# COMBINATIONAL LOGIC PART II: HALF ADDER, FULL ADDERS, MULTIPLEXERS, FUNCTIONAL COMPLETENESS, & DOMINO COMPUTER

Ayman Hajja, PhD

# HALF ADDER

- Let's try to build a circuit that adds two bits. This circuit is called the half adder.
  - We start by constructing the truth table for out input/output

# HALF ADDER

- Let's try to build a circuit that adds two bits. This circuit is called the half adder.
- We start by constructing the truth table for out input/output

| A | B | Result | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

# HALF ADDER

- Let's try to build a circuit that adds two bits. This circuit is called the half adder.
- We start by constructing the truth table for out input/output

| A | B | Result | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

What gate(s) does the carry represent?

# HALF ADDER

- Let's try to build a circuit that adds two bits. This circuit is called the half adder.
- We start by constructing the truth table for out input/output

| A | B | Result | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

What gate(s) does the carry represent?

AND gate

# HALF ADDER

- Let's try to build a circuit that adds two bits. This circuit is called the half adder.
- We start by constructing the truth table for out input/output

| A | B | Result | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

What gate(s) does the Result represent?

# HALF ADDER

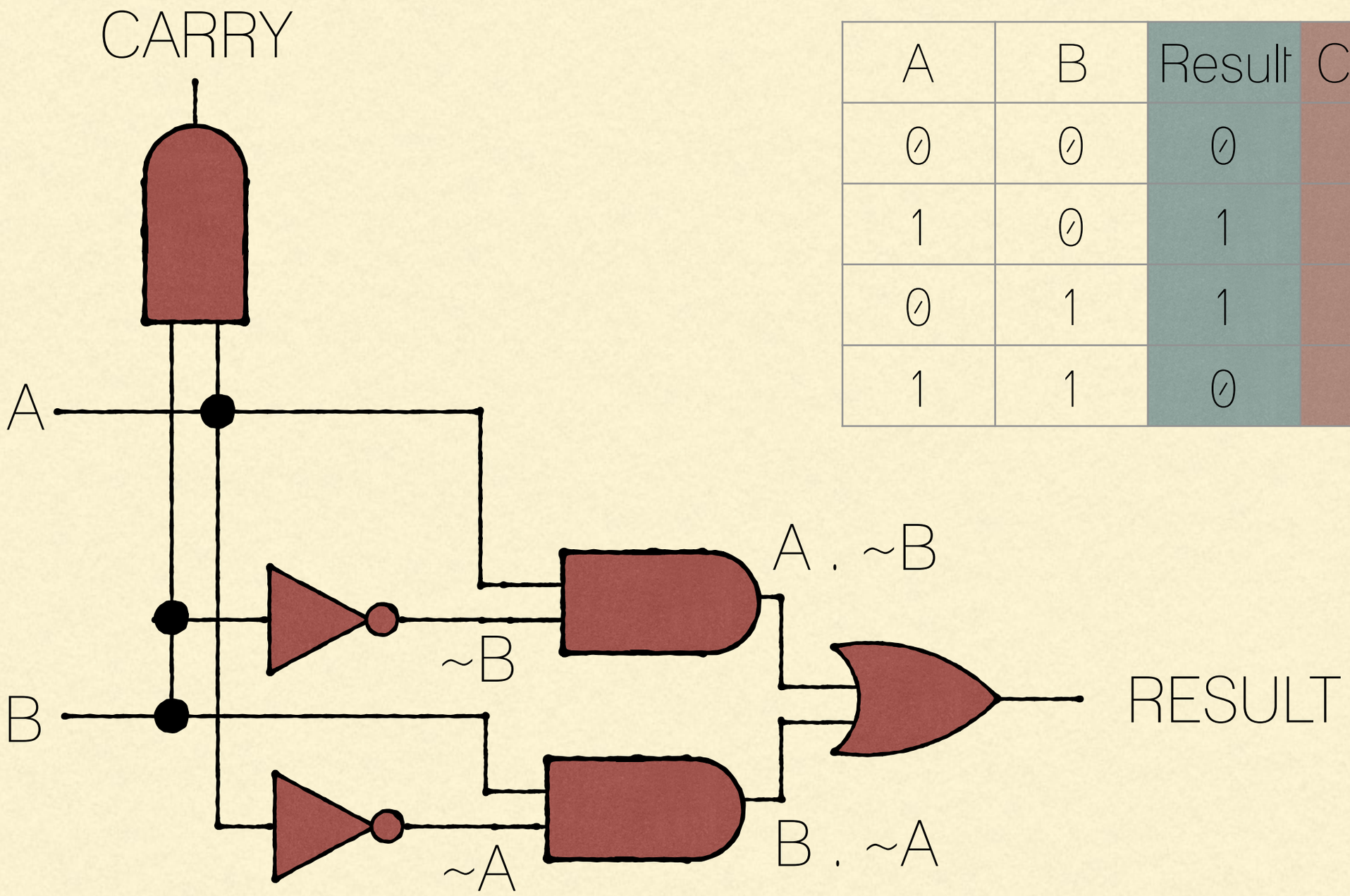
- Let's try to build a circuit that adds two bits. This circuit is called the half adder.
- We start by constructing the truth table for out input/output

| A | B | Result | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

What gate(s) does the Result represent?

EXCLUSIVE OR gate

# BUILDING A HALF ADDER




| A | B | Result | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

# HALF ADDER

Can we build 32 half adders to add a 32-bit sequence?

# HALF ADDER

Can we build 32 half adders to add a 32-bit sequence?

- No. The problem with a half-adder is that there it doesn't handle carries. We need a circuit that can add three bits.
  - That circuit is called a full adder.

# FULL ADDER

- Here are the characteristics of a full adder.
  - Data inputs; 3 (let's call them X, Y, and CARRY IN)
  - Outputs; 2 (let's call them SUM, and CARRY OUT)
- How many rows would our truth table for the full adder be?

# FULL ADDER

- Here are the characteristics of a full adder.
  - Data inputs; 3 (let's call them X, Y, and CARRY IN)
  - Outputs; 2 (let's call them SUM, and CARRY OUT)
- How many rows would our truth table for the full adder be?

THE ANSWER IS 8 ($2^3$)

# FULL ADDER

- Here are the characteristics of a full adder.
  - Data inputs; 3 (let's call them X, Y, and CARRY IN)
  - Outputs; 2 (let's call them SUM, and CARRY OUT)

| X | Y | CARRY IN | SUM | CARRY OUT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

# FULL ADDER

- Sum of products for the ‘CARRY OUT’;

$$X.Y.\sim C_{IN} + X.\sim Y.C_{IN} + \sim X.Y.C_{IN} + X.Y.C_{IN}$$

| X | Y | CARRY IN | SUM | CARRY OUT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

# FULL ADDER

- Sum of products for the ‘CARRY OUT’;

$$X.Y.\sim C_{IN} + X.\sim Y.C_{IN} + \sim X.Y.C_{IN} + X.Y.C_{IN}$$

Can be further simplified to; $X.Y + Y.C_{IN} + X.C_{IN}$

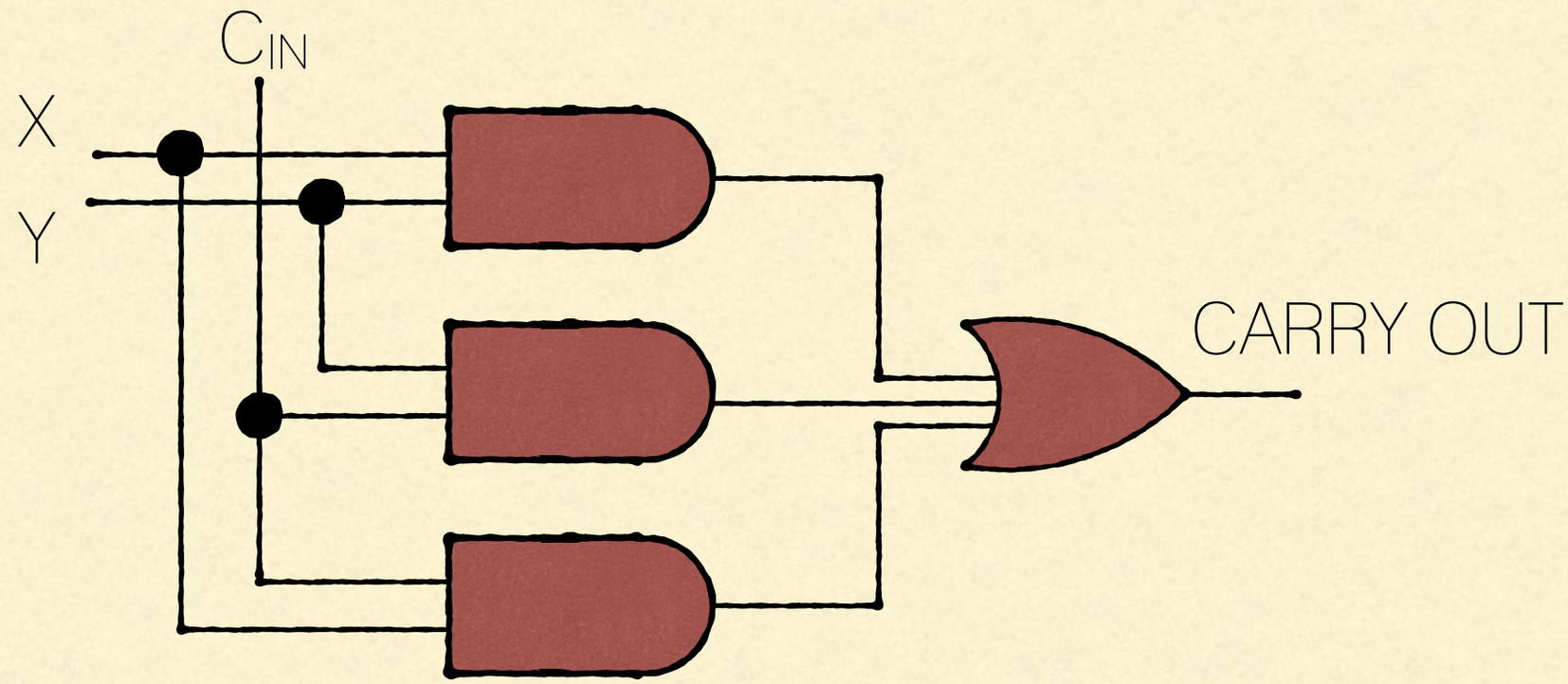
| X | Y | CARRY IN | SUM | CARRY OUT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

# FULL ADDER

- Sum of products for the ‘CARRY OUT’;

$$X.Y.\sim C_{IN} + X.\sim Y.C_{IN} + \sim X.Y.C_{IN} + X.Y.C_{IN}$$

Can be further simplified to; $X.Y + Y.C_{IN} + X.C_{IN}$



# FULL ADDER

- Sum of products for the ‘SUM’;

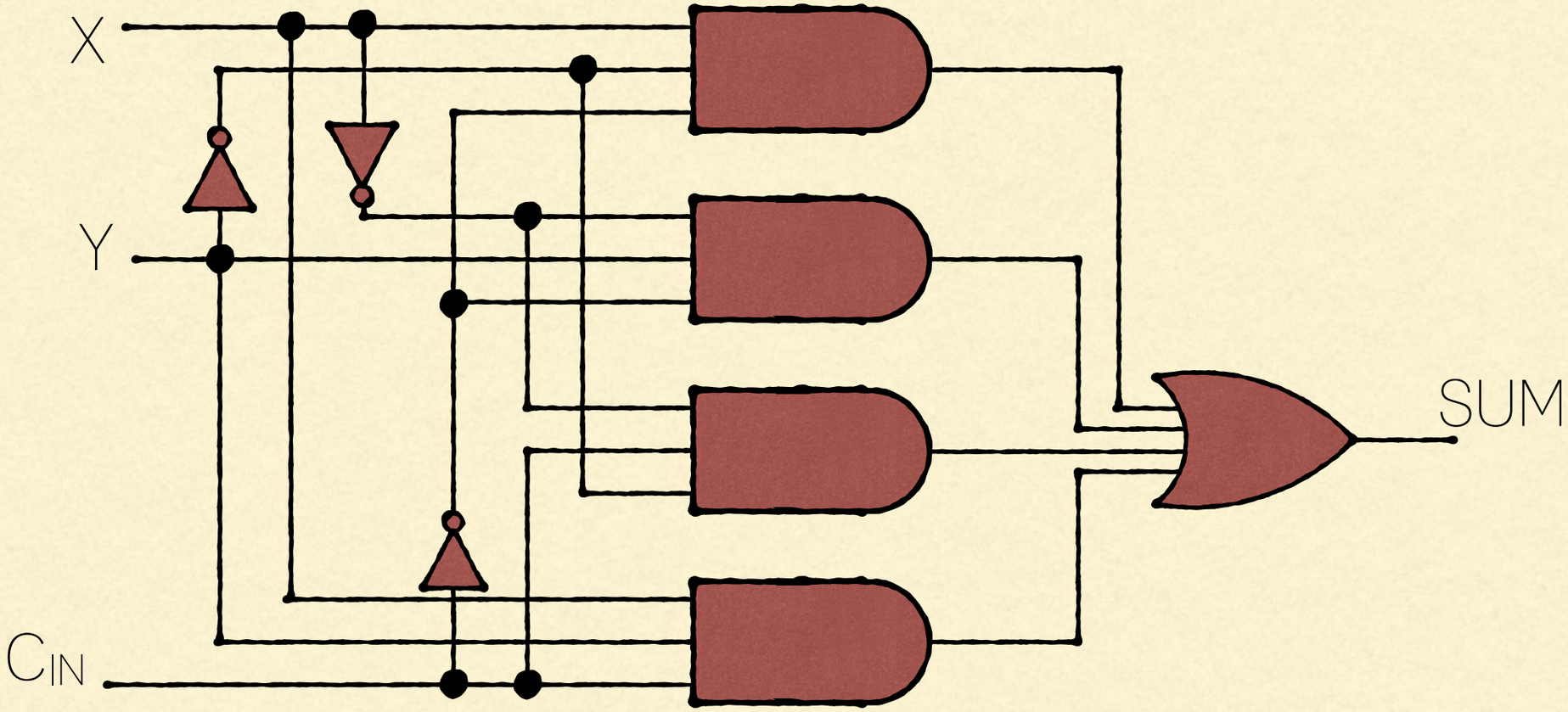
$$X.\sim Y.\sim C_{IN} + \sim X.Y.\sim C_{IN} + \sim X.\sim Y.C_{IN} + X.Y.C_{IN}$$

| X | Y | CARRY IN | SUM | CARRY OUT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

