The GMW Protocol

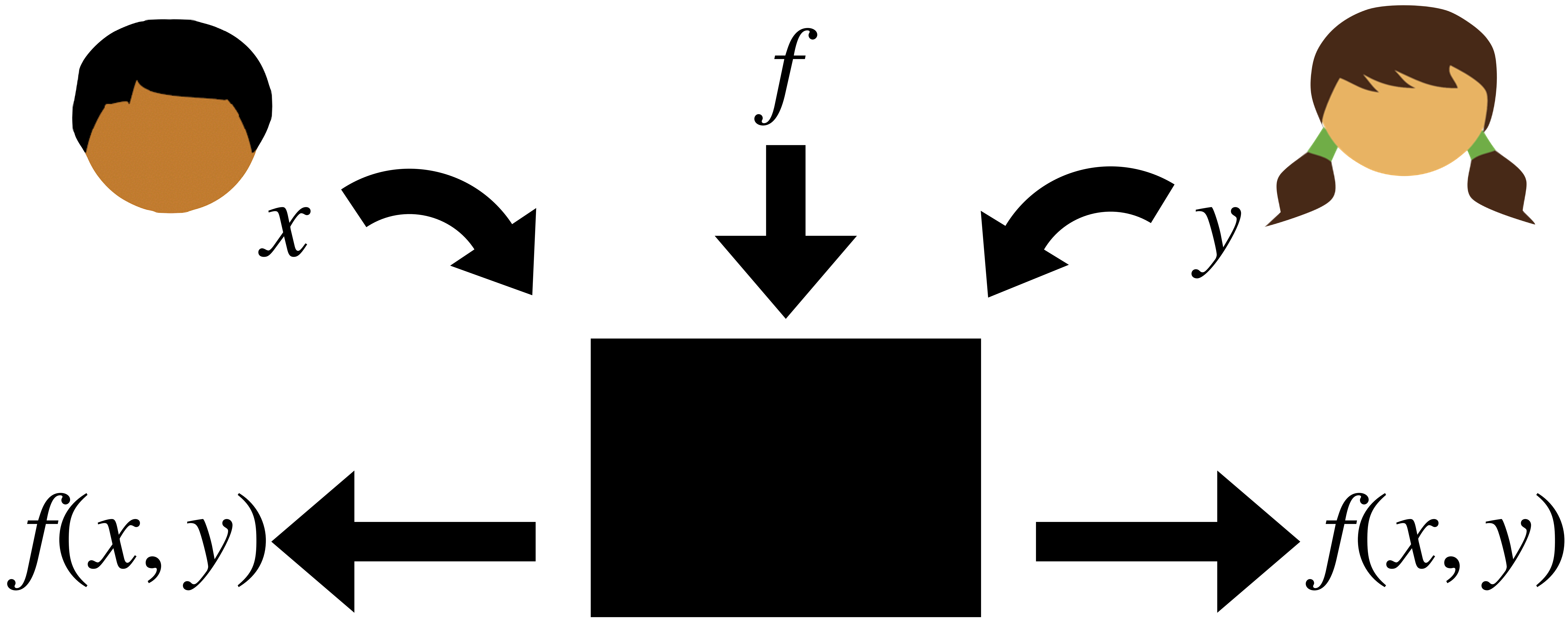
CS 598 DH

Today's objectives

Review oblivious transfer

Introduce XOR secret sharing

Build our first protocol for securely computing any program (with semi-honest security)


f
x
y
$f(x, y)$
$f(x, y)$

m_0, m_1
$b \in \{0,1\}$
1-out-of-2
Oblivious
Transfer
Sender
Receiver
$\perp$
m_b

HOW TO PLAY ANY MENTAL GAME

or

A Completeness Theorem for Protocols with Honest Majority

(Extended Abstract)

Oded Goldreich
Dept. of Computer Sc.
Technion
Haifa, Israel

Silvio Micali
Lab. for Computer Sc.
MIT
Cambridge, MA 02139

Avi Wigderson
Inst. of Math. and CS
Hebrew University
Jerusalem, Israel

Abstract

We present a polynomial-time algorithm that, given as a input the description of a game with incomplete information and any number of players, produces a protocol for playing the game that leaks no partial information, provided the majority of the players is honest.

Our algorithm automatically solves all the multi-party protocol problems addressed in complexity-based cryptography during the last 10 years. It actually is *a completeness theorem* for the class of distributed protocols with honest majority. Such completeness theorem is optimal in the sense that, if the majority of the players is not honest, some protocol problems have no efficient solution[C].

1. Introduction

Before discussing how to "make playable" a general game with incomplete information (which we do in section 6) let us address the problem of making playable a special class of games, the *Turing machine games* (*Tm-games* for short).

Informally, n parties, respectively and individually owning secret inputs $x_1,...,x_n$, would like to *correctly* run a given Turing machine M on these x_i's while keeping the maximum possible *privacy* about them. That is, they want to compute $y=M(x_1,...,x_n)$ without revealing more about the x_i's than it is already contained in the value y itself. For instance, if M computes the sum of the x_i's, every single player should not be able to learn more than the sum of the inputs of the other parties. Here M may very well be a probabilistic Turing machine. In this case, all players want to agree on a single string y, selected with the right probability distribution, as M's output.

The correctness and privacy constraint of a Tm-game can be easily met with the help of an extra, trusted party P. Each player i simply gives his secret input x_i to P. P will privately run the prescribed Turing machine, M, on these inputs and publically announce M's output. Making a Tm-game playable essentially means that the correctness and privacy constraints can be satisfied by the n players themselves, without invoking any extra party. Proving that Tm-games are playable retains most of the flavor and difficulties of our general theorem.

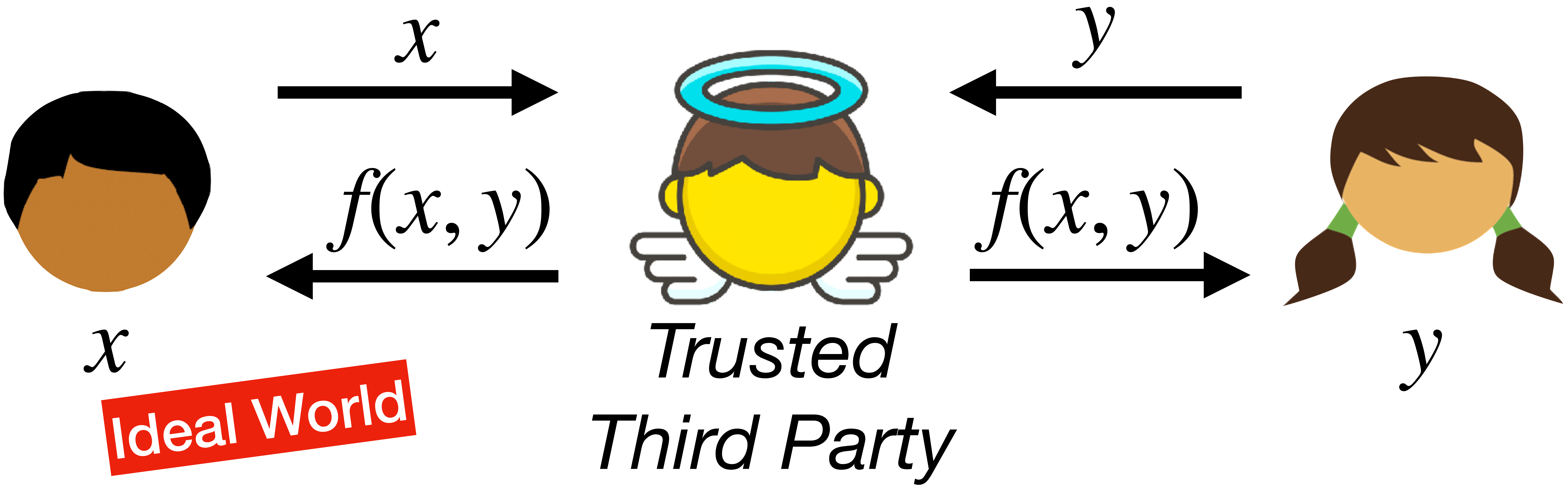
2. Preliminary Definitions

2.1 Notation and Conventions for Probabilistic Algorithms.

We emphasize the number of inputs received by an algorithm as follows. If algorithm A receives only one input we write "$A(\cdot)$", if it receives two inputs we write $A(\cdot,\cdot)$ and so on.

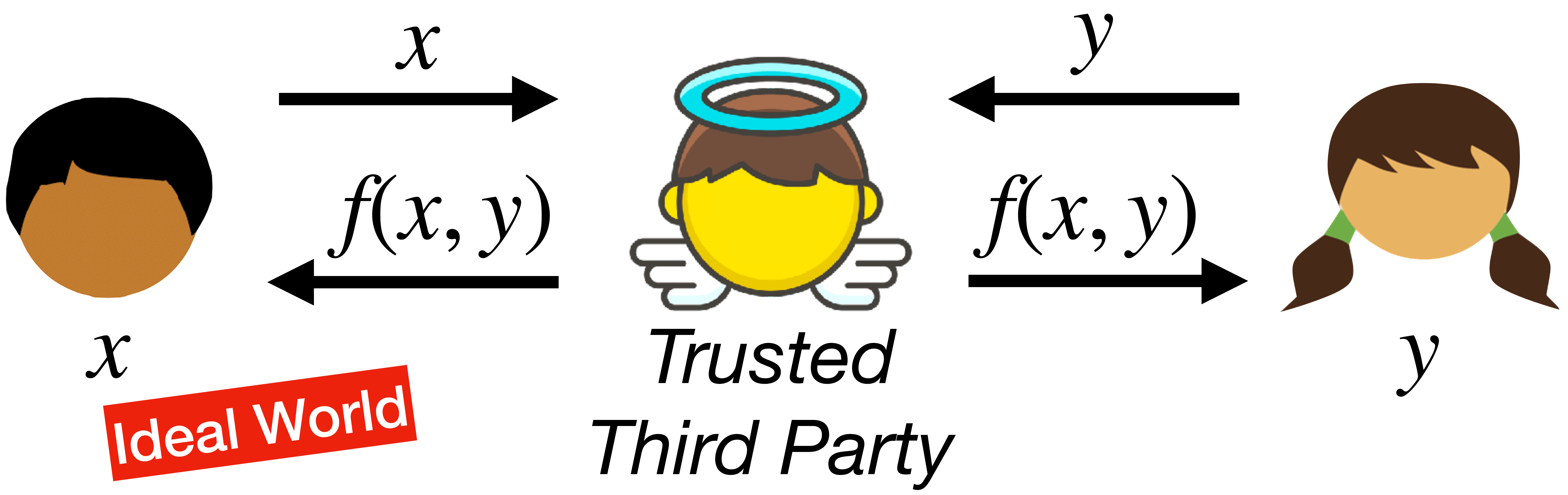
RV will stand for "random variable"; in this paper we only consider RVs that assume values in

Work partially supported by NSF grants DCR-8609905 and DCR-8413577, an IBM post-doctoral fellowship and an IBM faculty development award. The work was done when the first author was at the Laboratory for Computer Science at MIT; and the second author at the mathematical Sciences Research Institute at UC-Berkeley.



GMW Protocol

Real World



GMW Protocol

Hint: Lots of OT

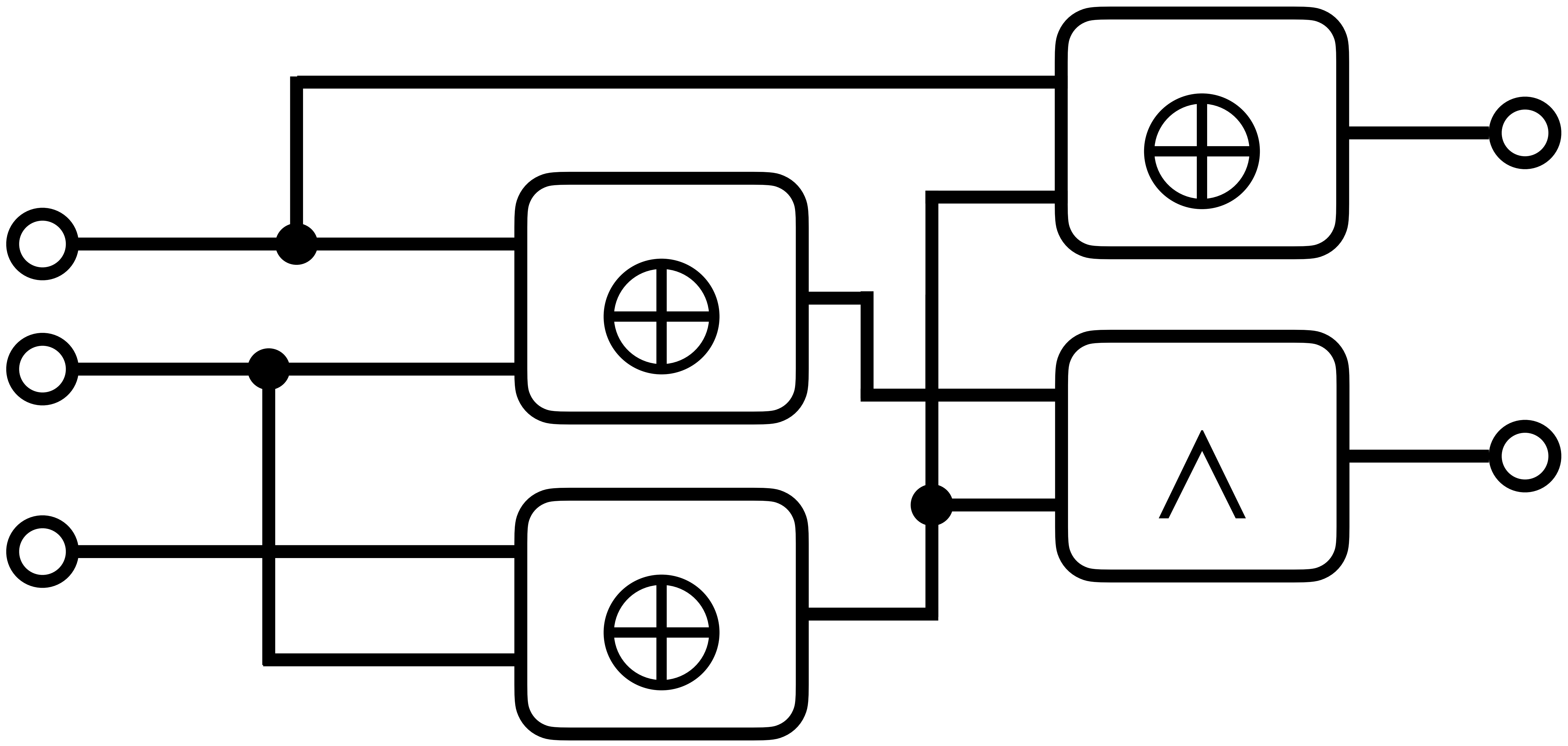
Real World

A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two.*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*

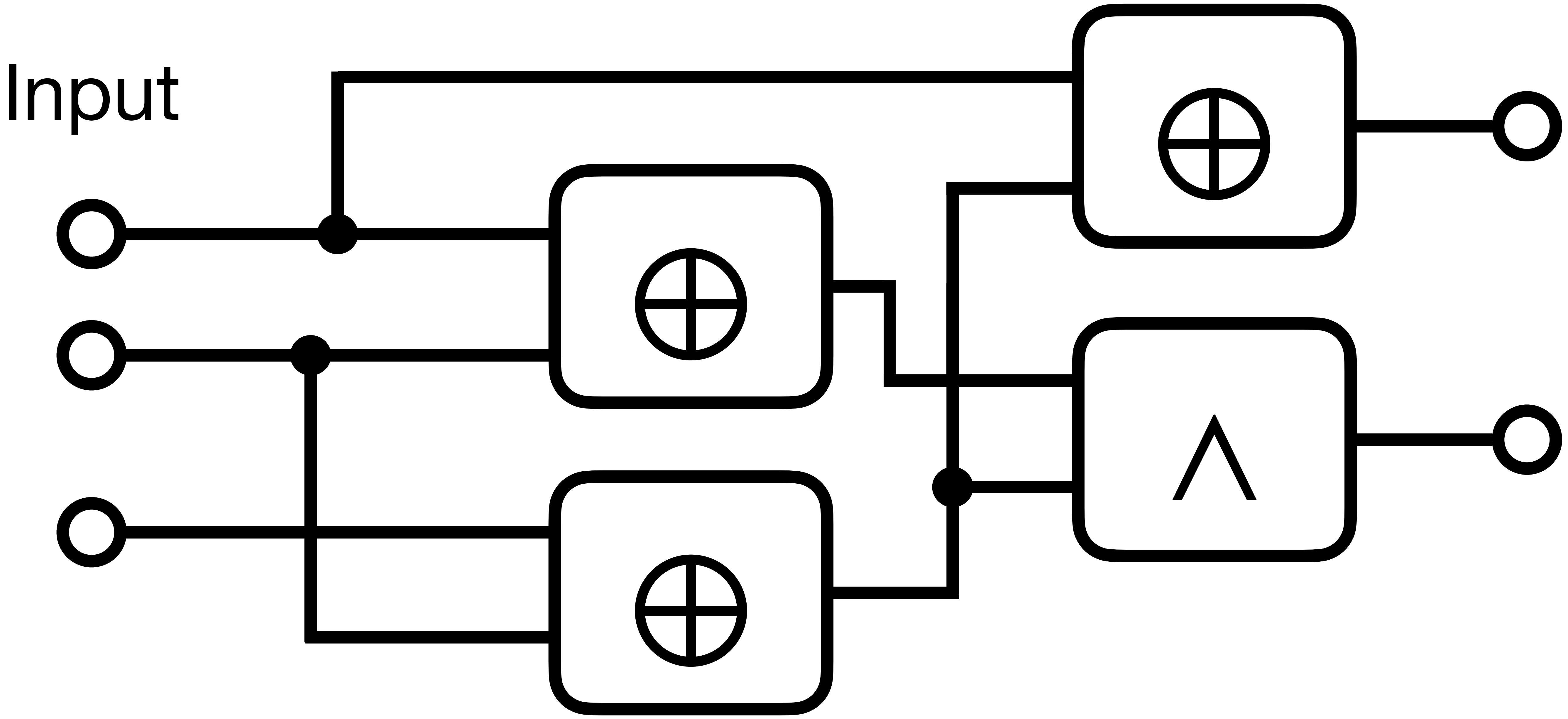
A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two (and unbounded fan-out).*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*



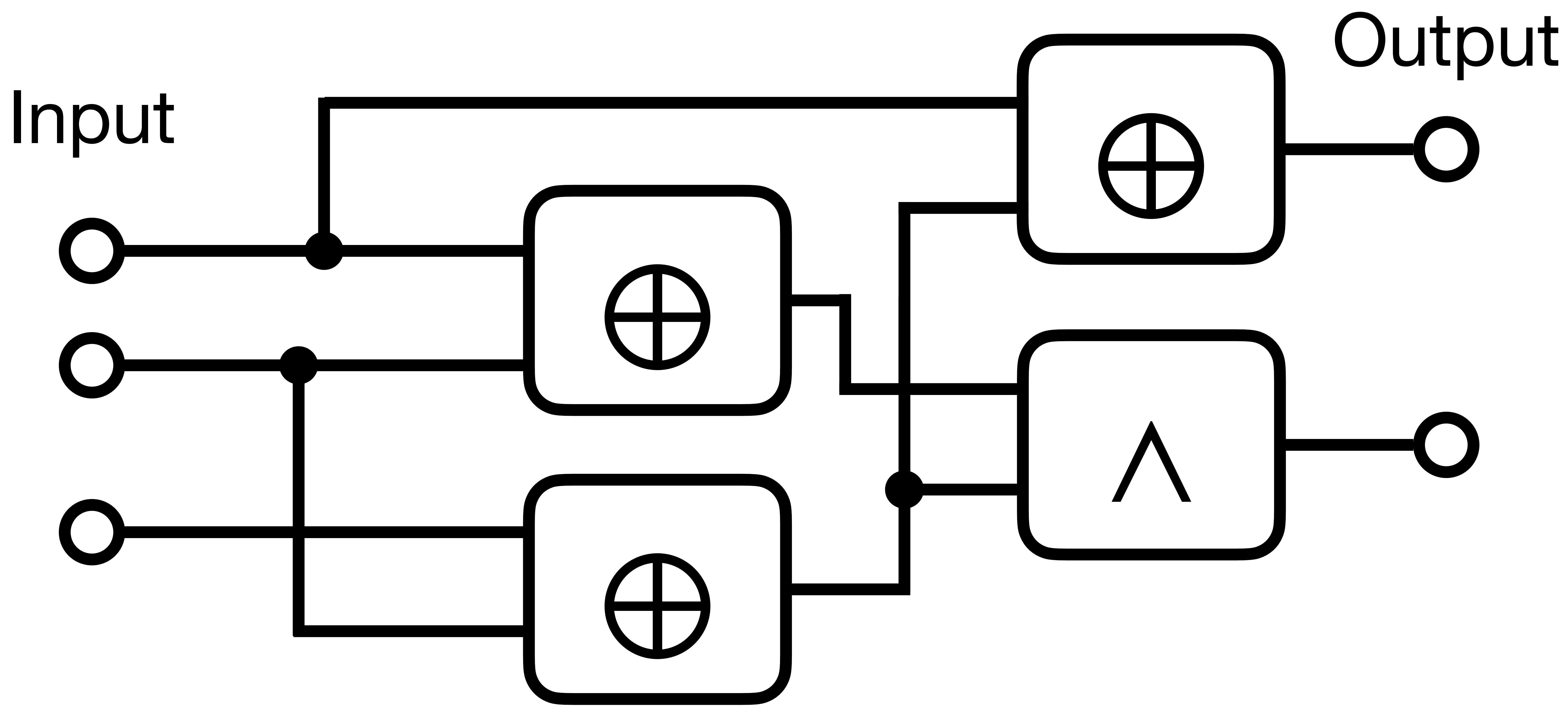
A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two (and unbounded fan-out).*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*



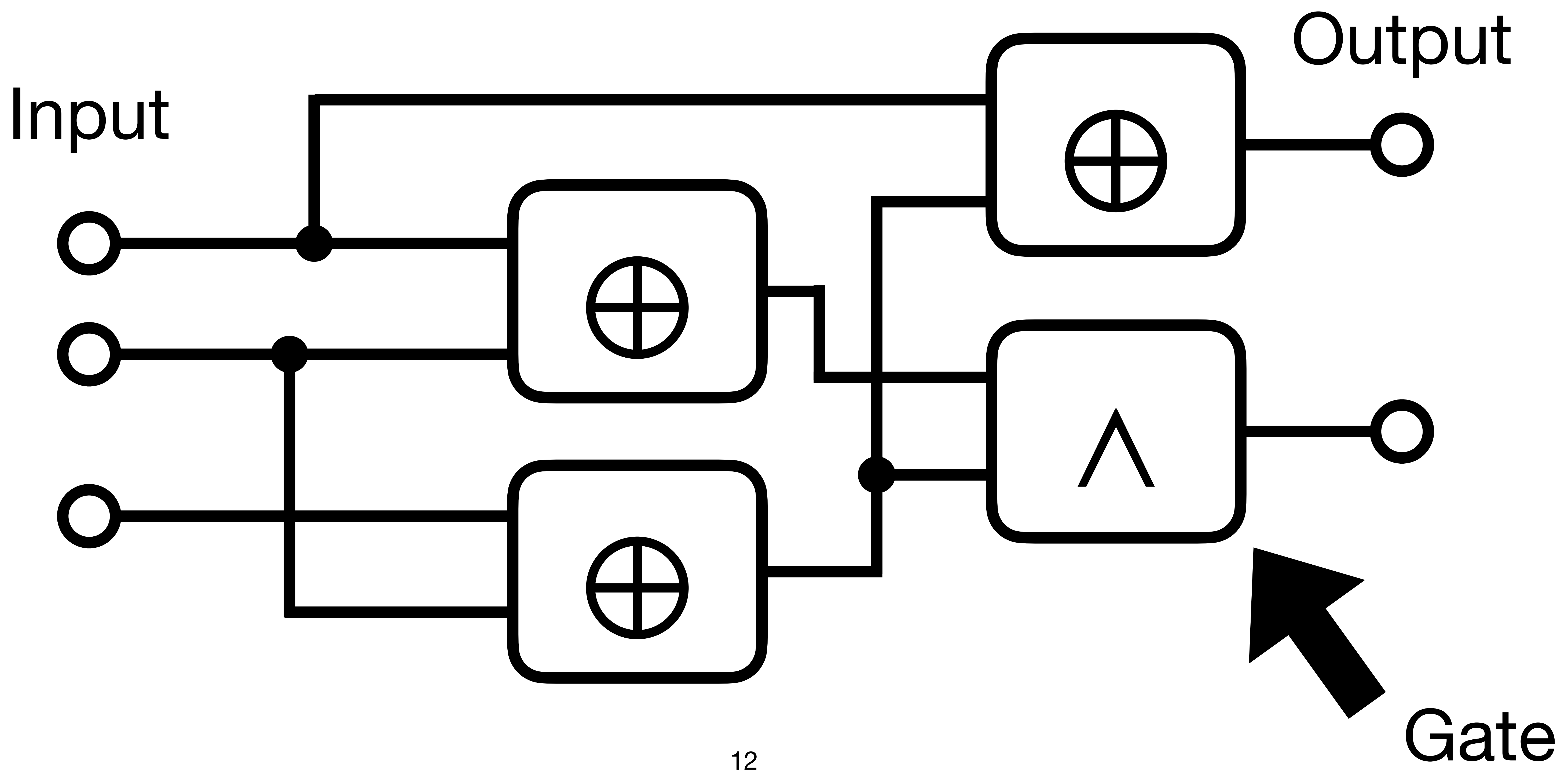
A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two (and unbounded fan-out).*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*



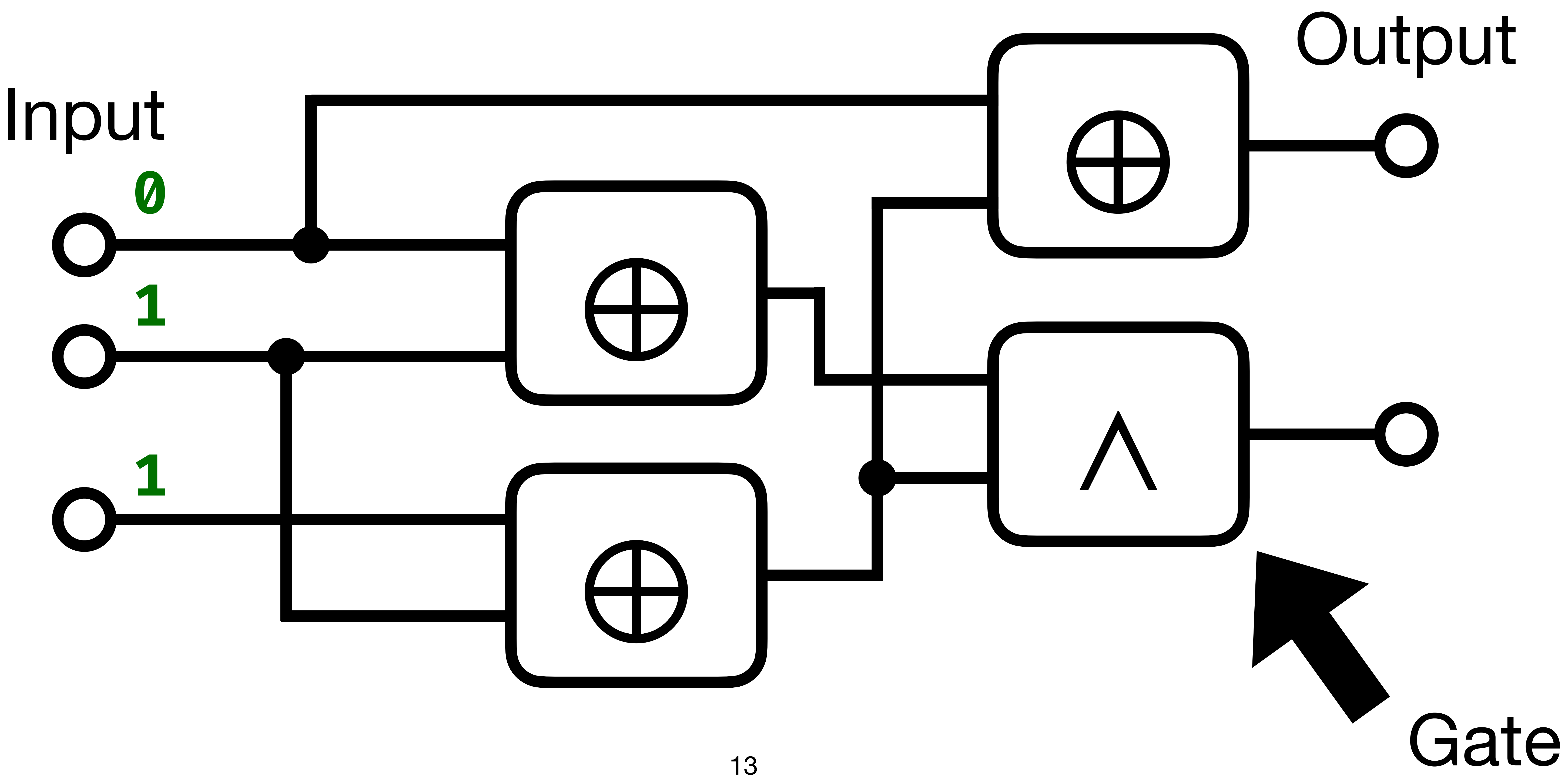
A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two (and unbounded fan-out).*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*



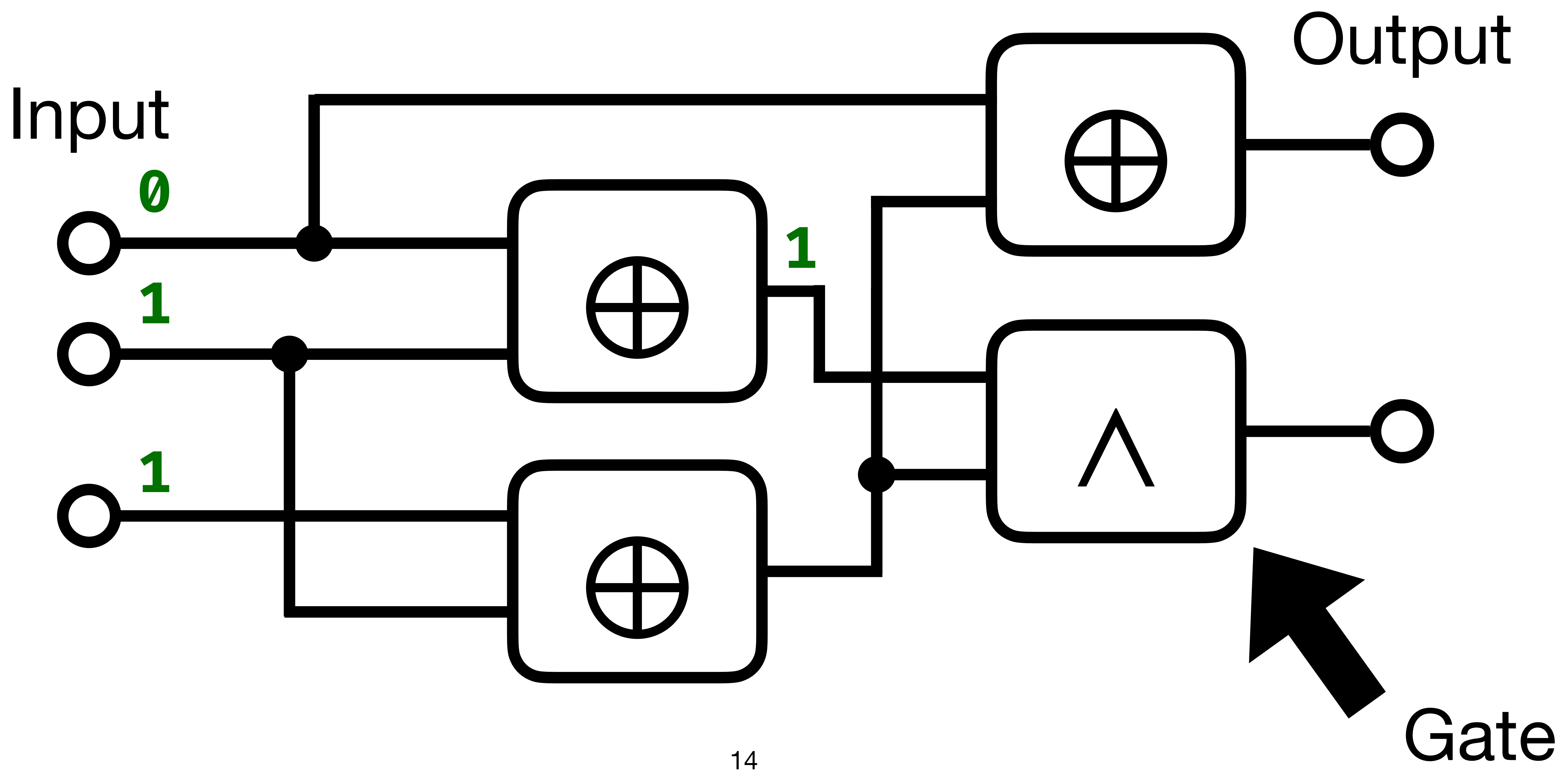
A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two (and unbounded fan-out).*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*



