What About the *Integer* Numbers?
Fast Arithmetic with Tagged Integers – A Plea for Hardware Support

Technical report MSR-TR-2022-17, 2022-07-11, v1.0. Presented at the ML workshop 2022.

DAAN LEIJEN, Microsoft Research, USA

In this article we consider how to efficiently support proper integers $\mathbb{Z}$ on modern hardware. The key issue we address is how to do fast tagged arithmetic where we use efficient small fixed-bit integers for most arithmetic, and use a slow path when an operation overflows or if one of the arguments happens to be a big (heap allocated) integer. We present three novel techniques (to the best of our knowledge) to do perform operations as fast as possible. In particular, the *sofa* technique can perform an addition on small integers with a single branch to check for both overflow and/or if one of the arguments was a big integer. Even though we improve upon common techniques, it is quite apparent that common instruction set architectures (x64, arm64, and riscV) do not support fast tagged integer arithmetic well and we also propose potential ISA extensions to support tagged arithmetic better in hardware. The techniques we discuss are widely applicable, not only for languages that natively support proper integers, like Python, Lisp, or Haskell, but also for example JavaScript that would use double's as a fallback.

1 INTRODUCTION

In the beautiful 1989 article "*What about the natural numbers?*" [14], Colin Runciman asks why modern programming languages lack deep support for natural numbers $\mathbb{N}$: "*In a great deal of programming, the more elaborate number systems – integers, floats, complexes and the like – are rarely, if ever, used. The natural numbers, on the other hand, get used almost all the time*"

In this article we consider a related question: What about the *integer* numbers $\mathbb{Z}$? Even sixty years into the history of machine-independent programming language design, modern hardware is surprisingly lacking in its support for proper integers. Of course, all common modern cpu's implement fixed N-bit integer operations like addition, substraction, and multiplication quite well. Such fixed N-bit numbers are limited in range though and wrap around on overflow. For example, on a two's complement system, adding 1 to the maximal signed integer overflows and becomes a negative number (as the minimal signed integer). This means that many of the laws that hold for mathematical integers $\mathbb{Z}$ are no longer valid – we cannot assume even the most basic properties of integers, such as $x + 1 > x$.

Such laws are not only useful to programmers but also for optimizations: one reason why the C standard defines integer overflow as undefined behaviour is to allow the optimizer to pretend that laws like $x + 1 > x$ hold in order to perform loop optimizations better. Moreover, a large fraction of all security issues are caused by unintended integer overflow at runtime. Even the Ethereum blockchain, which uses wide 256-bit integers, has been shown vulnerable to integer overflow in smart contracts [6].

Aside for some niche use cases, wrap around has no mathematical justification and higher level languages should generally trap on integer overflow (as is done in Rust for example). Even better, many higher level languages provide proper mathematical integers using "arbitrary precision" integers: instead of trapping on integer overflow, the implementation seamlessly switches the representation to heap allocated integers, where the range is only limited by the amount of available memory. This is done for example in languages like Python, Scheme, Lisp, Haskell, and Koka.

Unfortunately, even though modern hardware has high performance fixed N-bit integer instructions, the support for efficient arbitrary precision integers is sorely lacking. This is particularly grating as most modern hardware contains a plethora of exotic instructions for fairly uncommon domains – where are the *integer* operations? We see this lack of support for example in the Rust compiler where even basic overflow detection is disabled by default in release builds due to the performance overhead. In this paper we take a fresh look at supporting proper integer operations on modern hardware:

- We show how we can improve on the usual way of doing tagged integer arithmetic using three novel techniques (to the best of our knowledge):
  1. In Section 2.1 we show that by using "limited" two-bit tags (called *litbit*) we can improve upon the usual tagging techniques by detecting in a single step if the arguments were both small integers (and not heap allocated big integers).
  2. Even with the limited two-bit tag improvement, we still need two tests: one to detect if both arguments are small integers, and one to detect overflow. In Section 2.2 we propose a new scheme, called *sofa*, that can detect both situations using just a single test.
  3. In Section 2.8 we show a variation of the *sofa* technique using a pointer tag in the top bits, called *reno*. This leads to the shortest instruction sequence but seems to perform slightly worse overall than *sofa* in practice.
- We measure the performance of *litbit*, *sofa*, and *reno* against regular tagging techniques and fixed N-bit instructions (Section 3). We measure this both in isolated micro benchmarks, and also as part of larger programs in the Koka language. Overall, *sofa* seems slightly faster than the other approaches.
- Even though we improve upon common techniques, it is quite apparent that common instruction set architectures (x64, arm64, and riscV) do not support fast tagged integer arithmetic well, and generally need 3 to 8 instructions for a single tagged integer addition. In Section 4 we propose concrete tagged integer arithmetic instructions that we hope can be incorporated in future ISA extensions to improve this situation.
- We discuss how hardware support for tagged integer arithmetic is not only beneficial for seamless arbitrary precision integers, but can also directly benefit integer arithmetic in widely used languages like JavaScript for example. In particular, it interacts well with pointer biased double boxing (Section 4.2).

For concreteness, we use 64-bit examples in this article, but this work generalizes to most other bit sizes as well (and in particular 32-bit and 128-bit architectures).

A note to the reader: To the best of our knowledge, the *litbit*, *sofa*, and *reno* techniques are novel. However, we imagine some of these, or variants thereof, may have been used before – please write us if you know of such related work so we can include that here and give proper credit.

2 TAGGED INTEGER ARITHMETIC

To provide proper mathematical integers we need to be able to represent integers of arbitrary precision. In general this implies that such integers need to be heap allocated where the range is only limited by the amount of available memory. We call such heap allocated integers *big integers*. There are many available libraries for big integer arithmetic and we do not concern us further with a particular implementation.

Doing all integer operations on such heap allocated big integers would have a big performance impact though due to the overhead of allocation. In practice, languages with arbitrary precision integers use a range of *small* integers that do not need to be allocated; for example, Python represents the integers between -5 and 256 in a special way.

A common technique to avoid allocating small integers, is to use the least significant bit to distinguish small integers and pointers. Assuming pointers are always aligned to machine words, the least significant bit of a pointer is always zero, and we can set it to one for small integers. This encoding is widely used, ranging from statically typed languages like OCaml [9] to dynamic languages like Ruby, Common Lisp, and some JavaScript implementations.

