Topic 4
representing information

Computer technology

Topic 4

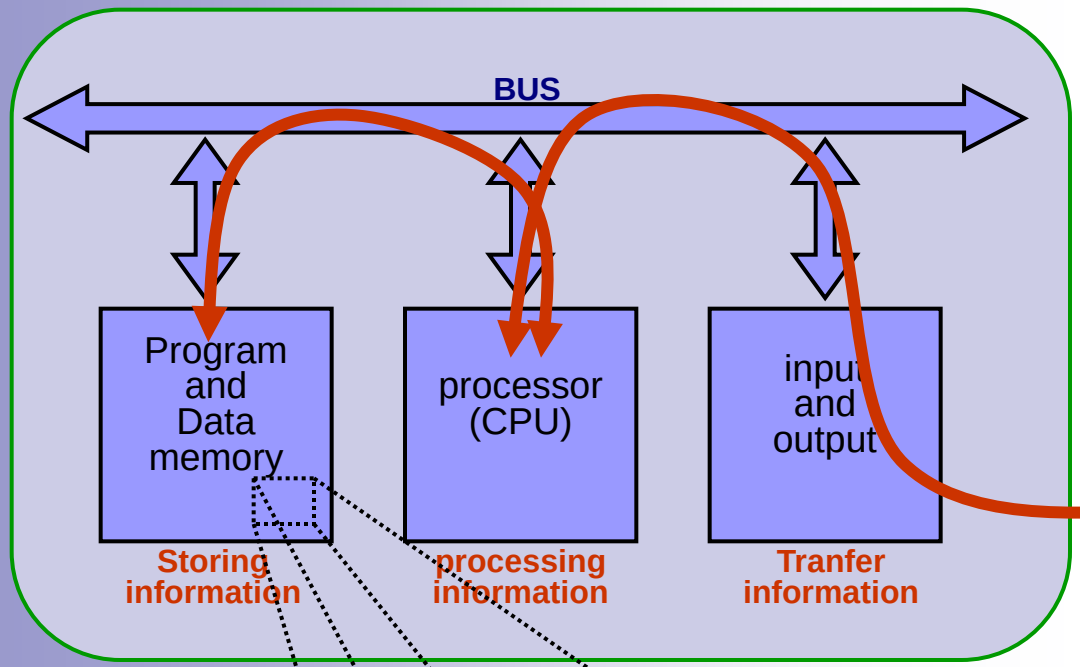
- introduction
  - representing information
- characters and strings
  - representing strings
- numerical information
  - representing integers
    - positional representation: unsigned, signed magnitude, two's complement.
    - nonpositional representation: excess and BCD
  - representing real numbers
    - fixed point
    - floating point: IEEE 754
- adding redundancy: detection and correction of errors
  - parity code
  - Hamming code SEC, SEC/DED

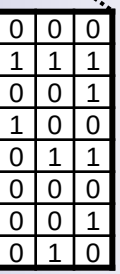
representing information

introduction

introduction

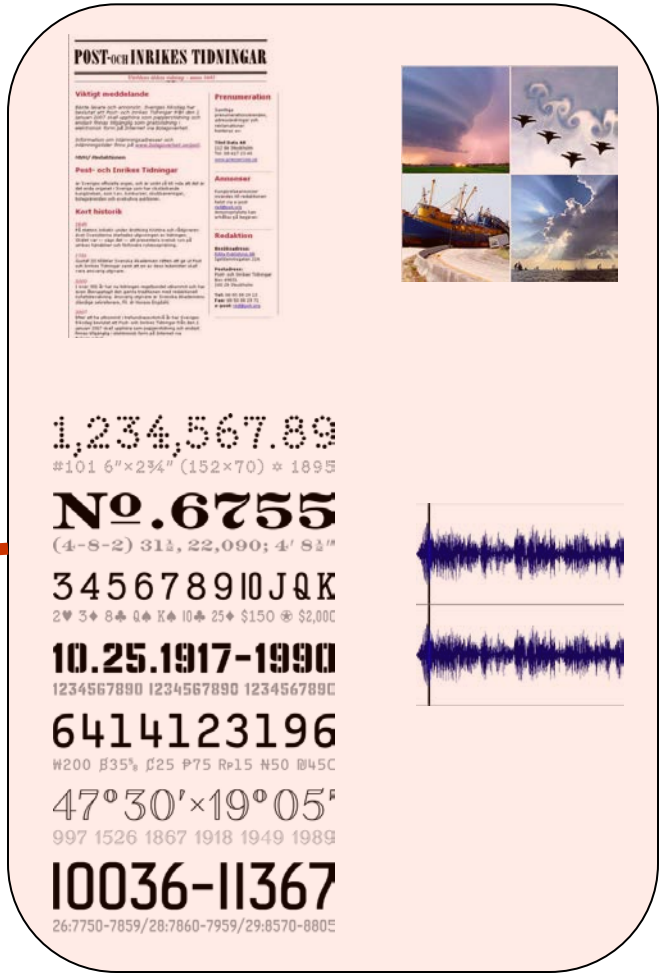
computer





representing

information



introduction

- Representing information of different size
  - Binary type: only can take two states
  - Finite range: bounded representations
  - Functional units work with bit string of a size (n)

  Between 0 and 1 there are infinitely many numbers
- Standard sizes
  - Characteristic size of the representation space
  - Computers use several sizes
  - Called: byte, word, double word, nibble …

introduction

How to represent several types of information in "n" bits. Focus on

- characters and strings: character strings
- numerical information
  - representing integers: positional representation and nonpositional representation
  - representing real numbers: fixed point and floating point

Topic 4

- introduction
  - representing information
- characters and strings
  - representing character strings
- numerical information
  - representing integers
    - positional representation: unsigned, signed magnitude, two's complement.
    - nonpositional representation: excess and BCD.
  - representing real numbers
    - fixed point
    - floating point: IEEE 754
- adding redundancy: detection and correction of errors
  - parity code
  - Hamming code SEC, SEC/DED

Representing alphanumerical info

characters and strings

Representing alphanumerical info

- representing each character by a binary code (n-bits)
  - n determines the number of characters that can be represented
  - code: table with representation of each character
  - example of standard:
    - ASCII: 7 bits, basic characters (letter, number, symbols …)
    - extended ASCII: 8 bits, includes also special characters of certain languages (e.g. ñ, á, é, ö, …)
    - UNICODE: 16 bits, characters of nearly all alphabets

ASCII code

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

extended ASCII

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	│	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	169	_	185	╣	201	╔	217	┘	233	Θ	249	·
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	▉	235	δ	251	√
139	ï	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	_
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	ϕ	253	²
141	ì	158	_	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	░	192	└	208	╨	224	α	240	≡		

Source: www.LookupTables.com

Representing alphanumerical info

- representing strings
  - How to represent string “Zero” in ASCII?
    - Z = $5A_{hex}$, e = 65_{hex}, r = 72_{hex}, o = $6F_{hex}$
    - Ending character (0_{hex}):
      - 01011010 01100101 01110010 01101111 00000000
    - number of starting characters:
      - 00000100 01000011 01100101 01110010 01101111

Topic 4

- introduction
  - representing information
- characters and strings
  - representing character strings
- numerical information
  - representing integers
    - positional representation: unsigned, signed magnitude, two's complement.
    - nonpositional representation: excess and BCD.
  - representing real numbers
    - fixed point
    - floating point: IEEE 754
- adding redundancy: detection and correction of errors
  - parity code
  - Hamming code SEC, SEC/DED

Integers: positional representation

numerical information

Integers: positional representation

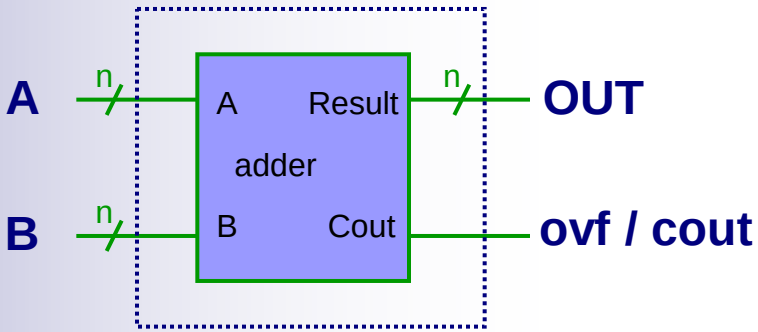
- Position: Number value depends on digit positions in the string
- number X represented by string $x_n..x_1x_0$ base *b*
  - value V(X) = $\sum_{i=0}^{n} x_i b^i$
  - digits $x_i \in \{0,1,..,b\text{-}1\}$
  - example:
    - string $x_3..x_1x_0$ =“1234” and *b* = 10
      - $V(X) = 1*10^3+2*10^2+3*10^1+4*10^0 = 1000+200+30+4$

Integers: positional representation: unsigned

- Pure binary
  - The “n” bit string represents a number in base 2
    - string X = $x_{n-1}\ x_{n-2}\ \ldots\ x_1\ x_0$
    - value V(X) = $2^{n-1}x_{n-1} + \ldots + 2x_1 + x_0$
  - Represent natural number by “n” bits
  - Value range of “n” bits representation
    - Minimum = 0…00 = 0
    - Maximum = 1…11 = 2^n-1
    - Range is [0, 2^n-1]
    - example n=8 bits => range is [0, 255]

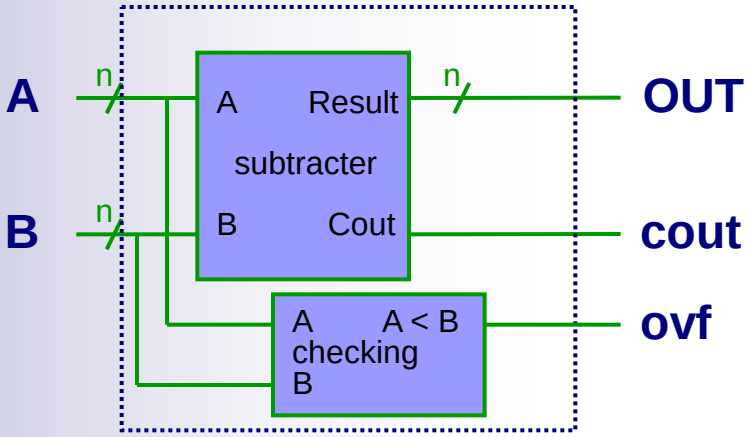
unsigned

- Pure binary operations
  - adding A+B:
    - normal adding in binary.
    - Overflow: result operation cannot be represented
      - Overflow → most significant is carried out (Section 3.2)



