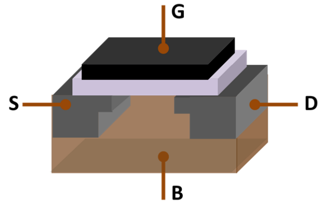
# Introduction to Informatics for Students from all Faculties

**Prof. Dr. Ingo Scholtes**
Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany
ingo.scholtes@uni-wuerzburg.de

**Lecture 02**
**Digital Logics and Data Representation**
October 29, 2024

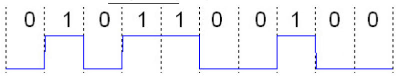
## Motivation

- **how are data and instructions represented in an electronic digital computer?**
- how can we implement logical operations based on digital electronic circuits?
- how can we perform (arithmetic) operations on digitally encoded data?
- how can we **represent instructions** that can be executed by a programmable computer? → L03
- answering these questions requires basic foundations in terms of **numeral systems** and **digital logics**




schematic view of a transistor






a digital signal





**Notes:**

- **Lecture L02: Digital Logics and Data Representation** 29.10.2024
- **Educational objective:** We show how data is represented in a digital computer. We introduce basics of digital logics and show how we can implement arithmetic operations based on logic circuits.
  - Digital Representation of Data
  - Digital Logics and Digital Circuits
  - From Logics to Arithmetics
- **Exercise Sheet 01** due 05.11.2024

**Notes:**

# Encoding numbers with numeral systems

**definition**

A **numeral system** is a system that can be used to consistently encode numbers based on a set of symbols that are called **digits**. In a **positional numeral system** the contribution of a digit to the value of the encoded number depends on the **position of the digit**.

**example: decimal numeral system**

- digits represent different **powers of ten** depending on their position
- we call ten **base** of the decimal number system
- rightmost position represents **power of zero**, i.e. $10^0 = 1$
- powers of ten associated with a position increase from right to left

- value of encoded number is given by the **sum of contributions of individual digits**
- **left-most digit** is called **most-significant digit**
- **right-most digit** is called **least-significant digit**

`MMXXIV`

number 2024 represented in the **Roman numeral system**, where `M` represents thousand, `X` represents ten, `V` represents five and `I` represents one

| $10^3$ | $10^2$ | $10^1$ | $10^0$ |
|---|---|---|---|
| 2 | 0 | 2 | 4 |

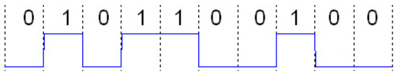
$$2 \cdot 1000 + 2 \cdot 10 + 4 \cdot 1 = 1011$$

number 2024 represented in the **decimal numeral system**, where the value of a digit depends on its position



# Binary numeral system

- in a digital electronic computer we can use voltage levels to encode two symbols 0 and 1
  - low or no voltage = 0
  - high voltage = 1
- voltage levels can be used to encode binary digits or "bits" (0 or 1) of the **binary numeral system**
- positional encoding analogous to decimal system, but using **base two instead of ten**
- depending on their position **digits represent powers of two**, where the least-significant bit represents $2^0 = 1$
- value of an encoded number is given as the **sum of powers of two**




