Introduction to Quantum Computation

Ioan Burda

Introduction to Quantum Computation

Universal Publishers
Boca Raton, Florida • USA
2005

ISBN: 1-58112- 466-X

www.universal-publishers.com

For my parents and wife

Contents

1 Introduction

The theory of computation has been long considered a completely theoretical field, detached from physics. One of the people most directly responsible for this concept of computing machines is Alan Turing (1912-1954). Turing and others early giants proposed mathematical models for computing which allowed for the study of algorithms and in absence of any particular computer hardware. This abstraction has proved invaluable in the field of computer science. Turing's model (Turing 1936) is called a Turing machine.

When examining what sort of problems can be solved by a computer, one need only to examine a Turing machine, and not one of the millions of potential computing devices to determine if a computation is possible. If a Turing machine can perform the computation, then it is computable. If a Turing machine can not, then the function can not be computed by any classical computer. Church's thesis (Church 1937) states:

Any physical computing device can be simulated by a Turing machine in a number of steps polynomial in the resources used by the computing device.

This thesis gives us insight into the *power* of computing machines. If a computer theoretically can solve all the problems a Turing machine can solve (given enough memory and time) then it is as powerful as a Turing machine.

An algorithm can be characterized by the number of operations and amount of memory it requires to compute an answer given an input of size n. These characterizations of the algorithm determine what is called the *algorithms complexity*. Specifically, the complexity of an algorithm is determined by the number of operations and memory usage required to complete the program scales with the size of the input of the program. The size of the input is conveniently defined here to be the number of bits needed to represent that number in binary. Computer Scientists have grouped problems into *complexity classes*, below are some of the more well know.

- *P: Polynomial time, the running time of the given algorithm is in the worst case some polynomial in the size of the input.*

- *NP: Nondeterministic polynomial time, a candidate for an answer can be verified as a correct answer or not in polynomial time.*

- *NP-complete: A set of problems for which if any can be solved in polynomial time, P equals NP.*

Problems, which can be solved in polynomial time or less, are generally deemed to be *tractable*. Problems, which require more than polynomial time are usually considered to be *intractable*, for example an algorithm which would take 2^n operations for an input size of n would be considered intractable, as the number of operations grows exponentially with the input size, not polynomial. In general, if an algorithm requires polynomial time or less to compute it is considered to be *tractable*, and if not it is considered to be *intractable*. Great attention was paid by Turing and others to attempt to make their models of computing mathematical abstractions, harboring no hidden assumptions about the computing machinery itself. For many years it appeared that these models were indeed not based in any assumptions as to the nature of the computer, but this is not so.

The number of atoms needed to represent a bit in memory has been decreasing exponentially since 1950. Likewise, the numbers of transistors per chip, clock speed, and energy dissipated per logical operation have all followed their own improving exponential trends. Despite these fantastic advances, the manner in which all computers function is essentially identical. At the current rate in the year 2020, one bit of information will requite only one atom to represent it. The problem is that at that level of miniaturization the behavior of the components of a computer will become dominated by the principles of quantum physics.

With the size of components in classical computers, shrinking to where the behavior of the components may soon be dominated

more by quantum physics than classical physics researchers have begun investigating the potential of these quantum behaviors for computation. When components are able to function in quantum, unexpectedly it seems the possibility to obtain a more powerful computer than any classical computer can be.

A *quantum computer* (if one could be constructed) is more powerful than Turing machines, in the sense that there are problems, which are solvable on a quantum computer, which are not solvable on a Turing machine. The quantum computer is more powerful than the Turing machine and hence any classical computer because it can do things, which were thought to be physically impossible when Church's thesis was written. Church's thesis is still valid for any computing machine, which functions in a purely classical manner.

The laws of quantum mechanics could support new types of algorithms (*quantum algorithms*) Deutsch was able to define new complexity classes (Deutsch 1985) and establish a new hierarchy. However, it was not recognized until recently that the class of problems that can be solved in polynomial time with a quantum algorithm.

- *QP: A set of problems for which the best classical algorithm runs exponentially*.

In other words, quantum computers are able to perform (in polynomial time) certain computations for which no polynomial time algorithm is known.

