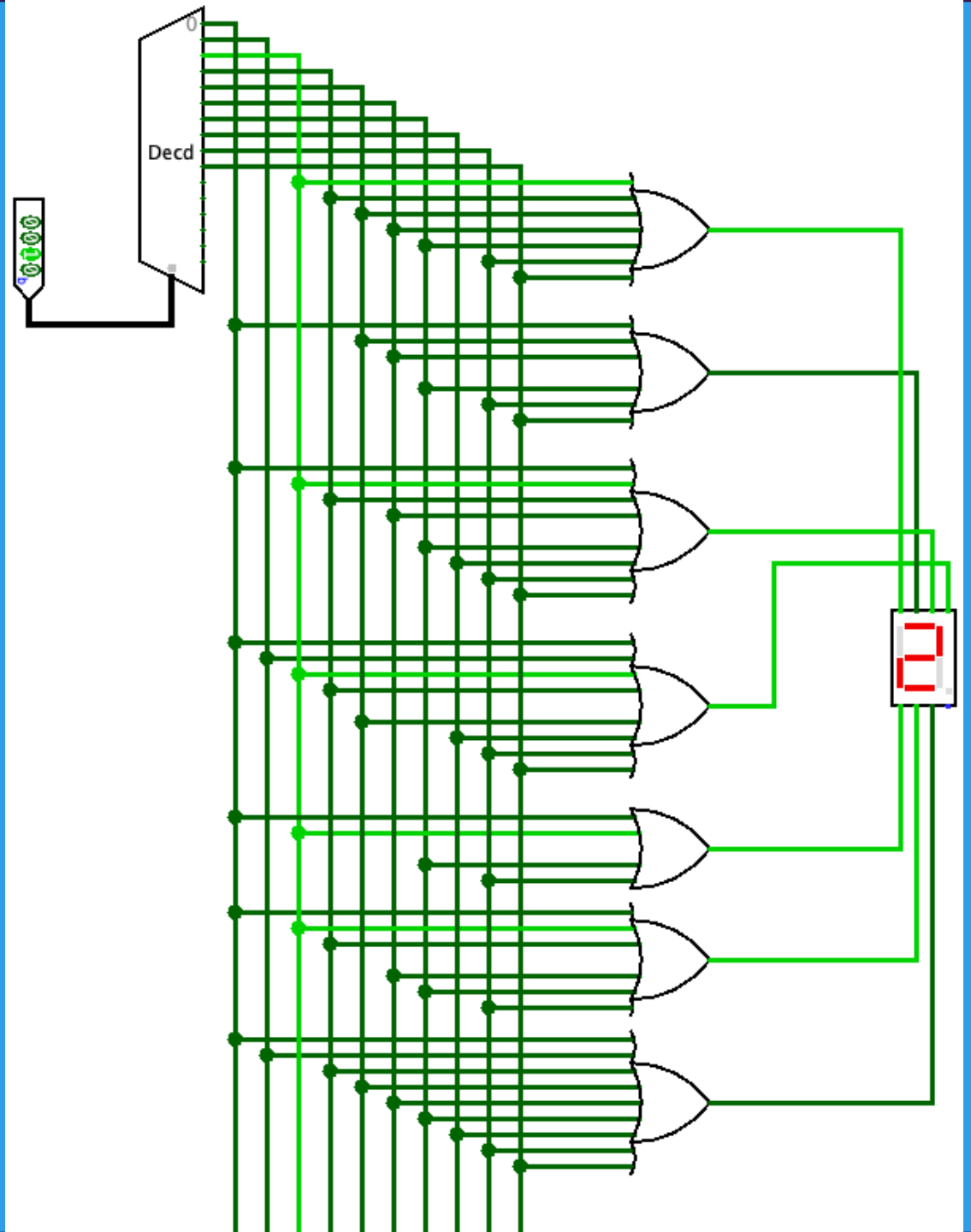


Computer Systems

Combinational Circuits

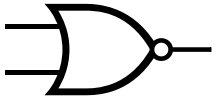
CS51 – Fall 2025

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.

Logic Circuits

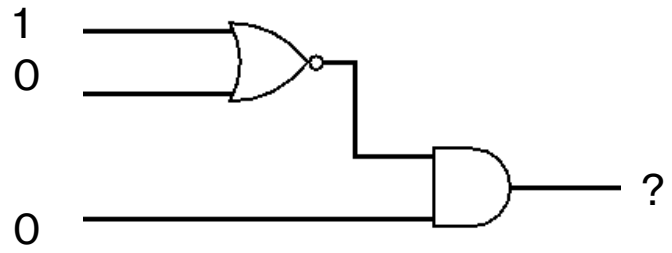
Gates and Boolean logic

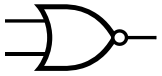
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND (⊼) 
- NOR (⊽) 

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

PRACTICE TIME – Using gates

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

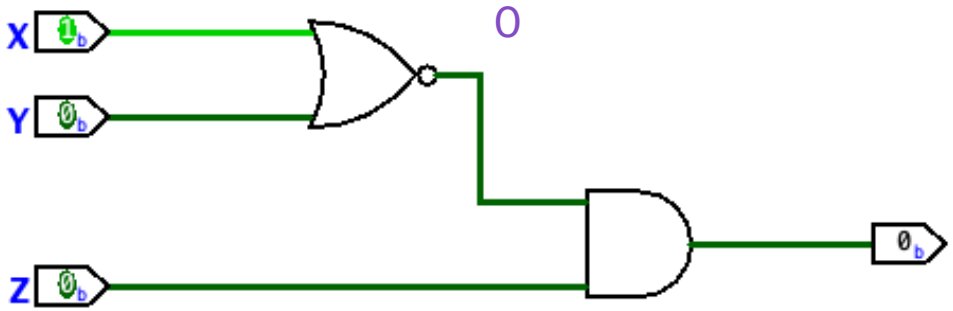


- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

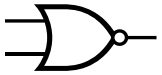
X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

ANSWER – Using gates

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



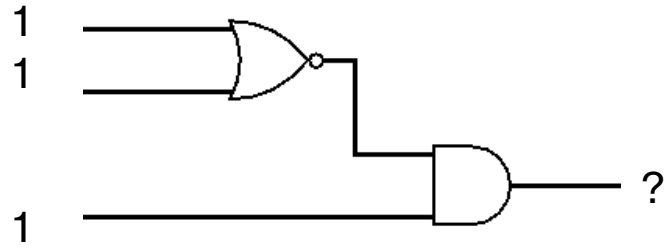
Light green is 1 and dark green is 0

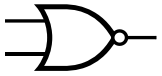
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

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

PRACTICE TIME – Using gates

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

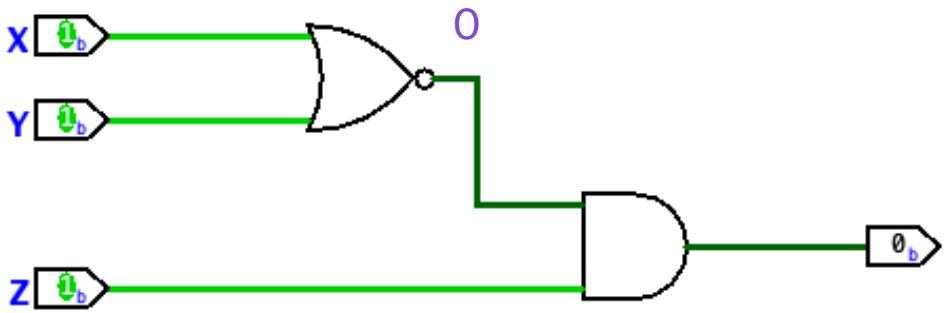


- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

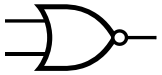
X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

ANSWER – Using gates

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



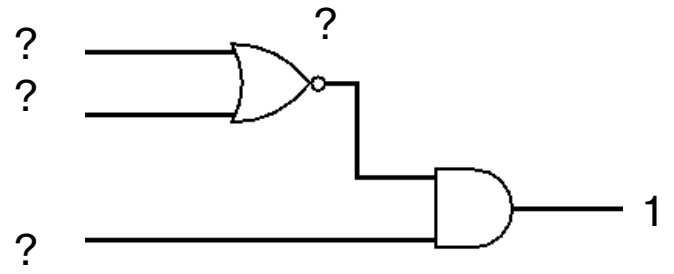
Light green is 1 and dark green is 0

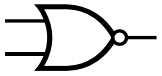
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

