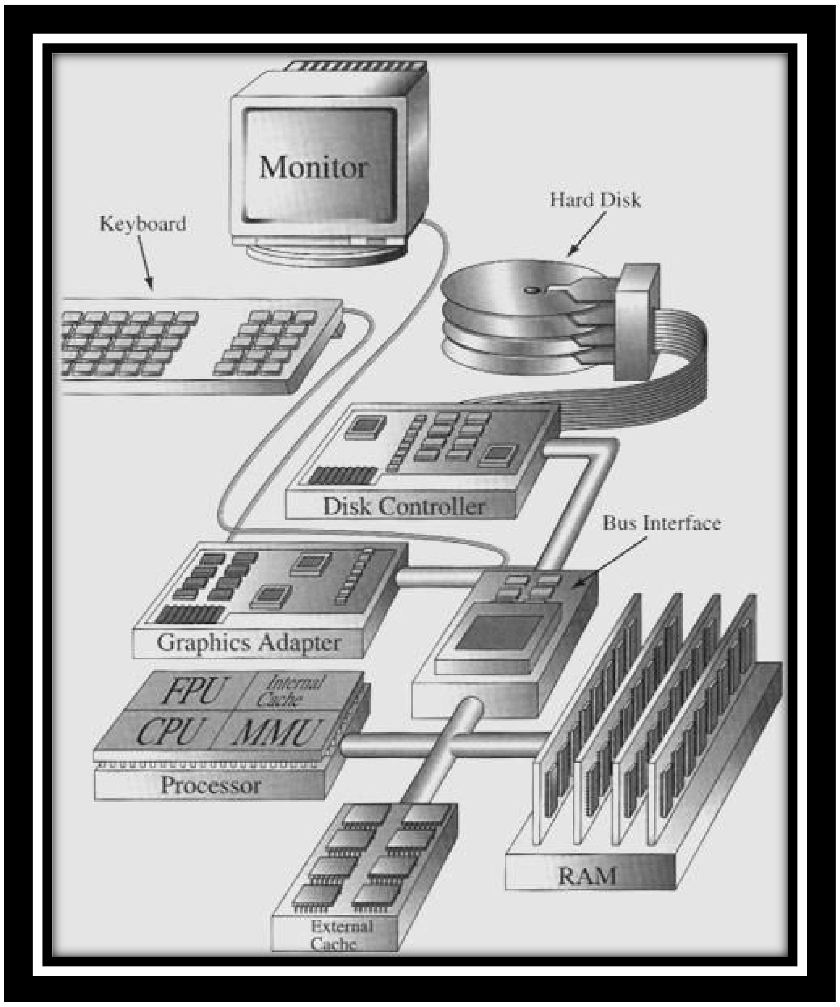
Condensed Notes
On
Computer Architecture
(According to BScCSIT syllabus, TU)



By Bishnu Rawal

Unit 1
Data Representation

Number System
Number of digits used in a number system is called its base or radix (r). We can categorize number system as below:

- Binary number system (r = 2)
- Octal Number System (r = 8)
- Decimal Number System (r = 10)
- Hexadecimal Number system (r = 16)

Number system conversions (quite easy guys, do it on your own)

Decimal Representation
We can normally represent decimal numbers in one of following two ways

- By converting into binary
- By using BCD codes

By converting into binary

Advantage

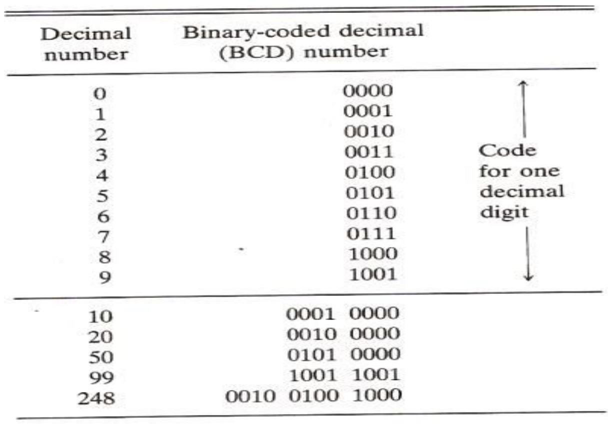
- Arithmetic and logical calculation becomes easy. Negative numbers can be represented easily.