In particular, a small integer n is represented as $\overline{n} = 2n+1$, and, assuming there is no overflow, we can add two small integers $\overline{n}$ and $\overline{m}$ as $\overline{n} + \overline{m} - 1$ (since $(2n+1) + (2m+1) - 1 = 2(n+m) + 1 = \overline{n+m}$). Here is how we can implement tagged integer addition in C (assuming we are on a 64-bit platform):

```
inline int64_t add_tagged(int64_t x, int64_t y) {
  if unlikely(((x&y)&1) == 0) return generic_add(x,y);
  int64_t z;
  const bool ovf = __builtin_add_overflow(x, y, &z);
  if unlikely(ovf) return generic_add(x,y);
  return (z-1);
}
```

The `__builtin_add_overflow` is a primitive provided by Clang and the GNU C compiler to detect if an addition overflowed. We use the `unlikely` macro to signify that those branches are unlikely to be taken at runtime – here we optimize for the common *fast path* where we assume that most additions are performed on small integers. In all other cases, when any of `x` or `y` are a big integer, or if the addition overflowed, we call the generic addition function `generic_add`. This routine can use big integer arithmetic to perform the addition. Since most integers will be small, the efficiency of `generic_add` is of less importance and we focus on the fast path instead. Using Clang 11, we get the following assembly on arm64 for the `add_tagged` routine:

```
and     x8, x0, x1         ; x8 = z = x&y
tbz     x8, #0, .gen_add   ; test if bit 0 is zero, and if so, goto .gen_add
adds    x8, x0, x1         ; x8 = z = x+y (and update the overflow flag)
b.vs    .gen_add           ; on overflow, goto .gen_add
sub     x0, x8, #1         ; x0 = z-1
ret
.gen_add:
b       generic_add        ; slow path: generic addition
```

This leads to about five instructions for a single addition. There are two branches but both of these should be quite predictable by modern hardware as the `generic_add` path is only used if big integers or overflow are involved.

This seems “not too bad” but note that this is five instructions for just a single addition which is a very common operation. Moreover, the arm64 ISA is very compact, on x64 for example we get:

```
mov     eax, edi           ; eax = x&y
and     eax, esi
test    al, 1              ; if (eax & 1) == 0
je      .gen_add           ;   then use generic addition
mov     rax, rdi           ; rax = z = x+y
add     rax, rsi
jo      .gen_add           ; on overflow use generic addition
add     rax, -1            ; rax = z-1
ret
```

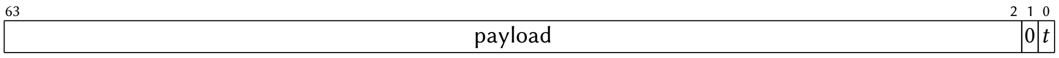
which uses eight instructions. On an architecture like riscV which does not have an overflow flag, it also takes eight instructions (and 2 extra registers):

```
and     a2, a0, a1         ; a2 = x&y&1
andi    a2, a2, 1
beqz    a2, .gen_add       ; if a2==0 use generic addition
add     a2, a0, a1         ; a2 = z = x+y
slt     a3, a2, a0         ; a3 = a2 < a0
slti    a4, a1, 0          ; a4 = a1 < 0
bne     a4, a3, .gen_add   ; on overflow, use generic addition
addi    a0, a2, -1         ; a0 = z-1
ret
```

In the following sections, we look at two novel ways (to the best of our knowledge) to improve the common tagging technique: using limited two-bit tags, and sign-extended overflow arithmetic.

2.1 Limited Two-Bit Tags

We can do bit better than the usual tagged addition by using *two* of the least significant bits for the tag. However, instead of allowing both bits to be used, we limit the tag to be either 0 or 1, and thus bit one is always clear. For a tag t, we have:



We call this a "limited two-bit tag" (or *litbit*). In particular, for pointers (big integers), the tag is zero (as $\boxed{\ldots \mid 0 \mid 0}$), while we represent a small integer n as $\overline{n} = 4n + 1$ (as $\boxed{n \mid 0 \mid 1}$). With bit one always clear, we can now add any arguments x and y together and check afterwards in one test on the result if the original arguments were actually small integers. These are the possible cases for the two least significant bits:

$\boxed{\ldots \mid 0 \mid 0}$	+	$\boxed{\ldots \mid 0 \mid 0}$	=	$\boxed{\ldots \mid 0 \mid 0}$	pointer + pointer
$\boxed{\ldots \mid 0 \mid 0}$	+	$\boxed{\ldots \mid 0 \mid 1}$	=	$\boxed{\ldots \mid 0 \mid 1}$	pointer + small
$\boxed{\ldots \mid 0 \mid 1}$	+	$\boxed{\ldots \mid 0 \mid 0}$	=	$\boxed{\ldots \mid 0 \mid 1}$	small + pointer
$\boxed{\ldots \mid 0 \mid 1}$	+	$\boxed{\ldots \mid 0 \mid 1}$	=	$\boxed{\ldots \mid 1 \mid 0}$	small + small

As we can see, only if we added two small integers the result of the addition has bit one set in the result. We can implement this check in C as:

```c
inline int64_t add_ovf(int64_t x, int64_t y) {
  int64_t z;
  const bool ovf = __builtin_add_overflow(x, y, &z);
  if unlikely(ovf || (z & 2) == 0) return generic_add(x, y);
  return (z-1);
}
```

On arm64, this compiles to only four instructions for addition:

```
adds    x8, x0, x1        ; x8 = z = x+y (and update the overflow flag)
b.vs    .gen_add          ; on overflow, goto generic addition
tbz     w8, #1, .gen_add  ; if bit 1 is zero, goto generic addition
sub     x0, x8, #0x1      ; x0 = z-1
ret
```

For x64 and riscV we need to adapt the C code to have separate cases for overflow or a mismatched tag in order to generate optimal code with Clang and GCC:

```c
noinline int64_t generic_add_tag_mismatch( int64_t x, int64_t y) {
  return generic_add(x,y)
}

inline int64_t add_ovf(int64_t x, int64_t y) {
  int64_t z;
  const bool ovf = __builtin_add_overflow(x, y, &z);
  if unlikely(ovf) return generic_add(x,y);
  if unlikely((z & 2) == 0) return generic_add_tag_mismatch(x, y);
  return (z-1);
}
```

Without this adaption both compilers fail to emit direct jump-on-overflow instructions. With the fix in place, the instructions for x64 become:

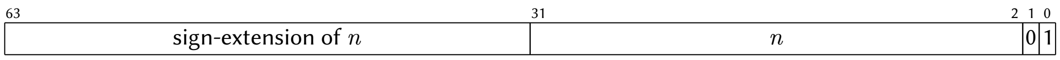
```
mov     rax, rdi
add     rax, rsi
jo      .gen_add
test    al, 2
je      .gen_add_mismatch
xor     rax, 3
ret
```

which is 2 instructions less than before, while the new sequence for riscV is one instruction less:

```
add     a2, a0, a1
slt     a3, a2, a0
slti    a4, a1, 0
bne     a4, a3, .gen_add
andi    a3, a2, 2
beqz    a3, .gen_add_mismatch
xori    a0, a2, 3
ret
```

2.2 Sign-Extended Overflow Arithmetic

We have improved the code, but we still need two branches for each addition. It turns out we can combine these in a single branch that tests for both overflow and big integer arguments at once. First, we restrict small integers to half-words. Every small integer n is still represented as $4n + 1$ but now limited in range to half word size only:



We can directly add such sign-extended small integers using full registers. A portable way to detect overflow for such half-word operations is to check for overflow after the addition: if the full result `z` does not equal the sign-extended lower half-word of the result, then the addition overflowed, i.e. `z != (int32_t)z`.

