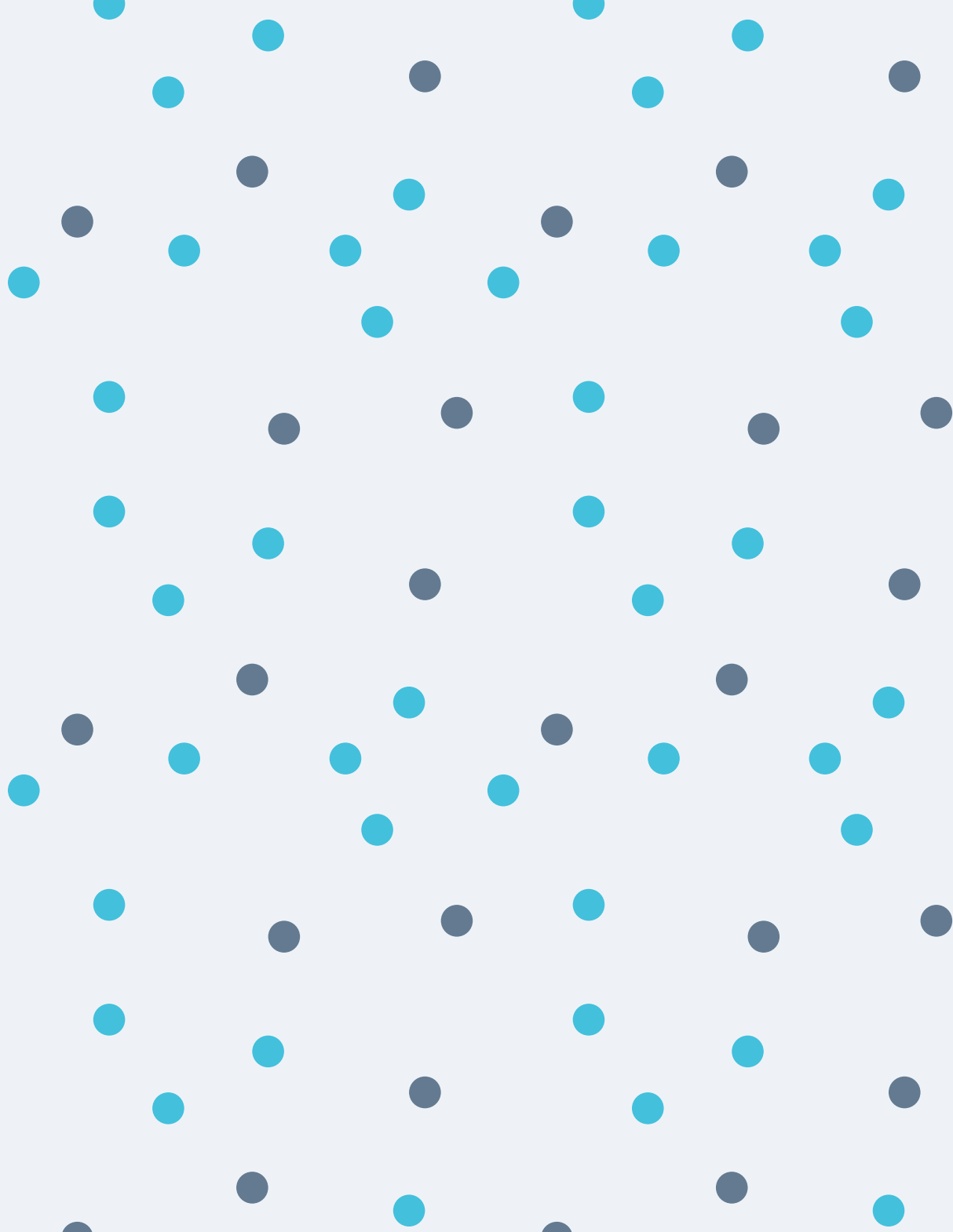
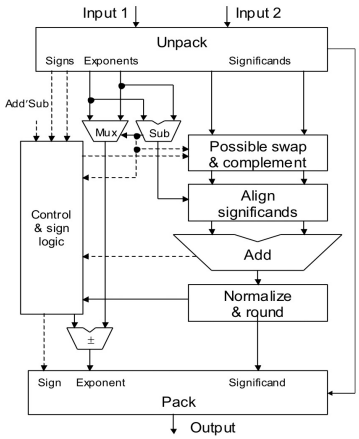


