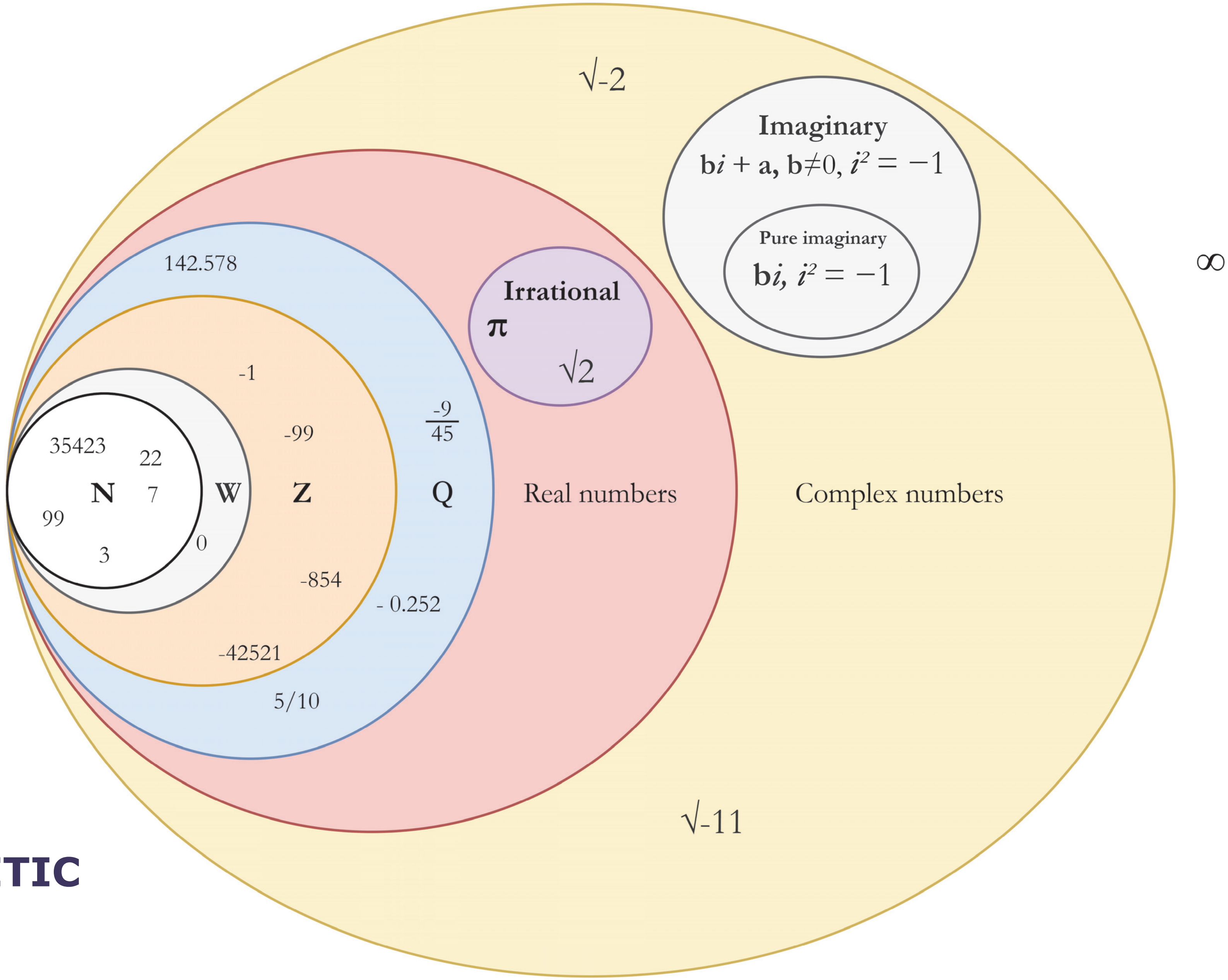
MICROPROCESSOR SYSTEMS (IAS0430)

