Notes – Unit 2

REVIEW OF NUMBER SYSTEMS

BINARY NUMBER SYSTEM
In the decimal system, a decimal digit can take values from 0 to 9. For the binary system, the counterpart of the decimal digit is the **bi**nary digi**t**, or bit (that can take the value of 0 or 1).

- **Bit**: Unit of Information that a digital computer uses to process and retrieve data.
- **Binary number**: This is represented by a string of bits using the positional number representation: $b_{n-1}b_{n-2}\cdots b_1b_0$

BINARY TO DECIMAL CONVERSION
The binary number $b_{n-1}b_{n-2}\cdots b_1b_0$ can be converted to the positive decimal number it represents by the following formula:

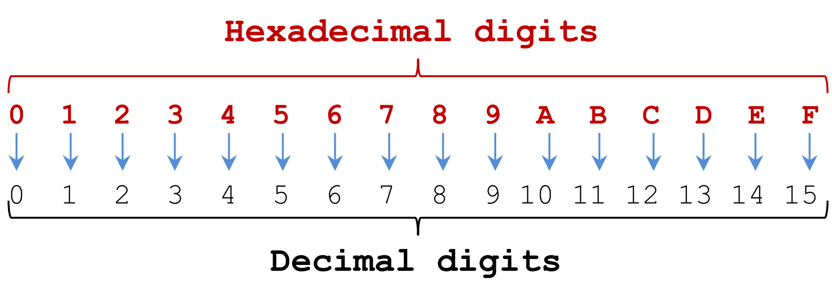
$$D = b_{n-1}\times 2^{n-1} + b_{n-2}\times 2^{n-2} + \cdots + b_1\times 2^1 + b_0\times 2^0$$

- **Maximum value for 'n' bits**: The maximum binary number is given by an n-bit string of 1's: $111\ldots111$. Then, the maximum decimal number is given by: $D = 2^{n-1} + 2^{n-2} + \cdots + 2^1 + 2^0 = 2^{n-1}$

- With 'n' bits, we can represent 2^n positive integer numbers from 0 to $2^n - 1$

HEXADECIMAL AND OCTAL NUMBER SYSTEMS
These number systems are very useful as they are short-hand notations for binary numbers.

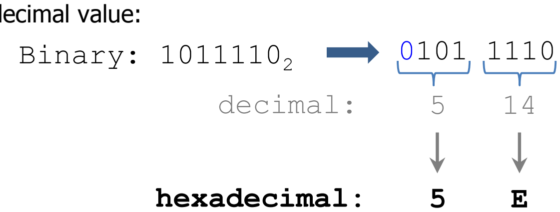
HEXADECIMAL NUMBERS
A hexadecimal digit is also called a *nibble*. A hexadecimal digit can take a value from 0 to 15. To avoid confusion, the numbers 10 to 15 are represented by letters (A-F):



- A hexadecimal number with 'n' nibbles is given by: $h_{n-1}h_{n-2}\cdots h_1h_0$. To convert a hexadecimal number into the positive decimal number it represents, we apply the following formula

$$D = h_{n-1}\times 16^{n-1} + h_{n-2}\times 16^{n-2} + \cdots + h_1\times 16^1 + h_0\times 16^0$$

- **Binary to hexadecimal conversion**: We group bits in groups of 4 (starting from the rightmost bit). If the last group of bits does not have bits, we append zeros to the left. Note that with 4 bits we can represent numbers from 0 to 15, i.e., 4 bits represent a hexadecimal digit. Therefore, to get the hexadecimal number, we independently convert each 4-bit group to its hexadecimal value:



Then: 01011110_2 = 0x5E

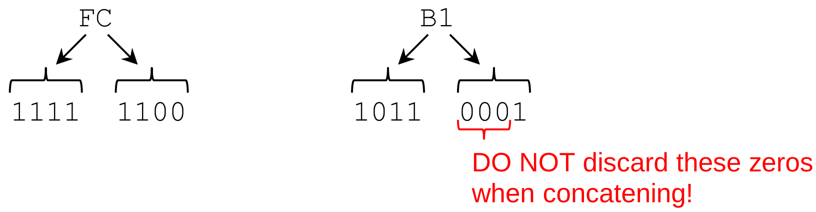
Verification:

$01011110_2 = 1\times 2^6 + 1\times 2^4 + 1\times 2^3 + 1\times 2^2 + 1\times 2^1 = 94$

$0x5E = 5\times 16^1 + E\times 16^0 = 94$

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

- **Hexadecimal to binary conversion**: We pick each hexadecimal digit and convert it to its 4-bit binary representation (always use 4 bits). The resulting binary number is the concatenation of all resulting 4-bit groups:



$0xFA = 11111100_2$
$0xB1 = 10110001_2$

OCTAL NUMBERS

An octal digit can take values between 0 and 7.

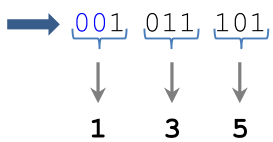
- An octal number with 'n' octal digits is given by: $o_{n-1}o_{n-2}\cdots o_1 o_0$. To convert an octal number into the positive decimal number it represents, we apply the following formula:

$$D = o_{n-1} \times 8^{n-1} + o_{n-2} \times 8^{n-2} + \cdots + o_1 \times 8^1 + o_0 \times 8^0$$

- The conversion between base-8 and base-2 resembles that of converting between base-16 and base-2. Here, we group binary numbers in 3-bit groups:

BINARY TO OCTAL

Binary: 1011101_2



octal:

Then: $01011101_2 = 135_8$

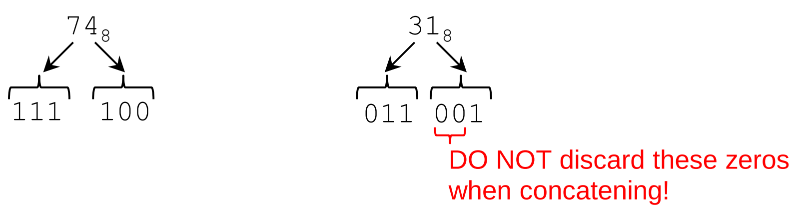
Verification:

$01011101_2 = 1\times2^6 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^0 = 93$

$135_8 = 1\times8^2 + 3\times8^1 + 5\times8^0 = 93$

binary	dec	oct
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7

OCTAL TO BINARY



$74_8 = 111100_2$
$31_8 = 011001_2$

UNITS OF INFORMATION

Nibble	Byte	KB	MB	GB	TB
4 bits	8 bits	2^{10}=1024 bytes	2^{20}=1024^2 bytes	2^{30}=1024^3 bytes	2^{40}=1024^4 bytes

- Note that the nibble (4 bits) is one hexadecimal digit. Also, one byte (8 bits) is represented by two hexadecimal digits.
- While KB, MB, GB, TB (and so on) should be powers of 10 in the International System, it is customary in digital jargon to use powers of 2 to represent them.
- In microprocessor systems, memory size is usually a power of 2 due to the fact that the maximum memory size is determined by the number of addresses the address bus can handle (which is a power of 2). As a result, it is very useful to use the definition provided here for KB, MB, GB, TB (and so on).
- Digital computers usually represent numbers utilizing a number of bits that is a multiple of 8. The simple hexadecimal to binary conversion may account for this fact as we can quickly convert a string of bits that is a multiple of 8 into a string of hexadecimals digits.
- The size of the data bus in a processor represents the computing capacity of a processor, as the data bus size is the number of bits the processor can operate in one operation (e.g.: 8-bit, 16-bit, 32-bit processor). This is also usually expressed as a number of bits that is a multiple of 8.

