Logic and Computer Design Fundamentals

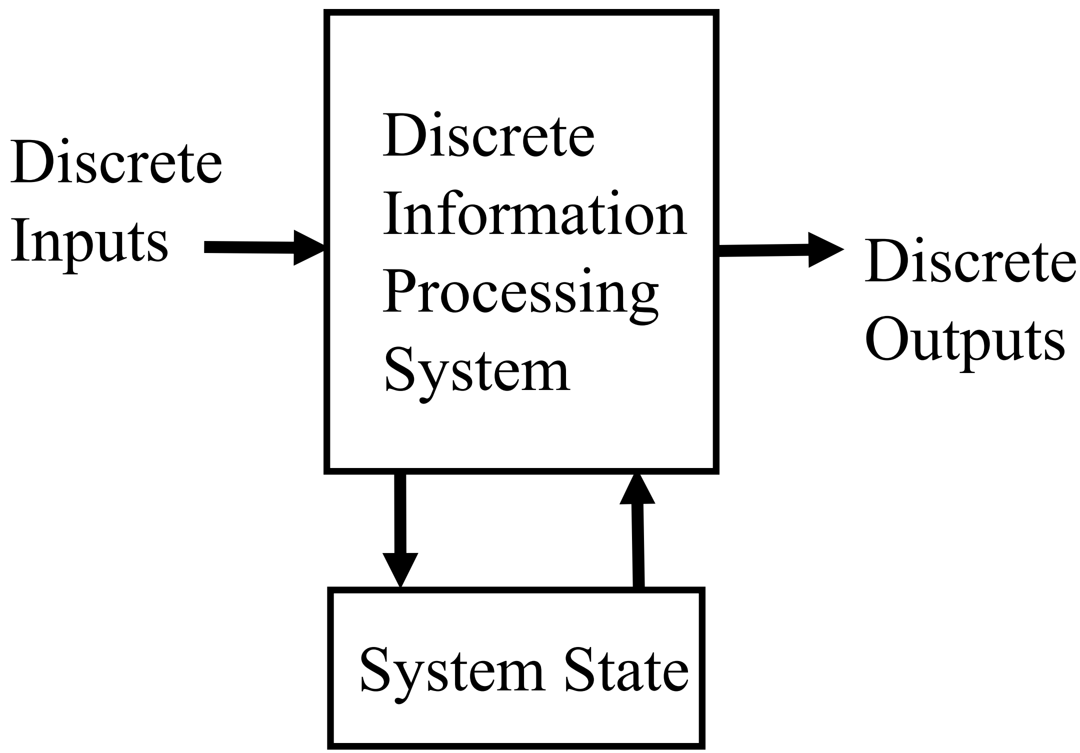
Digital Computers and Information

Overview

- **Digital Systems and Computer Systems**
- **Information Representation**
- **Number Systems** [binary, octal and hexadecimal]
- **Arithmetic Operations**
- **Base Conversion**
- **Decimal Codes** [BCD (binary coded decimal), parity]
- **Gray Codes**
- **Alphanumeric Codes**

Digital System

- **Takes a set of discrete information <u>inputs</u> and discrete internal information <u>(system state)</u> and generates a set of discrete information <u>outputs</u>.**



Types of Digital Systems

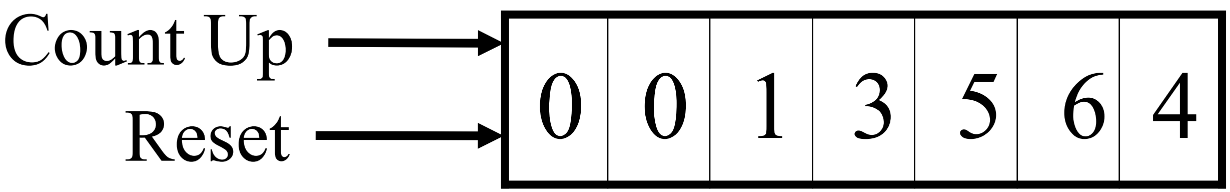
- **No state present**
  - **Combinational Logic System**
  - **Output = Function(Input)**
- **State present**
  - **State updated at discrete times**

    => Synchronous Sequential System
  - **State updated at any time**

    =>Asynchronous Sequential System
  - **State = Function (State, Input)**
  - **Output = Function (State)**
    or Function (State, Input)

Digital System Example:

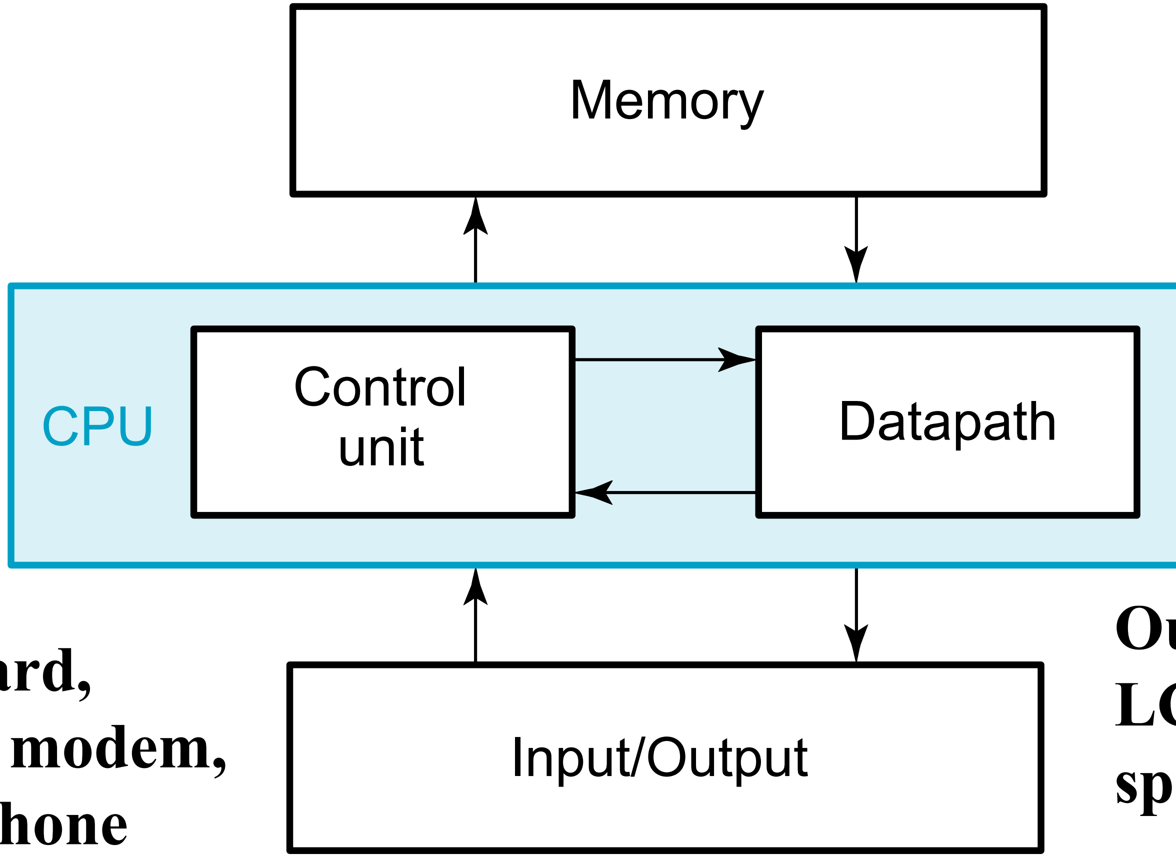
A Digital Counter (e. g., odometer):



