A UNIFORM MATHEMATICAL REPRESENTATION OF LOGIC AND COMPUTATION

by

Mahesh M. Bandi

B. E. (Computer Science & Engineering), University of Madras, 1998.

Submitted to the Graduate Faculty of

Department of Electrical Engineering, School of Engineering in partial fulfillment

of the requirements for the degree of

Masters in Electrical Engineering.

University of Pittsburgh

2002

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This dissertation was presented

by

Mahesh M. Bandi

It was defended on

July 03, 2002

and approved by

Prof. Frank Tabakin

Prof. Marlin H. Mickle

Prof. Steven P. Levitan

Prof. James T. Cain
Dissertation Director

ABSTRACT

A UNIFORM MATHEMATICAL REPRESENTATION OF LOGIC AND COMPUTATION

Mahesh M. Bandi , MSEE

University of Pittsburgh, 2002

Current models of computation share varying levels of correspondence with actual implementations. As a result, they can be arranged in a hierarchical structure depending upon their degree of abstraction with respect to physical implementations. Likewise, there exist computing paradigms based upon a different set of defining principles. The classical paradigm involves the scheme of computing as has been applied traditionally. The reversible paradigm requires invertible primitives in order to perform computation, while the paradigm of quantum computing applies the theory of quantum mechanics to perform computational tasks.

A brief analysis concludes that theoretical descriptions at the lowest level of abstraction hierarchy should be uniform across the three paradigms, but the same is not true in case of current descriptions.

We propose a mathematical representation of logic and computation that provides a uniform description across all three paradigms, while making a seamless transition to higher levels of the abstraction hierarchy. This representation, being based upon the theory of linear spaces, is referred to as the *linear representation*. The representation is first developed in the classical context, followed by an extension to the reversible paradigm by exploiting the theory on

invertible mappings. The quantum paradigm is reconciled with this representation through correspondence that unitary operators share with the proposed *linear representation*. The correspondence shared with finite automata models is shown to hold implicitly during development of the proposed representation. Most importantly, the *linear representation* accounts for Hamiltonians that define dynamics of the computational process, thereby resolving the correspondence shared with underlying physical principles.

The consistency of the *linear representation* is checked against a current application in VLSI CAD that exploits linearity of logic functions in symbolic circuit representation. Some possible applications of the *linear representation* to open problems are also discussed.

Keywords: Classical Computation, Reversible Computation, Quantum Computation, Boolean Logic, Universal Turing Machine, Universal Quantum Computer, Mathematical Description of Circuits.

PREFACE

My primary motivation to study computation has been to understand computational processes at the most fundamental level. My quest to find an answer has led through the study of software and hardware, and today I stand at the doorstep of Physics. This document assumes a small role in this lengthy quest, and was born out of a question I asked myself on December 18, 2001 at 0430 hrs. If quantum circuits are linear in structure, then what about classical circuits? This document essentially outlines my search for an answer to this question, and the search itself has been so non-standard that it took me a long time to realize that this is indeed what the community refers to as "research".

As I write this preface, I realize that I have come to look at computation in a manner that is markedly different from the general interpretation attached by the computer scientists and engineers themselves. I look at computation as the process of dynamical evolution of a physical system. The computational process involves information flow as the system continually undergoes transformation during its dynamical evolution. Such an interpretation of computation encompasses every physical process in nature, in fact, evolution of the universe itself is contained in such an interpretation. Such an interpretation to computation has a rather queer characteristic to it, is information an entity that is inseparable from the physical system, is information more fundamental than the elementary particles themselves? It also turns out that I might not be alone in holding such an understanding of information and computation. Ed Fredkin has asked a very similar question in the past in conjunction with his work in Cellular Automata, and it is also in agreement with Lloyd's hypothesis, put forth by Prof. Seth Lloyd, MIT, where he

states that everything that's worth understanding about a complex system, can be understood in terms of how it processes information.

Most of what I document in this thesis is already known to experts, and there might not be anything new contained in terms of ideas or interpretations. What is new, is the discovery of it all I have made by myself. Most of what I state in this thesis is what I have worked out and understood by myself, and to that effect, it has indeed been a most satisfying experience. My requirements have been met many times over, for I believe I might be closer to realizing my primary objective than ever before.

It would be very wrong to give the impression that this thesis is a culmination of one man's effort. I must acknowledge the support, and help provided by many individuals through the years, knowingly or unknowingly, in my arriving at this juncture in life. It is impossible to thank all of them, but they are always remembered with fondness.

- First and foremost, I must thank the two important people who brought me to this world and toiled for their son with no expectations, my mama and appa. Sometimes I wonder, what was their incentive in doing all that they did for me.
- My brothers, S. R. Bandi (Bobbie), and Suryakiran (Billoo), who have always stood by me through sunshine and rain, sharing my happiness when I was happy and lending a shoulder to cry upon when things did not go my way.
- If I could study once again, it was due to the active support of Prof. Cain. He has seen me through good and bad times, and provided the support and encouragement whenever I needed it.

- If Prof. Cain helped me study once again, Prof. Tabakin has helped me continue my studies. I look forward to some good times with him in Physics.
- Prof. Mickle, Prof. Levitan, and Prof. Kourtev have been of immense help at difficult points, when I found myself incapable of working further ahead, and they still continue to help me.
- I thank Prahladh Harsha, Amit Goel, Amit Singhee and Vikas Chandra for access to relevant material and many fruitful discussions.
- Last but not the least, my thanks go out to Prof. Robert B. Griffiths, Otto Stern Professor of Physics at Carnegie Mellon University for always having been there to answer all my questions.

In the past two years at Pittsburgh, I have observed people around me, and learnt from them, in turn motivating me through their actions. I learned discipline at work from Deepak S. Turaga, dedication towards one's work from Amit Itagi. Sujoy Chakravarty taught me the importance of working towards one's goal, rather than sit and romanticize about it. Amit Singhee taught me not to expect anything from anybody, but to be happy if somebody comes by and helps. Last but not the least, Prof. Neil Gershenfeld, Professor at Media Lab, MIT, in whom I have found a guru who can answer questions from the physical as well as computational standpoint. I am indebted to them all, and many more I have not acknowledged. My thanks go out to all of them, the debt may yet be repaid some day, but the gratitude shall remain forever.

This thesis brings to close another phase of my academic life, but there is yet a vast ocean of knowledge to fathom. I hope with passing time, my enthusiasm to learn, understand, and, if possible to contribute to the existing knowledgebase of humanity only increases. This thesis is dedicated to my father Bandi P. V. Sarma, my mother Satyavani Bandi and to my advisor Prof. James T. Cain.

Mahesh M. Bandi

Pittsburgh, PA.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION AND HISTORICAL BACKGROUND

1.1 Introduction

Computation is a functional process from its input to its output. In generic terms its goal is to achieve some result for a given input, through a sequence of functional steps. It is normally defined in terms of the functional result it computes from the inputs. The set of functional steps that characterize the computation of a given problem are collectively referred to as an algorithm. Therefore, an algorithm provides the functional description of a computational process. However, this functional description does not elucidate the actual realization of the computational process itself. Hence, the algorithm may be construed to be an abstract description, devoid of all implementation specific details.

The physical system that realizes an algorithm constitutes the computational system for the given problem. The need for an abstract representation arises due to the different physical implementation schemes made available to the designer to realize the computational system. However, it is possible to provide the functional description of an algorithm with the implementation specific details incorporated within the description itself. Such a description would be quite different for different implementation schemes, though they all represent the same process.

The mathematical or theoretical description of any given physical process is made with the intention to compactly explain that which is already understood about the process, and to also allow for predictions that have not been accounted for. Hence, any domain built upon scientific principles requires a theoretical description that explains the physical process, and the science of computation is no exception.

In discussing computation as a functional process, an extremely generic explanation has been provided so far. In doing so, the various physical schemes in existence, that may be applied to achieve the functional process have not been identified or examined. When taking up the theoretical description of the computational process as a systematic study, it becomes imperative to take into account the physical scheme being applied. This specification becomes imperative, owing to the fact that the description changes from one physical scheme to another. In studying the theoretical descriptions, we will primarily look into the following three schemes of computing:

1) Classical Computing: The traditional form of computing that applies semiconductor based transistor logic to perform computational tasks.
2) Reversible Computing: The scheme of computing that operates on invertible primitives and composition rules that preserve invertibility, the study of which was motivated by an effort to relate the abstract notion of information with the physical principles of thermodynamics.
3) Quantum Computing: The non-standard scheme of computing that applies the theory of quantum mechanics to perform computational tasks.

1.2 Classical Computation

The paradigm of classical computing is the scheme of computing that has been applied traditionally to perform computational tasks. It is referred to as classical computing in order to differentiate it from the more non-standard scheme of quantum computing, which allows for a different set of attributes that characterize the values of the bits. It normally applies semiconductor based transistor logic to realize computational systems using switching circuits developed to perform Boolean operations where the variables take on the binary values of '0' or '1'.

Classical computing was the first computing scheme to be discovered and has been accepted by the user community with phenomenal success. As a result its theoretical descriptions have been studied in great detail by the research community. As a direct fallout of its success, there exist theoretical representations in classical computing that are aimed at studying the functional descriptions of the computational process at various levels of abstraction. Each theoretical model is aimed towards studying a different set of attributes associated with the computational process. The following diagram provides a top-down approach to the various theoretical models that represent the computational process at varying stages of abstraction and complexity, and the level of correspondence they have with the physical implementation of the system they model. As one traverses down this hierarchy, the description attains a closer correspondence with its physical description. Likewise, the description becomes more abstract and further removed from the physical parameters that characterize the system as one moves up the hierarchy of these abstraction levels.

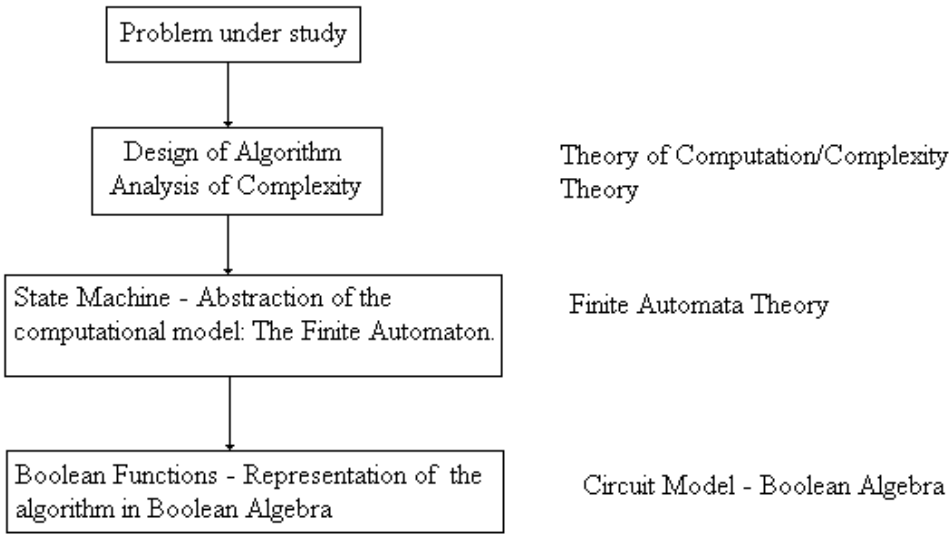


Figure 1.1: Abstraction hierarchy of theoretical descriptions in classical computation.

From Figure 1.1, it is apparent that the theory of computation and computational complexity provides the most abstract description of the computational process, while Boolean algebra forms the closest theoretical description of the physical realization of a computational problem.

In the year 1854, George Boole, an English Mathematician provided the mathematical foundations of symbolic logic [1], better known today as Boolean algebra. It forms the basis for the description of the functional sequence of a computational process in the circuit model. Every elementary logic gate in the circuit model corresponds to an atomic logical function in Boolean algebra. Every step in the functional sequence of an algorithm is comprised of an elementary Boolean function or a composition of elementary Boolean functions.

A level above in the hierarchy of abstraction lies the theory of Finite Automata. Automata theory may be defined as the study of the dynamic behavior of discrete-parameter information systems. The physical form of these discrete parameter systems is not of interest, rather, abstract models consisting of idealized components are developed, whose mathematical behavior closely approximates the properties of the system under investigation. Once a satisfactory model for a given system is developed, the mathematical properties of the model may be employed to study the system's overall behavior. In this manner the mathematical properties common to all discrete parameter systems can be characterized. By combining the results of these studies, the fundamental characteristics that serve to describe the behavior of this class of systems may be identified. Most of the theoretical principles outlined in Finite Automata theory are derived from Discrete Mathematics. Discrete Mathematics is a more abstract form of Boolean algebra. In fact, engineers and designers rely more often on finite automata techniques these days to study the functional characteristics of computing systems even at the circuit level. Switching to the Finite Automata level of abstraction allows them to handle the level of complexity in the system arising as a direct result of the rapid strides made by the industry in packaging an ever larger number of logic elements into computing systems. This marks the transition point between the Circuit model and Finite Automata theory.

The derivation of Finite Automata techniques from Discrete Mathematics is mostly attributed to Alan Turing. In the year 1928, David Hilbert, a celebrated mathematician from Göttingen, had put forth an important question, "Does there exist, an effectively computable solution to mathematical functions that form part of the first order predicate logic?" This is known today as Hilbert's Entscheidungsproblem [2]. It took the ingenuity of Alan Turing at UK

[3], and Alonzo Church at Princeton University, to tackle this problem. Thus was born the abstract model of the Turing Machine, very simple in its structure, but extremely powerful in its capabilities. Following two different approaches (Turing developed the Turing Model through Discrete Mathematics, while Church tackled the problem by applying what is known as lambda calculus) they showed that the answer to Hilbert's question was "No". Prior to this development, Kurt Gödel arrived at the same result through his "Impossiblity Theorems". This led to the now famous conjecture known as the Church-Turing hypothesis, "Every 'function which would naturally be regarded as computable' can be computed by the Universal Turing Machine". Thus was spawned the Theory of Computation and Complexity which forms the basis for determining the computability of a given problem, and the determination of the specific class of problems it belongs to. The Turing model forms the transition point between the Finite Automata theory (in which the Turing model is mathematically described), and the Theory of computation and computational complexity (which applies the Turing model to test the computability of a given problem).

It is seen that there are no clear boundaries that demarcate one theoretical model from another. To account for the blurring of these boundaries, the corrected block description of the hierarchy of theoretical models is presented in Figure 1.2.

The past few decades have led to questions regarding the applicability of seemingly unrelated domains to solve computational problems. This has led to research in non-standard computing schemes motivated by physical laws, for example quantum computing evolved from an answer to the question as to whether quantum mechanics may be applied to achieve

computation, while the proposal for reversible computing schemes arose from studies related to the thermodynamics involved in computational processes.

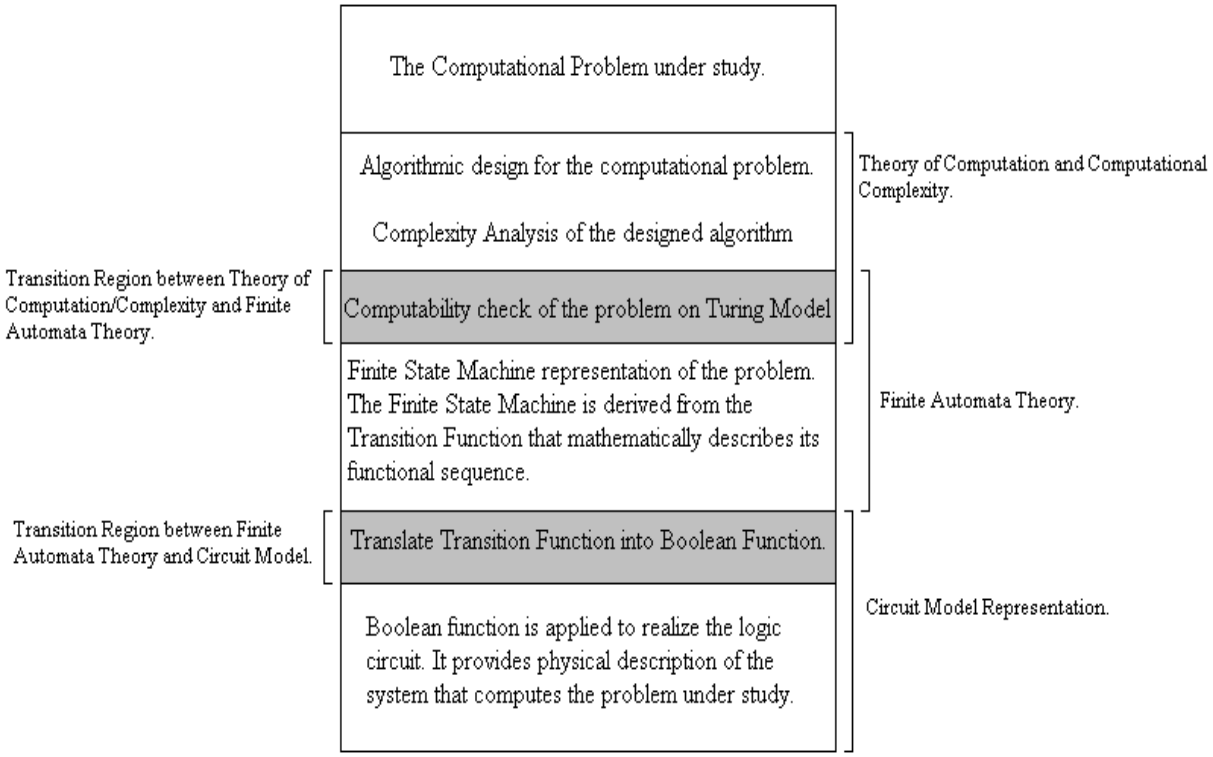
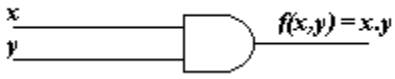


Figure 1.2: Abstraction hierarchy of theoretical abstractions in classical computation.

1.3 Reversible Computing

The theory of reversible computing is based upon invertible primitives and composition rules that preserve invertibility. It involves the capability to reverse a computational process to exactly reconstruct the previous state of the computational process from its current state. For example, consider the ordinary two input AND gate. Two input Boolean variables feed the gate

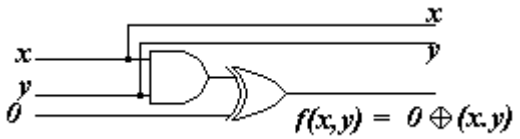
leading to a functional output, based upon the transition rules that are set forth in its truth table as detailed below.



x	*y*	*f(x,y)= x.y*
0	0	0
0	1	0
1	0	0
1	1	1

Figure 1.3: The standard two input AND gate and its corresponding truth table.

Given the output ***f(x,y)*** it is not possible to retrace the computational step performed by this gate to derive the original input variables ***x*** and ***y***. In this specific instance, given that the output ***f(x,y)*** = 1, one knows with certainty that the input values are ***x*** = 1, and ***y*** = 1. But if given that ***f(x,y)*** = 0, one cannot conclusively ascertain if the input values were ***x*** = 0, ***y*** = 0, or ***x*** = 0, ***y*** = 1, or ***x*** = 1, ***y*** = 0. Therefore, there is no unique output for three of the unique input combinations, and it is for this reason that the standard two input AND gate is rendered irreversible. Corresponding to this irreversible two input AND gate, there exists a reversible version that holds the capability to retrace the computational step and derive the values held by the two input variables.



x	*y*	*x,y,f(x,y) =0 ⊕ x.y*
0	0	000
0	1	010
1	0	100
1	1	111

Figure 1.4: The two input reversible AND gate, and the corresponding truth table.

This reversible two input AND gate (the third input with a constant Boolean value of "0" holds no significance functionally, it only helps render the AND gate reversible, and is referred to as the ancilla) is capable of retracing its computational step backwards to realize the two input variables. Please note from the truth table for the reversible AND structure that every unique input string has a unique output string. This is important in order to achieve a reversible circuit.

This alternate paradigm is of primary interest to circuit designers and nanotechnologists since such circuits represent thermodynamically reversible computers. The basis for reversible computing was laid out by Rolf Landauer in 1961 [4]. He proposed that information is a physical quantity, and showed that whenever a physical computer throws away information about its previous state it must generate a corresponding amount of entropy, thus establishing a relationship between the abstract notion of information and the physical principles of thermodynamics. Following upon Landauer's work, in 1973 Charles Bennett proved that the notion of a Reversible Universal Turing Machine is logically consistent and feasible [5]. In

1980, Tomasso Toffoli put forth the fundamental theorem where he proved that for every logic function, there exists an equivalent reversible version [6].

A breakup of the theoretical abstraction hierarchy in the reversible paradigm is presented in Figure 1.5. The algorithmic design for a given computational problem in the reversible paradigm is no different from the one followed in the classical paradigm of computing. The reversible nature of the computational process has a physical significance that is absent at the stage of algorithmic design. Hence, the effects arising due to reversibility start appearing only in the subsequent levels of abstraction. These effects first become apparent when describing transition functions at the finite automata stage of theoretical abstraction. The conditions imposed upon description of the transition function to account for reversibility are, firstly to carry all inputs of the function to the output stage (this ensures that information of the inputs that led to the corresponding outputs is available at the output stage), and secondly to equalize the size of tuples (the no. of bits) that mark the inputs and outputs of the transition function (in Boolean representation it implies the number of input bits should equal the number of output bits). Imposition of these conditions increases the number of inputs and outputs being treated, leading to an increase in the function's complexity. A measure of the added complexity in such functions becomes visible when studying Bennett's construction of the reversible universal Turing machine [3].

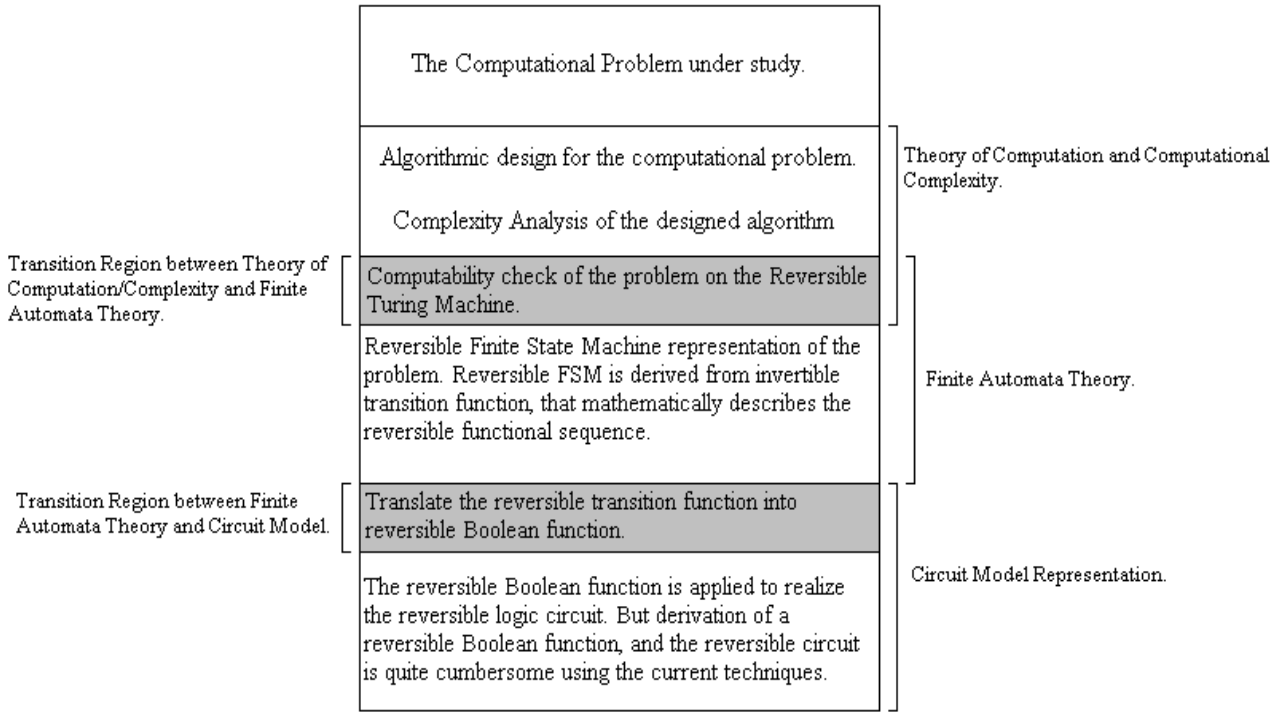


Figure 1.5: Hierarchy of theoretical abstractions in the paradigm of Reversible Computation.

The standard irreversible universal Turing machine is an abstract model with a single tape of infinite length. Bennett's construction of the reversible universal Turing machine requires three such tapes in order to store the information generated during the intermediate stages of the computational process, thus increasing the number of operations to record the positions and the information on the three tapes (he also discusses a reversible realization with a single tape, but also argues that the number of read, write and store operations required would increase by great degree).

The effect of invertible primitives on the theoretical descriptions becomes very obvious when one traverses further down to the Boolean or circuit model representation from the finite automata layer. The representation of reversible circuits also becomes difficult given that Boolean algebra was not developed to account for invertible primitives. Therefore, introduction of invertible primitives in Boolean circuits requires modification of the gates themselves. This gives rise to hybrid gate structures requiring multiple elementary Boolean gates. Moreover, there is no direct correspondence between the Boolean representation of reversible circuits and their actual realization at the transistor level. For instance, the reversible version of the AND gate discussed above does not require an XOR gate in the actual transistor level construction, though the Boolean representation requires an XOR gate in order to render the AND gate reversible. This inconsistency between the Boolean representation and the circuit construction arises due to the inadequacy of the Boolean representation to describe reversible circuits efficiently. With the exception of the inverter, all elementary Boolean gates do not include invertible primitives. They have to be introduced separately in order to render these gate structures reversible.

The paradigm of reversible computing was first studied in the context of the thermodynamics of computational processes. This paradigm assumed greater significance once it was understood that a relationship between classical and quantum computing can be established through the reversible paradigm. This relationship will be treated later in the document, before which the paradigm of quantum computing merits a brief introduction.

1.4 Quantum Computation

Quantum Computation involves the application of the theory of Quantum Mechanics to solve computational problems. It is a very simple statement to make, but a very profound concept. Both physics and computation lie embedded within this paradigm.

The bit is the fundamental entity of information in classical computation. Quantum computation is built around an analogous entity of information, the quantum bit, or qubit for short. Just as a classical bit has a state – 0 or 1 – a qubit also has a state. Two possible states for the qubit are the states $|0\rangle$ and $|1\rangle$, which correspond to the states 0 or 1 for a classical bit. Notation like "$|\ \rangle$" is called the Dirac notation, and it is the standard notation for states in quantum mechanics. It is a short hand notation where $|0\rangle$ corresponds to the column vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $|1\rangle$ corresponds to $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. The difference between bits and qubits is that a qubit can be in a state other than $|0\rangle$ or $|1\rangle$. It is possible to form linear combinations of states, often called superpositions:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle .$$

Put another way, the state of a qubit is a vector of unit length (normalized) in a two-dimensional complex vector space. The special states $|0\rangle$ and $|1\rangle$ are known as computational basis states, and form an orthonormal basis for this vector space. The state $|\psi\rangle$ which is referred to as a qubit in quantum computing, is referred to as a wave function in the theory of quantum mechanics. The physical interpretation attached to the qubit or wave function $|\psi\rangle$ in quantum mechanics is

that it may exist simultaneously in the computational basis states $|0\rangle$ and $|1\rangle$ with some probability. The probability of a qubit being in a particular computational basis state is derived from the complex numbers α and β in the linear combination. Hence they are referred to sometimes as the probability amplitudes of their respective states $|0\rangle$ and $|1\rangle$. α and β constitute components of $|\psi\rangle$ along the states $|0\rangle$ and $|1\rangle$ respectively. The absolute values of these complex coefficients α and β when squared, give the probabilities of the state vector $|\psi\rangle$ being in state $|0\rangle$ with probability $|\alpha|^2$ and in the state $|1\rangle$ with probability $|\beta|^2$. When a qubit is measured, the result 0 is measured with probability $|\alpha|^2$, or the result 1 with probability $|\beta|^2$. Naturally, it turns out that $|\alpha|^2 + |\beta|^2 = 1$, thus explaining the need to impose the normalization condition upon the wave function. One can examine a qubit to determine whether it is in the state $|0\rangle$ or $|1\rangle$. For example, computers do this all the time when they retrieve contents of their memory. Rather remarkably, one cannot examine a qubit to determine its quantum state, that is, the values of α and β. Instead quantum mechanics dictates that one can only acquire much more restricted information about the quantum state. This capability of the qubit to attain superposition states lies at the heart of the dichotomy that differentiates quantum computing schemes from their classical counterparts. The ability of the qubit to be in a superposition state runs counter to the 'common sense' understanding of the physical world.

The single qubit case discussed above can be extended to multiple qubits. Suppose we are given two qubits. If these were classical bits, then there would be four possible states, 00, 01, 10 and 11. Correspondingly, a two qubit system has four computational basis states denoted $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. A pair of qubits can also exist in superpositions of these four states,

so the quantum state of two qubits involves associating a complex coefficient with each computational basis state, such that the state vector describing the two qubits is

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle .$$

Similar to the case of the single qubit, the measurement result x (= 00, 01, 10, 11) occurs with probability $|\alpha_x|^2$, with the state of the qubits after the measurement being $|x\rangle$.

This capability of the qubit to move into a superposition of multiple states leads to a sense of quantum parallelism with no known classical counterparts. This quantum parallelism arises from capability of the qubits to take up all possible combinations of the individual qubit values simultaneously with certain probabilities attached to each value, thus achieving parallelism in the true sense of the term. The following example, illustrates a theoretically feasible application that is achievable only through quantum computation. It combines in a concrete, non-trivial way all the basic ideas of elementary quantum mechanics, and is therefore an ideal example of the information processing capabilities that can be accomplished using quantum mechanics.

Superdense coding involves two parties, conventionally known as 'Alice' and 'Bob', who are a long way away from one another. Their goal is to transmit some classical information from Alice to Bob. Suppose Alice is in possession of two classical bits of information which she wishes to send Bob, but is only allowed to send a single bit to Bob. Can she achieve her goal?

Superdense coding tells us that the answer to this question is yes. Suppose Alice and Bob initially shared a pair of qubits in the entangled[1] state

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Alice is initially in possession of the first qubit, while Bob has possession of the second qubit. Note that $|\psi\rangle$ is a fixed state; there is no need for Alice to have sent Bob any qubits in order to prepare this state. Instead some third party may prepare the entangled state ahead of time, sending one of the qubits to Alice, and the other to Bob.

By sending the single qubit in her possession to Bob, it turns out that Alice can communicate two bits of classical information to Bob. Here is the procedure she uses. If she wishes to send the bit string '00', '01', '10' or '11' to Bob, Alice has to perform corresponding linear transformations on her qubit such that it alters Bob's qubit due to the co-relation that exists between the two qubits arising from entanglement.

Before explaining what linear transformations are performed by Alice, it is important to understand the computational basis states involved in Superdense coding. In this application of quantum computation, the standard basis vectors do not constitute the computational basis

[1] Entanglement is a purely quantum mechanical phenomenon by which two quantum systems that interact with each other, say through collision of two elementary particles that constitute the two qubits in question, continue to share some correlation amongst themselves even after they have been spatially separated. Such a non-local correlation leads to situations where a measurement conducted upon one of the two qubits leads to a deterministic prediction on the value of the second qubit as well. For instance imagine two electrons that constitute two qubits, suppose a mechanism were set up such that they collided with each other and then separated apart spatially. During collision a physical process is undergone through which the two electrons become co-related to each other by some means no matter how far they are separated spatially, such that a change made in the state of one electron alters the state of the second as well in a predictable manner, such a co-relation is referred to as quantum entanglement. A detailed explanation of the process of entanglement is beyond the scope of the current work. A good reference on the quantum computing aspects of entanglement is [7].

vectors. Instead, a different set of basis vectors are chosen. The new set of basis vectors and their corresponding bit strings are shown below:

The bit string '00' is represented by the basis vector $\frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

The bit string '01' is represented by the basis vector $\frac{|00\rangle - |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix}$

The bit string '10' is represented by the basis vector $\frac{|10\rangle + |01\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$

The bit string '11' is represented by the basis vector $\frac{|10\rangle - |01\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix}$

In quantum mechanics these four states are referred to as the Bell basis, Bell states, or EPR pairs. The Bell states form an orthonormal basis, and can therefore be distinguished by appropriate quantum measurement. An interesting property of the Bell basis is that any of the Bell states can be transformed into any other Bell state through simple linear transformations. It is these linear transformations that Alice would have to perform, in order to transmit the relevant information across to Bob. The linear transformations involved are detailed below.