Arithmetic FLP



Addition

① Find FLP

a) Turn to bimal (if number is in decimal)

b) $1.\_\_\_\_ \times 2^{e}$ → Normalize

② Make exponents of both numbers the same

* Make smaller exponent equal the large exponent

③ Add Mantissas together

④ Normalize answer if needed

Example] 1) $N1=1.0011 \times 2^{23}$, $N_2=1.1111 \times 2^{27}$

① $N_1 =$ 0 | 1 0 0 1 0 1 1 0 | 0011 0000 0000000000000000
128 64 32 16 8 4 2 1
23 + 127 = 150

$N_2 =$ 0 | 1 0 0 1 1 0 1 0 | 1111 0000 0000000000000000
27 + 127 =

② $2^{23} \neq 2^{27}$ → Make them the same

$e^2 - e^1 = 27 - 23 = 4$ (Shift lower exponent by 4)

$N_1 = 0.00010011 \times 2^{27}$

$N_2 = 1.1111 \times 2^{27}$

③ $M = M_1 + M_2$

$$\begin{array}{r} {\scriptstyle 1\ 1\ 1\ 1} \\ 0.00010011 \\ +\ 1.11110000 \\ \hline 10.00000011 \end{array} \times 2^{27}$$

④ Normalize

$10.00000011 \times 2^{27}$

1.00000011×2^{28}

⑤ Turn to binary

0 | 1 0 0 1 1 0 1 1 | 0000000110000000...
28 + 127 = 155

⑥ Turn to decimal

1.00000011×2^{28}

256 128 64 32 16 8 4 2 1
$100000001.1 \times 2^{20}$
↓ 257 ↓ 0.5 ↳ $20 \times 0.3 = 6$

257.5×10^{6}

Multiplication

① Find FLP

a) Turn to bimal (if number is in decimal)

b) 1.____ x 2^e → Normalize

② Convert to binary

③ Find Sign , $S_3 = S_1 \oplus S_2$ (or gate)

OR	0	1
0	0	1
1	1	0

If one is negative, Final result is negative

④ Find exponent , $E_3 = E_1 + E_2 - bias$

⑤ Find mantissa , $M_3 = M_1 \times M_2$

⑥ Wite final result

Example] N_1=0.5, N_2= -0.4375

①

2x .5 → 0 (1)

2x 0.4375 → 2x 0.875 (0) → 2x 0.75 (1) → 2x 0.5 (1) → 0 (1)

$N_1 = 1 \times 2^{-1}$

$N_2 = -0.0111 = 1.11 \times 2^{-2}$

②

N_1 → 0 01111110 00000000 ....

-1 +127 = 126

N_2 → 1 01111101 10000000 ....

-2+127 = 125

③

$N_1 \rightarrow S_1 = 0$

$N_2 \rightarrow S_2 = 1$] or gate

$N_3 \rightarrow S_3 = 1$

④ $E_3 = E_1 + E_2 - bias$

$E_3 = 126 + 125 - 127 = 124$

$e_3 = 124 - 127 = -3$

⑤ $M_3 = M_1 \times M_2$

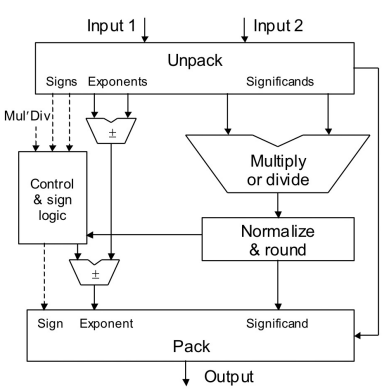
1.0 x 1.11

```
  1.0
x 1.11
------
  .010
 .10
1.0
------
1.110
```

$M_3 = 1.110$

⑥ 1 01111100 11000000000...

