**Chapter 4**

Table of Contents

[Design Procedure 3](#_Toc44600330)

[Analysis Procedure 4](#_Toc44600331)

[NAND and NOR Implementation with Analysis Procedure 5](#_Toc44600332)

[Code Conversion Between Systems 6](#_Toc44600333)

[Binary Adders 10](#_Toc44600334)

[Carry Lookahead Generator 13](#_Toc44600335)

[Subtractor 16](#_Toc44600336)

[BCD Adder 18](#_Toc44600337)

[Binary Multiplier 21](#_Toc44600338)

[Magnitude Comparator 24](#_Toc44600339)

[Decoders 26](#_Toc44600340)

[Decoder’s with NAND Gates 27](#_Toc44600341)

[Combining Decoders 28](#_Toc44600342)

[Combinational Logic Implementation with Decoders 29](#_Toc44600343)

[Encoders 30](#_Toc44600344)

[Priority Encoders 31](#_Toc44600345)

[Multiplexers 33](#_Toc44600346)

[Demultiplexer 35](#_Toc44600347)

[Boolean Function Implementation with Multiplexers 36](#_Toc44600348)

## Design Procedure

1. The problem is stated.
2. The number of available input variables and required output variables is determined.
3. The input and output variables are assigned letter symbols.
4. The truth table that defines the required relationship between inputs and outputs is derived.
5. The simplified Boolean function for each output is obtained.
6. The logic diagram is drawn.

* Input columns are obtained from the binary combinations available from input variables.
* Binary values of the output columns are determined from examination of the stated problem.
* Specifications may indicate some input combinations will not occur. These combinations are don’t care conditions.
* Word specifications are rarely complete or exact, which may lead to incorrect interpretations and hence incorrect combinational circuits that do not fulfill the requirements.
* There may be multiple possible simplified expressions. Restrictions such as minimum number of gates, minimum number of inputs to a gate, minimum propagation time of the signal through the circuit, minimum number of interconnections and limitations of the driving capabilities of each gate will help decided which expression to choose.
* A logic diagram is useful to visualize the gate implementation of the expression.

## Analysis Procedure

1. Ensure that the given circuit is combinational. This means checking for feedback paths (output of one gate leading to input of another gate, the output of which goes back into the input of that first gate) and memory elements, both of which would make it a sequential circuit and not a combinational one.
2. Give arbitrary labels to each intermediate output and obtain the Boolean function for each.
3. Repeat the process until the Boolean function for the circuit’s outputs are obtained.
4. By repeated substitution, obtain circuit output functions in terms of input variables only.
5. The truth table must then be derived from the output function.

The truth table can also be derived directly from the logic diagram.

1. Determine the number of input variables. input variables will give possible input combinations. List these combinations from to in binary form in the truth table.
2. Label the intermediate outputs with arbitrary symbols and find the output values of each. Put these outputs in the truth table as required.
3. Repeat this process until the circuits output is found

### NAND and NOR Implementation with Analysis Procedure

1. The Boolean function for the circuit’s output is found from the diagram as before. De Morgan’s law will be useful here.
2. The truth table is obtained.
3. The K’ Map method is then used to obtained the simplified expression.

Another way to do this is to transform the given block diagram.

1. For a NAND only circuit, alternate levels starting from the final gate are changed from the NAND gate to an INVERT-OR gate.
2. For a NOR only circuit, alternate levels starting from the final gate are changed from the NOR gate to an INVERT-AND gate.
3. Inversions that fall on the same line are then removed.

This method is the reverse of the design procedure used to implement a NAND only circuit.

## Code Conversion Between Systems

Different systems might use different coding formats and transferring data between them means the code must be converted using a converter circuit.

For example, converting from BCD to Excess 3:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Decimal | BCD | | | | Excess 3 | | | |
|  |  |  |  |  |  |  |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

The digits 10 – 15 can be created using the 10 digits above, so they are not being considered. They are don’t cares.

For ,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ab cd | 00 | 01 | 11 | 10 |
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 1 | 1 | 1 |
| 11 | X | X | X | X |
| 10 | 1 | 1 | X | X |

For ,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ab cd | 00 | 01 | 11 | 10 |
| 00 | 0 | 1 | 1 | 1 |
| 01 | 1 | 0 | 0 | 0 |
| 11 | X | X | X | X |
| 10 | 0 | 1 | X | X |

(this can be simplified to )

For ,

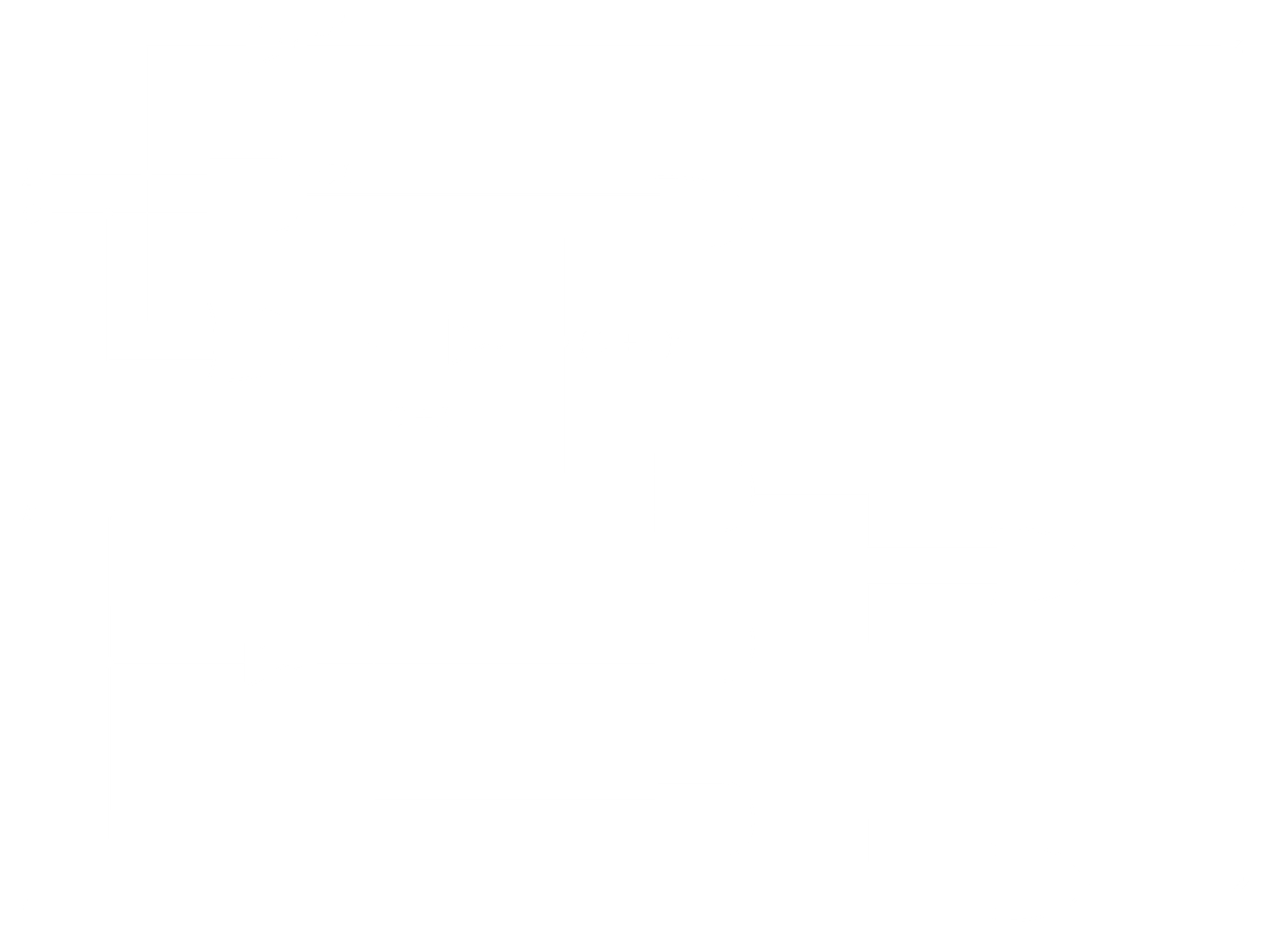
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ab cd | 00 | 01 | 11 | 10 |
| 00 | 1 | 0 | 1 | 0 |
| 01 | 1 | 0 | 1 | 0 |
| 11 | X | X | X | X |
| 10 | 1 | 0 | X | X |

(this can be simplified to )

For ,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ab cd | 00 | 01 | 11 | 10 |
| 00 | 1 | 0 | 0 | 1 |
| 01 | 1 | 0 | 0 | 1 |
| 11 | X | X | X | X |
| 10 | 1 | 0 | X | X |

This is now used to create a circuit. The circuit may be simplified in any way, including taking common parts between the 4 bits so that those parts are not repeated.



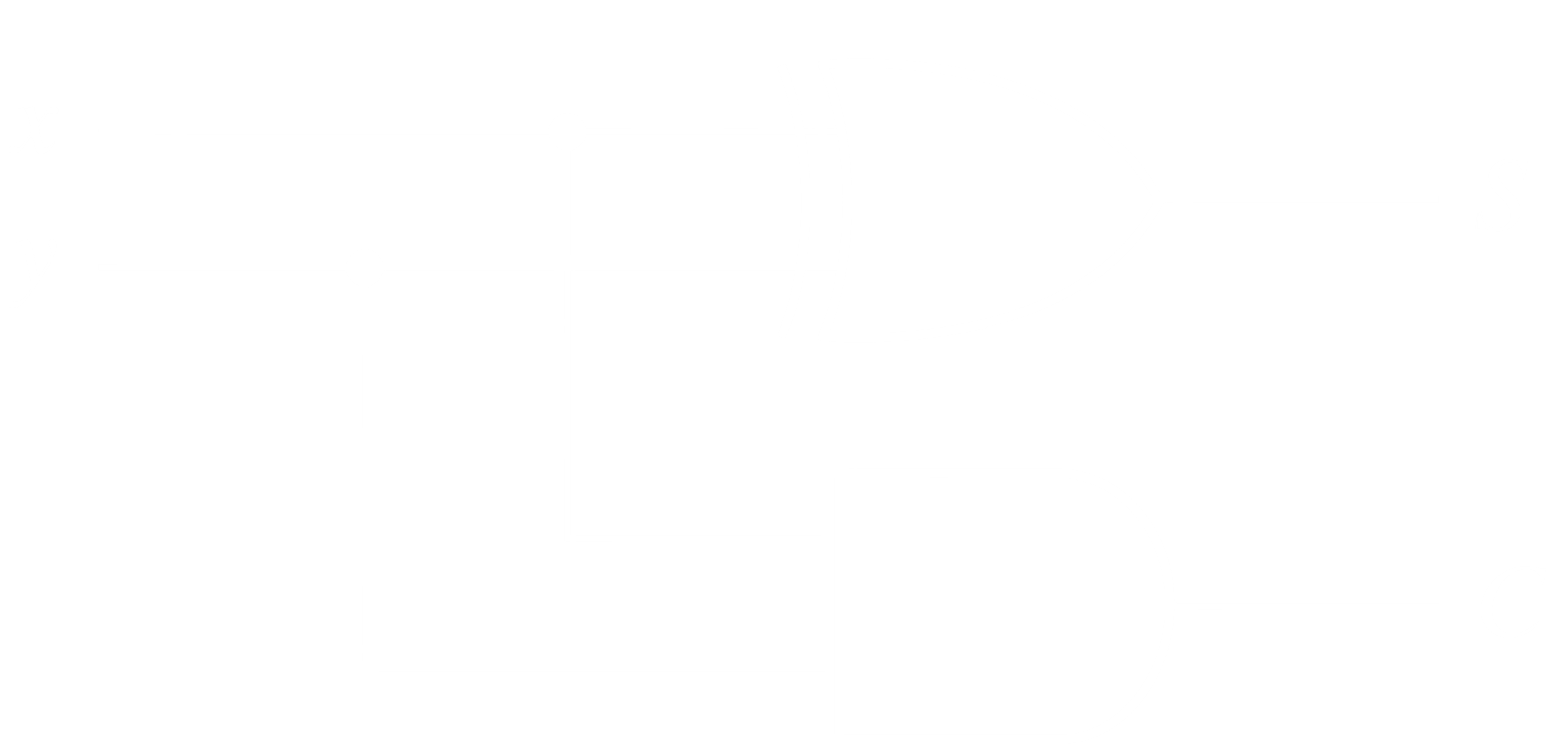
## Binary Adders

An adder performs addition between two binary values. They can be of 2 types, half-adders and full-adders.

A half-adder simply performs the addition of two inputs and , and gives the sum () and the carry () as output. It does not take into consideration if there was any carry from the previous addition, and thus it cannot be used to perform proper binary addition.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Thus, a half-adder is an XOR gate and an AND gate, each giving a single output, and respectively.



A full-adder is simply a combination of two half-adders (thus the naming). It takes into account the carry from the previous operation, and can thus be used to perform proper binary addition. It takes 3 inputs, , and , and gives 2 outputs, and .

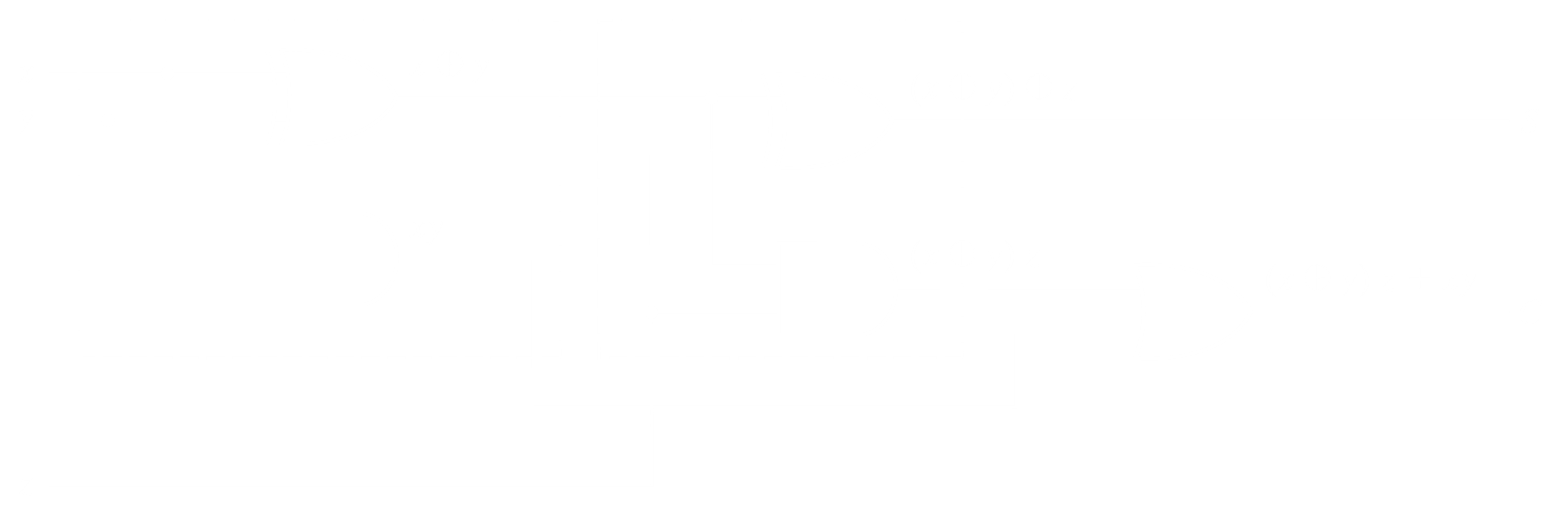
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

is only for an odd number of s, so .

