Computer System Architecture

UNIT I

Number System

The Number Systems used in computers are:

- Binary number system
- Octal number system
- Decimal number system
- Hexadecimal number system

Binary number system

- It has only two digits '0' and '1' so its base is **2**.
- Accordingly, In this number system, there are only two types of electronic pulses; absence of electronic pulse which represents '0'and presence of electronic pulse which represents '1'.
- Each digit is called a bit.
- A group of four bits (1101) is called a **nibble** and group of eight bits (11001010) is called a **byte**. The position of each digit in a binary number represents a specific power of the base (2) of the number system.

Octal number system

- It has eight digits (0, 1, 2, 3, 4, 5, 6, 7) so its base is **8**. Each digit in an octal number represents a specific power of its base (8).
- As there are only eight digits, three bits (2^3=8) of binary number system can convert any octal number into binary number.
- This number system is also used to shorten long binary numbers. The three binary digits can be represented with a single octal digit.

Decimal number system

- This number system has ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) so its base is **10**.
- In this number system, the maximum value of a digit is 9 and the minimum value of a digit is 0.
- The position of each digit in decimal number represents a specific power of the base (10) of the number system.

- This number system is widely used in our day-to-day life. It can represent any numeric value.

Hexadecimal number system

- This number system has 16 digits that ranges from 0 to 9 and A to F. So, its base is **16**. The A to F alphabets represent 10 to 15 decimal numbers.
- The position of each digit in a hexadecimal number represents a specific power of base (16) of the number system.
- As there are only sixteen digits, four bits (2^4=16) of binary number system can convert any hexadecimal number into binary number.
- It is also known as alphanumeric number system as it uses both numeric digits and alphabets.

Binary	Octal	Decimal	Hexadecimal
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F
10000	20	16	10
10001	21	17	11
10010	22	10	12
10011	23	19	13
10100	24	20	14
10101	25	21	15
10110	26	22	16
10111	27	23	17
11000	30	24	18
11001	31	25	19
11010	32	26	1A

11011	33	27	1B
11100	34	28	1C
11101	35	29	1D
11110	36	30	1E

Fixed Point Representation

- Fixed-point representation has a radix point known as **decimal point**.
- Fixed-point numbers having decimal points at the right end of the number are treated as **integers** and the fixed-point numbers having decimal points at the left end of the number are treated as **fractions**.
- In this method, the decimal point position is settled because the number saved in the memory is considered as an integer or as a fraction.
- The binary numbers that are unsigned are continually considered as positive integers and are defined as 0s in the MSB.
- The binary numbers that are registered contrast for negative numbers and are defined as 1s in the MSB.

The magnitude of the signed binary numbers can be described using three methods are as follows –

1. **Sign and Magnitude Representation**

- In this method, the leftmost bit in the number is used for denoting the sign
- 0 denotes a positive integer, and 1 denotes a negative integer.
- The remaining bits in the number provide the magnitude of the number.
- The range of values for the sign and magnitude representation is from **-127 to 127.**

Example: (-2410) is defined as 10011000

Note:- The magnitude for both positive and negative values is equal, but they alter only with their signs.

2. **Signed 1's Complement Representation**

- In this representation, a negative value is received by taking the 1's complement of the equivalent positive number.
- It can add a signed 1's complement method creates end carry during arithmetic operation that cannot be rejected.
- The range of values for the signed 1's complement representation is from **-127 to 128.**

Example:

$(29)_{10} = (00011101)_2$ = 0000111011's complement for positive value

$-(29)_{10} = -(00011101)_2$ = 111100010 1's complement for negative value

3. Signed 2's Complement Representation

- In signed 2's complement representation, the 2's complement of a number is discovered by first creating the 1's complement of that number, then incrementing the result by 1.
- The range of values for the signed 2's complement representation is from **-128 to 127.**

Example:

$(29)_{10}$= $(00011100)_2$ = (000011100)2's1's complement for positive value

$-(29)_{10} = -(00011100)_2$ = (11110010)2's1's complement for negative value

1's Complement

- The binary numbers can be easily converted into the 1's complement with the help of a simple algorithm.
- According to this algorithm, if we toggle or invert all bits of a binary number, the generated binary number will become the 1's complement of that binary number. That means we have to transform 1 bit into the 0 bit and 0 bit into the 1 bit in the 1's complement.
- **N'** is used to indicate the 1's complement of a number.

Example:

1. 1's complement of binary number **5 (0101)** is binary number **10 (1010)**
2. 1's complement of binary number **13 (1101)** is binary number **2 (0010)**

2's Complement

- The binary numbers can also be easily converted into the 2's complement with the help of a very simple algorithm.
- According to this algorithm, we can get the 2's complement of a binary number by first inverting the given number. After that, we have to add 1 into the LSB (least significant bit). That means we have to first perform 1's complement of a number, and then we have to add 1 into that number to get the 2's complement.

