# Computer Structure

## *Unit 6. Memory maps. Real Numbers Operations*

Escuela Politécnica Superior - UAM

# Outline

**6.1.- Interface between the processor and the peripherals: memory maps (book 8.1 and 8.5)**

6.1.1.- Aligned and not aligned blocks

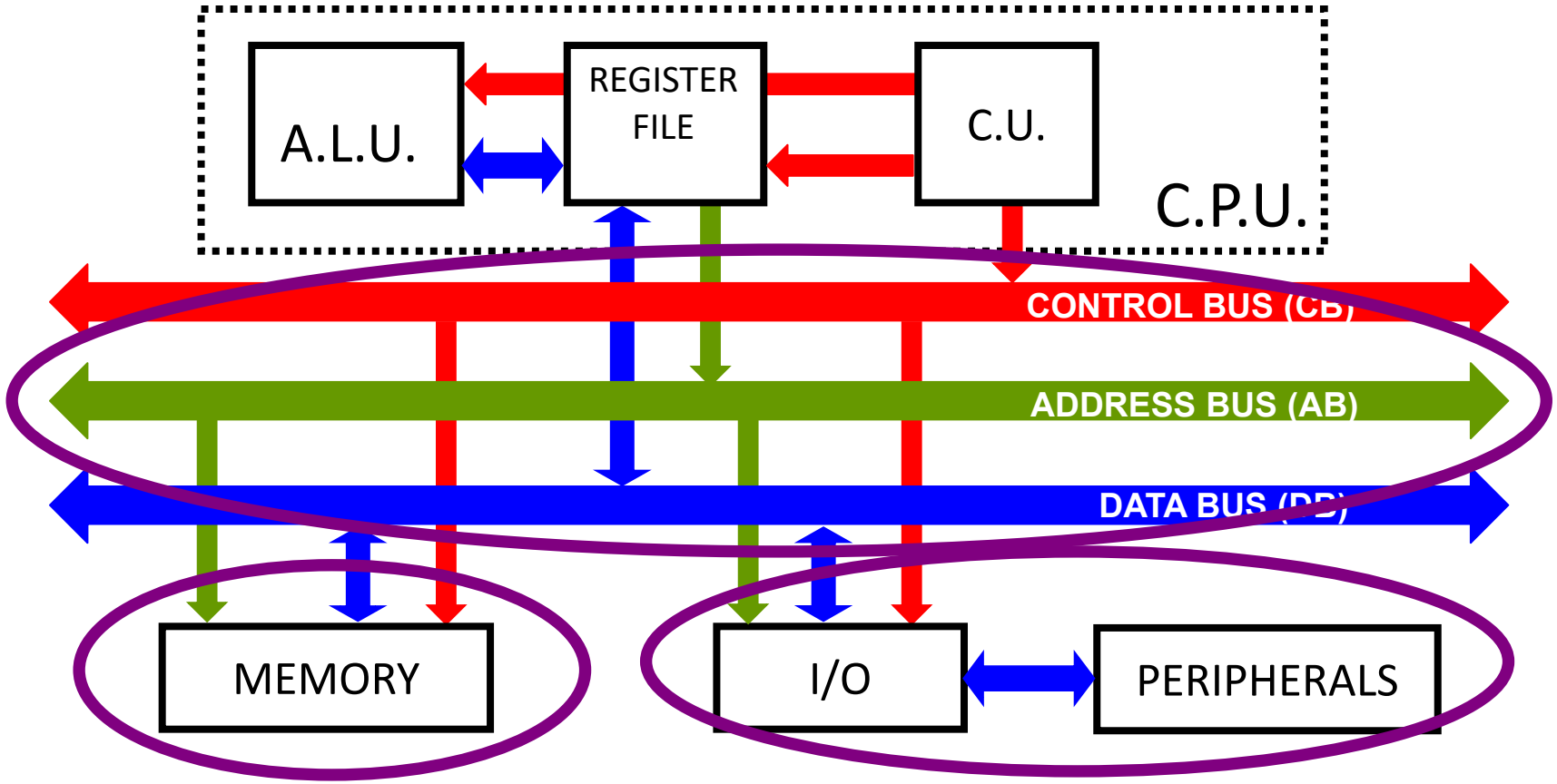
6.1.2.- Memory maps design

**6.2.- Real numbers operations**

6.2.1.- Fixed and floating point representation

6.2.2.- Addition, subtraction and multiplication with real numbers

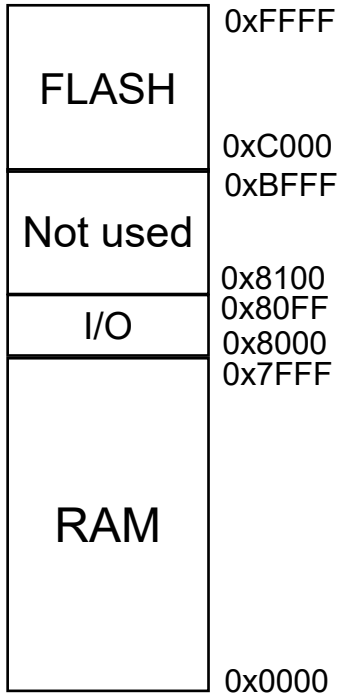
