Managing Multiple Precision Integer Data on Modern 64-Bit Architectures

Dominique Colnet
Dominique.Colnet@loria.fr
Université de Lorraine
Nancy, France

Benoît Sonntag*
Benoit.Sonntag@lisaac.org
Université de Strasbourg
Strasbourg, France

Abstract

From the perspective of integer numerical computation, a good programming language should allow both safe, overflow-free operations, and direct manipulation of processor-native types, which are inherently limited in bit width and encoded using two's complement. Current 64-bit architectures offer unprecedented low-level capabilities. For the first time in computing history, the size of address registers greatly exceeds the physical capacity of the memory bus. This discrepancy leaves several unused bits that can be exploited to design more compact and expressive data representations.

We present here a concrete implementation that leverages these vacant bits to efficiently represent signed integers within a single 64-bit memory word. Two variants are proposed: one designed for hardware implementation, and the other more suited to a software setting. In addition, both representations make it possible to unify the encoding of numbers using a single machine word, which also facilitates the adoption of functional, rather than procedural, notation.

Keywords: 64 bits architecture, address bus, large integer

1 Introduction

In the process of defining our *Omega* language [1], we have carefully considered the implementation of numbers. Omega (Ω) is a strongly typed language that offers both the type $\mathbb{Z}$, which represents signed integers with no limits other than memory size, and a range of machine-specific integer types such as $\mathbb{Z}_8$, $\mathbb{Z}_{16}$, $\mathbb{Z}_{32}$, and $\mathbb{Z}_{64}$, corresponding to signed integers constrained to 8, 16, 32, and 64 bits, respectively.

It is crucial to ensure that numerical operations can be handled without the risk of overflow for type $\mathbb{Z}$, while also leveraging machine-level data types when necessary ($\mathbb{Z}_8$, $\mathbb{Z}_{16}$, $\mathbb{Z}_{32}$, etc.). The procedural use of type $\mathbb{Z}$ involves progressively modifying the result of a calculation within the same memory space, optimizing for performance. At the same time, especially for beginners, it is important to offer a fully functional paradigm that maintains a high level of performance, ensuring flexibility and redability without sacrificing efficiency.

*Both authors contributed equally to this research.

2 The revenge of address registers

Several times in its history, in the competition to increase the power of computers, we've seen the emergence of tricks to first increase the size of the address bus before questioning the whole architecture.

For example, the mythical Z80 processor [2], with its 8-bit data bus and 16-bit addressing bus, should have been limited to 64 KB of RAM. But, with its two-memory bank access principle, the Amstrad 6128 [3] extends its memory to 128KB, doubling the capacity of its predecessor, the 464.

As part of the 8086 family [4] with its 16-bit address register architecture, this processor has been equipped with a 20-bit address bus thanks to the addition of segment registers. The address is made up of a pair of registers {segment,offset}. The offset provides access to a contiguous 64KB range in memory. As for the segment registers, they allow memory jumps in 16-byte steps. Thus, the segment registers provide the 4 bits of high address required to reach the megabyte of RAM (1 MB). The final address calculation is given by:

$$Address = R_{\text{segment}} \times 16 + R_{\text{offset}}$$

After a brief appearance of 24-bit processors with the 80286, came the 32-bit 80386 architecture [5], with a bus capable of addressing 4 GB of memory. But again, in the last years of his reign, with a clever combination of segmentation and pagination, the limit of this architecture has been pushed back to 36-bit addressing.

Then came 64-bit architecture [6], which for the first time in the history of computing has address registers that can go far beyond the memory capacities currently physically available. The astronomical number of 16 Eio ($2^{64} = 1.8e19$ bytes) that such a register could theoretically address is so far out of reach, even in the distant future, that designers preferred to truncate the logical address to 48 bits. In this apparently arbitrary choice, the number of indirections needed to manage pagination must also be taken into account. In fact, with 4 KB pages, we have the 12 least significant bits addressable contiguously, then 4 indirection tables of 9 bits each must be consulted to reach a 48-bit physical address[1].

So, quite surprisingly, this 48-bit logical address leaves a 16-bit high-order range unused for each address pointer location. In addition, we can observe that memory allocators always allocate structures aligned at least with the machine word size. So, if we consider a structure address, we also have

[1]512 entries for each indirection 4KB table: $2^9 \times 64$ bits = 4KB.

the 3 least significant bits, which are always 0. Interestingly, on a 64-bit architecture, all structure pointers have only 45 significant bits. However, their use requires a few precautions before considering them as pointers. It's worth noting that using a mask with a binary-and (&) or a 3-bit binary shift to the left (<<) are manipulations that have a marginal cost at runtime. For each address, we therefore have 16 high-order bits at our disposal, and potentially 3 low-order bits that are free for other uses!

The thread running through this article is how best to use these insignificant bits in addresses to store precise information in a given context. This practice is already widely used in the design of architectures and operating systems. For example, in a pagination indirection table, each page address is aligned on 4 KB, so the 12 least significant bits are ignored when the MMU reads the page address. These 12 unused bits contain other indicators, such as the right to write in this very memory page. There are also other indicators defined and used solely by the operating system.

Further away from hardware and operating systems, we also find this kind of approach at the software level, or more precisely at the language and compilation level in [7]. In Bonds et al (1992), *Tracking bad apples: Reporting the origin of null and undefined value errors*, the proposal is to internally replace the `Null` value of a source program with deliberately invalid pointers to encode information. Here, the information is used to trace the origin of the `Null` in the event of an application crash.

Let's now look at how we can take advantage of all these findings concerning address format, with the management of numbers without overflow.

3 Towards integers that don't overflow

For the vast majority of programming languages, built-in integer types are limited to a certain size. For example, `Java`'s `int` type is limited to 32 bits. A few rare cases of overflow are sometimes detected by the `Java` compiler only when the values are statically determinable. In all cases, no overflow test is performed at runtime. In this way, a positive value of type `int` can be made negative by simply incrementing by 1. This is often surprising, especially for novice programmers, but can sometimes be used deliberately by experienced developers.

