How Computers Represent and Manipulate Data

What We Will Cover

- Why computers use **binary**
- How **numbers** are represented
  - Integers
  - Signed integers
  - Decimals
- Why decimal arithmetic is approximate
- How humans read binary using **hexadecimal**
- How **characters** are represented
  - ASCII, Unicode and UTF-8

Section 1: Binary Basics

In this section, we will learn:

- Why computers use only **0 and 1**
- What **binary numbers** are
- What **bits** and **bytes** mean
- How simple binary arithmetic works

How Computers Represent Data

- Computers work with **data**
- Data means:
  - Numbers
  - Characters
- But inside a computer, everything becomes **0 and 1**

Why Only 0 and 1?

- Computers are electronic machines
- Electronic circuits are:
  - ON
  - OFF
- ON = 1
- OFF = 0

This system is called **Binary**

Binary Number System

- Decimal system → base 10
  - Uses 0-9
- Binary system → base 2
  - Uses only 0 and 1

Binary Place Values

Position	Value
1st	1
2nd	2
3rd	4
4th	8

Example:

```
1011 = 8 + 2 + 1 = 11
```

- Rightmost bit is the smallest value

Try This

Convert these to decimal:

- `0011`
- `0101`
- `1001`

Write the place values before adding.

Binary Addition

Rules:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10 (carry)

Binary Addition Example

```
  101
+ 011
--
1000
```

Try This (Binary Addition)

Add these binary numbers:

```
  0101
+ 0011
```

Write each step.

Mark where carry happens.

Binary Subtraction

Rules:

- 0 − 0 = 0
- 1 − 0 = 1
- 1 − 1 = 0
- 0 − 1 → borrow

Binary Subtraction Example

```
  1010
- 0011
--
  0111
```

What Is a Bit?

- **Bit** = Binary Digit
- Smallest unit of data
- Value can be:
  - 0
  - 1

What Is a Byte?

- **1 Byte = 8 bits**
- Basic unit of storage
- Memory and files use bytes

Example:

```
01000001
```

Why 8 Bits in a Byte?

- Standardized early
- $2^8 = 256$ values
- Enough for characters and small numbers

Real-Life Analogy: Boxes

- Bit → one switch
- Byte → box with 8 switches
- More data → more boxes

Summary

- Computers represent all data using **binary**
- A **bit** stores 0 or 1
- A **byte** is 8 bits
- Binary arithmetic is simple and reliable

Section 2: Representing Numbers

In this section, we will learn:

- How computers store **whole numbers**
- How **negative numbers** work
- What **overflow** means
- Why **decimals are approximate**

Types of data used in computing

- Numbers
  - Integers
  - Signed (Negative or Positive) Integers
  - Decimals
- Characters

How are they represented?

Integers (Whole Numbers)

- Integers are **whole numbers**
- No decimal point
- Examples:
  - 0, 1, 2, 10, 255

Computers store integers using **binary bits**

Unsigned Integers

- Can represent **only 0 and positive numbers**
- No negative values

With **n bits**, range is:

- 0 to $(2^n - 1)$

Example:

- 8 bits → 0 to 255

Think About This

- Unsigned numbers cannot be negative.

Where in real life do we *never* need negative numbers?

Try This (Range Math)

Answer without a calculator:

- With **4 bits**, what is the maximum number?
- With **8 bits**, why is the maximum **255**?
- With **16 bits**, what is the maximum?

Hint:

$2^n - 1$

Unsigned Integer Example (8-bit)

- All bits OFF → smallest number
- All bits ON → largest number

```
00000101 → 5
11111111 → 255
```

Signed Integers

- Signed integers can represent:
  - Positive numbers
  - Negative numbers
  - Zero

Computers commonly use:

- **Two’s complement** representation

Reflect

- Computers need a special trick to store negative numbers.
- Humans do not think about this at all.

Signed Integer Range (8-bit)

Using two’s complement:

- Range:
  - −128 to +127

Why?

- One bit is used for **sign**
- Negative numbers are stored using two’s complement.

