Signed Integer numbers

Fixed Length Numbers

show memory
• byte, consecutive bytes.

• Computer stores numbers in fixed-length memory locations.

Example : integer are usually stored in a memory location consisting of 4 consecutive bytes (total 32 bits).

• The amount of memory used to represent one type of data is fixed.

example :

type of data	amount of memory used
character	1 byte
integer	4 bytes.
float	4 bytes
double	8 bytes.
short	2 bytes.

• Unsigned number = sequence of bits of number viewed as corresponding to powers of 2.

eg $1010 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$

$= 8 + 2 = 10.$

Representing & Using signed numbers

- Methods used to represent signed numbers in computers:
  (fixed length !!!).

  (1) sign / magnitude representation.

  (2) two's complement

  (3) one's complement (obsolete).

  (4) excess 2^{n-1} (n = # bits used).

  skip

Sign/magnitude representation:

- people is accustumed to this representation,
  eg:

  +4 = pos. 4
  −4 = neg. 4.

  very clumpsy !!!

  The number is represented by a sign & its absolute value (magnitute).

- An analogic representation is available in binary.

  The sign (+/−) is represented by a bit 0/1

  0 = +
  1 = −

So the INTERPRETATION is:

$\boxed{A}\boxed{\qquad\qquad\qquad\qquad}$

Sign — magnitude. (value).

In general, the leftmost bit in a binary number is chosen as the sign bit.

eg. with 8 bits, we can interprete the numbers:

00001101 as: $\boxed{0}\underbrace{000\overset{3}{1}\overset{2}{1}\overset{1}{0}\overset{0}{1}}$

sign = +

magnitude = 8+4+1
= 13.

number = +13.

And:

10001101 as: $\boxed{1}000.\underbrace{1101}$

sign = −

magnitude = 13.

number = −13

- ~~How~~ What numbers can be represented with 8 bits?

$\boxed{0}\underbrace{\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}}$

Sign

magn. $\leq 2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$
$= 127.$

numbers represented $[-127, 127]$.

Note: what value a bitpattern represents depends on the context

eg: if we are given a bit pattern:

1 0 0 0 1 1 0 1

But not given the context under which the pattern is considered, we cannot know the value represented.

How do we specify the context: depends on the programming environment.

In C:

(1) unsigned char X ;

if X contains: $\boxed{1\ 0\ 0\ 0\ 1\ 1\ 0\ 1}$

the value represented (stored in) by X is

$$2^7 + 8 + 4 + 1 = 128 + 8 + 4 + 1$$
$$= 141$$

(2) char x

Now x is signed! different context!

Problems with sign/magnitude representation:

(1) 2 different patterns represent the value ∅

$+\emptyset$	0 | 000...0
$-\emptyset$	1 | 000...0

testing for ZERO is complicated ....

(2) Addition & subtraction are not uniform, but depends on values of operands.

eg:

$$\begin{array}{r} 5 \\ +\ 3 \\ \hline 8 \end{array} \quad \text{is an addition} \quad \begin{array}{r} 0000.0101 \\ 0000.0011 \ + \\ \hline 0000.1000 \end{array}$$

but:

$$\begin{array}{r} 5 \\ +\ -3 \\ \hline 2 \end{array} \quad \text{is actually a subtraction.}$$

$$\begin{array}{r} 0000.0101 \\ +\ 1000.0011 \\ \hline \end{array}$$

Convert

$$\begin{array}{r} 0000.0101 \\ -\ 0000.0011 \\ \hline 0000.0010 \end{array}$$

Stretch representation (of context)

Odometer numbers: intro to complementary arithmetic

- Five-digit mileage indicator (odometer) on exercise cycle.

$\boxed{0}\boxed{1}\boxed{2}\boxed{4}\boxed{9}$

- Initial odometer reading = 00000

- Pedal forward : +1
  Pedal backward : −1.

Odometer Reading	Distance Traveled
⋮	
00002	2 miles forward
00001	1 mile forward
00000	Initial reading
99999	1 mile backward
99998	2 miles backward
⋮	

- The concept of distance in one of 2 opposite directions is the basis of signed numbers in arithmetic.