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

PRACTICE TIME – Using gates

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

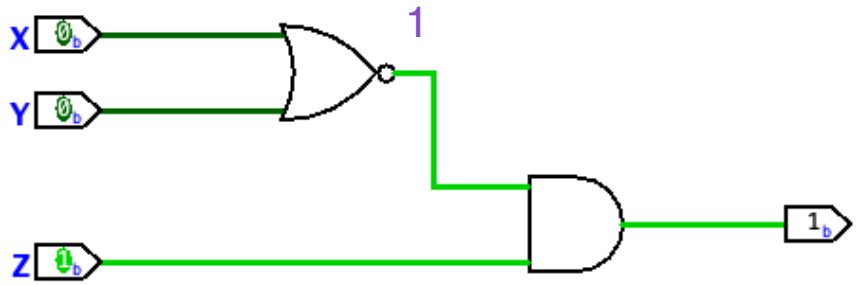


- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

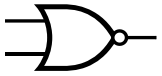
X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

ANSWER – Using gates

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



Light green is 1 and dark green is 0

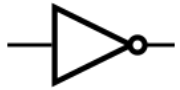
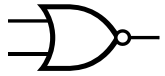
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

In1	In2	In3	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

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

PRACTICE TIME – Using gates

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

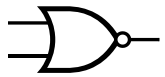
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

In1	In2	In3	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

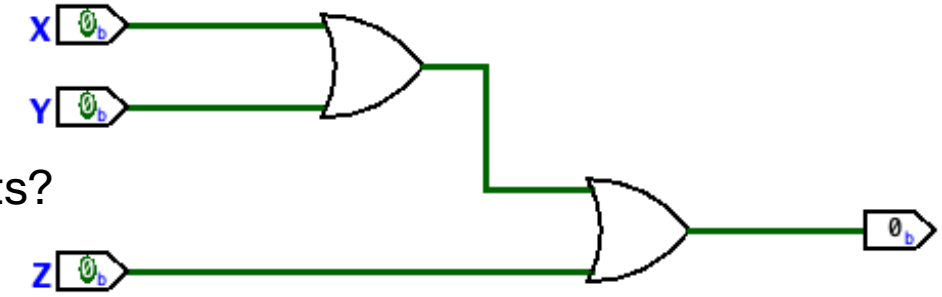
X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X $\bar{\wedge}$ Y	X $\bar{\vee}$ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

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	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

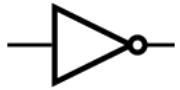
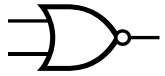


Light green is 1 and dark green is 0

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

PRACTICE TIME – Using gates

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

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

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

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

Minterm expansion

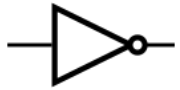
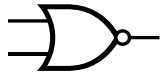
- We will use the following theorem we saw in the Boolean Algebra lecture:
- **Theorem (Boole, 1847):** Any Boolean function can be represented as a disjunction of conjunctions (one OR of a bunch of ANDs) of its arguments and their negation (NOT). This is also known as **disjunctive normal form** (DNF) 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).

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!

PRACTICE TIME – Using gates

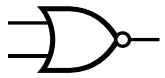
- What is a DNF for this function?

- 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

ANSWER – Using Gates

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

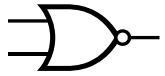
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

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)

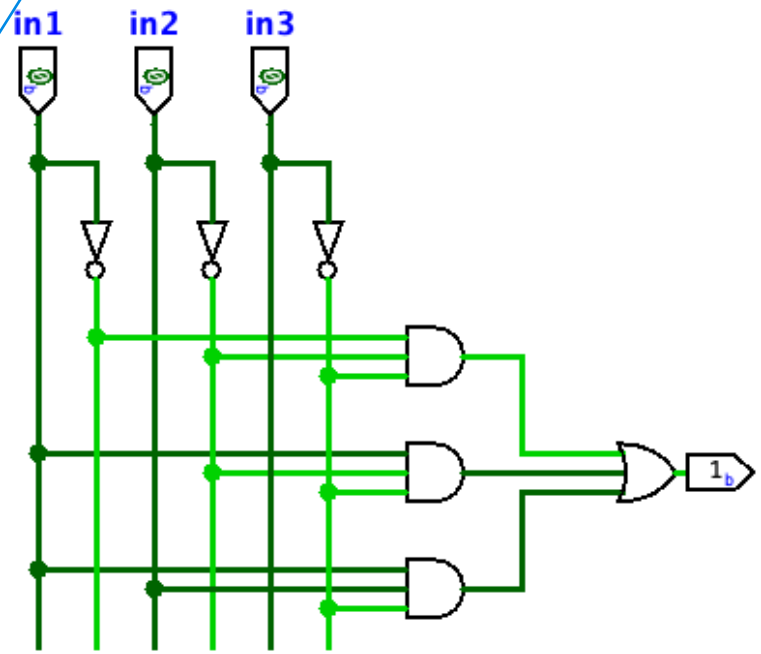
ANSWER – Using Gates

- 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

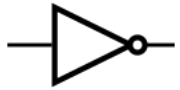
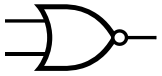
(¬in1 ∧¬in2 ∧ ¬in3) ∨
(in1 ∧ ¬in2 ∧ ¬in3) ∨
(in1 ∧ in2 ∧¬in3)



Light green is 1 and dark green is 0

PRACTICE TIME – Using Gates

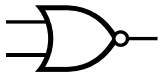
- Can you build a K-map for this table?

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

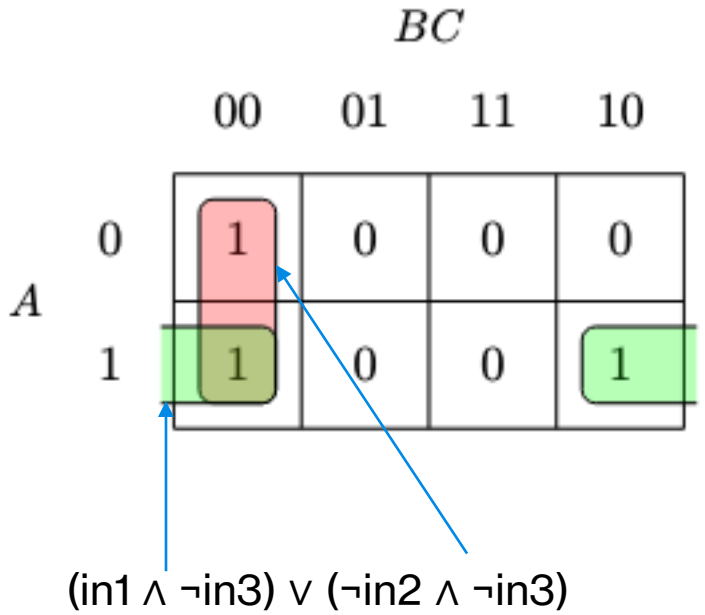
in1	in2	in3	output
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0

ANSWER – Using Gates

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

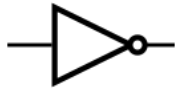
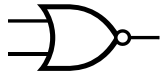
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

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

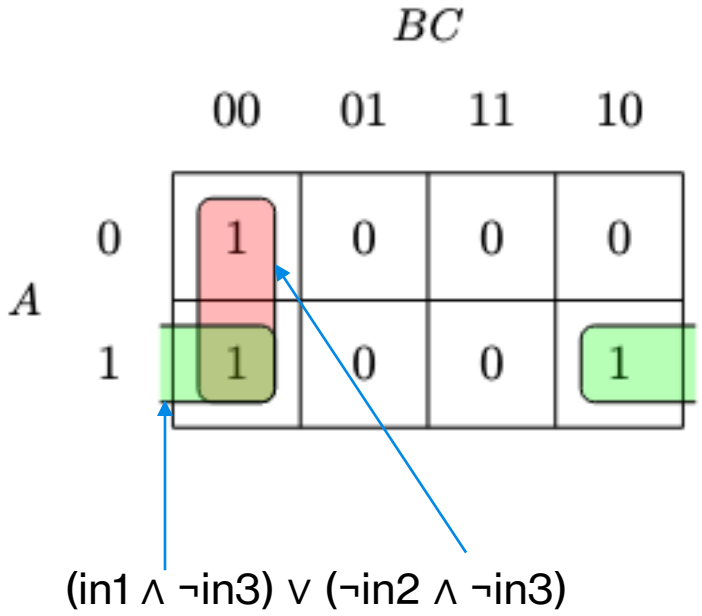


PRACTICE TIME – Using Gates

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

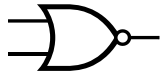
- NOT (¬) 
- AND (∧) 
- OR (∨) 
- XOR (⊕) 
- NAND ($\bar{\wedge}$) 
- NOR ($\bar{\vee}$) 