APPLICATIONS OF BINARY AND HEXADECIMAL REPRESENTATIONS

INTERNET PROTOCOL ADDRESS (IP ADDRESS):

- Hexadecimal numbers represent a compact way of representing binary numbers. The IP address is defined as a 32-bit number, but it is displayed as a concatenation of four decimal values separated by a dot (e.g., 129.26.53.76).
- The following figure shows how a 32-bit IP address expressed as a binary number is transformed into the standard IP address notation.

```
IP address (binary): 10000001000110100011010101001100

Conversion to      1000 0001 0001 1010 0011 0101 0100 1100
hexadecimal:         8    1    1    A    3    5    4    C    Grab pairs of
                                                            hexadecimal numbers
                       129       26        53        76     and convert each of
                                                            them to decimal.

IP address (hex): 0x811A354C

IP address notation: 129.26.53.76
```

- The 32-bit IP address expressed as binary number is very difficult to read. So, we first convert the 32-bit binary number to a hexadecimal number.
- The IP address expressed as a hexadecimal (0x811A354C) is a compact representation of a 32-bit IP address. This should suffice. However, it was decided to represent the IP address in a *'human-readable'* notation. In this notation, we grab pairs of hexadecimal numbers and convert each of them individually to decimal numbers. Then we concatenate all the values and separate them by a dot.
- **Important**: Note that the IP address notation (decimal numbers) is NOT the decimal value of the binary number. It is rather a series of four decimal values, where each decimal value is obtained by independently converting each two hexadecimal digits to its decimal value.

✓ Given that each decimal number in the IP address can be represented by 2 hexadecimal digits (or 8 bits), what is the range (min. value, max. value) of each decimal number in the IP address?
With 8 bits, we can represent $2^8 = 256$ numbers from 0 to 255.

✓ An IP address represents a unique device connected to the Internet. Given that the IP address has 32 bits (or 8 hexadecimal digits), the how many numbers can be represented (i.e., how many devices can connect to the Internet)?
$2^{32} = 4294967296$ devices.

✓ The number of devices that can be connected to the Internet is huge, but considering the number of Internet-capable devices that exists in the entire world, it is becoming clear that 32 bits is not going to be enough. That is why the Internet Protocol is being currently extended to a new version (IPv6) that uses 128 bits for the addresses. With 128 bits, how many Internet-capable devices can be connected to the Internet?
$2^{128} \approx 3.4 \times 10^{38}$ devices

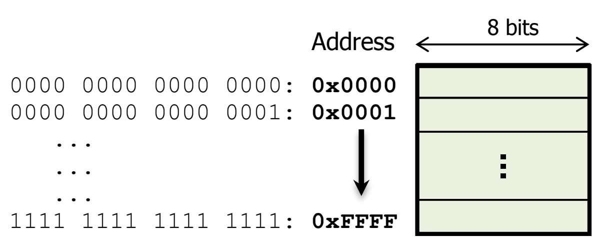
REPRESENTING GRAYSCALE PIXELS

A grayscale pixel is commonly represented with 8 bits. So, a grayscale pixel value varies between 0 and 255, 0 being the darkest (black) and 255 being the brightest (white). Any value in between represents a shade of gray.

MEMORY ADDRESSES

The address bus size in processors is usually determined by the number of memory positions it can address. For example, if we have a microprocessor with an address bus of 16 bits, we can handle up to 2^{16} addresses. If the memory content is one byte wide, then the processor can handle up to $2^{16} bytes = 64KB$.

Here, we use 16 bits per address, or 4 nibbles. The lowest address (in hex) is `0x0000` and highest address (in hex) is `0xFFFF`.



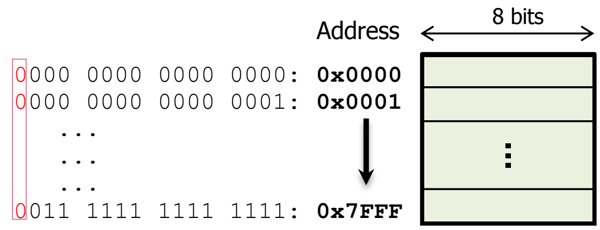
Examples:

- A microprocessor can only handle memory addresses from `0x0000` to `0x7FFF`. What is the address bus size? If the memory contents is one byte wide, what is the maximum size (in bytes) of the memory that we can connect?

  We want to cover all the cases from `0x0000` to `0x7FFF`:

  The range from `0x0000` to `0x7FFF` is akin to all possible cases with 15 bits. Thus, the address bus size is **15 bits**.

  We can handle $2^{15} bytes = 32KB$ of memory.

