Real number formats & arithmetic

Signal Processing Systems Fall 2025

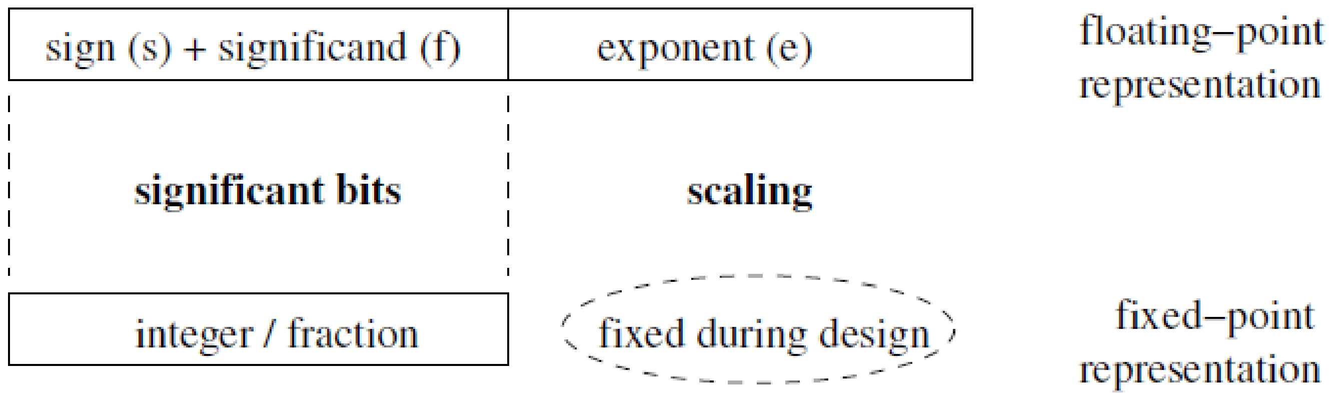
Lecture 2 (Thursday 30.10.)

Outline

- Types of real number representations
- Basis: representing integers
- Fixed-point numbers
    - Scaling, characterization, arithmetics
- Floating-point numbers
    - Modes, custom design, characterization
- Saturation & rounding

I. Types of real number formats

Compositions of the word bits:


sign (s) + significand (f)
exponent (e)
floating–point representation
significant bits
scaling
integer / fraction
fixed during design
fixed–point representation

In both forms, **significant bits** represent an integer* and **scaling** maps it to some real number.

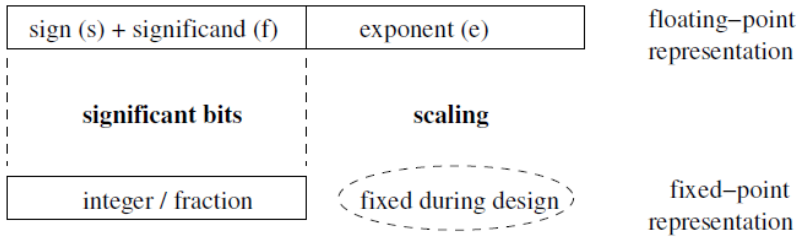
Floating-point HW performs automatic scaling actions (to maximize precision).

In the fixed-point design, one must be aware of the scalings. **To manage it, effort needed at design time!**

*in the case of floating-point it's actually a fraction (a scaled integer)

Example (16-bit word length)

Representing -24.203125


sign (s) + significand (f)
exponent (e)
floating–point representation
significant bits
scaling
integer / fraction
fixed during design
fixed–point representation

Floating point: IEEE 754-2008 half-precision (binary16)

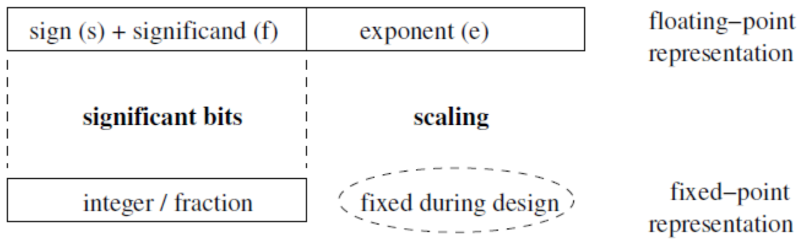
$-24.203125 = -1.5126953125 \times 2^{4}$

binary16 uses 5 bits for scaling

$$\underbrace{1}_{s}\ \underbrace{10011}_{e}\ \underbrace{1000001101}_{f}$$

Example (16-bit word length)

Representing -24.203125


sign (s) + significand (f)
exponent (e)
floating–point representation
significant bits
scaling
integer / fraction
fixed during design
fixed–point representation

Fixed point: some scaling associated with an integer

No bits used for scaling!

2's complement integer,
binary point scaling,
6 fraction bits

$-24.203125 = -1549 \times 2^{-6}$

1111100111.110011

HW treats the number as an integer -1549

7 fraction bits

$-24.203125 = -3098 \times 2^{-7}$

111100111.1100110

HW treats the number as an integer -3098

Example. Representing -2.5. Computing -2.5 x -2.5

Floating-point case.

Representing: A value in the range [1,2) is multiplied by two's power.
For -2.5, we get -1.25 x 2^1
Then, the fraction 0.25 is encoded with some bits. Some number of bits is used for exponent (1) and one bit for the sign (-). **Done by compiler.**

Computing -2.5 x -2.5: HW multiplies 1.25 x 1.25 = 1.5625, adds 2's exponents (1+1=2), and puts the result to floating-point format, that is, encodes fraction 0.5625, exponent 2, and sign (+).

[Check: 1.5625* 2^2 = 6.25].

Example. Representing -2.5. Computing -2.5 x -2.5

Fixed-point case.

Representing: Could be integer -5 with associated scaling 0.5 (**designer decides**).

Computing -2.5 x -2.5: HW computes -5 x -5 = 25. Designer knows associated scaling 0.25 (0.5 x 0.5).

[Check: 25*0.25 = 6.25]

Characterization of formats

$\underbrace{1}_{s}\ \underbrace{10011}_{e}\ \underbrace{1000001101}_{f}$

11001110000.01101

	IEEE 754-2008 half-precision (binary16)	2's complement, binary point scaling, 5 fraction bits
Word length	16	16
Range	-65504 ... +65504	-1024 ... +1023
Dynamic range	240 dB	90 dB
Precision (ulp)	variable (2^{-24} ... 2^{5}=32)	$2^{-5} = 0.03125$
Bit precision	11	16

- Word length: the number of bits used
    - short length desirable
    - affects memory sizes, bus widths
    - affects complexity of binary arithmetics implementation
    - typical lengths in processors: 8, 16, 32, 64
- Range: the smallest and largest representable value
    - if computations go out-of-range, we have **overflow**
- Dynamic range: the ratio between the smallest and largest absolute values
    - large value desirable
    - strength of floating-point formats
- Precision: unit in least position (ulp, weight of the 'last' bit)
    - related to **roundoff noise** and **underflow** problem
    - for fixed-point, all values are represented with the same precision
    - for floating-point, large values are represented with less precision
- Bit precision: number of significant bits
    - the number of bits that provide precision to the value represented

$$\text{Dynamic range in dB} = 20\log_{10}\frac{\textit{largest absolute value}}{\textit{smallest nonzero absolute value}}$$

History of use (1)

- In the past most of the embedded signal processing implementations were made with fixed-point representation, because hardware did not usually support floating-point
- As faithful implementation of floating-point algorithms with simplistic hardware was complicated, even the standards defined (e.g., for media codecs) relied on fixed-point algorithm descriptions
- Another reason for favoring fixed-point: belief on the inherent energy efficiency and speed of fixed-point implementations. In some cases, it can be true, but not in general
  - Complete HW implementations of floating-point standards are complex. However, all features are actually not needed => simplified HW design => efficiency increase

History of use (2)

- Floating-point formats have larger dynamic range, which makes implementation of an algorithm easier in many cases
  - designer is not so concerned about numerical issues
  - tool support can be better (e.g. code generation is easier when low-level manual coding effort is not needed)
  - Result: faster time-to-market
- As a result, nowadays many DSP processing architectures are based on floating-point pipelines
  - 32-bit precision (so-called single precision) is typically sufficient
- Neural networks: in embedded systems, interest to use fixed-point integer-based calculation (large number of network parameters, dynamic range not so important)

II. Binary representations of integers

- Assigning meaning to bit strings $a_{p-1}a_{p-2}\dots a_1a_0$ $a_i \in \{0,1\}$
- Unsigned integers $V_{int} = \sum_{i=0}^{p-1} a_i 2^i$
- Four binary representations for signed integers

$$V_{int} = -a_{p-1}2^{p-1} + \sum_{i=0}^{p-2} a_i 2^i$$

Two's complement

$$V_{int} = -a_{p-1}(2^{p-1} - 1) + \sum_{i=0}^{p-2} a_i 2^i$$

One's complement

$$V_{int} = (-1)^{a_{p-1}} \times \sum_{i=0}^{p-2} a_i 2^i$$

Sign-magnitude

$$V_{int} = \sum_{i=0}^{p-1} a_i 2^i - B$$

Excess-B: unsigned integer - bias

Example. Signed integers, 3-bit word length

Bit weights -3, +2, +1

Bit weights -4, +2, +1

$a_2a_1a_0$	sign-magnitude	1's complement	2's complement	excess-4
011	+3	+3	+3	-1
010	+2	+2	+2	-2
001	+1	+1	+1	-3
000	0	0	0	-4
111	-3	0	-1	+3
110	-2	-1	-2	+2
101	-1	-2	-3	+1
100	0	-3	-4	0

invert most significant bit

Bias $B = 2^{p-1}$ gives so-called offset binary format.

$$V_{int} = \sum_{i=0}^{p-1} a_i 2^i - B$$

000 = lowest value

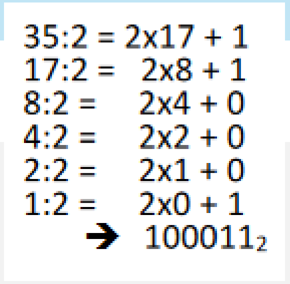
111 = highest

Example. Brush-up file, cases 1-4

Approaches there

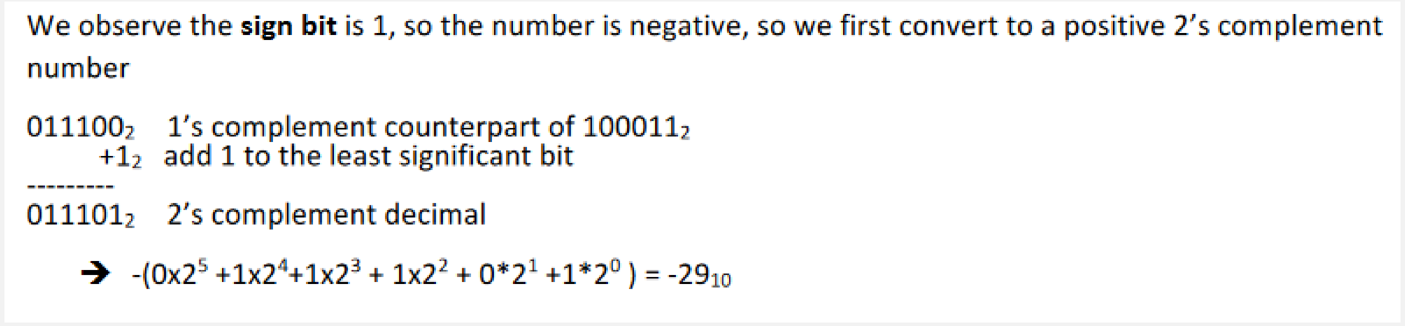
Converting integer decimal to binary

35_{10} to unsigned binary:



35:2 = 2x17 + 1
17:2 = 2x8 + 1
8:2 = 2x4 + 0
4:2 = 2x2 + 0
2:2 = 2x1 + 0
1:2 = 2x0 + 1
$\rightarrow$ 100011_2

100011_2 **2's complement binary to decimal**:



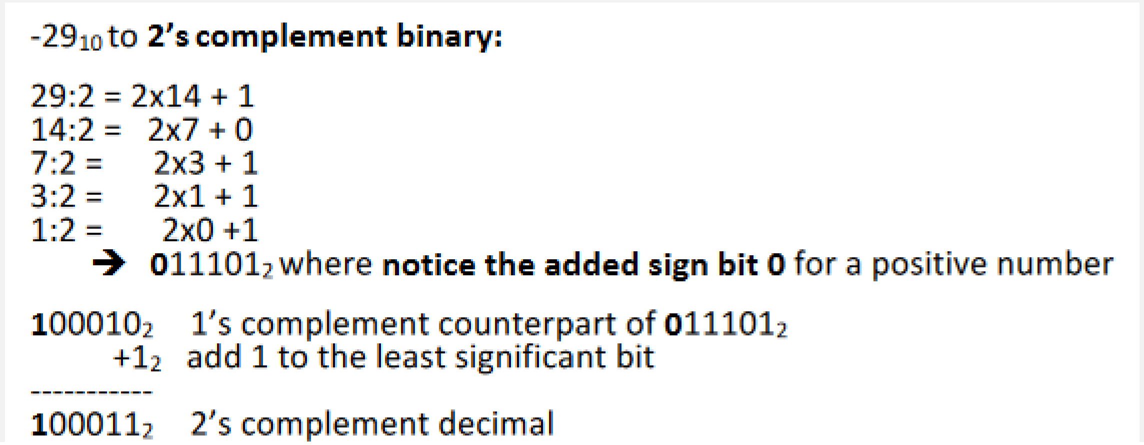
We observe the **sign bit** is 1, so the number is negative, so we first convert to a positive 2's complement number

011100_2 1's complement counterpart of 100011_2
$+1_2$ add 1 to the least significant bit

011101_2 2's complement decimal

$\rightarrow$ $-(0x2^5 +1x2^4+1x2^3 + 1x2^2 + 0*2^1 +1*2^0) = -29_{10}$

-29_{10} to **2's complement binary:**



-29_{10} to **2's complement binary:**

29:2 = 2x14 + 1
14:2 = 2x7 + 0
7:2 = 2x3 + 1
3:2 = 2x1 + 1
1:2 = 2x0 +1
$\rightarrow$ **0**11101_2 where **notice the added sign bit 0** for a positive number

1$00010_2$ 1's complement counterpart of **0**11101_2
$+1_2$ add 1 to the least significant bit

1$00011_2$ 2's complement decimal

Example. Brush-up file, cases 1-4

Solving problems using memorized powers of two: 1, 2, 4, 8, 16, 32, 64, 128, ... (i.e. weights of bits, starting from LSB)

1 **Converting integer decimal to binary**

35_{10} to unsigned binary:

Write it as a sum of two's powers 35 = 32 + 2 + 1. At least 6 bits are needed. If more than 6, then just add zeros as most significant bits.

