Nota 4: Sumadores

En [1], Fig.6.6 y 6.7, se describen un sumador digital básico (*ripple carry adder*) de n bits y el circuito cuántico correspondiente. Este realiza la siguiente operación sobre $2(n+1)$ qubits:

$$|c_0\rangle\times|x_0\, x_1 \ldots x_{n-1}\rangle\times|y_0\, y_1 \ldots y_{n-1}\rangle\times|0\rangle \rightarrow |c_0\rangle\times|x_0\, x_1 \ldots x_{n-1}\rangle\times|s_0\, s_1 \ldots s_{n-1}\rangle\times|\, c_n\rangle. \quad (1)$$

Incluye otros n-1 qubits auxiliares, inicialmente en el estado $|00\ldots0\rangle$, cuyo estado final es $|c_{n-1}\, c_{n-2} \ldots c_1\rangle$. De ahí el número total de qubits mencionado en [1], es decir, $3n+1$.

Este circuito sugiere dos comentarios:

- ¿No se puede reducir el número de qubits, evitando el uso de qubits auxiliares?
- Si se utilizan qubits auxiliares, sería conveniente que vuelvan al estado inicial $|00\ldots0\rangle$. De esta manera pueden ser utilizados como qubits auxiliares para otras operaciones. Además, el hecho de que aquellos qubits vuelvan al estado básico hace que el sistema sea más estable y resistente a ruidos externos.

A continuación, se describen dos sumadores cuánticos que tienen en cuenta los comentarios anteriores. En la Sec. 1 se describe un sumador incluyendo $3n+1$ qubits, cuyos n-1 qubits auxiliares vuelven a su estado inicial. En la Sec.2 se describe un sumador incluyendo $2n+2$ qubits. En ambos casos se calculan primero los acarreos c_1, c_2, …, c_n, y luego los bits de la suma s_{n-1}, s_{n-2}, …, s_0.

1. Sumador con cálculo previo de los acarreos, versión 1

El cálculo previo de los acarreos es una técnica habitual en los sumadores digitales (por ejemplo [4], Sec.11.1.2). El circuito descrito en [2] utiliza dos tipos de operadores:

- *QCC* (*Quantic Carry Chain*), Fig.1.a, actúa sobre cuatro qubits y ejecuta la siguiente operación

$$|c_i\rangle\times|x_i\rangle\times|y_i\rangle\times|0\rangle \rightarrow |c_i\rangle\times|x_i\rangle\times|p_i\rangle\times|c_{i+1}\rangle, \tag{2}$$

  donde

$$p_i = x_i \oplus y_i,\ c_{i+1} = x_i y_i \oplus c_i p_i. \tag{3}$$

- *QS* (*Quantic Sum*), Fig.1.b, actúa sobre tres qubits y ejecuta la operación

$$|c_i\rangle\times|x_i\rangle\times|y_i\rangle \rightarrow |c_i\rangle\times|x_i\rangle\times|s_i\rangle, \tag{4}$$

  con

$$s_i = c_i \oplus x_i \oplus y_i. \tag{5}$$

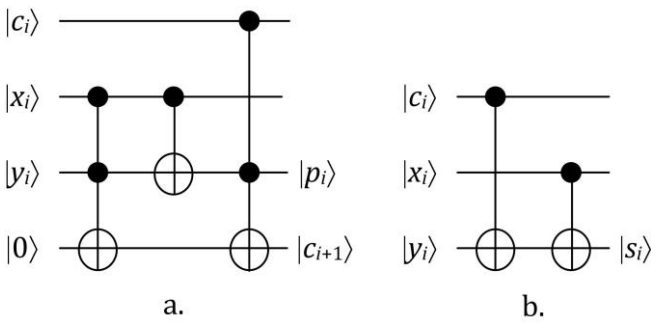


Figura 1 Componentes *QCC* y *QS*

El circuito completo (n = 3) se muestra en la Fig.2. Se observa que ejecuta la operación (1) y que los qubits auxiliares vuelven al estado básico $|0\rangle$.

