Implementation of Quantum Gates based Logic Circuits using IBM Qiskit

Enaul haq Shaik*
Department of Electronics and Communication Engineering
Vasireddy Venkatadri Institute of Technology
Nambur, Guntur (Dist), Andhra Pradesh, India-522201.
*haq.enamul11@gmail.com

Nakkeeran Rangaswamy
Department of Electronics Engineering
School of Engineering and Technology,
Pondicherry University
Pondicherry, India – 605014.
nakkeeranpu@gmail.com

***Abstract*—Quantum computing is an emerging field that depends upon the basic properties of quantum physics and principles of classical systems. This leads a way to develop systems to solve complex problems that a classical system cannot do. In this article, we present simple methods to implement logic circuits using quantum gates. Logic gates and circuits are defined with quantum gates using Qiskit in Python. Later, they are verified with quantum circuits created by using IBM Quantum. Moreover, we propose a way of instantiating the basic logic circuits to design high-end logic expressions. As per our knowledge, the proposed simple approach may be helpful to solve the complex logical problems in near future.**

Keywords—quantum gate; logic circuits; instantiation; Qiskit; IBM Quantum.

I. INTRODUCTION

High speed computing plays a very important role in our day to day activities from personal to public. But, it all happen at a cost of increased power consumption [1]. Advancements in transistor technology led to the invention of Complementary Metal-Oxide Semiconductor (CMOS) transistor accomplished the need of the time. However, various applications of computing technologies such as commerce, medical, governance and other logistics using CMOS still in wane due to high speed up demands. Thus, the present day computations need a substitute to the existing computing systems [2-3]. Quantum computing is one among the remarkable substitutes.

Quantum computer takes leverage of basic properties of quantum mechanics to perform computations [4] like superposition of states [5], interference [5], and entanglement [4]. Moreover, it gets a computational advantage with crucial properties of reversibility [6], and each computing subsytem must be reversible [4]. These properties lead quantum computers to promise wide capabilities of quantum information processing in the areas like sensing [7] and communication [8] also.

Quantum researchers, apart from complex computational tasks, working towards logical systems to come up with fast decision making capabilities. The basic building blocks of logic circuits are the basic logic gates, which can be realised by using quantum gates [4]. Quantum system designers have been working over the implementations of logic circuits [6,9]. These were implemented using the quantum gates theoretically with a qubit characterization [9]. Several quantum programming software are available to design and implement the quantum circuits such as #Q [10], Cirq [11], Qiskit [12] etc., among them #Q cannot generate quantum circuits.

In the proposed work, we implement logic circuits using quantum gates as well as a process of instantiation. High end logic functions can be implemented by instantiating the basic logic circuits. This leads to simplify the design aspects as the complex coding is not required. Qiskit, an open source software by IBM [14], is used to simulate the logic circuits with quantum gates through Python coding. Also, these circuits are verified by simulating the same over IBM Quantum [13] simulator. Moreover, a simple method of instantiating the basic circuits is proposed to define the high end logic structures. The remaining part of the article is organized as follows. Section II details about the basic of quantum gates followed by implementation of logic circuits using these gates in Section III. Section IV describes the instantiation of basic logic circuits to implement high end logic functions, and finally it is concluded in Section V.

II. QUANTUM GATES

In general, quantum scientists work over the two-level qubit systems to define quantum algorithms and, in turn quantum computers. In these systems, there are two states *viz.,* $|0\rangle$ and $|1\rangle$, and these are called to be the computational basis of the two-level qubit system [4] represented with vectors as follows;

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

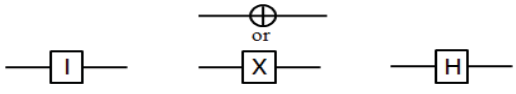
These states are same as binary bits, '0' and '1' used in classical systems. But the qubit differs from the classical bit as it may also be represented as a superposition of both states $|0\rangle$ and $|1\rangle$. In order to evolve the operations, some sort of operators are used over these states, which are in the form of matrices. There are three basic operators depend on number of inputs, and they are unary, binary and ternary. As far as the number of outputs are concerned, they are same as inputs due to reversibility [1]. Quantum gate is represented with its