For ,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 00 | 01 | 11 | 10 |
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |

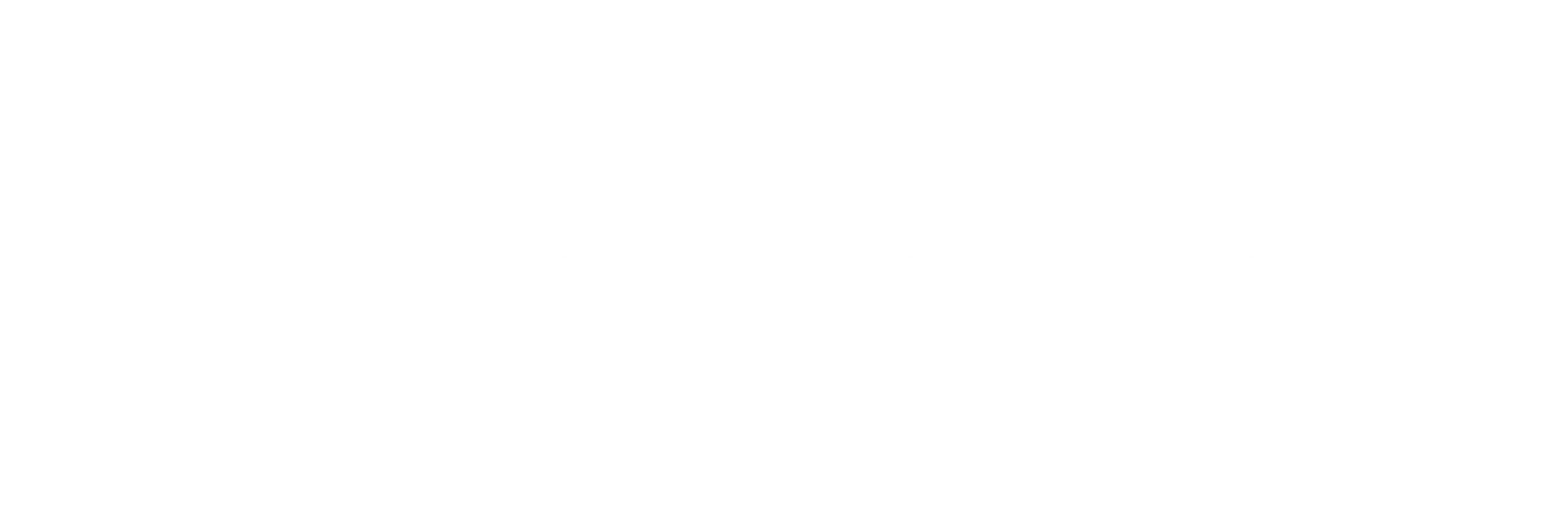
(This is valid and can be proven easily.)



Full-adders can be used to add two numbers.

Given two numbers, and , where each digit has a carry , thus giving us 9 input values, the result will be , each digit giving a carry - .

The arrangement of the circuit will look like this:

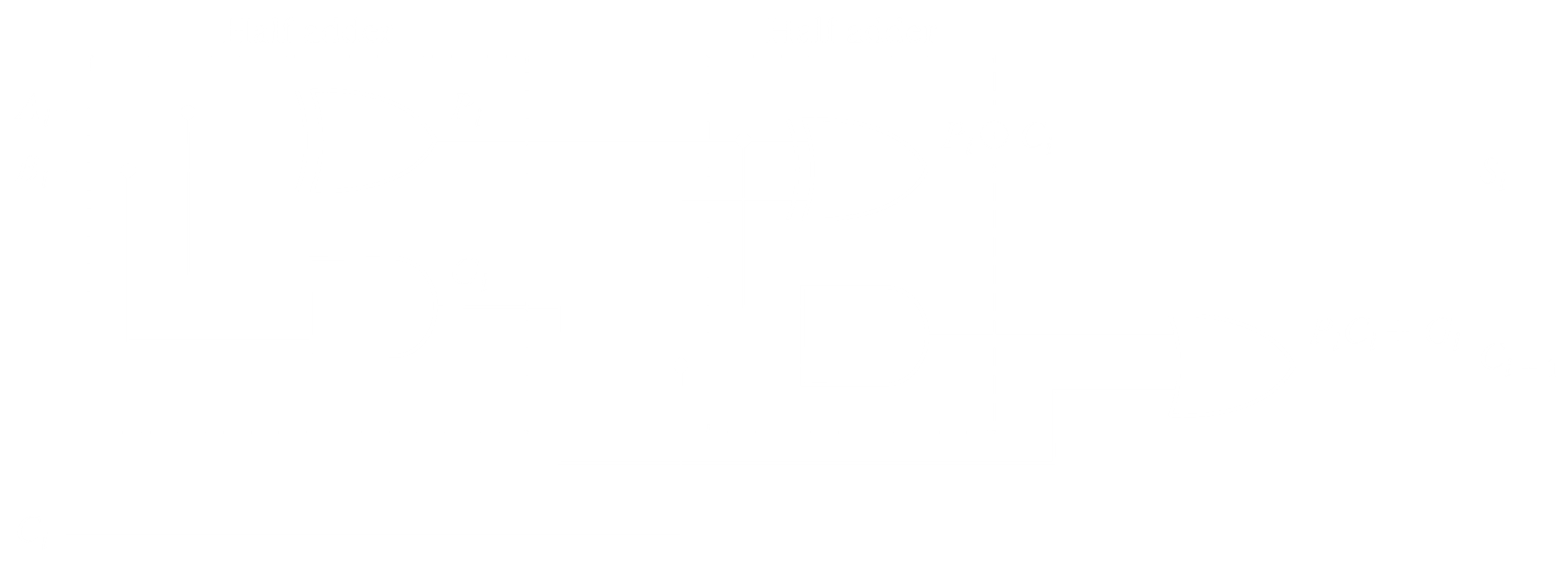


, , and each depend on the full-adder before it, which means each full-adder depends on the full-adder before it to function. This is called a Carry Ripple Adder.

If each adder takes time to complete an addition, the total system will require time to complete the addition. This is called the propagation period. Thus, if input is given at time , output will be received at time . There will be output before this time as well (since each required value of will be initially), but the results received will be incorrect.

### Carry Lookahead Generator

The propagation period can be reduced. From the truth table for a half-adder, it is seen that the carry for a particular addition is only when both values are . This can also be seen in the circuit for a full-adder, where an OR gate is added at the end where one of the input values is .

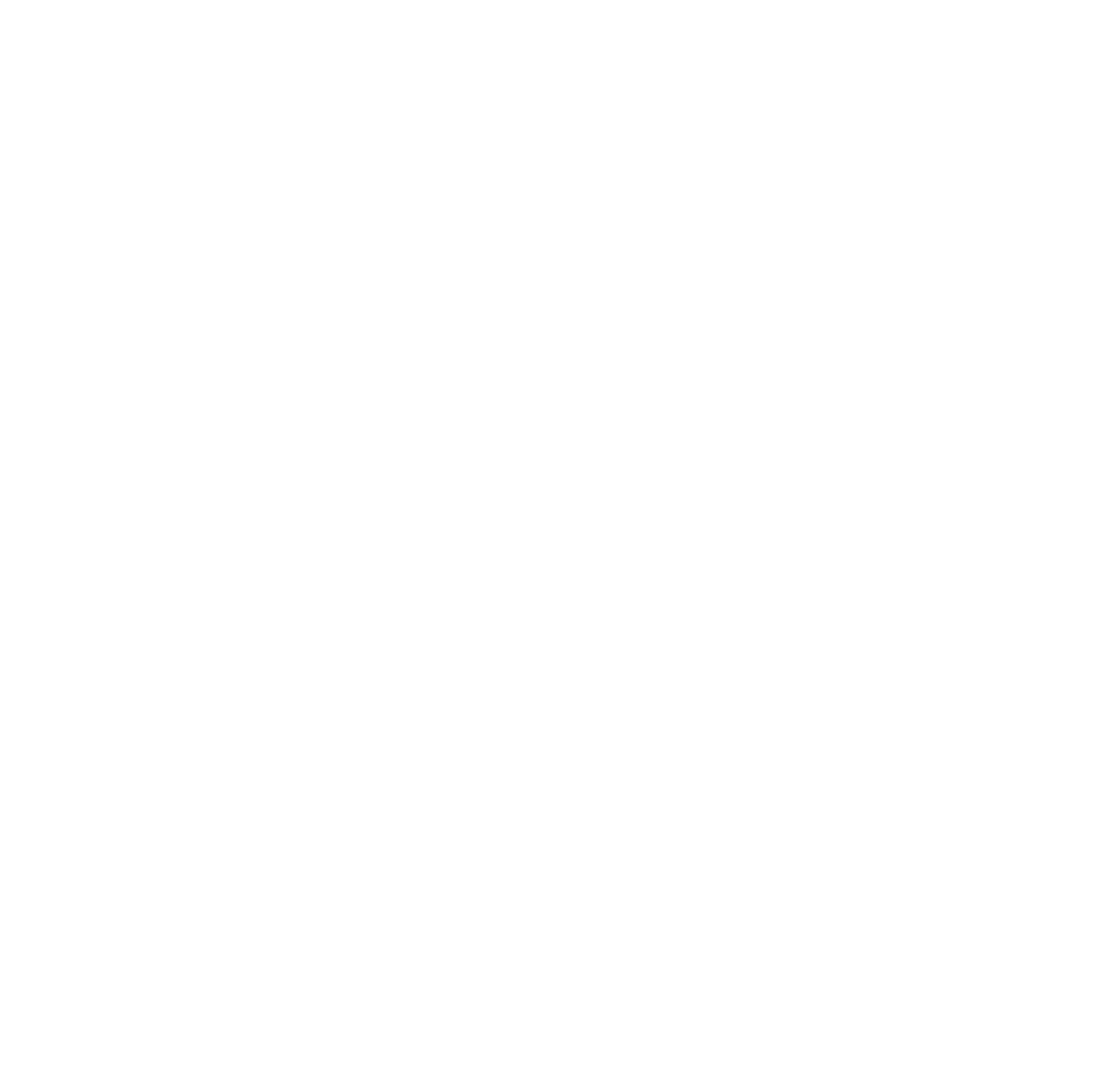


Here, and .

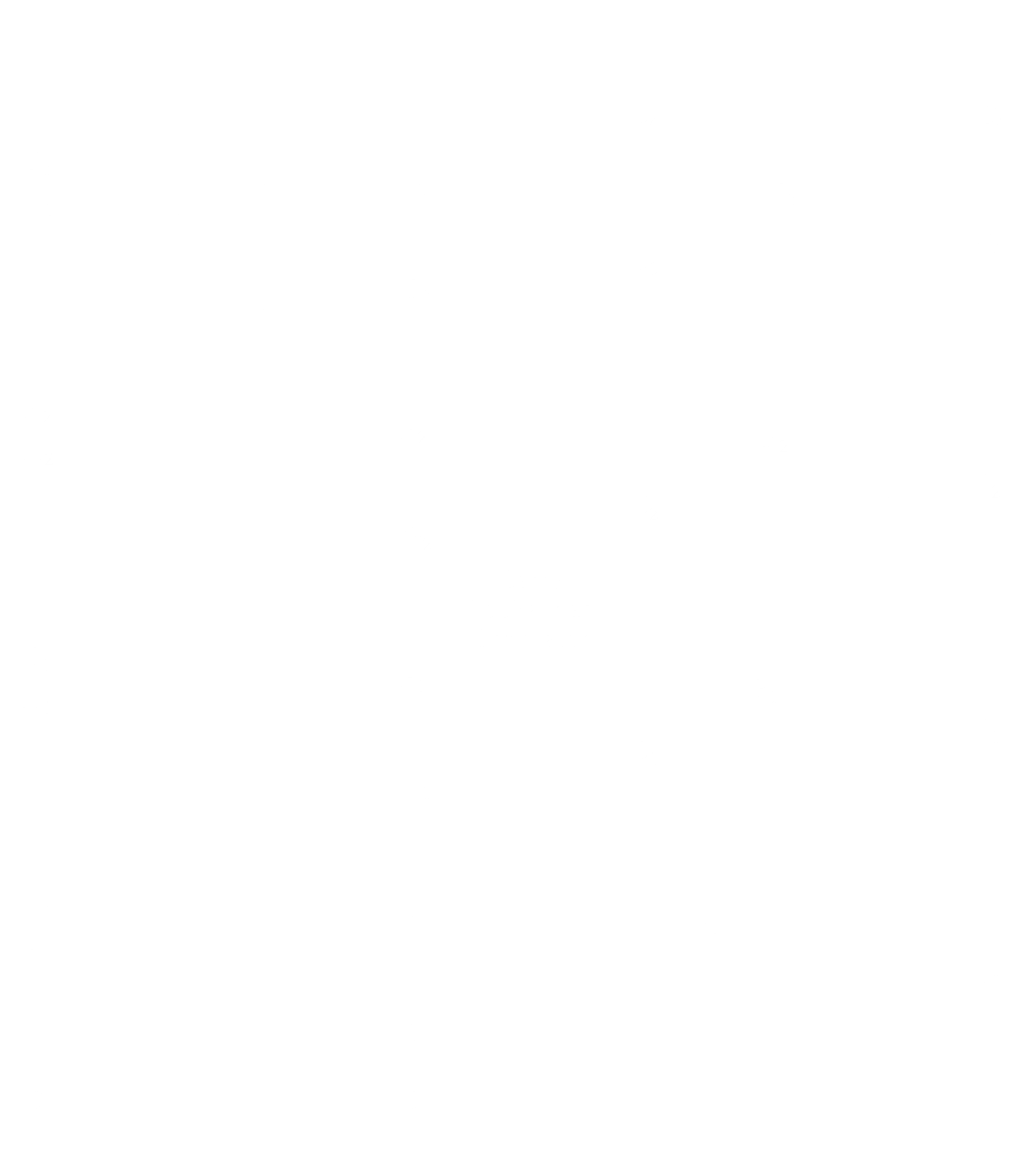
is known since it is an input.

and can also be written in the same manner, in terms of the 9 initial input values. This means that the carry for each full-adder can be found before the full-adder completes its addition, by setting up a separate circuit using the initial inputs. Thus, all 4 full-adders will be working at the same time, reducing the total propagation period to roughly . It will not be exactly , since the additional circuits will take some time to complete their functions.

Such a system is called Carry Lookahead Generator.



