(r-1)'s complement addition (B)

- Add corresponding digits, including signs
- If there's a carry out of the leftmost digit, add 1 to the result. This is different than r's complement, where end carry was ignored. This is called end-around carry

Example:

$$\begin{array}{r} +14 \\ -12 \\ \hline 2 \end{array} \rightarrow (001100) \qquad \begin{array}{r} 001110 \\ +\ 110011 \\ \hline 1000001 \\ +\ \quad 1 \\ \hline 000010 \end{array}$$

end-around carry

$$
\begin{array}{rl}
+ & +12 \\
 & -14 \rightarrow (\overline{001110}) \\
\hline
 & -2
\end{array}
\qquad
\begin{array}{rl}
+ & 001100 \\
 & \overline{110001} \\
\hline
 & 111101 \\
 & \uparrow \text{ no } c \\
 & (\overline{111101}) \rightarrow 000010
\end{array}
$$

(r-1)'s complement subtraction

Add the (r-1)'s complement of the subtrahend to the minuend.

Example:

$$
\begin{array}{rl}
- & +14 \\
 & +12 \rightarrow (\overline{001100}) \\
\hline
 & 2
\end{array}
\qquad
\begin{array}{rl}
+ & 001110 \\
 & \overline{110011} \\
\hline
 & 1000001 \\
+ & \quad\quad 1 \leftarrow \text{end-around carry} \\
\hline
 & 000010
\end{array}
$$

$$
\begin{array}{rl}
- & -14 \\
 & -12 \rightarrow (\overline{110011}) \\
\hline
 & -2 \quad 000010
\end{array}
\qquad
\begin{array}{rl}
+ & 110001 \\
 & 001100 \\
\hline
 & 111101 \\
 & \uparrow \text{ no } c \\
 & \leftarrow
\end{array}
$$

Overflow may occur as in 2's complement. In addition of same signs, the sign of result is known upfront. Hence, checking the sign of the result indicates whether overflow occurred. In subtraction of opposite signs, the result will have the sign of minuend, hence overflow can also be detected.

Multiplication of signed numbers

- The sign of result is determined by the signs of the multiplicand and multiplier
- Then unsigned multiplication takes place w/o sign bits by a repeated left-shift and add operations. Negative is first complemented to positive

Example:

```
          × 1 1 0 0 1
            0 1 1 0 1
          -------------
            1 1 0 0 1
          0 0 0 0 0
        1 1 0 0 1
      1 1 0 0 1
    0 0 0 0 0
  -------------------
  1 0 1 0 0 0 1 0 1
```