Inputs: **Count Up, Reset**
Outputs: **Visual Display**
State: **"Value" of stored digits**

Synchronous or Asynchronous?

A Digital Computer Example



Inputs: Keyboard, mouse, modem, microphone

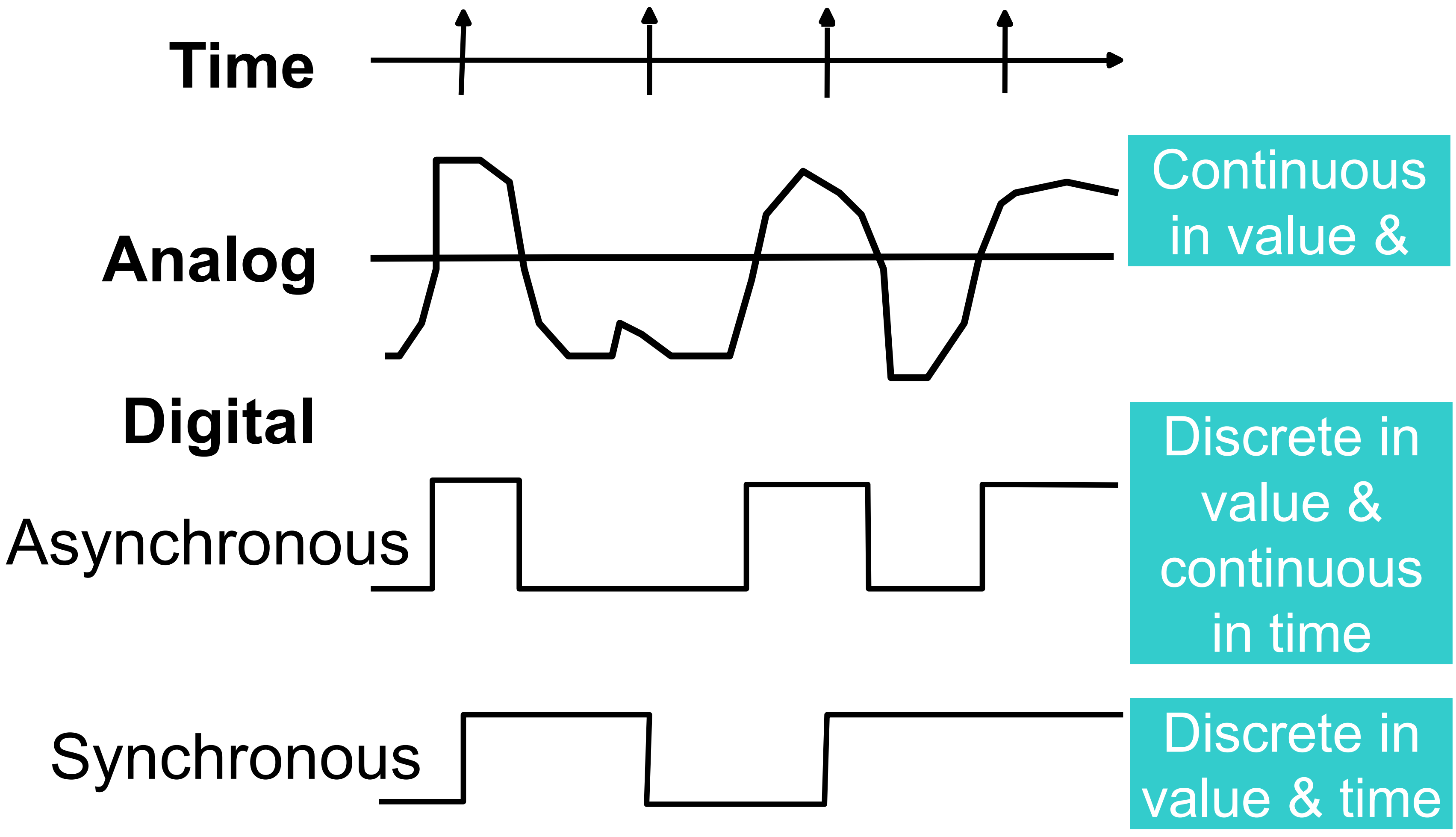
Outputs: CRT, LCD, modem, speakers

Synchronous or Asynchronous?

