

Chapter 1

Exploring Little Numbers

We manipulate numbers all the time and in this chapter we propose you a little journey into the way integers are mapped to their binary representations. We will open the box and take a language implementor perspective and explore how small integers are represented.

We will start with some simple maths reminders on math that are the basics of our digital world. Then we will have a look at how small integers are encoded. This is commonly forgotten over time and our goal is to aid in refreshing this knowledge.

1.1 Power of 2 and Numbers

Let's start with some simple math. In the digital world, information is encoded as powers of 2. Nothing really new. In Smalltalk raising a number to a power is performed by sending the message `raisedTo:` to a number. Here are some examples. Figure 1.1 shows the powers of 2.

```
2 raisedTo: 0  
  → 1  
2 raisedTo: 2  
  → 4  
2 raisedTo: 8  
  → 256
```

Using a sequence of powers of 2 we can encode numbers. For example, how can we encode the number 13? It cannot be higher than $2^4 = 16$. So it should be $8 + 4 + 1$, $2^3 + 2^2 + 2^0$. Now when we order the powers of two as a sequence as shown in Figure 1.2, we see that 13 is encoded as 1101. So we can encode a number with a sequence of powers of 2. An element

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Figure 1.1: Powers of 2 and their numerical equivalence.

of this sequence is called a bit. 13 is encoded with 4 bits, corresponding to $1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$. This sequence of bits represents a binary notation for numbers. The most significant bit is located on the left of a bit sequence and the least on the right (2^0 is the rightmost bit).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
													1	1	1

Figure 1.2: $13 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$.

Binary notation

Pharo has a syntax for representing numbers in different bases. We write `2r1101` where 2 indicates the base or radix, here 2, and the rest the number expressed in this base. Note that we could also write `2r01101` or `2r0001101` since this notation follows the convention that the least significant bit is the rightmost one.

```
2r1101
  → 13
13 printStringBase: 2
  → '1101'
Integer readFrom: '1101' base: 2
  → 13
```

Note that the last two messages `printStringBase:` and `readFrom:base:` do not handle the internal encoding of negative numbers well as we will see later. `-2 printStringBase: 2` returns `-10` but this is not the internal number representation (known as two's complement). These messages just print/read the number in a given base.

The radix notation can be used to specify numbers in different bases. Obviously 15 written in decimal base (`10r15`) returns 15, while 15 in base 16 returns $16 + 5 = 21$ as illustrated by the following expressions.

```
10r15
  → 15
16r15
  → 21
```

1.2 Bit shifting is multiplying by 2 powers

Since integers are represented as sequences of bits, if we shift all the bits from a given amount we obtain another integer. Shifting bits is equivalent to performing a multiplication/division by two. Figure 1.3 illustrates this point. Smalltalk offers three messages to shift bits: `>> aPositiveInteger`, `<< aPositiveInteger` and `bitShift: anInteger`. `>>` divides the receiver, while `<<` multiply it by a power of two.

The following examples show how to use them.

```
2r000001000
  → 8
2r000001000 >> 1      "we divide by two"
  → 4
(2r000001000 >> 1) printStringBase: 2
  → '100'
2r000001000 << 1      "we multiply by two"
  → 16
```

The message `bitShift:` is equivalent to `>>` and `<<`, but it uses negative and positive integers to indicate the shift direction. A positive argument offers the same behavior as `<<`, multiplying the receiver by a power of 2. A negative is similar to `>>`.

