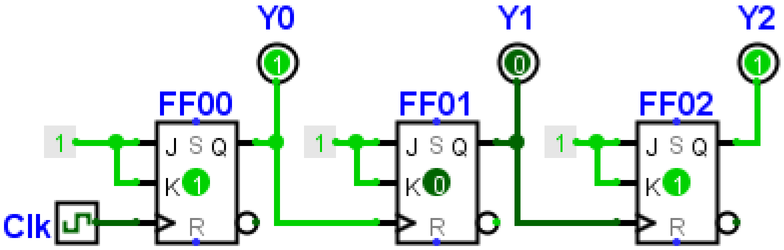


Figure 11.2: Asynchronous 3-Bit Counter

In Figure 11.2, the ripple of the clock pulse from one flip-flop to the next is more evident than in the two-bit counter. However, the overall operation of this counter is very similar to the two-bit counter.

More stages can be added so to any desired number of outputs. Asynchronous (or ripple) counters are very easy to build and require very few parts. Unfortunately, they suffer from two rather important flaws:

- Propagation Delay. As the clock pulse ripples through the various flip-flops, it is slightly delayed by each due to the simple physical switching of the circuitry within the flip-flop. Propagation delay cannot be prevented and as the number of stages increases the delay becomes more pronounced. At some point, one clock pulse will still be winding its way through all of the flip-flops when the next clock pulse hits the first stage and this makes the counter unstable.
- Glitches. If a three-bit counter is needed, there will be a very brief moment while the clock pulse ripples through the flip-flops that the output will be wrong. For example, the circuit should go from 111 to 000, but it will actually go from 111 to 110 then 100 then 000 as the "low" ripples through the flip-flops. These glitches are very short, but they may be enough to introduce errors into a circuit.

Figure 11.3 is a four-bit asynchronous counter.

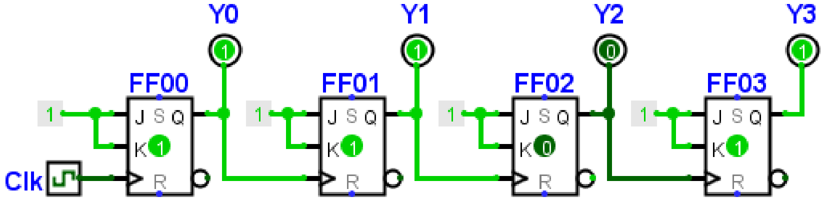


Figure 11.3: Asynchronous 4-Bit Counter

Figure 11.4 is the timing diagram obtained when the counter in Figure 11.3 is executing.

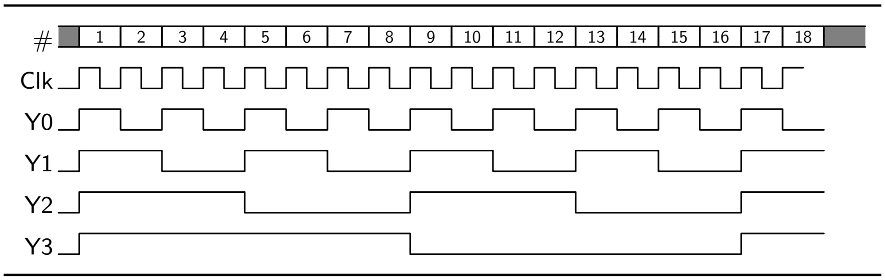


Figure 11.4: 4-Bit Asynchronous Counter Timing Diagram

Notice that *Y0* counts at half the frequency of the clock and then *Y1* counts at half that frequency and so forth. Each stage that is added will count at half the frequency of the previous stage.

11.2.3 *Synchronous Counters*

Both problems with ripple counters can be corrected with a synchronous counter, where the same clock pulse is applied to every flip-flop at one time. Here is the logic diagram for a synchronous two-bit counter:

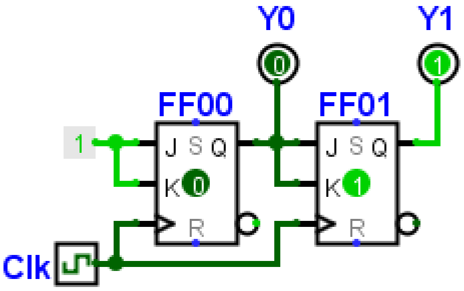


Figure 11.5: Synchronous 2-Bit Counter

Notice in this circuit, the clock is applied to both flip-flops and control is exercised by applying the output of one stage to both J and K inputs of the next stage, which effectively enables/disables that stage. When Q_FF00 is high, for example, then the next clock pulse will make FF01 change states.

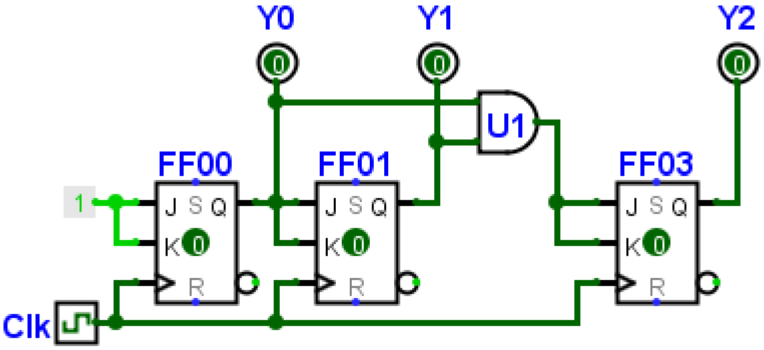


Figure 11.6: Synchronous 3-Bit Counter

A three-bit synchronous counter applies the clock pulse to each flip-flop. Notice, though, that the output from the first two stages must routed through an AND gate so *FF03* will only change when *Q_FF00* and *Q_FF01* are high. This circuit would then count properly from 000 to 111.

In general, synchronous counters become more complex as the number of stages increases since it must include logic for every stage to determine when that stage should activate. This complexity results in greater power usage (every additional gate requires power) and heat generation; however, synchronous counters do not have the propagation delay problems found in asynchronous counters.

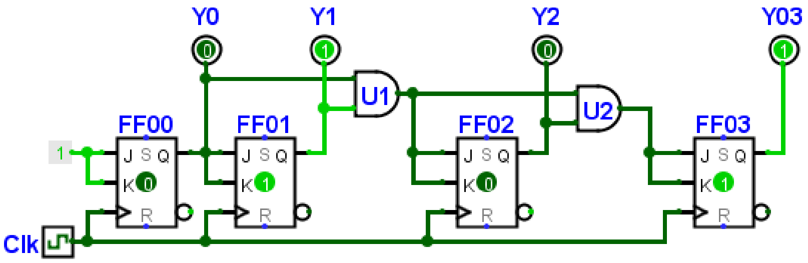


Figure 11.7: Synchronous 4-Bit Up Counter

Figure 11.8 is the timing diagram for the circuit illustrated in Figure 11.7.

