# Bits, Bytes, Integers and fractional decimal numbers

N. Navet - Computing Infrastructure 1 / Lecture 1

[edited NN]

# Today: Bits, Bytes, and Integers

- Representing information as bits
- Representing characters
- Numeral systems (additive and positional) and basis
- Data representations in memory
- Encoding integers: unsigned and signed
- Encoding fractional decimal numbers
- Encoding floating point numbers

We will cover the 3 most important representations of numbers:

✓ *Unsigned* encoding for positive integers

✓ *Two's-complement* encoding for signed integers

✓ *Floating-point* encoding for real numbers

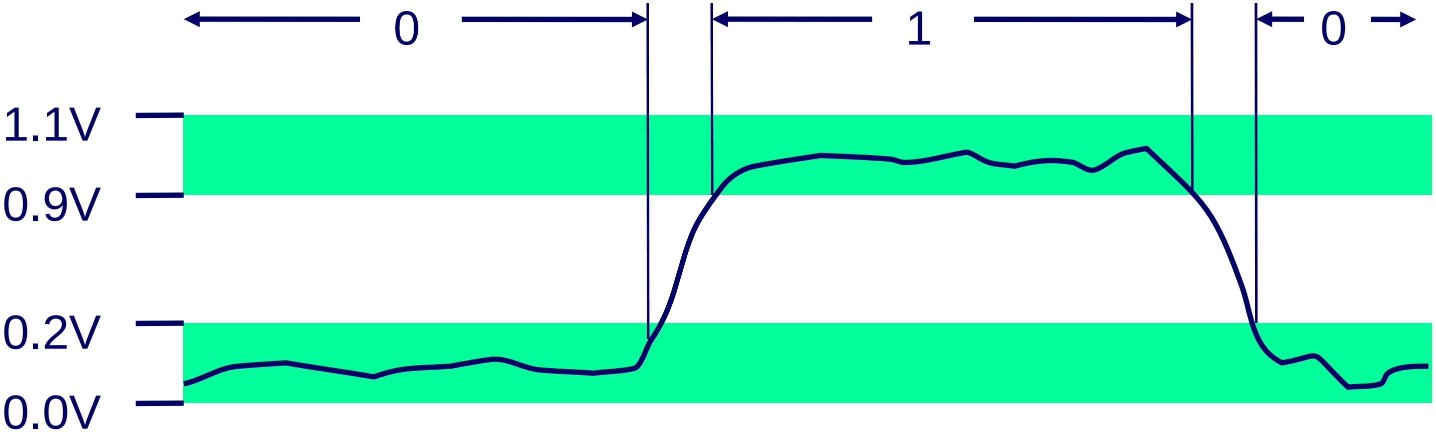
**[edited NN]**

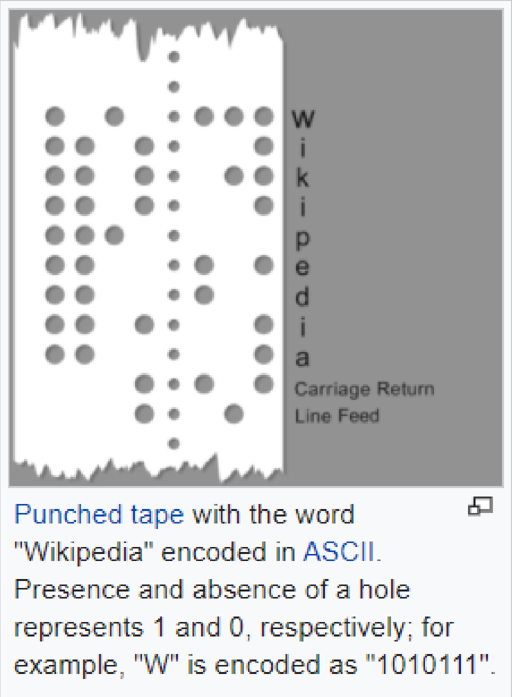
# Nowadays, everything is bits in computers

- **Each bit is 0 or 1 – a byte is a sequence of 8 bits**
- By encoding/interpreting sets of bits in various ways
  - **Computers determine what to do (instructions)**
  - **… and represent and manipulate data: numbers, strings, etc…**

**both**

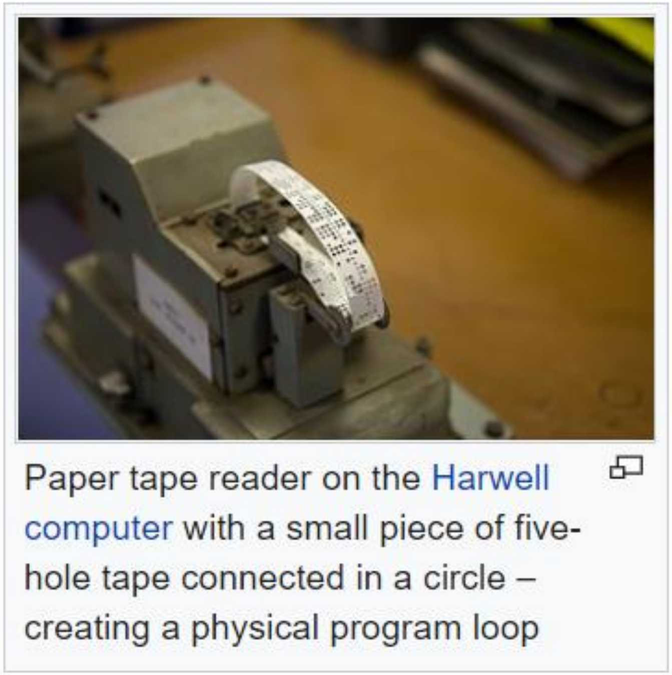
- **Why bits? Electronic implementation is cheap and reliable**
  - Easy to store in memory with "bistable" elements (only two stable configurations or states, corresponding to different voltages)
  - Reliably transmitted on noisy and inaccurate wires





Punched tape with the word "Wikipedia" encoded in ASCII. Presence and absence of a hole represents 1 and 0, respectively; for example, "W" is encoded as "1010111".