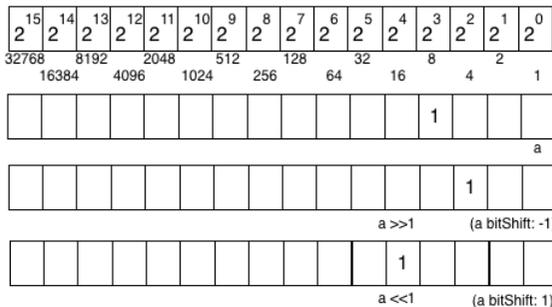


Figure 1.3: Multiplying and dividing by 2.

```

2r000001000
  → 8
2r000001000 bitShift: -1
  → 4
2r000001000 bitShift: 1
  → 16
    
```

Of course, we can shift by more than one bit at a time.

```

2r000001000
  → 8
2r000001000 >> 2           "we divide by four"
  → 2
(2r000001000 >> 2) printStringBase: 2
  → '10'
2r000001000 << 2          "we multiply by four"
  → 32
    
```

The previous examples only show bit shifting numbers with one or two bits, but there is no constraint at this level. The complete sequence of bits can be shifted as shown with 2r000001100 below and Figure 1.4.

```

(2 raisedTo: 8) + (2 raisedTo: 10)
  → 1280
2r010100000000
  → 1280
2r010100000000 >> 8
  → 5
    
```

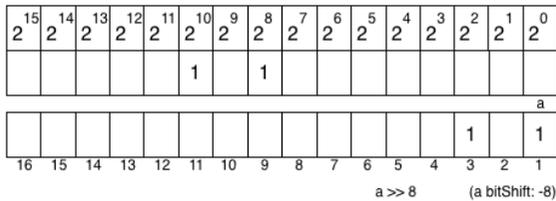


Figure 1.4: We move 8 times to the right. So from 1280, we get 5.

So far, there is nothing really special. Though you should have learned this in a basic math lectures, it is always good to walk on a hill before climbing a mountain.

1.3 Bit manipulation and access

Pharo offers common boolean operations for bit manipulation. Hence you can send the messages `bitAnd:`, `bitOr:`, and `bitXor:` to numbers. They will apply bit by bit the associated Boolean operation.

```
2r000001101 bitAnd: 2r01
  → 1
2r000001100 bitAnd: 2r01
  → 0
2r000001101 bitAnd: 2r1111
  → 1101
```

`bitAnd:` can then be used to select part of a number. For example, `bitAnd: 2r111` selects the three first bits.

```
2r000001101 bitAnd: 2r111
  → 5
2r000001101 bitAnd: 2r0
  → 0
2r0001001101 bitAnd: 2r1111
  → 13 "1101"
2r000001101 bitAnd: 2r111000
  → 8 "1000"
2r000101101 bitAnd: 2r111000
  → 40 "101000"
```

Using `bitAnd:` combined with a `bitShift:` we can select part of a number. Imagine that we encode three numbers on 10 bits: let's say one number encoded on 3 bits (a number between 0 and 7 — noted as XXX in ZZZYYYYXXX), one number encoded on 4 bits (0 to 15 — noted as YYYY in ZZZYYYYXXX), and finally the third one is encoded on 3 bits noted as ZZZ in ZZZYYYYXXX. The following expressions are handy to manipulate these three numbers. Accessing the second number cannot simply be done using only `bitShift:` because we will still have the third number present. We get `(2r1001111001 bitShift: -3)` returns 79, while we would like to get 15. The solution is either to use a `bitAnd:` to clear the third number and to do a `bitShift:` or to do a `bitShift:` and to clear the third number. The `bitAnd:` argument has to be adapted to select the right part of the encoding.

```
(2r1001111001 bitShift: -3)
  → 79
(2r1001111001 bitAnd: 2r0001111000)
  → 120
(2r1001111001 bitAnd: 2r0001111000) bitShift: -3
  → 15
(2r1001111001 bitShift: -3) bitAnd: 2r0001111
  → 15
```

Bit Access. Smalltalk lets you access bit information. The message `bitAt:` returns the value of the bit at a given position. It follows the Pharo convention that collection indexes start at one.

```
2r000001101 bitAt: 1
  → 1

2r000001101 bitAt: 2
  → 0

2r000001101 bitAt: 3
  → 1

2r000001101 bitAt: 4
  → 1

2r000001101 bitAt: 5
  → 0
```