- **N*** is used to show the 2's complement of a number.

Example:

1. 2's complement of binary number **5 (0101)** is binary number **11 (1011)**
2. 2's complement of binary number **13 (1101)** is binary number **3 (0011**)

Difference between 1's Complement and 2's Complement:

Parameters	1's complement 2representation	2's complement representation
Process of Generation	We can get the 1's complement of a given binary number by toggling or inverting all bits of that number.	We can get the 2's complement of a given binary number by first doing 1's complement of number and then adding 1 into that number.
Example	The 1's complement of binary number **9 (1001)** is **6 (0110).**	The 2's complement of binary number 9(1001) will be got by doing 1's complement of that number which is 6 (0110), and then adding 1 into it, which means 7 (0111). So the 2's complement of **9 (1001)** is **7 (0111).**
Logic Gates Used	The implementation of 1's complement is very simple. For every bit of input, it basically uses the **NOT gate.**	For every bit of input, the 2's complement basically uses the **NOT gate and a full adder.**
Number Representation	If we want to represent the sign binary number, we can use the 1's. If we have a number 0, then it will not be possible to use it in the form of ambiguous representation.	If we want to represent the sign binary number, we can also use the 2's. If we have a number 0, then it will possible to use it as an unambiguous representation of all given numbers.
K-bits Register	If there is a k-bit register, the 1's complement will use $-(2^{(k-1)} -1)$ to store the lowest negative number,	If there is a k-bit register, the 2's complement will use $-(2^{(k-1)})$ to store the lowest negative number and $(2^{(k-1)} -$

	and $(2^{(k-1)} -1)$ to store the largest positive number.	1) to store the largest positive number.
Representation of 0	There are two ways to represent the number 0 in 1's complement, i.e., +0 and -0. The plus 0 will be represented as 00000000, which is positive zero (+0) in an 8-bit register, and for negative zero (-0), it will be represented as 11111111 in an 8-bit register.	There is only one way to represent the number 0 in 2's complement for both +0 and -0. Both minus 0 or plus 0 can be represented as 0000000 (+0) in an 8-bit register because if we add 1 to 11111111 (-1), we will get 00000000 (+0), which is the same as positive zero. That's why the number 0 is always considered as a positive in the 2's complement. This is also the reason we generally use 2's complement.
Sign Extension	In the 1's complement, sign extension is used to convert the given sign into another sign for any signed integer.	The working of sign extension in 2's complement and in 1's complement is the same. Here it also converts a given sign into another sign for any signed integer.
Ease of operation	The 1's complement always requires the addition of end-around-carry-bit. That's why the 1's complement arithmetic operation is difficult as compared to the 2's complement arithmetic operation.	The 2's complement does not require the addition of end-around-carry-bit. That's why the 2's complement arithmetic operation is easier as compared to the 1's complement arithmetic operation.

Arithmetic Operations of Binary Numbers

Binary is a base-2 number system that uses two states 0 and 1 to represent a number. We can also call it to be a true state and a false state. A binary number is built the same way as we build the normal decimal number.

Binary arithmetic is an essential part of various digital systems. You can add, subtract, multiply, and divide binary numbers using various methods. These operations are much easier than decimal number arithmetic operations because the binary system has only two digits: 0 and 1.

Binary additions and subtractions are performed as same in decimal additions and subtractions. When we perform binary additions, there will be two outputs: **Sum (S)** and **Carry (C).**

1. There are four rules for binary addition:

Input A	Input B	Sum (S) A+B	Carry (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2. There are four rules for binary subtraction:

Input A	Input B	Subtract (S) A-B	Borrow (B)
0	0	0	0
0	1	0	1
1	0	1	0
1	1	0	0

Borrow 1 is required from the next higher order bit to subtract 1 from 0. So, the result became 0.

3. There are four rules for binary multiplication:

Input A	Input B	Multiply (M) AxB
0	0	0
0	1	0
1	0	0
1	1	1

Multiplication is always 0, whenever at least one input is 0.

4. There are four parts in any division: Dividend, Divisor, quotient, and remainder.

Input A	Input B	Divide (D) A/B
0	0	Not defined
0	1	0
1	0	Not defined
1	1	1

The result is always not defined, whenever the divisor is 0.

ASCII Codes

- ASCII (American Standard Code for Information Interchange) is the most common character encoding format for text data in computers and on the internet.
- In standard ASCII-encoded data, there are unique values for 128 alphabetic, numeric or special additional characters and control codes.
- ASCII encoding is based on character encoding used for telegraph data. The American National Standards Institute first published it as a standard for computing in 1963.

