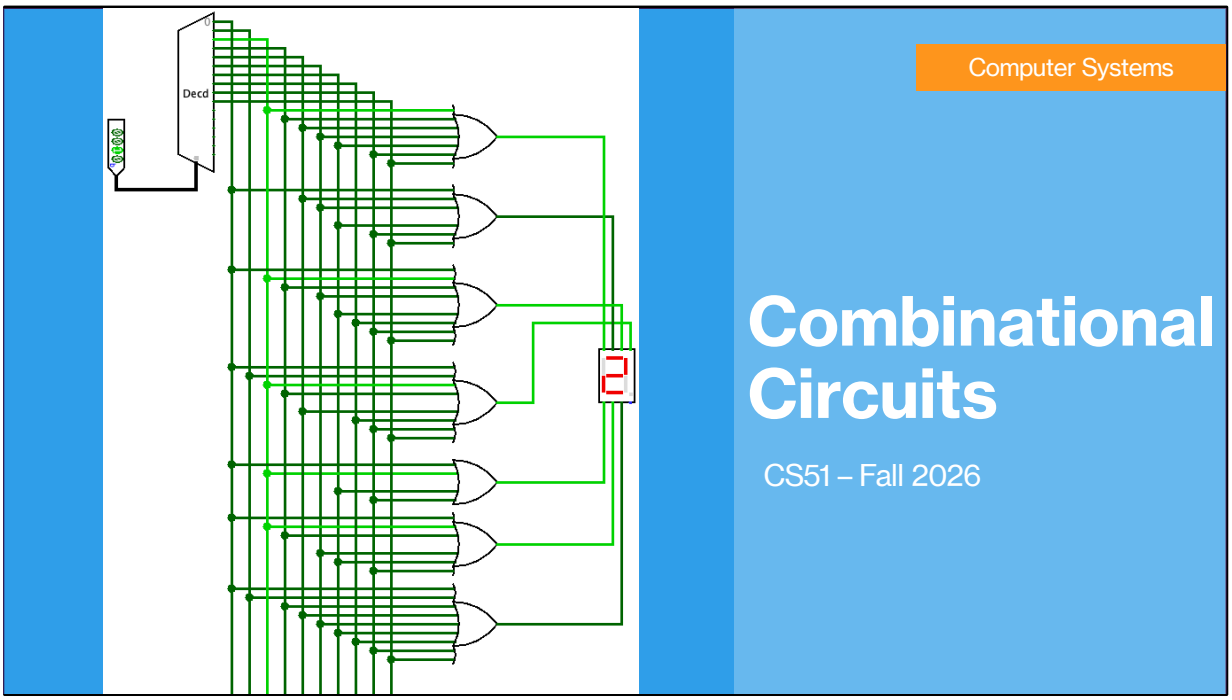


It's time to move one level higher in our hierarchy. Now that we know about bits and how we encode information, we can see how this information is stored and transmitted.

Bits, components, gates, and circuits

- Computers store and transmit information in the form of **bits**. This is accomplished by combining electronic components, such as **transistors**, in increasingly complex ways.
- For example, logic operations are evaluated by **logic gates** or simply **gates** built out of special types of transistors.
- In turn, logic gates can be combined to build more complex **circuits**.
- Today's lecture will be all about different types of **combinational circuits**: their output depends on the present inputs and there is no notion of memory of past inputs.
- To draw our circuits, we will use the software Logisim-Evolution.

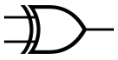
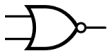
2

Computers store and transmit information in the form of bits. This is accomplished by combining electronic components, such as transistors, in increasingly complex ways. For example, logic operations are evaluated by logic gates or simply gates built out of special types of transistors. In turn, logic gates can be combined to build more complex circuits. Today's lecture will be all about different types of combinational circuits: their output depends on the present inputs and there is no notion of memory of past inputs. To draw our circuits, we will use the software Logisim-Evolution.

Logic Circuits

Let's start with the simplest circuits, the logic circuits.

Gates and Boolean logic

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

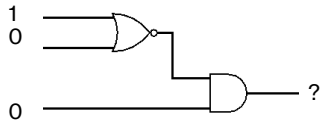
X	Y	¬X	X∧Y	X∨Y	X⊕Y	X$\bar{\wedge}$Y	X$\bar{\vee}$Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

4

These support the operations we have seen through Boolean Algebra. Logical NOT, AND, OR, and XOR, we saw already. What is new is NAND (the negation of AND) and NOR (the negation of NOR). At this stage, I want you to become familiar with the symbols for the logic gates. Remember, inversion (negation) is symbolized with a circle; you can see it at the end of NOT, NAND, and NOR. AND looks like a D shape; you can see it at the AND and NAND gate. OR is like an arrow/ you can see it at OR and NOR. Exclusive OR is like OR but with a smile.

PRACTICE TIME – Using gates

- What is the output of the following circuit for the provided input?



- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

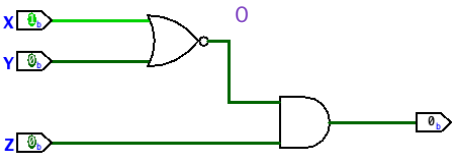
X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

5

Let's imagine we have built a small circuit with three inputs, two of which go to a NOR gate whose output is combined with the third input in a AND gate. If the inputs are 1, 0, and 0 respectively, what will the AND gate give us output? As an intermediate step, see what the output of NOR would be for an input of 1 and 0.

ANSWER – Using gates

- What is the output of the following circuit for the provided input?



Light green is 1 and dark green is 0

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

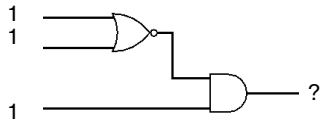
X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

6

NOR would produce 0. 0 AND 0 gives 0.

PRACTICE TIME – Using gates

- What is the output of the following circuit for the provided input?



- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

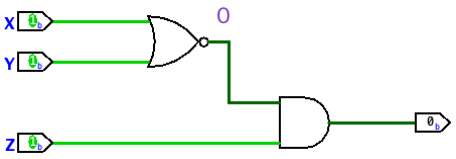
X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

7

What if all three inputs are 1?

ANSWER – Using gates

- What is the output of the following circuit for the provided input?



Light green is 1 and dark green is 0

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

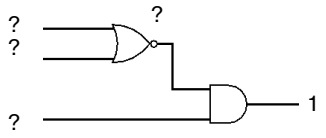
X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

8

The output of NOR is 0 combined as 0 AND 1 gives 0.

PRACTICE TIME – Using gates

- What inputs should we pass to get 1 as output?



- NOT (¬)
- AND (∧)
- OR (∨)
- XOR (⊕)
- NAND ($\bar{\wedge}$)
- NOR ($\bar{\vee}$)

X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

9

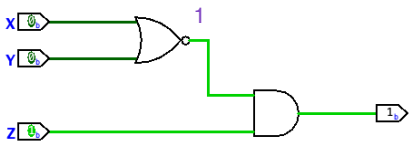
Let's try to reverse engineering this circuit. What inputs should we pass to get 1 as output?

ANSWER – Using gates

- What is the output of the following circuit for the provided input?

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

X	Y	Z	OUT
0	0	0	0
0	**0**	**1**	**1**
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



Light green is 1 and dark green is 0

X	Y	¬X	X∧Y	X∨Y	X⊕Y	X$\bar{\wedge}$Y	X$\bar{\vee}$Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

10

We know that AND will only produce 1 if it's given two inputs of 1. So the third input has to be 1. For NOR to produce 1, it needs a 0 and 0.

PRACTICE TIME – Using gates

- Can you build a circuit that reproduces the following inputs/outputs?

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

X	Y	Z	OUT
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

11

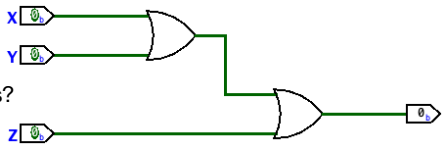
Here's another problem. If the output is always 0, unless the input is three 0s, what circuit would produce this result?

ANSWER – Using gates

- Can you build a circuit that reproduces the following inputs/outputs?

