2 | Number Systems

We have a **positional number system** that tells how a number is created or represented in any base. Thus, **number = Σ (digit$_i$ * base^i)**. This says for every number (like 251):

1. Take each digit
2. Multiply it by base raised to some *i* power
3. Add up all the values

For example, $251 = (2 \times 10^2) + (5 \times 10^1) + (1 \times 10^0)$ for base 10. Notice how the powers of 10 decrease with every place value. This formula applies for any other bases.

Binary (Base 2)

Counting from 0 to 12 in binary looks like this: *0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100*. To denote a base number, you would include a base number subscript, like 1_2 to demonstrate that it is a binary number.

Binary Addition	Binary Multiplication
0+0=0	0·0=0
0+1=1	0·1=0
1+0=1	1·0=0
1+1=10	1·1=1

For addition (and multiplication) rules, see the right image:

For subtraction, the only special case, 0 - 1, would result in borrowing 1 from the highest order neighbor, like so in the picture to the right (remember that 10 is the binary equivalent of 2).

```
      0  1  1  10     0  10
   0  Ø  Ø  Ø  Ø  1  Ø  Ø  =  70
-  0  0  0  0  1  1  0  1  =  13
   0  0  1  1  1  0  0  1  =  57
```

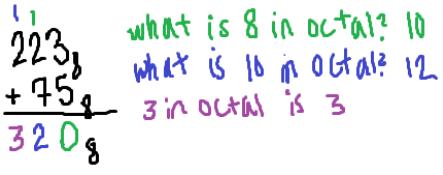
To check, you can always convert all the numbers to base 10, like the above picture.

octal	binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Octal (Base 8)

To the right is a picture of the 8 octal numbers and what they correspond to in binary. An example of a number in base 8 is 102_8 being 66 in base 10, because $(1 \times 8^2) + (0 \times 8^1) + (2 \times 8^0) = 64 + 0 + 2 = 66$.

For addition, add normally, but the resulting number should be the number in octal. For an example, see right. This is the same for subtraction (minus the borrowing).



Why do we bother with base 8?

- **It's shorthand notation for binary**
- Each octal digit represents 3 bits (see octal vs binary table)
- Easier to describe/remember binary data, so group by 3

Example: $000111101110_2 = 000\ 111\ 101\ 110_2 = 0756_8$

It's important to remember that the MIPS word length is 32 bits (not a multiple of 3) as well as other field widths.

hexadecimal	binary	decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

Hexadecimal (Base 16)

Hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. See the table above this text. Addition and subtraction work the same as in base 8.

Why do we bother with base 16?

- **It's also more efficient shorthand notation for binary (we'll use it exclusively)**
- Each hexadecimal digit represents 4 bits, thus used more often
- We will often show hexadecimal values with a subscript of '*x*' instead of '16'
- No subscripts in MARS software; thus, base 16 constants designated w/ prefix '0*x*'

Example: $000111101010_2 = 0001\ 1111\ 1010_2 = 1ea_{16}$

The groups used in both examples are called **fields** (for MIPS architecture, where word size is 32 bits), or words broken into groups of bits.

Two's Complement Representation

How can we represent negative numbers using binary numbers?

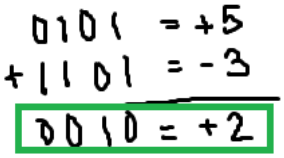
See the table on the right. Note that you cannot go higher or lower than the displayed numbers (out of ways to represent negative numbers in this case). Though it demonstrates for a 4 bit word, the same concepts below apply to a 32 bit word as well:

- Zero is neither positive nor negative
- More negative numbers than positive

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

- Every negative number begins w/ 1 in leftmost bit
- All ones represent -1 (for any word size)

Why are we using this particular representation? **It makes math easy; you can add and subtract in binary without worrying about positivity/negativity.** See right and how 0010 matches with +2.



HOWEVER, note that with this problem, there is an extra carryover that isn't included in the answer, called a **carryout**; if you toss it out, you get the correct result.

Some interesting properties (or consequences):

- Largest number + 1 = smallest number (Ex: *0111 + 0001 = 1000, which is +7 + 1 = -8*).
    - This is called **overflow**, and some computers can detect or ignore it. 4 bits is not enough to represent the result of 7 + 1, in this case. This is also true when you add Integer.MAX + 1; you get Integer.MIN in Java, and similarly for other properties.
    - This applies whether you have a 4 or a 32 bit word.
- Largest number + largest number = -2 (Ex: *0111 + 0111 = 1110, which is +7 + 7 = -2*).
- Smallest number + smallest number = 0 (Ex: *1000 + 1000 = 0000, which is -8 + -8 = 0*).

How do you know whether it is intended to be a two's complement representation?

Given no other information about the binary value, it's impossible to know. You have to be told that it is. The term for non-two's complement representation is **unsigned**.

How to negate a value in two's complement representation? Negation of a number = switch up the signs (eg. Make a positive number negative).

1. Subtract from 0 ($0 - x = -x$) (Ex: *0000 - 0101 = 1011, which is 0 - (+5) = -5*).
2. Change all 0s to 1s and change all 1s to 0s (called the **ones complement**), then add 1 (Ex: *0101; one's complaint is 1010, then 1010 + 0001 = 1011*).
3. Scan the bits from right to left. Copy the 0's to the result. The first 1 encountered should be copied as well. From there, complement all the remaining bits.
(Ex: *0100 (+4) → 00 → 100 → 1100 (-4)*).

Powers of 2

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

These numbers must be memorized (shown right). Let's also call these for the larger powers of 2 (*note that K and M represent values larger than 1000 for 1K*). See the other table below this paragraph.

- $2^{10} = 1024 = 1K$
- $2 \times 2^{10} = 2048 = 2K$
- $1024 \times 2^{10} = 2^{20} = 1024K = 1M$
- $1024 \times 2^{20} = 2^{30} = 1024M = 1G$
- $1024 \times 2^{30} = 2^{40} = 1024G = 1T$

Now we can express any power of 2, up to 2^{49}. If *n* is the exponent of 2,

- The first digit of *n* determines K, M, G, or T (first table).
- The second digit of *n* is a power of 2 from the second table.

n	2^n
3	8
13	8K
17	128K
21	2M
25	32M
32	4G
38	256G
40	1T
49	512T

Some examples are on the right table. With this, calculations can be simplified using exponents: $2^x \times 2^y = 2^{x+y}$, $2^x / 2^y = 2^{x-y}$, and $(2^x)^y = 2^{x*y}$.

Examples:

$$8 \times 64 = 2^3 \times 2^6 = 2^9 = 512$$
$$32 \times 8M = 2^5 \times 2^{23} = 2^{28} = 256M$$
$$256K \times 32M = 2^{18} \times 2^{25} = 2^{43} = 8T$$

$$128 / 16 = 2^7 / 2^4 = 2^3 = 8$$
$$4T / 2K = 2^{42} / 2^{11} = 2^{31} = 2G$$
$$128T / 256G = 2^{47} / 2^{38} = 2^9 = 512$$

$$8^5 = (2^3)^5 = 2^{15} = 32K$$
$$4^8 = (2^2)^8 = 2^{16} = 64K$$
$$64^7 = (2^6)^7 = 2^{42} = 4T$$

How to convert integers to binary by hand:
https://indepth.dev/posts/1019/the-simple-math-behind-decimal-binary-conversion-algorithms

Things like 127's two's complement representation are just its binary number (positive only).