- Characters in ASCII encoding include upper- and lowercase letters A through Z, numerals 0 through 9 and basic punctuation symbols. It also uses some non-printing control characters that were originally intended for use with teletype printing terminals.
- In total, there are 256 ASCII characters, and can be broadly divided into three categories:
  - ASCII control characters (0-31 and 127)
  - ASCII printable characters (32-126) (most commonly referred)
  - Extended ASCII characters (128-255)

Below are the ASCII values of printable characters (33, 126):

Character	Character Name	ASCII code		Character	Character Name	ASCII code		Character	Character Name	ASCII code
!	Exclamation point	33		A	Uppercase a	65		a	Lower case a	97
“	Double quotation	34		B	Uppercase b	66		b	Lower case b	98
#	Number sign	35		C	Uppercase c	67		c	Lower case c	99
$	Dollar sign	36		D	Uppercase d	68		d	Lower case d	100
%	Percent sign	37		E	Uppercase e	69		e	Lower case e	101
&	ampersand	38		F	Uppercase f	70		f	Lower case f	102

Character	Character Name	ASCII code		Character	Character Name	ASCII code		Character	Character Name	ASCII code
‘	apostrophe	39		G	Uppercase g	71		g	Lower case g	103
(	Left parenthesis	40		H	Uppercase h	72		h	Lower case h	104
)	Right parenthesis	41		I	Uppercase i	73		i	Lower case i	105
*	asterisk	42		J	Uppercase j	74		j	Lower case j	106
+	Plus sign	43		K	Uppercase k	75		k	Lower case k	107
,	comma	44		L	Uppercase l	76		l	Lower case l	108
–	hyphen	45		M	Uppercase m	77		m	Lower case m	109
.	period	46		N	Uppercase n	78		n	Lower case n	110
/	slash	47		O	Uppercase o	79		o	Lower case o	111

Character	Character Name	ASCII code		Character	Character Name	ASCII code		Character	Character Name	ASCII code
0	zero	48		P	Uppercase p	80		p	Lower case p	112
1	one	49		Q	Uppercase q	81		q	Lower case q	113
2	two	50		R	Uppercase r	82		r	Lower case r	114
3	three	51		S	upperc ases	83		s	Lower case s	115
4	four	52		T	Uppercase t	84		t	Lower case t	116
5	five	53		U	Uppercase u	85		u	Lower case u	117
6	six	54		V	Uppercase v	86		v	Lower case v	118
7	seven	55		W	Uppercase w	87		w	Lower case w	119
8	eight	56		X	Uppercase x	88		x	Lower case x	120
9	nine	57		Y	Uppercase y	89		y	Lower case y	121

Character	Character Name	ASCII code		Character	Character Name	ASCII code		Character	Character Name	ASCII code
:	colon	58		Z	Uppercase z	90		z	Lower case z	122
;	semi-colon	59		[	Left square bracket	91		{	Left curly brace	123
<	Less-than sign	60		\	backslash	92		|	Vertical bar	124
=	Equals sign	61		]	Right square bracket	93		}	Right curly brace	125
>	Greater-than sign	62		^	caret	94		~	tilde	126
?	Question mark	63		_	underscore	95				
@	At sign	64		`	Grave accent	96				

EBCDIC codes

- Extended binary coded decimal interchange code (EBCDIC) is an 8-bit binary code for numeric and alphanumeric characters.
- It was developed and used by IBM. It is a coding representation in which symbols, letters and numbers are presented in binary language.

BCD or Binary Coded Decimal

- Binary Coded Decimal, or BCD, is another process for converting decimal numbers into their binary equivalents.
- It is a form of binary encoding where each digit in a decimal number is represented in the form of bits.
- This encoding can be done in either 4-bit or 8-bit (usually 4-bit is preferred).
- It is a fast and efficient system that converts the decimal numbers into binary numbers as compared to the existing binary system.
- These are generally used in digital displays where the manipulation of data is quite a task.
- Thus BCD plays an important role here because the manipulation is done treating each digit as a separate single sub-circuit.
- The BCD equivalent of a decimal number is written by replacing each decimal digit in the integer and fractional parts with its 4-bit binary equivalent.
- The BCD code is more precisely known as 8421 BCD code , with 8,4,2 and 1 representing the weights of different bits in the four-bit groups, Starting from MSB and proceeding towards LSB. This feature makes it a weighted code , which means that each bit in the 4-bit group representing a given decimal digit has an assigned weight.

