Computer Organization and Networks

(INB.06000UF, INB.07001UF)

Chapter 1 - Combinational Circuits

Winter 2020/2021

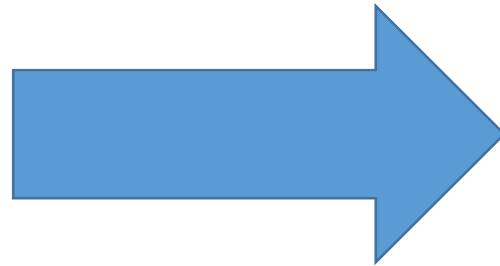
Stefan Mangard, www.iaik.tugraz.at

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
- …

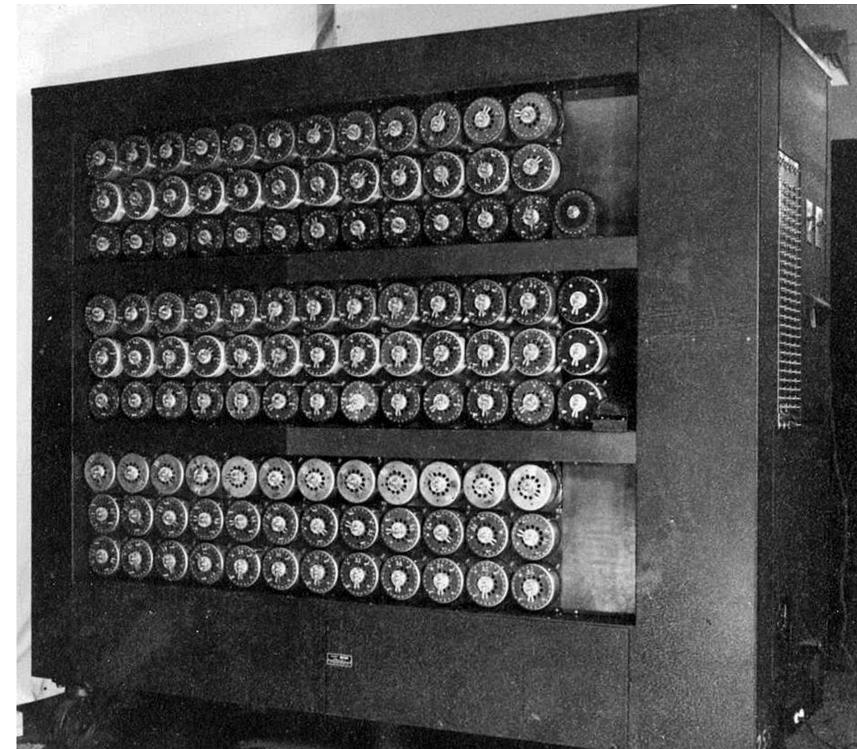
Examples of Computation Machines

Enigma
an electromechanical encryption machine

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

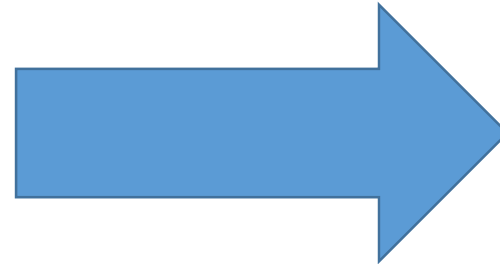
„British Bombe" by Alan Turing

An electromechanical machine to break Enigma



We Need to Map our Progra

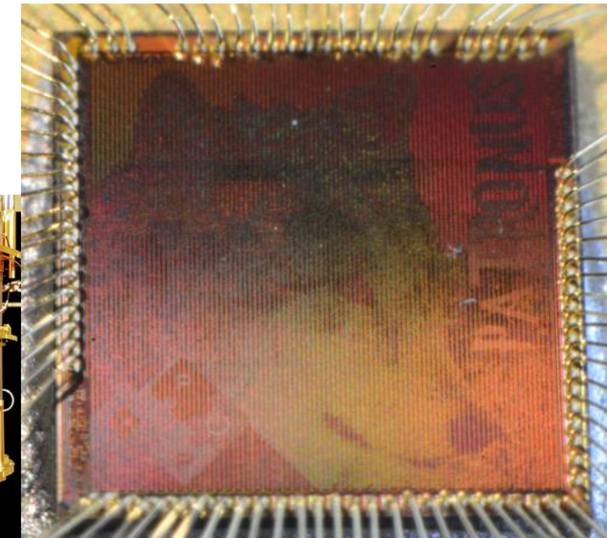
1 + 1 = ?



```
include <stdio.h>

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

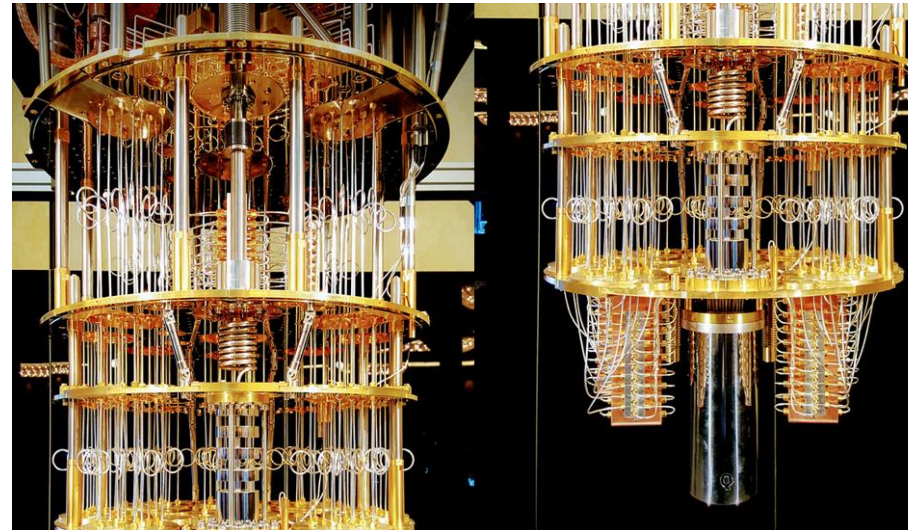
Zuse Z1

ComputerGeek via Wikipedia CC BY-SA 3.0



CMOS Processor

(http://asic.ethz.ch)

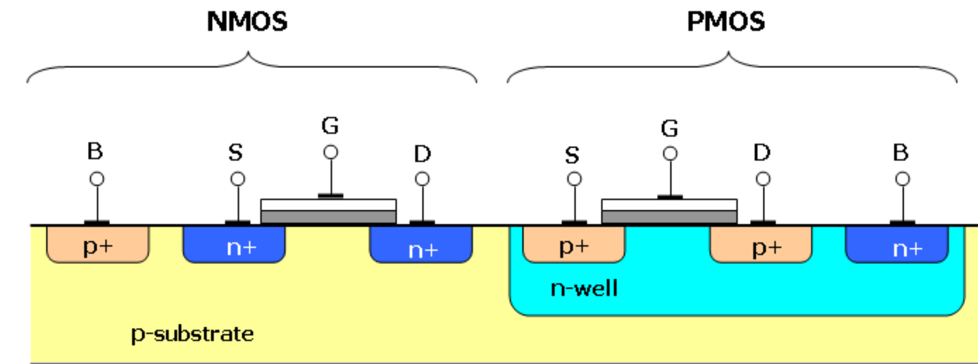


IBM quantum computer

(Lars Plougmann via flickr CC BY-SA 2.0)

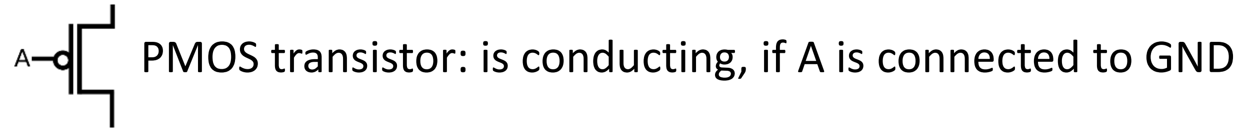
Complementary Metal-Oxide-Semiconductor (CMOS)

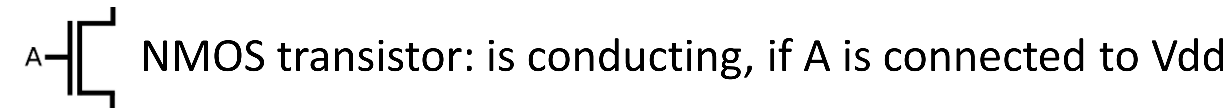
- Invented at Bell Labs by Mohamed Atalla and Dawon Kahng in 1959
- CMOS uses PMOS and NMOS transistors



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

Two Types of Transistors





- PMOS and NMOS transistors are essentially switches
  - PMOS: A=0 → switch is open; A=1 → switch is closed
  - NMOS: A=0 → switch is closed; A=1 → switch open

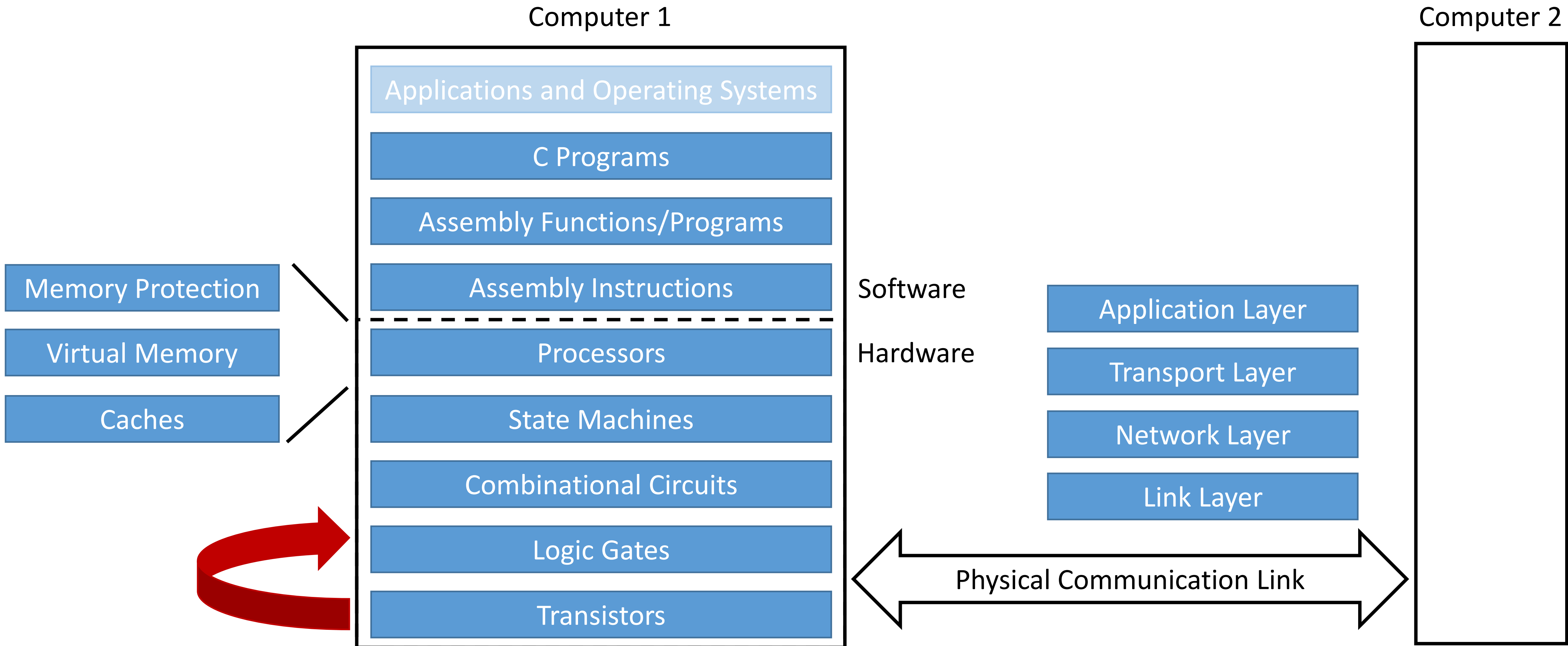
- **How do we build a computer from these two types of transistors?**

We Need Two Things

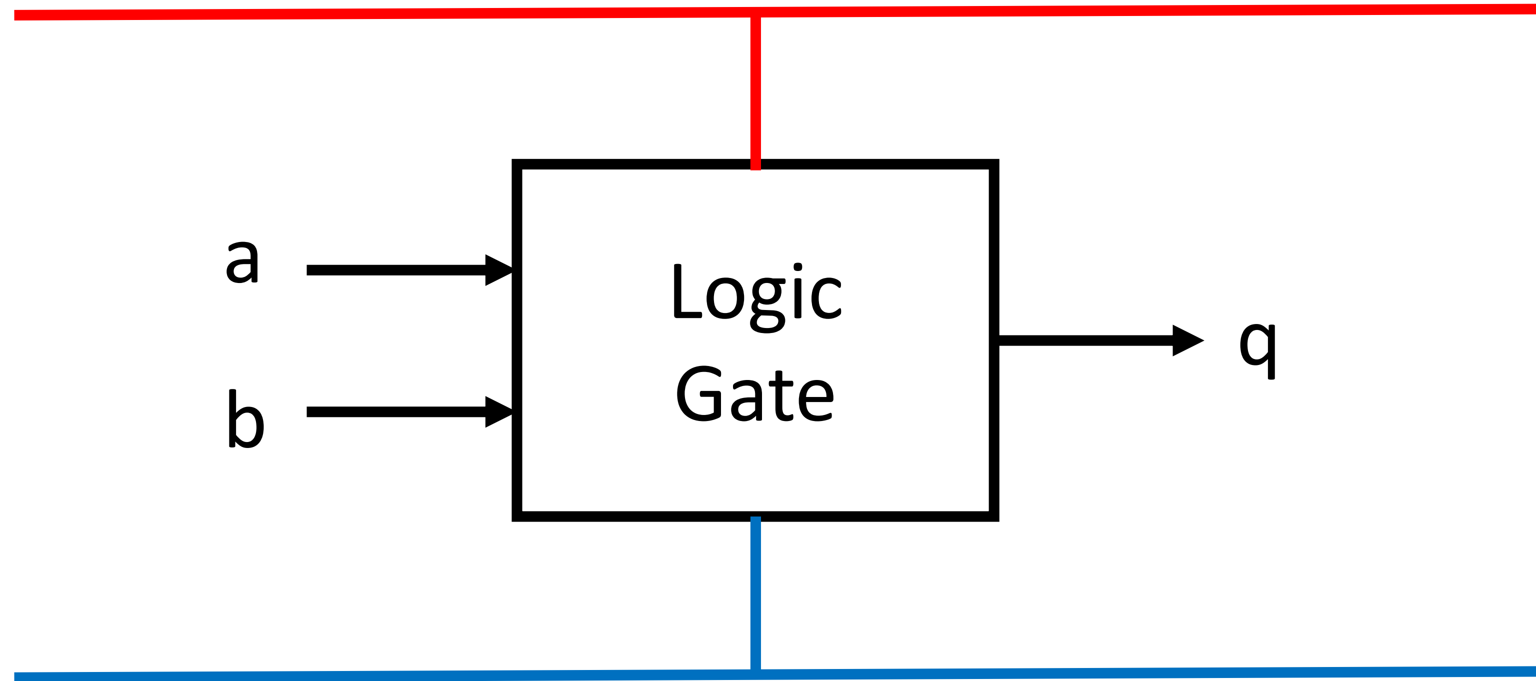
- Computation
  - How to apply a function on input data to generate an output?
- Storage
  - How to store data and intermediate results?

Logic Gates

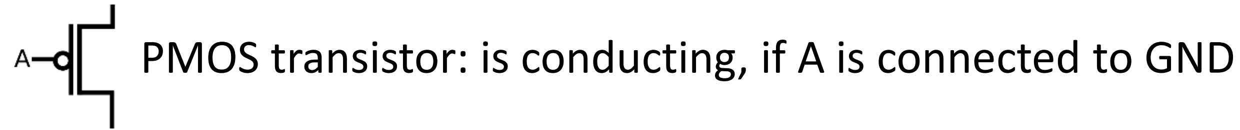
The Big Picture



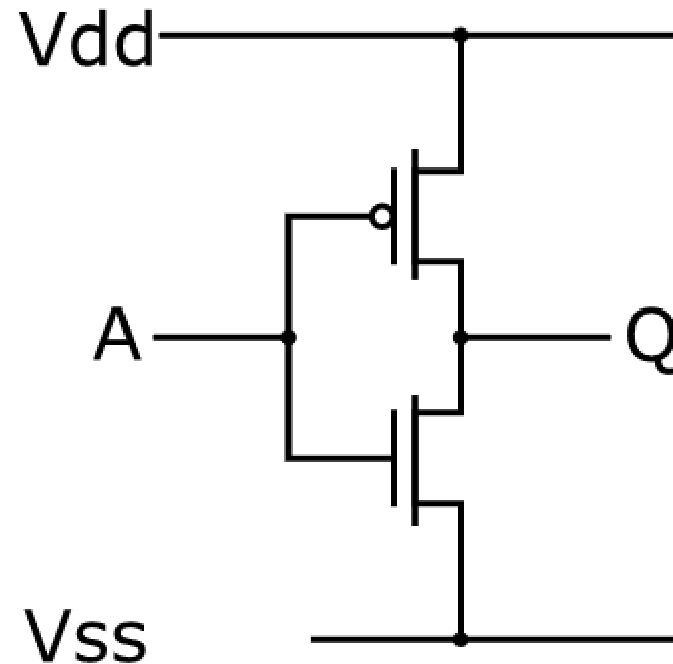
A Logic Gate – “The Smallest Functional Unit”



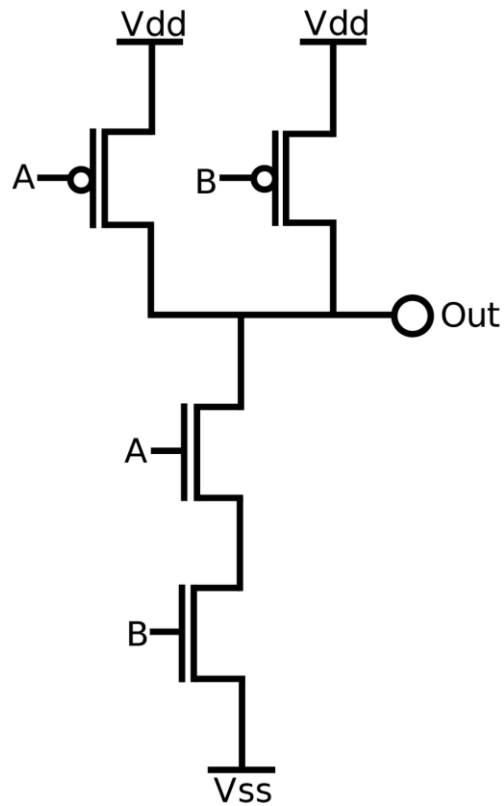
A short look inside – a CMOS Inverter



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

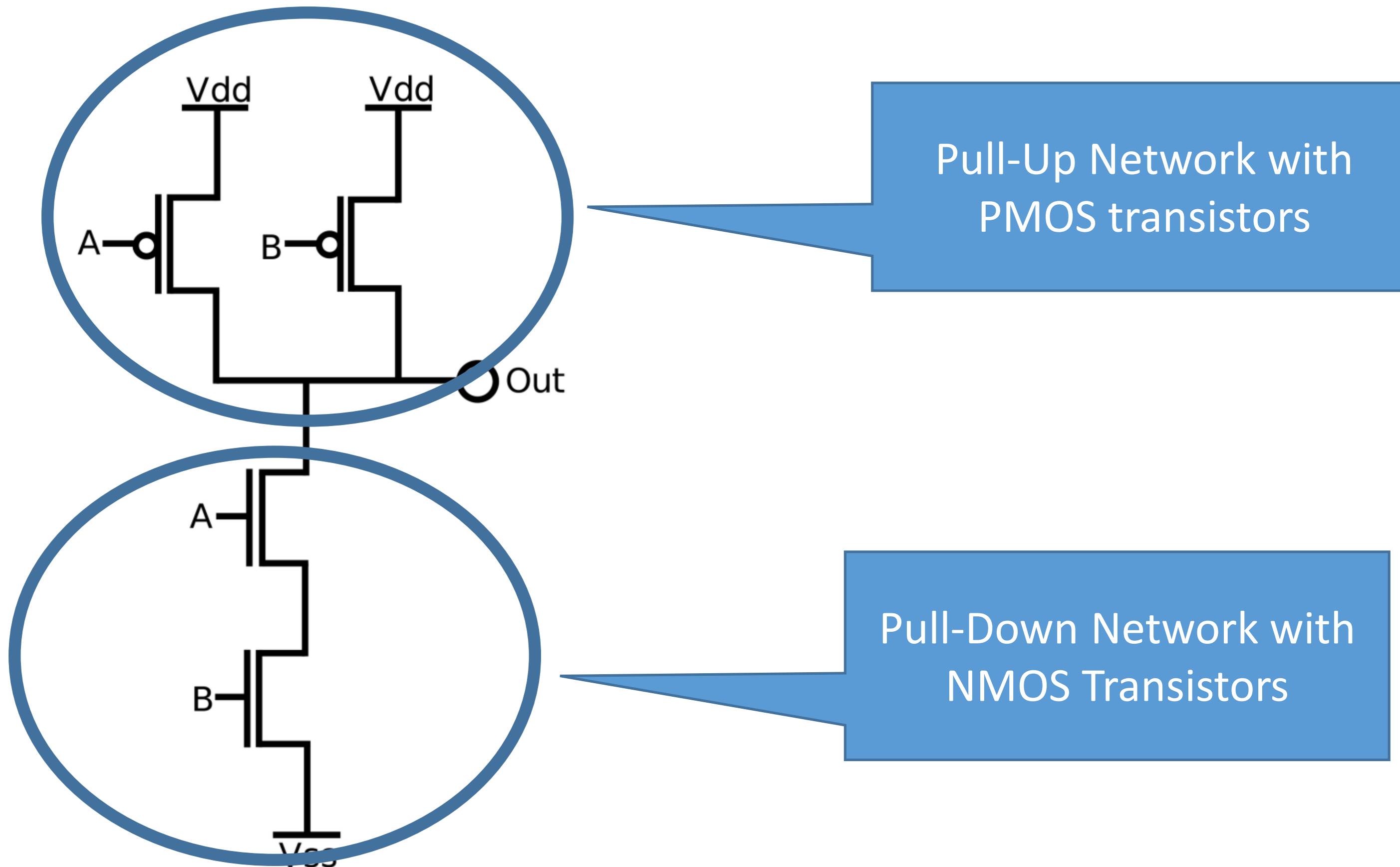


CMOS NAND gate



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

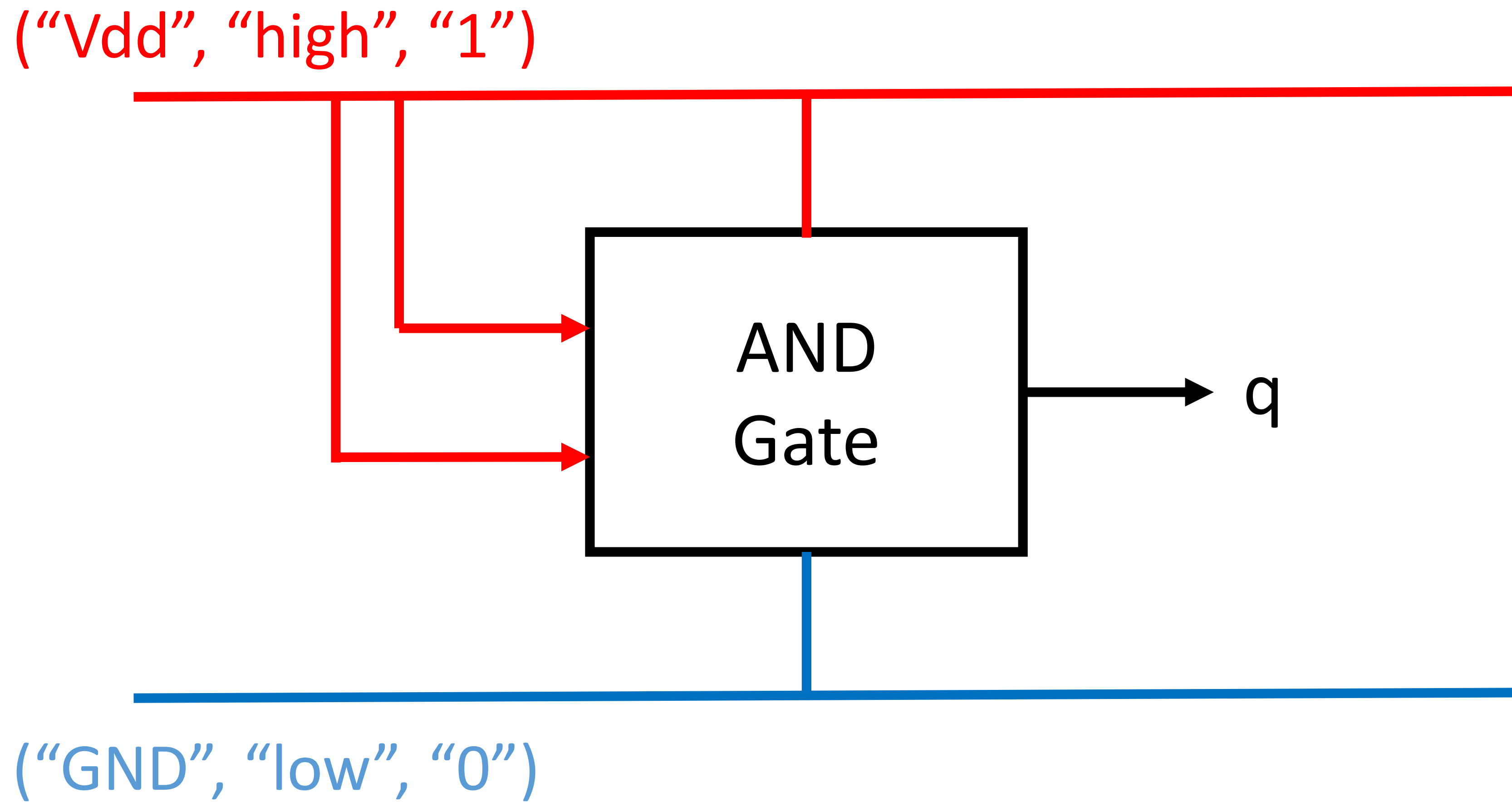
–> given static inputs, the output is either pulled up or pulled down

Based on this principle, different logic gates can be built

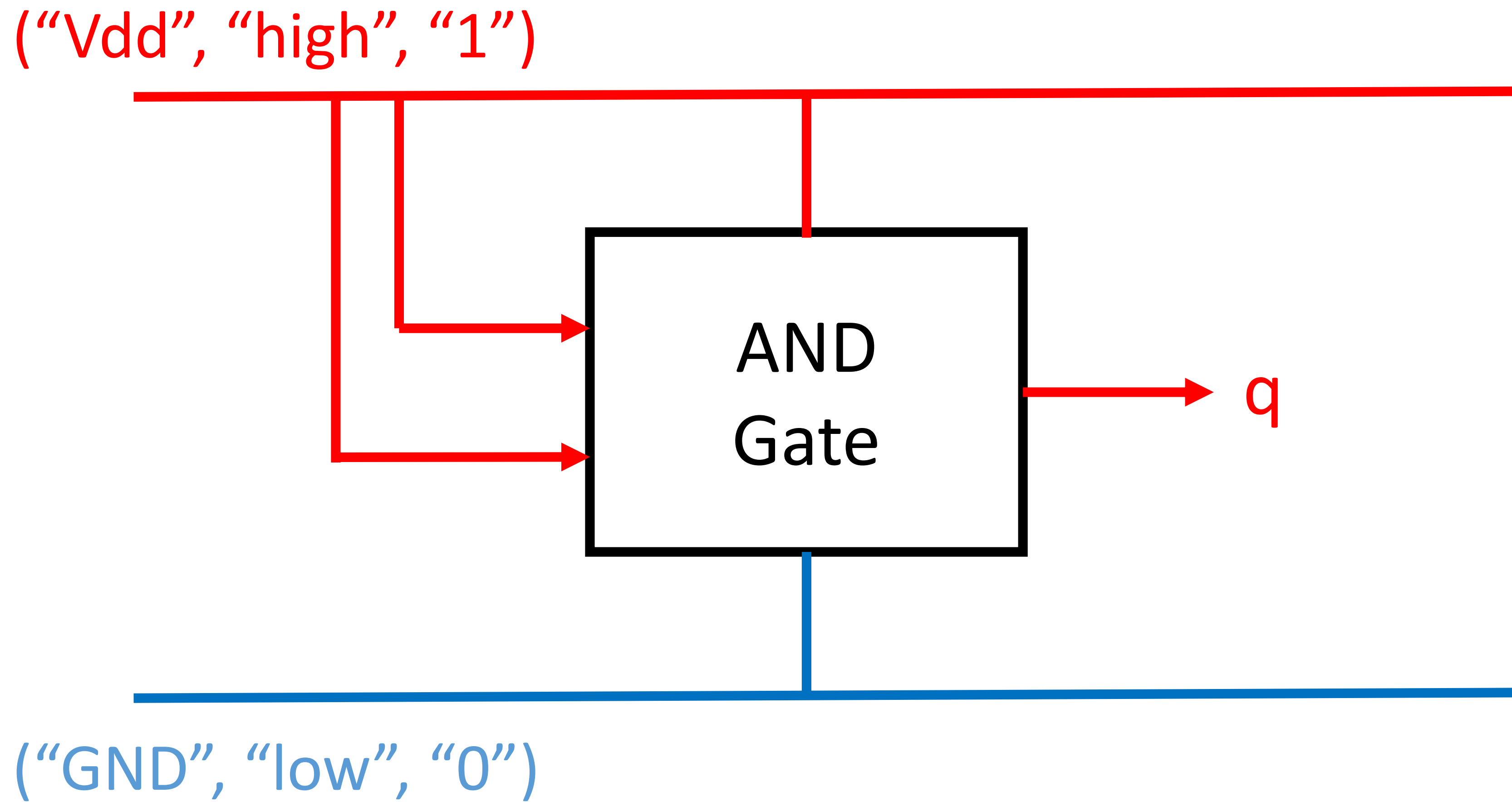
Building an AND Gate

- 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

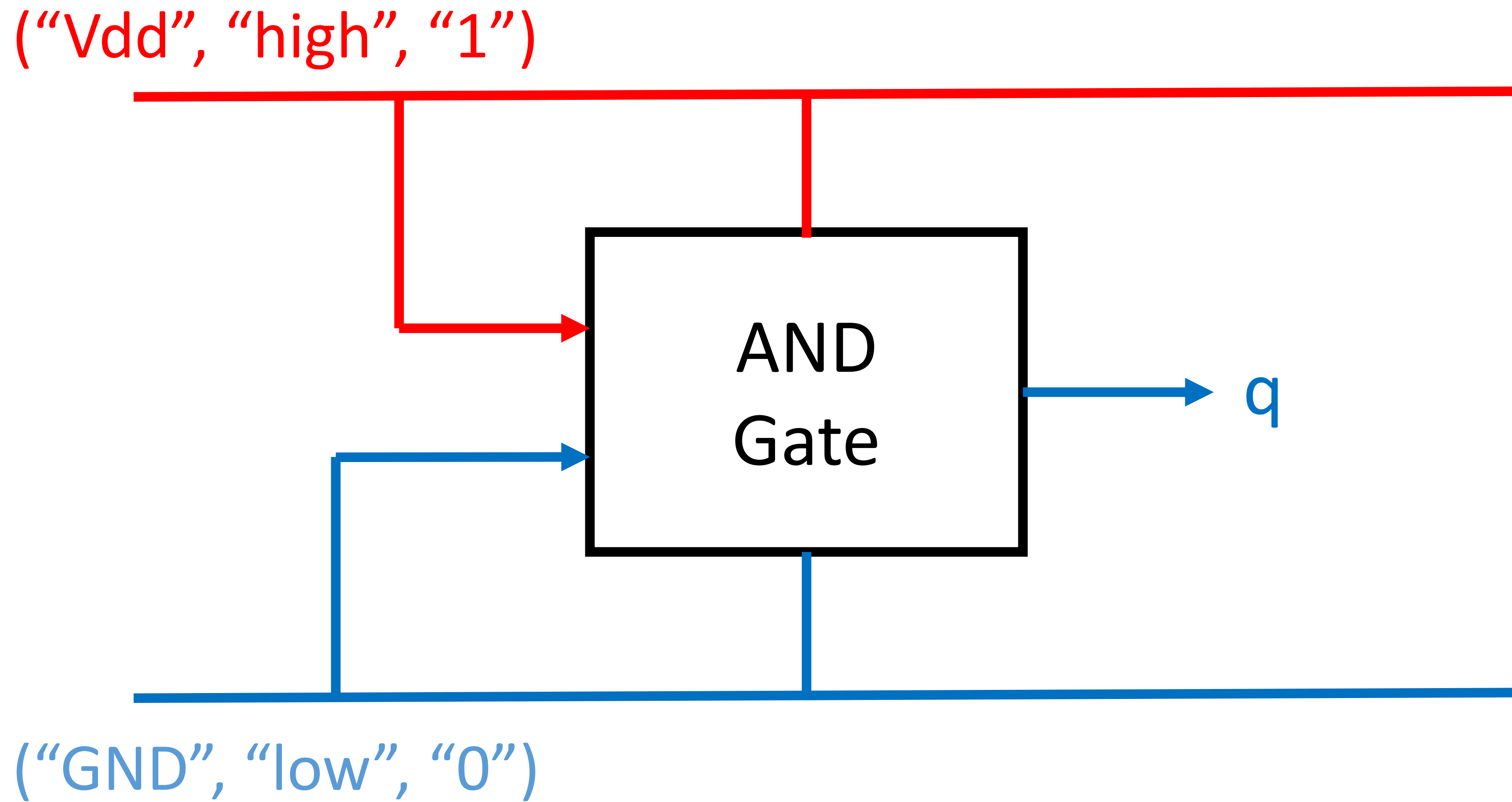
AND Gate



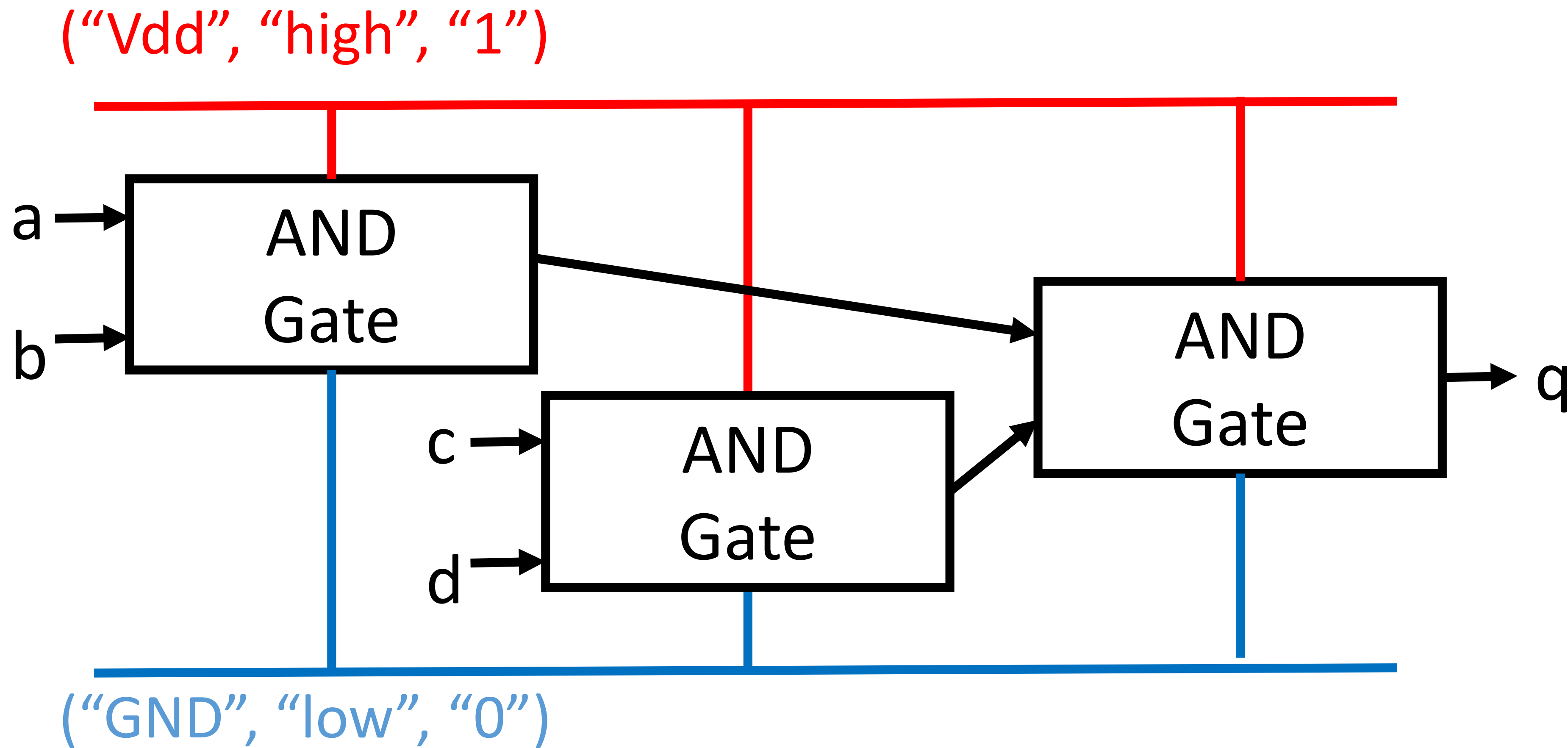
AND Gate



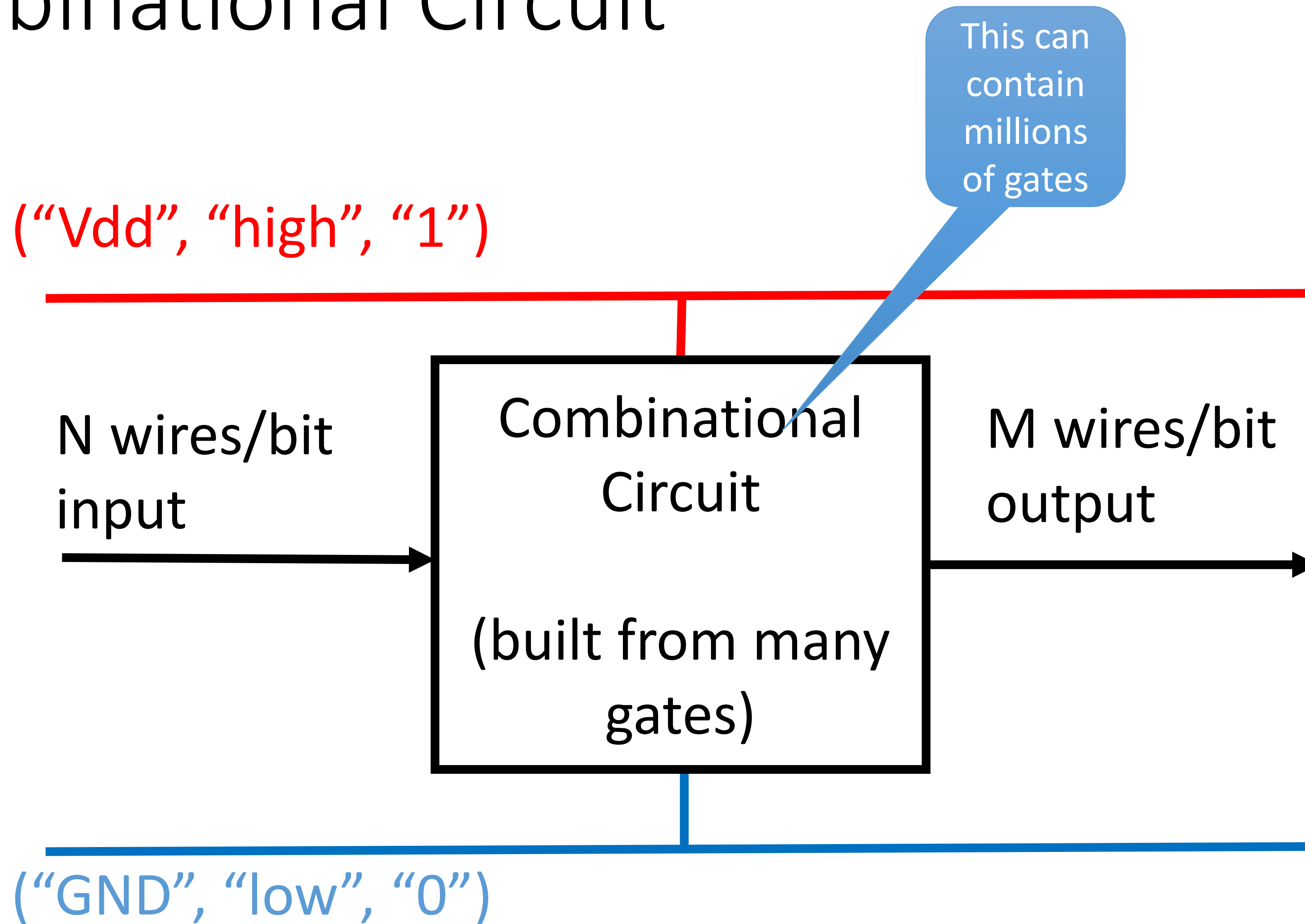
AND Gate



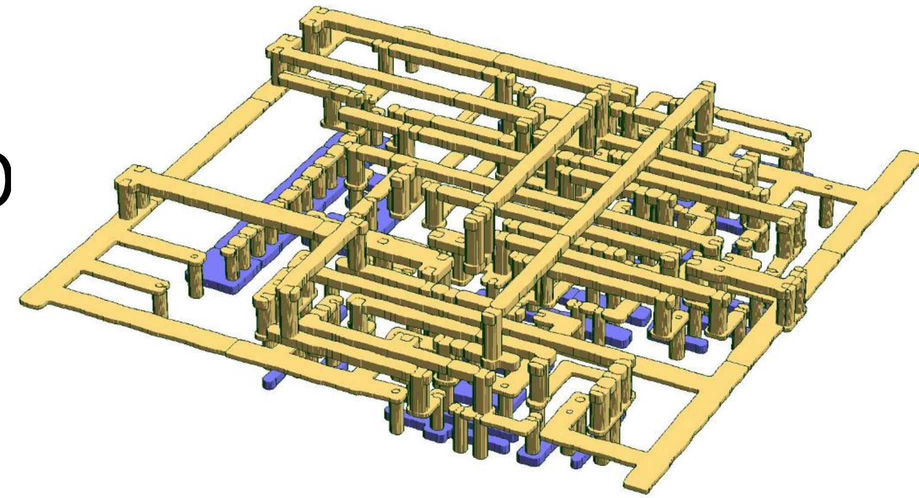
Cascading Gates



Combinational Circuit



The Complexity of a Microchip



David Carron

- Get an impression

