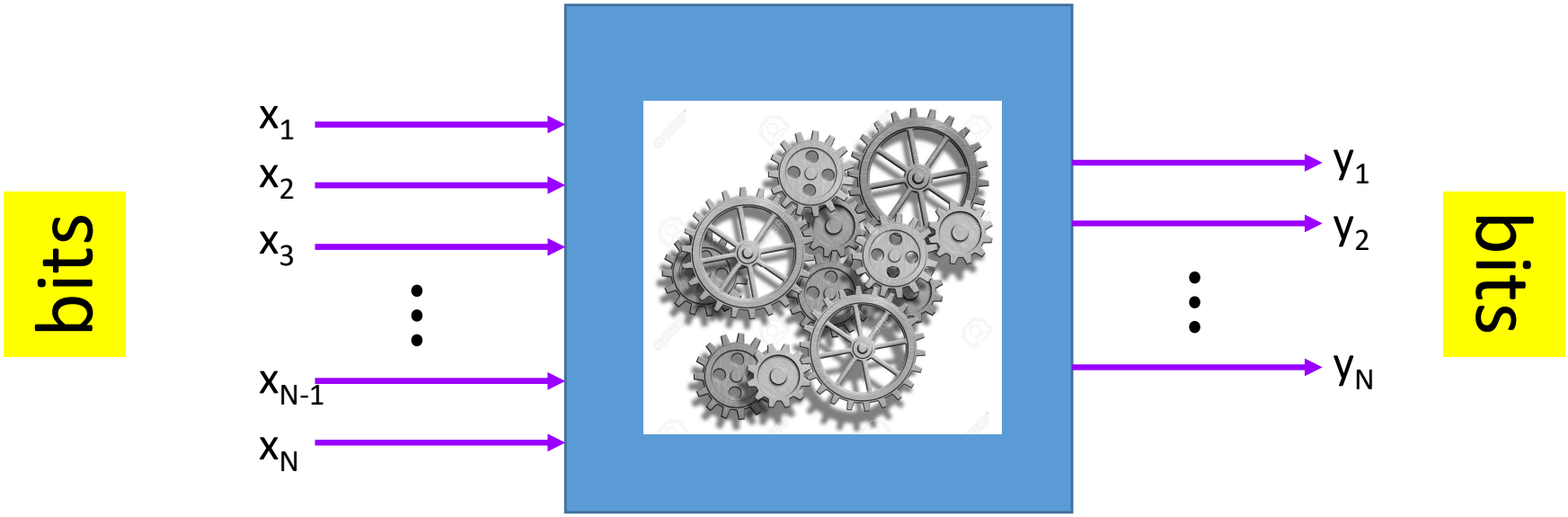
Quantum…

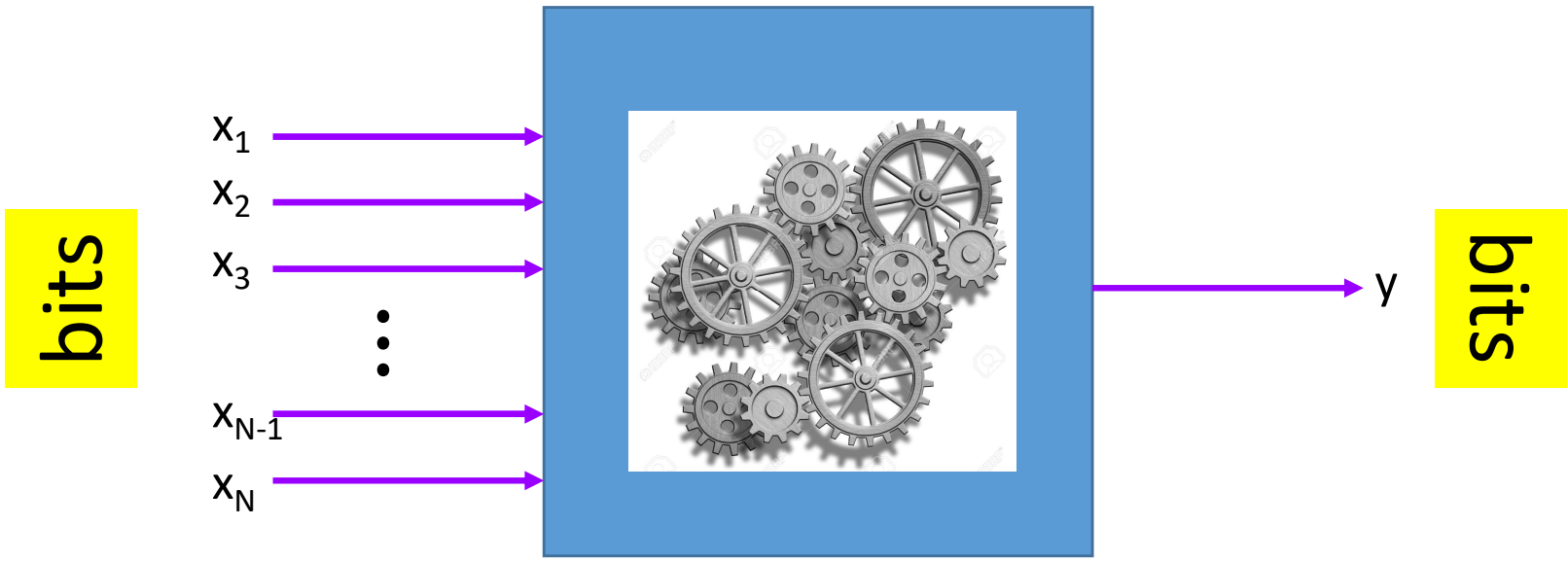
computing

A standard model for Boolean functions (aka algorithms)

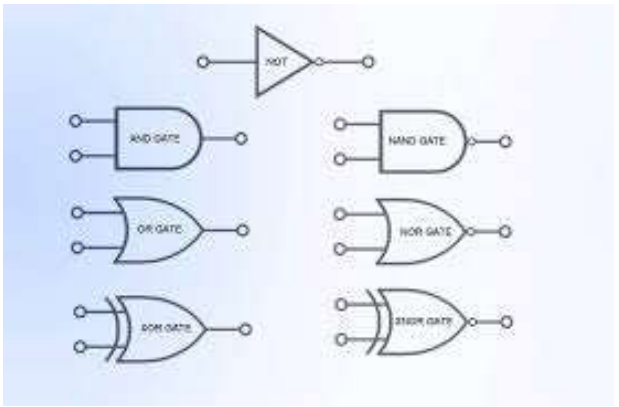


- A bunch of bits go in, and a bunch of bits come out

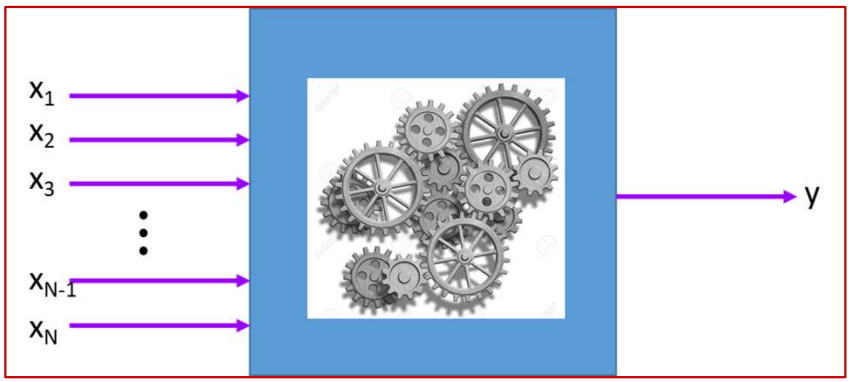
A simpler model for computation

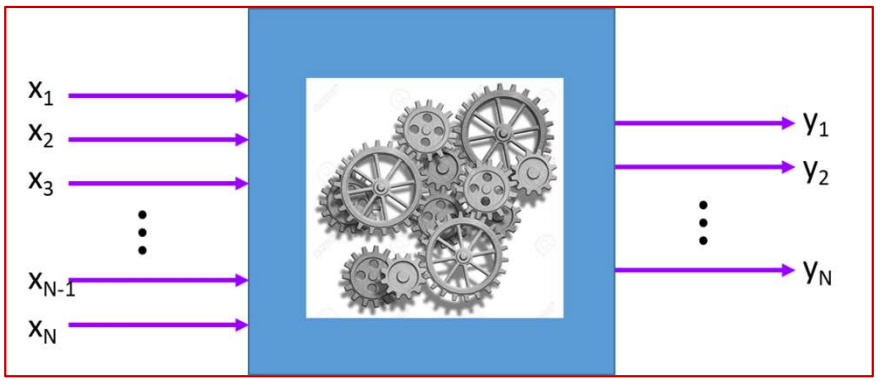


- A bunch of bits go in, and one bit comes out.
- Examples to the right



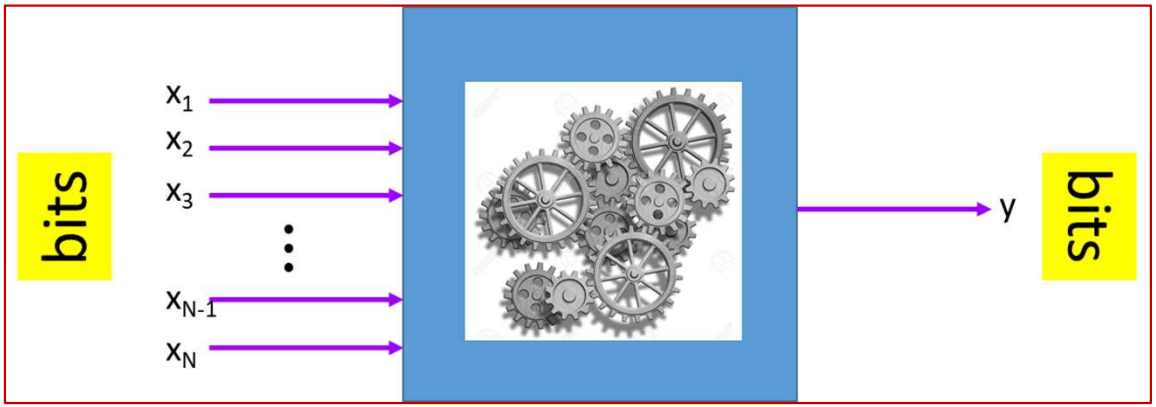
A simpler model for computation

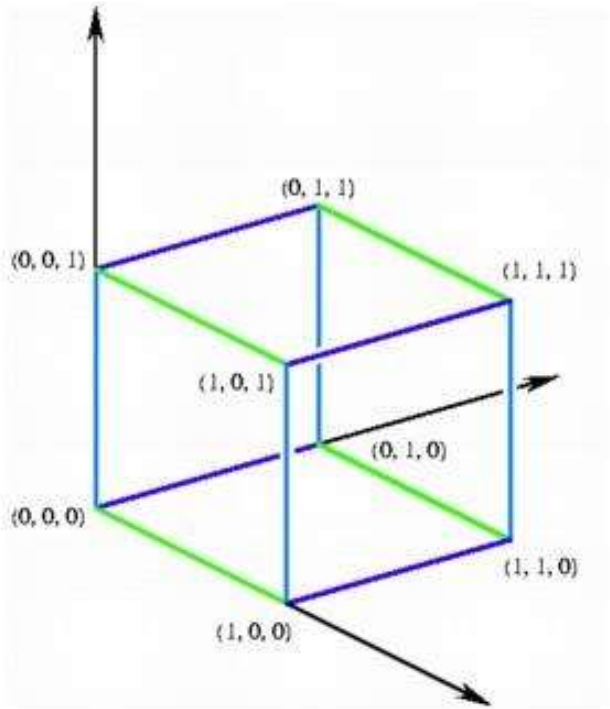




- The model to the left is in fact a generalization of the model to the right
  - How?

