# COMP2611 COMPUTER ORGANIZATION
# DATA REPRESENTATION

# Numbers

- **Bits** are able to represent both **data and instruction** in digital computers

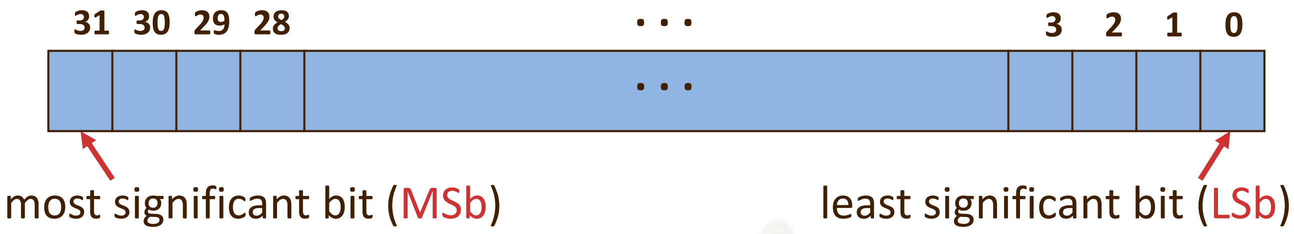
- **In this topic, you will learn:**
  - How to represent integer numbers (both signed and unsigned)?
  - How to represent fractions and real numbers?
  - What is a representable range of numbers in a computer?

- **To be covered in Computer Arithmetic topic:**
  - Arithmetic operations: How to add, subtract, multiply, divide binary numbers
  - How to build the hardware that takes care of arithmetic operations

# SIGNED AND UNSIGNED INTEGER

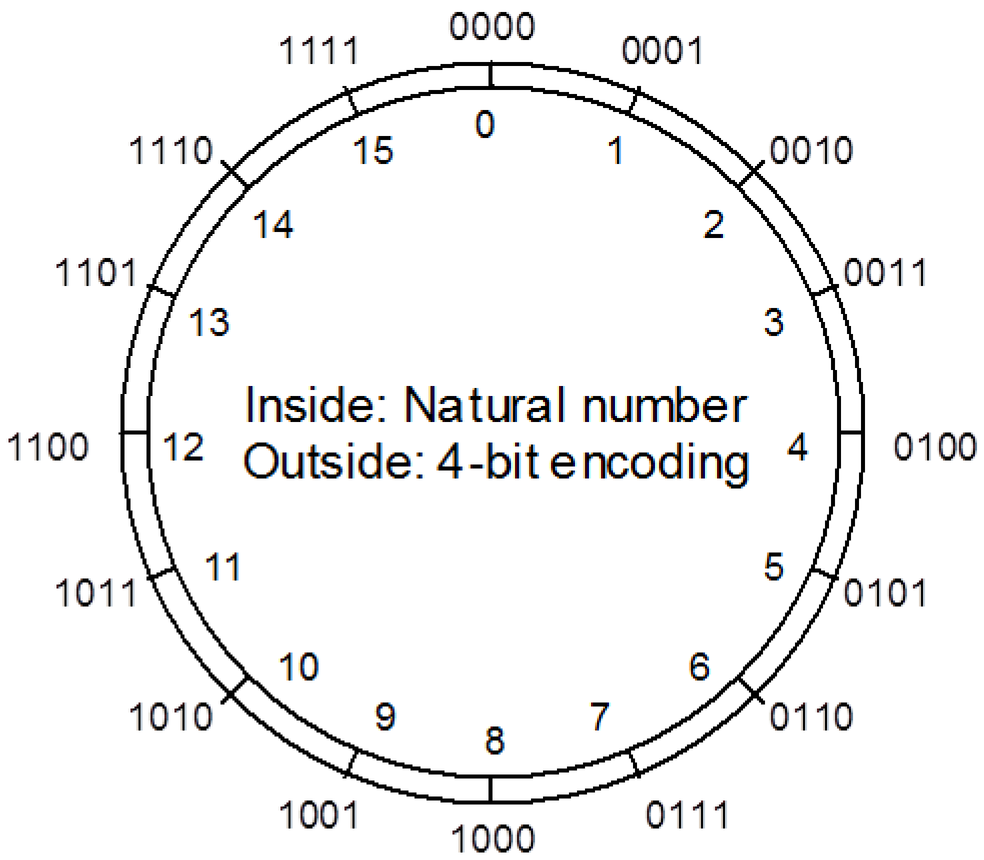
# Review: Base, Representation and Value