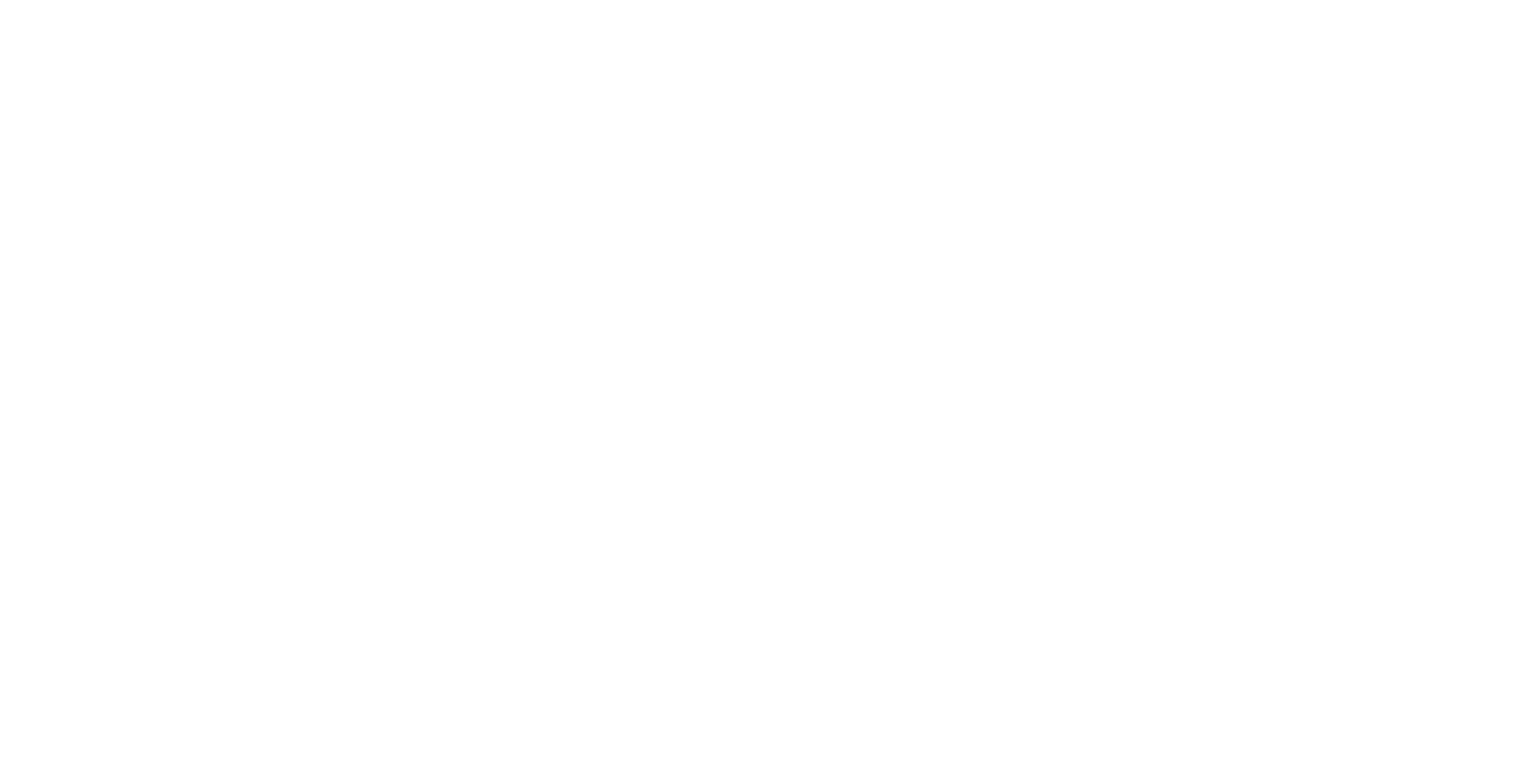
The generator is drawn as a block in the final circuit for the 4-bit binary adder.



### Subtractor

The subtraction is actually done by finding the 2’s complement of and adding it to using a 4-bit binary adder. The 2’s complement is found by inverting each of the digits of and adding .

However, a more effective approach is to use a 4-bit Adder-Subtractor. This is based on the concept that , so the inversion can be done if subtraction is needed, and removed if not needed.



If the value of is set to , is inverted, and the additional from creates the 2’s complement of . It is then added to to perform the subtraction.

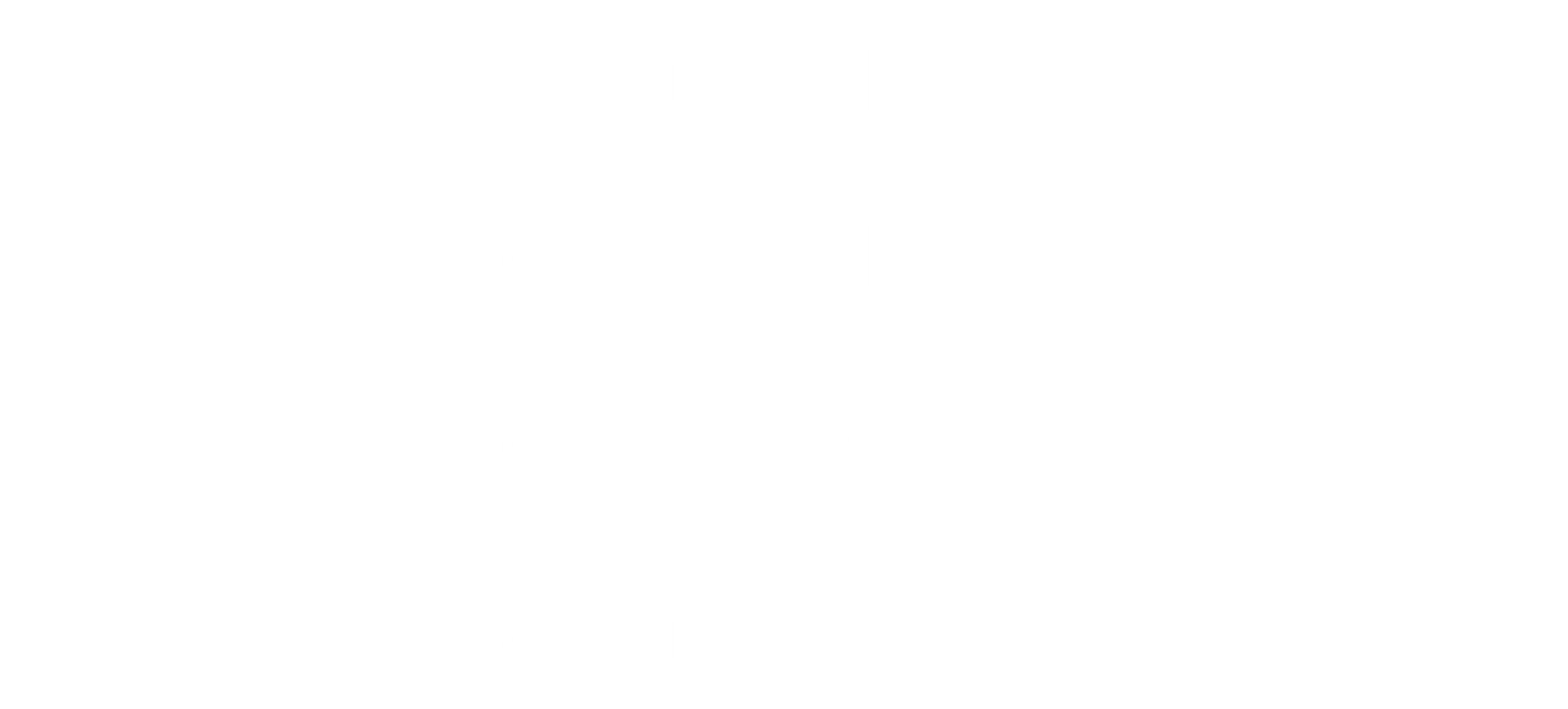
However, this in itself presents a problem. If sign magnitude is used to represent positive and negative values, incorrect values may occur if the result goes beyond the scope of the system. For example, 70+80 should give +150, but due to the effect of the carry, the sign magnitude will show 1 instead of 0. Similarly, -70-80 should give -150, thus 1 as the sign magnitude, but it gives 0. Both of these results are beyond the scope of the system.

Oddly, the last 2 carry values ( and for a 4-bit system) will both be , only if correct answers are obtained. If incorrect results are obtained, it will either show and or and . This fact is used to check whether the results obtained are correct. The XOR gate at the end takes and as inputs and gives an output . If the value of is , the results are incorrect.

### BCD Adder

Covered in Chapter 1, BCD code is the same as binary code, except that it only considers values from to . From onwards, the numbers can be represented by those same digits. So, for example, , which in binary would be , is represented as in BCD, literally and then .

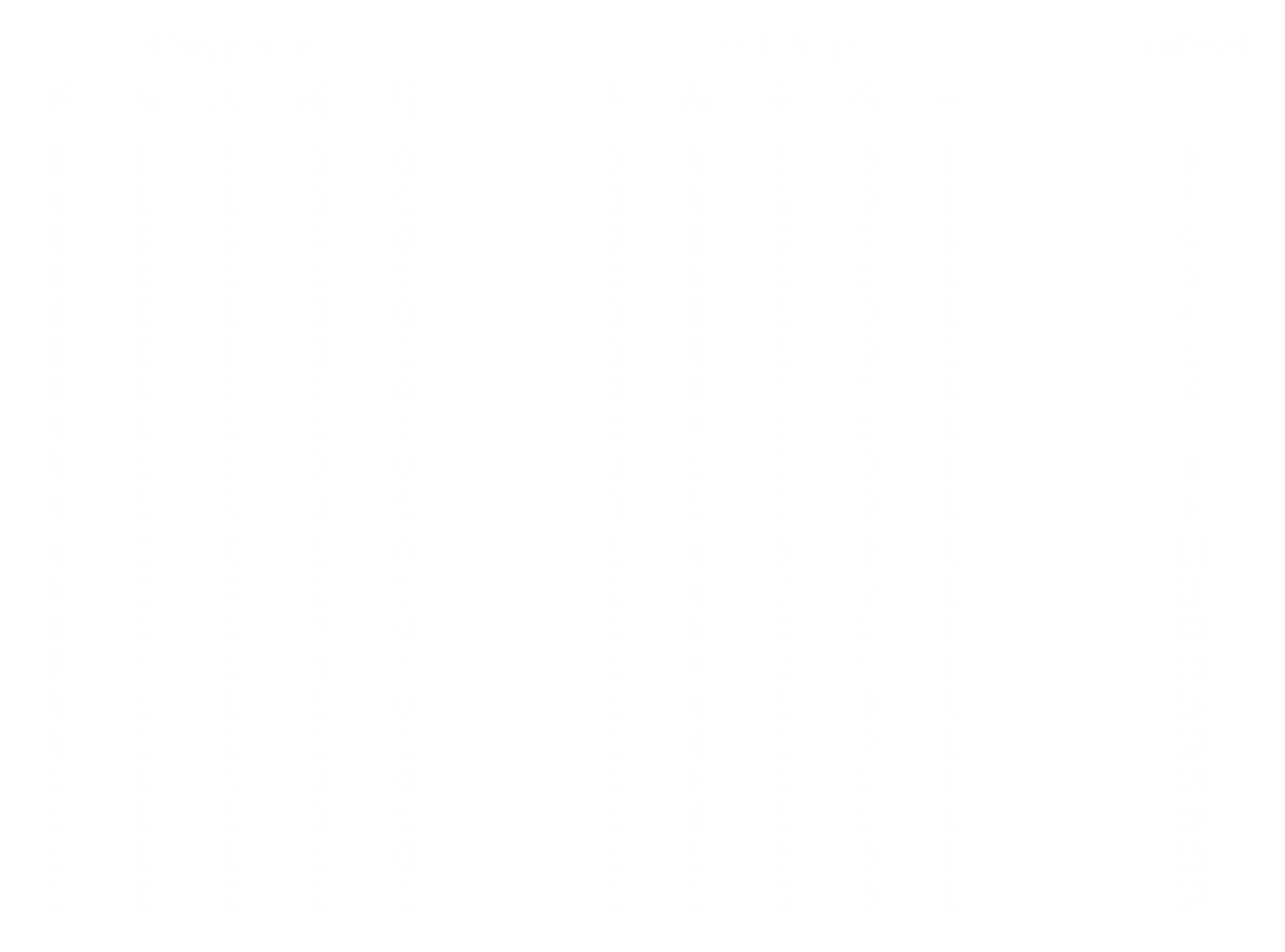
This method causes a problem when considering addition in BCD. Adding and , resulting in is fine since the result is still a single digit so its binary and BCD representations are the same. The addition can be done with a simple 4-bit binary adder. With results that are greater than however, a single binary adder will not give an answer in BCD form. For example, adding and resulting in will be represented in binary as , whereas its BCD representation is . Luckily, for results greater than , adding (i.e. ) to the result gives us the BCD representation of the result.



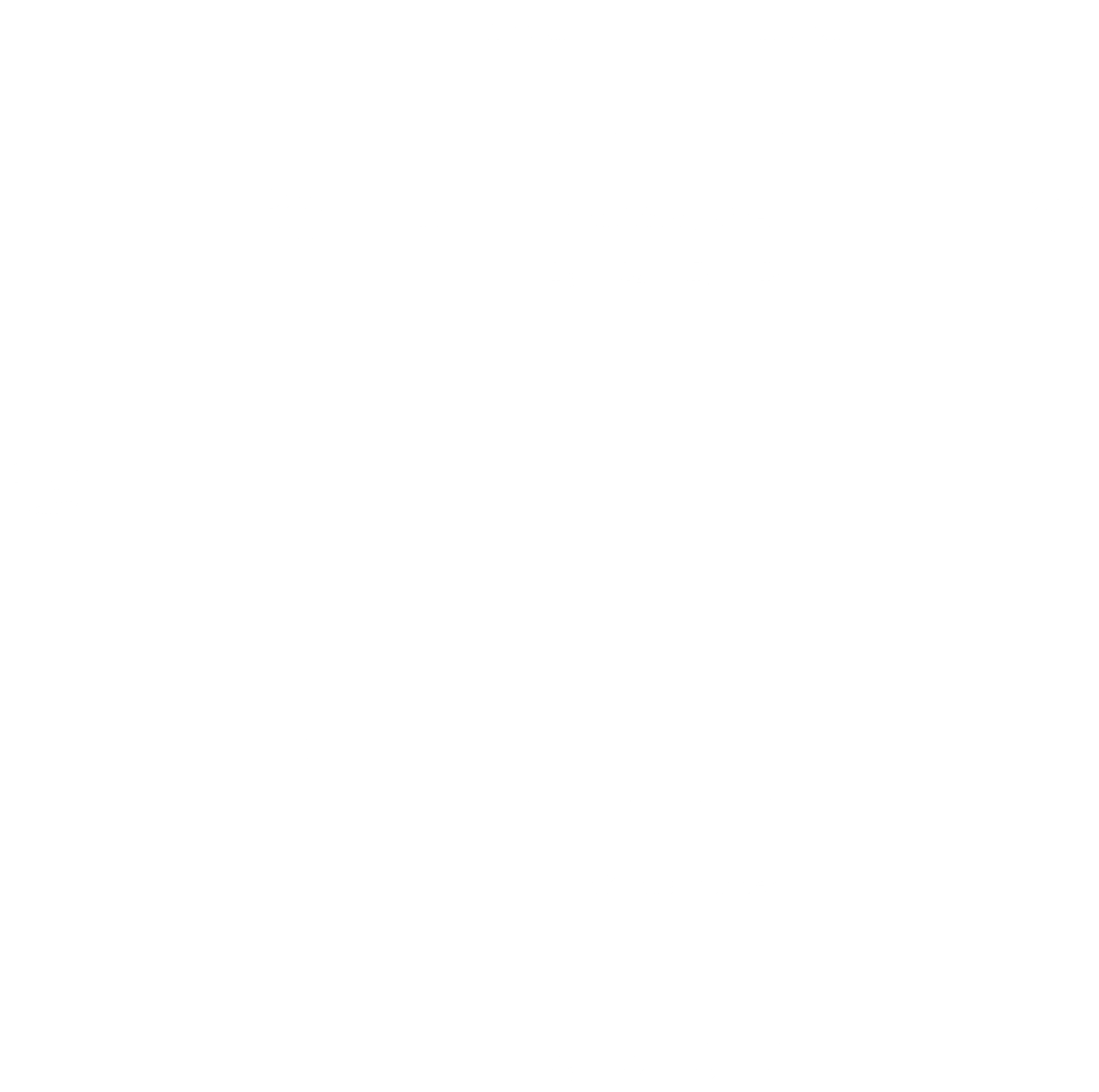
This is true for all values that give results greater than .

