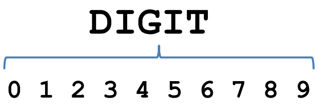
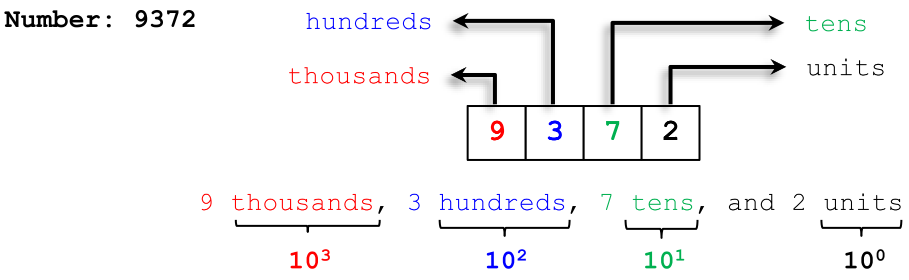
Notes - Unit 4

UNSIGNED INTEGER NUMBERS

DECIMAL NUMBER SYSTEM



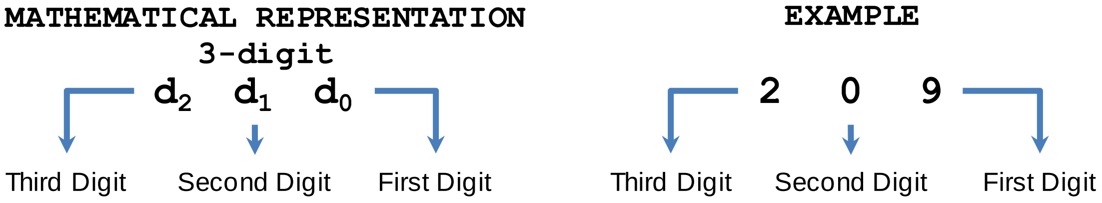
- A decimal digit can take values from 0 to 9:
- Digit-by-digit representation of a positive integer number (powers of 10):



$$9372 = 9\times10^3 + 3\times10^2 + 7\times10^1 + 2\times10^0$$

POSITIONAL NUMBER REPRESENTATION

- Let's consider the numbers from 0 to 999. We represent these numbers with 3 digits (each *digit* being a number between 0 and 9). We show a 3-digit number using the positional number representation:



- The *positional number representation* allows us to express the decimal value using **powers of ten**: $d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$. Example:

Decimal Number	3-digit representation $d_2d_1d_0$	Powers of 10: $d_2\times10^2 + d_1\times10^1 + d_0\times10^0$
0	000	$0\times10^2 + 0\times10^1 + 0\times10^0$
9	009	$0\times10^2 + 0\times10^1 + 9\times10^0$
11	011	$0\times10^2 + 1\times10^1 + 1\times10^0$
25	025	$0\times10^2 + 2\times10^1 + 5\times10^0$
90	090	$0\times10^2 + 9\times10^1 + 0\times10^0$
128	128	$1\times10^2 + 2\times10^1 + 8\times10^0$
255	255	$2\times10^2 + 5\times10^1 + 5\times10^0$

Exercise: Write down the 3-digit and the powers of ten representations for the following numbers:

Decimal Number	3-digit representation	$d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$
5		
254		
100		
99		

General Case:

- Positional number representation for an integer positive number with n digits: $d_{n-1}d_{n-2}\ ...\ d_1d_0$
  Decimal Value:

$$D = \sum_{i=0}^{i=n-1} d_i \times 10^i = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_1 \times 10^1 + d_0 \times 10^0$$

- Example: 1098324 (7 digits). $1098324 = 1 \times 10^6 + 0 \times 10^5 + 9 \times 10^4 + 8 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$
  203476 (6 digits). $203476 = 2 \times 10^5 + 0 \times 10^4 + 3 \times 10^3 + 4 \times 10^2 + 7 \times 10^1 + 6 \times 10^0$

Maximum value:

- The table presents the maximum attainable value for a given number of digits. What pattern do you find? Can you complete it for the highlighted cases (4 and 6)?

Number of digits	Maximum value	Range
1	$9 = 10^1-1$	$0 \rightarrow 9 \equiv 0 \rightarrow 10^1-1$
2	$99 = 10^2-1$	$0 \rightarrow 99 \equiv 0 \rightarrow 10^2-1$
3	$999 = 10^3-1$	$0 \rightarrow 999 \equiv 0 \rightarrow 10^3-1$
4		
5	$99999 = 10^5-1$	$0 \rightarrow 99999 \equiv 0 \rightarrow 10^5-1$
6		
…		
n	$999…999 = 10^n-1$	$0 \rightarrow 999…999 \equiv 0 \rightarrow 10^n-1$

- **Maximum value for a number with 'n' digits:** Based on the table, the maximum decimal value for a number with 'n' digits is given by:

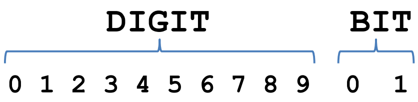
$$D = \underbrace{999...999}_{n\ digits} = 9 \times 10^{n-1} + 9 \times 10^{n-2} + \ ...\ + 9 \times 10^1 + 9 \times 10^0 = 10^n-1$$

⇒ **With 'n' digits, we can represent 10^n positive integer numbers from 0 to 10^n-1.**

✧ With 7 digits, what is the range (starting from 0) of positive numbers that we can represent? How many different numbers can we represent?

BINARY NUMBER SYSTEM

- We are used to the decimal number system. However, there exist other number systems: octal, hexadecimal, vigesimal, binary, etc. In particular, binary numbers are very practical as they are used by digital computers. For binary numbers, the counterpart of the decimal digit (that can take values from 0 to 9) is the binary digit, or bit (that can take the value of 0 or 1).

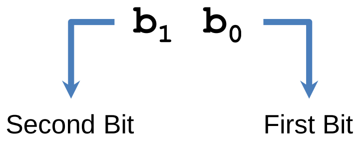


- **Bit**: Unit of information that a computer uses to process and retrieve data. It can also be used as a Boolean variable (see Unit 1).

- **Binary number**: This is represented by a string of bits using the positional number representation: $b_{n-1}b_{n-2} \ldots b_1b_0$

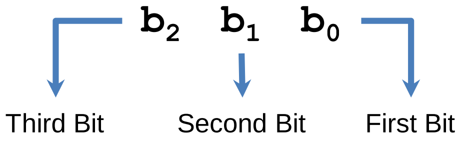
- **Converting a binary number into a decimal number**: The following figure depicts two cases: 2-bit numbers and 3-bit numbers. Note that the positional representation with powers of two let us obtain the decimal value (*integer positive*) of the binary number.

MATHEMATICAL REPRESENTATION
2-bit



Binary number b_1b_0	Powers of 2: $b_1\times2^1 + b_0\times2^0$	Decimal Number
00	$0\times2^1 + 0\times2^0$	0
01	$0\times2^1 + 1\times2^0$	1
10	$1\times2^1 + 0\times2^0$	2
11	$1\times2^1 + 1\times2^0$	3

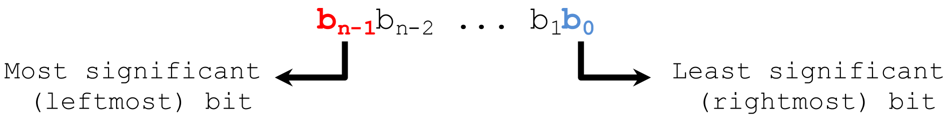
MATHEMATICAL REPRESENTATION
3-bit



Binary number $b_2b_1b_0$	Powers of 2: $b_2\times2^2 + b_1\times2^1 + b_0\times2^0$	Decimal Number
000	$0\times2^2 + 0\times2^1 + 0\times2^0$	0
001	$0\times2^2 + 0\times2^1 + 1\times2^0$	1
010	$0\times2^2 + 1\times2^1 + 0\times2^0$	2
011	$0\times2^2 + 1\times2^1 + 1\times2^0$	3
100	$1\times2^2 + 0\times2^1 + 0\times2^0$	4
101	$1\times2^2 + 0\times2^1 + 1\times2^0$	5
110	$1\times2^2 + 1\times2^1 + 0\times2^0$	6
111	$1\times2^2 + 1\times2^1 + 1\times2^0$	7

General case:

- Positional number representation for a binary number with 'n' bits:



The binary number can be converted to a positive decimal number by using the following formula:

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

- To avoid confusion, we usually write a binary number and attach a suffix '2': $(b_{n-1}b_{n-2} \dots b_1b_0)_2$
- Example: $6\ bits: (101011)_2 \equiv D = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 43$
  $4\ bits: (1011)_2 \equiv D = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$
- Maximum value for a given number of bits. Complete the tables for the highlighted cases (4 and 6):

Number of bits	Maximum value	Range
1	$1_2 \equiv 2^1-1$	$0 \rightarrow 1_2 \equiv 0 \rightarrow 2^1-1$
2	$11_2 \equiv 2^2-1$	$0 \rightarrow 11_2 \equiv 0 \rightarrow 2^2-1$
3	$111_2 \equiv 2^3-1$	$0 \rightarrow 111_2 \equiv 0 \rightarrow 2^3-1$
4		
5	$11111_2 \equiv 2^5-1$	$0 \rightarrow 11111_2 \equiv 0 \rightarrow 2^5-1$
6		
…		
n	$111…111_2 \equiv 2^n-1$	$0 \rightarrow 111…111_2 \equiv 0 \rightarrow 2^n-1$

- **Maximum value for 'n' bits:** The maximum binary number is given by an n-bit string of 1's: 111…111. Then, the maximum decimal numbers is given by:

$$D = \underbrace{111...111}_{n\ bits} = 1\times2^{n-1} + 1\times2^{n-2} + \ ...\ + 1\times2^1 + 1\times2^0 = 2^n-1$$

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

- The case **n=8** bits is of particular interest, as a string of 8 bits is called a byte. For 8-bit numbers, we have 256 numbers in the range 0 to 2^8-1 ≡ 0 to 255.