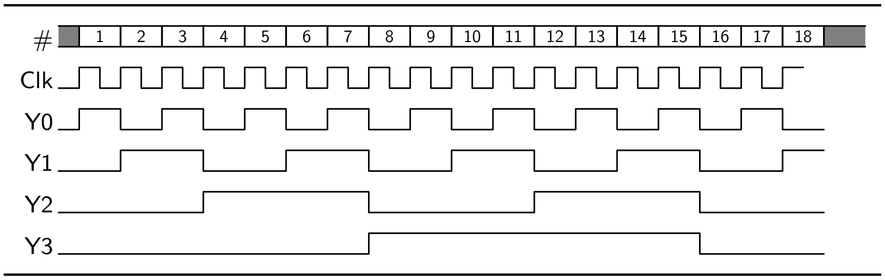


Figure 11.8: 4-Bit Synchronous Counter Timing Diagram

The timing diagram for the asynchronous counter in Figure 11.4 is the same as that for an synchronous counter in Figure 11.8 since the output of the two counters are identical. The only difference in the counters is in how the count is obtained, and designers would normally opt for a synchronous IC since that is a more efficient circuit.

11.2.3.1 *Synchronous Down Counters*

It is possible to create a counter that counts down rather than up by using the Q' outputs of the flip-flops to trigger the next stage. Figure 11.9 illustrates a four-bit down counter.

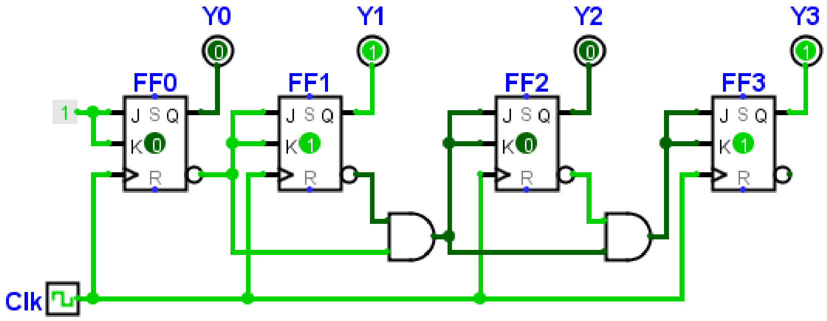


Figure 11.9: Synchronous 4-Bit Down Counter

Figure 11.10 is the timing diagram for the circuit illustrated in Figure 11.9.

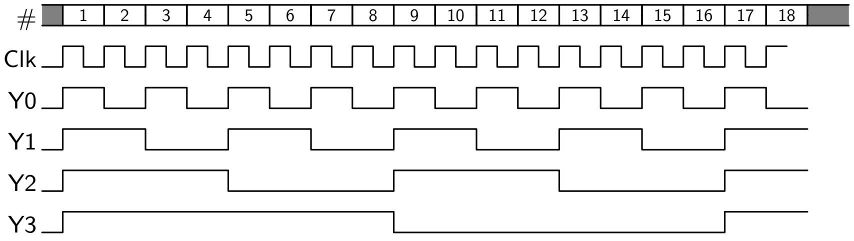


Figure 11.10: 4-Bit Synchronous Down Counter Timing Diagram

11.2.4 *Ring Counters*

A ring counter is a special kind of counter where only one output is active at a time. As an example, a four-bit ring counter may output this type of pattern:

1000 — 0100 — 0010 — 0001

Notice the high output cycles through each of the bit positions and then recycles. Ring counters are useful as controllers for processes where one process must follow another in a sequential manner. Each of the bits from the ring counter can be used to activate a different part of the overall process; thus ensuring the process runs in proper sequence. The circuit illustrated in Figure 11.11 is a 4-bit ring counter.

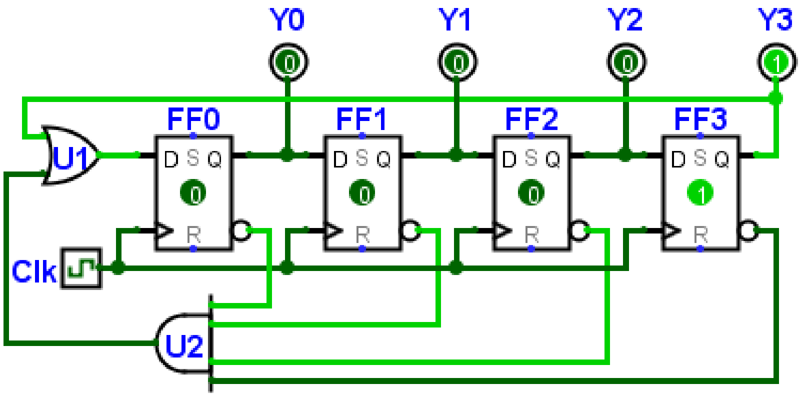


Figure 11.11: 4-Bit Ring Counter

When the circuit is initialized none of the flip-flops are active. Because Q' is high for all flip-flops, AND gate *U2* is active and that sends a high through *U1* and into the data port of *FF0*. The purpose of *U2* is to initialize *FF0* for the first count and then *U2* is never activated again. On each clock pulse the next flip-flop in sequence is activated. When *FF3* is active that output is fed back through *U1* to *FF0*, completing the ring and re-starting the process.

Figure 11.12 is the timing diagram for the ring counter illustrated in Figure 11.11.

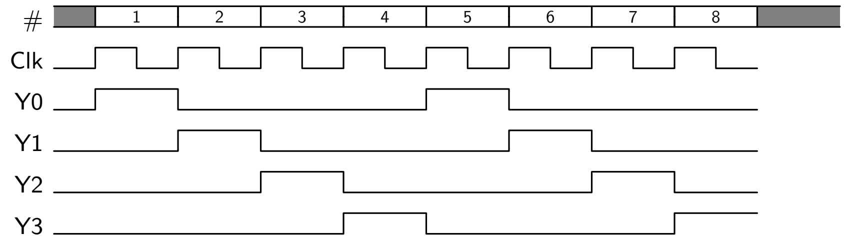


Figure 11.12: 4-Bit Ring Counter Timing Diagram

On each clock pulse a different bit is toggled high, proceeding around the four-bit nibble in a ring pattern.

11.2.4.1 *Johnson Counters*

One common modification of a ring counter is called a Johnson, or “Twisted Tail,” ring counter. In this case, the counter outputs this type of pattern.

$$1000 - 1100 - 1110 - 1111 - 0111 - 0011 - 0001 - 0000$$

The circuit illustrated in Figure 11.13 is a 4-bit Johnson counter.

