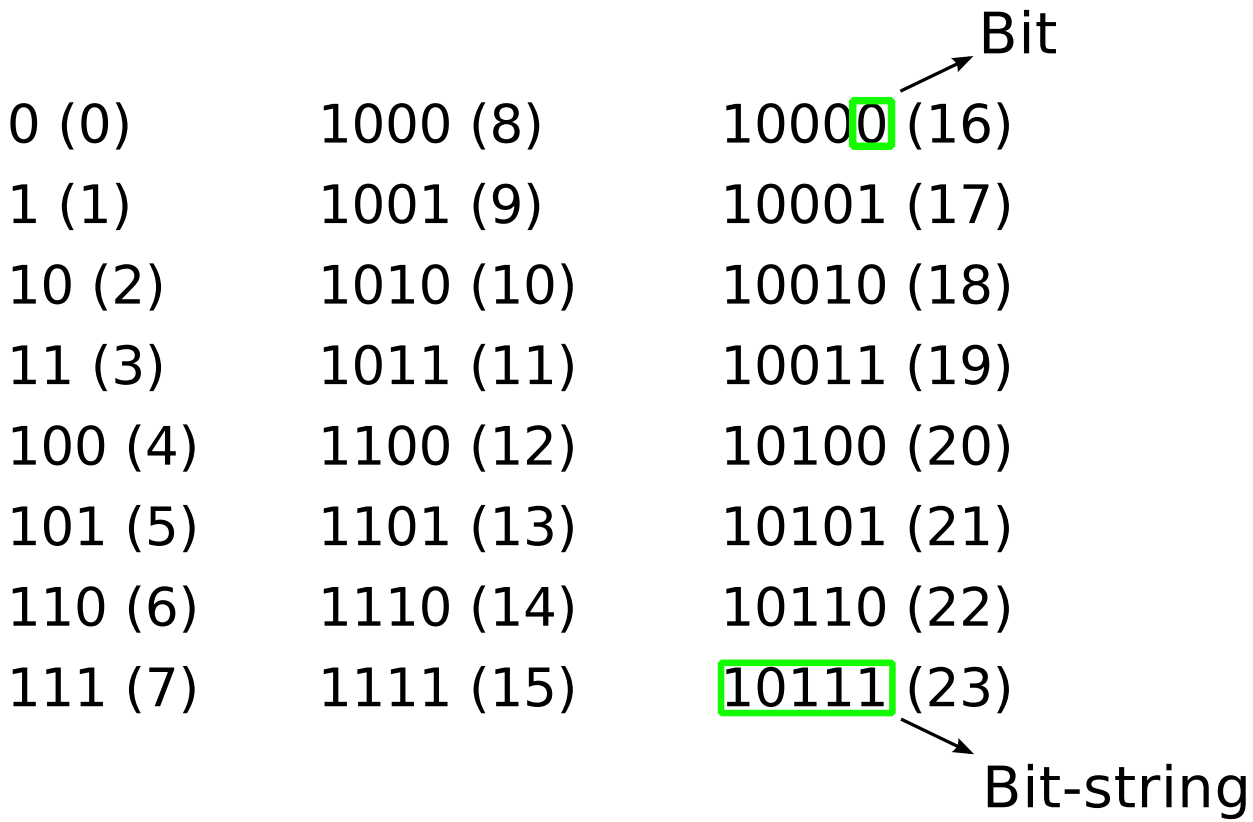
Inf-1100
Innføring i programmering og datamaskiners virkemåte

Åge Kvalnes

University of Tromsø, Norway

August 28, 2014

Counting in binary



The **binary** numeral system uses only two symbols, 1 and 0.

The **decimal** numeral system uses 10 symbols, 0..9.