978-1-7281-9180-5/20/$31.00 ©2020 IEEE

operator to perform an operation over the qubits. Basically, operators and gates are interchangeably used. In this article, only few operators are discussed which are required for the design of logic circuits. Also in this article, the term classical circuits which are being proposed using quantum operators or gates are basically logic circuits.

A. I, X, and H Gates

The Identity (I), NOT (X) and Hadamard (H) gates are represented with unary operators, whose symbols and matrix forms are illustrated in Fig.1.



$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

(a) (b) (c)

Fig. 1. Symbol and matrix representation of a) I, b) X, and c) H gates

Identity gate doesn't affect any qubit state, and thus it can be same as buffer in classical circuits. In Qiskit, usually, the state of the qubits is initialized with $|0\rangle$, and sometimes they need to be maintained as such for which this gate is used. The operation of this gate over the states of a qubit is given below.

$$I|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle \text{ and}$$

$$I|1\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

NOT or X gate inverts the state of the qubit i.e., $|0\rangle$ changes to $|1\rangle$ and $|1\rangle$ changes to $|0\rangle$. Any one of the symbols shown in Fig. 1(b) can be used to represent this gate. In Qiskit, as the state of the qubit at input is initialized with $|0\rangle$, this gate is used to flip the input i.e., if $|1\rangle$ is required. This operator is also known as bit flip operator [4] as it flips the input state as shown below;

$$X|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle \text{ and}$$

$$X|1\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$$

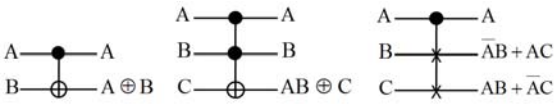
Hadamard or H gate is very crucial in quantum information and computing systems, which transforms a qubit into superposition of two states from a definite computational basis state [4]. This gate is used to define the inputs of a quantum circuit with all the possible combinations of the states of qubits. For example, in a 3-qubit quantum system, the possible input combinations are 2^3 i.e., 8. This gate or operator is represented with H symbol and its matrix form is shown in Fig. 1(c). When this operator is applied to $|0\rangle$ and $|1\rangle$, the superposition of both of them is as follows

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \text{ and}$$

$$H|1\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

B. CNOT, CCNOT and CSWAP Gates

Among the three gates shown in Fig. 2, the first one is binary operator and the remaining two are ternary operators, and these operators represent quantum gates. Their function conveys some logical operations, which are shown on the right side.



(a) (b) (c)

Fig. 2. Symbols of a) CNOT, b) CCNOT, and c) CSWAP gates

Controlled-NOT is abbreviated as CNOT and it is also called as Feynman gate. In normal, this gate provides XOR logic at the bottom output. If A = 1 (control input), the inverse of B is obtained (i.e., $\bar{B}$). Thus it is known as Controlled-NOT gate (NOT operation is controlled by A). This can be used as both NOT and XOR gates, provided the input is maintained as required.

As per the operation discussed above, for most of the binary and ternary quantum gates, there are two kinds of inputs *viz.,* control and target. In the CNOT gate, A is control input and B is target.

Controlled CNOT gate is abbreviated as CCNOT and is also called as Toffoli gate. In this, A and B are the control inputs and C is a target input. This gate can be used as a NOT gate to produce $\bar{C}$ when A = B = 1 Thus, it is called as a controlled CNOT as there are two control inputs. Apart from this, it can also be used as AND gate and XOR gate, to produce AB when C = 0 and A $\oplus$ C when B = 1 (also it can be B $\oplus$ C when A = 1), respectively.

