Complement Arithmetic

Objectives

In this lesson, you will learn:

- How additions and subtractions are performed using the complement representation,
- What is the Overflow condition, and
- How to perform arithmetic shifts.

Summary of the Last Lesson

Basic Rules

1. Negation is replaced by complementing (**- N** → **N'**)
2. Subtraction is replaced by addition to the complement.
   - *Thus,* (X – Y) *is replaced by* (X + Y')
3. For some number **N,** its complement **N' is** computed as **N'** = M –N, *where*
   - M = r^n for **R's** complement representation, where *n* is the number of *integral* digits of the register holding the number.
   - M = (r^n – *ulp*) for **(R-1)'s** complement representation
4. The operation **Z= X–Y,** *where* both X and Y are positive numbers (computed as **X + Y'**) yields two different results depending on the relative magnitudes of X & Y. (*Review page 12 of the previous lesson*).

a) First case Y > X → **(Negative Result)**

- The result **Z** is **–ive**, *where*

$$Z = -(Y-X) \rightarrow$$

- Being –ive**, Z** should be represented in the *complement form* as

$$\boxed{Z = M-(Y-X)} \quad \textbf{(1)}$$

Expected Correct Result

- Using the complement method:

$$Z = X + Y'$$

$$= X + (M-Y), \text{ i.e.}$$

$$\boxed{Z = M - (Y-X)} \qquad (2) \longleftarrow$$

Computed Result

= *Correct Answer in the Complement Form*

- ❑ **In this case, any value of M gives correct result.**

Note In this case the result fits in the n-digits of the operands. In other words, there is *no end carry* irrespective of the value of M.

Second case Y < X → **(Positive Result)**

The result **Z** is **+ive** *where,*

$$Z = +(X-Y).$$

Using complement arithmetic we get:

$$Z = X + Y'$$

$$= X + (M-Y)$$

$$\boxed{Z = M + (X-Y)} \quad (3) \longleftarrow$$

Computed Result

- which is *different* from the *expected correct result* **of**

$$\boxed{Z = +(X-Y)} \qquad (4)$$

Expected Correct Result

- ❑ In this case, a *correction step* is required for the final result.
- ❑ The *correction step* depends on the value of M.

Correction Step for R's and (R-1)'s Complements

The previous analysis shows that computing **$Z = (X-Y)$** using complement arithmetic gives:

- The correct complement representation of the answer if the result is negative, that is $\boxed{\mathbf{M - (Y-X)}}$.
- *Alternatively,* if the result is positive it gives an answer of $\boxed{\mathbf{M + (X-Y)}}$ which is *different* from the *correct answer* **of** $\boxed{\mathbf{+(X-Y)}}$ requiring a correction step.
- The correction step depends on the value of M