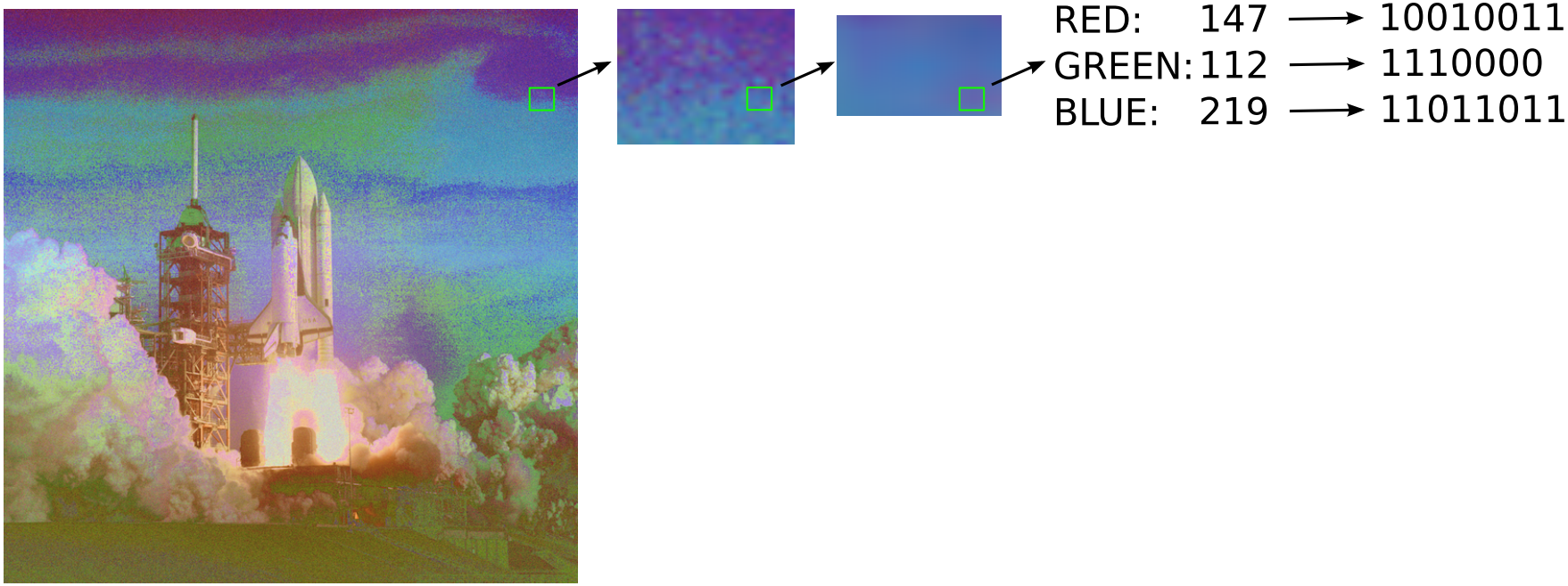
Computers manipulate bit-strings.

Bit-strings can represent anything

By associating a particular interpretation with a bit-string, the string can be made to represent information beyond a number.

- Numbers: Integers, rational numbers, irrational numbers, ...
- Images: Color, size, pixels,...
- Music: Midi, mp3, wav
- ...

Representing a picture as a bit-string



ASCII codes

To represent letters, numbers, punctuation marks, and other special characters in a computer, an intepretation is needed. ASCII is one such interpretation.

The ASCII standard defines a numerical representation for all the characters you find on a (US) computer keyboard. For example,

- The value 65 corresponds to the character 'A'
- The value 33 corresponds to the character '!'

Hexadecimal notation

A hexadecimal number is a number in base 16. The letters $A..F$ are used to denote numbers between 10 and 15.

Hexadecimal notation offers two useful features:

- Hexadecimal numbers are very compact (easier to memorize).
  - $1110111001101011001010000000000_2 = 1000000000_{10} = 3B9ACA00_{16}$
- Easy to convert a number from binary to hexadecimal and vice versa. Group the binary number into 4-bit groups (nibbles). Each nibble corresponds to a hexadecimal digit.
  - $110111101001_2 = 1101\ 1110\ 1001 = DE9_{16}$

Adding binary numbers

$$\begin{array}{r} {\scriptstyle 1\quad 11\quad\quad} \\ 10110 \\ +\ 10011 \\ \hline 101001 \end{array} \qquad \begin{array}{r} {\scriptstyle 1\quad} \\ 22 \\ +\ 19 \\ \hline 41 \end{array}$$

Decimal or binary, the math is the same

Unsigned integers as bit-strings

Weighted positional notation:

$$1101 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$$

Same as in the decimal system: $912 = 9 * 10^2 + 1 * 10^1 + 2 * 10^0$

Problem: How do we represent negative numbers as bit-strings?