2 Digital Systems

Before we can appreciate the meaning and implications of *digital systems*, it is necessary to look at the nature of *analog systems*. The world is *analog* is derived from the same root as the noun *analogy* and means a quantity that is related to, or corresponds to, another quantity. In a digital system, all variables and constants must take a value chosen from a set of values called an *alphabet*. In decimal arithmetic, we have an alphabet composed of the symbols 0 through 9. Other digital systems are Morse, Braille, semaphore and the day of the week.

An advantage of digital representation of information is that a symbol may be distorted, but as long as the level of distortion is not sufficient for one symbol to be taken for another, the original symbol may be recognized and reconstituted.

The alphabet selected for digital computers has two symbols, 0 and 1. The advantage of such an alphabet is that the symbols can be made unlike each other as possible to give the maximum discrimination between the two values. A single binary digit is known as a *bit* (*bi*nary digi*t*) and is the smallest unit of information possible.

2.1 Number Systems

The number system we use every day is a *positional number system*. In such a system, any number is represented by a string of digits in which the position of each digit has an associated weight. The value of a given number, then, is equivalent to the weighted sum of all its digits. A number written $a_{n-1}...a_1a_0.a_{-1}a_{-2}...a_{-m}$ when expressed in positional notation in the *base d* is defined as

$$a_{n-1}d^{n-1} + ... + a_1d^1 + a_0d^0 + a_{-1}d^{-1} + ... + a_{-m}d^{-m} = \sum_{i=-m}^{n-1} a_i d_i$$

where there are *n* digits to the left of the point, known as the *radix point*, and *m* digits to its right. Within a positional number system, the representation of every number is unique, except for possible leading and trailing zeros. Note that the leftmost digit in such a number system is called the most-significant digit (MSD), and the rightmost is the least-significant digit (LSD).

Since digital system use binary digits, we use a binary base to represent any number in a digital system. The general form of such a binary number is $b_{n-1}...b_1b_0.b_{-1}b_{-2}...b_{-m}$ and its value is equivalent to

$$B = \sum_{i=-m}^{n-1} b_i(2)^i$$

Similarly, to a decimal point in a decimal number, the radix point in a binary number is called the *binary point*.

2.2 Hexadecimal Numbers

The hexadecimal number system uses base 16. The hexadecimal system needs to express 16 different values, so it supplements the decimal digits 0 through 9 with the letters A through F. From our discussion of positional number systems, we can easily recognize the importance of base 10, which use in everyday life, and base 2, which is used by digital systems to process numbers.

The general form of such hexadecimal numbers is $h_{n-1}...h_1h_0.h_{-1}h_{-2}...h_{-m}$ and its value is equivalent to

$$H = \sum_{i=-m}^{n-1} b_i(16)^i$$

The binary integers 0 through 1111 and their binary-coded decimal and hexadecimal equivalents are given below

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

Hexadecimal numbers are often used to provide convenient shorthand representations for binary numbers, reducing the need for lengthy, indecipherable strings. The hexadecimal number system is quite popular, because it can easily be converted to and from binary, and because standard byte, word, double-word data can be documented efficiently by hexadecimal digits.

As a rule we cannot covert a number representation in one base into representation in another base simply by substituting numbers in one base from their equivalent in another base representation; this works only when both bases are powers of the same number. When this is not the case, we must use more complex conversion procedures that require arithmetic operations.

2.3 Coding Theory

In the world of digital systems, there are many different codes, each one the best suited to the particular job for which it was designed. A particularly widespread binary code is called BCD or

binary-coded decimal. BCD numbers accept the inevitability of two-state representation by coding the individual decimal digits groups of four bits as given below

DECIMAL	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The BCD is often called pure binary, natural binary, or 8421 weighted binary. The 8, 4, 2, and 1 represent the *weightings* of each of the columns in the positional code. Many other positional codes do not have a natural binary weighting. Some codes are called *unweighted* because the value of a bit does not depend on its *position* in a number. Each of these codes has special properties that make it suitable for a specific application.

Within a digital system, an error can be caused only by temporary or permanent physical failures and can be defined as the difference between transmitted and received data. To detect such errors, we need to secure the data with the help of error-detecting code. The goal of *coding theory* is to represent digital information in a form that allows for the recovery of the original data from its corrupted form, if the number of errors is not too large. This requires that some redundancy be incorporated into the stored information.