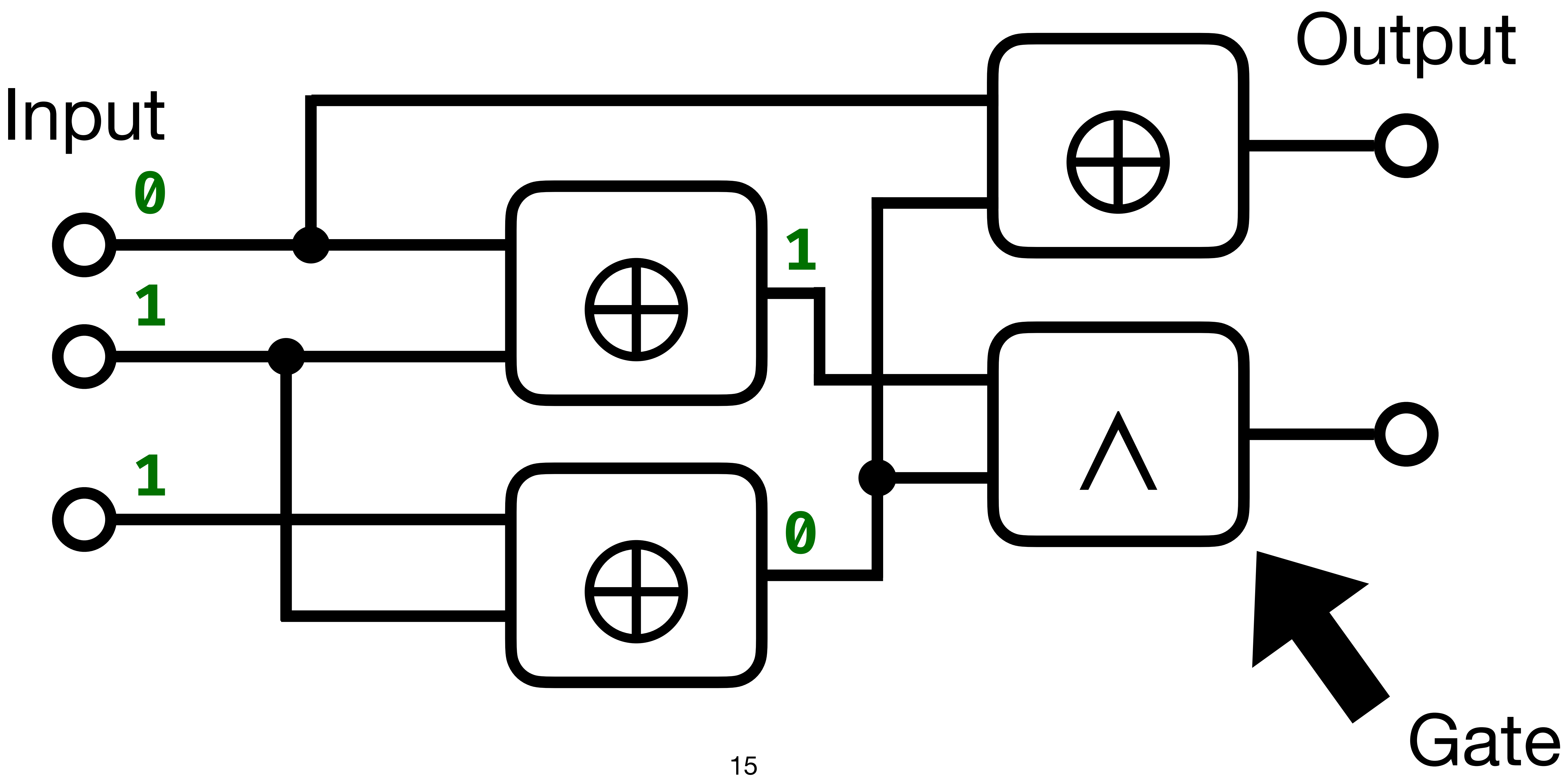
A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two (and unbounded fan-out).*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*



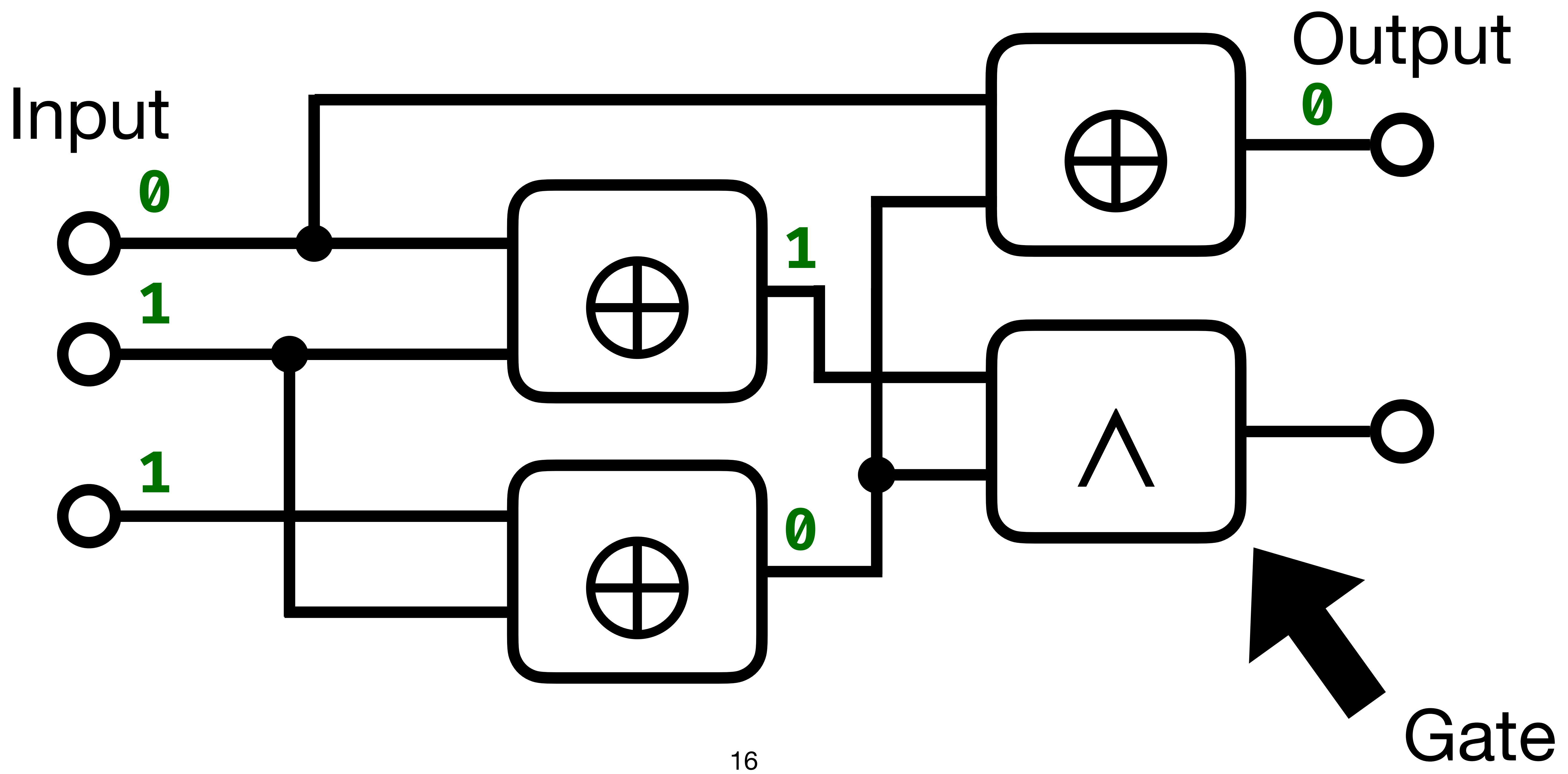
A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two (and unbounded fan-out).*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*



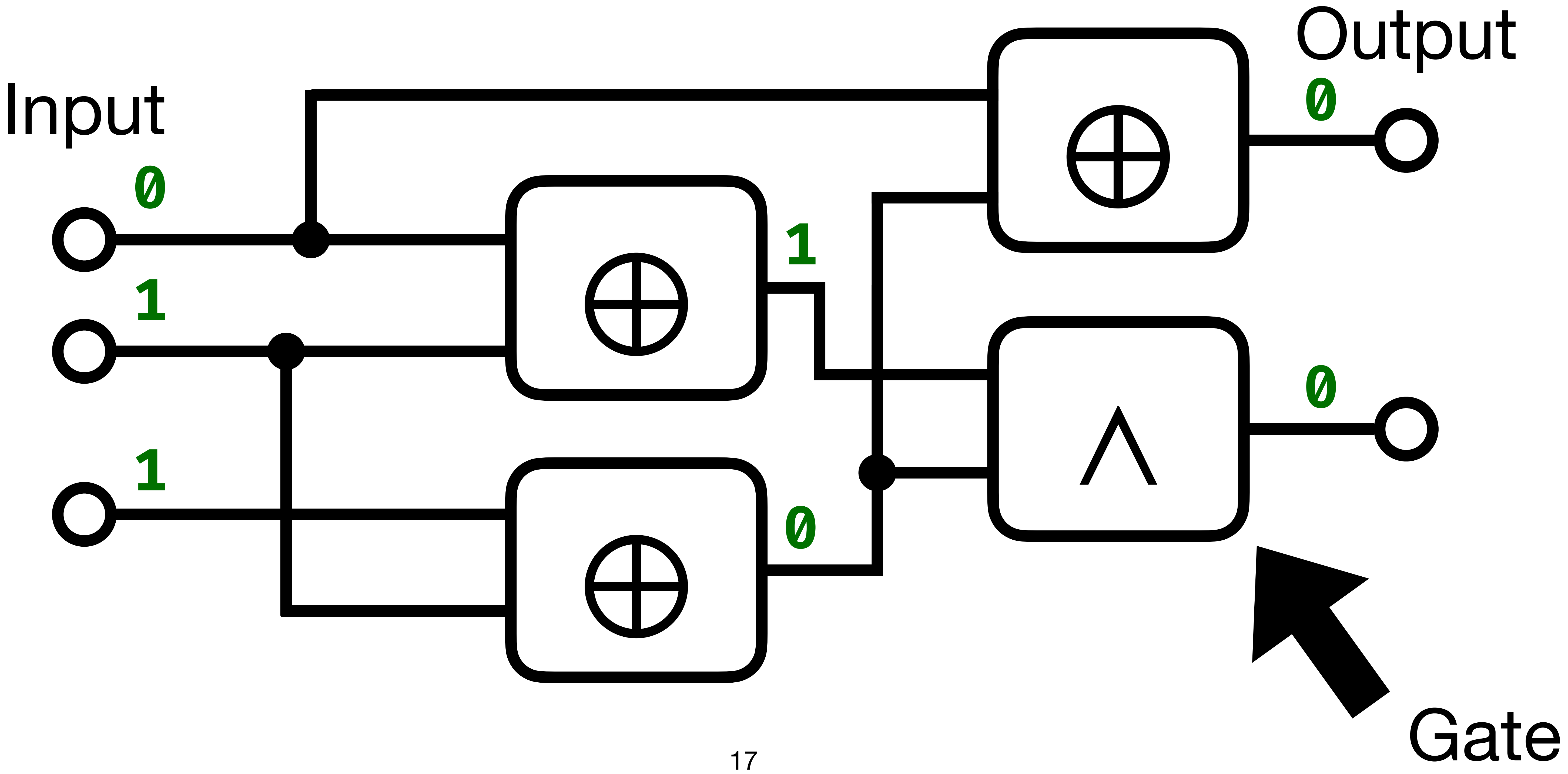
A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two (and unbounded fan-out).*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*

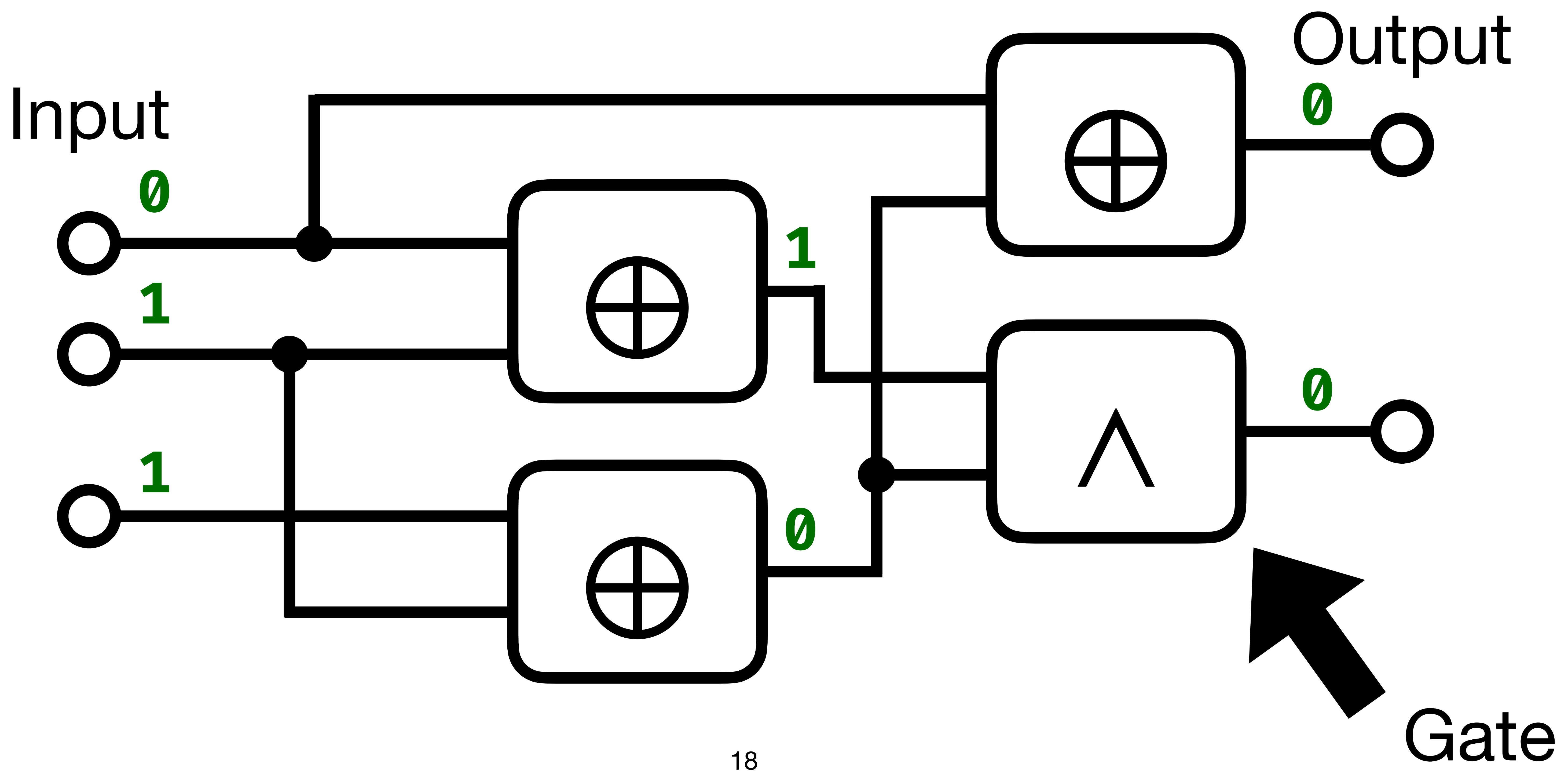


A Boolean Circuit *is a directed acyclic graph where*

- *Each node has* fan-in *two (and unbounded fan-out).*
- *Each node has a label* $\wedge$ *or* $\oplus$
- *There are two distinguished wires labelled 0 and 1*

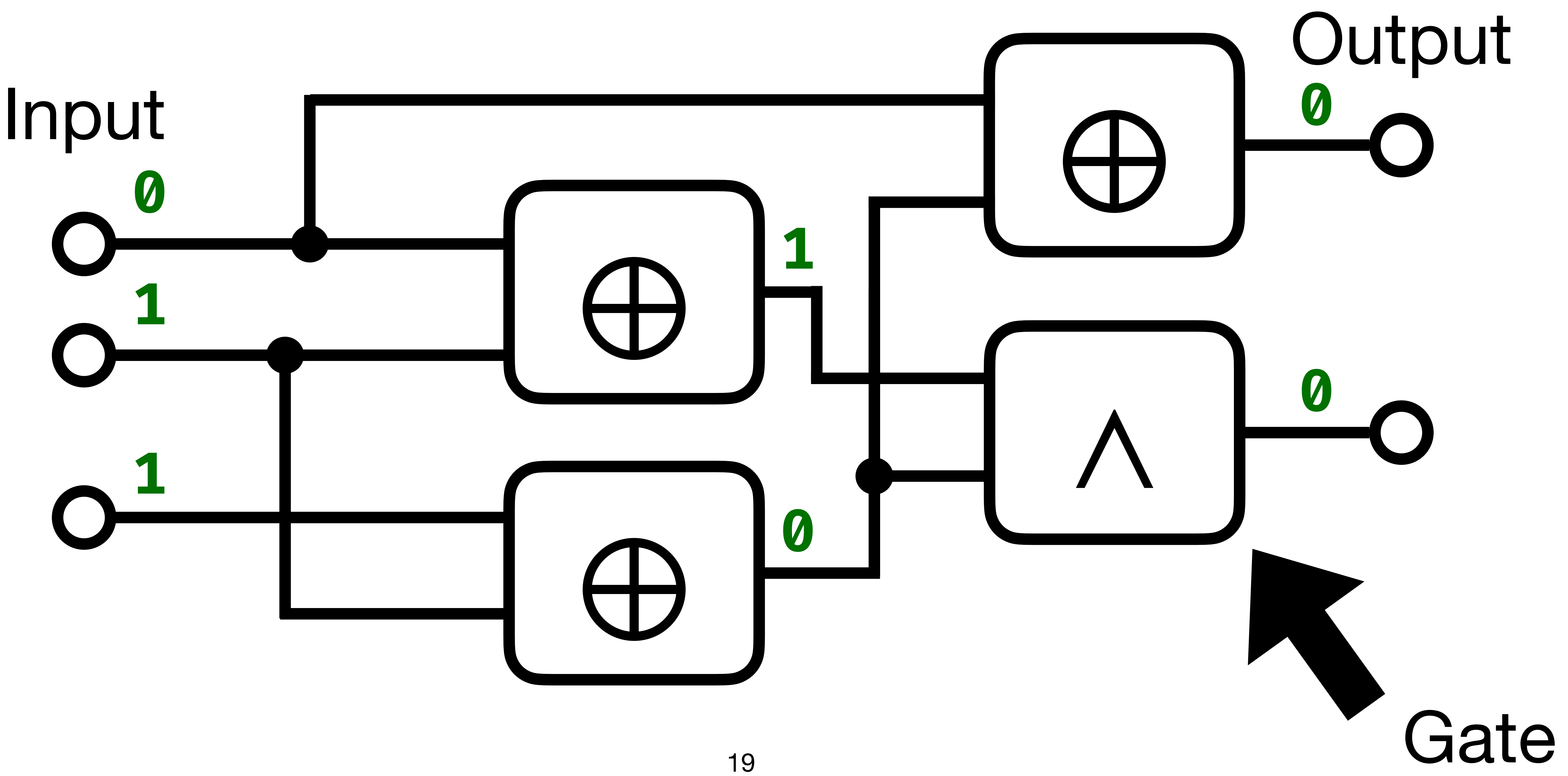


The size of C, written $|C|$, is the number of gates



The size of C, written $|C|$, is the number of gates

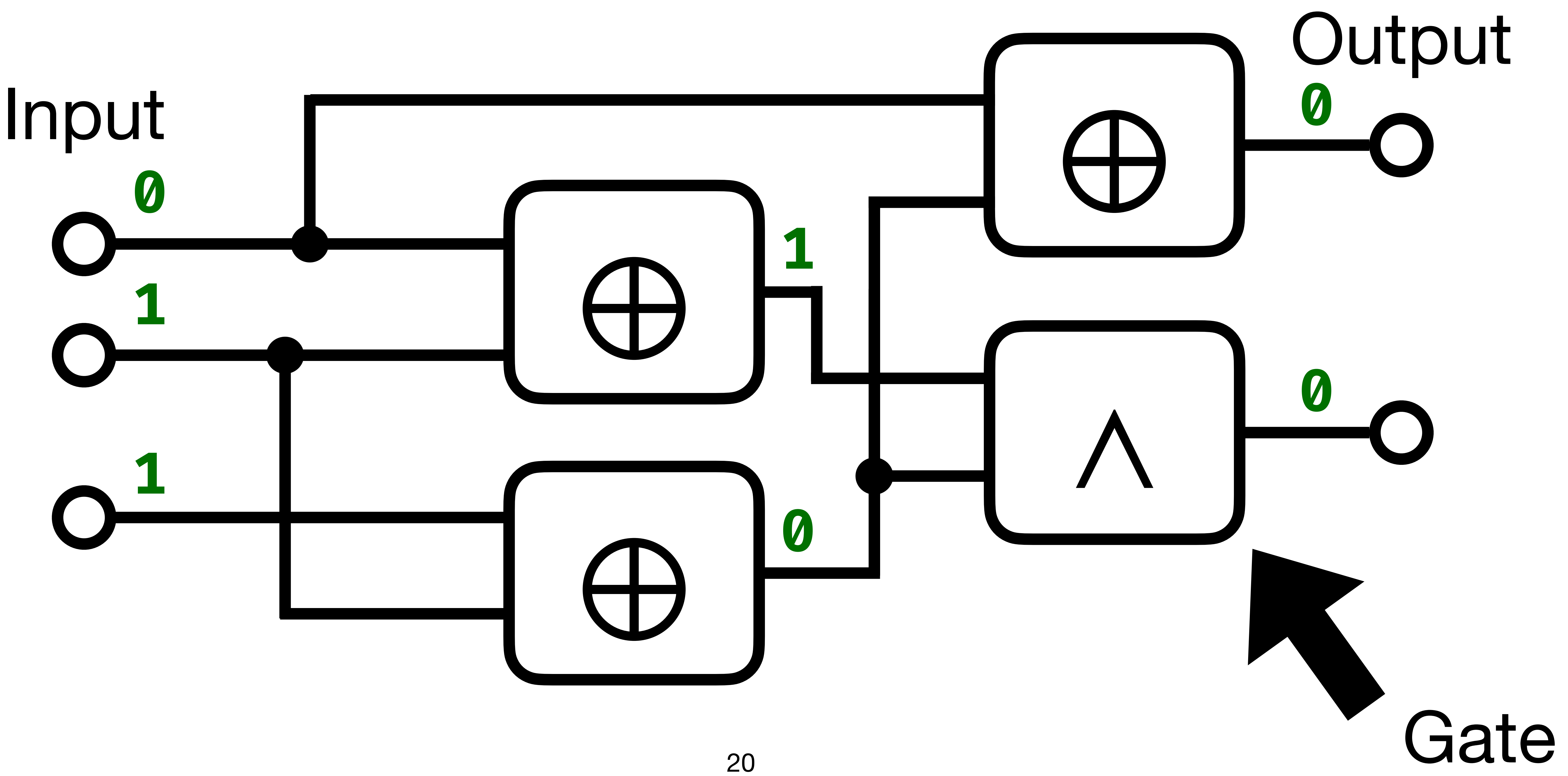
The depth of C is the length of the longest path from input to output



The size *of C, written $|C|$, is the number of gates*

The depth *of C is the length of the longest path from input to output*

The multiplicative depth *of C is the length of the longest path from input to output, counting only $\wedge$*



Fact: $\{\wedge, \oplus, 1\}$ *is a complete Boolean basis.*

Fact: $\{\wedge, \oplus, 1\}$ *is a complete Boolean basis.*

For any Boolean function $f : \{0,1\}^n \to \{0,1\}^m$*, there exists a Boolean circuit over* $\{\wedge, \oplus, 1\}$ *that computes* f*.*

I.e., Boolean circuits can compute any bounded function

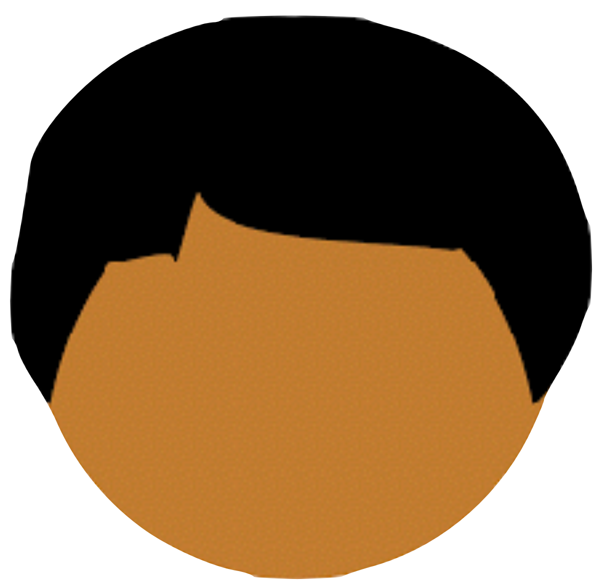
Step 1 of GMW:

Express function f as a Boolean circuit C

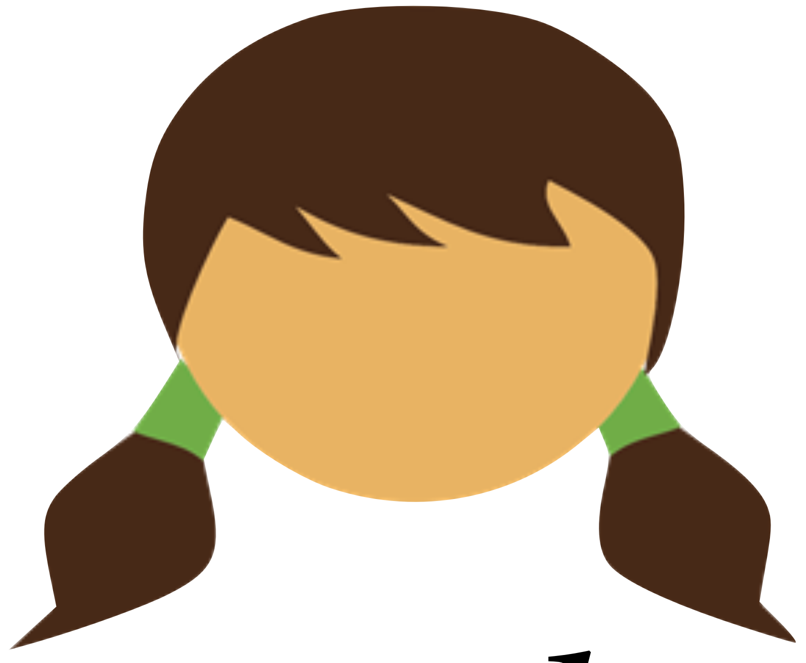
Step 1 of GMW:

Express function f as a Boolean circuit C

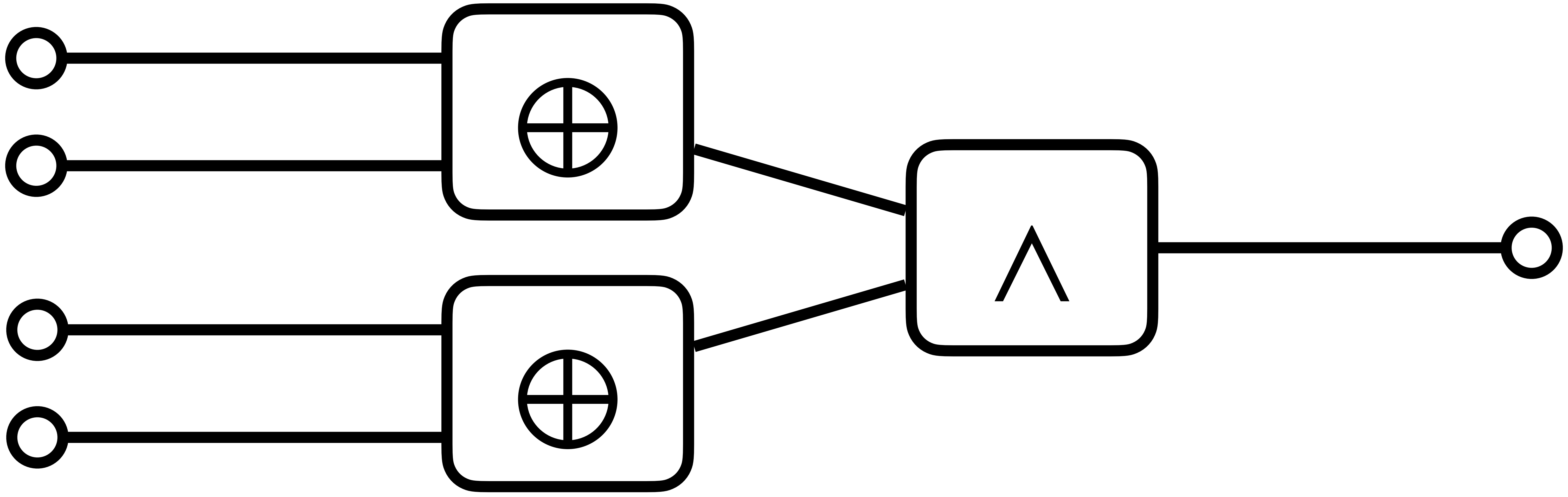
There is a lot more
to say about this!