Signed integers as bit-strings

Using n bits we have 2^n possible values.

- Use most significant bit (MSB) to hold the sign of the value.
  - Leaves about $2^n/2$ bits to hold actual value.
- Signed magnitude:
  - MSB=0: positive value, e.g. $00110 = 6$.
  - MSB=1: negative value, e.g. $10110 = -6$.

Note: A bit-string becomes a representation of a signed integer by associating an interpretation with the string (e.g. MSB tells sign).

Signed magnitude vs 2's complement

Problems with signed magnitude:

- Two representations of 0 (+0 and -0)
- What's the result of $-5 + 5$ (using normal addition)?
  - Signed magnitude: -10

Most common representation: 2's complement.

- $-5 + 5 = 0$ using normal addition.
- MSB=0:
  - Positive value
- MSB=1:
  - Negative value. To go from positive to negative, first flip every bit, *then add 1*.

2's complement arithmetic: Addition

Straightforward addition works as expected:

$$\begin{array}{rl} & 01111000\ (120) \\ + & 11101100\ (\text{-}20) \\ \hline & 01100100\ (100) \end{array}$$

2's complement arithmetic: Subtraction

Take 2's complement of subtrahend, and then perform a straightforward addition:

$$
\begin{array}{rl}
 & 01111000\ (120) \\
- & 00010100\ (20) \\
\hline
 & 01111000\ (120) \\
+ & 11101100\ (-20) \\
\hline
 & 01100100\ (100)
\end{array}
$$

2's complement arithmetic: Sign extension

When adding two numbers, both numbers must be represented with the same number of bits.

Wrong approach: Pad with zeroes

```
  01111000 (120)
+      1100 (-4)
  ----------
  01111000 (120)
+ 00001100 (12 not -4)
```

Correct approach: Pad with sign bit

```
  01111000 (120)
+      1100 (-4)
  ----------
  01111000 (120)
+ 11111100 (-4)
```

2's complement arithmetic: Overflow

Integers are usually represented in a computer using a fixed number of bits (e.g. 8, 16, 32, 64). What if there aren't enough bits to represent the result of an addition? **overflow!**

$$\begin{array}{rl} & 01111000\ (120) \\ + & 01101110\ (110) \\ \hline & 11100110\ (230,\ \textbf{but -26 using 8-bit 2's complement}) \end{array}$$

Easy to detect: Signs of operands are the same, but sign of result is different.

Note: In many popular programming languages, such as C and Java, overflows are silently ignored! $\Rightarrow$ be conscious of the overflow problem when writing code in these languages.

Logial operations

A logical operation is an operation on logical TRUE or FALSE.

- Represented using one bit (1=TRUE, 0=FALSE).
- An n-bit number is a collection of n logical values.

Operations:

- AND
- OR
- NOT

Logial operations: AND

Result is 1 only if both operands are 1, 0 otherwise:

- 0 AND 0 = 0
- 0 AND 1 = 0
- 1 AND 0 = 0
- 1 AND 1 = 1

$$\begin{array}{rr} & 10100111 \\ \text{AND} & 11111100 \\ \hline & 10100100 \end{array}$$

Logial operations: OR

Result is 1 if any operand is 1, 0 otherwise:

- 0 OR 0 = 0
- 0 OR 1 = 1
- 1 OR 0 = 1
- 1 OR 1 = 1

$$\begin{array}{rr} & 10100111 \\ \text{OR} & 11111100 \\ \hline & 11111111 \end{array}$$