Department of Computer Systems
Tallinn University of Technology

3.12.2021

Number groups	
N	Natural numbers
W	Whole numbers
Z	Integers
Q	Rational numbers



COMPUTER ARITMETIC NUMBER GROUPS

COMPUTER ARITHMETIC

- **Arithmetic** is a branch of mathematics that studies numbers and the properties of the traditional operations performed on them (addition, subtraction, multiplication, and division).
- Different **numeral systems** can be used to represent numbers.
- The most common one are **positional numeral systems** where the value is found as the weighted sum of numbers.
- General notation – $D = \Sigma\, d_i \cdot r^i$, where $i = -n, \ldots, p-1$
  - $d^{p-1}\ d^{p-2} \ldots d^2\ d^1\ d^0 .\ d^{-1}\ d^{-2} \ldots d^{-n}$
  - r – radix
- *Decimal* numbers – $\Sigma\, d_i \cdot 10^i$, where $i = -n, \ldots, p-1$
  - $173.4 = 1 \cdot 100 + 7 \cdot 10 + 3 \cdot 1 + 4 \cdot 0.1$
  - $173.4 = 1 \cdot 10^2 + 7 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1}$
- *Binary* numbers – $\Sigma\, d_i \cdot 2^i$, where $i = -n, \ldots, p-1$
  - $101.001_2 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 0 \cdot 0.5 + 0 \cdot 0.25 + 1 \cdot 0.125 = 5.125_{10}$

COMPUTER ARITHMETIC

- **Positional numeral systems**
- *Decimal* numbers – $\Sigma\ d_i \cdot 10^i$ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  - $173.4 = 1 \cdot 100 + 7 \cdot 10 + 3 \cdot 1 + 4 \cdot 0.1$
  - $173.4 = 1 \cdot 10^2 + 7 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1}$
- *Binary* numbers – $\Sigma\ d_i \cdot 2^i$ [0, 1]
  - $101.001_2 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 0 \cdot 0.5 + 0 \cdot 0.25 + 1 \cdot 0.125 = 5.125_{10}$
    - MSB - most significant bit / LSB - least significant bit
- *Octal* numbers – $D = d_i \cdot 8^i$ [0, 1, 2, 3, 4, 5, 6, 7]
  - $100011001110_2 = 100\ 011\ 001\ 110_2 = 4316_8$
- *Hexadecimal* numbers – $D = d_i \cdot 16^i$ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F]
  - $11101101110101001_2 = 0001\ 1101\ 1011\ 1010\ 1001_2 = 1DBA9_{16}$
- Integers vs Rational numbers – integer, fixed-point, floating-point
  - $10.1011001011_2 = 010\ .\ 101\ 100\ 101\ 100_2 = 2.5454_2$
  - $10.1011001011_2 = 0010\ .\ 1011\ 0010\ 1100_2 = 2.B2C_{16}$

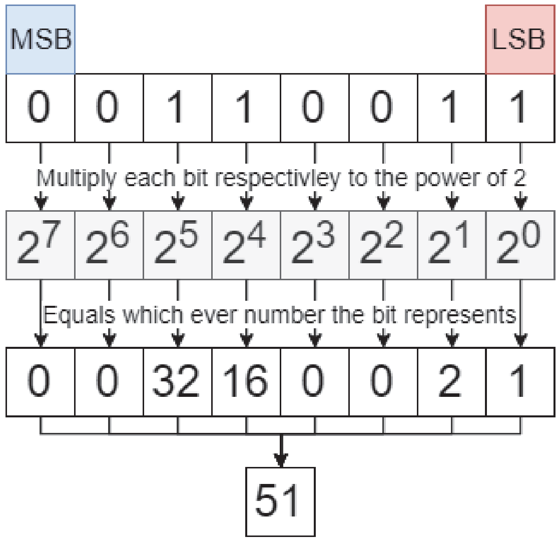
COMPUTER ARITHMETIC

- **Positional numeral systems - conversions**
  - Conversion over decimal-system (manually)
  - Conversion using internal number representation (in computers)
- **Generalized conversion**
  - $d^{p-1}\ d^{p-2}\ \ldots\ d^2\ d^1\ d^0$
  - $D = \Sigma\ d_i \cdot r^i\ \ (i = 0, \ldots, p-1) = d^{p-1} \cdot r^{p-1} + d^{p-2} \cdot r^{p-2} + \ldots + d^2 \cdot r^2 + d^1 \cdot r^1 + d^0 \cdot r^0 =$
  - $(((\ldots ((d^{p-1}) \cdot r + d^{p-2}) \cdot r + \ldots) \cdot r + d^2) \cdot r + d^1) \cdot r + d^0$
  - Recursive division – reminders gives the value of the position
- Examples
  - $F1AC_{16} = (((15) \cdot 16+1) \cdot 16+10) \cdot 16+12 = 61868$
    - 61868 / 16 = 3866, reminder 12
    - 3866 / 16 = 241, reminder 10
    - 241 / 16 = 15, reminder 1
  - $54_{10} = ??_{13}$ (6x9=42?!)
    - 54 / 13 = 4, reminder 2 – $54_{10} = 42_{13}$