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

X	Y	Z	OUT
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



Light green is 1 and dark green is 0

X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

12

Two OR gates would give us this result. Only 0,0,0 would produce 0.

PRACTICE TIME – Using gates

- Can you build a circuit that reproduces the following inputs/outputs?

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

X	Y	Z	output
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0

X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

13

But how do we generalize this process? Let's put a hold here for a moment.

Minterm expansion

- We will use the following theorem we saw in the Boolean Algebra lecture:
- **Theorem (Boole, 1847):** Any Boolean function can be represented as a disjunction of conjunctions (one OR of a bunch of ANDs) of its arguments and their negation (NOT). This is also known as **disjunctive normal form** (DNF) or a sum of products/minterms.
- How to find the disjunctive normal form of a function (minterm expansion):
  - Using truth tables, we isolate the rows where the function evaluates to TRUE/1. Each row contributes to the OR an AND statement of the arguments (if TRUE/1) or their negation (if FALSE/0).

14

Remember the theorem we saw last week that any Boolean function can be represented as a DNF. We went about it by isolating the rows that had as output 1 and combine with OR statements the corresponding minterms (Boolean expressions of literals that appear as is or negations and are connected with AND). This technique is known as minterm expansion.

Minterm expansion for building circuits

- Using minterm expansion, we are able to build *any* circuit as an OR gate of AND gates of inputs and their negation (gate NOT) based on a disjunctive normal form.
- Such circuits will always have a depth of 3 which is significant because any signal has to pass through at most three gates to reach the output. This is quite fast!
- On the other hand, the minterm expansion produces circuits with lots of gates, which affects the total power consumption and the reliability of the circuit.
  - Assignment 2 introduced Karnaugh maps that allow us to simplify Boolean functions and thus circuits, too!

15

We will use minterm expansion to build any circuit as an OR gate of AND gates of inputs and negation (which we achieve using NOT gates) based on a DNF. Such circuits will always have a depth of 3 which is great because it means that any signal must only pass at most three gates which is very quick. On the other hand, even if it's a shallow circuit, it still might have a lot of gates. We could go about it using K-maps which we learned about in assignment 2 to make our circuits even smaller!

PRACTICE TIME – Using gates

- What is a DNF for this function?
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

in1	in2	in3	output
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0

16

OK let's go back to our question and start by building a DNF.

ANSWER – Using Gates

- Can you build a circuit that reproduces the following inputs/outputs?

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

in1	in2	in3	output
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0

(¬in1 ∧¬in2 ∧ ¬in3) ∨
(in1 ∧ ¬in2 ∧ ¬in3) ∨
(in1 ∧ in2 ∧¬in3)

17

You should have gotten this.

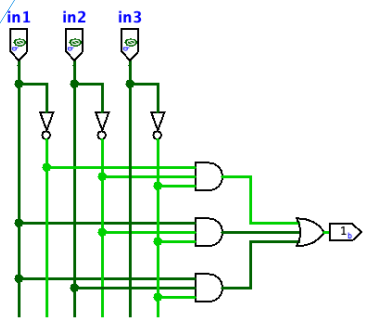
ANSWER – Using Gates

- Can you build a circuit that reproduces the following inputs/outputs?

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

in1	in2	in3	output
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0

(¬in1 ∧¬in2 ∧ ¬in3) ∨
(in1 ∧ ¬in2 ∧ ¬in3) ∨
(in1 ∧ in2 ∧¬in3)



Light green is 1 and dark green is 0

18

We will use it to build now the following circuit. Three inputs, in1, in2, in3, will feed to three AND gates which will feed to one OR gate. Note that each AND gate corresponds to a minterm so we need to pass the right literals to them. For example, the top AND gate will need to be given the negation of in1, the negation of in2, and the negation of in3. And so on.

PRACTICE TIME – Using Gates

- Can you build a K-map for this table?
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

in1	in2	in3	output
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0

19

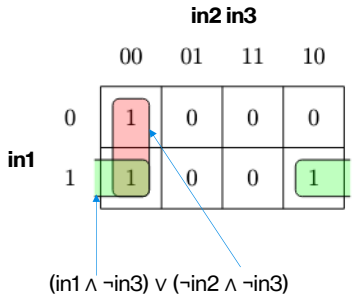
What if we use what we learned in assignment 2, can you build a K-map for this truth table?

ANSWER – Using Gates

- Can you build a K-map for this table?

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

in1	in2	in3	output
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0



20

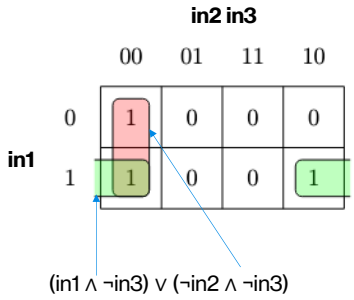
You should have ended up with this simple expression.

PRACTICE TIME – Using Gates

- Can you build a circuit using the Boolean expression you derived from the K-map?

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

in1	in2	in3	output
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0



21

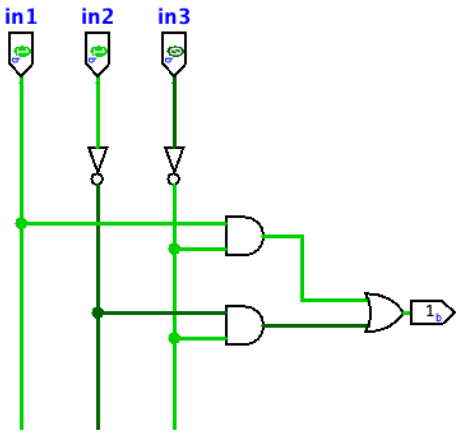
OK now try to build a circuit using the K-map you built.

ANSWER – Using Gates

- Can you build a circuit using the Boolean expression you derived from the K-map?

- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

in1	in2	in3	output
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0



22

Even simpler, right?

Arithmetic Circuits

Time to build some more interesting circuits.

Quick recap

$$\begin{array}{r} 0\,1\,0\,1\,0 \\ +\;0\,1\;1\,1\;1 \\ \hline \end{array}$$

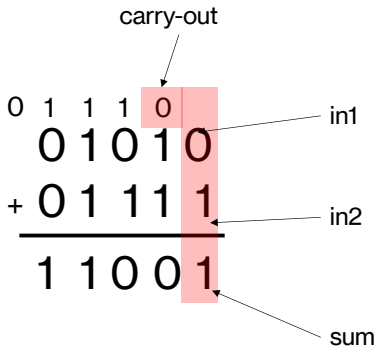
Do you remember how to add two binary numbers?

Quick recap

```
  0 1 1 1 0
    0 1 0 1 0
+   0 1 1 1 1
  ---------
    1 1 0 0 1
```

At the top, you can see the carry bit.

Think



- *How can we build a circuit that given input bits in1 and in2 produces carry-out and sum bits?*

If we focus on the last column of this addition we see that what we are doing is adding two bits, let's call them in1 and in2, and they give us back two bits: a carry-out and sum.

Think

- We want to design a circuit that given two input bits, in1 and in2, it will produce the sum and carry-out bits.
- Can you fill in the missing values for carry-out and sum?

Inputs		Outputs	
in1	in2	carry-out	sum
0	0	?	?
0	1	?	?
1	0	?	?
1	1	?	?

27

Can you fill in the missing values in the table?

Revisiting addition

- So far, our circuits have been performing logical operations. Now we will see how to build circuits that perform arithmetic operations
- We want to design a circuit that given two input bits, in1 and in2, it will produce the sum and carry-out bits shown below, that is it will add two bits.
- We can think of the circuit for each output independently.

Inputs		Outputs	
in1	in2	carry-out	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

28

So far, our circuits have been performing logical operations. Now we will see how to build circuits that perform arithmetic operations. We want to design a circuit that given two inputs, in1 and in2, it will produce the sum and carry-out bits shown below. We can think of the circuit for each output independently.

PRACTICE TIME – Carry out

X	Y	¬X	X∧Y	X∨Y	X⊕Y	X⊼Y	X⊽Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

- *Given input bits in1 and in2, what logical expression can we build to return the carry-out?*

Inputs		Outputs	
in1	in2	carry-out	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

29

If someone gave you this truth table on the right and asked you to design a circuit or a Boolean expression that given in1 and in2 gives you output carry out, what would you do?