DECIMAL NUMBER	BCD	DECIMAL NUMBER	BCD
0	0000	**8**	1000
1	0001	**9**	1001
2	0010	**10**	0001 0000
3	0011	**11**	0001 0001
4	0100	**12**	0001 0010
5	0101	**13**	0001 0011

6	0110	**14**	0001 0100
7	0111	**15**	0001 0101
16	0001 0110	**21**	0010 0001
17	0001 0111	**22**	0010 0010
18	0001 1000	**23**	0010 0011
19	0001 1001	**24**	0010 0100
20	0010 0000	**25**	0010 0101

In the BCD numbering system, the given decimal number is segregated into chunks of four bits for each decimal digit within the number. Each decimal digit is converted into its direct binary form (usually represented in 4-bits).

For example:

1. Convert $(123)_{10}$ in BCD

From the truth table above,
1 -> 0001
2 -> 0010
3 -> 0011
thus, BCD becomes -> **0001 0010 0011**

2. Convert $(324)_{10}$ in BCD

(324)10 -> 0011 0010 0100 (BCD)
3 -> 0011
2 -> 0010
4 -> 0100
thus, BCD becomes -> **0011 0010 0100**

Note:-

1. It is noticeable that the BCD is nothing more than a binary representation of each digit of a decimal number.
2. It cannot be ignored that the BCD representation of the given decimal number uses extra bits, which makes it heavy-weighted.

Gray Code

- The Gray Code is a sequence of binary number systems, which is also known as reflected binary code. The reason for calling this code as

reflected binary code is the first N/2 values compared with those of the last N/2 values in reverse order.

- In this code, two consecutive values are differed by one bit of binary digits.
- Gray codes are used in the general sequence of hardware-generated binary numbers.
- These numbers cause ambiguities or errors when the transition from one number to its successive is done.
- This code simply solves this problem by changing only one bit when the transition is between numbers is done.
- The gray code is a very light weighted code because it doesn't depend on the value of the digit specified by the position. This code is also called a cyclic variable code as the transition of one value to its successive value carries a change of one bit only.

Decimal Number	Binary Number	Gray Code
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

Excess-3 Code

- The excess-3 code is also treated as **XS-3** code. The excess-3 code is a non-weighted and self-complementary BCD code used to represent the decimal numbers.
- This code has a biased representation. This code plays an important role in arithmetic operations because it resolves deficiencies encountered when we use the BCD code for adding two decimal digits whose sum is greater than 9.
- The Excess-3 code uses a special type of algorithm, which differs from the binary positional number system or normal non-biased BCD.
- We can easily get an excess-3 code of a decimal number by simply adding 3 to each decimal digit. And then we write the 4-bit binary number for each digit of the decimal number.

The Excess-3 code for the decimal number is as follows:

Decimal Digit	BCD Code	Excess-3 Code
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

Note:-

In excess-3 code, the codes 1111 and 0000 are never used for any decimal digit.

Example 1: $(31)_{10}$

1) We find the BCD code of each digit of the decimal number.

Digit	BCD
3	0011
1	0001

2) Then, we add 0011 in both of the BCD code.

Decimal	BCD	Excess-3
3	0011+0011	0110
1	0001+0011	0100

3) So, the excess-3 code of the decimal number 31 is 0110 0100

Example 2: $(81.61)_{10}$

1. We find the BCD code of each digit of the decimal number.

Digit	BCD
8	1000
1	0001
6	0110
1	0001

Then, we add 0011 in both of the BCD code.