Another example from a more recent language is `Rust` [8], which notes this type `i32` or `u32` for the unsigned version. In addition to the more appropriate type name, `Rust` in *debug* mode only, offers overflow control. In *release* mode, overflow control is not performed. As you might expect, `Rust`'s objective is to take full advantage of the processor's power. There are even specialized functions to bypass overflow problems regardless of the compilation mode[2]. For large numbers without overflow, `Rust` also offers types such as `BigInt` or `BigUint`.

[2]`u32.wrapping_add`, `u64.wrapping_add`, `u64.wrapping_add_signed`, `u64.wrapping_sub`, etc.

Historically, Smalltalk was the first language to natively integrate the concept of an integer that never overflows. This choice is perfectly understandable in terms of comfort for programmers and for people unfamiliar with hardware constraints. For Python [9], which is also interpreted, the choice is similar to Smalltalk: no possible overflow and numbers that can grow in memory size as and when required. In our opinion, there are two major drawbacks to this choice: slowness and the impossibility of easily taking hardware into account.

3.1 Hardware limited types *and* flexible general type

In the Smalltalk language virtual machine [10], each 32-bit word representing an object is split into two parts. The first part stores the information in 30 bits, the remaining two bits being the object's basic type. We therefore have 4 internal object categories, one of which is reserved for a small 30-bit integer. At runtime, if a calculation exceeds 30 bits, the small integer becomes a complex object with a pointer to model a larger integer via a real object which uses an array. Smalltalk's object architecture was particularly well thought-out for its time, in the context of a 32-bit architecture and a pure, non-statically typed object environment. However, we can only note a 30-bit address limit representing only 1 GB of accessible memory. We've taken inspiration from this type of partitioning in the context of 64-bit architecture on a compiled language with static typing.

A good programming language needs to offer both programming comfort and full hardware speed. The choice of the `Rust` language, which makes it possible to preserve types that can exactly match the characteristics of the hardware, must be maintained (i.e. built-in types `i8`, `i32` and `i64` for signed and built-in types `u8`, `u32` and `u64` for unsigned). Rather than resorting to general types specialized in handling numbers without overflow, we propose a hybrid solution that allows, according to variations in calculations, to retain almost all the power of the small, limited numbers that exist natively. In fact, depending on the evolution of calculations, for example, two very large numbers that are subtracted from each other can return to the 32-bit representation interval. In such a case, it's interesting to get closer to the performance of native types for the result of this operation.

Unlike Smalltalk, which is completely and uniquely typed at runtime, we're working with a statically typed language. For our proposal, it doesn't matter whether the typing is explicit or based on type inference. Knowing that a given variable can only contain signed integers means that we can reuse the Smalltalk implementation idea with less variability in the entities represented. So, unlike Smalltalk, and thanks to static typing, we have the whole machine word to best encode our integer. In the following, we'll assume that the programming language offers a native type for handling signed integers, which we'll call $\mathbb{Z}$. The idea is to always use a 64-bit machine word for any variable of type $\mathbb{Z}$.

4 Hardware Oriented Solution

Internally, and completely transparently to the user, there are three possible representations for the $\mathbb{Z}$ type. The aim is to make the most of 64-bit and get the best performance out of it. The choice between these 3 representations is directly related to the size of the integer you need to model. Figure 1 illustrates the following explanations of our 3 encoding formats for integers.

4.1 Value requires less than 64 bits - Top of figure 1

The first encoding format is for a small integer that can be encoded on 63 bits, that is in two's complement the range from (-2^{62}) to $(2^{62}-1)$. This format is distinguished from the other two by the most significant bit being set to 0. Unsurprisingly, the use of the other 63 least significant bits stores our integer using two's complement representation. With a barely perceptible increase in computing time compared to using a 64-bit raw machine word, it enables fast management of small integers. As with Smalltalk, when this capacity is exceeded, we dynamically migrate the integer to the second format.

4.2 Value requires 64 to 2^{20} bits - Middle of figure 1

The second format for the $\mathbb{Z}$ type is the most complex and is particularly compact. It can encode integers requiring a maximum of 16384×64 bits (i.e. an integer of 131 072 bytes or 1 Mbits). Thanks to its already particularly wide range, it's more than sufficient for most applications. The 64-bit machine word representing the integer stores 3 pieces of information:

- The address of a contiguous memory area containing the integer using word of 64 bits.
- The maximum capacity of this memory area. If necessary, the memory area is reallocated with a capacity twice that of the previous one[3].
- The size actually used in the memory area. In other words, the number of machine words needed to represent the integer in binary form.

The distribution of information within the 64 bits is as follows:

63 Set to 1 to avoid being identified as the previously described encoding for small values.

59-62 The capacity in power of 2. This 4-bit number can be used to encode capacities ranging from 2^0 to 2^{14}. Setting the 4 bits to 1 (2^{15}) is forbidden, and we reserve this value to identify the next encoding format. We therefore have an array with a maximum capacity of 16384 64-bit cells.

45-58 This 14-bit range encodes the exact size actually used, from 1 to 16384.

0-44 A 45-bit range representing the address of the corresponding allocated area. As this area is 64-bit aligned, a 3-bit left shift gives the exact 48-bit valid address.

[3]The well-known heuristic of doubling the capacity of a dynamic array is especially relevant to our use case. In this case, the multiplication of two n-bit integers uses $2 \times n$ bits.

4.3 Requires More than 2^{20} bits - Bottom of figure 1

The last format for type $\mathbb{Z}$ is shown at the bottom of figure 1 and is of a more standard design. It is identified by the presence of 1 on all bits from 59 to 63. The least significant part of the first 48 bits is an address to a standard object structure. The corresponding object contains a `capacity`, `size` and `storage` field for the usual implementation of a dynamic size array containing the huge integer[4].