ANSWER – Carry out

X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X ⊼ Y	X ⊽ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

- *Given input bits in1 and in2, what logical expression can we build to return the carry-out?*
  - Carry-out is calculated as in1 ∧ in2.

Inputs		Outputs	
in1	in2	carry-out	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

30

You can combine them with the logical AND operator

PRACTICE TIME - Sum

X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X ⊼ Y	X ⊽ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

- *Given input bits in1 and in2, what logical expression can we build to return the carry-out?*
  - Carry-out is calculated as in1 ∧ in2.
- *How about the sum?*

Inputs		Outputs	
in1	in2	carry-out	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

31

How about the sum?

ANSWER – Sum

X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X ⊼ Y	X ⊽ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

- *Given input bits in1 and in2, what logical expression can we build to return the carry-out?*
  - Carry-out is calculated as in1 ∧ in2.
- *How about the sum?*
  - Sum is calculated as in1 ⊕ in2.

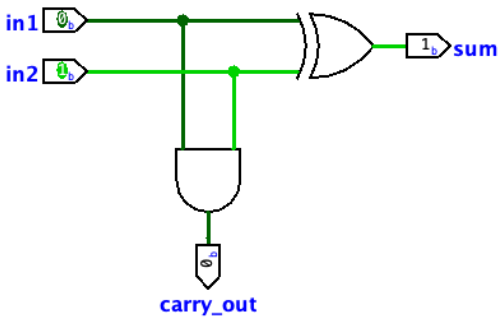
Inputs		Outputs	
in1	in2	carry-out	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

32

This would be in1 XOR in2

Half adder - circuit to add two bits

Inputs		Outputs	
in1	in2	carry-out	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

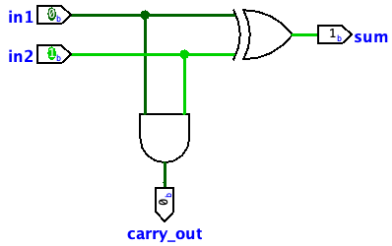


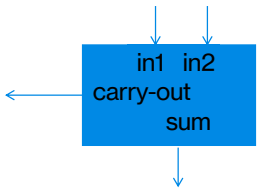
- A **half adder** is a circuit that computes the sum and carry of adding two bits, A and B, together.
  - Carry-out is calculated as in1 ∧ in2 and sum is calculated as in1 ⊕ in2.

33

Putting these two parts together, we can design a circuit called half adder whose job is to add two bits, A and B, together and produce the carry-out and sum as we saw.

Abstracting the half adder component



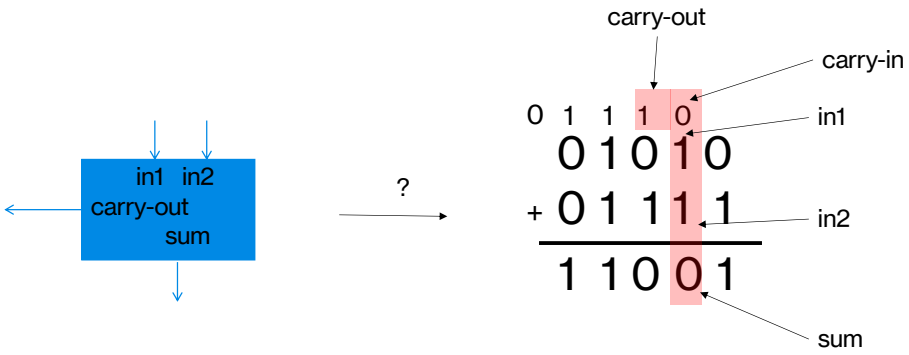


- Ignoring implementation details, we can treat the half-adder as a reusable component

34

We will next do something that is very common in Computer Science. We will ignore the details and abstract away from the implementation of the half-adder, treating it as a reusable component. The only thing we care at this point is that our component receives two input bits, in1 and in2, and gives us back two things: the carry-out bit and the sum bit.

Think

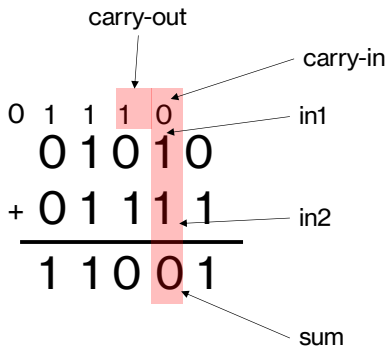


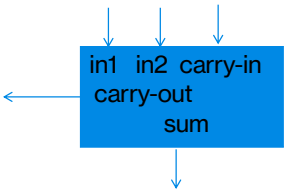
The half-adder adds two bits together and returns a carry-out and sum bit.
*How can we build a circuit that adds two bits **and a carry-in** to produce a sum and carry-out?*

35

The half-adder adds two bits together which only corresponds to adding the two least significant bits together. How can we build a circuit that adds two bits and a carry-in to produce a sum and carry-out, that is it corresponds to any other column in our addition?

Full adder - circuit to add two bits and a carry-in

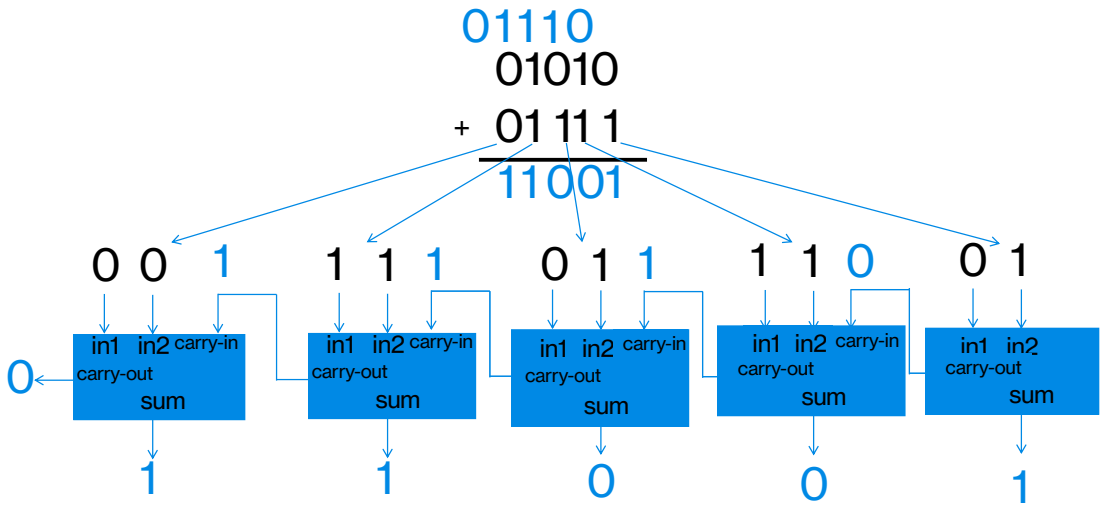




36

If we were to think abstractly, we realize that we want to build a component that does a full addition by taking three inputs, in1, in2, and carry-in, and giving us back two ouputs, a carry-out and a sum bit.

Revisiting addition



So we could stitch together our half-adder for the last column with four more full adders, each being responsible with propagating the right carry-out to the column to the left.

Think

Inputs			Outputs	
in1	in2	carry-in	sum	carry-out
0	0	0	?	?
0	0	1	?	?
0	1	0	?	?
0	1	1	?	?
1	0	0	?	?
1	0	1	?	?
1	1	0	?	?
1	1	1	?	?

38

If a full adder is a component that adds two bits and a carry-in together, what are the outputs sum and carry-out for each row of this truth table?