-1.11×2^{-3}



Q.6. Show the **multiplication** of the following numbers in the IEEE 754 standard:

a) $N_1=0.5$, $N_2= -0.4375$ and b) $N_1=1.11 \times 2^{20}$, $N_2=1.0011 \times 2^{-10}$

a) Already solved Page-3

b) ① Already given

$N_1 = 1.11 \times 2^{20}$

$N_2 = 1.0011 \times 2^{-10}$

② $N_1 \Rightarrow$ 0 | 1 0 0 1 0 0 1 1 | 1 1 0 0 0 0 0 0 0 0 0 0 ...

(128 64 32 16 8 4 2 1)

$E_1 = 20 + 127 = 147$

$N_2 \Rightarrow$ 0 | 0 1 1 1 0 1 0 1 | 0 0 1 1 0 0 0 0 0 0 0 0 0 ...

$E_2 = -10 + 127 = 117$

③ $S_3 = S_1 + S_2$

$S_1 \to 0$, $S_2 \to 0$

$S_3 = 0$

④ $E_3 = E_1 + E_2 - \text{bias}$

$E_3 = 147 + 117 - 127 = 137$

⑤ $M_3 = M_1 \times M_2$

```
        1.11        2
   ×  1.0011    + 4 = 6
   ------------
         111
        111
       000
      000
     111
   ------------
   10.000101
```

(6)

$\Rightarrow 10.000101 \times 2^{20-10} = 10.000101 \times 2^{10}$

⑥ 1.000101×2^{11}

Q.7. Show the addition of the following numbers in IEEE 754 standard:

1) $N1=1.0011 \times 2^{23}$, $N_2=1.1111 \times 2^{27}$ and

2) $N_1=1.001 \times 2^{-1}$, $N_2=1.11 \times 2^{-4}$.

a) ① Already in FLP

$N_1 = 1.0011 \times 2^{23}$

$N_2 = 1.1111 \times 2^{27}$

② $27 - 23 = 4$ (shift small exponent by 4)

0001.0011×2^{23}

0.00010011×2^{27}

③ $M = M_1 + M_2$

$$\begin{array}{r} {}^{1}0.0^{1}0^{1}10011 \\ +\ 1.1111\,0000 \\ \hline 10.0000\,0011 \end{array} \quad \Rightarrow \quad 10.0000\,0011 \times 2^{27}$$

④ Normalize

$1.0000\,0001\,1 \times 2^{28}$

b) ① Already in FLP

$N_1 = 1.001 \times 2^{-1}$

$N_2 = 1.11 \times 2^{-4}$

② $2^{-1} \neq 2^{-4}$

$-1 + 4 = 3$

$N_2 = 0001.11 \times 2^{-4} = 0.0111 \times 2^{-1}$

③ $M = M_1 + M_2$

$$\begin{array}{r} 1.0^{1}0^{1}1 \\ +\ 0.0011\,1 \\ \hline 1.0101\,1 \end{array} \quad \Rightarrow \quad 1.0101\,1 \times 2^{-1}$$

④ Already Normalized

$1.0101\,1 \times 2^{-1}$

Conversions

BCD

Decimal digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

$(7093)_{10}$ (8 4 2 1) → 0011, 1001, 0000, 0111 = 7 0 9 3 = 0111 0000 1001 0011

$(1000\ 0110)_{BCD}$ = $(86)_{10}$ → 6, 8

Convert BCD ↔ decimal ; digit by digit

BCD (signs)

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Zones	
1111	Unsigned
1100	Positive
1101	Negative

Convert BCD ↔ decimal

−1265 :-

a) Packed BCD (2 digit on a byte + sign at the end)

-1265 (8 4 2 1) → 0001, 0010, 0110, 0101 ; sign 1101

Should be 2 digits

b) EBDIC zoned-decimal (4 first bits of a byte [depends on zone], sign before end)

-1265 (8 4 2 1) → 0001, 0010, 0110, 0101 ; sign 1101

zone 1 | zone 2 | zone 6 | sign 5

zone = 1111

c) ASCII zoned-decimal (4 first bits of a byte [depends on zone], sign before end)

