

Computer Organization and Networks

(INB.06000UF, INB.07001UF)

Chapter 1 – Combinational Circuits

Winter 2022/2023



Stefan Mangard, www.iaik.tugraz.at

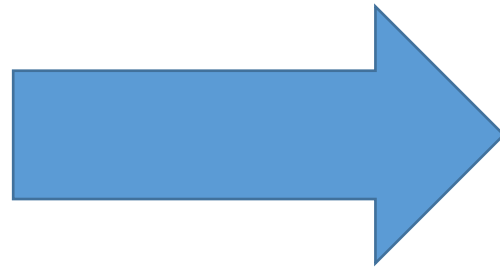
Computation and Physics

We Need to Map our Programs to Physics

1 + 1 = ?

```
include <stdio.h>

int main()
{
    printf("Hello World");
    return 0;
}
```



- Mechanics
- Voltage
- Current
- Quantum Mechanics
- ...

Examples of Computation Machines

Enigma

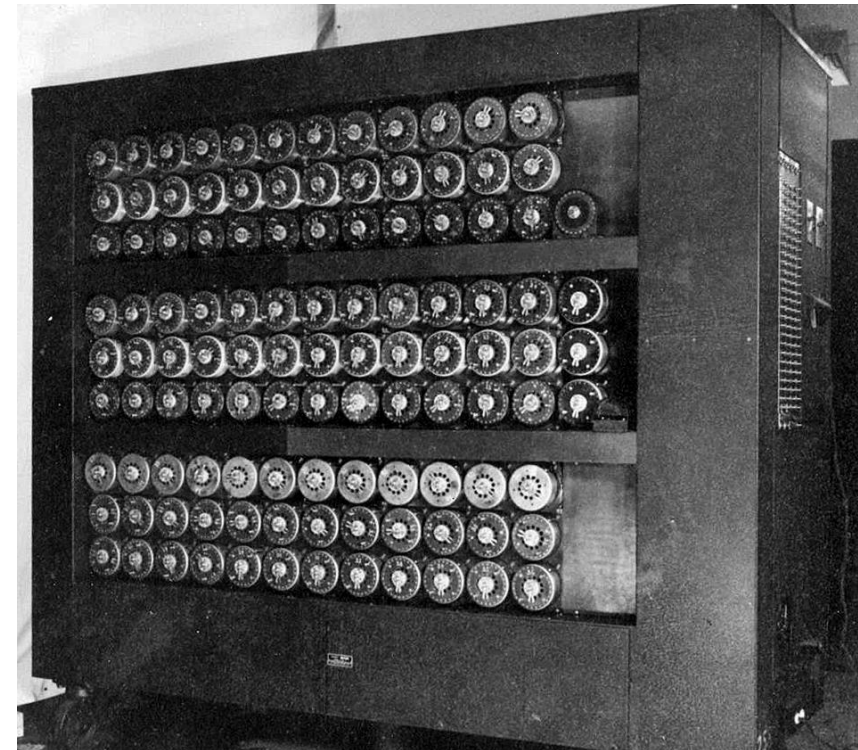
an electromechanical encryption machine



Museo della Scienza e della Tecnologia "Leonardo da Vinci" CC BY-SA

„British Bombe“ by Alan Turing

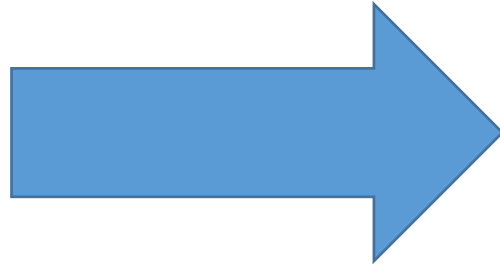
An electromechanical machine to break Enigma



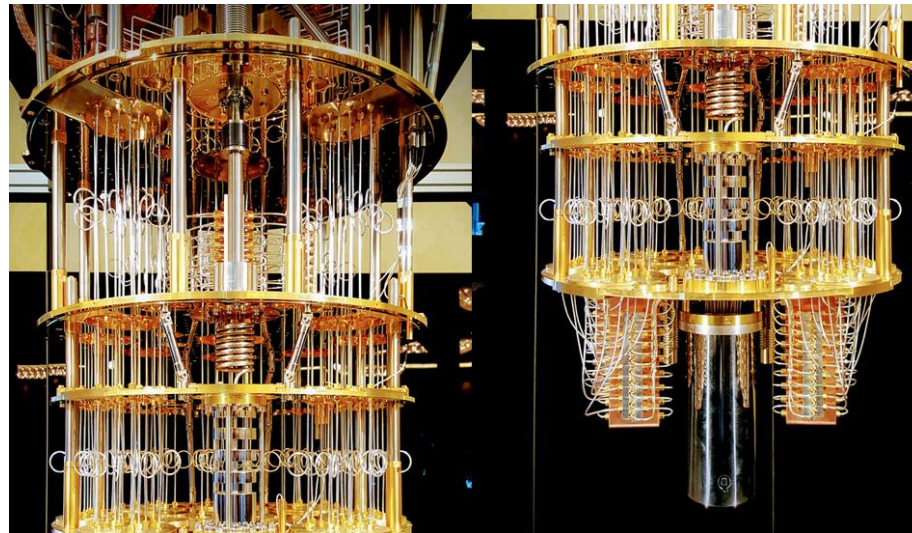
We Need to Map our Programs to Physics

1 + 1 = ?

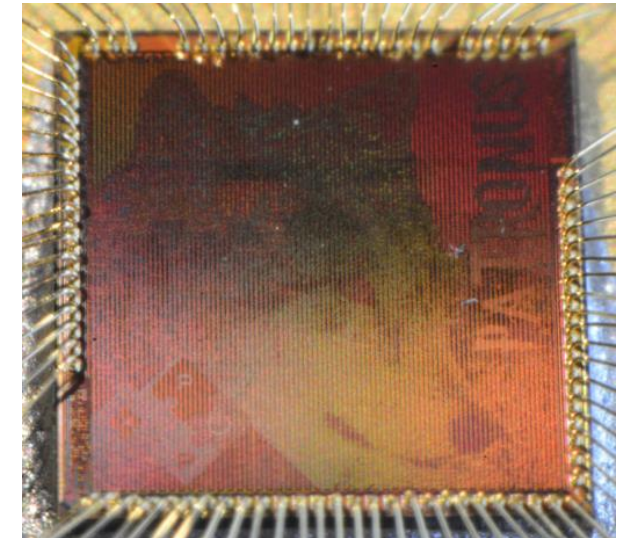
```
include <stdio.h>\n\nint main()\n{\n    printf("Hello World");\n    return 0;\n}
```



Zuse Z1 (mechanical)
ComputerGeek via Wikipedia CC BY-SA 3.0



IBM quantum computer
(Lars Plougmann via flickr CC BY-SA 2.0)

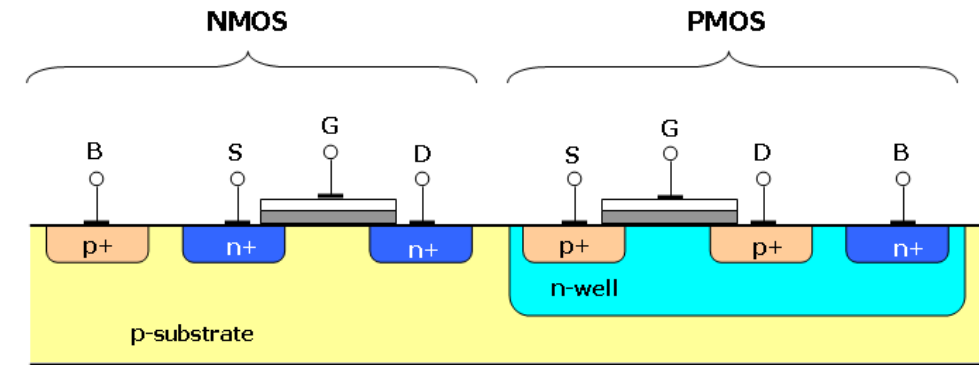


CMOS Processor
(<http://asic.ethz.ch>)

Complementary Metal-Oxide-Semiconductor (CMOS)

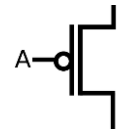
- Invented at Bell Labs by Mohamed Atalla and Dawon Kahng in 1959

- CMOS uses PMOS and NMOS transistors



- CMOS is the technology of almost all digital circuits (from contactless RFID chips to server CPUs)

Two Types of Transistors

 PMOS transistor: is conducting, if A is connected to GND

 NMOS transistor: is conducting, if A is connected to Vdd

- PMOS and NMOS transistors are essentially switches
 - PMOS: $A=0 \rightarrow$ switch is open; $A=1 \rightarrow$ switch is closed
 - NMOS: $A=0 \rightarrow$ switch is closed; $A=1 \rightarrow$ switch open
- **How do we build a computer from these two types of transistors?**

We Need Two Things From Transistors

- **Computation (Combinational Logic)**

- How to apply a function on input data to generate an output?

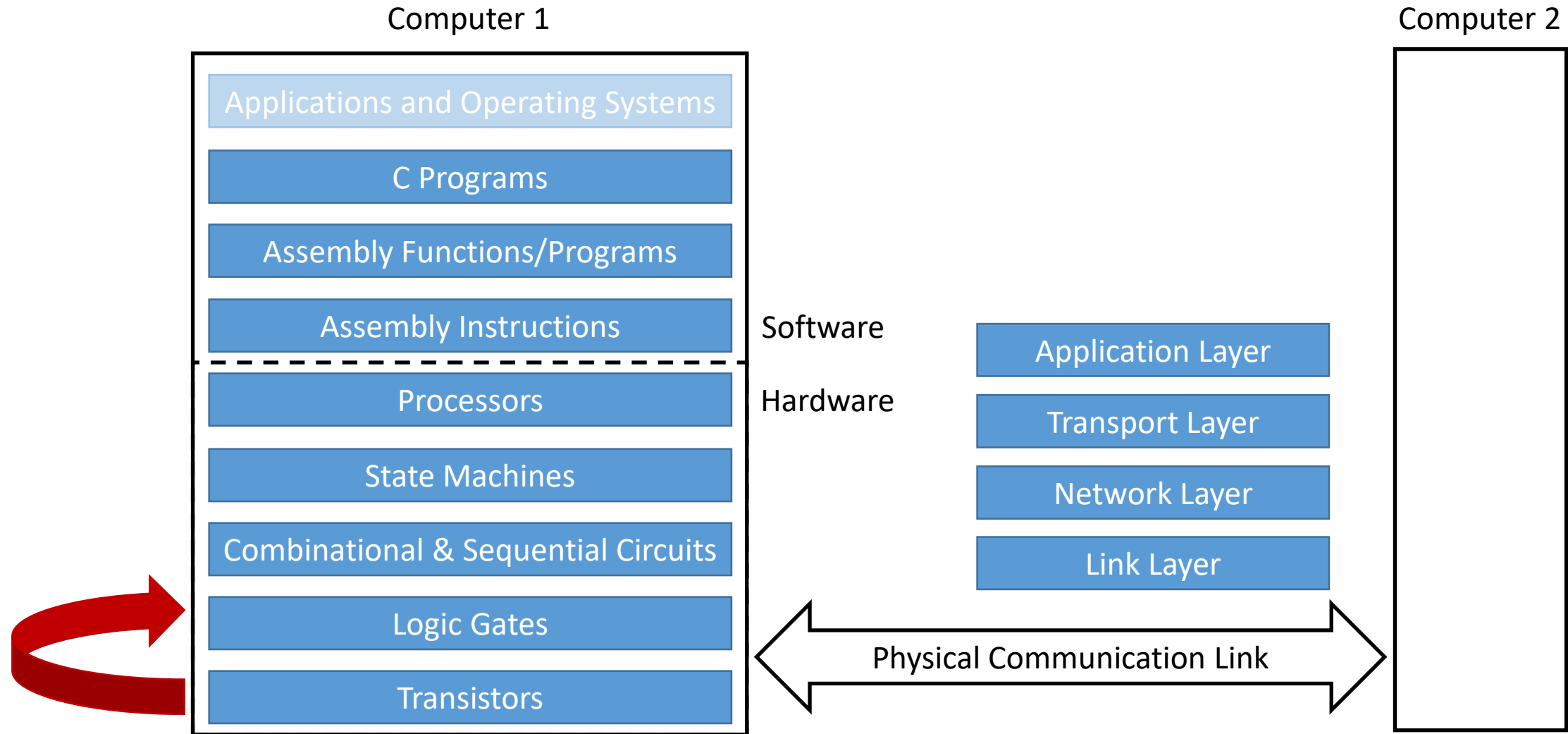
- **Storage (Sequential Logic)**

- How to store data and intermediate results?

Logic Gates

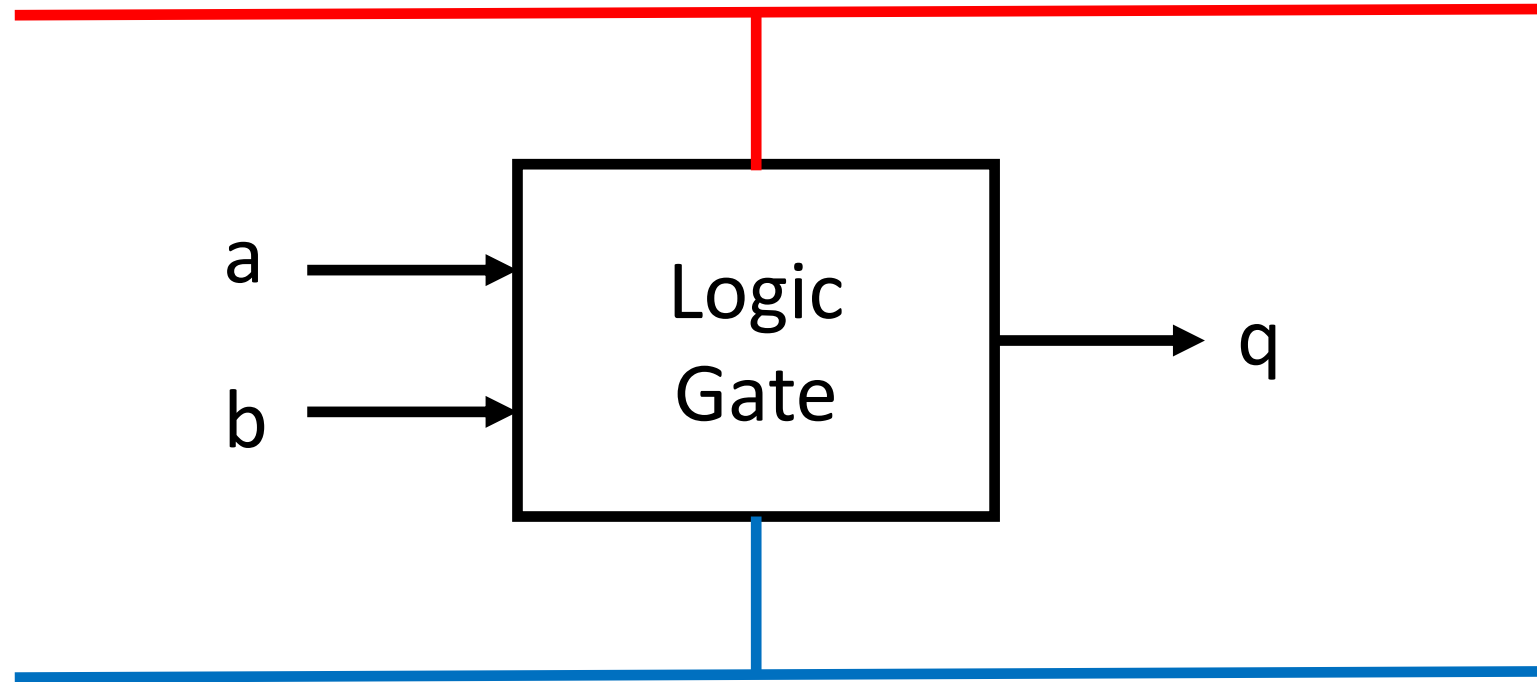
(The basis)

The Big Picture




A Logic Gate – “The Smallest Functional Unit”

(“Vdd”, “high”, “1”)



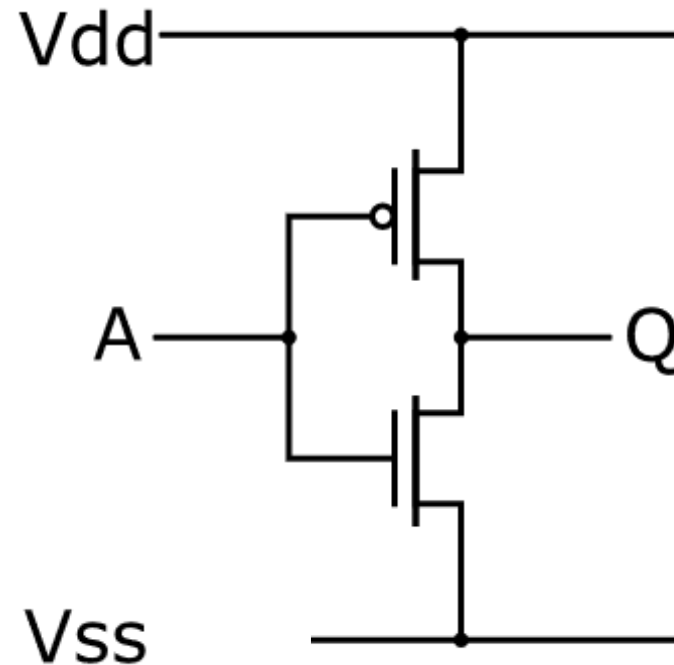
(“GND”, “Vss”, “low”, “0”)

