Number systems, digital number representations and Boolean logic

How a computer deals with numbers

Questions: **raise your hand** or type in at https://onlinequestions.org

Event number: 17582

Learning goals

- Understand Binary and hexadecimal number systems
- Understand basic encodings including bits, bytes, words, floats.
- Understand simple logical circuits and Boolean logic

Number systems in human culture

- A number system defines how to represent numbers using symbols sequences.
  - It defines the radix (the base number)
  - It defines a writing order (where goes the most and least significant digit)

- Decimal (10) system (most modern cultures)
  - Radix 10, 10 symbols (0…9)
  - Most significant (left) to least (right)
  - Example: 126 → 1*10^2 + 2*10^1 + 6*10^0

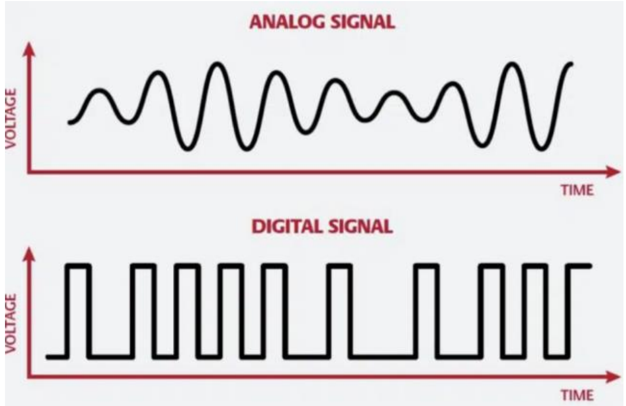
- Dozenal (12) (middle ages Europe, Romans)
  - Radix 12, 12 symbols (0..9, A, B)
  - Most significant (left) to least (right)
  - Example: A6 → 10*12^1 + 6*12^0 (same quantity as 126 decimal)

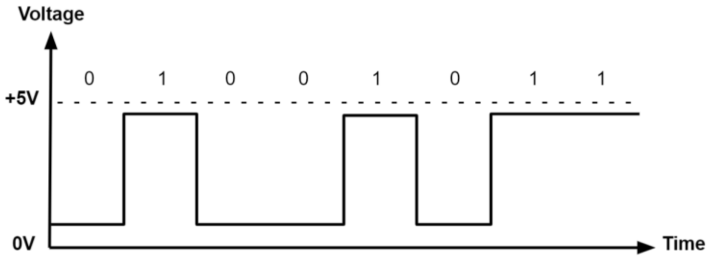
Number systems in computers

- Computers work with **discrete** electric signals.

- Fundamentally, digital hardware works only with
  - 0 (low voltage) and 1 (high voltage)
  - Alternation of the voltage generates a signal, e.g., 01001011
  - Storage of signals also uses high and low voltages
  - See the course essentials for (much more) detail.

- What is important is that your computer works with these signal:
  - Using the **binary number system,** and
  - **binary logic**





Binary number system and number representations

The binary number system

- Radix 2, 2 symbols (0,1)
- Most significant (left) to least significant (right).
- Example: 1101 → 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 (=13 decimal)

- Calculations in the binary system
- Addition: 101 + 101 = 1010
- Subtraction: 1101 – 10 = 1011
- Multiplication: 11 * 110 = (110 + 1100) = 10010

Bits and bytes

- A single binary digit is called a **bit**
  - 0 or 1
- When you store 8 bits, we call this a **byte**
  - 11111111 (255 in decimal)
- And then we use the standard Greek extensions for sizes
  - Kilo (1000), Mega (1000000), Giga (1000000000)
  - Except that in Computer Science we use the nearest powers of 2
    - Kilo, 2^10 : 1024
    - Mega 2^20: 1048576
    - Giga 2^30: 1024* 1048576 = 1073741824
- *How many **bytes** is your computer's RAM? And bits?*

The Hexadecimal number system

- Problems with binary and decimal number systems
  - the decimal system and the binary system have a rather incompatible Radix (10 is not a power of 2) hence they do not convert well.
  - writing down binary numbers is cumbersome as they take up lots of space.
  - Humans are not good at "reading the matrix" (watch the movie!)

- So the Hexadecimal number system is often used:
  - Radix 16 (0…9, A…F)
  - Most significant (left) to least significant (right).
  - Example FF → 15*16^1 + 15*16^0 = 255 (decimal) or 1111 1111 (binary)

- Binary data is easy to represent in hex form:
  - 1101 0111 → D7
  - 1111 0010 → F2
  - 1101 0111 1111 0010 → D7 F2

- *Why??*

Number representations in computing

- Careful: number representation is not the same as number system
  - all number representations in computers use the binary system!
- Number representations are about binary representations of common mathematical number types
  - Natural numbers, $\mathbb{N}$
  - Integers, $\mathbb{Z}$
  - Real numbers, $\mathbb{R}$
- Computers use standards for representing these numbers
  - Unsigned integer = binary representation → $\mathbb{N}$
  - Signed integer = signed binary with two's complement → $\mathbb{Z}$
  - Floating point = exponent plus mantissa → $\mathbb{R}$

Unsigned and signed integers

- Important: assume 8 bits for the number representation

- Unsigned integers (positive numbers including 0)
  - Simple binary representations
  - Most to least significant bits 00011011 → 27

