

# Lecture 3

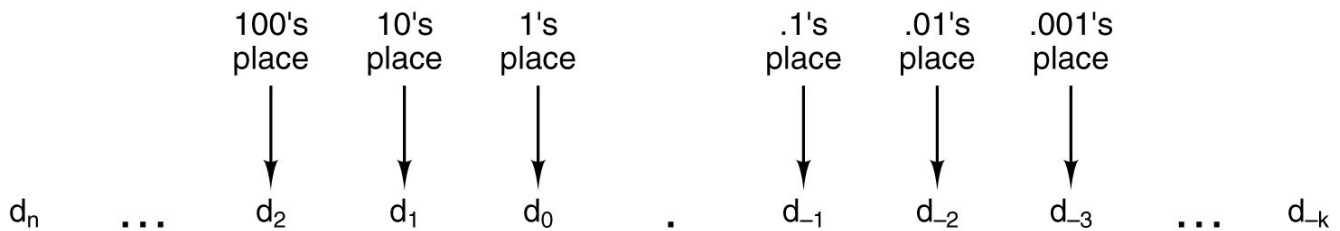
## Number Representation

### Integers

- Several bytes together are used to store integers
- 2 bytes (1/2 word) form a short integer
- 4 bytes (1 word) form a regular integer
- 8 bytes (2 words) form a long integer
- Computers work with fixed sized numbers, aka *finite precision numbers* (fixed when the computer is designed and built).
  - o What does this mean?
    - Computers can only deal with numbers that can be represented by a fixed number of digits.
    - Example: Only positive integer numbers represented by, say, 3 decimal digits
      - 1000 unique numbers (000, 001, 002, ..., 999)
      - Impossible numbers: > 999; negative numbers; fractions; irrational numbers; complex numbers

## Radix Number System

- Also known as *positional number system*, or *base k system*
- In a  $k$  radix system there are  $k$  symbols, typically 0 to  $(k-1)$
- For integers, value of the  $i^{\text{th}}$  digit  $d$  is:  $d * \text{base}^i = d * k^i$  where  $k$  is the base
  - o  $i$  starts from 0 and is counted from right to left



$$\text{Number} = \sum_{i=-k}^n d_i \times 10^i$$

## Binary Numbers (Radix – 2)

- We have never before had to worry about how many decimal places are required to represent a number
- But since we have *finite precision number* this is a concern here
- Typically, one memory word is used to store a number
- Number of bits is fixed (even with long integers or double precision numbers)
- Other Common Radices
  - o Octal (base 8, radix – 8) : requires 8 symbols, symbols 0, ..., 7
  - o Decimal (base 10, radix –10) : requires 10 symbols, symbols 0, ..., 9
  - o Hexadecimal (base 16, radix – 16) : requires 16 symbols, 0, ..., 9, A, B, C, D, E, F

Binary      1      1      1      1      1      0      1      0      0      0      1  
 $1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$   
 1024 + 512 + 256 + 128 + 64 + 0 + 16 + 0 + 0 + 0 + 1

Octal        3      7      2      1  
 $3 \times 8^3 + 7 \times 8^2 + 2 \times 8^1 + 1 \times 8^0$   
 1536 + 448 + 16 + 1

Decimal     2      0      0      1  
 $2 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$   
 2000 + 0 + 0 + 1

Hexadecimal 7      D      1      .  
 $7 \times 16^2 + 13 \times 16^1 + 1 \times 16^0$   
 1792 + 208 + 1

- **ALWAYS** write the radix or base along with the number to eliminate any ambiguity
  - o Example:  $111_2$  vs.  $111_{16}$  → in decimal: 7 vs. 273

HEX Values (base-16)

- HEX is commonly used in SPIM.
- Memory Address are usually represented in HEX
- ASCII characters should be referred to in their HEX form in SPIM
- Example: 41 in HEX is the ASCII character 'A' → 0 100 0001  
 → 0x41 (this how you should refer to it in your programs)
- The only way to refer to the non-printable ASCII characters (0x00 - 0x1F) is to use their HEX values

Decimal	Binary	Octal	Hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD



## Radix to Radix

- Convert number to binary.
- Regroup binary digits according to new radix.
- Always start counting from the decimal point! Fractional parts from the decimal point to the right, whole numbers from the decimal point to the left.