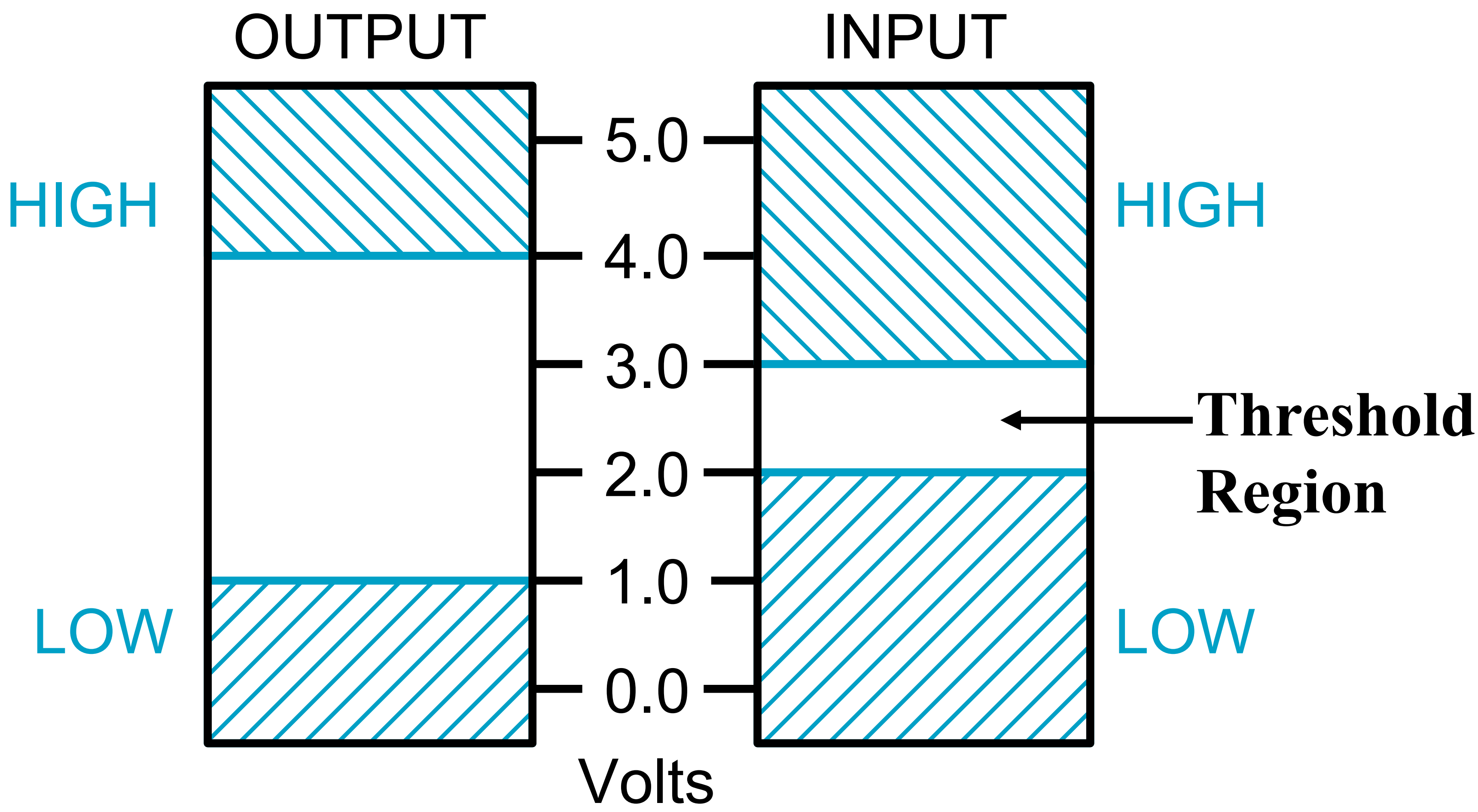
Signal

- **An information variable represented by physical quantity.**
- **For digital systems, the variable takes on discrete values.**
- **Two level, or binary values are the most prevalent values in digital systems.**
- **Binary values are represented abstractly by:**
  - **digits 0 and 1**
  - **words (symbols) False (F) and True (T)**
  - **words (symbols) Low (L) and High (H)**
  - **and words On and Off.**
- **Binary values are represented by values or ranges of values of physical quantities**

Signal Examples Over Time



Signal Example – Physical Quantity: Voltage



Binary Values: Other Physical Quantities

- **What are other physical quantities represent 0 and 1?**
  - **CPU** **Voltage**
  - **Disk** **Magnetic Field Direction**
  - **CD** **Surface Pits/Light**
  - **Dynamic RAM** **Electrical Charge**

Number Systems – Representation

- Positive radix, positional number systems
- A number with *radix* ***r*** is represented by a string of digits:

  $$A_{n-1}A_{n-2}\ \ldots\ A_1A_0\ .\ A_{-1}\ A_{-2}\ \ldots\ A_{-m+1}\ A_{-m}$$

  in which $\mathbf{0} \leq A_i < r$ and **.** is the *radix point*.
- The string of digits represents the power series:

$$(\text{Number})_r = \left(\sum_{i=0}^{i=n-1} A_i \cdot r^i\right) + \left(\sum_{j=-m}^{j=-1} A_j \cdot r^j\right)$$

(Integer Portion) + (Fraction Portion)

Number Systems – Examples

		General	Decimal	Binary
Radix (Base)		r	10	2
Digits		0 => r - 1	0 => 9	0 => 1
	0	r^0	1	1
	1	r^1	10	2
	2	r^2	100	4
	3	r^3	1000	8
Powers of	4	r^4	10,000	16
Radix	5	r^5	100,000	32
	-1	r^{-1}	0.1	0.5
	-2	r^{-2}	0.01	0.25
	-3	r^{-3}	0.001	0.125
	-4	r^{-4}	0.0001	0.0625
	-5	r^{-5}	0.00001	0.03125

Special Powers of 2

- 2^{10} (1024) is Kilo, denoted "K"
- 2^{20} (1,048,576) is Mega, denoted "M"
- 2^{30} (1,073, 741,824)is Giga, denoted "G"

Positive Powers of 2

- Useful for Base Conversion

Exponent	Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Exponent	Value
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536
17	131,072
18	262,144
19	524,288
20	1,048,576
21	2,097,152

Converting Binary to Decimal

- To convert to decimal, use decimal arithmetic to form Σ (digit × respective power of 2).
- Example:Convert 11010_2 to N_{10}:

Converting Decimal to Binary

- **Method 1**
  - **Subtract the largest power of 2 (see slide 14) that gives a positive remainder and record the power.**
  - **Repeat, subtracting from the prior remainder and recording the power, until the remainder is zero.**
  - **Place 1's in the positions in the binary result corresponding to the powers recorded; in all other positions place 0's.**
- **Example: Convert 625_{10} to N_2**

Commonly Occurring Bases

Name	Radix	Digits
Binary	2	0,1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

- The six letters (in addition to the 10 integers) in hexadecimal represent:

Numbers in Different Bases

- **Good idea to memorize!**

Decimal (Base 10)	Binary (Base 2)	Octal (Base 8)	Hexadecimal (Base 16)
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10

Conversion Between Bases

- **Method 2**
- **To convert from one base to another:**

  1) Convert the Integer Part

  2) Convert the Fraction Part

  3) Join the two results with a radix point

Conversion Details

- **To Convert the Integral Part:**

  Repeatedly divide the number by the new radix and save the remainders. The digits for the new radix are the remainders in *reverse order* of their computation. If the new radix is > 10, then convert all remainders > 10 to digits A, B, …

- **To Convert the Fractional Part:**

  Repeatedly multiply the fraction by the new radix and save the integer digits that result. The digits for the new radix are the integer digits in *order* of their computation. If the new radix is > 10, then convert all integers > 10 to digits A, B, …

Example: Convert 46.6875_{10} To Base 2

- **Convert 46 to Base 2**

- **Convert 0.6875 to Base 2:**

- **Join the results together with the radix point:**

Additional Issue - Fractional Part

- Note that in this conversion, the fractional part became 0 as a result of the repeated multiplications.
- In general, it may take many bits to get this to happen or it may never happen.
- Example: Convert 0.65_{10} to N_2
  - 0.65 = 0.1010011001001 …
  - The fractional part begins repeating every 4 steps yielding repeating 1001 forever!
- Solution: Specify number of bits to right of radix point and round or truncate to this number.

Checking the Conversion

- **To convert back, sum the digits times their respective powers of r.**
- **From the prior conversion of 46.6875_{10}**

$$101110_2 = 1\cdot 32 + 0\cdot 16 + 1\cdot 8 + 1\cdot 4 + 1\cdot 2 + 0\cdot 1$$
$$= 32 + 8 + 4 + 2$$
$$= 46$$

$$0.1011_2 = 1/2 + 1/8 + 1/16$$
$$= 0.5000 + 0.1250 + 0.0625$$
$$= 0.6875$$

Why Do Repeated Division and Multiplication Work?

- **Divide the integer portion of the power series on slide 11 by radix r. The remainder of this division is A_0, represented by the term A_0/r.**
- **Discard the remainder and repeat, obtaining remainders A_1, …**
- **Multiply the fractional portion of the power series on slide 11 by radix r. The integer part of the product is A_{-1}.**
- **Discard the integer part and repeat, obtaining integer parts A_{-2}, …**
- **This demonstrates the algorithm for any radix r >1.**

Octal (Hexadecimal) to Binary and Back

- **Octal (Hexadecimal) to Binary:**
  - **Restate the octal (hexadecimal) as three (four) binary digits starting at the radix point and going both ways.**
- **Binary to Octal (Hexadecimal):**
  - **Group the binary digits into three (four) bit groups starting at the radix point and going both ways, padding with zeros as needed in the fractional part.**
  - **Convert each group of three bits to an octal (hexadecimal) digit.**

Octal to Hexadecimal via Binary

- **Convert octal to binary.**
- **Use groups of <u>four bits</u> and convert as above to hexadecimal digits.**
- **Example: Octal to Binary to Hexadecimal**

$$6 \quad 3 \quad 5 \;.\; 1 \quad 7 \quad 7\;_8$$

- **Why do these conversions work?**

A Final Conversion Note

- You can use arithmetic in other bases if you are careful:
- Example: Convert 101110_2 to Base 10 using binary arithmetic:

Step 1 101110 / 1010 = 100 r 0110

Step 2 100 / 1010 = 0 r 0100

Converted Digits are 0100_2 | 0110_2

or 4 6 $_{10}$

Binary Numbers and Binary Coding

- **Flexibility of representation**
  - **Within constraints below, can assign any binary combination (called a code word) to any data as long as data is uniquely encoded.**
- **Information Types**
  - **Numeric**
    - **Must represent range of data needed**
    - **Very desirable to represent data such that simple, straightforward computation for common arithmetic operations permitted**
    - **Tight relation to binary numbers**
  - **Non-numeric**
    - **Greater flexibility since arithmetic operations not applied.**
    - **Not tied to binary numbers**

Non-numeric Binary Codes

- **Given *n* binary digits (called <u>bits</u>), a <u>binary code</u> is a mapping from a set of <u>represented elements</u> to a subset of the 2^n binary numbers.**
- **Example: A binary code for the seven colors of the rainbow**
- **Code 100 is not used**

Color	Binary Number
Red	000
Orange	001
Yellow	010
Green	011
Blue	101
Indigo	110
Violet	111

Number of Bits Required

- **Given M elements to be represented by a binary code, the minimum number of bits, *n*, needed, satisfies the following relationships:**

  $$2^n \geq M > 2^{(n-1)}$$

  $n = \lceil \log_2 M \rceil$ where $\lceil x \rceil$, called the *ceiling function*, is the integer greater than or equal to x.

- **Example: How many bits are required to represent <u>decimal digits</u> with a binary code?**

Number of Elements Represented

- **Given *n* digits in radix *r*, there are r^n distinct elements that can be represented.**
- **But, you can represent m elements, m < r^n**
- **Examples:**
  - **You can represent 4 elements in radix $r = 2$ with $n = 2$ digits: (00, 01, 10, 11).**
  - **You can represent 4 elements in radix $r = 2$ with $n = 4$ digits: (0001, 0010, 0100, 1000).**
  - **This second code is called a "<u>one hot</u>" code.**

Binary Codes for Decimal Digits

- There are over 8,000 ways that you can chose 10 elements from the 16 binary numbers of 4 bits. A few are useful:

Decimal	8,4,2,1	Excess3	8,4,-2,-1	Gray
0	0000	0011	0000	0000
1	0001	0100	0111	0100
2	0010	0101	0110	0101
3	0011	0110	0101	0111
4	0100	0111	0100	0110
5	0101	1000	1011	0010
6	0110	1001	1010	0011
7	0111	1010	1001	0001
8	1000	1011	1000	1001
9	1001	1100	1111	1000

Binary Coded Decimal (BCD)

- The BCD code is the 8,4,2,1 code.
- This code is the simplest, most intuitive binary code for decimal digits and uses the same powers of 2 as a binary number, but only encodes the first ten values from 0 to 9.
- Example: 1001 (9) = 1000 (8) + 0001 (1)
- How many “invalid” code words are there?
- What are the “invalid” code words?

Excess 3 Code and 8, 4, –2, –1 Code

Decimal	Excess 3	8, 4, –2, –1
0	0011	0000
1	0100	0111
2	0101	0110
3	0110	0101
4	0111	0100
5	1000	1011
6	1001	1010
7	1010	1001
8	1011	1000
9	1100	1111

- **What interesting property is common to these two codes?**

Gray Code

Decimal	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

- **What special property does the Gray code have in relation to adjacent decimal digits?**

Warning: Conversion or Coding?

- **Do NOT mix up conversion of a decimal number to a binary number with coding a decimal number with a BINARY CODE.**
- **$13_{10} = 1101_2$ (This is conversion)**
- **13 $\Leftrightarrow$ 0001|0011 (This is coding)**

Binary Arithmetic

- **Single Bit Addition with Carry**
- **Multiple Bit Addition**
- **Single Bit Subtraction with Borrow**
- **Multiple Bit Subtraction**
- **Multiplication**
- **BCD Addition**

Single Bit Binary Addition with Carry

Given two binary digits (X,Y), a carry in (Z) we get the following sum (S) and carry (C):

Carry in (Z) of 0:

Z	0	0	0	0
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

Carry in (Z) of 1:

Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

Multiple Bit Binary Addition

- Extending this to two multiple bit examples:

Carries	0	0
Augend	01100	10110
Addend	+10001	+10111
Sum		

- Note: The 0 is the default Carry-In to the least significant bit.

Single Bit Binary Subtraction with Borrow

- **Given two binary digits (X,Y), a borrow in (Z) we get the following difference (S) and borrow (B):**

- **Borrow in (Z):**

Z	$\cancel{1}$	**1**	$\cancel{1}$	$\cancel{1}$
X	**0**	**0**	**1**	**1**
- Y	**-0**	**-1**	**-0**	**-1**
BS	**0**	**1**	**1**	**0**

Where, $\cancel{1}$ **= No need to take borrow**

Multiple Bit Binary Subtraction

- Extending this to two multiple bit examples:

Borrows	<u>0</u>	<u>0</u>
Minuend	10110	10110
Subtrahend	<u>- 10010</u>	<u>- 10011</u>
Difference		

- Notes: The <u>0</u> is a Borrow-In to the least significant bit. If the Subtrahend > the Minuend, interchange and append a – to the result.

Binary Multiplication

The binary multiplication table is simple:

$$0 * 0 = 0 \mid 1 * 0 = 0 \mid 0 * 1 = 0 \mid 1 * 1 = 1$$

Extending multiplication to <u>multiple digits</u>:

Multiplicand	**1011**
Multiplier	**<u>x 101</u>**
Partial Products	**1011**
	0000 -
	<u>1011 - -</u>
Product	**110111**

BCD Arithmetic

- Given a BCD code, we use binary arithmetic to add the digits:

8	1000	Eight
<u>+5</u>	<u>+0101</u>	Plus 5
13	1101	is 13 (> 9)

- Note that the result is MORE THAN 9, so must be represented by two digits!
- To correct the digit, subtract 10 by adding 6 modulo 16.

8	1000	Eight
<u>+5</u>	<u>+0101</u>	Plus 5
13	1101	is 13 (> 9)
	<u>+0110</u>	so add 6
	carry = 1 0011	leaving 3 + cy
	0001 | 0011	Final answer (two digits)

- If the digit sum is > 9, add one to the next significant digit

BCD Addition Example

- **Add 2905_{BCD} to 1897_{BCD} showing carries and digit corrections.**

$$
\begin{array}{rcccc}
 & & & & 0 \\
 & 0001 & 1000 & 1001 & 0111 \\
+ & \underline{0010} & \underline{1001} & \underline{0000} & \underline{0101} \\
 & & & & \\
 & \underline{} & \underline{} & \underline{} & \underline{}
\end{array}
$$

Error-Detection Codes

- **Redundancy** (e.g. extra information), in the form of extra bits, can be incorporated into binary code words to detect and correct errors.
- A simple form of redundancy is **parity**, an extra bit appended onto the code word to make the number of 1's odd or even. Parity can detect all single-bit errors and some multiple-bit errors.
- A code word has **even parity** if the number of 1's in the code word is even.
- A code word has **odd parity** if the number of 1's in the code word is odd.

4-Bit Parity Code Example

- **Fill in the even and odd parity bits:**

Even Parity Message - Parity	Odd Parity Message - Parity
000 _	000 _
001 _	001 _
010 _	010 _
011 _	011 _
100 _	100 _
101 _	101 _
110 _	110 _
111 _	111 _

- **The codeword "1111" has even parity and the codeword "1110" has odd parity. Both can be used to represent 3-bit data.**

ASCII Character Codes

- **American Standard Code for Information Interchange (Refer to Table 1 -4 in the text)**
- **This code is a popular code used to represent information sent as character-based data.   It uses 7-bits to represent:**
  - **94 Graphic printing characters.**
  - **34 Non-printing characters**
- **Some non-printing characters are used for text format (e.g. BS = Backspace, CR = carriage return)**
- **Other non-printing characters are used for record marking and flow control (e.g. STX and ETX start and end text areas).**

ASCII Properties

ASCII has some interesting properties:

- Digits 0 to 9 span Hexadecimal values 30_{16} to 39_{16}.
- Upper case A-Z span 41_{16} to $5A_{16}$.
- Lower case a-z span 61_{16} to $7A_{16}$.
  - Lower to upper case translation (and vice versa) occurs by flipping bit 6.
- Delete (DEL) is all bits set, a carryover from when punched paper tape was used to store messages.
- Punching all holes in a row erased a mistake!

UNICODE

- **UNICODE extends ASCII to 65,536 universal characters codes**
  - **For encoding characters in world languages**
  - **Available in many modern applications**
  - **2 byte (16-bit) code words**

Number representation

- Representing whole numbers
- Representing fractional numbers

Integer Representations

- Unsigned notation
- Signed magnitude notion
- Excess notation
- Two's complement notation.

Unsigned Representation

- Represents positive integers.
- Unsigned representation of 157:

position	7	6	5	4	3	2	1	0
Bit pattern	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**
contribution	2^7			2^4	2^3	2^2		2^0

- Addition is simple:

1 0 0 1 + 0 1 0 1 = 1 1 1 0.

Advantages and disadvantages of unsigned notation

- Advantages:
  - One representation of zero
  - Simple addition
- Disadvantages
  - Negative numbers can not be represented.
  - The need of different notation to represent negative numbers.

Representation of negative numbers

- Is a representation of negative numbers possible?
- Unfortunately:
  - you can not just stick a negative sign in front of a binary number. (it does not work like that)
- There are three methods used to represent negative numbers.
  - Signed magnitude notation
  - Excess notation notation
  - Two’s complement notation

Signed Magnitude Representation

- Unsigned: - and + are the same.
- In signed magnitude
  - the left-most bit represents the sign of the integer.
    - 0 for positive numbers.
    - 1 for negative numbers.
- The remaining bits represent to magnitude of the numbers.