- Representation of signed numbers:

signed number:	-50000	.....	-2	-1	0	1	2	49999
odometer number	50000		99998	99999	00000	00001	00002	49999

- Problem:

An odometer number 60000 could be a result of:

(1) 60000 miles forwards. → +60000

(2) 40000 miles backwards... → -40000

- Some convention is needed to give each odo number a unique value:

We split the range of possible numbers

00000 - 99999

into 2 "equal" parts:

0-4 first digit → 00000 - 49999 represent "positive" numbers — forward

5-9 first digit → 50000 - 99999 represent "negative" numbers — backwards.

Adding odometer numbers:

$$\begin{array}{r} 99998 \\ +\ 00003 \\ \hline 1\,|\,00001 \end{array} \qquad \begin{array}{r} = \text{"}-2\text{"} \\ = +\ \text{"}+3\text{"} \\ \hline +1 \end{array}$$

discarded.

Subtraction:

$$\begin{array}{r} 99995 \\ -\ 00003 \\ \hline 99992 \end{array} \qquad \begin{array}{r} = \text{"}-5\text{"} \\ -\ \text{"}+3\text{"} \\ \hline = \text{"}-8\text{"}. \end{array}$$

Inverting a number:

00003 = +3.

99997 = −3.

To invert 00003 : subtract it from 100000.

To invert 99997 : also subtract it from 100000.

Usage: What value is associated with odometer number 04500 ?

04500 is a positive odometer number
read it as the value itself.

What value is associated with odometer number 55000 ?

55000 is a negative odometer number.
Invert:

$$\begin{array}{r} 100000 \\ -\ 55000 \\ \hline 45000. \end{array}$$

negative value is equal to 45000.

The value is then equal to -45000.

Converting:

	odometer
+ value →	decimal
− value →	100000 − decimal
actual value ←	[0..4] repr.
−(100000 − repr) ←	[5..9] repr.

- Odometer numbers are in fact called:

10's complement numbers.

- Overflow:

Due to the fact that ~~the~~ number of possible values represented is limited, overflow will / can occur.

eg:

$$\begin{array}{rrcr} & 04992 & = & 4992. \\ + & 00009 & = & 9. \\ \hline & 50001 & & \end{array}$$

it represents a negative value!

$$\begin{array}{r} 100000 \\ 50001 \\ \hline 49999 \end{array}$$

demo/int.c

Value represented = -49999.

The addition took 2 positive values & produced a negative value.

We said there was an overflow (more than the representation can handle → it can handle positive numbers upto 49999, result is 50001, more than the representation can handle).

Same effect as: old car: 99999
drive one mile → new car! 00000 !!!

10's complement encoding of signed DECIMAL numbers:

- Complement of digit d = $9 - d$.

~~eg: 2232 digits~~. $9 \leftrightarrow 0$ ... $5 \leftrightarrow 4$.
$8 \leftrightarrow 1$ ...

- To negate a number, take its digit wise complement and add 1.

- eg: 3 digits

$+123 \quad = \quad 123.$

$-123 \qquad +123$

$\downarrow$ complement.

$$\begin{array}{r} 876 \\ +1 \\ \hline 877 \end{array}$$

same as $\begin{array}{r} 1000 \\ -123 \\ \hline 877 \end{array}$

So "877" represents '-123 !!!

subtract from 1000

- The left most digit is a sign digit:

  digit ≥ 5 $\rightarrow$ number is negative.

  digit < 5 $\rightarrow$ number is positive.

Only one representation for $\emptyset$: 000.

$+0 = 000$

$-0 = 000$

$$\begin{array}{r} 000 \\ \downarrow \text{ complement} \\ 999 \\ +1 \\ \hline 1\,000 \end{array}$$

discard.

- Eg. What does 821 represent?

  (1) find sign: neg.

  (2) find value: 821 $\downarrow$ compl.

$$\begin{array}{r} 178 \\ +1 \\ \hline 179. \end{array} \Rightarrow \underline{\underline{-179}}$$

Now: Humans Encoding.

$$\begin{array}{r} 321 \\ + \; -123 \\ \hline 198 \end{array} \;\rightarrow\; \begin{array}{r} 321 \\ + \; 877 \\ \hline 1198 \end{array} = 198.$$

discard

The magic works through ENCODING (modulo - calculus).

modulo calculus.