Information is stored and transmitted as a stream of *letters* from a chosen *alphabet A*. Most popular is the *binary alphabet* $A = \{0,1\}$. More generally, $A = \{0,1,2,...,p-1\}$ with addition and multiplication mod p (where p is a prime number) is popular because A is a field. In this case

$$A^n = \{(a_1, a_2, ..., a_n) : a_i \in A\}$$

is an *n*-dimensional vector space over *A*. A *word* of length *n* is a string of *n* characters from the alphabet *A*. If $A = |q|$ then there are q^n words of length *n*. These are identified with the vectors of A^n. A *code of length n* is a subset $C \subseteq A^n$. Elements of *C* are *code words* and C is a *binary code* if $A = \{0,1\}$.

❖ Parity Check Code Example:

The following binary code $C_p = \{0000,00011,...11110\}$ of length 5 is formed by appending a *parity check bit* to the end of each message word.

MESSAGE WORD	CODE WORD
0000	00000
0001	00011
0010	00101
0011	00110
0100	01001
0101	01010
0110	01100
0111	01111
1000	10001
1001	10010
1010	10100
1011	10111
1100	11000
1101	11011
1110	11101
1111	11110

Using the code C_p, we can detect up to one-bit error during transmission, but we cannot correct any errors.

❖ 3-Repetitions Code Example:

The following binary code

$$C_3 = \{000000000000,000100010001,...111111111111\}$$

of length 12 is formed by repeating each message word three times.

MESSAGE WORD	CODE WORD
0000	000000000000
0001	000100010001
0010	001000100010
0011	001100110011
0100	010001000100
0101	010101010101
0110	011001100110
0111	011101110111
1000	100010001000
1001	100110011001
1010	101010101010
1011	101110111011
1100	110011001100
1101	110111011101
1110	111011101110
1111	111111111111

Using this code, we can correct up to one-bit error during transmission.

This gain comes at a price: C_3 has information rate 4/12 = 1/3, lower than the information rate of C_p which is 4/5. The information rate of a binary code is the ratio of the number of significant bits of information in each word, to the total length of each word. More generally for an alphabet of size $A = |q|$, the *information rate* of a code C of length n over A is $\log_q |C| / n$. We

seek codes with high information rate, *and* high error-correcting capability.

Richard W. Hamming (1915–1998), was a pioneer in computer design and error-correcting codes. The *Hamming distance* between two words $x, y \in A^n$, denoted $d(x, y)$, is the number of position in which they differ.

- Example:

$$d(10010,00111) = 3$$

The *minimum distance* of a code $C \subseteq A^n$ is the minimum of $d(x, y)$ for all $x \neq y$ in the code *C*. In 1950, Hamming proposed a general method for constructing error-correcting codes. Hamming codes are possibly the simplest class of error-detecting and correcting codes that can be applied to a single code word.

For every integer *m* there is a (2^m - 1) bit Hamming code which contains *m* parity bits and (2^m - 1 - *m*) information bits. Within this Hamming code, the parity bits are intermixed with the information bits as follows: If we number the bit positions from 1 to 2^m - 1, the bits in position 2^k, where $0 \leq k \leq m$ - 1, parity bits and the bits in the remaining positions are information bits. Thus, parity bits are always in position 2^0 through 2^{m-1}.

In general, the value of each parity bit is chosen so that the total number of 1's in a specific group of bits positions is even, and these groups are chosen so that no information bit is covered by the same combination of parity bits. It is this arrangement that gives the code its correcting capability. More precisely, for each parity bit in position 2^k, its corresponding group of information bits includes all those bits in the position whose binary representation has a 1 in position 2^k.

Hamming code described above can detect and correct a single error. By adding a future check bit, we can create a Hamming code that can detect two errors and correct one.

2.4 Boolean Algebra

George Boole was an English mathematician (1815 - 1864) who developed a mathematical analysis of logic and published it in his book *"An investigation of the laws of thought"* in 1854. In 1938, Claude Shannon published a paper entitled *"A symbolic analysis of relay and switching circuits"* which applied Boolean algebra to switching circuits using relay. In digital systems, Boolean algebra is used to design digital circuits and to analyze their behavior.

