COMPUTER ARCHITECTURE

5 Operands

5 Operands - objectives:

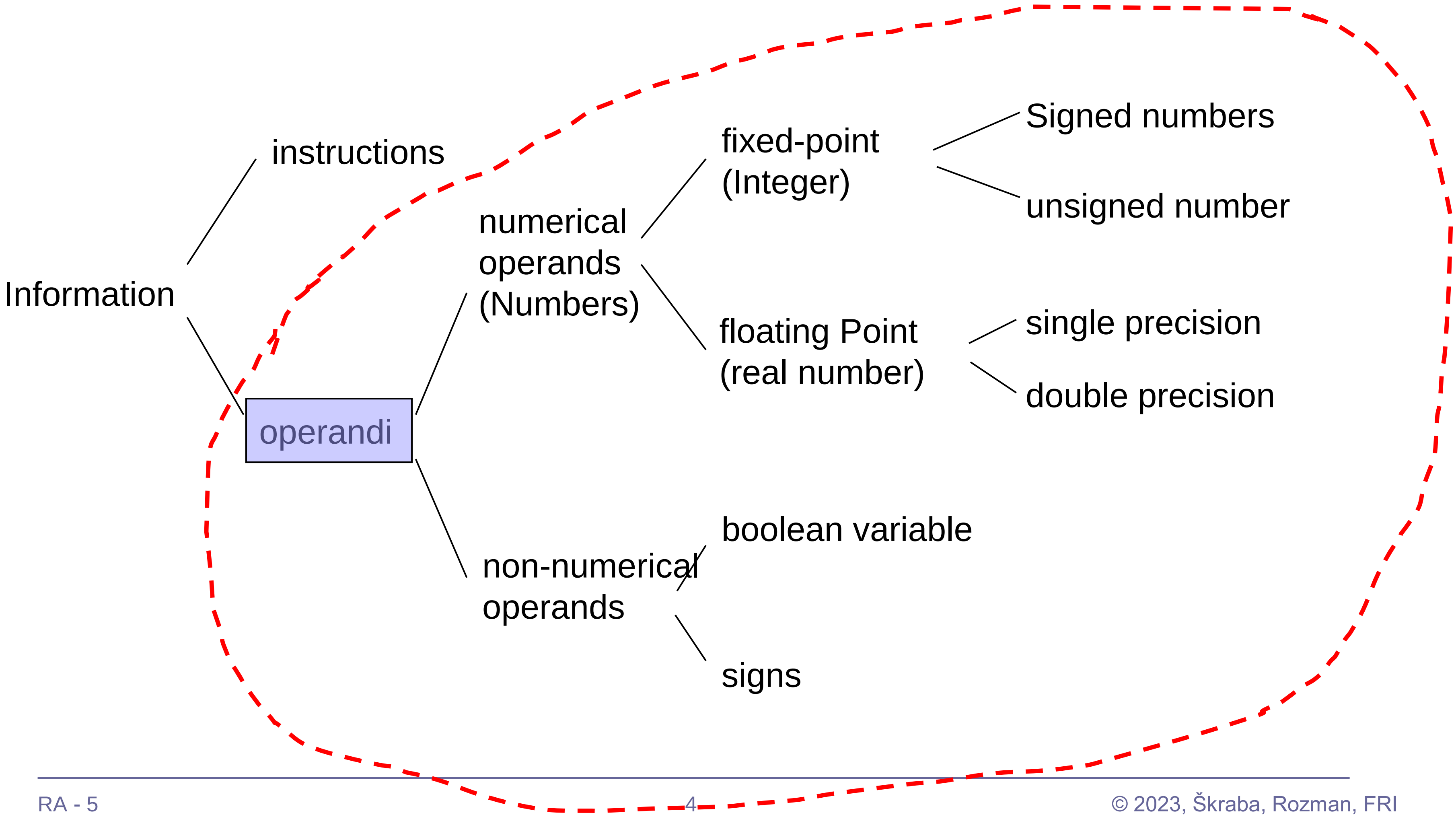
- Understanding the different formats of operands
  - Alphabets (characters)
  - Numbers in fixed-point format (unsigned, signed – two‘s complement)
  - Real numbers in floating point

- Understanding the implementation of the basic operations on operands
  - Strengths, weaknesses of formats, presentations, ...
  - The importance of the proper execution of operations

5 Operands - contents

- Presentation of non-numerical operands
  - ASCII alphabet
  - UNICODE alphabet
- Presentation of numerical operands in fixed-point format
  - Types of presentation
  - Carry and overflow
  - Example-1
- Arithmetic numbers in fixed-point arithmetic
- Presentation of numerical operands in floating point
  - The general form
  - Standard for the presentation of floating point
  - Basic features of IEEE 754
  - Example-2
- Arithmetic numbers in floating point
- Supplementing the IEEE Standard 754-2008

Basic types of information on your computer



Example of 32-bit content:

1110 0000 1000 0000 0101 0000 0000 0001 (bin) = E0805001 (hex)

- Occupies the 8-bit memory 4 successive memory words, and may represent:
  - □ Machine command (ARM 9): add r5, r0, r1 / * R5 ← R0 + R1
  - □ Unsigned integer: 3766505473
  - □ Integer with sign (two's complement): - 528461823
  - □ Real number in floating point (single precision): $- 73.967 * 10^{18}$

    exact: $- 73.96712907602604851 2 * 10^{18}$
  - □ Four characters in ASCII alphabetical order: r undefined sign P NUL
  - □ Many other

5.1 Introduction non-numerical operands

- Non-numerical operands
  - Characters
  - Strings – sequences of characters
  - Character is represented by an alphabet

Why use Unicode if your program is English only?
The company I work for, as a policy, will only release software in English, even though we have customers throughout the world.

What if I want to store a customer name which uses non-english characters? Or the name of a place in another country?

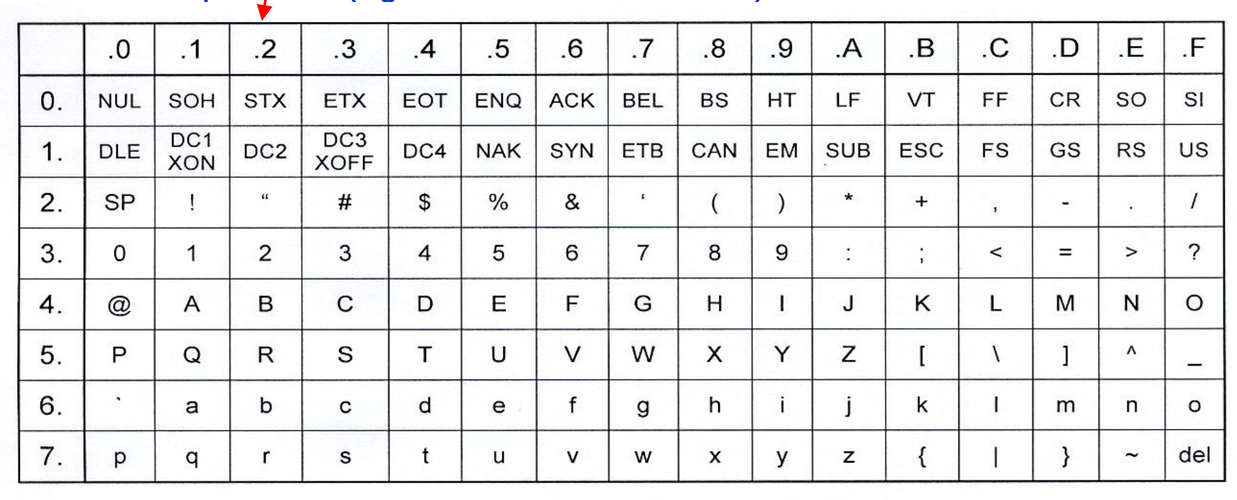
- Alphabet is a rule which provides the mapping of elements of one set to the elements of the second set.

Types of alphabets used in computers

- **BCD alphabet**
  - 6-bit (2^6 = 64 different characters)
  - 26 letters of the English alphabet, 10 digits, 28 special characters
  - In use until 1964. (6-bit words)
- **EBCDIC alphabet** (8-bit)
  - Used primarily from IBM in mainframe systems (IBM 1963/64 System/ 360 →)
- Today, the use of 8-bit ASCII, and 8-bit and more alphabet Unicode (UCS) is quite common.

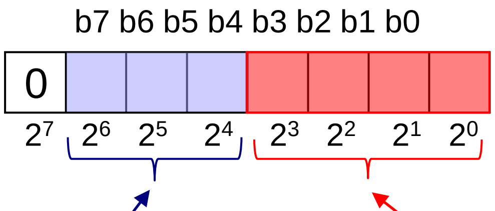
Presentation of non-numeric operands - Characters

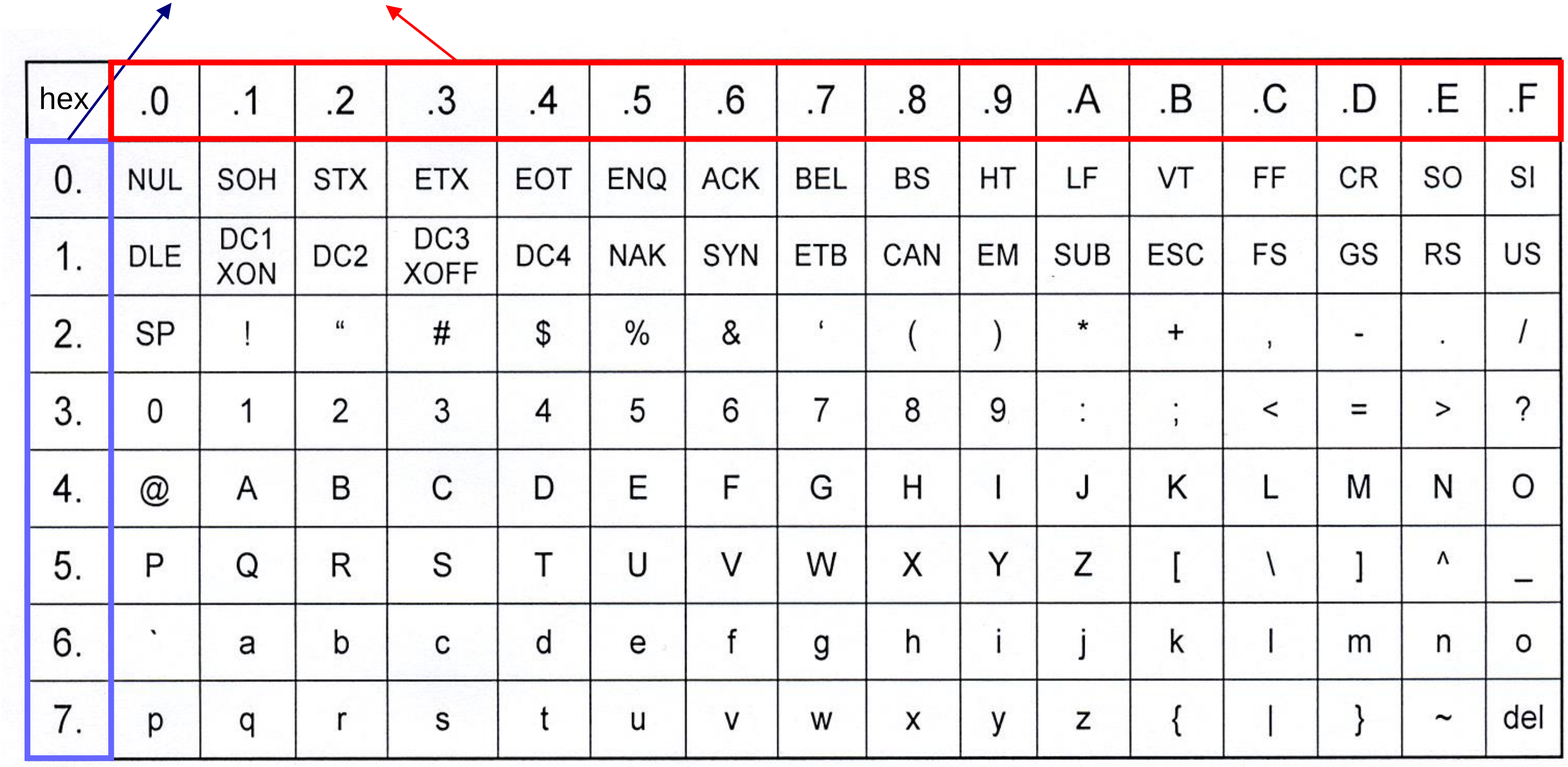
- **ASCII alphabet** (8-bit)
  - □ Originally 7-bit, but today computers use 8-bit format
    - Bit 7 = 0 - original form
    - Bit 7 = 1 - extended ASCII alphabet, the additional 128 characters (IBM PC)
      - □ An additional 128 characters are different for different countries to form national ASCII alphabets (eg. Latin2 = ISO 8859-2)



	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1.	DLE	DC1 XON	DC2	DC3 XOFF	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

