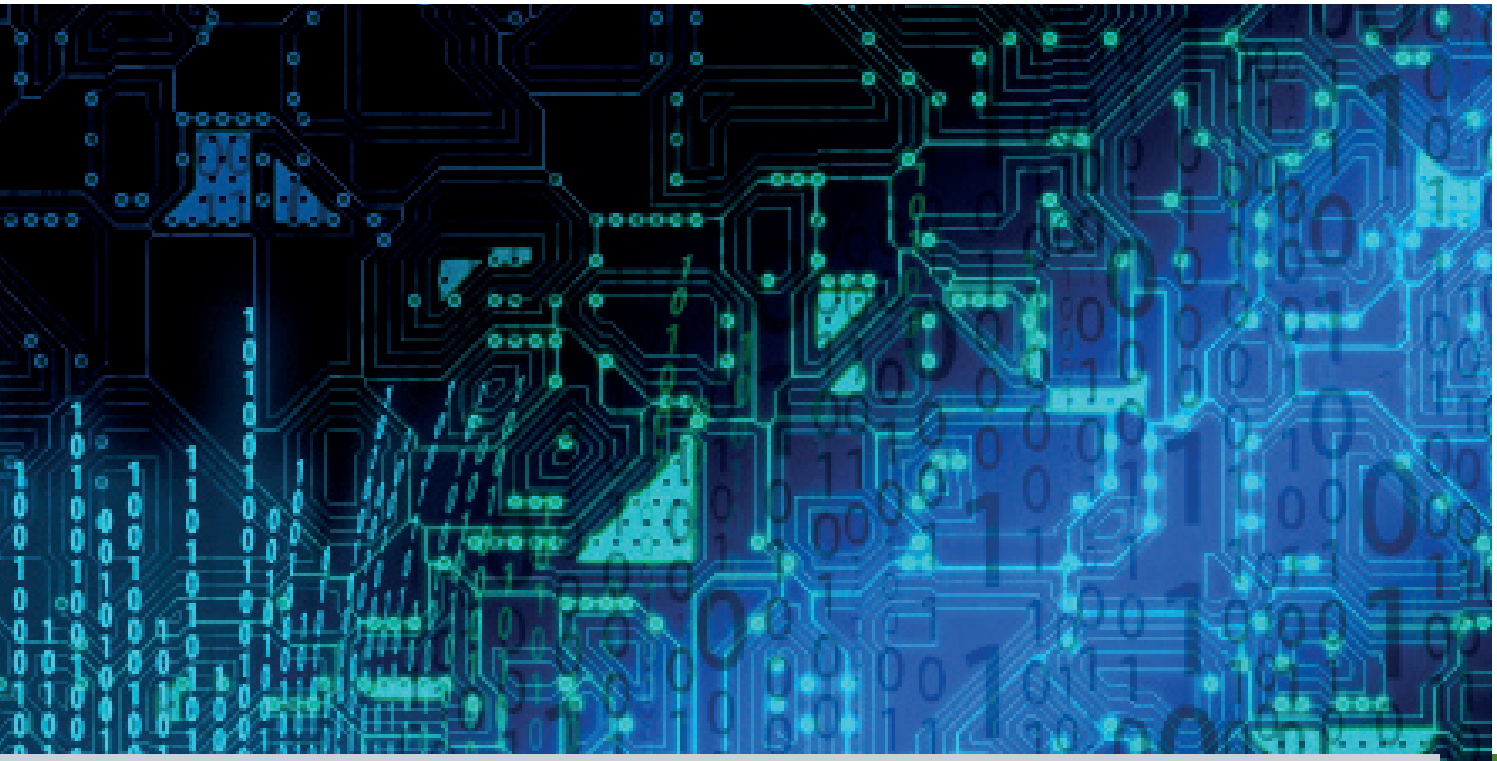
DHBW
Duale Hochschule
Baden-Württemberg
Stuttgart

Engineering

INSIGHTS



Schriftenreihe der Fakultät Technik: 1/2025

Tagungsband des Jahrestreffens 2025 der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“

Daniel Holle, Prof. Dr. Jens Knoop, Prof. Dr. habil. Martin Plümicke, Prof. Dr. Peter Thiemann,
Prof. Dr. habil. Baltasar Trancón y Widemann (Hrsg.)

Inhaltsverzeichnis

Multi-precision integer arithmetics

M. Anton Ertl
TU Wien
anton@mips.complang.tuwien.ac.at

Abstract

Multi-precision integer arithmetics is widely used, among other things in public-key cryptography and when computing many digits of transcendental numbers. The present paper discusses multi-precision addition and multiplication: architectural support and its use in hand-written assembly language, libraries that use such assembly-language code, and programming language support and how close the code generated by Clang and GCC is to the hand-written assembly language.

1 Introduction

Integer arithmetics on numbers that fit in a single machine word is single-precision arithmetics. With two machine words, it's double-precision arithmetics.[1]. With more machine words, it's multi-precision arithmetics.

