Designing a Quantum Hamming Code Generator, Detector and Error Corrector Using IBM Quantum Experience

Preprint · January 2020

DOI: 10.13140/RG.2.2.24318.33606

3 authors:

Hridey Narula
University of Delhi

Bikash K. Behera
Bikash's Quantum (OPC) Pvt. Ltd.

Prasanta K. Panigrahi
Indian Institute of Science Education and Research Kolkata

Designing a Quantum Hamming Code Generator, Detector and Error Corrector Using IBM Quantum Experience

Hridey Narula,[1,*] Bikash K. Behera,[2,3,†] and Prasanta K. Panigrahi[3,‡]

[1]*Sri Guru Tegh Bahadur Khalsa College, University of Delhi, 110007, Delhi, India*
[2]*Bikash's Quantum (OPC) Pvt. Ltd., Balindi, Mohanpur 741246, West Bengal, India*
[3]*Department of Physical Sciences,*
Indian Institute of Science Education and Research Kolkata, Mohanpur 741246, West Bengal, India

Since the advent of digital communication, various types of error detectors have been constructed. A particular class of these are based on the concept of parity. One family of these is Hamming code developed by R. W. Hamming. This code belongs to the linear-error correcting family and is widely used when the error rate is low. This paper aims to implement Hamming (7,4) version of Hamming code generator and detector/corrector on IBM quantum experience platform.

I. INTRODUCTION

Quantum computing[1–4] is a field that lies at the intersection of two different fields quantum mechanics and computer science. The field was initiated by multiple physicists and computer scientists in the late 20^{th} century. Feynman was one of the first advocates of the power of computing[5]. The interest in quantum computing reached new heights when Shor found out a quantum algorithm that was much faster than the classical counterpart, now called Shor's algorithm[6].

Hamming code[7–9] is a family of linear error-correcting codes developed by R. Hamming for detecting up to two bits of error and automatically correcting up to one bit of error. They are typically used when the expected error rate is low. Essential to understanding Hamming code is the concept of **parity**[10]. We will first overview *even* parity which refers to the 'evenness' of a number. Consider a binary number 101. Clearly, the number of 1's in this binary number is 2 (i.e. even). Therefore, the corresponding parity bit is 0 so that the final number is **0**101, where the bit in bold is parity bit, and the total number of 1's is even (hence the name *even* parity). Now, suppose that we have a binary number like, say, 100. Clearly, the number of 1's in this binary number is 1 (i.e. odd). Therefore, the corresponding parity bit is 1 so that the final number **1**100 has even number of 1's. One can see that the even parity bit is the one which converts the total numbers of 1's in the number to be an even number. Likewise, the *odd* parity bit is the one which converts total number of 1's to an odd number.

Parity allows detection of one bit errors. Suppose, for example, that Alice wants to send 1100 to Bob. She will also add another parity bit (suppose, even parity) **0**. Therefore she sends 01100 to Bob. Now if due to noise the third bit gets flipped and the number transmitted to Bob is 01000. Bob notices that the parity bit is 0, therefore the total number of 1's should be even. Clearly, it is not the case. Therefore, Bob correctly deduces that the message is *corrupted*. Parity bit, by itself cannot indicate which bit is corrupted. The entire message has to be scrapped and resent.

Hamming code works on the principle of parity bits, but uses them in such a way so that up to two corrupted bits can be detected or one error bit can be corrected. Basically, a simple formula dictates the number of parity bits to be used and sent along with the data bits at specific positions to the receiver. The receiver then generates checker bits to analyse which bits are corrupted and have to be corrected. Hamming codes are used widely in computer memory (ECC memory) where bit errors are expected to be low. One of the popular Hamming codes is Hamming code (7,4). It can be used to identify and correct up to one error bit (an addition of one more parity bit can identify two error bits). It is used when there are 4 data bits to be sent. Three additional parity bits are required at specific positions according to a certain algorithm. Hamming code can be conveniently modeled using matrix algebra[9]. Hamming code has various applications[11–13] and a quantum version would only be an improvement. The principal aim of this paper is to design circuit for Hamming code (7,4) along with a detector/corrector. We will be using IBM Quantum Experience platform for designing the quantum circuits and executing on the quantum simulator.