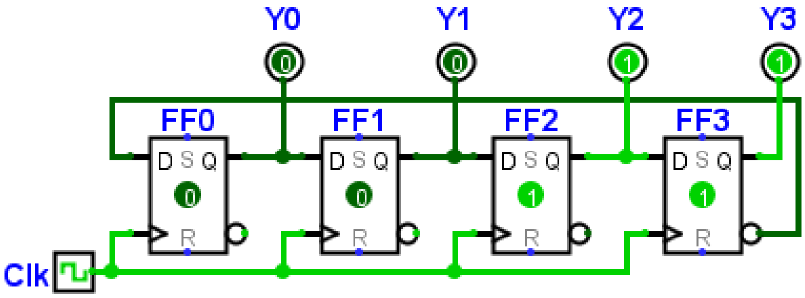


Figure 11.13: 4-Bit Johnson Counter

This is very similar to the ring counter except the feedback loop is from Q' rather than Q of $FF3$ and because of that, the AND gate initialization is no longer needed.

Figure 11.14 is the timing diagram for the Johnson counter illustrated in Figure 11.13.

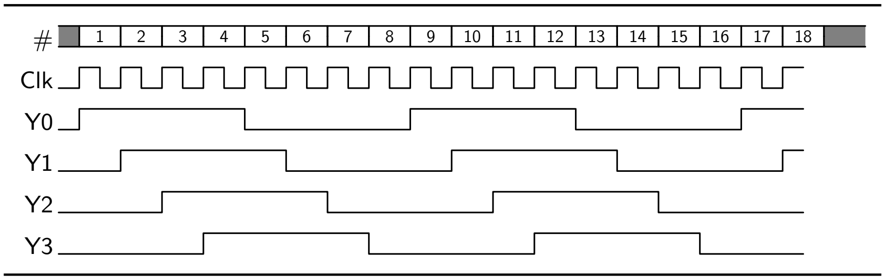


Figure 11.14: 4-Bit Johnson Counter Timing Diagram

11.2.5 *Modulus Counters*

Each of the counters created so far have had one significant flaw, they only count up to a number that is a power of two. A two-bit counter counts from zero to three, a three-bit counter counts from zero to seven, a four-bit counter counts from zero to 15, and so forth. With a bit more work a counter can be created that stops at some other number. These types of counters are called “modulus counters” and an example of one of the most common modulus counters is a decade counter that counts from zero to nine. The circuit illusted in Figure 11.15 is a decade counter.

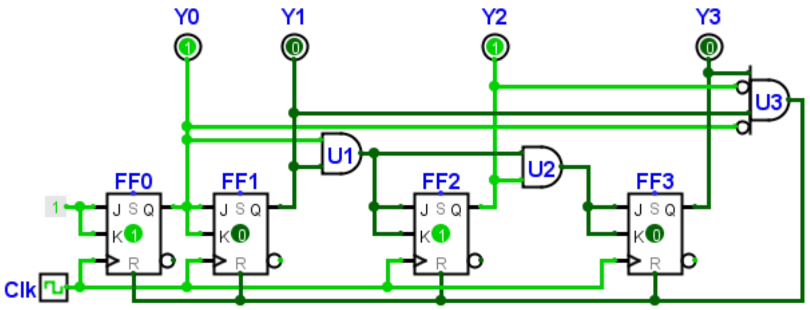


Figure 11.15: Decade Counter

This is only a four-bit counter (found in Figure 11.7) but with an additional AND gate (*U3*). The inputs for that AND gate are set such that when *Y0*-*Y3* are 1010 then the gate will activate and reset all four flip-flops.

Figure 11.16 is the timing diagram obtained from the decade counter illustrated in Figure 11.15.

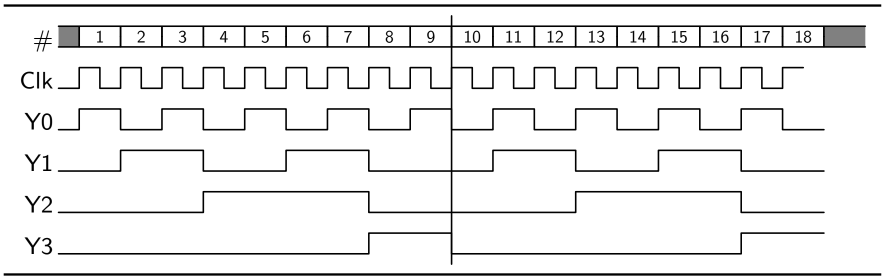


Figure 11.16: 4-Bit Decade Counter Timing Diagram

The timing diagram shows the count increasing with each clock pulse (for example, at 8 the outputs are 1000) until pulse number ten, when the bits reset to 0000 and the count starts over. A vertical line was added at count ten for reference.

11.2.6 *Up-Down Counters*

In this chapter both up and down counters have been considered; however counters can also be designed to count both up and down. These counters are, of course, more complex than the simple counters encountered so far and Figure 11.17 illustrates an up-down counter circuit.

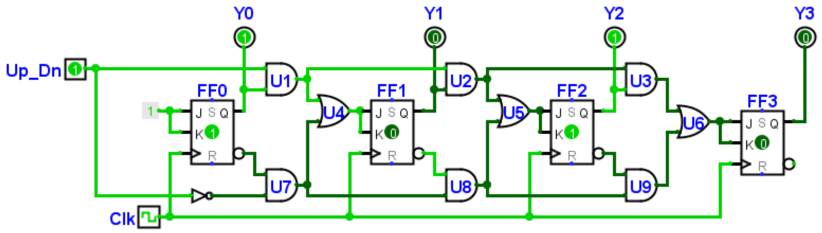


Figure 11.17: Up-Down Counter

When the *Up_Dn* bit is high the counter counts up but when that bit is low then the counter counts down. This counter is little more than a merging of the up counter in Figure 11.7 and the down counter in Figure 11.9. *U1-U3* transmit the *Q* outputs from each flip-flop to the next stage while *U7-U9* transmit the Q' outputs from each flip-flop to the next stage. The E bit activates one of those two banks of AND gates.

No timing diagram is provided for this circuit since that diagram would be the same as for the up counter (Figure 11.8 or the down counter (Figure 11.10), depending on the setting of the *Up_Dn* bit.

11.2.7 *Frequency Divider*

Often, a designer creates a system that may need various clock frequencies throughout its subsystems. In this case, it is desirable to have only one main pulse generator, but divide the frequency from that generator so other frequencies are available where they are needed. Also, by using a common clock source all of the frequencies are easier to synchronize when needed.

The circuit illustrated in Figure 11.18 is the same synchronous two-bit counter first considered in Figure 11.5.