The basic 7-bit ASCII alphabet
bit7 = 0



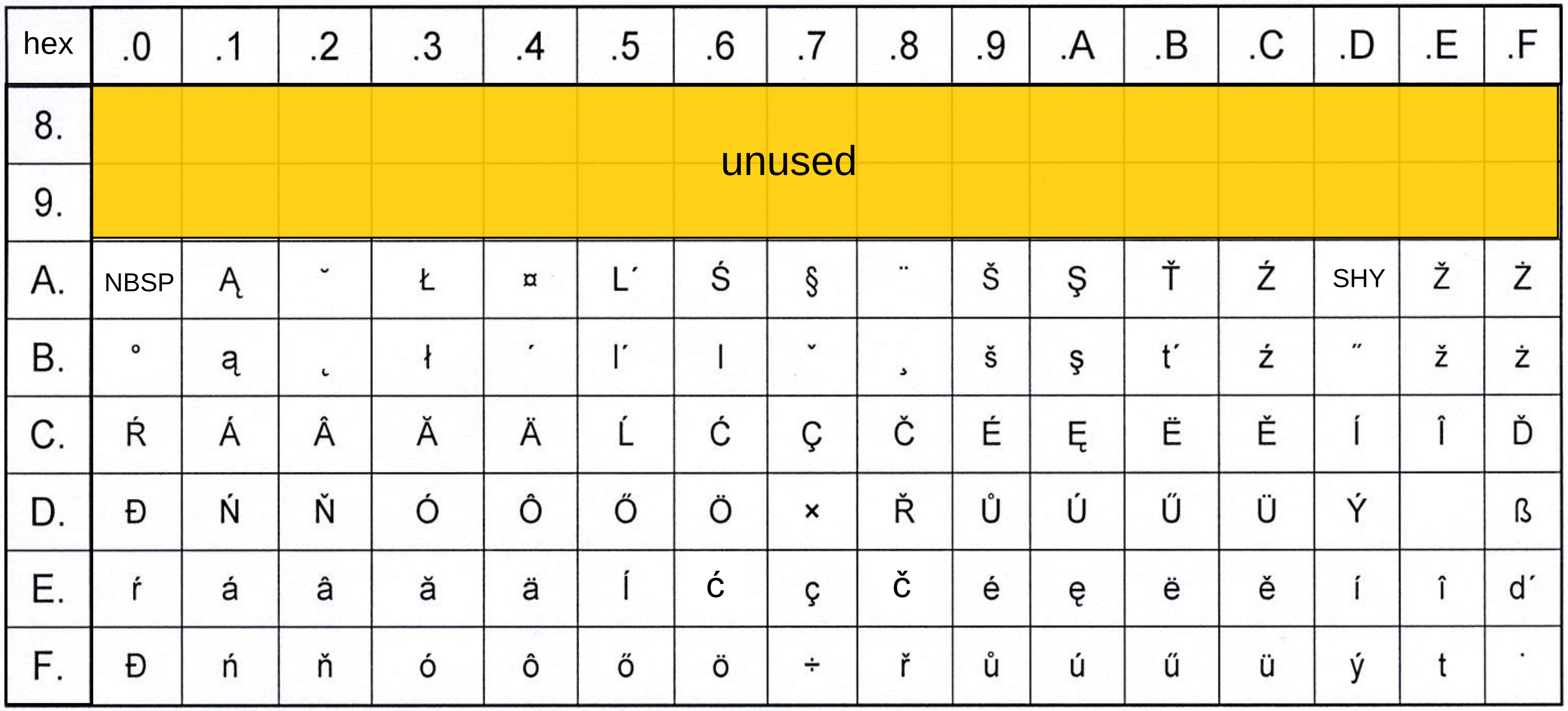


hex	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1.	DLE	DC1 XON	DC2	DC3 XOFF	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

Extended 8-bit ASCII alphabet Latin2
(ISO 8859-2), - the additional characters (b7 = 1)

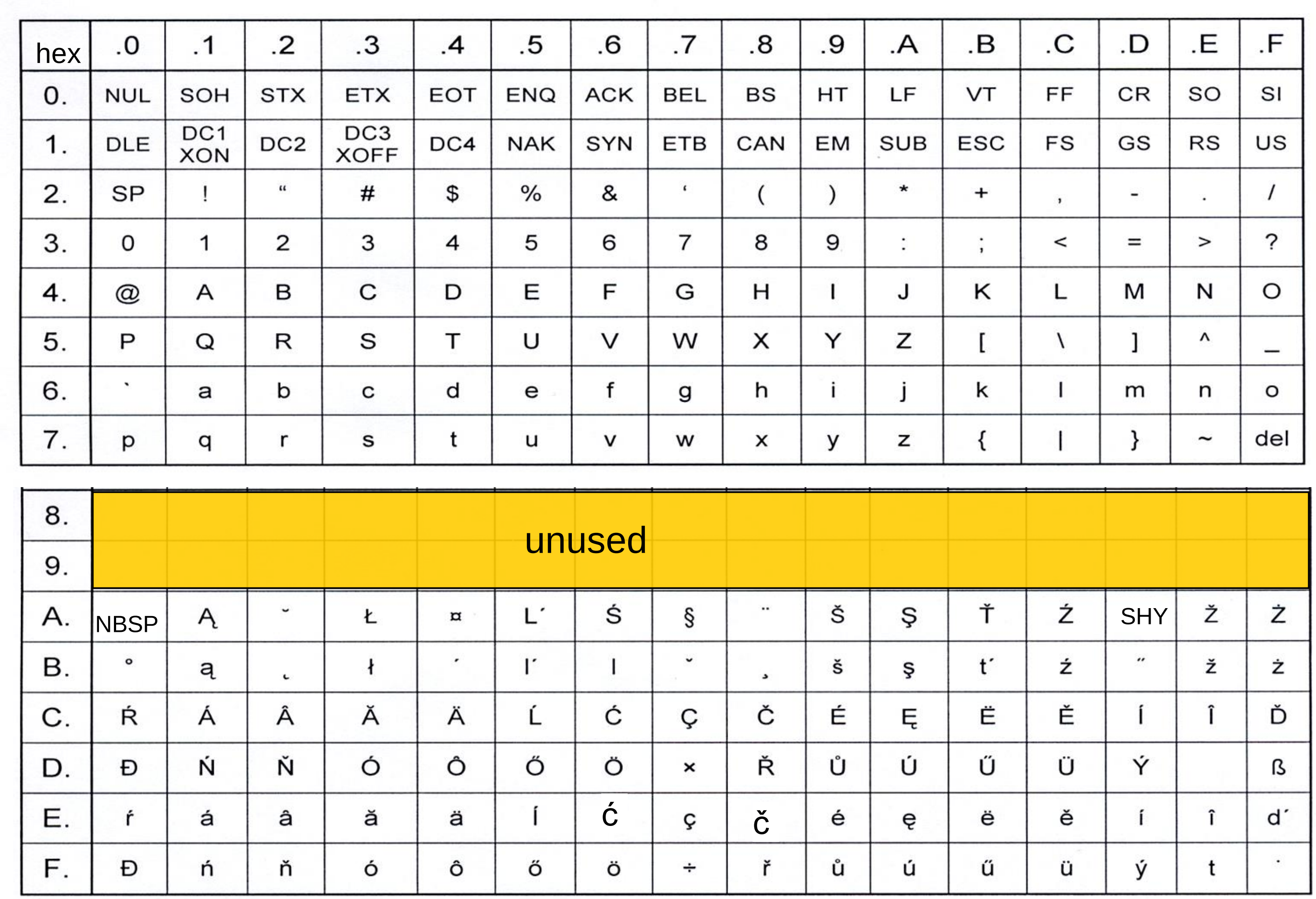
b7	b6	b5	b4	b3	b2	b1	b0
1							
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

NBSP = A0 (hex) Non Breaking Space
SHY = AD (hex) Soft HYPHEN



hex	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
8.	unused															
9.																
A.	NBSP	Ą	˘	Ł	¤	L´	Ś	§	¨	Š	Ş	Ť	Ź	SHY	Ž	Ż
B.	°	ą	˛	ł	´	l´	l	ˇ	¸	š	ş	t´	ź	˝	ž	ż
C.	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
D.	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý		ß
E.	ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	d´
F.	Đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	t	˙

Extended 8-bit ASCII alphabet Latin2 (ISO 8859-2)

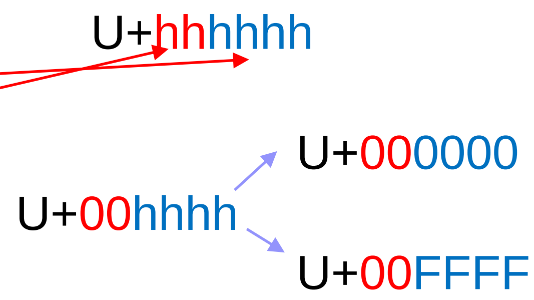


hex	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1.	DLE	DC1 XON	DC2	DC3 XOFF	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del
8.	unused															
9.																
A.	NBSP	Ą	˘	Ł	¤	L´	Ś	§	¨	Š	Ş	Ť	Ź	SHY	Ž	Ż
B.	°	ą	˛	ł	´	l´	l	ˇ	¸	š	ş	t´	ź	˝	ž	ż
C.	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
D.	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý		ß
E.	ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	d´
F.	Đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	t	˙

Unicode – UCS alphabet (standard ISO 10646)

- >= 8-bit: it allows the presentation of the characters in practically all known languages : (17 × 2^{16}) − 2^{11} = 1112064 different characters.

- UCS planes (Universal Coded Character Set):
  - hhhh subsets with 2^{16} characters
  - hh upper 16 bits represent plane



U+hhhhhh

U+00hhhh

U+000000

U+00FFFF

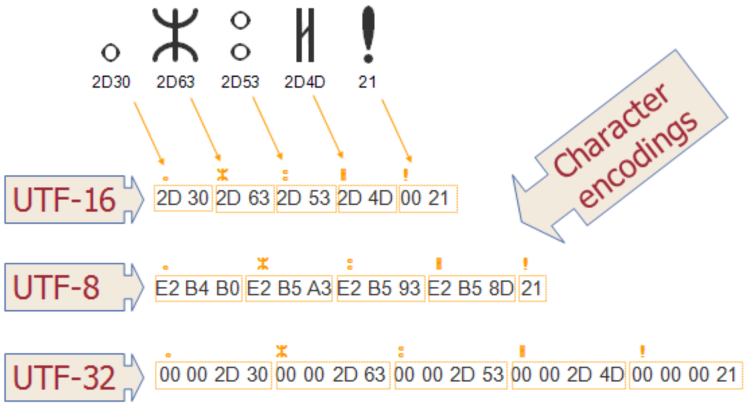
- BMP (Basic Multiling0ual Plane) or Plane 0:
  - Most frequently used characters with all the older standards
  - 0x000000 .. 0x00FFFF; usually shortened to U+hhhh.

- UCS provides each character code and the official name
  - Hexadecimal number (UCS or Unicode code), has prefix U+
    - e.g. *U+0041 for character A (Latin capital letter A)*
    - Unicode Utilities: Character Properties

Presentation of non-numeric operands - Unicode

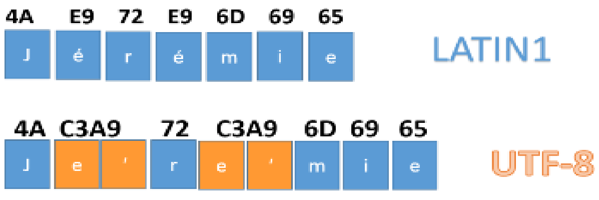
There are several types of transformations for the presentation of characters with a sequence of bytes, for example: UTF-8 and UTF-16 (UTF – UCS Transformation Format).

- □ 2 cases:
  - ■ **UTF-16** (Windows, Java)
    - □ One character occupies 2 bytes (or 4)
    - □ Variable order of bytes (big/little endian)
  - ■ **UTF-8** (www, E-mail)
    - □ Variable length of 1 to 4 bytes
    - □ Compatible with the 7-bit ASCII alphabet (128 chars.)

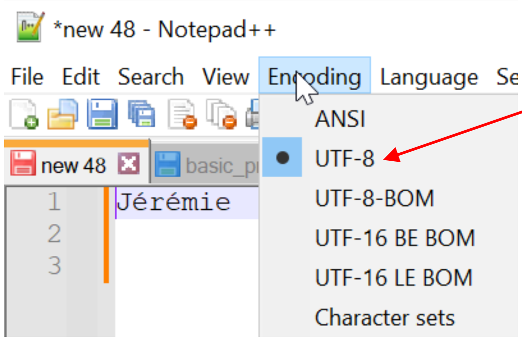

ⴰ ⵣ ⵓ ⵍ !
2D30 2D63 2D53 2D4D 21
Character encodings
UTF-16
2D 30 2D 63 2D 53 2D 4D 00 21
UTF-8
E2 B4 B0 E2 B5 A3 E2 B5 93 E2 B5 8D 21
UTF-32
00 00 2D 30 00 00 2D 63 00 00 2D 53 00 00 2D 4D 00 00 00 21

UTF-32 (rare)
Fixed length 4 bytes

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

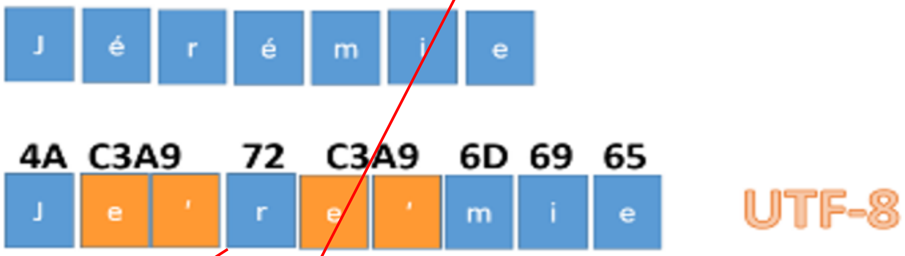


Practical case: Notepad++ (Encoding->UTF-8, Plugins->Hex-Editor)



● UTF-8

Jérémie

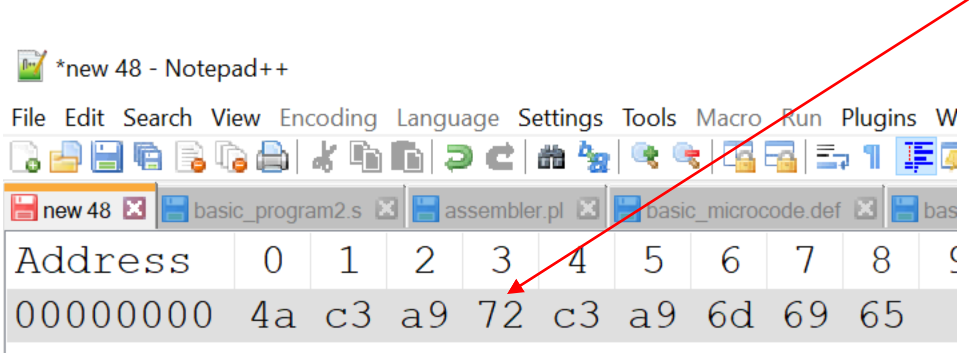


J é r é m i e

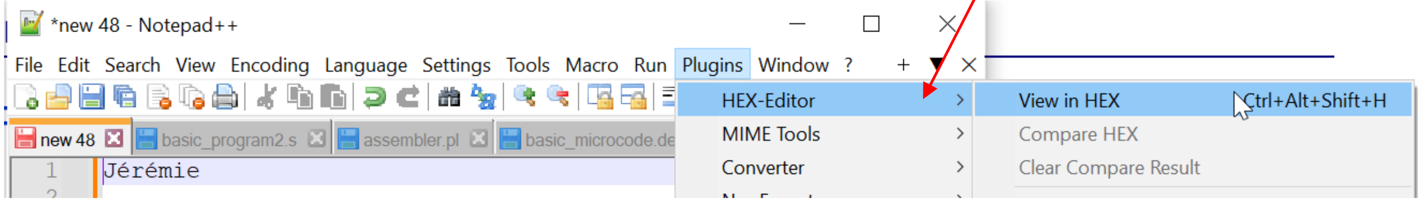
4A C3A9 72 C3A9 6D 69 65

J e ' r e ' m i e

UTF-8



```
Address   0  1  2  3  4  5  6  7  8  9
00000000  4a c3 a9 72 c3 a9 6d 69 65
```



Jérémie

Presentation of non-numeric operands - Unicode

- Unicode alphabet as the standard was adopted by IBM, Microsoft, Apple, HP, SunOracle and others.

