CLASSICAL AND QUANTUM COMPUTING

by

YORICK HARDY

DISSERTATION

submitted in fulfilment
of the requirements for the degree

MAGISTER SCIENTIAE

in

APPLIED MATHEMATICS

in the

FACULTY OF SCIENCE

at the

RAND AFRIKAANS UNIVERSITY

SUPERVISOR: PROF W.-H. STEEB

AUGUST 2000
Abstract

Part I.

Part I presents important techniques and results in classical computing. Algorithms and informal verification techniques are described. The definition of a Boolean algebra is given, and various properties of the algebra are introduced. Methods are discussed to obtain efficient implementations of Boolean functions. The powerful technique of problem solving using recursion is illustrated extensively in a number of problems. The implementation of recursion is also discussed. The description of recursive functions is also given, which is important for discussions on computability. I consider Hamming codes to correct single bit errors, and the technique of weighted checksums. Besides these techniques, the noiseless coding theorem (which gives bounds on how reliably information can be transmitted with limited resources) is also discussed in detail. Methods to encrypt information using a private key are considered. The public key cryptography technique, which uses a computationally difficult system to provide security and only public keys are exchanged, is described. Computing models in the form of finite state machines are introduced. A finite automaton is described, and improvements are considered leading to the Turing machine. Classical complexity is described in terms of the repetitiveness of bit strings. I also describe briefly Gödel’s incompleteness theorem and the halting problem for Turing machines. Finally numerical methods for problem solving are considered in the form of neural network computing and genetic algorithms.

Part II.

Part II considers the consequences of quantum mechanical implementations of computing devices. To this end all the relevant theory of quantum mechanics has been included. The basic unit of storage, how a quantum computation evolves, what operations can be performed and some important operations are discussed. Extensions to classical finite state machines are considered which can model quantum computations. The possibilities of quantum computing is revealed in an extensive description of quantum teleportation. I then proceed to cover six algorithms which display significant advantages over current classical algorithms. The problems include Deutsch’s problem, which cannot be solved classically, secure key distribution for cryptography, Shor’s factoring algorithms and Grover’s database searching. The Von Neumann entropy is introduced, and measurement of entanglement is considered. Finally bounds on communication of quantum states with limited resources are considered in the form of the quantum noiseless coding theorem and the Holevo bound. I show how to avoid errors in quantum states due to interaction with the environment. Some of the techniques from classical error correction apply to quantum error correction, but new theory is described to tolerate new errors possible in the quantum computation model. Some approaches to the physical implementation of a quantum computing device are presented.
Samevatting

Deel I.

Deel I bespreek die belangrike tegnieke en resultate vir klassieke berekenings. Algoritmes en informele tegnieke vir die bewys van korrektheid word beskryf. Die Boole-algebra word gedefinieer, en verskeie eieskappe van die algebra word ingevoer. Metodes vir die doeltreffende implementasie van Boole-funksies word ook beskryf. Die kragtige tegniek van rekursie word uitvoerig deur 'n aantal probleme geïllustreer. Die implementering van rekursie word ook bespreek. Die klas van rekursiewe funkies, wat belangrik is vir die bespreking van berekenbaarheid, word gegee. Om 'n fout in 'n enkele bis te korrigeer word die Hamming-kode beskryf. Die tegniek van die geweegde kontrolesom word ook bespreek. Die beperkings op die hoeveelheid inligting wat gestuur kan word met beperkte hulpbronne word deeglik behandel in die form van die geruislose kodering stelling. Metodes vir die enkripsie van inligting deur middel van 'n private sleutel word beskryf. Die openbare sleutel kriptografie tegniek, waar sekuriteit deur ingewikkelde berekeninge verskaf word en net openbare sleutels verraai word, word ook behandel. Modelle van rekenaars word in die vorm van eindige toestand masjiene beskryf. 'n Eindige otomaat word gedefinieer, en verbeterings lei tot die Turing-masjien. Die klassieke kompleksiteit word in terme van die herhaling van patrones in reekse van bisse beskryf. Gödel se onvolledigheidstelling en die half probleem van Turing-masjiene word ook kortliks beskryf. Die laaste klassieke metodes wat bespreek word is numeriese metodes, in die vorm van genetiese algoritmes en neurale netwerke.

Deel II.

Deel II handel oor die gevolge van kwantum meganiese implementasies van rekenaars. Dus is dit nodig om die teorie van kwantum meganika in te voer. Die kleinste eenheid van inligting, die manier waarop 'n kwantum berekening ontwikkel en die bewerkings wat beskikbaar is vir kwantum berekeninge word vir kwantum rekenaar stelsels beskryf. Klassieke eindige toestand masjiene word uitgebrei om as modelle vir kwantum rekenaars te dien. Die moontlikhede vir kwantum rekenaars word deur 'n deeglike beskrywing van kwantum teleportasie geïllustreer. Daarna word ses kwantum algoritmes beskryf wat aansienlike voordele het oor die huidige klassieke algoritmes. Die probleme wat bespreek word sluit in Deutsch se probleem (wat onmoontlik klassiek oplosbaar is), kwantum sleutel verspreiding vir kriptografie, die algoritme vir faktoriserings vir Shor en Grover se algoritme vir databasis navraag. Die Von Neumann entropie en die meting van verstrikking word beskryf. Die beperkings op kommunikasie van kwantum informasie met beperkte hulpbronne word gegee in die vorm van die kwantum geruislose kodering stelling en die Holevo beperking. Die wisselwerking van 'n kwantum stelsel en die omgewing maak dit noodwendig om fout korrigering metodes te vind. Van die klassieke fout korrigering tegnieke is van toepassing op kwantum stelsels. Nuwe teorie word egter beskryf wat foute hanteer wat spesifie in kwantumstelsels moontlik is. 'n Paar metodes vir die fisiese implementering van kwantum rekenaar stelsels word gegee.
Acknowledgements

I would like to thank Prof W.-H. Steeb for introducing me to the field of quantum computing, providing the opportunity to work on such an interesting project and for his patience. I would also like to thank L. Hoffman who proof read the manuscript, resulting in its present quality.
# Contents

I Classical Computing

1 Algorithms 3
  1.1 Algorithms .......................................................... 3
  1.2 Algorithm Verification ........................................... 6
  1.3 Random Algorithms ............................................... 10
  1.4 Total and Partial Functions ..................................... 15
  1.5 Alphabets and Words ............................................. 18

2 Boolean Algebra 23
  2.1 Introduction .......................................................... 23
  2.2 Definitions ........................................................... 24
  2.3 Rules and Laws of Boolean Algebra ............................... 26
  2.4 DeMorgan’s Theorem ................................................ 27
  2.5 Further Definitions ................................................ 27
  2.6 Boolean Function Implementation .................................. 33
    2.6.1 Karnaugh Maps .................................................. 36
    2.6.2 Quine-McCluskey Method ..................................... 39
  2.7 Example Programs .................................................. 42
    2.7.1 Efficient Set Operations Using Boolean Algebra .............. 42
    2.7.2 Quine-McCluskey Implementation .............................. 47

3 Number Representation 53
  3.1 Binary, Decimal and Hexadecimal Numbers ...................... 53
    3.1.1 Conversion ...................................................... 55
    3.1.2 Arithmetic ...................................................... 60
    3.1.3 Signed Integers ............................................... 62
    3.1.4 Overflow ....................................................... 69
    3.1.5 Binary-Coded Decimal Form ................................. 72
  3.2 Floating Point Representation ................................... 74
    3.2.1 Introduction ................................................... 74
    3.2.2 Representation ............................................... 76

4 Logic Gates 81
  4.1 Introduction ....................................................... 81
  4.2 Gates ................................................................. 82
4.2.1 AND Gate ........................................ 82
4.2.2 OR Gate ........................................ 83
4.2.3 XOR Gate ........................................ 84
4.2.4 NOT Gate (Inverter) ......................... 85
4.2.5 NAND Gate ..................................... 86
4.2.6 NOR Gate ...................................... 87
4.2.7 XNOR Gate .................................... 88
4.3 Buffer ............................................. 89
4.4 Tri-State Logic ................................... 90
4.5 Feedback and Gates ......................... 91

5 Combinational Circuits .................. 93
5.1 Introduction ................................... 93
5.2 Decoder .......................................... 94
5.3 Encoder ......................................... 95
5.4 Demultiplexer .................................. 98
5.5 Multiplexer ...................................... 99
5.6 Binary Adder .................................. 100
  5.6.1 Binary Half Adder ......................... 100
  5.6.2 Binary Full Adder ......................... 101
  5.6.3 Binary Four-Bit Adder ................. 102
  5.6.4 Faster Addition ......................... 103
5.7 Binary Subtraction ....................... 104
5.8 Binary Multiplication .................... 105
  5.8.1 Unsigned Integer Multiplication ... 105
  5.8.2 Fast Multiplication ..................... 107
  5.8.3 Signed Integer Multiplication ....... 108
5.9 Binary Division .............................. 109
5.10 Magnitude Comparator .................. 110
5.11 4-Bit ALU ..................................... 112
5.12 Read Only Memory (ROM) ............. 114
5.13 Combinational Programmable Logic Devices 115
5.14 Programmable Gate Arrays ............ 119
5.15 VHDL ......................................... 120

6 Latches and Registers .................. 121
6.1 Introduction ................................. 121
6.2 SR Latch ...................................... 122
6.3 D Latch ........................................ 123
6.4 JK Latch ...................................... 124
6.5 D Register .................................... 125
6.6 JK Register ................................... 126
<table>
<thead>
<tr>
<th>Section</th>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Computability and Complexity</td>
<td>255</td>
</tr>
<tr>
<td>13.1</td>
<td>Introduction</td>
<td>255</td>
</tr>
<tr>
<td>13.2</td>
<td>Computability</td>
<td>256</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Church’s Thesis</td>
<td>256</td>
</tr>
<tr>
<td>13.2.2</td>
<td>The Halting Problem</td>
<td>257</td>
</tr>
<tr>
<td>13.3</td>
<td>Gödel’s Incompleteness Theorem</td>
<td>258</td>
</tr>
<tr>
<td>13.3.1</td>
<td>Gödel Numbering</td>
<td>258</td>
</tr>
<tr>
<td>13.3.2</td>
<td>Gödel’s Incompleteness Theorem</td>
<td>260</td>
</tr>
<tr>
<td>13.4</td>
<td>Complexity</td>
<td>260</td>
</tr>
<tr>
<td>13.4.1</td>
<td>Complexity of Bit Strings</td>
<td>260</td>
</tr>
<tr>
<td>13.4.2</td>
<td>NP-class of Problems</td>
<td>263</td>
</tr>
<tr>
<td>14</td>
<td>Neural Networks</td>
<td>265</td>
</tr>
<tr>
<td>14.1</td>
<td>Introduction</td>
<td>265</td>
</tr>
<tr>
<td>14.2</td>
<td>Hyperplanes</td>
<td>270</td>
</tr>
<tr>
<td>14.3</td>
<td>Perceptron</td>
<td>272</td>
</tr>
<tr>
<td>14.3.1</td>
<td>Introduction</td>
<td>272</td>
</tr>
<tr>
<td>14.3.2</td>
<td>Boolean Functions</td>
<td>276</td>
</tr>
<tr>
<td>14.3.3</td>
<td>Perceptron Learning</td>
<td>279</td>
</tr>
<tr>
<td>14.3.4</td>
<td>Quadratic Threshold Gates</td>
<td>283</td>
</tr>
<tr>
<td>14.3.5</td>
<td>One and Two Layered Networks</td>
<td>286</td>
</tr>
<tr>
<td>14.3.6</td>
<td>Perceptron Learning Algorithm</td>
<td>287</td>
</tr>
<tr>
<td>14.3.7</td>
<td>The XOR Problem and Two-Layered Networks</td>
<td>293</td>
</tr>
<tr>
<td>14.4</td>
<td>Multilayer Perceptrons</td>
<td>298</td>
</tr>
<tr>
<td>14.4.1</td>
<td>Introduction</td>
<td>298</td>
</tr>
<tr>
<td>14.4.2</td>
<td>Cybenko’s Theorem</td>
<td>299</td>
</tr>
<tr>
<td>14.4.3</td>
<td>Back-Propagation Algorithm</td>
<td>300</td>
</tr>
<tr>
<td>15</td>
<td>Genetic Algorithms</td>
<td>317</td>
</tr>
<tr>
<td>15.1</td>
<td>Introduction</td>
<td>317</td>
</tr>
<tr>
<td>15.2</td>
<td>The Sequential Genetic Algorithm</td>
<td>319</td>
</tr>
<tr>
<td>15.3</td>
<td>Gray Code</td>
<td>324</td>
</tr>
<tr>
<td>15.4</td>
<td>Schemata Theorem</td>
<td>327</td>
</tr>
<tr>
<td>15.5</td>
<td>Markov Chain Analysis</td>
<td>330</td>
</tr>
<tr>
<td>15.6</td>
<td>Bit Set Classes in C++ and Java</td>
<td>332</td>
</tr>
<tr>
<td>15.7</td>
<td>A Bit Vector Class</td>
<td>337</td>
</tr>
<tr>
<td>15.8</td>
<td>Maximum of One-Dimensional Maps</td>
<td>341</td>
</tr>
<tr>
<td>15.9</td>
<td>Maximum of Two-Dimensional Maps</td>
<td>350</td>
</tr>
<tr>
<td>15.10</td>
<td>The Four Colour Problem</td>
<td>361</td>
</tr>
<tr>
<td>15.11</td>
<td>Problems with Constraints</td>
<td>365</td>
</tr>
<tr>
<td>15.11.1</td>
<td>Introduction</td>
<td>365</td>
</tr>
<tr>
<td>15.11.2</td>
<td>Knapsack Problem</td>
<td>367</td>
</tr>
<tr>
<td>15.11.3</td>
<td>Traveling Salesman Problem</td>
<td>373</td>
</tr>
<tr>
<td>15.12</td>
<td>Other Applications for Genetic Algorithms</td>
<td>385</td>
</tr>
<tr>
<td>15.13</td>
<td>Distributed Global Optimization</td>
<td>386</td>
</tr>
</tbody>
</table>
II Quantum Computing

16 Quantum Mechanics
  16.1 Hilbert Spaces .......................... 399
  16.2 Schmidt Decomposition .................. 415
  16.3 Linear Operators in Hilbert Spaces ....... 418
  16.4 Spin Matrices and Kronecker Product .... 433
  16.5 Postulates of Quantum Mechanics ......... 441

17 Quantum Bits and Quantum Computation .... 449
  17.1 Introduction ............................ 449
  17.2 Quantum Bits and Quantum Registers .... 450
    17.2.1 Quantum Bits ....................... 450
    17.2.2 Quantum Registers .................. 451
  17.3 Entangled States ......................... 453
  17.4 Quantum gates ........................... 460
    17.4.1 Introduction ......................... 460
    17.4.2 NOT Gate ............................ 461
    17.4.3 Walsh-Hadamard Gate ................. 462
    17.4.4 XOR and the Controlled NOT Gate .... 464
    17.4.5 Other Quantum Gates ................ 465
    17.4.6 Universal Sets of Quantum Gates .... 468
    17.4.7 Functions ............................ 469
  17.5 Garbage Disposal ......................... 472
  17.6 Quantum Copying ......................... 473
  17.7 Example Programs ......................... 476

18 Measurement and Quantum States .......... 487
  18.1 Introduction ............................ 487
  18.2 Measurement Problem ..................... 488
  18.3 Copenhagen Interpretation ............... 489
  18.4 Hidden Variable Theories ................ 491
  18.5 Everett Interpretation ................... 492
  18.6 Basis Degeneracy Problem ................. 494
  18.7 Information Theoretic Viewpoint ......... 496

19 Quantum State Machines .................. 497
  19.1 Introduction ............................ 497
  19.2 Quantum Automata ......................... 497
  19.3 Quantum Turing Machines .................. 500
20 Teleportation 503
20.1 Introduction .................................................. 503
20.2 Teleportation Algorithm .................................... 504
20.3 Example Program ............................................. 507

21 Quantum Algorithms 511
21.1 Deutsch’s Problem ............................................ 511
21.2 Quantum Key Distribution ................................. 513
21.3 Dense Coding .................................................. 515
21.4 Quantum Fourier Transform ............................... 517
21.5 Factoring (Shor’s Algorithm) .............................. 518
21.6 Unstructured Search (Grover’s Algorithm) ............ 522

22 Quantum Information Theory 525
22.1 Introduction .................................................... 525
22.2 Von Neumann Entropy ........................................ 526
22.3 Measures of Entanglement .................................. 527
22.3.1 Bell’s Inequality .......................................... 527
22.3.2 Entanglement of Formation ............................ 528
22.3.3 Conditions on Entanglement Measures ............... 529
22.4 Quantum Coding .............................................. 531
22.5 Holevo Bound .................................................. 537

23 Quantum Error Detection and Correction 539
23.1 Introduction .................................................... 539
23.2 The Nine-qubit Code ......................................... 540
23.3 The Seven-qubit Code ........................................ 542
23.4 Efficiency and the Five-qubit Code ....................... 543
23.5 Stabilizer Codes ................................................. 545

24 Quantum Hardware 547
24.1 Introduction .................................................... 547
24.2 Trapped Ions .................................................... 548
24.3 Cavity Quantum Electrodynamics ......................... 549
24.4 Quantum Dots .................................................. 550
24.5 Nuclear Magnetic Resonance Spectroscopy .............. 553

25 Internet Resources 555

Bibliography 557
## List of Tables

2.1 AND, OR and Complement .............................................. 27  
2.2 Parity Function ....................................................... 29  
2.3 XOR Truth Table ....................................................... 29  
2.4 Full Adder ............................................................... 35  
2.5 4-bit Decimal Incremener ............................................. 38  
2.6 Two’s Complement Operation on 2 Bits ............................... 40  

4.1 Function Table and Truth Tables for a Logic Circuit ................ 81  
4.2 Truth Table for the AND Gate ......................................... 82  
4.3 Truth Table for the OR Gate ........................................... 83  
4.4 Truth Table for the XOR Gate ......................................... 84  
4.5 Truth Table for the NOT Gate .......................................... 85  
4.6 Truth Table for the NAND Gate ...................................... 86  
4.7 Truth Table for the NOR Gate ........................................ 87  
4.8 Truth Table for the XNOR Gate ...................................... 88  
4.9 Truth Table for the Buffer ............................................ 89  

5.1 Truth Table for the CMOS 4532 ....................................... 97  
5.2 Truth Table for CMOS 4555 .......................................... 98  
5.3 Half Adder Truth Table ............................................... 100  
5.4 Full Adder Truth Table ............................................... 101  
5.5 Truth Table for the CMOS 4585 ...................................... 111  
5.6 Function Table for CMOS 74LV688 .................................. 111  

6.1 Characteristic Table for the $SR$ Latch ............................... 122  
6.2 Characteristic Table for the $D$ Latch ............................... 123  
6.3 Characteristic Table for the $JK$ Latch .............................. 124  

7.1 Counting Sequence .................................................... 131  

8.1 Coefficients for Three Wavelet Functions ............................... 160  

12.1 Parity Check Finite Automaton - Transitions ........................ 235  
12.2 Hamming Code Finite Automaton - Transitions .................... 236  
12.3 Moore Machine for the NOT Operation - Transitions ............. 238  
12.4 $n$-bit Incremener Moore Machine - Transitions ................ 239  
12.5 Mealy Machine for the NOT Operation - Transitions ............. 240
12.6  n-bit Incrementer Mealy Machine - Transitions . . . . . . . . . . . . . . 241
12.7 Parity Check Turing Machine - Transitions . . . . . . . . . . . . . . . 243
12.8 Parity Calculation Turing Machine Transitions . . . . . . . . . . . . 244
12.9 Turing Machine for the NOT Operation - Transitions . . . . . . . . 245
12.10 Bit Reversal Turing Machine Transitions . . . . . . . . . . . . . . . 246

14.1 Function Table for the Boolean Function \((\overline{x_1} \cdot x_2) + (x_2 \cdot \overline{x_3})\) . . . . . . 275
14.2 Training Set for Parity Function . . . . . . . . . . . . . . . . . . . . . . 305

15.1 3 Bit Binary Gray Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 324

21.1 Dense Coding: Alice’s Transformations . . . . . . . . . . . . . . . . . . . 516
21.2 Dense Coding: Bob’s Transformations . . . . . . . . . . . . . . . . . . . . 516

23.1 Error Syndrome for the 5 Qubit Error Correction Code . . . . . . . 545
# List of Figures

4.1 Symbol for 2-input AND Gate .......................... 82  
4.2 Symbol for 2-input OR Gate .......................... 83  
4.3 Symbol for 2-input XOR Gate .......................... 84  
4.4 Symbol for the NOT Gate .......................... 85  
4.5 Symbol for 2-input NAND Gate .......................... 86  
4.6 XOR Implemented With NAND Gates .......................... 86  
4.7 Symbol for 2-input NOR Gate .......................... 87  
4.8 XOR Implemented With NOR Gates .......................... 87  
4.9 Symbol for 2-input XNOR Gate .......................... 88  
4.10 Symbol for the Buffer .......................... 89  
4.11 (a) A tri-state inverter with an enable line, (b) a tri-state buffer with a disable line .......................... 90  
4.12 NAND Gate With Feedback .......................... 91  

5.1 Truth Table and Circuit of a 1-out-of-4 Decoder .......................... 94  
5.2 Typical IC Encoder .......................... 95  
5.3 Circuit for the CMOS 4532 .......................... 96  
5.4 Demultiplexer Circuit .......................... 98  
5.5 Multiplexer Circuit .......................... 99  
5.6 Half Adder Circuit .......................... 100  
5.7 Full Adder Circuit .......................... 101  
5.8 Two Full Adders in Parallel .......................... 102  
5.9 Four Bit Adder Consisting of Four Adders .......................... 102  
5.10 Circuit for the Carry Bit of a 3-bit Adder .......................... 103  
5.11 Binary Subtraction Using the Two’s Complement .......................... 104  
5.12 Unsigned 4-bit Multiplication .......................... 106  
5.13 2-bit Fast Unsigned Multiplication .......................... 107  
5.14 Logic Diagram for the CMOS 74LV688 .......................... 111  
5.15 Logic Diagram for a ROM .......................... 114  
5.16 Input Representation for Programmable Gates .......................... 115  
5.17 PROM Device Architecture .......................... 116  
5.18 PROM Implementation of XOR .......................... 116  
5.19 PAL Device Architecture .......................... 117  
5.20 PAL Implementation of XOR .......................... 117  
5.21 PLA Device Architecture .......................... 118  
5.22 PLA Implementation of XOR .......................... 118  

xiii
15.1 A Map for the Four Colour Problem .......................... 361
17.1 NOT Gate .................................................. 461
17.2 Walsh-Hadamard Gate ...................................... 463
17.3 XOR Gate ....................................................... 464
17.4 Phase Shift Gate ............................................. 465
17.5 Tofolli Gate ................................................... 466
17.6 Quantum Circuit to Generate Bell States ................. 479
17.7 Quantum Circuit to Swap a Pair of Bits ..................... 479
20.1 Teleportation ..................................................... 503
20.2 Experimental Realization of Teleportation .................... 505
20.3 Quantum Circuit for Teleportation ......................... 506
21.1 Quantum Key Distribution ................................... 513
23.1 Encoding for the 5-qubit Error Correction Code ........... 544
24.1 Two Possible Configurations for Quantum Dot Cells ...... 550
## List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>empty set</td>
</tr>
<tr>
<td>$\mathbb{N}$</td>
<td>natural numbers</td>
</tr>
<tr>
<td>$\mathbb{N}_0$</td>
<td>$\mathbb{N} \cup {0}$</td>
</tr>
<tr>
<td>$\mathbb{Z}$</td>
<td>integers</td>
</tr>
<tr>
<td>$\mathbb{Q}$</td>
<td>rational numbers</td>
</tr>
<tr>
<td>$\mathbb{R}$</td>
<td>real numbers</td>
</tr>
<tr>
<td>$\mathbb{R}^+$</td>
<td>nonnegative real numbers</td>
</tr>
<tr>
<td>$\mathbb{C}$</td>
<td>complex numbers</td>
</tr>
<tr>
<td>$\mathbb{R}^n$</td>
<td>$n$-dimensional Euclidean space</td>
</tr>
<tr>
<td>$\mathbb{C}^n$</td>
<td>$n$-dimensional complex linear space</td>
</tr>
<tr>
<td>$\mathcal{H}$</td>
<td>Hilbert space</td>
</tr>
<tr>
<td>$i$</td>
<td>$\sqrt{-1}$</td>
</tr>
<tr>
<td>$\Re z$</td>
<td>real part of the complex number $z$</td>
</tr>
<tr>
<td>$\Im z$</td>
<td>imaginary part of the complex number $z$</td>
</tr>
<tr>
<td>$A \subset B$</td>
<td>subset $A$ of set $B$</td>
</tr>
<tr>
<td>$A \cap B$</td>
<td>the intersection of the sets $A$ and $B$</td>
</tr>
<tr>
<td>$A \cup B$</td>
<td>the union of the sets $A$ and $B$</td>
</tr>
<tr>
<td>$f \circ g$</td>
<td>composition of two mappings $(f \circ g)(x) = f(g(x))$</td>
</tr>
<tr>
<td>$\psi,</td>
<td>\psi\rangle$</td>
</tr>
<tr>
<td>$t$</td>
<td>independent time variable</td>
</tr>
<tr>
<td>$x$</td>
<td>independent space variable</td>
</tr>
<tr>
<td>$x \in \mathbb{R}^n$</td>
<td>element $x$ of $\mathbb{R}^n$</td>
</tr>
<tr>
<td>$||_\cdot$</td>
<td>norm</td>
</tr>
<tr>
<td>$x \times y$</td>
<td>vector product</td>
</tr>
<tr>
<td>$\otimes$</td>
<td>Kronecker product, tensor product</td>
</tr>
<tr>
<td>$\wedge$</td>
<td>exterior product (Grassmann product, wedge product)</td>
</tr>
<tr>
<td>$\langle \cdot, \cdot \rangle, \langle \cdot</td>
<td>\cdot \rangle$</td>
</tr>
<tr>
<td>$\det$</td>
<td>determinant of a square matrix</td>
</tr>
<tr>
<td>$\tr$</td>
<td>trace of a square matrix</td>
</tr>
<tr>
<td>${, }$</td>
<td>Poisson product</td>
</tr>
<tr>
<td>$[,]$</td>
<td>commutator</td>
</tr>
<tr>
<td>$[,]^+$</td>
<td>anticommutator</td>
</tr>
<tr>
<td>$\delta_{jk}$</td>
<td>Kronecker delta</td>
</tr>
<tr>
<td>$\delta$</td>
<td>delta function</td>
</tr>
<tr>
<td>$\sgn(x)$</td>
<td>the sign of $x$, $1$ if $x &gt; 0$, $-1$ if $x &lt; 0$, $0$ if $x = 0$</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>eigenvalue</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>real parameter</td>
</tr>
</tbody>
</table>
\( I \)  
unit operator, unit matrix

\( U \)  
unitary operator, unitary matrix

\( \Pi \)  
projection operator, projection matrix

\( H \)  
Hamilton function

\( \hat{H} \)  
Hamilton operator

\( V \)  
potential

\( b_j, b_j^\dagger \)  
Bose operators

\( c_j, c_j^\dagger \)  
Fermi operators

\( p \)  
momentum

\( \hat{p} \)  
momentum operator

\( L \)  
angular momentum

\( \hat{L} \)  
angular momentum operator

\( |\beta\rangle \)  
Bose coherent state

\( D \)  
differential operator \( \partial/\partial x \)

\( \Omega_+ \)  
Møller operator

\( Y_{lm}(\theta, \phi) \)  
spherical harmonics

\( \text{AND} \)  
AND operation in Boolean algebra

\( \text{OR} \)  
OR operation in Boolean algebra

\( \oplus \)  
XOR operation in Boolean algebra

\( \overline{A} \)  
negation of \( A \) in Boolean algebra

\( [x] \)  
the greatest integer which is not greater than \( x \)
Part I

Classical Computing
Chapter 1

Algorithms

1.1 Algorithms

An algorithm [49, 67, 83, 103, 129] is a precise description of how to solve a problem. For example algorithms can be used to describe how to add and subtract numbers or to prove theorems. Usually algorithms are constructed with some basic accepted knowledge and inference rules or instructions. Thus programs in programming languages such as C++ and Java are algorithms. Thus an algorithm is a map \( f : E \rightarrow A \) of the input data \( E \) to the set of output data \( A \).

Knuth [111] describes an algorithm as a finite set of rules which gives a sequence of operations for solving a specific type of problem similar to a recipe or procedure. According to Knuth [111] an algorithm has the following properties

1. **Finiteness.** An algorithm must always terminate after a finite number of steps.

2. **Definiteness.** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.

3. **Input.** An algorithm has zero or more inputs, i.e., quantities which are given to it initially before the algorithm begins.

4. **Output.** An algorithm has one or more outputs, i.e., quantities which have a specified relation to the inputs.

5. **Effectiveness.** This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can, in principle, be done exactly and in a finite length of time by a man using pencil and paper.

Not every function can be realized by an algorithm. For example, the task of adding two arbitrary real numbers does not satisfy finiteness.
Example. The Euclidean algorithm is a method to find the greatest common divisor (GCD) of two integers. The GCD $d$ of two integers $a$ and $b$ is the integer that divides $a$ and $b$, and if $c < d$ is a divisor of $a$ and $b$ then $c$ divides $d$.

1. Let $a' := a$ and $b' := b$
2. Let $r$ and $q$ be integers such that $a' = qb' + r$ and $0 \leq r < b'$
3. If $r$ is not zero
   (a) Let $a' := b'$ and $b' := r$
   (b) Goto 2
4. The GCD is $b'$.

To illustrate this we find the GCD of 21 and 18:

- 1) $a' = 21$ and $b' = 18$
- 2) $q = 1$ and $r = 3$
- 3) $a' = 18$ and $b' = 3$
- 2) $q = 6$ and $r = 0$
- 4) The GCD is 3.

Now we find the GCD of 113 and 49:

- 1) $a' = 113$ and $b' = 49$
- 2) $q = 2$ and $r = 15$
- 3) $a' = 49$ and $b' = 15$
- 2) $q = 3$ and $r = 4$
- 3) $a' = 15$ and $b' = 4$
- 2) $q = 3$ and $r = 3$
- 3) $a' = 4$ and $b' = 3$
- 2) $q = 1$ and $r = 1$
- 3) $a' = 3$ and $b' = 1$
- 2) $q = 3$ and $r = 0$
- 4) The GCD is 1.
**Definition.** *Execution* of an algorithm refers to the following (or execution) of the steps given in the algorithm.

**Definition.** *Termination* of an algorithm is when an algorithm finishes, there is nothing more to be done.

An algorithm executes uniquely if, for a given input, the termination of the algorithm is always the same, i.e. the variables, memory, state, output and position of termination in the algorithm are always the same.

**Definition.** An algorithm is said to be *deterministic* if the algorithm execution is uniquely determined by the input.

**Example.** The Euclidean algorithm is deterministic, in other words for any given \( a \) and \( b \) the algorithm will always give the same result (the \( GCD \) of the given values).

**Definition.** An algorithm which is not deterministic is said to be *non-deterministic*.

**Example.** An algorithm which follows a certain path with probability is non-deterministic. For example a learning algorithm can use probabilities as follows:

- Suppose there are \( n \) options (paths of execution) available.
- The algorithm assigns probabilities \( p_1, p_2, \ldots, p_n \) according to merit for each of the options, where

\[
p_i \geq 0, \quad \sum_{i=1}^{n} p_i = 1.
\]

- The algorithm calculates the *Shannon entropy*

\[
S := - \sum_{i=1}^{n} p_i \log_2(p_i)
\]

which is a measure of the information available about the options.

- The algorithm then chooses an option (according to the given probabilities).
- The outcome of the event is used for learning where the learning is weighted using \( S \).
1.2 Algorithm Verification

To illustrate the method of program verification we first introduce mathematical induction.

If $P$ is a property with respect to natural numbers $N$ and

- $P$ holds for 1,
- if $P$ holds for $k$ then $P$ holds for $k + 1$,

then $P$ holds for all natural numbers.

So if $P$ holds for 1 it also holds for 2 and therefore also for 3 and so on.

Example.

- For $n = 1$

$$
\sum_{i=1}^{n} i = 1 = \frac{n(n + 1)}{2}
$$

- Suppose

$$
\sum_{i=1}^{k} i = \frac{k(k + 1)}{2}
$$

then

$$
\sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + k + 1 = \frac{k(k + 1)}{2} + k + 1 = \frac{(k+1)(k+2)}{2}
$$

Thus

$$
\sum_{i=1}^{n} i = \frac{n(n + 1)}{2}
$$

for $n$ a natural number.

\clubsuit
Example. Let \( x \geq 1 \) and \( n \in \mathbb{N} \).

- For \( n = 1 \)
  \[
  (1 + x)^n \geq 1 + nx
  \]
- Suppose
  \[
  (1 + x)^k \geq 1 + kx
  \]
  then
  \[
  (1 + x)^{k+1} = (1 + x)^k(1 + x) \geq (1 + kx)(1 + x)
  \]
  now
  \[
  (1 + kx)(1 + x) = 1 + (k + 1)x + x^2 \geq 1 + (k + 1)x
  \]
  Thus
  \[
  (1 + x)^n \geq 1 + nx
  \]
  for \( n \) a natural number.

