Binary Addition & Subtraction

Unsigned and Sign & Magnitude numbers

Addition and subtraction of unsigned or sign & magnitude binary numbers ‘by hand’ proceeds exactly as with decimal numbers. (In fact this is true for any radix numbers)

Examples - Addition (unsigned)

$$\begin{array}{rrr} 30. & 31. & 32. \\ \\ 0110010_2 & 31_5 & A3D_{16} \\ +\,1100110_2 & +\,14_5 & +\;\;F2_{16} \\ \hline 10011000_2 & 100_5 & B2F_{16} \end{array}$$

If the result has more digits than can fit in the assigned storage word (i.e, there is a carry out of the last bit that can be held), then an overflow has occurred. Usually a computer error handler (program) is invoked to deal with the problem.

Examples - Subtraction (unsigned)

$$\begin{array}{rrr} 33. & 34. & 35. \\ \\ 0110010_2 & 14_5 & A3D_{16} \\ -\,0010110_2 & -\,31_5 & -\;\;F2_{16} \\ \hline 0011100_2 & -12_5 & 94B_{16} \end{array}$$

As in decimal, if the numbers are of different signs:

1. when adding you subtract the smaller from the larger and assign the sign of the larger number;
2. when subtracting, you must keep track of whether the negative number is the subtrahend (in which case you change its sign and add, getting a positive result) or the minuend (in which case you add without changing any signs, and the result is negative.)

Practice Problems - Unsigned Arithmetic

1. Do the following binary additions

```
a.    1001101       b.    11111 c.    101010
    + 1011011           + 00001      +010101
    ---------------     -------------  --------------
```

2. Do the following binary subtractions

```
a.    100000        b.    110010      c.    1111111
    -  000001            - 010001           - 1010101
    --------------      -------------     -----------------
```

2's complement numbers

When the numbers are in 2's complement form, there is no need to keep track of whether the numbers have different signs and, if so, which is bigger. In the case of subtraction, however, the subtrahend must have its sign reversed; the numbers are then simply added normally. This sign reversal is accomplished by the complement and add procedure we've already discussed.

Examples - Addition (2's complement in 5-bit registers)

```
36.  01001        37.  01001        38.  01001        39.  10101
   + 00011           + 01001         + 11111            + 11010
     01100             10010           1  01000           1  01111
```

In 36, both numbers are positive and the expected result is achieved (in decimal, 9 + 3 = 12).

In37, both numbers are again positive, but the result is negative (note the leading 1)! It appears that 9 + 9 = - 14. This is considered an **overflow** situation, even though no carry actually overflowed out of the most significant bit. In a computer, hardware is implemented to detect such overflows, and they do so as follows.

Consider the two most significant bit positions, b_{n-1} and b_{n-2}. Call the carries out of these c_{n-1}, c_{n-2}, where c_{n-1} is the bit that actually overflows into the next (non-existent) bit b_n. Then an overflow is detected whenever $c_{n-1} <> c_{n-2}$. In example b), $c_{n-1} = 0$ (there is no carry) and c_{n-2} is 1 (there is a carry); Since they are different, an overflow is detected, and the answer is known to be incorrect.

In 38, one number is positive and the other is negative. There *appears* to be an overflow, but in fact the answer (ignoring c_{n-1}) is correct (9 + (- 1) = 8). Notice that the conditions for detecting an actual overflow as described above do not exist: both c_{n-1} (c_5)and c_{n-2} (c_4) are 1; since they are the same, there is no overflow detected, and the answer, ignoring $c_{5,}$ is correct.

In 39, both numbers are negative. The answer appears to be +15, an unlikely result when adding two negative numbers. However, since c_4 and c_3 are different (c_3=0, c_4=1), we know that an overflow has occurred, and therefore the answer is incorrect.

Examples - subtraction (2's complement in 5-bit registers)

$$\begin{array}{rrrrrrrr} 40. & 01001 & 41. & 11001 & 42. & 01001 & 43. & 10101 \\ & \underline{-\ 00011} & & \underline{-\ 01010} & & \underline{-\ 11111} & & \underline{-\ 11010} \end{array}$$

In each of the above cases, the subtrahend must be converted to its 2's complement value and then normal addition is done.

$$\begin{array}{rrrr} 01001 & 11001 & 01001 & 10101 \\ \underline{+\ 11101} & \underline{+\ 10110} & \underline{+\ 00001} & \underline{+\ 00110} \\ 1\ 00110 & 1\ 01111 & 01010 & 11011 \end{array}$$

Checking for overflows in each of these examples shows one only in example 41. We can conclude, then, that all the results except41 are correct.