Controlled SWAP gate is abbreviated as CSWAP as it swaps the values of B and C at the output when A = 1. Also, it can be used as OR gate to produce A + B at the middle output when C = 1 and as an AND gate to produce AB at the bottom output when C = 0. This gate is also known as Fredkin Gate. Also, in CCNOT and CSWAP gates, A and B are interchangeable.

The logic functions of XNOR, NAND and NOR can also be derived by appending X gate at the corresponding output line. According to the functions these gates are providing, they can be connected with each other in an appropriate and implement the required logic expressions. In the next section, we describe the way to define, create and simulate logic circuits with quantum gates using Qiskit and IBM Quantum.

III. Implementation of Logic Circuits using Quantum Gates

In this section, implementation of logic circuits with quantum gates is described by using Qiskit and IBM Quantum. Python programming language is used to define the logic gates and circuits. The quantum gates *viz.,* I, X, H, CNOT, CCNOT and CSWAP described in the above section are designated as i, x, h, cx, ccx, and cswap, respectively in the code. In the proposed work, the code of logic circuits using quantum gates is defined as functions and by calling them the logic circuit is simulated. XOR gate code is given in the following subsection and only *if* clauses are explained for the remaining circuits. In the codes, initially QuantumRegisters

978-1-7281-9180-5/20/$31.00 ©2020 IEEE

(holds qubits), ClassicalRegisters (holds output classical bits) and QuantumCircuits (store operations for the quantum system) are the classes to be declared [14]. This declaration also includes size of the quantum and classicalregisters. Later, an algorithm for the logic circuit using quantum gates, and then measurement at the required outputs to store them in classical register. To run the QuantumCircuit, Qiskit backend 'qasm_simulator' is used. The initial states of qubits are at $|0\rangle$, and if $|1\rangle$ is needed then X operator is applied over that particular qubit.

A. Implementation of Logic Gates

For any basic logic gate the number of outputs are one, and the number of inputs are two except NOT which has only one. For these logic gates, the four input combinations are applied and output is observed. In order to have these four input combinations, H gates are to be applied to both the qubits. But when we simulate it, a histogram plot of the probability of getting classical output logic '1' and '0' will be generated wherefrom a prompt output for an input combination cannot be predicted. Thus, python codes are written in such a way that, the output for a particular input combination is described separately. The code shown below is of XOR gate defined with CNOT gate (cx), and the input variation at qubits q[0] and q[1] in all the four parts of the *if* clause are described by I and X operators. For the quantum circuit defined by the *if* clause, the qubits q[1] and q[0] are inputs A and B, and the output is measured from q[1] into classical register c[0]. The input at the qubits need to vary between '0' and '1' for the measurement to be done. These are defined with I and X operators, respectively.

```
def XOR(input1,input2):

    q = QuantumRegister(2)
    c = ClassicalRegister(1)
    qc = QuantumCircuit(q, c)

    if input1=='0' and input2=='0':
        qc.i(q[0])
        qc.i(q[1])
        qc.cx(q[0],q[1])
    elif input1=='0' and input2=='1':
        qc.x(q[0])
        qc.i(q[1])
        qc.cx(q[0],q[1])
    elif input1=='1' and input2=='0':
        qc.i(q[0])
        qc.x(q[1])
        qc.cx(q[0],q[1])
    else:
        qc.x(q[0])
        qc.x(q[1])
        qc.cx(q[0],q[1])

    qc.measure(q[1],c[0])
    backend = Aer.get_backend('qasm_simulator')
    job = execute(qc,backend,shots=1024,memory=True)
    output = next(iter(job.result().get_counts()))
    return output
```

The print statement shown below displays the output by calling XOR gate function, and the results can be observed through the statements given below.

```
print('\nResults of XOR gate ')
for input1 in ['0','1']:
    for input2 in ['0','1']:
        print('Input combination',input1,input2,
              'gives output',XOR(input1,input2))
```

```
Results of XOR gate
Input combination 0 0 gives output 0
Input combination 0 1 gives output 1
Input combination 1 0 gives output 1
Input combination 1 1 gives output 0
```