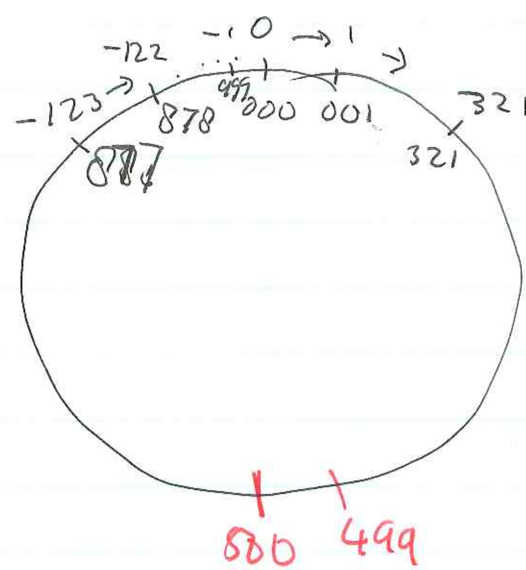
$$\begin{array}{r} 321 \\ + \; -123 \\ \hline \end{array} \quad \begin{array}{r} 321 \\ 877 \\ \hline \end{array}$$

actually:

$$\Leftrightarrow \begin{array}{r} 321 \\ + \;(1000-123) \\ \hline \end{array}$$

$$\begin{array}{r} 321 \\ -123 \\ \hline 198. \end{array} \Leftrightarrow \begin{array}{r} 321 \\ + \; -123 \\ \hline \end{array}$$

Explanation:



overflow can happen here also.

Example:

$$\begin{array}{r} 123 \\ -\ 321 \\ \hline 802 \end{array}$$

$\downarrow$ "802" represents -198.

$8 \geq 5 \rightarrow$ number neg.

Invert:

$$802 \xrightarrow{\text{complement}} 197$$

$$\begin{array}{r} 197 \\ +1 \\ \hline 198. \end{array}$$

Thus: "802" represents $\underline{-198}$.

How it is computed: by humans.

$$\begin{array}{r} 321 \\ -\ 123 \\ \hline 198 \end{array}$$

The result is negative

Thus: -198.

Two's complement Binary numbers.

- Concepts for 10's complementary numbers (odometer numbers) can be extended to fixed-length binary numbers.

  The resulting representation is called "Two's complementary representation")

- Assume we have 8 bits fixed length binary numbers.

Two's complementary number	Value represented by the number
⋮	
0 0 0 0 0 0 1 1	3_{10}
0 0 0 0 0 0 1 0	2_{10}
0 0 0 0 0 0 0 1	1_{10}
0 0 0 0.0 0 0 0	0_{10}
1 1 1 1 1 1 1 1	-1_{10}
1 1 1 1 1 1 1 0	-2_{10}
1 1 1 1 1 1 0 1	-3_{10}
1 1 1 1 1 1 0 0	-4_{10}.
⋮	

- The range of possible values:

0000.0000 - 1111.1111

is divided into 2 "equal" pieces:

all numbers starts with 0 0000.0000 - 0111.1111 represents the values from 0 - 127.

all starts with 1 → 1000.0000 - 1111.1111 represents negative values (-1 to -128).

1111.1111 represents -1_{10}.
1000.0000 represents -128_{10}.

- How to negate a 2's complement number?

eg: 0000.0001 represents +1.

What is the 2's complement repr. for the number -1?

Ov: 10000.0000
− 0000.0001
1111.1111 → represents -1.

- How to obtain the value represented by a 2's complement number?

(1) if number is positive (first digit = 0), use the formula :

$$\sum_{i=0}^{N} a_i . 2^i$$

(2) if number is negative (first digit = 1), do:

(2a) ~~find the~~ negate the number. — ~~value~~ result is positive

(2b) apply (1). above.

Example: 0001.1011_2 = ?

Answer $= 2^4 + 2^3 + 2^1 + 2^0.$
$= 16 + 8 + 2 + 1.$
$= 27.$

Example: 1110.0101_2 = ?

Answer : get inverse :

$$\begin{array}{r} 10000.0000 \\ -\ 1110.0101 \\ \hline 0001\ 1111 \end{array}$$

$16 + 8 + 2 + 1 = 27.$

final answer = -27.

Binary Arithmetic in two's complement:

- When signed numbers are represented in two's complement, addition of signed numbers do not require special conversion. ••• Think: people operate on signed numbers in different ways.