To transmit bit string ‘00’:

Note that the two qubits shared by Alice and Bob are already in the state ‘00’ to start with. Therefore Alice need not do anything to her qubit. This is equivalent to stating that Alice applies the Identity on her qubit.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}.$$

To transmit bit string ‘01’:

In order to transmit the bit string ‘01’, Alice has to apply a transformation such that the state representing ‘00’ is transformed to the state representing ‘01’. This is achieved as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} = \frac{|00\rangle - |11\rangle}{\sqrt{2}}.$$

To transmit bit string ‘10’:

In order to transmit the bit string ‘10’, Alice applies a transformation that takes the state representing ‘00’ to the state representing ‘10’. This is achieved as follows:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \frac{|10\rangle + |01\rangle}{\sqrt{2}}.$$

To transmit bit string '11':

In order to transmit the bit string '11' to Bob, Alice has to apply the transform that takes the state '00' to state '11'. This is achieved as follows:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix} = \frac{|10\rangle - |01\rangle}{\sqrt{2}}.$$

Summarizing, Alice, interacting with only a single qubit, is able to transmit two bits of information to Bob. Of course, two qubits are involved in the protocol, but Alice never need interact with the second qubit. Classically, the task Alice accomplishes would have been impossible had she transmitted only a single classical bit. However, a key point can already be seen in this beautiful example: information is physical, and surprising physical theories such as quantum mechanics may predict surprising information processing capabilities. In fact, it is the purely quantum mechanical attributes of superposition and entanglement that render the unique nature to quantum computing, with no equivalent classical analogs with which to compare them.

The birth of quantum computing itself is attributed to the fact that it is difficult to efficiently simulate quantum mechanical systems by means of traditional computing schemes. In 1982, Richard Feynman, a Nobel prize winning physicist from Caltech, noted that simulation of quantum mechanical systems on computers was exponentially difficult because quantum systems can efficiently create superpositions with exponentially many terms (simply put, if we consider a quantum computing system for instance, comprising of x qubits, with each qubit being a two

level system, where each level corresponds to state $|0\rangle$ and $|1\rangle$ of the qubit, the composite quantum system can exist in a superposition of 2^x states 'simultaneously'[8]. The addition of every new qubit to this system increases the number of superpositions by an exponential quantity). Stemming from this observation, he suggested that it might be worthwhile to develop "quantum computers" for the purpose of directly simulating quantum systems of interest to theoretical physicists. He observed that precisely because they could efficiently create superpositions with exponentially many terms, quantum computers would be capable of a form of "quantum parallelism" as discussed earlier.

Quantum computing really took off with a paper by David Deutsch [9] that is considered a classic in the field of quantum computing. Deutsch re-interpreted the Church-Turing hypothesis in physical terms and came up with its modified version, "Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means". Taking this modified version of the Church-Turing principle as his basis, Deutsch argued that there could conceptually exist a universal quantum computer, operating by principles embodied within the laws of quantum mechanics, which would prove to be more powerful than the universal Turing machine. The generalized quantum computer was represented by a particular class of linear operators known as unitary operators that would render the computational process reversible. In this paper, he also proved that the reversible universal Turing machine formed a special case of the universal quantum computer by drawing upon the relevant proofs provided by Charles Bennett on the feasibility and logical consistency of the reversible universal Turing machine, and Toffoli's fundamental theorem where he proved that for every irreversible logic function in the classical paradigm of computing, there exists a

corresponding functionally equivalent reversible logic function. Deutsch's proof implies that any algorithm realizable in the classical paradigm may be realized in the quantum paradigm as well. Toffoli's fundamental theorem assumes significance here because quantum circuits that solve classical logic functions are basically reversible versions of the classical logic functions.

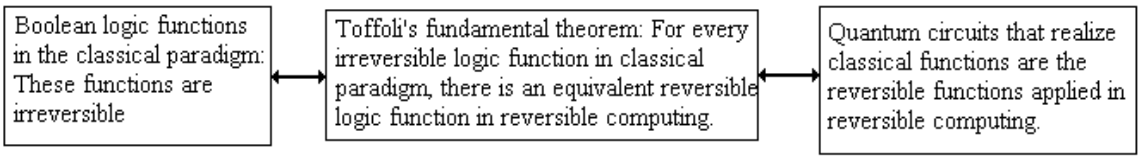


Figure 1.6: The relationship between classical, reversible, and quantum circuits.

At the fundamental level, computational problems solved in quantum computing involve a quantum analog of the circuits applied in classical computing, referred to as quantum circuits. It is known that Boolean algebra is applied to provide the theoretical description of classical circuits, but it cannot provide a description of quantum circuits. The application of quantum mechanical principles to computing gives rise to a special class of circuits with no known classical counterparts. These circuits cannot be described by means of Boolean algebra. Therefore, quantum computing takes recourse to the theory of quantum mechanics to resolve this complication. In quantum mechanics, the time evolution of any quantum mechanical system is defined by the Hamiltonian, which is a linear operator. Since the process of computation in quantum computing is equivalent to describing the time evolution of a quantum mechanical system, quantum circuits are described by means of unitary operators. Therefore, quantum

computing essentially relies on linear algebra to represent quantum circuits. Some elementary quantum gates are listed below.

Consider an arbitrary qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. This state corresponds to the column vector $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$.

Applying the linear transformation $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ to this qubit flips the qubit $|\psi\rangle$ as shown below:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} |\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \beta\,|0\rangle + \alpha\,|1\rangle .$$

This linear transformation is referred to as the bit-flip gate, the quantum NOT gate, and is more famous as the Pauli ***x*** matrix (referred to as $\boldsymbol{\sigma_x}$ also).

Applying the linear transformation $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ flips the sign of $|1\rangle$ to -$|1\rangle$ as shown below:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} |\psi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ -\beta \end{bmatrix} = \alpha|0\rangle - \beta|1\rangle .$$

An important single qubit gate is the Hadamard gate that takes a qubit into superposition states. The Hadamard gate is represented by $\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$. Its action upon a qubit is shown below:

Consider a qubit that is initially in state $|0\rangle$, corresponding to the classical bit with value 0. Applying the Hadamard transformation upon this qubit results in:

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ 1\end{bmatrix} = \frac{1}{\sqrt{2}}\left(\begin{bmatrix}1\\ 0\end{bmatrix}+\begin{bmatrix}0\\ 1\end{bmatrix}\right) = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

This implies, if applied upon a qubit with initial state $|0\rangle$, the Hadamard gate transforms it into a superposition of the states $|0\rangle$ and $|1\rangle$. Now consider the case where the qubit is initially in state $|1\rangle$, corresponding to the classical bit with value 1. Applying the Hadamard transformation upon this qubit results in:

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ -1\end{bmatrix} = \frac{1}{\sqrt{2}}\left(\begin{bmatrix}1\\ 0\end{bmatrix}-\begin{bmatrix}0\\ 1\end{bmatrix}\right) = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

This implies, if applied upon a qubit with initial state $|1\rangle$, the Hadamard gate transforms it into a superposition of the states $|0\rangle$ and $|1\rangle$ with a negative phase attached to state $|1\rangle$. The Hadamard gate is one of the most useful quantum gates since it holds the capability to take a qubit from a classically deterministic state to a superposition state that is purely quantum.

Another important quantum gate is the Controlled NOT or CNOT gate. Its symbolic representation, and the corresponding matrix are shown below.

$$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}$$

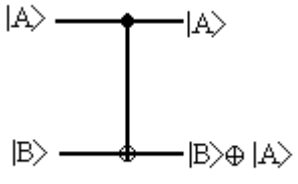


Figure 1.7: The unitary operator and symbolic representation corresponding to CNOT gate.

This circuit has the property that it takes two inputs $|A\rangle$ and $|B\rangle$ and provides the original input $|A\rangle$, and the exclusive OR of $|A\rangle$ and $|B\rangle$.

Before proceeding further, it is important to consider the abstraction hierarchy in the paradigm of quantum computing. The block diagram presented in Figure 8 provides a basic listing of the hierarchy of theoretical abstractions available in the scheme of quantum computing at present. Owing to the fact that this is a very young field, most of the theoretical descriptions are at a fundamental stage of evolution. The study of complexity theory in the context of quantum computing is less than half a decade old, and is currently an area of great curiosity to complexity theorists. The study of quantum complexity is directed towards classifying the set of problems that can be solved exclusively through application of quantum mechanics to computation. Owing to the fact that this domain is at a very elementary stage of evolution, not much is known about problems that fall within quantum complexity classes. Likewise, the first known study on quantum finite automata was conducted by Benioff [10], though not in the context of quantum computing. The study of quantum finite automata in the context of regular expressions and grammar commenced less than two years ago with initial observations being made. Major results and conclusions have yet to be derived from the active research currently underway in this nascent field.

Quantum circuits on the other hand are better understood due to two primary reasons. First, due to studies conducted in reversible circuits in the paradigm of reversible computing, and, second, owing to the fact that these circuits are represented by unitary matrices. Unitary matrices are heavily applied in describing the time evolution of quantum mechanical systems in

physics, and, therefore, their properties have been extensively studied by mathematicians and physicists alike.

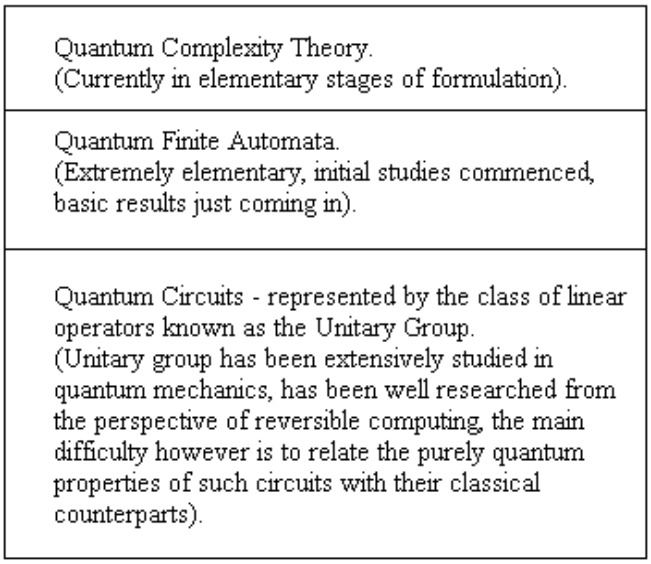


Figure 1.8: Hierarchy of theoretical abstractions in the paradigm of quantum computation.

1.5 Conclusion

We have, thus far, briefly looked into different computing paradigms. The case of classical computing is quite special, where the entire field of semiconductor physics has been spawned to sustain progress in the physical implementation and fabrication techniques of traditional computing systems. On the other hand the paradigm of reversible computing was born out of efforts to relate the abstract notion of information and the process of computation with the principles of thermodynamics. Similarly, quantum computing as a domain was formalized after some unrelated questions led to the same answer. Is computing, as it is presently applied,

complete? Why are certain computational problems exponentially hard to solve, and some completely intractable? What is the ultimate limit of computation? What is the ultimate limit of the transistor size? These and many more questions surprisingly led to answers that all pointed in the same direction, that of quantum mechanics.

The theoretical techniques as applied to computing today, such as Boolean algebra in the circuit model of representation, theory of finite automata, theory of computation and computational complexity etc. were developed specifically to explain a particular paradigm of computing, that of classical computing. The other paradigms of computing were never considered at a time when these theoretical techniques were being designed. As a result, the current theoretical techniques either fall short of, or completely fail to provide a consistent basis that can successfully include, and explain different paradigms of computing. This disparity is obvious from the following tabular analysis of the theoretical models available against the paradigms of computing being considered.

An obvious observation that may be made from the tabular analysis is that the theoretical descriptions at the lowest level of abstraction vary by a great degree from one paradigm to the other. Classical computing applies Boolean algebra to describe circuits at level 1 of the abstraction hierarchy. Reversible computing also applies Boolean algebra, but its unsuitability to the reversible paradigm has been clearly established in Section 1.3. Quantum computing applies linear algebra to describe its circuits. Such a disparity at Level 1 of the theoretical abstraction could be argued to be quite consistent, owing to the fact that these descriptions provide attributes

Abstraction Level 3: Algorithm design and Complexity Analysis.	Theory of Computation & Computational Complexity.	Theory of Computation & Complexity Theory	Quantum Complexity Theory (Evolutionary stage).
Abstraction Level 2: Mathematical Description through Finite Automata.	Finite Automata Theory.	Finite Automata Theory. (known to be cumbersome)	Quantum Finite Automata. (Initial proposals and results)
Abstraction Level 1: Implementation specific theoretical description.	Circuit Model. (applies Boolean Algebra)	Circuit Model - applies Boolean algebra, but very cumbersome.	Linear Operators. (Unitary Group)
	Classical Computing	Reversible Computing	Quantum Computing

Figure 1.9: A tabular analysis of theoretical descriptions in different computing paradigms.

characteristic of their respective physical implementations. These implementations being different from one paradigm to the other, the techniques applied should vary as well. The argument sounds quite logical in itself if one were to consider each paradigm of computing as a separate distinguishable entity. But, should one attempt to consider the scientific study of computation as a unified domain that accounts for all paradigms within one unified setting, the argument seems inconsistent and collapses. Which of the two arguments is the right one to adopt? It is obviously not possible for both the arguments to be consistent and yet be mutually contradictory to each other. It turns out that the second argument is correct, as shall be derived from the arguments to follow in the next chapter.

2.0 STATEMENT OF PROBLEM

2.1 Basis for the Problem

In Chapter 1, a lack of uniformity was noted in the theoretical descriptions for varying computing paradigms at the lowest level of abstraction. The ambiguity arising from the two contradictory arguments was noted to occur primarily due to the non-uniformity amongst current theoretical descriptions of varying paradigms at level 1 of the abstraction hierarchy. It was also stated that the second argument (that the scientific study of computation as a unified domain that accounts for all paradigms within one unified setting) is logically the correct one to adopt. This thesis forms an effort to provide a uniform theoretical description of computational logic at the lowest level of abstraction that remains consistent through all computing paradigms, and yet at the same time can take into account the varying implementation schemes that mark the difference from one paradigm to the other. The logical consistency of the second argument is, therefore, automatically validated during the process of development of this uniform theoretical description. The reasons in favor of the second argument are provided below.

The logical premise behind the validity of the second argument lies in the two physical assertions made by Rolf Landauer and David Deutsch that were discussed briefly in Chapter 1. Rolf Landauer's assertion that "information is physical" implicitly states that information does not exist by itself as an abstract entity, but is required to take on a physical form. The physical form that information takes on is dependent upon the specific implementation scheme being applied. For example, if the designer chooses to employ transistor based logic as the implementation scheme, the information takes on the form of a sequence of electronic pulses.

Similarly, in an optical system, information is identified by the presence or absence of a light pulse. The second assertion by David Deutsch: "every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means", extends Landauer's argument by explaining that not only is information physical, but so is the computational process that acts upon information. Please note that the computational process is quite different from the computational system, the two are quite mutually exclusive. The computational system is the physical system that implements the algorithm, whereas a computational process is the physical process of realizing an algorithm through application of a computational system to generate a functional output.

Going back to the two mutually contradictory arguments presented in Chapter 1, the fallacious argument becomes obvious once the above stated assertions are understood. The assertions above state that information and the computational process are both physical, leading to the premise that no matter what scheme is applied to realize the computational process, the underlying physical principle should be same for all of them. This argument logically leads to the understanding that there ought to exist a uniform theoretical technique that accounts for all descriptions and paradigms at the lowest level of abstraction.

Given that the second argument is correct, it automatically follows that if seemingly disparate domains may be applied to perform computational tasks, there must be exist, attributes common to all these domains that allow them to be suitable candidates to achieve computation. These common attributes would then form the right basis for a uniform theoretical description that transcends all paradigms of computing. The acceptance of this argument renders the tabular

analysis in Figure 1.9 incomplete, since the existing level 1 of theoretical description in the abstraction hierarchy is quite non-uniform across the paradigms being considered. This necessitates a correction in the tabular analysis through introduction of a new level in the abstraction hierarchy. This new level should provide the necessary uniform theoretical description that validates the consistency of the second argument. Being at the lowest level of the abstraction hierarchy, it also comes closest to the physical aspects of the computational process. Please note however, that the physical principles underlying the computational process are far removed from the different physical schemes applied to implement the computational systems. The physical principles define the process of computation itself and are common through all paradigms, and therefore, allow a common representation that we seek. On the other hand, the physical implementations differ from one scheme to the other, and their theoretical descriptions vary accordingly. These descriptions come at a layer below Level 0 of the abstraction hierarchy, where the description cannot be uniform. For example, the device physics applied to semiconductor based computing systems is different from optical computing systems in classical computing and so are the corresponding theoretical descriptions. Similarly, quantum computation may be achieved through quantum dots, nuclear magnetic resonance, or polarized photons, with each scheme having its specific theoretical description.

Abstraction Level 3: Algorithm design and Complexity Analysis.	Theory of Computation & Computational Complexity.	Theory of Computation & Complexity Theory	Quantum Complexity Theory (Evolutionary stage).
Abstraction Level 2: Mathematical Description through Finite Automata.	Finite Automata Theory.	Finite Automata Theory. (known to be cumbersome)	Quantum Finite Automata. (Initial proposals and results)
Abstraction Level 1: Implementation specific theoretical description.	Circuit Model. (applies Boolean Algebra)	Circuit Model - applies Boolean algebra, but very cumbersome.	Linear Operators. (Unitary Group)
Abstraction Level 0: Uniform Theoretical Description of computation across all implementations.	**The Missing Link**		
	Classical Computing	Reversible Computing	Quantum Computing

Figure 2.1: The corrected tabular analysis introduces level 0 in the abstraction hierarchy.

2.2 Problem Definition and Suggested Approach to Solution

The corrected tabular listing in Figure 2.1 defines the problem this thesis proposes to solve. This thesis is an effort to define a theoretical description of logic that uniformly binds the three paradigms of classical, reversible, and quantum computation. In order to provide a logically consistent theoretical description of such nature, the definition of a formal framework of constraints to be satisfied is imperative. These formal set of constraints are enumerated below:

1) In order for the proposed theoretical description to be considered uniform across the three paradigms of computing, it should make a seamless transition from one paradigm to the other, and at the same time account for unique attributes that characterize each paradigm.
2) It should make a similar seamless transition to the higher level of theoretical abstraction immediately above in the abstraction hierarchy as shown in the tabular analysis above.

3) The constructs of any mathematical formalism applied to develop the description must not be violated.
4) The proposed theoretical description lies at the lowest level in the abstraction hierarchy. The physical principles dictating the computational process would have a strong bearing upon the description. Therefore, the physical principles underlying the computational process must be accounted for by the theoretical description.
5) The proposed description must successfully explain that which is currently understood about, and applied in computation in order to prove its validity and consistency. If possible, it should also make experimentally verifiable predictions that have hitherto not been accounted for.

This set of formal constraints actually form the guidelines upon which to base the approach to adopt in the development of the uniform description. The approach adopted to solve the problem outlined above proceeds in the following manner.

1) Owing to the fact that theoretical constructs in classical computing has been extensively studied, we will develop the theoretical description starting with the classical paradigm of computing.
2) The proposed description will be subsequently extended to the reversible paradigm, followed finally by the quantum paradigm of computation. This will help achieve a smooth transition across the three paradigms of interest.
3) In the process of developing the proposed description, adequate interpretations of the formalism adopted will be provided with existing equivalent constructs in the current descriptions. This will help gain an understanding of the process that the formalism

attempts to describe and simultaneously validate its logical consistency with the higher levels of description in the abstraction hierarchy.

4) The consistency of the description will also be checked against the physical principles underlying computational processes in order to validate that the proposed description does indeed fit in at the lowest level of the abstraction hierarchy.
5) The final verification towards logical consistency of the proposed description will be made against a few existing applications within the domain of computation, and through independent validation of important proofs in the respective paradigms of interest.

2.3 Organization of the Document

Chapter 3 provides the necessary theoretical background and definitions for specific topics in the domain of computation, physics and mathematics that will be applied in solving the problem.

Chapter 4 introduces the development of the linear representation of logic in the context of classical computation. The development of this theoretical description will commence with elementary Boolean gates and evolve to include combinational functions.

Chapter 5 will extend the linear representation to the paradigms of reversible and quantum computing, outlining some unique properties of these paradigms in the course of the development of the theoretical description, and also relating the principles of physics underlying the computational process.

Chapter 6 will discuss some existing applications of computation in context of the proposed theoretical description, thereby establishing its validity and logical consistency.

Chapter 7 will discuss some implications of the proposed theoretical description to a few open problems before summarizing the contributions documented in the thesis and concluding the document.

3.0 BACKGROUND AND DEFINITIONS

3.1 Introduction

This chapter introduces the necessary background, definitions, and standard formalism for concepts required in the development of the proposed description. Among the concepts to be discussed in this chapter are the Turing model and the Halting Problem in the context of Finite Automata theory. Euclidean spaces form an important part of the proposed representation. Hence, the structure of Euclidean spaces is reviewed in the context of linear algebra. The paradigm of quantum computation relies heavily upon the theory of quantum mechanics. Therefore, a short treatment of this theory and the short hand Dirac notation associated with it are included.

3.2 Finite Automata Theory

Automata theory may be defined as the study of dynamic behavior of discrete-parameter information systems. The behavior of any computational system can be represented in terms of mathematical relations between three sets of variables that describe the input, the output, and the state of the system. The input set represents those external quantities that can be applied to the system to produce a change in the system's behavior, and the output set represents the possible observable behavior of the system in response to these inputs. One of the basic characteristics of any system is that its current output is a function not only of the current input but also of the past inputs and outputs. Due to this feature, one can think of a system as possessing a "memory",

which stores information about the past behavior of the system. The state set, the third set of variables, is used to represent the amount of information stored by the system.

The response of a system to a given input can be represented by a set of equations that describe the functional relationships between a set of independent and dependent variables. In many systems these functions are describable in terms of integral and differential equations where the variables take on a continuum of values. In automata theory, however, the interest lies in a different class of systems. The systems dealt with are characterized by the fact that all of the variables can only assume discrete values. As would be expected, the functions that describe the behavior of discrete parameter systems can no longer be represented by integral and differential equations. There is a branch of mathematics, referred to as abstract algebra that provides a source of mathematical techniques to describe the operation of discrete parameter systems. The mathematical function that characterizes the operation of the discrete parameter system is a transition function (also referred to sometimes as the transition table). The transition function in automata theory is functionally equivalent to the logic function in the circuit model representation, and shares the same set of transition rules. This is made possible by the fact that the circuit model representation is polynomially equivalent to the automata representation.

The application of finite automata techniques becomes imperative every time a memory element is introduced into the system, thereby introducing time dependency into the mathematical definition of its structure. In the circuit model representation, such systems with time dependency are characterized by sequential networks. They are represented in automata theory by finite-state sequential machines. However, there are many systems of interest that

cannot be adequately or conveniently represented by a finite-state sequential machine. This inadequacy arises from the complexity involved in handling large numbers of inputs, or memory elements. In such cases, the model is reformulated in such a way that the information not currently being processed is stored externally. The actual information processing is then represented by a sequential machine that can call upon this external store of information as needed. This form of a system model is called a Turing machine in honor of Alan M. Turing who first proposed and described the general properties of such a model in 1936 [3]. Turing's primary motivation behind the description of this abstract machine was to define the fundamental relationships involved in making computations. Since his original work, his results have been applied to solve many problems that occur in automata theory.

3.2.1 The Turing Machine

A Turing machine is essentially a finite-state sequential machine that has the ability to communicate with an external store of information. Therefore, a great deal of similarity is observed between the properties of Turing machines and sequential machines. In fact, one of the major elements of a Turing machine is a sequential machine.

A Turing machine is composed of three parts: a control element, a reading and writing head, and an infinite computing tape. The tape, which represents the external information store, is divided into a sequence of squares, each square containing either a blank symbol or a symbol from a finite set $\boldsymbol{A}$. Only one tape square may be scanned at a time by the reading head.

During each cycle of operation the tape square under the reading head is scanned by the control element to determine the symbol printed in the square. After reading this symbol the control element executes one of the four possible moves detailed below:

1) A new symbol may be written in the tape square under the reading head.
2) The reading head is positioned over the square to the right of the current square.
3) The reading head is position over the square to the left of the current square.
4) The operation of the machine is halted.

Since the control element is a finite-state machine, the actual operation performed will of course be influenced by the previous operations performed by the machine. In operation, a finite portion of the tape is prepared with a starting sequence of symbols and the remainder of the tape is left blank. The reading head is placed at a particular starting square and the sequential machine is placed in an initial state. The machine then proceeds to compute in accordance with the rules of operation. If, during the computation, the sequential machine generates a halt command, the computation terminates, and the answer of the computation corresponds to the sequence written on some finite portion of the tape. If the sequential machine never generates a halt command, the computation proceeds indefinitely without stopping. With these introductory ideas, the formal definition of a Turing machine is presented.

Definition: A Turing machine is a system $\boldsymbol{T} = \langle \boldsymbol{I, Q, Z, \delta, \omega, q_0} \rangle$, such that

1. $\boldsymbol{I} = \boldsymbol{A} \vee \boldsymbol{b}$ is a non-empty set of symbols where $\boldsymbol{b}$ is the special blank symbol.
2. $\boldsymbol{Q}$ is a set of states where $\boldsymbol{Q} \wedge \boldsymbol{I} = \phi$.

3. $\boldsymbol{Z} = \boldsymbol{A} \vee \boldsymbol{b} \vee \{r, l, h\}$ is the output set where r and l correspond to the commands to position the reading head over the square to the right or left of the currently scanned square and h is the halt instruction.
4. $\boldsymbol{\delta}$, the next state mapping, is a mapping of $\boldsymbol{I} \times \boldsymbol{Q}$ into $\boldsymbol{Q}$.
5. $\boldsymbol{\omega}$, the output mapping, is the mapping of $\boldsymbol{Q}$ into $\boldsymbol{Z}$.
6. $\boldsymbol{q_0}$ is the initial state of the machine at the start of a computation.

The mappings $\boldsymbol{\delta}$ and $\boldsymbol{\omega}$, which describe the control unit of a Turing machine, can be specified by a transition table (or function) as they are described for a sequential machine. The external behavior of a Turing machine differs, however, from that of a sequential machine in that the computing tape is always under the direct supervision of the control element. The control element of a Turing machine completely determines the computation performed by the machine, and the transition function of the sequential machine that makes up the control element can be thought of as describing the program that guides the computation. Therefore, to specify the control element for any particular problem one must first define an algorithm that indicates the basic operations and the order in which they are to be carried out.

To illustrate the operation of the Turing machine, consider a sample function $\boldsymbol{f}(x_1, x_2) = x_1 + x_2$ where x_1 and x_2 are any non-negative integers. In developing a machine to compute this function, one must indicate how the input and output information is encoded on the computing tape and give an algorithm which describes the steps of the computations. The algorithm is then transformed into a state-table representation of the control element.

Although the set $\boldsymbol{A} = \{0, 1, 2, \ldots, 9\}$ may be used to represent the integer that might appear on the computing tape, it is more convenient to use the set $\boldsymbol{A} = \{1\}$ and a convention to represent the integer n with the tape expression of $n + 1$ consecutive 1's (3 = 1111 is one such expression). To begin the calculation of $\boldsymbol{f}(x_1, x_2) = x_1 + x_2$, it is assumed that (x_1, x_2) is written on the computing tape and that the reading head is scanning the leftmost symbol of x_1. The machine then uses the following algorithm to compute $\boldsymbol{f}(x_1, x_2)$.

Step 1: Move right along the tape until a blank is reached.

Step 2: Write a 1 in place of the blank.

Step 3: Move right along the tape until a blank is reached.

Step 4: Move left along the tape one square.

Step 5: Write a blank.

Step 6: Move left along the tape one square.

Step 7: Write a blank and halt computation.

At the completion of this program the tape expression will correspond to the integer x_1+x_2. Consider for instance, that we are presented with the values $x_1 = 3$, and $x_2 = 2$. They are represented in the adopted convention as $x_1 = 1111$, and $x_2 = 111$. At the start of computation, the tape under the reading head is placed at the start of the first expression, and the expression on the tape reads 1111$\boldsymbol{b}$111$\boldsymbol{b}$. The computation proceeds in the following manner:

Step 1: Move right by four squares up to the end of the first expression i.e. $x_1 = 1111\boldsymbol{b}$. The reading head is placed on the blank square present at the end of first expression.

Step 2: A 1 is written in place of the blank. The complete tape expression now reads 11111111***b***.

Step 3: The reading head reads the expression corresponding to x_2. The reading head is now over the square corresponding to the blank space following the second expression i.e. 11111111***b***.

Step 4: The reading head moves one square to the left.

Step 5: The head erases the 1 in the square and inserts a blank in its place. The complete tape expression now reads 1111111***bb***.

Step 6: The reading head moves one square to the left once again.

Step 7: The head erases the 1 in the square and inserts a blank in its place. The complete tape expression now reads 111111***bbb*** and the computation halts.

The result expected from this computation is $x_1 + x_2 = 3 + 2 = 5$. In the adopted convention the result corresponds to the tape expression 111111 (six consecutive 1's). The expression on the tape when the computation halted was 111111, which coincides with the expected tape expression.

In this manner it is possible to construct a Turing machine that computes a given problem. Please note that the basic structure of the Turing machine does not change by a good degree, it is the control element that loads the representative finite state machine for the given algorithm that changes. Of course, there may be instances where a modified Turing machine may prove more useful to perform certain computational tasks, an example of such machines being multiple tape Turing machines.

3.2.2 Universal Turing Machine and Church-Turing Thesis

Up to now, the primary objective has been to construct a Turing machine that will compute a given function. It is understood that for every computable function, one can specify a Turing machine that computes this function by going through a series of steps dictated by the transition table or function associated with the machine's control element. This statement is based upon a famous conjecture made by Alan Turing and Alonzo Church, commonly referred to as the Church-Turing thesis or the Church-Turing principle, which states, "Every 'function which would naturally be regarded as computable' can be computed by the Universal Turing Machine". This conjecture has been verified time and again by experts against an array of computational problems with great success. This brings us to the definition of the universal Turing machine that the Church-Turing principle refers to. It is now shown that it is possible to define a universal Turing machine that can carry out any computation that can be performed by any specific Turing machine.

A universal Turing machine is not programmed to carry out any specific computation. Instead, it is designed to interpret the information contained on the computing tape. Suppose one wishes to evaluate the function $\boldsymbol{f}(x_1, x_2,..., x_n)$, which is known to be a computable function. If the regular Turing machine were to be employed, it would require the design of a finite-state control element that would process the computing tape starting with $x_1\boldsymbol{b}x_2\boldsymbol{b},...,\boldsymbol{b}x_n$ (where $\boldsymbol{b}$ represents a blank) as the initial tape expression. To accomplish the same result with a universal machine one would encode the transition function associated with the regular Turing machine in a suitable manner and place this encoded information on part of the computing tape used by the universal machine (this portion of the tape expression is called a program). Next the sequence

$x_1\mathbf{b}x_2\mathbf{b},\ldots,\mathbf{b}x_n$ would be placed on the universal machine tape. This sequence is called the data sequence. These two sequences would then form the initial tape expression associated with the universal machine.

The universal machine would then proceed to carry out the desired calculation. First it observes the initial symbol of the data sequence. Then it transfers operation to the program and determines how the data sequence should be modified. Using this information, the machine returns to the program for the next instruction. The universal machine continues in this manner until the computation is completed.

From the above description, it is obvious that a universal Turing machine is nothing more than a simple stored-program computer with an infinite memory.