Division of signed numbers

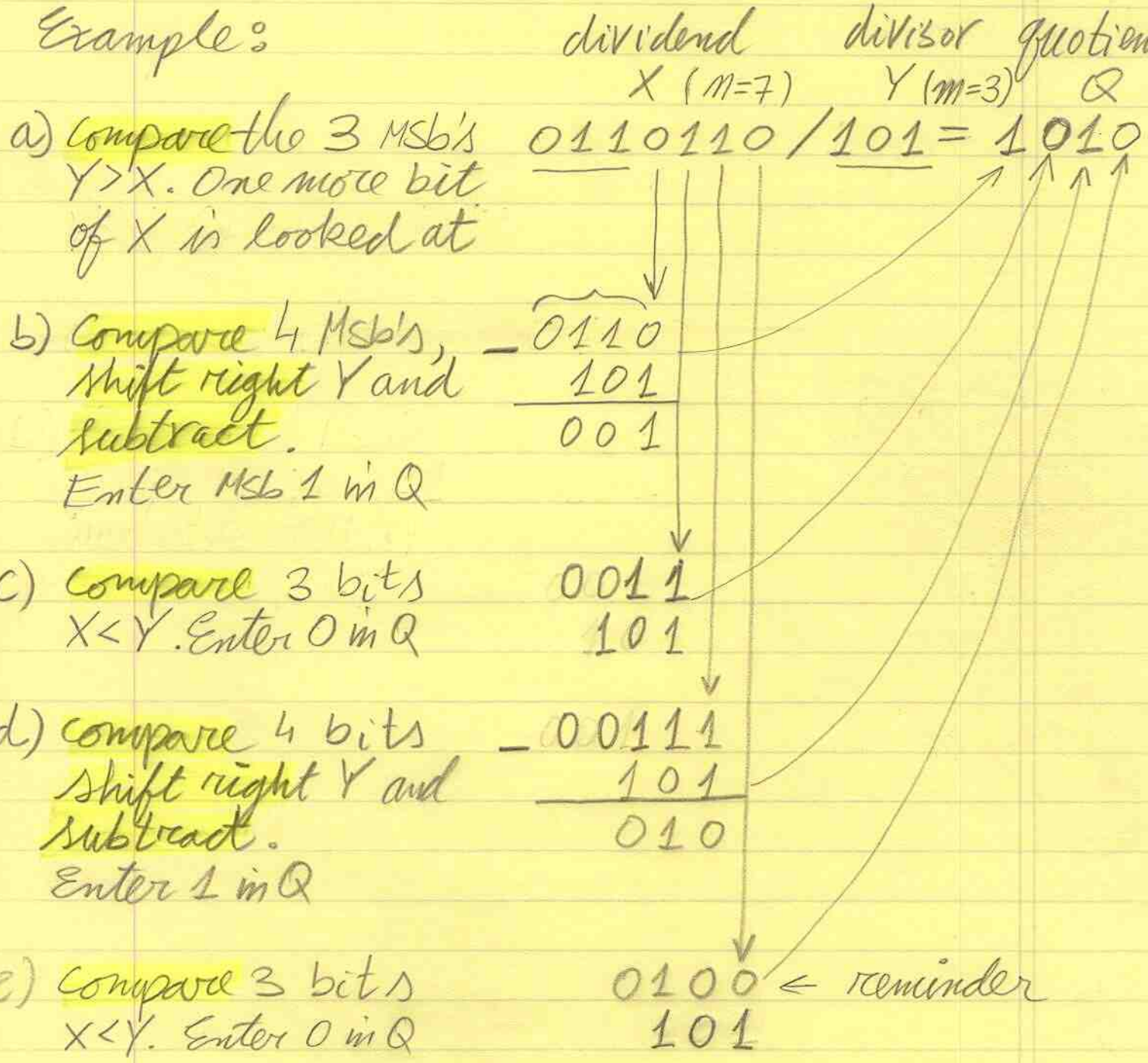
- sign of result is handled similar to multiplication
- Repeated process of compare, right-shift and subtract. (Subtraction will take place in hardware by 2's complement.)

X- n-bit dividend

Y- n-bit divisor

We can always assume $n > m$ by adding 0's (right-shift) and then left-shift the quotient

Example:



f) Reminder is 100

checking

$$
\begin{array}{r}
1010 \\
\times\ 101 \\
\hline
1010 \\
+\ 0000 \\
1010 \\
\hline
+\ 110010 \\
100 \\
\hline
110110
\end{array}
$$

Binary Codes

We'll consider more numerical codes, non-numerical and error-correcting codes.

For 2^m distinct elements we need at least m bits, but more can be used.

For instance, representing k elements we may use k bits: $0\cdots01$, $0\cdots10$, ..., $10\cdots0$.

Decimal Codes - representing decimal digits with bits

Decimal Digit	BCD 8421	2421	84(-2)(-1)	Excess 3
0	0000	0000	0000	0011
1	0001	0001	0111	0100
2	0010	0010	0110	0101
⋮	⋮	⋮	⋮	⋮
8	1000	1110	1000	1011
9	1001	1111	1111	1100
	weighted codes			unweighted code

BCD disadvantage is the self-complementing calculation

2421, 84(-2)(-1) and Excess 3 are self complementing, which means that the 9's complement of any digit is obtained by complementing the bits.

