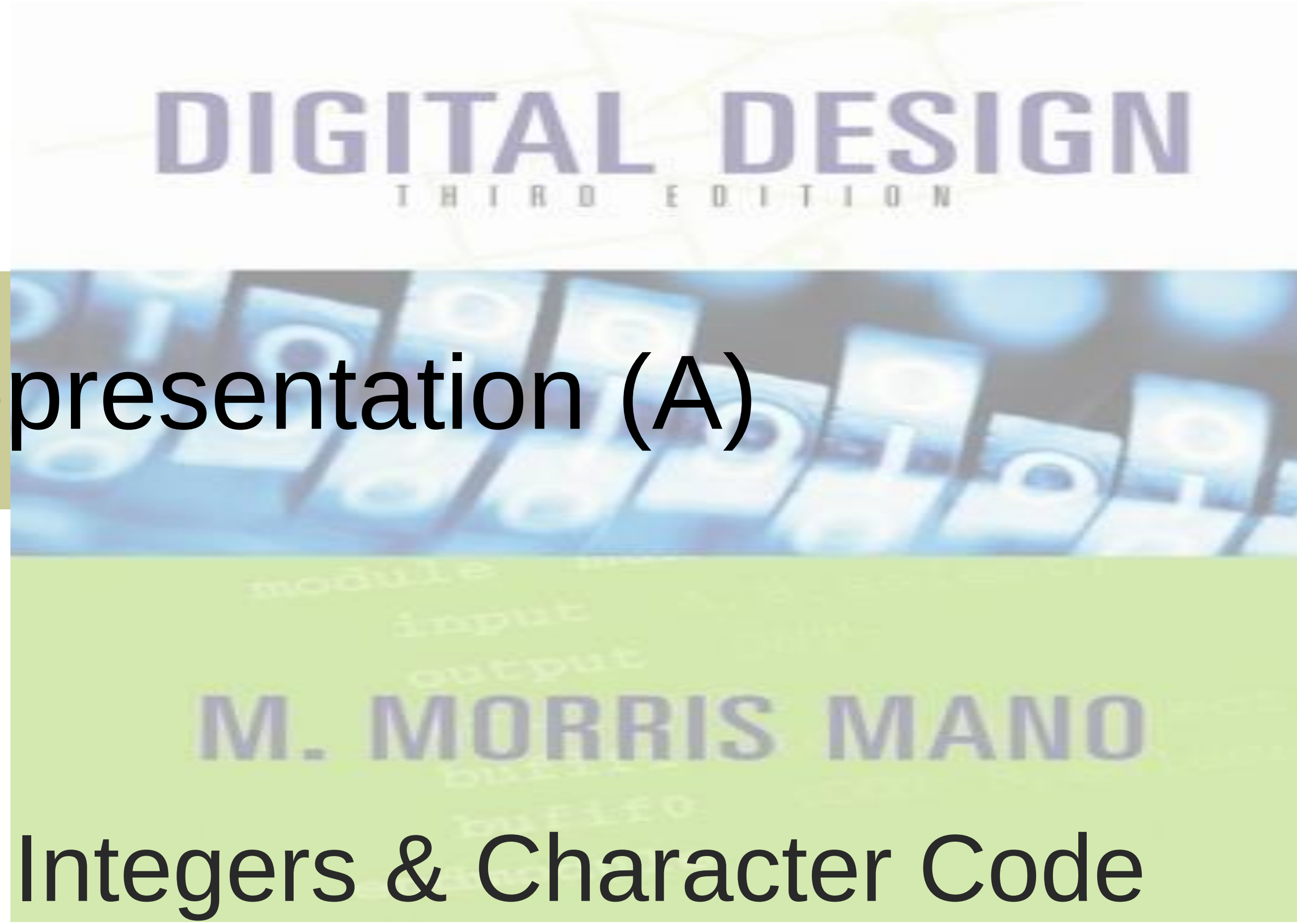
Data Representation (A)



Integers & Character Code

School of Information Technology, KMUTT
CS@SIT

Signals & Binary Systems

- In modern digital systems, signals are viewed as a two-value discrete element
  - A binary value
  - Representations: L/H, 0/1, On/Off
  - When using digits 0 and 1, the term *bit* is used
- Group of bits are used to represent "discrete elements of information"
  - How? ➔ Depends on the *binary code*

COMSCI INTER - BANGMOD -

Decimal Numbers

$$a_n a_{n-1} \ldots a_2 a_1 a_0 . a_{-1} a_{-2} \ldots a_{-m} = 10^n a_n + 10^{n-1} a_{n-1} + \ldots + 10^2 a_2 + 10^1 a_1 + 10^0 a_0 + 10^{-1} a_{-1} + 10^{-2} a_{-2} + \ldots + 10^{-m} a_{-m}$$