The Simplest Gate – A CMOS Inverter

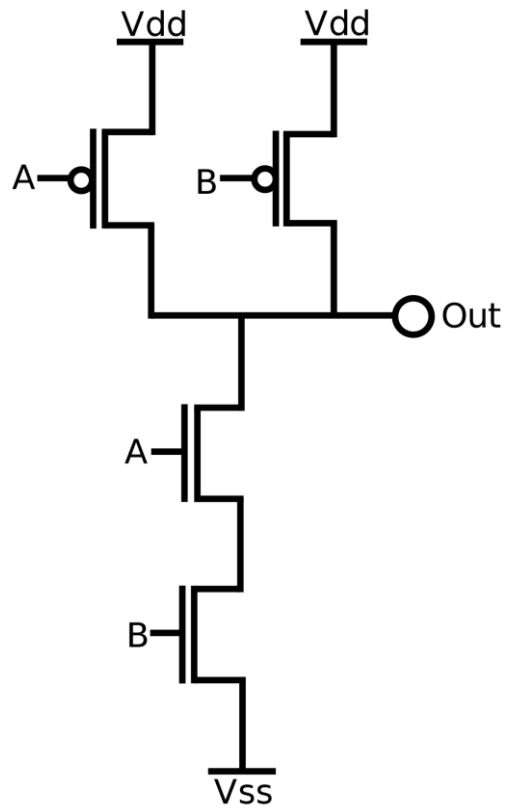
 PMOS transistor: is conducting, if A is connected to GND

 NMOS transistor: is conducting, if A is connected to Vdd

A	Q
High (1)	Low (0)
Low (0)	High (1)

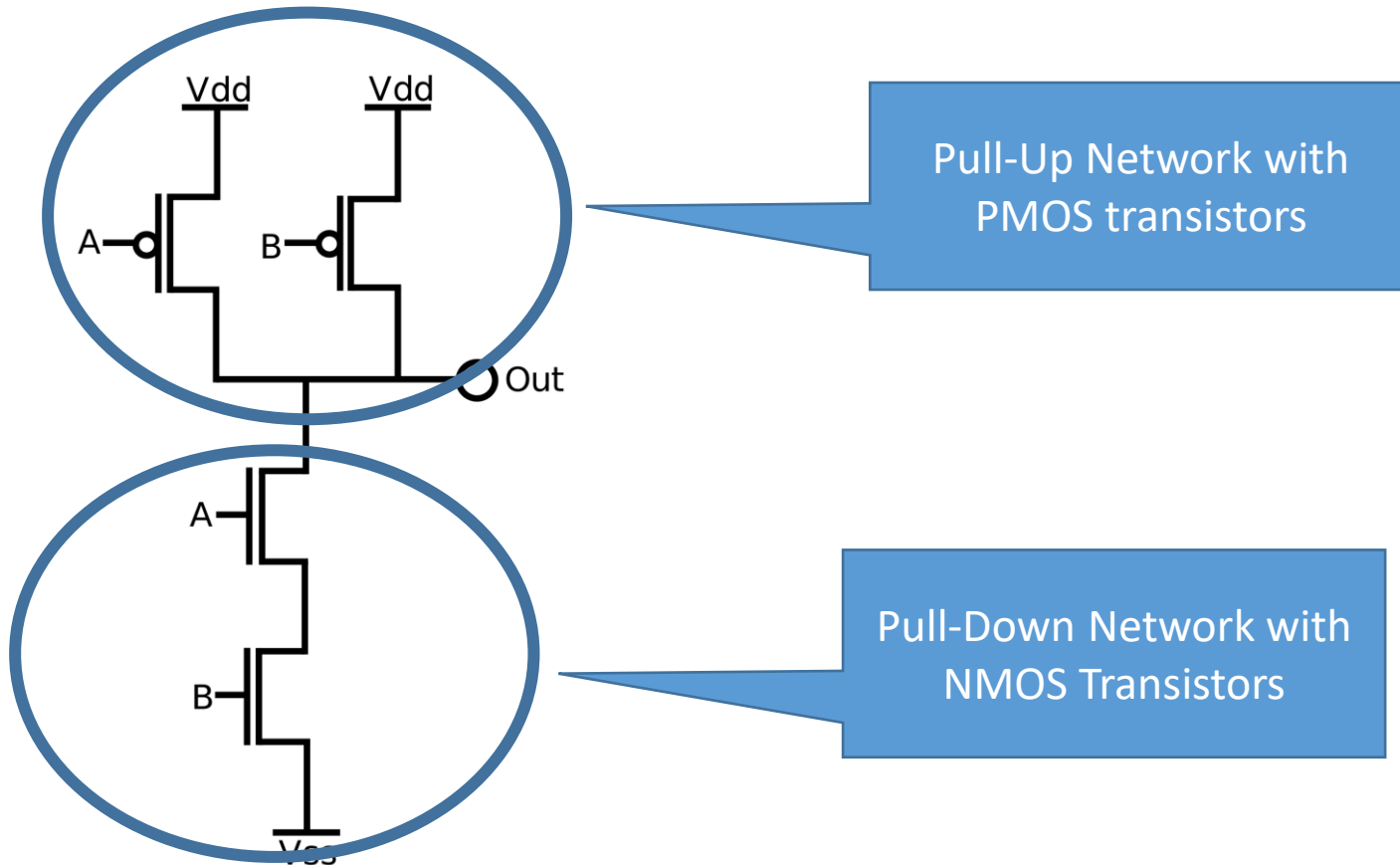


CMOS NAND gate



A	B	Out
Low (0)	Low (0)	High (1)
Low (0)	High (1)	High (1)
High (1)	Low (0)	High (1)
High (1)	High (1)	Low (0)

CMOS Design Principle



Pull-Up and Pull-Down networks are complementary

→ given static inputs, the output is either pulled up or pulled down

Based on this principle, different logic gates can be built

CMOS NOR gate

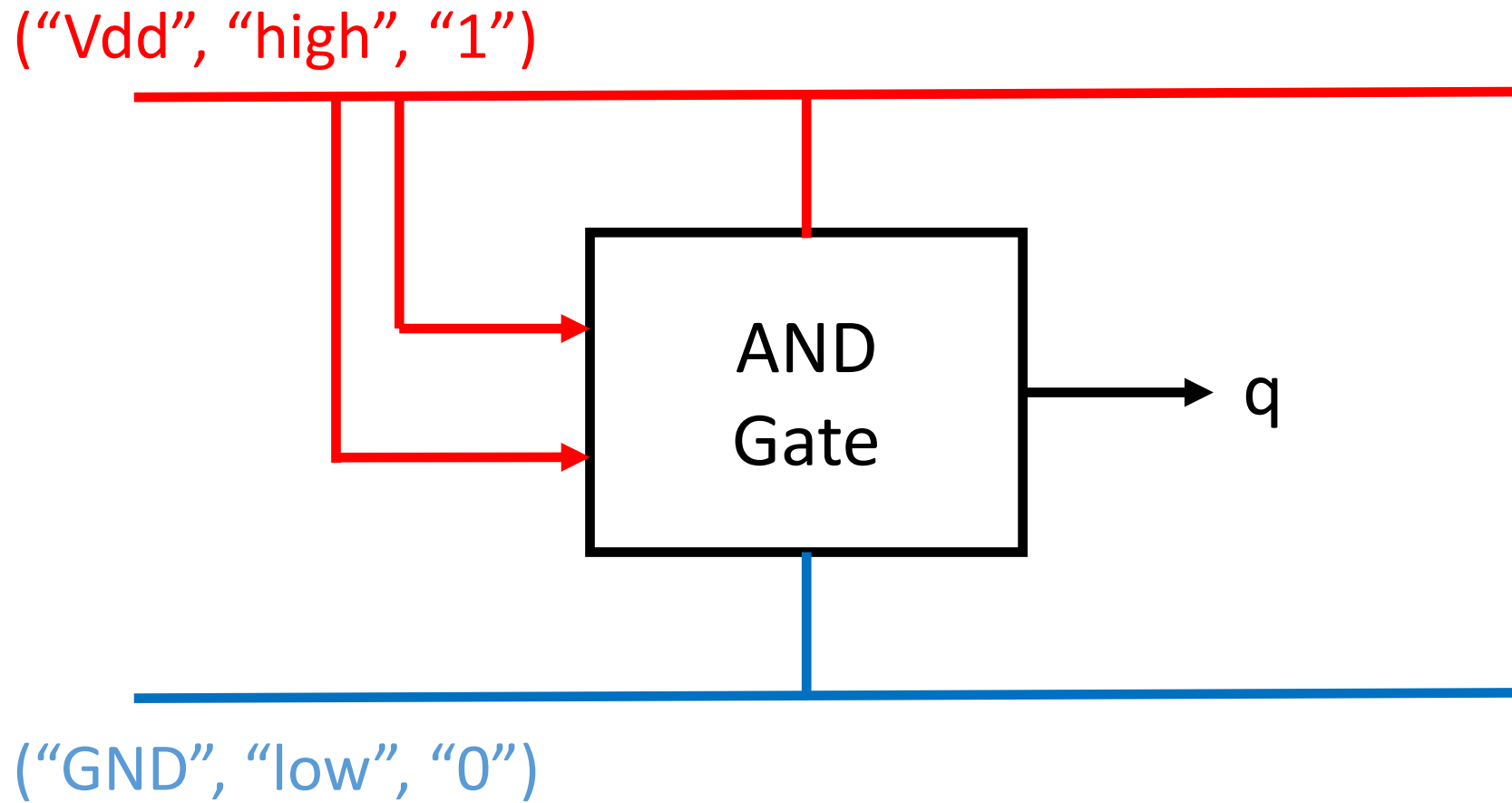
A	B	Out
Low (0)	Low (0)	High (1)
Low (0)	High (1)	Low (0)
High (1)	Low (0)	Low (0)
High (1)	High (1)	Low (0)

More Complex Gates

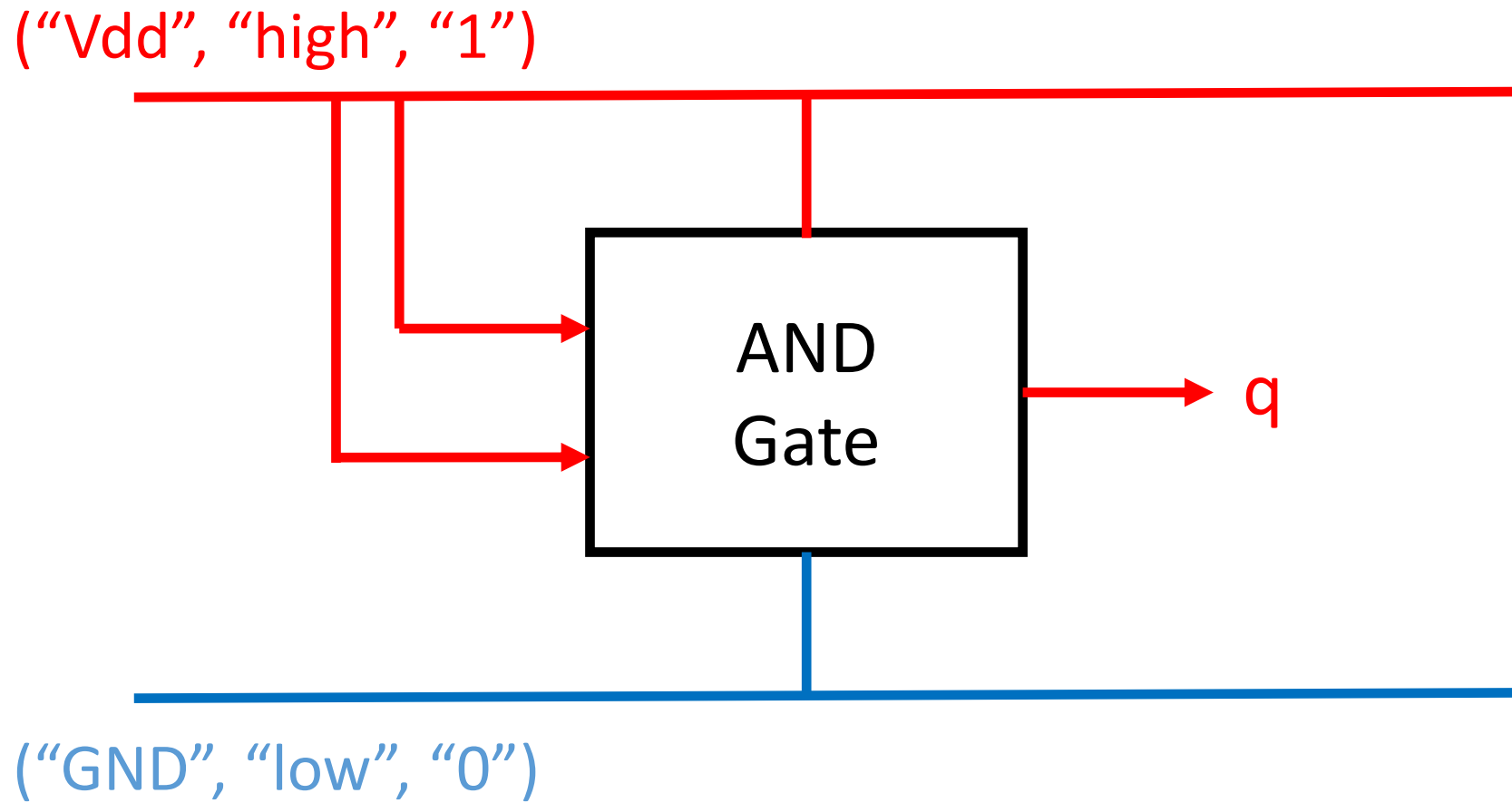
Building an AND Gate

- An AND gate cannot be built using a single pull-up/pull-down network
- It is built by a NAND gate followed by an inverter