Example:

Obtain 9's complement of $(91)_{10}$

BCD: $(91)_{10} = (1001\ 0001)$
$(08)_{10} = (0000\ 1000)$

2421: $(91)_{10} = (1111\ 0001)$
$(08)_{10} = (0000\ 1110)$

Excess-3: $(91)_{10} = (1100\ 0100)$
$(08)_{10} = (0011\ 1011)$

} self-complement

Decimal codes with more than 4 bits + Error detection codes

Decimal	Biquinary 5 0 4 3 2 1 0	2-out-of-5
0	0 1 0 0 0 0 1	0 0 0 1 1
1	0 1 0 0 0 1 0	0 0 1 0 1
2	0 1 0 0 1 0 0	0 0 1 1 0
3	0 1 0 1 0 0 0	0 1 0 0 1
4	0 1 1 0 0 0 0	0 1 0 1 0
5	1 0 0 0 0 0 1	0 1 1 0 0
6	1 0 0 0 0 1 0	1 0 0 0 1
7	1 0 0 0 1 0 0	1 0 0 1 0
8	1 0 0 1 0 0 0	1 0 1 0 0
9	1 0 1 0 0 0 0	1 1 0 0 0

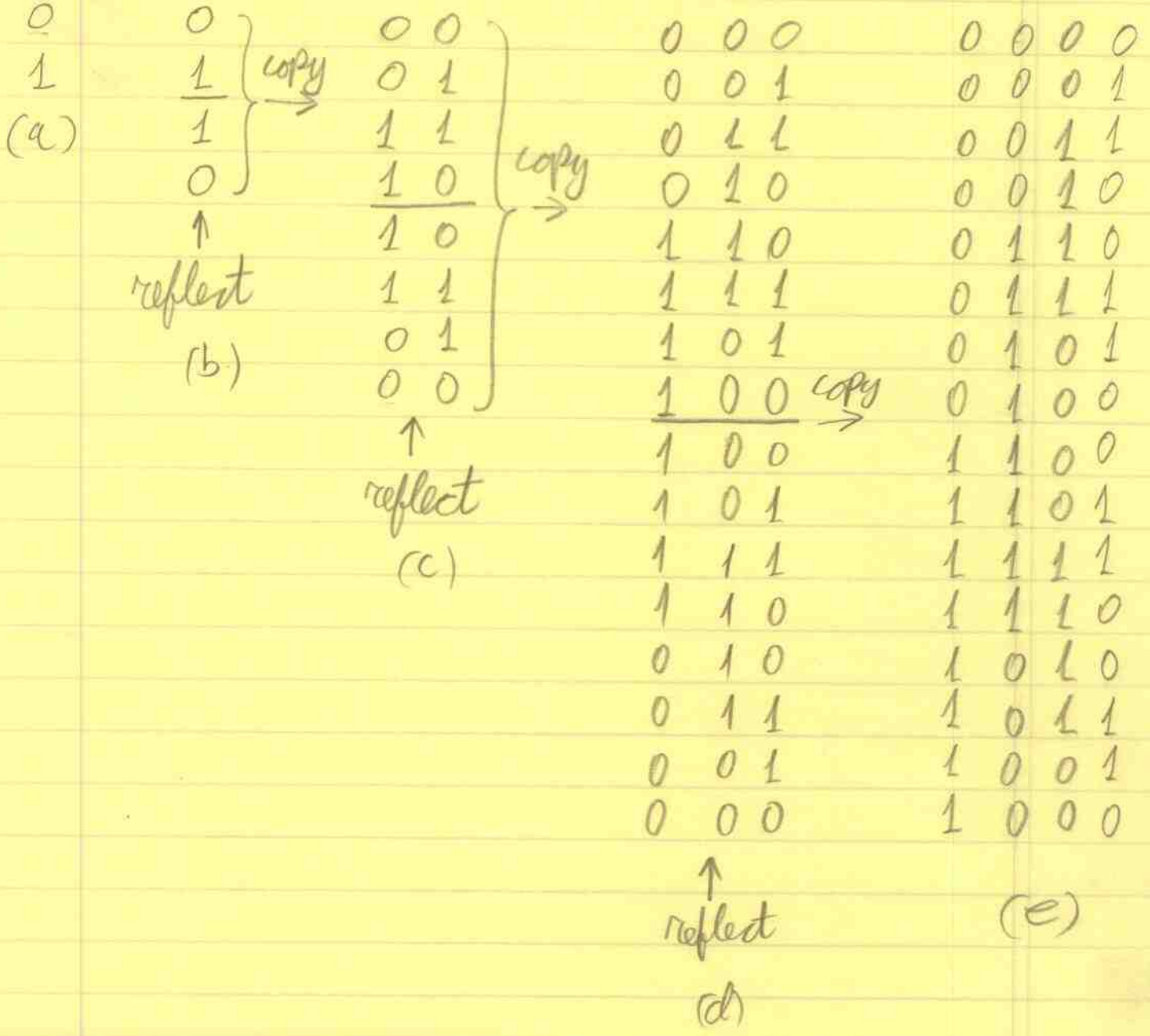
Gray code

Successive digits differ in one-bit only. This is very useful for data transmission changing smoothly (analog, continious).

Assume that we'd transmit digit 7 and then 8. In BCD 0111 will change to 1000 and a race may occur, resulting in erroneous code.

Decimal	Gray	
0	0000	Switching Power
1	0001	
2	0011	
3	0010	
.	.	
.	.	
.	.	
13	1011	
14	1001	
15	1000	