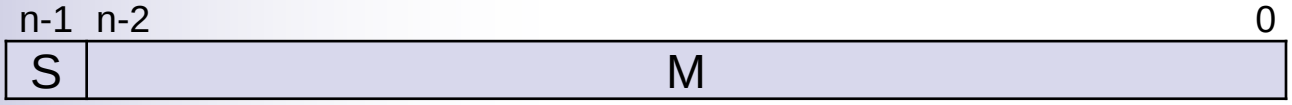
unsigned

- Pure binary operations
  - Subtract, A-B:
    - Check if A >= B as negative values cannot be represented
    - Usual binary subtraction
      - Different circuit than adder
    - No overflow occurs if A >= B



Integers: positional representation

- signed magnitude (n bits)
  - 1 bit represents the sign (S)
  - "n-1" bits string represents the magnitude (M) of the number (absolute value) in base 2
  - example:



    - string X = $x_{n-1}\ x_{n-2}\ \ldots\ x_1\ x_0$
    - value V(X) = $(-1)^{x_{n-1}} \times (2^{n-2} x_{n-2} + \ldots + 2x_1 + x_0)$
  - How to represent?
    - represent the absolute number value by "n-1" bits
    - Add sign bit
      - 0 if positive
      - 1 if negative
  - important: the sign bit does not contribute to the value of the number

Ex. 4 bits	
7	0111
-3	1011

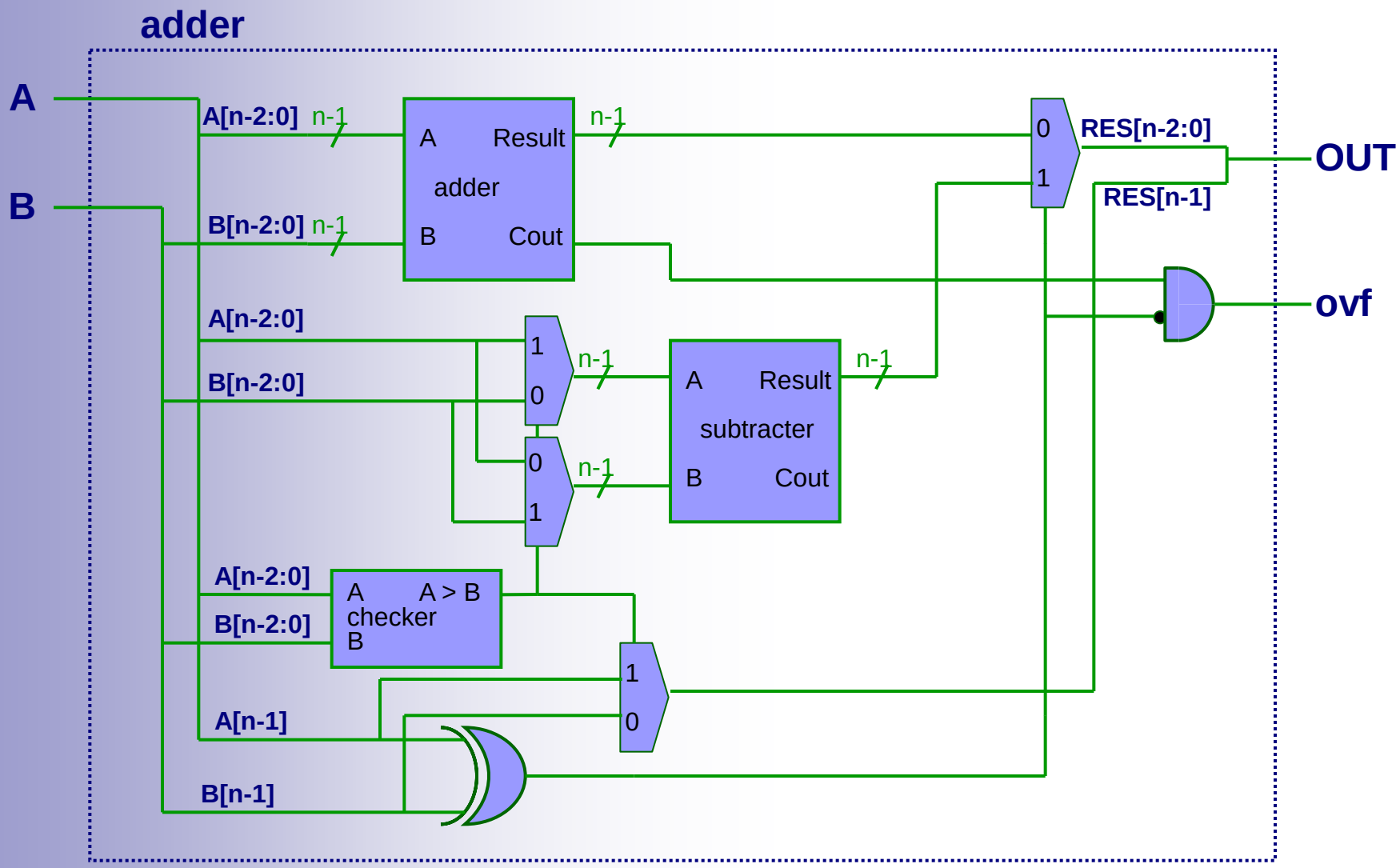
signed magnitude

- Range of values with “n” bits
  - minimum = 11…11 = $-(2^{n-1}-1)$
  - maximum = 01…11 = $2^{n-1}-1$
  - Range is $[-(2^{n-1}-1), 2^{n-1}-1]$
  - Double representation of 0
    - 00…00
    - 10…00
  - example n=8 bits => range is [-127, 127]
- Extending from n (A) to m (B) bits (m>n)
  - $B_{m-1} = A_{n-1}$ (copy the sign)
  - $B_{m-2} \ldots B_{n-1} = 0$ (fill added bits up with 0)
  - B[n-2:0] = A[n-2:0]
  - Example (from 8 to 16): 10001110 => 1000000000001110

signed magnitude

- signed magnitude operations
  - add A+B:
    - Check sign bit of the operands (SA, SB)
    - if SA=SB
      - Usual binary add of the n-1 magnitude bits
      - sign of the result = SA = SB
      - Overflow →carry out adding bit n-2
    - if SA <> SB
      - Subtract from operand with larger absolute value the n-1 magnitude bits of the operand with smaller absolute value
      - sign of the result = sign of the larger operand
      - overflow cannot occur
  - subtract A-B:
    - Is A+(-B) , i.e. change the sign bit of B and apply add

signed magnitude



two's complement (2C)

- if number positive

  Represent in binary with 0 in most significant bit

Ex. 4 bits	
7	0111
-3	1101

  if number negative

  Make a binary representation of absolute value with 0 in most significant bit and then perform a two's complement

- Value range with “n” bits
  - minimum = 10…00 = $-(2^{n-1})$
  - maximum = 01…11 = $2^{n-1}-1$
  - Range is $[-(2^{n-1}), 2^{n-1}-1]$
  - No double representation of 0
  - example n=8 bits => range is [-128, 127]

two's complement (2C)

Extending from n (A) to m (B) bits (m>n)

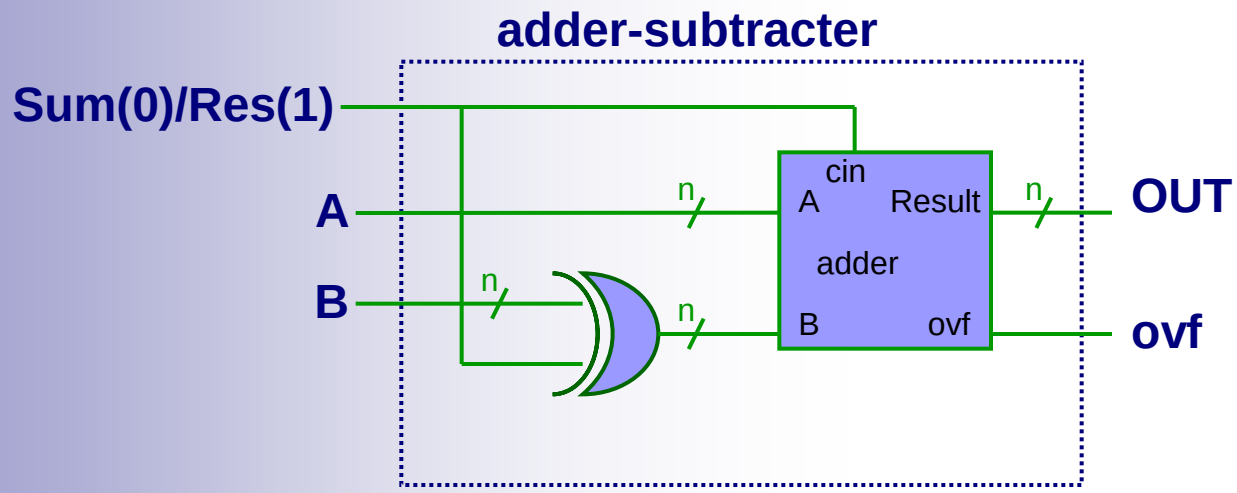
- sign extension (repeat sign) $B_{m-1} \ldots B_n = A_{n-1}$
- B[n-1:0] = A[n-1:0]
- Example (from 8 to 16): 10001110 => 1111111110001110

two's complement (2C)

two's complement operations

- add A+B:
  - Usual binary adding of the n bits of the number, ignoring the last carry out
    - Overflow → if carry out, then bit n-2 differs from carry out of adding bit n-1
      - if adding two equal sign numbers changes the sign
      - Adding two numbers of different sign cannot cause overflow
- subtract A-B is A+(-B)
  - Change sign B => perform two's complement on B

two's complement (2C)



- examples (4 bits):
  - add -2,3
    - -2 => -(0010) => (1110)
    - 3 => 0011
    - Res = 0001 (pos) => 1
  - subtract -2,3
    - -2+(-3)
    - -3 => -(0011) => 1101
    - Res = 1011 (neg) => -(0101) = -5

add	subtract
1110	1110
0011	1101
X 0001	X 1011

integer nonpositional representation

numerical information

integer nonpositional representation: Excess-K (offset binary)