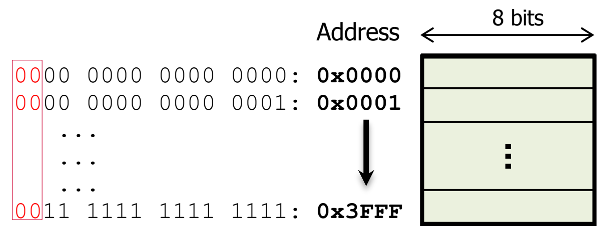


- A microprocessor can only handle memory addresses from `0x0000` to `0x3FFF`. What is the address bus size? If the memory contents is one byte wide, what is the maximum size (in bytes) of the memory that we can connect?

  We want to cover all the cases from `0x0000` to `0x3FFF`:

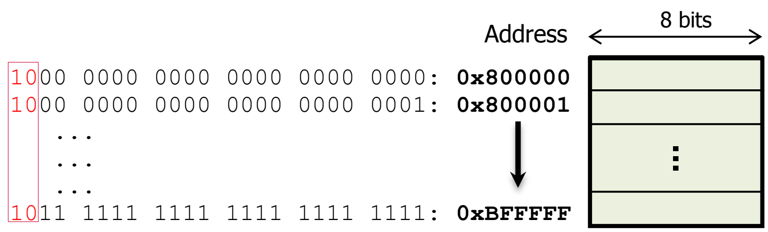
  The range from 0x0000 to 0x3FFF is akin to all possible cases with 14 bits. Thus, the address bus size is **14 bits**.

  We can handle $2^{14} bytes = 16KB$ of memory.



- A microprocessor has a 24-bit address line. We connect a memory chip to the microprocessor. The memory chip addresses are assigned the range `0x800000` to `0xBFFFFF`. What is the minimum number of bits required to represent addresses in that individual memory chip? If the memory contents is one byte wide, what is the memory size (in bytes)?

  By looking at the binary numbers from `0x80000` to `0xBFFFFF`, we notice that the addresses in that range require 24 bits. But all those addresses share the same first two MSBs: `10`. Thus, if we were to use only that memory chip, we do not need those 2 bits, and we only need **22 bits**.



  We can handle $2^{22} bytes = 4MB$ of memory.

- A memory has a size of 512KB, where each memory content is 8-bits wide. How many bits do we need to address the contents of this memory?

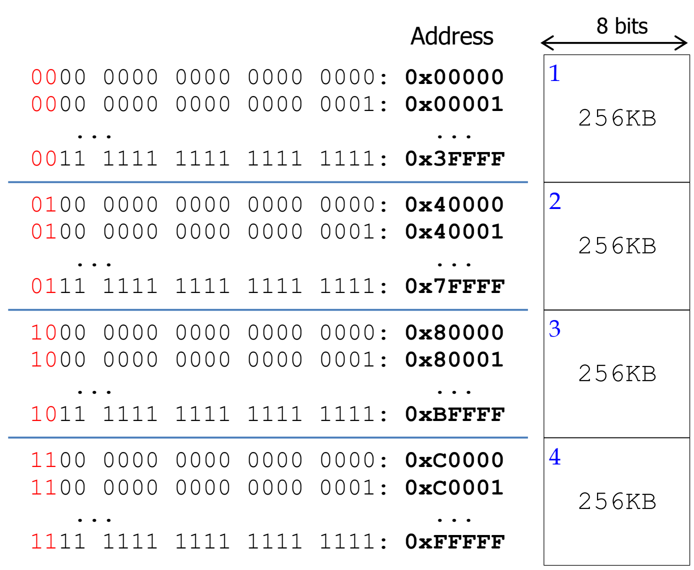
  Recall that: $512KB = 2^{19} bytes$. So we need 19 bits to address the contents of this memory.
  In general, for a memory with N address positions, the number of bits to address those position is given by: $\lceil \log_2 N \rceil$

- A 20-bit address line in a microprocessor with an 8-bit data bus handles 1 MB (2^{20} $bytes$) of data. We want to connect four 256 KB memory chips to the microprocessor. Provide the address ranges that each memory device will occupy.