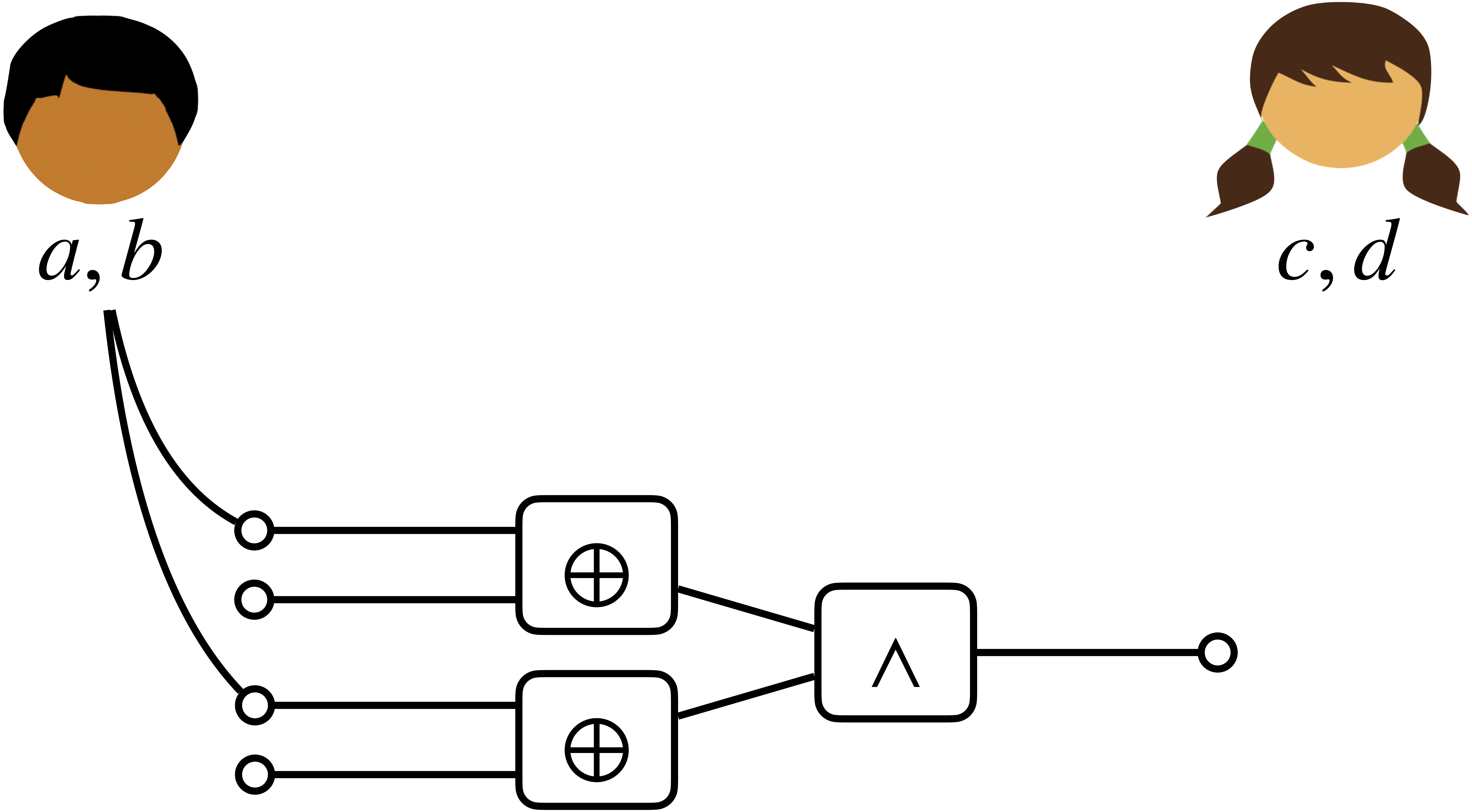


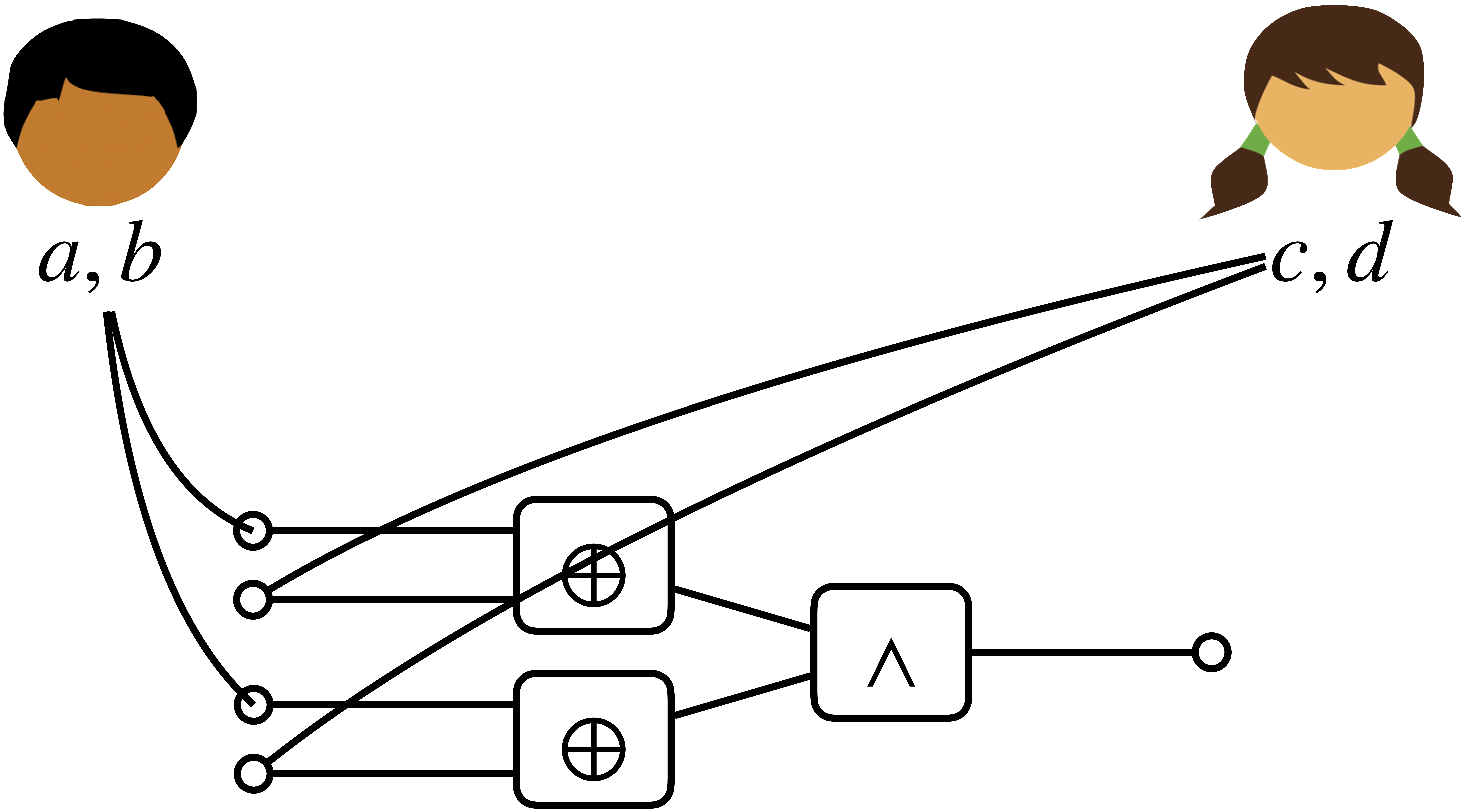
a, b

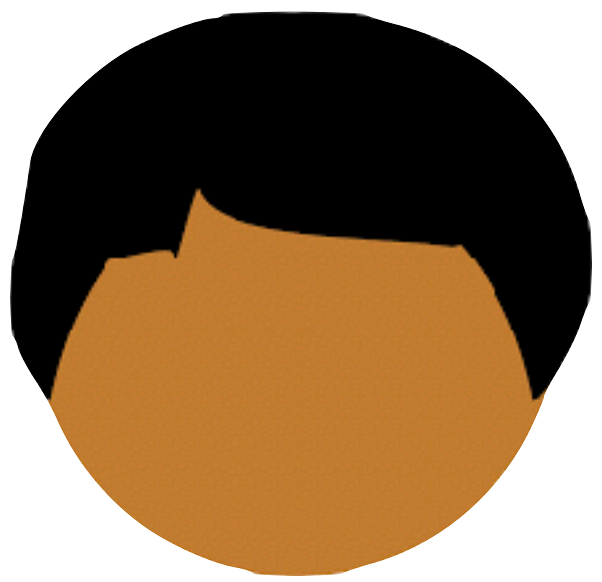


c, d

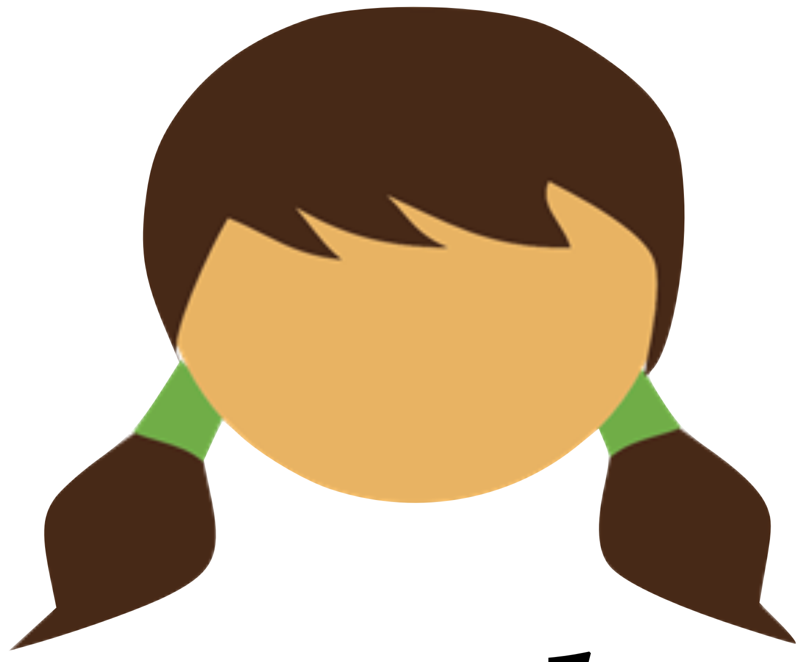



a, b
c, d
⊕
⊕
∧

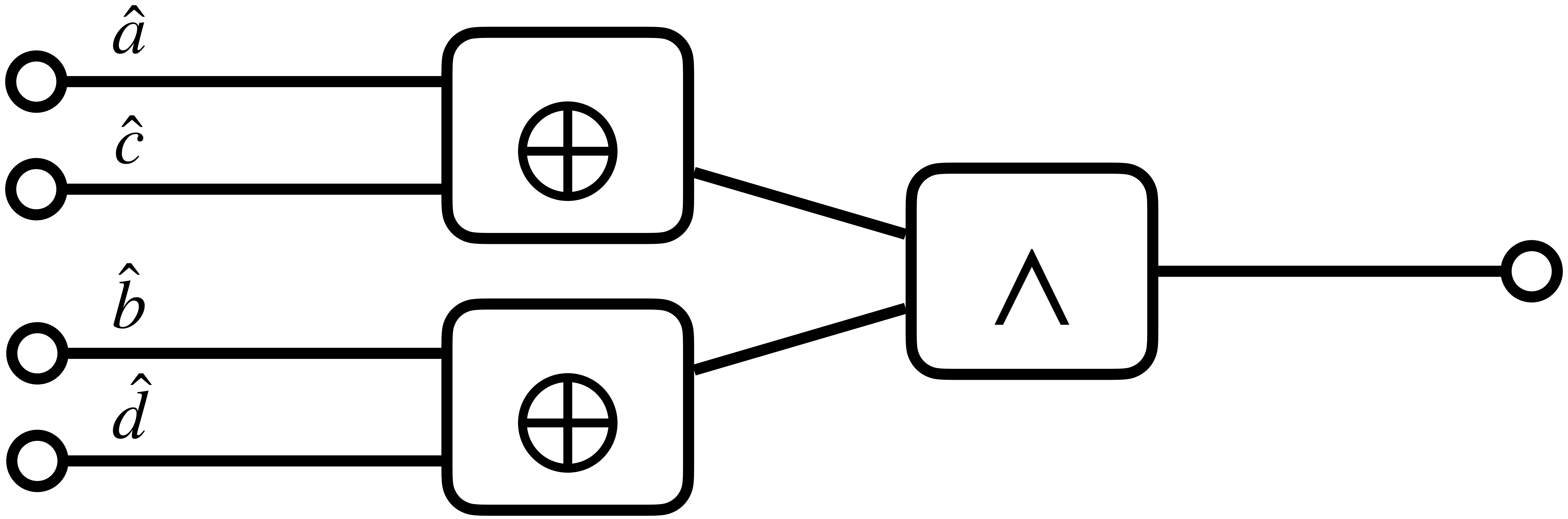

a, b
c, d
⊕
⊕
∧

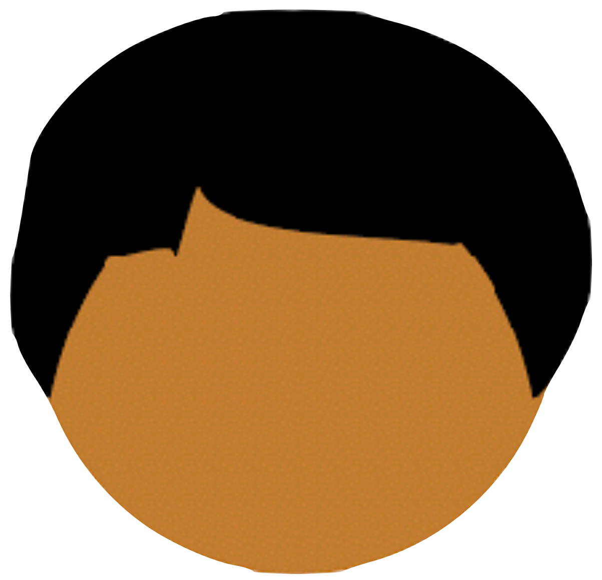


a, b



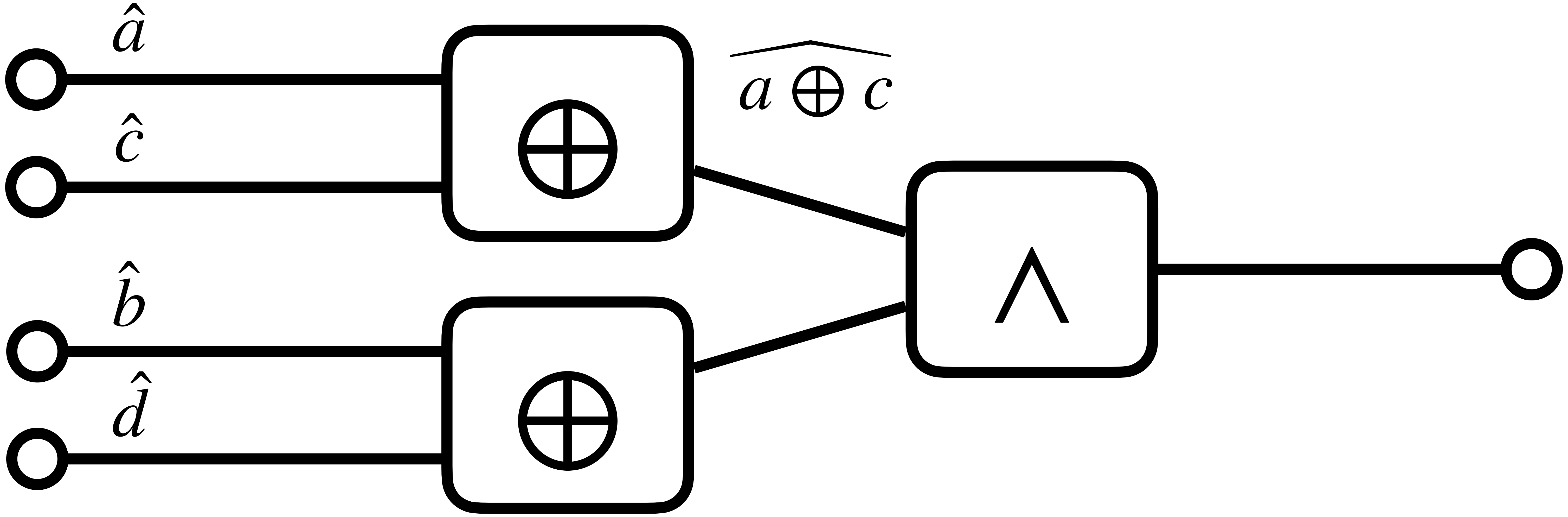
c, d

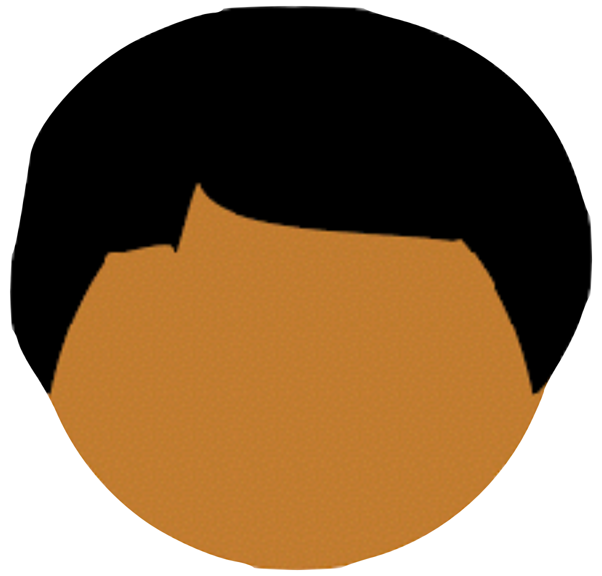




a, b

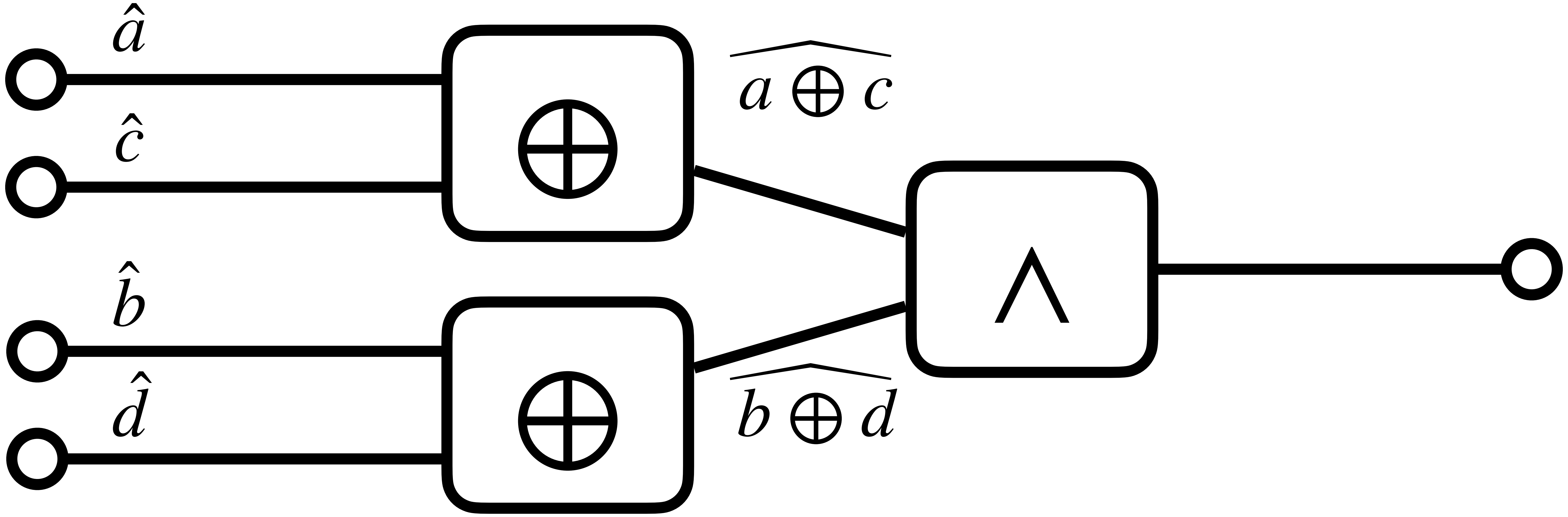
c, d

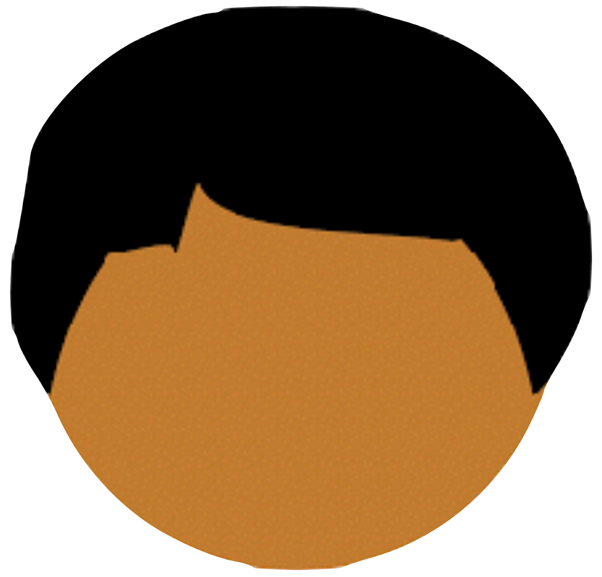




a, b

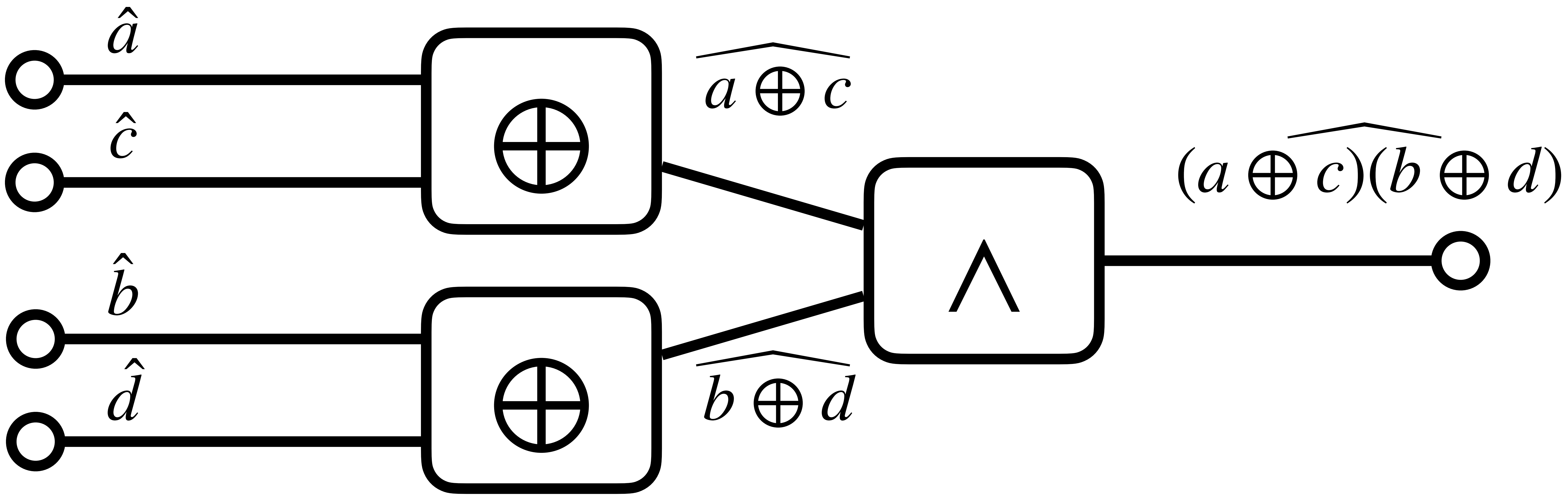
c, d


$\hat{a}$
$\hat{c}$
$\hat{b}$
$\hat{d}$
$\oplus$
$\oplus$
$\widehat{a \oplus c}$
$\widehat{b \oplus d}$
$\wedge$



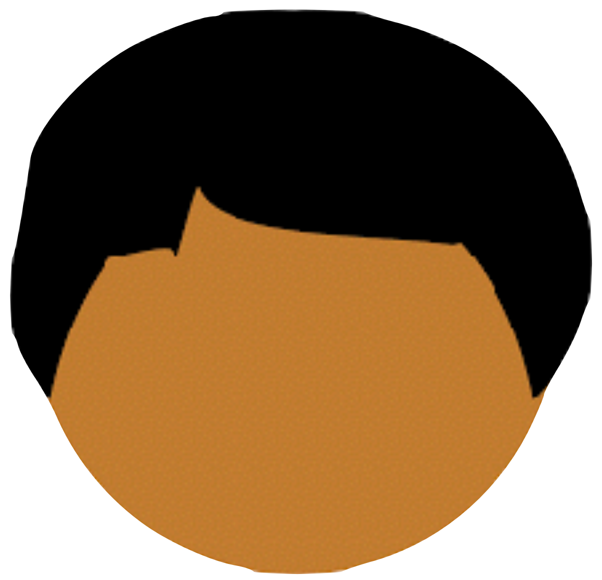
a, b

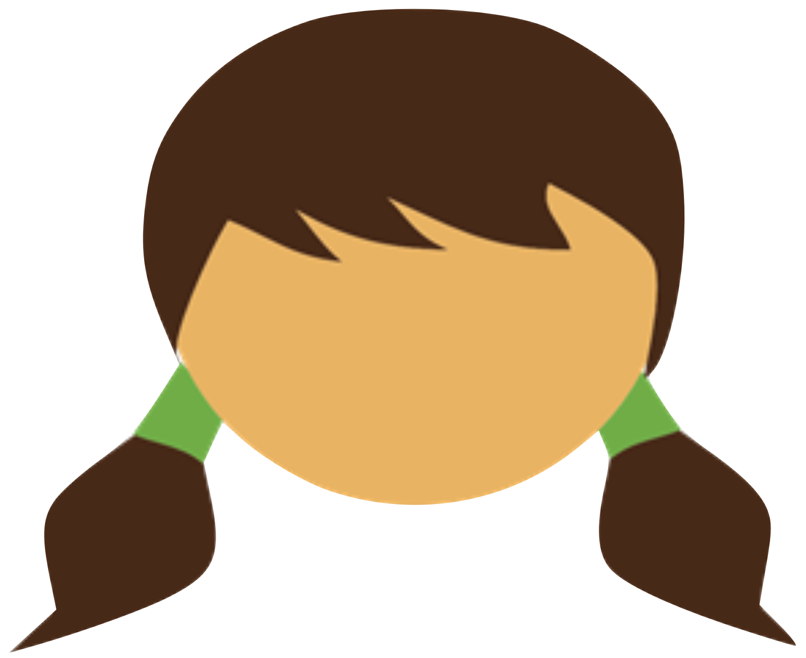
c, d



$\hat{a}$
$\hat{c}$
$\hat{b}$
$\hat{d}$
$\oplus$
$\oplus$
$\widehat{a \oplus c}$
$\widehat{b \oplus d}$
$\wedge$
$\widehat{(a \oplus c)(b \oplus d)}$

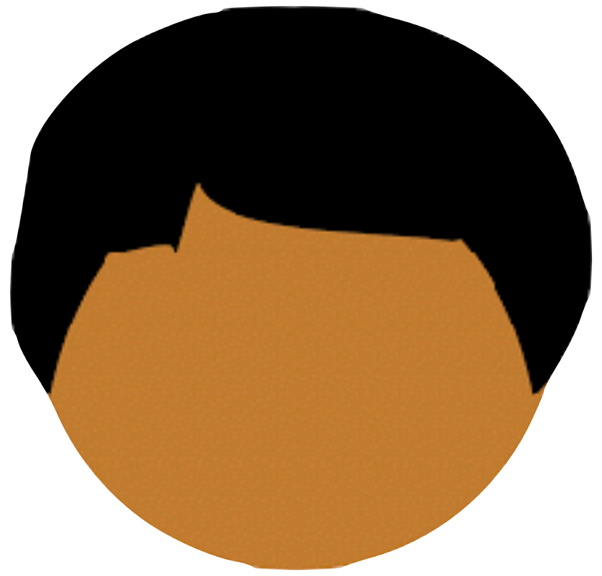
XOR Secret Shares

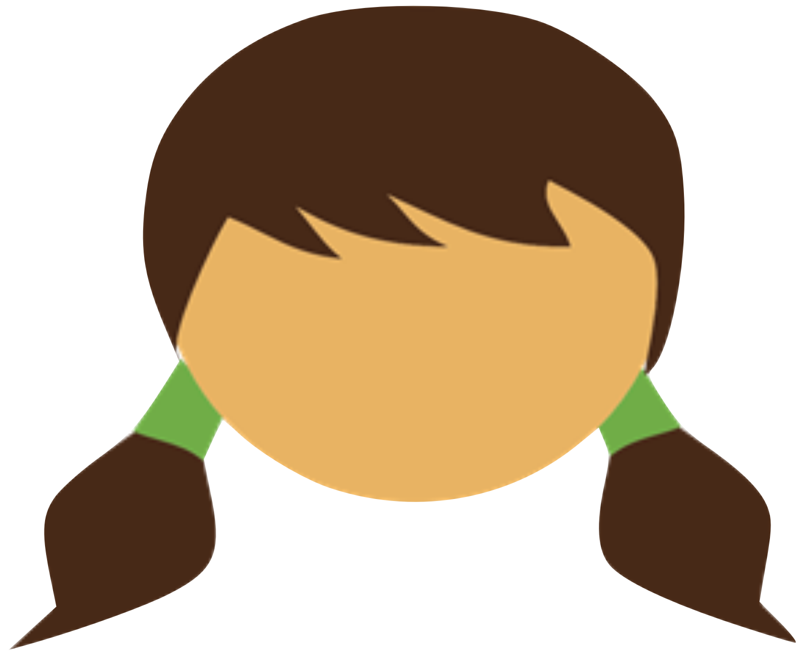




The XOR secret sharing *of a bit x is a pair of bits $\langle x_0, x_1 \rangle$ where P_0 holds x_0 and P_1 holds x_1, and where $x_0 \oplus x_1 = x$*

XOR Secret Shares

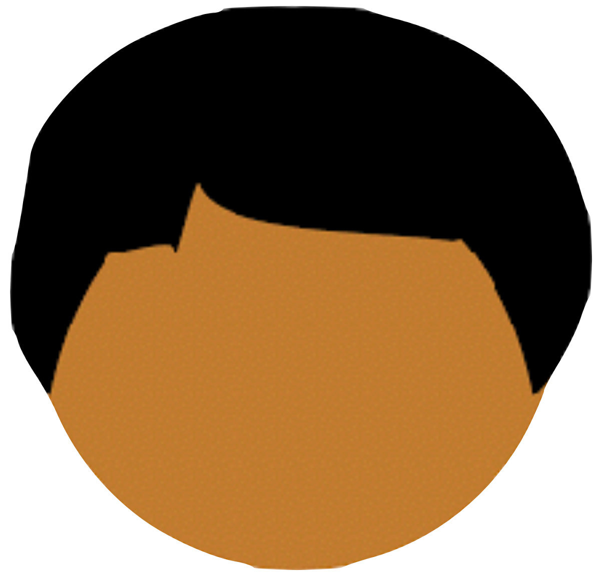


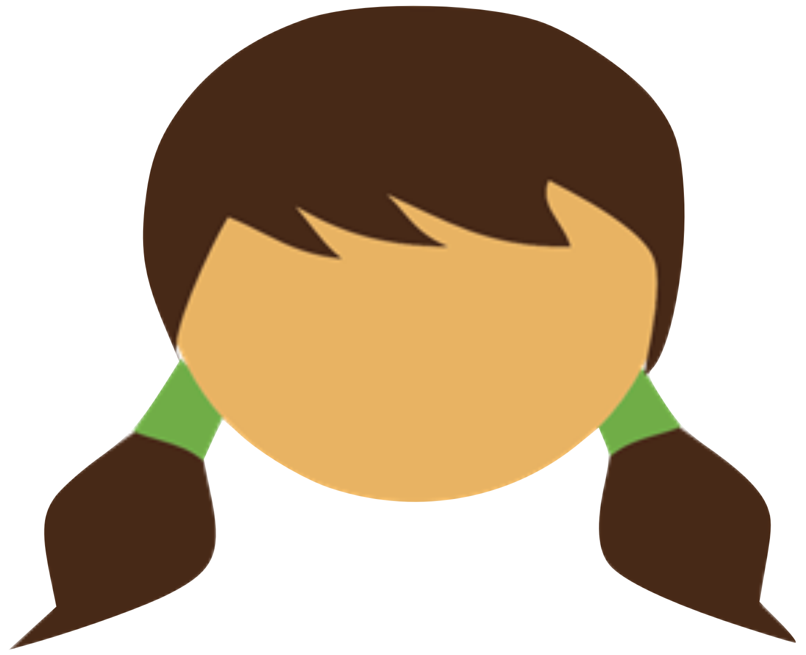


The XOR secret sharing *of a bit x is a pair of bits $\langle x_0, x_1 \rangle$ where P_0 holds x_0 and P_1 holds x_1, and where $x_0 \oplus x_1 = x$*

We sometimes denote such a pair by $[x]$

XOR Secret Shares

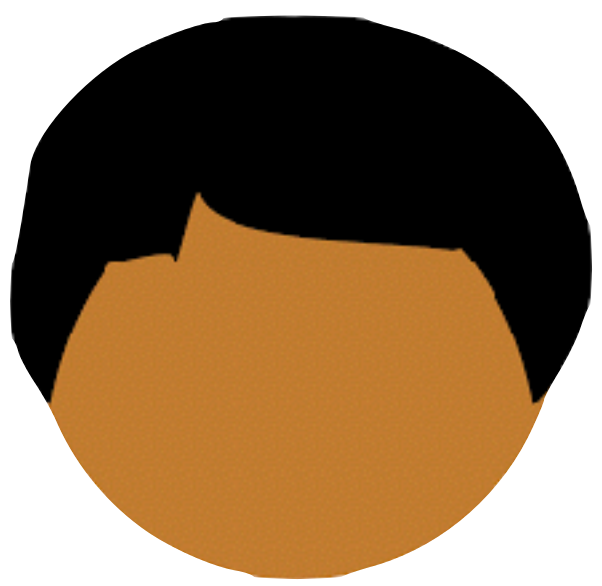




The XOR secret sharing *of a bit x is a pair of bits $\langle x_0, x_1 \rangle$ where P_0 holds x_0 and P_1 holds x_1, and where $x_0 \oplus x_1 = x$*

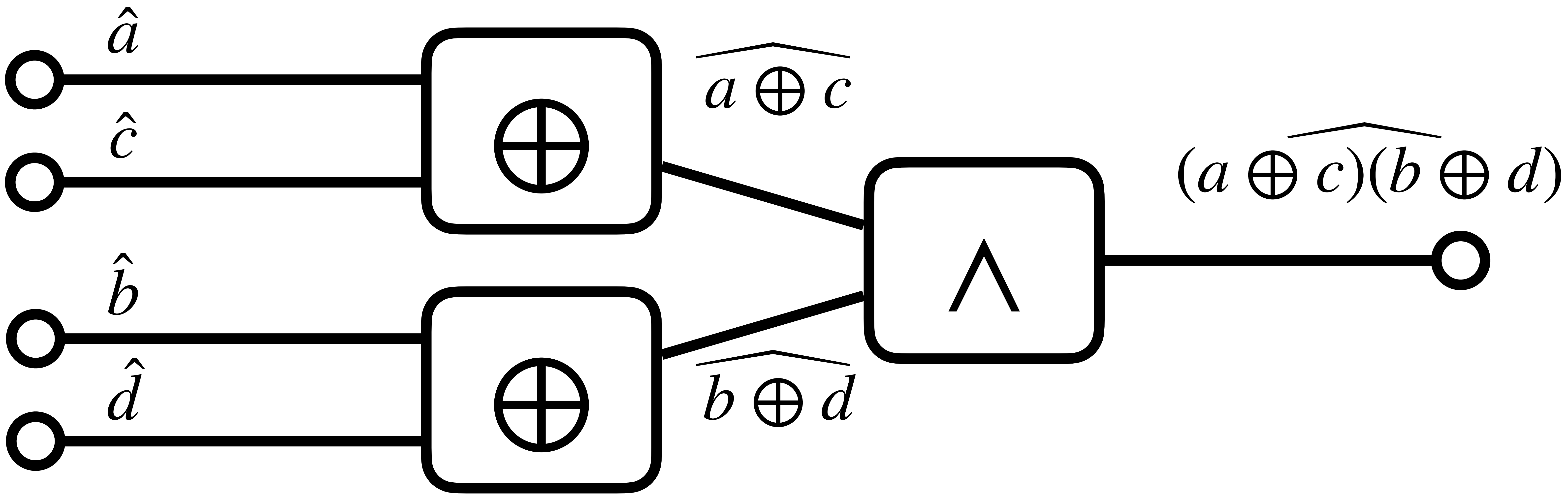
We sometimes denote such a pair by $[x]$

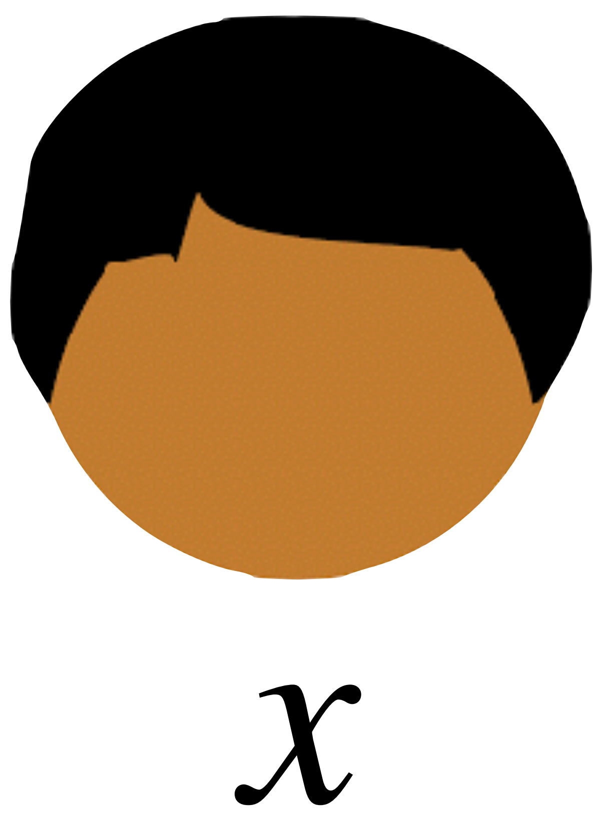
Intuition: P_0's share x_0 acts as a mask, hiding x from P_1 (and vice versa)