COMPUTER ARITHMETIC

- **Whole numbers:**
  - Using one byte, 256 natural numbers can be represented.
  - A combination of 00110011 in binary, written as 00110011_b represents the following:
    - Each of the **1** bits represent one magnitude of the power of 2
    - A **0** represents represent the absence of a magnitude of power of 2
    - The **Least Significate Bit (LSB),** is the bit representing the **lowest magnitude** of power of 2
    - The **Most Significate Bit (MSB),** is the bit representing the **highest magnitude** of power of 2
    - This is a very effective and straight forward method of decimal representation of a natural number in a computer

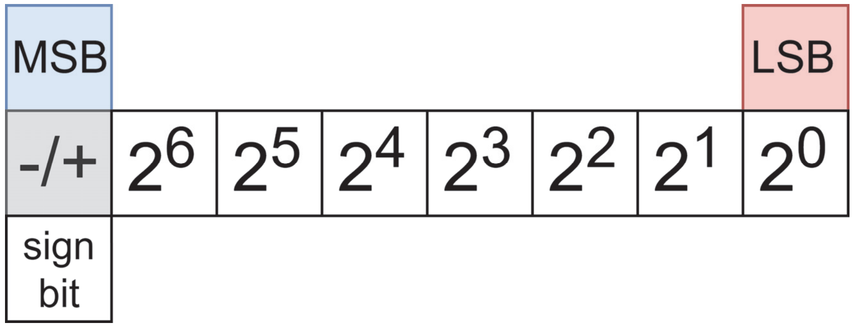
    - **What about negative numbers?**



Powers of 2 are then added to give the decimal

COMPUTER ARITHMETIC

- **Integers**
  - Integers require the addition of negative numbers to the representation method
  - What makes a **negative number** different than natural number is the need to specify they **are negative** – the sign must be represented somehow
  - While a natural number is represented without a sign (called ***unsigned*** i.e. 00110011), an integer requires a bit to be reserved to indicate the sign.
  - To represent integers, the MSB is reserved as a sign bit.
    - If the MSB is **0**, the integer is **positive**
    - If the MSB is **1**, the integer is **negative**



  - This means that less numbers can be represented since there is less powers of 2…

COMPUTER ARITHMETIC

- **Integers**
  - **Negative number representation:**
    - **Sign Magnitude:**
      - In this method, the sign bit is changed while the rest of the bits represent the number
        - 00001001 will be +9 – positive nine
        - 10001001 will be -9 – negative nine
      - This is an issue! Why?

COMPUTER ARITHMETIC

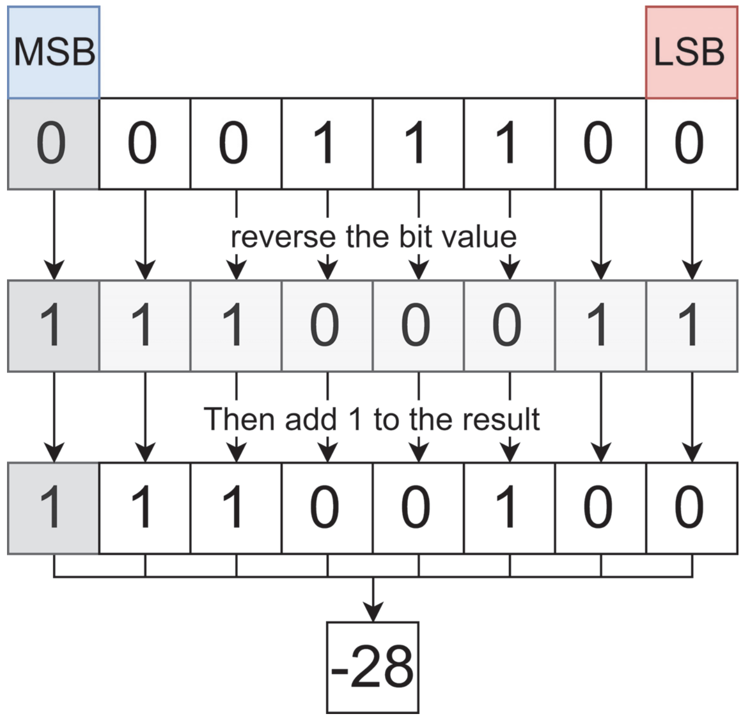
- **Integers**
  - **Negative number representation:**
    - **Sign Magnitude:**
      - In this method, the sign bit is changed while the rest of the bits represent the number.
        - 00001001 will be +9 – positive nine
        - 10001001 will be -9 – negative nine
      - This is an issue! Why?
        - 00000000 will be +0 – positive zero
        - 10000000 will be -0 – negative zero
          - Zero is not positive nor negative! It is unsigned
          - Two zeros will make addition (and the other operations) more complex
      - Therefore, the total number represented by a sign magnitude N bits is:
        - $(-2^{N-1}-1)$ to $(2^{N-1}-1)$
        - **For 8 bits: -127 to 127 ---- Which is 255 numbers (0 included)**