In the present work, I focus on multi-precision addition and multi-precision multiplication. Subtraction can be derived from addition, while division would merit a paper of its own. I also focus on unsigned arithmetic; for the common twos-complement representation of signed numbers signed addition and multiplication is the same (addition) or very similar (multiplication).

This paper is a first try at overview of uses and implementation techniques for efficient multi-precision integer arithmetics. The relevance of the topic can be seen from the use of multi-precision arithmetics in software (Section 2). Section 3 introduces the mathematical foundation, concepts, and notation for multi-precision arithmetics. Section 4 explains how various computer architectures support multi-precision arithmetics. This architectural support can be used to write routines in assembly language (Section 6) and provide them to application programs through libraries (Section 5), or programming language compilers can make use of these features and provide language support for them (Section 7).

[1] Single and double precision are more commonly used in connection with floating-point, but the concept also applies to integer arithmetics

2 Uses

How relevant is multi-precision arithmetic? In particular, how relevant is the performance of multi-precision arithmetic?

We can try to answer this question by looking at uses of the GMP (GNU multi-precision) library. E.g., 1839 packages (out of 65523, i.e., 2.8%) in Debian 12 depend on GMP. Unfortunately, that does not tell us anything about how relevant the performance of multiprecision arithmetics is for these packages. The GMP implementors invest a lot of effort into making GMP efficient; e.g., GMP 6.3.0 contains 155,715 lines in 930 files of assembly language code. This makes it plausible that multi-precision arithmetic in general and GMP in particular are used in applications where performance matters.

There are two applications dmains that I know where multi-precision arithmentics is used and where performance matters:

- Public-key cryptography uses arithmetic with sizes of 256–4096 bits in pre-quantum cryptography.[2]
- Computing many digits of transcendental constants such as γ or π, which involve large multiplications. The numbers involved in these computations are bigger than the caches, so memory bandwidth becomes an issue; nevertheless, computing is also an issue.[3] A well-known program for this purpose is y-cruncher.

Another use is in implementing the integers of higher-level languages, which often support arbitrary size. In most programs this support is hardly used and is only a safety net in case the computation overflows a single-precision integer, but programs can also use this support as a convenient way of performing computations with big numbers.

3 Basics

We can use the same algorithms for synthesizing multi-precision arithmetics out of single and double-precision arithmetics as has been used for multi-digit arithmetics, so we use the same notation: A machine word can hold $b = 2^{\text{bits}}$ values: $0...b-1$. Single-precision numbers typically start with s, double-precision numbers with d, and multi-precision numbers with m. Double-precision numbers are represented with two machine words $d = s_1 b + s_0$, multi-precision numbers with l machine words $m = s_{l-1}b^{l-1} + ...s_0$. Each s_i is called a *limb* of m. In code fragments that deal with a single limb of a multiprecision number mx, we call that limb as `mx`, too.

3.1 Addition

When adding n machine words, the result r may be $\geq b$, but $r < nb$. So r can be represented as $r = cb + s$, where s is the part of r that fits in a machine word and $c < n$ is the *carry*.

Multi-precision addition is performed starting from the least significant limbs and progresses towards more significant limbs. After the first step, the carry has to be added in addition to the limbs of the summands (*carry-in*): $r_i = \mathsf{s1}_i + ... + \mathsf{sn}_i + c_{i-1}$. For the carry-out of such a step, $c < n$ still holds.

3.2 Multiplication

When we multiply two single-precision numbers, the result has values of up to $b^2 - 2b + 1$ and fits in a double-precision number. This widening multiplication is often a primitive in implementing multi-precision arithmetics, and we will use it as such here.

For multi-precision multiplication ef, with e having limbs $e_{l-1}...e_0$ and f having limbs $f_{n-1}...f_0$, the result is

$$r = \sum_{i=0}^{l-1} \sum_{j=0}^{n-1} e_i f_j b^{i+j}$$

Here is an example showing the terms of a multiplication with $l = 4, n = 3$:

$$\begin{array}{cccccc} & & e_3f_0 & e_2f_0 & e_1f_0 & e_0f_0 \\ & e_3f_1 & e_2f_1 & e_1f_1 & e_0f_1 & \\ e_3f_2 & e_2f_2 & e_1f_2 & e_0f_2 & & \end{array}$$

All the terms in one column are for the same $i + j$. This table looks like the usual way long multiplication is presented, but note that each array entry is a double-precision result of a multiplication, unaffected by surrounding multiplications, whereas the usual long multiplication algorithm shows, in each line, the digits coming out of the more complex operation ef_i. So all the widening single-precision multiplications are independent of each other.