Full adder - circuit to add two bits and a carry-in

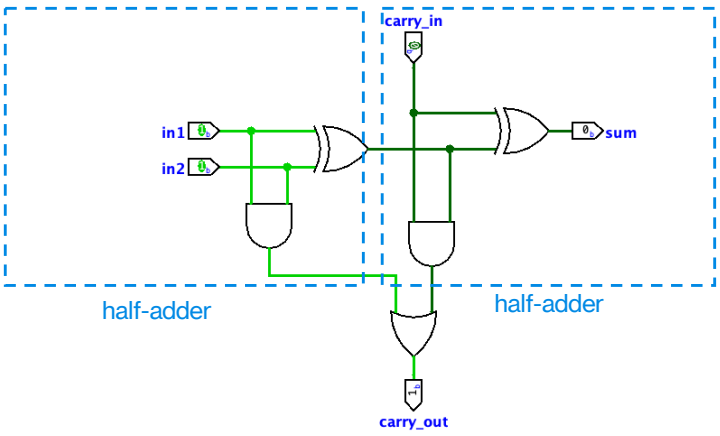
Inputs			Outputs	
in1	in2	carry-in	sum	carry-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

39

If a full adder is a component that adds two bits and a carry-in together, what are the outputs sum and carry-out for each row of this truth table?

Implementing a full adder

- The sum is calculated as (in1 ⊕ in2) ⊕ carry-in.
- The carry-out is calculated as (in1 ∧ in2) ∨ ((in1 ⊕ in2) ∧ carry-in).



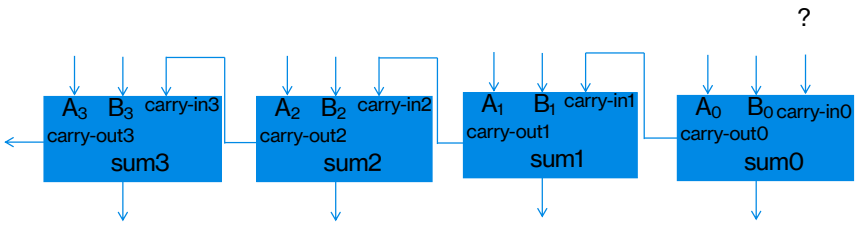
Inputs			Outputs	
in1	in2	carry-in	sum	carry-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

40

I am not going to ask you to implement a full adder as it's slightly more complicated. The sum is calculated as before in1 XOR in2 but then we XOR with the carry-in. The carry out is either in1 AND in2 or in1 XOR in2 AND carry-in. Take a moment to confirm it works as intended.

Think

- A full adder calculates the sum of a single column of the addition operation.
- If we want to add two n-bit numbers together, we chain n full adders, each responsible for one of the n columns of the addition operation. This circuit is known as a **ripple-carry adder**.
- For example, let's say we want to add two 4-bit numbers A and B, with $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$.
- *What goes in carry-in1?*

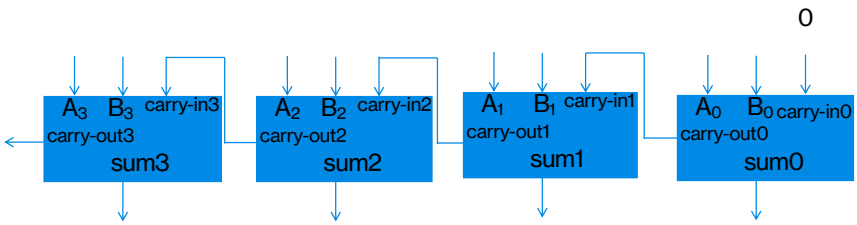


41

If we want to go beyond a single column to adding two n-bit numbers together, we can chain n full adders (instead of n-1 full adders and 1 half adder), each responsible for one of the n columns of the addition operation. This circuit is known as a ripple-carry adder. What would the first carry-in bit?

Ripple-carry adder

- A full adder calculates the sum of a single column of the addition operation.
- If we want to add two n-bit numbers together, we chain n full adders, each responsible for one of the n columns of the addition operation. This circuit is known as a **ripple-carry adder**.
- For example, let's say we want to add two 4-bit numbers A and B, with $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$.
- The first carry-in1 will be 0.



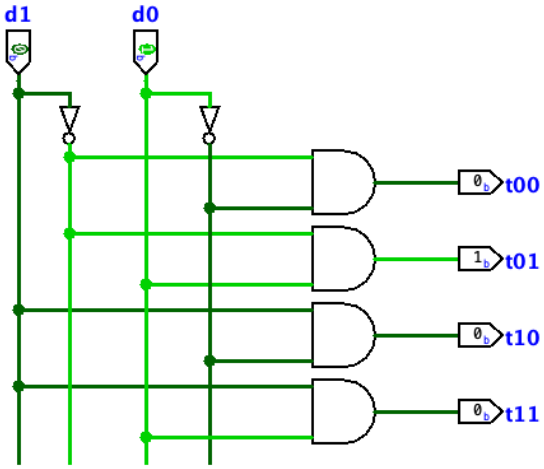
42

The first carry-in is always 0. How can we detect overflow if we're working with signed two's complement numbers?

Decoders

Next, we will talk about another type of combinational circuits called decoders.

Think

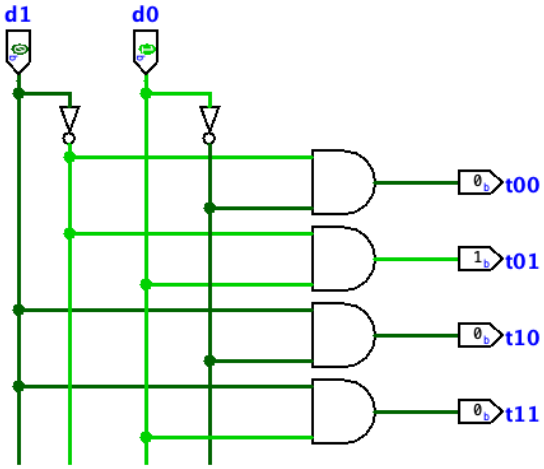


Inputs		Outputs			
d_1	d_0	t_{00}	t_{01}	t_{10}	t_{11}
0	0				
0	1				
1	0				
1	1				

44

Given the following circuit, can you fill out the truth table?

2-bit decoders

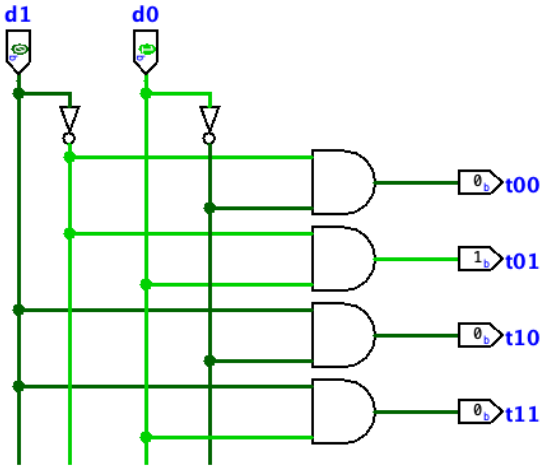


Inputs		Outputs			
d_1	d_0	t_{00}	t_{01}	t_{10}	t_{11}
0	0	**1**	0	0	0
0	1	0	**1**	0	0
1	0	0	0	**1**	0
1	1	0	0	0	**1**

45

Did you notice that it sends a 1 along one of the output lines only? This circuit is called a 2-bit decoder.