Boolean algebra consists of a set of *elements S*, a set of *functions f* that operate on members of *S*, and a set of basic laws called *axioms* that define the properties of *S* and *f*. The set of elements making up a Boolean algebra have two possible values, 0 or 1. These elements may be variables or they may be literals (i.e. constants) that have fixed values of 0 or 1. A Boolean algebra with *n* variables has a set of 2^n possible sets of values of these variables.

Only three functions or operations are permitted in Boolean algebra. The first two are the logical *OR* represented by a plus '+', and the logical *AND* represented by a dot '·'. The third operation permitted in Boolean algebra is that of negation (*NOT*) or complementation. The complement of 0 (i.e. $0'$) is 1, and the complement of the 1 (i.e. $1'$) is 0. The priority of an *AND* operator is higher than that of an *OR* operator. The effect of the three operations *OR*, *AND*, *NOT*, is illustrated by means of the truth table given below.

NOT	*AND*	*OR*
$0' = 1$	$0 \cdot 0 = 0$	$0 + 0 = 0$
$1' = 0$	$0 \cdot 1 = 0$	$0 + 1 = 1$
	$1 \cdot 0 = 0$	$1 + 0 = 1$
	$1 \cdot 1 = 1$	$1 + 1 = 1$

These rules may be extended to any number of variables. Boolean variables obey similar commutative, distributive, and associative laws as the variables of conventional algebra. If and only if for every $(x_0, x_1) \in S$, $x_0 \cdot x_1$, $x_0 + x_1$, x_0', x_1', also belong to the set

of Boolean elements. This axiom is called the enclosure property and implies that Boolean operations on Boolean variables or constants always yield Boolean results.

The basic axioms of Boolean algebra governing variables, operators and constants are given below.

AND	*OR*	*NOT*
$0 \cdot x = 0$	$0 + x = x$	$(x')' = x$
$1 \cdot x = x$	$1 + x = 1$	
$x \cdot x = x$	$x + x = x$	
$x \cdot x' = 0$	$x + x' = 1$	

These equations may be proved by substituting all the possible value for x (i.e. 0 or 1). One of each pair of axioms can be obtained from the other by using an important property of Boolean algebra called the *duality principle*. This principle states that any algebraic equality derived from these axioms will still be valid whenever the *OR* and *AND* operators, and identity elements 0 and 1, have been interchanged.

In addition to the axioms that are given in the definition of a Boolean algebra, we can also derive additional laws, called *theorems* of Boolean algebra. The lists of six basic theorems of Boolean algebra are given below.

Theorem 1 *(idempotency)*	$x_0+x_0 = x_0$	$x_0 \cdot x_0 = x_0$
Theorem 2	$x_0+1 = 1$	$x_0 \cdot 0 = 0$
Theorem 3 *(absorption)*	$x_1 \cdot x_0+x_0 = x_0$	$(x_1+x_0) \cdot x_0 = x_0$
Theorem 4 *(involution)*	$(x_0')' = x_0$	
Theorem 5 *(associativity)*	$(x_0+x_1)+x_2 = x_0+(x_1+x_2)$	$x_0 \cdot (x_1 \cdot x_2) = (x_0 \cdot x_1) \cdot x_2$
Theorem 6 *(De Morgan's law)*	$(x_0+x_1)' = x_0' \cdot x_1'$	$(x_0 \cdot x_1)' = x_0'+x_1'$

These theorems are particular useful when we perform algebraic manipulation of Boolean expressions. Axioms are given and need no proof. Theorems must be proven, either from axioms or from other theorems that have already been proven. Since the two-valued Boolean algebra has only two elements, we can also show the validity of these theorems by using truth tables. A truth table is constructed for each side of the equation that is stated in the theorem. Then both sides of the equation are checked to see that they yield identical results for all possible combinations of variable values.

2.5 Boolean Functions

In general, functions can be defined as algebraic expressions that are formed from variables, operators, parentheses, and an equal sign. More specifically, Boolean expressions are formed from binary variables and the Boolean operators *AND*, *OR*, and *NOT*.