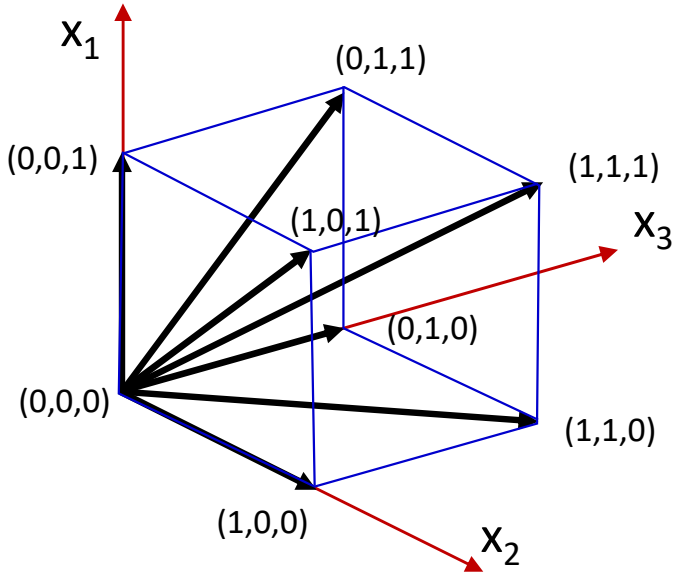
The actual model of computation





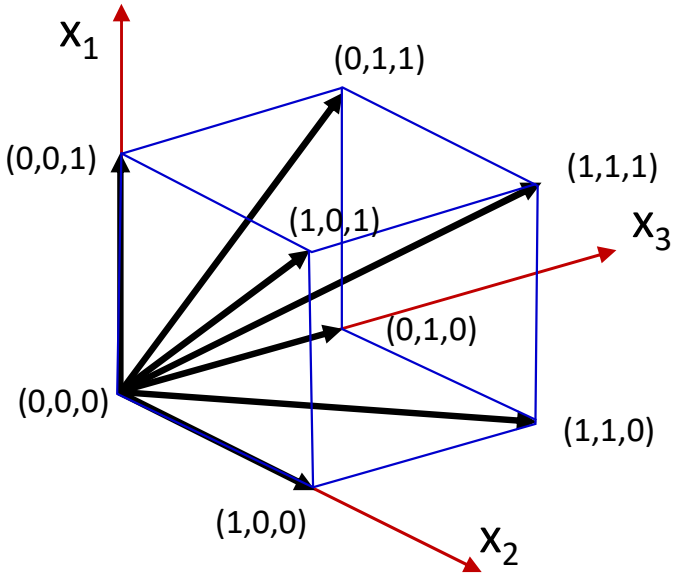
- It is a function of N bits
  - An input goes in, a unique value comes out
- The inputs lie in an N-dimensional space
  - Each input dimension can take only two values [0,1].
  - Other values are not defined
- The entire set of all possible inputs lies on the corners of an N-dimensional hypercube
  - The interior of the cube is infeasible
- The function is defined on the corners of this hypercube

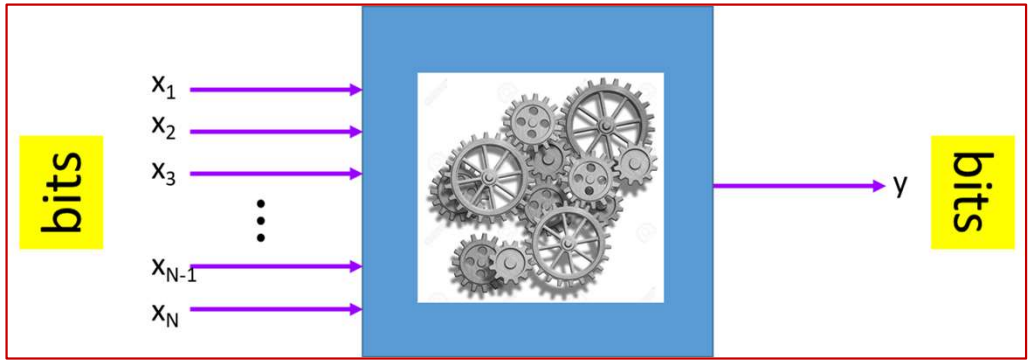
The input space representation



- For a function over N bits, the function is defined over an N-dimensional input space
  - **Each input bit represents an orthogonal direction in the space!**
- Each feasible input combination is an N-dimensional vector in this space
  - The feasible set of inputs is a countable, finite set of 2^N points.
- In order to fully represent an input, N values must be provided
  - One number for each input coordinate

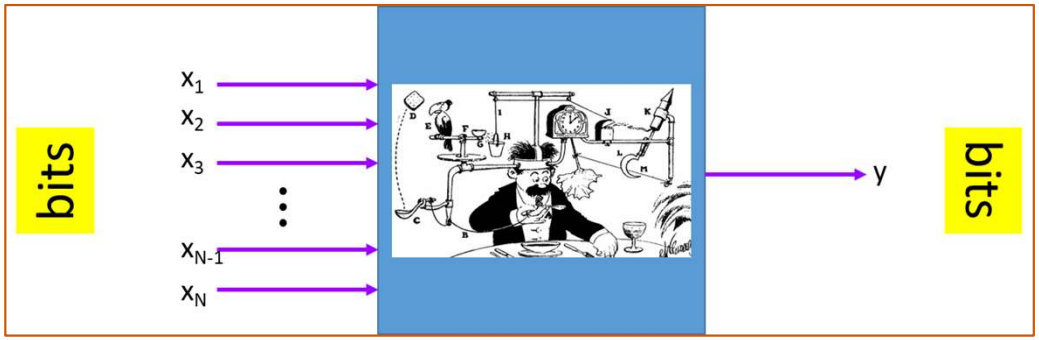
The function

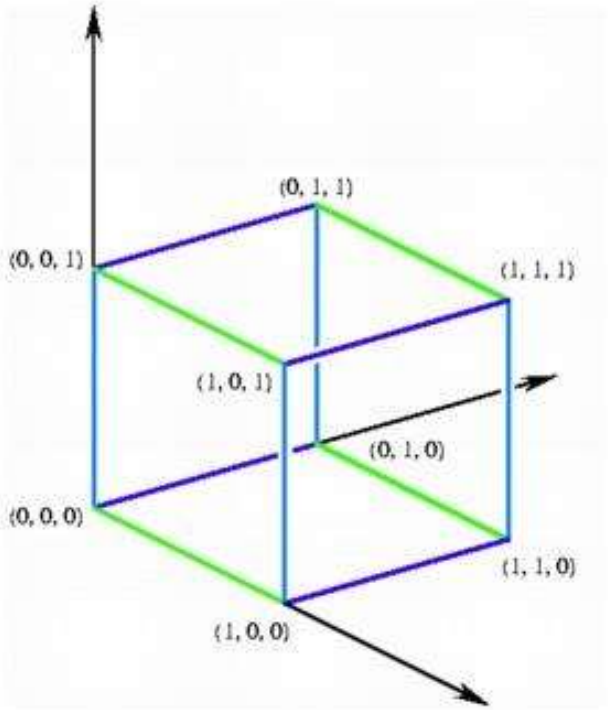




- The algorithm is simply a function that operates on this input space
  - It transforms each input vector to a Boolean value
- Note again that each input vector represents a *single* possible combination of bits
  - When operated on a vector, the function computes the output for that combination of input bits

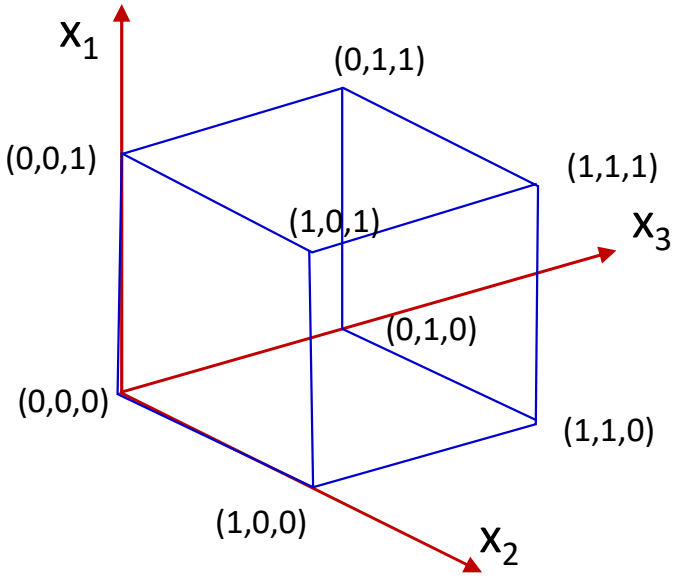
The unknown function problem

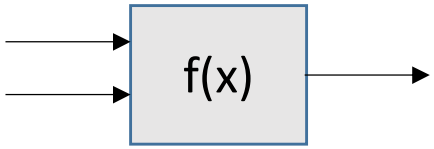




- You are given an uncharacterized function
  - N inputs
  - You know the formulae in it, but don't know what outputs it will produce for any input
- You must characterize the function fully
- How many measurements must you make?
  - "Measurement" : provide an input and note the output

“Discovering” a 2-bit function

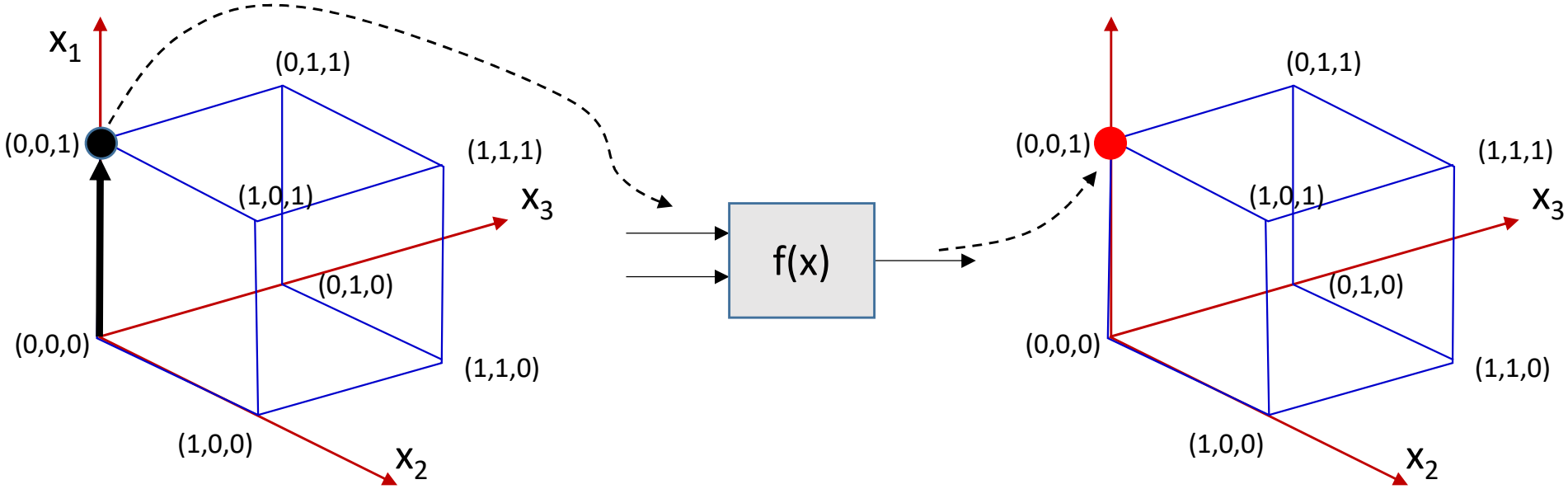




Determine this function fully!

- Find out how it responds to any input

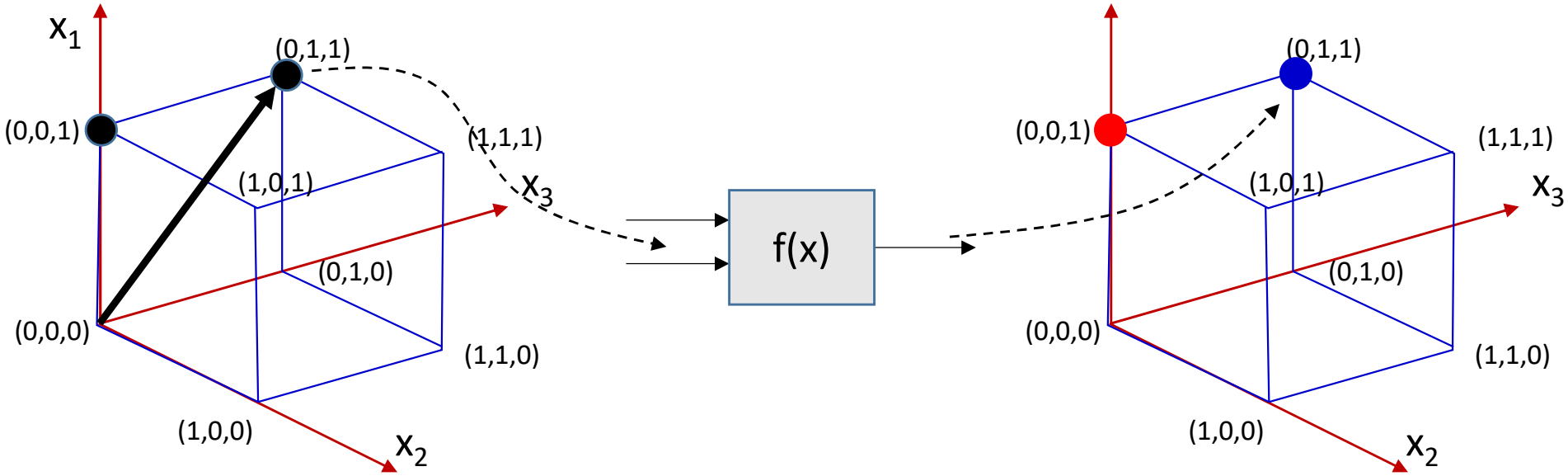
"Discovering" a 2-bit function



Determine this function fully!