3.2.3 Solvability and the Halting Problem

Given a set $\boldsymbol{G}$ of r-tuples $\{(x_1, x_2, \ldots, x_r)\}$, one is often interested in determining whether a particular r-tuple $(a_1, a_2, \ldots, a_r)$ is or is not an element of $\boldsymbol{G}$. To solve this problem, a predicate of the form $\boldsymbol{P_G}(x_1,x_2,\ldots,x_r)$ may be defined, which indicates the conditions that must be satisfied by a given r-tuple if it is to be included in the set $\boldsymbol{G}$. If this predicate is to be applied, one must be able to show that it can be evaluated, for an arbitrary r-tuple, in a finite number of steps. Whenever this is true it is said that the decision problem for $\boldsymbol{G}$ is solvable or decidable. Otherwise the problem is unsolvable or undecidable. Therefore, a decision problem is solvable if and only if the predicate $\boldsymbol{P_G}(x_1,x_2,\ldots,x_r)$ is a computable predicate.

One of the basic problems encountered when starting a calculation with a Turing machine is whether the machine will stop after a finite number of steps. Therefore, one would like to be able to solve the following decision problem. Let ***M*** be any Turing machine and let x be the initial tape expression presented to ***M***. Then the problem translates into determining if ***M*** will continue to compute indefinitely or if it will stop after a finite number of steps. This is referred to as the Halting Problem for Turing machines. There is a very important precedent to the Halting Problem. In the year 1928, David Hilbert had posed in his address at the Bologna Mathematical Congress, an important problem. Hilbert's Entscheidungsproblem for first order logic was also raised the same year in the famous textbook by Hilbert and Ackerman, and called "the fundamental problem of mathematical logic" [2]. The problem was to give an algorithm for deciding whether a given formula was a logical consequence (in the semantic sense) of a given (finite) set of premises. Hilbert singled out first order logic for this attention, presumably because it seemed clear that all mathematical reasoning could, in principle, be carried out in this formalism. The general form of the Halting Problem was a proof provided by Alan Turing to show that the answer to Hilbert's Entscheidungsproblem was "No". However, it must be noted that the negative answer to the Entscheidungsproblem was provided by Kurt Gödel a few years earlier in his "Impossibility Theorems" where it was shown that given a system of mathematics based upon a set of axioms it is not possible to formally prove or verify all theorems within the given mathematical system.

Theorem: There is no Turing machine that solves the Halting Problem on all inputs ($\boldsymbol{M}, x$).

Proof: Assume that $\boldsymbol{H}$ is a Turing machine, such that $\boldsymbol{H}(\boldsymbol{M}, x)$ constitutes the result "yes" if $\boldsymbol{M}(x)$ halts and "no" otherwise. Modify $\boldsymbol{H}$ to obtain $\boldsymbol{H}^*$, such that

$$\boldsymbol{H}(\boldsymbol{M}, \boldsymbol{M}) = \text{"yes"} \rightarrow \boldsymbol{H}^*(\boldsymbol{M}) \text{ enters an infinite loop.}$$

$$\boldsymbol{H}(\boldsymbol{M}, \boldsymbol{M}) = \text{"no"} \rightarrow \boldsymbol{H}^*(\boldsymbol{M}) = \text{"yes".}$$

The modification can be achieved easily by replacing a few rules in the transition function of $\boldsymbol{H}$'s finite-state control element. A rule which writes "yes" on the tape and causes $\boldsymbol{H}$ to halt is replaced by a rule that takes the machine into an infinite loop. A rule which writes "no" on the tape and causes $\boldsymbol{H}$ to halt is replaced by a rule that writes "yes" on the tape and then halts $\boldsymbol{H}$. This way, $\boldsymbol{H}^*$ is a twisted version of $\boldsymbol{H}$. Now, does $\boldsymbol{H}^*(\boldsymbol{H}^*)$ halt or not? A contradiction is obtained both ways. Suppose it does halt. This implies that $\boldsymbol{H}(\boldsymbol{H}^*, \boldsymbol{H}^*)$ = "no" so $\boldsymbol{H}^*(\boldsymbol{H}^*)$ does not halt. Instead if $\boldsymbol{H}^*(\boldsymbol{H}^*)$ does halt, it implies $\boldsymbol{H}(\boldsymbol{H}^*, \boldsymbol{H}^*)$ = "yes, so $\boldsymbol{H}^*(\boldsymbol{H}^*)$ does halt.

This interesting proof is testament of the fact that there do exist functions that are not computable. This ends the short treatment on Turing machines and the halting problem.

3.3 Linear Algebra

3.3.1 Euclidean Structure

The basic structure of the Euclidean space is reviewed in the language of vectors. A point **0** is chosen as the origin in a real ***n***-dimensional Euclidean space. The length of any vector ***x*** in this space, denoted ||***x***||, is defined as its distance to the origin.

Consider a Cartesian coordinate system, where $x_1, \ldots, x_n$ denote the Cartesian coordinates of ***x*** in this coordinate system. By repeated use of the Pythagorean theorem, the length of ***x*** may be expressed in terms of its Cartesian coordinates as

$$\| \boldsymbol{x} \| = \sqrt{x_1^2 + \ldots + x_n^2}$$

The scalar product of two vectors ***x*** and ***y***, denoted (***x, y***), is defined by $(\boldsymbol{x}, \boldsymbol{y}) = \sum x_j y_j$

The relationship between the two concepts becomes obvious from the following expression for the length of a vector $\| \boldsymbol{x} \|^2 = (\boldsymbol{x}, \boldsymbol{x})$

This gives rise to the expression for the scalar product of two vectors

$$(\boldsymbol{x}, \boldsymbol{y}) = \|(\boldsymbol{x}+\boldsymbol{y})/2\|^2 + \|(\boldsymbol{x}-\boldsymbol{y})/2\|^2$$

It follows that the scalar product has the same value in all Cartesian coordinate systems. By choosing special coordinate axes, the first one parallel to ***x***, the second so that ***y*** is contained in the plane spanned by the first two axes, the geometric meaning of (***x, y***) is understood.

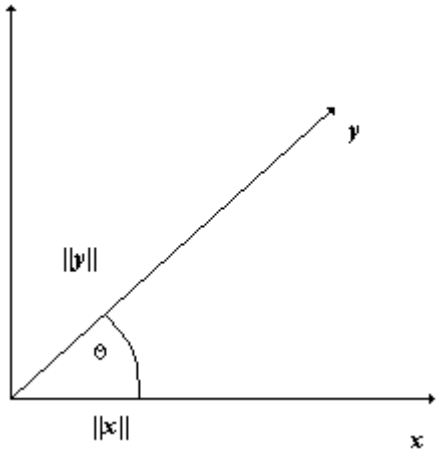


Figure 3.1: Interpretation of the scalar product in Euclidean geometry.

The coordinates of the vector $\boldsymbol{x}$ and $\boldsymbol{y}$ in this coordinate system are $\boldsymbol{x} = (||\boldsymbol{x}||, \mathbf{0}, \ldots, \mathbf{0})$ and $\boldsymbol{y} = (||\boldsymbol{y}||\cos\theta \ldots)$. Therefore

$$(\boldsymbol{x}, \boldsymbol{y}) = ||\boldsymbol{x}||\ ||\boldsymbol{y}||\ \cos\theta,$$

θ being the angle between $\boldsymbol{x}$ and $\boldsymbol{y}$.

Having defined the scalar product, the definition of a Euclidean space follows.

Definition: A Euclidean structure in a linear space $\boldsymbol{X}$ over the reals is furnished by a real valued function of two vector arguments called a scalar product and denoted as $(\boldsymbol{x}, \boldsymbol{y})$, which has the following properties:

1) $(\boldsymbol{x}, \boldsymbol{y})$ is a bilinear function; that is, it is a linear function of each argument when the other is kept fixed.
2) It is symmetric: $(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{y}, \boldsymbol{x})$

3) It is positive: $(\boldsymbol{x}, \boldsymbol{x}) > \mathbf{0}$ except for $\boldsymbol{x} = 0$.

Note that the scalar product satisfies these axioms. Conversely, all Euclidean geometry is contained in these simple axioms. Euclidean length (also called norm) of $\boldsymbol{x}$ is defined as

$$||\boldsymbol{x}|| = (\boldsymbol{x}, \boldsymbol{x})^{1/2}.$$

Note that with this definition of length, it follows from bilinearity and symmetry that

$$||\boldsymbol{x}+\boldsymbol{y}||^2 = ||\boldsymbol{x}||^2 + 2(\boldsymbol{x}, \boldsymbol{y}) + ||\boldsymbol{y}||^2.$$

From this identity, the scalar product of two vectors can be deduced. This is also referred to as the parallelogram law.

Definition: The distance of two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ in a linear space with Euclidean norm is defined by $||\boldsymbol{x} - \boldsymbol{y}||$.

The two following inequalities form important characteristics of the Euclidean structure.

Schwarz Inequality: For all $\boldsymbol{x}, \boldsymbol{y}$, $|(\boldsymbol{x}, \boldsymbol{y})| \leq ||\boldsymbol{x}|| \, ||\boldsymbol{y}||$

Triangle Inequality: For all $\boldsymbol{x}, \boldsymbol{y}$, $||\boldsymbol{x}+\boldsymbol{y}|| \leq ||\boldsymbol{x}|| + ||\boldsymbol{y}||$

Definition: Two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are called orthogonal (perpendicular), denoted as $\boldsymbol{x} \perp \boldsymbol{y}$, if

$$(\boldsymbol{x}, \boldsymbol{y}) = \mathbf{0}.$$

Definition: Let $\boldsymbol{X}$ be a finite dimensional linear space, $\boldsymbol{x^{(1)}, \ldots, x^{(n)}}$ a basis for $\boldsymbol{X}$. This basis is called orthonormal with respect to the given Euclidean structure if

$$(\boldsymbol{x^{(j)}, x^{(k)}}) = \begin{cases} 0, j \neq k \\ 1, j = k \end{cases}.$$

This property is normally represented in the form of the Kronecker-Delta function δ_{jk}, i.e.

$$(x^{(j)}, x^{(k)}) = \delta_{jk}.$$

3.3.2 Self Adjoint Mappings

We now turn to linear mappings of a Euclidean space $\boldsymbol{X}$ into itself. Since a Euclidean space can be identified in a natural way with its own dual, the transpose of a linear map $\mathbf{A}$ of such a space $\boldsymbol{X}$ into itself again maps $\boldsymbol{X}$ into $\boldsymbol{X}$. To indicate this distinction, and for yet another reason explained later, the transpose of a map $\mathbf{A}$ of Euclidean $\boldsymbol{X}$ into $\boldsymbol{X}$ is called the adjoint of $\mathbf{A}$ and is denoted by $\mathbf{A}^*$. When $\boldsymbol{X} = \mathbf{R^n}$, with the standard Euclidean structure and $\mathbf{A}$ a matrix, $\mathbf{A}^*$ is the same as the transpose.

The formal definition of the adjoint of a mapping $\mathbf{A}: \boldsymbol{X} \rightarrow \boldsymbol{X}$ states, given $\mathbf{A}$ is linear and the scalar product bilinear, for given $\boldsymbol{y}$, $\boldsymbol{l}(\boldsymbol{x}) = (\mathbf{A}\boldsymbol{x}, \boldsymbol{y})$ is a linear function of $\boldsymbol{x}$. It is implicitly understood that every linear function $\boldsymbol{l}(\boldsymbol{x})$ can be represented as $(\boldsymbol{x}, \boldsymbol{z})$, $\boldsymbol{z}$ some vector in $\boldsymbol{X}$. Therefore $(\mathbf{A}\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}, \boldsymbol{z})$. The vector $\boldsymbol{z}$ is dependent on $\boldsymbol{y}$. It follows immediately that this dependence is linear. Denote this relation between $\boldsymbol{y}$ and $\boldsymbol{z}$ as $\boldsymbol{z} = \mathbf{A}^*\boldsymbol{y}$. Thus we arrive at the conclusion that $(\mathbf{A}\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}, \mathbf{A}^*\boldsymbol{y})$. This defines the adjoint $\mathbf{A}^*$.

Adjoint mappings possess the following properties:

1) $(\mathbf{A} + \mathbf{B})^* = \mathbf{A}^* + \mathbf{B}^*$
2) $(\mathbf{AB})^* = \mathbf{B}^*\mathbf{A}^*$
3) $(\mathbf{A}^{-1})^* = (\mathbf{A}^*)^{-1}$
4) $(\mathbf{A}^*)^* = \mathbf{A}$

3.3.3 Hermitian Operators

Definition: A matrix $\mathbf{A} = [a_{ij}] \in \mathbf{M_n}$ is said to be Hermitian if $\mathbf{A} = \mathbf{A}^* = \overline{\mathbf{A}}^{\mathbf{T}} = [\overline{a}_{ji}]$. It is skew Hermitian if $\mathbf{A} = -\mathbf{A}^*$, where $\mathbf{M_n}$ is the set of all n x n matrices, and $\mathbf{A}^*$ is the Hermitian complex conjugate and $\overline{\mathbf{A}}$ represents the complex conjugate of **A**.

Hermitian matrices are characterized by the following properties:

Consider two matrices $\mathbf{A, B} \in \mathbf{M_n}$:

1) $\mathbf{A} + \mathbf{A}^*$, $\mathbf{AA}^*$, and $\mathbf{A}^*\mathbf{A}$ are all Hermitian for all $\mathbf{A} \in \mathbf{M_n}$.
2) If **A** is Hermitian, then $\mathbf{A}^{\mathbf{k}}$ is Hermitian for all $\mathbf{k} = 1, 2, 3, \ldots$ If **A** is nonsingular as well, then $\mathbf{A}^{-1}$ is Hermitian.
3) If **A, B** are Hermitian, then $a\mathbf{A} + b\mathbf{B}$ is Hermitian for all real scalars a, b.
4) $\mathbf{A} - \mathbf{A}^*$ is skew Hermitian for all $\mathbf{A} \in \mathbf{M_n}$.
5) If **A, B** are skew Hermitian, then $a\mathbf{A} + b\mathbf{B}$ is skew Hermitian for all real scalars a, b.
6) If **A** is Hermitian, then $i\mathbf{A}$ skew Hermitian.
7) If **A** is skew Hermitian, then $i\mathbf{A}$ Hermitian.
8) Any $\mathbf{A} \in \mathbf{M_n}$ can be written as

$\mathbf{A} = \frac{1}{2}(\mathbf{A} + \mathbf{A}^*) + \frac{1}{2}(\mathbf{A} - \mathbf{A}^*) = \mathbf{H(A)} + \mathbf{S(A)}$, where **H(A)** is the Hermitian part of **A**, and **S(A)** is the skew Hermitian part of **A**.

9) If **A** is Hermitian, the main diagonal entries of **A** are all real. In order to specify the n^2 elements of **A** one may specify freely any n numbers (for the main diagonal entries) and any $1/2n(n-1)$ complex numbers (for the off-diagonal entries).

Please note that generally matrices that satisfy the definition above for the real Euclidean space are referred to as Symmetric matrices. Hermitian matrices are matrices satisfying the definition in complex Euclidean space. Given the fact, that the real space is a subspace of the complex Euclidean space, Symmetric matrices form a subset of the set of Hermitian matrices.

3.3.4 Isometric Mappings

What mappings **M** of a Euclidean space into itself preserve the distance of any pair of points, that is, satisfy for all $\boldsymbol{x}, \boldsymbol{y}$, $\| \mathbf{M}(\boldsymbol{x}) - \mathbf{M}(\boldsymbol{y}) \| = \|\boldsymbol{x} - \boldsymbol{y}\|$?

Such a mapping is called an Isometry. It is obvious from the definition that the composite of two isometries is an isometry. An elementary example of an isometry is translation:

$$\mathbf{M}(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{a}, \ \boldsymbol{a} \text{ some fixed vector.}$$

Given an isometry, one can compose it with a translation and produce an isometry that maps zero to zero. Conversely, any isometry is the composite of one that maps zero to zero and a translation.

Properties of Isometric mappings:

Let **M** be an isometric mapping of a Euclidean space into itself that maps zero to zero,

$$\mathbf{M}(\mathbf{0}) = \mathbf{0}$$

Then

(i) **M** is linear

(ii) $\mathbf{M}^* \mathbf{M} = \mathbf{I}$ (conversely if this is satisfied, then **M** is an isometry)

(iii) Det $\mathbf{M} = \pm 1$.

Definition: A matrix that maps $\mathbf{R}^n$ into itself isometrically is called orthogonal.

The orthogonal matrices of a given order form a group under matrix multiplication. Clearly, composites of isometries are isometric, and so are their inverses. The orthogonal matrices whose determinant is +1 form a subgroup, called the special orthogonal group. Examples of orthogonal matrices with determinant +1 in three-dimensional space are rotations. A matrix **M** is orthogonal if and only if its columns are unit vectors that are pairwise orthogonal and vice versa, that is, a matrix **M** is orthogonal if and only if its rows are unit vectors and are pairwise orthogonal. An orthogonal matrix **M** is called a permutation operator if every row and column in the matrix has only one non-zero element of value 1, with all other elements being 0. From this it is inferred that the placement of rows and columns of a permutation matrix are permutations of the identity **I**, hence the name permutation matrix.

3.3.5 Complex Euclidean Structure and Unitary Operators

We conclude this section with a brief discussion of complex Euclidean structure. In the concrete definition of complex Euclidean space, the definition of the scalar product in $\mathbf{R}^n$ has to be replaced by $\mathbf{C}^n$ by

$$(\boldsymbol{x}, \boldsymbol{y}) = \sum x_i \overline{y_i},$$

where the bar $\overline{\ }$ denotes the complex conjugate. The definition of the adjoint of a matrix remains the same, but in the complex case has a slightly different interpretation. Writing

$$\mathbf{A} = (\boldsymbol{a_{ij}}), \qquad (\mathbf{A}\boldsymbol{x})_i = \sum_j a_{ij} x_j$$

Applying the standard notation this may be written as:

$$(\mathbf{A}\boldsymbol{x}, \boldsymbol{y}) = \sum_i \left(\sum_j a_{ij} x_j \right) \overline{y_i},$$

which may be alternatively re-written as follows:

$$\sum_j x_j \left(\overline{\sum \overline{a_{ij}} y_i} \right),$$

in turn unequivocally establishing that $(\mathbf{A}^* \boldsymbol{y})_j = \sum_j \overline{a_{ij}} y_i$ that is, the adjoint $\mathbf{A}^*$ of the matrix $\mathbf{A}$ is the complex conjugate of the transpose of $\mathbf{A}$.

The definition of the complex Euclidean space follows from this.

Definition: A complex Euclidean structure in a linear space $\boldsymbol{X}$ over the complex numbers is furnished by a complex valued function of two vector arguments, called a scalar product $(\boldsymbol{x}, \boldsymbol{y})$, with the following properties:

(i) $(\boldsymbol{x}, \boldsymbol{y})$ is a linear function of $\boldsymbol{x}$ for $\boldsymbol{y}$ fixed.

(ii) Skew symmetry: for all $\boldsymbol{x}, \boldsymbol{y}$,

$$\overline{(\boldsymbol{x}, \boldsymbol{y})} = (\boldsymbol{y}, \boldsymbol{x})$$

Note that skew symmetry implies that $(\boldsymbol{x}, \boldsymbol{x})$ is real for all $\boldsymbol{x}$.

(iii) Positivity:

$$(\boldsymbol{x}, \boldsymbol{x}) > \mathbf{0} \text{ for all } \boldsymbol{x} \neq \mathbf{0}.$$

The theory of complex Euclidean spaces is analogous to that for real ones, with a few changes where necessary. For example, it follows from (i) and (ii) in the definition above that for $\boldsymbol{x}$ fixed, $(\boldsymbol{x}, \boldsymbol{y})$ is a skew linear function of $\boldsymbol{y}$, that is, additive in $\boldsymbol{y}$ and satisfying for any complex number $\mathbf{k}$,

$$(\boldsymbol{x}, \mathbf{k}\boldsymbol{y}) = \bar{\mathbf{k}}\,(\boldsymbol{x}, \boldsymbol{y}).$$

Instead of repeating the theory, those places are indicated where a slight change is needed. In the complex case:

$$||\boldsymbol{x}+\boldsymbol{y}||^2 = ||\boldsymbol{x}||^2 + (\boldsymbol{x}, \boldsymbol{y}) + (\boldsymbol{y}, \boldsymbol{x}) + ||\boldsymbol{y}||^2$$

$$= ||\boldsymbol{x}||^2 + 2\mathbf{Re}(\boldsymbol{x}, \boldsymbol{y}) + ||\boldsymbol{y}||^2$$

where $\mathbf{Re}(\mathbf{k})$ denotes the real part of the complex number $\mathbf{k}$.

Definition: A linear map of a complex Euclidean space into itself that is isometric is called unitary.

A unitary map **M** satisfies the relations $\mathbf{M}^{*}\ \mathbf{M} = \mathbf{I}$, and conversely, that every map **M** that satisfies this relationship is unitary. If a linear map **M** is unitary, then so are $\mathbf{M}^{-1}$ and $\mathbf{M}^{*}$. The unitary maps form a group under multiplication and the determinant of a unitary map **M** is $|\det\mathbf{M}| = 1$.

This concludes a brief discussion on the Euclidean structure of linear spaces.

3.4 Quantum Mechanics

3.4.1 Introduction

In the study of physical systems, the design of theoretical techniques becomes imperative in order to study their abstract characteristics. Any such theory that attempts to embody the characteristics of the physical system must take into account the distinction between objective reality, which is independent of the theory, and the physical concepts with which the theory operates. These concepts are intended to correspond with the objective reality. Therefore, the essential function of physical theory is to make quantitative predictions about experiments. Such a description, reconciliation of observed behavior, and predictions regarding the structure and dynamical evolution of physical systems at the microscopic level of atoms and elementary particles is provided by the theory of quantum mechanics.

3.4.2 Basic Postulates of Quantum Mechanics

At the core of this theory lie three fundamental postulates that define the structure of quantum mechanics.

1) Representation of quantum mechanical systems: Physical states in the theory of quantum mechanics are represented by vectors, Ψ, in a complex Hilbert space. By Hilbert space, it is implied that the complex space in question is complete and dense. In simpler terms it implies that the entire space can be spanned by the set of basis vectors that define the space. In quantum mechanics the complex Euclidean space represented by square integrable functions constitutes the Hilbert space.
2) Dynamical evolution of quantum mechanical systems: The dynamics of physical systems are specified by the Hermitian operator **H**, where **H** constitutes the Hamiltonian of the system. The time-evolution is described by the Schrödinger Equation $\boldsymbol{i}\hbar\,\boldsymbol{\Psi} = \mathbf{H}\,\boldsymbol{\Psi}$. The term $\boldsymbol{i}$ represents the imaginary number and $\hbar = h/2\pi$ and $h \cong 6.6261 \times 10^{-34}$ [J sec] is Planck's constant.
3) The measurement postulate: The mutually exclusive measurement outcomes are determined through orthogonal projection operators $\{\mathbf{P}_0, \mathbf{P}_1, \ldots\}$. The probability of a particular outcome $\boldsymbol{i}$ is given by $|\mathbf{P}_i\,\boldsymbol{\Psi}|^2$.

3.4.3 Dirac Notation for Quantum States

The theory of quantum mechanics relies heavily upon the theory of linear spaces to describe physical systems. However, directly applying the mathematical formalism of linear algebra proves to be cumbersome. Therefore, a short hand notation known as the Dirac notation (developed originally by Paul A. M. Dirac) is adopted to represent the mathematical formalism

involved in quantum mechanics. This section develops the Dirac formalism of quantum mechanics.

Physical states: ket and bra

Physical states may be written as 'ket', where the ket constitutes a directional vector in a linear space

$$|\psi_x\rangle \rightarrow \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix},$$

or 'bra', which forms the dual of the ket. For every ket, there exists a corresponding bra vector.

$$\langle\psi_x| \rightarrow [x_0^* \;\; x_1^* \;\; x_2^*].$$

Please note that a bra or a ket may have any number of elements corresponding to the dimensionality of the state space in question. The use of three elements in the representation above serves only as an example.

Bras and kets share a unique one-to-one correspondence and are related through the operation of Hermitian conjugation as shown below:

$$|\psi_x\rangle = (\langle\psi_x|)^\dagger, \qquad \langle\psi_x| = (|\psi_x\rangle)^\dagger.$$

By convention, state vectors are assumed to be normalized: $\sum_i |a_i|^2 = 1$. This condition is referred to as the normalization condition.

The inner product of a bra and a ket is a complex number:

$$\langle \psi_x | \, | \psi_y \rangle = \begin{bmatrix} x_0^* & x_1^* & x_2^* \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = x_0^* y_0 + x_1^* y_1 + x_2^* y_2.$$

In usual notation one typically drops one of the vertical bars and writes $\langle \psi_x \, | \psi_y \rangle$. Normalized states satisfy $\langle \psi \, | \psi \rangle = 1$.

Basis

Definition: The definition of a basis from elementary linear algebra follows:

Vectors $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$ belonging to a linear space $\boldsymbol{S}$ are said to form a basis of $\boldsymbol{S}$ if:

a) Every vector in $\boldsymbol{S}$ may be represented as a linear combination of $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$.

b) $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$ are linearly independent.

Note that (a) and (b) above are equivalent to the statement that every vector in $\boldsymbol{S}$ is uniquely expressible as a linear combination of $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$. There are a countably infinite set of bases for any given linear space.

It is often convenient to work with basis kets or bras. For example,

$$|\psi_x\rangle \rightarrow x_0|0\rangle + x_1|1\rangle + x_2|2\rangle$$

$$|0\rangle \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, |1\rangle \rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, |2\rangle \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

$$\langle \psi_a | = a_0^* \langle 0| + a_1^* \langle 1| + a_2^* \langle 2|,$$

where all the basis kets (and bras) are mutually orthonormal, and, therefore, adhere to the Kronecker-Delta condition $\langle i \,|\, j \rangle = \boldsymbol{\delta}_{\mathrm{ij}}$.

Hence

$$(a_0|0\rangle + a_2|2\rangle)(b_0^*\langle 0| + b_1^*\langle 1|) = a_0b_0^*|0\rangle\langle 0| + a_2b_0^*|2\rangle\langle 0| + a_0b_1^*|0\rangle\langle 1| + a_2b_1^*|2\rangle\langle 1|$$

$$\rightarrow \begin{bmatrix} a_0b_0^* & a_0b_1^* & 0 \\ 0 & 0 & 0 \\ a_2b_0^* & a_2b_1^* & 0 \end{bmatrix}.$$

As detailed in the definition of a basis, the same physical state $|\psi_a\rangle$ can be expressed in (uncountably) many different bases. For example, if

$$|\psi_a\rangle = a_0|0\rangle + a_1|1\rangle,$$

in the original basis defined by $\{|0\rangle, |1\rangle\}$.

Then in the new basis $\{|x\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle),\ |y\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\}$,

$$|\psi_a\rangle = \frac{1}{\sqrt{2}}(a_0 + a_1)\,|x\rangle + \frac{1}{\sqrt{2}}(a_0 - a_1)\,|y\rangle$$

Linear Operators

The outer product of a ket and a bra is a linear operator (matrix):

$$|\psi_a\rangle\langle\psi_b| = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \begin{bmatrix} b_0^* & b_1^* & b_2^* \end{bmatrix} = \begin{bmatrix} a_0b_0^* & a_0b_1^* & a_0b_2^* \\ a_1b_0^* & a_1b_1^* & a_1b_2^* \\ a_2b_0^* & a_2b_1^* & a_2b_2^* \end{bmatrix}.$$

Hermitian conjugation of operators in Dirac notation is represented by $(|a\rangle\langle b|)^\dagger = |b\rangle\langle a|$.

Operators act on kets from the left and on bras from the right. Using orthonormal basis vectors, it's easy to compute the results. For example,

$$\mathbf{O} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

$$\mathbf{O}|\psi_a\rangle = (|0\rangle\langle 1| + |1\rangle\langle 0|)(a_0|0\rangle + a_1|1\rangle + a_2|2\rangle)$$

$$= a_1|0\rangle + a_0|1\rangle,$$

$$\langle\psi_a|\mathbf{O} = (a_0^*\langle 0| + a_1^*\langle 1| + a_2^*\langle 2|)(|0\rangle\langle 1| + |1\rangle\langle 0|)$$

$$= a_0^*\langle 1| + a_1^*\langle 0|.$$

The outer product of any vector with itself is a projection operator. Projection operators are generally referred to as "dyads" in the Dirac formalism.

$$|\psi\rangle\langle\psi| = \mathbf{P}_\Psi,$$

$$(\mathbf{P}_\Psi)^2 = |\psi\rangle\langle\psi|\psi\rangle\langle\psi| = \mathbf{P}_\Psi.$$

3.4.4 The Schrödinger Equation

The dynamics of a quantum system is specified by a Hermitian operator **H**, called the Hamiltonian. Time-evolution of quantum systems is given by the Schrödinger Equation,

$$i\hbar\frac{d}{dt}|\psi\rangle = \mathbf{H}|\psi\rangle,$$

where $\hbar = h/2\pi$ and $h \cong 6.6261 \times 10^{-34}$ [J sec] is Planck's constant

For finite-dimensional systems the Schrödinger equation is a coupled system of linear ordinary differential equations. If the physical system is truly isolated (autonomous), then **H** must be constant and we may write the formal solution

$$|\psi(t)\rangle = \exp[\frac{-i}{\hbar}\mathbf{H}t]|\psi(0)\rangle$$

In some cases it is actually possible to compute the operator exponential, which is defined via Taylor expansion:

$$\exp[i\alpha\mathbf{O}] = 1 + i\alpha\mathbf{O} - (\alpha^2/2)\mathbf{O}^2 - i(\alpha^3/3!)\mathbf{O}^3 + (\alpha^4/4!)\mathbf{O}^4 + \ldots$$

Here α is an arbitrary (real) scalar.

Note that if **O** is a Hermitian operator,

$$(\exp[i\alpha\mathbf{O}])^\dagger = 1 - i\alpha\mathbf{O} - (\alpha^2/2)\mathbf{O}^2 + i(\alpha^3/3!)\mathbf{O}^3 + (\alpha^4/4!)\mathbf{O}^4 + \ldots$$

and

$$\exp[i\alpha\mathbf{O}]\,(\exp[i\alpha\mathbf{O}])^\dagger = (\exp[i\alpha\mathbf{O}])^\dagger \exp[i\alpha\mathbf{O}] = 1.$$

That is, $\exp[i\alpha\mathbf{O}]$ is a unitary operator. This is the same unitary operator that was studied in the context of isometric transformations in complex Euclidean spaces in Section 3.3.4.

In the case of the Schrödinger Equation, we write

$$\mathbf{U}(t_2,, t_1) = \exp[\frac{-i}{\hbar}\mathbf{H}(t_2,- t_1)]$$

and refer to $\mathbf{U}(t_2,, t_1)$ as the system's unitary "propagator" or "time development operator" from time t_1 to t_2.

Note that $(\mathbf{U}(t_2,, t_1))^{-1} = (\mathbf{U}(t_2,, t_1))^{\dagger} \sim \mathbf{U}(t_1,, t_2)$ can be thought of as an operator that evolves a state backwards in time

Recall from Section 3.3.4 that unitary operators are isometric transformations in complex Euclidean space, and may be thought of as the complex generalization of rotation operators in a real vector space. Hence, quantum evolution for an isolated system corresponds to a "rigid rotation" of the state space. As a consequence, time evolution preserves the norms of individual state vectors, and preserves the inner product (angle) between arbitrary pairs of state vectors.
Note that by taking the Hermitian conjugate of the entire Schrödinger Equation, we get a time evolution for bras:

$$-i\hbar \frac{d}{dt} \langle \psi | = \langle \psi | \mathbf{H},$$

$$\langle \psi(t_2) | = \langle \psi(t_1) | \mathbf{T}(t_1, t_2).$$

Accordingly,

$$\langle \psi_a(t_2) | \psi_b(t_2) \rangle = \langle \psi_a(t_1) | \mathbf{T}(t_1, t_2) \mathbf{T}(t_2, t_1) | \psi_b(t_1) \rangle$$

$$= \langle \psi_a(t_1) | \psi_b(t_1) \rangle,$$

as noted above.