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

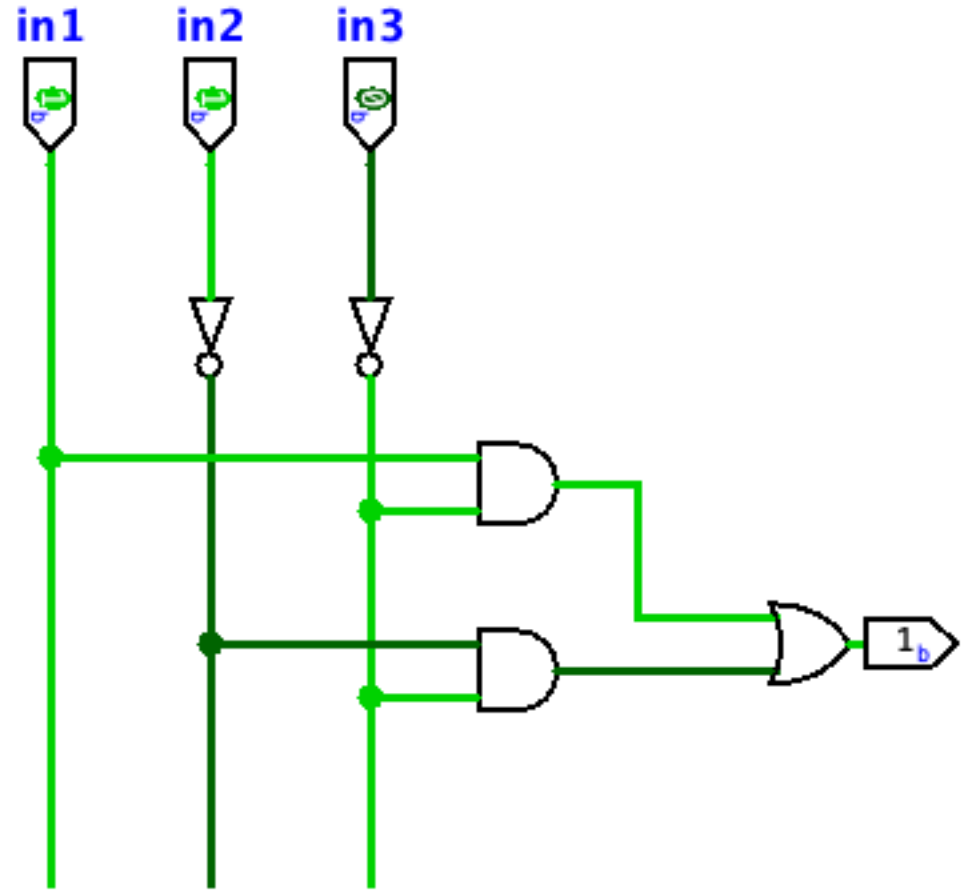


ANSWER – Using Gates

- 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



Arithmetic Circuits

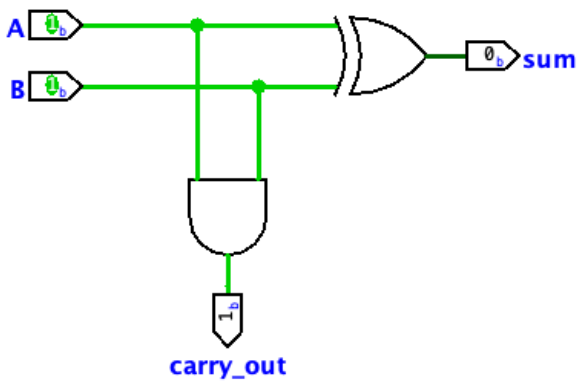
Revisiting addition

- 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 inputs, A and B, it will produce the sum and carry-out bits shown below.
- We can think of the circuit for each output independently.

Inputs		Outputs	
A	B	Carry-Out	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Half adder - circuit to add two bits

Inputs		Outputs	
A	B	carry-out	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

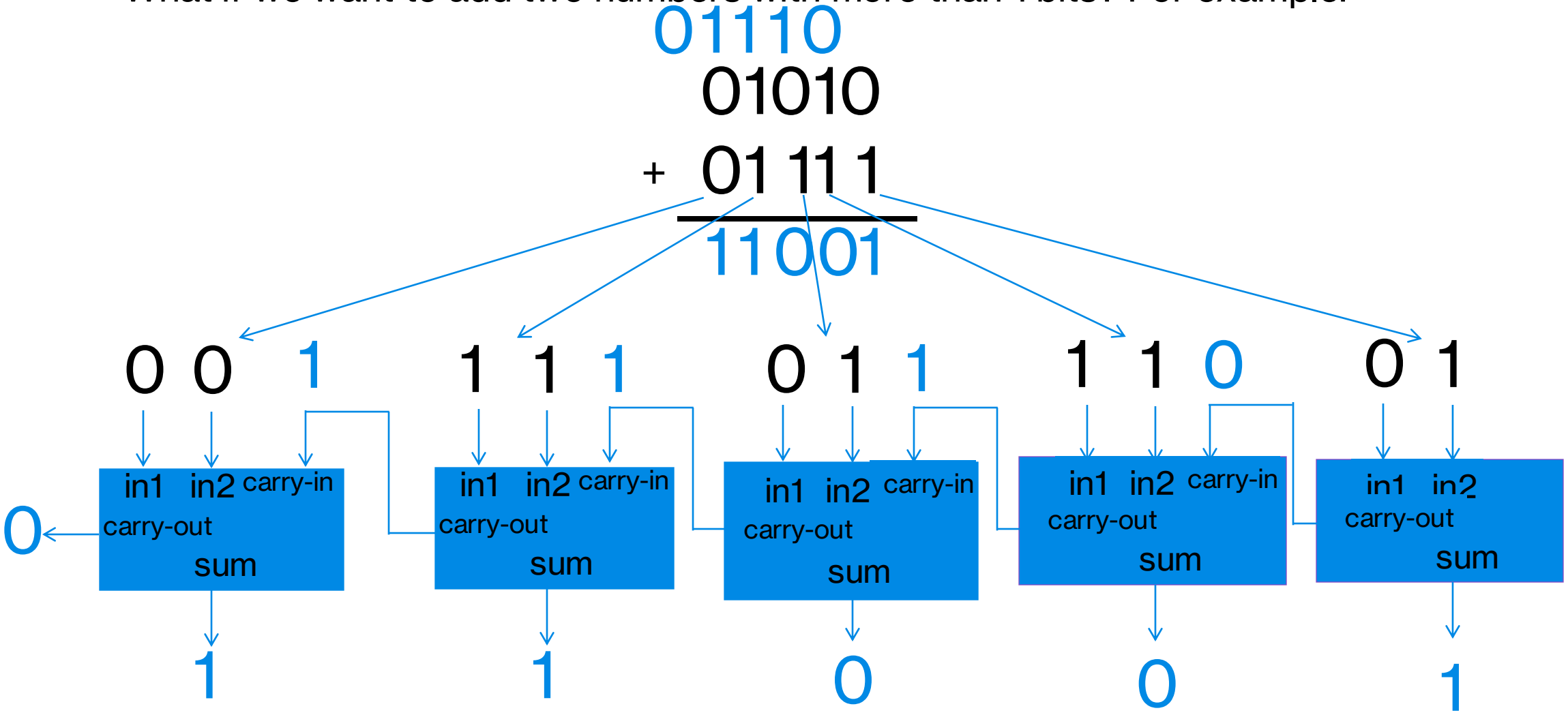


X	Y	¬ X	X ∧ Y	X ∨ Y	X ⊕ Y	X ⊼ Y	X ⊽ Y
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

- A **half adder** is a circuit that computes the sum and carry of adding two bits A and B.
- Carry-out is calculated as A ∧ B. This corresponds to the most-significant bit (MSB) of the result.
- Sum is calculated as A ⊕ B. This corresponds to the least-significant bit (LSB) of the result.