- Pass each possible input vector through the function and compute its response

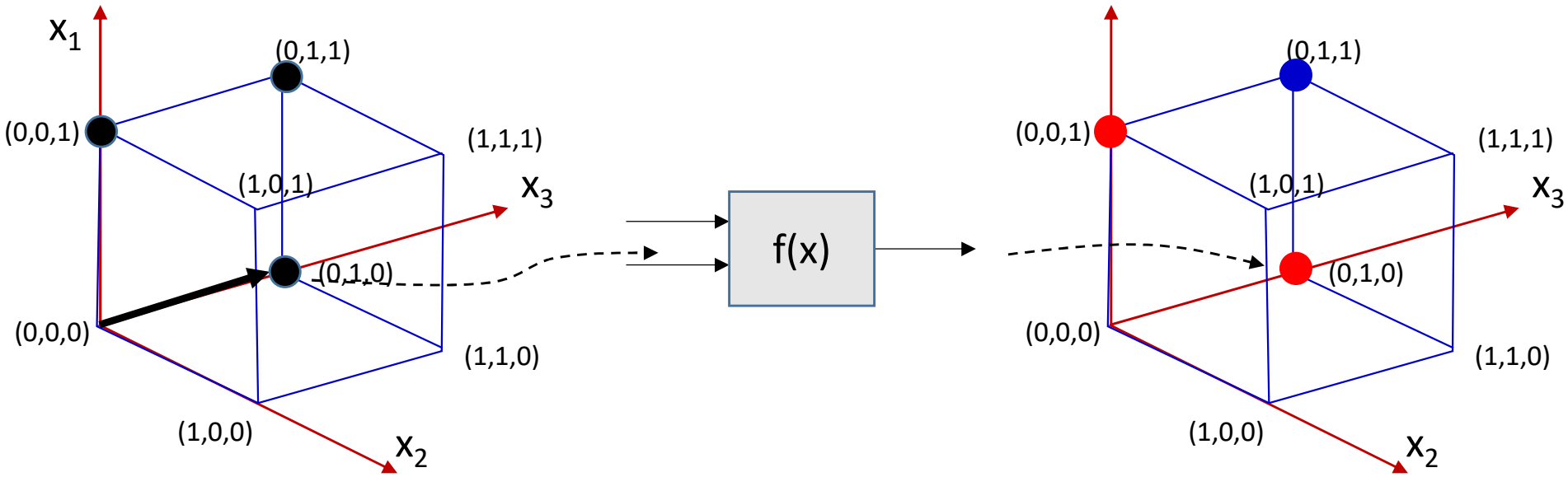
"Discovering" a 2-bit function



Determine this function fully!

- Pass each possible input vector through the function and compute its response

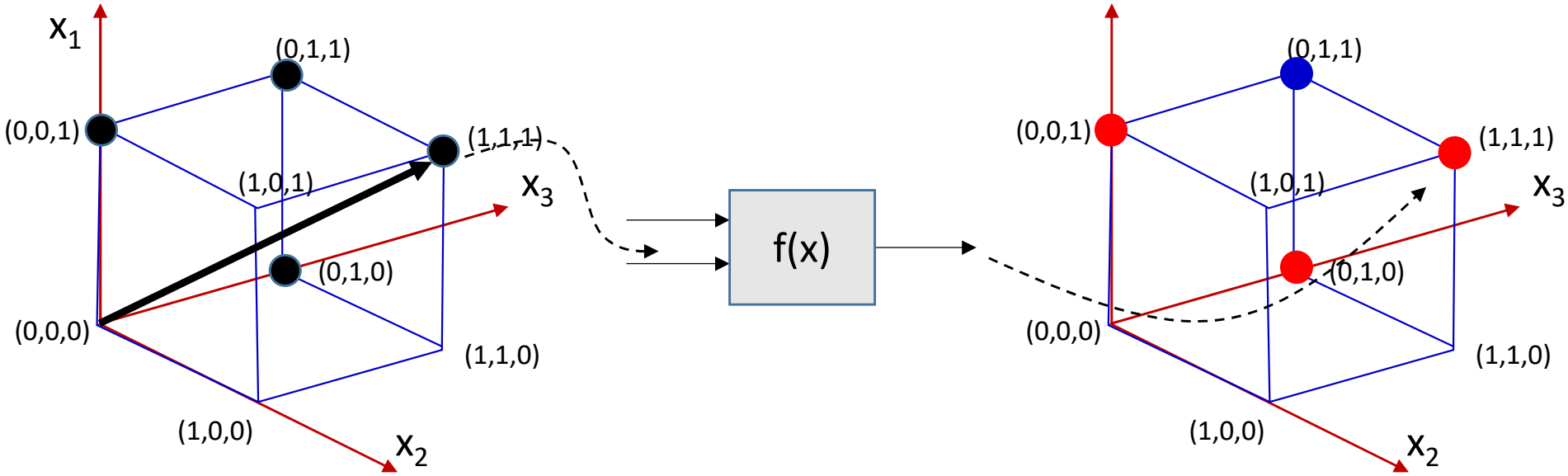
“Discovering” a 2-bit function



Determine this function fully!

- Pass each possible input vector through the function and compute its response

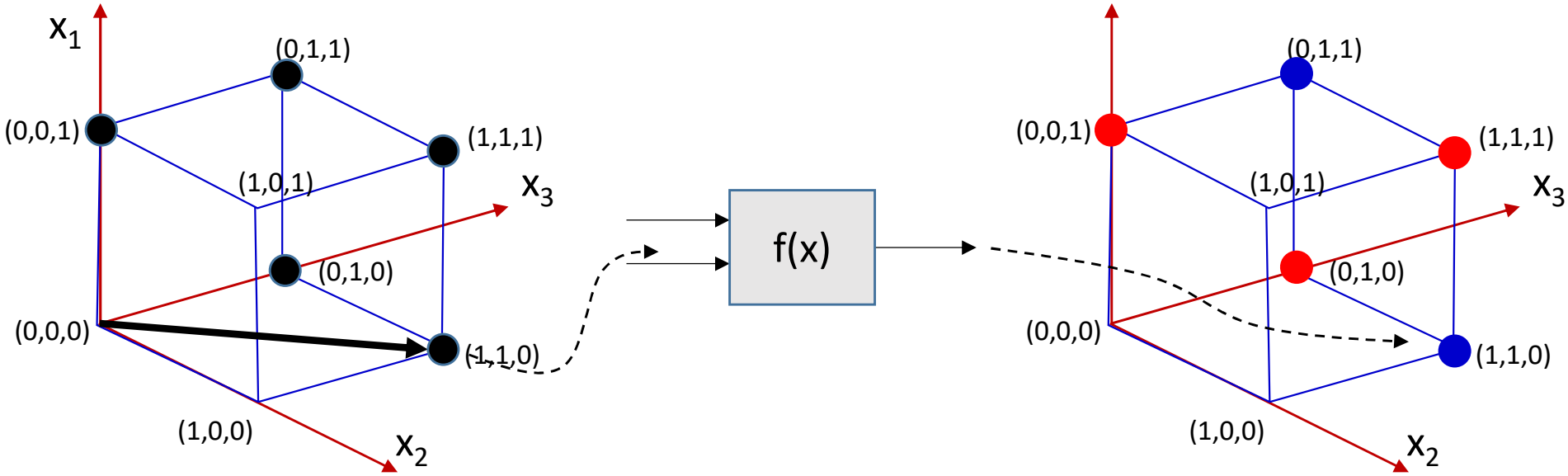
"Discovering" a 2-bit function



Determine this function fully!

- Pass each possible input vector through the function and compute its response

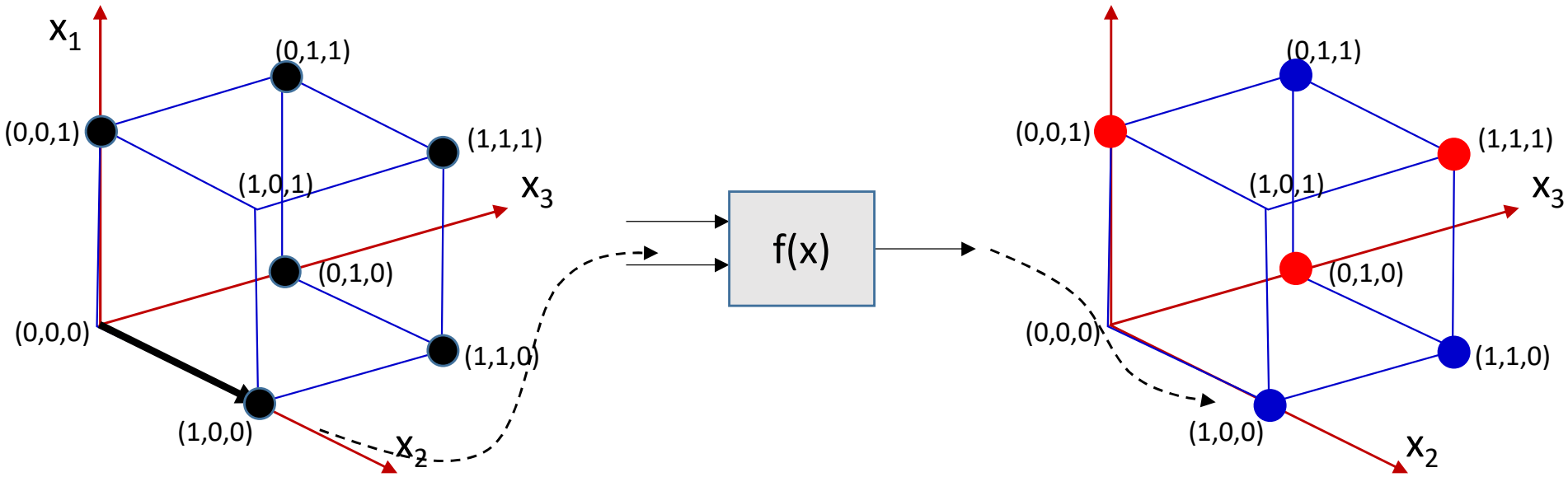
"Discovering" a 2-bit function



Determine this function fully!

- Pass each possible input vector through the function and compute its response

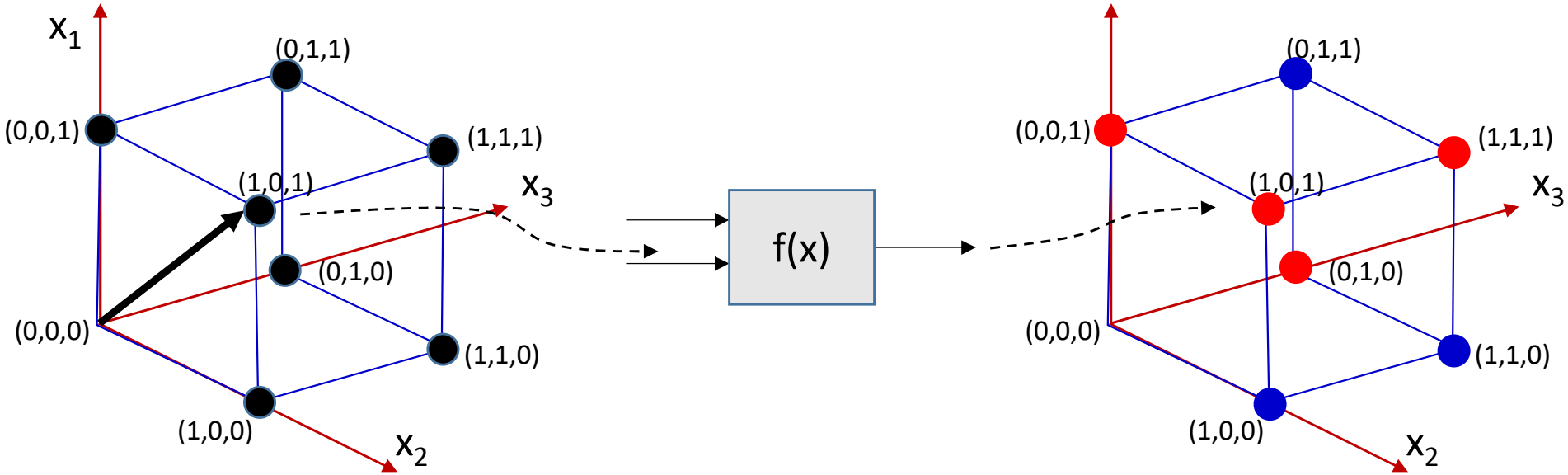
“Discovering” a 2-bit function



Determine this function fully!

- Pass each possible input vector through the function and compute its response

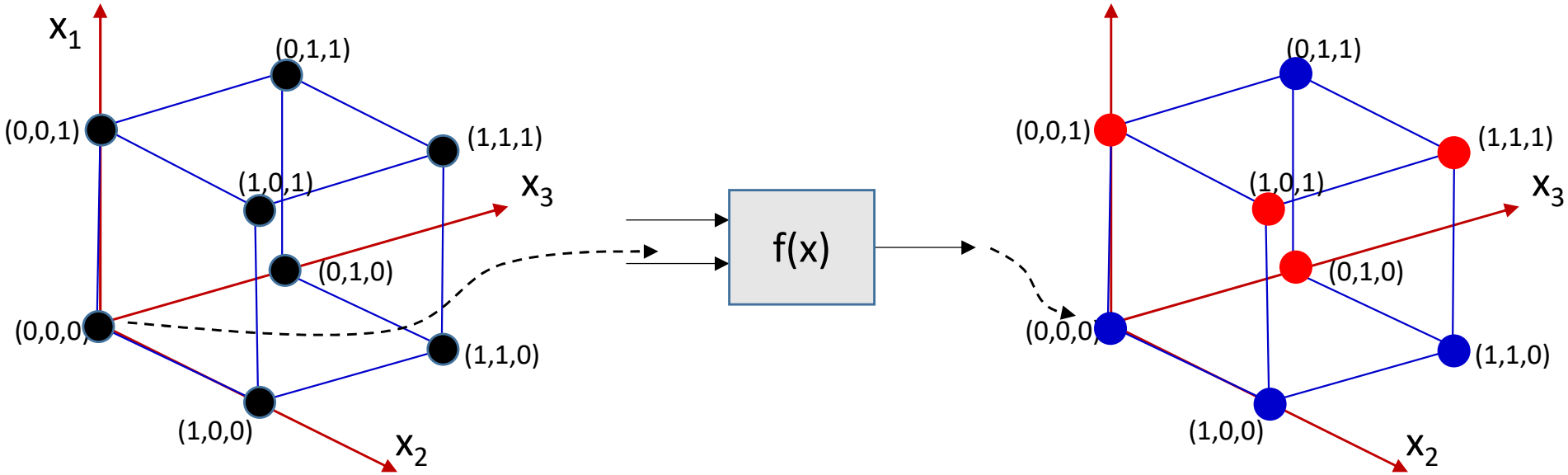
"Discovering" a 2-bit function



Determine this function fully!