COMPUTER ARITHMETIC

- **Integers**
  - **Negative number representation:**
    - **One’s Complement (1’s complement):**
      - In this method, the negative representation is done by reversing the value of all the bits in positive representation --- -A = ~A
        - 00001001 will be +9 – positive nine
        - 11110110 will be -9 – negative nine
      - This is an issue! Why?
        - 00000000 will be +0 – positive zero
        - 11111111 will be -0 – negative zero
          - Zero is not positive nor negative! It is unsigned
          - Two zeros will make addition (and the other operations) more complex
      - Therefore, the total number represented by an 1’s complement N bits is:
        - $(-2^{N-1}-1)$ to $(2^{N-1}-1)$
        - **For 8 bits: -127 to 127 ---- Which is 255 numbers (0 included)**

COMPUTER ARITHMETIC

- **Integers**
  - **Negative number representation:**
    - **Two's Complement (2's complement):**
      - In this method, the negative representation is done by reversing the value of all the bits in positive representation and adding 1 to the result --- -A = ~A + 1
        - 00011100 will be +28
        - To produce -28, we reverse the bit values
        - And add 1 to the result
        - This way, we only have one value of 0
        - 00000000 will be 0 (unsigned)
        - 10000000 will be -128
      - Therefore, the total number represented by a sign magnitude N-bit is:
        - (-2^{N-1}) to $(2^{N-1}-1)$
        - **For 8 bits:-128 to 127 ---- 256 numbers**



COMPUTER ARITHMETIC

- **Rational numbers? ---** Integer vs Fixed-point vs Floating-point
    - Integers – $b^{N-1}b^{N-2}\ldots b^{2}b^{1}b^{0}$ – range (-2^{N-1}) to $(2^{N-1}-1)$
    - Fixed-point numbers – $b^{N-1}b^{N-2}\ldots b^{2}b^{1}b^{0}.b^{-1}b^{-2}\ldots d^{-m}$ – range (-2^{N-1}) to $(2^{N-1}-2^{-m})$
    - Floating-point numbers:
        - Exponent: integer (can be with bias)
        - Mantissa: (normalized) fixed-point number
  - Main operations like between integers
        - The position of the point (dot) may need correction(s) [==normalization]
    - Integers --- ☺ simple operations / ☹ no fraction
    - Fixed-point numbers
        - ☺ simple addition and subtraction
        - ☹ normalization needed for multiplication and division
    - Floating-point numbers
        - ☺ flexible range
        - ☹ normalization may be needed before and after operations

COMPUTER ARITHMETIC

- **Rational numbers?**
  - **Integers** --- 1+15 bits ≈ -32000 to +32000, precision 1
  - **Fixed-point numbers** – operations
    - $b^{N-1}b^{N-2}...b^{2}b^{1}b^{0}.b^{-1}b^{-2}...d^{-m}$ – range (-2^{N-1}) to $(2^{N-1}-2^{-m})$
    - N bits, m bits after point $(2^{N-m}-2^{-m}) \approx D/2^{m}$
    - 0.625 = 1/2 + 1/8 = 0000.10100000 = 160 / 256
    - Addition & subtraction
      - $a+b = (A/2^{m})+(B/2^{m}) = (A+B)/2^{m}$ --- OK
    - Multiplication
      - $a*b = (A/2^{m})*(B/2^{m}) = (A*B)/2^{2m}$ --- “too small”
      - n & n bits → 2n bits → cutting m bits off...
    - Division
      - $a/b = (A/2^{m})/(B/2^{m}) = (A/B)/2^{0}$ --- “too large”
    - 1+5+10 bits ≈ -32 to +32, precision ~0.001 (~0.03%)
    - Normalization == shifting