Similarly, other logic gates can be designed by changing qubit input and quantum gate in the *if* clause. The body of a condition of *if* clause of AND and OR gates is given below. The first two lines are used to vary input combinations and the remaining two lines are according to the use of quantum gates CCNOT and CSWAP described in the previous section. The measurement is done at q[2] and q[1] of the corresponding quantum gates for AND and OR, respectively. Similarly, the *if* clause of NOT gate is given further below which is self explanatory.

```
# AND gate
    qc.i(q[0])
    qc.i(q[1])
    qc.i(q[2])
    qc.ccx(q[0],q[1],q[2])
```

```
# OR gate
    qc.i(q[0])
    qc.i(q[1])
    qc.x(q[2])
    qc.cswap(q[0],q[1],q[2])
```

```
# NOT gate
    if input=='0':
        qc.x(q[0])
    else:
        qc.i(q[0])
```

Qiskit can also be used to create quantum circuits, but the operator observed at the input would be X as the last input combination '11'. Thus, in the proposed work, the quantum circuit is designed using IBM Quantum and verified the results with generated histogram wherein the probability of getting output '0' and '1' is illustrated. The quantum circuits of XOR and AND gates are created and simulated to produce the result in histogram, which are shown in Fig. 3. The quantum circuit consists of three sections, first section is encoding of the inputs, middle one is algorithm to perform an operation, and the last is measurement at the selected output lines to store them in Classical register (c) [14]. These are separated by barriers (broken lines). The H gates in both the quantum circuits are applied to the qubits q[0] and q[1] for encoding so that the circuit can be simulated for all the input combinations shown above in the results of XOR gate. As the H gate performs superposition over the state of qubit, all the four combinations evolved. Similarly, if there are three qubits which are processing inputs of quantum circuits and applied with H gate, then 8 input combinations will be evolved. The qubit q[3] (non-processing input) in Fig.3(b) is to be maintained at state $|0\rangle$ for AND logic as described in the previous section, and thus it is left unapplied with any gate. Because, all the qubits are initialized with state $|0\rangle$. For the XOR gate, the probability of getting '0' or '1' is 50% as it is observed from the results produced by Qiskit. The histogram in Fig. 3(a) shows the same generated upon simulating quantum circuit of XOR over IBM Quantum hardware. As far as the AND gate is concerned, it provides '1' when both the inputs are '1's, and so the probability of getting '1' as output is

978-1-7281-9180-5/20/$31.00 ©2020 IEEE

25% and '0' is 75%. These probabilities in the histogram are illustrated by Fig. 3(b).