32	=	1	0	0	0	0	0
2	=	0	0	0	0	1	0
1	=	0	0	0	0	0	1
Bit OR		**1**	**0**	**0**	**0**	**1**	**1**

2 100011_2 **unsigned binary to decimal:**

Compute the sum of bit weights. Start from LSB. 1 + 2 + 32 = 35

3 100011_2 **2's complement binary to decimal:**

The weight of the first bit is negative, so we have 1 + 2 + -32 = -29

$$V_{int} = -a_{p-1}2^{p-1} + \sum_{i=0}^{p-2} a_i 2^i$$

Two's complement

4 -29_{10} **to 2's complement binary:**

Here, convert 29 to binary first: 29 = 16 + 8 + 4 + 1 = 1 1 1 0 1
The number of bits must be at least 6 to represent -29.
Then, 29 = **0** 1 1 1 0 1. Bitwise complement: 1 0 0 0 1 0.
Add 1 to LSB: 1 0 0 0 1 1 = -29

16	=	1	0	0	0	0
8	=	0	1	0	0	0
4	=	0	0	1	0	0
1	=	0	0	0	0	1
		1	**1**	**1**	**0**	**1**

Check: 1 0 0 0 1 1 = 1 + 2 + 32 - 64 = -29.

Sign extension allows conversion to a longer format (e.g. 9 bits)

1 1 1 1 0 0 0 1 1 = 1 + 2 + 32 + 64 + 128 – 256 = -29.

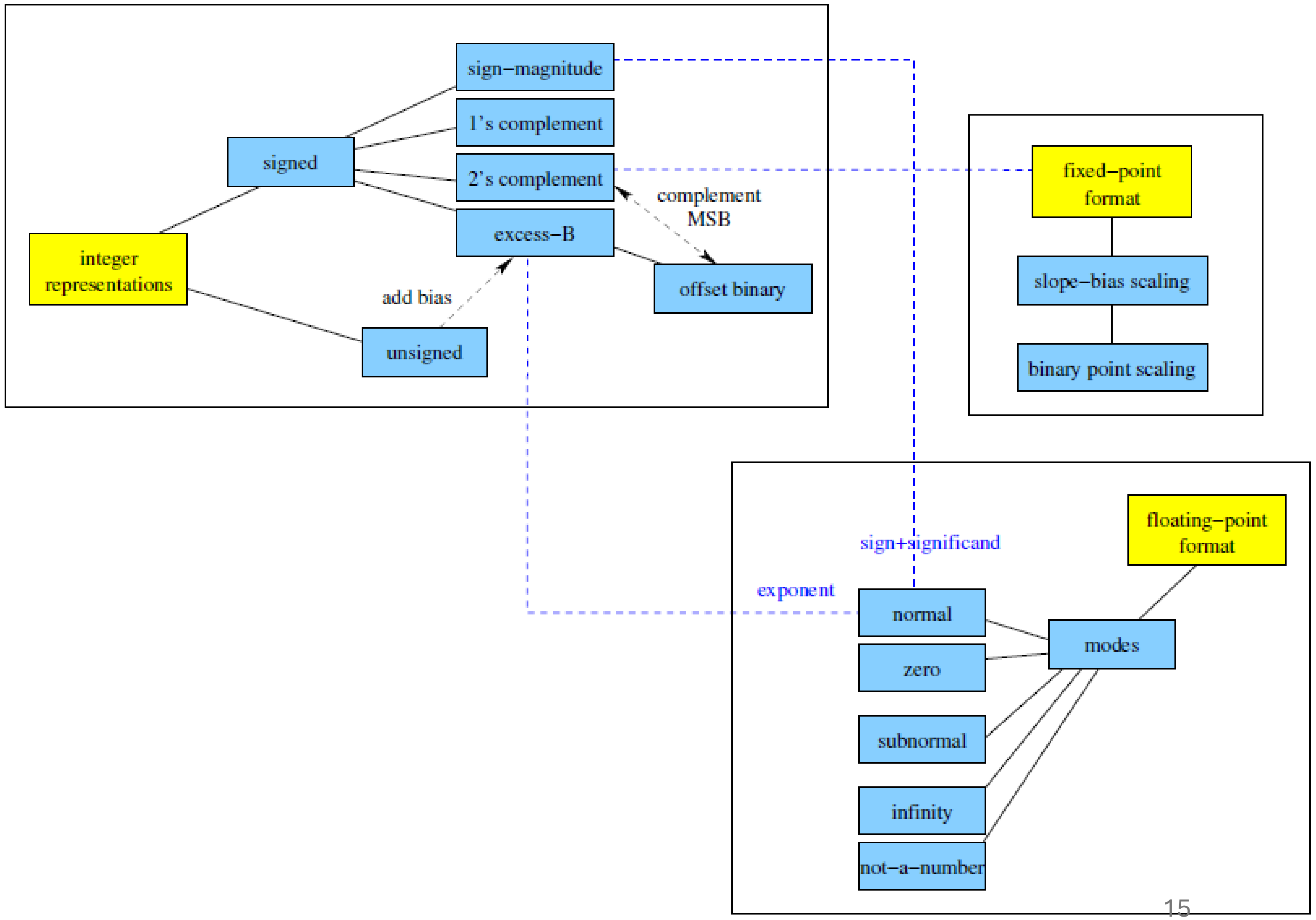
Integer representations and real number formats

Choice of the integer encodings for real number formats depends on what is the best choice from hardware design viewpoint.

Therefore, **two's complement format** is mostly used as a basis for fixed-point representation.

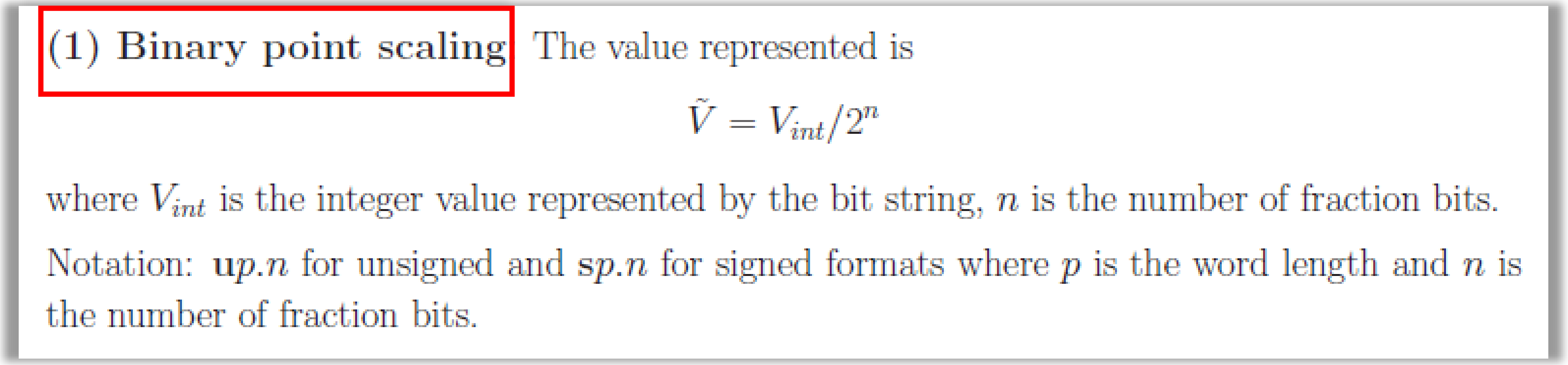
In floating-point representations we can see hints of **sign-magnitude** (s & 1.f) and **excess-B** (e).

$$\underbrace{1}_{s}\ \underbrace{10011}_{e}\ \underbrace{1000001101}_{f}$$



III. Fixed-point formats

- Two’s complement integers used as a basis for signed fixed-point
- Scaling is assigned to integers in two ways
  - (1) Binary point scaling
  - (2) Slope bias scaling



(1) Binary point scaling The value represented is

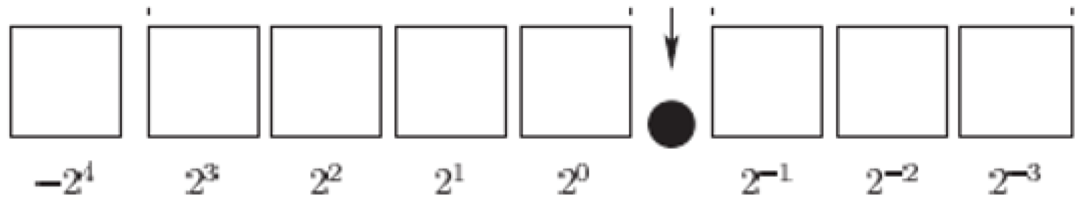
$$\tilde{V} = V_{int}/2^n$$

where V_{int} is the integer value represented by the bit string, n is the number of fraction bits.

Notation: u$p.n$ for unsigned and s$p.n$ for signed formats where p is the word length and n is the number of fraction bits.

sp.n

For example, s8.3 denotes the following format:



Note on format notation **s***p*.*n*

- In this course, we use ***sp.n*** as it shows clearly the word length ***p*** and fraction length ***n***.
- There are alternatives like **Q notation**. Meaning of it varies:

  1) Texas Instruments variant:

  Qm.n* = s*p.n, where ***m*** = ***p-n-1***

  2) ARM variant:

  Qm.n* = s*p.n, where ***m*** = ***p-n***

Example. Determining value of bit string

2’s complement, binary point scaling, word length = 9, 3 fraction bits (**s9.3**)

110000.011

What is its value?

Answer.

Bit weights	-256	128	64	32	16	8	4	2	1
Bits	1	1	0	0	0	0	0	1	1

V_{int} = 1 + 2 + 128 - 256 = -125

3 fraction bits => divide by 2^3

-125 / 8 = <u>-15.625</u>

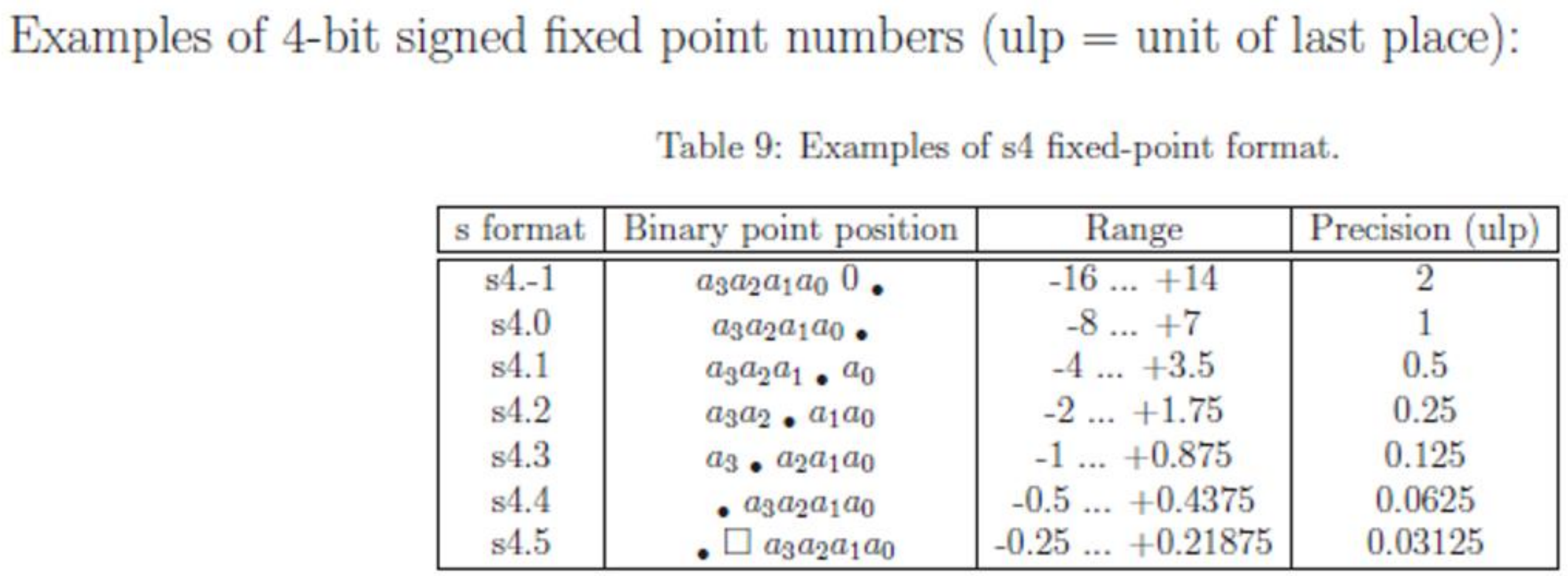
$$\tilde{V} = V_{int}/2^n$$

Lesson: 2’s complement ➔ Do not interpret fraction separately !!!

Bits .011 do not represent here $2^{-2} + 2^{-3}$!!! (common error in exercises)

You must compute the integer first and then scale it.

Example. ***sp.n*** and position of binary point



Examples of 4-bit signed fixed point numbers (ulp = unit of last place):

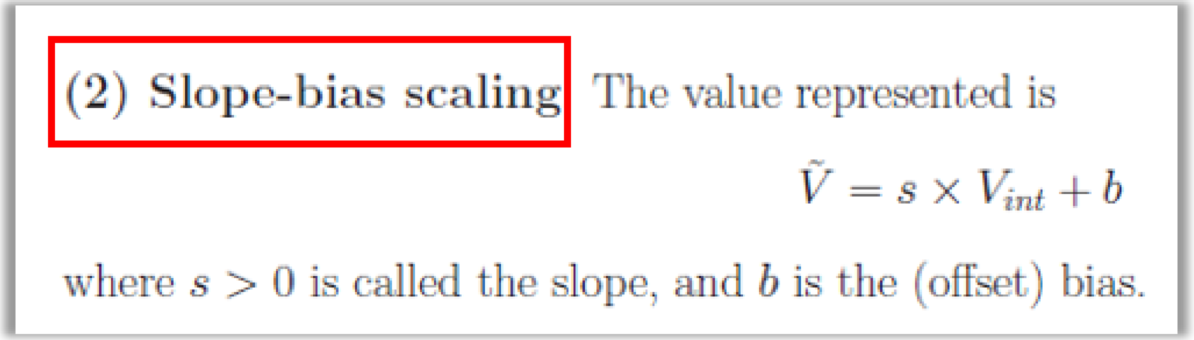
Table 9: Examples of s4 fixed-point format.

s format	Binary point position	Range	Precision (ulp)
s4.-1	$a_3a_2a_1a_0\,0\,\bullet$	-16 ... +14	2
s4.0	$a_3a_2a_1a_0\,\bullet$	-8 ... +7	1
s4.1	$a_3a_2a_1\,\bullet\,a_0$	-4 ... +3.5	0.5
s4.2	$a_3a_2\,\bullet\,a_1a_0$	-2 ... +1.75	0.25
s4.3	$a_3\,\bullet\,a_2a_1a_0$	-1 ... +0.875	0.125
s4.4	$\bullet\,a_3a_2a_1a_0$	-0.5 ... +0.4375	0.0625
s4.5	$\bullet\,\square\,a_3a_2a_1a_0$	-0.25 ... +0.21875	0.03125

Precision = 2^{-n}

Fraction length ***n*** can be negative.
n can be greater than word length ***p***.

