Introduction

Who shoud read this tutorial?

Anyone who is interested in knowing what Quantum Computing is about or is new to Quantum Computing.

Requirements:

What should I know?

Knowledge of high school Mathematics should be enough. Also, this course is in Julia, so knowing basic syntax of Julia might help... but knowing Julia syntax is not necessary. Even if you know the basics of Python, R or Ruby, and how to write programs in them, you should be fine.

The required software :-

- **Julia**
- **Yao.jl**

Install Julia on your system if you haven't already. This tutorial assumes you already have Julia and you opened this notebook in your system.

Quantum

Now recently, if you're in any way affiliated to Science or Computer Science or even if you're just a tech lover, you must have stumbled across the word "*quantum*". You must have wondered, " What does it mean?" .

So, let's just say that historically, some experiments were conducted by some scientists in Physics, and their results happened to defy what laws of Physics stated, before then. Now this was around the year 1900. The scientists came up with theories that explained those results and the most widely accepted (and working) theory was quantum mechanics, or in Einstein's words, "*Real Black Magic Calculus*". On the basis of quantum mechanics, we now have Quantum Physics, Quantum Chemistry, Quantum Biology, Quantum Computers and lots of other new stuff. The quantum computers just exploit some weird phenomenons of quantum mechanics.

Quantum Mechanics will not be explained in detail, in this tutorial. Only the part required for

computation. If you're interested in Quantum Mechanics, **Modern Quantum Mechanics by J. J. Sakurai** might be a good read.

Bits

Okay, what makes quantum computers different than a present day or *classical* one. To understand that, we must understand a few things about classical computers.

"*A bit (short for binary digit) is the smallest unit of data in a computer*". You must have read this in a book or article about computers. Computers use bits, represented by the digits 0 and 1, to store data. We organize these bits to store information and manipulate these bits and perform operations on them to get a variety of things done. Like storing numbers, or images or videos. Adding numbers, subtracting numbers. How? Well... Consider storing a number. In this case, the collection of bits 1 0 1 1 can be considered as, $2^3 + 0 + 2^1 + 2^0 = 11$. The genereal idea being, the summation of $2^{(position\ of\ 1s (starting\ from\ 0)\ from\ right)}$.

Then just divide your computer screen into a matrix of 1000s of cells, and every cell containing a number, corresponding to the colour in that cell. Yeah, images are stored that way, those numbers are called pixels. Addition looks a bit more complicated than this. It's done using something called 'gates'. Gates take one or more bits and perform a logical operation on them to give a certain output. There's an "AND gate" which, takes in 2 bits and multiplies them to give an output. There're more gates like the OR gate, XOR gate etc. These gates are arranged in a certain manner to perform feats like addition, subtraction, etc.

Qubits

Like the *bits* of a classical computer, quantum computers have their own fundamental unit of data called, a *qubit*.

Qubits can make use of some quantum mechanical properties like superposition. For the sake of understanding how qubits work, imagine the quantum computer as a box, full of qubits. Now these qubits are objects which have some mathematical and physical properties, and can store data and can be used to manipulate data to get some computation done.

Bits are represented by two states, 0 and 1. At any time, a qubit is in a *superposition* of two states, represented by $a|0\rangle + b|1\rangle$. When we *measure* a qubit, its state *collapses* to, either $|0\rangle$ or $|1\rangle$. The chances(probability) of a qubit collapsing to the state, $|0\rangle$ is a^2 and to $|1\rangle$ is b^2. Hence, it must be satisfy $|a^2| + |b^2| = 1$. a and b are also known as *probability amplitudes* of a given qubit.

Note : The notation of $|\ \rangle$ and $\langle\ |$ are known as Dirac's notation, or the bra-ket notation. For this tutorial, you need to understand that column vectors(matrices of size n × 1) are called kets and are represented by $|\ \rangle$, and row vectors(matrices of size 1 × n) are called bras, and are represented by $\langle\ |$.

Working with qubits

So what do the terms, superposition and measurement, mean...? Well, to keep it simple, superposition of two or more states just means their linear combination. The superposition of states $|0\rangle$ and $|1\rangle$ just means a linear combination of $|0\rangle$ and $|1\rangle$. If you still don't get it, no problem! Your intuition about superposition will build up by the time this tutorial reaches multiple qubits(probably).

When we use terms like measurement, we refer to *looking* at the state of the qubit. And yeah, we can't know the state of the qubit, because just when we look at it, it's state changes to $|0\rangle$ or $|1\rangle$, and all the information about that qubit is lost. So the state of the qubit(the values of a and b) can't be determined.

Quantum Gates and Circuits

So consider that the state of a qubit is, $a|0\rangle + b|1\rangle$. We want to do something with this..... say, we want to change the state to $a|1\rangle + b|0\rangle$... how do we manipulate the qubits or perform any operation on them?

We use quantum gates, the building blocks of quantum circuit, to manipulate qubits to get some task done.

Note: For the information that follows, please note that we can represent a qubit or a system of qubits with their state vector. Consider the qubit, $a|0\rangle + b|1\rangle$, we can represent this as $\begin{bmatrix} a \\ b \end{bmatrix}$.

Single qubit gates

As the name suggests, these gates take one qubit for an input, and give a qubit with changed state for an output.

1. The X gate

It changes the state of the qubit from $a|0\rangle + b|1\rangle$ to $a|1\rangle + b|0\rangle$. It *flips* the state of the qubit. Mathematically, its represented by $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, and applying this gate to a qubit is mathematically equivalent to multiplying the vector representing the qubit to the above matrix. It looks somewhat like this, when implemented in a circuit.

2. The Y gate

It changes the state of the qubit from $a|0\rangle + b|1\rangle$, to $b|0\rangle - a|1\rangle$. It does a *bit flip* and a *phase flip* at the same time. The Y gate is mathematically represented by the matrix, $i\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$. Mathematically, passing a qubit through this gate is equivalent to multiplying the state of the qubit, i.e., $\begin{bmatrix} a \\ b \end{bmatrix}$, to the above matrix. It's represented in a circuit by

3. The Z gate

It changes the state of the qubit from $a|0\rangle + b|1\rangle$, to $a|0\rangle - b|1\rangle$. It does a *sign flip* on the qubit. The Z gate is mathematically represented by the matrix, $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$. Mathematically, passing a qubit through this gate is equivalent to multiplying the state of the qubit, i.e., $\begin{bmatrix} a \\ b \end{bmatrix}$, to the above matrix. It looks like this when implemented in a circuit.

Note: *The matrix representation of the above three gates are known as Pauli's matrices, represented by* $\sigma_x,\ \sigma_y\ and\ \sigma_z$, *for the X gate, the Y gate and the Z gate, respectively*

4. The H gate

When a qubit is passed through H gate, the $|0\rangle$ changes to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, and the $|1\rangle$ changes to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

For an example, $a|0\rangle + b|1\rangle$, changes to, $\frac{a}{\sqrt{2}}(|0\rangle + |1\rangle) + \frac{b}{\sqrt{2}}(|0\rangle - |1\rangle)$. It can be simplified to give, $\frac{a+b}{\sqrt{2}}\ |0\rangle + \frac{a-b}{\sqrt{2}}\ |1\rangle$. Mathematically, its represented by the matrix $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$ and passing a qubit through the H gate is mathematically equivalent to multiplying the vector representing the state of the qubit, to the above matrix. In a circuit, the H gate is represented by

Note: *The state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is often called $|+\rangle$, and the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ is often called $|-\rangle$.*

Multiqubit Gates

As the title suggests, these gates takes in and operate on more than one qubits.

The CNOT gate

It takes two qubits as an input, and if the first qubit is a $|1\rangle$ the state of the second qubit is flipped to $|0\rangle$ if it was $|1\rangle$, and $|0\rangle$ if it was $|1\rangle$. If the state of the first qubit is $|0\rangle$, no change is made to the second qubit. Mathematically, its represented by the matrix $\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}$, and passing two qubits through it is equivalent to first collecting the vector of two two qubits into one and multiplying it to the above matrix. In a circuit, the CNOT or the **CX** gate is represented by

Note: *To represent a system of two qubits, $a_1|00\rangle + a_2|01\rangle + a_3|10\rangle + a_4|11\rangle$ in vector form, we can write them as, $\begin{bmatrix}a_1\\ a_2\\ a_3\\ a_4\end{bmatrix}$, where $|a_1^2| + |a_2^2| + |a_3^2| + |a_4^2| = 1$. Also, we can use the CNOT or Control NOT or any other control gate(CY and CZ) to entangle two qubits.*

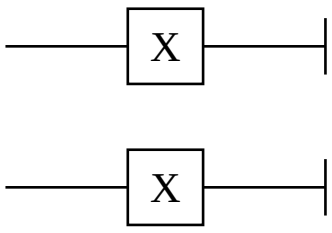
Using Yao - The basics of quantum computing in Julia using Yao.jl

At the current moment, we don't have have quantum computers. How do we make quantum circuits then? Well, two things we can do right now are, simulate a few qubits or use the qubits created by corporates like IBM and D'Wave. Using Yao, we can simulate the qubits, without having a quantum computer(based on the known mathematical and physics rules), although the support to run your circuits on Yao using real qubits is coming to Yao soon.

We can make a cicuit in Yao using *chain* function. For parameters we define the number of qubits and the operations we've to perform on them. Lets say we want to pass two qubits through two X gates. We do this by chain(number of qubits, operations). To use the X gate, we use the put() parameter. Run the cell below to see what happens.

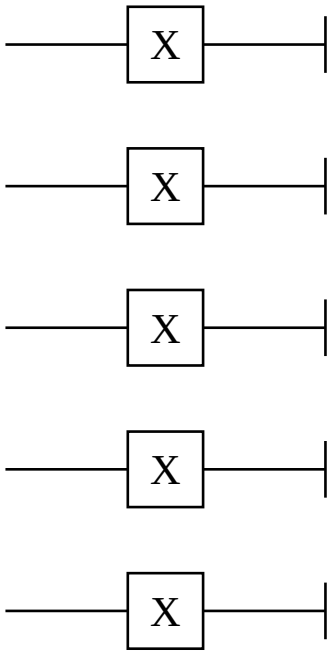
or single # sign mean comments in julia. It means that anything written after # or ## won't be read as a part of the program, in the line you used them.

```
using Yao, YaoPlots #calling the Yao and YaoPlots package
```



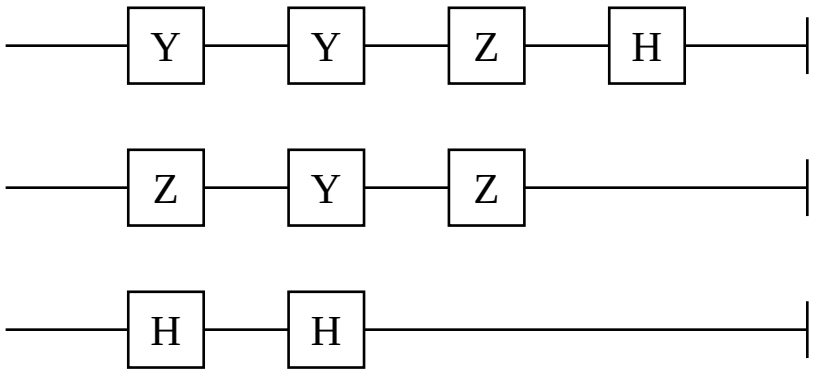

```
let
    circuit = chain(2, put(1=>X), put(2=>X)); #define a variable "circuit" and "put" an X gate on the first qubit, and then put an X gate on the second qubit
    plot(circuit) #plot function, which takes a circuit for a parameter and prints the circuit diagram.
end
```

Assume we have 5 qubits and we have to pass each through an X gate. We can use the *repeat()* parameter to pass the given number of qubits through the same gate.



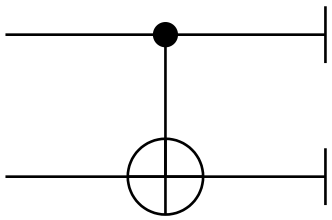

```
plot(chain(5, repeat(X,1:5))) #plot function takes a circuit, which repeats the X gate on the qubits 1:5 or from 1st qubit to 5th qubit
```

What about the Y, Z and H gate?




```
let
    circuit = chain(3, put(1=>Y), put(2=>Z), put(3=>H), repeat(Y, 1:2), repeat(Z, 1:2), repeat(H, [1 3]))
    plot(circuit)
end
```

What about multiqubit gates? We can use the control( ) parameter.



```
plot(chain(2, control(1, 2=>X))) #Which translates to if the state of the 1st qubit is |1>, perform X gate to the 2nd qubit or "put" the 2nd qubit through the X gate.
```

```
using Yao, YaoPlots
```

Applications

We'll discuss a few applications of qubits and quantum circuits. Namely the Bell States, Superdense Coding and Quantum Teleportation.

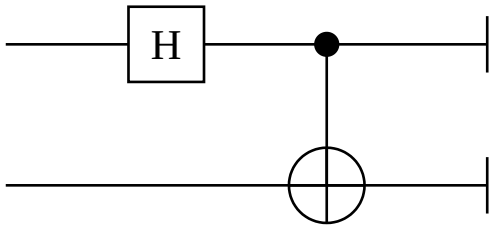
Bell States

The states,

$\frac{|00\rangle+|11\rangle}{\sqrt{2}}$, $\frac{|01\rangle+|10\rangle}{\sqrt{2}}$

$\frac{|00\rangle-|11\rangle}{\sqrt{2}}$ and $\frac{|01\rangle-|10\rangle}{\sqrt{2}}$

are known as the bell states. They are made by the bell circuit. The bell circuit looks like this.




```
begin
    bellcircuit = chain(2, put(1=>H), control(1, 2=>X))
    plot(bellcircuit)
end
```

As you can see, the circuit takes in two qubits as input, and operates on them to give the bell states.

Feeding qubits to a circuit in Yao

There are many ways to create qubits to feed to quantum circuits in Yao.

```
q1 = ArrayReg{1, Complex{Float64}, Array...}
        active qubits: 2/2
```

```
q1 = ArrayReg(bit"00") #creating the system of two qubits with state |00>.
```

```
4×1 Array{Complex{Float64},2}:
```

```
1.0 + 0.0im
0.0 + 0.0im
0.0 + 0.0im
0.0 + 0.0im
```

```
state(q1) #state of a qubit in vector form
```

Note: ***Normalizing*** *basically means making the summation of squares of probability amplitudes, equal to 1.*

Other ways of doing it are

```
ArrayReg{1, Complex{Float64}, Array...}
    active qubits: 2/2
```

```
ArrayReg(bit"00") + ArrayReg(bit"11") |> normalize! #Equivalent to (1/√2)*(|00>+|11>)
```

```
ArrayReg{1, Complex{Float64}, Array...}
    active qubits: 2/2
```

```
zero_state(2) #2 qubits, both with the state |0> and |0>
```

There! We have a system of two qubits! Let's try feeding the qubits to the Bell circuit we made!

```
a = ArrayReg{1, Complex{Float64}, Array...}
        active qubits: 2/2
```

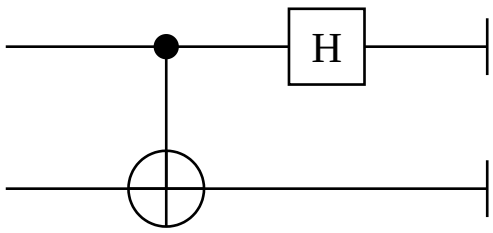
```
a = (q1 |> bellcircuit) #Passing the qubit q1 through the bell circuit
```

```
4×1 Array{Complex{Float64},2}:
 0.7071067811865475 + 0.0im
                0.0 + 0.0im
                0.0 + 0.0im
 0.7071067811865475 + 0.0im
```

```
state(a)
```

Reverse Bell Circuit

A circuit which reverses the effects of the bell circuit on a qubit. It's represented in a circuit as follows




```
begin
    reversebellcircuit = chain(2, control(1,2=>X), put(1=>H))
    plot(reversebellcircuit)
end
```

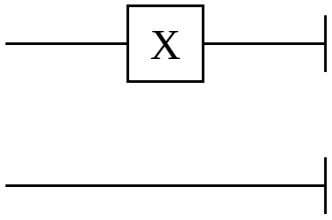
You can input the output state you got from the bell circuit, into the reverse bell circuit, and you'll get back your original state. Why not give it a try?

Let's pass the qubits we got by passing two qubits in the Bell circuit, into the Reverse Bell Circuit.

```
4×1 Array{Complex{Float64},2}:
 0.9999999999999998 + 0.0im
                0.0 + 0.0im
                0.0 + 0.0im
                0.0 + 0.0im
```

```
let
        res = (a |> reversebellcircuit)
        state(res)
end
```

What do you think is the effect of single qubit gates on a qubit, which is entangled with another qubit? Lets check it out!




```
begin
    singlequbitcircuit = chain(2, put(1=>X))
    plot(singlequbitcircuit)
end
```

```
4×1 Array{Complex{Float64},2}:
                0.0 + 0.0im
 0.7071067811865475 + 0.0im
 0.7071067811865475 + 0.0im
                0.0 + 0.0im
```

```
begin
    bellstate = ArrayReg(bit"00") + ArrayReg(bit"11") |> normalize!
    re = bellstate |> singlequbitcircuit
    state(re)
end
```

Can you notice how the circuit behaves as if its operating on one qubit? If the input is $|\{0\}[1]\rangle + |\{1\}[1]\rangle$, it behaves like the bits in the curly braces are the state of one qubit, while the bits in sqare brackets are the state of the other qubit, and it operates on them accordingly?