Practice Problems - 2's Complement Arithmetic

1. Do the following 2's complement additions. In all cases assume results are placed in an 8-bit register. For each problem, indicate whether the addition resulted in an overflow.

```
a.   01100001   b.   01100001   c.   01101111   d.   11110000
     00001111        00101110        11111111        00010000
     ------------    ------------    -------------   ------------
```

2. Do the following 2's complement subtractions. In all cases assume results are placed in an 8-bit register. For each problem, indicate whether the subtraction resulted in an overflow.

```
a.    01010100   b.   00000110   c.   10000000   d.   11100011
   -  00101001      - 10000011      - 00000001      - 11100011
   -------------    -------------   -------------   -------------
```

BCD numbers

Consider the addition of the two unsigned BCD numbers 29 and 37.

```
44.    0010 1001
     + 0011 0111
       0110 0000
```

The answer shown, 60, is clearly incorrect. It looks like it might be correct, since both digits are less than 9. Consider another example, 16 and 69.

```
45.   0001 0110
    + 0110 1001
      0111 1111
```

This answer is clearly wrong since one of the digits is not a legal decimal digit. You might ask, why are we doing binary arithmetic on BCD numbers? Shouldn't we be building hardware to do BCD arithmetic. The answer is that, yes we could build hardware to do this (some IBM System/360 models did this), but it is very wasteful to design and build special (BCD) hardware that duplicates the function of other (binary) hardware, if the binary hardware can be modified inexpensively to do the same job.

In the case of BCD arithmetic, this is easy to do, and it is done using something called **excess-3 coding** (or excess-6 coding, or both). Excess-3 coding simply recodes every digit to be used in the arithmetic calculation by adding 3 to it, as in Table TN5

Decimal	BCD	Excess-3
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

Table TN5. Excess-3 Coding

Now consider example 44. Rewriting

```
   0010 1001 (29)
+  0011 0111 (37)
```

in excess-3 coding results in

```
                                   0101 1100
                                +  0110 1010
giving an answer of                1100 0110 (C6)
```

This still isn't correct, but notice: the low order nybble does have the correct answer (6), and the high order nybble differs from the correct answer by 6. In fact, we should expect both digits to be 6 more than they should be since we added 3 to all of the digits before we started. As we can see, we need to subtract 6 from the first digit and leave the second digit alone. The reason is, again, involved with carries, or overflows, out of the digits. The general rule is,

1. if the digit has a carry out of it, it will have the correct answer;
2. if there is no carry out of a digit, then we must subtract six from the digit to correct it.

Applying this principle to the current example, the first digit had no carry out, while the second digit did. The adjusted result, after subtracting six from the first digit, is 0110 0110 (66). Let's apply these rules to example 45..

```
  0001 0110 (16)
+ 0110 1001 (69)
```

Adding 3 to every digit gives

```
                           0100 1001
                         + 1001 1100
    which, adding, gives   1110 0101
```

The carries in this case happen to be the same as in the previous example, so we need to subtract 6 from the first digit, giving the result 1000 0101 (85), the correct answer.

Now consider the subtraction of two BCD numbers. We will assume unsigned binary subtraction algorithms, rather than 9's complement or 10's complement numbers, as is sometimes done. Let's subtract 23 from 85

```
    1000 0101 (85)
-   0010 0011 (-23)
```

We again convert these numbers into excess-3 notation:

```
    1011 1000
-   0101 0110
```

Applying the rules of unsigned binary subtraction gives a result of 0110 0010 (62) which is the correct answer, without any adjustment being necessary! We should expect this to occur, since subtracting removes all the excess-3 coding automatically. In this case, (5+3) - (3+3) = 5+3- 3-3 = 5-3 and (8+3) - (2+3) = 8+3-2-3 = 8-2. Of course, as you might expect, we were just lucky. Let's take another example, say 81 - 23.

```
1000 0001    becomes    1011 0100
0010 0011               0101 0110
```

and subtracting gives 0101 1110. If we subtract 6 from the second digit, the answer is now correct: 0101 1000. Thus, a similar rule applies for subtraction as we used for addition, only now instead of looking for a carry out of a digit, we are looking for a borrow into a digit. If such a borrow exists, then subtracting 6 will correct the digit.

It is important to note that even though the answer may consist of all legal digits, adjustments may still be necessary. One last example addition example

```
46.     0001 0000 (10)       ==excess-3==>     0100 0011
      + 0010 0011 (23)                         0101 0110
                                               --------------
                                               1001 1001
```

Although the answer, 99, contains valid decimal digits, it is incorrect. The correct answer is 33, and we know this because neither digit had a carry out, requiring that we subtract 6 from each of them.