The idea is to combine this method for overflow detection with the earlier limited two-bit tag check, where we ensure that in the result `z` bit one is set after the addition, i.e. `(z|2) != (int32_t)z` (or right biased as `z != (((int32_t)z)|2)`). We can now test both for overflow and the correct tags in a single test! We call this Sign-extended OverFlow Arithmetic (or *sofa*), and the new technique can be implemented in C as:

```
inline int64_t add_sofa(int64_t x, int64_t y) {
  const int64_t z = (int64_t)((uint64_t)x + (uint64_t)y);
  if unlikely((z|2) != (int32_t)z) return generic_add(x, y);
  return (z-1);
}
```

Unfortunately, since signed integer overflow is undefined behaviour in C, we cannot directly add the arguments as `x + y` as these might be pointers and the addition could thus overflow at runtime. Instead, we cast the arguments and use unsigned addition which is defined in C to use modulo 2^n arithmetic (and does not have undefined behaviour). Compiling the *sofa* addition results in the following assembly on arm64:

```
add     x8, x1, x0        ; x8 = z = x+y
orr     x9, x8, #2        ; x9 = x8|2 (set bit one in the result)
cmp     x9, w8, sxtw      ; does x9 equal the sign extended half-word w8 ?
b.ne    .gen_add          ; if not equal, use generic addition
sub     x0, x8, #1        ; x0 = z-1
ret
```

We are back to five instructions but now only need a single branch to test for all exceptional situations. On x64, if we use right-biased extension, the code generated by Clang 14 is:

```
lea     rax, [rsi + rdi]  ; rax = z = x+y
movsxd  rcx, eax          ; rcx = sign extended (int32_t)z
or      rcx, 2            ; rcx |= 2
cmp     rax, rcx          ; if rax != rcx
jne     .gen_add          ;   then use generic addition
add     rax, -1           ; rax = z-1
ret
```

which is now just six instructions (versus eight before). On riscV, the Clang 14 compiler optimizes the (righ-biased) sign extension by doing a direct sign-extended half-word addition:

```
add     a2, a1, a0        ; a2 = z = x+y
addw    a3, a1, a0        ; a3 = sign extended (int32_t)x + (int32_t)y
ori     a3, a3, 2         ; a3 |= 2
bne     a2, a3, .gen_add  ; if a3 != a4 use generic addition
addi    a0, a2, -1        ; a0 = z-1
ret
```

which is now five instructions (versus eight before). In Section 3 we present detailed benchmarks where the *sofa* technique turns out to be slightly faster than the previous approaches.

2.3 Subtraction

The two-bit tag check works well for addition, but not directly for subtraction. If we directly subtract the arguments, we cannot detect two small integers as a unique case:

...	0	0	−	...	0	0	=	...	0	0	pointer − pointer
...	0	0	−	...	0	1	=	...	1	1	pointer − small
...	0	1	−	...	0	0	=	...	0	1	small − pointer
...	0	1	−	...	0	1	=	...	0	0	small − small

where both two pointers and two small integers set the two least significant bits to zero. We can fix this by initially xor'ing the first argument with 3 which flips the tag bits:

...	1	1	−	...	0	0	=	...	1	1	pointer − pointer
...	1	1	−	...	0	1	=	...	1	0	pointer − small
...	1	0	−	...	0	0	=	...	1	0	small − pointer
...	1	0	−	...	0	1	=	...	0	1	small − small

where we can now check afterwards if bit one is clear. The nice thing about this scheme is that unlike addition, the final result already has the right tag bits and does not need to be adjusted. We can implement this in C as:

```
inline int64_t sub_sofa(int64_t x, int64_t y) {
  const int64_t z = (int64_t)(((uint64_t)x^3) - (uint64_t)y);
  if unlikely((z&~2) != (int32_t)z) return generic_sub(x, y);
  return z;
}
```

Giving the following assembly on arm64 with Clang 14:

```
eor     x8, x0, #3        ; x8 = x^3
sub     x8, x8, x1        ; x8 = x^3 - y
and     x9, x8, #-3       ; x9 = x8 & ~2
cmp     x9, w8, sxtw      ; does x9 equal the sign-extended w8 ?
b.ne    .gen_sub          ; if not, use generic subtraction
mov     x0, x8
ret
```

and similarly for x64 and riscV.

2.4 Multiplication and Division

Unfortunately, we cannot use a single limited two-bit tag check for multiplication and division. This may matter less though since both of these operations are both less common, and more expensive than an addition or subtraction. We can still use the half-word sign extension though to check for overflow. Using Euclidean integer division [1, 7], we have $\overline{nm} = 4nm + 1 = 2n2m + 1 = \frac{4n}{2} \cdot \frac{4m}{2} + 1 = \frac{(4n+1)}{2} \cdot \frac{(4m+1)}{2} + 1 = \frac{\overline{n}}{2} \cdot \frac{\overline{m}}{2} + 1$, and we can implement multiplication as:

```
inline int64_t mul_sofa(int64_t x, int64_t y) {
  if unlikely(((x & y) & 1) == 0) return generic_mul(x, y);
  const int64_t z = (x>>1) * (y>>1);
  if unlikely(z != (int32_t)z) return generic_mul(x, y);
  return (z+1);
}
```

There are various subtle parts in this code:

- We need to check if the arguments are small integers before the (signed) multiplication to avoid possible overflow at runtime which is undefined behaviour in C.
- We assume right-shift (>>) is compiled to an arithmetic right shift where the sign bit is preserved. This is implementation defined in C but generally the case on architectures that use two's complement representation for integers.
- We must use arithmetic right shift (`x>>1`) instead of C division (`x/2`). Our derivation above uses Euclidean division where the modulus is always positive, and as a consequence we always have that $\frac{x}{2} = x >> 1$ [7]. C division is defined though as truncating towards zero, which can give wrong results in our case. In particular, the equality $\frac{4n}{2} = \frac{4n+1}{2}$ which we used in our earlier derivation does *not* hold when using division that truncates towards zero: for example with $n = -1$ and using / for C division, we have $-4/2 \neq -3/2$.

When we compile with GCC 11, we get the following assembly on arm64:

```
and     w8, w0, w1        ; w8 = w0&w1
tbz     w8, #0, .gen_mul  ; if bit 0 of x8 is zero, goto generic multiplication
asr     x8, x0, #1        ; x8 = (x8>>1) * (x9>>1)
asr     x9, x1, #1
mul     x8, x9, x8
cmp     x8, w8, sxtw      ; if x8 != (int32_t)x8, goto generic multiplication
b.ne    .gen_mul
add     x0, x8, #1
ret
```

Division is similar: using again / for C division (which truncates towards zero), we have $\overline{n/m} = 4(n/m) + 1 = 4(2n/2m) + 1 = 4(\frac{4n}{2}/\frac{4m}{2}) + 1 = 4(\frac{4n+1}{2}/\frac{4m+1}{2}) + 1 = 4(\frac{\overline{n}}{2}/\frac{\overline{m}}{2}) + 1$. When we implement this, we need again be careful to test upfront for small integers and use arithmetic right-shift instead of C division:

```
inline int64_t div_sofa(int64_t x, int64_t y) {
  if unlikely(((x & y) & 1) == 0) return generic_div(x, y);
  const int64_t z = 4*((x>>1) / (y>>1));
  if unlikely(z != (int32_t)z) return generic_div(x,y);
  return (z+1);
}
```

One might be surprised that we still need an overflow test – is this even possible with division? Such is the case though for a single combination of arguments where we divide the smallest small integer, $\overline{-2^{29}}$ ($= -2^{31} + 1$), by $\overline{-1}$ ($= -3$), resulting in $z = 4 \cdot 2^{29}$ which overflows the 32 bits of our small integers. The C code is compiled to the following assembly on arm64 (using GCC 11):

```
and     x2, x0, x1
tbz     x2, 0, .gen_div
asr     x3, x1, 1
asr     x2, x0, 1
sdiv    x2, x2, x3
lsl     x2, x2, 2
sxtw    x3, w2
cmp     x2, w2, sxtw
bne     .gen_div
add     x0, x3, 1
ret
```

2.5 Comparisons

Comparison operations can be implemented quite efficiently if we assume that integers are *normalized*: that is, any big integer argument only holds integers that cannot be represented as a small integer. This way, we can directly compare small integers. For example, we can implement ($\geqslant$) as:

```
inline bool is_gte_sofa(int64_t x, int64_t y) {
  if unlikely(((x&y)&1) == 0) return generic_gte(x,y);
  return (x >= y);
}
```

For equality and inequality, we can further optimize this by only testing one of the arguments if it is a small integer:

```
inline bool is_eq_sofa(int64_t x, int64_t y) {
  if unlikely((x&1) == 0) return generic_eq(x,y);
  return (x == y);
}
```

This works especially well when comparing against a constant argument `x` where the compiler can usually discard the small integer test.

2.6 Constant Arguments

We can optimize various other operations as well if one of the arguments is a known constant. For example, we could define addition of a small constant `c` as:

```
inline int64_t add_sofa_const(int64_t x, int64_t c) {
  const int64_t z = (int64_t)((uint64_t)x + 4*(uint64_t)c);
  if unlikely((z|2) != (int32_t)z) return generic_add(x, 4*c + 1);
  return z;
}
```

This improves the generated instructions slightly since we no longer require the adjustment to `z` (as `z-1`) since we add the constant directly without its tag bit.

2.7 Switching the Tag Bits

In our design, the limited two-bit tags use 0 for pointers, and 1 for small integers. However, we could also turn this around and tag pointers with 1, representing an aligned pointer p as $p + 1$, while a small integer n would be represented as $4 * n$. The advantage of such encoding is that small integers can be directly added, since $\overline{n} + \overline{m} = 4 * n + 4 * m = \overline{n + m}$, without needing the -1 adjustment. Dereferencing a pointer though does need a -1 adjustment now but on most architectures such offset adjustment is part of the load- and store instructions and thus comes for free.

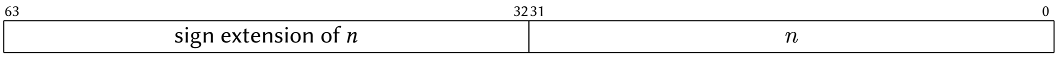
This argument holds for systems that use tags only for boxing (like the OCaml runtime system for example [9]), but for our goal of supporting seamless arbitrary precision integers the trade-off is less clear. In particular, we still need to check for overflow and if both arguments were small integers. With the tags switched, the addition now needs to check if both of the lower two bits in the result are clear. That is still fine (the `z|2` becomes `z&~3`), and we need an instruction less since addition no longer needs the adjustment (`z-1`). However, we now need an extra adjustment for subtraction where the C code becomes:

```
int64_t sub_sofa_inv(int64_t x, int64_t y) {
  const int64_t z = (int64_t)((uint64_t)x - ((uint64_t)y^3) + 3);
  if (unlikely((z&~3) != (int32_t)z)) return sub_generic(x, y);
  return z;
}
```

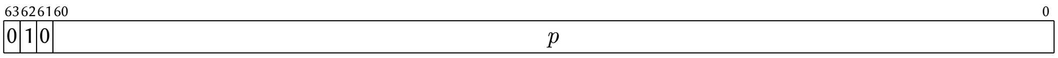
Moreover, if the architecture directly supports tagged integer instructions (as discussed in Section 4), one may prefer to keep pointers *as is* for simplicity (since the hardware can process a tag bit just fine). In Koka, we prefer to keep pointers aligned. The choice is not clear cut though – in the benchmarks we measured the switched tag bits as *xsofa* and it seems to perform very similar to *sofa*.

2.8 Range Extended Overflow Arithmetic

The *sofa* technique suggest another way to have a single test for both overflow and big integers. In particular, we can leave out the lower tag bits, and represent small integers n directly as sign-extended half-words:



We represent a pointer now as $\overline{p} = (p >> 3) \mid (1 << 62)$ where we use the top 3 bits as a tag:



This gives use the following ranges:

$$-2^{63} \leqslant inv < \underline{-2^{31} \leqslant \overline{n} < 2^{31}} < inv < \underline{2^{62} \leqslant \overline{p} < 2^{62} + 2^{61}} < inv \leqslant 2^{63} - 1$$

where inv denotes invalid ranges. This is set up this way such that the addition of two pointers, or a pointer to a small integer is always outside the range of small integers. That is, the addition of a pointer is at least $p_{min} + n_{min} = 2^{62} - 2^{31} > 2^{31}$, and the addition of two pointers is at most $p_{max} + p_{max} = 2^{63} + 2^{62} =_{(signed\ 64-bit)} -2^{62} < -2^{31}$. Moreover, testing if an integer is in the range of small integers can be done using sign-extension again (as `z == (int32_t)z`), and addition becomes:

```
inline int64_t add_reno(int64_t x, int64_t y) {
  const int64_t z = (int64_t)((uint64_t)x + (uint64_t)y);
  if unlikely(z != (int32_t)z) return generic_add(x, y);
  return z;
}
```

which leads to the following instructions on arm64:

```
add     x0, x1, x0       ; z = x + y
cmp     x0, w0, sxtw     ; z == (int32_t)z ?
bne     .gen_add         ; if not, use generic addition
ret
```

This is very good and only needs 3 instructions! On x64 the code is similarly dense:

```
lea     rax, [rsi + rdi]  ; z = x + y
movsxd  rcx, eax          ; rcx = sign extended (int32_t)z
cmp     rcx, rax          ; z == (int32_t)z ?
jne     .gen_add          ; if not, use generic addition
ret
```

2.8.1 Subtraction.

For substraction the same trick does not work though as subtracting two pointers also brings it in the range of small integers. It seems we need again two tests where we test upfront of the second argument is a pointer:

```
inline int64_t sub_reno(int64_t x, int64_t y) {
  const int64_t z = (int64_t)((uint64_t)x - (uint64_t)y);
  if unlikely(y > INT32_MAX || z != (int32_t)z) return generic_sub(x, y);
  return z;
}
```

Note that if the first argument is a pointer, the range check on `z` will detect this so we only need to check the second argument `y`.