Gray code is a type of reflected code obtained as follows

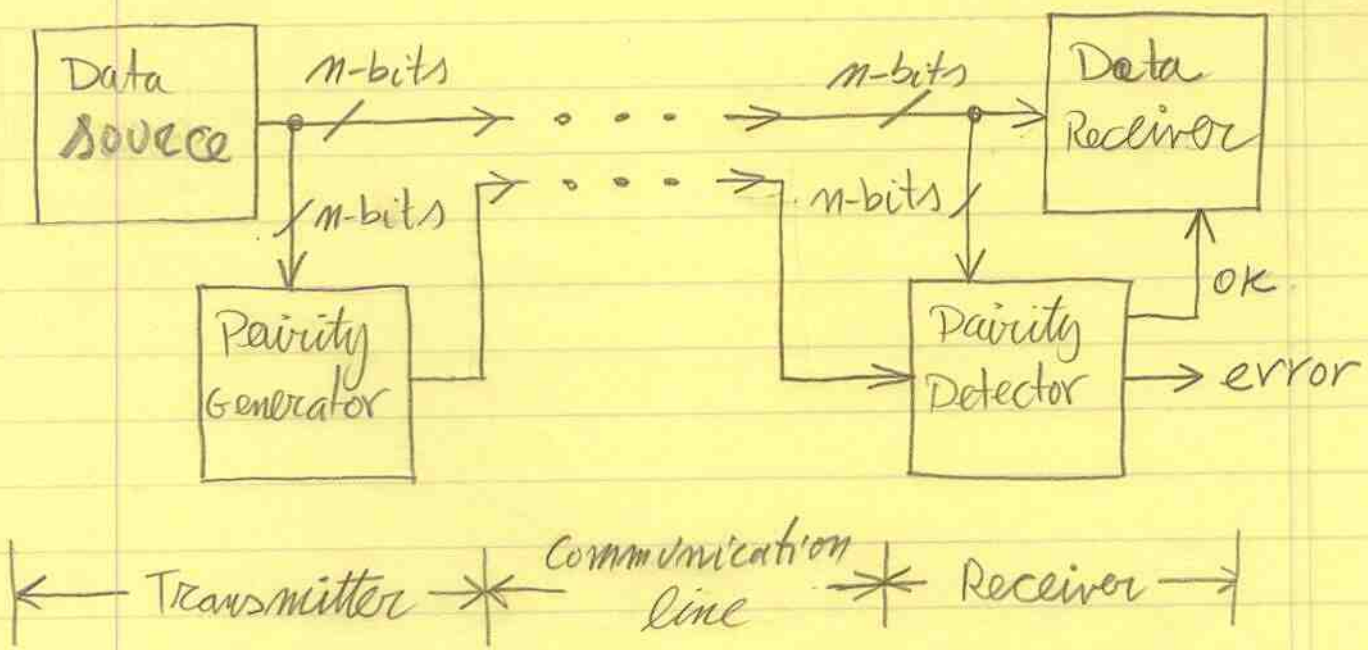


Error-Detection codes

Binary data is suspect to errors occurring at its transmission 1→0 or 0→1 unintendently. 2-out-of-5 and Biquinary codes allow detection of

one error (or any odd) since pairity of 1's change.

There's a method applicable to any binary code by adding a pairity bit, making each word to be even number (odd number) of 1's. Pairity bit is added as the rightmost bit. The word 01011011 is sent as 010110 111, while the word 11000011 is sent as 110000 110



Floating Point Numbers

So far we dealt with fixed point numbers.
If a number is represented by 32-bit word then the range of values is limited to $[(2^{31}-1), (-2^{31})]$ in 2's complement. This is not enough for scientific calculations involving real numbers

$(3.14159265\ldots)_{10}$ - π

$(2.71828\ldots)_{10}$ - e

$(1.0 \times 10^{-9})_{10}$ Sec - nano second

$(3,155,760,000)_{10}$ Sec - # Sec. in a century)

Decimal numbers in scientific notation are represented by a single digit left to the decimal point, multiplied by a power of 10

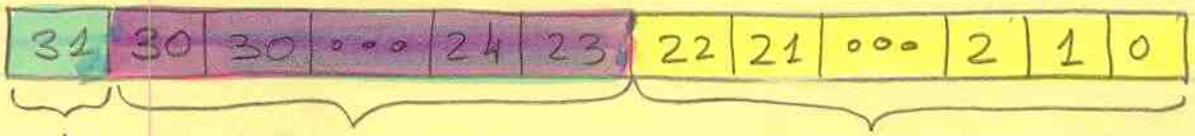
$$\pm(\underset{\substack{\uparrow \\ 1-9}}{d}.\underbrace{xx\cdots x}_{0-9})_{10} \times 10^{yy\ldots y}$$