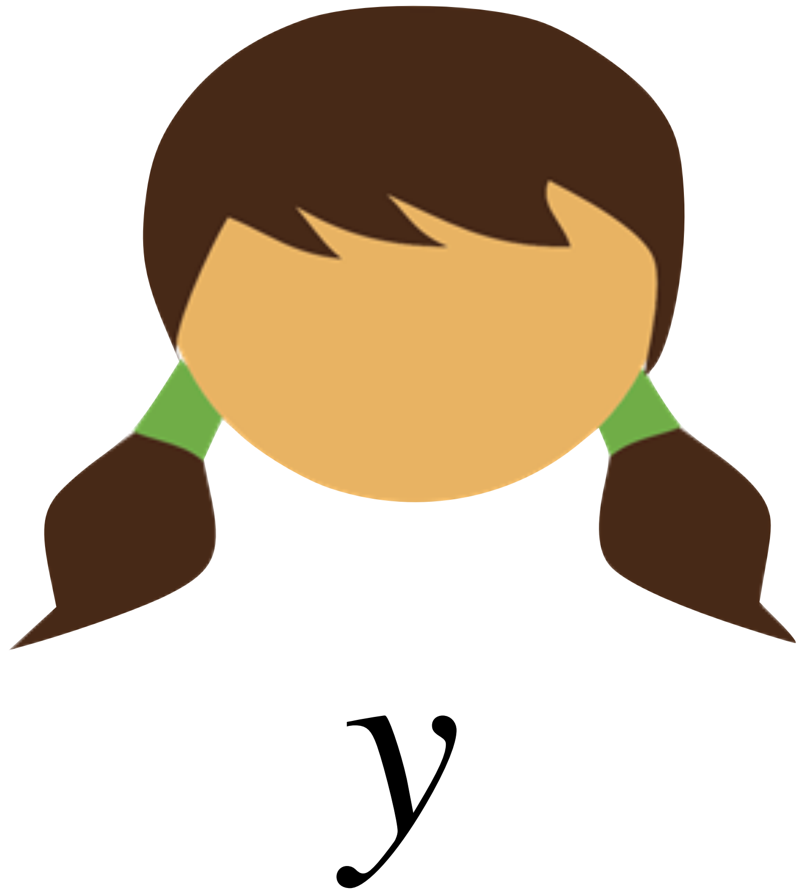


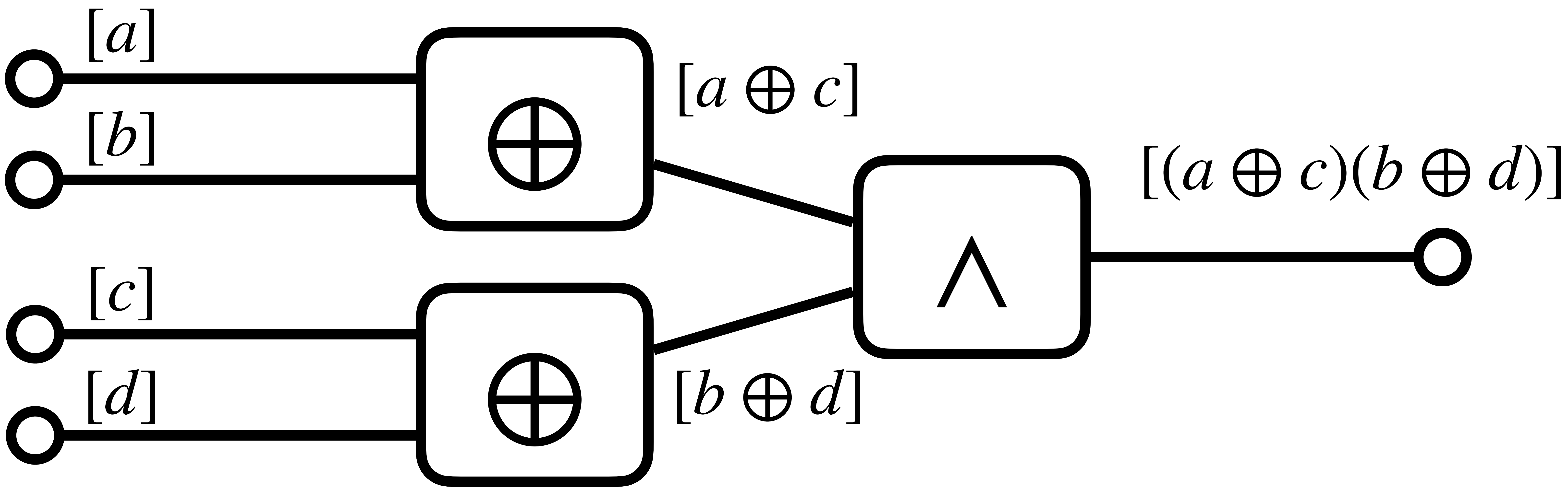
x

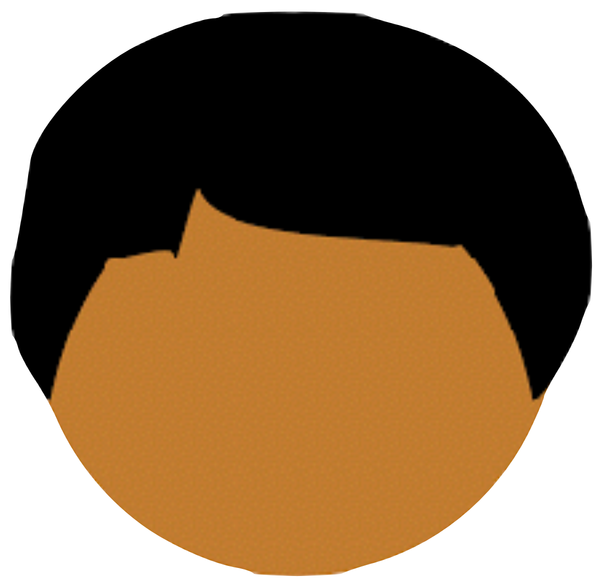
y


$\hat{a}$
$\hat{c}$
$\hat{b}$
$\hat{d}$
$\widehat{a \oplus c}$
$\widehat{b \oplus d}$
$\oplus$
$\oplus$
$\wedge$
$\widehat{(a \oplus c)(b \oplus d)}$

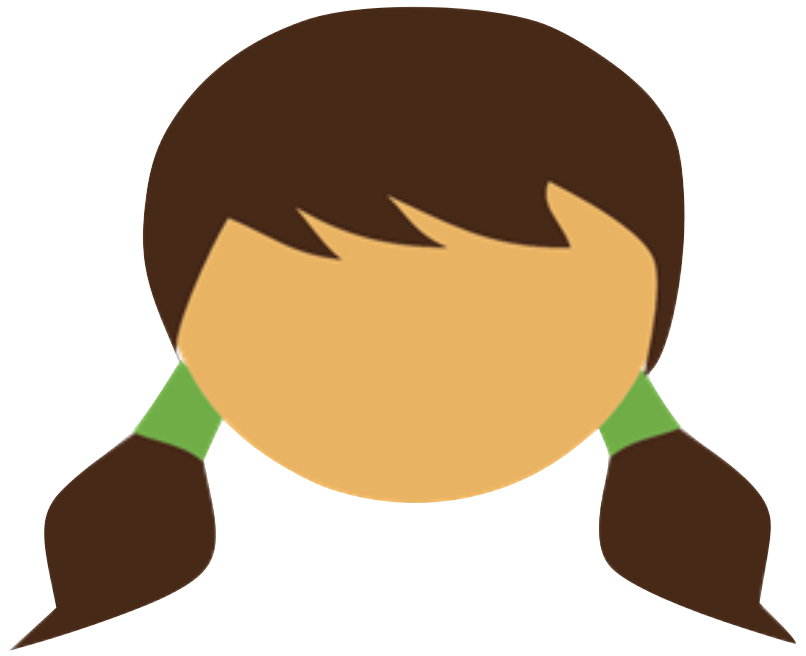

x


y

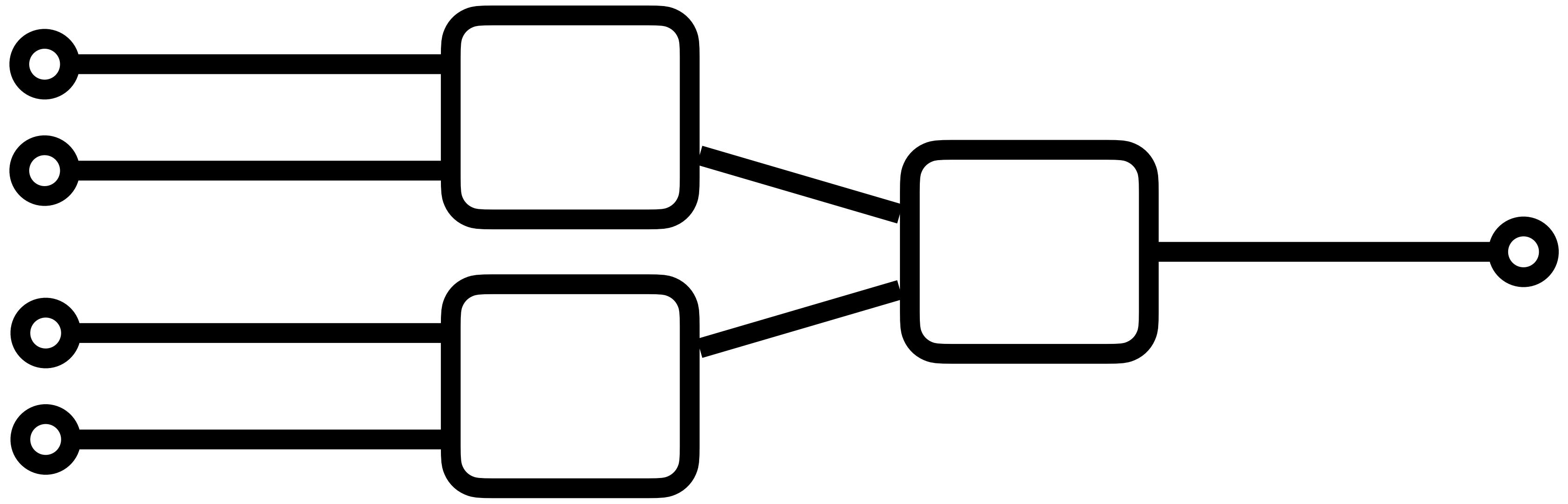

[a]
[b]
[c]
[d]
⊕
⊕
[a ⊕ c]
[b ⊕ d]
∧
[(a ⊕ c)(b ⊕ d)]

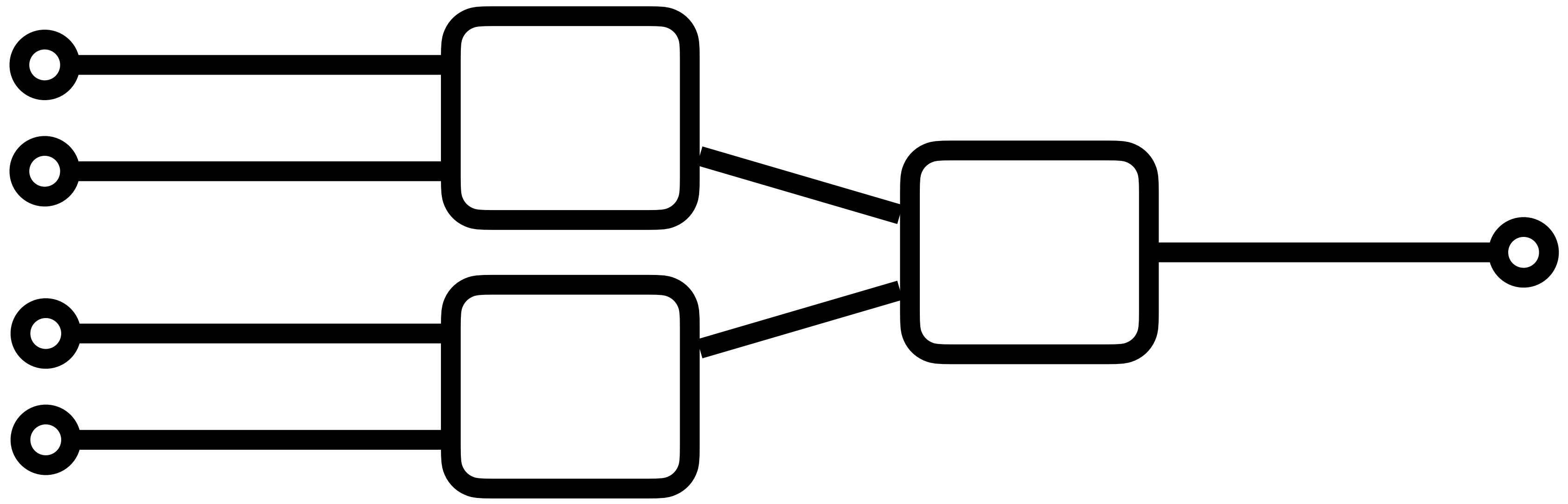


x

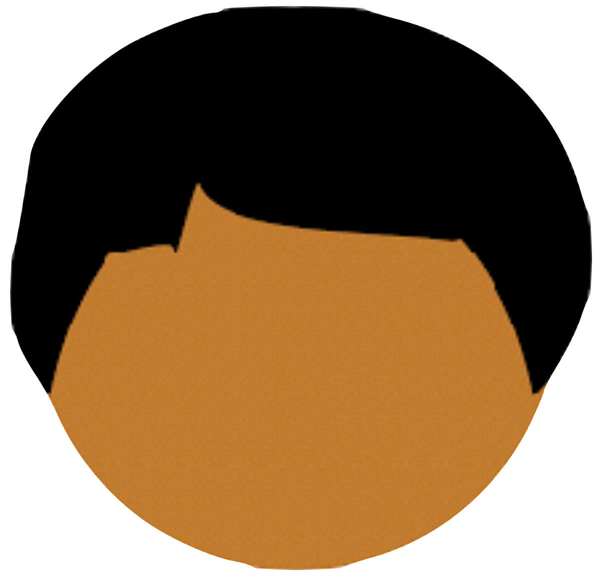


y





Each party in its head maintains a local copy of the circuit, placing its shares on the wires



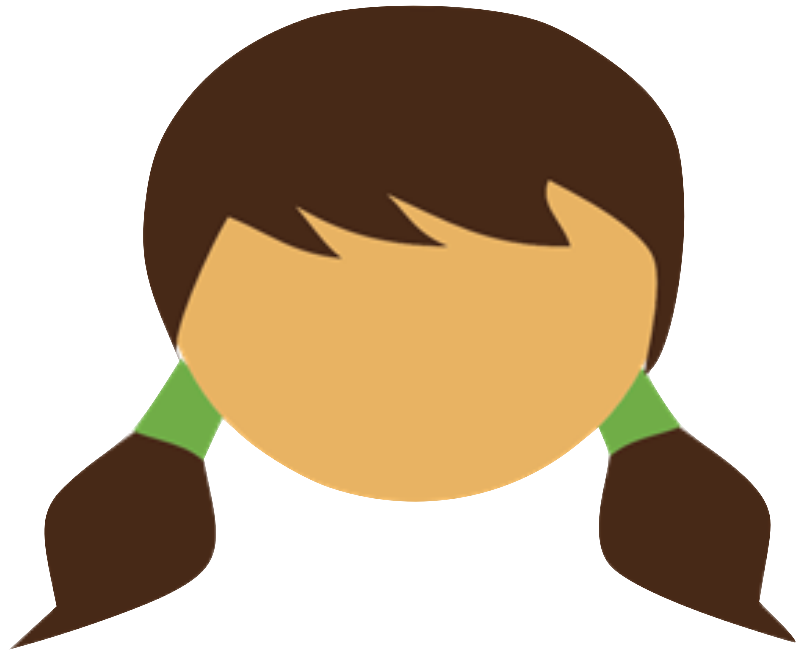
x

Where do input shares come from?

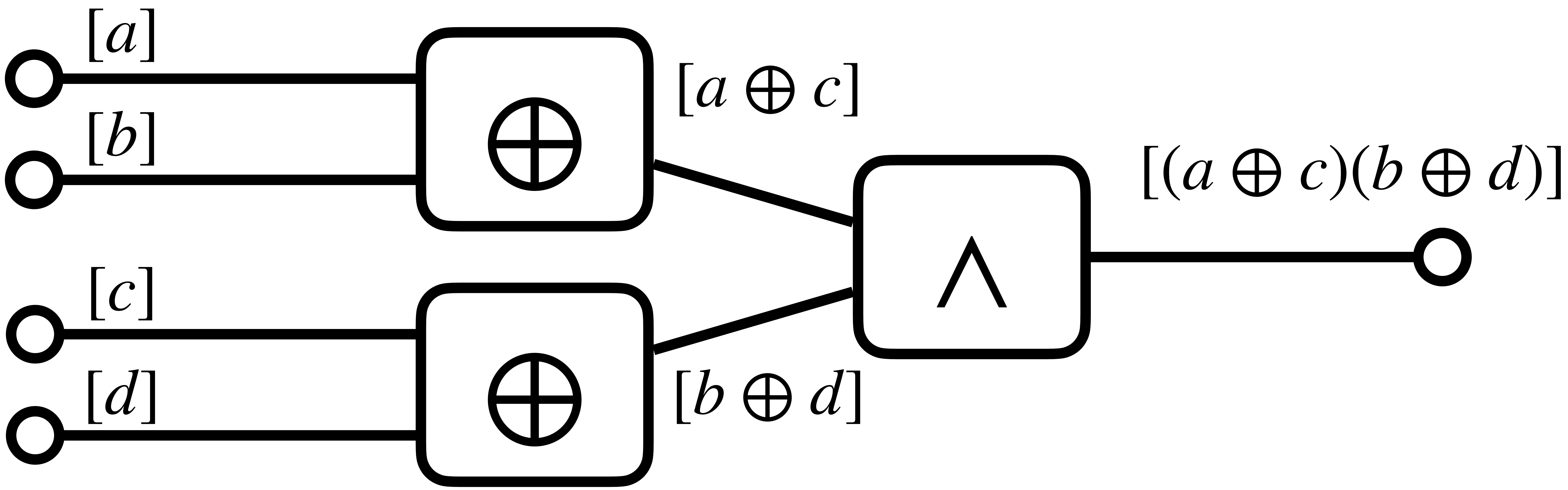
How do we XOR two shares?

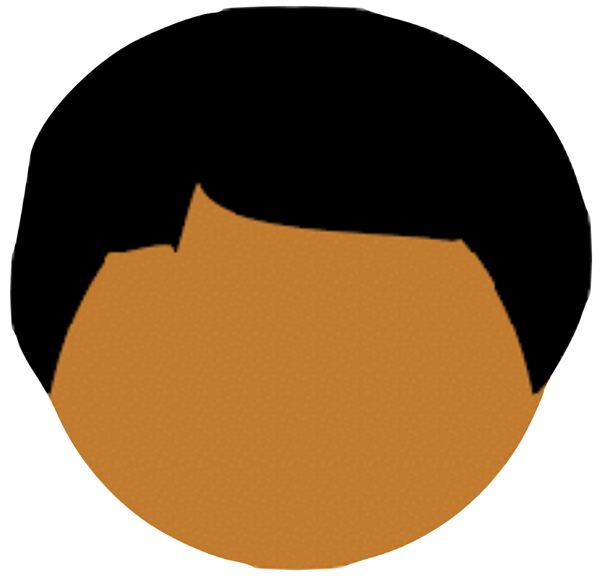
How do we AND two shares?

How do we "decrypt" output shares?



y

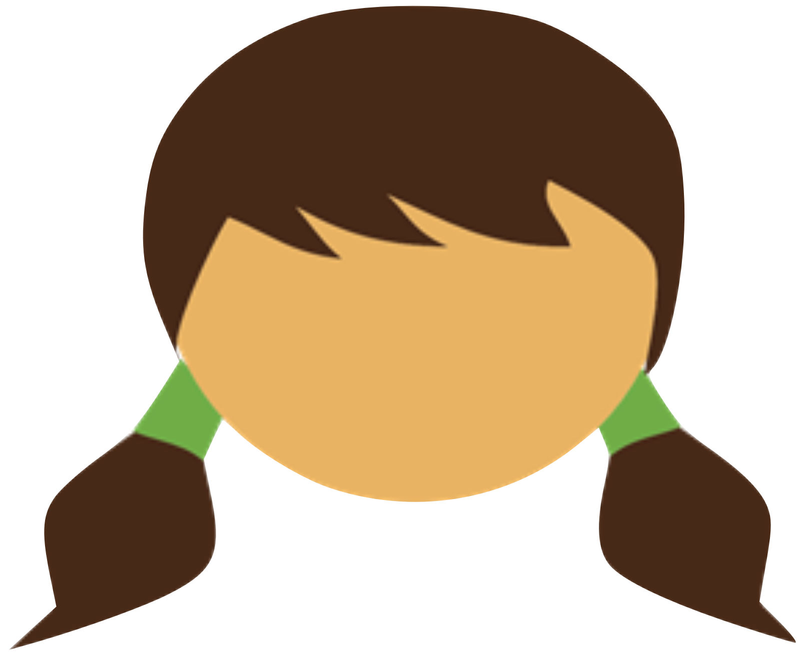




x

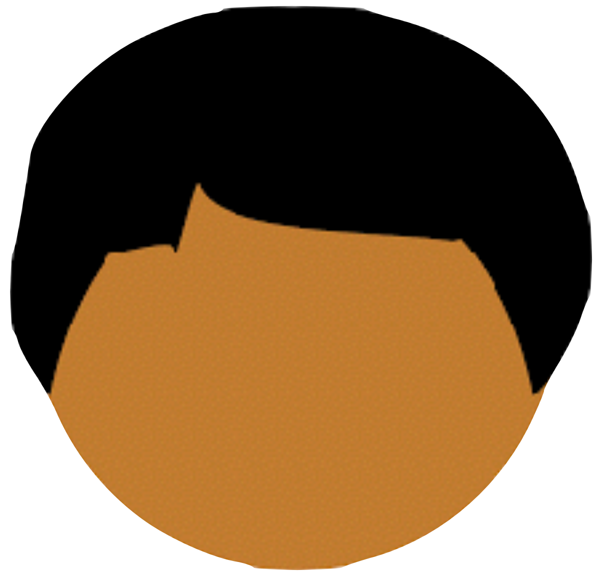
Where do input shares come from?

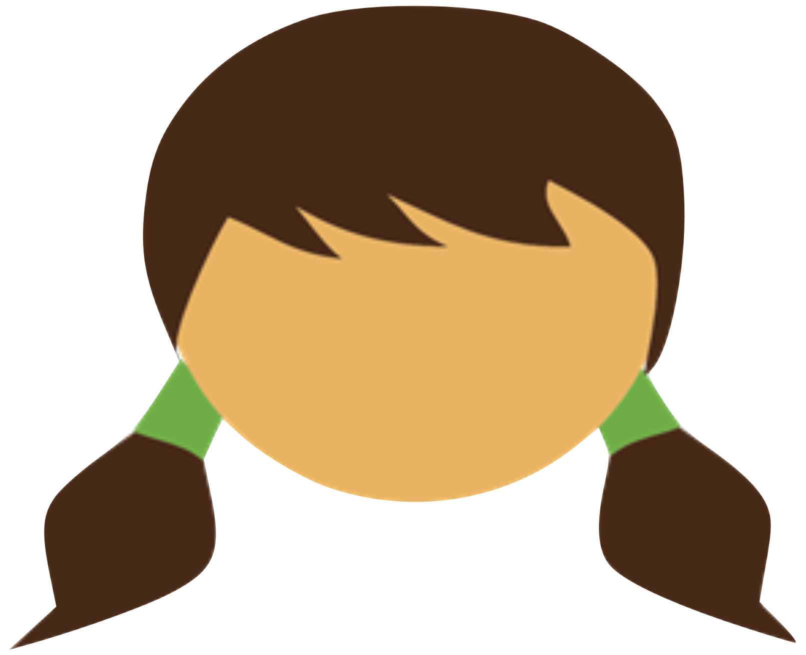
Goal: put $[x]$ on the input wire



Where do input shares come from?

Goal: put $[x]$ on the input wire



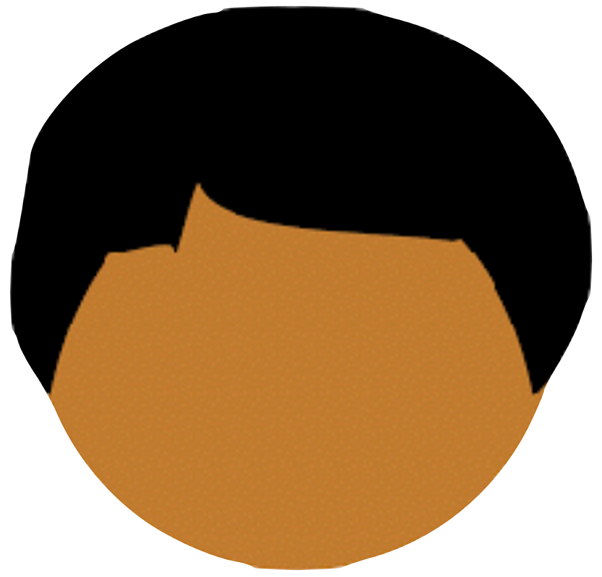


x

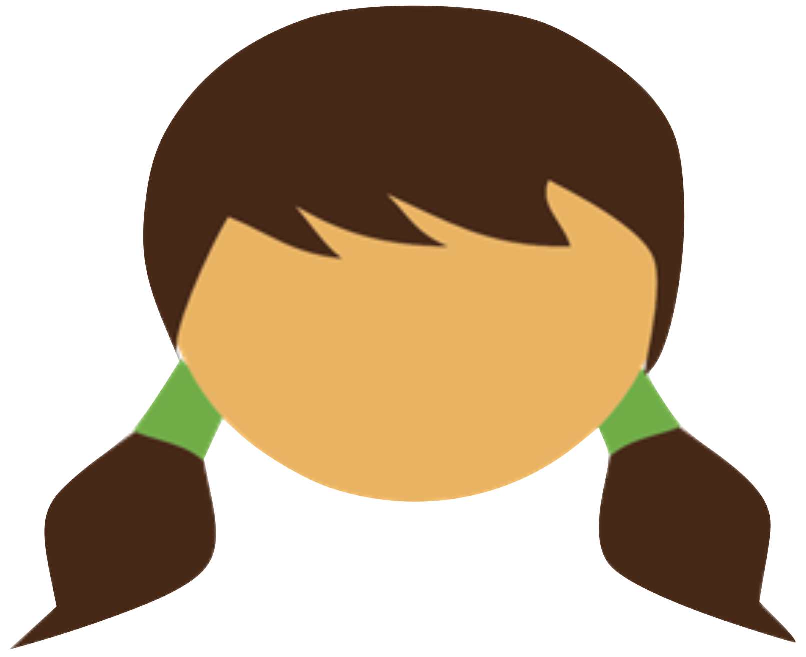
$$r \xleftarrow{\$} \{0,1\}$$

Where do input shares come from?

Goal: put $[x]$ on the input wire

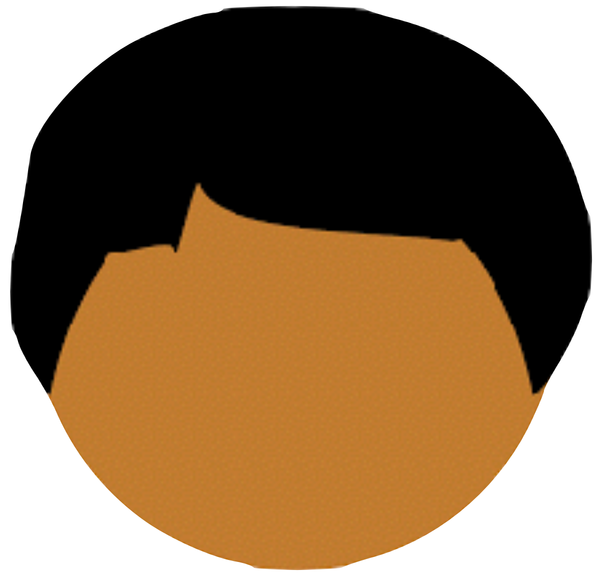


x



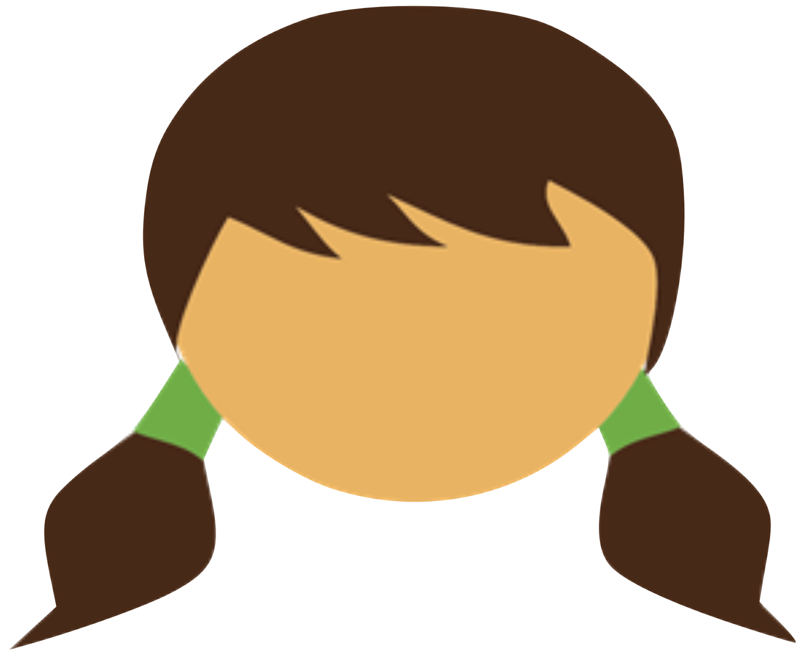
$$r \xleftarrow{\$} \{0,1\}$$

$$x \oplus r$$



Where do input shares come from?