COMPUTER ARITHMETIC

- **Rational numbers?**
  - **Floating-point numbers** – operations
    - S - sign 1 bit; E - exponent k bits & M - mantissa m bits
    - $\pm(D/2^m)*2^k$; $1 > D \geq 0.5$ & $2^{k-1} > K \geq -2^{k-1}$
    - $1.5 = 0.75*2^1 = 0|0001|11000000$
  - Addition & subtraction
    - $a+b = (A*2^{AE})+(B*2^{BE}) = (A*2^{AE})+(B*2^{AE-x}) = (A+B/2^x)*2^{AE}$
      - AE > BE (i.e., AE=BE+x) and 1 > mantissa > 0
      - Corrections may be needed before and after operations!
  - Multiplication & division
    - $a*b = (A*2^{AE})*(B*2^{BE}) = (A*B)*2^{AE+BE}$
    - $a/b = (A*2^{AE})/(B*2^{BE}) = (A/B)*2^{AE-BE}$
    - Normalization may be needed after operations!
  - 1+5+10 bits ≈ -64000 to +64000, precision ~0.1%
  - Normalization == analysis & shifting

COMPUTER ARITHMETIC

- **Floating-point representation**

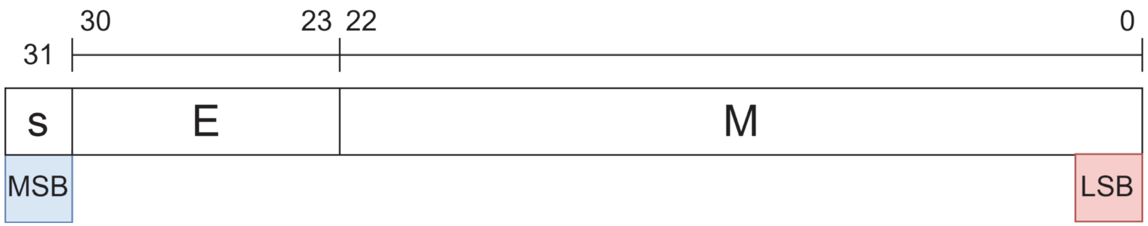
125 . 3625		
whole number	**point**	**fraction**
125	**.**	**3625**

- Floating point representation in the binary system require **high degree of accuracy**
- Since the number of bits available for the representation is limited, the accuracy of the representation is highly dependant on the number of dedicated bits.
- Example:
  - 12.879
  - First we convert the whole number: 12 = **1100**
  - Then we convert the fraction:
    - We can use: 0.5 (2^{-1}) and 0.25 (2^{-2}) and 0.125 (2^{-3})
      - .879 is equal to **0.5** + **0.25** + **0.125** + 0.004
      - We have the first three (0.875 = **.111**), but the fourth fraction can not be represented as we do not have enough bits to represent it, therefore it is left unrepresented – the binary representation is not precise.
        - **12.879 = 1100.111**

MSB								LSB
-/+	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
sign bit								

COMPUTER ARITHMETIC

- **Floating Point representation**
  - We can represent integers using their exponents:
    - -1,850,000 is equal to $-\mathbf{1.85} \times \mathbf{10^6}$
    - 0.000732 is equal to $\mathbf{7.32} \times \mathbf{10^{-4}}$
  - Floating points can also be represented using exponents of 2.
  - Example: **32-bit representation floating point numbers IEEE standard**
    - To understand this representation, floating point numbers are automatically represented in the following equation:
      - $s\ 1.M \ \times 2^E$
      - M : <u>M</u>antissa (23 bits), s: <u>S</u>ign (1 bit), E: <u>E</u>xponent (8 bits)
      - This called the single precision representation
    - The 32 bits are organized as follows:



COMPUTER ARITHMETIC

- **Floating Point representation**
  - **32-bit representation floating point numbers IEEE standard**
    - Example: 245.28125
    - We first convert the Whole number: 245 = 11110101
    - Then convert the fraction: 0.28125 = 0.25 + 0.03125 = $(2^{-2} + 2^{-5})$ = .01001
    - We end up with the representation – 11110101.01001
    - This number is then converted to an exponent:
      - The point is moved behind the very first 1 in the representation, while we keep count of the number of steps it was moved.
      - 1.**111010101001** – floating point moved **7** times (**Exponent**). Add bias of 127
      - **Exponent Representation** = 127 + 7 = 134 = 1000 0110
    - The exponent is as follows: **0** $1.\mathbf{111010101001} \times 2^{\mathbf{10000110}}$