| $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|
| 1 | 0 | 1 | 1 |

$$1 \cdot 8 + 1 \cdot 2 + 1 \cdot 1 = 11$$

image credit: Wikipedia, CC-By-SA



**Notes:**

**Notes:**

# Powers of two

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $2^1$ | 2 | $2^9$ | 512 | $2^{17}$ | 131,072 | $2^{25}$ | 33,554,432 |
| $2^2$ | 4 | $2^{10}$ | 1,024 | $2^{18}$ | 262,144 | $2^{26}$ | 67,108,864 |
| $2^3$ | 8 | $2^{11}$ | 2,048 | $2^{19}$ | 524,288 | $2^{27}$ | 134,217,728 |
| $2^4$ | 16 | $2^{12}$ | 4,096 | $2^{20}$ | 1,048,576 | $2^{28}$ | 268,435,456 |
| $2^5$ | 32 | $2^{13}$ | 8,192 | $2^{21}$ | 2,097,152 | $2^{29}$ | 536,870,912 |
| $2^5$ | 64 | $2^{14}$ | 16,384 | $2^{22}$ | 4,194,304 | $2^{30}$ | 1,073,741,824 |
| $2^7$ | 128 | $2^{15}$ | 32,768 | $2^{23}$ | 8,388,608 | $2^{31}$ | 2,147,483,648 |
| $2^8$ | 256 | $2^{16}$ | 65,536 | $2^{24}$ | 16,777,216 | $2^{32}$ | 4,294,967,296 |

**rules of thumb for orders of magnitude**

$$2^0 = 1 = 10^0$$
$$2^{10} \approx 1,000 = 10^3$$
$$2^{20} \approx 1,000,000 = 10^6$$
$$2^{30} \approx 1,000,000,000 = 10^9$$
$$2^{40} \approx 1,000,000,000,000 = 10^{12}$$
$$\ldots$$



# Group Exercise 02-01

- Convert the the decimal number 126 into the binary number system.

| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

We thus have

$$126 = 64 + 32 + 16 + 8 + 4 + 2$$

- Convert the binary number 10010101 into the decimal number system.

| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

And thus $128 + 16 + 4 + 1 = 149$



**Notes:**

**Notes:**

# Hexadecimal numbers

- long binary numbers are **difficult to read and memorize for humans**
- **hexadecimal numeral system** (base 16) yields human-friendly representation of large (binary) numbers
- to distinguish 16 numbers from 0 to 15 with single digit, we extend the symbols $0, \ldots, 9$ by letters $A, \ldots F$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

- **prefix** $0x$ used to denote hexadecimal number, e.g. 0x10 = 16
- sice $2^4 = 16$, each hexadecimal digit corresponds to **four bits**, i.e. **easy to convert binary/hexadecimal numbers**

0101101011110011

| | $16^3$ | $16^2$ | $16^1$ | $16^0$ |
|---|---|---|---|---|
| 0x | 5 | A | F | 3 |

$$5 \cdot 4096 + 10 \cdot 256 + 15 \cdot 16 + 3 \cdot 1 = 23283$$

| 0x | 5 | A | F | 3 |
|---|---|---|---|---|
| | 0101 | 1010 | 1111 | 0011 |



# Group Exercise 02-01



Convert the following decimal numbers into the **hexadecimal and binary numeral system**.

- 3

  3 = 0x3= 0011

- 4

  4 = 0x4 = 0100

- 19

  19 = 0x13 = 0001 0011

- 64

  64 = 0x40 = 0100 0000

- 255

  255 = 0xFF = 1111 1111



**Notes:**

**Notes:**

# Group Exercise 02-02



Convert the following hexadecimal numbers into the **decimal and the binary numeral system.**

- 0x10

  0x10 = 16 = 0001 0000

- 0xF0

  0xF0 = 240 = 1111 0000

- 0xAA

  0xAA = 170 = 1010 1010

- 0x0100

  0x100 = 256 = 0001 0000 0000

- 0xFFFF

  0xFFFF = 65535 = 1111 1111 1111 1111



# Encoding text

- how can we **encode text in a digital computer**?
- idea: use numbers to encode text characters and represent each number by group of bits

**ASCII encoding**

American Standard Code for Information Interchange (ASCII) defines a 7-bit encoding of 128 different characters.