# Von NEUMANN architecture



# Interface between I/O devices and the processor

**How does the microprocessor access the information of the I/O devices?**

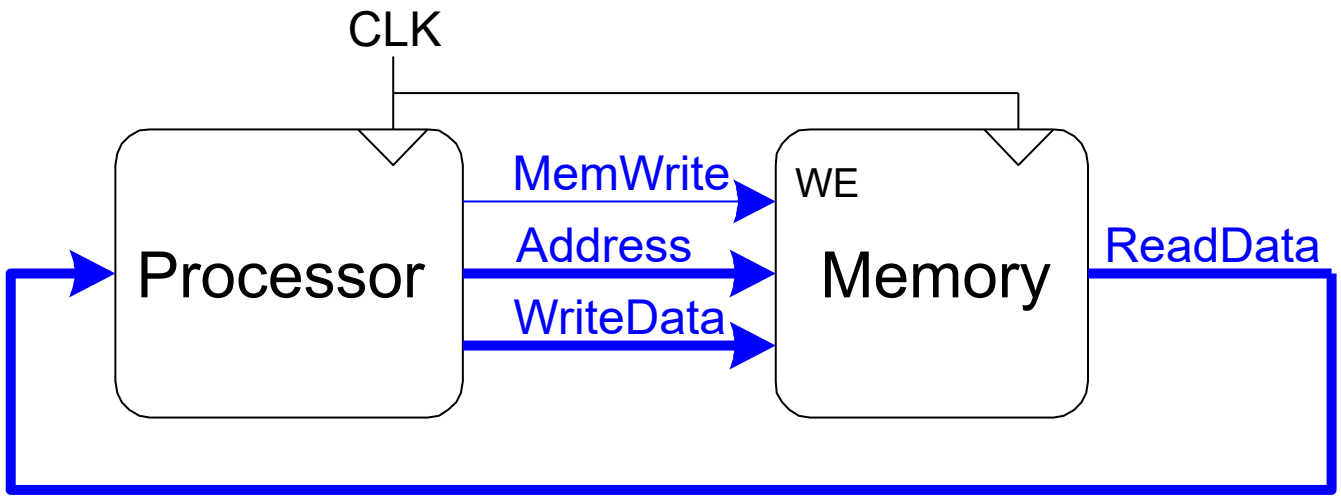
- ✓ The processor only communication is through the bus. For the processor there are only different addresses, not devices.
- ✓ The I/O devices are mapped as if they were memory devices
- ✓ Reading some addresses the processor reads the I/O devices, and writing in some others the processor sends info to the I/O devices




Memory map
(64 kbytes)