Super-dense Coding and Quantum Teleportation

Suppose we have two entangled qubits in a bell state, represented by $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$. Alice and Bob are two friends. Alice gets one qubit and Bob gets the other one. Both of them travel far apart with their

qubits, and not measuring their qubits so as to preserve the entangled state. The above information is the premise for both:

1. Super-dense coding
2. Quantum Teleportation

1. Super-dense coding

Alice wants to send Bob two classical bits of information. That means, one of the four states $00, 01, 10$ and 11. How does she do this? Lets say she achieves this by passing her qubits through one of the following gates, corresponding to the information she wants to send.

1. If she wants to send Bob 00, then she'll send her qubit to Bob, as it is
2. If she wants to send Bob 01, then she'll send her qubit to Bob, after passing it through X gate.
3. If she wants to send Bob 10, then she'll send her qubit to Bob, after passing it through Z gate.
4. If she wants to send Bob 11, then she'll send her qubit to Bob, after passing it through Y gate.

Bob will then run both the qubits through the reverse bell circuit and measure them and he'll get what Alice wanted to send him.

Measuring qubits in Yao can be done by using the measure function.

Assume you've the a qubit or a system of qubits in q, and you want to measure it, the syntax is,

```
q |> r->measure(r, nshots=number_of_runs)
```

We can't determine the probability amplitudes of a qubit. But, we can run the measurement many times, and count the frequency of each bit combination it gives.

If you don't get that, just know the syntax and keep nshots=1024.

Now, lets try measuring the result of the previous assignment!

```
measuredqubits =
▸ BitBasis.BitStr{2,Int64}[01 ₍₂₎, 10 ₍₂₎, 01 ₍₂₎, 10 ₍₂₎, 01 ₍₂₎, 01 ₍₂₎, 10 ₍₂₎, 01
```

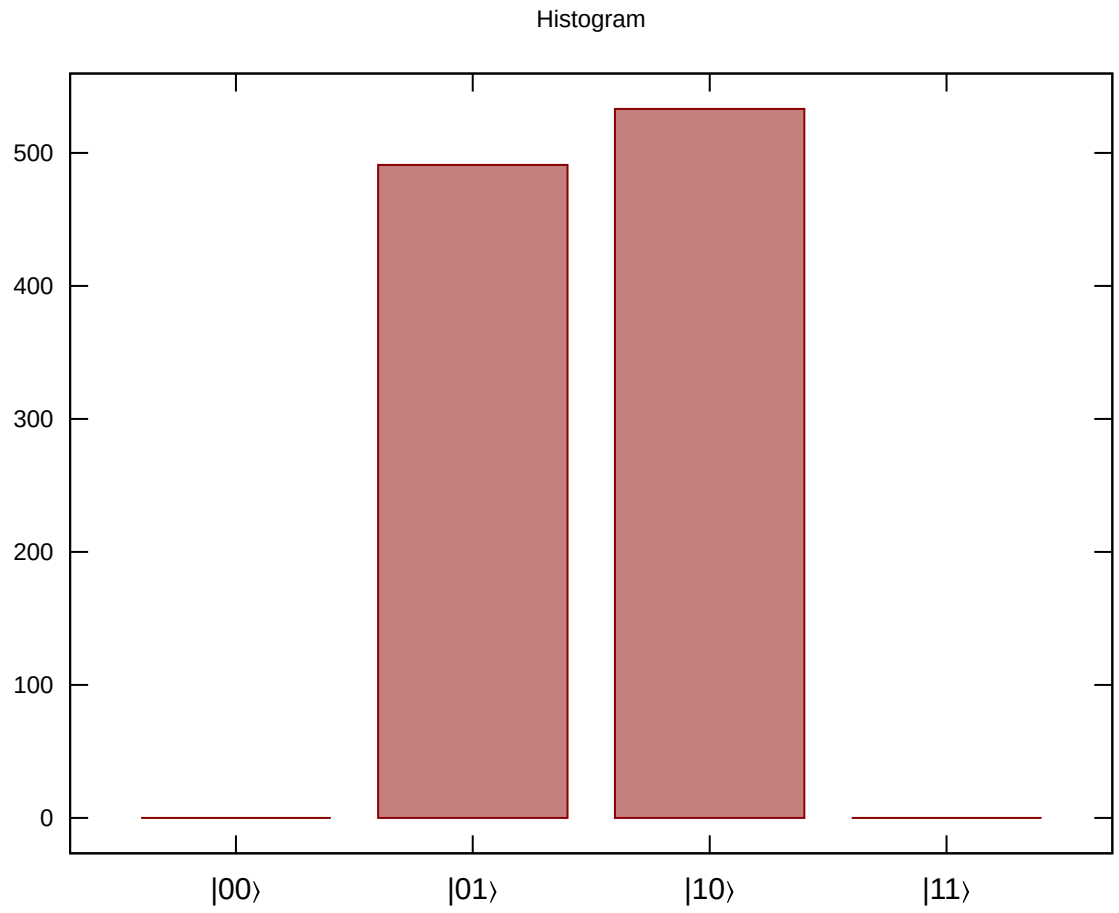
```
measuredqubits = re |> r->measure(r, nshots=1024)
```

Sometimes the measurement gives $|01\rangle$ and sometimes $|10\rangle$. The probability of getting a $|01\rangle$ is same as getting the probability of $|10\rangle$, which is $(1/\sqrt{2})^2$ that is 0.5 .

Wanna see how we can test that? The below function plots the probability as histograms. You don't need to know its inner workings to use it.

```
plotmeasure (generic function with 1 method)
```

```
begin
    using BitBasis: BitStr
    using StatsBase: fit, Histogram
    using Gaston: bar, set, Axes
    set(showable="svg")
    function plotmeasure(x::Array{BitStr{n,Int},1}) where n
        set(preamble="set xtics font ',$(n<=3 ? 15 : 15/(2^(n-3)))'")
        hist = fit(Histogram, Int.(x), 0:2^n)
        bar(hist.edges[1][1:end-1], hist.weights, fc="'dark-red'", Axes(title =
:Histogram, xtics = (0:(2^n-1), "|" .* string.(0:(2^n-1), base=2, pad=n) .* "⟩")))
    end
end
```




```
plotmeasure(measuredqubits)
```

The probability of the measurement giving $|01\rangle$ is 47.94921875% and the number of times the measurement result is $|10\rangle$ is 52.05078125%.

Implementing the superdense coding.

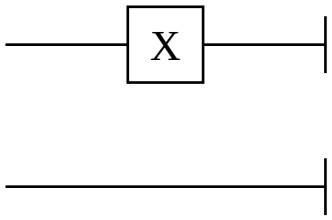
```
"01"
```

```
begin
    Alice_and_Bobs_entangled_qubits = ArrayReg(bit"00") + ArrayReg(bit"11") |>
normalize!
    input = ["00" "01" "10" "11"][rand(1:4)] #Assuming Alice wants to send one of
these inputs to Bob.
```

```
end
```

```
4×1 Array{Complex{Float64},2}:
 0.7071067811865475 + 0.0im
                0.0 + 0.0im
                0.0 + 0.0im
 0.7071067811865475 + 0.0im
```

```
state(Alice_and_Bobs_entangled_qubits)
```




```
begin
    if(input == "00")
        Alices_circuit = chain(2)
    elseif(input == "01")
        Alices_circuit = chain(2, put(1=>X))
    elseif(input == "10")
        Alices_circuit = chain(2, put(1=>Z))
    elseif(input == "11")
        Alices_circuit = chain(2, put(1=>Y))
    end
    plot(Alices_circuit)
end
```

```
Bobs_part =
```

▸ BitBasis.BitStr{2,Int64}[10 ₍₂₎, 10 ₍₂₎, 10 ₍₂₎, 10 ₍₂₎, 10 ₍₂₎, 10 ₍₂₎, 10 ₍₂₎, 10

```
Bobs_part = ((Alice_and_Bobs_entangled_qubits |> Alices_circuit) |>
reversebellcircuit) |> r->measure(r, nshots=1024)
#The content in the first round bracket, outputs qubits, which are then fed to
reversebellcircuit we saw before, which is then fed to the measure function.
```

2. Quantum Teleportation

Alice now has one more qubit, in the state $a|0\rangle + b|1\rangle$. She wants to send her qubit to Bob, by changing the state of Bob's entangled qubit to Alice's extra qubit.

Confusing? Let's try naming the qubits. Alice and Bob have the qubits A and B respectively, which are entangled in the bell state. Alice has another qubit C.

Alice wants to send her qubit to Bob by changing the state of B, to that of C. She wants the state of the qubit B to be $a|0\rangle + b|1\rangle$. Alice doesn't know the values of a and b. Also, Bob can't make a measurement in any case as doing so will destroy B's state.

How does Alice do it? Well, she first passes both her qubits through the Reverse Bell circuit. She then measures both of her qubits. She gets one of the following outputs: $|00\rangle, |01\rangle, |10\rangle or |11\rangle$, each with probability 1/4. Alice sends these two classical bits to Bob, via any classical means, example, she texts him or calls and tells him. Bob knows that corresponding to her message, there are 4 options, and he takes action accordingly for each.

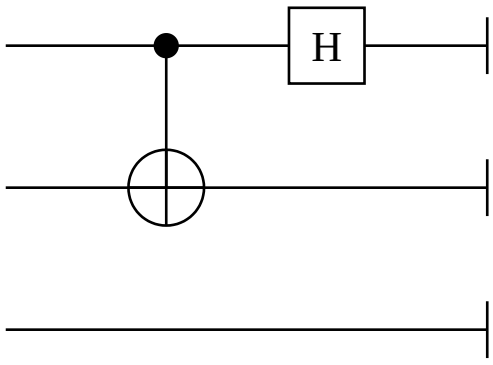
1. He gets the message, $|00\rangle$, from Alice and knows that his qubit(qubit B) changed its state to that of qubit C.
2. He gets the message, $|01\rangle$, and knows that his qubit changed its state to $a|1\rangle + b|0\rangle$, and he passes his qubit through X gate to change its state to that of qubit C.
3. He gets the message, $|10\rangle$, and knows that his qubit changed its state to $a|0\rangle - b|1\rangle$, and he passes his qubit through Z gate to change its state to that of qubit C.
4. He gets the message, $|11\rangle$, and knows that his qubit changed its state to $a|1\rangle - b|0\rangle$, and he passes his qubit through Y gate to change its state to that of qubit C.

```
2×1 Array{Complex{Float64},2}:
 0.44489907827390895 - 0.82452279055161461im
 -0.2056595065687468 + 0.282720967333347841im
```

```
begin
    Alices_and_Bobs_entangled_qubits = ArrayReg(bit"00") + ArrayReg(bit"11") |> normalize!
    Alicequbit = rand_state(1) #This function creates a qubit with a random state.
    state(Alicequbit)
end
```

```
4×1 Array{Complex{Float64},2}:
 0.7071067811865475 + 0.0im
                0.0 + 0.0im
                0.0 + 0.0im
 0.7071067811865475 + 0.0im
```

```
state(Alices_and_Bobs_entangled_qubits)
```




```
begin
    teleportationcircuit = chain(3, control(1,2=>X), put(1=>H))
    plot(teleportationcircuit)
end
```

```
8×1 Array{Complex{Float64},2}:
  0.22244953913695442 - 0.41226139527580724im
  0.22244953913695442 - 0.41226139527580724im
 -0.10282975328437338 + 0.14136048366667 39im
  0.10282975328437338 - 0.14136048366667 39im
 -0.10282975328437338 + 0.14136048366667 39im
  0.10282975328437338 - 0.14136048366667 39im
  0.22244953913695442 - 0.41226139527580724im
  0.22244953913695442 - 0.41226139527580724im
```

```
begin
    feeding = join(Alices_and_Bobs_entangled_qubits, Alicequbit) |> teleportationcircuit
    state(feeding)
end
```

The `join(qubit1, qubit2.....,qubitn)` function is used to join multiple qubits. Remember, the circuit takes the qubits as inputs, in reverse order.

```
Alices_measuredqubits = 00 ₍₂₎
```

```
Alices_measuredqubits = measure_remove!(feeding, 1:2)
```

```
ArrayReg{1, Complex{Float64}, Array...}
    active qubits: 1/1
```

```
if(Alices_measuredqubits == bit"00")
    Bobs_qubit = feeding
elseif(Alices_measuredqubits == bit"01")
    Bobs_qubit = feeding |> chain(1, put(1=>Z))
elseif(Alices_measuredqubits == bit"10")
    Bobs_qubit = feeding |> chain(1, put(1=>X))
else
    Bobs_qubit = feeding |> chain(1, put(1=>Y))
end
```

The difference between `measure(input, location of qubits(optional, measures all qubits if absent), nshots=number of runs)` and `measure_remove!(input, the location of qubits(optional, measures all qubits if absent))` is that the latter collapses the state of the qubit, and removes the $0 + 0im$ rows of the state vector, while the former only measures the state of the qubit.

```
2×1 Array{Complex{Float64},2}:
    0.444899078273909 - 0.8245227905516147im
 -0.20565950656874682 + 0.282720967333479im
```

```
state(Bobs_qubit)
```

Is Alice's qubit same as Bob's qubit now? You can see that for yourself!

```
2×2 Array{Complex{Float64},2}:
 0.444899-0.824523im  0.444899-0.824523im
 -0.20566+0.282721im  -0.20566+0.282721im
```

```
[state(Alicequbit) state(Bobs_qubit)]
```

Left side : State of Alice's qubit. Right side : State of Bob's qubit. Almost equivalent!

More Quantum Gates

As you might have thought, there do exist gates, other than the previously defined ones.

Single qubit gates

The R_φ^Z gate

Passing a qubit through the R_φ^Z is equivalent to multiplying its state vector by $\begin{bmatrix} 1 & 0 \\ 0 & e^{\varphi i} \end{bmatrix}$. Remember, $e^{i\theta} = cos(\theta) + isin(\theta)$.

The R_φ^Z gate can be alternatively denoted by, $\begin{bmatrix} e^{-\varphi i/2} & 0 \\ 0 & e^{\varphi i/2} \end{bmatrix}$. Its just the original matrix, multiplied by $e^{-\varphi i/2}$. We can do this since multiplication by $e^{-\varphi i/2}$ is not *observable* during measurement as its a complex unit and $|e^{i\theta}| = |cos(\theta) + isin(\theta)| = 1$. Remember that the abstract value of a complex number $a + ib$, i.e., $|a + ib| = \sqrt{a^2 + b^2}$ and $sin^2\theta + cos^2\theta = 1$.

Considering a qubit, $a|0\rangle + b|0\rangle$, passing it through the $R_{\frac{\pi}{2}}^Z$ gate is equivalent to $\begin{bmatrix} 1 & 0 \\ 0 & e^{\pi i/2} \end{bmatrix}$. And since $cos(\frac{\pi}{2}) = 0$ and $sin(\frac{\pi}{2}) = 1$, we can rewrite the above as, $\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$.

Lets try the above in Yao! The R_φ^Z gate can be used in Yao with the shift *block*.

```
using Yao, YaoPlots
```

```
2×1 Array{Complex{Float64},2}:
   0.6924477114839013 - 0.6829633197845618im
 -0.10091113544888164 - 0.20950945905131718im
```

```
begin
    qubit = rand_state(1)
    state(qubit)
end
```

```
2×1 Array{Complex{Float64},2}:
  0.6924477114839013 - 0.6829633197845618im
 0.20950945905131718 - 0.10091113544888165im
```

```
state(qubit |> chain(1, put(1=>shift(π/2))))
```

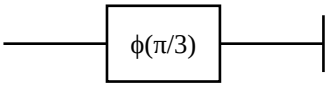
As expected the output was $\begin{bmatrix} a \\ ib \end{bmatrix}$. Remember, $i = \sqrt{-1}$ and $i^2 = -1$. (Also, note that in Julia, imaginary number i is represented by im.)

Also, R_π^Z gate is equivalent to Z gate.

```
true
```

```
round.(Matrix(chain(1, put(1=>shift(π)))) == round.(Matrix(chain(1, put(1=>Z))))
#The round functions "rounds-off" the elements of the matrices
```

Its represented in a circuit diagram by,



The T Gate

The T gate is equivalent to $R_{\frac{\pi}{4}}^Z$. In its matrix form, it can be written as $\begin{bmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \end{bmatrix}$. Nevertheless, in Yao, it can be used by using the **T** *block* .

```
2×1 Array{Complex{Float64},2}:
 0.6924477114839013 - 0.6829633197845618im
 0.2195005073910501 + 0.07679061104477325im
```

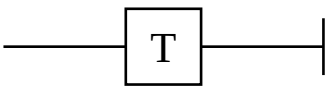
```
state(qubit |> chain(1, put(1=>T)))
```

Also,

```
true
```

```
Matrix(chain(1, put(1=>shift(π/4)))) == Matrix(chain(1, put(1=>T)))
```

Its circuit diagram representation looks somewhat like -



The R_φ^X gate

Similar to the R_φ^Z gate, the R_φ^X gate can be represented by $\begin{bmatrix} cos(\frac{\varphi}{2}) & -sin(\frac{\varphi}{2})i \\ -sin(\frac{\varphi}{2})i & cos(\frac{\varphi}{2}) \end{bmatrix}$.