- Example: (8 bits) With 8 bits

ADD

$$\begin{array}{r} 3 \\ +\ 5 \\ \hline 8 \end{array} \qquad \begin{array}{r} 0000.0011 \\ +\ 0000.0101 \\ \hline 0000.1000 \end{array}$$

Actually ↓ SUBTRACT

$$\begin{array}{r} 3 \\ +\ -5 \\ \hline -2 \end{array} \qquad \begin{array}{r} 0000.0011 \\ +\ 1111.1011 \\ \hline 1111.1110 \end{array} \qquad \begin{array}{r} 0000.0101 \\ 1111.1010 \\ +1 \\ \hline 1111.1011. \end{array}$$

↓ neg.

$$\begin{array}{r} 0000.0001 \\ +1 \\ \hline 0000.0010 \end{array} = 2 \Rightarrow \boxed{-2}$$

$$\begin{array}{r} -3 \\ +\ 5 \\ \hline 2. \end{array} \qquad \begin{array}{r} 1111.1101 \\ +\ 0000.0101 \\ \hline (1)\ 0000.0010 \end{array} = 2. \qquad \begin{array}{r} 0000.0011 \\ 1111.1100 \\ +\ 1 \\ \hline 1111.1101 \end{array}$$

actually subtract.

discard

actually ADD!!!

$$\begin{array}{r} -3 \\ + \quad -5 \\ \hline -8. \end{array} \qquad \begin{array}{rr} & 1111.1101 \\ + & 1111.1011 \\ \hline 1 & 1111.1000 \end{array}$$

discard.

neg.

$$\begin{array}{r} 0000.0111 \\ +1 \\ \hline 0000.1000 \end{array} = 8. \rightarrow -8.$$

$$\begin{array}{r} 3 \\ -5 \\ \hline \end{array} \qquad \begin{array}{r} 0000.0011 \\ -\ 0000.0101 \\ \hline \cancel{1111.1010} \\ 1111.1110 \end{array} = \underline{\underline{-2}}.$$

$$\begin{array}{r} 3 \\ -\ (-5) \\ \hline \end{array} \qquad \begin{array}{r} 0000.0011 \\ -\ 1111.1011 \\ \hline 00001000 \end{array} = \underline{\underline{8}}.$$

$$\begin{array}{r} -3 \\ -\quad 5 \\ \hline \end{array} \qquad \begin{array}{r} 1111.1101 \\ -\ 0000.0101 \\ \hline 11111000 \end{array} = -8.$$

demo/int1.c

$$\begin{array}{r} -3 \\ -\ (-5) \\ \hline \end{array} \qquad \begin{array}{r} 1111.1101 \\ -\ 1111.1011 \\ \hline 00000010 \end{array} = 2.$$

How is this "magic" achieved?

Consider the case with $n = 3$ bits.

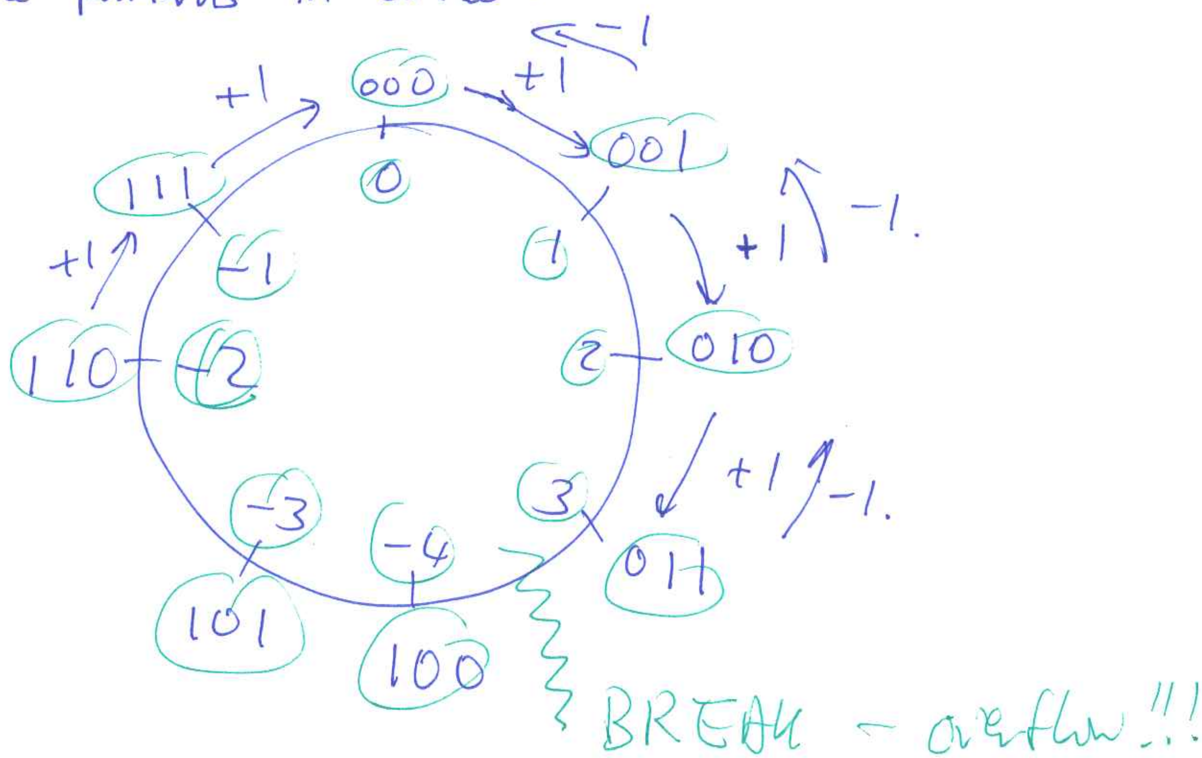
The 3-bit patterns represent the following numbers:

bit pattern	signed decimal value
000	= 0
001	= 1
010	= 2
011	= 3
100	= −4
101	= −3
110	= −2
111	= −1.

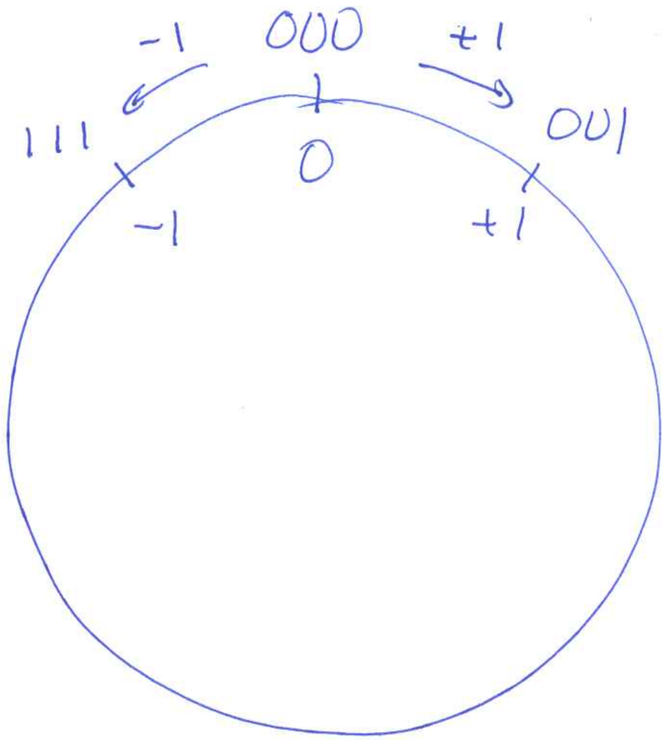
$011 + 1 = 100 (4)$

$010 + 1 = 011 (3)$

List the patterns in circle:



Draw like this:



and so on.

- 111 represents -1, so that when 1 is added to 111, we get $1000 \to 000$

which represents the addition:

$$\begin{array}{r} -1 \\ +\ \ 1 \\ \hline 0 \end{array}.$$

- Add 1 = move from pattern to the pattern on it's right.

Sub 1 = move from pattern to the pattern on it's left.

~~Property~~

2-complement encoding:

- Advantage : operation (add, subtract, multiply) on positive AND negative numbers are identical.

  (ie : (pos) + (neg)

  is just add

  No need to :

  (1) convert neg to pos.

  (2) subtract.

- This advantage is gained by

  ENCODING

  NOT by working in the binary system.