This method allows us to verify that a property is true for all natural numbers by building on initial truths. The same method can be extended for algorithm verification. A program starts with some conditions known to be true and verification is the process of determining whether certain desired properties always hold during execution to give the desired result.

Definition. An assertion is a statement about a particular condition at a certain point in an algorithm.

Definition. A precondition is an assertion at the beginning of a sub-algorithm.

Definition. A postcondition is an assertion at the end of a sub-algorithm.

Definition. An invariant is a condition which is always true at a particular point in an algorithm.

To prove that an algorithm is correct it is necessary to prove that the postcondition for each sub-algorithm holds if the precondition for the same sub-algorithm holds. It is also necessary to prove that invariants hold at their positions in the algorithm. The preconditions, postconditions and invariants must of course reflect the purpose of the algorithm.
Example. We consider the algorithm to add \( n \) numbers \( x_1, x_2, \ldots, x_n \):

1. Let \( \text{sum} \) be a variable which takes numerical values initially set to 0.
2. Precondition: \( x_1, \ldots, x_n \) are the \( n \) numbers to be added
3. For each of the \( n \) numbers \( x_i \) \( (i = 1, 2, \ldots, n) \) do \( \text{sum} := \text{sum} + x_i \)
   Invariant:
   \[
   \text{sum} = \sum_{j=1}^{i} x_j
   \]
4. Postcondition:
   \[
   \text{sum} = \sum_{j=1}^{n} x_j
   \]
5. The desired result is given by \( \text{sum} \).

Example. The correctness of the *Euclidean algorithm* is based on a simple invariant.

1. Let \( a' := a \) and \( b' := b \)
2. Invariant: \( \gcd(a, b) = \gcd(a', b') \)
   Let \( r \) and \( q \) be integers such that \( a' = qb' + r \) and \( r < b' \)
3. If \( r \) is not zero
   (a) Let \( a' := b' \) and \( b' := r \)
   (b) Goto 2
4. The GCD is \( b' \).

To prove the invariant holds we need to prove that after step 2
\[
\gcd(a', b') = \gcd(b', r).
\]
Obviously \( \gcd(b', r) \) divides \( \gcd(a', b') \). The reverse argument is also easy. \( \gcd(a', b') \) divides both \( a' \) and \( b' \) and therefore divides \( r \). When \( r = 0 \) the GCD is \( b' \). ♦
In C and C++ the function `assert` in the header file `assert.h` is provided to help with debugging. The function takes one argument which must be an expression with numerical value. The function `assert` aborts the program and prints an error message if the expression evaluates to 0.

**Example.** We can use `assert` to make sure that, whenever a program calls the function `sum`, the function adds at least one number.

```cpp
// sum.cpp
#include <iostream>
#include <cassert>

using namespace std;

double sum(int n,double x[])
{
  // precondition: x is an array of n doubles
  assert(n > 0);

  int i;
  double sum = 0.0;
  for(i=0;i < n;i++) sum += x[i]; // invariant sum = x[0]+...+x[i]
  // postcondition: sum = x[0]+...+x[n-1]
  return sum;
}

void main(void)
{
  double x[5] = { 0.5,0.3,7.0,-0.3,0.5 };

  cout << "sum=" << sum(5,x) << endl;
  cout << "sum=" << sum(-1,x) << endl;
}

The output is:

sum=8
Assertion failed: n>0, file sum.cpp, line 9
abnormal program termination

♣
1.3 Random Algorithms

Random or stochastic algorithms use random numbers to try solve a problem. Generally the technique is used where approximations are acceptable and a completely accurate answer is difficult to obtain.

Examples include Monte Carlo methods, genetic algorithms, simulated annealing and neural networks. These algorithms are usually non-deterministic.

Random algorithms exist for numerical integration but other numerical methods are generally better for not too large dimensions.

In C++ the functions rand and

```cpp
void srand(unsigned)

in stdlib.h and

unsigned time(time_t *)

in time.h can be used to generate uniformly distributed random numbers. The function call

srand(time(NULL))

initializes the random number generator. The function rand() generates a random number between 0 and RAND_MAX. Note that the random number sequences generated in this way by the computer are not truly random and are eventually periodic. The number sequences have properties which make them appropriate approximations for random number sequences for use in algorithms. The statement double(rand())/RAND_MAX takes the integer returned by rand() and casts it to type double so that the division by RAND_MAX gives a random number of type double in the unit interval [0, 1].

Example. To calculate the value of $\pi$ we use the fact that the area of a quadrant of the unit circle

$$S := \{ (x, y) \mid x^2 + y^2 \leq 1 \}$$

is $\frac{\pi}{4}$. By generating random coordinates in the first quadrant of the unit circle the proportion of coordinates in the unit circle to the total number of coordinates generated approximates the area.

A few examples of the output are given below

```plaintext
pi=3.13994
pi=3.13806
pi=3.14156
pi=3.13744
```
// calcpi.cpp

#include <iostream>
#include <time.h>
#include <stdlib.h>

using namespace std;

void main(void)
{
    const int n = 500000;
    double x,y,pi;
    int i;

    // initialize the counter of the number of points
    // found in the unit circle to zero
    int in_count=0;

    // initialize the random number with a
    // seed value given by the current time
    srand(time(NULL));

    for(i=0;i<n;i++)
    {
        x = double(rand())/RAND_MAX;
        y = double(rand())/RAND_MAX;

        if(x*x+y*y<=1)
            in_count++;
    }

    pi = 4.0*double(in_count)/n;
    cout << "pi=" << pi << endl;
}
Example. Annealing [188] is the process of cooling a molten substance with the objective of condensing matter into a crystalline solid. Annealing can be regarded as an optimization process. The configuration of the system during annealing is defined by the set of atomic positions $r_i$. A configuration of the system is weighted by its Boltzmann probability factor,

$$e^{-E(r_i)/kT}$$

where $E(r_i)$ is the energy of the configuration, $k$ is the Boltzmann constant, and $T$ is the temperature. When a substance is subjected to annealing, it is maintained at each temperature for a time long enough to reach thermal equilibrium.

The iterative improvement technique for combinatorial optimization has been compared to rapid quenching of molten metals. During rapid quenching of a molten substance, energy is rapidly extracted from the system by contact with a massive cold substrate. Rapid cooling results in metastable system states; in metallurgy, a glassy substance rather than a crystalline solid is obtained as a result of rapid cooling. The analogy between iterative improvement and rapid cooling of metals stems from the fact that iterative improvement and rapid cooling of metals accepts only those system configurations which decrease the cost function. In an annealing (slow cooling) process, a new system configuration that does not improve the cost function is accepted based on the Boltzmann probability factor of the configuration. This criterion for accepting a new system state is called the Metropolis criterion. The process of allowing a fluid to attain thermal equilibrium at a temperature is also known as the Metropolis process.

The simulated annealing procedure is presented below. Simulated annealing essentially consists of repeating the Metropolis procedure for different temperatures. The temperature is gradually decreased at each iteration of the simulated annealing algorithm.

If the initial temperature is too low, the process gets quenched very soon and only a local optimum is found. If the initial temperature is too high, the process is very slow. Only a single solution is used for the search and this increases the chance of the solution becoming stuck at a local optimum. The changing of the temperature is based on an external procedure which is unrelated to the current quality of the solution, that is, the rate of change of temperature is independent of the solution quality. These problems can be rectified by using a population instead of a single solution. The annealing mechanism can also be coupled with the quality of the current solution by making the rate of change of temperature sensitive to the solution quality.

In the following program we apply simulated annealing to find the minimum of the function

$$f(x) = x^2 \exp(-x/15) \sin x.$$
// anneal.cpp
// simulated annealing
// x range: [0 : 100]

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <time.h>

using namespace std;

inline double f(double &x)
{
    return sin(x)*x*x*exp(-x/15.0);
}

inline int accept(double &Ecurrent, double &Enew, double &T, double &s)
{
    double dE = Enew - Ecurrent;
    double k = 1380662e-23;

    if(dE < 0.0)
        return 1;
    if(s < exp(-dE/(k*T)))
        return 1;
    else return 0;
}

int main()
{
    cout << "Finding the minimum via simulated annealing:" << endl;
    double xlow = 0.0;  double xhigh = 100.0;
    double Tmax = 500.0;  double Tmin = 1.0;
    double Tstep = 0.1;
    double T;

    srand(time(NULL));
    double s = rand()/double(RAND_MAX);

    double xcurrent = s*(xhigh - xlow);
    double Ecurrent = f(xcurrent);

    for(T=Tmax; T>Tmin; T-=Tstep)
    {
        s = rand()/double(RAND_MAX);
double xnew = s*(xhigh - xlow);
double Enew = f(xnew);
if(accept(Ecurrent,Enew,T,s))
{
xcurrent = xnew;
Ecurrent = Enew;
}
}

cout << "The minimum found is " << Ecurrent << " at x = "
<< xcurrent << endl;

return 0;
}

Typical outputs are given below.

Finding the minimum via simulated annealing:
The minimum found is -121.796 at x = 29.8397

Finding the minimum via simulated annealing:
The minimum found is -121.796 at x = 29.8397

Finding the minimum via simulated annealing:
The minimum found is -121.749 at x = 29.874

The global minimum of $f$ is found as one of the solutions to the transcendental equation

$$\tan(x^*) = \frac{15x^*}{x^* - 30}$$

in the interval $[0, 100]$ with $x \approx \frac{19\pi}{2}$. 
1.4 Total and Partial Functions

Functions and algorithms are closely related since algorithms are used to implement functions in computing and algorithms can be described in terms of functions.

**Definition.** A function \( f : A \to B \) is a total function if \( f \) associates every \( a \in A \) with exactly one image \( f(a) \) in \( B \).

**Definition.** A function \( f : A \to B \) is a partial function if it is total on some subset \( A' \) of \( A \). The set \( A' \) is then called the domain of \( f \) and is denoted \( \text{dom}(f) \). If \( a \in A \setminus A' \), \( f(a) \) is said to be undefined otherwise it is said to be defined.

**Definition.** Suppose \( f : A \to B \) and \( g : A \to B \). By definition, \( f \) and \( g \) are equal if and only if for each \( a \in A \), either

1. both \( f(a) \) and \( g(a) \) are defined, and \( f(a) = g(a) \); or
2. both \( f(a) \) and \( g(a) \) are undefined.

**Definition.** A function
\[
f : A_1 \times A_2 \times \ldots \times A_n \to B
\]
is said to be \( n \)-ary. Unary, binary and ternary are synonyms for 1-ary, 2-ary and 3-ary respectively. In the expression
\[
f(a_1, a_2, \ldots, a_n),
\]
we say that \( a_1, a_2, \ldots, a_n \) are the arguments of \( f \).

**Definition.** The range of a function \( f : A \to B \) is the set \{ \( f(a) \mid a \in \text{dom}(f) \) \} and is denoted \( \text{rng}(f) \).

**Definition.** The function \( f : A \to B \) is onto if \( \text{rng}(f) = B \). \( f \) is one to one if \( f(a) = f(a') \) implies \( a = a' \) for all \( a, a' \in A \).

**Example.** Let \( A = \mathbb{N}_0 \times \mathbb{N}_0 \) and \( B = \mathbb{N}_0 \). The function \( f : A \to B \) defined by
\[
f(x, y) = x + y
\]
is a total binary function. ♠

**Example.** The function \( f : A \to B \) defined by
\[
g(x, y) = x - y
\]
is a partial function with domain \{ \( (x, y) \mid x \geq y \) \}. ♠
**Definition.** A partial or total \( n \)-ary function \( f \) is said to be effectively computable if there is an effective process which, when given any \( n \) argument values \( x_1, \ldots, x_n \), will either

1. eventually halt, yielding \( f(x_1, \ldots, x_n) \) if it is defined, or
2. never halt if \( f(x_1, \ldots, x_n) \) is undefined.

**Definition.** The characteristic function of a set \( A \) is defined as

\[
\chi(x) := \begin{cases} 
1 & x \in A \\
0 & x \notin A
\end{cases}
\]

**Definition.** Let \( f : A \to B \) and \( g : B \to C \) where \( f \) is partial and \( g \) is total. The composition \( g \circ f : A \to C \) is defined as

\[
(g \circ f)(a) := g(f(a)).
\]

If \( f \) is total then \( g \circ f \) is total.

**Example.** In the theory of Lie transformation groups the following function

\[
\exp(\alpha D)f
\]

plays a central role, where \( f : \mathbb{R}^n \to \mathbb{R} \) is an analytic function, \( D \) is the differential operator

\[
D := D_1(x) \frac{\partial}{\partial x_1} + \cdots + D_n(x) \frac{\partial}{\partial x_n}
\]

where \( D_i : \mathbb{R}^n \to \mathbb{R} \) are analytic functions and \( \alpha \) is a parameter \((\alpha \in \mathbb{R})\). If \( n = 1 \) and \( D := d/dx \) we have

\[
\exp(\alpha D)f(x) = f(x + \alpha).
\]

Thus the argument \( x \) of the function \( f \) maps to \( x + \alpha \) (translation). The following C++ program shows an implementation of this function.
// trans.cpp

#include <iostream>
#include <math.h>

using namespace std;

template <class T> T translation(T (*f)(T), T x, T alpha)
{
    return f(x + alpha);
}

double f1(double x)
{
    return sin(x);
}

int f2(int x)
{
    return x*x;
}

void main()
{
    double x1 = 1.0;
    double alpha1 = 0.5;

cout << "f1(x1= " << x1 << ") = " << f1(x1) << endl;
cout << "f1(" << x1 << " + " << alpha1 << ") = "
            << translation(f1,x1,alpha1) << endl;

    int x2 = 5;
    int alpha2 = 3;

cout << "f2(x2= " << x2 << ") = " << f2(x2) << endl;
cout << "f2(" << x2 << " + " << alpha2 << ") = "
            << translation(f2,x2,alpha2) << endl;
}
1.5 Alphabets and Words

Alphabets and words are used as the inputs and outputs for computing. They are used in complexity and computability analysis. Words can also be considered as sequences of characters, i.e. strings.

**Definition.** An *alphabet* is any finite set of symbols.

**Definition.** A *word* over an alphabet $\Sigma$ is any finite string of symbols from $\Sigma$. $\Sigma^*$ denotes the set of all words over $\Sigma$.

**Definition.** The *length* of a word $x$ is the number of symbols contained in $x$ and is denoted by $|x|$.

**Definition.** The word of length 0 is called the *null* or *empty* word and is denoted by $\epsilon$ or $\lambda$.

**Definition.** Let $x, y \in \Sigma^*$ where $x = a_1a_2\ldots a_n$ and $y = b_1b_2\ldots b_m$. The *concatenation* of $x$ and $y$ is $xy = a_1a_2\ldots a_nb_1b_2\ldots b_m$.

**Definition.** $x \in \Sigma^*$ is a *prefix* of $y \in \Sigma^*$ if there exists $z \in \Sigma^*$ such that $y = xz$.

For any symbol $a \in \Sigma^*$, $a^m$ denotes the word of length $m$ consisting of $m$ a’s.

**Definition.** Let $X, Y \subseteq \Sigma^*$.

- $XY = \{ xy \mid x \in X, y \in Y \}$
- 1. $X^0 = \{ \epsilon \}$
- 2. $X^{n+1} = X^nX$, for $n \geq 0$

- $X^* = \bigcup_{n=0}^{\infty} X^n$

- $X^+ = \bigcup_{n=1}^{\infty} X^n$

The set $\Sigma^n$ is the set of all words of length $n$ over $\Sigma$. 
1.5. ALPHABETS AND WORDS

Example. Lindenmayer systems or L-systems consist of a set of rules for modifying a word to produce a new word. Lindenmayer systems play a role in modelling biological systems. The rules specify for each symbol in the alphabet, a word with which to replace it. This system is called a 0L-system. The L-language corresponding to a ruleset is the set of all words derived by successive application of the ruleset to all symbols in the alphabet. An example ruleset for the alphabet \{0, 1\} is

\[ 0 \to 1, \quad 1 \to 01. \]

Thus beginning with 0, this produces a series of derivations as follows

\[ \{0, 1, 01, 101, 01101, 10101101, \ldots \}. \]

This is the L-language for this ruleset. Each word in the derivation is simply the concatenation of the previous two words in the derivation. We can prove this fact by induction. Let \( L(w_j) \) denote the mapping from the bit string \( w_j \) to the next derivation using the ruleset and starting from 0, and \( w_j \) be the \( j \)-th bit string in the derivation. We have

\[
\begin{align*}
  w_0 &= 0 \\
  w_1 &= L(w_0) = 1 \\
  w_2 &= L(w_1) = L(1) = 01 = w_0w_1 \\
  w_3 &= L(w_2) = L(01) = 101 = w_1w_2 \\
  & \vdots \\
  w_{j+1} &= L(w_j)
\end{align*}
\]

By induction

\[
\begin{align*}
  w_{j+1} &= L(w_j) \\
  &= L(w_{j-2}w_{j-1}) \\
  &= L(w_{j-2})L(w_{j-1}) \\
  &= w_{j-1}w_j
\end{align*}
\]

The following Java program shows how to implement the derivation. We use the \texttt{StringBuffer} class which is built into Java. The \texttt{StringBuffer} class implements a mutable sequence of characters. The method

\[
\text{StringBuffer append(String str)}
\]

in class \texttt{StringBuffer} appends the \texttt{String str} to the \texttt{StringBuffer}. 
// LSystem.java

class LSystem
{
    public static void map(StringBuffer sold, StringBuffer snew)
    {
        int i;
        for(i=0; i < sold.length(); i++)
        {
            if(sold.charAt(i) == '0') snew.append("1");
            if(sold.charAt(i) == '1') snew.append("01");
        }
    } // end method map

    public static void main(String[] args)
    {
        StringBuffer sold = new StringBuffer("01101");
        StringBuffer snew = new StringBuffer(""");

        map(sold, snew);
        System.out.println("snew = " + snew); // 10101101

        StringBuffer s0 = new StringBuffer("0");
        StringBuffer s1 = new StringBuffer(""");

        int j;
        for(j=0; j < 6; j++)
        {
            map(s0, s1);
            s0 = s1;
            System.out.println("s = " + s0);
            s1 = new StringBuffer(""");
        }
    }
}
Example. UTF-8 encoding is an efficient method of coding characters and words from many languages as integers. The encoding uses variable length codes to obtain the efficiency by noting that the most common characters used are from the ASCII character set. The Java language uses the methods writeUTF() in class DataOutputStream and readUTF() in class DataInputStream to implement the encoding and decoding to and from UTF-8. ASCII codes (the codes numbered from 1 to 127) are stored in 8 bits with the highest order bit set to zero.

The encoding for a String begins with two bytes for the length of the string. The first byte is the high order byte and the second byte is the low order byte. The character encoding follows this. A zero value is encoded as two bytes

11000000, 10000000.

The bytes are written in left to right order. All ASCII codes from 1 to 127 are written using a single byte with a leading 0 bit,

0 (0 - 6)

where (0-6) indicates that the bits indexed by 0,1,...,6 are written in the remaining bit positions. All codes in the range 128 to 2047 are encoded as two bytes

110(6 - 10), 10(0 - 5).

Finally all codes in the range 2048 to 65535 are encoded as three bytes

1110(12 - 15), 10(6 - 11), 10(0 - 5).

Thus the string “UTF example” would be encoded as the bytes (in hexadecimal)

00, 0B, 55, 54, 46, 20, 65, 78, 61, 6D, 70, 6C, 65.

The following Java program uses the above methods to illustrate the encoding.
// UTFExample.java

import java.io.*;

public class UTFExample
{
    public static void main(String[] args) throws IOException
    {
        DataOutputStream output =
        new DataOutputStream(new FileOutputStream("myout.dat"));

        String s = new String("UTF example");
        System.out.println("s = " + s);

        output.writeUTF(s);
        output.flush();
        output.close();

        DataInputStream input =
        new DataInputStream(new FileInputStream("myout.dat"));

        String t = input.readUTF();
        input.close();

        System.out.println("t = " + t);
    }
}
Chapter 2

Boolean Algebra

2.1 Introduction

Boolean algebra forms the theoretical basis for classical computing. It can be used to describe the circuits which are used as building blocks for classical computing.

In this chapter we introduce the definitions of Boolean algebra and the rules for manipulation. We introduce the standard forms for manipulation and describe how Boolean algebra can be used to describe functions. Efficiency is an important issue in computing and we describe the methods of Karnaugh maps and Quine-McKluskey to simplify expressions.

At the end of the chapter two programs are given to illustrate the concepts. The first example program uses the properties of Boolean algebra to efficiently implement sets in C++. This implementation reduces the memory requirements for a set since only one bit of information is needed for each element of the set. The second example is an implementation of the Quine-McKluskey method in C++. The Quine-McKluskey method is easier to implement on computer whereas the Karnaugh map method is easier to do by hand.

The smallest Boolean algebra consists of two elements usually labelled 0 and 1 or false and true but larger Boolean algebras exist.
2.2 Definitions

Definition. A Boolean algebra is a closed algebraic system containing a set $B$ of two or more elements and two operations

$$
\cdot : B \times B \to B, \quad + : B \times B \to B
$$

with the following properties:

- **Identity Elements.** There exist unique elements $0, 1 \in B$ such that for every $A \in B$
  1. $A + 0 = A$
  2. $A \cdot 1 = A$
- **Commutativity.** For every $A_0, A_1 \in B$
  1. $A_0 + A_1 = A_1 + A_0$
  2. $A_0 \cdot A_1 = A_1 \cdot A_0$
- **Associativity.** For every $A_0, A_1, A_2 \in B$
  1. $A_0 + (A_1 + A_2) = (A_0 + A_1) + A_2$
  2. $A_0 \cdot (A_1 \cdot A_2) = (A_0 \cdot A_1) \cdot A_2$
- **Distributivity.** For every $A_0, A_1, A_2 \in B$
  1. $A_0 + (A_1 \cdot A_2) = (A_0 + A_1) \cdot (A_0 + A_2)$
  2. $A_0 \cdot (A_1 + A_2) = (A_0 \cdot A_1) + (A_0 \cdot A_2)$
- **Complement.** For every $A \in B$ there exists $\overline{A} \in B$ such that
  1. $A + \overline{A} = 1$
  2. $A \cdot \overline{A} = 0$

The operations $\cdot$ and $+$ are referred to as the AND and OR operations respectively. 0 is called the identity element for the OR operation and 1 is called the identity element for the AND operation. The complement will also be referred to as the NOT or negation operation. The AND operation is sometimes referred to as conjunction. The OR operation is sometimes referred to as disjunction.

From the properties of identity elements and complements we find

$$
\overline{\overline{A}} = A \quad \text{and} \quad \overline{1} = 0.
$$
2.2. **DEFINITIONS**

**Example.** The smallest Boolean algebra consists of the identity elements \{0, 1\}. The Boolean algebra can be summarised in a table.

<table>
<thead>
<tr>
<th>$A_0$</th>
<th>$A_1$</th>
<th>$A_0 + A_1$</th>
<th>$A_0 \cdot A_1$</th>
<th>$\overline{A_0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example.** The set $P(X)$ (set of all subsets of the finite set $X$) of a non-empty set $X$ with $\cdot$ the intersection of sets, $+$ the union of sets and the complement with respect to $X$ as negation forms a Boolean algebra with identity elements $0 = \emptyset$ and $1 = X$. This Boolean algebra has $2^{|X|}$ members, where $|X|$ denotes the cardinality (number of elements) of $X$. ♠

**Example.** The set $A$ of all functions from the set \{\(p_1, p_2, \ldots, p_n\)\} into \{0, 1\} (i.e. a function in the set assigns 0 or 1 to each of \(p_1, p_2, \ldots, p_n\)) and $\cdot$, $+$ and negation described pointwise by the definitions in the first example forms a Boolean algebra. For example, if $f_1, f_2 \in A$ then

\[(f_1 + f_2)(p_i) = f_1(p_i) + f_2(p_i)\]

and

\[(f_1 \cdot f_2)(p_i) = f_1(p_i) \cdot f_2(p_i).\]

The Boolean algebra has $2^n$ members and is called a free Boolean algebra on the generators $p_1, p_2, \ldots, p_n$. ♠

**Example.** Let \{\(p_1, p_2, \ldots\)\} be a countably infinite set. Then we can again form a free Boolean algebra on this generating set by considering finite Boolean expressions in the $p_i$. ♠
2.3 Rules and Laws of Boolean Algebra

The following are consequences of the definitions.

- **Double negation.** $\overline{\overline{A}} = A$
- **Idempotence.**
  1. $A \cdot A = A$
  2. $A + A = A$
- **Absorption.**
  1. $A + 1 = 1$
  2. $0 \cdot A = 0$
  3. $A_0 + A_0 \cdot A_1 = A_0$
  4. $A_0 \cdot (A_0 + A_1) = A_0$
  5. $A_0 \cdot A_1 + \overline{A_1} = A_0 + \overline{A_1}$
  6. $(A_0 + A_1) \cdot \overline{A_1} = A_0 \cdot \overline{A_1}$

The double negation property is obvious. The idempotence property follows from

1. $A \cdot A = A \cdot A + 0 = (A \cdot A) + (A \cdot \overline{A}) = A \cdot (A + \overline{A}) = A \cdot 1 = A$
2. $A + A = (A + A) \cdot 1 = (A + A) \cdot (A + \overline{A}) = A + (A \cdot \overline{A}) = A + 0 = A$

The absorption properties are derived as follows

1. $A = A + A = A + (A \cdot 1) = (A + A) \cdot (A + 1) = A \cdot (A + 1)$
2. $A = A \cdot A = A \cdot (A + 0) = (A \cdot A) + (A \cdot 0) = A + (A \cdot 0)$
3. $A_0 + A_0 \cdot A_1 = A_0 \cdot 1 + A_0 \cdot A_1 = A_0 \cdot (1 + A_1) = A_0$
4. $A_0 \cdot (A_0 + A_1) = (A_0 \cdot A_0) + (A_0 \cdot A_1) = A_0 + A_0 \cdot A_1$
5. $A_0 \cdot A_1 + \overline{A_1} = (A_0 + \overline{A_1}) \cdot (A_1 + \overline{A_1}) = (A_0 + \overline{A_1}) \cdot 1 = A_0 + \overline{A_1}$
6. $(A_0 + A_1) \cdot \overline{A_1} = (A_0 \cdot \overline{A_1}) + (A_1 \cdot \overline{A_1}) = (A_0 \cdot \overline{A_1}) + 0 = A_0 \cdot \overline{A_1}$
2.4 DeMorgan’s Theorem

Another property of Boolean algebra is given by DeMorgan’s theorem

\[
\begin{align*}
A_0 \cdot A_1 &\equiv \overline{A_0} + \overline{A_1} \\
\overline{A_0 + A_1} &\equiv \overline{A_0} \cdot \overline{A_1}
\end{align*}
\]

Thus the left hand side of the two identities involves two operations and the right hand side three operations. DeMorgan’s theorem can be proved using the properties given above. It describes the relationships between the operations +, \cdot and negation.

\[
(\overline{A_0 + A_1}) \cdot (A_0 \cdot A_1) = (\overline{A_0} \cdot A_0 + A_1 + (A_0 \cdot A_1 \cdot \overline{A_1})
\]
\[
= 0 \cdot A_1 + A_0 \cdot 0
\]
\[
= 0 + 0 = 0
\]

\[
(\overline{A_0 + A_1}) + (A_0 \cdot A_1) = (A_0 + \overline{A_0} + A_1 \cdot (A_1 + \overline{A_0} + \overline{A_1})
\]
\[
= (1 + \overline{A_1}) \cdot (1 + \overline{A_0})
\]
\[
= 1 \cdot 1 = 1
\]

This theorem is very important for building combinational circuits consisting of only one type of operation.

2.5 Further Definitions

We will use the set \(B = \{0, 1\}\) and the operations AND, OR and complement defined by Table 2.1.

<table>
<thead>
<tr>
<th>(A_0)</th>
<th>(A_1)</th>
<th>(A_0 \cdot A_1)</th>
<th>(A_0 + A_1)</th>
<th>(\overline{A_0})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.1: AND, OR and Complement.
**Definition.** A *Boolean function* is a map $f : \{0, 1\}^n \rightarrow \{0, 1\}$ where $\{0, 1\}^n$ is the set of all $n$-tuples consisting of zeros and ones.

**Definition.** *Boolean variables* are variables which may only take on the values of 0 or 1.

**Definition.** *Bit* is short for *binary digit* which refers to a 0 or 1.

**Definition.** A *literal* is a variable or the complement of a variable.

We will use the notation $B^n := B \times B \times \ldots \times B$ (n times). Thus $B^n = \{0, 1\}^n$.

**Definition.** Any function $f : B^n \rightarrow B$ can be represented with a *truth table*.

<table>
<thead>
<tr>
<th>$A_0$</th>
<th>$A_1$</th>
<th>$\ldots$</th>
<th>$A_{n-1}$</th>
<th>$f(A_0, A_1, \ldots, A_{n-1})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$\ldots$</td>
<td>0</td>
<td>$f(A_0 = 0, A_1 = 0, \ldots, A_{n-1} = 0)$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$\ldots$</td>
<td>1</td>
<td>$f(A_0 = 0, A_1 = 0, \ldots, A_{n-1} = 1)$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\ldots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$\ldots$</td>
<td>1</td>
<td>$f(A_0 = 1, A_1 = 1, \ldots, A_{n-1} = 1)$</td>
</tr>
</tbody>
</table>

The rows of the table are over all combinations of $A_0, A_1, \ldots, A_{n-1}$.

There are $2^n$ such combinations. Thus the truth table has $2^n$ rows.

**Definition.** Two functions $f : B^n \rightarrow B$, $g : B^n \rightarrow B$ are *equivalent* if

$$f(A_0 = a_0, \ldots, A_{n-1} = a_{n-1}) = g(A_0 = a_0, \ldots, A_{n-1} = a_{n-1})$$

for all $A_0, \ldots, A_{n-1} \in \{0, 1\}$.

**Definition.** A *product form* is an AND of a number of literals $l_1 \cdot l_2 \ldots \cdot l_m$.

**Definition.** A *sum of products (SOP)* form is an OR of product forms

$$P_1 + P_2 + \ldots + P_m.$$ 

**Definition.** *Disjunctive normal form* is a disjunction of conjunctions of literals. This is equivalent to SOP form.

**Definition.** *Conjunctive normal form* is a conjunction of disjunctions of literals.
Theorem. Any function $f : B^n \to B$ can be represented in SOP form.

To see this we construct product forms $P_j = l_{j,1} \cdot l_{j,2} \cdots l_{j,n}$ for each row in the truth table of $f$ where $f = 1$ with $l_{j,i} = A_i$ if the entry for $A_i$ is 1 and $l_{j,i} = \overline{A_i}$ if the entry for $A_i$ is 0. If $f = 1$ in $m$ of the rows of the truth table then

$$f(A_0, A_1, \ldots, A_{n-1}) = P_1 + P_2 + \ldots + P_m.$$  

Example. Consider the parity function for two bits with truth table Table 2.2.

<table>
<thead>
<tr>
<th>$A_0$</th>
<th>$A_1$</th>
<th>$P(A_0, A_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.2: Parity Function

Then $P(A_0, A_1) = \overline{A_1} \cdot \overline{A_2} + A_0 \cdot A_1$. ♦

Definition. A canonical SOP form is a SOP form over $n$ variables, where each variable or its negation is present in every product form, in other words a Boolean expression $E$ is in canonical SOP form if it can be written as

$$E = l_{1,1} \cdot l_{1,2} \cdots l_{1,n} + l_{2,1} \cdot l_{2,2} \cdots l_{2,n} + \ldots + l_{m,1} \cdot l_{m,2} \cdots l_{m,n}$$

where $l_{i,j} = A_j$ or $l_{i,j} = \overline{A_j}$.

Definition. The exclusive OR function is defined by the following table.

<table>
<thead>
<tr>
<th>$A_0$</th>
<th>$A_1$</th>
<th>$A_0 \oplus A_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.3: XOR Truth Table
Some more properties of the XOR operation are given below:

- \( A \oplus A = 0 \)
- \( A \oplus \overline{A} = 1 \)
- \( A_0 \oplus A_1 = A_1 \oplus A_0 \)
- \( A_0 \oplus A_1 = \overline{A_0} \oplus \overline{A_1} \)
- \( (A_0 \oplus A_1) \oplus A_2 = A_0 \oplus (A_1 \oplus A_2) \)
- \( A_0 \oplus A_1 = A_0 \cdot \overline{A_1} + \overline{A_0} \cdot A_1 \)
- \( (A_0 \cdot \overline{A_1}) \oplus A_0 = (A_0 \oplus \overline{A_1}) \cdot A_0 = A_0 \cdot A_1 \)

The XOR operation can be used to swap two values \( a \) and \( b \) (for example integers in C++ and Java):

1. \( a := a \oplus b \)
2. \( b := a \oplus b \)
3. \( a := a \oplus b \)

By analysing the variables at each step in terms of the original \( a \) and \( b \) the swapping action becomes clear. In the second step we have \( (a \oplus b) \oplus b = a \oplus 0 = a \). In the third step we have \( (a \oplus b) \oplus a = b \oplus 0 = b \).