-1265 (8 4 2 1) → 0001, 0010, 0110, 0101 ; sign 1101

zone 1 | zone 2 | zone 6 | sign 5

zone = 0011

8-bit EBCDIC

LSB

Zone	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NUL	SOH	STX	ETX	PF	HT	LC	DEL		RLF	SMM	VT	FF	CR	SO	SI
0001	DLE	DC1	DC2	TM	RES	NL	BS	IL	CAN	EM	CC	CU1	IFS	IGS	IRS	IUS
0010	DS	SOS	FS		BYP	LF	ETB	ESC			SM	CU2		ENQ	ACK	BEL
0011			SYN		PN	RS	UC	EOT				CU3	DC4	NAK		SUB
0100	SP										[	.	<	(	+	!
0101	&										]	$	*	)	;	^
0110	–	/								'	:	#	@	'	=	"
0111																
1000		a	b	c	d	e	f	g	h	i						
1001		j	k	l	m	n	o	p	q	r						
1010		~	s	t	u	v	w	x	y	z						
1011																
1100	{	A	B	C	D	E	F	G	H	I						
1101	}	J	K	L	M	N	O	P	Q	R						
1110	/		S	T	U	V	W	X	Y	Z						
1111	0	1	2	3	4	5	6	7	8	9						

MSB

ASCII

Rightmost Four Bits	Leftmost Three Bits 000	001	010	011	100	101	110	111
0000	NUL	DLE	Space	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

LSB

MSB

Even parity (OR logic

00 → 0	000
01 → 1	011
10 → 1	101
11 → 0	110

Odd parity (~OR logic [opposite of OR logic])

00 → 1	001
01 → 0	010
10 → 0	100
11 → 1	111

Note:- code is stored in one byte

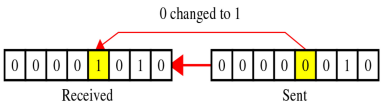
- Add a leading 0
- 8th bit is used for parity

Unicode

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

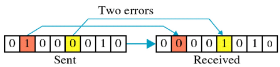
Types of bit errors

Single - bit error



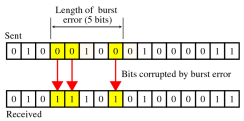
Only one bit is changed

Multi - bit errors



Two or more bits are changed (Not consecutive)

Burst errors



Two or more consecutive bits are changed

Techniques for error detection / correction

- Parity check
- Block parity check
- Systematic error detection
  - ↳ Cyclic Redundancy check
  - ↳ Checksum
- Hamming codes

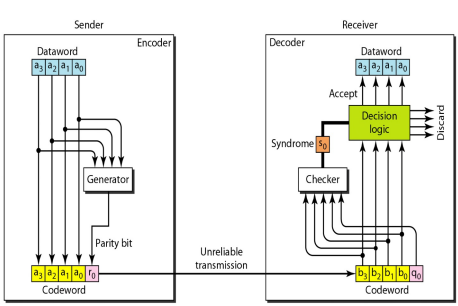
Parity check

Even parity (OR logic

00 → 0	000
01 → 1	011
10 → 1	101
11 → 0	110

Odd parity (~OR logic [opposite of OR logic])

00 → 1	001
01 → 0	010
10 → 0	100
11 → 1	111



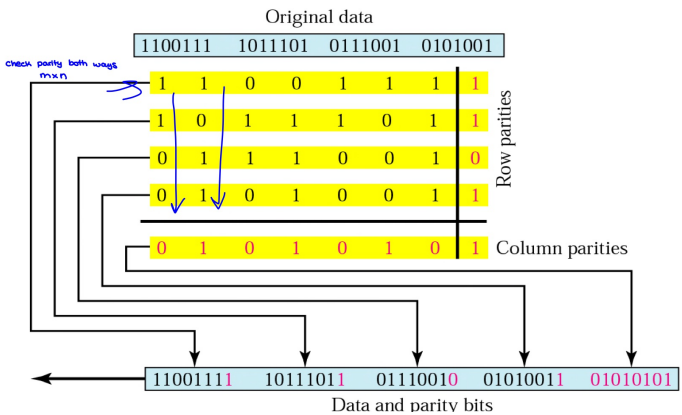
Suppose the sender wants to send the word ***world***. (even parity)