3.4.5 The Measurement Postulate

Traditional quantum measurement theory deals with orthogonal projection measurements. The effect of an ideal orthogonal measurement is to determine which of a given set of mutually-exclusive propositions is true. In quantum measurement theory, mutually-exclusive propositions correspond to orthogonal projectors (projection operators) on the system state space. Two projectors $\mathbf{P_a}$, $\mathbf{P_b}$ are orthogonal if

$$\mathbf{P_a P_b} |\psi\rangle = 0$$

for every state $|\psi\rangle$ in the Hilbert space. For simplicity, one often writes $\mathbf{P_a P_b} = 0$.

A "complete" or "exhaustive" set of propositions is a set for which at least one proposition must be true. Correspondingly, we define a complete set of orthogonal projectors to be a set $\{ \mathbf{P_0}, \mathbf{P_1}, \mathbf{P_2}, \ldots \}$ such that

$$\sum_i \mathbf{P}_i = 1.$$

Note that by this definition, the number of projectors in a complete orthogonal set must be $\leq$ the dimension of the Hilbert space.

The postulate:

A complete set of orthogonal projectors specifies an exhaustive measurement. For a system prepared in state $|\psi\rangle$, the probability of the i^{th} outcome is given by

$$\mathbf{Pr}(i) = |\mathbf{P}_i |\psi\rangle|^2$$

$$= (\mathbf{P}_i |\psi\rangle)^\dagger \, \mathbf{P}_i |\psi\rangle$$

$$= \langle\psi| \mathbf{P}_i^\dagger \mathbf{P}_i |\psi\rangle$$

$$= \langle \psi | (\mathbf{P}_i^2) | \psi \rangle$$

$$= \langle \psi | \mathbf{P}_i | \psi \rangle$$

Note that $\sum_i \Pr(i) = 1$ by completeness of the set of projectors.

Furthermore, the state of the system after the outcome i has been obtained is given by

$$|\psi\rangle \rightarrow \frac{\mathbf{P}_i |\psi\rangle}{\sqrt{\langle \psi | \mathbf{P}_i | \psi \rangle}}$$

It must, however, be noted that while every complete set of orthogonal projectors specifies a performable measurement, not every performable measurement corresponds to an orthogonal set of projectors. Likewise, more general post-measurement states than shown above are possible.

3.4.6 Physical Observables

Every physically-meaningful quantity, $\boldsymbol{q}$ (energy, position, component of spin etc.), is represented by a Hermitian operator $\mathbf{O}_{\boldsymbol{q}}$. Such operators are typically called "observables". An observable specifies an exhaustive measurement via its spectral decomposition:

$$\mathbf{O}_{\boldsymbol{q}} = \sum_i \lambda_{i_q} \mathbf{P}_{i_q},$$

where $\mathbf{P}_{\boldsymbol{i}}^{\mathbf{q}} = |i_q\rangle \langle i_q|$, and λ_i is the ith eigenvalue of $\mathbf{O}_{\boldsymbol{q}}$ and $|i_q\rangle$ is the corresponding eigenvector defined by the relation $\mathbf{O}_{\boldsymbol{q}} |i_q\rangle = \lambda_i |i_q\rangle$.

In the case of a degenerate eigenvalue λ_i, let $\mathbf{P}_{\boldsymbol{i}}^{\mathbf{q}}$ be the projector into the corresponding subspace.

Note that $\mathbf{O}_q$ is Hermitian, all the eigenvalues are real, the $\{\mathbf{P}_i^{\mathbf{q}}\}$ are orthogonal, and $\sum_i \mathbf{P}_i^{\mathbf{q}} = \mathbf{1}$. It is customary to speak of "measuring" the observable $\mathbf{O}_q$, by which one implies measuring the set $\{\mathbf{P}_i^{\mathbf{q}}\}$ and associating the value $\boldsymbol{q} = \lambda_i^{\,q}$ with the occurrence of the ith outcome. Note that the expected (average) result for a state $|\psi\rangle$ is then given by

$$\langle \boldsymbol{q} \rangle = \sum_i \lambda_{i_q} \langle \psi | \mathbf{P} i_q | \psi \rangle$$

$$= \langle \psi | \mathbf{O}_q | \psi \rangle$$

$$= \langle \mathbf{O}_q \rangle .$$

This concludes a basic introduction to the theory of quantum mechanics and its associated mathematical formalism.

4.0 THE LINEAR REPRESENTATION: CLASSICAL COMPUTATION

4.1 Introduction

Discrete Mathematics, Abstract Algebra and Finite Automata Theory have traditionally formed the theoretical bases for the study of computational models. These theoretical techniques are found to be inadequate at representing models in the paradigms of reversible and quantum computing. This chapter marks the development of the representation being proposed in the classical context, with extensions to the reversible and quantum paradigms following in the next chapter.

The development of the proposed representation, and further analysis of its applications, employs concepts from the domains of mathematics, computer engineering, and physics. Some of the terms used here have different nomenclature in their specific domains. Likewise their interpretation is found to vary from one domain to the other. These distinctions in interpretation will be made clear wherever the confusion is likely to arise. It is important to understand the distinctions and the common aspects that these terms share. They form the defining points that help enmesh all domains to form the superstructure of the theoretical description to be developed.

4.2 The Classical Bit: A Two Dimensional Complex Euclidean Space.

Consider a complex vector space with Euclidean structure (by Euclidean structure it is implied that the vector space in question is a bonafide inner product space). This linear space forms a mathematical model that may be applied to represent information, provided it strictly operates within certain well defined rules. These rules constitute the specific attributes that characterize the bit; therefore it becomes imperative that the vector space in question operate within the bounds of these conditions in order to successfully represent the classical bit (the classical bit constitutes the binary digit as represented traditionally, unlike the quantum bit or "qubit" which has different properties). The conditions imposed upon the vector space are enumerated below:

1) The bit is represented in a two dimensional complex Euclidean space.
2) The bit value "0" is represented by the x-axis or unit column vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, and the bit value "1" is represented by the y-axis or unit column vector $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. These vectors are generally referred to as the standard basis by physicists and mathematicians. In the context of quantum computation, these vectors are also referred to as the computational basis.
3) It is not a requirement that the two values of the bit be represented by the standard basis. Any pair of mutually orthonormal (vectors that are orthogonal with respect to each other i.e. their inner, scalar or dot product is zero, and have a unit magnitude) vectors may represent the two values of the bit, the choice of the standard basis is made out of convenience. Throughout the document, the standard basis will form the orthonormal

basis for representation of the bit, and will be referred to as the computational basis, unless specified otherwise.

4) The bit is represented by a unit directional vector in this two dimensional vector space. The directional orientation of the vector representing the bit constitutes the value held by that bit. Physicists generally refer to such a space (as the one representing the bit) as the "state space" (of the bit), since it represents the current state or value held by the bit. The directional vector in this state space that represents the value held by the bit is called the bit state vector. For sake of consistency, the nomenclature of bit state space and bit state vector will be adopted in the new representation.

Computer Engineering	⇔	**Physics & Mathematics**
Bit or Binary Digit	⇔	**Bit State Space**
Value of the bit (0 or 1)	⇔	**Bit state vector (x or y-axis)**

5) The classical bit (classical bit implies the bit as denoted in traditional computing in order to differentiate it from the quantum bit or "qubit") may exist in any one of two discrete states, "0", or "1". Therefore, the directional vector is allowed to orient itself either along the horizontal x-axis (representing state "0"), or the vertical y-axis (representing state "1") of the bit state space.

6) The bit state vector may not exist as a linear combination (also referred to as superposition or linear superposition in physics, and, therefore, in quantum computing as well) of the computational basis vectors in classical computation. The imposition of this

condition should lead to a violation of the principle of linearity, but it is seen at a later stage that this is not the case.

7) A change in value of the bit ($0 \rightarrow 1$ or $1 \rightarrow 0$) constitutes a rotation of the bit state vector through $\pi/2$ radians (from x to y-axis if changing from 0 to 1, or from y to x-axis if changing from 1 to 0) in the state space. This ensures that the bit state vector always lies on (or is aligned in the same direction as) one of the computational basis vectors.

In computational logic where Boolean algebra is employed to perform logical operations, Boolean functions employ Boolean variables to represent inputs and outputs. A Boolean variable constitutes a bit. Therefore, the representation of the bit in terms of a linear space constitutes the representation of the Boolean variable. The Boolean variable has been successfully represented in terms of a linear space, but a single variable only forms the most atomic unit of a logical operation. A computational operation generally involves operations upon multiple variables. Even elementary Boolean gates require a minimum input comprising two Boolean variables. Multiple bits (or combination of multiple Boolean variables as seen in the classical context) are represented in the proposed description through a mathematical operator known as the tensor direct product.

Definition: The tensor product of two vector spaces $\boldsymbol{V}$ and $\boldsymbol{W}$ denoted $\boldsymbol{V} \otimes \boldsymbol{W}$ and also called the tensor direct product is a way of creating a new vector space analogous to multiplication of integers. For instance,

$$\mathbf{C}^{n} \otimes \mathbf{C}^{m} = \mathbf{C}^{nm}.$$

In particular, $\mathbf{C} \otimes \mathbf{C}^m = \mathbf{C}^m$, where, $\mathbf{C}^m$ is m dimensional complex space and $\otimes$ represents the tensor direct product.

Therefore, tensor products of the two dimensional linear spaces representing their respective bits, provide a convenient representation for the case of multiple bits in logic functions. For instance, consider the case of two Boolean variables, x and y, containing one bit value each. The four possible combinations of these two variables and their corresponding bit state vectors in the linear interpretation are tabulated below.

Table 4.1: Composition of multiple Boolean variables in the proposed representation.

x	y	$x \otimes y$
0	0	$\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1x1\\1x0\\0x1\\0x0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$
0	1	$\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1x0\\1x1\\0x0\\0x1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$
1	0	$\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0x1\\0x0\\1x1\\1x0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
1	1	$\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0x0\\0x1\\1x0\\1x1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

Please note that the four possible combinations of the Boolean variables ***x***, and ***y*** give rise to four column vectors. The new linear space, arising from the tensor direct product of the two states spaces of variables ***x,*** and ***y*** is four dimensional. It gives rise to four computational basis states that correspond to the four possible combinations of ***x***, and ***y***. They are mutually orthonormal. Armed with the new notation, Boolean variables can be represented in a trivial manner through application of the tensor direct product. The dimensionality of the linear space that represents such Boolean variables is always 2^n (where n is the number of variables). This count of dimensionality corresponds to the number of computational basis states in the bit state space of the Boolean variable in the proposed representation, and to the number of possible combinations of the bit values in the standard representation.

4.3 Logic Gates & Logic Functions: Linear Operators

Boolean variables, by themselves do not hold any significance. They have to be operated upon by logic functions or circuits to transform them to their respective output values. These transformations should occur within the framework of the transformation rules that dictate the behavior of the logic function. In the representation being proposed, logic functions constitute linear transformations performed upon the linear space representing the input variables, resulting in a transformed state space that coincides with the output of the logic functions. These linear transformations are represented by linear operators.

Elementary logic gates act upon input variables and provide an output value depending upon the combination of inputs provided to the gate, and the rules governing the operation of the gate. Therefore, in linear algebraic terms these logic gates represent a mapping from one state

space or a tensor product of spaces (representing the input variables) into a target state space (representing the output). Customarily, the output is a single bit in case of an elementary logic gate. Therefore, the corresponding linear algebraic representation would be a two dimensional state space. However, this is not a rigid restriction binding upon the elementary gate, and multiple outputs may also be present (as will be shown in the reversible paradigm where all linear operators are represented by square matrices). It would be impossible to study linear operators corresponding to all possible logic functions. However, since logic functions are composites of elementary logic gates, it suffices to analyze the linear operators corresponding to the elementary logic gates.

4.3.1 The Inverter

The inverter is a single input, single output gate. It acts upon an input variable comprising a single bit, and provides the complement of the input variable as its output. As the name suggests, the gate acts upon an available input variable and inverts its bit value. The inverter gate is symbolically represented as shown in the figure below. The inverter is also referred to as the NOT gate in standard nomenclature.

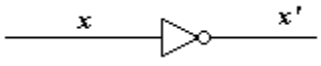


Figure 4.1: Symbolic representation of the Inverter gate.

Table 4.2: Truth Table for the Inverter/NOT gate.

x	$f(x)=x'$
0	1
1	0

As noted earlier, logical operations performed by logic gates or functions upon Boolean variables, constitute linear transformations in the new representation being proposed. Therefore the inverter represents a linear operator that conducts a linear transformation upon the input variable (in our specific case the variable $\boldsymbol{x}$).

Consider a linear operator $\mathbf{L_{NOT}}$ (the letter **L** is chosen for sake of convenience since this operator constitutes a logical operation, and the subscript details the specific logical operation being performed) that represents the inverter. Therefore, the operator $\mathbf{L_{NOT}}$ acting upon the computational basis vectors $\begin{bmatrix}1\\0\end{bmatrix}$, and $\begin{bmatrix}0\\1\end{bmatrix}$, is expected to invert the values they represent. Therefore, $\mathbf{L_{NOT}}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$, and $\mathbf{L_{NOT}}\ \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$. This constitutes a system of four linear simultaneous equations with four unknown coefficients, representing the elements of a 2x2 matrix of the form $\begin{bmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix}$.

Therefore, the system of equations breaks down to:

$a_{11}(1) + a_{12}(0) = 0;$ $\quad\quad$ $a_{21}(1) + a_{22}(0) = 0;$

$a_{11}(0) + a_{12}(1) = 1;$ $\quad\quad$ $a_{21}(0) + a_{22}(1) = 1.$

Upon solving the four equations, the following values the four elements of the matrix take on the values $a_{11} = 0$; $a_{12} = 1$; $a_{11} = 0$; $a_{12} = 1$.

Therefore, the matrix representing the linear operator $\mathbf{L_{NOT}}$ is $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. The matrix is verified against the input bit state vectors $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. The resulting transformed bit state vectors do confirm the expected results as shown below:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ and } \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

4.3.2 The AND Gate

The AND gate takes a minimum of two Boolean variables as its input, with an output result of a single variable. Let us consider the generic case of a two input AND gate. Please note that the results obtained for the two input case may be trivially extended to a higher number of inputs, without any loss of generality. The symbolic representation of the two input AND gate in the circuit model and the corresponding truth table are provided below.

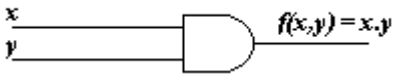


Figure 4.2: Symbolic representation of the two input AND gate.

Table 4.3: Truth table for the two input AND gate.

x	***y***	***f(x,y)= x.y***
0	0	0
0	1	0
1	0	0
1	1	1

The AND gate under study is comprised of two input variables. Therefore, a tensor product of two state spaces (each of dimensionality 2) is taken as the composite input that the linear operator would act upon. The dimensionality of the state space representing the input variables ***x***, and ***y***, is four. The four computational basis states correspond to the four possible value combinations of the two input variables.

Let the matrix representing the AND gate under question be $\mathbf{L}_{\mathbf{AND2}}$ (2 in the subscript implies this is a two input AND gate). Translating the transformation rules of the truth table in Boolean representation to the new representation, we arrive at the following possibilities:

$\mathbf{L_{AND2}}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$ (input $\boldsymbol{x} = 0, \boldsymbol{y} = 0$); $\mathbf{L_{AND2}}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$ (input $\boldsymbol{x} = 0, \boldsymbol{y} = 1$);

$\mathbf{L_{AND2}}\begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$ (input $\boldsymbol{x} = 1, \boldsymbol{y} = 0$); $\mathbf{L_{AND2}}\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$ (input $\boldsymbol{x} = 1, \boldsymbol{y} = 1$)

These four possibilities give rise to a system of eight linear simultaneous equations, corresponding to eight unknown elements of the $\mathbf{L_{AND2}}$ matrix, which is a 2x4 matrix of the form $\begin{bmatrix}a_{11} & a_{12} & a_{13} & a_{14}\\ a_{21} & a_{22} & a_{23} & a_{24}\end{bmatrix}$.

The eight linear simultaneous equations that arise from the four operations are:

$a_{11}(1) + a_{12}(0) + a_{13}(0) + a_{14}(0) = 1$; $a_{11}(0) + a_{12}(1) + a_{13}(0) + a_{14}(0) = 1$;

$a_{21}(1) + a_{22}(0) + a_{23}(0) + a_{24}(0) = 0$; $a_{21}(0) + a_{22}(1) + a_{23}(0) + a_{24}(0) = 0$;

Implies $a_{11} = 1$; $a_{21} = 0$. Implies $a_{12} = 1$; $a_{22} = 0$.

$a_{11}(0) + a_{12}(0) + a_{13}(1) + a_{14}(0) = 1$; $a_{11}(0) + a_{12}(0) + a_{13}(0) + a_{14}(1) = 0$;

$a_{21}(0) + a_{22}(0) + a_{23}(1) + a_{24}(0) = 0$; $a_{21}(0) + a_{22}(0) + a_{23}(0) + a_{24}(1) = 1$.

Implies $a_{13} = 1$; $a_{23} = 0$. Implies $a_{14} = 0$; $a_{24} = 1$.

Therefore the matrix representing $\mathbf{L_{AND2}}$ is $\begin{bmatrix}1 & 1 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$.

Upon verifying with the four possible input combinations, the expected results are observed for the transformed states.

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 1);$$

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 1).$$

Hence it is noted that the operation of the linear operator represented by the matrix $\mathbf{L}_{\mathbf{AND2}}$ in the new representation being developed, is in conformance with the operation of the two input AND gate in the standard Boolean representation.

4.3.3 The OR Gate

The OR gate takes a minimum of two input variables, and provides a standard one variable output. The symbolic representation of the OR gate and the rules governing its operation for the possible input combinations (once again, the two input case will be studied, but the results may be trivially extrapolated for inputs greater than the minimum required) are provided below.

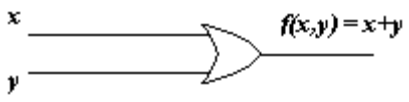


Figure 4.3: Symbolic representation of the two input OR gate.

Table 4.4: Truth table for the two input OR gate.

x	y	$f(x,y) = x+y$
0	0	0
0	1	1
1	0	1
1	1	1

As seen in case of the two input AND structure, the linear operator for the two input OR gate also requires a tensor product of two state spaces, forming the composite input to the linear operator.

Let the matrix representing the OR gate under question be $\mathbf{L}_{\mathbf{OR2}}$. Translating the transformation rules of the truth table in the Boolean representation to the new representation being developed, gives rise to the following possibilities:

$$\mathbf{L}_{\mathbf{OR2}}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 0); \qquad \mathbf{L}_{\mathbf{OR2}}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 1);$$

$$\mathbf{L}_{\mathbf{OR2}}\begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 0); \qquad \mathbf{L}_{\mathbf{OR2}}\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 1).$$

These four possibilities give rise to a system of eight linear simultaneous equations, corresponding to eight unknown elements of the $\mathbf{L_{OR2}}$ matrix. The $\mathbf{L_{OR2}}$ is a 2x4 matrix of the form $\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix}$.

The eight linear simultaneous equations that arise from the four operations shown above are:

$a_{11}(1) + a_{12}(0) + a_{13}(0) + a_{14}(0) = 1$; $a_{11}(0) + a_{12}(1) + a_{13}(0) + a_{14}(0) = 0$;

$a_{21}(1) + a_{22}(0) + a_{23}(0) + a_{24}(0) = 0$; $a_{21}(0) + a_{22}(1) + a_{23}(0) + a_{24}(0) = 1$;

Implies $a_{11} = 1$; $a_{21} = 0$. Implies $a_{12} = 0$; $a_{22} = 1$.

$a_{11}(0) + a_{12}(0) + a_{13}(1) + a_{14}(0) = 0$; $a_{11}(0) + a_{12}(0) + a_{13}(0) + a_{14}(1) = 0$;

$a_{21}(0) + a_{22}(0) + a_{23}(1) + a_{24}(0) = 1$; $a_{21}(0) + a_{22}(0) + a_{23}(0) + a_{24}(1) = 1$.

Implies $a_{13} = 0$; $a_{23} = 1$. Implies $a_{14} = 0$; $a_{24} = 1$.

Therefore the matrix representing $\mathbf{L_{OR2}}$ is $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$.

Upon verifying the four possible inputs, the expected results are observed for the transformed states.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 1)$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 1).$$

From the results above it is evident that the matrix $\mathbf{L}_{OR2}$ conforms to the transformation rules in the truth table for the two input OR gate. Therefore the linear operator corresponds to the OR gate.

4.3.4 The NAND Gate

The NAND gate forms a composite of the AND gate followed by the inverter, hence it may be conceived as a gate that inverts the output of an AND gate. Although it is formed by the composite of two elementary logic gates, it falls amongst the set of elementary logic gates. The reason for this classification lies in the property of the NAND structure being a universal gate (any elementary gate may be constructed through a combination of NAND gates). The NAND gate takes a minimum of two inputs. The symbolic representation of the NAND gate and its rules of operation are provided below.

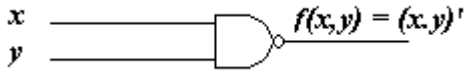


Figure 4.4: Symbolic representation of the two input NAND Gate.

Table 4.5: Truth table for the two input NAND gate.

x	***y***	***f(x,y)=(x.y)'***
0	0	1
0	1	1
1	0	1
1	1	0

There are three approaches available to represent the linear operator corresponding to the NAND gate. Nevertheless, all representations are equivalent as they lead to the same linear operator. The alternative representation schemes arise from De-Morgan's laws, and their validity in the new representation being developed further buttresses the consistency of this interpretation.

The three equivalent representations of the NAND gate are:

1) To directly construct the gate structure from the transformation rules.
2) Representing the NAND gate as a composite of the AND, followed by the NOT structure.
3) Applying the equivalence relationship provided by De-Morgan's laws, and representing the NAND gate as a composite of two NOT gates applied on the individual inputs, followed by the OR gate (since $f(x,y) = (x.y)' = x' + y'$ by De-Morgan's Theorem).

4.3.4.1 Representation 1: Developing NAND Gate from Transformation Rules

Let the matrix representing the two input NAND gate be $\mathbf{L_{NAND2}}$. Translating the transformation rules of the truth table in the Boolean representation to the new representation, gives rise to the following possibilities:

$$\mathbf{L_{NAND2}}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 0); \qquad \mathbf{L_{NAND2}}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 1);$$

$$\mathbf{L_{NAND2}}\begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 0); \qquad \mathbf{L_{NAND2}}\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 1).$$

These four possibilities give rise to a system of eight linear simultaneous equations, corresponding to eight unknown elements of the $\mathbf{L_{NAND2}}$ matrix. $\mathbf{L_{NAND2}}$ is a 2x4 matrix of the form $\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix}$.

The eight linear simultaneous equations that arise from the four operations shown above are:

$a_{11}(1) + a_{12}(0) + a_{13}(0) + a_{14}(0) = 0;$ $\qquad$ $a_{11}(0) + a_{12}(1) + a_{13}(0) + a_{14}(0) = 0;$

$a_{21}(1) + a_{22}(0) + a_{23}(0) + a_{24}(0) = 1;$ $\qquad$ $a_{21}(0) + a_{22}(1) + a_{23}(0) + a_{24}(0) = 1;$

Implies $a_{11} = 0$; $a_{21} = 1$. $\qquad$ Implies $a_{12} = 0$; $a_{22} = 1$.

$a_{11}(0) + a_{12}(0) + a_{13}(1) + a_{14}(0) = 0;$ $\qquad$ $a_{11}(0) + a_{12}(0) + a_{13}(0) + a_{14}(1) = 1;$

$a_{21}(0) + a_{22}(0) + a_{23}(1) + a_{24}(0) = 1;$ $\qquad$ $a_{21}(0) + a_{22}(0) + a_{23}(0) + a_{24}(1) = 0.$

Implies $a_{13} = 0$; $a_{23} = 1$. $\qquad$ Implies $a_{14} = 1$; $a_{24} = 0$.

Therefore the matrix representing $\mathbf{L_{NAND2}}$ is $\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$.

Upon verifying the four possible inputs, the expected results are observed for the transformed states.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 1);$$

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 1).$$

4.3.4.2 Representation 2: NAND gate, Composite of the AND and Inverter

Here the NAND gate may be represented as follows:

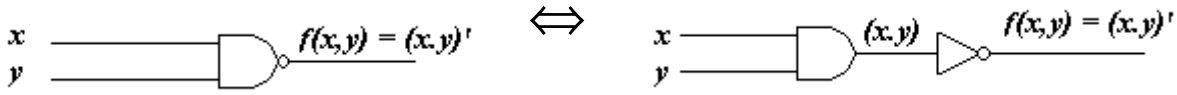


Figure 4.5: The two equivalent representations of the NAND gate.

The equivalent structure in the new representation is $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$.

Please note that the composition of gates in the Boolean representation corresponds to the product of the corresponding matrices in the representation being developed. The composition of gates will be discussed at length later in this Chapter.

Table 4.6: Truth table for the alternate representation.

x	***y***	***x'***	***y'***	***(x.y)***	***(x.y)'***
0	0	1	1	0	1
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	0

4.3.4.3 <u>Representation 3: NAND Gate from De-Morgan's Theorem.</u>

De-Morgan's theorem shows that every logic function in a product-of-sums form has an equivalent sum-of-products expression, and vice versa. Therefore, the NAND gate represented by ***f(x,y) = (x.y)'*** may be alternatively represented as ***f(x,y) = (x'+y')***. The truth table below demonstrates the equivalence of both functional forms by showing that the transformation rules for both forms are the same.

Table 4.7: The truth table proves the equivalence of the two expressions.

x	*y*	*x'*	*y'*	*(x.y)*	*(x.y)'*	*(x' + y')*
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

This alternative representation in symbolic notation is shown below.

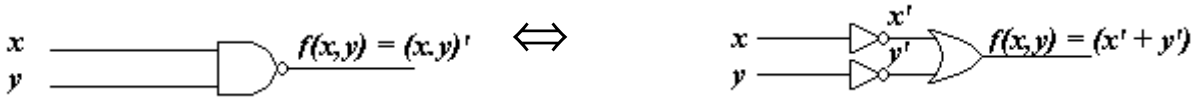


Figure 4.6: Alternative representation through application of De-Morgan's Theorem.

The development of the matrix that represents this function is slightly non-standard, since the NOT gates act upon the individual input variables, where as the OR gate acts upon the composite of the two inverted input variables. Therefore, the composite matrix representation comprises the product of the $\mathbf{L}_{\mathbf{OR2}}$ matrix, and the tensor product of two $\mathbf{L}_{\mathbf{NOT}}$ matrices. This representation is shown below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \left\{ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0x0 & 0x1 & 1x0 & 1x1 \\ 0x1 & 0x0 & 1x1 & 1x0 \\ 1x0 & 1x1 & 0x0 & 0x1 \\ 1x1 & 1x0 & 0x1 & 0x0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

The tensor product of the two $\mathbf{L_{NOT}}$ matrices performs the task of acting upon the input state spaces representing the individual input variables, and not the composite state space of the two variables.

The three alternative representations lead to the same linear operator corresponding to the two input NAND gate. Moreover, the linear operator conforms to the transformation rules detailed in the truth table for the two input NAND gate.

4.3.5 The NOR Gate

The NOR gate is quite similar to the NAND gate in certain aspects, viz. it is a universal gate like the NAND. Therefore, the NOR structure may be applied to represent any elementary logic gate and, hence, any logic function. It has three equivalent representations like the NAND gate, arising once again from the equivalence relationships laid down by De-Morgan's theorem. The symbolic representation of the two input NOR gate and its corresponding truth table are provided below.

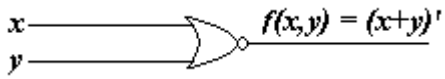


Figure 4.7: Symbolic representation of the two input NOR gate.

Table 4.8: Truth table for the two input NOR gate.

x	***y***	***f(x,y)=(x+y)'***
0	0	1
0	1	0
1	0	0
1	1	0

The three equivalent representations of the NOR gate are detailed below:

1) To directly construct the gate structure from the transformation rules.
2) Representing the NOR gate as a composite of the OR Gate followed by the NOT gate.
3) To apply the equivalence relationship provided by De-Morgan's laws, and representing the NOR gate as a composite of two NOT gates applied on the individual inputs, followed by the AND gate (since ***f(x,y) = (x + y)' = x'.y'*** by De-Morgan's Theorem).

4.3.5.1 <u>Representation 1: Building NOR from Transformation Rules</u>

Let the matrix representing the two input NOR gate be $\mathbf{L_{NOR2}}$. Translating the transformation rules of the truth table in Boolean representation to the representation being proposed, we arrive at the following possibilities:

$$\mathbf{L_{NOR2}}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 0); \qquad \mathbf{L_{NOR2}}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 1);$$

$\mathbf{L_{NOR2}}\begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$ (input $\boldsymbol{x} = 1, \boldsymbol{y} = 0$); $\mathbf{L_{NOR2}}\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$ (input $\boldsymbol{x} = 1, \boldsymbol{y} = 1$).

These four possibilities give rise to a system of eight linear simultaneous equations, corresponding to eight unknown elements of the $\mathbf{L_{NOR2}}$ matrix. $\mathbf{L_{NOR2}}$ is a 2x4 matrix of the form $\begin{bmatrix}a_{11} & a_{12} & a_{13} & a_{14}\\ a_{21} & a_{22} & a_{23} & a_{24}\end{bmatrix}$.

The eight linear simultaneous equations that arise from the four operations shown above are:

$a_{11}(1) + a_{12}(0) + a_{13}(0) + a_{14}(0) = 0$; $a_{11}(0) + a_{12}(1) + a_{13}(0) + a_{14}(0) = 1$;

$a_{21}(1) + a_{22}(0) + a_{23}(0) + a_{24}(0) = 1$; $a_{21}(0) + a_{22}(1) + a_{23}(0) + a_{24}(0) = 0$;

Implies $a_{11} = 0$; $a_{21} = 1$. Implies $a_{12} = 1$; $a_{22} = 0$.

$a_{11}(0) + a_{12}(0) + a_{13}(1) + a_{14}(0) = 1$; $a_{11}(0) + a_{12}(0) + a_{13}(0) + a_{14}(1) = 1$;

$a_{21}(0) + a_{22}(0) + a_{23}(1) + a_{24}(0) = 0$; $a_{21}(0) + a_{22}(0) + a_{23}(0) + a_{24}(1) = 0$.

Implies $a_{13} = 1$; $a_{23} = 0$. Implies $a_{14} = 1$; $a_{24} = 0$.

Therefore, the matrix representing $\mathbf{L_{NOR2}}$ is $\begin{bmatrix}0 & 1 & 1 & 1\\ 1 & 0 & 0 & 0\end{bmatrix}$.

Upon verifying the four possible inputs, the expected results are observed for the transformed states.

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 1);$$

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 1).$$

4.3.5.2 Representation 2: NOR as Composite of OR and Inverter

Here the NOR gate may be represented as follows.

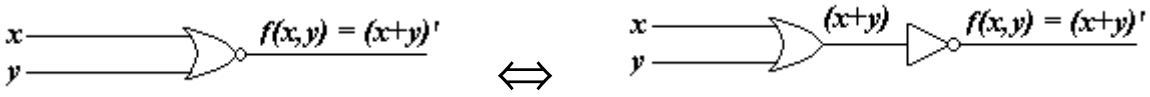


Figure 4.8: The two equivalent alternative representations of the NOR structure.

The equivalent linear operator is obtained through matrix product of the individual matrices representing the inverter and two input OR gate: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$

Table 4.9: Truth table for the alternate representation.

x	*y*	*x'*	*y'*	*(x+y)*	*(x+y)'*
0	0	1	1	0	1
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	1	0

4.3.5.3 Representation 3: NOR Gate through De-Morgan's Theorem.

According to De-Morgan's theorem, for every logic function in a product-of-sums form, there exists an equivalent sum-of-products expression, and vice versa. Therefore, the NOR gate represented by ***f(x,y) = (x+y)'*** may be alternatively represented as ***f(x,y) = (x'.y')***. Both the forms are equivalent. This can easily be demonstrated by verifying the results for both forms in the truth table.