The R_φ^Y gate

Similar to the R_φ^Z gate, the R_φ^X gate can be represented by $\begin{bmatrix} cos(\frac{\varphi}{2}) & -sin(\frac{\varphi}{2}) \\ sin(\frac{\varphi}{2}) & cos(\frac{\varphi}{2}) \end{bmatrix}$.

They can be represented in Yao using the **Rx** and **Ry** *blocks* respectively

```
2×1 Array{Complex{Float64},2}:
 0.07679061104477329 - 0.21950050739105015im
 -0.6829633197845618 - 0.6924477114839013im
```

```
state(qubit |> chain(1, put(1=>Rx(π))))
```

```
2×1 Array{Complex{Float64},2}:
  0.6829633197845618 + 0.6924477114839013im
 0.07679061104477325 - 0.2195005073910502im
```

```
state(qubit |> chain(1, put(1=>Ry(π))))
```

There's also an **Rz** *block* which which represents the alternative form of R_φ^Z matrix, i.e., $\begin{bmatrix} e^{-\varphi i/2} & 0 \\ 0 & e^{\varphi i/2} \end{bmatrix}$.

```
2×1 Array{Complex{Float64},2}:
 0.6924477114839013 - 0.6829633197845618im
 0.2195005073910502 + 0.07679061104477324im
```

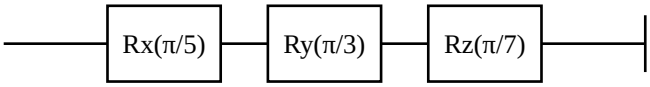
```
state(qubit |> chain(1, put(1=>Rz(π))))
```

Note that the absolute value of both the shift and Rz blocks are same.

```
true
```

```
abs.(Matrix(chain(1, put(1=>shift(π/5))))) == abs.(Matrix(chain(1, put(1=>Rz(π/5)))))
```

The circuit diagram representations of Rx, Ry and Rz blocks, respectively



Multi-qubit Gates

The SWAP Gate

The SWAP gate swaps the state of two qubits. It can be represented by the matrix, $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

Its represented in Yao via the **swap** *block*.

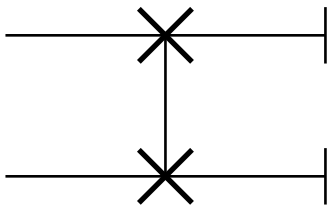
```
4×1 Array{Complex{Float64},2}:
  0.37506700539820803 + 0.5151843025868749im
   0.6485827649421082 + 0.19688812148771873im
  -0.1304450041897797 - 0.19463159785685502im
 -0.17552844135986223 + 0.22085685123666945im
```

```
begin
    q = rand_state(2)
    state(q)
end
```

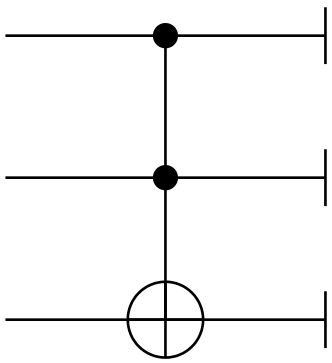
```
4×1 Array{Complex{Float64},2}:
  0.37506700539820803 + 0.5151843025868749im
  -0.1304450041897797 - 0.19463159785685502im
   0.6485827649421082 + 0.19688812148771873im
 -0.17552844135986223 + 0.22085685123666945im
```

```
state(q |> chain(2, swap(1,2)))
```

The SWAP gate has the following circuit diagram representation

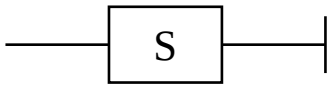


There's a Toffoli gate, an S gate, a CSWAP gate, a $CR_{\varphi}^{X,Y,Z}$ gate and probably a lot more. They can all be constructed using the existing blocks in Yao.

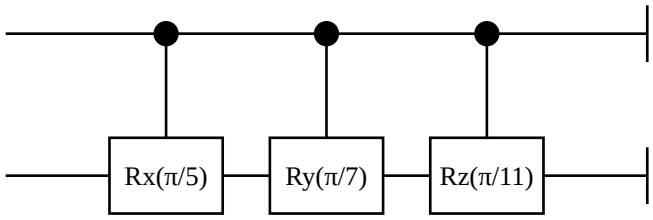


```
let
    toffoli_Gate = chain(3, control(1:2, 3=>X)) #The toffoli gate
    plot(toffoli_Gate)
```

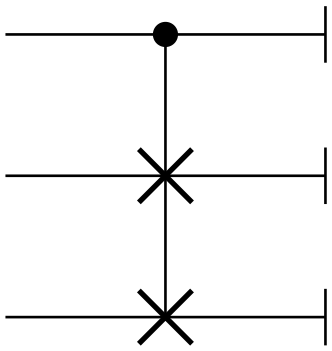
```
end
```




```
let
    S_Gate = chain(1, put(1 => label(shift(π/2), "S"))) #The S gate
    plot(S_Gate)
end
```



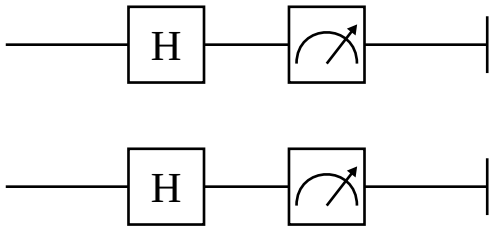

```
let
    CRXYZ_Gates = chain(2, control(1, 2=>Rx(π/5)), control(1, 2=>Ry(π/7)), control(1,
2=>Rz(π/11)))
    plot(CRXYZ_Gates)
end
```



```
let
    CSWAP = chain(3, control(1, 2:3=>SWAP)) #The CSWAP gate
    plot(CSWAP)
end
```

The Measure Gate

We already know how to measure the qubits. We can do it in the circuit itself too.




```
begin
    MeasureGate = chain(2, repeat(H, 1:2), Measure(2, locs=1:2))
    plot(MeasureGate)
end
```

Note that now, when we measure them using the measure block, the output remains unchanged, even though we should've a 25% chance of getting $|00\rangle$, $|01\rangle$, $|10\rangle$ or $|11\rangle$.

```
BitBasis.BitStr{2,Int64}[00 ₍₂₎, 00 ₍₂₎, 00 ₍₂₎, 00 ₍₂₎, 00 ₍₂₎, 00 ₍₂₎, 00 ₍₂₎, 00
```

```
zero_state(2) |> MeasureGate |> r->measure(r, nshots=1024)
```

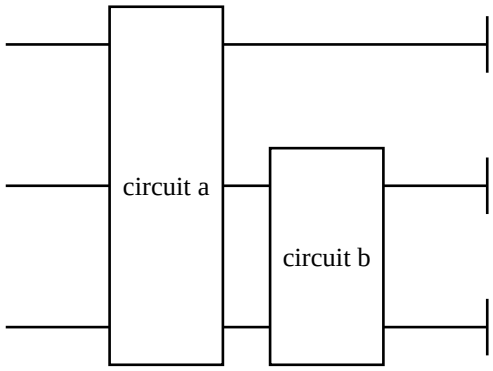
Without the Measure gate,

```
BitBasis.BitStr{2,Int64}[01 ₍₂₎, 00 ₍₂₎, 10 ₍₂₎, 11 ₍₂₎, 11 ₍₂₎, 10 ₍₂₎, 10 ₍₂₎, 10
```

```
zero_state(2) |> repeat(2, H, 1:2) |> r->measure(r, nshots=1024)
```

The LabeledBlock

Its used for easily plotting circuits as boxes for simpler visualization.

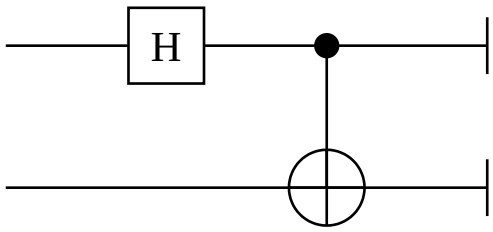



```
let
    a = chain(3, repeat(H, 1:2), put(3=>X))
    b = chain(2, repeat(Y, 1:2))
    circuit = chain(3, put(1:3 => label(a, "circuit a")), put(2:3 => label(b,
"circuit b")))
    plot(circuit)
end
```

Daggered Block

We use Daggered block to build circuits which undo the effects of a particular circuit.

Let's take an example. Remember Bell Circuit?




```
begin
    bellcircuit = chain(2, put(1=>H), control(1, 2=>X))
    plot(bellcircuit)
end
```

```
4×1 Array{Complex{Float64},2}:
 0.7071067811865475 + 0.0im
                0.0 + 0.0im
                0.0 + 0.0im
 0.7071067811865475 + 0.0im
```

```
state(zero_state(2) |> bellcircuit)
```

Building the Reverse Bell Circuit is as easy as,

```
4×1 Array{Complex{Float64},2}:
 0.9999999999999998 + 0.0im
                0.0 + 0.0im
                0.0 + 0.0im
                0.0 + 0.0im
```

```
state(zero_state(2) |> bellcircuit |> Daggered(bellcircuit))
```

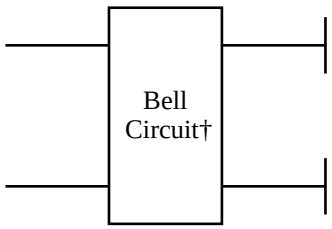
Plotting the Daggered Block is a bit tricky! YaoPlots doesn't support DaggeredBlock yet. There are two alternatives to this.

```
block type YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2} does not support visualization.

  1. error(::String) @ error.jl:33
  2. draw!(::YaoPlots.CircuitGrid, ::YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2},
     ::Array{Int64,1}, ::Array{Any,1}) @ vizcircuit.jl:138
  3. >>(::YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2},
     ::YaoPlots.CircuitGrid) @ vizcircuit.jl:267
  4. (::YaoPlots.var"#23#24"{YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2}})
     (::YaoPlots.CircuitGrid) @ vizcircuit.jl:252
  5. (::YaoPlots.var"#26#27"{YaoPlots.var"#23#24"
     {YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2}},YaoPlots.CircuitGrid})
     () @ vizcircuit.jl:259
  6. canvas(::YaoPlots.var"#26#27"{YaoPlots.var"#23#24"
     {YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2}},YaoPlots.CircuitGrid}) @ brush.jl:1
```

```
 7. #circuit_canvas#25(::Float64, ::Float64, ::typeof(YaoPlots.circuit_canvas),
    ::YaoPlots.var"#23#24"{YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2}},
    ::Int64) @ vizcircuit.jl:258
 8. #vizcircuit#22(::Float64, ::Float64, ::Float64, ::typeof(YaoPlots.vizcircuit),
    ::YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2}) @ vizcircuit.jl:251
 9. vizcircuit(::YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2}) @ vizcircuit.jl:251
10. #plot#21(::Base.Iterators.Pairs{Union{},Union{},Tuple{},NamedTuple{(),Tuple{}}},
    ::typeof(YaoPlots.plot),
    ::YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2}) @ vizcircuit.jl:249
11. plot(::YaoBlocks.Daggered{YaoBlocks.ChainBlock{2},2}) @ vizcircuit.jl:249
12. top-level scope @ Local: 1 [inlined]
```

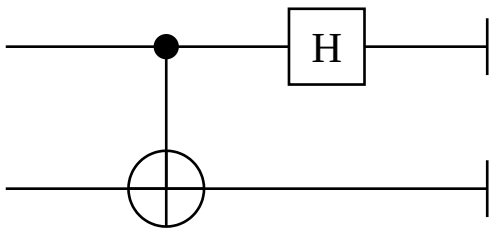
```
plot(Daggered(bellcircuit))
```

One way is to use the label block




```
let
    reversebellcircuit = put(2, 1:2 => label(Daggered(bellcircuit), "Bell\n
Circuit†"))
    plot(reversebellcircuit)
end
```

Another way, is to use `'`.



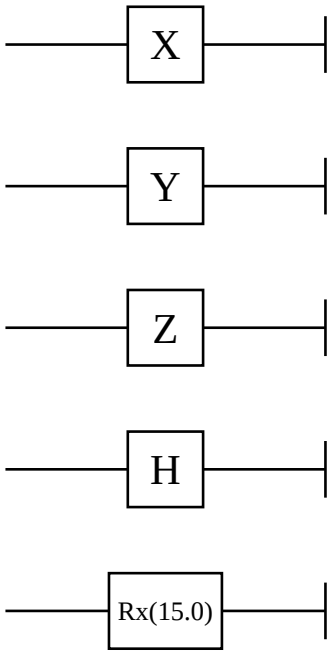

```
plot(bellcircuit')
```

Whats the difference between the DaggeredBlock and adjoint `'`?

One is a function of Yao, while another of Base julia. Both perform the same operation on their input, but DaggeredBlock is many times faster.

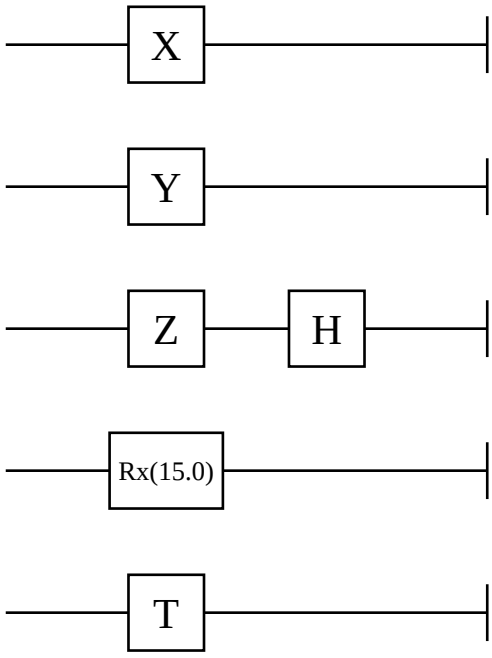
The Kron Block

Consider you've to make the below circuit.



Tired of using `put` block after `put` block?

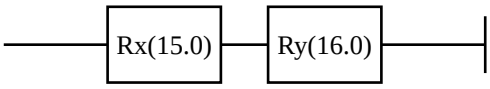
Presenting the kron block, where you can just input the gates on every qubit, one by one.




```
plot(chain(5, put(1:3 => kron(X,Y,Z)), put(3:5 => kron(H,Rx(15),T))))
```

The Rotation Gate

Its the general version of the Rx, Ry and Rz gates you saw above




```
plot(chain(1, rot(X, 15), rot(Y, 16)))
```

Arithmetic using qubits

Binary addition

Suppose we've to add two numbers in binary form! Say... 5 and 7.

In binary form, 5 can be represented by 101 and 7 can be represented by 111. Remember when adding two numbers in binary form,

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$ with a carry of 1

So adding 5 and 7 in binary form looks somewhat like this.

- $\quad {}_{1}{}_{1}{}_{1}{}_{0} \leftarrow$ **Carry**
- $\quad\ 1\ 0\ 1$
- $+\ 1\ 1\ 1$
- $\quad \_\_\_\_\_$
- $\quad 1\ 1\ 0\ 0$

Starting from right, and moving to left

- $1 + 1 = 0$ with a carry of 1
- $0 + 1 = 1$ which is added to the carried value, which is 1, so $1 + 1 = 0$ with a carry of 1, again
- $1 + 1 = 0$ with a carry of 1. The result is added again to the carried value, which was 1, so $0 + 1 = 1$
- The remaining carried value, which is 1, is put as it is.

The result is $[(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)] = 12$.

Quantum addition circuit for one pair of qubits

We'll try making the adder circuit for addition of two numbers $(0-7)$. We'll need 3 qubits to represent each of the two numbers, which means a total of 6 qubits.

```
using Yao,YaoPlots
```

```
1×3 Array{ArrayReg{1,Complex{Float64},Array{Complex{Float64},2}},2}:
 ArrayReg{1, Complex{Float64}, Array...}
    active qubits: 1/1  …  ArrayReg{1, Complex{Float64}, Array...}
    active qubits: 1/1
```

```
begin
    FirstNumber = [ArrayReg(bit"1") ArrayReg(bit"1") ArrayReg(bit"1")] #The first number is 7
    SecondNumber = [ArrayReg(bit"1") ArrayReg(bit"0") ArrayReg(bit"1")] #The second number is 5
end
```

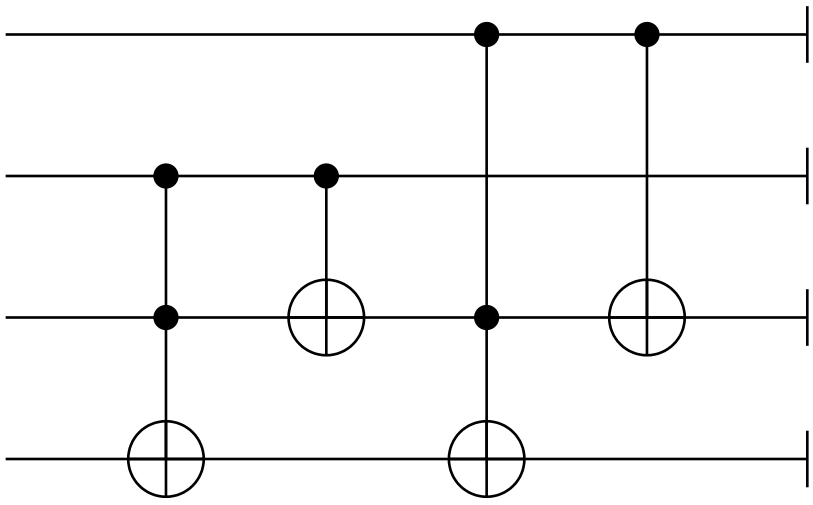
For adding each pair of qubits, we'll need two more qubits to hold the carried in value and carried out value. Since the carry out of the 1st pair acts as the carry in of second pair, the carry out of the second pair acts as the carry in of the 3rd pair, we need 4 more qubits for carry in and carry out, with

- 1 qubit for the carry-in of first pair
- 1 qubit for the carry-out of first pair and the carry-in of the second pair
- 1 qubit for the carry-out of second pair and the carry-in of the third pair
- 1 qubit for the carry-out of third pair

In total, we require a total of 10 qubits to add two numbers in the range $0-7$.