For the R's Complement

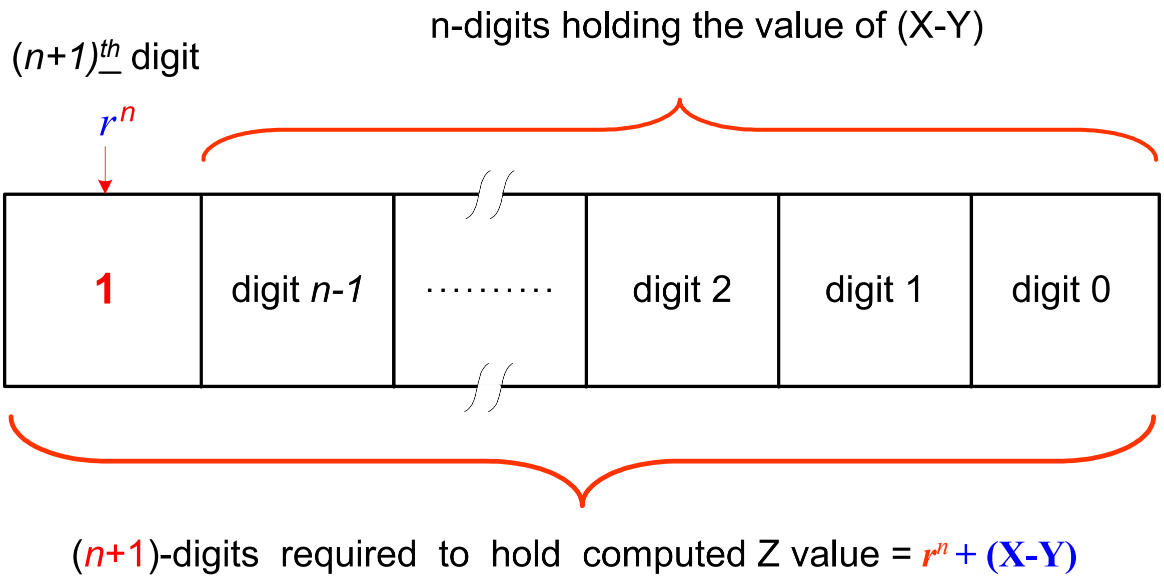
Note that

$$\mathbf{M}_R = r^n = \underset{(n+1)^{th}\ Position}{1}\underbrace{000\ldots00}_{n\ Positions}.000$$

Thus, the computed result (**M + (X-Y)**) is given by

$$Z = r^n + \mathbf{(X-Y)}$$

Since **(X-Y)** is *positive*, the computed Z **value** $\{r^n + \mathbf{(X-Y)}\}$ requires $(n + 1)$ integral digits to be expressed as shown in Figure.



In this case, it is clear that **$Z = r^n + (X\text{-}Y)$** consists of the digit **1** in the $(n+1)^{th}$ digit position while the least significant *n* digits will hold the *expected correct result* of (X-Y).

Since X, Y, and the result Z are stored in registers of *n* digits, the correct result (X- Y) is simply obtained by neglecting the 1 in the $(n+1)^{th}$ digit.

The **1** in the $(n+1)^{th}$ digit is typically referred to as "*end carry*".

Conclusion:

- For the R's complement method;
  - i. If the computed result has *no end carry*. This result is the correct answer.
  - ii. In case the computed result has an *end carry*, this end carry is *DISACRDED* and the remaining digits represent the correct answer.

For the (R-1)'s Complement

- $M_{R-1} = r^{n}- ulp$

Thus, the computed result (**M + (X-Y)**) is given by

$$Z = (r^n - ulp) + (X\text{-}Y)$$

For a *positive* value of **(X-Y)**, the computed Z **value** $\{(r^n - ulp) + (X\text{-}Y)\}$ requires $(n + 1)$ integral digits for its representation.

Again, r^n represents a **1** in the $(n+1)^{th}$ digit position (i.e. an end carry) while the least significant *n* digits will hold the value (X-Y-*ulp*).

Since the *expected correct answer* is (X-Y), the correct result is obtained by adding a *ulp* to the least significant digit position.

Q. What does the computed result represent in case X=Y **?**

Conclusion:

- For the (R-1)'s complement method;
  - a. If the computed result has no end carry. This result is the correct answer.
  - b. In case the computed result has an end carry, this end carry is added to the least significant position (i.e., as *ulp*).

Important Note:

- *The previous conclusions are valid irrespective of the signs of X or Y and for both addition and subtraction operations.*

Add/Subtract Procedure

It is desired to compute **Z = X ± Y**, *where* **X, Y** and **Z**:

(a) are signed numbers represented in one of the complement representation methods.

(b) have *n* integral digits including the sign digit.

The procedure for computing the value of **Z** depends on the used complement representation method:

R's Complement Arithmetic

1. If the operation to be performed is addition compute **Z = X + Y**, otherwise if it is subtraction, **Z = X – Y**, compute **Z = X + Y'** instead.
2. If the result has no end carry, the obtained value is the correct answer.
3. If the result has an end carry, discard it and the value in the remaining digits is the correct answer.

(R-1)'s Complement Arithmetic

1. If the operation to be performed is addition compute **Z = X + Y**, otherwise if it is subtraction, **Z = X – Y**, compute **Z = X + Y'** instead.
2. If the result has no end carry, the obtained value is the correct answer.
3. If the result has an end carry, this end carry should be added to the least significant digit (*ulp*) to obtain the final correct answer.

Examples

RADIX COMPLEMENT

Compute (M-N) and (N-M), *where* **M=$(072532)_{10}$ N=$(003250)_{10}$**

Both M & N must have the same # of Digits (*Pad with 0`s if needed*).