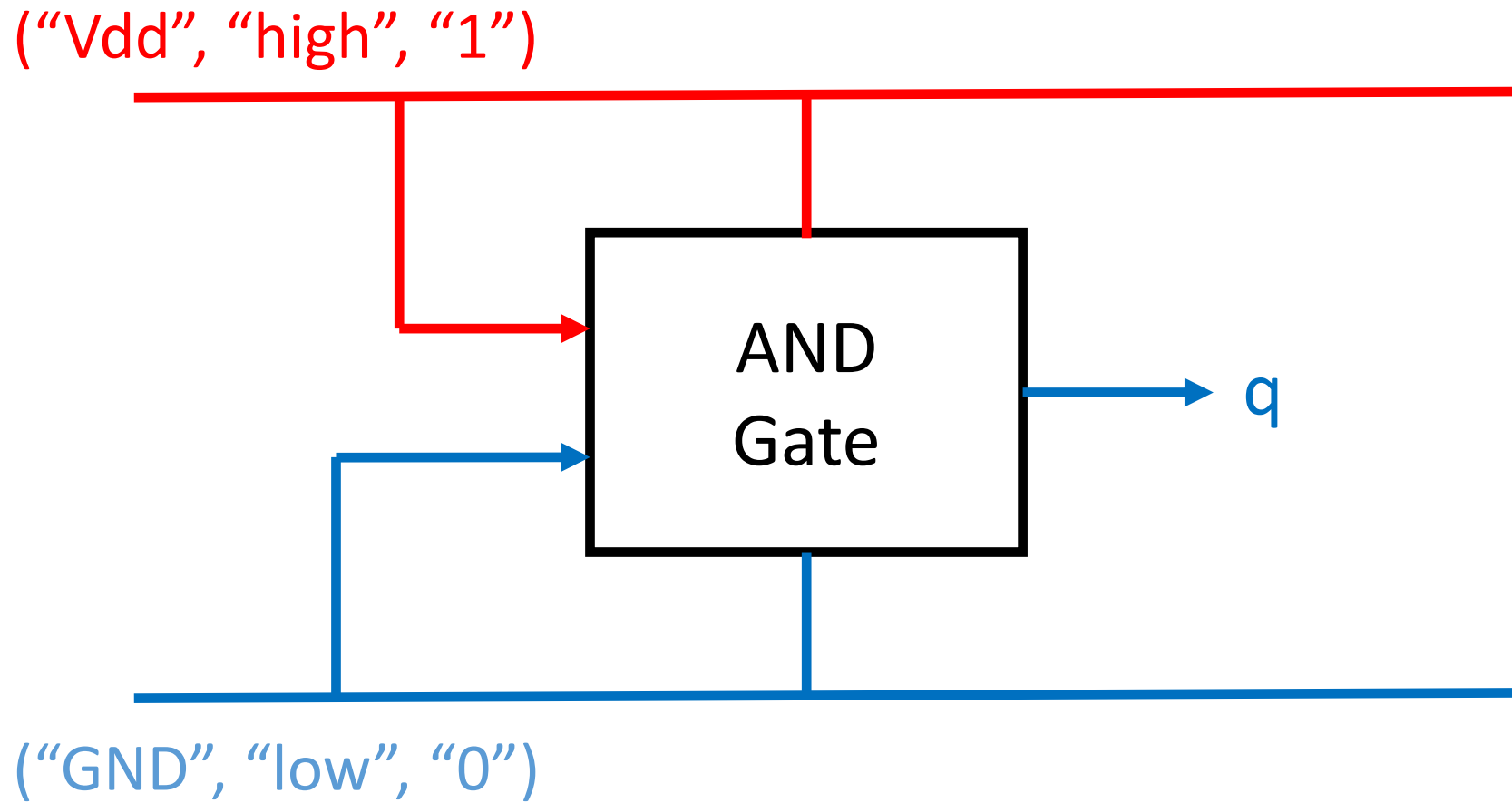
AND Gate



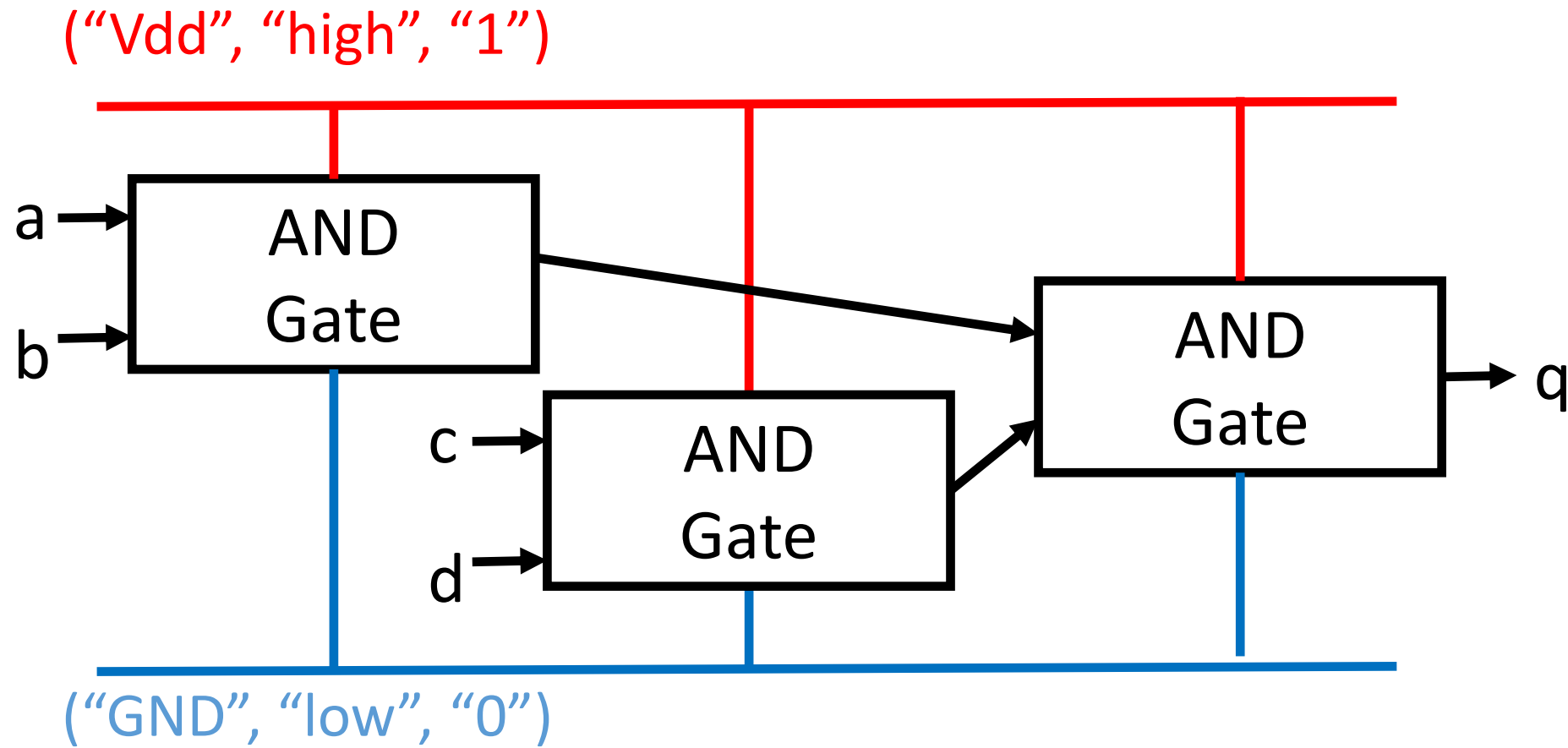
AND Gate



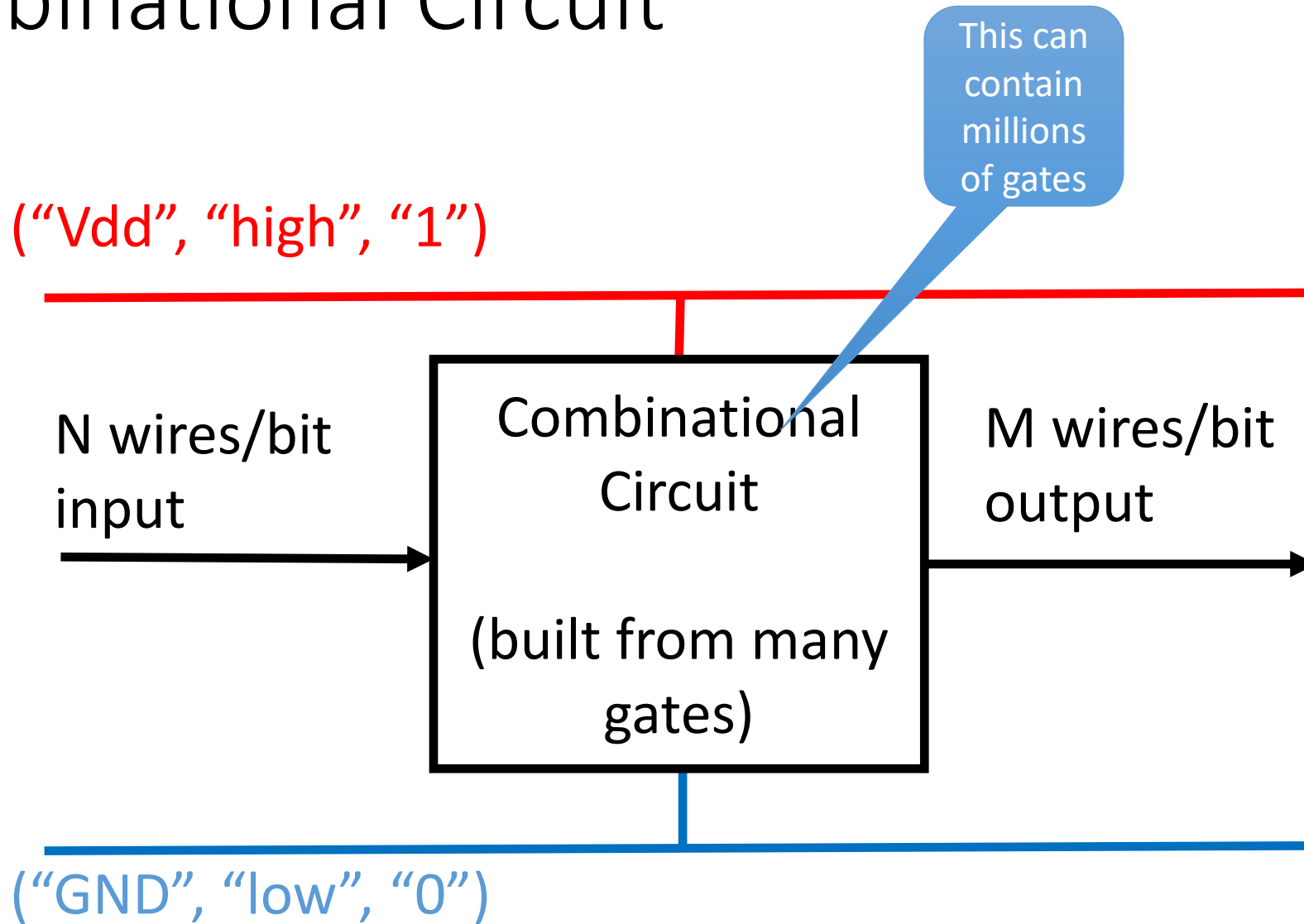
AND Gate



Cascading Gates



Combinational Circuit



The Mathematical View of a Combinational Circuit

- Combinational circuits (physical view) realize logic functions (mathematical view)
- With “function” we mean a **mapping** from a set of inputs to a set of outputs
- In mathematics, there exist many ways to express such a mapping, e.g.: $y = f(x) = x^2$
- If you choose a value for x , you get a value for y . We call x the independent value and y the dependent value

Logic Functions (or Boolean Functions)

- The “input” of a logic function is a tuple consisting of 0’s and 1’s
- The “output” of a logic function is, depending on the input values, 0 or 1
- Example: $y = a \ \& \ b$ (“logic-AND”)

http://en.wikipedia.org/wiki/Boolean_function

$\mathbf{B}^k \rightarrow \mathbf{B}$, where $\mathbf{B} = \{0, 1\}$ is a [Boolean domain](#) and k is a non-negative integer called the [arity](#) of the function.

In the case where $k = 0$, the "function" is essentially a constant element of \mathbf{B} .

Truth Table

- A truth table **uniquely** describes a logic function.
- Example: The logic-AND function with 2 input variables x_1 and x_0

x_1	x_0	y
0	0	0
0	1	0
1	0	0
1	1	1

Elements of a truth table

$$y = f(x_1, x_0)$$

Input
variables

x_1	x_0	

Elements of a truth table

$$y = f(x_1, x_0)$$

List all combinations of input variables. It is convenient to list them in sorted order. We usually start with all zeroes.

Input variables

x_1	x_0	
0	0	
0	1	
1	0	
1	1	

Elements of a truth table

$$y = f(x_1, x_0)$$

List all combinations of input variables. It is convenient to list them in sorted order. We usually start with all zeroes.

Input variables

Output

x_1	x_0	y
0	0	
0	1	
1	0	
1	1	

Elements of a truth table

$$y = f(x_1, x_0)$$

List all combinations of input variables. It is convenient to list them in sorted order. We usually start with all zeroes.

Input variables

Output

x_1	x_0	y
0	0	$f(0,0)$
0	1	$f(0,1)$
1	0	$f(1,0)$
1	1	$f(1,1)$

Size of truth tables

x	f(x)
0	f(0)
1	f(1)

1 input variable

2^1 possible values for x

Size of truth tables

$n=2$

x1	x0	f(x1, x0)
0	0	f(0,0)
0	1	f(0,1)
1	0	f(1,0)
1	1	f(1,1)

2^n

2 input variables

2^2 possible combinations for (x1, x0)

Size of truth tables

$n=3$

x2	x1	x0	f(x2,x1,x0)
0	0	0	f(0,0,0)
0	0	1	f(0,0,1)
0	1	0	f(0,1,0)
0	1	1	f(0,1,1)
1	0	0	f(1,0,0)
1	0	1	f(1,0,1)
1	1	0	f(1,1,0)
1	1	1	f(1,1,1)

2^n

3 input variables

2^3 possible
combinations
for (x2, x1, x0)

Size of truth tables

n input variables

2^n possible combinations

The size of a truth table grows exponentially with n .

How many logic functions are possible?

- Let's start with logic functions with 1 input variable

How many logic functions are possible?

- Let's start with logic functions with 1 input variable
- There exist 4 possible different truth tables:

x	y
0	0
1	0

x	y
0	1
1	0

x	y
0	0
1	1

x	y
0	1
1	1

How many logic functions are possible?

- Let's start with logic functions with 1 input variable
- There exist 4 possible different truth tables
- In the y-column we see **all possible combinations**

x	y
0	0
1	0

x	y
0	1
1	0

x	y
0	0
1	1

x	y
0	1
1	1

How many logic functions are possible?

- Let's start with logic functions with 1 input variable
- There exist 4 possible different truth tables
- In the y-column we see **all possible combinations**

x	Y_0	Y_1	Y_2	Y_3
0	0	1	0	1
1	0	0	1	1

How many logic functions are possible?

- Two input variables x_1 and x_0

x_1	x_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9	Y_{10}	Y_{11}	Y_{12}	Y_{13}	Y_{14}	Y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

How many logic functions are possible?

- Two input variables x_1 and x_0

$n=2$

x_1	x_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9	Y_{10}	Y_{11}	Y_{12}	Y_{13}	Y_{14}	Y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

How many logic functions are possible?

- Two input variables x_1 and x_0

$n=2$

	x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
2 ⁿ	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

How many logic functions are possible?

- Two input variables x_1 and x_0

$n=2$

	x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
2^n	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

$$2^{2^n} = 2^{2^2} = 16$$

With $n = 3, 4, \dots$

- $n = 3$, $2^3 = 8$ lines, $2^8 = 256$ functions
- $n = 4$, $2^4 = 16$ lines, $2^{16} = 65536$ functions
- $n = 5$, $2^5 = 32$ lines, $2^{32} = 4294967296$ functions
- $n = 6$, $2^6 = 64$ lines, $2^{64} = 18446744073709551616$ functions

Back to $n = 2$

- Some functions are “popular” and have names:

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

AND



Popular 2-input functions

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

AND

OR



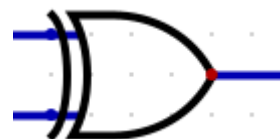
Popular 2-input functions

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

XOR

AND

OR



Popular 2-input functions



NAND

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

XOR

AND

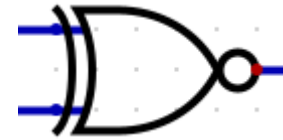
OR

Popular 2-input functions



x_1	x_0	y_0	NOR y_1	y_2	y_3	y_4	y_5	NAND y_6	y_7	AND y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	OR y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Popular 2-input functions



x_1	x_0	y_0	NOR y_1	y_2	y_3	y_4	y_5	NAND y_6	y_7	XNOR y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
								XOR		AND						OR	

Some functions are **trivial**

x_1	x_0	y_0	NOR y_1	y_2	y_3	y_4	y_5	XOR NAND y_6	AND y_7	XNOR y_8	y_9	y_{10}	y_{11}	x_1 y_{12}	y_{13}	OR y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Some functions are **trivial**

x_1	x_0	y_0	NOR y_1	y_2	y_3	y_4	y_5	NAND y_6	y_7	AND y_8	XNOR y_9	x_0 y_{10}	y_{11}	x_1 y_{12}	y_{13}	OR y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Some functions are **trivial**

x_1	x_0	NOR		y_2	y_3	y_4	y_5	NAND		XNOR		y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
		y_0	y_1					y_6	y_7	y_8	y_9						
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
		0						XOR	AND			x_0	x_1		OR		

Some functions are **trivial**

x_1	x_0	NOR		y_2	y_3	y_4	y_5	NAND		XNOR		y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
		y_0	y_1					y_6	y_7	y_8	y_9						
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
		0						XOR	AND			x_0	x_1		OR	1	

Some functions are “almost trivial”

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
		0			$\sim x_1$			XOR	AND			x_0		x_1		OR	1

Some functions are “almost trivial”

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
		0			$\sim x_1$		$\sim x_0$	XOR		AND		x_0		x_1		OR	1

Some functions are “implications”

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

NOR (above y_0, y_1) NAND (above y_6, y_7, y_8, y_9) XNOR (above $y_{10}, y_{11}, y_{12}, y_{13}$)
 x_1 implies x_0 (above y_{11})
 0 (below y_0) $\sim x_1$ (below y_3) $\sim x_0$ (below y_5) XOR (below y_6, y_7) AND (below y_8, y_9) x_0 (below y_{10}) x_1 (below y_{12}) OR (below y_{14}) 1 (below y_{15})

Some functions are “implications”

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

NOR (above y_0, y_1) NAND (above y_6, y_7, y_8, y_9) XNOR (above y_9)

x_1 implies x_0 (purple arrow pointing to y_{11})

x_1 is implied by x_0 (purple arrow pointing to y_{13})

Labels below the table: 0 (green, below y_0), $\sim x_1$ (grey, below y_3, y_5), $\sim x_0$ (grey, below y_4, y_6, y_7, y_8), XOR (red, below y_6, y_7), AND (red, below y_8, y_9), x_0 (green, below y_{10}), x_1 (green, below y_{12}), OR (red, below y_{14}), 1 (green, below y_{15})

And some functions are “inverse implications”

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

x_1 does not imply x_0 (pointing to y_4)
 x_1 implies x_0 (pointing to y_{11})
 x_1 is implied by x_0 (pointing to y_{13})

NOR, NAND, XNOR, XOR, AND, OR, 1

0 , $\sim x_1$, $\sim x_0$, x_0 , x_1

And some functions are “inverse implications”

x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

x_1 does not imply x_0 (points to y_4)
 x_1 implies x_0 (points to y_{11})

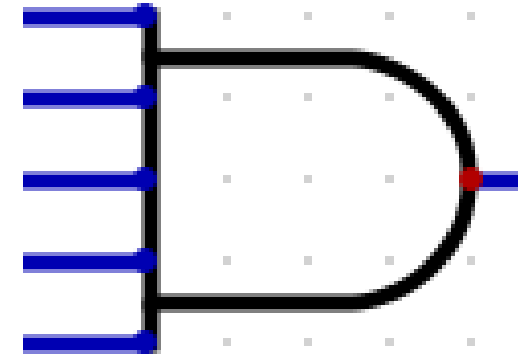
NOR (points to y_1)
 NAND (points to y_6)
 XNOR (points to y_9)

0 (points to y_0)
 $\sim x_1$ (points to y_2)
 $\sim x_0$ (points to y_5)
 XOR (points to y_6)
 AND (points to y_8)
 x_0 (points to y_{10})
 x_1 (points to y_{12})
 OR (points to y_{14})
 1 (points to y_{15})

x_1 is not implied by x_0 (points to y_2)
 x_1 is implied by x_0 (points to y_{14})

The popular functions can also have more than 2 inputs

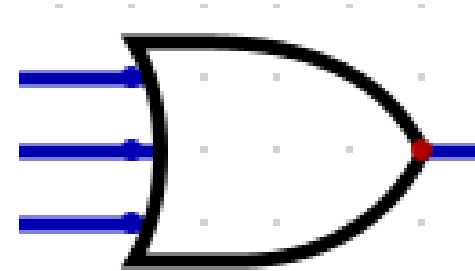
- Example: 5-input AND



- Only if **all** input values are 1, the output is 1

The popular functions can also have more than 2 inputs

- Example: 3-input OR

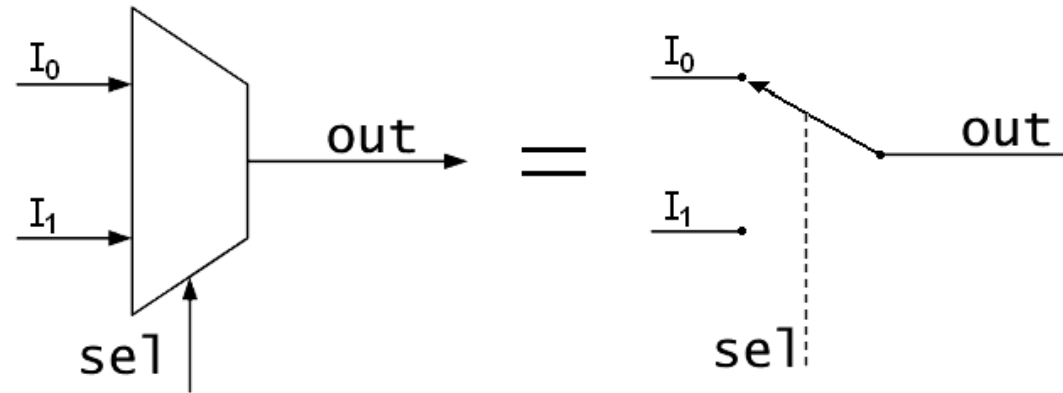


- If at least **one** input values is 1, the output is 1

Be careful with XOR function with more than 2 inputs

- Interpretation #1: Output is 1, if an **odd** number of input values is 1
- Interpretation #2: Output is 1, if **exactly 1** input value is 1
- Interpretation #1 is the “common” interpretation! → This is what we use in the lecture

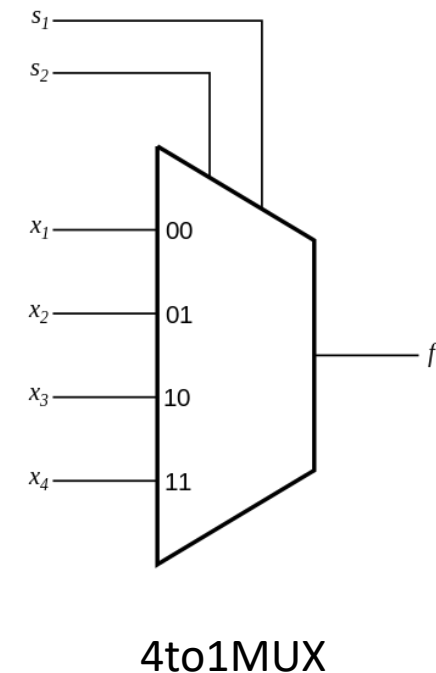
Other Important Gates – Multiplexer (MUX)



- The select signal (sel) determines whether out is equal to I_0 or I_1 :
 - $sel = 0$ means $out = I_0$
 - $sel = 1$ means $out = I_1$

Scaling to more inputs

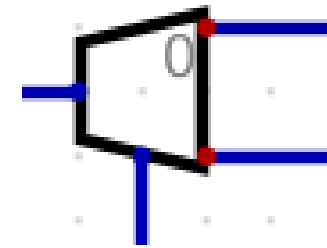
- With each additional select signal, the number of selectable inputs doubles
- 2to1MUX: 1 select signal
- 4to1MUX: 2 select signals
- 8to1MUX: 3 select signals
- ...