So lets try making the quantum circuit for adding two qubits. We need a qubit for carry-in, which will be zero for the rightmost pair, and a qubit for carry-out, which will act as the carry in for the next pair

Consider this circuit



```
begin
```

```
    OneqbQFA = chain(4, control(2:3, 4=>X), control(2, 3=>X), control([1 3], 4=>X),
control(1, 3=>X))
    plot(OneqbQFA)
end
```

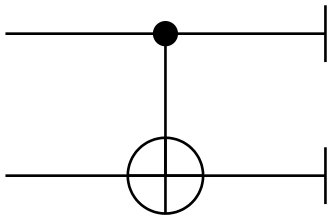
- The top qubit holds the carry-in value.
- The bottom qubit has the state $|0\rangle$.
- The 2nd and the 3rd qubits hold the pair to be added.

After passing throught the circuit,

- The top qubit holds the carry-in value.
- The bottom qubit holds the carry-out value.
- The 2nd qubit is as it was, while the 3rd qubit holds the addition of the 2nd and 3rd qubit.

The use of CX gate in Arithmetics

Consider this circuit



```
begin
    a = chain(2, control(1, 2=>X))
    plot(a)
end
```

Lets try passing different values through this circuit

```
BitBasis.BitStr{2,Int64}[00 ₍₂₎]
```

```
measure((ArrayReg(bit"00") |> a)) #The output is 00, when we passed 00
```

```
BitBasis.BitStr{2,Int64}[11 ₍₂₎]
```

```
measure(ArrayReg(bit"01") |> a) #The output is 11, when we passed 01
#Remember, the circuit takes the qubits as inputs, in reverse order. The rightmost
qubit is the 1st qubit here, and since its |1⟩, the 2nd qubit gets flipped to |1⟩
too.
```

```
BitBasis.BitStr{2,Int64}[10 ₍₂₎]
```

```
measure(ArrayReg(bit"10") |> a) #The output is 10, when we passed 10
```

```
#Remember, the circuit takes the qubits as inputs, in reverse order. The rightmost
qubit is the 1st qubit here, and since its |0⟩, the 2nd qubit is left untouched.
```

```
BitBasis.BitStr{2,Int64}[01 ₍₂₎]
```

```
measure(ArrayReg(bit"11") |> a) #The output is 01, when we passed 11
#Remember, the circuit takes the qubits as inputs, in reverse order. The rightmost
qubit is the 1st qubit here, and since its |1⟩, the 2nd qubit is flipped to |0⟩.
```

Lets analyze the output

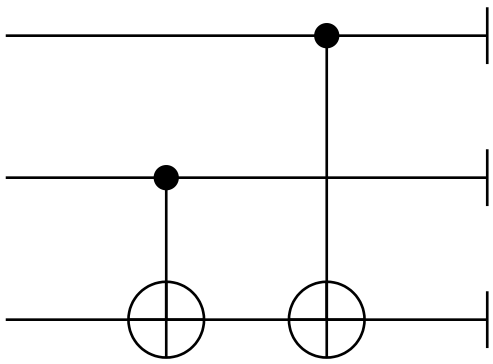
- When the top qubit is $|0\rangle$ and the bottom qubit is $|0\rangle$, the bottom qubit is left untouched.
- When the top qubit is $|1\rangle$ and the bottom qubit is $|0\rangle$, the bottom qubit is flipped to $|1\rangle$.
- When the top qubit is $|0\rangle$ and the bottom qubit is $|1\rangle$, the bottom qubit is left untouched.
- When the top qubit is $|1\rangle$ and the bottom qubit is $|1\rangle$, the bottom qubit is flipped to $|0\rangle$.

Compare this with,

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$

In all the cases, the bottom qubit holds the added value.

How about adding 3 qubits?



```
begin
    b = chain(3, control(2, 3=>X), control(1, 3=>X))
    plot(b)
end
```

```
Array{BitBasis.BitStr{3,Int64},1}[BitBasis.BitStr{3,Int64}[000 ₍₂₎],  BitBasis.BitStr{3,I
```

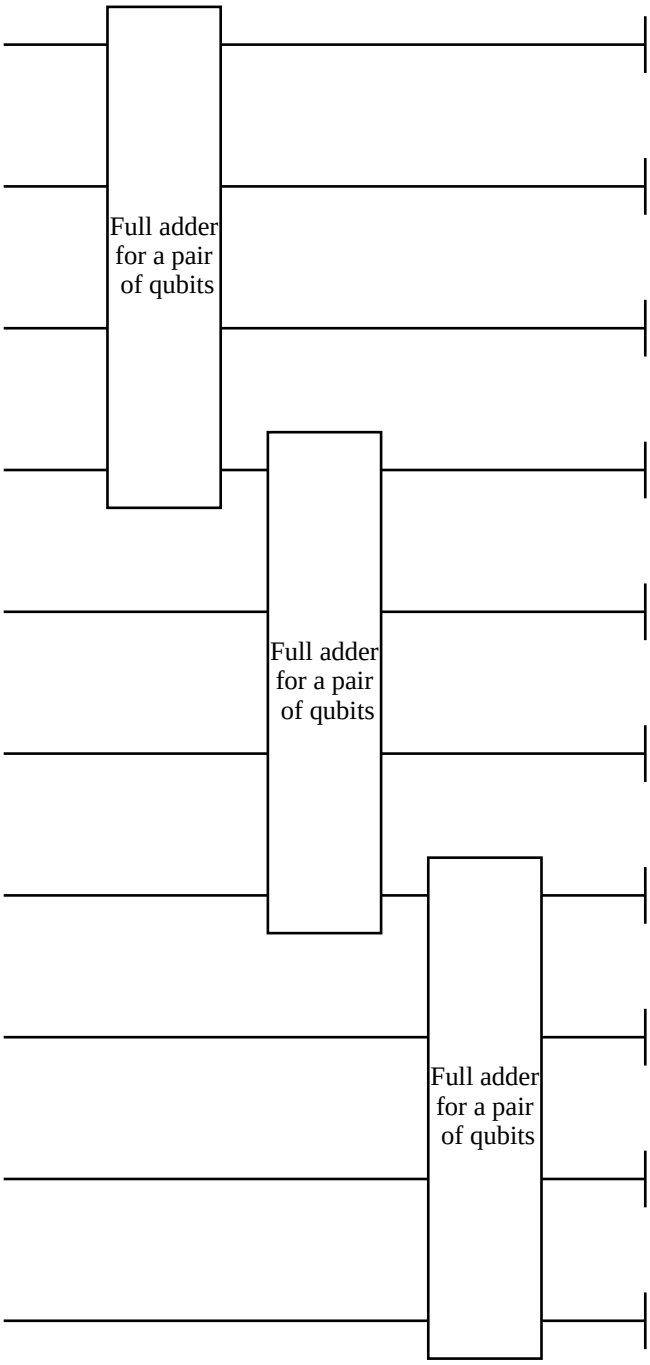
```
measure.([(ArrayReg(bit"000") |> b)
        (ArrayReg(bit"001") |> b)
        (ArrayReg(bit"010") |> b)
        (ArrayReg(bit"011") |> b)
```

```
        (ArrayReg(bit"110") |> b)
        (ArrayReg(bit"101") |> b)
        (ArrayReg(bit"100") |> b)
        (ArrayReg(bit"111") |> b)])
#If the output is not visible, click on the output to expand it.
```

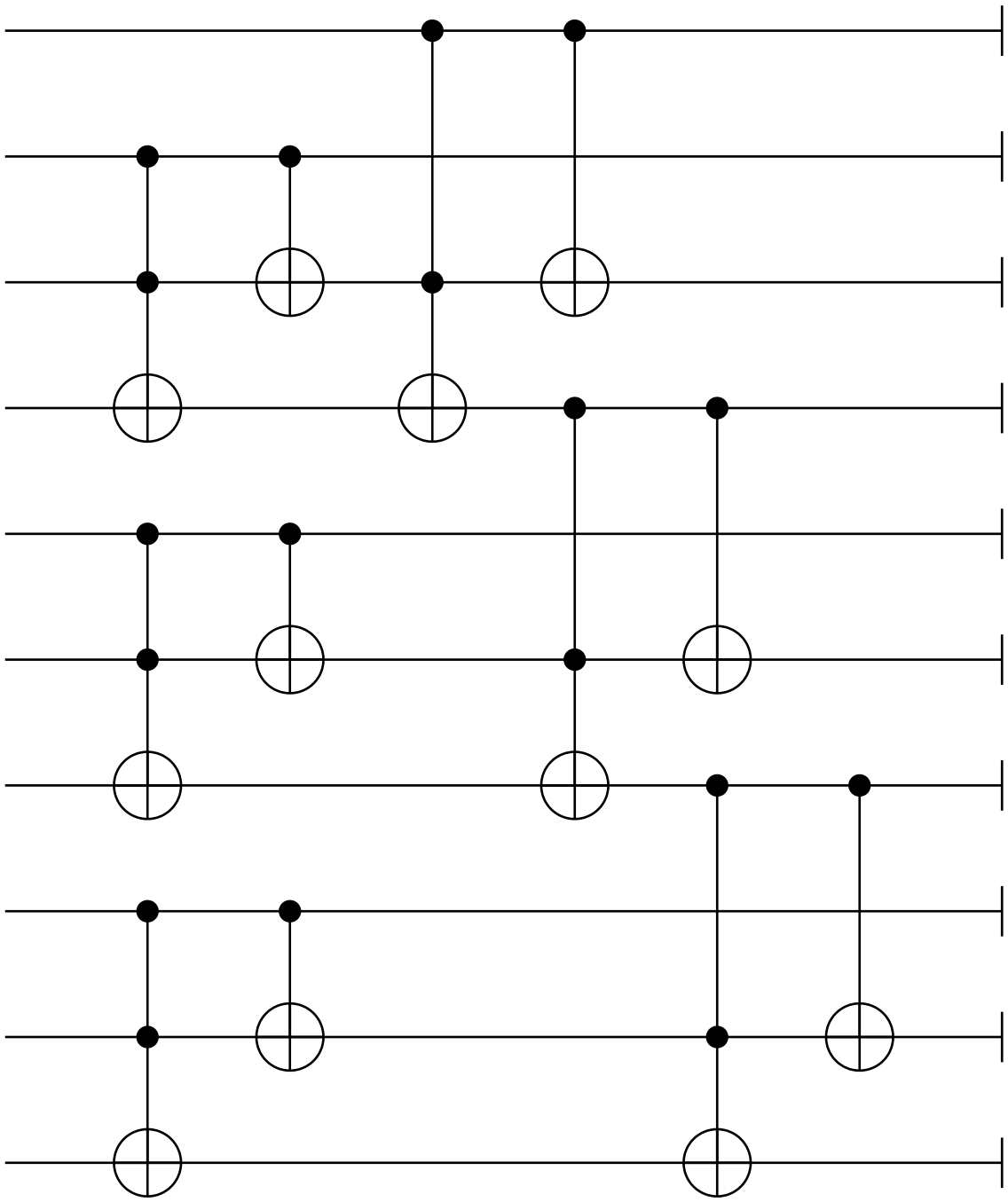
The toffoli gate acts as the carry-out. If both input qubits are $|1\rangle$, the 3rd qubit is flipped.

The Quantum Adder Circuit

We already made the circuit for adding two qubits. Remember that the carry-out for the first pair, becomes the carry in for the second pair. With this, here's the circuit for a quantum adder for 2 numbers between $0 - 7$.



Below is a complete circuit for quantum adder for 2 numbers between $0 - 7$.



```
begin
    nqbQFA = chain(10, put(1:4=>OneqbQFA), put(4:7=>OneqbQFA), put(7:10=>OneqbQFA))
    plot(nqbQFA)
end
```

The input to the circuit being -

```
input = ArrayReg{1, Complex{Float64}, Array...}
            active qubits: 10/10
```

```
input = join(zero_state(1), SecondNumber[1], FirstNumber[1], zero_state(1),
SecondNumber[2], FirstNumber[2], zero_state(1), SecondNumber[3], FirstNumber[3],
zero_state(1))
```

```
results =
```

```
▸ BitBasis.BitStr{10,Int64}[1111011010 ₍₂₎, 1111011010 ₍₂₎, 1111011010 ₍₂₎, 1111011010 ₍₂
```

```
results = input |> nqbQFA |> r->measure(r, nshots=1024)
```

```
stringres = "0101101111"
```

```
stringres = reverse(string(Int(results[1]), base=2, pad=10)) #To convert it to string
```

Remember, the 3, 6 and 9th qubits hold the added values. The 10 qubit holds the final carry-out, if any.

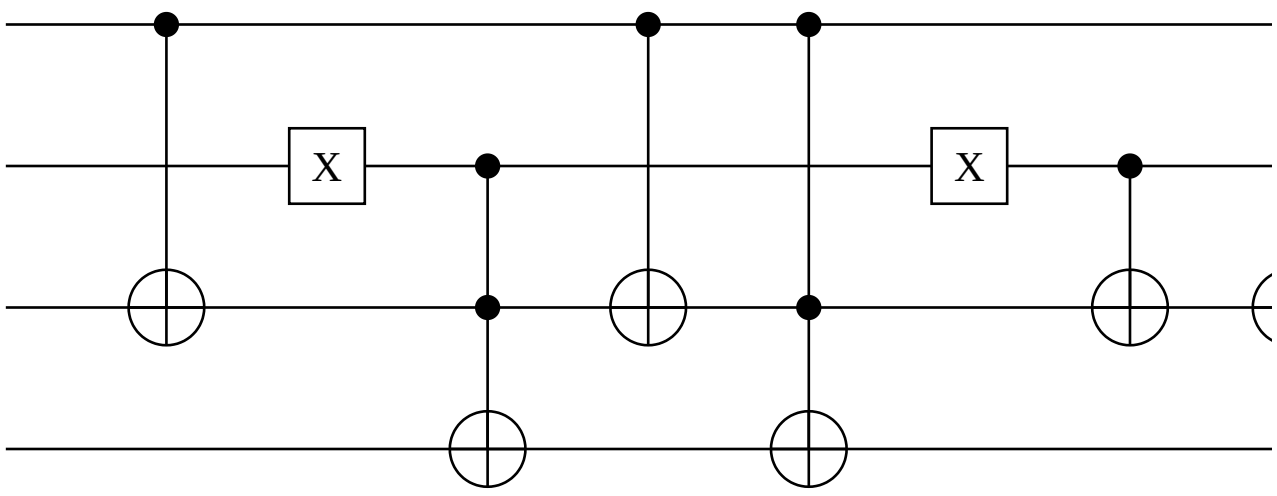
```
output = 12
```

```
output = parse(Int64, reverse(stringres[3] * stringres[6] * stringres[9] *
stringres[10]), base=2)
```

Quantum subtractor

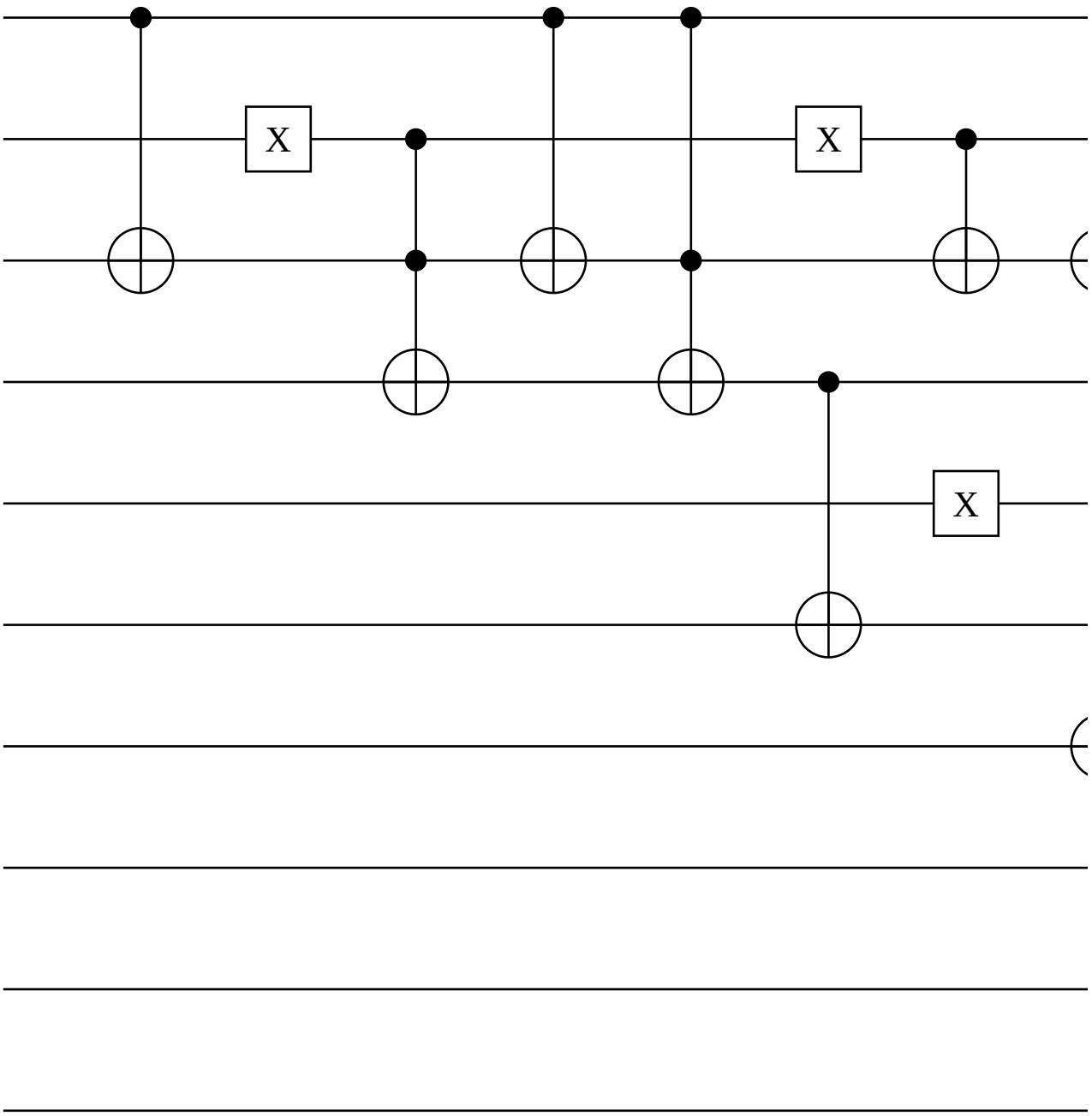
Binary subtraction has a borrow instead of carry. The rules of binary subtraction look somewhat like this.