COMPUTING (M – N)

Regular Subtraction

M		0	7	2	5	3	2
N	–	0	0	3	2	5	0
		0	6	9	2	8	2

Complement Method

Compute (M+N’)

M		0	7	2	5	3	2
N’	+	9	9	6	7	5	0
	1	0	6	9	2	8	2

Correct Result

Discard End Carry

COMPUTING (N – M)

Regular Subtraction

N		0	0	3	2	5	0
M	–	0	7	2	5	3	2
	–	0	6	9	2	8	2

-ive sign

Equivalent Results
The –ive Result is Represented by the 10's Complement

Complement Method

Compute (N + M')

N		0	0	3	2	5	0
M'	+	9	2	7	4	6	8
		9	3	0	7	1	8

No End Carry

This is the 10's complement representation of a –ive number, i.e. the result (930718) represents the number (-069282)

Example : **(2`s Comp) M=$(01010100)_2$ N=$(01000100)_2$**

Note: Both M & N are positive 8-bit numbers

COMPUTING (M – N)

Regular Subtraction

M		0	1	0	1	0	1	0	0
N	–	0	1	0	0	0	1	0	0
		0	0	0	**1**	0	0	0	0

Correct Result

Complement Method

Compute (M+N')

M			0	1	0	1	0	1	0	0
N'	+		1	0	1	1	1	1	0	0
		1	0	0	0	**1**	0	0	0	0

Sign Bit

Discard
Carry Out

COMPUTING (N – M)

Regular Subtraction

N		0	1	0	0	0	1	0	0
M	–	0	1	0	1	0	1	0	0
	–	0	0	0	1	0	0	0	0

-ive sign

Equivalent Results The –ive Result is Represented by the 2's Complement

Complement Method

Compute (N + M')

N		0	1	0	0	0	1	0	0
M'	+	1	0	1	0	1	1	0	0
		1	1	1	1	0	0	0	0

Sign Bit

No End Carry

This is the 2's complement representation of a –ive number, i.e. the result (11110000) represents the number (-00010000)

DIMINISHED / (R-1)'s RADIX COMPLEMENT

Compute (M-N) and (N-M), *where* **$M=(072532)_{10}$ $N=(003250)_{10}$**

Both M & N must have the same # of Digits (*Pad with 0`s if needed*).

COMPUTING (M – N)

Regular Subtraction

$$\begin{array}{llcccccc} M & & 0 & 7 & 2 & 5 & 3 & 2 \\ N & - & 0 & 0 & 3 & 2 & 5 & 0 \\ \hline & & 0 & 6 & 9 & 2 & 8 & 2 \end{array}$$

Complement Method

Compute (M+N')

$$\begin{array}{llccccccc} M & & & 0 & 7 & 2 & 5 & 3 & 2 \\ N' & + & & 9 & 9 & 6 & 7 & 4 & 9 \\ \cline{4-9} & & 1 & 0 & 6 & 9 & 2 & 8 & 1 \\ & + & & & & & & & 1 \\ \cline{4-9} & & & 0 & 6 & 9 & 2 & 8 & 2 \end{array}$$

End Carry

Correct Result

COMPUTING (N – M)

Regular Subtraction

N		0	0	3	2	5	0
M	–	0	7	2	5	3	2
	–	0	6	9	2	8	2

-ive sign

Equivalent Results
The –ive Result is Represented by the 9's Complement

Complement Method

Compute (N + M')

N		0	0	3	2	5	0
M'	+	9	2	7	4	6	7
		9	3	0	7	1	7

No End Carry

This is the 9's complement representation of a –ive number, i.e. the result (930717) represents the number (-069282)

Example : **(1`s Comp) M=(01010100)$_2$ N=(01000100)$_2$**

Note: Both M & N are positive 8-bit numbers

COMPUTING (M – N)

Regular Subtraction

M		0	1	0	1	0	1	0	0
N	–	0	1	0	0	0	1	0	0
		0	0	0	**1**	0	0	0	0

Complement Method

Compute (M+N’)

