International Journal of Pure and Applied Mathematics

Volume 21 No. 1 2005, 127-134

FUNCTION LANGUAGE IN DIGITAL LOGICS

Kenneth K. Nwabueze

Department of Mathematics
University of Brunei
Gadong, BE 1410, BRUNEI
e-mail: nwabueze@fos.ubd.edu.bn

Abstract: The concept of function is an example of a topic which has connections in almost all areas of computer science, especially in digital logics. In designing digital logics, there may be situations in which the designer may need the system to be able to "undo" certain processes. If that is a requirement, what the designer does, in simple mathematical terms, is to restrict the domain of operation in order to allow the process to be a function which is one-to-one. The purpose of this short note is to discuss some simple logical operations and terms associated with computer science in the context of functions. This will provide beginning students of computer science a thorough understanding and some examples of the concept of functions.

AMS Subject Classification: 03D05, 03E20
Key Words: functions, digital logics, logical operation, cryptography, number conversions

1. Introduction

Like any other machine, a computer hardware is normally switched on in order for electrons to flow through and activate it. This is just all that a computer hardware requires in order to start functioning. A computer is like a little child that needs to be told specifically what to do, and computer programs are the tools that tell the computer precisely what to do after being switched on. Computers understand only one language, namely the machine code. The machine code is a sequence of bits (binary digits), namely, 0 and 1. Be that as

Received: April 27, 2005

it may, the programs to be executed with the bits, are written in such a way that they obey all the rules that make up the definition of a function in the classical sense. However, this relationship is usually not very transparent if not carefully considered.

Computers use binary digits for internal representations of integers, and either base 8 (octal) or 16 (hexadecimal) for display purposes. Therefore, we start this note by discussing operations involved in the conversion of numbers from one base to the other in terms of the concept of mathematical functions in the classical sense. We shall extend the discussion by summarizing the process involved in logical operations, compilers, error message generators, and simple cryptography. Although these concepts are usually presented without the benefit of functions, student's understanding of the topic can be enhanced if the concepts are viewed in a broader context via functions.

2. Number Base Conversion

The first obvious example of a function is the conversion of numbers from one number base to another. To see this, let $f_{a,b}(x)$ be the number base conversion process which takes a value expressed in base a into base b; for example $f_{2,10}(11001) = 25$ and $f_{16,10}(AFB2) = 44978$ (cf [1]). It is easy to see that $f_{a,b}(x)$ is a function in the classical sense.

Recall that in the function $f_{a,a}(x)$, from one base into itself, does not change the number; that is, $f_{a,a}(x) = x$. Moreover, $f_{a,a}(x)$ is one-to-one, and so has an inverse. Also recall that when we convert from a first base into a second base and then convert the result in the second base into the first base, we get the original number back again, that is, $f_{b,a}(f_{a,b}(x)) = x$, and this is a well known property of a one-to-one function. The idea of one-to-one functions is usually utilized in designing digital electronics, where there may be situations in which the designer may need the system to be able to "undo" certain processes. When that is the case, what the designer does, in simple mathematical terms, is to restrict the domain so that the process becomes a one-to-one function. Finally recall that a practical method for conversion between two bases different from base 10, is to convert from the first base to base 10 and then convert from base 10 into the second base. This means that $f_{a,b}(x) = f_{10,b}(f_{a,10}(x))$, and this is simply a composition of functions.

2.1. Functions from Base 2, 8 and 16

We have seen that the conversion of numbers from one base to another base is a well defined function. We now give particular examples of functions arising from conversions between some of the bases used by the computer for internal representation and external display, namely bases 2, 8, and 16. Note that these bases are related, since each base is a power of 2, that is $2^1 = 2,\ 2^3 = 8$, and $2^4 = 16$. We now display the conversion of a base 2 number into a base 8 number as a function $f_{a,b}$; for example when $a = 2$ and $b = 8$ one has $f_{2,8}(1111101)$. Recall that this is done by separating the base 2 number into groups of three binary digits (going from right to left) as follows: 1 111 101. Each group of digits is then converted into the appropriate octal, that is, $f_{2,8}(1111101) = f_{2,8}(1\ 111\ 101) = 175$. The reverse process of going from base 8 into base 2 is equally easy. Recall that the value of $f_{8,2}(2716)$ is computed by taking each of the octal digits in the base 8 number and converting them into three binary digits: $f_{8,2}(2716) = 010\ 111\ 001\ 110 = 010111001110$. A similar process is used between base 2 and base 16, except that each of the hexadecimal digits represents 4 binary digits. For example, $f_{2,16}(10101001) = f_{2,16}(1010\ 1001) = AD$. To convert from base 16 to base 2, each hexadecimal number is replaced with 4 binary digits. For example: $f_{16,2}(A3C7) = 1010\ 0011\ 1100\ 0111 = 1010001111000111$. It will be easy for most students to see that the steps used in the above conversions are properties of a function in the classical sense.