- Use: The Java programming language, Javascript, XML, ...

- http://www.unicode.org

Sign	Unicode	UTF-16 Big Endian	UTF-16 Little Endian	UTF-8
Z	U+005A	005A	5A00	5A
Ž	U+017D	017D	7D01	C5BD

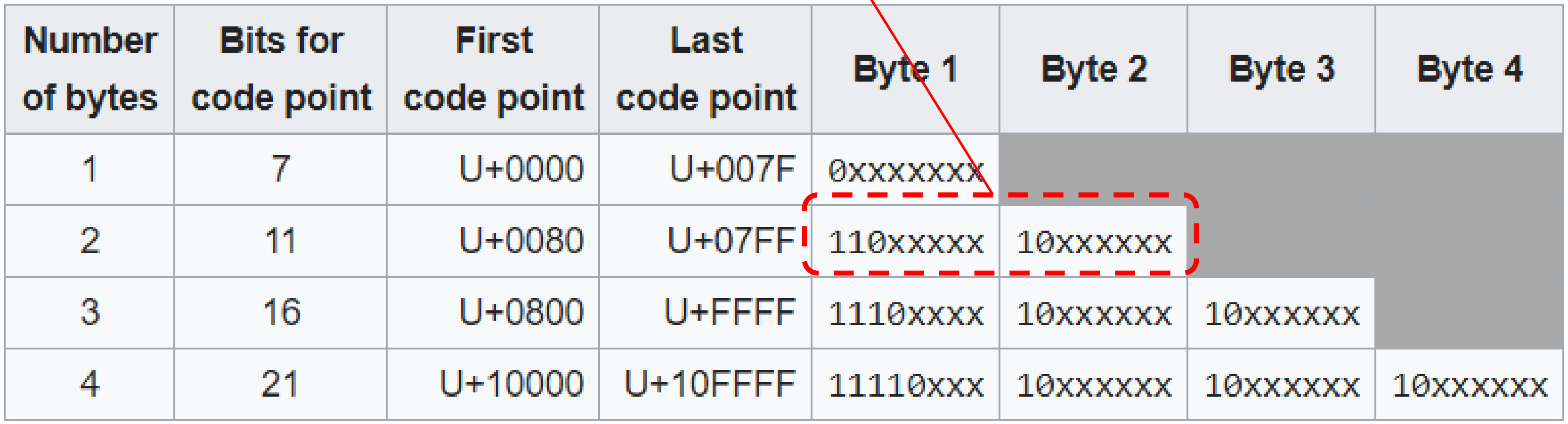
Example: char ‚Ž‘ in UTF-8:

- □ Ž (Unicode) = U + 017D = 0000 0001 0111 1101

0001 0111 1101

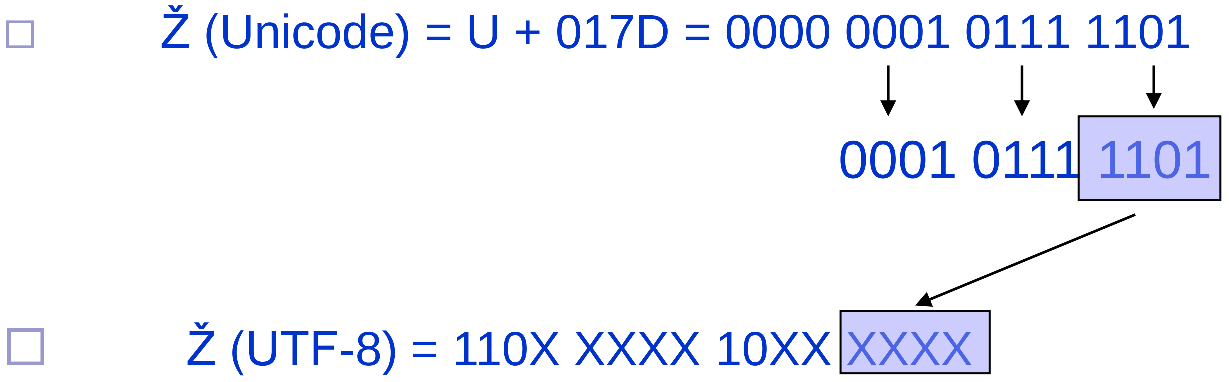
- □ The rule for transformation in the form of UTF-8 character codes of the U+00000080 to U+000007FF is:

110XXXXX 10XXXXXX



Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

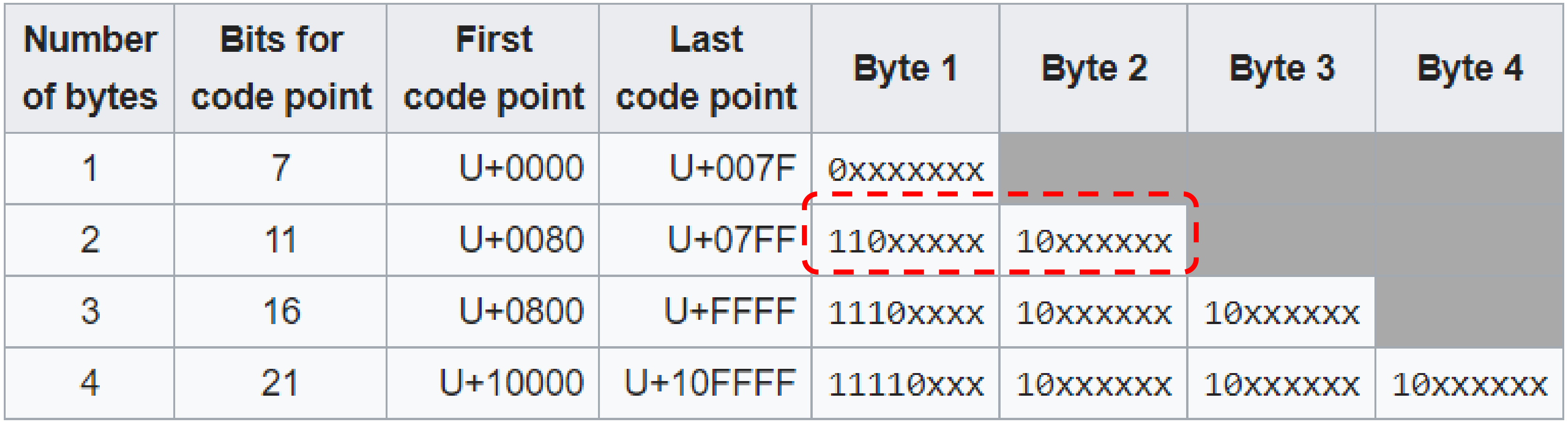
Example: char ‚Ž‘ in UTF-8:



□ Ž (Unicode) = U + 017D = 0000 0001 0111 1101

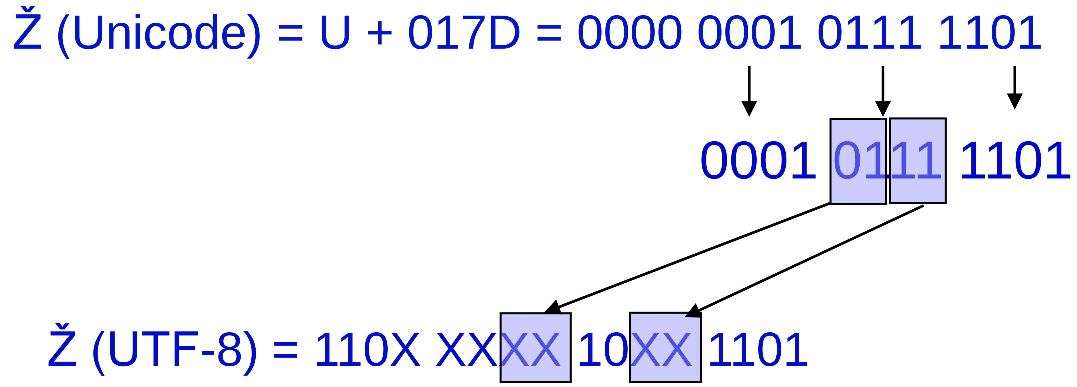
0001 0111 1101

□ Ž (UTF-8) = 110X XXXX 10XX XXXX



Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

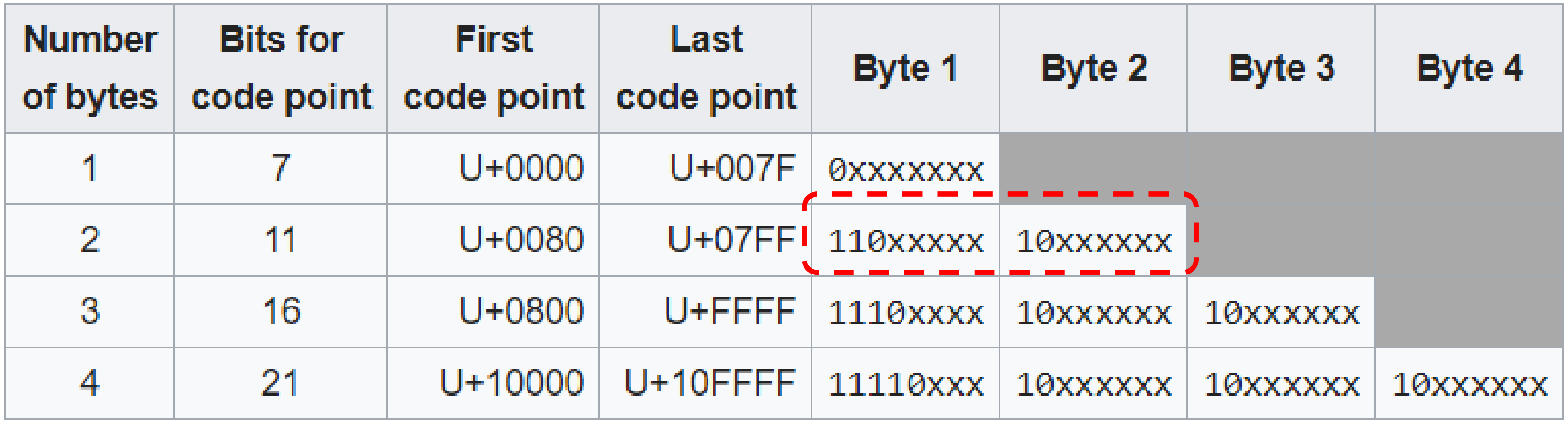
Example: char ‚Ž' in UTF-8:



□ Ž (Unicode) = U + 017D = 0000 0001 0111 1101

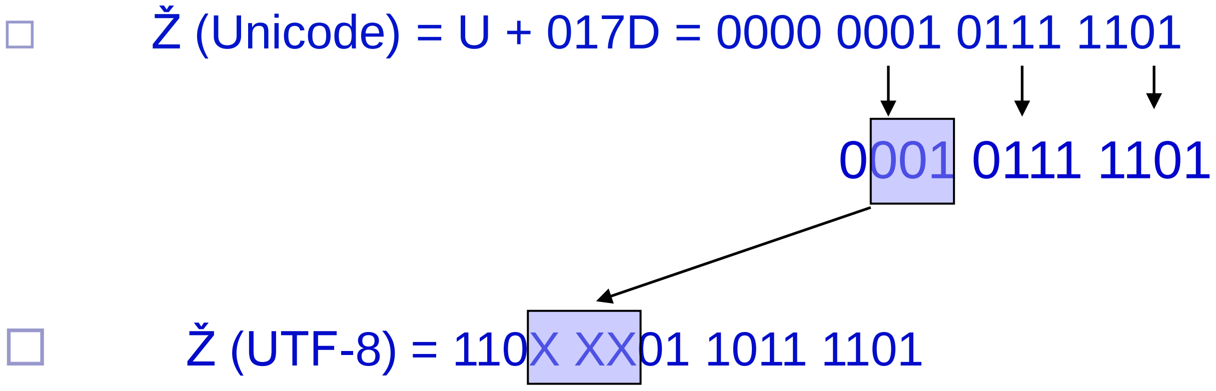
0001 0111 1101

□ Ž (UTF-8) = 110X XXXX 10XX 1101



Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

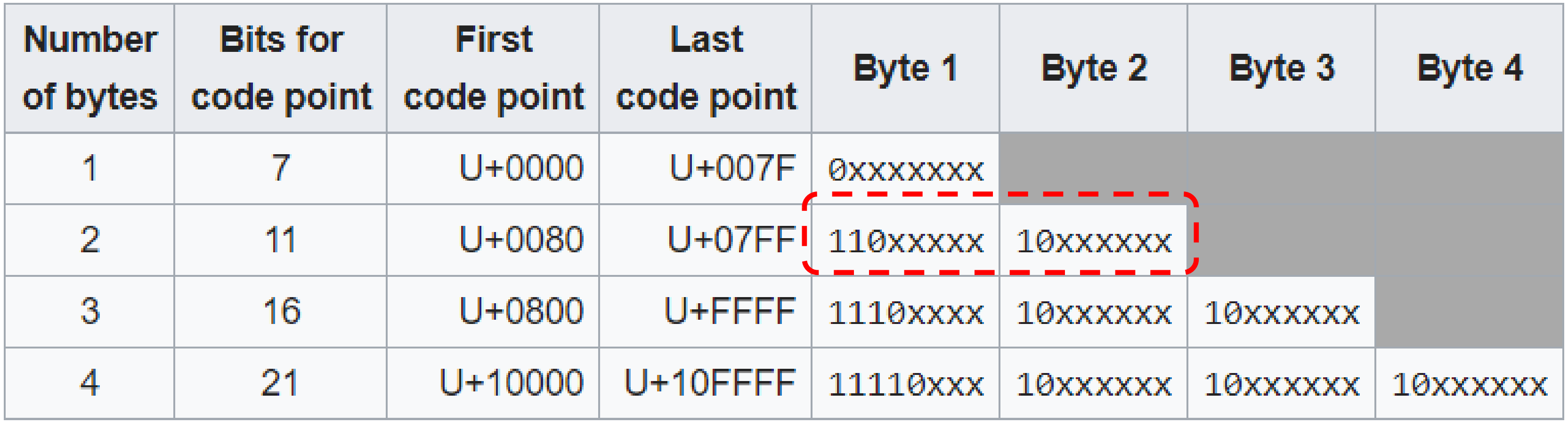
Example: char ‚Ž‘ in UTF-8:



- □ Ž (Unicode) = U + 017D = 0000 0001 0111 1101

0001 0111 1101

- □ Ž (UTF-8) = 110X XX01 1011 1101



Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

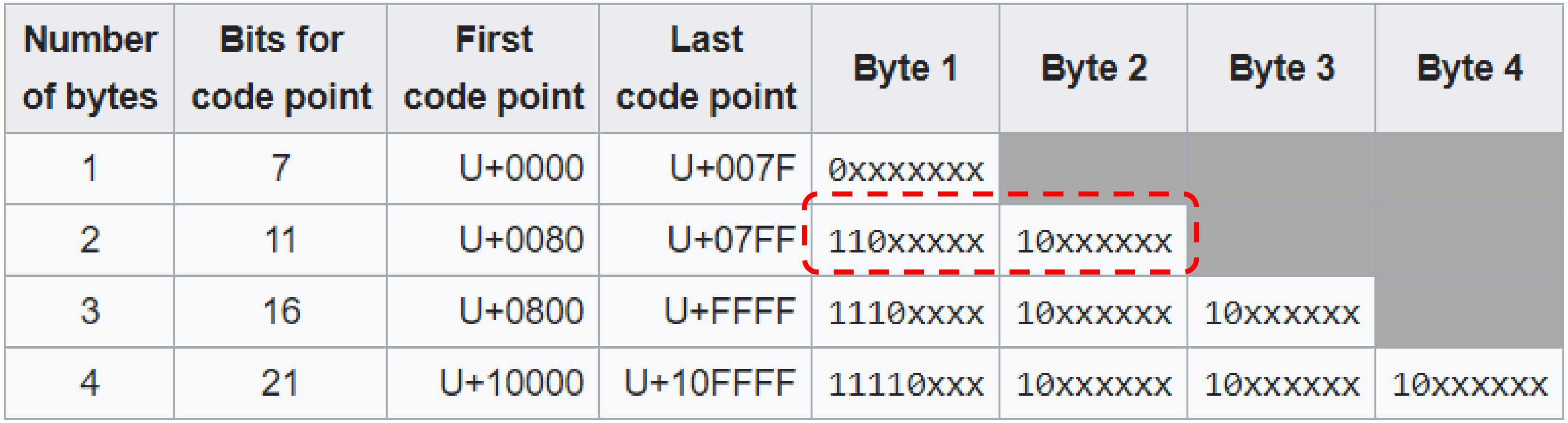
Example: char ‚Ž‘ in UTF-8:

□ Ž (Unicode) = U+017D = 0000 0001 0111 1101

↓ ↓ ↓

0001 0111 1101

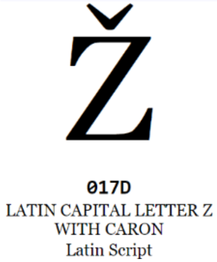
□ Ž (UTF-8) = 1100 0101 1011 1101

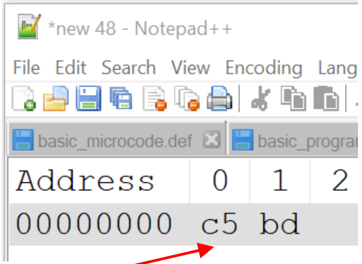


Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Example: char ‚Ž‘ in UTF-8:

- □ Ž (Unicode) = U+017D = 0000 0001 0111 1101

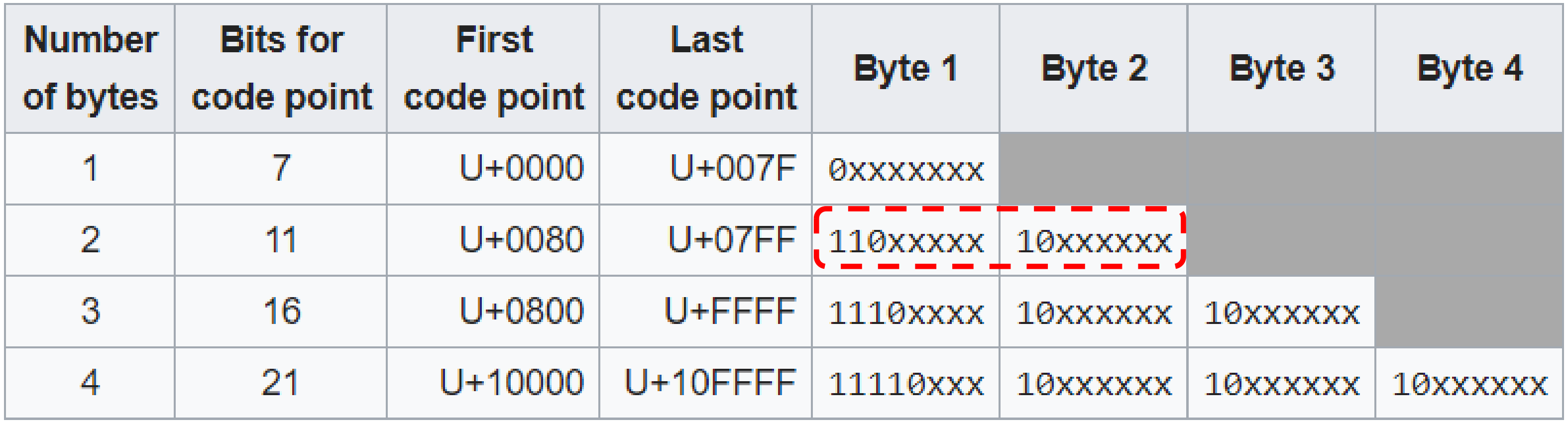




```
Address    0  1  2
00000000  c5 bd
```

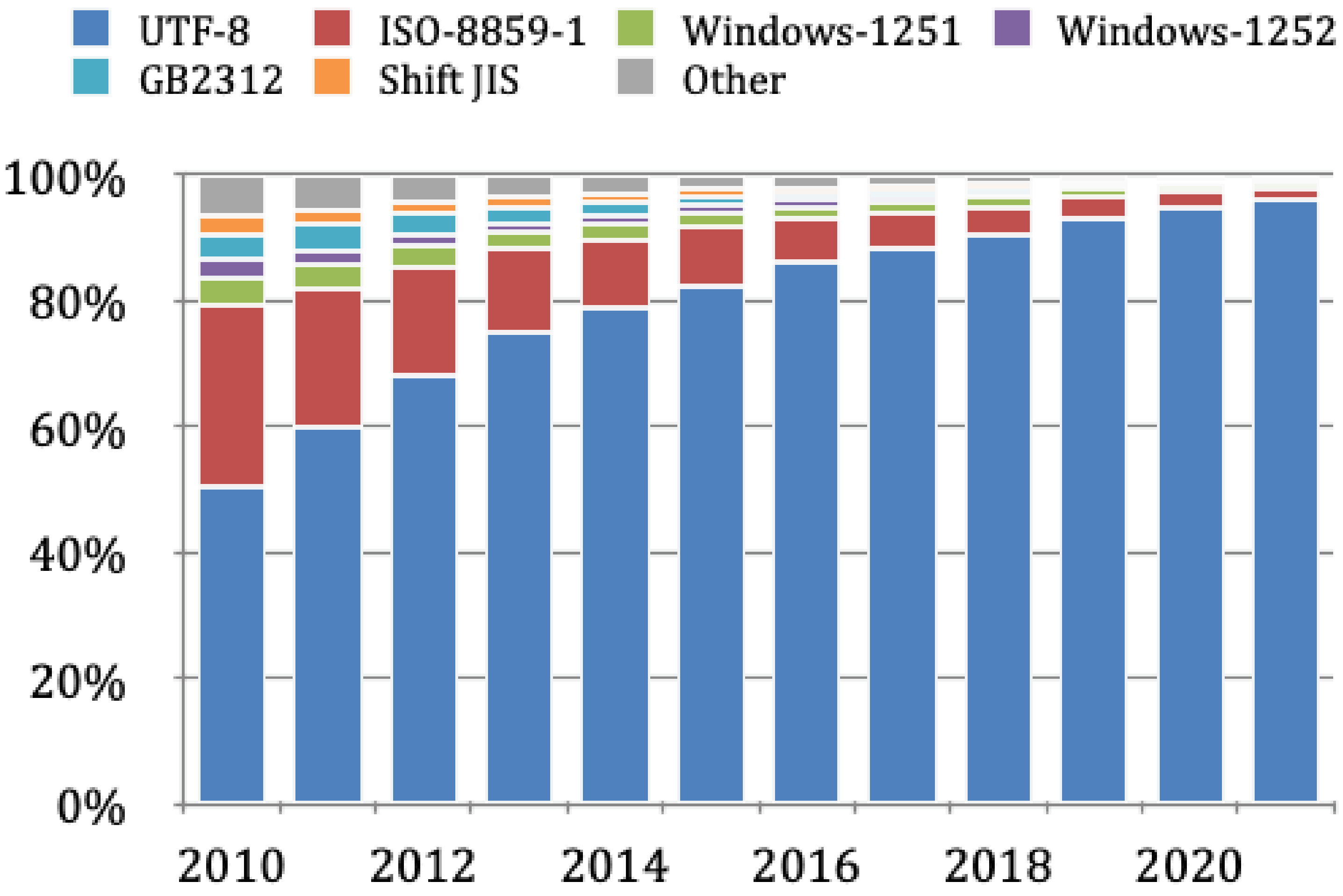
- □ Ž (UTF-8) = 1100 0101 1011 1101 = C5BD (hex)

PRESENTATION: C 5 B D



Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Declared character set for the 10 million most popular websites since 2010



https://en.wikipedia.org/wiki/UTF-8

5.2 Presentation of numerical operands in fixed-point arithmetic

- The comma is at a predetermined fixed position - a presentation with a fixed point.
- If the comma is on the right of the bit with the lowest weight, then the number is integer, otherwise it is not integer.
- Integers are partially also a synonym for a fixed-point presentation

Unsigned number:

- The minimum and maximum conceivable unsigned (positive) number that can be represented by *n bits* is:

$$0 \le x \le 2^n - 1$$

- case of 8-bit length (n = 8) $n = 8 \qquad 0_D \le x \le 255_D$

- case of 32-bit length (n = 32) $n = 32 \quad 0_D \le x \le 4.294.967.295$

- **Carry** - if the result of adding or subtracting positive (unsigned) numbers is outside of the range, there is a carry (transfer) from the highest bit (place)

Signed number:

- For integers with the sign, there are four modes of presentation used (or were used) :
  - □ Sign and magnitude
  - □ Offset
  - □ Ones‘ complement (the complement is for only negative numbers)
  - □ Two's complement (the complement is for only negative numbers)

- *n*-bit sequence $b_{n-1}\dots b_2 b_1 b_0$ in any mode represents a signed integer

b7	b6	b5	b4	b3	b2	b1	b0	
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	weights of bits

8-bit sequence

Presentation of the numbers in fixed point - Signed numbers

1. Sign and magnitude :

$$V(b) = (-1)^{b_{n-1}} \sum_{i=0}^{n-2} b_i 2^i$$

- The highest bit is the sign (1 - negative, 0 positive number)

$10001110_{(2)} = (-1)^{1}(1x2^{3}1x2 +^{2}1x2 +^{1}) = (-1)\ (14) = -14_{(10)}$

- The process of conversion from the decimal number into an n-bit binary number
  1. Convert a number in binary to n-1 bits
  2. The highest bit is set according to the sign

1. Sign and magnitude :

- Examples:

  $-25_{(10)}$ = 10011001

  $33_{(10)}$ = 00100001

- Maximum number in 8 bits
  - □ $01111111_{(2)} = +127_{(10)}$
- Minimum number in 8 bits
  - □ $11111111_{(2)} = -127_{(10)}$
- Zero
  - □ $00000000_{(2)} = +0_{(10)}$
  - □ $10000000_{(2)} = -\ 0_{(10)}$

2. Presentation with offset :

VALUE = PRESENTATION - OFFSET

8b: -128 .. 127 = 0 .. 255 - 128

$$V(b) = \sum_{i=0}^{n-1} b_i 2^i - odmik$$

- After the conversion to a decimal number subtract the offset
  - □ in this case offset is 2^{n-1}

$$10001110_{(2)} =$$

$$(1x2^7 1x2 +^3 1x2 +^2 1x2 +^1) - (2^7) = 128 + 8 + 4 + 2-128= 14_{(10)}$$

2. Presentation with offset :

- Process conversion from the decimal number into an n-bit binary number
  1. Add the offset to the value
  2. Convert like unsigned number

PRESENTATION = **VALUE** + **OFFSET**

- The conversion process from n-bit binary number in decimal number
  1. Convert like unsigned number
  2. Subtract offset to get value

VALUE = **PRESENTATION** - **OFFSET**

2. Presentation with offset :

VALUE = PRESENTATION - OFFSET

8b: -128 .. 127 = 0 .. 255 - 128

- Example (offset = 2^{n-1}):