- Excess-K (n bits)
  - add K and do unsigned binary representation
    - string X = $x_{n-1}\ x_{n-2}\ \ldots\ x_1\ x_0$
    - value V(X) = $(2^{n-1}x_{n-1} + \ldots + 2x_1 + x_0) - K$
  - representation
    - represent the value V+K in unsigned binary
    - Example, let K = 7

Ex. 4 bits	
7	1110
-7	0000
-3	0100

Excess-K

- Value range with “n” bits
  - minimum = 00…00 = (0-K) = -K
  - maximum = 11…11 = 2^n-1-K
  - Range is [-K, 2^n-1-K]
  - No double representation of 0
  - example n=8 bits K= 127 => range is [-127, 128]
    - Bigger K => more negative numbers represented
    - Smaller K => more positive numbers represented
- Extending from n (A) to m (B) bits (m>n)
  - Add 0’s: $B_{m-1} \ldots B_n = 0$
  - B[n-1:0] = A[n-1:0]
  - Example (from 8 to 16): 10001110 => 0000000010001110

integer nonpositional representation:
BCD: binary coded decimal

- Represent each decimal digit of the number in binary
  - 4 bits for each decimal digit
- Example
  - 3248 => 0011 0010 0100 1000
  - 0101 1001 0000 => 590
- Low efficiency
  - We do not use 37.5% of representation capacity for each digit (10 of 16)
- applications where decimal error or rounding is important

BCD: binary coded decimal

- Arithmetic operations follow decimal digit conventions
- add
  - Add decimal digits in groups of 4 bits
  - Decimal carry out if
    - Binary carry out occurs adding the 4 bits of the decimal digits
    - If the sum of the numbers represented by the 4 bits is larger than 9
  - At decimal carry out, correct the digit of the sum
    - add 6 to the digit result (16-10 = 6) → example