### Example 1

Hexadecimal	1	9	4	8	.	B	6		
Binary	0001	1001	1010	0100	.	1011	101100		
Octal	1	4	5	1	0	.	5	5	4

### Example 2

Hexadecimal	7	B	A	3	.	B	C	4		
Binary	0111	1011	1010	0011	.	1011	1100	0100		
Octal	7	5	6	4	3	.	5	7	0	4

Or

- Divide by the appropriate radix (more difficult).

## Converting Decimal Fractions to Binary

Step-by-step method for computing the binary expansion on the right-hand side of the point. We will illustrate the method by converting the decimal value .625 to a binary representation.

**Step 1:** Begin with the decimal fraction and multiply by 2. The whole number part of the result is the first binary digit to the right of the point.

Because  $.625 \times 2 = 1.25$ , the first binary digit to the right of the point is a **1**

So far, we have  $.625 = .1??? \dots$  (base 2) .

**Step 2:** Next we disregard the whole number part of the previous result (the 1 in this case) and multiply by 2 once again. The whole number part of this new result is the *second* binary digit to the right of the point. We will continue this process until we get a zero as our decimal part or until we recognize an infinite repeating pattern.

Because  $.25 \times 2 = 0.50$ , the second binary digit to the right of the point is a **0**

So far, we have  $.625 = .10?? \dots$  (base 2) .

**Step 3:** Disregarding the whole number part of the previous result (this result was .50 so there actually is no whole number part to disregard in this case), we multiply by 2 once again. The whole number part of the result is now the next binary digit to the right of the point.

Because  $.50 \times 2 = 1.00$ , the third binary digit to the right of the point is a **1**

So now we have  $.625 = .101?? \dots$  (base 2) .

**Step 4:** In fact, we do not need a Step 4. We are finished in Step 3, because we had 0 as the fractional part of our result there.

Hence the representation of  $.625 = .101$  (base 2) . Double-check our result by expanding the binary representation.

Ex ample: Decimal value 10.3

1. Convert the integer part.  $10_{10} \rightarrow 1010_2$
2. Decimal point becomes the binary point.
3. Start addressing the fraction. (.3) Multiply by 2  $\rightarrow 0.6$  . The whole number part becomes the digit for the binary representation. So far,  $1010.0\dots_2$
4. Repeat the process.  $0.6 * 2 = 1.2$  The whole number part becomes the digit for binary. So far  $1010.01\dots_2$   
Disregard the whole number part, leaving 0.2
5. Repeat.  $0.2 * 2 = 0.4 \rightarrow 1010.010\dots_2$
6. Repeat.  $0.4 * 2 = 0.8 \rightarrow 1010.0100\dots_2$
7. Repeat.  $0.8 * 2 = 1.6 \rightarrow 1010.01001\dots_2$
8. Repeat.  $0.6 * 2 = 1.2 \rightarrow 1010.010011\dots_2$
9. Repeat.  $0.2 * 2 = 0.4 \rightarrow 1010.0100110\dots_2$

***Will it ever stop???***

## Infinite Binary Fractions

The method we just explored can be used to demonstrate how some decimal fractions will produce infinite binary fraction expansions. We illustrate this by using the method to show that the binary representation for the decimal fraction  $1/10$  is, in fact, infinite.

**Step 1:** Begin with the decimal fraction and multiply by 2. The whole number part of the result is the first binary digit to the right of the point.

Because  $.1 \times 2 = 0.2$ , the first binary digit to the right of the point is a **0**

So far, we have  $.1$  (decimal) =  $.0???$  . . . (base 2) .