Sign Bit

Correct Result

M			0	1	0	1	0	1	0	0
N’	+		1	0	1	1	1	0	1	1
		1	0	0	0	0	1	1	1	1
										1
			0	0	0	**1**	0	0	0	0

End Carry

COMPUTING (N – M)

Regular Subtraction

N		0	1	0	0	0	1	0	0
M	–	0	1	0	1	0	1	0	0
	–	0	0	0	1	0	0	0	0

-ive sign

Equivalent Results The –ive Result is Represented by the 1’s Complement

Complement Method

Compute (N + M’)

N		0	1	0	0	0	1	0	0
M’	+	1	0	1	0	1	0	1	1
		1	1	1	0	1	1	1	1

Sign Bit

No End Carry

This is the 1’s complement representation of a –ive number, i.e. the result (11101111) represents the number (-00010000)

Overflow Condition

- If adding two *n*-digit *unsigned* numbers results in an *n*+1 digit sum, this represents an *overflow* condition.
- In digital computers, overflow represents a problem since register sizes are fixed, accordingly a result of *n*+1 bits cannot fit into an n-bit register and the most significant bit will be lost.
- Overflow condition is a problem whether the added numbers are signed or unsigned.
- In case of signed numbers, overflow may occur only if the two numbers being added have the same sign, i.e. either both numbers are positive or both are negative.
- For 2's complement represented numbers, *the sign bit is treated as part of the number and an end carry does not necessarily indicate an overflow*.
- In 2's complement system, an overflow condition always changes the sign of the result and gives an erroneous n-bit answer. Two cases are possible:
  1. Both operands are positive (sign bits=0). In this case, an overflow will result from a carry of 1 into the sign bit column; causing the sum to be interpreted as a negative number.
  2. Both operands are negative (sign bits=1). In this case, an overflow will result when no carry is received at the sign bit column causing the two sign bits to be added resulting in a 0 in the sign bit column and a carry out in the (*n*+1)th. bit position which will be discarded. This causes the sum to be interpreted as a positive number.
- Accordingly, an overflow condition is detected if one of the two following conditions occurs:
  - (a) There is a carry into the sign bit column but no carry out of that column.
  - (b) There is a carry out of the sign bit column but no carry into that column.

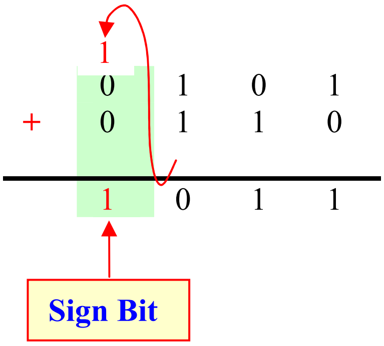
Example:

- Consider the case of adding the binary values corresponding to $(+5)_{10}$ and $(+6)_{10}$ where the correct result should be (+11).
- Even though the operands $(+5)_{10}$ & $(+6)_{10}$ can be represented in 4-bits, the result $(+11)_{10}$ cannot be represented in 4-bits.
- Accordingly, the 4-bit result will be erroneous due to "*overflow*".

Add (+5) to (+6) using 4-bit registers and 2's complement representation.

$(+5)_{10} \rightarrow (0101)_2$

$(+6)_{10} \rightarrow (0110)_2$



*There is a carry **into** the sign bit column but no carry **out** of it*

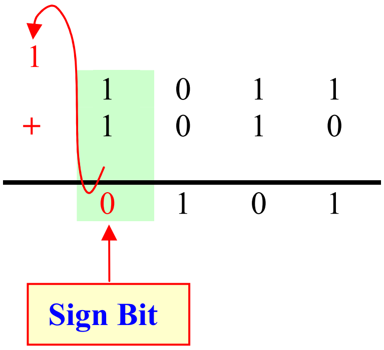
- If this overflow condition is not detected, the resulting sum would be erroneously interpreted as a negative number (1011) which equals $(-5)_{10}$.

Example:

Add (-5) to (-6) using 4-bit registers and 2's complement representation.

$(-5)_{10} \rightarrow (1011)_2$

$(-6)_{10} \rightarrow (1010)_2$



There is a carry out of the sign bit column but no carry into it.

- If this overflow condition is not detected, the resulting sum would be erroneously interpreted as a positive number (0101) which equals $(+5)_{10}$.

