电子科技大学 University of Electronic Science and Technology of China

University of Glasgow

UESTC2004: Embedded Processors

Semester 2 – 2020/2021

Data Representations – *Part 2*

Lecture 8

Dr. Lina Mohjazi (Lina.Mohjazi@glasgow.ac.uk)
Lecturer
Glasgow College, UESTC
James Watt School of Engineering

➢Objectives - To understand:

- ❖ Unsigned binary number systems
- ❖ Textual information stored as ASCII
- ❖ Floating point representations
- ❖ Encoding schemes

University of Glasgow

Representing Numbers - Integer Representation

Signed Integers

2's Complement Representation

- Again, MSB is the sign bit: **0** represents a positive integer and **1** represents a negative integer.

- The remaining bits represents the magnitude of the integer, as follows:

  - For positive integers, the absolute value of the integer is the magnitude of the remaining bits.
    Similar to the Sign-Magnitude representation!

  - For negative integers, the absolute value of the integer is the magnitude of the ***complement*** of the remaining bits plus one (**hence called 2's complement**).

97 =	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
~97 =	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0
															+	1
−97 =	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1

Invert

Add 1; if the addition causes a carry bit past the most significant bit, discard the high carry

Representing Numbers - Integer Representation

Signed Integers

2's Complement Representation

2s COMPLEMENT - CONVERSIONS

- Converting positive numbers to 2s complement:
  - Same as converting to binary
- Converting negative numbers to 2s complement:

Decimal to 2s Complement

- 4 (decimal)

↓ Convert decimal to binary

0 1 0 0

↓ 1s complement

1 0 1 1

↓ Add 1

- 4 = 1 1 0 0 (2s Complement)

2s Complement to Binary

1 1 0 0 (2s C)

↓ 1s complement

0 0 1 1

↓ Add 1

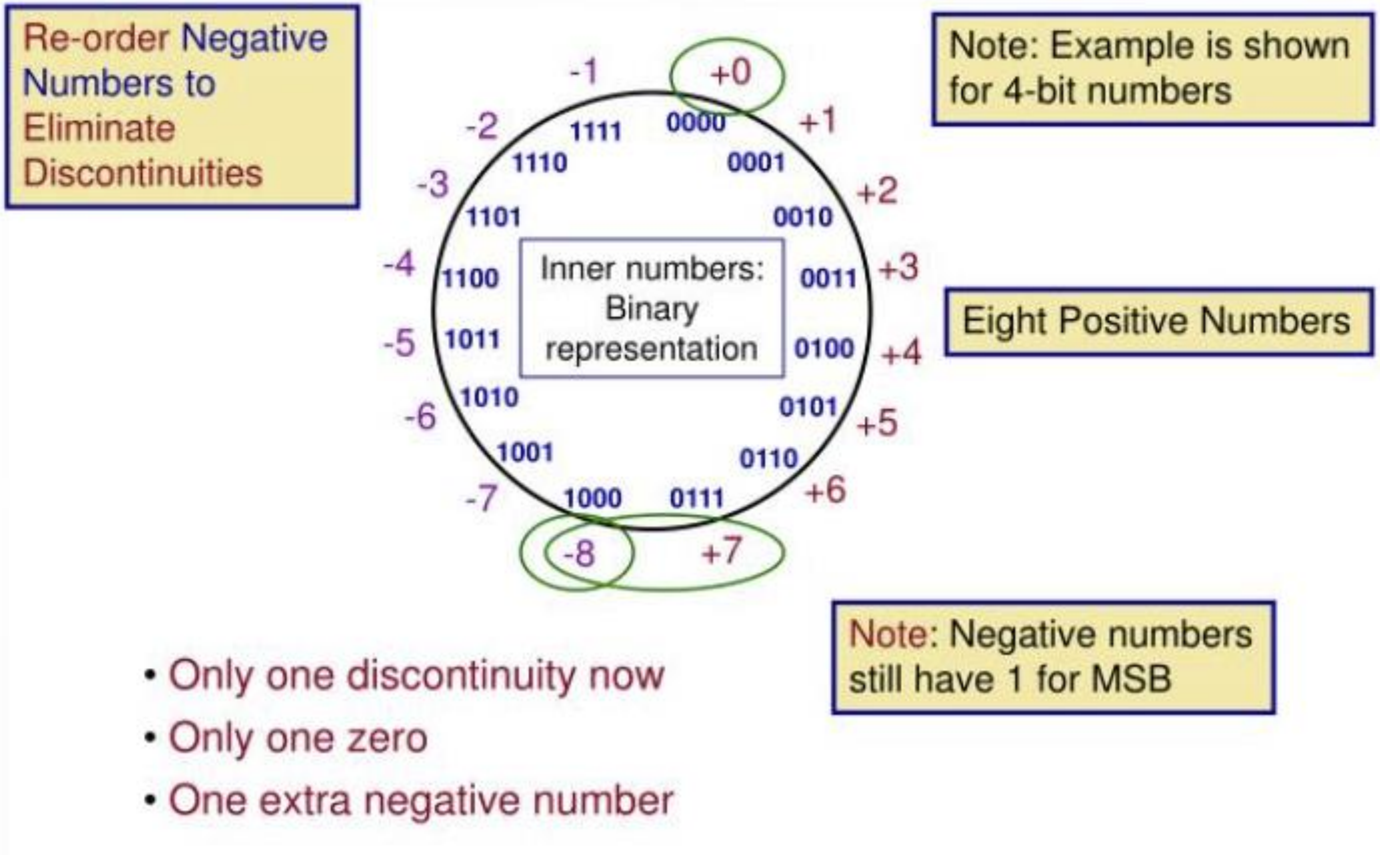
0 1 0 0 (Binary)