2-bit decoders



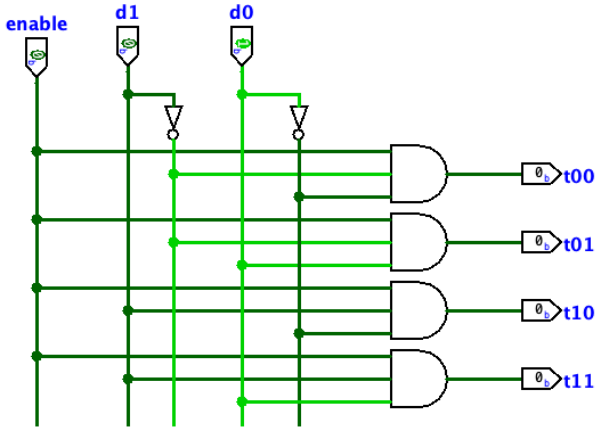
Inputs		Outputs			
d_1	d_0	t_{00}	t_{01}	t_{10}	t_{11}
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- 2-bit decoders are basic circuits that take 2 inputs and convert them up to 4 outputs.
- Exactly one of the outputs will be 1. Which one is determined by the value, in binary, of the two inputs.
- You can think of them as a binary to unary converter (from 2 to 1)

46

2-bit decoders are basic circuits that take 2 inputs and convert them up to 4 outputs. Exactly one of the outputs will be 1. Which one is determined by the value, in binary, of the two inputs as seen in the table above. You can think of them as a binary to unary converter. How it's done, is by passing all combinations of d1 and d0 to four AND gates,

Think

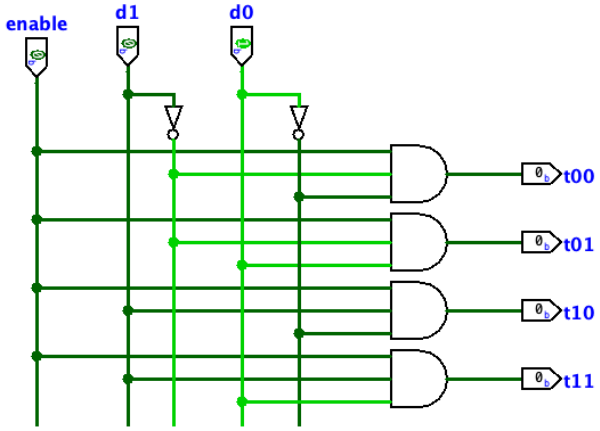


Inputs			Outputs			
enable	d_1	d_0	t_{00}	t_{01}	t_{10}	t_{11}
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

47

What if we add one more input, let's say enable? What does the table look like?

2-bit decoders with enable input



Inputs			Outputs			
enable	d_1	d_0	t_{00}	t_{01}	t_{10}	t_{11}
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

- When enable is on (1) the turned on and works as usual. When enable is off (0), the decoder is off.

48

Here's the table. What this means is that when enable is on (that is it gets passed the value 1), the decoder works as usual. When enable is off (that is 0), then no flow passes the decoder and all outputs are zero. What changes now is that our AND gates take three inputs.

Think

- *What if we wanted to build a 3-bit decoder? How many lines would we have?*

49

What if we were to move from 2 to 3 input bits and a build a decoder. How many output lines would we need?

3-bit decoder

- *What if we wanted to build a 3-bit decoder? How many lines would we have?*
  - 8

Inputs			Outputs							
d_2	d_1	d_0	t_{000}	t_{001}	t_{010}	t_{011}	t_{100}	t_{101}	t_{110}	t_{111}
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

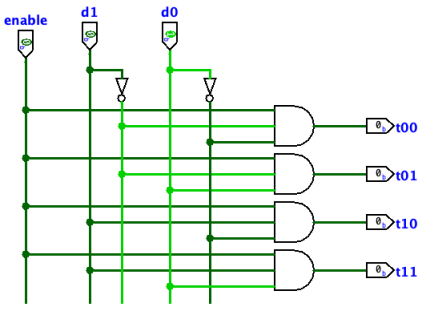
50

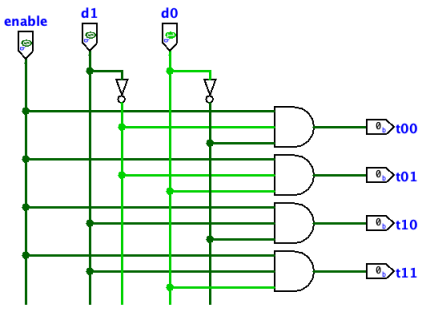
2^3=8

Building a 3-bit decoder

- We could design a circuit for scratch, but instead we will reuse 2-bit decoders.
- We will also use the idea of the enable input.

Inputs			Outputs							
d_2	d_1	d_0	t_{000}	t_{001}	t_{010}	t_{011}	t_{100}	t_{101}	t_{110}	t_{111}
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

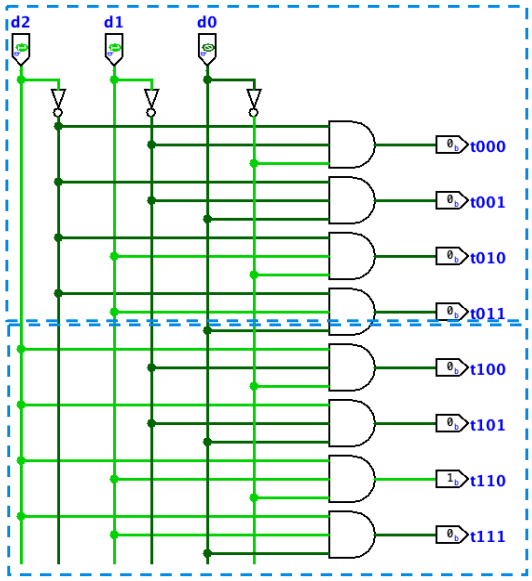




51

How can we build a 3-bit decoder? We could design a circuit for it from scratch, but instead, we will try to take advantage of prior work and reuse the 2-bit decoder. Remember, this is a picture of what 2 decoders with an enable input would look like.

Building a 3-bit decoder



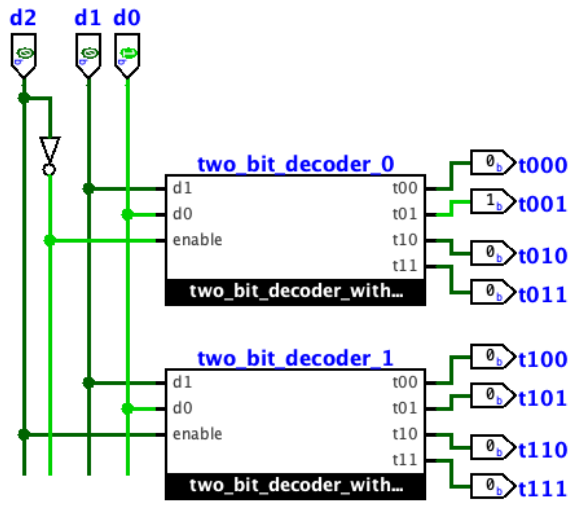
Inputs			Outputs							
d_2	d_1	d_0	t_{000}	t_{001}	t_{010}	t_{011}	t_{100}	t_{101}	t_{110}	t_{111}
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

- d_2 acts as the enable bit to two decoders.
- When d_2 is 0, the top decoder is activated, sending output to one of t0xx.
- When d_2 is 1, the bottom decoder is activated, sending output to one of t1xx.
- Complicated design

52

We are going to consolidate the two enable inputs into a single input d2. d2 acts as the enable bit for both decoders. When d_2 is 0, the top decoder is activated, sending output to one of t0xx. When d_2 is 1, the bottom decoder is activated, sending output to one of t1xx. But the design is kinda complicated.

Abstraction in decoders



Inputs			Outputs							
d_2	d_1	d_0	t_{000}	t_{001}	t_{010}	t_{011}	t_{100}	t_{101}	t_{110}	t_{111}
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

- d_2 acts as the enable bit to two decoders.
- When d_2 is 0, the top decoder is activated, sending output to one of t0xx.
- When d_2 is 1, the bottom decoder is activated, sending output to one of t1xx.
- Cleaner design

53

Instead, we will introduce abstraction in decoders and hide away the components of how the two bit decoders are implemented. Instead, we will treat them as two black boxes that we give them three inputs and they give us back one of 8 outputs as 1.

k-bit decoders

- We can generalize the idea of 2-bit decoders to k-bit decoders. There will now be k inputs and up to 2^k outputs.
- Exactly one of the 2^k outputs will be 1 for a given row in the truth table. This is known as "1-hot." Which one is determined by the value, in binary, of the k inputs.

54

We can generalize the idea of 2-bit decoders to k-bit decoders. There will now be k inputs and up to 2^k outputs. Exactly one of the 2^k outputs will be 1 for a given row in the truth table. This is known as 1-hot. Which one is determined by the value, in binary, of the k inputs.

7-Segment Display Decoders

55

Last type of combinational circuits is the 7-segnent display decoders.

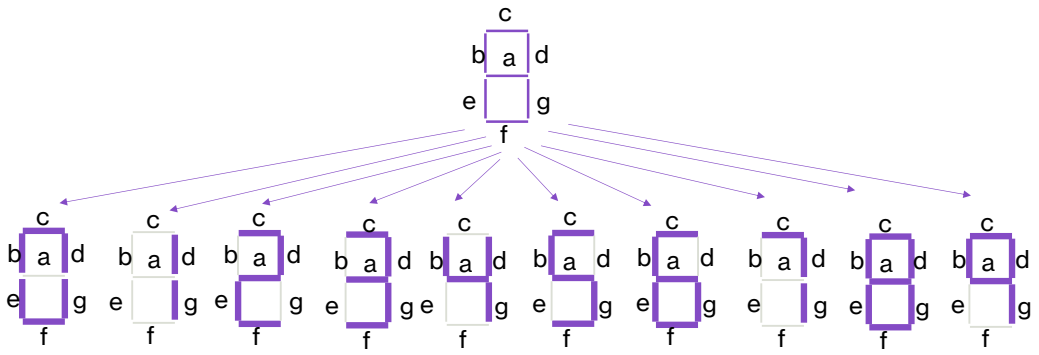
Decoder application example

How do clocks like this work?

56

Have you ever thought how clocks like this work?

Decoding 7-segment displays

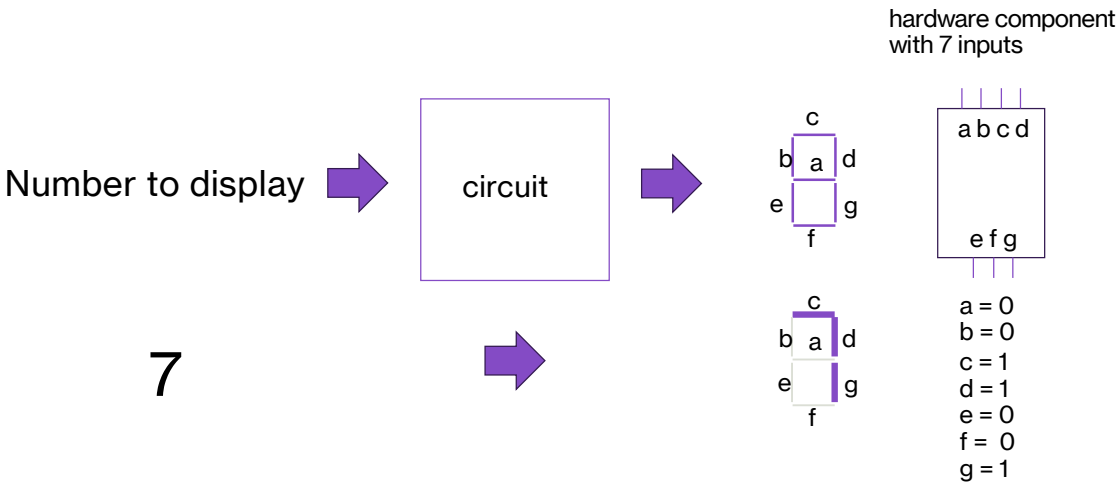


- We will assume that there are 7 segments, labeled a to g, as shown above.
- By illuminating different combinations of these segments, we can generate the digits 0-9.

57

Decoders provide a general idea of how we can generate different types of circuits.
One such example is a 7-segment display found on digital clocks and watches.
We will assume that there are 7 segments, labeled a to g, as shown above.
By illuminating different combinations of these segments, we can generate the digits 0-9.

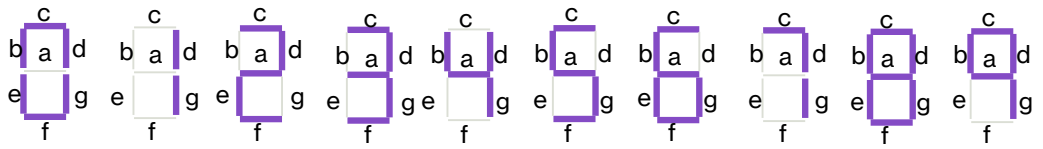
7-segment display



58

The way that this will happen is by activating the right segments to display a specific digit. For example, the number 7 gets displayed if we activate c, d, and g. All other segments need to be off.

Decoding 7-segment displays in clocks



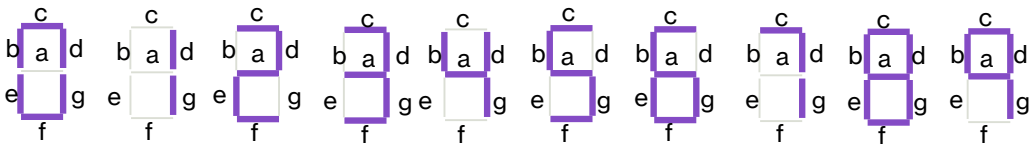
Digit	Illuminated segments						
	a	b	c	d	e	f	g
0		x	x	x	x	x	x
1				x			x
2	x		x	x	x	x	
3	x		x	x		x	x
4	x	x		x			x
5	x	x	x			x	x
6	x	x	x		x	x	x
7			x	x			x
8	x	x	x	x	x	x	x
9	x	x	x	x			x

- Each cell is marked with an X if the corresponding segment is illuminated in that particular digit.
- Let's see how we can use this information to build a 7-segment display in Logisim-Evolution.

59

Using these illustrations, we can build a table where each row corresponds to one of the 10 digits 0-9, each column to one of the 7 segments a-g, and each cell is marked with an X if the corresponding segment is illuminated in that particular digit. How can we build this on Logisim-Evolution?

Think



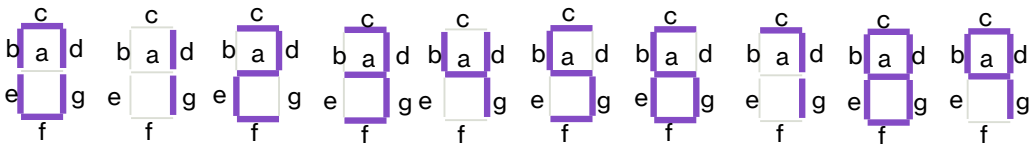
Digit	Illuminated segments						
	a	b	c	d	e	f	g
0		x	x	x	x	x	x
1				x			x
2	x		x	x	x	x	
3	x		x	x		x	x
4	x	x		x			x
5	x	x	x			x	x
6	x	x	x		x	x	x
7			x	x			x
8	x	x	x	x	x	x	x
9	x	x	x	x			x

- Each cell is marked with an X if the corresponding segment is illuminated in that particular digit.
- *How many binary inputs do we need for the numbers 0-9?*

60

We will start with asking the question, how many binary inputs do we need for 0-9?

Decoding 7-segment displays in clocks



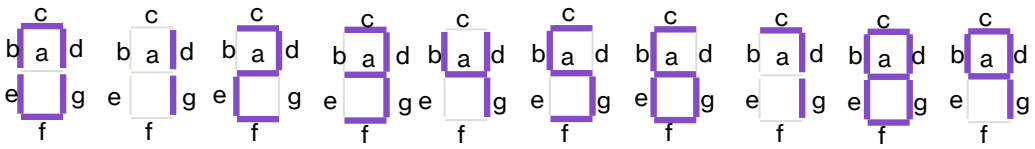
Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x
0001	1				x			x
0010	2	x		x	x	x	x	
0011	3	x		x	x		x	x
0100	4	x	x		x			x
0101	5	x	x	x			x	x
0110	6	x	x	x		x	x	x
0111	7			x	x			x
1000	8	x	x	x	x	x	x	x
1001	9	x	x	x	x			x

- Each cell is marked with an X if the corresponding segment is illuminated in that particular digit.
- *How many binary inputs do we need for the numbers 0-9?*
  - Since we have 10 digits, we need 4 bits of input to encode them.

61

Since we have 10 digits, we need at least 4 bits of input to encode them. 3 wouldn't be enough

Think



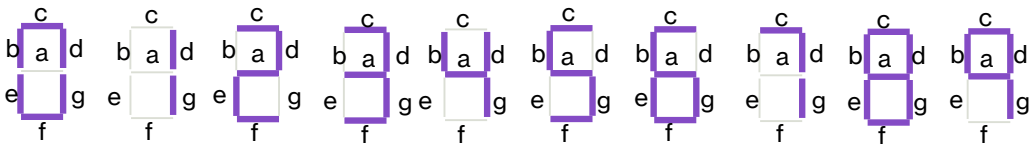
Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x
0001	1				x			x
0010	2	x		x	x	x	x	
0011	3	x		x	x		x	x
0100	4	x	x		x			x
0101	5	x	x	x			x	x
0110	6	x	x	x		x	x	x
0111	7			x	x			x
1000	8	x	x	x	x	x	x	x
1001	9	x	x	x	x			x

- Since we have 10 digits, we need 4 bits of input to encode them.
- *If we use a 4-bit decoder, how many output lines will we have?*

62

With a 4-bit decoder, how many lines will we have?

Think



Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x
0001	1				x			x
0010	2	x		x	x	x	x	
0011	3	x		x	x		x	x
0100	4	x	x		x			x
0101	5	x	x	x			x	x
0110	6	x	x	x		x	x	x
0111	7			x	x			x
1000	8	x	x	x	x	x	x	x
1001	9	x	x	x	x			x

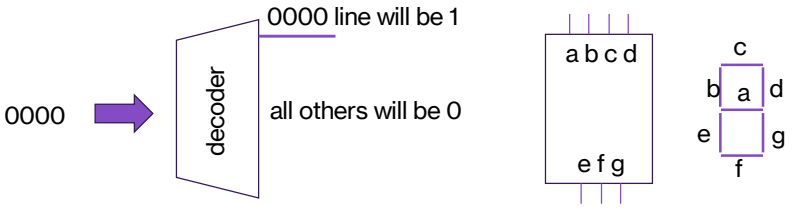
- Since we have 10 digits, we need 4 bits of input to encode them.
- *If we use a 4-bit decoder, how many output lines will we have?*
  - 2^4=16. exactly 1 will be activated depending on the input

63

It would be 2^4=16

Our friend the decoder

Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x

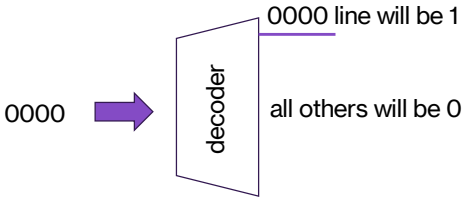


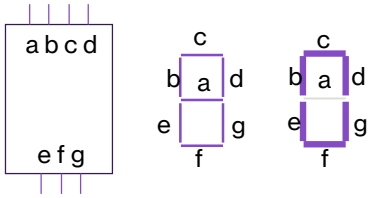
64

Let's assume we are given the input 0000, That means that the output line 0000 will be 1 and all others will be 0.

Think

Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x





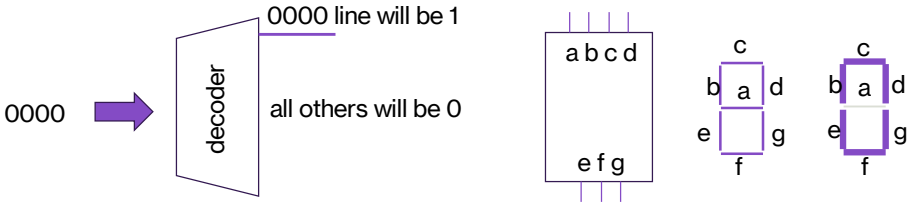
How do we get it to display a 0?

65

How do we get it to display the number 0?

Our friend the decoder

Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x



Connect the 0000 line to: b c d e f g

When the line is active those segments will illuminate

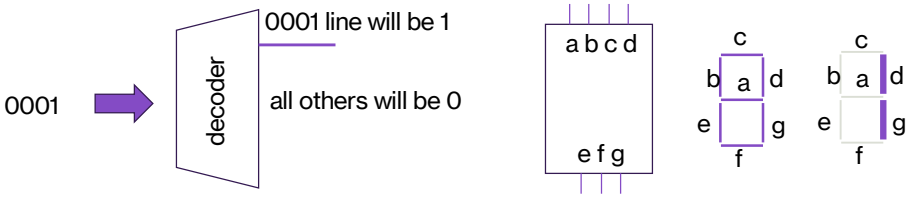
When it's not active, no display

66

We would connect the 0000 line to b, c, d, e, g, and g. Whne the line is active those segments will illuminate. When it's not active, nothing is displayed.

Our friend the decoder

Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0001	1				x			x



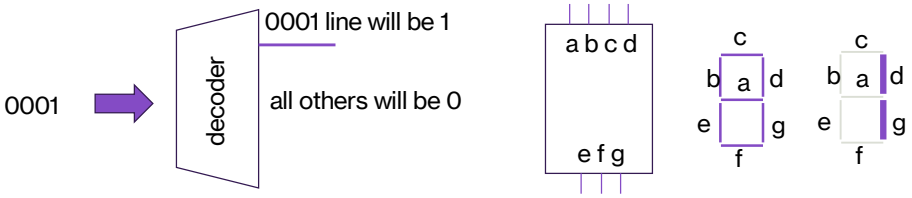
How do we get it to display a 1?

67

What if we wanted to display 1 (that is our input is 0001)?

Our friend the decoder

Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0001	1				x			x



Connect the 0001 line to: d g

When the line is active those segments will illuminate

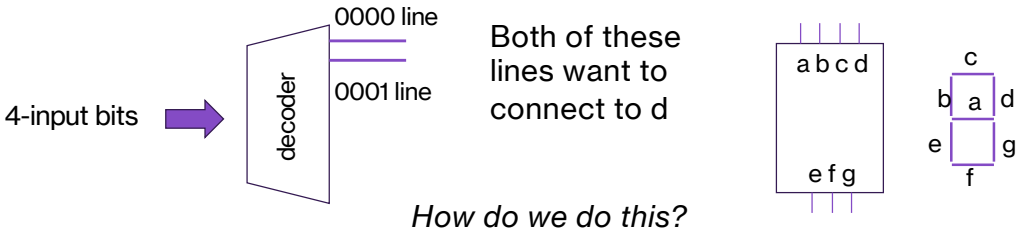
When it's not active, no display

68

We'd similarly connect 0001 to d and g.

Our friend the decoder

Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x
0001	1				x			x



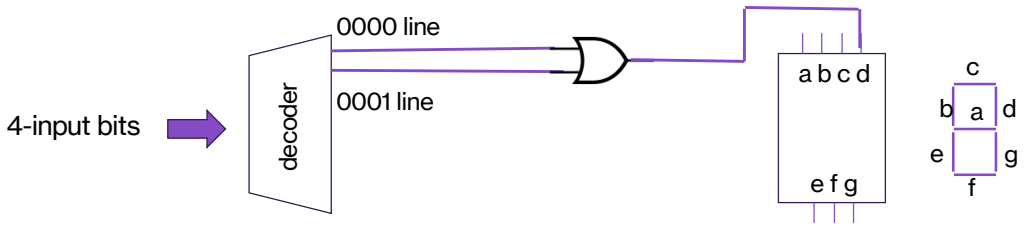
How do we do this?
Hint: gate??

69

We just saw that both 0 and 1 want to connect to segment d. What is the gate that we'd need for that?

Our friend the decoder

Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x
0001	1				x			x

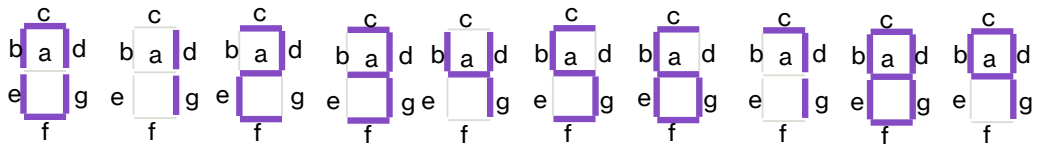


Since only one line will be active at a time, use OR gate

70

We can connect them with an OR gate!

Building a 7-segment display in Logisim



Digit	Illuminated segments						
	a	b	c	d	e	f	g
0		x	x	x	x	x	x
1				x			x
2	x		x	x	x	x	
3	x		x	x		x	x
4	x	x		x			x
5	x	x	x			x	x
6	x	x	x		x	x	x
7			x	x			x
8	x	x	x	x	x	x	x
9	x	x	x	x			x

- For each segment a-f of the clock, we will go to the corresponding table column and find the digits that have that segment illuminated. We will then connect the corresponding outputs of the decoder to an OR gate.
  - For example, the segment e is illuminated in 0, 2, 6, and 8. We will connect the 0, 2, 6, and 8 outputs of the 4-bit decoder to an OR gate.
- We will repeat this process for all 7 segments, ending up with 7 OR gates.
- We will then connect each OR gate with the appropriate input of a 7-segment display.

71

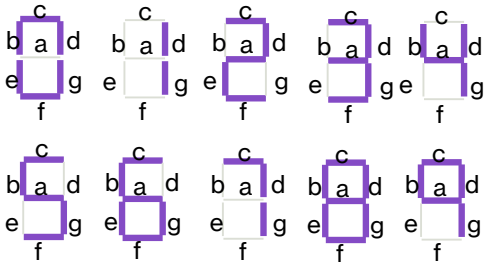
Putting it all together, for each segment a-f of the clock, we will go to the corresponding table column and find the digits that have that segment illuminated. We will then connect the corresponding outputs of the decoder to an OR gate.

> For example, the segment e is illuminated in 0, 2, 6, and 8. We will connect the 0, 2, 6, and 8 outputs of the 4-bit decoder to an OR gate.

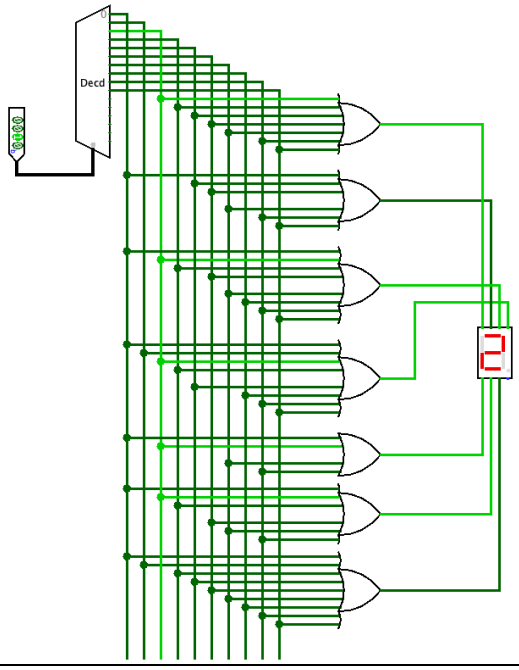
We will repeat this process for all 7 segments, ending up with 7 OR gates.

We will then connect each OR gate with the appropriate input of a 7-segment display.

A 7-segment display in Logisim

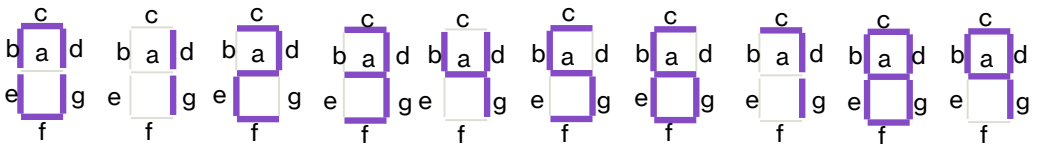


Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x
0001	1				x			x
0010	2	x		x	x	x	x	
0011	3	x		x	x		x	x
0100	4	x	x		x			x
0101	5	x	x	x			x	x
0110	6	x	x	x		x	x	x
0111	7			x	x			x
1000	8	x	x	x	x	x	x	x
1001	9	x	x	x	x			x

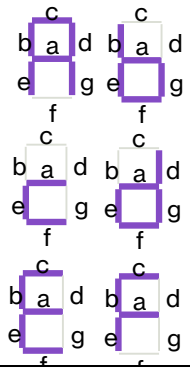


Cool stuff?

PRACTICE TIME – Expanding to 16 characters



- Imagine that the 7-segment display we have built so far is expanded to include 16 characters, the existing 10 digits, and A-F, as depicted here to differentiate among them.
- Expand the table on the right to reflect all 16 characters.
- *How will the design of the circuit change on Logisim-Evolution?*

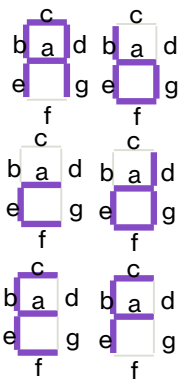


Input	Character	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x
0001	1				x			x
0010	2	x		x	x	x	x	
0011	3	x		x	x		x	x
0100	4	x	x		x			x
0101	5	x	x	x			x	x
0110	6	x	x	x		x	x	x
0111	7			x	x			x
1000	8	x	x	x	x	x	x	x
1001	9	x	x	x	x			x

Your turn now. Imagine that the 7-segment display we have built so far is expanded to include 16 characters, the existing 10 digits, and A-F, as depicted here to differentiate among them. Expand the table on the right to reflect all 16 characters. How will the design of the circuit change on Logisim-Evolution?

ANSWER – Expanding to 16 characters

- Expanded table:
- Circuit in Logisim-Evolution is expanded to include all outputs of decoders connected to 6 additional or gates



Input	Character	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x
0001	1				x			x
0010	2	x		x	x	x	x	
0011	3	x		x	x		x	x
0100	4	x	x		x			x
0101	5	x	x	x			x	x
0110	6	x	x	x		x	x	x
0111	7			x	x			x
1000	8	x	x	x	x	x	x	x
1001	9	x	x	x	x			x
1010	A	x	x	x	x	x		x
1011	B	x	x			x	x	x
1100	C	x				x	x	
1101	D	x			x	x	x	x
1110	E	x	x	x		x	x	
1111	F	x	x	x		x		

74

You can see how we have added more rows to capture all 16 possible outputs and have matched the illuminated segments. The circuit in Logisim-Evolution is expanded to include all outputs of decoders connected to 6 additional or gates

Combinational circuits

- So far, all circuits we have seen have been **combinational**: their output depends on the present inputs and there is no notion of memory of past inputs.
- Roughly speaking, combinational circuits perform **arithmetic and logical operations**, **transmit data**, or **convert code**. So far, we have seen for
  - *Arithmetic and logical operations*: logic circuits and half/full/and ripple-carry adders
  - Transmit data: decoders (and we will see multiplexers in the next assignment)
  - Convert code: 7-segment display decoders
- Other combinational circuits you might hear about are *encoders* (the opposite of decoders, they compress multiple lines of inputs into a smaller number of outputs) and *demultiplexers* (the opposite of multiplexers, you go from one input to multiple outputs, instead of multiple inputs to one output).

75

To summarize, so far, all circuits we have seen have been combinational: their output depends on the present inputs and there is no notion of memory of past inputs. Roughly speaking, combinational circuits perform arithmetic and logical operations, transmit data, or convert code. So far, we have seen for

Arithmetic and logical operations: logic circuits and half/full/and ripple-carry adders

Transmit data: decoders (and we will see multiplexers in the next assignment). A multiplexer (MUX) selects one of many data inputs and sends it to a single output, acting like a digital switch, while a decoder converts a binary code into a unique set of output signals, activating one specific output out of many for a given input code. The key difference is the direction of data flow and purpose: a MUX combines multiple data lines into one, and a decoder distributes a coded signal to multiple possible lines

Convert code: 7-segment display decoders

Other combinational circuits you might hear about are encoders (the opposite of

decoders, they compress multiple lines of inputs into a smaller number of outputs) and demultiplexers (the opposite of multiplexers, you go from one input to multiple outputs, instead of multiple inputs to one output).

Resources

- Download Logisim Evolution from https://github.com/logisim-evolution/logisim-evolution?tab=readme-ov-file#download
- Combinational circuits shown in lecture today if you want to work with them on Logisim Evolution https://cs.pomona.edu/classes/cs51/circuits/combinational_circuits.circ