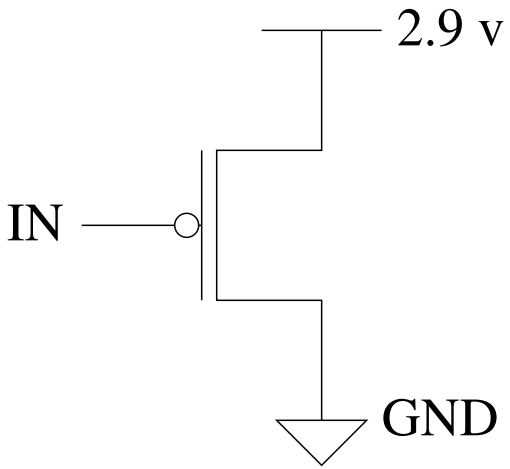
Logial operations: NOT

Flips every bit (complement).

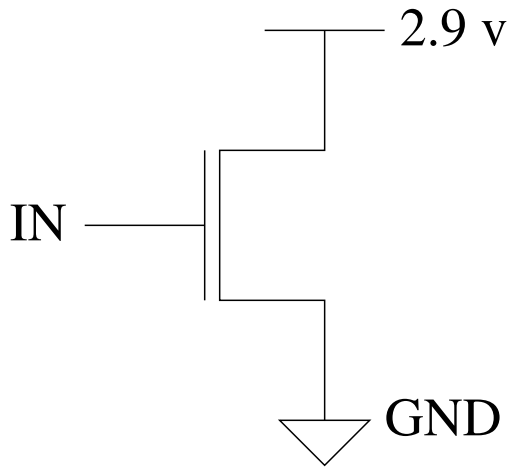
$$\begin{array}{rr} \text{NOT} & 11111100 \\ \hline & 00000011 \end{array}$$

The transistor

A transistor is a semiconductor component that acts like a switch, controlling the flow of an electric current.



(a) **p-type: conducts when IN=0**

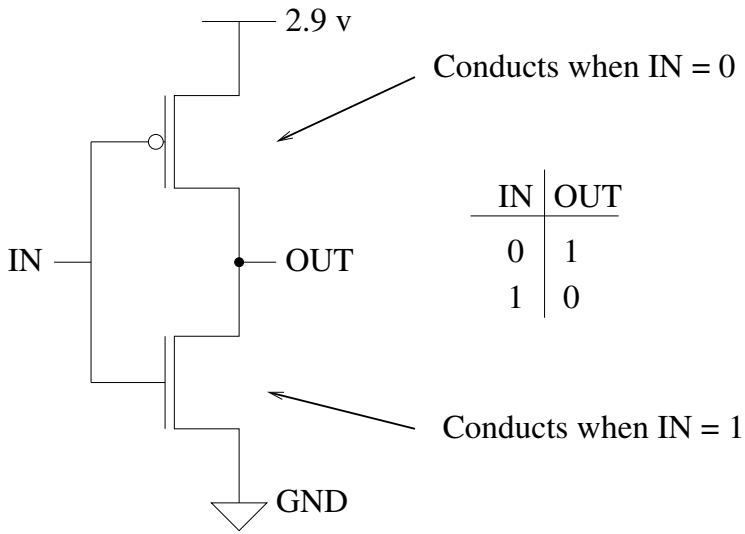


(b) **n-type: conducts when IN=1**

Think of IN as a binary digit. The value 0 is a voltage low enough to keep the p-type transistor open. The value 1 is a voltage high enough to keep the n-type transistor open. And vice versa.

Transistors: Inverting

Combine a p-type and an n-type transistor to invert a binary digit. 1 becomes 0 and 0 becomes 1.

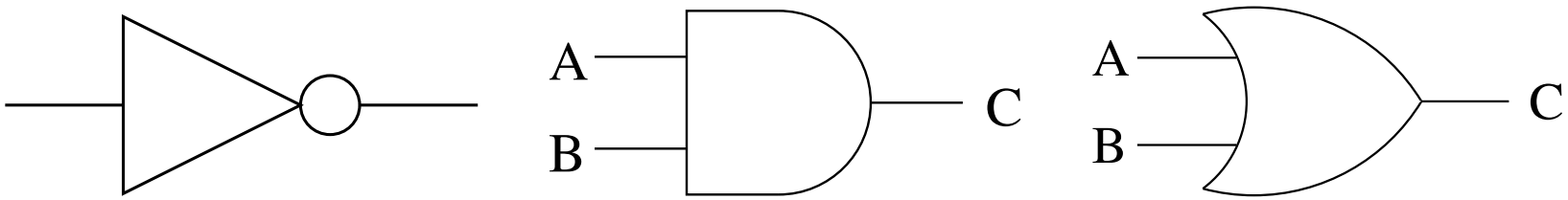


IN	OUT
0	1
1	0

IN represents the binary digit. Depending on the value of IN, OUT is either GND (value 0) or 2.9v (value 1).