When we compute the values of Boolean expressions, we must adhere to a specific order of computation: namely, *NOT*, *AND*, and *OR*. Binary variables in these expressions can be take only two values, 0 and 1. For a given value of the variables, then, the value of the function is either, 0 and 1. The Boolean expression can be characterized as having *OR terms* and *AND terms*. Each term contains literals, where a *literal* indicates a variable or its complement. The number of terms and literals is usually used as a measure of the expression's complexity and frequently as a measure of its implementation cost. Consider, for example, the Boolean function $f(x_0, x_1, x_2) = x_0 \cdot x_1 + x_0 \cdot x_1' \cdot x_2' + x_0' \cdot x_1 \cdot x_2$ can be characterized as having one *OR* term and three *AND* terms.

Any Boolean function can also be defined by a truth table, which lists the value the function has for each combination of its variables values. As a rule, the truth table for any Boolean function of n variables has 2^n rows, which represent every possible combination of variable values, so that value entered in the

function column in each row is the value of the function for that particular combination of variable values.

2.6 Canonical Forms

In this section, we show how truth tables can be converted into algebraic expressions. For an *n*-variable function, each row in the truth table represents a particular assignment of binary values to those *n* variables. We can define a Boolean function, usually called *minterm*, which is equal to 1 in only one row in the truth table and 0 in all other rows. Since there are 2^n different rows in the truth table, there are also 2^n minterms for any *n* variables. Each of these minterms, denoted as m_i, can also be expressed as a product, or *AND* term, of *n* literals, in which each variable is complemented if the value assigned to it is 0, and uncomplemented if it is 1. More formally, this means that each minterm m_i can be defined as follows.

Let $i = b_{n-1}...b_0$ be a binary number between 0 and $2^n - 1$, which represents an assignment of binary values to the *n* binary variables x_j such that $x_j = b_j$ for all *j* where $0 \le j \le n - 1$. In this case a minterm of *n* variables $x_{n-1}, x_{n-2}, ..., x_0$, could be represented as

$$m_i(x_{n-1}, x_{n-2}, ..., x_0) = a_{n-1}a_{n-2}...a_0$$

where for all k such that $0 \le k \le n - 1$,

$$a_k = \begin{cases} x_k & if\ b_k = 1 \\ x_k^{'} & if\ b_k = 0 \end{cases}$$

The unique algebraic expression for any Boolean function can be obtained from its truth table by using an *OR* operator to combine all minterms for which the function is equal to 1. In other words, any Boolean function can be expressed as a sum of its minterms

$$f(x_{n-1},...,x_0) = \sum_{F?=1} m_i$$

for which the function is equal to 1. The *OR*ing of minterms is called a sum because of its resemblance to summation.

Similarly to an *AND* term of *n* literals being called a minterm, an *OR* term of n literals is called a *maxterm*. A maxterm can be defined as a Boolean function that is equal to 0 in only one row of its truth table and 1 in all other row. Each maxterm, M_i, can be expressed as a sum, of *OR* term, of *n* literals in which each variable would be uncomplemented if the value assigned to it is 0, and complemented if it is 1. Note that each maxterm is the complement of its corresponding minterm, and vice versa, $(m_i)' = M_i$ and $(M_i)' = m_i$.

The unique algebraic expression for any Boolean function can be obtained from its truth table by using an AND operator to combine all the maxterms for which the function is equal to 0. Any Boolean expression can be expressed as a product of its maxterms

$$f(x_{n-1},...,x_0) = \prod_{F?=0} M_i$$

for which the function is equal to 0. The *AND*ing of maxterms is called a product because of its resemblance to multiplication.

Any Boolean function that is expressed as a sum of minterms or as a product of maxterms is said to be in its canonical form. These two canonical forms provide unique expression for any Boolean function defined by truth table.

2.7 Logic Gates

To implement the Boolean functions we construct *logic circuit*, which contains one or more *logic gates*. Each logic gates performs one or more Boolean operation (*AND*, *OR*, *NOT*). The collection

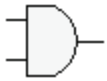
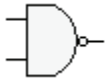
of logic gates that we use in constructing logic circuits is called the *gate library*, and the gates in the library are called *standard gates*.