- code table maps numbers 0 to 127 (represented by 7 bits) to characters and vice-versa

**example**

**Binary ASCII-encoded text "Jurist"**
1001010 1110101 1110010 1101001 1110011 1110100
**Hexadecimal ASCII-encoded text "Jurist"**
0x 4A 75 72 59 73 74

- UNICODE text encoding supports **all writing systems** in the world

*USASCII code chart*

| Bits b7 b6 b5 → | | | | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| b4 | b3 | b2 | b1 | Row ↓ / Column → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 0 | 0 | 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 0 | 0 | 0 | 1 | 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0 | 0 | 1 | 0 | 2 | STX | DC2 | " | 2 | B | R | b | r |
| 0 | 0 | 1 | 1 | 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 0 | 1 | 0 | 0 | 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0 | 1 | 0 | 1 | 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 0 | 1 | 1 | 0 | 6 | ACK | SYN | & | 6 | F | V | f | v |
| 0 | 1 | 1 | 1 | 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 1 | 0 | 0 | 0 | 8 | BS | CAN | ( | 8 | H | X | h | x |
| 1 | 0 | 0 | 1 | 9 | HT | EM | ) | 9 | I | Y | i | y |
| 1 | 0 | 1 | 0 | 10 | LF | SUB | * | : | J | Z | j | z |
| 1 | 0 | 1 | 1 | 11 | VT | ESC | + | ; | K | [ | k | { |
| 1 | 1 | 0 | 0 | 12 | FF | FS | , | < | L | \ | l | \| |
| 1 | 1 | 0 | 1 | 13 | CR | GS | − | = | M | ] | m | } |
| 1 | 1 | 1 | 0 | 14 | SO | RS | . | > | N | ^ | n | ~ |
| 1 | 1 | 1 | 1 | 15 | SI | US | / | ? | O | — | o | DEL |

ASCII code table from printer handbook, 1971

image credit: public domain



**Notes:**

**Notes:**

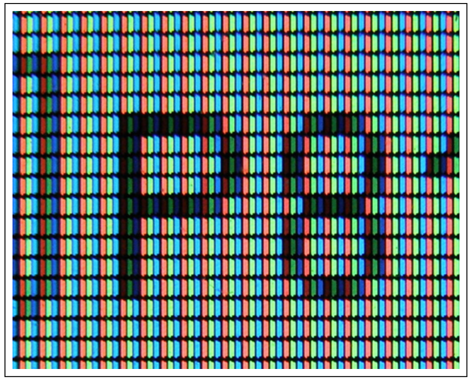
# Encoding images

- how can we represent **image data** in a digital computer?

**digital images**

A digital (raster) image is a picture that is composed of a **rectangular arrangement of pixels**, where each pixel either represents a brightness and (possibly a color value).

- idea: use numbers to represent birghtness (and colors) of pixels
  - **grayscale pixels**: 8 bits encoding 255 brightness levels from black (0) to white (255)
  - **color pixels**: $3 \times 8$ bits encoding 255 brightness levels of red (R), green (G), blue (B)
- image can be digitally encoded by **sequence of bits**, where groups of 8 or 24 bits represent grayscale or color pixels in rectangular grid



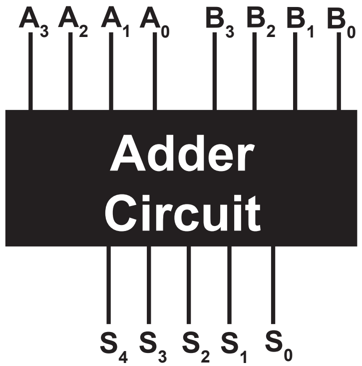
Closeup of pixels (consisting of a red, green, and blue subpixel) on a liquid crystal display (LCD) laptop screen





# From binary numbers to arithmetics

- we introduced the **digital representation of numbers, text, and images** by bits
- how can a digital computer perform **arithmetic operations** like addition, multiplication, etc.?
- as example, consider **addition of two binary numbers with 4 bits each**
- given **input of 8 bits** (i.e. binary numbers $A$ and $B$ with 4 bits each), we must compute **5 bits of output** that encode the sum $A + B$





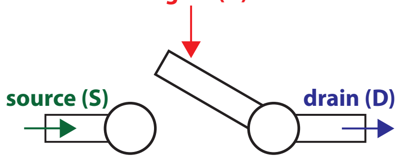

**Notes:**

**Notes:**

- Why do we need five bits for the sum of two four bit binary numbers?
- The largest sum that we can have for two four bit binary numbers is the sum of $1111_b = 15$ and $1111_b = 15$ which is $30 = 1110_b$. This requires 5 bits.