Method 2: Slope-bias scaling



(2) Slope-bias scaling The value represented is

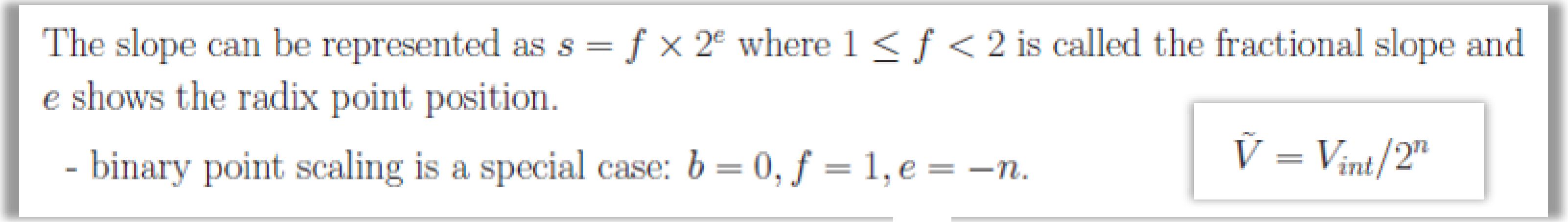
$$\tilde{V} = s \times V_{int} + b$$

where $s > 0$ is called the slope, and b is the (offset) bias.

So, completely specifed by the triplet (p, s, b)

- p = word length
- s = slope (how the represented integer is weighted)
- b = bias (= zero to use 2’s complement arithmetic)

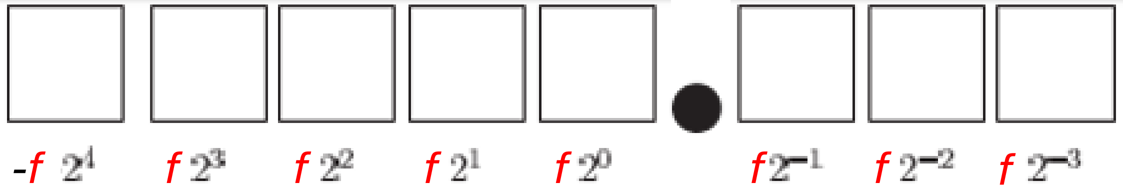
Relating this to the binary point:



The slope can be represented as $s = f \times 2^e$ where $1 \leq f < 2$ is called the fractional slope and e shows the radix point position.

- binary point scaling is a special case: $b = 0, f = 1, e = -n$.

$$\tilde{V} = V_{int}/2^n$$



Example. Encoding angles

- Encoding angles in the range [-π, π), when the word length (bit precision) is 6.

1) Binary point scaling:

- we must have two integer bits as $011 = 3$.

- thus the format to be used is s6.3.

- the range is $[-4, 4 - 2^{-3}]$.

- the precision of the format is $\text{ulp} = 2^{-3} = 0.125$.

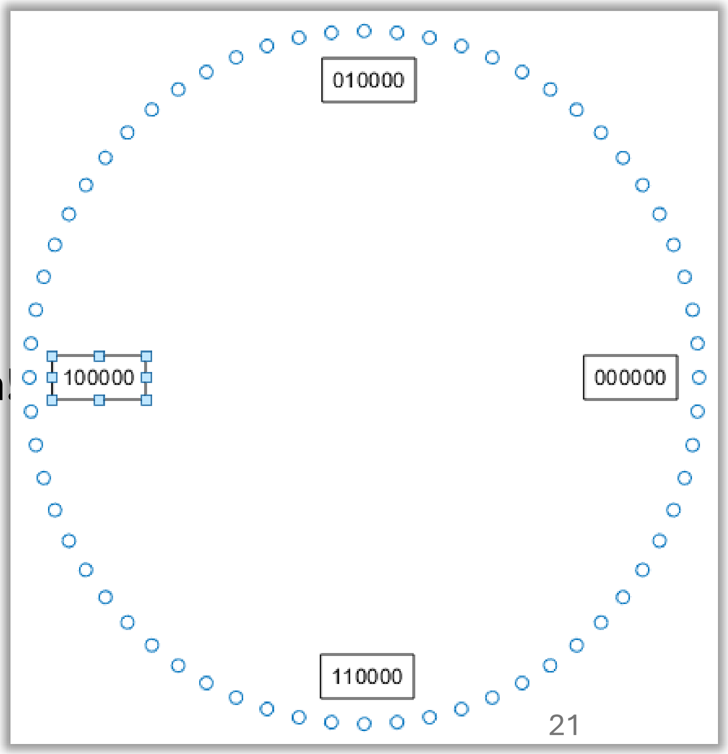
2) Slope-bias scaling:

- using zero bias, we use $s = \pi/2^5$ to use the full dynamic range.

- the range is then $\pi \times [-1, 1 - 2^{-5}]$ and $\text{ulp} = \text{s} \approx 0.0982$ **Better precision!**

Important:

In 2) the **quantized angle points are placed regularly** on the circle.



Example. Asymmetric coding of neural network coefficients

- Can be seen as slope-bias scaling
  - A parameter related to it is **zero point** = - **bias** / **slope**
  - Zero point constrained to be an integer
- Useful if coefficients are asymmetrically distributed around bias
- More about this in the next lecture
  - As pre-reading, see white paper by Nagel et al referred to in **intro1.pdf**

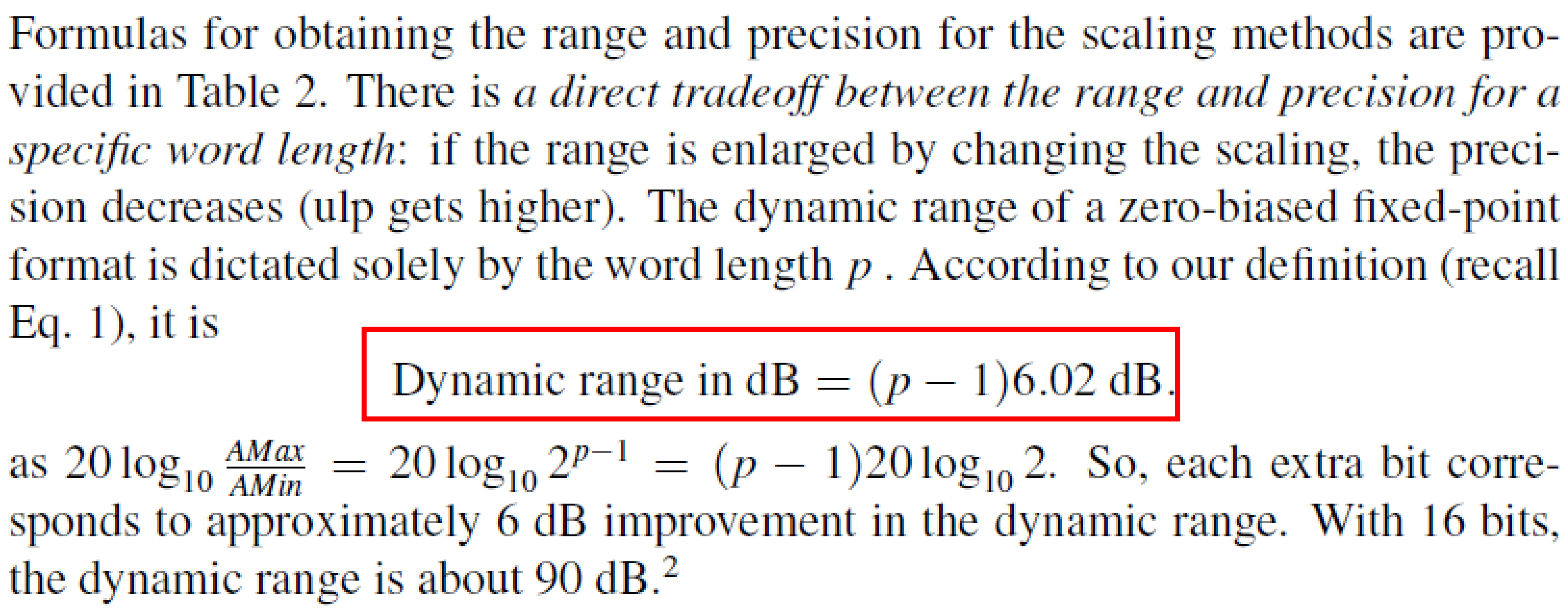
Quick test

- Fixed-point scalings
- Goto https://presemo.oulu.fi/spslec2

$$\tilde{V} = V_{int}/2^n$$

$$\tilde{V} = s \times V_{int} + b$$

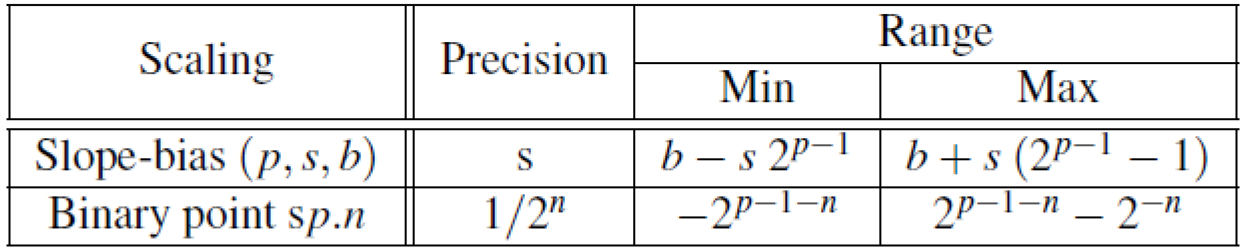
Characteristics of fixed-point formats



Formulas for obtaining the range and precision for the scaling methods are provided in Table 2. There is *a direct tradeoff between the range and precision for a specific word length*: if the range is enlarged by changing the scaling, the precision decreases (ulp gets higher). The dynamic range of a zero-biased fixed-point format is dictated solely by the word length p . According to our definition (recall Eq. 1), it is

$$\boxed{\text{Dynamic range in dB} = (p-1)6.02\ \text{dB}}.$$

as $20\log_{10}\frac{AMax}{AMin} = 20\log_{10}2^{p-1} = (p-1)20\log_{10}2$. So, each extra bit corresponds to approximately 6 dB improvement in the dynamic range. With 16 bits, the dynamic range is about 90 dB.[2]



Scaling	Precision	Range	
		Min	Max
Slope-bias (p, s, b)	s	$b - s\,2^{p-1}$	$b + s\,(2^{p-1} - 1)$
Binary point $sp.n$	$1/2^n$	-2^{p-1-n}	$2^{p-1-n} - 2^{-n}$

Modes of fixed-point arithmetic

- There are two types of arithmetics used in the fixed-point case:
  - Integer arithmetic: bit strings are treated as integers and do ordinary 2’s-complement arithmetics. This is the standard mode used in MCUs and fixed-point DSPs
  - Fractional arithmetic: bit strings are treated as fractions **s***p*.(*p*-1). For example, from the viewpoint of multiplications it is convenient as multiplication of fractions produces fractions.

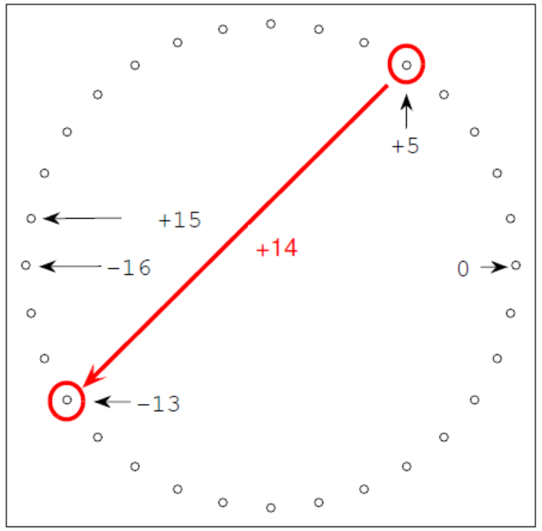
- Let us consider bit requirements in integer arithmetic ...