Decimal	BCD	Excess-3
8	1000+0011	1011
1	0001+0011	0100
6	0110+0011	1001

So, the excess-3 code of the decimal number 81.61 is 1011 0100.1001 0100

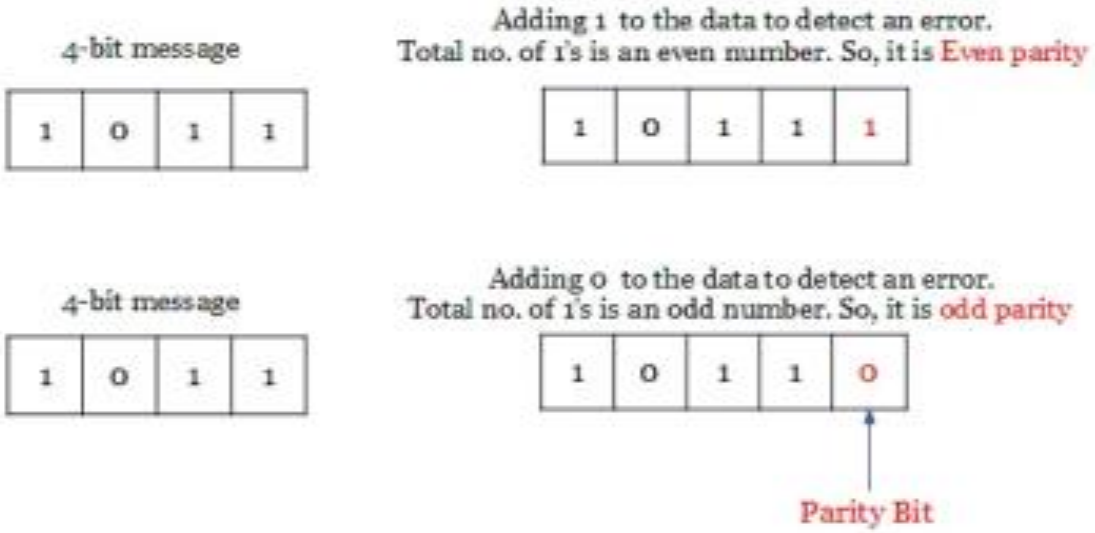
Error Detection & Correction Codes

(Parity Code)

- The parity code is used for the purpose of detecting errors during the transmission of binary information. The parity code is a bit that is included with the binary data to be transmitted.
- The inclusion of a parity bit will make the number of 1's either odd or even.

Based on the number of 1's in the transmitted data, the parity code is of two types.

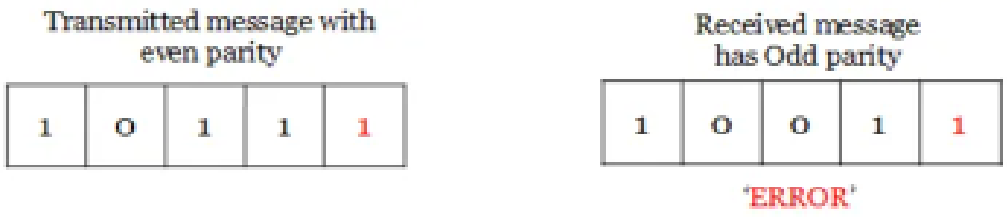
1. **Even parity code** - In even parity, the added parity bit will make the total number of 1's an even number.
2. **Odd parity code** - If the added parity bit make the total number of 1's as odd number, such parity code is said to be odd parity code.



The following table shows the some of the 4-bit messages to be transmitted along with the parity bits. The bits in bold are the parity bits.

4-bit message	Message with Odd parity	Message with Even Parity
0000	0000**1**	0000**0**
0001	0001**0**	0001**1**
0010	0010**0**	0010**1**
0011	0011**1**	0011**0**
0100	0100**0**	0100**1**
0101	0101**1**	0101**0**
0110	0110**1**	0110**0**
0111	0111**0**	0111**1**

On the receiver side, if the received data is other than the sent data, then it is an error. If the sent date is even parity code and the received data is odd parity, then there is an error.



Hamming Code

- Hamming code is a block code that is capable of detecting up to two simultaneous bit errors and correcting single-bit errors. It was developed by **R.W. Hamming** for error correction.
- Hamming code is a set of error-correction codes that can be used to detect and correct the errors that can occur when the data is moved or stored from the sender to the receiver.
- In this coding method, the source encodes the message by inserting redundant bits within the message. These redundant bits are extra bits that are generated and inserted at specific positions in the message itself to enable error detection and correction. When the destination receives this message, it performs recalculations to detect errors and find the bit position that has error.
- The number of redundant bits can be calculated using the following formula:

$$2^r \geq m + r + 1$$

where, r = redundant bit, m = data bit

Encoding a message by Hamming Code:

The procedure used by the sender to encode the message encompasses the following steps –

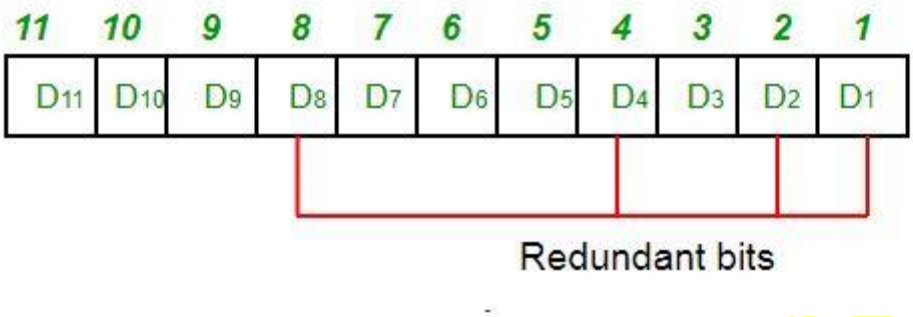
Step 1 – Calculation of the number of redundant bits.