Other Important Gates – Demultiplexer/Decoder

- The select signals (sel) of a demultiplexer determine whether to which output the input is mapped:

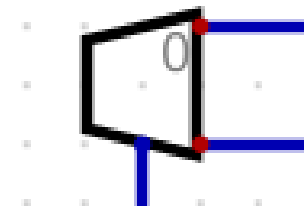
- Sel = 0 means $out_0 = in$
- Sel = 1 means $out_1 = in$



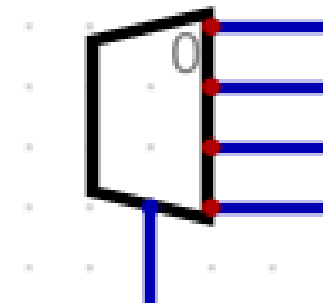
1to2 DEMUX

- The select signals (sel) of a decoder determines which output is high

- Sel = 0 means $out = 1$
- Sel = 1 means $out = 1$



1to2 Decoder



1to4 Decoder

How many different types of gates are needed to be able to implement any logic function?

Functional Completeness

- A functionally complete set of logic gates is a set that allows to build all possible truth tables by combining gates of this set.
- Important sets are:
 - **{NAND}**: Any circuit can be built just by using NAND gates (try it out in Digital!)
 - **{NOR}**: Any circuit can be built just by using NOR gates
 - **{AND, NOT}**: Any circuit can be built just by using AND and NOT gates
 - **{AND, OR, NOT}**: The set we use to map truth tables to equations

Combinational Circuits

(Composing logic gates to implement a desired functionality)

Boolean Functions – The Mathematical Description of a Combinational Circuit

- Symbol for inversion: \sim
- Symbol for AND: $\&$
- Symbol for OR: $|$

- Example:

$$y = (\sim x_1 \& x_0) | (x_2 \& \sim x_0) | (x_2 \& x_1)$$

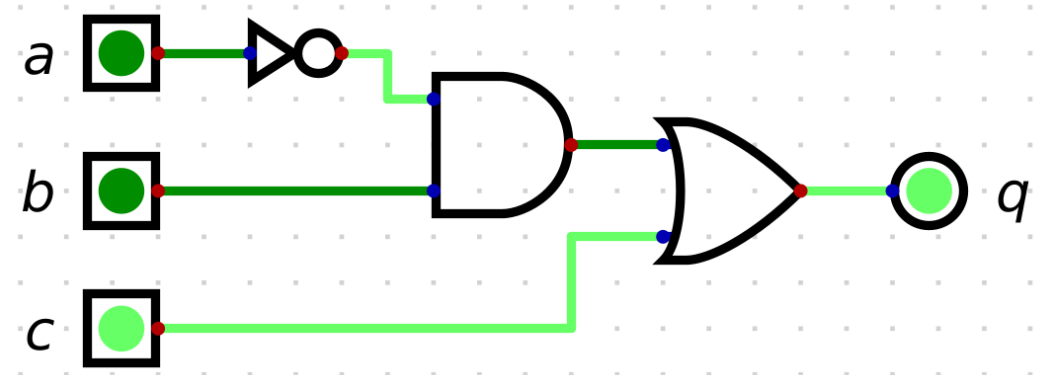
Tooling

Recommended tools for understanding logic circuits

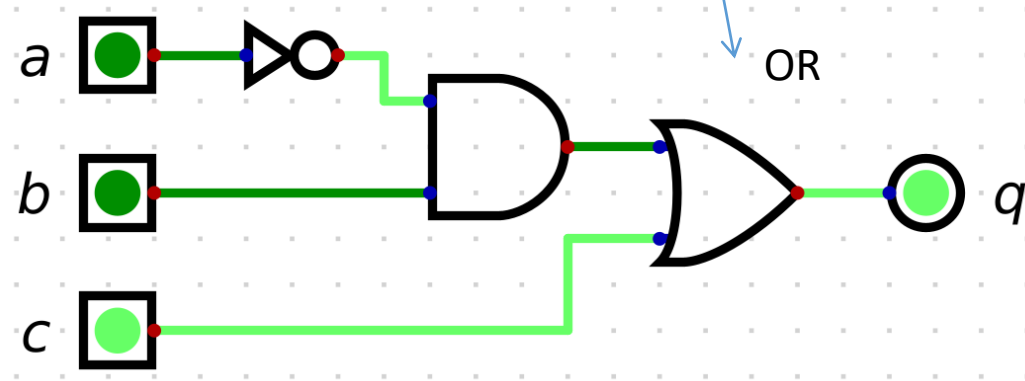
- Tool DIGITAL (installed on your VM)
 - <https://github.com/hneemann/Digital>
- Online tool for logic circuit design
 - <https://circuitverse.org/>

Logic functions map to Combinational Circuits and Vice Versa

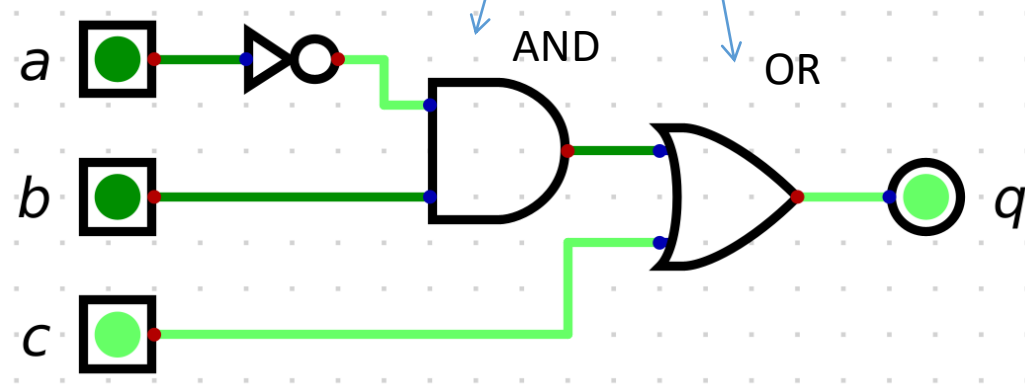
$$q = (\sim a \& b) \mid c$$



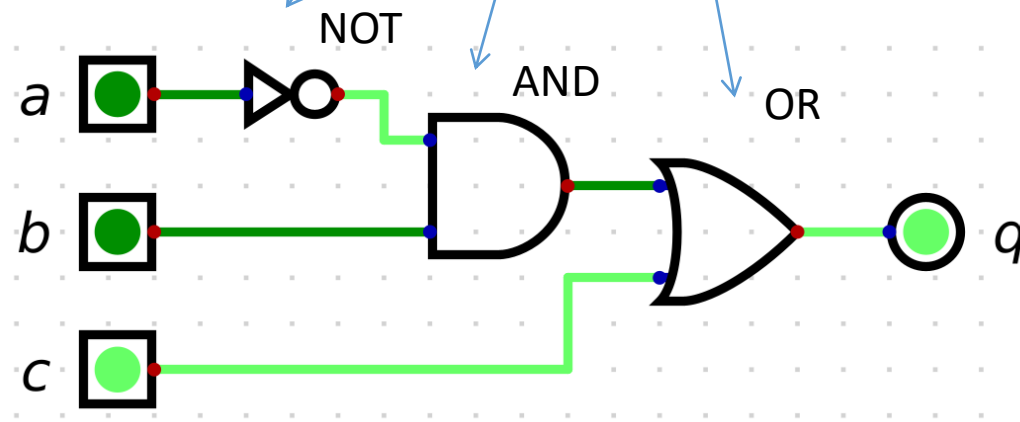
$$q = (\sim a \ \& \ b) \ | \ c$$



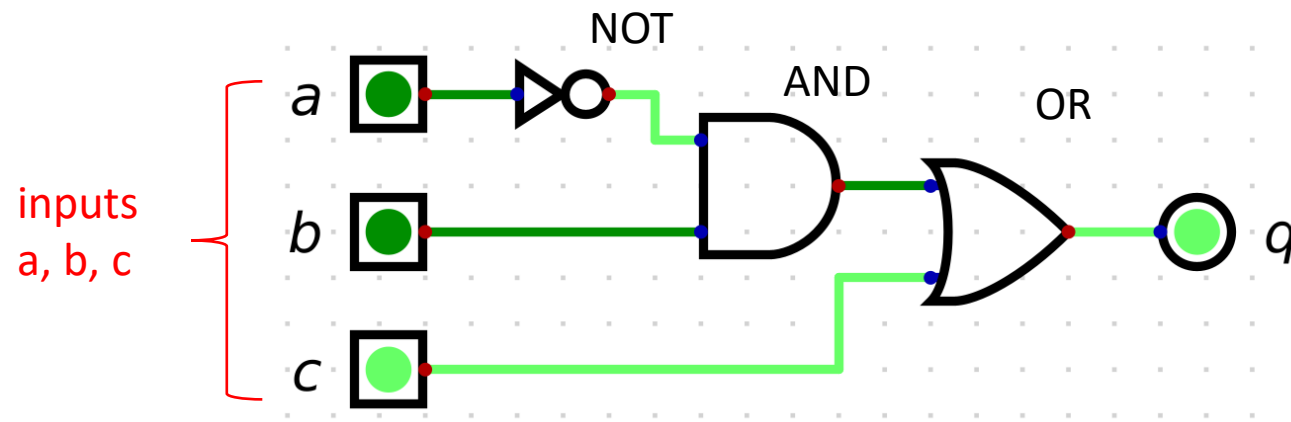
$$q = (\sim a \ \& \ b) \ | \ c$$



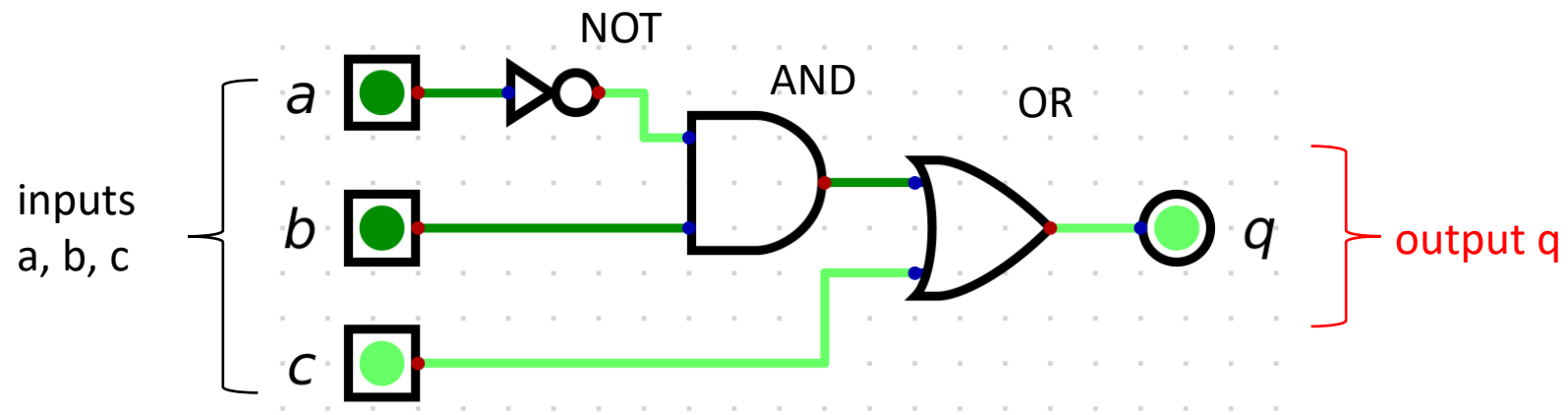
$$q = (\sim a \ \& \ b) \ | \ c$$



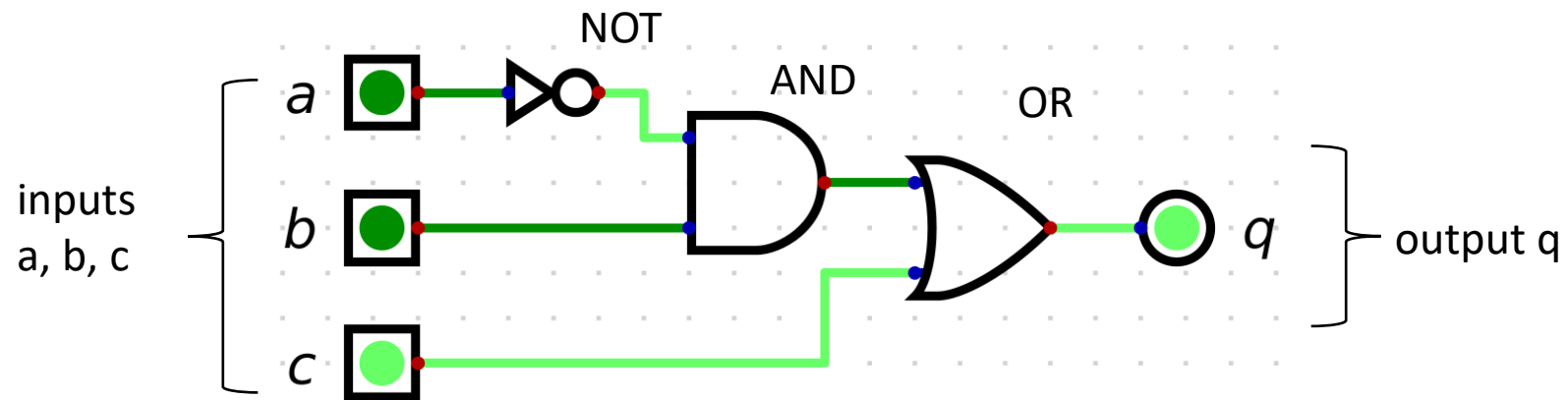
$$q = (\sim a \ \& \ b) \ | \ c$$



$$q = (\sim a \& b) \mid c$$



$$q = (\sim a \& b) \mid c$$



Inputs are **independent**;
(can be chosen or
are defined externally)

Outputs **dependent**
on the inputs

Implementing a logic function:

A design flow

- Start with developing a truth table
- Example: Adding three binary variables u , v and w : $s = u + v + w$
- With 3 variables we have 2^3 possible combinations for input situations.

Implementing a logic function: A design flow

U	+	v	+	w
0	+	0	+	0
0	+	0	+	1
0	+	1	+	0
0	+	1	+	1
1	+	0	+	0
1	+	0	+	1
1	+	1	+	0
1	+	1	+	1

8 possible combinations;
sorted from (0, 0, 0) to (1, 1, 1)

Implementing a logic function: A design flow

u	+	v	+	w	=	s
0	+	0	+	0	=	0
0	+	0	+	1	=	1
0	+	1	+	0	=	1
0	+	1	+	1	=	2
1	+	0	+	0	=	1
1	+	0	+	1	=	2
1	+	1	+	0	=	2
1	+	1	+	1	=	3

The result for each possible case

Implementing a logic function: A design flow

u	+	v	+	w	=	s_1	s_0
0	+	0	+	0	=	0	0
0	+	0	+	1	=	0	1
0	+	1	+	0	=	0	1
0	+	1	+	1	=	1	0
1	+	0	+	0	=	0	1
1	+	0	+	1	=	1	0
1	+	1	+	0	=	1	0
1	+	1	+	1	=	1	1

Re-writing the result as a
binary number

Implementing a logic function: A design flow

u	v	w	s_1	s_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The truth table

Implementing a logic function: A design flow

u	v	w	s_1	s_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The logic function for s_0 :

We only look at lines
where s_0 gets “true”
i.e. “1”.