Representing Numbers - Integer Representation

Signed Integers

2's Complement Representation

If $n=4$

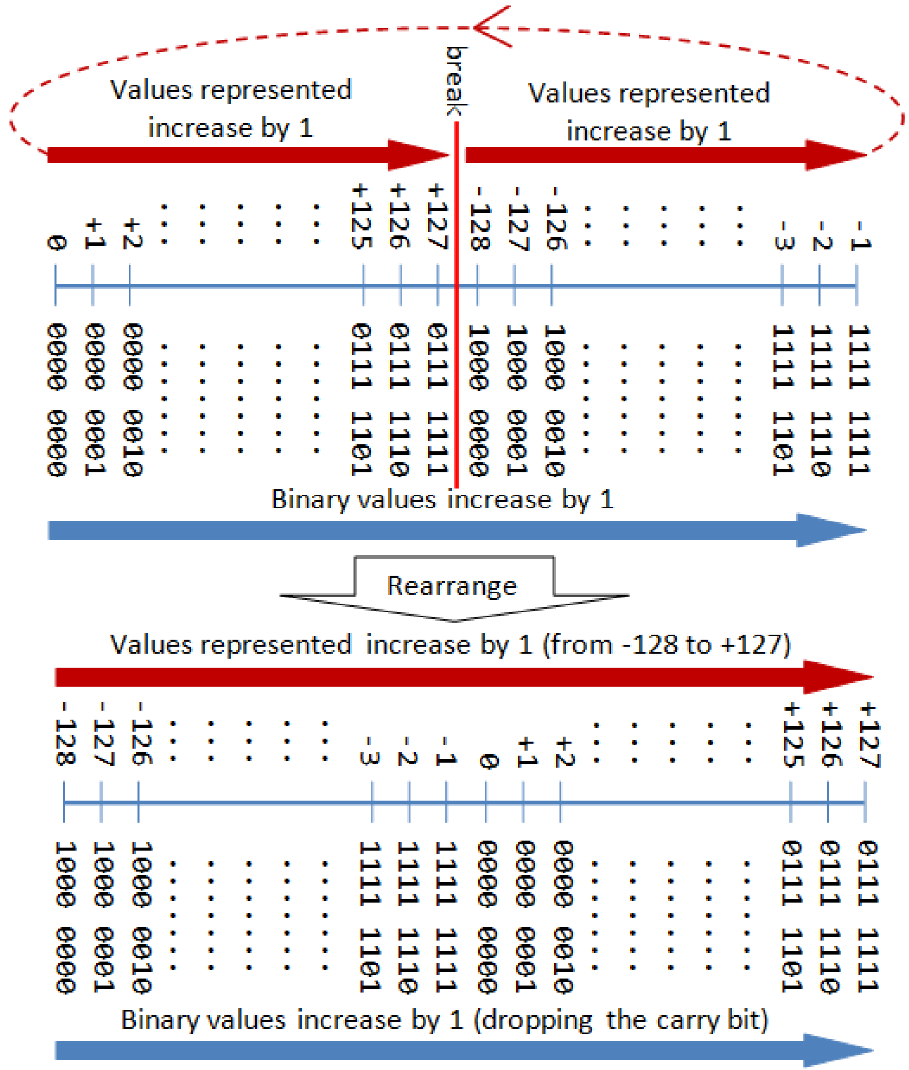


Representing Numbers - Integer Representation

Signed Integers

2's Complement Representation

If ***n*=8**



2's Complement Representation

Representing Numbers - Integer Representation
Signed Integers
2's Complement Representation - Arithmetic

- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.
  - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19.

```
1   1 1
    0 0 1 1 0 0 0 0
  + 1 1 1 0 1 1 0 1
  -----------------
    0 0 0 1 1 1 0 1
```

We note that 19 in one's complement is: 00010011,
so -19 in one's complement is: 11101100,
and -19 in two's complement is: 11101101.

Representing Numbers - Integer Representation

Signed Integers

2's Complement Representation - Arithmetic

- While we can't always prevent ***overflow***, we can always ***detect*** overflow.
- In complement arithmetic, an overflow condition is easy to detect.
- Example:
  - Using two's complement binary arithmetic, find the sum of 107 and 46.
- We see that the nonzero carry from the seventh bit ***overflows*** into the sign bit, giving us the erroneous result: 107 + 46 = -103.

```
 1 1   1 1 1
   0 1 1 0 1 0 1 1
 + 0 0 1 0 1 1 1 0
 -----------------
   1 0 0 1 1 0 0 1
```

Rule for detecting two's complement overflow: When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred.

University of Glasgow

Representing Numbers - Integer Representation

Signed Integers

2's Complement Representation - Arithmetic

Example:

Find the sum of 23 and -9.

```
1 ← 1 1 1   1 1 1
    0 0 0 1 0 1 1 1
  + 1 1 1 1 0 1 1 1
  -----------------
    0 0 0 0 1 1 1 0
```

We see that there is carry into the sign bit, but the final result is correct: 23 + (-9) = 14.

This is because there is also carry out of the sign bit!

Rule for detecting signed two's complement overflow: **When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred. If the carry into the sign bit equals the carry out of the sign bit, no overflow has occurred.**

Carry and Overflow

- Carry is important when …
  - Adding or subtracting unsigned integers
  - Indicates that the unsigned sum is out of range
  - Either < 0 or > maximum unsigned *n*-bit value
- Overflow is important when …
  - Adding or subtracting signed integers
  - Indicates that the signed sum is out of range
- Overflow occurs when
  - Adding two positive numbers and the sum is negative
  - Adding two negative numbers and the sum is positive
  - Can happen because of the fixed number of sum bits

Representing Numbers - Integer Representation
Signed Integers
2's Complement Representation - Addition

- It is important to note  each occur **without** the other.

- In unsigned numbers, c ow.

- In two's complement, c overflow.

- The reason for the rule ement occurs, not when a bit is carried out of the rried into it. *That is, when there is a carry into the*

- The rules detect this e he result. A negative and positive added togeth the sum is between the addends.

- Since both of the adde ge of numbers, and their sum is between them, i

Carry and Overflow Examples

- We can have carry without overflow and vice-versa
- Four cases are possible

				1					
	0	0	0	0	1	1	1	1	15
+	0	0	0	0	1	0	0	0	8
	0	0	0	1	0	1	1	1	23

Carry = 0 Overflow = 0

1	1	1	1	1					
	0	0	0	0	1	1	1	1	15
+	1	1	1	1	1	0	0	0	245 (-8)
	0	0	0	0	0	1	1	1	7

Carry = 1 Overflow = 0

	1								
	0	1	0	0	1	1	1	1	79
+	0	1	0	0	0	0	0	0	64
	1	0	0	0	1	1	1	1	143 (-113)

Carry = 0 Overflow = 1

1			1	1					
	1	1	0	1	1	0	1	0	218 (-38)
+	1	0	0	1	1	1	0	1	157 (-99)
	0	1	1	1	0	1	1	1	119

Carry = 1 Overflow = 1

Let's play a game!

Go to: https://www.menti.com

Enter the code: 2615 6548

Representing Numbers - Integer Representation

Signed Integers

2's Complement Representation - Addition

- -19 + -7 = -26:

$\boxed{1}$	1	1	1	1			1	
	1	1	1	0	1	1	0	1
+	1	1	1	1	1	0	0	1
	1	1	1	0	0	1	1	0

Carryout without overflow. Sum is correct.

- 127 + 1 = 128:

	1	1	1	1	1	1	1	
	0	1	1	1	1	1	1	1
+	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	0	0

Overflow, no carryout. Sum is not correct.

Representing Numbers - Integer Representation

Signed Integers

Advantages of 2's Complement Representation

Advantages:

1. The value zero is uniquely represented by having all bits set to zero:

0 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
~0 =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
															+	1
−0 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Advantages:

2. When performing an arithmetic operation (for example, addition, subtraction, multiplication, division) on two signed integers in 2's Complement representation, **the same method** is followed as if two unsigned integers are used, **EXCEPT**, the high carry (or the high borrow for subtraction) is thrown away.

3. The only extra capabilities needed are:

    - Flipping all the bits.
    - Throwing away the high carry (or the high borrow).

4. This reduces the amount and cost of the circuitry needed for sign-magnitude or 1s Complement.

Representing Numbers
Floating-Point Number Representation

- A floating-point number is expressed in the scientific notation, with a fraction (**F**), and an exponent (**E**) of a certain radix (**r**), in the form of **F × r^E**.

- Decimal numbers use radix of 10 (F × 10^E); while binary numbers use radix of 2 (F × 2^E).

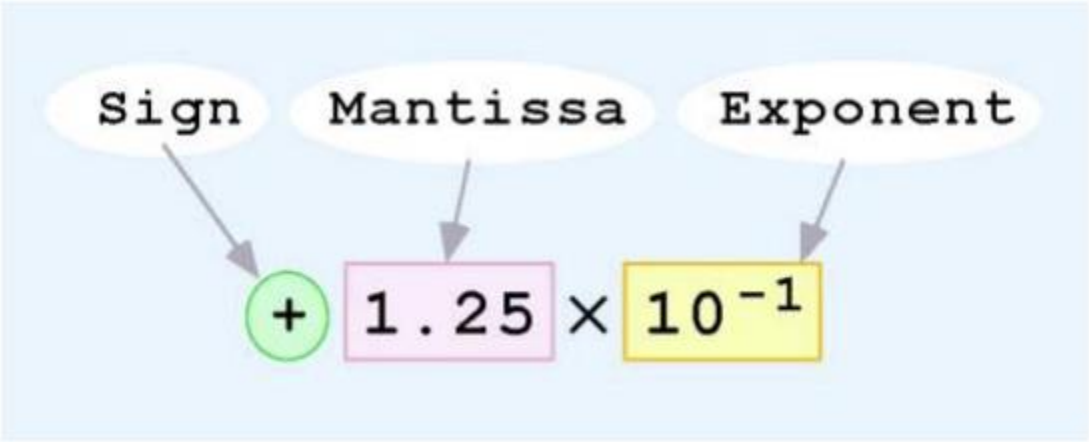
- Representation of floating-point number is not unique. For example, the number 55.66 can be represented as **5.566 × 10^1**, **0.5566 × 10^2**, **0.05566 × 10^3**.

- Floating-point numbers suffer from loss of precision when represented with a fixed number of bits (e.g., 32-bit or 64-bit).

- Floating number arithmetic is very much less efficient than integer arithmetic. It could be speed up with a so-called dedicated floating-point co-processor.

Representing Numbers
Floating-Point Number Representation

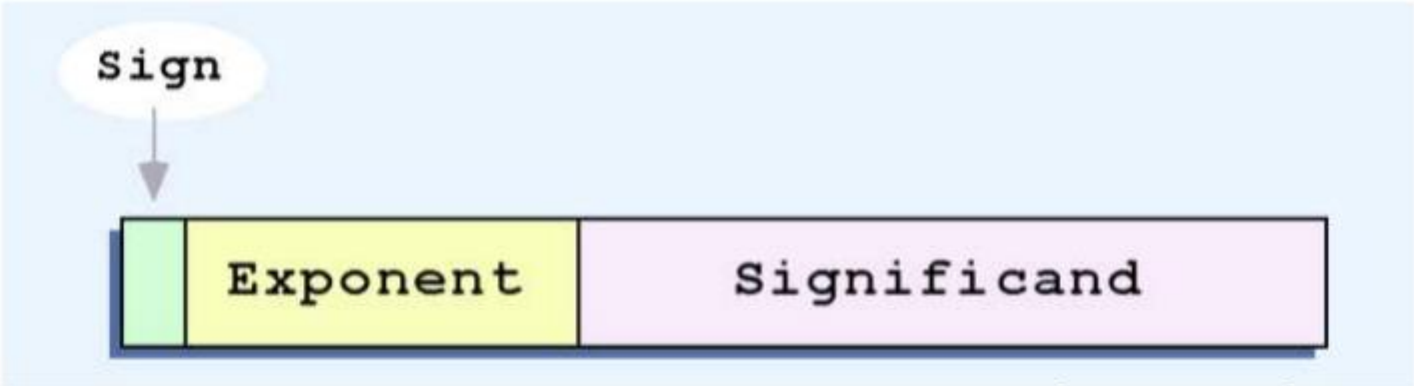
- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



Representing Numbers

Floating-Point Number Representation

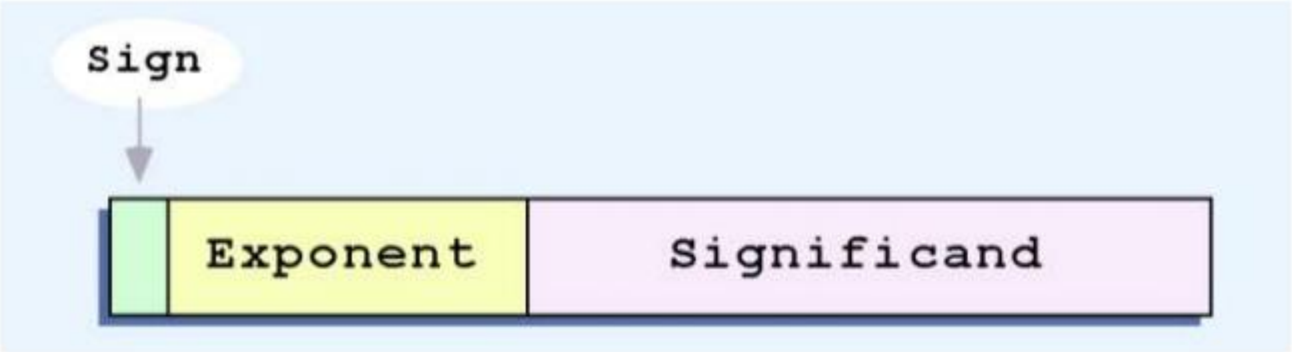
- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

Representing Numbers
Floating-Point Number Representation



- The one-bit sign field is the sign of the stored value.
- The size of the exponent field, determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

University of Glasgow

Representing Numbers
Floating-Point Number Representation

Example:

– Express 32_{10} in the simplified 14-bit floating-point model.

- We know that 32 is 2^5. So in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.

- Using this information, we put 110 ($= 6_{10}$) in the exponent field and 1 in the significand as shown.

0	0 0 1 1 0	1 0 0 0 0 0 0 0

Representing Numbers
Floating-Point Number Representation

Problem

These are *all* equivalent representations for 32:

- 0.1×2^6
- 0.01×2^7
- 0.001×2^8
- 0.0001×2^9

0	0 0 1 1 0	1 0 0 0 0 0 0 0
0	0 0 1 1 1	0 1 0 0 0 0 0 0
0	0 1 0 0 0	0 0 1 0 0 0 0 0
0	0 1 0 0 1	0 0 0 1 0 0 0 0

Not only do these synonymous representations **waste** space, but they also require **complex hardware** to test

Representing Numbers

Floating-Point Number Representation

- Modern computers adopt **IEEE 754 standard** for representing floating-point numbers. There are two representation schemes: 32-bit single-precision and 64-bit double-precision.