Revisiting addition

- What if we want to add two numbers with more than 1 bits? For example:



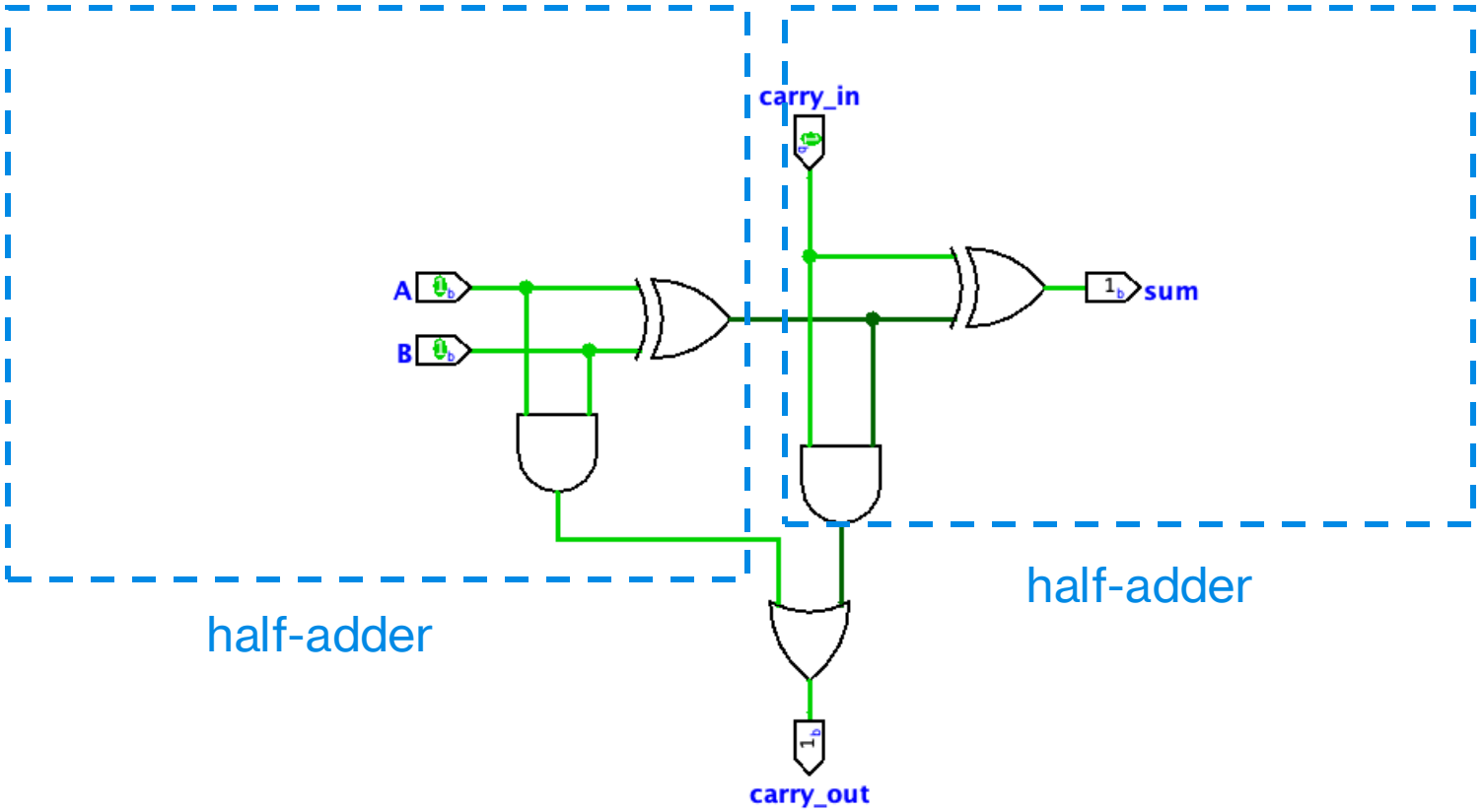
Full adder - circuit to add two bits and a carry-in

- Two half-adders can be combined to form a **full adder**.
- A full adder corresponds to one column of the addition.
- The inputs are the bits A and B and the carry-in bit from the previous column.
- The outputs are the sum bit and the carry-out bit to the next column.

Inputs			Outputs	
A	B	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

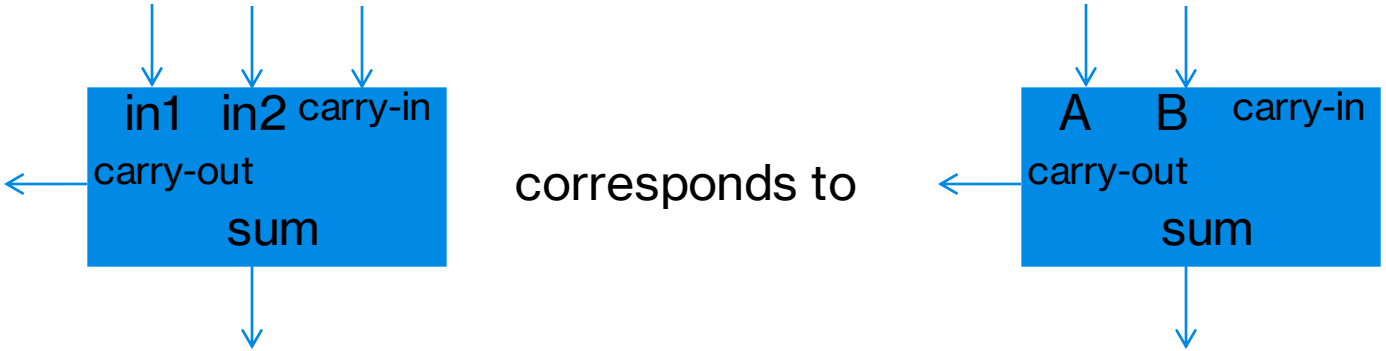
Implementing a full adder

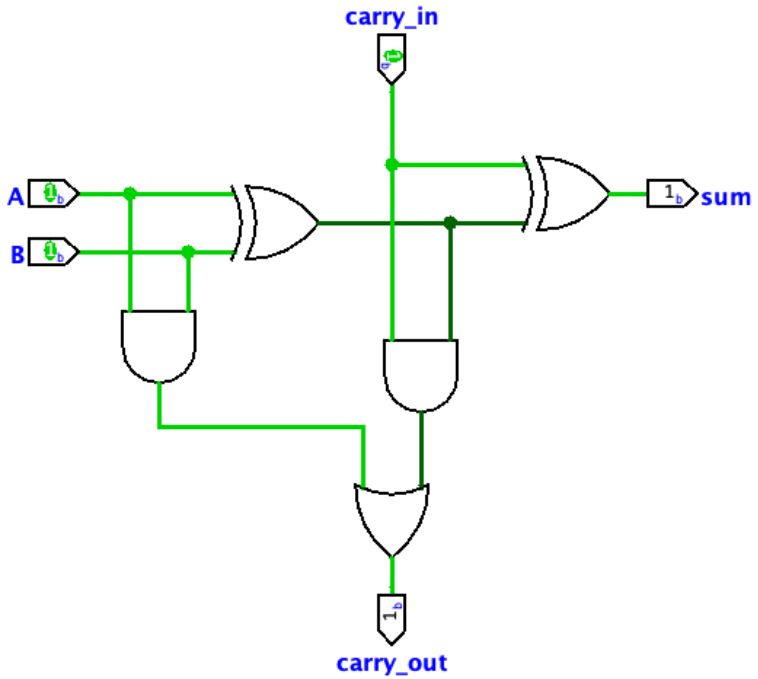
- The sum is calculated as (A ⊕ B) ⊕ carry-in.
- The carry-out is calculated as (A ∧ B) ∨ ((A ⊕ B) ∧ carry-in).



Inputs			Outputs	
A	B	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

Implementing a full adder

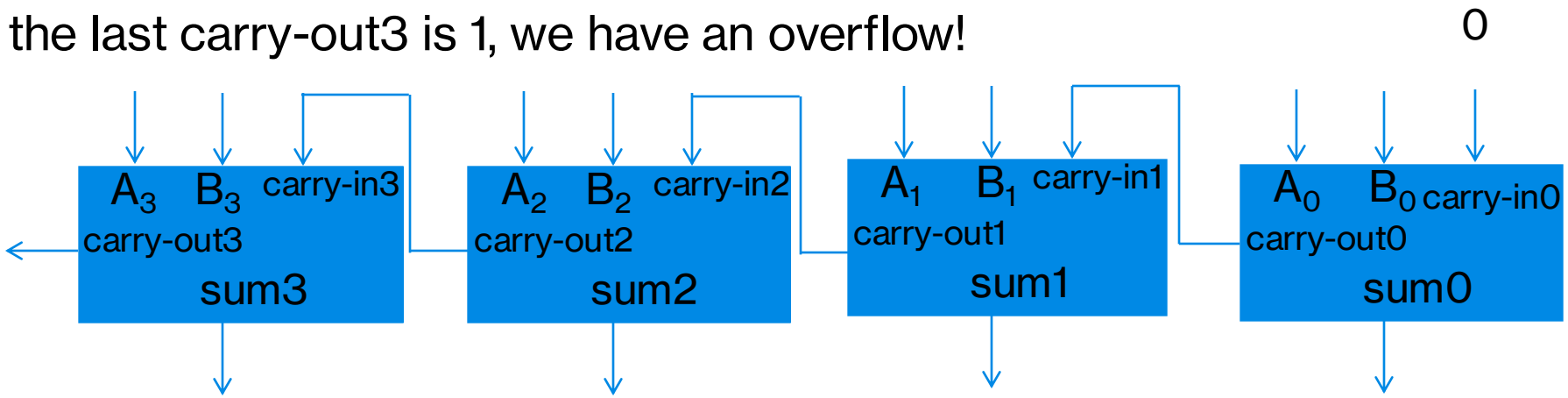




Inputs			Outputs	
A	B	carry-in	sum	carry-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