Example:

Using 8-bit registers, show the ***binary*** number representation of the decimal numbers (37), (-37), (54), and (-54) using the following systems:

	Signed magnitude system	**Signed 1's complement System**	**Signed 2's complement system**
37	00100101	00100101	00100101
-37	10100101	11011010	11011011
54	00110110	00110110	00110110
-54	10110110	11001001	11001010

Compute the result of the following operations in the ***signed 2's complement*** system.

I. (+37) – (+54)

Subtraction is turned into addition to the complement, i.e.

(+37) – (+54) → (+37) + (+54)'

$$\begin{array}{ccccccccc} & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ + & & & & & & & & \\ & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ \hline & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{array}$$

$= (-17)_{10}$

II. (-37) – (+54)

Subtraction is turned into addition to the complement, i.e.

(-37) – (+54) → (-37) + (+54)'

$$\begin{array}{ccccccccc} & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ + & & & & & & & & \\ & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{array}$$

Discard End Carry

$= -(01011011) =- (91)_{10}$

III. (54) + (-37)

$$\begin{array}{ccccccccc} & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ + & & & & & & & & \\ & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ \hline \mathbf{1} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

Discard End Carry

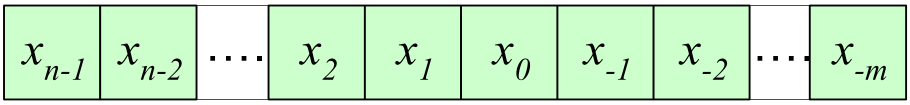
$= + (17)_{10}$

Range Extension of 2’s Complement Numbers

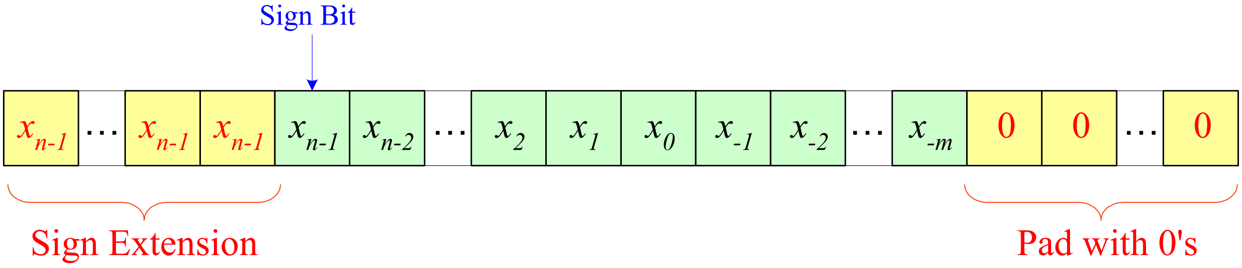
- To extend the representation of some 2’s complement number X from *n-bits to n`-bits where n` > n.
  1. If **X** is +ive → pad with 0`s to the ***right*** of fractional part and/or to the ***left*** of the integral part.
  2. If **X** is -ive → pad with 0`s to the ***right*** of fractional part and/or with 1`s to the ***left*** of the integral part.

In General

- Pad with 0`s to the ***right*** of fractional part and/or **extend sign bit to the *left* of the integral part** (*Sign Bit Extension*).