Example

- Suppose 10011101 is a signed magnitude representation.
- The sign bit is 1, then the number represented is negative

position	7	6	5	4	3	2	1	0
Bit pattern	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**
contribution	-			2^4	2^3	2^2		2^0

- The magnitude is 0011101 with a value $2^4+2^3+2^2+2^0= 29$
- Then the number represented by 10011101 is –29.

Exercise 1

1. 37_{10} has 0010 0101 in signed magnitude notation. Find the signed magnitude of -37_{10} ?

2. Using the signed magnitude notation find the 8-bit binary representation of the decimal value 24_{10} and -24_{10}.

3. Find the signed magnitude of –63 using 8-bit binary sequence?

Disadvantage of Signed Magnitude

- Addition and subtractions are difficult.
- Signs and magnitude, both have to carry out the required operation.
- They are two representations of 0
  - 00000000 = $+ 0_{10}$
  - 10000000 = $- 0_{10}$
  - To test if a number is 0 or not, the CPU will need to see whether it is 00000000 or 10000000.
  - 0 is always performed in programs.
    - Therefore, having two representations of 0 is inconvenient.

Signed-Summary

- In signed magnitude notation,
  - The most significant bit is used to represent the sign.
  - 1 represents negative numbers
  - 0 represents positive numbers.
  - The unsigned value of the remaining bits represent The magnitude.
- Advantages:
  - Represents positive and negative numbers
- Disadvantages:
  - two representations of zero,
  - Arithmetic operations are difficult.

Excess Notation

- In excess notation:
  - The value represented is the unsigned value with a fixed value subtracted from it.
    - For n-bit binary sequences the value subtracted fixed value is $2^{(n-1)}$.
  - Most significant bit:
    - 0 for negative numbers
    - 1 for positive numbers

Excess Notation with n bits

- 1000…0 represent 2^{n-1} is the decimal value in unsigned notation.

Decimal value In unsigned notation	$- 2^{n-1}$	=	Decimal value In excess notation

- Therefore, in excess notation:
  - 1000…0 will represent 0 .

Example (1) - excess to decimal

- Find the decimal number represented by 10011001 in excess notation.
  - Unsigned value
    - $10011000_2 = 2^7 + 2^4 + 2^3 + 2^0 = 128 + 16 + 8 + 1 = 153_{10}$
  - Excess value:
    - excess value = $153 - 2^7 = 152 - 128 = 25$.

Example (2) - decimal to excess

- Represent the decimal value 24 in 8-bit excess notation.
- We first add, 2^{8-1}, the fixed value
  - $24 + 2^{8-1} = 24 + 128 = 152$
- then, find the unsigned value of 152
  - $152_{10} = 10011000$ (unsigned notation).
  - $24_{10} = 10011000$ (excess notation)

example (3)

- Represent the decimal value -24 in 8-bit excess notation.
- We first add, 2^{8-1}, the fixed value
  - $-24 + 2^{8-1} = -24 + 128 = 104$
- then, find the unsigned value of 104
  - $104_{10} = 01101000$ (unsigned notation).
  - $-24_{10} = 01101000$ (excess notation)

Example (4) (10101)

- Unsigned
  - $10101_2 = 16+4+1 = 21_{10}$
  - The value represented in unsigned notation is 21
- Sign Magnitude
  - The sign bit is 1, so the sign is negative
  - The magnitude is the unsigned value $0101_2 = 5_{10}$
  - So the value represented in signed magnitude is -5_{10}
- Excess notation
  - As an unsigned binary integer $10101_2 = 21_{10}$
  - subtracting $2^{5-1} = 2^4 = 16$, we get $21-16 = 5_{10}$.
  - So the value represented in excess notation is 5_{10}.

Advantages of Excess Notation

- It can represent positive and negative integers.
- There is only one representation for 0.
- It is easy to compare two numbers.
- When comparing the bits can be treated as unsigned integers.
- Excess notation is not normally used to represent integers.
- It is mainly used in floating point representation for representing fractions (later floating point rep.).

Exercise 2

1. Find 10011001 is an 8-bit binary sequence.
   - Find the decimal value it represents if it was in unsigned and signed magnitude.
   - Suppose this representation is excess notation, find the decimal value it represents?
2. Using 8-bit binary sequence notation, find the unsigned, signed magnitude and excess notation of the decimal value 11_{10}?

Excess notation - Summary

- In excess notation, the value represented is the unsigned value with a fixed value subtracted from it.
  - i.e. for n-bit binary sequences the value subtracted is $2^{(n-1)}$.
- Most significant bit:
  - 0 for negative numbers .
  - 1 positive numbers.
- Advantages:
  - Only one representation of zero.
  - Easy for comparison.

Two's Complement Notation

- The most used representation for integers.
  - All positive numbers begin with 0.
  - All negative numbers begin with 1.
  - One representation of zero
    - i.e. 0 is represented as 0000 using 4-bit binary sequence.

Two’s Complement Notation with 4-bits

Binary pattern for 2’s complement	Value.
0 1 1 1	7
0 1 1 0	6
0 1 0 1	5
0 1 0 0	4
0 0 1 1	3
0 0 1 0	2
0 0 0 1	1
0 0 0 0	0
1 1 1 1	-1
1 1 1 0	-2
1 1 0 1	-3
1 1 0 0	-4
1 0 1 1	-5
1 0 1 0	-6
1 0 0 1	-7
1 0 0 0	-8

Properties of Two's Complement Notation

- Positive numbers begin with 0
- Negative numbers begin with 1
- Only one representation of 0, i.e. 0000
- Relationship between +n and –n.

• 0 1 0 0	+4	0 0 0 1 0 0 1 0	+18
• 1 1 0 0	-4	1 1 1 0 1 1 1 0	-18

Advantages of Two's Complement Notation

- It is easy to add two numbers.

```
   0 0 0 1   +1            1 0 0 0  -8
 + 0 1 0 1   +5          + 0 1 0 1  +5
 ------------            ------------
   0 1 1 0   +6            1 1 0 1  -3
```

- **Subtraction can be easily performed.**
- **Multiplication is just a repeated addition.**
- **Division is just a repeated subtraction**
- **Two's complement is widely used in *ALU***

Evaluating numbers in two's complement notation

- Sign bit = 0, the number is positive. The value is determined in the usual way.
- Sign bit = 1, the number is negative. three methods can be used:

Method 1	decimal value of (n-1) bits, then subtract 2^{n-1}
Method 2	-2^{n-1} is the contribution of the sign bit.
Method 3	● Binary rep. of the corresponding positive number. ● Let V be its decimal value. ● - V is the required value.