  For a 20-bit address: we have 5 hexadecimal digits that go from `0x00000` to `0xFFFFF`.

  We need to divide the 2^{20} memory positions into 4 groups, each with 2^{18} memory positions. Each group will correspond to the memory positions of one of the 256KB memory chips. Note how at each group, the 2 MSBs are the same.

  * Each memory chip can handle 256KB of memory. $256KB = 2^{18} bytes$. Thus, each memory chip only requires 18 bits.



BINARY CODES

- We know that with 'n' bits we can represent 2^n numbers from 0 to $2^n - 1$. This is a commonly used range. However, with 'n' bits, we can represent 2^n numbers in any range. Moreover, we can represent 2^n symbols.
- If we have N symbols to represent, the number of bits required is given by: $\lceil \log_2 N \rceil$. For example:
  What is the minimum number of bits to represent?
  - Minimum number of bits to represent 70,000 colors: → Number of bits: $\lceil \log_2 70000 \rceil = 17$ bits
  - Minimum number of bits to represents numbers between 15,000 and 19,096?
    → There are 19,096-15,000+1=4097 numbers → Number of bits: $\lceil \log_2 4097 \rceil = 13$ bits

7-bit US-ASCII character-encoding scheme. Each character is represented by 7 bits, so we have $2^7 = 128$ characters. Each character (or symbol) is said to have a binary code:

Hex	Dec	Char		Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL	null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH	Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX	Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX	End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT	End of transmission	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ	Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK	Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL	Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS	Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB	Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF	New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT	Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF	Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR	Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO	Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI	Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE	Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1	Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2	Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3	Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4	Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK	Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN	Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB	End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN	Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM	End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB	Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC	Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS	File separator	0x3C	60	<	0x5C	92	\	0x7C	124	|
0x1D	29	GS	Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS	Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US	Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

Unicode: it can represent more than 110,000 characters and attempts to cover all world's scripts. A common character encoding is UTF-16, which uses 2 pair of 16-bit units: For most purposes, a 16 bit unit suffices ($2^{16} = 65536$ characters):

ϑ (Greek theta symbol) = 03D1 Ω (Greek capital letter Omega): 03A9 Ж (Cyrillic capital letter zhe): 0416

BCD Code:
In this coding scheme, decimal numbers are represented in binary form by independently encoding each decimal digit in binary form (4 bits). Note that only values from 0 are 9 are represented here.

- This is a very useful code for input devices (e.g.: keypad). But it is not a coding scheme suitable for arithmetic operations. Also, recall that 6 binary values (from 1010 to 1111) wasted.
- Decimal number **47**: In BCD format, this would be: **0100 0111**$_2$
  Note that the BCD is NOT the binary number, since 47 is represented by 101111 in binary form (requiring 6 bits).

REPRESENTATION OF SIGNED NUMBERS

- For an n-bit number $b_{n-1}b_{n-2}\cdots b_1b_0$, there exist three common signed representations: signed-magnitude, 1's complement, and 2's complement. In these 3 representations, the MSB always tells us whether the number is positive (MSB=0) or negative (MSB=1).

SIGN-AND-MAGNITUDE (SM):
- Here, the sign and the magnitude (value) are represented separately. The MSB represents the sign and the remaining n-1 bits the magnitude.
- **Examples** (n=4): `0110 = +6` `1110 = -6`

1'S COMPLEMENT (1C):
- In this representation, if the MSB=0, the remaining n-1 bits represent the magnitude. Negative numbers (MSB=1) are obtained by inverting the sign of the positive numbers. To invert the sign of a number in 1's complement representation, we apply the 1's complement *operation* to the number, which consists on inverting all the bits.
- **Examples** (n=4):
  - ✓ +6=0110 → -6=1001, +5=0101 → -5=1010, +7=0111 → -7=1000.
  - ✓ If -6=1001, we get +6 by applying the 1's complement *operation* to 1001 → +6 = 0110
  - ✓ What is the decimal value of 1100? We first apply the 1's complement *operation* to 1100, which results in 0011 (+3). Thus 1100=-3.
  - ✓ What is the 1's complement representation of -4? We know that +4=0100. To get -4, we apply the 1's complement operation to 0100, which results in 1011. Thus 1011=-4.