- Numbers can be represented in any **base**
  - Decimal, binary, hexadecimal
- Positional notation helps to calculate the numerical value represented
- A sequence of bits (a.k.a. **bit sequence**) usually work together
- Bits are grouped and numbered 0, 1, 2, 3 … from right to the left:



# Unsigned Binary Integers

- Given k bits
- Min = $000.....00_2 = 0_{10}$
- Max = $111.....11_2 = 2^k - 1_{10}$
- Representable range is $[0, 2^k - 1]$



# 2's Complement

- Most common scheme of representing negative numbers in computers
- Affords natural arithmetic (no special rules!)
- The most significant bit is called the sign bit
  - 0 for non-negative, 1 for negative
- The positive half uses the same representation as before
- To represent a negative number in 2's complement notation…
  1. Decide upon the number of bits (n)
  2. Find the binary representation of the positive value in n-bits
  3. Flip all the bits (change 1's to 0's and vice versa)
  4. Add 1

# 2's Complement Example

- What is the representation of -6 in 2's complement on 4 bits?

i) Start from the representation of +6

$0110_2 = 6_{10}$

ii) Invert bits to get 1's complement

$1001_2 = -7_{10}$

iii) Add 1 to get 2's complement

$1010_2 = -6_{10}$

# 2's Complement Example (cont.)

- What is the representation of -6 in 2's complement on 8 bits?

| | |
|---|---|
| i) Representation of +6 | $0000\ 0110_2 = 6_{10}$ |
| ii) Invert: | $1111\ 1001_2 = -7_{10}$ |
| iii) Add 1 | $1111\ 1010_2 = -6_{10}$ |

- What is the representation of -6 in 2's complement on 32 bits?

i) Start from the representation of +6

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_2 = 6_{10}$

ii) Invert bits to get 1's complement

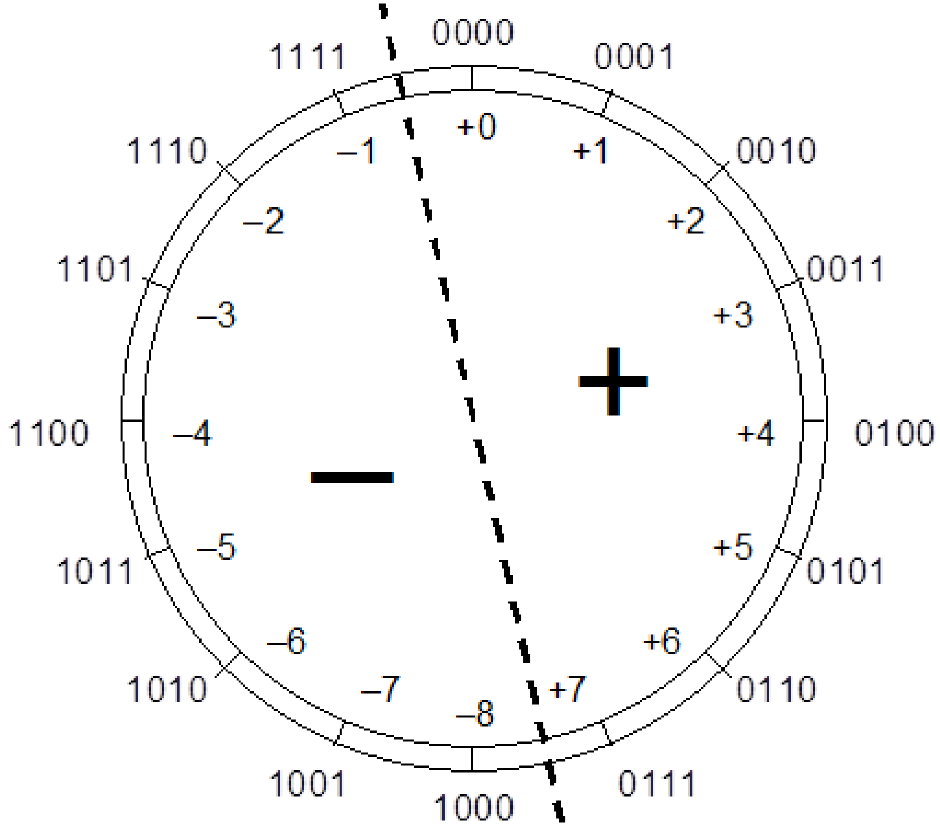
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1001_2 = -7_{10}$

iii) Add 1 to get 2's complement

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_2 = -6_{10}$

# Signed Binary Integers

- **Given k bits**
- Min = $100\ldots..00_2 = -2^{k-1}{}_{10}$
- Max = $011\ldots..11_2 = 2^{k-1} - 1_{10}$
- **Representable range is** $[-2^{k-1}, 2^{k-1} - 1]$



# Largest and Smallest Signed Integer

- Example: what is largest and smallest signed integer represented by 8 bits and 16 bits
- **largest**

$0111\ 1111_2 = 2^7 - 1 = 127$
$0111\ 1111\ 1111\ 1111_2 = 2^{15} - 1 = 32768 - 1 = 32767$

- **smallest**

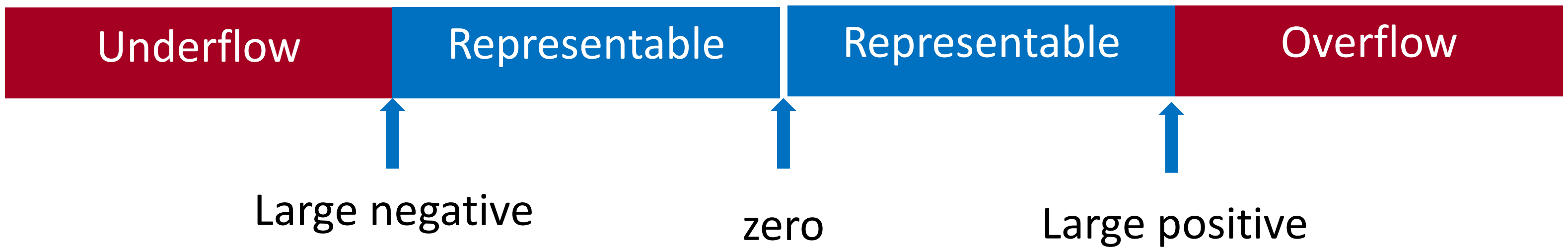
$1000\ 0000_2$
Invert and add 1: $0111\ 1111_2 + 1 = 1000\ 0000_2 = 2^7 = 128$
=> - 128
$1000\ 0000\ 0000\ 0000_2$
Invert – add 1: $0111\ 1111\ 1111\ 1111_2 + 1 = 2^{15} = 32768$
=> - 32768

# Representable Range

- **All integers in range [smallest, largest] can be represented**
- [smallest, largest] is the **representable range**

- Largest integer represented by a 32 bit word
  - 0111 1111 1111 1111 1111 1111 1111 $1111_2$ = $(2^{31} - 1)_{10}$ = $2{,}147{,}483{,}647_{10}$
- Smallest integer represented by a 32 bit word
  - 1000 0000 0000 0000 0000 0000 0000 $0000_2$ = $-2^{31}{}_{10}$ = $-2{,}147{,}483{,}648_{10}$

# Overflow and Underflow in Signed Integer




- **Given the number of bits used in representing a signed integer**
  - **Overflow** (signed integer)
    - The value is bigger than the largest integer that can be represented
  - **Underflow** (signed integer)
    - The value is smaller than the smallest integer that can be represented

# Signed vs. Unsigned Integers

- **Signed**
  - **negative** or **non-negative** integers, e.g. `int` in C/C++
- **Unsigned**
  - **non-negative** integers, e.g. `unsigned int` in C/C++

- **Ranges for signed and unsigned numbers**
  - 32 bit words `signed`:
    - from

      $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = (2^{31} - 1)_{10} = 2{,}147{,}483{,}647_{10}$
    - to

      $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2^{31}{}_{10} = -2{,}147{,}483{,}648_{10}$
  - 32 bit words `unsigned`:
    - from

      $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$
    - to

      $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = (2^{32} - 1)_{10} = 4{,}294{,}967{,}295_{10}$