The rest of the paper is structured as follows: preliminary knowledge of the Hamming code is introduced in Section II. Section III is concerned with designing and implementing the Hamming code generator while section IV is concerned with designing and implementing Hamming code detector/corrector. Section V concludes the paper. A circuit diagram is provided in the appendix as a reference for the reader.

II. PRELIMINARIES

We will be designing Hamming code (7,4) generator and detector/corrector. Here, we describe the working of Hamming code. Consider that Alice wants to send a n-bit data to Bob. To use the Hamming code, she has to add some parity bits. Let the number of parity bits be 'm'. Then, the number of data bits and parity bits satisfy the relation:

$$2^m \geq m + n + 1 \tag{1}$$

D7	D6	D5	P4	D3	P2	P1

TABLE I. Table showing the arrangement of different bits. D indicate data bits and P indicate parity bits.

If we have to send 4 bits, i.e., n = 4 then by trial and error we can calculate m = 3 ($2^3 \geq 3 + 4 + 1$). The bits are distributed as in Table I.

The explanation is as follows: Parity bits occupy the positions of the form 2^i where i is a positive integer. Therefore, parity bits occupy positions 1, 2 and 4. The parity bits are represented as Pi i.e., P1, P2 and P4. The remaining places are the data bits represented as Dj where j is the position i.e., D3, D5, D6 and D7. It should be noted that if the bits were transmitted without parity then D3 would have actually been D1.

From the representation, we can see that the name (7,4) corresponds to 4 data bits and 7 total bits (including 3 parity bits). We will represent 0 using $|0\rangle$ and 1 using $|1\rangle$. We will be using some *Quantum Registers* for input/output and working, and one *Classical Register* for measurement. For all the circuits, all lines are initially set to $|0\rangle$. User can change the input by either placing a Pauli **X** gate in the circuit, or by adding an appropriate line of code in the input section of the program.
The Hamming code generator is relatively simple and is designed using IBM Quantum Experience's circuit composer. The detector/corrector is designed using QISKit notebook due to its relative complexity.

The output comes in the form of histogram as well as the value of classical register for the given number of shots. For ease of readability, we have defined the multiple functions ("gates") which act as the basic blocks of our circuits, see Table II for reference.

III. GENERATOR

A. Theory

We have to design a Hamming code generator. We will be using *even* parity. P1 bit is concerned with the data bits having 1 as their LSB. P2 is concerned with bits having 1 as their second LSB and so on. It can be seen using Boolean algebra[14] that we get the following equations for Parity bits:

$$
\begin{aligned}
P1 &= D3 \oplus D5 \oplus D7 \\
P2 &= D3 \oplus D6 \oplus D7 \\
P4 &= D5 \oplus D6 \oplus D7
\end{aligned} \tag{2}
$$

Clearly, we just have to use a *3 input XOR* gate for generating the Parity bits.

S.No.	Name	Working
1	fun_or(c,r1,r2,r3)	Takes quantum circuit and two quantum registers as input and returns their OR through a third quantum register passed in argument. Applies a series of CCNOT and NOT gates.
2	or3(c,r1,r2,r3,b,r4)	Takes quantum circuit, three quantum registers and a buffer quantum register as input and returns their OR through a fourth quantum register passed in argument. Uses fun_or() function.
3	or4(c,r1,r2,r3,r4,b1,b2,r5)	Takes quantum circuit, four quantum registers and two buffer quantum registers as input and returns their OR through a fifth quantum register passed in argument. Uses fun_or() and or3() functions.
4	xor4(c,r1,r2,r3,r4,r5)	Takes quantum circuit, four quantum registers as input and returns their XOR through a fifth quantum register passed in argument. Uses a series of CNOT gates.
5	mod _xor(c,r1,r2,b)	Takes quantum circuit, two quantum registers and one buffer quantum register as input and stores the XOR of the input registers in the first register. Uses a series of CNOT gates and $\|0\rangle$ operations.