  $-26_{(10)} = 01100110$

  $32_{(10)} = 10100000$

- The maximum number of bits per 8ih
  - □ $11111111_{(2)} = +127_{(10)}$
- The minimum number of bits per 8ih
  - □ $00000000_{(2)} = -128_{(10)}$
- Zero
  - □ $10000000_{(2)} = 0_{(10)}$

3. Ones‘ complement:

$$V(b) = \sum_{i=0}^{n-2} b_i 2^i - b_{n-1}(2^{n-1} - 1)$$

- Converting into decimal value: subtract 2^{n-1}-1 from the number if the most significant bit is one

$10001110_{(2)}$ = $(1x2^{3}1x2 +^{2}1x2 +^{1})$ 1x $(2^{7}-1)$ = 8 + 4 + 2-127 = $-113_{(10)}$

- The process of conversion from the decimal number into an n-bit binary number
  1. Convert as unsigned number
  2. If the number of negative, negate ("invert") all bits

3. Ones‘ complement:

8b: -127 .. 127

- Examples

  $-25_{(10)} = 11100110$

  $33_{(10)} = 00100001$

- The maximum number of bits per 8ih
  - □ $01111111_{(2)} = +127_{(10)}$
- The minimum number of bits per 8ih
  - □ $10000000_{(2)} = -127_{(10)}$
- zero
  - □ $00000000_{(2)} = +0_{(10)}$
  - □ $11111111_{(2)} = -0_{(10)}$

4. Two's complement:

$$V(b) = \sum_{i=0}^{n-2} b_i 2^i - b_{n-1}(2^{n-1})$$

- When converting to a decimal number, subtract 2^{n-1} if the most significant bit is one

$10001110_{(2)}$ = (1x2^{3}1x2 +^{2}1x2 +^{1}) 1x (2^7) = 8 + 4 + 2-128 = -114$_{(10)}$

4. Two's complement:

- The process of conversion from the decimal value into an n-bit binary number
  1. If the number is positive, convert as unsigned number
  2. If the number is negative, invert the bits in absolute value and add 1

- The process of conversion from n-bit binary presentation to the decimal value
  1. If the presentation is negative, invert the bits and add 1, then add negative sign
  2. converts as unsigned number (including a sign)

4. Two's complement:

- Examples

  $-25_{(10)} = 11100111$

  $33_{(10)} = 00100001$

- The maximum number in 8 bits
  - $01111111_{(2)} = +127_{(10)}$
- The minimum number in 8 bits
  - $10000000_{(2)} = -128_{(10)}$
- Zero
  - $00000000_{(2)} = 0_{(10)}$

Presentation of the numbers in fixed-point arithmetic - Example-1

- Example 1: Which decimal value represents 8-bit presentation 10010100 in each of the four fixed-point signed presentations?

b7	b6	b5	b4	b3	b2	b1	b0	
1	0	0	1	0	1	0	0	
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	weights of bits

Presentation of the sign and magnitude: b7 = 1 ⇒ number is negative
Value = $0x2^6$ $0x2 +^5$ $1x2 +^4$ $0x2 +^3$ $1x2 +^2$ $0x2 +^1$ $0x2 +^0$ = 16 + 4 = 20 (dec)
In presentation of the sign and magnitude, this presentation implies value -20(dec)

Presentation by offiset: offset can be 2^{n-1}= 128, or 2^{n-1}-1 = 127; let‘s select 128 (dec)
The decimal value of the 8-bit presentation 10010100 includes an offset, and is 128 + 16 + 4 = 148,
To get value, we subtract offset: 148-128 = 20
In a presentation by offset of 128, the value is +20(dec)

Presentation in ones‘ complement: b7 = 1 ⇒ number is negative, therefore, the presentation 10010100 is a complement of the corresponding positive number.
10010100 ⇒ ones‘ complement = 01101011 = 64 + 32 + 8 + 2 + 1 = 107 (DEC)
Presentation 10010100 in ones‘ complement represents the value of -107(dec)

Presentation of the numbers in fixed-point arithmetic - Example-1

Presentation of two‘s the binary complement: b7 = 1 ⇒ number is negative, therefore, the presentation 10010100 is a complement of the corresponding positive number.
10010100 ⇒ two's complement = 01101100 = 64 + 32 + 8 + 4 = 108 (DEC)
Presentation 10010100 in two‘s complement represents the number of -108(dec)

b7	b6	b5	b4	b3	b2	b1	b0
1	0	0	1	0	1	0	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

= -20 (dec) in a presentation „sign and magnitude“

b7	b6	b5	b4	b3	b2	b1	b0
1	0	0	1	0	1	0	0

= + 20 (dec), in a presentation by offset

b7	b6	b5	b4	b3	b2	b1	b0
1	0	0	1	0	1	0	0

= -107 (dec), in a presentation ones‘ complement

b7	b6	b5	b4	b3	b2	b1	b0
1	0	0	1	0	1	0	0

= -108 (dec), in a presentation two‘s complement

Presentation of the numbers in fixed-point arithmetic - Example-2

- Which decimal number represents 8-bit combination 00010100 in each of the four fixed-point signed presentations?

b7	b6	b5	b4	b3	b2	b1	b0	
0	0	0	1	0	1	0	0	
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	weights of bits

Presentation of the sign and magnitude: b7 = 0 ⇒ number is positive
Value = $0x2^6$ $0x2$ $+^5$ $1x2$ $+^4$ $0x2$ $+^3$ $1x2$ $+^2$ $0x2$ $+^1$ $0x2$ $+^0$ = 16 + 4 = 20 (dec)
The presentation of the sign and magnitude, this presentation implies value +20 (dec)

Presentation by offset: offset may be 2^{n-1}= 128, or 2^{n-1}-1 = 127; let‘s select 128 (dec)
The decimal value of the 8-bit combination 00010100 includes offset and is 16 + 4 = 20
Subtract offset 20-128 = -108
In a presentation by offset, this combination represents the number of -108(dec)

Presentation in ones‘ complement: b7 = 0 ⇒ number is positive, therefore, combination 00010100 is not the complement and the value can be calculated directly.
00010100 = 16 + 4 = + 20(dec)
Combination 00010100 in ones‘complement represents the value of +20 (dec)

Presentation of the numbers in fixed-point arithmetic - Example-2

Presentation of the binary complement: b7 = 0 ⇒ number is positive, therefore, combination 00010100 is not the complement and the value can be calculated directly.
00010100 = 16 + 4 = + 20 ° C (dec)
Combination 00010100 in binary complement represents the number of +20 (dec)

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	1	0	1	0	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

= +20 (dec) in the presentation of the sign and magnitude

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	1	0	1	0	0

= -108 ° C (dec), in a presentation at a distance

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	1	0	1	0	0

= +20 (dec) in the presentation of the complement eniškim

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	1	0	1	0	0

= +20(dec), in a presentation using a binary complement

Singed number – range, overflow:

- The maximum and minimum number you can present with *n bits* in two‘s complement is:

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

- In case of 8-bit

$$n = 8 \qquad -2^{7} \leq x \leq 2^{7} - 1$$
$$-128_D \leq x \leq +127_D$$

- In case of 32-bit

$$n = 32 \quad -2.147.483.648_D \leq x \leq +2.147.483.647_D$$

- **Oveflow** - if the result is outside the range that is presentable in two‘s complement

Presentation of the numbers in fixed-point arithmetic - carry and overflow

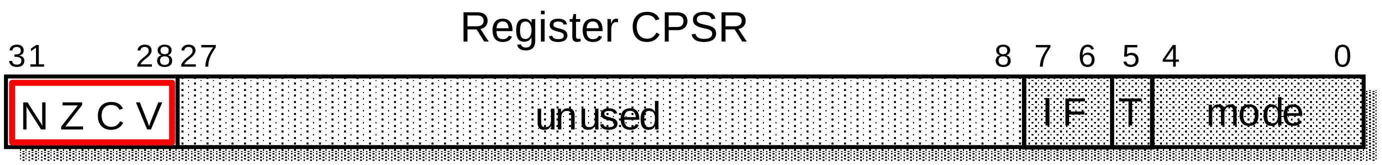
- Carry or overflow can be the cause of the error.

- The CPU must include the mechanism by which a programmer can determine whether the outcome of the operation has a carry or overflow.

- Bits (flags) C (Carry) and V (oVerflow) in the condition (or status) register in the CPU are set to values, that indicate, whether an operation caused carry or overflow.

The example of the condition (status) register:

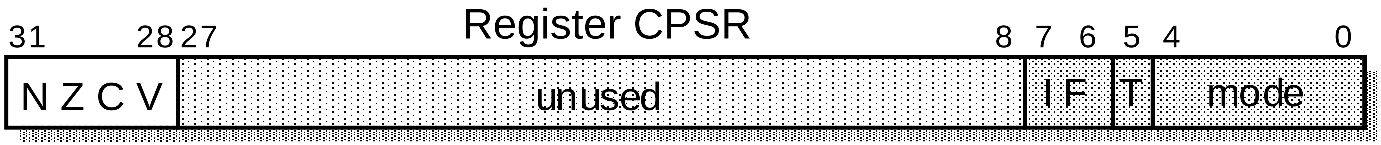
- register CPSR (Current Program Status Register) in ARM9 CPU



- Being N, Z, C and V - a flag (flag bitsthe status flags)

- Be the flags They can put in the state of 1 or 0, after they executed an arithmetic or logical operation according to the result of the operation.

- register CPSR (Current Program Status Register) in ARM9 CPU



Register CPSR

31 – 28	27 – 8	7 – 6	5	4 – 0
N Z C V	unused	I F	T	mode

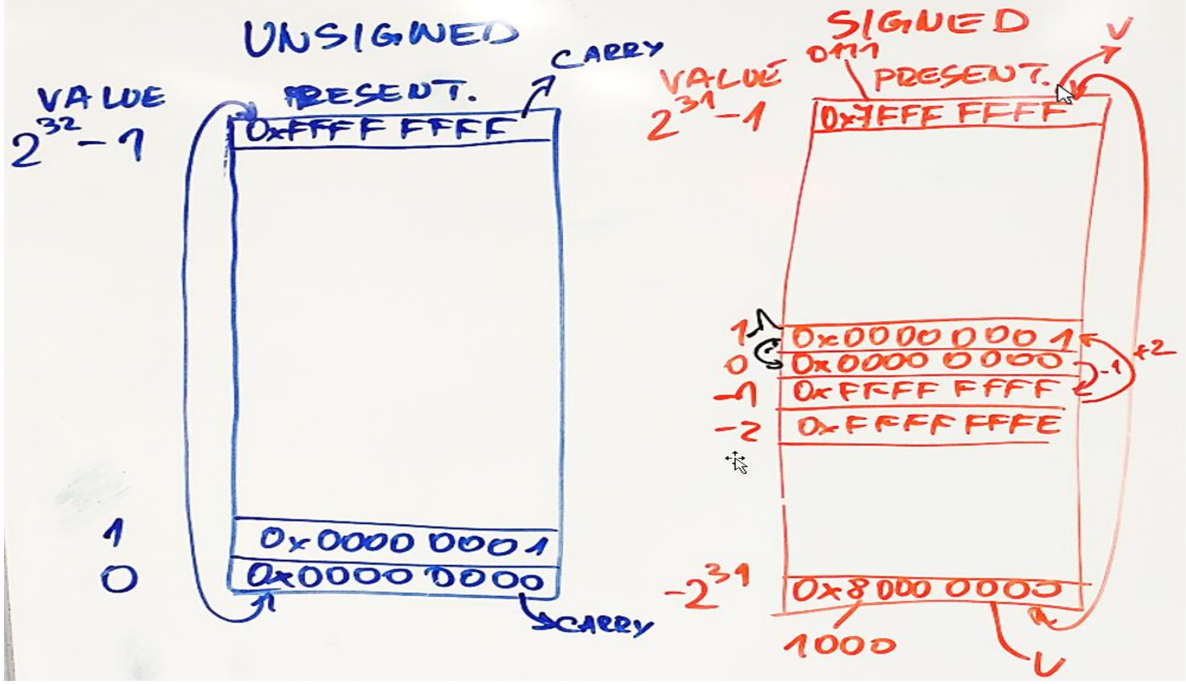
- □ **oVerflow** (Bit 28 in the CPSR) V = 1: the result has an overflow;
  V = 0: no overflow

- □ **Carry** (Bit 29 in the CPSR) addition:
  C = 1: the result has a carry;
  C = 0, no carry
  subtraction:
  C = 0: the result has a carry;
  C = 1, no carry

- □ **Zero** (Bit 30 in the CPSR) Z = 1: result is 0;
  Z = 0: result is not 0

- □ **Negative** (Bit 31 in the CPSR) N = 0, bit 31 of the result is 0;
  N = 1: bit 31 of the result is 1

Unsigned and signed numbers – comparison on 32 bits



$$0 \le x \le 2^{n} - 1$$

$$-2^{n-1} \le x \le 2^{n-1} - 1$$

$$0_D \le x \le 4.294.967.295$$

$$-2.147.483.648_D \le x \le +2.147.483.647_D$$

5.3 Arithmetic with numbers in fixed-point

- Arithmetic - four basic operations: addition, subtraction, multiplication and division.

- Arithmetic operations are executed in the arithmetic-logic unit (ALU), which is part of the CPU.

- The type and number of operations that are executed by ALU differ between different computers – at simplest computers, only the addition and logical operations are done by ALU, other operations are implemented by programs.

Arithmetic with numbers in fixed-point

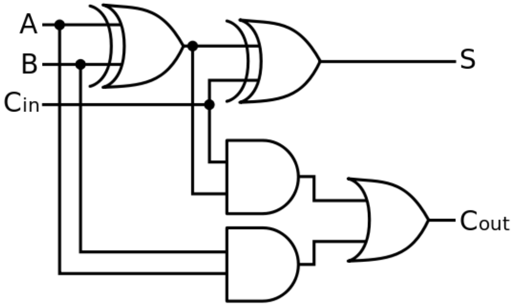
- The key circuit for the realization of the arithmetic operations, is the n-bit parallel universal binary adder that calculates sum of two unsigned integers.