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

Most significant (leftmost) bit ← b_7

b_0 → Least significant (rightmost) bit

- The table shows some examples:

Decimal Number	8-bit format $b_7b_6b_5b_4b_3b_2d_1d_0$	$b_7\times2^7 + b_6\times2^6 + b_5\times2^5 + b_4\times2^4 + b_3\times2^3 + b_2\times2^2 + b_1\times2^1 + b_0\times2^0$
0	00000000	$0\times2^7 + 0\times2^6 + 0\times2^5 + 0\times2^4 + 0\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0$
9	00001001	$0\times2^7 + 0\times2^6 + 0\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$
11	00001011	$0\times2^7 + 0\times2^6 + 0\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
25	00011001	$0\times2^7 + 0\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$
90	01011010	$0\times2^7 + 1\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$
128	10000000	$1\times2^7 + 0\times2^6 + 0\times2^5 + 0\times2^4 + 0\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0$
255	11111111	$1\times2^7 + 1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0$

Exercise: Convert the following binary numbers (positive integers) to their decimal values:

8-bit representation	$b_7\times2^7+b_6\times2^6+b_5\times2^5+b_4\times2^4+b_3\times2^3+b_2\times2^2+b_1\times2^1+b_0\times2^0$	Decimal Number
00000001		
00001001		
10000101		
10000111		
11110011		

CONVERSION OF A NUMBER IN ANY BASE TO THE DECIMAL SYSTEM

- To convert a number of base 'r' (r = 2, 3,4 ,…) to decimal, we use the following formula:
  Number in base 'r': $(r_{n-1}r_{n-2}\dots r_1r_0)_r$
  Conversion to decimal:

$$D=\sum_{i=0}^{i=n-1} r_i\times r^i = r_{n-1}\times r^{n-1}+r_{n-2}\times r^{n-2}+\cdots+r_1\times r^1+r_0\times r^0$$

  Also, the maximum decimal value for a number in base 'r' with 'n' digits is:

$$D=rrr\dots rrr=r\times r^{n-1}+r_{n-2}\times r^{n-2}+\cdots+r\times r^1+r\times r^0=r^n-1$$

- Example: Base-8:

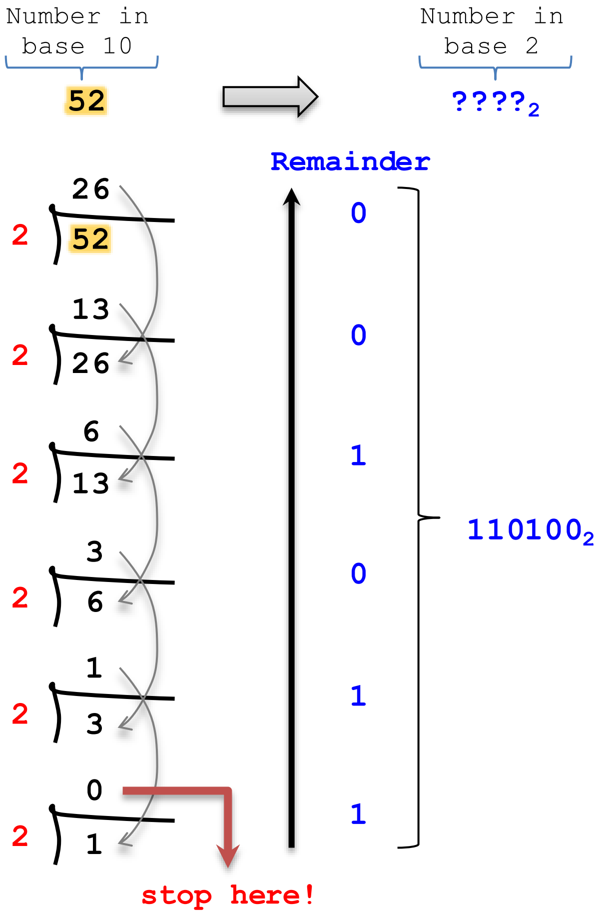
Number of digits	Maximum value	Range
1	$7_8\ \equiv\ 8^1-1$	$0\ \rightarrow\ 7_8\ \equiv\ 0\ \rightarrow\ 8^1-1$
2	$77_8\ \equiv\ 8^2-1$	$0\ \rightarrow\ 77_8\ \equiv\ 0\ \rightarrow\ 8^2-1$
3	$777_8\ \equiv\ 8^3-1$	$0\ \rightarrow\ 777_8\ \equiv\ 0\ \rightarrow\ 8^3-1$
…		
n	$777\dots777_8\ \equiv\ 8^n-1$	$0\ \rightarrow\ 777\dots777_8\ \equiv\ 0\ \rightarrow\ 8^n-1$

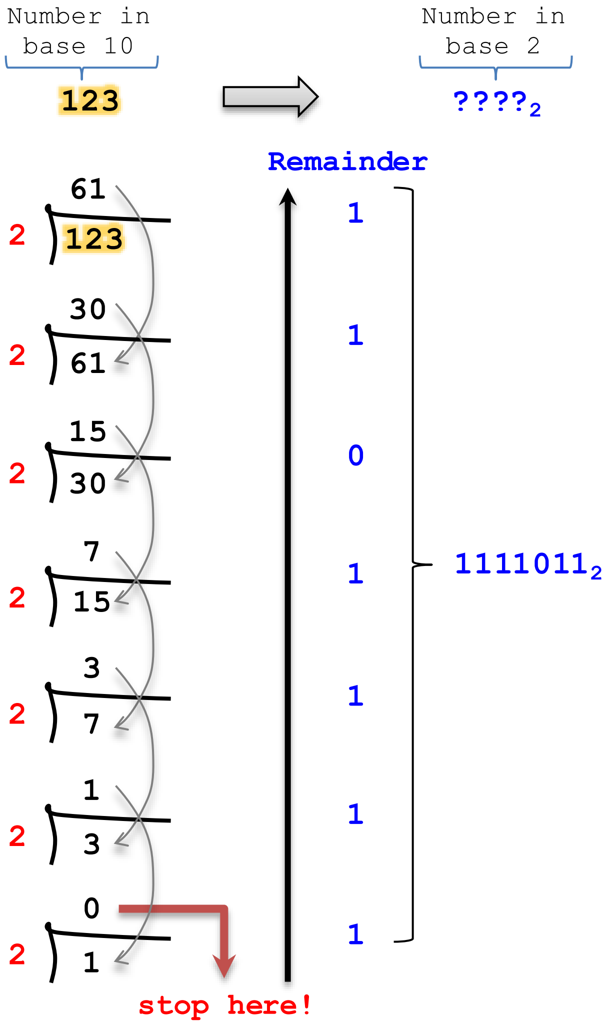
Examples:

- $(50632)_8$: Number in base 8 (octal system)
  Number of digits: n = 5
  Conversion to decimal: $D=5\times8^4+0\times8^3+6\times8^2+3\times8^1+2\times8^0=20890$
- $(3102)_4$: Number in base 4 (quaternary system)
  Number of digits: n = 4
  Conversion to decimal: $D=3\times4^3+1\times4^2+0\times4^1+2\times4^0=210$

CONVERSION OF DECIMAL (INTEGER POSITIVE) TO BINARY NUMBERS

- Examples:



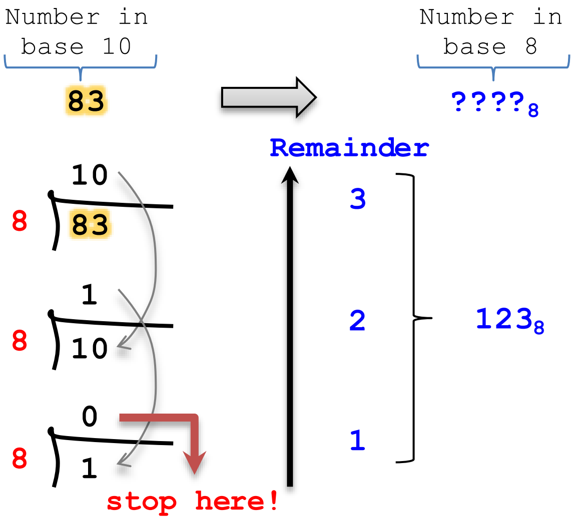


- Note that some numbers require fewer bits than others. If we want to use a specific bit representation, e.g., 8-bit, we just need to append zeros to the left until the 8 bits are completed. For example:

$110100_2 \equiv 00110100_2$ (8-bit number)
$1111011_2 \equiv 01111011_2$ (8-bit number)

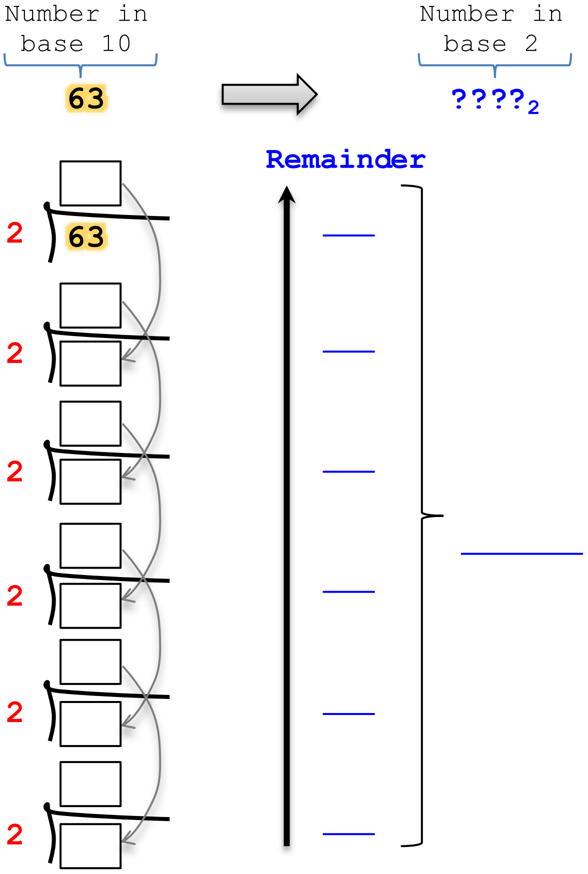
- Actually, you can use this method to convert a decimal number into any other base. For example, if you want to convert it into a base-8 number, just divide by 8 and group the remainders.

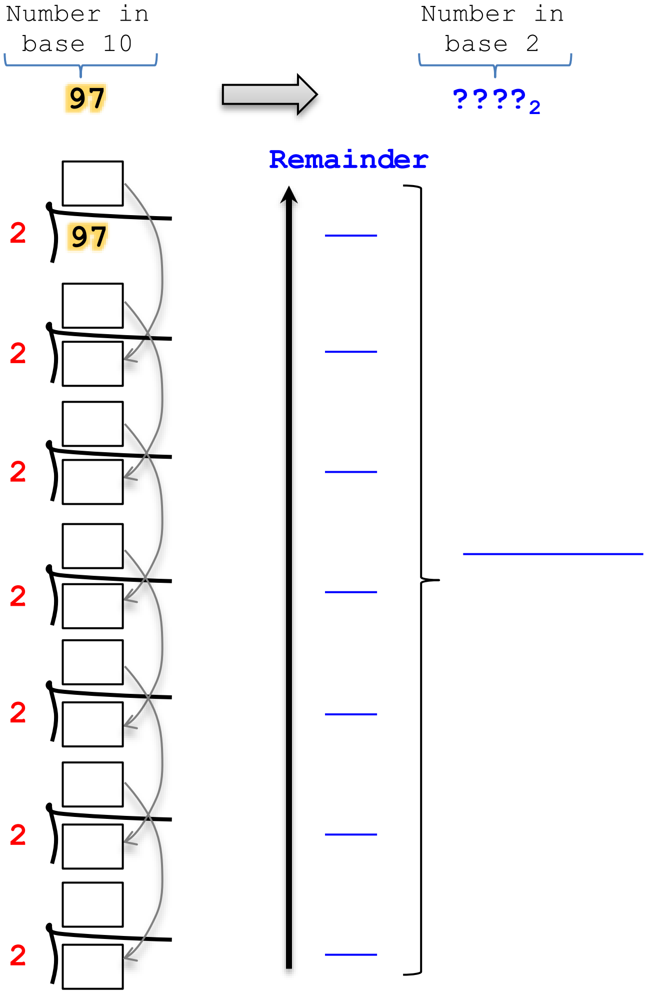
- **Example**: Converting a decimal number to base-8:



Exercise:

- Convert the following two decimal numbers to binary numbers. Fill in the blanks in the figure below.