- Pass each possible input vector through the function and compute its response

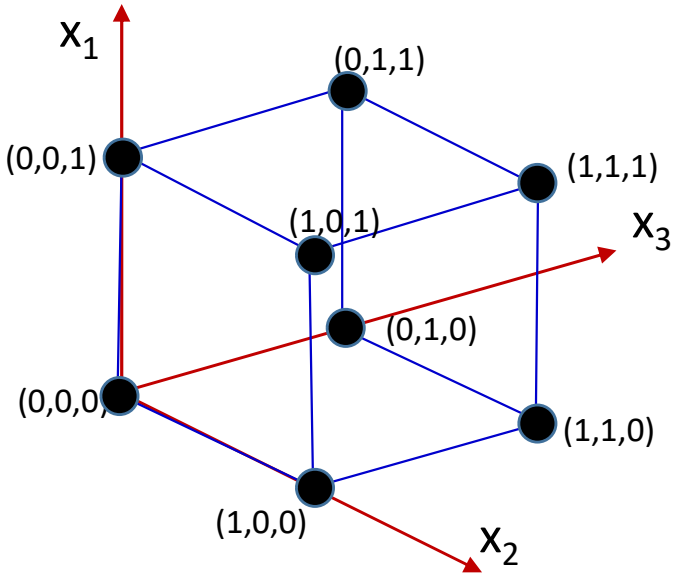
"Discovering" a 2-bit function

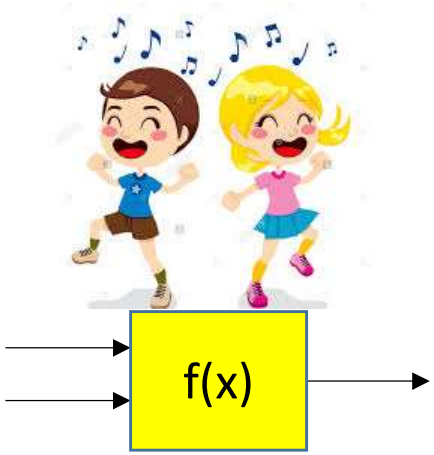


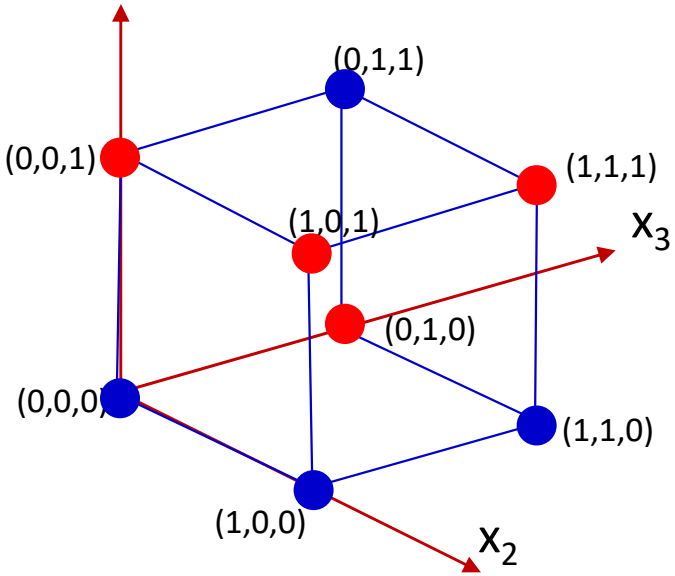
Determine this function fully!

- Pass each possible input vector through the function and compute its response

“Discovering” a 2-bit function

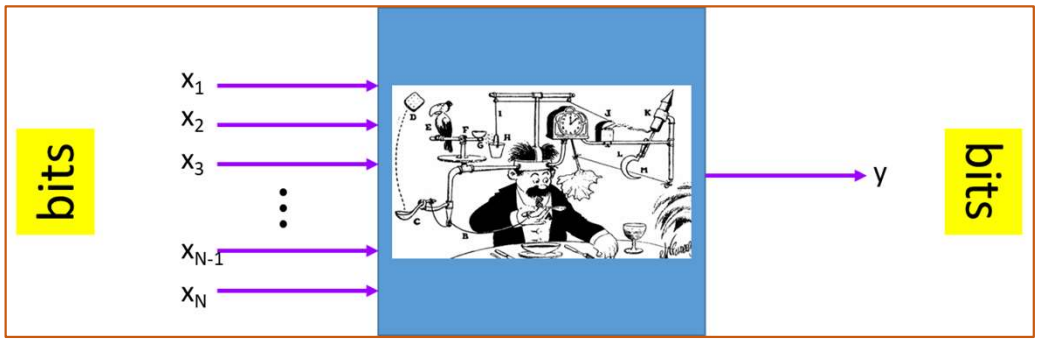


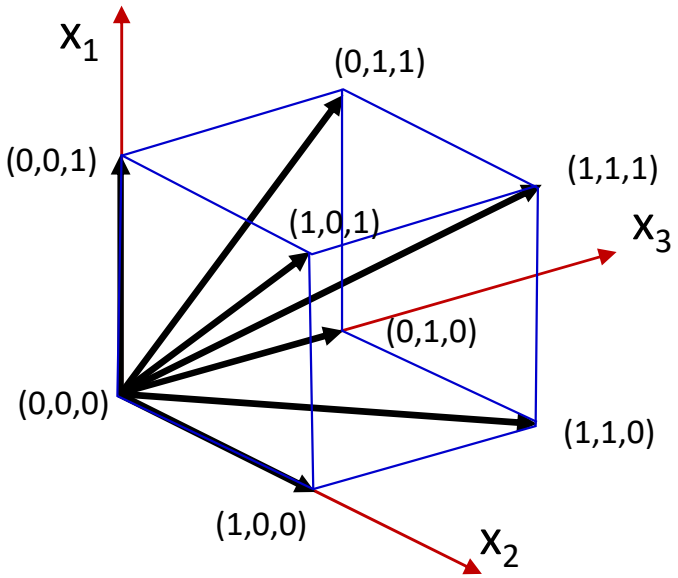




- Pass each possible input vector through the function and compute its response
- *And now its known*

The N-bit unknown function problem

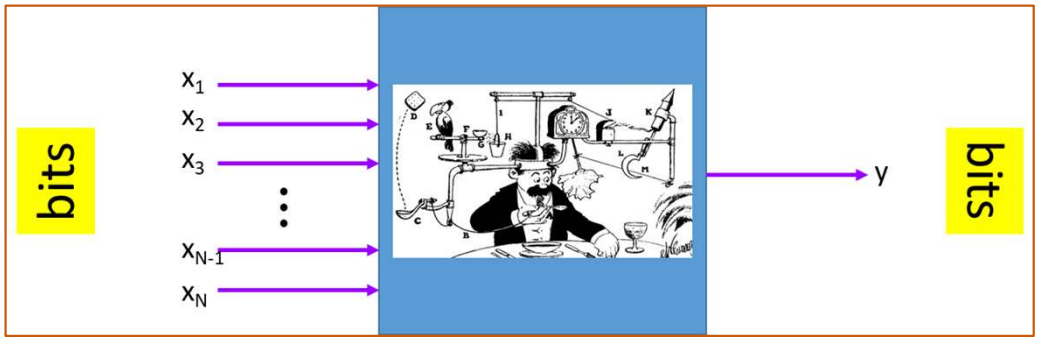


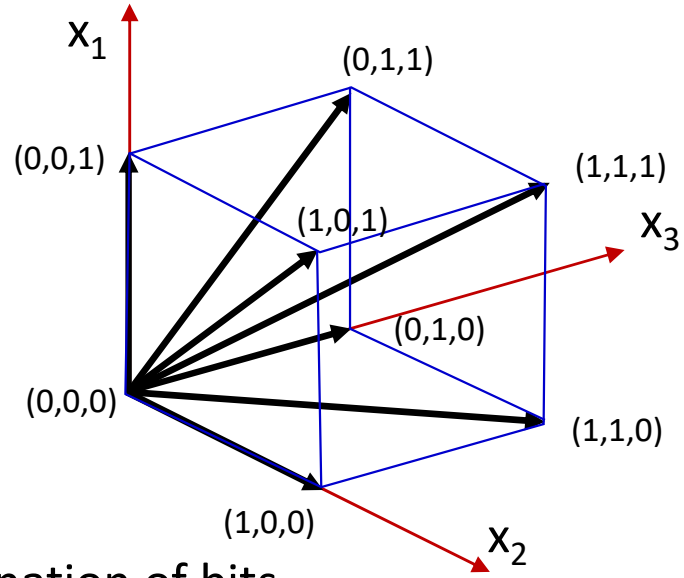


- Recall: For a function over N bits, the function is defined over an N-dimensional input space

- The feasible set of inputs is a countable, finite set of 2^N vectors.
- The function must be individually evaluated at each of these 2^N vectors to define it fully
  - Requiring 2^N evaluations
  - This cannot be reduced

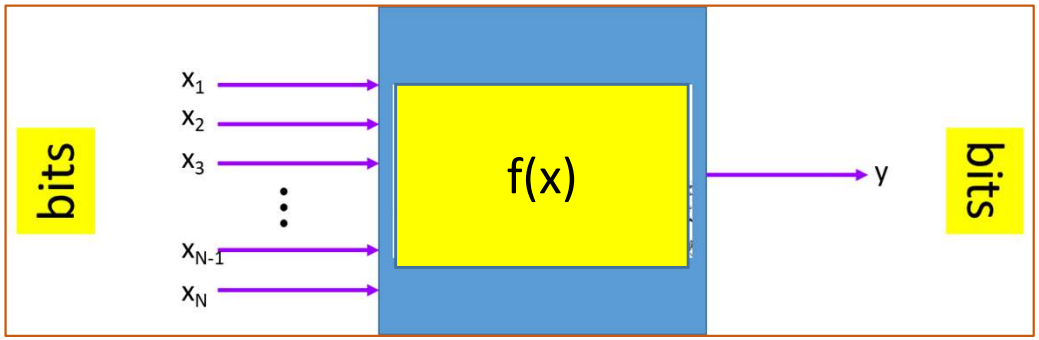
Why this limitation?

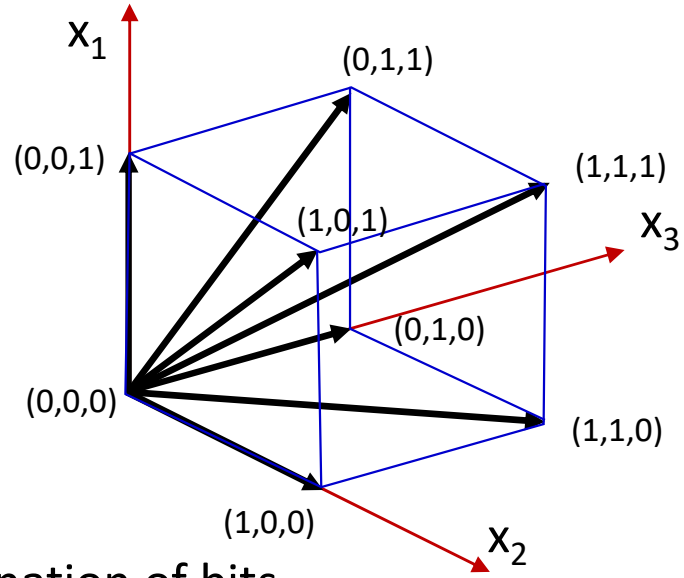




- A vector in the space represents only a *single* combination of bits
- The function operates on individual vectors. So, each operation produces the output for a *single* combination of bits
- To determine the function fully, the function must be evaluated on *every* feasible vector in the input space
  - All 2^N feasible input vectors

Why this limitation?





- A vector in the space represents only a *single* combination of bits
- The function operates on individual vectors. So, each operation produces the output for a *single* combination of bits
- To determine the function fully, the function must be evaluated on *every* feasible vector in the input space
  - All 2^N feasible input vectors

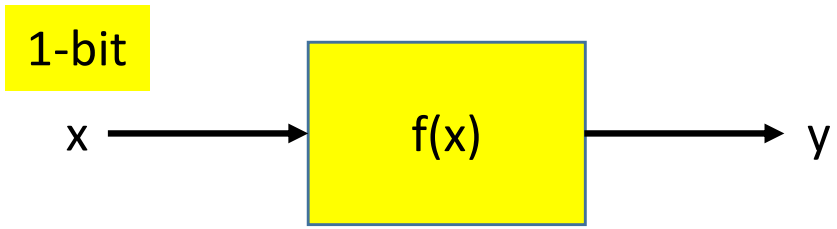
- Can we *change* the mathematical paradigm itself to change this scenario?
  - Change the representation paradigm itself to make things more efficient, somehow
  - **E.g have a single function computation compute the outputs for multiple combinations of input bits?**

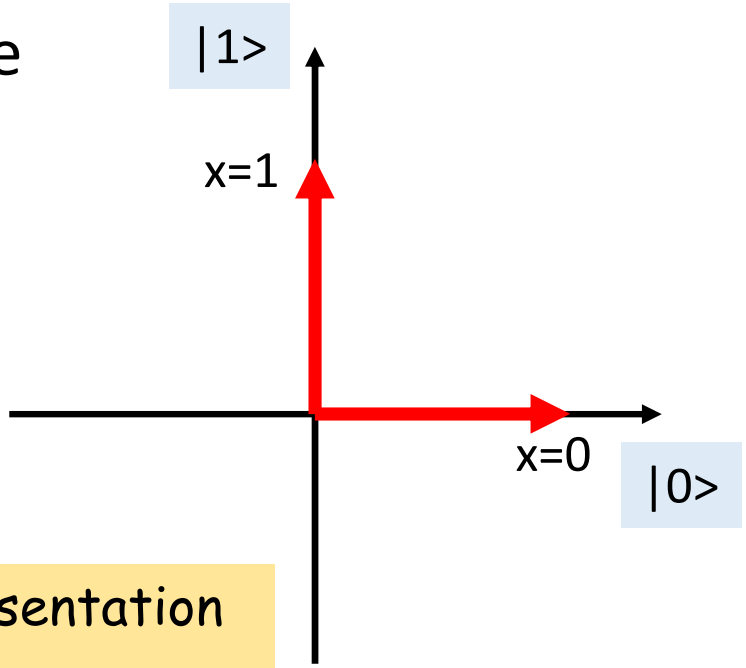
A different approach

- **Modify the representation**

- Instead of each *bit* representing a (orthogonal) coordinate direction we will make each *combination of bits* represent a orthogonal coordinate direction!