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

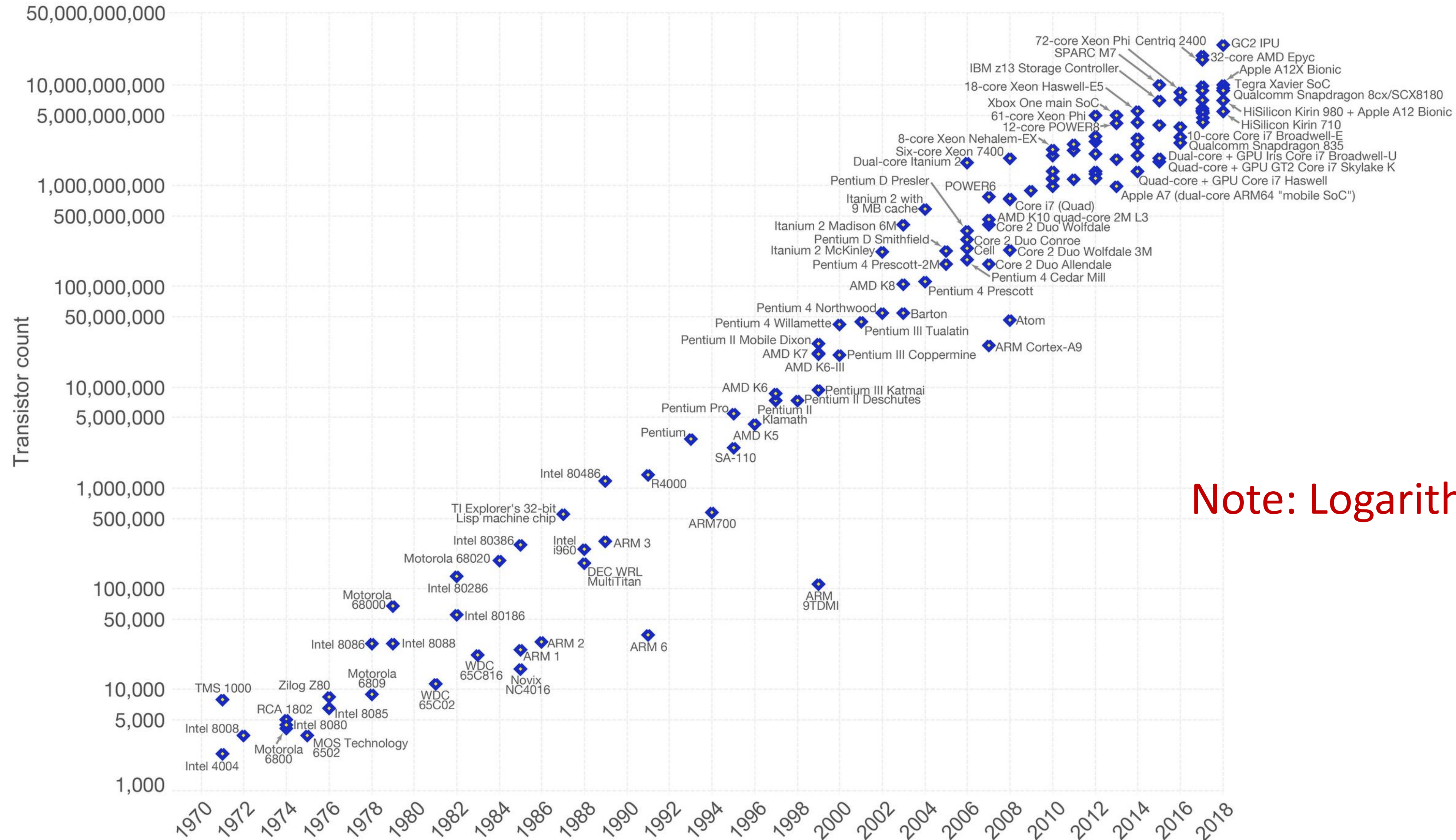
  https://www.youtube.com/watch?v=2z9qme_ygRI

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

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Our World in Data

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

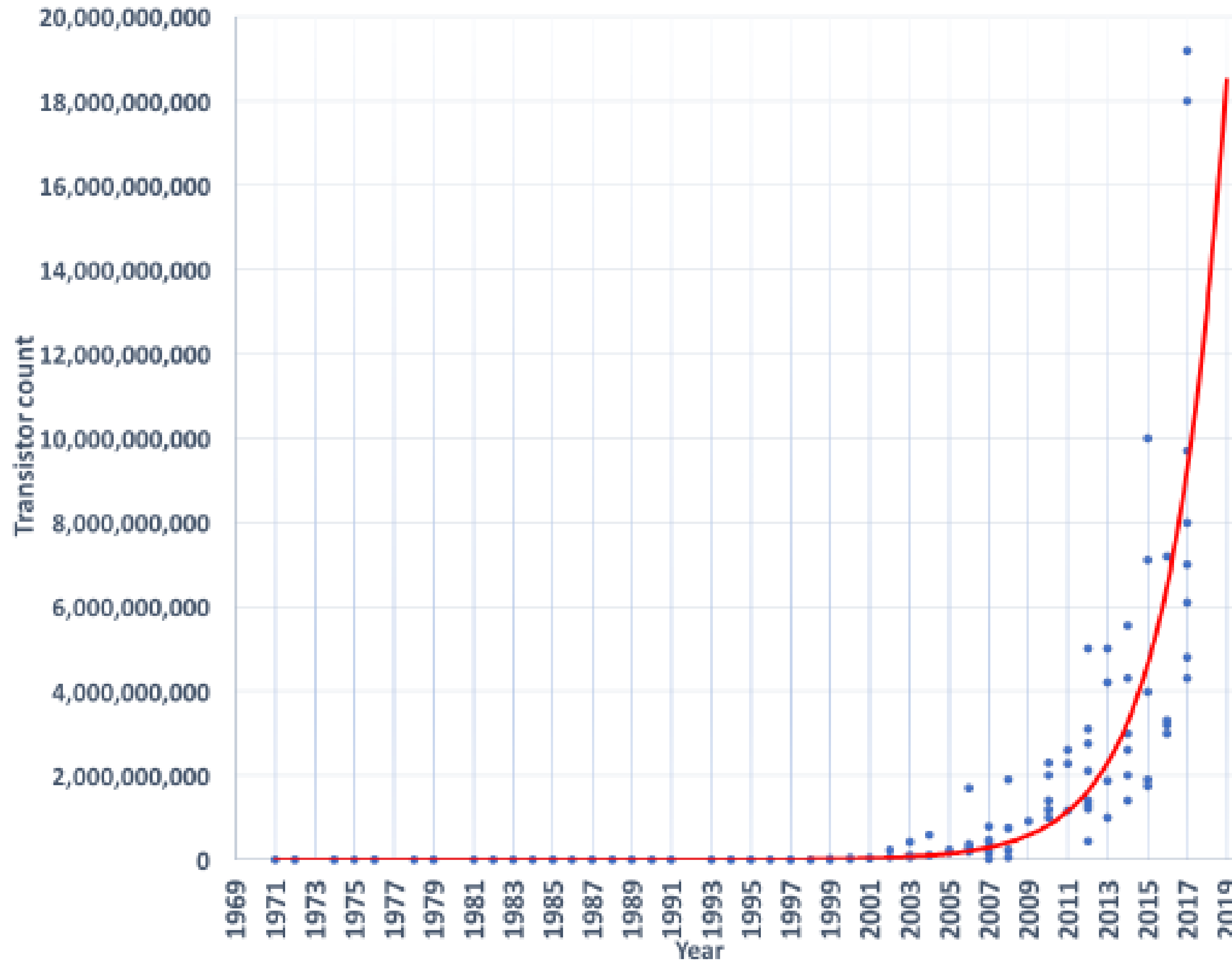


Note: Logarithmic Scale!

Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.

Linear Scaling



Beauty and Joy of Computing by University of California, Berkeley and Education Development Center, Inc, CC-BY-SA-NC
https://bjc.edc.org/bjc-r/cur/programming/6-computers/3-history-impact/2-moore.html

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

B^k → **B**, where **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 **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

$y = f(x_1, x_0)$

Input variables

x_1	x_0	
0	0	
0	1	
1	0	
1	1	

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

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

$n=2$

2^n

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 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.

Just to make sure…

$$2^n$$

is not

$$n^2$$

exponential

square

Popular logic functions

- Inversion (1 input variable)

x	f(x)
0	1
1	0

Popular logic functions

- Inversion (1 input variable)
- AND function (with 2 input variables)

x	f(x)
0	1
1	0

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

Popular logic functions

- Inversion (1 input variable)
- AND function (with 2 input variables)
- OR function (with 2 input variables)

x	f(x)
0	1
1	0

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

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

Popular logic functions

- Inversion (1 input variable)
- AND function (with 2 input variables)
- OR function (with 2 input variables)
- XOR function (with 2 input variables)

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

Popular logic functions

- Inversion (1 input variable)
- AND function (with 2 input variables)
- OR function (with 2 input variables)
- XOR function (with 2 input variables)
- Buffer (double inversion)

Popular logic functions

- Inversion (1 input variable)
- AND function (with 2 input variables)
- OR function (with 2 input variables)
- XOR function (with 2 input variables)
- Buffer (double inversion)
- NAND function (AND followed by inversion)

Popular logic functions

- Inversion (1 input variable)
- AND function (with 2 input variables)
- OR function (with 2 input variables)
- XOR function (with 2 input variables)
- Buffer (double inversion)
- NAND function (AND followed by inversion)
- NOR function (OR followed by inversion)

Popular logic functions

- Inversion (1 input variable)
- AND function (with 2 input variables)