Integer arithmetic & addition

- Addition: output word length without overflow
  - *N* *p*-bit integers to be added => output word length $p + \text{ceil}(\log_2 N)$ bits is always sufficient
  - DSP processor can have provision for those extra bits (so-called guard bits in accumulator)
  - We may get along with shorter output word length. It depends on the magnitude of the integers added
- Addition overflow handling
  - If too few guard bits, wrap-up occurs
  - To prevent it, some processors may offer a saturating adder
  - In some cases, like CIC filters, wrap-up property is exploited!
    - Wrap-ups in intermediate computations are ok, **if it can shown that the final result is in the range of the fixed-point output format**

N	$\text{ceil}(\log_2 N)$
2	1
3	2
4	2
5	3
6	3
7	3
8	3
9	4

5 + 14



Wrap-around / modularity of 2's complement addition

Example. 5 + 14 – 10

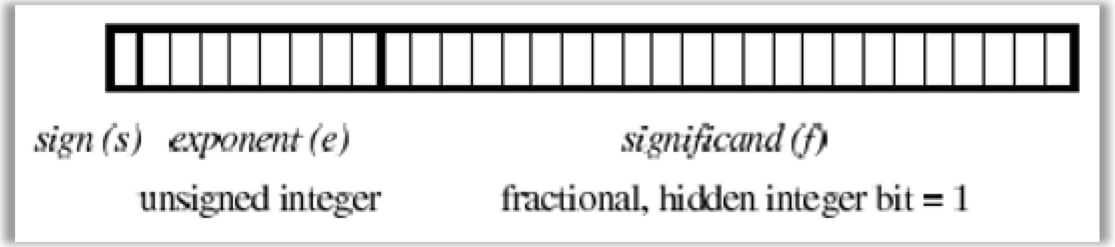
Integer arithmetic & multiplication

- Multiplication: output word length
  - **In general**: two *p*-bit integers => output word length 2*p* can provide full-precision result *”law of conservation of bits”*
  - Multiplier can have saturation
  - Rounding to drop least significant bits may be possible
    - Experiment on this in DT 2

- Multiplication by 2^k = just arithmetic shift by *k* bits to left
  - Negative k: shift to right
  - Cheap operation, but possibility of overflow / loss of precision
  - Arithmetic shifts are used in CORDIC implementation
    - More about this later

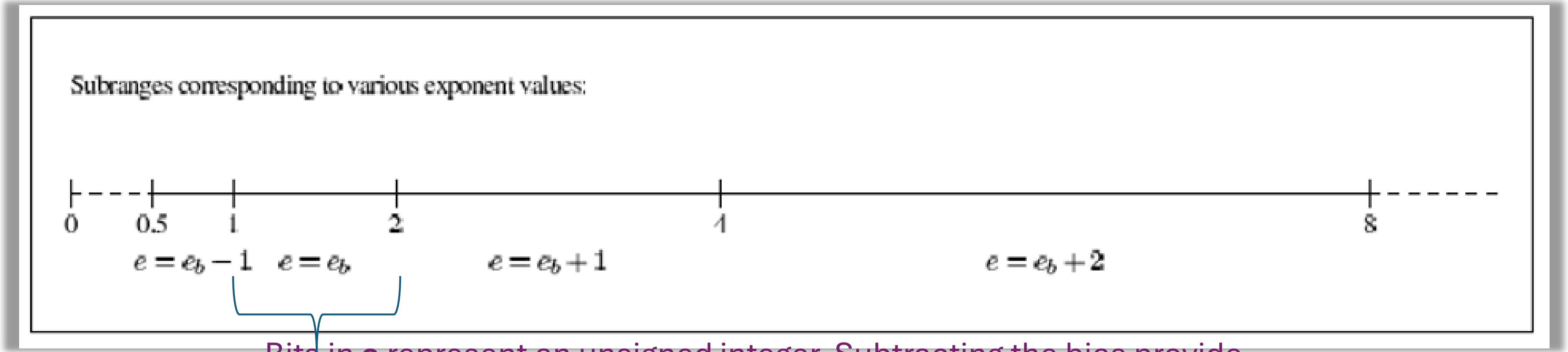
IV. Floating-point formats

- Design of the representation is based on HW implementation issues
- Bit string parts



Note: with this ordering, order of two floats is maintained when their bit patterns are interpreted as integers.

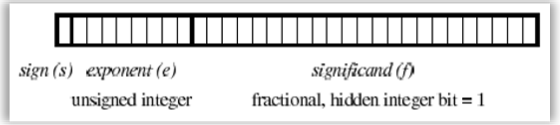
- Exponent defines which range of real numbers we are dealing with



Bits in **e** represent an unsigned integer. Subtracting the bias provide the represented exponent value. So, if e = bias, it represents zero.

Modes

Five modes of interpretation are recognized in IEEE 754 standard formats

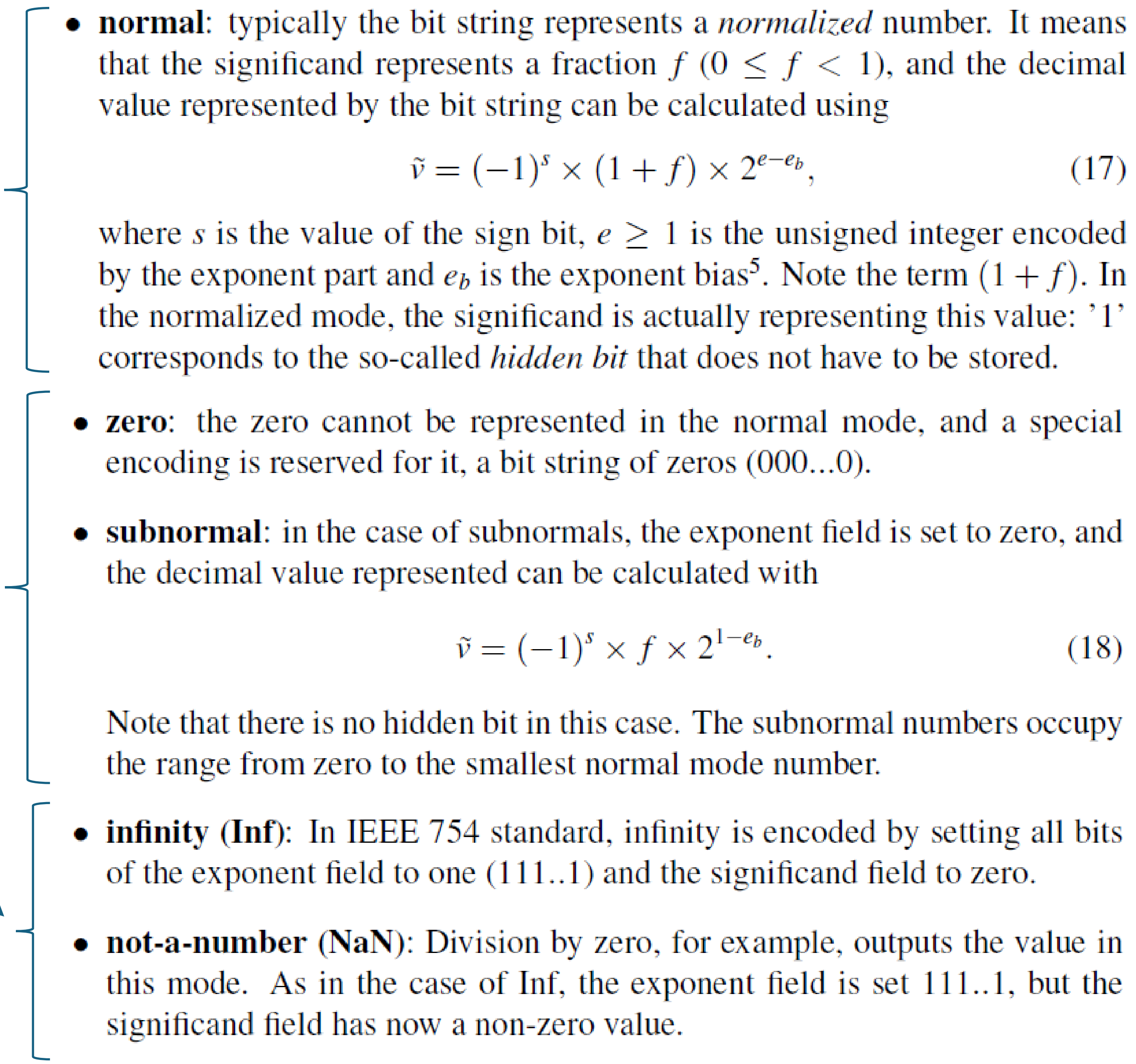


Exponent field value?

1 … MAX-1

ZERO

MAX



- **normal**: typically the bit string represents a *normalized* number. It means that the significand represents a fraction f $(0 \leq f < 1)$, and the decimal value represented by the bit string can be calculated using

$$\tilde{v} = (-1)^s \times (1 + f) \times 2^{e - e_b}, \tag{17}$$

  where s is the value of the sign bit, $e \geq 1$ is the unsigned integer encoded by the exponent part and e_b is the exponent bias[5]. Note the term $(1 + f)$. In the normalized mode, the significand is actually representing this value: '1' corresponds to the so-called *hidden bit* that does not have to be stored.

- **zero**: the zero cannot be represented in the normal mode, and a special encoding is reserved for it, a bit string of zeros (000...0).

- **subnormal**: in the case of subnormals, the exponent field is set to zero, and the decimal value represented can be calculated with

$$\tilde{v} = (-1)^s \times f \times 2^{1 - e_b}. \tag{18}$$

  Note that there is no hidden bit in this case. The subnormal numbers occupy the range from zero to the smallest normal mode number.

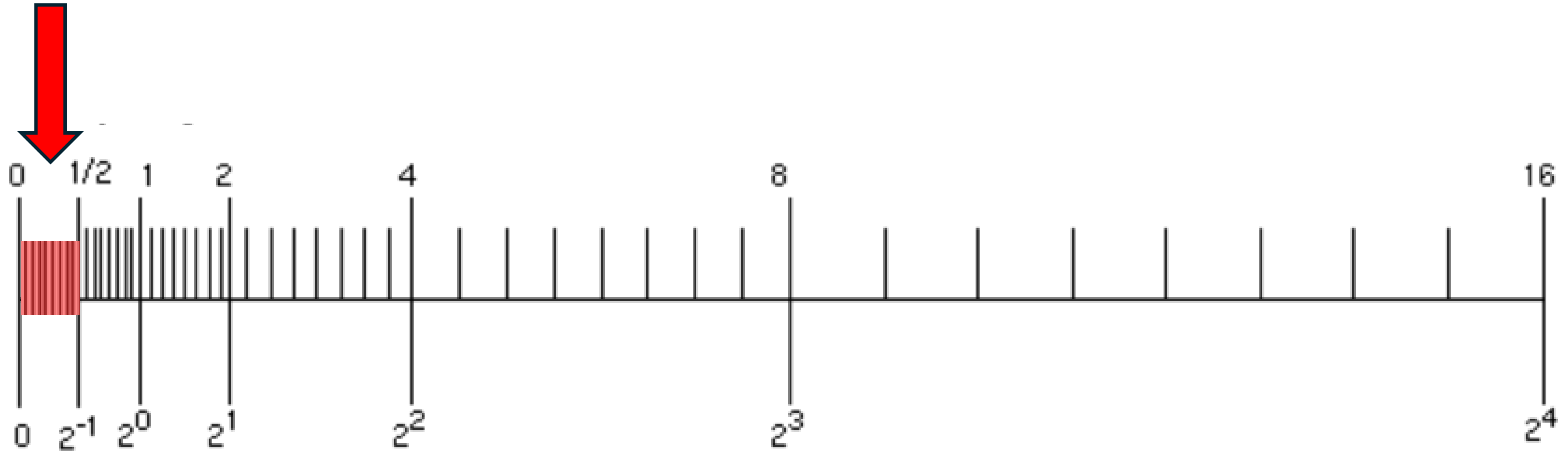
- **infinity (Inf)**: In IEEE 754 standard, infinity is encoded by setting all bits of the exponent field to one (111..1) and the significand field to zero.

- **not-a-number (NaN)**: Division by zero, for example, outputs the value in this mode. As in the case of Inf, the exponent field is set 111..1, but the significand field has now a non-zero value.

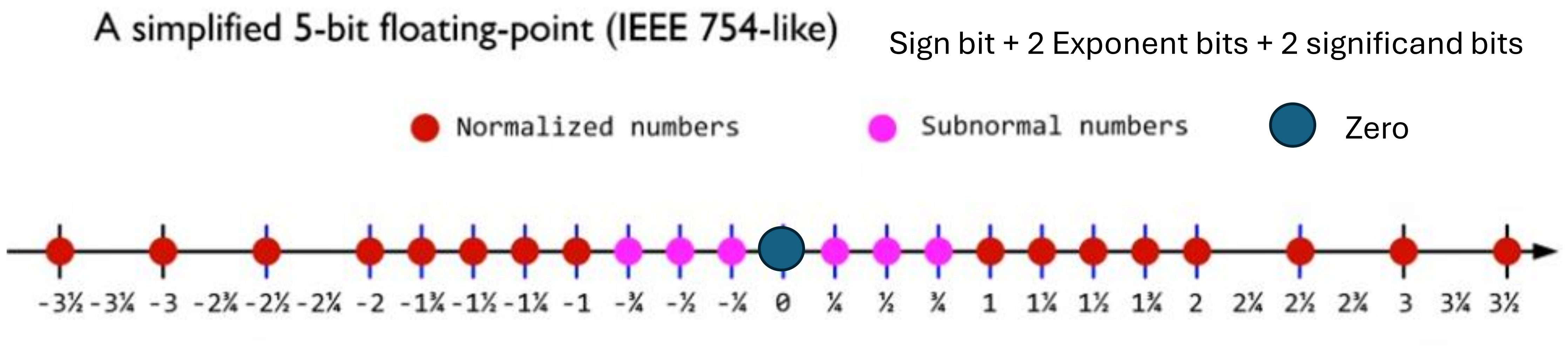
Subnormal mode

Normal mode numbers:
Different precision (ulp) in ranges ..., 1-2, 2-4, 4-8, ...
Large numbers represented with less precision

Subnormal mode fills the gap close to zero



Example. Normal / subnormal / zero

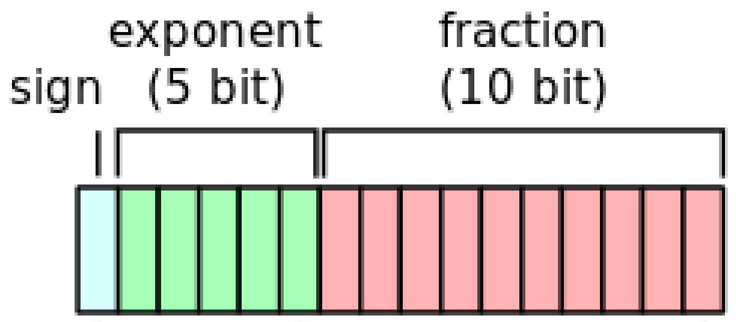


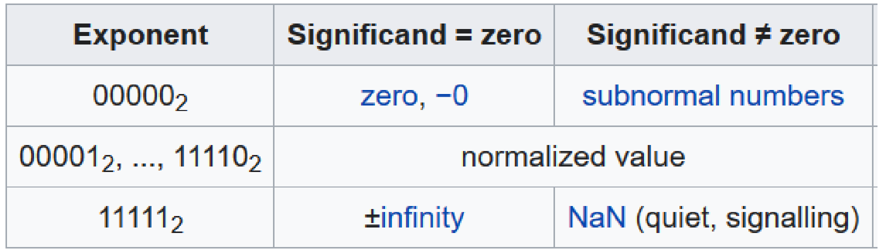
Sign bit + 2 Exponent bits + 2 significand bits

Varying precision = gap between adjacent number

Example. Bit string interpretation

- How the floating-point bit string is interpreted?
- Let us consider IEEE754 standard half precision format





Exponent	Significand = zero	Significand ≠ zero
00000_2	zero, −0	subnormal numbers
00001_2, ..., 11110_2	normalized value	
11111_2	±infinity	NaN (quiet, signalling)

Interpreting the bit string 1000110110001101:

Parts are: 1 00011 0110001101

1. The exponent part is $00011_2 = 3_{10}$
2. We have normal mode (normalized value) according to the table.
3. The bits of significand are 0110001101.
4. It corresponds to the fraction

   .0110001101 = $2^{-2} + 2^{-3} + 2^{-7} + 2^{-8} + 2^{-10} = 0.3876953125$

   .0110001101 = $(1 + 4 + 8 + 128 + 256) / 2^{10} = 397 / 1024 = 0.3876953125$
5. Exponent bias in the format is 15 and as the sign bit is 1 here,

   the value presented is $-1 \times (1 + 0.3876953125) \times 2^{3-15} \approx -3.3879 \times 10^{-4}$

Normal mode rule:

$$\tilde{v} = (-1)^s \times (1 + f) \times 2^{e - e_b}$$

Example. Mapping value to a bit string

Again, let us consider IEEE754 standard half precision format

-7.8 ?