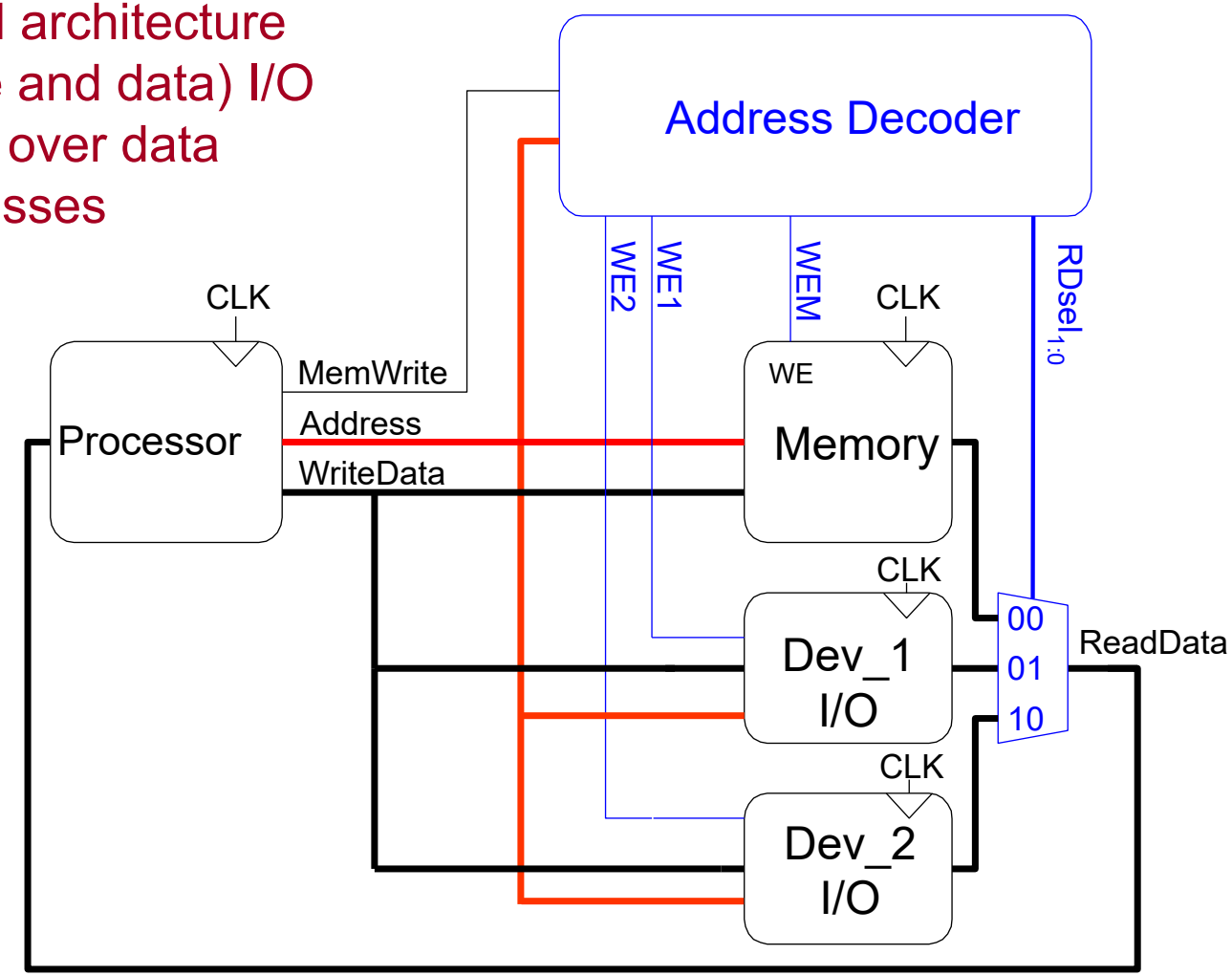
# Memory map: without I/O

Assuming Von Neumann architecture (code and data in the same logic memory, even if they are different chips)



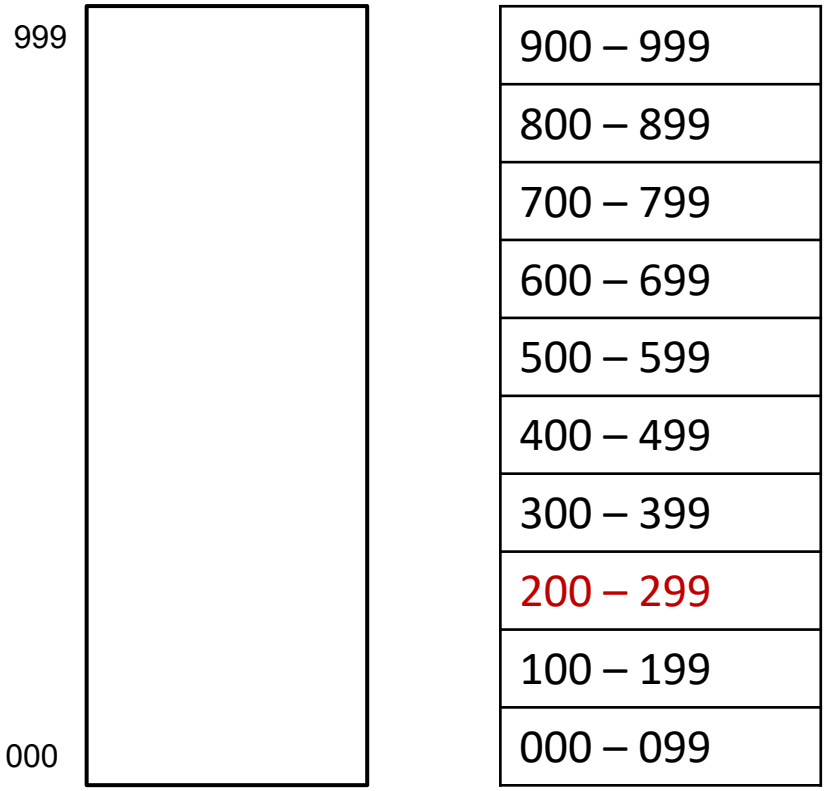
# Memory map: with I/O (address decoder)

If it is Harvard architecture (separate code and data) I/O is mapped over data addresses



# Aligned and not aligned blocks (in decimal)

- Each set of addresses representing a device is called “block”. Usually, the size of each block is a power of two (in decimal it would be a power of ten).
- A **block is aligned** if all the bits/digits not used for internal addressing are constant. The block must be a size of $2^X$ ($10^X$).