https://en.wikipedia.org/wiki/Character_encoding



Paper tape reader on the Harwell computer with a small piece of five-hole tape connected in a circle – creating a physical program loop

https://en.wikipedia.org/wiki/Punched_tape

**Characters and code stored on paper, not numerically**

# REPRESENTING CHARACTERS

# Characters encoding

- ❑ ASCII (American Standard Code for Information Interchange)
  - ❑ 127 char. including 95 printable char. – stored using 1 byte (= 8bits) per character – ok for English but not for most languages: French ('ç'), German, Greek, Chinese, …
- ❑ Unicode (industry standard developed by the *Unicode consortium* since 1990)
  - ❑ Latest: over 143,00 characters covering 154 modern and historic scripts
  - ❑ Standard defines UTF-8, UTF-16, and UTF-32 (each can represent anything the others can represent but their size is ≠, UTF-32 char. always 4 bytes long)
    - ❑ ex: UTF-8, dominantly used by websites (over 90%), uses one byte for the first 128 "code points" (index of the character in the table), and up to 4 bytes for other characters. The first 128 Unicode code points are the ASCII characters, which means that any ASCII text is also a UTF-8 text"
- ❑ EBCDIC (Extended Binary Coded Decimal Interchange Code) → from IBM, disappearing

**How many languages in the world?**

Around 6900, see
https://www.linguisticsociety.org/content/how-many-languages-are-there-world
nb: not all are written, and many rely on the same characters set

# Representing ASCII strings

```
char S[6] = "18213";
```

- **Example: “null terminated string” in C/C++**
  - Represented by array of characters
  - Each character encoded in ASCII format:
    - E.g: character “0” has code 0x30 - Digit *i* has code 0x30+*i*
  - String should be null-terminated
    - Final character = 0 (NUL character whose Ascii value is zero)

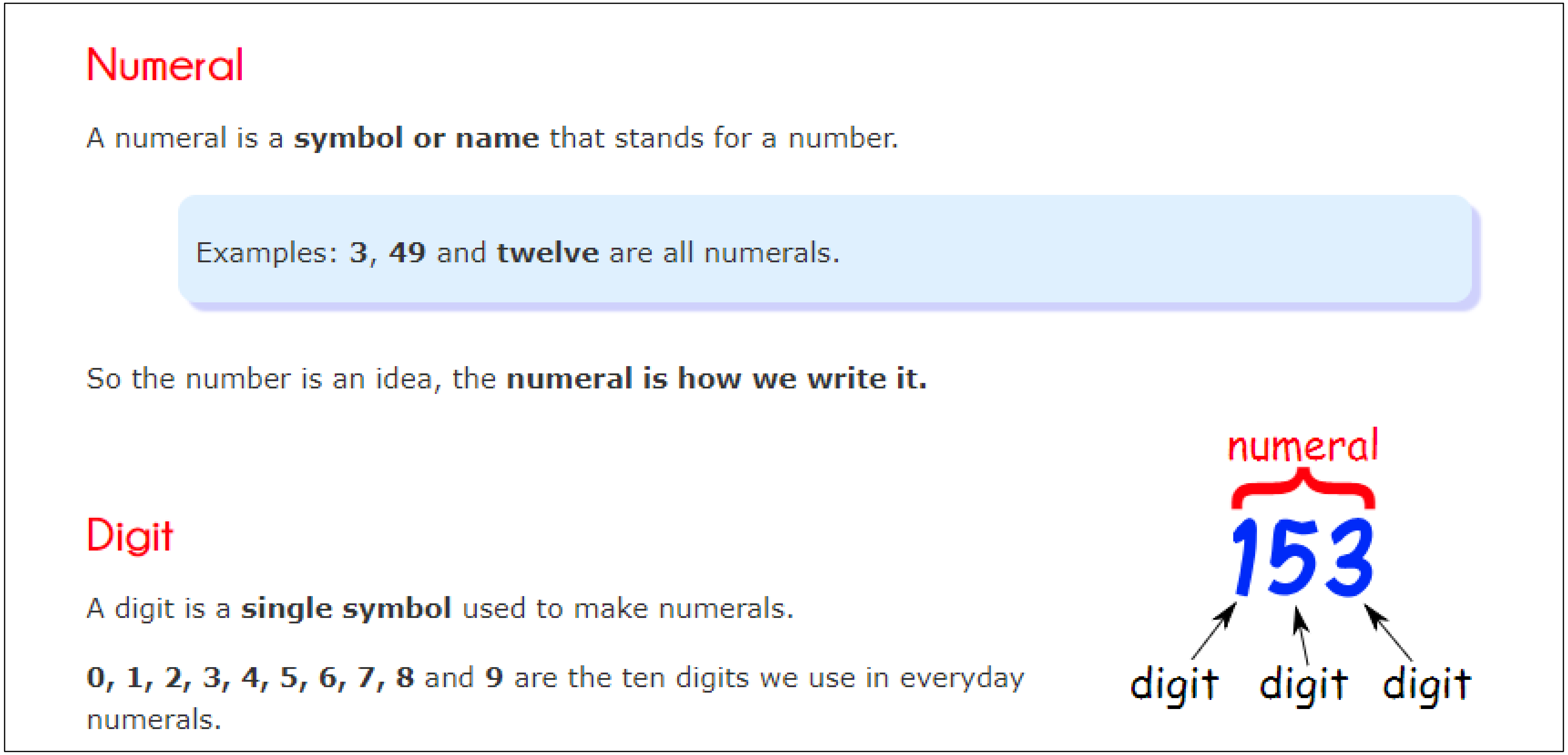
**Memory**

| | |
|---|---|
| 31 | 0x100 |
| 38 | 0x101 |
| 32 | 0x102 |
| 31 | 0x103 |
| 33 | 0x104 |
| 00 | 0x105 |

in hex.

- **In other programming languages**
  - Memory representation depends on the character set and the programming language
  - “null terminated string” is not at all a standard, e.g. strings can be stored as records with a field indicating the size

Quite complex in Python! see
https://rushter.com/blog/python-strings-and-memory/



## Numeral

A numeral is a **symbol or name** that stands for a number.

> Examples: **3**, **49** and **twelve** are all numerals.