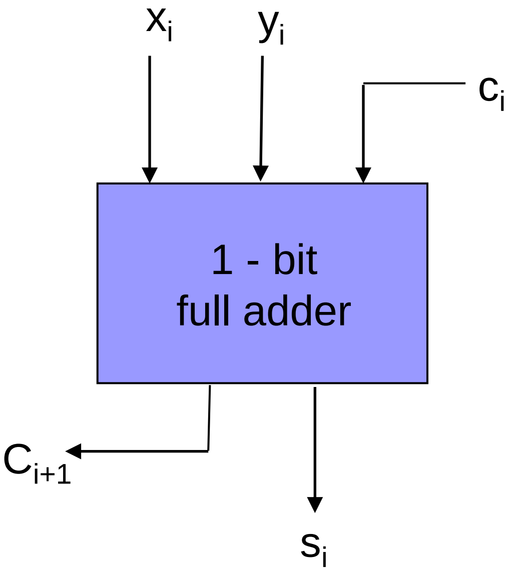
- With this device, we can implement all basic operations, including subtraction (to represent negative numbers we commonly use two's complement), and also multiplication, and division (if specific units are not present)

- The basic element, with which we build *n*-bit adder, is *1*-bit full adder.

Arithmetic with numbers in fixed-point

- *1*-bit full adder has three inputs
  - □ two summands x_i and y_i
  - □ input carry c_i
- and two outputs
  - □ sum s_i
  - □ output transfer c_{i+1}

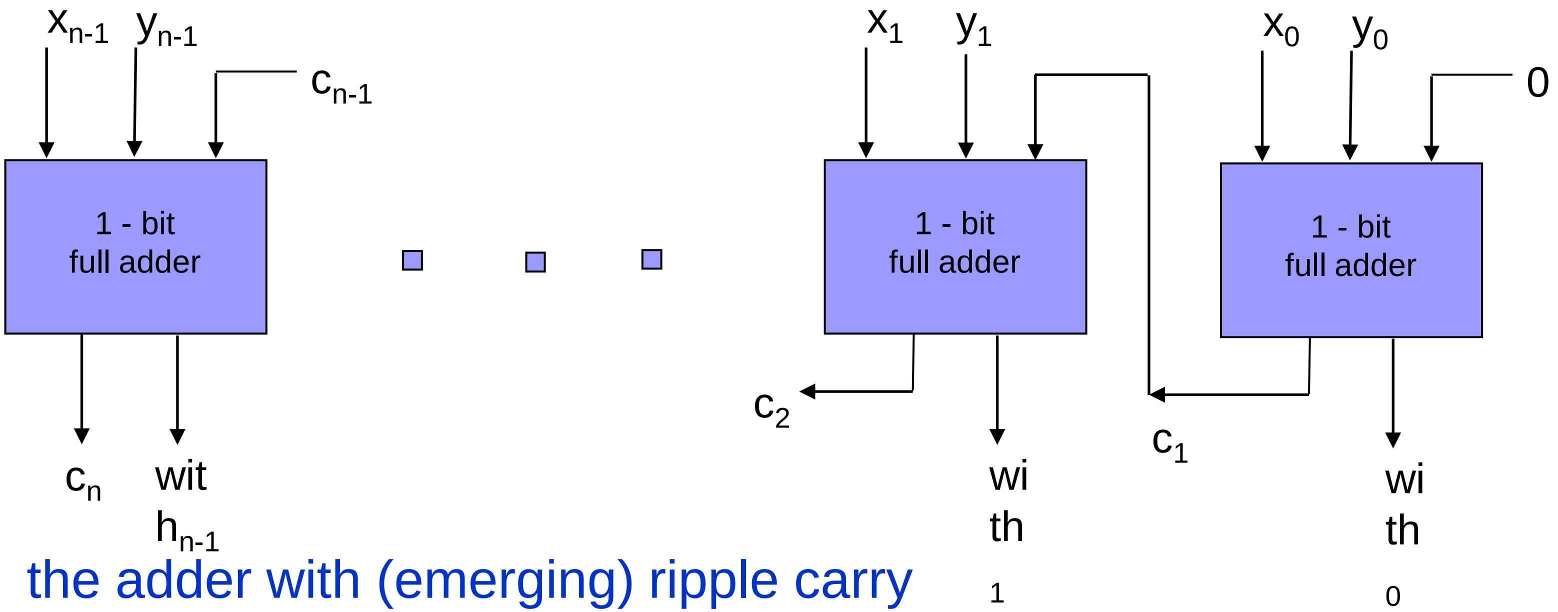




Truth Table

Inputs			Outputs	
x_i	y_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

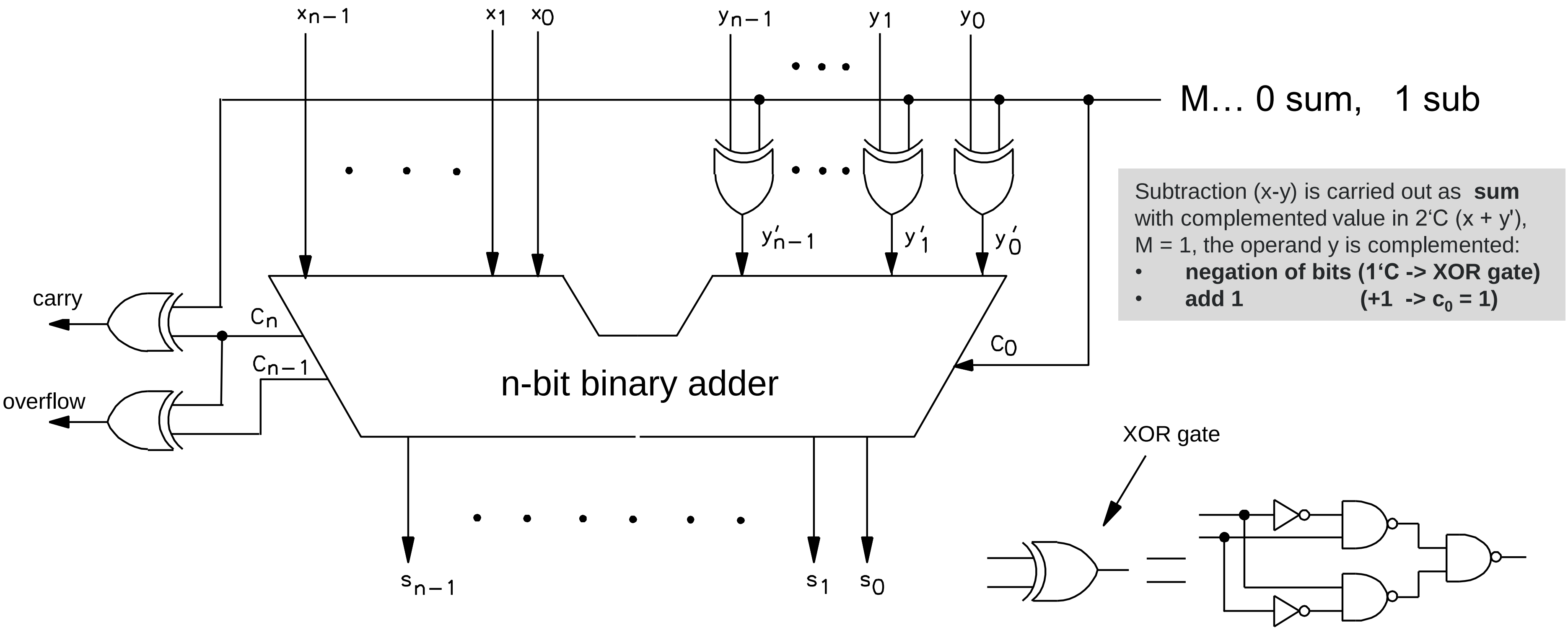
- *n*-bit adder is obtained by connecting *n* 1-bit adders – we get an adder with the emerging carry



- the adder with (emerging) ripple carry
  - □ Disadvantage: slow speed (increasing with the number of bits)
  - □ Faster, more complex solution is

-> "the carry-lookahead adder"

Universal adder
(addition, subtraction, unsigned and signed numbers)



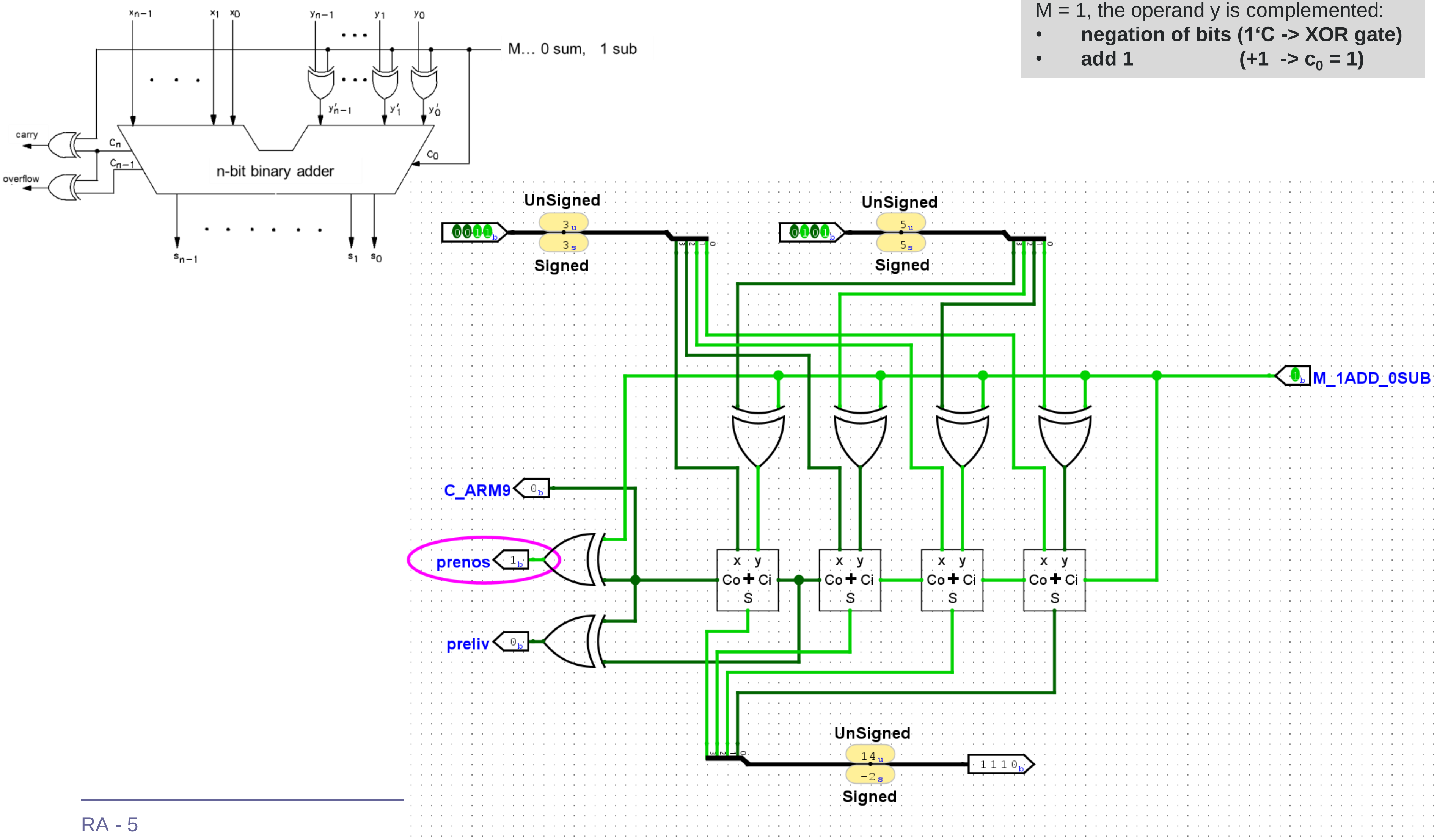
Subtraction (x-y) is carried out as **sum** with complemented value in 2‘C (x + y'), M = 1, the operand y is complemented:

- **negation of bits (1‘C -> XOR gate)**
- **add 1 (+1 -> c_0 = 1)**

Universal adder – Logisim model

Subtraction (x-y) is carried out as **sum** with complemented value in 2‘C (x + y'), M = 1, the operand y is complemented:

- **negation of bits (1‘C -> XOR gate)**
- **add 1 (+1 -> c_0 = 1)**



5.4 presentation of numerical operands in floating point

- The range of numbers that can be represented in a presentation with fixed point, is usually for technical problems too small.

- These numbers are usually written in scientific notation, which allows the presentation with the relatively small number of digits

$$3.200.000{,}00 = 3{,}20000000 \cdot 10^{6} = 0{,}03200000 \cdot 10^{8} = 32000000{,}0 \cdot 10^{-1} =$$

- Presentation of numbers in floating point format is only a computer-modified form of a scientific notation.

- The general form

$$m \cdot r^{e} \rightarrow npr.: 0{,}03200000 \cdot 10^{8}$$

- □ *m* - mantissa (significand, fraction) = 0.03200000
- □ *r* - base (radix) = 10
- □ *e* - exponent = 8

Standard for the presentation in floating point

- The numbers in floating point can be presented in many ways:
  - various number of bits for the representation of mantissa and an exponent,
  - various ways of presenting exponent and mantissa,
  - various methods of rounding.

- Computer manufacturers have for many years used a variety of formats, that were not compatible. Therefore, the same program on different computers gave different results.

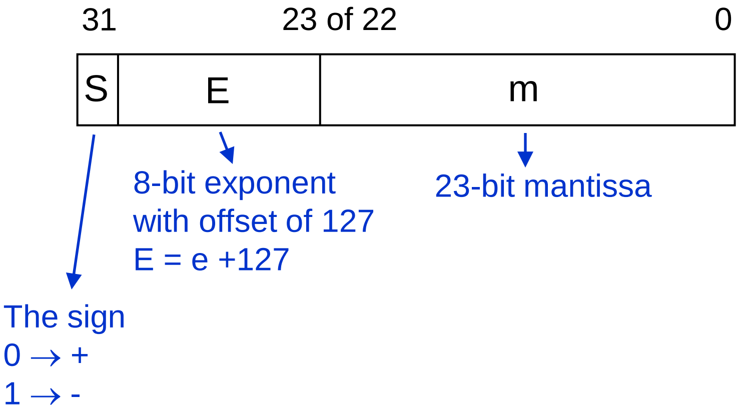
- In 1981, in the context of the IEEE organization, a standard for floating point arithmetic was proposed, and in 1985 adopted in the final form marked as „IEEE 754“ and is still used by majority of computers.