With Pharo you can access the full environment and learn from the system itself. Here is the implementation of the method `bitAt:` on the `Integer` class.

```
Integer>>bitAt: anInteger
  "Answer 1 if the bit at position anInteger is set to 1, 0 otherwise.
  self is considered an infinite sequence of bits, so anInteger can be any strictly positive
  integer.
  Bit at position 1 is the least significant bit.
  Negative numbers are in two-complements.

  This is a naive implementation that can be refined in subclass for speed"

  ^ (self bitShift: 1 - anInteger) bitAnd: 1
```

We shift to the right from an integer minus one (hence `1 - anInteger`) and with a `bitAnd:` we know whether there is a one or zero in the location. Imagine that we have `2r000001101`, when we do `2r000001101 bitAt: 5` we will shift it from 4 and doing a `bitAnd: 1` with select that bits (*i.e.*, returns 1 if it was at 1 and zero otherwise, so its value). Doing a `bitAnd: 1` is equivalent to tell whether there is a 1 in the least significant bit.

Again, nothing really special here, but this was to refresh our memories. Now we will see how numbers are internally encoded in Pharo using 2's complement. We will start by understanding the 10's complement and look at 2's complement.

1.4 Ten's complement of a number

To fully understand 2's complement it is worth to see how it works with decimal numbers. There is no obvious usage for 10's complement, but here the point we want to show is that a complement is the replacement of addition with subtraction (*i.e.*, adding the complement of A to B is equivalent to subtracting A from B).

The 10's complement of a positive decimal integer n is 10 to the power of (k), minus n , where k is the number of digits in the decimal representation of n . $Complement_{10}(n) = 10^k - n$. For example $Complement_{10}(8) = 10^1 - 8$, $Complement_{10}(1968) = 10^4 - 1968 = 8032$. Here $k = 4$ since 1968 is composed of 4 digits.

It can be calculated in the following way:

1. replace each digit d of the number by $9 - d$ and
2. add one to the resulting number.

Examples. The 10's complement of 1968 is $9 - 1, 9 - 9, 9 - 6, 9 - 8 + 1$ *i.e.*, $8031 + 1$ *i.e.*, 8032. Using the rule two we compute $9 - 1, 9 - 9, 9 - 6, 10 - 8$ *i.e.*, 8032. So our 10's complement is 8032. Indeed $1968 + 8032 = 10000 = 10^5$. Therefore it correctly follows the definition above: 8032 is the result of $10000 - 1968$.

The 10's complement of 190680 is then $9 - 1, 9 - 9, 9 - 0, 9 - 6, 9 - 8, 9 - 0 + 1$ *i.e.*, $809319 + 1$ *i.e.*, 809320. Let's verify: $190680 + 809320 = 1000000$.

To compute the 10's complement of a number, it is enough to perform $9-d$ for each digit and add one to the result.

Some books propose another equivalent way of computing the 10's complement: (1) All the zeros at the right-hand end of the number remain as zeros, (2) The rightmost non-zero digit d of the number is replaced by $10 - d$, and (3) Each other digit d is replaced by $9 - d$.

Computer scientists will probably prefer the first way since it is more regular and adding 1 is cheaper than making more tests.

Subtraction at work

The key point of complement techniques is to convert subtractions into additions. Let us check that.

Examples. Suppose we want to perform the subtraction $8 - 3 = 5$. We will transform such a subtraction into an addition using the 10's complement.

The 10's complement of 3 is $9 - 3 + 1 = 7$. We add 7 to 8 and get 15. We drop the carry that we obtained from the addition and we obtain 5. In fact, the idea is that $8 - 3 = 8 - (10 - 7) = 8 + 7 - 10 = 15 - 10 = 5$.

Now let's compute $98 - 60$. The 10's complement of 60 is $9 - 6, 9 - 0$ *i.e.*, $39 + 1$ *i.e.*, 40. $98 - 60 = 98 + 40 - 100 = 138 - 100 = 38$. Therefore we could say that $98 - 60 = 98 + 40 = (1)38$ and we drop the carry.

Now to perform $-98 + 60$ we compute the 10's complement of 98, then the sum, then the 10's complement of the sum and negate the result. *i.e.*, $-98 + 60$ becomes $2 + 60 = 62$. 62 10's complement is 38. We negate it and therefore $-98 + 60 = -38$.