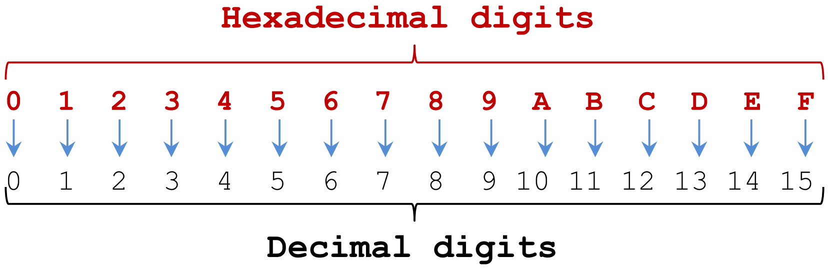


- Now, convert the following decimal numbers to binary numbers. The final binary number must have 8 bits (append zeros to the left to complete).

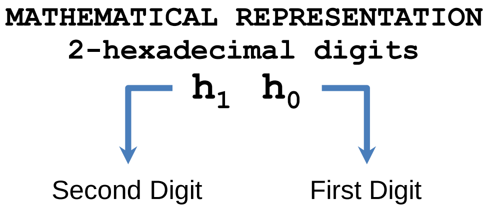
Decimal number	Binary number with 8 bits $b_7b_6b_5b_4b_3b_2b_1b_0$
40	
255	
111	
126	
9	

HEXADECIMAL NUMBER SYSTEM

- This is a very useful system as it is a short-hand notation for binary numbers
- In the decimal number system, a digit can take a value from 0 to 9.
- 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 letter (A-F):



- The following figure shows a 2-digit hexadecimal number. Note that the positional representation with powers of 16 let us obtain the decimal value (*integer positive*) of the hexadecimal number. This is the same as converting a hexadecimal number into a decimal number.



Hex. number h_1h_0	Powers of 16: $h_1\times16^1 + h_0\times16^0$	Decimal Value
5A	$5\times16^1 + A\times16^0$	**90**
10	$1\times16^1 + 0\times16^0$	**16**
08	$0\times16^1 + 8\times16^0$	**8**
FB	$F\times16^1 + B\times16^0$	**251**
3E	$3\times16^1 + E\times16^0$	**62**
A7	$A\times16^1 + 7\times16^0$	**167**

- Note that when we use the letters A-F in the multiplications inside the powers of 16 representation (e.g., $A\times16^1 + 7\times16^0$), we need to replace the hexadecimal symbol by its decimal value.

A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

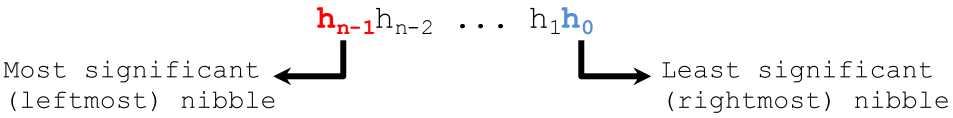
For example: $A\times16^1 \equiv (10)\times16^1$.

EXERCISE: Convert the following hexadecimal numbers (positive integers) to their decimal values:

2-hex. digit representation	$h_1\times16^1 + h_0\times16^0$	Decimal Number
AB		
CE		
05		
70		
F0		
E9		

General case:

- Positional number representation for a hexadecimal number with 'n' nibbles (hexadecimal digits):



- To convert a hexadecimal number into a decimal, we apply the following formula:
  Decimal Value (*integer positive*):

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

- To avoid confusion, it is sometimes customary to append the prefix '0x' to a hexadecimal number:

$$0\times h_{n-1} h_{n-2} \ldots h_1 h_0$$

- Examples: $\texttt{FD0A90: 0xFD0A90} \equiv F\times 16^5 + D\times 16^4 + 0\times 16^3 + A\times 16^2 + 9\times 16^1 + 0\times 16^0$
  $\texttt{0B871C: 0x0B871C} \equiv 0\times 16^5 + B\times 16^4 + 8\times 16^3 + 7\times 16^2 + 1\times 16^1 + C\times 16^0$

- The table presents the maximum attainable value for the given number of nibbles (hexadecimal digits). What pattern do you find? Can you complete it for the highlighted cases (4 and 6)?

Number of nibbles	Maximum value	Range
1	F ≡ 16^1-1	0 → F ≡ 0 → 16^1-1
2	FF ≡ 16^2-1	0 → FF ≡ 0 → 16^2-1
3	FFF ≡ 16^3-1	0 → FFF ≡ 0 → 16^3-1
4		
5	FFFFF ≡ 16^5-1	0 → 11111_2 ≡ 0 → 16^5-1
6		
…		
n	FFF…FFF ≡ 16^n-1	0 → FFF…FFF ≡ 0 → 16^n-1

- **Maximum value for 'n' nibbles:** The maximum decimal value with 'n' nibbles is given by:

$$D = \underbrace{FFF...FFF}_{n\ nibbles} = F\times 16^{n-1} + F\times 16^{n-2} + \ldots + F\times 16^1 + F\times 16^0$$
$$15\times 16^{n-1} + 15\times 16^{n-2} + \ldots + 15\times 16^1 + 15\times 16^0 = 16^n - 1$$

⇒ With 'n' nibbles, we can represent positive integer numbers from 0 to 16^n-1. (16^n numbers)

UNITS OF INFORMATION

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

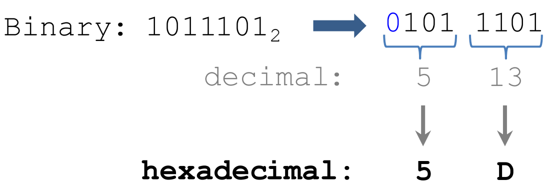
RELATIONSHIP BETWEEN HEXADECIMAL AND BINARY NUMBERS

- Conversions between hexadecimal and binary systems are very common when dealing with digital computers. In this activity, we will learn how these 2 systems are related and how easy it is to convert between one and the other.
- **Hexadecimal to binary:** We already know how to convert a hexadecimal number into a decimal number. We can then can convert the decimal number into a binary number (using successive divisions).
- **Binary to hexadecimal**: We can first convert the binary number to a decimal number. Then, using an algorithm similar to the one that converts decimals into binary, we can convert our decimal number into a hexadecimal number.

SIMPLE METHOD TO CONVERT HEXADECIMAL TO BINARY NUMBERS AND VICEVERSA

- The previous two conversion processes are too tedious. Fortunately, hexadecimal numbers have an interesting property that allows quick conversion of binary numbers to hexadecimals and viceversa.

- **Binary to hexadecimal**: We group the binary numbers in groups of 4 (starting from the rightmost bit). If the last group of bits does not have four bits, we append zeros to the left. Then, we independently convert each group of 4 bits to its decimal value.
  Notice that 4 bits can only take decimal values between 0 and 2^4-1 ≡ 0 to 15, hence 4 bits represent only one hexadecimal digit. In other words, for each group of 4 bits, there are only 16 possible hexadecimal digits to pick from. The figure below shows an example.



Then: 01011101_2 = 0x5D

Verification:

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

$0x5D = 5\times16^1 + D\times16^0 = 93$

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

- **Exercise**: Group the following binary numbers in groups of 4 bits and obtain the hexadecimal representation. Use the table in the previous figure to pick the correspondent hexadecimal digit for each group of 4 bits.

Binary number	Hexadecimal number
1101111	
101	
1110	
1100011	
11111110	
100001	

- **Hexadecimal to binary**: It is basically the inverse of the process of converting a binary into a hexadecimal numbers. We pick each hexadecimal digit and convert it (always using 4 bits) to its 4-bit binary representation. The binary number is the concatenation of all resulting group of 4 bits.

$0xFA = 11111010_2$
$0xC1 = 11000001_2$

DO NOT discard these zeros when concatening!

Exercise:

- Convert these hexadecimal numbers to binary. Verify it by converting both the binary and hexadecimal number (they should match).

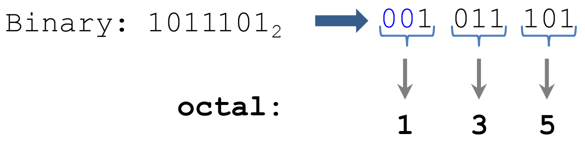
Hexadecimal number	Binary number	Decimal value
A10		
891		
43		
A2		
FACE		

- The reason hexadecimal numbers are popular is because hexadecimal numbers provide a short-hand notation for binary numbers.

OCTAL NUMBERS

- An octal digit can takes between 0 and 7. This is another common number system in computers is base-8 (octal). 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	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

$\boxed{\text{Then: } 01011101_2 = 135_8}$

Verification:

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

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

OCTAL TO BINARY

DO NOT discard these zeros when concatening!

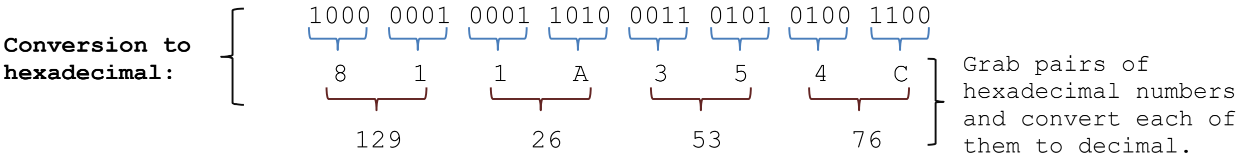
$75_8 = 111101_2$
$31_8 = 011001_2$

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`



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

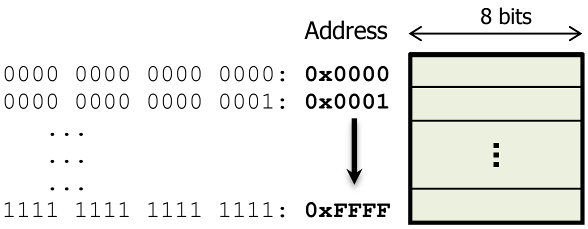
REPRESENTING GRAYSCALE PIXELS

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`.