Two’s complement

- Numbers “wrap around”
- This makes addition work naturally
- Take the positive number → invert bits (1's complement) → + 1 = gives -ve number

```
+1          = 00000001
invert      = 11111110
add 1       = 11111111   → −1
```

Integer Arithmetic

Computers can do:

- Addition
- Subtraction
- Multiplication
- Division

But with **fixed size**, problems can happen:

- **Overflow**
- **Underflow**

Overflow Example (8-bit Unsigned)

```
255 + 1 = ?

In Binary:

11111111 + 00000001 → 00000000
```

Result is incorrect because:

- No more bits available

Decimals Are Harder

Decimals:

- 0.1
- 3.14
- 1.25

Binary cannot represent all decimals exactly

Floating-Point Numbers

- Computers use **floating-point representation** (Standard: IEEE 754) for decimal

A floating-point number has:

- Sign
- Exponent
- Fraction (mantissa)

Why Decimals Are Hard in Binary

- Computers store numbers in **binary**
- Binary works well for:
  - 1/2, 1/4, 1/8
- Binary does **not** work well for:
  - 0.1, 0.2, 3.14

Many decimal fractions cannot be stored exactly

Try This

Which of these are easy in binary?

- 0.5
- 0.25
- 0.1

Why?

How Decimals Become Binary

To convert a decimal number:

- Split it into:
  - Integer part
  - Fractional part
- Integer part:
  - Divide by 2
- Fractional part:
  - Multiply by 2

Both parts are converted separately

Why Multiply by 2?

- Binary is a **base-2** number system
- Each fractional position is:
  - 1/2, 1/4, 1/8, ...

Multiplying by 2:

- Shifts the **next binary digit** into the integer place

Repeat the Process

For the fractional part:

```
fraction × 2 = integer + fraction
```

- Integer part → next binary digit
- Fractional part → repeat

Repeat until:

- Fraction becomes 0 (exact)
- Or bits run out (approximate)

Repeating Fractions in Binary

Some fractions **never end** in binary:

- 0.1
- 0.2
- 3.14

Just like:

- 1/3 = 0.3333... in decimal

Binary has its own repeating fractions

Finite Storage Means Approximation

- Computers have **limited bits**
- Infinite fractions must be:
  - Cut
  - Rounded

The stored value is:

- Very close
- But not exact

The Famous Example

```
0.1 + 0.2 = 0.30000000000000004
```

Why?

- 0.1 and 0.2 are already approximations
- Adding them adds tiny error

Why You Usually See Correct Answers

- Floating point has **high precision**
  - ~15 decimal digits
- Output is **rounded for display**

```
Internally:
0.30000000000000004
Displayed:
0.3
```

Getting Reliable Results

- Use **integers** when possible
- Avoid direct equality checks with decimals
- Accept approximation in science
- Use special decimal math for money

Summary

- Computers do not calculate decimals exactly
- They calculate them **predictably and approximately**
- High-precision math is done in software using integers and algorithms, not hardware floats.

Choosing Number Types

- Use **integers** for:
  - Counts
  - IDs
  - Loop counters
- Use **decimals (floating point)** for:
  - Measurements
  - Scientific values

Be careful with money!

Summary

- Integers are stored **exactly**
- Signed numbers use **two’s complement**
- Fixed size can cause **overflow**
- Decimals are stored **approximately**

Section 3: Hexadecimal

In this section, we will learn:

- Why binary is hard for humans
- What **hexadecimal** is
- How hex maps neatly to binary
- Where hex is used in real systems

Problem with Binary for Humans

Binary numbers are long:

```
110101101011
```

Hard to read and debug for humans.

Think About This

- Computers like binary.
- Humans do not.

Why do you think programmers still learn binary at all?

Hexadecimal Representation

- Base 16 number system
- Uses:
  - 0–9
  - A–F

Hex	Decimal
A	10
F	15

Binary and Hex Relationship

- 1 hex digit = 4 bits
- 2 hex digits = 1 byte

Example:

```
Binary: 01000001
Hex: 41
```

Try This (Binary ↔️ Hex)

Convert:

- Binary: `1111`
- Binary: `1010`

Into hex.