This can be used to fix the circuit. We simply need to use an additional adder to add to the results whenever the results exceed .

The following is the truth table for binary to BCD conversion. BCD addition is only valid for values between to , so the results can only be between to . Considered any previous carry, the maximum result is , so there are possible results. Among these, the values to are greater than , and we must add only in these scenarios.



Here, is used to represent for binary addition, so for results greater than , and is the output carry for the BCD adder, which is for results greater than . We see that results exceed (i.e. ) under 3 conditions, if , if and or if and . So, we simply need to design a smaller circuit after the initial adder that checks these conditions, and adds to the results if .

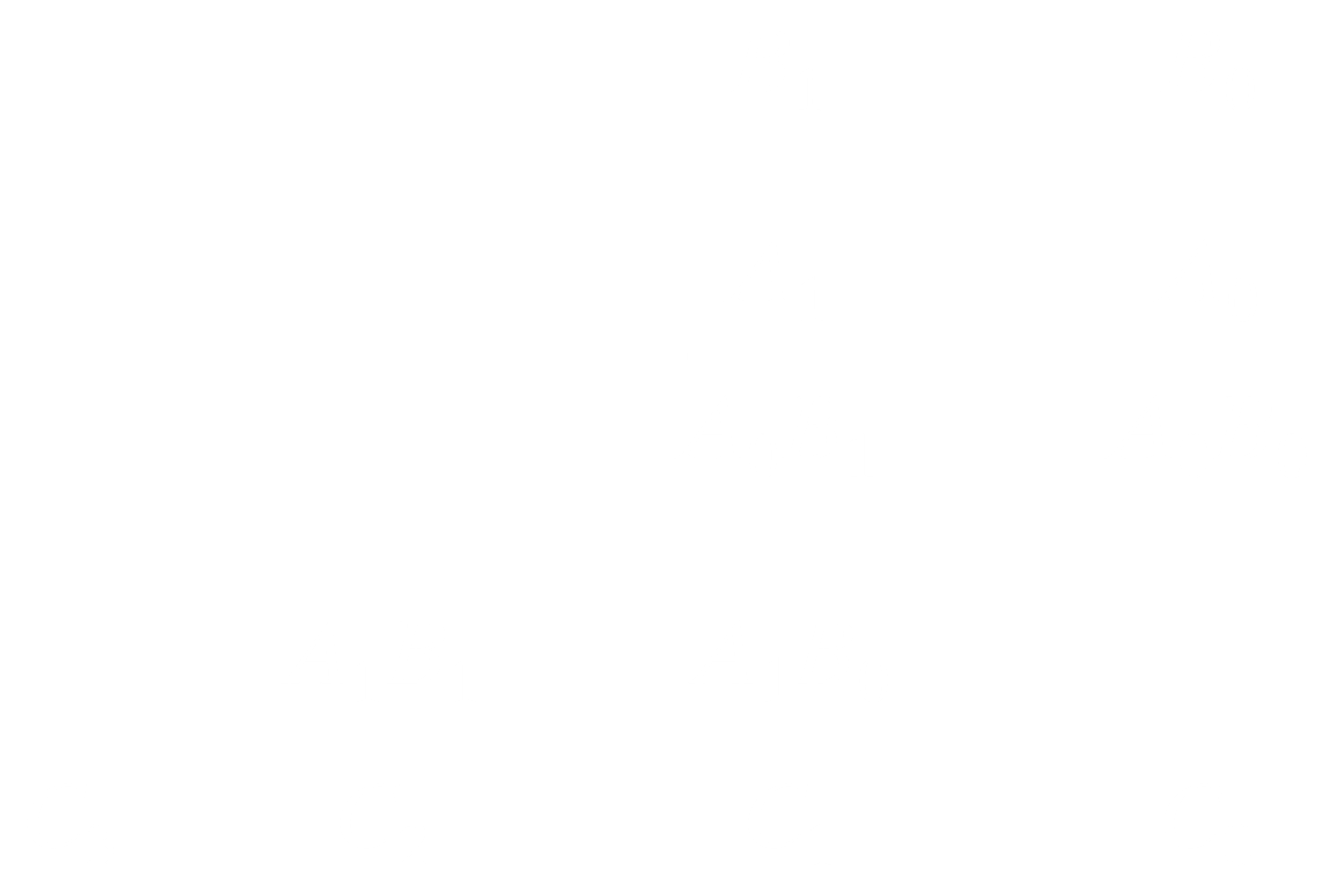


The two AND gates along with the OR gate are added after the first adder to check these conditions. If any of them are true, the output carry, . This then goes to the second adder which adds it the initial result, giving 4 values of . These 4 values, along with the value of , is the final result in BCD.

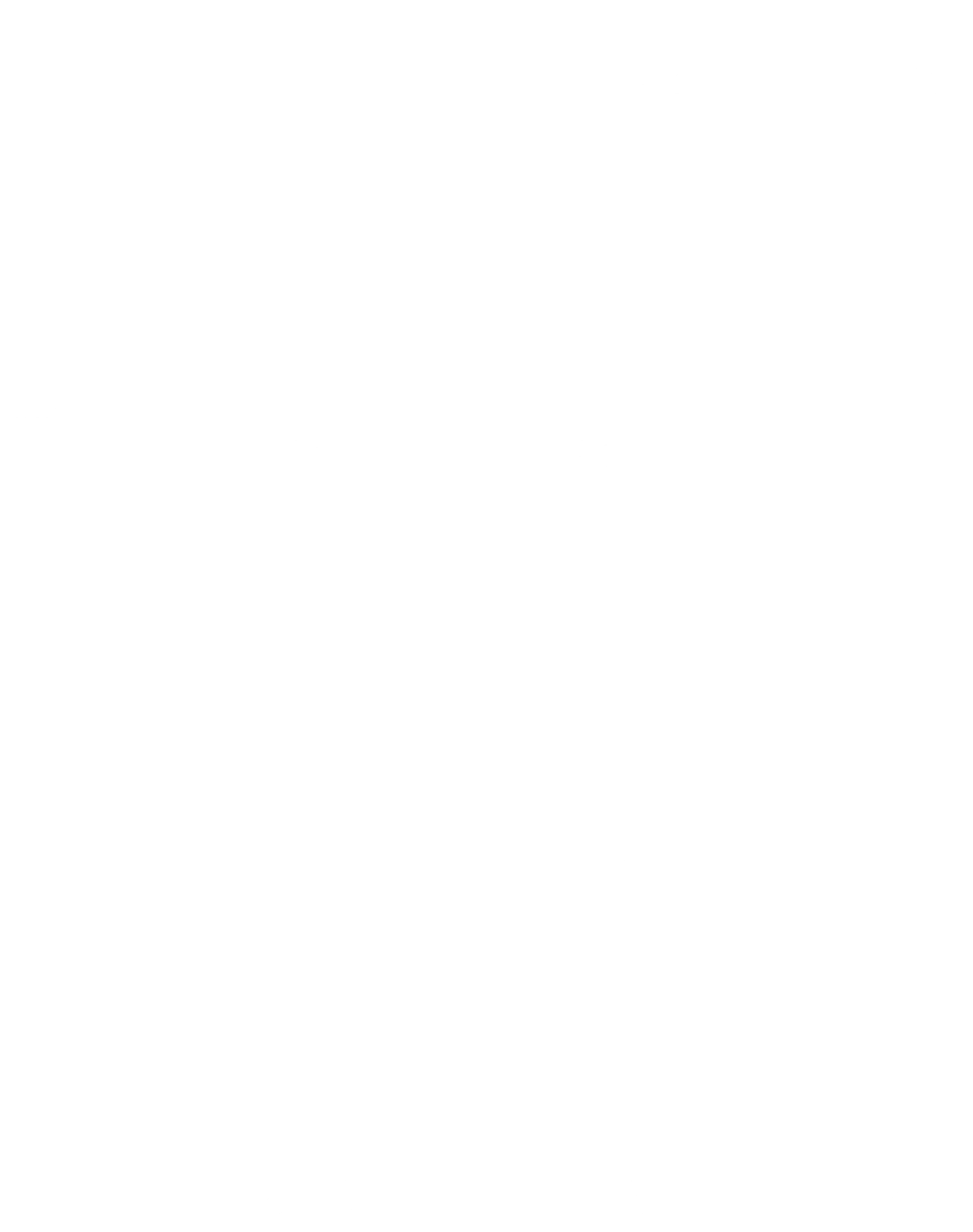
Note that if , the second adder adds to the initial result, so that they remain unchanged, as they should.

### Binary Multiplier

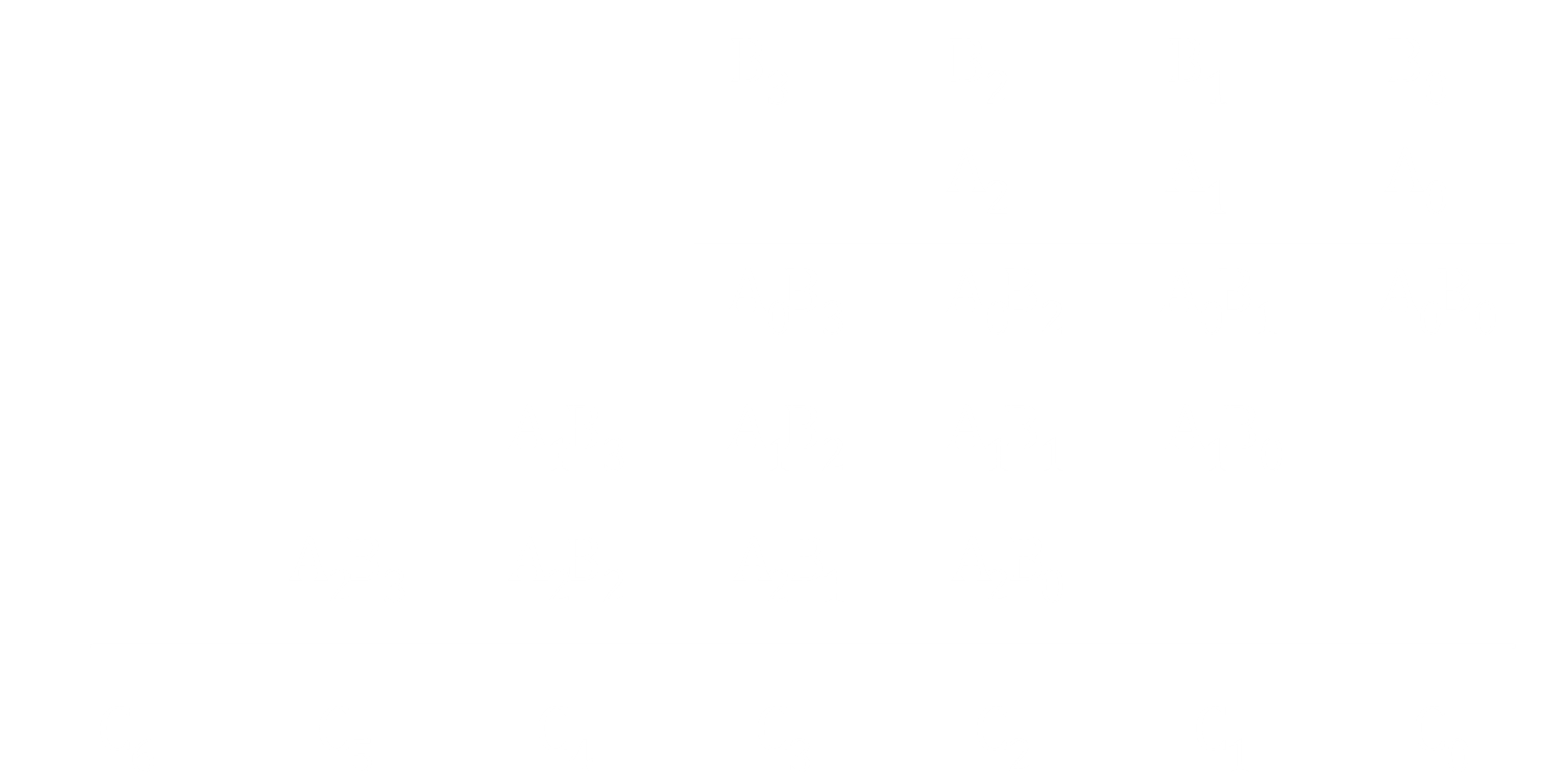
Binary multiplication is done in the same way as normal multiplication. In the addition stage, the carry of each addition must be carried to the next addition.



The additions are done by adders. For the -bit addition done above, half adders are used. Each half adder gives two outputs, one for the result of that particular addition, and another for the carry that is taken to the next addition.

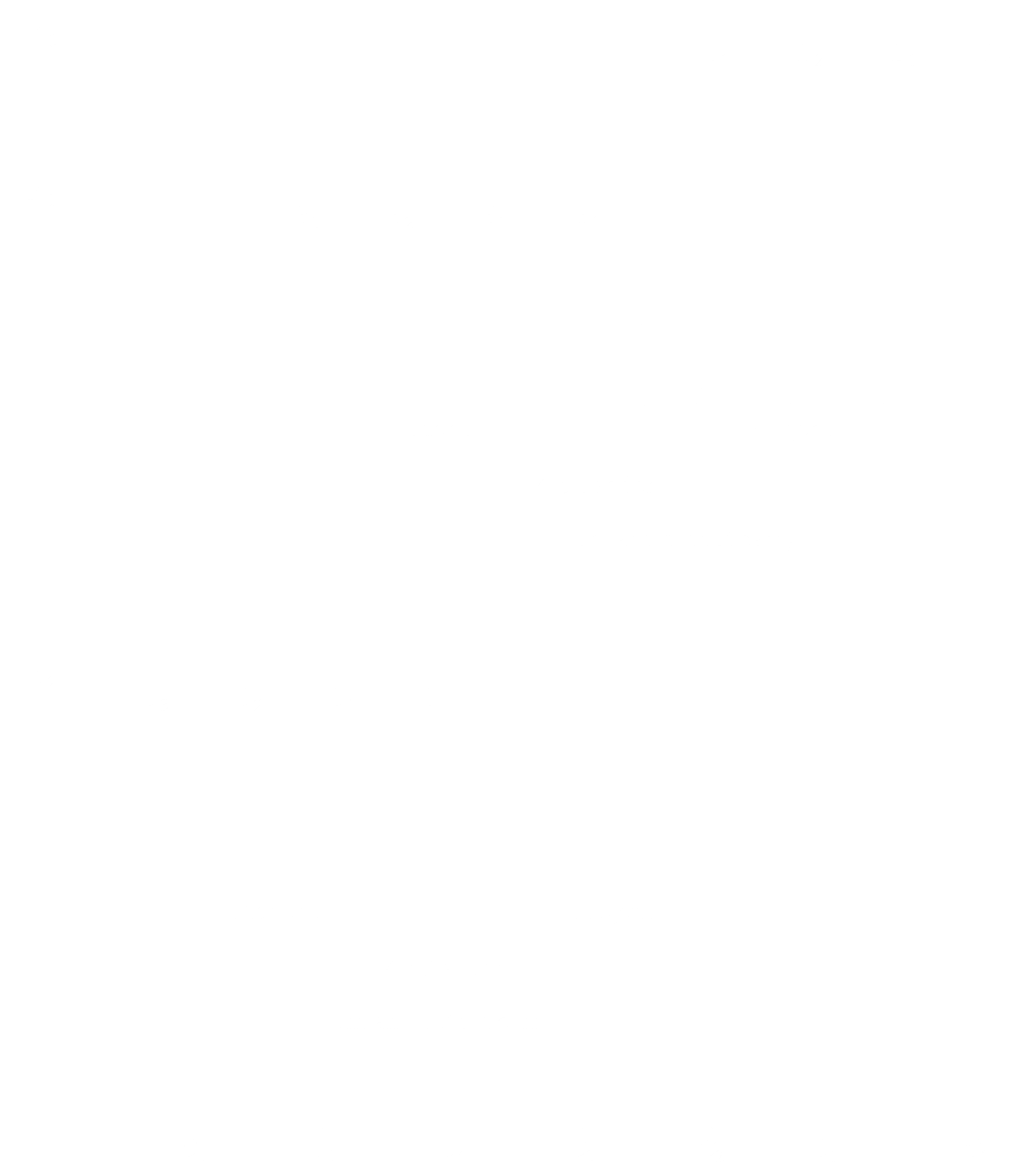


Multiplying a -bit number to a -bit number looks like this:



Each stage of addition is done by a single adder. For example, in the first step, is taken directly as , and the binary number is added with

using a single adder. This adder must be a -bit adder. The first result for this addition (i.e. ) is taken directly as and the rest of the results are added to the next stage with another -bit adder. The following is the circuit for the addition:



Note the extra that had to be added to the first adder. This was the carry from the previous addition, which was since there was no previous addition. This does not occur in the next adders since the actual carry from the previous addition is taken instead.

The multiplication of two numbers consists of the multiplicand (here ), which is of bits, and the multiplier (here ), which is of bits. For a multiplication, we need adders (notice that the above multiplication had adders), each of bits (in the example above they were -bit adders). We will also need AND gates (above, we need AND gates for each step since was a -bit number, and there were steps in total since was a bit number). The result will be of bits.

## Magnitude Comparator

A magnitude comparator compares two numbers and and determines whether , or . The circuit for two -bit numbers has possible values in the truth table, so designing the circuit from the truth table is bothersome. However, the comparator circuit is very repetitive, so the circuit can be designed using an algorithm instead.

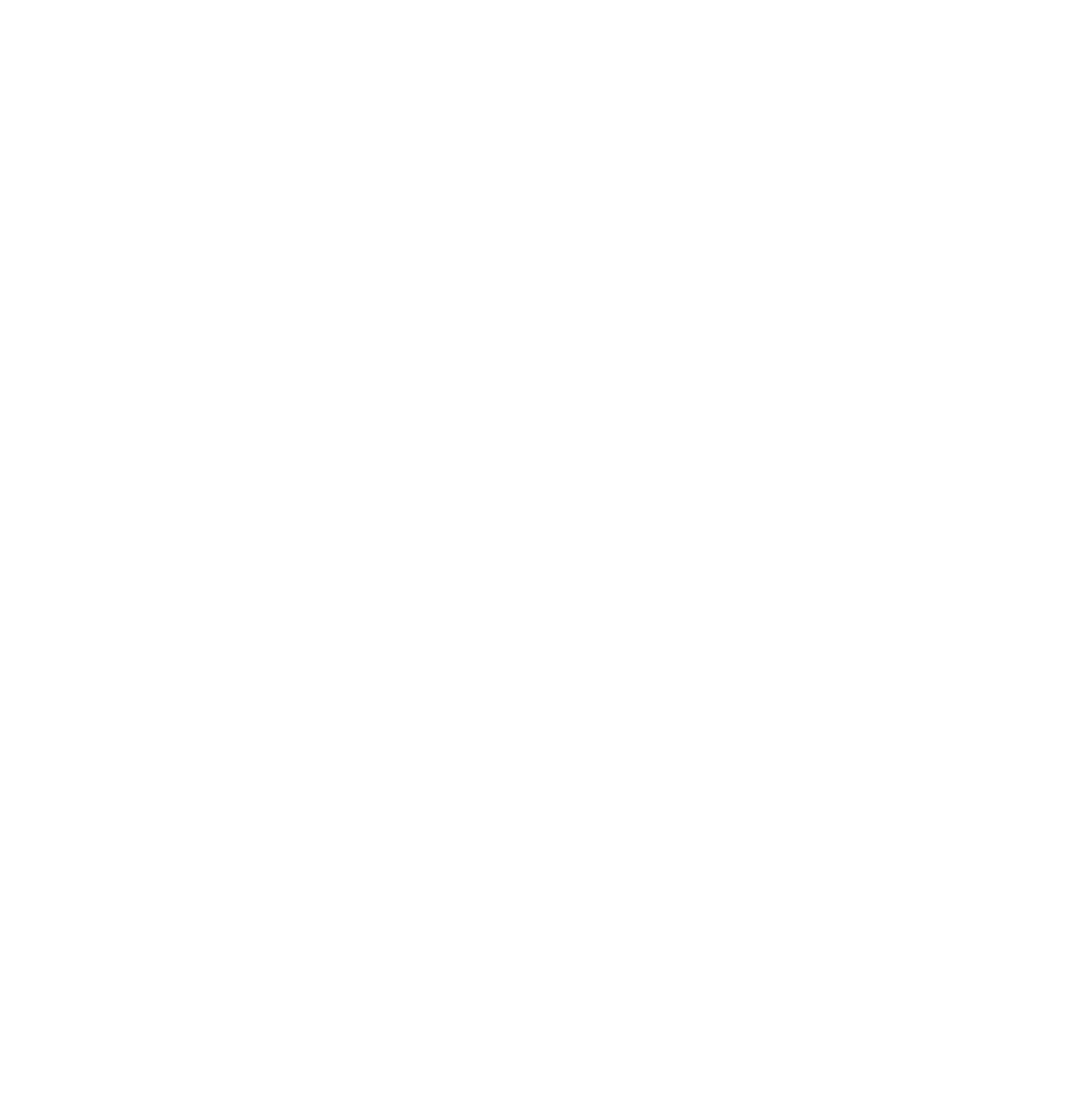
For two -bit binary numbers and , if for all values of from to . Since there are only two possible values or ,

only when both and have the same value. Thus,

If , for some value of , . For binary numbers this simply means while or . To check for , we have to go through each set of and , and check if they match this condition. If they match the condition of equality instead, we move onto the next set. Thus,

For , we check the same condition in the opposite sense. Thus,

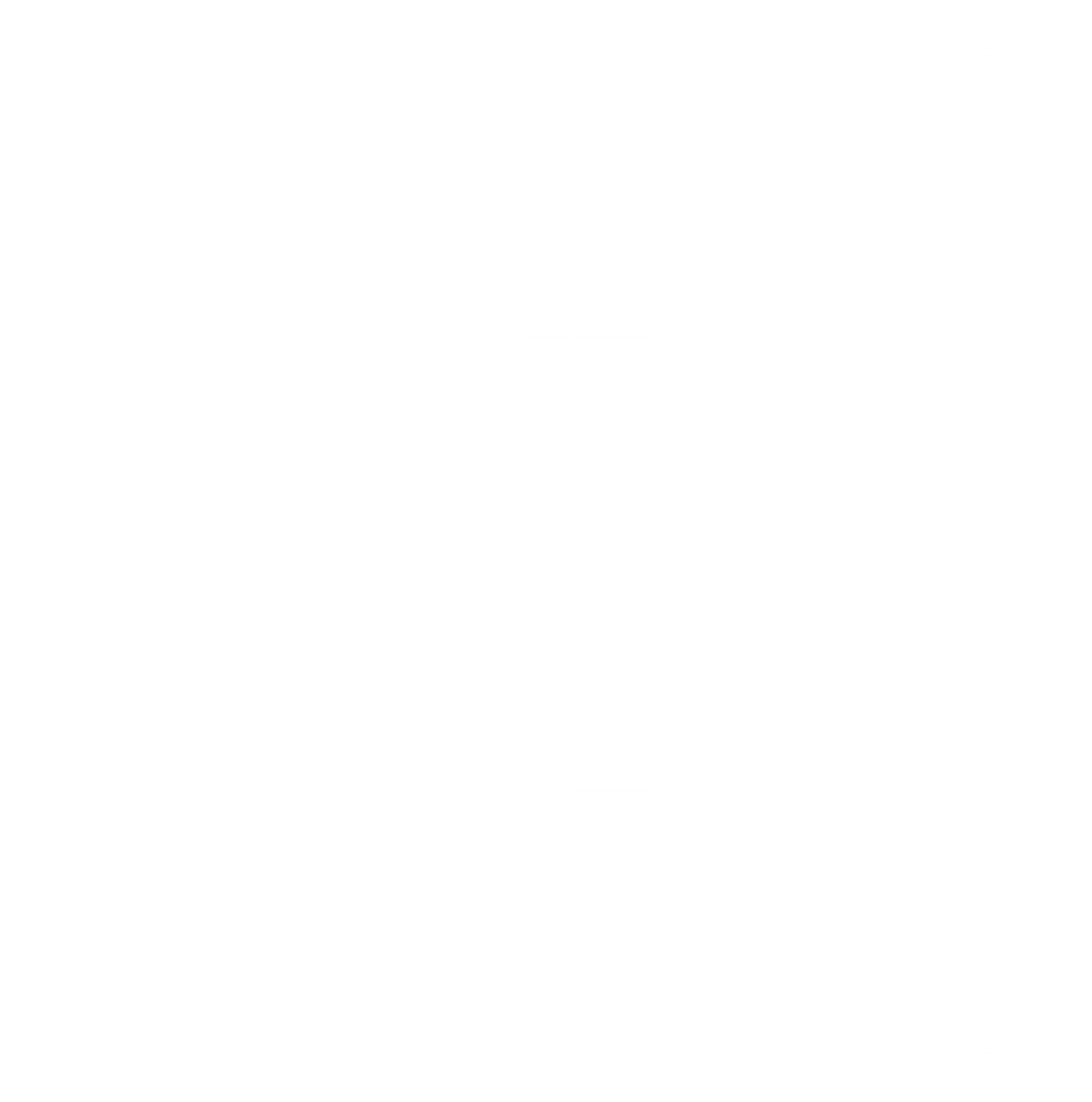
Binary numbers of any bits can be compared in the same pattern. The circuit for the -bit comparator, although complicated, is very repetitive.



## Decoders

3 separate inputs and can give a total of possible distinct outputs. Each of these outputs can be made to perform a separate action, meaning each combination of inputs will give a separate output. For example, pressing one button on a machine can cause the input , which will cause one action, while pressing another will cause the input , which will cause another action. A decoder converts these input lines into a maximum of output lines. There may be fewer outputs.

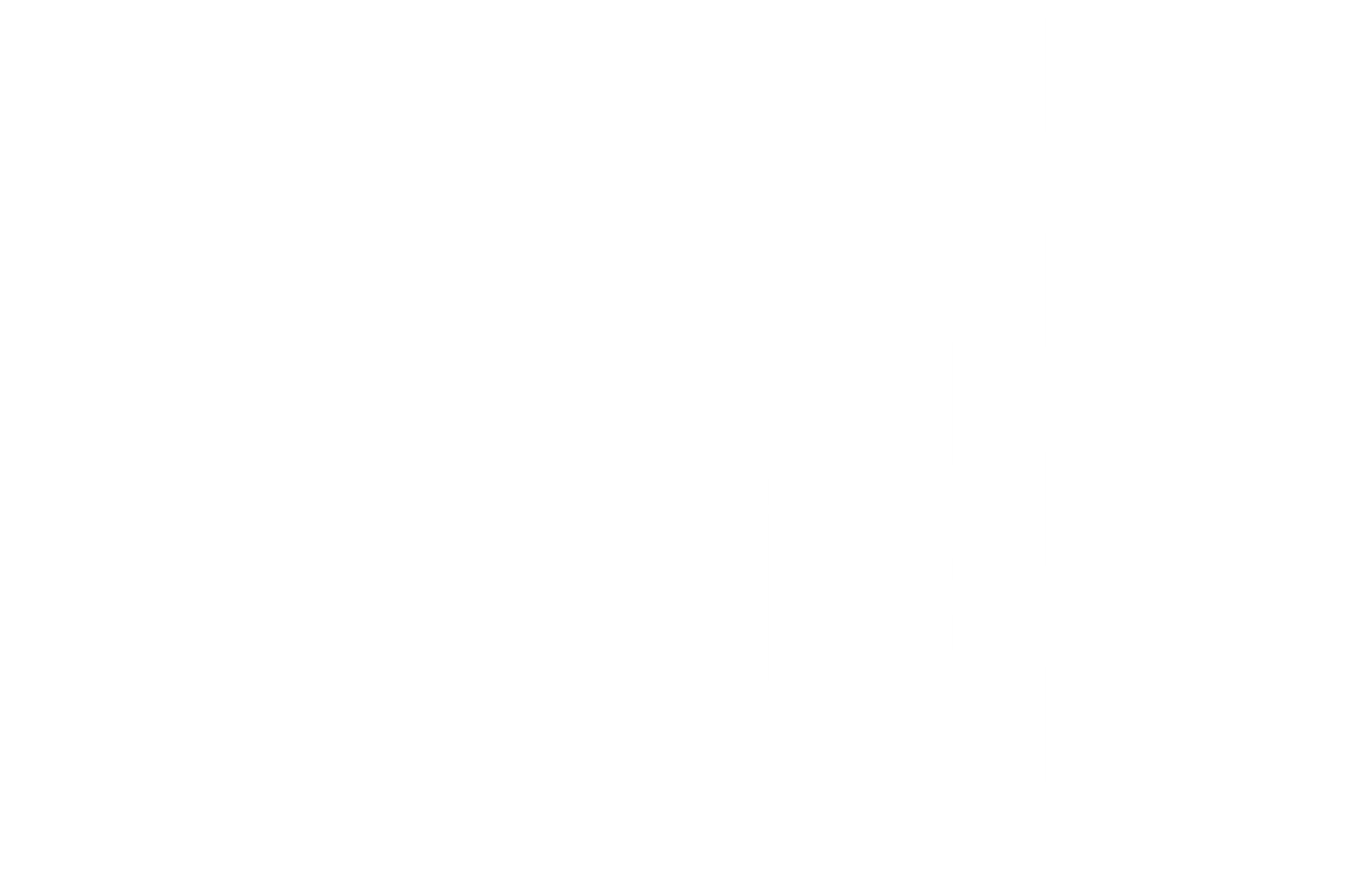
Decoders are referred to as -to--line decoders. The following is a -to--line decoder:



This can be used as a binary to octal converter. The input variables represent a binary number, and the output represents the eight digits of the number in the octal system.