Another look at it. Replacing a number by its 10's complement is based on $a - b = a - (10 - c)$ where $10 = b + c$. Imagine that we want to perform the following expression $190680 - 109237$ which equals to 81443. The 10's complement takes advantage of the fact that 109237 is also $999999 - 890762$ or $1000000 - 890763$ where 890763 is the 10's complement of 109237.

$$\begin{aligned} 109237 &= 999999 - 890762 \\ 109237 &= 999999 - 890762 (+1 - 1) \\ 109237 &= 1000000 - 890762 - 1 \end{aligned}$$

Now the first subtraction is expressed as:

$$\begin{aligned} 190680 - 109237 & \\ &= 190680 - (1000000 - 890762 - 1) \\ &= 190680 - 1000000 + 890762 + 1 \\ &= 190680 + 890762 + 1 - 1000000 \\ &= 1081443 - 1000000 \\ &= 81443 \end{aligned}$$

Before getting into 2's complement we will look at negative number representation.

1.5 Negative numbers

To know the value of a positive number is simple: we just add all the powers of 2 given by the binary representation as explained at the beginning of this chapter. Getting the value of a negative number is quite simple: we do the same except that we count the *sign bit* as negative and all the other ones as positive. The sign bit is the most significant bit *i.e.*, the bit that represents the largest value (see Figure 1.5). For example, on 8 bit representation it will be the one associated with the weight 2^7 .

Let us illustrate that: -2 is represented on 8 bit encoding as: 1111 1110.

most significant bit								
0	1	1	1	1	1	1	1	127
0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	-1
1	1	1	1	1	1	1	0	-2
-2 ⁷	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	0	-2
1	0	0	0	0	0	0	1	-127
1	0	0	0	0	0	0	0	-128

Figure 1.5: Negative numbers on 8 bits.

To get the value out of the bit representation, we simply add: $-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 0 * 2^0$, *i.e.*, $-128 + 64 + 32 + 16 + 8 + 4 + 2$ and we get -2 .

-69 is represented on 8 bit encoding as: 1011 1011. To get the value out of the bit representation is simple. We add: $-2^7 + 0 * 2^6 + 2^5 + 2^4 + 2^3 + 0 * 2^2 + 2^1 + 2^0$, *i.e.*, $-128 + 32 + 16 + 8 + 2 + 1$ and we get -69 .

Following the same principle, check that the value of -1 is the one described in Figure 1.5.

Let us count a bit: on an 8 bit representation we can then encode 0 to 255 positive integers or -128 to $64 + 32 + 16 + 8 + 4 + 2 + 1$ 127. In fact we can encode from $-1 * 2^7$ to $2^7 - 1$. More generally on N bits we can encode $-1 * 2^{N-1}$ to $2^{N-1} - 1$ integer values.

1.6 Two's complement of a number

Now we have all the pieces of the puzzle: we know how we can encode positive and negative numbers, we know how to use the complement to turn a subtraction into an addition. Let us see how the 2's complement is used to negate numbers and perform subtraction.

The 2's complement is a common method to represent signed integers. The advantages are that addition and subtraction are implemented without having to check the sign of the operands and 2's complement has only one representation for zero (avoiding negative zero). Adding numbers of different sign encoded using 2's complement does not require any special processing: the sign of the result is determined automatically. The 2's complement of a positive number represents the negative form of that number.

Note that the 2's complement of a negative number is the corresponding positive value as shown by the following expressions: -2 two complement is 2. First we compute the direct inversion (bitwise NOT) and add one.

```
-2 bitString
  → '11111111111111111111111111111110'

-2 bitInvert bitString
  → '00000000000000000000000000000001'

(-2 bitInvert + 1) bitString
  → '00000000000000000000000000000010'

2 bitString
  → '00000000000000000000000000000010'
```

As you see negating a number and computing its two's complement gives the same binary representation.

```
(2r101 bitInvert + 1) bitString
  returns '111111111111111111111111111011'
2r101 negated bitString
  returns '111111111111111111111111111011'
```

There is one exception. On a given number of bits, let's say 8 bits as in Figure 1.6, we compute the negative of a number by computing its two's complement (flipping all the bits and adding 1), except for the most negative number. On an 8 bits representation, the most negative number is -128 (1000 0000), inverting it is (0111 1111), and adding one results in itself (1000 0000). We cannot encode 128 on 8 bits signed convention. Here, the carry is "eaten" by the sign bit.