# Digital logics

- **digital logics** is the basis for any (arithmetic) operation of a digital computer
- all functions of a digital computer can eventually be reduced to simple logical operations implemented by electronic switches (e.g. transistors)
- consider one bit representing voltage levels of source (S), gate (G), and drain (D)
- considering S and G as inputs, the output D represents a **logical AND operation**




| S | G | D |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

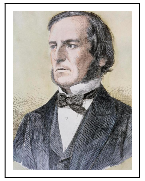


# Boolean Logic

- formal treatment of logics pioneered by mathematician **George Boole**
- Boole **formalized logics** in analogy to arithmetic operations like $+$, $-$, $\times$ → G Boole, 1854
- basic logical operations `AND`, `OR` and `NOT` can be mapped to **arithmetic operations** on numbers $0$ and $1$ representing `False` and `True`

**basic operations in Boolean logics**

- $A$ AND $B \equiv A \times B$
- $A$ OR $B \equiv A + B - A \times B$
- NOT $A \equiv 1 - A$



George Boole (1815 – 1864)

image credit: Wikimedia Commons, public domain



**Notes:**

**Notes:**

# Truth tables

- **truth tables** define output of logical operations
- each row in the truth table is one **possible combination of inputs**
- we use 0 and 1 to represent logical values `False` and `True`

**AND**

| $A$ | $B$ | $A$ AND $B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**OR**

| $A$ | $B$ | $A$ OR $B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**NOT**

| $A$ | NOT $A$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

**XOR**

| $A$ | $B$ | $A$ XOR $B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**basic operations in Boolean logics**

- $A$ AND $B \equiv A \times B$
- $A$ OR $B \equiv A + B - A \times B$
- NOT $A \equiv 1 - A$
- $A$ XOR $B \equiv (A - B)^2$



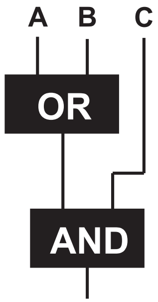
# Boolean functions and digital circuits

- basic Boolean logical operations `NOT`, `AND`, `OR` and `XOR` are defined for one or two inputs, respectively
- we can use these as building blocks of more complex **Boolean functions** with more than one or two inputs
- in formular notation we use brackets to **determine order in which operations are executed**

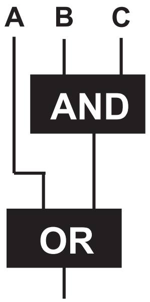
**examples**

- $A$ OR NOT $B$
- [$A$ OR $B$] AND $C$
- $A$ OR [$B$ AND $C$]

- Boolean functions can be represented as **digital circuits** where **logic gates** represent Boolean operations




digital circuit implementing Boolean function [$A$ OR $B$] AND $C$




digital circuit implementing Boolean function $A$ OR [$B$ AND $C$]

image credit:



**Notes:**

**Notes:**

# Group Exercise 02-03



- Give the truth table for the Boolean function $A$ OR [$B$ AND NOT $C$] with three inputs $A$, $B$, $C$.

| A | B | C | A OR [B and NOT C] |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |



# Group Exercise 02-03



- Give a formula for the Boolean function represented by the following truth table.

| A | B | C | ? |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

This can be expressed as [$B$ XOR $C$] OR [$A$ AND $B$ AND $C$]



**Notes:**

**Notes:**

## Group Exercise 02-03



- Show that the **logical operation** `XOR` (with two inputs $A$ and $B$) can be constructed as Boolean function that only uses `AND`, `OR` and `NOT` operations.

| A | B | A OR B | NOT [A AND B] | [A OR B] AND NOT [A AND B] = A XOR B |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |



## Group Exercise 02-03



- Show that the **logical operation** `OR` (with two inputs $A$ and $B$) can be constructed as Boolean function that only uses `AND` and `NOT` operations.

| A | B | NOT A | NOT B | NOT [[NOT A] AND [NOT B]] = A OR B |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |

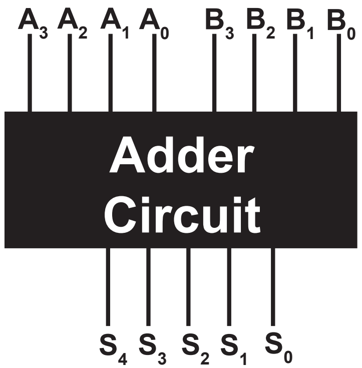


**Notes:**

**Notes:**

# From Digital Circuits to Arithmetics

- we can implement complex Boolean functions by composing basic logical operations `AND`, `OR`, and `NOT`
- assume that we have digital inputs that represent **two numbers $A$ and $B$ in the binary numeral system**
- how can we perform **arithmetic operations** like addition or multiplication?
- idea: specify **Boolean functions** that give correct digital output for each combination of digital inputs






# Adding decimal numbers

- consider **pencil-and-paper algorithm** to **add numbers** in the decimal numeral system

| | | | |
|---|---|---|---|
| 1 | 2 | 5 | 7 |
| $2_1$ | 9 | 3 | 2 |
| 4 | 1 | 8 | 9 |

**pecil-and-paper algorithm to add two numbers**

step 1 start at right-most position
step 2 add digits at current position
step 3 write last digit of sum below current position
step 4 for sums $\geq 10$ additionally **carry over** 1 to position on the left
step 5 move one position to left and go to step 2

- algorithm reduces addition of numbers with **any number of digits** to repeated addition of **single digit numbers**

**open questions**

- how can we apply this to binary numbers ?
- how can we map addition to logical operations?



**Notes:**

**Notes:**

# Adding binary numbers

- how does pencil-and-paper addition work for two **binary numbers** with arbitrary number of digits?
- we can apply the same algorithm but we only have binary digits (bits) 0 and 1

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| $0_1$ | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 |

**pecil-and-paper algorithm to add two numbers**

- step 1 start at right-most position
- step 2 add digits at current position
- step 3 write last digit of sum below current position
- step 4 for sums $\geq 2$ additionally **carry over** 1 to position on the left
- step 5 move one position to left and go to step 2

- **carry bit** of one is created whenever the sum is larger than base two of the binary numeral system



# Half adder circuit

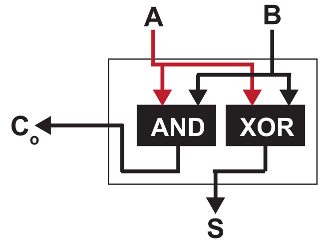
- we can write down the truth table of a Boolean function that generates the **sum** $S$ for two binary digits $A$ and $B$
- sum bit can be generated by a **single** XOR **operation** on $A$ and $B$
- we can further write a Boolean function that generates the **carry-over bit** $C_o$
- carry bit can be generated by a **single** AND **operation** on $A$ and $B$
- we call the resulting digital circuit a **half adder**
- is this enough to add two binary numbers with **any number of digits**?

| $A$ | $B$ | $S$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$S = A \texttt{ XOR } B$

| $A$ | $B$ | $C_o$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$C_o = A \texttt{ AND } B$



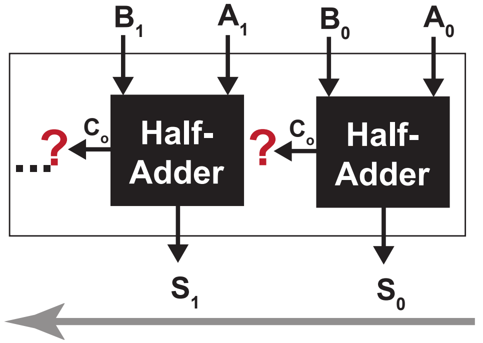



**Notes:**

**Notes:**

## Adding numbers with more than one bit?

**Can we connect multiple half adder circuits to add numbers $A$ and $B$ that consist of more than one bit?**




**problem**

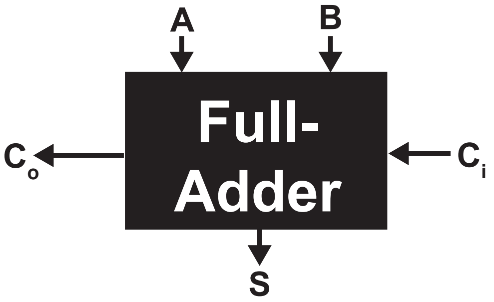
Half adder produces carry-out bit for next position but does not have input that allows to consider carry bit from previous position!