### Decoder’s with NAND Gates

It is also possible to implement decoders using NAND gates. This makes it more economical. They also make it possible to include enable inputs to control how the circuit operates. The following is a -to--line decoder:

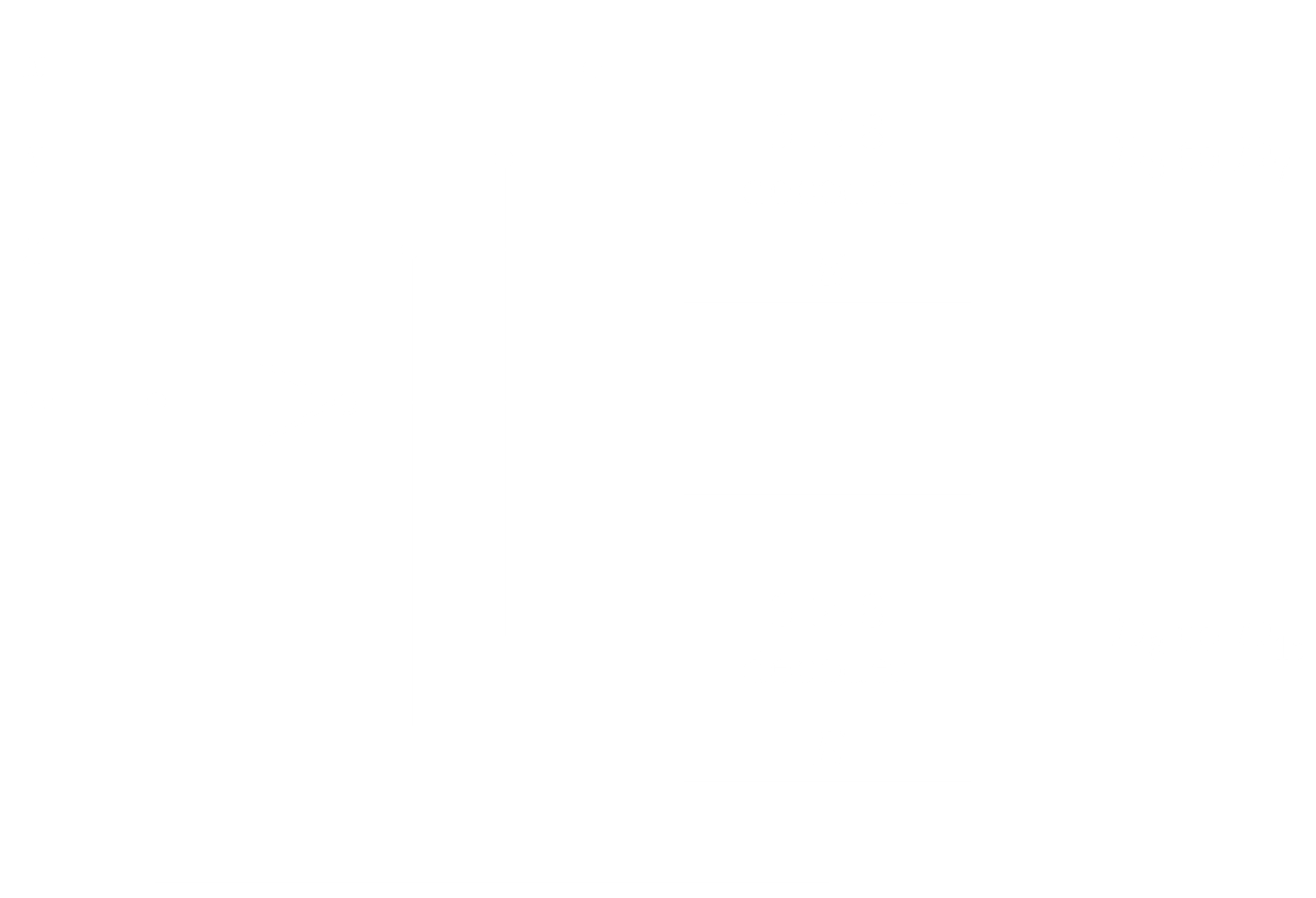


Using NAND gates, the output from the correct minterm will be instead of , so the result must look for s. For example, if and , the correct output should be from . The corresponding NAND gate will receive inputs and from and , and will invert it to after performing the AND operation. All the other gates will receive one or both inputs as , and will thus AND them and invert it into . This is called the active-low.

The enable function works in a similar manner. If , it gets inverted and each AND gates receive as input from that line. This means that the previously stated operations will work correctly. When , all the AND gates will receive an input of from that line, thus giving as their outputs. The circuit is essentially disabled. Thus, can work as a switch.

### Combining Decoders

Decoders with enable inputs can be connected together to form larger decoders. The following shows two -to--line decoders with enable inputs forming a -to--line decoder.

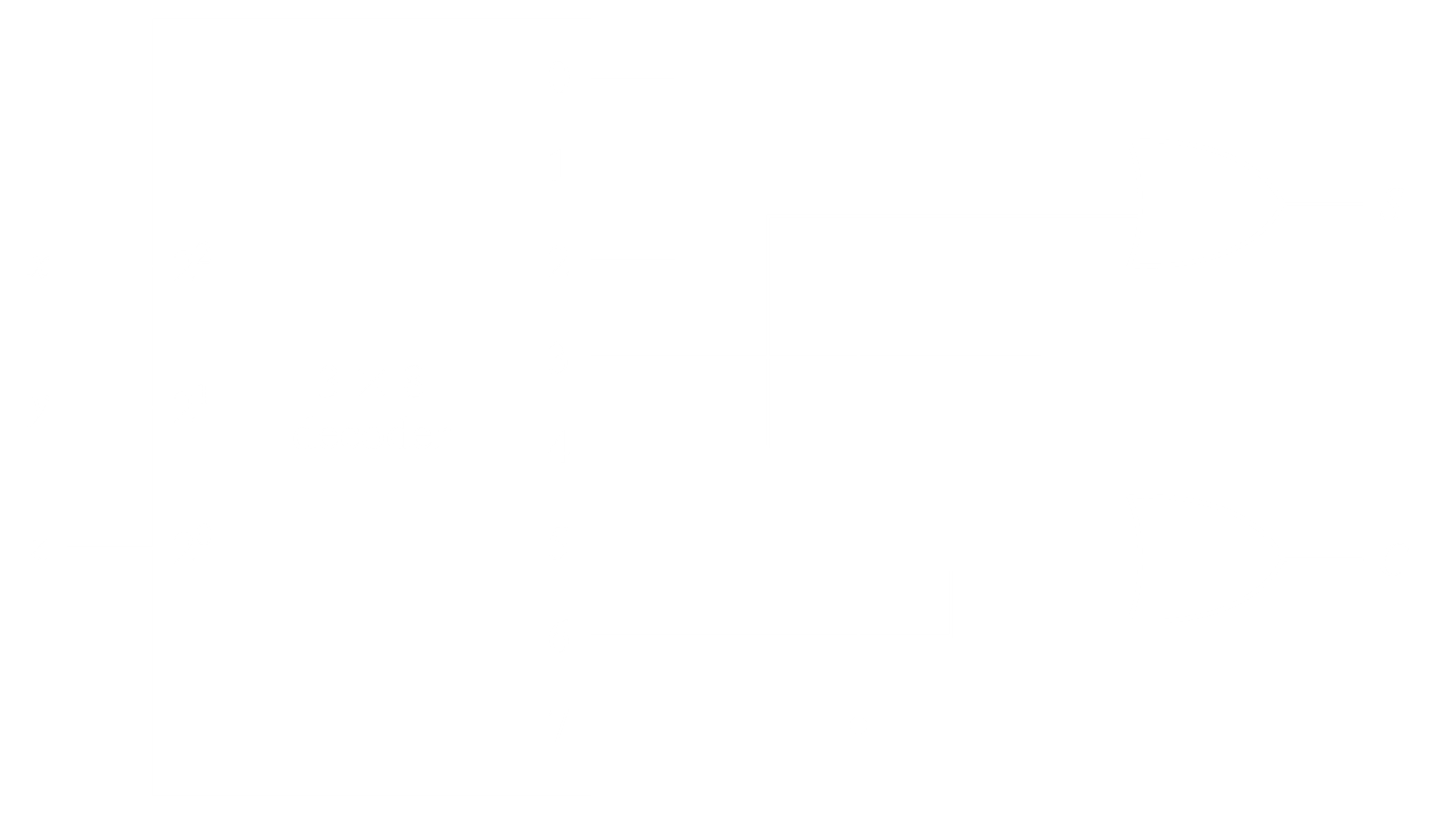


When , the top decoder is enabled, which covers values to . When , the bottom decoder is enabled, which covers values to .

### Combinational Logic Implementation with Decoders

Any Boolean function can be expressed in sum-of-minterms form, and since a decoder gives the outputs for all possible combinations of inputs, it can be used, along with an OR gate, to create any combinational logic circuit.

For example, a full adder with input variables can have possible outputs. Here, and . Thus, using a -to--line decoder along with two OR gates, we get the following circuit:

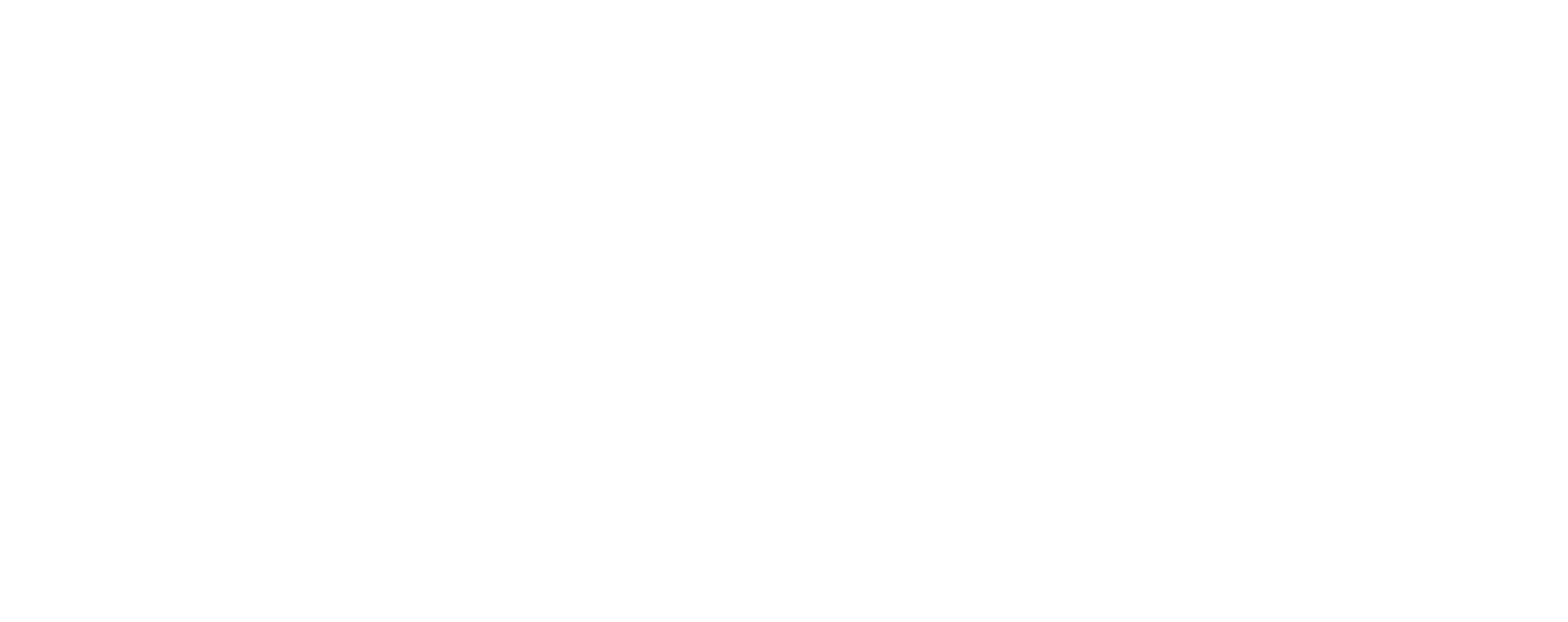


If NAND gates are used inside the decoder, the external gate must also be a NAND gate instead of an OR gate. This is because such a decoder would give the complemented output for the function.

If, for a particular function , we find that has a larger number of minterms that , it is advantageous to take the outputs for instead and invert them, thus using a NOR gate instead of an OR gate.

## Encoders

An encoder performs the opposite function of a decoder. It takes inputs, and gives a maximum of outputs. The output lines combine to generate the binary code corresponding to the input value. The following is the truth table for an octal-to-binary encoder:



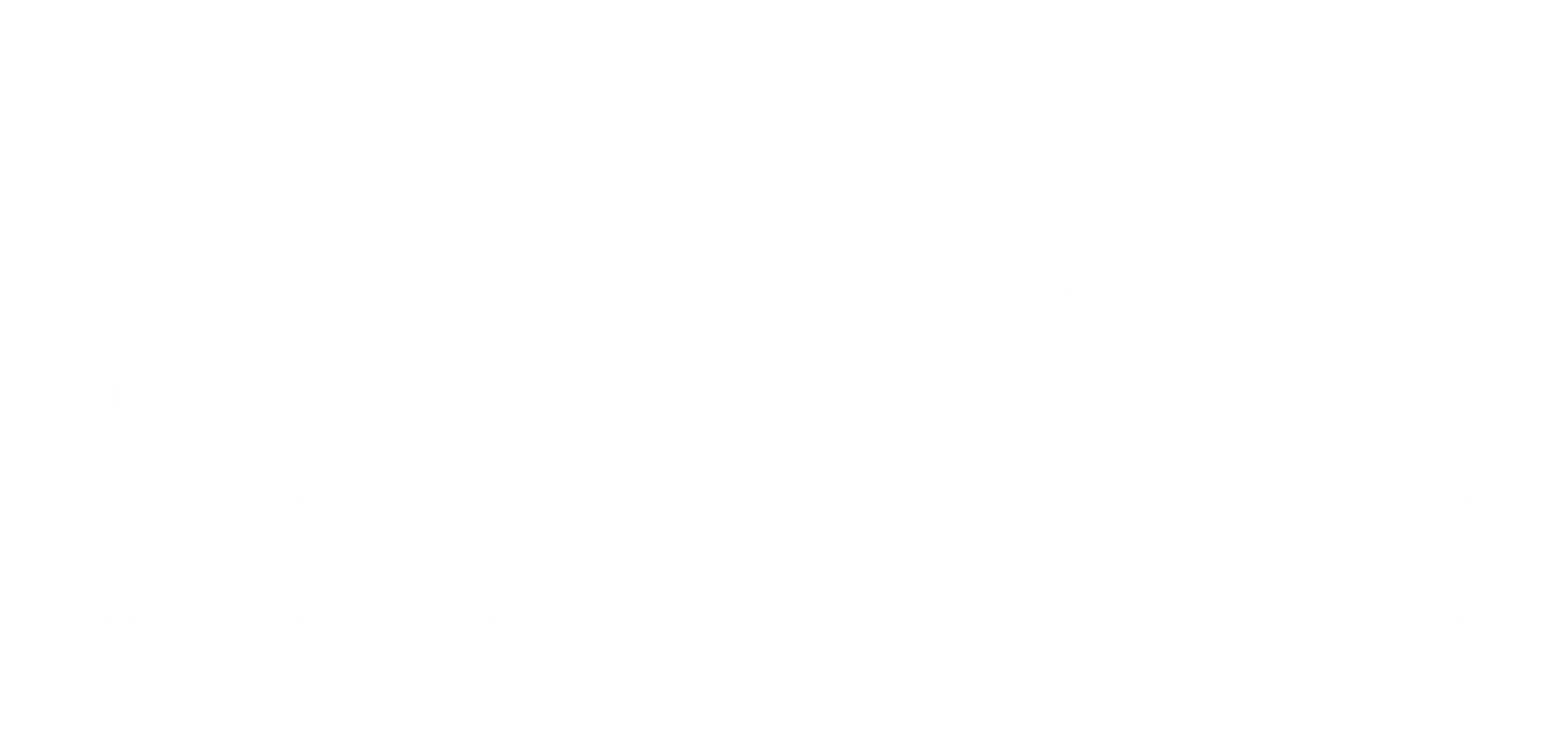
The encoder can be implemented with 3 OR gates. For example, whenever , , or is equal to .