Subtracting. To subtract a number from another one, we just add the second number's 2's complement to the first one.

When we want to compute $110110 - 101$, we compute the 2's complement of 101 and add it. We add 110110 and 111011, and get 110001. This is correct: $54 - 5 = 49$.

```
110110 - 101
  110110
+ 111011
-----
 110001
```

Let us test this in Pharo.

```
(2r110110 - 2r101) bitString
```



```
(SmallInteger maxVal + 1) class
  → LargePositiveInteger
"But the smallest negative is one less than the negated largest positive small integer"
(SmallInteger maxVal negated - 1)
  → -1073741824
(SmallInteger maxVal negated - 1) class
  → SmallInteger
```

Understanding some methods. If you want to know the number of bits used to represent a SmallInteger, just evaluate:

```
SmallInteger maxVal highBit + 1
returns 31
```

SmallInteger maxVal highBit tells the highest bit which can be used to represent a positive SmallInteger, and + 1 accounts for the sign bit of the SmallInteger (0 for positive, 1 for negative).

Let us explore a bit.

```
2 raisedTo: 29
  → 536870912

536870912 class
  → SmallInteger

2 raisedTo: 30
  → 1073741824

1073741824 class
  → LargePositiveInteger

(1073741824 - 1) class
  → SmallInteger

-1073741824 class
  → SmallInteger

2 class maxVal
returns 1073741823

-1 * (2 raisedTo: (31-1))
  → -1073741824

(2 raisedTo: 30) - 1
  → 1073741823

(2 raisedTo: 30) - 1 = SmallInteger maxVal
```

```
→ true
```

1.8 Hexadecimal

We cannot finish this chapter without talking about hexadecimal. Pharo uses the same syntax for hexadecimal than for binary. `16rF` indicates that `F` is encoded in 16 base.

We can get the hexadecimal equivalent of a number using the message `hex`. Using the message `printStringHex` we get the number printed in hexadecimal without the radix notation.

```
15 hex
  → '16rF'

15 printStringHex
  → 'F'

16rF printIt
  → 15
```

The following snippet lists some equivalence between a number and its hexadecimal representation.

```
{(1->'16r1'). (2->'16r2'). (3->'16r3'). (4->'16r4'). (5->'16r5'). (6->'16r6'). (7->'16r7').
 (8->'16r8'). (9->'16r9'). (10->'16rA'). (11->'16rB'). (12->'16rC'). (13->'16rD'). (14
->'16rE'). (15->'16rF')}
```

When doing bit manipulation it is often shorter to use an hexadecimal notation over a binary one. Even if for `bitAnd`: the binary notation may be more readable

```
16rF printStringBase: 2
  → '1111'
2r00101001101 bitAnd: 2r1111
  → 2r1101
2r00101001101 bitAnd: 16rF
  → 2r1101
```

1.9 Chapter summary

Pharo uses 2's complement encoding for its internal small integer representation and supports bit manipulation of their internal representation. This

is useful when we want to speed up algorithms using simple encoding. We have reviewed the following points:

- Numerical values use complement to encode their negative value.
- Shifting a bit to the left multiple is equivalent to multiply it by 2, modulo the maximum value of its encoding size.
- On the opposite, shifting a bit to the right divides it by 2.
- Bits operations can be performed on any numerical values.
- Complement are useful to turn an addition into a subtraction, thus simplifying the operation.
- SmallInteger are coded on 31 bits on Pharo.

Note that Pharo supports large numbers whose limit in size is mainly the memory you have at your disposal.