- Same advantage is achieved in

  "10's complement encoding".

Overflow

- With a fixed (limited) number of digits, it is impossible to represent all possible values.

  Some values cannot be represented by a given number of digits.

- In some arithm. operations, it can result in a value that is not representable.

  In these cases, we say "an overflow has occured" and the result of the operation is INCORRECT!

- Example of overflow:

  adding : overflow can only happen when we add 2 numbers of same sign.

  subtract : overflow can only happen when we subtract 2 numbers of different signs !

Examples:

$(100)_{10}$ = 0110.0100
$(26)_{10}$ = 0001.1010 +

$(126)_{10}$ = 0111.1110

positive result. no overflow.

$(100)_{10}$ = 0110.0100
$(100)_{10}$ = 0110.0100 +.

1100.1000

negative result.

overflow detected
pos + pos
→ neg result!

1100.1000 represents:

1100.1000

0011.0111
+1

0011.1000 = 8+16+32
= 56

−56. not +200.

What can we do about overflow?

- Use more bits for representation.

  eg: the addition $(100)_{10} + (100)_{10}$ will not result in overflow with 16 bits:

```
  0000.0000.0110.0100
+ 0000.0000.0110.0100
---------------------
  0000.0000.1100.1000
            32 16 8 4 2 1
  ↑
positive
result
```

$$8 + 64 + 128 = 200.$$

- Conclusion:

  Computer has a need to convert 2's complement representations of various length

  eg: 8 bits → 16 bits → 32 bits ect.

Example:

Value	8 bit 2's compl	16 bit 2's compl repr
3	0000.0011	0000.0000.0000.0011
−3	1111.1110	1111.1111.1111.1110

What's the difference in representation?

How does conversion behave:

Key: the value represented in each representation system must be the same (unchanged).

- How do we convert: eg: 8 bit → 16 bit.

  (1) Look at the sign bit in the 8 bit repn

  (2) repeat this signbit into the left most part of the 16 bit repr.

eg: 0000.0011 → 0 . . . . 0. 0000.0011 (repeat 0)

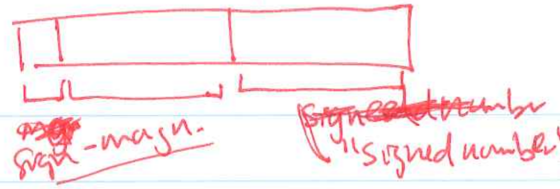
1111.1110 → $\underbrace{1 \ldots 1.\ 1111.1110}_{\text{16 bit repr.}}$ (repeat 1)

- Bottom line: extend sign bit towards left.
- This operation is thus also called
  "sign extension".

demo/int2.c

I think we should skip it.

float :



Excess encoding

- Let m = # bits in the representation.
- The excess encoding corresponding to m bits is called "excess 2^{m-1}".

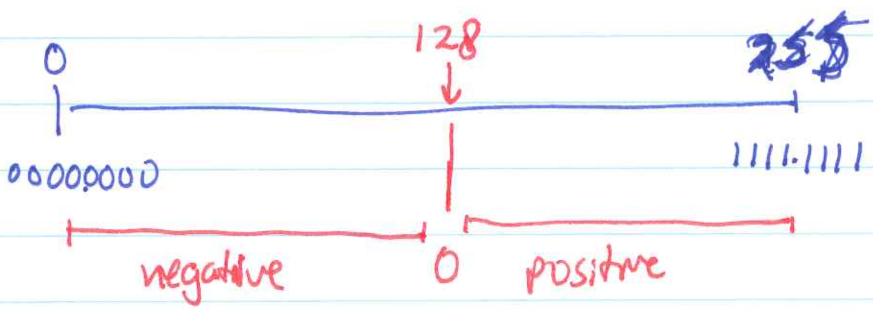
eg: $m = 8$ bits.

The excess encoding scheme is: "excess 128".

- In excess 2^{m-1} encoding scheme, the value x is represented by the <u>unsigned</u> value $x + 2^{m-1}$.