## Full adder circuit

- paper-and-pencil algorithm requires to **include carry bit** from the right when summing digits for any given position
- need Boolean function taking three bits $A$, $B$ and **"carry-in" bit $C_i$ as input** and generating sum $S$ and **"carry-out" bit $C_o$ as output**
- resulting digital circuit is called **full adder**
- **carry-out output** generated for one position is used as **carry-in input** for next position to the left
- this **digital circuit design** implements our simple pen-and-pencil method

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| $0_1$ | $1_0$ | $0_0$ | $1_0$ |
| 1 | 0 | 1 | 1 |






**Notes:**

**Notes:**

# Group Exercise 02-04

- Write down truth tables for the sum and carry-out bit of a full-adder.
- Give formulas for Boolean functions for the sum and the carry-out bit of a full-adder.

| $A$ | $B$ | $C_i$ | $S$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

$S = A \text{ XOR } B \text{ XOR } C_i$

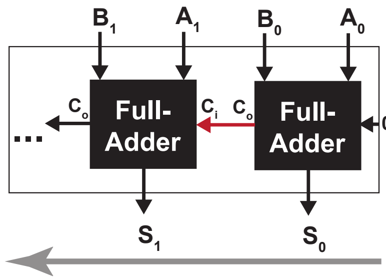
| $A$ | $B$ | $C_i$ | $C_o$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

$C_o = [C_i \text{ AND } [A \text{ XOR } B]] \text{ OR } [A \text{ AND } B]$



# Carry-Ripple Adder

- we can now construct a **full adder** based on logical operations `AND`, `OR` and `XOR`.
- we connect multiple full adders to a digital circuit that adds binary numbers with **any number of digits**




**idea for adding two 4 bit numbers**

- use sequence of 4 full adders, one for each pair of digits of inputs $A$ and $B$
- each full adder generates one bit of sum $S$
- connect carry-out of each full adder with carry-in of next full adder in sequence
- last carry-out is **most-significant bit** of the sum
- we can set **carry-in of first full adder** to zero

- we call this design a **carry-ripple adder**

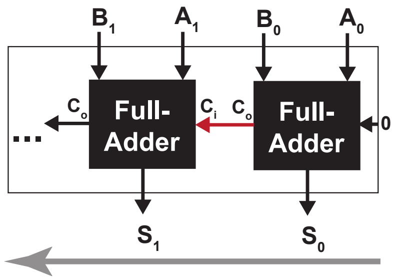


**Notes:**

**Notes:**

## Computation time of Carry-Ripple Adder

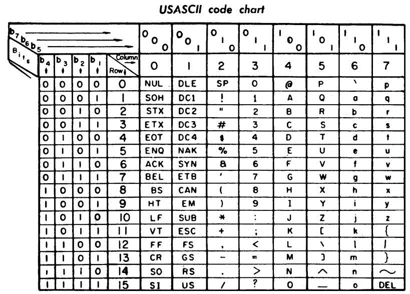
- what is the **computation time** of a carry-ripple adder?
- after setting input bits, we must wait until all carry bits have "rippled" through sequence of full adders
- "'gate delay" until sum is computed corresponds to number of full adders in sequence
- assume that it takes full adder **one nanosecond** ( $= 10^{-9}$ = 1 billionth of a second) to compute sum of two bits
- adding two 32 bit numbers with carry-ripple adder would then take **32 nanoseconds**, i.e. approx. 31 million additions per second
- in practice we use other designs like **carry-lookahead** or **carry-select adder** that speed up computation