- In addition to the format for the presentation of numbers specified in the standard, also the implementations of arithmetic operations (rounding) and procedures in case of errors (overflow, divide by 0, etc.) are specified.

- Basic features presentations numbers in IEEE 754
  - □ Standard uses a base *r* = 2
  - □ Mantissa is presented as „the sign and magnitude“.
  - □ The implicit representation of the normal bit: the comma is right of the normal bit (= left from the first bit of mantissa).
  - □ Exponent is presented in a presentation with offset.
  - □ Defined are two formats:
    - *32*-bit format or single precision and
    - *64*-bit format or double precision.

Presentation of numbers in floating point - IEEE 754 32-bit and 64-bit format

32-bit format (single precision)



31 23 of 22 0

S E m

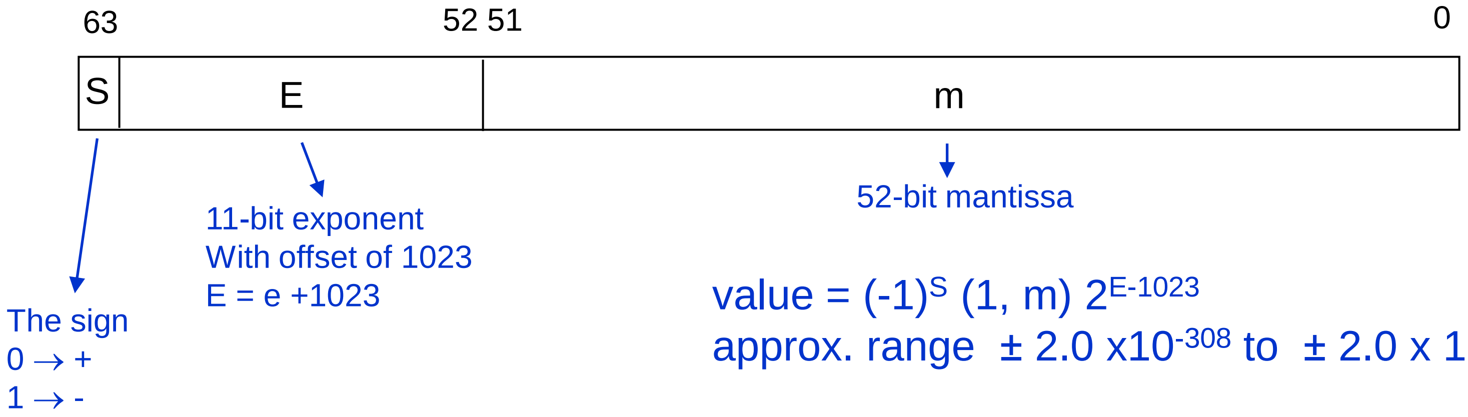
8-bit exponent
with offset of 127
E = e +127

23-bit mantissa

The sign
0 → +
1 → -

value $= (-1)^S\ (1, m)\ 2^{E-127}$
approx. range $\pm 2.0 \times 10^{-38}$ to $\pm 2.0 \times 10^{38}$

64-bit format (double precision)



63 52 51 0

S E m

11-bit exponent
With offset of 1023
E = e +1023

52-bit mantissa

The sign
0 → +
1 → -

value $= (-1)^S\ (1, m)\ 2^{E-1023}$
approx. range $\pm 2.0 \times 10^{-308}$ to $\pm 2.0 \times 10^{308}$

Presentation of numbers – standard IEEE 754

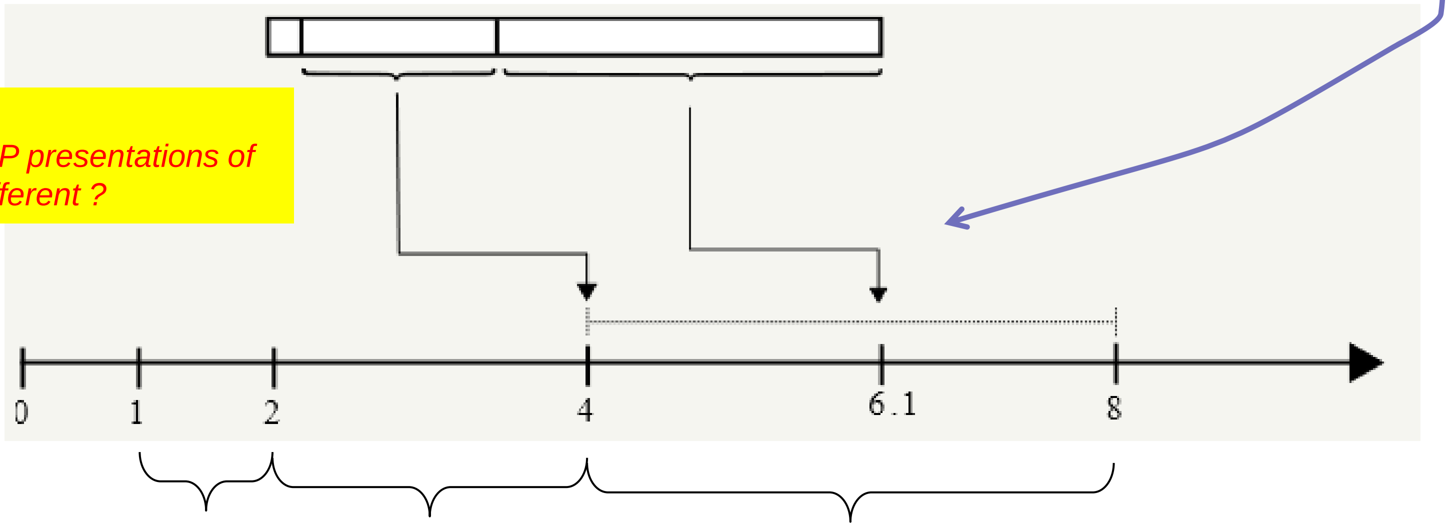
Presented number	Exponent E	Mantissa m
normalized number (1,m)	000 ... 001 to 111 ... 110	any
denormalized number (0,m)	000 ... 000	different from 0
Zeros ±0	000 ... 000	000 ... 000
Infinity ± ∝	111 ... 111	000 ... 000
Not A Number NaN	111 ... 111	different from 0

Important facts related to presentation of numbers in floating point 1:

- □ Interpretation aspect :
  - □ 1 <=   Mantissa <= 2,   if exponent = 0
  - □ 2 <= 2*Mantissa <= 4,   if exponent = 1
  - □ 4 <= 4*Mantissa <= 8,   if exponent = 2
- ■ Exponent defines window, mantissa is offset

S	EXPONENT	MANTISSA
S	WINDOW	OFFSET

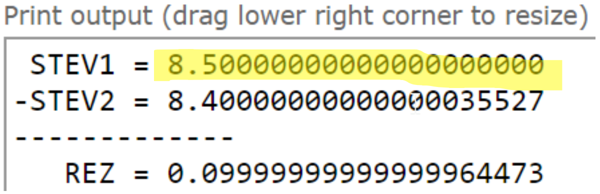
Challenge:
- How are FP presentations of x in 2*x different ?



Important facts related to presentation of numbers in floating point 2:

- □ Still only a finite number of bits, and consequently, the number of values !
- □ Limitations (base=2): ->
  - Example: (8.5 - 8.4)
  - Example: (0.1, 0.125)

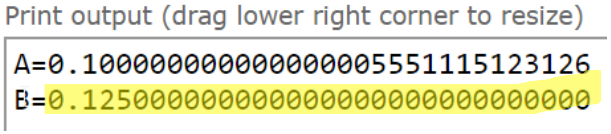
```
STEV1=8.5
STEV2=8.4
REZ = STEV1 – STEV
```



Print output (drag lower right corner to resize)

```
 STEV1 = 8.500000000000000000000
-STEV2 = 8.400000000000000035527
---------------
   REZ = 0.099999999999999964473
```

```
A=0.1
B=0.125
print "A=%.30f" % (A)
print "B=%.30f" % (B)
```



Print output (drag lower right corner to resize)

```
A=0.100000000000000005551115123126
B=0.125000000000000000000000000000
```

$2^0 = 1$

$2^{-1} = \frac{1}{2^1} = \frac{1}{2} = 0.5$

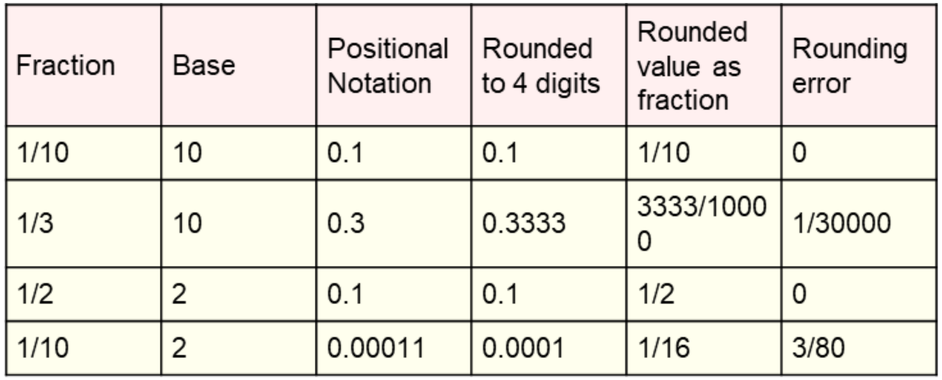
$2^{-2} = \frac{1}{2^2} = \frac{1}{4} = 0.25$

$2^{-3} = \frac{1}{2^3} = \frac{1}{8} = 0.125$

$2^{-4} = \frac{1}{2^4} = \frac{1}{16} = 0.0625$

$2^{-5} = \frac{1}{2^5} = \frac{1}{32} = 0.03125$

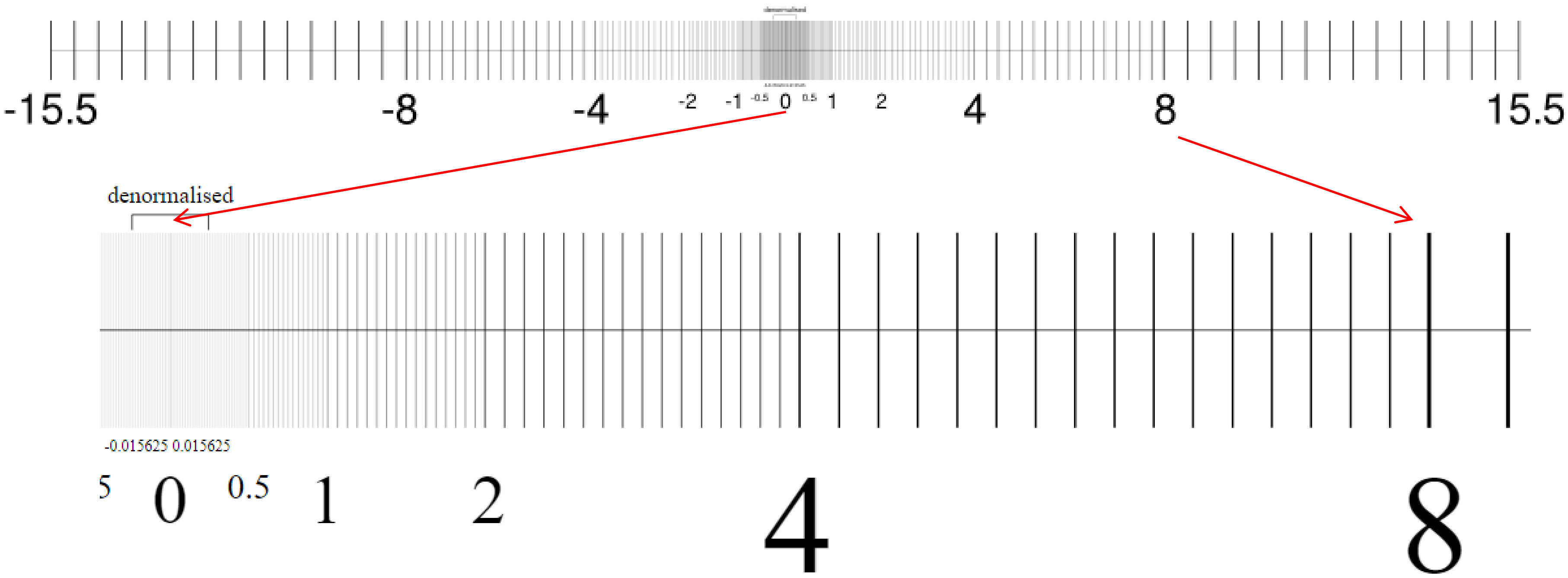
Explanation?



Fraction	Base	Positional Notation	Rounded to 4 digits	Rounded value as fraction	Rounding error
1/10	10	0.1	0.1	1/10	0
1/3	10	0.3	0.3333	3333/10000	1/30000
1/2	2	0.1	0.1	1/2	0
1/10	2	0.00011	0.0001	1/16	3/80

Important facts related to presentation of numbers in floating point 3:

- □ Non-uniform density in „windows“ :
  - In each „window“ equal number of points, but different range !
  - Example: 8-bit FP format (4-bit mantissa, 3-bit exponent)



Important facts related to presentation of numbers in floating point 4:

- □ Only fractions with powers of 2 in denominator can be exactly presented!

$$2^{0} = 1$$

$$2^{-1} = \frac{1}{2^{1}} = \frac{1}{2} = 0.5$$

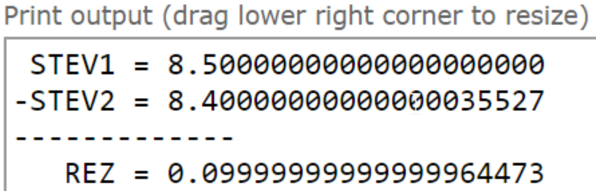
$$2^{-2} = \frac{1}{2^{2}} = \frac{1}{4} = 0.25$$

$$2^{-3} = \frac{1}{2^{3}} = \frac{1}{8} = 0.125$$

$$2^{-4} = \frac{1}{2^{4}} = \frac{1}{16} = 0.0625$$

$$2^{-5} = \frac{1}{2^{5}} = \frac{1}{32} = 0.03125$$

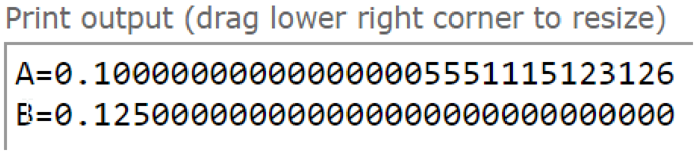
STEV1=8.5
STEV2=8.4
REZ = STEV1 – STEV



Print output (drag lower right corner to resize)

```
 STEV1 = 8.50000000000000000000
-STEV2 = 8.40000000000000035527
--------------
   REZ = 0.09999999999999964473
```

A=0.1
B=0.125
print "A=%.30f" % (A)
print "B=%.30f" % (B)



Print output (drag lower right corner to resize)

```
A=0.100000000000000005551115123126
B=0.125000000000000000000000000000
```

- □ Non-negative FP presentations can be compared as integers ?