Disadvantage

- At the time of input conversion from decimal to binary is needed and at the time of output conversion from binary to decimal is needed.

Therefore this approach is useful in the systems where there is much calculation than input/output.

By using BCD codes



Decimal number	Binary-coded decimal (BCD) number	
0	0000	↑
1	0001	
2	0010	
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111	
8	1000	
9	1001	↓
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

Disadvantage

- Arithmetic and logical calculation becomes difficult to do. Representation of negative numbers is tricky.

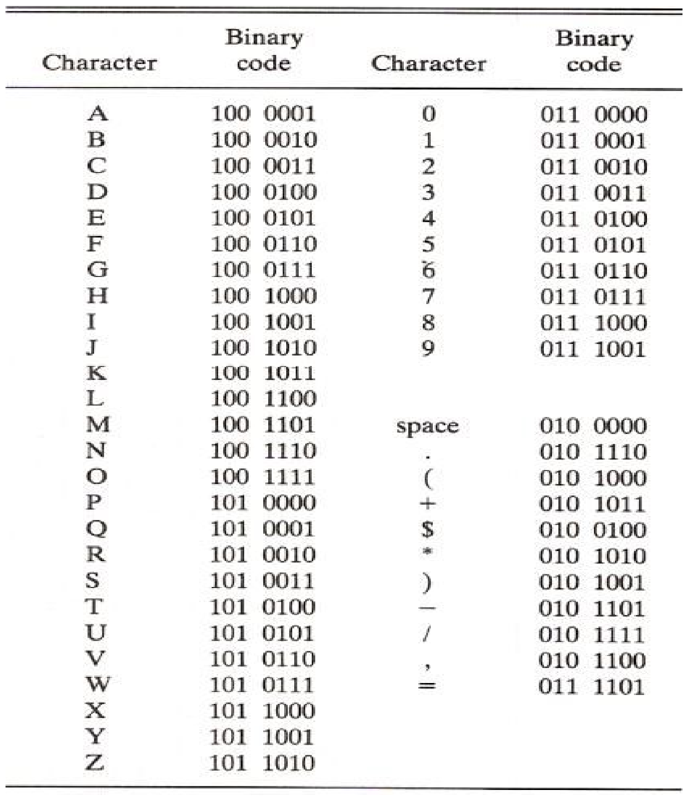
Advantage

- At the time of input conversion from decimal to binary and at the time of output conversion from binary to decimal is not needed.

Therefore, this approach is useful in the systems where there is much input/output than arithmetic and logical calculation.

Alphanumeric Representation

Alphanumeric character set is a set of elements that includes the 10 decimal digits, 26 letters of the alphabet and special characters such as $, %, + etc. The standard alphanumeric binary code is ASCII(American Standard Code for Information Interchange) which uses 7 bits to code 128 characters (both uppercase and lowercase letters, decimal digits and special characters).



Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	—	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

NOTE: Decimal digits in ASCII can be converted to BCD by removing the three higher order bits, 011.

Complements

(r-1)'s Complement

(r-1)'s complement of a number N is defined as $(r^n -1) - N$

Where **N** is the given number
r is the base of number system
n is the number of digits in the given number

To get the (r-1)'s complement fast, subtract each digit of a number from (r-1).

Example:

- 9's complement of 835_{10} is 164_{10} (Rule: $(10^n -1) - N$)
- 1's complement of 1010_2 is 0101_2 (bit by bit complement operation**)**

r's Complement

r's complement of a number N is defined as $r^n - N$

Where **N** is the given number
r is the base of number system
n is the number of digits in the given number

To get the r's complement fast, add 1 to the low-order digit of its (r-1)'s complement.