- OR function (with 2 input variables)
- XOR function (with 2 input variables)
- Buffer (double inversion)
- NAND function (AND followed by inversion)
- NOR function (OR followed by inversion)
- NXOR function (XOR followed by inversion)

Boolean Algebra

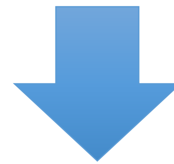
- Symbol for inversion: ~
- Symbol for AND: &
- Symbol for OR: |

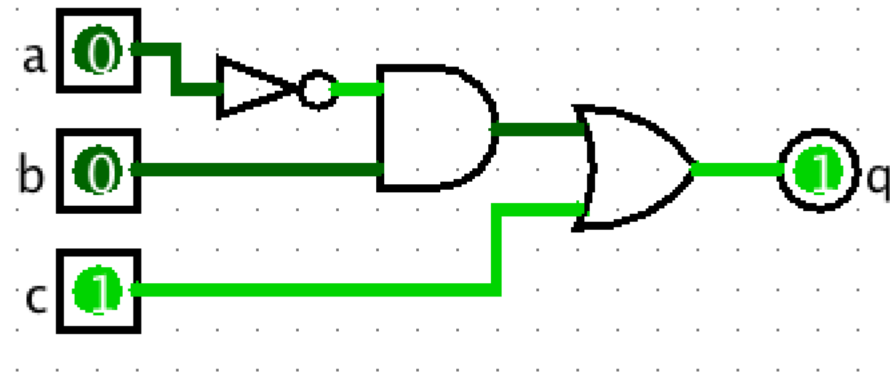
- Example:

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

Logic gates:
Technical realizations of logic functions

q = (~a & b) | c

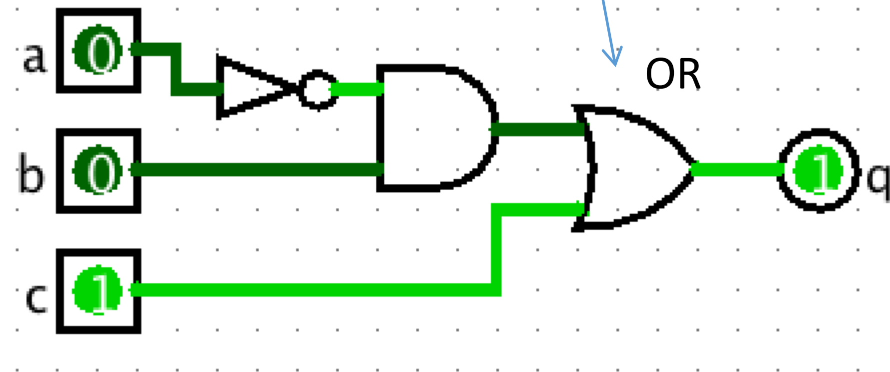




Logic gates:
Technical realizations of logic functions

http://www.burch.com/logisim/

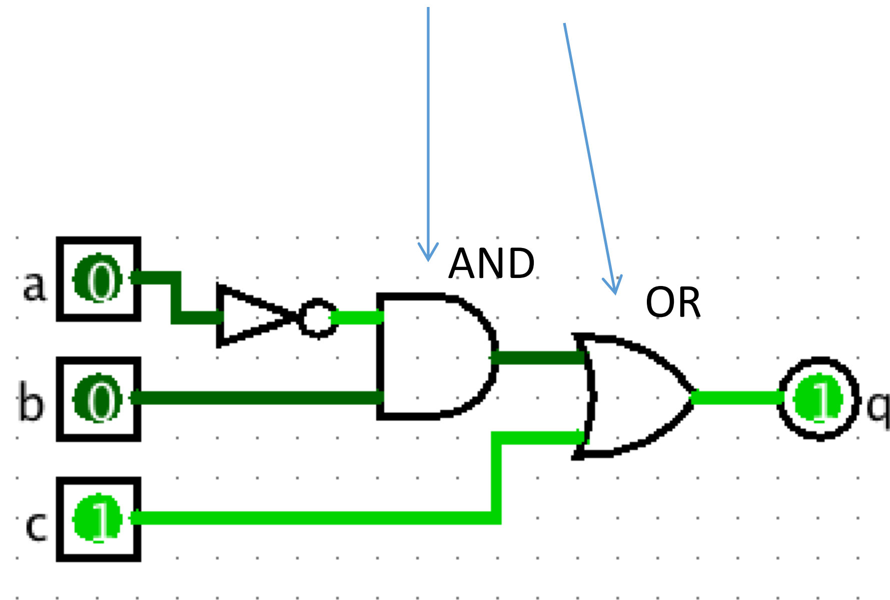
q = (~a & b) | c



Logic gates:
Technical realisations of logic functions

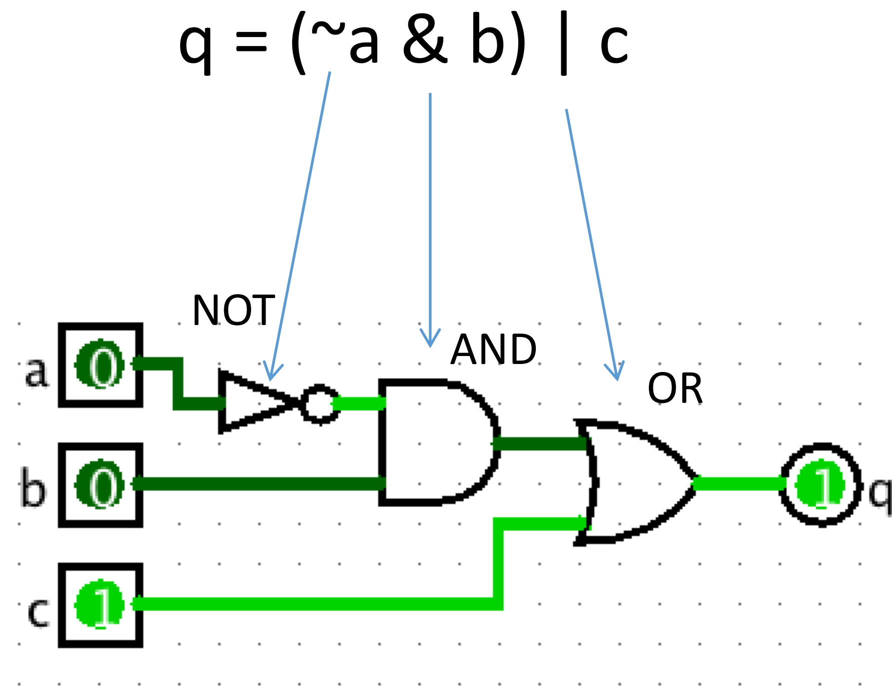
http://www.burch.com/logisim/

q = (~a & b) | c



Logic gates:
Technical realizations of logic functions

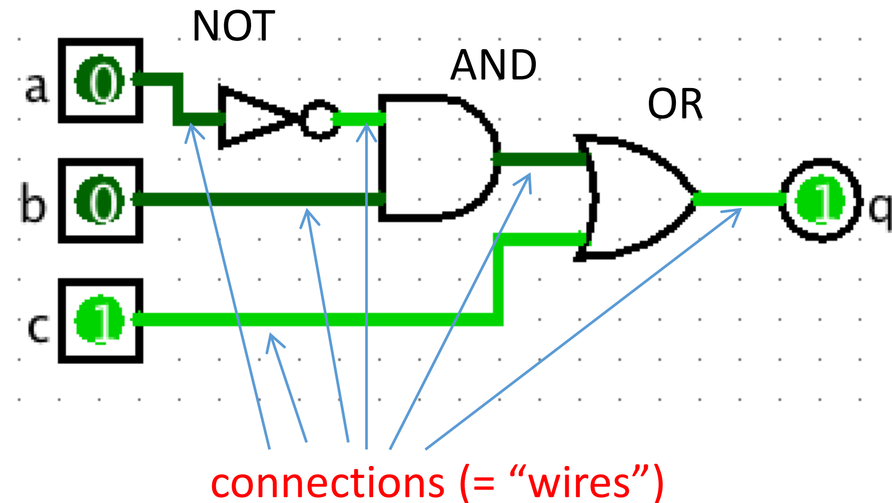
http://www.burch.com/logisim/



Logic gates:
Technical realizations of logic functions

http://www.burch.com/logisim/

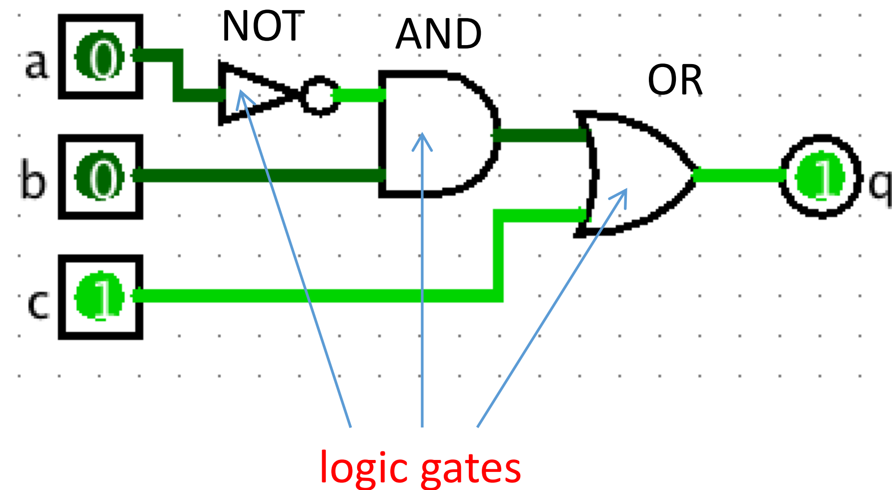
$$q = (\sim a \,\&\, b) \mid c$$



Logic gates:
Technical realizations of logic functions

http://www.burch.com/logisim/

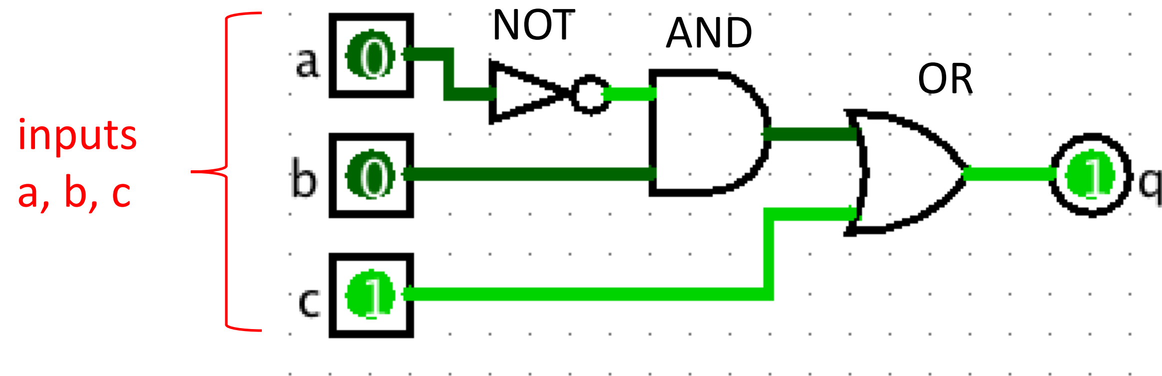
q = (~a & b) | c



Logic gates:
Technical realizations of logic functions

http://www.burch.com/logisim/

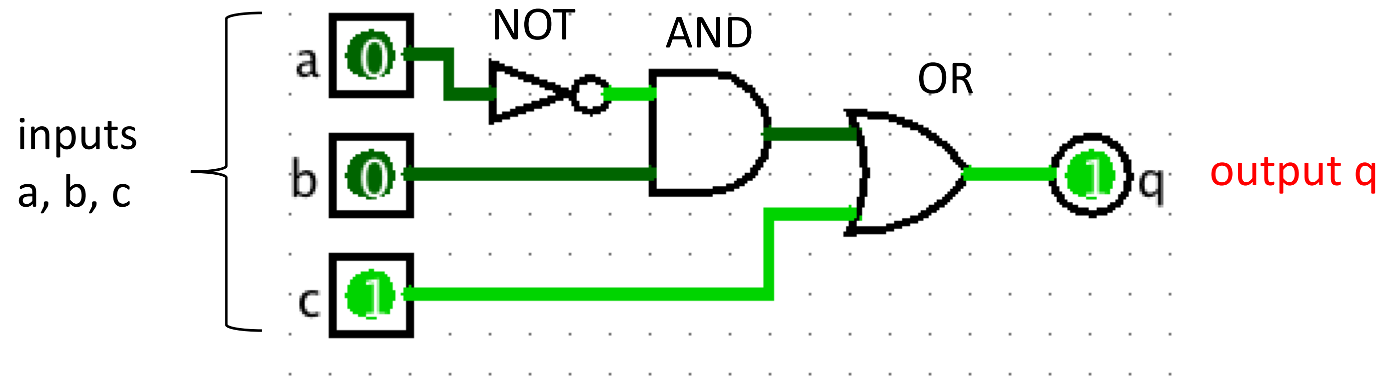
q = (~a & b) | c



Logic gates: Technical realizations of logic functions

http://www.burch.com/logisim/

q = (~a & b) | c

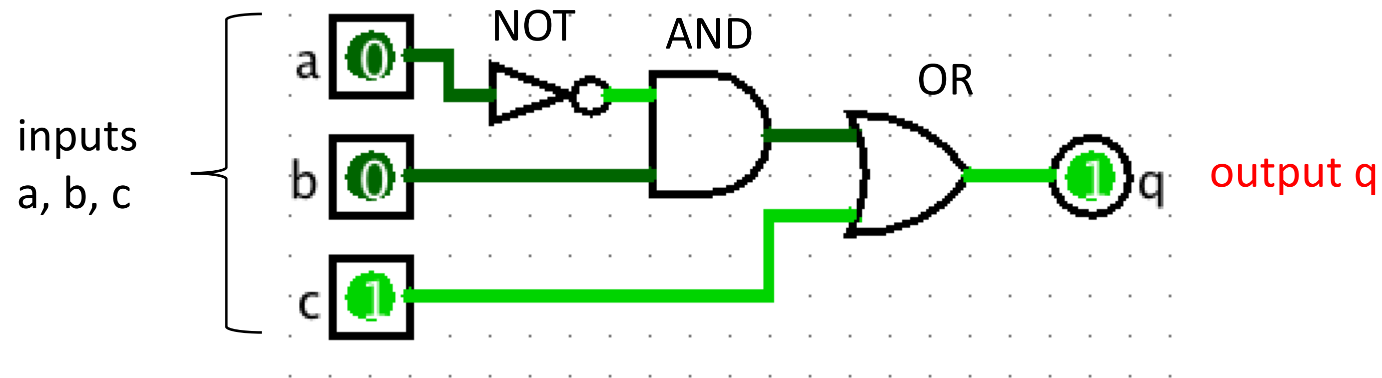


Logic gates:
Technical realizations of logic functions

http://www.burch.com/logisim/

q = (~a & b) | c

inputs
a, b, c



output q

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 = (~u & ~v & w) ...

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 = (~u & ~v & w) |
(~u & v & ~w) ...

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 = (~u & ~v & w) |
(~u & v & ~w) |

(u & ~v & ~w) …

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

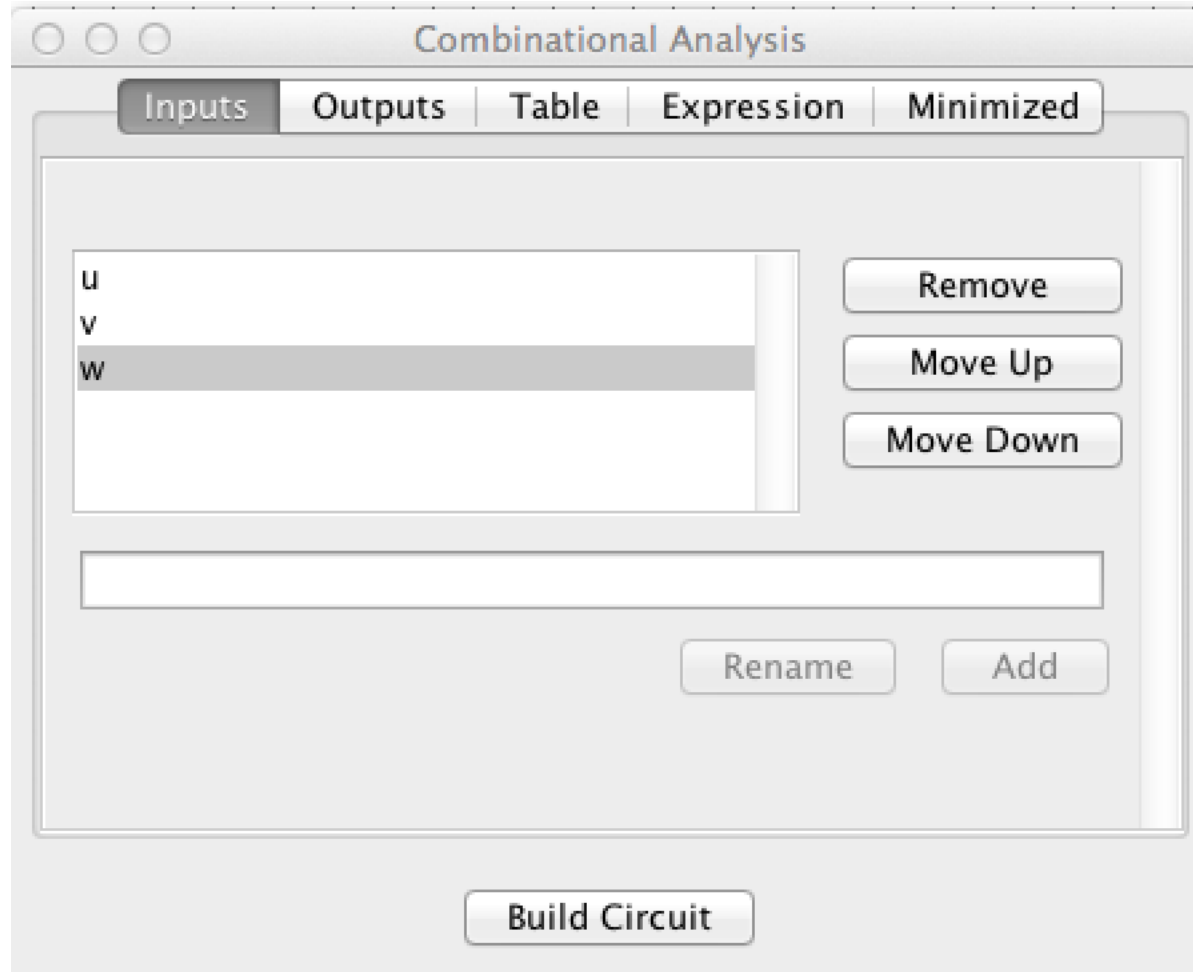
The logic function for s_0:

s_0 = (~u & ~v & w) |
(~u & v & ~w) |
(u & ~v & ~w) |
(u & v & w)

Implementing a logic function: With a little help from Logisim

http://www.burch.com/logisim/

Implementing a logic function: With a little help from Logisim

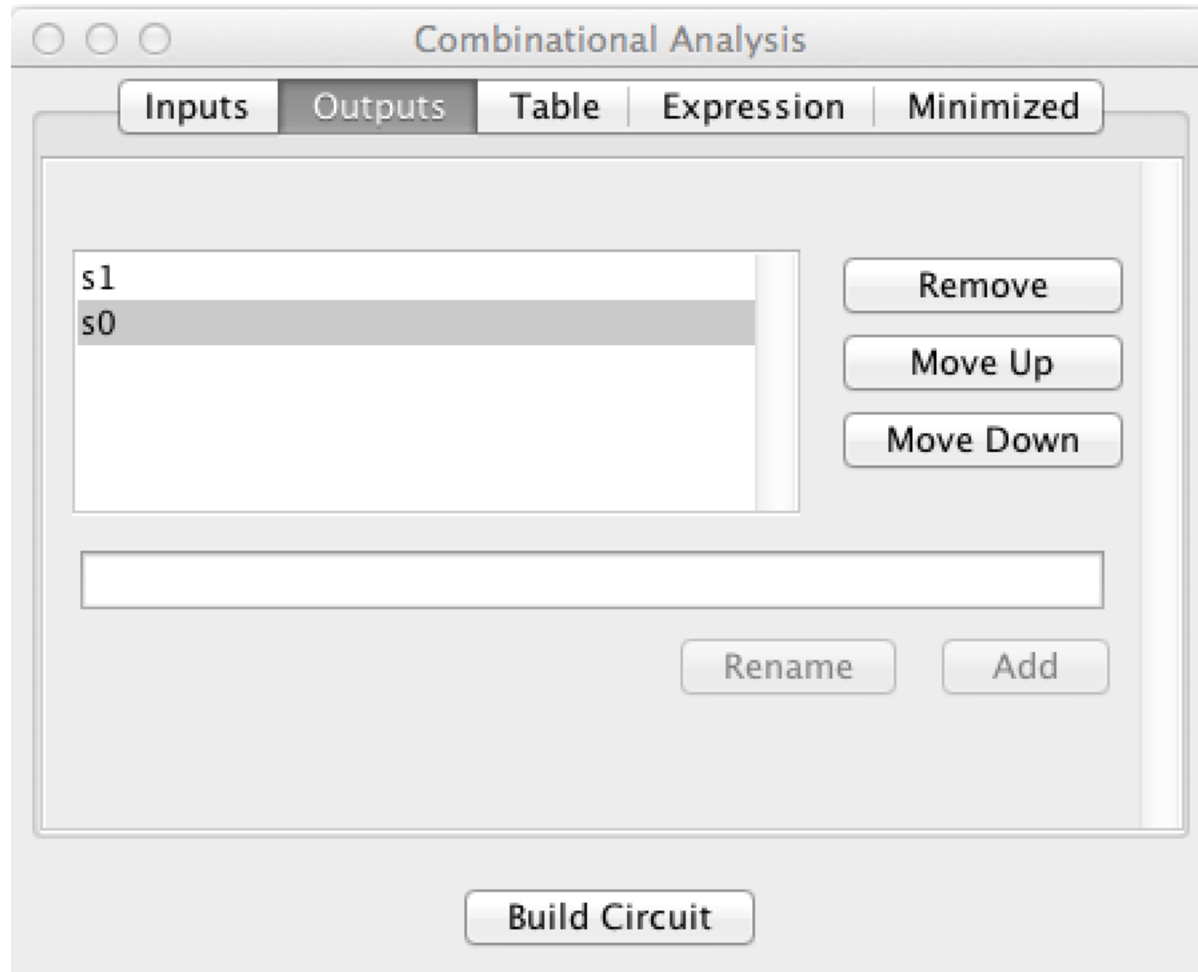


Start Logisim

Goto Project→Analyze Circuit

Specify the inputs

Implementing a logic function: With a little help from Logisim



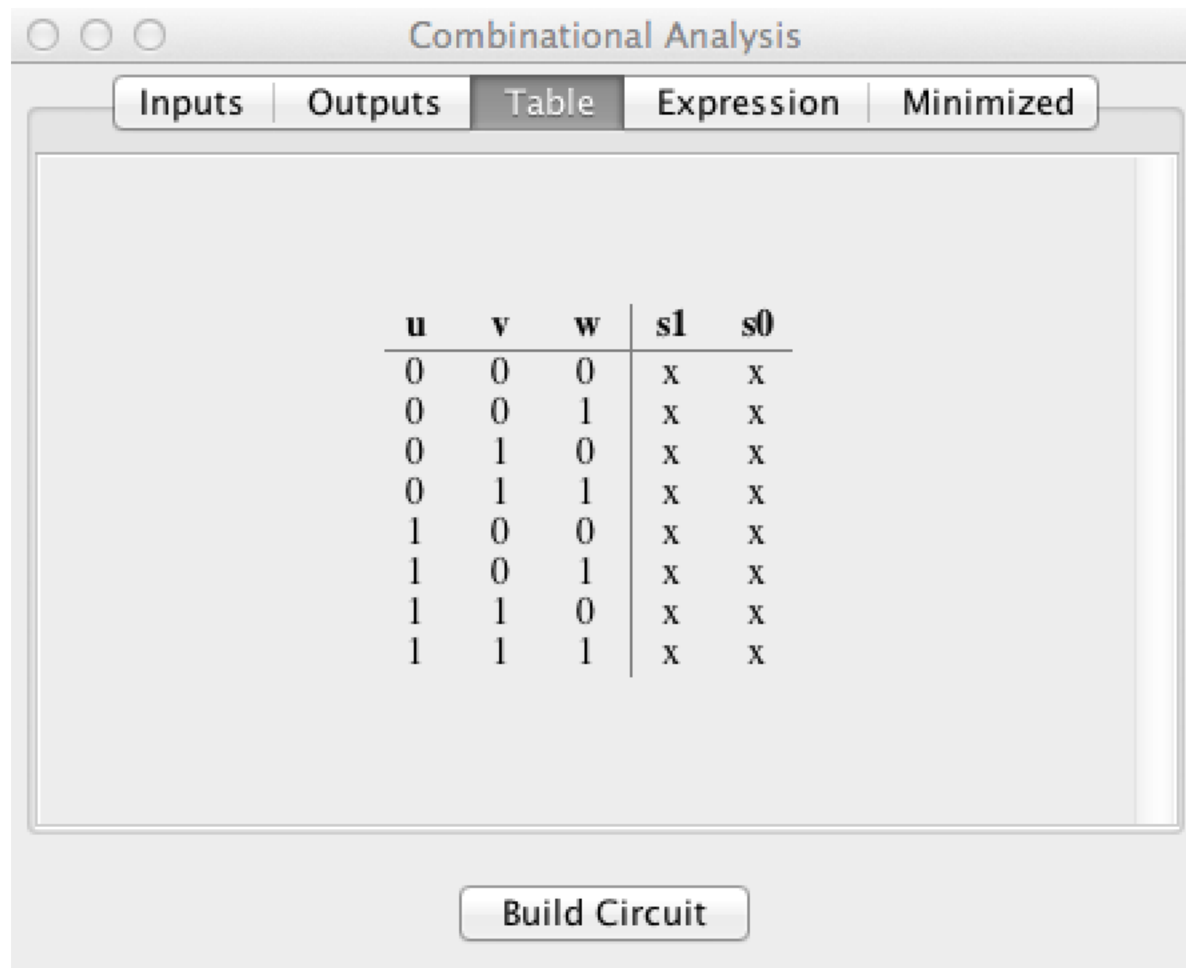
Start Logisim

Goto Project→Analyze Circuit

Specify the inputs

Specify the outputs

Implementing a logic function: With a little help from Logisim



Start Logisim

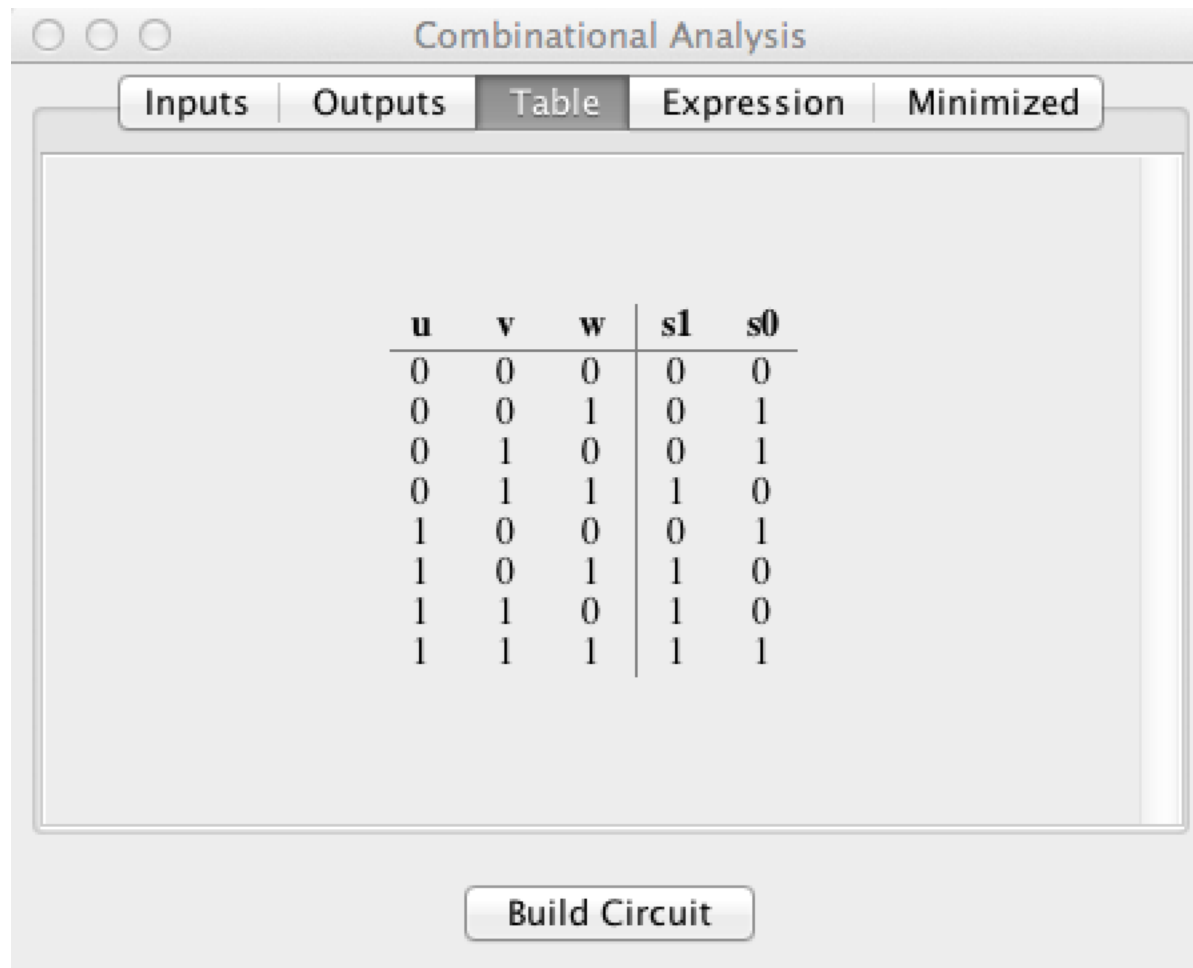