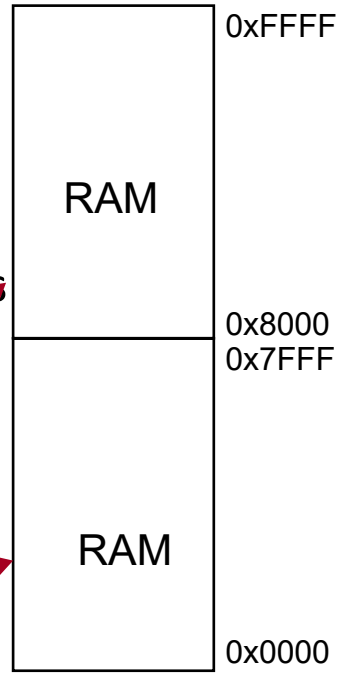

| |
|---|
| 900 – 999 |
| 800 – 899 |
| 700 – 799 |
| 600 – 699 |
| 500 – 599 |
| 400 – 499 |
| 300 – 399 |
| 200 – 299 |
| 100 – 199 |
| 000 – 099 |

- ✓ Example: map of 1000 ($10^3$) addresses, from 000 to 999.
- ✓ Where to place a 100 addresses block?
- ✓ The block is aligned if all its addresses start with the same digit/s.
- ✓ Example: 200 to 299 (2XX)
- ✓ The block goes from 00 to 99 which is “mapped” into 200 to 299.

# Aligned and not aligned blocks

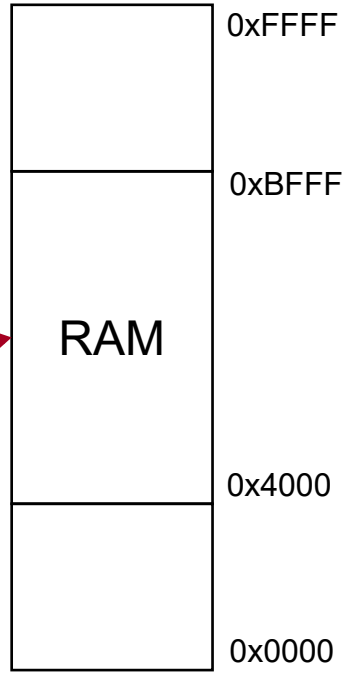
- Each set of addresses representing a device is called “block”. Usually, the size of each block is a power of two.
- A **block is aligned** if all the bits not used for internal addressing are constant. The decoding is then minimized.

- ✓ Example: Microprocessor byte-addressable of 64 kBytes total memory map. 16 bits for the addresses.
- ✓ 3 Blocks: RAM of 32 kB, Flash of 16 kB and I/O of 256 Bytes.
- ✓ The RAM block of 32 kB uses 15 bits for internal addressing: it is aligned if the remaining bit is constant:
  - ✓ 1XXX XXXX XXXX XXXX ($8000_{16}$ to $FFFF_{16}$): $A_{15} = 1$
  - ✓ 0XXX XXXX XXXX XXXX ($0000_{16}$ to $7FFF_{16}$): $A_{15} = 0$



# Aligned and not aligned blocks

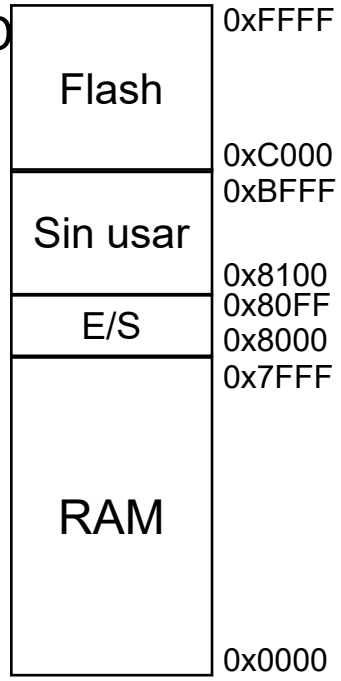
- If the block **is not aligned**, the decoding is more complex:
  - ✓ ($4000_{16}$ to $BFFF_{16}$): $A_{15}A_{14}$=01 or $A_{15}A_{14}$=10
  - ✓ In fact, it is using two 16 kB (14 bits) aligned blocks:
  - ✓ 01XX XXXX XXXX XXXX ($4000_{16}$ a $7FFF_{16}$)
  - ✓ 10XX XXXX XXXX XXXX ($8000_{16}$ a $BFFF_{16}$)




- The address decoder circuit (combinational), generates the enable signals for each block (CE, *Chip Enable* and/or WE, *Write Enable*) and the control signals of the multiplexer that choses among the data output of each block.

# Complete / incomplete mapping

- 3 Blocks: RAM of 32 kB (15 bits), Flash of 16 kB (14 bits) and I/O of 256 Bytes (8 bits).
  - ✓ RAM: ($0000_{16}$ to $7FFF_{16}$) =>0XXX XXXX XXXX XXXX
  - ✓ Flash: ($C000_{16}$ to $FFFF_{16}$) =>11XX XXXX XXXX XXXX
  - ✓ E/S: ($8000_{16}$ to $80FF_{16}$) =>1000 0000 XXXX XXXX
  - ✓ Not used (empty): ($8100_{16}$ to $BFFF_{16}$)
- Decoding:
  - ✓ RAM: $A_{15} = 0$
  - ✓ Flash: $A_{15}A_{14} = 11$
  - ✓ I/O, complete (exhaustive) mapping: $A_{15}A_{14}A_{13}A_{12}\ A_{11}A_{10}A_9A_8 = 1000\ 0000$
  - ✓ I/O, incomplete (partial) mapping: $A_{15}A_{14} = 10$ (enough for distinguishing)