Table 4.10: The truth table shows the equivalence of the two expressions.

x	*y*	*x'*	*y'*	*(x'.y')*	*(x+y)*	*(x' + y')*
0	0	1	1	1	0	1
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	1	0	0	0	1	0

This alternative representation in symbolic notation is shown below.

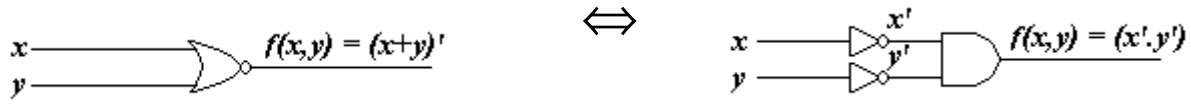


Figure 4.9: Alternative notation through application of De-Morgan's Theorem.

The matrix corresponding to the linear operator that represents this function is to be taken in a fashion, similar to the second representation of the NAND structure, owing to the action of the inverters on single input bits before they feed the AND structure. Therefore the composite matrix representation comprises of the product of the matrix corresponding to the AND gate and the tensor product of two matrices that represent the NOT gate. This representation is shown below.

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \left\{ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0x0 & 0x1 & 1x0 & 1x1 \\ 0x1 & 0x0 & 1x1 & 1x0 \\ 1x0 & 1x1 & 0x0 & 0x1 \\ 1x1 & 1x0 & 0x1 & 0x0 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Therefore, it is noted that all representations lead to the same matrix structure.

The possible operations, the matrix could perform on the four possible input states are provided below.

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 0, \boldsymbol{y} = 1);$$

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 0); \qquad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (input } \boldsymbol{x} = 1, \boldsymbol{y} = 1).$$

4.4 Properties of Logic Gates in Boolean and the New Representation

1) The matrix $\mathbf{L_{NOT}}$ corresponding to the NOT gate is invertible (i.e. an inverse to the matrix exists). In fact the inverse $\mathbf{L_{NOT}}^{-1}$ is the same as the matrix $\mathbf{L_{NOT}}$ itself. Therefore, applying the inverse matrix $\mathbf{L_{NOT}}^{-1}$ to the output state vector results in the original input. This operation is shown below:

   $$\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ and } \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

2) Property 1 does not arise by mere co-incidence. In the Boolean representation, the Inverter, when reversed acts just the same, i.e. if the output of the Inverter were plugged in at the input end of the gate, it would provide the original input variable.

3) It is also known from elementary linear algebra that the product of a matrix with its inverse results in the Identity. In case of $\mathbf{L_{NOT}}$, since $\mathbf{L_{NOT}} = \mathbf{L_{NOT}}^{-1}$, $(\mathbf{L_{NOT}})^2 = \mathbf{I}$, the Identity. This implies, if the Inverter is applied twice, successively on an input variable, the output corresponds to the value of the original input variable itself. This property of

the Inverter is therefore seen to be consistent in the Boolean representation as well as the proposed representation under development.

4) The dimensionality of a matrix corresponding to a Boolean gate/function in the new representation is a function of the number of inputs and outputs to the gate/function. The number of rows in the matrix is related to the number of variables constituting the function's output. The number of rows of the matrix is always 2^n (where n is the number of output variables). As there generally exists a single variable at the output end of all elementary Boolean gates, the number of rows is 2 in their case. On the other hand, the number of columns in the matrix is dependent upon the number of inputs provided to the gate. The relationship shared between the number of columns and the number of input variables is given by the formula: number of columns = 2^n (where n is the total no. of input variables). Since the elementary logic gates take two input variables, the number of columns is $2^2 = 4$.

5) All matrices representing elementary logic gates, with the exception of the inverter, are rectangular, given the fact that they take a minimum of two input variables (leading to a four dimensional input state space) and result in one functional output variable (leading to a two dimensional output state space). Therefore, the dimensionality of the state spaces representing the inputs and outputs to the corresponding linear operators is not the same. For a matrix to be invertible, the number of rows and columns must be equal (implying that the input and output state spaces share the same dimensionality). This forms a necessary condition to be satisfied in order to achieve invertible mappings. This property bears great significance when studying the paradigm of reversible computing.

6) From the computational standpoint, property 5 implies that these gates are not reversible by nature. In case of the AND gate for instance, given that values of the function output are known, values of the input variables are trivially known to be $\boldsymbol{x} = 1$, and $\boldsymbol{y} = 1$ when output $\boldsymbol{f(x,y)} = 1$. Alternatively, if the output value $\boldsymbol{f(x,y)} = 0$, it cannot be conclusively ascertained whether the inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ hold values $\boldsymbol{x} = 0, \boldsymbol{y} = 0$, or $\boldsymbol{x} = 0, \boldsymbol{y} = 1$, or $\boldsymbol{x} = 1, \boldsymbol{y} = 0$. This information about the individual values of inputs $\boldsymbol{x}$, and $\boldsymbol{y}$ is not available. Hence the gate is not reversible (or the linear operator is not invertible). The same argument is applicable to the OR, NAND and NOR gate structures.

7) The Boolean representation allows three equivalent representations of the NAND/NOR structures. These representations are shown to be equivalent in the new representation as well.

4.5 Combinational Logic Functions

Any combinational logic function may be realized through a composition of elementary logic gates. The placement of the elementary logic gates in the circuit is determined by the transformation rules of the logic function in question. Each elementary logic gate contributes in some manner to realize the final set of transformation rules that characterize the logic function. Therefore, the elementary logic gates are all that are required in order to represent any realizable logic function. Combinational logic, unlike its sequential counterpart, has no time dependency or dependency upon previous outputs of the function for its successful operation. These functions take inputs, process them and provide the output results with no time controlled or memory controlled parameters. Construction of linear operators corresponding to the

combinational logic functions shares a correspondence with the circuit model representation of logic functions.

In the circuit model representation of combinational logic, the inputs are generally fed in from the left, processed by the individual elementary logic gates at different stages and arrive at the output stage at the right hand side. Therefore, logic gates to the left act upon the inputs and transform them before they arrive at the succeeding stage of logic gates further to the right in the circuit structure.

Every combinational function in the Boolean or circuit model representation has an equivalent linear operator in the representation being developed. The composition of logic gates to represent a combinational logic function in the Boolean representation shares a one-to-one correspondence with the construction of an equivalent linear operator in the representation being proposed. However, some rules of composition are imposed in order to maintain the correspondence between the circuit model and the alternate representation. The composition rules that define the correspondence are enumerated as follows:

1) The linear operator that represents the combinational logic function is constructed by multiplication of the individual matrices corresponding to the elementary logic gates that make up the composite function.
2) Elementary logic gates falling to the left in the circuit model representation of the combinational function have their corresponding matrices falling to the right of the multiplication operator in the proposed representation. This occurs since the input states

acted upon fall to the right of the linear operator. Therefore, the first set of matrices to act upon the input state must fall immediately to the left of the input. These are followed by the matrices that represent the logic blocks that form the successive stage of logical operation.

3) In certain instances a logic gate acts only on a partial set of inputs and not all. In such cases, the tensor product is applied so that the linear operator acts upon the specific inputs with the identity being applied upon the remaining inputs so that they do not undergo transformation. In this manner the complete state space representing the entire set of inputs does not undergo transformation. Instead, the subspace representative of the particular set of variables being acted upon must be exclusively taken into account for transformation.

The rules of composition stated above will be apparent from a simple example that follows. We will construct a logic function with two equivalent circuit representations arising as a result of De-Morgan's theorem. The transformation rules of the logic function are provided by its truth table and the two equivalent representations of the logic function are derived from this table. The circuits and their corresponding linear operators are then constructed accordingly.

As is apparent from the truth table below, there are two equivalent representations of the logic function labeled ***f1*** and ***f2***.

Table 4.11: Truth Table for the example combinational function.

A	**B**	**C**	**C'**	**(A.B)**	**(A.B)'**	***f1*=[(A.B)'+C]**	**[(A.B).C']**	***F2*=[(A.B).C']'**
0	0	0	1	0	1	1	0	1
0	0	1	0	0	1	1	0	1
0	1	0	1	0	1	1	0	1
0	1	1	0	0	1	1	0	1
1	0	0	1	0	1	1	0	1
1	0	1	0	0	1	1	0	1
1	1	0	1	1	0	0	1	0
1	1	1	0	1	0	1	0	1

Representation 1: ***f1*(A, B, C) = [(A.B)' + C]**

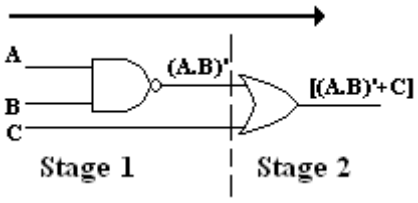


Figure 4.10: Circuit model representation of the combinational function ***f1*=[(A.B)'+C]**.

The circuit presented above, consists of a two input NAND gate acting upon inputs **A** and **B**, with the input **C** remaining unchanged. This marks the first stage of the composite function. The linear operator for the first stage is constructed by forming a tensor direct product of the matrix corresponding to the two input NAND gate and the Identity matrix (since the input variable **C** is not being acted upon in stage 1 of the logical operation). In stage two, the output of the NAND gate and the input **C** are fed to a two input OR gate. Therefore, the linear operator resulting from the tensor product of NAND gate and Identity is acted upon by the matrix representing the two input OR gate. The construction of the linear operator corresponding to the logic function f1 is provided below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \left\{ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right\} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0x1 & 0x0 & 0x1 & 0x0 & 0x1 & 0x0 & 1x1 & 1x0 \\ 0x0 & 0x1 & 0x0 & 0x1 & 0x0 & 0x1 & 1x0 & 1x1 \\ 1x1 & 1x0 & 1x1 & 1x0 & 1x1 & 1x0 & 0x1 & 0x0 \\ 1x0 & 1x1 & 1x0 & 1x1 & 1x0 & 1x1 & 0x0 & 0x1 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}.$$

Representation 2: ***f2*(A, B, C) = [(A.B).C']'**

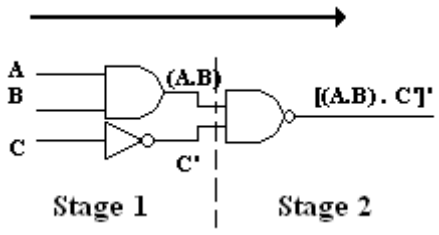


Figure 4.11: Circuit model representation of the logic function ***f2***.

This circuit involves the action of a two input AND gate on the inputs **A** and **B**, with input **C** being complemented through action of an inverter in stage 1. The linear operator corresponding to the stage 1 of the logic function is a tensor product of the matrix corresponding the two input AND gate and that of the inverter. The outputs of this stage, **(A.B)**, and **C'** form the inputs for the stage 2 comprising a two input NAND gate. Therefore the matrix corresponding to the two input NAND acts upon the linear operator resulting from the previous stage. The formation of the linear operator through the successive stages is shown below.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \left\{ \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1x0 & 1x1 & 1x0 & 1x1 & 1x0 & 1x1 & 0x0 & 0x1 \\ 1x1 & 1x0 & 1x1 & 1x0 & 1x1 & 1x0 & 0x1 & 0x0 \\ 0x0 & 0x1 & 0x0 & 0x1 & 0x0 & 0x1 & 1x0 & 1x1 \\ 0x1 & 0x0 & 0x1 & 0x0 & 0x1 & 0x0 & 1x1 & 1x0 \end{bmatrix} =$$

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}.$$

Please note that the matrices that result finally are one and the same for both representations, which proves the equivalence of the logic functions they represent.

4.6 Sequential Networks

The extension of the alternate representation being proposed from combinational to sequential networks is quite trivial. Functions that possess a regular structure that is time-

iterative can be represented compactly in terms of sequential networks. Consider for example, an arbitrary combinational function that represents function composition of the following form:

$$q_{i+1} = F(x_i, q_i)$$

$$y_{i+1} = F(x_i, q_i)$$

where x_i is an input to the function,

q_i is an output of function F forming an input for the successive stage,

y_i forms an output of the combinational function at a given stage.

i forms the time step of the iteration procedure ($-\infty < i < +\infty$).

This constitutes a combinational circuit represented by an infinite causal network. The function scheme may be represented in a diagrammatic representation as shown below.

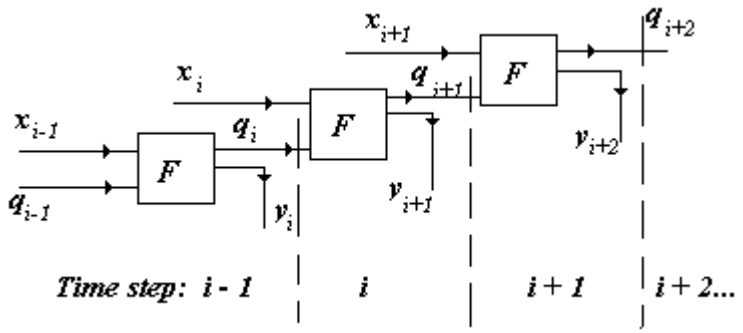


Figure 4.12: Symbolic representation of the time-iterative combinational function shown above.

Such a function scheme has a self repeating pattern of infinite length, and the practical realization of such a circuit is quite impossible. However, the same combinational function scheme may be represented by means of a finite sequential circuit as shown below.

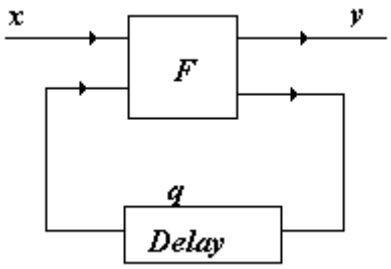


Figure 4.13: A sequential circuit representation of the self-repeating combinational function.

By comparing the two structures it becomes apparent that the role of the "delay" element in the sequential network of Figure 4.13 is to mark out those places that correspond to the boundary between one stage and the next in the equivalent time-iterative combinational function represented by Figure 4.12. The transformation rules continue to be the same as in the combinational case, except that they assume a time dependence introduced by the time-delay memory element in the feedback loop of the sequential network. But this assumes no major significance in the operation of the combinational part of the network. The combinational function merely takes in the inputs provided by the input line represented by $\boldsymbol{x}$ and the output of the previous stage represented by $\boldsymbol{q}$. Hence, sequential networks can also be represented by linear operators, and their matrices correspond to the combinational function of the network, except for the fact that the input bit state vector that the matrix operates has a dependency upon the previous output provided by the matrix.

In the linear representation, a sequential circuit may be represented as follows.

Let $\mathbf{L_F}$ constitute the linear operator that corresponds to the arbitrary function discussed in the example above, and let $\boldsymbol{x_i}$, $\boldsymbol{q_i}$ and $\boldsymbol{y_i}$ represent the bit state vectors for the input, output of the

previous stage feeding input of the subsequent stage, and the output of the stage $\boldsymbol{i}$ respectively. Then the discrete time evolution of the sequential circuit may be mathematically represented as $\mathbf{L_F}\,(\boldsymbol{x_i} \otimes \boldsymbol{q_i}) = \boldsymbol{y_i} \otimes \boldsymbol{q_{i+1}}$.

4.7 An alternative Derivation of the Linear Operator

So far, we have seen the standard mathematical method of solving linear simultaneous equations to derive the linear operator corresponding to a given logic function. This method becomes quite cumbersome for large functions since the number of equations to solve increases exponentially with the number of inputs and outputs. Even in cases where the linear operators may be constructed through the composition rules discussed in the previous section, it is observed that the derivation of the final linear operator can be cumbersome, especially when tensor products are involved. An easier alternative technique for derivation of the linear operator is developed in this section.

As noted earlier, the linear operator represents a linear mapping from one linear space into another. The spaces being related by the mapping need not necessarily share equal dimensionality. The logic gates in the Boolean representation and the corresponding linear operators in the alternate representation share a common set of transformation rules, and it is from these transformation rules that the matrix representing the linear operator is derived.

A matrix constitutes a rectangular array of elements, where the individual columns represent the possible inputs that may be provided to the logic function in question and the rows correspond to the possible outputs. The intersection of the columns and rows forms a rectangular

grid of elements. These elements constitute the transformation rules that characterize the logic function and its corresponding linear operator. Traversing down a column from one row to the next, the value '1' is placed if the input corresponding to the column leads to the output corresponding to the particular row, if not the value '0' is plugged in. Consider the example discussed in Section 4.5. The possible inputs of this logic function are {000, 001, 010, 011, 100, 101, 110, 111} (since the function inputs comprise three input variables), and the possible outputs are {0, 1} (since the function's output is characterized by a single variable).

$$\begin{array}{c} \quad 000 \quad 001 \quad 010 \quad 011 \quad 100 \quad 101 \quad 110 \quad 111 \\ \begin{array}{c} 0 \\ 1 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

What do the transformation rules say?

Input →	Output
000	1 [Place '0' position Row1, Column1 and '1' in Row2, Column 1]
001	1 [Place '0' position Row1, Column2 and '1' in Row2, Column 2]
010	1 [Place '0' position Row1, Column3 and '1' in Row2, Column 3]
011	1 [Place '0' position Row1, Column4 and '1' in Row2, Column 4]
100	1 [Place '0' position Row1, Column5 and '1' in Row2, Column 5]
101	1 [Place '0' position Row1, Column6 and '1' in Row2, Column 6]
110	0 [Place '1' position Row1, Column7 and '0' in Row2, Column 7]
111	1 [Place '0' position Row1, Column8 and '1' in Row2, Column 8]

The rationale behind the matrix representation is that of transformation rules representing a set of linear simultaneous equations. However, since the formation of truth tables comes naturally to people in computing, the construction of the matrix becomes quite trivial from the truth table. If observed, it is seen that the matrix represents the individual functional outputs of the truth table if traversed from left to right, only they are represented as column vectors and not in the usual binary notation (provided the input columns are logged in the standard procedure as done in truth tables). This technique of deriving the linear operator directly from the transformation rules laid forth in the truth table of the function is not a mere coincidence. There currently exists a very successful application to VLSI CAD tool design that exploits the linearity of the transformation rules. This application will be discussed at length in Chapter 6.

4.8 Observations

The following observations may be made upon re-interpreting Boolean logic functions in the new representation we have developed.

1) The "bit" may be represented in a two-dimensional complex vector space, where it is aligned along any of the two computational basis vectors of the state space, but never a linear combination. Classical logic functions do not permit the existence of the bit as a linear combination of the basis states, or a "superposition" state as it is sometimes referred to.
2) Multiple bits may be conveniently treated by borrowing the concept of tensor products from the theory of tensor mechanics. The scenario of a gate or logic function taking

multiple bits as its/their input becomes possible as a tensor direct product of the individual state spaces that represent the respective bits.

3) The elementary logic gates that act upon input bits within the framework of a set of transformation rules (collectively referred to as the truth table) constitute, what are known as linear operators in the linear algebraic domain. Linear operators are represented by matrices that act upon bit state spaces or a tensor product of bit state spaces, resulting in an output bit state space or the target space.
4) The NAND and NOR gates which share alternative but equivalent representations arising from the application of De-Morgan's theorem are seen to be equivalent in both representations.
5) Combinational logic functions are formed by a compound interaction of multiple elementary logic gates. The linear operator corresponding to a combinational logic function is constructed through products and/or tensor products (depending upon the action of the respective gates upon their inputs) of matrices corresponding to the elementary logic gates. In this manner it is seen that the construction of combinational logic functions and their corresponding linear operators share a one-to-one correspondence.
6) From the last two points above, it is established that the linear operators represented by matrix structures lead to the same results that the Boolean gates and functions do. More importantly the matrix structures strictly operate within the rules that dictate their operation in the theory of linear spaces. Hence they necessarily represent isomorphic representations of the same logical structure.

7) The linear operators corresponding to sequential networks are no different from the combinational part of such networks, except for the fact that the inputs they act upon are dependent upon the previous outputs provided by the linear operators.
8) Linear operators are canonical representations of the respective logic functions. This implies that the linear operator corresponding to a given logic function is unique. This is true in the context of classical computing only. As will be seen, the same is not true in the paradigms of reversible and quantum computing.
9) Transformation of bit states through application of linear operators constitutes rotation of the bit state vector through multiples of $\pi/2$ radians. The transformation of the state space itself might involve a change in dimensionality, depending upon the number of output bit states resulting from the transformation.

4.9 Conclusion

This chapter marked the development of a new theoretical description based on the theory of linear spaces. Owing to the fact that logic elements exhibit a high degree of linearity in this description, this representation shall henceforth be referred to as the *linear representation*. The classical bit was successfully represented as a two dimensional complex Euclidean space and the same was extended to the case of multiple bits through application of the tensor direct product of the individual bit state spaces. Elementary Boolean gates and combinational logic functions were shown to correspond to linear transformations of the bit state spaces in the new representation. The linear operators affecting the linear transformations were represented by matrices. The representation was further extended to the case of sequential networks and logical arguments were provided in favor of the validity and logical consistency of such representation.

Some properties of the matrices in the *linear representation* were discussed and it was observed that these attributes coincide with those of the Boolean gates and functions in the Boolean or circuit model representation of logic. This shows that constraint 2 specified in Section 2.2 detailing that "the proposed description should make a seamless transition to the higher level of theoretical abstraction immediately above in the abstraction hierarchy" has been satisfactorily met. The *linear representation* has been developed without compromising or violating any constructs of the mathematical formalism (the theory of linear spaces in this case), hence constraint 3 is also met satisfactorily.

5.0 THE LINEAR REPRESENTATION: REVERSIBLE AND QUANTUM COMPUTATION.

5.1 Introduction

The previous chapter introduced the *linear representation* in the context of classical computation. The representation was shown to include all attributes of Boolean algebra and the circuit model by developing the necessary constructs for elementary Boolean gates, and the same was extended to provide a coherent explanation of combinational logic functions and sequential networks.

The paradigm of reversible computing was primarily developed as an extension to classical computing. It assumed a significant role when this paradigm was understood to form an important link that relates classical computing to quantum computing. Quantum computation holds promise of providing a solution to problems that are deemed difficult or even intractable in the classical paradigm, apart from being able to solve what is already considered computable in classical computing. This possibility has led to suggestions that classical computation might form a subset of the quantum paradigm. This chapter extends the *linear representation* to the domains of reversible and quantum computing and resolves the explicit relationship shared by the three paradigms.

5.2 Linear Representation of Reversible Computation: Satisfiability Criteria

The theory of reversible computing is based on invertible primitives and composition rules that preserve invertibility. The most important attribute common to all logic functions in this paradigm is that they are reversible. It involves the capability to reverse a computational process to exactly reconstruct the previous state of the computational process from its current state, that is feeding the output of a circuit representing a function back into itself provides the original inputs. The *linear representation* developed in Chapter 4 shows that any logic function may be represented as a linear operator. It was noted that the matrices corresponding to logic functions were, in general, not invertible. This observation coincides with the fact that Boolean functions are not reversible by nature. The attribute of reversibility of the circuits corresponds to invertibility of the linear operators in the *linear representation*.

In the theory of linear spaces, a linear operator or linear mapping is considered invertible if and only if the mapping is one-to-one and onto. These necessary and sufficient conditions imply:

1) Every vector in the domain space maps to a unique vector in the range space, implying that the mapping is one-to-one.
2) Every vector in the range space has a unique pre-image in the domain space, which implies the mapping is onto, that is, all elements or vectors in the range space are mapped and they have unique counterparts in the domain space.

These two conditions define primitives for invertibility of the linear operator. They translate to the following satisfiability criteria for the matrices representing the linear mappings that characterize logic functions. For sake of consistency, these criteria will be referred to as the invertibility criteria for linear operators.

1) The linear spaces involved in the mapping must have the same dimensionality, i.e. the dimensionality of the bit state space representing the inputs must equal the dimensionality of the bit state space representing the outputs. This leads to a linear operator that is represented by a square matrix.
2) The matrix representing the linear mapping must be a full rank matrix. This implies that the row-reduced echelon form of the square matrix must form the Identity. This ensures that every vector in the range space has a unique pre-image in the domain vector space.
3) In computational terms the criterion for equal dimensionality of the input and output bit state spaces requires that the number of inputs of the logic function be equal to the number of outputs.
4) The onto condition translates in computational terms to the requirement that every output of the logic function always lead to a unique input value.
5) The final criterion to be satisfied at all times is that the modification of logic functions to accommodate invertible primitives should not alter its functional properties. This implies the logic function must preserve the set of transformation rules that characterize it, for a modification of the transformation rules would modify the function itself.

5.3 Modified Elementary Boolean Gates to Accommodate Invertible Primitives

The invertibility criteria will be met in a logical progression. The number of inputs and outputs to the gates are equalized at the first stage in order to meet the requirement for equal dimensionality of the bit state spaces. The case of the inverter will not be considered since it already satisfies the necessary and sufficient conditions for invertibility.

5.3.1 The Modified Non-Invertible AND Gate

The AND gate is not reversible by nature, and the matrix that represents the AND gate in the linear representation is not invertible (in fact, it is not even a square matrix). A modified version of the AND gate is shown below.

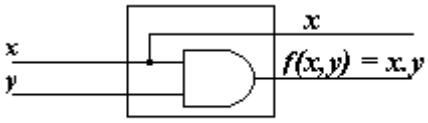


Figure 5.1: The modified two-input, two-output AND gate.

The linear operator corresponding to this gate can be derived using the techniques discussed in Section 4.7. The linear operator corresponding to the modified AND gate is represented by the following matrix.

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As is obvious, this matrix is square since the number of outputs and inputs are equal. The selection of the input $\boldsymbol{x}$ as the secondary output is trivial, one is free to choose the input $\boldsymbol{y}$ in place of $\boldsymbol{x}$, the matrix will change in order to accommodate for the change in the output tensor states, but the transformation rules themselves are preserved. This is verified below, where the matrix operates upon the possible input combinations resulting in the expected outputs.

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ (output } \boldsymbol{x} = 0, \boldsymbol{f(x,y)} = 0).$$

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ (output } \boldsymbol{x} = 0, \boldsymbol{f(x,y)} = 0).$$

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \text{ (output } \boldsymbol{x} = 1, \boldsymbol{f(x,y)} = 0).$$

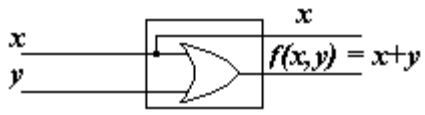
$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \text{ (output } \boldsymbol{x} = 1, \boldsymbol{f(x,y)} = 1).$$

It is apparent that the matrix is square but it is still not invertible. The reason being that the matrix is not a full-rank matrix, thereby not satisfying the one-to-one and onto mapping conditions. Consider the output $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, it can result from the action of the linear operator on either

$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ or $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$. This leads to a many-to-one mapping as multiple inputs lead to the same output. In terms of the Boolean representation it is seen that one still does not have enough information about the value of the two input bits at the output end. Given the fact that the input value of ***x*** and the output ***f(x,y) = x.y*** are available, the value of the input bit ***y*** cannot be conclusively ascertained. In the event of the function output being ***f(x,y)*** = 1, it is definitively known that the input values were ***x*** = 1, ***y*** =1, similarly when the output ***f(x,y)*** = 0, and input ***x*** = 1, value of input ***y*** = 0 is definitely known. But in the event of the output being ***f(x,y)*** = 0, and input ***x*** = 0, it is not known whether input ***y*** = 1 or 0. Therefore, by both accounts, Boolean and linear, it is seen that there does not exist enough information at the output end in order to make the AND gate (alternatively, its matrix) reversible (invertible).

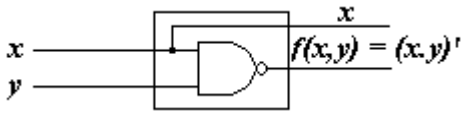
5.3.2 Modified Non-Invertible OR, NAND, and NOR Gates

The same problem as noticed in case of the modified AND gate above, is observed in case of the modified OR, NAND, and NOR gate structures as well. The matrices corresponding to the modified gates do not satisfy the one-to-one and onto mapping conditions, and, therefore, are not full-rank matrices. Their respective Boolean and linear representations are provided below.



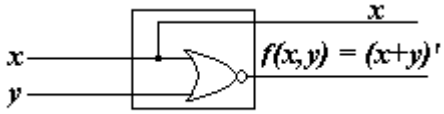
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Figure 5.2: Boolean and linear representations for the modified OR structure.



$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 5.3: Boolean and linear representations for the modified NAND structure.



$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 5.4: Boolean and linear representations for the modified NOR structure.

The following observations are made from the modified elementary logic gates developed in this section.

1) The elementary logic gates can be modified to satisfy the criterion of equal number of input and output bits (or equal dimensionality of the respective bit state spaces).

2) The matrices corresponding to the modified gates lead to square matrices; the first step in arriving at an invertible mapping.
3) Nevertheless, the matrices are still not invertible owing to the fact that they are not full-rank matrices - that is, there always exists at least one output arising from more than one input combination, thus leading to a violation of the one-to-one and onto mapping conditions to achieve invertibility.
4) Correspondingly, observation 3 leads to the interpretation in the Boolean representation that the modified gates still do not possess enough information as regards what input combination leads to a specific output.
5) The value of the second input, which is not carried to the output end of the gate gives rise to the ambiguity since there is no book keeping procedure to record the value of the second input bit. This may be resolved by carrying over the second input also to the output end of the gate structure, but it creates an imbalance in the number of inputs and outputs of the gate (the dimensionality of the operator).

5.4 Reversible Boolean Gates with Ancilla Bits

As noticed above, merely equating number of inputs and outputs does not lead to reversible elementary logic gates. One requires some form of a labeling procedure to record the individual value of each input bit. This may be achieved through the introduction of an ancilla or book keeping bit. However, the introduction of ancilla bits to the input and/or output of the gate should not modify the gate characteristics; the transformation rules must be preserved. Therefore, the following rules must always be observed when introducing ancilla bits:

1) Introduction of the ancilla at the input or output end of the gate structure must be made in a manner such that the transformation rules characterizing the functionality of the gate are preserved. The significance of the ancilla bit is to help provide the extra information required to achieve reversibility of the gate.
2) In the linear representation, the ancilla bit helps make the matrix full-rank, thereby achieving full invertibility for the matrix corresponding to the elementary gate.

Modified NAND Gate with Ancilla

As is already known, the NAND gate being universal gate may be applied to realize any elementary logic gate. The ancilla bit will be applied to the NAND gate in manner that renders it reversible, while preserving the functionality of the NAND gate. From thereon, reversible versions of all elementary gates may be realized through compositions of the reversible NAND structure. The reversible NAND gate with the ancilla bit is represented below.

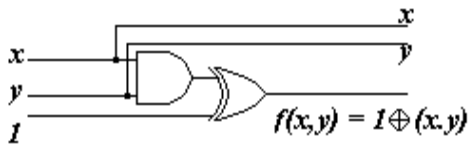


Figure 5.5: Reversible NAND gate with ancilla bit.

Table 5.1: Truth table for the modified NAND gate with ancilla bit.

x	*y*	*x.y*	*1 ⊕ (x.y)*	*(x.y)'*
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

From the reversible gate structure presented in Figure 29 it is apparent that the AND gate may be realized by simply setting the ancilla bit to the value 0.

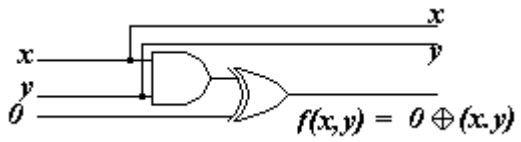


Figure 5.6: Realization of reversible AND gate by setting ancilla bit to 0.

Table 5.2: Truth table for reversible version of AND gate.

x	*y*	*x.y*	*0 ⊕ (x.y)*
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Since the same gate structure doubles as reversible AND and NAND gates through appropriate choice of the ancilla value, this gate structure will, henceforth, be referred to as the reversible AND/NAND gate. This is a three-input, three-output gate structure, therefore the bit combinations constitute a tensor product of three, two dimensional bit state spaces. The possible input combinations are provided below.