3. Logical Operations

We now discuss some logical operations in terms of functions.

3.1. The $NOT(x)$ Function

The unary NOT operation is a function in the classical sense, and has the following two function calls: $NOT(True) = False,\ \ NOT(False) = True$. Note that the function NOT is one-to-one, and so there exists an inverse function, say NOT^{-1}, where $NOT^{-1}(False) = True$ and $NOT^{-1}(True) = False$. Because we have that $NOT(False) = True$ and $NOT(True) = False$, we conclude that the function NOT is its own inverse. A quick way of looking at this situation is to note that $NOT(NOT(True)) = True$ and $NOT(NOT(False)) = False$, or $NOT(NOT(x)) = x$, showing again that NOT is its own inverse. This function is called an idempotent function. Therefore, the function NOT is an example of an idempotent function.

3.2. The $OR(x, y)$ Function

The binary $OR(x, y)$ operation is a function in the classical sense, and we have the following four function calls:

$$\begin{aligned} OR(True, True) &= True\,, \\ OR(True, False) &= True\,, \\ OR(False, True) &= True\,, \\ OR(False, False) &= False\,. \end{aligned}$$

Note that the above four function calls imply that, although $OR(x, y)$ is a function, it is not a one-to-one function; and so the inverse does not exist. To see this, observe that there are three instances where $OR(x, y) = True$ and we cannot predict the specific values of x and y which produced the result of $True$. The only thing that can be inferred is that at least one of the two values must be $True$. Be that as it may, we can make the $OR(x, y)$ function a one-to-one function by an appropriate restriction on the domain; that is, we need to specify a subset of the function which is one-to-one. One example of a typical restriction of the domain of $OR(x, y)$ would be that $x = False$. Another example of a restriction would be that $y = False$. A third trivial example of a restriction would be that $x = y = False$. The restrictions in the three examples above result in a one-to-one function, and so an inverse exists for each of these examples.

3.3. The $XOR(x)$ Function

The binary XOR operation is a function in the classical sense, with the following four function calls:

$$\begin{aligned} XOR(True, True) &= False\,, \\ XOR(True, False) &= True\,, \\ XOR(False, True) &= True\,, \\ XOR(False, False) &= False\,. \end{aligned}$$

This implies that $XOR(x, y)$ does not have an inverse, since it is not a one-to-one function. To see that this is not one-to-one, observe that there are two instances where $XOR(x, y) = True$ and the specific values of x and y which produced that result of $True$ cannot be predicted. However, one can predict that the x and y values had to be different. Note also that there are two instances where $XOR(x,y) = False$, and the specific values of x and y which produced that result of $False$ cannot be predicted. However, we can conclude that x and y have to be equal. Although $XOR(x,y)$ is not one-to-one, one can restrict the domain by specifying a subset of the function which is one-to-one. For an inverse to exist, one can, for example, restrict the domain of $XOR(x,y)$ to be $x = True$. Other examples of a restriction would be that $x = False,\ y = True$, or $y = False$. Similar to the $OR(x,y)$ and $XOR(x,y)$ functions, one can derive equivalent conclusion for the $NOR(x,y),\ NAND(x,y)$, and $AND(x,y)$ operations. We present their function calls next.