For computing a single limb r_k, we have to sum up a number of terms, and there is also a carry-out of this computation:

$$c_k b + r_k = \left(\sum_i (e_i f_{k-i}) \bmod b\right) + \left(\sum_i (e_i f_{k-i-1})/b\right) + c_{k-1}$$

The addition for computing a single limb can have up to $l + n + 1$ terms, with the carry being correspondingly large.

However, with different primitives, some of the additions can be integrated with the multiplications,

[2] Post-quantum cryptography often uses larger keys, but I do not know which of those approaches use multi-precision arithmetics. In any case, the keys still fit in the caches of modern processors.

[3] `https://www.numberworld.org/y-cruncher/faq.html#gpu` discusses some of the issues.

which can reduce the later summing. In particular, one can add two single-precision numbers to the double-precision result of a widening multiplication without overflowing the double-precision number: the maximum result of this operation is the maximum double number:

$$(b-1)*(b-1)+2(b-1) = b^2-2b+1+2b-2 = b^2-1$$

4 Architectural support

This section discusses the architectural support that existing architectures have for multi-precision arithmetics and that one could add to architectures to improve that support:

4.1 Addition

The most common support for multi-precision addition is in the form of carry flags. Many architectures have one carry flag C. On most such architectures the common addition instruction only sets carry-out, and there is an additional add-with-carry instruction that adds carry-in as well as producing carry-out, typically with a latency of one cycle.

E.g., AMD64 (without ADX) has one carry flag, and one can perform addition of two n-word multi-precision numbers `m = m1+m2`:

```
L: movq (m1,i,8),tmp
   adcq (m2,i,8),tmp
   movq tmp, (m,i,8)
   incq i
   decq n
   jnz  L
```

Note that this loop was carefully written with instructions that do not trample over the carry flag (many AMD64 instructions do).

ARM A64 also has one carry flag, but both the carry-in and flag-setting (including carry-out) is optional:

writes flags			
no	yes		
`add`	`adds`	no	carry-in
`adc`	`adcs`	yes	

Compilers often have problems dealing with a single carry flag, and while you can ask the compiler to use add with carry-in and carry-out, even when the compiler uses the appropriate instruction, the code around it is not so great (see Section 7.2.2).

The Intel ADX extension allows using the O(verflow) flag as a second carry flag in an instruction that uses the O flag as both carry-in and carry-out. The motivation for adding ADX seems to have been particularly for multi-precision multiplication [2]. A simple example of its use is to add three n-word numbers `m = m1+m2+m3`:

```
L: movq (m1,i,8),tmp
   adcxq (m2,i,8),tmp
   adoxq (m3,i,8),tmp
   movq tmp, (m,i,8)
   incq i
   decq n
   jnz  L
```

Here each addition chain has its own carry bit: `tmp=m1+m2` uses C, `tmp=m3+tmp` uses O.

A number of architectures have no carry flag, among them RISC-V. They typically require a five-instruction sequence with a minimum latency of three cycles to replace one add with carry-in and carry-out instruction. The addition `m = m1+m2` looks as follows on RISC-V:

```
L: ld   summand1, 0(m1p)
   ld   summand2, 0(m2p)
   add  tmp1, carry, summand1
   sltu carry1, tmp1, summand1
   add  sum, tmp1, summand2
   sltu carry2, sum, tmp1
   add  carry, carry2, carry1
   sd   sum, 0(mp)
   addi n, n, -1
   addi mp, mp, 8
   addi m2p, m2p, 8
   addi m1p, m1p, 8
   bnez n, L
```

Finally, in yet-unpublished work [1] I propose adding a carry bit (and another bit for indicating signed overflow) to every general-purpose register. These bits should be easier to manage in a compiler than the traditional carry flag, and they allow having more than one addition chain active at once.

```
L: ld   summand1, 0(m1p)
   ld   summand2, 0(m2p)
   add    tmp1, summand1, summand2
   addc   sum, tmp1, sum
   sd   sum, 0(mp)
   addi   n, n, -1
   addi mp, mp, 8
   addi m2p, m2p, 8
   addi m1p, m1p, 8
   bnez n, L
```

The proposal follows the typical RISC-V style of having instructions with two source and one destination register, and therefore splits the computation `sum = summand1 + summand2 + carry(sum)` into two instructions: the `add` adds summand1 and summand2, and `addc` adds the carry from the previous iteration (in the carry bit of `sum`) to `tmp1`. Note that the `add` is not in the dependency cycle from the last iteration, that involves only the `addc`, which I expect to take one cycle of latency.

4.2 Multiplication

Most architectures support widening multiplication, either in one instruction with a double-word result or in one instruction for the lower word of the result, and one instruction for the upper word of the result.[4]

An instruction that computes $d = s_a s_b + s_c$ (integer multiply-add) would be useful, but I am not aware that any architecture has it. ARM A64 has `madd`, an instruction that computes the bottom word of d, but the corresponding instruction `umaddh` for computing the most significant word of d is missing. However, we will show how such an instruction could be used (see Section 6.3).

ARM A64 also has `umaddl`, which performs $s = h_a h_b + s_c$, where h indicates a 32-bit value. Unfortunately, this instruction is not useful for multi-precision arithmetics, because one needs 4 widening 32-bit multiplications plus additional work to replace one widening 64-bit multiplication.

I also explored the option of having an instruction `m2add` that computes $d = s_a s_b + s_x + s_y$ or `madc` that computes $d = s_a s_b + (c_z b + s_z)$ (so that an `add` followed by an `madc` would be equivalent to `m2add`), as well as upper/lower instruction-pair variants of these instructions. However, instructions are usually implemented in out-of-order execution engines to wait for all operands; so such instructions would result in carry-to-carry latencies that include the latency of the multiplication, which may be undesirable.

[4] Some architectures have additional instructions for multiplication where one or both operands are signed, but these are outside the scope of this paper.

5 Libraries

The programming language support for high-performance multi-precision arithmetics has not been satisfactory, so programmers have developed libraries for this purpose. Wikipedia lists 36 libraries that deal with arbitrary-precision integers[5].

The most well-known among them is GMP, which has first been released in 1991. It contains a lot of functions for various purposes. In the present context, the low-level functions (`mpn_`...[6]) are the most interesting ones, and among them these functions.

- $m = m_1 + m_2$: `mpn_add_n(m,m1,m2,n)`
- $m = sm_1$: `mpn_mul_1(m,m1,n,s)`
- $m = m + sm_1$: `mpn_addmul_1(m,m1,n,s)`
- $m = m_1 m_2$: `mpn_mul(m,m1,n1,m2,n2)`

These functions are usually written in assembly language, with the central code similar to what is shown in Section 6, but with lots of macros and parameterization for unrolling and to generate code for similar functions, and are therefore not as easy to understand as one might like.

We take a closer look at how to implement the first and third of these functions. The third is a better stepping stone for implementing long multiplication (`mpn_mul`) than the second, and it also illustrates some implementation problems and how to deal with them.

A disadvantage of libraries is that you can combine their functions by calling one after the other, but the sequence cannot be optimized. It will always cost as much as the two individual calls. E.g., while it is possible to implement $m = m+sm_1$ by first calling `mpn_mul_1` and then `mpn_add_n`, GMP provides `mpn_addmul_1` as a separate function, in order to support more efficient implementations.

[5] https://en.wikipedia.org/w/index.php?title=List_of_arbitrary-precision_arithmetic_software&oldid=1305070005#Libraries

[6] https://gmplib.org/manual/Low_002dlevel-Functions

6 Hand-written assembler code

In the following we take a look at hand-written (hopefully optimal) code for the central part of one limb of three computations: $m = m_a + m_b$, $m = m_a + m_b + m_c$, $m = sm_a + m_b$. This provides a reference for comparing with the compiler-generated code we see in Section 7.

The first and third computation directly correspond to `mpn_add_n()` and `mpn_addmul_1()`. The three-summand addition is instructive because it shows the limits of having a single carry flag. It can also be seen as a simplified variant of a problem we have in $m = sm_a + m_b$; while it looks like there are only two summands here, we actually have to add, in each iteration, three words: the upper word of sm_a from the previous iteration, the lower word of sm_a from the present iteration, and m_b.

Central part means the computation without loading the source limbs and storing the the result limb, and without loop overhead. We show assembly language code for RISC-V, ARM A64, and AMD64. On RISC-V and ARM A64, the destination register is leftmost, on AMD64 rightmost (AT&T syntax). In this section we use variable names instead of registers, to make the code easier to follow.

The latency numbers given are based on realistic assumptions about the timing of the instructions in a core with wide out-of-order execution; in particular, `add`, `adcs` etc. have one-cycle latencies, and `mov` has a zero-cycle[7] latency.

6.1 Two-summand addition

For $m = m_a + m_b$:

```
RISC-V:
add sum, summand1, summand2
sltu carry1, sum, summand1
add sum, sum, carry
sltu carry2, sum, carry
add carry, carry1, carry2

ARM A64:
adcs sum, summand1, summand2

AMD64:
#summand1 is in sum
adc summand1, sum
```

[7] In many recent cores with out-of-order execution, `mov` is usually performed by register renaming

On ARM A64 and AMD64, carry is in the C flag before this and the new value of carry is in the C flag afterwards. The RISC-V code has three cycles of carry-to-carry latency, the ARM A64 and AMD64 codes have 1 cycle of carry-to-carry latency on typical implementations of these architectures.

6.1.1 Kogge-Stone addition

Alexander Yee has implemented Kogge-Stone addition (normally a hardware adder design technique) in AVX-512.[8] This is pretty far from anything discussed in this paper, so the rest of this paper ignores this possibility. However, Yee reports that his AVX512-F code "is about 2x faster than the classic approach of chaining adc instructions on Skylake X.". So we should look at whether SIMD instruction sets can be more profitably used to achieve the best performance before investing much time in tuning the performance of carry-flag-based approaches.

6.1.2 Breaking dependencies

Two-summand and three-summand addition with chains of carries can be latency-bound on wide cores, with the loop-carried dependencies through carry forming the critical path. Michael S presents[9] and evaluates[10] a technique for converting a data dependency into an extremely predictable control dependency, which eliminates the dependency in case of a correct prediction (i.e., almost always).

```
   xorq carry, carry
L: movq (ma,i,8),sum
   xorq carry2, carry2
   addq (mb,i,8),sum
   setc carry2
   addq carry, sum
   jc   C
M: movq carry2, carry
   movq sum, (m,i,8)
   incq i
   decq n
   jnz  L
   ... finish ...
   ret
C: incq carry2
   jmp  M
```

[8] `http://www.numberworld.org/y-cruncher/internals/addition.html`

[9] `news:20250805014356.0000073c@yahoo.com`

[10] `news:20250806204333.00006869@yahoo.com`

This code starts a new dependecy chain by clearing carry2 at the start, which is then set to the carry of $\mathrm{ma}_i + \mathrm{mb}_i$. The old carry is then added to sum, and there is a low probability that this addition has a carry, which would need to be added to carry2. The carry-out of this addition is not added to carry2 in a data-dependent way (e.g., with `adc`) to avoid producing a data dependency cycle. Instead, the carry-out is checked with a conditional branch, and on correct prediction (the usual case for this very predictable branch) iteration $i+2$ uses nothing from iteration i except loop counter updates, which can be reduced by unrolling.

While this loop reduces the dependence chains through carry, one iteration contains a lot of instructions, and therefore this code is likely to take more than one cycle per output word despite the reduction in dependencies. One way to counteract this may be to unroll the loop by a factor of n, and do the first $n-1$ original iterations with `adc` and only the last using the conditional branch, in order to achieve a better balance between general resource consumption and dependencies.

6.2 Three-summand addition

For $m = m_a + m_b + m_c$, things become more interesting:

```
RISC-V:
add m, ma, mb
sltu carry1, m, ma
add m, m, mc
sltu carry2, m, mc
add m, m, carry
sltu carry3, m, carry
add carry, carry1, carry2
add carry, carry, carry3

RISC-V with carry in GPRs:
# carry = carry(m)+carry(mab)
add  tmp, ma, mb
addc mab, tmp, mab
add  tmp, mab, mc
addc m, tmp, m

ARM A64:
# carry = carry1+C
adcs m1, ma, carry1
adc carry1, xzr, xzr
adds m2, mb, mc
adc  carry1, carry1, xzr
adds m, m1, m2

AMD64:
# carry = carry1+C
adc ma, mb
setb carry2
add $-1, carry1
adc mc, mb
mov carry2, carry1
#sum in mb

AMD64 with ADX:
#ma in m
adcx mb, m
adox mc, m
```

The RISC-V, ARM A64, and AMD64 code has three cycles of carry-to-carry latency, the ADX code has one cycle. For the AMD64 code the `setb` may not break the dependency on the earlier contents of `carry1`, and it may be useful to insert a `movl $0, carry1` before the `setb`.

In the ARM A64 variant, carry1 can have values 0–2, but the complete carry is the sum of carry1 (in a general-purpose register) and the C flag; the sum can only have values 0–2. In the AMD64 variant, carry1 can only be 0–1. One could use the approach used for AMD64 for ARM A64 and vice versa. For generalizing to more summands, the ARM A64 variant may be better.

Another approach for implementing three-summand addition is to combine two two-summand additions, but doing this naively would increase the number of stores and loads, and the loop overhead. Instead, one can unroll the loop to do two two-summand additions while the intermediate result fits in registers, and, at the unrolling boundaries, use the carry-handling approach used by the AMD64 code above:

```
ARM A64:
#carry = carry1+c
#minus1=-1
adcs mab0, ma0, mb0
adcs mab1, ma1, mb1
adcs mab2, ma2, mb2
adcs mab3, ma3, mb3
adc  carry2, xzr, xzr
adds xzr, carry1, minus1
adcs sum0, mab0, mc0
adcs sum1, mab1, mc1
adcs sum2, mab2, mc2
adcs sum3, mab3, mc3
mov  carry1, carry2
```

For an unrolling factor of n, this approach has a latency of $n+2$ (if the CPU can perform two `adcs` per cycle). If the loads are arranged to be as late as possible and the stores to be as early as possible, the number of registers needed is $n+3$ ($n+4$ when using load-pair and store-pair), plus another 4 for addresses and loop control. The same approach can also be used on AMD64, with similar code, latency, and register requirements.