- $0 - 0 = 0$
- $0 - 1 = 1$ with a borrow of 1
- $1 - 0 = 1$
- $1 - 1 = 0$




```
begin
    OneqbSubtractor = chain(4, control(1, 3=>X), put(2=>X), control(2:3, 4=>X),
control(1, 3=>X), control([1 3], 4=>X), put(2=>X), control(2, 3=>X), control(1,
3=>X))
    plot(OneqbSubtractor)
```

```
end
```




```
begin
    nqbQFS = chain(10, put(1:4=>OneqbSubtractor), put(4:7=>OneqbSubtractor),
put(7:10=>OneqbSubtractor))
    plot(nqbQFS)
end
```

▸ BitBasis.BitStr{4,Int64}[1100 ₍₂₎]

```
measure(join(zero_state(1), ArrayReg(bit"1"), ArrayReg(bit"0"),zero_state(1)) |>
OneqbSubtractor)
```

```
x = 1×3 Array{ArrayReg{1,Complex{Float64},Array{Complex{Float64},2}},2}:
     ArrayReg{1, Complex{Float64}, Array...}
        active qubits: 1/1  …  ArrayReg{1, Complex{Float64}, Array...}
        active qubits: 1/1
```

```
x = [ArrayReg(bit"1") ArrayReg(bit"0") ArrayReg(bit"0")]
```

```
y = 1×3 Array{ArrayReg{1,Complex{Float64},Array{Complex{Float64},2}},2}:
     ArrayReg{1, Complex{Float64}, Array...}
        active qubits: 1/1  …  ArrayReg{1, Complex{Float64}, Array...}
        active qubits: 1/1
```

```
y = [ArrayReg(bit"0") ArrayReg(bit"1") ArrayReg(bit"0")]
```

```
input1 = ArrayReg{1, Complex{Float64}, Array...}
             active qubits: 10/10
```

```
input1 = join(zero_state(1), y[1], x[1], zero_state(1), y[2], x[2], zero_state(1),
y[3], x[3], zero_state(1))
```

```
result =
```

```
▸ BitBasis.BitStr{10,Int64}[0011100000 ₍₂₎, 0011100000 ₍₂₎, 0011100000 ₍₂₎, 0011100000 ₍₂
```

```
result = input1 |> nqbQFS |> r->measure(r, nshots=1024)
```

```
stringsub = "0000011100"
```

```
stringsub = reverse(string(Int(result[1]), base=2, pad=10)) #To convert it to string
```

```
out = 2
```

```
out =  parse(Int64, reverse(stringsub[3] * stringsub[6] * stringsub[9] *
stringsub[10]), base=2)
```

This circuit only subtracts numbers if the answer is expected to be positive. It can't solve for calculations like $5 - 6 = -1$.

Grover's Algorithm

Suppose you've got 10 boxes, each with a paper with a random number on it, and you're searching for a number which may or may not be in one of the boxes. You'll have to search the first box, then the second, then the third... and so on, until you've found what you were looking for. Best case scenario - it just took one search (the number could be in the first box)! Worst case - it took 10 searches(it could've been in the last box, or in no box at all!).

Even if you get a bit clever, sort all the boxes according to the numbers in them, in ascending or descending order, and apply something like **binary search**, it'll almost take $\log_2(n)$ searches.

Grover's Algorithm is a search algorithm, which solves the problem in $O(\sqrt{n})$ **time complexity**(it takes $\pi\sqrt{N}/4$ searches, for one possible match).

Sign flipping

Consider you've 3 qubits, with the state vector :-

$$\frac{1}{\sqrt{8}}|000\rangle + \frac{1}{\sqrt{8}}|001\rangle + \frac{1}{\sqrt{8}}|010\rangle + \frac{1}{\sqrt{8}}|011\rangle + \frac{1}{\sqrt{8}}|100\rangle + \frac{1}{\sqrt{8}}|101\rangle + \frac{1}{\sqrt{8}}|110\rangle + \frac{1}{\sqrt{8}}|111\rangle$$

```
using Yao, YaoPlots
```

```
8×1 Array{Complex{Float64},2}:
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
```

```
begin
    qubits = uniform_state(3)
    state(qubits)
end
```

Suppose there was a circuit U, and if you passed the three qubits to U, the resultant state vector would look somewhat like this :-

$$\frac{1}{\sqrt{8}}|000\rangle + \frac{1}{\sqrt{8}}|001\rangle - \frac{1}{\sqrt{8}}|010\rangle + \frac{1}{\sqrt{8}}|011\rangle + \frac{1}{\sqrt{8}}|100\rangle + \frac{1}{\sqrt{8}}|101\rangle + \frac{1}{\sqrt{8}}|110\rangle + \frac{1}{\sqrt{8}}|111\rangle$$

One thing we know about this *magical* circuit is that its matrix representation looks like this,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

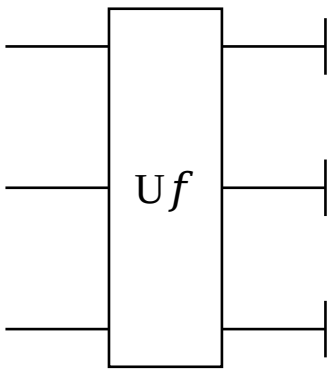
Try multiplying the above state vector to the new matrix.

```
8×1 Array{Complex{Float64},2}:
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
 -0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
```

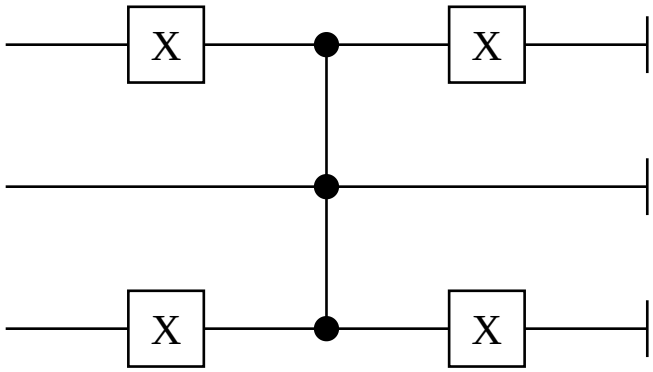
```
let
    U = rand(8,8) |> U->round.(round.(U * inv(U)))
    U[3,3] = -1
    U * state(qubits)
end
```

Creating the magic circuit

You'll have to think about each circuit individually, according to the the element you want to flip. We're flipping $|010\rangle$ in this case. The circuit will be denoted by U_f.

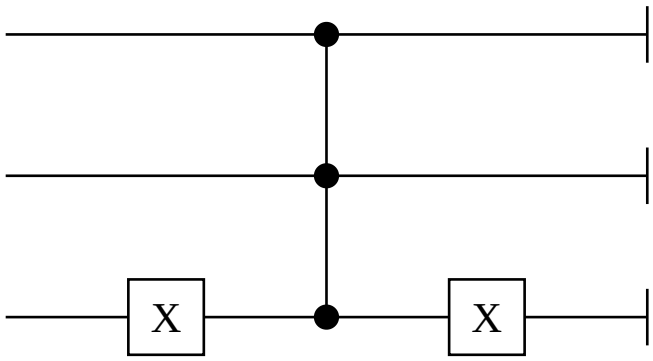


Below is my implementation of this magic circuit




```
begin
    Uf = chain(3, repeat(X, [1 3]), control(1:2, 3=>Z), repeat(X, [1 3]))
    plot(Uf)
end
```

The circuit for $|011\rangle$ will be




```
begin
    Uf_1 = chain(3, repeat(X, [3]), control(1:2, 3=>Z), repeat(X, [3]))
    plot(Uf_1)
end
```

```
8×1 Array{Complex{Float64},2}:
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
```

```
state(uniform_state(3))
```

After passing the uniform qubits through Uf.

```
8×1 Array{Complex{Float64},2}:
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
 -0.35355339059327373 - 0.0im
  0.35355339059327373 + 0.0im
```

```
0.35355339059327373 + 0.0im
0.35355339059327373 + 0.0im
0.35355339059327373 + 0.0im
0.35355339059327373 + 0.0im
```

```
state(uniform_state(3) |> Uf)
```

Amplitude Amplification

Lets consider you want to increase one particular probability amplitude and decrease the rest.

If your qubits were initially in the state :-

$$\frac{1}{\sqrt{8}}|000\rangle + \frac{1}{\sqrt{8}}|001\rangle + \frac{1}{\sqrt{8}}|010\rangle + \frac{1}{\sqrt{8}}|011\rangle + \frac{1}{\sqrt{8}}|100\rangle + \frac{1}{\sqrt{8}}|101\rangle + \frac{1}{\sqrt{8}}|110\rangle + \frac{1}{\sqrt{8}}|111\rangle$$

Then you want them in the state :-

$$1 \times |010\rangle$$

That means, ideally, the probability amplitude of $|010\rangle$ being close to 1, while of others being close to 0.

Inversion about the mean is a neat trick which helps you achieve that.

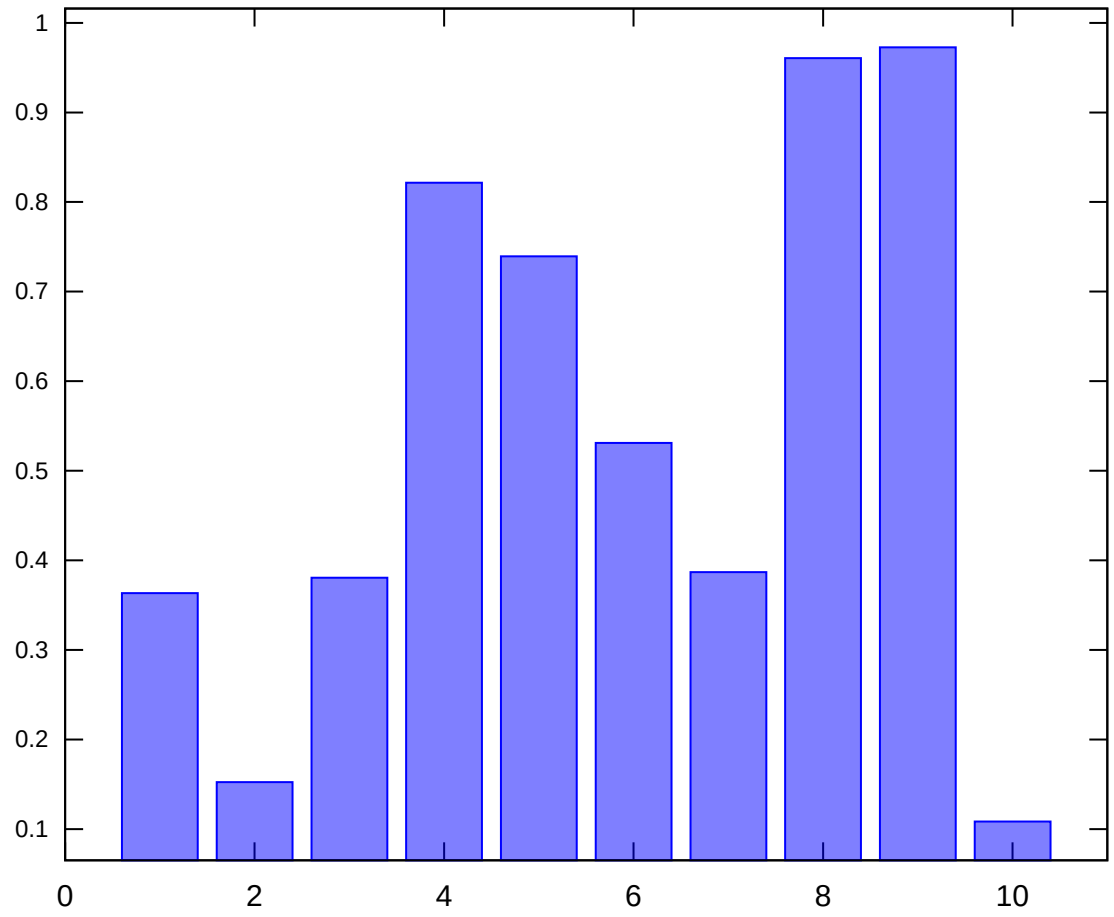
Inversion about the mean

Its simple. I'll give an example.

```
testmat =
▸ Float64[0.363365, 0.152511, 0.38071, 0.821608, 0.739469, 0.531123, 0.386858, 0.96065
```

```
testmat = rand(10)
```

Lets plot it as a histogram




```
bar(testmat)
```

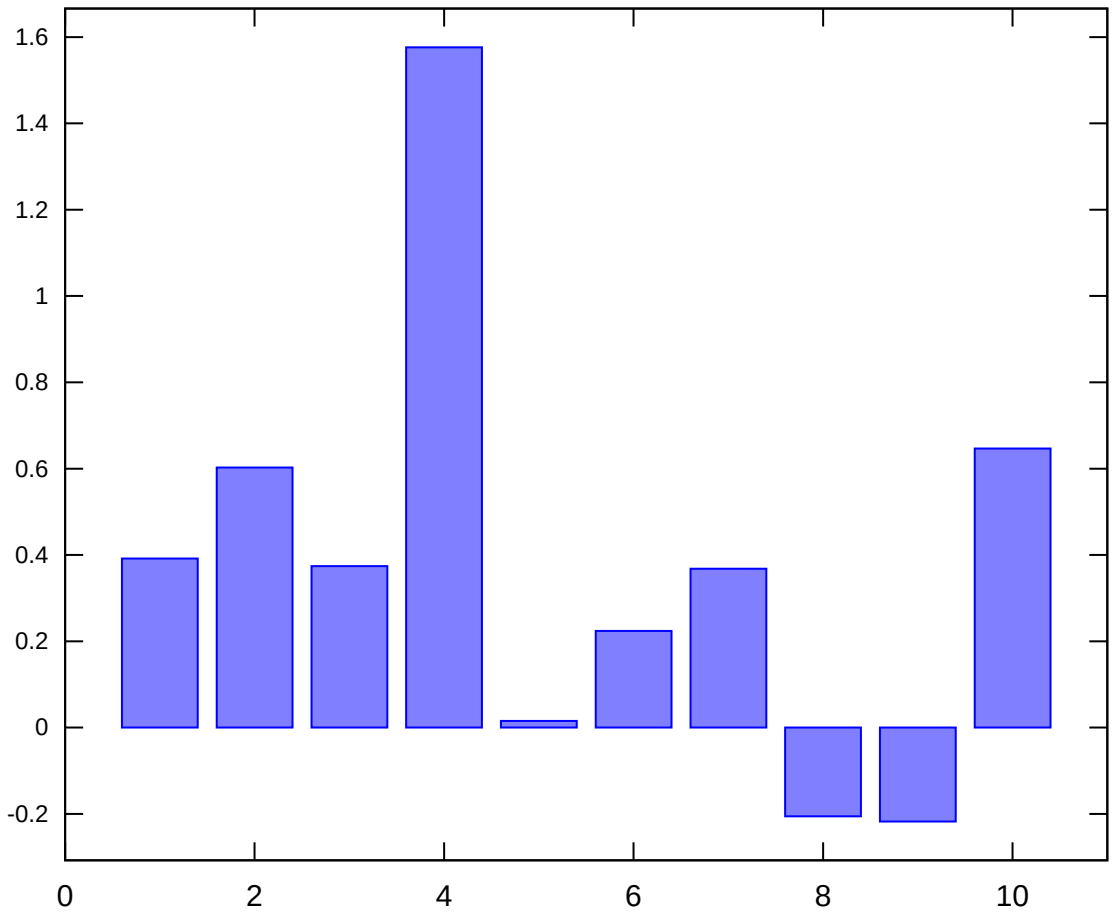
Say, I want to amplify the 4th element. Here's the procedure

```
mean (generic function with 1 method)
```

```
mean(x) = sum(x)/length(x)
```

```
Float64[0.391478, 0.602332, 0.374132, 1.57645, 0.0153731, 0.22372, 0.367985, -0.2058
```

```
begin
    matamplified = copy(testmat)
    matamplified[4] = -matamplified[4]
    matamplified = (2 .* mean(matamplified)) .- matamplified
end
```




```
bar(matamplified)
```