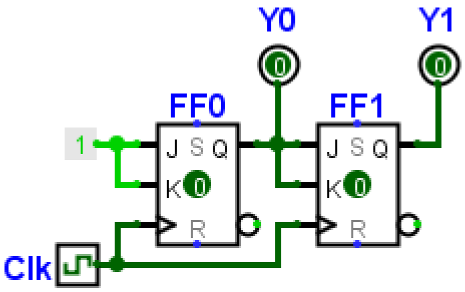


Figure 11.18: Synchronous 2-Bit Counter

The circuit in Figure 11.18 produces the timing diagram seen in Figure 11.19

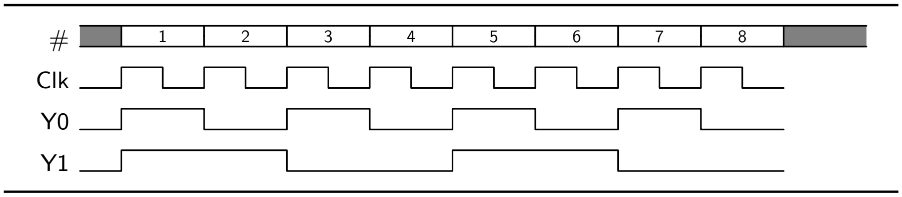


Figure 11.19: Frequency Divider

Notice that *Y0* is half of the clock's frequency and *Y1* is one-quarter of the clock's frequency. If the circuit designer used Y1 as a timer for some subcircuit then that circuit would operate at one-quarter of the main clock frequency. It is possible to use only one output port from a modulus counter, like the decade counter found in Figure 11.15, and get any sort of division of the main clock desired.

11.2.8 *Counter Integrated Circuits (IC)*

In practice, most designers do not build counter circuits since there are so many types already commercially available. When a counter is needed, a designer can select an appropriate counter from those available on the market. Here are just a few as examples of what is available:

IC	Function
74x68	dual four-bit decade counter
74x69	dual four-bit binary counter
74x90	decade counter (divide by 2 and divide by 5 sections)
74x143	decade counter/latch/decoder/seven-segment driver
74x163	synchronous four-bit binary counter
74x168	synchronous four-bit up/down decade counter
74x177	presettable binary counter/latch
74x291	four-bit universal shift register and up/down counter

Table 11.1: Counter IC's

11.3 MEMORY

11.3.1 *Read-Only Memory*

ROM is an IC that contains tens-of-millions of registers (memory locations) to store information. Typically, ROM stores microcode (like bootup routines) and computer settings that do not change. There are several types of ROM: mask, programmable, and erasable.

- Mask ROM are manufactured such that memory locations already filled with data and that cannot be altered by the end user. They are called "mask" ROM since the manufacturing process includes applying a mask to the circuit as it is being created.
- PROM are a type of ROM that can be programmed by the end user, but it cannot be altered after that initial programming. This permits a designer to distribute some sort of program on a chip that is designed to be installed in some device and then never changed.
- Erasable PROM can be programmed by the end user and then be later altered. An example of where an erasable PROM would be used is in a computer's BIOS (the operating system that boots up a computer), where the user can alter certain "persistent" computer specifications, like the device boot order.

Two major differences between ROM and RAM are 1) ROM's ability to hold data when the computer is powered off and 2) altering the data in RAM is much faster than in an erasable PROM.

11.3.2 *Random Access Memory*

RAM is an integrated circuit that contains tens-of-millions of registers (memory locations) to store information. The RAM IC is designed to quickly store data found at its input port and then look-up and return stored data requested by the circuit. In a computer operation, RAM contains both program code and user input (for example, the *LibreOffice Writer* program along with whatever document the user is working on). There are two types of RAM: dynamic and static. DRAM stores bits in such a way that it must be "refreshed" (or "strobed") every few milliseconds. SRAM uses flip-flops to store data so it does not need to be refreshed. One enhancement to DRAM was to package several in a single IC and then synchronize them so they act like a single larger memory; these are called SDRAM and they are very popular for camera and cell phone memory.

Both ROM and RAM circuits use registers, flip-flops, and other components already considered in this chapter, but by the millions rather than just four or so at a time. There are no example circuits or timing diagrams for these devices in this book; however, the lab manual that accompanies this book includes activities for both ROM and RAM devices.

12

SIMULATION

12.1 INTRODUCTION

One of the important benefits of using a simulator like *Logisim-evolution* is that a circuit designer can simulate digital systems to determine logic flaws or weaknesses that must be addressed before a physical IC is manufactured. This chapter introduces the concept of simulation and includes several examples of simulated systems.

12.2 FINITE STATE MACHINES

12.2.1 *Introduction*

Sequential logic circuits are dynamic and the combined inputs and outputs of the circuit at any given stable moment is called a *state*. Over time, a circuit changes states as triggering events take place. As an example, a traffic signal may be green at some point but change to red because a pedestrian presses the "cross" button. The current state of the system would be "green" but the triggering event (the "cross" button) changes the state to "red."

The mathematical model of a sequential circuit is called a FSM. A FSM is an abstract model of a sequential circuit where each state of the circuit is indicated by circles and various triggering events are used to sequence a process from one state to the next. The behavior of many devices can be represented by a FSM model, including traffic signals, elevators, vending machines, and robotic devices. FSM are an analysis tool that can help to simplify sequential circuits.

There are two fundamental FSM models: Moore and Mealy. These two models are generalizations of a state machine and differ only in the way that the outputs are generated. A Moore machine generates an output as a function of only the current state while a Mealy machine generates an output as a function of the current state plus the inputs into that state. Moore machines tend to be safer since they are only activated on a clock pulse and are less likely to create unwanted feedback when two different modules are interconnected; however, Mealy machines tend to be simpler and have fewer states than a Moore machine. Despite the strengths and weaknesses for each of these two FSM, in reality, they are so similar that either can be effectively used to model any given circuit and the actual FSM chosen by the designer is often little more than personal preference.

12.2.2 *Moore Finite State Machine*

The Moore FSM is named after Edward F. Moore, who presented the concept in a 1956 paper, *Gedanken-experiments on Sequential Machines*. The output of a Moore FSM depends only on its current state. The Moore FSM is typically simpler than a Mealy FSM so modeling hardware systems is usually best done using a Moore FSM.

As an example of a Moore FSM imagine a simple candy vending machine that accepts either five cents or ten cents at a time and vends a handful of product when 15 cents has been deposited (no change is returned). Figure 12.1 is a Moore FSM diagram for this vending machine.

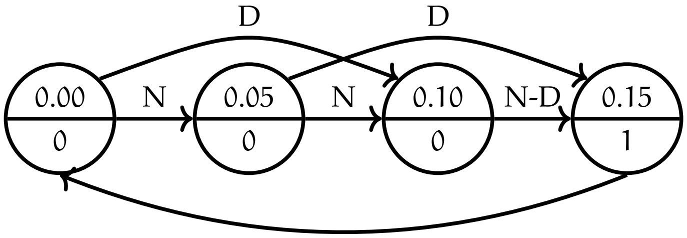


Figure 12.1: Moore Vending Machine FSM

In Figure 12.1, imagine that five cents is deposited between each state circle (that action is indicated by the arrows labeled with an N, for Nickel). The output at each state is zero (printed at the bottom of each circle) until the state reaches 0.15 in the last circle, then the output changes to one (the product is vended). After that state is reached the system resets to state 0.00 and the entire process starts over. If a user deposits ten cents, a Dime, then one of the nickel states is skipped.