TABLE II. Functions (gates) defined for making the Hamming code corrector/detector. c indicates quantum circuit, r is used to indicate quantum registers and b indicates quantum registers used as buffers.

B. Implementation

First, we have to design a XOR gate[15]. This has already been done using quantum gates[16]. The parity bit generation, involves some XOR operations and appropriate measurements. The circuit is shown in Fig. 1.

C. Results

The resulting histogram is shown in Fig. 2. We had given 1000 as input. By Eq. (2), we can see that P1 = 1, P2 = 2 and P4 = 1 as well. Therefore, the output

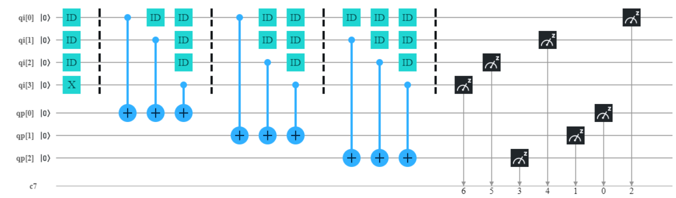

FIG. 1. Quantum circuit for a Hamming code generator. The extreme left side of barrier contains the inputs. The input is 1000 since X acts on MSB and I acts on the rest of the bits. It is to be noticed that the order of measurements corresponds to Table I

.

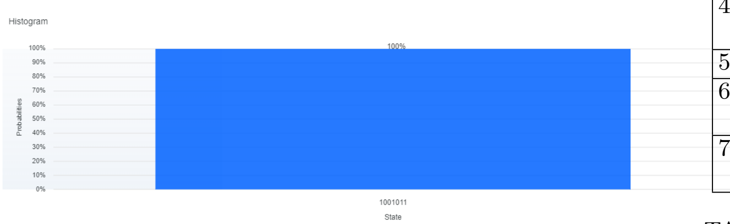

FIG. 2. The result of input 1000 in the Hamming code generator with 1024 shots.

from Hamming code generator should be 1001011, since all three parity bits are 1. Clearly, this matches with the output shown in Fig. 2 and hence we can say that our Hamming code generator works correctly.

IV. DETECTOR AND CORRECTOR

A. Theory

The detector first generates a checker bit corresponding to each parity bit. In Eq. (2), we XORed the data bits, now we will XOR the parity bits as well. The checker bits are given by the following equations :

$$\begin{aligned} C1 &= P1 \oplus D3 \oplus D5 \oplus D7 \\ C2 &= P2 \oplus D3 \oplus D6 \oplus D7 \\ C4 &= P4 \oplus D5 \oplus D6 \oplus D7 \end{aligned} \tag{3}$$

Assuming that no bit is corrupt, we can safely expect all the checker bits to be 0 indicating no bit has to be corrected. However, if one or more checker bits are non-zero it indicates there is some corrupted bit. To identify and correct it, we first find its position. The position is given by $C_4C_2C_1$. To find this bit, we use a 3×8 decoder[17]. After decoding, for each bit we have Table III.

We can clearly see that this is also a XOR operation.

B	S	B^+	Remark
0	0	0	Non corrupt 0 bit
0	1	1	Corrupt 0 bit
1	0	1	Non corrupt 1 bit
1	1	0	Corrupt 0 bit

TABLE III. Table showing the value of bits after going through a 'judgement' to determine if it's corrupted. B is the bit, S is the output from decoder, B^+ is the bit after correction.

S.No.	Name	Type	Purpose
1	qp(3)	Quantum	Stores parity inputs
2	qd(1)	Quantum	Stores data inputs
3	qch(3)	Quantum	Calculates checker bits
4	qb(8)	Quantum	Acts as buffer for intermediate calculations
5	qe(8)	Quantum	An extra buffer
6	qxc(3)	Quantum	Used for storing complemented checker bits
7	c(7)	Classical	Used for measurement of corrected bits

TABLE IV. Table showing the registers used for making the detector/corrector. Number in parentheses indicate the size of the register.