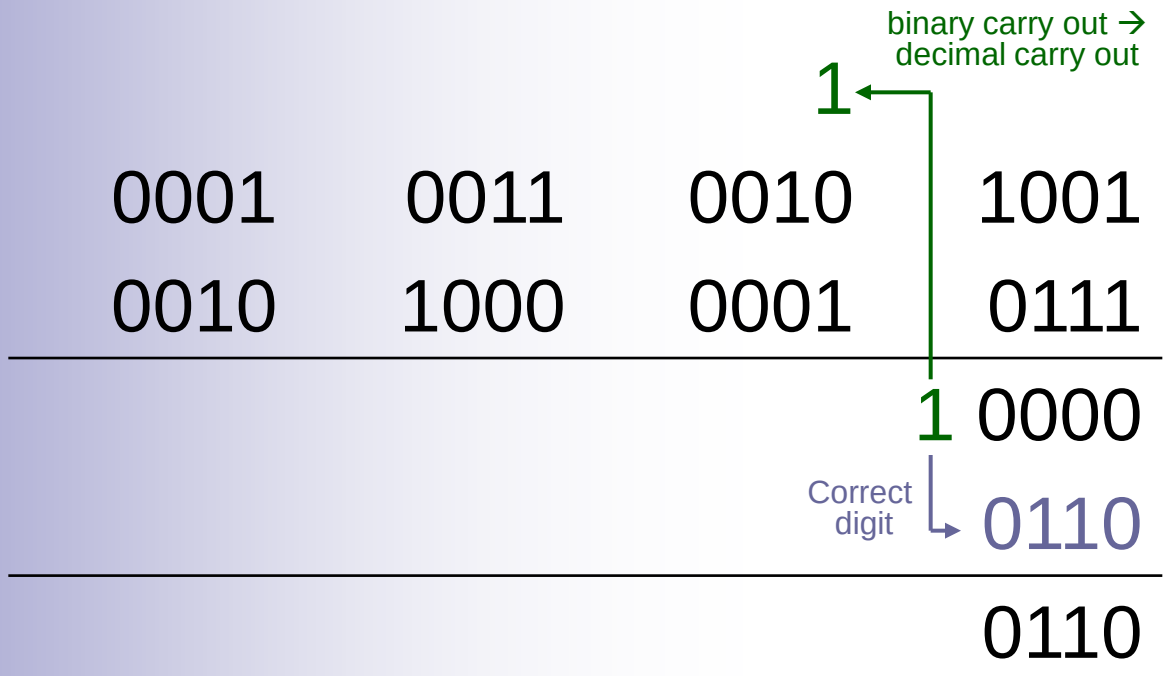
BCD: binary coded decimal

- example: 1329 + 2817 = 4146

0001	0011	0010	1001
0010	1000	0001	0111

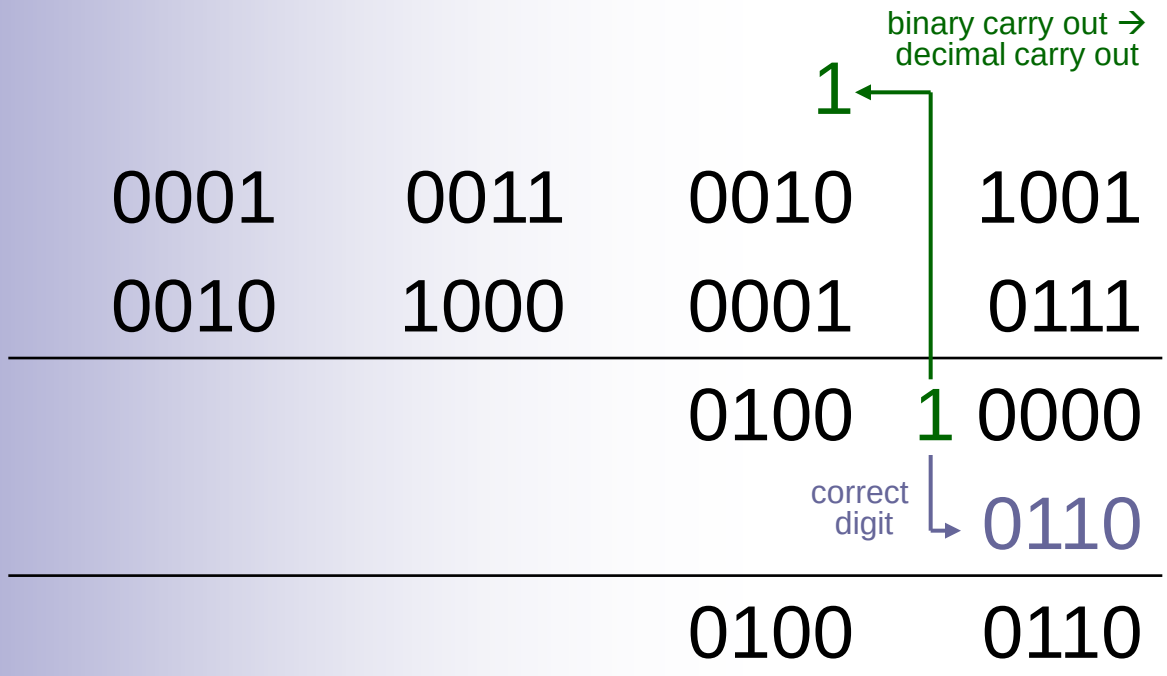
BCD: binary coded decimal

- example: 1329 + 2817 = 4146



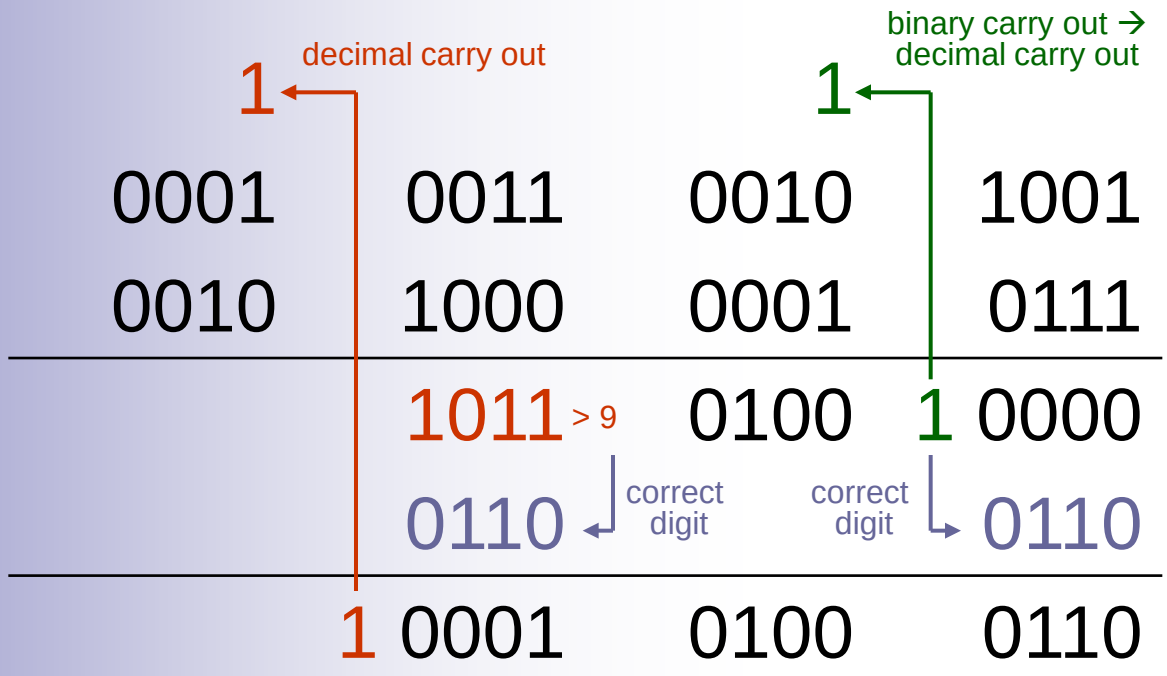
BCD: binary coded decimal

- example: 1329 + 2817 = 4146



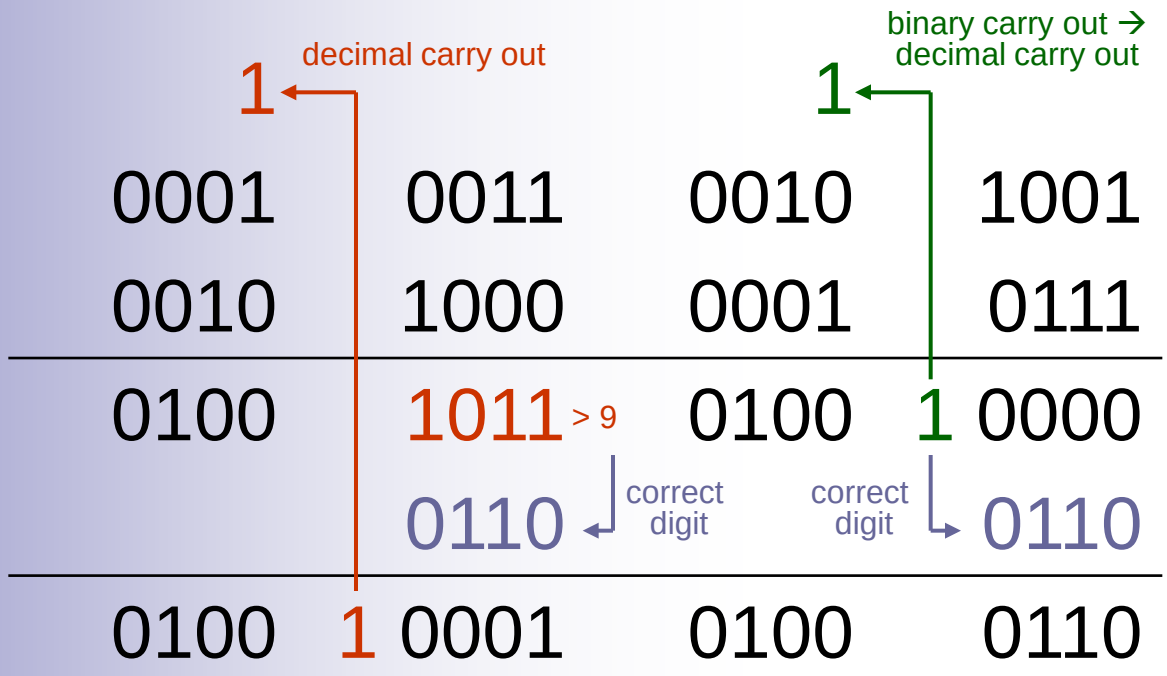
BCD: binary coded decimal

- example: 1329 + 2817 = 4146

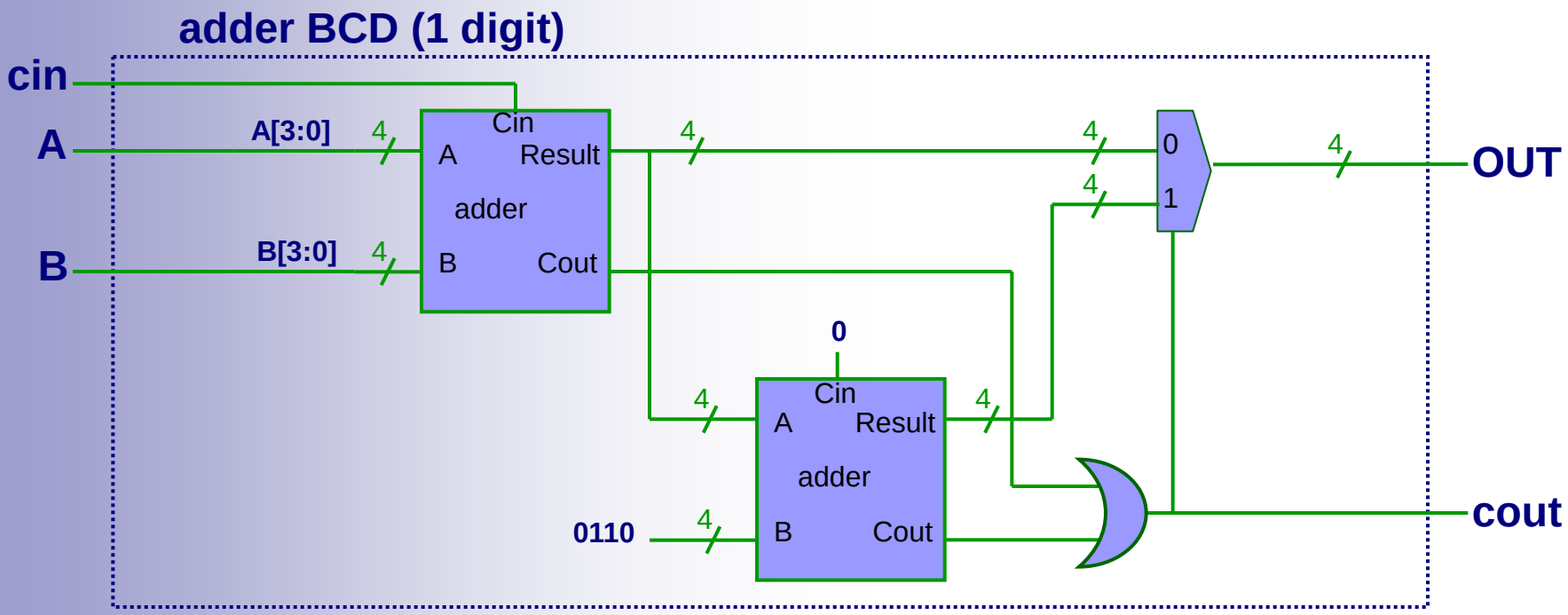


BCD: binary coded decimal

- example: 1329 + 2817 = 4146



BCD: binary coded decimal



integers: examples

- Give the represented decimal value of the bit strings following the different representation systems

Binary string	unsigned	SM	2C	excess 128	BCD
01111111					
10000000					
00000001					
10000001					

integers: examples

- Give the binary string of 8 bits that represents the given decimal natural numbers according to the representation system

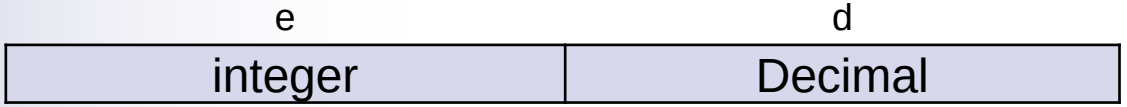
value	unsigned	SM	2C	excess 128	BCD
25					
-10					
0					
-99					

Fixed point representation of real numbers

numerical information

real numbers: fixed point

- real numbers in fixed point (n bits)
  - Fix position for the decimal point in the n bits
    - e bits for integer part
    - d bits for decimal part
    - e+d = n



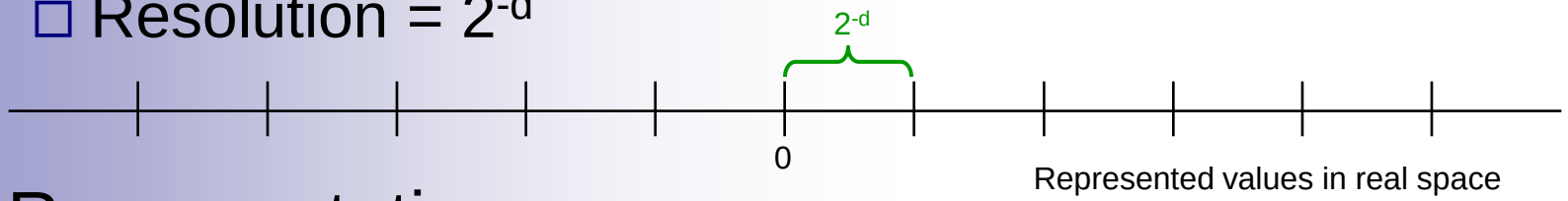
- string decimal part X = $x_{e-1}..x_1x_0x_{-1}x_{-2}..x_{-d}$
  - V(X) = $\sum_{i=e-1}^{-d} x_i 2^i$
  - example:
    - string X="1000.101" (n=7, e=4, d=3)
      - V(X)=$1*2^3+1*2^{-1}+1*2^{-3}$ = 8+0.5+0.125 = 8.625

real numbers: fixed point

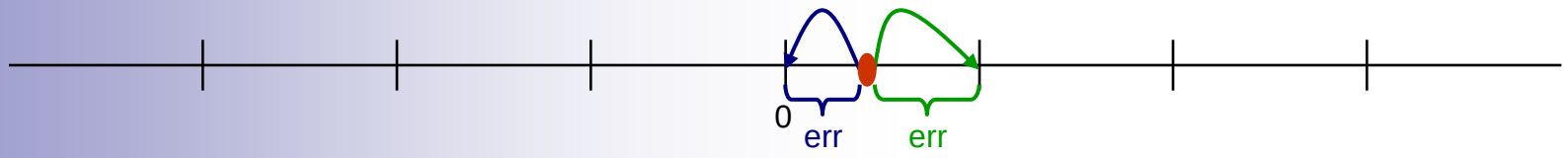
- representing the sign
  - Any of the positional representation systems for integer numbers (SM, 2C)
- Example -19.325 with e=6 and d=4
  - Towards binary representation: -10011.0101
    - SM: 1100110101
    - 2C: 1011001011
- Value range (2C)
  - minimum = 1000000000 = -(100000.0000) = -32
  - maximum = 0111111111 = 011111.1111 = 31.9375
  - Range is[-32, 31.9375]

real numbers: fixed point

- Resolution / precision
  - Difference between two consecutive represented values: value of the least significant bit
  - Uniform over the complete represented range
  - Resolution = 2^{-d}



- Representation error
  - difference between real value and its representation is less than the resolution
  - $e_a \leq 2^{-d}$



representing real numbers in floating point

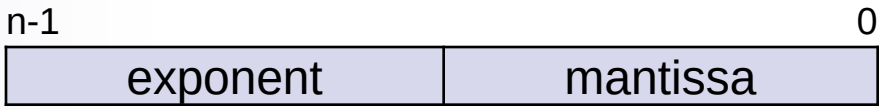
numerical information

real numbers: floating point

- Drawback of fixed point:
  - Fixed resolution → same format for numbers of very different size
    - Ex. N=10 digits
      - resolution 10^{10} : can represent 2000000000
      - resolution 10^{-10} : can represent 0.0000000002
    - What format to represent both?
- floating point representation
  - Compact representation
  - Base for scientific notation

real numbers: floating point

- Example
  - 20000000000 → $2*10^{10}$
    - mantissa = 2, base = 10, exponent = 10
  - 0.0000000002 → $2*10^{-10}$
    - mantissa = 2, base = 10, exponent = -10
- value determined by three elements
  - Fixing the base (2), no explicit mentioning
- Represent the two elements in n bits
  - exponent (characteristic) → integer
  - mantissa (significand) → fixed point