12.2.3 *Mealy Finite State Machine*

The Mealy machine is named after George H. Mealy, who presented the concept in a 1955 paper, *A Method for Synthesizing Sequential Circuits*. The Mealy FSM output depends on both its current state and the inputs. Typically a Mealy machine will have fewer states than a Moore machine, but the logic to move from state to state is more complex.

As an example of a Mealy FSM, the simple candy vending machine introduced in Figure 12.1 can be redesigned. Recall that the machine accepts either five cents or ten cents at a time and vends a handful of product when 15 cents has been deposited (no change is returned). Figure 12.2 is a Mealy FSM diagram for this vending machine.

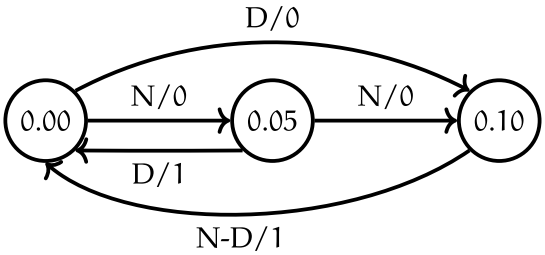


Figure 12.2: Mealy Vending Machine FSM

In Figure 12.2 the states are identified by the amount of money that has been deposited, so the first state on the left (0.00) is when no money has been deposited. Following the path directly to the right of the first state, if five cents (indicated by "N" for a nickel) is deposited, then the output is zero (no candy is dispensed) and the state is changed to 0.05. If another five cents is deposited, the output remains zero and the state changes to 0.10. At that point, if either five or ten cents ("N" or "D") is deposited, then the output changes to 1 (candy is dispensed) and the state resets to 0.00. By following the transition arrows various combinations of inputs and their resulting output can be traced.

Because the Mealy FSM reacts immediately to any input it requires one less state than the Moore machine (compare Figures 12.1 and 12.2). However, since Moore machines only change states on a clock pulse they tend to be more predictable (and safer) when integrated into other modules. Finally, Mealy machines tend to react faster than Moore machines since they do not need to wait for a clock pulse and, generally, are implemented with fewer gates. In the end, whether to design with a Mealy or a Moore machine is left to the designer's discretion and, practically speaking, most designers tend to favor one type of FSM over the other. The simulations in this book use Moore machines because they tend to be easier to understand and more stable in operation.

12.2.4 *Finite State Machine Tables*

Many designers enjoy using Moore and Mealy FSM diagrams as presented in Figures 12.1 and 12.2; however, others prefer to design with a finite state machine table that lists all states, inputs, and outputs. As an example, imagine a pedestrian crosswalk in the middle of a downtown block. The crosswalk has a traffic signal to stop traffic and a *Cross / Don't Cross* light for pedestrians. It also has a button that pedestrians can press to make the light change so it is safe to cross.

This system has several states which can be represented in a *State Table*. The traffic signal can be Red, Yellow, or Green and the pedestrian signal can be Walk or Don't Walk. Each of these signals can be either zero (for *Off*) or one (for *On*). Also, there are two triggers for the circuit; a push button (the *Cross* button that a pedestrian presses) and a timer (so the walk light will eventually change to don't walk). If the button is assumed to be one when it is pressed and zero when not pressed, and the timer is assumed to be one when a certain time interval has expired but zero otherwise, then State Table 12.1 can be created.

	Current State					**Trigger**		**Next State**				
	R	Y	G	W	D	Btn	Tmr	R	Y	G	W	D
1	0	0	1	0	1	0	0	0	0	1	0	1
2	0	0	1	0	1	1	0	0	1	0	0	1
3	0	1	0	0	1	X	0	1	0	0	1	0
4	1	0	0	1	0	X	1	0	0	1	0	1

Table 12.1: Crosswalk State Table

The various states are R (for *Red*), Y (for *Yellow*), and G (for *Green*) traffic lights, and W (for *Walk*) and D (for *Don't Walk*) pedestrian lights. The *Btn* (for *Cross Button*) and *Tmr* (for *Timer*) triggers can, potentially, change the state of the system.

Row One on this table shows that the traffic light is Green and the pedestrian light is Don't Walk. If the button is not pressed (it is zero) and the timer is not active (it is zero), then the next state is still a Green traffic light and Don't Walk pedestrian light; in other words, the system is quiescent. In Row Two, the button was pressed (*Btn* is one); notice that the traffic light changes state to Yellow, but the pedestrian light is still Don't Walk. In Row Three, the current state is a Yellow traffic light with Don't Walk pedestrian light (in other words, the *Next State* from Row Two), the X for the button means it does not matter if it is pressed or not, and the next state is a Red traffic light and Walk pedestrian light. In Row Four, the timer expires (it changes to one at the end of the timing cycle), and the traffic light changes back to Green while the pedestrian light changes to Don't Walk.

While Table 12.1 represents a simplified traffic light system, it could be extended to cover all possible states. Since Red, Yellow, and Green can never all be one at one time, nor could Walk and Don't Walk, the designer must specifically define all of the states rather than use a simple binary count from 00000 to 11111. Also, the designer must be certain that some combinations never happen, like a Green traffic light and a Walk pedestrian light at the same time, so those must be carefully avoided.

State tables can be used with either Mealy or Moore machines and a designer could create a circuit that would meet all of the requirements from the state table and then realize that circuit to actually build a traffic light system.

13

SIMULATION

13.1 INTRODUCTION

One of the important benefits of using a simulator like *Logisim-evolution* is that a circuit designer can simulate digital systems to determine logic flaws or weaknesses that must be addressed before a physical IC is manufactured. This chapter introduces the concept of simulation and includes several examples of simulated systems.

13.2 ELEVATOR

As a simple example of a FSM, imagine an elevator control circuit. For simplicity, this elevator is in a five story building and buttons like “door open” will be ignored. There are two ways the elevator can be called to a given floor: someone could push the floor button on the elevator to ride to that floor or someone could push the call button beside the elevator door on a floor.

Figure 13.1 is the Moore FSM for this circuit:

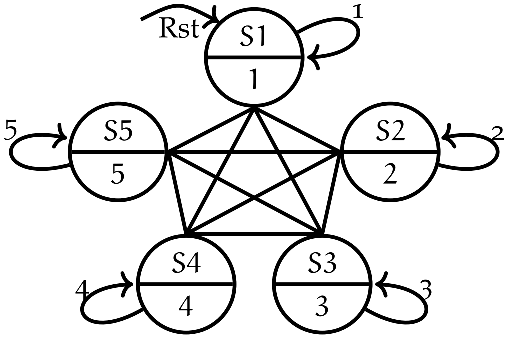


Figure 13.1: Elevator

In Figure 13.1 the various floors are represented by the five states (S1-S5). While the elevator is at a floor then the output from the circuit is the floor number. The elevator is quiescent while in any one state and if a user pushes the button for that same floor then nothing will happen, which is illustrated by the loops starting and coming back to the same state. The star pattern in the middle of the FSM illustrates that the elevator can move directly from one state to any other state. Finally, a Reset signal will automatically move the elevator to state S1.