If the message contains **m** number of data bits, **r** number of redundant bits are added to it so that $\mathbf{m}_r$ is able to indicate at least **(m + r+ 1)** different states. Here, **(m + r)** indicates location of an error in each of $(m + r)$ bit positions and one additional state indicates **no error**. Since, **r** bits can indicate

2^r states, 2^r must be at least equal to (**m + r + 1**). Thus the following equation should hold **$2^r \geq m+r+1$.**

Step 2 – Positioning the redundant bits.

The **r** redundant bits placed at bit positions of powers of 2, i.e. 1, 2, 4, 8, 16 etc. They are referred in the rest of this text as r_1 (at position 1), r_2 (at position 2), r_3 (at position 4), r_4 (at position 8) and so on.



Step 3 – Calculating the values of each redundant bit.

The redundant bits are **parity bits**. A parity bit is an extra bit that makes the number of 1s either even or odd. The two types of parity are –

Even Parity – Here the total number of bits in the message is made even.

Odd Parity – Here the total number of bits in the message is made odd.

Each redundant bit, r_i, is calculated as the parity, generally even parity, based upon its bit position. It covers all bit positions whose binary representation includes a 1 in the i^{th} position except the position of r_i.

Thus –

- r_1 is the parity bit for all data bits in positions whose binary representation includes a 1 in the least significant position excluding 1 (3, 5, 7, 9, 11 and so on)
- r_2 is the parity bit for all data bits in positions whose binary representation includes a 1 in the position 2 from right except 2 (3, 6, 7, 10, 11 and so on)
- r_3 is the parity bit for all data bits in positions whose binary representation includes a 1 in the position 3 from right except 4 (5-7, 12-15, 20-23 and so on)

Once the redundant bits are embedded within the message, this is sent to the user.

Decoding a message in Hamming Code:

Once the receiver gets an incoming message, it performs recalculations to detect errors and correct them. The steps for recalculation are –

Step 1 – Calculation of the number of redundant bits

Using the same formula as in encoding, the number of redundant bits are ascertained.

$$2^r \geq m + r + 1$$

where m is the number of data bits and r is the number of redundant bits.

Step 2 – Positioning the redundant bits

The **r** redundant bits placed at bit positions of powers of 2, i.e. 1, 2, 4, 8, 16 etc.

Step 3 – Parity checking

Parity bits are calculated based upon the data bits and the redundant bits using the same rule as during generation of c1,c2 ,c3 ,c4 etc. Thus

c1 = parity(1, 3, 5, 7, 9, 11 and so on)

c2 = parity(2, 3, 6, 7, 10, 11 and so on)

c3 = parity(4-7, 12-15, 20-23 and so on)

Step 4 – Error detection and correction

The decimal equivalent of the parity bits binary values is calculated. If it is 0, there is no error. Otherwise, the decimal value gives the bit position which has error. For example, if **c1c2c3c4 = 1001**, it implies that the data bit at position 9, decimal equivalent of 1001, has error. The bit is flipped to get the correct message.

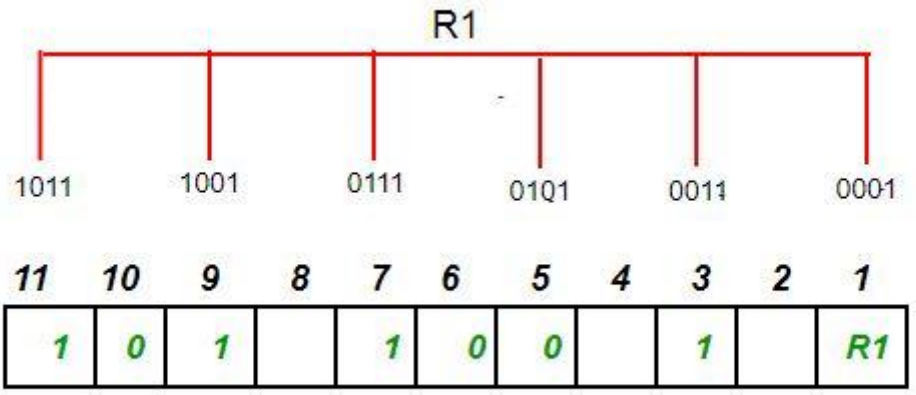
Example:

Suppose the data to be transmitted is 1011001, the bits will be placed as follows:

11	10	9	8	7	6	5	4	3	2	1
1	0	1	R8	1	0	0	R4	1	R2	R1

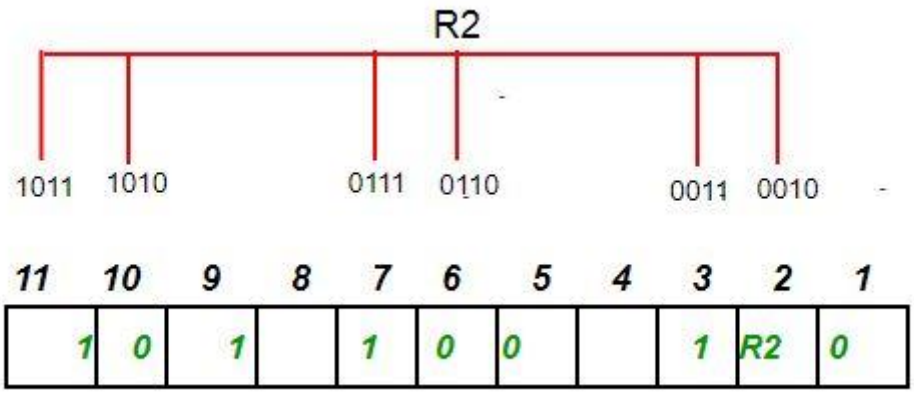
Determining the Parity bits:

R1 bit is calculated using parity check at all the bits positions whose binary representation includes a 1 in the least significant position. R1: bits 1, 3, 5, 7, 9, 11



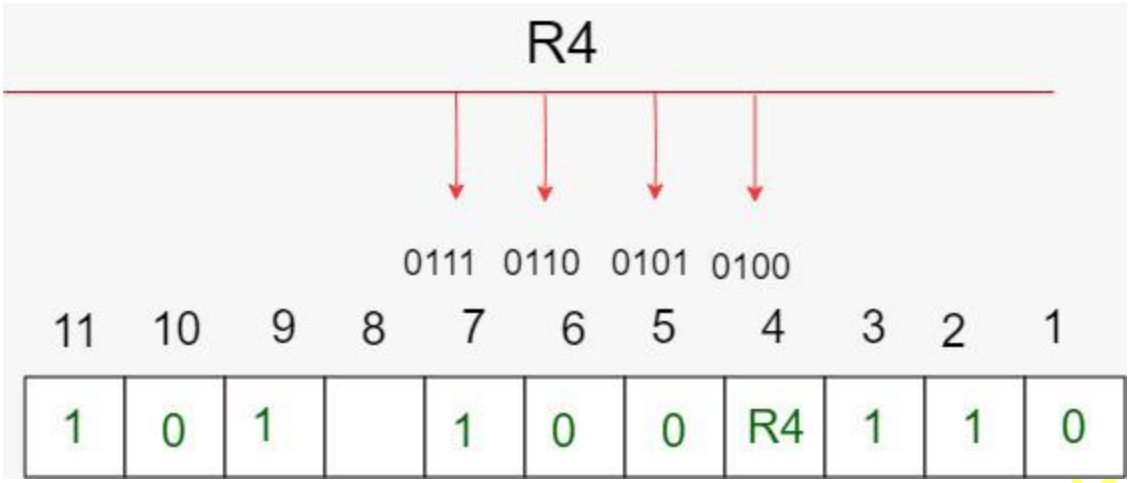
To find the redundant bit R1, we check for even parity. Since the total number of 1's in all the bit positions corresponding to R1 is an even number the value of R1 (parity bit's value) = 0

R2 bit is calculated using parity check at all the bits positions whose binary representation includes a 1 in the second position from the least significant bit.
R2: bits 2,3,6,7,10,11



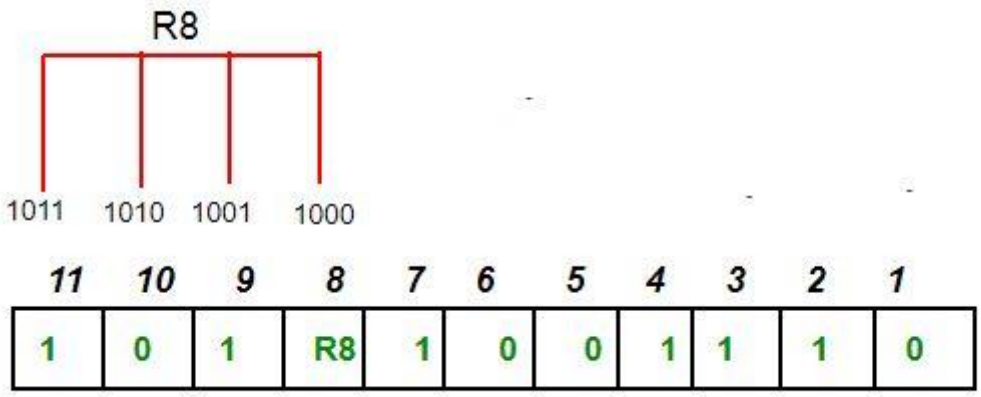
To find the redundant bit R2, we check for even parity. Since the total number of 1's in all the bit positions corresponding to R2 is odd the value of R2(parity bit's value)=1