Goto Project→Analyze Circuit

Specify the inputs

Specify the outputs

Select truth table

Implementing a logic function: With a little help from Logisim



u	v	w	s1	s0
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 Logisim

Goto Project→Analyze Circuit

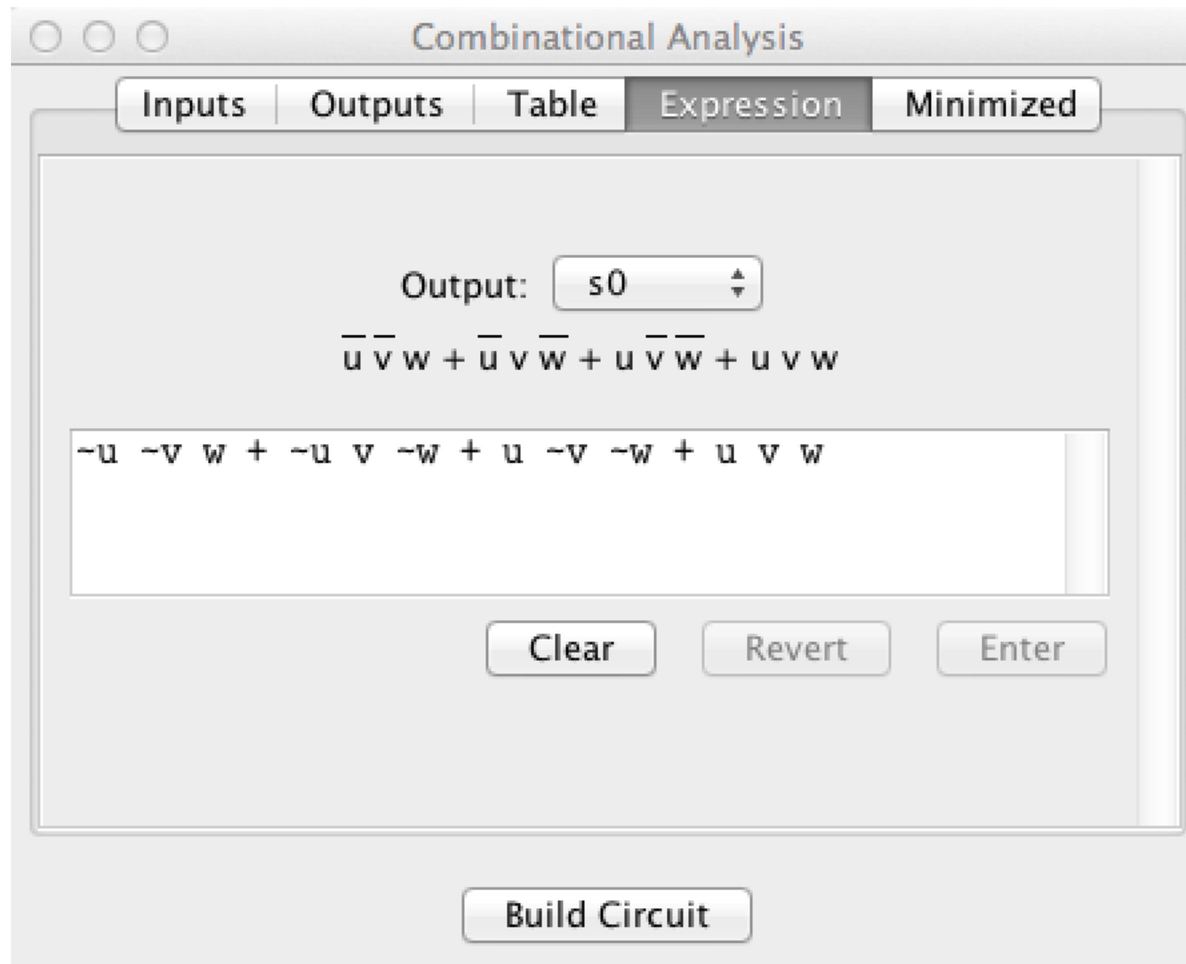
Specify the inputs

Specify the outputs

Select Truth table

Specify the output section of the truth table

Implementing a logic function: With a little help from Logisim



Start Logisim

Goto Project→Analyze Circuit

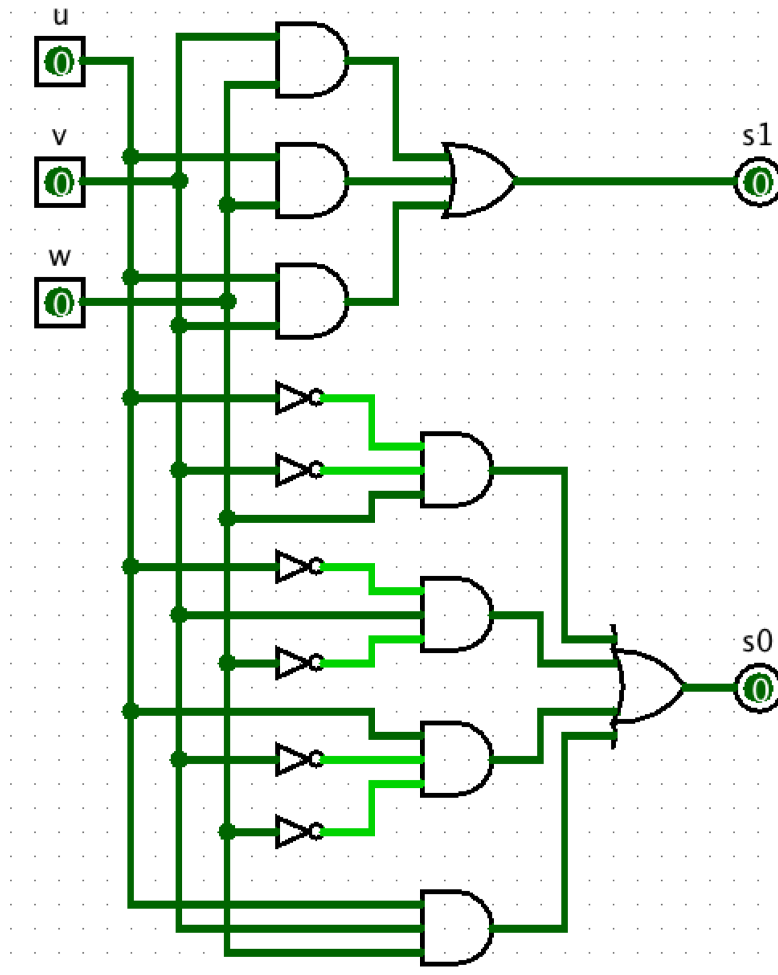
Specify the inputs

Specify the outputs

Specify the output section of the truth table

Optionally: Check the algebraic expressions

Implementing a logic function: With a little help from Logisim



Start Logisim

Goto Project→Analyze Circuit

Specify the inputs

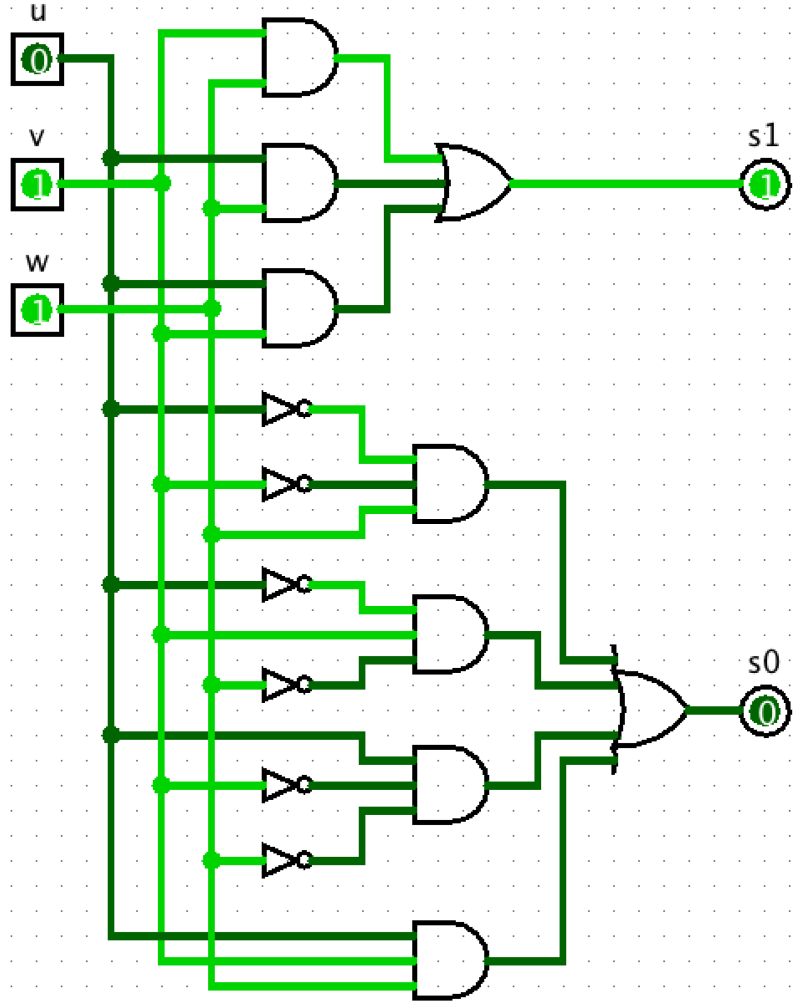
Specify the outputs

Specify the output section of the truth table

Optionally: Check the algebraic expressions

Click “Build Circuit” and “OK”.

Implementing a logic function: With a little help from Logisim



Switch to Simulation Mode:

Simulate circuit with
all possible input combinations.

Boolean Algebra

- Values of variables are only 0 or 1.
- 3 main operations:
  - Negation, also known as “inversion” ~
  - Conjunctions, also known as “ANDing” &
  - Disjunction, also known as “ORing” |

- 1 other popular operation: “exclusive OR” ^

Boolean Algebra

a | 0 = a

a | 1 = 1

a & 0 = 0

a & 1 = a

a ^ 0 = a

a ^ 1 = ~a

a | a = a

a | ~a = 1

a & a = a

a & ~a = 0

a ^ a = 0

a ^ ~a = 1

a | (a & b) = a

a & (a | b) = a

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

Boolean Algebra

Associative Law:

(a | b) | c = a | (b | c)

(a & b) & c = a & (b & c)

(a ^ b) ^ c = a ^ (b ^ c)

Commutative Law:

a | b = b | a

a & b = b & a

a ^ b = b ^ a

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

Boolean Algebra

Distributive Law:

a | (b & c) = (a | b) & (a | c)

a & (b | c) = (a & b) | (a & c)

a & (b ^ c) = (a & b) ^ (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:

~(a & b)   = ~a | ~b

~(a | b)   = ~a & ~b

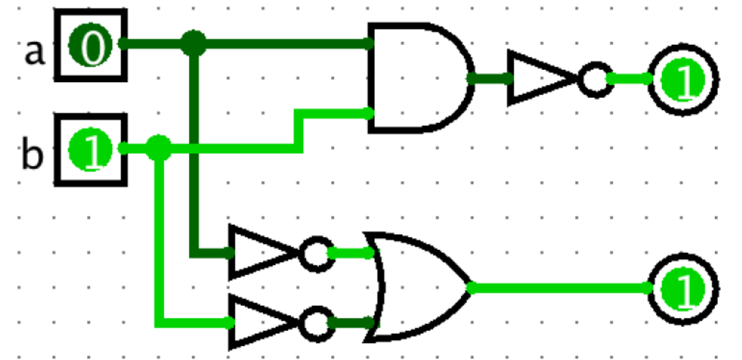
The proof of all these facts is rather easy: Take a truth table and check all possibilities.

Boolean Algebra

De Morgan's Law:

~(a & b) = ~a | ~b

~(a | b) = ~a & ~b

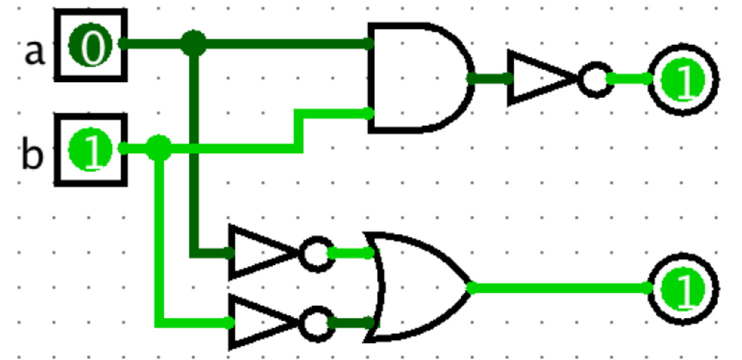


The proof of all these facts is rather easy: Take a truth table and check all possibilities.

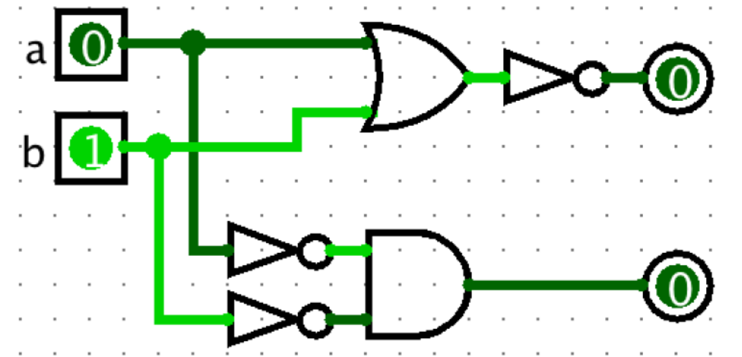
Boolean Algebra

De Morgan's Law:

~(a & b) = ~a | ~b



~(a | b) = ~a & ~b



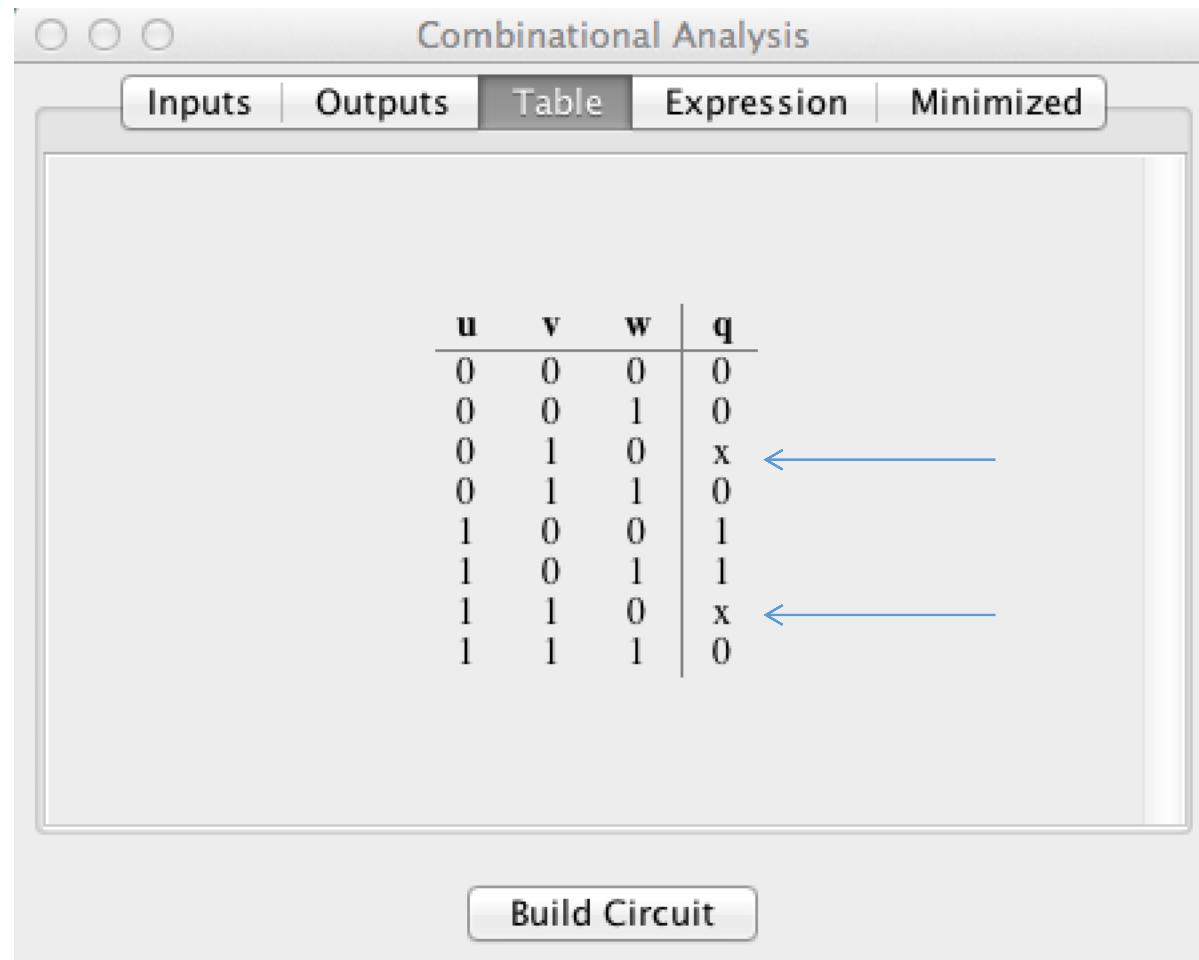
The proof of all these facts is rather easy: Take a truth table and check all possibilities.

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”.