14

SIMULATION

14.1 INTRODUCTION

One of the important benefits of using a simulator like *Logisim-evolution* is that a circuit designer can simulate digital systems to determine logic flaws or weaknesses that must be addressed before a physical IC is manufactured. This chapter introduces the concept of simulation and includes several examples of simulated systems.

14.2 CENTRAL PROCESSING UNIT

14.2.1 *Introduction*

The CPU is the core of any computer, tablet, phone, or other computer-like device. While many definitions of CPU have been offered, many quite poetic (like the "heart" or "brain" of a computer); probably the best definition is that the CPU is the intersection of software and hardware. The CPU contains circuitry that converts the ones and zeros that are stored in memory (a "program") into controlling signals for hardware devices. The CPU retrieves and analyzes bytes that are contained in a program, turns on or off multiplexers and control buffers so data are moved to or from various devices in the computer, and permits humans to use a computer for intellectual work. In its simplest form, a CPU does nothing more than fetch a list of instructions from memory and then execute those instructions one at a time. This chapter explores CPU from a theoretical perspective.

14.2.1.1 *Concepts*

A CPU processes a string of ones and zeros and uses the information in that binary code to enable/disable circuits or hardware devices in a computer system. For example, a CPU may execute a binary code and create electronic signals that first place data on the computer's data bus and then spins up the hard drive to store that data. As another example, perhaps the CPU detects a key press on a keyboard, transfers that key's code to memory and also sends a code to the monitor where specific pixels are activated to display the letter pressed.

Figure 14.1 illustrates a very simple circuit used to control the flow of data on a data bus.

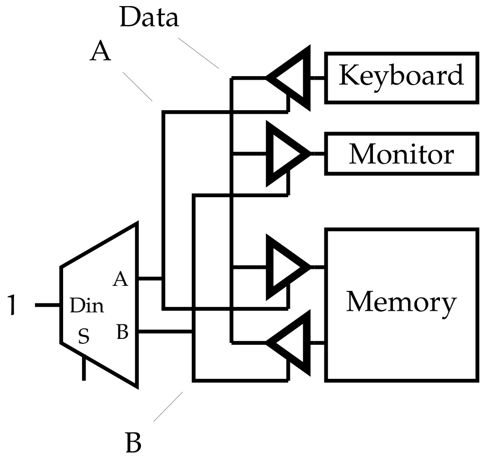

Data
A
Keyboard
Monitor
Memory
1
Din
S
A
B
B

Figure 14.1: Simple Data Flow Control Circuit

In Figure 14.1, the demultiplexer at the bottom left corner of the circuit controls four control buffers that, in turn, control access to the data bus. When output A is active then input from the Keyboard is stored in Memory; but when output B is active then output from memory is sent to the monitor. By setting the select bit in the demultiplexer the circuit's function can be changed from reading the keyboard to writing to the monitor using a single data bus.

In a true CPU, of course, there are many more peripheral devices, along with an ALU, registers, and other internal resources to control. Figure 14.2 is a block diagram for a simplified CPU.

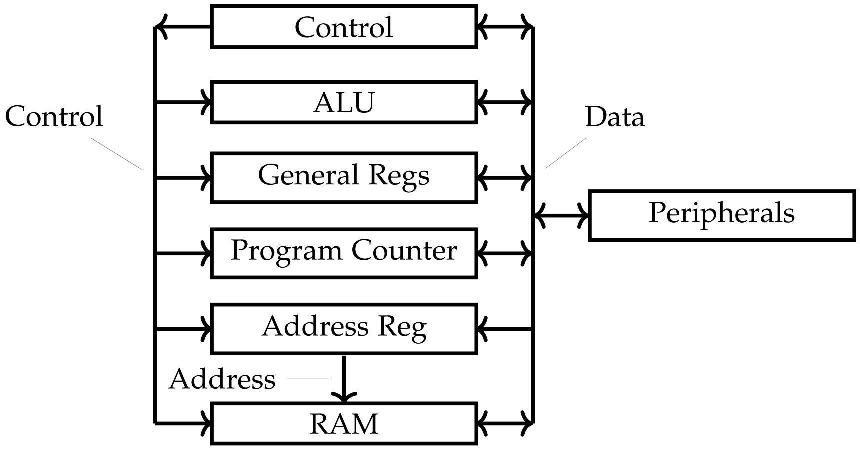

Control
ALU
General Regs
Program Counter
Address Reg
Address
RAM
Control
Data
Peripherals

Figure 14.2: Simplified CPU Block Diagram

The CPU in Figure 14.2 has three bus lines:

- Control. This bus contains all of the signals needed to activate control buffers, multiplexers, and demultiplexers in order to move data.
- Data. This contains the data being manipulated by the CPU.

- ADDRESS. This is the address for the next instruction to fetch from RAM.

There are several blocks in the CPU in Figure 14.2:

- CONTROL. This block contains the circuitry necessary to decode an instruction that was fetched from RAM and then activate the various devices needed to control the flow of data within the CPU.
- ALU. This is an ALU designed for the application that is using this CPU.
- GENERAL REGISTERS. Most CPU include a number of general registers that temporarily hold binary numbers or instructions.
- PROGRAM COUNTER. This is a register that contains the address of the next instruction to fetch from RAM.
- ADDRESS REGISTER. This contains the address for the current RAM operation.
- RAM. While most computers and other devices have a large amount of RAM outside the CPU, many CPU are constructed with a small amount of internal RAM for increased operational efficiency. This type of high-speed RAM is usually called cache (pronounced "cash").

In operation, the CPU moves the value of the Program Counter to the Address Register and then fetches the instruction contained at that RAM address. That instruction is then sent to the Control circuit where it is decoded. The Control circuit activates the appropriate control devices to execute the instruction. This process is repeated millions of times every second while the CPU executes a program.

14.2.1.2 *History*

CPU from the early days of computing (circa 1950) were custom made for the computer on which they were found. Computers in those early days were rather rare and there was no need for a general-purpose CPU that could function on multiple platforms. By the 1960s *IBM* designed a family of computers based on the design of the *System*/360, or *S*/360. The goal was to have a number of computers use the same CPU so programs written for one computer could be executed on another in the same family. By doing this, *IBM* hoped to increase their customer loyalty.