- Even 1 bit is now in a 2-D input space

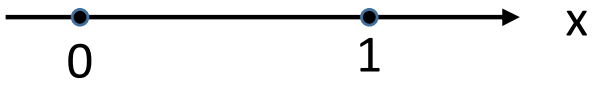




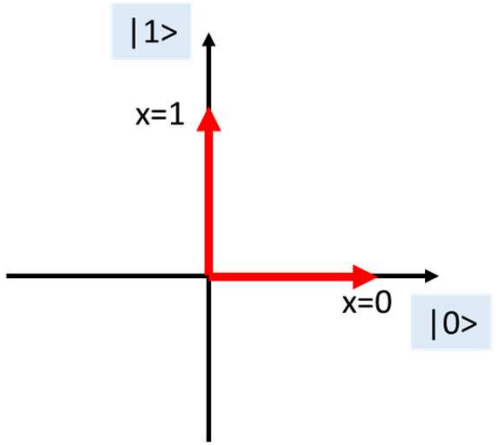
Note that this a fundamentally different representation from standard representations!

A different approach

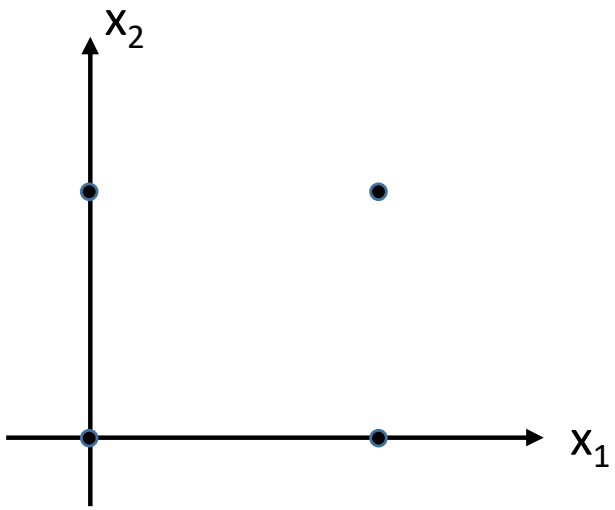
- **1 bit: Old representation**



1 bit: New representation



- **2 bits: Old representation**



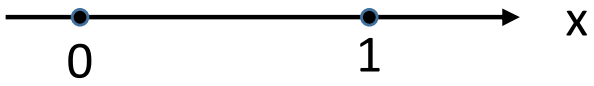
2 bits: New representation

Can't really visualize

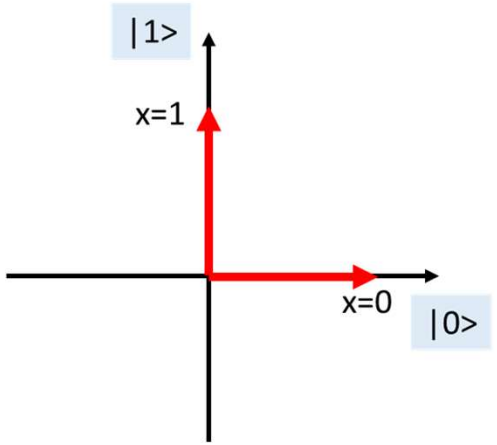
(why)?

A different approach

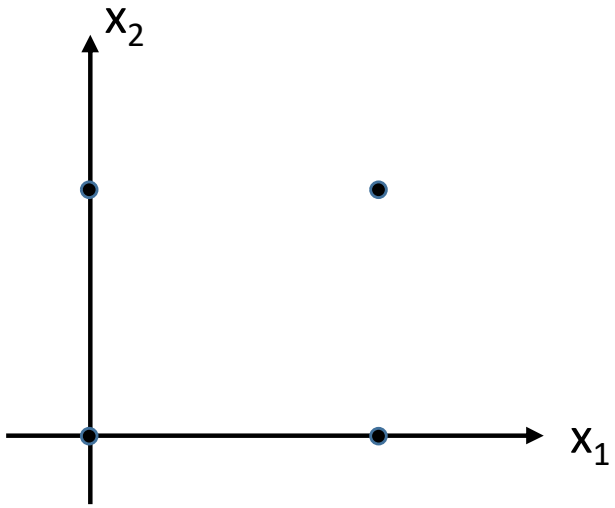
- **1 bit: Old representation**



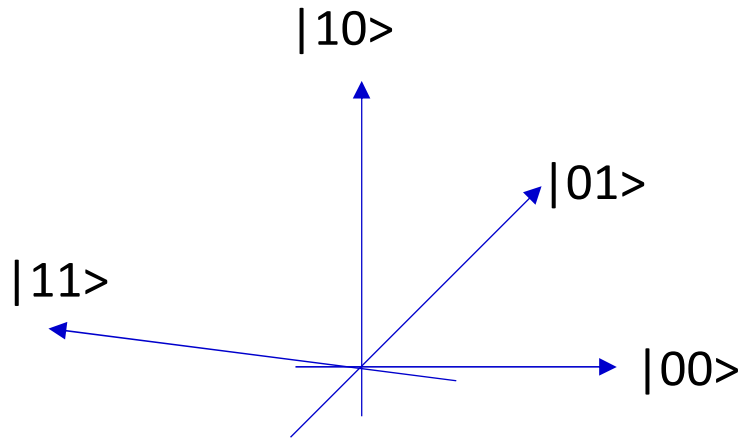
1 bit: New representation



- **2 bits: Old representation**



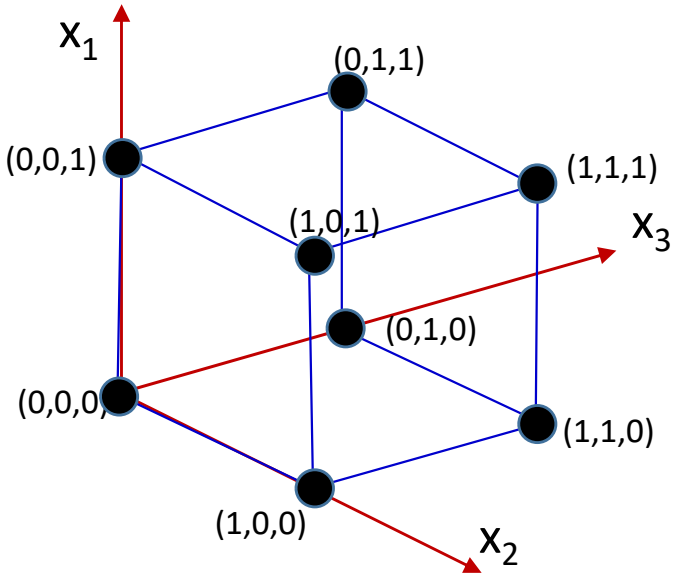
2 bits: New representation



Lame attempt at visualizing 4D

A different approach

- **3 bits: Old representation**



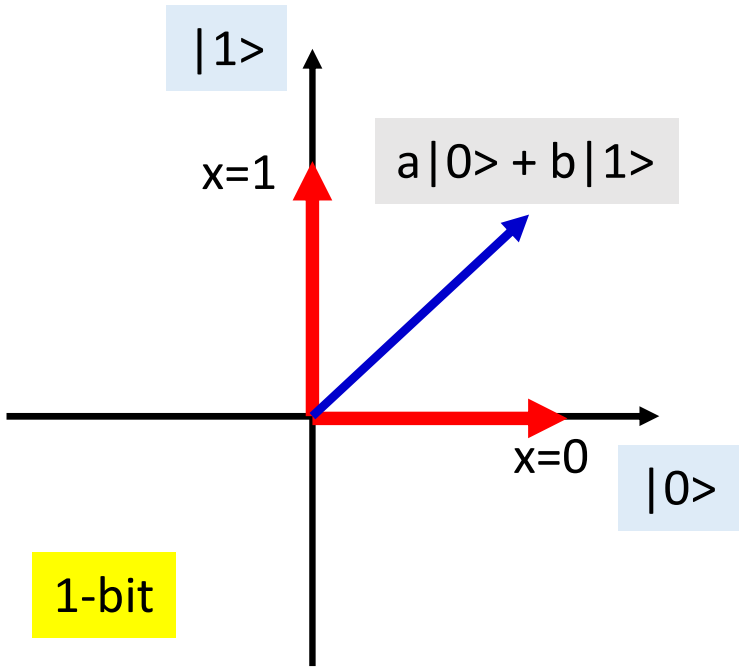
3 bits: New representation

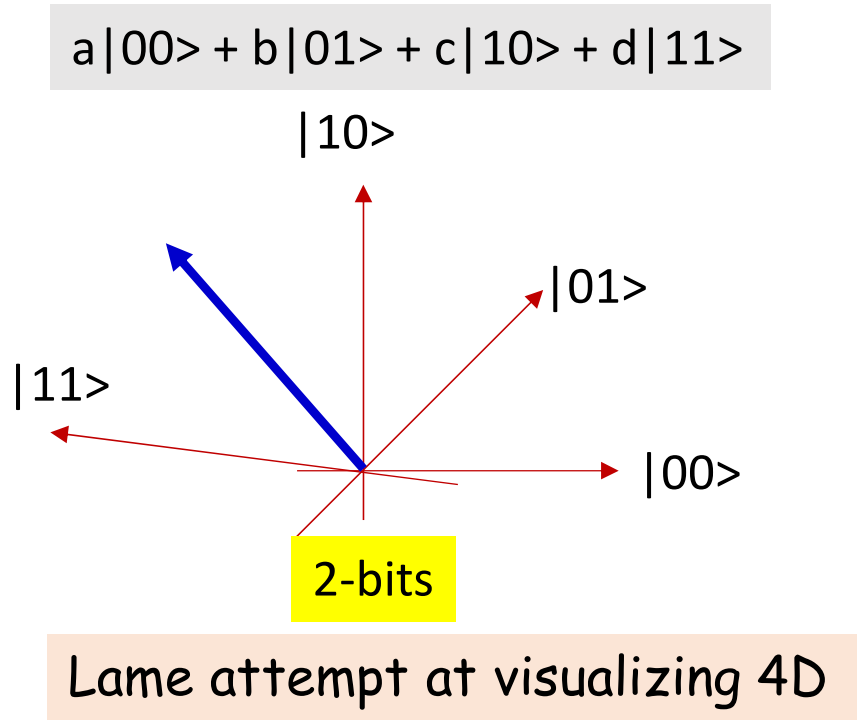
Can't really visualize

(why)?

The modified representation

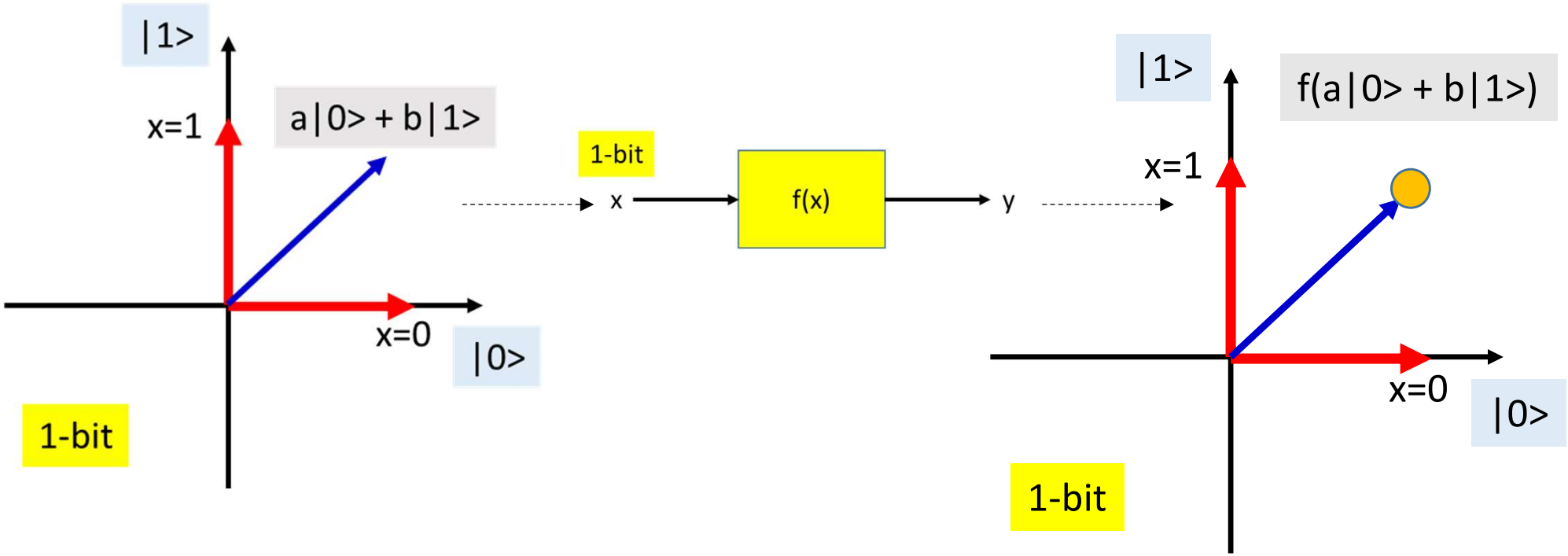
- A vector in this representation is a linear, *unambiguous* combination of *all* input bit patterns
- Unambiguous because we set each bit pattern to be an *orthogonal* direction to every other pattern