1. Negative => Sign bit is one.

2. Represent the absolute value as U·W, where U in [1,2) and W is a power of two.

$$7.8 = 1.95 \cdot 2^2$$

3. Represent fraction part of U. Multiply it by $2^{(\text{number of fraction bits})}$. Represent the closest integer as a sum of two's powers.

$$0.95 \cdot 2^{10} = 972.8 \approx 973 = 512 + 256 + 128 + 64 + 8 + 4 + 1 =$$ 1111001101 (10 fraction bits)

4. Add exponent bias to log2(W). Represent the resulting unsigned integer as a sum of two's powers.

$$2 + 15 = 17 = 16 + 1 =$$ 10001 (5 exponent bits)

5. The result is: 1 10001 1111001101

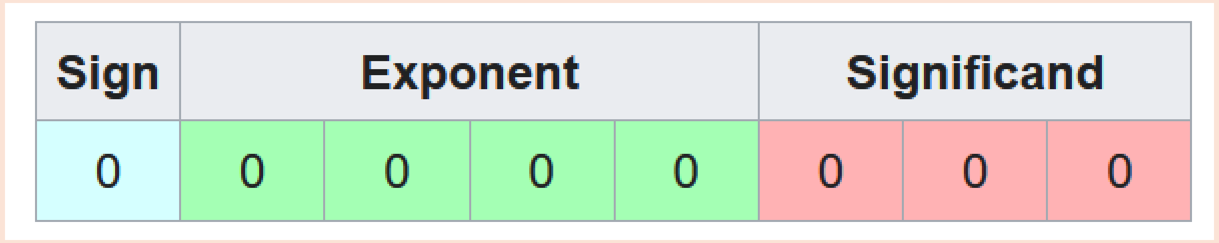
$10001_2 = 17_{10}$

$.1111001101 = 973 / 2^{10} = 973 / 1024 = 0.9501953125$ $\quad -1 \times (1 + 0.9501953125) \times 2^{17-15} \approx -7.80078125$

Quick test

- Floating-point conversions
- Consider minifloat **8-bit (1.4.3)**

See https://en.wikipedia.org/wiki/Minifloat



Sign	Exponent				Significand		
0	0	0	0	0	0	0	0

Exponent bias = 7

- Goto https://presemo.oulu.fi/spslec2 and provide your solutions
  - Or vote for solutions looking right

Custom floating-point

- Considerable amount of logic is needed to support all standard modes
- However, in DSP processor implementation we can have freedom to design our own format & arithmetic

1. Considering modes in the IEEE754 standard
   - **Normal and zero mode can already be sufficient** in a custom format
   - Subnormal mode can complement them, if better dynamic range is needed
   - Special Infinity and NaN cases not needed
2. We can also **tune** the word length and the length of exponent and significand parts
   - Note: Developer must be aware about any possible overflow, underflow, and precision problems!

- **Result**: simplified HW – floating-point becomes more attractive to use in implementations!

Range and precision characteristics

- Let us consider a case, where just the normal and zero modes are implemented:
  - range: the smallest and largest values are $\pm(1 + f_{\max})2^{e_{\max}-e_b}$
  - dynamic range :

$$\frac{(1 + \underbrace{f_{\max}}_{\approx 1})2^{e_{\max}-e_b}}{(1 + \underbrace{f_{min}}_{=0})2^{e_{\min}-e_b}} \approx 2^{e_{\max}-e_{min}+1} = 6.02(e_{\max} - e_{\min} + 1) \text{ dB}$$

  - precision (ulp) depends on the exponent: the LSB of the significand has the weight 2^{-n}, where n is the number of bits in the significand. Therefore, for the exponent value e, the precision is

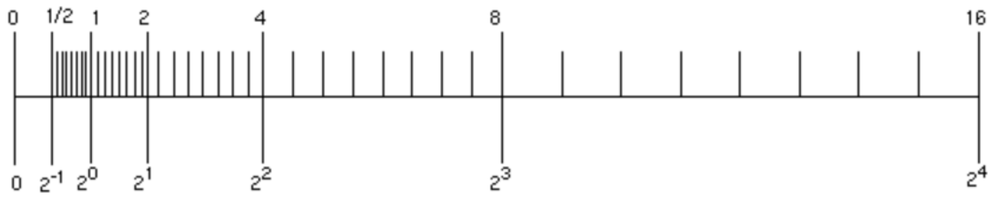
$$\text{ulp}(e) = 2^{e-e_b-n}.$$

$f_{\max}$= maximum significand fraction value
.1111…1 = 1-2^{-n} ≈ 1

e_b= exponent bias

$e_{\max}$= maximum exponent integer value
$e_{\min}$= minimum exponent integer value
- $e_{\min}$= 000..1 = 1 due to zero mode
- $e_{\max}$ can correspond to 111..1 as there are no Inf and NaN modes

Dynamic range = 6.02 × $e_{\max}$ dB



Increased dynamics with subnormal mode

Minimum non-zero significand fraction value is

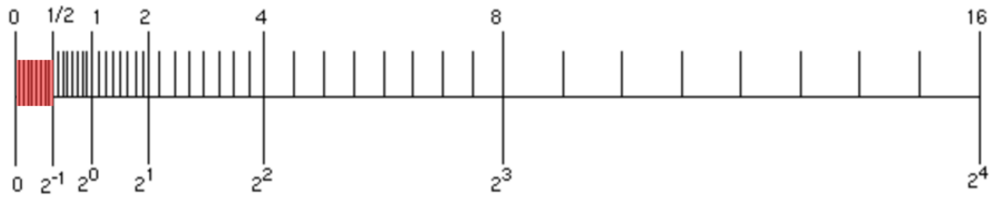
.000…01 = 2^{-n}

=> Smallest representable non-zero value is now $2^{-n}2^{e_{min}-e_b}$

=> $\frac{(1+f_{max})2^{e_{max}-e_b}}{2^{-n}2^{e_{min}-e_b}} \approx \frac{2^{e_{max}-e_b+1}}{2^{e_{min}-e_b-n}} = \frac{2^{e_{max}+1}}{2^{e_{min}-n}} = 2^{e_{max}-e_{min}+1+n} = 6.02(e_{max} - e_{min} + 1 + n)$ dB

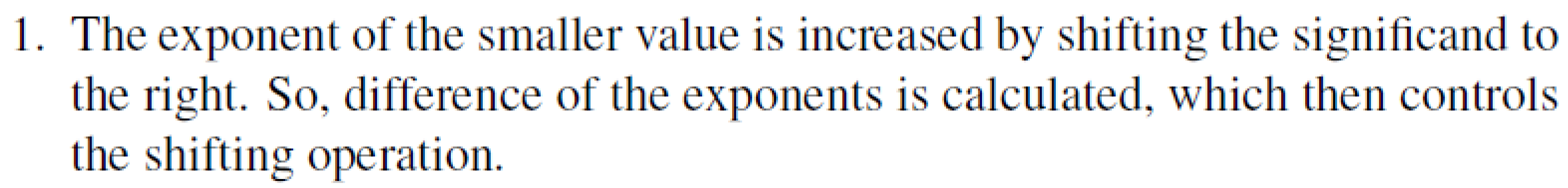
As $e_{min} = 1$

Dynamic range = $6.02 \times (e_{\mathrm{max}}$+n) dB



Arithmetics: Addition in floating-point HW

- 3-step procedure
  1. Make exponents same
  2. Add significands
  3. Normalization
- Note: advantage of sign-magnitude in step 3
- See numeric example in intro1.pdf (Example 12)



1. The exponent of the smaller value is increased by shifting the significand to the right. So, difference of the exponents is calculated, which then controls the shifting operation.

Example. Normal mode number:

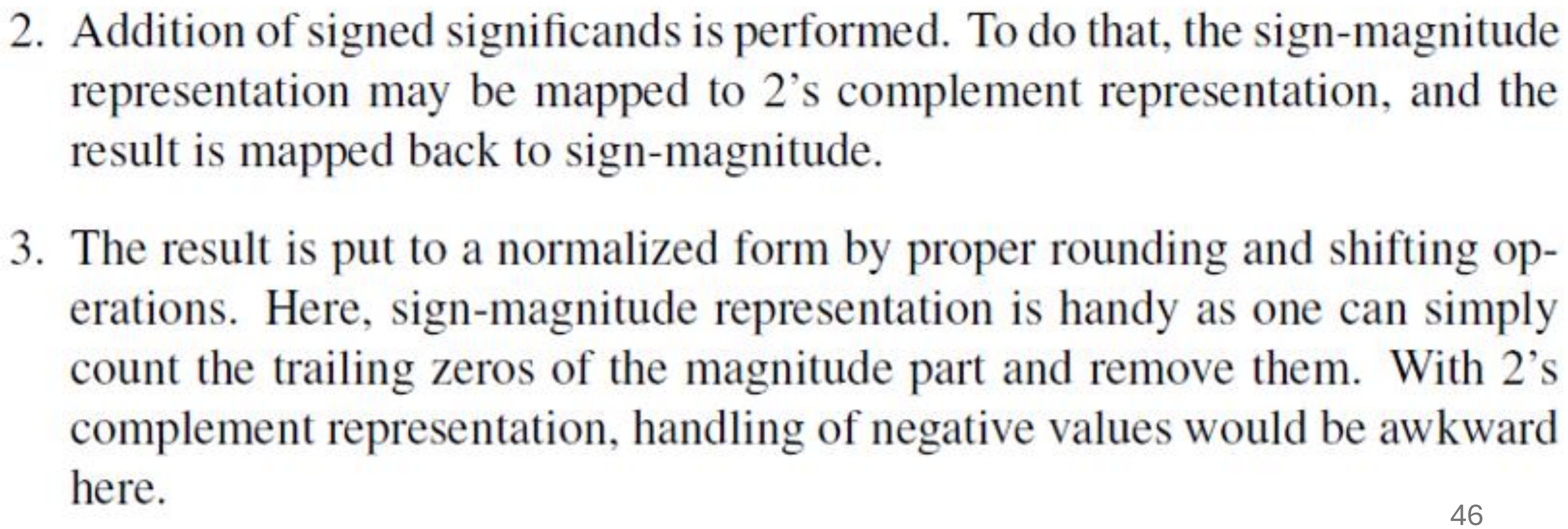
1 00011 0110001101

Equivalent representations for step 1

1 00011 1.0110001101000

= 1 00100 0.1011000110100

= 1 00101 0.0101100011010



2. Addition of signed significands is performed. To do that, the sign-magnitude representation may be mapped to 2's complement representation, and the result is mapped back to sign-magnitude.

3. The result is put to a normalized form by proper rounding and shifting operations. Here, sign-magnitude representation is handy as one can simply count the trailing zeros of the magnitude part and remove them. With 2's complement representation, handling of negative values would be awkward here.

Example. (1.2 + 1.1 + 1.1) == (3 * 1.1 + 0.1) ?

- Matlab returns FALSE (ans = logical 0). Why?

- By default, Matlab uses 64-bit double precision. However, only the number 3 is represented precisely.

- One can map numbers to lower 32-bit precision with ‘single’.
  - (single(1.2) + single(1.1) + single(1.1)) == (single(3) * single(1.1) + single(0.1)) ? TRUE (ans = logical 1) !!!
  - But this is just coincidence, only the value 3 is represented accurately with single precision.

Example. Angle between two vectors

- Computing the angle α between two 3D vectors $\boldsymbol{x_i}$ = (x_i, y_i, z_i). What are numerical issues, when implementation is based on

$$\boldsymbol{x}_1 \cdot \boldsymbol{x}_2 = |\boldsymbol{x}_1||\boldsymbol{x}_2| \cos\alpha \text{ , that is,}$$

$$\alpha = \cos^{-1} \frac{\boldsymbol{x}_1 \cdot \boldsymbol{x}_2}{|\boldsymbol{x}_1||\boldsymbol{x}_2|}$$

$$= \cos^{-1} \frac{\boldsymbol{x}_1 \cdot \boldsymbol{x}_2}{\sqrt{\boldsymbol{x}_1 \cdot \boldsymbol{x}_1}\sqrt{\boldsymbol{x}_2 \cdot \boldsymbol{x}_2}}$$

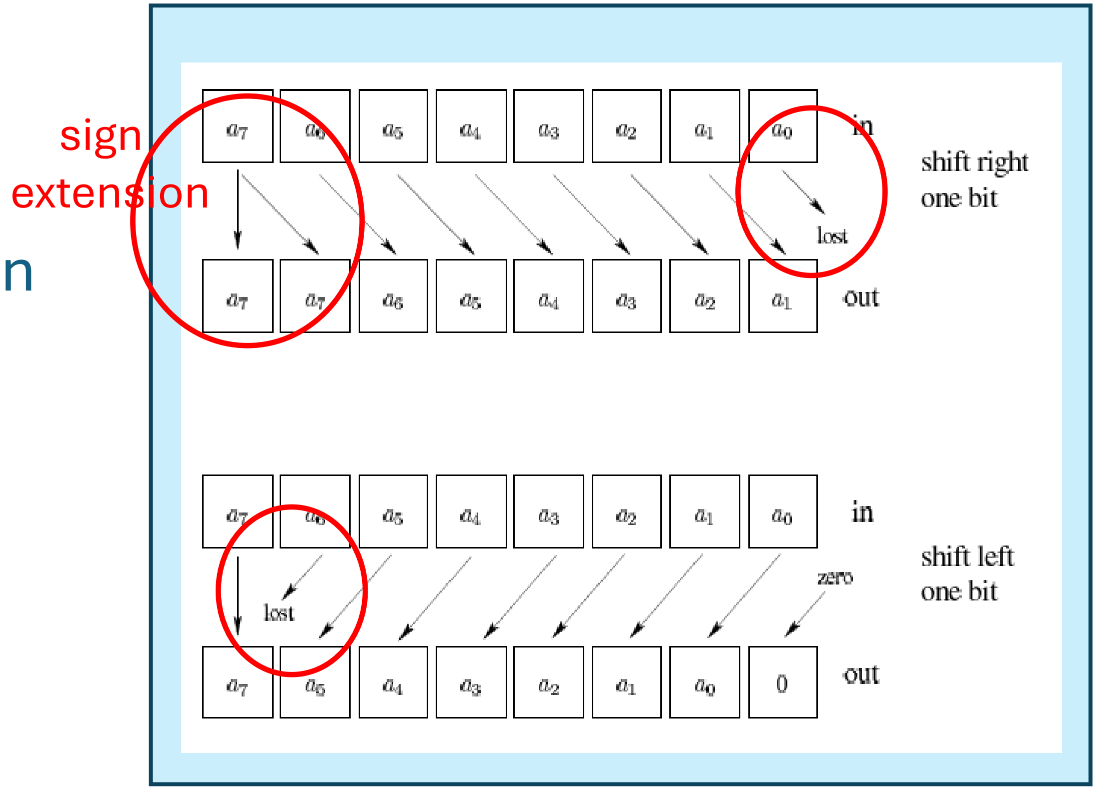
DISCUSSED IN THE LECTURE.