This representation is called floating point

Floating point in binary look as

$$\pm(1.xx\cdots x)_2 \times 2^{yy\cdots y}$$

Consider a computer whose word is 32-bit



A number is then represented by

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{\text{Exponent}}$$

What is the range of numbers that can be represented?

1 + Fraction ranges from 1 to $(2 - 2^{-23})$

Exponent has 8 bits where bit #30 is its sign, hence exponent range $[2^{2^7-1}, 2^{-2^7}] =$

$= [2^{127}, 2^{-128}] \approx [10^{38}, 10^{-38}]$

In summary, 32-bit floating point represents numbers in range $[+2\times10^{38}, -2\times10^{38}]$ with very high precision of 10^{-38}, compared to $[2^{31}-1, -2^{31}] \approx [10^{10}, -10^{10}]$ in fixed point

The above representation is called **IEEE 754 standard**

Example: $(-0.375)_{10} = (?)_{FP(IEEE\ 754)}$

$(-0.375)_{10} = -(\frac{1}{2^2} + \frac{1}{2^3})_{10} = (-0.011)_2 \times 2^0 =$

$= (-1.1)_2 \times 2^{-2} = (-1)^1 \times (1+0.1)_2 \times 2^{-2}$

sign — fraction — Exponent

31	30	29	28	27	26	25	24	23	22	21	20	...	2	1	0
1	1	1	1	1	1	1	1	0	1	0	0	...	0	0	0

negative — -2 in 2's comp. — $(0.1)_2$

The fraction part is called also **significands, mantissa**

Example:

What decimal number is represented?