In C, C++ and Java the XOR operation is denoted by \(^\sim\). The following C++ program illustrates the swapping.

```
// xor.cpp

#include <iostream>

using namespace std;

void main(void)
{
    int a=23;
    int b=-565;

    cout << "a = " << a << " , b = " << b << endl;
    a ^= b;
    b ^= a;
    a ^= b;
    cout << "a = " << a << " , b = " << b << endl;
}
```
The results are

\[ a = 23 \ , \ b = -565 \]
\[ a = -565 \ , \ b = 23 \]
Definition. The operation $A_0 + A_1$ is called the $NOR$ function.

Example. Let $A_0 = 0$ and $A_1 = 0$. Then

$$A_0 + A_1 = 1$$

Definition. The operation $A_0 \cdot A_1$ is called the $NAND$ function.

Example. Let $A_0 = 1$ and $A_1 = 0$. Then

$$A_0 \cdot A_1 = 1$$

Definition. The operation $A_0 \oplus A_1$ is called the $XNOR$ function.

Example. Let $A_0 = 1$ and $A_1 = 1$. Then

$$A_0 \oplus A_1 = 1$$

Definition. A universal set of operations is a set of operations which can be used to build any Boolean function.

For Boolean functions there exist universal sets of operations with only one element.

For simplicity of implementation, it is useful to know the minimum number of parameters a Boolean function must take in order to be able to build any other Boolean function. Obviously functions taking only a single parameter cannot fulfill this requirement. The minimum number of parameters is thus at least two.

The NAND and NOR operations can be used to build any other function which we will show in the next section.
2.6 Boolean Function Implementation

The physical implementation of a Boolean function is achieved by interconnection of gates. A gate is an electronic circuit that produces an output signal (representing 0 or 1) according to the signal states (again representing 0 or 1) of its inputs. The task is then to build an implementation of a Boolean function with gates of prescribed types. This is not always possible, for example the NOT operation cannot implement two bit operations and the OR operation cannot be used to implement the AND and NOT operations. In the previous section it was shown informally that any Boolean function can be implemented with AND, OR and NOT operations (in SOP form). Therefore to show that any Boolean function can be implemented with a set of gates it is sufficient to show that AND, OR and NOT can be implemented with the set of gates.

The NAND gate is sufficient to build an implementation of any Boolean function:

- $\overline{A} = A \cdot A$
- $A_0 \cdot A_1 = \overline{A_0 \cdot A_1} = \overline{A_0} \cdot \overline{A_1} \cdot A_0 \cdot A_1$
- $A_0 + A_1 = \overline{A_0 \cdot A_1} = \overline{A_0} \cdot \overline{A_0} \cdot \overline{A_1} \cdot A_0 \cdot A_1 \cdot A_1$

\(\clubsuit\)

**Example.** We show now how to implement the NOR operation using only NAND operations. As mentioned earlier De Morgan’s laws are important to achieve this.

$$A_0 + A_1 = \overline{A_0 \cdot A_0 \cdot A_1 \cdot A_1} = \overline{A_0} \cdot \overline{A_0} \cdot \overline{A_1} \cdot A_0 \cdot A_0 \cdot A_1 \cdot A_1$$

\(\heartsuit\)

It can also be shown that the NOR gate is sufficient to build an implementation of any Boolean function.

Data are represented by bit strings $a_{n-1}a_{n-2}\ldots a_0$, $a_i \in \{0, 1\}$. Bit strings of length $n$ can represent up to $2^n$ different data elements. Functions on bit strings are then calculated by

$$f(a_{n-1}a_{n-2}\ldots a_0) = f_{m-1}(a_{n-1}a_{n-2}\ldots a_0)f_{m-2}(a_{n-1}a_{n-2}\ldots a_0)\ldots f_0(a_{n-1}a_{n-2}\ldots a_0)$$

with $f : B^n \rightarrow B^m$ and $f_i : B^n \rightarrow B$. In other words a function of a bit string of length $n$ gives a bit string of length say $m$, each bit in the output string is therefore a function of the input bits. It is sufficient then to consider functions with an output of only one bit.
Example. The set
\[ \{ z \mid z \in \mathbb{N}_0, \ 0 \leq z < 2^n \} \]
can be represented by
\[ a_{n-1}a_{n-2} \ldots a_0 \rightarrow \sum_{i=0}^{n-1} a_i 2^i. \]

If \( n = 32 \) the largest integer number we can represent is
\[ \sum_{i=1}^{n-1} 2^i = 2^n - 1 = 4294967295. \]

This relates to the data type unsigned long in C and C++. Java has only signed
data types. ♠

Example. The set
\[ \{ x \mid x \in \mathbb{R}, \ x = b + \frac{c - b}{2^n - 1}, \ j = 0, 1, \ldots, 2^n - 1 \} \]
where \( b, c \in \mathbb{R} \) and \( c > b \) can be represented by
\[ a_{n-1}a_{n-2} \ldots a_0 \rightarrow b + \frac{c - b}{2^n - 1} \sum_{i=0}^{n-1} a_i 2^i. \]

So we find
\[ a_{n-1}a_{n-2} \ldots a_0 = 00 \ldots 0 \rightarrow b \]
and
\[ a_{n-1}a_{n-2} \ldots a_0 = 11 \ldots 1 \rightarrow c. \]

♠
2.6. BOOLEAN FUNCTION IMPLEMENTATION

Minimizing the number of gates in an implementation decreases cost and the number of things that can go wrong.

One way to reduce the number of gates is to use the properties of the Boolean algebra to eliminate literals.

**Example.** The full adder (Table 2.4) consists of two outputs (one for the sum and one for the carry) and three inputs (the carry from another adder and the two bits to be added).

<table>
<thead>
<tr>
<th>$C_{in}$</th>
<th>$A_0$</th>
<th>$A_1$</th>
<th>$S$</th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.4: Full Adder

Thus

$$S = A_0 \cdot A_1 \cdot \overline{C_{in}} + A_0 \cdot \overline{A_1} \cdot \overline{C_{in}} + A_0 \cdot A_1 \cdot C_{in} + A_0 \cdot A_1$$

and

$$C_{out} = A_0 \cdot A_1 \cdot \overline{C_{in}} + A_0 \cdot A_1 \cdot C_{in} + A_0 \cdot \overline{A_1} \cdot C_{in} + A_0 \cdot A_1 \cdot C_{in}$$

Simplification for $C_{out}$ yields

$$C_{out} = A_0 \cdot A_1 \cdot (C_{in} + \overline{C_{in}}) + (A_0 + A_0) \cdot A_1 \cdot C_{in} + A_0 \cdot (A_1 + A_1) \cdot C_{in} + A_0 \cdot A_1 \cdot C_{in} + A_0 \cdot A_1 \cdot C_{in}$$

$$= A_0 \cdot A_1 + A_1 \cdot C_{in} + A_0 \cdot C_{in}$$

The half adder is a full adder where $C_{in} = 0$ is fixed. ♠

Karnaugh maps and the Quine-McCluskey method [174] for simplification of Boolean expressions are discussed next.
2.6.1 Karnaugh Maps

Karnaugh maps can be used to simplify Boolean expressions of up to six variables. When an expression has more than four variables the Karnaugh map has dimension greater than 2. For 2, 3 and 4 variables the Karnaugh map is represented by $2 \times 2$, $2 \times 4$ and $4 \times 4$ grids, respectively. The rows and columns of the grid represent the values of variables. Each square contains the expression value for the variables with values given by the row and column.

**Example.** The Karnaugh map for the carry flag

$$C_{out} = A_0 \cdot A_1 \cdot \overline{C_{in}} + A_0 \cdot A_1 \cdot C_{in} + A_0 \cdot \overline{A_1} \cdot C_{in} + A_0 \cdot A_1 \cdot C_{in}$$

of the full adder is as follows

<table>
<thead>
<tr>
<th>$C_{in}$</th>
<th>$A_0A_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00 01 11 10</td>
</tr>
<tr>
<td>1</td>
<td>0 1 1 1</td>
</tr>
</tbody>
</table>

Note that adjacent columns and rows only differ in the assignment of one variable (only one bit differs). This is important for the simplification algorithm to work correctly. Suppose two adjacent squares have the value 1 and only differ in the variable $A$. Writing the corresponding product forms as $P \cdot A$ and $P \cdot \overline{A}$ the canonical SOP form can be simplified using

$$P \cdot A + P \cdot \overline{A} = P.$$ 

In fact this can be extended to any $2^n$ adjacent squares in a row or column. The first column is adjacent to the last column ("wrap around"), and the same applies to rows. The simplification is indicated by circling the adjacent squares involved in the simplification. Overlapping circles are allowed due to the idempotence property. If two circles are "adjacent" in the sense that they cover the same columns(rows) in adjacent rows(columns) they may be joined to form one circle encircling all the appropriate squares. The only restriction is that the number of rows and columns encircled are a power of 2, i.e. 1,2,4,8, ... This is due to the algebraic simplification used. Each set of encircled squares is called a *group* and the squares are said to be *covered*. There are two algorithms for this method.
2.6. BOOLEAN FUNCTION IMPLEMENTATION

Algorithm 1.

1. Count the number of adjacencies (adjacent 1-squares) for each 1-square on the Karnaugh map.

2. Select an uncovered 1-square with the fewest number of adjacencies. (An arbitrary choice may be required.)

3. Circle the 1-square so that the circle covers the most uncovered 1-squares.

4. If all the 1-squares are not yet covered goto 2

Algorithm 2.

1. Circle all 1-squares so that the circle covers the most 1-squares.

2. Eliminate all circles that do not contain at least one 1-square that is not covered by another circle.

3. Introduce a minimum number of circles to complete the cover.

The SOP form is the OR of product forms representing the groups of the Karnaugh map. The variable \( A_i \) is in the product form if \( A_i = 1 \) is constant in the group, \( \overline{A}_i \) is in the product form if \( A_i = 0 \) is constant in the group.

Example. The Karnaugh map for the carry flag

\[
C_{out} = A_0 \cdot A_1 \cdot C_{in} \oplus A_0 \cdot A_1 \cdot C_{in} \oplus A_0 \cdot \overline{A}_1 \cdot C_{in} \oplus A_0 \cdot A_1 \cdot C_{in}
\]

is the same after application of either algorithm

<table>
<thead>
<tr>
<th>( C_{in} )</th>
<th>( \overline{A}_0 \overline{A}_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

So \( E = A_0 \cdot A_1 + A_1 \cdot C_{in} + A_0 \cdot C_{in} \)
An advantage of Karnaugh maps is that simpler expressions can be found when certain inputs cannot occur. A "don’t care" symbol is placed in the squares on the Karnaugh map which represent the inputs which cannot occur. These d-squares can be interpreted as 1-squares or 0-squares for optimal grouping.

**Example.** The truth table for a decimal incremenetor (4-bit) with 4 inputs and 4 outputs is given by Table 2.5.

<table>
<thead>
<tr>
<th>Number in</th>
<th>Number out</th>
<th>$I_3$</th>
<th>$I_2$</th>
<th>$I_1$</th>
<th>$I_0$</th>
<th>$O_3$</th>
<th>$O_2$</th>
<th>$O_1$</th>
<th>$O_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: 4-bit Decimal Incrementer

The Karnaugh map for $O_0$ and $O_3$ is:

\[
\begin{array}{c|c|c|c|c|c}
I_0I_1 & 00 & 01 & 11 & 10 \\
\hline
I_2I_3 & 00 & 1 & 0 & 0 \\
01 & d & d & d \\
11 & d & d & d & d \\
10 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
I_0I_1 & 00 & 01 & 11 & 10 \\
\hline
I_2I_3 & 00 & 0 & 0 & 0 \\
01 & 1 & d & d & d \\
11 & d & d & d & d \\
10 & 0 & 0 & 1 & 0 \\
\end{array}
\]

Therefore $O_0 = \overline{I_0}$ and $O_3 = I_0 \cdot I_3 + I_0 \cdot I_1 \cdot I_2$. ♦
2.6.2 Quine-McKluskey Method

This method provides an algorithmic description for simplification which lends itself to implementation in programming languages such as C++ and Java. It is also more general than the method of Karnaugh maps and can handle an arbitrary number of Boolean variables. Suppose the Boolean expression to be simplified has the representation in canonical SOP form

\[
E = P_{0,1}(A_0, A_1, \ldots, A_n) + P_{0,2}(A_0, A_1, \ldots, A_n) + \ldots \\
+ P_{1,1}(A_0, A_1, \ldots, A_n) + P_{1,2}(A_0, A_1, \ldots, A_n) + \ldots \\
+ \ldots \\
+ P_{n,1}(A_0, A_1, \ldots, A_n) + P_{n,2}(A_0, A_1, \ldots, A_n) + \ldots
\]

where \( P_{i,j} \) denotes the \( j \)th product form with exactly \( i \) negated Boolean variables. The method is as follows

1. Let \( QM(n) := \{ P_{i,j} \mid i = 0,1,\ldots,n \quad j = 1,2,\ldots \} \)
2. Set \( m := n \).
3. Set
   \[
   QM(m - 1) := QM(m)
   \]
   and
   \[
   QM_{m,i} := \{ P \in QM(m) \mid P \text{ has } m \text{ Boolean variables of which } i \text{ are negated} \}
   \]
4. For each pair of elements
   \[
e_1 = l_{1,1} \cdot l_{1,2} \cdot \ldots \cdot l_{1,m} \in QM_{m,i} \\
   \text{and} \quad e_2 = l_{2,1} \cdot l_{2,2} \cdot \ldots \cdot l_{2,m} \in QM_{m,i+1}
   \]
   where \( i = 0,1,\ldots,m-1 \)
   which differ in only one literal \( l_{1,j} \neq l_{2,j} \)
   set
   \[
   QM(m - 1) := (QM(m - 1) - \{ e_1, e_2 \}) \cup \{ l_{1,1} \cdot \ldots \cdot l_{1,j-1} \cdot l_{1,j+1} \cdot \ldots \cdot l_{1,m} \}
   \]
5. Set \( m := m - 1 \).
6. If \( m > 0 \) goto step 3
7. \( E \) is the OR of all the elements of \( QM(0) \).
Example. We consider the two’s complement operation on two bits.

<table>
<thead>
<tr>
<th>$I_0$</th>
<th>$I_1$</th>
<th>$O_0$</th>
<th>$O_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.6: Two’s Complement Operation on 2 Bits

Thus $O_1 = \overline{T_0} \cdot I_1 + I_0 \cdot I_1$.

- $m = 2$.
  
  $QM(2) = \{T_0 \cdot I_1, I_0 \cdot I_1\}$
  $QM_{0,2} = \{I_0 \cdot I_1\}$
  $QM_{1,2} = \{T_0 \cdot I_1\}$
  $QM_{2,2} = \emptyset$

- $m = 2$.
  
  $QM(1) = \{I_1\}$
  $QM_{0,2} = \{I_0 \cdot I_1\}$
  $QM_{1,2} = \{T_0 \cdot I_1\}$
  $QM_{2,2} = \emptyset$

- $m = 1$.
  
  $QM(0) = \{I_1\}$
  $QM_{0,1} = \{I_1\}$
  $QM_{1,1} = \emptyset$

The method yields $O_1 = I_1$. ♦

Example. The method applied to the carry flag

$$C_{out} = A_0 \cdot A_1 \cdot \overline{C_{in}} + \overline{A_0} \cdot A_1 \cdot C_{in} + A_0 \cdot \overline{A_1} \cdot C_{in} + A_0 \cdot A_1 \cdot C_{in}$$

of the full adder is as follows

- $m = 3$.
  
  $QM(3) = \{A_0 \cdot A_1 \cdot \overline{C_{in}}, \overline{A_0} \cdot A_1 \cdot C_{in}, A_0 \cdot \overline{A_1} \cdot C_{in}, A_0 \cdot A_1 \cdot C_{in}\}$
  $QM_{0,3} = \{A_0 \cdot A_1 \cdot C_{in}\}$
  $QM_{1,3} = \{\overline{A_0} \cdot A_1 \cdot C_{in}, A_0 \cdot \overline{A_1} \cdot C_{in}, A_0 \cdot A_1 \cdot \overline{C_{in}}\}$
  $QM_{2,3} = \emptyset$
  $QM_{3,3} = \emptyset$
2.6. BOOLEAN FUNCTION IMPLEMENTATION

- $m = 3.$
  \[QM(2) = \{A_0 \cdot A_1 \cdot \overline{C_{in}}, \ A_1 \cdot C_{in}, \ A_0 \cdot \overline{A_1} \cdot C_{in}\}\]
  \[QM_{0,3} = \{A_0 \cdot A_1 \cdot C_{in}\}\]
  \[QM_{1,3} = \{\overline{A_0} \cdot A_1 \cdot C_{in}, \ A_0 \cdot \overline{A_1} \cdot C_{in}, \ A_0 \cdot A_1 \cdot \overline{C_{in}}\}\]
  \[QM_{2,3} = \emptyset\]
  \[QM_{3,3} = \emptyset\]

- $m = 3.$
  \[QM(2) = \{A_0 \cdot A_1 \cdot \overline{C_{in}}, \ A_1 \cdot C_{in}, \ A_0 \cdot C_{in}\}\]
  \[QM_{0,3} = \{A_0 \cdot A_1 \cdot C_{in}\}\]
  \[QM_{1,3} = \{\overline{A_0} \cdot A_1 \cdot C_{in}, \ A_0 \cdot \overline{A_1} \cdot C_{in}, \ A_0 \cdot A_1 \cdot \overline{C_{in}}\}\]
  \[QM_{2,3} = \emptyset\]
  \[QM_{3,3} = \emptyset\]

- $m = 3.$
  \[QM(2) = \{A_0 \cdot A_1, \ A_1 \cdot C_{in}, \ A_0 \cdot C_{in}\}\]
  \[QM_{0,3} = \{A_0 \cdot A_1 \cdot C_{in}\}\]
  \[QM_{1,3} = \{\overline{A_0} \cdot A_1 \cdot C_{in}, \ A_0 \cdot \overline{A_1} \cdot C_{in}, \ A_0 \cdot A_1 \cdot \overline{C_{in}}\}\]
  \[QM_{2,3} = \emptyset\]
  \[QM_{3,3} = \emptyset\]

- $m = 2.$
  \[QM(1) = \{A_0 \cdot A_1, \ A_1 \cdot C_{in}, \ A_0 \cdot C_{in}\}\]
  \[QM_{0,2} = QM(1)\]
  \[QM_{1,2} = \emptyset\]
  \[QM_{2,2} = \emptyset\]
  \[QM_{3,2} = \emptyset\]

- $m = 1.$
  \[QM(0) = \{A_0 \cdot A_1, \ A_1 \cdot C_{in}, \ A_0 \cdot C_{in}\}\]
  \[QM_{0,1} = \emptyset\]
  \[QM_{1,1} = \emptyset\]
  \[QM_{2,1} = \emptyset\]
  \[QM_{3,1} = \emptyset\]

- $E = A_0 \cdot A_1 + A_1 \cdot C_{in} + A_0 \cdot C_{in}$

Thus we have reduced the expression to one consisting of only two types of operations and 5 operations in total. This is a large reduction compared to the original total of 11 operations. The example also illustrates that the process is long but simple enough to implement, making it a good application for a computing device.

\[\Diamond\]
2.7 Example Programs

2.7.1 Efficient Set Operations Using Boolean Algebra

Let

\[ U := \{ o_0, o_1, \ldots, o_{n-1} \} \]

be the universal set of \( n \) objects. Any subset \( A \) of \( U \) can be represented with a sequence of bits

\[ A := a_0a_1\ldots a_{n-1} \]

where \( a_i = 1 \) if \( o_i \in A \) and \( a_i = 0 \) otherwise. For example, let \( n = 8 \) and consider the bitstring

\[ 10010111. \]

Then \( o_0, o_3, o_5, o_6, o_7 \) are in the set and \( o_1, o_2, o_4 \) are not in the set.

Now the set \( A \cup B \) (union of \( A \) and \( B \)) corresponds to

\[ A + B := (a_0 + b_0)(a_1 + b_1)\ldots(a_{n-1} + b_{n-1}), \]

and \( A \cap B \) (intersection of \( A \) and \( B \)) corresponds to

\[ A \cdot B := (a_0 \cdot b_0)(a_1 \cdot b_1)\ldots(a_{n-1} \cdot b_{n-1}). \]

The complement of \( A \) corresponds to

\[ \overline{A} := a_0\overline{a_0}\ldots a_{n-1}. \]

For example if

\[ A = 11010100 \]
\[ B = 01101101 \]

then

\[ A \cup B = 11111101 \]
\[ A \cap B = 01000100 \]
\[ \overline{A} = 00101011. \]
The following C++ program `bitset.cpp` implements these concepts. The class `BitSet` implements all the bitwise operations introduced above. We could also use the `bitset` which is part of the standard template library, which includes all the methods needed to implement complement, intersection and union.

```cpp
// bitset.cpp

#include <iostream>
#include <string>

using namespace std;

class SetElementBase
{
    public:
        virtual void output(ostream&) = 0;
};

template <class T>
class SetElement: public SetElementBase
{
    protected:
        T data;
    public:
        SetElement(T t) : data(t) {}
        virtual void output(ostream& o) { o << data; }
};

class BitSet
{
    protected:
        char *set;
        int len;
    SetElementBase **universe;
    static int byte(int);
    static char bit(int);
    public:
        BitSet(SetElementBase**, int, int*, int);
        BitSet(const BitSet&);
        BitSet &operator=(const BitSet&);
        BitSet operator+(const BitSet&) const; // union
        BitSet operator*(const BitSet&) const; // intersection
        BitSet operator~(void) const; // complement
        void output(ostream&) const;
```
```cpp
- BitSet();

int BitSet::byte(int n) { return n>>3; }  
char BitSet::bit(int n) { return 1<<(n%8); }

// Create a BitSet with universe un of n elements,  
// with m elements given by el
BitSet::BitSet(SetElementBase **un, int n, int *el, int m)  
{  
    int i;  
    len = n;  
    universe = un;  
    set = new char[byte(len)+1];  
    for(i=0;i<byte(len)+1;i++) set[i] = 0;  
    if(m > 0)  
        for(i=0;i<m;i++) set[byte(el[i])] |= bit(el[i]);
}

BitSet::BitSet(const BitSet &b)  
{  
    int i;  
    len = b.len;  
    universe = b.universe;  
    set = new char[byte(len)+1];  
    for(i=0;i<byte(len)+1;i++) set[i] = b.set[i];
}

BitSet &BitSet::operator=(const BitSet &b)  
{  
    if(this!=&b)  
    {  
        int i;  
        delete[] set;  
        len = b.len;  
        universe = b.universe;  
        set = new char[byte(len)+1];  
        for(i=0;i<byte(len)+1;i++) set[i] = b.set[i];
    }  
    return *this;
}

BitSet BitSet::operator+(const BitSet &b) const  
{  
    if(universe == b.universe)
```
{  
    int i;
    BitSet c(universe, len, NULL, 0);
    for (i = 0; i < byte(len) + 1; i++) c.set[i] = set[i] | b.set[i];
    return c;
}  
else return *this;
}

BitSet BitSet::operator*(const BitSet &b) const
{
    if (universe == b.universe)
    {
        int i;
        BitSet c(universe, len, NULL, 0);
        for (i = 0; i < byte(len) + 1; i++) c.set[i] = set[i] & b.set[i];
        return c;
    }
    else return *this;
}

BitSet BitSet::operator~(void) const
{
    int i;
    BitSet b(universe, len, NULL, 0);
    for (i = 0; i < byte(len) + 1; i++) b.set[i] = ~set[i];
    return b;
}

void BitSet::output(ostream &o) const
{
    int i, start = 0;
    o << "{";
    if ((set[byte(0)] & bit(0)) != 0) universe[0] -> output(o);
    for (i = 0; i < len; i++)
        if ((set[byte(i)] & bit(i)) != 0)
        {
            if (start) o << ", "; universe[i] -> output(o); start = 1; }
    o << "}";
}

BitSet::~BitSet() { delete[] set; }

ostream &operator<<(ostream& o, const BitSet &b)
{
    b.output(o);
return o;
}

void main(void)
{
    SetElement<int> s1(5);
    SetElement<string> s2(string("element"));
    SetElement<double> s3(3.1415927);
    SetElement<int> s4(8);
    SetElement<int> s5(16);
    SetElement<int> s6(3);
    SetElement<string> s7(string("string"));
    SetElement<double> s8(2.7182818);
    SetElement<int> s9(32);
    SetElement<int> s10(64);
    SetElementBase *universe[10]=
    {
        &s1,&s2,&s3,&s4,&s5,&s6,&s7,&s8,&s9,&s10
    };

cout << "Universe=" << (!BitSet(universe,10,NULL,0)) << endl;
cout << "Empty set=" << BitSet(universe,10,NULL,0) << endl;
    int a[7] = {1,2,5,6,8,9};
    int b[4] = {3,5,7,8};
    BitSet A(universe,10,a,7);
    BitSet B(universe,10,b,4);
    cout << "A=" << A << endl;
    cout << "B=" << B << endl;
    cout << "-A=" << (-A) << endl;
    cout << "A+B=" << (A+B) << endl;
    cout << "A*B=" << (A*B) << endl;
}

The output of the bitset.cpp program is

Universe={55, element, 3.14159, 8, 16, 3, string, 2.71828, 32, 64}
Empty set={}
A={55, element, 3.14159, 3, string, 32, 64}
B={8, 3, 2.71828, 32}
-A={8, 16, 2.71828}
A+B={55, element, 3.14159, 8, 3, string, 2.71828, 32, 64}
A*B={3, 32}
2.7.2 Quine-McKluskey Implementation

The next C++ program illustrates the Quine-McKluskey method for the full adder. The algorithm is modified slightly to make the implementation easier. For an $n$-Boolean variable expression the program maintains $n + 1$ sets $S_0, \ldots, S_n$ where $S_i$ contains product forms of exactly $n$ variables. Initially only $S_n$ is non-empty and contains all the product forms from the expression to be simplified. The program fills the set $S_i$ by combining and simplifying two product forms from $S_{i+1}$ which only differ in one literal (using previously discussed methods). Once all such product forms have been simplified the product forms used in simplification are removed from $S_{i+1}$. As input, the program takes an array of product forms (where product forms are arrays of characters) where the value 1 means the variable is present in the product form and 0 means the negation of the variable is present in the product form. Thus the arrays can be constructed directly from truth tables.

The program quine.cpp simplifies the carry and sum bits from the full adder. In the main function we consider the expressions for $C_{out}$ and $S$ for the full adder. We use an array of three char to represent a product form, a 1 indicates that the literal in the product form is not a negated variable and a 0 indicates that the literal is a negated variable. The variable is identified by the index in the array, for example the program uses index 0 for $A_0$, index 1 for $A_1$ and index 2 for $C_{in}$. These arrays (representing product forms) are placed in an array representing the final SOP form.

The function complimentary searches for complimentary literals in a set of product forms in order to perform simplification, AddItem is used to add elements to the sets maintained in the algorithm, DeleteItem removes elements from these sets (the simplification). The function QuineRecursive is the main implementation of the algorithm. It is implemented using recursion. This is not necessary but simplifies the implementation. The function QuineMcKluskey prepares the data for the QuineRecursive function.

The output of the program is

$C_{out}=A_0 \cdot A_1 + A_1 \cdot C_{in} + A_0 \cdot C_{in}$
$S=\text{NOT}(A_0) \cdot A_1 \cdot \text{NOT}(C_{in}) + A_0 \cdot \text{NOT}(A_1) \cdot \text{NOT}(C_{in})$
$+ \text{NOT}(A_0) \cdot \text{NOT}(A_1) \cdot C_{in} + A_0 \cdot A_1 \cdot C_{in}$

which is the same result obtained in earlier examples. The sum bit could not be simplified.
// quine.cpp

#include <iostream>

using namespace std;

struct QMelement
{
    int nvars,used;
    char *product;
    int *vars;
    QMelement *next;
};

int complementary(QMelement *p1,QMelement *p2)
{
    int sum = 0,i;
    if(p1->nvars != p2->nvars) return 0;
    for(i=0;i<p1->nvars;i++) sum += (p1->vars[i]!=p2->vars[i]);
    if(sum == 0)
        for(i=0;i<p1->nvars;i++)
            sum += (p1->product[p1->vars[i]]!=p2->product[p2->vars[i]]);
    else sum = 0;
    return (sum == 1);
}

void AddItem(QMelement* &list,char *product,int nvars,int *vars)
{
    int i;
    QMelement *item;
    if(list == (QMelement*)NULL)
    {
        list = new QMelement;
        list->nvars = nvars;
        list->next = (QMelement*)NULL;
        list->product = new char[nvars];
        list->vars = new int[nvars];
        list->used = 0;
        for(i=0;i<nvars;i++)
        {
            list->product[i] = product[i];
            list->vars[i] = vars[i];
        }
    }
    else
```c
{  
    item = list;
    while(item->next != (QMelement*)NULL) item = item->next;
    item = (item->next = new QMelement);
    item->nvars = nvars;
    item->next = (QMelement*)NULL;
    item->product = new char[nvars];
    item->vars = new int[nvars];
    item->used = 0;
    for(i=0;i<nvars;i++)
    {
        item->product[i] = product[i];
        item->vars[i] = vars[i];
    }
}

void DeleteItem(QMelement **&set,QMelement *item)
{
    QMelement *last = set;
    if(item == set) set = set->next;
    else
    {
        while(last->next != item) {last = last->next;}
        last->next = item->next;
    }
    delete[] item->product;
    delete[] item->vars;
    delete item;
}

void DeleteItems(QMelement *set)
{
    QMelement *item = set,*next;
    while(item != (QMelement*)NULL)
    {
        next = item->next;
        delete[] item->product;
        delete[] item->vars;
        delete item;
        item = next;
    }
}

void QuineRecursive(QMelement **sets,int index)
```c
{  
if(index<1) return;
if(sets[index] == (QMelement*)NULL) return;
int i,j;
QMelement *item1 = sets[index],*item2;
while(item1 != (QMelement*)NULL)
{
  if(item1->next != (QMelement*)NULL)
  {
    item2 = item1->next;
    while(item2 != (QMelement*)NULL)
    {
      if(complementary(item1,item2))
      {
        char *product = new char[item1->nvars-1];
        int *vars = new int[item1->nvars-1];
        for(i=0,j=0; i<item1->nvars; i++)
          if(item1->product[i] == item2->product[i])
          {
            product[j] = item1->product[i];
            vars[j++] = i;
          }
        AddItem(sets[index-1],product,item1->nvars-1,vars);
        delete[] product;
        delete[] vars;
        item1->used = item2->used=1;
      }
      item2 = item2->next;
    }
  }
  item2 = item1;
  item1 = item1->next;
  if(item2->used) DeleteItem(sets[index],item2);
}
QuineRecursive(sets,index-1);
}

void QuineMcKluskey(char **sop,int nproducts,int nvars,char **names)
{
  int i,j,*vars = new int[nvars];
  QMelement **sets = new QMelement*[nvars+1];
  for(i=0;i<=nvars;i++)
  {sets[i] = (QMelement*)NULL; if(i < nvars) vars[i] = i;}
  for(i=0;i<nproducts;i++) AddItem(sets[nvars],sop[i],nvars,vars);
  QuineRecursive(sets,nvars);
}
2.7. EXAMPLE PROGRAMS

delete[] vars;
for(i=0;i<=nvars;i++)
{
    QMelement *item = sets[i];
    while(item != (QMelement*)NULL)
    {
        for(j=0;j<item->nvars;j++)
        {
            if(item->product[j] == 1) cout << names[item->vars[j]];
            else cout << "NOT(" << names[item->vars[j]] << ");"
            if(j != item->nvars-1) cout << ";"
        }
        item = item->next;
        if(item != (QMelement*)NULL) cout << "+";
        else if(i != nvars)
            if(sets[i+1] != (QMelement*)NULL) cout << "+";
    }
    DeleteItems(sets[i]);
}

void main(void)
{
    //carry flag
    char c1[3]={1,1,0}, c2[3]={0,1,1}, c3[3]={1,0,1}, c4[3]={1,1,1};
    //sum
    char s1[3]={0,1,0}, s2[3]={1,0,0}, s3[3]={0,0,1}, s4[3]={1,1,1};
    char *Cout[4];
    char *S[4];
    char *names[3] = {"A0","A1","Cin"};
    Cout[0] = c1;
    Cout[1] = c2;
    Cout[2] = c3;
    Cout[3] = c4;
    S[0] = s1;
    S[1] = s2;
    S[2] = s3;
    S[3] = s4;
    cout << "Cout="; QuineMckluskey(Cout,4,3,names);
    cout << endl << "S="; QuineMckluskey(S,4,3,names);
    cout << endl;
}
Chapter 3

Number Representation

3.1 Binary, Decimal and Hexadecimal Numbers

We are accustomed to using the *decimal number system*. For example the (decimal) number 34062 can be written as

\[ 34062 = 3 \cdot 10^4 + 4 \cdot 10^3 + 0 \cdot 10^2 + 6 \cdot 10^1 + 2 \cdot 10^0 \]

where \(10^1 = 10\) and \(10^0 = 1\). In general, any positive integer can be represented in one and only one way in the form

\[ a_0 \cdot 10^0 + a_1 \cdot 10^1 + a_2 \cdot 10^2 + \ldots + a_k \cdot 10^k \]

where \(0 \leq a_i \leq 9\) for \(0 \leq i \leq k\) and \(a_k > 0\). This number is denoted

\[ a_k a_{k-1} \ldots a_2 a_1 a_0 \]

in standard decimal notation.