# Sign Extension

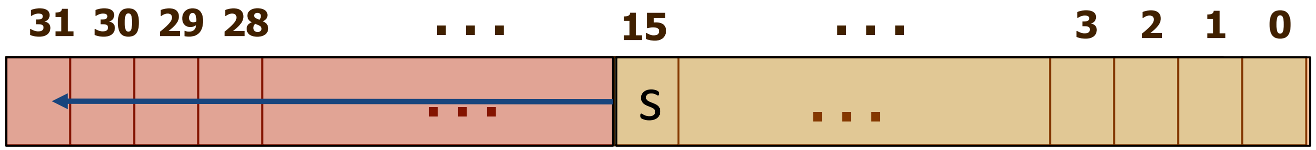
- **Consider using a cast in C/C++ on a 32 bit machine**

```
int i;  /* signed integer represented on 32 bits */
char a; /* Character represented on 8 bits */
i = (int)a; /* What are the values of upper 24 bits in i? */
```

- **Similar things happen in hardware when an instruction loads a 16 bit number into a 32 bit register (hardware variable)**
  - Bits 0~15 of the register will contain the 16-bit value
  - What should be put in the remaining 16 bits (16~31) of the register?

# Sign Extension

- **Sign extension is a way to extend signed integer to more bits**
  - Check the sign bit, and extend it
  - If sign is 0 then fill with 0, If sign is 1 then fill with 1




- **Example: load a 16-bit signed value to a 32-bit register**
  - 2 (16 bits -> 32 bits):

0000 0000 0000 0010 -> 0000 0000 0000 0000 0000 0000 0000 0010

  - -2 (16 bits -> 32 bits):

1111 1111 1111 1110 -> 1111 1111 1111 1111 1111 1111 1111 1110

- **Does sign extension preserve the same value?**

# Zero Extension

- **Zero extension fills missing bits with 0**
- **Example:**
  - Bitwise logical operations (e.g. bitwise AND, bitwise OR)
  - Casting unsigned numbers to larger width

# FLOATING POINT NUMBERS

# Why Floating Point?

- Representation for non-integral numbers
  - **Numbers with fractions** (called real numbers in mathematics)
    - e.g. 3.1416
  - **Very small numbers**
    - e.g., 0.00000000001
  - **Very large numbers**
    - e.g., $1.23456 \times 10^{10}$ (a number a 32-bit integer can't represent)
- In decimal representation, we have **decimal point**
- In binary representation, we call it **binary point**

$$101.11_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2})_{10} = 5.75_{10}$$

# Scientific Notation & Normalized Scientific Notation

- **Scientific notation**
  - A single digit to the left of the decimal point
  - e.g. $1.23 \times 10^{-3}$, $0.5 \times 10^{5}$
- **Normalized scientific notation**
  - Scientific notation with no leading 0's
  - e.g. $1.23 \times 10^{-3}$, $5.0 \times 10^{4}$

- **Binary numbers can also be represented in scientific notation**
- **All normalized binary numbers always start with a 1**

$$1.xxx \dots xxx_2 \times 2^{yyy \dots yyy_2}$$

- **Example** $5.75 = 101.11_2 = 1.0111_2 \times 2^{10_2}$

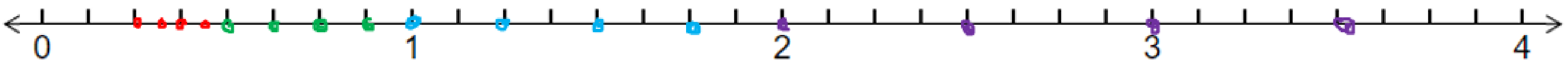
- **Such numbers are called floating point in computer arithmetic**
- **Because the binary point is not fixed in the representation**

# Distribution of Floating Point Numbers

- $\underbrace{1.xxx \ldots xxx_2}_{\text{significand or mantissa}} \times \underbrace{2^{yyy \ldots yyy_2}}_{\text{exponent}}$

- Example: 2-bit significand, exponent = {-2, -1, 0, 1}

| e= -2 | e = -1 | e = 0 | e = 1 |
|---|---|---|---|
| 1.00 X 2^(-2)=4/16 | 1.00 X 2^(-1) = 8/16 | 1.00 X 2^0 = 16/16 | 1.00 X 2^1 = 32/16 |
| 1.01 X 2^(-2)=5/16 | 1.01 X 2^(-1) = 10/16 | 1.01 X 2^0 = 20/16 | 1.01 X 2^1 = 40/16 |
| 1.10 X 2^(-2)=6/16 | 1.10 X 2^(-1) = 12/16 | 1.10 X 2^0 = 24/16 | 1.10 X 2^1= 48/16 |
| 1.11 X 2^(-2)=7/16 | 1.11 X 2^(-1) = 14/16 | 1.11 X 2^0 = 28/16 | 1.11 X 2^1 = 56/16 |