31	30	29	28	27	26	25	24	23	22	21	20	19	...	0
1	1	1	1	1	1	1	0	0	0	0	1	0		0

↑ 1's c -4 in 2's complement 2^{-3}

$= (-1)^1 \times (0.0625)_{10} \times (1 + 0.125)_{10} = -0.0703125$

Floating Point addition - decimal

Assume we can store 4 digits of significand and 2 digits of exponent.

$$(9.999)_{10} \times 10^1 + (1.610)_{10} \times 10^{-1} = (?)_{FP}$$

① align the number with smaller exponent to the larger exponent (left shift of decimal point)

$(1.610)_{10} \times 10^{-1} = (0.01610)_{10} \times 10^1 = (0.016)_{10} \times 10^1$

↑ round to 4 digits

② addition of significands

$$\begin{array}{r} (9.999)_{10} \\ (0.016)_{10} \\ \hline (10.015)_{10} \end{array}$$

③ Normalize to scientific notation (by shift)

$$(10.015)_{10} \times 10^1 = (1.0015)_{10} \times 10^2$$

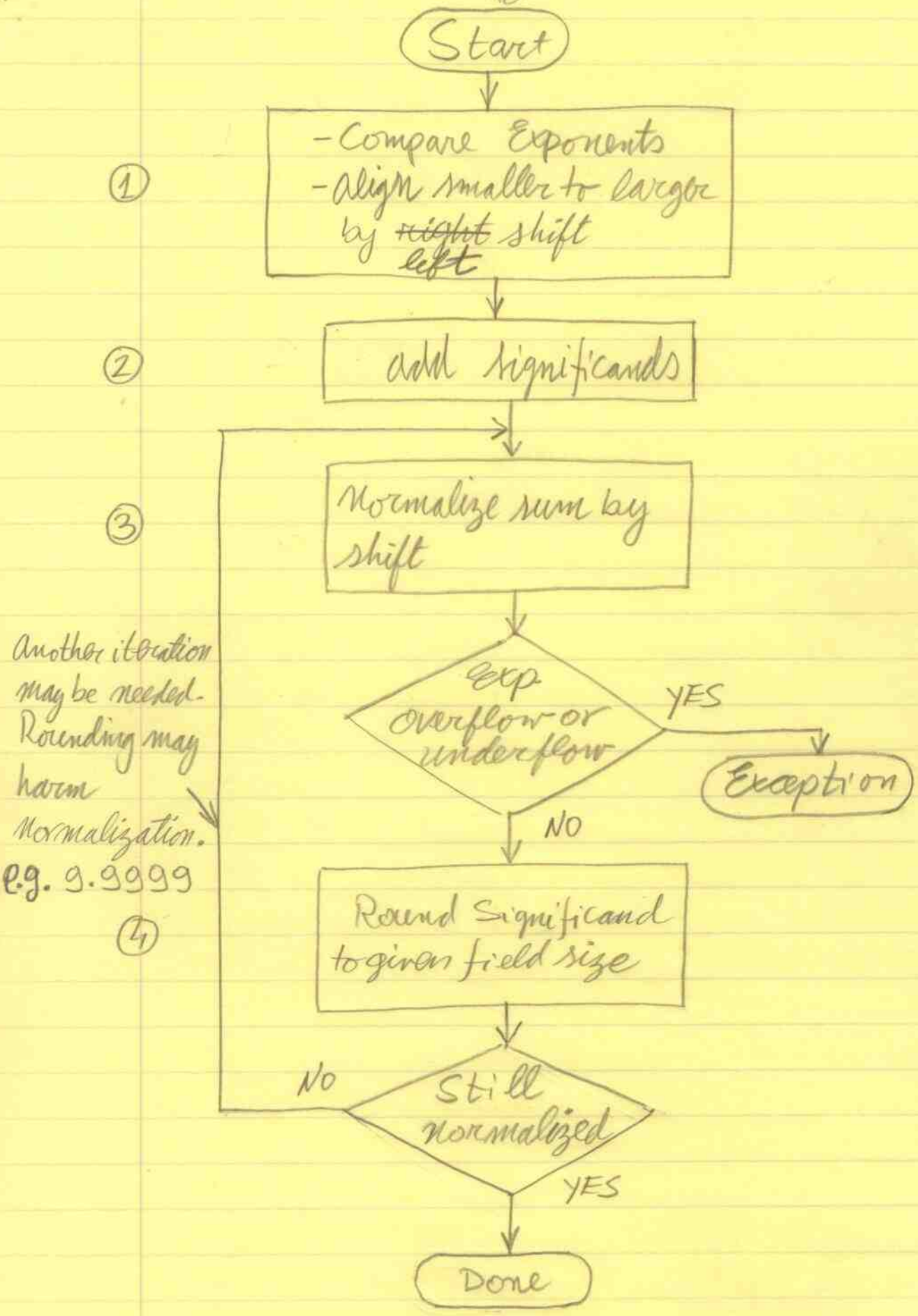
Normalization must check for overflow of the exponent, namely, it must fit its given field.