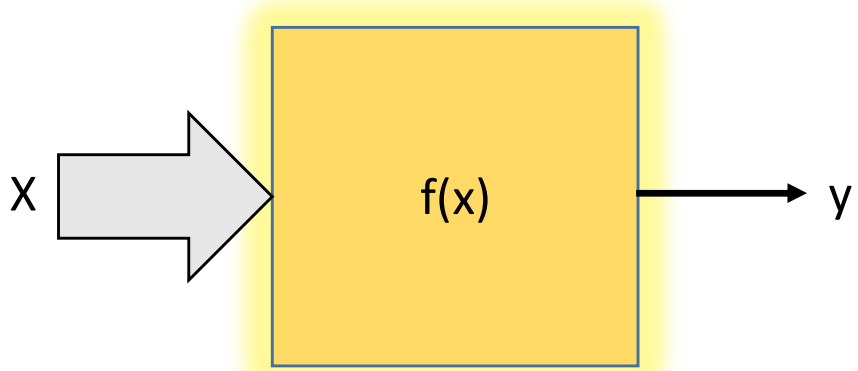
Lame attempt at visualizing 4D

Working with the modified representation



- When the function operates on a vector, it operates on a linear combination of all possible input values
  - How does this help?
  - Not directly, we need to make some assumptions

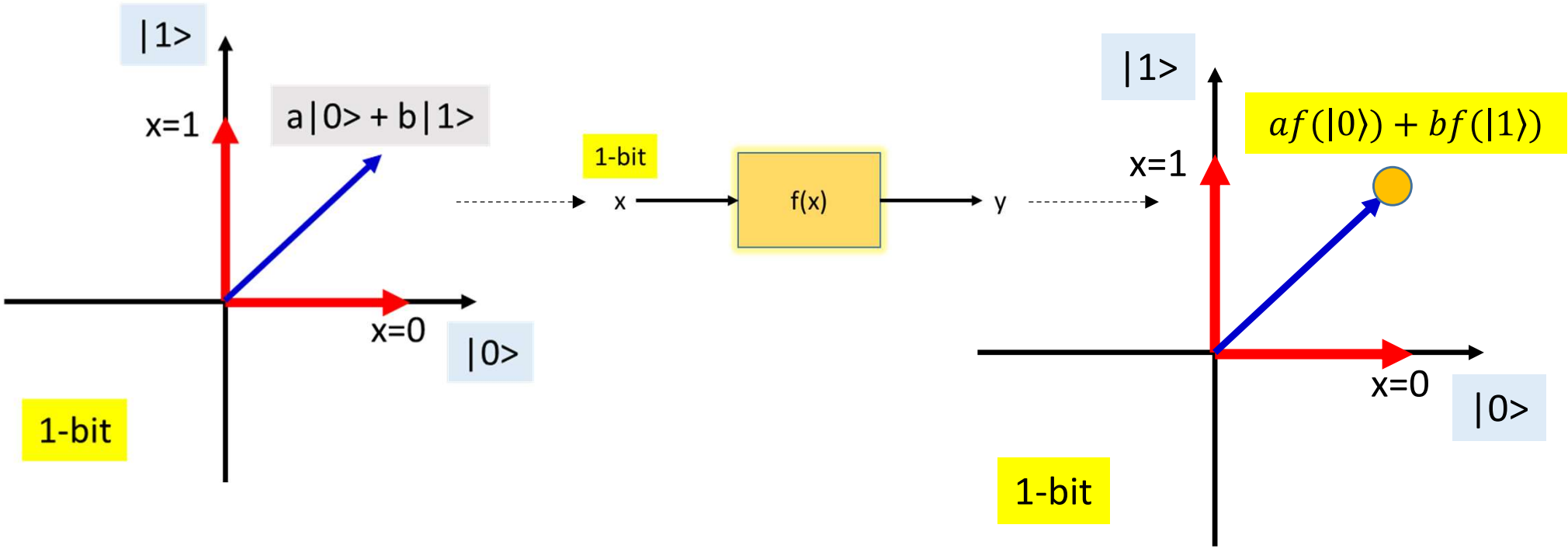
Adjustment 1: Linear functions



- A function $f(x)$ is *linear* if
$$f(ax + by) = af(x) + bf(y)$$
for any two scalars a and b

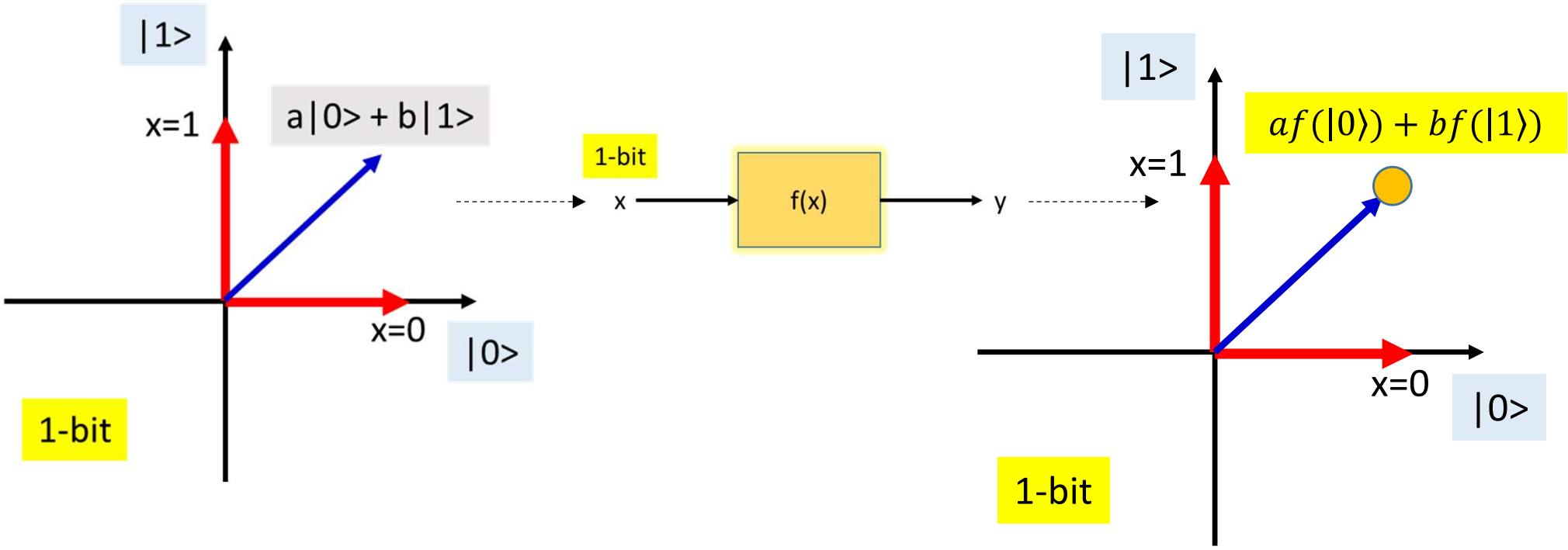
- We assume our function $f(x)$ to be linear
  - As it happens, Boolean functions can always be cast as linear operators

For linear f(x)



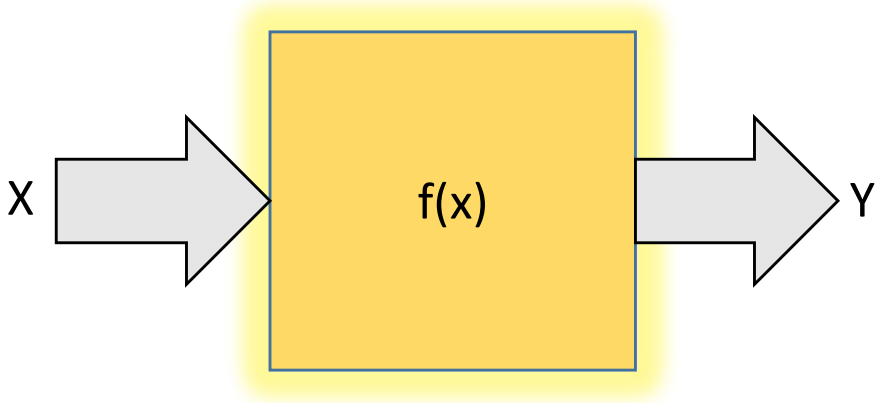
- For linear f(x), the output is a linear combination of the outputs for the individual bit patterns!
- By simply measuring the output at a *single* input vector, we obtain the combined outputs for *all* input bit patterns!
  - *One evaluation!!* (As opposed to 2^N)
- But are we done yet?

For linear f(x)



- The combined output for the individual bit patterns doesn't tell us what the output is for any *single* bit pattern
  - I.e you can't divine $f(|0\rangle)$ and $f(|1\rangle)$ from $af(|0\rangle) + bf(|1\rangle)$
- We need the output to maintain the distinction
  - The function must separately compute the output for each input combination and keep the answers distinct

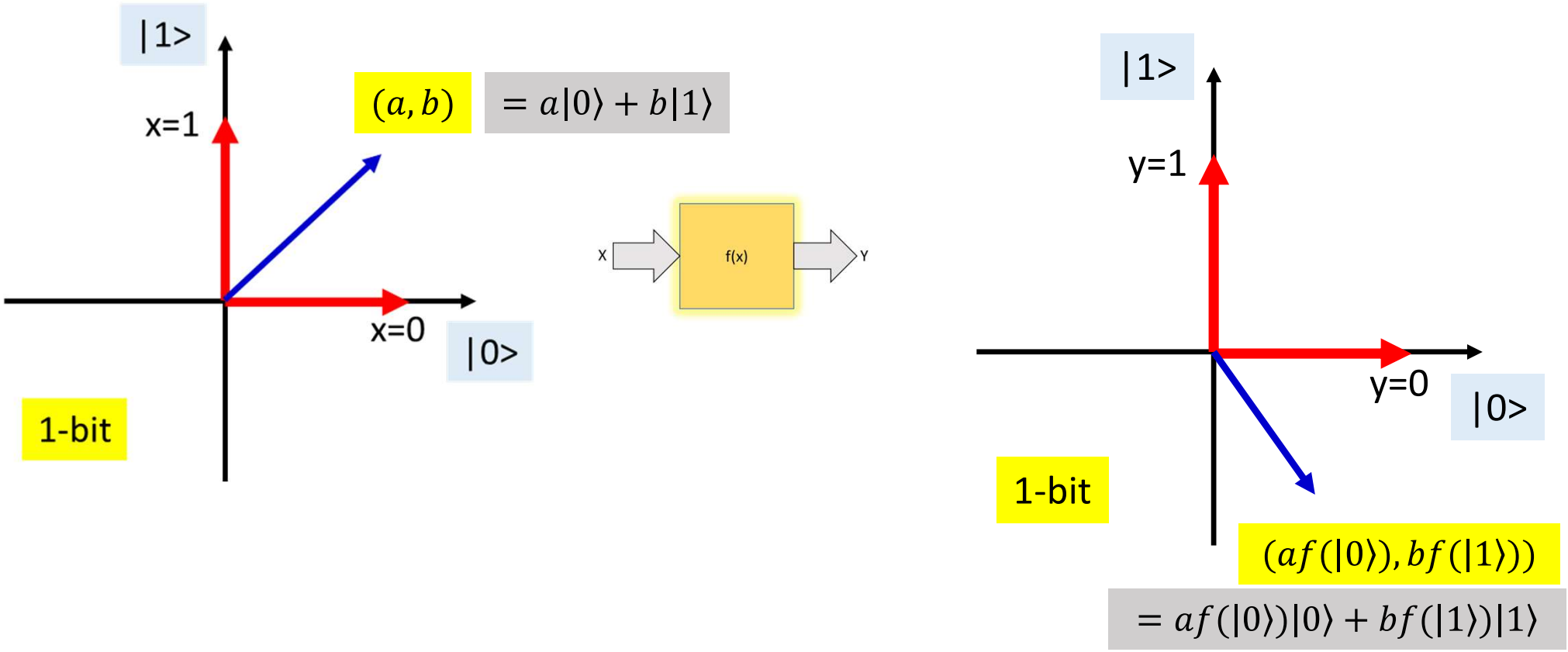
Adjustment 2: Vector functions



- The output too is a vector

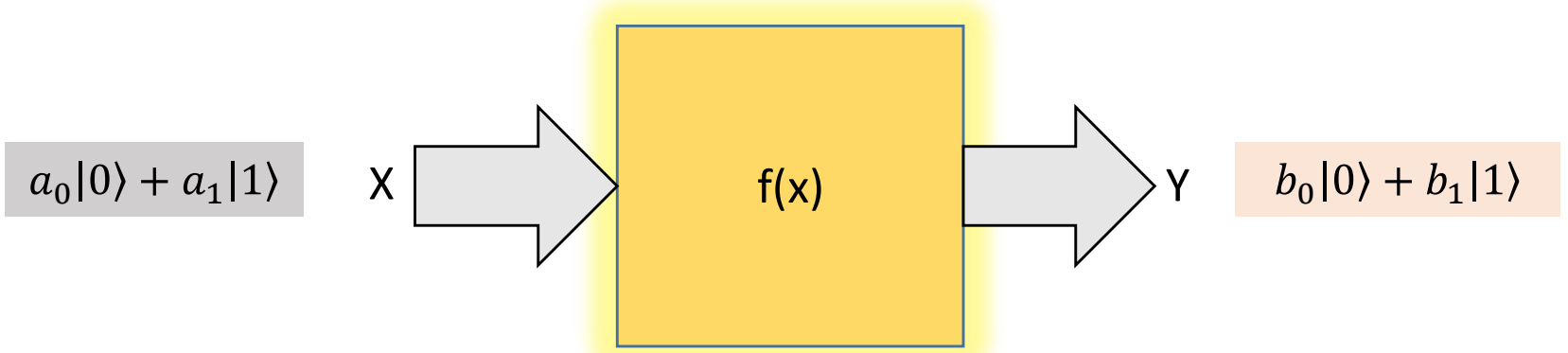
$$f(a|0\rangle + b|1\rangle) = af(|0\rangle)|0\rangle + bf(|1\rangle)|1\rangle$$