Goal: put $[x]$ on the input wire



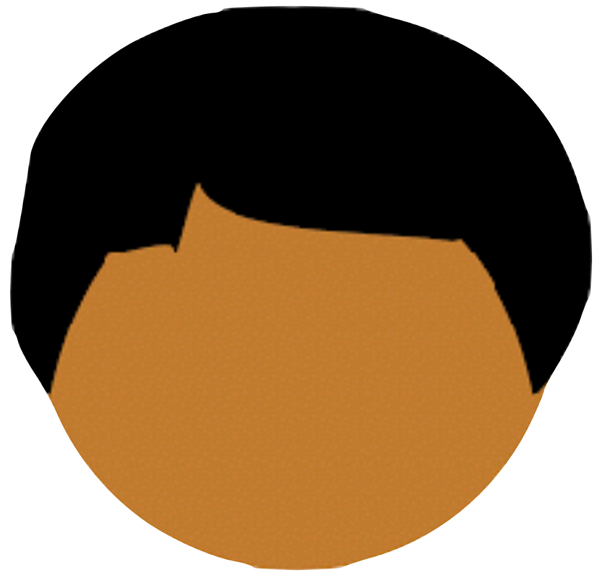
x

$r \xleftarrow{\$} \{0,1\}$

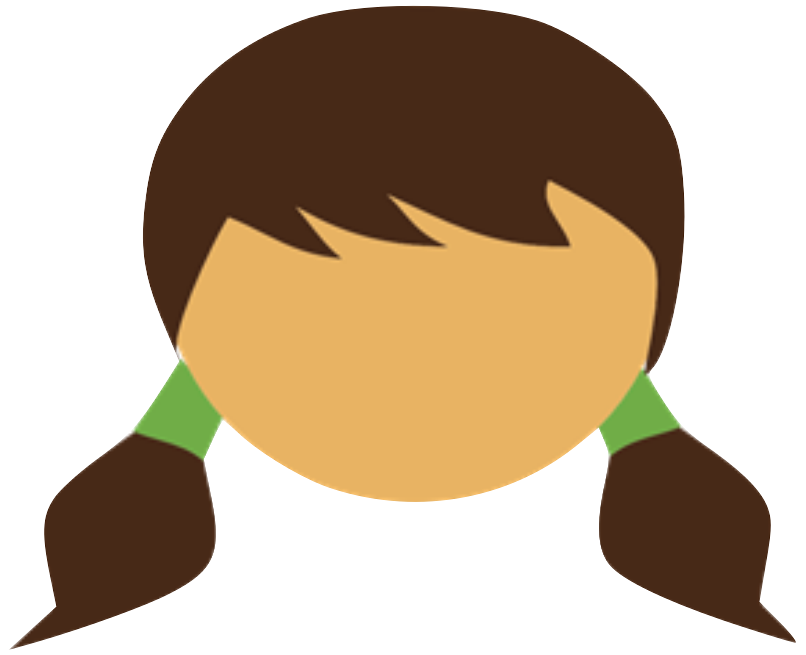
$x \oplus r$

r

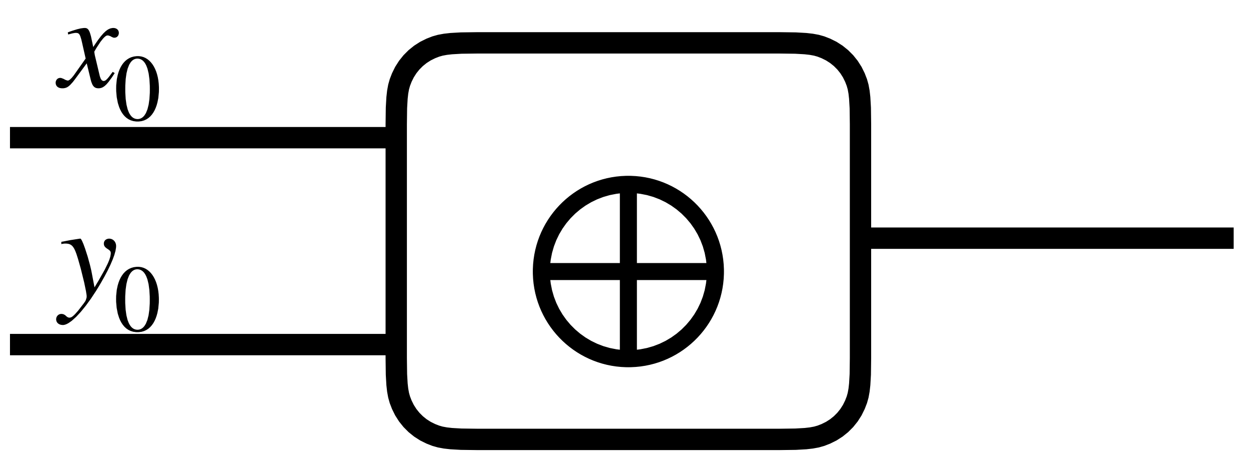
$x \oplus r$

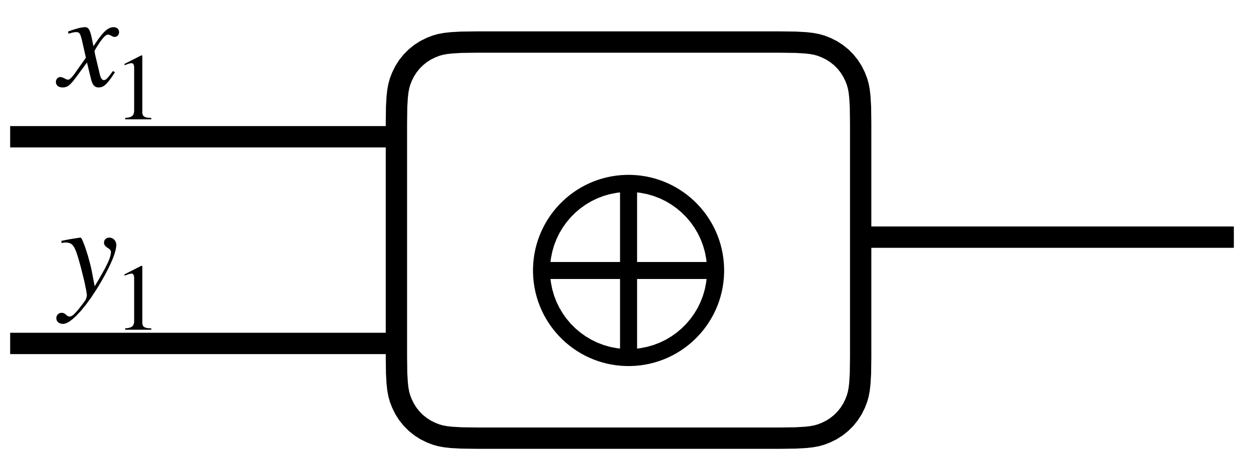


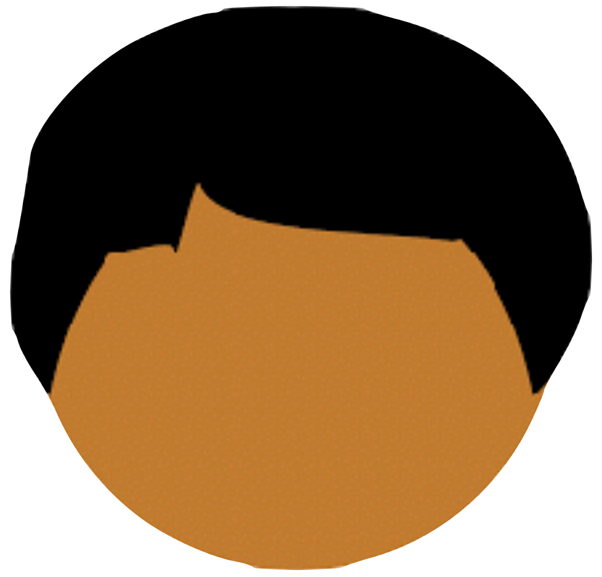
How do we XOR two shares?



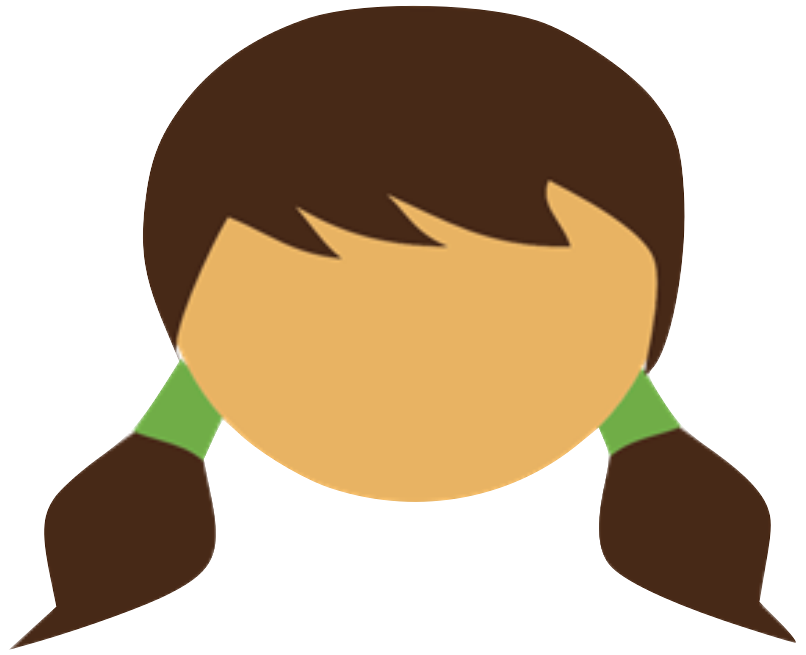
Goal: given gate input wires holding $[x]$, $[y]$, put $[x \oplus y]$ on the gate output



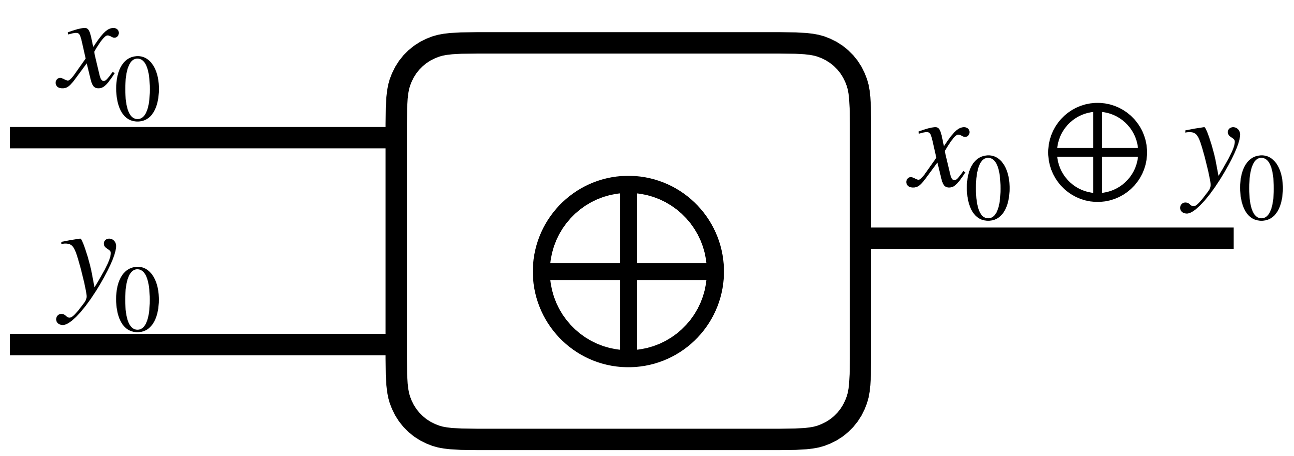


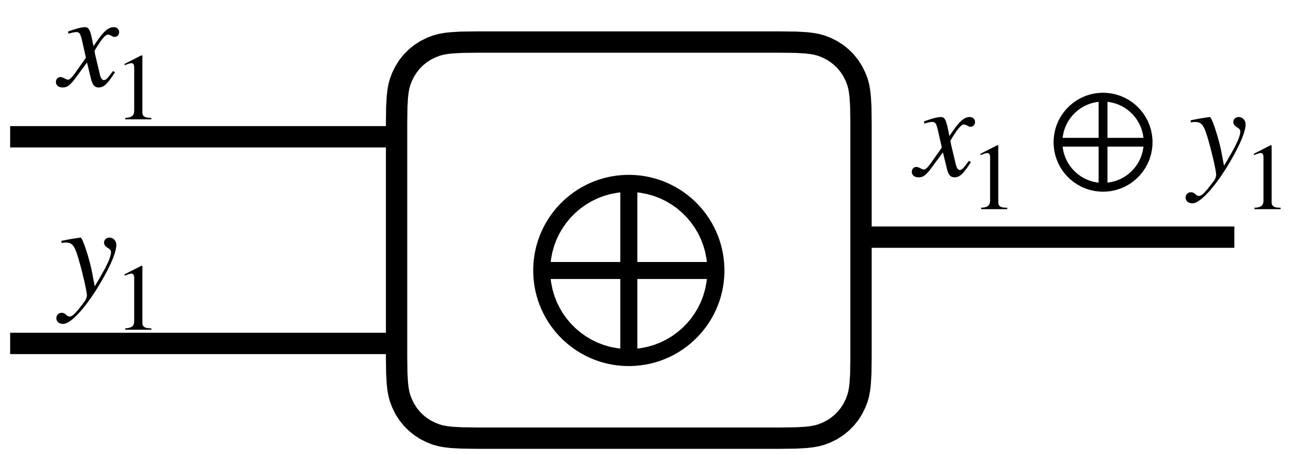


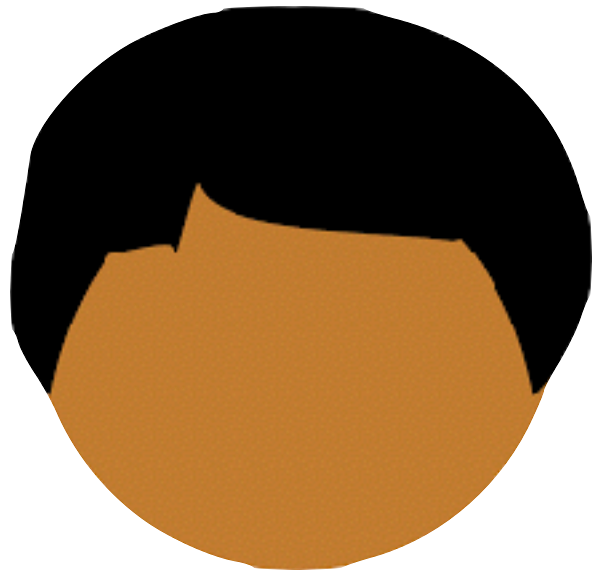
How do we XOR two shares?



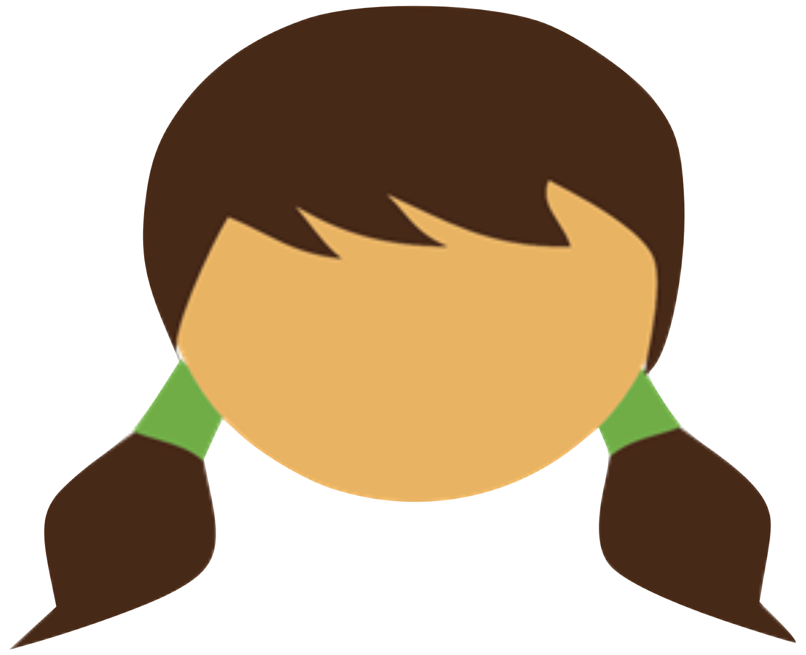
Goal: given gate input wires holding $[x]$, $[y]$, put $[x \oplus y]$ on the gate output



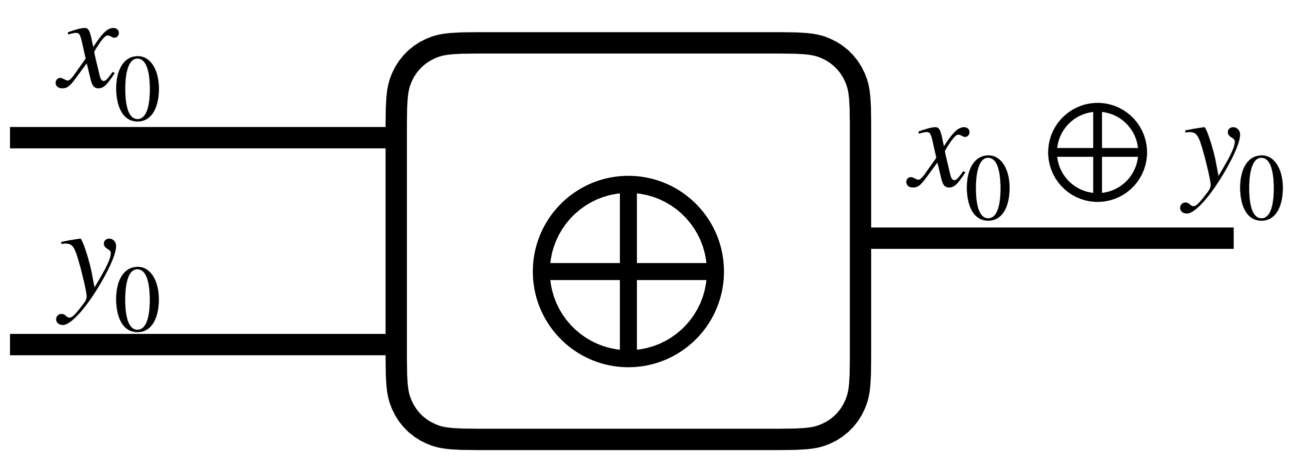




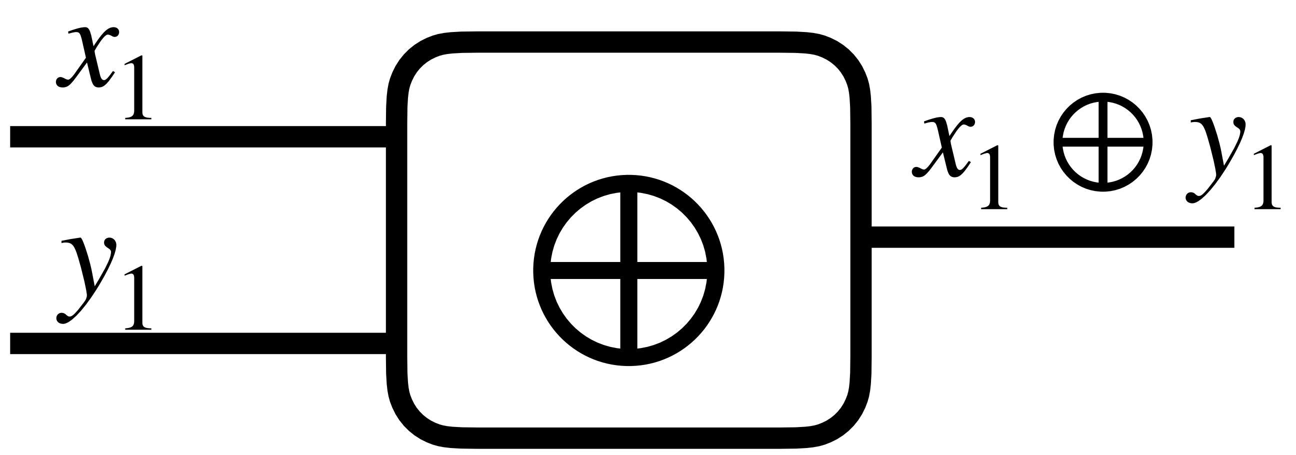
How do we XOR two shares?

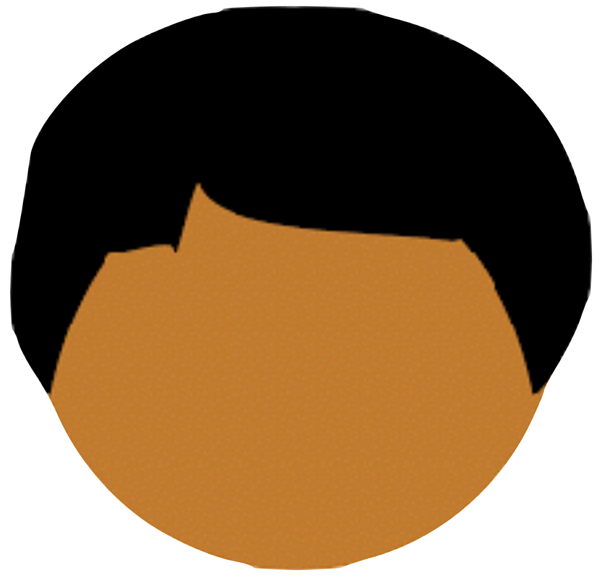


Goal: given gate input wires holding $[x]$, $[y]$, put $[x \oplus y]$ on the gate output

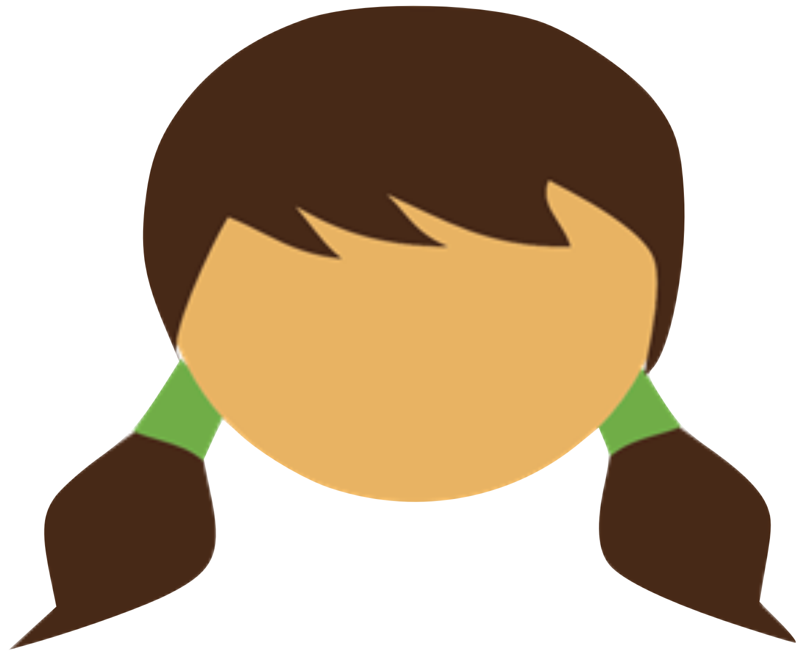


XOR is “free”



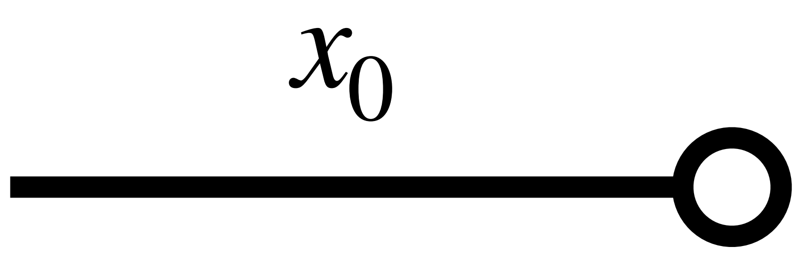


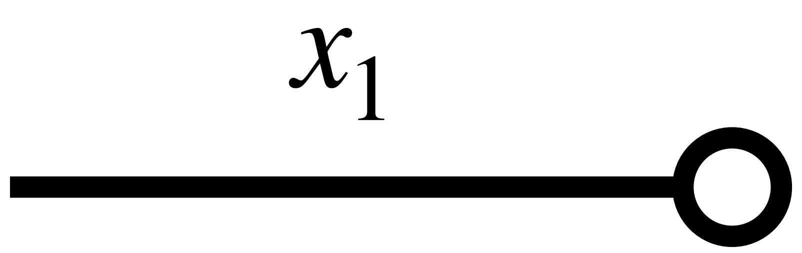
How do we “decrypt” output shares?

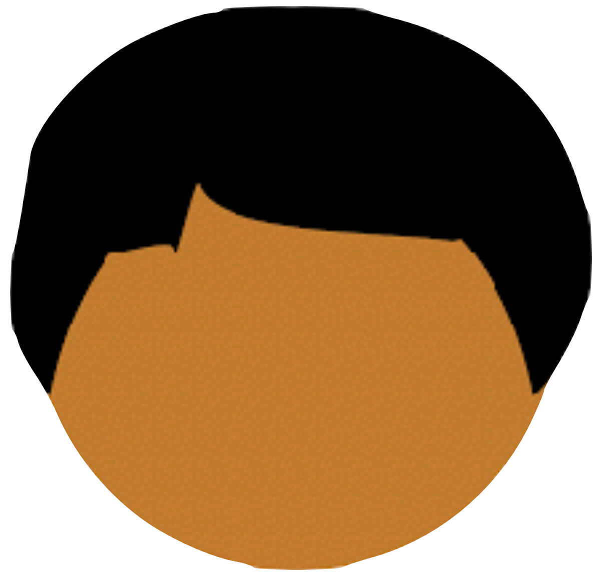


Goal: given wire holding $[x]$,

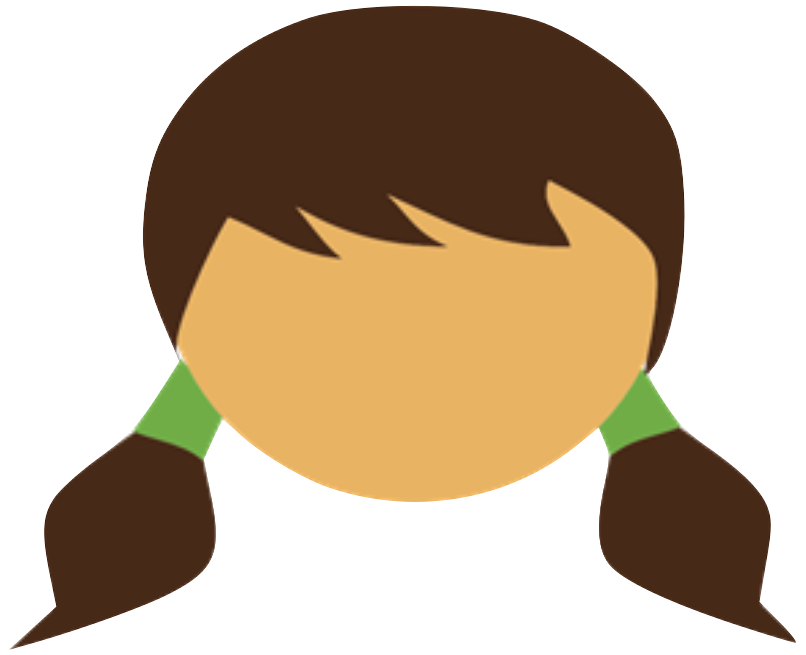
reveal x to each party



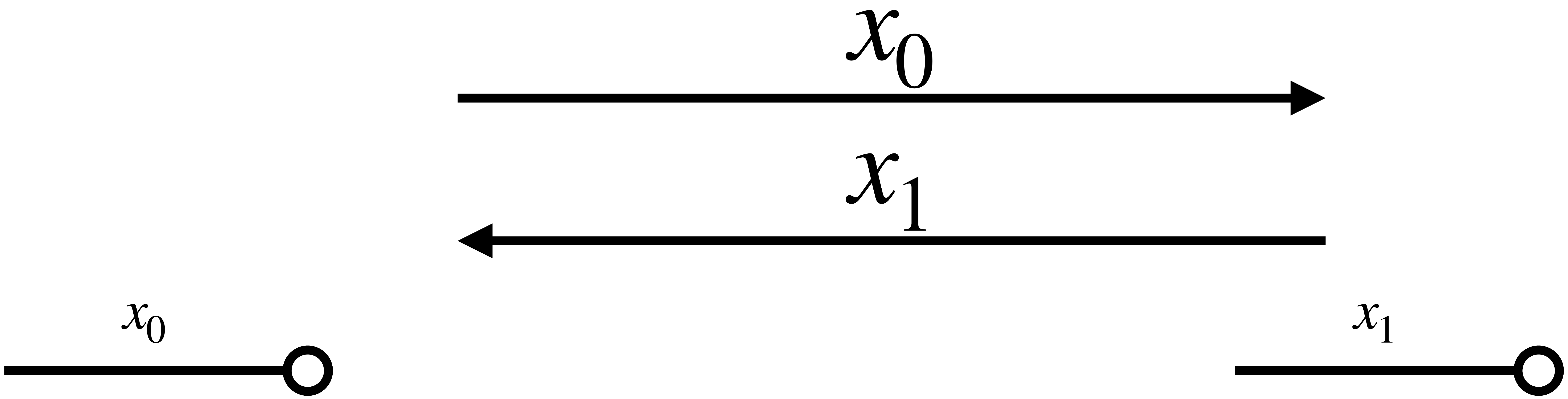


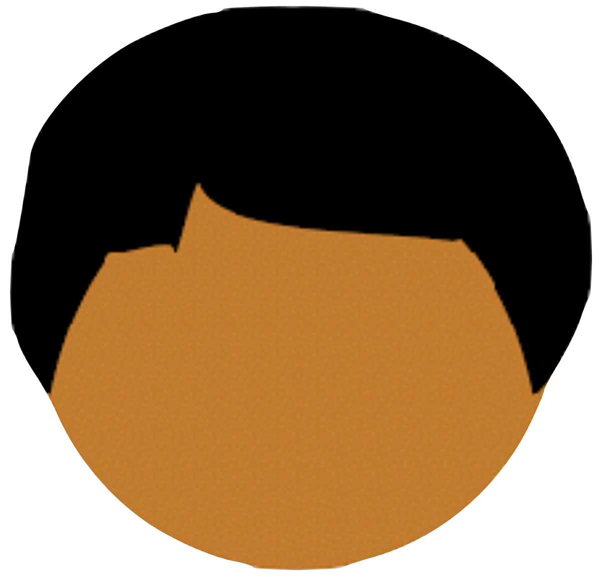


How do we “decrypt” output shares?



Goal: given wire holding $[x]$,
reveal x to each party



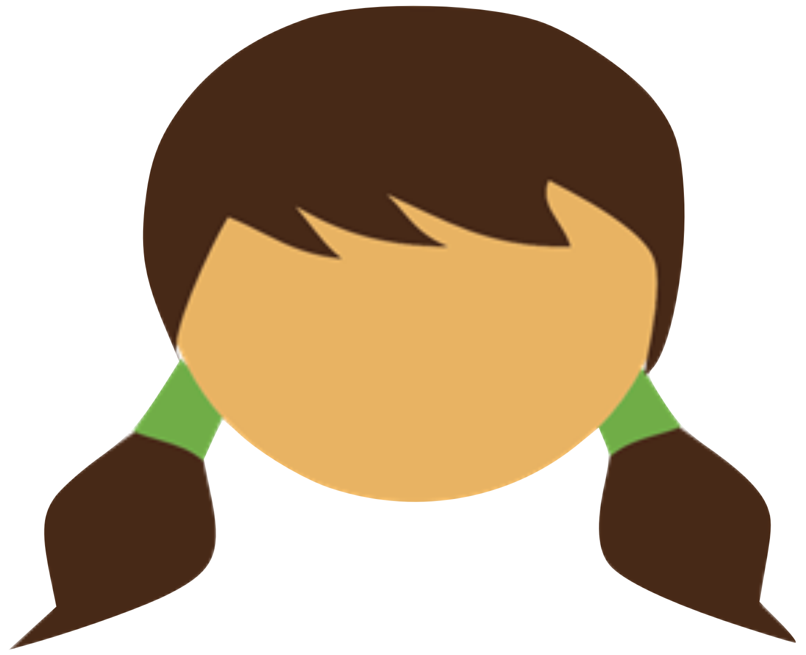


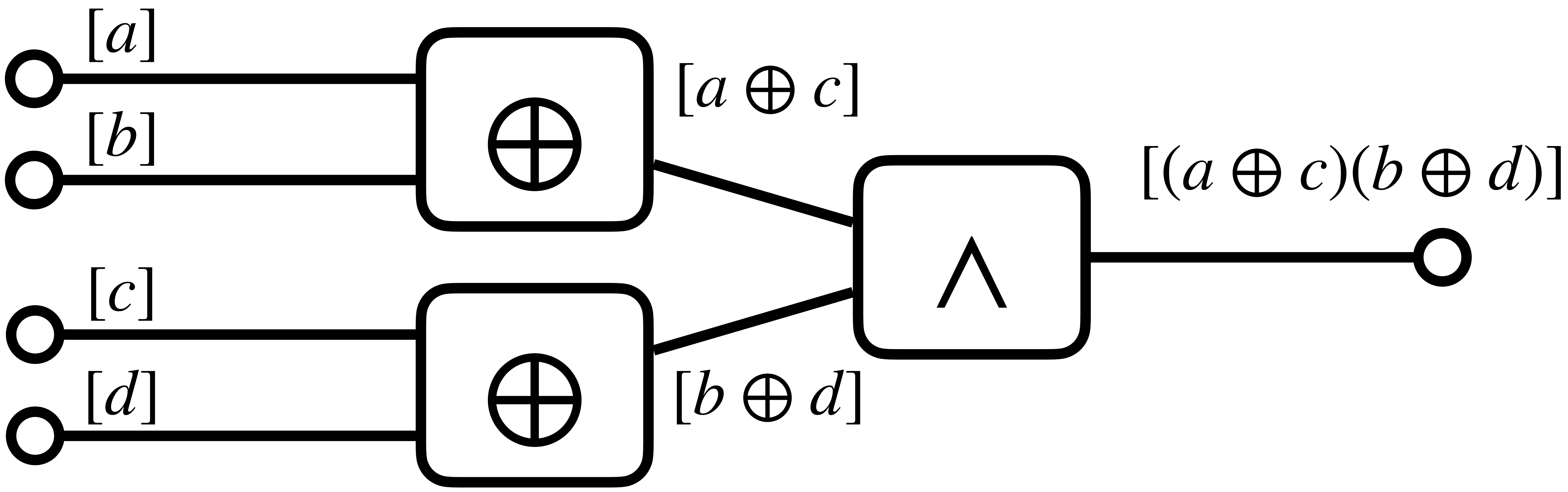
Where do input shares come from?

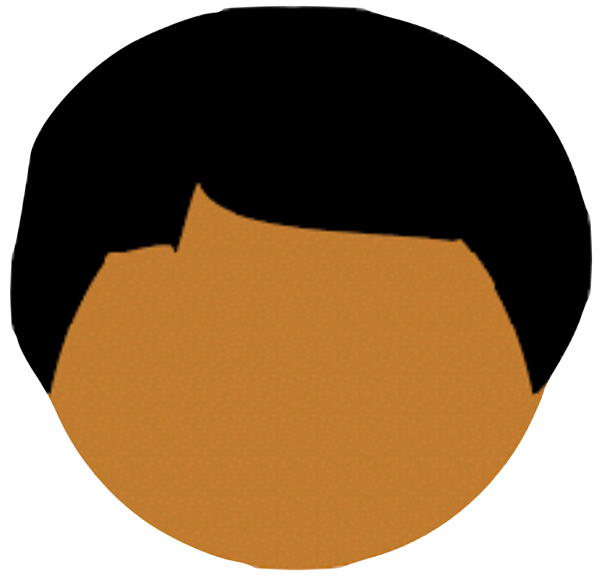
How do we XOR two shares?

How do we AND two shares?

How do we “decrypt” output shares?