u	v	w	q
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

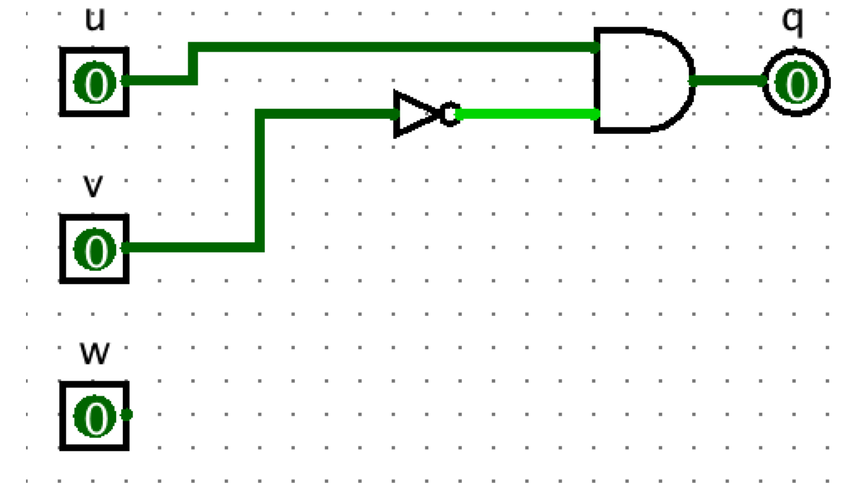
Result after "synthesizing" with Logisim

Combinational Analysis

Inputs | Outputs | Table | Expression | Minimized

u	v	w	q
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

Build Circuit



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

2^n

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

2^n

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

With n = 3, 4, …

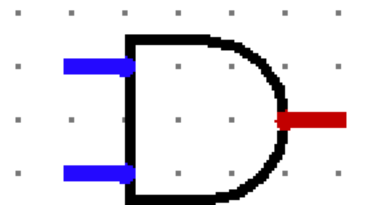
- n = 3, 2^3 = 8 lines, 2^8 = 256 functions
- n = 4, 2^4 = 16 lines, 2^{16} = 65536 functions
- n = 5, 2^5 = 32 lines, 2^{32} = 4294967296 functions
- n = 6, 2^6 = 64 lines, 2^{64} = 18446744073709551616 functions

Back to n = 2

- Some functions are "popular" and have names:

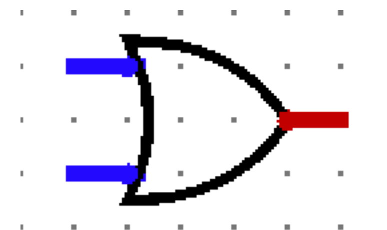
x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

AND



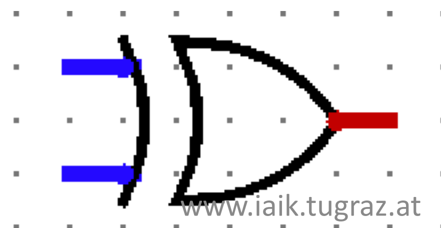
Popular 2-input functions

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	

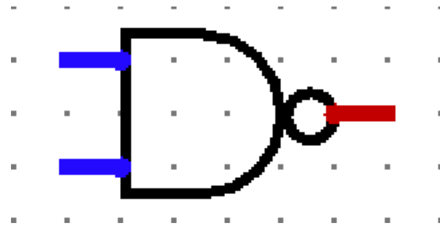


Popular 2-input functions

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	

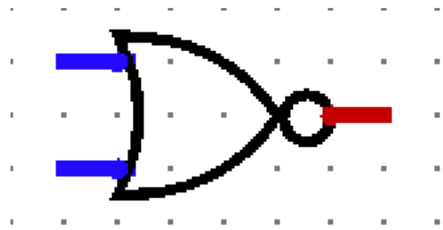


Popular 2-input functions



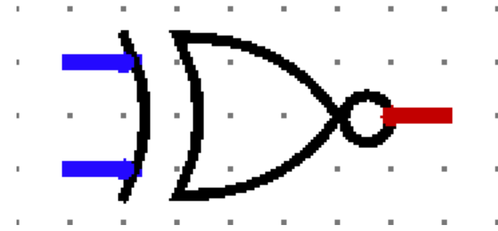
x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
								XOR	NAND	AND						OR	
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Popular 2-input functions



x_1	x_0	y_0	y_1 (NOR)	y_2	y_3	y_4	y_5	y_6 (XOR)	y_7 (NAND)	y_8 (AND)	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14} (OR)	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	y_1 (NOR)	y_2	y_3	y_4	y_5	y_6 (XOR)	y_7 (NAND)	y_8 (AND)	y_9 (NXOR)	y_{10}	y_{11}	y_{12}	y_{13}	y_{14} (OR)	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

			NOR						NAND		NXOR						
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				x_1		OR	

Some functions are trivial

			NOR						NAND		NXOR						
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		x_0		x_1		OR	

Some functions are trivial

			NOR						NAND		NXOR						
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						XOR		AND		x_0		x_1		OR	

Some functions are trivial

			NOR						NAND		NXOR						
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						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}
			NOR						NAND		NXOR						
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”

			NOR						NAND		NXOR						
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"

			NOR						NAND		NXOR		x_1 implies 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
		0			$\sim x_1$		$\sim x_0$	XOR		AND		x_0		x_1		OR	1

Some functions are “implications”

			NOR						NAND		NXOR		x_1 implies 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