What just happened:-

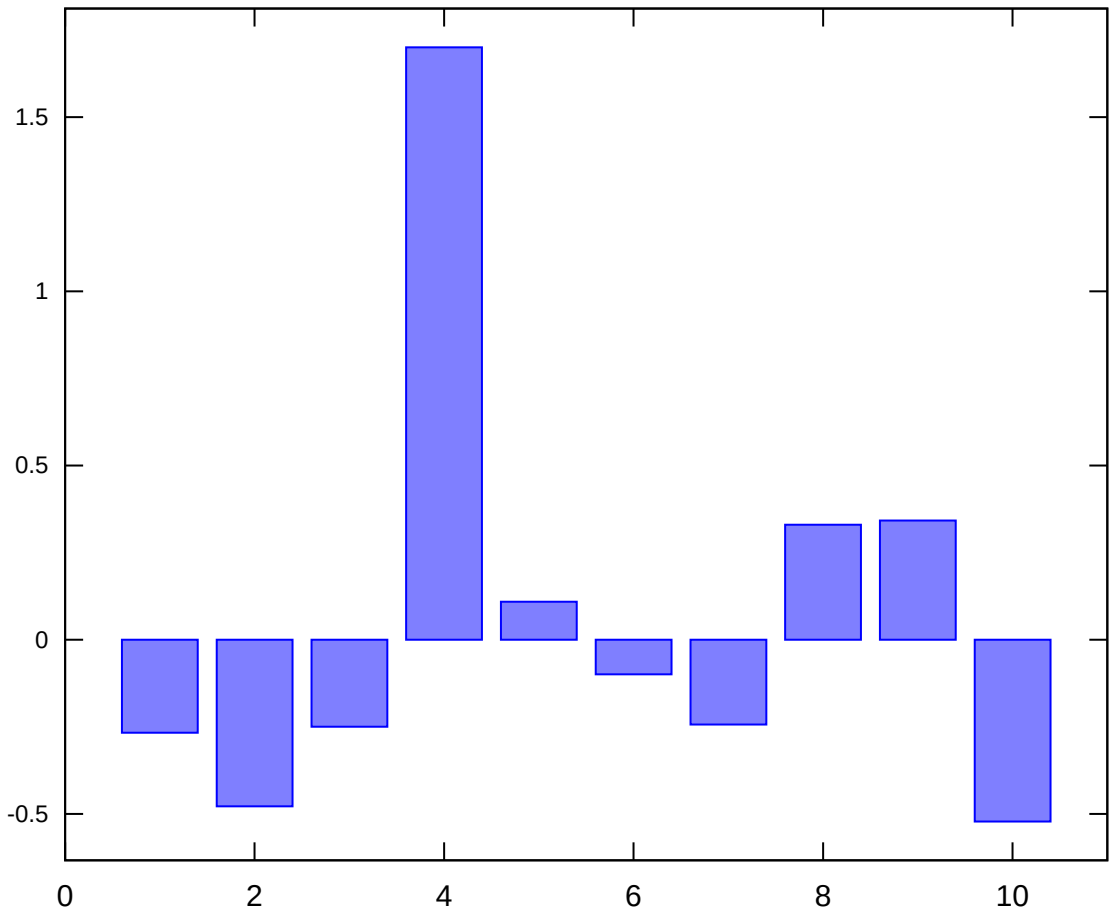
- In an array, choose the element you want to amplify.
- Flip the sign of that element.
- Then the new array with the element amplified, will have the elements :-

$$Amplified\ array = (2 \times mean) - (the\ original\ array\ with\ the\ flipped\ element).$$

What if we do it again? Will it get amplified again?

```
▸ Float64[-0.267216, -0.478069, -0.24987, 1.70071, 0.108889, -0.0994576, -0.243723, 0.
```

```
begin
    newmatamplified = copy(matamplified)
    newmatamplified[4] = -newmatamplified[4]
    newmatamplified = (2 .* mean(newmatamplified)) .- newmatamplified
end
```

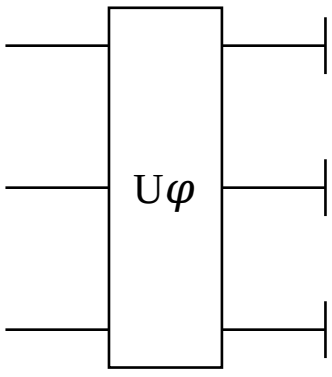



```
bar(newmatamplified)
```

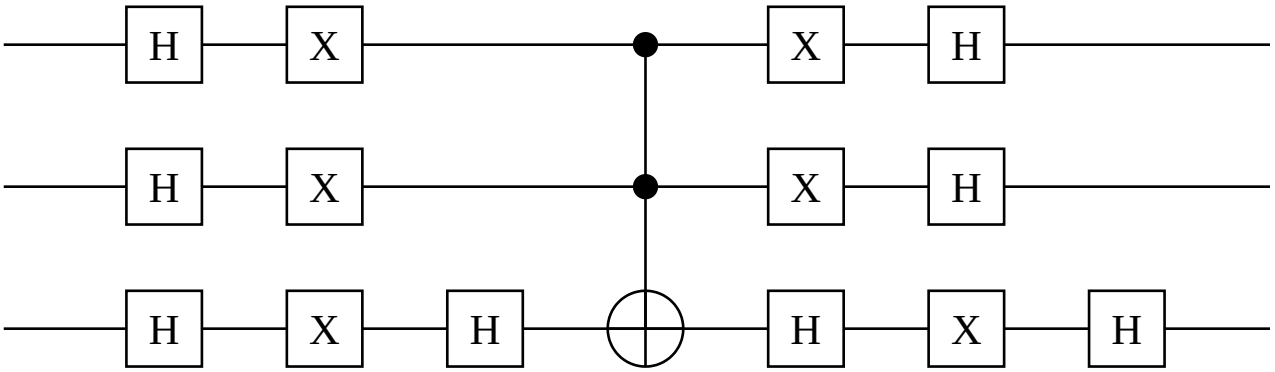
So yes, it does amplified again

The Circuit Implementation

The circuit for amplification looks like this, if the sign of the desired state is flipped

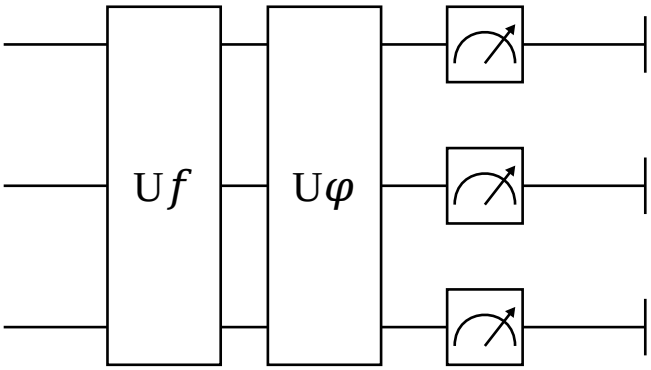


Again, below is my implementation of it.

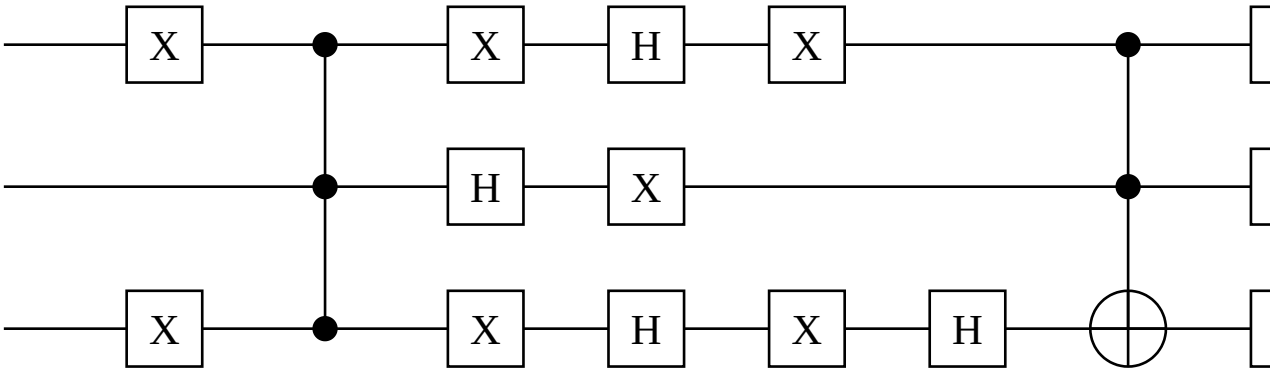



```
begin
    Uφ = chain(3, repeat(H, 1:3), repeat(X, 1:3), put(3=>H), control(1:2, 3=>X), put(3=>H), repeat(X, 1:3), repeat(H, 1:3))
    plot(Uφ)
end
```

Combining this with the circuit for flipping, we get the Grovers Search Circuit, to which we feed qubits with ***uniform state***.



Below is the complete circuit implementation. Remember, the input to the circuit is quits with uniform state.




```
begin
    GroversSearchCircuit = chain(3, put(1:3=>Uf), put(1:3=>Uφ), Measure(3, locs=1:3))
    plot(GroversSearchCircuit)
```

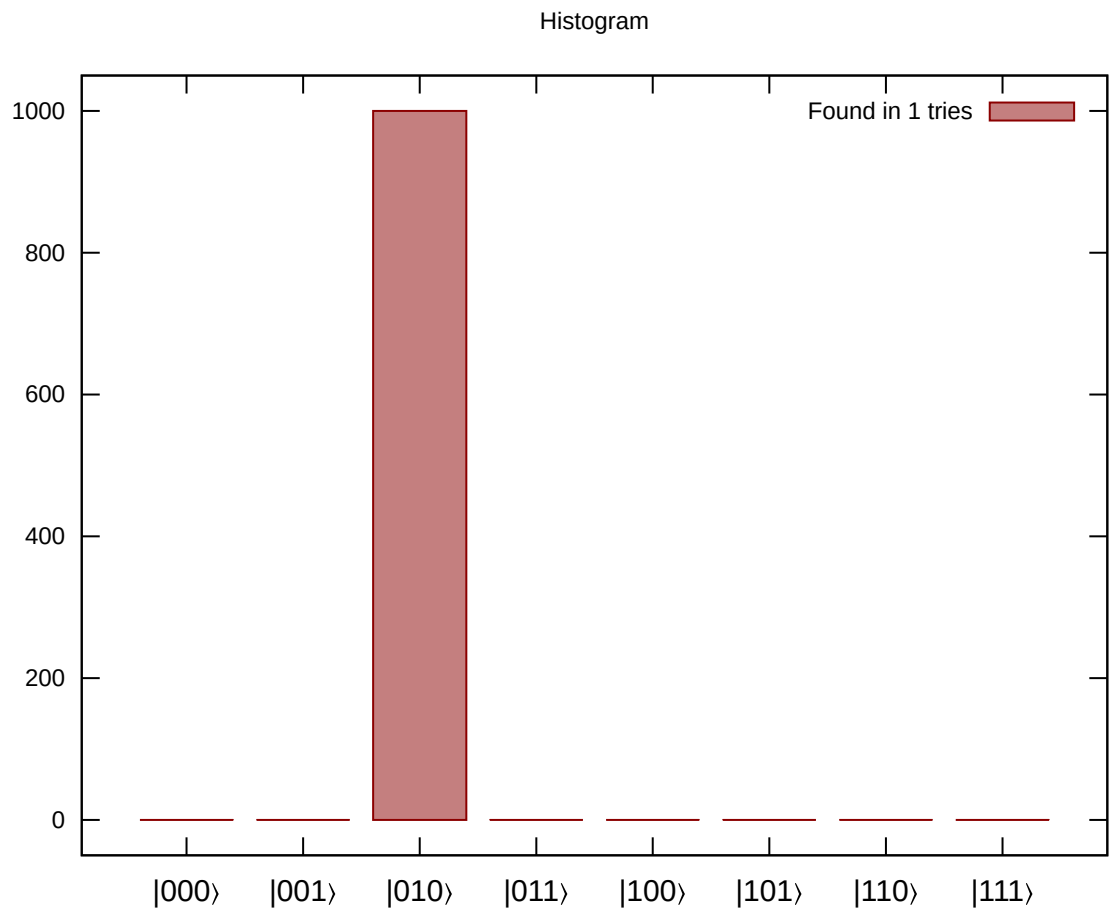
```
end
```

The below function plots the measurement function, just pass the measured values to it. You don't need to know its inner mechanisms to use it.

```
plotmeasure (generic function with 2 methods)
```

```
begin
    using BitBasis: BitStr
    using StatsBase: fit, Histogram
    using Gaston: bar, set, Axes
    set(showable="svg")
    function plotmeasure(x::Array{BitStr{n,Int},1}, s = "#") where n
        set(preamble="set xtics font ',$(n<=3 ? 15 : 15/(2^(n-3)))'")
        hist = fit(Histogram, Int.(x), 0:2^n)
        bar(hist.edges[1][1:end-1], hist.weights, fc="'dark-red'", legend="'Found in $s tries'", Axes(title = :Histogram, xtics = (0:(2^n-1), "|" .* string.(0:(2^n-1), base=2, pad=n) .* "⟩")))
    end
end
```

Below is the implementation of the ***Grover's Search Algorithm*** to find $|010\rangle$.




```
begin
    n = 3
    output = 0
    j = 0
    for i in 1:(2^3)
```

```
        input = uniform_state(n)
        global output = input |> GroversSearchCircuit |> r->measure(r, nshots=1000)
        if(output[1] == bit"010") #Checking for |010⟩
            break
        end
        global j = i
    end
    plotmeasure(output, j+1)
end
```

You can keep running the above block of code, and you'll find that it takes a maximum of 4 tries to find the state $|010\rangle$, which would be the worst case. Most of the times, you can find it in the first try!

Which sounds about right.

Deutsch Algorithm

Deutsch's Algorithm was the first algorithm to prove that quantum computers can perform some tasks faster than classical ones, although the speedup in this case is a trivial one.

The problem looks somewhat like this :-

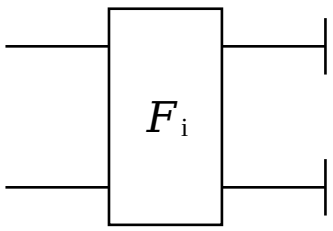
```
Consider 4 functions f₀, f₁, f₂ and f₃. Each of them can take 0 or 1 as input.

f₀(0) = 0, f₀(1) = 0 -> Constant function
f₁(0) = 0, f₁(1) = 1 -> Balanced function
f₂(0) = 1, f₂(1) = 0 -> Balanced function
f₃(0) = 1, f₃(1) = 1 -> Constant function

Given one of these four functions at random, how many evaluations must we make to co
nfirm whether the given function is balanced or constant.
```

Classically, it takes at least two evaluations to confirm whether the given function is balanced or constant.

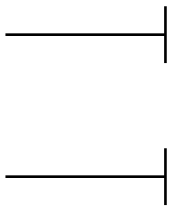
Using quantum computers, we can confirm whether the given function is balanced or not using one evaluation.

First we construct gates that correspond to the 4 functions. Here, let's consider a circuit, which takes the qubits $|x\rangle$ and $|y\rangle$, and returns the qubits $|x\rangle$ and $|y \oplus f_i(x)\rangle$ respectively, where i can be any random number between 0 to 3. Let's call this circuit F_i .

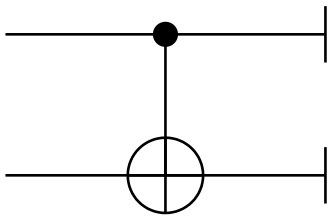
```
using Yao, YaoPlots
```



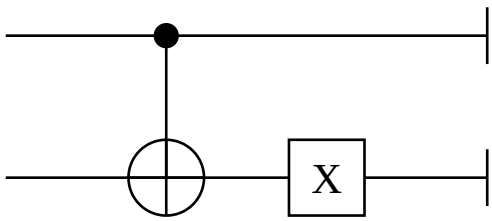
Below is my implementation of F_i



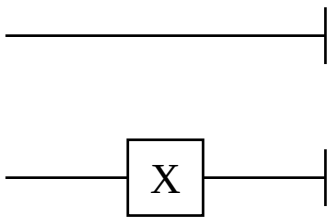
```
begin
    F₀ = chain(2)
    plot(F₀)
end
```



```
begin
    F₁ = chain(2, control(1, 2=>X))
    plot(F₁)
end
```




```
begin
    F₂ = chain(2, control(1, 2=>X), put(2=>X))
    plot(F₂)
end
```




```
begin
    F₃ = chain(2, put(2=>X))
    plot(F₃)
end
```

```
circuit = 1×4 Array{ChainBlock{2},2}:
           nqubits: 2
          chain
            … nqubits: 2
          chain
          └─ put on (2)
             └─ X
```

```
circuit = [F₀ F₁ F₂ F₃]
```

```
r = 2
```

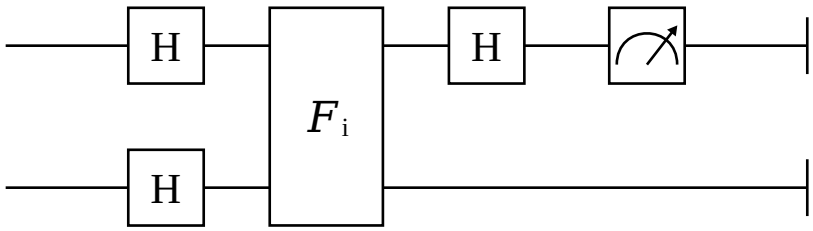
```
r = rand(0:3)
```

The new question now is :-

> Given one of the four circuits at random, how many evaluations would it take to find whether the underlying function is balanced or constant?