Note: The two transistors are performing a logical NOT operation on a bit!

Transistors: Logical operations



(a) **NOT gate** (b) **AND gate** (c) **OR gate**

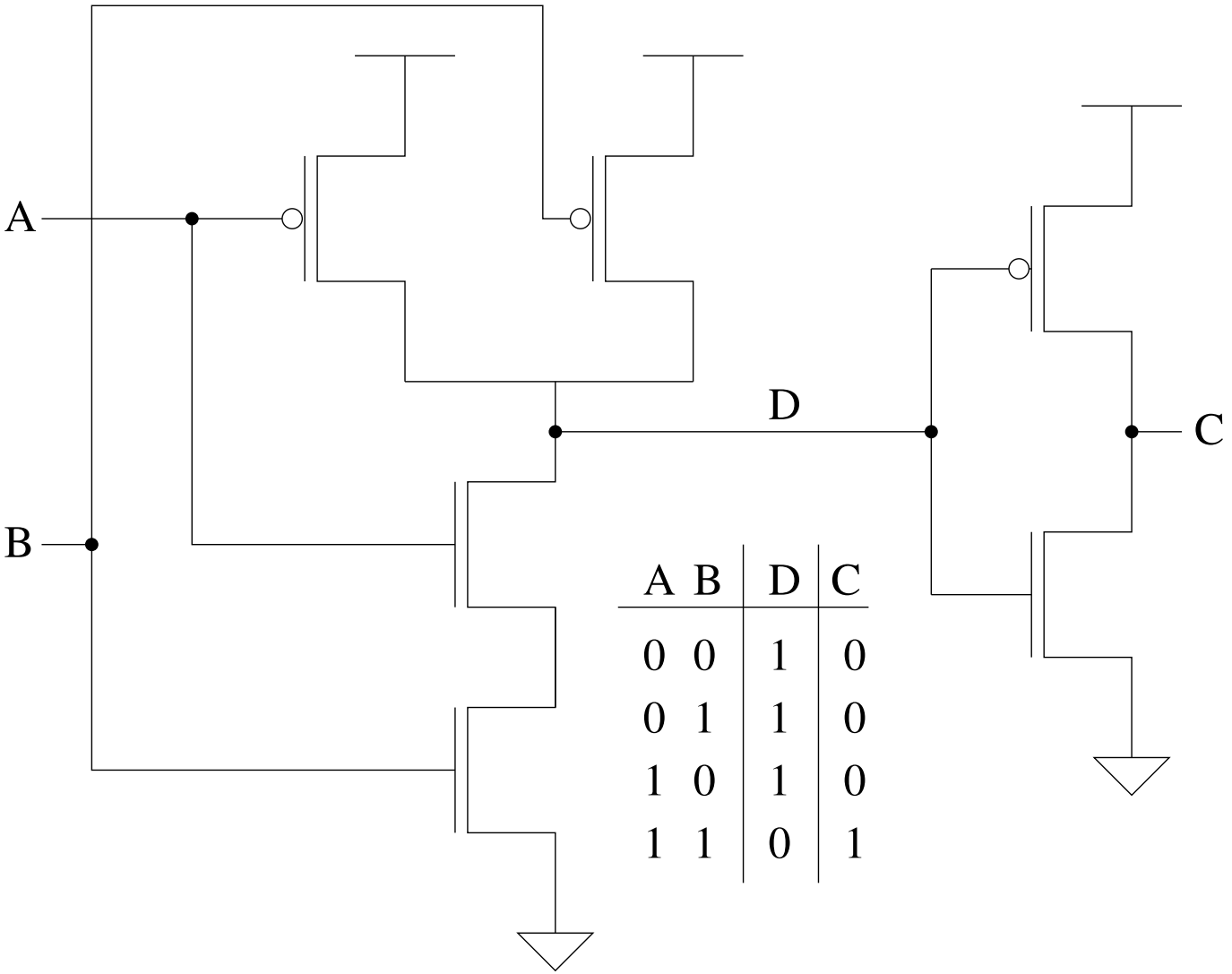
a) Inverts 0 to 1, and 1 to 0.