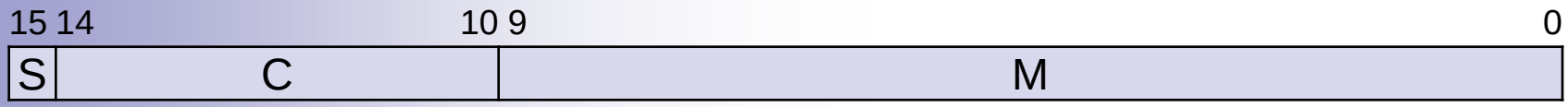


real numbers: floating point

- Infinitely many representations of a value
  - 10000000000 → $1*2^{10} = 10*2^{9} = 100*2^{8} = 0.1*2^{11} = 0.001*2^{13} = \ldots$
  - Which one to choose?
- Normalization
  - for each value determine its unique representation
    - value → unique exponent and mantissa
    - Example Normalize $0.5 \leq$ mantissa < 1 towards $0.1*2^{11}$
- A floating point format is determined by
  - Total number of used bits (exponent, mantissa)
  - format of representing the exponent and mantissa
  - Normalization

real numbers: floating point

- example format floating point:
  - size 16 bits
    - 5 bits for the characteristic
    - 11 bits for the mantissa
  - characteristic: exponent represented in excess 16
  - mantissa normalized fraction represented in SM
    - Normalization: 0.5 ≤ mantissa < 1
      - Adjust exponent up to having the mantissa from 0.1… (the most significant 1 of the mantissa at the right of the decimal point)
  - value= $(-1)^S \times M \times 2^{C-16}$

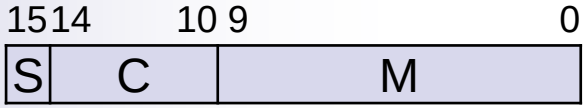


real numbers: floating point

- Example: represent -17.171875

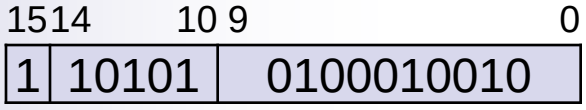
$$-17.171875 = -17\ \frac{11}{2^6} = -16 - 1\ \frac{11}{2^6}$$

1. Make binary
2. Normalize (0.5 ≤ mantissa < 1)
3. Represent as

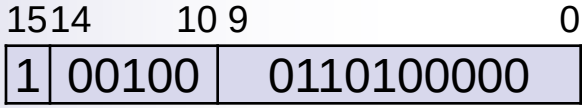


real numbers: floating point

- Example: represent -17.171875
  1. Make binary
     - 10001.001011
  2. Normalize (0.5 ≤ mantissa < 1)
     - 0.10001001011×2^5
  3. represent
     - C = E+16 = 5+16 = 21 → 10101
     - S = 1 (negative)
     - M = 0100010010

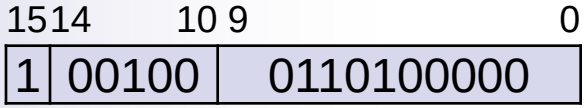


real numbers: floating point



- Example: value represented by 1001000110100000

real numbers: floating point

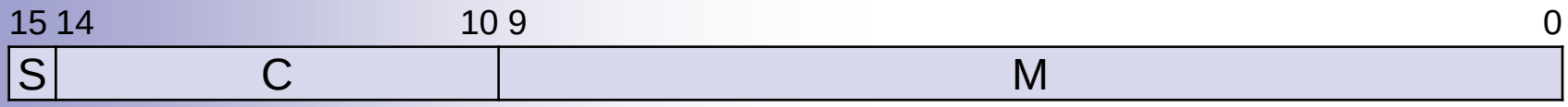


- Example: value represented by 1001000110100000
  1. Field values
     - S = 1
     - C = 00100 → E = C-16 = 4-16 = -12
     - M = 01101 → 0.1101
  2. Represented value $V=(-1)^S \times M \times 2^{C-16}$
     - $V = (-1)^1 \times 0.1101 \times 2^{-12} = -1101 \times 2^{-16} = -13 \times 2^{-16} = -1.98 \times 10^{-4}$

real numbers: floating point

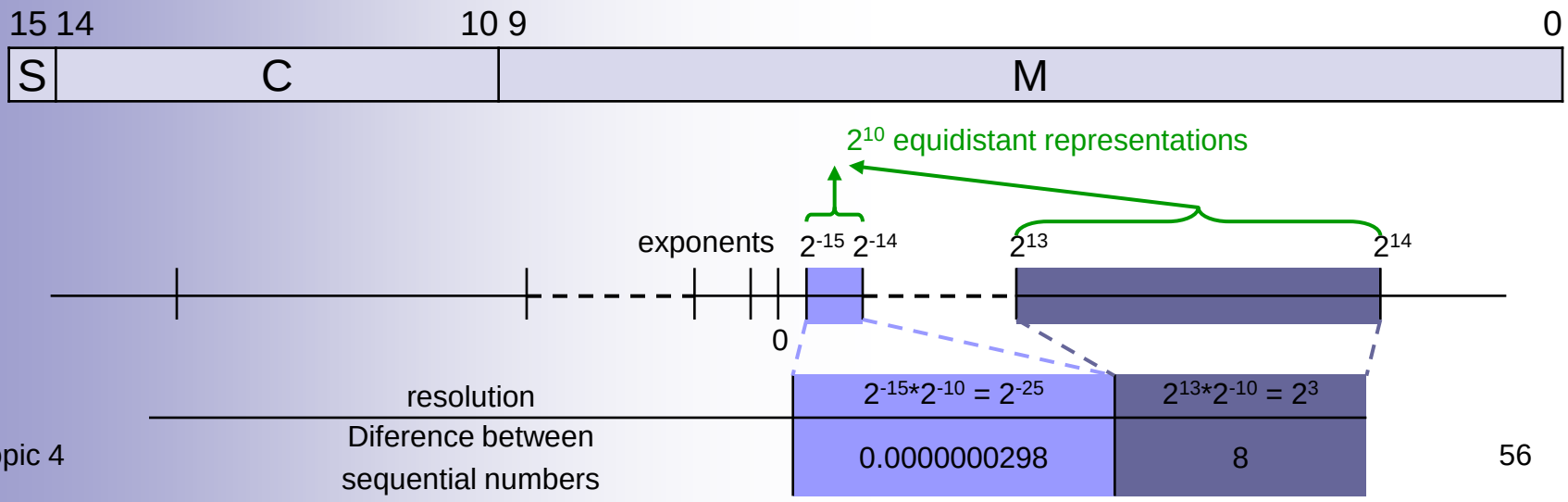
- format questions
  - loss of 2 bits in the mantissa
    - All mantissas start by 01 (2 fixed bits)
    - Gain 2 bits in the representation by making them implicit
      - represent in M the mantissa bits at the right side of 0.1
      - $(-1)^S \times 0.1M \times 2^{C-16}$
  - How to represent the number 0?
    - Special case, mantissa should not be normalized:
      - M=0000000000 (0.5 ≤ mantissa < 1)

X



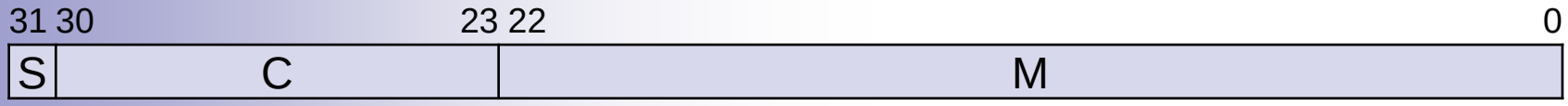
real numbers: floating point

- Resolution
  - value least significant bit scaled by exponent value → non-uniform resolution
  - higher resolution close to 0
  - less resolution for large values



real numbers: IEEE 754 standard

- Standardize for sharing float data among different architectures
- IEEE 754 standard simple precision
  - size 32 bits
    - 8 bits for the characteristic
    - 24 bits for the mantissa (1 sign, 23 magnitude)
  - characteristic: exponent represented in excess 127
  - mantissa normalized fraction represented in SM
    - Normalization: 1 ≤ mantissa < 2
      - Adjust exponent such that the mantissa has the shape 1. (the most significant 1 left of the point)
      - 1 implicit most significant 1
  - value number = $(-1)^S \times 1.M \times 2^{C-127}$



real numbers: IEEE 754 standard

- special cases

characteristic (C)	mantissa (M)	
255 (11111111)	0	S=0 → +infinite S=1 → -infinite
	≠ 0	NaN (Not a Number), (0/0 …)
0 (00000000)	0	zero (0)
	≠ 0	non-normalized numbers: $V=(-1)^S \times 0.M \times 2^{-126}$