      OR **+**$1.\mathbf{111010101001} \times 2^{\mathbf{7}}$
    - The representation is as follows (not the first missing bit in mantissa!):
      - **0 10000110 11101010100100000000000**
      - **s E M**

COMPUTER ARITHMETIC

- **Operations on Binary**
  - Now, since we can represent integers (positive and negative), let us perform operations on those integers:
    - **Addition**:
      - Addition is a simple operation to preform on integers.
      - The binary addition table is
        - 0 + 0 = 0
        - 0 + 1 = 1
        - 1 + 0 = 1
        - 1 + 1 = 0 and 1 as carry out

+	**0**	**1**
0	0	1
1	1	0/1 c

      - Simply put, adding a 1 to 1 is in fact 2 in binary
        - 2_{10} is 10_2 in binary
        - Which means that one bit value of 1 is carried to the next power of 2 magnitude
      - Addition is a very simple operation that can be performed using a full adder

COMPUTER ARITHMETIC

- **Operations on Binary**
  - Now, since we can represent integers (positive and negative), let us perform operations on those integers:
    - **Subtraction**
      - Subtraction is a simple operation to preform on integers.
      - The binary subtraction table
        - 0 - 0 = 0
        - 1 - 0 = 1
        - 1 - 1 = 0
        - 0 - 1 = 1 and 1 as borrow

-	**0**	**1**
0	0	1
1	1/1 b	0

      - Simply put, subtracting a 1 from 0 is not possible, so we use the value found in the higher magnitude to raise the value of the 0 to 10.
      - Subtraction can be done using 2 methods:
        - Direct subtraction if the minuend is larger than the subtrahend
        - Or using addition → (4 – 10) is equivalent to (4 + (-10))
      - No need to build a subtractor if you can build a Positive to Negative and an adder

COMPUTER ARITHMETIC

- **Addition – Example**

```
                  0011000.   carry
   13             00001101
 + 24           + 00011000
 ----           ----------
   37             00100101   result
```

- **Subtraction – Examples**

```
                  1110000.   borrow                          1
   13             00001101                 00001101         00001101
 - 24           - 00011000               + 11101000       + 11100111
 ----           ----------               ----------       ----------
  -11             11110101   result        11110101         11110101
                  [ A - B ]               [ A + -B ]       [A+(~B)+1]
```

COMPUTER ARITHMETIC

- **Operations on Binary – Overflow?**
  - 2's complement
    - 8 bits: range -128…+127
    - What if: 125+5=? [130]

```
 125   01111101
+  5   00000101
[130]  10000010 == -126
```

    - Number scale

```
  -1   11111111
       ........
-128   10000000
+127   01111111
       ........
   0   00000000
```

- Extra sign bit – 00 or 11 → OK

```
     75   001001011
+    15   000001111
   [90]   001011010

    125   001111101
+     5   000000101
  [130]   010000010

    -75   110110101
+   -15   111110001
  [-90]   110100110

   -125   110000011
+    -5   111111011
 [-130]   101111110
```

COMPUTER ARITHMETIC

- **Operations on Binary**
  - **Multiplication**
    - Similar to multiplication of decimal numbers

```
13 * 24         00001101 * 00011000
-------         -------------------
     52                 00001101...
    26                 00001101
-------         -------------------
    312                000100111000
```

    - Adding & shifting – all implementations are based on this simple algorithm
      - A binary multiplier can be implemented using a sequence of additions
      - $3_{10} \times 4_{10} \rightarrow 0011 \times 0100 = 0100 + 1000 + 0000 + 0000 = 1100 = 12_{10}$

COMPUTER ARITHMETIC

- **Operations on Binary**
  - **Multiplication**
    - Adding & shifting – all implementations are based on this simple algorithm

```
   00011000 * 00001101    [ 24 * 13 ]
   -------------------
1.            00000000
2.           00000000.
3.          00000000..
4.         00001101...
5.        00001101....
6.       00000000.....
7.      00000000......
8.     00000000.......
   -------------------
       000000100111000    [ 312 ]
```

COMPUTER ARITHMETIC

- **Operations on Binary**
  - **Division**
    - Similar to division of decimal numbers

```
312 / 13 = 24         0100111000 / 01101 = 011000
26                   - 01101
---                   ----------
 52                   0001101000
 52                  -  01101
---                   ----------
  0                   0000000000  [reminder]
===                   ==========
```

- Subtracting, checking & shifting – all implementations are based on this algorithm
  - A binary divider can be implemented using a sequence of subtractions