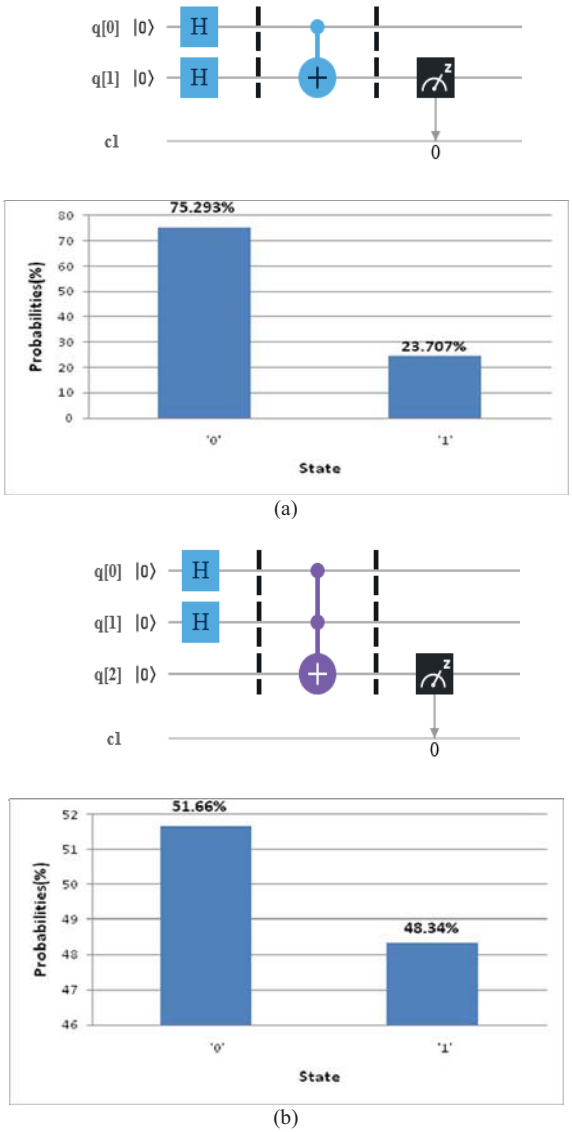
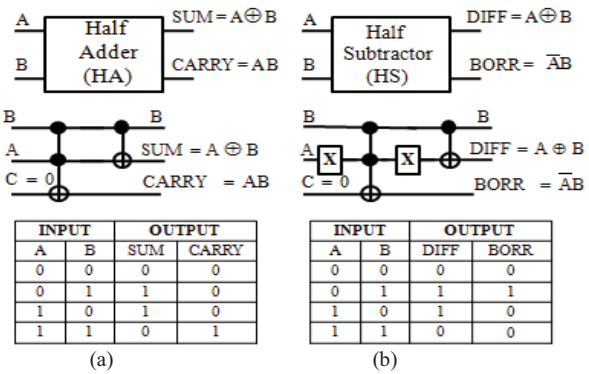


Fig. 3. Quantum circuit (top) and histogram (bottom) of (a) XOR and (b) AND gates.

B. *Implementation of Logic Circuits*

Apart from the logic gates, logic circuits can also be designed and simulated using the quantum gates. Also, these gates can be instantiated to implement high end logic functions. In this subsection, implementation of half adder and half subtractor is described using both Qiskit and IBM Quantum. The logic and quantum circuits of half adder and half subtractor are shown in Fig. 4 along with truth table. As per the output functions of SUM and CARRY for half adder, XOR and AND gates are required. For the design of it, CNOT and CCNOT gates are sufficient with inputs operated using H gates and measured for two outputs. This quantum circuit needs three qubits and two classical registers to store these two output bits SUM and CARRY. The body of the *if* clause of the half adder is given below.



INPUT		OUTPUT	
A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a)

INPUT		OUTPUT	
A	B	DIFF	BORR
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

(b)

Fig. 4. Block diagram (top), quantum circuit (middle), and truth table (bottom) of a) half adder and b) half subtractor.

```
    qc.x(q[0])
    qc.x(q[1])
    qc.i(q[2])
    qc.ccx(q[0],q[1],q[2])
    qc.cx(q[0],q[1])

qc.measure(q[1],c[1])
qc.measure(q[2],c[0])
```

CCNOT gate is followed by CNOT, and thus q[1] and q[2] gives XOR and AND of q[0] and q[1], respectively. So, q[1] and q[2] are measured into c[1] and c[0] for SUM and CARRY, respectively. The last two statements of the *print* function to produce the output is given below with results. As the size of the classical register is of two bits, the result returned is of two bits into *HA_res*.

```
HA_res = HA(input1,input2)
print('For the input combination',input1,input2,
      'the SUM  is ', HA_res[0],
      ' & CARRY is', HA_res[1])
```

```
Results of Half Adder
For the input combination 0 0 the SUM  is  0  & CARRY is 0
For the input combination 0 1 the SUM  is  1  & CARRY is 0
For the input combination 1 0 the SUM  is  1  & CARRY is 0
For the input combination 1 1 the SUM  is  0  & CARRY is 1
```

The quantum circuit and histogram generated using IBM Quantum is shown in Fig. 5(a). As per the truth table of half adder, the probability of getting output '00' and '01' is 25%, and that of '10' is 50% which can be verified with the histogram.