B. Implementation

Since the detector/corrector is more complicated we have programmed it using QISKit. We have Table IV showing registers used for the same. We have given 1011011 as input assuming D5 to be corrupted.

The python code for the same is:

```

%matplotlib inline
# Importing standard Qiskit libraries and
    configuring account
from qiskit import QuantumCircuit, execute, Aer,
    IBMQ,QuantumRegister,ClassicalRegister
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *

# Loading IBM Q account
provider = IBMQ.load_account()

#Getting simulator
simulator=Aer.get_backend('qasm_simulator')

'''
There are 6 quantum registers and 1 classical
    register used in this circuit.
qp QuantumRegister is composed of parity input
    lines
qd QuantumRegister is composed of data input
    lines
qch QuantumRegister is composed of checker lines
qb,qe QuantumRegisters are composed of buffer
    lines which are used for intermediate
    calculations
qx QuantumRegister is composed of inverted input
    lines for calculations
qo QuantumRegister is composed of output lines
```

```
    which displays the output
c  ClassicalRegister is used for meausring qo
circuit is the QuantumCircuit composed of the
    above the register
'''

qp=QuantumRegister(3)
qd=QuantumRegister(4)
qch=QuantumRegister(3)
qb=QuantumRegister(1)
qe=QuantumRegister(8)
qxc=QuantumRegister(3)
c=ClassicalRegister(7)

circuit=QuantumCircuit(qp,qd,qch,qb,qe,qxc,c)

#Functions are defined below

def fun_or(qc,q0,q1,q2):
    qc.x(q0)
    qc.x(q1)
    qc.ccx(q0,q1,q2)
    qc.x(q2)
    qc.x(q1)
    qc.x(q0)

def or3 (qc,q0, q1, q2, b, q3):
    fun_or(qc,q0, q1, b)
    fun_or(qc,b,q2,q3)
    qc.reset(b)


def or4 (qc,q0, q1, q2,q3, b1,b2,q4):
    or3(qc,q0, q1, q2,b1,b2)
    fun_or(qc,b2,q3,q4)
    qc.reset(b1)
    qc.reset(b2)

def xor4(qc,q0,q1,q2,q3,q4):
    qc.cx(q0,q4)
    qc.cx(q1,q4)
    qc.cx(q2,q4)
    qc.cx(q3,q4)

def mod_xor(qc,q1,q2,qb):
    circuit.cx(q1,qb)
    circuit.cx(q2,qb)
    circuit.reset(q1)
    circuit.cx(qb,q1)
    circuit.reset(qb)


#Input Below

#1011011

circuit.x(qd[3])
circuit.x(qd[1])
circuit.x(qp[2])
circuit.x(qp[1])
circuit.x(qp[0])

#Input Above

#Generating Checker Bits

xor4(circuit,qp[0],qd[0],qd[1],qd[3],qch[0])
xor4(circuit,qp[1],qd[0],qd[2],qd[3],qch[1])
xor4(circuit,qp[2],qd[2],qd[1],qd[3],qch[2])

#Using a 3 to 8 decoder

for i in range(0,3):
    circuit.cx(qch[i],qxc[i])
    circuit.x(qxc[i])

or3(circuit,qxc[0],qxc[1],qxc[2],qb[0],qe[7])
or3(circuit,qch[0],qxc[1],qxc[2],qb[0],qe[6])
or3(circuit,qxc[0],qch[1],qxc[2],qb[0],qe[5])
or3(circuit,qch[0],qch[1],qxc[2],qb[0],qe[4])
or3(circuit,qxc[0],qxc[1],qch[2],qb[0],qe[3])
or3(circuit,qch[0],qxc[1],qch[2],qb[0],qe[2])
or3(circuit,qxc[0],qch[1],qch[2],qb[0],qe[1])
or3(circuit,qch[0],qch[1],qch[2],qb[0],qe[0])

for i in range(0,8):
    circuit.x(qe[i])

#Using a simple XOR gate, with output getting
    stored in bit 1

mod_xor(circuit,qp[0],qe[1],qb[0])
mod_xor(circuit,qp[1],qe[2],qb[0])
mod_xor(circuit,qd[0],qe[3],qb[0])
mod_xor(circuit,qp[2],qe[4],qb[0])
mod_xor(circuit,qd[1],qe[5],qb[0])
mod_xor(circuit,qd[2],qe[6],qb[0])
mod_xor(circuit,qd[3],qe[7],qb[0])

#Measurements

circuit.measure(qp[0],c[0])
circuit.measure(qp[1],c[1])
circuit.measure(qd[0],c[2])
circuit.measure(qp[2],c[3])
circuit.measure(qd[1],c[4])
circuit.measure(qd[2],c[5])
circuit.measure(qd[3],c[6])

#Execution of the circuit

job = execute(circuit, simulator, shots=1)
result=job.result()
counts = result.get_counts(circuit)

#Output

print(counts)
circuit.draw()
plot_histogram(counts)
```