V. Arithmetic

In earlier slides:

- Addition
  - overflow problem
  - wrap-up (modularity of 2’s complement)
  - Protection for wrap-up: guard bits, saturation
- Multiplication
  - law of conservation of bits **s**$p.n$ x **s**$p.n$ => **s**$2p.2n$,
    - Extra integer bit introduced
    - E.g., **s**4.3 x **s**4.3 => **s**8.6, not **s**7.6
  - multiplication by 2^k can be implemented using arithmetic shift

Arithmetic shift to right and left:


sign extension
a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
in
shift right one bit
lost
a_7 a_7 a_6 a_5 a_4 a_3 a_2 a_1
out
a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
in
shift left one bit
lost
zero
a_7 a_5 a_4 a_3 a_2 a_1 a_0 0
out

Reducing output word length

Possibilities for reducing output word length

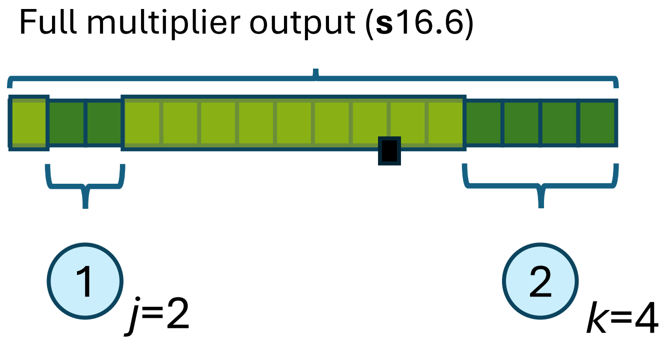
Case: Multiplying two **s***p*.*n* numbers (output: **s**2*p*.2*n*)

1. Multiplier can have a saturation property
   - Discarding most significant bits coming after the sign bit
   - The output format is **s(**2*p*-*j)*.2*n* where *j* > 0
   - Output bits may not be exactly copy of the input

2. Least significal bits may be discarded by rounding
   - Such operation may not reduce signal quality too much
   - The output format is **s(**2*p*-*k)*.(2*n*-*k)* where *k* > 0
   - Output bits may not be exactly copy of the input

- Combined => output format **s**(2*p*-*j*-*k*).(2*n*-*k*)

s8.3 *p*=8, *n*=3



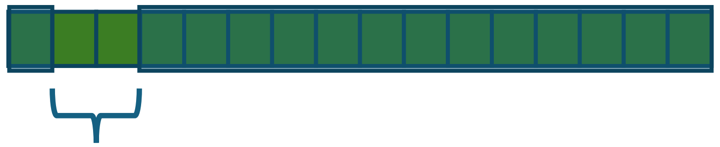
Combined =>**s**(16-2-4).(6-4) = **s**10.2

Saturation

Saturation in 2’s complement arithmetic – how it works?

Discarding *j* MSB bits coming after the sign bit

If the input is **s***p*.*n* , the output is **s(***p-j***)**.*n*



Saturation is not just simply discarding bits. Example:

11101101 = -19_{10} 8-bit word, range -128 .. +127

Discard 1 bit: 1~~1~~101101 = 1101101 = -19_{10} 7-bit word, range -64 .. +63

Discard 2 bits: 1~~11~~01101 = 101101 = -19_{10} 6-bit word, range -32 .. +31

Discard 3 bits: 1~~110~~1101 = 11101 = -3_{10} 5-bit word, range -16 .. +15

-3 is not correct, the result should be -16.

Now the last discarded bit is not equal to the sign.

Then, the saturation logic must output 10000 = -16_{10}

Discard 4 bits: 1~~1101~~101 = 1101 = -3_{10}

Discarding one bit not equal to the sign.

Saturation logic must output 1000 = -8_{10}

For a positive number: 00010011 = +19

Discard 3 bits => 01111 = +15

Saturation properties (1)

- saturating operations are not associative
  - example, next slide

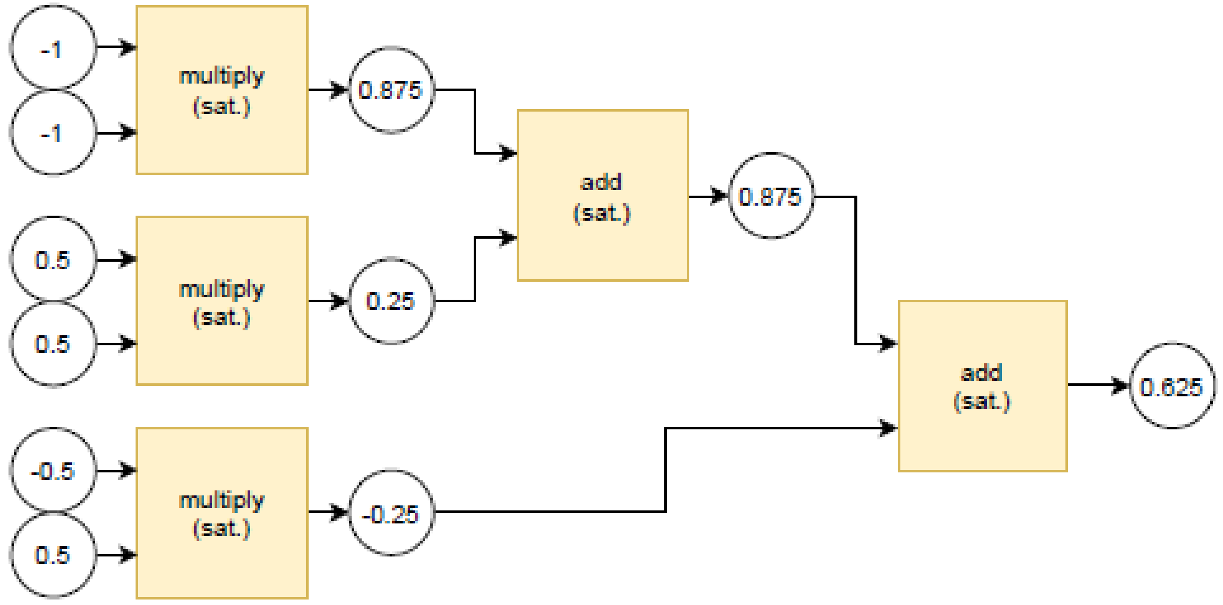
Associativity of operation (op):

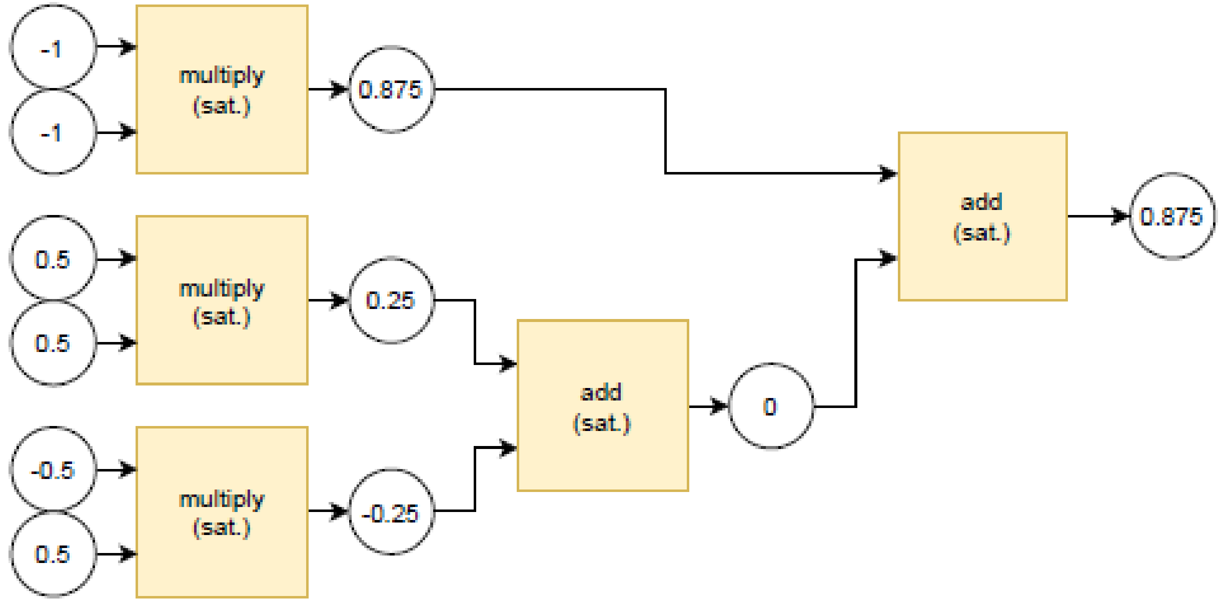
(A op B) op C = A op (B op C)

=> some standards for algorithms specify **exact order** of operations

- results become predictable
- important because helps testing that an implementation fulfills the standard

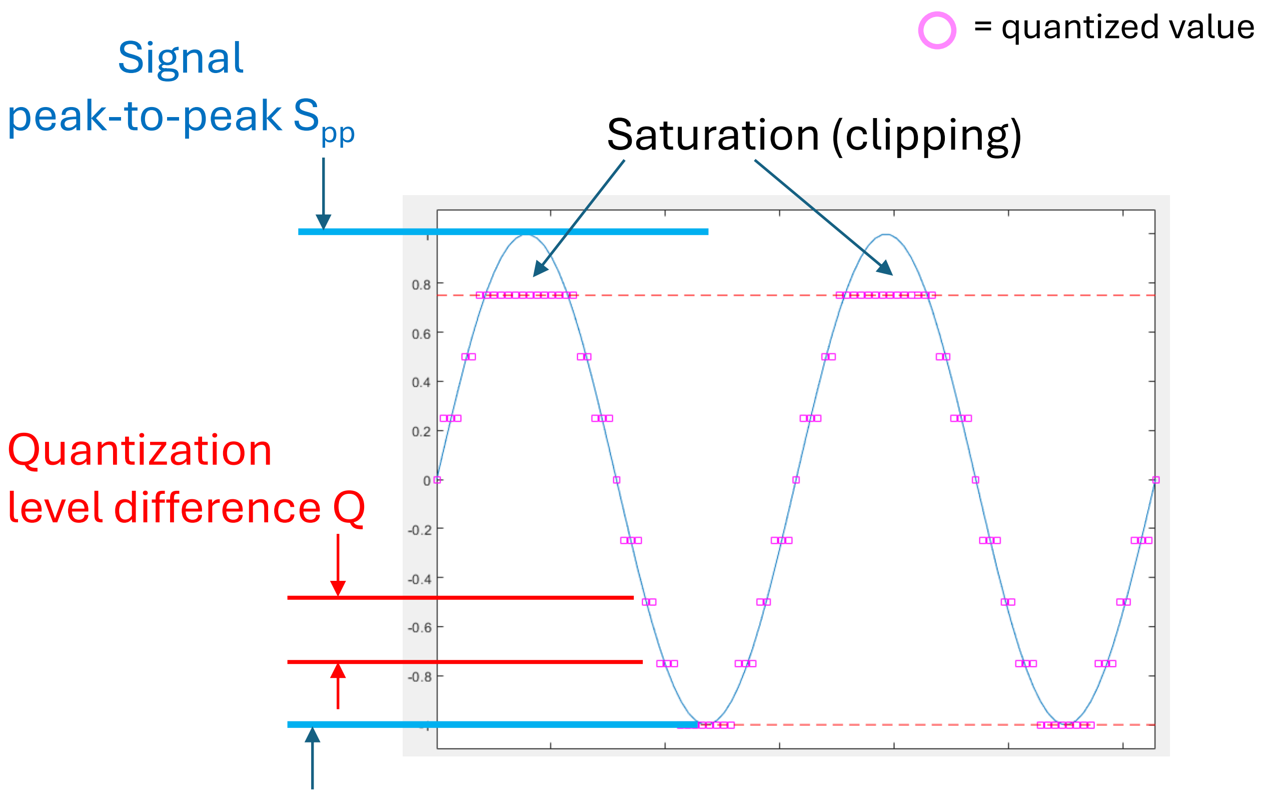
Example. Saturating addition of 3 saturating multiplication results, inputs & outputs s4.3 (i.e., range -1.000 … +0.875)

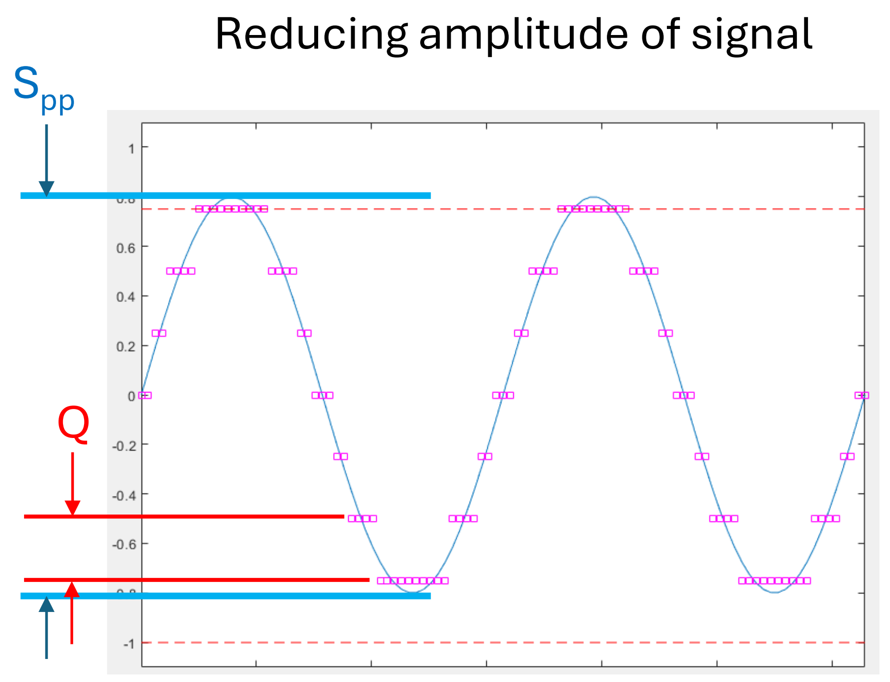




Saturation properties (2)