IEEE-754 32-bit Single-Precision Floating-Point Numbers

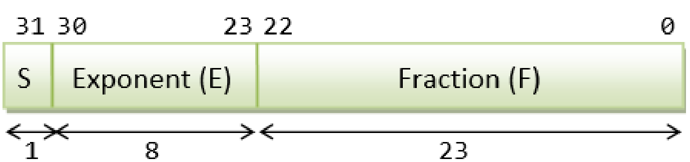
In 32-bit single-precision floating-point representation:

- The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.
- The following 8 bits represent exponent (E).
- The remaining 23 bits represents fraction (F).

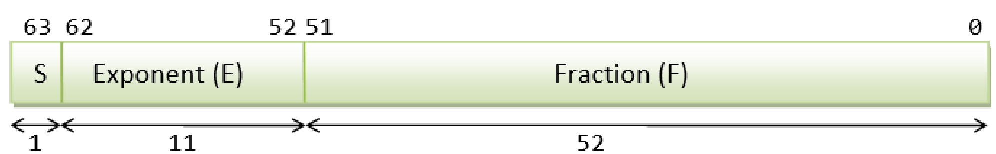
IEEE-754 64-bit Double-Precision Floating-Point Numbers

The representation scheme for 64-bit double-precision is similar to the 32-bit single-precision:

- The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.
- The following 11 bits represent exponent (E).
- The remaining 52 bits represents fraction (F).



32-bit Single-Precision Floating-point Number

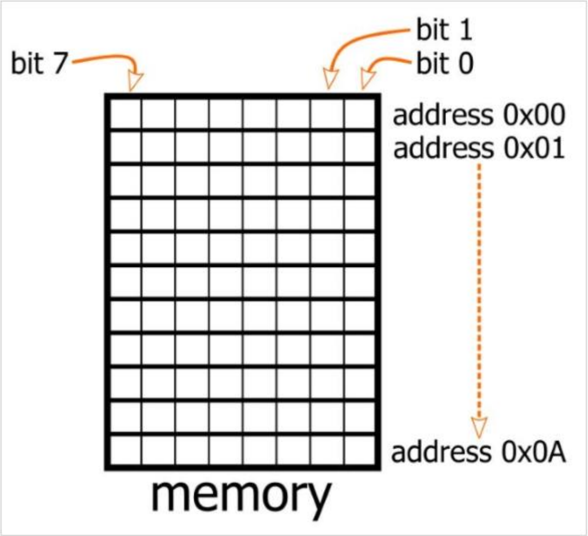


64-bit Double-Precision Floating-point Number

Understanding Byte Arrangements in Memory
Big Endian and Little Endian

Endianness in Memory:

Imagine that we're using an 8-bit microcontroller. All of the hardware in this device is designed for 8-bit data, including the memory locations. Thus, memory address 0x00 can store one byte, address 0x01 stores one byte, and so forth.



- Program a microcontroller using a C compiler that allows us to define 32-bit (i.e., 4-byte) variables
- The compiler needs to store these variables in memory
- Store them in contiguous memory locations
- MSB or LSB should be stored in the lowest memory address? (here B is for byte)
- Big-endian memory arrangement or a little-endian memory arrangement?

Understanding Byte Arrangements in Memory

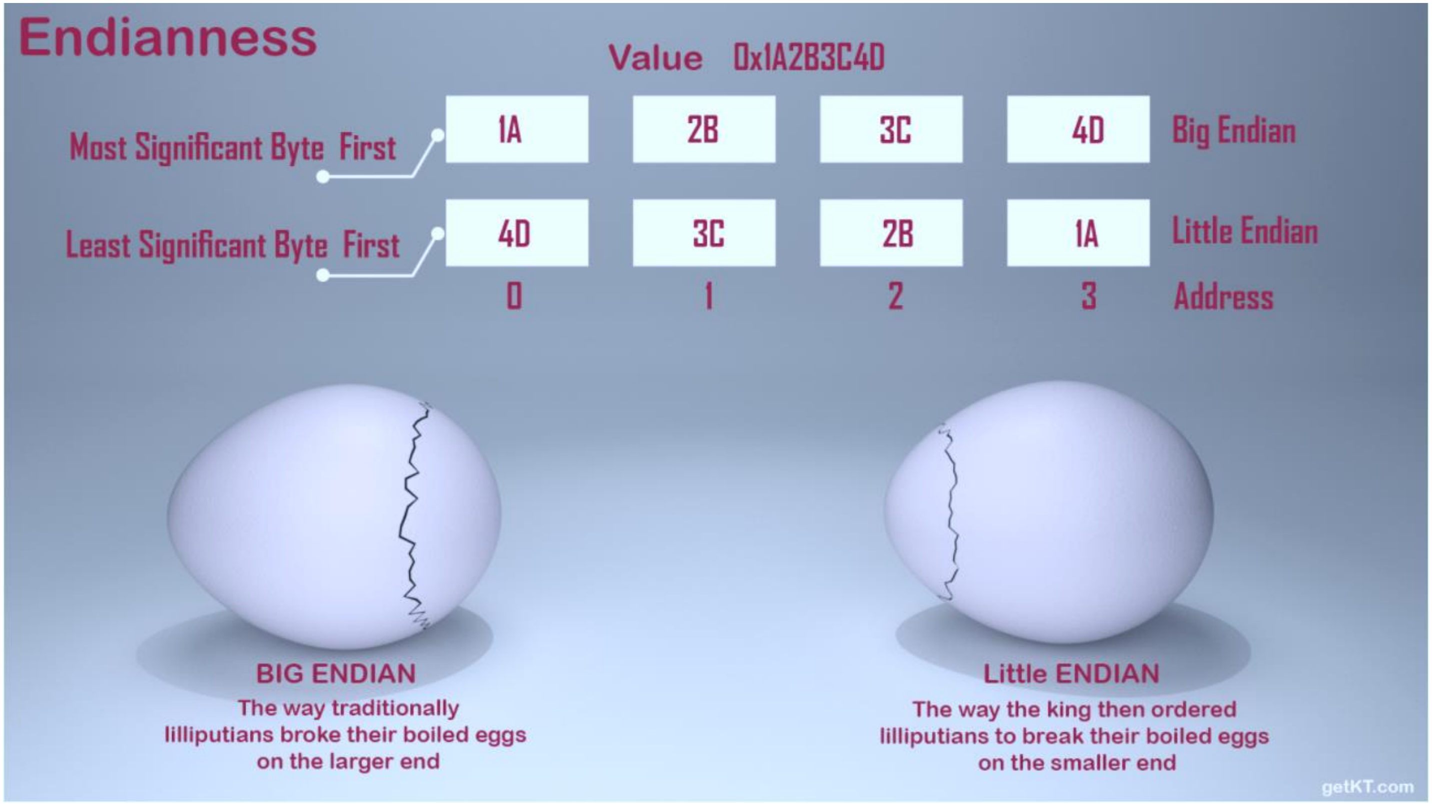
Big Endian and Little Endian

<u>Where does the term “Endian” come from?</u>

- Jonathan Swift was a satirist in the 18th century.
- His most famous book is “**Gulliver’s Travels**“, in which he talks about a civil war that broke out between those who favour breaking boiled eggs on the big end ("**big-endians**") and those who favour breaking them on the little end ("**little-endians**").
- **Endianness** is also called as Byte Order which describes order in which bytes of large values are stored in memory or transmitted over network in a transmission protocol or a stream (ex. an audio, video streams).