- Example:

  $2345.67 = 10^3(2) + 10^2(3) + 10^1(4) + 10^0(5) + 10^{-1}(6) + 10^{-2}(7)$

  $-54.09 = 10^1(-5) + 10^0(-4) + 10^{-1}(0) + 10^{-2}(-9)$

- Decimal numbers are of *base* or *radix* 10
  - They use 10 digits, and
  - Coefficients are multiplied by powers of 10

Binary Numbers

- Are of *base* or *radix* 2
  - 0 and 1 are used
  - Coefficients multiplied by powers of 2
- $11010.11 = 2^4(1) + 2^3(1) + 2^2(0) + 2^1(1) + 2^0(0) + 2^{-1}(1) + 2^{-2}(1) = 26.75$
- General expression

$$2^n a_n + 2^{n-1} a_{n-1} + \ldots + 2^2 a_2 + 2^1 a_1 + 2^0 a_0 + 2^{-1} a_{-1} + 2^{-2} a_{-2} + \ldots + 2^{-m} a_{-m}$$

Converting unsigned whole number (DEC2xxx)

- Two methods

EXAMPLE 2.2 Convert 104_{10} to base 3 using subtraction.

The highest radix that is < 104

$$
\begin{array}{rl}
104 & \\
\underline{-81} & = 3^4 \times 1 \\
23 & \\
\underline{-0} & = 3^3 \times 0 \\
23 & \\
\underline{-18} & = 3^2 \times 2 \\
5 & \\
\underline{-3} & = 3^1 \times 1 \\
2 & \\
\underline{-2} & = 3^0 \times 2 \\
0 &
\end{array}
$$

The answer will be
x x x x x

$$104_{10} = 10212_3$$

Division-remainder

$$
\begin{array}{rl}
3 \,\lfloor 104 & 2 \\
3 \,\lfloor 34 & 1 \\
3 \,\lfloor 11 & 2 \\
3 \,\lfloor 3 & 0 \\
3 \,\lfloor 1 & 1 \\
0 &
\end{array}
$$

Decimal point to Binary point Conversions

- **Converting 0.8125 to binary . . .**
  - You are finished when the product is zero, or until you have reached the desired number of binary places.
  - Our result, reading from top to bottom is:

    $0.8125_{10} = 0.1101_2$
  - This method also works with any base. Just use the target radix as the multiplier.

$$
\begin{array}{r}
.8125 \\
\times \quad 2 \\
\hline
1.6250
\end{array}
\qquad
\begin{array}{r}
.6250 \\
\times \quad 2 \\
\hline
1.2500
\end{array}
\qquad
\begin{array}{r}
.2500 \\
\times \quad 2 \\
\hline
0.5000
\end{array}
\qquad
\begin{array}{r}
.5000 \\
\times \quad 2 \\
\hline
1.0000
\end{array}
$$

Base-*r* System

- A general expression for any base-*r* number is

$$r^n a_n + r^{n-1} a_{n-1} + \ldots + r^2 a_2 + r^1 a_1 + r^0 a_0 + r^{-1} a_{-1} + r^{-2} a_{-2} + \ldots + r^{-m} a_{-m}$$

where $0 \le a_j \le r-1$

- Examples:
  - $(4021)_5$
  - $(127)_8$
  - $(B65F)_{16}$
  - $(110101)_2$

COMSCI INTER - BANGMOD -

What is $(41)_{10}$ in binary?

Integer Quotient	Remainder	Coefficient

Decimal-to-base-r conversion

- Let p be a decimal number
- Procedure for an *integer* part
  - Divide p by radix r
    - p_0 = integer quotient & a_0 = remainder
  - Divide p_0 by radix r
    - p_1 = integer quotient & a_1 = remainder
  - Repeat until p_n = integer quotient = 0 and a_n = remainder
  - The integer p in *base-r* = $(a_n \ldots a_1 a_0)_r$

COMSCI INTER - BANGMOD -

Convert $(153)_{10}$ to octal

Integer Quotient	Remainder	Coefficient

Binary, Octal, and Hexadecimal

- 3 most popular numerical formats for digital computers
- Octal uses digits 0-7
  - Requires 3 binary bits 000, 001, …, 111 to represent all 8 values
- Hexadecimal uses digits 0-9, and letters A-F (for 10-15)
  - Requires 4 binary bits 0000, 0001, …, 1111 to represent all 16 values

COMSCI INTER - BANGMOD -

Conversions

- From binary
  - By grouping binary bits into groups of 3 (for octal) or 4 (for hexadecimal)
- Example:

$(101100011011011)_2$

Conversions (Continued)