X- Before Extending its Range

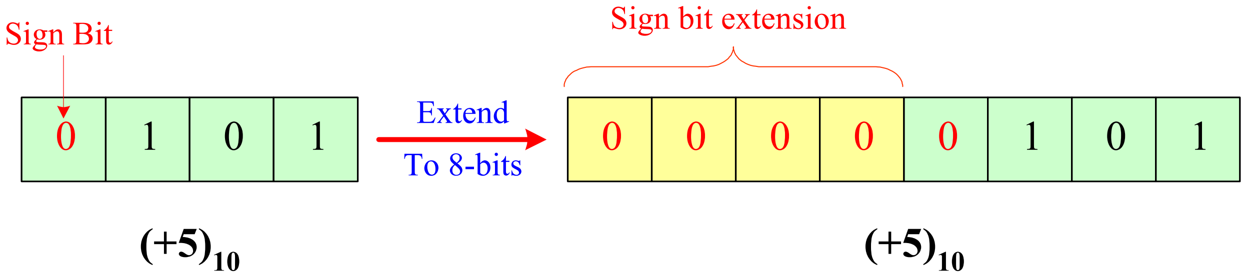


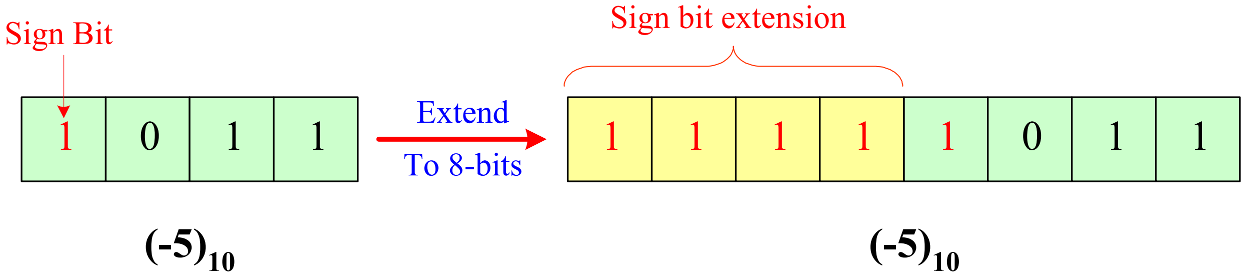
X- After Extending its Range

(*0’s Padded to the Right of Fractional Part and the Sign is Extended to the Left of the Integral Part*)

Example:

Show how the numbers $(+5)_{10}$ and $(-5)_{10}$ are represented in 2’s complemenr using 4-bit registers then extend this representation to 8-bit registers.





Arithmetic Shifts

Effect of 1-Digit Shift

- Left Shift → *Multiply* by radix ***r***
- Right Shift → *Divide* by radix ***r***

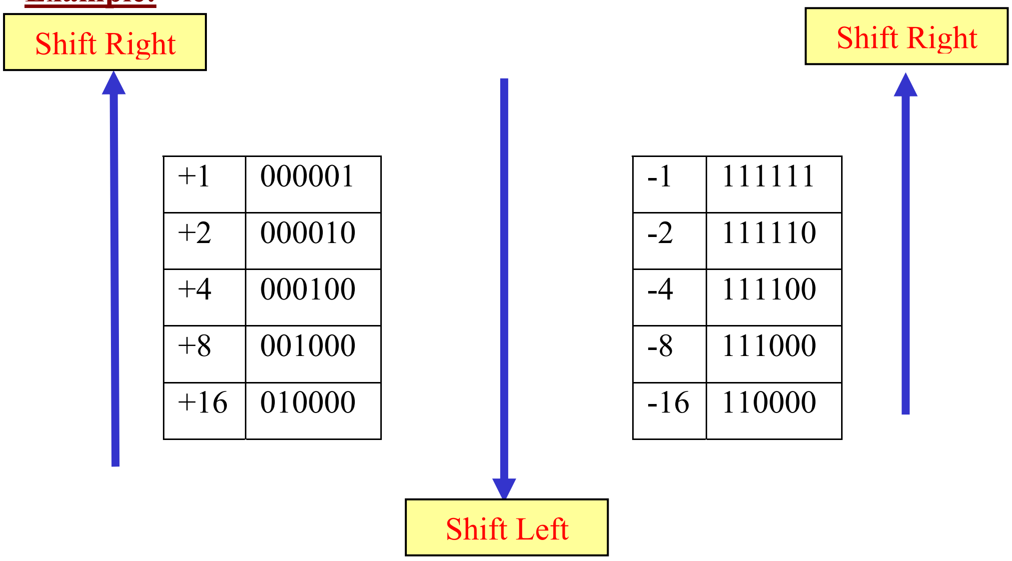
(a) **Shifting Unsigned Numbers**

- Shift-in **0`s** (for both Left & Right Shifts)

(b) **Shifting 2's Complement Numbers**

- Left Shifts: 0`s are shifted-in
- Right Shifts: *Sign Bit Extended*

Example:



+1	000001
+2	000010
+4	000100
+8	001000
+16	010000

-1	111111
-2	111110
-4	111100
-8	111000
-16	110000