Understanding Byte Arrangements in Memory

Big Endian and Little Endian

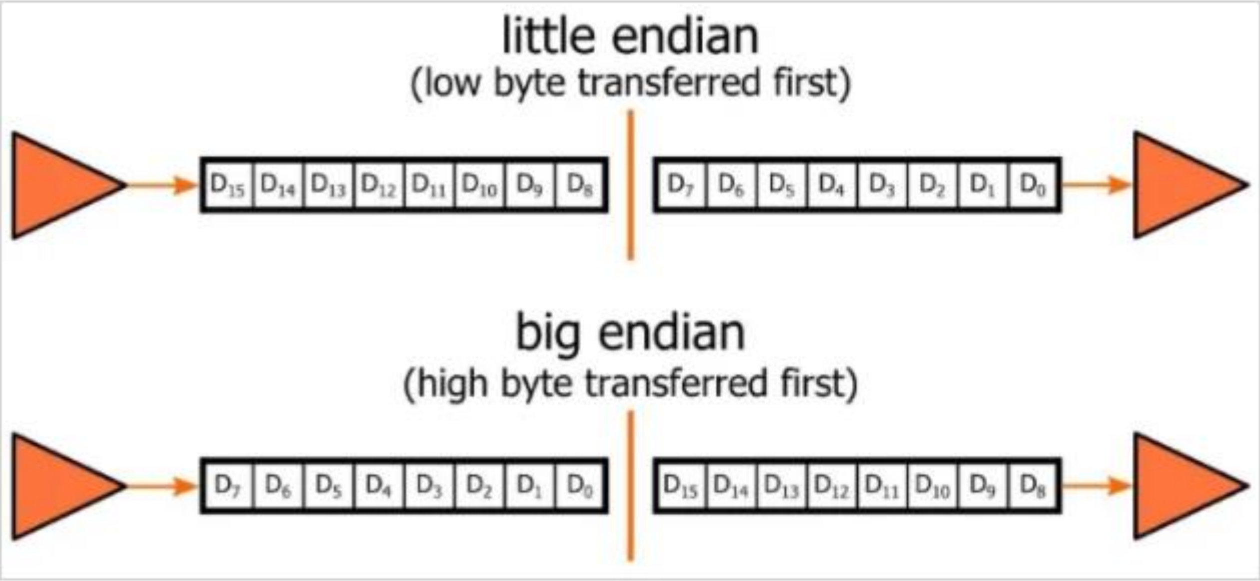


The **ARM processor** is **little endian** by default;
and can be programmed to operate as **big endian**

Understanding Byte Arrangements in Memory

Big Endian and Little Endian

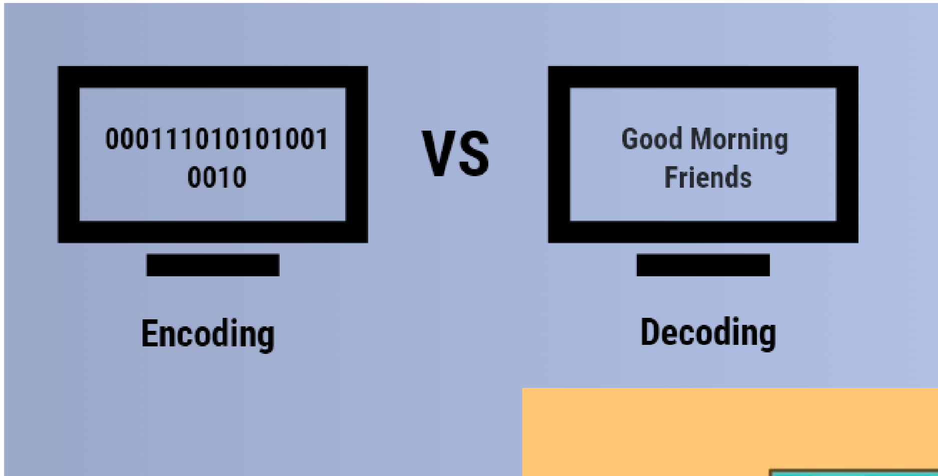
Engineers need to be aware of endianness when data is being stored, transferred, or interpreted. Serial communication is especially susceptible to endian issues, because it is inevitable that the bytes contained in a multi-byte data word will be transferred sequentially, usually either MSB to LSB or LSB to MSB.

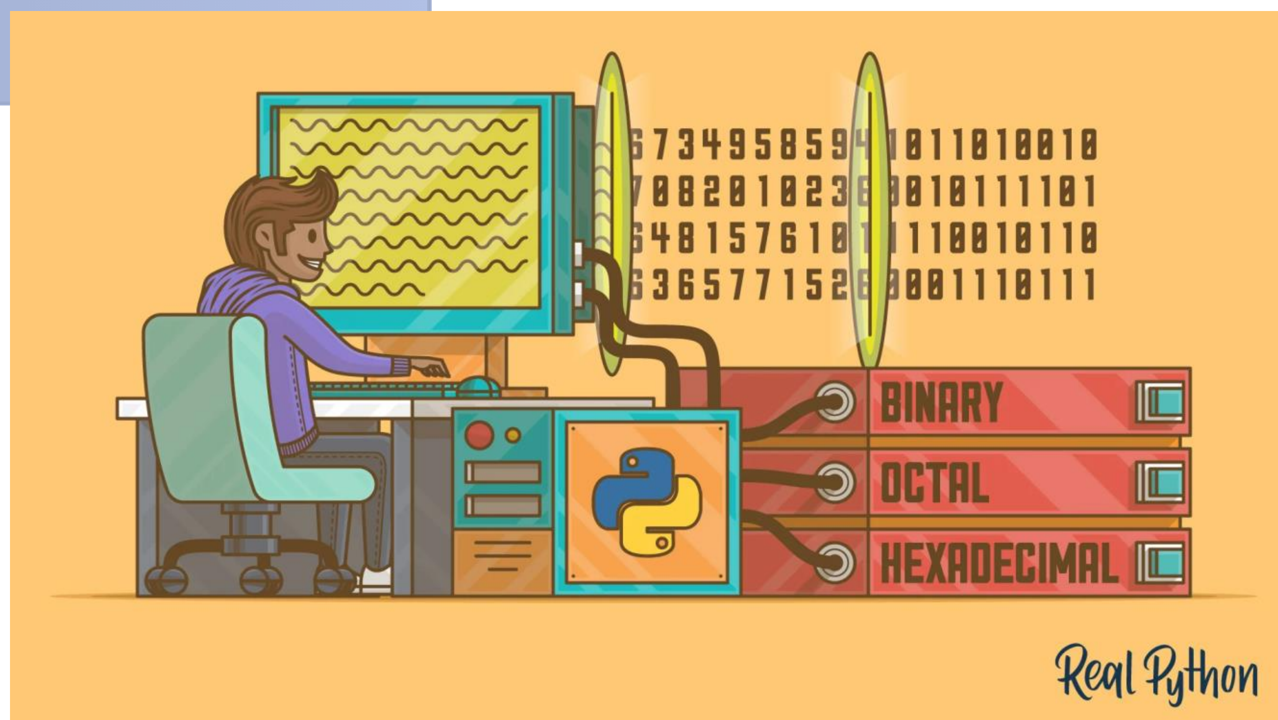


Endianness in the context of serial data transfer.