- case C=0, M≠0

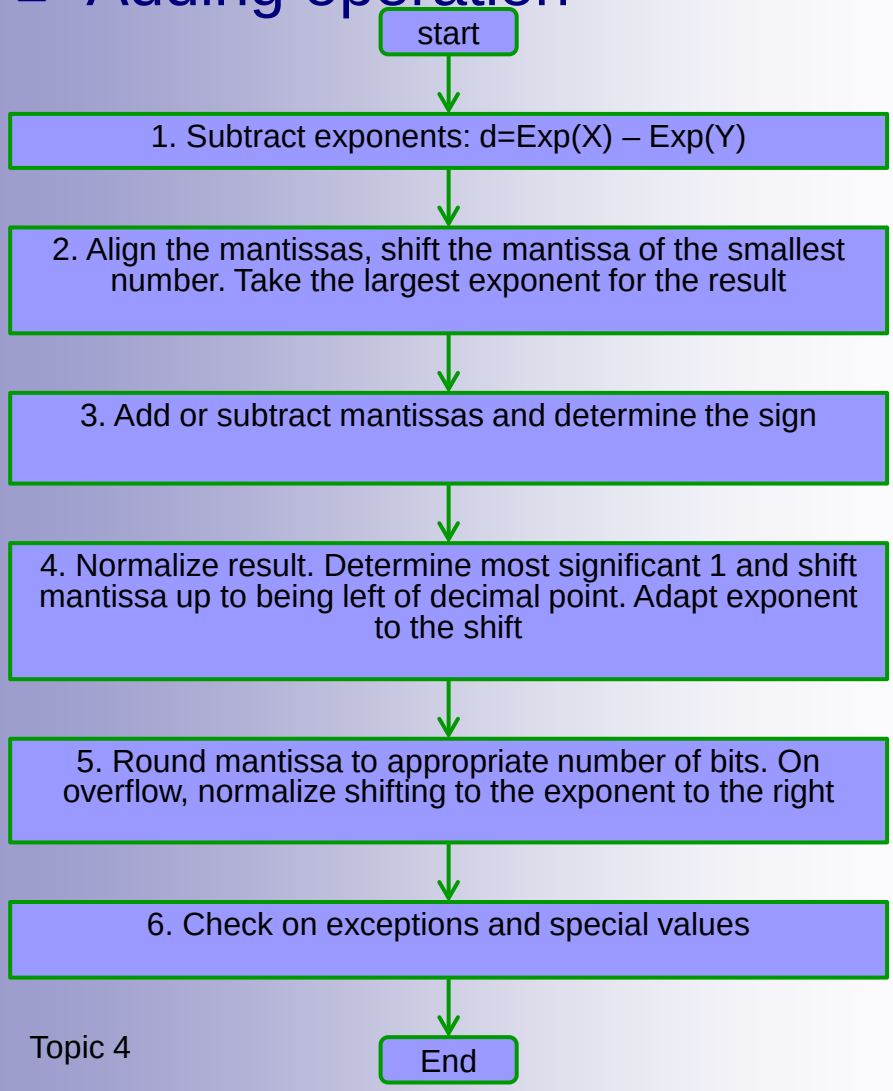
  Representing very small numbers (close to 0)

  Without normalization, i.e. without implicit “1”

  - The most significant “1” may be in any position in M

real numbers: IEEE 754 standard

- Adding operation



example of add for 4 bits mantissa

(X=14) + (Y=3)

$X = 1110 = 1.110 \times 2^3$ and $Y = 11 = 1.100 \times 2^1$

Additional precision bits in adder

d = 3 – 1 = 2

1.1100×2^3

$+ \ 1.1000 \times 2^1$ → 2

Desp. 2 a the derecha Y

1.1100×2^3

$+0.0110 \times 2^3$

Same sign → add

10.0010×2^3

Most significant 1 is not just left of decimal point. Shift 1

10.0010×2^3 → 1

1.0001×2^4

Round to correct result number of bits

1.0001×2^4

$1.001 \times 2^4 = 18$

real numbers: IEEE 754 standard

- Hardware adder

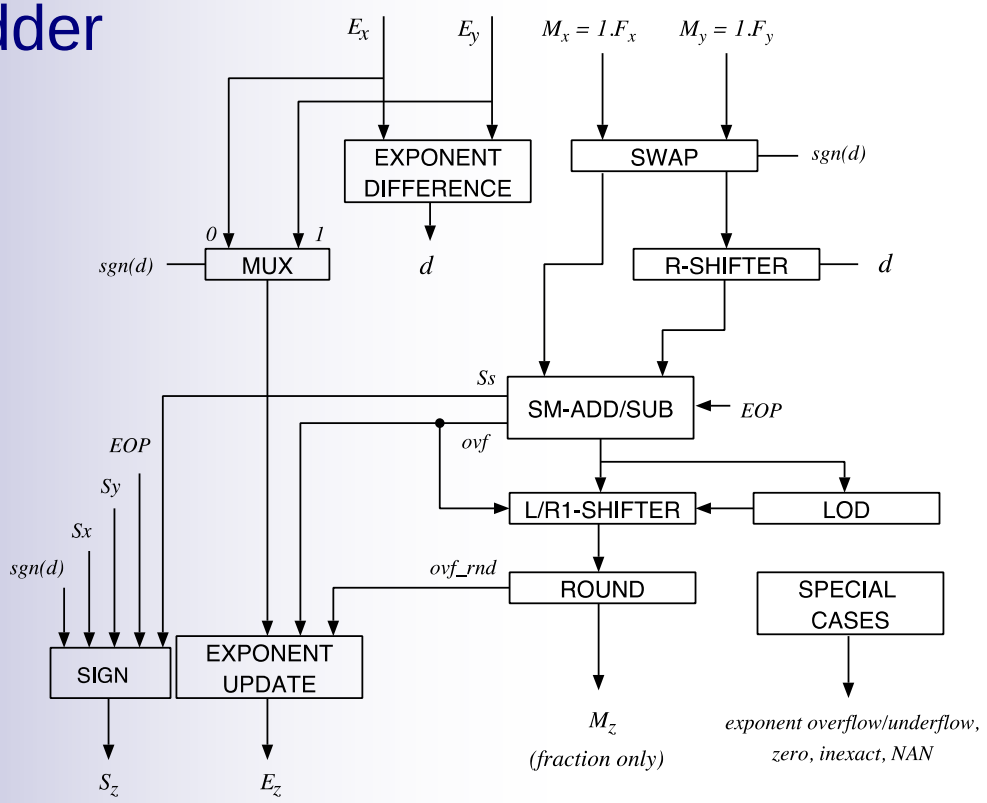


Figure 8.5: BASIC IMPLEMENTATION OF FLOATING-POINT ADDITION.

IEEE 754: examples

- Determine the decimal values of the following binary strings that represent IEEE 754 simple precision floating points
  - $80280000_{(16}$
  - $41200000_{(16}$
- represent the following real number in simple precision IEEE 754 format
  - 13×2^{-132}

IEEE 754: examples

- Determine the decimal values of the following binary strings that represent IEEE 754 simple precision floating points
  - $80280000_{(16}$ = 1000 0000 0010 1000 0000 0000 0000 0000
    - S=1, C=0, M=0101
      - C=0 → special case, M ≠ 0 → non-normalized number
      - $V=(-1)^S \times 0.M \times 2^{-126}$ → $(-1)^1 \times 0.0101 \times 2^{-126} = -101 \times 2^{-130} = -5 \times 2^{-130}$
  - $41200000_{(16}$
- represent the following real number in simple precision IEEE 754 format
  - 13×2^{-132}

IEEE 754: examples

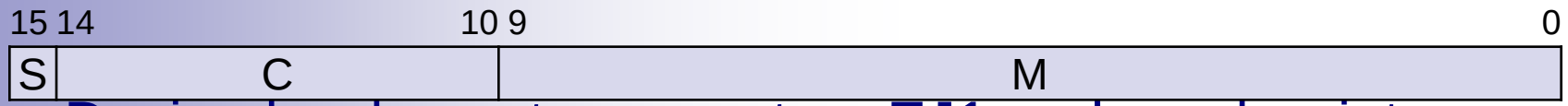
- Determine the decimal values of the following binary strings that represent IEEE 754 simple precision floating points
  - $80280000_{(16}$ = 1000 0000 0010 1000 0000 0000 0000 0000
    - S=1, C=0, M=0101
      - C=0 → case especial, M ≠ 0 → non-normalized number
      - $V=(-1)^S \times 0.M \times 2^{-126} \rightarrow (-1)^1 \times 0.0101 \times 2^{-126} = -101 \times 2^{-130} = -5 \times 2^{-130}$
  - $41200000_{(16}$ = 0100 0001 0010 0000 0000 0000 0000 0000
    - S=0, C=10000010 = 130, M=01
      - $(-1)^S \times 1.M \times 2^{C-127} \rightarrow (-1)^0 \times 1.01 \times 2^{130-127} = 1.01 \times 2^3 = 1010 = 10_{(10}$
- represent the following real number in simple precision IEEE 754 format
  - 13×2^{-132}

IEEE 754: examples

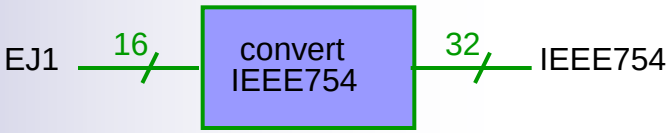
- Determine the decimal values of the following binary strings that represent IEEE 754 simple precision floating points
  - $80280000_{(16}$ = 1000 0000 0010 1000 0000 0000 0000 0000
    - S=1, C=0, M=0101
      - C=0 → case especial, M ≠ 0 → non-normalized number
      - $V=(-1)^S \times 0.M \times 2^{-126} \rightarrow (-1)^1 \times 0.0101 \times 2^{-126} = -101 \times 2^{-130} = -5 \times 2^{-130}$
  - $41200000_{(16}$ = 0100 0001 0010 0000 0000 0000 0000 0000
    - S=0, C=10000010 = 130, M=01
      - $(-1)^S \times 1.M \times 2^{C-127} \rightarrow (-1)^0 \times 1.01 \times 2^{130-127} = 1.01 \times 2^3 = 1010 = 10_{(10}$
- represent the following real number in simple precision IEEE 754 format
  - 13×2^{-132}
    - $1101 \times 2^{-132} = 1.101 \times 2^{-129}$ → exponent cannot be represented in excess 127
    - ¿non-normalized number? Can the exponent -126 be used?
      - $1101 \times 2^{-132} = 0.001101 \times 2^{-126}$ → S=0, M=001101
      - 0 00000000 00110100000000000000000