④ Fitting the significand to given field by rounding

$$(1.0015)_{10} \times 10^2 = (1.002)_{10} \times 10^2$$

↑ 4 digits

↑ 1 digit of exponent


Start
- Compare Exponents
- align smaller to larger by ~~right~~ shift left
1
add significands
2
Normalize sum by shift
3
Another iteration may be needed. Rounding may harm normalization.
e.g. 9.9999
4
Exp overflow or underflow
YES
Exception
NO
Round Significand to given field size
Still normalized
NO
YES
Done

addition - IEEE 754

$$(0.5)_{10} + (-0.4375)_{10} = (\cdot)_{FP(IEEE\ 754)}$$

$$(0.5)_{10} = \left(\frac{1}{2}\right)_{10} = (1 \times 2^{-1})_{10} = (0.1)_2 = (1.000)_2 \times 2^{-1}$$

$$(-0.4375)_{10} = -\left(\frac{1}{4} + \frac{1}{8} + \frac{1}{16}\right)_{10} = (-0.0111)_2 = (-1.110)_2 \times 2^{-2}$$

① Shift smaller exponent to align with larger

$$(-1.110)_2 \times 2^{-2} = (-0.111)_2 \times 2^{-1}$$

② add significands

$$(1.000)_2 \times 2^{-1} + (-0.111)_2 \times 2^{-1} = (0.001)_2 \times 2^{-1}$$

↑ use 2's complement

③ Normalize sum and check overflow or underflo

$$(0.001)_2 \times 2^{-1} = (1.000)_2 \times 2^{-4}$$

No exponent overflow or underflow