CPU today can be divided into two broad groups: CISC (pronounced like "sisk") and RISC. Early computers, like the IBM S/360 had a large, and constantly growing, set of instructions, and these types of CPU were referred to as CISC. However, building the circuits needed to execute all of those instructions became ever more challenging until, in the late 1980s, computer scientists began to design RISC CPU with fewer instructions. Because there were fewer CPU instructions circuits could be smaller and faster, but the trade off was that occasionally a desired instruction had to be simulated by combining two or more other instructions, and that creates longer, more complex computer programs. Nearly all computers, cell phones, tablets, and other computing devices in use today use a RISC architecture.

More recent developments in CPU include "pipelining" where the CPU can execute two or more instructions simultaneously by overlapping them, that is, fetching and starting an instruction while concurrently finishing the previous instruction. Another innovation changed compilers such that they can create efficient VLIW codes that combine several instructions into a single step. Multi-threading CPU permit multiple programs to execute simultaneously and multi-core CPU use multiple CPU cores on the same substrate so programs can execute in parallel.

CPU, indeed, all hardware devices, are normally designed using an HDL like Verilog to speed development and ensure high quality by using peer review before an IC is manufactured. It is possible to find Open Source Verilog scripts for many devices, including CPU cores[1], so designers can begin with mature, working code and then "tweak" it as necessary to match their particular project.

14.2.1.3 *CPU Design Principles*

CPU design commonly follows these steps:

ANALYSIS OF INTENDED USE Designing a CPU starts with an analysis of its intended use since that will determine the type of CPU that must be built. A simple four-bit CPU — that is, all instructions are only four-bits wide — is more than adequate for a simple device like a microwave oven, but for a cell phone or other more complex device, a 16-bit, or larger, CPU is needed.

THE INSTRUCTION SET After the purpose of the CPU is defined then an instruction set is created. CPU instructions control the physical flow of bits through the CPU and various components of a computer. While an instruction set looks something like a programming language, it is important to keep in mind that an instruction set is very different from the more familiar higher-level programming languages like *Java* and *C++*.

CPU instructions are 16-bit (or larger) words that are conceptually divided into two sections: the operational code (*opcode*) and data. There are several classes of instructions, but three are the most common:

---

[1] http://www.opencores.org

- R (Register). These instructions involve some sort of register activity. The quintessential R-Type instruction is ADD, where the contents of two registers are added together and the sum is placed in another register.
- I (Immediate). These instructions include data as part of the instruction word and something happens immediately with that data. As an example, the LDI instruction immediately loads a number contained in the instruction word into a specified register.
- J (Jump). These instructions cause the program flow to jump to a different location. In older procedural programming languages (like *C* and *Basic*) these were often called GOTO statements.

ASSEMBLY LANGUAGE A computer program is nothing more than a series of ones and zeros organized into words the bit-width of the instruction set, commonly 32-bit or 64-bit. Each word is a single instruction and a series of instructions forms a program in *machine code* that looks something like this:

$$\begin{gathered} 0000000000000000 \\ 1001000100001010 \\ 1001001000001001 \\ 0111001100011000 \\ 0110000000100011 \end{gathered} \tag{14.1}$$

A CPU fetches and executes one 16-bit word of the machine code at a time. If a programmer could write machine code directly then the CPU could execute it without needing to compile it first. Of course, as it is easy to imagine, no one actually writes machine code due to its complexity.

The next level higher than machine code is called *Assembly*, which uses easy-to-remember abbreviations (called "mnemonics") to represent the available instructions. Following is the assembly language program for the machine code listed above:

Label	Mnemonic	Operands	Comment
START	NOP	0 0 0	No Operation
	LDI	1 0 a	R1 ¡- 0ah
	LDI	2 0 9	R2 ¡- 09h
	SHL	3 1 4	R3 ¡- R1 ¡¡ 8
	XOR	0 2 3	Acc ¡- R2 XOR R3

Each line of assembly has four parts. First is an optional *Label* that can be used to indicate various sections of the program and facilitates "jumps" around the program; second is the mnemonic for the code being executed; third are one or more operands; and fourth is an optional comment field.

Machine code is CPU specific so code written for one type of computer could not be used on any other type of computer.

Once the program has been written in Assembly, it must be "assembled" into machine code before it can be executed. An assembler is a fairly simply program that does little more than convert a file containing assembly code into instructions that can be executed by the CPU. The assembly program presented above would be assembled into Machine Code 14.1.

PROGRAMMING LANGUAGES Many high level programming languages have been developed, for example *Java* and *C++*. These languages tend to be easy to learn and can enable a programmer to quickly create very complex programs without digging into the complexity of machine code.

Programs written in any of these high-level languages must be either interpreted or compiled before they can be executed. Interpreters are only available for "scripting" languages like *PERL* and *Python* and they execute the source code one line at a time. In general, interpreters cannot optimize the code so they are not efficient; but they enable a programmer to quickly "try out" some bit of code without having to compile it. A compiler, on the other hand, converts the entire program to machine code (normally referred to as "object" code by a compiler) and then creates an executable file. A compiler also optimizes the code so it executes as efficiently as possible.

In the end, there are dozens of different programming languages, but they all eventually reduce programming instructions to a series of ones and zeros which the CPU can execute.

14.2.1.4 *State Machine*

Because a CPU is little more than a complex FSM, the next step in the design process is to define the various states and the operations that take place within each state. In general, an operating CPU cycles endlessly through three primary states:

- FETCH an instruction from memory. Instructions take the form of 16-bit numbers similar in appearance to 1001000100001010.
- DECODE the instruction. The instruction fetched from memory must be decoded into something like "Add the contents of Register 2 to Register 3 and save the results in Register 1."
- EXECUTE the instruction.

In general, the way that a CPU functions is to fetch a single instruction from RAM and then decode and execute that instruction. As an example, consider the Assembly example introduced above:

```
Label Mnemonic Operands    Comment
START NOP      0 0 0       No Operation
      LDI      1 0 a       R1  <- 0ah
      LDI      2 0 9       R2  <- 09h
      SHL      3 1 4       R3  <- R1 << 8
      XOR      0 2 3       Acc <- R2 XOR R3
```

Each line is an instruction and the CPU would fetch and execute each instruction from RAM in order. The purpose of this short code snip is to load a 16-bit register with a number when only 8 bits are available in the opcode. The eight high order bits are loaded into Register one and the eight low order bits are loaded into Register two. The high order bits are shifted to the left eight places and then the two registers are XOR'd together:

1. NOP: This is a "no operation" instruction so the CPU does nothing.
2. LDI: The number $0A_{16}$ is loaded into register one. This is the value of the high-order bits desired in the 16-bit number.
3. LDI: The number 09_{16} is loaded into register two. This is the value of the low-order bits desired in the 16-bit number.
4. SHL: Register three is loaded with the value of register one shifted left eight places.
5. XOR: The Accumulator is loaded with the value of register two XOR'd with 4egister three. This leaves the accumulator with a 16-bit number, $0A09_{16}$ that was loaded eight bits at a time.

