# NOTE SULL'ANALISI DI STRINGHE NEL MODELLO QUANTISTICO

Gian Maria Alvau

19/09/2024

# Indice

# Processori Quantistici

Un processore quantistico basa sui principi della meccanica quantistica per compiere operazioni di calcolo estremamente complesse, con una velocità e un'efficienza impossibili da raggiungere con i processori tradizionali. Mentre i computer classici utilizzano bit, che possono assumere solo due stati, 0 o 1, i processori quantistici si basano su unità di informazione chiamate qubit. A differenza dei bit classici, i qubit possono trovarsi in una sovrapposizione di stati, cioè possono essere simultaneamente 0 e 1. Questo fenomeno consente a un processore quantistico di esplorare molteplici soluzioni di un problema in parallelo, aumentando esponenzialmente la sua capacità di elaborazione rispetto a un calcolatore tradizionale.

Un altro aspetto straordinario di un processore quantistico è il fenomeno dell'entanglement, per cui due o più qubit possono essere correlati tra loro in modo tale che lo stato di uno influenzi immediatamente lo stato dell'altro, indipendentemente dalla distanza che li separa. Questo tipo di connessione permette una comunicazione interna estremamente rapida e coordinata all'interno del processore. Inoltre, i processori quantistici sfruttano l'interferenza quantistica, utilizzando meccanismi di amplificazione o cancellazione di probabilità per enfatizzare le soluzioni corrette di un calcolo e attenuare quelle errate.

Il potenziale di un processore quantistico risiede nel fatto che, per certe categorie di problemi, come la fattorizzazione di numeri molto grandi o l'ottimizzazione di sistemi complessi, può essere esponenzialmente più efficiente rispetto ai processori classici. Ad esempio, le attuali tecnologie di crittografia basate sulla difficoltà di fattorizzare numeri possono essere messe a rischio con l'avvento dei computer quantistici, che potrebbero eseguire tali operazioni in tempi drasticamente inferiori. Tuttavia, questa tecnologia non è priva di sfide. I qubit sono estremamente fragili e suscettibili a interazioni indesiderate con l'ambiente esterno, un fenomeno noto come decoerenza, che può distruggere l'informazione quantistica e compromettere il calcolo. Per questo, è necessario sviluppare tecniche avanzate di correzione degli errori per garantire risultati affidabili. Inoltre, la scalabilità dei processori quantistici è un obiettivo ancora lontano: mentre attualmente è possibile costruire dispositivi con poche centinaia di qubit, il passaggio a processori con migliaia o milioni di qubit stabili è ancora un grande ostacolo tecnologico.

I processori quantistici esistono in diverse architetture, tra cui quelle basate su circuiti superconduttori, come quelli utilizzati da aziende come IBM e Google, o quelle che utilizzano trappole di ioni o fotoni per rappresentare i qubit.

## Processori a superconduttori

I processori quantistici a superconduttori sfruttano un fenomeno noto come superconduttività, che si manifesta in determinati materiali quando vengono raffreddati al di sotto di una temperatura critica. A questa temperatura, i materiali perdono completamente la loro resistenza elettrica, permettendo alla corrente di fluire senza alcuna dissipazione di energia.

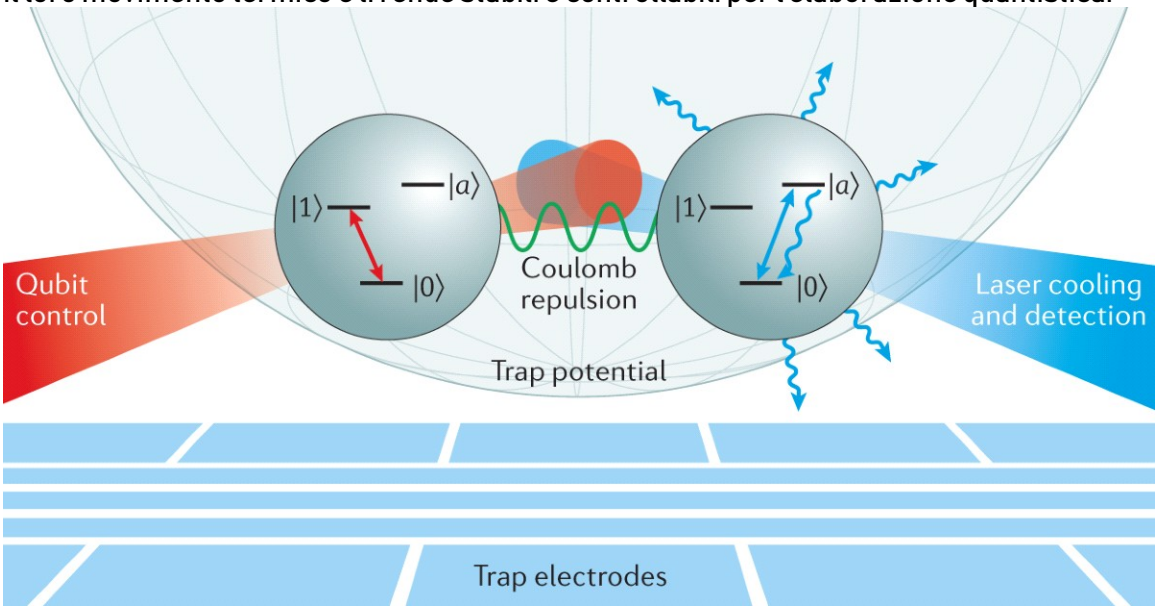
Il cuore di un processore quantistico a superconduttori è il qubit superconduttivo, spesso basato su un dispositivo chiamato Josephson junction. Questo componente sfrutta la superconduttività per creare una barriera che permette agli elettroni di attraversare il circuito senza resistenza in maniera quantistica. I qubit superconduttivi possono essere manipolati con impulsi di microonde o segnali magnetici.