For any integer \(r > 1\), every positive integer \(n\) can be represented uniquely in the form

\[ a_0 \cdot r^0 + a_1 \cdot r^1 + a_2 \cdot r^2 + \ldots + a_m \cdot r^m \]

where \(0 \leq a_i \leq r - 1\) for \(0 \leq i \leq m\) and \(a_m > 0\) and \(r^0 = 1\). This can be proved by induction on \(n\).

In particular, every positive integer can be represented in *binary notation*

\[ a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + \ldots + a_m \cdot 2^m \]

where \(a_i \in \{0, 1\}\) for \(0 \leq i \leq m\) and \(a_m = 1\).
Obviously we have $1 = 1 \cdot r^0$. Further if
\[ a = a_0 \cdot r^0 + a_1 \cdot r^1 + a_2 \cdot r^2 + \cdots + a_m \cdot r^m \]
where $0 \leq a_i \leq r - 1$ for $0 \leq i \leq m$ and $a_m > 0$ and $r^0 = 1$. Let $k$ be the least integer in $\{0, 1, \ldots, m\}$ with $a_k < r - 1$. Either $k < m$ which gives
\[ a + 1 = (a_k + 1)r^k + \cdots + a_m \cdot r^m, \]
or $k = 1$ which gives
\[ a + 1 = r^{m+1}. \]

**Example.** The number 23 (in decimal notation) has the binary representation 10111, since $2^4 + 2^2 + 2 + 1 = 23$. The decimal number 101 has the binary representation 1100101, i.e. $2^6 + 2^5 + 2^2 + 1$. ♠

A procedure for finding the binary representation of a number $n$ is to find the highest power $2^m$ which is $\leq n$, subtract $2^m$ from $n$, then find the highest power $2^j$ which is $\leq n - 2^m$, etc.

Although the computer operates on binary-coded data, it is often more convenient for us to view this data in *hexadecimal* (base 16). There are three reasons for this:

1. Binary machine code is usually long and difficult to assimilate. Hexadecimal, like decimal, is much easier to read.

2. There is a direct correspondence between binary and hexadecimal. Thus, we can easily translate from hexadecimal to binary.

3. In today's CPU the length of the storage elements (called *registers*) are generally multiples of 8 bits (typically 32 or 64 bits). The general purpose registers are 32 bits long or 64 bits long. Thus it is convenient to show contents as multiples and fractions of 16 – hexadecimal. The storage sizes are 8 bits (a *byte*), 16 bits (a *word*), 32 bits (a *doubleword*), 64 bits (a *quadword*), and 80 bits (a *tenbyte*) – all multiples and fractions of 16.

Thus, although we think in decimal and the computer thinks in binary, hexadecimal is a number system that captures some of the important elements of both. In the remainder of this section we discuss the binary, decimal, and hexadecimal number systems and the methods for converting from one number system to another.
3.1.1 Conversion

In this section we describe the conversion from binary to hexadecimal, from hexadecimal to binary, binary to decimal, decimal to binary, decimal to hexadecimal, and hexadecimal to decimal.

Binary to Hexadecimal. To see the one-to-one correspondence between hexadecimal and binary, notice that if we use $b$ to represent a bit and

$$b_n b_{n-1} \ldots b_2 b_1 b_0$$

is a binary number, then it has a value of

$$2^n b_n + \ldots + 2^8 b_8 + 2^7 b_7 + 2^6 b_6 + 2^5 b_5 + 2^4 b_4 + 2^3 b_3 + 2^2 b_2 + 2^1 b_1 + 2^0 b_0$$

or

$$\ldots + 256b_8 + 128b_7 + 64b_6 + 32b_5 + 16b_4 + 8b_3 + 4b_2 + 2b_1 + b_0$$

which can be written

$$\ldots + 1b_8) \cdot 16^2 + (8b_7 + 4b_6 + 2b_5 + 1b_4) \cdot 16^1 + (8b_3 + 4b_2 + 2b_1 + 1b_0) \cdot 16^0.$$  

Each of the sums in parentheses is a number between 0 (if all the $b$ values are 0) and 15 (if all the $b$ values are 1). These are exactly the digits in the hexadecimal number system. Thus to convert from binary to hexadecimal, we must gather up groups of 4 binary digits.

Example. Convert the following binary word to hexadecimal.

$$\underbrace{0010}_{2} \underbrace{1011}_{B} \underbrace{0011}_{3} \underbrace{1000}_{8} b$$

That is, $0010101100111000b = 2838h$. ♠

The notation in assembly language is as follows. The letter $b$ indicates that the number is in binary representation and the letter $h$ indicates that the number is in hexadecimal representation. This is the notation used in assembly language. The letter $d$ indicates a decimal number. The default value is decimal.

In C, C++, and Java decimal, octal and hexadecimal numbers are available. Hexadecimal numbers are indicated by 0x ... in C, C++ and Java. For example the decimal number 91 would be expressed as $0x5B$, since

$$5 \cdot 16^1 + 11 \cdot 16^0 = 91.$$
**Hexadecimal to Binary.** To convert from hexadecimal to binary, we perform the opposite process from that used to convert from binary to hexadecimal. We must expand each hexadecimal digit to four binary digits.

**Example.** Convert the hexadecimal number D0h to binary. We find

\[
\begin{align*}
&\overbrace{D}^b \overbrace{0}^b \\
\rightarrow &\quad 1101 0000 b
\end{align*}
\]

That is, \( D0h = 11010000b \). ♦

**Example.** Convert FFh to binary.

\[
\begin{align*}
&\overbrace{F}^b \overbrace{F}^b \\
\rightarrow &\quad 1111 1111 b
\end{align*}
\]

Thus \( FFh = 11111111b \). ♦

**Binary to Decimal.** Write the binary sequence in its place-value summation form and then evaluate it.

**Example.**

\[
10101010b = 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0.
\]

Thus

\[
10101010b = 2^7 + 2^5 + 2^3 + 2^1 = 128 + 32 + 8 + 2 = 170d.
\]

♦

**Decimal to Binary.** Divide the decimal number successively by 2; remainders are the coefficients of \( 2^0, 2^1, 2^2, \ldots \). We know that any non-negative integer \( a \) can be written in the form

\[
a = a_0 + a_1 \cdot 2^1 + \ldots + a_m \cdot 2^m.
\]

The technique follows simply from

\[
\sum_{i=k}^{m} a_i 2^{i-k} = 2 \sum_{i=k+1}^{m} a_i 2^{i-k-1} + a_k.
\]

The remainder after integer division by 2 gives \( a_k \), and we continue until the division gives 0. The following example illustrates this.
3.1. BINARY, DECIMAL AND HEXADECIMAL NUMBERS

Example. Convert 345\(_d\) to binary.

\[
\begin{align*}
345/2 & = 172, \text{ remainder } 1; \text{ coefficient of } 2^0 \text{ is } 1 \\
172/2 & = 86, \text{ remainder } 0; \text{ coefficient of } 2^1 \text{ is } 0 \\
86/2 & = 43, \text{ remainder } 0; \text{ coefficient of } 2^2 \text{ is } 0 \\
43/2 & = 21, \text{ remainder } 1; \text{ coefficient of } 2^3 \text{ is } 1 \\
21/2 & = 10, \text{ remainder } 1; \text{ coefficient of } 2^4 \text{ is } 1 \\
10/2 & = 5, \text{ remainder } 0; \text{ coefficient of } 2^5 \text{ is } 0 \\
5/2 & = 2, \text{ remainder } 1; \text{ coefficient of } 2^6 \text{ is } 1 \\
2/2 & = 1, \text{ remainder } 0; \text{ coefficient of } 2^7 \text{ is } 0 \\
1/2 & = 0, \text{ remainder } 1; \text{ coefficient of } 2^8 \text{ is } 1
\end{align*}
\]

Thus, 345\(_d\) = 101011001\(_b\). ♣

This method works because we want to find the coefficients \(b_0, b_1, b_2, \ldots\) (which are 0 or 1) of \(2^0, 2^1, 2^2, \ldots\) and so on. Thus, in the preceding example,

\[
345 = b_{10}2^{10} + b_{0}2^{9} + b_{8}2^{8} + \cdots + b_{1}2^{1} + b_{0}2^{0}
\]

Dividing by 2,

\[
345/2 = b_{10}2^{9} + b_{0}2^{8} + \cdots + b_{1} + (b_{0}/2)
\]

Thus \(b_0\) is the remainder on division by 2 and

\[
(b_{10}2^{9} + b_{0}2^{8} + \cdots + b_{1})
\]

is the quotient.

The following C++ program finds the binary representation of a non-negative integer. The operator \(\%\) is used to calculate the remainder after integer division.
// remain.cpp

#include <iostream.h>

void main()
{
    int i;
    unsigned long N = 345;
    unsigned long array[32];
    for(i=0;i<32;i++) { array[i] = 0; }
    for(i=0;i<32;i++) { array[i] = N%2; N = N/2; }
    for(i=31;i>=0;i--) { cout << array[i]; }
}

Decimal to Hexadecimal. Divide the decimal number successively by 16; remainders are the coefficients of $16^0, 16^1, 16^2, \ldots$

Example.

\[
\begin{align*}
302/16 &= 18, \text{ remainder 14; coefficient of } 16^0 \text{ is E} \\
18/16 &= 1, \text{ remainder 2; coefficient of } 16^1 \text{ is 2} \\
1/16 &= 0, \text{ remainder 1; coefficient of } 16^2 \text{ is 1}
\end{align*}
\]

Therefore, $302_{10} = 12E_{16}$. ♦

This works for the same reason that the method for decimal-to-binary conversion works. That is, division by 16 produces as a remainder the coefficient ($h_0$) of $16^0$, and as a quotient the decimal number minus the quantity ($h_0 \cdot 16^0$).

Hexadecimal to Decimal. Write the hexadecimal number in its place-value summation form and then evaluate.

Example.

\[ C A 1 4_h = C \cdot 16^3 + A \cdot 16^2 + 1 \cdot 16^1 + 4 \cdot 16^0. \]

Thus
\[ C A 1 4_h = 12 \cdot 4096 + 10 \cdot 256 + 1 \cdot 16 + 4 = 51732_{10}. \]

♦
3.1. BINARY, DECIMAL AND HEXADECIMAL NUMBERS

The following C++ program finds the hexadecimal representation of a non-negative integer.

// remain2.cpp

#include <iostream.h>

void main()
{
    int i;
    unsigned long N = 15947;
    unsigned char array[8];

    for(i=0;i<8;i++)
    array[i] = 0;

    for(i=0;i<8;i++)
    {
        array[i] = N%16;
        if(array[i]>9)
            array[i] += 'A'-10;
        else
            array[i] += '0';
        N = N/16;
    }

    for(i=7;i>=0;i--)
        cout << array[i];
}

The output is 00003E4B.
3.1.2 Arithmetic

The rules for addition of binary numbers are:

\[
\begin{align*}
0 + 0 &= 0 \\
0 + 1 &= 1 \\
1 + 0 &= 1 \\
1 + 1 &= (1) 0
\end{align*}
\]

where (1) denotes a carry of 1. Note that 10\text{b} is the binary equivalent of 2 decimal. Thus the sum 1 + 1 requires two bits to represent it, namely 10, the binary form of the decimal number 2. This can be expressed as follows: one plus one yields a sum bit \( s = 0 \) and a carry bit \( c = 1 \). If we ignore the carry bit and restrict the sum to the single bit \( s \), then we obtain \( 1 + 1 = 0 \). This is a very useful special form of addition known as \textit{modulo-2 addition}.

Doing arithmetic in the binary and hexadecimal number systems is best shown by examples and best learned by practice.

**Example.** Decimal Arithmetic

\[
\begin{array}{c}
45 \\
+  \; 57 \\
\hline
102
\end{array}
\]

\[\text{♣} \]
Remember that 7 + 5 is 2 with a 1 carry in decimal. 5 + 4+ the carried 1 is 0 with a 1 carry.

**Example.** Binary Arithmetic

\[
\begin{array}{c}
1011 \\
+  \; 1001 \\
\hline
10100
\end{array}
\]

\[\text{♣} \]
Remember that 1 + 1 is 0 with a 1 carry in binary.

**Example.** Binary Arithmetic

\[
\begin{array}{c}
1111 \\
+  \; 1111 \\
\hline
11110
\end{array}
\]

\[\text{♣} \]
3.1. **BINARY, DECIMAL AND HEXADECIMAL NUMBERS**

**Example.** Hexadecimal Arithmetic

\[
\begin{array}{c}
1\text{A} \\
+ \quad 5 \\
\hline
1\text{F}
\end{array}
\]

\[\text{♣}\]

In decimal $1 \text{A} + 5$ is $10 + 5 = 15$. Thus $15d = Fh$.

**Example.** Hexadecimal Arithmetic

\[
\begin{array}{c}
\text{FF} \\
+ \quad 3 \\
\hline
102
\end{array}
\]

\[\text{♣}\]

$F + 3$ is $15 + 3$ in decimal. Thus $18d = 12h$. Thus we write down a 2, carry a 1.

*Binary multiplication* can be done by repeated addition. The following example shows how to multiply the two binary numbers $1001$ (9 decimal) and $110$ (6 decimal).

\[
\begin{array}{c}
1001 \text{ multiplicand} \\
\times \quad 110 \text{ multiplier} \\
\hline
0000 \text{ first partial product} \\
1001 \quad \text{second partial product} \\
1001 \quad \text{third partial product} \\
\hline
110110 \text{ product (54 decimal)}
\end{array}
\]

High speed multiplication techniques use addition and subtraction or uniform multiple shifts. *Binary divisions* can be performed by a series of subtractions and shifts.
3.1.3 Signed Integers

We can easily see how positive integers are stored. For example, 345 is stored as
101011001. This will not fit into a byte because it has more than 8 bits, but it fits
into a word (2 consecutive bytes). A byte has 8 bits, a word has 16 bits and a double
word has 32 bits.

**Example.** Show $65712_d$ as a binary (a) byte, (b) word, (c) double word. We have

$$65712_d = 10000000010110000_b$$

Thus

(a) Does not fit in a byte (it is too large).

(b) Does not fit in a word (it is too large).

(c) $0000000000000000100000010110000$

The largest integer number which fits into a register with 32 bits is

$$2^{32} - 1 \equiv \sum_{k=0}^{31} 2^k = 4294967295.$$  

The largest integer number which fits into a register with 64 bits is

$$2^{64} - 1 \equiv \sum_{k=0}^{31} 2^k = 18446744073709551615.$$  

Storing negative integers presents a more difficult problem since the negative sign
has to be represented (by a 0 or a 1) or some indication has to be made (in binary!) that
the number is negative. There have been many interesting and ingenious ways
invented to represent negative numbers in binary. We discuss three of these here:

1. Sign and magnitude

2. One’s complement

3. Two’s complement
3.1. BINARY, DECIMAL AND HEXADECIMAL NUMBERS

Sign and Magnitude

The sign and magnitude representation is the simplest method to implement negative integers. Knuth [112] used sign and magnitude in his mythical MIX computer. In sign and magnitude representation of signed numbers, the leftmost (most significant) bit represents the sign:

\[ 0 \text{ for positive} \]

and

\[ 1 \text{ for negative.} \]

**Example.** The positive integer number 31 stored in a double word (32 bits) using sign and magnitude representations is

\[ 00000000000000000000000011111b \]

Thus the negative integer −31 becomes

\[ 1000000000000000000000011111b \]

There are two drawbacks to sign and magnitude representation of signed numbers:

1. There are two representations of 0:

\[ +0 = 0000000000000000000000000000b \]

and

\[ -0 = 1000000000000000000000000000b. \]

Thus the CPU has to make two checks every time it tests for 0. Checks for 0 are done frequently, and it is inefficient to make two such checks.

2. Obviously,

\[ a + (-b) \]

is not the same as

\[ a - b. \]

What this means is that the logic designer must build separate circuits for subtracting; the adding circuit used for \( a + b \) is not sufficient for calculating \( a - b \).

**Example.** The following shows that

\[ 52 - 31 \]

and

\[ 52 + (-31) \]

are not the same in sign and magnitude representation.
\[ 52 = \quad 00000000 \quad 00000000 \quad 00000000 \quad 00110100b \]
\[-31 = -00000000 \quad 00000000 \quad 00000000 \quad 00011111b \]

\[ \begin{array}{c}
\hline \\
21 = & 00000000 \quad 00000000 \quad 00000000 \quad 00010101b \\
\hline
\end{array} \]

On the other hand

\[ 52 = \quad 00000000 \quad 00000000 \quad 00000000 \quad 00110100b \]
\[ + \quad -31 = \quad 10000000 \quad 00000000 \quad 00000000 \quad 00011111b \]

\[ \begin{array}{c}
\hline \\
21 = & 10000000 \quad 00000000 \quad 00000000 \quad 01010011b \\
\hline
\end{array} \]

Thus this shows that the sign and magnitude representation is not useful for implementations on CPU’s.

Furthermore \(31 - 31\) gives

\[ \begin{array}{c}
\hline \\
31 = & 00000000 \quad 00000000 \quad 00000000 \quad 00011111b \\
-31 = & -00000000 \quad 00000000 \quad 00000000 \quad 00011111b \\
\hline
0 = & 00000000 \quad 00000000 \quad 00000000 \quad 00000000b \\
\hline
\end{array} \]

and \(31 + (-31)\) gives

\[ \begin{array}{c}
\hline \\
31 = & 00000000 \quad 00000000 \quad 00000000 \quad 00011111b \\
+ \quad -31 = \quad 10000000 \quad 00000000 \quad 00000000 \quad 00011111b \]

\[ \begin{array}{c}
\hline \\
0 = & 10000000 \quad 00000000 \quad 00000000 \quad 00111110b \\
\hline
\end{array} \]
One’s Complement

One’s complement method of storing signed integers was used in computers more in the past than it is currently. Here we assume again that 32 bits are given (double word). In one’s complement, the leftmost bit is still 0 if the integer is positive. For example,

\[00000000000000000000000001111b\]

still represents +31 in binary. To represent the negative of this, however, we replace all 0’s with 1’s and all 1’s with 0’s. Thus

\[1111111111111111111111110000b\]

represents –31. Note that the leftmost bit is again 1. Notice that in assembly language one starts counting from zero from the rightmost bit.

Example. Using a double word of storage, –1 is stored as

\[11111111111111111111111110b\]

since 1 is stored in binary as

\[00000000000000000000000001b\]

Thus the second drawback to sign and magnitude representation has been eliminated. This means \(a - b\) is the same as \(a + (-b)\). Thus the circuit designer need only include an adder; it can also be used for subtraction by replacing all subtractions \(a - b\) with \(a + (-b)\).

The following example shows, however, that this adder must do a little more than just add.

Example. We show that \(52 - 31\) and \(52 + (-31)\) are the same in one’s complement representation. For \(52 - 31\) we have

\[
\begin{align*}
52 &= 00000000 \ 00000000 \ 00000000 \ 00110100b \\
-31 &= -00000000 \ 00000000 \ 00000000 \ 00011111b \\
\hline
21 &= 00000000 \ 00000000 \ 00000000 \ 00010101b
\end{align*}
\]

Next we consider the one’s complement. Since

\[31 = 00000000 \ 00000000 \ 00000000 \ 00011111b\]
we find for the one's complement

\[-31 = 11111111 11111111 11111111 11100000b\]

Addition of the two terms \(52 + (-31)\) yields

\[00000000 00000000 00000000 00010100b\]

plus an overflow bit. Addition of the overflow bit to the right-most bit yields

\[00000000 00000000 00000000 00010101b\]

which is 21 in binary representation.

The adder for one's complement arithmetic is more complicated; it must carry around any overflow bit in order to work correctly for subtraction. The first drawback is still with us, however. In one's complement, there are still two representations of 0

\[00000000 00000000 00000000 00000000b \text{ positive 0}\]

and

\[11111111 11111111 11111111 11111111b \text{ negative 0}\]

when viewed as a double word.

One's complement is implemented in C, C++ and Java with the \(-\) operator. The following program shows an application.

```cpp
// complement.cpp

#include <iostream.h>

char *binary(unsigned int N)
{
    static char array[36];

    for(int i=34,j=27;i>=0;i--)
        if((i == j) && (i != 0)) { array[i]=','; j=-9;}
        else { array[i] = N%2 + '0'; N = N/2; }
    array[35]='
';
    return array;
}

void main()
{
    int a = 17; // binary 000000000 00000000 00000000 0010001
    cout << "a = " << a << endl << binary(a) << endl;
    int b = ~a; // binary 11111111 11111111 11111111 1101110
    cout << "-a = " << b << endl << binary(b) << endl;
}
Two’s Complement

The two’s complement method of storing signed integers is used in most present-day CPU’s, including the 386, 486, Pentium and Alpha Dec. The two’s complement is formed by

(1) forming the one’s complement and then

(2) adding 1.

Example. Using two’s complement and a double word (32 bits). Thus the decimal number 31 is stored as

$$00000000000000000000000011111b.$$ 

Consequently −31 is stored as

$$1111111111111111111111100001b.$$ 

We can easily check that

$$a - b = a + (-b)$$

and that there is only one way to represent 0, i.e., +0 and −0 are stored the same, namely

$$00000000000000000000000000000000b.$$ 

The two’s complement of a number is the true (one’s) complement of the number plus 1.

With $n$ bits we can represent numbers from

$$-2^{n-1} \text{ to } 2^{n-1} - 1$$

in two’s complement. If we have registers with 32 bits then we can store the integer numbers ($n = 32$)

$$-2147483648 \text{ to } 2147483647.$$ 

Although taking the two’s complement of a number is more difficult than taking its one’s complement, addition of two’s complement numbers is simpler than addition in one’s complement or in signed-magnitude representations.

Next we consider some examples of addition in two’s complement. We assume that 32 bits are given.
Addition of Two Positive Numbers

+3 = 00000000 00000000 00000000 00000011b
+4 = 00000000 00000000 00000000 00000100b

+7 = 00000000 00000000 00000000 00000111b

Addition of Two Negative Numbers

-4 = 11111111 11111111 11111111 11111100b
-1 = 11111111 11111111 11111111 11111111b

-5 = 11111111 11111111 11111111 11111011b

Addition of One positive and One Negative Number

-7 = 11111111 11111111 11111111 11111001b
+5 = 00000000 00000000 00000000 00000101b

-2 = 11111111 11111111 11111111 11111100b

In two’s complement, it is possible to add or subtract signed numbers, regardless of the sign. Using the usual rules of binary addition, the result comes out correctly, including the sign. The carry is ignored. This is a very significant advantage. If this were not the case, we would have to correct the result for sign every time, causing a much slower addition or subtraction time. For the sake of completeness, let us state that two’s complement is simply the most convenient representation to use for microprocessors. All signed integers will be implicitly represented internally in two’s complement notation.

The following Java program shows and implementation of two’s complement.

// Twocomp.java

public class Twocomp
{
    public static void main(String[] args)
    {
        int r1 = 14;       // binary 1110

        // two’s complement to find the negative number
        // of a given integer number
        // The operation - gives the one complement
        // and then we add 1 to find the two’s complement
        int r2 = ~r1; r2++;        
        System.out.println(r2); // => -14
    }
}

3.1.4 Overflow

If we do arithmetic operations with 32 bit registers overflow will occur in cases:

1. if we go beyond the range 0 - 4294967295 for the data type unsigned long in C and C++. This means we add numbers so that the sum is larger than 4294967295. Also negative numbers are out of range.

2. if we go out of the range $-2^{31}$ to $2^{31}-1$ (long in C and C++). This means if we add or subtract numbers which go beyond this range.

Example. Consider the sum

$$4294967295 + 1 = 4294967296$$

The number on the right-hand side is out of the range for a 32 bit register for the C and C++ data type unsigned long. Since

$$4294967295 = 11111111111111111111111111111111b$$

for unsigned long, the addition of 1 yields

$$00000000000000000000000000b$$

with one overflow bit. Thus the output is 0. ☺

Example. Consider the sum

$$+(-2147483648) + (-3) = -2147483651$$

The number on the right-hand side is out of range for a 32 bit register for long. Since

$$-2147483648 = 10000000000000000000000000000000b$$

and

$$-3 = 1111111111111111111111111101b$$

we obtain

$$0111111111111111111111111101b$$

Thus the output is 2147483651. ☺
Consider the following C++ program.

```c++
// overflow.cpp

#include <iostream>

int main()
{
    unsigned long a = 4294967295;
    unsigned long b = 1;
    unsigned long r1;
    r1 = a + b;
    cout << "r1 = " << r1 << endl; // 0

    unsigned long c = 4294967295;
    unsigned long d = 2;
    unsigned long r2;
    r2 = c + d;
    cout << "r2 = " << r2 << endl; // 1

    unsigned long e = 0;
    unsigned long f = -1;
    unsigned long r3 = e + f;
    r3 = e + f;
    cout << "r3 = " << r3 << endl; // 4294967295

    long g = -2147483648;
    long h = -3;
    long r4 = g + h;
    cout << "r4 = " << r4 << endl; // 2147483645

    unsigned long i = 0xFFFFFFFF; // hexadecimal number
    unsigned long j = 0x1; // hexadecimal number
    unsigned long r5 = i + j;
    cout << "r5 = " << r5 << endl; // 0

    return 0;
}
```
The range of the data type `unsigned long` is 0 - 4294967295. The binary representation of 4294967295 is

\[11111111 \ 11111111 \ 11111111 \ 11111111_{b}\]

This is the largest number which fits into 32 bits. Thus if we add 1 to this binary number under the assumption that 32 bits are given we find

\[00000000 \ 00000000 \ 00000000 \ 00000000_{b}\]

with 1 overflow bit. The output of the C++ program ignores the overflow bit and displays the output 0.

*Remark.* Obviously the overflow flag of the CPU will be set.

Analogously we understand the other three outputs,

\[r_2 = 1\]

\[r_3 = 4294967295\]

and

\[r_4 = 2147483645.\]

of the program.

Java has the signed data type `long`. The size is 64 bits. Thus the range is

\[-9223372036854775808 \text{ to } 9223372036854775807.\]
3.1.5 Binary-Coded Decimal Form

The discussion so far has assumed that decimal numbers are translated into base-2 form for processing by digital circuits. An alternative approach is to encode the decimal digits into binary form, but maintain the base-10 positional notation in which all digits are weighted by powers of 10. Such numbers are called binary-coded decimal numbers or, if the context is clear, simply decimal numbers. We usually restrict the term binary-coded decimal, which is abbreviated BCD, to refer to the most widely used code of this sort.

An unsigned decimal number

\[ N_{10} = d_{n-1}d_{n-2} \ldots d_1d_0 \]

is converted into the standard BCD form by mapping each digit \( d_i \) separately into a 4-bit binary number

\[ B_i = b_{i3}b_{i2}b_{i1}b_{i0} \]

where \( (B_i)_2 = (d_i)_{10} \). Thus, a 9 in \( N_{10} \) is mapped into 1001, an 8 into 1000, a 7 into 0111, and so on. For example, if \( N_{10} = 7109_{10} \), then the decimal-to-BCD conversion process takes the form

\[ N_{10} = \begin{array}{cccc}
7 & 1 & 0 & 9 \\
0111 & 0001 & 0000 & 1001
\end{array} \]

leading to

\[ N_{10} = 0111000100001001_{10} \]

where the underlined subscript \( _{10} \) is our notation for binary-coded decimal. This conversion process is, in fact, the same as that used for changing a hexadecimal number to binary. In this case, there are 10 digits instead of 16, so only 10 of the 16 possible 4-bit binary numbers are needed. Also, each 4-bit group must be assigned weight 10 rather than 16. For example, we get

\[ N = 0111_2 \cdot 10^3 + 0001_2 \cdot 10^2 + 0000_2 \cdot 10^1 + 1001_2 \cdot 10^0 \]

assuming \( N \) as an integer. The weight of an individual bit in \( N_{10} \) is of the form \( j \cdot 10^i \), where \( j \) is 8, 4, 2, or 1. Standard BCD is therefore sometimes called 8421 decimal code.

Conversion from BCD to ordinary decimal form is achieved by replacing 4-bit groups with the equivalent decimal digit. For instance,

\[ N'_{10} = 0010 \ 1000 \ 0100 \ 1001 \ 0000 \ 0101 \]

\[ N'_{10} = 2 \ 8 \ 4 \ 9 \ 0 \ 5 \]
3.1. BINARY, DECIMAL AND HEXADECIMAL NUMBERS

implying that \( N_{10}' = 284905_{10} \). Conversion between binary (base 2) and BCD requires the decimal-binary conversion procedure, in addition to the decimal digit-encoding procedure discussed above.

Not all the possible binary patterns correspond to BCD numbers. The six 4-bit patterns

\[
\{ 1010, 1011, 1100, 1101, 1110, 1111 \}
\]

are not needed to represent decimal digits, and therefore cannot appear in the digit positions of a BCD number. These six digit patterns appear in BCD numbers only as the result of an error, such as failure of a hardware component or a programming mistake. The fact that some \( n \)-bit patterns are unused implies that a larger \( n \) is needed to represent a given range of numbers if BCD code is employed in place of binary. For example, to represent all integers from zero to a million requires 24 bits of BCD code, but only 20 bits of binary code. Note that \( 2^{20} > 10^6 \). Therefore, BCD numbers have slightly greater storage requirements than binary numbers. Arithmetic operations on BCD numbers are more complicated than their binary counterparts. The main advantage of BCD code is that it eliminates most of the need for time-consuming base-10-to-base-2 and base-2-to-base-10 conversions. Digital computers are designed primarily to process binary numbers, but many have features to support BCD operations.

The following C++ program converts a number \( (N = 15947) \) to BCD form.