- Floating point representation is approximate arithmetic
- Finite range, limited precision

# Single-Precision

- Single-precision floating point uses **32-bit** sign-and-magnitude representation
- 8 bits for exponent, 23 bits for significand

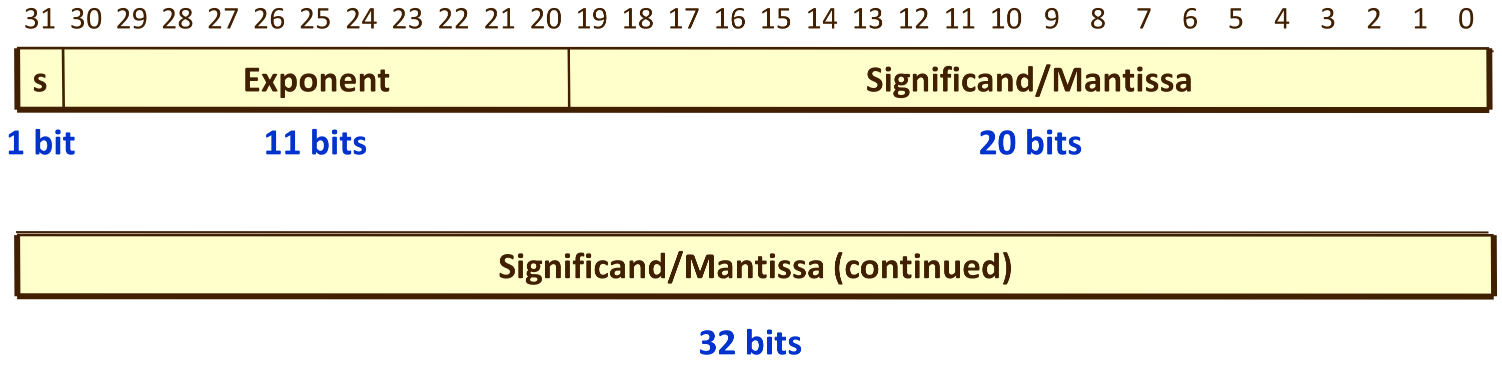
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| s | exponent | Significand/Mantissa |
| --- | --- | --- |
| 1 bit | 8 bits | 23 bits |

- **Interpretation**
  - Roughly gives 7 decimal digits in precision
  - Exponent scale of about $10^{-38}$ to $10^{+38}$

# Double-Precision

- Double-precision floating-point uses 64 bits
- 11 bits for exponent, 52 bits for significand




- **Interpretation**
  - Provides precision of about 16 decimals
  - Exponent scale from $10^{-308}$ to $10^{+308}$

# IEEE754 Floating Point Standard

- **Defined by IEEE in 1985**
- **Developed in response to divergence of representations**
  - Portability issues for scientific code
- **Now almost universally adopted**
- **Two representations**
  - Single precision (32-bit)
  - Double precision (64-bit)

# IEEE754 Floating-Point Format

| | single: 8 bits<br>double: 11 bits | single: 23 bits<br>double: 52 bits |
|---|---|---|
| **S** | **Exponent** | **Significand** |

$$x = (-1)^S \times (1 + Significand) \times 2^{Exponent-Bias}$$

- **Sign bit 0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative**
- **Biased exponent**
  - Exponent field saves actual exponent + Bias, which ensures to be an unsigned value
  - Single: Bias = 127; Double: Bias = 1023
- **Significand with implicit 1**
  - Significand is normalized to be 1.xxxxx
  - There's always a leading 1 before binary point
  - The leading 1 is implicit (hidden bit) and not saved in significand field
- **Special cases are used to represent denormalized significand, NaN, etc.**

# IEEE754 Example

- Give the IEEE754 representation of $-0.75_{10}$ in single & double precisions
- Answer
  - Scientific notation: $-0.75 = -0.11_2 \times 2^0$
  - Normalized scientific notation: $-1.1_2 \times 2^{-1}$
  - Sign = 1 (negative), exponent = -1

  - Single precision:

    S = 1, Significand = 100...00 (23 bits)

    Biased exponent = -1+127 = 01111110

    1 01111110 10000000000000000000000

  Double precision:

    S = 1, significand = 100...00 (52 bits)

    Biased exponent = -1+1023 = 01111111110

    1 01111111110 1000000000000000 ....000

# IEEE754 Example 2

- What decimal number is represented by this word (single precision)?