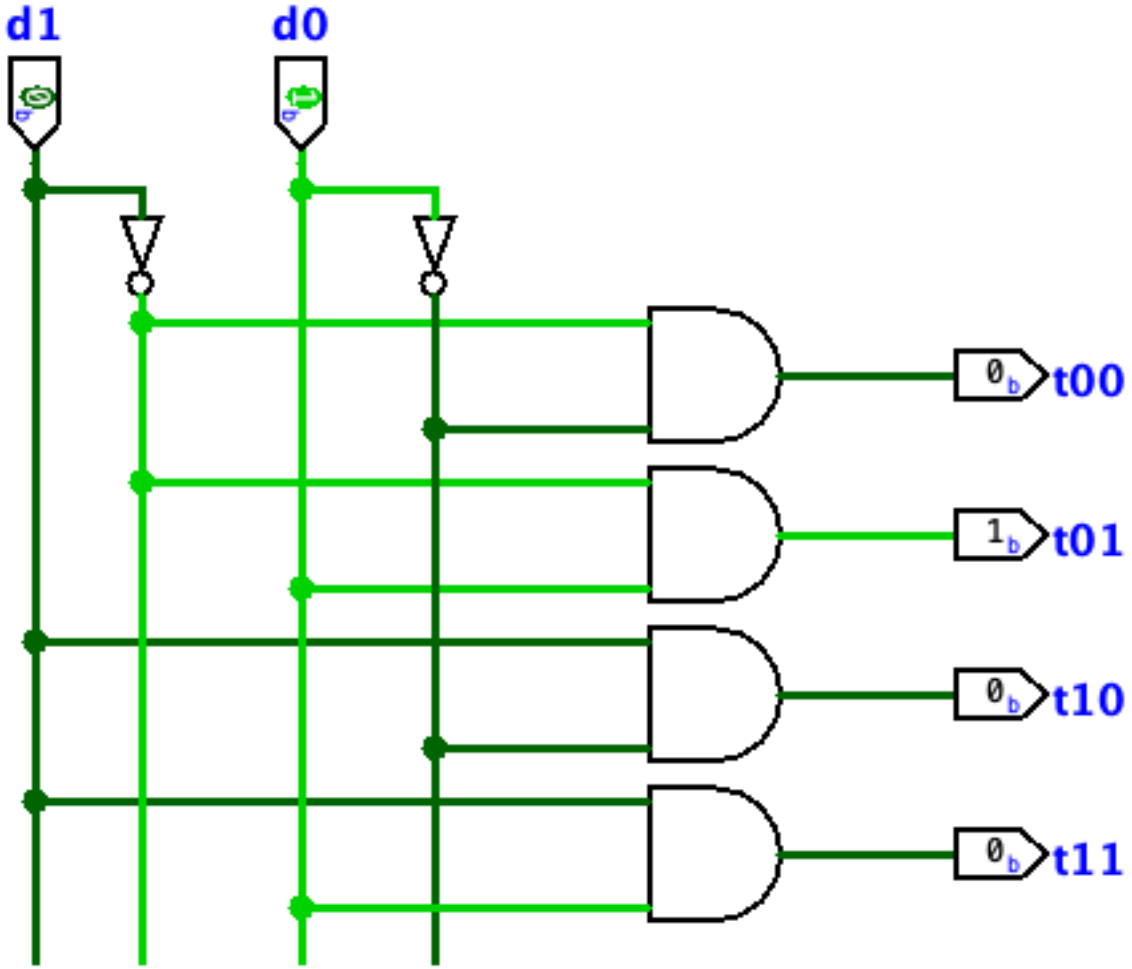
Ripple-carry adder

- 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.
- If the last carry-out3 is 1, we have an overflow!



Decoders

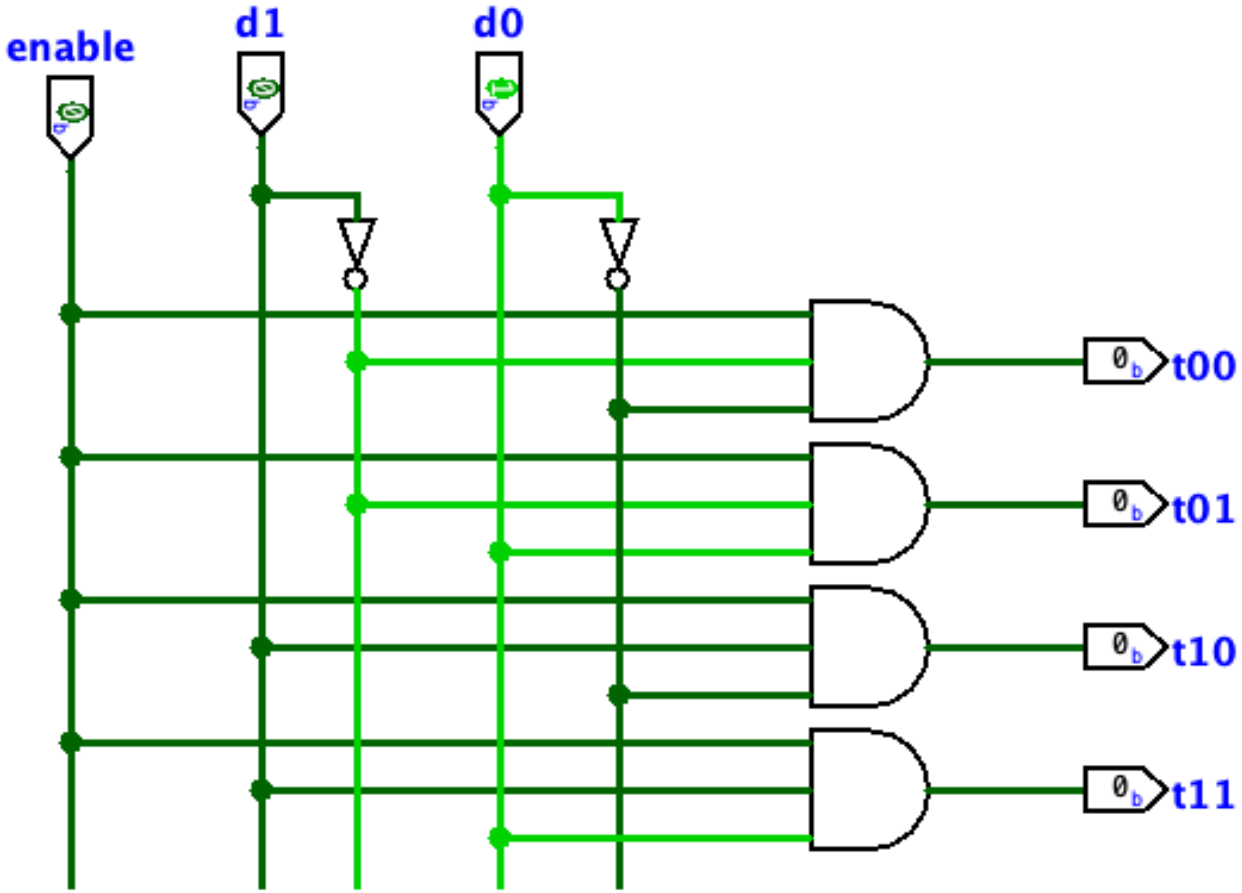
2-bit decoders



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.
- For example, if the input value $d_1 d_0$ is 01 then only the output t_{01} is 1.
- You can think of them as a binary to unary converter.

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

- We can add 1 more wire for an enable input.
- When enable is on (1) the turned on and works as usual. When enable is off (0), the decoder is off.

k-bit decoders

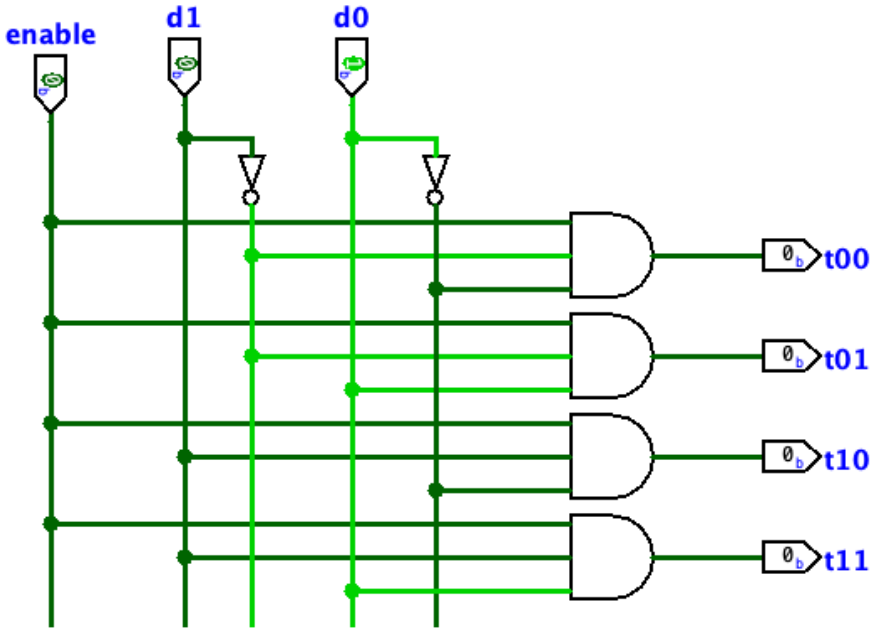
- We can generalize the idea of 2-bit decoders to k-bit decoders. There will now be k inputs and up to 2^k outputs.
- Exactly one of the 2^k outputs will be 1 for a given row in the truth table. This is known as "1-hot." Which one is determined by the value, in binary, of the k inputs.
- For example, here is the truth table for a 3-bit decoder.