```cpp
// bcd.cpp

#include <iostream.h>

void main()
{
    int i;
    unsigned long N = 15947;
    unsigned char array[4];
    unsigned char mask[2]={0x0F, 0x0F};
    unsigned char shift[2]={0, 4};
    for(i=0; i<4; i++)
        array[i] = 0;
    for(i=0; i<8; i++)
        { array[i/2] |= (N%10)<<shift[i/2]; N = N/10; }
    for(i=7; i>=0; i--)
        { cout << char(((array[i/2]&mask[i/2])>>shift[i/2])\'0\'); }
}
```
3.2 Floating Point Representation

3.2.1 Introduction

We have seen how to store integer numbers in bit sequences, and in Chapter 2 an example illustrated a similar method for representing real numbers on a certain interval with a guaranteed accuracy. The interval was divided into equal subintervals according to the number of bit string combinations allowed. Certain real numbers can be converted without loss of accuracy. These are the numbers which are boundaries for the subintervals. The other real numbers will be encoded, with error, to one of these numbers. This is the fixed point method of storing real numbers. The maximum accuracy of the fractions is uniquely specified by the size of the subintervals. Thus we can identify a fractional part and an integer part giving a fixed separation of bits representing the integer and fractional parts respectively. Traditionally this separation is indicated by a decimal point in the symbolic representation. Floating point number representations use unequal subintervals to represent real numbers. For example small numbers require better accuracy in calculations and thus are represented with smaller intervals. Larger numbers use larger intervals. To compromise we may require that the ratio of the error in representation to the number being represented be approximately constant.

Storing floating-point numbers presents a problem similar to that of storing signed integers. For integers, some indication of a positive or negative sign has to be represented. For floating-point instructions some method must be devised for showing where the decimal point should go. That is, we must distinguish between the fractional part to the right of the decimal point – called the mantissa – and the integer portion to the left of the decimal point. Different methods have been used in the past and different methods continue to be used by the various manufacturers of computers. There have been so many different ways of coding a floating-point number into binary that the Institute of Electrical and Electronics Engineers (IEEE) has proposed a standard format.

There are actually three formats – one that requires 32 bits, one that is used for 64 bits, and one for 80 bits. We describe the 32-bit format, called the short real format, here.

The table lists seven numeric data types showing the data format for each type. The table also shows the approximate range of normalized values that can be represented with each type. Denormal values are also supported in each of the real types, as required by IEEE Std 854.
Table: Numeric Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Bits</th>
<th>Significant Digits (Decimal)</th>
<th>Approximate Normalized Range (Decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Integer</td>
<td>16</td>
<td>4</td>
<td>(-32768 \leq x \leq +32767)</td>
</tr>
<tr>
<td>Short Integer</td>
<td>32</td>
<td>9</td>
<td>(-2 \times 10^9 \leq x \leq +2 \times 10^9)</td>
</tr>
<tr>
<td>Long Integer</td>
<td>64</td>
<td>18</td>
<td>(-9 \times 10^{18} \leq x \leq +9 \times 10^{18})</td>
</tr>
<tr>
<td>Packed Decimal</td>
<td>80</td>
<td>18</td>
<td>(-99 \ldots 99 \leq x \leq +99 \ldots 99) (18 digits)</td>
</tr>
<tr>
<td>Single Real</td>
<td>32</td>
<td>7</td>
<td>(1.18 \times 10^{-38} &lt;</td>
</tr>
<tr>
<td>Double Real</td>
<td>64</td>
<td>15-16</td>
<td>(2.23 \times 10^{-308} &lt;</td>
</tr>
<tr>
<td>Extended Real</td>
<td>80</td>
<td>19</td>
<td>(3.37 \times 10^{-4932} &lt;</td>
</tr>
</tbody>
</table>

All operands are stored in memory with the least significant digits starting at the initial (lowest) memory address. Numeric instructions access and store memory operands using only this initial address.
3.2.2 Representation

The first step to understanding how a binary fraction is stored using short real format is to normalize it. This is similar to putting a decimal point number into the familiar scientific notation in which we have a sign, an exponent, and a mantissa. To normalize a binary fraction, we write it so that the first 1 is just to the left of the binary point.

Example. Consider the binary number

\[
0.000111101 = 0 \cdot \frac{1}{2} + 0 \cdot \frac{1}{2^2} + 0 \cdot \frac{1}{2^3} + 1 \cdot \frac{1}{2^4} + 1 \cdot \frac{1}{2^5} + 1 \cdot \frac{1}{2^6} + 0 \cdot \frac{1}{2^7} + 1 \cdot \frac{1}{2^8} + 1 \cdot \frac{1}{2^9}
\]

Then the normalized representation is

\[
1.11101 \cdot 2^{-4} \quad \bullet
\]

The next step is to represent the important parts of the normalized fraction in 32 bits. The important parts are those that will allow us to recover the original number (and allow the computer to perform operations on it). These parts are the

1. Sign
2. Exponent (whose base is understood to be 2)
3. Mantissa

In the IEEE short real format, the sign is stored in the leftmost bit, the exponent is stored in the next 8-bits, after some alteration, and the mantissa is stored in the rightmost 23 bits, again after a minor adjustment.

1. To store the sign. 0 for positive, 1 for negative.
2. To store the exponent. Add 127 (11111112) to it. The number 127 is called a bias, and the resulting exponent is called a biased exponent. Biased exponents may range from 1 to 254, so that exponents range from −126 to +128.
3. To store the mantissa. Remove the leftmost 1 and store the rest of the fraction left-adjusted. This technique of not storing the first 1 before the binary point is a common way to store mantissas. It is called hidden bit storage. Computer circuitry knows that the 1 is really part of the mantissa.
3.2. FLOATING POINT REPRESENTATION

Example. Find 0.0390625 (base 10) as it would be stored in short real format.

Step 1. Convert the fraction to binary

\[0.0390625_{10} = .000101_2\]

Step 2. Normalize the binary fraction.

\[.000101\text{ normalized is } 1.01 \cdot 2^{-5}\]

Step 3. Calculate the sign, the exponent, and the mantissa.

Sign: 0, since this is a positive number

Exponent: \(-5 + 127 = 122\) (base 10) = 0111010 (base 2)

Mantissa: .01 left-adjusted into a field of width 23 is

\[.01000000000000000000000\]

Thus the entire number is represented by the bitstring

\[
\begin{array}{c}
\text{Sign} \\
0
\end{array}
\begin{array}{c}
\text{Exponent} \\
0111010
\end{array}
\begin{array}{c}
\text{Fraction} \\
01000000000000000000000
\end{array}
\]

The following C++ program implements this algorithm. The only difference is that the actual conversion to binary is delayed until after the normalization procedure. We use the above test example. For the output we find

\[0.0390625 \text{ (base 10)} =
0.0111010 \ 01000000000000000000000 \text{ (floating point base 2)}\]
// float2bin.cpp

#include <iostream.h>
#include <math.h>

void normalize(float &f, char &e)
{
    e = 0;

    // numbers larger than 2 we reduce
    // down to 1 plus a fraction,
    // and a compensating exponent
    while(fabs(f) > 2)
    {
        f /= 2;
        e++;
    }

    // numbers smaller than 1 we promote
    // up to 1 plus a fraction,
    // and a compensating exponent
    while(fabs(f) < 1)
    {
        f *= 2;
        e--;
    }
}

void float2bin(float f, char *b)
{
    char e1;
    int e;
    int i;

    normalize(f,e1);

    // add the bias
    e = int(e1) + 127;

    // set the sign bit
    b[0] = (f < 0) ? '1': '0';

    f = fabs(f);

    // remove the leftmost 1 bit
3.2. FLOATING POINT REPRESENTATION

```c
f -= 1;


// convert the exponent
for(i=8;i>0;i--)
{
    b[i+1] = e%2 + ’0’;
    e/=2;
}

// convert the mantissa
for(i=1;i<24;i++)
{
    int bit = (f>=pow(2,-i));

    b[i+10] = (bit) ? ’1’ : ’0’;
    if (bit) f -= pow(2,-i);
}

b[34]=’\0’;
}

void main(void){
    char b[35];
    float f=0.0390625;

    float2bin(f,b);
    cout << f << " (base 10) = 
        << b << " (floating point base 2)"<<endl;
}
```
**Example.** What number is stored as

\[101111101111101000000000000000000\]?

We recover the parts as

\[1|01111110111010000000000000000000\]

Sign: 1, so the number is negative

Exponent: \(0111110_2 = 125_{10}\), \(125 - 127 = -2\)

Mantissa: affixing 1 to the left of

\[.111010000000000000000000\]

results in

\[1.111010000000000000000000\]

which is

\[1.11101_2.\]

Multiplying by \(2^{-2}\) (provided by the exponent) yields

\[.0111101_2 = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{128} = -0.4765625_{10} \] ✧
Chapter 4

Logic Gates

4.1 Introduction

A digital electronic system uses a building-block approach. Many small operational units are interconnected to make up the overall system. The system’s most basic unit is the gate circuit. These circuits have one output and one or more inputs. The most basic description of operation is given by the function table, which lists all possible combinations of inputs along with the resulting output in terms of voltage, high and low. Table 4.1(a) shows a function table for a 2-input circuit. This table indicates that if both inputs are low or both are high, the output will be low. If one input is high and the other is low, a high level will result on the output line. As we deal with logic design, it is appropriate to use 1s and 0s rather than voltage levels. Thus, we must choose a positive \((H = 1, \ L = 0)\) or negative \((H = 0, \ L = 1)\) logic scheme. Once this choice is made, we use the function table to generate a truth table. The function table describes inputs and outputs in terms of 1s and 0s rather than voltage levels. Function tables are used by manufacturers of logic gates to specify gate operation. The manufacturer conventionally defines gates in terms of positive logic.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_1)</td>
<td>(A_2)</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

\(L=\text{low voltage level}\)
\(H=\text{high voltage level}\)

 positive logic

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_1)</td>
<td>(A_2)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

 negative logic

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_1)</td>
<td>(A_2)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Function Table and Truth Tables for a Logic Circuit
4.2 Gates

4.2.1 AND Gate

The AND gate has one output and two or more inputs. The output will equal 0 for all combinations of input values except when all inputs equal 1. When each input is 1, the output will also equal 1. Figure 4.1 shows the AND gate. Table 4.2 shows the function and positive logic truth tables. The AND gate will function as an OR gate for negative logic, but the gate is named for its positive logic function.

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.2: Truth Table for the AND Gate

\[
A_1 \quad \& \\
A_2 \quad X = A_1 \cdot A_2
\]

Figure 4.1: Symbol for 2-input AND Gate

The AND operation can be interpreted as the multiplication of a set of 1 bit numbers; a 0 among the input variables makes the result (product) 0; the product is 1 if and only if all the inputs are 1. For this reason the AND function is written as a product expression

\[
X_{\text{AND}} := A_1 \cdot A_2
\]

or

\[
X_{\text{AND}} := A_1 \cdot \ldots \cdot A_n
\]

if we have $n$ inputs. Alternative AND symbols in common use are $\land$ and $\&$. The latter is the AND designator in the standard box symbol for an AND gate. As with multiplication the symbol $\cdot$ is sometimes omitted from AND expressions, so that $A_1 \cdot A_2$ reduces to $A_1A_2$.

In CMOS the 4081 provides quad two-input AND gates.
4.2. GATES

4.2.2 OR Gate

The OR gate has one output and two or more inputs. If all inputs are equal to 0, the output will be equal to 0. The presence of a 1 bit leads to an output of 1. Table 4.3 describes this operation in terms of a truth table. The standard symbol for a 2-input OR gate is shown in Figure 4.2.

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.3: Truth Table for the OR Gate

$$ \begin{array}{c|c|c}
  A_1 & A_2 & X \\
  \hline
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
\end{array} $$

Figure 4.2: Symbol for 2-input OR Gate

The OR operation takes its name from the fact that the output $X$ is 1 if and only if $A_1$ is 1 or $A_2$ is 1. In other words, the output $X$ of an OR gate is 1 if and only if the number of 1s applied as input is one or greater.

We can have more than 2 input lines. The $X$ is 1 if and only if $A_1$ is 1 or $A_2$ is 1 or ... $A_n$ is 1. This interpretation leads to the use of the symbol $\geq 1$ in the OR box. By a somewhat weak analogy with numerical addition, the OR function is usually written as a sum expression

$$ X_{OR}(A_1, A_2, \ldots, A_n) := A_1 + A_2 + \ldots + A_n. $$

Thus, $+$ denotes OR in this context, and is read as "or" rather than plus. An alternative OR symbol is $\lor$.

In CMOS 4071 is a quad two-input gate and the CMOS 4072 is a two quad-input gate.
4.2.3 XOR Gate

If the exclusive OR gate (XOR gate) has two inputs, then the output gives a 1 when either input is 1, but not when both are 1. If the input is \( A_1 = 0 \) and \( A_2 = 0 \), then the output is 0. Table 4.4 shows the truth table for the XOR gate.

<table>
<thead>
<tr>
<th>( A_1 )</th>
<th>( A_2 )</th>
<th>( X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.4: Truth Table for the XOR Gate

The generalization of XOR to \( n \) input variables is most easily specified in terms of the parity of the number of 1’s among the \( n \) input variables

\[
X_{\text{XOR}}(A_1, A_2, \ldots, A_n) := \begin{cases} 
1 & \text{if an odd number of inputs are 1} \\
0 & \text{otherwise}
\end{cases}
\]

For this reason, XOR is also called the *odd-parity function*, and is the basis of error-handling circuits. This versatile function can also be interpreted as (numerical) summation modulo 2. Thus, another definition of XOR equivalent to the definition given above is

\[
X_{\text{XOR}}(A_1, A_2, \ldots, A_n) := A_1 + A_2 + \ldots + A_n \pmod{2}.
\]

The XOR gate is a special gate and is widely employed in digital circuits that perform mathematical functions.

The symbol for the XOR gate are shown in the next figure.

\[
\begin{array}{c}
A_1 \\
A_2 \\
2k+1 \\
\hline
X \end{array}
= A_1 \oplus A_2
\]

Figure 4.3: Symbol for 2-input XOR Gate

The use of the generic odd number \( 2k + 1 \) as the function designator in the standard box symbol reflects the fact that the output is 1 if and only if \( 2k + 1 \) inputs are 1, for \( k = 0, 1, 2, \ldots \). In logic expressions, the XOR operator is \( \oplus \) which is read as *exclusive OR*, ring-sum, or sum modulo 2. Thus, we can write

\[ X_{\text{XOR}} = A_1 \oplus A_2 \oplus \ldots \oplus A_n \]

The CMOS 4030 is a quad two-input exclusive OR gate.
4.2.4 NOT Gate (Inverter)

The *inverter* (also called NOT gate) performs the NOT or INVERT function and has one output and one input. The output level is always opposite to the input level. Thus an inverter converts a 0 to 1 and a 1 to 0, an operation known variously as inversion, complementation, or the NOT function. NOT is denoted by an overbar in functional expressions and by a small circle, the inversion symbol, in circuit diagrams. We write

\[ X_{\text{NOT}} = \overline{A}. \]

Table 4.5 shows the truth table for the inverter. Figure 4.4 shows the symbol for the inverter.

<table>
<thead>
<tr>
<th>( A )</th>
<th>( X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.5: Truth Table for the NOT Gate

![Figure 4.4: Symbol for the NOT Gate](image)

In CMOS the 4069 is a hex inverter. Each of the six inverters is a single stage.

The NOT gate can be combined with the AND, OR and XOR gate to provide the NAND, NOR and XNOR gate.
4.2.5 NAND Gate

The NAND gate is an AND gate followed by an inverter. A NAND gate can have two or more inputs. Thus the NAND gate is formed by appending NOT to AND. The output will be 0 only when all inputs are 1. Its logic expression is

\[ X = A_1 \cdot A_2 \cdots \cdot A_n \]

which indicates that the inputs \( A_1, A_2, \ldots, A_n \) are first ANDed and then the result inverted. Thus a NAND gate always produces an output that is the inverse (opposite) to an AND gate. The gate symbol is therefore formed by appending the graphic inversion symbol (a small circle) to the corresponding AND symbol.

<table>
<thead>
<tr>
<th>( A_1 )</th>
<th>( A_2 )</th>
<th>( X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.6: Truth Table for the NAND Gate

![Figure 4.5: Symbol for 2-input NAND Gate](image)

Figure 4.5: Symbol for 2-input NAND Gate

Since both inverters and AND gates can be constructed from NAND gates, the NAND gate is seen to be a functionally complete set itself. The AND gate and inverter form a functionally complete set. This means that any logic function realized by logic gates can be realized with the AND and NOT functions. For example the XOR gate can be represented by

![Figure 4.6: XOR Implemented With NAND Gates](image)

Figure 4.6: XOR Implemented With NAND Gates

In CMOS the 4011 provides a quad two-input NAND gate.
4.2.6 NOR Gate

The NOR gate is an OR gate followed by an inverter. The NOR gate can have two or more inputs. Thus the NOR gate combines the OR and NOT operations such that the output will be 0 when any input is 1. Its logical expression is

\[ X = A_1 + A_2 + \ldots + A_n \]

which indicates that \( A_1, A_2, \ldots, A_n \) are first ORed and then the result is inverted. A NOR gate always gives an output that is the inverse of the OR gate. The gate is characterized by the tables and symbols of Table 4.7 and Figure 4.7.

<table>
<thead>
<tr>
<th>( A_1 )</th>
<th>( A_2 )</th>
<th>( X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.7: Truth Table for the NOR Gate

![Figure 4.7: Symbol for 2-input NOR Gate](image)

All other gates can be constructed from NOR gates. For example, the XOR gate can be found as

![Figure 4.8: XOR Implemented With NOR Gates](image)

In CMOS the 4001B is a quad two-input NOR gate.
4.2.7 XNOR Gate

The exclusive-NOR or XNOR gate produces a 1 output only when the inputs are at the same logic level. The exclusive-NOR gate is also known as the even-parity function for obvious reasons. The truth table is given in Table 4.8

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.8: Truth Table for the XNOR Gate

\[
A_1 \quad 2k+1 \quad X = A_1 \oplus A_2
\]

Figure 4.9: Symbol for 2-input XNOR Gate

The XNOR gate is not a universal gate.

In CMOS the 4077 provides the quadruple exclusive-NOR gate.
4.3 Buffer

The buffer is an IC device that provides no change in logic at the output, but does provide a high input load impedance, and therefore good output drive capability. It works the same way as an emitter-follower circuit. The output of a MOS microprocessor, for example, has very poor drive capability when driving a TTL device. By inserting a buffer between the output of the MOS microprocessor and the input of the TTL device, we can solve the problem. The buffer provides an input load the processor can handle and an output drive that is TTL-compatible. The truth table and the symbol for a buffer are shown in Table 4.9 and Figure 4.10.

<table>
<thead>
<tr>
<th>A</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.9: Truth Table for the Buffer

![Figure 4.10: Symbol for the Buffer](image)

As an example consider the buffering of MPU buses. The MPU, RAM and ROM are chips that are generally manufactured using CMOS technology. The decoders, gates, inverters, tri-state buffers, and output register are all TTL devices, usually LS-TTL to minimize power requirements and loading.

In CMOS the 4041B is a quadruple true/complement buffer which provides both an inverted active LOW output and a non-inverted active HIGH output (O) for each input (I).
4.4 Tri-State Logic

The development of bus organized computers led to the development of a type of logic circuitry that has three distinct output states. These devices, called tri-state logic (TSL) devices, have a third output condition in addition to the normal HIGH and LOW logic voltage levels. This third output condition is called the high-impedence, or high-Z state. Thus tri-state circuits have the high impedance state Z as a normal output variable in addition to the usual 0 and 1 values. We can convert any logic circuit C to tri-state form simply by inserting a switch S in its output line \( X \). The control input signal \( E \) of \( S \) is called an enable signal if \( E = 1 \) makes \( X = Y \), and \( E = 0 \) makes \( X = Z \). Otherwise it is a disable signal. When \( E = 1 \) the output \( X \) is said to be enabled, and assumes its normal 0-1 levels determined by \( C \). The ENABLE input determines the output operation so that the output either acts as a normal TTL output (ENABLE=1) or as a high-Z output (ENABLE=0). In the enabled condition, the circuit behaves exactly as any logic buffer gate, producing an output voltage level equivalent to the input logic level. In the disabled high-Z state, the output terminal acts as if it were disconnected from the buffer; in other words, think of it as a virtual open circuit.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>( E )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>( E )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.11: (a) A tri-state inverter with an enable line, (b) a tri-state buffer with a disable line

Tri-state buffers are often used in applications where several logic signals are to be connected to a common line called a bus. Many types of logic circuits are currently available with tri-state outputs. Other tri-state circuits include flip-flops, registers, memories, and almost all microprocessors and microprocessor interface chips. In CMOS the 40097 is a hex non-inverting buffer with 3-state outputs. The 3-state outputs are controlled by two enable inputs.
4.5 Feedback and Gates

Any logical circuit in which signal flow is unidirectional, a so-called feedforward circuit, has a finite memory span bounded by the maximum forward value of the combined propagation delays along any path from a primary output to a primary input. In order to construct a circuit with unbounded memory span from unidirectional logic elements, it is necessary to create a closed signal or feedback loop. Feedback is a basic property of sequential circuits (flip-flops and latches). The problem caused by feedback in a purely combinational logic circuit - that is, one with gate delay zero, is that it can lead to a logical inconsistency.

![NAND Gate With Feedback](image)

Figure 4.12: NAND Gate With Feedback

In Figure 4.12 there is a feedback loop from the output to the input of the NAND gate, implying the Boolean equation

\[ X(t) = \overline{A(t) \cdot X(t)} \]

must be satisfied. This equation is satisfied by \( A(t) = 0 \) and \( X(t) = 1 \), since

\[ 0 \cdot 1 = \overline{0} = 1. \]

Consequently, the signal configuration is consistent and stable. If \( A(t) = 1 \) we obtain a logically inconsistent situation, since

\[ \overline{A(t) \cdot X(t)} = 0. \]

Similarly, for \( X(t) = 0 \) we have a logically inconsistent situation, since \( X(t) \) cannot be 0 and 1 at the same time.

The inconsistency present in this example dissappears if the NAND gate has a nonzero propagation delay \( t_{pd} \), which also makes a better model for the behaviour of a physical gate. Our equation changes to

\[ X(t) = \overline{A(t - t_{pd}) \cdot X(t - t_{pd})}. \]

The output signal \( A(t) \) is no longer a function of its present value. Instead, it depends on the past value \( X(t - t_{pd}) \), which can differ from \( X(t) \). In particular, when \( A(t - t_{pd}) = 1 \), we can satisfy the equation with

\[ X(t) = \overline{X(t - t_{pd})}. \]
Hence, if \( A(t) \) changes from 0 to 1 at some time \( t \), this change will cause \( X(t) \) to change from 0 to 1 at some time \( t + t_{pd} \). Owing to our equation, this second change will change \( X(t) \) from 1 to 0 at \( t + 2t_{pd} \), and so on. Hence, the value of \( X(t) \) must change every \( t_{pd} \) time units. This type of regular and spontaneous changing, called oscillation, is an extreme form of unstable behaviour. However it is not logically inconsistent. This type of behaviour plays an important role in generating the clock signal that controls synchronous circuits. Spontaneous oscillation of the above kind involves narrow pulses of width \( t_{pd} \) that tend to be filtered out by the gate through which they pass. Consequently, such an oscillation usually dies out quickly.
Chapter 5

Combinational Circuits

5.1 Introduction

A combinational circuit consists of gates representing Boolean connectives; it is free of feedback loops. A combinational circuit has no state; its output depends solely on the momentary input values. Examples are the full adder, comparator, decoder and multiplexer. In reality, however, signal changes propagate through a sequence of gates with a finite speed. This is due to the capacitive loads of the amplifying transistors. Hence circuits have a certain propagation delay.

In this chapter we consider circuits such as adders, multipliers and comparators which can be used to build an arithmetic logic unit. Optimizations for some of these circuits are also considered.

Every Boolean function can be expressed in a normal form consisting of disjunctions of conjunctions, and it can therefore be implemented by two levels of gates only. Of considerable technical relevance are devices which represent two levels of gates in a general form. A specific function is selected by opening (or closing) connections between specific gates. This is called programming the device, and the device is a programmable logic device (PLD). The gates in a PLD are of the AND, OR and NOT types. Programming happens electrically under computer control. PLD’s are highly attractive in order to reduce the number of discrete components in circuits. A specific form of a PLD is the read-only memory (ROM). Another example is programmable array logic (PAL) where the AND gates are programmable.
5.2 Decoder

In digital computers, binary codes are used to represent many different types of information, such as instructions, numerical data, memory addresses, and control commands. A code group that contains \( N \) bits can have \( 2^N \) different combinations, each of which represents a different piece of information. A logic circuit is required which can take the \( N \)-bit code as logic inputs and then generate an appropriate output signal to identify which of the \( 2^N \) different combinations is present. Such a circuit is called a decoder.

Thus a 1-out-of-\( n \) decoder is a circuit with \( n \) outputs and \( N = \log_2 n = \ln n \) inputs, outputs \( X_j \) are numbered from 0 to \((n - 1)\). An output goes to 1 when the input number \( A \) is identical to the number \( j \) of the relevant output. Figure 5.1 shows the truth table for a 1-out-of-4 decoder. The variables \( A_0 \) and \( A_1 \) represent the binary code of the decimal number \( m \). The sum of the products (disjunctive normal form) of the recoding functions can be taken directly from the truth table. The circuit is also shown using AND and NOT gates. The functions are

\[
X_0 = \overline{A_0} \cdot \overline{A_1}, \quad X_1 = A_0 \cdot \overline{A_1}, \quad X_2 = \overline{A_0} \cdot A_1, \quad X_3 = A_0 \cdot A_1.
\]

Most integrated-circuit decoders can decode 2-, 3-, or 4-bit input codes.

In CMOS 4028 is a 4-bit BCD to 1-of-10 active high decoder.

<table>
<thead>
<tr>
<th>( m )</th>
<th>( A_0 )</th>
<th>( A_1 )</th>
<th>( X_0 )</th>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( X_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.1: Truth Table and Circuit of a 1-out-of-4 Decoder
5.3 Encoder

A decoder takes an input code and activates the one corresponding output. An encoder performs the opposite operation; it generates a binary code corresponding to which input has been activated. A commonly used IC encoder is represented in Figure 5.2. It has eight active LOW inputs, which are kept normally high. When one of the inputs is driven to 0, the binary output code is generated corresponding to that input. For example, when input \( T_3 \) = 0, the outputs will be \( CBA = 011 \), which is the binary equivalent of decimal 3. When \( T_6 \) = 0, the outputs will be \( CBA = 110 \). For some encoders, if more than one input is made low the output would be garbage. For a priority encoder, the outputs would be the binary code for the highest-numbered input that is activated. For example, assume that the encoder of the Figure is a priority encoder and that inputs \( T_4 \) and \( T_7 \) are simultaneously made low. The output code will be \( CBA = 111 \) corresponding to \( T_7 \). No matter how many inputs are activated, the code for the highest one will appear at the output.

![8-Line-to-3-line encoder](image)

Figure 5.2: Typical IC Encoder

A simple 8-input encoder is given by

\[
\begin{align*}
A &= I_1 + I_3 + I_5 + I_7 \\
B &= I_2 + I_3 + I_6 + I_7 \\
C &= I_4 + I_5 + I_6 + I_7 \\
V &= I_0 + I_1 + I_2 + I_3 + I_4 + I_5 + I_6 + I_7
\end{align*}
\]

The output \( V \) is used to indicate when an input is 1 for the encoder; it differentiates between the 0 input \( I_0 \) and when no inputs are 1. The encoder is not a priority encoder, it performs a bitwise OR on all the inputs which are set to 1.
In CMOS the 4532 is an 8-input priority encoder with eight active HIGH priority inputs \((I_0 \text{ to } I_7)\), three active HIGH outputs \((O_0 \text{ to } O_2)\), an active HIGH enable input \((E_{in})\), an active HIGH enable output \((E_{out})\) and an active HIGH group select output \((GS)\). Data is accepted on inputs \(I_0 \text{ to } I_7\). The binary code corresponding to the highest priority input \((I_0 \text{ to } I_7)\) which is HIGH, is generated on \(O_0 \text{ to } O_2\) if \(E_{in}\) is HIGH. Input \(I_7\) is assigned the highest priority. GS is HIGH when one or more priority inputs and \(E_{in}\) are HIGH. \(E_{out}\) is HIGH when \(I_0 \text{ to } I_7\) are LOW and \(E_{in}\) is HIGH. \(E_{in}\), when LOW, forces all outputs \((O_0 \text{ to } O_2, GS, E_{out})\) LOW. The circuit is given below.

![Circuit diagram](image)

Figure 5.3: Circuit for the CMOS 4532
The logic equations are

\[ O_2 = E_{in} \cdot (I_4 + I_5 + I_6 + I_7) \]

\[ O_1 = E_{in} \cdot (I_2 \cdot \overline{I_4} \cdot \overline{I_5} + I_3 \cdot \overline{I_4} \cdot \overline{I_5} + I_7) \]

\[ O_0 = E_{in} \cdot (I_1 \cdot \overline{I_2} \cdot \overline{I_4} \cdot \overline{I_6} + I_3 \cdot \overline{I_4} \cdot \overline{I_5} \cdot I_6 + I_7) \]

\[ E_{out} = E_{in} \cdot (I_0 \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot \overline{I_5} \cdot \overline{I_6} \cdot \overline{I_7}) \]

\[ GS = E_{in} \cdot (I_0 + I_1 + I_2 + I_3 + I_4 + I_5 + I_6 + I_7). \]

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_{in}$</td>
<td>$I_7$</td>
</tr>
<tr>
<td>L</td>
<td>X</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
</tbody>
</table>

Table 5.1: Truth Table for the CMOS 4532
5.4 Demultiplexer

A demultiplexer can be used to distribute input information $D$ to various outputs. It represents an extension of the 1-out-of-$n$ decoder. The addressed output does not go to one, but assumes the value of the input variable $D$. Figure 5.4 shows its implementation using AND and NOT gates. If we make $D = \text{const} = 1$, the demultiplexer operates as a 1-out-of-$n$ decoder.

The logic functions are

$$X_0 = D \cdot \overline{A_0} \cdot \overline{A_1}, \quad X_1 = D \cdot A_0 \cdot \overline{A_1}, \quad X_2 = D \cdot \overline{A_0} \cdot A_1, \quad X_3 = D \cdot A_0 \cdot A_1$$

The following figure shows the basic mode of operation and the circuit.

![Demultiplexer Circuit](image)

Figure 5.4: Demultiplexer Circuit

In CMOS the 4555 is a dual 1-of-4 decoder/demultiplexer. Each has two address inputs ($A_0$ and $A_1$), an active LOW enable input ($\overline{E}$) and four mutually exclusive outputs which are active HIGH ($O_0$ to $O_3$). When used as a decoder ($\overline{E}$ is HIGH), then $O_0$ to $O_3$ is LOW. When used as a demultiplexer, the appropriate output is selected by the information on $A_0$ and $A_1$ with $\overline{E}$ as data input. All unselected outputs are LOW.

The truth table is given by Table 5.2.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\overline{E}$</td>
<td>$A_0$</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 5.2: Truth Table for CMOS 4555
5.5 Multiplexer

A multiplexer or data selector is a logic circuit that accepts several data inputs and allows only one of them at a time to get through to the output. It is an extension of an encoder. The routing of the desired data input to the output is controlled by SELECT inputs (sometimes referred to as ADDRESS inputs). There are many IC multiplexers with various numbers of data inputs and select inputs. Thus the opposite of a demultiplexer is a multiplexer. The following figure shows the multiplexer circuit.

![Multiplexer Circuit](image)

Figure 5.5: Multiplexer Circuit

The logic function is

\[
X = \overline{A_0} \cdot \overline{A_1} \cdot D_0 + A_0 \cdot \overline{A_1} \cdot D_1 + \overline{A_0} \cdot A_1 \cdot D_2 + A_0 \cdot A_1 \cdot D_3.
\]

In CMOS technology, a multiplexer can be implemented using both gates and analog switches (transmission gates). When analog switches are employed, signal transmission is bidirectional. In this case, therefore, the multiplexer is identical to the demultiplexer. The circuit is then known as an analog multiplexer/demultiplexer.

In CMOS the 4019 provides four multiplexing circuits with common select inputs \((S_A, S_B)\). Each circuit contains two inputs \((A_n, B_n)\) and one output \((O_n)\). It may be used to select four bits of information from one of two sources. The \(A\) inputs are selected when \(S_A\) is HIGH, the \(B\) inputs are selected when \(S_B\) is HIGH. When \(S_A\) and \(S_B\) are HIGH, the output \((O_n)\) is the logical OR of the \(A_n\) and \(B_n\) inputs \((O_n = A_n + B_n)\). When \(S_A\) and \(S_B\) are LOW, the output \((O_n)\) is LOW independent of the multiplexer inputs.
5.6 Binary Adder

5.6.1 Binary Half Adder

Consider the task of adding two 1-bit numbers $A_0$ and $A_1$. Obviously,

$$0 + 0 = 0, \quad 0 + 1 = 1, \quad 1 + 0 = 1.$$  

The sum 1+1 requires two bits to represent it, namely 10, the binary form of two (decimal). This can be expressed as follows: one plus one yields a sum bit $S = 0$ and a carry bit $C = 1$. If we ignore the carry bit and restrict the sum to the single bit $s$, then we obtain $1+1 = 0$. This is a very useful special form of addition known as modulo-2 addition.

The half adder circuit can be realized using an XOR gate and an AND gate. One output gives the sum of the two bits and the other gives the carry. In CMOS for the AND gate the 4081 can be used and for the XOR gate the 4030 can be used.

The circuit and the figure for the input and output is shown in the following figure.

![Half Adder Circuit](image)

Figure 5.6: Half Adder Circuit

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_0$</td>
<td>$A_1$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.3: Half Adder Truth Table

The logic function for $S$ and $C$ are

$$S = A_0 \oplus A_1, \quad C = A_0 \cdot A_1.$$
5.6.2 Binary Full Adder

A full adder adds three bits at a time, a necessary operation when two multi-bit binary numbers are added. The figure shows a full adder. It consists of three AND gates, an OR gate and an XOR gate. The full adder computes the numerical sum of the three input bits in unsigned binary code. For example, when all three inputs are 1, the output is \( Y_0 Y_1 = 11 \), which is the binary representation of the decimal number three. One of the inputs is the carry bit from the previous addition.

We can construct logic expressions for these operations. The five gates (three AND gates, one OR gate and one XOR gate) used in the full adder given below lead to the logic equations

\[
X_0 = A_0 \cdot A_1, \quad X_1 = A_1 \cdot A_2, \quad X_2 = A_0 \cdot A_2, \\
Y_0 = X_0 + X_1 + X_2, \quad Y_1 = A_0 \oplus A_1 \oplus A_2.
\]

Note that the \( + \) is the logical OR operation, and \( \cdot \) is the AND operation and \( \oplus \) is the XOR operation.

![Figure 5.7: Full Adder Circuit](image)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>A₁</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.4: Full Adder Truth Table
5.6.3 Binary Four-Bit Adder

To add $3 + 3$ (decimal) (i.e. $11 + 11$ in binary), two full adders in parallel are required, as shown in the figure. Actually the first full adder, $FA_1$, need only be a half adder since it handles just two bits. The output is 6 (decimal) or 110 in binary.

![Figure 5.8: Two Full Adders in Parallel](image)

To add two four-bit numbers, four adders are connected in parallel as shown for the addition of binary 1110 (14 decimal) and 0111 (7 decimal) to give the sum 10101 (21 in decimal). By joining more full adders to the left of the system, numbers with more bits can be added.

![Figure 5.9: Four Bit Adder Consisting of Four Adders](image)

In CMOS the 4008 is a 4-bit binary full adder with two 4-bit data inputs, a carry input, four sum outputs, and a carry output. The IC uses full look-ahead across 4-bits to generate the carry output. This minimizes the necessity for extensive look-ahead and carry cascading circuits.
5.6.4 Faster Addition

The 4-bit adder calculates a carry bit three times before the final carry value can be passed to another 4-bit adder. This is known as carry cascading. A speed increase is obviously possible if the final carry value can be calculated immediately from the input values (to reduce the carry cascading). Let \( A_i, B_i \) and \( C_{i-1} \) \((i = 0, 1, 2)\) denote the inputs for each stage \( i \) of a 3-bit adder. The equations for the carry bits are given by

\[
C_{-1} := C_{\text{in}} \\
C_i = A_i \cdot B_i + A_i \cdot C_{i-1} + B_i \cdot C_{i-1} \quad i = 0, 1, 2.
\]

Thus for \( C_{\text{out}} \) we find

\[
C_2 = A_2 \cdot B_2 + A_2 \cdot A_1 \cdot B_1 + A_1 \cdot B_1 \cdot B_1 + \\
A_2 \cdot A_1 \cdot A_0 \cdot B_0 + A_2 \cdot A_0 \cdot B_1 \cdot B_0 + A_1 \cdot A_0 \cdot B_2 \cdot B_0 + A_0 \cdot B_2 \cdot B_1 \cdot B_0 + \\
A_2 \cdot A_1 \cdot A_0 \cdot C_{\text{in}} + A_2 \cdot A_1 \cdot B_0 \cdot C_{\text{in}} + A_2 \cdot A_0 \cdot B_1 \cdot C_{\text{in}} + A_1 \cdot A_0 \cdot B_2 \cdot C_{\text{in}} + \\
A_2 \cdot B_1 \cdot B_0 \cdot C_{\text{in}} + A_1 \cdot B_2 \cdot B_0 \cdot C_{\text{in}} + A_0 \cdot B_2 \cdot B_1 \cdot C_{\text{in}} + B_2 \cdot B_1 \cdot B_0 \cdot C_{\text{in}}.
\]

Each full adder requires 2 levels of gates to calculate the carry bit. Thus we have reduced the carry computation from 6 levels of gates to 2 levels of gates. The circuit for the computation is given below.

![Circuit diagram](image)

Figure 5.10: Circuit for the Carry Bit of a 3-bit Adder

Similary the calculation for the carry bit of a 4-bit adder will be reduced from 8 levels of gates to 2 levels of gates.
5.7 Binary Subtraction

Binary subtraction can be built in the same way as binary addition. However, it is possible to perform binary subtraction using binary addition which can simplify circuits designed to perform binary arithmetic. In chapter 3 a number of ways to represent negative integers in binary format were introduced. The two’s complement method made it possible to add signed integers using standard binary addition. Thus it makes sense to use the two’s complement format to enable subtraction using existing methods.

To subtract $B$ from $A$ where $B$ and $A$ are 4-bit numbers we use

$$A - B = A + (-B)$$

where the negation is two’s complement.

The circuit is as follows

![Binary Subtraction Circuit](image1)

Figure 5.11: Binary Subtraction Using the Two’s Complement
5.8 Binary Multiplication

5.8.1 Unsigned Integer Multiplication

Binary multiplication can be implemented with simple addition. This is very slow. Some improvement can be obtained by using the distributive properties of binary arithmetic. Suppose

\[ A = \sum_{i=0}^{n} a_i 2^i \quad \text{and} \quad B = \sum_{i=0}^{n} b_i 2^i \]

with \( a_i, b_i \in \{0, 1\} \) are to be multiplied. Using distributivity we find

\[ A \cdot B = \sum_{j=0}^{n} b_j \left( \sum_{i=0}^{n} a_i 2^{i+j} \right). \]

Multiplying by a power of 2 can be implemented as a simple shift operation. The following algorithm summarizes the technique.

1. \( j := 0, \text{result} := 0 \)
2. if \( b_j \) is 1 add \( A \) to \( \text{result} \)
3. shift left \( A \)
4. increment \( j \)
5. if \( j \leq n \) goto (2)

The algorithm requires a number of steps. To keep the circuits simple, control mechanisms introduced in Chapter 7 will be needed. The alternative is a more complicated implementation consisting of calculating all the shifted values of \( A \) first and then adding each input where the input lines of \( B \) are 1. The shift left operation is simple. To perform addition controlled by the \( b_j \) inputs it suffices to examine for each output line the input value \( I_i \), the calculated output \( O \) (in this case for addition) and \( b_j \). Thus the final output is given by

\[ b_j \cdot O + \overline{b_j} \cdot I. \]

To ensure that the product of two \( n \)-bit numbers can be represented the output may be extended to \( 2n \) bits.
The Russian peasant method [111] uses the same technique for multiplication, with a small change to simplify implementation. It is a practical method for multiplication by hand, since it involves only the operations of doubling, halving and adding. Western visitors to Russia in the nineteenth century found the method in wide use there, from which the method derives its name.

1. $j := 0, \text{result} := 0$
2. if $b_0$ is 1 add $A$ to result
3. shift left $A$
4. shift right $B$
5. increment $j$
6. if $j \leq n$ goto (2)
5.8.2 Fast Multiplication

Instead of using an iterative algorithm, multiplication can be performed in parallel by determining each output bit from the input bits only (at the same time). Consider the following multiplication of two bits $A_1 A_0$ and $B_1 B_0$.

\[
\begin{array}{ccc}
 & B_1 & B_0 \\
 A_1 & A_0 & \\
 & A_0 \cdot B_1 & A_0 \cdot B_0 \\
 A_1 \cdot B_1 & A_1 \cdot B_0 \\
P_3 & P_2 & P_1 & P_0
\end{array}
\]

The product is given by

\[
\begin{align*}
P_0 &= A_0 \cdot B_0 \\
P_1 &= (A_0 \cdot B_1) \oplus (A_1 \cdot B_0) \\
P_2 &= (A_1 \cdot B_1) \oplus (A_1 \cdot A_0 \cdot B_1 \cdot B_0) \\
P_3 &= A_1 \cdot A_0 \cdot B_1 \cdot B_0
\end{align*}
\]

Figure 5.13: 2-bit Fast Unsigned Multiplication
5.8.3 Signed Integer Multiplication

Two signed integers can be multiplied by multiplying their absolute values and setting the appropriate sign afterwards. For two’s complement numbers Booth’s algorithm is used. It is an efficient algorithm for multiplying signed integers and optimizes the calculation in the following way: suppose an \( N \)-bit unsigned integer has a block of consecutive 1’s in its binary representation. If the first 1 is at bit \( k \) and the last 1 is at bit \( n \) then the decimal representation for this block is \( 2^{n+1} - 2^k \). This follows from the fact that

\[
\sum_{k=0}^{n} 2^k = 2^{n+1} - 1.
\]

This can be extended for two’s complement numbers. The two’s complement (negation) of the same number gives \( 2^N - 2^n + 2^{k+1} \). The contribution of \( 2^N \) is an overflow and does not influence the operation. Thus addition and subtraction can be performed whenever a 1 and 0 are adjacent in the bit representations.

For Booth’s algorithm we introduce an extra bit \( Q_{-1} \) which is used to determine the boundaries of blocks of 0’s or 1’s. The final product is in \( AQ \). \( A \) and \( Q \) are \( n \)-bit registers. The arithmetic shift right (SHR) operation shifts all bits one position right, and leaves the highest order right (sign bit) at its previous value.

1. \( A := 0, Q_{-1} := 0 \)
   \( M := \) Multiplicand
   \( Q := \) Multiplier
   \( C := n \)
2. If \( Q_0 = Q_{-1} \) goto (5)
3. If \( Q_0Q_{-1} = 01 \) \( A := A + M \)
4. If \( Q_0Q_{-1} = 10 \) \( A := A - M \)
5. Arithmetic SHR \( A, Q, Q_{-1} \)
   \( C = C - 1 \)
6. If \( C \) is nonzero goto (2)
5.9 Binary Division

Binary division is modelled on the long division process, for example

- $A := 0$
  $M := $Divisor, $Q := $Dividend, $C := n$
- SHL $AQ$
- $A := A - M$
- if $(A < 0)$ goto (6)
- $Q_0 := 1$, goto (7)
- $Q_0 := 0$, $A := A + M$
- increment $C$
- if $(C > 0)$ goto (2)

Two's complement division:

- Load the divisor into the $M$ register and the dividend into the $AQ$ registers. The dividend must be expressed as a $2n$ two's complement number. Thus, for example, the 4-bit number 0111 becomes 0000111, and 1001 becomes 11111001.
- Shift left $AQ$ 1 bit position
- If $M$ and $A$ have the same signs, perform $A := A - M$; otherwise, $A := A + M$.
- The above operation is successful if the sign of $A$ is the same before and after the enumeration.
  1. If the operation is successful or $(A = 0$ AND $Q = 0$) then set $Q_0 = 1$.
  2. If the operation is unsuccessful and $(A \neq 0$ OR $Q \neq 0$), then set $Q_0 = 0$ and restore the previous value of $A$.
- Repeat steps (2) through 4 as many times as there are bit positions in $Q$.
- The remainder is in $A$. If the signs of the divisor and dividend are the same, the the quotient is in $Q$; otherwise the correct quotient is the two's complement of $Q$.
5.10 Magnitude Comparator

One of the basic operations in computation is comparing two integer numbers. Let \( a \) and \( b \) be two integers. We have to consider three cases \( a > b \), \( a = b \) and \( a < b \). For combinational circuits the comparison is done bit by bit. For example, 5 in binary is 0101b and 3 in binary is 0011b. First we compare the most significant bits of 5 and 3. Both are 0. Thus we move to the next bit (from left to right). For 5 the bit is set, namely 1 and for 3 we have 0. Thus 5 > 3.

In CMOS the 4585 is a four-bit magnitude comparator which compares two 4-bit words (\( A \) and \( B \)), whether they are ‘less than’, ‘equal to’ or ‘greater than’. Each word has four parallel inputs (A\(_0\) to A\(_3\)) and (B\(_0\) to B\(_3\)); A\(_3\) and B\(_3\) being the most significant inputs. Three outputs are provided. \( A \) greater than \( B \) (\( O_{A>B} \)), \( A \) less than \( B \) (\( O_{A<B} \)) and \( A \) equal to \( B \) (\( O_{A=B} \)). Three expander inputs (\( I_{A>B} \), \( I_{A<B} \) and \( I_{A=B} \)) allow cascading of the devices without external gates. For proper comparison operation the expander inputs to the least significant position must be connected as follows:

\[
I_{A=B} = I_{A>B} = \text{HIGH}, \quad I_{A<B} = \text{LOW}
\]

For words greater than 4-bits, units can be cascaded by connecting output \( O_{A<B} \) and \( O_{A=B} \) to the corresponding inputs of the next significant comparator (input \( I_{A>B} \) is connected to a HIGH). Operation is not restricted to binary codes, the devices will work with any monotonic code. Table 5.5 displays the truth table for the CMOS 4585. The following notation is used. H=HIGH state (the more positive voltage), L=LOW state (the less positive voltage), X = state is immaterial. The upper 11 lines describe the normal operation under all conditions that will occur in a single device or in a serial expansion scheme. The lower 2 lines describe the operation under abnormal conditions on the cascading inputs. These conditions occur when the parallel expansion technique is used. The circuit consists of 8 XNOR gates and one NAND gate.

In CMOS the 74LV688 is an 8-bit magnitude comparator. It takes two 8-bit numbers provided by the inputs \( P_0 \) to \( P_7 \) and \( Q_0 \) to \( Q_7 \). The output is

\[
P = Q.
\]

Table 5.6 shows the function table for the CMOS 74LV688 and Figure 5.14 the logic diagram for the CMOS 74LV688.
### Table 5.5: Truth Table for the CMOS 4585

<table>
<thead>
<tr>
<th>Comparing inputs</th>
<th>Cascading inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_3, B_3$</td>
<td>$A_2, B_2$</td>
<td>$A_1, B_1$</td>
</tr>
<tr>
<td>$A_3 &gt; B_3$</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$A_3 &lt; B_3$</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$A_3 = B_3$</td>
<td>$A_2 &gt; B_2$</td>
<td>X</td>
</tr>
<tr>
<td>$A_3 = B_3$</td>
<td>$A_2 &gt; B_2$</td>
<td>$A_1 &gt; B_1$</td>
</tr>
<tr>
<td>$A_3 = B_3$</td>
<td>$A_2 = B_2$</td>
<td>$A_1 = B_1$</td>
</tr>
<tr>
<td>$A_3 = B_3$</td>
<td>$A_2 = B_2$</td>
<td>$A_1 = B_1$</td>
</tr>
<tr>
<td>$A_3 = B_3$</td>
<td>$A_2 = B_2$</td>
<td>$A_1 = B_1$</td>
</tr>
<tr>
<td>$A_3 = B_3$</td>
<td>$A_2 = B_2$</td>
<td>$A_1 = B_1$</td>
</tr>
</tbody>
</table>

### Table 5.6: Function Table for CMOS 74LV688

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data $P_n, Q_n$</td>
<td>Enable $E$</td>
</tr>
<tr>
<td>$P = Q$</td>
<td>L</td>
</tr>
<tr>
<td>X</td>
<td>H</td>
</tr>
<tr>
<td>$P &gt; Q$</td>
<td>L</td>
</tr>
<tr>
<td>$P &lt; Q$</td>
<td>L</td>
</tr>
</tbody>
</table>

### Figure 5.14: Logic Diagram for the CMOS 74LV688
5.11 4-Bit ALU

The Arithmetic logic unit (ALU) is responsible for the arithmetic and logic operations discussed so far. Typically some input lines are used to select the required functionality. The ALU combined with memory (such as registers) and control logic is essentially all that is required for a classical computer. In CMOS the MC14581B is a 4-bit ALU capable of providing 16 functions of two Boolean variables and 16 binary arithmetic operations on two 4-bit words. The level of the mode control input determines whether the output function is logic or arithmetic. The desired logic function is selected by applying the appropriate binary word to the select inputs (S_0 thru S_3) with the mode control input HIGH, while the desired arithmetic operation is selected by applying a LOW to the mode control input, the required level to carry in, and the appropriate word to the select inputs. The word inputs and function outputs can be operated on with either active high or active low data. The arithmetic functions interpret the input words as two’s complement numbers. As noted in the table, when C_n is opposite to the given value, the result is the given arithmetic function plus 1 because C_n is interpreted as a carry bit.

Carry propagate (P) and carry generate (G) outputs are provided to allow a full look-ahead carry scheme for fast simultaneous carry generation for the four bits in the package. Fast arithmetic operations on long words are obtainable by using the MC14582B as a second order look ahead block. An inverted ripple carry input (C_n) and a ripple carry output (C_{n+4}) are included for ripple through operation.

When the device is in the subtract mode (LHHL), comparison of two 4-bit words present at the A and B inputs is provided using the A = B output. It assumes a high-level state when indicating equality. Also, when the ALU is in the subtract mode the C_{n+4} output can be used to indicate relative magnitude as shown in this table.

<table>
<thead>
<tr>
<th>Data level</th>
<th>C_n</th>
<th>C_{n+4}</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active High</td>
<td>H</td>
<td>H</td>
<td>A ≤ B</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>H</td>
<td>A ≤ B</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>H</td>
<td>A &lt; B</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>L</td>
<td>A &gt; B</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>L</td>
<td>A ≥ B</td>
</tr>
<tr>
<td>Active Low</td>
<td>H</td>
<td>H</td>
<td>A ≤ B</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>L</td>
<td>A ≤ B</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>L</td>
<td>A &lt; B</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>H</td>
<td>A &gt; B</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>H</td>
<td>A ≥ B</td>
</tr>
</tbody>
</table>
The truth table is as follows.

<table>
<thead>
<tr>
<th>Function Select</th>
<th>Inputs/Outputs Active Low</th>
<th>Inputs/Outputs Active High</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_3$</td>
<td>$S_2$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

The * indicates that the inputs are expressed in two’s complement form. For arithmetic functions with $C_n$ in the opposite state, the resulting function is as shown plus 1.

Thus, for active high inputs, the basic logic functions are achieved with the following selections and $MC = H$.

<table>
<thead>
<tr>
<th>$S_3$</th>
<th>$S_2$</th>
<th>$S_1$</th>
<th>$S_0$</th>
<th>Logic Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>NOT</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>AND</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>OR</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>NAND</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>NOR</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>XOR</td>
</tr>
</tbody>
</table>

The basic arithmetic operations of addition, subtraction, increment and decrement are provided when $MC = L$.

<table>
<thead>
<tr>
<th>$S_3$</th>
<th>$S_2$</th>
<th>$S_1$</th>
<th>$S_0$</th>
<th>$C_n$</th>
<th>Arithmetic Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>Addition</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>Subtraction</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>Increment</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>Decrement</td>
</tr>
</tbody>
</table>
5.12 Read Only Memory (ROM)

Another frequently encountered purely combinational circuit is the read-only memory. Its structure is such that any Boolean function of \( n \) variables, where \( n \) is the number of inputs, can be generated. Since complicated functions are usually not perceived as functions, but rather as individual values corresponding to the possible combinations of input values, the device is called a memory rather than a function generator.

A ROM essentially consists of a decoder of the binary encoded input number, called the address, an array of OR gates, and a set of output drivers. The decoder yields a selector signal for each input value, addressing each cell.

![Logic Diagram for a ROM](image)

Figure 5.15: Logic Diagram for a ROM

As an example when the combination \( A_2 = 1, A_1 = 1 \) and \( A_0 = 0 \) the decoder selects line 6 which means that the output lines give the bit string

\[
D_7D_6D_5D_4D_3D_2D_1D_0 = 00011111
\]

which is the binary representation of 31. The ROM can be used to speed up certain tasks. For example it can store multiplication tables. Of course ROMs can also be used to store identification strings or any other data.
5.13 Combinational Programmable Logic Devices

In the previous section the ROM was introduced. To make circuit design easier more flexible approaches are available. Many hardware implementations implement the SOP form (disjunctive normal form) of an expression directly. The circuit designed determines the connections to make for a given configuration of gates (for example selecting the inputs to the OR gates in the ROM configuration).

A programmable gate is one where the inputs to the gate can be selected from a given set of inputs (for example from other gates). If no inputs are selected we assume that all inputs are 0. We introduce a new notation to simplify circuit representation. A cross \( \times \) indicates a programmable connection to a gate, in other words a connection which can be removed (for example a fuse that can be burnt open). A dot \( \cdot \) indicates a fixed connection. The following figure shows an AND gate with programmable inputs (the inputs from \( A_0 \) and \( A_2 \) can still be removed) and an OR gate with two fixed inputs.

\[
\begin{array}{ccc}
A_0 & A_1 & A_2 \\
\times & \times & & \& \\
\geq 1 & & & 0 + A_1 + A_2 \\
\end{array}
\]

Figure 5.16: Input Representation for Programmable Gates

In the following examples we use two inputs, four AND gates and one OR gate for the output. In general, for \( n \) inputs and \( m \) outputs, \( 2^n \) AND gates and \( m \) OR gates are required. One way to implement a programmable AND gate is to have an AND gate with \( 2n \) inputs (for an input and its inverse) and to set the input to 1 whenever an input is not connected. Similarly for the OR gate an input can be set to 0. A special case is when no input is selected, the output of the gate must be zero (as if the gate is not present). In this case we set all inputs to the gate to 0. In this way gates with a fixed number of inputs can be used as programmable gates. For each architecture we show the circuit before programming and after programming the XOR operation.
**PROM** stands for programmable read only memory. These devices consist of a number of fixed AND gates (fixed in input) and programmable OR gates. The AND gates are over all possible inputs. For an $n$ variable system there are $2^n$ AND gates. All connections are initially closed, the unwanted connections are then burnt by applying a voltage to the appropriate inputs. Once a PROM is programmed it cannot be reprogrammed. The *EPROM* or *erasable* PROM can be erased (all connections are closed). The *EEPROM* is an *electrically erasable* PROM.

![Figure 5.17: PROM Device Architecture](image)

![Figure 5.18: PROM Implementation of XOR](image)
**PAL** stands for programmable array logic. A number of programmable AND gates feed fixed OR gates in these devices. The AND gates represent the product forms of the desired expression’s SOP form. Specific AND gates are dedicated to specific OR gates.

**GAL** stands for generic array logic. They are used to emulate PALs. Different types of PALs can then be replaced with a single device type (the GAL device).

![PAL Device Architecture](image1)

**Figure 5.19: PAL Device Architecture**

![PAL Implementation of XOR](image2)

**Figure 5.20: PAL Implementation of XOR**
PLA stands for programmable logic array. These devices provide the greatest programming flexibility through the use of programmable AND gates feeding programmable OR gates. Any AND gate can feed any OR gate.

Figure 5.21: PLA Device Architecture

Figure 5.22: PLA Implementation of XOR
5.14 Programmable Gate Arrays

A programmable gate array (PGA) consists of an array (or matrix) of cells. Each cell’s function and connections can be selected. Two kinds of field programmable gate arrays (FPGA) are commonly used. The first kind consists of cells which are loadable and erasable as a whole like an EPROM (i.e. all the cells are erased and written at the same time). The second type consists of cells which are not modifiable but the connections between them are. For example, suppose a cell can implement any of the 16 possible Boolean function of two variables. The cell can be programmed using a multiplexer to select the inputs used and a second multiplexer to determine the output. Of course, a multiplexer to select only the output function (where all 16 functions are implemented separately) is also possible. In the following figure the input multiplexers are assumed to be fixed (until reconfigured) and selection signals are not shown.

![Diagram of FPGA cell](image)

Figure 5.23: Example of a Combinational FPGA Cell

To design a circuit using FPGA the cell functions must be specified as well as the connections between cells. Determining which connections must be closed is called *routing*. For example a FPGA may have a grid pattern where the output can be connected to four adjacent cells. Each outward going arrow is a duplicate of the function output. Each input arrow can be configured to be closed or open.

![Grid pattern for FPGA](image)

Figure 5.24: Grid Pattern for PGA Design
5.15 VHDL

VHDL [171] is a standardized language that is not tied to any single tool vendor or hardware manufacturer. It is a complete programming language with built-in mechanisms to handle and synchronize parallel processes, and also supports abstract data types and high level modelling. The IEEE adopted VHDL as a standard in 1987.

VHDL was initially intended to describe digital electronics systems. It can be used to model existing hardware for verification and testing, and also for synthesis.

The following code example is a VHDL description of a 4-bit equality comparator. The data types bit and bit_vector and are basic data type in VHDL. The assignment operator is <= and the comparison operator is =. A comment is preceded by --, everything following on the same line is part of the comment. The entity declaration describes the communication mechanism. It describes pins (port) used for input and output. In the example below a and b represent 2 four bit inputs. Thus a(0) refers to the first bit of a and b(3) refers to the last bit of b. We refer to the logic values as '0' and '1'. The data type bit_vector uses literals in double quotes, for example "0010". The architecture declaration describes the functionality. In this case the architecture dataflow which is associated with the entity eqcomp4. It ensures that the output pin equals is always defined.