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

- Answer:

$$
\begin{aligned}
& (-1)^s \times (1 + Significand) \times 2^{(Exponent - Bias)} \\
&= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\
&= -1 \times 1.25 \times 2^2 \\
&= -1.25 \times 4 \\
&= -5.0
\end{aligned}
$$

# Why Implicit 1 and Biased Exponent?

- **Implicit 1** **packs even more bits into the significand**
  - The number is actually 24 bits long in single precision (implied 1 and 23-bit fraction), and 53 bits long in double precision (1 + 52)
- **Increase the precision of your numbers**

- **Biased exponent** **is for faster comparisons (for sorting, etc.), allow integer comparisons of floating point numbers**
  - Unbiased exponent

| | Sign | Exponent | Fraction |
|---|---|---|---|
| 1/2 | 0 | 1111 1111 | 000 0000 0000 0000 0000 0000 |
| 2 | 0 | 0000 0001 | 000 0000 0000 0000 0000 0000 |

  - Biased exponent

| | Sign | Exponent | Fraction |
|---|---|---|---|
| 1/2 | 0 | 0111 1110 | 000 0000 0000 0000 0000 0000 |
| 2 | 0 | 1000 0000 | 000 0000 0000 0000 0000 0000 |

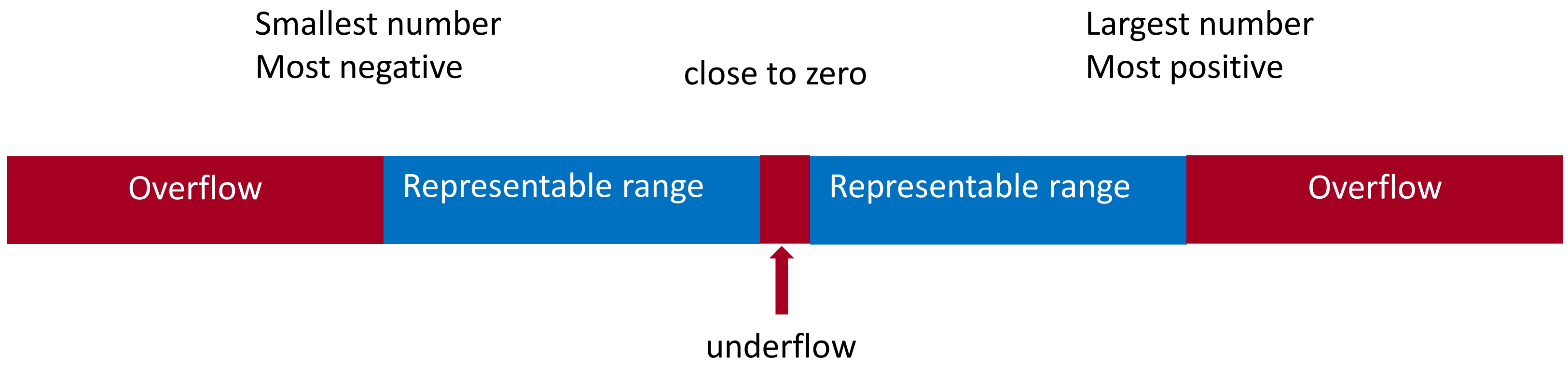
# Single-Precision Range (for Normalized Numbers)

- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value**
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

# Double-Precision Range (for Normalized Numbers)

- **Exponents 0000…00 and 1111…11 reserved**
- **Smallest value**
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

# Overflow and Underflow in Floating Point




- **Overflow**
  - A positive exponent becomes too large to fit in the exponent field
- **Underflow**
  - A negative exponent becomes too large to fit in the exponent field

# IEEE 754 Standard for Floating Point Arithmetic

## Single precision:

Denormalized | Normalized | Special Cases

| Significand \ Exponent | 0 | **1 - 254** | 255 |
|---|---|---|---|
| 0 | 0 | $(-1)^S \times (1.F) \times (2)^{E-127}$ | $(-1)^S \times (\infty)$ |
| ≠ 0 | $(-1)^S \times (0.F) \times (2)^{-126}$ | | non-numbers e.g. 0/0 , $\sqrt{-1}$ |

## Double precision:

| Significand \ Exponent | 0 | **1 - 2046** | 2047 |
|---|---|---|---|
| 0 | 0 | $(-1)^S \times (1.F) \times (2)^{E-1023}$ | $(-1)^S \times (\infty)$ |
| ≠ 0 | $(-1)^S \times (0.F) \times (2)^{-1022}$ | | non-numbers e.g. 0/0 , $\sqrt{-1}$ |