C. Results

The resulting circuit is given in Fig. 4. The histogram is given in Fig. 3.

The output is 1001011 which is the same as the input as in Fig. 2. Clearly, the original message has been recovered and the corrupted bit has been corrected. Therefore, our detector/corrector works correctly.

V. CONCLUSION

To summarize, we have designed and implemented quantum circuits for Hamming code (7,4) generator and

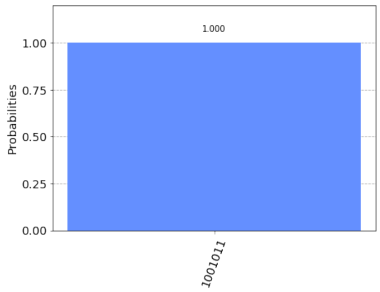

FIG. 3. Histogram showing results for input being 1011011, i.e., D5 corrupted bit from the output generated by Fig. 1.

detector/corrector using IBM quantum experience and QISKit. This is extremely useful in protecting data from noise. In future works, the authors will try to expand the present work by trying to implement more error detector techniques. With the increasing interest in quantum communication, the future of this field looks promising.

ACKNOWLEDGEMENTS

H.N. acknowledges the hospitality provided by IISER Kolkata and Bikash's Quantum (OPC) Pvt. Ltd. during this project work. He also acknowledges the support of his family, friends and contemporaries. B.K.B. acknowledges the financial support of Institute fellowship provided by IISER Kolkata. We acknowledge IBM Q Experience team for providing access to IBM quantum simulator and allowing us to perform the experiments.

Appendices

The quantum circuit for the detector/corrector is given in Figs. 4. Here, q83 lines are parity lines, q84 lines are data lines, q85 are check lines, q18 is the output line, q86 is buffer line, q87 is extra buffer, q88 is the inverted check bit line and c12 is the classical register.