Practice Problems - BCD Arithmetic

1. Do the following BCD additions. For each problem, state which digit(s), if any, required adjustment.

```
a.     00100101        b.     00010000        c.     01010111
     + 01000101             + 00100110             + 10000101
    -----------------     -----------------      ------------------
```

2. Do the following BCD subtractions. For each problem, state which digit(s), if any, required adjustment.

```
a.     10010010        b.     00111000        d.     10011001
     - 00010011             -  00000110            - 01100100
     ----------------     ------------------     ------------------
```

Floating point arithmetic

Addition and subtraction of floating point numbers is the same as adding and subtraction sign & magnitude numbers, with one little adjustment: aligning the decimal points.

Consider the addition of two radix-10 floating point numbers

$$\begin{array}{r} 35.71732 \times 10^{3} \\ +\quad 451.213 \times 10^{-1} \end{array}$$

Just for the sake of discussion let's look at these numbers in their fixed-point form:

```
35717.32
    45.1213
```

Notice that we've lined up the decimal points, a necessity in fixed-point addition. It is also a necessity in floating-point arithmetic, and consists of adjusting one or the other of the exponents so that the two exponents are the same. Of course, whichever number has its exponent adjusted will also have the decimal point moved an appropriate amount.

Which number to adjust? We generally choose to **increase the size of the smaller exponent**. This is because *increasing* the exponent of a number will require moving the decimal point to the *left*. In hardware, shifting a radix point to the left is accomplished by shifting the entire number to the *right* in its register; if any digits are lost in this shifting process they are the least significant digits at the rightmost end of the register.

(Note that sometimes it is necessary to shift both numbers. Remember that in any practical implementation of floating point numbers there is a limit on the maximum and minimum values that the exponent may have. It may be necessary to adjust both exponents to ensure that they are both within the allowed range.)

In our current example, these guidelines suggest that the second number be adjusted by adding 4 to its exponent and shifting the decimal point left four places, giving

$$\begin{array}{rl} & 35.71732 \times 10^3 \\ + & .0451213 \times 10^3 \end{array}$$

The numbers can now be added, giving 35.7634413×10^3. While this is a correct answer, it is usual to normalize the result, giving a final answer of

$$3.57634413 \times 10^4$$

We can now formulate the procedure for floating point arithmetic which we can use for binary numbers as well as for decimal numbers. We will assume that floating point numbers to be added or subtracted are already in IEEE format.

1. Subtract the exponents to determine which is smaller and determine the difference.
2. Add the difference to the smaller exponent.

   Notice that since we are using biased format for the exponents we need to nothing special to determine the sign of the result.

However, we do need to ensure that the resultant exponent is less than or equal to the maximum attainable. (E.g. in IEEE single precision the maximum binary value is 11111111 (= 255_{biased} = $+128_{10}$)

3. Shift the mantissa to the right a number of places equal to the difference determined in 1. *Don't forget to add the implied '1' in after the number has been shifted one place*.
4. Add or subtract the mantissas as required by your original problem, taking into account the signs of the operands and adjusting the sign of the result, just as you would in decimal arithmetic.
5. The result of 4. may have 0, 1, or 2 significant digits to the left of the radix point (neither more not less), so it will have to be normalized, if necessary.

Example 47: Add the two floating point numbers 3E340000 and 40FC0000

3E340000 = 0 01111100 011010... = + 1.01101 x 2^{124} (exponent is excess 127)
40FC0000 = 0 10000001 11111000... = + 1.1111100 x 2^{129}

1. The first number has the smaller exponent, and the difference between the exponents is 5.

2. & 3. The first number's exponent becomes 124+5 = 129 and the mantissa is shifted right to produce .0000101101000...

4. Adding

$$\begin{array}{rl} .000010110100\ldots & \times 2^{129} \\ +\ 1.11111000\ldots & \times 2^{129} \\ \\ = 10.000000110100\ldots & \times 2^{129} \end{array}$$

5. Normalizing gives a final answer of 1.0000000110100... x 2^{130}

Notice that, since one of the numbers in step 4. will always be in the original normalized format, step 5., if required, will always involve a shift right or left (of at most one bit position) and a corresponding increase or decrease in the exponent by at most 1.

Restoring the result to IEEE notation and then to hexadecimal shorthand gives
0 10000010 00000001101000000000000 = $4100D000_{16}$

Practice problems - Binary Floating Point Arithmetic

1. Add the following Binary floating point numbers. Normalize the result.

   $11100.1000100111100.... \times 2^{100}$ and $.00110111000... \times 2^{106}$

2. Add the following IEEE floating point numbers. Express the result in IEEE hexadecimal format

   659B8000 and E5310000

3. Show the hexadecimal IEEE format for the values zero and one.