For half subtractor, the difference output (DIFF) is same as SUM of half adder, and the borrow is $\bar{A}B$. In order to produce these outputs, an additional H gate is to be taken at qubit q[1] to produce $\bar{A}$, and again another H gate to make it A for DIFF to generate. The body of the *if* clause is below. Qubit q[2] is always made at state $|0\rangle$ for AND function. The IBM's quantum circuit and the histogram are shown in Fig. 5(b) wherein, the probability of getting output '00', '10' and '11' is nearly 50%, 25% and 25%, respectively.

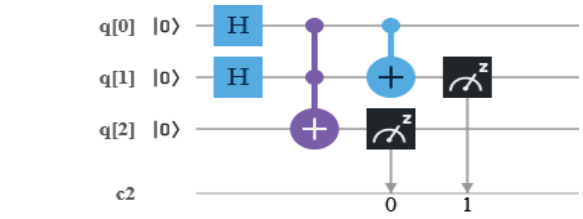
978-1-7281-9180-5/20/$31.00 ©2020 IEEE

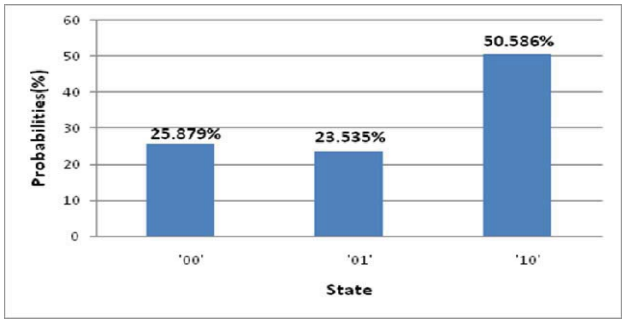
```
    qc.i(q[0])
    qc.i(q[1])
    qc.i(q[2])
    qc.x(q[1])
    qc.ccx(q[0],q[1],q[2])
    qc.x(q[1])
    qc.cx(q[1],q[0])

qc.measure(q[0],c[1])
qc.measure(q[2],c[0])
```

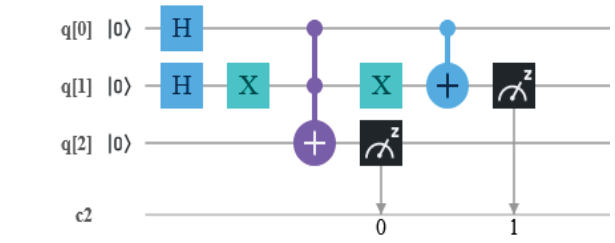
```
HS_res = HS(input1,input2)
print('For the input combination',input1,input2,
      'the DIFF  is ', HS_res[0],
      ' & BORR is', HS_res[1])
```

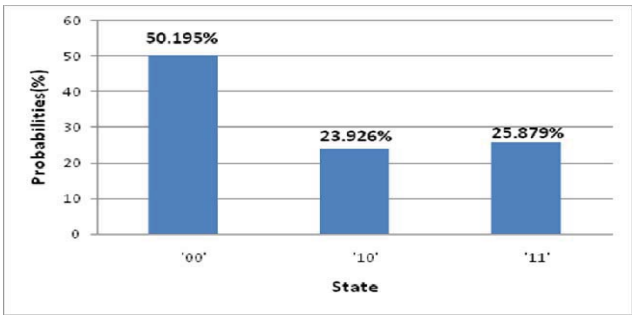
```
Results of Half Subtractor
For the input combination 0 0 the DIFF  is  0  & BORR is 0
For the input combination 0 1 the DIFF  is  1  & BORR is 1
For the input combination 1 0 the DIFF  is  1  & BORR is 0
For the input combination 1 1 the DIFF  is  0  & BORR is 0
```





(a)





(b)

Fig. 5. Quantum circuit (top) and histogram of a) half adder and b) half subtractor.