Example:

- 10's complement of 835_{10} is $164_{10} + 1 = 165_{10}$
- 2's complement of 10102 is $0101_2 + 1 = 0110_2$

Subtraction of unsigned Numbers (Using complements)

When subtraction is implemented in digital hardware, borrow-method is found to be less efficient than the method that uses complements. The subtraction of two n-digit unsigned numbers M-N (N≠0) in base r can be done as follows:

1. Add the minuend M to the r's complement of the subtrahend N. This performs $M + (r^n - N) = M - N + r^n$.
2. If $M \geq N$, the sum will produce an end carry r^n which is discarded, and what is left is the result $M - N$.
3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of $(N - M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Consider, for example, the subtraction 72532 − 13250 = 59282. The 10's complement of 13250 is 86750. Therefore:

$$\begin{array}{rr} M = & 72532 \\ \text{10's complement of } N = & +\underline{86750} \\ \text{Sum} = & 159282 \\ \text{Discard end carry } 10^5 = & -\underline{100000} \\ \text{Answer} = & 59282 \end{array}$$

Now consider an example with $M < N$. The subtraction 13250 − 72532 produces negative 59282. Using the procedure with complements, we have

$$\begin{array}{rr} M = & 13250 \\ \text{10's complement of } N = & +\underline{27468} \\ \text{Sum} = & 40718 \end{array}$$

There is no end carry, so answer is negative 59282 = 10's complement of 40718.

Subtraction with complements is done with binary numbers in similar manner using same procedure outlined above.

NOTE: negative numbers are recognized by the absence of the end carry and the complemented result.

Fixed-Point Representation

Positive integers, including 0 can be represented as unsigned numbers. However for negative numbers, we use convention of representing left most bit of a number as a sign-bit: 0 for positive and 1 for negative. In addition, to represent fractions, integers or mixed integer-fraction numbers, number may have a binary (or decimal) point. There are two ways of specifying the position of a binary point in a resister:

- by employing a floating-point notation.(discussed later)
- by giving it a fixed position (hence the name)
  - A binary point in the extreme left of the resister to make the stored number a fraction.
  - A binary point in the extreme right of a resister to make the stored number an integer.

Integer representation

There is only one way of representing positive numbers with sign-bit 0 but when number is negative the sign is represented by 1 and rest of the number may be represented in one of three possible ways:

- Signed magnitude representation

- Signed 1’s complement representation
- Signed 2’s complement representation

Signed magnitude representation of a negative number consists of the magnitude and a negative sign. In other two representations, the negative number is represented in either 1's or 2's complement of its positive value.

Examples: Representing negative numbers

Signed Magnitude Notation
- Complement only the sign bit
- Example:
  +9 → 0 001001
  -9 → 1 001001

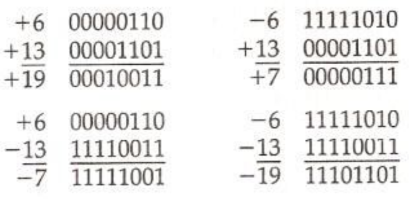
Signed 1’s complement Notation
- Complement *all* the bits including sign bit.
- Example:
  +9 → 0 001001
  -9 → 1 110110

Signed 2’s complement Notation
- Take the 2's complement of the number, *including* its sign bit
- Example:
  +9 → 0 001001
  -9 → 1 110111

Arithmetic addition and subtraction of signed numbers

Addition

Mostly signed 2's complement system is used. So, in this system only addition and complementation is used.
Procedure: add two numbers including sign bit and discard any carry out of the sign bit position. (note: negative numbers initially be in the 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form).



```
 +6  00000110      −6  11111010
+13  00001101     +13  00001101
+19  00010011      +7  00000111

 +6  00000110      −6  11111010
−13  11110011     −13  11110011
 −7  11111001     −19  11101101
```

In each of the 4 cases, the operation performed is always addition, including the sign-bits. Any carry out of the sign bit is discarded and negative results are automatically in 2's complement form.

Subtraction

Subtraction of two signed binary numbers is done as: take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign-bit). The carry out of the sign bit position is discarded.