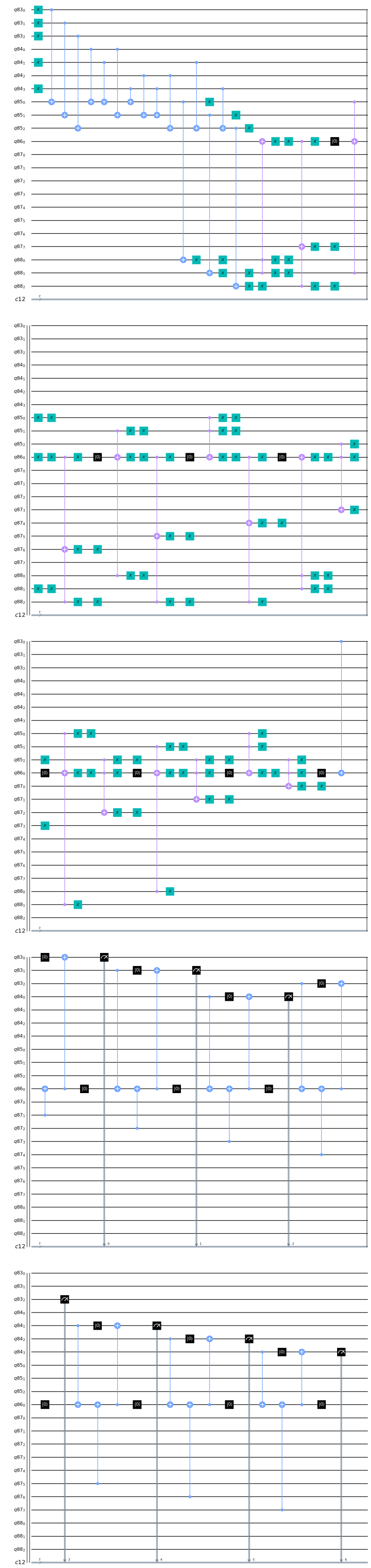

FIG. 4. The quantum circuit for detector/corrector. Every buffer line is reset after every operation by using $|0\rangle$ on them. This is to clear the previous value. The combinations of NOT and CCNOT gates are used for making the classical OR gate. The XOR gate is implemented using CNOT gates in series.

* hrideynarula74@gmail.com
† bikash@bikashsquantum.com
‡ pprasanta@iiserkol.ac.in

[1] M. A. Nielsen and I. L. Chuang, Quantum computation and Quantum Information, 10th ed. Cambridge, U.K.: Cambridge Univ. Press (2010).
[2] P. Kaye, R. Laflamme, and M. Mosca, An introduction to quantum computing, Oxford University Press (2007).
[3] H.-K. Lo, Quantum Cryptology, Introduction to Quantum Computation and Information, **77**, 76-119 (1998).
[4] C. P. Williams, Quantum computing and quantum communications, in Proc. 1st NASA Int. Conf. QCQC **1509**, Palm Springs, CA, USA, 200-217 (1998).
[5] R. P. Feynman, Simulating physics with computers, Int. J. Theor. Phys. **21(6/7)**, 467-488 (1982).
[6] P. W. Shor, Algorithms for quantum computation: discrete logarithms and factoring, Proceedings 35th Annual Symposium on Foundations of C.Sc.,pp. 124-134 (1994).
[7] Digital Principles and Applications, Leach, Malvino and Saha. McGraw Hill Education , 8^{th} edition (2014).
[8] R.W. Hamming, Error Detecting and Error Correcting Codes, Bell Labs Tech. J. **29**, 147 (1950).
[9] https://en.wikipedia.org/wiki/Hamming _code
[10] https://en.wikipedia.org/wiki/Parity _bit
[11] C.-. Chan, and C.-C. Chang, An efficient image authentication method based on Hamming code, Pattern Recog. Sci. Direct, **40**, 681 (2007).
[12] K. Furutani, K. Arimoto, H. Miyamoto, T. Kobayashi, K. Yasuda and K. Mashiko, A built-in Hamming code ECC circuit for DRAMs, IEEE J. of Solid-State Circuits, **24**, 50-56 (1989).
[13] R. Hentschke, F. Marques, F. Lima, L. Carro, A. Susin and R. Reis, Analyzing area and performance penalty of protecting different digital modules with Hamming code and triple modular redundancy, Proceedings. 15th Symp. on Int. Circuits and Systems Design, 95-100 (2002).
[14] J. E. Whitesitt, Boolean Algebra and Its Applications, Dover Books on C.Sc. (2010).
[15] D. R. Choudhury and K. Podder, Design of Hamming Code Encoding and Decoding Circuit Using Transmission Gate Logic, IRJET **2**, (2015).
[16] K. Anand, B. K. Behera and P. K. Panigrahi, Experimental Realization of NEQR and Encryption Algorithm using Generalized Affine Transform and Logistic Map, DOI: 10.13140/RG.2.2.12533.32485 (2019).
[17] H. Narula, B. K. Behera and P. K. Panigrahi, Designing a Simple Quantum Morse Encoder and Decoder Using IBM Quantum Experience, DOI: 10.13140/RG.2.2.18770.71366 (2019).