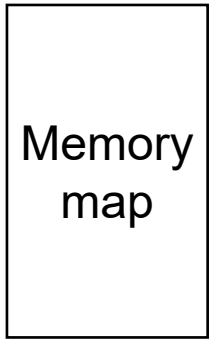



**And what if the processor uses an empty address?** ($0x9000_{16}$)

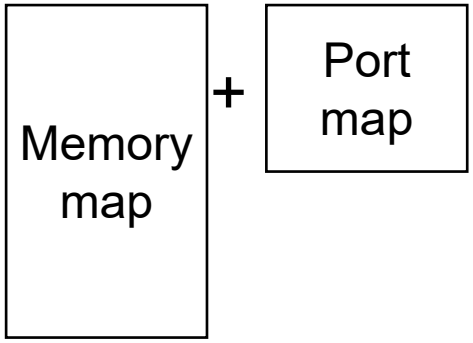
- ✓ In complete mapping nothing is enabled
- ✓ In incomplete mapping other block will be enabled (I/O in this case)

# Memory mapped vs Port mapped

- Most architectures map I/O into the only address map (memory map).
  - ✓ For reading I/O, use **load**
  - ✓ For writting I/O, use **store**




- Other architectures (e.g. IA-32, know as x86) use other address map (port map).
  - ✓ There is an extra pin for distinguishing between memory or I/O (M/$\overline{IO}$)
  - ✓ **load** and **store** use memory (M/$\overline{IO}$ = '1')
  - ✓ Other instructions (e.g. **in** and **out**) use the ports (M/$\overline{IO}$ = '0')




**P6.2, P6.4, P6.8**

# Outline

6.1.- Interface between the processor and the peripherals: memory maps

6.1.1.- Aligned and not aligned blocks

6.1.2.- Memory maps design

**6.2.- Real numbers operations**

6.2.1.- Fixed and floating point representation

6.2.2.- Addition, subtraction and multiplication with real numbers

# Number systems

- Representing numbers in binary:
  - **Integer positive numbers**
    - *Unsigned binary*
  - **Integer (positive and negative) numbers**
    - T*wo's complement*
    - *Sign/magnitude*
  - **Fractional numbers**
    - *Fixed-point*
    - *Floating-point*. IEEE-754:
      - Simple precision (32 bits)
      - Double precision (64 bits)

# Fixed point numbers

- Fixed point representation of 6.75 with 4 bits for the integer part and 4 bit for the fractional part:

01101100

0 1 1 0.1 1 0 0

8 4 2 1 0.5 0.25 0.125 0.0625

$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

- The point is not represented, it is implicit.
- The number of bits for each part must be known for both those generating the number and those reading it. Once set, it is fixed.

# Fixed point sign

- Same as integers:
  - Sign/magnitude
  - Two's complement
- Representing $-7.5_{10}$ using 8-bits, 4 for the integer part and 4 for the fractional part.
  - Sign/magnitude: 1111.1000
  - Two's complement:

| | |
|---|---|
| 1. +7.5: | 0111.1000 |
| 2. Invert bits: | 1000.0111 |
| 3. Add 1: | + 1 |
| | 1000.1000 |

# Floating point numbers

- The point is set just to the right of the '1' most significant bit.
- Similar to scientific notation in decimal.
- For example, $273_{10}$ in scientific notation:

$$\mathbf{273 = 2.73 \times 10^2}$$

- A number is represented as:

$$\mathbf{\pm M \times B^E}$$

where,

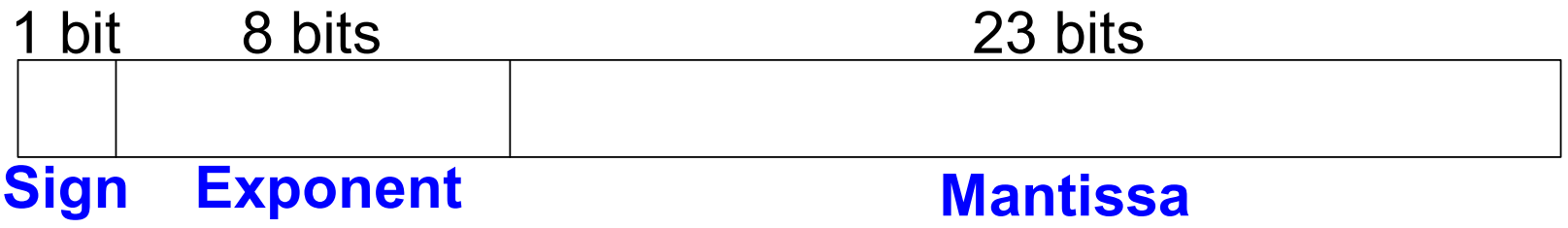
  - **M** = mantissa (significand) **B** = base **E** = exponent
  - In the example, M = 2.73; B = 10; E = +2