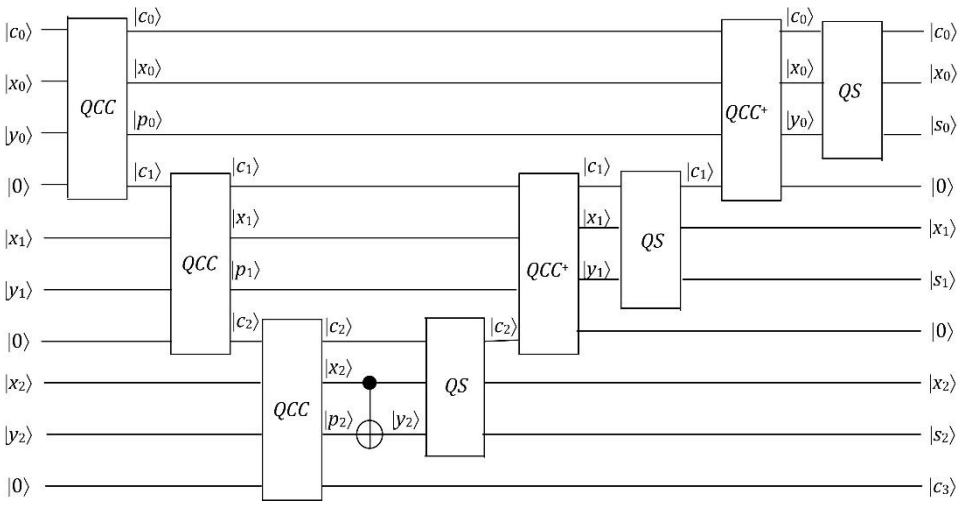


Figura 2 Sumador de 3 bits

Ejemplo 1

Defínanse operadores *QCC* y *QS* encapsulando los circuitos de la Fig.1 en sendas clases:

```
class QCC(cirq.Gate):
    def __init__(self):
        super(QCC, self)
    def _num_qubits_(self):
        return 4
    def _decompose_(self, qubits):
        I, A, B, O = qubits
        yield cirq.CCX(A, B, O)
        yield cirq.CX(A, B)
        yield cirq.CCX(I, B, O)
    def _circuit_diagram_info_(self, args):
        return ["QCC"] * self.num_qubits()
qcc = QCC()
```

```
class QS(cirq.Gate):
    def __init__(self):
        super(QS, self)
    def _num_qubits_(self):
        return 3
    def _decompose_(self, qubits):
        I, A, B = qubits
        yield cirq.CX(I, B)
        yield cirq.CX(A, B)
    def _circuit_diagram_info_(self, args):
        return ["QS"] * self.num_qubits()
qs = QS()
```

Se define también el operador QCC^+, invirtiendo el orden de las operaciones de *QCC*:

```
class QCCAd(cirq.Gate):
    def __init__(self):
        super(QCCAd, self)
    def _num_qubits_(self):
        return 4
    def _decompose_(self, qubits):
        I, A, B, O = qubits
        yield cirq.CCX(I, B, O)
        yield cirq.CX(A, B)
        yield cirq.CCX(A, B, O)
    def _circuit_diagram_info_(self, args):
        return ["QCCAd"] * self.num_qubits()
qccAd = QCCAd()
```