6.3 Multiplication step

For $m = sm_a + m_b$:

```
RISC-V:
mulhu tmp1, s, ma
mul   sma, s, ma
add   tmp2, sma, mb
sltu  tmp3, tmp2, sma
add   tmp1, tmp1, tmp3
add   m, carry, tmp2
sltu  tmp4, m, carry
add   carry, tmp1, tmp4

RISC-V with carry in GPRs:
mulhu tmp1, s, ma
mul   sma, s, ma
add   tmp2, sma, mb
addc  tmp1, tmp1, tmp2
add   m, carry, tmp2
addc  carry, tmp1, m

ARM A64:
# carry = carry1+C
umulh tmp1, s, ma
mul   sma, s, ma
adcs  tmp2, sma, mb
adc   carry2, tmp1, xzr
adcs  m, tmp2, carry1
mov   carry1, carry2

ARM A64 with umaddh:
# carry = carry1+C
umaddh carry2, s, ma, mb
madd   smab, s, ma, mb
adcs   m, smab, carry1
mov    carry1, carry2

AMD64:
#rax = ma
mulq s
addq mb, rax
adcq $0, rdx
addq carry, rax
adcq $0, rdx
mov  rdx, carry

AMD64 with ADX:
#rdx = s
#carry = carry1+C+O
mulx ma, m, carry2
adcx mb, m
adox carry1, m
mov  carry2, carry1
```

While the carry can be represented in one general-purpose register, in some cases it reduced the instruction count and sometimes the latency to leave a part of the carry in the C (and, with ADX, the O) flag, so I used this in some cases.

Concerning latency, on a wide-enough processor with out-of-order execution the execution speed is determined by loop-carried dependencies. In the loop surrounding these code fragments the `carry` register (or `carry1` and the C flag) are loop-carried, so the latency from the carry coming from the last iteration to the carry passed to the next iteration is relevant. Therefore this code has been written to first perform the multiplication and the addition of m_b, and only then perform the addition of `carry`. In the code above this latency has been minimized, in some cases at the cost of additional instructions; in some cases `mov` there is a mov instruction that can be eliminated by adding the carry-in earlier if latency is not an issue.

The carry-to-carry latency is: 3 cycles for RISC-V, 2 cycles for ARM A64, AMD64 and RISC-V with carry in GPRs, 1 cycle for AMD64 with ADX and ARM A64 with `umaddh`.

In the code with upper/lower pairs, I keep the pairs together in the same order produced by compilers, in the hope that the hardware combines the halves and reduces the resource consumption of the code (for RISC-V this is recommended). This results in an additional `mov` on the ARM A64 with `umaddh`.

7 Programming language support

7.1 High-level languages

Many high-level languages provide arbitrary-precision integers (aka bignums). However, most programs deal with integers small enough to fit

in one machine word all the time, and the arbitrary decision is just a better way (than terminating the program or modulo arithmetics) to deal with cases where the result of an operation does not fit in one machine word. Moreover, in these languages multi-word integers are dynamically sized and therefore implemented boxed, with dynamic memory allocation and usually with dynamic typing; all of these implementation techniques cost performance, so one would not implement a high-performance application like y-cruncher in such a language, or at least rewrite it in a low-level language such as "C/C++ with Intel SSE intrinsics"[11] soon.

It would be possible to improve the performance of multi-precision arithmetics in these languages, by a lot, but I don't expect it to happen anytime soon, thanks to the vicious circle of it not happening due to lack of demand and the existing demand switching to lower-level languages.

7.2 Low-level languages

In particular, I look at C, but the techniques should work in other low-level languages, too. The generated code depends on the compiler, however.

The code presented below is the output of clang 14.0.6 `-march=x86-64-v4` on AMD64, and clang 11.0.1 on ARM A64 and RV64GC (RISC-V), with -Os to reduce the amount of unrolling in order to get a result that is easier to understand and present; however, the compilers tend to be better in making use of the flag during one iteration than between iterations, so unrolling generally helps in more ways than just reducing the loop overhead.

I have also looked at the output of gcc in some cases, and have not seen significantly better results, except through unrolling.

7.2.1 Double precision

64-bit integers have been supported on C implementations with 32-bit word size since around 1990. It took until around 2005 until 128-bit integers appeared in some gcc targets with 64-bit words (and then it took several years until all the operations on these integers worked correctly in gcc).

[11] https://www.numberworld.org/nagisa_runs/euler-14.9b_log2-15.5b.html

With these double-precision types available (we use `d_t` for double-precision and `s_t` for single-precision unsigned integers), we can use them to implement multi-precision. For two-summand multi-precision addition, the central loop is:

```
for (i=0; i<n; i++) {
    s_t s1=m1[i];
    s_t s2=m2[i];
    d_t d=s1+(d_t)s2+carry;
    m[i] = d;
    carry = d>>64;
}
```

The `d>>64` extracts the most significant word of `d`, and should be changed appropriately if the word size is not 64 bits.