```vhdl
-- eqcomp4.vhd

-- eqcomp4 is a four bit equality comparator
entity eqcomp4 is
  port (a, b : in bit_vector(3 downto 0);
       equals: out bit); -- equals is active high
end eqcomp4;

architecture dataflow of eqcomp4 is
begin
  equals <= '1' when (a = b) else '0';
end dataflow
```
Chapter 6

Latches and Registers

6.1 Introduction

The combinational circuits introduced so far perform a function and, except for the ROM, do not provide any memory. The ROM provides a static memory, the content is predetermined. A system providing dynamic memory is required to store data which cannot be predetermined. Any two state (bistable) system which can be dynamically controlled will provide this function. Many systems acting in parallel can provide the required data width for operations provided by combinational circuits. The bistable systems are called latches and the parallel combination registers. Combinational circuits are free of loops. In this chapter we examine circuits with feedback loops. This is what allows them to store information. Propagation delays are important in the analysis of these circuits.

The following chapter introduces mechanisms for an external source of timing. The timing system helps describe the logic functions of these circuits under specific conditions.

In this chapter we discuss the SR latch and JK latch which use two inputs. One sets the logical value of the latch to 0 and the other sets the logical value of the latch to 1. The $D$ latch has only one input and remembers the logical value of the input for one time interval. The $D$ register and $JK$ register use $D$ latches and $JK$ latches respectively, to provide the same logical action as the latches, but with different physical characteristics.
6.2 SR Latch

This circuit has two inputs labelled $S$ (Set) and $R$ (Reset), and two outputs $Q$ and $\overline{Q}$, and consists of two NOR gates connected in a feedback arrangement.

![Logic Diagram for the SR Latch](image)

Figure 6.1: Logic Diagram for the SR Latch

The following table summarizes the characteristics of the operation of the latch. The $S$ and $R$ should not both be set at the same time as this gives an undetermined value for $Q$.

<table>
<thead>
<tr>
<th>$S_t$</th>
<th>$R_t$</th>
<th>$Q_t$</th>
<th>$Q_{t+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.1: Characteristic Table for the SR Latch

The circuit is stable when $S = R = 0$ ($Q_{t+1} = Q_t$). The output is time dependent and there is a delay from the time that one of $S$ or $R$ are set to one and the time when the circuit is stable again. If $S = 0$ and $R = 1$ the system is reset. If $S = 1$ and $R = 0$ the system is set. The logical equation for $Q_{t+1}$ (if $S_t$ and $R_t$ are not 1 at the same time) is

$$Q_{t+1} = S_t + \overline{R_t} \cdot Q_t.$$
6.3 D Latch

The input $S = R = 1$ must be avoided when using the $SR$ Latch. The $D$ latch overcomes this by using only a single $D$ input. The output is always the same as the last $D$ input.

![Logic Diagram for the D Latch](image)

Figure 6.2: Logic Diagram for the $D$ Latch

The $D$ latch is sometimes called the data latch because it stores 1 bit of information. It is also called the delay latch because it delays the output of the 0 or 1 (in an environment where a CLOCK input is provided, the delay is one clock cycle). The characteristic table for the $D$ latch is as follows:

<table>
<thead>
<tr>
<th>$D$</th>
<th>$Q_{t+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.2: Characteristic Table for the $D$ Latch

The latch described above is called *transparent* since the output $Q$ is the same as the input $D$. An extra input can be introduced to indicate when to set the output identical to the given input.

![Logic Diagram for the D Latch With Enable](image)

Figure 6.3: Logic Diagram for the $D$ Latch With Enable

The $G$ input is called an enable input.
6.4 JK Latch

The JK latch takes two inputs. Unlike the SR latch all input combinations are valid. The J input performs the set function while the K input performs the reset function. When \( J = K = 1 \) the toggle function is performed (the outputs are inverted).

![Logic Diagram for the JK Latch](image)

Figure 6.4: Logic Diagram for the JK Latch

The characteristic table describes the functionality of the circuit.

<table>
<thead>
<tr>
<th>( J_t )</th>
<th>( K_t )</th>
<th>( Q_t )</th>
<th>( Q_{t+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.3: Characteristic Table for the JK Latch

The logic equation for \( Q_{t+1} \) is

\[
Q_{t+1} = J_t \cdot \overline{K_t} + J_t \cdot \overline{Q_t} + \overline{K_t} \cdot Q_t.
\]
6.5 D Register

The transparency of a $D$ latch can be undesirable. It may be preferable to accept an input value upon the rising edge of a control signal, and retain the stored value before and after the transition. Latches are level-sensitive whereas registers are edge-sensitive. An example implementation of this is the master-slave latch pair. It consists of two latches connected in series with each enable input the inverse of the other. This separates the storage of the input $D$ and the output $Q$. The boxes with the symbols $D$, $G$ and $Q$ represent $D$ latches.

![D Register Using Two D Latches](image)

This is called an edge-triggered $D$ register. Typically $CK$ is a periodical signal. The signals $CK$ and $\overline{CK}$ must never be active at the same time to guarantee nontransparency. The output of a transparent $D$ latch can be used for this purpose. The following figure shows a D-Master-Slave register.

![Logic Diagram for the D Register](image)
6.6 JK Register

Similar to the $D$ register, the principle for the $JK$ register is based on the $JK$ latch. A master-slave configuration can again be used to implement this register.

![Diagram of JK Register Using Two JK Latches](image)

Figure 6.7: JK Register Using Two JK Latches

Each $JK$ latch has two additional AND gates for each input where the appropriate $CK$ or $\overline{CK}$ is the second input to each AND gate. This construction has the same purpose as the $G$ input in a $D$ latch. A variation on the register is to use the $Q$ and $\overline{Q}$ feedback loops directly to the first input and not for each latch. The following figure shows a $JK$-Master-Slave register.

![Diagram of Logic Diagram for the JK Register](image)

Figure 6.8: Logic Diagram for the $JK$ Register
Chapter 7

Synchronous Circuits

7.1 Introduction

Circuits that react immediately to the stimulus of the input are called \textit{asynchronous}. This term is a combination of the greek words meaning “without regard to time”. In digital systems it is important that outputs change at precise points in time. Circuits that operate in this manner are called \textit{synchronous}. Digital circuits often use time reference signals called \textit{clocks}. A clock signal is nothing more than a square wave that has a precise known period. The clock will be the timing reference that synchronizes all circuit activity and tells the device when it should execute its function. Thus the clock signal is the signal that causes things to happen at regularly spaced intervals. In particular, operations in the system are made to take place at times when the clock signal is making a transition from 0 to 1 or from 1 to 0. These transitions are pointed out in the figure. The 0-to-1 transition is called the rising edge or positive-going edge of the clock signal. The synchronous action of the clock signal is the result of using clocked latches, which are designed to change states on either (but not both) the rising edge or the falling edge of the clock signal. In other words, the clocked latches will change states at the appropriate clock transition and will rest between successive clock pulses. The frequency of the clock pulses is generally determined by how long it takes the latches and gates to respond to the level changes by the clock pulse, that is, the propagation delays of the various logic circuits.

![Figure 7.1: Example Clock Signal](image-url)
Many ways of designing and controlling latches have evolved over the years. They differ not only in their logic design but also how they use the clock signal. Let us consider a latch. During the period \( t_3 : t_4 \) when the clock is enabled \( C = 1 \), any change made to the data signal may enter the latch immediately. After some propagation delay, these changes affect the latch’s data output \( Q \) (and also \( \overline{Q} \)) during the period \( t_3 : t_4 \). Thus, ignoring the brief and somewhat uncertain transition periods when the data and clock signals are actually changing values, the latch responds to all input changes that occur when \( C \) is at the inactive 1 level. For this reason latches are said to be level sensitive or **level-triggered**.

![Level Sensitive Latch](image1)

**Figure 7.2: Level Sensitive Latch**

To obtain latch behavior, we must ensure that the period \( t_1 : t_2 \) (when input data changes are accepted) and the period \( t_3 : t_4 \) (when the output data changes) do not overlap. One way a latch can meet this requirement is by accepting input changes when \( C = 1 \), and changing its output when \( C = 0 \). This pulse mode of operation was used in some early designs for bistables. The clocking method most commonly used in modern latch design is **edge triggering**, in which a transition or edge of the clock signal \( C \) causes the actions required in \( t_1 : t_2 \) and \( t_3 : t_4 \) to take place, as shown in the figure.

![Edge Triggered Latch](image2)

**Figure 7.3: Edge Triggered Latch**
7.2 Shift Registers

Shift registers are classed as sequential logic circuits, and as such they are constructed from latches. Thus shift registers are chains of latches which allow data applied to the input to be advanced by one latch with each clock pulse. After passing through the chain, the data is available at the output with a delay but otherwise is unchanged. Shift registers are used as temporary memories and for shifting data to the left or right. Shift registers are also used for changing serial to parallel data or parallel to serial data.

Identification of shift registers may be made by noting how data is loaded into and read from the storage unit. In the following figure we have a register 8 bits wide. The registers are classified as:

1. Serial-in serial-out
2. Serial-in parallel-out
3. Parallel-in serial-out
4. Parallel-in Parallel-out

![Diagram of Shift Registers]

Figure 7.4: Types of Shift Registers
A simple four-bit shift register is displayed in the following figure. It uses four D-latches. Data bits (0s and 1s) are fed into the $D$ input of latch 1. This input is labelled as the serial data input. The clear input will reset all four D latches to 0 when activated by a LOW. A pulse at the clock input will shift the data from the serial-data input to the position $A$ ($Q$ of latch 1). The indicators $(A, B, C, D)$ across the top of the figure show the contents of the register. This register can be classified as a serial-in parallel-out unit if data is read from the parallel inputs $(A, B, C, D)$ across the top.

![Logic Diagram of a 4-bit Serial-Load Shift-Right Register](image)

Figure 7.5: Logic Diagram of a 4-bit Serial-Load Shift-Right Register

In CMOS the 4014B is a fully synchronous edge-triggered 8-bit static shift register with eight synchronous parallel inputs, a synchronous serial data input, a synchronous parallel enable, a LOW to HIGH edge-triggered clock input and buffered parallel outputs from the last three stages.


7.3 Binary Counter

Latches can be connected in various arrangements to function as binary counters that count input clock pulses. A wide variety of counters are available as standard integrated-circuit packages and it is seldom necessary to construct a counter from individual latches. We review the external operating characteristics of currently available IC counters.

Next we discuss the basic counter operation. The Figure 7.6 shows the schematic representation of a 4-bit counter. This counter contains four latches, one per bit, with outputs labelled A, B, C, and D. Two inputs are shown, the clock pulse input, CP, and Reset. The counter operates such that the states of the four latches represent a binary number equal to the number of pulses that have been applied to the CP input. The diagram shows the sequence which the latch outputs follow as pulses are applied. The A output represents the LSB (least significant bit) and D is the MSB (most significant bit) of the binary count. For example, after the fifth input pulse, the outputs DCBA = 0101, which is the binary equivalent of 5. The CP input has a small circle and triangle to indicate that the latches in the counter change states on the negative going edge of the clock pulses. Counters that trigger on positive-going edges are also available and they do not have the circle on the CP input.

![Diagram of a 4-bit binary counter](image)

**Figure 7.6: Four-Bit Binary Counter**

<table>
<thead>
<tr>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Before 1st input pulse</td>
</tr>
<tr>
<td>16 Different possible states</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| Sequence repeats | 0 | 0 | 0 | 0 | After 16th input pulse |
| | : | : | : | : | : |

Table 7.1: Counting Sequence
In general, a counter with \( N \) latches can count from 0 up to \( 2^N - 1 \), for a total of \( 2^N \) different states. The total number of different states is called the counter’s \textit{MOD number}. The counter in the Figure is a MOD-16 counter. A counter with \( N \) latches would be a MOD-\( 2^N \) counter. Some IC counters are designed so that the user can vary the counting sequence through the appropriate external additional logic connections. These are usually referred to as \textit{variable-MOD counters}.

In addition to counting pulses, all counters can perform \textit{frequency division}. This is illustrated in the following figure for the 4-bit, MOD-16 counter. The state of the \( A \) output is seen to change at a rate exactly \( \frac{1}{2} \) that of the CP input. The \( C \) output is \( \frac{1}{2} \) that of the \( A \) output and \( \frac{1}{4} \) of the \( CP \) input. The \( C \) output is \( \frac{1}{2} \) the frequency of the \( B \) output and \( \frac{1}{8} \) the input frequency, and the \( D \) output is \( \frac{1}{2} \) the frequency of the \( C \) output and \( \frac{1}{16} \) the input frequency. In general, the waveform out of the MSB latch of a counter will divide the input frequency by the MOD number.

![Counter Waveforms Showing Frequency Division](image)

The counters described above can count up from zero to some maximum count and then reset to zero. There are several IC counters that can count in either direction and are called up/down counters. The following figure shows the two basic up/down counter arrangements. The counter in this Figure has a single \( CP \) input that is used for both count-up and count-down operations. The UP/DOWN input is used to control the counting direction. One logic level applied to this input causes the counter to count up from 0000 to 1111 as pulses are applied to \( CP \). The other logic level applied to UP/DOWN causes the counter to count down from 1111 to 0000 as pulses are applied to \( CP \). The second counter does not use and UP/DOWN control input. Instead, it uses separate clock inputs \( CP_U \) and \( CP_D \) for counting up and down, respectively. Pulses applied to \( CP_U \) cause the counter to count up, and pulses applied to \( CP_D \) cause the counter to count down. Only one \( CP \) input can be pulsed at one time, or erratic operations will occur.

In CMOS the 4516 is an edge triggered synchronous up/down 4-bit binary counter with a clock input and an up/down count control input.
Figure 7.8: Representation of Two Types of Up/Down Counters

An example of a mod-4 ripple counter implemented with clocked $JK$ latches is given below. The $JK$ latch was chosen for the simple toggle capability.

Figure 7.9: 2-bit Binary Ripple Counter

The $CLK$ input of the second latch is driven by the output of the first latch. The $CLK$ input of the first latch is driven by an external clock signal. Every second toggle action of the first latch will cause the second latch to toggle. The output $A$ is the least significant bit and $B$ is the most significant bit of the binary counter.

This ripple action of one latch depending on the output of the previous latch can necessitate potentially large clock cycles, due to propagation delays. To avoid this lag, latches can be updated in parallel. The latches are driven by the same external clock at their $CLK$ inputs. This is illustrated below in another mod-4 counter.

Figure 7.10: 2-bit Binary Parallel Counter
VHDL can also be used to simulate a synchronous circuit. For example, consider

![Circuit Diagram]

Figure 7.11: Synchronous Circuit Specified in VHDL Program

The VHDL program is given below.