So the number is an idea, the **numeral is how we write it.**

## Digit

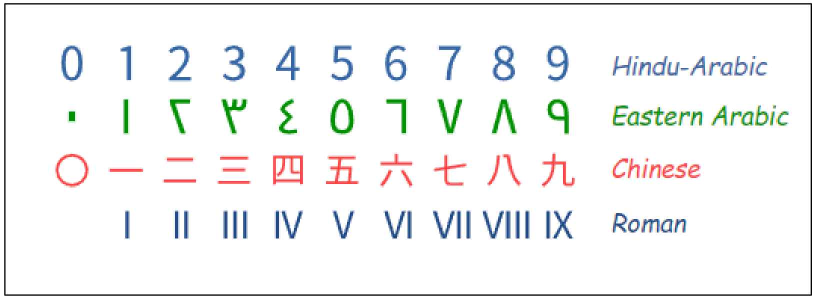
A digit is a **single symbol** used to make numerals.

**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8** and **9** are the ten digits we use in everyday numerals.

https://www.mathsisfun.com/numbers/numbers-numerals-digits.html

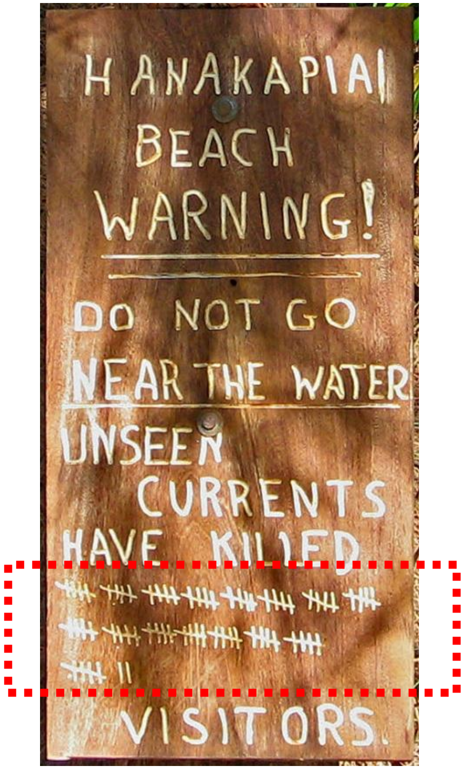
# NUMERAL SYSTEMS AND BASIS

Other Types of Digits and Numerals throughout history



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | *Hindu-Arabic* |
| ٠ | ١ | ٢ | ٣ | ٤ | ٥ | ٦ | ٧ | ٨ | ٩ | *Eastern Arabic* |
| 〇 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 九 | *Chinese* |
| | I | II | III | IV | V | VI | VII | VIII | IX | *Roman* |

# Numeral Systems – ways to represent numbers



[Wikipedia Tally Marks]

**Additive systems**
**With two symbols**

Positional system = the contribution of a digit to the value of the number depends on its position

## VS Positional Systems

E.g. in the decimal system (base 10), the numeral 4327 means $(\mathbf{4}\times10^3) + (\mathbf{3}\times10^2) + (\mathbf{2}\times10^1) + (\mathbf{7}\times10^0)$ noting that $10^0 = 1$.

**Numeral system with just 1 symbol?**

# Positional Numeral Systems – different basis

- **Base (=radix) :**
  - The number of different symbols (digits) needed to represent any given number

- **The larger the base, the more digits are used**
  - **Base 10:** 0,1,2,3,4,5,6,7,8,9
  - **Base 2:** 0,1
  - **Base 8:** 0,1,2,3,4,5,6,7
  - **Base 16 :** 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

**Why do people use base 10 representation ?**

- **For a given number, the larger the base**
  - **the more symbols required**
  - **but the fewer digits needed for a number**

For programmers, Hexadecimal (base 16) is a good tradeoff between base 2 (too verbose) and base 10 (not easy to convert to base 2)

1 digit in Hexa = 4 bits

# Encoding Byte Values

Notation: $X_Y$ denotes string of digits X expressed in base Y
Assume Base 10 if not mentioned

- **Byte = 8 bits**
  - Binary $00000000_2$ to $11111111_2$
  - Decimal: $0_{10}$ to $255_{10}$
  - Hexadecimal $00_{16}$ to $FF_{16}$
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - $FA1D37B_{16}$ in many programming languages is written as:
      - 0xFA1D37B or
      - 0xfa1d37b (case insensitive)

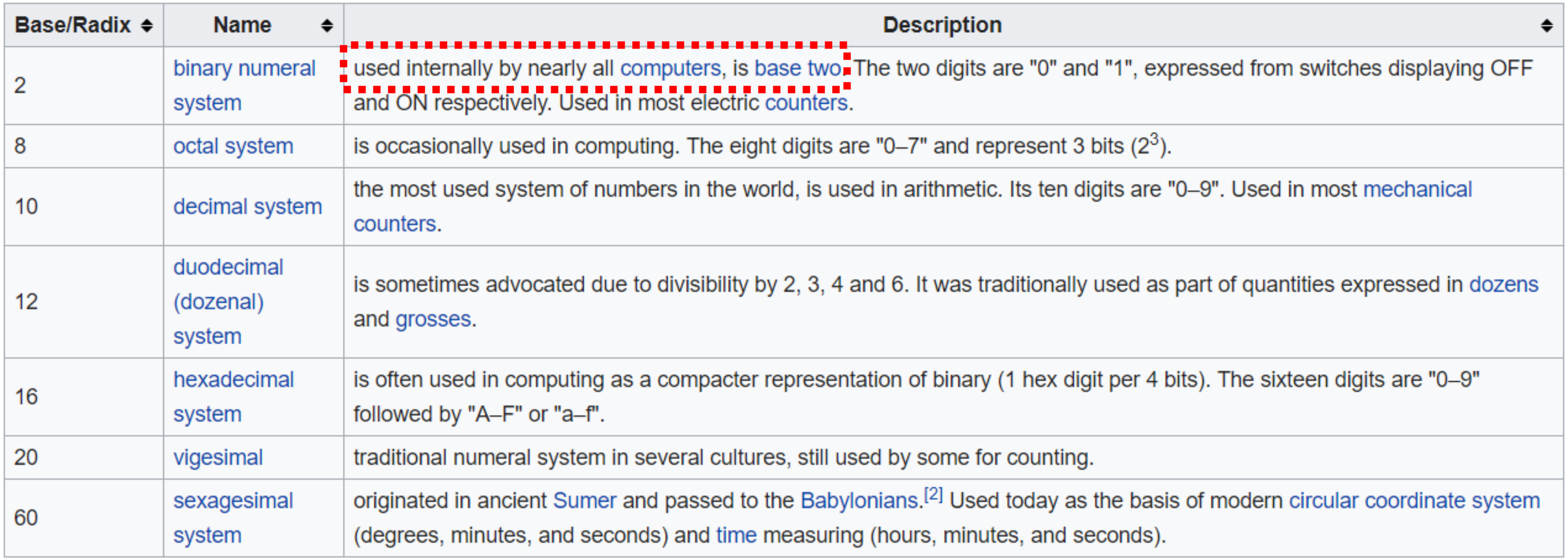
| Hex | Decimal | Binary |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

Example: 13 = 2^3 + 2^2 + 2^0

[edited NN]

# Different basis

**Notation: $X_Y$ denotes string of digits X expressed in base Y**



| Base/Radix | Name | Description |
|---|---|---|
| 2 | binary numeral system | used internally by nearly all computers, is base two. The two digits are "0" and "1", expressed from switches displaying OFF and ON respectively. Used in most electric counters. |
| 8 | octal system | is occasionally used in computing. The eight digits are "0–7" and represent 3 bits ($2^3$). |
| 10 | decimal system | the most used system of numbers in the world, is used in arithmetic. Its ten digits are "0–9". Used in most mechanical counters. |
| 12 | duodecimal (dozenal) system | is sometimes advocated due to divisibility by 2, 3, 4 and 6. It was traditionally used as part of quantities expressed in dozens and grosses. |
| 16 | hexadecimal system | is often used in computing as a compacter representation of binary (1 hex digit per 4 bits). The sixteen digits are "0–9" followed by "A–F" or "a–f". |
| 20 | vigesimal | traditional numeral system in several cultures, still used by some for counting. |
| 60 | sexagesimal system | originated in ancient Sumer and passed to the Babylonians.[2] Used today as the basis of modern circular coordinate system (degrees, minutes, and seconds) and time measuring (hours, minutes, and seconds). |

[Wikipedia Radix]

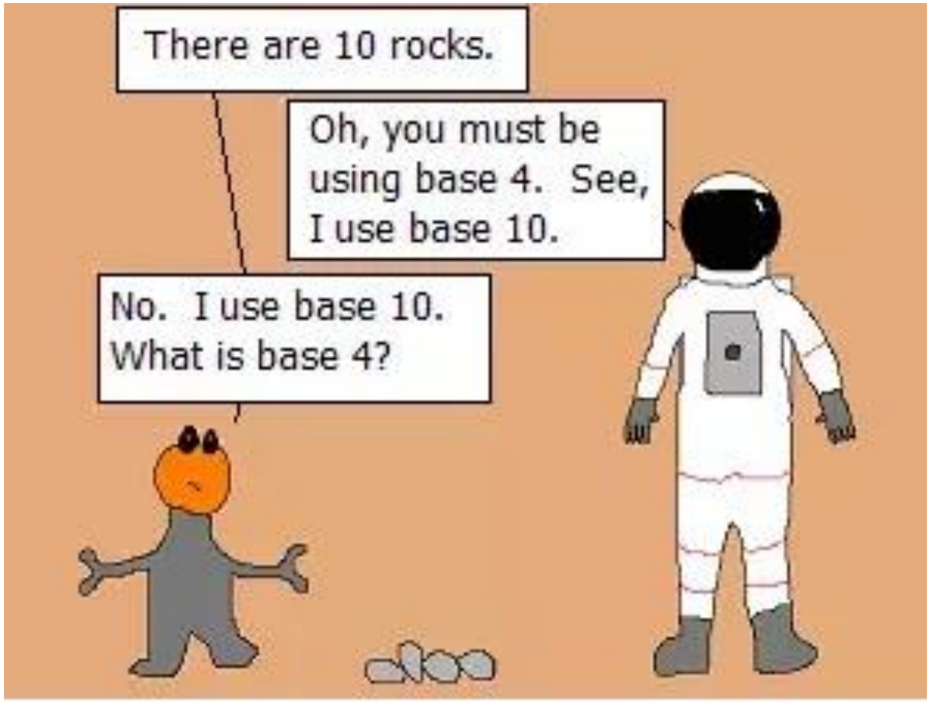
- In base $b$ ($b > 1$), a **string of digits** $d_1 \dots d_n$ denotes the number $d_1 \cdot b^{n-1} + d_2 \cdot b^{n-2} + \dots + d_n \cdot b^0$, where $0 \le d_i < b$.

**1) Express $20_{10}$ in binary, hexadecimal and octal basis**

**2) What is the decimal value of $20_{16}$ , $20_8$ and $20_2$ ?**

# Positional Numeral Systems – different basis

Notice there are 4 rocks and the alien has 4 fingers!


There are 10 rocks.
Oh, you must be using base 4. See, I use base 10.
No. I use base 10. What is base 4?

Every base is base 10.

[Sanjay Kulkarni]

- ✓ The alien cannot understand 4 as digit 4 does not exist in its base (4 in based 10 is expressed as '10' in base 4)
- ✓ the number of unique symbols in any base is '10' in that base

# REPRESENTATION IN MEMORY

# Machine Words

- **Any given computer has a “Word Size”**
  - **Nominal size of integers, memory addresses and operands of most instructions manipulating integers**

  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB ($2^{32}$ bytes)

- Most machines now have 64-bit word size
  - Potentially, could have 18 EB (exabytes) of addressable memory
  - That’s 18.4 x $10^{18}$

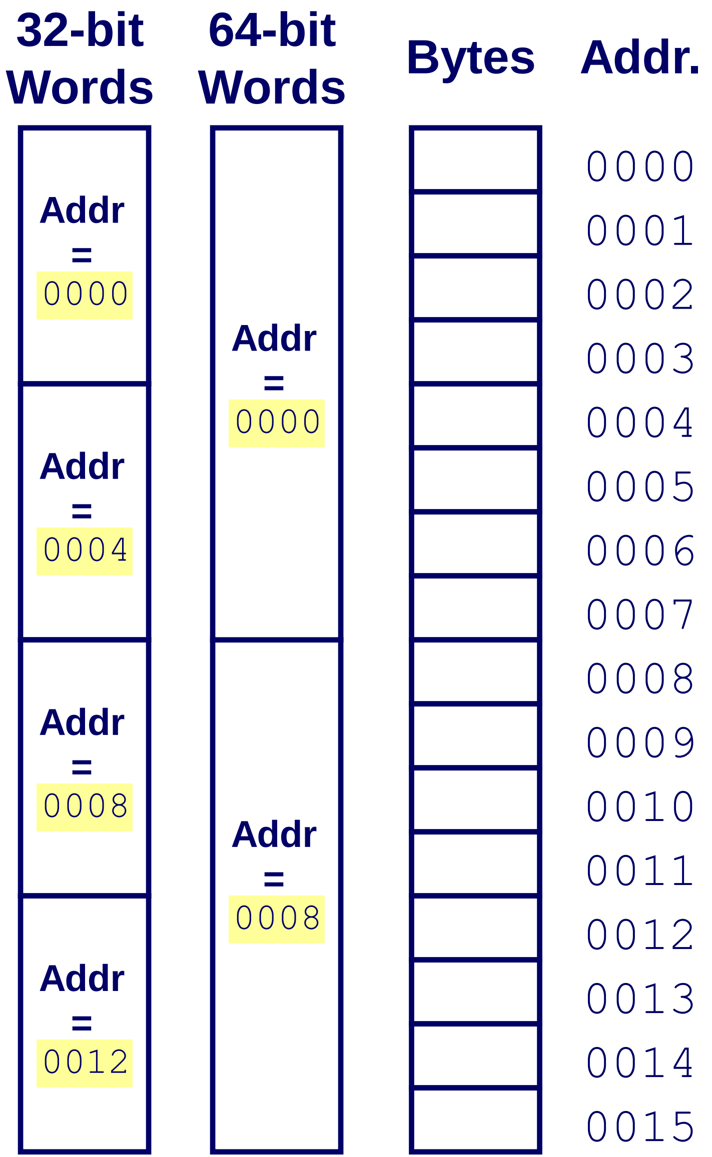
  - Machines have instructions for manipulating multiple data formats
    - Fractions or multiples of word size. Ex: 2, 4, 8-byte integers
    - But always an integral number of bytes!

# Word-Oriented Memory Organization

Conceptually, memory is a very large array of bytes storing instructions and data but organized in words

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit) depending on word size

- **Note: system provides private address spaces** (= the virtual address space) **to each "process"**
  - A process is a program being executed
  - So, a program can work like it has a dedicated machine



[edited NN]

# Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**
- **Conventions**
  - **Big Endian:** Sun (outdated), PPC Mac (outdated), Internet protocols
    - Least significant byte has highest address
  - **Little Endian:** x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address
- Bi-endian: "Some architectures (e.g., Intel Itanium - IA-64) feature a setting which allows for switchable endianness in data fetches and stores, instruction fetches, or both."

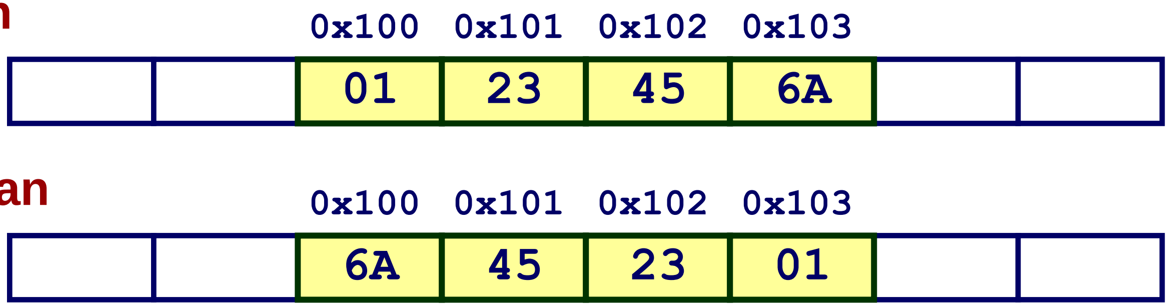
Big-endian is the most common format in data networking - fields in the protocols of the Internet protocol suite, such as IPv4, IPv6, TCP, and UDP, are transmitted in big-endian order.

# Byte Ordering Example

✓ Different compilers & OS assign different locations in memory to objects
✓ Possible to have different memory locations at each run

- **Example**
  - Variable x has 4-byte value of 0x0123456A
  - Stored at memory address 0x100 (from 0x100 to 0x103)

**Represent how variable x in stored in memory from address 0x100 on a Big Endian and Little Endian machine (least significant byte has lowest address)**



**Big Endian**

|  |  | 0x100 | 0x101 | 0x102 | 0x103 |  |  |
|---|---|---|---|---|---|---|---|
|  |  | 01 | 23 | 45 | 6A |  |  |

**Little Endian**

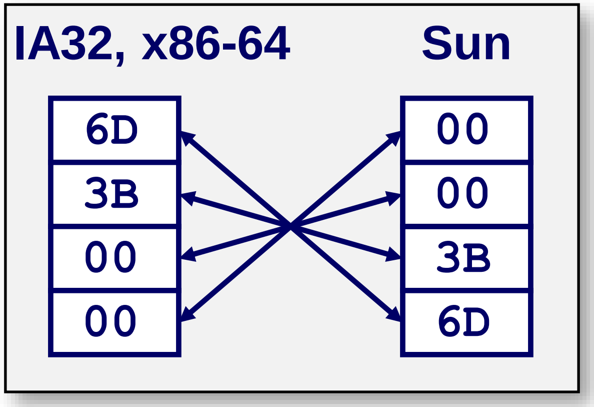
|  |  | 0x100 | 0x101 | 0x102 | 0x103 |  |  |
|---|---|---|---|---|---|---|---|
|  |  | 6A | 45 | 23 | 01 |  |  |

# Representing Integers

***long int*** **in C is 4 bytes on 32bit CPUs and 8 bytes on 64 bit CPUs.**
**Better off using *fixed width integer types* such as *int64_t* available in C99**

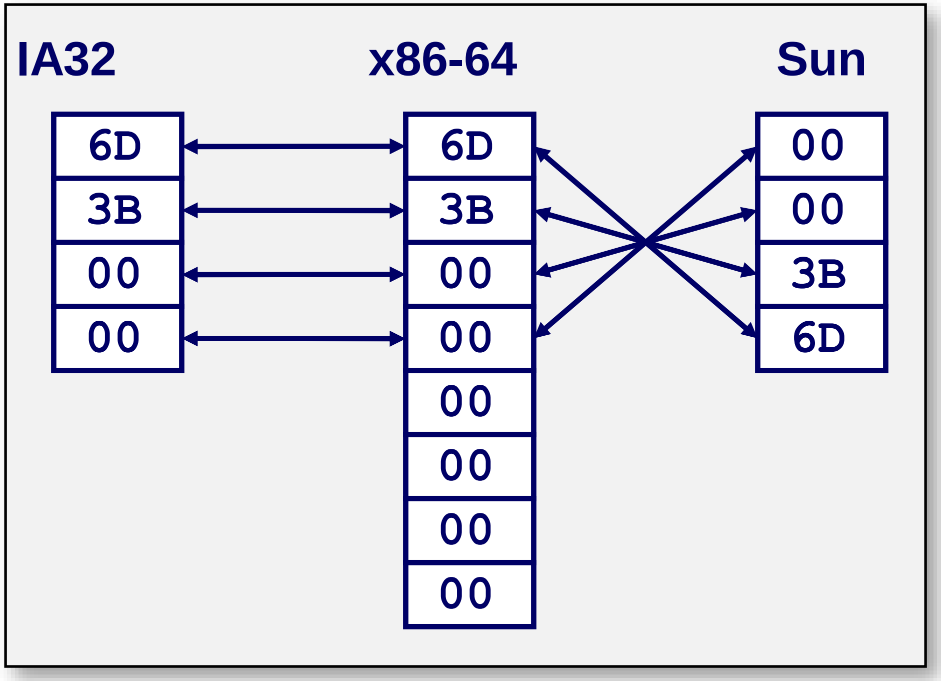
| | | | | |
|---|---|---|---|---|
| **Decimal:** | 15213 | | | |
| **Binary:** | 0011 | 1011 | 0110 | 1101 |
| **Hex:** | 3 | B | 6 | D |

```
int A = 15213; /* 4 bytes */
```



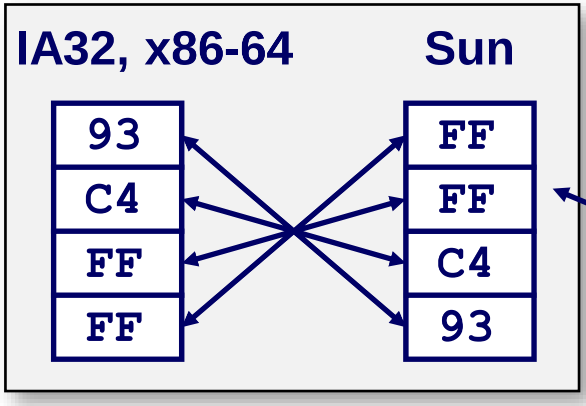
| IA32, x86-64 | Sun |
|---|---|
| 6D | 00 |
| 3B | 00 |
| 00 | 3B |
| 00 | 6D |

```
long int C = 15213;
```



| IA32 | x86-64 | Sun |
|---|---|---|
| 6D | 6D | 00 |
| 3B | 3B | 00 |
| 00 | 00 | 3B |
| 00 | 00 | 6D |
| | 00 | |
| | 00 | |
| | 00 | |
| | 00 | |

```
int B = -15213;
```

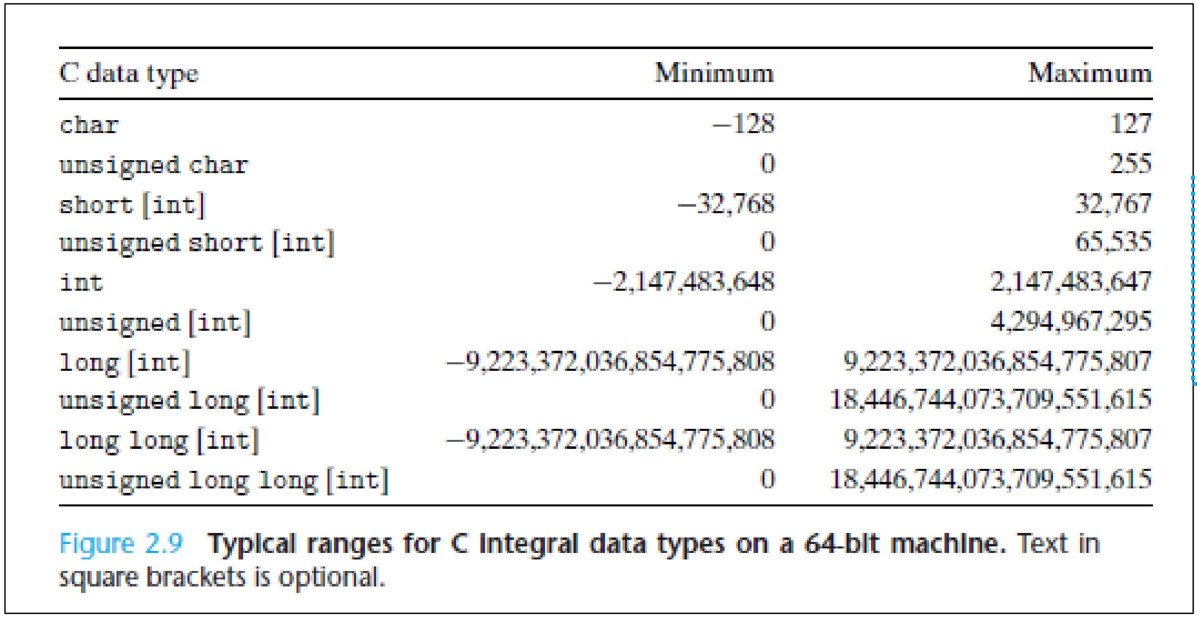


| IA32, x86-64 | Sun |
|---|---|
| 93 | FF |
| C4 | FF |
| FF | C4 |
| FF | 93 |

**Two's complement representation used for signed integers, introduced later**

# ENCODING INTEGERS
# A) UNSIGNED ENCODING
# B) TWO’S COMPLEMENT FOR SIGNED INTEGERS

# Ranges for integer types



| C data type | Minimum | Maximum |
|---|---:|---:|
| `char` | −128 | 127 |
| `unsigned char` | 0 | 255 |
| `short [int]` | −32,768 | 32,767 |
| `unsigned short [int]` | 0 | 65,535 |
| `int` | −2,147,483,648 | 2,147,483,647 |
| `unsigned [int]` | 0 | 4,294,967,295 |
| `long [int]` | −9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| `unsigned long [int]` | 0 | 18,446,744,073,709,551,615 |
| `long long [int]` | −9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| `unsigned long long [int]` | 0 | 18,446,744,073,709,551,615 |

Figure 2.9 **Typical ranges for C integral data types on a 64-bit machine.** Text in square brackets is optional.

**#bits needed to store an *unsigned int* ?**

**Range depends on the language and word size of the machine**

Integers in Python 3 are of unlimited size (capped by machine memory) – drawback is speed as CPU instructions not directly used

# Encoding Integers

- **Commonly-used lengths for integers are 8,16,32,64 bits. There are 2 types of integers:**
- **Unsigned Integers:** can represent zero and positive integers.
- **Signed Integers:** can represent zero, positive **and** negative integers. Several distinct representation schemes have been proposed for signed integers, e.g:
  - Sign-Magnitude representation
  - 1's Complement representation
  - 2's Complement representation

Modern computers all operate based on 2's complement representation because it allows for cheap and fast hardware

# Notations

- We denote the vector made up of $[x_{w-1}, x_{w-2}, \ldots, x_0]$ the individual bits  of an integer data type of *w* bits written in binary notation
- Function *B2U()* (binary-to-unsigned) takes as input this vector and returns its unsigned interpretation
- Similarly function *B2T()* (binary-to-two's complement) returns its signed interpretation

# Encoding Integers

Binary vector of bits $[x_{w-1}, x_{w-2}, \ldots, x_0]$

## Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

## Two's Complement

$$B2T(X) = \boxed{-x_{w-1} \cdot 2^{w-1}} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

**Sign Bit**

```
short int x =  15213;
short int y = -15213;
```

- **C `short` type is assumed 2 bytes long here**

| | Decimal | Hex | Binary |
|---|---|---|---|
| **x** | 15213 | 3B 6D | 00111011 01101101 |
| **y** | -15213 | C4 93 | 11000100 10010011 |

- **Sign Bit**
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

**Let's consider integers of 4 bits, what is the value of 1111 in unsigned and two's complement encodings ?**

# Encoding Integers

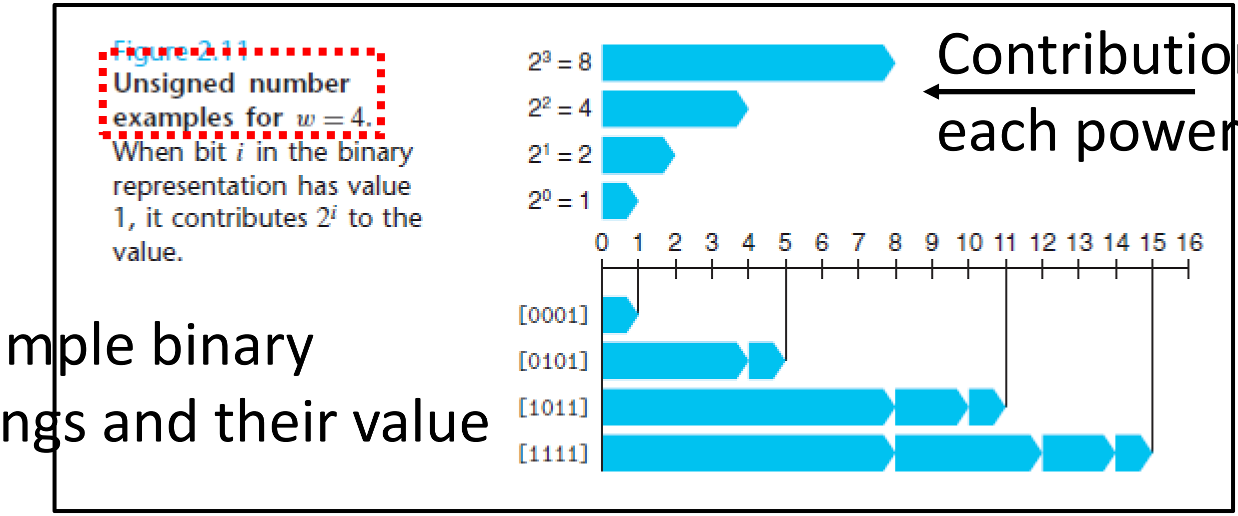


Figure 2.11 Unsigned number examples for $w = 4$. When bit $i$ in the binary representation has value 1, it contributes $2^i$ to the value.

Contribution of each power of two

Example binary strings and their value

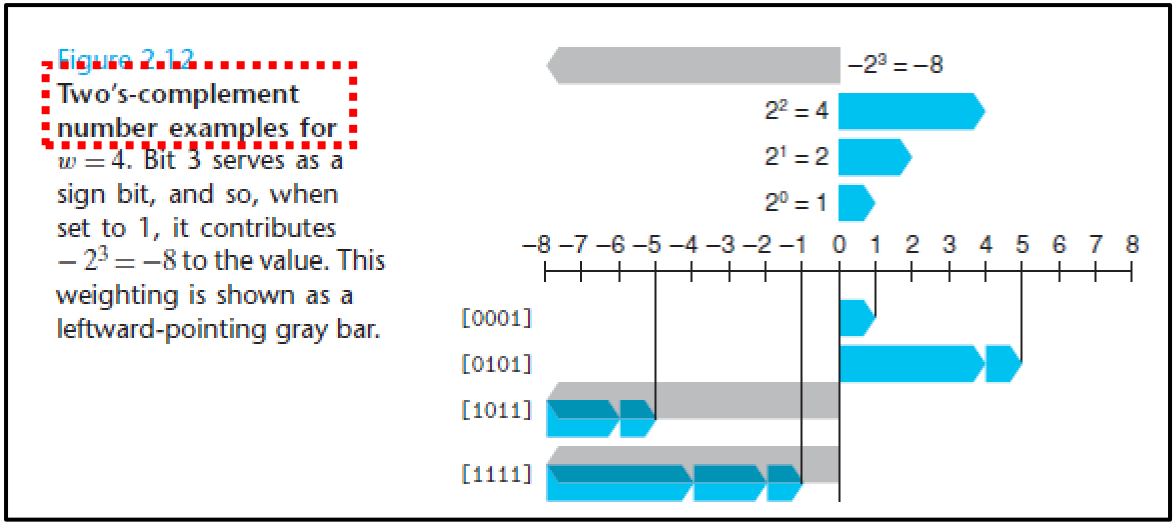


Figure 2.12 Two's-complement number examples for $w = 4$. Bit 3 serves as a sign bit, and so, when set to 1, it contributes $-2^3 = -8$ to the value. This weighting is shown as a leftward-pointing gray bar.