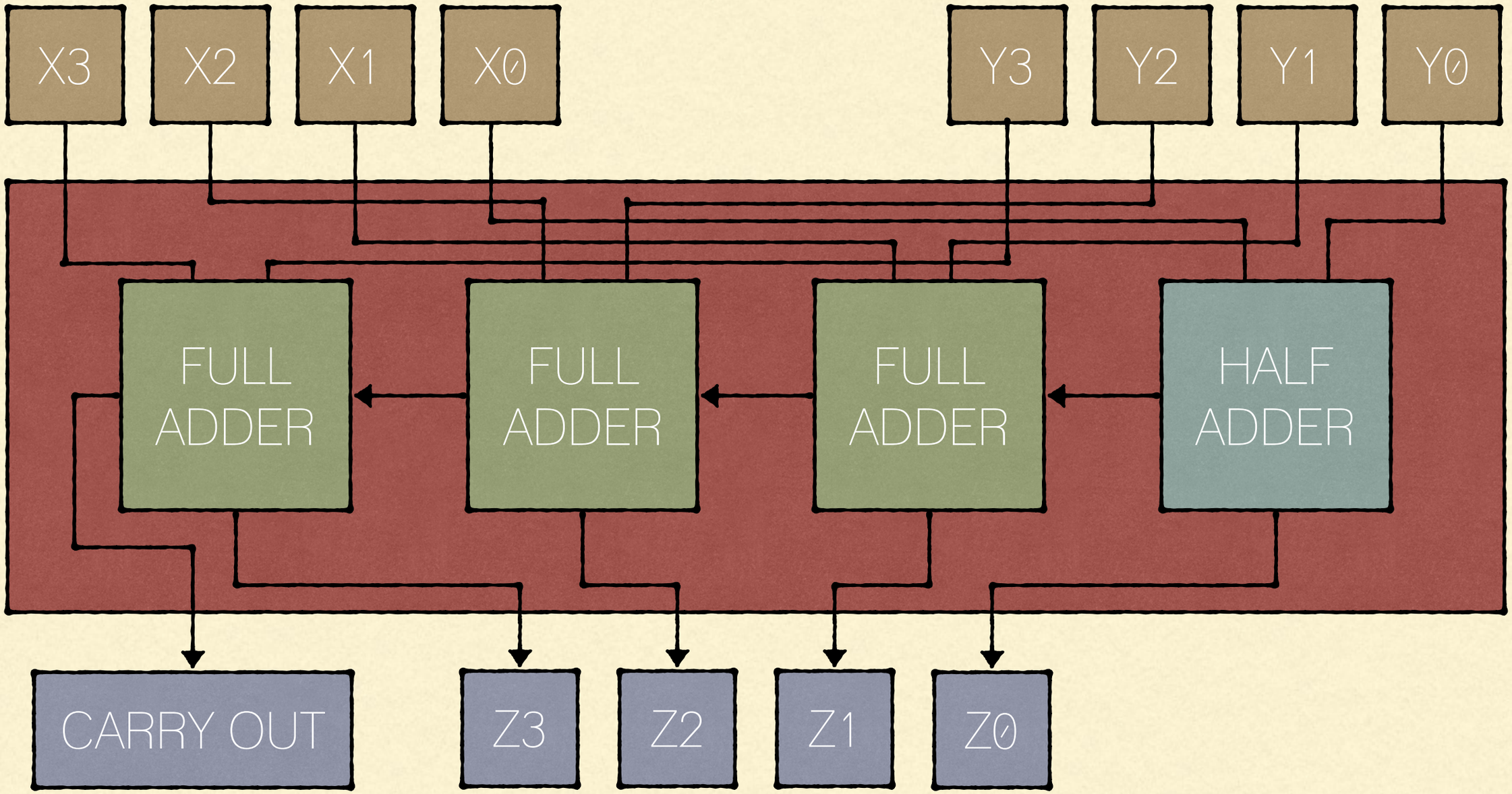
# FULL ADDER; CIRCUIT FIXED

- Sum of products for the ‘SUM’;

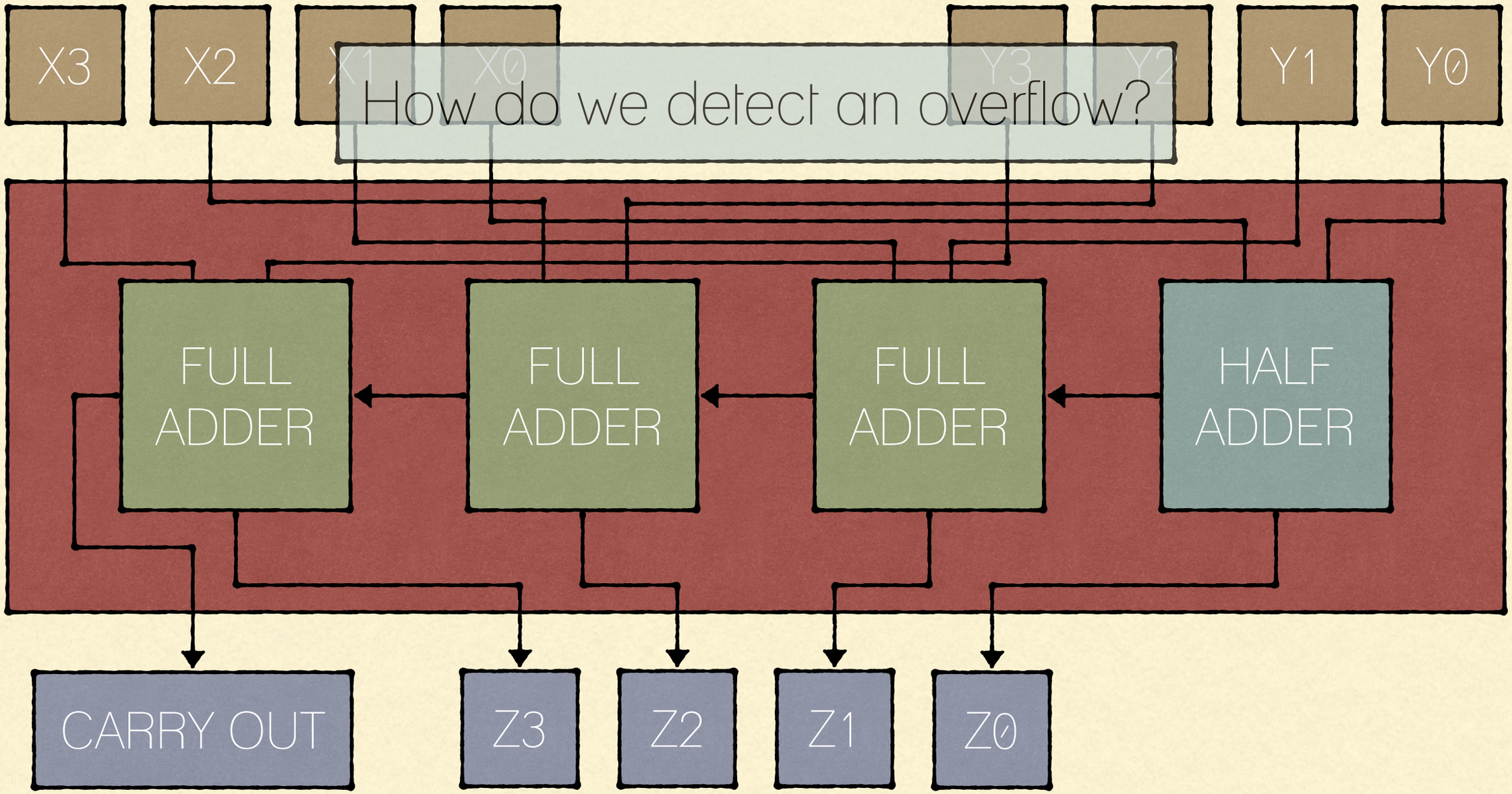
$$X.\sim Y.\sim C_{IN} + \sim X.Y.\sim C_{IN} + \sim X.\sim Y.C_{IN} + X.Y.C_{IN}$$