Such an encoder has two major limitations. The first limitation is that for both the scenarios where and when there is no input, the output is . There is no way to distinguish between the two. Secondly, the encoder can only take one input at a time. If two inputs are given, the result is undefined. For example, if and are both , the output is , which represents neither nor .

The first limitation can be fixed by simply an extra output , called the valid bit indicator, that checks if any of the inputs is . If , the other outputs are not checked and are said to be don’t cares. The second limitation is fixed by using a priority encoder.

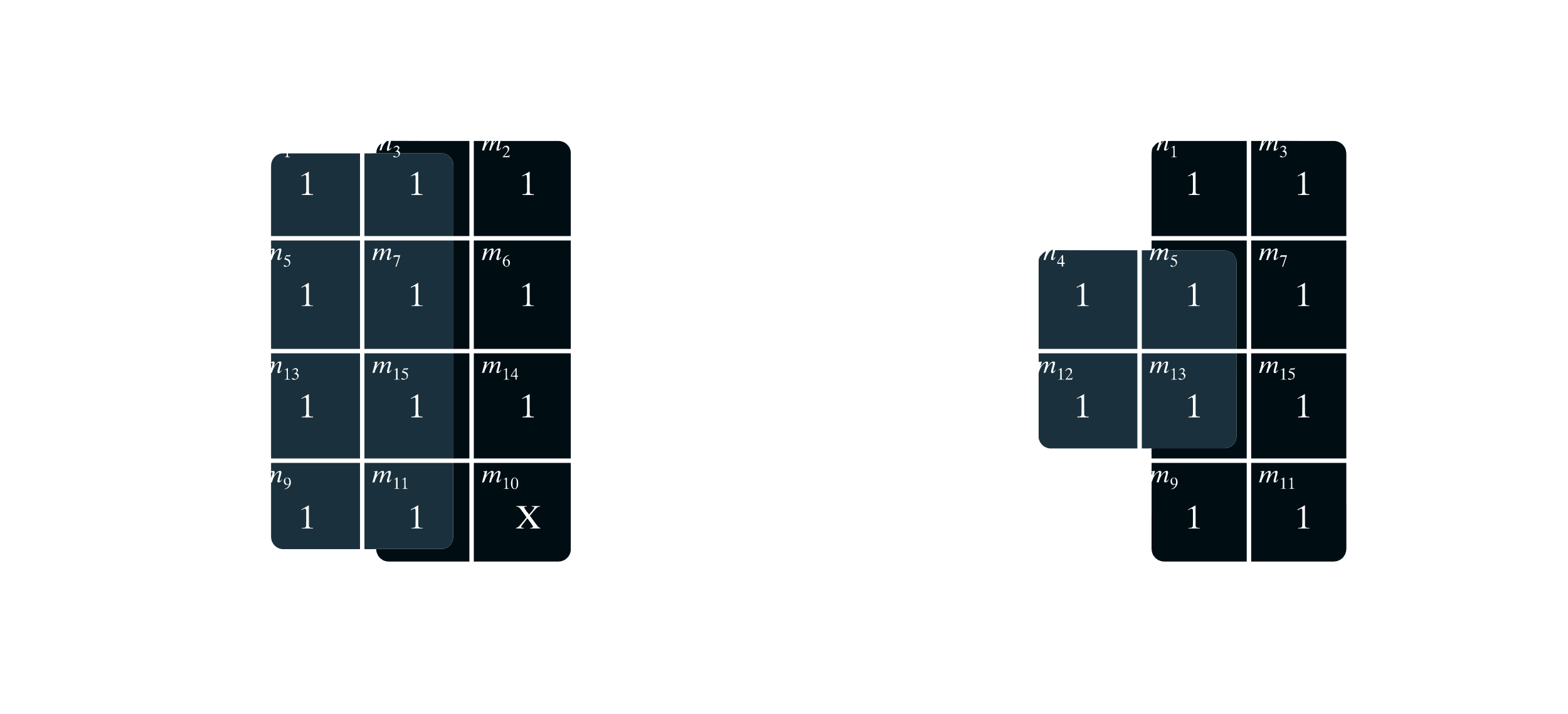
### Priority Encoders

A priority encoder checks the order of priority in scenarios where there are multiple inputs, and gives an output accordingly. The priority order can be set manually by the user, but generally follows the order of subscript numbers. The following is the truth table for a -input priority encoder. Note that the ’s in the output columns indicate don’t care conditions, but the ’s in the input column are used to condense the truth table. For example, represents , , and . Thus, all minterms are covered.

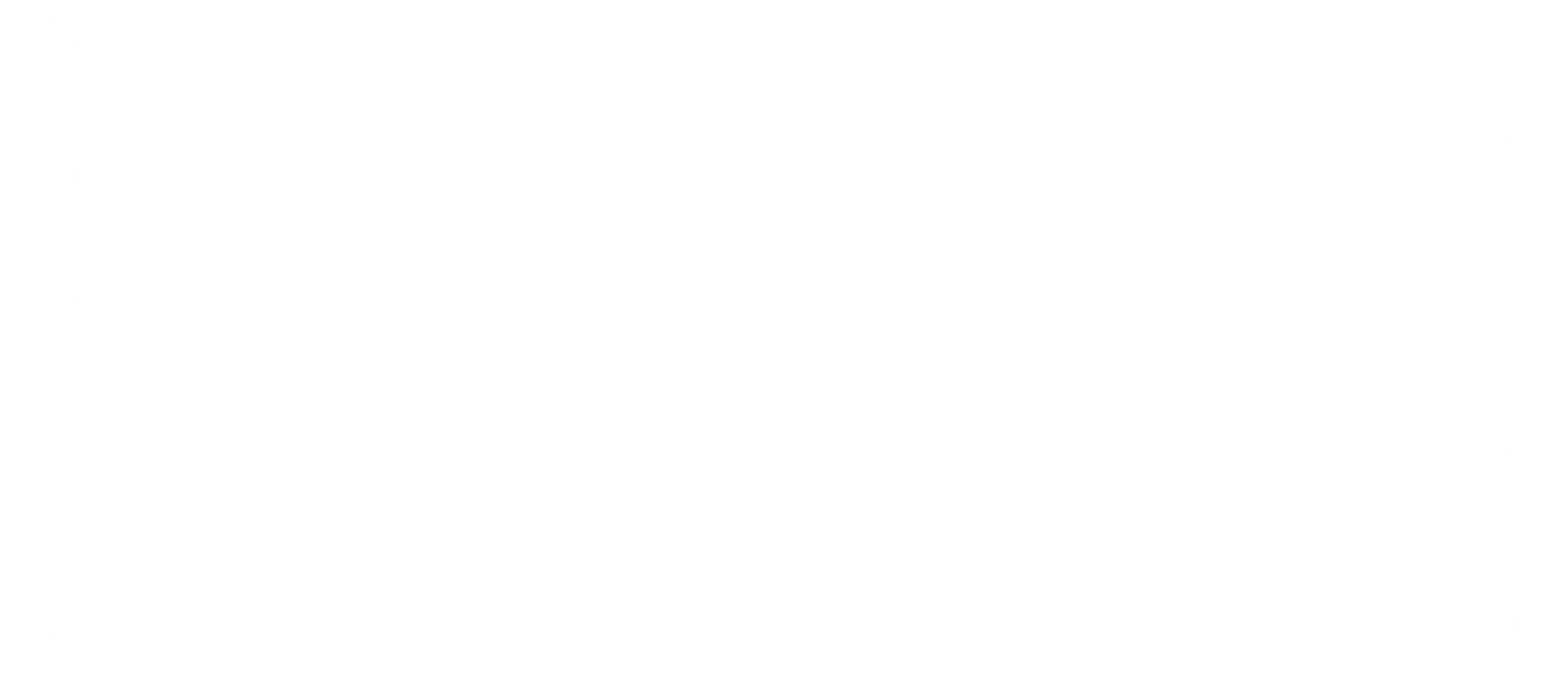


According to the table, has the highest priority, so the output is when , regardless of the other inputs. Similarly, the output is if and , regardless of the other inputs and so on.

The outputs and are simplified using K’ Maps.



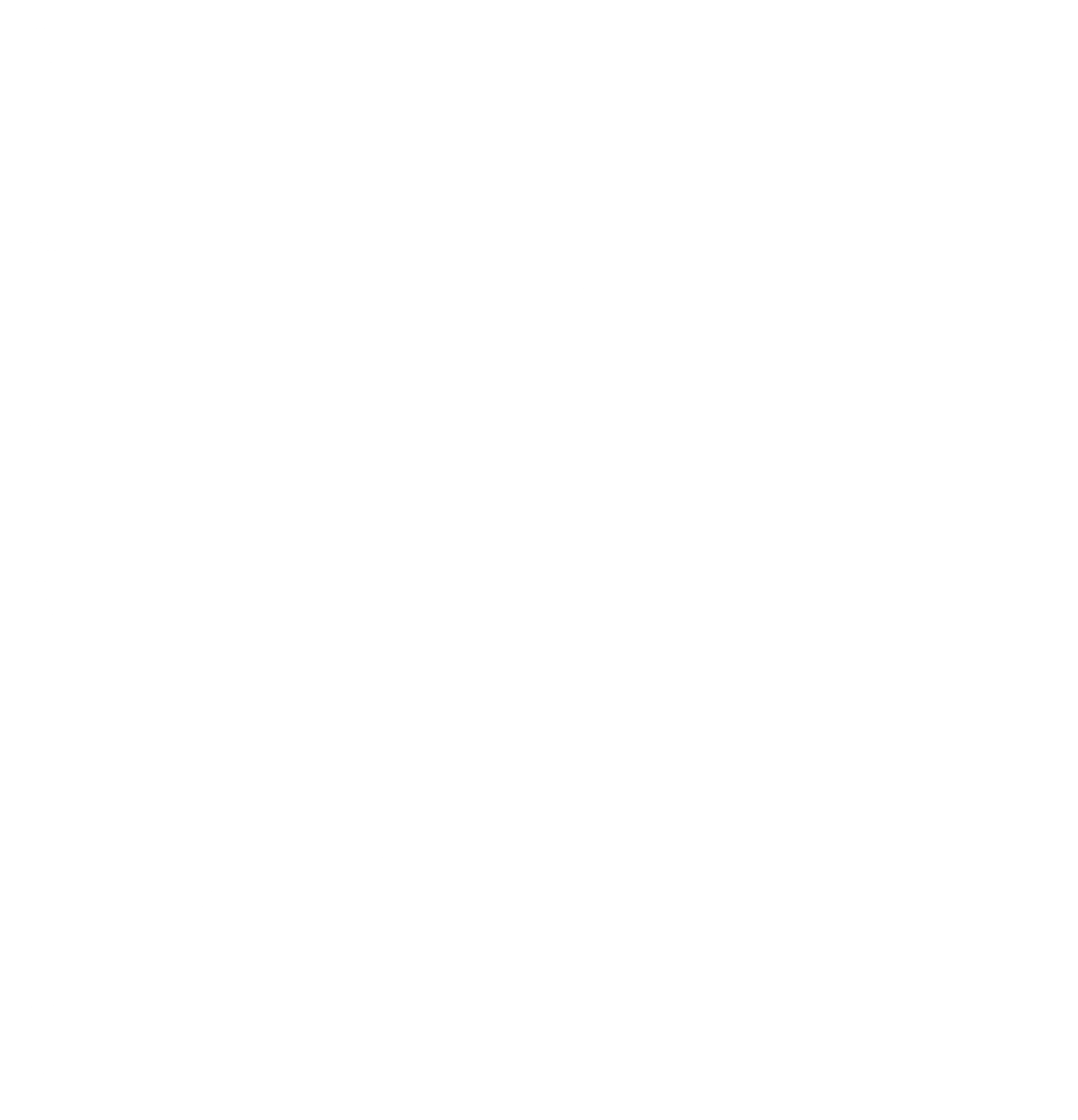
Thus, and .