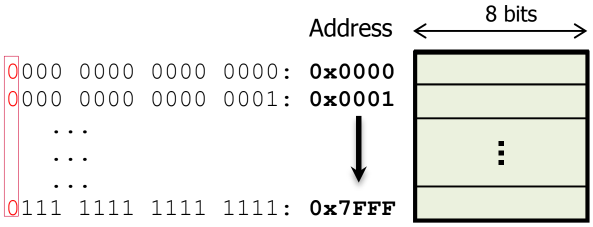
Examples:

- A microprocessor can only handle memory addresses from `0x0000` to `0x7FFF`. What is the address bus size? If each memory position 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.

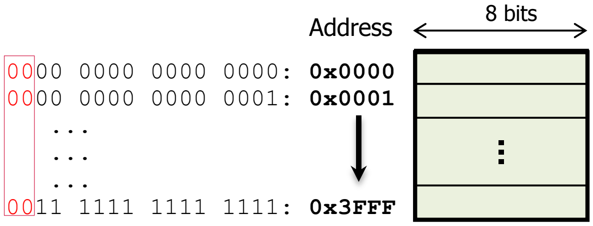


- A microprocessor can only handle memory addresses from `0x0000` to `0x3FFF`. What is the address bus size? If each memory position is one byte wide, what is the maximum size (in bytes) of the memory that we can connect?

  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**.

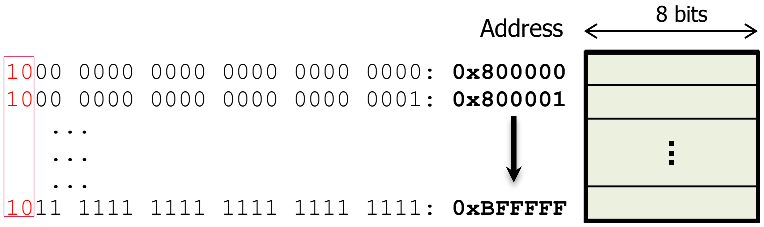
  We can handle $2^{14} bytes = 16KB$ of memory.



- 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 each memory position 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.

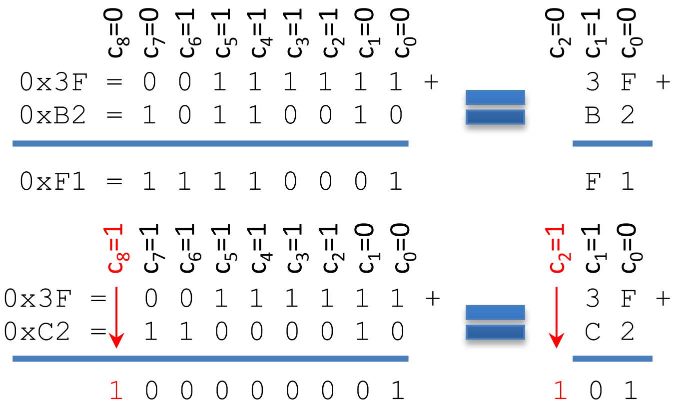
  For a 20-bit address: we have 5 hexadecimal digits that go from `0x00000` to `0xFFFFF`.

  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.

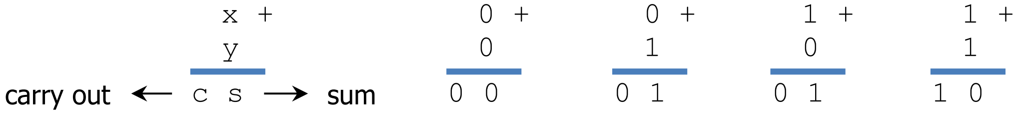
UNSIGNED NUMBERS - ADDITION

- 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. c_0 is usually zero.
- 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.

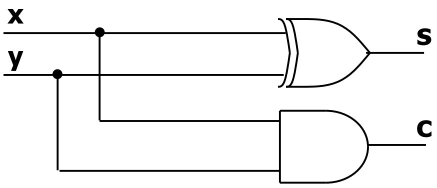


DIGITAL CIRCUIT

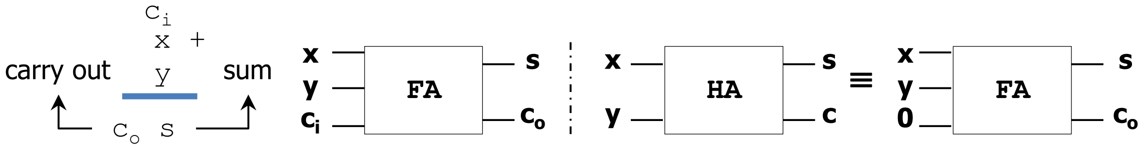
- **1-bit Addition**:
  - ✓ Addition of a bit with carry in: The circuit that performs this operation is called Half Adder (HA).



x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

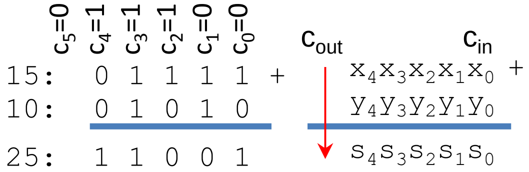


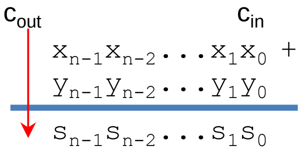
  - ✓ Addition of a bit with carry in: The circuit that performs this operation is called Full Adder (FA).

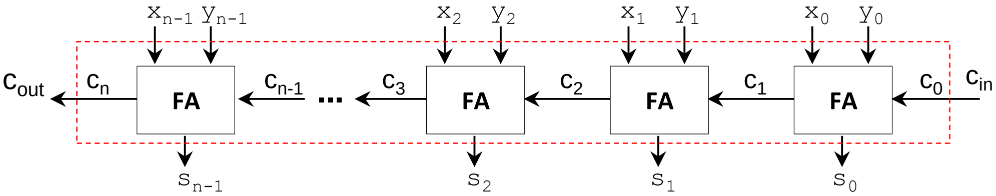


- **n-bit Addition**:
  The figure on the right shows a 5-bit addition. Using the truth table method, we would need 11 inputs and 6 outputs. This is not practical! Instead, it is better to build a cascade of Full Adders.

  For an n-bit addition, the circuit will be:

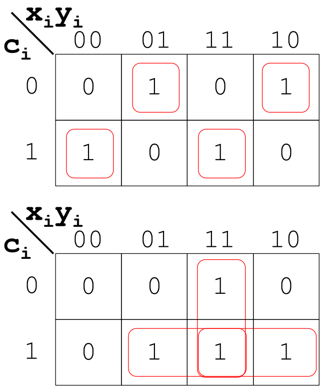






Full Adder Design

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



c_i \ x_iy_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$s_i = \overline{x_i}y_i\overline{c_i} + x_i\overline{y_i}\,\overline{c_i} + \overline{x_i}\,\overline{y_i}c_i + x_iy_ic_i$$

$$s_i = (x_i \oplus y_i)\overline{c_i} + \overline{(x_i \oplus y_i)}c_i$$

$$s_i = x_i \oplus y_i \oplus c_i$$

c_i \ x_iy_i	00	01	11	10
0	0	0	1	0
1	0	1	1	1

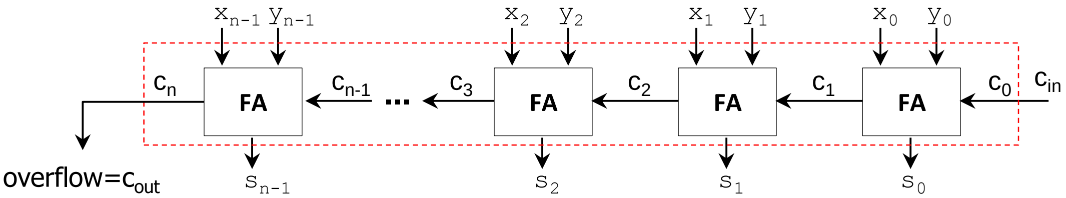
$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

ARITHMETIC OVERFLOW:

- Suppose we have only 4 bits to represent binary numbers:

$$\text{cout=0}\quad \begin{array}{r} 0101\ + \\ 1001 \\ \hline 1110 \end{array} \qquad\qquad \begin{array}{c}\text{cout=1}\\ \Rightarrow \text{OVERFLOW!!!}\end{array} \begin{array}{r} 1011\ + \\ 0110 \\ \hline 10001 \end{array}$$

- Overflow occurs when an arithmetic operation require more bits than the bits we are using to represent our numbers. For 4 bits, the range is 0 to 15. If the summation is greater than 15, then there is overflow.
- For n bits, overflow occurs when the sum is greater than $2^n - 1$. Also, for unsigned numbers: overflow bit = c_{out}.



- To avoid overflow in addition operation, a common technique is to sign-extend the two summands. For example, if the two summands are 4-bits wide, then we add one more bit. So, we use 5 bits to represent our numbers. In the case of unsigned numbers, sign-extend amounts to zero-extend.
- In general, if the two summands are n-bits wide, the result will have at most $n+1$ bits.

$$\begin{array}{r} 01011\ + \\ 00110 \\ \hline 10001 \end{array} \qquad \text{cout=0}$$

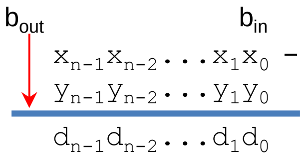
UNSIGNED NUMBERS - SUBTRACTION

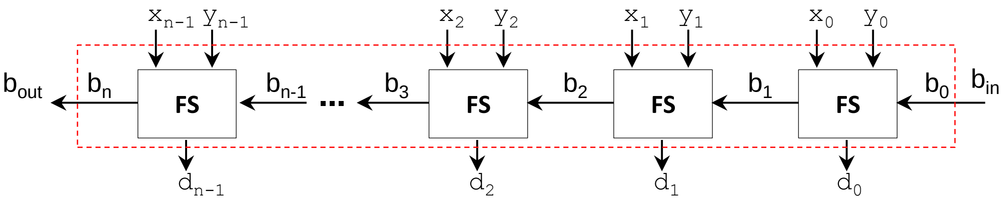
- In the example, we subtract two 8-bit numbers using the binary and hexadecimal (this is a short-hand notation) representations. A subtraction of two digits (binary or hexadecimal) generates a borrow when the difference is negative. So, we borrow 1 from the next digit so that the difference is positive. 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.
- 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.

	$b_8=0$	$b_7=0$	$b_6=0$	$b_5=0$	$b_4=1$	$b_3=1$	$b_2=1$	$b_1=1$	$b_0=0$			$c_2=0$	$c_1=1$	$c_0=0$	
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	

	$b_8=1$	$b_7=1$	$b_6=0$	$b_5=0$	$b_4=0$	$b_3=1$	$b_2=0$	$b_1=1$	$b_0=0$			$b_2=1$	$b_1=0$	$b_0=0$	
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	

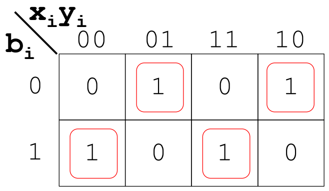
- We can build an **n-bit subtractor** for unsigned numbers using Full Subtractor circuits. In practice, subtraction is better performed in the 2's complement representation (for signed numbers).