University of Glasgow

Binary Codes





What are Binary Codes?

- Representations of info (set) obtained by associating one or more codewords (a binary pattern/string) with each element in the set.

- *n*-bit binary code: a group of *n* bits that can encode up to 2^n distinct elements:

  e.g. A set of 4 distinct numbers can be represented by 2-bit codes s.t. each number in the set is assigned exactly one of the combinations/codes in {00,01,10,11}.

- To encode *m* distinct elements with an *n*-bit code: $2^n >= m$

- Note: The codeword associated with each number is obtained by coding the number, NOT converting the number to binary.

- We will see: BCD, ASCII, Unicode, Gray codes

Binary-Coded Decimal (BCD)
4-bit BCD

- BCD is employed by computer systems to **encode** the decimal number into its equivalent binary number.

- This is generally accomplished by encoding each digit of the decimal number into its equivalent binary sequence.

- The main advantage of BCD system is that in comparison to binary positional systems, it provides **a faster and more accurate representation and rounding of decimal quantities**, as well as its ease of conversion into conventional human-readable representations.

- A decimal code: Decimal numbers (0..9) are coded using 4-bit distinct binary words
- Observe that the codes 1010 to 1111 (**decimal 10 to 15**) are **NOT** represented (invalid BCD codes).

- The 4-bit BCD system is usually employed by the computer systems to represent and process **numerical data only**.

Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Binary-Coded Decimal (BCD)
4-bit BCD

To code a number with *n* decimal digits, we need *4n* bits in BCD

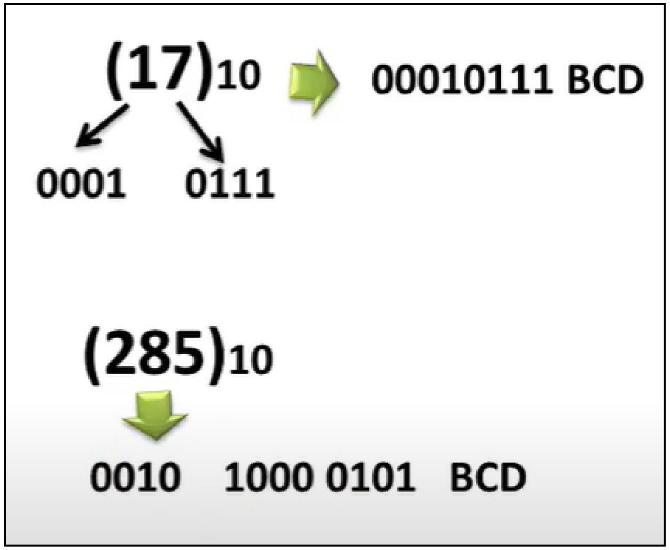
e.g. $(365)_{10} = (0011\ 0110\ 0101)_{BCD}$

This is different from converting to binary, which is

$(365)_{10} = (101101101)_2$

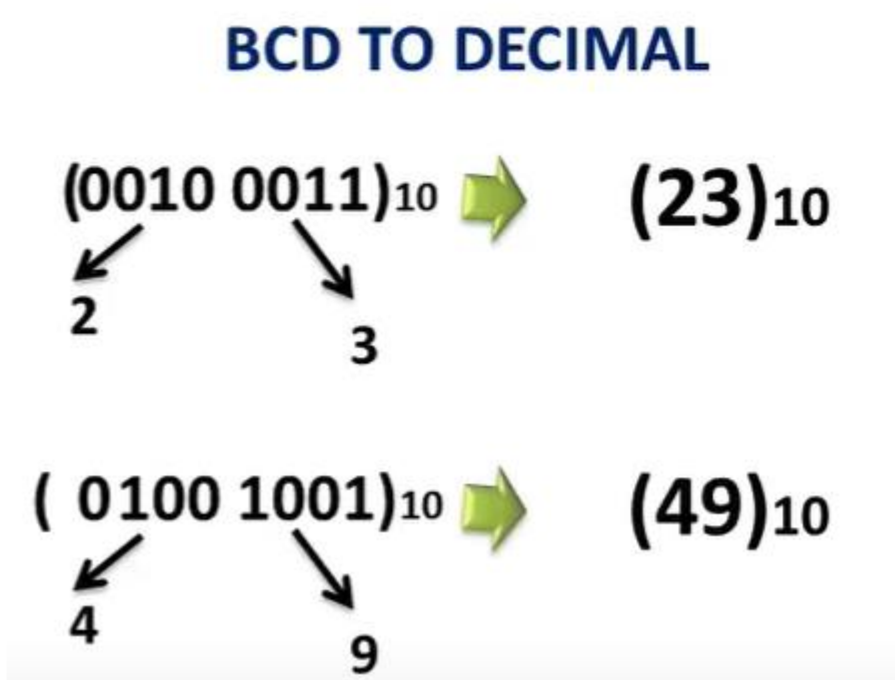
Clearly, BCD requires more bits. **BUT,** it is easier to understand/interpret

Another Example



Binary-Coded Decimal (BCD)
4-bit BCD

Convert from BCD to Decimal:



Binary-Coded Decimal (BCD)
4-bit BCD

Hex	Binary	BCD	Decimal
0	0000	0000 0000	0
1	0001	0000 0001	1
2	0010	0000 0010	2
3	0011	0000 0011	3
4	0100	0000 0100	4
5	0101	0000 0101	5
6	0110	0000 0110	6
7	0111	0000 0111	7
8	1000	0000 1000	8
9	1001	0000 1001	9
A	1010	0001 0000	10
B	1011	0001 0001	11
C	1100	0001 0010	12
D	1101	0001 0011	13
E	1110	0001 0100	14
F	1111	0001 0101	15

Binary-Coded Decimal (BCD)

BCD - Arithmetic

BCD Addition

Addition is done BCD digit by BCD digit

~ 4 bits at the time

Can we use normal binary addition?

* Problem: digits adding up to more than 9

~ Binary addition will result in invalid BCD codes

~ 1010 … 1111 are not valid

* Solution: check if resulting value is greater than 9

~ if so, add 6

~ 6 will offset the invalid BCD codes and generate the carry

Binary-Coded Decimal (BCD)

BCD - Arithmetic

BCD Addition

- BDC numbers
  - They are between 0 and 9
  - Hence each decimal number is represented by 4 bits
- Add each number between 0-9 individually

- 0 1 1 0 ← BCD for 6
  0 1 1 1 ← BCD for 7
  ____
  1 1 0 1 ← 13 but invalid!
  0 1 1 0 ← Add 6 to correct
  ____

0 0 0 1 0 0 1 1 ← BDC for 13!!