Why?

S	EXPONENT	MANTISSA

Presentation of numbers in floating point - IEEE 754 – Example 1

- Example-1: Write negative decimal number -4.625 in the presentation for floating point numbers in single precision.

First, we convert number in binary format (integer and fractional part separately)

- 4.625 = - (4 + 0.625)

4 (dec) = 100 (bin) ⇐

4: 2 = 2 remainder 0 b0 (LSB) = 0
2: 2 = 1 remainder 0 b1 = 0
1: 2 = 0 remainder 1 b2 = 1

0.625 (dec) = 0.101 (bin) ⇐

0.625 x 2 = 1.25 ⇒ 0.1
0.25 x 2 = 0.5 ⇒ 0.10
0.5 x 2 = 1.0 ⇒ 0.101
0.0 x 2 = 0 ⇒ 0.1010

4.625 = 100.101 = 100.1010000 .... we can add zeros on the right end

Presentation of numbers in floating point - IEEE 754 – Example 1

Number is normalized as follows: the decimal point is moved to the extreme left position after the first bit with value of 1, and the value is corrected by multiplying with the power of 2

Normalizing number ⇒ 100.101 = 1.00101 x 2^2

normal bit

Each shift of point one place to the left means division by two, shift right by one means multiplication by two.

To preserve the value of the number, we multiply by the power of two

When we move point to the left for n places, we multiply by 2^n **.**

When we move point to the right for n places, we multiply by 2-n .

Presentation of numbers in floating point - IEEE 754 – Example 1

$- 4.625 = - 1.00101 \times 2^2$

31	23 of 22	0
S	E	m

The number is negative ⇒ S = 1

Mantissa without the normal bit ⇒ m = 001010 ... 0

Exponent ⇒ e = 2

Exponent in the presentation by offset 127 (dec) ⇒ E = e + 127 = 127 + 2 = 129 (dec)

E = 129 (dec) = 10000001 (bin)

31	23 of 22	0
1	10000001	00101000000000000000000

A decimal number - 4.624 presented
in floating point single precision format

5.5 Arithmetic with numbers in floating point

- Arithmetic in floating point has always been considered in computers separately from the fixed-point arithmetic

- Basic differences with respect to the fixed-point arithmetic operations are:

  - □ The operations should use in addition to the mantissa also exponent - these operations requires arithmetic in fixed-point arithmetic

  - □ Rounding - the result of the operation should be the mathematically correct values, which are then rounded to the length of mantissa

  - □ When the result of floating point operations, in addition to the oveflow, also underflow can occur

- Overflow: if the result of the operation is greater than the maximum representable number (the exponent is greater than presentable by bits of the exponent)
  - □ If overflow occurs, the result is presented as +∞ or -∞.

- Underflow
  - □ In presenting numbers in floating point also underflow can occur, if the result of the operation is smaller than the smallest presentable number (when the negative exponent is too small for the number of bits of the exponent).
  - □ If there is a underflow, then the number is replaced with zero, or presented as denormalized number.

Execution time of operations (instructions) - comparison

Intel Core architecture (2007): [Kodek]

Ukaz	Latenca
ADD, SUB	1
IMUL	3
IDIV	22
FADD,FSUB	3
FMUL	5
FDIV	32
FSQRT	58
FCOS	119

ARM Cortex M7 (STM32H750)

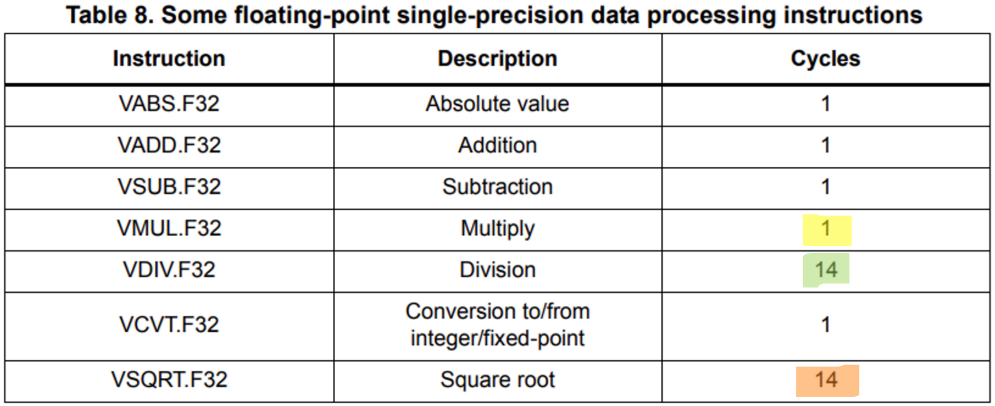


Table 8. Some floating-point single-precision data processing instructions

Instruction	Description	Cycles
VABS.F32	Absolute value	1
VADD.F32	Addition	1
VSUB.F32	Subtraction	1
VMUL.F32	Multiply	1
VDIV.F32	Division	14
VCVT.F32	Conversion to/from integer/fixed-point	1
VSQRT.F32	Square root	14

- Execution time of operations (instructions) – SW vs HW

  ARM Cortex M7 (STM32H750)

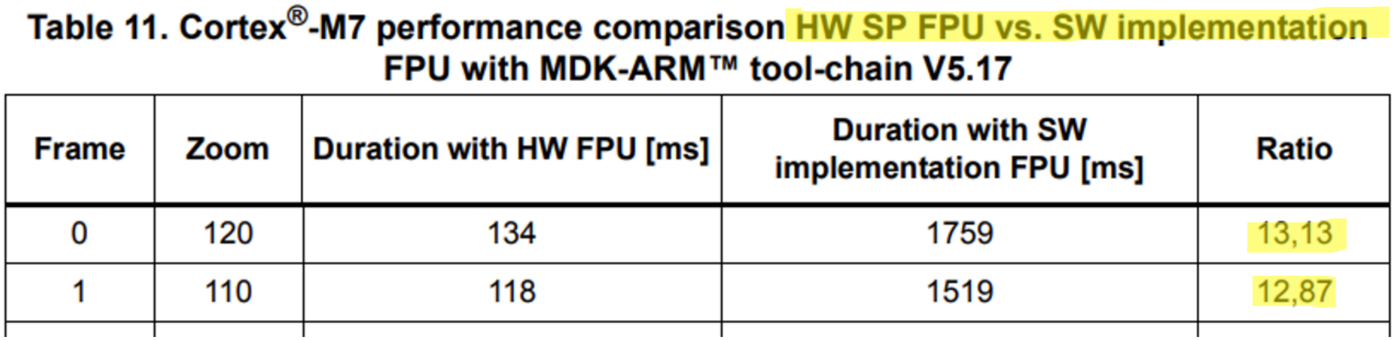


Table 11. Cortex®-M7 performance comparison HW SP FPU vs. SW implementation FPU with MDK-ARM™ tool-chain V5.17

Frame	Zoom	Duration with HW FPU [ms]	Duration with SW implementation FPU [ms]	Ratio
0	120	134	1759	13,13
1	110	118	1519	12,87

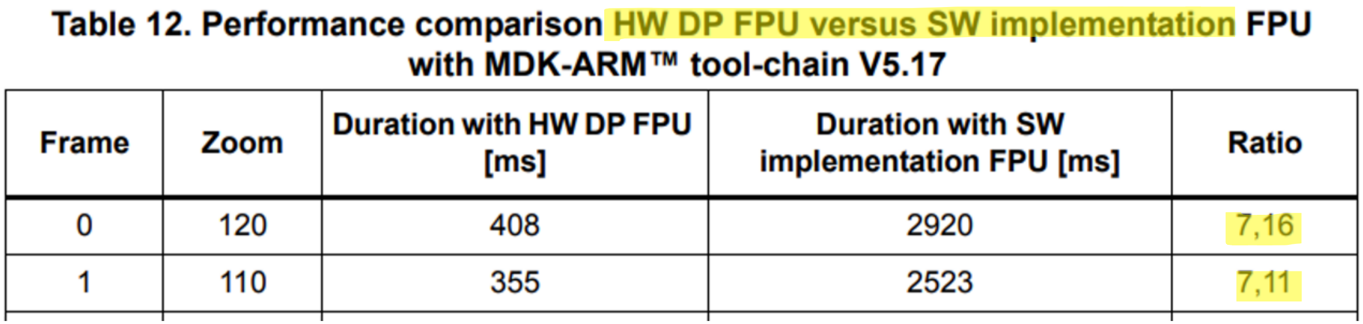
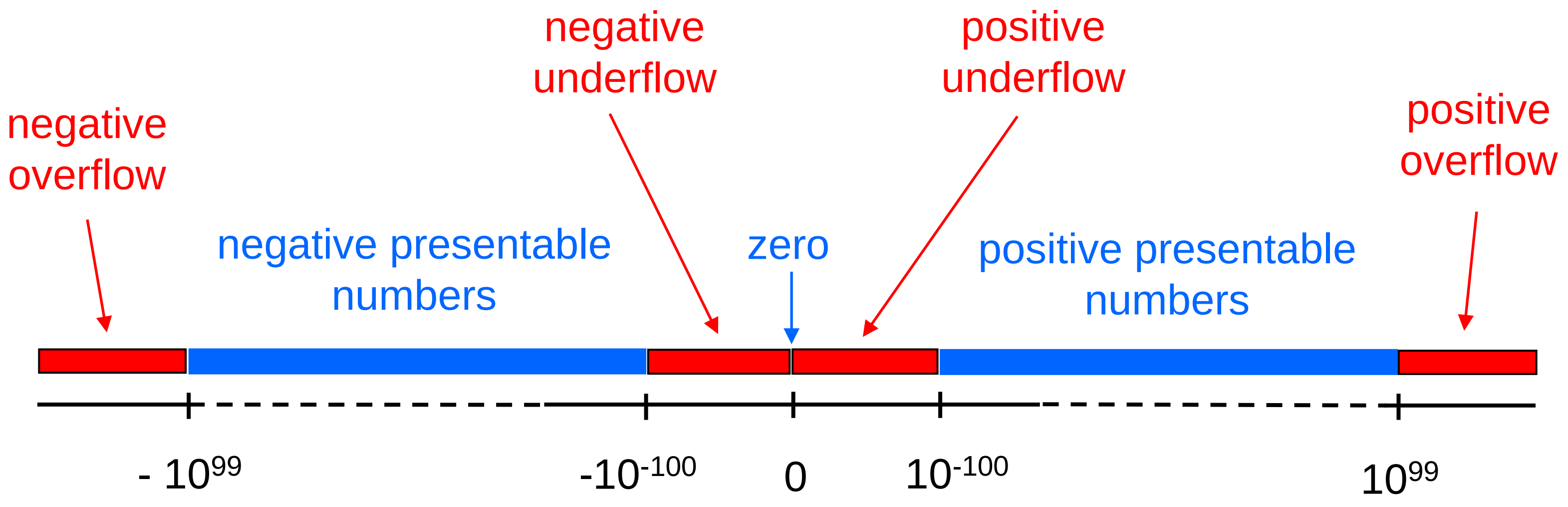


Table 12. Performance comparison HW DP FPU versus SW implementation FPU with MDK-ARM™ tool-chain V5.17

Frame	Zoom	Duration with HW DP FPU [ms]	Duration with SW implementation FPU [ms]	Ratio
0	120	408	2920	7,16
1	110	355	2523	7,11

Example: Numeric lines of decimal real numbers with double-digit exponent and 3-digit mantissa with the range of 0.1 ≤ | m | < 1



Positive: Min 0.1 * 10^-99 = 10^-100 Max 0.999*10^99

5.6 Revisions of the standard IEEE 754:

(IEEE 754 → IEEE 754-2008)

- August 2008: a revised standard IEEE 754-2008 was published that replaces IEEE 754 from y. 1985
  - The most important additions:
    - Two new binary format with a base *r = 2*
      - 128-bit format (quadruple precision) with 112-bit mantissa and 15-bit exponent.
      - 16-bit format (half-precision) with a 10-bit mantissa and a 5-bit exponent.
    - Two new decimal format with a base *r = 10*
      - 64-bit format: 16-digit mantissa (16 decimal digits)
      - 128-bit format: 34-digit mantissa

Presentation of floating point numbers - updated IEEE 754-2008

- The standard IEEE 754-2008 defines:
  - Six basic formats, four binary and two decimal.
  - Arithmetic formats that are used in arithmetic and other operations.
  - Exchange formats used for exchanging operands in floating point
  - Algorithms for rounding, which determine the methods of rounding numbers on calculations and conversions
  - Arithmetic and other operations on the arithmetic formats.
  - Procedures in case of extraordinary events (division by 0, overflow, underflow, ...).
- Latest revision : IEEE 754-2019

Presentation of floating point numbers - updated IEEE 754-2008

Designation	Name	Base	Number of places in mantissa *	E min	E max	Decimal accuracy	Max decimal exponent
binary32	Single precision	2	23 + 1	-126	+127	7.22	38,23
binary64	Double precision	2	52 + 1	-1022	+1023	15.95	307.95
binary128	quadruple accuracy	2	112 + 1	-16,382	+16383	34.02	4931.77
decimal64		10	15 + 1	-383	+384	16	384
decimal128		10	33 + 1	-6143	+6144	34	6144

* mantissa + 1 bit for the sign