The four possible input combinations with the third bit representing the ancilla set to value 1:

$$\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1x0\\1x1\\0x0\\0x1\\0x0\\0x1\\0x0\\0x1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix} \quad (\boldsymbol{x} = 0, \boldsymbol{y} = 0, \text{ and ancilla} = 1)$$

$$\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0x0\\0x1\\1x0\\1x1\\0x0\\0x1\\0x0\\0x1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix} \quad (\boldsymbol{x} = 0, \boldsymbol{y} = 1, \text{ and ancilla} = 1)$$

$$\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0x0\\0x1\\0x0\\0x1\\1x0\\1x1\\0x0\\0x1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix} \quad (\boldsymbol{x} = 1, \boldsymbol{y} = 0, \text{ and ancilla} = 1)$$

$$\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0x0\\0x1\\0x0\\0x1\\0x0\\0x1\\1x0\\1x1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix} \quad (\boldsymbol{x} = 1, \boldsymbol{y} = 1, \text{ and ancilla} = 1)$$

The four possible input combinations with the third bit representing the ancilla set to value 0:

$$\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1x1\\1x0\\0x1\\0x0\\0x1\\0x0\\0x1\\0x0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix} \quad (\boldsymbol{x} = 0, \boldsymbol{y} = 0, \text{ and ancilla} = 0)$$

$$\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0x1\\0x0\\1x1\\1x0\\0x1\\0x0\\0x1\\0x0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\\0\\0\\0\\0\end{bmatrix} \quad (\boldsymbol{x} = 0, \boldsymbol{y} = 1, \text{ and ancilla} = 0)$$

$$\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0x1\\0x0\\0x1\\0x0\\1x1\\1x0\\0x1\\0x0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\end{bmatrix} \quad (\boldsymbol{x} = 1, \boldsymbol{y} = 0, \text{ and ancilla} = 0)$$

$$\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0x1\\0x0\\0x1\\0x0\\0x1\\0x0\\1x1\\1x0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix} \quad (\boldsymbol{x} = 1, \boldsymbol{y} = 1, \text{ and ancilla} = 0)$$

Note that the value set for the ancilla decides between two sets of mutually exclusive input combinations, one corresponding to the AND and the other to the NAND. The two sets of input vectors constitute two orthogonal subspaces of the input state space which are mutually exclusive. When the ancilla bit is set to value 0, the subspace spanned by computational basis vectors corresponding to the AND structure are activated. On the other hand, setting the ancilla bit to value 1 activates the second subspace that lies orthogonal to the AND's subspace. Therefore, in a certain sense, the ancilla does the required book keeping operation for the inputs, it also controls the selection of the appropriate subspace of the input state space to perform the AND or the NAND operation.

Applying the techniques for derivation of the linear operator in Chapter 4, the linear operator corresponding to the reversible AND/NAND gate is represented by the matrix shown below:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

It was noted that the value of ancilla bit decides selection of the subspace of the input state space to be acted upon by the linear operator. Owing to this selective property of the ancilla, the even numbered columns of the matrix correspond to the AND, and likewise the odd numbered columns correspond to the NAND. This correspondence arises from the fact that the third bit, corresponding to the ancilla, is always 0 in the even numbered columns, and is 1 in the odd numbered columns. Recall from Section 4.7 that the columns correspond to the inputs and the rows correspond to outputs.

The matrix when acting upon the four possible input combinations for the reversible NAND structure (with the ancilla bit set to value 1) leads to the following linear transformations:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$ (for $\boldsymbol{x} = 0$, $\boldsymbol{y} = 0$, ancilla = 1).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$ (for $\boldsymbol{x} = 0$, $\boldsymbol{y} = 1$, ancilla = 1).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$ (for $\boldsymbol{x} = 1$, $\boldsymbol{y} = 0$, ancilla = 1).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$ (for $\boldsymbol{x} = 1$, $\boldsymbol{y} = 1$, ancilla = 1).

Likewise, the following four linear transformations result when the matrix acts upon the possible input combinations for the reversible AND gate (with ancilla bit set to value 0):

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{(for } \boldsymbol{x} = 0, \boldsymbol{y} = 0, \text{ ancilla} = 0).$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{(for } \boldsymbol{x} = 0, \boldsymbol{y} = 1, \text{ ancilla} = 0).$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{(for } \boldsymbol{x} = 1, \boldsymbol{y} = 0, \text{ ancilla} = 0).$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \text{(for } \boldsymbol{x} = 1, \boldsymbol{y} = 1, \text{ ancilla} = 0).$$

The addition of an ancilla bit at the input stage and carrying on the second input to the output stage in addition to the first input, therefore, results in a reversible gate that doubles as a NAND and AND gate. The matrix that corresponds to this gate structure is observed to be invertible.

The following properties are noted for this structure:

1) Every input combination leads to one and only one output value, therefore satisfying the first necessary condition for invertibility by being a one-to-one map.
2) Each output combination arises from a unique input combination, implying that all elements of the state space representing the function output have a unique pre-image in the state space representing the inputs. This satisfies the onto condition for invertible maps.
3) From points 1 and 2 it becomes apparent that the gate structure satisfies invertible primitives, therefore the gate is reversible and its corresponding matrix is invertible.
4) The inverse of the matrix is the same as the matrix itself. This property holds significance in the paradigm of reversible computing and its relationship with quantum computing which will be discussed later in the document.

5) Given that the state spaces representing the inputs and outputs share the same dimensionality, both spaces have the same set of computational basis vectors. Moreover the one-to-one and onto conditions for mapping necessarily lead to transformations that constitute permutations between computational basis vectors. This property is inextricably related to point 4 above, the relation will be discussed later in the document.
6) Explaining point 5 above in terms of the Boolean representation, the number of inputs and outputs being equal, the possible values of input and output combinations are same. The condition that each input lead to a unique output and all possible output combinations be realized by the function leads to a situation where the output of the function is either equal to the input combination provided, or is a circular permutation of the input combination. Consider the specific case of the AND/NAND matrix above. There are eight possible input combinations, and there are eight unique output combinations. Of these six outputs equal their inputs, in case of the other two outputs, they are circular permutations of their input values.
7) The distance of the element '1' in each row or column from the diagonal element position in that particular row or column marks the Hamming distance of that particular bit. For example, row 7 has element '1' separated from the diagonal element of that row by one position. This implies the Hamming distance between value of the input and that of the output is 1. This property is related to the transformation rules defined by the matrix. For example, consider a column of the matrix corresponding to a particular input combination. If the position of the element 1 in that column lies on the diagonal (that is the element 1 of the column lies on a row corresponding to the same input value), then the transformation is equivalent to an identity transformation (because the input and

output have the same value). Instead, if the element 1 is placed away from the diagonal (that is the row corresponds to a circular permutation of the input value) then the transformed output value is a permutation of the input value

Similar to the reversible AND/NAND gate structure, a similar OR/NOR structure may be designed where the selection of the ancilla value decides whether the structure operates as a reversible OR gate or a NOR gate. The symbolic Boolean representation of the two reversible gate structures is presented below.

Figure 5.7: The reversible OR/NOR gate structure.

The corresponding matrix derived using the techniques outlined in Chapter 4 is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

5.5 Attributes of Reversible Functions

This section details some attributes that characterize linear operators representing reversible circuits. An independent verification of Toffoli's fundamental theorem with respect to the existence of a reversible version for every irreversible logic function is also provided.

5.5.1 The Fundamental Theorem

In a technical memo submitted to the Laboratory of Computer Science, MIT [6], Toffoli proved that for every irreversible logic function in the classical paradigm of computing, there exists a corresponding equivalent reversible function in the paradigm of reversible computing. This theorem forms the definitive thread that ties these two paradigms. An independent verification of this theorem in the *linear representation* occurs when analyzing some properties of the matrices that correspond to reversible logic functions. Toffoli's proof of the Fundamental Theorem of Reversible Computing follows:

Statement of Theorem: For a given Boolean logic function in the classical paradigm, there exists a corresponding functionally equivalent reversible Boolean logic function in the reversible paradigm.

Proof of Theorem:

For sake of simplifying the arguments to follow, we introduce a shorthand notation that was followed by Toffoli. Consider a string of $\boldsymbol{m}$ bits. Let the set of all bit combinations $000\ldots00_m$, $000\ldots01_m$, through $111\ldots11_m$ be represented by the symbol $\boldsymbol{X}$.

Consider an arbitrary irreversible Boolean function **F** with ***m*** input bits and ***n*** output bits.

- The truth table for the function **F** would be of the form:

$$\boldsymbol{X} \rightarrow \boldsymbol{Y}.$$

The set ***X*** obviously represents all the valid input combinations to the Boolean function **F**. The set ***Y*** represents the outputs that result for the values of the set of inputs characterized by ***X***. Note that the entries in ***Y*** could repeat themselves since each input combination of ***X*** may not lead to a unique output value (a simple example is the case of the two input AND gate where three input combinations 00, 01 and 10 lead to the same output value 0, where as the output value 1 results only for the input combination 11).

- Define a Boolean function $\mathbf{F_{in}}$ with $(\boldsymbol{m}+\boldsymbol{n})$ input bits and $(\boldsymbol{m}+\boldsymbol{n})$ output bits.
- Let the Boolean function $\mathbf{F_{in}}$ be characterized by a truth table with the following transformation rules:

Inputs → Outputs

0	***X***
1	***X***
…	***X***
2^n-1	***X***

Y	***X***
Y + **1**	***X***
…	***X***
Y + $\mathbf{2^n}$**-1**	***X***

- Note that each cell in the table above represents $2^{\boldsymbol{m}}$ entries of the truth table, and there are n such cells. Therefore the total number of entries in the table are $2^{\boldsymbol{m+n}}$, which is the expected number of entries (since there are $2^{\boldsymbol{m+n}}$ input combinations in total).

Toffoli's argument proceeds in the following manner:

- Each cell of the form $\boldsymbol{k}$ ($0 \leq \boldsymbol{k} < 2^{\boldsymbol{n}}$) consists of $2^{\boldsymbol{m}}$ identical bit strings, each representing the integer $\boldsymbol{k}$ written in base 2.
- Each cell of the form $\boldsymbol{Y} + \boldsymbol{k}$ ($0 \leq \boldsymbol{k} < 2^{\boldsymbol{n}}$) consists of $2^{\boldsymbol{m}}$ entries of $\boldsymbol{Y}$, each treated as a base 2 integer and incremented by $\boldsymbol{k} \bmod 2^{\boldsymbol{n}}$.
- Therefore the sequence of "$\boldsymbol{Y} + \boldsymbol{k}$" cells differs from the "$\boldsymbol{k}$" sequence by a circular permutation.
- By construction each side of the truth table contains each combination of the bit string of length ($\boldsymbol{m}+\boldsymbol{n}$) exactly once.

Therefore **F** has a corresponding reversible form, and is represented by the function $\mathbf{F}_{\mathbf{in}}$.

5.5.2 Common Properties of the Linear Operators.

Linear Operators representing Reversible Functions form a Group.

1) The constraint of invertibility placed on linear operators representing reversible functions translates to the following conditions being imposed upon the linear operators:
   - The dimensionality of the input state space equals that of the output state space.
   - Therefore, they have the same set of computational basis states.
   - The one-to-one and onto conditions dictate that each computational basis state vector in the input state space map to a unique computational basis state vector in the output state space, and each computational basis state vector in the output state space has a unique pre-image in the input state space (i.e. every computational basis state vector in the output state space is being mapped into).

2) This implies that the linear transformation representing the reversible gate is essentially a permutation operation from one computational basis state to another. Geometrically, this may be interpreted as a set of orthogonal rotational transformations of the bit state vector (since computational basis states are mutually orthonormal).
3) The set of matrices that represent such permutation transformations are called isometric transformations, and form a group under multiplication and tensor direct products. They are referred to as permutation matrices, since the columns (or rows) vary from the identity by a permutation of their respective positions. [11].
4) Isometric transformations in the real Euclidean space are referred to as permutation operators. But the bit state space was defined to be a two dimensional complex Euclidean space. Isometric transformations in complex Euclidean spaces are referred to as unitary operators. Since the real space forms a subspace of the complex space, it logically follows that permutation operators form a sub group of the unitary group. Unitary operators are characterized by an interesting property. Given an arbitrary unitary matrix $\mathbf{U}$, its inverse $\mathbf{U}^{-1}$ equals its Hermitian complex conjugate (the transpose of the complex conjugate) represented by $\mathbf{U}^{\dagger}$, i.e. $\mathbf{U}^{-1} = \mathbf{U}^{\dagger}$.
5) It must be noted that not all unitary operators represent classical reversible functions. There are unitary matrices that transform the bit state vector such that it forms a normalized linear combination of the computational basis vectors. Since a classical bit may not take up such superposition states, such unitary matrices do not represent any classical reversible logic function.
6) The unitary matrices that correspond to classical reversible functions must necessarily have only one element per row and column that is non-zero (it may be real or imaginary). Such

matrices form a subgroup of the unitary group. This property has major implications when relating the classical, reversible, and quantum paradigms of computing as will be seen later in the document.

Functions constructed with Reversible Elements are Reversible.

1) All elementary reversible Boolean gates may be represented by three linear operators:
   - The inverter or NOT gate is represented by $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
   - The AND/NAND gate structure is represented by $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$.
   - The OR/NOR gate structure is represented by $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$.
2) Any arbitrary logic function may be constructed through a composition of elementary Boolean gates.
3) The composition of elementary logic gates in the Boolean representation corresponds to matrix multiplication or the tensor direct product of matrices in the linear representation.

4) The three matrices corresponding to NOT, AND/NAND, and OR/NOR share a common attribute. They belong to the set of permutation matrices.
5) The matrix resulting from the product or tensor direct product of the three matrices also belongs to the set of permutation matrices since they form a group.
6) Hence, any reversible logic function resulting from a composition of reversible elementary Boolean gates has a corresponding matrix that is a permutation resulting from the product or tensor direct product of permutation matrices.

Please note that in the process of showing two important attributes that characterize linear operators representing reversible functions, we validate Toffoli's fundamental theorem and thereby provide independent confirmation, we do not provide an alternative proof.

Eigenvectors coincide with Truth Table Entries of Logic Functions.

A relationship can be observed between the eigenvectors and truth table entries of these mappings. Consider the following matrix and their corresponding eigenvectors.

<u>For non-invertible modified AND gate:</u>

The matrix corresponding to the modified non-invertible AND gate is $\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

The eigenvectors for this matrix are $\lambda=1, \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$; $\lambda=0, \begin{bmatrix}-1/\sqrt{2}\\1/\sqrt{2}\\0\\0\end{bmatrix}$; $\lambda=1, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$; $\lambda=1, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$.

Consider the truth table for the modified non-invertible AND gate.

Table 5.3: Truth table for modified non-invertible AND gate.

x	***y***	***x,(x.y)***
0	0	00
0	1	00
1	0	10
1	1	11

This truth table may be alternatively represented in the *linear representation*. The input combinations derived from the input variables ***x*** and ***y*** form a four element column vector. Likewise, the output combinations derived from the original input variable ***x*** that is led to the output stage and the function output variable ***f(x,y)***. The truth table in the *linear representation,* with the eigenvectors corresponding to the respective truth table entries is shown in the tabular listing below.

Table 5.4: Truth table for modified in *linear representation* with corresponding eigenvectors.

$x \otimes y$	$x \otimes f(x,y)$	Corresponding Eigenvector
$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$
$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$	$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\\0\\0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} - \frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

The first two columns in Table 5.4 correspond to the *linear representation* of the original truth table provided in Table 5.3. The third column represents the eigenvectors derived from the linear operator that correspond to the specific row entries in the truth table. Upon analyzing individual rows of Table 5.4, it is observed that the eigenvector forms a normalized linear combination of the input and output combinations. Entries in the first, third and fourth rows of this truth table are trivial cases, since the input combinations in the respective rows lead to output

combinations with the same value (this implies, in first row, input combination '00' results in output '00', likewise in third row the input combination '10' results in the output '10', and similarly the fourth row gives rise to input combination '11' and an output combination of '11' too). The second row in this table is the special case, where the input combination is '01', while the resulting output is value '00'. Please note, that the corresponding eigenvector in this case is a normalized linear combination of the column vectors that represent the function's input and output.

For the Modified Non-invertible OR gate:

The matrix corresponding to the modified non-invertible OR gate is $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$.

The eigenvectors for this matrix are $\lambda=1, \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$; $\lambda=1, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$; $\lambda=1, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$; $\lambda=0, \begin{bmatrix} 0 \\ 0 \\ 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix}$.

Table 5.5: Truth table for modified non-invertible OR gate.

x	***y***	***x,(x+y)***
0	0	00
0	1	01
1	0	11
1	1	11

For modified non-invertible NAND gate:

The matrix corresponding to the non-invertible NAND gate is $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

The eigenvectors are $\lambda=1, \begin{bmatrix} 0 \\ 0 \\ 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$; $\lambda=-1, \begin{bmatrix} 0 \\ 0 \\ 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix}$; $\lambda=1, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$; $\lambda=0, \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \\ 0 \\ 0 \end{bmatrix}$.

Table 5.6: Truth table for modified non-invertible NAND gate.

x	***y***	***x,(x.y)'***
0	0	01
0	1	01
1	0	11
1	1	10

For modified non-invertible NOR gate:

The matrix corresponding to the non-invertible NOR gate is $\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$.

The eigenvectors are λ=1, $\begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \\ 0 \end{bmatrix}$; λ=-1, $\begin{bmatrix} 0 \\ 0 \\ 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix}$; λ=1, $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$; λ=0, $\begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \\ 0 \\ 0 \end{bmatrix}$.

Table 5.7: Truth table for modified non-invertible NOR gate.

x	***y***	***x,(x+y)'***
0	0	01
0	1	01
1	0	10
1	1	10

The reversible AND/NAND structure:

The matrix corresponding to the reversible AND/NAND gate structure is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The eigenvectors corresponding to this matrix are:

$$\lambda=-1,\begin{bmatrix}0\\0\\0\\0\\0\\0\\-1/\sqrt{2}\\1/\sqrt{2}\end{bmatrix};\ \lambda=1,\begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix};\ \lambda=1,\begin{bmatrix}0\\0\\1\\0\\0\\0\\0\\0\end{bmatrix};\ \lambda=-1,\begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix};\ \lambda=1,\begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\end{bmatrix};\ \lambda=1,\begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix};\ \lambda=1,\begin{bmatrix}0\\0\\0\\0\\0\\0\\1/\sqrt{2}\\1/\sqrt{2}\end{bmatrix};\ \lambda=1,\begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}.$$

Table 5.8: Truth tale for reversible AND/NAND gate.

x	***y***	***ancilla***	***x,y,ancilla⊕(x.y)'***
0	0	0	000
0	0	1	001
0	1	0	010
0	1	1	011
1	0	0	100
1	0	1	101
1	1	0	111
1	1	1	110

The same observation regarding the correspondence shared between eigenvectors of the matrix and entries of the truth table can be made in case of the modified non-invertible OR, NAND and NOR gates, and the reversible AND/NAND structure.

It is quite interesting to note that the eigenvectors of these matrices turn out to be normalized linear combinations of the input and output state vectors. In effect the set of eigenvectors provides all the information that may be derived from the truth table of the corresponding logic function.

The relationship noted between eigenvectors and the entries of the truth table cannot be mere coincidence. There ought to exist a rational explanation for such a relationship. A logical explanation for this correspondence does exist, and it provides a very clear understanding of the role physics plays in the functionality of these circuit structures. The mathematical formalism and definition of the eigenvector corresponds to the physical phenomenon of resonance. Would it imply that the operation of gate structures and circuits that realize an algorithm, and, therefore, the physical process of computation is itself a resonance phenomenon? The definitive answer to the question lies in the dynamical evolution of a computational process from its start to finish. We provide some arguments that lead to an answer to this question.

1) In Chapter 1 it was shown that a circuit is the physical implementation of the abstract algorithm that solves a given computational problem. It was, therefore, interpreted that the operation of the circuits to solve the problem constitutes the physical computational process.

2) The physical computational process is a dynamical evolution of the computational system in discrete time steps from the time instant when the inputs are fed to the system to the point where the output is made available.
3) In physics, the dynamical evolution of a physical system is defined by its Hamiltonian. The Hamiltonian of a given system provides all the required information related to the state of the system at a given time instant. The linear operators corresponding to the Boolean functions also provide the same information about the system. The linear operators representing reversible circuits in the *linear representation* are time evolution operators that define the state of the system at a given time instant in the discrete time evolution process of computation. Applying the operator to the initial state of the system (i.e. the inputs) and specifying the time instant at which the state of the system is to be recorded, is equivalent to running the Hamiltonian between start and stop times of the computational process.

Therefore, the relationship noted between eigenvectors of the linear operators and the entries of the truth table is no mere coincidence, but is the result of a physical principle that defines the discrete time evolution of the computational system. This interpretation bears a greater significance on a computational process. It is well understood that the complexity class of an algorithm depends upon how the algorithm behaves in terms of its execution time for the number of inputs provided, and the computational resources required. This implies the complexity class to which an algorithm belongs is essentially a function of the discrete time evolution of the physical system implementing the algorithm for the number of inputs provided and resources utilized. But, the discrete time evolution of the physical system is defined by the Hamiltonian, so diagonalizing the Hamiltonian to resolve its eigenvalues and corresponding eigenvectors

should give an indication of what complexity class the algorithm belongs to. Unfortunately that is not so, owing to the fact that diagonalizing the Hamiltonian of a computational system to find its eigenvectors is of the same computational complexity class as actually evolving the system represented by the Hamiltonian, providing some inputs in order to find the solution[2].

This concludes a discussion of some attributes of reversible logic functions and the linear operators that correspond to them.

5.6 Reversible Combinational Circuits

Reversible combinational circuits are constructed in the same manner as their classical irreversible counterparts discussed in Section 4.5. Reversible combinational circuits are constructed through compositions of reversible elementary Boolean gates. Therefore, the correspondence shared in the *linear representation* amounts to constructing a linear operator that is built through products and tensor direct products of the corresponding invertible linear operators. For instance, consider a function $f(x, y) = x.y'$.

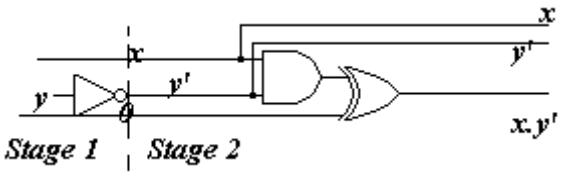


Figure 5.8: Symbolic representation of reversible function $f(x, y) = x.y'$.

[2] This complication was understood during private communication with Prof. Neil Gershenfeld, Media Lab, MIT.

This function is developed in the *linear representation* in the following manner:

Multiply the linear operator corresponding to the reversible AND/NAND structure corresponding to Stage 2 with the tensor product of (**I** ⊗ **NOT** ⊗ **I**) in Stage 1.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} =$$

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

This matrix represents the linear operator corresponding to the reversible function ***f*(*x, y*) = *x.y'***.

In this manner, any reversible combinational function can be constructed by following the standard matrix composition rules applicable to the *linear representation*.

5.7 Reversible Sequential Circuits

We have so far studied a class of combinational circuits with respect to the construction of reversible logic functions. The same may be extended to the other class of circuits represented by sequential networks. Sequential circuits were studied in the classical context in Section 4.6. Sufficient arguments were provided in favor of extending the *linear representation* to include sequential circuits.

The extension of reversible functions to include sequential circuits may be achieved in a simple manner.

1) The presence of the memory element in the feedback loop characterizes the primary distinction of sequential circuits from combinational functions.
2) It is already known from the previous section that every irreversible function can be represented by a functionally equivalent reversible function.
3) Therefore, if the combinational part of the sequential functions were to be isolated from the memory feedback element and replaced by its reversible function, it would constitute a reversible sequential circuit.
4) This has no bearing upon the functionality since it is preserved when replacing the irreversible logic function by its reversible counterpart.
5) Please note that the memory feedback element plays no role in processing of the input. As noted earlier, it strictly demarcates the boundary between the current stage and the

subsequent one. The functional aspects of the circuit are exclusively controlled by the combinational part of the function scheme.

In the *linear representation*, the substitution of the non-invertible linear operator $\mathbf{L_F}$ by the invertible unitary operator representing the classical reversible function say, $\mathbf{U_F}$ constitutes a reversible sequential circuit. A comparison of the changes introduced in a reversible sequential circuit with respect to its standard irreversible form is presented below in terms of a generic block diagram.

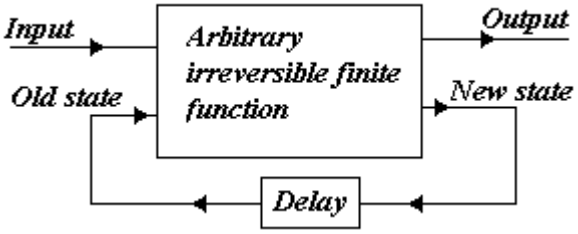


Figure 5.9: A block diagram representation of an arbitrary sequential circuit or finite automaton.

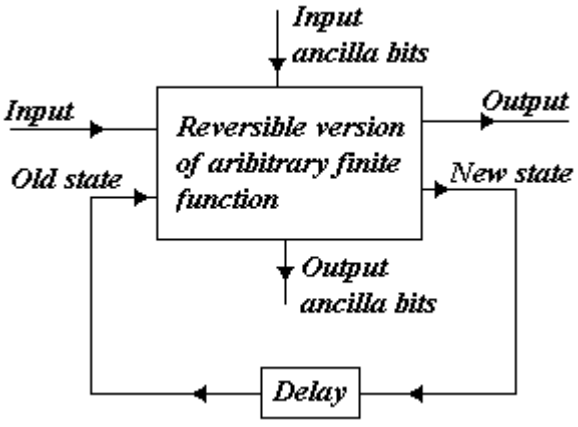


Figure 5.10: Reversible version of the arbitrary sequential circuit in Figure 5.9.

If a reversible sequential circuit were to go through ***n*** iterations and lead to a certain output, feeding the output as the original set of inputs back into the sequential circuit and going through a set of ***n*** reverse iterations would therefore lead to the original set of inputs that were fed into the sequential circuit.

The abstract mathematical models of computational systems defined in finite automata theory consist of three sets, the set of input variables, the set of output variables, and the state of the system. A finite automaton is fed in with a set of input variables that are acted upon by the finite state machine that provides the mathematical description of the computational system. The functional processing scheme is characterized by the discrete time evolution of the finite state machine as it moves from one state to the next, depending upon the current set of values held by the input variables, as they undergo transformation at every state. By definition, a finite automaton is reversible if its transition function is invertible. The sequential networks in the circuit model representation correspond to finite automatons, and, therefore, the transition function of the automaton corresponds to the linear operator in the linear representation.

It has been shown that reversible sequential networks may be constructed by replacing their irreversible combinational logic functions by the corresponding reversible counterparts. This implies a reversible finite automaton can be similarly constructed, by replacing its non-invertible transition function by the corresponding bijective (invertible) transition function. In such manner any finite automaton may be made reversible through the introduction of invertible primitives to the transition function that provides the mathematical description of the system. Hence, whatever may be computed by an arbitrary finite automaton may be computed by a

reversible construction of the finite automaton. Therefore, it logically follows that the invertible transition function of a reversible finite automaton is basically an invertible linear operator characterized by a unitary matrix.

5.8 Reversible Universal Turing Machine

The structure and operation of the Turing machine was introduced in Chapter 3. It was shown that for every computable function one can specify a Turing machine that computes it via a sequence of steps dictated by the transition function or table. The same was extended to define the universal Turing machine that can carry out a computation that may be performed by any specific Turing machine.

Sufficient arguments were provided in the previous section in support of the fact that the transition function of any given finite automaton can be replaced by its invertible version, thus rendering the finite automaton reversible. From this it follows that a Turing machine is considered reversible if its transition function is invertible. In the linear representation this implies the linear operator corresponding to the Turing machine in question must be invertible and more specifically an element of the unitary group, in order for the Turing machine to be considered reversible.

This raises the question of whether it is possible to generalize the inclusion of invertible primitives and composition rules to include all classes of Turing machines. If this were possible, it would have major implications to the field of computing, since it effectively tells us that any problem with a computable solution in the classical paradigm can be computed in the reversible

paradigm. The answer to this question depends upon whether a reversible universal Turing machine is possible, since it contains the descriptions for all classes of Turing machines.

The proof in favor of the reversible universal Turing machine was provided by Charles Bennett in 1973 [5] where he showed that, given an ordinary Turing machine **S**, one can construct a reversible three-tape Turing machine **R**, from where it is easy to define a reversible universal Turing machine that contains instances of all classes of reversible Turing machines. Bennett's arguments in favor of the proof are enumerated below

- The usually studied single tape Turing machine is comprised of a control unit, a read/write head, and an infinite tape divided into squares. The behavior of the machine is governed by the rules in the transition table or the transition formulae, generally referred to as quintuples. These quintuples have the form

  $$\boldsymbol{AT \rightarrow T' \sigma A'}$$

  This implies, if the control unit is in state ***A*** and head scans the tape symbol ***T***, the head first writes ***T'*** in the place of ***T***; then shifts left one square of the tape, to the right one square or remains in the same position depending upon the value of $\boldsymbol{\sigma}$ (-, +, or 0, respectively). The control unit finally reverts to state ***A'***.
- Every quintuple defines a partial one-to-one mapping of the present machine state consisting of information regarding tape contents, head positions and control state onto its successor and, therefore, is as such deterministic and reversible.
- Stemming from the observation above, Bennett provided two constraints for a Turing machine to be deterministic and reversible. The first constraint states that a Turing machine is deterministic if and only if its quintuples have no overlapping domains. The

second constraint states that in order for the Turing machine to be reversible, its quintuples may not have overlapping ranges.

These two constraints coincide with the invertibility criteria specified at the start of this chapter for a linear operator to successfully represent a reversible function where the domain space corresponds to the input state space and the range space to the output state space. The two criteria were that the mapping must be one-to-one and onto. The one-to-one mapping condition ensures that the elements in the domain space have unique mappings hence avoiding an overlap where as the onto condition requires that there exist no elements in the range space that are mapped to multiple elements in the domain space, thereby eliminating any possible overlap of range space.

- It was shown earlier that all linear operators representing reversible functions satisfy the invertibility criteria, and moreover that all such linear operators are elements of the unitary group.
- Secondly, if one were to define a Turing machine for an arbitrary reversible function in the linear representation, the linear operator of the Turing machine would coincide with the linear operator representing the function itself. This is validated by the argument that the truth table (or transition table in case of sequential networks and finite automata) of the function would coincide with the corresponding operations on the Turing machine.
- By definition, the universal Turing machine is capable of computing a function that is deemed computable in the classical paradigm. Then a reversible universal Turing machine comprises the set of all linear operators in the unitary group that correspond to

reversible functions (given the fact that every computable function in the classical paradigm has a reversible function, by virtue of Toffoli's fundamental theorem).

Hence, the arguments provided above in favor of the reversible universal Turing machine in the *linear representation*, provide an independent check of Bennett's original proof. Please note that we do not provide an alternate proof to what was originally proved by Bennett. The arguments summed above provide a logical progression that leads to a verification of Bennett's proof. Second, it must be borne in mind that the entire unitary group does not represent classical reversible functions; there are certain elements in the unitary group with no corresponding classical reversible functions associated with them. These linear operators represent quantum circuits which cannot be replicated classically, and their study forms the main aspect of the next section on the paradigm of quantum computation. The interesting relationship between reversible and quantum paradigms of computing is also understood in the context of universal computational models when the reversible universal Turing machine is related to the universal quantum computer model.

5.9 Quantum Computation

The paradigm of quantum computing may be defined as the study of computational systems that operate by means of physical principles embodied within the theory of quantum mechanics. An introduction to the quantum bit or qubit, and the nature of information processing tasks in quantum computing were provided in Chapter 1. This section provides a more rigorous description of the qubit and the unique properties that characterize quantum circuits as a result of the qubit being allowed to take on superposition states. Given the

possibility of the qubit to exist as a linear combination of the computational basis vectors, quantum circuits are, by default, described in the *linear representation*. The insights gained about quantum circuits are generalized to understand universal computational models as outlined in the definition of the universal quantum computer model defined by David Deutsch. The relationship between the classical and quantum contexts becomes clear upon analyzing their common attributes.