Binary-Coded Decimal (BCD)

BCD - Arithmetic

BCD Addition

- BDC numbers
  - They are between 0 and 9
  - Hence each decimal number is represented by 4 bits
- Add each number between 0-9 individually

- 0 1 1 0 ← BCD for 6
  0 1 1 1 ← BCD for 7

  ____

  1 1 0 1 ← 13 but invalid!
  0 1 1 0 ← Add 6 to correct

____

0 0 0 1 0 0 1 1 ← BDC for 13!!

Binary-Coded Decimal (BCD)

BCD - Arithmetic

BCD Addition – Another Example

- BDC numbers
  - They are between 0 and 9
  - Hence each decimal number is represented by 4 bits
- Add each number between 0-9 individually

```
     1
0 1 0 1   1 0 0 1 ← BCD for 59
0 0 1 1   1 0 0 0 ← BCD for 38
---------------------
1 0 0 1   0 0 0 1 ← 91 but invalid!
          0 1 1 0 ← Add 6
-------------------------
1 0 0 1   0 1 1 1 ← BDC for 97!!
```

Binary-Coded Decimal (BCD)

BCD - Arithmetic

BCD Addition

Example: 184 + 576

BCD carry		1	1		
		0001	1000	0100	184
	+	0101	0111	0110	+576
Binary sum					
Add 6					
BCD sum					

University of Glasgow

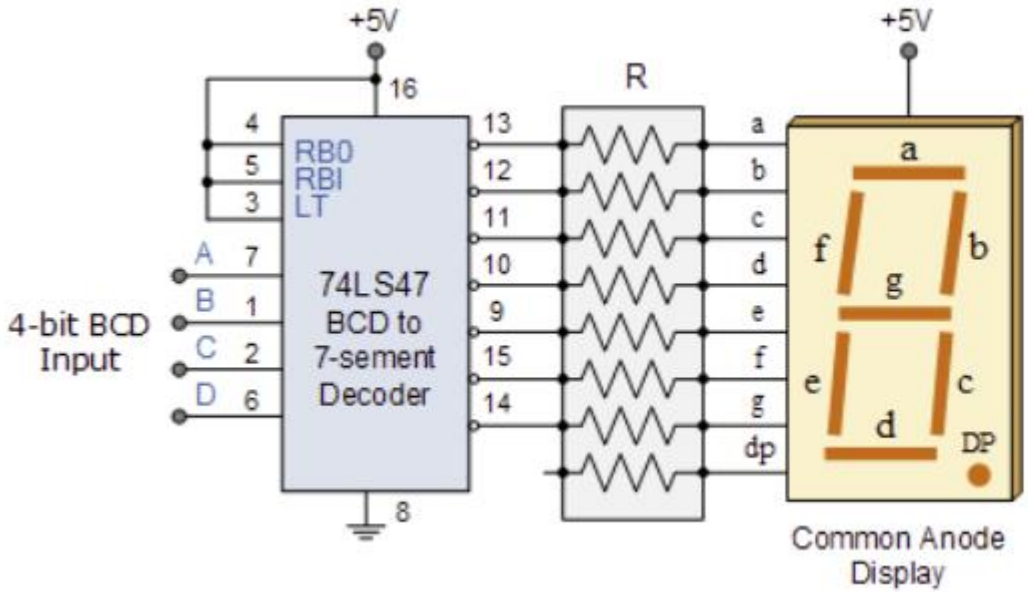
Let’s play a game again!

Go to: https://www.menti.com

Enter the code: 2615 6548

Binary-Coded Decimal (BCD)
4-bit BCD

Can you think of a popular application for BCD encoding? **Digital Displays!**



Representing Text

8-bit and 16-bit Systems

- We also need to represent letters and other symbols → *alphanumeric codes*

- The **6-bit BCD** systems can handle numeric as well as non-numeric data but *with few special characters*.

- The 8-bit and 16-bit coding systems were developed to overcome the limitations of 6-bit BCD systems, **which can handle numeric as well as nonnumeric data** with almost all the special characters such as +, -, *, /, @, $, etc.

- The most popular codes are:
  - ✓ Extended Binary Coded Decimal Interchange Code (EBCDIC)
  - ✓ American Standard Code for Information Interchange (ASCII)
  - ✓ ASCII Parity Bit
  - ✓ Unicode
  - ✓ Gray Code

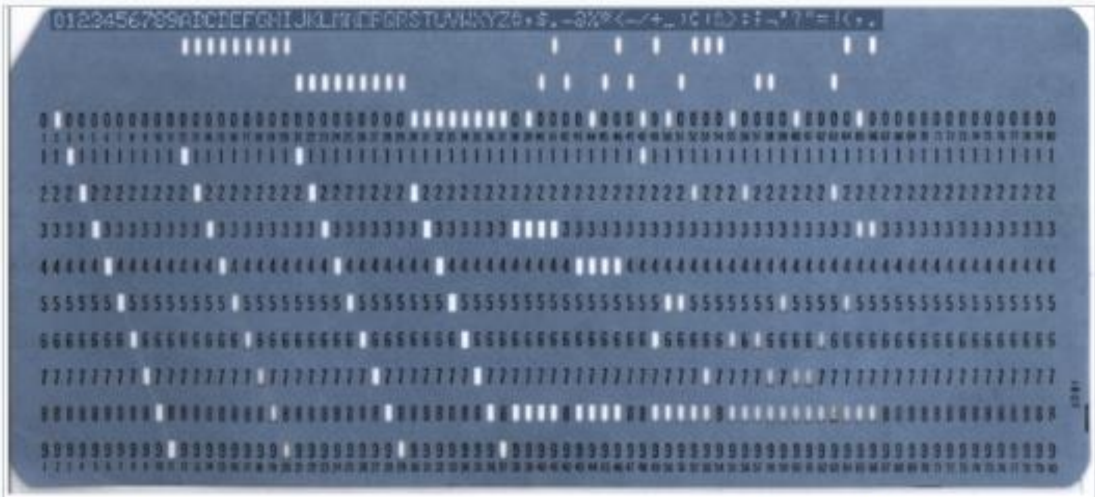
Representing Text

EBCDIC code

- The EBCDIC code is an **8-bit** alphanumeric code that was developed by IBM to represent alphabets, decimal digits and special characters, including control characters.

- The EBCDIC codes are generally the decimal and the hexadecimal representation of different characters.

- This code is rarely used by non-IBM-compatible computer systems.



Punched card with the Hollerith encoding of the 1964 EBCDIC character set. Contrast at the top is enhanced to show the printed characters.