For linear f(x)



- Instead of merely computing a value at the vector, the function moves the vector to a new position, where the individual components represent the responses to individual bit patterns
  - Amazingly enough, every Boolean function can be recast in this manner

Adjustment 2: Vector functions



- $f(.)$is a linear transform

$$f \circ \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

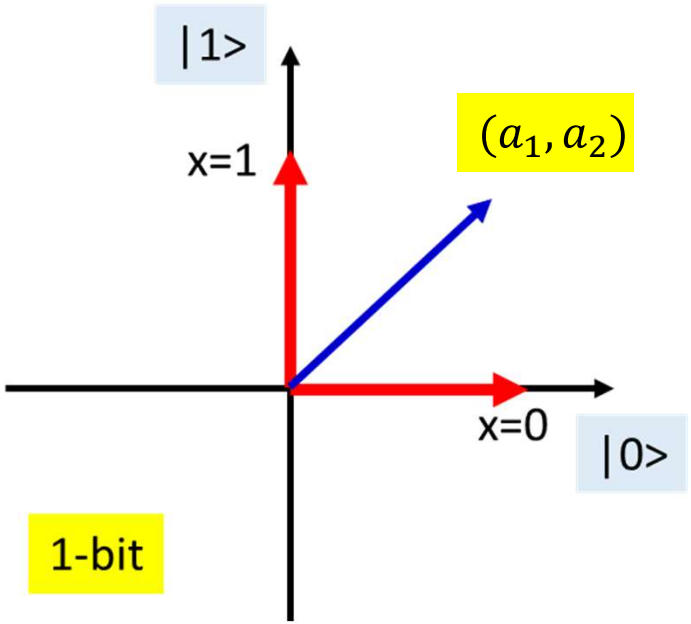
- More generally:

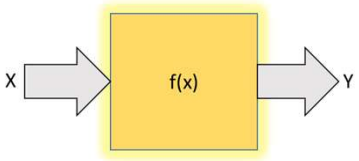
$$f \circ \begin{bmatrix} a_{00\ldots0} \\ a_{00\ldots1} \\ \vdots \\ a_{11\ldots1} \end{bmatrix} = \begin{bmatrix} b_{00\ldots0} \\ b_{00\ldots1} \\ \vdots \\ b_{11\ldots1} \end{bmatrix}$$

In other words $f()$ is a matrix operator.

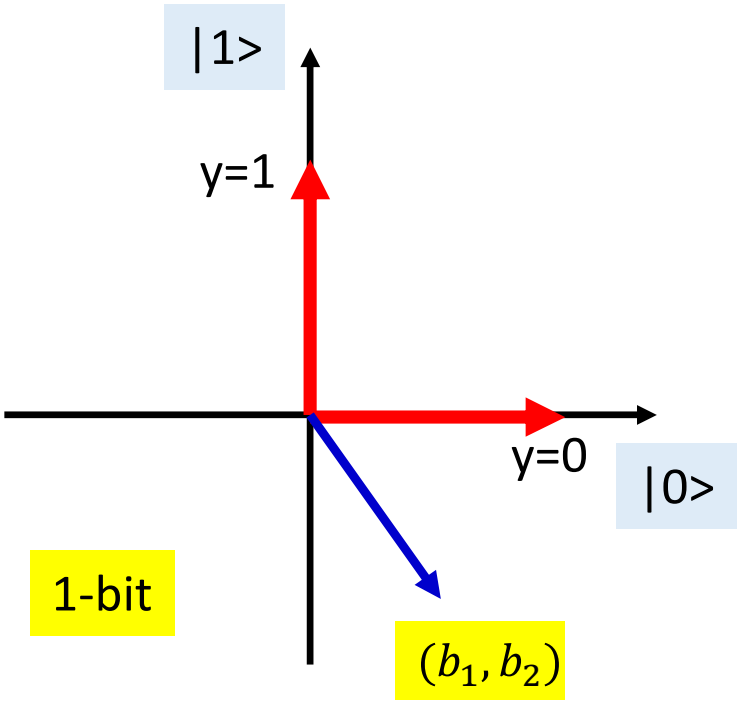
Note that this is a linear operation

For f(x) to retain information



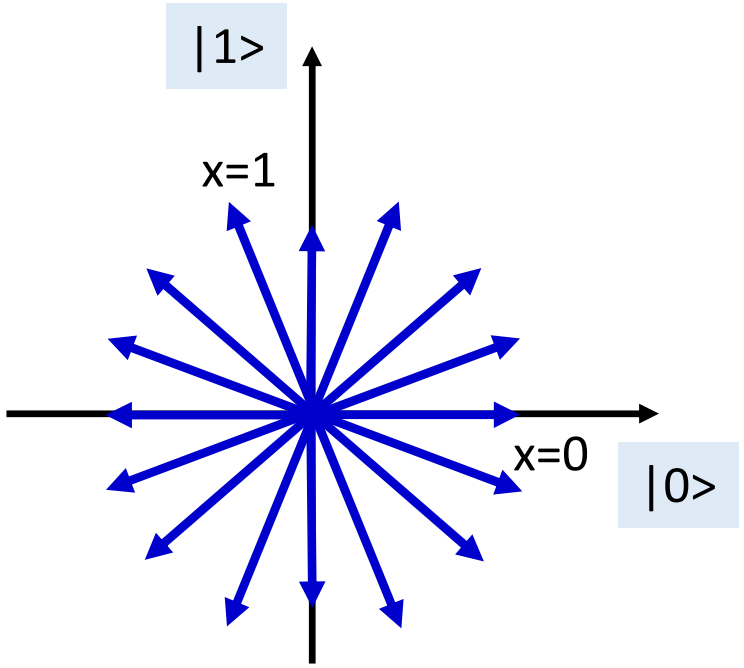


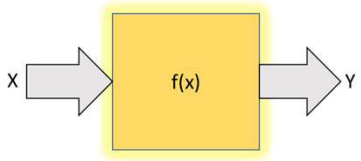
$$Fa = b$$



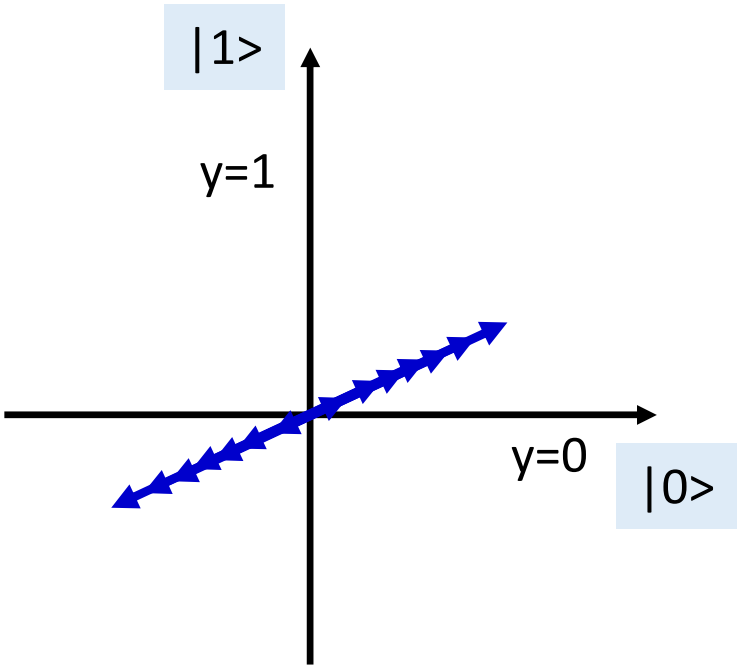
- The function is a linear transform that transforms an input vector to an output vector
  - In the process, it determines the output for *every input bit pattern* in *one step!!*
  - A single computation uncovers the entire function!
  - But there is one more requirement… (what)?

For f(x) to retain information



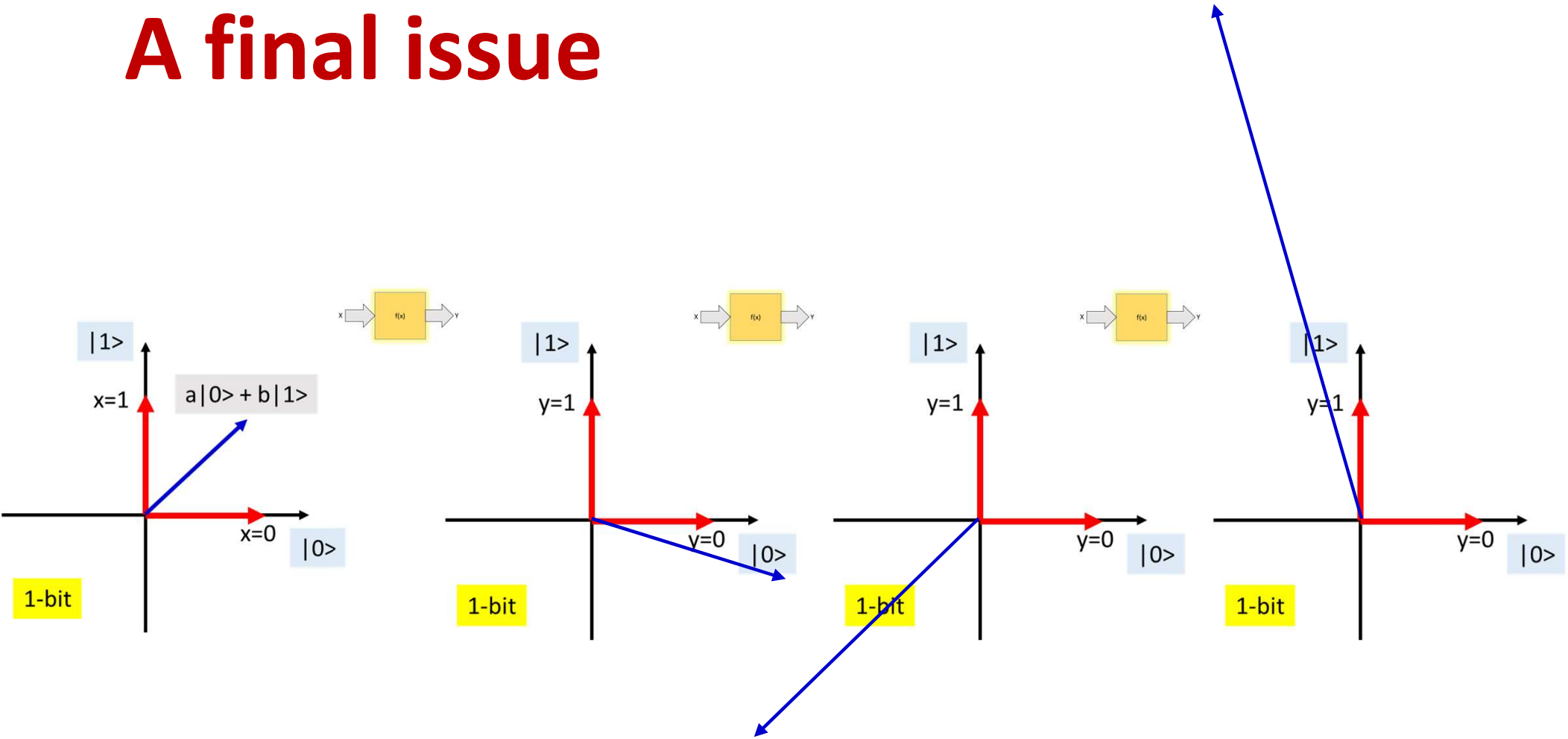


$$Fa = b$$



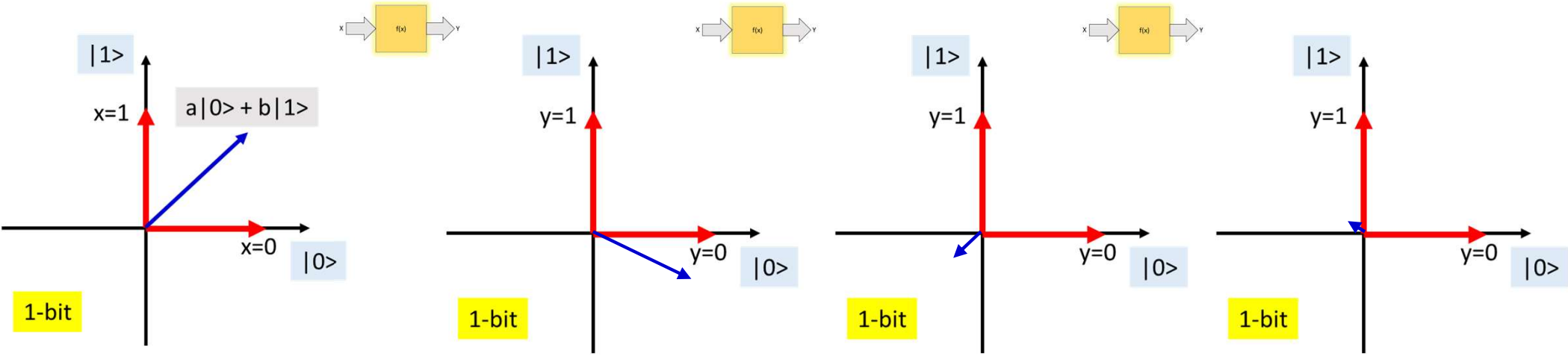
- F must be full rank
  - Otherwise, we cannot recover the contribution of all components of the input vector
    - I.e. we cannot resolve the contributions of all bit patterns to the result
- In other words, F must be *invertible!!*