14.2.2 *CPU States*

The first task that a CPU must accomplish is to fetch and decode an instruction held in memory. Figure 14.3 is a simplified state diagram that shows the first two CPU states as *Fetch* and *Decode*. After that, all of the different instructions would create their own state (for simplicity, only three instructions are shown in the Figure 14.3).

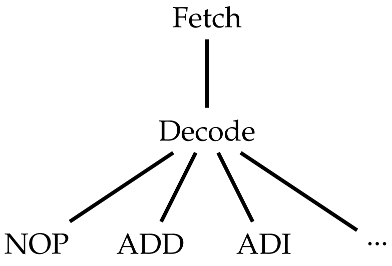


Figure 14.3: CPU State Diagram

The CPU designer would continue to design states for all instructions and end each state with a loop back to Fetch in order to get the next instruction out of RAM.

Designing a CPU is no insignificant task and is well beyond the scope of this book. However, one of the labs in the accompanying lab manual design a very simple processor that demonstrates how bits are moved around a circuit based upon control codes.

Part III

APPENDIX

a

BOOLEAN PROPERTIES AND FUNCTIONS

AND Truth Table

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

OR Truth Table

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

XOR Truth Table

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

NAND Truth Table

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

NOR Truth Table

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

XNOR Truth Table

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

NOT Truth Table

Input	Output
0	1
1	0

Buffer Truth Table

Input	Output
0	0
1	1

Table a.1: Univariate Properties

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

Table a.2: Multivariate Properties

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

Table a.3: Boolean Functions

A	0	0	1	1	
B	0	1	0	1	
F_0	0	0	0	0	Zero or Clear. Always zero (Annihilation)
F_1	0	0	0	1	Logical AND: $A * B$
F_2	0	0	1	0	Inhibition: AB' or $A > B$
F_3	0	0	1	1	Transfer A to Output, Ignore B
F_4	0	1	0	0	Inhibition: $A'B$ or $B > A$
F_5	0	1	0	1	Transfer B to Output, Ignore A
F_6	0	1	1	0	Exclusive Or (XOR): $A \oplus B$
F_7	0	1	1	1	Logical OR: $A + B$
F_8	1	0	0	0	Logical NOR: $(A + B)'$
F_9	1	0	0	1	Equivalence: $(A = B)'$
F_{10}	1	0	1	0	Not B and ignore A, B Complement
F_{11}	1	0	1	1	Implication, $A + B'$, $B >= A$
F_{12}	1	1	0	0	Not A and ignore B, A Complement
F_{13}	1	1	0	1	Implication, $A' + B$, $A >= B$
F_{14}	1	1	1	0	Logical NAND: $(A * B)'$
F_{15}	1	1	1	1	One or Set. Always one (Identity)

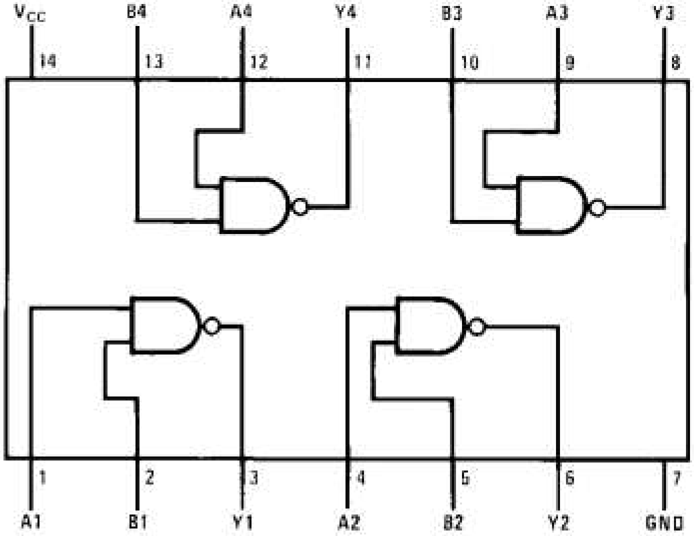
b

IC REFERENCE

The following information is provided as help for the TTL IC devices available within *Logisim-evolution* .

B.1 7400: QUAD 2-INPUT NAND GATE

7400 IC is a Quad 2-Input NAND Gate that contains four independent gates each of which performs the logic NAND function.


V_{CC}
B4
A4
Y4
B3
A3
Y3
14
13
12
11
10
9
8
1
2
3
4
5
6
7
A1
B1
Y1
A2
B2
Y2
GND

B.2 7402: QUAD 2-INPUT NOR GATE

B.3 7404: HEX INVERTER

B.4 7408: QUAD 2-INPUT AND GATE

B.5 7410: TRIPLE 3-INPUT NAND GATE

B.6 7411: TRIPLE 3-INPUT AND GATE

B.7 7413: DUAL 4-INPUT NAND GATE (SCHMITT-TRIGGER)

B.8 7414: HEX INVERTER (SCHMITT-TRIGGER)

B.9 7418: DUAL 4-INPUT NAND GATE (SCHMITT-TRIGGER INPUTS)

B.10 7419: HEX INVERTER (SCHMITT-TRIGGER INPUTS)

B.11 7420: DUAL 4-INPUT NAND GATE

B.12 7421: DUAL 4-INPUT AND GATE

B.13 7424: QUAD 2-INPUT NAND GATE (SCHMITT-TRIGGER)

B.14 7427: TRIPLE 3-INPUT NOR GATE

B.15 7430: SINGLE 8-INPUT NAND GATE

B.16 7432: QUAD 2-INPUT OR GATE

B.17 7436: QUAD 2-INPUT NOR GATE

B.18 7442: BCD TO DECIMAL DECODER

B.19 7443: EXCESS-3 TO DECIMAL DECODER

B.20 7444: GRAY TO DECIMAL DECODER

B.21 7447: BCD TO 7-SEGMENT DECODER

B.22 7451: DUAL AND-OR-INVERT GATE

B.23 7454: FOUR-WIDE AND-OR-INVERT GATE

B.24 7458: DUAL AND-OR GATE

B.25 7464: 4-2-3-2 AND-OR-INVERT GATE

B.26 7474: DUAL D-FLIPFLOPS WITH PRESET AND CLEAR

B.27 7485: 4-BIT MAGNITUDE COMPARATOR

GLOSSARY

BCD	Binary Coded Decimal. 4
CPU	Central Processing Unit. 4
IC	Integrated Circuit. 7

COLOPHON

The logic diagrams in this book are screen captures of *Logisim-evolution* circuits.

This book was typeset using the typographical look-and-feel classicthesis developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". classicthesis is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of classicthesis usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of September 16, 2019 (Edition 6.0).

Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL*) are used. The "typewriter" text is typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)