```vhdl
-- simple.vhd

entity simple is
  port(A, B, CLK: in bit;
       X, Y : out bit);
end simple;

architecture break_out of simple is
begin
  Y <= A or B;

  p1: process begin
      wait until CLK = '1';
      X <= A xor B;
  end process;
end break_out;
```


7.4 Example Program

For the PIC16F8X processor, the rotate instruction can be used to do fast multiplication. For example if we want to multiply two 8 bit numbers and store the result in two 8 bit registers (HBYTE and LBYTE), we apply the instruction rotate right \( f \) through carry, where \( f \) is an 8 bit register. For the PIC16F8X, RRF is the “rotate right \( f \) through carry” instruction. The following diagram illustrates the operation.

![Diagram of rotate right instruction](image)

The instruction BTFSC is the “bit test \( f \) and skip if clear” instruction. If the tested bit of \( f \) is 0 the next instruction is skipped. Thus if the BTFSC is executed with the operands STATUS and 0, the carry flag (STATUS register bit 0) is tested to determine if the next instruction is executed. The instruction DECFSZ is the “decrement \( f \) and skip if zero” instruction. The value of register \( f \) is decremented, and if the result is zero the next instruction is skipped.

```
;**********************************************************************************************************
; MULTIPLY.ASM
;
; Multiplies two 8 bit numbers
; 00000011 (decimal 3)
; and
; 01100101 (decimal 101)
; and stores the result (16 bits)
; 00000001 00101111 (decimal 303)
; in LBYTE and HBYTE
; LBYTE: 00101111
; HBYTE: 00000001
;
; RRL rotate right \( f \) through carry
; The contents of register \( f \) are rotated
; one bit to the right through the Carry Flag.
;
;**********************************************************************************************************
;
PROCESSOR 16f84
INCLUDE "p16f84.inc"
;
; Variable Declarations
LBYTE EQU H'11'; variable at address 0x11 in SRAM
HBYTE EQU H'12'; variable at address 0x12 in SRAM
COUNT EQU H'13'; variable at address 0x13 in SRAM
```
NQA EQU H’20’ ; first number at address 0x20 in SRAM
NOB EQU H’21’ ; second number at address 0x21 in SRAM

ORG H’00’

Start
BSF STATUS, R0
MOV LW B’1111111’
MOVWF PORTA
MOV LW B’00000000’
MOVWF PORTB
BCF STATUS, R0

CLR LBYTE
CLR HBYTE
MOV LW 8
MOVWF COUNT

MOV LW B’00000011’
MOVWF NOA
MOV LW B’01100101’
MOVWF NOB

MOVF NOB, W
BCF STATUS, 0

LOOP
RRF NOA
BTFSC STATUS, 0
ADDWF HBYTE
RRF HBYTE
RRF LBYTE
DECFSZ COUNT, F
GOTO LOOP

MOVF HBYTE, 0
MOVWF PORTB
Stop GOTO Stop

END
Chapter 8

Recursion

8.1 Introduction

Recursion is a fundamental concept in mathematics and computer science. It is a useful tool for simplifying solutions to problems. A recursive solution is possible if a problem can be solved using the solution of a simpler problem of the same type and a solution to the simplest of problems of the same type is known. A recursive solution to a problem consists of

- A solution to a simplest problem of the same type (base problem or stopping condition)
- A method to solve the problem if the solution to a simpler problem of the same type is known

Let us now list some recursive structures. One of the most important recursive structures are strings. The string manipulation functions can be implemented using recursion, for example to find the length of a string and reverse a string. The linear linked list is a recursive structure; it has a head followed by a linked list. An example implementation of a recursive linked list is given later in the next chapter, it allows lists to be copied, compared, searched and items to be inserted and deleted recursively. Another structure which is recursive is the binary tree.

In mathematics, recursion is the name given to the technique of defining a function in terms of itself. Any recursive definition must have an explicit definition for some value or values of the argument(s), otherwise the definition is circular. Recursion can also occur in another form if a process is defined in terms of subprocesses, one of which is identical to the main process.
For example, consider the double integral

\[ \int_a^b \int_c^d f(x, y) \, dx \, dy. \]

One method of evaluation is to write the double integral as a repeated integral

\[ \int_a^b \left( \int_c^d f(x, y) \, dy \right) \, dx. \]

The evaluation of the outer integral requires us to know the value of the integrand at selected points, and calculation of the integrand requires the evaluation of an integral, so that the subprocess is the same as the main process.

**Example.** The set \( \mathbb{N}_0^2 \) is bijective with \( \mathbb{N}_0 \). To see this we write the elements of \( \mathbb{N}_0^2 \) in a table.

\[
\begin{array}{ccccccc}
(0,0) & (0,1) & (0,2) & (0,3) & \ldots \\
(1,0) & (1,1) & (1,2) & (1,3) & \ldots \\
(2,0) & (2,1) & (2,2) & (2,3) & \ldots \\
(3,0) & (3,1) & (3,2) & (3,3) & \ldots \\
\vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
\]

We now write down the elements of this table by moving along the diagonals which go from north-east to south-west, that is, we write them in the sequence

\((0,0), (0,1), (1,0), (0,2), (1,1), (2,0), (0,3), (1,2), \ldots\)

Since there are \((k + 1)\) pairs \((r, s)\) with \(r + s = k\), we see that the pair \((m, n)\) occurs in the position

\[ 1 + 2 + \ldots + (m + n) + m = \frac{(m + n)(m + n + 1)}{2} + m. \]

Hence we have a bijection \( f : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0 \) given by

\[ f(m, n) = \frac{1}{2}(m + n)(m + n + 1) + m. \]
We have two functions $g$ and $h$ from $\mathbb{N}_0 \to \mathbb{N}_0$ such that $f^{-1}(r) = (g(r), h(r))$. They are given by the following formulas. Find $s \in \mathbb{N}_0$ so that

$$\frac{1}{2}s(s + 1) \leq r < \frac{1}{2}(s + 1)(s + 2).$$

Let $m$ be

$$r - \frac{1}{2}s(s + 1).$$

Then $m \leq s$, and $g(r) = m$, and $h(r) = s - m$.

We can use $f$ to obtain bijections

$$f_k : \mathbb{N}_0^k \to \mathbb{N}_0$$

for all $k$ using recursion. We define $f_1$ to be the identity, and we define $f_3$ by

$$f_3(x_1, x_2, x_3) = f(x_1, f(x_2, x_3)).$$

If $f_k$ has been defined, then $f_{k+1}$ is defined by

$$f_{k+1}(x_1, x_2, \ldots, x_{k+1}) := f(x_1, f_k(x_2, \ldots, x_{k+1})).$$

The inverse of $f_k$ has as its components composites of $g$ and $h$. For instance

$$f_3^{-1}(r) = (g(r), g(h(r)), h(h(r))).$$

\hfill \blacklozenge

**Example.** Let $n = 0, 1, \ldots$ and $f(n) = 2^n$. Then we can find the recursive definition as follows

$$f(n + 1) = 2^{n+1} = 2 \cdot 2^n = 2f(n)$$

Thus $f(n + 1) = 2f(n)$ where $f(0) = 1$.

\hfill \blacklozenge

**Example.** Another typical example of recursion is the *Fibonacci sequence* given by

$$F_{n+2} = F_{n+1} + F_n \quad n = 0, 1, 2, \ldots$$

where $F_0 = F_1 = 1$.

\hfill \blacklozenge
Example. The Bessel functions $J_n(x)$ are solutions of the linear second order differential equation

$$x \frac{d^2 y}{dx^2} + \frac{d y}{dx} + (x^2 - n^2)y = 0 \quad n = 0, 1, 2, \ldots$$

A recurrence formula for Bessel functions is given by

$$J_{n+2}(x) = \frac{2(n+1)}{x} J_{n+1}(x) - J_n(x) \quad n = 0, 1, 2, \ldots$$

where

$$J_0(x) = \sum_{j=0}^{\infty} (-1)^j \frac{x^{2j}}{\prod_{k=1}^{j}(2k)^2}$$

Example. Given a first order differential equation

$$\frac{dy}{dx} = f(x, y(x)), \quad y(x_0) = y_0$$

where $f$ is an analytic function of $x$ and $y$. Formal integration yields

$$y(x) = y_0 + \int_{x_0}^{x} f(s, y(s))ds.$$ 

Thus a recursive definition for an approximation of $y(x)$ is given by

$$y_{n+1}(x) = y_0 + \int_{x_0}^{x} f(s, y_n(s))ds.$$ 

This is known as Picard’s method.
As an example we consider

\[
\frac{dx}{dy} = x + y, \quad x_0 = 0, \quad y(x_0) = 1.
\]

The approximation at each step is given by

\[
y_{n+1}(x) = 1 + \int_0^x (s + y_n(s))ds.
\]

The method yields a polynomial approximation after each step. Thus

\[
y_0(x) = 1
\]

\[
y_1(x) = \frac{x^2}{2} + x + 1
\]

\[
y_2(x) = \frac{x^3}{6} + x^2 + x + 1
\]

\[
y_3(x) = \frac{x^4}{24} + \frac{x^3}{3} + x^2 + x + 1
\]

\[
y_4(x) = \frac{x^5}{120} + \frac{x^4}{12} + \frac{x^3}{3} + x^2 + x + 1
\]

\[
y_5(x) = \frac{x^6}{720} + \frac{x^5}{60} + \frac{x^4}{12} + \frac{x^3}{3} + x^2 + x + 1
\]
8.2 Example Programs

Example. The Towers of Hanoi problem illustrates the benefits of recursion very well. The problem has an easy recursive solution. The problem is as follows

- There are 3 pegs A, B and C.
- There are \( n \) discs of different sizes.
- Initially all \( n \) discs are on peg A with the largest disc at the bottom and discs decrease in size towards the top of the pile. If disc 1 is above disc 2 then disc 1 is smaller than disc 2.
- Only one disc may be moved at a time. A disc must be moved from the top of a pile on one peg to the top of a pile on another peg. A larger disc may not be placed on a smaller one.
- The task is to move all the discs from peg A to peg B.

If \( n=1 \) we can move the disc from A to B. If \( n=2 \) we can move a disc from A to C, A to B, C to B. This is the inspiration for the solution to the general problem. If \( n > 2 \) move the pile of \( n-1 \) discs from A to C, move the disc on A to B and move the pile on peg C to peg B.

```cpp
// hanoi.cpp

#include <iostream>

using namespace std;

void hanoi(unsigned long n, char A, char B, char C)
{
    if(n==1)
        cout << A << " -> " << B << endl;
    else
    {
        hanoi(n-1,A,C,B);
        cout << A << " -> " << B << endl;
        hanoi(n-1,C,B,A);
    }
}

void main(void)
{
    cout << "Tower of Hanoi with 1 disc:" << endl;
    hanoi(1,'A','B','C');
    cout << "Tower of Hanoi with 2 discs:" << endl;
```
8.2. EXAMPLE PROGRAMS

```cpp
hanoi(2,'A','B','C');
cout << "Tower of Hanoi with 3 discs:" << endl;
hanoi(3,'A','B','C');
cout << "Tower of Hanoi with 4 discs:" << endl;
hanoi(4,'A','B','C');
}
```

The output of the program is

Tower of Hanoi with 1 disc:
A → B

Tower of Hanoi with 2 discs:
A → C
A → B
C → B

Tower of Hanoi with 3 discs:
A → B
A → C
B → C
A → B
C → A
C → B
A → B

Tower of Hanoi with 4 discs:
A → C
A → B
C → B
A → C
B → A
B → C
A → C
A → B
C → B
C → A
B → A
C → B
A → C
A → B
C → B
Example. The number of multiplications involved in calculating an integer power of a number can be reduced significantly with a recursive solution. Using the identity

\[ a^n = \begin{cases} 
a^{n/2}a^{n/2} & n \text{ even} \\
a^{n/2}a^{n/2} & n \text{ odd}
\end{cases} \]

with \( a \in \mathbb{R} \) and \( n \in \mathbb{N} \) and \( \frac{n}{2} \) is calculated using integer division (i.e. \( \lfloor \frac{n}{2} \rfloor \)). The program power.cpp implements the solution.

// power.cpp

#include <iostream>
#include <iomanip>

using namespace std;

double power(double a, unsigned int n)
{
    double power_ndiv2;

    if(n == 0) return 1.0;

    power_ndiv2 = power(a, n/2);
    if(n)
    {
        return a*power_ndiv2*power_ndiv2;
    }
    return power_ndiv2*power_ndiv2;
}

void main(void)
{
    cout << "13.4^0=\n" << power(3.4, 0) << endl;
    cout << "2^24=" << setprecision(9) << power(2, 24) << endl;
    cout << "3.1415^7=" << setprecision(9) << power(3.1415, 7) << endl;
}

The output of the program is

13.4^0=1
2^24=16777216
3.1415^7=3019.66975
8.2. EXAMPLE PROGRAMS

**Example.** If $R$ is a relation on a set $A$ and $S$ is a sequence of elements from $A$ then the sequence $S$ can be sorted. For $R$ to be a relation we require

1. $R \subseteq A \times A$.
   We view the statement $(a, b) \in R$ with $a, b \in A$ as a proposition. We also write $(a, b) \in R$ as $aRb$

2. $aRb$ and $bRc$ implies $aRc$

A fast sorting method would be to place the elements of $S$ in a tree as they occur in the sequence and traverse the tree to find the sorted sequence. Another fast sorting algorithm called *quicksort* is implemented using recursion. The algorithm first partitions the sequence around an element $s_i$ such that all elements on the left of $s_i$ have the property $s_jR s_i$ and all elements to the right of $s_i$ do not. The next step is to sort each of the partitions, and we use use quicksort to do this (i.e. recursively). The program `qsort.cpp` uses the function `partition` to partition the sequence at each step of the qsort algorithm. This is the most important part of the algorithm. The function takes an element of the array and rearranges the array such that all elements before are less than the given element and all elements after are greater than the given element.

// qsort.cpp

```cpp
#include <iostream>
#include <string>

using namespace std;

// general definition of ordering R(t1,t2)
// returns >0 if t2 R t1, <=0 otherwise
template <class T>
void partition(T *array, int n, int (*R)(T,T), int &p)
{
    // partition around the first element of the array
    // any element could have been used.

    // p is the index of the element around which the
    // partition is made
    // pe (declared below) points to the element after
    // the second partition
    int i = n-1, pe = 1;
    T temp1,temp2;

    p=0;
    while(i > 0)
    {
```


if(R(array[p],array[pe]) > 0)
{
    temp1 = array[p]; temp2 = array[p+1];
    array[p++] = array[pe]; // put element in first partition
    array[p] = temp1; // move element around which partition
    // is made, one element right
    if(pe-p > 0) // if the second partition is not empty
        array[pe] = temp2; // move second partition one element right
}
pe++;
i--;}

template <class T>
void qsort(T *array,int n,int (*R)(T,T))
{
    int pelement;
    if(n <= 1) return;

    partition(array,n,R,pelement);
    qsort(array,pelement,R);
    qsort(array+pelement+1,n-pelement-1,R);
}

int less_int(int n1,int n2) { return n1>n2; }
int less_string(string n1,string n2) { return (n1>n2); }

void main(void)
{
    int test1[9] = {1,5,3,7,2,9,4,6,8};
    string test2[6] = {"orange","grape","pear","apple","banana","peach");
    int i;

    qsort<int>(test1,9,less_int);
    qsort<string>(test2,6,less_string);
    for(i=0;i<9;i++) cout << test1[i] << " ";
    cout << endl;
    for(i=0;i<6;i++) cout << test2[i] << " ";
    cout << endl;
}

The output of the program is
1 2 3 4 5 6 7 8 9
apple banana grape orange peach pear
8.2. EXAMPLE PROGRAMS

Example. The Ackermann function \( f : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N} \) is defined as

\[
f(n, m) := \begin{cases} 
  m + 1 & \text{if } n = 0 \\
  f(n - 1, 1) & \text{if } m = 0 \\
  f(n - 1, f(n, m - 1)) & \text{otherwise}
\end{cases}
\]

Thus \( f \) is defined recursively. Ackermann’s function \( f \) is a total function; each pair of numbers \((n, m)\) yields a value \( f(n, m) \) of the function. We see that \( f(n, m) \) depends only on the values of \( f(r, p) \) with \( r < n \) and \( p < m \).

// acker.cpp
#include <iostream>

using namespace std;

unsigned long ackermann(unsigned long n,unsigned long m)
{
    if(n==0) return m+1;
    if(m==0) return ackermann(n-1,1);
    return ackermann(n-1,ackermann(n,m-1));
}

void main(void)
{
    cout<<"f(1,1)="<<ackermann(1,1)"
    "<<"f(2,1)="<<ackermann(2,1)"
    "<<"f(3,1)="<<ackermann(3,1)<<endl;
    cout<<"f(1,2)="<<ackermann(1,2)"
    "<<"f(2,2)="<<ackermann(2,2)"
    "<<"f(3,2)="<<ackermann(3,2)<<endl;
    cout<<"f(1,3)="<<ackermann(1,3)"
    "<<"f(2,3)="<<ackermann(2,3)"
    "<<"f(3,3)="<<ackermann(3,3)<<endl;
}

The output of the program is

\[
\begin{align*}
  f(1,1) &= 3 & f(2,1) &= 5 & f(3,1) &= 13 \\
  f(1,2) &= 4 & f(2,2) &= 7 & f(3,2) &= 29 \\
  f(1,3) &= 5 & f(2,3) &= 9 & f(3,3) &= 61
\end{align*}
\]
Example. The logistic map $f : [0, 1] \rightarrow [0, 1]$ is given by

$$f(x) = 4x(1 - x)$$

The map can be written as a difference equation

$$x_{t+1} = 4x_t(1 - x_t) \quad t = 0, 1, 2, \ldots$$

where $x_0 \in [0, 1]$ is the initial value. Thus we can implement the function to compute $x_t$ recursively. Of course it makes more sense to implement the function using iteration [188].

// logistic.cpp
#include <iostream>

using namespace std;

double logistic(unsigned int t, double x0)
{
    double x;
    if(t==0) return x0;
    x=logistic(t-1,x0);
    return 4.0*x*(1.0-x);
}

void main(void)
{
    cout << "x100 = " << logistic(100,0.3899)
         << " when x0=0.3899" << endl;
    cout << "x500 = " << logistic(500,0.5)
         << " when x0=0.5" << endl;
    cout << "x10000 = " << logistic(10000,0.89881)
         << " when x0=0.89881" << endl;
}

The output of the program is

x100 = 0.744501 when x0=0.3899
x500 = 0 when x0=0.5
x10000 = 0.311571 when x0=0.89881
Example. Horner’s rule is used to reduce the number of multiplications in evaluating a polynomial. Consider, for example the polynomial

\[ P_5(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \]

where \( x, a_5, a_4, a_3, a_2, a_1 \) and \( a_0 \) are given numbers. Finding \( P_5 \) would involve \( 5 + 4 + 3 + 2 + 1 = 15 \) multiplications and 5 additions. Rewriting this in the form (Horner’s rule)

\[ P_5 = (((a_5x + a_4)x + a_3)x + a_2)x + a_1)x + a_0 \]

reduces the number of multiplications to five and we still have five additions. In general, let

\[ P_n(x) = a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0 \]

which can be rewritten as

\[ P_n(x) = (\ldots ((a_nx + a_{n-1})x + \ldots a_1)x + a_0. \]

Then we have \( n \) multiplications and \( n \) additions. The next program shows a non-recursive implementation of Horner’s rule in C++.

// horner1.cpp

#include <iostream>

using namespace std;

template <class T>
T P(T x, const T *a, int n)
{
    T s = a[n];
    while (--n >= 0) s = s*x + a[n];
    return s;
}

void main(void)
{
    const double a[5] = { 1.0, 0.5, 0.0, -18.0, 3.0 };

    cout << "P(x) = 3x^4-18x^3+3x/2+1" << endl;
    cout << "P(0.0) = " << P(0.0, a, 4) << endl;
    cout << "P(-1.0) = " << P(-1.0, a, 4) << endl;
    cout << "P(5.0) = " << P(5.0, a, 4) << endl;
}
A recursive implementation of Horner’s rule is given in the next program.

```
// horner2.cpp

#include <iostream>

using namespace std;

template <class T>
T P(T x, const T *a, int n)
{
    if(n==0)
        return a[0];
    return a[0]+x*P(x,a+1,n-1);
}

void main(void)
{
    const double a[5] = { 1.0,0.5,0.0,-18.0,3.0 };  
    cout << "P(x) = 3x^4-18x^3+x/2+1" << endl;
    cout << "P(0.0) = " << P(0.0,a,4) << endl;
    cout << "P(-1.0) = " << P(-1.0,a,4) << endl;
    cout << "P(5.0) = " << P(5.0,a,4) << endl;
}
```

The output of both programs is

```
P(x) = 3x^4-18x^3+x/2+1
P(0.0) = 1
P(-1.0) = 21.5
P(5.0) = -371.5
```
Example. The following Java program is used to construct a graphic pattern called a Hilbert curve. Each curve $H_i$ consists of four half-sized copies of $H_{i-1}$ with a different orientation. The Hilbert curve is the limit of this construction process, i.e $H_\infty$. Thus we can implement the methods $A()$, $B()$, $C()$ and $D()$ to draw the four copies for each step in the construction of the Hilbert curve using recursion. Lines are drawn to connect the four copies. For example, the first three steps in constructing the Hilbert curve are given below.

![Hilbert curve steps](image)

Figure 8.1: First 3 Steps in the Construction of the Hilbert Curve

```java
// Hilbert.java
import java.awt.*;
import java.awt.event.*;

public class Hilbert extends JFrame {
    public Hilbert()
    {
        addWindowListener(this);
        drawButton.addActionListener(this);
        setTitle("Hilbert");
        Panel parameterPanel = new Panel();
        parameterPanel.setLayout(new GridLayout(2,1));
        Panel nStepsPanel = new Panel();
        nStepsPanel.add(new Label("no of steps = ");
        nStepsPanel.add(nStepsField);
        Panel buttonPanel = new Panel();
        buttonPanel.add(drawButton);
        parameterPanel.add(nStepsPanel);
        parameterPanel.add(buttonPanel);
        add("North",parameterPanel);
        add("Center",hilbertCurve);
        setSize(400,400); setVisible(true);
    }
```
public static void main(String[] args)
{ new Hilbert(); }

class HilbertCurve extends Canvas
{
    private int x, y, h, n, len;

    public HilbertCurve() { n = 5; }

    public void A()
    {
        if(n > 0)
        {
            Graphics g = getGraphics(); n--;
            D(); g.drawLine(x, y, x-h, y); x-=h;
            A(); g.drawLine(x, y, x-h, y); y-=h;
            A(); g.drawLine(x, y, x+h, y); x+=h;
            B(); n++;
        }
    }

class HilbertCurve extends Canvas
{
    private int x, y, h, n, len;

    public HilbertCurve() { n = 5; }

    public void A()
    {
        if(n > 0)
        {
            Graphics g = getGraphics(); n--;
            D(); g.drawLine(x, y, x-h, y); x-=h;
            A(); g.drawLine(x, y, x-h, y); y-=h;
            A(); g.drawLine(x, y, x+h, y); x+=h;
            B(); n++;
        }
    }

class HilbertCurve extends Canvas
{
    private int x, y, h, n, len;

    public HilbertCurve() { n = 5; }

    public void A()
    {
        if(n > 0)
        {
            Graphics g = getGraphics(); n--;
            D(); g.drawLine(x, y, x-h, y); x-=h;
            A(); g.drawLine(x, y, x-h, y); y-=h;
            A(); g.drawLine(x, y, x+h, y); x+=h;
            B(); n++;
        }
    }
```java
{  
    Graphics g = getGraphics(); n--;  
    C(); g.drawLine(x, y, x, y+h); y+=h;  
    B(); g.drawLine(x, y, x+h, y); x+=h;  
    B(); g.drawLine(x, y, x-h, y); x-=h;  
    A(); n++;  
}
}
public void C()  
{  
    if(n > 0)  
    {  
        Graphics g = getGraphics(); n--;  
        B(); g.drawLine(x, y, x+h, y); x+=h;  
        C(); g.drawLine(x, y, x, y+h); y+=h;  
        C(); g.drawLine(x, y, x-h, y); x-=h;  
        D(); n++;  
    }  
}
public void D()  
{  
    if(n > 0)  
    {  
        Graphics g = getGraphics(); n--;  
        A(); g.drawLine(x, y, x, y-h); y-=h;  
        D(); g.drawLine(x, y, x-h, y); x-=h;  
        D(); g.drawLine(x, y, x+h, y); y+=h;  
        C(); n++;  
    }  
}

public void paint(Graphics g)  
{  
    Dimension size = getSize();  
    h = 4*Math.min(size.width,size.height)/5;  
    x = size.width/2+h/2;  
    y = size.height/2+h/2;  

    for(int i=len-1;i<n;i++) len = 2*len+1;  
    h/=len; A();  
}

public void setSteps(int nSteps)  
{  
    n = nSteps; repaint();  
}  
}```
8.3 Mutual Recursion

If the solution to a problem relies on the solution to another problem which in turn
relies on a simpler problem of the first type, a recursive solution can be implemented.
Mutual recursion refers to the recursive dependence of one solution on another. This
concept can be extended for more than two problems.

The Jacobi elliptic functions can be defined as inverse of the elliptic integral of first
kind [56]. Thus, if we write

\[
x(\phi, k) = \int_0^\phi \frac{ds}{\sqrt{1 - k^2 \sin^2 s}}
\]  

(8.1)

where \( k \in [0, 1] \) we then define the following functions

\[
\begin{align*}
\text{sn}(x, k) &:= \sin(\phi), & \text{cn}(x, k) &:= \cos(\phi), & \text{dn}(x, k) &:= \sqrt{1 - k^2 \sin^2 \phi}.
\end{align*}
\]

(8.2)

For \( k = 0 \) we obtain

\[
\begin{align*}
\text{sn}(x, 0) &\equiv \sin(x), & \text{cn}(x, 0) &\equiv \cos(x), & \text{dn}(x, 0) &\equiv 1
\end{align*}
\]

(8.3)

and for \( k = 1 \) we have

\[
\begin{align*}
\text{sn}(x, 1) &\equiv \tanh(x), & \text{cn}(x, 1) &\equiv \text{dn}(x, 1) &\equiv \frac{2}{e^x + e^{-x}}.
\end{align*}
\]

(8.4)

We have the following identities

\[
\begin{align*}
\text{sn}(x, k) &\equiv \frac{2\text{sn}(x/2, k)\text{cn}(x/2, k)\text{dn}(x/2, k)}{1 - k^2 \text{sn}^4(x/2, k)}
\end{align*}
\]

(8.5)

\[
\begin{align*}
\text{cn}(x, k) &\equiv \frac{1 - 2\text{sn}^2(x/2, k) + k^2 \text{sn}^4(x/2, k)}{1 - k^2 \text{sn}^4(x/2, k)}
\end{align*}
\]

(8.6)

\[
\begin{align*}
\text{dn}(x, k) &\equiv \frac{1 - 2k^2 \text{sn}^2(x/2, k) + k^2 \text{sn}^4(x/2, k)}{1 - k^2 \text{sn}^4(x/2, k)}.
\end{align*}
\]

(8.7)
8.3. MUTUAL RECURSION

The expansions of the Jacobi elliptic functions in powers of \( x \) up to order 3 are given by

\[
\text{sn}(x, k) = x - (1 + k^2)^{x^3} + \ldots \\
\text{cn}(x, k) = 1 - \frac{x^2}{2!} + \ldots \\
\text{dn}(x, k) = 1 - k^2 \frac{x^2}{2!} + \ldots
\]  

(8.8)  
(8.9)  
(8.10)

For \( x \) sufficiently small these will be good approximations.

We can now use the identities (8.5)-(8.7) and the expansions (8.8)-(8.10) to implement the Jacobi elliptic functions using one recursive call. The recursive call in \texttt{scan} uses half of the provided parameter \( x \). In other words the absolute value of the parameter passed in the recursive call is always smaller (by \( \frac{1}{2} \)). This guarantees that for fixed \( \epsilon > 0 \) the parameter \( |x| \) will satisfy \( |x| < \epsilon \) after a finite number of recursive calls. At this point a result is returned immediately using the polynomial approximation (8.8)-(8.10). This ensures that the algorithm will complete successfully. The recursive call is possible due to the identities for the \text{sn}, \text{cn} and \text{dn} functions given in (8.5)-(8.7). Since the identities depend on all three functions \text{sn}, \text{cn} and \text{dn} we can calculate all three at each step instead of repeating calculations for each of \text{sn}, \text{cn} and \text{dn} [88]. Lastly some optimization was done to reduce the number of multiplications used in the double angle formulas. We also use the fact that the denominator of all three identities is the same.

The advantage of this approach is that all three Jacobi elliptic functions are found with one function call. Furthermore the cases \( k = 0 \) and \( k = 1 \) include the sine, cosine, tanh and sech functions. Obviously, for these special cases faster routines are available. Elliptic functions belong to the class of doubly periodic functions in which \( 2K \) plays a similar role to \( \pi \) in the theory of circular functions, where \( K = F(1, k) \) is the complete elliptic integral of first kind. We have the identities

\[
\text{sn}(x \pm 2K, k) \equiv -\text{sn}(x, k), \quad \text{cn}(x \pm 2K, k) \equiv -\text{cn}(x, k), \quad \text{dn}(x \pm 2K, k) \equiv \text{dn}(x, k).
\]
To reduce the argument of the Jacobi elliptic functions we can also apply these identities.

The recursion method described above can be implemented using C++ as follows. The arguments to the function \texttt{scdn} are

- \( x \), the first argument to \( \text{sn} \), \( \text{cn} \) and \( \text{dn} \).
- \( k^2 \), the square of the second argument to \( \text{sn} \), \( \text{cn} \) and \( \text{dn} \).
- \( \text{eps} \), the upper bound on the argument \( x \) for application of the Taylor expansion approximation.
- \( s \), a variable for the value of \( \text{sn}(x,k) \).
- \( c \), a variable for the value of \( \text{cn}(x,k) \).
- \( d \), a variable for the value of \( \text{dn}(x,k) \).