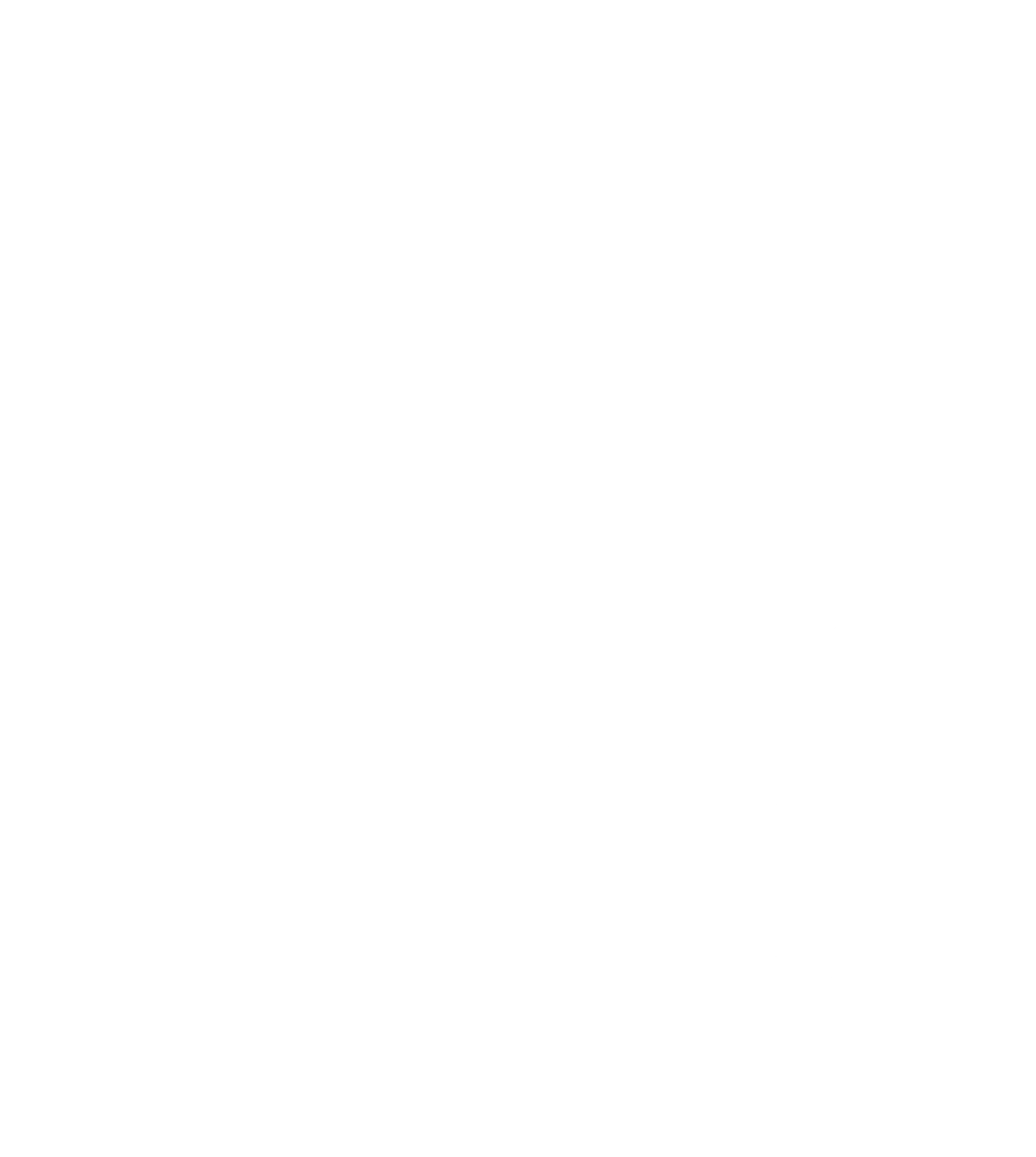
## Multiplexers

A multiplexer selects one of many input lines and directs it to a single output line. The selection is done by a set of selection lines. For input lines, selection lines would be needed to be able to select every possible combination.

For example, inputs would need selection lines. Each of the inputs is connected to an AND gate, along with the two selection lines. The AND gate will only give an output of 1 for the input that is selected. The following is the diagram for a -to--line multiplexer:



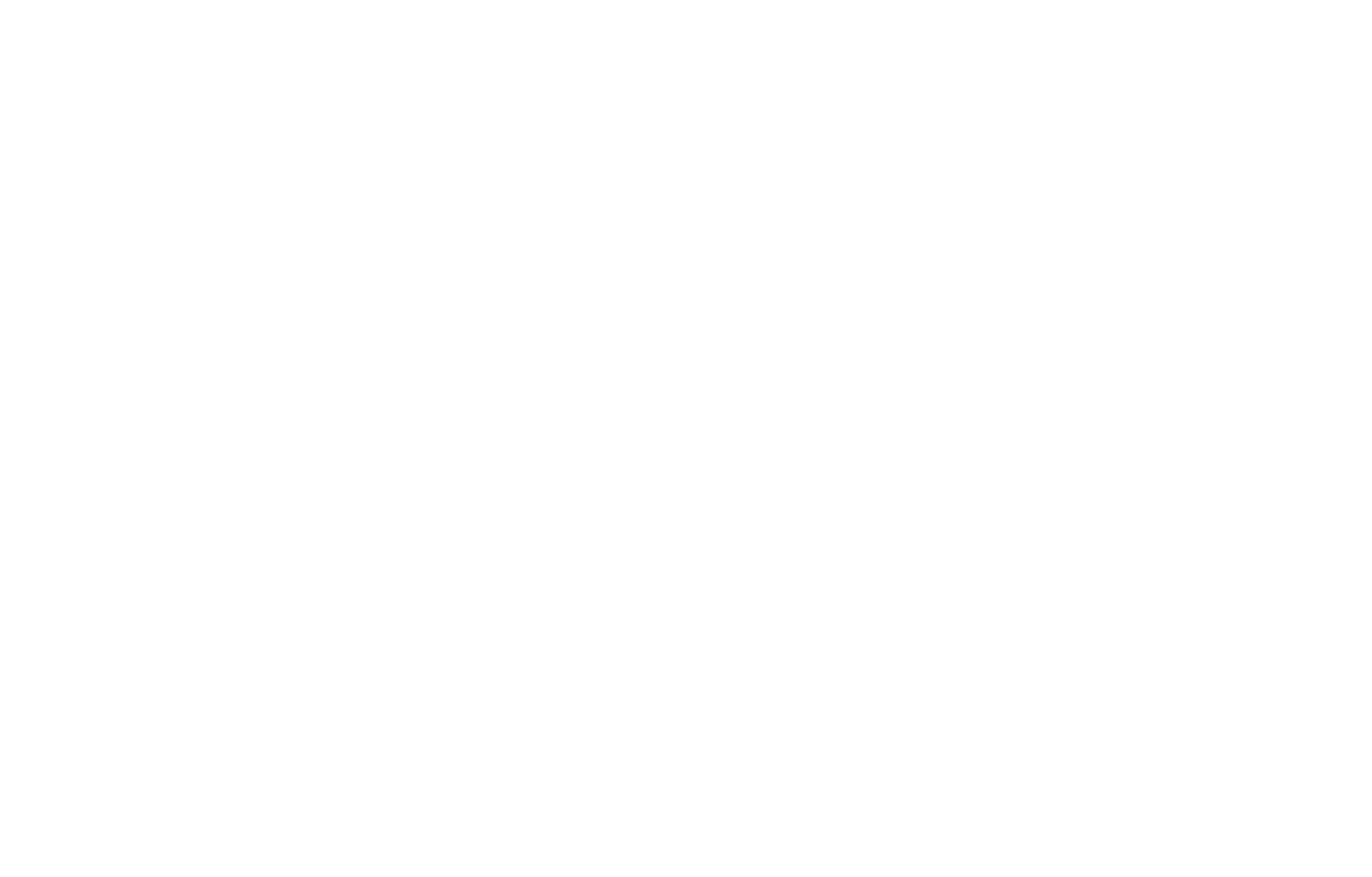
If and , then only the AND gate connected to will give an output of , since it is connected to and the inversion of . All the other AND gates will receive an input of from at least one of the selection lines, thus giving an output of . All the AND gates are connected to an OR gate, which will give an output of if any of the AND gates give an output of . The following is the associated truth table for input selection:



A multiplexer may also have an enable input that selects whether the system is active or not, as was done with decoders.

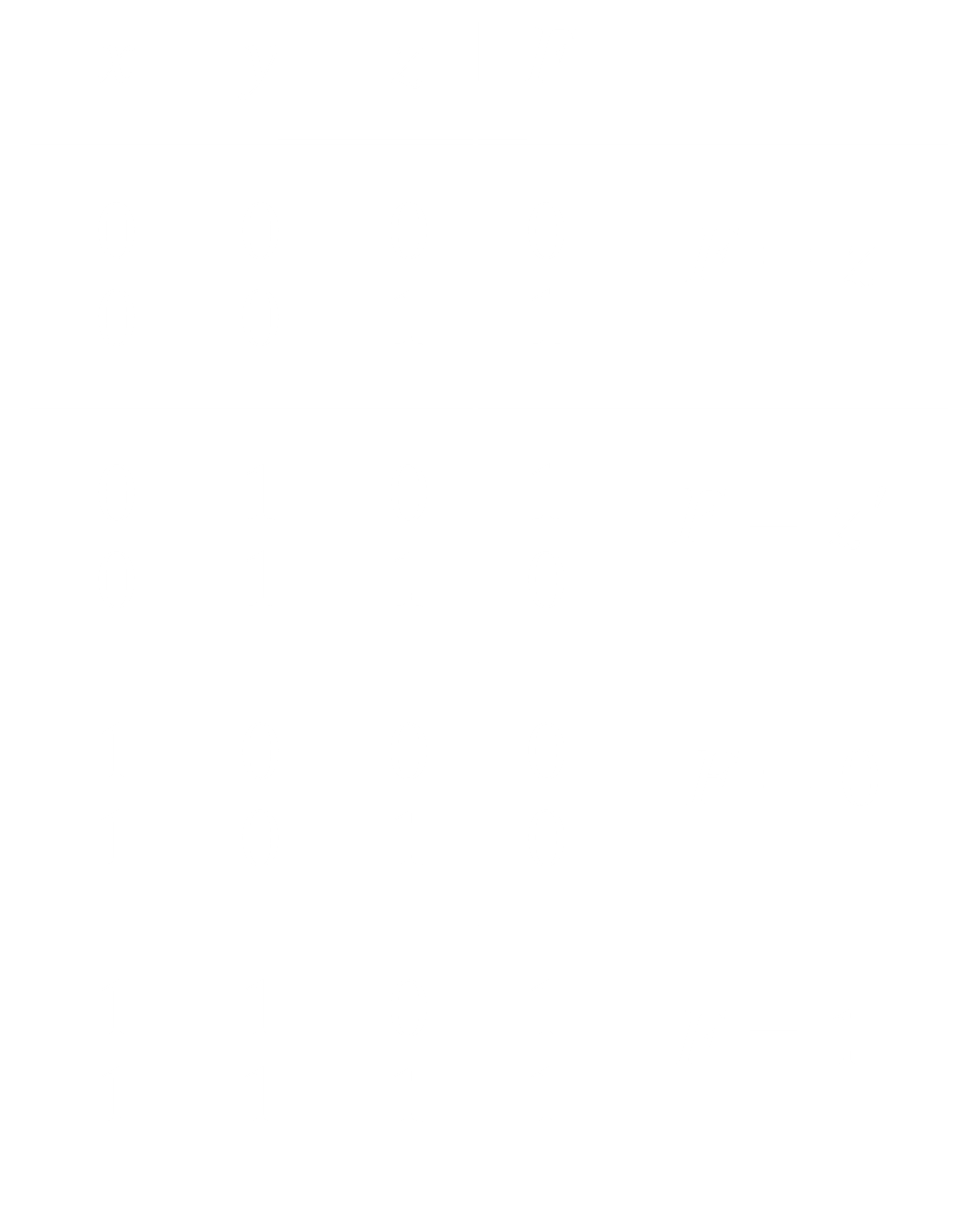
### Demultiplexer

A demultiplexer takes the input from a single input line and directs it to one of possible output lines, where is the number of selection lines. A demultiplexer is just a decoder with an enable input. When being considered as a demultiplexer, the enable input is considered to be the input line, while the other inputs are the selection lines. For example, the following -to--line decoder is a -to--line demultiplexer. Which specific NAND gate will give an output is decided by the selection lines and . Note here that the circuit is an active-low type.

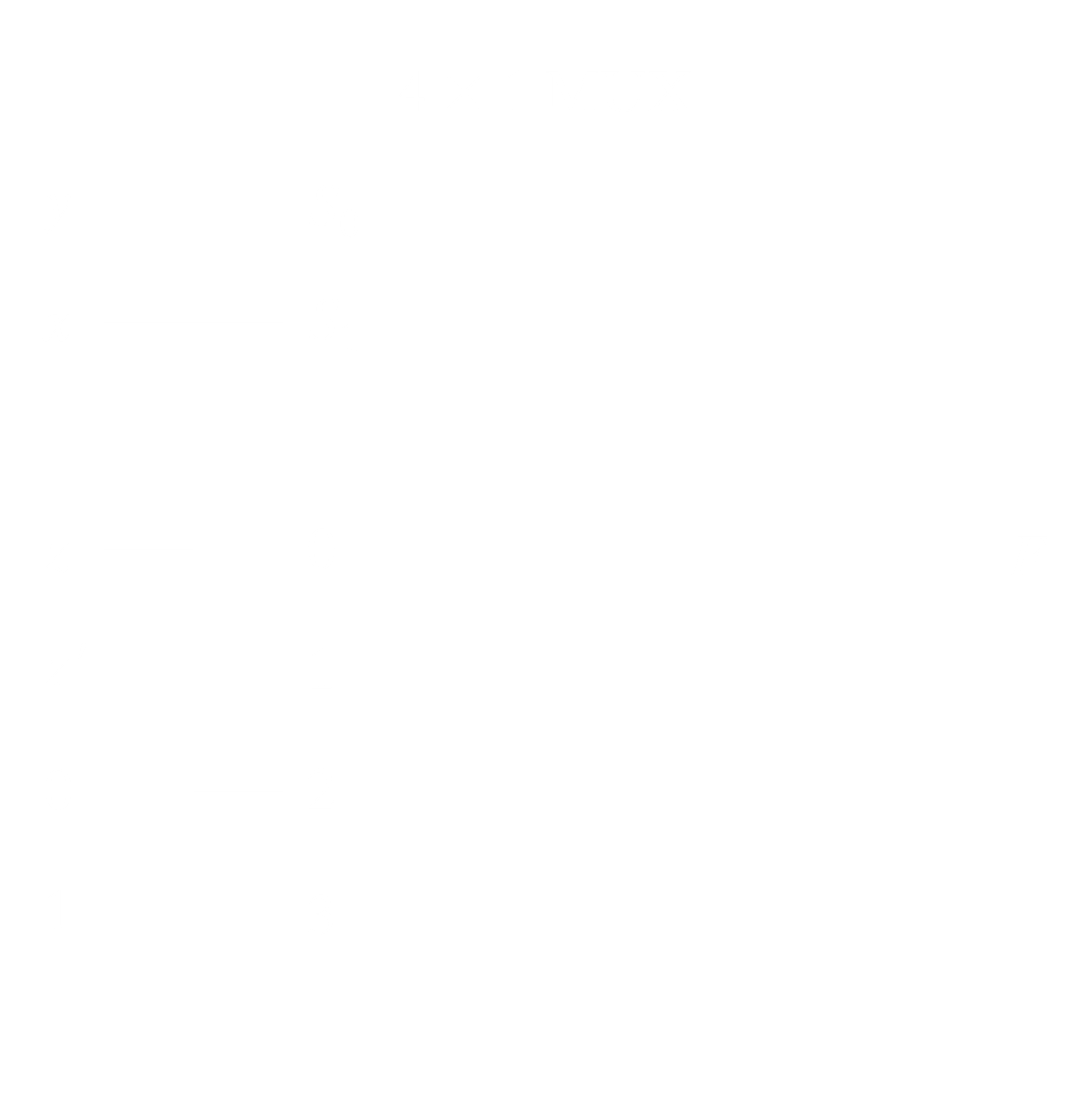


### Boolean Function Implementation with Multiplexers

Multiplexers can also be used to implement Boolean functions. For example, for the function , if we consider and to be the selection bits and to be the only input, the truth table looks like this:



The corresponding multiplexer implementation looks like this:



Every two combinations give a single value of for a single combination of and . Thus, we have possible values of , from which is selected using the selection bits. For example, we see that for and , , so is connected to (since ). When the required and values are selected, will give an output of . For a function with variables, there will be selection bits, inputs, and multiplexers.