Representing Text
ASCII Code

- The ASCII code is pronounced as ASKEE and is used for the same purpose for which the EBCDIC code is used. *However, this code is more popular than EBCDIC code as unlike the EBCDIC code this code can be implemented by most of the non-IBM computer systems.*

- Initially, this code was developed as a 7-bit BCD code to handle 128 characters but later it was modified to an 8-bit code, called the ***ASCII Parity Bit Code.***

Representing Text

ASCII Code

- ASCII code contains 128 characters:
  - ✓ 94 printable (26 upper case and 26 lower case letters, 10 digits, 32 special symbols)
  - ✓ 34 non-printable (for control functions)

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Representing Text
ASCII Parity Bit Code

- Parity coding is used to detect errors in data communication and processing.
- An 8th bit is added to the 7-bit ASCII code.
- Even (Odd) parity: set the parity bit so as to make the # of 1's in the 8-bit code even (odd).
- For example:
  - ✓ Make the 7-bit code 1011011 an 8-bit even parity code → 11011011
  - ✓ Make the 7-bit code 1011011 an 8-bit odd parity code → 01011011
- Both even and odd parity codes can detect an odd number of error. An even number of errors goes undetected.

Representing Text
Gray Code

- Gray code is another important code that is also used to convert the decimal number into **8-bit binary sequence**. However, this conversion is carried in a manner that the contiguous digits of the decimal number differ from each other by one bit only.

- If we go from one decimal number to next, **only one bit** of the gray code changes. Because of this feature, an amount of **switching is minimized** and the reliability of the switching systems is improved.

Decimal Number	4 bit Binary Number ABCD	4 bit Gray Code $G_1G_2G_3G_4$
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1
10	1 0 1 0	1 1 1 1
11	1 0 1 1	1 1 1 0
12	1 1 0 0	1 0 1 0
13	1 1 0 1	1 0 1 1
14	1 1 1 0	1 0 0 1
15	1 1 1 1	1 0 0 0

Representing Text
Unicode

- Established standard (16-bit alphanumeric code) for international character sets.
- Unlike ASCII, which uses 7 or 8 bits for each character, Unicode uses 16 bits, which means that it can represent more than 65,000 unique characters.
- Represented by 4 Hex digits and the ASCII lies in between $(0000)_{16}$ .. $(007B)_{16}$
- The standard is maintained by the **Unicode Consortium**, and as of March 2020, there is a total of 143,859 characters, with Unicode 13.0 (these characters consist of 143,696 graphic characters and 163 format characters) covering 154 modern and historic scripts, as well as multiple symbol sets and emoji.

Representing Text

Unicode

Unicode supports all languages, including Asian languages like Chinese (both simplified and traditional characters), Japanese and Korean (collectively called CJK). There are more than 20,000 CJK characters in Unicode.

Unicode Table

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
0000																																	Symbols
0020		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	Number
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	Alphabet
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~		
0080	€		‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ		Ž			‘	’	“	”	•	–	—	˜	™	š	›	œ		ž	Ÿ	
00A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿	
00C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	Latin
00E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	
0100	Ā	ā	Ă	ă	Ą	ą	Ć	ć	Ĉ	ĉ	Ċ	ċ	Č	č	Ď	ď	Đ	đ	Ē	ē	Ĕ	ĕ	Ė	ė	Ę	ę	Ě	ě	Ĝ	ĝ	Ğ	ğ	
0120	Ġ	ġ	Ģ	ģ	Ĥ	ĥ	Ħ	ħ	Ĩ	ĩ	Ī	ī	Ĭ	ĭ	Į	į	İ	ı	Ĳ	ĳ	Ĵ	ĵ	Ķ	ķ	ĸ	Ĺ	ĺ	Ļ	ļ	Ľ	ľ	Ŀ	
0140	ŀ	Ł	ł	Ń	ń	Ņ	ņ	Ň	ň	ŉ	Ŋ	ŋ	Ō	ō	Ŏ	ŏ	Ő	ő	Œ	œ	Ŕ	ŕ	Ŗ	ŗ	Ř	ř	Ś	ś	Ŝ	ŝ	Ş	ş	
0160	Š	š	Ţ	ţ	Ť	ť	Ŧ	ŧ	Ũ	ũ	Ū	ū	Ŭ	ŭ	Ů	ů	Ű	ű	Ų	ų	Ŵ	ŵ	Ŷ	ŷ	Ÿ	Ź	ź	Ż	ż	Ž	ž	ſ	
0180	ƀ	Ɓ	Ƃ	ƃ	Ƅ	ƅ	Ɔ	Ƈ	ƈ	Ɖ	Ɗ	Ƌ	ƌ	ƍ	Ǝ	Ə	Ɛ	Ƒ	ƒ	Ɠ	Ɣ	ƕ	Ɩ	Ɨ	Ƙ	ƙ	ƚ	ƛ	Ɯ	Ɲ	ƞ	Ɵ	
01A0	Ơ	ơ	Ƣ	ƣ	Ƥ	ƥ	Ʀ	Ƨ	ƨ	Ʃ	ƪ	ƫ	Ƭ	ƭ	Ʈ	Ư	ư	Ʊ	Ʋ	Ƴ	ƴ	Ƶ	ƶ	Ʒ	Ƹ	ƹ	ƺ	ƻ	Ƽ	ƽ	ƾ	ƿ	
01C0	ǀ	ǁ	ǂ	ǃ	Ǆ	ǅ	ǆ	Ǉ	ǈ	ǉ	Ǌ	ǋ	ǌ	Ǎ	ǎ	Ǐ	ǐ	Ǒ	ǒ	Ǔ	ǔ	Ǖ	ǖ	Ǘ	ǘ	Ǚ	ǚ	Ǜ	ǜ	ǝ	Ǟ	ǟ	
01E0	Ǡ	ǡ	Ǣ	ǣ	Ǥ	ǥ	Ǧ	ǧ	Ǩ	ǩ	Ǫ	ǫ	Ǭ	ǭ	Ǯ	ǯ	ǰ	Ǳ	ǲ	ǳ	Ǵ	ǵ	Ƕ	Ƿ	Ǹ	ǹ	Ǻ	ǻ	Ǽ	ǽ	Ǿ	ǿ	
0200	Ȁ	ȁ	Ȃ	ȃ	Ȅ	ȅ	Ȇ	ȇ	Ȉ	ȉ	Ȋ	ȋ	Ȍ	ȍ	Ȏ	ȏ	Ȑ	ȑ	Ȓ	ȓ	Ȕ	ȕ	Ȗ	ȗ	Ș	ș	Ț	ț	Ȝ	ȝ	Ȟ	ȟ	
0220	Ƞ	ȡ	Ȣ	ȣ	Ȥ	ȥ	Ȧ	ȧ	Ȩ	ȩ	Ȫ	ȫ	Ȭ	ȭ	Ȯ	ȯ	Ȱ	ȱ	Ȳ	ȳ	ȴ	ȵ	ȶ	ȷ	ȸ	ȹ	Ⱥ	Ȼ	ȼ	Ƚ	Ⱦ	ȿ	
0240	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	


THANK YOU