I.1 Integers

In this chapter we discuss the following:

1. Binary representation: Any real number can be represented in binary, that is,

by an infinite sequence of 0s and 1s (bits). We review binary representation. 2. Unsigned integers: We discuss how computers represent non-negative integers using only p-bits, via modular arithmetic. 3. Signed integers: we discuss how negative integers are handled using the Two's-complement format. 4. As an advanced (non-examinable) topic we discuss `BigInt`, which uses variable bit length storage.

Before we begin its important to have a basic model of how a computer works. Our simplified model of a computer will consist of a Central Processing Unit (CPU)—the brains of the computer—and Memory—where data is stored. Inside the CPU there are registers, where data is temporarily stored after being loaded from memory, manipulated by the CPU, then stored back to memory.

Memory is a sequence of bits: `1` s and `0` s, essentially "on/off" switches. These are grouped into bytes, which consist of 8 bits. Each byte has a memory address: a unique number specifying its location in memory. The number of possible addresses is limited by the processor: if a computer has a a p-bit CPU then each address is represented by p bits, for a total of 2^p addresses (on a modern 64-bit CPU this is $2^{64} \approx 1.8 \times 10^{19}$ bytes). Further, each register consists of exactly p-bits.

A CPU has the following possible operations:

1. load data from memory addresses (up to p-bits) to a register
2. store data from a register to memory addresses (up to p-bits)
3. Apply some basic functions ("+", "-", etc.) to the bits in one or two registers

and write the result to a register.

Mathematically, the important point is CPUs only act on 2^p possible sequences of bits at a time. That is, essentially all functions f implemented on a CPU are either of the form $f : \mathbb{Z}_{2^p} \to \mathbb{Z}_{2^p}$ or $f : \mathbb{Z}_{2^p} \times \mathbb{Z}_{2^p} \to \mathbb{Z}_{2^p}$, where we use the following notation:

Definition 1 ($\mathbb{Z}_m$, signed integers) Denote the

$$\mathbb{Z}_m := \{0, 1, \ldots, m-1\}$$

The limitations this imposes on representing integers is substantial. If we have an implementation of $+$, which we shall denote $\oplus_m$, how can we possibly represent $m+1$ in this implementation when the result is above the largest possible integer?

The solution that is used is straightforward: the CPU uses modular arithmetic. E.g., we have

$$(m-1) \oplus_m 1 = m \pmod m = 0.$$

In this chapter we discuss the implications of this approach and how it works with negative numbers.

We will use Julia in these notes to explore what is happening as a computer does integer arithmetic. We load an external package which implements functions `printbits` (and `printlnbits`) to print the bits (and with a newline) of numbers in colour:

In [1]:
```julia
using ColorBitstring
```

1. Binary representation

Any integer can be presented in binary format, that is, a sequence of `0` s and `1` s.

Definition 2 (binary format) For $B_0, \ldots, B_p \in \{0, 1\}$ denote an integer in *binary format* by:

$$\pm(B_p \ldots B_1 B_0)_2 := \pm \sum_{k=0}^{p} B_k 2^k$$

Example 1 (integers in binary) A simple integer example is $5 = 2^2 + 2^0 = (101)_2$. On the other hand, we write $-5 = -(101)_2$. Another example is $258 = 2^8 + 2 = (1000000010)_2$.

2. Unsigned Integers

Computers represent integers by a finite number of p bits, with 2^p possible combinations of 0s and 1s. For *unsigned integers* (non-negative integers) these bits dictate the first p binary digits: $(B_{p-1} \ldots B_1 B_0)_2$.

Integers on a computer follow modular arithmetic: Integers represented with p-bits on a computer actually represent elements of $\mathbb{Z}_{2^p}$ and integer arithmetic on a computer is equivalent to arithmetic modulo 2^p. We denote modular arithmetic with $m = 2^p$ as follows:

$$\begin{aligned} x \oplus_m y &:= (x + y) (\text{mod } m) \\ x \ominus_m y &:= (x - y) (\text{mod } m) \\ x \otimes_m y &:= (x * y) (\text{mod } m) \end{aligned}$$

When m is implied by context we just write $\oplus, \ominus, \otimes$.