b) C is 1 only if A **and** B are 1.

c) C is 1 if either A **or** B are 1.

All these **gates** can be implemented by a circuit of transistors

Transistors: The AND gate



A	B	D	C
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Binary addition revisited

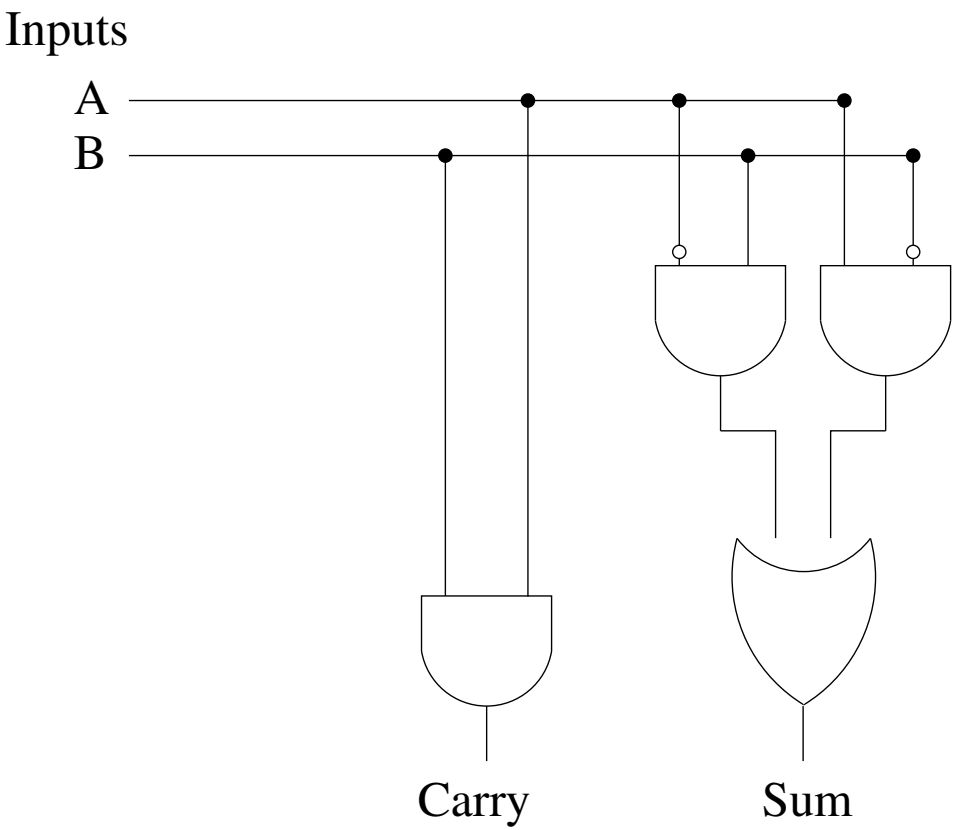
$$\begin{array}{r} \overset{1}{} \overset{1\,1}{} \\ 10110 \\ +\ 10011 \\ \hline 101001 \end{array}$$

Operation on each digit:

$$0 + 0 = 0, \text{ carry} = 0$$
$$0 + 1 = 1, \text{ carry} = 0$$
$$1 + 0 = 1, \text{ carry} = 0$$
$$1 + 1 = 0, \text{ carry} = 1$$

The operation on each digit can be expressed as a series of logical operations...which can be performed by transistors!

Transistors: Adding two binary digits



A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Add the binary digits A and B.