exercise floating point

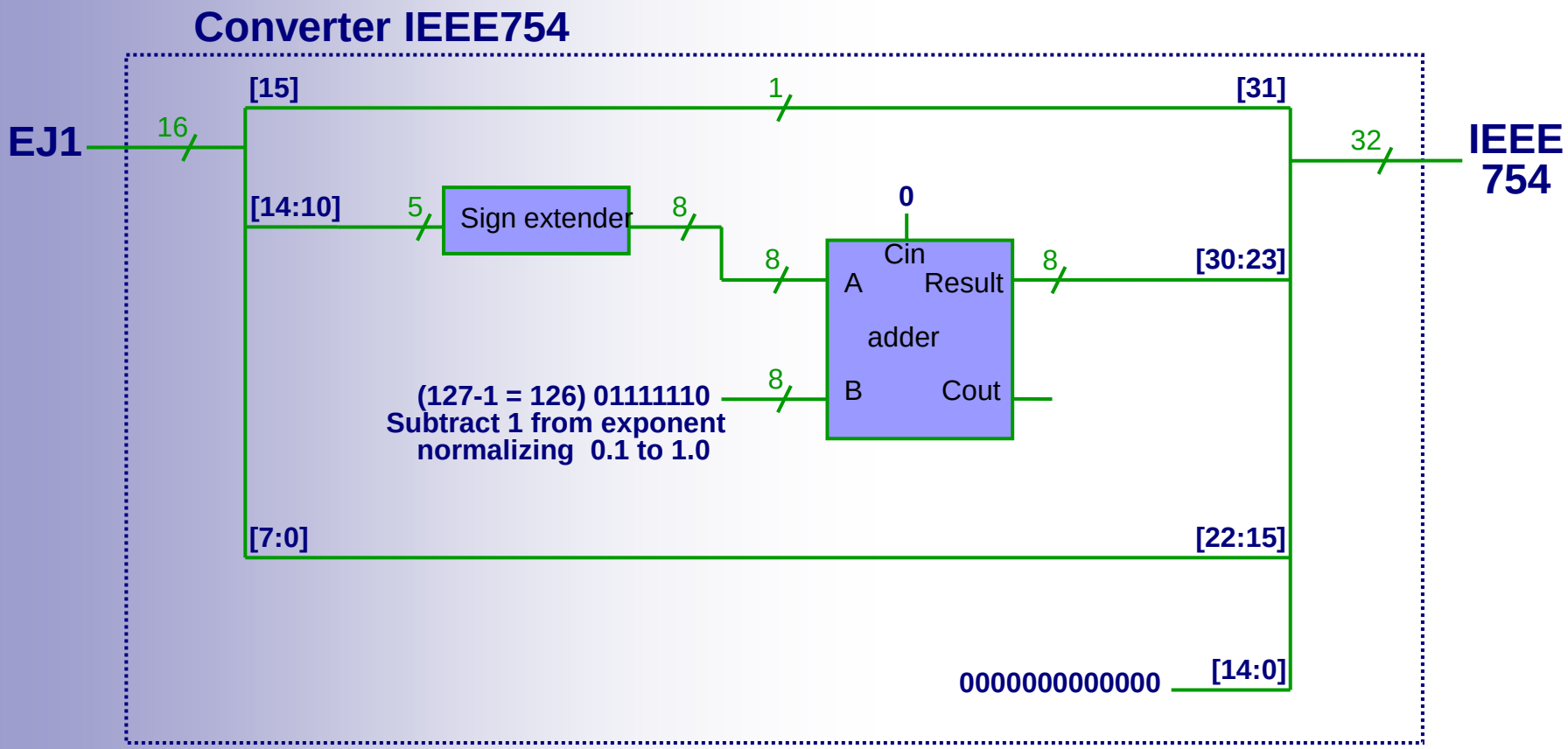
- format floating point Ej1:
  - 16 bits size: 5 bits characteristic, 11 bits mantissa
  - characteristic: exponent (E) 2C represented
  - normalized mantissa in Sign Magnitude
    - Normalization: 0.5 ≤ mantissa < 1
      - Adjust exponent up to the mantissa has the shape 0.1… (most significant 1 right of the point)
  - value = $(-1)^S \times M \times 2^E$ (M=01xxxxxxx)



- Design hardware to convert an EJ1 real number into an IEEE754 equivalent

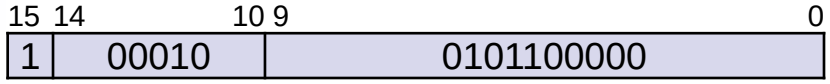


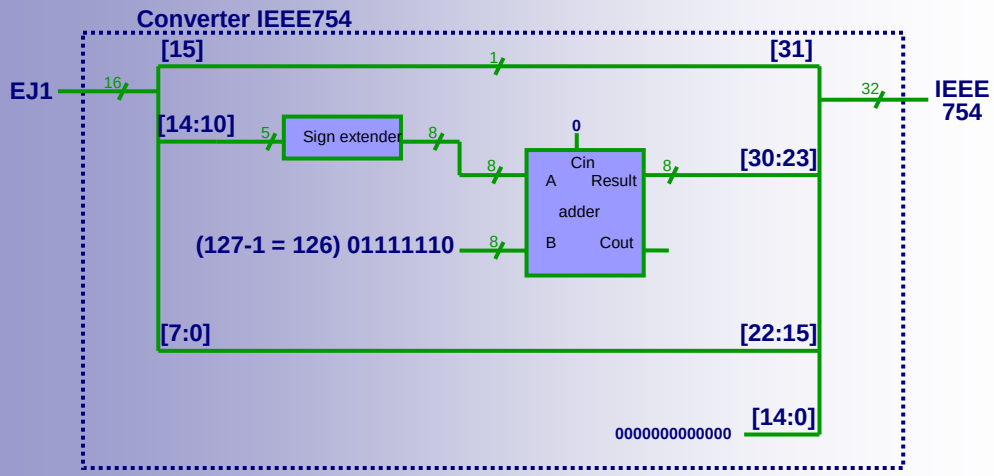
exercise floating point



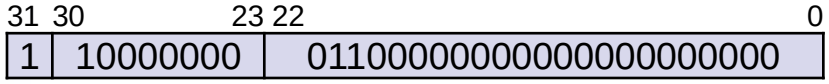
exercise floating point

- example of conversion: -2.75
- format floating point EJ1:
  - -2.75 = -10.11 = $-0.1011x2^2$
    (Make binary; Normalize (0.1x))
  - S=1, C=2=00010, M=0101100000





- IEEE[31] = EJ[15] = 1
- IEEE[30:23] = 00000010+01111110 = 10000000
- IEEE[22:15] = 01100000
- IEEE[14:0] = 00000000000000



- value IEEE → $V=(-1)^s x1.M x2^{C-127} = (-1)^1 x1.011x2^{128-127} = -1.011x2$ = -10.11 = -2.75

Topic 4

- introduction
  - representing information
- characters and strings
  - representing character strings
- numerical information
  - representing integers
    - positional representation: unsigned, signed magnitude, two's complement.
    - nonpositional representation: excess and BCD.
  - representing real numbers
    - fixed point
    - floating point: IEEE 754
- adding redundancy: detection and correction of errors
  - parity code
  - Hamming code SEC, SEC/DED

Detection and correction of errors

- adding redundancy:
  - Safeguard information for possible errors
  - Adding information to data (redundancy)
    - More bits than necessary to represent the information
  - data + redundancy follow rules
    - On an error the rule is not followed → error can be detected
    - sufficient redundant information to detect where is the error, such that it can be repaired

Hamming distance

- Distance between two binary strings
- Hamming distance between A and B = number of different bit values in A and B

Alternatively: the number of bits that have to change to walk from A to B.

- Example: H(100010, 010010) = 2

code of minimum distance M

- codes at minimum distance M:
  - the Hamming distance between each pair of strings in the code is at least M
- Example
  - H(A,B) = 4
  - H(A,C) = 2
  - H(A,D) = 2
  - H(B,C) = 2
  - H(B,D) = 2
  - H(C,D) = 4

A	0101
B	1010
C	0000
D	1111

→ code of distance 2

code of minimum distance M

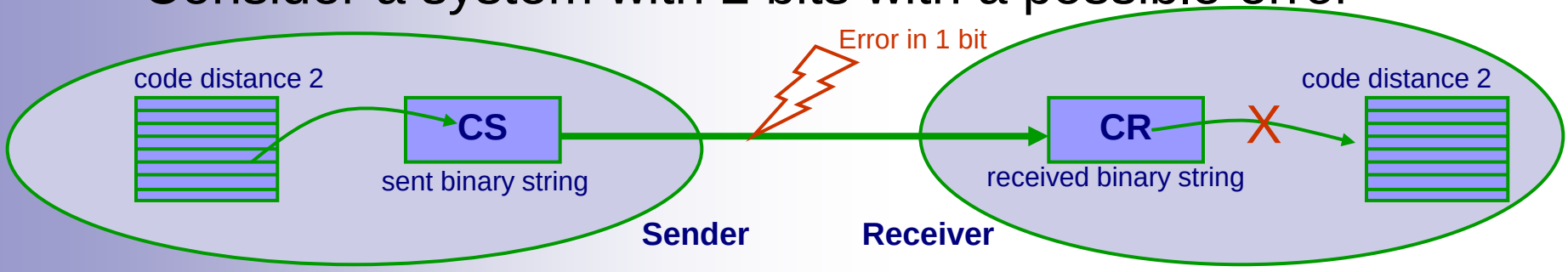
- Let a code have distance M
  - A code of minimum distance M can detect D wrong bits and correct C bits of them if:
    - $M \geq C + D + 1$
    - $D \geq C$
- Example

M	C	D	
1	-	-	No detection, no correction
2	0	1	can detect an error in 1 bit, but not correct
3	1	1	can detect and correct an error in 1 bit
3	0	2	Can detect an error in 2 bits, but correct none

codes of minimum distance M

- How to detect an error in a bit when M=2?

  Consider a system with 2 bits with a possible error



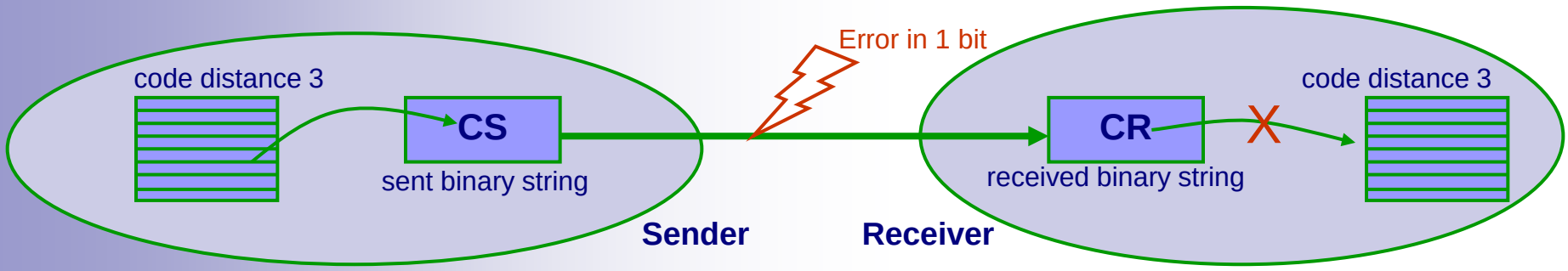
  On 1 bit error → H(CS,CR) = 1

  - In code M=2, H(CS, Ci) ≥ 2 → CR cannot belong to that, as it differs only 1 bit with CS
  - Receiver sees CR does not belong to the code and detects error

codes of distance M

- How can an M=3 code detect and correct a bit error?