# Two-complement Encoding Example (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

| Weight | 15213 | | -15213 | |
|---:|:---:|---:|:---:|---:|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 2 |
| 4 | 1 | 4 | 0 | 0 |
| 8 | 1 | 8 | 0 | 0 |
| 16 | 0 | 0 | 1 | 16 |
| 32 | 1 | 32 | 0 | 0 |
| 64 | 1 | 64 | 0 | 0 |
| 128 | 0 | 0 | 1 | 128 |
| 256 | 1 | 256 | 0 | 0 |
| 512 | 1 | 512 | 0 | 0 |
| 1024 | 0 | 0 | 1 | 1024 |
| 2048 | 1 | 2048 | 0 | 0 |
| 4096 | 1 | 4096 | 0 | 0 |
| 8192 | 1 | 8192 | 0 | 0 |
| 16384 | 0 | 0 | 1 | 16384 |
| -32768 | 0 | 0 | 1 | -32768 |
| **Sum** | | **15213** | | **-15213** |

# Numeric Ranges

Considering a *w*-bit long signed integer, what is the numeric range *[Tmin,Tmax]* with two's complement encoding ?

- **Unsigned Values**
  - *UMin* = 0

    000…0
  - *UMax* = $2^w - 1$

    111…1

*w* is the length of the vector of bits

$$B2T(X) = \boxed{-x_{w-1} \cdot 2^{w-1}} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

- **Two's Complement Values**
  - *TMin* = $-2^{w-1}$

    100…0
  - *TMax* = $2^{w-1} - 1$

    011…1

- **Other noteworthy value**
  - Minus 1

    111…1

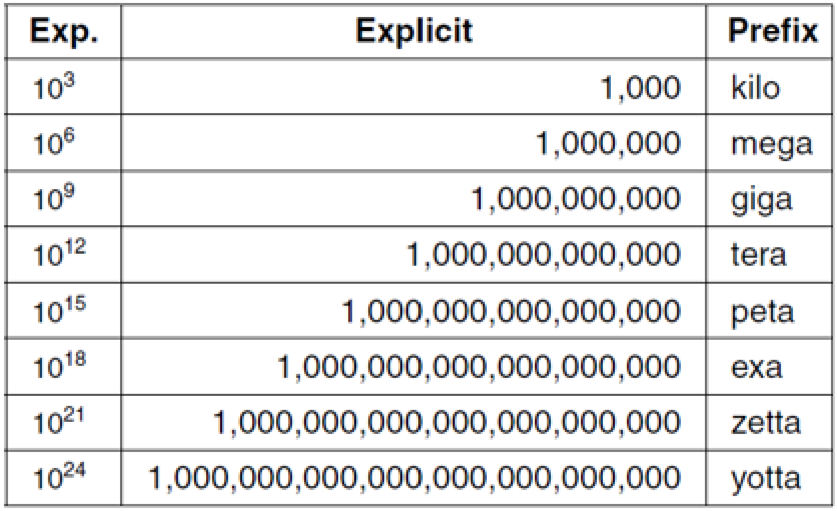
**Important values for *word size* of 16**

| | Decimal | Hex | Binary |
|---|---:|---:|---|
| UMax | 65535 | FF FF | 11111111 11111111 |
| TMax | 32767 | 7F FF | 01111111 11111111 |
| TMin | -32768 | 80 00 | 10000000 00000000 |
| -1 | -1 | FF FF | 11111111 11111111 |
| 0 | 0 | 00 00 | 00000000 00000000 |

# Numeric ranges for different Word Sizes

| | W | | | |
|---|---|---|---|---|
| | **8** | **16** | **32** | **64** |
| **UMax** | 255 | 65,535 | 4,294,967,295 | 18,446,744,073,709,551,615 |
| **TMax** | 127 | 32,767 | 2,147,483,647 | 9,223,372,036,854,775,807 |
| **TMin** | -128 | -32,768 | -2,147,483,648 | -9,223,372,036,854,775,808 |

- **Observations**
  - |*TMin* | = *TMax* + 1
    - **Asymmetric range: there is one more negative value than positive value**
  - *UMax* = (2 * *Tmax)* + 1



| Exp. | Explicit | Prefix |
|---|---|---|
| $10^3$ | 1,000 | kilo |
| $10^6$ | 1,000,000 | mega |
| $10^9$ | 1,000,000,000 | giga |
| $10^{12}$ | 1,000,000,000,000 | tera |
| $10^{15}$ | 1,000,000,000,000,000 | peta |
| $10^{18}$ | 1,000,000,000,000,000,000 | exa |
| $10^{21}$ | 1,000,000,000,000,000,000,000 | zetta |
| $10^{24}$ | 1,000,000,000,000,000,000,000,000 | yotta |

# Unsigned & Signed Numeric Values

| *X* | B2U(*X*) | B2T(*X*) |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | –8 |
| 1001 | 9 | –7 |
| 1010 | 10 | –6 |
| 1011 | 11 | –5 |
| 1100 | 12 | –4 |
| 1101 | 13 | –3 |
| 1110 | 14 | –2 |
| 1111 | 15 | –1 |

- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

# Conversions between types

Many programming languages like C/C++ provide two kinds of functions:

- Binary re-interpretation of the bit field in a new type: the bit field stay identical but how these bits are interpreted change – also called casting

**Let’s consider a variable of type unsigned int (16 bits) assigned to FFFF. What is the result of “casting” it into a 16-bit signed int ?**

- Proper conversion: the bit field may not remain identical after conversion (e.g., conversion from a float to an integer)

Python only provides conversion!