If, in the above circuits we pass 0 or 1, it'll be the same as classical computers - it'll take 2 evaluations. To achieve this, we'll pass a superposition of 0 and 1, to the F_i circuit, and then we'll pass an H gate to the first qubit and measure it.

The final circuit for Deutsch Algorithm where the input is $|01\rangle$ looks somewhat like this



Below is the final implementation, using the F_i we constructed before.

```
BitBasis.BitStr{1,Int64}[1 ₍₂₎]
```

```
begin
    input = ArrayReg(bit"10") #Remember, the circuit takes the qubits in reverse order
    result = input |> chain(2, repeat(H,1:2), put(1:2=>circuit[r+1]), put(1=>H)) |> r->measure(r,1)
end
```

```
"Balanced"
```

```
result[1] == bit"1" ? "Balanced" : "Constant"
```

As you can see from the value of r, the result is correct.

Deutsch–Jozsa Algorithm

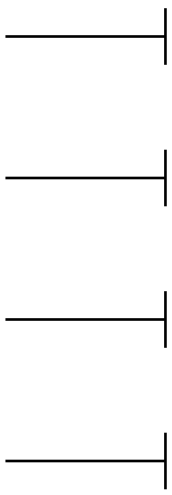
The Deutsch-Jozsa Algorithm is the *general* version of the Deutsch Algorithm. The problem is : –

```
The functions are now of n variables. The inputs to each of these n variables can ei
ther be 0 or 1. So can be the output. The function can either be constant, where all
the inputs get sent to 0 or all the inputs get sent to 1, or balanced, where half th
e inputs get sent to 0 and the rest get sent to 1.
To illustrate this, let's take an example of 4 qubits. Which means 2⁴ possible input
s, as each input can either be 0 or 1.
Let's check each input -
(0,0,0,0)   (1,1,1,1)
(0,0,0,1)   (1,1,1,0)
(0,0,1,0)   (1,1,0,1)
(0,1,0,0)   (1,0,1,1)
(1,0,0,0)   (0,1,1,1)
(0,0,1,1)   (1,1,0,0)
(1,1,0,0)   (0,0,1,1)
(0,1,1,0)   (1,0,0,1)
So if f(0,0,0,0) = 0, f(all 2⁴ combinations) = 0
or f(0,0,0,0) = 1, f(all 2⁴ combinations) = 1
The function is constant. Else, it's balanced.
```

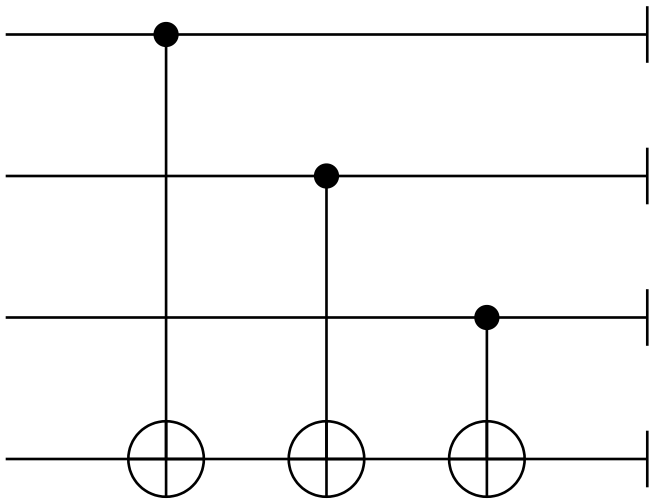
Deutsch Algorithm is a case of Deutsch-Josza Algorithm, where the input is 1 qubit. Drawing inspiration from our above circuit, let's make a circuit for n inputs.

```
n = 3
```

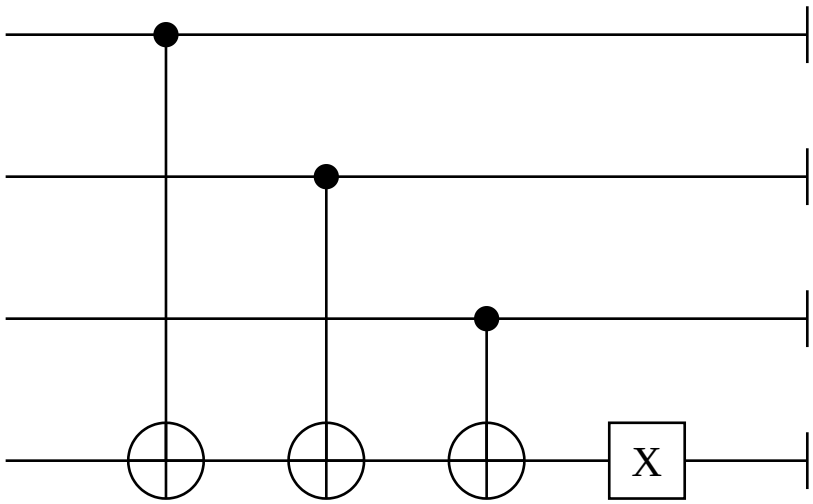
```
n = 3
```



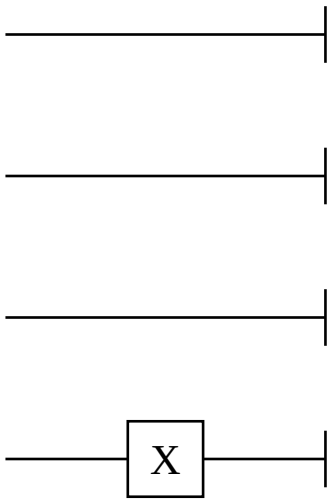
```
begin
    f₀ = chain(n+1)
    plot(f₀)
end
```



```
begin
    f₁ = chain(n+1, [control(k, n+1=>X) for k in 1:n])
    plot(f₁)
end
```




```
begin
    f₂ = chain(n+1, chain(n+1, [control(k, n+1=>X) for k in 1:n]), put(n+1=>X))
    plot(f₂)
end
```




```
begin
    f₃ = chain(n+1, put(n+1=>X))
    plot(f₃)
end
```

```
f = 1×4 Array{ChainBlock{4},2}:
 nqubits: 4
chain
  … nqubits: 4
chain
└─ put on (4)
   └─ X
```

```
f = [f₀ f₁ f₂ f₃]
```

```
newr = 0
```

```
newr = rand(0:3)
```

```
▸ BitBasis.BitStr{3,Int64}[000 ₍₂₎]
```

```
begin
    i = join(ArrayReg(bit"1"), zero_state(n))
    output = i |> chain(n+1, repeat(H, 1:n+1), put(1:n+1=>f[newr+1]), repeat(H,1:n)) |> r->measure(r,1:n)
end
```

If output of the measured qubits is 000, then the function is constant, else if, the output is 111, the function is balanced

```
"Constant"
```

```
output == measure(zero_state(n)) ? "Constant" : "Balanced"
```

Please note, you can change the value of n above to get the Deutsch-Josza Algorithm for different inputs. $n = 1$ will give the Deutsch Algorithm.

Simon's Algorithm

Simon's algorithm was the algorithm that inspired Shor in making the Shor's Algorithm. This is a great algorithm to have a look at hybrid algorithms, as this is a hybrid algorithm.

The problem is : –

```
Suppose there's a binary string of length n. Binary string is a string composed of 0
s and 1s. There's a function f, where f(x) = f(y), if and only if, y = x or y = x ⊕
 s. x and y are binary strings of length n. s is a “secret” binary string of length
 n, where s is not all 0s. So, s can be any one of the possible (2ⁿ - 1) binary stri
ngs.

We've to find the secret string s.

Example : -
    Suppose that n = 3.
    Suppose we find that f(000) = f(101), it means that 000 ⊕ s = 101.
    Remember, these are binary strings and not binary numbers. So 111 ⊕ 100 = 011.
    From the above information, we know that s is 101, as 000 ⊕ 101 = 101.

The question is, how many times do we need to evaluate f, to find s?
Also, we don't know what the secret string s or the function f are.

Classically, we need at least 5 evaluations. We evaluate any four strings of length
 n, on the the function f, and they might all give different results. But the evalua
tion of the fifth string, on the function f, is bound to repeat one of the values fr
om the previously evaluated four strings. Suppose f(010) and f(110) give the same re
sult. That means, f(010) = f(110), which means 010 ⊕ s = 110. Adding 010 to both sid
es, 010 ⊕ 010 ⊕ s = 010 ⊕ 110 => 000 ⊕ s = 100, s = 100.

In general, for binary string of length n, we need to make (2ⁿ⁻¹ + 1) evaluations.
```

Kronecker Product of Hadamard Gate

We know that the Hadamard Gate can be represented by the matrix, $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$.

- Applying H gate on 2 qubits, in the state $|00\rangle$, we get, $|00\rangle + |01\rangle + |10\rangle + |11\rangle$

- Applying H gate on 2 qubits, in the state $|01\rangle$, we get, $|00\rangle - |01\rangle + |10\rangle - |11\rangle$
- Applying H gate on 2 qubits, in the state $|10\rangle$, we get, $|00\rangle + |01\rangle - |10\rangle - |11\rangle$
- Applying H gate on 2 qubits, in the state $|11\rangle$, we get, $|00\rangle - |01\rangle - |10\rangle + |11\rangle$

The Matrix representation for $H^{\otimes 2}$ can be given as, $\frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$

```
using Yao, YaoPlots
```

Which we can verify below

```
4×4 Array{Complex{Float64},2}:
 0.5+0.0im   0.5+0.0im   0.5+0.0im   0.5+0.0im
 0.5+0.0im  -0.5+0.0im   0.5+0.0im  -0.5+0.0im
 0.5+0.0im   0.5+0.0im  -0.5+0.0im  -0.5+0.0im
 0.5+0.0im  -0.5+0.0im  -0.5+0.0im   0.5-0.0im
```

```
Matrix(repeat(2, H, 1:2))
```

We can rewrite the above as

$$\frac{1}{\sqrt{2}}\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Which can be again rewritten as

$$H^{\otimes 2} = \frac{1}{\sqrt{2}}\begin{bmatrix} H & H \\ H & -H \end{bmatrix}$$

Following this trend,

$$H^{\otimes 3} = \frac{1}{\sqrt{2}}\begin{bmatrix} H^{\otimes 2} & H^{\otimes 2} \\ H^{\otimes 2} & -H^{\otimes 2} \end{bmatrix}$$

$$H^{\otimes 4} = \frac{1}{\sqrt{2}}\begin{bmatrix} H^{\otimes 3} & H^{\otimes 3} \\ H^{\otimes 3} & -H^{\otimes 3} \end{bmatrix}$$

-
-
-

$$H^{\otimes n} = \frac{1}{\sqrt{2}} \begin{bmatrix} H^{\otimes n-1} & H^{\otimes n-1} \\ H^{\otimes n-1} & -H^{\otimes n-1} \end{bmatrix}$$

This is known as the Kronecker product of Hadamard Gate.

Dot Product of binary strings

*The dot product of two binary strings, a and b, both of length n, where $a = a_0a_1a_2....a_{n-1}$, and $b = b_0b_1b_2....b_{n-1}$, the **dot product** of a and b, $a \cdot b$, is defined as*

$a \cdot b = a_0 \times b_0 \oplus a_1 \times b_1 \oplus a_2 \times b_2 .... a_{n-1} \times b_{n-1}$

It's always equal to 0 or 1. If $a = 0010$ and $b = 0101$, then

$a \cdot b = 0 \times 0 \oplus 0 \times 1 \oplus 1 \times 0 \oplus 0 \times 1 = 0 \oplus 0 \oplus 0 \oplus 0 = 0$

Lets check out the dot products of all possible combinations for binary strings where $n = 2$.

$\begin{bmatrix} 00 \cdot 00 & 00 \cdot 01 & 00 \cdot 10 & 00 \cdot 11 \\ 01 \cdot 00 & 01 \cdot 01 & 01 \cdot 10 & 01 \cdot 11 \\ 10 \cdot 00 & 10 \cdot 01 & 10 \cdot 10 & 10 \cdot 11 \\ 11 \cdot 00 & 11 \cdot 01 & 11 \cdot 10 & 11 \cdot 11 \end{bmatrix}$. Which calculates to $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$.

Remember $H^{\otimes 2}$, which could be represented by, $\frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$?

It can also be represented by, $\frac{1}{2}\begin{bmatrix} (-1)^{00 \cdot 00} & (-1)^{00 \cdot 01} & (-1)^{00 \cdot 10} & (-1)^{00 \cdot 11} \\ (-1)^{01 \cdot 00} & (-1)^{01 \cdot 01} & (-1)^{01 \cdot 10} & (-1)^{01 \cdot 11} \\ (-1)^{10 \cdot 00} & (-1)^{10 \cdot 01} & (-1)^{10 \cdot 10} & (-1)^{10 \cdot 11} \\ (-1)^{11 \cdot 00} & (-1)^{11 \cdot 01} & (-1)^{11 \cdot 10} & (-1)^{11 \cdot 11} \end{bmatrix}$

So yeah, we can use dot products to denote Knonecker products of Hadamard Gates

Now, assume $s = 11$. We're going to add the columns with the pairs `x` and `x ⊕ s`. In other words, in this case, columns 1 and 4, and, columns 2 and 3.

Adding columns 1 and 4,

$$\frac{1}{2}\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 2 \\ 0 \\ 0 \\ 2 \end{bmatrix}$$

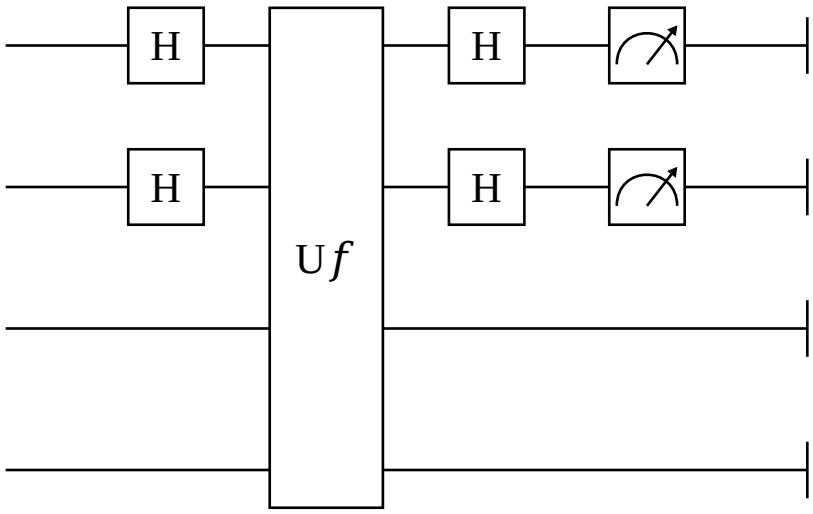
Similarly adding columns 2 and 3,

$$\frac{1}{2}\begin{bmatrix}1\\-1\\1\\-1\end{bmatrix} + \frac{1}{2}\begin{bmatrix}1\\1\\-1\\-1\end{bmatrix} = \frac{1}{2}\begin{bmatrix}2\\0\\0\\-2\end{bmatrix}$$

As the above vectors are state vectors, when doing the operation `x ⊕ s`, we see that some probability amplitudes are getting amplified and some are getting cancelled. If you've studied exponents, you know that $(-1)^{a\cdot(b\oplus s)} = (-1)^{a\cdot b}(-1)^{a\cdot s}$. It means, if $a \cdot s = 0$, then $(-1)^{a\cdot(b\oplus s)} = (-1)^{a\cdot b}$, hence they get added, and if, $a \cdot s = 1$, then $(-1)^{a\cdot(b\oplus s)} = -(-1)^{a\cdot b}$, hence they get cancelled out.