Idea: subtraction operation can be changed to the addition operation if the sign of the subtrahend is changed:

$$(\pm A) - (+B) = (\pm A) + (-B)$$
$$(\pm A) - (-B) = (\pm A) + (+B)$$

Example: (-6)-(-13) = +7, in binary with 8-bits this is written as:

-6 → 11111010
-13 → 11110011 (2's complement form)

Subtraction is changed to addition by taking 2's complement of the subtrahend (-13) to give (+13).

-6 → 11111010
+13 → 00001101

+7 → 100000111 (discarding end carry).

<u>Overflow</u>

When two numbers of n digits are added and the sum occupies n+1 digits, we say that an overflow has occurred. A result that contains n+1 bits can't be accommodated in a resister with a standard length of n-bits. For this reason many computers detect the occurrence of an overflow setting corresponding flip-flop.

An overflow may occur if two numbers added are both positive or both negative. For example: two signed binary numbers +70 and +80 are stored in two 8-bit resisters.

```
carries: 0 1                 carries: 1 0
    +70    0 1000110             -70    1 0111010
    +80    0 1010000             -80    1 0110000
   +150    1 0010110            -150    0 1101010
```

Since the sum of numbers 150 exceeds the capacity of the resister (since 8-bit resister can store values ranging from +127 to -128), hence the overflow.

<u>Overflow detection</u>

An overflow condition can be detected by observing two carries: carry into the sign bit position and carry out of the sign bit position.

Hey boys, consider example of above 8-bit resister, if we take the carry out of the sign bit position as a sign bit of the result, 9-bit answer so obtained will be correct. Since answer can not be accommodated within 8-bits, we say that an overflow occurred.

If these two carries are equal ==> no overflow
If these two carries are not same ==> overflow condition is produced.

If two carries are applied to an exclusive-OR gate, an overflow will be detected when output of the gate is equal to 1.

Decimal Fixed-Point Representation

Decimal number representation = f(binary code used to represent each decimal digit). Output of this function is called the Binary coded Decimal (BCD). A 4-bit decimal code requires 4 flip-flops for each decimal digit.
Example: 4385 = $(0100\ 0011\ 1000\ 0101)_{BCD}$

While using BCD representation,
Disadvantages:

- wastage of memory (Viz. binary equivalent of 4385 uses less bits than its BCD representation)
- Circuits for decimal arithmetic are quite complex.

Advantages:

- Eliminate the need for conversion to binary and back to decimal. (since applications like Business data processing requires less computation than I/O of decimal data, hence electronic calculators perform arithmetic operations directly with the decimal data (in binary code))

For the representation of signed decimal numbers in BCD, sign is also represented with 4-bits, plus with 4 0's and minus with 1001 (BCD equivalent of 9). Negative numbers are in 10's complement form.

RULE: While adding if nibble (4-bit)-result > 9, 0110 (15-9 = 6) is added to the result and formatting 5-bit ultimate sum with two nibbles, we get the actual BCD sum. On 8-bit sum (two nibbles), most significant nibble is propagated to next nibble position as **carry** and **residue** (Least significant nibble) will be the result for that position.

Consider the addition: (+375) + (-240) = +135 [0→positive, 9→negative in case of radix 10]

0 375 $(0000\ 0011\ 0111\ 0101)_{BCD}$

+ 9 760 $(1001\ 0111\ 0110\ 0000)_{BCD}$

-- --

0 135 $(0000\ 0001\ 0011\ 0101)_{BCD}$ (Will be explained in class)

Floating-Point Representation

The floating-point representation of a number has two parts: *mantissa* and *exponent*

Mantissa : represents a signed, fixed-point number. May be a fraction or an integer

Exponent: designates the position of the decimal (or binary) point

Example1: decimal number +6132.789 is represented in floating-point as:

Fraction	exponent
+0.6132789	+04

Floating-point is interpreted to represent a number in the form: $m * r^e$. Only the mantissa *m* and exponent *e* are physically represented in resisters. The radix *r* and the radix-point position are always assumed.