Implementing a logic function: A design flow

u	v	w	s_1	s_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The logic function for s_0 :

$$s_0 = (\sim u \ \& \ \sim v \ \& \ w) \dots$$

Implementing a logic function: A design flow

u	v	w	s_1	s_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The logic function for s_0 :

$$s_0 = (\sim u \ \& \ \sim v \ \& \ w) \mid (\sim u \ \& \ v \ \& \ \sim w) \dots$$

Implementing a logic function: A design flow

u	v	w	s_1	s_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The logic function for s_0 :

$$s_0 = (\sim u \& \sim v \& w) \mid$$

$$(\sim u \& v \& \sim w) \mid$$

$$(u \& \sim v \& \sim w) \dots$$

Implementing a logic function: A design flow

u	v	w	s_1	s_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

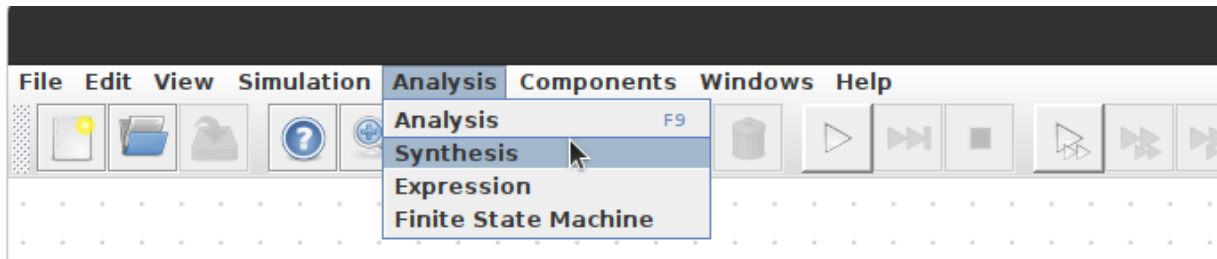
The logic function for s_0 :

$$\begin{aligned}
 s_0 = & (\sim u \ \& \ \sim v \ \& \ w) \mid \\
 & (\sim u \ \& \ v \ \& \ \sim w) \mid \\
 & (u \ \& \ \sim v \ \& \ \sim w) \mid \\
 & (u \ \& \ v \ \& \ w)
 \end{aligned}$$

Implementing a logic function: With a little help from **Digital**

<https://github.com/hneemann/Digital>

Implementing a logic function: With a little help from Digital



Start Digital

Goto Analysis → Synthesis

Table				
File	New	Edit	Create	K-Map
A	B	C	Y	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	0	

Y = 0

Implementing a logic function: With a little help from Digital

File	New	Edit	Create	K-Map
	A	Undo	Ctrl-Z	Y
	0	Redo	Ctrl-Y	0
	0	Reorder/Delete Input Variables		0
	0	Add Input Variable		0
	0	Reorder/Delete Output Columns		0
	0	Add Output Column		0
	1	Set X to 0		0
	1	Set X to 1		0
	1	Set all to X		0
	1	Set all to 0		0
	1	Set all to 1		0
		Invert all bits		

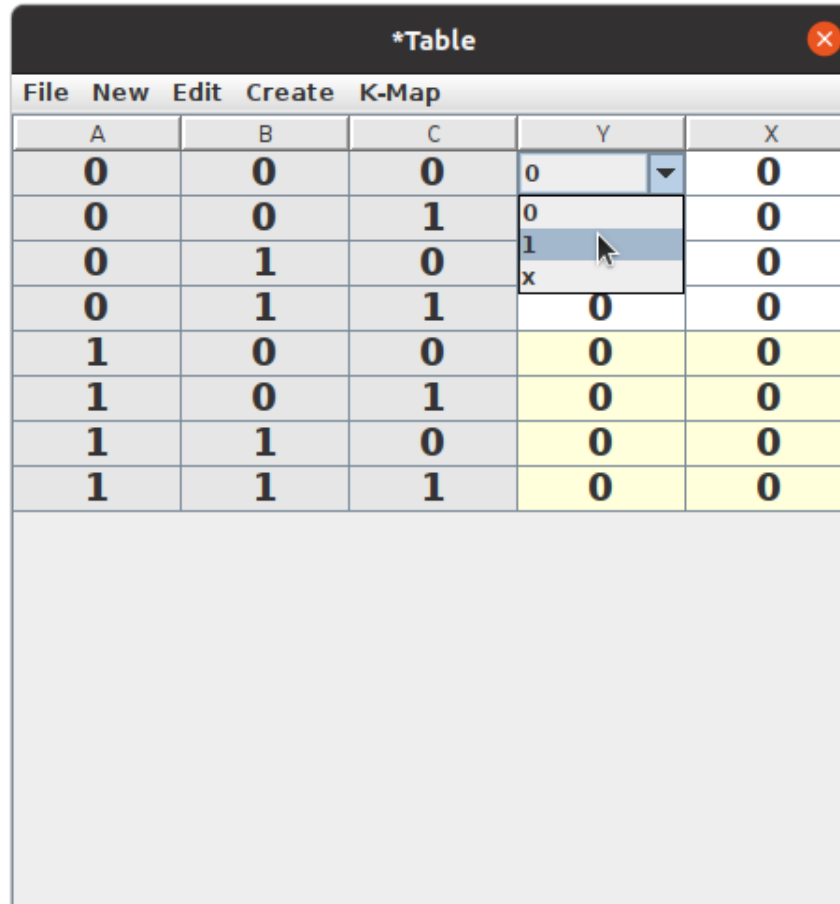
Y = 0

Start Digital

Goto Analysis → Synthesis

Adjust inputs and outputs

Implementing a logic function: With a little help from Digital



A	B	C	Y	X
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

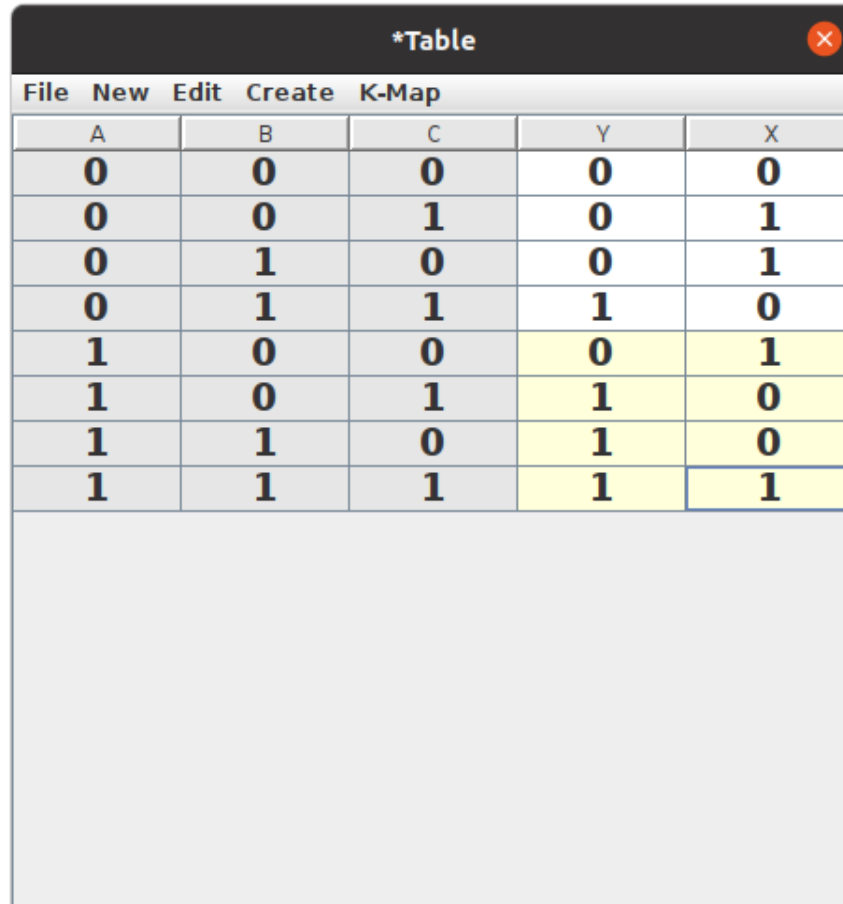
Start Digital

Goto Analysis → Synthesis

Adjust inputs and outputs

Specify the output section of
the truth table

Implementing a logic function: With a little help from Digital



A	B	C	Y	X
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Start Digital

Goto Analysis → Synthesis

Adjust inputs and outputs

Specify the output section of
the truth table

Implementing a logic function: With a little help from Digital

The screenshot shows the Digital software interface. The main window is titled '*Table' and contains a truth table with columns A, B, C, Y, and X. The table is as follows:

A	B	C	Y	X
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

An 'Export' dialog box is open in the foreground, displaying the following logic expressions:

$$Y = (A \wedge C) \vee (A \wedge B) \vee (B \wedge C)$$

$$X = (\neg A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge \neg C) \vee (A \wedge B \wedge C)$$

The dialog has 'Clipboard' and 'OK' buttons.

Start Digital

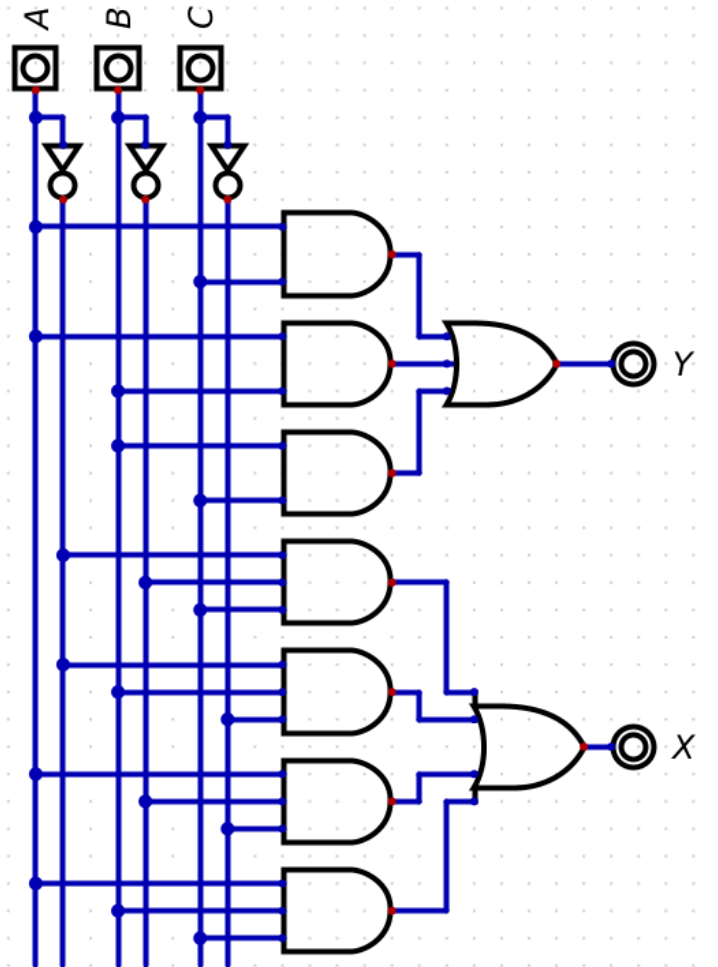
Goto Analysis → Synthesis

Adjust inputs and outputs

Specify the output section of
the truth table

Optionally: Check plain text export

Implementing a logic function: With a little help from Digital



Start Digital

Goto Analysis → Synthesis

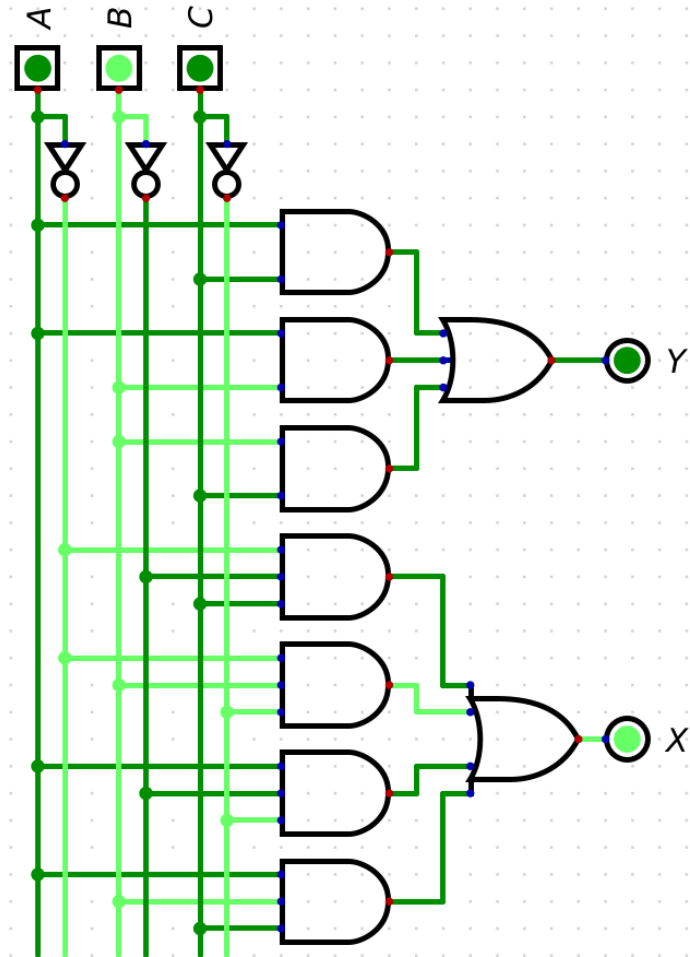
Adjust inputs and outputs

Specify the output section of
the truth table

Optionally: Check plain text export

Click “Create” and “Circuit”.

Implementing a logic function: With a little help from Digital

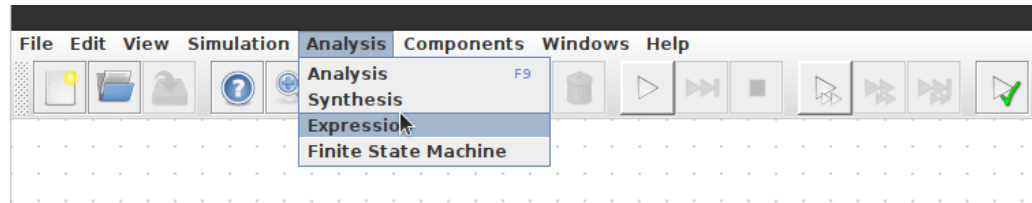


Switch to Simulation Mode:



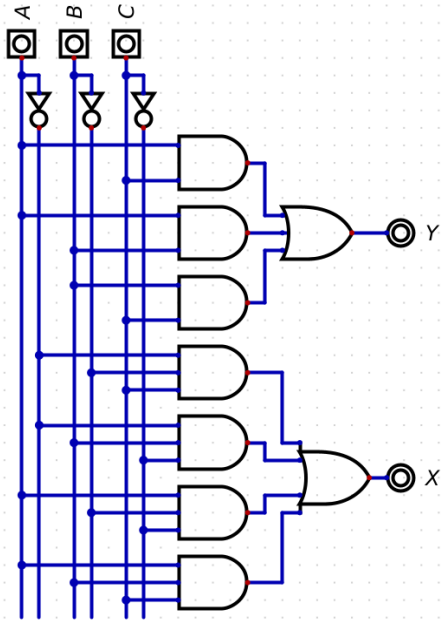
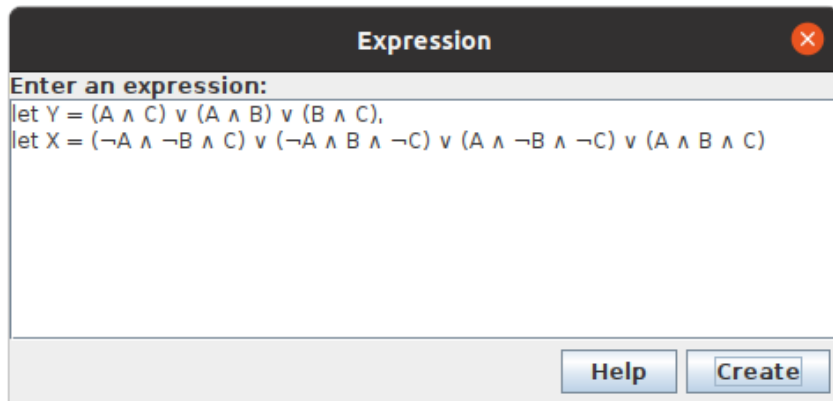
Simulate circuit with
all possible input combinations.

Implementing a logic function: With a little help from Digital



Alternative: specify the **expression**

Use “let” to specify outputs.
Separate formulas by comma.



Boolean Algebra

(Rules to transform/simplify logic functions)

Motivation

- A truth table is a unique representation of a logic function
- Describing a given functionality as a Boolean function is **not unique**
 - The same functionality can be described using different Boolean functions
- There is also **no unique circuit representation** for a logic function
 - The same functionality can be described by different combinational circuits

When building digital circuits, there is always an optimization towards different targets, such as minimum area, shortest latency, minimum power consumption,

Boolean Algebra

- Values of variables are only 0 or 1.
- 3 main operations:
 - Negation, also known as “inversion” \sim
 - Conjunctions, also known as “ANDing” $\&$
 - Disjunction, also known as “ORing” $|$
- 1 other popular operation: “exclusive OR” \wedge
- Boolean Algebra allows to transform Boolean functions/equations

Boolean Algebra

$$a \mid 0 = a$$

$$a \mid 1 = 1$$

$$a \& 0 = 0$$

$$a \& 1 = a$$

$$a \wedge 0 = 0$$

$$a \wedge 1 = \sim a$$

$$a \mid a = a$$

$$a \mid \sim a = 1$$

$$a \& a = a$$

$$a \& \sim a = 0$$

$$a \wedge a = a$$

$$a \wedge \sim a = 0$$

$$a \mid (a \& b) = a$$

$$a \& (a \mid b) = a$$

The proof of all these facts is rather easy: Take a truth table and check all possibilities.

Boolean Algebra

Associative Law:

$$(a \mid b) \mid c = a \mid (b \mid c)$$

$$(a \& b) \& c = a \& (b \& c)$$

$$(a \wedge b) \wedge c = a \wedge (b \wedge c)$$

Commutative Law:

$$a \mid b = b \mid a$$

$$a \& b = b \& a$$

$$a \wedge b = b \wedge a$$

The proof of all these facts is rather easy: Take a truth table and check all possibilities.