4.4 Decoding Algorithm of figure 1

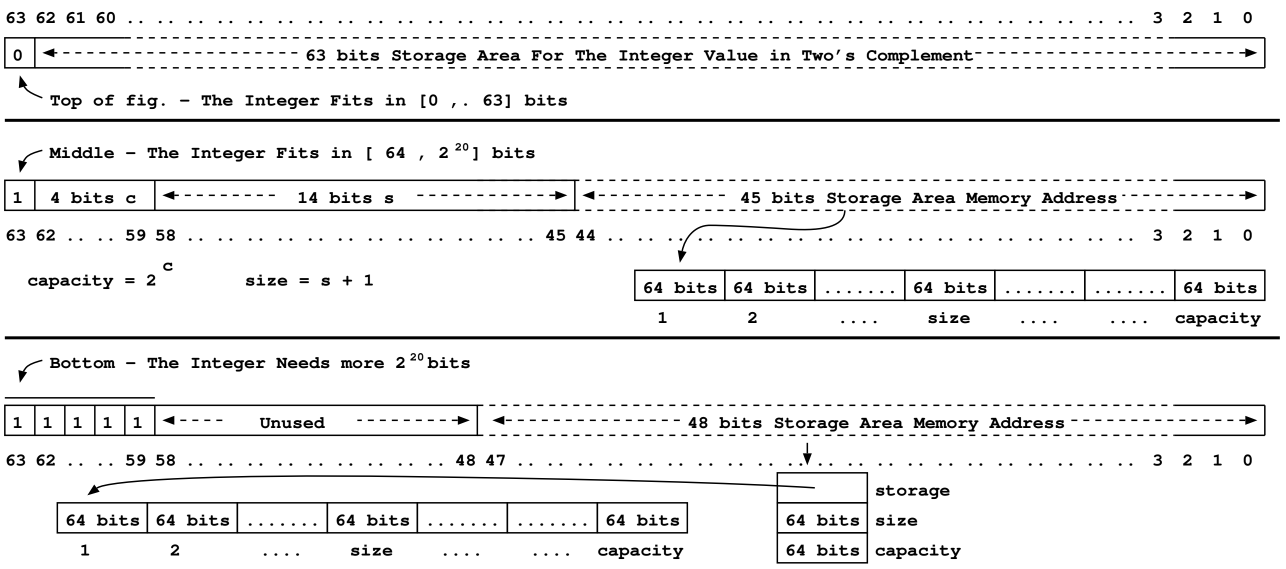
Algorithm 1 illustrates in pseudo-code the decoding of different formats of the type $\mathbb{Z}$. In this algorithm W is the machine word representing the integer encoded in one of the 3 formats, and INT represents bit access to the integer. Note that when using a small integer of less than 64 bits, the extra cost compared to the standard 64-bit basic type is just sign detection (or the position of bit 63) and a jump.

Algorithm 1 Decoding Algorithm (hardware-oriented)

```
if ($W \geq 0$) then
    // Less than 64 bits (top fig. 1)
    $INT \leftarrow ((((signed64)W) \ll 1) \gg 1)$
else
    $cap \leftarrow (W \gg 59)\ \&\ Fh$
    if ($cap = 1111b$) then
        // More than $2^{20}$ bits (bottom fig. 1)
        $a \leftarrow W\ \&\ FFFF\ FFFF\ FFFFh$
        $buf \leftarrow a$.storage
        $siz \leftarrow a$.size
        $cap \leftarrow a$.capacity
    else
        // Range 64 to $2^{20}$ bits (middle fig. 1)
        $buf \leftarrow (W\ \&\ 1FFF\ FFFF\ FFFFh) \ll 3$
        $siz \leftarrow ((W \gg 45)\ \&\ 3FFFh) + 1$
        $cap \leftarrow 1 \ll cap$
    end if
    $INT \leftarrow \mathbf{tab}(buf, siz, cap)$
end if
```

Note also that our representation is deliberately canonical. Depending on its size in number of bits, a given number has to be represented in just one of the three memory formats. This makes it easy to compare numbers with each other. In

[4]This representation is named `ArrayList` in the `Java` library. In `C++` this data structure is also known as `std::vector`.



63 62 61 60 . . . 3 2 1 0
0
63 bits Storage Area For The Integer Value in Two's Complement
Top of fig. - The Integer Fits in [0 ,. 63] bits
Middle - The Integer Fits in [64 , 2^20] bits
1
4 bits c
14 bits s
45 bits Storage Area Memory Address
63 62 . . . 59 58 . . . 45 44 . . . 3 2 1 0
capacity = 2^c
size = s + 1
64 bits
1
2
size
capacity
Bottom - The Integer Needs more 2^20 bits
1 1 1 1 1
Unused
48 bits Storage Area Memory Address
63 62 . . . 59 58 . . . 48 47 . . . 3 2 1 0
storage
size
capacity

Figure 1. The three memory representations for a variable of type $\mathbb{Z}$, depending on the number of bits required for the value.

particular, the comparison between two small numbers (i.e. less than 64 bits) is made with the usual machine instruction for comparing two memory words.

However, this solution has a major drawback, especially when performing a binary operation. It is crucial to determine whether both operands are in the small format, the medium format, or if they belong to different formats. With three possible formats for each operand, this leads to six cases to handle. Moreover, the unpredictability of the format in which the result will be encoded further slows down the computation. That said, this very compact format could benefit from a particularly efficient hardware implementation thanks to the parallelization of encoding and decoding operations. In the next section, we introduce a representation that is better suited to a purely software implementation.

5 Software Oriented Representation

The solution we have chosen allows for an efficient implementation, even when only software-based tagging is available. This is therefore a purely sequential case study. The approach consists in deliberately limiting ourselves to two cases, while still retaining the ability to represent very large numbers. As in the previous representation, a variable of type $\mathbb{Z}$ always corresponds to a 64-bit memory location, that is, a single machine word. Figure 2 illustrates the two possible representations of a $\mathbb{Z}$ variable, where only the least significant bit serves as the selector. When this bit is equal to 0, the number lies within the range $[-(2^{62}-1),\ (2^{62}-1)]$. Conversely, if this bit is equal to 1, the number lies outside this range.

5.1 Small Integer - Top of figure 2

When the least significant bit is 0, a simple arithmetic right shift by one position is sufficient to obtain the corresponding 64-bit value.

5.2 Large Integer - Bottom of figure 2

When the least significant bit is 1, this indicates that the value lies outside the interval $[-(2^{62}-1),\ (2^{62}-1)]$. In this case, the absolute value of the signed integer is stored in a separate memory area, organized as an array of 64-bit words. A pointer to this memory area is encoded in the lower 48 bits of the memory word corresponding to the $\mathbb{Z}$ variable. Bits 48 to 55 (see Figure 2) contain a value *cap*, which is used to compute the capacity of the memory area, denoted as `capacity`, according to the formula `capacity` $= 2^{cap}$. The `capacity` index corresponds to the last slot of the storage array.

This *cap* value also makes it possible to retrieve the size of the allocated memory block, particularly during deallocation or reallocation. Since the storage area may be only partially used, the `size` variable is stored at index -1 of this memory region.

Regarding capacity management, we follow the same approach as in [11]: the size of the memory area is systematically doubled whenever a resize is needed. As a result, the capacity is always a power of 2.

Given that *cap* is constrained to the interval $[1, 255]$, the capacity can range from 2^1 to 2^{255} 64-bit words, which far exceeds the capabilities of current hardware architectures.

5.3 Decoding Algorithm of figure 2

Algorithm 2 illustrates in pseudo-code the decoding of the two different formats of the type $\mathbb{Z}$. In this algorithm W

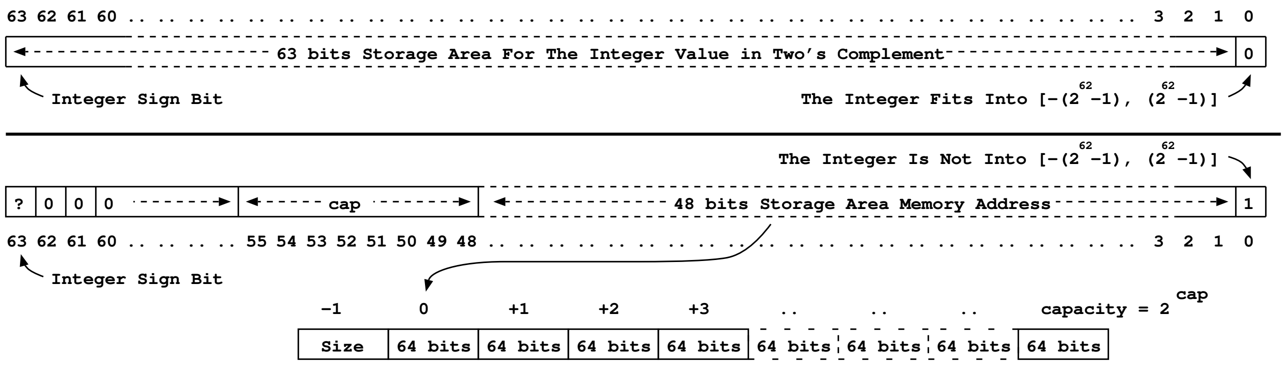

63 62 61 60 .. .. 3 2 1 0
63 bits Storage Area For The Integer Value in Two's Complement
0
Integer Sign Bit
The Integer Fits Into [−(2^62 −1), (2^62 −1)]
The Integer Is Not Into [−(2^62 −1), (2^62 −1)]
? 0 0 0
cap
48 bits Storage Area Memory Address
1
63 62 61 60 .. 55 54 53 52 51 50 49 48 .. 3 2 1 0
Integer Sign Bit
−1 0 +1 +2 +3 .. .. ..
capacity = 2^cap
Size
64 bits

Figure 2. The two memory mappings of type $\mathbb{Z}$ based on necessary bit count: top of figure small values, otherwise bottom.

Algorithm 2 Decoding Algorithm (software-oriented)

```
if $(W \& 1)$ then
    // Not Within $[-(2^{62}-1),\ (2^{62}-1)]$ (bottom of fig. 2)
    $cap \leftarrow 1 \ll ((W \gg 48) \,\&\, \mathrm{F}h)$
    $buf \leftarrow W \,\&\, \mathrm{FFFF\,FFFF\,FFFD}h$
    $siz \leftarrow buf[-1]$
    $INT \leftarrow \mathbf{tab}(buf, siz, cap)$
else
    // Within $[-(2^{62}-1),\ (2^{62}-1)]$ (top of figure 2)
    $INT \leftarrow (((signed64)W) \gg 1)$
end if
```

is the machine word representing the integer encoded in one of the two formats, and *INT* represents bit access to the integer. Note also that our representation is deliberately canonical. Depending on whether it lies within the interval or not $[-(2^{62}-1),\ (2^{62}-1)]$, a given number has to be represented in just one of the two memory format. This makes it easy to compare numbers with each other. In particular, the comparison between two small numbers (i.e. less than 64 bits) is made with the usual machine instruction for comparing two memory words.

6 Benchmarks

This section presents a set of measurements we performed to evaluate the $\mathbb{Z}$ type representation in our Ω language. Comparisons were made against GMP [12], which is arguably the fastest library for exact-precision arithmetic [13] [14]. Moreover, since the Ω language generates C code and GMP is also implemented in C, this comparison is particularly relevant. Achieving performance even modestly close to that of GMP already represents a significant challenge. We also included *mini*GMP [15] in these benchmarks, providing an additional point of comparison.

Naturally, we remain focused on our initial objective: integrating into our Ω language a type that allows manipulation of signed integers using functional notation. In other words, providing syntax that is close to mathematical expressions. For example, in Ω, the instruction $a \longleftarrow b + c$ avoids the need to manually manage memory allocation for the result. In contrast, GMP is designed for procedural use, relying on accumulators that are modified across successive operations. Thus, with GMP, the previous instruction would correspond to the procedural call `mpz_add(a,b,c);` where the memory space for the result must have been allocated beforehand.

For all the benchmarks presented below, we systematically perform the same number of operations for each type. We use exactly the same operand values regardless of the library used: whether it is $\mathbb{Z}$, GMP, or *mini*GMP. To avoid disadvantaging GMP and to follow its best practices, the three variables a, b, and c are reused throughout the computations, rather than reallocated before each operation. Finally, we also used GMP to verify the correctness of the computations performed by the $\mathbb{Z}$ library in our new Ω language[5].

6.1 Benchmarks Using Small Values (63 bits)

All the benchmarks presented in this section concern what we refer to as small values, that is, values that fit within 63 bits. The relevance of testing this range of values is twofold. First, it allows us to include the $\mathbb{Z}_{64}$ type, which corresponds exactly to the 64-bit signed integer type supported by the processor[6]. Second, it allows us to evaluate the 63-bit value representation (top part of Figure 2), and thereby quantify any potential loss in execution speed compared to the $\mathbb{Z}_{64}$ type. Finally, it also enables us to quantify the performance gain compared to the GMP implementation, which is particularly well-suited for large numbers, but not necessarily for such small values.

We compare the following operations: addition (Fig. 3), subtraction (Fig. 4), multiplication (Fig. 5), division (Fig. 6), left shift (Fig. 7), and finally right shift (Fig. 8). For each operation, we always perform the same number of computations, in order to compare the differences in execution time. In the

[5]Being able to rely on a well-established library like GMP, both for performance and correctness, greatly facilitates the debugging of a new library such as the one provided by our Ω language.

[6]The $\mathbb{Z}_{64}$ type in the Ω language corresponds exactly to the `int64_t` type in the C language, which is often equivalent to `long` on most architectures.

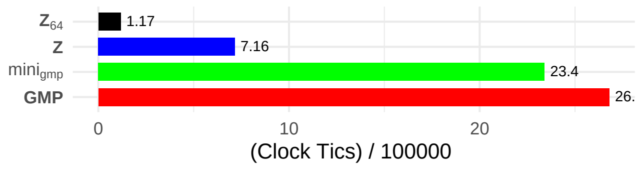


Figure 3. Addition of Small Values (63 bits).

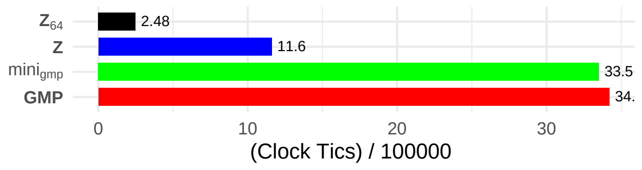


Figure 4. Substraction of Small Values (63 bits).

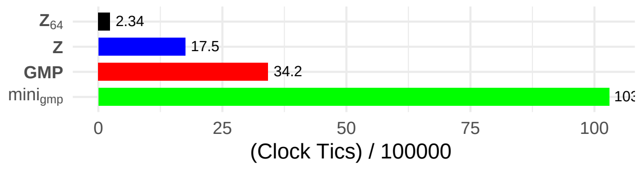


Figure 5. Multiplication of Small Values (63 bits).

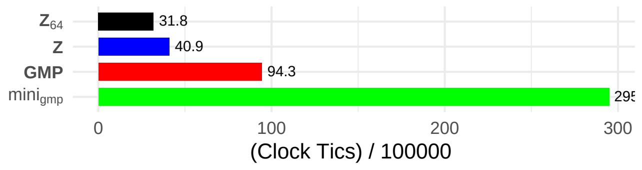


Figure 6. Division of Small Values (63 bits).

graphs below, and to facilitate comparison, we consistently assign the same color to each of the types used.

The first test, illustrated in Figure 3, focuses on addition for all types, including the $\mathbb{Z}_{64}$ type, which is naturally the fastest. It is important to note that the $\mathbb{Z}_{64}$ type does not check for potential overflows, which also explains its high performance.

Somewhat disappointingly, GMP performs almost identically to *mini*GMP for these very small values, which is rather unexpected or at least unusual. In contrast, the $\mathbb{Z}$ type stands out for its performance, ranking second after $\mathbb{Z}_{64}$, but clearly ahead of both GMP and *mini*GMP. For reference, this test consists of performing 500 million additions. All tests are executed on the same computer, fully dedicated to these benchmarks. Each test is repeated 10 times for each type, and the reported time corresponds to the minimum number of clock cycles measured.

The test shown in Figure 4 focuses on subtraction and was carried out under the same conditions as before: on the same machine, with 500 million operations as well. Unsurprisingly, the results are similar to those observed for addition, with

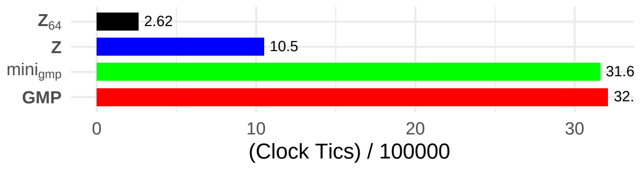


Figure 7. Left shift of Small Values (63 bits).

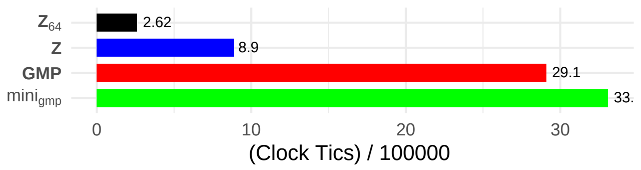


Figure 8. Right shift of Small Values (63 bits).

the $\mathbb{Z}_{64}$ type remaining by far the fastest. It is worth noting, however, that for all tested types, subtraction consistently takes more time than addition. For the $\mathbb{Z}$, GMP, and *mini*GMP types, this can be explained by the need to explicitly handle potential overflows. In contrast, for the $\mathbb{Z}_{64}$ type, this performance gap is more surprising, since addition and subtraction are normally handled by the same hardware circuit using two's complement logic.

Figure 5 shows the comparison of multiplication performance for all types, including $\mathbb{Z}_{64}$, which again proves to be the fastest. For the $\mathbb{Z}_{64}$ type, it is worth noting that the execution time for multiplication remains fairly close to the reference time, which is that of addition. For the other types, computation time deteriorates noticeably in the case of multiplication. In this context, GMP clearly outperforms *mini*GMP.

Figure 6 presents the comparison of division performance. Here again, and not unexpectedly, the ranking remains the same, although the gap between GMP and *mini*GMP is particularly pronounced. Once more, the performance difference between $\mathbb{Z}$ and GMP remains significant.

To conclude this test on small values, Figures 7 and 8 show the results for left and right bit shifts. Here again, since we are dealing with small values, the $\mathbb{Z}_{64}$ and $\mathbb{Z}$ types are clearly faster.

6.2 Comparing to a Small Compile-Time Constant

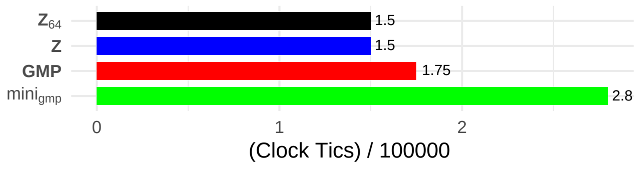


Figure 9. Testing equality with zero for a general value.

In numerical computations, comparing a value to a small fixed constant—especially to zero—is a common operation.

The representation used for the $\mathbb{Z}$ type encodes the value zero by setting all bits to zero, exactly as the native $\mathbb{Z}_{64}$ type does, i.e., following the processor's native representation. As a result, the C code used to test for equality with zero is strictly identical for both $\mathbb{Z}$ and $\mathbb{Z}_{64}$ types. Given the frequency and importance of this operation, the GMP library provides a dedicated function: `mpz_sgn`. Figure 9 shows the execution times for 500 million zero comparisons, performed on the same machine as in the previous experiments. We observe that the `mpz_sgn` implementation in GMP delivers excellent performance, very close to that of the native type. The result obtained with *mini*GMP is slightly slower, but still very reasonable in the case of a comparison with zero.

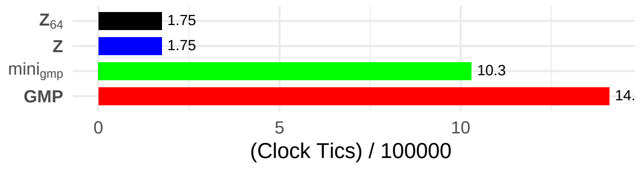


Figure 10. Testing equality with a small nonzero value.

For comparisons with a small constant value, different from zero but known statically, the representation chosen for the $\mathbb{Z}$ type also enables performance identical to that of the native $\mathbb{Z}_{64}$ type. For example, when comparing with the value 2, the Ω compiler can directly generate a comparison instruction using the left-shifted value 4, which corresponds to the internal representation of 2 in the $\mathbb{Z}$ type. As a result, the operation remains as efficient as with a native integer. In contrast, such an optimization is not possible with the GMP library. It requires the use of the `mpz_cmp` function, which is more general in behavior but also more costly in terms of performance. Figure 10 shows the execution times for this type of comparison.

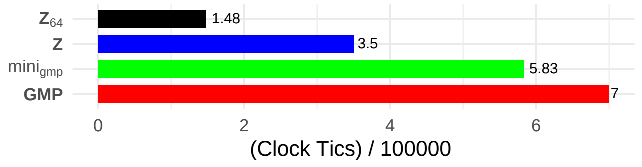


Figure 11. Comparison test: large vs. small value.

In the case of a comparison (<, >, ≤, ≥) between a large number and a small one, the representation adopted for the $\mathbb{Z}$ type also proves to be highly efficient, as illustrated in Figure 11.

6.3 Benchmarks Using Large Values (256 bits)

In this section, we repeat most of the tests previously performed (i.e., all except the specific case of comparisons), using operands consisting of 4 words on a 64-bit architecture, that is, signed integers requiring 256 bits. To allow comparison

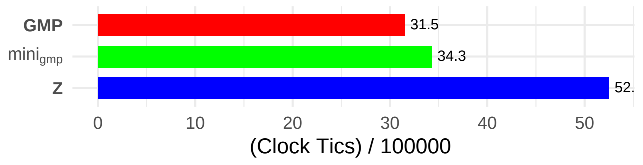


Figure 12. Addition of Large Values (256 bits).

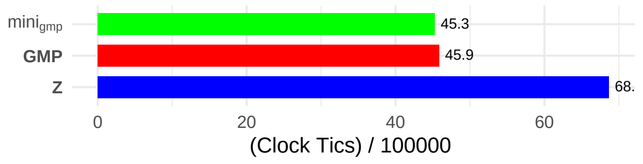


Figure 13. Substraction of Large Values (256 bits).

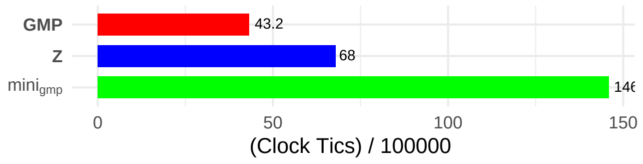


Figure 14. Multiplications of Large Values (256 bits).

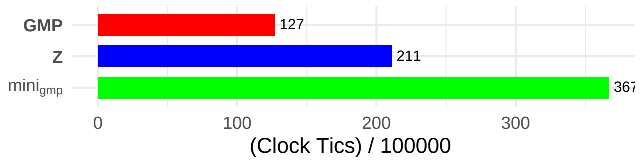


Figure 15. Division of Large Values (256 bits).

with the operands used in the previous section, we perform the same number of operations, namely 500 million, under the same conditions and on the same machine. Naturally, the $\mathbb{Z}_{64}$ type can no longer be used, as it is inherently limited to 64-bit integers.

Figures 12 to 17 show the results we obtained. This time, and not too surprisingly — though without the difference becoming dramatic — the situation is reversed. As expected, GMP achieves the best overall performance. However, it is worth noting that in the case of subtraction (figure 13), somewhat surprisingly, *mini*GMP performs on par with GMP.

6.4 Benchmarks Using Huge Values (2^{20} bits)

In this section, we revisit the previous benchmarks using values that can be considered quite large, specifically operands of 2^{20} bits[7].

[7]GMP allows computations on integers of virtually arbitrary size, limited only by the available memory. On 64-bit platforms, the theoretical limit is around 2^{37} bits (approximately 137 billion bits, or 17 GB of data), due to the internal use of a 32-bit signed integer to represent the number of limbs.

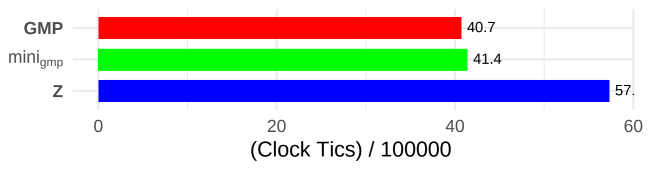


Figure 16. Shift left of Large Values (256 bits).

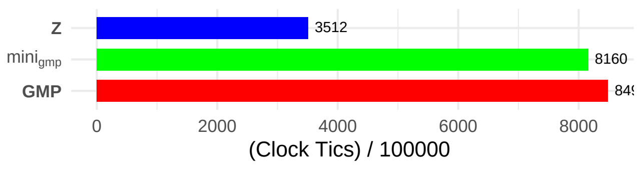


Figure 20. Multiplication of Huge Values (2^{20} bits).

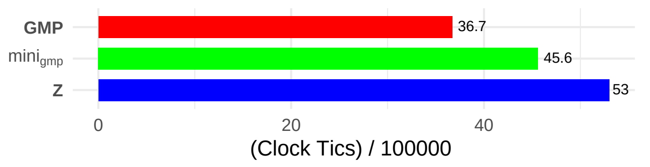


Figure 17. Shift Right of Large Values (256 bits).

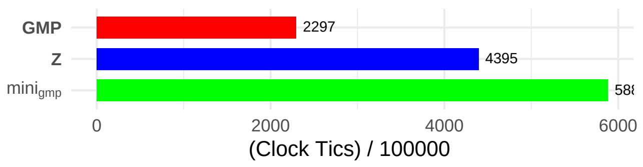


Figure 21. Division of Huge Values (2^{20} bits).

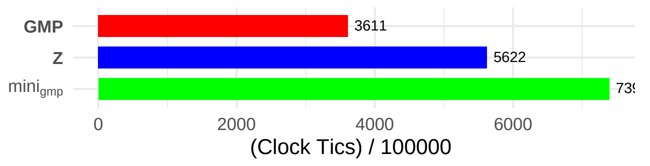


Figure 18. Addition of Huge Values (2^{20} bits).

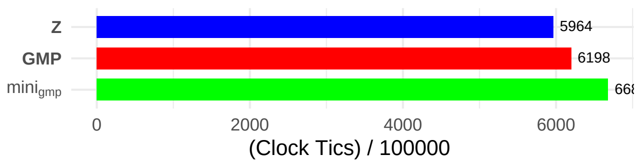


Figure 22. Shift Left of Huge Values (2^{20} bits).

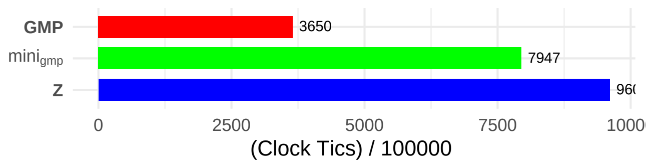


Figure 19. Substraction of Huge Values (2^{20} bits).

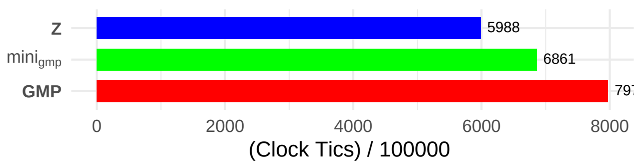


Figure 23. Shift Right of Huge Values (2^{20} bits).

For such values, and as expected, GMP demonstrates its full power, especially for fundamental operations such as addition (Fig.18) and subtraction (Fig.19).

As in previous experiments, each test involves 500 million operations and is run on the same machine. This consistency allows us to observe how computation time evolves with operand size. For instance, in the case of addition: small values (Fig.3), large values (Fig.12), and extremely large values (Fig. 18).

Interestingly, for certain operations, the $\mathbb{Z}$ type is able to match or even outperform the excellent performance of GMP. This is particularly evident in shift operations (Figs. 22 and 23). As for multiplication (Fig. 20), the results suggest a possible performance issue in GMP, as even *mini*GMP performs slightly better in this case.

6.5 Comparison with other programming languages.

In this section, we compare the four types previously evaluated —$\mathbb{Z}_{64}$, $\mathbb{Z}$, GMP, and *mini*GMP— with other languages that, when possible, support a native functional notation. The benchmarking protocol is identical across all languages.

It consists in running a command-line program with two parameters: the first specifies the number of iterations, and the second the increment value. The program follows a simple logic: starting from an initial value of 0, it performs repeated additions by the given increment, as many times as specified by the first parameter. Finally, it prints the resulting value to ensure that the computation is actually performed. This safeguard is necessary, as some compilers apply aggressive optimizations and might eliminate computations whose results are not used.

Moreover, passing parameters via the command line prevents any compile-time evaluation by the compiler. Our goal is to assess the execution speed of arithmetic operations themselves, not the ability of compilers to optimize away redundant calculations.

While preparing this comparison test, we observed that for all the modern languages we tested, execution time deteriorates rapidly as the size of the manipulated numbers increases—sometimes drastically so. In fact, as soon as the resulting number reaches or exceeds 512 bits in size, GMP significantly outperforms every other language we evaluated.