2'S COMPLEMENT (2C):
- In this representation, if the MSB=0, the remaining n-1 bits represent the magnitude. Negative numbers (MSB=1) are obtained by inverting the sign of the positive numbers. To invert the sign of a number in 2's complement representation, we apply the 2's complement *operation* to the number, which consists on inverting all the bits and add 1.
- **Examples** (n=4):
  - ✓ +6=0110 → -6=1010, +5=0101 → -5=1011, +7=0111 → -7=1001.
  - ✓ If -6=1010, we get +6 by applying the 2's complement *operation* to 1010 → +6 = 0110
  - ✓ What is the decimal value of 1101? We first apply the 2's complement *operation* to 1101, which results in 0011 (+3). Thus 1101=-3.
  - ✓ What is the 2's complement representation of -4? We know that +4=0100. To get -4, we apply the 2's complement operation to 0100, which results in 1100. Thus 1100=-4.

The following table summarizes the signed representations for a 4-bit number:

n=4:	SIGNED REPRESENTATION		
$b_3b_2b_1b_0$	**Sign-and-magnitude**	**1's complement**	**2's complement**
0 0 0 0	0	0	0
0 0 0 1	1	1	1
0 0 1 0	2	2	2
0 0 1 1	3	3	3
0 1 0 0	4	4	4
0 1 0 1	5	5	5
0 1 1 0	6	6	6
0 1 1 1	7	7	7
1 0 0 0	0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	0	-1
Range for n bits:	$[-(2^{n-1}-1), 2^{n-1}-1]$	$[-(2^{n-1}-1), 2^{n-1}-1]$	$[-2^{n-1}, 2^{n-1}-1]$

- 1C and 2C are representations of signed numbers. 1C and 2C represent both negative and positive numbers. Do not confuse the 1C and 2C representations with the 1C and 2C operations.
- Note that the sign-and-magnitude and the 1's complement representations have a redundant representation for zero. This is not the case in 2's complement, which can represent an extra number.
- In 2C, the number -8 can be represented with 4 bits: -8=1000. To obtain +8, we apply the 2C operation to 1000, which results in 1000. But 1000 cannot be a positive number. This means that we require 5 bits to represent +8=01000.

BINARY ARITHMETIC

UNSIGNED NUMBERS

ADDITION:

- In the example, we add two 8-bit numbers using binary representation and hexadecimal representation (this is a short-hand notation). Note that every summation of two digits (binary or hexadecimal) generates a carry when the summation requires more than one digit. Also, note that c_0 is the *carry in* of the summation. This is usually zero.

```
       c8=0 c7=0 c6=1 c5=1 c4=1 c3=1 c2=1 c1=0 c0=0        c2=0 c1=1 c0=0
0x3F =  0    0    1    1    1    1    1    1   +              3    F   +
0xB2 =  1    0    1    1    0    0    1    0                  B    2
------------------------------------------------            -----------
0xF1 =  1    1    1    1    0    0    0    1                  F    1
```

- The last carry (c_8 when n=8) is the *carry out* of the summation. If it is zero, it means that the summation can be represented with 8 bits. If it is one, it means that the summation requires more than 8 bits (in fact 9 bits); this is called an overflow. In the example, we add two numbers and overflow occurs: an extra bit (in red) is required to correctly represent the summation.

```
       c8=1 c7=1 c6=1 c5=1 c4=1 c3=1 c2=1 c1=0 c0=0        c2=1 c1=1 c0=0
0x3F =  0    0    1    1    1    1    1    1   +              3    F   +
0xC2 =  1    1    0    0    0    0    1    0                  C    2
------------------------------------------------            -----------
     1  0    0    0    0    0    0    0    1             1    0    1
```

- **Multi-precision addition**: Microprocessors usually have fixed arithmetic units such as an 8-bit adder that has a *carry in* input and a *carry out* output. In the example, we add two 16-bit numbers, and we do it in two operations: The first one adds the two least significant bytes. If there is a *carry out*, it is stored in a special register. The second operation adds the two most significant bytes where the *carry in* corresponds to the *carry out* of the previous operation. We can keep doing this in order to add larger numbers, but we have to make sure the microprocessor can store that result somewhere.

```
c4=0 c3=0 c2=1 c1=1 c0=0      c2=0 c1=0 c0=1      c2=1 c1=1 c0=0
      4    5    3    F  +           4    5  +           3    F  +
      A    1    C    2              A    1              C    2
   ------------------            ----------          ----------
      E    7    0    1              E    7              0    1
```