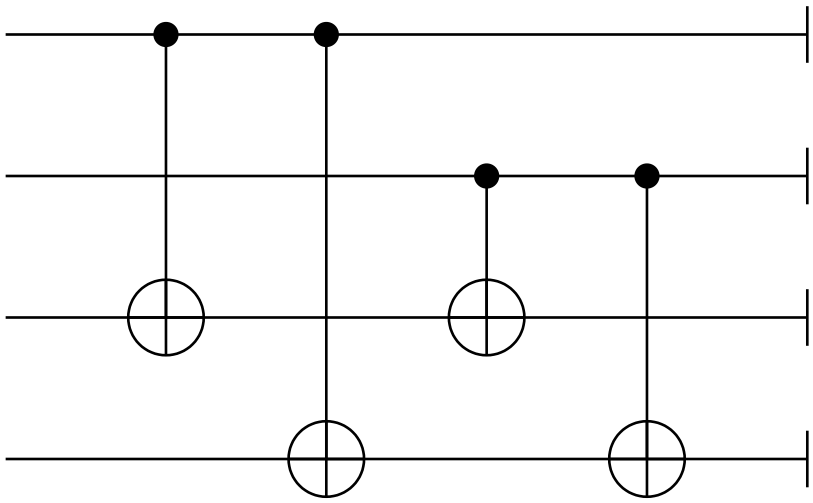
Circuit Implementation

The circuit looks like this, where it takes a string of 0^{2n} as input, and the first n inputs, i.e. x, return x after passing through the circuit, and the next n inputs, i.e. y, return `y ⊕ f(x)`, after passing through the circuit. Let's call this circuit U_f. This is for $n = 2$.

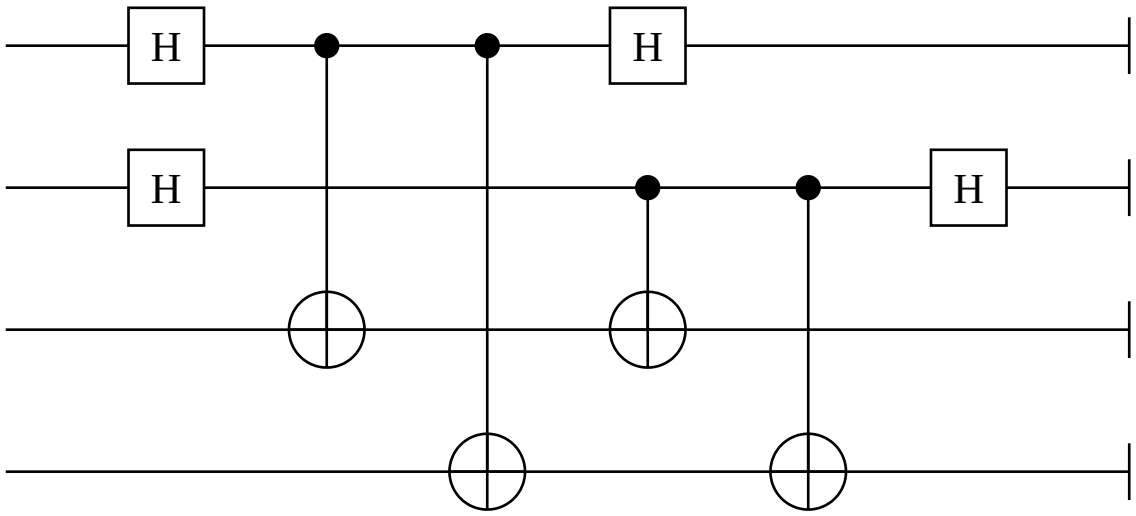



```
s = "11"
```

```
s = string(rand(1 : (2^2 - 1)), base=2, pad=2)
```



```
begin
    if s == "11"
        Uf = chain(4, control(1,3=>X), control(1,4=>X), control(2,3=>X),
control(2,4=>X))
    elseif s == "01"
        Uf = chain(4, control(1,3=>X), control(1,4=>X))
    elseif s == "10"
        Uf = chain(4, control(2,3=>X), control(2,4=>X))
    end
    plot(Uf)
end
```




```
begin
    SimonAlgoCircuit_for_n_2 = chain(4, repeat(H, 1:2), put(1:4=>Uf), repeat(H, 1:2))
    plot(SimonAlgoCircuit_for_n_2)
end
```

```
output =
▸ BitBasis.BitStr{2,Int64}[11 ₍₂₎, 00 ₍₂₎, 11 ₍₂₎, 00 ₍₂₎, 11 ₍₂₎, 11 ₍₂₎, 00 ₍₂₎, 00
```

```
output = zero_state(4) |> SimonAlgoCircuit_for_n_2 |> r->measure(r, 1:2, nshots=1024)
```

The reason it's a hybrid algorithm, is that we got two states, for $n = 2$, which have equal chances of being the secret string s. From here on, we've to classically deduce which of the measured states can be the output. Since s can't be 00, s = 11.

Note that this implementation is specific to $n = 2$

Deduction gets really complicated as n increases, and while its very very unlikely, on real quantum machines, there's a chance that you'll never get the secret string s for any number of runs, or nshots. This algorithm doesn't have much use/application cases either. Shor was inspired by this algorithm to make a general period finding algorithm.

qRAM and Uncomputation

Consider encoding data in qubits. Assume the following array.

```
a = ▸Int64[0,  0,  1,  0,  0]
```

```
a = [0, 0, 1, 0, 0]
```

It'll take 5 qubits to encode this.

```
using Yao, YaoPlots
```

```
▸BitBasis.BitStr{5,Int64}[00100 ₍₂₎]
```

```
ArrayReg(bit"00100") |> r->measure(r)
```

or

```
▸BitBasis.BitStr{5,Int64}[00100 ₍₂₎]
```

```
zero_state(5) |> put(5, 3=>X) |> r->measure(r)
```

Either way, lets assume we've to encode 4 such arrays, in qubits.

```
b = ▸Int64[0,  1,  1,  0,  1]
```

```
b = [0, 1, 1, 0, 1]
```

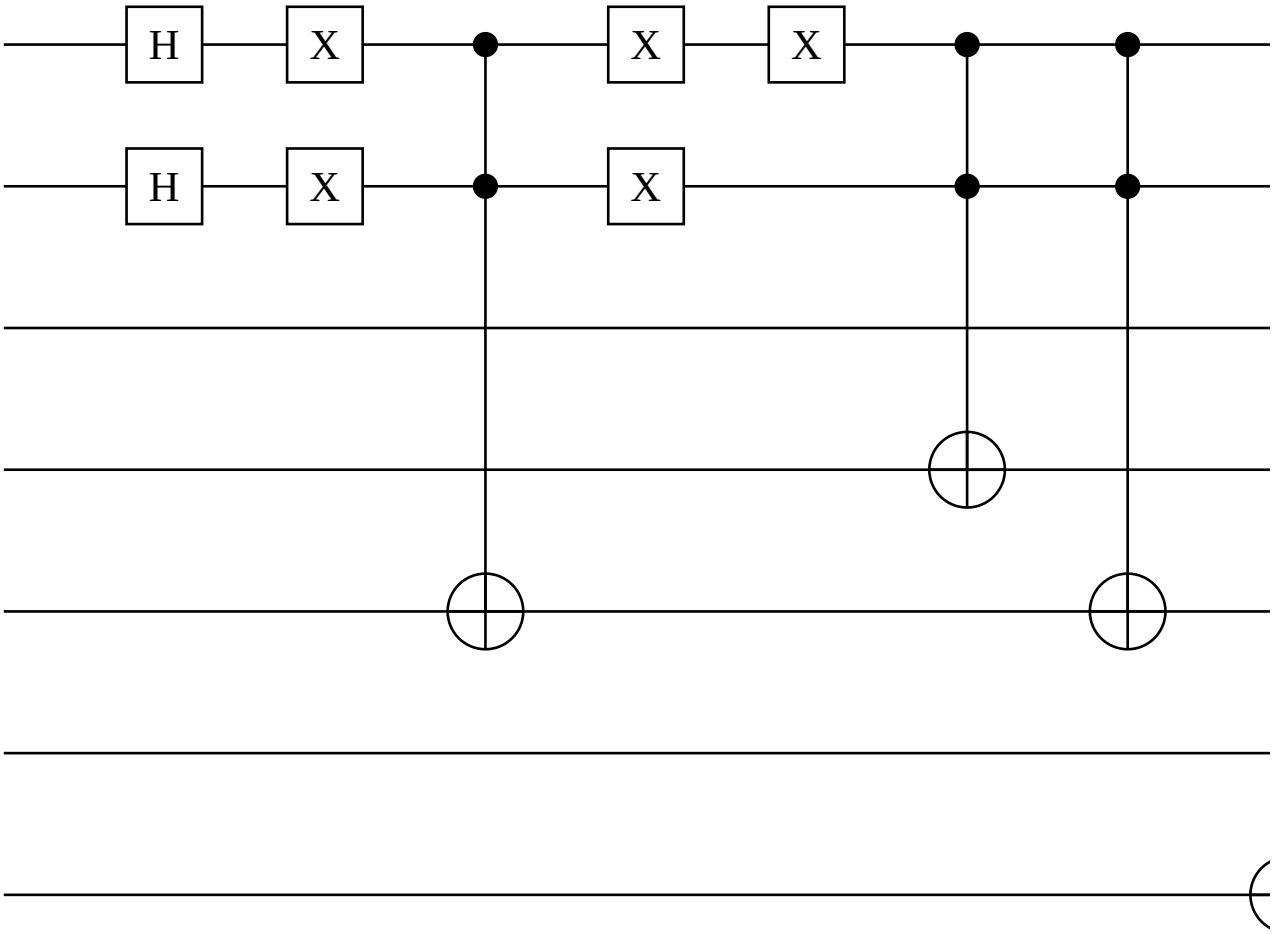
```
c = ▸Int64[1,  1,  0,  0,  0]
```

```
c = [1, 1, 0, 0, 0]
```

```
d = ▸Int64[1,  0,  1,  1,  1]
```

```
d = [1, 0, 1, 1, 1]
```

To encode these 4 arrays, it'd take 20 qubits, judging by the above approach. But using QRAMS, we can use 7 qubits, to encode all the 4 arrays.




```
let
    f(x) = chain(7, [control(1:2, (k+2)=>X) for k in findall(isone, x)])
    global QRAM = chain(7, repeat(H, 1:2), repeat(X, 1:2), f(a), repeat(X, 1:2),
put(1=>X), f(b), put(1=>X), put(2=>X), f(c), put(2=>X), f(d))
    plot(QRAM)
end
```

The first two qubits, are called the address qubits.

- When the address qubits give 00 or 0 in decimal, for the next 5 qubits, we get 00100.
- When the address qubits give 01 or 1 in decimal, for the next 5 qubits, we get 01101.
- When the address qubits give 10 or 2 in decimal, for the next 5 qubits, we get 11000.
- When the address qubits give 11 or 3 in decimal, for the next 5 qubits, we get 10111.

The input to the QRAM is 0000000.

```
output =
▸ BitBasis.BitStr{7,Int64}[1110111 ₍₂₎, 1011010 ₍₂₎, 1110111 ₍₂₎, 1110111 ₍₂₎, 0001101 ₍
```

```
output = zero_state(7) |> QRAM |> r->measure(r, nshots = 1024)
```

The below code records the frequency of measurements.

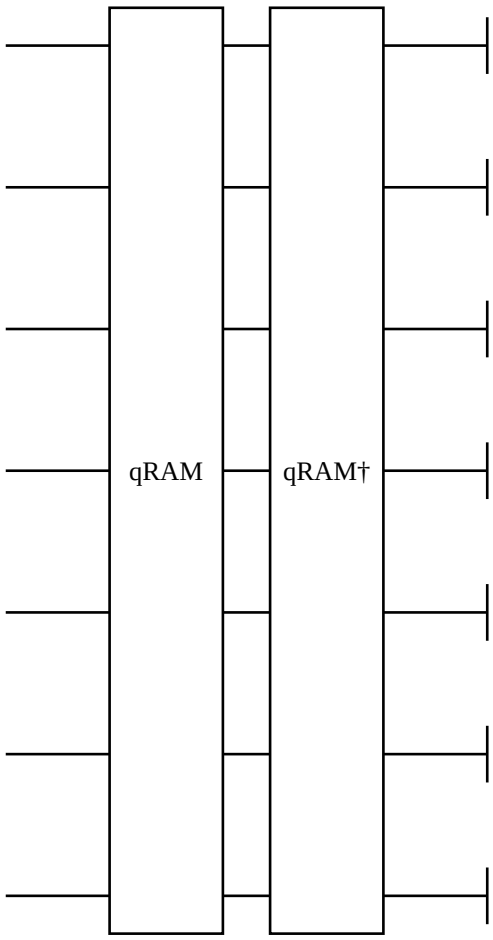
```
String["1011000", "0000100", "0101101", "1110111"]
```

```
begin
    using StatsBase: fit, Histogram
    hist = fit(Histogram, Int.(output), 0:2^7)
    o1 = hist.weights[findall(!iszero, hist.weights)]
    o2 = reverse.(string.(0:(2^7-1), base=2, pad=7)[findall(!iszero, hist.weights)])
end
```

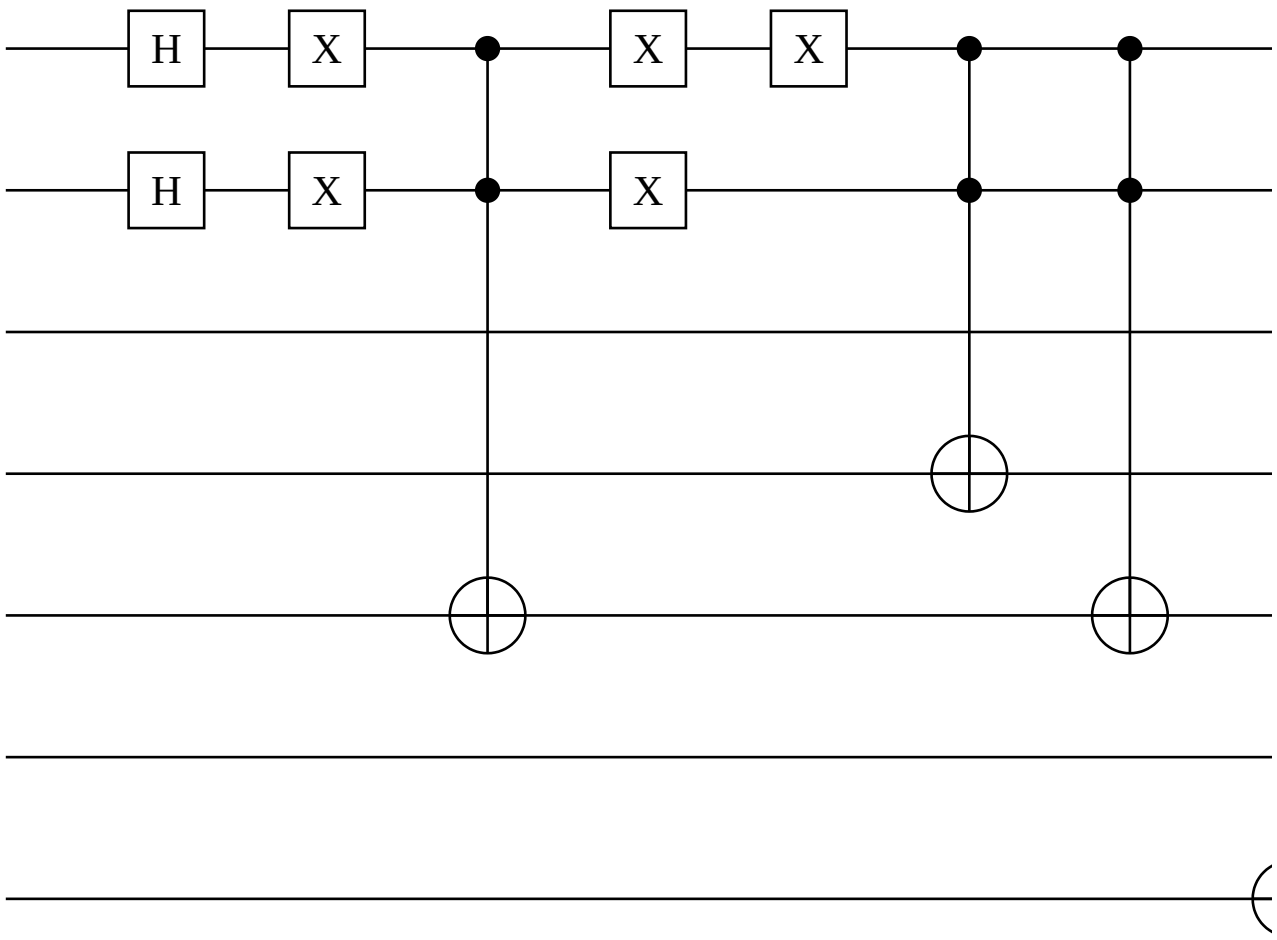
- When address qubits were 10, we got 11000, with the frequency 251.
- When address qubits were 00, we got 00100, with the frequency 240.
- When address qubits were 01, we got 01101, with the frequency 236.
- When address qubits were 11, we got 10111, with the frequency 297.

We use Uncomputation to reverse the everything we do in a circuit to reverse it to its former state. It's often used with qRAMs.

An example would be



```
plot(chain(7, put(1:7 => label(QRAM,"qRAM")), put(1:7 =>
label(Daggered(QRAM),"qRAM†"))))
```




```
begin
    uncomputation = chain(7, put(1:7 => QRAM), put(1:7 => QRAM'))
    plot(uncomputation)
end
```

```
▸ BitBasis.BitStr{7,Int64}[0000000 (2), 0000000 (2), 0000000 (2), 0000000 (2), 0000000 (
```

```
zero_state(7) |> uncomputation |> r->measure(r, nshots=1024)
```

As you can see, we first apply the QRAM circuit, and then its dagger, which undoes the effect.