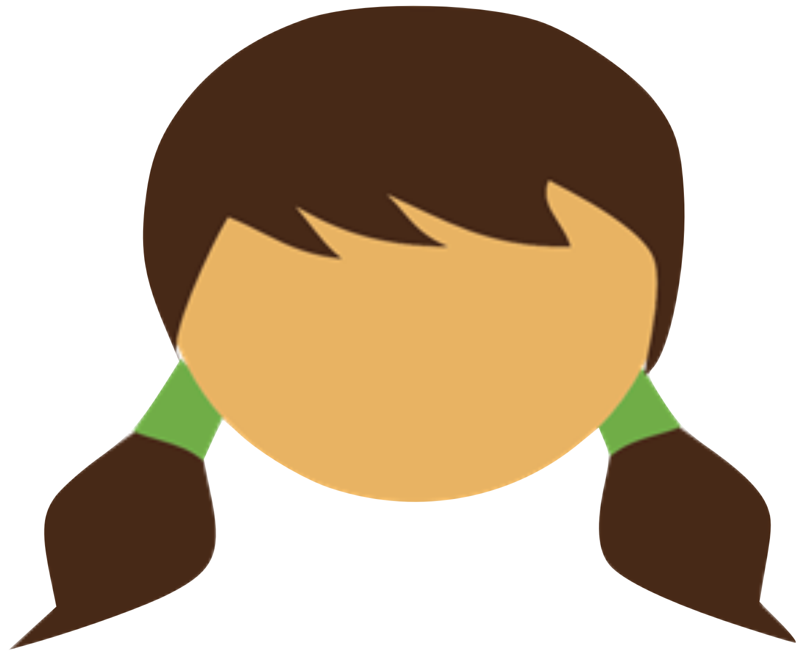


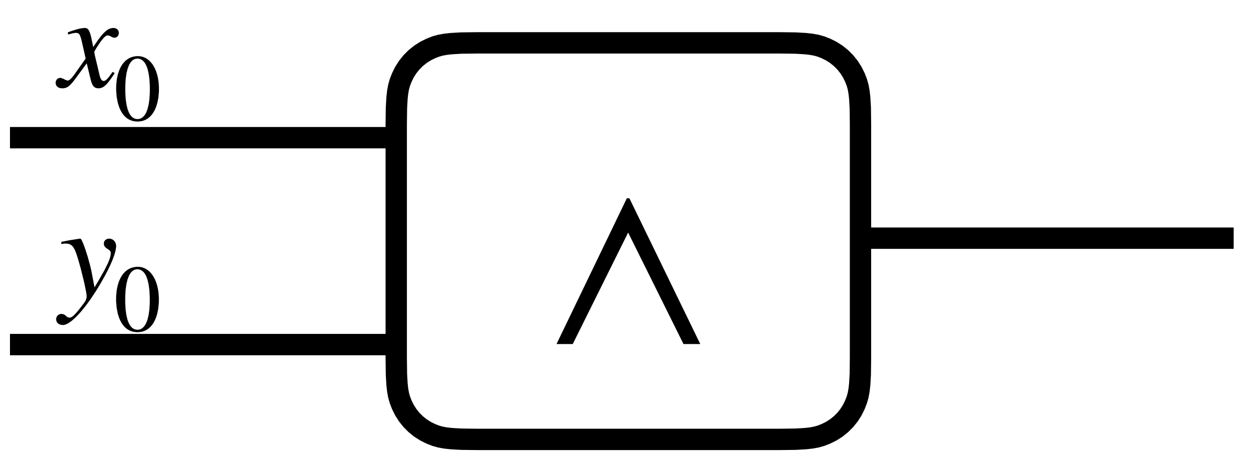


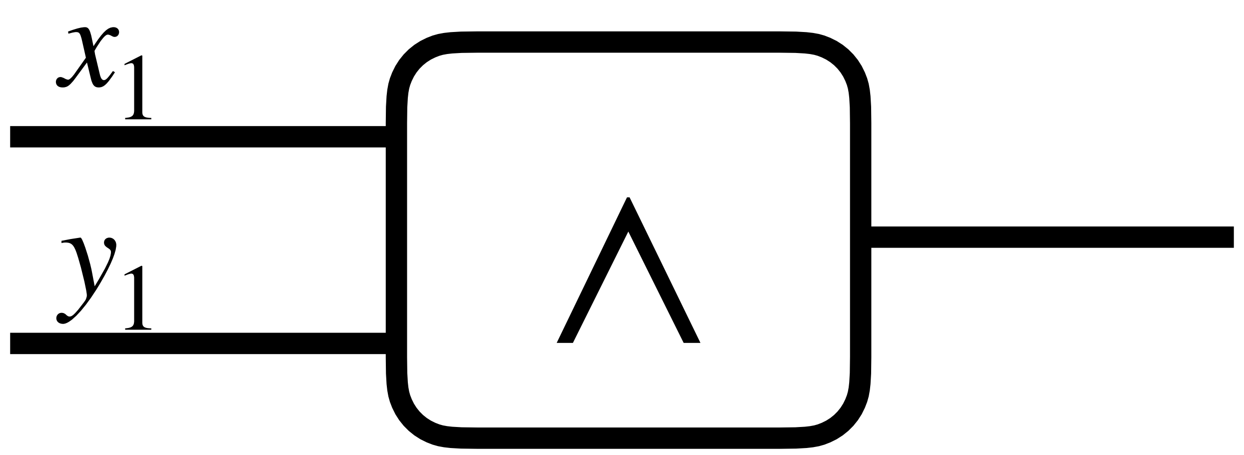


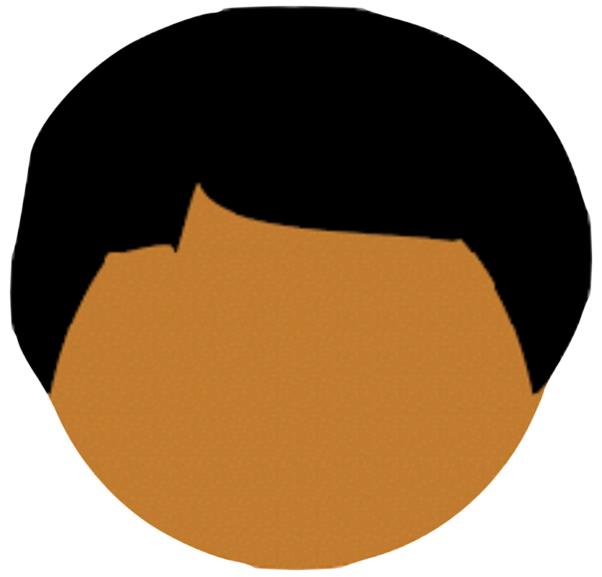
How do we AND two shares?

Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output

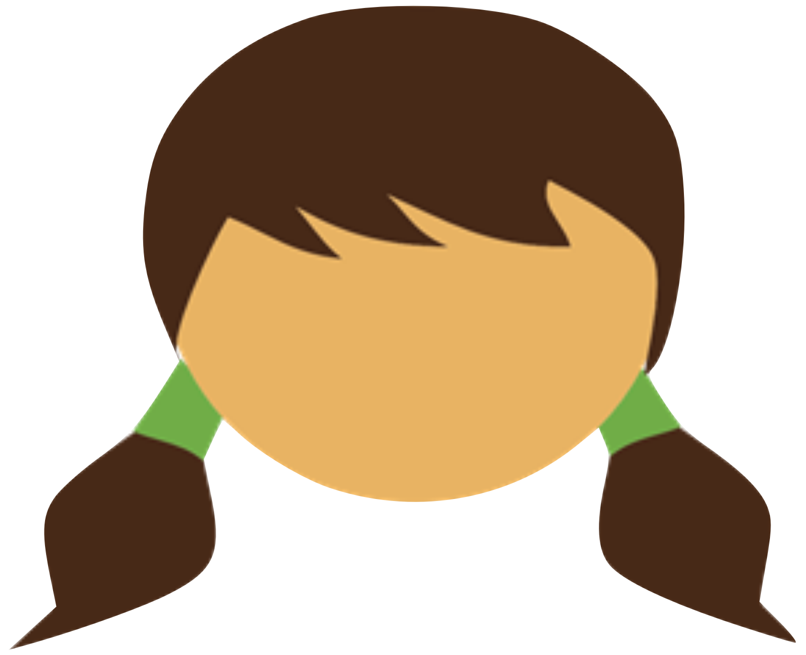






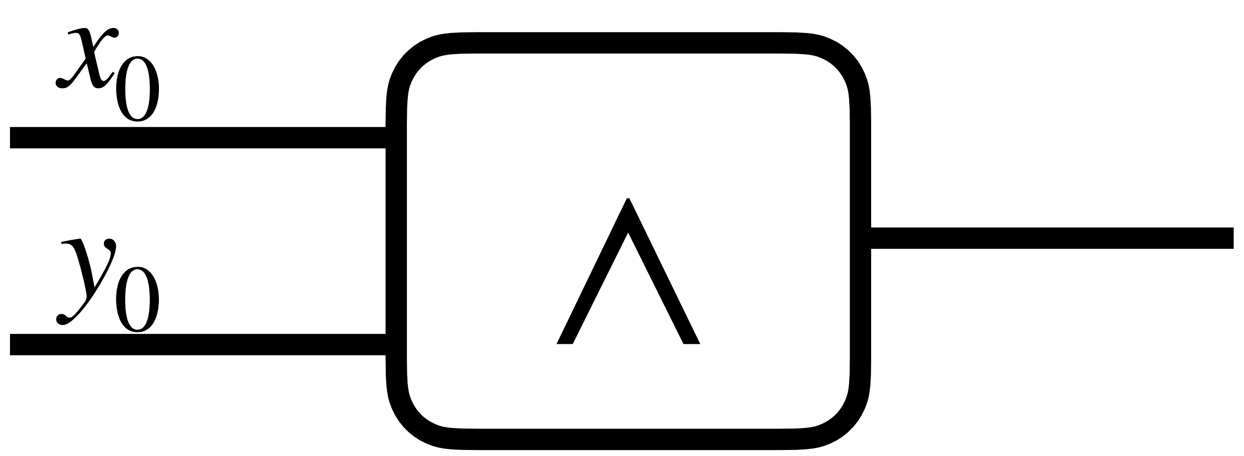


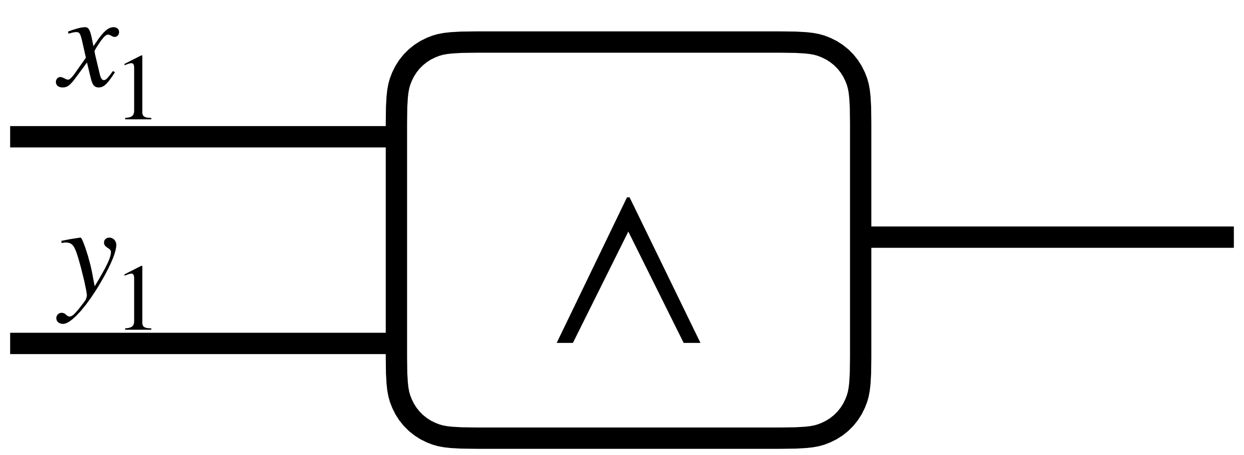
How do we AND two shares?

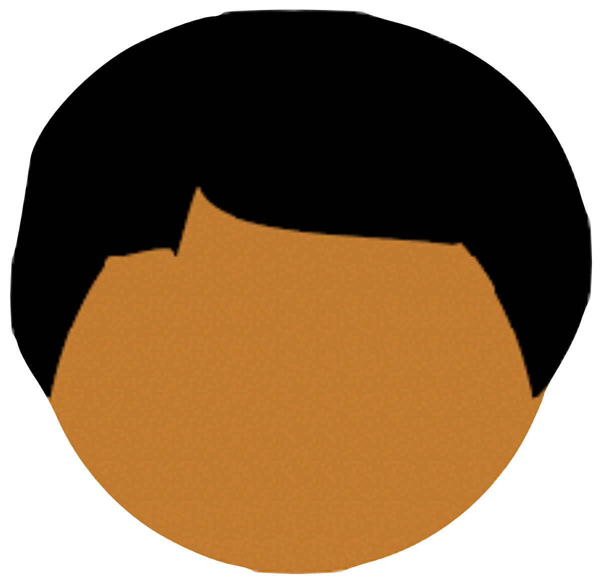


Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output

$$(x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$$

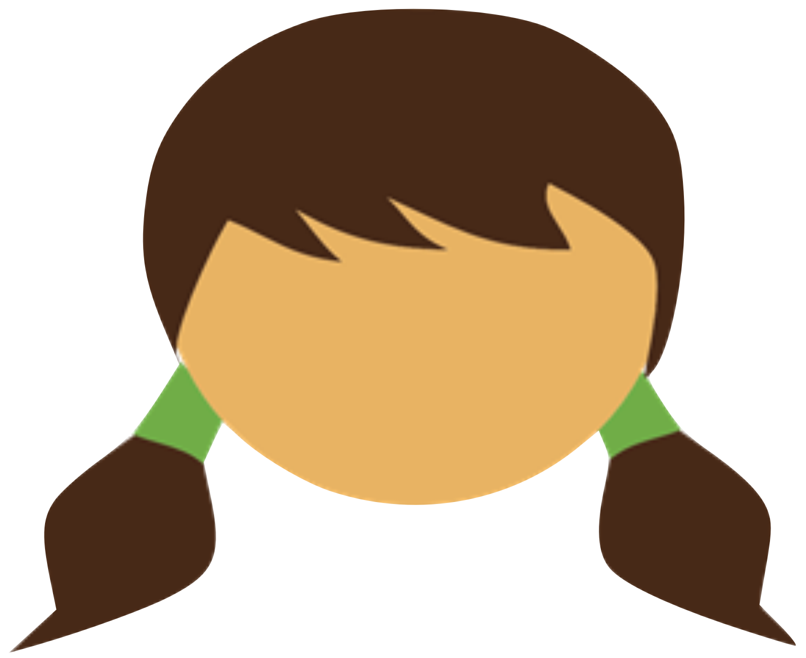




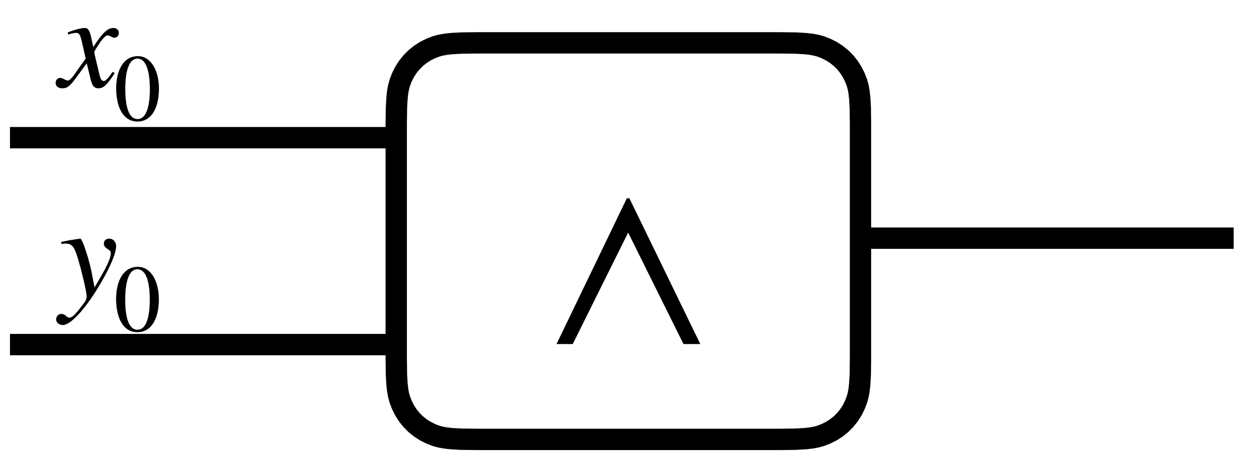


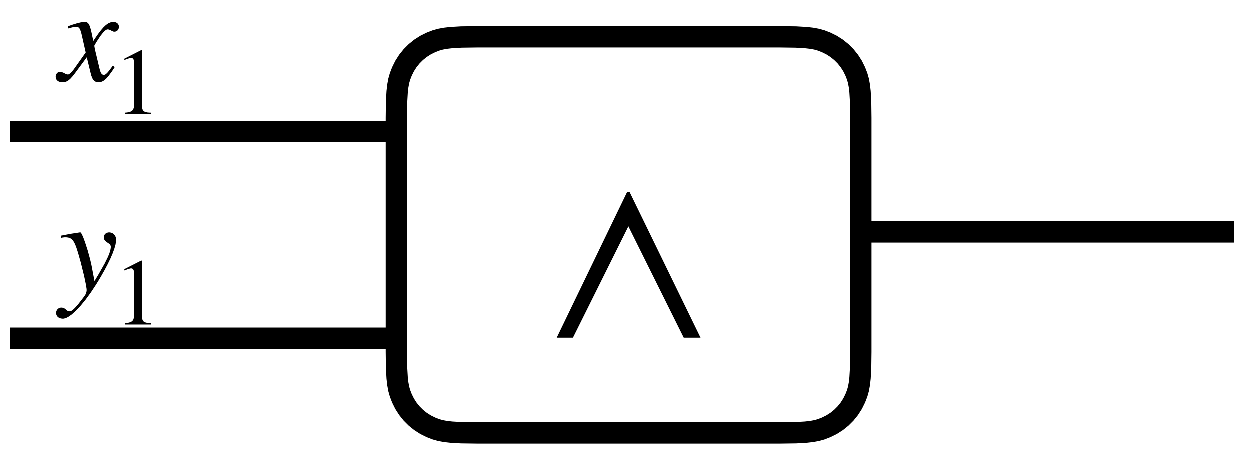
How do we AND two shares?

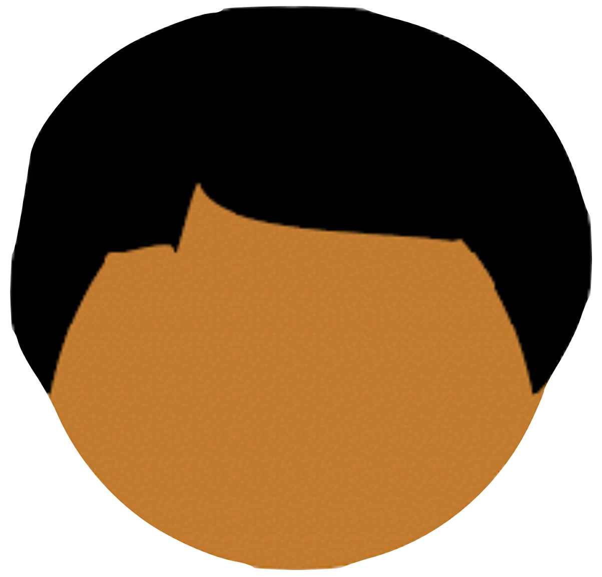
Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output



$$(x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$$
$$= (x_0 \wedge y_0) \oplus (x_0 \wedge y_1) \oplus (x_1 \wedge y_0) \oplus (x_1 \wedge y_1)$$

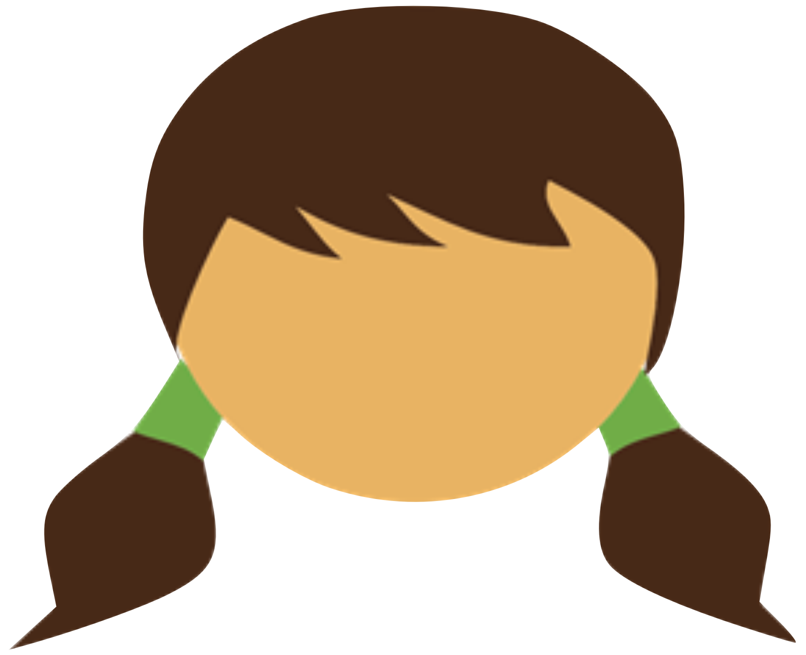






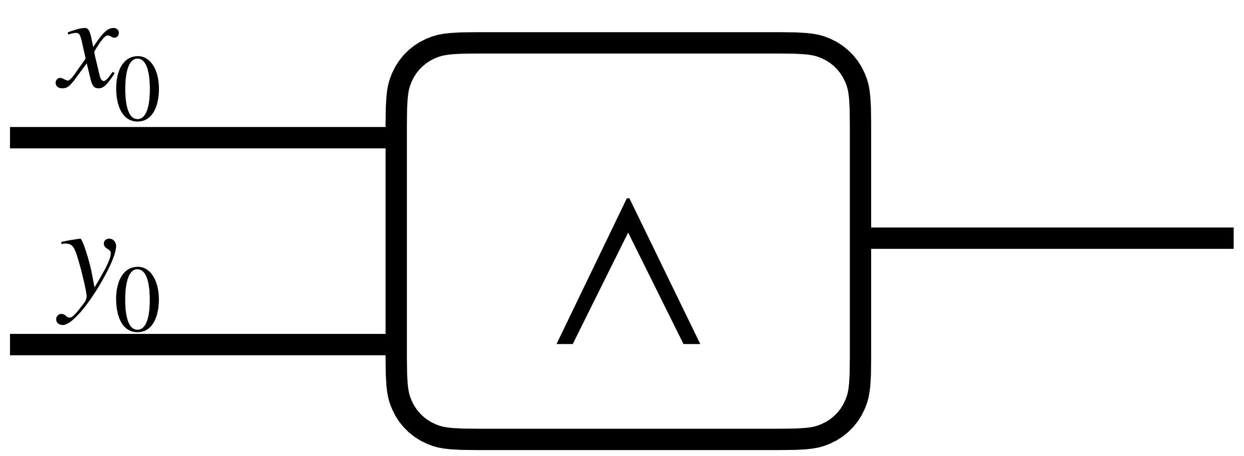
How do we AND two shares?

Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output

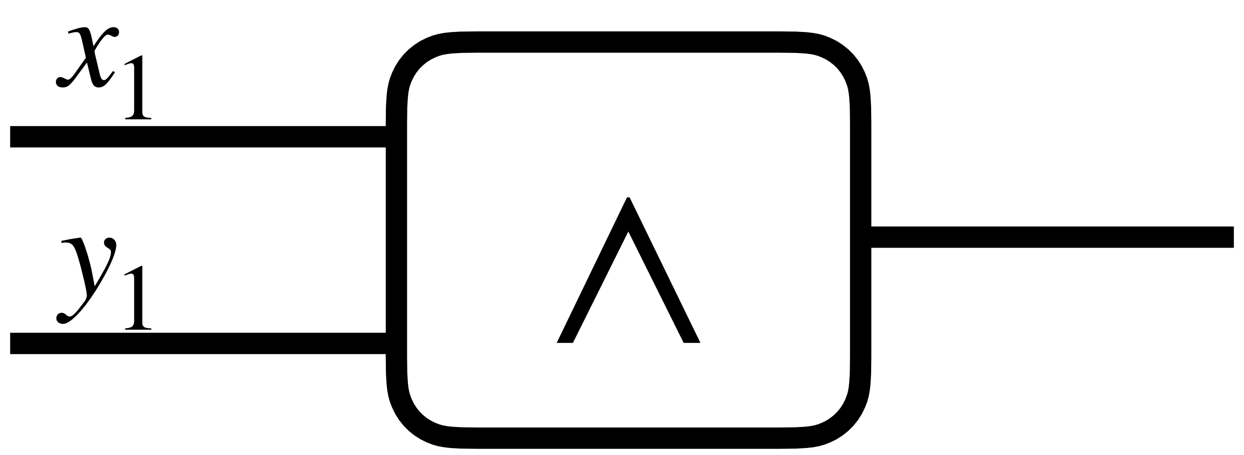


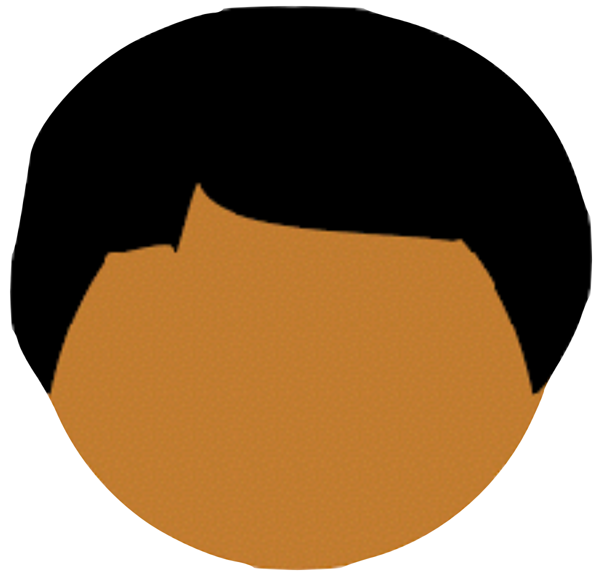
$$(x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$$

$$= \boxed{(x_0 \wedge y_0)} \oplus (x_0 \wedge y_1) \oplus (x_1 \wedge y_0) \oplus \boxed{(x_1 \wedge y_1)}$$

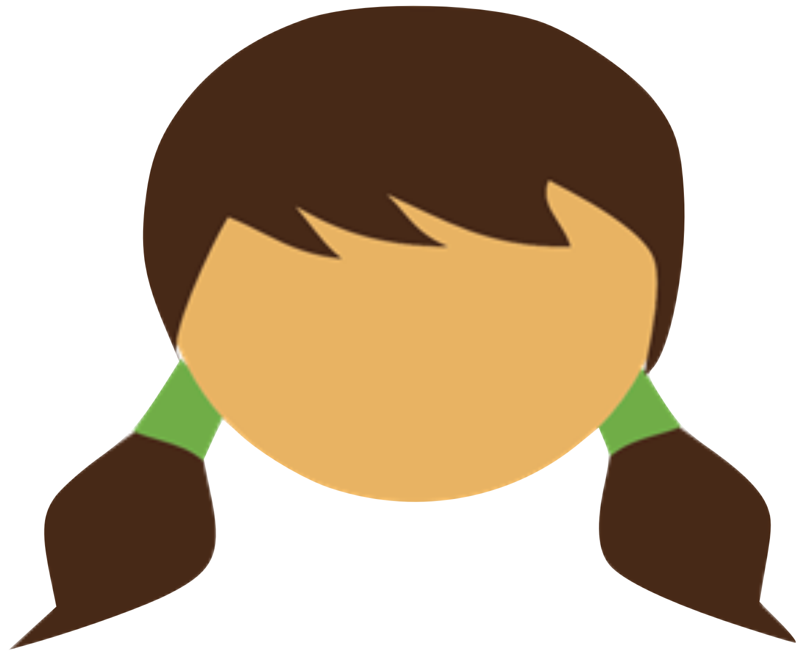


"Free"





How do we AND two shares?

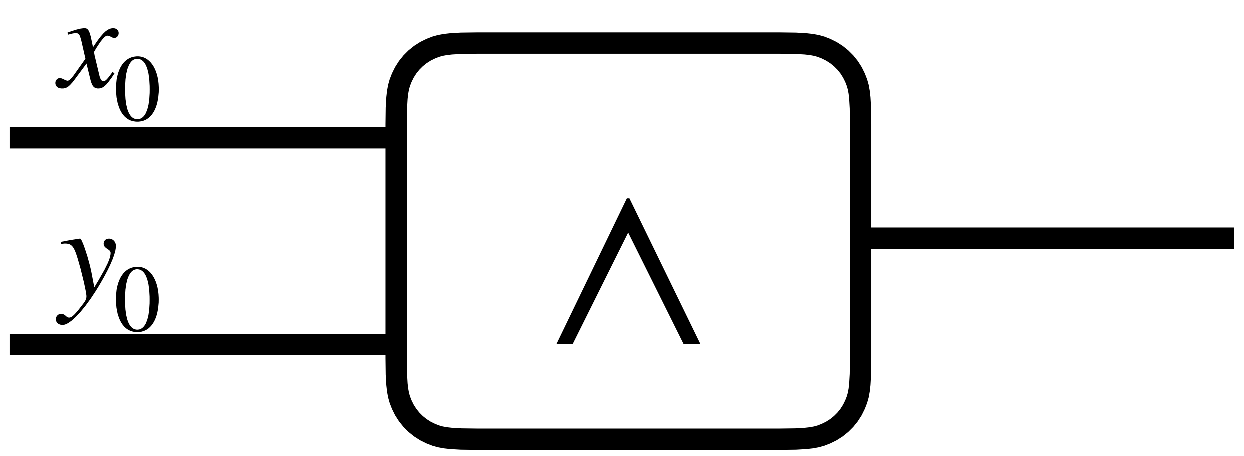


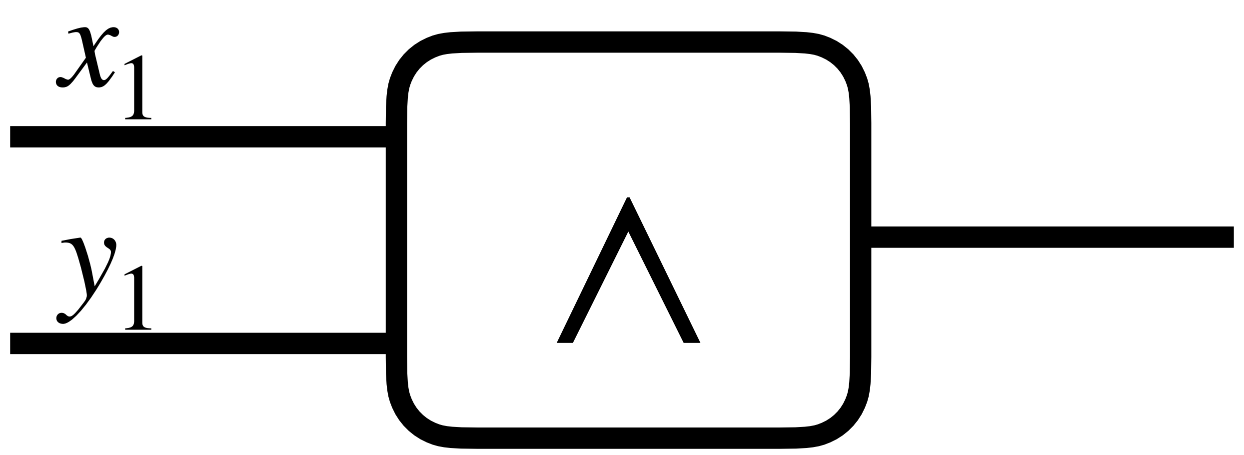
Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output

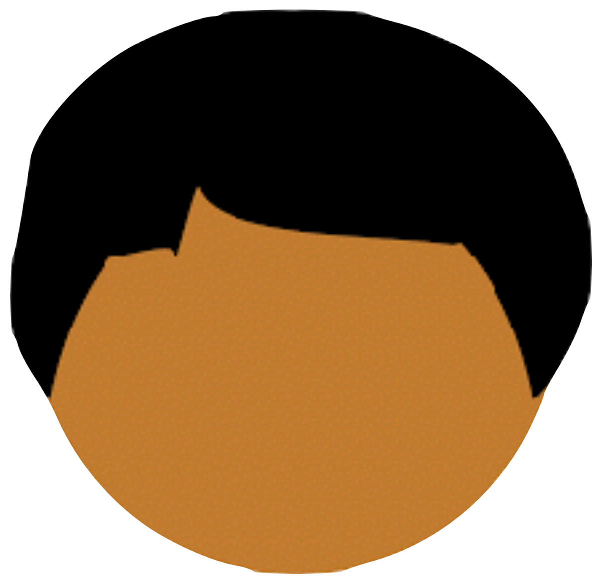
$$(x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$$

$$= (x_0 \wedge y_0) \oplus (x_0 \wedge y_1) \oplus (x_1 \wedge y_0) \oplus (x_1 \wedge y_1)$$