- Saturation **adds noise** to the signal (clipping)
  - Tradeoff: saturation noise / quantization noise
- Example: quantizing sine signal for s3.2





Less saturation noise, but lower SQNR S_{pp} / Q

Rounding

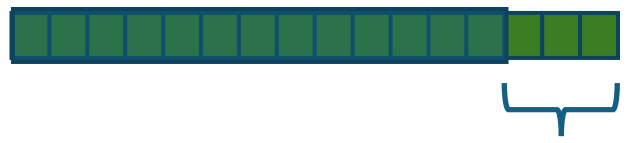
Rounding

- Rounding drops least significant bits
- Some rounding technique used
  - discarded bits may have effect on the final result

- Six techniques can be recognized for rounding
  - IEEE 754 floating point standard recognizes five
  - Techniques have varying implementation complexity
  - Techniques differ in **statistical bias properties**

- Truncation (floor) is the simplest
  - Does not care about discarded bits
  - Very biased operation

- Convergent (ties-to-even) rounding least biased
  - Default mode in IEEE754 floating-point standard

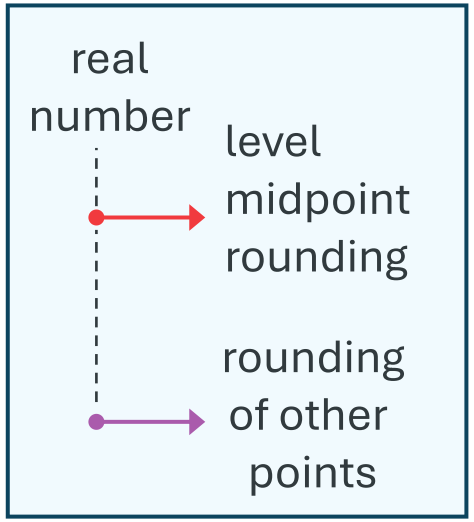


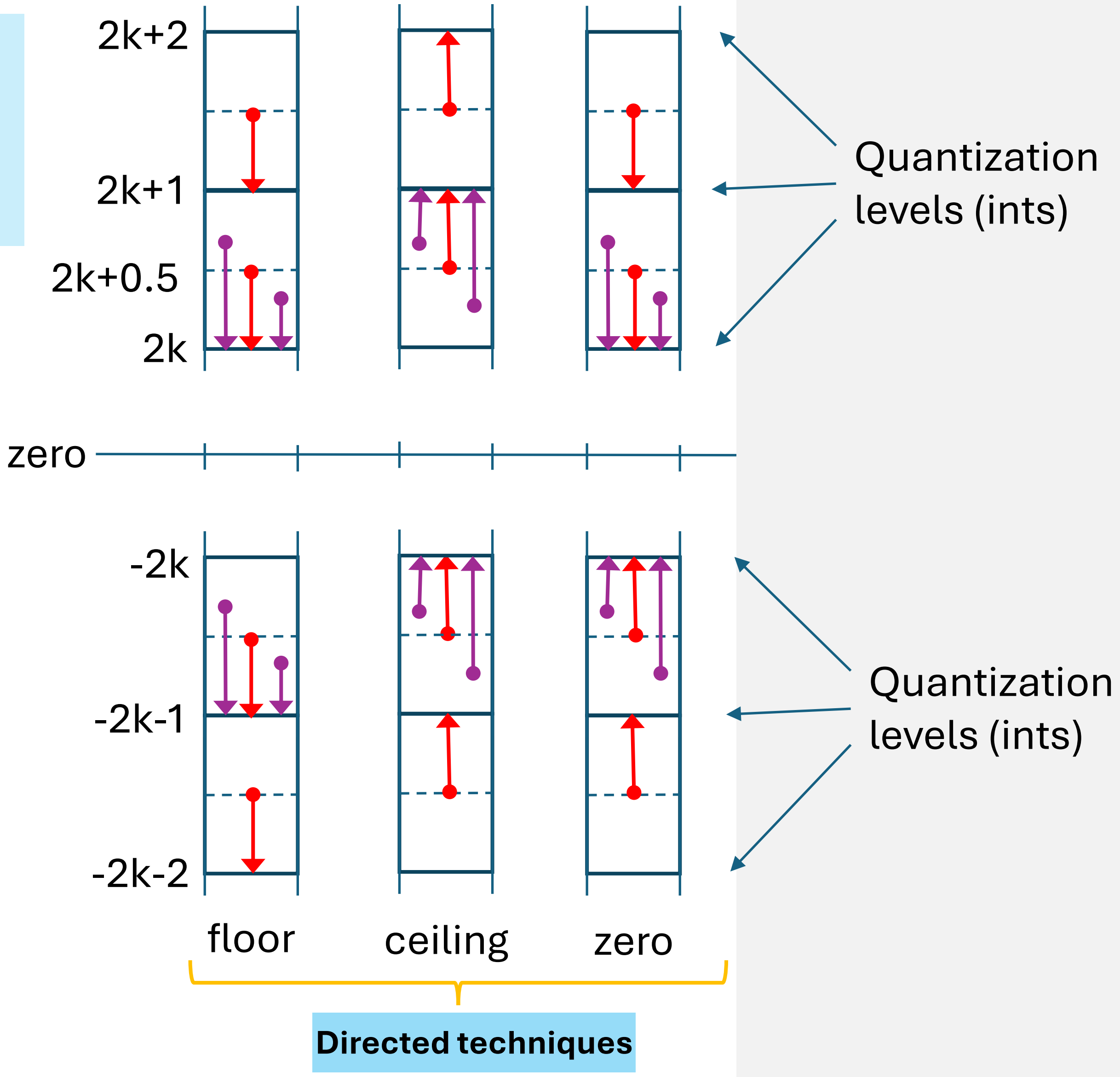
Note: Related to implementation, IEEE 754 guides the implementation by defining three bits (**Guard, Round, Sticky) which control the rounding operation.**

These are called ”three guard bits”. Don’t confuse with guard bits related to accumulator overflow prevention.

Types of rounding operations

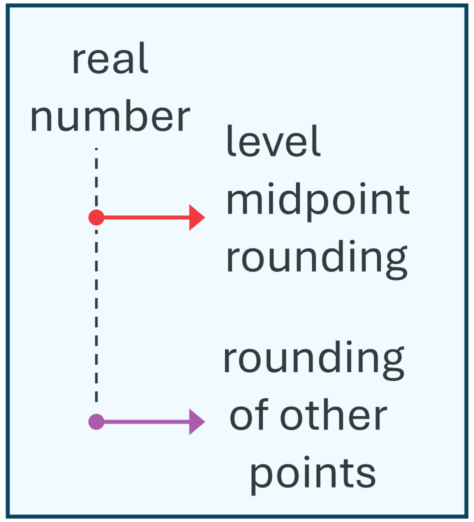
In this illustration, **rounding of real numbers to integers,** positive and negative numbers

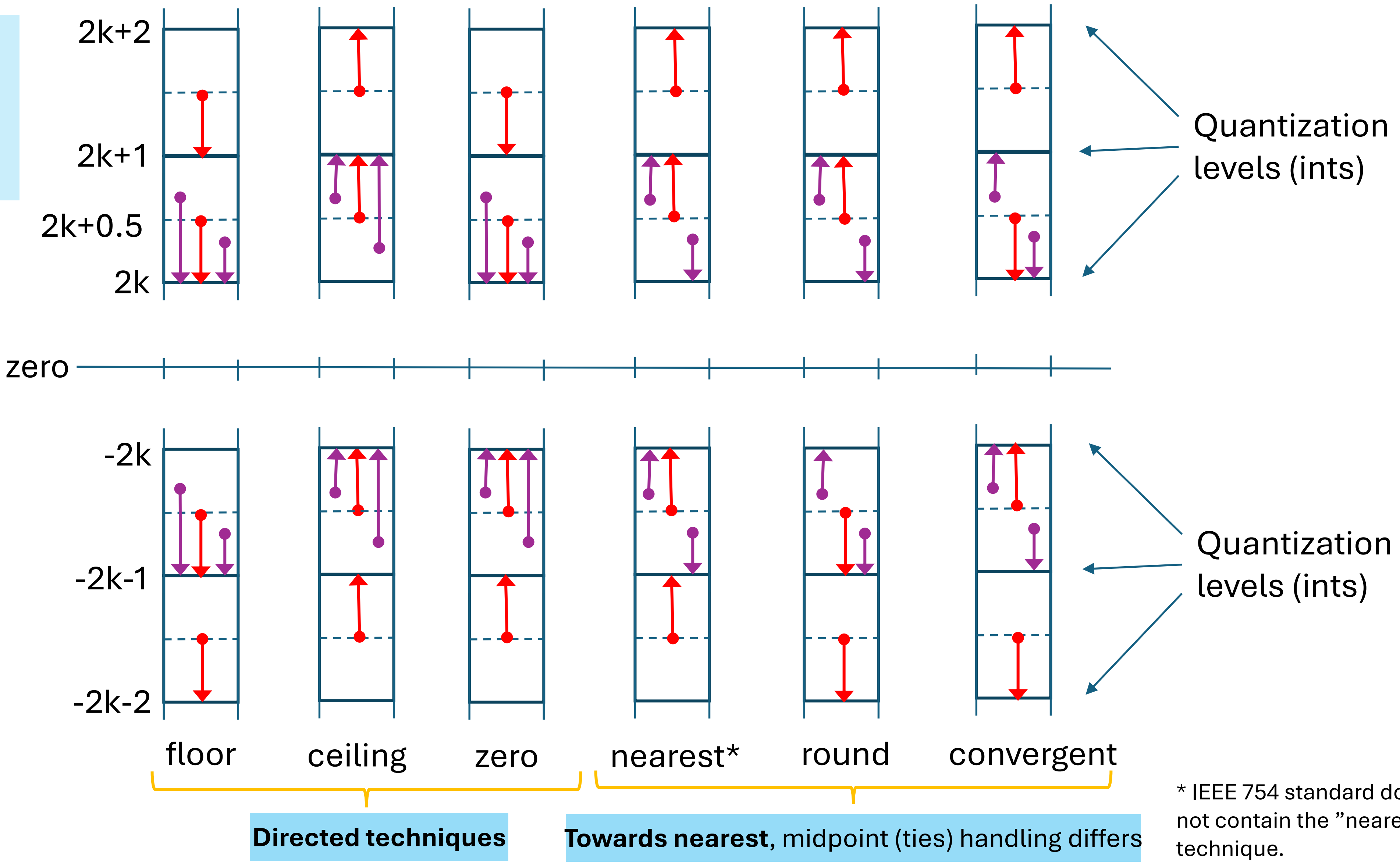




Types of rounding operations

In this illustration, **rounding of real numbers to integers,** positive and negative numbers



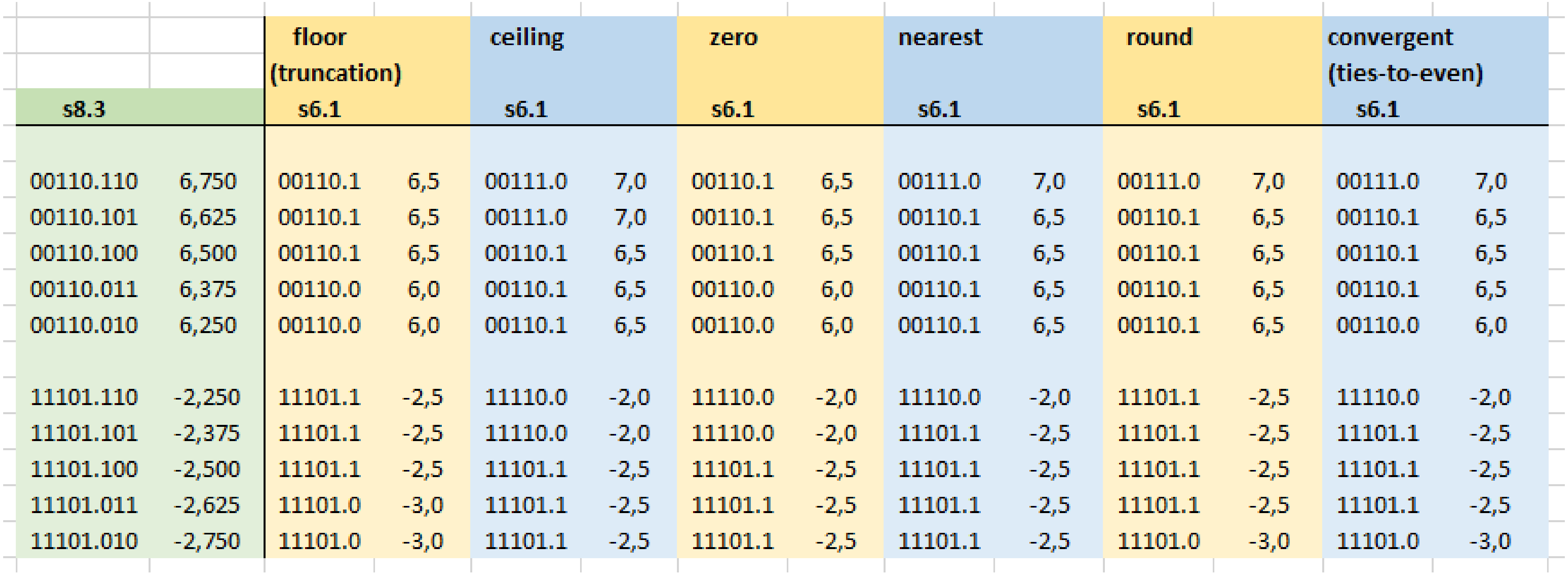


2k+2
2k+1
2k+0.5
2k
zero
-2k
-2k-1
-2k-2
Quantization levels (ints)
Quantization levels (ints)
floor
ceiling
zero
nearest*
round
convergent
Directed techniques
Towards nearest, midpoint (ties) handling differs

* IEEE 754 standard does not contain the "nearest" technique.