# Floating point IEEE-754, precision

- *Single-precision*:
  - 32-bits in total
  - 1 sign bit, 8 for the exponent and 23 for the significand
  - Exponent bias = 127
- *Double-precision*:
  - 64-bits in total
  - 1 sign bit, 11 for the exponent and 52 for the significand
  - Exponent bias = 1023

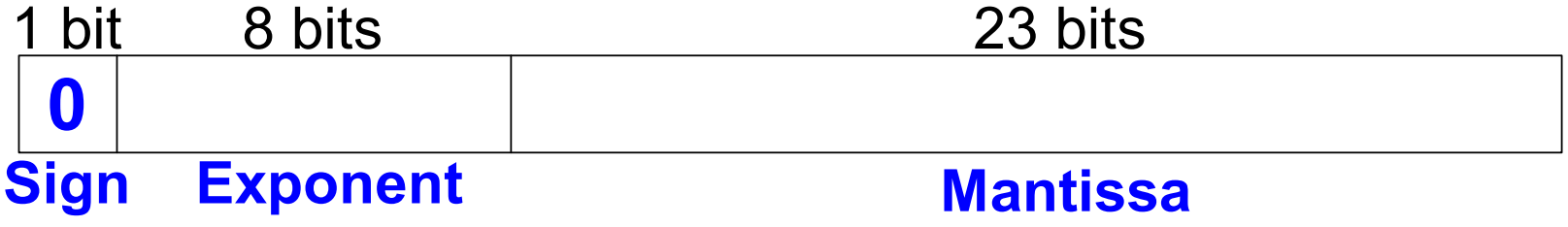
**Note:** The exponent is represented in unsigned binary, but with a bias (offset). That is the way of representing negative exponents, with a bias equal to $2^{(exp-1)}-1$. For example, in *single-precision*, the bias is $2^{(8-1)}-1 = 127$. Therefore, the exponent "00000000"(0) represents -127. And the exponent "11111111"(255) represents +128.

# Floating point numbers (IEEE-754)




**Example:** represent $228_{10}$ in IEEE-754 (32bits)

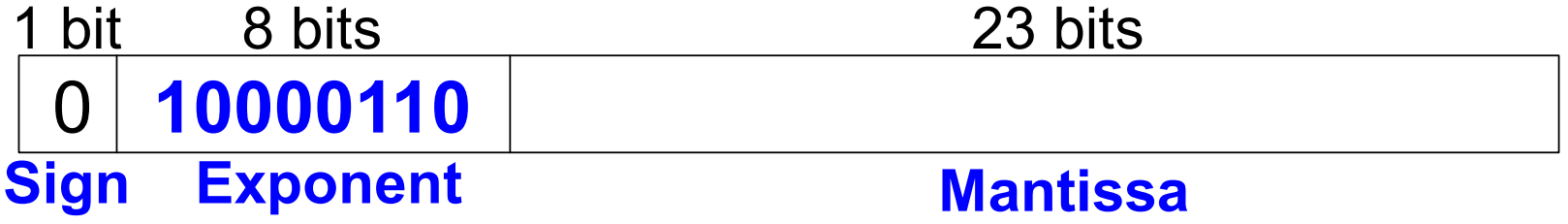
- Convert from decimal to binary (scientific notation):
  - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Fill each field of the IEEE-754 format:
  - Sign bit (0 for positive)



# Floating point numbers (IEEE-754)

**Example:** represent $228_{10}$ in IEEE-754 (32bits)

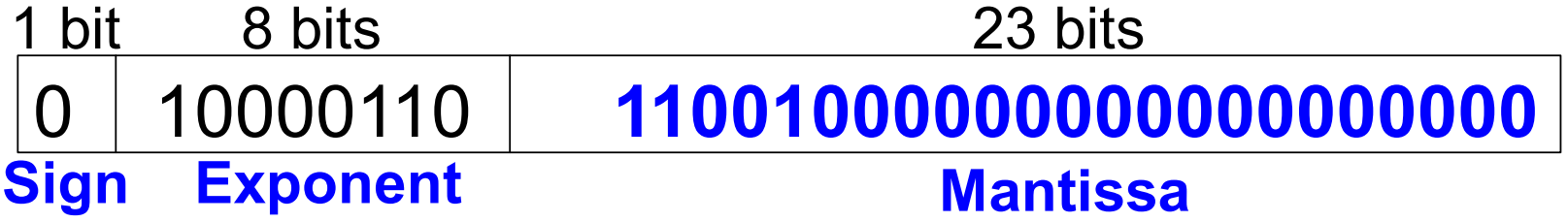
- Convert from decimal to binary:
  - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Fill each field of the IEEE-754 format:
  - Exponent has a bias => bias + exponent
  - Bias => 127 = $01111111_2$
  - The exponent, 7, is stored as: 127+7 = 134 = $10000110_2$



# Floating point numbers (IEEE-754)

**Example:** represent $228_{10}$ in IEEE-754 (32bits)

- Convert from decimal to binary:
  - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Fill each field of the IEEE-754 format:
  - The first bit of the significand is always '1'
  - That bit is not stored (would be redundant).
  - The next 23 bits are stored.