Example- 10101 in Two's Complement

- The most significant bit is 1, hence it is a negative number.
- Method 1
  - 0101 = +5 $(+5 - 2^{5-1} = 5 - 2^4 = 5\text{-}16 = \text{-}11)$
- Method 2

4	3	2	1	0		
1	0	1	0	1		
-2^4		2^2		2^0	=	-11

- Method 3
  - Corresponding + number is 01011 = 8 + 2+1 = 11 the result is then –11.

Two's complement-summary

- In two's complement the most significant for an n-bit number has a contribution of $-2^{(n-1)}$.
- One representation of zero
- All arithmetic operations can be performed by using addition and inversion.
- The most significant bit: 0 for positive and 1 for negative.
- Three methods can the decimal value of a negative number:

Method 1	decimal value of (n-1) bits, then subtract 2^{n-1}
Method 2	-2^{n-1} is the contribution of the sign bit.
Method 3	• Binary rep. of the corresponding positive number. • Let V be its decimal value. • - V is the required value.

Exercise - 10001011

- Determine the decimal value represented by 10001011 in each of the following four systems.
  1. Unsigned notation?
  2. Signed magnitude notation?
  3. Excess notation?
  4. Tow’s complements?

Fraction Representation

- To represent fraction we need other representations:
  - Fixed point representation
  - Floating point representation.

Fixed-Point Representation

old position	7	6	5	4	3	2	1	0	
New position	4	3	2	1	0	-1	-2	-3	
Bit pattern	**1**	**0**	**0**	**1**	**1 .**	**1**	**0**	**1**	
Contribution	2^4			2^1	2^0	2^{-1}		2^{-3}	=19.625

▪**Radix-point**

Limitation of Fixed-Point Representation

- To represent large numbers or very small numbers we need a very long sequences of bits.

- This is because we have to give bits to both the integer part and the fraction part.

Floating Point Representation

In decimal notation we can get around this problem using **scientific notation** or **floating point notation.**

Number	Scientific notation	Floating-point notation
1,245,000,000,000	$1.245\ 10^{12}$	$0.1245\ 10^{13}$
0.0000001245	$1.245\ 10^{-7}$	$0.1245\ 10^{-6}$
-0.0000001245	$-1.245\ 10^{-7}$	$-0.1245\ 10^{-6}$

Floating Point

-15900000000000000

could be represented as

- Mantissa
- Base
- Sign
- Exponent

$- 159 * 10^{14}$

$- 15.9 * 10^{15}$

$- 1.59 * 10^{16}$

A calculator might display 159 E14

Floating point format

- $\pm M \times B^{\pm E}$

- Sign
- mantissa or significand
- base
- exponent

Sign	Exponent	Mantissa

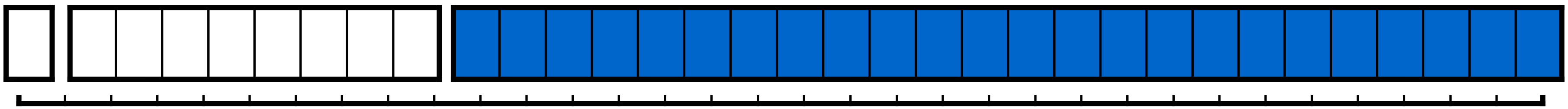
Floating Point Representation format

▪Sign	▪Exponent	▪Mantissa

- The exponent is biased by a fixed value b, called the bias.
- The mantissa should be normalised, e.g. if the real mantissa if of the form 1.f then the normalised mantissa should be f, where f is a binary sequence.

IEEE 745 Single Precision

- The number will occupy 32 bits



- The first bit represents the sign of the number;
- 1= negative 0= positive.

- The next 8 bits will specify the exponent stored in biased 127 form.

- The remaining 23 bits will carry the mantissa normalised to be between 1 and 2.

- i.e. 1<= mantissa < 2

Representation in IEEE 754 single precision

- sign bit:
  - 0 for positive and,
  - 1 for negative numbers
- 8 biased exponent by 127
- 23 bit normalised mantissa

▪Sign	▪Exponent	▪Mantissa

Basic Conversion

- Converting a decimal number to a floating point number.
  - 1.Take the integer part of the number and generate the binary equivalent.
  - 2.Take the fractional part and generate a binary fraction
  - 3.Then place the two parts together and normalise.

IEEE – Example 1

- **Convert 6.75 to 32 bit IEEE format.**
- 1. The Mantissa. The Integer first.
- 6 / 2 = 3 r 0
- 3 / 2 = 1 r 1
- 1 / 2 = 0 r 1

- **= 110_2**

IEEE – Example 1

- **Convert 6.75 to 32 bit IEEE format.**
- 1. The Mantissa. The Integer first.
- 6 / 2 = 3 r 0
- 3 / 2 = 1 r 1
- 1 / 2 = 0 r 1

= 110_2

- 2. Fraction next.
- .75 * 2 = 1.5
- .5 * 2 = 1.0

= 0.11_2

IEEE – Example 1

- **Convert 6.75 to 32 bit IEEE format.**
- 1. The Mantissa. The Integer first.
- 6 / 2 = 3 r 0
- 3 / 2 = 1 r 1
- 1 / 2 = 0 r 1

- **= 110_2**

- 2. Fraction next.
- .75 * 2 = 1.5
- .5 * 2 = 1.0

= 0.11_2

- 3. put the two parts together… 110.11
- Now normalise 1.1011 * 2^2

IEEE Biased 127 Exponent

- **To generate a biased 127 exponent**
- Take the value of the signed exponent and add 127.
- Example.
- **2^{16} then $2^{127+16} = 2^{143}$ and my value for the exponent would be $143 = 10001111_2$**
- So it is simply now an unsigned value ....