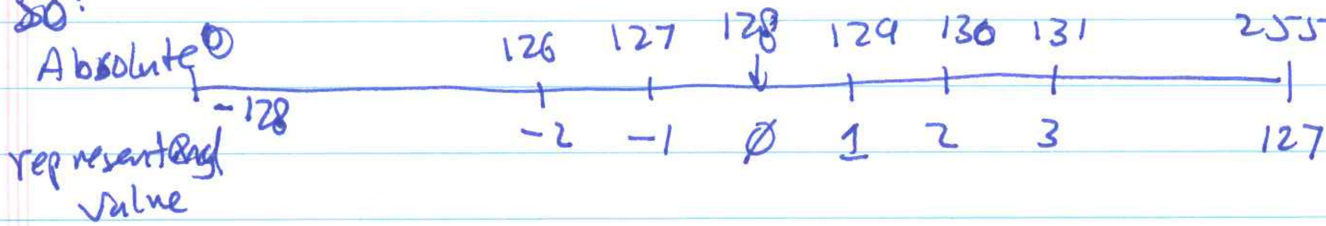
For example: in excess 128 $(m = 8)$, the value of x is represented by the unsigned value $x + 128$.

Reason: with 8 bits, the unsigned values represented are:

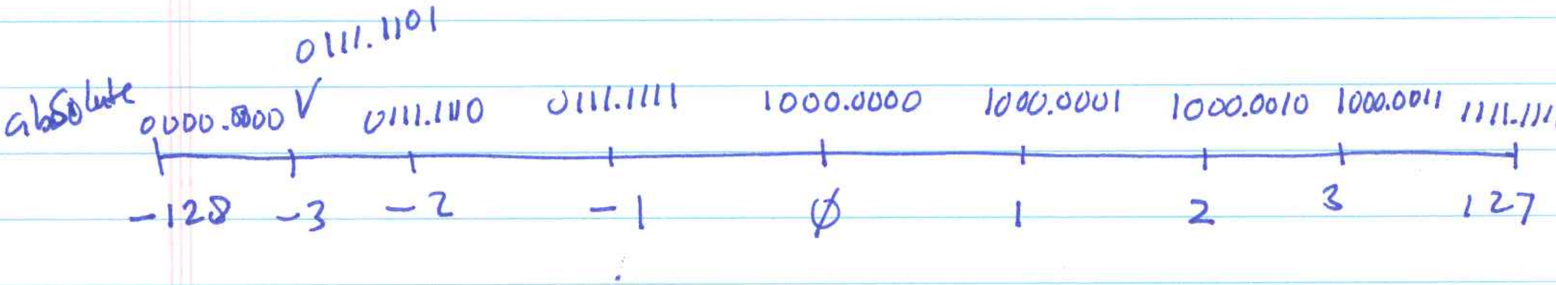


We use half of the numbers to represent negative values and half to represent positive.

So:



- Excess encoding in binary:



Converting a value to its excess 128 representation:

given value x.
Add 128 → x+128
Write x+128 in unsigned representation.

~~eg: 2~~

~~Converting an excess~~

eg: x = 3 :
add 128 → 3+128 = 131.

Write 128 in unsigned : $2\overline{)131}$
⋮

or: 128 = 1000.0000
3 = 0000.0011 +

131 1000.0011.

eg: x = -3 :
add 128 → -3+128 = 125.

Write 128 in unsigned : $2\overline{)125}$
⋮

or: 128 = 1000.0000
− 3 = 0000.0011

01111101

~~Converting an exce~~

Finding the value represented by an excess 128 encoding:

(1) Find the ~~value~~ (absolute value). A

(2) Solve: $x + 128 = A$.

to obtain value x.

eg: Repr = 0110,1010

$$A = 2^6 + 2^5 + 2^3 + 2^1$$
$$= 64 + 32 + 8 + 2$$
$$= 106.$$

Solve: $x + 128 = 106$

$$x = 106 - 128$$
$$= -22.$$

ϕ.