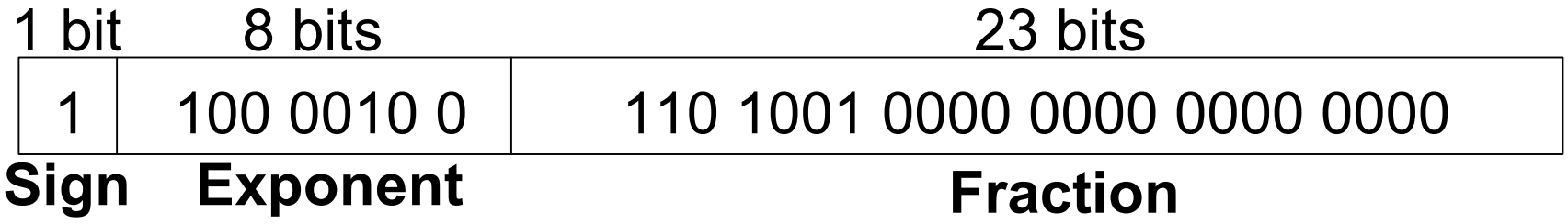



- In hexadecimal: 0x43640000

# Floating point numbers (IEEE-754)

**Example:** represent $-58.25_{10}$ in IEEE-754 (32bits)

- Convert from decimal to binary (scientific notation) without sign:
  - $58.25_{10} = 111010.01_2 = 1.1101001 \times 2^5$
- Fill in the three fields:
  - Sign bit: 1 (negative)
  - Exponent bias in 8 bits: $(127 + 5) = 132 = 10000100_2$
  - 23 bits for the significand: 110 1001 0000 0000 0000 0000




- In hexadecimal: 0xC2690000

# IEEE-754, especial cases

- IEEE 754 standard has especial cases for some numbers. For instance number 0 has no implicit '1' in the significand.

| Number | Sign | Exponent | Significand |
|---|---|---|---|
| 0 | X | 00000000 | 00000000000000000000000 |
| ∞ | 0 | 11111111 | 00000000000000000000000 |
| - ∞ | 1 | 11111111 | 00000000000000000000000 |
| NaN | X | 11111111 | Different from zero |

NaN (*Not a Number*) is used for numbers that do not exist, such as √-1 or log(-5).

P6.11

# Addition in floating point

1. Extract fields (exponents and significands)
2. Add a ‘1’ in the left to the significand
3. Compare exponents
4. Shift the significand of the small exponent (if needed)
5. Add significands
6. Normalize significand and adjust exponent (if needed)
7. Round result
8. Compose the result (sign, exponent and significand)

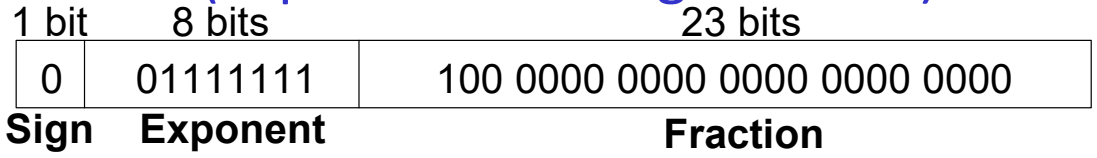
# Addition in floating point: example

Example. Add the following floating point numbers:

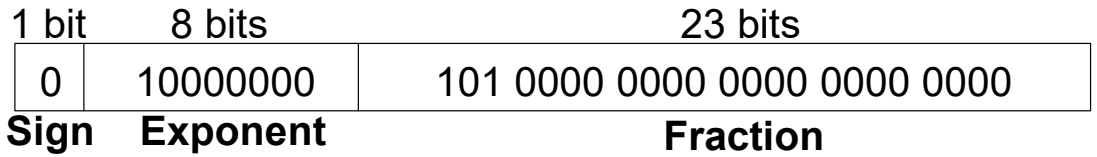
N1 = 0x3FC00000 + N2 = 0x40500000

1. Extract fields (exponents and significands)

N1:




N2:




For N1: S = 0, E = 127, F = .1

For N2: S = 0, E = 128, F = .101

2. Add a '1' in the left to the significand

N1: 1.1

N2: 1.101

# Addition in floating point: example

3. Compare exponents

   127 – 128 = -1, so N1 is shifted for the right 1 position

4. Shift the significand of the small exponent (if needed)

   Shift the significand of N1: 1.1 >> 1 = 0.11 ($\times 2^1$)

5. Add significands

$$\begin{array}{r} 0.11 \times 2^1 \\ \underline{+1.101 \times 2^1} \\ 10.011 \times 2^1 \end{array}$$