In case there can only be an error in one bit



- Like in case M=2, a 1 bit error shows CR does not belong to the code → detection
- With M=3, for any string Ci of the code
  - H(CS, Ci) ≥ 3
  - H(CR, Ci ≠ CS) ≥ 2
  - H(CR, CS) = 1 → correction
- Error can be corrected. The only string of the code at distance 1 to the received (CR) is the sent string (CS), the others have a distance of at least 2.

Parity codes

adding redundancy

parity code

- How to create distance 2 codes easily?
- Add a parity bit to the original information→ parity code
  - Add a bit (redundancy) such that the string has an odd or even number of zeros/ones
    - Even parity bit → make the number of 1's even
    - Odd parity bit → make the number of 1's odd
    - Zero even parity bit → make the number of 0's even
    - Zero odd parity bit → make the number of 0's odd
  - Default: even parity bit

parity code

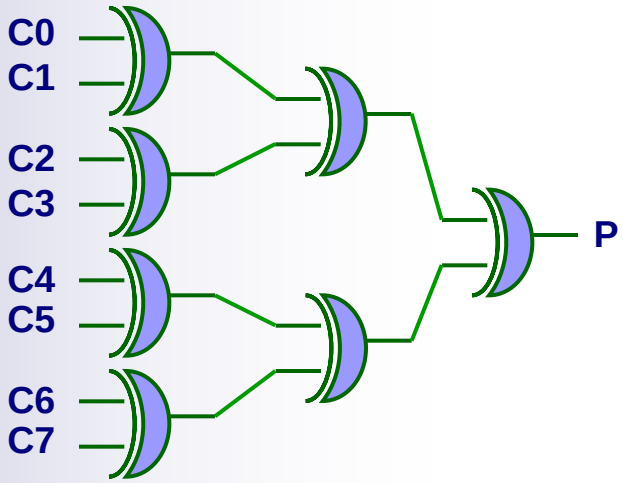
- Why does a parity code have distance M=2?
  - Let the original code have M=1
  - Let A and B be two strings of the code at distance 1, i.e. H(A,B) = 1, so they differ in only 1 bit
    - What is the distance after adding a parity bit? Consider
      - A has even number of 1's → B has odd number of 1's
        - PA = 0, PB = 1 → parity bit has different value
      - A has odd number of 1's → B has even number of 1's
        - PA = 1, PB = 0 → parity bit has different value
      - In Parity code the strings differ more → H(ParityA, ParityB) = 2
  - strings with H(A,B) = 2 → PA = PB

  → H(ParityA, ParityB) = 2

parity code

- example of generating a parity bit for string C of 8 bits
  - Generate string D of 9 bits, with the 8 of C and parity bit P, D=[C P]
    - P= 0 if the number of 1's in C is even
    - P= 1 if the number of 1's in C es odd
  - P = C0 xor C1 xor … xor C7

C	D
10111001	101110011
00000000	000000000
10000000	100000001



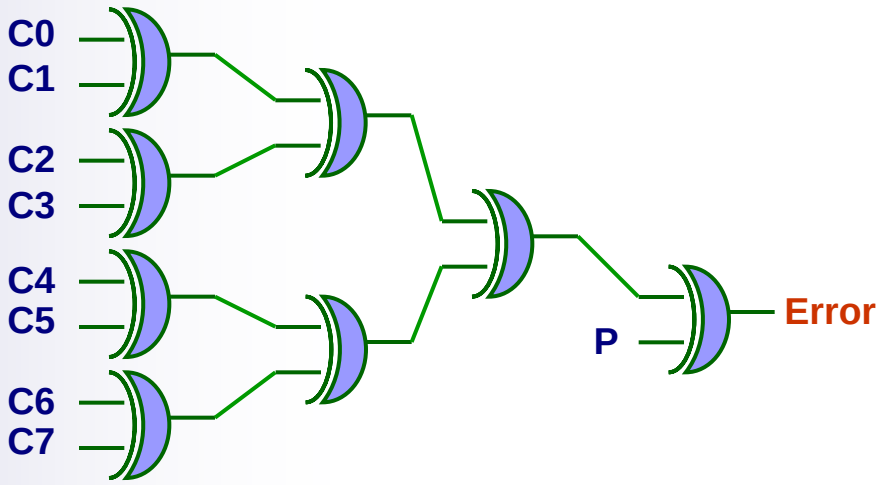
parity code

- example of error detection of error in C of 8 bits and 1 parity bit P

In case there can only be an error in one bit

- Count number of 1's in the received string [C P]:
  - if the number of 1's is even → no error
  - if the number of 1's is odd → Error
- Error = P xor C0 xor C1 xor … xor C7

[C P]	Error?
101110010	Si
000000001	Si
100000001	No



Hamming code

Single Error Correction (SEC)

Double Error Detection (DED)

adding redundancy

SEC Hamming code

- Consider a code of distance M = 3
  - Detect and correct single bit errors (Single Error Correction)
    - The probability of double errors is neglected
- Add (odd) partial parity bits
  - In this way we can check parity in parts of the string

SEC Hamming code

- SEC codes
  - Given a string of *n* bits. Add *p* parity bits as follows:
    - $2^p \geq n + p + 1$
  - Let R be the resulting string of *n*+*p* bits
    - The parity bits P_i are placed on positions *i*=1, 2, 4, … so a power of 2 in combined string R
    - The other bits R_j are taken from the given *n* string as data D_j
      - R looks like …$D_{10}D_9P_8D_7D_6D_5P_4D_3P_2P_1$
    - Parity bit P_i protects positions of data D_j such that:
      - Position *j* reflects in a bit *i*

SEC Hamming code

- example SEC code for 4 bits of data.
  - 3 parity bits are needed: $2^3 = 8 \geq 4 + 3 + 1 = 8$
  - SEC code: R=[$D_7D_6D_5P_4D_3P_2P_1$]
    - (odd) parity bits at positions 1,2,4, i.e. power of 2
      - $P_1 = D_3$ xor D_5 xor D_7
      - $P_2 = D_3$ xor D_6 xor D_7
      - $P_4 = D_5$ xor D_6 xor D_7
    - Parity check
      - $C_1 = P_1$ xor D_3 xor D_5 xor D_7
      - $C_2 = P_2$ xor D_3 xor D_6 xor D_7
      - $C_4 = P_4$ xor D_5 xor D_6 xor D_7
    - Error detection: error on $C_i = 1$ for one of the *i*
    - Error correction: $4C_4+2C_2+C_1$ provides the error bit position
    - In general $\sum_{i=1}^{p} 2^i C_i$

	position j
P_1	001
P_2	010
D_3	**011**
P_4	100
D_5	**101**
D_6	**110**
D_7	**111**

SEC/DED Hamming code

- Consider a code of distance M = 4
  - Detect and correct single bit errors and also detect double bit errors

  Single Error Correction / Double Error Detection)
    - So a double error may occur
- Add 1 (odd) parity bit to the SEC code
- example SEC/DED code for 4 bits of data.
  - SEC/DED code has the shape: $D_7D_6D_5P_4D_3P_2P_1P$

SEC/DED Hamming code

- SEC/DED code: $D_7D_6D_5P_4D_3P_2P_1P$
  - Let the value P and P_i be given by
    - $P_1 = D_3$ xor D_5 xor D_7
    - $P_2 = D_3$ xor D_6 xor D_7
    - $P_4 = D_5$ xor D_6 xor D_7
    - $P = D_7$ xor D_6 xor D_5 xor P_4 xor D_3 xor P_2 xor P_1
  - Error check
    - $C_1 = P_1$ xor D_3 xor D_5 xor D_7
    - $C_2 = P_2$ xor D_3 xor D_6 xor D_7
    - $C_4 = P_4$ xor D_5 xor D_6 xor D_7
    - $C = P$ xor D_7 xor D_6 xor D_5 xor P_4 xor D_3 xor P_2 xor P_1
  - cases
    - $C = C_i = 0$ → No error
    - $C = 1$, $C_i = 1$ for an i → single bit error in position $\sum_{i=1}^{p} 2^i C_i$
    - $C = 0$, $C_i = 1$ for an i → double bit error, no correction possible
    - $C = 1$, $C_i = 0$ → error in parity bit P itself

detection/correction codes: examples

- Generate binary strings of the given data in the codes

$D_7D_6D_5D_3$	Even parity 1's $D_7D_6D_5D_3P$	Odd parity 0's $D_7D_6D_5D_3P$	SEC Hamming $D_7D_6D_5P_4D_3P_2P_1$	SEC/DED $D_7D_6D_5P_4D_3P_2P_1P$
0110				
1101				

- For the following strings, check if there has been an error and if possible provide the corrected data

	Even parity 1's $D_7D_6D_5D_3P$	Odd parity 0's $D_7D_6D_5D_3P$	SEC Hamming $D_7D_6D_5P_4D_3P_2P_1$	SEC/DED $D_7D_6D_5P_4D_3P_2P_1$/P
0110010/0				
1101100/0				

Bibliografía

- P.M. ANASAGASTI (1998). Fundamentos of the computeres. 8ª edition. Madrid. Ed. Thomson
  - Capítulo 2.
- G. BANDERA, F.J. CORBERA and OTROS (2000). Tecnología of computeres. 1ª edition. Málaga. Servicio of publicaciones UMA.
  - Capítulo 2.
- ERCEGOVAC and LANG (2003). Digital Arithmetic. 1ª edition. Morgan Kaufmann.
  - Capítulo 8.