5.9.1 Quantum Bits

A brief introduction of the qubit was provided in Section 1.4. This section provides a more rigorous definition of the qubit and reconciles this definition with the *linear representation*. The relationship between the classical and quantum bit becomes clear once the difference in the structure of the respective bit state spaces is understood.

During the development of bit state space to represent the classical bit in Section 4.2, a condition was imposed on its operation (Condition 6). The condition applied was that the bit state vector may not exist as a linear combination of the computational basis vectors. The qubit is characterized by the absence of this condition upon its representative bit state space. By doing so the bit state vector of the qubit is allowed to span the entire bit state space by taking on normalized linear combinations of the computational basis vectors. The removal of this condition necessitates that the bit state space be redefined.

The qubit is mathematically represented in a two dimensional complex Euclidean Hilbert space. By Hilbert space, it is implied that the vector space in question is complete and dense,

and every vector in this space can be represented through a linear combination of the basis vectors. A state space that contains vectors or elements that cannot be represented in terms of its basis vectors is not complete or dense, and, therefore, does not constitute a Hilbert space. For sake of consistency, the state representing the qubit will be referred to as the qubit state space in order to differentiate it from the bit state space that represents the classical bit.

The value of the qubit is characterized by the qubit state vector. It is a vector of unit magnitude that is allowed to span the entire qubit state space by taking on linear combinations (or superpositions) of the computational basis vectors.

We will now resolve the relationship between the classical and quantum bits through an example. Consider a sample qubit of the form $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. This may be represented as follows in the qubit state space.

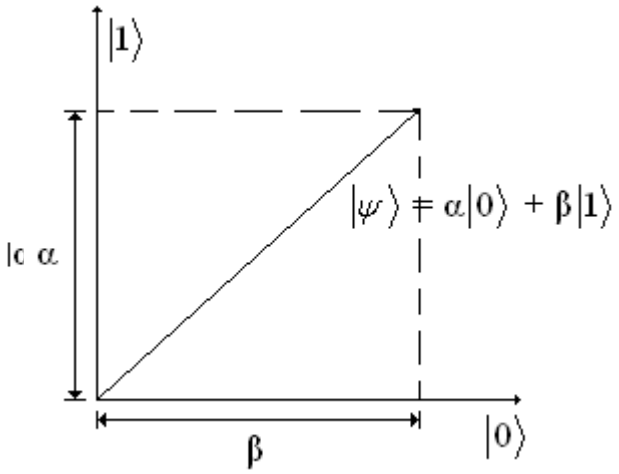


Figure 5.11: Qubit state space representation of $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$.

In Figure 5.11 above, the components α and β constitute the components of the qubit state vector $|\psi\rangle$ along the computational basis vectors $|0\rangle$ and $|1\rangle$ respectively. The qubit state vector is normalized, and hence, carries unit magnitude. As a result of the normalization condition, $|\alpha|^2 + |\beta|^2 = 1$. This gives rise to the interpretation that the qubit state vector is in state $|0\rangle$ with probability $|\alpha|^2$, and in state $|1\rangle$ with probability $|\beta|^2$. The normalization condition imposed upon the qubit state vector leads to satisfaction of the condition that the sum of all probabilities equal 1. This example constitutes a generic interpretation of the qubit state space and the values of the qubit state vector. Consider two special cases, one where the qubit state vector is $|\psi\rangle = 1|0\rangle + 0|1\rangle$ or $|\psi\rangle = i|0\rangle + 0|1\rangle$. The probability of $|\psi\rangle$ being in state $|0\rangle$ is $|1|^2 = 1$ or $|i|^2 = 1$ and in state $|1\rangle$ is 0. This particular state of qubit $|\psi\rangle$ coincides with that of the classical bit when it has value 0. Similarly consider the second special case where the qubit state vector is $|\psi\rangle = 0|0\rangle + 1|1\rangle$ or $|\psi\rangle = 0|0\rangle + i|1\rangle.|1\rangle$. The probability of $|\psi\rangle$ being in state $|0\rangle$ is 0 and in state $|1\rangle$ is $|1|^2 = 1$ or $|i|^2 = 1$. This particular state of qubit $|\psi\rangle$ coincides with that of the classical bit when it has value 1. From this, it is inferred that that two states of the bit state vector (in classical computation) form two special cases of the states that a qubit state vector (in quantum computation) can take up.

Instead of working with specific instances, the case of an arbitrary ***n*** number of qubits will be studied, the state of which is a unit vector in the complex Hilbert space $\boldsymbol{C^2} \otimes \boldsymbol{C^2} \otimes \ldots \otimes \boldsymbol{C^2}$ (recall the description of the Hilbert space in Section 3.4.2). The basis consisting of 2^n computational basis vectors, therefore forms the natural basis of this space, where the

computational basis vectors of the composite state space can be derived from a tensor direct product of the computational basis vectors of the individual bit state spaces as shown below.

$$|0\rangle \otimes |0\rangle \otimes \ldots \otimes |0\rangle$$

$$|0\rangle \otimes |0\rangle \otimes \ldots \otimes |1\rangle$$

.

.

$$|1\rangle \otimes |1\rangle \otimes \ldots \otimes |1\rangle$$

The treatment of multiple qubits in the *linear representation* extends in the same manner as was seen in case of the classical bit. Multiple qubits are represented through tensor products of their representative qubit state spaces. For brevity, the tensor direct product is omitted, and a more short hand formalism (this notation is quite common in the domain of quantum computing) of the following nature is adopted.

$$|i_1\rangle \otimes |i_2\rangle \otimes \ldots \otimes |i_n\rangle = |i_1, i_2 \ldots, i_n\rangle \equiv |i\rangle$$

where $\boldsymbol{i_1, i_2, \ldots, i_n}$ is the binary representation of the integer $\boldsymbol{i}$, a number between **0** and $\mathbf{2^n-1}$. This is an important step, as this representation allows one to apply a generic quantum system to encode integers, therefore such a quantum mechanical system starts being a computer. The representation of the general state that describes this system can also be extended from the single qubit case to multiple qubits without loss of generality. In the generic case of $\boldsymbol{n}$ qubits the general state describing the system is a complex unit vector in the Hilbert space and is represented as:

$$\sum_{i=0}^{2^n-1} c_i \left| i \right\rangle$$

where $\sum_i |c_i|^2 = 1$. The relationship shared between classical bits and qubits becomes obvious at this point. Given a set of ***n*** qubits, the qubits take on the values of the classical bits when the bit state vector representing the qubit lies on any one of the computational basis states. This may be represented as follows:

$$i_1 i_2 \ldots i_n \leftrightarrow \left| i_1 \right\rangle \otimes \left| i_2 \right\rangle \otimes \ldots \left| i_n \right\rangle = \left| i_1 i_2 \ldots i_n \right\rangle .$$

Therefore, the relationship observed between the classical bit and qubit may be carried forward to a generic system comprising an arbitrary number of qubits without loss of generality.

5.9.2 Quantum Circuits

The mechanism of performing computational tasks through application of logic gates is well understood in the classical context. This gives rise to the question of what constitutes a quantum circuit. A quantum circuit is a system built of two state quantum particles, the qubits.

Suppose one wants to compute a function of the form ***f:*** $i_1 i_2 \ldots i_n \rightarrow \boldsymbol{f}(i_1 i_2 \ldots i_n)$. The quantum mechanical system that computes the system evolves in accordance with the time evolution operator **U** such that:

$$\left| i_1 i_2 \ldots i_n \right\rangle \rightarrow \mathbf{U} \left| i_1 i_2 \ldots i_n \right\rangle = \left| f(i_1 i_2 \ldots i_n) \right\rangle .$$

The requirement, therefore, is to find the suitable Hamiltonian for the system that would generate its evolution in accordance with the Schrödinger equation:

$$i\hbar \frac{d}{dt}\left|\psi(t)\right\rangle = \mathbf{H}\left|\psi(t)\right\rangle .$$

Recall the treatment related to the dynamics of a quantum mechanical system in Section 3.4.4. This implies that one has to solve for the suitable Hamiltonian, **H**, given the desired unitary operator **U**:

$$\left|\psi_f\right\rangle = e^{(\frac{-i}{\hbar}\int \mathbf{H}dt)}\left|\psi_0\right\rangle = \mathbf{U}\left|\psi_0\right\rangle .$$

It is understood that as long as the linear operator **U** being applied stays unitary, a solution always exists for the Hamiltonian (A good example in favor of this argument in the context of the classical bit, is the case of reversible functions where a unique solution always exists for any arbitrary input combination. This feature arises as a result of the invertibility criteria met by these functions. On the other hand a good counter example is that of linear operators representing classical irreversible functions where the matrices are not invertible. Therefore, a unique solution does not arise, as is apparent from the fact that multiple inputs lead to the same output for a given function). It is important to pay attention to the unitarity restriction. Note that the quantum analog of a classical operation is unitary only if ***f*** is one-to-one and onto, or reversible. This explains the reason why reversible classical functions can be implemented by a physical Hamiltonian (this also explains why a relationship was observed in case of the linear operators representing classical reversible functions whereby their eigenvectors coincided with the function's truth table entries, but not in the case of classical irreversible functions where no eigenvectors can be computed in the first place). A direct result derivable from this relationship is the fact that quantum systems can imitate all computations that can be performed by classical systems (since all classical functions have corresponding reversible

counterparts, and the class of linear operators representing all classical reversible functions form a subgroup of the unitary group). The interest in quantum computation does not arise from the fact that it can solve all computable classical functions, but primarily because it can achieve more than that.

For the sake of definiteness let it be assumed that for an input string ***i***, the initial state of the system is $|i\rangle$:

$$\boldsymbol{i} \rightarrow |i\rangle$$

Elementary operations will be performed on the system, where the operations correspond to the computational steps similar to the classical logic gates that constitute elementary operations in classical computers. The interaction of the external environment on a quantum mechanical system leads to an irreversible process called quantum decoherence, due to which the entire system collapses and is rendered incapable of performing computational operations. Moreover the description of the system's evolution in time becomes difficult as it cannot be represented by a unitary operator. Therefore, it is assumed that all operations are performed on an isolated system, so the evolution can always be described by a unitary operator or matrix operating on the state of the system. Recall from Section 3.3.4 on the complex Euclidean structure that a unitary operator satisfies the condition $\mathbf{U}\mathbf{U}^{\dagger} = \mathbf{I}$.

Definition: A quantum gate acting upon **n** qubits is a unitary matrix **U** of dimensions 2^n x 2^n. Here is an example of a simple quantum gate, operating on one qubit.

$$\mathbf{NOT} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Recalling from standard Dirac notation introduced in Section 3.4.3 that $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, we have that $\mathbf{NOT}|0\rangle = |1\rangle$ and $\mathbf{NOT}|1\rangle = |0\rangle$. Hence, this gate flips the bit, and thus it is justified to call this gate the **NOT** gate. The **NOT** gate can operate on superpositions as well. From linearity of the operation,

$$\mathbf{NOT}(c_0|0\rangle + c_1|1\rangle) = c_0|1\rangle + c_1|0\rangle .$$

When the **NOT** gate operates only upon the first qubit in the system of ***n*** qubits, in the state $\sum_i c_i|i_1 i_2 \ldots i_n\rangle$ this state transforms to $\sum_i c_i(NOT|i_1\rangle)\,|i_2 \ldots i_n\rangle = \sum_i c_i|\neg i_1 i_2 \ldots i_n\rangle$. The time evolution of the system is described by a unitary matrix, which is a tensor product of the gate operating on the first qubit and the identity matrix **I** operating on the rest of the qubits as shown below:

$$\mathbf{U_{NOT1} = NOT} \otimes \mathbf{I_2} \otimes \mathbf{I_3} \ldots \otimes \mathbf{I_n}$$

Note that the linear operator representing the classical NOT gate coincides with the analogous quantum version. This is yet another instance where the relationship between the classical and quantum paradigms is seen explicitly. Please do note that this relationship exists on account of the fact that the classical NOT gate is by nature reversible.

Another important quantum gate is the controlled **NOT** gate acting on two qubits, which computes the classical function: $(a, b) \rightarrow (a, a \oplus b)$ where $a \oplus b = (a + b) \bmod 2$ and $a, b \in 0, 1$. This function can be represented by the matrix operating on all 4 configurations of 2 bits:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This gate coincides with the classical reversible XOR gate. The XOR gate applies a NOT on the second bit, called the target bit, conditioned that the first control bit is 1. If a black circle denotes the bit being conditioned upon, the XOR gate may then be denoted by:

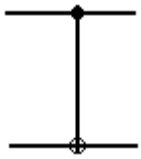


Figure 5.12: Symbolic representation of CNOT gate.

In the same manner, all classical Boolean functions can be transformed to quantum gates. The matrix representing a classical gate which computes a reversible function, (in particular the number of inputs to the gate equals the number of outputs) is a permutation on all the possible classical strings. Such a permutation was shown to be unitary when discussing properties of linear operators representing reversible classical functions in Section 5.5.2. Of course, not all functions are reversible, but they can easily be converted to reversible functions, by writing down the input bits instead of erasing them as was proved by Toffoli in his fundamental theorem.

Applying this method, for example, to the logical AND gate, $(a, b) \rightarrow ab$ becomes the classical reversible AND/NAND structure known as the Toffoli gate $(a, b, c) \rightarrow (a, b, c \oplus ab)$, which is described by the unitary matrix on three qubits:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The Toffoli gate applies the NOT on the last bit, conditioned that the other bits are 1, it is represented by the following diagram. Please note that the Toffoli gate and the reversible AND/NAND gate share the same matrix. The reversible AND/NAND gate was originally proposed by Toffoli [6], and was later adopted in quantum paradigm and named the Toffoli gate.

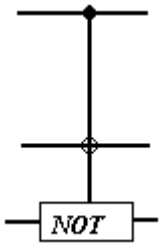


Figure 5.13: Symbolic representation of the Toffoli gate.

Quantum gates can perform more complicated tasks than simply computing classical functions. An example of such a quantum gate, which is not a classical gate applies a general rotation on one qubit:

$$\boldsymbol{G}_{\theta,\phi} = \begin{bmatrix} \cos(\theta) & \sin(\theta)e^{i\phi} \\ -\sin(\theta)e^{-i\phi} & \cos(\theta) \end{bmatrix}$$

During the discussion of linear operators corresponding to classical reversible functions in Section 5.5.2, it was pointed out that only a certain class of unitary operators represent classical reversible functions. This restriction and the implications that arise as a result of it become clear from the generic unitary operator represented above.

The class of unitary operators where every row and column has only a single non-zero element that takes on a value +1, +*i*, or –*i* can represent classical reversible functions. Generic examples of such linear operators are:

1) $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ where every row and column has only one non-zero element and these non-zero elements are represented by the real value 1.

2) $\begin{bmatrix} 0 & -i \\ +i & 0 \end{bmatrix}$, where every row and column has only one non-zero element and they are represented by the imaginary values +*i* and –*i*. Note that the condition that the inverse of a unitary operator equal its Hermitian complex conjugate requires that if one value is +*i*, then the other has to equal its complex conjugate –*i*.

3) $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & +i & 0 \end{bmatrix}$, where every row and column still has only one non-zero element, but these non-zero elements may be a mix of real value 1 or imaginary values $+i$ and $-i$.

The class of unitary operators that fall solely in the quantum category, are ones that lead to superpositions. These operators have no restriction of a single non-zero element per row or column imposed upon them. The removal of this restriction allows these operators to take qubits into superposition states. A perfect example of such an operator is the Hadamard gate discussed in Section 1.4, and is represented by $\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

As a result, it can be inferred that the set of all unitary operators that correspond to classical reversible functions, also fall in the class of unitary operators that represent quantum circuits, but not vice-versa. Hence the set of quantum circuits forms a superset of the set of classical reversible circuits.

5.9.3 Output Readout from Quantum Circuits

Performing a quantum computational task requires the application of a sequence of elementary quantum gates on qubits in the given system. Supposing that all quantum gates in an algorithm under study have been applied and the computation has come to an end. The input state which was initially a basis state has been rotated to a state $|\alpha\rangle \in C^{2^n}$. The final state $|\alpha\rangle$ after all unitary transformations representing the appropriate quantum circuits have been

performed upon it, now exists as a superposition of the computational basis states with some probability attached to each basis state. This state does not constitute an output in the classical sense since one cannot deterministically tell what the output of the computation is. The state has to be observed, or measured in order to determine the output. This may seem a very obvious statement in the context of classical computing, but not so in the quantum case. The information that constitutes the output is extracted from the circuit by means of a process generally referred to as the measurement. The entire ambiguity arising from the indeterminacy of the quantum state reaches a flashpoint at this specific stage. For example, consider a measurement of a qubit in the state $|\alpha\rangle = c_0|0\rangle + c_1|1\rangle$, where the qubit may be said to be neither in the state $|0\rangle$ nor in $|1\rangle$, or alternatively it may be construed that the qubit exists in both states at the same time. Yet, the measurement postulate studied in Section 3.4.5 asserts that when the state of this qubit is observed, it must decide on one of the two possibilities, a decision that is made non-deterministically. The classical outcome of the measurement would be **0** with probability $|c_0|^2$ and **1** with probability $|c_1|^2$. After the measurement, the state of the qubit is either $|0\rangle$ or $|1\rangle$, in consistency with the classical outcome of the measurement. But until the measurement procedure is conducted upon the qubit to study whether it is in state $|0\rangle$ or state $|1\rangle$ there is absolutely no means of knowing with certainty, what the current state of the qubit might be. The usual interpretation attached to state of the qubit just before and after the measurement proceeds on the following lines. Just before the measurement process is carried out the qubit is in a superposition of the two computational basis states that span the state space of the qubit. Upon conducting a measurement, the qubit is forced to collapse onto state $|0\rangle$ or state $|1\rangle$. Therefore, the probability associated with the qubit being in a particular basis state merely says how likely it

is for the qubit to collapse onto that specific computational basis state. Please note that the qubit may very well collapse onto a basis state with a lower probability. This possibility arises on account of the capability to determine how "probable" it is for the qubit to be in a given state and not a distinct "possibility" of being in it.

Geometrically, this process can be interpreted as a projection of the state on one of the two orthogonal subspaces, S_0 and S_1, where S_0 = span$\{|0\rangle\}$ and S_1 = span$\{|1\rangle\}$. The measurement of the state of the qubit $|\alpha\rangle$ is actually an observation in which of the subspaces the state is, inspite of the fact that the state might be in neither. The probability that the decision is S_0 is the norm squared of the projection of $|\alpha\rangle$ on S_0, and likewise for 1. Due to the fact that the norm of $|\alpha\rangle$ is one, these probabilities add up to one. After the measurement, $|\alpha\rangle$ is projected to the space S_0 if the answer is 0, and to the space S_1 if the answer is 1. This projection constitutes that so called collapse of the wave function or the qubit discussed above. Now what if a measurement is conducted upon a qubit in a system of n qubits? The state is once again projected onto one of two subspaces, S_0 and S_1 where S_a is the subspace spanned by all basis states in which the measured qubit is a. The rule is that if the measured superposition is $\sum_i c_i|i_1 i_2 ... i_n\rangle$, a measurement of the first qubit will give the outcome 0 with probability $\sum_{i_2,...i_n}\left|c_{0,i_2,...i_n}\right|^2$, and the superposition will collapse to

$$\frac{1}{\Pr ob(0)}\sum_{i_2,...,i_n} c_{0,i_2,...i_n}|0,i_2,...i_n\rangle,$$

and likewise with 1.

We can summarize the definition of the model of quantum circuits. A quantum circuit is a directed acyclic graph, where each node in the graph is associated a quantum gate [12]. This is exactly the definition of classical Boolean circuits, except that the gates are quantum. The input for the circuit is a basis state, which evolves in time according to the operation of the quantum gate. At the end of the computation we apply measurements on the output bits (The order does not matter). The string of classical outcome bits is the classical output of the quantum computation. This output is in general probabilistic. This concludes the definition of the model.

5.9.4 The Universal Quantum Computer

In 1985, David Deutsch at University of Oxford defined the notion of the universal quantum computer in his seminal paper on quantum computation titled Quantum Theory, the Church-Turing principle and the universal quantum computer [9]. In this paper Deutsch argued that, underlying the Church-Turing hypothesis, there is an implicit physical assertion and presented it as a physical principle.

The original conjecture posed by Alonzo Church and Alan Turing in 1936 was, "every 'function which would naturally be regarded as computable' can be computed by the universal Turing machine." Deutsch reinterpreted the statement 'functions which would naturally be regarded as computable' as those functions which may, in principle be computed by a real physical system. He came up with the physical version of the Church-Turing principle which states, "Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means." Further extending the argument, it was suggested that every general computational model was effectively classical in nature and that

classical physics and the classical universal Turing machine do not obey the Church-Turing principle, thus suggesting the motivation for seeking a truly quantum model of computation.

Therefore, by applying the theory of quantum mechanics, Deutsch set out to provide the theoretical description of the physical assertion of the Church-Turing principle, culminating in the definition of the universal quantum computer. The theoretical description of the universal quantum computer model is quite similar to the original Turing model, but the theoretical superstructure of this model is described within the principles of quantum mechanics in order to circumvent the earlier stated inadequacies posed by the application of classical physics.

A basic description of the universal quantum computer model or the universal quantum Turing machine is provided below. Deutsch's construction follows an almost quantum mechanical approach to this model, whereas the description adopted here is an analogous construction that may be considered more in line with the Turing specification. The need to do so was felt in order to provide a better insight into the relationship between classical and quantum computing. For a complete definition of the model, the reader is referred to the original paper by Deutsch [9].

A universal quantum computer model is specified by the following items:

a. A finite alphabet $\sum = \{Ц, 0, 1, \ldots\}$ where Ц represents the blank symbol.

b. A finite set $K = \{q_0, q_1, \ldots, q_s\}$ of machine states, with $h, s \in K$ are two special states.

c. A transition function $\delta: Q \times \sum \times Q \times \sum \times \{-1, 0, 1\} \rightarrow C$.

As in case of the classical Turing machine, the tape is associated with a head that reads from and writes to the tape. A classical configuration, c, of the Turing machine is specified by the head's position, the contents of the tape and the machine's state. The Hilbert space of this quantum Turing machine is defined as the vector space, spanned by all possible classical configurations $\{|c\rangle\}$. The dimensionality of this space is infinite. The computation commences with the universal quantum computer being in a basis state $|c\rangle$, which corresponds to the following classical configuration: An input of n symbols is written in positions 1,…, n on the tape, all symbols except these n symbols are blank (Ц) and the head is at position 1. Each time step, the machine evolves according to an infinite unitary matrix defined in the following manner. $\mathbf{U}_{c,c'}$, the probability amplitude to transform from configuration c to c' is determined by the transition function δ. If in c, the state of the machine is q and the symbol in the current place of the tape head is σ then δ(q, σ, q', σ', ε) is the probability amplitude to go from c to c', where c' is equal to c everywhere except locally. The machine state in c', q, is changed to q', the symbol under the head is changed to σ' and the tape head moves one step in the direction ε. Note that the operation of such a system is local, i.e. it depends only on the current state of the machine and the symbol now read by the tape. Unitarity of infinite matrices is not easy to check, and conditions for unitarity were given by Bernstein and Vazirani [13].

It is the opinion of many experts, and quite a justified one, that the quantum Turing machine is less appealing than the model of quantum circuits, for certain reasons. Firstly, the quantum Turing machines involve infinite unitary matrices. Secondly, it seems highly unlikely that a physical quantum computer will actually resemble such a model, owing to the fact that the

head, or the apparatus executing the quantum operations, is most likely to be classical in its position and state. Thirdly, the model specified above is a sequential model, implying that it applies only one operation per time step. Moreover, it is the general notion of experts in this field that quantum algorithms are better constructed in the circuit model than using the universal quantum computer model. This concludes the description of the universal quantum computer model.

The relationship between the reversible universal Turing machine and the universal quantum computer model becomes obvious when it is understood that any linear operator representing the transition function of the reversible universal Turing machine forms a part of the infinite unitary matrix that characterizes the universal quantum computer model, for the simple reason that the infinite unitary matrix would include within itself all functional characteristics of the unitary operator characterizing the reversible universal Turing machine. From Bennett's proof discussed in the context of the reversible universal Turing machine, it is known that the reversible universal Turing machine marks the reversible version of the standard irreversible universal Turing machine. Hence it logically follows that all functions considered computable within the framework of the universal Turing machine in the classical paradigm are computable on the universal quantum computer model as well.

5.10 Conclusion

This marks the completion of development of the uniform theoretical description, referred to now as the *linear representation*. As has been shown in the previous and current chapters, this description is capable of handling all three paradigms of computing while simultaneously

accounting for the unique attributes that characterize them. The transition from one paradigm to the other is attained in a seamless manner while preserving the unique attributes that are characteristic to each paradigm. Therefore, constraint 1 outlined in Section 2.2 is met. The transition is smooth when taking the higher levels of theoretical abstraction in the abstraction hierarchy, in all three paradigms, thereby meeting constraint 2 in Section 2.2. Constraint 3 in the same section required that constructs of the mathematical formalism employed (in this case, the theory of linear spaces) should not be violated. The fact that the entire development of the *linear representation* takes place within the framework of these constructs is proof that constraint 3 of Section 2.2 is met. The relationship between the linear operators and the physical principles underlying computational processes was unequivocally established with regard to all three paradigms in Sections 5.5.2 and 5.9.2, therefore criterion 4 is satisfied.

6.0 APPLICATIONS

6.1 Introduction

The *linear representation* was developed in Chapters 5 and 6 in the context of the classical, reversible, and quantum paradigms of computation. In order for any proposed theoretical representation that models a physical process to succeed, it is important that the model compactly explain that which is already understood about the physical process being modeled. This chapter presents a few existing applications in the domain of computation that exploit the linearity in the structure of logic.

There are at present many open problems in specific areas ranging from complexity theory, and finite automata, to formal verification techniques. Some open problems will be discussed. The proposed description shows initial promise to solve or provide adequate workarounds to some current problems. A short treatment of these possibilities is provided.

6.2 VLSI CAD: Binary Decision Diagrams

Many tasks in digital system design, combinatorial optimization, mathematical logic, and artificial intelligence can be formulated in terms of operations over small, finite domains. By introducing binary encoding of elements in these domains, these problems can be further reduced to operations over Boolean values. For instance, the set of truth values is often denoted by means of a binary representation of the form $\mathbf{B} = \{0, 1\}$. If an ordering of the variables of a

Boolean expression, say ***t***, were to be fixed, then ***t*** may be viewed as a function from $\mathbf{B}^{\mathbf{n}}$ to **B**, where **n** is the number of variables. Please note that the particular ordering of the variables is essential for what function is defined. Two Boolean expressions ***t*** and ***t'*** are said to be equal if they yield the same truth value for all truth assignments. The set of all truth value assignments for given set of variables, resulting in an output constitutes the truth table of the function. This may be represented by means of a directed acyclic graph.

It is possible to express a problem in a generic form through symbolic representation of Boolean functions. A binary decision diagram represents a Boolean function as a rooted, directed acyclic graph, with internal vertices of the graph corresponding to the Boolean variables over which the Boolean function has been defined, and terminal vertices constitute the function outputs. Such directed acyclic graph forms form the symbolic representations of Boolean functions. Binary decision diagrams are of paramount interest in the circuit design industry, where many electronic design automation tasks from circuit representation to formal verification and testing are performed through symbolic representation schemes.

The symbolic representation of Boolean functions through binary decision diagrams was first introduced by Lee [14] and Akers [15]. Interestingly, these representations were found to be canonical in nature. A sample binary decision diagram (BDD) is shown for the simple AND gate.

Table 6.1: Truth table for the two input AND gate.

x	***y***	***f(x,y)= x.y***
0	0	0
0	1	0
1	0	0
1	1	1

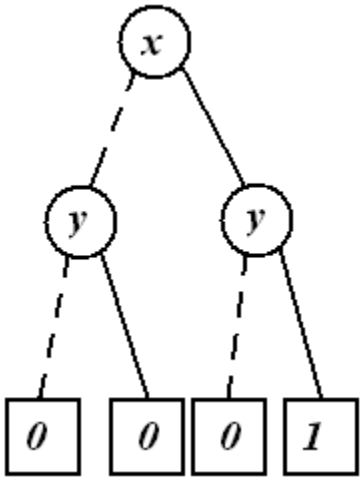


Figure 6.1: Binary Decision Diagram representing the AND gate.

In Figure 6.1, the dashed lines denote the binary value '0' of the Boolean variable they branch out from and the solid lines represent binary value '1' of the Boolean variable they branch from. The internal vertices represent the Boolean variables ***x*** and ***y*** of the two input AND gate in question. The terminal vertices constitute the function outputs. It is known that directed

acyclic graphs may be represented in terms of tree structures. Such binary trees representing Boolean functions are also referred to as “decision trees”, especially in complexity theory.

Comparing entries in Table 6.1 with Figure 6.1, it becomes obvious that each entry in the truth table representing the two input AND gate corresponds to a unique path traversed from the uppermost parent node to a leaf node.

First entry of the truth table leads an input combination ‘00’ to an output value ‘0’. This entry corresponds to the path traversed from the parent node representing variable ***x***, traversing down to the node representing variable ***y*** via the dashed line (representing value 0), down to the left most leaf node with value 0, via the dashed line.

The second entry corresponds to the path traversed from parent node down via the dashed line to variable ***y***, where the path changes to the solid line (since the value of variable ***y*** is 1), to end in the second leaf node from the left.

The third entry corresponds to the path traversed from the parent node down to variable ***y*** via solid line, down to the third leaf node with value 0, via the dashed line.

Similarly, the fourth entry corresponds to the path traversed from the parent node down the solid line to variable ***y***, down the solid line (since both variables have values 1), ending the leaf node representing value 1.

In the same manner, binary decision diagrams may be constructed for any given Boolean function comprising composites of elementary Boolean elements. In any given binary decision diagram, each unique path traversal from the parent to the leaf node constitutes a truth table entry. Please note that the binary decision diagrams, as represented by means of binary decision trees are not canonical forms. Other equivalent decision trees can be constructed for the same function through a different variable ordering mechanism.

The problem arising with this representation in the specific case of VLSI CAD applications is that the diagrams turn out to be as big as the truth tables themselves. When representing complex circuits with a large number of variables, the span of the decision tree becomes very difficult to manage. This difficulty arises primarily because the number of possibilities increases exponentially for every new variable added to a logic function (recall that the number of input combinations, and, therefore, the number of truth table entries is of the order of 2^n where n is the number of variables in the function). As a result, the number of paths increases exponentially as one traverses down the tree structure.

In order to introduce canonicity into the structures, and to counter the problem of exponentially increasing tree structures with increasing depth, Prof. Randal Bryant at CMU, proposed a variant to binary decision diagrams by exploiting the redundancy in the output possibilities in 1986 [16]. This variant is referred to as an Ordered Binary Decision Diagram or Reduced Ordered Binary Decision Diagram.

A Binary Decision Diagram is ordered, if on all paths through the graph the variables respect a given linear order $x_1 < x_2 < \ldots < x_n$. An Ordered binary decision diagram is a Reduced Ordered Binary Decision Diagram if:

- **Uniqueness:** No two distinct nodes *u* and *v* have the same variable name and low and high successor (where low successor represents the dashed line corresponding to value 0, and high successor represents the solid line corresponding to the value 1), i.e.

  $$var(u) = var(v),\ low(u) = low(v),\ high(u) = high(v) \text{ implies } u = v.$$

  and

- **Non-redundant tests:** No variable node *u* has identical low and high successor, i.e.

  $$low(u) \neq high(v).$$

ROBDDs have some interesting properties. They provide compact representations of Boolean functions, and there are efficient algorithms for performing all kinds of logical operations on ROBDDs. They are all based on the crucial fact that for any given arbitrary function $f: \mathbf{B}^n \rightarrow \mathbf{B}$ there is exactly one ROBDD representing it. This means, in particular, that there is exactly one ROBDD for the constant true (and constant false) function on $\mathbf{B}^n$: the terminal node 1 (and 0 in case of false). Hence, it is possible to test in constant time whether a ROBDD is constantly true or false. This problem was actually shown to fall in the complexity class of NP-complete[3] problems by Cook [17].