- Signed integers (sign bit, then two's complement)
  - 1 sign bit (leftmost), then number bits (most to least)
  - Positive numbers, just as unsigned integers:
    - **0** 0011011 → 27
  - Negative numbers use first bit 1 then two's complement
    - **1** 1100101 →-27
    - Subtract 1 from absolute number, then invert (XOR)
    - Why: so that addition and subtraction is still properly defined:
    - 00011011 + 11100101 = 00000000 = 0 = 27 – 27
- *What is 10000000 in two's complement signed integer in decimal value?*
- *How do you represent -15?*

n1	n2	n1+n2
00000011 (3)	11111101 (-3)	00000000 (0)
00001000 (8)	11110111 (-9)	11111111 (-1)
00010100 (20)	11111011 (-5)	00001111 (15)

Words and Longs

- Some languages / computers distinguish between 8, 16 and 32 bit integer sizes
- An **8-bit integer (**1 byte) stores values from:
  - Unsigned: 0 to 255
  - Signed: −128 to +127 (using two's complement)
- A **word** is typically a 16-bit (2 bytes) integer:
  - Unsigned: 0 to 65,535
  - Signed: −32,768 to +32,767
- A **long** is often a 32-bit (4 bytes) integer:
  - Unsigned: 0 to 4,294,967,295
  - Signed: −2,147,483,648 to +2,147,483,647

Floating points

- Again assume 8 bits for the representation

- The problem with integers: loads of bits to represent large numbers.
  - E.g. the number 1230000000 would need at least 30 bits (10^9 in binary notations would need log2(10^9) digits at minimum)
  - But, if an approximation is ok, the precision we need is only 3 decimal digits, which would fit in 1 byte!
  - So, sometimes better to represent the precision and the exponent! **Floating points to the rescue!**

- Floating point 8 bit (minifloat standard, not official!).
  - 1 sign bit (1 is negative)
  - 3 exponent bits (-3 to +3, 000 - 110)
  - 4 mantissa (precision) bits
  - (-1)^Sign * 1.mantissa * 2^(exponent - bias)

- Trade-off between bit-for-bit accuracy in favor of compactness and dynamic range

Floating point examples

- Formula: (-1)^Sign * 1.mantissa * 2^(exponent - bias) = decimal
- 0 100 1000 → ?:
  - Sign = positive
  - Exponent = 4 (-3 for bias range) = 1
  - Mantissa = **1**.1000 (pay attention to the leading binary 1 used to represent more precision) = 1.5 (decimal)
  - (-1)^0 * 1.5 * 2^(1) = 1 * 1.5 * 2 = **3**

- -0.45 → ?
  - Sign = negative → sign bit = 1
  - Write the number in normalized binary form.
    - Mantissa (repeat times 2 to retrieve the powers of 2 in the fraction) 0.45 = 0.011100…
    - Normalized 1.1100 * 2^-2 (if you shift to the left 2 times, you end up at 1.XXXX)
  - Exp of normalized form = -2 so -2+3=1=001
  - 1 001 1100, which is -0.4375 and closest possible to original

Boolean (or binary) logic

Boolean logic

- Very often when you program you need to check for things:
  - If *the user presses a Buy button* **and** *there is an item in the basket* then order the item
- This assumes you understand **Boolean logic**
  - This will be covered in much more detail in the courses Essentials and Logic 1, but you need to know the basics now.
- Boolean logic is a set of rules about operations you can do on symbols that are either True or False
- Computers represent False as 0 and True as 1.
- Important mental note for later: programming languages represent True and False more flexibly
  - True = 1 bit, False = 0 bit, or,
  - Any number not 0 = True, otherwise False
  - A variable that is not defined is False, one that is defined = True
  - Etc…

Boolean logic example

- If *the-user-presses-a-Buy-button* **and** *there-is-an-item-in-the-basket* then order the item
- If *buyButtonPressed()* **and** *itemInBasket()* then *orderBasket()*
- If BP and IB then OB
- (BP & IB) → OB
- This says, if BP is True and IB is True then OB must be true as well

Boolean logic operators

Operator	Symbol	Meaning	Remark
AND	$A \land B$	A and B must both be True	In programming we use &
OR	$A \lor B$	A or B or both are True	In programming we use |
NOT	$\neg A$	A is False (inverts truth value)	In programming we use !
XOR	$A \oplus B$	A or B is True but not both	In programming we use ^ but this can also be power so be careful
NAND (NOT AND)	$A \barwedge B$	NOT (A AND B)	Used in hardware (generic)
NOR (NOT OR)	$A \veebar B$	NOT (A OR B)	Used in hardware (generic)
IF (implication)	$A \rightarrow B$	if A is true then B must be true, otherwise the operator decides True (don't care if A is false)	Used in Logic and in programming we use *then* to define what to do when the if is True
IFF (if and only if)	$A \leftrightarrow B$	B is True only if A is True	Mostly used in logic, not so much in programming

Logic systems and Truth tables

- If *buyButtonPressed()* **and** *itemInBasket()* then *orderBasket()*
- A way to "proof" if a set of **logical sentences** is ever True is by *enumeration*
  - List all combinations of True/False values for all symbols
  - Add the rules as columns to the table
  - Check where all rules are True
- See table right, proofing that indeed
  - We only order when the rule is true

BP	IB	OB	BP ∧ IB	(BP ∧ IB) ↔ OB
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	**1**

Questions and exercise.

- See Brightspace (or for those without access use the following QR)