## Conclusion

- we have shown how we can **digitally represent data like numbers, text, or images**
- we introduced **basic operations in Boolean logics** like `AND`, `OR`, `NOT`, and `XOR`
- we expressed **Boolean functions** given in a truth table using basic Boolean operations
- we demonstrated how to implement arithmetic operations on binary representations of numbers using **digital circuits**
- digital logics is the **foundation of all digital computers and technology**



USASCII code chart

| $b_4$ | $b_3$ | $b_2$ | $b_1$ | Row \ Column ($b_7 b_6 b_5$) | 0 (000) | 1 (001) | 2 (010) | 3 (011) | 4 (100) | 5 (101) | 6 (110) | 7 (111) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 0 | 0 | 0 | 1 | 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0 | 0 | 1 | 0 | 2 | STX | DC2 | " | 2 | B | R | b | r |
| 0 | 0 | 1 | 1 | 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 0 | 1 | 0 | 0 | 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0 | 1 | 0 | 1 | 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 0 | 1 | 1 | 0 | 6 | ACK | SYN | & | 6 | F | V | f | v |
| 0 | 1 | 1 | 1 | 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 1 | 0 | 0 | 0 | 8 | BS | CAN | ( | 8 | H | X | h | x |
| 1 | 0 | 0 | 1 | 9 | HT | EM | ) | 9 | I | Y | i | y |
| 1 | 0 | 1 | 0 | 10 | LF | SUB | * | : | J | Z | j | z |
| 1 | 0 | 1 | 1 | 11 | VT | ESC | + | ; | K | [ | k | { |
| 1 | 1 | 0 | 0 | 12 | FF | FS | , | < | L | \ | l | \| |
| 1 | 1 | 0 | 1 | 13 | CR | GS | - | = | M | ] | m | } |
| 1 | 1 | 1 | 0 | 14 | SO | RS | . | > | N | ^ | n | ~ |
| 1 | 1 | 1 | 1 | 15 | SI | US | / | ? | O | _ | o | DEL |

ASCII code table

image credit: public domain



**Notes:**

**Notes:**

# Self-study questions

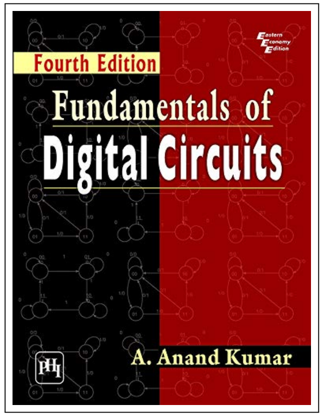
1. What is a bit?
2. Give and example for a positional and a non-positional numeral system to represent numbers.
3. Convert the decimal number 42 into the binary numeral system.
4. Convert the binary number 101010 into the decimal numeral system.
5. Convert the hexadecimal number $0x2A$ into the decimal and the binary numeral system.
6. Explain how you can use electronic switches to implement the basic logical operations AND, OR, and NOT.
7. Use a truth table to explain the difference between the OR and the XOR operation.
8. Give the truth table for the Boolean formula $[A \text{ OR } \text{ NOT } B] \text{ AND } C$.
9. Use a truth table to show that the logical operator X OR Y corresponds to NOT (NOT X AND NOT Y).
10. Give the truth table for the outputs of a half and a full adder.
11. Explain the difference between a half and a full adder.
12. What is the largest output that a carry-select adder for two four-bit inputs and a carry-in input can produce?
13. Draw a diagram of the digital circuit implementation of a full adder.



# Literature

**reading list**


- A Anand Kumar: **Fundamentals of Digital Circuits**, PHI Learning, 2016
- K Fricke: **Digitaltechnik**, Springer Vieweg, 2018
- U Schöningh: **Logik für Informatiker**, Spektrum, 2005
- C Meinel, M Mundhenk: **Mathematische Grundlagen der Informatik**, BG Teubner, 2000




**Notes:**

**Notes:**