Boolean Algebra

Distributive Law:

$$a \mid (b \& c) = (a \mid b) \& (a \mid c)$$

$$a \& (b \mid c) = (a \& b) \mid (a \& c)$$

$$a \& (b \wedge c) = (a \& b) \wedge (a \& c)$$

The proof of all these facts is rather easy: Take a truth table and check all possibilities.

Boolean Algebra

De Morgan's Law:

$$\sim(a \ \& \ b) \ = \ \sim a \ | \ \sim b$$

$$\sim(a \ | \ b) \ = \ \sim a \ \& \ \sim b$$

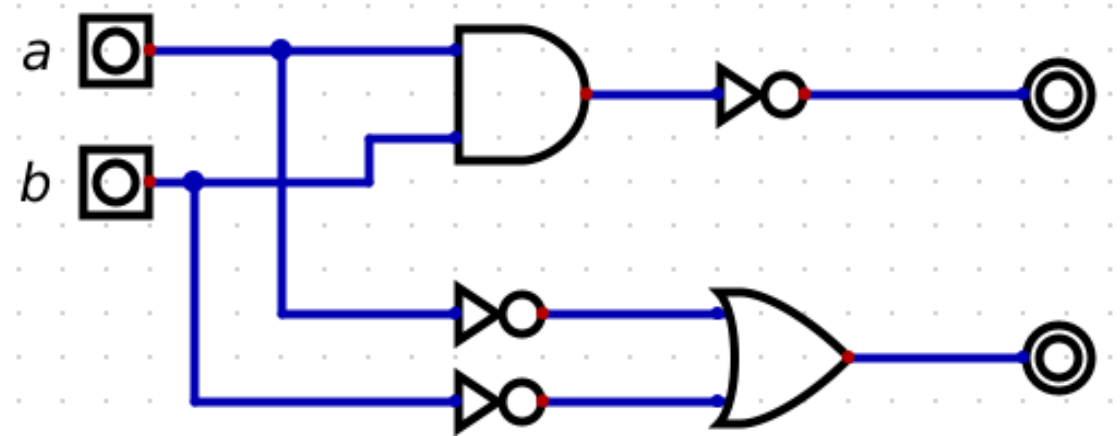
The proof of all these facts is rather easy: Take a truth table and check all possibilities.

Boolean Algebra

De Morgan's Law:

$$\sim(a \& b) = \sim a \mid \sim b$$

$$\sim(a \mid b) = \sim a \& \sim b$$

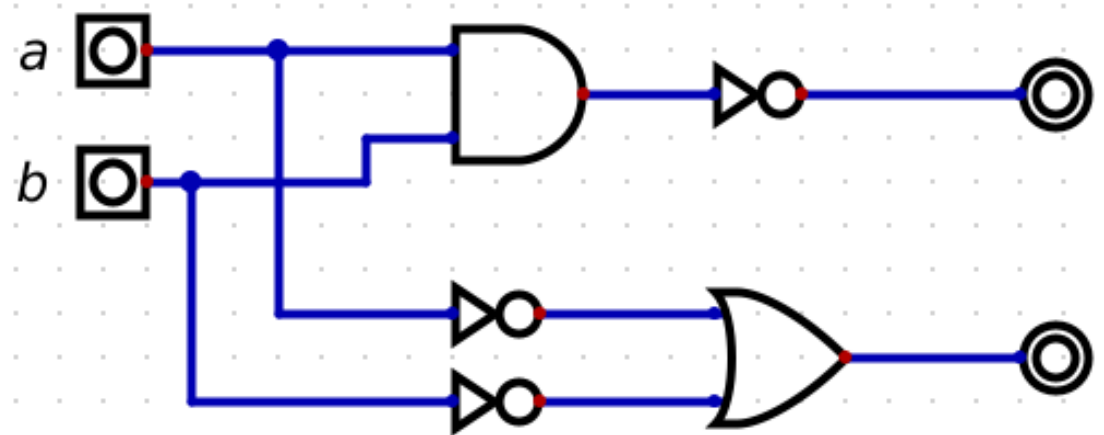


The proof of all these facts is rather easy: Take a truth table and check all possibilities.

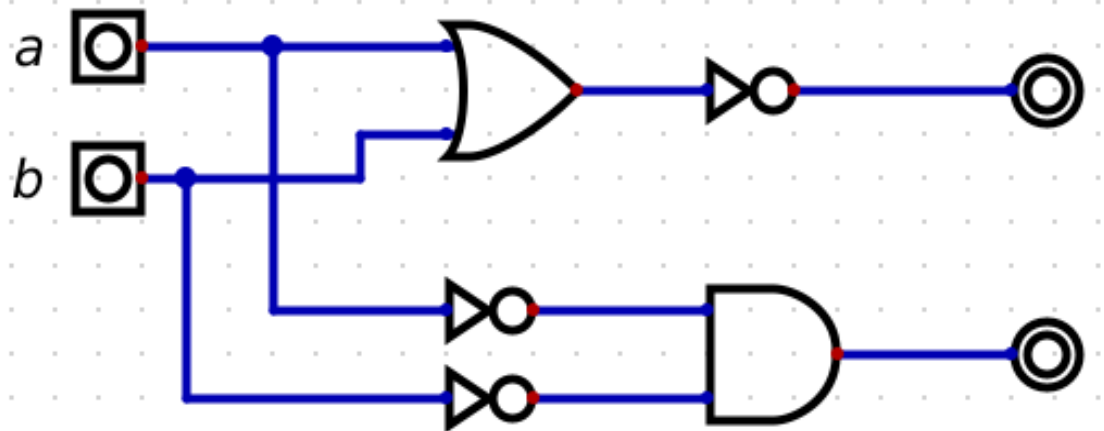
Boolean Algebra

De Morgan's Law:

$$\sim(a \& b) = \sim a \mid \sim b$$



$$\sim(a \mid b) = \sim a \& \sim b$$



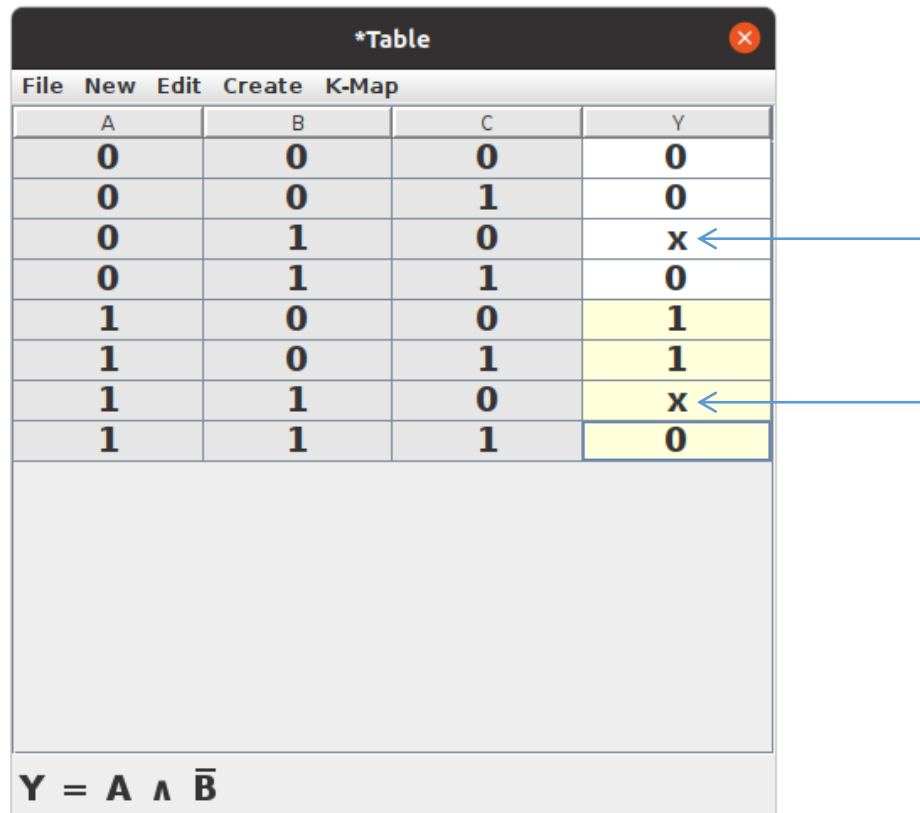
The proof of all these facts is rather easy: Take a truth table and check all possibilities.

Incomplete specification: “Don’t Cares”

- Sometimes we are not interested in some input combinations; we “**don’t care**” about the output of the logic function in this case.
- This is the case, when not all input combinations occur in some context.

Incomplete specification: “Don’t Cares”

- Example: 3 inputs, 2 “don’t cares”.



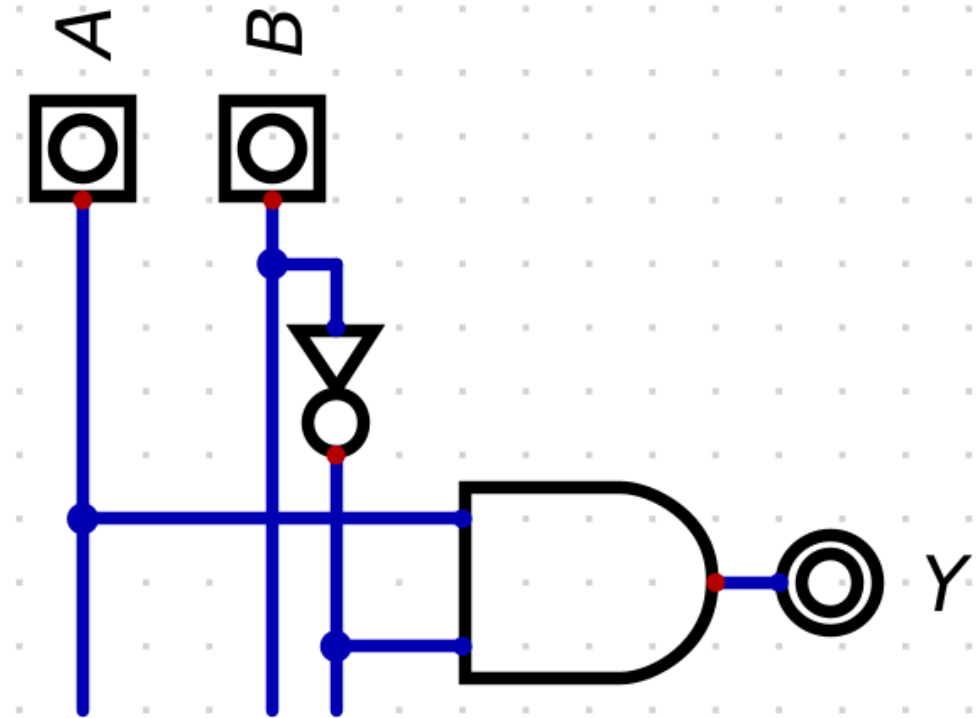
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	x
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	0

$Y = A \wedge \bar{B}$

Result after “synthesizing” with Digital

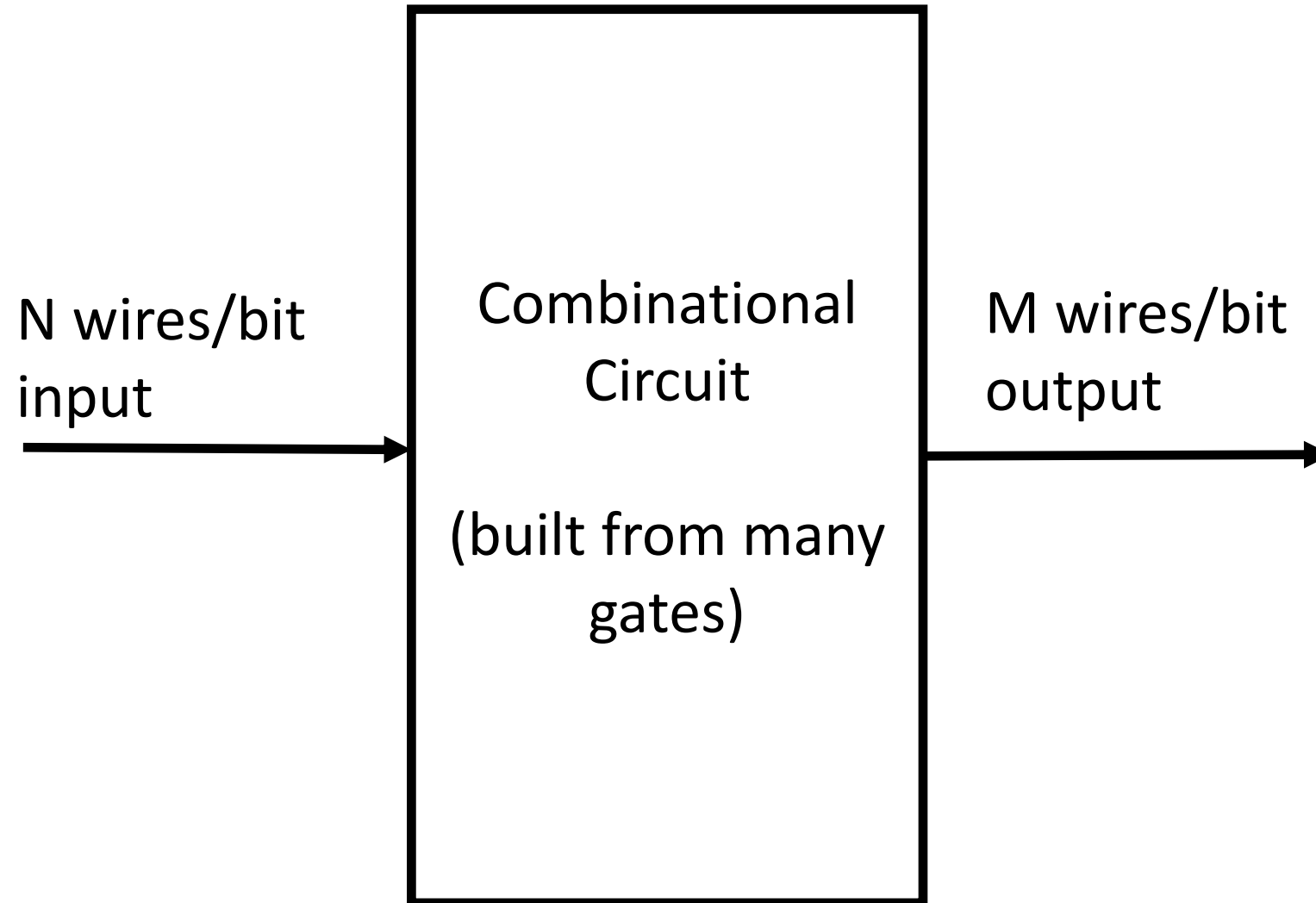
*Table				
File	New	Edit	Create	K-Map
A	B	C	Y	
0	0	0	0	
0	0	1	0	
0	1	0	x ←	
0	1	1	0	
1	0	0	1	
1	0	1	1	
1	1	0	x ←	
1	1	1	0	