Possible Representations of an Exponent

Binary	Sign Magnitude	2's Complement	Biased 127 Exponent XXX			Biased 127 Exponent IEEE 754	
00000000	0	0	0+127	127		0-127	-127
00000001	1	1	1+127	128		1-127	-126
00000010	2	2	2+127	129		2-127	-125
01111110	126	126	126+127	253		126-127	-1
01111111	127	127	127+127	254		127-127	0
10000000	-0	-128	0+127	127		128-127	1
10000001	-1	-127	-1+127	126		129-127	2
11111110	-126	-2	-126+127	1		254-127	127
11111111	-127	-1	-127+127	0		255-127	128

Why Biased ?

- The smallest exponent 00000000
- Only one exponent zero 01111111
- The highest exponent is 11111111
- To increase the exponent by one simply add 1 to the present pattern.

Back to the example

- **Our original example revisited…. 1.1011 * 2^2**
- **Exponent is 2+127 =129 or 10000001 in binary.**
- **NOTE: Mantissa always ends up with a value of '1' before the Dot. This is a waste of storage therefore it is implied but not actually stored. 1.1000 is stored .1000**

- **6.75 in 32 bit floating point IEEE representation:-**
- **0 10000001 10110000000000000000000**
- **sign(1) exponent(8) mantissa(23)**

Representation in IEEE 754 single precision

- sign bit:
  - 0 for positive and,
  - 1 for negative numbers
- 8 biased exponent by 127
- 23 bit normalised mantissa

▪Sign	▪Exponent	▪Mantissa

Example (2)

- which number does the following IEEE single precision notation represent?

1	1000 0000	0100 0000 0000 0000 0000 000

- The sign bit is 1, hence it is a negative number.
- The exponent is 1000 0000 = 128_{10}
- It is biased by 127, hence the real exponent is

  128 –127 = 1.
- The mantissa: 0100 0000 0000 0000 0000 000.
- It is normalised, hence the true mantissa is

  $1.01 = 1.25_{10}$
- Finally, the number represented is: $-1.25 \times 2^1 = -2.50$

Single Precision Format

- The exponent is formatted using excess-127 notation, with an implied base of 2
  - Example:
    - Exponent: **10000111**
    - Representation: **135 – 127 = 8**
- The stored values 0 and 255 of the exponent are used to indicate special values, the exponential range is restricted to
- 2^{-126} to 2^{127}
- The number 0.0 is defined by a mantissa of 0 together with the special exponential value 0
- The standard allows also values +/-∞ (represented as mantissa +/-0 and exponent 255
- Allows various other special conditions

In comparison

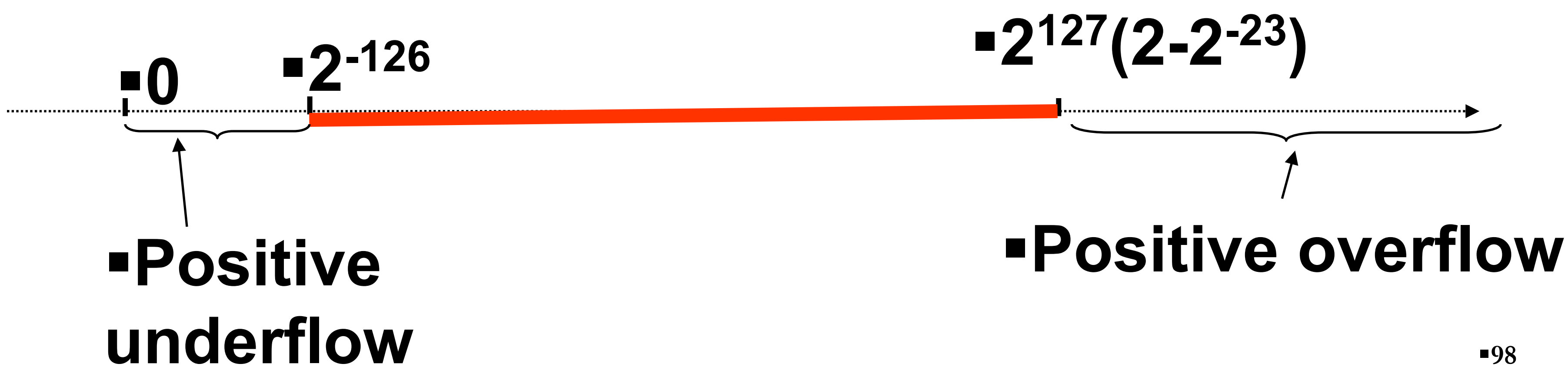
- The smallest and largest possible 32-bit integers in two's complement are only -2^{32} and $2^{31} - 1$

Range of numbers

- Normalized (positive range; negative is symmetric)

smallest: 00000000100000000000000000000000 → $+2^{-126} \times (1+0) = 2^{-126}$

largest: 01111111011111111111111111111111 → $+2^{127} \times (2-2^{-23})$



Representation in IEEE 754 double precision format

- It uses 64 bits
  - 1 bit sign
  - 11 bit biased exponent
  - 52 bit mantissa

Sign	Exponent	Mantissa

IEEE 754 double precision Biased = 1023

- 11-bit exponent with an excess of 1023.
- For example:
  - If the exponent is **-1**
    - we then add 1023 to it. -1+1023 = 1022
    - We then find the binary representation of 1022
      - Which is 0111 1111 110
    - The exponent field will now hold 0111 1111 110
  - This means that we just represent -1 with an excess of 1023.

IEEE 754 Encoding

Single Precision		Double Precision		Represented Object
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	non-zero	0	non-zero	**+/- denormalized number**
1~254	anything	1~2046	anything	**+/- floating-point numbers**
255	0	2047	0	+/- infinity
255	non-zero	2047	non-zero	NaN (Not a Number)

Floating Point Representation format (summary)

Sign	Exponent	Mantissa

- the sign bit represents the sign
  - 0 for positive numbers
  - 1 for negative numbers
- The exponent is biased by a fixed value b, called the bias.
- The mantissa should be normalised, e.g. if the real mantissa if of the form 1.f then the normalised mantissa should be f, where f is a binary sequence.

Character representation- ASCII

- ASCII (American Standard Code for Information Interchange)
- It is the scheme used to represent characters.
- Each character is represented using 7-bit binary code.
- If 8-bits are used, the first bit is always set to 0
- See (table 5.1 p56, study guide) for character representation in ASCII.

ASCII – example

Symbol	decimal	Binary
7	55	00110111
8	56	00111000
9	57	00111001
:	58	00111010
;	59	00111011
<	60	00111100
=	61	00111101
>	62	00111110
?	63	00111111
@	64	01000000
A	65	01000001
B	66	01000010
C	67	01000011