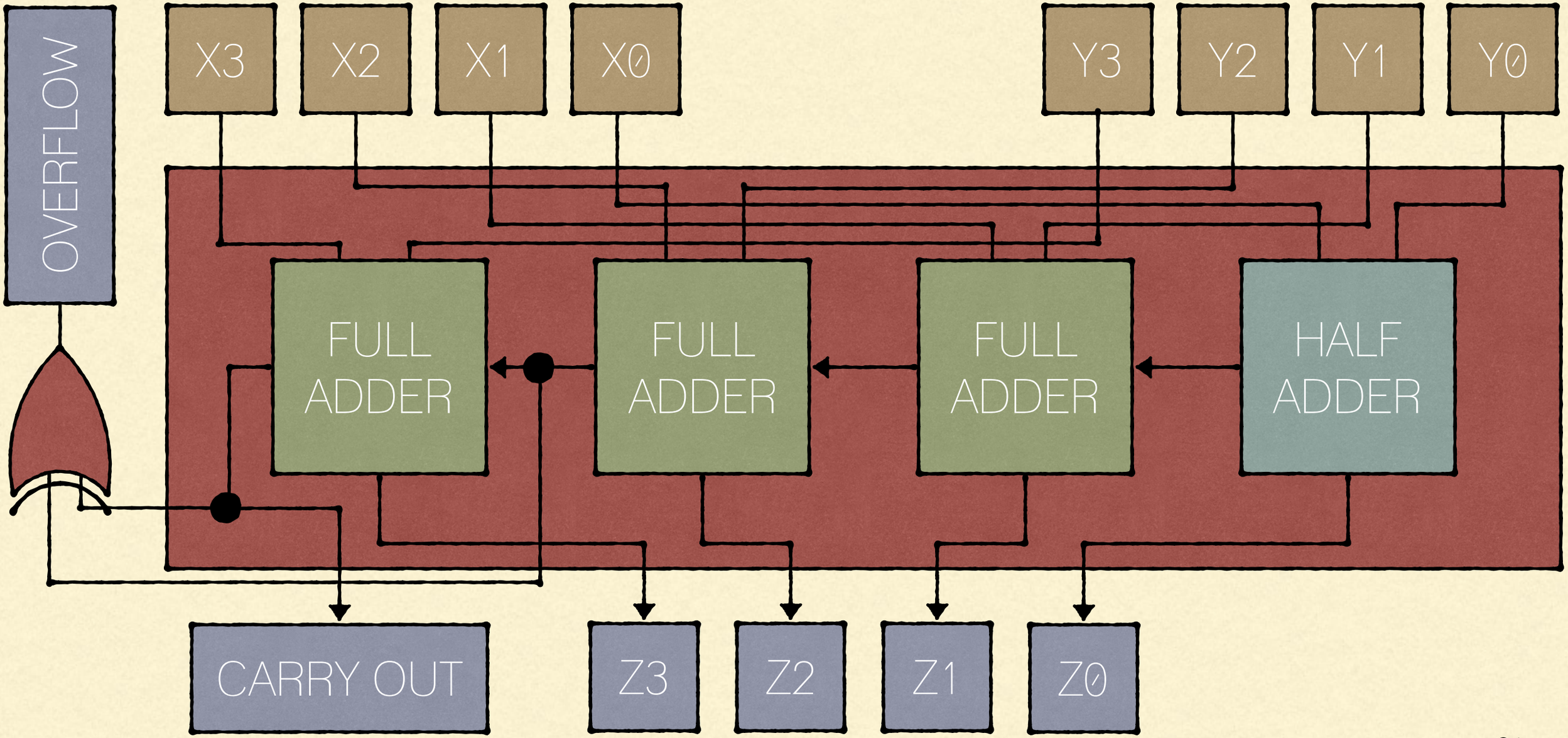
# ADDING A 4-BIT SEQUENCE



# ADDING A 4-BIT SEQUENCE



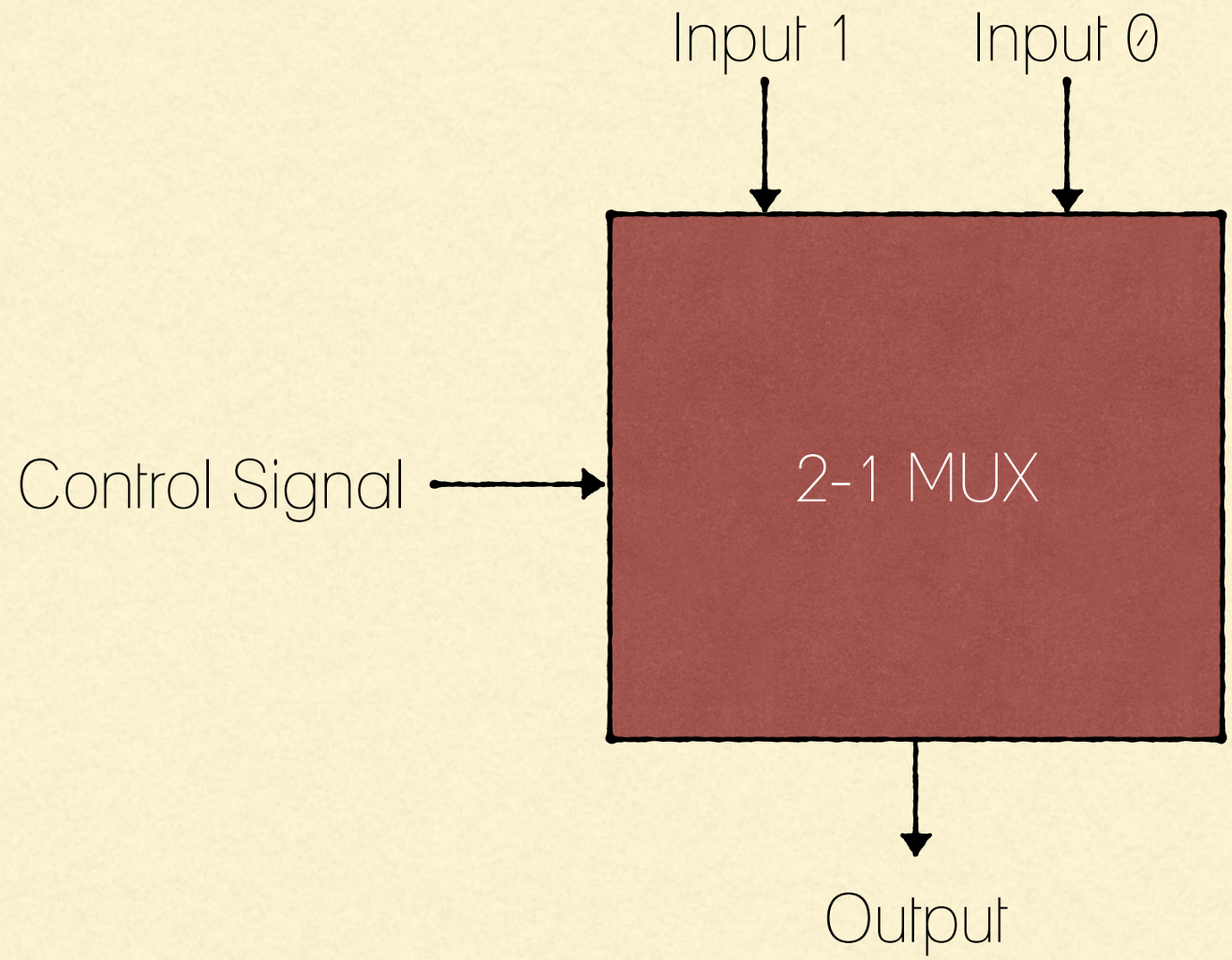
# ADDING A 4-BIT SEQUENCE



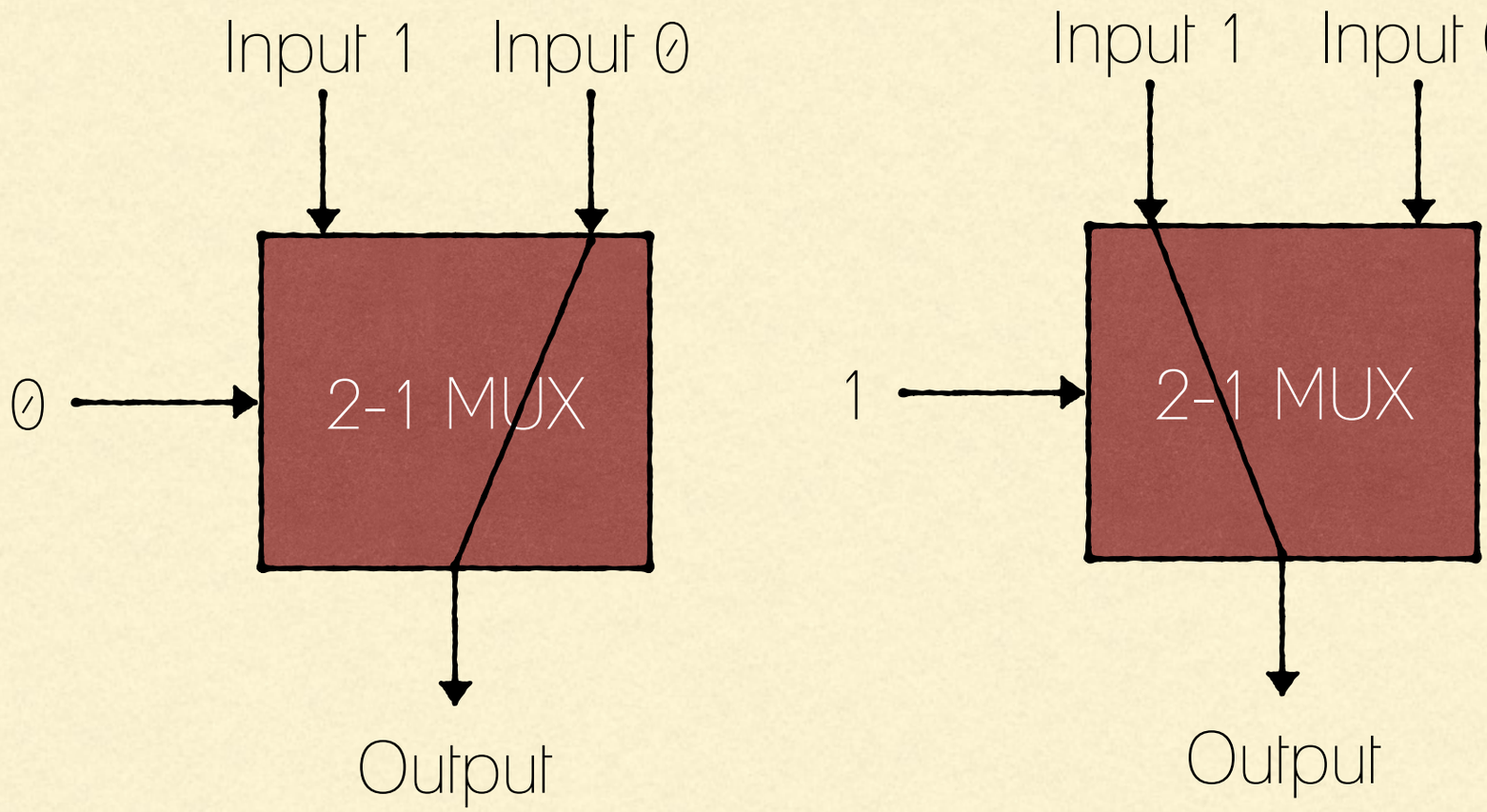
# MULTIPLEXERS

- As we construct a CPU, one very useful combinational logic circuit is a multiplexer.
- A n-1 multiplexer, or MUX, for short, is a device that allows you to pick one of n inputs and direct it to an output.
- An n-1 MUX consists of the following:
  - Data inputs; n
  - Control inputs; ceil($\log_2$ n) — Number of bits needed to select any particular input
  - Outputs; 1
- For example, if you have 16 possible inputs, you need 4 bits to specify one of 16 values. If you have, say, 12 possible inputs, you still need 4 bits, even though some of the 4 bit patterns may not correspond to any of the 12 choices.

# DIAGRAM OF 2-1 MUX



# BEHAVIOR OF 2-1 MUX



# TRUTH TABLE OF 2-1 MUX

Before generating the boolean expressions for output, let's introduce “DON'T CARES”

| INPUT 0 | INPUT 1 | CONTROL | OUTPUT |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

# TRUTH TABLE OF 2-1 MUX

Input “DON'T CARE” arise when an output depends on only some of the inputs, and they are shown as Xs, in the input portion of the truth table

| INPUT 0 | INPUT 1 | CONTROL | OUTPUT |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

When CONTROL = 0, we only care about INPUT 0. We DON'T CARE about INPUT 1

When CONTROL = 1, we only care about INPUT 1. We DON'T CARE about INPUT 0

# TRUTH TABLE OF 2-1 MUX

Input “DON'T CARE” arise when an output depends on only some of the inputs, and they are shown as Xs, in the input portion of the truth table

| INPUT 0 | INPUT 1 | CONTROL | OUTPUT |
|---|---|---|---|
| 0 | X | 0 | 0 |
| X | 0 | 1 | 0 |
| 0 | X | 0 | 0 |
| X | 1 | 1 | 1 |
| 1 | X | 0 | 1 |
| X | 0 | 1 | 0 |
| 1 | X | 0 | 1 |
| X | 1 | 1 | 1 |

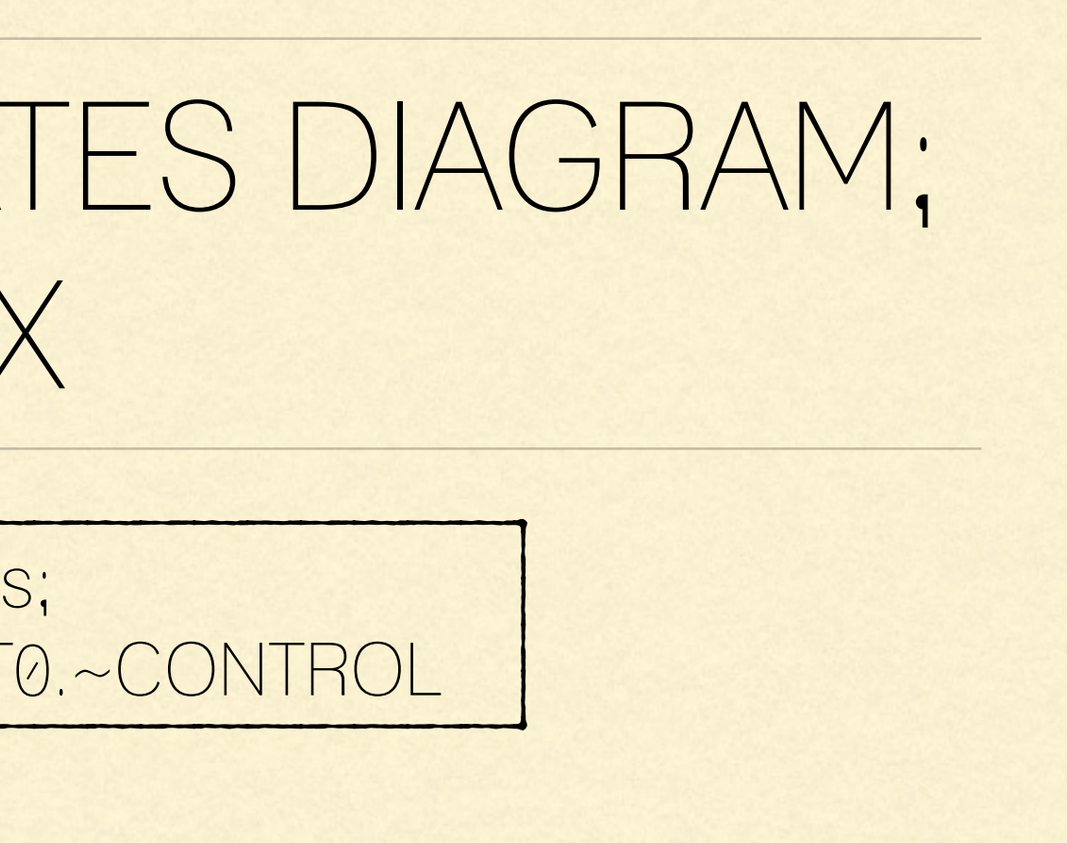
When CONTROL = 0, we only care about INPUT 0. We DON'T CARE about INPUT 1

When CONTROL = 1, we only care about INPUT 1. We DON'T CARE about INPUT 0

# TRUTH TABLE TO GATES DIAGRAM; 2-1 MUX

Sum of products;
INPUT1.CONTROL + INPUT0.~CONTROL

| INPUT 0 | INPUT 1 | CONTROL | OUTPUT |
|---|---|---|---|
| 0 | X | 0 | 0 |
| X | 0 | 1 | 0 |
| 0 | X | 0 | 0 |
| X | 1 | 1 | 1 |
| 1 | X | 0 | 1 |
| X | 0 | 1 | 0 |
| 1 | X | 0 | 1 |
| X | 1 | 1 | 1 |



# FULL ADDER: ANOTHER EXAMPLE OF DON'T CARES

- Sum of products for the ‘CARRY OUT’;

$$X.Y.\sim C_{IN} + X.\sim Y.C_{IN} + \sim X.Y.C_{IN} + X.Y.C_{IN}$$

| X | Y | CARRY IN | CARRY OUT |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

# FULL ADDER: ANOTHER EXAMPLE OF DON'T CARES

- Sum of products for the ‘CARRY OUT’;

$$X.Y.\sim C_{IN} + X.\sim Y.C_{IN} + \sim X.Y.C_{IN} + X.Y.C_{IN}$$

| X | Y | CARRY IN | CARRY OUT |
|---|---|---|---|
| 0 | 0 | X | 0 |
| X | 0 | 0 | 0 |
| 0 | X | 0 | 0 |
| 1 | 1 | X | 1 |
| 0 | 0 | X | 0 |
| 1 | X | 1 | 1 |
| X | 1 | 1 | 1 |
| 1 | 1 | X | 1 |

# FULL ADDER: ANOTHER EXAMPLE OF DON'T CARES

- Sum of products for the ‘CARRY OUT’;

$$X.Y.\sim C_{IN} + X.\sim Y.C_{IN} + \sim X.Y.C_{IN} + X.Y.C_{IN}$$

| X | Y | CARRY IN | CARRY OUT |
|---|---|---|---|
| 0 | 0 | X | 0 |
| X | 0 | 0 | 0 |
| 0 | X | 0 | 0 |
| 1 | 1 | X | 1 |
| 1 | X | 1 | 1 |
| X | 1 | 1 | 1 |

# FULL ADDER: ANOTHER EXAMPLE OF DON'T CARES

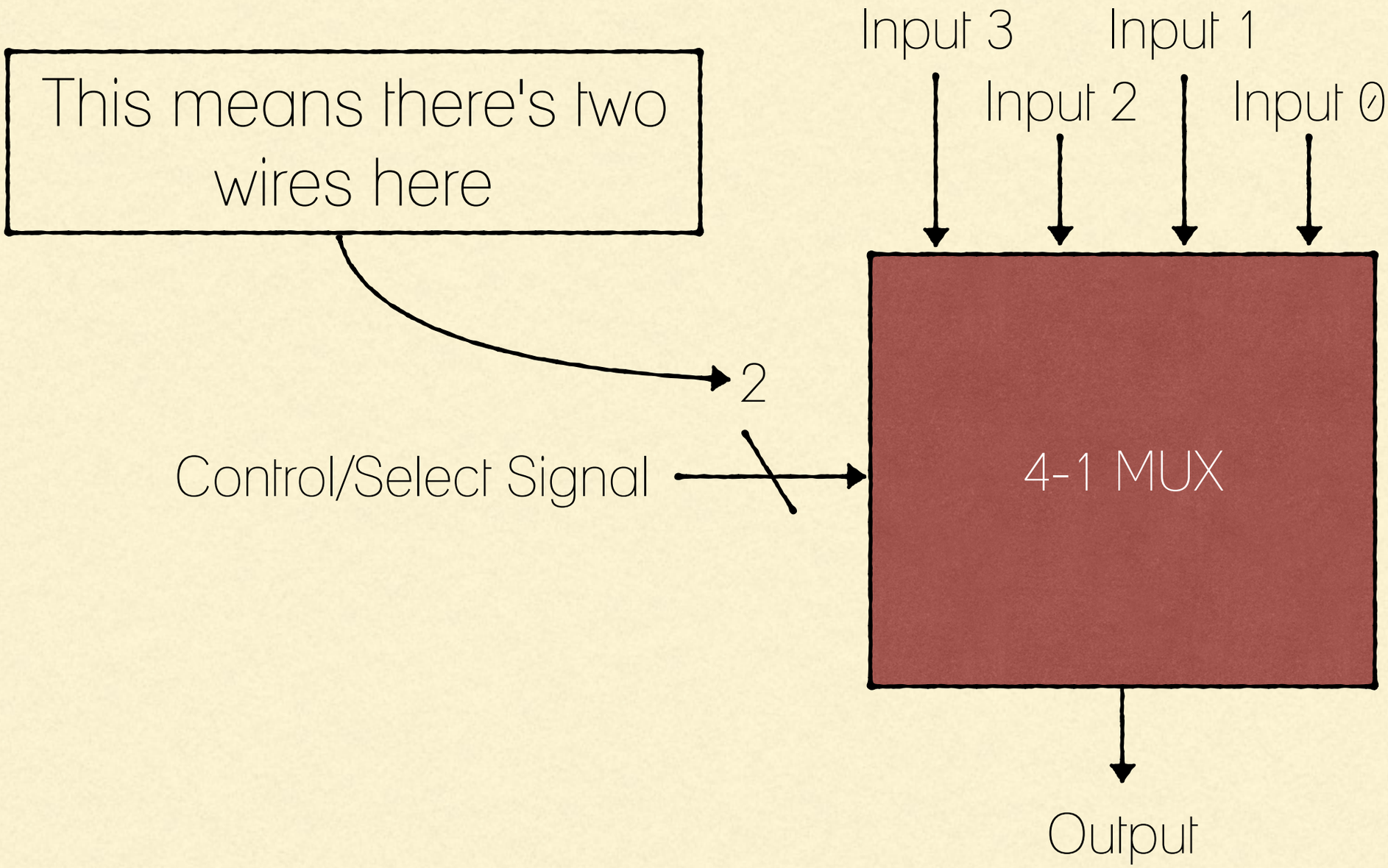
- Sum of products for the ‘CARRY OUT’;