- Even numbers of 1's, add 0
- odd numbers of 1's, add 1

w	o	r	l	d
01110111	01101111	01110100	01101100	01100100
↑ 6	↑ 6	↑ 4	↑ 4	↑ 3

Block parity Check

checking parity in 2 directions



m + n + 1 parity bits

Cyclic Redundancy Check

Modulo 2 arithmetic

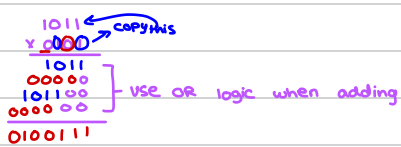
- Modulo 2 Addition/Subtraction

OR logic

	0	1
0	0	1
1	1	0

- Modulo 2 Multiplication

1) if Q is 1, copy
2) if Q is 0, all 0
3) Shift



- Modulo 2 division (explaination on next page)

- Subtract the denominator (the bottom number) from the leading parts of the numerator (the top number).
- Proceed along the numerator until its end is reached.
- Remember that we are using modulo 2 subtraction, which is equivalent to XOR.
  This has the effect that X/Y = Y/X.

Modulo 2 division

polynomial generator $x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$

$1\ 0\ 1\ 0\ 0\ 1\ 1\ 1$

Divisor = 10100111

① Find the divisor by polynomial generator

② Generate an augmented data-word from highest polynomial of 0s

③ divisor | binary aug data-word

④ a] If MSB is 1, copy & shift

b] if MSB is 0, all 0s & shift

⑤ Subtract till you get a remainder using OR logic $00 \to 0$, $01 \to 1$, $10 \to 1$, $11 \to 0$

⑥ Change augmented data-word to what receiver gets

⑦ Do the same for receiver if remainder is 0, there is no error

Example 1] Data word to be sent - 100100
generator polynomial x3 + x2 + 1]

① $x^3 + x^2 + 1$

$1\ 1\ 0\ 1$ => divisor

② x^3

000

③

```
1101 | 100100000
     - 1101
       01000
     -  1101
       01010
     -   1101
       01110
     -    1101
       00110
     -     0000
       01100
     -    1101
       00001
```

Sender will send :-

100100001

```
1101 | 100100001
     - 1101
       01000
     -  1101
       01010
     -   1101
       01110
     -    1101
       00110
     -     0000
       01101
     -    1101
       00000
```

↳ All 0s, no error is detected

Example 2] A computer program would like to send the binary message **110101100** to another computer using the **Cyclic Redundancy Check** method, with a **polynomial generator** of x^4+x^2+x+1.

a) **Calculate the sent message** showing all details.

b) What would the **response of the receiver** be, if the received message is **1101011000101**?

a) ① $x^4 + x^2 + x + 1$

1 0 1 1 1 => divisor

② x^4

0000

③ 10111 | 110101100 0000

- 10111
- 01101 1
- 10111
- 01100 1
- 10111
- 01110 0
- 10111
- 00101 0
- 00000
- 01010 0
- 00000
- 10100 0
- 10111
- 00011 1 => Remainder

Sent message :-

110101100011 1

b) 10111 | 110101100 101

- 10111
- 01101 1
- 10111
- 01100 1
- 10111
- 01110 0
- 10111
- 00010 1
- 00000
- 00101 0
- 00000
- 01010 0
- 00000
- 10101 1
- 10111
- 00010

We didn't get 0000, there is an error detected

- receiver will send 0101 to the sender requesting for a re-transmission of the message

Checksum

1. Split codeword into 8-bits
2. Add all bytes
3. Flip result to find 1's complement
4. Check if one's complement of the result is 0
   a) If 0, message is correct
   b) else, message is wrong

Example 1] 10101001 00111001

1. Already split
2.
```
  1 1 1   1
  10101001
+ 00111001
  11100010
```
3. 00011101

Sent pattern :- 10101001 00111001 00011101

4. 00111001 ≠ 00000000

   There is an error, message is wrong