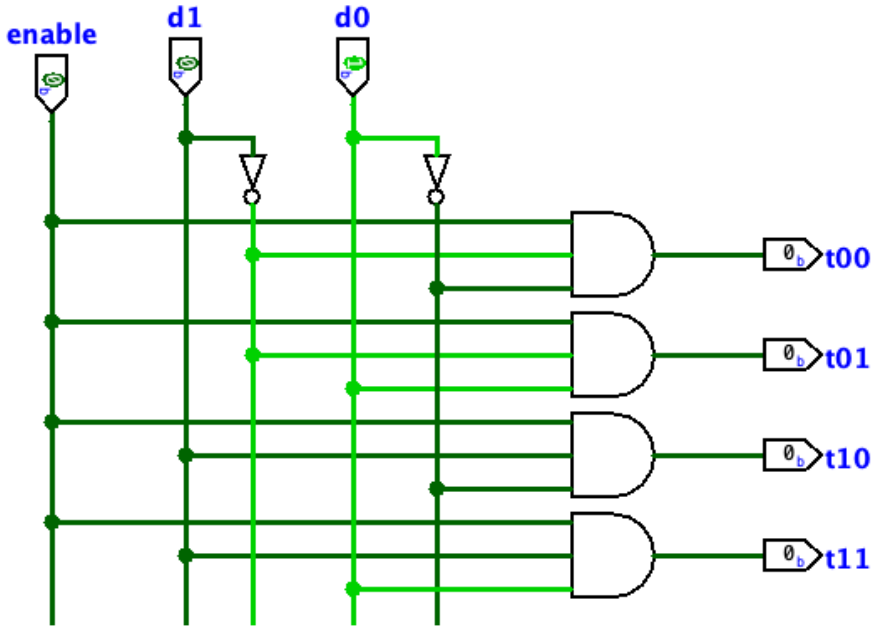
Inputs			Outputs							
d_2	d_1	d_0	t_{000}	t_{001}	t_{010}	t_{011}	t_{100}	t_{101}	t_{110}	t_{111}
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Building a 3-bit decoder

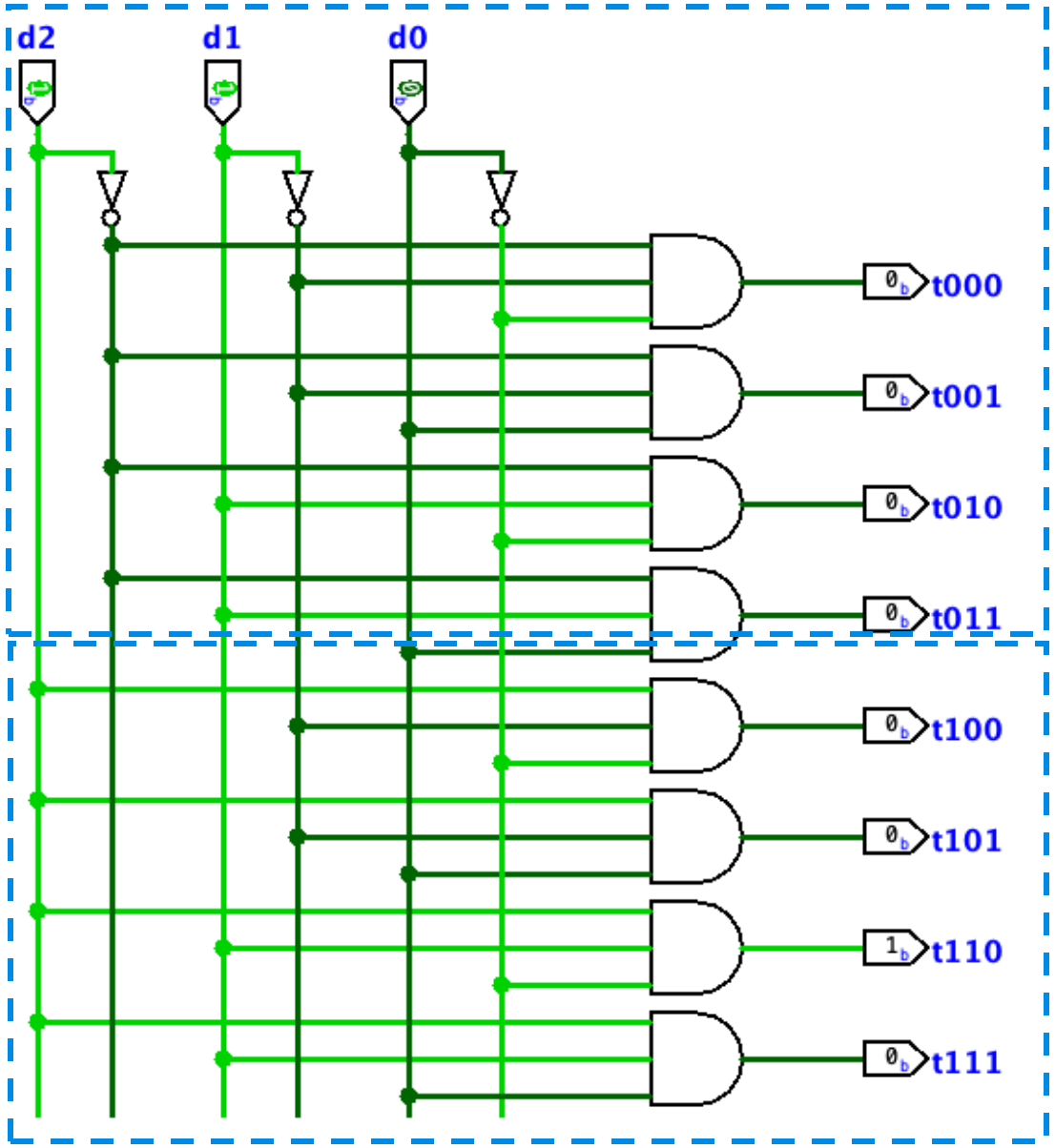
- We could design a circuit for scratch, but instead we will reuse 2-bit decoders.
- We will also use the idea of the enable input.