$$X.Y.\sim C_{IN} + X.\sim Y.C_{IN} + \sim X.Y.C_{IN} + X.Y.C_{IN}$$

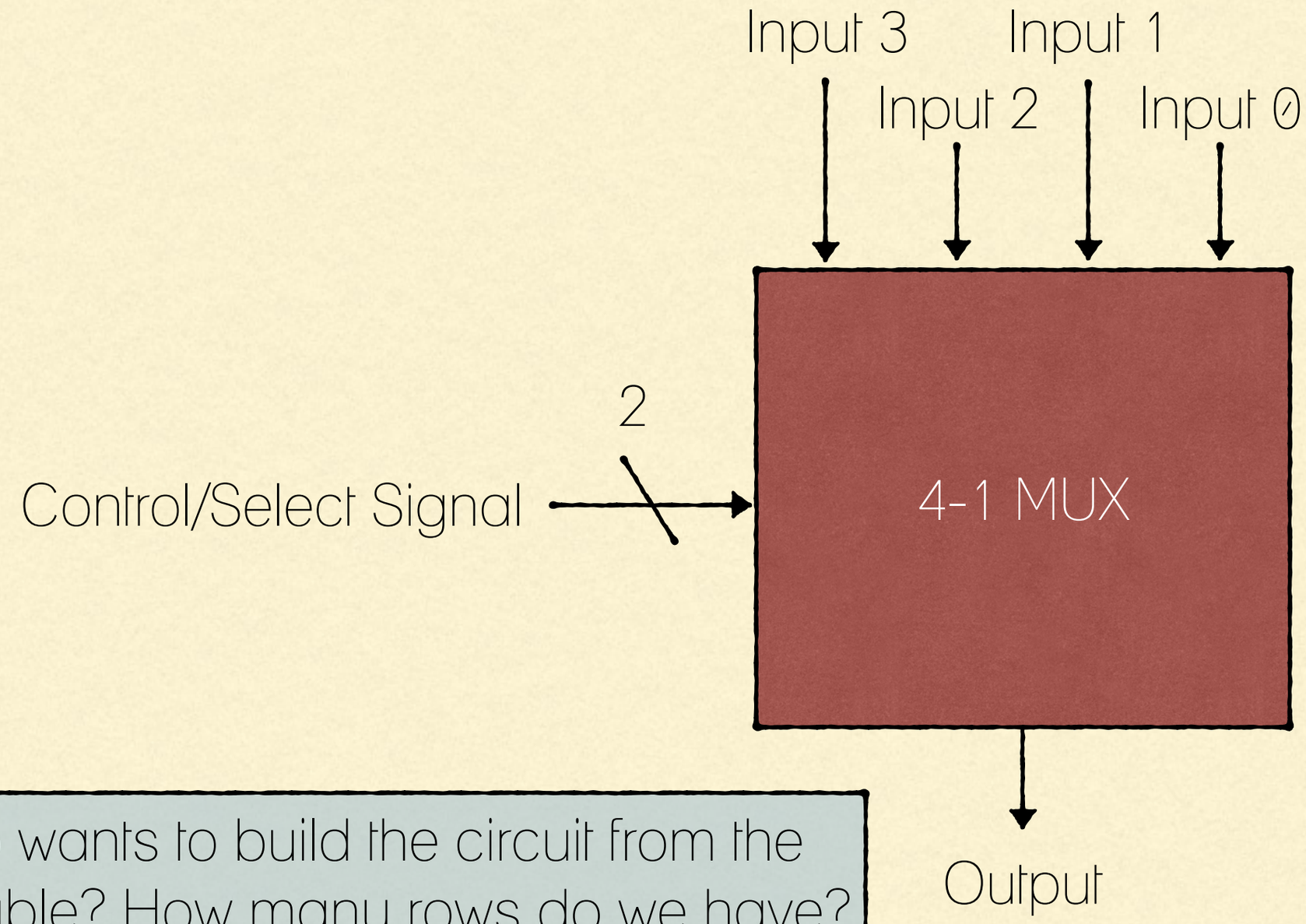
$$X.Y + Y.C_{IN} + X.C_{IN}$$

| X | Y | CARRY IN | CARRY OUT |
|---|---|---|---|
| 0 | 0 | X | 0 |
| X | 0 | 0 | 0 |
| 0 | X | 0 | 0 |
| 1 | 1 | X | 1 |
| 1 | X | 1 | 1 |
| X | 1 | 1 | 1 |

# 4-1 MUX

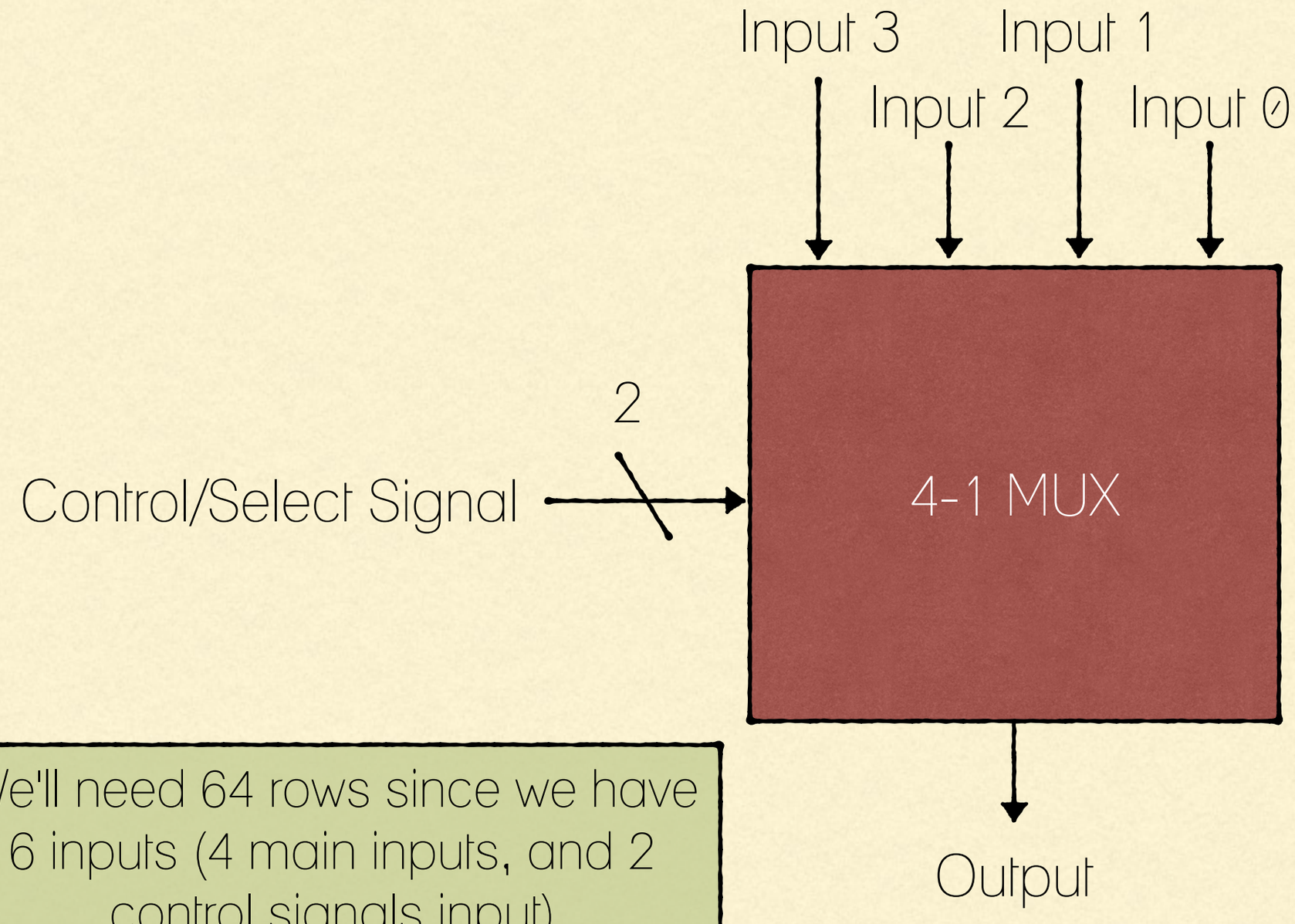

This means there's two wires here
2
Control/Select Signal
Input 3
Input 2
Input 1
Input 0
4-1 MUX
Output

# 4-1 MUX




Who wants to build the circuit from the truth table? How many rows do we have?

# 4-1 MUX

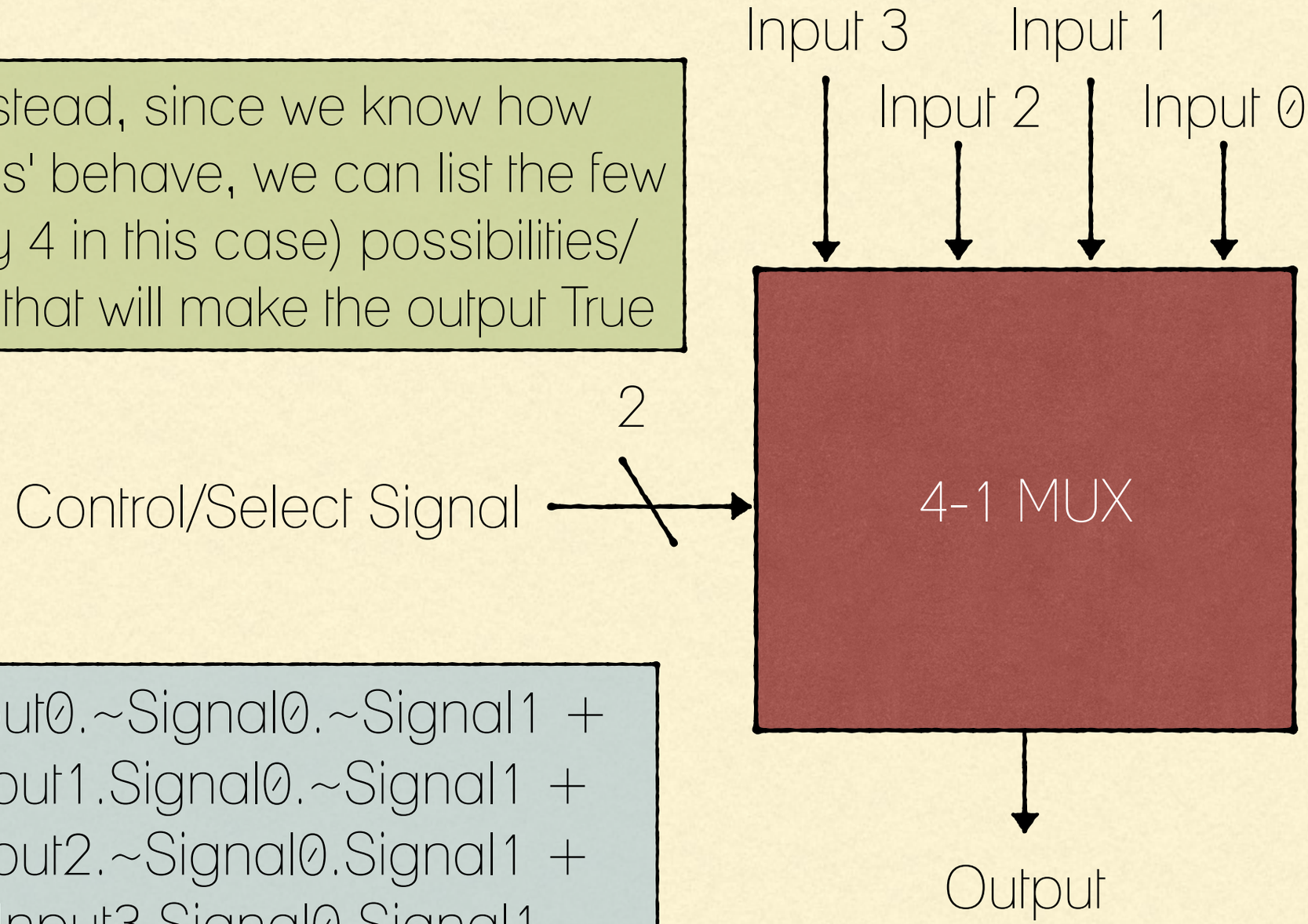



We'll need 64 rows since we have 6 inputs (4 main inputs, and 2 control signals input)

| Input 0 | Input 1 | Input 2 | Input 3 | Signal 0 | Signal 1 | Result |
|---|---|---|---|---|---|---|
| | | | | | | |