In particular case of two variables we have $2^2 = 4$ possible different combinations. We can associate a different function with each of these $4^2 = 16$ values to create all possible functions of two variables. In general, n variables have $(2^2)^n$ functions which are given below

NAME	FUNCTION VALUE FOR (x_0, x_1) = 00 01 10 11	ALGEBRAIC EXPRESSION $y_i = f(x_0,x_1)$
Zero	0 0 0 0	$y_0 = 0$
AND	0 0 0 1	$y_1 = x_0 \cdot x_1$
Inhibition	0 0 1 0	$y_2 = x_0 \cdot x_1'$
Transfer	0 0 1 1	$y_3 = x_0$
Inhibition	0 1 0 0	$y_4 = x_0' \cdot x_1$
Transfer	0 1 0 1	$y_5 = x_1$
XOR	0 1 1 0	$y_6 = x_0 \cdot x_1' + x_0' \cdot x_1$
OR	0 1 1 1	$y_7 = x_0 + x_1$
NOR	1 0 0 0	$y_8 = (x_0 + x_1)'$
XNOR	1 0 0 1	$y_9 = x_0 \cdot x_1 + x_0' \cdot x_1'$
Complement	1 0 1 0	$y_{10} = x_1'$
Implication	1 0 1 1	$y_{11} = x_0 + x_1'$
Complement	1 1 0 0	$y_{12} = x_0'$
Implication	1 1 0 1	$y_{13} = x_0' + x_1$
NAND	1 1 1 0	$y_{14} = (x_0 \cdot x_1)'$
One	1 1 1 1	$y_{15} = 1$

Two of these functions called *NOR* and *NAND* are intrinsically better than others are because this function (*De Morgan's law*) can provide *AND*, *OR* and *NOT* Boolean operation.
Based of technological criteria only eight functions was selected to be implemented as *standard gate*, namely, the *Complement*, *Transfer*, *AND*, *OR*, *NAND*, *NOR*, *XOR*, and *XNOR* functions. In

table given below, we show the graphic symbols and Boolean expression of each of these eight gates.

NAME	GRAPHIC SYMBOL	FUNCTIONAL EXPRESSION
Inverter		$y = x_0'$
Driver		$y = x_0$
AND		$y = x_0 \cdot x_1$
OR		$y = x_0 + x_1$
NAND		$y = (x_0 \cdot x_1)'$
NOR		$y = (x_0 + x_1)'$
XOR		$y = x_0 \cdot x_1' + x_0' \cdot x_1$
XNOR		$y = x_0 \cdot x_1 + x_0' \cdot x_1'$

The function of the *inverter* is to complement the logic value of its input; we place a small circle at the output of its graphic symbol to indicate this *logic complementation*. We also use a triangle symbol to designate a *driver* circuit, which implements the *transfer function* by replicating the input value at its output. A diver is equivalent to two inverters that are connected in cascade, so that the output of the first inverter serves as the input of the second.

The *AND* and *OR* gates are used to implement the Boolean operators *AND* and *OR*, whereas the *NAND* and *NOR* gates are used to implement those functions that are the complement of *AND* and *OR*. A small circle is used at each of these outputs, which indicate this complementation. The *NAND* and *OR* gates are used extensively and are far more popular than the *AND* and *OR* gates, simply because their implementation is more simple in comparison with *AND* and *OR* gates. Note that the *XNOR* gate is the complement of *XOR* gate, as indicated by the small circle on its output line. When we are implementing Boolean functions with this basic gate library, we usually try to find the Boolean expression, of the canonical form of the function, which will best satisfy a given set of design requirements.

2.8 Finite State Machine

Each logic circuit we have encountered up to this point has been a *combinational circuit* whose output is a function of its input only. That is, given knowledge of a combinational circuit's inputs together with its Boolean functions, we can always calculate the state of its outputs. Circuits, whose outputs depend not only on their current inputs, but also on their past inputs, are called *sequential circuits*. Even if we know the structure of a sequential circuit (i.e. its Boolean function) and its current input, we cannot determine its output state without knowledge of its past history (i.e. its past internal state).

The basic building block of sequential circuits is *bistable*, just as the basic building block of the combinational circuit is the gate. A bistable is so called because, for a given input, its output can remain in one of two stable states indefinitely. For a particular set of inputs, the output may assume either a logical zero or a logical one, the actual value depending on the previous inputs. Such a circuit has the ability to remember what has happened to it in the past and is therefore a form of *memory element*.