R4 bit is calculated using parity check at all the bits positions whose binary representation includes a 1 in the third position from the least significant bit.
R4: bits 4, 5, 6, 7

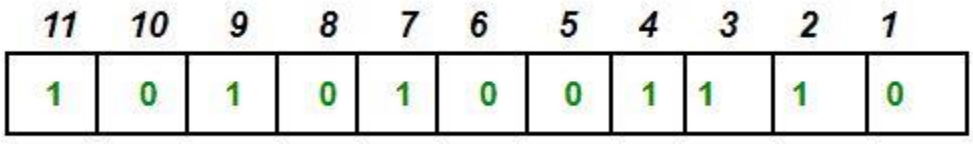


To find the redundant bit R4, we check for even parity. Since the total number of 1's in all the bit positions corresponding to R4 is odd the value of R4(parity bit's value) = 1

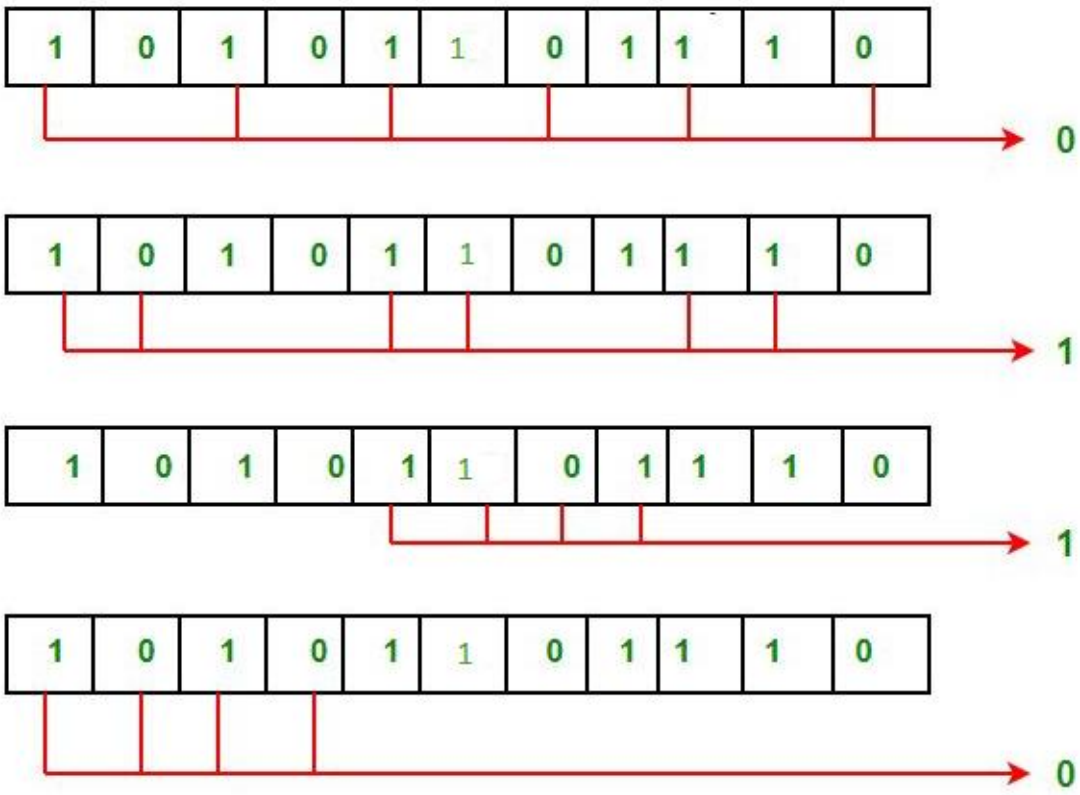
R8 bit is calculated using parity check at all the bits positions whose binary representation includes a 1 in the fourth position from the least significant bit.
R8: bit
8,9,10,11



To find the redundant bit R8, we check for even parity. Since the total number of 1's in all the bit positions corresponding to R8 is an even number the value of R8(parity bit's value)=0. Thus, the data transferred is:



Error detection and correction: Suppose in the above example the 6th bit is changed from 0 to 1 during data transmission, then it gives new parity values in the binary number:



The bits give the binary number 0110 whose decimal representation is 6. Thus, bit 6 contains an error. To correct the error the 6th bit is changed from 1 to 0.