2.5.2 Offset Binary representation (Excess-K)

- Offset Binary is where one subtracts K (usually half the largest possible number) from the representation to get the value.
- Has the advantage that the number sequence from the most negative to the most positive is a *simple binary progression*, which makes it a natural for binary counters.
- note that the *MSB* still carries the sign information.
- *Excess K* is used in conjunction with floating point representations for the *exponent*. We will meet this again shortly.
- A note on arithmetic in Excess K:

  Assume a, b, c are three values:

  $$(a + b) = c \qquad \text{(values)}$$

  $$\begin{aligned}&(a + k) + (b + k) \quad \text{(representations)}\\ &= (a + b) + 2k\\ &= (c + k) + k\end{aligned}$$

  Rewriting A, B, C in *Excess-K* representations:

  $$\begin{aligned}A + B &= C + k\\ C &= (A + B) - k\end{aligned}$$

- Try this for (-1) + (+1) = 0

2.5.3 2's complement

- 2's complement represents the method most widely used for integer computation.
- Positive numbers are represented in simple unsigned binary.
- The system is rigged so that a negative number is represented as the binary number that when added to a positive number of the same magnitude gives zero.
- To get the two's complement, first take the ones complement, then add one.

2.5.4 1's complement

- Exchange all the 1's for 0's and vice versa.

value	1's complement	2's complement
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
0	0000	0000
-1	1110	1111
-2	1101	1110
-3	1100	1101
-4	1011	1100
-5	1010	1011
-6	1001	1010
-7	1000	1001
-8	-	1000
-0	1111	-

2.6 Performing Arithmetic

2.6.1 In 1's complement

Some examples of arithmetic with 1's complement.

Example 2.6.1 (addition)

$$\begin{array}{rl} 0011 & (+3) \\ \underline{+0010} & (\underline{+2}) \\ 0101 & (+5) \end{array}$$

Example 2.6.2 (subtraction)

$$\begin{array}{rl} 0011 & (+3) \\ \underline{+1101} & (\underline{-2}) \\ (1)0000 & (0?) \end{array}$$

Example 2.6.3 (subtraction from a negative)

1100	*(-3)*
+1101	*(-2)*
(1)1001	*(-6?)*

The solution is to wrap the carry back in to the LSB.

Exercise 2.6.4 *Can you explain why this works?*

2.6.2 In 2's complement

The Arithmetic operations are perhaps easiest in 2's complement.

- *To add* ... just like in any other base.

Example 2.6.5 (addition 5 + (-2):)

0101	*(+5)*
+1110	*(-2)*
0011	*(+3)*

- *To subtract* B from A take the 2's complement of B and add to A.

Example 2.6.6 (subtraction 2 - 5:)

0010	*(+2) (2 + (-5))*
+1011	*(-5) since +5 = 0101:*
1101	*(-3)*

value	Sign Magnitude	Offset Binary	2's complement
+7	0111	1111	0111
+6	0110	1110	0110
+5	0101	1101	0101
+4	0100	1100	0100
+3	0011	1011	0011
+2	0010	1010	0010
+1	0001	1001	0001
0	0000	1000	0000
-1	1001	0111	1111
-2	1010	0110	1110
-3	1011	0101	1101
-4	1100	0100	1100
-5	1101	0011	1011
-6	1110	0010	1010
-7	1111	0001	1001
-8	-	0000	1000
-0	1000	-	-

- Multiplication also works right in 2's complement. Long multiplication reduces to shifts and adds
- We have implicitly used the concept of carry. In particular we dropped/ignored the carry bit in the case of the two's complement number representation.

Example 2.6.7 (3 − 3 =)

0011 (3)
+1101 (−3)
(1) 0000 (0)

(c.f. above 1's complement example)

- It should be clear that for the unsigned binary the *carry has relevance*.

Example 2.6.8 (3 + 13 =)

0011	*(3)*
+1101	*(+13)*
(1) 0000	*(16)*

- We can also perform subtraction directly and still ignore the carry/borrow bit.

Example 2.6.9 (2's complement borrow)

0011	*(3)*
–0100	*(–4)*
(1) 1111	*(–1)*

- However, for subtraction with the *unsigned binary* the borrow is important, particularly in multiple word operations.

Example 2.6.10 (multiple word addition/carry)

0011 0011	*(51)*
+0000^{1} 1101	*(13)*
0100 0000	*(64)*

Example 2.6.11 (multiple subtraction/borrow)

0100 0000	*(64)*
–0000^{1} 1101	*(–13)*
0011 0011	*(51)*

- These still could represent two's complement. but low order words must treated as if unsigned.