		0			$\sim x_1$		$\sim x_0$	XOR		AND		x_0		x_1	x_1 is implied by x_0	OR	1

And some functions are “inverse implications”

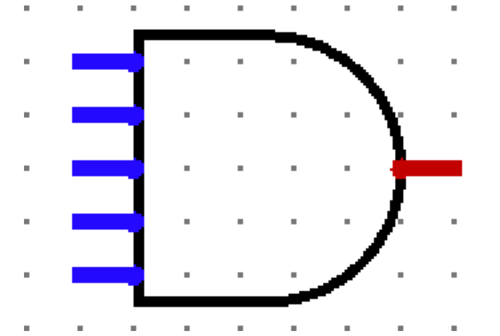
			NOR			x_1 does not imply x_0			NAND		NXOR		x_1 implies x_0				
x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
		0			$\sim x_1$		$\sim x_0$	XOR		AND		x_0		x_1	x_1 is implied by x_0	OR	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
		0	NOR	x_1 is not implied by x_0	$\sim x_1$	x_1 does not imply x_0	$\sim x_0$	XOR	NAND	AND	NXOR	x_0	x_1 implies x_0	x_1	x_1 is implied by x_0	OR	1

The popular functions can also have more than 2 inputs

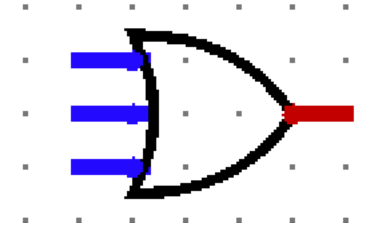
- Example: 5-input AND



- 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

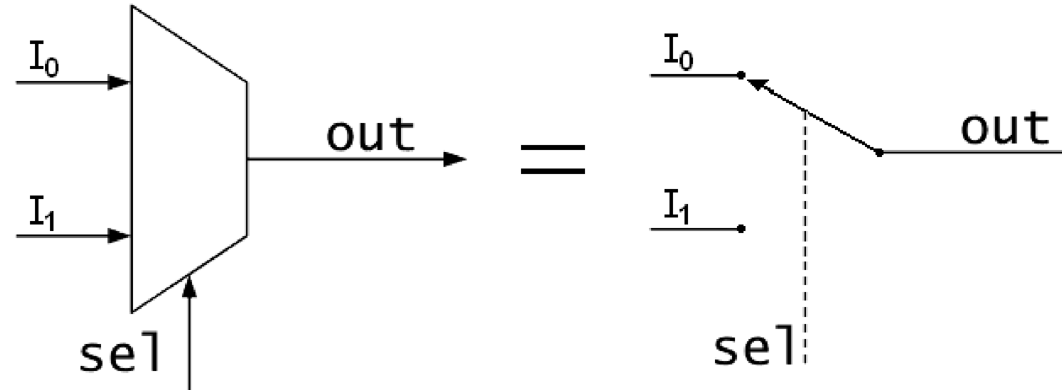


- If at least one input values is 1, the output is 1

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!
- In Logisim you can choose, which interpretation you want to have.

Other Important Gates – Multiplexer (MUX)

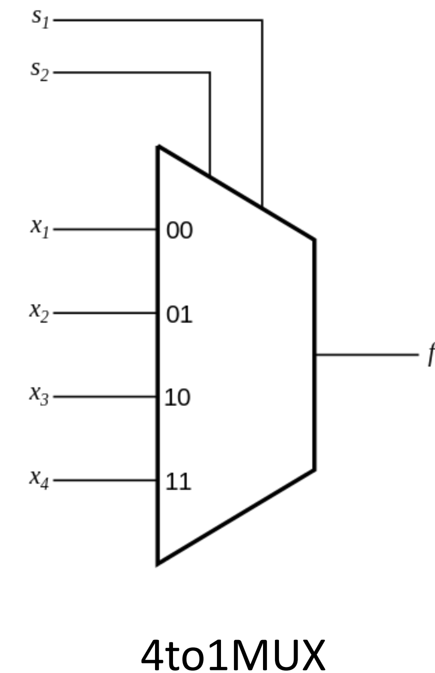


- The select signal (sel) determines whether out is equal to I_0 or I_1:
  - Sel = 0 means out = I_0
  - Sel = 1 means out = I_1

Scaling to more inputs

- With each additional select signal, the number of selectable inputs doubles

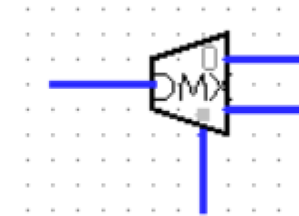
- 2to1MUX: 1 select signal
- 4to1MUX: 2 select signals
- 8to1MUX: 3 select signals
- …



4to1MUX

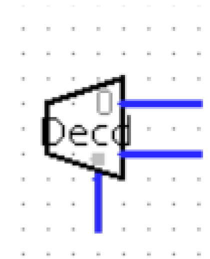
Other Important Gates – Demultiplexer/Decoder

- 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

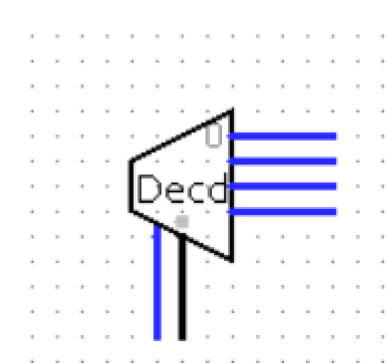


1to2 DEMUX

- 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?

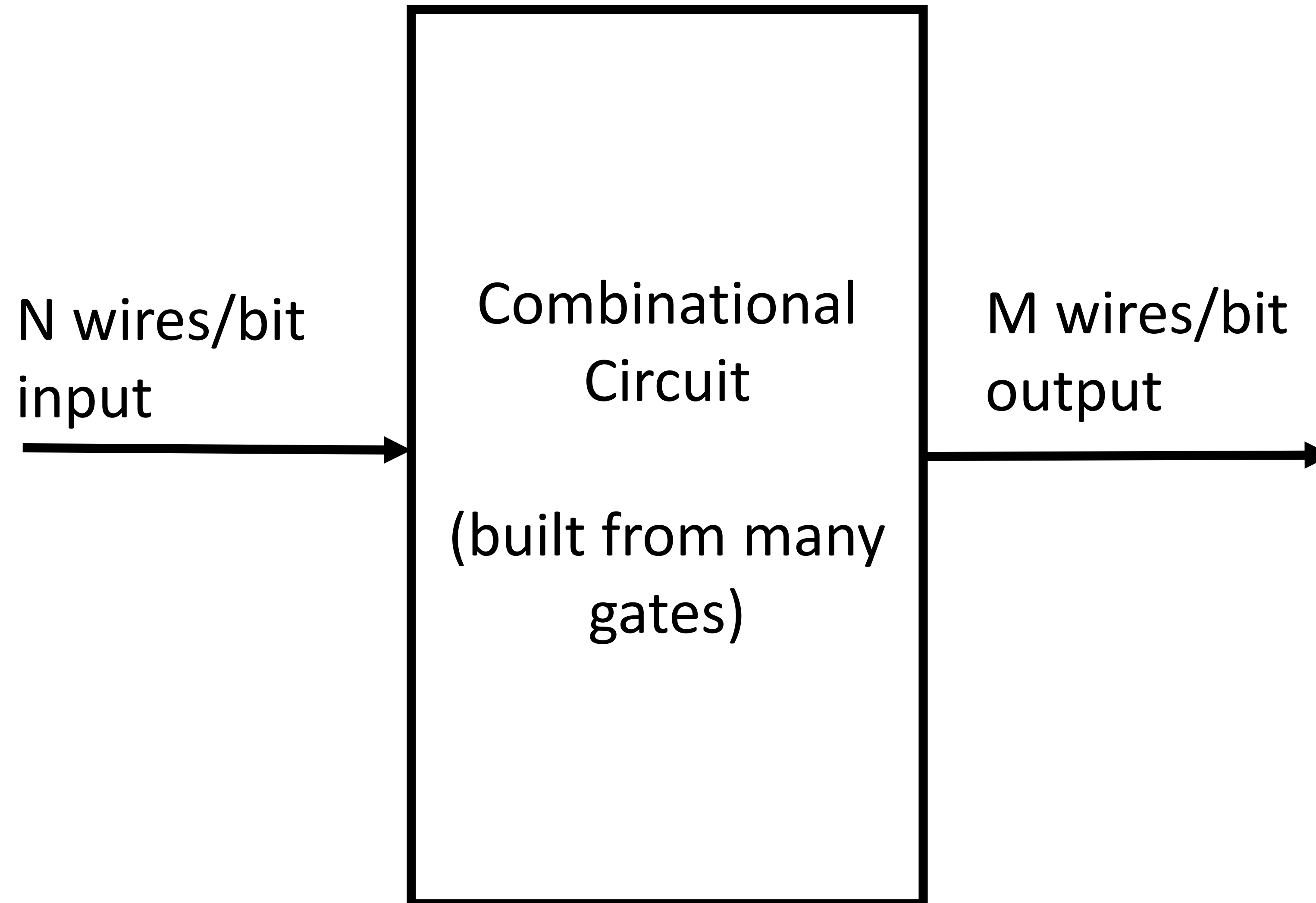
Functional Completeness

- A functionally complete set of logic gates is a set that allows to build all possible truth tables by combining gates of this set.

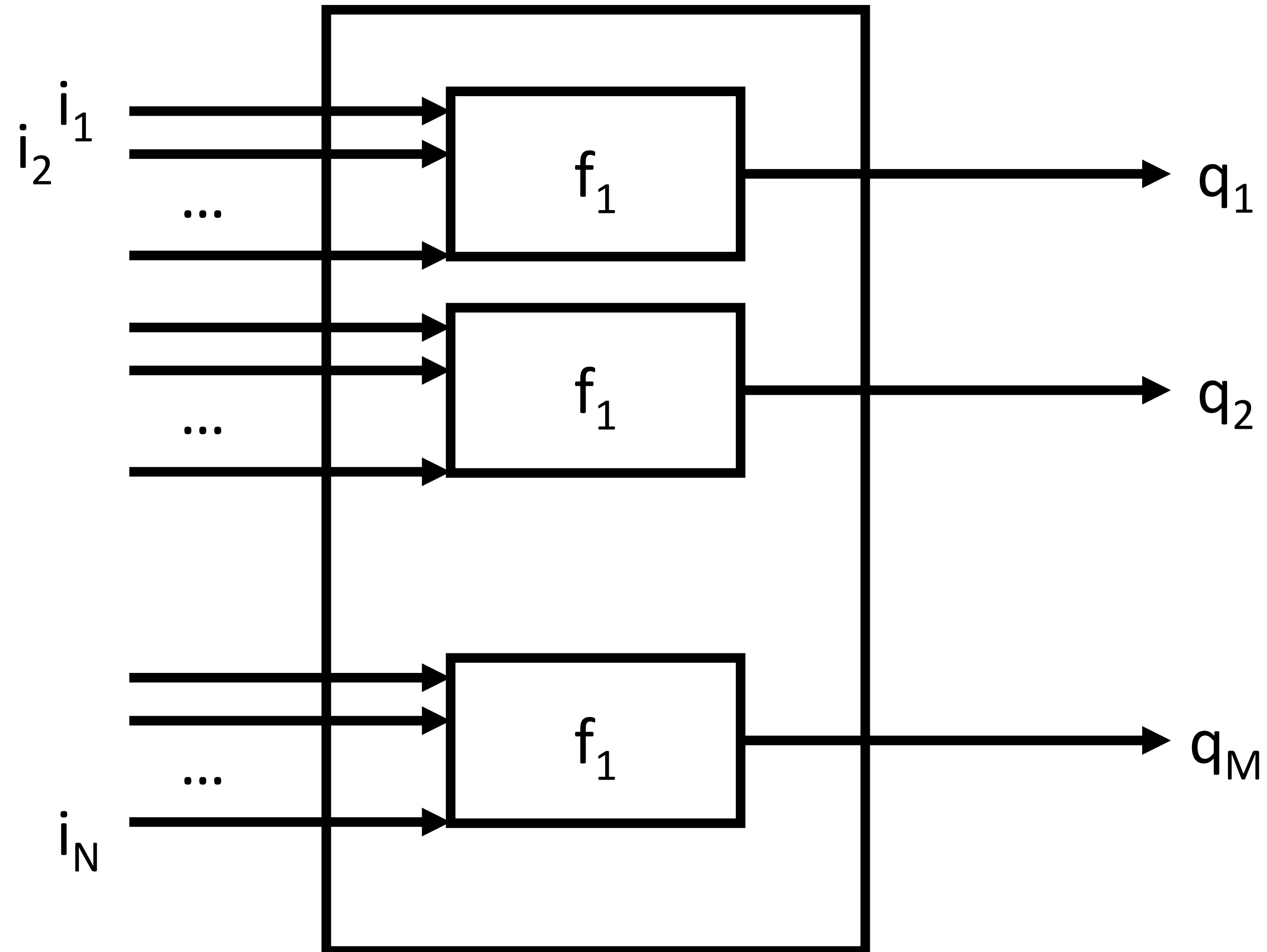
- Important sets are:
  - **{NAND}:** Any circuit can be built just by using NAND gates (try it out in Logisim!)
  - **{NOR}:** Any circuit can be built just by using NAND gates
  - **{AND, NOT}:** Any circuit can be built just by using AND and NOT gates
  - **{AND, OR, NOT}:** The set we use to map truth tables to equations

Describing Combinational Circuits

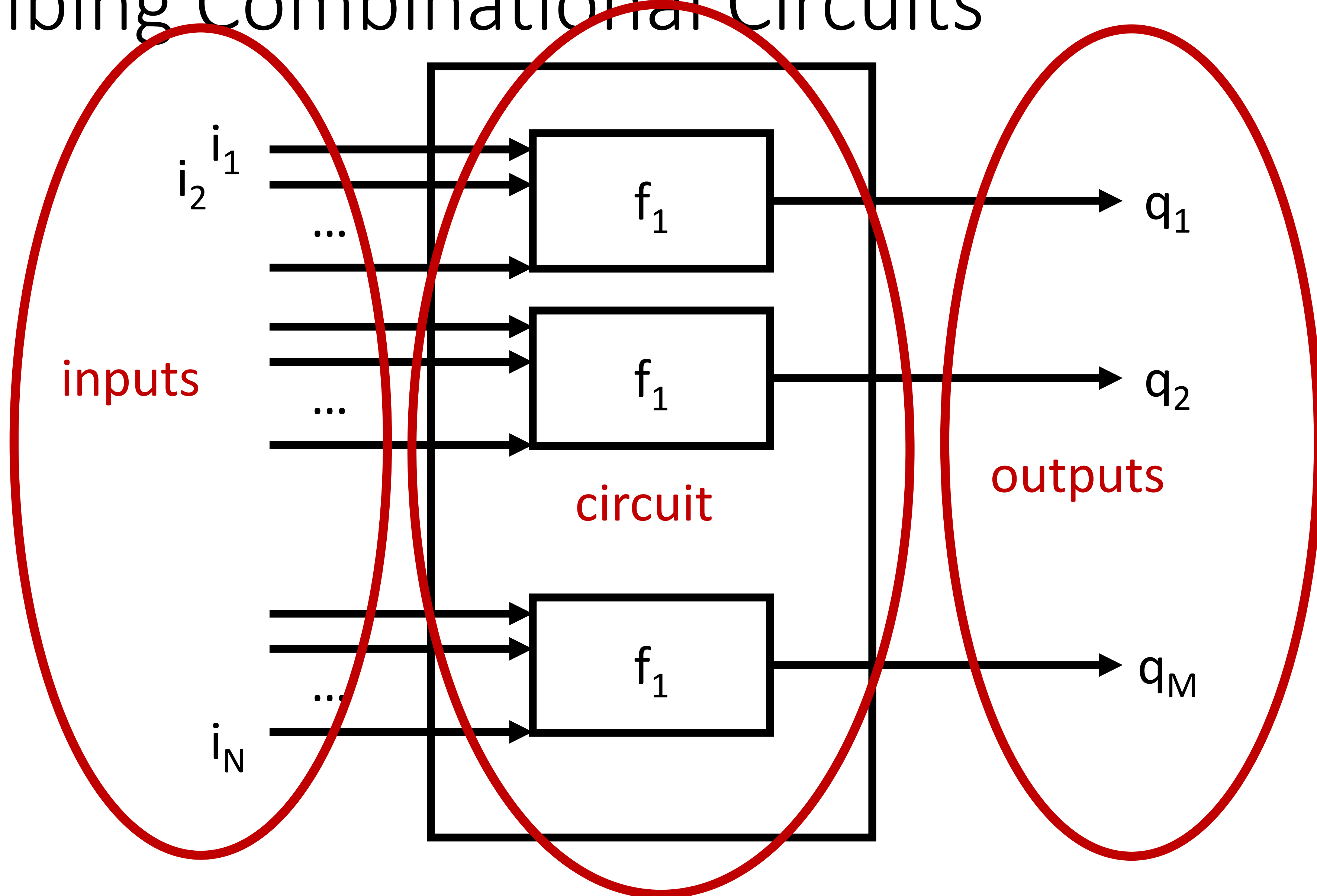
Describing Combinational Circuits



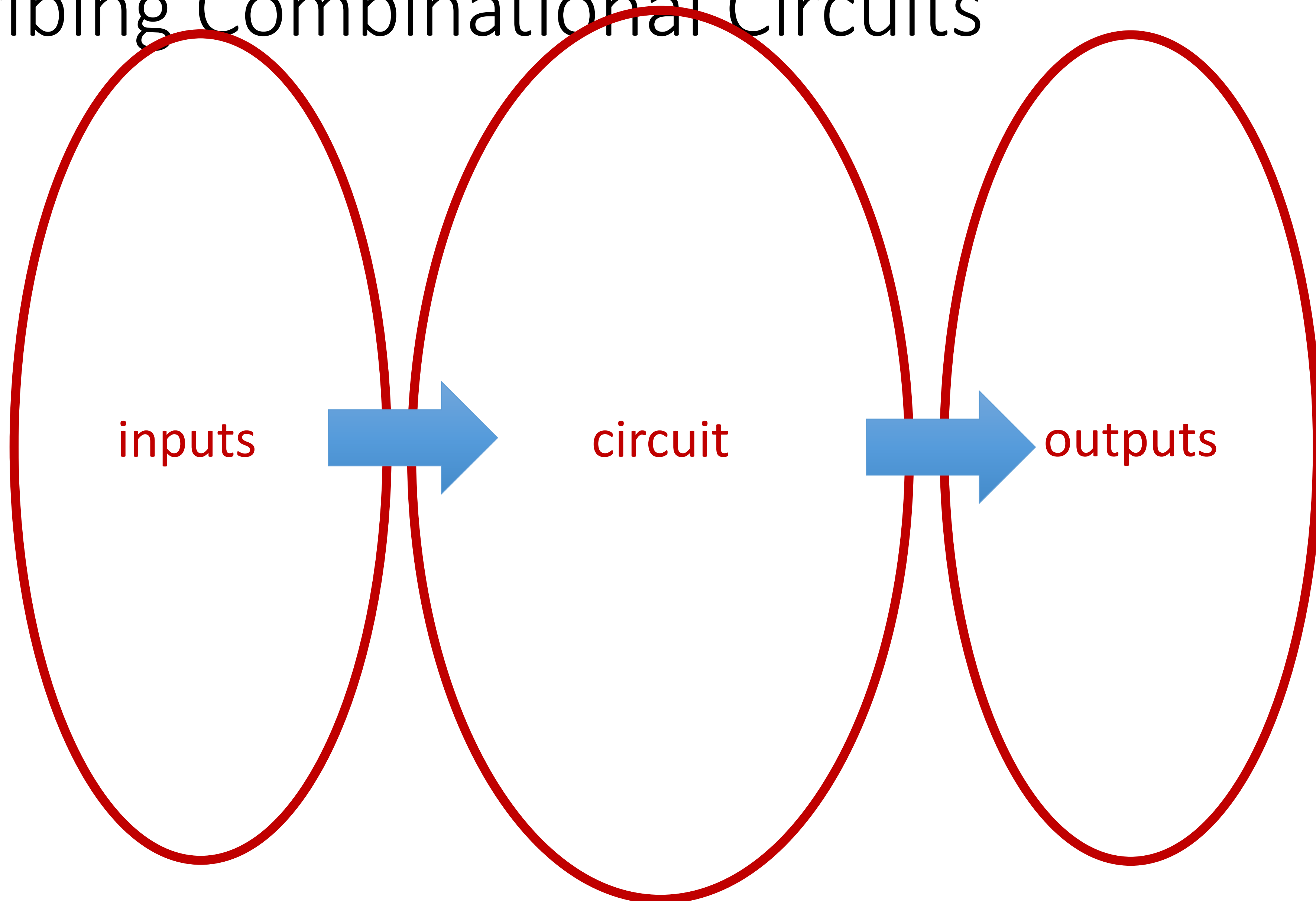
Describing Combinational Circuits



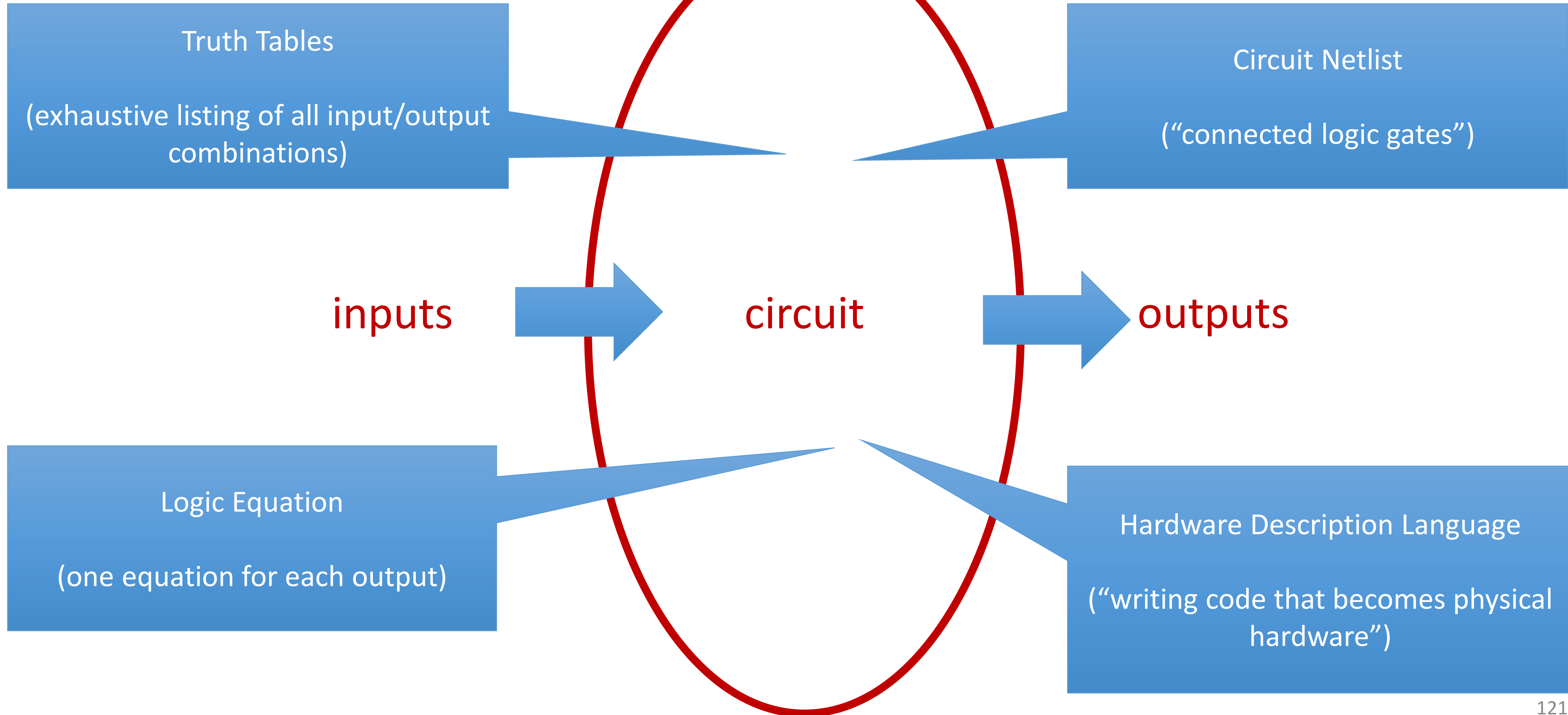
Describing Combinational Circuits



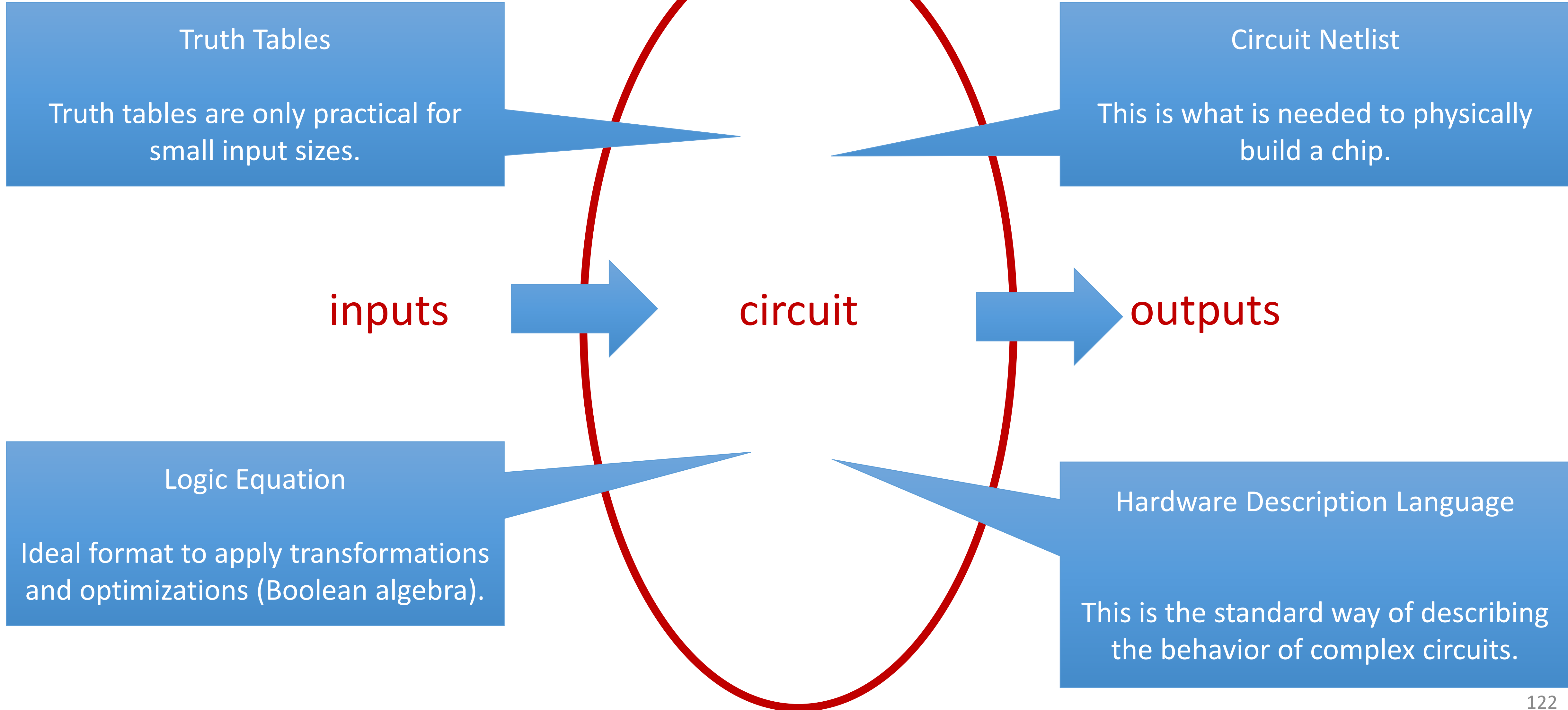
Describing Combinational Circuits



Describing Combinational Circuits



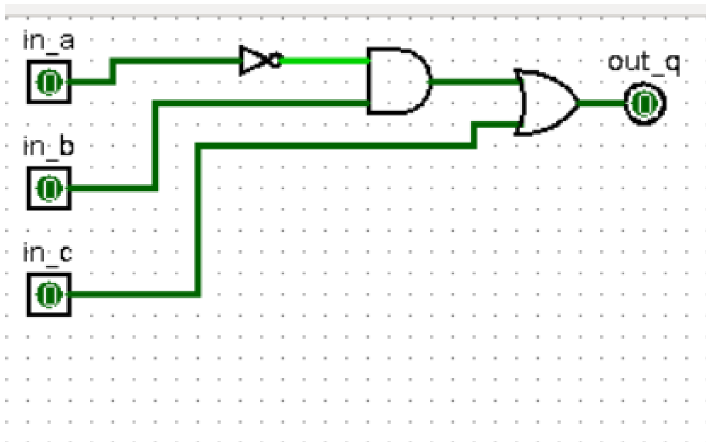
Describing Combinational Circuits



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

out_q = (~in_a & in_b) | in_c