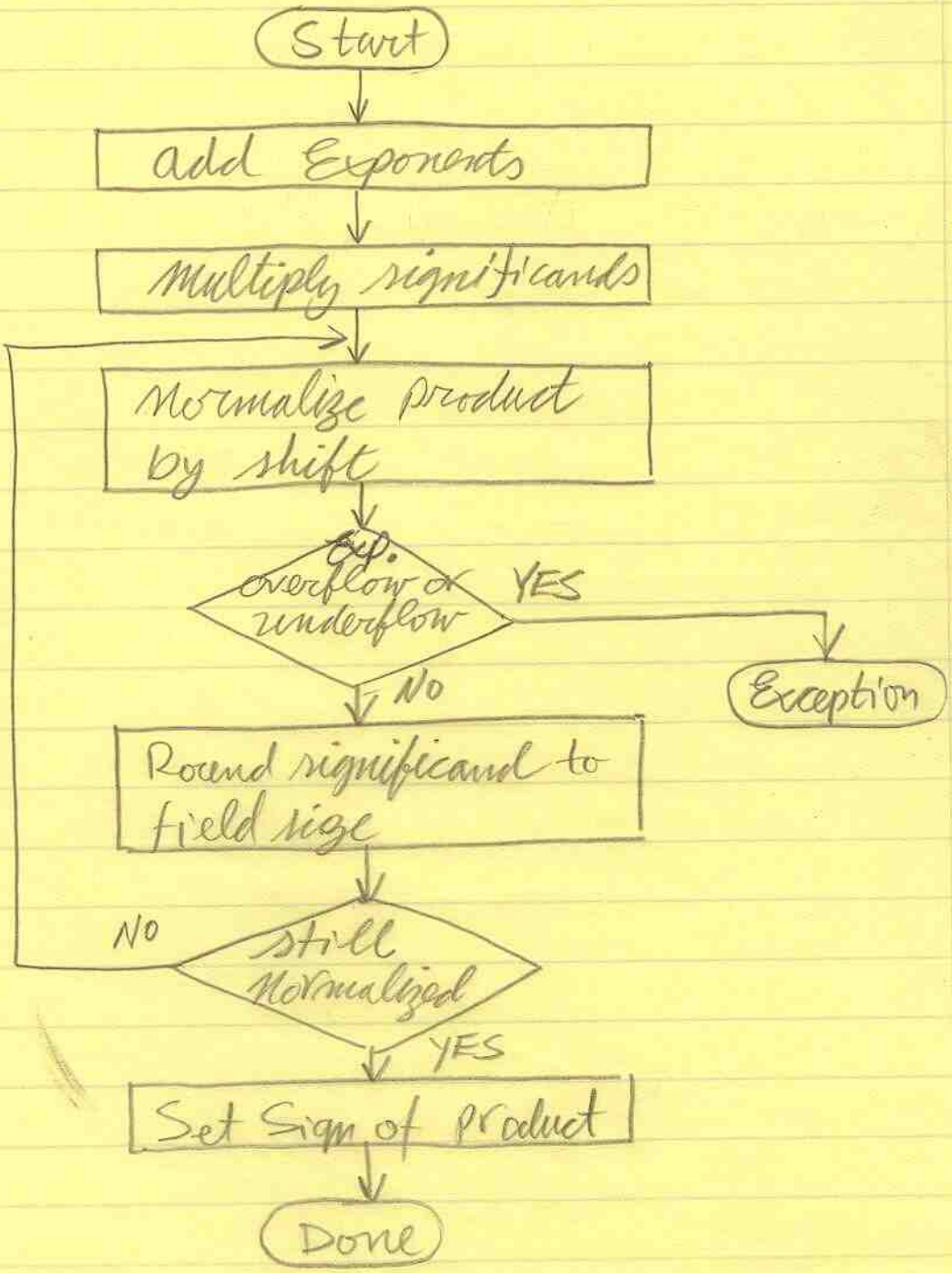
④ Round the sum -

sum is already rounded (less than 23 digits)

Indeed, $(1.000)_2 \times 2^{-4} = (0.0625)_{10} =$

$= (0.5)_{10} + (-0.4375)_{10}$ as expected

Floating-point multiplication



Example: (4 digit significand, 2 digit exponent)

$(0.5)_{10} \times (-0.4375)_{10} = (\quad)_{FP}$

Conversion to binary as done in addition example

obtains $[(1.000)_2 \times 2^{-1}] \times [(-1.110) \times 2^{-2}]$

① add exponents

$$-1+(-2) = -3$$

② multiply significands

$$\begin{array}{r} (1.000)_2 \\ \times\ (1.110)_2 \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline (1.110000)_2 = 1.110 \end{array}$$

↑ 4 digits significand

③ normalize significand

It is already normalized

no change of exponent

④ Round significand (4 digits)

already rounded 1.110×2^{-3}

⑤ set sign

since sign of operands differ, the result is negative

$$-1.110 \times 2^{-3}$$

Indeed, $-1.110 \times 2^{-3} \overset{\text{conversion}}{=} (-1.75 \times 0.125)_{10} =$

$$(-0.21875)_{10} = (0.5)_{10} \times (-0.4375)_{10}$$

operator priority

① Parenthesis ( )

② {'}

③ {·}

④ {+}