OT



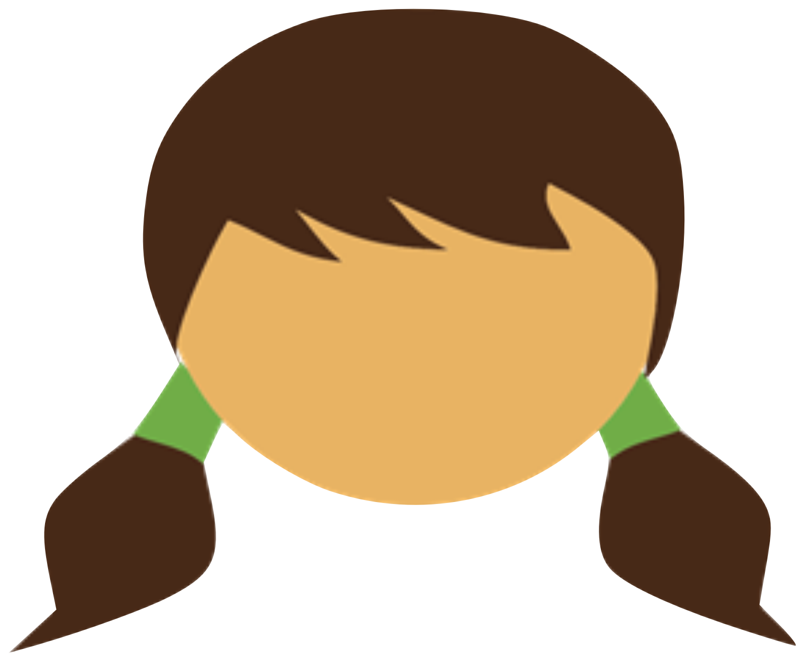




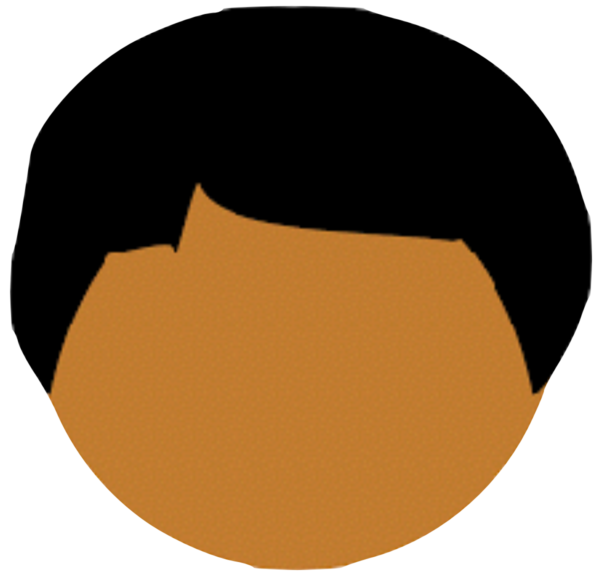
x

Important Subgoal

Goal: given gate input bits x, y, compute random secret share $[x \wedge y]$ s.t. neither party learns $x \wedge y$



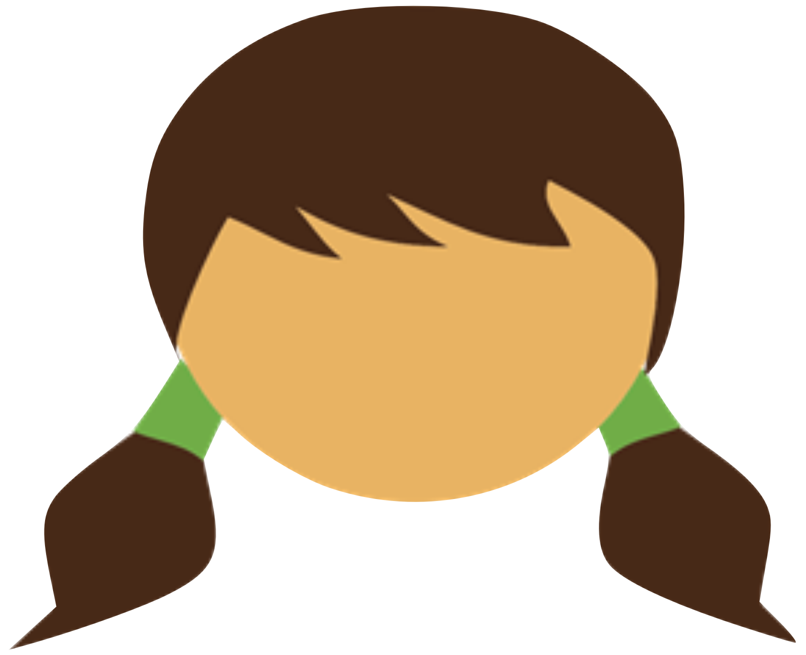
y



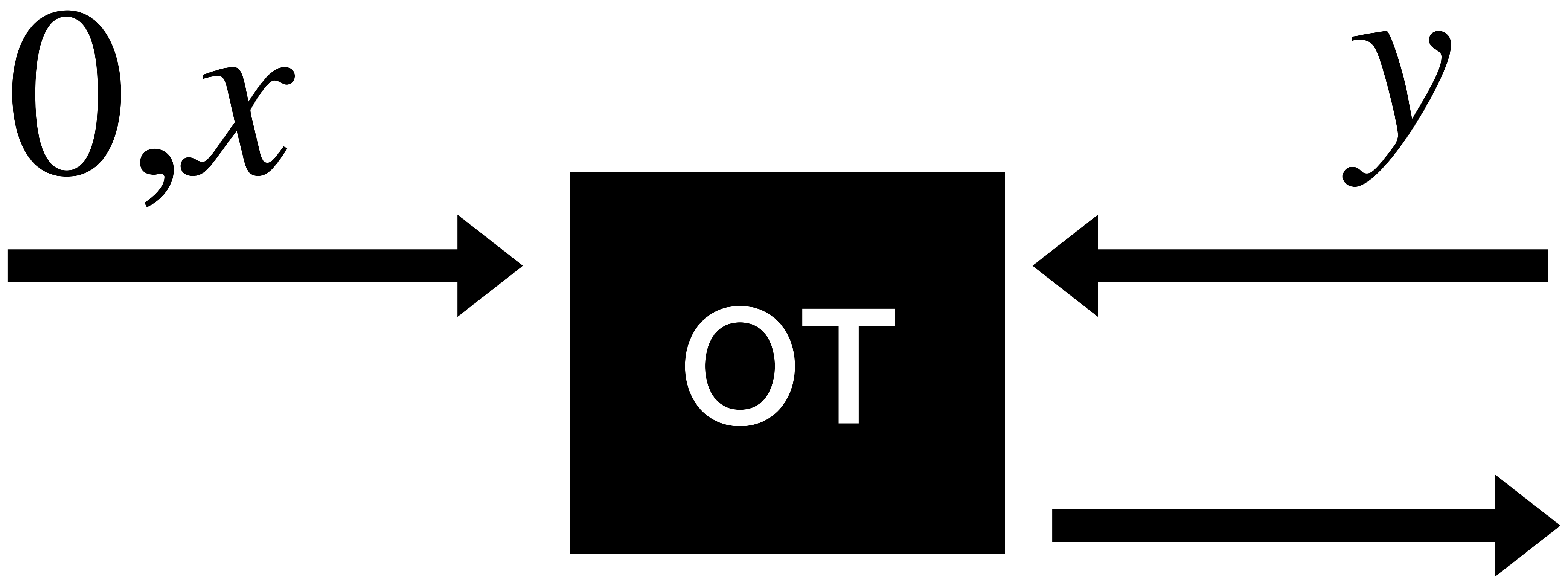
x

Important Subgoal

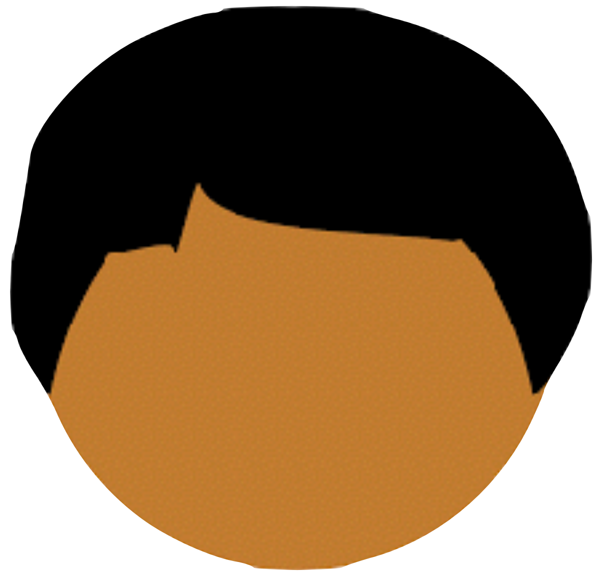
Goal: given gate input bits x, y, compute random secret share $[x \wedge y]$ s.t. neither party learns $x \wedge y$



y



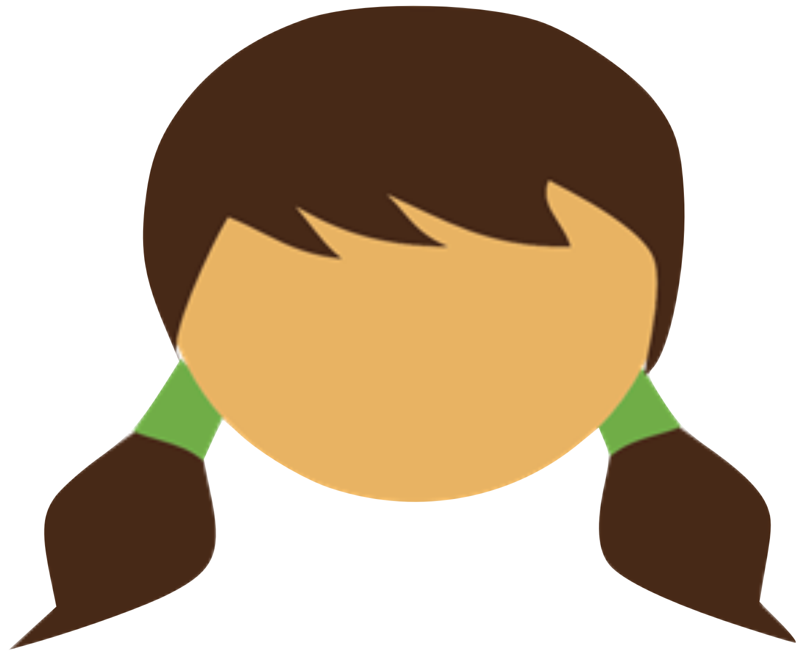
$$\left(\begin{cases} 0 & \text{if } y = 0 \\ x & \text{if } y = 1 \end{cases} \right) = x \wedge y$$



x

Important Subgoal

Goal: given gate input bits x, y, compute random secret share $[x \wedge y]$ s.t. neither party learns $x \wedge y$

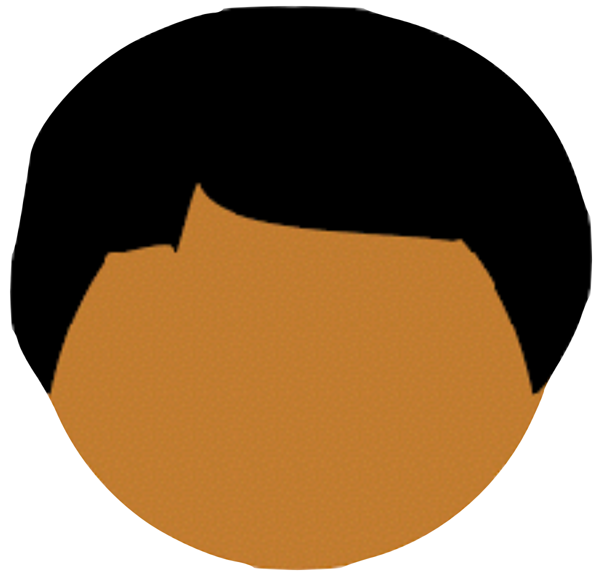


y

$$0, x \longrightarrow \boxed{\text{OT}} \longleftarrow y$$

Good enough?

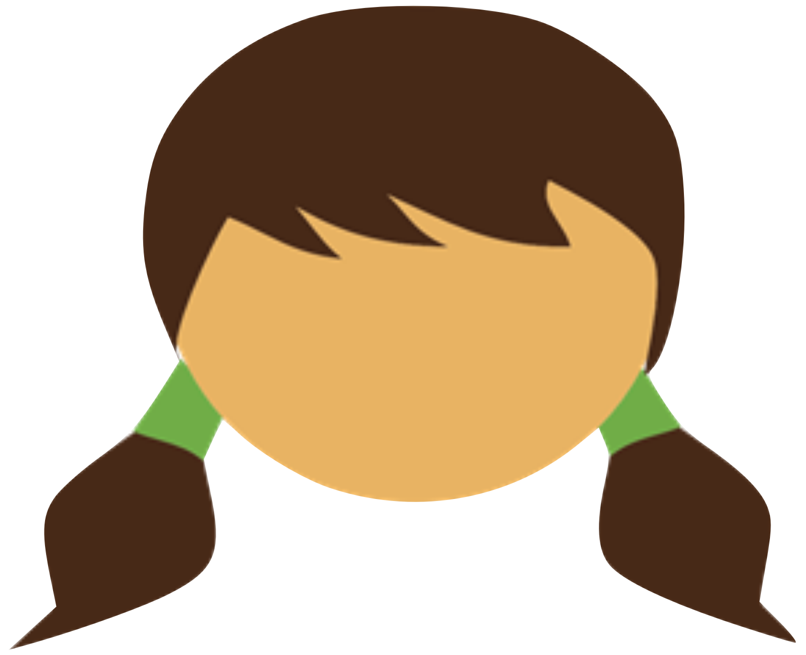
$$\longrightarrow \left(\begin{cases} 0 & \text{if } y = 0 \\ x & \text{if } y = 1 \end{cases} \right) = x \wedge y$$



x

Important Subgoal

Goal: given gate input bits x, y, compute random secret share $[x \wedge y]$ s.t. neither party learns $x \wedge y$



y

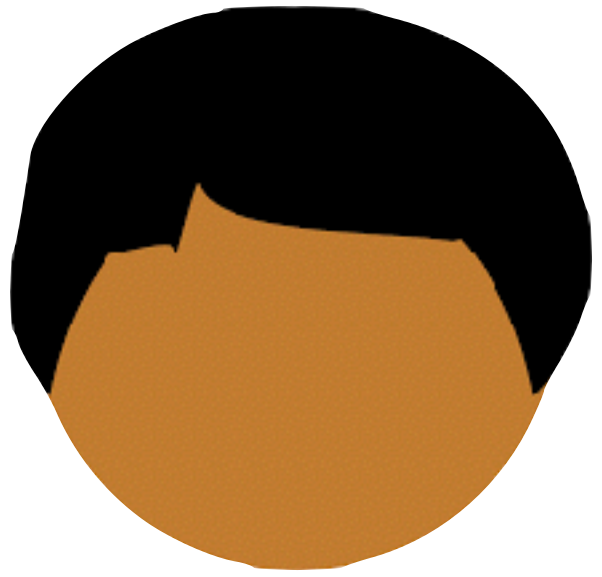
$0, x$ → OT ← y

OT →

$$\left(\begin{cases} 0 & \text{if } y = 0 \\ x & \text{if } y = 1 \end{cases} \right) = x \wedge y$$

Good enough?

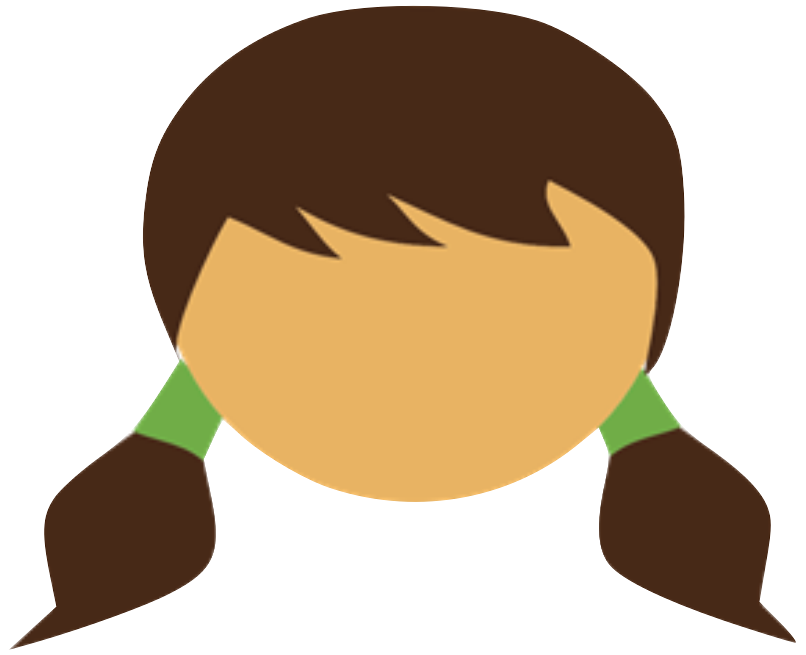
No! Receiver learns information about x



x

Important Subgoal

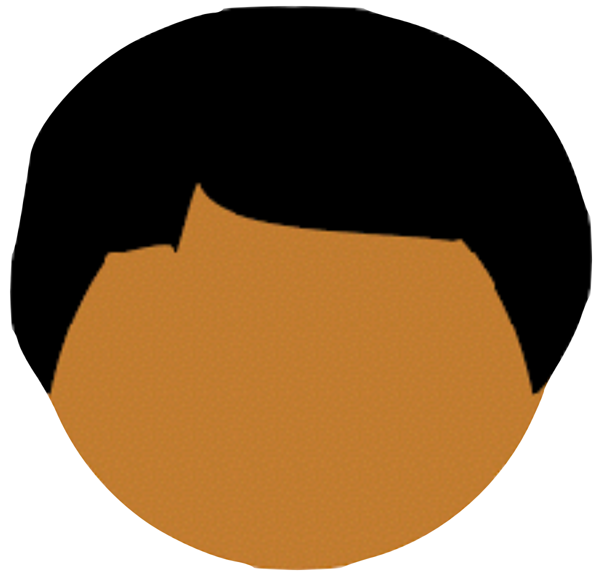
Goal: given gate input bits x, y, compute random secret share $[x \wedge y]$ s.t. neither party learns $x \wedge y$



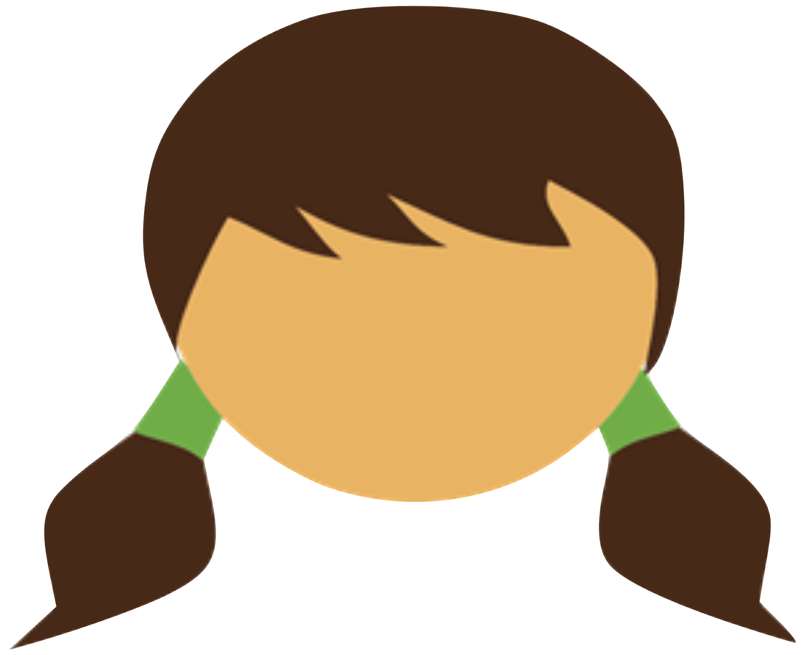
y

$$r \xleftarrow{\$} \{0,1\}$$

Important Subgoal



x



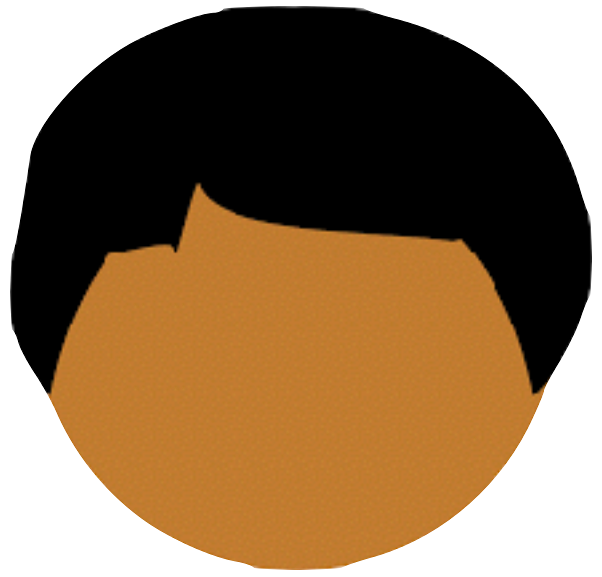
y

Goal: given gate input bits x, y, compute random secret share $[x \wedge y]$ s.t. neither party learns $x \wedge y$

$$r \xleftarrow{\$} \{0,1\}$$

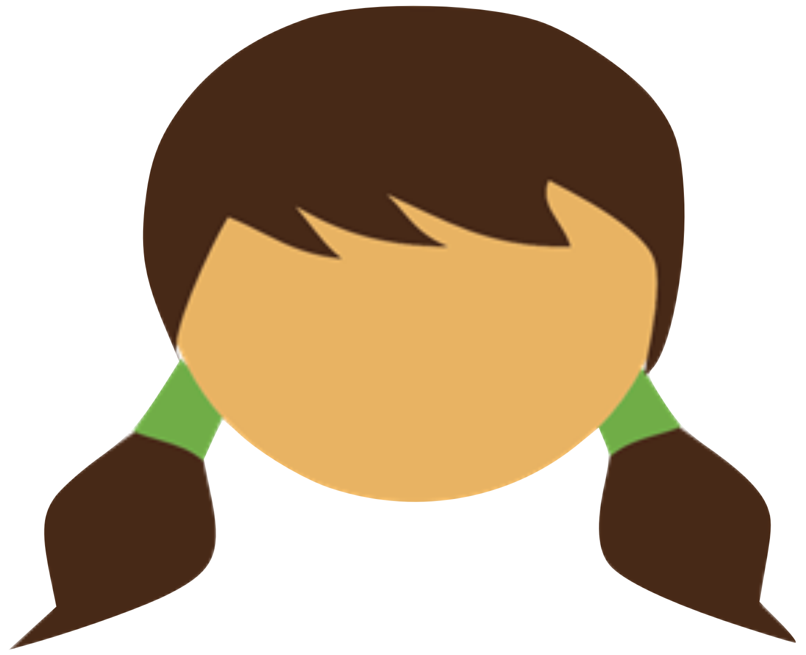
$$r, r \oplus x \qquad\qquad y$$

Important Subgoal



x

Goal: given gate input bits x, y, compute random secret share $[x \wedge y]$ s.t. neither party learns $x \wedge y$



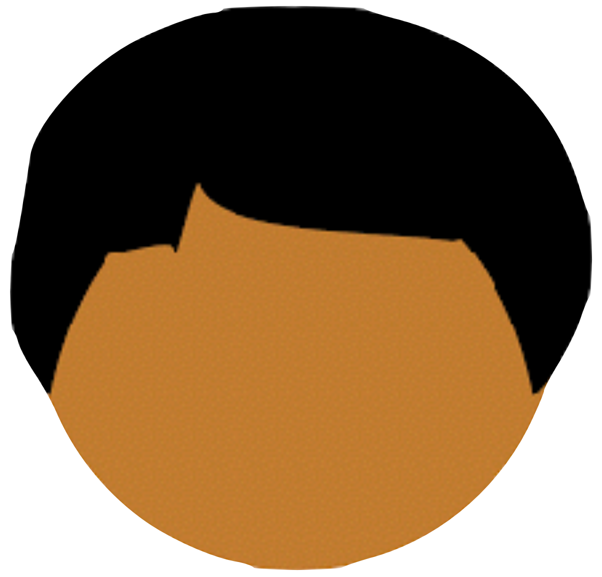
y

$r \xleftarrow{\$} \{0,1\}$

$r, r \oplus x$

y

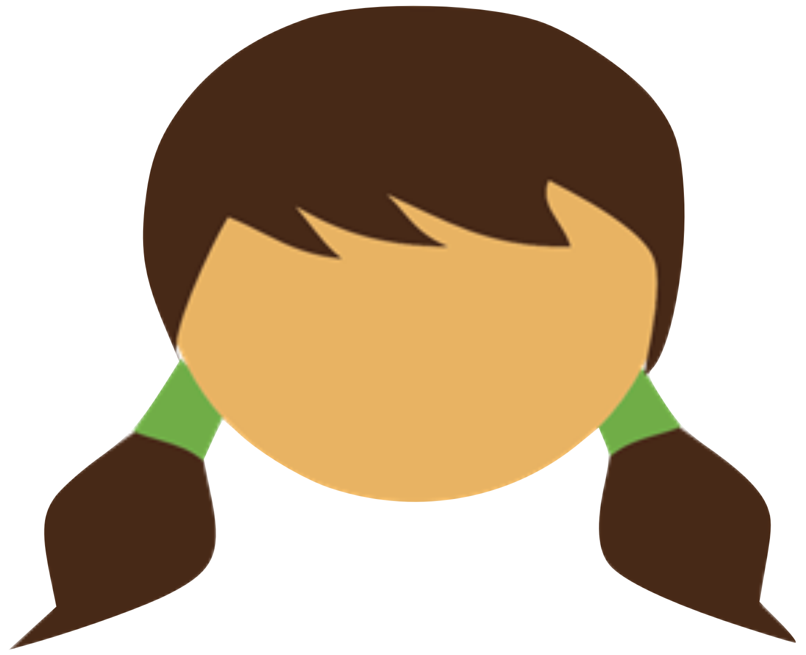
$r \oplus (x \wedge y)$



x

Important Subgoal

Goal: given gate input bits x, y, compute random secret share $[x \wedge y]$ s.t. neither party learns $x \wedge y$

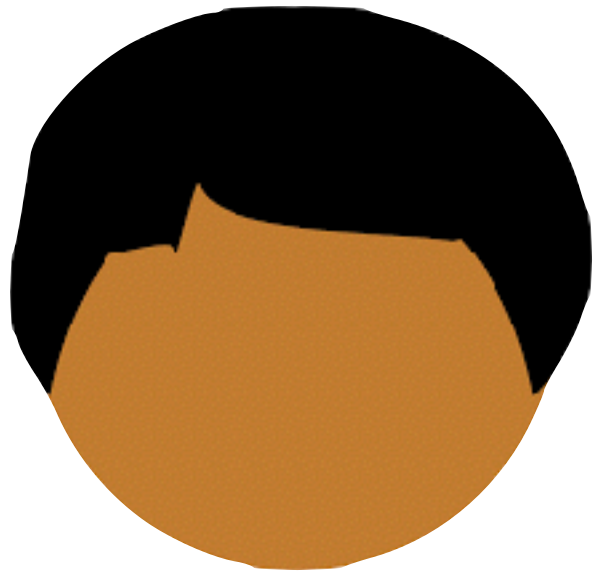


y

$r \xleftarrow{\$} \{0,1\}$

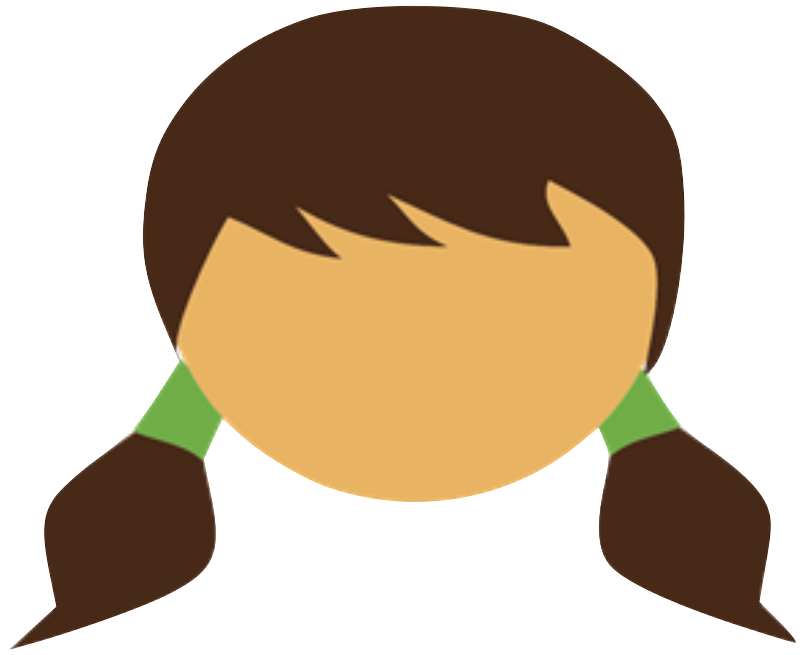
$r, r \oplus x$ $\qquad$ y

$\langle r, r \oplus (x \wedge y) \rangle = [x \wedge y]$ $\qquad$ $r \oplus (x \wedge y)$



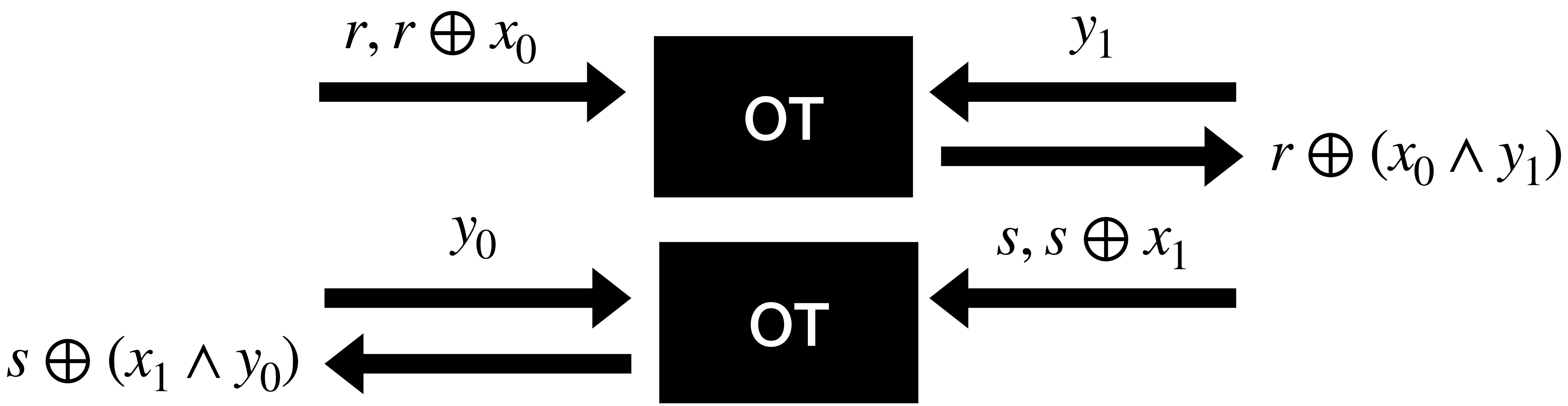
How do we AND two shares?