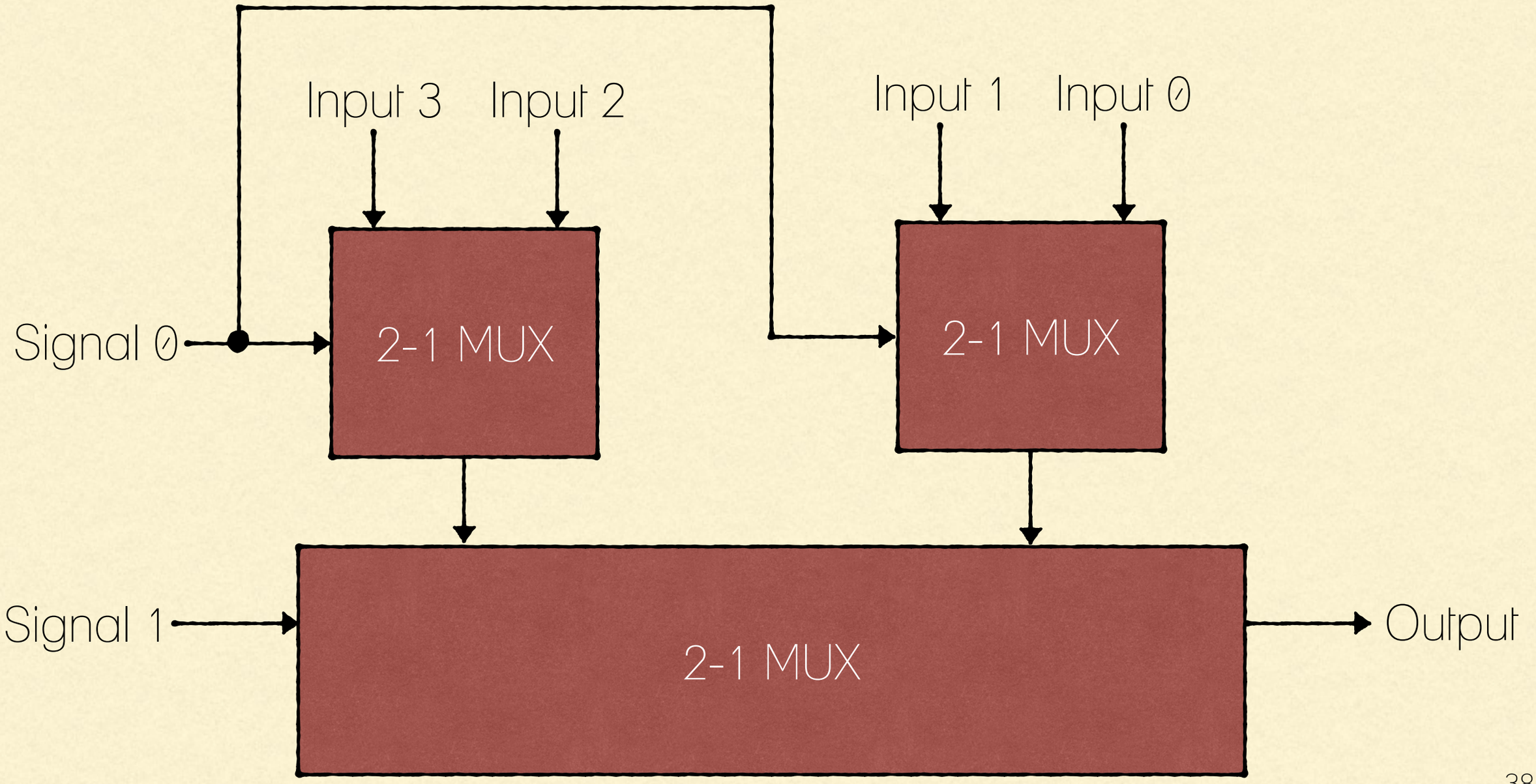
# 4-1 MUX

Instead, since we know how 'muxes' behave, we can list the few (only 4 in this case) possibilities/ rows that will make the output True

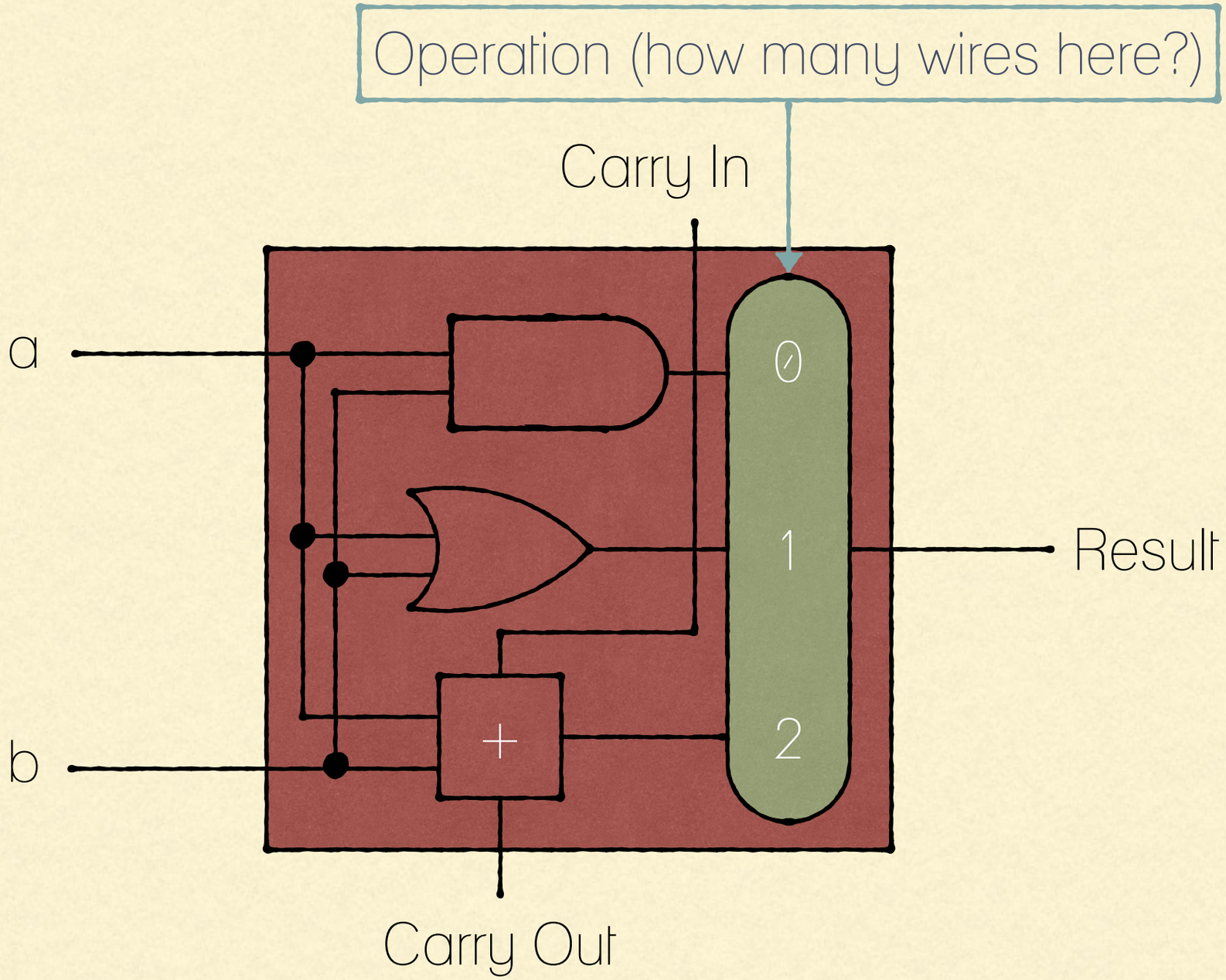



Input0.~Signal0.~Signal1 +
Input1.Signal0.~Signal1 +
Input2.~Signal0.Signal1 +
Input3.Signal0.Signal1

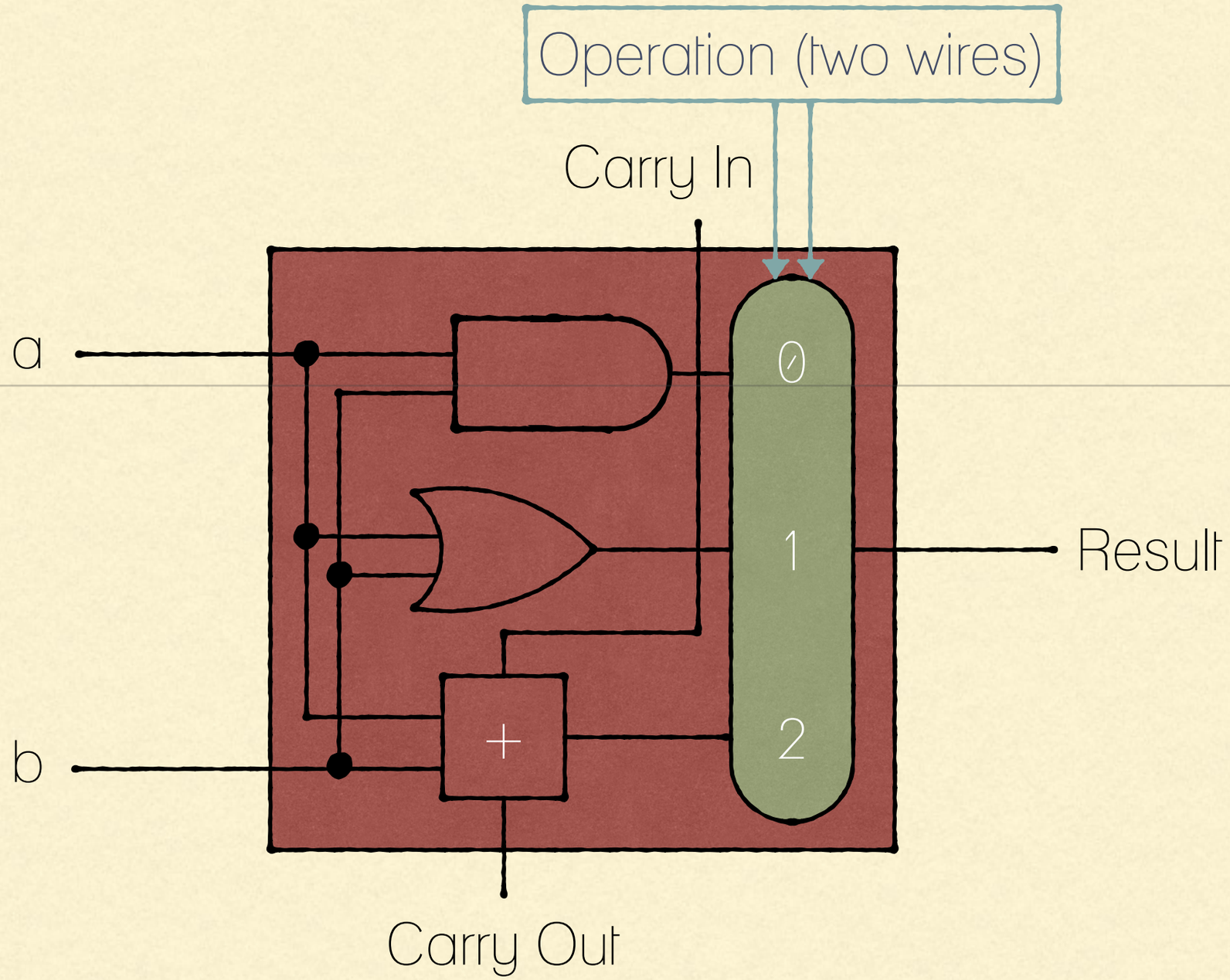
# WE CAN ALSO USE THREE 2-1 MUXES TO CONSTRUCT A 4-1 MUX



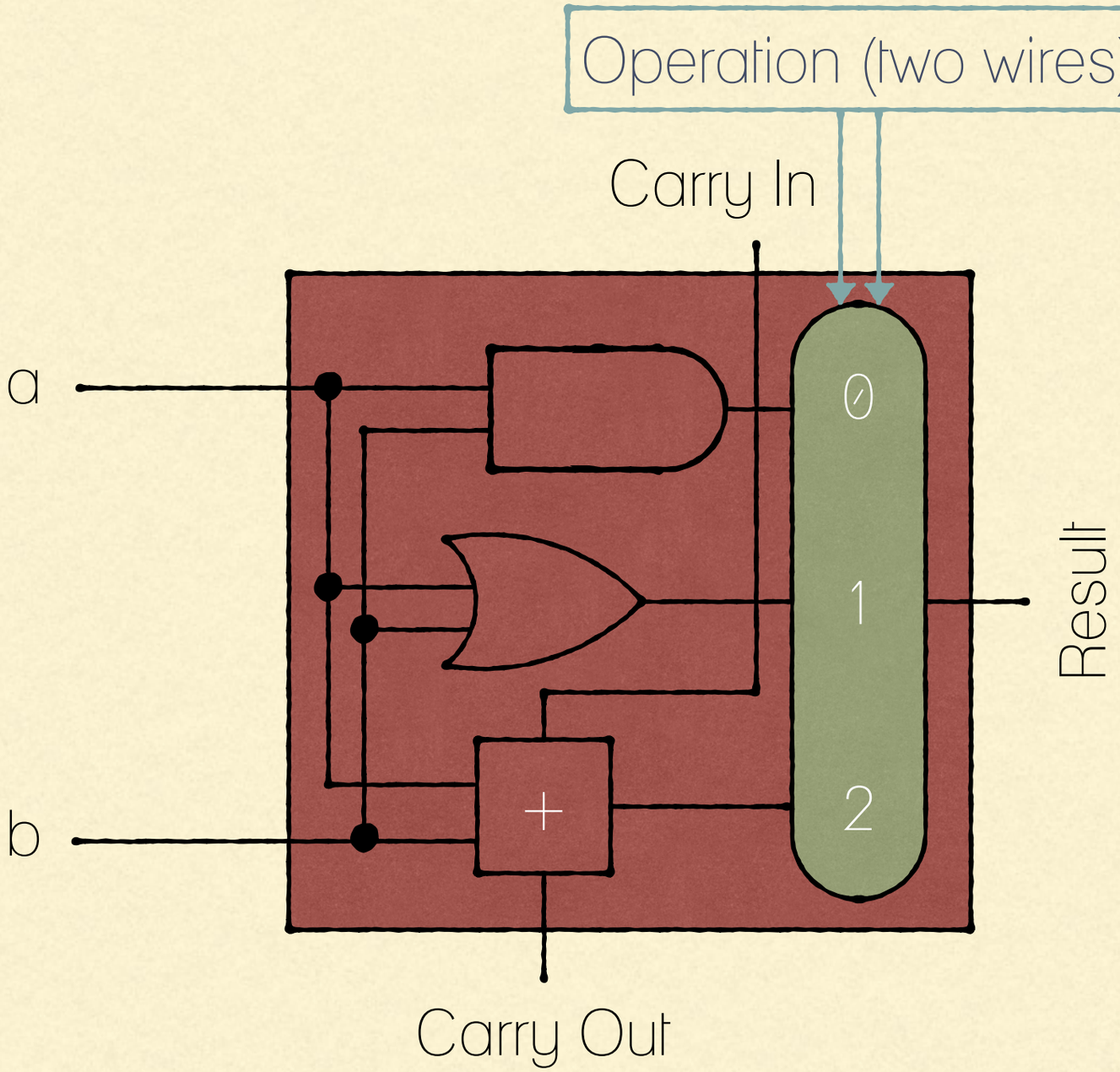
# 1-BIT ALU THAT PERFORMS 'AND', 'OR', & ADDITION