Con esos operadores se define un sumador de 4 bits:

```
n = 4
c = 1
x = [0,1,0,1]
y = [1,0,1,1]
###CIRCUITO###
reg = cirq.LineQubit.range(3*n+1)
circuito = cirq.Circuit()
if c == 1:
    circuito.append(cirq.X(reg[0]))
for i in range(n):
```

```
    if x[i] == 1:
        circuito.append(cirq.X(reg[3*i+1]))
    if y[i] == 1:
        circuito.append(cirq.X(reg[3*i+2]))
    circuito.append(qcc(reg[3*i],reg[3*i+1],
      reg[3*i+2],reg[3*i+3]))
circuito.append(cirq.CX(reg[3*(n-1)+1],
   reg[3*(n-1)+2]))
for i in range(n-1):
    circuito.append(qs(reg[3*(n-1-i)],
       reg[3*(n-1-i)+1],reg[3*(n-1-i)+2]))
    circuito.append(qccAd(reg[3*(n-2-i)],
       reg[3*(n-2-i)+1],reg[3*(n-2-i)+2],
          reg[3*(n-2-i)+3]))
circuito.append(qs(reg[0],reg[1],reg[2]))
###MEDICIÓN###
for i in range(n-1):
    circuito.append(cirq.measure(reg[3*i+3]))
for i in range(n):
    circuito.append(cirq.measure(reg[3*i+2]))
circuito.append(cirq.measure(reg[3*n]))
simulador = cirq.Simulator()
resultado = simulador.run(circuito, repetitions=1)
print(resultado)
```

La ejecución del programa completo genera el siguiente resultado:

```
0: ---X---QCC-------------------------------------------------------------QCCAd---QS-------
       |                                                                   |     |
1: -------QCC-------------------------------------------------------------QCCAd---QS-------
          |                                                                |     |
2: ---X---QCC-------------------------------------------------------------QCCAd---QS---M---
       |                                                                   |
3: -------QCC---QCC-------------------------------------------QCCAd---QS---QCCAd---M-------
             |                                                  |     |
4: ---X-------QCC-------------------------------------------QCCAd---QS-------------------
       |      |                                               |     |
5: -------------QCC-------------------------------------------QCCAd---QS---M---------------
       |        |                                             |
6: -------------QCC---QCC-----------------QCCAd---QS---QCCAd---M-----------------------
       |          |                       |     |
7: -------------------QCC-----------------QCCAd---QS-------------------------------------
       |          |                       |     |
8: ---X-----------QCC-----------------QCCAd---QS---M-----------------------------------
       |          |                       |
9: -------------------QCC---QCC-------QS---QCCAd---M-------------------------------------
                     |     |
10: --X--------------------QCC---@---QS-------------------------------------------------
                     |     |   |
11: --X--------------------QCC---X---QS---M-------------------------------------------
                     |
12: -----------------------QCC---M-------------------------------------------------

q(3)=0
```

```
q(6)=0
q(9)=0
q(2)=0
q(5)=0
q(8)=0
q(11)=1
q(12)=1
```

Se comprueba que la medición de los tres qubits auxiliares (3, 6, y 9) da como resultado 000, y que la medición de los qubits 2, 5, 8, 11 y 12 da como resultado 00011. Efectivamente, 1010 + 1101 + 1 = 11000.

2. Sumador con cálculo previo de los acarreos, versión 2

El circuito descrito en [3] utiliza dos tipos de operadores:

- *CY*, Fig.3.a, actúa sobre tres qubits. Teniendo en cuenta que
$$(c_i \oplus y_i)(x_i \oplus y_i) \oplus y_i = c_i x_i \oplus c_i y_i \oplus y_i x_i = c_{i+1}$$
se deduce que el operador *CY* ejecuta la siguiente operación:
$$|c_i\rangle \times |x_i\rangle \times |y_i\rangle \rightarrow |c_i \oplus y_i\rangle \times |x_i \oplus y_i\rangle \times |c_{i+1}\rangle. \quad (6)$$

- *XOR*, Fig.3.b, actúa sobre tres qubits y ejecuta la operación
$$|a\rangle \times |b\rangle \times |c\rangle \rightarrow |ab \oplus a \oplus c\rangle \times |ab \oplus a \oplus b \oplus c\rangle \times |ab \oplus c\rangle. \quad (7)$$
Obsérvese que si $a = c_i \oplus y_i$, $b = x_i \oplus y_i$ y $c = c_{i+1}$, entonces

  $ab = (c_i \oplus y_i)(x_i \oplus y_i) = y_i \oplus x_i y_i \oplus x_i c_i \oplus y_i c_i = y_i \oplus c_{i+1}$,

  $ab \oplus c = y_i$,

  $ab \oplus a \oplus c = y_i \oplus c_i \oplus y_i = c_i$,

  $ab \oplus a \oplus b \oplus c = c_i \oplus x_i \oplus y_i = s_i$.