- To binary
  - By expanding each octal or hexadecimal digit to its 3-bit or 4-bit binary equivalent
- Example:

$(306.D)_{16}$

COMSCI INTER - BANGMOD -

Binary arithmetic

- Addition table:

0 + 0 = 0	0 + 1 = 1
1 + 0 = 1	1 + 1 = 0 (with carry)

- Subtraction table:

0 – 0 = 0	0 – 1 = 1 (borrow)
1 – 0 = 1	1 – 1 = 0

- Multiplication table:

0 x 0 = 0	0 x 1 = 0
1 x 0 = 0	1 x 1 = 1

Binary operations

```
  101101         101101        1011
 +100111       - 100111      x  101
```

Booth's Algorithm

In Booth's algorithm:

- If the current multiplier bit is 1 and the preceding bit was 0, subtract the multiplicand from the product
- If the current multiplier bit is 0 and the preceding bit was 1, we add the multiplicand to the product
- If we have a 00 or 11 pair, we simply shift.
- Assume a mythical "0" starting bit
- Shift after each step

```
        0011
    x   0110
    --------
    +   0000     (shift)
   -   0011      (subtract)
  +   0000       (shift)
 +   0011        (add)
 -----------
    00010010
```

We see that 3 × 6 = 18!

COMSCI INTER - BANGMOD -

Booth's Algorithm

- Here is a larger example.

```
                  00110101
x                 01111110
--------------------------
+ 0000000000000000
+ 111111111001011
+ 00000000000000
+ 0000000000000
+ 000000000000
+ 00000000000
+ 0000000000
+ 000110101
--------------------------
 10001101000010110
```

Ignore all bits over 2n.

53 × 126 = 6678!

Diminished Radix Complement

- (*r-1*)'s complement of the *base-r*
- a 9's complement of a base-10 number is obtained by subtracting each digit in the number from 9
- a 1's complement of a base-2 number is obtained by
  - subtracting each bit in the number from 1

  OR

  - changing bits 1 → 0, 0 → 1

Examples

- **9's complement of**:

  546700 = 999999 – 546700

  = 453299

  012398 = 999999 – 012398

  = 987601

- **1's complement of**:

  1011000 = 0100111

  0101101 = 1010010

Radix Complement

- *r*'s complement of the *base-r*
- How?
  - Find the (*r-1*)'s complement
  - Add 1
- Example:

10's complement of 546700

453299 + 1 = 543300

2's complement of 1101100

0010011 + 1 = 0010100

COMSCI INTER - BANGMOD -

Complement system

- One’s complement
  - Actually, 1’s complement of 1011 = 1111-1011 = 0100
  - Luckily, it is equivalent to switching all digits
    - Very simple to implement in computer hardware
  - Very useful to represent negative number
    - Automatically, leftmost bit = 1 for negative numbers (0111 → 1000)
    - Simplify the subtraction by turning it into addition
    - E.g. 5 – 2 = 5 + (-2)

COMSCI INTER - BANGMOD -

Example of 1's complement arithmetic

EXAMPLE 2.17 Add 23_{10} to -9_{10} using one's complement arithmetic.

	1 ←	1 1 1 1 1	⇐ carries
		0 0 0 1 0 1 1 1	(23)
The last		+ 1 1 1 1 0 1 1 0	+ (–9)
carry is added		0 0 0 0 1 1 0 1	
		+ 1	
to the sum.		0 0 0 0 1 1 1 0	14_{10}

EXAMPLE 2.18 Add 9_{10} to -23_{10} using one's complement arithmetic.

The last	0 ←	0 0 0 0 1 0 0 1	(9)
carry is zero		+ 1 1 1 0 1 0 0 0	+ (–23)
so we are done.		1 1 1 1 0 0 0 1	-14_{10}

Disadvantages
- Two representation of zero (000,111)
- Two-step arithmetic operation

Advantages
- Addition only
- Get result instantly

2's Complement

- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.
  - Example: Using two's complement binary arithmetic, find the sum of 48 and - 19.

```
1       1 1
          0 0 1 1 0 0 0 0
        + 1 1 1 0 1 1 0 1
        -----------------
          0 0 0 1 1 1 0 1
```

We note that 19 in one's complement is: 00010011,
so -19 in one's complement is: 11101100,
and -19 in two's complement is: 11101101.

COMSCI INTER - BANGMOD -

Addition of 2's complement

- Using only binary addition
- ALU signals overflow, if out of range
- Operable range:

  -8 to 7
- Rule: for the sign bit,

Overflow when:

Carry in ≠ Cary out

(at the Most significant bit)

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Binary Multiplication by Shifting

- We can do binary multiplication and division by 2 very easily using an *arithmetic shift* operation
- A *left arithmetic shift* inserts a 0 in for the rightmost bit and shifts everything else left one bit; in effect, it multiplies by 2
- A *right arithmetic shift* shifts everything one bit to the right, but copies the sign bit; it divides by 2
- Let’s look at some examples.