$Y = A \wedge \bar{B}$

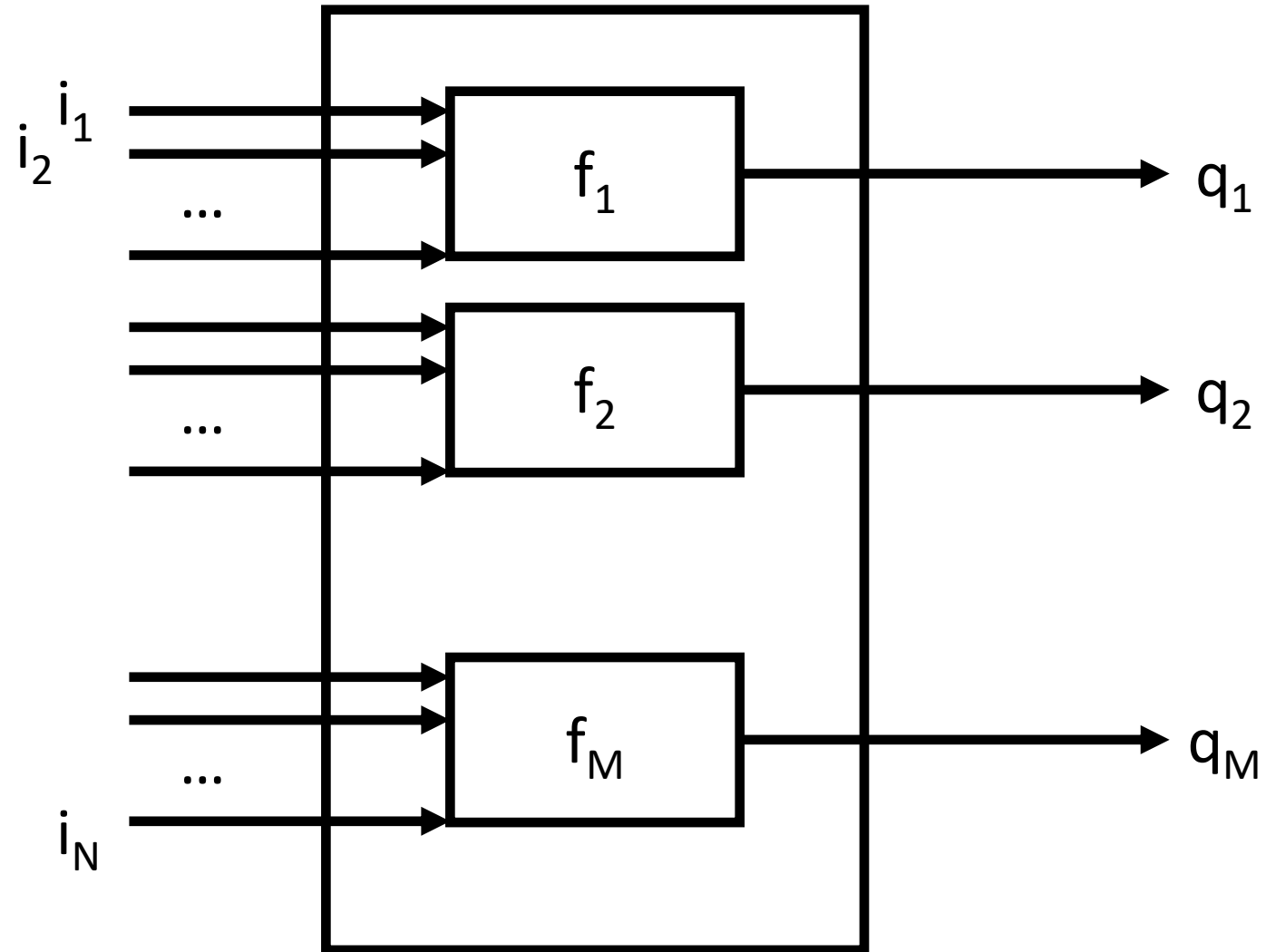


Describing Combinational and Sequential Circuits

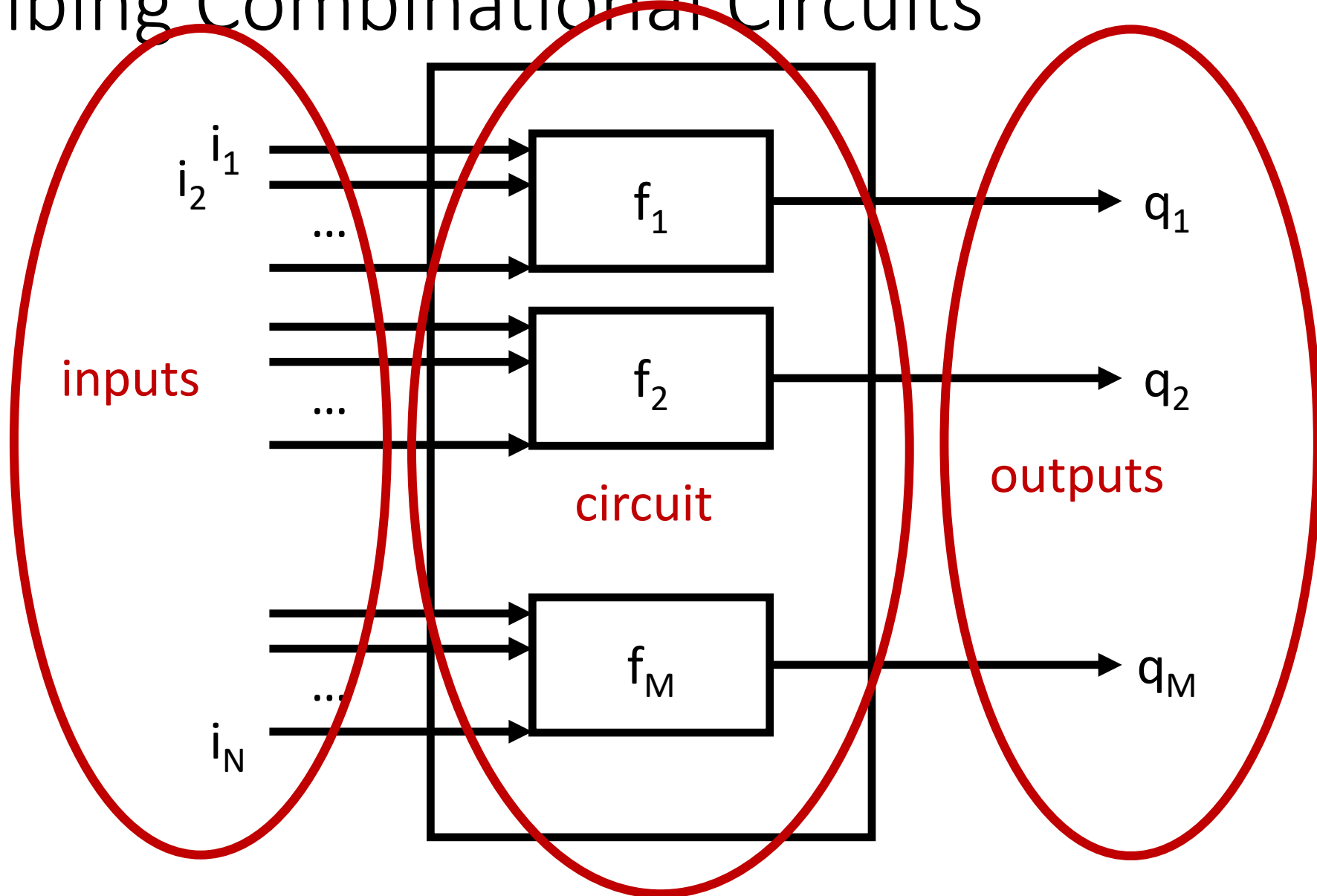
Describing Combinational Circuits



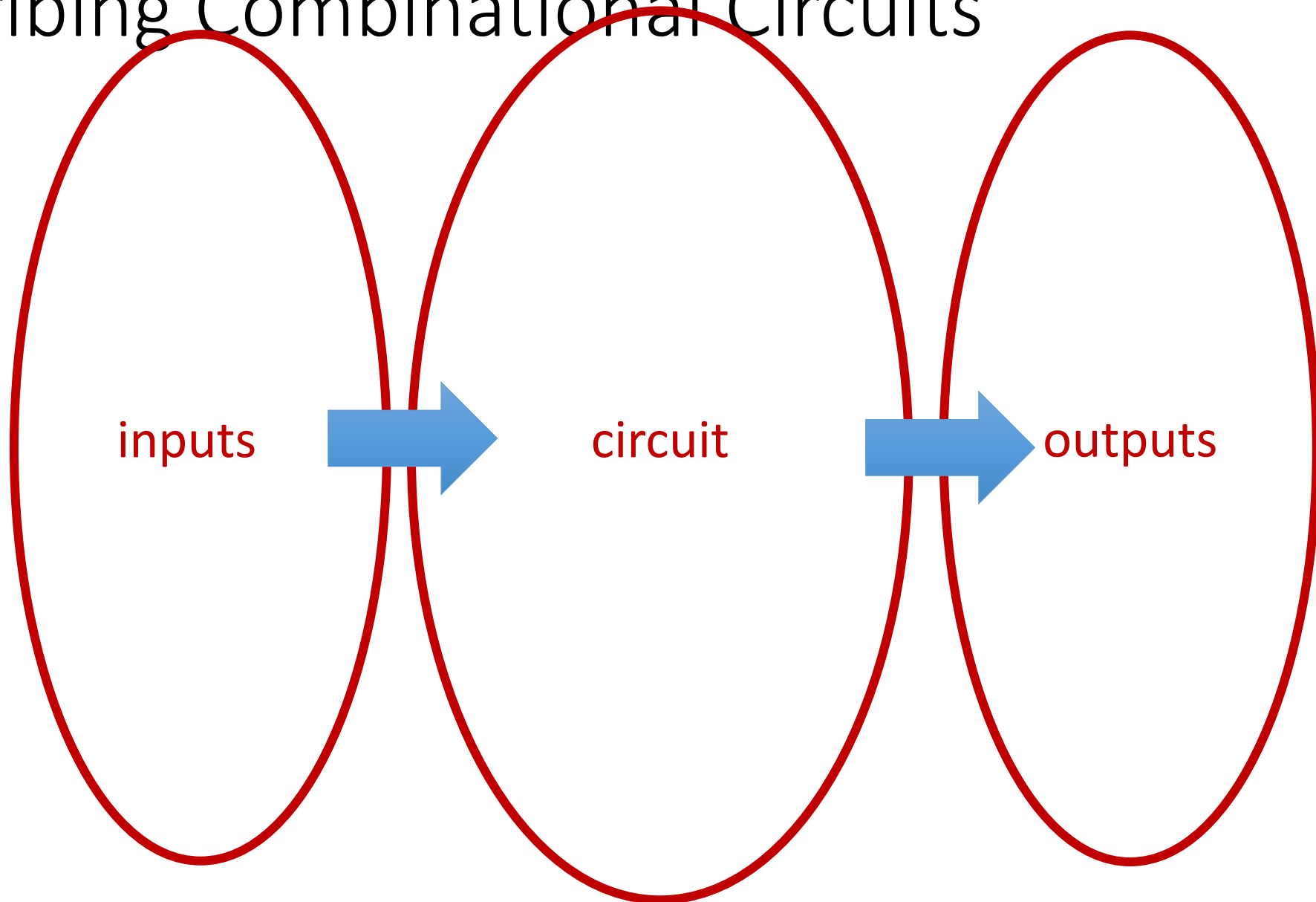
Describing Combinational Circuits



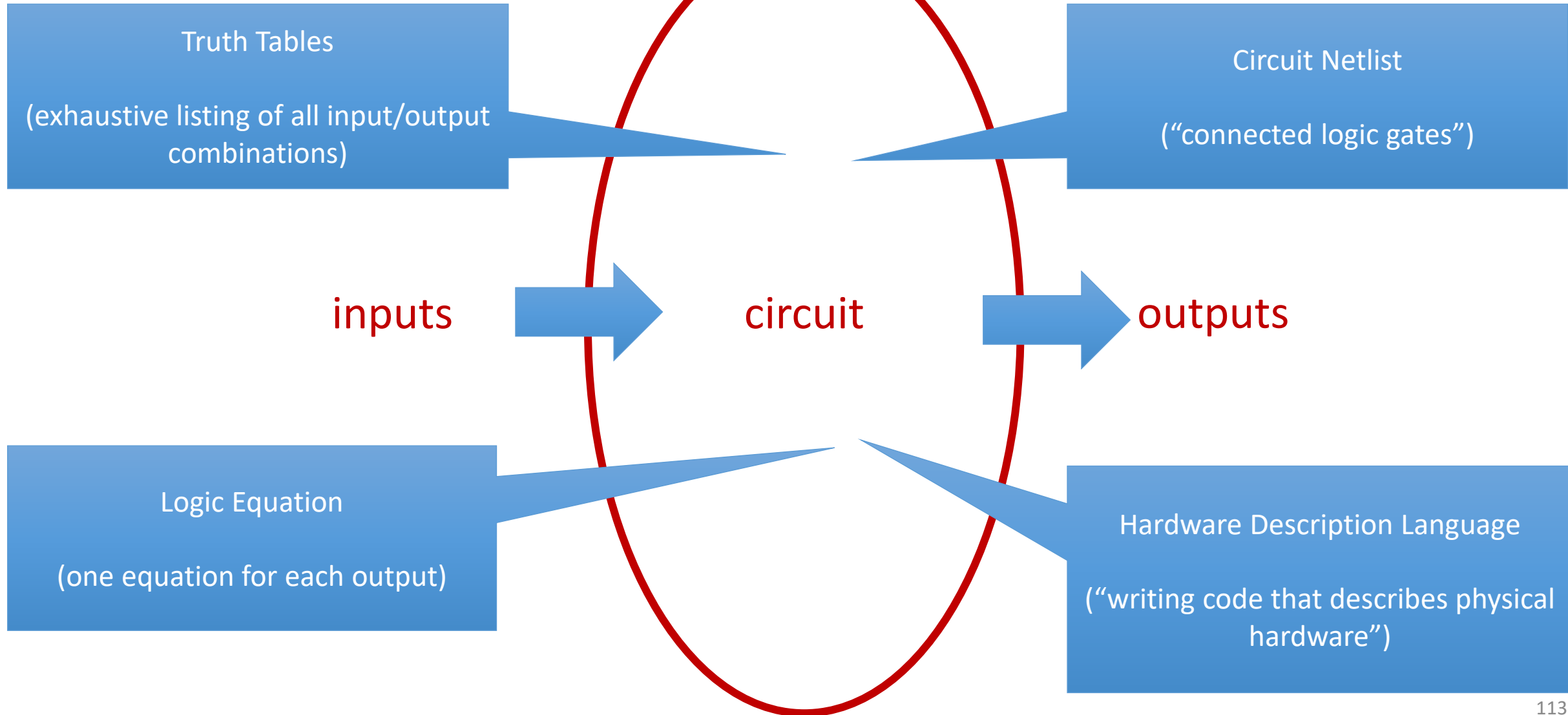
Describing Combinational Circuits



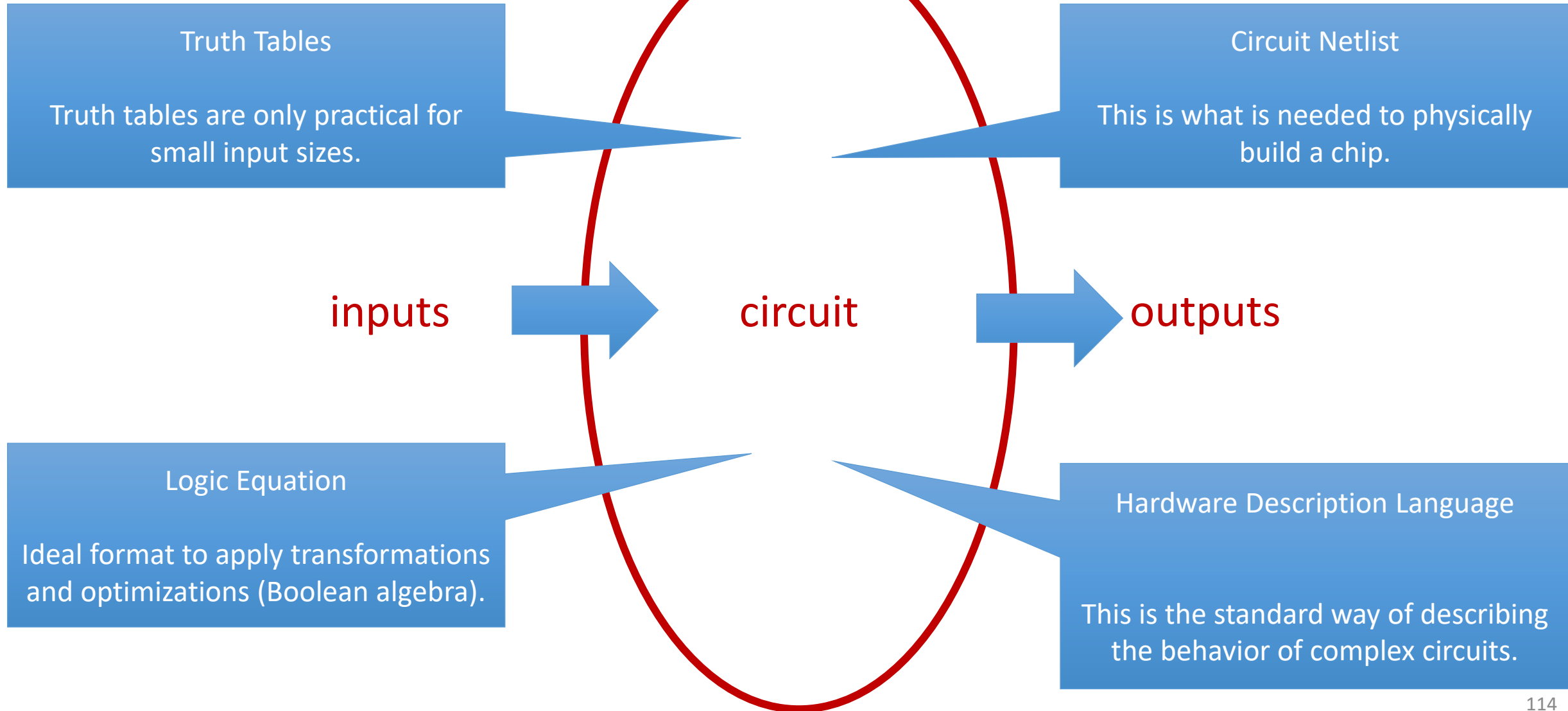
Describing Combinational Circuits



Describing Combinational Circuits



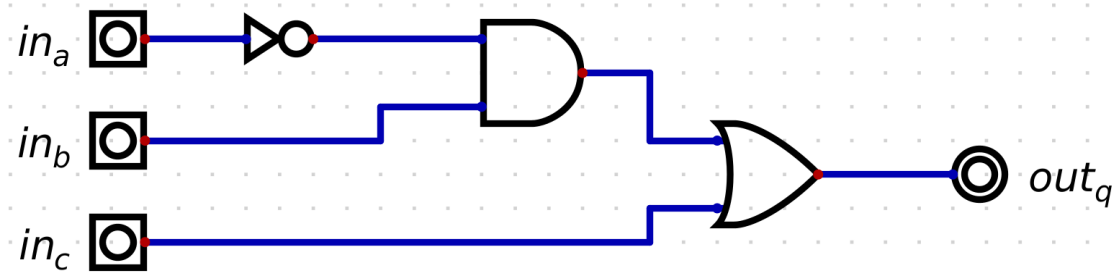
Describing Combinational Circuits



Example 1

in_a	in_b	in_c	out_q
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$$\text{out}_q = (\sim \text{in}_a \ \& \ \text{in}_b) \ | \ \text{in}_c$$



```

module simple_circuit (
  input in_a,
  input in_b,
  input in_c,
  output out_q
);
  assign out_q = ((~ in_a & in_b) | in_c);
endmodule

```

Truth Table

e 1

in_a	in_b	in_c	out_q
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

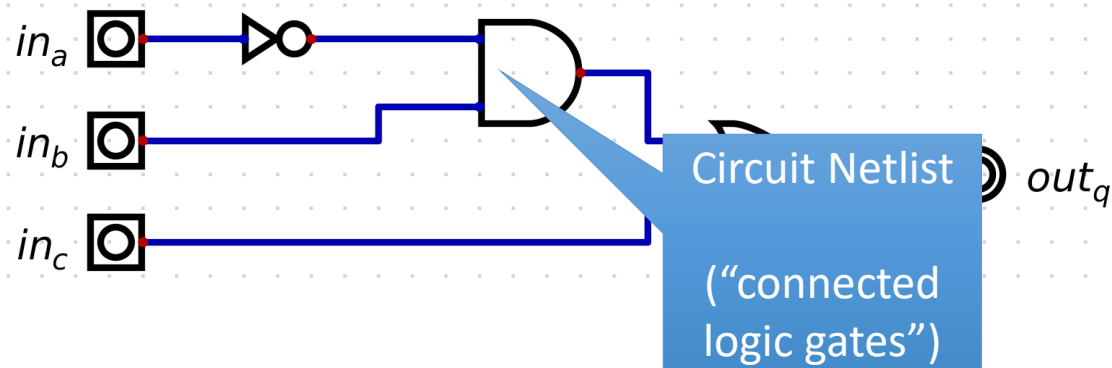
Logic equation

$$\text{out}_q = (\sim \text{in}_a \ \& \ \text{in}_b) \ | \ \text{in}_c$$

```

module simple_circuit (
  input in_a,
  input in_b,
  input in_c,
  output out_q
);
  assign out_q = ((~ in_a & in_b) | in_c);
endmodule

```

Hardware
Description
Language

SystemVerilog – A Hardware Description Language

```
module simple_circuit (  
    input in_a,  
    input in_b,  
    input in_c,  
    output out_q  
);  
    assign out_q = ((~ in_a & in_b) | in_c);  
endmodule
```

Example 2

```

module simple_circuit_with_mux (
  input logic in_a,
  input logic in_b,
  input logic in_c,
  input logic in_x,
  input logic in_y,
  input logic in_z,
  input logic mux_sel,
  output logic out_q
);

```

```

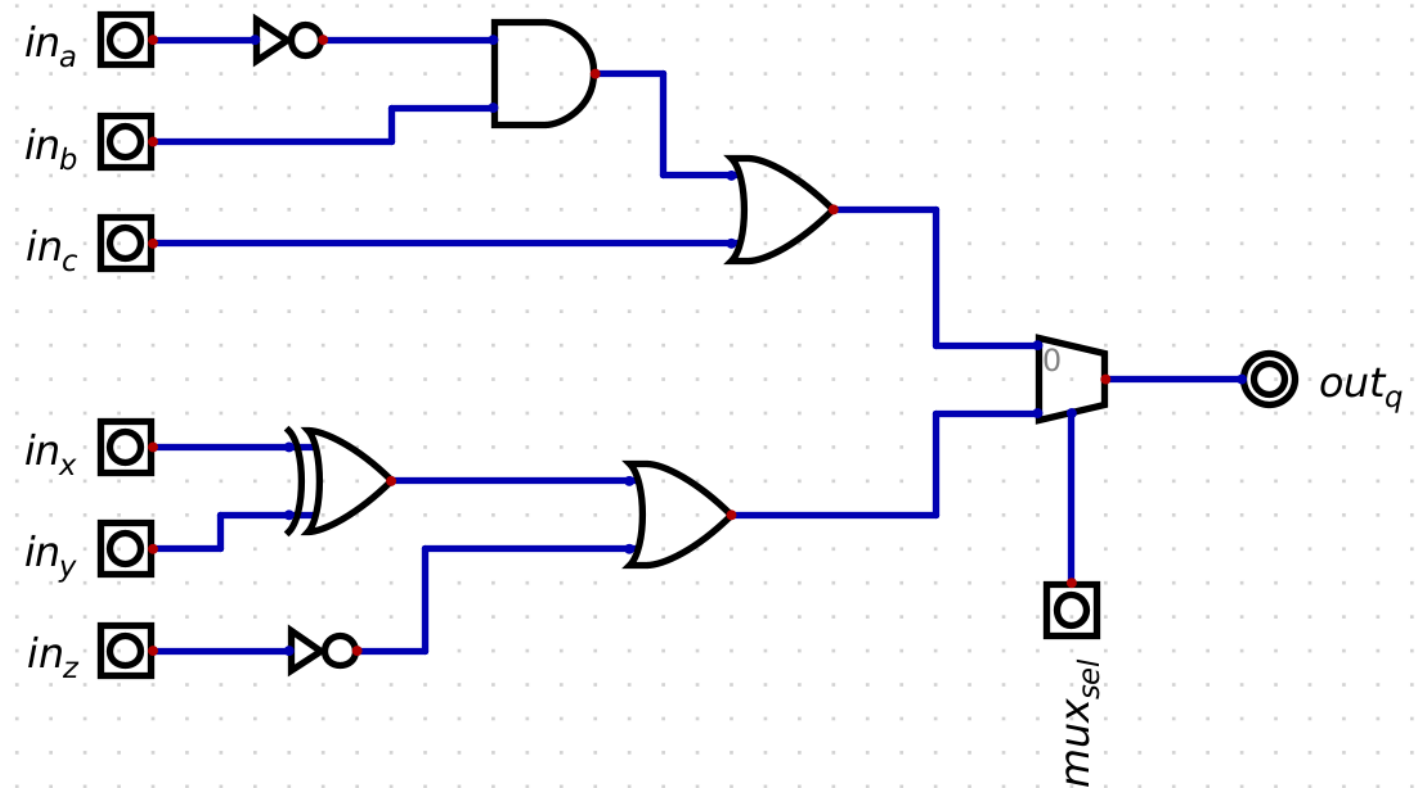
  always_comb begin
    if (mux_sel == 0)
      out_q = (~in_a & in_b) | in_c;
    else
      out_q = (in_x ^ in_y) | ~in_z;
    end

```

```

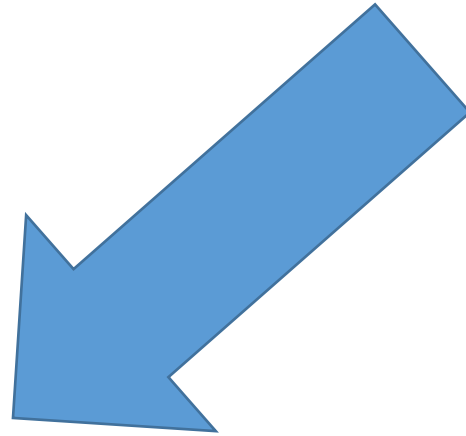
endmodule

```

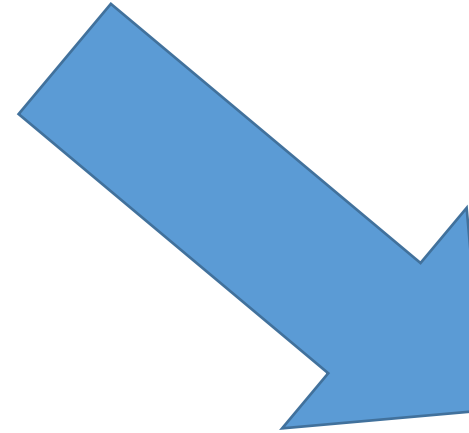


**I have created a SystemVerilog design
- how do I built a physical device that
implements this circuit?**

My
SystemVerilog
Code



Field Programmable Gate Array
(FPGA)



Application-Specific Integrated Circuit
(ASIC)

ASIC – Application-Specific Integrated Circuit

A chip that physically realizes your circuit

- Basic steps to building your ASIC (very high level view):
 - Select your favorite semiconductor manufacturing plant (see https://en.wikipedia.org/wiki/List_of_semiconductor_fabrication_plants)
 - Receive the standard cell library from the plant (“the list of logic gates that the plant can build”)
 - Map our circuit to the available cells (called “synthesis”)
 - Place and route the cells
 - Let the plant physically build your circuit



FPGAs – Field Programmable Gate Arrays

Existing hardware that can be configured to correspond to your circuit (“programmable hardware”)

- Basic concept (high level view):
 - FPGA vendors build huge arrays of LUTs (Look-Up-Tables) and switches (highly regular repeated physical structure)
 - You can map your design to this hardware (the gates are mapped to LUTs and the wiring is mapped to the switches connecting the LUTs)
 - An FPGA bitfile stores how a given FPGA needs to be configured to realize your circuit (format is vendor-specific)
 - Load the bitfile into the FPGA and the FPGA realizes your circuit

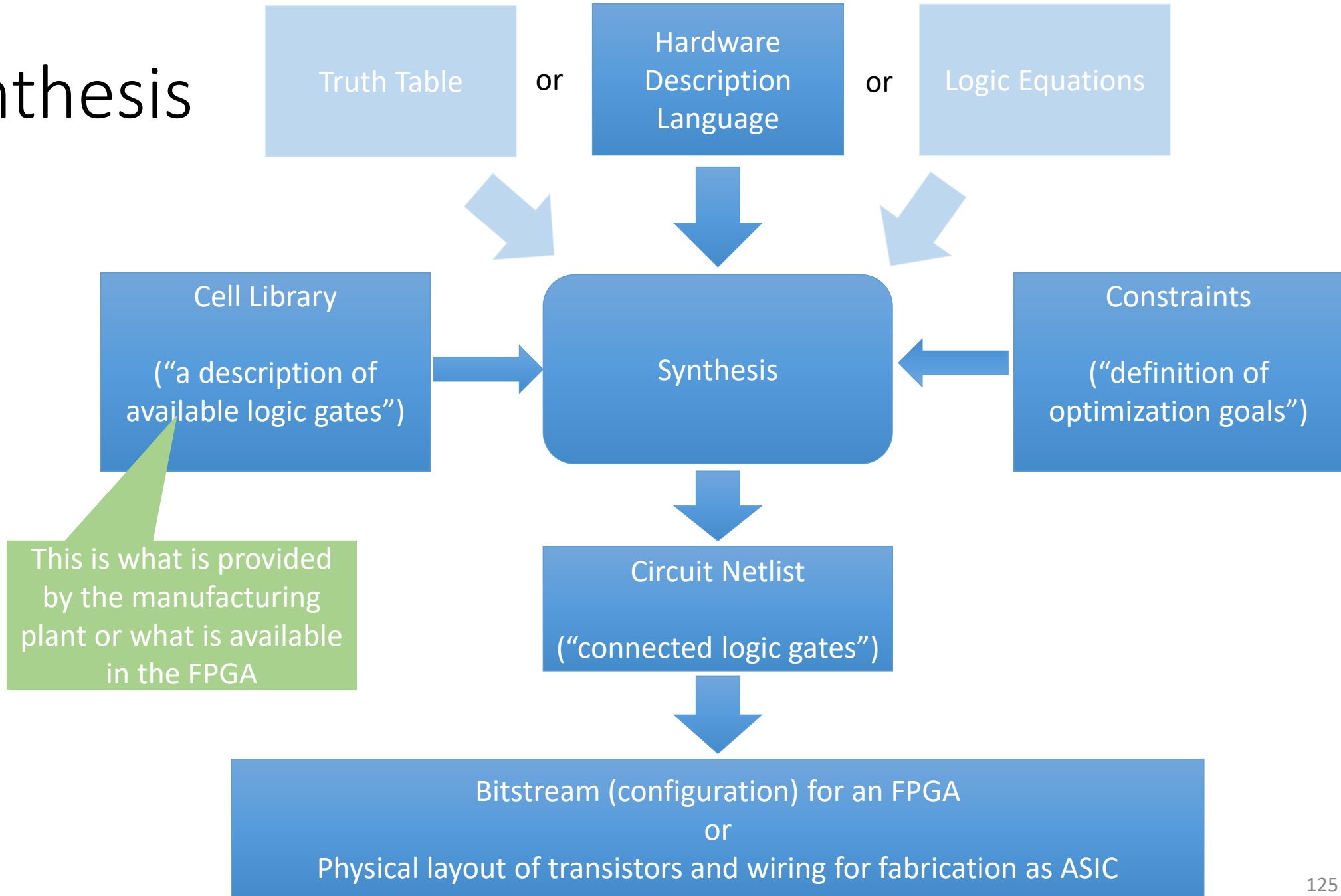
FPGA boards

- FPGAs are trade-off between hardware and software
 - Less efficient than hardware, but more efficient than software
 - Less expensive than hardware, but more expensive than software
- You can get small FPGA boards already for less than EUR 50.
 - Interested in putting your practical of this semester on physical hardware?
 - Basically, any FPGA works for this purpose; ICE40 FGPA's offer an open source toolflow based on the tools we also use in this class (e.g. <https://www.mouser.at/ProductDetail/Lattice/ICE40HX1K-STICK-EVN?qs=hJ2CX3hEdVEyBLaHAEXeIA%3D%3D>)

Mapping a Circuit to the Cells of an FPGA or ASIC

- Logic Synthesis is the process of mapping an abstract description (typically done in a hardware description language) of a circuit to a list of available logic gates
- Synthesis can be parametrized to optimize different properties, like speed or area

Logic Synthesis



The Toolchain

- iVerilog:
 - Simulator for Verilog code
- SV2V
 - Converts SystemVerilog to Verilog
- Yosys:
 - Synthesis Tool

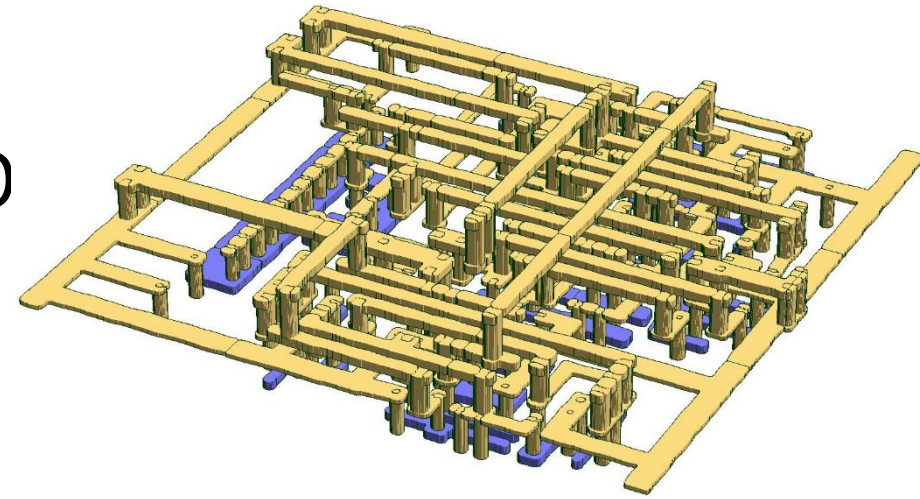
The Commands for Our Examples

- Make
 - **build:** Compile code
 - **run:** Run simulation
 - **view:** View simulation result in wave viewer (not relevant yet)

 - **syn:** Synthesize code
 - **build-syn:** compile synthesized code
 - **run-syn:** Run Simulation based on netlist (synthesis result)
 - **show:** Show netlist after synthesis

A Note on Complexity

The Complexity of a Microchip



David Carron

- Get an impression

<https://www.youtube.com/watch?v=Fxv3JoS1uY8>

https://www.youtube.com/watch?v=2z9qme_ygRI

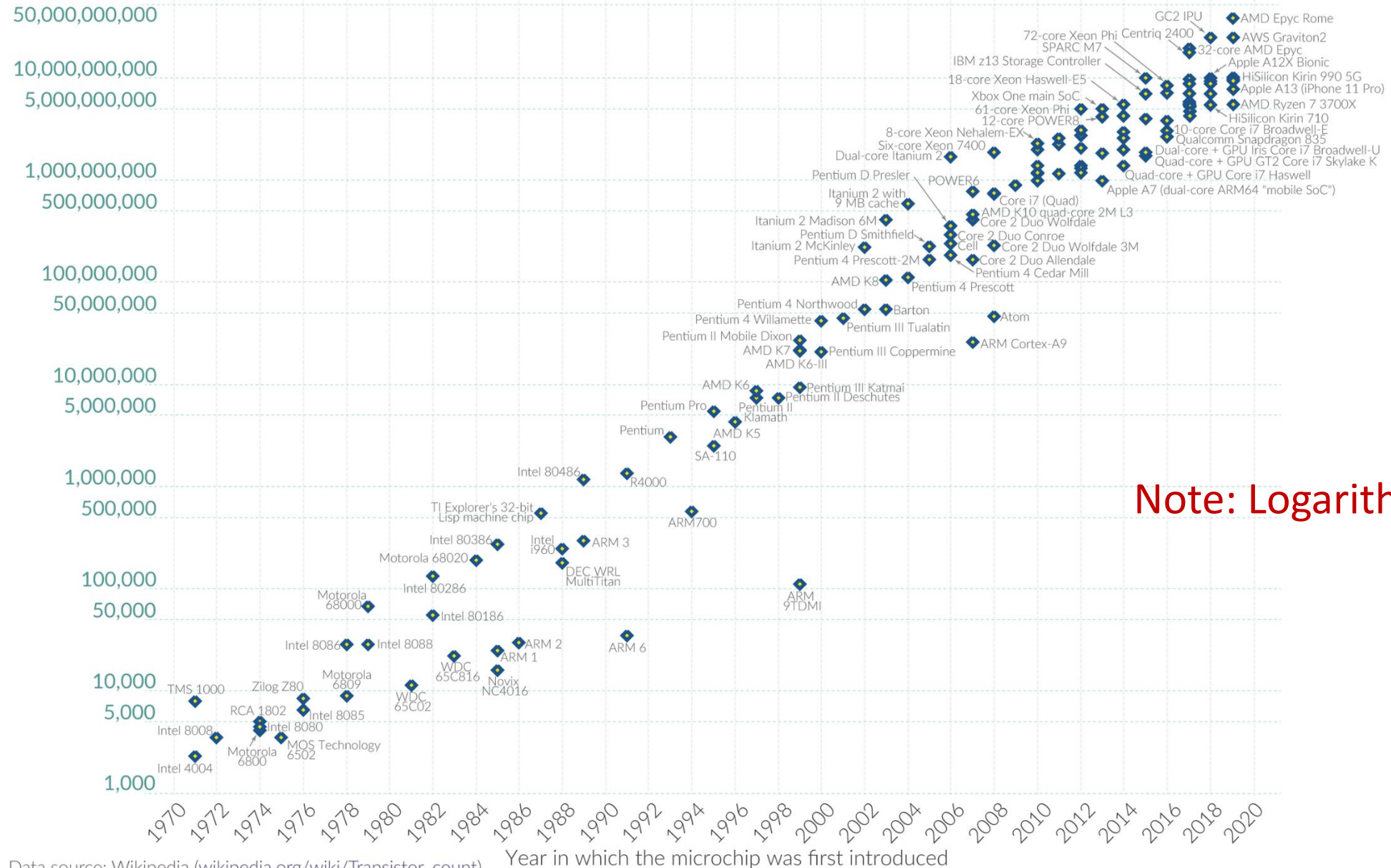
- Today's chips contain billions of transistors connected by multiple layers of metal

Recent Example: Apple M1 Ultra has about 114.000.000.000 transistors

Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Transistor count



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

Linear Scaling