# 1-BIT ALU THAT PERFORMS 'AND', 'OR', & ADDITION

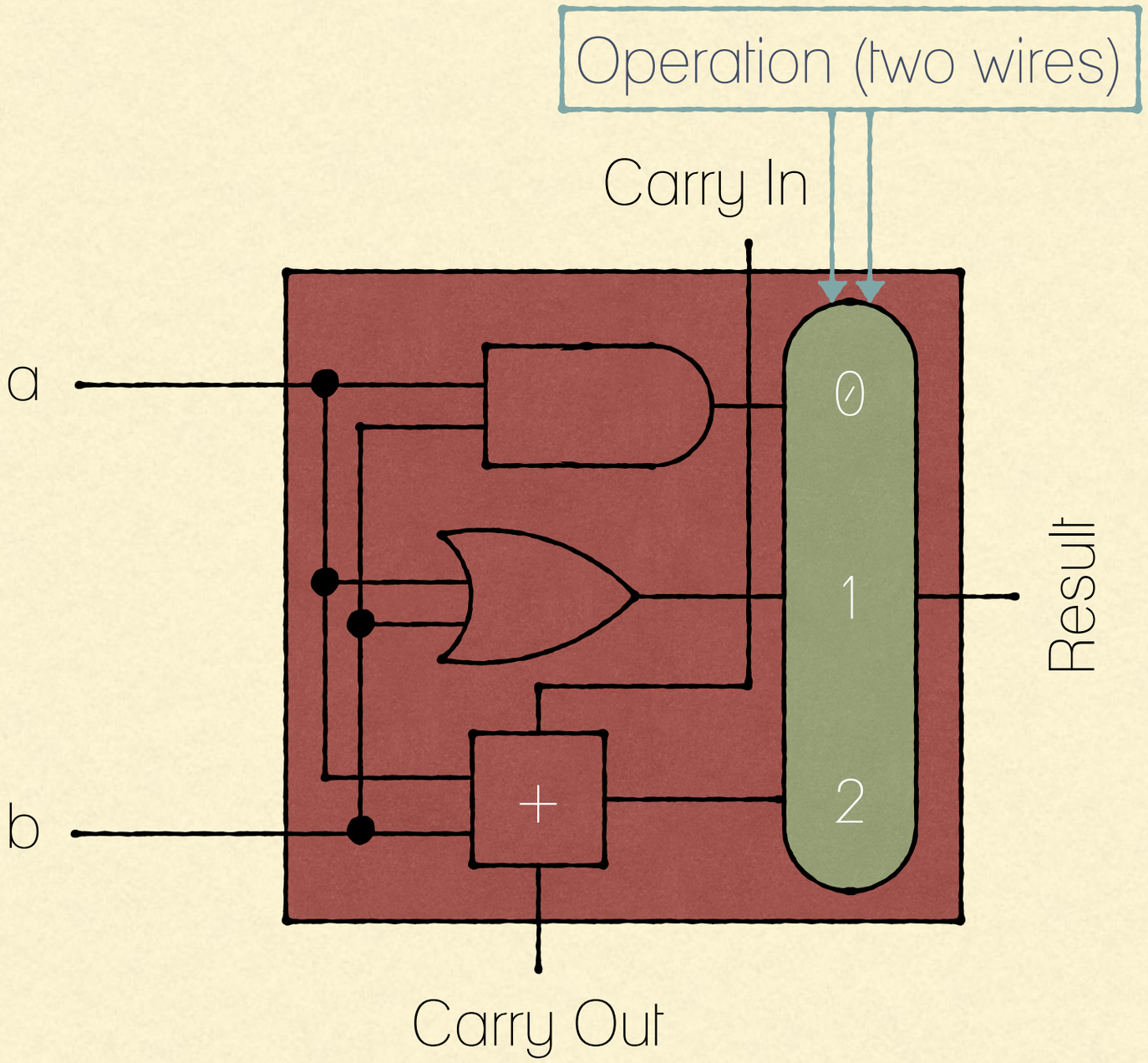


# 1-BIT ALU THAT PERFORMS 'AND', 'OR', & ADDITION




| a | b | CS1 | CS0 | Result |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | ? |
| 0 | 0 | 0 | 1 | ? |
| 0 | 0 | 1 | 0 | ? |
| 0 | 1 | 0 | 0 | ? |
| 0 | 1 | 0 | 1 | ? |
| 0 | 1 | 1 | 0 | ? |
| 1 | 0 | 0 | 0 | ? |
| 1 | 0 | 0 | 1 | ? |
| 1 | 0 | 1 | 0 | ? |
| 1 | 1 | 0 | 0 | ? |
| 1 | 1 | 0 | 1 | ? |
| 1 | 1 | 1 | 0 | ? |

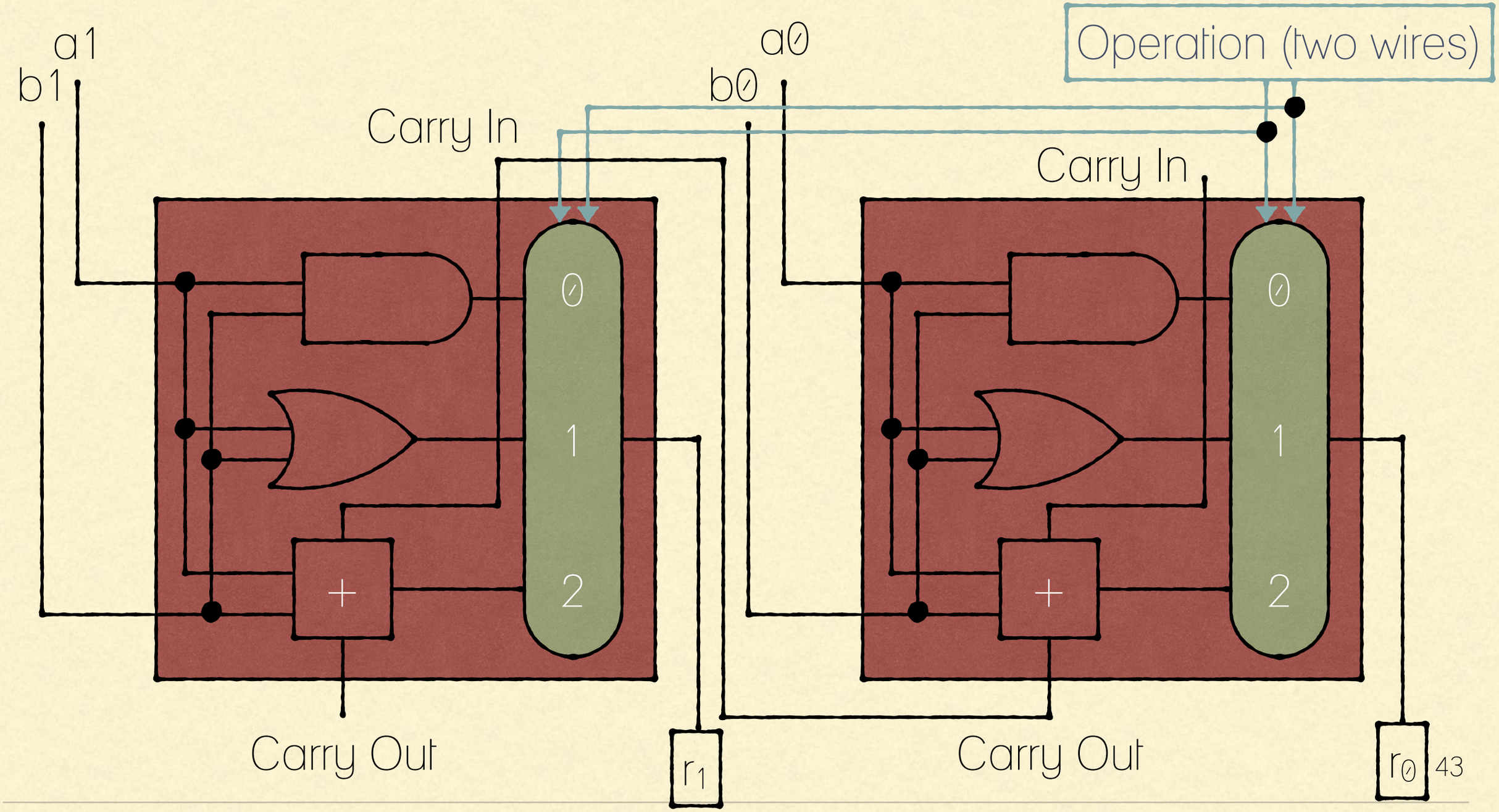
# 1-BIT ALU THAT PERFORMS 'AND', 'OR', & ADDITION



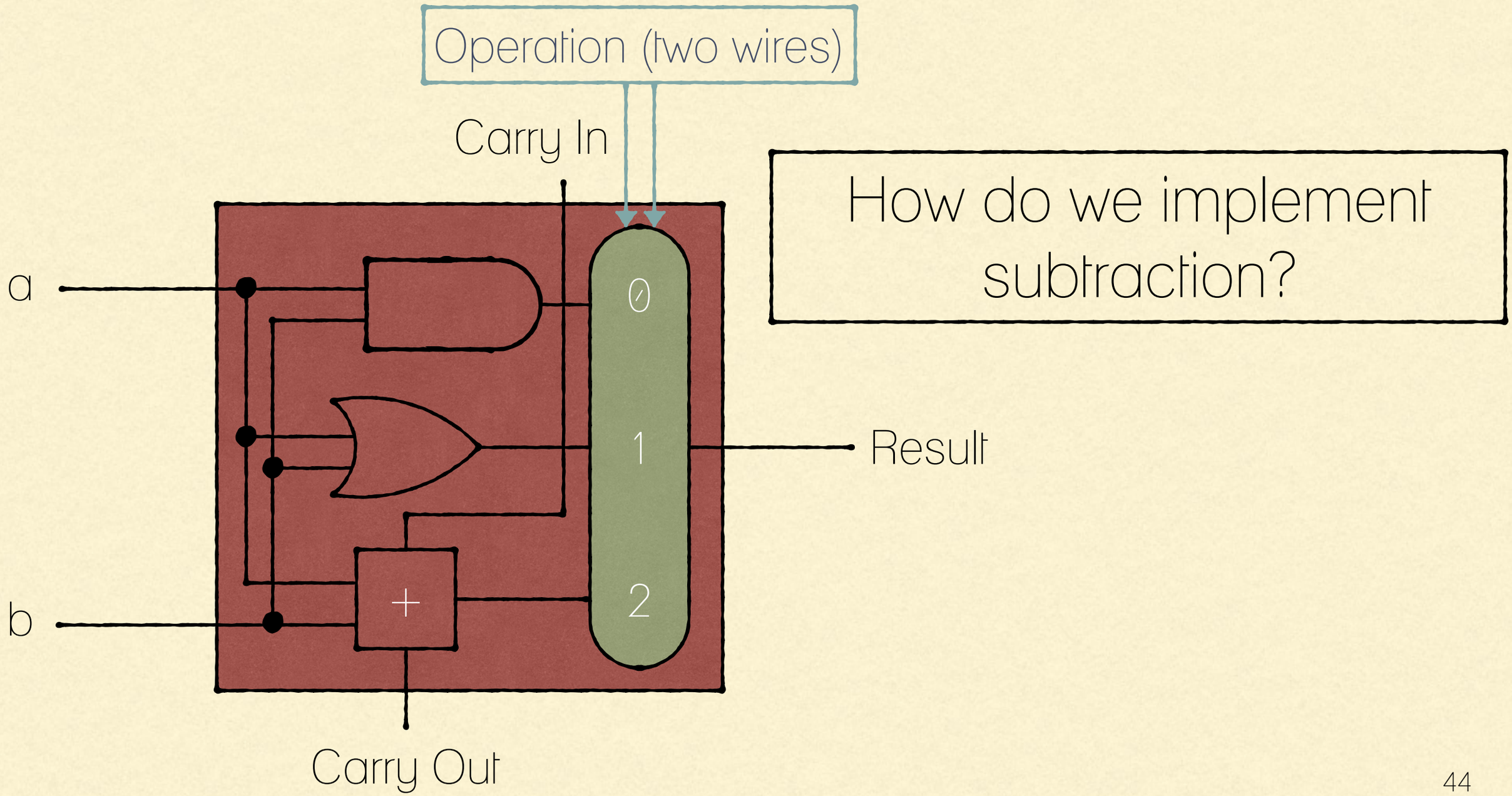

| a | b | CS1 | CS0 | Result |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |

# 2-BIT ALU THAT PERFORMS 'AND', 'OR', & ADDITION

$a_1a_0$ AND $b_1b_0$ | $a_1a_0$ OR $b_1b_0$ | $a_1a_0$ + $b_1b_0$

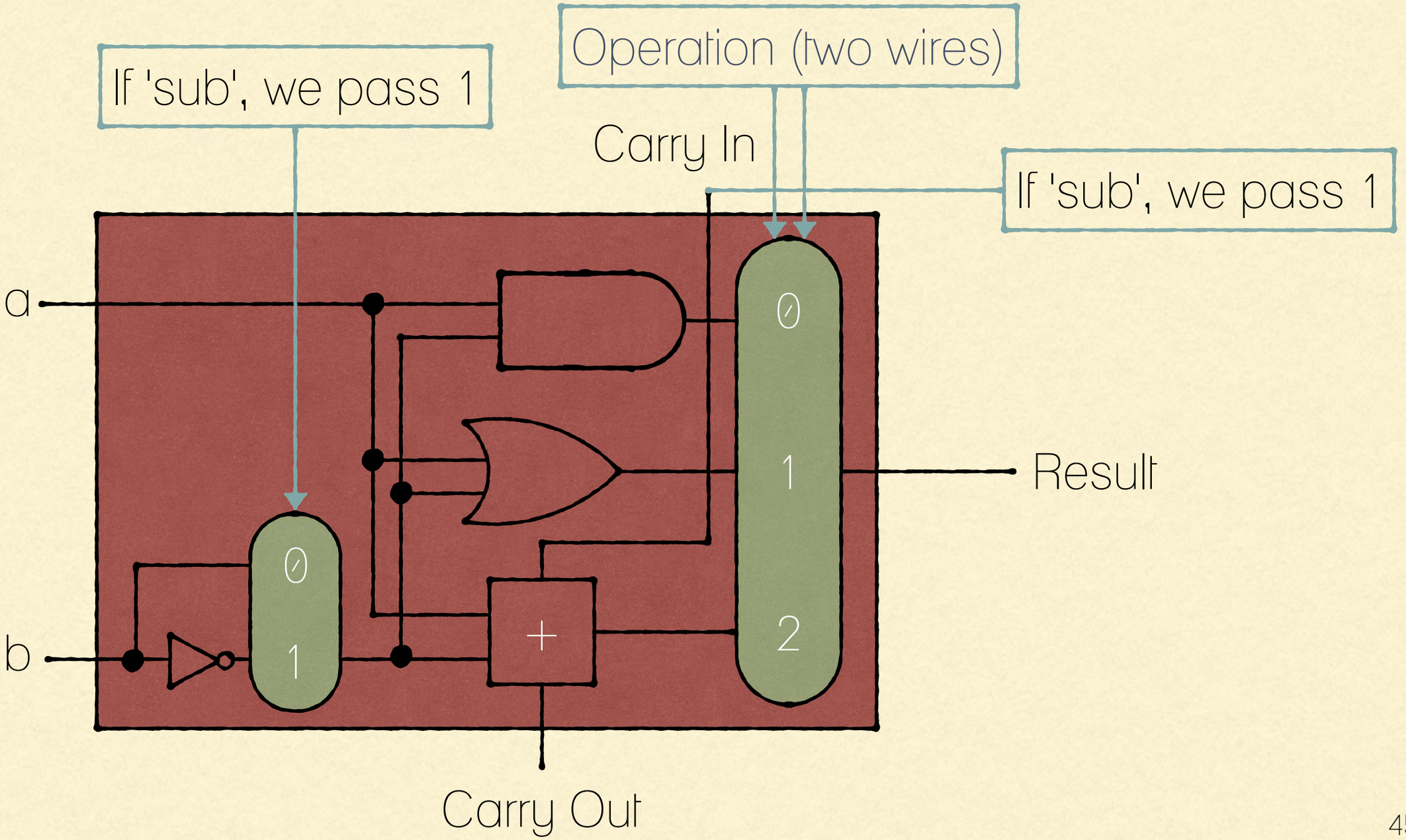


# 1-BIT ALU THAT PERFORMS 'AND', 'OR', ADDITION, & SUBTRACTION

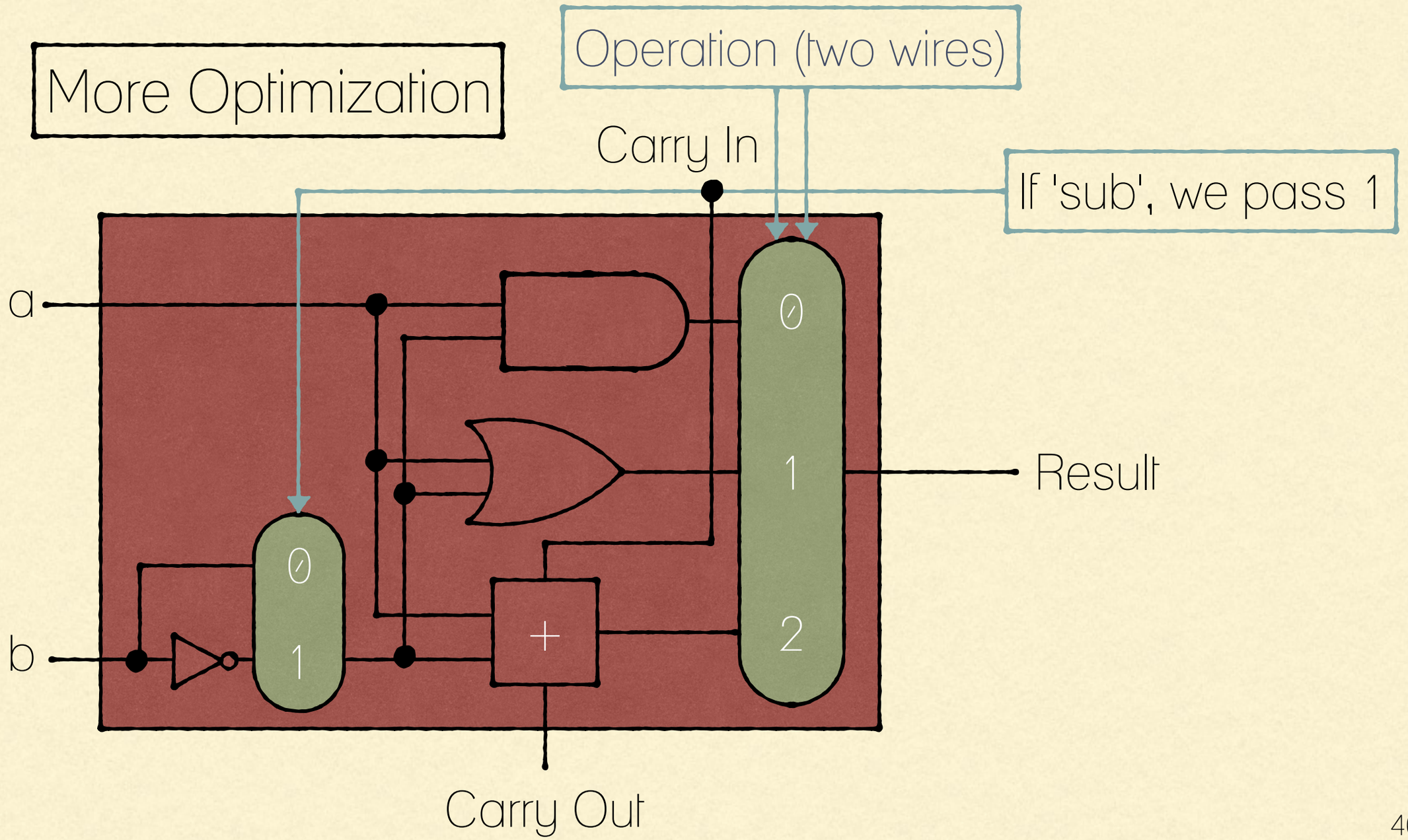



How do we implement subtraction?

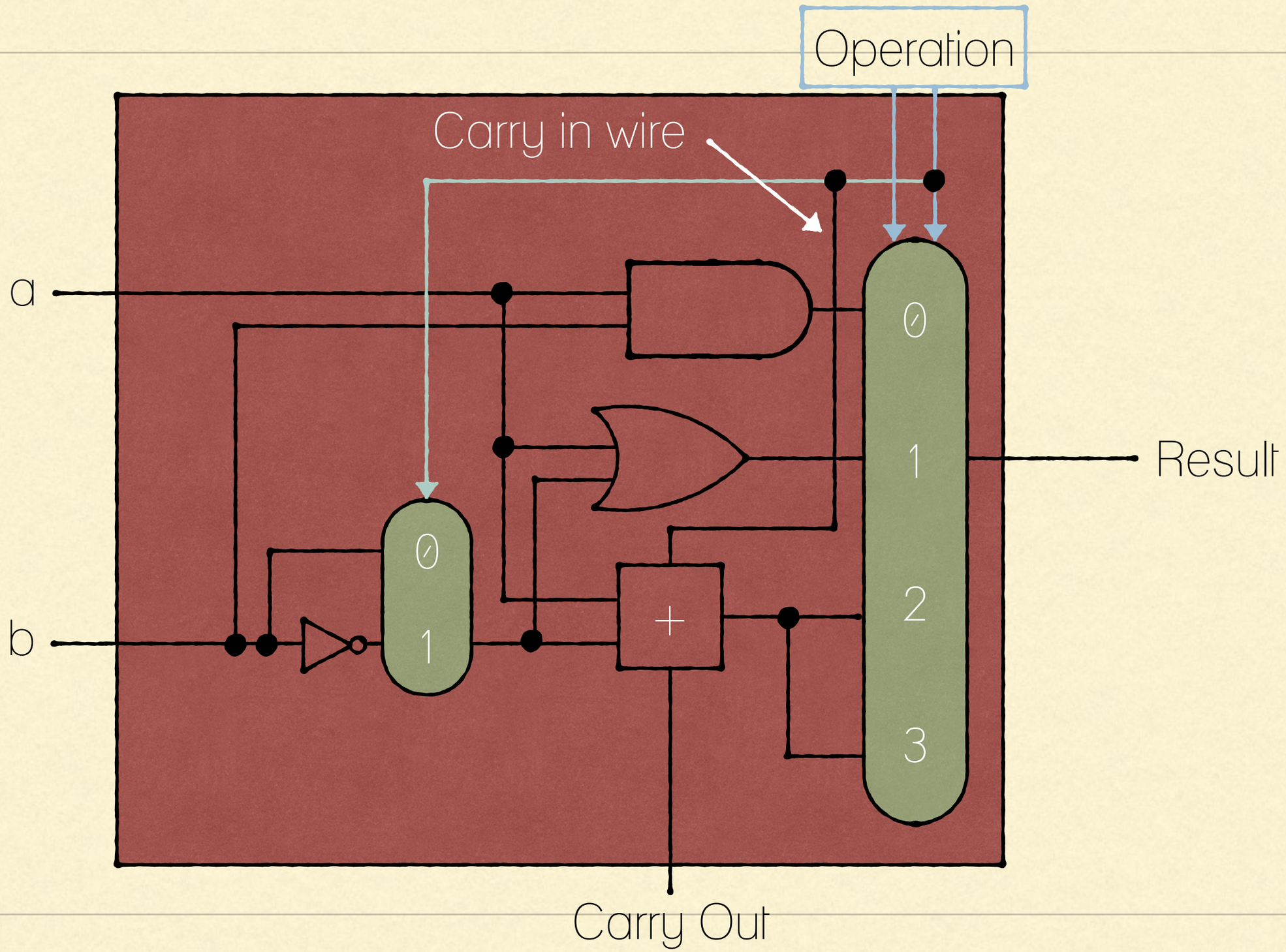
# 1-BIT ALU THAT PERFORMS 'AND', 'OR', ADDITION, & SUBTRACTION



# 1-BIT ALU THAT PERFORMS 'AND', 'OR', ADDITION, & SUBTRACTION

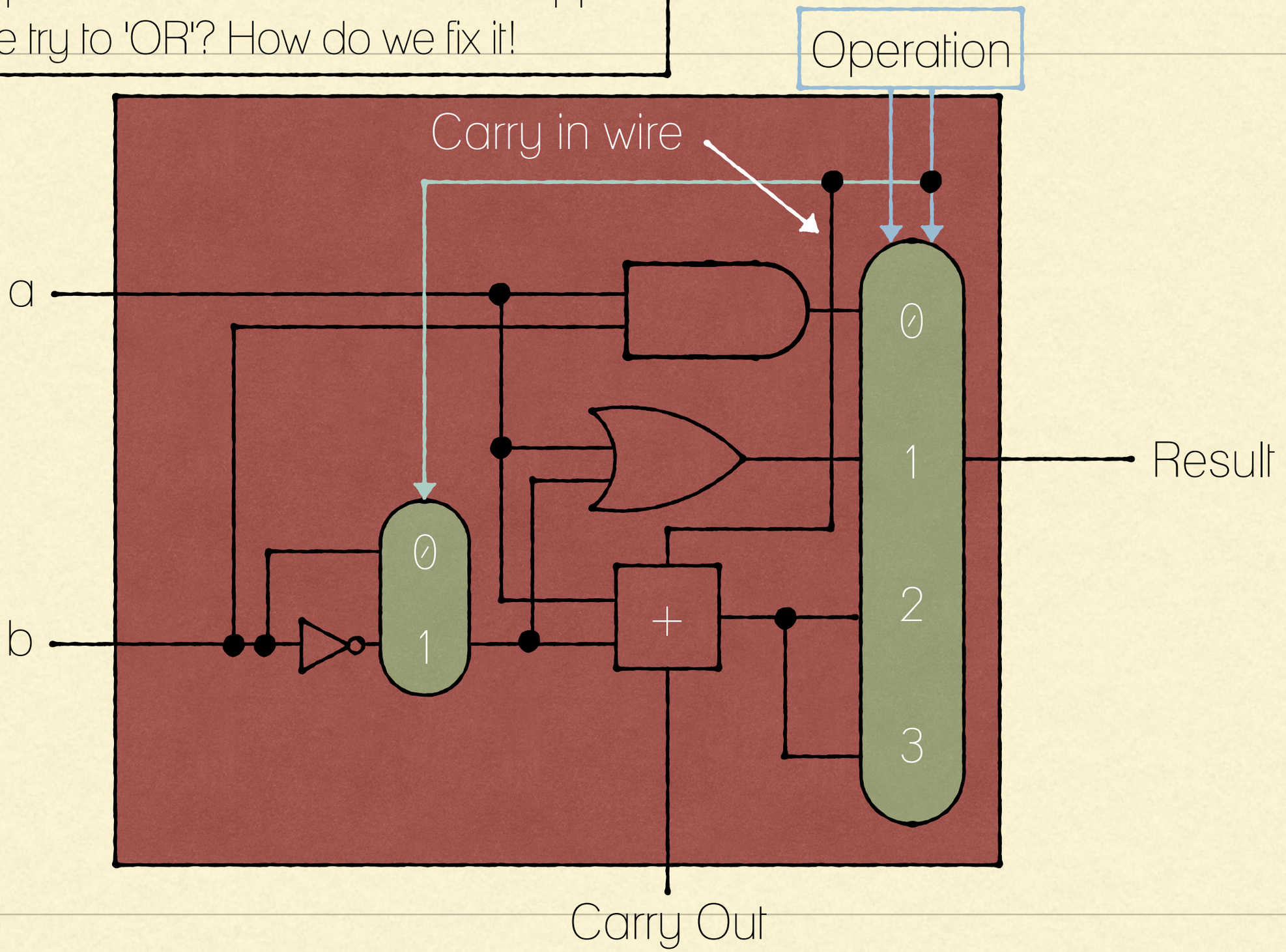


Let's use the 1st control signal from the main operation multiplexer.
00 is AND, 01 is OR, 10 is ADD, & 11 is SUB