Example 2 (arithmetic with 8-bit unsigned integers) If arithmetic lies between 0 and $m = 2^8 = 256$ works as expected. For example,

$$
\begin{aligned}
17 \oplus_{256} 3 = 20 (\text{mod } 256) = 20 \\
17 \ominus_{256} 3 = 14 (\text{mod } 256) = 14
\end{aligned}
$$

This can be seen in Julia:

In [2]:
```
x = UInt8(17)  # An 8-bit representation of the number 255, i.e. with bits 0
y = UInt8(3)   # An 8-bit representation of the number   1, i.e. with bits 0
printbits(x); println(" + "); printbits(y); println(" = ")
printlnbits(x + y) # + is automatically modular arithmetic
printbits(x); println(" - "); printbits(y); println(" = ")
printbits(x - y) # - is automatically modular arithmetic
```

```
00010001 +
00000011 =
00010100
00010001 -
00000011 =
00001110
```

Example 3 (overflow with 8-bit unsigned integers) If we go beyond the range the result "wraps around". For example, with integers we have

$$255 + 1 = (11111111)_2 + (00000001)_2 = (100000000)_2 = 256$$

However, the result is impossible to store in just 8-bits! So as mentioned instead it treats the integers as elements of $\mathbb{Z}_{256}$:

$$255 \oplus_{256} 1 = 255 + 1 \ (\text{mod } 256) = (00000000)_2 \ (\text{mod } 256) = 0 \ (\text{mod } 256)$$

We can see this in code:

In [3]:
```
x = UInt8(255) # An 8-bit representation of the number 255, i.e. with bits 1
y = UInt8(1)   # An 8-bit representation of the number   1, i.e. with bits 0
printbits(x); println(" + "); printbits(y); println(" = ")
printbits(x + y) # + is automatically modular arithmetic
```

```
11111111 +
00000001 =
00000000
```

On the other hand, if we go below 0 we wrap around from above:

$$3 \ominus_{256} 5 = -2 (\text{mod } 256) = 254 = (11111110)_2$$

In [4]:
```
x = UInt8(3) # An 8-bit representation of the number   3, i.e. with bits 000
y = UInt8(5) # An 8-bit representation of the number   5, i.e. with bits 000
printbits(x); println(" - "); printbits(y); println(" = ")
printbits(x - y) # + is automatically modular arithmetic
```

```
00000011 -
00000101 =
11111110
```

Example 4 (multiplication of 8-bit unsigned integers) Multiplication works similarly: for example,

$$254 \otimes_{256} 2 = 254 * 2 \pmod{256} = 252 \pmod{256} = (11111100)_2 \pmod{256}$$

We can see this behaviour in code by printing the bits:

In [5]:
```julia
x = UInt8(254) # An 8-bit representation of the number 254, i.e. with bits 1
y = UInt8(2)   # An 8-bit representation of the number   2, i.e. with bits 0
printbits(x); println(" * "); printbits(y); println(" = ")
printbits(x * y)
```

```
11111110 *
00000010 =
11111100
```

Hexadecimal and binary format

In Julia unsigned integers are displayed in hexadecimal form: that is, in base-16. Since there are only 10 standard digits (`0-9`) it uses 6 letters (`a-f`) to represent 11–16. For example,

In [6]:
```julia
UInt8(250)
```

Out[6]:
```
0xfa
```

because `f` corresponds to 15 and `a` corresponds to 10, and we have

$$15 * 16 + 10 = 250.$$

The reason for this is that each hex-digit encodes 4 bits (since 4 bits have $2^4 = 16$ possible values) and hence two hex-digits are encode 1 byte, and thus the digits correspond exactly with how memory is divided into addresses. We can create unsigned integers either by specifying their hex format:

In [7]:
```julia
0xfa
```

Out[7]:
```
0xfa
```

Alternatively, we can specify their digits. For example, we know $(f)_{16} = 15 = (1111)_2$ and $(a)_{16} = 10 = (1010)_2$ and hence $250 = (fa)_{16} = (11111010)_2$ can be written as

In [8]:
```julia
0b11111010
```

Out[8]:
```
0xfa
```

3. Signed integer

Signed integers use the Two's complemement convention. The convention is if the first bit is 1 then the number is negative: the number $2^p - y$ is interpreted as $-y$. Thus for $p = 8$ we are interpreting 2^7 through $2^8 - 1$ as negative numbers. More precisely:

Definition 3 ($\mathbb{Z}_{2^p}^s$, unsigned integers)

$$\mathbb{Z}_{2^p}^s := \{-2^{p-1}, \ldots, -1, 0, 1, \ldots, 2^{p-1} - 1\}$$

Definition 4 (Shifted mod) Define for $y = x \pmod{2^p}$

$$x \ (\mathrm{mod}^{\mathrm{s}}\ 2^p) := \begin{cases} y & 0 \le y \le 2^{p-1} - 1 \\ y - 2^p & 2^{p-1} \le y \le 2^p - 1 \end{cases}$$

Note that if $R_p(x) = x(\mathrm{mod}^{\mathrm{s}}\ 2^p)$ then it can be viewed as a map $R_p : \mathbb{Z} \to \mathbb{Z}_{2^p}^s$ or a one-to-one map $R_p : \mathbb{Z}_{2^p} \to \mathbb{Z}_{2^p}^s$ whose inverse is $R_p^{-1}(x) = x \mod 2^p$.

Example 5 (converting bits to signed integers) What 8-bit integer has the bits `01001001`? Because the first bit is 0 we know the result is positive. Adding the corresponding decimal places we get:

In [9]:
```
2^0 + 2^3 + 2^6
```

Out[9]: 73

What 8-bit (signed) integer has the bits `11001001`? Because the first bit is `1` we know it's a negative number, hence we need to sum the bits but then subtract `2^p`:

In [10]:
```
2^0 + 2^3 + 2^6 + 2^7 - 2^8
```

Out[10]: -55

We can check the results using `printbits`:

In [11]:
```
printlnbits(Int8(73)) # Int8 is an 8-bit representation of the signed intege
printbits(-Int8(55))
```

```
01001001
11001001
```

Arithmetic works precisely the same for signed and unsigned integers, e.g. we have

$$x \oplus_{2^p}^s y := x + y(\mathrm{mod}^{\mathrm{s}}\ 2^p)$$

Example 6 (addition of 8-bit integers) Consider `(-1) + 1` in 8-bit arithmetic. The number -1 has the same bits as $2^8 - 1 = 255$. Thus this is equivalent to the previous question and we get the correct result of `0`. In other words:

$$-1 \oplus_{256} 1 = -1 + 1 \pmod{2^p} = 2^p - 1 + 1 \pmod{2^p} = 2^p \pmod{2^p} = 0 \pmod{2^p}$$

Example 7 (multiplication of 8-bit integers) Consider `(-2) * 2`. -2 has the same bits as $2^{256} - 2 = 254$ and -4 has the same bits as $2^{256} - 4 = 252$, and hence from the previous example we get the correct result of `-4`. In other words:

$$(-2) \otimes_{2^p}^{s} 2 = (-2) * 2 \pmod{^s 2^p} = (2^p - 2) * 2 \pmod{^s 2^p} = 2^{p+1} - 4 \pmod{^s 2^p} = -$$

Example 8 (overflow) We can find the largest and smallest instances of a type using `typemax` and `typemin`:

In [12]:
```
printlnbits(typemax(Int8)) # 2^7-1 = 127
printbits(typemin(Int8)) # -2^7 = -128
```

```
01111111
10000000
```

As explained, due to modular arithmetic, when we add `1` to the largest 8-bit integer we get the smallest:

In [13]:
```
typemax(Int8) + Int8(1) # returns typemin(Int8)
```

Out[13]:
```
-128
```

This behaviour is often not desired and is known as *overflow*, and one must be wary of using integers close to their largest value.