[3] For readers unfamiliar with the notion of NP-completeness the following short summary of the pragmatic consequences suffices. Problems that are NP-complete can be solved by algorithms that run in exponential time. No polynomial time algorithms are known to exist for any NP-complete problems and it is very unlikely that polynomial time algorithms should indeed exist although nobody has yet been able to prove their non-existence.

To generate a ROBDD from a generic BDD requires that the generic BDD be subjected to three transformation rules in a manner that does not alter the function in question. The three transformation rules are enumerated below.

- **Removal of duplicate terminals:** Eliminate all but one terminal vertex with a given label and redirect all arcs into the eliminated vertices to the remaining one.
- **Removal of duplicate Non-terminals:** If non-terminal vertices *u* and *v* have *var(u) = var(v), low(u) = low(v), high(u) = high(v)*, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.
- **Removal of redundant tests:** If non-terminal vertex *v* has *low(v) = high(v)*, then eliminate *v* and redirect all incoming arcs to *low(v)*.

The reduction of a BDD into a ROBDD is shown by means of an example. In fact, this is the example referred to by Prof. Bryant in his original paper. Consider a Boolean function characterized by the truth table provided below.

Table 6.2: Truth table for the sample function under study.

x	*y*	*z*	*f*(*x*, *y*, *z*)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

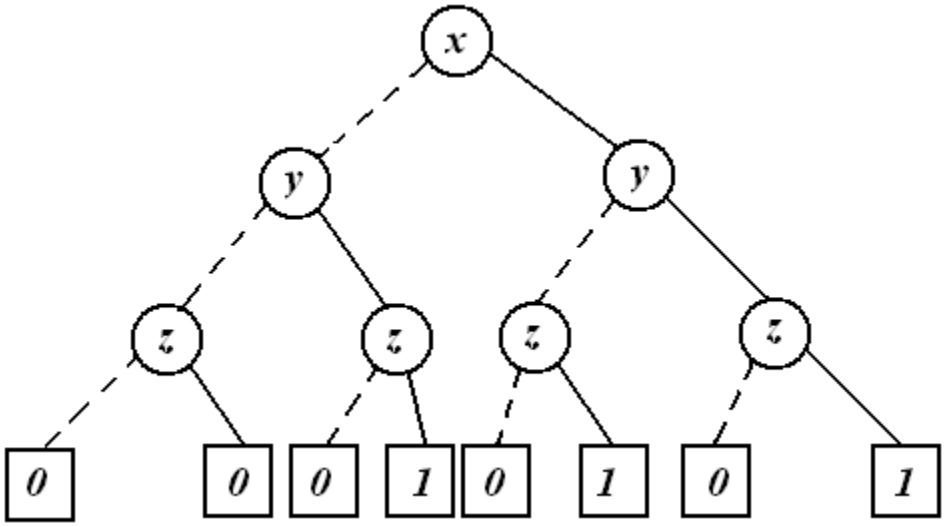


Figure 6.2: Binary Decision Diagram corresponding to truth table in Table 6.2.

Now, we apply the three transformation rules to realized the ROBDD for the BDD presented in Figure 6.2. The first transformation rule involves the removal of duplicate terminal nodes. This is achieved through removal of all duplicate terminal nodes, save one each with a unique label, and redirecting all arcs to the single instance of the terminal with relevant label.

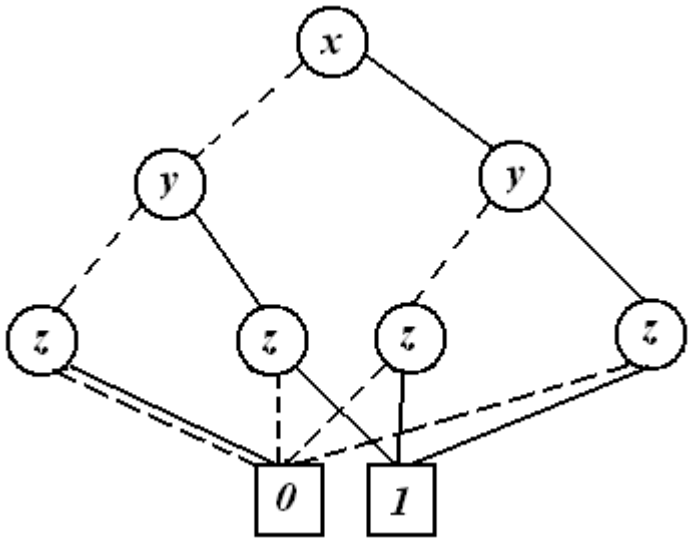


Figure 6.3: Binary Decision Diagram after application of first transformation rule.

The second transformation rule is now applied to the binary decision diagram resulting from application of the first transformation rule represented in Figure 6.3. The second transformation rule requires that if non-terminal vertices *u* and *v* have *var(u) = var(v), low(u) = low(v), and high(u) = high(v)*, then one of the two vertices is eliminated and all incoming arcs are redirected to the other vertex. The binary decision diagram that results from the application of the second transformation rule is shown below.

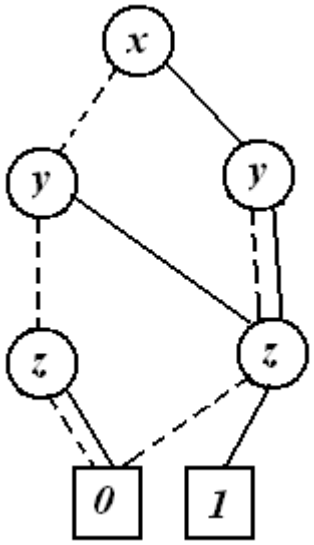


Figure 6.4: Binary Decision Diagram resulting after application of second transformation rule.

The third transformation rule is now applied to the binary decision diagram resulting after application of second transformation rule, as represented by Figure 6.4. The third transformation rule requires that if non-terminal vertex *v* has *low(v) = high(v)*, then *v* is eliminated and all incoming arcs are redirected to *low(v)*. This results in a ROBDD as shown in Figure 6.5.

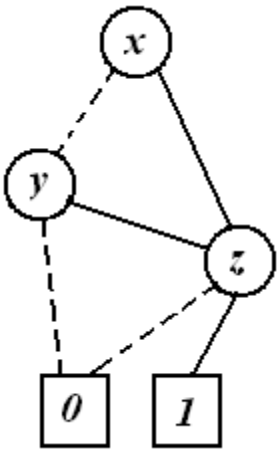


Figure 6.5: The ROBDD resulting after application of third transformation rule.

For a given ordering, two ROBDDs for a function are always isomorphic. As a result, the ROBDD function of a function is canonical. This property leads to several important consequences in the VLSI CAD industry, especially in automated circuit design and formal verification. Functional equivalence can be tested easily. A function is satisfiable if and only if its ROBDD representation does not correspond to the single terminal vertex labeled 0. Any tautological function must have the terminal vertex labeled 1 as its ROBDD representation. If a function is independent of variable $\boldsymbol{x}$, then its ROBDD representation cannot contain any vertices labeled by $\boldsymbol{x}$. Thus, once ROBDD representations of functions have been generated, many functional properties become easily testable.

Having understood the rationale behind the symbolic representation of circuits in terms of BDDs and their ordered reduction to canonical forms represented by ROBDDs, the question that arises is, what attributes of Boolean logic make these symbolic representations of circuits possible?

The answer to this question lies in the linear structure of Boolean functions. It is known that matrices and graphs are isomorphic representations. Given any arbitrary scheme (may or may not be related to computation), if it is possible to represent it in terms of a matrix, then it is also possible to represent the same scheme by a graph structure. Both representations provide the same information regarding the scheme. The reason for preferring one representation over the other being, sometimes the required information is easily extracted by using a matrix in place of a graph structure, and vice versa.

In Chapter 4, we developed the *linear representation* in the context of classical computation, where Boolean functions were studied extensively. Going back to the classical paradigm, let us take a closer look at the two input AND gate for instance. The linear operator corresponding to this elementary Boolean gate was derived to be $\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. Compare it with Figure 6.1 representing the BDD for the two input AND gate. We notice that both representations provide the same information, as does the truth table for the AND gate given in Table 6.1. If each path of the BDD traversed from the top most vertex to the terminal vertex coincides with an entry of the truth table, the matrix provides the same information from the values taken by the elements at the intersection of the respective columns and rows. Applying the three transformation rules on BDD corresponding to this gate provided in Figure 6.1, leads to the ROBDD as represented in Figure 6.6.

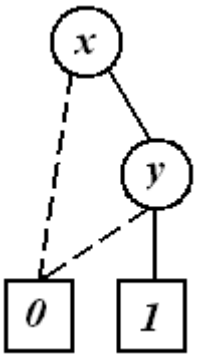


Figure 6.6: The ROBDD corresponding to the two input AND gate.

Extending the analysis of BDD structures to the paradigm of reversible computing it is observed that the BDD structures that result for classical reversible functions are by default optimal. An ordering and reduction scheme to realize ROBDDs from BDDs representing classical reversible functions is not possible, as a result of the restrictions arising from imposition of invertibility criteria upon these circuit structures. The one-to-one mapping conditions give rise to unique trajectories, hence, no redundant paths arise. Second, the onto mapping condition requires that all possible output combinations be realized, leading to 2^n terminal vertices, where n is the number of input/output lines to the circuit, or alternatively the depth of the graph itself. Moreover, the BDD structures are not canonical. The reason being, given a reversible function, the ordering of the output lines does not alter the function itself, but it alters the graph, and correspondingly the linear operator that represents the reversible function. Therefore, depending upon the ordering of the outputs, multiple variants of graphs (and therefore, linear operators) can be realized that represent the same functional characteristics.

In the quantum paradigm, it is not possible to represent unitary operators of the superposition class through BDDs. The capability of the qubit to attain superposition states invalidates the possibility of representing quantum circuits using BDDs. The problem arises primarily because the probability amplitudes (complex coefficients in the linear combinations) of the intermediate states (the qubit states during computation) depend upon the specific input provided to the circuit, whereas the actual sequence of gates that is applied is independent of the input. There seems to be no good input-independent method to describe a circuit in a way where the states may be represented by vertices of the graph. Consider, a single qubit $|\psi\rangle$, acted upon by a Hadamard gate, **H** say. The transformation gives rise to following three possibilities:

1. If $|\psi\rangle = |0\rangle$, initially, applying the Hadamard transform upon this qubit results in:

   $$\mathbf{H}|\psi\rangle = \mathbf{H}|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

2. If $|\psi\rangle = |1\rangle$, initially, applying the Hadamard transform upon this qubit results in:

   $$\mathbf{H}|\psi\rangle = \mathbf{H}|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

3. If $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, initially, applying the Hadamard transform upon this qubit results in:

   $$\mathbf{H}|\psi\rangle = \mathbf{H}(\alpha|0\rangle + \beta|1\rangle) = \frac{1}{\sqrt{2}}[\alpha(|0\rangle + |1\rangle) + \beta(|0\rangle - |1\rangle)] =$$

   $$\frac{1}{\sqrt{2}}[(\alpha + \beta)|0\rangle + (\alpha - \beta)|1\rangle].$$

There is currently no known method to model this operation in terms of a directed acyclic graph such that the qubit states form vertices of the graph as in the classical case.

However, an alternate scheme could be applied that involves representing the linear operators by vertices instead of the qubits (which are analogous to the Boolean variables). This alternate scheme is applied when studying decision tree complexity of quantum circuits, in the context of quantum complexity theory [12]. Consider, yet again, the case of the single qubit $|\psi\rangle$, being acted upon by a Hadamard gate. The directed acyclic graph corresponding to the Hadamard structure, as represented in the alternate scheme is shown below.

Figure 6.7: Alternate graph representation of the Hadamard gate.

In more complex cases, where multiple unitary operators are involved, the order in which the respective unitary operators are applied occurs in top to bottom cascaded approach. In here, we have adopted a method to distinguish the vertices denoting qubits from those denoting the unitary gates. The initial vertex (i.e. parent node) and the final vertex (leaf node) represent the input qubit state and the resulting output qubit state after measurement respectively. The gates are represented by square vertices. The traversal from top to bottom of the graph, from the initial qubit state to the final state represents the unitary time evolution of the quantum mechanical system itself.

This section outlined a very crucial application in computation that exploits the linearity of Boolean functions in applications related to electronic design automation. The primary objective behind the treatment of an existing well developed application was to validate the consistency of the *linear representation* by reconciling it to that which is currently known, and applied within the domain of computation, thereby satisfying constraint 5 in Section 2.2. In the following

section, a possible extension of this application through the *linear representation* is presented, which could possibly provide improved spatial savings for such representations.

6.3 Possible Applications of the *Linear Representation*

In this section we propose a few possible applications in the domain of computation that extend the current subfields or may aid in determining a solution to open problems.

6.3.1 VLSI CAD: Binary Decision Diagrams

A short treatment of Binary Decision Diagrams was provided in Section 6.2 with the objective of meeting constraint 5 of Section 2.2. We now propose a possible application of the *linear representation* to gain spatial and/or temporal savings in the representation of BDDs in software. There are currently, many software packages directed towards VLSI CAD applications that apply BDDs heavily to resolve net-lists in the construction of circuits, as well in formal verification.

Linked lists generally form the natural choice when representing graph structures in software. The primary disadvantages faced when applying linked list form of data structures in software are the incremental memory requirements, the extra set of program subroutines for creation, deletion, traversal, and updating of linked list structures.

The best known BDD package available in the industry today is CUDD (CU Decision Diagram) package developed at the Department of Electrical and Computer Engineering,

University of Colorado at Boulder [18]. This package represents BDD nodes in the form of a class for a generic decision diagram node, and a derived class for the various types of decision diagram nodes, such as BDD, ADD, ZDD etc. Every BDD node in this package usually requires four words (one word each for the variable index, the if-then-else children, and next pointer). Assuming a four byte word (for 32 bit machines), the size of each BDD node works out to be 32 bytes.

Considering the case of the two input AND gate, its BDD structure holds three variable nodes and four leaf nodes, hence, a total of seven nodes. Therefore, the memory required to represent a simple two input AND gate is 224 bytes. The ROBDD corresponding to the two input AND gate contains three variable nodes and two leaf nodes, a total of five nodes. Therefore, the total memory required to for the canonical symbolic representation of the two input AND gate is usually 160 bytes.

On the other hand matrix structures can be represented through an easier data structure: indexed arrays. A very basic analysis of the spatial requirements for circuit representations using matrices shows promise of cutting down spatial costs. Now consider the linear operator corresponding to the two input AND gate, it is a 2 x 4 matrix of 8 elements. It may be represented as a two dimensional bit array (an array comprising of 1 bit per array position) since the matrix takes only 1s and 0s as its elements. Therefore, the memory required to represent the AND gate as a matrix requires only 8 bits of memory, or 1 byte.

However, owing to the fact that this does not form the primary focus of this thesis effort, no formal analysis has been performed to this effect. Therefore, we are only in a position to suggest a possibility.

6.3.2 Quantum Finite Automata

The development of the *linear representation* through Chapters 4 and 5 has established the clear relationship shared between the theory of Finite Automata and the theory of linear spaces. Of course, the relationship has been long known, an ample validation of this fact being the application of group theory to both domains.

The theory of finite automata has traditionally dealt with mathematical computational models in the classical paradigm. Extension of this theory to include quantum mechanical systems has been extremely limited. The earliest known documented work in this direction we have come across is that of P. A. Benioff [10], where he proposed a model for computation within quantum kinematics and dynamics, but it is still effectively classical in most respects. During the last decade, two different models of quantum finite automata have been proposed. The first model, introduced by Moore and Crutchfield [19], makes one measurement at the end of its computation and is called a measure-once quantum finite automata, which is a semi-classical model in the sense that it uses a classical head that reads the string once. The second model, introduced by Kondacs and Watrous [20], makes a measurement of its state after every transition and is called a measure-many quantum finite automata. The crucial difference between the two models is that measure many – QFA makes a measurement of state after each symbol is read, allowing it to decide whether to accept, reject or continue, while a measure once

– QFA makes no measurements until the end of the computation. The ability to accept or reject a string without having to read all of it makes the measure many – QFA model significantly more powerful. It has also been shown that any reversible finite automata can be simulated by a measure many – QFA with certainty [21].

Current results in this domain are very elementary in nature. The difficulty in extending the theory of finite automata to the quantum regime arises primarily due to the highly non-classical properties that quantum mechanics exhibits. We are already well aware of the qubit's capability to exhibit superpositions, but there are other purely non-classical features in quantum mechanics that can be exploited in computation, quantum entanglement being one such example. The finite automata representation of non-local correlations shared between two qubits is a knotty problem. Nevertheless, it is felt that understanding the correspondence shared by the *linear representation* with finite automata techniques in the classical and reversible paradigms can aid in extension of the theory of finite automata to the quantum regime.

6.3.3 Formal Verification of Hardware Designs

Formal verification of hardware focuses primarily on establishing the functional correctness of a proposed hardware design. Two major aspects of formal verification are:

1. **Implementation verification:** For all feasible inputs, ensure that the behavior of the circuit is consistent with that required by the specification.
2. **Design verification:** For all feasible inputs, determine that the design has a number of properties required by the specification

The greatest benefit of formal verification techniques is in uncovering bugs, not necessarily in proving the design correct.

There are at present many approaches adopted towards formal verification of designs.

1. **Theorem proving:** Relationship between a specification and an implementation is regarded as a theorem in logic, to be proved within the framework of a proof calculus. This technique is primarily used for verification of arithmetic circuits in the hardware design industry.
2. **Model checking:** The specification is in the form of a logic formula, the truth of which is determined with respect to a semantic model provided by an implementation
3. **Equivalence checking:** The equivalence of a specification and an implementation is checked, i.e. equivalence of functions, finite state machines etc. This is the most commonly used technique of formal verification in the hardware industry.
4. **Language containment:** The language representing an implementation is shown to be contained in the language representing a specification. Here, the emphasis is upon formally proving that **L(imp)** $\subseteq$ **L(spec)**.
5. **Symbolic trajectory evaluation:** The properties are specified as assertions about circuit state in the form of pre and post conditions and verified using symbolic simulation. This technique of verification is used in verifying embedded memory arrays and transistor level designs in industry.

Application to Equivalence Checking

Equivalence checking is the most common technique of formal verification in industry today. Typically, the gate level implementation is compared with the representation at a higher level (RTL). A canonical representation allows easy comparison of two given functions. This canonical representation is provided through the ROBDD representation studied in Section 6.2. The limitation of this technique is that some functions like multipliers require exponential space for their representation.

We had proposed the representation of circuits using the canonical linear operators using indexed arrays in place of ROBDDs using linked list structures in Section 6.3.1. If possible, then the application of matrix structures as proposed in Section 6.3.1 might lead to significant spatial savings in circuit representations of such nature in the technique of equivalence checking.

Application to Check Illegal Output Possibilities

We propose a possible application where reversible logic functions may possibly be applied to perform model checking of a given hardware design in the classical paradigm.

1. Consider any arbitrary classical combinational or sequential circuit that is being designed.
2. We have shown through the *linear representation* that corresponding to any given classical function, there exists a corresponding reversible function.
3. Replace every atomic logic unit in the developed design with its reversible version.
4. Running the model checking tests backwards on the reversible circuit by providing the outputs, should lead to the original expected inputs of the circuit.

5. Suppose, there are certain illegal outputs (outputs that should never occur) in a given circuit.
6. Feed in the illegal output into the ancilla lines of the input stage that correspond to the function output line, along with all possible legal or valid input combinations to see what outputs result.
7. The original input lines at the output stage would obviously provide the legal input values that were fed in.
8. Our interest would be on the function output lines that should now hold a constant value representing the ancillas.
9. Remember that the ancilla lines always hold constant values. Therefore, if at any point in the testing process, the ancilla lines show a transition of values, it implies that an input combination has been entered, that results in the illegal output (whose possibility of occurrence) is being tested.
10. Immediately reverting back to the original irreversible design, one can test if the particular input combination indeed does lead to an illegal output state that is not desired.

In this manner, testing of hardware designs for illegal outputs could be possible.

Let us consider a simple example test case, that of the two input AND gate. The test gate takes in two input variables ***x*** and ***y***, and provides a function output $f(x,y) = x.y$.

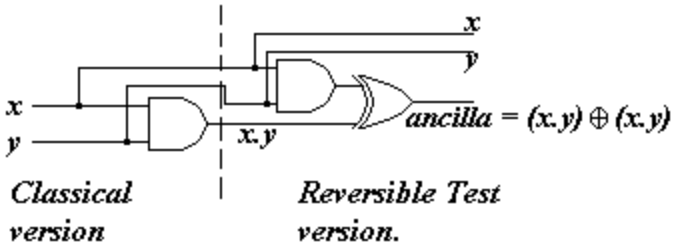


Figure 6.8: Representation of a test bed, first stage being the classical AND gate, and second stage being the reversible AND gate.

In the example above is a classical two input AND gate, that feeds its outputs into the ancilla line of the reversible AND gate, the two original input variables ***x*** and ***y*** are also fed into the reversible AND gate. The output of the reversible AND gate is the two original inputs, and the ancilla bit.

We now apply the procedure detailed above to check the possibility of testing circuits for illegal outputs through ancilla transitions.

1. Feed the input combination '00' (***x*** = 0, ***y*** = 0) to the classical AND gate.
2. The expected function output is '0' (***f*(*x*,*y*) = *x*.*y*** = 0) for the classical stage.
3. At the start of the reversible stage, the inputs are '000', corresponding to values of the two input variables and the function output from classical stage (***x*** = 0, ***y*** = 0, ***f*(*x*,*y*)** = 0) .
4. The output obtained from the reversible version is value '000'after performing the XOR operation (***x*** = 0, ***y*** = 0, ancilla = ***x*.*y*** $\oplus$ ***x*.*y*** = 0).
5. Assume that we received an illegal output 1 at the end of the classical stage.
6. The inputs to the reversible stage then are '001' (***x*** = 0, ***y*** = 0, ***f*(*x*,*y*)** = 1).
7. The function output at the end of the reversible stage now reads, '001' (***x*** = 0, ***y*** = 0, ancilla = ***x*.*y*** $\oplus$ ***x*.*y*** = 1).

8. It is known that the constant value of the ancilla line for a the reversible two input AND gate is always ‘0’. If the ancilla makes a transition to value ‘1’, then the output obtained at the end of the classical stage was definitely an illegal output.

The example discussed was very trivial in nature. As a result, the true capabilities of such a testing scheme might not be immediately apparent. We believe, such testing schemes could prove invaluable for testing illegal outputs for complex circuits. Such a testing scheme could be run through a simulation procedure also. It is not necessary to construct a physical realization of the reversible circuit in order to test the validity of its classical version.

6.4 Conclusion

This brings to close a short discussion on how the linearity in logic elements makes it possible for symbolic representation of circuits in VLSI CAD, thereby validating constraint 5 specified in Section 2.2. We have also pointed out some possible applications. Please note that the treatment of this thesis covers a broad spectrum of issues, simply put it encompasses the entire domain of computation. Therefore, the applicability of the *linear representation* is possible to the entire domain of computation in general. It would therefore, be nearly impossible to look into all possible applications of the proposed representation. The core effort of this thesis detailed in Chapters 5 and 6 has been to understand the theoretical structure of computational processes at a level that comes closest to their physical realization. To this end, it may be construed more as an interesting exercise in mathematical formulation and interpretation.

7.0 SUMMARY AND CONCLUSION

7.1 Summary

The development of the *linear representation* commenced with Chapter 4, where the structure of the classical bit was re-interpreted in terms of a two dimensional complex Euclidean space. The elementary Boolean gates were shown to constitute linear operators in this representation. The composition rules for construction of composite Boolean functions from elementary logic elements were defined. The same was extended to the case of classical sequential circuits, thereby successfully representing the paradigm of classical computation in the *linear representation*.

Chapter 5 extended this representation to the reversible paradigm. The invertibility criteria required in order for a Boolean function to include invertible primitives were defined. With the aid of these invertibility criteria, reversible elementary Boolean gate structures were developed. Some characteristic properties of the linear operators representing these gate structures were analyzed, in the process of which, Toffoli's Fundamental Theorem of Reversible Functions was independently validated in the *linear representation*. The unique correspondence between the eigenvectors of the linear operators and the truth table entries of the logic gates corresponding to the operators was noted. It was also shown that such a correspondence arises as a result of the underlying physical principles that define the dynamics of a computational process. The translation of the composition rules for construction of linear operators for reversible combinational functions was shown to follow automatically from the rules developed in the

context of the classical paradigm. The construction of reversible sequential circuits was shown to be feasible by isolating the memory element in the feedback loop of the circuit. The construction of the reversible universal Turing machine as developed by Charles Bennett, was explained, with an independent validation in the *linear representation*.

The second half of Chapter 5 dealt with the paradigm of quantum computation. The capability of the qubit to attain superposition states required an extension of the structure of the complex Euclidean space of the classical bit in order to include the superposition principle. This was achieved by making it a Hilbert space. The standard representation of quantum circuits in terms of unitary operators was reconciled with the *linear representation*. In the process, the relationship between the reversible and quantum paradigms of computing was established. The generic relationship between the two paradigms is explained by showing that the reversible universal Turing machine forms a special case of the universal quantum computer model.

During the definition of the problem in Section 2.2, certain constraints were defined. The satisfaction of these constraints was said to be imperative for any proposed theoretical model to fit in at the lowest level of the abstraction hierarchy. We will now see if these constraints are indeed met.

Constraint 1: In order for the proposed theoretical description to be considered uniform across the three paradigms of computing, it should make a seamless transition from one paradigm to the other, and at the same time account for unique attributes that characterize each paradigm.

Validation: The proposed *linear representation* has been shown to successfully explain computational models in all three paradigms of computing during the development of this representation in Chapters 4 and 5.

Constraint 2: It should make a similar seamless transition to the higher level of theoretical abstraction immediately above in the abstraction hierarchy as shown in the tabular analysis above.

Validation: The linear representation was shown to account for computational models that are traditionally described using finite automata techniques. The consistency of the definition of the Turing model in the *linear representation* has been shown to be logically consistent.

Constraint 3: The constructs of any mathematical formalism applied to develop the description must not be violated.

Validation: Throughout the development of the *linear representation*, the mathematical constructs of the theory of linear spaces are never violated. All constructs developed in the *linear representation* strictly adhere to the defining principles of the theory of linear spaces.

Constraint 4: The proposed theoretical description lies at the lowest level in the abstraction hierarchy. The physical principles dictating the computational process would have a strong bearing upon the description. Therefore, the physical principles underlying the computational process must be accounted for by the theoretical description.

Validation: The close correspondence shared by the *linear representation* with the underlying physical principles that define the computational process was established in Section 5.5.2 where the linear operators were shown to constitute the time evolution operators that define the dynamics of the system, thereby relating them to the Hamiltonians of the systems in question. A second correspondence was shown during the discussion on the dynamics of quantum circuits and the unitary time evolution of the circuits in accordance with the Schrödinger equation in Section 5.9.2.

Constraint 5: The proposed description must successfully explain that which is currently understood about, and applied in computation in order to prove its validity and consistency. If possible, it should also make experimentally verifiable predictions that have hitherto not been accounted for.

Validation: The validation of this constraint is provided in Chapter 6, where it was shown that the symbolic representation of circuits as binary decision diagrams was shown to arise as a result of the linear properties of the circuit structures. Some possible applications and extensions of current domains were also proposed.

The main contribution of this thesis has been to rationalize the structure of logic and computational models across all paradigms through a uniform theoretical description. Owing to the fact that Boolean algebra cannot represent the class of unitary operators that lead to bit superpositions, the *linear representation* was developed by extending the mathematical

formalism as applied in the paradigm of quantum computing, back in the classical and reversible paradigms. The logical consistency of this representation has already been shown in the process of developing this representation. The secondary contributions of this effort have been to suggest possible applications of this representation to formal verification techniques in hardware design, where reversible circuits may be employed to check for illegal output possibilities in classical circuit structures. The extension of the theory of finite automata to the quantum regime, holds potential of revealing new insights and hitherto unknown possibilities in computation. The reconciliation of the *linear representation* with finite automata techniques in the classical and reversible paradigms could prove beneficial in extending these techniques to the quantum regime.

7.2 Conclusion and Future work

This thesis has been an effort to learn and to understand computation as a domain in its entirety, with special emphasis upon the physical principles that define computational processes. The paradigm of quantum computing, and its special relationship with reversible computing is quite well understood by experts, but most of the literature explains these concepts from a physics point of view. It has been the author's personal experience that such a treatment leads to difficulties in understanding, and more importantly realizing the beauty of structures in these paradigms of computing and their symmetry. To this extent, we hope this thesis helps discerning readers would find the treatment in this document easy to follow.

The quest to understand the domain of computation and computational processes is still an unfinished agenda, with many an unsolved problem to be looked into. The focus in the near

future is two fold, first, to pursue work in the theoretical aspects of quantum computation from the physics point of view, and, second, to continue the work from the computational standpoint in areas related to VLSI CAD and formal verification, and if possible, to also work towards extending the theory of finite automata to the quantum regime. The possible applications of the *linear representation* outlined in Section 6.3 are very basic in nature. Each one of the proposed applications holds promise of being taken up as an independent research topic, and to help extend the specific sub-domains of computation that they deal with.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Boole G., "An investigation of the laws of thought, on which are founded the Mathematical Theories of Logic and Probabilities", 1854.

[2] Hilbert D, Ackerman W., "Grundzuege der Theoretischen Logik", 1928.

[3] Turing A. M., "On Computable Numbers, with an Application to the Entscheidungsproblem", Proc. London Math. Soc., ser. 2, 43 (1936), 544 – 546.

[4] Landauer R., "Irreversibility and heat generation in the computing process", IBM J. Res. Dev, 5 (1961), 247 – 253.

[5] Bennett C. H., "Logical reversibility of computation", IBM J. Res. Dev, 6 (1973), 525 – 532.

[6] Toffoli T., "Reversible Computing", 1980, MIT/LCS/TM-151.

[7] Nielsen M. A., Chuang I. L., "Quantum Computation and Quantum Information", 2000, Cambridge Press.

[8] Feynman R. P., 1982, Int. J. Theor. Phys, 21, 467.

[9] Deutsch D., "Quantum theory, the Church-Turing principle and the Universal Quantum Computer", Proc. Roy. Soc., 400 (1985), 97 – 117.

[10] Benioff P. A., 1982, Int. J. Theor. Phys, 21, 177.

[11] Horn R., Johnson C. R., "Matrix Analysis", Cambridge University Press.

[12] Buhrman H., de Wolf R., "Complexity Measures & Decision Tree Complexity: A Survey", 2001.

[13] Bernstein E., Vazirani U., "Quantum Complexity Theory", SIAM Journal of Computation, 265, pp 1411 – 1473, Oct 1997.

[14] Lee C. Y., "Representation of switching circuits by binary decision programs", 1959, Bell System Technical Journal 38, pp 985 – 999.

[15] Akers S. B., "Binary Decision Diagrams", IEEE Transactions on Computers, Aug 1978, C-27, pp 509 – 516.

[16] Bryant R. E., "Graph-based algorithms for Boolean function manipulation", IEEE Transactions on Computers, C-35, Aug 1986, pp 677 – 691.

[17] Cook S. A., "The complexity of theorem-proving procedures", Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing, pp 151 – 158, 1971, ACM.

[18] http://vlsi.colorado.edu/~fabio/CUDD

[19] Moore C., Crutchfield J. C., "Quantum Automata and quantum grammars", Theoretical Computer Science, 1998.

[20] Kondacs A., Watrous J., "On the power of quantum finite state automata", Proceedings of 38th Annual Symposium on Foundations of Computer Science, pp. 66 – 75, 1997.

[21] Ambainis A., Frievalds R., "1 – way quantum finite automata: Strengths, weaknesses, and generalizations", Proceedings of 39th Annual Symposium on Foundations of Computer Science, pp 332 – 342, Nov 1998.