IV. INSTANTIATING QUANTUM CIRCUITS

The instantiation of the quantum gates based logic circuits is possible by calling the functions with the output returned from other functions. In this section, the way of implementing the logic functions using instantiation with function call is described. As discussed in the previous section, the logic gates (XOR, OR, AND, NOT) and circuits like half adder and half subtractor are simulated using code, which are defined as functions.

The outputs returned by the functions are used as inputs for other logic gate functions and circuits. For the implementation of half adder, XOR and AND gate functions are used and output in the classical register c[0] is shown as SUM and CARRY, respectively. Similarly, for the implementation of half subtractor, the returned output from the NOT gate function is instantiated with AND gate function as an input. The result obtained for both of these codes of half adder and half subtractor are same as that of the results observed in Fig. 6.

```
print('\nResults for the Half Adder')
for input1 in ['0','1']:
    for input2 in ['0','1']:
        print('For the input combinaitions',input1,input2,
              'the SUM is',XOR(input1,input2),
              '& CARRY is',AND(input1,input2))
```

```
print('\nResults of Half Subtractor')
for input1 in ['0','1']:
    for input2 in ['0','1']:
        print('For the input combination',input1,input2,
              'the DIFF is',XOR(input1,input2),
              '& BORR is',AND(NOT(input1),input2))
```

Similarly, any logic expression can be implemented by instantiating the logic gates. For example;

$Y = \overline{A}\overline{B} + A\overline{B} + AB$, by calling the logic gate functions, they can be instantiated to realize the expression.

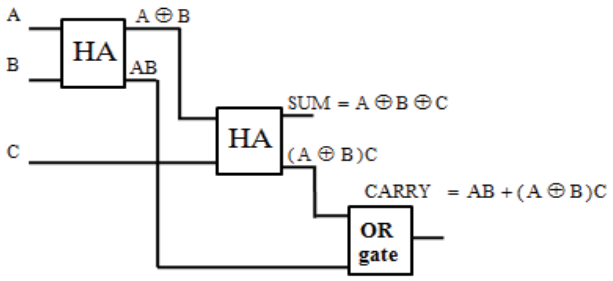
```
print('\nResults of the logic expression Y ')
for input1 in ['0','1']:
    for input2 in ['0','1']:
        print('Input combination',input1,input2,
              'gives output',OR(AND(input1,NOT(input2)),
              OR(AND(NOT(input1),NOT(input2)),AND(input1,input2))))
```

```
Results of the logic expression Y
Input combination 0 0 gives output 1
Input combination 0 1 gives output 0
Input combination 1 0 gives output 1
Input combination 1 1 gives output 1
```

When it comes to the full adder, the two half adder and one OR gate functions are there according to the schematic diagram illustrated in Fig. 6. The size of the classical register is of two bits c[0] and c[1]. Initially, the second *for* clause, the classical register is assigned to variable HA1 in which least significant bit HA1[0] is SUM and most significant bit HA1[1] is CARRY. In the second *for* clause, two bit combinations are occurred and with the third *for* clause all three bit combinations are evolved. Thus, the second half adder and OR gate need to be instantiated in the third *for* clause. HA1[0] is applied as input for second half adder

978-1-7281-9180-5/20/$31.00 ©2020 IEEE

function along with third qubit q[2]. The output from the classical register of second half adder is assigned to HA2. HA2[0] gives final SUM of the full adder, while HA1[1] and HA2[1] are applied as input for OR gate function to produce final CARRY of the full adder. The result produced is the truth table of the full adder. As the schematic diagram of the full subtractor is same as that of full adder, same process can be used to implement it but with half subtractor.