Where You See Hex

- Memory addresses
- Debuggers
- Machine code
- Colors:
  - `#FF0000` = Red

Real-Life Analogy: Short Names

- Binary → long phone number
- Hex → saved contact name

Same value, easier to read.

Summary

- Hexadecimal is **base 16**
- 1 hex digit = **4 bits**
- Hex is easier for humans to read and debug

Section 4: Representing Text

In this section, we will learn:

- How characters become **numbers**
- What **ASCII** can and cannot do
- Why **Unicode** exists
- How **UTF-8** saves space

Representing Characters

- Computers store:
  - Letters
  - Digits
  - Symbols
- Characters are stored as **numbers**

ASCII

- ASCII = American Standard Code for Information Interchange
- Each character → number
- Stored in one byte

Examples:

- `A` → 65
- `a` → 97

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Credit: Wikipedia - ASCII Page

Think About This

- ASCII worked well for English.
- It failed for most of the world.
- What do you think happens when technology ignores language diversity?
- How do you think this shaped who could use computers first?

ASCII Example

Character → Number → Binary

- A → 65 → `01000001`
- B → 66 → `01000010`

Problem with ASCII

- Works only for English
- Cannot represent:
  - Indian languages
  - Many world scripts
  - Emojis

Why Unicode?

- One system for **all languages**
- Every character gets a unique number
- A standard body, the Unicode Consortium, defines this

Examples:

- A → U+0041
- ಅ → U+0C85
- 😍 → U+1F60A

Credit: Unicode Kannada Chart - PDF

Unicode Needs More Than One Byte

- Unicode assigns **a number** to every character
- Some numbers are **small**
- Some numbers are **very large**

So one byte (8 bits) is **not enough** for all characters

- Remember: Unicode assigns numbers, not how they are encoded or stored

Fixed Length Encodings

- UCS-2
  - Uses **2 bytes** (16 bits)
  - Cannot encode emojis or many modern characters
  - Obsolete
- UCS-4
  - Uses **4 bytes** (32 bits)
  - Can encode all Unicode
- Every character uses the **same space**
- UCS-2 and UCS-4 are **Fixed-width encoding **

Why This Wastes Space

Example:

- English letter `A`
  - Needs very small number
  - But still uses **2 or 4 bytes**

Result:

- Simple text wastes memory
- Files become bigger
- Slower storage and transfer

We Need a Better Way

What we want:

- Small characters → **use less space**
- Large characters → **use more space**
- Still support all languages
- This leads to **variable-length encoding**

Variable-Length Encodings

- UTF-8
  - Fully variable-length (1–4 bytes)
- UTF-16
  - Uses 2 or 4 bytes (mostly fixed, sometimes variable)
  - UTF-16 is partially variable-length
- UTF-8 is the most popular one, why?

UTF-8: A Smarter Encoding

- UTF-8 uses **variable number of bytes**
- 1 to 4 bytes per character

Rules:

- English → 1 byte
- Indian languages → 2 or 3 bytes
- Emojis → 4 bytes

Why UTF-8?

- Uses 1 to 4 bytes
- English uses less space
  - Which is the most popular language on the web
- Compatible with ASCII
  - No need to re-encode
- Used on the web and all modern operating systems

Example: English Character

Character: `A`

Unicode number: `U+0041`

Encoding	Bytes Used
UTF-8	1 byte

UTF-8 saves space for English text

Example: Kannada Character

Character: ಅ

Unicode number: `U+0C85`

Encoding	Bytes Used
UTF-8	3 bytes

UTF-8 uses more bytes only when needed

Summary

- Characters are stored as **numbers**
- ASCII is limited to English
- Unicode supports **all languages**
- UTF-8 is efficient and widely used

Big Picture

- Computers use **binary**
- Bit → smallest unit
- Byte → 8 bits
- Numbers and characters become binary
- Decimals are difficult
- Unicode supports all languages
- Hex helps humans read binary

Think About This

- Why not decimal computers?
- Why do emojis need more bytes?
- Why do programmers like hex?

Thank you!

Any questions?

Thejesh GN