```
module combinational_logic
(
  input logic in_a,
  input logic in_b,
  input logic in_c,
  output logic out_q
);

  // actions that happen if in_a, in_b or in_c changes
  always @(*)
  begin
       out_q = (~in_a & in_b) | in_c;
  end

endmodule
```

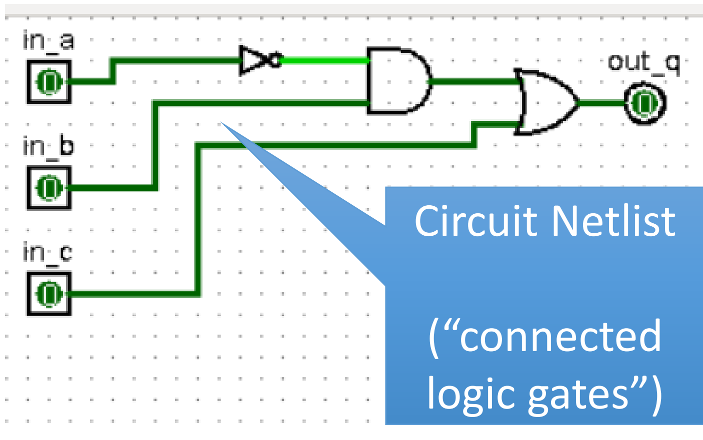
e 1

Truth Table

in_a	in_	n_c	out_q
0	0	0	0
0	0	1	1
0	1	0	1
0	1		
1	0		
1	0		
1	1		
1	1		

Logic equation

out_q = (~in_a & in_b) | in_c



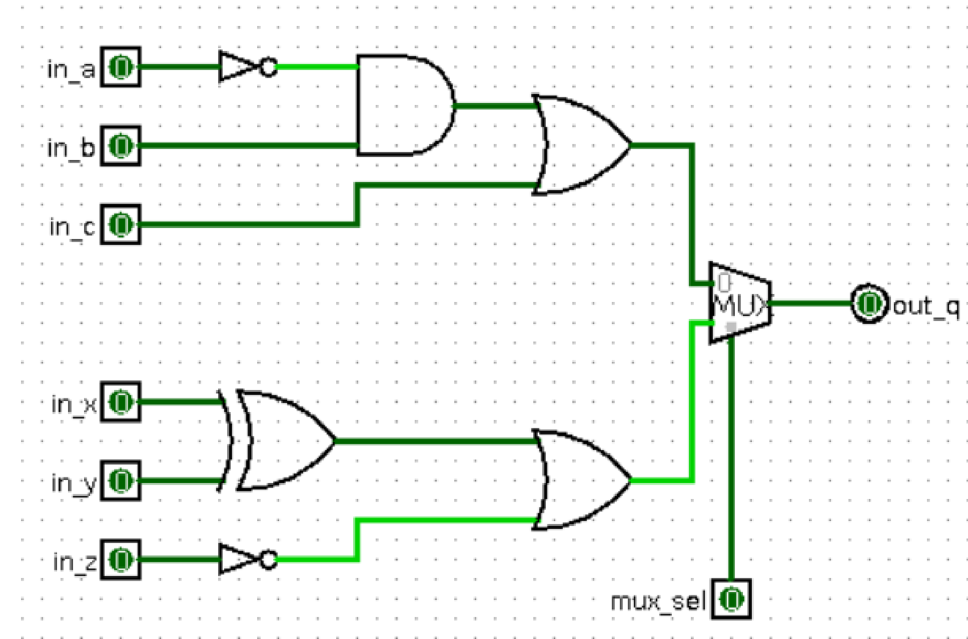