I processori quantistici a superconduttori si distinguono per la loro capacità di operare a frequenze estremamente alte, il che consente una velocità di esecuzione eccezionale nelle operazioni logiche. Grazie alla loro superconduttività, possono minimizzare la dissipazione di energia, riducendo drasticamente il consumo energetico rispetto ai processori classici. Tuttavia, come in tutti i sistemi basati sulla superconduttività, devono essere mantenuti a temperature molto basse, prossime allo zero assoluto, per mantenere le loro proprietà quantistiche. Questo comporta l'utilizzo di sofisticati sistemi di raffreddamento, che rappresentano una delle maggiori sfide tecniche nella costruzione di questi processori.

## Processori a ioni intrappolati

I processori a ioni intrappolati si basano sull'utilizzo di campi magnetici per intrappolare singoli ioni che svolgono al funzione di qubit. Questi ioni vengono intrappolati e manipolati in una trappola

elettromagnetica, come la trappola di Paul, che utilizza campi elettrici oscillanti per confinare e mantenere gli ioni in una posizione fissa nello spazio. Una volta intrappolati, gli ioni vengono raffreddati quasi fino allo zero assoluto utilizzando tecniche di raffreddamento laser, il che riduce il loro movimento termico e li rende stabili e controllabili per l'elaborazione quantistica.




Il funzionamento dei processori quantistici a ioni intrappolati si basa sulla capacità di manipolare questi ioni attraverso impulsi laser per eseguire operazioni quantistiche. I qubit sono codificati negli stati interni degli ioni, tipicamente negli stati elettronici di due livelli. Gli impulsi laser vengono utilizzati per eccitare o far cambiare lo stato degli ioni, permettendo loro di passare tra lo stato di 0 e 1, o di creare una sovrapposizione quantistica di entrambi gli stati simultaneamente. Inoltre, utilizzando interazioni controllate tra i singoli ioni e il campo laser, è possibile creare entanglement tra gli ioni, che è fondamentale per eseguire operazioni quantistiche complesse e correlare i qubit tra loro.

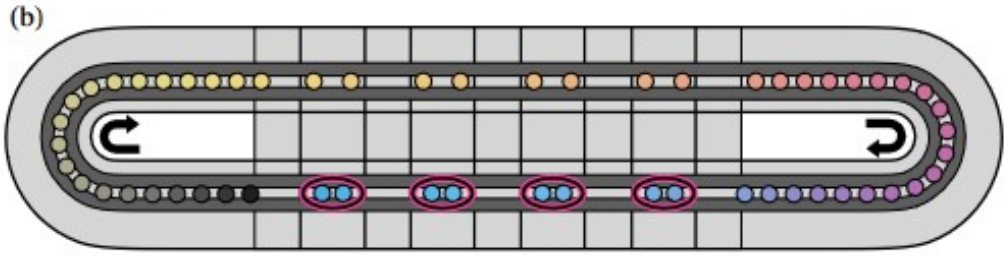
Una delle caratteristiche distintive dei processori a ioni intrappolati è la loro elevata fedeltà nelle operazioni quantistiche, grazie alla stabilità intrinseca degli ioni e alla precisione con cui possono essere manipolati. Poiché gli ioni sono particelle elementari con stati quantistici ben definiti, essi offrono un alto grado di coerenza, mantenendo le loro proprietà quantistiche per tempi più lunghi rispetto ad altre tecnologie quantistiche. Questo rende i processori a ioni intrappolati particolarmente adatti per operazioni di lunga durata e per eseguire algoritmi quantistici che richiedono una grande precisione.

# H1

H1 è il primo modello di processore quantistico sviluppato da Quantinuum, caratterizzato da una struttura lineare, ovvero in cui gli ioni sono trasportati su una singola rotaia. Il processore utilizza atomi di Itterbio che viene ionizzato tramite un laser. La possibilità di spostamento di tutti i 20 ioni disponibili lungo la rotaia rende i qubit completamente connessi, riducendo la difficoltà di

progettazione ed il rumore dovuti agli swap necessari per l'interazione tra qubit che si hanno in un modello a superconduttori. La possibilità di spostamento non esclude la possibilità di errore, anche se i due ioni su cui viene applicata una two-qubit gate sono identici ci potrà essere una differenza d'errore data dalla posizione in cui viene effettuata l'operazione. Il modello H1 permette misurazioni mid-circuit ed il riutilizzo di qubit, con una single-qubit gate fidelity del 99,99% ed una two-qubit gate fidelity del 99,97%.

Il modello successivo, H2,  è basato su architettura chiamata "racetrack" in cui gli ioni sono mossi in una rotaia ad anello con 56 qubit. L'H2 raggiunge valori di single-qubit fidelity del 99,997% e two-qubit fidelity di 99,87%.




## API e risorse

Sia il modello H1 che i modelli successivi dei processori di Quantinuum possono ricevere codice python con Tket o con Qiskit.

Tket è un toolkit compreso di compilatore sviluppata da Quantinuum basata sul modello qasm2, utilizzabile con il modulo pytket da Python3.10 in poi, che permette di creare e visualizzare circuiti quantistici facilmente ed è in grado di transpilare circuiti creati in qiskit o qasm2.

Per interfacciarsi con i servizi di Quantinuum ci sono vari metodi: avere un account Quantinuum a pagamento, tramite l'Oak Ridge National Laboratory (USA), con una licenza InQuanto oppure tramite Azure con Azure Quantum. Azure risulta la modalità migliore di accesso poichè fornisce 500$ di crediti per vari processori quantistici anche con un account gratuito per scopi didattici o di ricerca e consente di ottenere fino a 10000$ in crediti Azure Quantum tramite l'Azure Quantum Credit program. Altri crediti possono essere acquistati tramite Azure.

La creazione del workspace di Azure Quantum permette ad altre persone autorizzate di usufruire dei crediti e delle risorse disponibili, oltre al monitoraggio dei crediti e dei job inviati.

Per inviare il proprio codice direttamente da Azure è necessario utilizzare Notebook Q#, un linguaggio di alto livello sviluppato da Microsoft per il quantum programming che supporta codice Python e Qiskit, che possono essere caricati ed eseguiti direttamente sulla piattaforma di Azure.

In alternativa si può utilizzare VSCode per scrivere codice senza l'utilizzo di Q#(utilizzato solo per settare il profilo di esecuzione) con le API di azure e, rendendo possibile lanciare codice dall'IDE tramite il nostro workspace utilizzando solo Python, Pytket e, se necessario, Qiskit.

Sono forniti 4 backend diversi: un simulatore offline, un syntax checker, vari simulatori online che rispecchiano il processore specifico selezionato e l'invio all'hardware. Solo l'invio all'hardware consumerà crediti.

## Invio al processore

Dato che l'utilizzo del modello H2 è possibile solo a pagamento, è stato utilizzato solo il modello H1 per l'esecuzione del codice.

Per entrambi i programmi i circuiti sono stati creati con Qiskit, poichè il backend di Quantinuum effettua automaticamente la transpilazione dei circuiti in qasm2 prima dell'esecuzione. Questo processo è problematico nel caso si utilizzi il backend offline per i circuiti con gate di rotazione creati, che vengono tradotti male da qiskit a qasm2, motivo per cui non è possibile eseguire offline il codice per la rotazione di stringhe.

Il simulatore H1-1 è stato utilizzato per l'analisi della stringa di dimensioni maggiori possibilied il confronto dei risultati. Il simulatore utilizza un modello di noise ad alta fedeltà del modello H1, mantenendo la connettività completa  ed il riutilizzo di qubit.

# Stringhe Palindrome

## Spiegazione del codice

```
# Job definition
def palindrome():


        x = [1, 1 , 0, 0, 1, 1]

        size = len(x)

        # Definire i registri quantistici e classici
        qx = QuantumRegister(size, name='string_x')
        q_results = QuantumRegister(size // 2, name='output')
        qbit = QuantumRegister(1)
        cr = ClassicalRegister(1)

        qc = QuantumCircuit(qx, q_results, qbit, cr)

        # Inizializzare i qubit con i valori di x
        for i in range(size):
                if x[i] == 1:
                        qc.x(qx[i])
        qc.barrier()


        for i in range(size // 2):
                qc.ccx(qx[i], qx[size - 1 - i], q_results[i])  # Use CCX to copy parity check result

        qc.barrier()

        for i in range(size):
                qc.x(qx[i])

        qc.barrier()
```

Lo scopo di questo codice è verificare se una stringa si palindroma, per cui prendiamo una stringa di dimensione di 6 per verificare questa proprietà.

Creiamo un registro quantistico delle stesse dimensioni della stringa (qx) che la rappresenterà ed un registro di appoggio di dimensione equivalente alla metà della stringa (q_results). Questo sarà utilizzato per il conteggio della parità tra i qubit. Creiamo infine un ultimo registro quantistico di un solo qubit (qbit) ed un registro classico di un bit per la misurazione (cr). Creiamo dunque il circuito qc con i tre registri quantistici ed il registro classico.

```
    for i in range(size // 2):
        qc.ccx(qx[i], qx[size - 1 - i], q_results[i])  # Use CCX to copy parity check result

    qc.barrier()

    for i in range(size):
        qc.x(qx[i])

    qc.barrier()

    for i in range(size // 2):
        qc.ccx(qx[i], qx[size - 1 - i], q_results[i])  # Use CCX to copy parity check result

    qc.barrier()

    for i in range(size):
        qc.x(qx[i])


    #for i in range(size//2):
    #     qc.x(q_results[i])
    # Aggiungere la porta MCX
    mcx_gate = MCXGate(size // 2)
    qc.append(mcx_gate, q_results[:] + qbit[:])

    # Misurare il qubit
    qc.measure(qbit[0], cr[0])

    # Visualizzare il circuito
    print(qc.draw())
```

Inizializzamo i qubit del registro qx con i valori corretti utilizzando una porta not

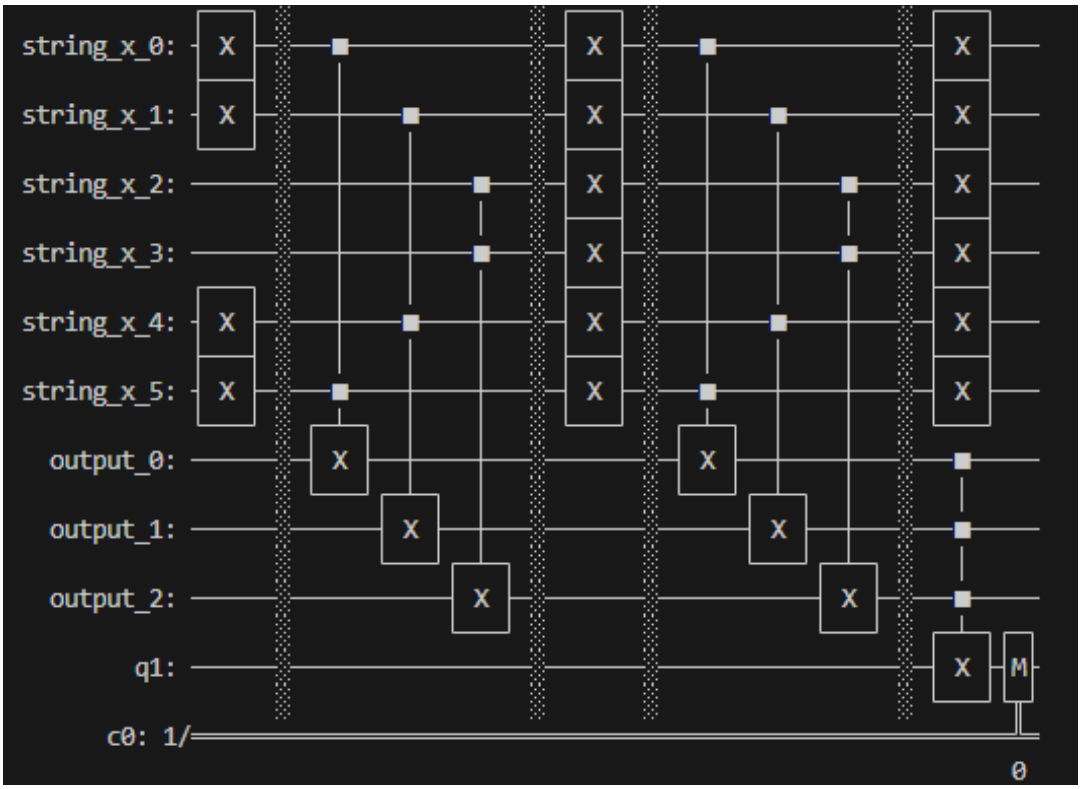
Controlliamo con una Toffoli Gate(CCX) che i qubit di qx, in coppia dai più esterni ai più interni (il primo qubit con l'ultimo qubit, il secondo qubit con il penultimo qubit, ecc.), siano entrambi settati a 1 e settiamo il qubit corrispondente alla coppia in q_results a 1.

Con una porta NOT applicata a tutto il registro qx otteniamo la stringa opposta a quella iniziale, in cui gli zeri ora valgono uno e gli uni valgono zero.

Ripetiamo la Toffoli Gate, che ora controllerà che i bit posti a zero nella stringa originale siano uguali a coppie dai più esterni ai più interni e varierà i qubit del registro q_results in modo appropriato.

Il registro q_results conterrà ora il conteggio di qubit uguali a coppie.

Andiamo poi a creare una porta MCX (MultiControlledNot), che varierà il valore del registro qubit solo nel caso in cui tutti i qubuit del registro q_results siano posti a 1, ovvero nel caso in cui tutti i bit della stringa originale siano uguali a coppie e la stringa sia palindroma. Infine misuriamo il valore del registro qbit e mostriamo il circuito.




# Risultati H1

Eseguito sul processore H1 di Quantinuum con 100 shots otteniamo i seguenti risultati:

```
Job id: 3eaf3dd8-728f-11ef-81f4-d43b048f8599
...........{'0': 2, '1': 98}
```

Ovvero la stringa viene riconosciuta correttamente come palindroma 98 volte su 100 ed erroneamente come non palindroma 2 volte su 100.

Eseguendo lo stesso codice sul simulatore H1-1 otteniamo un risultato molto simile : `{'1': 97, '0': 3}`, con un solo errore in più.

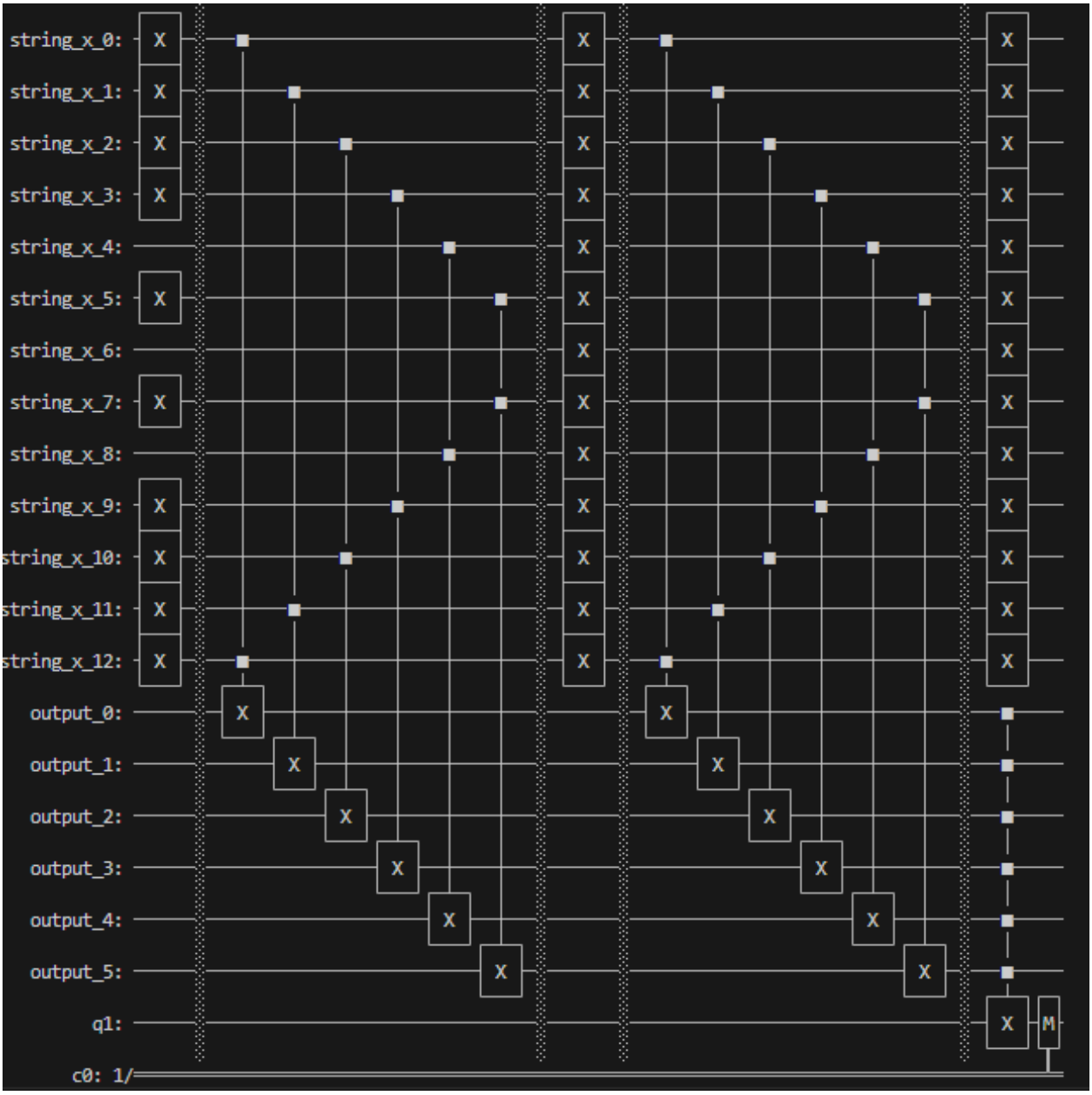
# Stringa massima

Il processore H1 ha 20 qubit disponibili, quindi la massima stringa che può essere analizzata da questo processore è di dimensione 12, poiché abbiamo bisogno di 12 qubit per rappresentarla, 6 qubit per il controllo della parità dei bit ed 1 qubit di controllo. Usiamo quindi 19 qubit per analizzare una stringa di 12 bit.

Analizziamo la seguente stringa sul simulatore H1-1

```
x = [1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1 ,1 ,1]
```

Otteniamo il circuito




Che fornisce come risultato

```
Job id: 24154d36-7292-11ef-80cd-d43b048f8599
.............{'1': 96, '0': 4}
```

, che indica che la stringa viene riconosciuta correttamente come palindroma 96 volte su 100 ed erroneamente come non palindroma 4 volte su 100.

La percentuale di errore risulta dunque raddoppiata, da 2% a 4%, su una stringa di dimensione doppia a quella originale. Si potrebbe pensare che la percentuale di errori sia legata al numero di qubit analizzati e numero o tipo di operazioni effettuate che renderebbero il sistema più tendente all'errore, ma sarebbe necessario uno studio più approfondito per vedere una possibile relazione

tra i dati.

# Rotazione di Stringhe

## Spiegazione codice

```
encoding = 1
anc = 2
alphabet = ['0', '1']
sigma = len(alphabet)
dim = 2
string = [1] + [0] * (dim-1)
n = len(string)
target_qubits = encoding*n
control_qubits = math.ceil(np.log2(n))
ancilla_qubits = anc
ancilla_bits = anc*n
block_size = int(np.log2(sigma))  # Esempio, usando un alfabeto di dimensione 4

control_register = QuantumCircuit(control_qubits, name='control')
target_register = QuantumRegister(target_qubits, name='target')
ancilla_register = QuantumRegister(ancilla_qubits, name='ancilla')
target_indices = list(range(control_qubits, target_qubits + control_qubits))
control_indices = list(range(control_qubits))
```

Lo scopo di questo codice è, presa in esame una stringa di dimensione n, ottenere tutte le rotazioni possibili di questa tramite il principio di sovrapposizione quantistica. Questo compito si è mostrato difficoltoso per i modelli a superconduttori a causa dell'insicurezza sulla mappatura fisica dei dati sul processore e a causa del rumore tipico delle operazioni di swap tra qubit di questo tipo.

Encoding e gli ancilla bits e qubits vengono utilizzati per il codice di error correction che tralascerò poichè non facenti parte del mio compito,  anche se dovrebbe essere direttamente utilizzabile anche sull'hardware H1 senza bisogno di traduzione manuale.  Encoding, nello specifico, rappresenta in quanti bit viene rappresentato uno stesso valore del nostro alfabeto per motivi di ridondanza.

Alphabet rappresenta l'alfabeto utilizzato, in questo caso quello binario composto da 0 e 1. N è la dimensione della stringa presa in esame.

Il registro dei qubit target viene creato in base alla dimensione della stringa e al codice di correzione degli errori, che richiede una codifica in più qubit della stringa (encoding>1 significherebbe che ogni bit è rappresentato su più qubit), e quindi richiederebbe molti qubit per la rappresentazione, il che sarebbe molto dispendioso sull'hardware di Quantinuum.

Il numero di qubits di controllo sarà il logaritmo in base 2 della dimensione dell'alfabeto .

Block_size conterrà il numero di qubit necessari per rappresentare un elemento del nostro alfabeto, in questo caso 1.

Creiamo i registri quantistici ed il circuito

```
ancilla = QuantumCircuit(anc)
# Crea un circuito combinato con una dimensione totale di n + log2(n) qubit
qc = QuantumCircuit(control_qubits)
qc.add_register(target_register)
qc.add_register(ancilla_register)
```

```
#sovrapposizione dei qbit di controllo
for j in range(control_qubits):
    qc.h(j)

#qc.x(3)
#qc.x(0)


#codifico la stringa nei qbit target
temp = 0
for i in range(n):
    if string[i] == 1:
        for j in range(encoding):
            qc.x(target_register[temp+j])
    temp += encoding
        #qc.x(target_register[i+1])
        #qc.x(target_register[i+2])



qc.barrier()
control_indices = 0
for c_qbits in range(0, control_qubits):
    k = 2**c_qbits
    my_crot_gate = lcs.crot_qec(target_qubits, k, block_size, encoding)
    qc.append(my_crot_gate, [c_qbits] + target_indices)
```

Applichiamo un Hadamard Gate ai qubit di controllo per metterli in sovrapposizione e poi applichiamo un gate not ai qubit che corrispondono ai bit posti a 1 della stringa per inizializzarli. Nel caso in cui il singolo valore fosse rappresentato su più qubits, ciclando su encoding applicherremo il gate not a tutti i qubit che rappresentano quel valore.

Prima di spiegare cosa avviene nel ciclo for alla fine di quesa porzione di codice vediamo la funzione crot_qec contenuta nel file lcs.py che crea i gate crot.

```python
def crot_qec(n, k, block_size, encoding):
    # Creiamo il gate di rotazione come prima
    rot_gate = rot_qec(n, k, block_size, encoding)

    # Aggiungiamo un qubit di controllo al gate per renderlo controllato
    c_rot_gate = rot_gate.control(1)

    return c_rot_gate
```

```python
#Rotation gate (not controlled) -> rot
def rot_qec(n, k, block_size, encoding): # n*block_size vs n, block_size
    qc = QuantumCircuit(n, name=f'rot_k={k}')
    n=n//encoding
    stop = (int(np.log2(n)) - int(np.log2(k*block_size)) + 2)
    #print("stop = ", stop)
    for i in range(block_size, stop):
        #print("\nnumero = ", int(n/(k*(2**i))))
        for j in range(0, int(n/(k*(2**i)))):
            for x in range(j*k*(2**i), k*((j*2**i+1))):
                for offset in range(0, block_size):
                    inizio_swap = x + k*offset
                    fine_swap = x + 2**(i-1)*k + k*offset
                    for b in range(encoding):
                        qc.swap(inizio_swap*encoding + b , fine_swap*encoding + b
                        print("\n inizio swap = ", inizio_swap)
                        print("\n fine swap = ", fine_swap)


                        #print("\ninizio_swap + b", inizio_swap + b)
                        #print("\nfine_swap + b", fine_swap + b)
                    #qc.swap(inizio_swap + 1, fine_swap + 1)
                    #qc.swap(inizio_swap + 2, fine_swap + 2)

    print(qc)
    rot_gate = qc.to_gate()
    return rot_gate
```

La funzione rot_qec prende in input i qubit target su cui deve agire, la variabile k che rappresenta la distanza dei qubit per la rotazione, block size ed encoding, per eseguire le rotazioni. Non descriverò gli indici scelti per le rotazioni poichè derivano da un paper pubblicato dal Professor Faro.

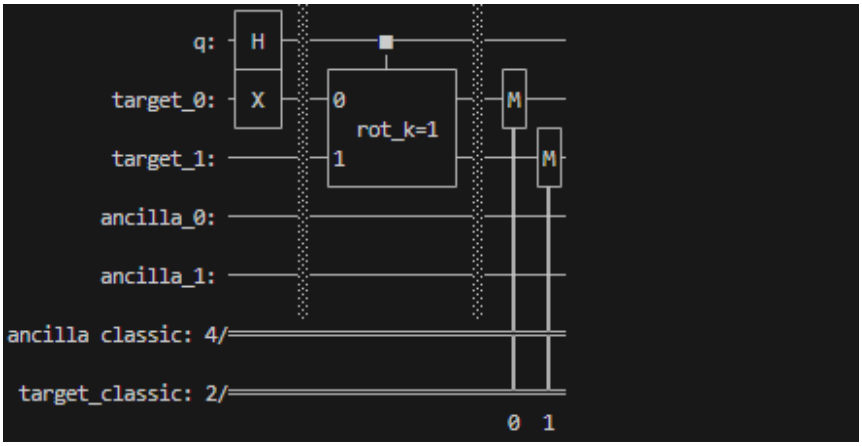
Infine viene creato il gate custom di rotazione, la funzione crot_qec aggiunge un bit di controllo ed il gate viene inserito nel circuito iniziale.

```
###############misura#####################################################
qc.barrier()
classic_register = ClassicalRegister(target_qubits, name='target_classic')
qc.add_register(classic_register)

qc.measure(target_register, classic_register)
#for i in range(target_qubits):
    #qc.measure(control_qubits + i, i)
depth = qc.depth()
print("\ndepth = ", depth)
```

Infine aggiungiamo il registro di bit classici al circuito e effettuiamo la misurazione.

Otteniamo dunque il seguente circuito:




# Risultati H1

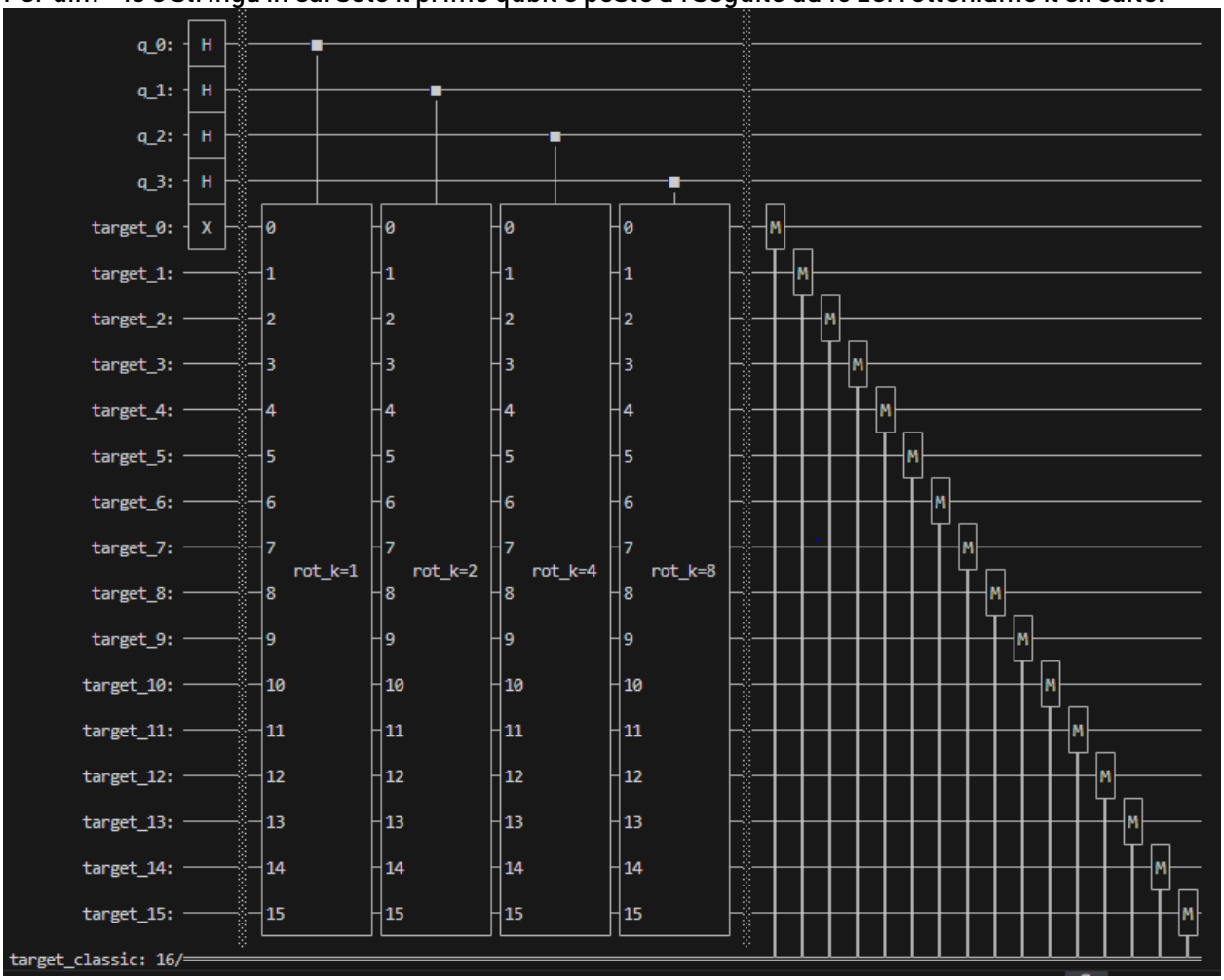
Questo circuito è stato eseguito con la stringa '10' come input sul processore H1 per 1000 shots e ha restituito come risultato `{'11': 8, '01': 485, '10': 499, '00': 8}` ovvero 16 errori (8 volte 11 + 8 volte 00) che non sono rotazioni della stringa iniziale e 984 volte un risultato corretto. Notiamo che per 499 volte è stata restituita la stringa di partenza ed essendo la rotazione dettata da un singolo bit in sovrapposizione è poco lontano dal risultato teorico aspettato di '01':500, '10':500 .

Eseguendo lo stesso circuito sul simulatore H1-1 otteniamo un risultato molto simile , sempre con 16 errori.`{'01': 508, '10': 476, '00': 8, '11': 8}`

# Stringa Massima

La dimensione massima della stringa per il processore H1 è 16 qubit +4 qubit di controllo se ,non utilizzando l'error correction, rimuovessimo i qubit ancilla.

Per dim = 16 e stringa in cui solo il primo qubit è posto a 1 seguito da 15 zeri otteniamo il circuito:




Questo circuito è stato eseguito sul simulatore H1-1 e restituisce un output molto difficile da leggere, di cui inserisco solo una porzione


```
Job id: f48243f6-7601-11ef-a1e6-d43b048f8599
.....................................................{'0001100010100000': 1, '1001000100010000': 1, '0000000101010000': 1, '1000001100000000': 1, '1100010000000101': 1
, '0000000101000010': 1, '0000000111010001': 1, '0000110000010000': 1, '1111111001111011': 1, '0101001000000000': 1, '0001001100000010': 1, '1000001000000100': 1, '
0100000000100001': 1, '0001010000000010': 1, '0000000000101000': 3, '0000100000000001': 2, '0000000111000001': 1, '0000000100000001': 2, '0000000010000101': 1, '000
1000101000000': 2, '0101000000000000': 6, '1100000001000000': 1, '1000000000000010': 1, '1011000110110000': 1, '0000010001000001': 1, '0000011100000001': 1, '000010
0000001000': 4, '0100111101001001': 1, '0000001000000000': 13, '0000010000100010': 1, '1001000100010001': 1, '0000011110111000': 1, '0000000000000101': 7, '01011100
00100100': 1, '0000100010001001': 1, '1000001000001000': 1, '0010010000100010': 1, '0100000010100000': 1, '0000001110000000': 3, '0001000100010000': 2, '00000001000
00111': 1, '1101000010000000': 1, '1101000001010001': 1, '0000000000100110': 3, '1000000001010001': 1, '0100110000101101': 1, '0000000010000010': 1, '1110001101000001
01': 1, '0001010100000110': 1, '0110000100000001': 1, '0010000100000000': 2, '0000000000001101': 1, '0010010001000001': 1, '0000100110000000': 1, '0000000000000011'
: 2, '0111000000000011': 1, '0100011100000000': 1, '0000010000100100': 1, '0000000011001100': 1, '0000001010010100': 1, '0011000000100010': 1, '1100101000010110': 1
, '0000000000010000': 18, '0000000000110000': 3, '0000000010010100': 2, '0000001110010000': 1, '0000001110100000': 2, '0100001000000110': 1, '0010001000001000': 1,
'0101000000110010': 1, '0010000000000000': 12, '0001000000001010': 1, '1101000000000000': 2, '0010010000000000': 3, '0110000000000010': 1, '0010001100000000': 1, '1
1010000000000001': 2, '0000001011010110': 1, '0010000110000000': 1, '1000000000001010': 1, '0000000000100100': 3, '0000000000010110': 1, '0110100000100100': 1, '1000
000110010000': 1, '1010001000000110': 1, '0000001100000100': 2, '0000000000001000': 9, '0010000000000010': 5, '0101000000000011': 1, '0110000010000000': 1, '0001100
000010000': 1, '1001001000000000': 1, '1001000001000001': 1, '0001001100000001': 1, '0010011000001000': 1, '0100000001010000': 1, '1101100000000000': 2, '1001000000
000100': 1, '0000001100000001': 1, '0001011100000110': 1, '0100000001000000': 1, '0000000010010001': 1, '1100011000000100': 1, '0000000010010101': 1, '1100001000000
011': 1, '1010000000000010': 1, '1000100000010000': 1, '0000001100100001': 1, '0010001000000100': 2, '1101000100000001': 1, '0010000001000010': 1, '0000000000000000
': 6, '0000000000010001': 5, '0100000000100000': 1, '0101000100000110': 1, '0110000101000000': 1, '0000000100000101': 1, '0000000101000000': 10, '0000000000011100':
 2, '0100001100000000': 1, '0000100000010100': 1, '0000001000000010': 3, '0001001100010000': 1, '0000000000010100': 8, '0000000101010000': 2, '0000001010111000': 1,
 '1000000000010000': 1, '0001010000010000': 3, '0000000001000001': 1, '0001010010000000': 1, '0011000100000001': 1, '0000000000100011': 1, '1101000010000000': 1, '0
000001000000011': 2, '0000000101100000': 3, '0000101000001010': 1, '0010000010000000': 1, '0110000010110001': 1, '0010001101100000': 1, '0001010000001010': 1, '1000
```


Il risultato viene restituito sotto forma di dizionario, quindi effettuando la somma dei valori per cui la chiave contiene un solo '1' otteniamo

```
Risultato corretto 199 volte, errato 801, stringhe corrette = 16
```

Con stringhe corrette intendiamo stringhe in cui il valore '1' appare solo una volta e che sono,

dunque, rotazioni della stringa iniziale. Per 9 istanze è stata restituita la stringa inziale stessa, quindi meno dell'1% dei casi, anziché il circa 6% aspettato (4 hadamard gate su qubit singolo -> 1/16 che siano tutti a 0). Il numero di errori risulta molto elevato, rappresentando l'80% dei casi.