Full Subtractor Design

x_i	y_i	b_i	b_{i+1}	d_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

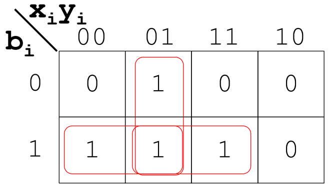


b_i \ x_iy_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$d_i = \overline{x_i}\,y_i\,\overline{b_i} + x_i\,\overline{y_i}\,\overline{b_i} + \overline{x_i}\,\overline{y_i}\,b_i + x_i y_i b_i$$

$$d_i = (x_i \oplus y_i)\overline{b_i} + \overline{(x_i \oplus y_i)}\,b_i$$

$$d_i = x_i \oplus y_i \oplus b_i$$



b_i \ x_iy_i	00	01	11	10
0	0	1	0	0
1	1	1	1	0

$$b_{i+1} = \overline{x_i}\,y_i + \overline{x_i}\,b_i + y_i b_i$$

SIGNED INTEGER NUMBERS

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

SIGN-AND-MAGNITUDE (SM):

- Here, the sign and the magnitude (value) are represented separately. The MSB only represents the sign and the remaining $n-1$ bits the magnitude.
- **Example** (n=4): `0110 = +6` `1110 = -6`

1'S COMPLEMENT (1C):

- Here, if the MSB=0, the number is positive and the remaining $n-1$ bits represent the magnitude. If the MSB=1, the number is negative and the remaining $n-1$ bits do not represent the magnitude. To invert the sign of a number in 1's complement representation, we apply the 1's complement operation to the number, which consists of inverting all the bits.
  - ✓ Let $B = b_{n-1}\dots b_1b_0$ be a number represented in 1's complement. Let $K = k_{n-1}\dots k_1k_0$ represent $-B$. We get K by applying the 1's complement operation to B. K is also called the 1's complement of B (and viceversa).
  - ✓ **Definition:** The 1's complement of B is defined as $K = (2^n - 1) - B$, $n = \#\, of\ bits\ (including\ sign\ bit)$, where $K = \sum_{i=0}^{n-1} k_i 2^i$ and $B = \sum_{i=0}^{n-1} b_i 2^i$. Note that K and B are treated as unsigned numbers in this formula. And $(2^n - 1)$ is the largest n-bit unsigned number. We can then show that the 1's complement operation amounts to inverting all the bits:

    $$\sum_{i=0}^{n-1} k_i 2^i = (2^n - 1) - \sum_{i=0}^{n-1} b_i 2^i \rightarrow \sum_{i=0}^{n-1} (k_i + b_i) 2^i = 2^n - 1 \rightarrow k_i + b_i = 1, \forall i \Rightarrow k_i = \overline{b_i}$$

    Example: Given $B = 01001_2 = 9$ in 1's complement, get the 1's complement representation of -9 using the formula:
    $\rightarrow K = (2^n - 1) - B = (2^5 - 1) - 9 = 22 = 10110_2$.
    Recall that the formula treats K and B as unsigned integers. So, $K = 22$ in unsigned representation, and $K = -9$ in 1's complement representation.
    It is much simpler to just invert each bit!

- With n bits, we can represent $2^n - 1$ numbers from $-2^{n-1} + 1\ to\ 2^{n-1} - 1$. When using the 1's complement representation, it is mandatory to specify how many bits we are using.

- **Examples:**
  - ✓ `+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`
  - ✓ Get the 1's complement representation of 8: This is a positive number, thus the MSB=0. The remaining $n-1$ bits represent the magnitude. The magnitude is represented with a minimum number of 4 bits as $8=1000_2$. Thus, using a minimum number of 5 bits, the number 8 in 1's complement representation is $8=01000_2$.
  - ✓ 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):

- Here, if the MSB=0, the remaining $n-1$ bits represent the magnitude. If the MSB=1, the number is negative and the remaining $n-1$ bits do not represent the magnitude. 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.
  - ✓ Let $B = b_{n-1}\dots b_1b_0$ be a number represented in 2s complement. Let $K = k_{n-1}\dots k_1k_0$ represent $-B$. We get K by applying the 2's complement operation to B. K is also called the 2's complement of B (and viceversa).
  - ✓ **Definition:** The 2's complement of B is defined as $K = (2^n - 1) - B + 1$, $n = \#\, of\ bits\ (including\ sign\ bit)$, where $K = \sum_{i=0}^{n-1} k_i 2^i$ and $B = \sum_{i=0}^{n-1} b_i 2^i$. Note that K and B are treated as unsigned numbers in this formula. We can see that we can first get the term $(2^n - 1) - B$ by inverting all the bits; then we add 1 to complete the equation.

    Example: Given $= 01001_2 = 9$ in 2's complement, get the 2's complement representation of -9 using the formula:
    $\rightarrow K = (2^n - 1) - B + 1 = (2^5 - 1) - 9 + 1 = 23 = 10111_2$.
    Recall that the formula treats K and B as unsigned integers. So, $K = 23$ in unsigned representation, and $K = -9$ in 2's complement representation.
    It is much simpler to just invert each bit (i.e., apply 1's complement operation) and then add 1!

- With n bits, we can represent 2^n numbers from $-2^{n-1}\ to\ 2^{n-1} - 1$. When using the 2's complement representation, it is mandatory to specify how many bits we are using.

- **Examples**:
  - ✓ `+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`
  - ✓ Represent 12 in 2's complement: This is a positive number, → MSB=0. The remaining $n-1$ bits represent the magnitude. We can get the magnitude with a minimum 4 bits: `12=1100`$_2$. Thus, using a minimum of 5 bits, the number 12 in 2's complement representation is `12=01100`$_2$.
  - ✓ What is the decimal value of `1101`? We first apply the 2's complement *operation* (or take the 2's complement) 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`.

Getting the decimal value of a number in 2's complement representation:

- If the number B is positive, then MSB=0: $b_{n-1}=0$.

$$\rightarrow B=\sum_{i=0}^{n-1} b_i 2^i = b_{n-1}2^{n-1}+\sum_{i=0}^{n-2} b_i 2^i = \sum_{i=0}^{n-2} b_i 2^i \quad (a)$$

- If the number B is negative, $b_{n-1}=1$ (MSB=1). If we take the 2's complement of B, we get K (which is a positive number). In 2's complement representation, K *represents* $-B$. Using $K=2^n-B$ (K and B are treated as unsigned numbers):

$$\sum_{i=0}^{n-1} k_i 2^i = 2^n - \sum_{i=0}^{n-1} b_i 2^i$$

- We want a to express $-K$ in terms of b_i, since the integer value $-K$ is the actual integer value of B.

$$-K=-\sum_{i=0}^{n-1} k_i 2^i = \sum_{i=0}^{n-1} b_i 2^i - 2^n = b_{n-1}2^{n-1}+\sum_{i=0}^{n-2} b_i 2^i - 2^n = 2^{n-1}(b_{n-1}-2)+\sum_{i=0}^{n-2} b_i 2^i$$

$$B=-K=2^{n-1}(1-2)+\sum_{i=0}^{n-2} b_i 2^i = -2^{n-1}+\sum_{i=0}^{n-2} b_i 2^i \quad (b)$$

- Using (a) and (b), the formula for the decimal value of B (either positive or negative) is:

$$B=-b_{n-1}2^{n-1}+\sum_{i=0}^{n-2} b_i 2^i$$

- **Examples**:

  $10110_2=-2^4+2^2+2^1=-10$ $\qquad$ $11000_2=-2^4+2^3=-8$

SUMMARY

- 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`.

SIGN EXTENSION

- **UNSIGNED NUMBERS**: Here, if we want to use more bits, we just append zeros to the left.
  Example: $12 = 1100_2$ with 4 bits. If we want to use 6 bits, then $12 = 001100_2$.

- **SIGNED NUMBERS:**
  - ✓ **Sign-and-magnitude**: The MSB only represents the sign. If we want to use more bits, we append zeros to the left. The leftmost bit is always the sign.
    Example: $-12 = 11100_2$ with 5 bits. If we want to use 7 bits, then $-12 = 1001100_2$.

  - ✓ **2's complement** (also applies to 1's complement): In many circumstances, we might want to represent numbers in 2's complement with a fixed number of bits. For example, the following two numbers require a minimum of 5 bits:
    $10111_2 = -2^4 + 2^2 + 2^1 + 2^0 = -9$ $\qquad$ $01111_2 = 2^3 + 2^2 + 2^1 + 2^0 = +15$

    What if we wanted to use 8 bits to represent them? In 2's complement, we need to sign-extend: If the number is positive, we append zeros to the left. If the number is negative, we attach ones to the left. In the example, note how we added three bits to the left in each case:
    $11110111_2 = -2^4 + 2^2 + 2^1 + 2^0 = -9$ $\qquad$ $00001111_2 = 2^3 + 2^2 + 2^1 + 2^0 = +15$

Demonstration of sign-extension in 2's complement:

- To increase the number of bits for representing a number, we append the MSB to the left as many times as needed:

$$b_{n-1}b_{n-2}\dots b_0 \equiv b_{n-1}\dots b_{n-1}b_{n-1}b_{n-2}\dots b_0$$

Examples: $00101_2 = 00000101_2 = 2^2 + 2^0 = 5$
$10101_2 = 11110101_2 = -2^4 + 2^2 + 2^0 = -2^6 + 2^5 + 2^4 + 2^2 + 2^0 = -11$

We can think of the sign-extended number as an m-bit number, where $m > n$:

$$b_{n-1}\dots b_{n-1}b_{n-1}b_{n-2}\dots b_0 = b_{m-1}\dots b_n b_{n-1}b_{n-2}\dots b_0, \text{ where: } b_i = b_{n-1}, i = n, n+1, \dots, m-1$$

- We need to demonstrate that $b_{n-1}b_{n-2}\dots b_0$ represents the same decimal number as $b_{n-1}\dots b_{n-1}b_{n-1}b_{n-2}\dots b_0$, i.e., that the sign-extension is correct for any $m > n$.
  We need that: $b_{m-1}\dots b_n b_{n-1}b_{n-2}\dots b_0 = b_{n-1}\dots b_{n-1}b_{n-1}b_{n-2}\dots b_0 = b_{n-1}b_{n-2}\dots b_0$