6. Normalize significand and adjust exponent (if needed)

   $10.011 \times 2^1 = 1.0011 \times 2^2$

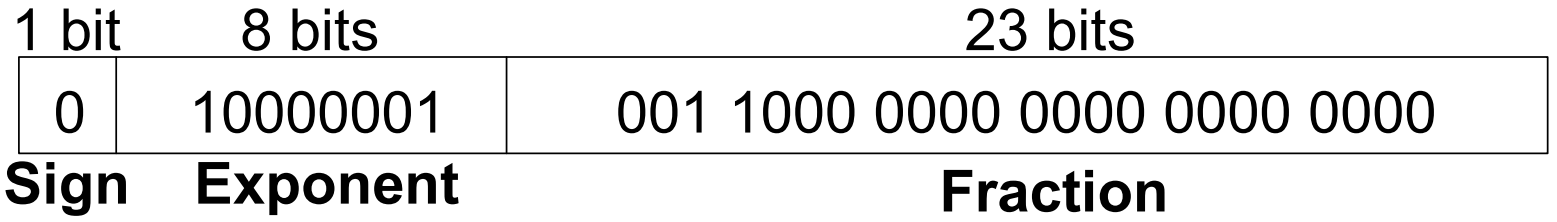
# Addition in floating point: example

7. Round result

   Not needed (significand fits in 23 bits)

8. Compose the result (sign, exponent and significand)

   S = 0, E = 2 + 127 = 129 = $10000001_2$, M = 001100..




In hexadecimal: 0x40980000

0x3FC00000 + 0x40500000 = 0x40980000

# Multiplication in floating point

1. Extract fields (exponents and significands)
2. Add a ‘1’ in the left to the significands
3. Add unbiased exponents
4. Multiply significands
5. Normalize significand and adjust exponent (if needed)
6. Round result
7. Renormalize the significand after rounding (if needed)
8. Compose the result (sign, exponent and significand)

# Multiplication in floating point: example

Example. Multiply the following floating point numbers:

N1 = 0x3FC00000 x N2 = 0xC0500000

1. Extract fields (exponents and significands)

N1:

| 1 bit | 8 bits | 23 bits |
|---|---|---|
| 0 | 01111111 | 100 0000 0000 0000 0000 0000 |
| **Sign** | **Exponent** | **Fraction** |

N2:

| 1 bit | 8 bits | 23 bits |
|---|---|---|
| 1 | 10000000 | 101 0000 0000 0000 0000 0000 |
| **Sign** | **Exponent** | **Fraction** |

For N1: S = 0, E = 127, F = .1

For N2: S = 1, E = 128, F = .101

2. Add a '1' in the left to the significands

N1: 1.1

N2: 1.101

# Multiplication in floating point: example

3. Add unbiased exponents

   127 => 0; 128 => 1; 1 + 0 = 1 (will be 128 with bias)

4. Multiply significands

$$\begin{array}{r} 1.101 \\ \times\ 1.1 \\ \hline 0.1101 \\ +\ 1.101 \\ \hline 10.0111 \end{array}$$

5. Normalize significand and adjust exponent (if needed)

   $10.0111 \times 2^1 = 1.00111 \times 2^2$

6. Round result

   Not needed (significand fits in 23 bits)

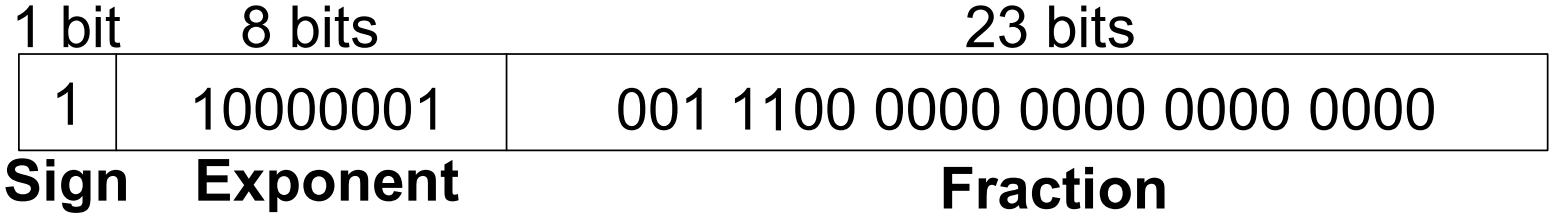
# Multiplication in floating point: example

7. Renormalize the significand after rounding (if needed)

   Not needed (still 1.M)

8. Compose the result (sign, exponent and significand)

$S = S_{N1} \oplus S_{N\,2} = 1$; $E = 2+127 = 129 = 10000001_2$; $M = 0011100..$




In hexadecimal: 0xC09C0000

0x3FC00000 x 0xC0500000 = 0xC09C0000

P6.14, P6.15

# IEEE-754 references

- From IEEE-754 to decimal:
  - http://babbage.cs.qc.cuny.edu/IEEE-754.old/32bit.html
- From decimal to IEEE-754
  - http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html
- Calculator in IEEE-754 format (following class steps):
  - https://docencia.hctlab.com/ieee754/

# *Unit 6. Memory maps. Real Numbers Operations*

Escuela Politécnica Superior - UAM