Introduction to Embedded Systems

Unit 1.2: Number Systems and Data Formats

Alexander Yemane
BCIT
INCS 3610
Fall 2025

Number Systems

From zero and one to beyond

- Embedded systems understand logic 1 and 0 and process the information that consists of them
- For some devices, logic 1 corresponds to 5V, while logic 0 corresponds to about 0V.
- For other devices, logic 1 can be 3.3V or even may be less while logic 0 is 0V.
- By combining these two symbols in any consecutive combination or in different combinations; data, instruction codes and different signals are produced in a format that digital systems can understand.

Bits, bytes, and words

- **Nibble**: 4-bit-width
- **Byte**: 8-bit width
- **Word**: 16-bit width
- **Double word**: 32-bit width
- **Quad**: 64-bit width

- Individual bits in a word are named after their position, starting from the right with 0: bit 0 (b0), bit 1 (b1), and so on. Symbolically, an n-bit word is denoted as
  - $b_{n-1}b_{n-2}...b_1b_0$

Binary, octal, and hexadecimal number systems

- Binary number system = base 2.
  - For binary numbers, use suffix 'B' or 'b'
- Octal number system = base 8.
  - For octal numbers, use suffix 'Q' or 'q'
- Hexadecimal number system = base 16.
  - For hex numbers use suffix 'H' or 'h', or else prefix 0x.
- Base ten numbers have no suffix.
- Hence, we write 1011B or 1011b for 1011_2, 25Q or 25q for 25_8, and OA5H or OA5h or OxA5 for $A5_{16}$.

Binary, octal, and hexadecimal number systems

Table 2.2 Basic numeric systems

Decimal	Binary	Octal	Hex	Decimal	Binary	Octal	Hex
0	0000	0	0	8	1000	10	8
1	0001	1	1	9	1001	11	9
2	0010	2	2	10	1010	12	A
3	0011	3	3	11	1011	13	B
4	0100	4	4	12	1100	14	C
5	0101	5	5	13	1101	15	D
6	0110	6	6	14	1110	16	E
7	0111	7	7	15	1111	17	F

Hexadecimal representation

- Convenient to divide any size of binary numbers into nibbles
- Represent each nibble as hexadecimal, e.g.:

0100	1101	0110	1011	1000	0011	0000	1111
4	D	6	B	8	3	0	F

- Hexadecimal number system has been universally adopted in the embedded systems literature as well as in debuggers.
- Thus, memory addresses, register contents, and pretty much everything else are expressed in hexadecimal integers without implying that they are a number.

Data Formats

Representing text in ASCII

- Textual information must also be stored as binary numbers.
- Each character is represented as a 7-bit number known as ASCII codes
  - E.g., ‘A’ is represented by 41h and ‘a’ by 61h

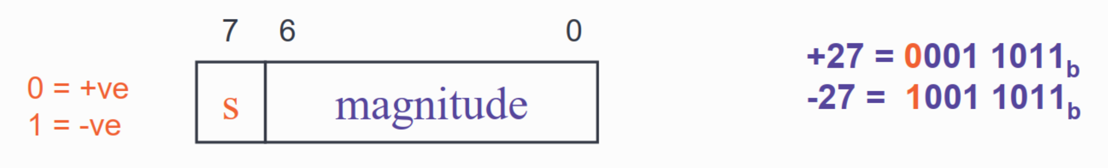
$b_3 - b_0$

$b_6 - b_4$	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	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
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

Signed numbers

- So far, numbers are assumed to be unsigned (i.e. positive)
- How to represent signed numbers?
- Remember: the range for unsigned and signed numbers are different
  - 8-bit unsigned number: 0 … +255
  - 8-bit signed number: –128 … +127

- Solution 1: *Sign-magnitude* - Use one bit to represent the *sign*, the remaining bits to represent *magnitude*



Signed numbers

- Solution 2: *Two's complement* – represent a negative number x by the number $2^N + x$, in an n-bit representation:
  - E.g., Encode –27 in 8-bit two's complement

$$256 - 27 = 229$$
$$229_{10} = 1110\ 0101_b$$

- So long as we only want to represent numbers with magnitude less than 2^{N-1}, the MSB is still the *sign* bit
  - E.g., Encode 27 in 8-bit two's complement

$$27_{10} = 0001\ 1011_b$$

Why two’s complement representation?

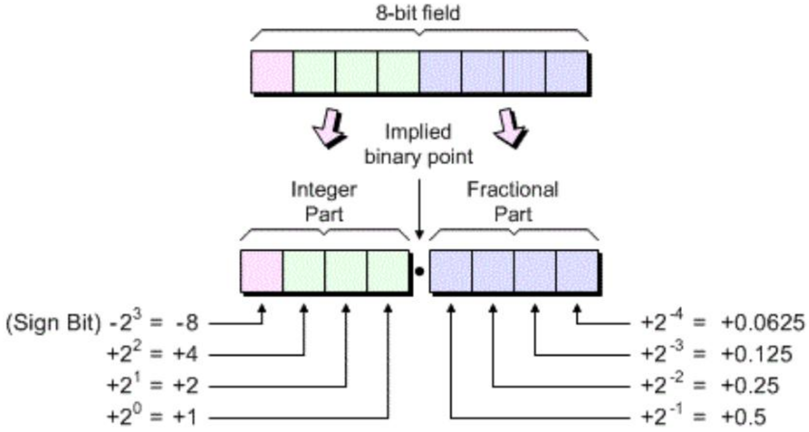
- Using the sign-magnitude pattern is easy to understand but requires quite complex circuitry
- If we represent signed numbers in two’s complement form, subtraction is the same as addition to the (two’s complemented) number (if we ignore any carry out)

$$
\begin{array}{rr}
27 & 0001\ 1011_b \\
-\ 17 & 0001\ 0001_b \\
\hline
+\ 10 & 0000\ 1010_b
\end{array}
$$

$$
\begin{array}{rr}
+27 & 0001\ 1011_b \\
+\ -17 & 1110\ 1111_b \\
\hline
+10 & 0000\ 1010_b
\end{array}
$$

Fixed point representation

- So far, we have concentrated on *integer* representation with the *fractional* part
- There is an implicit binary point to the right
- In general, the binary point can be in the middle of the word



Floating point representation

- Although fixed point representation can cope with numbers with fractions, the range of values that can represented is still limited.
- Alternative: use the equivalent of “scientific notation”, but in binary:

$$\text{number} = \underbrace{s}_{\text{sign}} \times \underbrace{m}_{\text{mantissa}} \times 2^{\overbrace{e}^{\text{exponent}}}$$

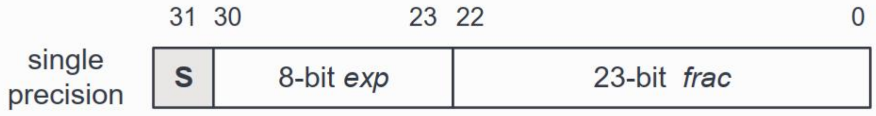
- E.g.,

10.5 in fixed point	1010.1_b
Move binary point to left	$1.0101_b \times 2^3$

$$10.5 = 1.3125 \times 8$$

IEEE-754 standard floating point

- 32-bit single precision floating point:



$$x = -1^{s} \times 2^{exp-127} \times 1.frac$$

$$1.175 \times 10^{-38} < |x| < 1.7 \times 10^{38}$$