Using the formula for 2's complement numbers:

$$-2^{m-1}b_{m-1} + \sum_{i=0}^{m-2} 2^i b_i = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

$$-2^{m-1}b_{m-1} + \sum_{i=n-1}^{m-2} 2^i b_i + \sum_{i=0}^{n-2} 2^i b_i = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i \Rightarrow -2^{m-1}b_{m-1} + \sum_{i=n-1}^{m-2} 2^i b_i = -2^{n-1}b_{n-1}$$

$$-2^{m-1}b_{n-1} + b_{n-1}\sum_{i=n-1}^{m-2} 2^i = -2^{n-1}b_{n-1},$$

$$Recall: \sum_{i=k}^{l} r^i = \frac{r^k - r^{l+1}}{1-r}, r \neq 1 \rightarrow \sum_{i=k}^{l} 2^i = \frac{2^k - 2^{l+1}}{1-2} = 2^{l+1} - 2^k$$

Then:

$$-2^{m-1}b_{n-1} + b_{n-1}(2^{m-1} - 2^{n-1}) = -2^{n-1}b_{n-1}$$

$$-2^{m-1}b_{n-1} + 2^{m-1}b_{n-1} - 2^{n-1}b_{n-1} = -2^{n-1}b_{n-1} \therefore -2^{n-1}b_{n-1} = -2^{n-1}b_{n-1}$$

SIGNED NUMBERS – ADDITION AND SUBTRACTION

- We will use the 2's complement representation for signed numbers.
- 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 4-bit signed numbers. Note that the *carry out* bit does not necessarily indicate overflow. In some cases, the carry out must be ignored, otherwise the result is incorrect.

$$\begin{array}{l} +5 = 0101\ + \\ +2 = 0010 \\ \hline +7 = 0111 \end{array} \qquad \begin{array}{l} -5 = 1011\ + \\ +2 = 0010 \\ \hline -3 = 1101 \end{array} \qquad \begin{array}{l} +5 = 0101\ + \\ -2 = 1110 \\ \hline +3 = \cancel{1}0011 \end{array} \qquad \begin{array}{l} -5 = 1011\ + \\ -2 = 1110 \\ \hline -7 = \cancel{1}1001 \end{array}$$

cout=0 $\qquad$ cout=0 $\qquad$ cout=1 $\qquad$ cout=1

- Now, 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: $[-2^7, 2^7-1]$.

$$\begin{array}{rcccccccccccc}
 & & c_8=0 & c_7=1 & c_6=0 & c_5=1 & c_4=1 & c_3=1 & c_2=0 & c_1=0 & c_0=0 & \\
+92 & = & & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & + \\
+78 & = & & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & \\ \hline
+170 & = & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 &
\end{array}$$

overflow = $c_8 \oplus c_7=1$ -> overflow!

$+170 \notin [-2^7, 2^7-1]$ -> overflow!

$$\begin{array}{rcccccccccccc}
 & & c_8=1 & c_7=0 & c_6=1 & c_5=0 & c_4=0 & c_3=0 & c_2=0 & c_1=0 & c_0=0 & \\
-92 & = & & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & + \\
-78 & = & & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & \\ \hline
-170 & = & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 &
\end{array}$$

overflow = $c_8 \oplus c_7=1$ -> overflow!

$-170 \notin [-2^7, 2^7-1]$ -> overflow!

$$\begin{array}{rcccccccccccc}
 & & c_8=1 & c_7=1 & c_6=1 & c_5=1 & c_4=0 & c_3=0 & c_2=0 & c_1=0 & c_0=0 & \\
+92 & = & & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & + \\
-78 & = & & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & \\ \hline
+14 & = & \cancel{1} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 &
\end{array}$$

overflow = $c_8 \oplus c_7=0$ -> no overflow

$+14 \in [-2^7, 2^7-1]$ -> no overflow

$$\begin{array}{rcccccccccccc}
 & & c_8=0 & c_7=0 & c_6=0 & c_5=0 & c_4=1 & c_3=1 & c_2=0 & c_1=0 & c_0=0 & \\
-92 & = & & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & + \\
+78 & = & & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & \\ \hline
-14 & = & \cancel{0} & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 &
\end{array}$$

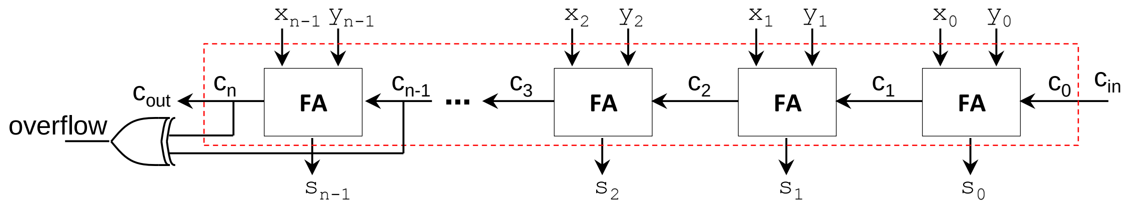
overflow = $c_8 \oplus c_7=0$ -> no overflow

$-14 \in [-2^7, 2^7-1]$ -> 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 $overflow = c_n \oplus c_{n-1}$. Also, $c_{out} = c_n$.

DIGITAL CIRCUIT

- The figure depicts a n-bit adder for 2's complement numbers:



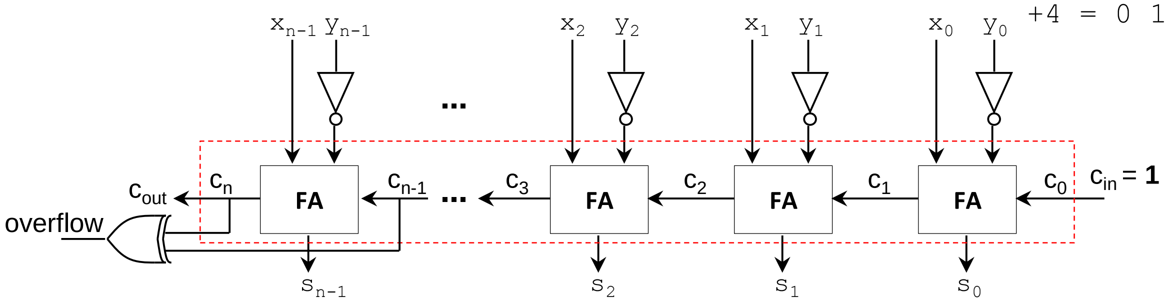
- **Subtraction**: Note that $A-B=A+2C(B)$. To subtract two numbers represented in 2's complement arithmetic, we first apply the 2's complement operation to B (the subtrahend), and then add the numbers. So, in 2's complement arithmetic, subtraction is actually an addition of two numbers.
- The digital circuit for 2's complement subtraction is based on the adder. We account for the 2's complement operation for the subtrahend by inverting every bit in the subtrahend and by making the c_{in} bit equal to 1.

7 - 3 = 7 + (-3):

$+3=0011 \rightarrow -3=1101$

cout = 1
overflow = 0

$$\begin{array}{rccccccc}
 & & c_4=1 & c_3=1 & c_2=1 & c_1=1 & c_0=0 & \\
+7 & = & & 0 & 1 & 1 & 1 & + \\
-3 & = & & 1 & 1 & 0 & 1 & \\ \hline
+4 & = & & 0 & 1 & 0 & 0 &
\end{array}$$



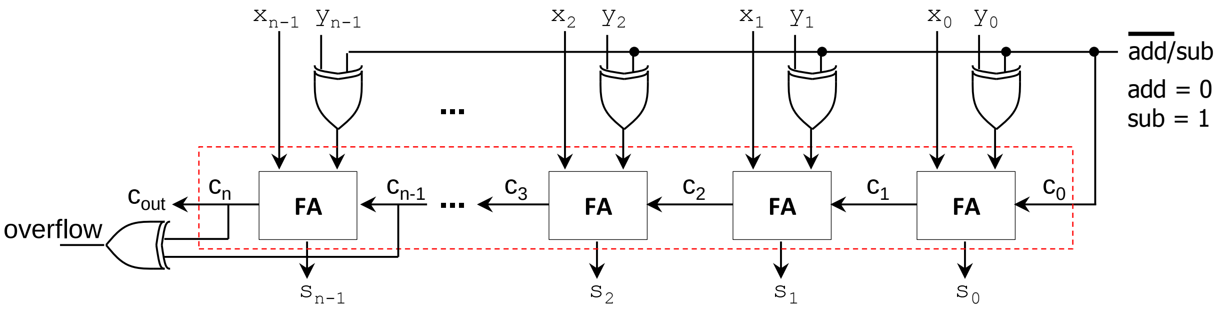
- To avoid overflow when adding/subtracting, a common technique is to sign-extend the two summands. For example, for two 4-bits summands, we add an extra bit; so, we use 5 bits to represent our numbers.
- In general, if the two summands are n-bits wide, the result will have at most $n+1$ bits.
- Recall that if there is no overflow in a summation result, the carry out bit must not be part of the result.

$$\begin{array}{rcccccccc}
 & & c_5=0 & c_4=0 & c_3=1 & c_2=1 & c_1=0 & c_0=0 & \\
+7 & = & & 0 & 0 & 1 & 1 & 1 & + \\
+2 & = & & 0 & 0 & 0 & 1 & 0 & \\ \hline
+9 & = & & 0 & 1 & 0 & 0 & 1 &
\end{array}$$

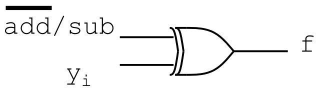
$$\begin{array}{rcccccccc}
 & & c_5=1 & c_4=1 & c_3=0 & c_2=0 & c_1=0 & c_0=0 & \\
-7 & = & & 1 & 1 & 0 & 0 & 1 & + \\
-2 & = & & 1 & 1 & 1 & 1 & 0 & \\ \hline
-9 & = & & 1 & 0 & 1 & 1 & 1 &
\end{array}$$

Adder/Subtractor Unit for 2's complement numbers:

- We can combine the adder and subtractor in a single circuit if we are willing to give up the input `cin`.