```
module combinational_logic
(
  input logic in_a,
  input logic in_b,
  input logic in_c,
  output logic out_q
);

  // actions that happen if in_a, in_b or in_c changes
  always @(*)
  begin
      out_q = (~in_a & in_b) | in_c;
  end

endmodule
```

Hardware Description Language

SystemVerilog – A Hardware Description Language

```
module combinational_logic
(
  input logic in_a,
  input logic in_b,
  input logic in_c,
  output logic out_q
);

  // actions that happen if in_a, in_b or in_c changes
  always @(*)
  begin
      out_q = (~in_a & in_b) | in_c;
  end

endmodule
```

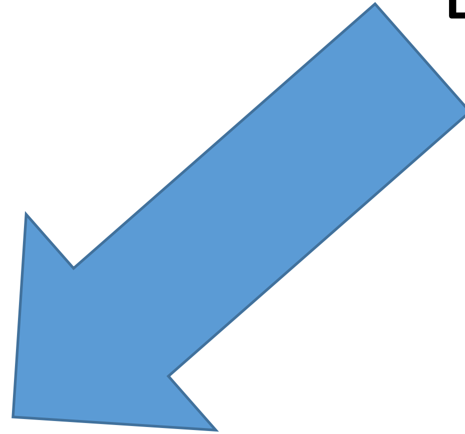
Example 2

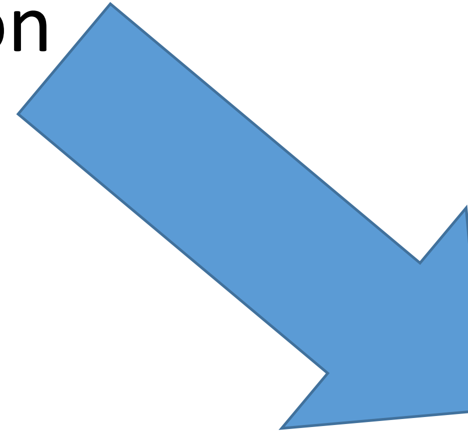



```
module combinational_logic
(
  input logic in_a,
  input logic in_b,
  input logic  in_c,
  input logic  in_x,
  input logic  in_y,
  input logic  in_z,
  input logic  mux_sel,
  output logic  out_q
);

  // actions that happen if in_a, in_b or in_c or in_x, in_y or in_z or mux_sel changes
  always @(*)
  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 designed a circuit - how do I built a physical device that implements this circuit?

My Circuit Description





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
  - 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 (very 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

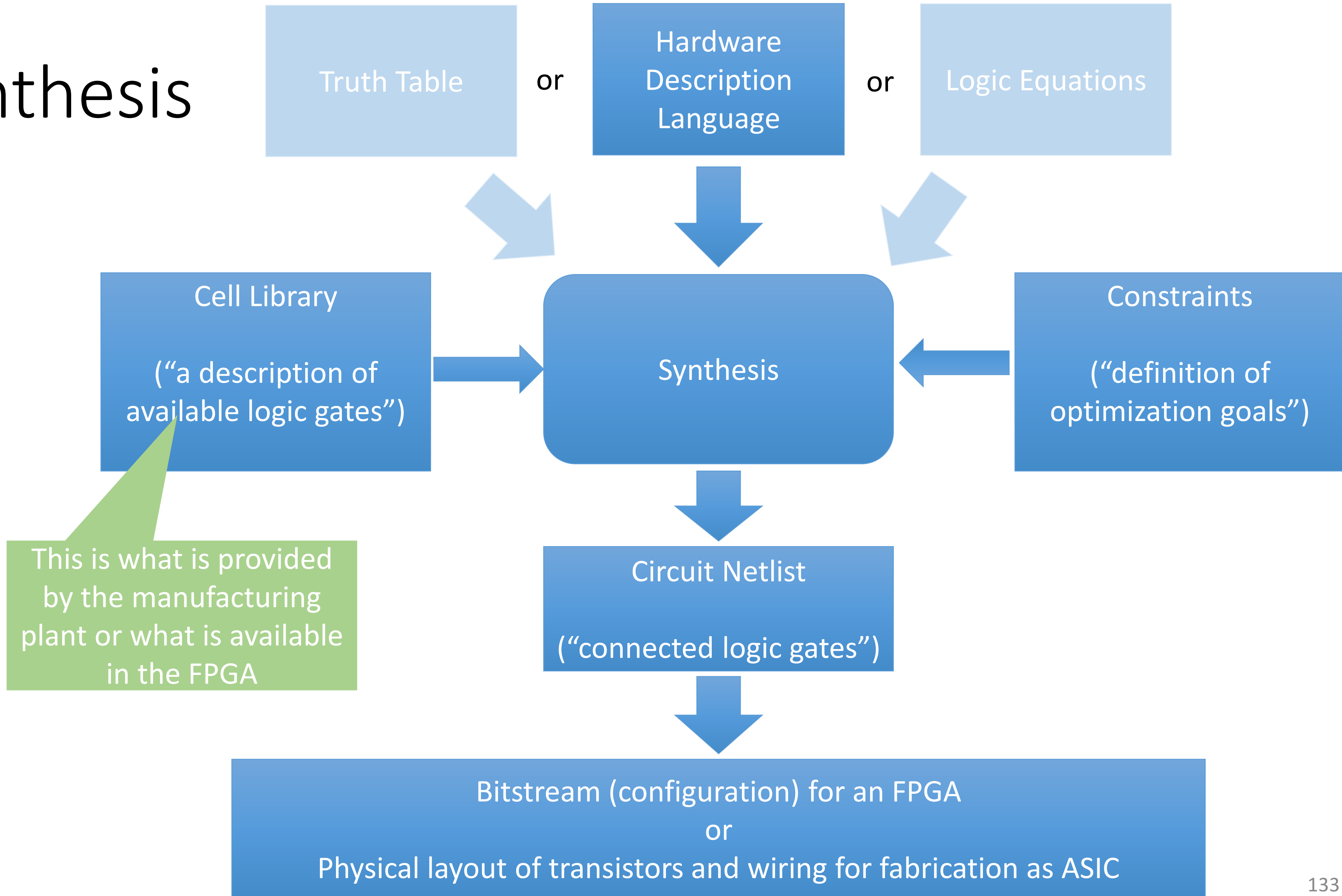
FPGA boards

- 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 FGPAs 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=hJ2CX3hEdVEyBLaHAEXelA%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
- 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

Let’s Try This Out

- See examples con01 available at

  https://extgit.iaik.tugraz.at/con/examples-2020.git

Final remarks

- Every logic function can be uniquely specified by its truth table.
- Every truth table can be implemented by an unlimited number of logic circuits.
- Never talk about optimization without specifying what aspect should be optimized: area, speed, energy consumption, number of gates, security,…?

Appendix

Example 1

Develop the truth table of a multiplier for unsigned binary numbers with 2 digits each. Develop a technical realization of this multiplier by showing its logical equations.

Example 1

Step 1: Unsigned two-bit numbers can have the values 0, 1, 2, and 3.
We first derive the multiplication table

0 * 0 = 0
0 * 1 = 0
0 * 2 = 0
0 * 3 = 0

1 * 0 = 0
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3

2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6

3 * 0 = 0
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9

Example 1

Decimal	Binary
0 * 0 = 0	00 * 00 = 0000
0 * 1 = 0	00 * 01 = 0000
0 * 2 = 0	00 * 10 = 0000
0 * 3 = 0	00 * 11 = 0000
1 * 0 = 0	01 * 00 = 0000
1 * 1 = 1	01 * 01 = 0001
1 * 2 = 2	01 * 10 = 0010
1 * 3 = 3	01 * 11 = 0011
2 * 0 = 0	10 * 00 = 0000
2 * 1 = 2	10 * 01 = 0010
2 * 2 = 4	10 * 10 = 0100
2 * 3 = 6	10 * 11 = 0110
3 * 0 = 0	11 * 00 = 0000
3 * 1 = 3	11 * 01 = 0011
3 * 2 = 6	11 * 10 = 0110
3 * 3 = 9	11 * 11 = 1001

Step 2: We re-write the multiplication table in binary notation.

Note that we have ordered the sequence of mulitplications in such a way that the Binary input pattern is sorted from 0000 to 1111.

Example 1

Step 3: The multiplication table becomes the truth table.

0 * 0 = 0
0 * 1 = 0
0 * 2 = 0
0 * 3 = 0

1 * 0 = 0
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3

2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6

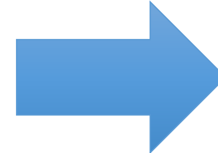
3 * 0 = 0
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9

00 * 00 = 0000
00 * 01 = 0000
00 * 10 = 0000
00 * 11 = 0000

01 * 00 = 0000
01 * 01 = 0001
01 * 10 = 0010
01 * 11 = 0011

10 * 00 = 0000
10 * 01 = 0010
10 * 10 = 0100
10 * 11 = 0110

11 * 00 = 0000
11 * 01 = 0011
11 * 10 = 0110
11 * 11 = 1001



a1 a0 b1 b0	p3 p2 p1 p0
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 0
0 0 1 0	0 0 0 0
0 0 1 1	0 0 0 0
0 1 0 0	0 0 0 0
0 1 0 1	0 0 0 1
0 1 1 0	0 0 1 0
0 1 1 1	0 0 1 1
1 0 0 0	0 0 0 0
1 0 0 1	0 0 1 0
1 0 1 0	0 1 0 0
1 0 1 1	0 1 1 0
1 1 0 0	0 0 0 0
1 1 0 1	0 0 1 1
1 1 1 0	0 1 1 0
1 1 1 1	1 0 0 1

Example 1

Step 4: Derive the logic equation for p3.

p3 = a1 & a0 & b1 & b0

a1	a0	b1	b0	**p3**	p2	p1	p0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	**1**	**1**	**1**	**1**	0	0	1

Example 1

Step 4: Derive the logic equation for p3.

p3 = a1 & a0 & b1 & b0

Step 5: Derive the logic equation for p2.

p2 = (a1 & ~a0 & b1 & ~b0) | (a1 & a0 & b1 & ~b0)

a1	a0	b1	b0	p3	**p2**	p1	p0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	**0**	**1**	**0**	0	**1**	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	**1**	**1**	**0**	0	**1**	1	0
1	1	1	1	1	0	0	1

Example 1

Step 6: Derive the logic equation for p1.

p1 = (~a1 & a0 & b1 & ~b0) |
(~a1 & a0 & b1 & b0) |
(a1 & ~a0 & ~b1 & b0) |
(a1 & ~a0 & b1 & b0) |
(a1 & a0 & ~b1 & b0) |
(a1 & a0 & b1 & ~b0)

= (~a1 & a0 & b1) |
(a1 & ~a0 & b0) |
(a1 & a0 & ~b1 & b0) |
(a1 & a0 & b1 & ~b0)

a1	a0	b1	b0	p3	p2	**p1**	p0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	**1**	**1**	**0**	0	0	**1**	0
0	**1**	**1**	**1**	0	0	**1**	1
1	0	0	0	0	0	0	0
1	**0**	**0**	**1**	0	0	**1**	0
1	0	1	0	0	1	0	0
1	**0**	**1**	**1**	0	1	**1**	0
1	1	0	0	0	0	0	0
1	**1**	**0**	**1**	0	0	**1**	1
1	**1**	**1**	**0**	0	1	**1**	0
1	1	1	1	1	0	0	1

Example 1

Step 6: Derive the logic equation for p1.

```
p1 =    (~a1 & a0  &  b1 & ~b0) |
        (~a1 & a0  &  b1 &  b0) |
        ( a1 & ~a0 & ~b1 &  b0) |
        ( a1 & ~a0 &  b1 &  b0) |
        ( a1 &  a0 & ~b1 &  b0) |
        ( a1 &  a0 &  b1 & ~b0)
   =    (~a1 & a0  &  b1) |
        ( a1 & ~a0 & b0) |
        ( a1 &  a0 & ~b1 &  b0) |
        ( a1 &  a0 &  b1 & ~b0)
```

Step 7: Derive the logic equation for p0.

```
p0 =    ( ~a1 &  a0 & ~b1 &  b0) |
        ( ~a1 &  a0 &  b1 &  b0) |
        (  a1 &  a0 & ~b1 &  b0) |
        (  a1 &  a0 &  b1 &  b0)
```

a1	a0	b1	b0	p3	p2	p1	p0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	**1**	**0**	**1**	0	0	0	**1**
0	1	1	0	0	0	1	0
0	**1**	**1**	**1**	0	0	1	**1**
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	**1**	**0**	**1**	0	0	1	**1**
1	1	1	0	0	1	1	0
1	**1**	**1**	**1**	1	0	0	**1**

Example 2

- Develop the logic equations for a 2-to-1 multiplexor.

Example 2

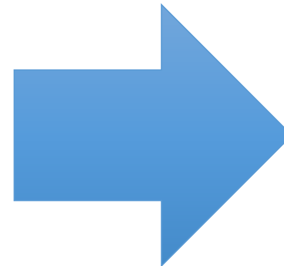
Step 1: Specify the behavior of a 2-to-1 multiplexer.

```
if (sel == 0)
        q = d0;
else if (sel == 1)
        q = d1;
```

Example 2

Step 2: Set up a truth table for the 2-to-1 multiplexer

```
if (sel == 0)
        q = d0;
else if (sel == 1)
        q = d1;
```



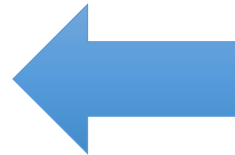
sel	d1	d0	q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Note that the input combinations are ordered from 000 to 111.

Example 2

Step 3: Derive the logic equation from the truth table

```
q =     (~sel & ~d1 &  d0) |
        (~sel &  d1 &  d0) |
        ( sel &  d1 & ~d0) |
        ( sel &  d1 &  d0)
```



sel	d1	d0	q
0	0	0	0
0	**0**	**1**	**1**
0	1	0	0
0	**1**	**1**	**1**
1	0	0	0
1	0	1	0
1	**1**	**0**	**1**
1	**1**	**1**	**1**

…and optionally simplify to …

```
  =  (~sel & d0) |
     ( sel & d1)
```

Example 2

Optional step 4: Draw logic equations as circuit diagram

sel	d1	d0	q
0	0	0	0
0	**0**	**1**	**1**
0	1	0	0
0	**1**	**1**	**1**
1	0	0	0
1	0	1	0
1	**1**	**0**	**1**
1	**1**	**1**	**1**

q = (~sel & ~d1 & d0) |
(~sel & d1 & d0) |
(sel & d1 & ~d0) |
(sel & d1 & d0)

= (~sel & d0) |
(sel & d1)