On RISC-V the code above results in the canonical five-instruction sequence for add with carry-in and carry-out (Section 6.1) for the computation part of this loop. Unfortunately, the results on AMD64 and ARM A64 are far from optimal; for the central addition in this loop:

```
AMD64
r9=carry
xorl %eax, %eax
addq (%rsi,%r8,8), %r9
setb %al
addq (%rdx,%r8,8), %r9
adcq $0, %rax
#after storing r9:
movq %rax, %r9

ARM A64
x8=carry x9=s1 x10=s2
adds x8, x9, x8
adcs x9, xzr, xzr
adds x10, x8, x10
adcs x8, x9, xzr
```

The compilers obviously do not make optimal use of a carry flag for this code.

For three-summand addition, the C code with double precision is:

```
for (i=0; i<n; i++) {
    s_t s1=l1[i];
    s_t s2=l2[i];
    s_t s3=l3[i];
    d_t d=s1+(d_t)s2+(d_t)s3+carry;
    l[i] = d;
    carry = d>>64;
}
```

Note that `carry` can have the value 2 in this code, so in order for the compiler to use two carry flags (with ADX), it would have to represent that sum distributed over the two carry flags, and fuse the addition of each carry flag into a different one of the two addition of the three limbs. Not beyond today's compiler technology, but it would require an amount of development that apparently has not been invested yet, and probably never will. The code for two architectures is:

```
RV64GC              AMD64
#a6=carry           #r10=carry
#t1,a7,t0=s1/2/3
add  a5, t1, a6     xorl %eax, %eax
sltu a6, a5, t1     addq (%rsi,%r9,8), %r10
add  a7, a7, a5     setb %al
sltu a5, a7, a5     addq (%rdx,%r9,8), %r10
add  a6, a6, a5     adcq $0, %rax
add  t0, t0, a7     addq (%rcx,%r9,8), %r10
sltu a5, t0, a7     adcq $0, %rax
add  a6, a6, a5     #after storing r10:
                    movq %rax, %r10
```

The RISC-V code has the same number of instructions as the hand-written code, but does not minimize the carry-to-carry latency; It may be possible to fix that aspect by source-code changes.

For ARM A64 the code is slightly worse than the hand-written code, for AMD64 significantly worse. Concerning ADX, clang does not generate ADX instructions with the options I used, possibly because ADX has not been included in x86-64-v4. The ARM A64 code follows the pattern of the two-summand addition, i.e., it now has 6 instructions.

The central loop of $m = sm_a + m_b$ can be written as:

```
for (i=0; i<n; i++) {
    s_t s1=ma[i];
    s_t s2=mb[i];
    d_t d = (s*(d_t)s1+s2)+carry;
    m[i] = d;
    carry = d>>64;
}
```

Here `carry` is a full word. Here's what clang produces for the central computation:

```
RV64GC
#a6=carry
#t0=s1 t1=s2
mulhu t2, t0, a1
mul   t0, t0, a1
add   a5, t1, a6
sltu  a6, a5, t1
add   t0, t0, a5
sltu  t1, t0, a5
add   a5, a6, t2
add   a6, a5, t1
```

```
ARM A64:
#x8=carry
#x11=s1 x12=s2
umulh x13, x11, x1
adds  x8, x12, x8
mul   x11, x11, x1
adcs  x12, xzr, xzr
adds  x11, x8, x11
adcs  x8, x12, x13
#x11=bottom(d)
```

```
AMD64
#rax=carry
mulxq (%r9,%r10,8), %rbx, %r11
xorl  %esi, %esi
addq  (%rcx,%r10,8), %rax
setb  %sil
addq  %rbx, %rax
adcq  %r11, %rsi
#after storing rax:
movq  %rsi, %rax
```

Again, the RISC-V code uses as many instructions as the hand-written version, but with worse carry-to-carry latency, which may be fixable by writing the source code a little differently. Likewise, the ARM A64 code uses as many instructions but worse carry-to-carry latency.

The AMD64 code also has a worse carry-to-carry latency than the hand-written code. It uses `mulxq` from the BMI2 extension (which is in x86-64-v4) to achieve more flexibility in register allocation. Compiling the C code to for straight AMD64 code results in an additional register-to-register move.

7.2.2 C with Carry

There are two ways to make carry explicit in some C compilers:

- The Intel-specific way is to use the intrinsic `c_out=_addcarry_u64(c_in, s1, s2, &sum)`
- The Clang-specific way is to use the builtin `sum= __builtin_addcll(s1, s2, c_in, &c_out)`

GCC does not have a builtin that supports carry-in.

I show the in the following, because it works for architectures other than IA-32 and AMD64.

For two-summand addition, the loop with the Clang-specific way looks as follows:

```
for (i=0; i<n; i++) {
  s_t s1=l1[i];
  s_t s2=l2[i];
  l[i] =
    __builtin_addcll(s1,s2,carry,&carry);
}
```

For RISC-V a five-instruction sequence for add with carry-in and carry-out is produced, slightly different from the one shown earlier.