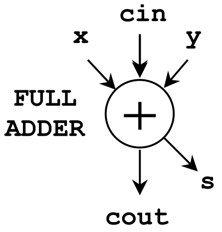
$\overline{add}/sub$	y_i	f
0	0	0
0	1	1
1	0	1
1	1	0

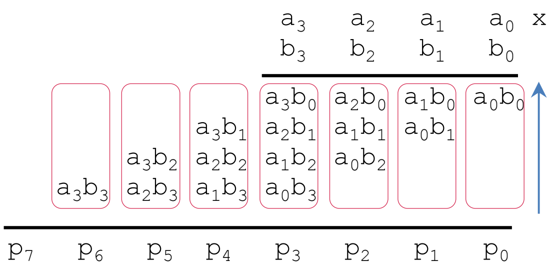


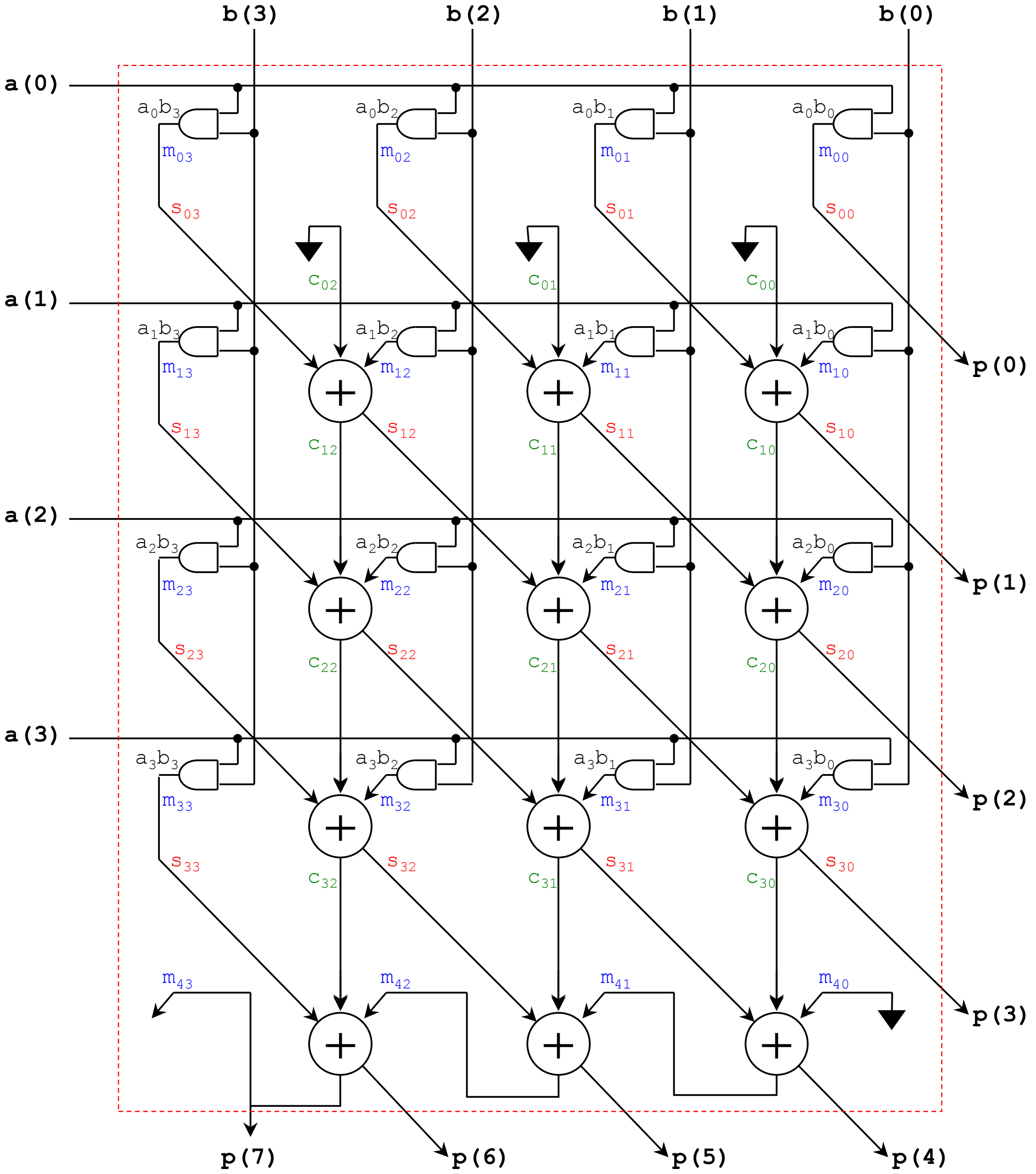
MULTIPLICATION OF INTEGER NUMBERS

ARRAY MULTIPLIER FOR UNSIGNED NUMBERS

- A straightforward implementation of the multiplication operation is depicted in the figure: at every diagonal of the circuit, we add up all terms in a column of the multiplication.
- The figure shows the process and circuit for multiplying two unsigned numbers of 4 bits.







MULTIPLICATION OF SIGNED NUMBERS

- A straightforward implementation consists of checking the sign of the multiplicand and multiplier. If one or both are negative, we change the sign by applying the 2's complement operation. This way, we are left with unsigned multiplication.
- As for the final output: if only one of the inputs was negative, then we modify the sign of the output. Otherwise, the result of the unsigned multiplication is the final output.

```
1 0 1 x ➡ 0 1 1 x
0 1 0     0 1 0
          -----
          0 0 0
        0 1 1
      0 0 0
    -----------
    0 0 0 1 1 0
         ⬇
    1 1 1 0 1 0
```

```
0 1 0 x ➡ 0 1 0 x
1 1 0     0 1 0
          -----
          0 0 0
        0 1 0
      0 0 0
    -----------
    0 0 0 1 0 0
         ⬇
    1 1 1 1 0 0
```

```
1 1 1 x ➡ 0 0 1 x
1 1 0     0 1 0
          -----
          0 0 0
        0 1 1
      0 0 0
    -----------
    0 0 0 1 1 0
```

```
    0 1 1 x
    0 1 0
    -----
    0 0 0
  0 1 1
0 0 0
-----------
0 0 0 1 1 0
```

- **Note**: If one of the inputs is -2^{n-1}, then the 2's complement of it is 2^{n-1}, which requires $n+1$ bits. Here, we are allowed to use only n bits; in other words, we do not have to change its sign.
  This will not affect the final result since if we were to use $n+1$ bits for 2^{n-1}, the MSB=0, which implies that the last row is full of zeros.

```
1 0 0 x ➡ 1 0 0 x
0 1 1     0 1 1
          -----
          1 0 0
        1 0 0
      0 0 0
    -----------
    0 0 1 1 0 0
         ⬇
    1 1 0 1 0 0
```

```
0 1 1 x ➡ 0 1 1 x
1 0 0     1 0 0
          -----
          0 0 0
        0 0 0
      0 1 1
    -----------
    0 0 1 1 0 0
         ⬇
    1 1 0 1 0 0
```

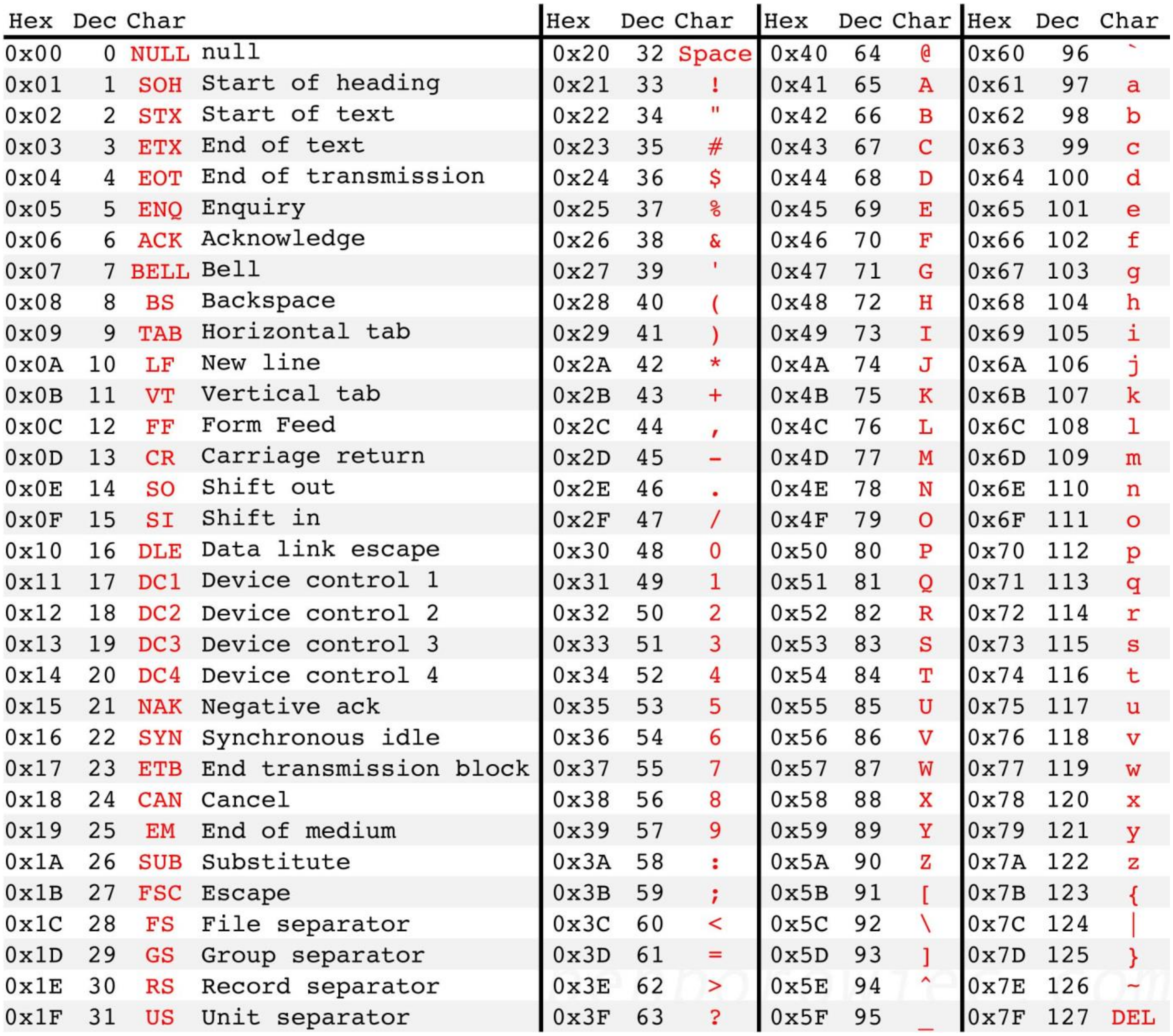
```
1 0 0 x ➡ 1 0 0 x
1 0 0     1 0 0
          -----
          0 0 0
        0 0 0
      1 0 0
    -----------
    0 1 0 0 0 0
```

- Final output: It requires $2n$ bits. Note that it is only because of the multiplication of -2^{n-1} by -2^{n-1} that we require those $2n$ bits (in 2's complement representation)

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 also represent 2^n numbers in any range.
- Moreover, with n bits we can represent 2^n different symbols. For example, in 24-bit color, each color is represented by 24 bits, providing 2^{24} distinct colors. Each color is said to have a binary code.
- N = 5 symbols. With 2 bits, only 4 symbols can be represented. With 3 bits, 8 symbols can be represented. Thus, the number of bits required is $n = 3 = \lceil log_2 5 \rceil = log_2 8$. Note that 8 is the power of 2 closest to N=5 that is greater than or equal to 5.
- In general, if we have N symbols to represent, the number of bits required is given by $\lceil \log_2 N \rceil$. For example:
  - ✓ Minimum number of bits to represent 70,000 colors: → Number of bits: $\lceil \log_2 70000 \rceil = 17\ bits$.
  - ✓ Minimum number of bits to represent numbers between 15,000 and 19,096: → There are 19,096-15,000+1=4097. Then, number of bits: $\lceil \log_2 4097 \rceil = 13\ bits$.