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





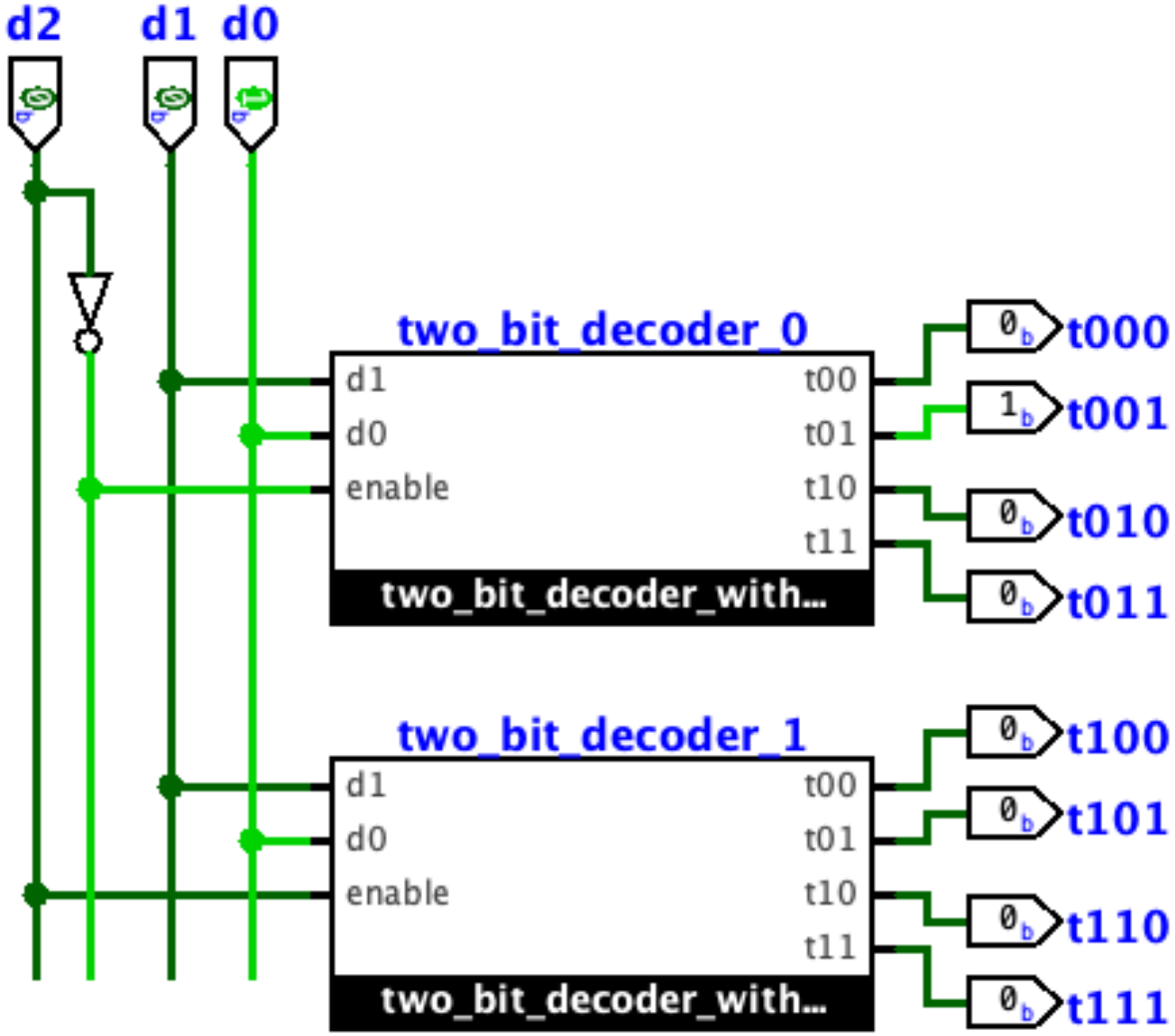
Building a 3-bit decoder



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

Abstraction in decoders

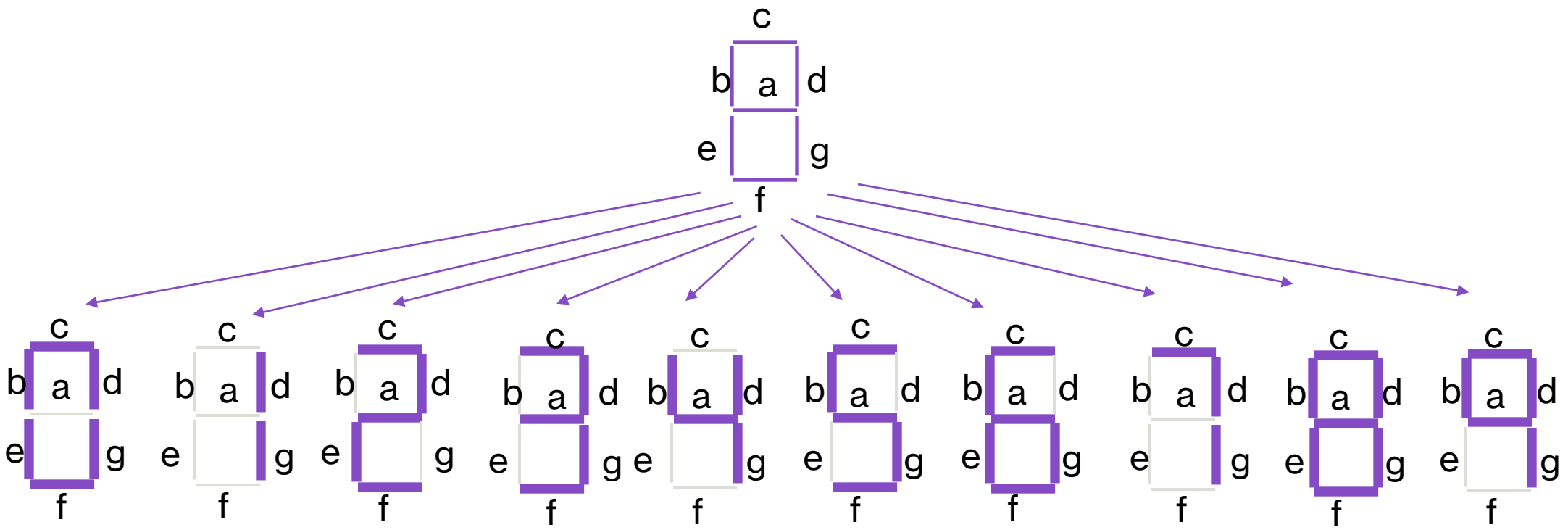


Inputs			Outputs							
d_2	d_1	d_0	t_{000}	t_{001}	t_{010}	t_{011}	t_{100}	t_{101}	t_{110}	t_{111}
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

- 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

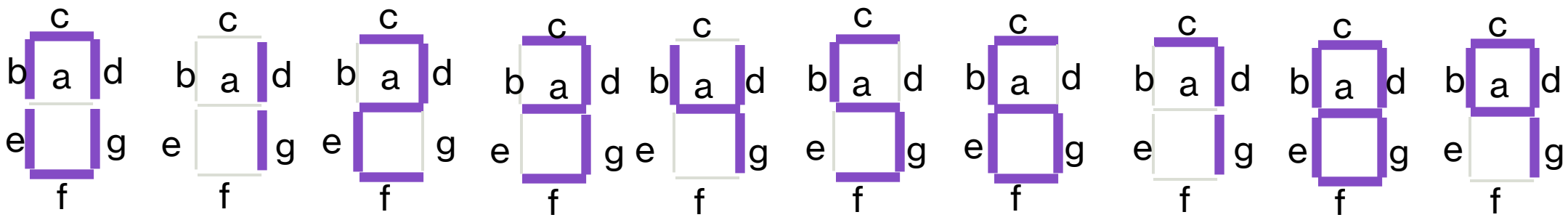
7-Segment Display Decoders

Decoding 7-segment displays



- 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.

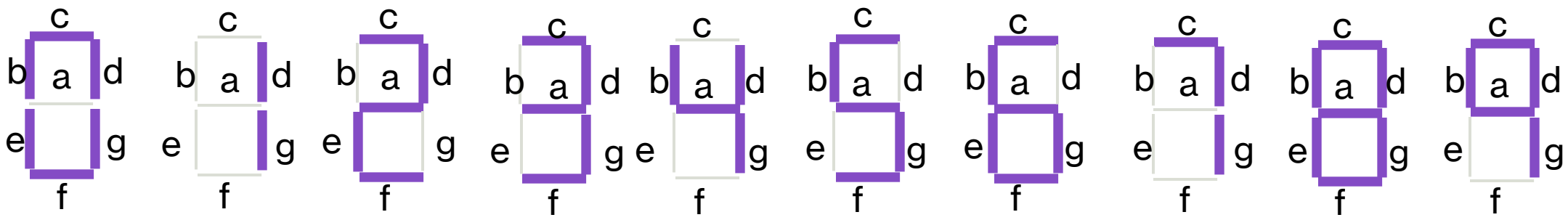
Decoding 7-segment displays in clocks



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

- 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.
- Let's see how we can use this information to build a 7-segment display in Logisim-Evolution.