Example2: binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as,

Fraction	exponent
+01001110	000100

or equivalently,

$$m * 2^e = +(.1001110)_2 * 2^{+4}$$

<u>Normalization</u>

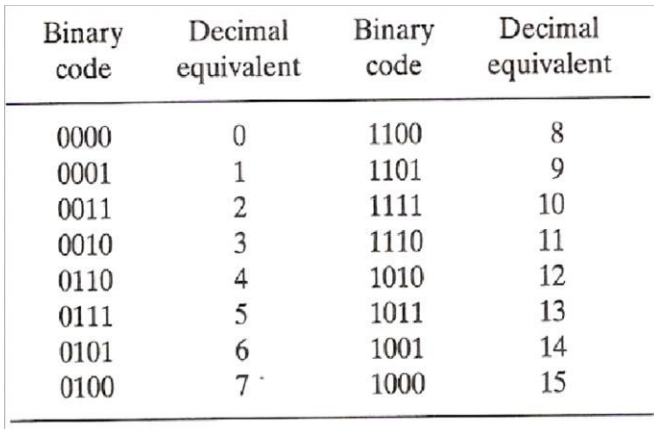
A floating-point number is said to be *normalized* if the most significant digit of the mantissa is nonzero. For example, decimal number 350 is normalized but 00035 is not.

Other Binary codes

Most common type of binary-coded data found in digital computer is explained before. A few additional binary codes used in digital systems (for special applications) are explained below.

Gray code

The reflected binary or Gray code is used to represent digital data converted from analog information. Gray code changes by only one bit as it sequences from one number to the next.



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

Table: 4-bit Gray code

Weighted code (2421)

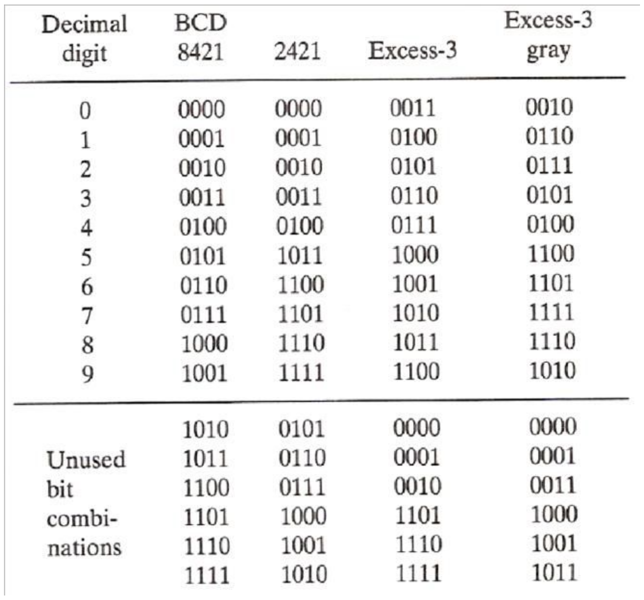
2421 is an example of weighted code. In this, corresponding bits are multiplied by the weights indicated and the sum of the weighted bits gives the decimal digit.

Example: 1101 when weighted by the respective digits 2421 gives 2*1+4*1+2*0+1*1 = 7.

NOTE: Ladies and gentlemen…☺, you have already studied about BCD codes. BCD can be assigned the weights 8421 and for this reason it is sometimes called 8421 code.

Excess-3 codes
The excess-3 code is a decimal code used in older computers. This is un-weighted code.
Excess-3 code = BCD binary equivalent + 3(0011)
NOTE: excess-n code is possible adding n to the corresponding BCD equivalent.

Excess-3 Gray
In ordinary Gray code, the transition from 9 back to 0 involves a change of three bits (from 1101 to 0000). To overcome this difficulty, we start from third entry 0010 (as first number) up to the twelfth entry 1010, there by change of only one bit is possible upon transition from 1010 to 0010. Since code has been shifted up three numbers, it is called the excess-3 Gray.



Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
	1010	0101	0000	0000
Unused	1011	0110	0001	0001
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1000
nations	1110	1001	1110	1001
	1111	1010	1111	1011

Table: 4 different binary codes for the decimal digit

Error Detection Codes

Binary information transmitted through some form of communication medium is subject to external noise that could change bits from 1 to 0 and vice versa. An error detection code is a binary code that detects digital errors during transmission. The detected errors can not be corrected but their presence is indicated. The most common error detection code used is the *parity bit*. A parity bit(s) is an extra bit that is added with original message to detect error in the message during data transmission.

Even Parity
One bit is attached to the information so that the total number of 1 bits is an even number.

Message	Parity
1011001	0
1010010	1

Odd Parity

One bit is attached to the information so that the total number of 1 bits is an odd number.

Message	Parity
1011001	1
1010010	0

Parity generator

Parity generator and checker networks are logic circuits constructed with exclusive-OR functions. Consider a 3-bit message to be transmitted with an odd parity bit. At the sending end, the odd parity is generated by a parity generator circuit. The output of the parity checker would be 1 when an error occurs i.e. no. of 1's in the four inputs is even.

P = $\overline{x \oplus y \oplus z}$

Message (xyz)	Parity bit (odd)
000	1
001	0
010	0
011	1
100	0
101	1
110	1
111	0

Parity Checker

Considers original message as well as parity bit

e = $\overline{p \oplus x \oplus y \oplus z}$

e= 1 => No. of 1's in pxyz is even => Error in data

e= 0 => No. of 1's in pxyz is odd => Data is error free

Circuit diagram for parity generator and parity checker

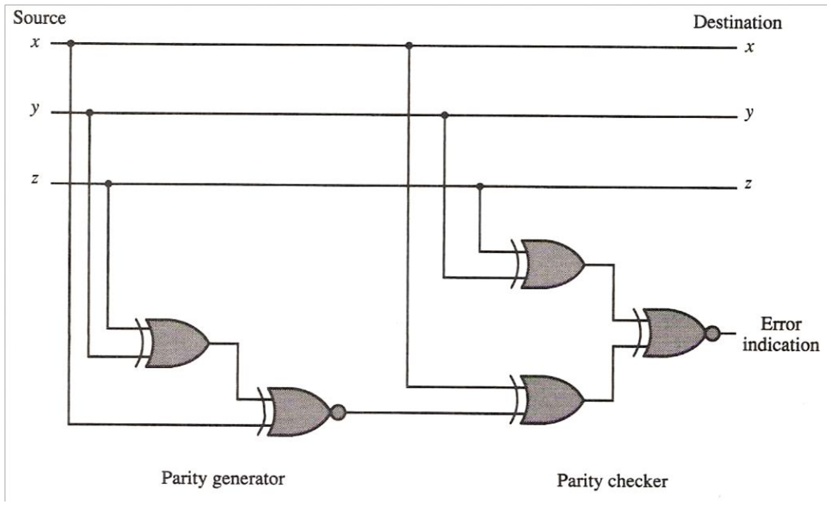


Fig: Error detection with odd parity bit.

EXERCISES: Text Book chapter3➔ 3.15, 3.17, 3.22, 3.26

3.15 (Solution)

(a)	(b)	(c)	(d)
11010	11010	000100	1010100
+10000	10011	010000	0101100
1)01010	1)01101	0)010100	1)0000000
		↓	
(26-16=10)	(26-13=13)	-101100	(84-84=0)
		(4-48=-44)	

3.17 HINT: see notes

3.22 (Solution)

(a)	BCD	1000	0110	0010	0000
(b)	XS-3	1011	1001	0101	0011
(c)	2421	1110	1100	0010	0000
(d)	Binary	10000110101100 (8192+256+128+32+8+4)			

3.26 (Solution)

Same as in Fig. 3-3 but without the complemented circles in the outputs of the gates.

$$P = x \oplus y \oplus z$$

$$\text{Error} = x \oplus y \oplus z \oplus P$$