Division

In addition to `+`, `-`, and `*` we have integer division `÷`, which rounds towards zero:

In [14]:
```
5 ÷ 2 # equivalent to div(5,2)
```

Out[14]:
```
2
```

Standard division `/` (or `\` for division on the right) creates a floating-point number, which will be discussed in the next chapter:

In [15]:
```
5 / 2 # alternatively 2 \ 5
```

Out[15]:
```
2.5
```

We can also create rational numbers using `//`:

In [16]:
```
(1//2) + (3//4)
```

Out[16]:
```
5//4
```

Rational arithmetic often leads to overflow so it is often best to combine `big` with rationals:

In [17]:
```
big(102324)//132413023 + 23434545//4243061 + 23434545//42430534435
```

Out[17]:

```
2633903783500764847754154O//476780487870754436459646l
```

4. Variable bit representation (non-examinable)

An alternative representation for integers uses a variable number of bits, with the advantage of avoiding overflow but with the disadvantage of a substantial speed penalty. In Julia these are `BigInt` s, which we can create by calling `big` on an integer:

In [18]:

```julia
x = typemax(Int64) + big(1) # Too big to be an `Int64`
```

Out[18]:

```
9223372036854775808
```

Note in this case addition automatically promotes an `Int64` to a `BigInt`. We can create very large numbers using `BigInt`:

In [19]:

```julia
x^100
```

Out[19]:

```
3082994025277634745700106821545665721371798533305697458855342277921093731 98
4476404705966539412410898240561729912372038501228893141921080152404642393 77
6599077294434061519905424124601394226943601430916434383714716724720227331 59
6950613701661034548948388721097667275438763758128508403297199458260277707 30
1202460980093818414167080563342761482395862435185093942443540722363151770 02
2221783243959592531336062998494209914752408019060720805124534382646051093 61
3814848646062038662423487504326044361203708430489305864234333801401547140 02
3376295718383390360728662900230671437151716615826286842267917560749586018 16
5739492101920429719261285640125596833063891562865262157026023955919873792 84
6823095854484520920509345944712871675691790827690907778485058829248588945 68
1685288179787963931181062068092463984296225973082494056307958089189726701 67
8735576365394146232076917088075949053636690459581128773097212746967276496 49
6010810878000638239143750075543163240049874489986642327436441234458040254 48
0825038220479904594615300602390556385799245276805580024937804723029319565 94
2013515817048714543455250235208789745701165279569026248145395218985062991 83
1707830217974393158466067785199581037714968820628241051867119832966361530 04
7910339065726550260741036716100932205969655083257714244071120221654679340 46
1084001560321676025443801248355439305974923873624147980728110581452806109 01
1739005060060604228087667499288851218705078807364237925455813890575257569 98
1450090997117697469299234094394984840574025401463942099019413361096233909 05
6117427663439764954916401592565651111571414769257187704568268701243082044 83
8400201357613851006471104244828842270232637747398962711875413488415772647 08
8571125272932490717217468263604683325933469555629785507020775366368002753 61
2709901526248456328209643292122899677436613886360765877886748185299249994 92
1843183573130403496311896614949399409796011301191280067209053259341918813 96
755254317653234915 7376
```

Note the number of bits is not fixed, the larger the number, the more bits required to represent it, so while overflow is impossible, it is possible to run out of memory if a number is astronomically large: go ahead and try `x^x` (at your own risk).