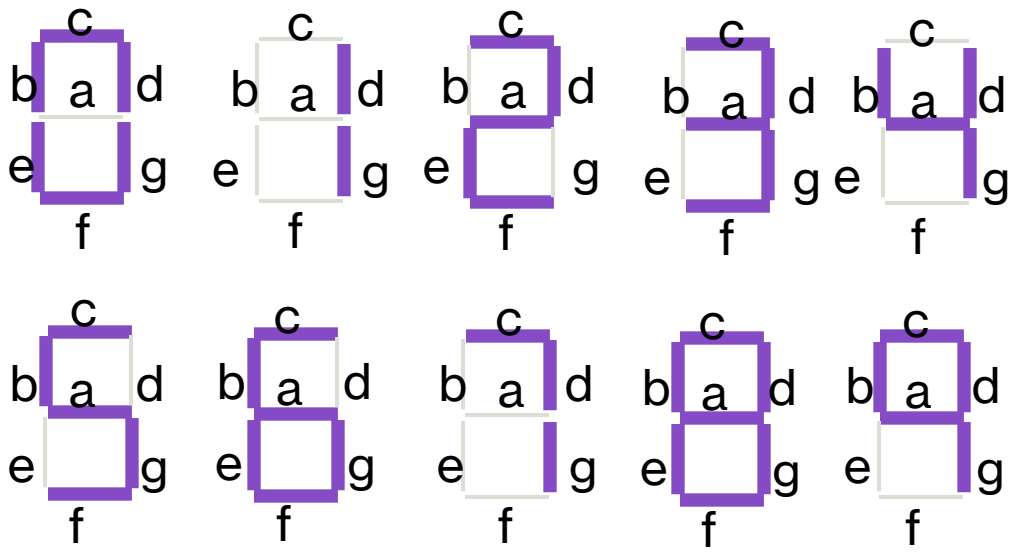
Building a 7-segment display in Logisim



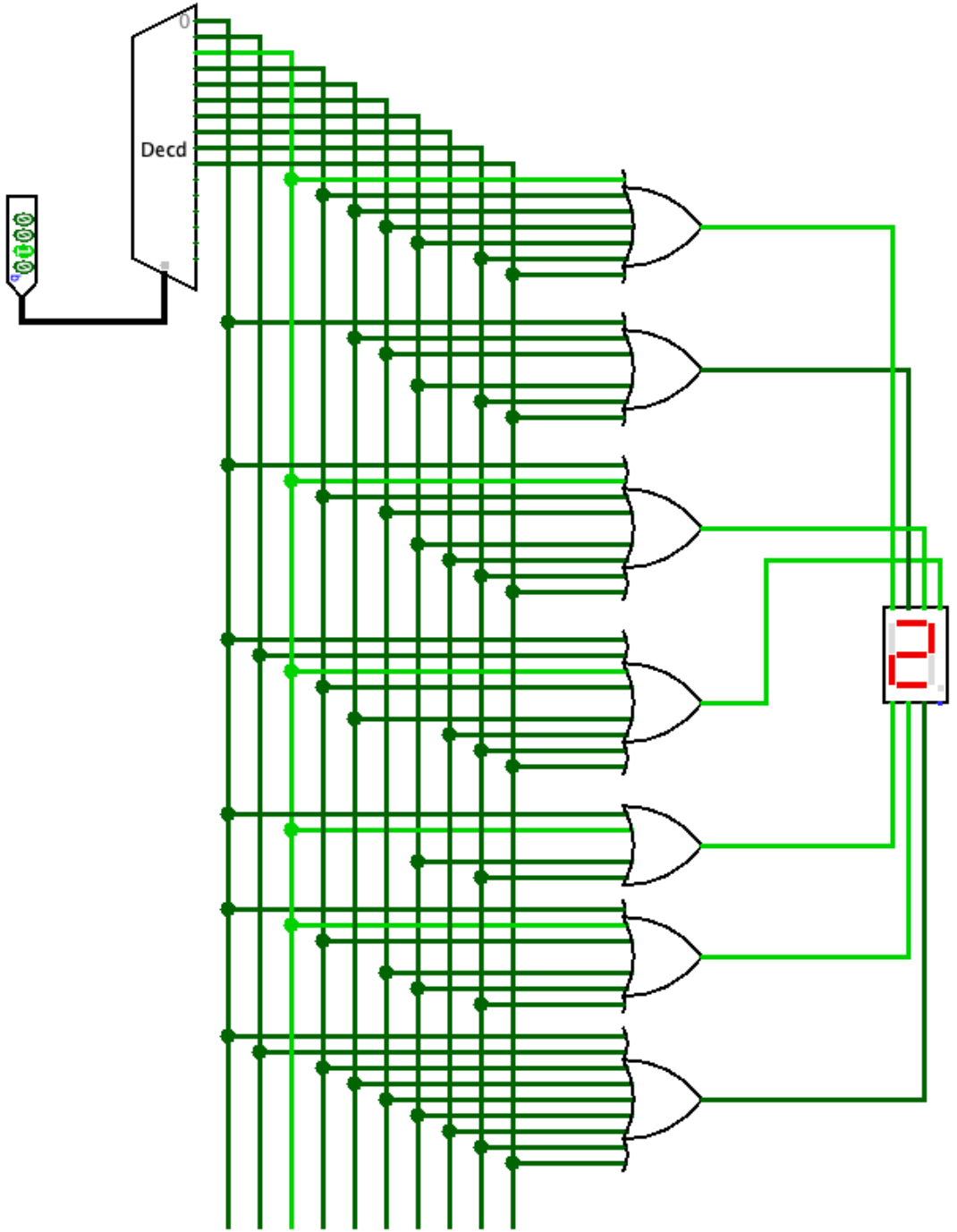
Digit	Illuminated segments						
	a	b	c	d	e	f	g
0		x	x	x	x	x	x
1				x			x
2	x		x	x	x	x	
3	x		x	x		x	x
4	x	x		x			x
5	x	x	x			x	x
6	x	x	x		x	x	x
7			x	x			x
8	x	x	x	x	x	x	x
9	x	x	x	x			x

- Since we have 10 digits, we need 4 bits of input to encode them.
- We will use a 4-bit decoder, utilizing only the first 10 outputs (0-9) and ignoring the last 6 (10-15).
- 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, 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.

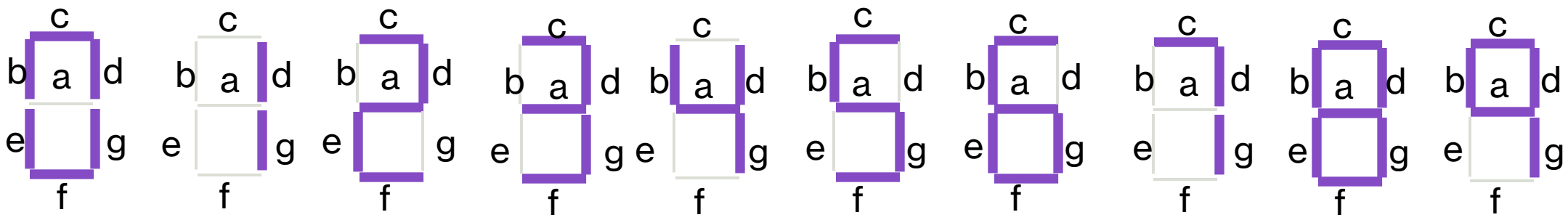
A 7-segment display in Logisim



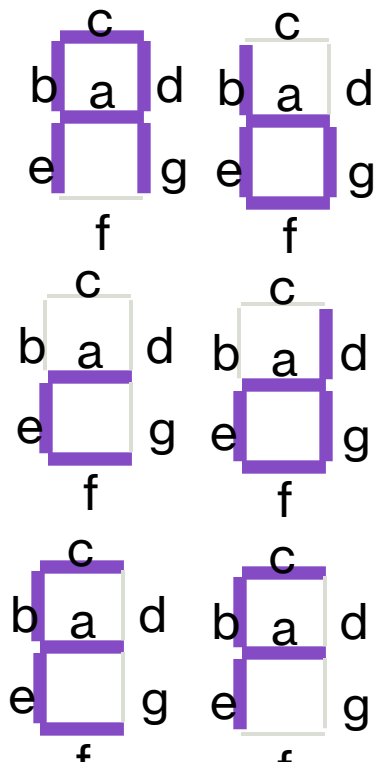
Input	Digit	Illuminated segments						
		a	b	c	d	e	f	g
0000	0		x	x	x	x	x	x
0001	1				x			x
0010	2	x		x	x	x	x	
0011	3	x		x	x		x	x
0100	4	x	x		x			x
0101	5	x	x	x			x	x
0110	6	x	x	x		x	x	x
0111	7			x	x			x
1000	8	x	x	x	x	x	x	x
1001	9	x	x	x	x			x



PRACTICE TIME – Expanding to 16 characters



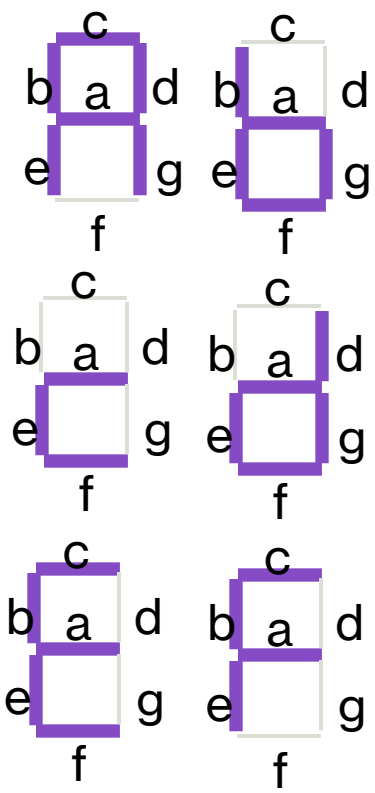
- Imagine that the 7-segment display we have built so far is expanded to include 16 characters, the existing 10 digits, and A-F, as depicted here to differentiate among them.
- Expand the table on the right to reflect all 16 characters.
- How will the design of the circuit change on Logisim-Evolution?



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

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		

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).

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