Using the implementation we calculate the sine, cosine, identity, hyperbolic tan, hyperbolic sec and hyperbolic cosec functions for the value 3.14159.

```cpp
// jacobi.cpp
#include <iostream.h>
#include <math.h>

// forward declaration
void scdn(double, double, double, double&, double&, double&);

void main(void)
{
    double x, k, k2, eps;
    x = 3.14159;
    eps = 0.01;

    double res1, res2, res3;

    cout << "x = " << x << endl;

    // \sin, \cos, 1 of x
    k = 0.0;
    k2 = k*k;
    scdn(x, k2, eps, res1, res2, res3);
    cout << "\sin(x) = " << res1 << endl;
    cout << "\cos(x) = " << res2 << endl;
    cout << "1(x) = " << res3 << endl;
```
8.3. MUTUAL RECURSION

// tanh, sech, sech of x
k = 1.0;
k2 = k*k;
scdn(x,k2,eps,res1,res2,res3);
cout << "tanh(x) = " << res1 << endl;
cout << "sech(x) = " << res2 << endl;
cout << "sech(x) = " << res3 << endl;
}

void scdn(double x,double k2,double eps,double &s,double &c,double &d)
{
  if(fabs(x) < eps)
  {
    double x2 = x*x/2.0;
    s = x*(1.0 - (1.0 + k2)*x2/3.0);
    c = 1.0 - x2;
    d = 1.0 - k2*x2;
  }
  else
  {
    double sh,ch,dh;
    scdn(x/2.0,k2,eps,sh,ch,dh); // recursive call
    double sh2 = sh*sh;
    double sh4 = k2*sh2*sh2;
    double denom = 1.0 - sh4;
    s = 2.0*sh*ch*dh/denom;
    c = (1.0 - 2.0*sh2+sh4)/denom;
    d = (1.0 - 2.0*k2*sh2+sh4)/denom;
  }
}
8.4 Wavelets and Recursion

The discrete wavelet transform (or DWT) is an orthogonal function which can be applied to a finite group of data. Functionally, it is very much like the discrete Fourier transform, in that the transforming function is orthogonal, a signal passed twice through the transformation is unchanged, and the input signal is assumed to be a set of discrete-time samples. Both transforms are convolutions. Whereas the basis function of the Fourier transform is sinusoidal, the wavelet basis is a set of functions which are defined by a recursive difference equation

\[ \phi(x) = \sum_{k=0}^{M-1} c_k \phi(2x - k) \]

where the range of the summation is determined by the specified number of nonzero coefficients \( M \). The number of the coefficients is not arbitrary and is determined by constraints of orthogonality and normalization. Owing to the periodic boundary condition we have

\[ c_k \equiv c_{k+nM} \]

where \( n \in \mathbb{N} \). Generally, the area under the wavelet curve over all space should be unity, i.e.

\[ \int_\mathbb{R} \phi(x) dx = 1. \]

It follows that

\[ \sum_{k=0}^{M-1} c_k = 2. \]

In the Hilbert space \( L_2(\mathbb{R}) \), the function \( \phi \) is orthogonal to its translations; i.e.

\[ \int_\mathbb{R} \phi(x)\phi(x - k) dx = 0, \quad k \neq 0. \]

What is desired is a function \( \psi \) which is also orthogonal to its dilations, or scales, i.e.,

\[ \int_\mathbb{R} \psi(x)\psi(2x - k) dx = 0. \]
8.4. WAVELETS AND RECURSION

Such a function \( \psi \) does exist and is given by (the so-called associated wavelet function)

\[
\psi(x) = \sum_{k=1} \left(-1\right)^k c_{1-k} \phi(2x - k)
\]

which is dependent on the solution of \( \phi \). Normalization requires that

\[
\sum_k c_k c_{k-2m} = 2\delta_{0m}
\]

which means that the above sum is zero for all \( m \) not equal to zero, and that the sum of the squares of all coefficients is two. Another equation which can be derived from the above conditions is

\[
\sum_k (-1)^k c_{1-k} c_{k-2m} = 0.
\]

A way to solve for \( \phi \) is to construct a matrix of coefficient values. This is a square \( M \times M \) matrix where \( M \) is the number of nonzero coefficients. The matrix is designated \( L \) with entries

\[
L_{ij} = c_{2i-j}.
\]

This matrix has an eigenvalue equal to 1, and its corresponding (normalized) eigenvector contains, as its components, the value of the function \( \phi \) at integer values of \( x \). Once these values are known, all other values of the function \( \phi \) can be generated by applying the recursion equation to get values at half-integer \( x \), quarter-integer \( x \), and so on down to the desired dilation. This determines the accuracy of the function approximation.

An example for \( \psi \) is the Haar function

\[
\psi(x) := \begin{cases} 
1 & 0 \leq x < \frac{1}{2} \\
-1 & \frac{1}{2} \leq x < 1 \\
0 & \text{otherwise}
\end{cases}
\]

and \( \phi \) is given by

\[
\phi(x) = \begin{cases} 
1 & 0 \leq x < 1 \\
0 & \text{otherwise}
\end{cases}
\]

The functions

\[
\psi_{m,n}(x) := 2^{-m} \psi(2^{-m} x - n), \quad m, n \in \mathbb{Z}
\]
form a basis in the Hilbert space $L_2(\mathbb{R})$.

This class of wavelet functions is constrained, by definition, to be zero outside of a small interval. This is what makes the wavelet transform able to operate on a finite set of data, a property which is formally called compact support. The recursion relation ensures that a wavelet function $\phi$ is non-differentiable everywhere. The following table lists coefficients for three wavelet transforms. The pyramid algorithm

<table>
<thead>
<tr>
<th>Wavelet</th>
<th>$c_0$</th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
<th>$c_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haar</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Daubechies-4</td>
<td>$\frac{1}{4}(1 + \sqrt{3})$</td>
<td>$\frac{1}{4}(3 + \sqrt{3})$</td>
<td>$\frac{1}{4}(3 - \sqrt{3})$</td>
<td>$\frac{1}{4}(1 - \sqrt{3})$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Daubechies-6</td>
<td>0.332671</td>
<td>0.806891</td>
<td>0.459877</td>
<td>-0.135011</td>
<td>-0.085441</td>
<td>0.035226</td>
</tr>
</tbody>
</table>

Table 8.1: Coefficients for Three Wavelet Functions

operates on a finite set on $N$ input data, where $N$ is a power of two; this value will be referred to as the input block size. These data are passed through two convolution functions, each of which creates an output stream that is half the length of the original input. These convolution functions are filters, one half of the output is produced by the “low-pass” filter

$$a_i = \frac{1}{2} \sum_{j=0}^{N-1} c_{2i-j+1} f_j, \quad i = 0, 1, \ldots, \frac{N}{2} - 1$$

and the other half is produced by the “high-pass” filter function

$$b_i = \frac{1}{2} \sum_{j=0}^{N-1} (-1)^j c_{j-2i} f_j, \quad i = 0, 1, \ldots, \frac{N}{2} - 1$$

where $N$ is the input block size, $c_j$ are the coefficients, $f$ is the input function, and $a$ and $b$ are the output functions. In the case of the lattice filter, the low- and high-pass outputs are usually referred to as the odd and even outputs, respectively. In many situations, the odd or low-pass output contains most of the information content of the original input signal. The even, or high-pass output contains the difference between the true input and the value of the reconstructed input if it were to be reconstructed from only the information given in the odd output. In general, higher order wavelets (i.e. those with more nonzero coefficients) tend to put more information into the odd output, and less into the even output. If the average amplitude of the even output is low enough, then the even half of the signal may be discarded without greatly affecting the quality of the reconstructed signal. An
important step in wavelet-based data compression is finding wavelet functions which cause the even terms to be nearly zero.

The Haar wavelet represents a simple interpolation scheme. After passing these data through the filter functions, the output of the low-pass filter consists of the average of every two samples, and the output of the high-pass filter consists of the difference of every two samples. The high-pass filter contains less information than the low pass output. If the signal is reconstructed by an inverse low-pass filter of the form

\[ f^L_j = \sum_{i=0}^{N/2-1} c_{2i-j+1} a_i, \quad j = 0, 1, \ldots, N - 1 \]

then the result is a duplication of each entry from the low-pass filter output. This is a wavelet reconstruction with 2× data compression. Since the perfect reconstruction is a sum of the inverse low-pass and inverse high-pass filters, the output of the inverse high-pass filter can be calculated. This is the result of the inverse high-pass filter function

\[ f^H_j = \sum_{i=0}^{N/2-1} (-1)^j c_{j-1-2i} b_i, \quad j = 0, 1, \ldots, N - 1. \]

The perfectly reconstructed signal is

\[ f = f^L + f^H \]

where each \( f \) is the vector with elements \( f_j \). Using other coefficients and other orders of wavelets yields similar results, except that the outputs are not exactly averages and differences, as in the case using the Haar coefficients.

The following C++ program implements the Haar wavelet transform.

```cpp
// wavelet.cpp

#include <iostream>
#include <math.h>

void main()
{
    const double pi = 3.14159;
    int n = 16; // n must be a power of 2
    double* f = new double[n];
```
// input signal
int k;
for(k=0; k < n; k++)
    f[k] = sin(2.0*pi*(k+1)/n);

double* c = new double[n];
for(k=0; k < n; k++)
    c[k] = 0.0;
c[0] = 1.0; c[1] = 1.0;

double* a = new double[n/2];
for(k=0; k < n/2; k++)
    a[k] = 0.0;
double* b = new double[n/2];
for(k=0; k < n/2; k++)
    b[k] = 0.0;

int i, j;
for(i=0; i < n/2; i++)
{
    for(j=0; j < n; j++)
    {
        if(2*i-j+1 < 0) a[i] += c[2*i-j+1+n]*f[j];
        else a[i] += c[2*i-j+1]*f[j];
    }
    a[i] = 0.5*a[i];
}

for(i=0; i < n/2; i++)
{
    for(j=0; j < n; j++)
    {
        if(j-2*i < 0) b[i] += pow(-1.0,j)*c[j-2*i+n]*f[j];
        else b[i] += pow(-1.0,j)*c[j-2*i]*f[j];
    }
    b[i] = 0.5*b[i];
}

for(k=0; k < n/2; k++)
cout << "a[" << k << "] = " << a[k] << endl;
for(k=0; k < n/2; k++)
cout << "b[" << k << "] = " << b[k] << endl;

//inverse
double* fL = new double[n];
double* fH = new double[n];

for(j=0; j < n; j++)
fL[j] = 0.0;
for(j=0; j < n; j++)
fH[j] = 0.0;

for(j=0; j < n; j++)
{
    for(i=0; i < n/2; i++)
    {
        if(2*i-j+1 < 0) fL[j] += c[2*i-j+1+n]*a[i];
        else fL[j] += c[2*i-j+1]*a[i];
    }
}

for(k=0; k < n; k++)
cout << "fL[" << k << "] = " << fL[k] << endl;

for(j=0; j < n; j++)
{
    for(i=0; i < n/2; i++)
    {
        if(j-1-2*i < 0) fH[j] += pow(-1.0,j)*c[j-1-2*i+n]*b[i];
        else fH[j] += pow(-1.0,j)*c[j-1-2*i]*b[i];
    }
}

for(k=0; k < n; k++)
cout << "fH[" << k << "] = " << fH[k] << endl;

// input signal reconstructed
double* g = new double[n];
for(k=0; k < n; k++)
g[k] = fL[k] + fH[k];

for(k=0; k < n; k++)
cout << "g[" << k << "] = " << g[k] << endl;
8.5 Primitive Recursive Functions

Let \( \mathbb{N}_0 \) be the natural numbers including 0, i.e. \( \{0, 1, 2, \ldots \} \). For a function to be computable, there must be an algorithm or procedure for computing it. So in a formal definition of this class of functions we must replace the intuitive, semantic ideas with precise descriptions of the functions \([49, 67, 83, 103, 129]\).

To begin, we take as variable the letters \( n, x_1, x_2, \ldots \). We write \( \textbf{x} \) for \( (x_1, \ldots, x_k) \).

Next we list the basic, incontrovertibly computable functions which we use as building blocks for all others.

- **zero function** \( Z : \mathbb{N}_0 \to \mathbb{N}_0 \) \( Z(n) = 0 \) for all \( n \)
- **successor function** \( S : \mathbb{N}_0 \to \mathbb{N}_0 \) \( S(n) = n + 1 \)
- **projection functions** \( P^i_k : \mathbb{N}_0^k \to \mathbb{N}_0 \) \( P^i_k(x_1, \ldots, x_k) = x_i \) for \( 1 \leq i \leq k \)

These functions are called *initial functions*. We sometimes call the projections pick-out functions, and \( P^1_1 \) the identity function, written \( id(x) = x \).

Next, we specify the ways we allow new functions to be defined from ones we already have.

- **Composition.** If \( g \) is a function of \( m \)-variables and \( h_1, \ldots, h_m \) are functions of \( k \) variables, which are already defined, then composition yields the function

  \[
  f(\textbf{x}) = g(h_1(\textbf{x}), \ldots, h_m(\textbf{x})).
  \]

- **Primitive recursion.** For functions of one variable the schema is

  \[
  f(0) = d \\
  f(n + 1) = h(f(n), n)
  \]

  where \( d \) is a number and \( h \) is a function already defined.

For functions of two or more variables, if \( g \) and \( h \) are already defined then \( f \) is given by primitive recursion on \( h \) with basis \( g \) as

\[
\begin{align*}
  f(0, \textbf{x}) &= d \\
  f(n + 1, \textbf{x}) &= h(f(n, \textbf{x}), n, \textbf{x})
\end{align*}
\]

The reason we allow both \( n \) and \( \textbf{x} \) as well as \( f(n, \textbf{x}) \) to appear in \( h \) is that we may wish to keep track of both the stage we are at and the input.
The primitive recursive functions are exactly those which are either an initial function or can be obtained from the initial functions by a finite number of applications of the basic operations.

**Example.** The sum $x_1 + x_2$ is primitive recursive defined by

$$\begin{align*}
\text{sum}(0, x_2) &= P_1(x_2) \\
\text{sum}(x_1 + 1, x_2) &= S(\text{sum}(x_1, x_2))
\end{align*}$$

**Example.** The product $x_1 x_2$ is primitive recursive defined by

$$\begin{align*}
\text{product}(0, x_2) &= 0 \\
\text{product}(x_1 + 1, x_2) &= \text{sum}(\text{product}(x_1, x_2), x_2)
\end{align*}$$

**Example.** The predecessor of $x$ is primitive recursive defined by

$$\begin{align*}
\text{pred}(0) &= 0 \\
\text{pred}(x + 1) &= x
\end{align*}$$

**Example.** The function

$$\text{minus}(x_1, x_2) = \begin{cases} 
  x_1 - x_2 & x_1 \geq x_2 \\
  0 & \text{otherwise}
\end{cases}$$

is primitive recursive.

$$\begin{align*}
\text{minus}(0, x_2) &= 0 \\
\text{minus}(x_1 + 1, x_2) &= S(\text{minus}(x_1, x_2))
\end{align*}$$

**Example.** We can now describe the mod function

$$\text{mod}(n, m) = \begin{cases} 
  n & n < m \\
  \text{mod}(n - m, m) & \text{otherwise}
\end{cases}$$

The definition is as follows.

$$\begin{align*}
\text{mod}(0, m) &= 0 \\
\text{mod}(n + 1, m) &= k(\text{mod}(n, m), n, m)
\end{align*}$$

$$k(p, n, m) = \text{minus}(S(p), \text{product}(m, \text{minus}(S(p), m)))$$

The function $k$ is a composition of primitive recursive functions (some of which are compositions themselves) and is primitive recursive. Thus $\text{mod}(n, m)$ is primitive recursive. ◆
The Ackermann function \( f \) is not primitive recursive. It is obviously not an initial function. Nor can it be defined for the case \( f(0, m) \) as a number independent of \( m \). Thus the only way the Ackermann function can be primitive recursive is if it is a composition of primitive recursive functions. Since for \( m \neq 0 \) and \( n \neq 0 \) \( f \) relies on an evaluation of \( f \) itself it cannot be a composition of primitive recursive functions.

**Definition.** A set \( C \) of total functions from \( \mathbb{N}^n \) to \( \mathbb{N} \) (for all \( n \)) is called *primitive recursively closed* if it satisfies the following conditions.

1. all initial functions are in \( C \)
2. \( C \) is closed under primitive recursion, if \( f \) comes from \( g \) and \( h \) by primitive recursion and \( g, h \in C \) then \( f \in C \)
3. \( C \) is closed under composition, if \( f \) comes from \( g \) and \( h_1, \ldots, h_r \) by composition \( (f(x) = g(h_1(x), \ldots, h_r(x))) \) and \( g, h_1, \ldots, h_r \in C \) then \( f \in C \).

**Theorem.** The set of all primitive recursive functions is primitive recursively closed. [49, 67, 103]

**Theorem.** Any primitive recursively closed set contains every primitive recursive function. [49, 67, 103]

**Definition.** The definition of \( \mu \)-recursive is

- The primitive recursive functions are \( \mu \)-recursive.
- If \( f : \mathbb{N}^{n+1} \rightarrow \mathbb{N} \) is \( \mu \)-recursive then \( \mu f : \mathbb{N}^n \rightarrow \mathbb{N} \) defined by

  \[ \mu f := \min \{ x \mid f(x, y) = 0 \text{ with } f(u, y) \text{ defined for } u \leq x \} \]

  is \( \mu \)-recursive. The function \( \mu f \) will be undefined if no such \( x \) exists.

- Functions defined by composition and primitive recursion of \( \mu \)-recursive functions are also \( \mu \)-recursive.

In other words \( \mu \) acts as a minimalization operator in the sense that it maps to the minimum \( x \) such that \( f(x, y) = 0 \).

The Ackermann function is \( \mu \)-recursive.
8.6 Backtracking

A common approach to finding a solution, when no simple solution algorithm is available, is trial and error. Suppose a configuration is built up by a number of well defined steps and then tested to see if it is a solution. If it is not a solution we return to an earlier step and try a different option. Backtracking is the technique of returning to an earlier stage in a solution process to make a different choice in the attempt to find solutions.

Example. 8-Queens problem. A chessboard is 8 columns wide and 8 rows high. The 8 queens problem requires us to place 8 queens on the chessboard so that no queen is attacking another. A queen attacks another if it is on the same row, column or diagonal as the other queen. An example solution is

![Figure 8.2: A Solution to the 8-Queens Problem](image)

The recursive solution is to place a queen on a position which is not attacked, row by row. If there is no position available for a queen, the algorithm return to the previous row and moves the queen to the next position which is not attacked. To check every possible placement of the 8 queens includes configurations which are obviously incorrect and will also take a long time. This algorithm uses a technique called pruning to reduce the number of configurations to test. We can form a tree according to the square we place each queen in. The root of the tree corresponds to an empty board. The branches from the root correspond to each possible placement of a queen. By rejecting certain options earlier, the corresponding branches and entire sub-trees are removed from consideration.
// queens.cpp

#include <iostream>

using namespace std;

const char QUEEN = 'Q';
const char SPACE = '#';

void printboard(char board[8][8])
{
    int i,j;
    for(i=0; i<8; i++)
    {
        for(j=0; j<8; j++)
            cout << board[i][j];
        cout << endl;
    }
    cout << endl;
}

int attacking(char board[8][8], int row, int col)
{
    int i;
    for(i=0; i<8; i++)
    {
        if((board[row][i] == QUEEN) || (board[i][col] == QUEEN)) return 1;
        if((i+row-col>=0) && (i+row-col<8))
            if(board[i+row-col][i] == QUEEN)
                return 1;
        if((col+row-i>=0) && (col+row-i<8))
            if(board[col+row-i][i] == QUEEN)
                return 1;
    }
    return 0;
}

void queens(char board[8][8], int row)
{
    int i;
    if(row<0) return;
    if(row>7) { printboard(board); return; }
    for(i=0; i<8; i++)
        if(!attacking(board, row, i))
        {

8.6. BACKTRACKING

```c
board[row] [i]=QUEEN;
queens(board, row+1);
board[row] [i]=SPACE;
}
}

void main(void)
{
  char board[8][8];
  int i, j;

  for(i=0; i<8; i++)
    for(j=0; j<8; j++)
      board[i] [j]=SPACE;
  queens(board, 0);
}

The program output is

Q#########
####Q###
####Q##
####Q###
####Q###
####Q###
####Q###
###Q###

:
8.7 Stacks and Recursion Mechanisms

8.7.1 Recursion Using Stacks

Most implementations of recursion rely on a stack which is a last-in first-out (LIFO) storage mechanism. The stack has two operations, namely to push data onto the top of a stack and pop data off the top of the stack. For a recursive function call all the local data must be preserved. This is done by pushing the local data onto a stack. After the recursive function call has completed, the local data is popped off the stack into registers or other local storage. This stack can also be used to return the result of a function call. Next we introduce two routines called CALL and RETURN to implement the recursion. CALL has three arguments.

1. The address of the function to enter.

2. The address of the first register of workspace to be preserved.

3. The number of registers to be preserved.

The CALL routine pushes the specified registers and return address onto the stack, and then transfers control to the specified function.

RETURN has two arguments.

1. The address of the first register of the work-space to be restored.

2. The number of registers to be restored.

The RETURN routine restores the specified registers from the stack and then returns control to the return address on the stack. In shortened form

\[
\text{CALL}(\text{function}, \text{registers}) : \text{PUSH current address} \\
\quad \text{PUSH registers} \\
\quad \text{GOTO function}
\]

\[
\text{RETURN}(\text{registers}) : \text{POP into registers} \\
\quad \text{POP into returnaddress} \\
\quad \text{GOTO returnaddress}
\]

A simple recursive function would then be

\[
\text{function} : \text{IF (base case) ... RETURN(registers)} \\
\quad \ldots \\
\quad \text{CALL(function, registers)} \\
\quad \ldots \\
\quad \text{RETURN(registers)}
\]
8.7. STACKS AND RECURSION MECHANISMS

8.7.2 Stack Free Recursion

It is sometimes possible to convert a recursive function to an iterative one. The stack is very important in implementing recursion. It allows the recursive function to return to a previous state which it has stored. This is also possible if the function to be computed has an inverse. The inverse allows an iterative routine to return to previous values of the function evaluation.

**Example.** The Fibonacci sequence defined recursively by

\[ F_{n+2} = F_{n+1} + F_n \quad n = 0, 1, 2, \ldots \]

where \( F_0 = F_1 = 1 \) can be implemented iteratively by simply storing the previous two function evaluations. In this case the inverse is given by

\[ F_n = F_{n+2} - F_{n+1} \quad n = 0, 1, 2, \ldots \]

but it is not necessary to use this information.

```
// fibonacci.cpp

#include <iostream>

using namespace std;

void main(void)
{
    int i;
    unsigned long F0 = 1, F1 = 1;
    unsigned long temp;

    for(i=0; i<10; i++)
    {
        cout << F0 << " ";
        temp = F1;
        F1 = F0 + F1;
        F0 = temp;
    }
    cout << endl;
}
```
To remove the dependence on a stack, the changes made to variables by a recursive call must be reversible. This excludes some variables such as those used exclusively for return values. If we consider the towers of Hanoi problem, the algorithm just swaps variables and decrements a variable. This can obviously be reversed. Swapping two variables is reversed by the same action and the decrement is reversed by an increment. The changes must be made immediately before the recursive call and reversed immediately after the recursive call.

```
// hanoi2.cpp

#include <iostream>

using namespace std;

char A,B,C;
unsigned long n;

void hanoi()
{
  if(n==1) cout << A << " -> " << B << endl;
  else
  {
    n--; C = B^C; B = B^C; C = B^C; // swap B and C
    hanoi();
    C = B^C; B = B^C; C = B^C; // swap B and C back
    cout << A << " -> " << B << endl;
    C = A^C; A = A^C; C = A^C; // swap A and C
    hanoi();
    n++; C = A^C; A = A^C; C = A^C; // swap A and C back
  }
}

void main(void)
{
  A = 'A'; B = 'B'; C = 'C'; n = 1;
  cout << "Tower of Hanoi with 1 disc:" << endl;
  hanoi();
  A = 'A'; B = 'B'; C = 'C'; n = 2;
  cout << "Tower of Hanoi with 2 discs:" << endl;
  hanoi();
  A = 'A'; B = 'B'; C = 'C'; n = 3;
  cout << "Tower of Hanoi with 3 discs:" << endl;
  hanoi();
}
Chapter 9

Abstract Data Types

9.1 Introduction

Programming languages such as C++ and Java have built in data types (so-called basic data types or primitive data types) such as integers that represent information and have operations that can be performed on them (such as multiplication and addition). For example the built in basic data types in C++ are short, int, long, float, double and char.

An abstract data type (ADT) consists of data and the operations which can be performed on it. Generally the data is represented with standard data types of the language in which it is implemented but can also include other abstract data types. The operations defined on the ADT provide access to the information and manipulation of the data without knowing the implementation of the ADT. The abstract data type is implemented using constructors, data fields and methods (functions). Information hiding is when ADT data is inaccessible (no operation can retrieve the data). Encapsulation refers to the hiding of inner details (such as implementation). In C++ the concepts of public, private and protected data fields and methods are important in the implementation of an ADT. Public members of an ADT are always accesible. Private members are only accesible by the ADT itself and protected members are only accesible by the ADT and any derived ADT’s. A derived ADT may override the accessibility of members by forcing all inherited members to a specified level if they are more accessible.

For example the Standard Template Library [4] in C++ introduces many ADT’s such as Vector, list, stack, queue and set. Standard C++ now includes the abstract data type string. Symbolic C++ [192] includes the template classes Rational, Complex, Quaternion, Vector, Matrix, Polynomial and Sum. Operations such as addition and multiplication, determinant, trace and inverse of matrices are included in the matrix class. An instance of Matrix could then be used in the same way that integers are used without knowing the internal differences.

In the following sections various useful ADT’s will be introduced.
9.2 Linked List

The linked list is a useful data structure that can dynamically grow according to data storage requirements. This is done by viewing data as consisting of a unit of data and a link to more units of data.

A linked list is useful in the implementations of dynamic arrays, stacks, strings and sets. The linked list is the basic ADT in some languages, for example LISP. LISP stands for List Processing. All the program instructions in LISP operate on lists.

Linked lists are most useful in environments with dynamic memory allocation. With dynamic memory allocation dynamic arrays can grow and shrink with less cost than in a static memory allocation environment. Linked lists are also useful to manage dynamic memory environments. Diagrammatically a linear linked list can be viewed as follows.

![Diagram of a Linked List](image)

Figure 9.1: Diagrammatic Representation of a Linked List

The list consists of data elements. Each data element has an associated link to the next item in the list. The last item in the list has no link. In C++ we can implement this by using a null pointer. The first element of the list is called the head, the last element is called the tail.

Extensions to the ADT include double-linked lists where links exist for the next data and the previous data allowing easier access to data earlier in the list, and sorted linked lists. We use a template class definition so that the linked list can store any kind of data without having to change or reimplement any functionality.

The following class is a C++ implementation of the ADT list. It has methods for creating and destroying a list, copying one list to another (assignment operator), adding items to and removing items from the list (additem, insertitem, removeitem), merging lists (operators for addition), iteration (first, next, last, position, data) and indexing elements (operator[]).
// list.h

#ifndef LIST_HEADER
#define LIST_HEADER

using namespace std;

template <class T>
struct listitem
{
    T data;
    listitem *next;
};

template <class T>
class list
{
    protected:
        listitem<T> *head;
        listitem<T> *current;
        int size;

    public:
        list();
        list(const list&);
        ~list();
        list &operator=(const list&);
        void additem(T);
        void insertitem(T);
        int insertitem(T,int);
        void removeitem(void);
        int removeitem(int);
        list operator+(const list&) const;
        list &operator+=(const list&);
        T operator[](int);
        T data(void);
        int next(void);
        int first(void);
        int last(void);
        int position(int);
        int getsize(void);

};

template <class T>
list<T>::list()
{

head=current=NULL;
size=0;
}

template <class T>
list<T>::list(const list &l)
{
    listitem<T> *li=l.head;
    head=current=NULL;
    size=l.size;
    if(li!=NULL)
    {
        head=new listitem<T>;
        head->data=li->data;
        head->next=NULL;
        current=head;
        li=li->next;
    }
    while(li!=NULL)
    {
        current->next=new listitem<T>;
        current=current->next;
        current->data=li->data;
        current->next=NULL;
        li=li->next;
    }
    current=head;
}

template <class T>
list<T>::~list()
{
    while(head!=NULL)
    {
        current=head;
        head=head->next;
        delete current;
    }
}

template <class T>
list<T> &list<T>::operator=(const list &l)
{
    listitem<T> *li=l.head;
if(this==&l) return *this;

size=l.size;
while(head!=NULL)
{
    current=head;
    head=head->next;
    delete current;
}

head=current=NULL;
if(li!=NULL)
{
    head=new listitem<T>;
    head->data=li->data;
    head->next=NULL;
    current=head;
    li=li->next;
}
while(li!=NULL)
{
    current->next=new listitem<T>;
    current= current->next;
    current->data=li->data;
    current->next=NULL;
    li=li->next;
}
current=head;
return *this;

template <class T>
void list<T>::additem(T t)
{
    listitem<T> *li=head;
    if(head==NULL)
    {
        head=new listitem<T>;
        head->data=t;
        head->next=NULL;
        current=head;
    }
    else
    {
        while(li->next!=NULL) li=li->next;
        li->next=new listitem<T>;
        li->next->data=t;
        li->next->next=NULL;
    }
}
li->next=new listitem<T>;
li=li->next;
li->data=t;
li->next=NULL;
}
size++;
}

template <class T>
void list<T>::insertitem(T t)
{
  listitem<T> *li;
  if(head==NULL)
  {
    head=new listitem<T>;
    head->data=t;
    head->next=NULL;
    current=head;
  }
  else
  {
    li=current->next;
    current->next=new listitem<T>;
    current->next->data=t;
    current->next->next=li;
  }
  size++;
}

template <class T>
int list<T>::insertitem(T t, int i)
{
  int j=0;
  listitem<T> *li1, *li2;
  li1=head;
  while((j<i)&&(li1->next!=NULL)) { li1=li1->next; j++; }
  if(j==i)
  {
    li2=li1->next;
    li1->next=new listitem<T>;
    li1=li1->next;
    li1->data=t;
    li1->next=li2;
    size++;
    return 1;
9.2. LINKED LIST

```cpp
template <class T>
void list<T>::removeItem(void)
{
    listItem<T> *li=head;
    if(head==NULL) return;
    if(head==current)
    {
        delete head;
        head=current=NULL;
        size--;
    }
    if(current!=NULL)
    {
        while(li->next==current) li=li->next;
        li->next=current->next;
        delete current;
        current=li;
        size--;
    }
}

template <class T>
int list<T>::removeItem(int i)
{
    int j=0;
    listItem<T> *li1, *li2;
    li1=head;
    if(head==NULL) return 0;
    while((j<i-1)&&(li1->next!=NULL)) { li1=li1->next; j++; } // Line 154
    if(j==i-1)
    {
        li2=li1->next;
        li1->next=li1->next->next;
        if(li2!=NULL) delete li2;
        size--;
        return 1;
    }
    return 0;
}
```
list<T> list<T>::operator+(const list &l) const
{
    list<T> l2(*this);
    l2+=l;
    return l2;
}

template <class T>
list<T> &list<T>::operator+=(const list &l)
{
    listitem<T> *li=l.head;
    while(li!=NULL) { additem(li->data); li=li->next; } 
    return *this;
}

template <class T>
T list<T>::operator[](int i)
{
    int j=0;
    listitem<T> *li=head;
    if(li==NULL) return T();
    while((j<i)&&(li->next!=NULL)) { li=li->next; j++; } 
    if(j==i) return li->data;
    return T();
}

template <class T>
T list<T>::data(void)
{
    if(current==NULL) return T();
    return current->data;
}

template <class T>
int list<T>::next(void)
{
    if(current->next==NULL) return 0;
    current=current->next;
    return 1;
}

template <class T>
int list<T>::first(void)
{
    current=head;
}
9.2. LINKED LIST

```c++
if(head==NULL) return 0;
return 1;
}

template <class T>
int list<T>::last(void)
{
    current=head;
    if(head==NULL) return 0;
    while(current->next!=NULL) current=current->next;
    return 1;
}

template <class T>
int list<T>::position(int i)
{
    int res;
    res=first();
    while(res&&(i>0)) { next();i--; }
    return res;
}

template <class T>
int list<T>::getsize(void) { return size; }

#endif
```
Now the ADT is used in an example program to illustrate the available operations.

```
// listeg.cpp

#include <iostream>
#include "list.h"

using namespace std;

void main(void)
{
    int i;
    list<int> l1;
    l1.additem(1);
    l1.additem(2);
    l1.additem(3);
    l1.additem(5);
    l1.additem(8);
    list<int> l2(l1), l3;
    l3=l2;
    l1.next();
    l1.insertitem(13); l1.insertitem(21,5);
    l1.first();
    cout<<"l1: ";
do cout<<l1.data()<<" ";
while(l1.next());
cout<<endl;
cout<<"12: ";
do cout<<l2.data()<<" ";
while(l2.next());
cout<<endl;
cout<<"13: ";
do cout<<l3.data()<<" ";
while(l3.next());
cout<<endl;
list<int> l4=l1+l2;
cout<<"14: ";
do cout<<l4.data()<<" ";
while(l4.next());
cout<<endl;
l4+=l3;
l4.first();
cout<<"14: ";
do cout<<l4.data()<<" ";
while(l4.next());
```