

## Arithmetic in $\mathbb{Z}$

- ▶ Imagine you want to represent and perform various operations on **integers**, i.e., members of the set

$$\mathbb{Z} = \{\dots, -3, -2, -1, 0, +1, +2, +3, \dots\}.$$

- ▶ A given processor can represent integers as  $w$ -bit **words** and can **naively** operate on such integers, e.g., add them together.
- ▶ We might therefore approximate  $\mathbb{Z}$  by using an appropriate native integer data type with the range  $-2^{w-1} \dots + 2^{w-1} - 1$

$$\begin{array}{lcl} \text{char} & \mapsto & \mathbb{Z}_{\text{char}} = \{ -2^7, \dots, 0, \dots, +2^7 - 1 \} \\ \text{short} & \mapsto & \mathbb{Z}_{\text{short}} = \{ -2^{15}, \dots, 0, \dots, +2^{15} - 1 \} \\ \text{int} & \mapsto & \mathbb{Z}_{\text{int}} = \{ -2^{31}, \dots, 0, \dots, +2^{31} - 1 \} \end{array}$$

- ▶ Looking at the real  $\mathbb{Z}$ , the ominous “...” are a problem: the magnitude of some  $x \in \mathbb{Z}$  can be **much** larger than a  $w$ -bit word can cope with, so we need
  1. a **data structure** to represent  $x$ , and
  2. some **algorithms** to operate on  $x$ .
- ▶ **GNU Multiple Precision (GMP)** is the standard library of this type.

## Data Structure (1)

- ▶ One can express the value of any **unsigned** integer  $x$  using a **base- $b$  expansion**

$$\sum_{i=0}^{n-1} x_i \cdot b^i$$

where  $x_i$  is the  $i$ -th **digit** “weighted” by some power of  $b$  and taken from the **digit set**  $\{0, 1, \dots, b - 1\}$ .

- ▶ Examples are easy to come by; you (hopefully) use base-10 or **decimal** expansions all the time

$$\begin{aligned} 123 &= 3 \cdot 10^0 + 2 \cdot 10^1 + 1 \cdot 10^2 \\ &= 3 \cdot 1 + 2 \cdot 10 + 1 \cdot 100 \end{aligned}$$

where each  $x_i \in \{0, 1, \dots, 9\}$  ...

- ▶ ... but clearly we can select **any**  $b \geq 2$  we want, e.g., base-2 or **binary**

$$\begin{aligned} 123 &= 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 \\ &= 1 \cdot 1 + 1 \cdot 2 + 0 \cdot 4 + 1 \cdot 8 + 1 \cdot 16 + 1 \cdot 32 + 1 \cdot 64 \end{aligned}$$

where each  $x_i \in \{0, 1\}$ .

## Data Structure (2)

► Some things to note:

1. If we write a constant as  $x_{(b)}$ , this means  $x$  is written in base- $b$ ; where there isn't a base, assume decimal.
2. The value  $|x_{(b)}|$  is the number of digits in the base- $b$  expansion of  $x$ , i.e.,  $n$ .
3. The digit  $x_{n-1}$  is the **most-significant** while  $x_0$  is the **least-significant**.
4. Writing a constant such as  $123_{(10)}$  is equivalent to writing the digits as a sequence

$$\langle 3, 2, 1 \rangle_{(10)}.$$

since in both cases it is clear which the  $i$ -th digit is.

5. Using a **sign-magnitude** scheme allows us to cope with **signed** integers by associating a single **sign bit**  $s$  with  $x$ , i.e., letting the value be

$$-1^s \cdot \sum_{i=0}^{n-1} x_i \cdot b^i.$$

However, native integer data types more commonly use the **twos-complement** scheme.

## Data Structure (3)

- ▶ Imagine we'd like to represent and operate on a 128-bit integer  $x$  using a 32-bit processor; we could represent  $x$  as a long base-2 sequence ...

$$x = \langle x_0, x_1, \dots, x_{127} \rangle_{(2)}$$

- ▶ ... but this doesn't really make sense: the processor offers us the use of  $w$ -bit words, so why not just set  $b = 2^w$ ? For  $w = 32$ , we'd let

$$x = \langle x_0, x_1, x_2, x_3 \rangle_{(2^{32})}$$

- ▶ The **same** principles apply to interpreting the value, e.g.,

$$\begin{aligned} \langle 3, 2, 1 \rangle_{(10)} &= 3 \cdot 10^0 + 2 \cdot 10^1 + 1 \cdot 10^2 \\ &= 123_{(10)} \end{aligned}$$

where each  $x_i \in \{0, 1, \dots, 9\}$ , or

$$\begin{aligned} \langle 3, 2, 1 \rangle_{(2^{32})} &= 3 \cdot (2^{32})^0 + 2 \cdot (2^{32})^1 + 1 \cdot (2^{32})^2 \\ &= 18446744082299486211_{(10)} \end{aligned}$$

where each  $x_i \in \{0, 1, \dots, 2^{32} - 1\}$ ; GMP calls these “limbs”.

## Data Structure (4)

- ▶ The **data structure** we'll use is based exactly on this approach ...
  1. Each digit will be represented by an unsigned integer data type that matches the processor word size; for  $w = 32$

```
typedef unsigned int      WORD;  
typedef unsigned long long DWORD;
```

2. A given  $x \in \mathbb{Z}$  will be represented by `mpz`

```
typedef struct __mpz  
{  
    WORD data[ MPZ_SIZE_MAX ];  
  
    int  size;  
    int  sign;  
}  
mpz;
```

where if  $x$  is an instance of said structure, then

- ▶ `x.data` is the sequence of digits representing the magnitude of  $x$ .
  - ▶ `x.size` is  $|x_{(2^w)}|$  or  $n$ , the number digits used within `x.data`.
  - ▶ `x.sign` is the sign-bit of  $x$ , i.e., this tells us when  $x$  is positive or negative.
- ▶ ... you may hear this described as a “**multi-precision**” or “**big**” integer data type.

## Data Structure (5)

- ▶ Like  $\mathbb{Z}_{\text{int}}$  et. al, the magnitude of an integer represented by `mpz` is **still** limited (since we only have finite memory).
- ▶ In a sense, we just improved this limit: we **still** approximate  $\mathbb{Z}$ , but on the other hand **aren't** limited by native integer data types.
- ▶ Even so, we've made quite a major design decision:
  1. We **statically** allocate a **fixed** number of digits to `x.data`, and keep track of how many are used in `x.sign`.
    - ▶ The magnitude of an integer represented by `mpz` is limited by how much memory we **initially allocate** ...
    - ▶ ... for use in cryptography this isn't usually a problem.
  2. GMP, for example, **dynamically** manages the memory by resizing `x.data` (GMP allows it to grow, but doesn't shrink it) as needed.
    - ▶ The magnitude of an integer represented by `mpz` is limited by how much memory we have **in total** ...
    - ▶ ... but, this implies some performance overhead.

# Algorithms (1)

- ▶ The next step is use of the data structure within algorithms that provide the behaviour we want; our strategy will be to construct several **layers**:
- ▶ **Layer #1** (the “low-level” layer):
  - ▶ Provides arithmetic on digits using **digit operations**.
- ▶ **Layer #2** (the “mid-level” layer):
  - ▶ Provides arithmetic on sequences of digits using **digit sequence operations**.
  - ▶ This essentially gives us unsigned integer arithmetic, i.e., arithmetic in  $\mathbb{N}$ , so (roughly speaking) we can perform

$$\langle 3, 2, 1 \rangle_{(2^w)} + \langle 6, 5, 4 \rangle_{(2^w)}$$

- ▶ Relies on digit operations, and deals with issues such as carry-propagation between digits.
- ▶ **Layer #3** (the “high-level” layer):
  - ▶ Provides arithmetic on `mpz` using **integer operations**.
  - ▶ This essentially gives us signed integer arithmetic, i.e., arithmetic in  $\mathbb{Z}$ , so (roughly speaking) we can perform

$$\pm \langle 3, 2, 1 \rangle_{(2^w)} + \pm \langle 6, 5, 4 \rangle_{(2^w)}$$

- ▶ Relies on digit sequence operations, and deal with issues such as management of `x.sign` and `x.size`.

## Algorithms (2): Layer #1, LIMB\_ADD and LIMB\_MAC macros

- ▶ The low-level layer would consist of (at least) two macros that provide basic digit operations for the case where  $b = 2^w \dots$

### 1. LIMB\_ADD (a digit-based “add with carry”)

- ▶ The idea is to compute the equivalent of

$$r_1 \cdot 2^w + r_0 = e + f + g.$$

- ▶ If  $e$  and  $f$  are  $w$ -bit digits, and  $g$  is a 1-bit carry, this yields at most a  $(w + 1)$ -bit result  $r$ , since

$$(2^w - 1) + (2^w - 1) + 1 = 2^{w+1} - 1 < 2^{w+1},$$

which is **split** back into  $w$ -bit digits:  $r_1$  is the 1 most-significant bit,  $r_0$  is the  $w$  least-significant bits.

### 2. LIMB\_MAC (a digit-based “multiply-accumulate”)

- ▶ The idea is to compute the equivalent of

$$r_1 \cdot 2^w + r_0 = e \cdot f + g + h.$$

- ▶ If  $e$ ,  $f$ ,  $g$  and  $h$  are all  $w$ -bit digits, this yields at most a  $2w$ -bit result, since

$$(2^w - 1)(2^w - 1) + (2^w - 1) + (2^w - 1) = 2^{2w} - 1 < 2^{2w},$$

which is **split** back into  $w$ -bit digits:  $r_1$  is the  $w$  most-significant bits,  $r_0$  is the  $w$  least-significant bits.

- ▶ ... the problem is, how do we realise these (preferably in an efficient way) ?

## Algorithms (3): Layer #1, LIMB\_ADD and LIMB\_MAC macros

- ▶ LIMB\_ADD option #1:

```
#define LIMB_ADD(r1,r0,e,f,g) \
{ \
    DWORD t0 = (DWORD)(e); \
    t0 = t0 + (DWORD)(f); \
    t0 = t0 + (DWORD)(g); \
    \
    r0 = (WORD)(t0) ; \
    r1 = (WORD)(t0 >> BITSOF(WORD)) & 1; \
}
```

- ▶ Consider an example where  $w = 32$ , and we set  $e = 4294967295_{(10)}$ ,  $f = 1_{(10)}$  and  $g = 1_{(10)}$ ; the sequence works as follows ...
  1.  $t_0 = e = 4294967295_{(10)}$ .
  2.  $t_0 = t_0 + f = 4294967295_{(10)} + 1_{(10)} = 4294967296_{(10)}$ .
  3.  $t_0 = t_0 + g = 4294967296_{(10)} + 1_{(10)} = 4294967297_{(10)}$ .
  4.  $r_0 = t_0 \bmod 2^{32} = 4294967297_{(10)} \bmod 2^{32} = 1_{(10)}$ .
  5.  $r_1 = \lfloor t_0 / 2^{32} \rfloor \wedge 1_{(10)} = \lfloor 4294967297_{(10)} / 2^{32} \rfloor \wedge 1_{(10)} = 1_{(10)}$ .
- ▶ ... to give the result  $r_1 \cdot 2^{32} + r_0 \cdot 1 = 1 \cdot 2^{32} + 1 \cdot 1 = 4294967297_{(10)}$ .

## Algorithms (4): Layer #1, LIMB\_ADD and LIMB\_MAC macros

► LIMB\_ADD option #2:

```
#define LIMB_ADD(r1,r0,e,f,g) \  
{ \  
    WORD t0 = e + f;          \  
    WORD t1 = t0 < e;        \  
    r0 = t0 + g;             \  
    WORD t2 = r0 < t0;      \  
    r1 = t1 | t2;           \  
}
```

► The sequence works as follows ...

1.  $t_0 = e + f = 4294967295_{(10)} + 1_{(10)} = 2^{32}$  so the actual result “wraps around” to give  $t_0 = 0_{(10)}$ .
  2.  $t_0 < e$  so  $t_1 = 1_{(10)}$ , i.e., the first add caused a carry-out.
  3.  $r_0 = t_0 + g = 0_{(10)} + 1_{(10)} = 1_{(10)}$ .
  4.  $r_0 \not< t_0$  so  $t_2 = 0_{(10)}$ , i.e., the second add didn't cause a carry-out.
  5.  $r_1 = t_1 \vee t_2 = 1_{(10)} \vee 0_{(10)} = 1_{(10)}$  so there was a carry-out overall.
- ... to give the result  $r_1 \cdot 2^{32} + r_0 \cdot 1 = 1 \cdot 2^{32} + 1 \cdot 1 = 4294967297_{(10)}$ .

## Algorithms (5): Layer #1, LIMB\_ADD and LIMB\_MAC macros

- ▶ LIMB\_ADD option #3:

```
#define LIMB_ADD(r1,r0,e,f,g) \
{ \
    asm( "movl %2,%0; movl $0,%1; \
        addl %3,%0; adcl $0,%1; \
        addl %4,%0; adcl $0,%1;" \
        : "+&r" (r0), "+&r" (r1) \
        : "r" (e), "r" (f), \
          "r" (g) \
        : "cc" ); \
}
```

- ▶ The sequence works as follows ...
  - ▶ The first `movl` instruction moves  $e$  into  $r_0$ , the second sets  $r_1 = 0$ .
  - ▶ The first `addl` instruction adds  $f$  to  $r_0$ ; any carry-out is stored on the carry flag so the first `adcl` adds the carry-out directly to  $r_1$ .
  - ▶ The second `addl` instruction adds  $g$  to  $r_0$ ; again any carry-out is stored on the carry flag so the second `adcl` adds the carry-out directly to  $r_1$ .
- ▶ ... to give the result  $r_1 \cdot 2^{32} + r_0 \cdot 1 = 1 \cdot 2^{32} + 1 \cdot 1 = 4294967297_{(10)}$ .

## Algorithms (6): Layer #1, LIMB\_ADD and LIMB\_MAC macros

▶ LIMB\_MAC option #1:

```
#define LIMB_MAC(r1,r0,e,f,g,h)          \
{                                        \
    DWORD t0 =      ( DWORD )( e );    \
        t0 = t0 * ( DWORD )( f );      \
        t0 = t0 + ( DWORD )( g );      \
        t0 = t0 + ( DWORD )( h );      \
                                        \
    r0 = ( WORD )( t0                    ); \
    r1 = ( WORD )( t0 >> BITSOF( WORD ) ); \
}
```

- ▶ The sequence is similar in style to LIMB\_ADD option #1, but the problems are more subtle:
- ▶ This doesn't scale: for  $w = 64$ , there usually isn't really a 128-bit integer data type in C ...
  - ▶ ... and even though there **is** a 64-bit integer data type, with  $w = 32$  there may be no native 64-bit addition so the compiler might have to synthesise one.
  - ▶ Plus, x86-32 **forces** the operands of multiplication to be in specific registers; this can cause the compiler to misbehave.

## Algorithms (7): Layer #1, LIMB\_ADD and LIMB\_MAC macros

► LIMB\_MAC option #2:

```
#define LIMB_MAC(r1,r0,e,f,g,h)      \
{                                     \
    asm ( "movl %2,%%eax; mull %3      ; \
          addl %4,%%eax; adcl $0,%%edx; \
          addl %5,%%eax; adcl $0,%%edx; \
          movl %%eax,%0; movl %%edx,%1;" \
          : "=&g" (r0), "=&g" (r1)    \
          : "1" (e), "r" (f),        \
            "r" (g), "0" (h)         \
          : "%eax", "%edx", "cc"     ); \
}
```

► The sequence works as follows ...

- The first `movl` moves `e` into `%eax`, then the `mull` multiplies this by `f`.
- The first `addl` adds `g` to `%eax`; any carry-out is stored on the carry flag so the first `adcl` adds the carry-out directly to `%edx`.
- The second `addl` instruction adds `c` to `%eax`; again any carry-out is stored on the carry flag so the second `adcl` adds the carry-out directly to `%edx`.
- Finally we move `%eax` and `%edx` into `t0` and `t1`.

## Algorithms (8): Layer #2, `mpn_add`, `mpn_sub` and `mpn_mul` functions

- ▶ Consider some examples of school-book (or “long”) addition and subtraction of **unsigned** integers:

### Example

$$\begin{array}{r} 6 \quad 2 \quad 3 \\ 5 \quad 6 \quad 7 \quad + \\ \hline \quad \quad 1 \quad 0 \\ \quad \quad 8 \\ \hline 1 \quad 1 \\ \hline 1 \quad 1 \quad 9 \quad 0 \end{array}$$

### Example

$$\begin{array}{r} 6 \quad 2 \quad 3 \\ 5 \quad 6 \quad 7 \quad - \\ \hline \quad \quad -1 \quad 6 \\ \quad -1 \quad 6 \\ \hline 1 \\ \hline \quad \quad 5 \quad 6 \end{array}$$

- ▶ Some things to note:
  - ▶ If we add together  $n$ -digit and  $m$ -digit base- $b$  numbers, the result has at most  $\max(n + m) + 1$  digits.
  - ▶ Subtraction is essentially the same method as addition except any carry-in becomes a borrow-from.
  - ▶ Since we cannot accommodate negative integers yet, we need to ensure  $x \geq y$  so that  $x - y \geq 0$ : trying to compute  $x - y$  if  $x < y$  produces an resulting borrow we can't deal with.

## Algorithms (9): Layer #2, mpn\_add, mpn\_sub and mpn\_mul functions

### Algorithm (N-ADD)

**Input:** Two multi-precision integers,  $x$  and  $y$ , represented in base- $b$ .

**Output:** A multi-precision integer  $r = x + y$ , represented in base- $b$ .

$n \leftarrow \text{MAX}(|x|, |y|)$

$c \leftarrow 0$

**for**  $i = 0$  **upto**  $n - 1$  **step 1** **do**

$t \leftarrow x_i + y_i + c$

$r_i \leftarrow t \bmod b$

$c \leftarrow \lfloor t/b \rfloor$

**end**

$r_n \leftarrow c$

**return**  $r$

### Example

For  $b = 10$ ,  $x = \langle 3, 2, 6 \rangle_{(10)}$  and  $y = \langle 7, 6, 5 \rangle_{(10)}$ :

$i$	$r$	$c$	$x_i$	$y_i$	$t$	$r$	$c$
	$\langle 0, 0, 0, 0 \rangle$						
		0					
0	$\langle 0, 0, 0, 0 \rangle$	0	3	7	10	$\langle 0, 0, 0, 0 \rangle$	1
1	$\langle 0, 0, 0, 0 \rangle$	1	2	6	9	$\langle 0, 9, 0, 0 \rangle$	0
2	$\langle 0, 9, 0, 0 \rangle$	0	6	5	11	$\langle 0, 9, 1, 0 \rangle$	1
		1				$\langle 0, 9, 1, 1 \rangle$	

## Algorithms (10): Layer #2, mpn\_add, mpn\_sub and mpn\_mul functions

### Algorithm (N-SUB)

**Input:** Two multi-precision integers,  $x$  and  $y$  with  $x \geq y$ , represented in base- $b$ .

**Output:** A multi-precision integer  $r = x - y$ , represented in base- $b$ .

$n \leftarrow \text{MAX}(|x|, |y|)$

$c \leftarrow 0$

**for**  $i = 0$  **upto**  $n - 1$  **step** 1 **do**

$t \leftarrow x_i - y_i + c$

$r_i \leftarrow t \bmod b$

$c \leftarrow \lfloor t/b \rfloor$

**end**

**return**  $r$

### Example

For  $b = 10$ ,  $x = \langle 3, 2, 6 \rangle_{(10)}$  and  $y = \langle 7, 6, 5 \rangle_{(10)}$ :

$i$	$r$	$c$	$x_i$	$y_i$	$t$	$r$	$c$
	$\langle 0, 0, 0, 0 \rangle$						
		0					
0	$\langle 0, 0, 0, 0 \rangle$	0	3	7	-4	$\langle 6, 0, 0, 0 \rangle$	-1
1	$\langle 6, 0, 0, 0 \rangle$	1	2	6	-5	$\langle 6, 5, 0, 0 \rangle$	-1
2	$\langle 6, 5, 0, 0 \rangle$	0	6	5	0	$\langle 6, 5, 0, 0 \rangle$	0



## Algorithms (12): Layer #2, `mpn_add`, `mpn_sub` and `mpn_mul` functions

### Algorithm (N-MUL)

**Input:** Two multi-precision integers,  $x$  and  $y$ , represented in base- $b$ .

**Output:** A multi-precision integer  $r = x \cdot y$ , represented in base- $b$ .

```

n ← |x|
m ← |y|
r ← 0
for i = 0 upto n - 1 step 1 do
  c ← 0
  for j = 0 upto m - 1 step 1 do
    t ← xi · yj + ri+j + c
    ri+j ← t mod b
    c ← ⌊t/b⌋
  end
  ri+m ← c
end
return r
    
```

### Example

For  $b = 10$ ,  $x = \langle 3, 2, 6 \rangle_{(10)}$  and  $y = \langle 7, 6, 5 \rangle_{(10)}$ :

$i$	$j$	$r$	$c$	$x_i$	$y_j$	$t$	$r$	$c$
		$\langle 0, 0, 0, 0, 0, 0 \rangle$	0					
0	0	$\langle 0, 0, 0, 0, 0, 0 \rangle$	0	3	7	21	$\langle 1, 0, 0, 0, 0, 0 \rangle$	2
0	1	$\langle 1, 0, 0, 0, 0, 0 \rangle$	2	3	6	20	$\langle 1, 0, 0, 0, 0, 0 \rangle$	2
0	2	$\langle 1, 0, 0, 0, 0, 0 \rangle$	2	3	5	17	$\langle 1, 0, 7, 0, 0, 0 \rangle$	1
0		$\langle 1, 0, 7, 0, 0, 0 \rangle$	1				$\langle 1, 0, 7, 1, 0, 0 \rangle$	
1			0					
1	0	$\langle 1, 0, 7, 1, 0, 0 \rangle$	0	2	7	14	$\langle 1, 4, 7, 1, 0, 0 \rangle$	1
1	1	$\langle 1, 4, 7, 1, 0, 0 \rangle$	1	2	6	20	$\langle 1, 4, 0, 1, 0, 0 \rangle$	2
1	2	$\langle 1, 4, 0, 1, 0, 0 \rangle$	2	2	5	13	$\langle 1, 4, 0, 3, 0, 0 \rangle$	1
1		$\langle 1, 4, 0, 3, 0, 0 \rangle$	1				$\langle 1, 4, 0, 3, 1, 0 \rangle$	
2			0					
2	0	$\langle 1, 4, 0, 3, 1, 0 \rangle$	0	6	7	42	$\langle 1, 4, 2, 3, 1, 0 \rangle$	4
2	1	$\langle 1, 4, 2, 3, 1, 0 \rangle$	4	6	6	43	$\langle 1, 4, 2, 3, 1, 0 \rangle$	4
2	2	$\langle 1, 4, 2, 3, 5, 0 \rangle$	4	6	5	35	$\langle 1, 4, 2, 3, 5, 0 \rangle$	3
2			3				$\langle 1, 4, 2, 3, 5, 3 \rangle$	

#### ► Some things to note:

- Unlike addition and subtraction, we **can't** perform the multiplication in-place;  $r$  can't alias  $x$  or  $y$ , and we need to zero it before use.
- The algorithm complexity is  $O(n^2)$ , so for large  $n$  it can make sense to consider alternatives (e.g., Karatsuba-Ofman, Toom-Cook or even FFT).

## Algorithms (13): Layer #2, `mpn_add`, `mpn_sub` and `mpn_mul` functions

- ▶ Implementing the algorithms gives us the functions `mpn_add`, `mpn_sub` and `mpn_mul`:
  - ▶ `mpn_add`, `mpn_sub` and `mpn_mul` each provide **digit sequence operations**, i.e., arithmetic in  $\mathbb{N}$  ...
  - ▶ ... they each utilise **digit operations** provided by `LIMB_ADD` and `LIMB_MAC` (and `LIMB_SUB` which we didn't discuss).
  - ▶ The algorithms deal with digit sequences that the implementation realises using arrays; we'll pass `x` and `y` around as pointer/length pairs.
  - ▶ We'll rely on a simple auxiliary function

```
int mpn_trim( const WORD* x, int xn )
{
    while( ( xn > 1 ) && ( x[ xn - 1 ] == 0 ) ) {
        xn--;
    }

    return xn;
}
```

that, starting with an initial length, finds the real length of a sequence by “trimming” off any high-order zero digits.

## Algorithms (14): Layer #2, mpn\_add, mpn\_sub and mpn\_mul functions

```
int mpn_add( WORD* r, const WORD* x, int xn, const WORD* y, int yn )
{
    int rn = MAX( xn, yn );

    WORD w0, w1, w2, c0 = 0;

    for( int i = 0; i < rn; i++ ) {
        w0 = ( i < xn ) ? x[ i ] : 0;
        w1 = ( i < yn ) ? y[ i ] : 0;

        LIMB_ADD( c0, w2, w0, w1, c0 );

        r[ i ] = w2;
    }

    r[ rn++ ] = c0;

    return mpn_trim( r, rn );
}
```

## Algorithms (15): Layer #2, mpn\_add, mpn\_sub and mpn\_mul functions

```
int mpn_sub( WORD* r, const WORD* x, int xn, const WORD* y, int yn )
{
    int rn = MAX( xn, yn );

    WORD w0, w1, w2, c0 = 0;

    for( int i = 0; i < rn; i++ ) {
        w0 = ( i < xn ) ? x[ i ] : 0;
        w1 = ( i < yn ) ? y[ i ] : 0;

        LIMB_SUB( c0, w2, w0, w1, c0 );

        r[ i ] = w2;
    }

    return mpn_trim( r, rn );
}
```

## Algorithms (16): Layer #2, mpn\_add, mpn\_sub and mpn\_mul functions

```
int mpn_mul( WORD* r, const WORD* x, int xn, const WORD* y, int yn )
{
    int rn = xn + yn;

    WORD w0, w1, w2, w3, c0, T[ rn ];

    memset( T, 0, rn * sizeof( WORD ) );

    for( int i = 0; i < xn; i++ ) {
        c0 = 0;

        for( int j = 0; j < yn; j++ ) {
            w0 = x[ i ];
            w1 = y[ j ];
            w2 = T[ i + j ];

            LIMB_MAC( c0, w3, w0, w1, w2, c0 );

            T[ i + j ] = w3;
        }

        T[ i + yn ] = c0;
    }

    memcpy( r, T, rn * sizeof( WORD ) );

    return mpn_trim( r, rn );
}
```

## Algorithms (17): Layer #3, `mpz_add`, `mpz_sub` and `mpz_mul` functions

- ▶ Finally, we can implement `mpz_add`, `mpz_sub` and `mpz_mul`:
  - ▶ `mpz_add`, `mpz_sub` and `mpz_mul` each provide **integer operations**, i.e., arithmetic in  $\mathbb{Z}$  ...
  - ▶ ... they each utilise **digit sequence operations** provided by `mpn_add`, `mpn_sub` and `mpn_mul`.
  - ▶ To get the right behaviour `mpz_add`, for example, must manage the **magnitude** and **sign** of the result according to some rules ...

<code>x +ve</code>	<code>y +ve</code>		$\mapsto$	magnitude is $ABS(x + y)$ , sign is +ve
<code>x -ve</code>	<code>y -ve</code>		$\mapsto$	magnitude is $ABS(x + y)$ , sign is -ve
<code>x -ve</code>	<code>y +ve</code>	$ABS(x) > ABS(y)$	$\mapsto$	magnitude is $ABS(x - y)$ , sign is -ve
<code>x -ve</code>	<code>y +ve</code>	$ABS(x) \leq ABS(y)$	$\mapsto$	magnitude is $ABS(x - y)$ , sign is +ve
<code>x +ve</code>	<code>y -ve</code>	$ABS(x) < ABS(y)$	$\mapsto$	magnitude is $ABS(y - x)$ , sign is -ve
<code>x +ve</code>	<code>y -ve</code>	$ABS(x) \geq ABS(y)$	$\mapsto$	magnitude is $ABS(y - x)$ , sign is +ve

... which are easily achieved by using `mpn_add` and `mpn_sub` since they support unsigned arithmetic.

- ▶ Other arithmetic (e.g., exponentiation) may use an integer operation directly (e.g., `mpz_mul`) rather than the lower-layers (e.g., `mpn_mul`).

## Algorithms (18): Layer #3, mpz\_add, mpz\_sub and mpz\_mul functions

```
void mpz_add( mpz* r, const mpz* x, const mpz* y )
{
  if ( ( x->sign == MPZ_SIGN_NEG ) && ( y->sign == MPZ_SIGN_POS ) ) {
    if( mpn_isgth( x->data, x->size, y->data, y->size ) ) {
      r->size = mpn_sub( r->data, x->data, x->size, y->data, y->size );
      r->sign = MPZ_SIGN_NEG;
    }
    else {
      r->size = mpn_sub( r->data, y->data, y->size, x->data, x->size );
      r->sign = MPZ_SIGN_POS;
    }
  }
  else if( ( x->sign == MPZ_SIGN_POS ) && ( y->sign == MPZ_SIGN_NEG ) ) {
    if( mpn_islth( x->data, x->size, y->data, y->size ) ) {
      r->size = mpn_sub( r->data, y->data, y->size, x->data, x->size );
      r->sign = MPZ_SIGN_NEG;
    }
    else {
      r->size = mpn_sub( r->data, x->data, x->size, y->data, y->size );
      r->sign = MPZ_SIGN_POS;
    }
  }
  else if( ( x->sign == MPZ_SIGN_NEG ) && ( y->sign == MPZ_SIGN_NEG ) ) {
    r->size = mpn_add( r->data, x->data, x->size, y->data, y->size );
    r->sign = MPZ_SIGN_NEG;
  }
  else {
    r->size = mpn_add( r->data, x->data, x->size, y->data, y->size );
    r->sign = MPZ_SIGN_POS;
  }
}
```

## Algorithms (19): Layer #3, mpz\_add, mpz\_sub and mpz\_mul functions

```
void mpz_sub( mpz* r, const mpz* x, const mpz* y )
{
  if ( ( x->sign == MPZ_SIGN_NEG ) && ( y->sign == MPZ_SIGN_POS ) ) {
    r->size = mpn_add( r->data, x->data, x->size, y->data, y->size );
    r->sign = MPZ_SIGN_NEG;
  }
  else if( ( x->sign == MPZ_SIGN_POS ) && ( y->sign == MPZ_SIGN_NEG ) ) {
    r->size = mpn_add( r->data, x->data, x->size, y->data, y->size );
    r->sign = MPZ_SIGN_POS;
  }
  else if( ( x->sign == MPZ_SIGN_NEG ) && ( y->sign == MPZ_SIGN_NEG ) ) {
    if( mpn_isgth( x->data, x->size, y->data, y->size ) ) {
      r->size = mpn_sub( r->data, x->data, x->size, y->data, y->size );
      r->sign = MPZ_SIGN_NEG;
    }
    else {
      r->size = mpn_sub( r->data, y->data, y->size, x->data, x->size );
      r->sign = MPZ_SIGN_POS;
    }
  }
  else {
    if( mpn_islth( x->data, x->size, y->data, y->size ) ) {
      r->size = mpn_sub( r->data, y->data, y->size, x->data, x->size );
      r->sign = MPZ_SIGN_NEG;
    }
    else {
      r->size = mpn_sub( r->data, x->data, x->size, y->data, y->size );
      r->sign = MPZ_SIGN_POS;
    }
  }
}
```

## Algorithms (20): Layer #3, mpz\_add, mpz\_sub and mpz\_mul functions

```
void mpz_mul( mpz* r, const mpz* x, const mpz* y )
{
    int sign;

    if( x->sign == y->sign ) {
        sign = MPZ_SIGN_POS;
    }
    else {
        sign = MPZ_SIGN_NEG;
    }

    if ( mpz_iseqz( x ) ) {
        mpz_cpz( r );
    }
    else if( mpz_iseqz( y ) ) {
        mpz_cpz( r );
    }
    else if( mpz_iseqo( x ) ) {
        mpz_cpy( r, y );
    }
    else if( mpz_iseqo( y ) ) {
        mpz_cpy( r, x );
    }
    else {
        r->size = mpz_mul( r->data, x->data, x->size, y->data, y->size );
        r->sign = sign;
    }
}
```

# Conclusions

- ▶ There are a **lot** of missing integer operations that are useful (and sometimes vital), but we didn't cover ...
  1. Comparisons, e.g.,  $x = y$ ,  $x > y$ ,  $x < y$ ,  $x = 0$ .
  2. Left- and right-shifts, i.e.,  $x \cdot 2^k$ .
  3. Squaring, i.e.,  $x^2$ , which is more efficient than general-purpose multiplication.
  4. Division, e.g.,  $\lfloor \frac{x}{y} \rfloor$ ,  $x \bmod y$ .
  5. Number theoretic algorithms, e.g.,  $\gcd(x, y)$  and  $\text{xgcd}(x, y)$ .
- ▶ ... but they all take a roughly similar layered strategy; abstraction like this yields some significant advantages:
  - ▶ We can change behavioural characteristics internally as long as we retain the same functional characteristics externally, e.g.,
    1. if we move to a dynamic memory management, usage need not change, or
    2. we might dynamically select an algorithm based on operand length at run-time rather than statically at compile-time.
  - ▶ We can exploit architectural features in a particular processor without altering the higher-layers; this makes it easy to port the implementation.

## Further Reading

- ▶ V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2008. ISBN: 0-521-85154-8.
  - ▶ Chapter 3 – Computing with large integers
- ▶ A.J. Menezes, P.C. van Oorschot and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 0-8493-8523-7.
  - ▶ Chapter 14 – Efficient implementation
- ▶ GNU Multiple Precision Arithmetic Library (GMP). <http://gmplib.org/>