7-bit US-ASCII character-encoding scheme: Each character is represented by 7 bits. Thus, the number of characters that can be represented is given by $2^7 = 128$. Each character 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: This code 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. Each digit requires 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, notice that the binary numbers 1011_2(10) to 1111_2(15) are not used. Only 10 out of 16 values are used to encode each decimal digit.
- **Examples:**
  - ✓ Decimal number **47**: This decimal number can be represented as a binary number: 101111_2. In BCD format, this would be: **0100 0111**$_2$
  - ✓ Decimal number **58**: This decimal number can be represented as a binary number: 111010_2. In BCD format, the binary representation would be: **01011000**$_2$
  - ✓ The BCD code is not the same as the binary number!

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

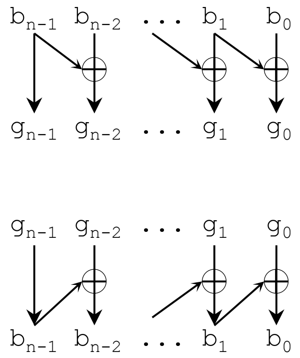
- There exist many other binary codes (e.g., reflective gray code, 6-3-1-1 code, 2-out-of-5 code) to represent decimal numbers. Usually, each of them is tailored to an specific application.

REFLECTIVE GRAY CODE:

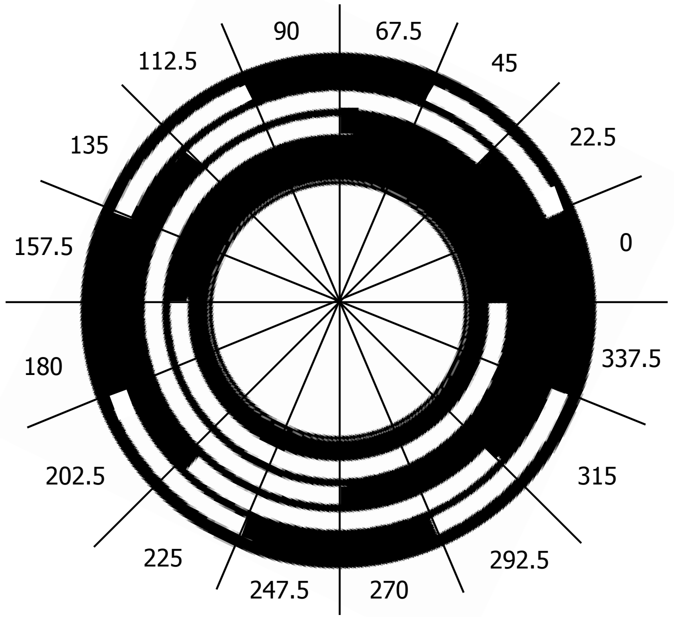
g_1g_0
0 0
0 1
1 1
1 0

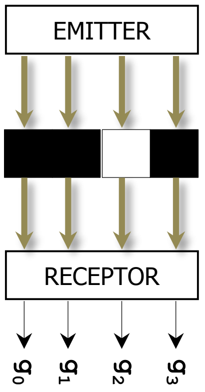
Decimal Number	$b_2b_1b_0$	$g_2g_1g_0$
0	0 0 0	0 0 0
1	0 0 1	0 0 1
2	0 1 0	0 1 1
3	0 1 1	0 1 0
4	1 0 0	1 1 0
5	1 0 1	1 1 1
6	1 1 0	1 0 1
7	1 1 1	1 0 0

$g_3g_2g_1g_0$
0 0 0 0
0 0 0 1
0 0 1 1
0 0 1 0
0 1 1 0
0 1 1 1
0 1 0 1
0 1 0 0
1 1 0 0
1 1 0 1
1 1 1 1
1 1 1 0
1 0 1 0
1 0 1 1
1 0 0 1
1 0 0 0



- **Application:** Measuring angular position with 4-bit RGC. 4 beams are emitted along an axis. When a light beam passes (transparent spots, represented as whites), we get a logical 1, 0 otherwise. The RGC encoding makes that between areas only one bit changes, thereby reducing the possibility of an incorrect reading (especially when the beam between adjacent areas). For example: from `0001` to `0011` only one bit flips. If we used `0001` to `0010`, two bits would flip: that would be prone to more errors, especially when the beams are close to the line where the two areas meet.





Angle	$g_3g_2g_1g_0$
0 - 22.5	0 0 0 0
22.5 - 45	0 0 0 1
45 - 67.5	0 0 1 1
67.5 -90	0 0 1 0
90 - 112.5	0 1 1 0
112.5 - 135	0 1 1 1
135 - 157.5	0 1 0 1
157.5 - 180	0 1 0 0
180 - 202.5	1 1 0 0
202.5 - 225	1 1 0 1
225 - 247.5	1 1 1 1
247.5 - 270	1 1 1 0
270 - 292.5	1 0 1 0
292.5 - 315	1 0 1 1
315 - 337.5	1 0 0 1
337.5 - 360	1 0 0 0

INTRODUCTION TO FIXED-POINT ARITHMETIC

- We have been representing positive integer numbers. But what if we wanted to represent numbers with fractional parts?
- Fixed-point arithmetic: Binary representation of positive decimal numbers with fractional parts.

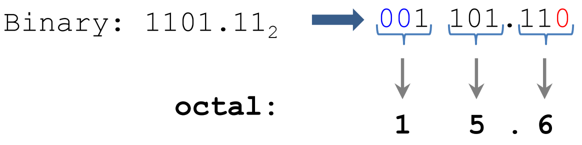
Given the following binary number:

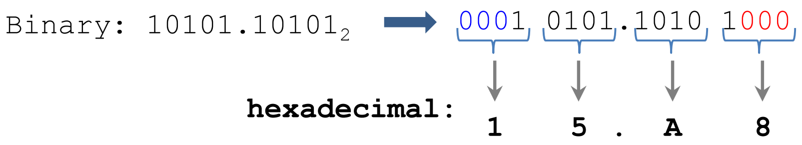
$$(b_{n-1}b_{n-2}\dots b_1b_0.\boldsymbol{b_{-1}b_{-2}\dots b_{-k}})_2$$

Formula to convert it to decimal:

$$D=\sum_{i=-k}^{n-1} b_i\times 2^i = b_{n-1}\times 2^{n-1}+b_{n-2}\times 2^{n-2}+\dots+b_1\times 2^1+b_0\times 2^0+\boldsymbol{b_{-1}}\times\boldsymbol{2^{-1}}+\boldsymbol{b_{-2}}\times\boldsymbol{2^{-2}}+\dots\boldsymbol{b_{-k}}\times\boldsymbol{2^{-k}}$$

Conversion from binary to hexadecimal (or octal): (unsigned numbers)

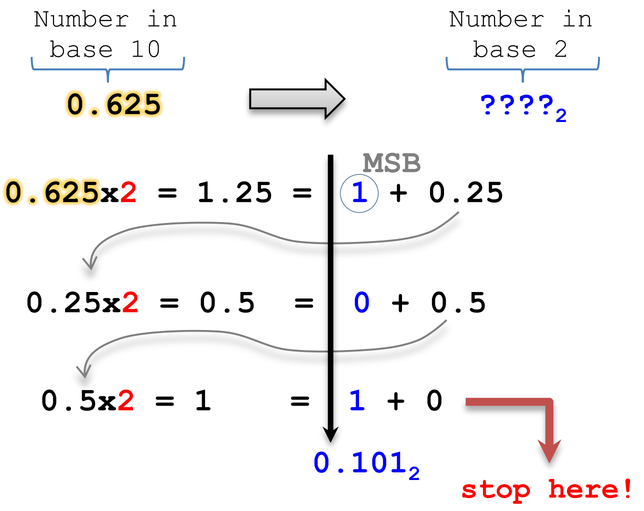


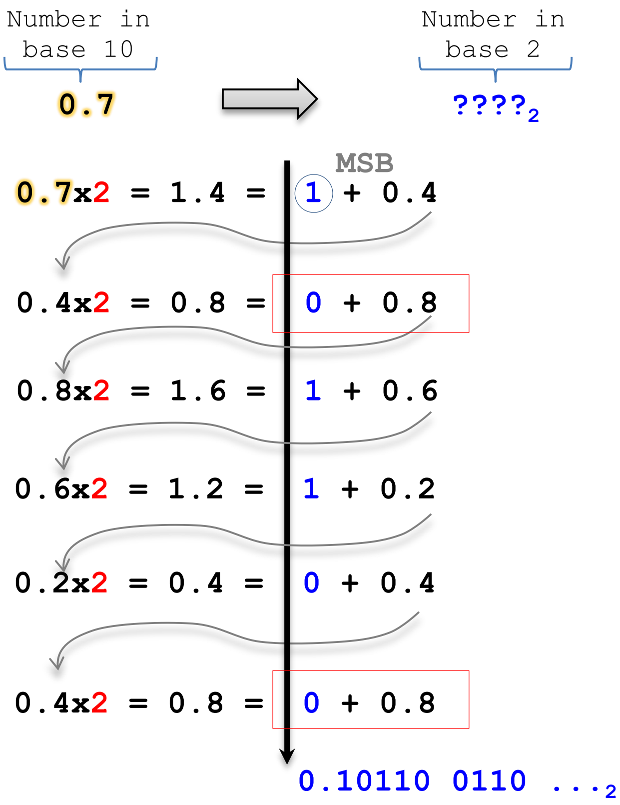


- **Example:** (unsigned number)

$$1011.101_2 = 1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0+1\times\boldsymbol{2^{-1}}+\boldsymbol{0}\times\boldsymbol{2^{-2}}+\boldsymbol{1}\times\boldsymbol{2^{-3}} = 11.625$$

- **Example:** Now, what if we have a decimal number with fractional part? What we do is we divide the integer part and the fractional part. We obtain the binary representation of the integer part using what we know. As for the fractional part, what we do is successive multiplications by 2, the resulting integer parts resulting is the result.





- **Example (signed number):** Convert -379.21875 to the 2's complement representation.
  First, we get the binary representation of +379.21875, and then apply the 2's complement operation to that result.
  - ✓ 379 = 101111011_2. In 2's complement: 379 = 0101111011_2, 0.21875 = 0.00111_2.
  - ✓ Then: 379.21875 = 0101111011.00111_2. This is the 2's complement representation of 379.21875.
  - ✓ Finally, we get -379.2185 by getting the 2's complement of the previous result: -379.21875 = 1010000100.11001_2= `0xE84.C8` (to convert to hexadecimal, we append zeros to the LSB and sign-extend the MSB)