# Denormalized Numbers and Special Cases

| Exponent | Significand | Usage |
|---|---|---|
| 111.....11 | 000.....00 | ❑ ±Infinity<br>❑ Can be used in subsequent calculations, avoiding need for overflow check |
| 111.....11 | ≠ 000...00 | ❑ Not-a-Number (NaN)<br>❑ Indicates illegal or undefined result<br>❑ e.g., 0.0 / 0.0<br>❑ Can be used in subsequent calculations |
| 000.....00 | 000.....00 | ❑ $\pm 0.0$ |
| 000.....00 | ≠ 000...00 | ❑ Hidden bit is 0 (no implicit 1)<br>❑ $x = (-1)^S \times (0 + Significand) \times 2^{-126}$<br>❑ Even smaller than normal number |

# IEEE754 Examples

| | | | |
|---|---|---|---|
| 0 | 00000000 | 00000000000000000000000 = | 0 |
| 1 | 00000000 | 00000000000000000000000 = | -0 |
| 0 | 11111111 | 00000000000000000000000 = | + infinity |
| 1 | 11111111 | 00000000000000000000000 = | - infinity |
| 0 | 11111111 | 01001100010001000001000 = | **NaN (Not a Number)** |
| 1 | 11111111 | 01001100010001000001000 = | **NaN** |
| 0 | 10000000 | 00000000000000000000000 = | $+(1.0_2)\times(2)^{128-127}=2$ |
| 0 | 10000001 | 10100000000000000000000 = | $+(1.101_2)\times(2)^{129-127}=6.5$ |
| 1 | 10000001 | 10100000000000000000000 = | $-(1.101_2)\times(2)^{129-127}=-6.5$ |
| 0 | 00000001 | 00000000000000000000000 = | $+(1.0_2)\times(2)^{1-127}=(2)^{-126}$ |
| 0 | 00000000 | 10000000000000000000000 = | $+(0.1_2)\times(2)^{-126}=(2)^{-127}$ |
| 0 | 00000000 | 00000000000000000000001 = | $+(2)^{-23}\times(2)^{-126}=(2)^{-149}$ |

# CHARACTERS

# Communicating with People

- Characters are unsigned bytes e.g., in C++ Char
- Usually follow the ASCII standard
- Uses 8 bits unsigned to represent a character

| Bits | | | | $b_7$ $b_6$ $b_5$ → | 0 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1 0 0 | 1 0 1 | 1 1 0 | 1 1 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b_4$ ↓ | $b_3$ ↓ | $b_2$ ↓ | $b_1$ ↓ | Column → Row ↓ | 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 | FC | , | < | 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 |

# Exercise

## What does the following 32 bit pattern represent: 0x32363131

- If it were a 2’s complement integer
- If it were an unsigned number
- A sequence of ASCII encoded bytes: 2611
  - Checking the ASCII table gives 0x32 = code for character ‘2’, 0x36 = code for character ‘6’, 0x31 = code for character ‘1’, 0x31 = code for character ‘1’
- A 32 bit IEEE 754 floating point number
  - s= 0, E = 01100100, S = 01101100011000100110001
  - This is a normalized number so E is biased.

# Exercises

- **Consider building a floating point number system like the IEEE754 standard on 8 bit only, with 3 bits being reserved for the exponent.**
  - What is the value of the bias? **3**
  - What is the representation of 0? **0 000 0000**
  - What is the representation of -4?

    **$-4 = -1.0 \times 2^2$**

    **S=1, F= 0 and the biased exponent must be**

    **E – 3 = 2 or E = +5**

    **So - 4 = 1 101 0000**
  - What is the next negative value representable after – 4? **1 101 0001 = - 4.25**
  - What does the byte 1 111 1011 represent? **- NAN**
  - What is the representation of -∞? **1 111 0000**

# Concluding Remarks

- **2's complement representation for signed numbers**
  - Smallest and largest, representable range
- **Floating-point numbers**
  - Representation follows closely the **scientific notation**
  - Almost all computers, including MIPS, follow **IEEE 754 standard**
- **Single-precision floating-point representation takes 32 bits**
- **Double-precision floating-point representation takes 64 bits**
- **Overflow and underflow in signed integer and floating number**