However, we can do this still in a single test, if we first double the first argument ensuring we never reach the small integer range if either `x` or `y` is a pointer. In particular, the result of `x+x-y` is at least: $p_{min} + p_{min} - p_{max} = 2^{63} - 2^{62} - 2^{61} = 2^{62} - 2^{61} = 2^{61} > 2^{31}$ and at most $p_{max} + p_{max} - p_{min} = 2 \cdot (2^{62} + 2^{61}) - 2^{62} = 2^{63} =_{(signed\ 64-bit)} = -2^{63} < -2^{31}$ We can implement this as:

```
inline int64_t sub_reno(int64_t x, int64_t y) {
  const int64_t z = (int64_t)(2*(uint64_t)x - (uint64_t)y);
  if unlikely(z != (int32_t)z) return generic_sub(x, y);
  return (int64_t)((uint64_t)z - (uint64_t)x);
}
```

which gives the following instructions on arm64:

```
lsl     x8, x0, #1        ; x8 = 2*x
sub     x8, x8, x1        ; x8 = 2*x8 + y
cmp     x8, w8, sxtw      ; (z == (int32_t)z) ?
b.ne    .gen_sub          ; if not, use generic substraction
sub     x0, x8, x0        ; x0 = z - x
ret
```

Testing showed though that the first version of substraction with an extra test is always a bit faster than the second variant, and in the benchmarks in Section 3 that is the one we use.

2.8.2 Drawbacks.

The *reno* technique seems a clear winner on paper, but in the benchmark section (Section 3) we see that it does not perform as well as *sofa* in practice. We believe the main reason for this is that it is more expensive in *reno* to test upfront if a value is a pointer or integer. For example, for comparisons or multiplication, we need to test upfront if two arguments are both small integers:

```
bool are_small_sofa( int64_t x, int64_t y ) {
  return ((x&y)&1) != 0;
}

bool are_small_reno( int64_t x, int64_t y ) {
  return ((x>>61)+(y>>61)) <= 0;
}
```

Here we optimize the small integer test for *reno* by observing the if `x>>61` is always 0 or -1 for small integers, and 2 for pointers – which makes the addition of `x` and `y` positive only if either or both are pointers. On x64, the *sofa* method generates code like:

```
and     eax, edi
test    eax, 1
jz      .notsmall
```

while for the *reno* method this results in:

```
sar     rdi, 61
sar     rsi, 61
add     rsi, rdi
cmp     rsi, 0
jg      .notsmall
```

This seems like it should make not much difference but in practice it does seem to make *sofa* slightly faster than *reno* even though the addition and substraction operations in isolation are better. Another item that may cause this is that both clang and gcc tend to optimize the tests for small integers in *reno* by using large constants (like `1UL<<62`) to do direct comparisons but such constants need to be loaded as large immediates and not all architectures deal well with this.

3 BENCHMARKS

We measured the performance of fixed 32-bit instructions of addition, subtraction, and multiplication (*int32*), against the performance of the regular tagged arithmetic (*tagged*) (as discussed in Section 2),

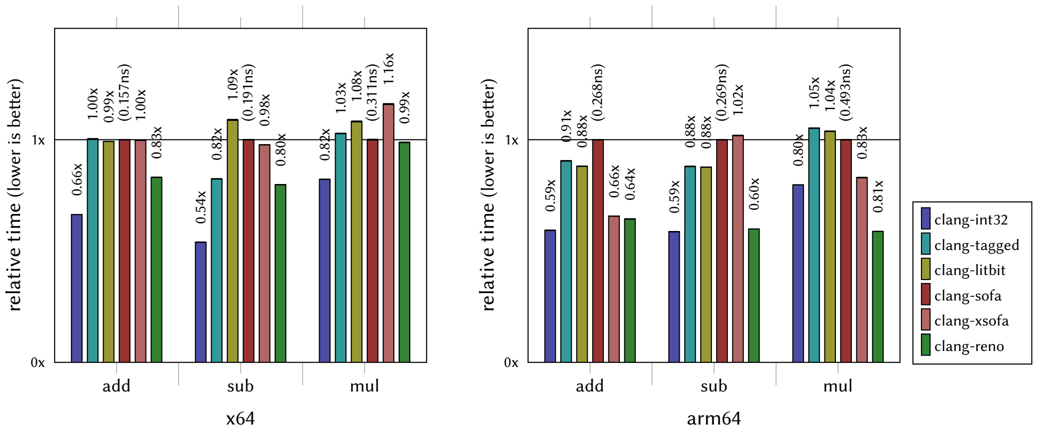


Fig. 1. Instruction cost per iteration on x64 (AMD 5950x, clang 10.0) and arm64 (Apple M1, clang 13.1).

the limited two-bit tag approach (*litbit*) (as described in Section 2.1), the *sofa* approach with sign-extension overflow checking (Section 2.2), and finally the *reno* technique with sign-extension overflow checking with a pointer tag in the top bits (Section 2.8, using the two-test version of subtraction).

We implemented a tight loop doing 1 billion of such operations, with the loop unrolled to 16 operations per iteration. We measured the time needed per operation using high resolution timings on both x64 (on an AMD 5950x, Ubuntu 20.04, clang 10.0), and arm64 (on an Apple M1, macOS 12.3.1, clang 13.1). Figure 1 shows the results relative to the performance of *sofa*.

We can clearly see that plain *int32* operations are 1.5× to 2× faster for addition and subtraction, and about 1.2× faster for multiplication – checking for overflow and small integers does have a real impact on performance. As expected, the *reno* technique is more efficient and rivals the *int32* performance on the M1. Somewhat surprisingly, the other methods are all very close even if some use twice the amount of branches. We conjecture that on this kind of micro benchmark, a modern super scalar processor with deep speculation and register renaming can essentially do those extra operations "for free".

Figure 2 shows benchmark results for more realistic workloads. Here we used four integer heavy benchmark programs written in Koka [8], *nqueens*, *hamming*, *tak*, and *pyth*. We also ran various other benchmarks taken from Lorenzen and Leijen [10] but these showed little differences as the workloads were dominated by other operations (like allocations or pattern matching). For each program, we wrote one version using direct 32-bit integers (`int32`) (without overflow checks), and one version with arbitary precision integers (`int`). The latter were then compiled with five different variants of the compiler: using either *tagged*, *litbit*, *sofa*, *xsofa*, or *reno* style arithmetic. The *xsofa* variant is as *sofa* but with the tag bits inverted as described in Section 2.7.

The *nqueens* benchmark calculates all solutions to the n-queens 13 problem. This benchmark does a lot of allocation but still a large fraction of the runnig time is spend in the `safe` function that checks if a queen can be positioned safely, and uses integer addition, subtraction, and comparisons:

```
fun safe( queen : int, diag : int, ^xs : list<int> ) : bool
  match xs
    Cons(q,qs) -> (queen != q && queen != (q + diag) &&
                   queen != (q - diag) && safe(queen, diag+1, qs))
    _          -> True
```

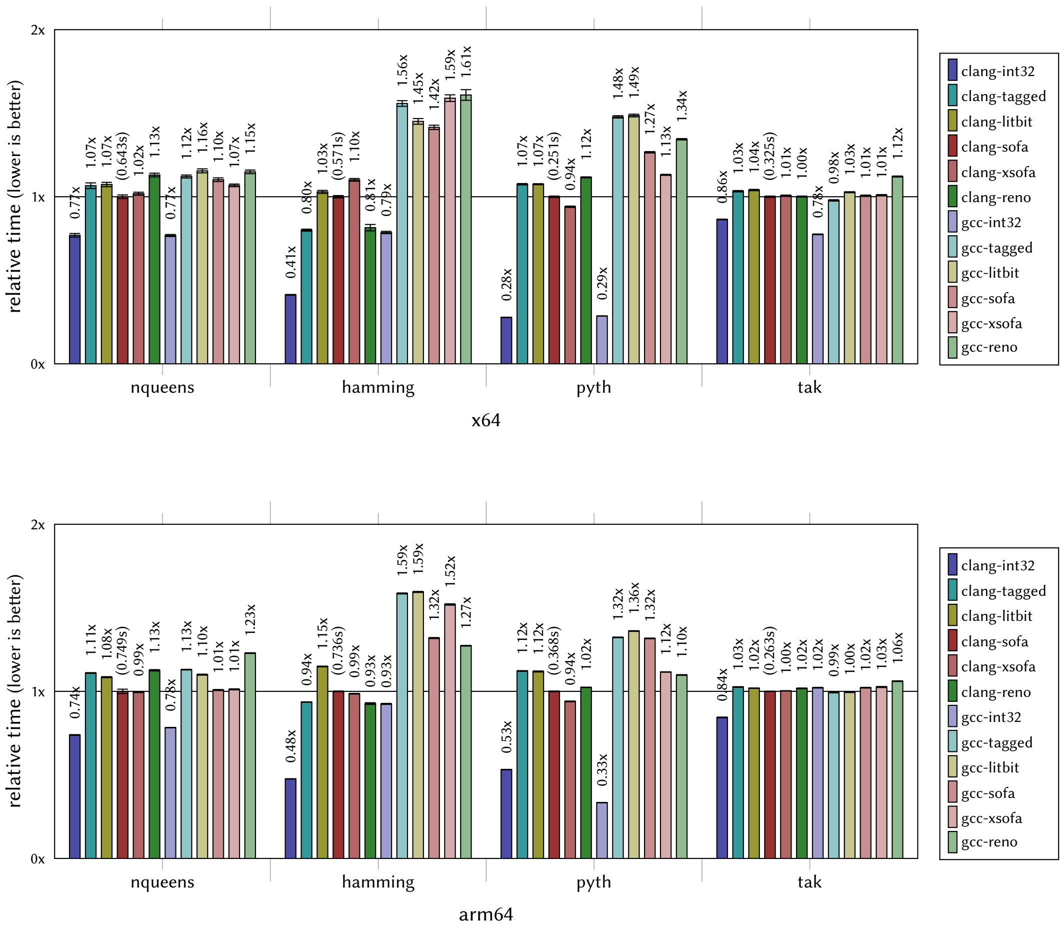


Fig. 2. Benchmarks on x64 (AMD 5950x, clang 10, gcc 9.4) and arm64 (Apple M1, clang 13.1, gcc 11.2).

The *tak* benchmark calculates the Takeuchi number for `tak(36,24,14)`, and is defined as:

```
fun tak(x : int, y : int, z : int ) : div int
  if y < x:r
    then tak( tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y) )
    else z
```

The *hamming* benchmark calculates Hamming numbers (of the form $2^i \cdot 3^j \cdot 5^k$) using the Euclid's method of finding the greatest common divisor of 42 by using repeated subtraction:

```
fun gcd( x : int, y : int ) : div int
  if x > y   then gcd( x - y, y )
  elif x < y then gcd( x, y - x )
  else x

fun is-hamming( x : int ) : div bool
  gcd(x,42) == 1
```

Finally, the *pyth* benchmark calculates the number of Pythagorean triples up to some bound `n`, defined as:

```
fun pyth(n : int ) : int
  fold-int(1, n/3, 0) fn(x,xcount)
    val xx = x*x
    fold-int(x+1, n/2, xcount) fn(y,ycount)
      val yy = y*y
      fold-int-while(y+1, n/2, ycount) fn(z,zcount)
        val zz = z*z
        if (xx+yy == zz) then Just(zcount + 1)
        elif (xx+yy >= zz) && (x+y+z <= n) then Just(zcount)
        else Nothing  // break
```

Normally, Koka uses Perceus style reference counting [10, 13, 15] but that would impact our measurements as integers are reference counted since they might point to a big integer (even though this does not happen in our benchmarks). Therefore, in our benchmarks we turned off reference counting for integers. We compiled all programs with gcc 9.4 and clang 10.0 on Ubuntu 20.04 on x64, and gcc 11.2 and clang 13.1 on macOS 12.3.1 on arm64. In our benchmarks clang generally did better and shown in the first 6 darker bars, with gcc following with 6 lighter bars.

One thing that stands out when we look at the results in Figure 2, is that using arbitrary precision integers can have a large cost: the *int32* approach is between 1.2× to 4× faster! This shows that direct hardware support for tagged integer arithmetic is sorely needed; given the speed of a plain addition or substraction *any* extra needed instructions can impose a heavy penalty.

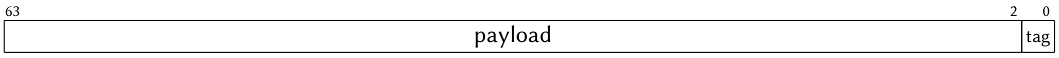
On both x64 and arm64, across clang and gcc, the *sofa* approach seems almost always the fastest. An exception is *clang-tagged* and *clang-reno* on *hamming*, but it turns out that in that particular case both the comparison, and the following subtraction call `are_small(x,y)` and clang is able to optimize out the second check. Surprisingly, outside of *hamming*, *reno* does not do so well. As discussed in Section 2.8.2, we believe this is due to needing a slightly more expensive check to distinquish pointers from small integers (as inspecting the generated assembly does not indicate any other obvious reason).

Even though the *sofa* approach was slower than the others in the micro benchmarks of Figure 1, on the larger benchmarks the *sofa* approach is now overall the fastest. We conjecture that on modern hardware the *sofa* approach lends itself better to take advantage of multiple execution units and the use of less branches may improve overall branch prediction.