Example. Rounding fixed-point numbers

Rounding from s8.3 format to s6.1 (0.5 difference in quantization levels)

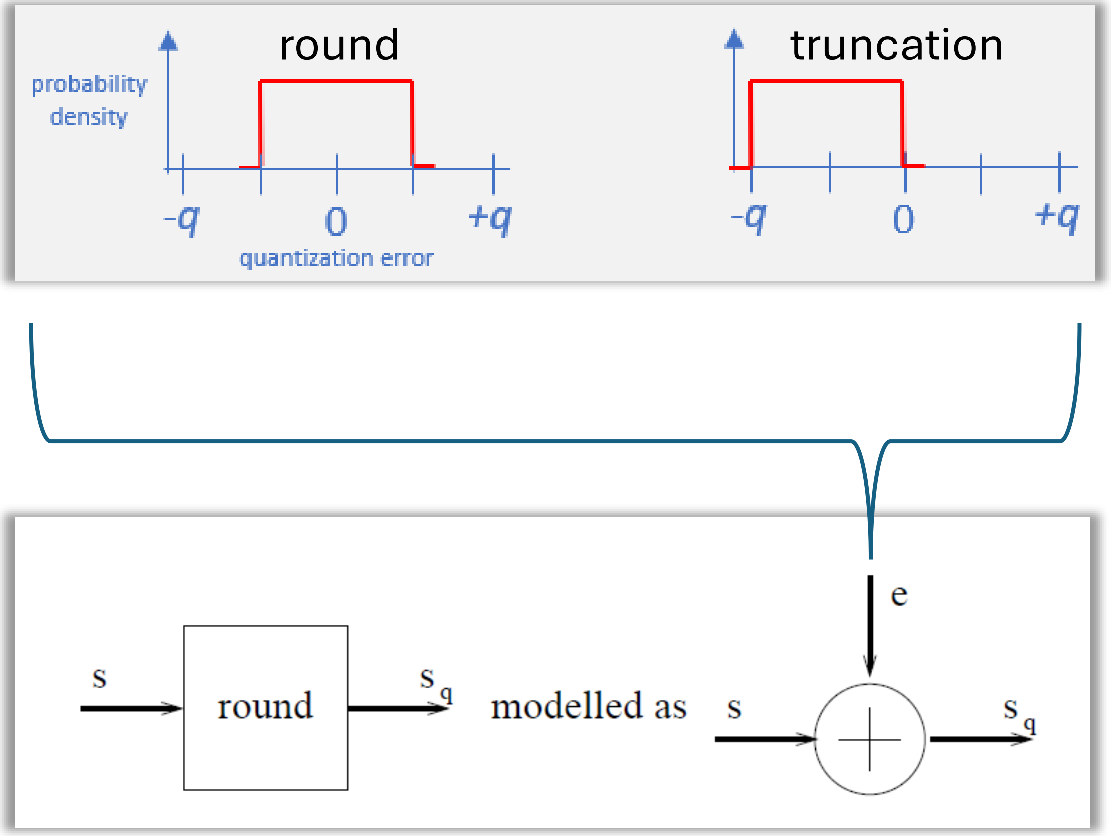


		floor (truncation)		ceiling		zero		nearest		round		convergent (ties-to-even)	
s8.3		s6.1		s6.1		s6.1		s6.1		s6.1		s6.1	
00110.110	6,750	00110.1	6,5	00111.0	7,0	00110.1	6,5	00111.0	7,0	00111.0	7,0	00111.0	7,0
00110.101	6,625	00110.1	6,5	00111.0	7,0	00110.1	6,5	00110.1	6,5	00110.1	6,5	00110.1	6,5
00110.100	6,500	00110.1	6,5	00110.1	6,5	00110.1	6,5	00110.1	6,5	00110.1	6,5	00110.1	6,5
00110.011	6,375	00110.0	6,0	00110.1	6,5	00110.0	6,0	00110.1	6,5	00110.1	6,5	00110.1	6,5
00110.010	6,250	00110.0	6,0	00110.1	6,5	00110.0	6,0	00110.1	6,5	00110.1	6,5	00110.0	6,0
11101.110	-2,250	11101.1	-2,5	11110.0	-2,0	11110.0	-2,0	11110.0	-2,0	11101.1	-2,5	11110.0	-2,0
11101.101	-2,375	11101.1	-2,5	11110.0	-2,0	11110.0	-2,0	11101.1	-2,5	11101.1	-2,5	11101.1	-2,5
11101.100	-2,500	11101.1	-2,5	11101.1	-2,5	11101.1	-2,5	11101.1	-2,5	11101.1	-2,5	11101.1	-2,5
11101.011	-2,625	11101.0	-3,0	11101.1	-2,5	11101.1	-2,5	11101.1	-2,5	11101.1	-2,5	11101.1	-2,5
11101.010	-2,750	11101.0	-3,0	11101.1	-2,5	11101.1	-2,5	11101.1	-2,5	11101.0	-3,0	11101.0	-3,0

N.B. The means on rounded values are 6.5 and -2.5 for **convergent** rounding.
Nearest and **round** have slight bias, towards negative and positive direction, respectively
Directed techniques (floor, ceiling, and round) have even larger biases.

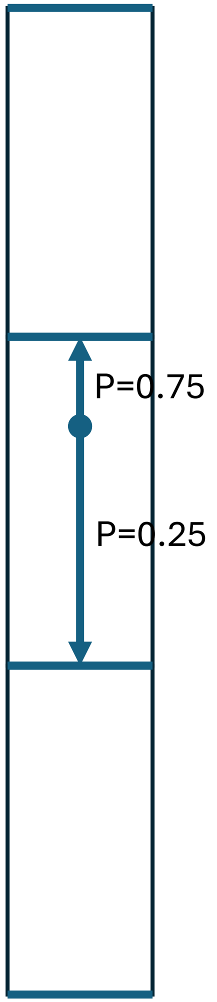
Modelling rounding for analysis

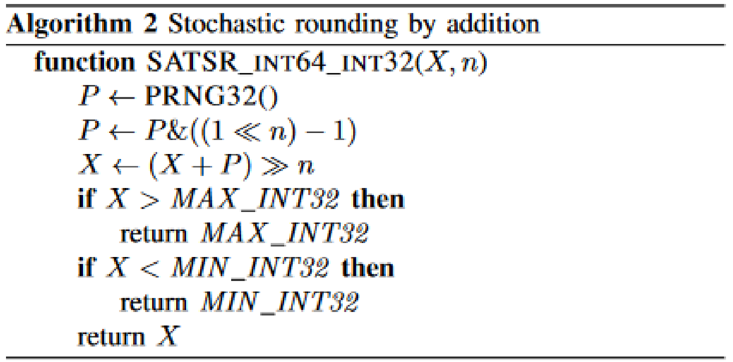
- Rounding is a nonlinear operation
- In algorithm analysis, a statistical model can be used
- In the case of ”round” operation
  - Uniform pdf over the range [–$q/2$, +$q/2$], where q is the quantization step size
  - Noise power $q^2/12$
- If operation like ”truncation” (floor) is used, the bias $q/2$ must be included in the model
- In analysis, an error signal (e) is added to the signal (s) at the signal quantization point
- In filter structures, the quantization noise gets amplified through noise transfer function
  - Deriving it is based on what path the noise signal takes through the filter
  - More on this later



Correlation problem

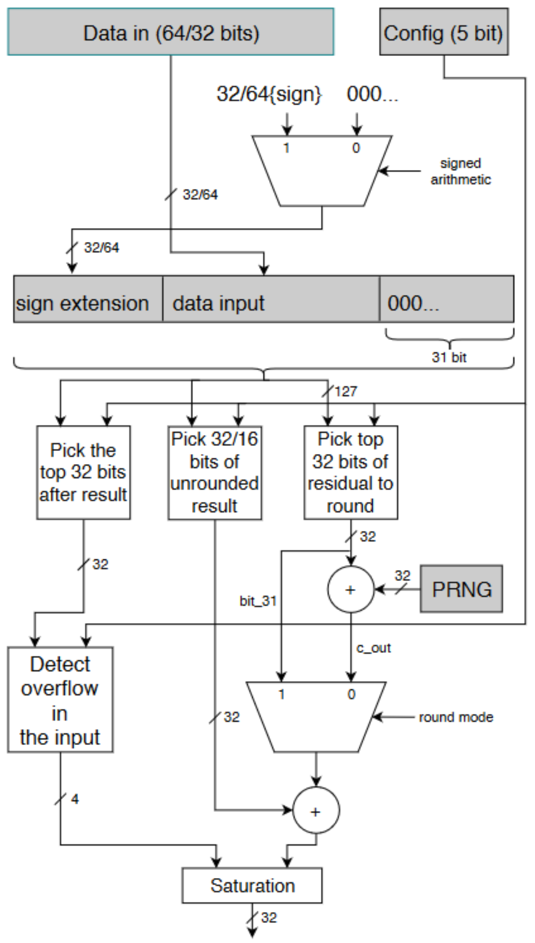
- Previous six methods are deterministic
- **Paper**: M. Mikaitis *”Stochastic rounding: algorithms and hardware acceleration”*. arXiv:2001.01501v4 (2020).
  - Rounding which utilizes pseudorandom number generator (PRNG)
- **Idea**: Depending on the distance from the value to the closest two quantization levels, used level is selected probabilistically
- **Why interesting?** When multiple quantizations are done, **the quantization errors may be correlated** and this correlation can have bad effect on the final result.
- Stochastic rounding could prevent such correlations





Algorithm 2 Stochastic rounding by addition

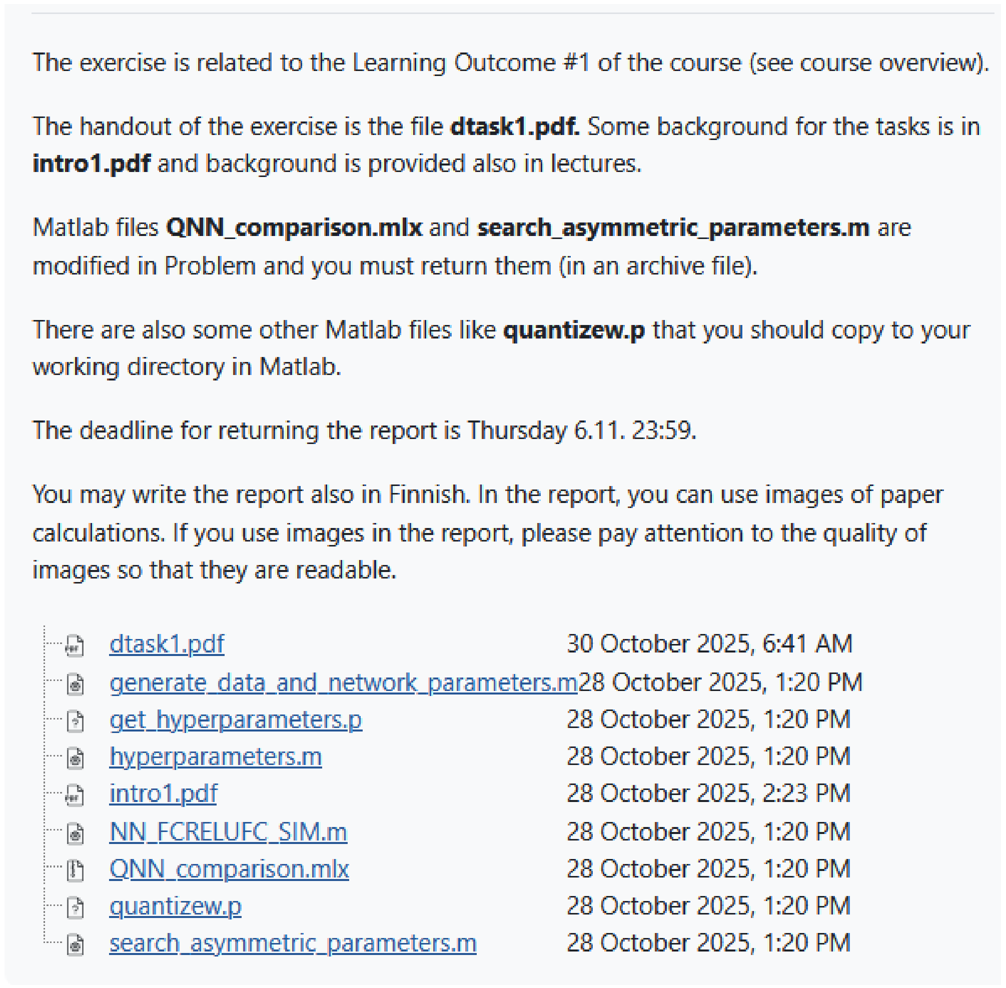
```
function SATSR_INT64_INT32($X, n$)
    $P \leftarrow$ PRNG32()
    $P \leftarrow P\&((1 \ll n) - 1)$
    $X \leftarrow (X + P) \gg n$
    if $X > MAX\_INT32$ then
        return $MAX\_INT32$
    if $X < MIN\_INT32$ then
        return $MIN\_INT32$
    return $X$
```



(Mikaitis, 2020)

Homework: Design Task 1

- Three problems:
    1. **Fixed-point numbers, scaling**.
    - both binary point & slope-bias cases
    2. **Custom floating point format**.
    - Containing also subnormal mode
    3. **MAC data path & neural network quantization**
    - Will be presented in Monday's lecture
- Different parameters for each group
- Finnish can be used in reports
- **PDF report** expected + some codes in Problem 3
- PDF may contain images of calculations on paper



The exercise is related to the Learning Outcome #1 of the course (see course overview).

The handout of the exercise is the file **dtask1.pdf.** Some background for the tasks is in **intro1.pdf** and background is provided also in lectures.

Matlab files **QNN_comparison.mlx** and **search_asymmetric_parameters.m** are modified in Problem and you must return them (in an archive file).

There are also some other Matlab files like **quantizew.p** that you should copy to your working directory in Matlab.

The deadline for returning the report is Thursday 6.11. 23:59.

You may write the report also in Finnish. In the report, you can use images of paper calculations. If you use images in the report, please pay attention to the quality of images so that they are readable.

dtask1.pdf	30 October 2025, 6:41 AM
generate_data_and_network_parameters.m	28 October 2025, 1:20 PM
get_hyperparameters.p	28 October 2025, 1:20 PM
hyperparameters.m	28 October 2025, 1:20 PM
intro1.pdf	28 October 2025, 2:23 PM
NN_FCRELUFC_SIM.m	28 October 2025, 1:20 PM
QNN_comparison.mlx	28 October 2025, 1:20 PM
quantizew.p	28 October 2025, 1:20 PM
search_asymmetric_parameters.m	28 October 2025, 1:20 PM