For the other architectures, the sequences are:

```
ARM A64:
adds x9, x9, x10
cset w10, hs
adds x9, x9, x8
cset w8, hs
orr  w8, w10, w8

AMD64:
addq   (%rdx,%r8,8), %r9
setb   %r10b
addq   %rax, %r9
setb   %al
orb    %r10b, %al
movzbl %al, %eax
```

All these sequences fail to make use of the `adcs` (ARM A64) or `adc` (AMD64) instruction and produce longish, RISC-V-like workarounds.

The source code for the Intel-specific way differs only in the call to the intrinsic in the obvious way, and we only get code for AMD64:

```
addb $-1, %r8b
adcq (%rdx,%rax,8), %r9
setb %r8b
```

This code actually makes use of the `adc` instruction. However, because the compiler is pretty bad at preserving the C flag, it transfers it to a general-purpose register with `setb` right after the `adc` instruction, and transfers it back into the C flag with the `addb` right before it.

For the three-summand addition, using the Clang-specific builtin leads to two instances of the code shown above (with variations in register allocation) on all three architectures, which is worse than the code that uses double precision. This is particularly unexpected for those architectures that actually have a carry flag and an add with carry-in and carry-out.

When using the Intel-specific intrinsic, the result is also two instances of the three-instruction sequence shown above for this intrinsic, but that's not that much worse than the hand-written code.

7.2.3 `_BitInt`

The C23 standard allows defining integer types for which you can specify the number of bits, e.g, as follows:

```
typedef unsigned _BitInt(4096) u4096;
```

This feature is supported in gcc-15.1 and clang-20.1 on AMD64 (not on all architectures), with an implementation-dependent maximum number of bits. gcc-15.1 supports up to 65,535 bits, while clang-20.1 supports up to 8,388,608 bits. I looked at the code generated by these two compilers with `-Os` for a three-summand addition with 65535 bits.[12]

Clang produces straight-line code, which tends to make good use of the carry flag (no ADX instructions). However, the number of live words in this computation does not fit in the registers, and the register allocator deals with this situation not as well as one might like, resulting in about 9 instructions for each word of the result.

GCC produces a loop unrolled by a factor of 2, but unfortunately first performs the two additions from the first original iteration and then the ones from the second original iteration, so it has to move the carries between general-purpose registers and the carry flags all the time, in the way

[12] `https://godbolt.org/z/a5c9c8non`

that we see with the Intel intrinsic. Overall gcc executes 10 instructions per result word.

Still, these are early days for this language feature. Hopefully the generated code will improve over time. Certainly the potential exists: With this feature the programmer directly expresses the intended meaning, instead of expressing it through lower-level features such as double-precision integers, builtins or intrinsics, where the compiler may have to extract the intended meaning from this lowered code in order to transform it to the best code for the architecture (maybe Kogge-Stone addition using AVX-512).

8 Further Work

While the present work contains some performance estimations, actual measurements are missing. This is especially relevant given that the main focus in this paper has been latency, while resource constraints may limit performance more on today's processors.

9 Conclusion

The present work looks at several multi-precision integer operations, how AMD64, ARM A64 and RISC-V support implementing them and what the corresponding assembly language code looks like. These operations can implemented through libraries written in assembly language, or through programming languages. In particular, the present work looks at the code resulting from C with double-precision integers (`uint128_t`), C with a builtin or intrinsic for add with carry-in and carry-out, and C with `_BitInt(...)`. The code using double precision often works better than the code with intrinsics or (worst) builtins; while `_BitInt(...)` is not yet as good as one would naively expect, it has the best future potential.

References

[1] M. Anton Ertl. "Extending General-Purpose Registers with Carry and Overflow Bits". Paper rejected from ARITH 2024. 2024. URL: http://www.complang.tuwien.ac.at/anton/tmp/carry.pdf.

[2] Erdinc Ozturk et al. *New Instructions Supporting Large Integer Arithmetic on Intel Architecture Processors*. White Paper 327831-001. Intel, 2012. URL: https://www.intel.cn/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf.

IMPRESSUM

Schriftenreihe INSIGHTS
Themenreihe Engineering INSIGHTS

Herausgeber:
Fakultät Technik der
Dualen Hochschule Baden-Württemberg Stuttgart
Postfach 10 05 63, 70004 Stuttgart

Prof. Dr.-Ing. Harald Mandel
Prorektor für Forschung, Transfer und Nachhaltigkeit & Dekan Fakultät Technik
Lerchenstraße 1, 70174 Stuttgart

E-Mail: harald.mandel@dhbw-stuttgart.de
Tel.: +49 (0)711 1849-605

www.dhbw-stuttgart.de/technik/insights

Umschlaggestaltung: Kerstin Faißt
Bildnachweis: Gerd Altmann auf Pixabay
ISSN 2193-9098

ISSN 2193-9098

www.dhbw-stuttgart.de/technik/insights