Binary Multiplication by Shifting

Example:

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

00001011 (+11)

We shift left one place, resulting in:

00010110 (+22)

The sign bit has not changed, so the value is valid.

To multiply 11 by 4, we simply perform a left shift twice.

COMSCI INTER - BANGMOD -

Binary Multiplication by Shifting

Example:

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

00001100 (+12)

We shift left one place, resulting in:

00000110 (+6)

(Remember, we carry the sign bit to the left as we shift.)

To divide 12 by 4, we right shift twice.

COMSCI INTER - BANGMOD -

Signed binary numbers

- In a signed binary number system, the most significant bit (MSB) designates the sign, i.e.
  - MSB = ‘0’ ➔ positive number
  - MSB = ‘1’ ➔ negative number
- Example:

signed bit

$0\ \underbrace{1011100}_{\text{numerical value}}$

Signed binary formats

- Each format differs in how the numerical part is represented…
- From what you have learned so far, what format can be used to represented the numerical part?
  - 2's complement ➔ "Signed 2's complement"
  - 1's complement ➔ "Signed 1's complement"
  - Magnitude ➔ "Signed magnitude"

Examples

- +6 = 0110, -6 = ?
  - Signed 2's complement ➔ 1010
  - Signed 1's complement ➔ 1001
  - Signed magnitude ➔ 1110
- +1 = 001, -1 = ?
  - Signed 2's complement ➔
  - Signed 1's complement ➔
  - Signed magnitude ➔
- +6 = 000110, -6 = ?
  - Signed magnitude ➔
  - Signed 1's complement ➔
  - Signed 2's complement ➔

Note

- In the signed 1's complement and signed magnitude, there exist: +0 & -0

  +0 = 0000

  -0 = 1000

  -0 = 1111

- -0 is meaningless in the signed 2's complement. However, an n-bit signed 2's complement can represent a decimal values from $-2^{(n-1)}$ to $(2^{(n-1)}-1)$

COMSCI INTER - BANGMOD -

Possible Representations

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: (1) number of zeros, (3) ease of operations
- Which one is best?

Signed Integer Representation - Summary

- Although the "end carry around" adds some complexity, one's complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- Two's complement solves this problem.
- Two's complement is the radix complement of the binary numbering system; the *radix complement* of a non-zero number N in base r with d digits is $r^d - N$.

Binary codes

- Binary numbers we have studied so far are coded systematically, using bits '0' and '1'.
  - They represent a partial set of existing binary codes.
- Other binary codes exist that find their use in different applications.

COMSCI INTER - BANGMOD -

Binary-Coded Decimal (BCD)

93725

1001 0011 0111 0010 0101

Unsigned binary numbers

Thus

$93725 = (10010011011100100101)_2$

In BCD

Weighted codes

- In a weighted code, each bit position is assigned a weighting factor such that the sum of the product of each bit value and its weight equals the decimal digit it represents.
- The BCD is an example of a weighted code, whose weights are 8 4 2 1
  - BCD code is also called BCD 8421.

COMSCI INTER - BANGMOD -

BCD 8421

93725

1001 0011 0111 0010 0101

1(8) + 0(4) + 0(2) + 1(1) = 9

1(2) + 1(1) = 3

1(4) + 1(2) + 1(1) = 7

1(2) = 2

1(4) + 1(1) = 5

Gray code

- Gray codes for successive decimal digits differ in exactly one bit
- When converting continuous signals to digital form, Gray codes can help reduce faulty states during successive transition

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

ASCII Character Code

- Standard binary code for alphanumeric characters
- Uses 7 bits to code 128 characters
- Usually stored in computer as an 8-bit unit, a *byte*, where an extra bit is used for other purposes such as:
  - Providing additional 128 8-bit characters when MSB set to ‘1’
  - Recognized by printer as ASCII characters when MSB set to ‘0’
  - Providing a *parity bit*

COMSCI INTER - BANGMOD -

Parity Bit

- An extra bit included with a message to make the total number of 1's either *even* or *odd*
  - Helps detect transmission errors

	Even Parity	Odd Parity
ASCII A = 1000001	**0**1000001	**1**1000001
ASCII T = 1010100	**1**1010100	**0**1010100

Unicode

- Many of today's systems embrace Unicode, a 16-bit system that can encode the characters of every language in the world.
  - The Java programming language, and some operating systems now use Unicode as their default character code.
- The Unicode code space is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

Unicode Codespace

- The Unicode code space allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE