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 happily explore how small integers are represented.

We will start with some simple reminders on math that are the basics of our digital world. Then we will have a look at how integers and in particular small integers are encoded. This is a typical simple knowledge that people can forget over time and our goal is simply to refresh it.

1.1 Power of 2 and Numbers

Let's start with some simple maths. In 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 power of 2 we can encode numbers. For example, how can we encode the number 13? It cannot be higher than 2^4 because 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 of this sequence is called a bit. 13 is encoded with 4 bits, corresponding to

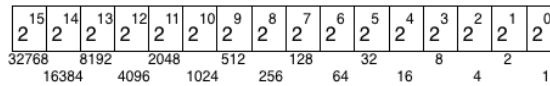

Figure 1.1: Powers of 2 and their numerical equivalence.

$1*2^3+1*2^2+0*2^1+1*2^0$. 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).

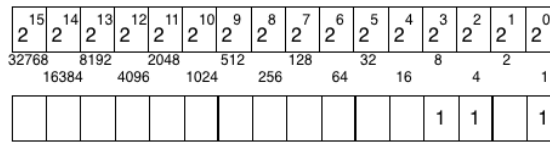

Figure 1.2: 13 = $1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$.

Binary notation

Smalltalk 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 the leading zeroes do not change the number. This notation follows the convention that the least significant bit is the rigthmost 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 well the internal encoding of negative numbers 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 use 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 bit is equivalent to perform 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 than <<, multiplying the receiver by a power of 2. A negative is similar to >>.

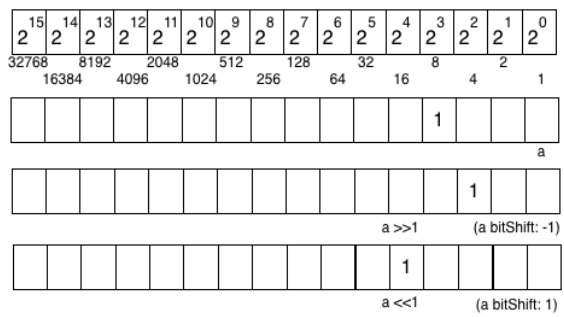

Figure 1.3: Multiplying and dividing by 2.

```
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
      ⟶    2r0101
   returns 5
```

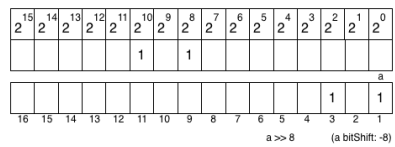

Figure 1.4: We move 8 times to the right. So from 1280 we get 5.

So far nothing really special. You should have learned that in any basic math lecture, but this 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 them.

Accessing the second number cannot simply be done with a 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 Smalltalk convention that collections' indexes start with one.

```
2r000001101 bitAt: 1
      ⟶    1

2r000001101 bitAt: 2
      ⟶    0

2r000001101 bitAt: 3
      ⟶    1

2r000001101 bitAt: 4
      ⟶    1

2r000001101 bitAt: 5
      ⟶    0
```

This is interesting to 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 but this was to refresh our memories. Now we will see how numbers are internally encoded in Pharo using two's complement.

Stéf ▶*up to here*◀

1.4 Ten's complement of a number

To fully understand 2's complement it is interesting 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 + 1, minus n, where k is the number of digits in the textual decimal representation of n. $Complement_{10}(n) = 10^{k+1} - n$. For example $Complement_{10}(8) = 10^1 - 8$, $Complement_{10}(1968) = 10^5 - 1968 = 8032$

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.

This two steps rule is equivalent to the following one which looks more complex. Computer scientists will probably prefer the first way since it is more regular and adding 1 is cheaper than making more tests.

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 $10 - d$.
3. Each other digit d is replaced by $9 - d$.

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: $190680 + 809320 = 1000000$.

So 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.

Subtraction at work

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

Examples. Suppose we want to turn the subtraction $8 - 3 = 5$ into an addition. The 10's complement of 3 is $9 - 3 + 1 = 7$. We add 7 to 8 and get 15. We drop the carry we obtained when from the addition. So we obtain 5. (Note that when using binary representation as we will show later, the carry results in overflow because there will be not enough bits to represent it. Therefore implementors should use tricks to determine if the subtraction is correct - usually the trick is to perform a bitXor on the last two digits).

Now $98 - 60$. The 10's complement of 60 is $9 - 6$, $9 - 0$ *i.e.*, $39 + 1$ *i.e.*, 40. $98-60 = 98+40-100 = 138-100 = 38$. Note that the 100 subtraction is a way to show that we drop the carry. We could have written $98 - 60 = 98 + 40 = 138 = 38$ but this is an incorrect use of the $=$ mathematical operatordo you want to say that 138 is equal to 38 modulo 100?.

Now performing $60 - 80$ works too. 80 10's complement is $9 - 8$, $9 - 0$, so 19 + 1, so 20. $60 - 80 = -20$. Again $60 - 80$ just requires to compute the complement and perform one addition.

Another look at it. Imagine that we want to perform the following expression $190680 - 109237$ which is equals to 81443. The 10's complement takes advantage of the fact that 109237 is also $999999 - 890762$.

```
109237 = 999999 − 890762
109237 = 999999 − 890762 (+ 1 − 1)
109237 = 1000000 − 890762 − 1
```

Now the first subtraction is expressed as:

```
190680 − 109237
= 190680 − (1000000 − 890762 − 1)
= 190680 − 1000000 + 890762 + 1
= 190680  + 890762 + 1 − 1000000
= 1081443 − 1000000
= 81443
```

1.5 Two's complement of a number

The two'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 two's complement has only one representation for zero (avoiding negative zero). Adding numbers of different sign encoded using two's complement does not require any special processing: the sign of the result is determined automatically.

What the 10's complement shows us that it is achieved by taking the difference of each digit with the largest number available in the base system,

9 in decimal base and adding one. Now in the case of binary, the base is one. Due to the fact that $1 - 0 = 1$ and $1 - 1 = 0$, computing the *two's complement of number is exactly the same as flipping 1's to 0's and vice versa and adding 1.*

Try the following expressions in Pharo and experiment with them. We compute the direct inversion (bitwise NOT) and add one. Remember a bitwise NOT is turning any 1s into 0s and conversely.

```
2 bitString
        '00000000000000000000000000000010'
2 bitInvert bitString
        '11111111111111111111111111111101'
(2 bitInvert + 1) bitString
        '11111111111111111111111111111110'
-2 bitString
        '11111111111111111111111111111110'
```

If you check the length of the printed results, you will notice that they are 31 bits long instead of 32. This is because in Pharo and most Smalltalk implementations, small integers are specially encoded and this encoding requires one bit.

Note that the two'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'
```

Negative number value. To know the value of a positive number it is simple: we just sum all the powers of 2 given by the binary representation as explained at the beginning of this Chapter. Now getting the value of a negative number (in two's complement) is quite simple: we do the same except that we count the *sign bit* as negative, all the other ones as positive. The sign bit is the most significant bit *i.e.*, the bit that represents the largest value. For example, on 8 bit representation it will be the one associated with the weight 2^7.

Let us illustrate that: -69 is represented on 8 a bit encoding as: 1011 1011.

So get the value out of the bit representation is simple we sum: $-2^7 + 0 * 2^6 + 2^5 + 2^4 + 2^3 + 0 * 2^2 + 2^1 + 2^0$, *i.e.*, $-128 + 32 + 16 + 8 + 2 + 1$ and we get -69.

Subtracting. To subtract a number to another one, we will add the second number's two complement to the first one.

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

```
110110 − 101
  110110
+ 111011
−−−−−−−−−−
  110001
```

Let us test in Pharo

```
(2r110110 − 2r101) bitString
   returns '00000000000000000000000000110001'
(2r110110 bitString)
   returns '00000000000000000000000000110110'
2r101 bitString
   returns '00000000000000000000000000000101'

(2r101 bitInvert + 1) bitString
   returns '11111111111111111111111111111011'
2r101 negated bitString
   returns '11111111111111111111111111111011'
```

Posing the addition using only 8 bits we see the following:

```
carry   1111111
       00110110
+      11111011
   −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
       00110001
```

Note that the overflowing carry is dropped.

Representing negative numbers. What is nice with the 2's complement is that we can use it to represent a negative number. Indeed, we can negate a number by computing its two complement. The two's complement of a positive number represents the negative form of that number.

Let's look at 2. 2 is encoded on 8 bits as 000000010 and -2 as 11111110 as shown in Figure 1.5. 000000010 flipped is: 11111101 and we add one: so we get 11111110.

The difference between -2 (11111110) and 126 (01111110) is given by the most significant bit which conveys the sign of the integer.

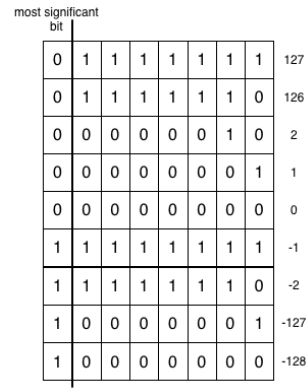

Figure 1.5: Overview of two's complement on 8 bits.

There is one exception. One a given number of bits, let's say 8 bits as in Figure 1.5, we obtained the negative of a number but computing its two's complement (flipping all the bits and adding 1), except for the most negative number. On a 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.

In Pharo. Let's try with Pharo to check a bit our understanding.

```
SmallInteger>>bitString
    "Returns the bit representation of the receiver."
    ↑ String streamContents: [:s |
        30 to: 1 by: −1 do: [:i | s nextPut: ((self bitAt: i) + 48) asCharacter ] .
        s contents ]
```

In Pharo, you can experiment to confirm the fact that the two's complement is the negative version of a number.

```
2 bitString
    returns  '00000000000000000000000000000010'

'00000000000000000000000000000010'
    collect: [:each | $0 = each ifTrue: [$1] ifFalse: [$0]]
    returns '11111111111111111111111111111101'

−2 bitString
```

```
   returns '11111111111111111111111111111110'
```

Creating a two complement version of a number equals negating the number bits and adding one Andrew ►*I have the impression we are repeating ourselves*◄.

```
3 bitString
   returns '00000000000000000000000000000011'
3 bitInvert bitString
   returns '11111111111111111111111111111100'
(3 bitInvert + 1) bitString
   returns '11111111111111111111111111111101'
−3 bitString
   returns '11111111111111111111111111111101'
```

Now the case where the result is a negative number is also well handled. For example, if we want to compute $-15 + 5$, we should get 10 and this is what we get.

```
−15 bitString
   returns '11111111111111111111111111110001'

5 bitString
   returns '00000000000000000000000000000101'

−10 bitString
   returns '11111111111111111111111111110110'
```

Maximum value encoding. As we will show in a subsequent section, Pharo's small integers are encoded on 31 bits (because their internal representation requires one bit) and the smallest (small integer) negative integer is SmallInteger maxVal negated − 1. Here we see the exception of the most negative integer.

```
"we negate the maximum number encoded on a small integer"
SmallInteger maxVal negated
      returns −1073741823
"we still obtain a small integer"
SmallInteger maxVal negated class
      returns SmallInteger
"adding one to the maximum number encoded on a small integer gets a large positive
      integer"
(SmallInteger maxVal + 1) class
      returns LargePositiveInteger
"But the smallest negative is one less than the negated largest positive small integer"
(SmallInteger maxVal negated − 1)
      returns −1073741824
```

```
(SmallInteger maxVal negated − 1) class
    returns SmallInteger
```

Understanding some methods. Now you should be able to understand the implementation of SmallInteger>>bitInvert method.

```
SmallInteger>>bitInvert
  "Answer an Integer whose bits are the logical negation of the receiver's bits.
  Numbers are interpreted as having 2's−complement representation."

  ↑ −1 − self.
```

```
2 bitString
  returns '00000000000000000000000000000010'
2 bitInvert.
  returns '11111111111111111111111111111101'

−1 bitString
  returns '11111111111111111111111111111111'
2 negated (two complement)
  returns '11111111111111111111111111111110'
```

1.6 SmallIntegers in Pharo

Smalltalk small integers use a two's complement arithmetic on 31 bits. A N-bit two's-complement numeral system can represent every integer in the range $-1 * 2^{N-1}$ *to* $2^{N-1} - 1$. So for 31 bits Smalltalk systems small integers values are the range -1 073 741 824 to 1 073 741 823. Remember in Smalltalk integers are special objects and this marking requires one bit, therefore on 32 bits we have 31 bits for small signed integers. Of course since we also have automatic coercion this is not really a concern for the end programmer. Here we take a language implementation perspective. Let's check that a bit (this is the occasion to say it).

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
   returns 536870912

536870912 class
   returns SmallInteger

2 raisedTo: 30
   returns 1073741824

1073741824 class
   returns LargePositiveInteger

(1073741824 − 1) class
   returns SmallInteger

−1073741824 class
   returns SmallInteger

2 class maxVal
   returns 1073741823

−1 * (2 raisedTo: (31−1))
   returns −1073741824

(2 raisedTo: 30) − 1
   returns 1073741823

(2 raisedTo: 30) − 1 = SmallInteger maxVal
   returns true
```

1.7 Hexadecimal

We cannot finish this Chapter without talking about hexadecimal. In Smalltalk, the same syntax than for binary is used for hexadecimal. `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
   returns '16rF'

15 printStringHex
   returns 'F'
```

```
16rF printIt
   returns 15
```

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

```
{(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
   returns '1111'
2r00101001101 bitAnd: 2r1111
   returns 2r1101
2r00101001101 bitAnd: 16rF
   returns 2r1101
```

Information subpart extraction

Often numbers are used to encode in a compact manner multiple information: we could imagine to use 1 bit for gender, 6 bits for age.... The resulting number would be meaningless but information extracted from it is meaningful. Now we are armed to select subpart of number to get such information.

```
2r1100010100000000
   returns 50432
```

Imagine that only 4 bits interest us starting at the 8th bit. To get the encoded value, we shift out the 8 first ones and clear all the others after the next 4: so we bit shift and bit and as follows:

```
(2r1100010100000000 >> 8)
   returns 2r11000101
   returns 197

(2r1100010100000000 >> 8) bitAnd: 16rF
   returns 2r0101
   returns 5

2r11000101 bitAnd: 2r1111
   returns 2r0101
   returns 5
```

1.8 Conclusion

Smalltalk uses two's complement encoding for its internal number 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:

Alex ►*I inserted the following*◄

- 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.