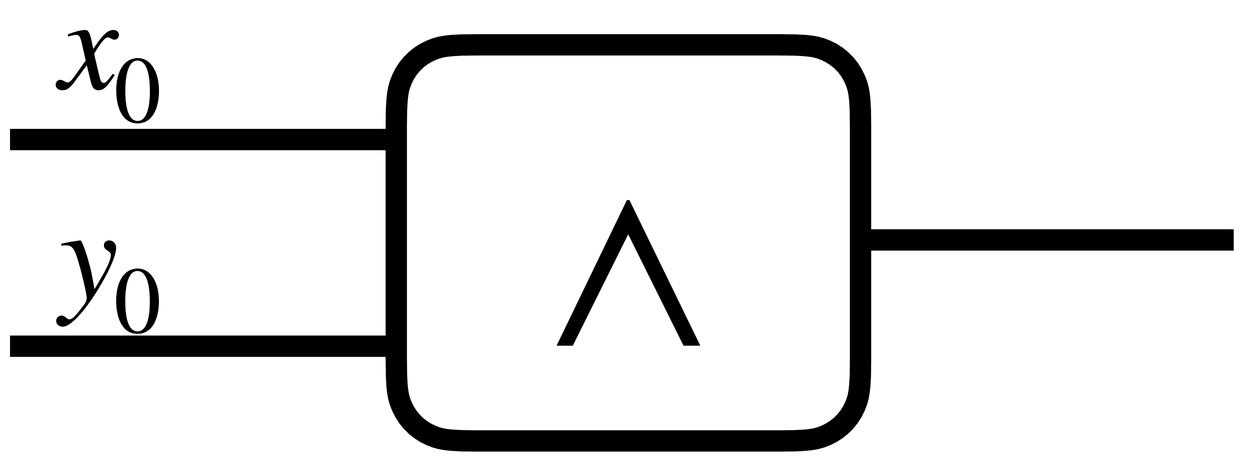
Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output

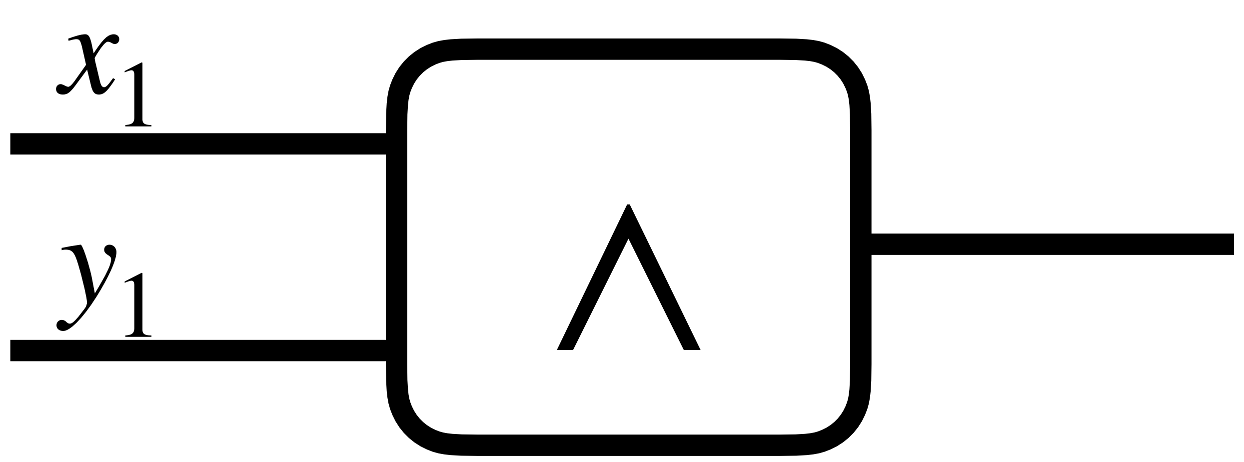


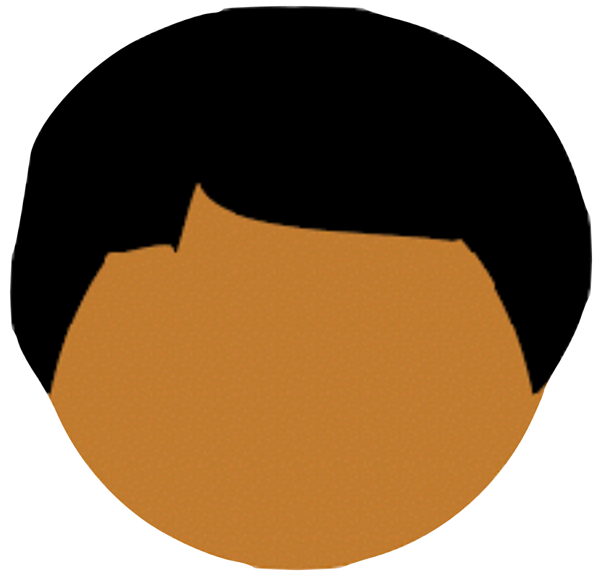
$r \xleftarrow{\$} \{0,1\}$

$s \xleftarrow{\$} \{0,1\}$

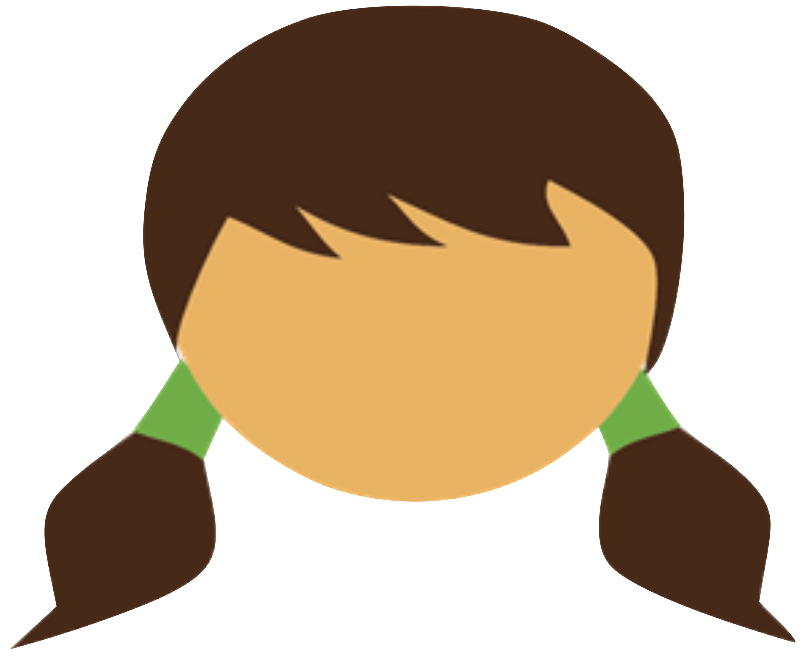








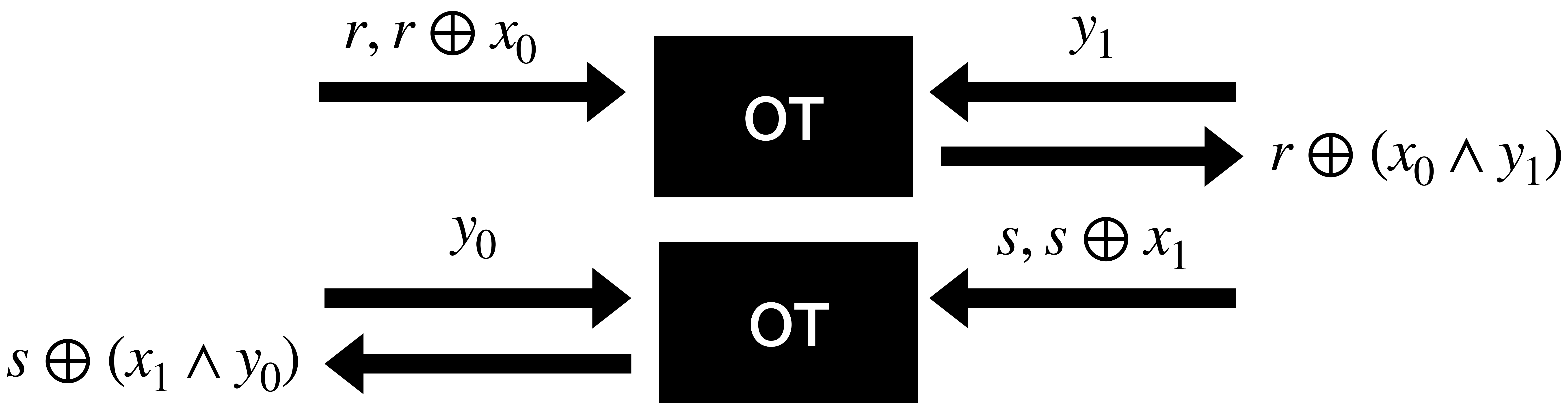
How do we AND two shares?



Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output

$r \xleftarrow{\$} \{0,1\}$

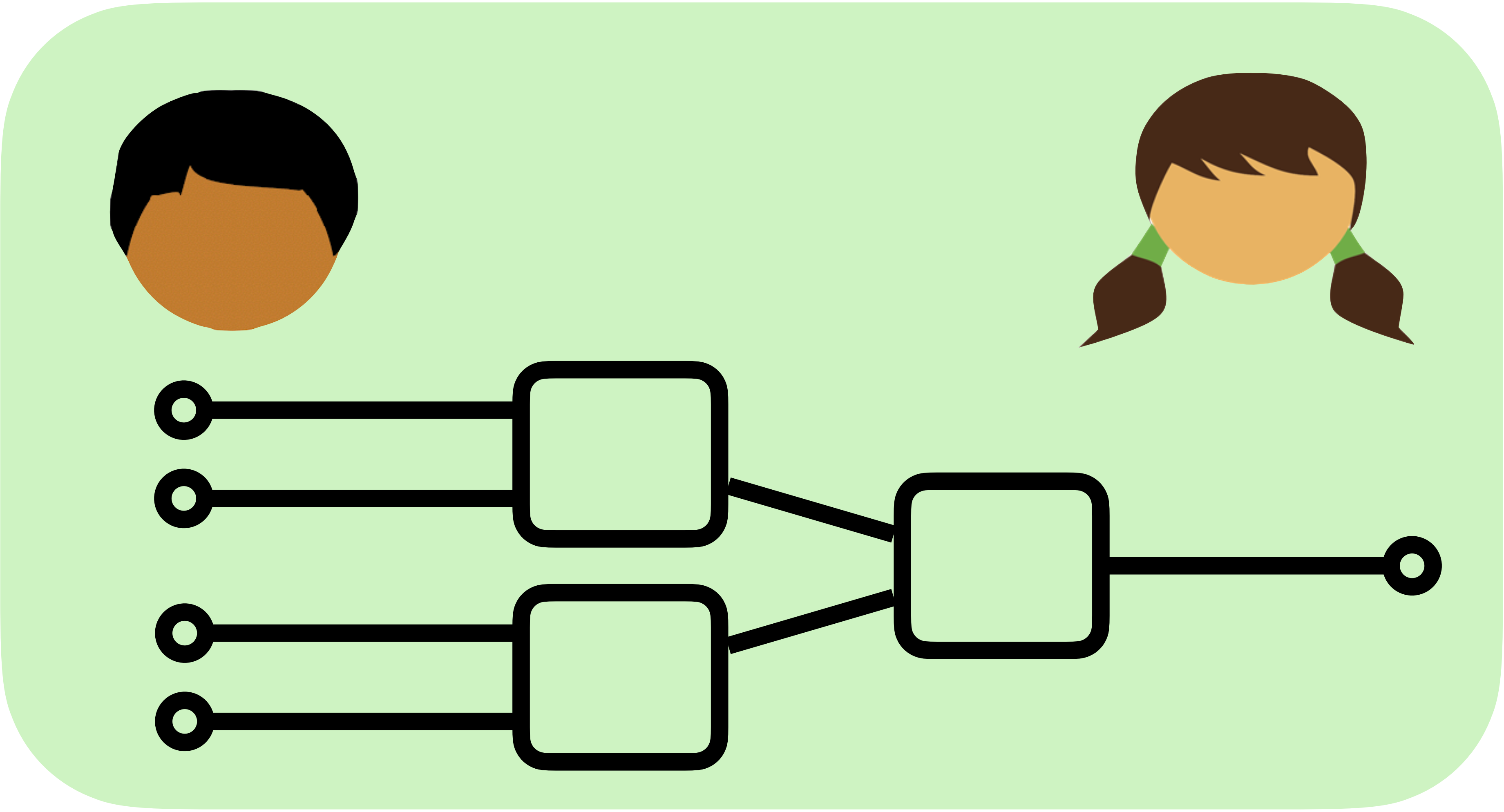
$s \xleftarrow{\$} \{0,1\}$



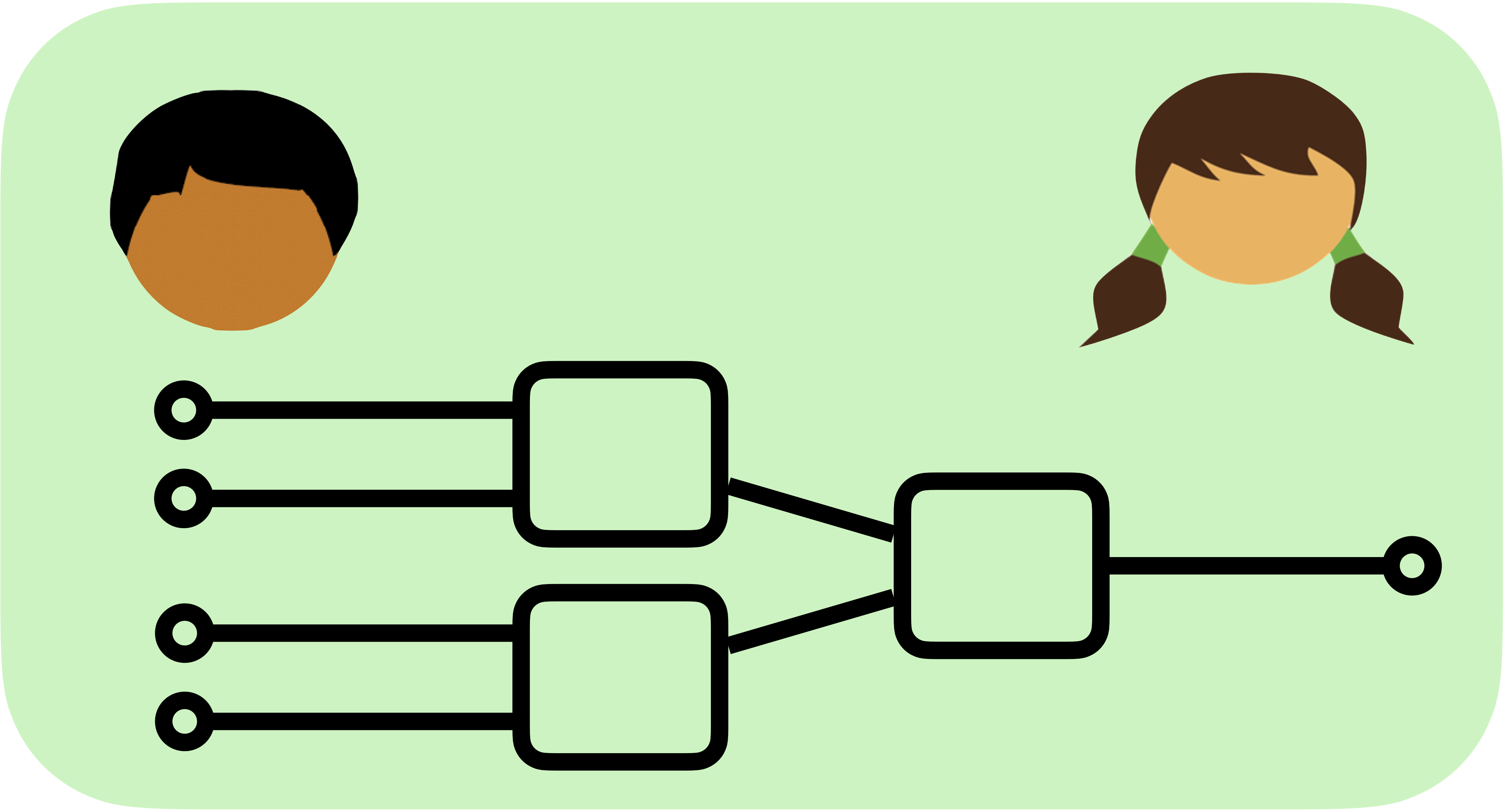
$$\langle r \oplus (s \oplus x_1 \wedge y_0) \oplus (x_0 \wedge y_0), s \oplus (r \oplus x_0 \wedge y_1) \oplus (x_1 \wedge y_1) \rangle$$

$$= [x \wedge y]$$

GMW Protocol

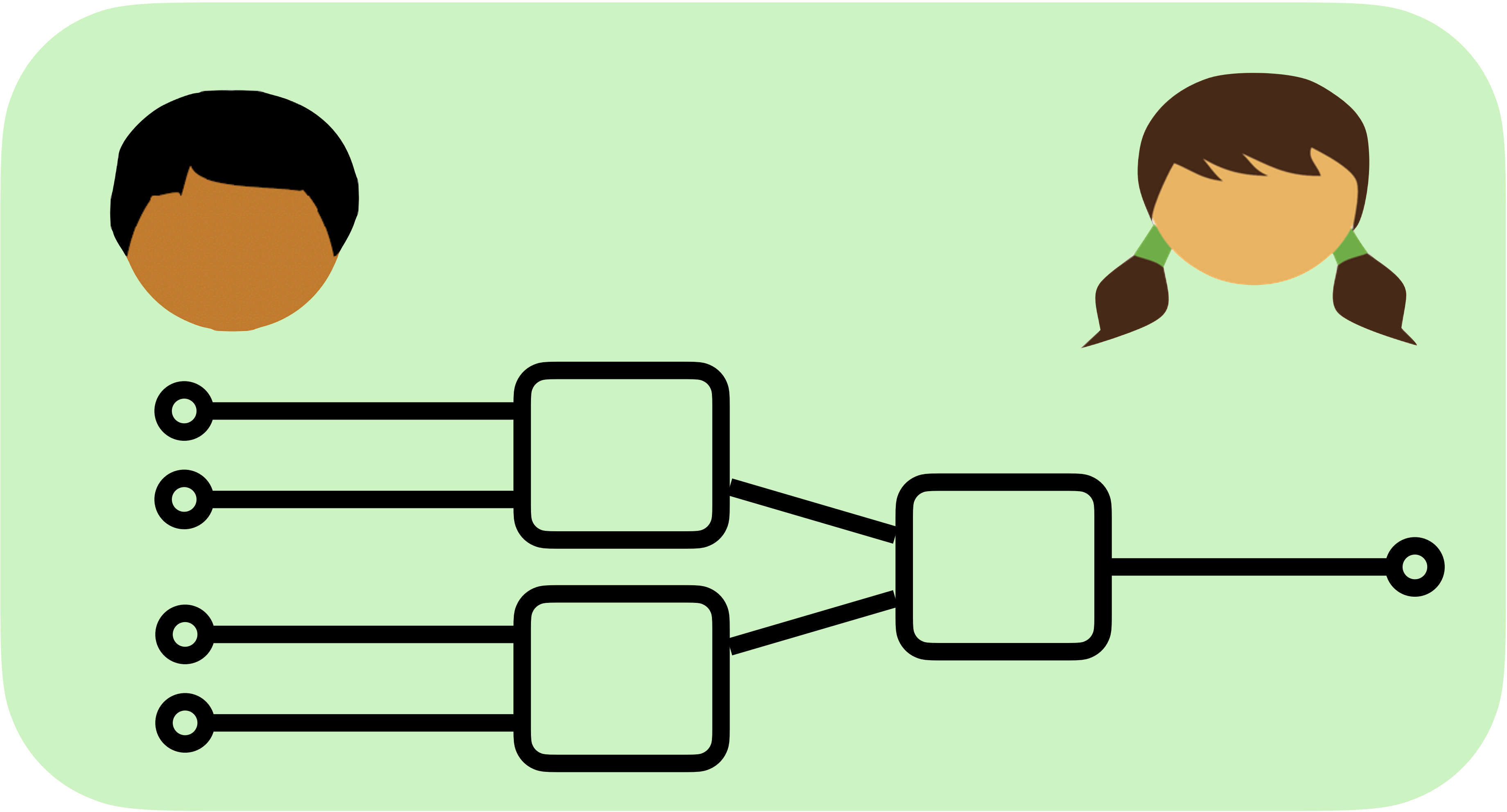


GMW Protocol



Propagate secret shares from input wires to output wires

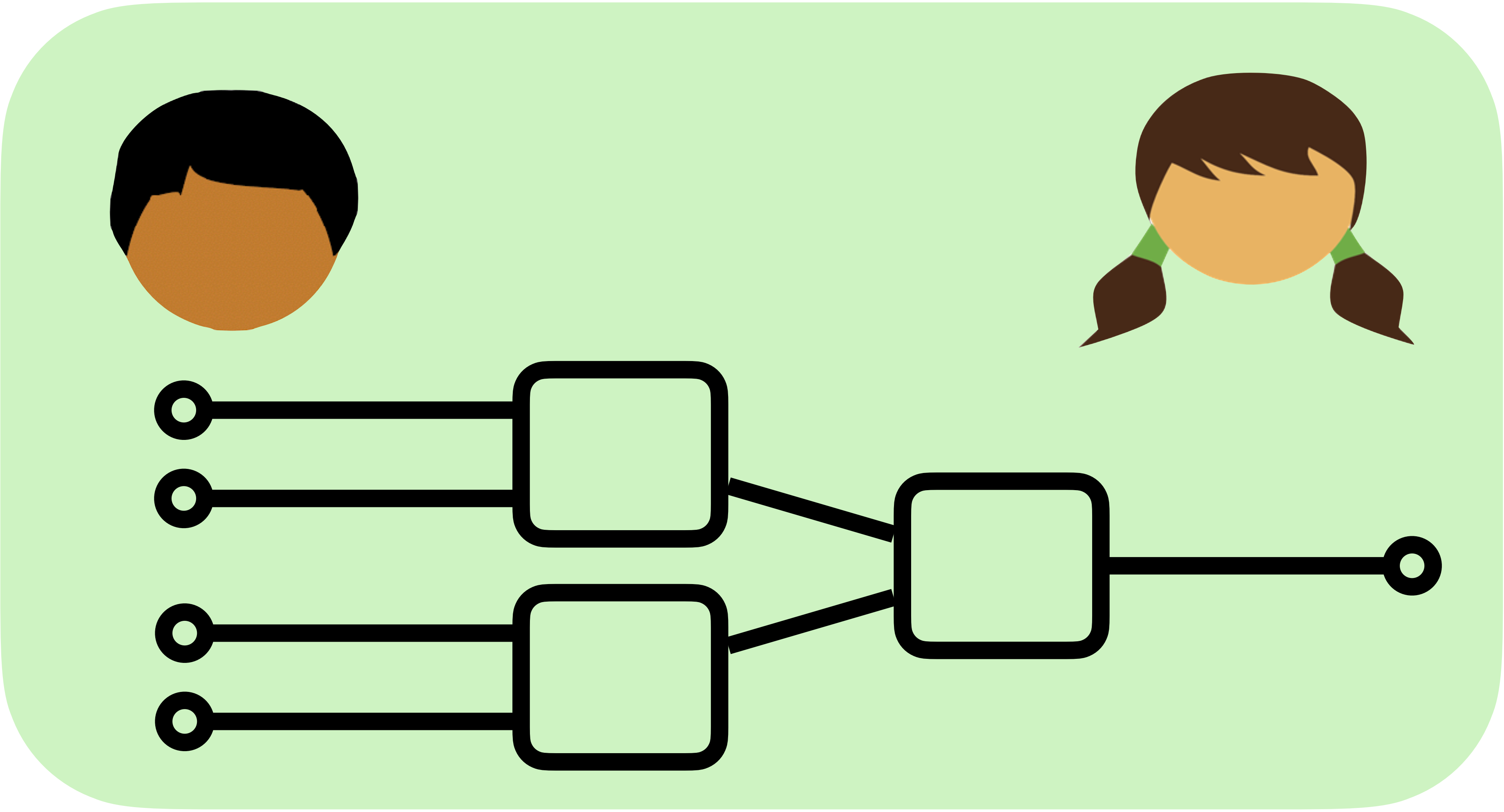
GMW Protocol



Propagate secret shares from input wires to output wires

Use OT to implement AND gates

GMW Protocol

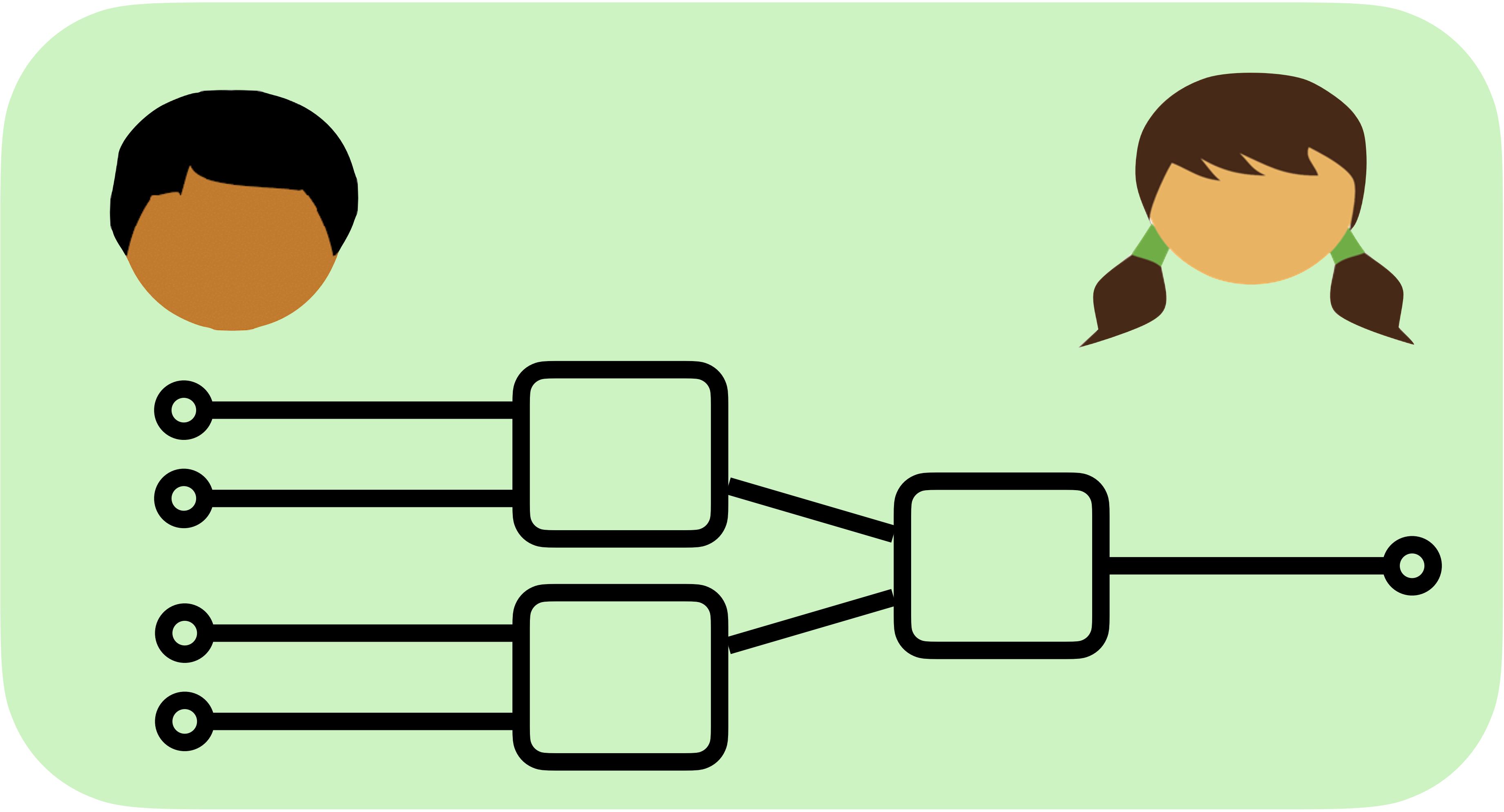


Propagate secret shares from input wires to output wires

Use OT to implement AND gates

Cost:

GMW Protocol



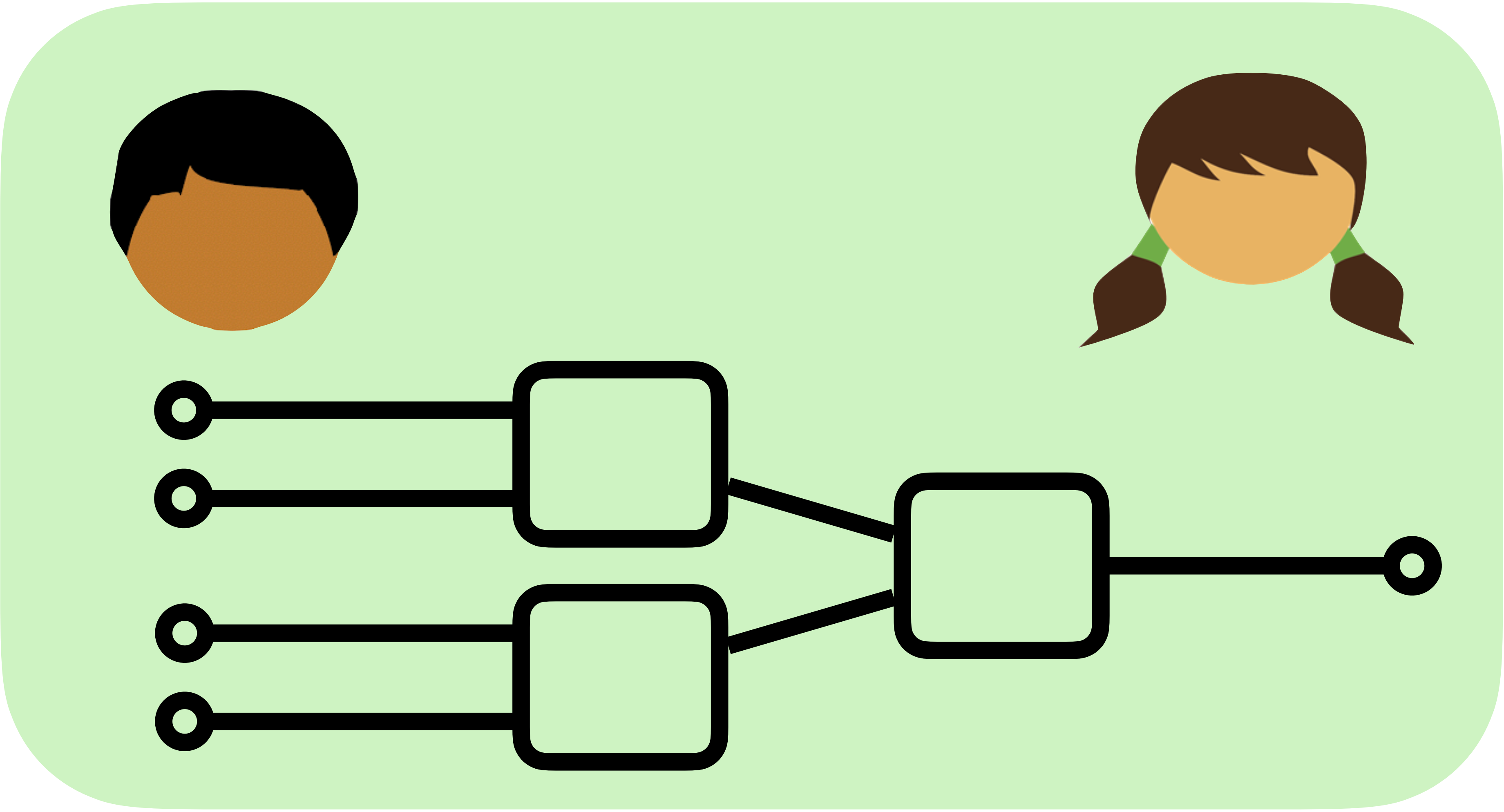
Propagate secret shares from input wires to output wires

Use OT to implement AND gates

Cost:

$O(|C|)$ OTs

GMW Protocol



Propagate secret shares from input wires to output wires

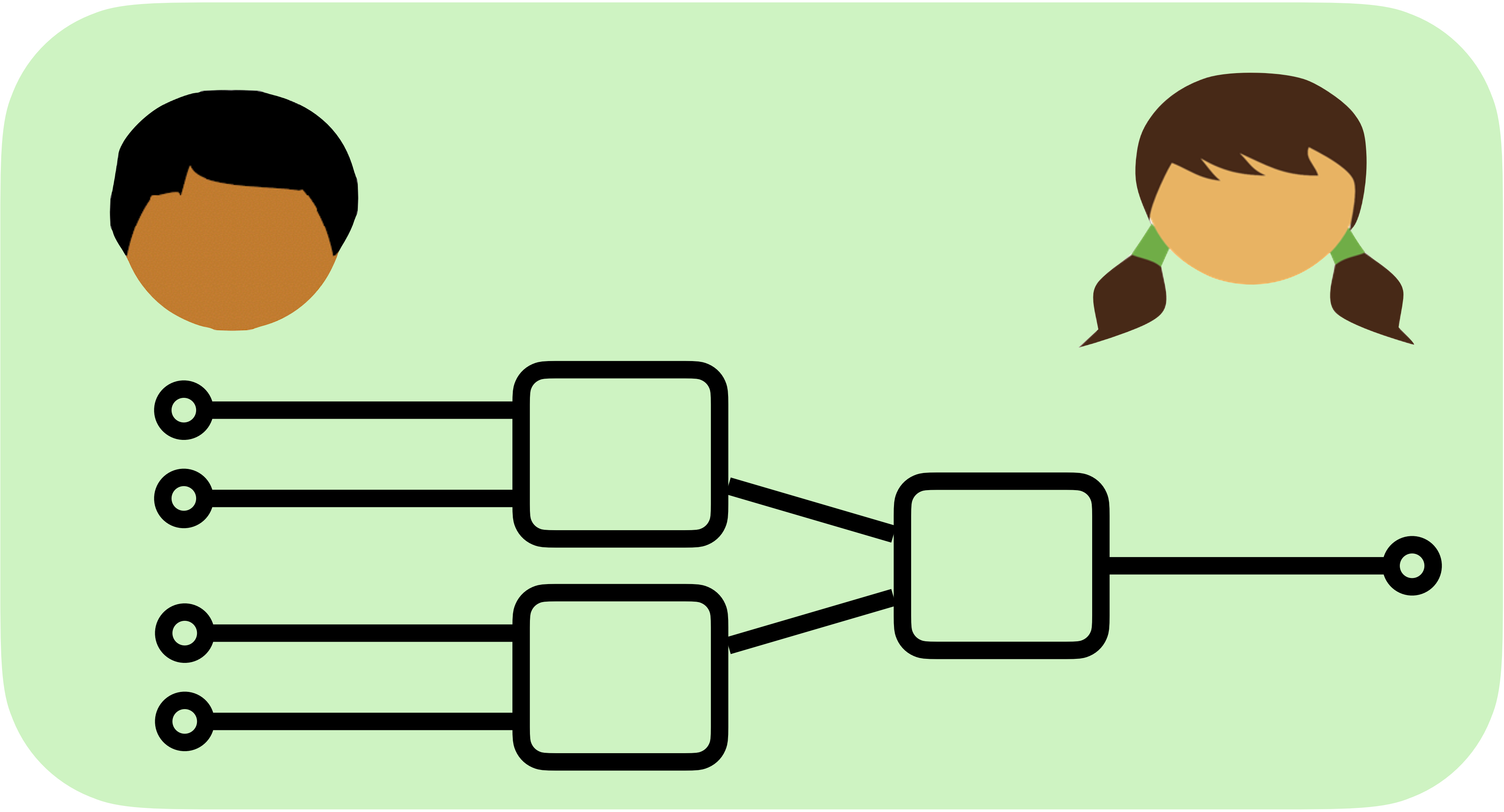
Use OT to implement AND gates

Cost:

$O(|C|)$ OTs

Number of protocol rounds scales with multiplicative depth of C

Where do we go from here?



More Parties

Stronger Security Notions

Decrease Cost
  Fewer rounds, fewer cryptographic operations, etc.

Today's objectives

Review oblivious transfer

Introduce XOR secret sharing

Build our first protocol for securely computing any program (with semi-honest security)