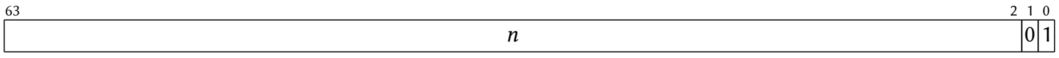
4 INSTRUCTIONS FOR TAGGED ARITHMETIC

Given the high cost of tagged integer arithmetic, we propose a design for direct tagged integer arithmetic instructions that we hope can be added to future iterations of hardware architectures. We believe that nowadays the use of such instructions would be ubiquitious. In particular, such instructions not only benefit the performance of arbitrary precision integers (like in Haskell or Koka), but would also directly improve the performance widely used languages like JavaScript and Python (see Section 4.2).

We propose to use a representation where the two least significant bits are used for a two-bit *tag*:



Just like our limited two-bit tags (Section 2.1), a tag of 1 (… 0 1) is used for small integers, while the tag 0 is used for pointers (to big integers). Since the operations are implemented in hardware, we do not need to limit the tags though and can allow tags values of 2 and 3 for user defined tags. A signed small integer n is thus represented as $\overline{n} = 4n + 1$:



The instructions `tadd`, `tsub`, `tcmp`, `tmul`, and `tdiv` perform tagged addition, substraction, comparison, multiplication and division. Tagged arithmetic instructions *fail* if either argument has a tag not equal to 1, or if the operation overflows. On failure, the destination register is set to 0, and, on architectures with flags, the overflow flag is set. On success, the destination register is set to the result of the operation ($\overline{-1}$, $\overline{0}$, or $\overline{1}$ for `tcmp`).

For example, on x64 with a `tadd` instruction, we can do tagged addition as:

```
mov     rax, rdi          ; tagged add rdi and rsi
tadd    rax, rsi
jo      .gen_add          ; on overflow or invalid tags, goto generic addition
ret
```

and similarly for arm64:

```
tadd    x0, x1, x0        ; tagged add x1 and x0
b.vs    .gen_add          ; on overflow or invalid tags, goto generic addition
ret
```

On architectures without flags (like riscV), we can check if the result register is set to zero since this is never a valid result otherwise. Here is how that would look on riscV:

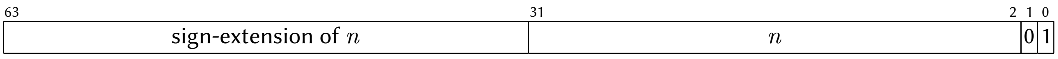
```
tadd    a0, a1, a0        ; tagged addition
beqz    a0, .gen_add      ; if a0==0 goto generic addition
ret
```

Note that it would be possible to use just a single *tag* bit to support tagged addition. However, having two bits for the tag makes it easier for various dynamic languages that may need to support many different kinds of datatypes. Having two tag bits also helps with portability where implementations can fall back seamlessly to a *sofa* software implementation if there is no hardware support.

4.1 Half-Word Arithmetic

On 64-bit systems it may be advantageous to also support half-word tagged arithmetic instructions as used by *sofa*. In particular for languages like JavaScript that use pointer-biased double boxing as shown in the next section.

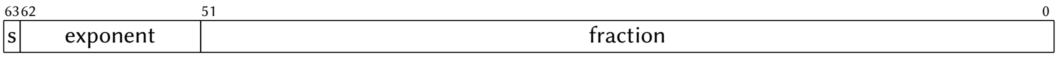
The `taddw`, `tsubw`, `tcmpw`, `tmulw`, and `tdivw` instructions are used for half-word addition, substraction, comparison, multiplication, and division. Small integers are still represented as $4n + 1$ but limited to 32-bits for these instructions (as shown in Section 2.2):



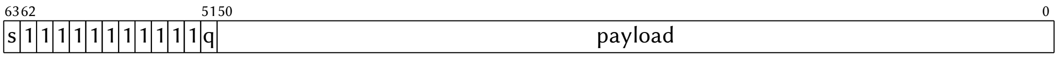
A half-word instruction fails as before if either argument has a tag unequal to 1 or on (32-bit) overflow, but *also* fails if for either argument the upper 32 most significant bits are not a proper sign extension of the 32 least significant bits.

4.2 Pointer Biased Double Boxing

A common technique in dynamic languages is "NaN boxing" where pointers and small integers are encoded in the NaN space of 64-bit double values. This is used to avoid needing to allocate double values in the heap. In particular, a 64-bit IEEE 754 double [5] is represented with a sign-bit s, an 11-bit exponent, and a 52-bit fraction:



The NaN values are represented with an exponent where all bits are set, and a fraction that is non-zero:



The q bit determines if the NaN is quiet or signalling and is usually set for quiet NaN's. The default NaN representation on many architectures is positive with the sign-bit cleared (Sparc, MIPS, PA-RISC, ARM, and riscV for example), while some architectures, like x86/x64 and Alpha, use a negative NaN with the sign-bit and quiet-bit set, and the payload set to zero [16, page 54]. As such, it is fairly portable for implementations to use the negative NaN's between FFF8:0000:0000:0001 and FFFF:FFFF:FFFF:FFFF to encode other values, like pointers and small integers.

Such encoding is is not ideal though as one can no longer directly dereference pointers (as the upper 13 bits need to be masked out). A better approach is to use *pointer-biased* double boxing (also known as "NuN-boxing"), as used in the JavaScript Core engine for example [11]. To use this technique, we assume that the upper range of the negative NaN's starting at FFFE:0000:0000:0000 up to FFFF:FFFF:FFFF:FFFF, are never generated by floating point operations. A double is now encoded by adding 0001:0000:0000:0000, putting the resulting encoded doubles in the range 0001:0000:0000:0000 up to FFFE:FFFF:FFFF:FFFF. Any other 49-bit sign-extended number, that is, with the 16 most significant bits all clear or all set, can now be used for pointers and other values:

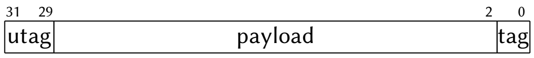
0000:0000:0000:0000	...	0000:FFFF:FFFF:FFFF	positive pointers and values
0001:0000:0000:0000	...	FFFE:FFFF:FFFF:FFFF	encoded doubles
FFFF:0000:0000:0000	...	FFFF:FFFF:FFFF:FFFF	negative pointers and values

The beauty of this encoding is that pointers and small integers can now be directly represented using the regular *sofa* technique with the tag in the two least significant bits. Moreover, if we use half-word 32-bit sign-extended small integers, then the half-word tagged arithmetic instructions work directly on such representation: not only any pointer or value with the *tag* bits not set to 1 will fail, but also any encoded doubles! Since the encoded doubles never have the top 32 bits all set or all clear, these are never a proper sign extension of the least significant 32 bits. For example, in JavaScript we could use this to have fast arithmetic on small 32-bit integers most of the time, and falling back seamlessy to double arithmetic when needed (where the `generic_add` function would use doubles instead of big integers).