**Step 2:** Next we disregard the whole number part of the previous result (0 in this case) and multiply by 2 once again. The whole number part of this new result is the *second* binary digit to the right of the point. We will continue this process until we get a zero as our decimal part, or until we recognize an infinite repeating pattern.

Because  $.2 \times 2 = 0.4$ , the second binary digit to the right of the point is also a **0**

So far, we have  $.1$  (decimal) =  $.00??$  . . . (base 2) .

**Step 3:** Disregarding the whole number part of the previous result (again, a 0), we multiply by 2 once again. The whole number part of the result is now the next binary digit to the right of the point.

Because  $.4 \times 2 = 0.8$ , the third binary digit to the right of the point is also a **0**

So now we have  $.1$  (decimal) =  $.000??$  . . . (base 2) .

**Step 4:** We multiply by 2 once again, disregarding the whole number part of the previous result (again, a 0 in this case).

Because  $.8 \times 2 = 1.6$ , the fourth binary digit to the right of the point is a **1**

So now we have  $.1$  (decimal) =  $.0001??$  . . . (base 2) .

**Step 5:** We multiply by 2 once again, disregarding the whole number part of the previous result (a 1 in this case).

Because  $.6 \times 2 = 1.2$ , the fifth binary digit to the right of the point is a **1**

So now we have  $.1$  (decimal) =  $.00011??$  . . . (base 2) .

**Step 6:** We multiply by 2 once again, disregarding the whole number part of the previous result. Let's make an important observation here. Notice that this next step to be performed (multiply  $2. \times 2$ ) is **exactly the same action we had in Step 2**. We are thus bound to repeat steps 2-5, then return to Step 2 again indefinitely. In other words, we will never get a 0 as the decimal fraction part of our result. Instead we will just cycle through steps 2-5 forever. This means we will obtain the sequence of digits generated in steps 2-5, namely 0011, over and over. Hence, the final binary representation will be.

$.1$  (decimal) =  $.00011001100110011 \dots$  (base 2) .

The repeating pattern is more obvious if we highlight it in color as below:

$.1$  (decimal) =  $.00011001100110011 \dots$  (base 2) .

- Depending on how many bits the computer has for the number, we get as exact a representation as possible.
- For some decimal fractions, there is no exact representation.
- For every decimal integer, we have an exact binary integer.

## Excess $N$ or Bias $N$ Notation

The point of excess something representation for values is that numbers stay in the order you expect

- the greatest negative is the smallest value
- 0 is in the middle
- the greatest positive is the largest value

For Excess  $N$ , integer  $X$  is represented by  $X + N$

Bit patterns yield a sequence that represents the values  $-N$  to  $-N + (2^m - 1)$ , where  $m$  is the bit-group size →

if  $N = 2^{m-1}$  where  $m$  is the bit-group size, then the Excess  $N$  value = 2's complement value with the sign bit inverted.

Binary	Excess 1	Excess 4	Excess 9
000	-1	-4	-9
001	0	-3	-8
010	1	-2	-7
011	2	-1	-6
100	3	0	-5
101	4	1	-4
110	5	2	-3
111	6	3	-2

Excess 127, for example, is used in representing floating point exponents.

- The value 43 would be represented as  $43 + 127 = 170$
- The value -43 would be represented as  $-43 + 127 = 84$
- The smaller values (the negatives) start with 0 in binary; the larger values (the positives) start with binary 1. This is the opposite of 2's complement, where the negatives start with 1 and the positives start with 0.

## Binary-coded decimal

- Represent each DECIMAL digit by its 4-bit binary value
- Example: 93 → 1001 0011
- Use 2 of the six spare bit-patterns for the sign symbols, e.g. 1110 for +, 1111 for –  
Example: -93 → 1111 1001 0011
- Very fast conversion to/from decimal. Ideal for applications which perform little arithmetic
- BCD integers can be variable-length

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
	1010
	1011
	1100
	1101
+	1110
-	1111