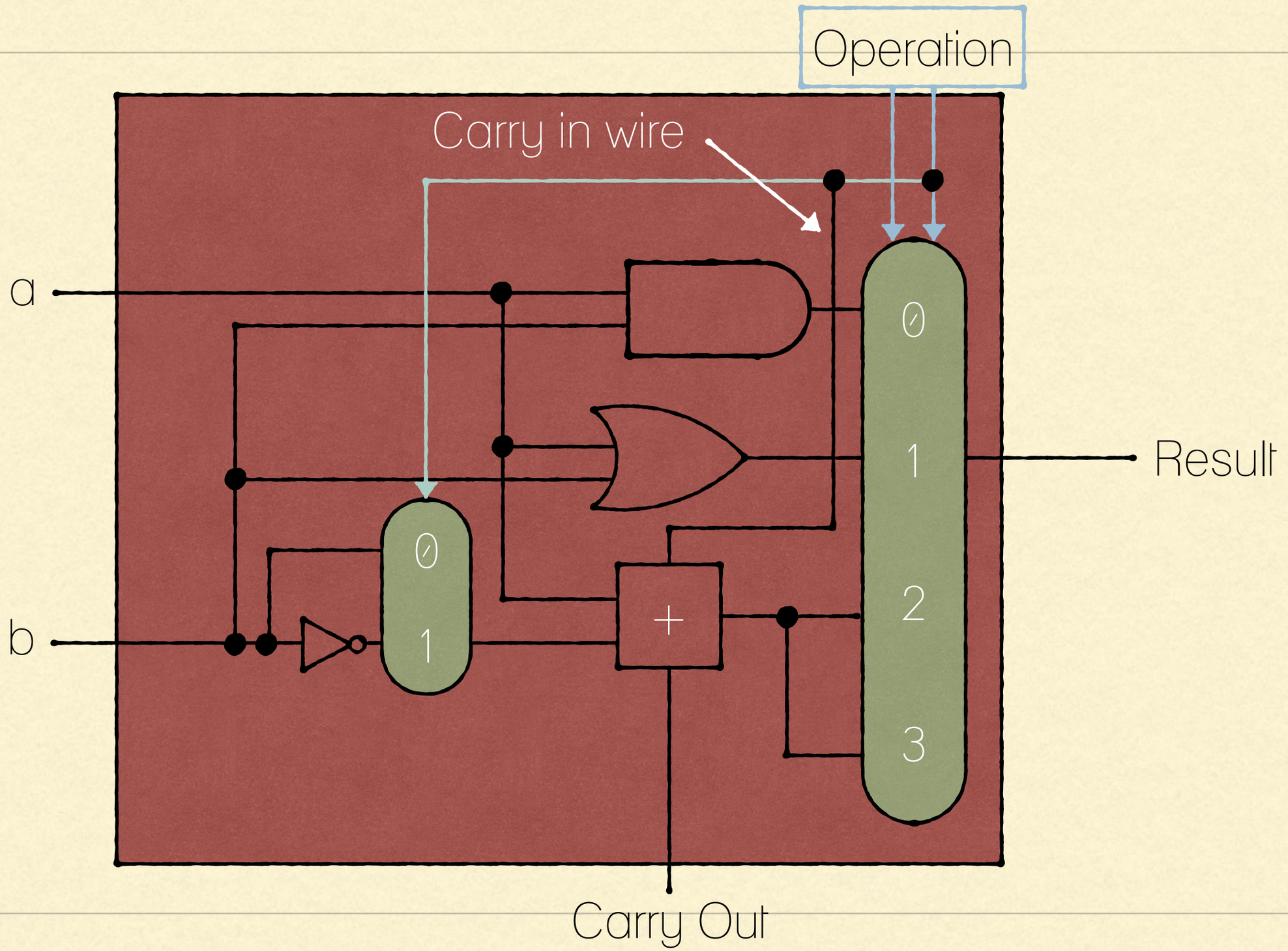


Let's use the 1st control signal from the main operation multiplexer.
00 is AND, 01 is OR, 10 is ADD, & 11 is SUB

What's the problem here? What would happen if we try to 'OR'? How do we fix it!



Let's use the 1st control signal from the main operation multiplexer.
00 is AND, 01 is OR, 10 is ADD, & 11 is SUB



# FUNCTIONAL COMPLETENESS

- A set of Boolean functions is functionally complete, if all other Boolean functions can be constructed from this set and a set of input variables are provided.
- The set of functions {AND, OR, NOT} are functionally complete

# FUNCTIONAL COMPLETENESS

- A set of Boolean functions is functionally complete, if all other Boolean functions can be constructed from this set and a set of input variables are provided.
- The set of functions {AND, OR, NOT} are functionally complete

> Can we reconstruct ‘OR’ from ‘AND’ & ‘NOT’? In other words, can we find an expression that is logically equivalent to ‘A OR B’ using only ‘AND’s & ‘NOT’s?

# FUNCTIONAL COMPLETENESS

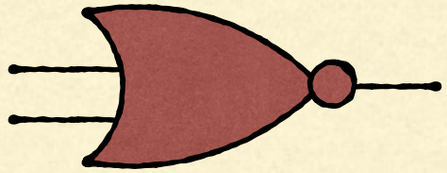
- A set of Boolean functions is functionally complete, if all other Boolean functions can be constructed from this set and a set of input variables are provided.
- The set of functions {AND, OR, NOT} are functionally complete

Can we reconstruct ‘OR’ from ‘AND’ & ‘NOT’? In other words, can we find an expression that is logically equivalent to ‘A OR B’ using only ‘AND’s & ‘NOT’s?

Yes we can. ‘A OR B’ is logically equivalent to ~(~A . ~B)

# MORE COMBINATIONAL LOGIC GATES: NOR

Boolean operator that gives the value one if and only if all operands have a value of zero, and otherwise has a value of zero.

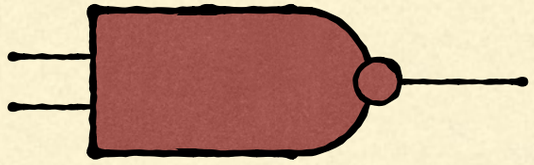


NOR Truth Table

| A | B | A NOR B |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

# MORE COMBINATIONAL LOGIC GATES: NAND

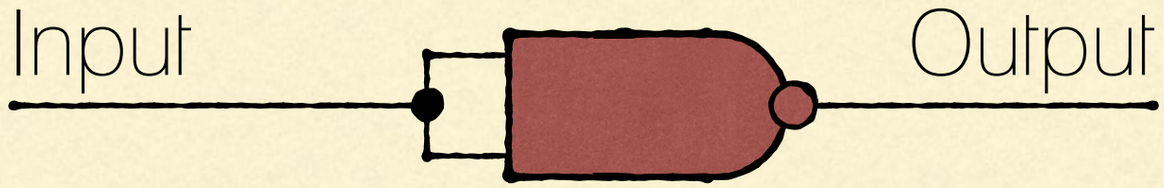
Boolean operator that gives the value zero if and only if all the operands have a value of one, and otherwise has a value of one.



NAND Truth Table

| A | B | A NAND B |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

# CONSTRUCTING ‘NOT’ FROM ‘NAND’



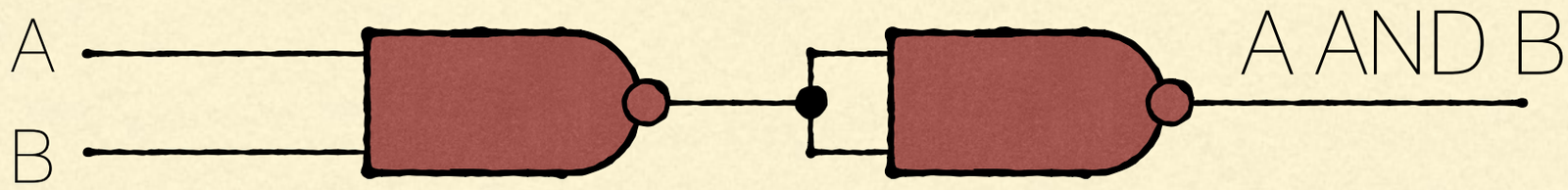

| A | A NAND A |
|---|---|
| 1 | 0 |
| 0 | 1 |

# CONSTRUCTING ‘NOT’ FROM ‘NAND’


| A | A NAND A |
|---|---|
| 1 | 0 |
| 0 | 1 |

How do we reconstruct AND from NAND?

# CONSTRUCTING ‘AND’ FROM ‘NAND’




| A | B | (A NAND B) NAND (A NAND B) |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 0 | 0 | 0 |

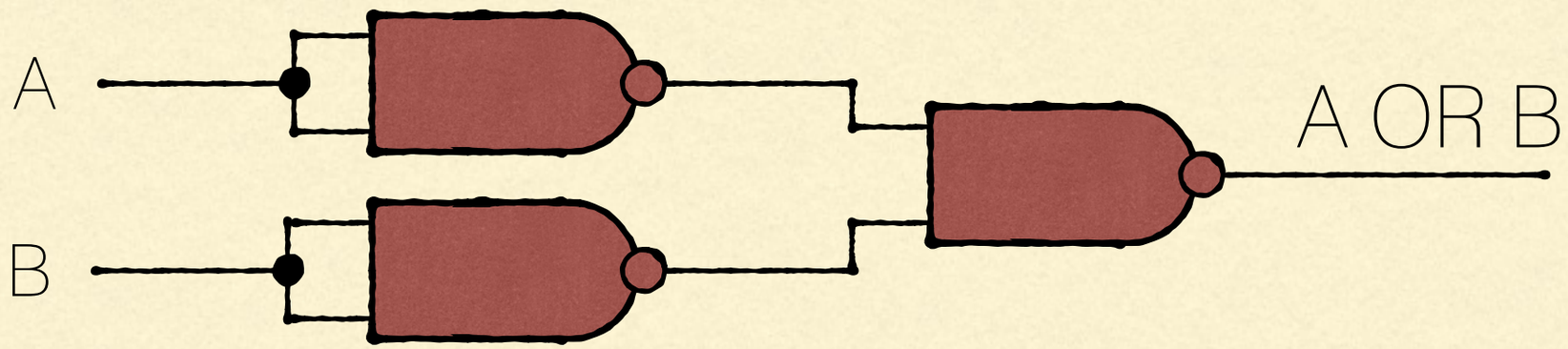
# CONSTRUCTING ‘OR’ FROM ‘NAND’

Recall that A OR B can be rewritten as:

~(~A . ~B) — which is ~A NAND ~B

# CONSTRUCTING ‘OR’ FROM ‘NAND’

Recall that A OR B can be rewritten as:
~(~A . ~B) — which is ~A NAND ~B

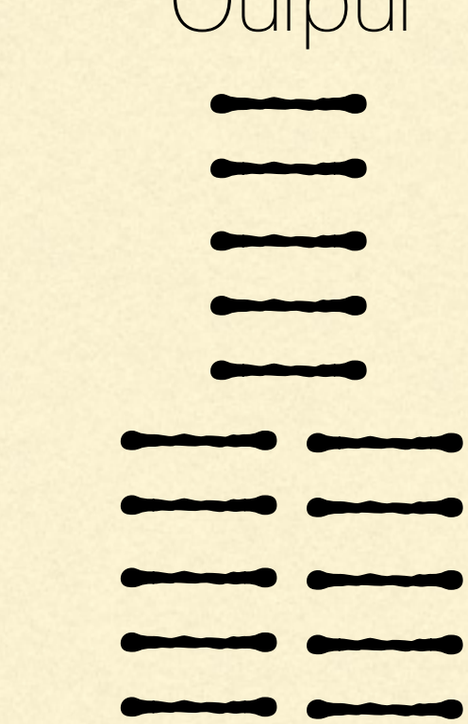


# DOMINO COMPUTER

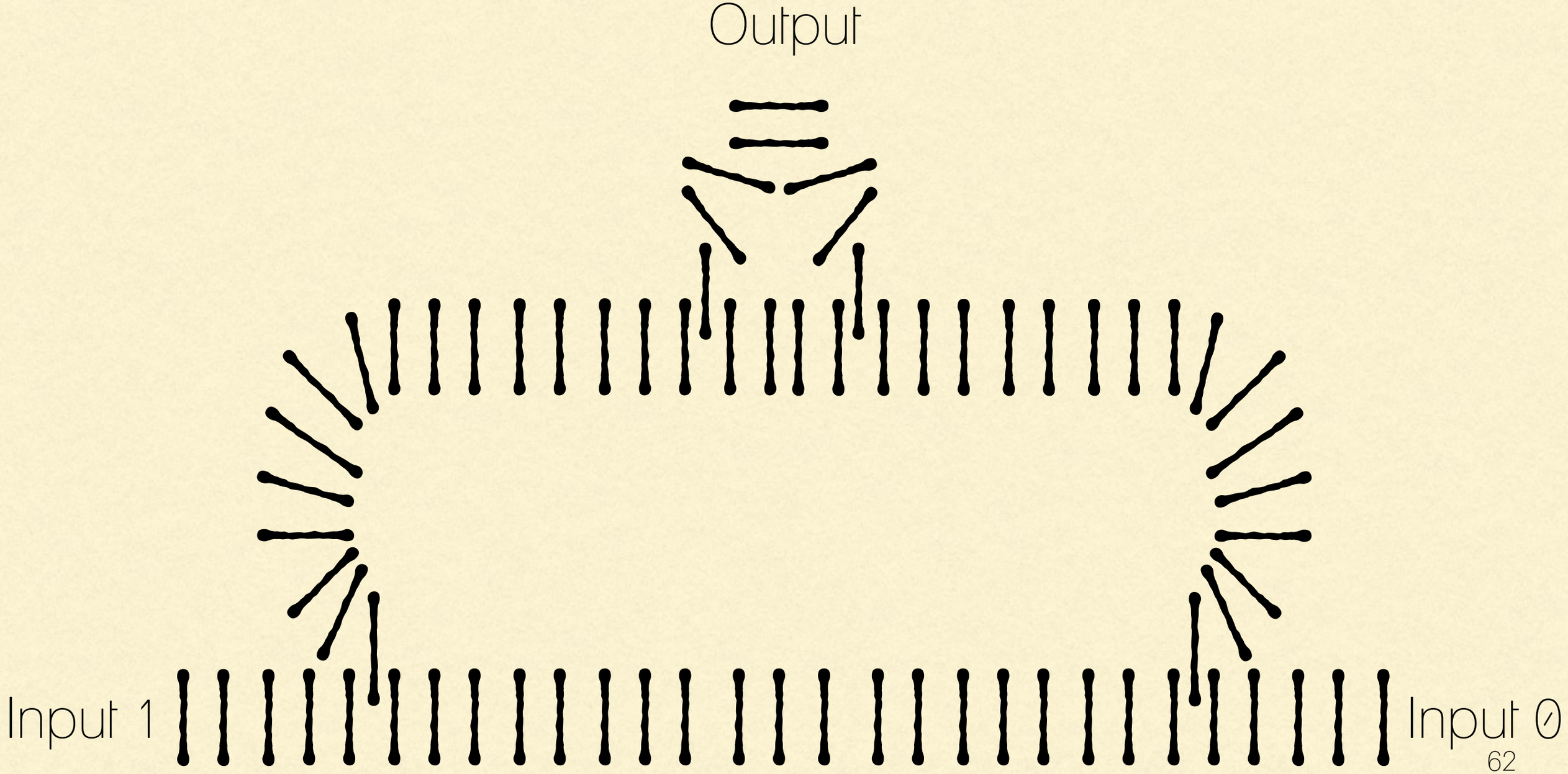
- A domino computer is a mechanical computer built using dominoes to represent logic gating of digital signals.

# DOMINO COMPUTER

What logic gate does this domino configuration represent?



# HOW ABOUT THIS ONE?



# DOMINO COMPUTER

https://www.youtube.com/watch?v=OpLU__bhu2w