5 RELATED WORK

One of the first languages to support seamless arbitrary precision integers is Lisp. We believe support for big integers started with the Stanford LISP 1.6 implementation [12] on the PDP-10 in the early sixties. This implementation is perhaps also the first one to use a form of tagged integers where small integers where represented by pointers into unaddressable space.

To our knowledge, the first implementation to use the least significant bits to encode small integers is the NIL (New Implementation of Lisp) language done at MIT for the new VAX family of 32-bit processors [3,page 54] in the early seventies. The word layout in NIL is very similar to our two-bit tag scheme:



The NIL language uses a *tag* of 0 to represent 30-bit small integers. When the *tag* is not zero, the full five bits of *utag* and *tag* are used to determine the object type. The NIL implementation was in particular geared to run on stock hardware and became one of the main influences in the design of Common Lisp [3].

Even though systems as early as the Burrough's B5700 in 1969 operated on tagged data, this was only used for a form of ad-hoc polymorphism where a single add instruction could perform either an integer addition or a floating-point addition for example.

The SPARC architecture [17] incorporated tagged arithmetic instructions since its inception in 1987 (called `TADDCC` and `TSUBCC`). Just as we propose in Section 4, these instructions use a 2-bit tag and set the overflow flag if the arguments are not small integers. Unlike our proposal, it uses inverted flag bits as discussed in Section 2.7 where the tag is 0 for small integers and 1 for pointers.

The use of these instructions on SPARC was never popular, although they were used for example by the Franz Allegro Common Lisp system and Lucid Common Lisp. Even though at that time these instructions did not see much use, we believe that nowadays the situation is quite different – in particular widely used languages like JavaScript or Python can directly benefit from such instructions.

The PowerPC architecture has the `rlimn`/`rlwinm` instructions [2, 4], or “Rotate Left Word Immediate Then AND with Mask”. This is ideal for isolating tag bits especially if they are in the top bits as with *reno*. Having such instructions may make using top bits more competitive with the other approaches.

The arm64 architecture also has “feature-full” instructions that help with sign-extended overflow arithmetic. In particular, the compare instruction can directly compare a full register against a sign extended one, and thus implement the test `z!=(int32_t)z` in a single instruction as `cmp x0,w0,sxtw`.

In this article we focused on the use of tagging to provide seamless arbitrary precision integers, but tagging is also used to provide runtime systems with a way to uniformly distinguish pointers from values. This is used for example in the OCaml runtime system [9, page 39] to box values and provides a clean interface to the garbage collector. Even in such environments, dedicated hardware instructions can still be benificial as it allows direct arithmetic on such tagged values (even without checking for overflow or incorrect tags).

6 ACKNOWLEDGEMENTS

The *reno* technique was initially conceived by Renzo Duin and we thank him for the many insightful discussions. We would also like to thank Nikolaj Bjørner, Anton Lorenzen, Michał Moskal, Leonardo de Moura, Madan Musuvathi, and Nikhil Swamy for their encouraging feedback on earlier versions of this article.

7 CONCLUSION AND FUTURE WORK

In this article we show how languages can efficiently support proper *integer* numbers $\mathbb{Z}$, and show three novel ways to improve upon the usual tagged integer arithmetic. Unfortunately, the benchmarks show that the performance penalty is still quite large on modern hardware and we hope that future ISA extensions on common platforms will add direct tagged arithmetic operations; this will benefit not only languages with proper integer support, but also improve performance for dynamic languages like JavaScript and Python.

REFERENCES

[1] Raymond Boute. “The Euclidean Definition of the Functions Div and Mod.” *ACM Transactions on Programming Languages and Systems (TOPLAS)* 14 (April): 127–144. Apr. 1992. doi:10.1145/128861.128862.

[2] Raymond Chen. “The PowerPC 600 Series, Part 5: Rotates and Shifts.” Aug. 2018. `https://devblogs.microsoft.com/oldnewthing/20180810-00/?p=99465`. The Old New Thing blog.

[3] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. Aug. 1985. `https://ia800200.us.archive.org/16/items/PerformanceAndEvaluationOfLispSystems/Performance%20and%20Evaluation%20of%20Lisp%20Systems.pdf`.

[4] IBM. “AIX 7.2, PowerPC Assembler Language Reference.” Jul. 2022. `https://www.ibm.com/docs/en/aix/7.2?topic=is-rlwinm-rlinm-rotate-left-word-immediate-then-mask-instruction`.

[5] *IEEE Std 754-2019 (Revision of IEEE 754-2008)*. “IEEE Standard for Floating-Point Arithmetic,” July, 1–84. Jul. 2019. doi:10.1109/IEEESTD.2019.8766229.

[6] SECBIT Labs. “A Disastrous Vulnerability Found in Smart Contracts of BeautyChain (BEC).” Apr. 2018. `https://medium.com/secbit-media/a-disastrous-vulnerability-found-in-smart-contracts-of-beautychain-bec-dbf24ddbc30e`.

[7] Daan Leijen. “Division and Modulus for Computer Scientists.” University of Utrecht. Mar. 2001. `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/divmodnote-letter.pdf`.

[8] Daan Leijen. “The Koka Language.” 2021. `https://koka-lang.github.io`.

[9] Xavier Leroy. *The ZINC Experiment: An Economical Implementation of the ML Language*. Technical report 117. INRIA. 1990. `https://xavierleroy.org/publi/ZINC.pdf`.

[10] Anton Lorenzen, and Daan Leijen. *Reference Counting with Frame Limited Reuse (extended Version)*. MSR-TR-2021-30. Microsoft Research. Nov. 2021.

[11] Harri Porten, Peter Kelly, and Apple Inc. “JavaScript Core, NuN Boxing.” 1999. `https://github.com/WebKit/WebKit/blob/main/Source/JavaScriptCore/runtime/JSCJSValue.h`.

[12] Lynn H. Quam, and Whitfield Diffie. *STANFORD LISP 1.6 Manual*. Operating note 28.6. Stanford Artificial Intelligence Laboratory. 1966. `http://www.softwarepreservation.net/projects/LISP/stanford/SAILON-28.6.pdf`.

[13] Reinking, Xie, de Moura, and Leijen. “Perceus: Garbage Free Reference Counting with Reuse.” In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 96–111. PLDI 2021. ACK, New York, NY, USA. 2021. doi:10.1145/3453483.3454032.

[14] Colin Runciman. “What about the Natural Numbers?” *Computer Languages* 14 (3): 181–191. 1989. doi:10.1016/0096-0551(89)90004-0.

[15] Sebastian Ullrich, and Leonardo de Moura. “Counting Immutable Beans – Reference Counting Optimized for Purely Functional Programming.” In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL’19)*. Singapore. Sep. 2019.

[16] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. UCB/EECS-2016-1. Electrical Engineering and Computer Sciences, University of California at Berkeley. Jan. 2016. `https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf`.

[17] David L. Weaver, and Tom Germond. *The SPARC Architecture Manual, Version 9*. SAV09R1459912. SPARC International, Inc. 1994. `http://temlib.org/pub/SparcStation/Standards/SparcV9.pdf`.

Created with Madoko.net.