SUBTRACTION:

- In the example, we subtract two 8-bit numbers using the binary and hexadecimal (this is a short-hand notation) representations. A subtraction of two digits (binary or hexadecimal) generates a borrow when the difference is negative. So, we borrow 1 from the next digit so that the difference is positive. Recall that a borrow in a subtraction of two digits is an extra 1 that we need to subtract. Also, note that b_0 is the *borrow in* of the summation. This is usually zero.

```
       b8=0 b7=0 b6=0 b5=0 b4=1 b3=1 b2=1 b1=1 b0=0        c2=0 c1=1 c0=0
0x3A =  0    0    1    1    1    0    1    0   -              3    A   -
0x2F =  0    0    1    0    1    1    1    1                  2    F
------------------------------------------------            -----------
0x0B =  0    0    0    0    1    0    1    1                  0    B
```

- The last borrow (b_8 when n=8) is the *borrow out* of the subtraction. If it is zero, it means that the difference is positive and can be represented with 8 bits. If it is one, it means that the difference is negative and we need to borrow 1 from the next digit. In the example, we subtract two 8-bit numbers, the result we have borrows 1 from the next digit.

```
       b8=1 b7=1 b6=0 b5=0 b4=0 b3=1 b2=0 b1=1 b0=0        b2=1 b1=0 b0=0
0x3A =  0    0    1    1    1    0    1    0   -              3    A   -
0x75 =  0    1    1    1    0    1    0    1                  7    5
------------------------------------------------            -----------
0xC5 =  1    1    0    0    0    1    0    1                  C    5
```

- **Multi-precision subtraction**: A fixed arithmetic unit such as an 8-bit subtractor usually has a *borrow in* input and a *borrow out* output. In the example, we subtract two 16-bit numbers in two steps: First, we subtract the two least significant bytes. If there is a *borrow out*, it is stored in a special register. Next, we subtract the two most significant bytes where the *borrow in* corresponds to the *borrow out* of the previous operation. We can keep doing this to subtract larger numbers, but we have to make sure the microprocessor can store that result somewhere. If the final result has a borrow, the result is incorrect.

```
b4=0 b3=0 b2=1 b1=1 b0=0      b2=0 b1=1 b0=1      b2=1 b1=0 b0=0
      B    2    3    A  -           B    2  -           3    A  -
      3    7    7    5              3    7              7    5
   ------------------            ----------          ----------
      7    A    C    5              7    A              C    5
```

SIGNED NUMBERS (2'S COMPLEMENT)

- The advantage of the 2's complement representation is that the summation can be carried out using the same circuitry as that of the unsigned summation. Here the operands can either be positive or negative.
- We show addition examples of two 8-bit signed numbers. The *carry out* c_8 is not enough to determine overflow. Here, if $c_8 \neq c_7$ there is overflow. If $c_8 = c_7$, no overflow and we can ignore c_8. Thus, the overflow bit is equal to c_8 XOR c_7.
- Note that overflow happens when the summation falls outside the 2's complement range for 8 bits: $[-2^7, 2^7-1]$.

```
       c8=0 c7=1 c6=0 c5=1 c4=1 c3=1 c2=0 c1=0 c0=0
 +92  =  0 1 0 1 1 1 0 0 +
 +78  =  0 1 0 0 1 1 1 0
 ---------------------------
+170  =  0 1 0 1 0 1 0 1 0

overflow = c8⊕c7=1 -> overflow!
+170 ∉ [-2^7, 2^7-1] -> overflow!
```

```
       c8=1 c7=0 c6=1 c5=0 c4=0 c3=0 c2=0 c1=0 c0=0
 -92  =  1 0 1 0 0 1 0 0 +
 -78  =  1 0 1 1 0 0 1 0
 ---------------------------
-170  =  1 0 1 0 1 0 1 1 0

overflow = c8⊕c7=1 -> overflow!
-170 ∉ [-2^7, 2^7-1] -> overflow!
```

```
       c8=1 c7=1 c6=1 c5=1 c4=0 c3=0 c2=0 c1=0 c0=0
 +92  =  0 1 0 1 1 1 0 0 +
 -78  =  1 0 1 1 0 0 1 0
 ---------------------------
 +14  =  X 0 0 0 0 1 1 1 0
overflow = c8⊕c7=0 -> no overflow
+14 ∈ [-2^7, 2^7-1] -> no overflow
```

```
       c8=0 c7=0 c6=0 c5=0 c4=1 c3=1 c2=0 c1=0 c0=0
 -92  =  1 0 1 0 0 1 0 0 +
 +78  =  0 1 0 0 1 1 1 0
 ---------------------------
 -14  =  X 1 1 1 1 0 0 1 0
overflow = c8⊕c7=0 -> no overflow
-14 ∈ [-2^7, 2^7-1] -> no overflow
```