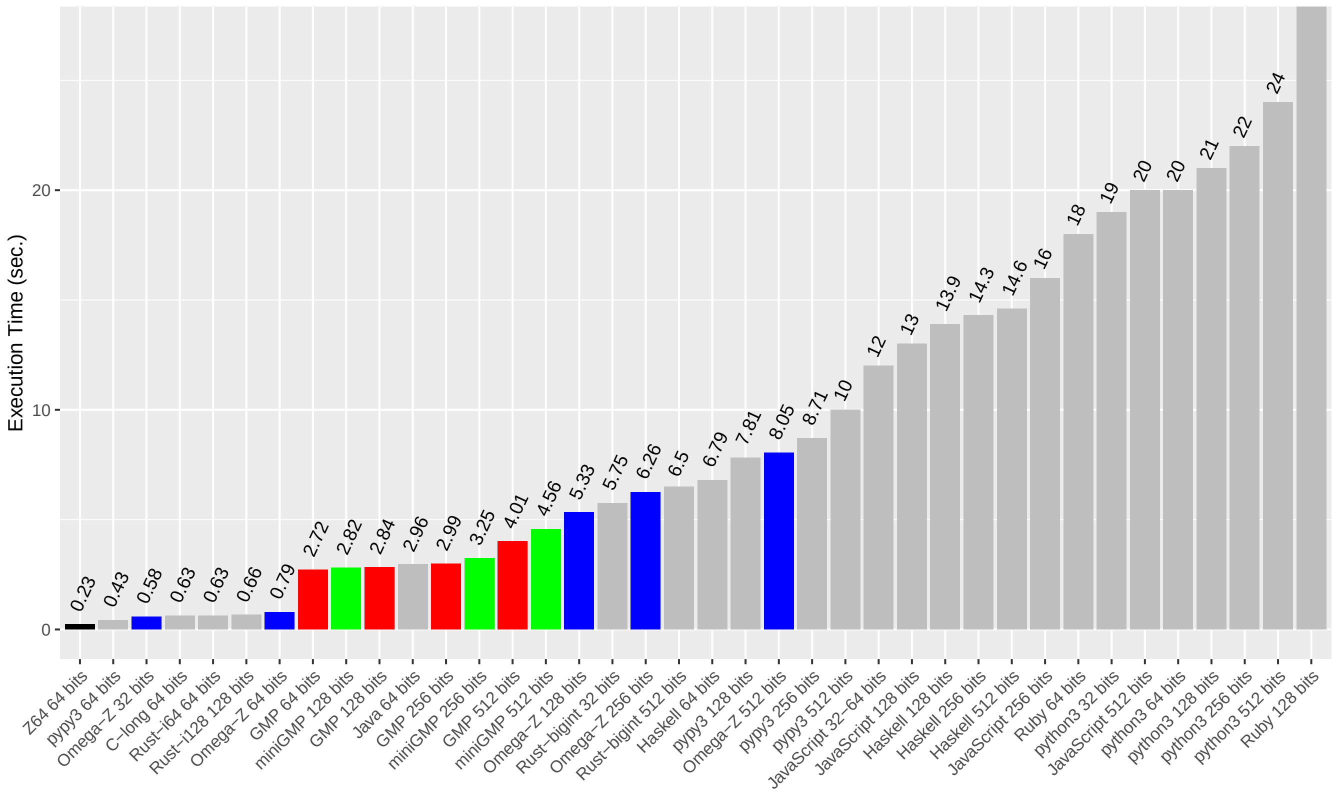


Figure 24. Comparison with other programming languages.

This confirms that GMP remains the most efficient library for high-precision integer arithmetic[8].

Therefore, for each language, we performed three tests by varying only the increment value. The number of iterations passed as an argument remained unchanged from the previous benchmarks—i.e., 500 million additions.

The first test stops at a result that fits within 32 bits. The second reaches a 64-bit value, and the third one goes up to a 120-bit value. The results are shown in Figure 24. The color scheme for the bars remains consistent for our four types, while the new languages are represented in gray.

7 Conclusion

Since the advent of 64-bit architectures, it has been well understood that they lead to a mechanical increase in the size of executables and data structures compared to 32-bit architectures. Part of this increase stems from memory alignment constraints, while another part is due to the size of pointers, which are stored on 64 bits even though they effectively use only 48 significant bits.

Moreover, since pointers are generally aligned to word boundaries, the last three bits of a 64-bit word address are consistently unused. Overall, between 16 and 19 bits are wasted for every pointer stored in main memory.

This observation has motivated the design of two compact representations for memory information. The first, described in Section 4, is designed for a hardware-oriented implementation and allows storing the address of a memory region in just 45 bits. The second, presented in Section 5, is more suitable for a software implementation. This latter approach is the one used in our performance measurements.

The advantage of this representation is that it enables storing either a small value or a pointer to an auxiliary memory region within a single machine word, without requiring additional external memory. This not only allows for a uniform and compact representation of data structures such as number matrices, but also enables a natural functional notation for our Ω language [1].

In this paper, we have proposed an implementation that preserves a functional notation while maintaining precision in computations involving signed integers. It leverages the fact that memory addresses are effectively limited to 48 bits on 64-bit architectures.

[8]As mentioned in [13], the general-purpose arbitrary-precision integer library GMP is often faster than specialized cryptographic libraries such as OpenSSL or MIRACL. Also cited in [13]: "Software like Crypto++, BorZoi, or OpenSSL is tailored for cryptography, but a general-purpose arbitrary-precision library like GMP is often superior in terms of performance (see for example http://gmplib.org/32vs64.html)."

Compared to GMP, when working with small values, our approach provides a significant performance gain. In addition, overflow issues become a low-level concern, often transparent to most programmers.

Finally, this optimization could also benefit procedural libraries such as GMP, which could adopt this approach to replace their current implementation. Performance on very large numbers would remain unaffected, while computations involving small values would be considerably faster. In particular, when numerical computations operate on values smaller than 2^{61}, performance improvements become clearly measurable, as shown by the benchmarks presented in this paper. Performance gains are also observed in equality comparisons involving small values, where this technique achieves the best results (see Fig. 10).

References

[1] Benoit Sonntag and Dominique Colnet. Omega: The power of visual simplicity. In *Future of Information and Communication Conference, April 2025, Berlin, 25 pages.*, 2025.

[2] Zilog, Inc. *Z80 CPU User Manual.* Zilog, Inc., um008011-0816 edition, 2016.

[3] Amstrad Consumer Electronics plc. *Amstrad CPC 6128 User Instructions.* Amstrad, 1985.

[4] Intel Corporation. *Intel 8086 Family User's Manual: Programmer's Reference.* Intel, 1979.

[5] Intel Corporation. *Intel 80386 Programmer's Reference Manual.* Intel, 1986.

[6] Intel Corporation. *Intel®64 and IA-32 Architectures Software Developer's Manual.* Intel, 2025.

[7] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, and Samuel Z. Guyer. Tracking bad apples: Reporting the origin of null and undefined value errors. In *In 22th Annual ACM Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'2007)*, pages 405–422. ACM Press, 2007.

[8] Steve Klabnik, Carol Nichols, and Chris Krycho. *The Rust Programming Language.* No Starch Press, San Francisco, CA, 2nd edition, 2023.

[9] Python Software Foundation. *Python Language Reference.* Python Software Foundation, 2025.

[10] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, 1983. ISBN 0201113716, 1366 pages.

[11] Benoit Sonntag and Dominique Colnet. Rip linked list – empirical study to discourage you from using linked lists any further. *International Journal of Future Computer and Communication*, 2024.

[12] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library.* Free Software Foundation, 2020. Version 6.3.0.

[13] Jean-Guillaume Dumas. *Contributions au calcul exact intensif.* Accreditation to supervise research, Université de Grenoble, July 2010.

[14] Fredrik Johansson. Faster arbitrary-precision dot product and matrix multiplication, 2019.

[15] Torbjörn Granlund and the GMP development team. mini-gmp: A minimalistic implementation of a subset of the gmp library, 2020. Included in the GMP source distribution.