3.4. The $NOR(x,y)$ Function

$NOR(x,y)$ is a binary function with the four function calls:

$$\begin{aligned} NOR(True,True) &= False\,, \\ NOR(True,False) &= False\,, \\ NOR(False,True) &= False\,, \\ NOR(False,False) &= True\,. \end{aligned}$$

3.5. The $AND(x,y)$ Function

$AND(x,y)$ is the binary function with the following four function calls:

$$\begin{aligned} AND(True,True) &= True\,, \\ AND(True,False) &= False\,, \\ AND(False,True) &= False\,, \\ AND(False,False) &= False\,. \end{aligned}$$

3.6. The $NAND(x)$ Function

$NAND(x)$ is a binary function with the four function calls:

$$\begin{aligned} NAND(True,True) &= False\,, \\ NAND(True,False) &= True\,, \\ NAND(False,True) &= True\,, \\ NAND(False,False) &= True\,. \end{aligned}$$

4. Compilers and Error Message Generators

Recall that the compiler evaluates the entire computer program and then translates all the programming statements into a machine language program, which is then executed at once. A computer language compiler can be regarded as a function because each valid command in the source file is converted into a predictable series of machine language commands (cf. [2]). If one considers only the assembly language programming, then one has a one-to-one function, because each mnemonic corresponds to one machine language command (cf. [2]). Although some machine code carries more than one byte, we still have that it is a one-to-one function because we have a single unique command. On the other hand, the part of the compiler which provides error codes or error messages can be regarded as a many to one function, because many different errors are identified with the same error code or error message. One common example of such errors is the SYNTAX ERROR message which can arise from something like: a spelling error command, a use of invalid command, a punctuation error, a use of improper command, and so on. This implies that if the computer generates the SYNTAX ERROR message, then it is impossible to predict what actually is responsible for that error unless you examine the specifics of the line where the problem is and correct it.

5. Cryptographic Functions

Encryption means the process of transforming data or a message into some unintelligible or unrecognizable, in such a way that the transformation is reversible. This means that it is possible to restore the original data or message. The process of encryption usually involves a set of instructions that enables a step-by-step procedure for transforming the data or message. This process is in most cases a one-to-one or an idempotent function because an encryption algorithm is always associated with a reverse procedure that restores the original message. The reverse process is called decryption. Taken together, the techniques of encryption and decryption are called cryptography.

A simple illustration of a key transformation or function is the XOR operation defined in Section 3. The XOR function can be defined with the following rules of binary addition:

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

Since these define addition rules in the binary system, we have used the notation

"+" to denote *XOR* addition above. By the use of the *XOR* addition, we can use any string of zeros and ones as a key to encrypt and decrypt data or a message. For example suppose we have a message which is a string of four binary digits, $M = 1011$, and we are given a four digit key $k = 1010$. We can encrypt the message M, with the key k, by addition, that is, a function f defined by

$$f(M) \;=\; M\text{“}+\text{”}k \;= C \;:=\; 1011 + 1010 \;=\; 0001 \;\; \text{(with no carry)}.$$

So we have that the encrypted message C is 0001. Note that the function f is an idempotent function. Also note that anyone seeing this value (without knowing the encryption key) will be unable to interpret the original message M, because it could have been any one of sixteen possible combinations of four zeros or ones. However, if you know the key, it is easy to recover the original message. All that needs to be done to recover the message is to add the key k to the encrypted message C following the *XOR* rules of addition defined above. This implies, applying the function f twice. So the original message, M, is recovered thus:

$$f^2(M) \;=\; f(M+k) \;=\; (M+k)+k \;=\; C+k \;=\; 0001+1010 \;=\; 1011.$$

In other words, performing an *XOR* addition with the same key twice restores the original message. This process simply says that $f^2(M) = M$. The security of this system depends on the length of the key k; and our definition of f depends on this key, since $f(M) = M + k = C$. Note here that there are sixteen possible keys of length four, that is, 24 possibilities. In general, for a key of length n, there are 2^n possible keys.

Finally, note that most commercial *DES* uses keys of length 56, that is, the key space has 2^{56} possibilities. Given current techniques of cryptanalysis, this key space is no longer large enough for security. With the current state of technology it is possible decrypt the message by employing a *brute force* attack on a message, where by a brute force attack we mean the process of trying out every possible key. If we increase the key space to 2^{128} then the system is quit secure; at least with respect to a brute force attack. Note that a key space of 2^{128} is 2^{72} times as large as the common *DES* key space of 2^{56}.

References

[1] K.K. Nwabueze, *Basic Number Theory: A First Course,* Educational Technology Centre, Universiti Brunei (2003).

[2] V. Strong, Functions: Computer science connections, *IMSA Math. Journal*, **2** (1993).