- In general, for an n-bit number, overflow occurs when the summation falls outside the range $[-2^{n-1}, 2^{n-1}-1]$. The overflow bit can quickly be computed as c_n XOR c_{n-1}.

- **Subtraction**: Note that $A - B = A + 2C(B)$. To subtract two numbers represented in 2's complement arithmetic, we first apply the 2's complement operation to B, and then add the numbers. So, in 2's complement arithmetic, subtraction is actually an addition of two numbers.

BCD ADDITION

- BCD addition is the typical decimal addition. If we want a circuit that performed BCD addition, this is what we would get:

```
28(BCD) = |0010|1000| +          1(BCD) = 0001
47(BCD) = |0100|0111|            2(BCD) = 1000 +        8(BCD) = 1000 +
-----------------------    =     4(BCD) = 0111          7(BCD) = 0111
75(BCD) = |0111|0101|            ---------------        -------------------
                                 7(BCD) = 0111        15(BCD) = 0001 0101
```

- To avoid designing a custom circuit to do this, we want to use the same circuitry for binary addition. If we input the BCD codes of 28 and 47 in a binary adder, they would be interpreted as 0x28+0x47=0x6F which is not the BCD number 75 = 0111 0101 that we want. Note that for the lower order nibble, the sum is 0x8+0x7=0xF=1111. And what we want is 15 (BCD) = 0001 0101, where 1 is the carry to the next higher nibble. There is a difference of 0x6 between 0x15 and 0xF. So, to get the proper BCD result we need to add 0x6 to 0x6F = 0x75.
- Another example: 19+57=76. 0x19+0x47=0x70. The results looks like a BCD code but it is incorrect, we need to add 0x06: 0x70+0x06=0x76.
- In general, if the summation of two nibbles is greater than 0x9, we add 0x6 to the result.

```
0x28 = 0010 |1000| +
0x47 = 0100 |0111|
-------------------
0x6F = 0110 |1111| +
0x06 =       0110
-------------------
0x75 = 0111 0101
```

- For example: 197995 + 353375 = 0x4EAD0A. To correct it, we do 0x4EAD0A + 066666 = 0x551370. We can also do this using **multiprecision addition**, where only bytes can be added at a time. The carries can come from either the normal binary addition or from the operation that adds 6 to a particular nibble. This is depicted in the figure:

```
c6=0 c5=0 c4=0 c3=0 c2=1 c1=0 c0=0
 1 9 7 9 9 5 +
 3 5 3 3 7 5
 -------------
 4 E A D 0 A +
 0 6 6 6 6 6
 -------------
 5 5 1 3 7 0
```

```
c2=0 c1=0 c0=1        c2=0 c1=0 c0=1        c2=1 c1=0 c0=0
 1 9 +                 7 9 +                 9 5 +
 3 5                   3 3                   7 5
 -----                 -----                 -----
 4 F +          c2=1  |A| D +                0 A +
 0 6                  |6| 6                  6 6
 -----                 -----                 -----
 5 5                  |1| 3                  7 0
```