A final issue



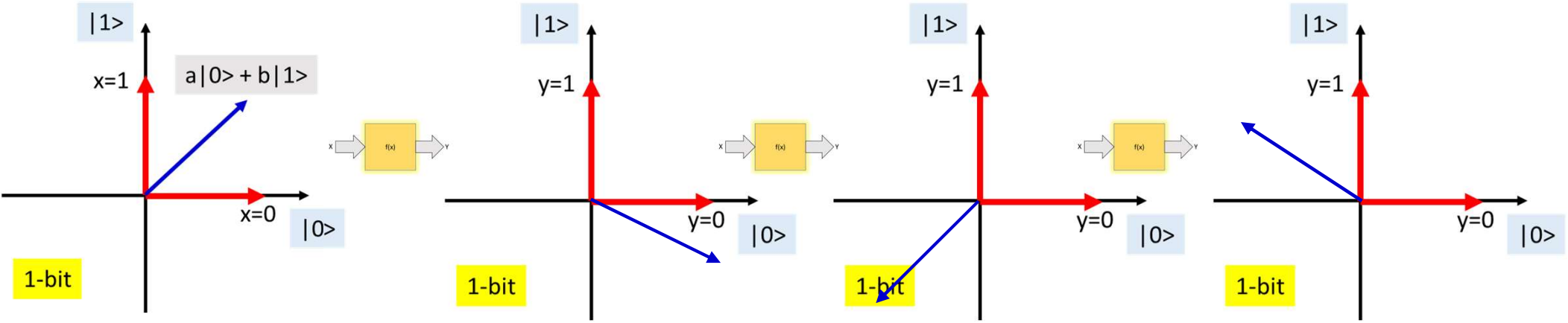
- Repeated applications of linear transforms can make a vector longer and longer and blow up…

A final issue



- Repeated applications of linear transforms can make a vector longer and longer and blow up…
- … or shrink and vanish

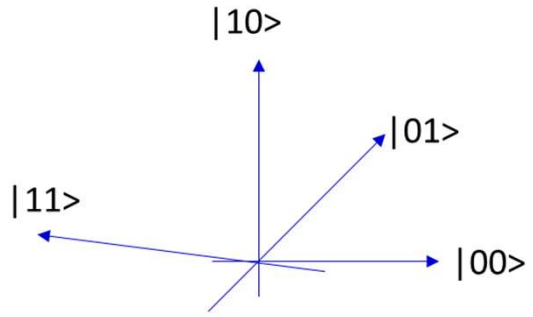
The solution



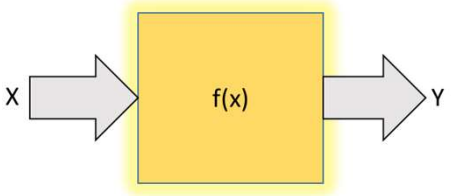
- F must not change the length of the vector
- i.e. it must be a rotation transform!

The new paradigm

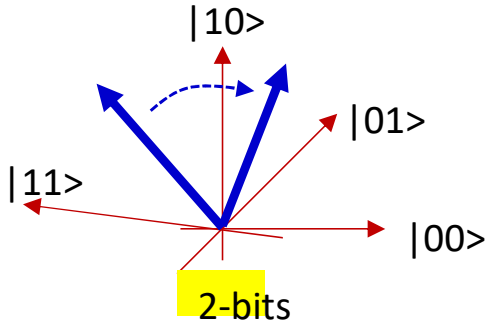
- Modified representation: Every bit *pattern* is an orthogonal direction of the input space
  - A vector in this space is a linear combination of all bit patterns



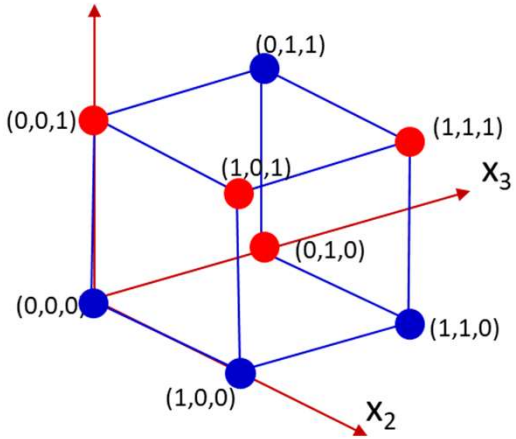
- The function being computed is representable as an invertible linear transform
  - More specifically a rotation

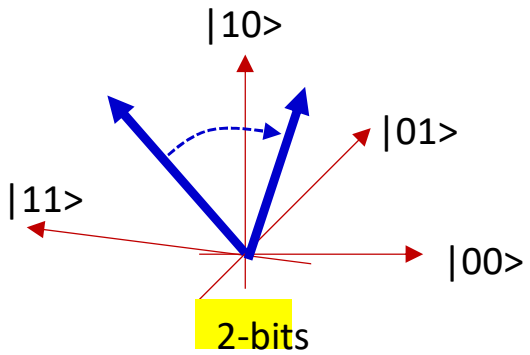


- Evaluation of the function on a single vector can compute the output for every possible bit pattern, in an identifiable way



Old vs. the new paradigm

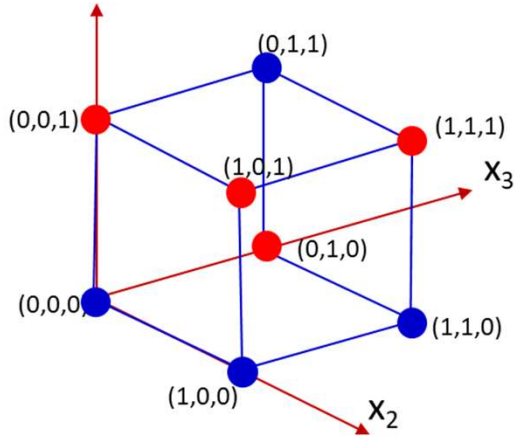


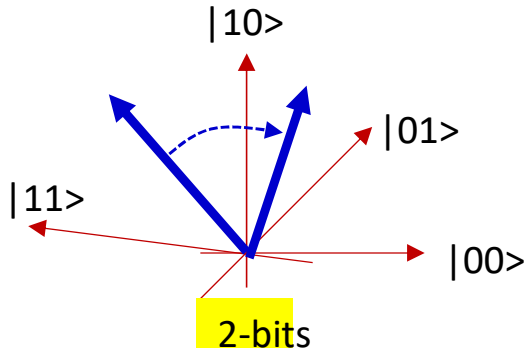


- Each bit is a coordinate dimension
- Must explicitly evaluate function for every input to fully determine it
- In reverse: Given only the output, must evaluate every input to determine which one generated it
  - 2^Ncomputations

- Each bit pattern is a coordinate dimension
- A single evaluation fully determines the function
- In reverse: Given an output, determining which input produced it is a single-step computation
  - Because computation is reversible

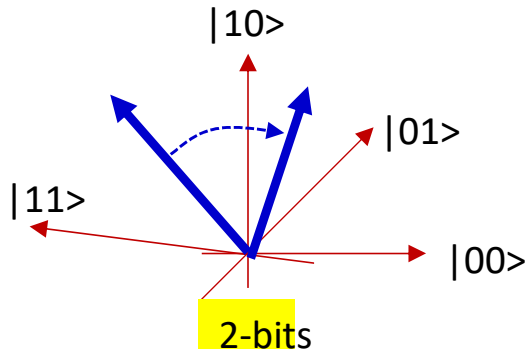
Where is this useful





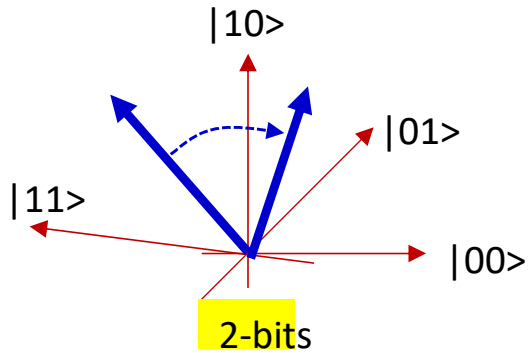
- Satisfiability problems
  - Does any bit pattern produce the output 1
- Search problems
  - Is any bit pattern in my library exactly equal to 11001010
    - Equivalent to SAT problems
- Combinatorial optimization problems
- ….
- Any problem that can be set as a SAT problem

What are the practical issues?



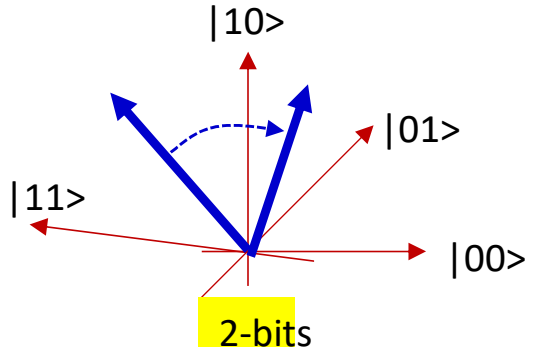
- Is this practically realizable?

What are the practical issues?



- Is this practically realizable?
- A conventional (classical) computer requires a *single* N-bit register to represent a value
  - A function takes in a single N-bit value and produces a single bit
- The new representation requires 2^N numbers to represent a single value!
  - For even N=100 bits, this requires ~1000000000000000000000000000000 numbers
  - A function take in 2^{100} values and produces 2^{100} values
    - I.e it's a 2^{200}-valued transform!
  - Which is why it's not a very *useful* way of thinking about things
- Is there a *parsimonious* way of representing a 2^{100} component vector without taking up the entire universe?

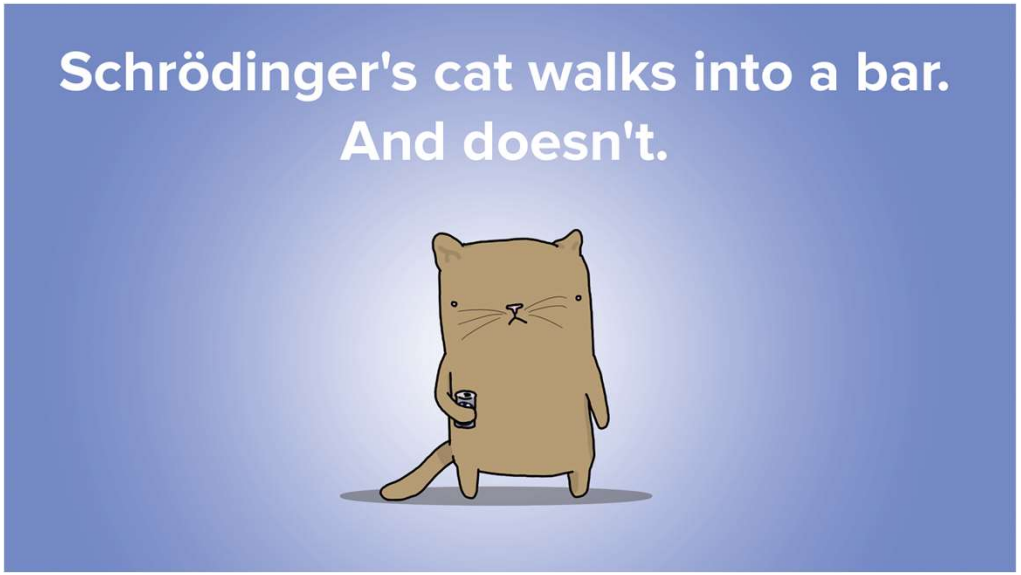
What are the practical issues?



Fact that may only interest me: "Graham's number" is a number that's so large there isn't enough space in the universe to write it..

- Is this practically realizable?
- A conventional (classical) computer requires a *single* N-bit register to represent a value
  - A function takes in a single N-bit value and produces a single bit
- The new representation requires 2^N numbers to represent a single value!
  - For even N=100 bits, this requires ~1000000000000000000000000000000 numbers
  - A function take in 2^{100} values and produces 2^{100} values
    - I.e it's a 2^{200}-valued transform!
  - Which is why its not a very *useful* way of thinking about things
- Is there a *parsimonious* way of representing a 2100 component vector without taking up the entire universe?

Enter.. The cat!!



- Introducing Quantum, the cat