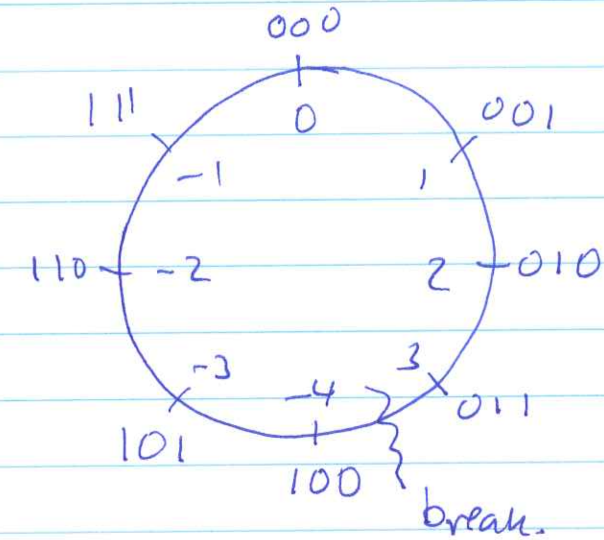
Repr = 1010.0110

$$A = 2^7 + 2^5 + 2^2 + 2^1$$
$$= 128 + 32 + 4 + 2$$
$$= 166$$

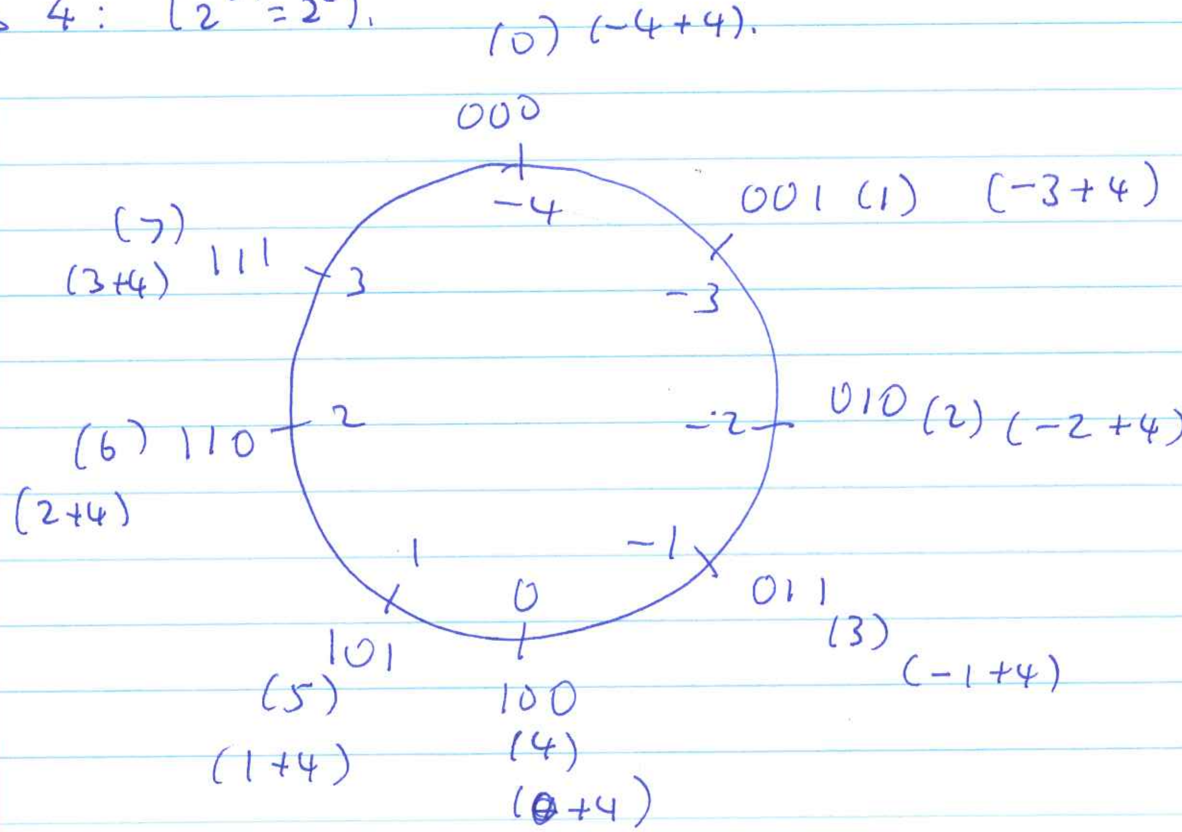
Solve: $x + 128 = 166$
$$x = 166 - 128$$
$$= +38.$$

Excess 2^{m-1} codes are very similar to 2's complement codes in nature:

2's complement



Excess 4: $(2^{3-1}=2^2)$.



It's shifted.