  Por tanto
$$|c_i\rangle \times |x_i\rangle \times |y_i\rangle \xrightarrow{XOR} |c_i\rangle \times |s_i\rangle \times |y_i\rangle. \quad (8)$$

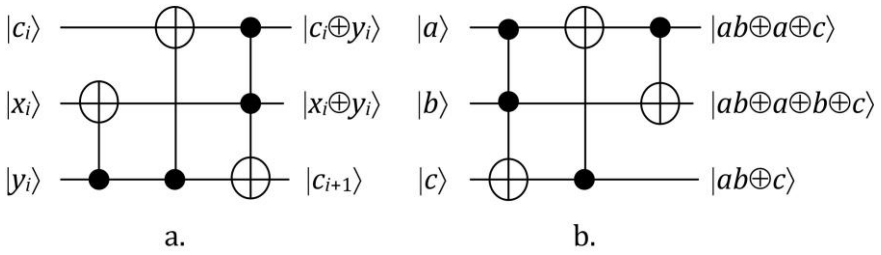


Figura 3 Componentes *CY* y *XOR*

El circuito completo (n = 3) se muestra en la Fig.4. Ejecuta la operación

$$|c_0\rangle\times|x_0\, x_1 \ldots x_{n-1}\rangle\times|y_0\, y_1 \ldots y_{n-1}\rangle\times|0\rangle \rightarrow |c_0\rangle\times|s_0\, s_1 \ldots s_{n-1}\rangle\times|y_0\, y_1 \ldots y_{n-1}\rangle\times|\, c_n\rangle. \quad (9)$$

En total se utilizan $2n$+2 qubits.