INPUT			OUTPUT	
A	B	C	SUM	CARRY
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig. 6. Schematic of full adder (top) using half adder and OR gate and truth table (bottom).

```
print('\nResults of Full Adder')
for input1 in ['0','1']:
    for input2 in ['0','1']:
        HA1 = HA(input1,input2)
        for input3 in ['0','1']:
            HA2 = HA(HA1[0],input3)
            OR_res= OR(HA1[1],HA2[1])
            print('    Inputs',input1,input2,input3,
                  'gives SUM  = ', HA2[0],
                  ' & CARRY = ', OR_res)
```

```
Results of Full Adder
Inputs 0 0 0 gives SUM  =  0  & CARRY =  0
Inputs 0 0 1 gives SUM  =  1  & CARRY =  0
Inputs 0 1 0 gives SUM  =  1  & CARRY =  0
Inputs 0 1 1 gives SUM  =  0  & CARRY =  1
Inputs 1 0 0 gives SUM  =  1  & CARRY =  0
Inputs 1 0 1 gives SUM  =  0  & CARRY =  1
Inputs 1 1 0 gives SUM  =  0  & CARRY =  1
Inputs 1 1 1 gives SUM  =  1  & CARRY =  1
```

V. CONCLUSION

Advancements in the CMOS technology have been in wane to cope up with need of high speed and complex problem solving requisites that led to develop quantum systems. In this article, the implementation of logic circuits with quantum gates was presented. Logic circuits defined using Qiskit were verified with IBM Quantum simulator. Further, a simple method of instantiating these quantum gate based logic circuits was proposed to realize high-end logic functions.

ACKNOWLEDGMENT

Authors are very much thankful to IBM Q Experience team to provide us access to IBM Quantum simulator and perform the experiments.

REFERENCES

[1] W.D. Pan, and M. Nalasani, “Reversible logic,” IEEE Potentials, vol. 24, pp. 38-41, March 2005.
[2] J.M. Shalf, and R. Leland, “Computing beyond Moore's law,” Computer, vol. 48, pp. 14-23, December 2015.
[3] T.M. Conte, E. Track, and E. DeBenedictis, “Rebooting computing: New strategies for technology scaling,” Computer, vol. 48, pp. 10-13, December 2015.
[4] J.D. Hidary, Quantum Computing: An Applied Approach, Springer Nature, Switzerland AG, 2019, pp. 3-10.
[5] M. Soeken, H. Thomas, and R. Martin, “Programming quantum computers using design automation,” proceedings of Design, Automation and Test in Europe, 2018.
[6] H. Thapliyal, “Mapping of subtractor and adder-subtractor circuits on reversible quantum gates,” Transactions on Computational Sciences XXVII, vol. 9570, pp. 10-34, April 2016.
[7] C. Degen, F. Reinhard, and P. Cappellaro, “Quantum sensing: Reviews of modern physics,” vol. 89, pp. 035002, July 2017.
[8] M. Krenn, M. Malik, T. Scheidl, R. Ursin, and A. Zeilinger, “Quantum communication with photons,” Optics in Our Time, Springer, December 2016.
[9] T.S. Humble, H. Thapliyal, M-C. Edgard, F.A. Mohiyaddin, and R.S. Bennink, “Quantum computing circuits and devices,” IEEE Design and Test, vol. 36, pp. 69-94, April 2019.
[10] https://docs.microsoft.com/en-us/quantum/
[11] https://cirq.readthedocs.io/en/stable/
[12] https://www.ibm.com/quantum-computing/technology/experience/
[13] https://qiskit.org/
[14] D. Koch, L. Wessing, and P.M. Alsing, “Introduction to coding quantum algorithms: A tutorial series using Qiskit,” arXiv:1903.04359v1, March 2019.

978-1-7281-9180-5/20/$31.00 ©2020 IEEE