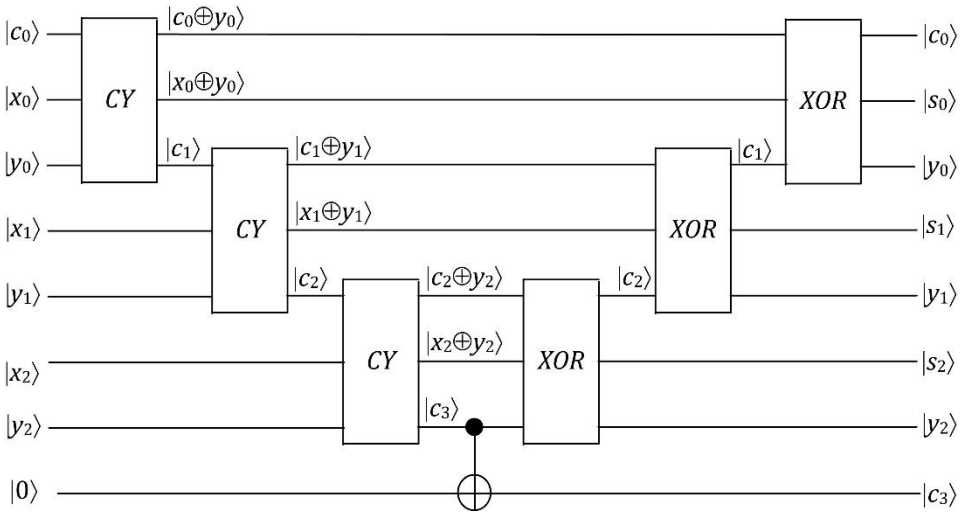


Figura 4 Sumador de 3 bits

Ejemplo 2

Defínanse los operadores *CY* y *XOR*:

```
class CY(cirq.Gate):
```

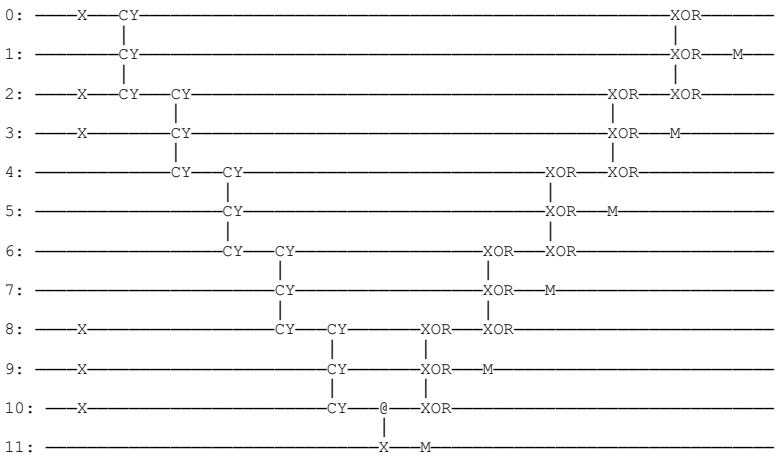
```
    def __init__(self):
        super(CY, self)
    def _num_qubits_(self):
        return 3
    def _decompose_(self, qubits):
        I, A, B = qubits
        yield cirq.CX(B, A)
        yield cirq.CX(B, I)
        yield cirq.CCX(I, A, B)
    def _circuit_diagram_info_(self, args):
        return ["CY"] * self.num_qubits()
cy = CY()
class XOR(cirq.Gate):
    def __init__(self):
        super(XOR, self)
    def _num_qubits_(self):
        return 3
    def _decompose_(self, qubits):
        I, A, B = qubits
        yield cirq.CCX(I, A, B)
        yield cirq.CX(B, I)
        yield cirq.CX(I, A)
    def _circuit_diagram_info_(self, args):
        return ["XOR"] * self.num_qubits()
xor = XOR()
```

Con esos operadores se define un sumador de 5 bits:

```
n = 5
c = 1
x = [0,1,0,0,1]
y = [1,0,0,1,1]
###CIRCUITO####
reg = cirq.LineQubit.range(2*n+2)
circuito = cirq.Circuit()
if c == 1:
    circuito.append(cirq.X(reg[0]))
for i in range(n):
    if x[i] == 1:
        circuito.append(cirq.X(reg[2*i+1]))
    if y[i] == 1:
        circuito.append(cirq.X(reg[2*i+2]))
circuito.append(cy(reg[2*i],reg[2*i+1],reg[2*i+2]))
```

```
circuito.append(cirq.CX(reg[2*n],reg[2*n+1]))
for i in range(n):
    circuito.append(xor(reg[2*(n-1-i)],
         reg[2*(n-1-i)+1],reg[2*(n-1-i)+2]))
###MEDICIÓN####
for i in range(n+1):
    circuito.append(cirq.measure(reg[2*i+1]))
print(circuito)
simulador = cirq.Simulator()
resultado = simulador.run(circuito, repetitions=1)
print(resultado)
```

Resultado de la ejecución:




```
q(1)=0
q(3)=0
q(5)=1
q(7)=1
q(9)=0
q(11)=1
```

Compruébese que 11010 + 11101 + 1 = 101100.

Referencias

[1] J.P.Deschamps, Computación Cuántica, Marcombo, Barcelona, 2023.

[2] V. Vedral, A. Barenco, and A. Ekert, “Quantum networks for elementary arithmetic operations,” *Phys. Rev. A*, vol. 54, no. 1, p. 147, 1996.

[3] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, “A new quantum ripple-carry addition circuit,” 2004, *arXiv:quant-ph/0410184*.

[4] J.P.Deschamps, G.J.Bioul, and G.D.Sutter, « Synthesis of Arithmetic Circuits », *Wiley¸* 2006.