64-bit Integer Division for the JavaScript Platform

Sébastien Doeraene
EPFL
Lausanne, Switzerland
sebastien.doeraene@epfl.ch

Tobias Schlatter
Datahouse
Zürich, Switzerland
tobias.schlatter@datahouse.ch

***Abstract*—The JavaScript platform natively supports binary64 floating point numbers, and, through a twist of fate, 32-bit integer arithmetic. However, it does not offer support for 64-bit integers. As a result, compilers targeting JavaScript have traditionally struggled to offer efficient 64-bit integers. We first survey how existing algorithms can be used to implement most 64-bit integer arithmetic operations. We then propose a new algorithm for 64-bit integer division and remainder, and prove its correctness. It combines 32-bit integer arithmetic and 64-bit floating point operations, and contains no loop. We also present and prove close cousins for constant divisors, and for conversion to string in all radices supported by JavaScript. We implemented these algorithms in the Scala.js compiler. They outperform previously-known algorithms.**

***Index Terms*—JavaScript, integer arithmetic, division, floating-point.**

I. Motivation

The JavaScript platform natively supports binary64 floating point numbers, and, through a twist of fate, 32-bit integer arithmetic. However, it does not offer support for 64-bit integers. Compilers targeting JavaScript have traditionally struggled to offer efficient 64-bit integers. Notable such compilers include emscripten [15], GHCJS [4], J2CL [8], Js_of_ocaml [9], Kotlin/JS [10] and Scala.js [12].

Most of these compilers implement 64-bit integers as instances of a class with 2 (sometimes 3) integer fields. Only emscripten and Scala.js use a flat representation, similar to standard compiler techniques for double-length integers. Even those, however, lack a good division algorithm. The existing algorithms all require loops. They fall in two broad categories:

- shift-and-subtract division, sometimes with an early exit,
- repeated approximation by double division.

Additionally, conversions of 64-bit integers to strings require one or more invocations of the expensive division.

We propose a new algorithm for 64-bit integer division and remainder on the JavaScript platform, and prove its correctness. It combines 32-bit integer arithmetic and 64-bit floating point operations, and contains no loop. We also present and prove close cousins for constant divisors, and for conversion to string in all radices supported by JavaScript. We implemented these algorithms in the Scala.js compiler. They outperform previously-known algorithms for 64-bit integer divisions.

A. Alternatives

When targeting the JavaScript platform, there are two alternatives to the above schemes. Some compilers like Kotlin/JS and Scala.js can be configured to use the alternatives below.

First, we can compile to WebAssembly [13] (Wasm), which natively supports 64-bit integers. That is not always an option, as we may target environments that do not (yet) support Wasm. Moreover, Wasm is not free of performance concerns: interoperability with JavaScript, in particular, is still difficult to implement efficiently. There may be a performance trade-off between efficient interoperability and fast 64-bit integers.

Second, we can represent 64-bit integers as JavaScript `bigint`s. We can force arithmetic modulo 2^{64} by wrapping every operation with `BigInt.asIntN(64, ·)`. Some JavaScript engines, such as V8, produce efficient code for those as long as they are local within a method. When they escape, the requirement for a universal representation prevents the optimizations, and the engines need to allocate expensive instances of big integers on-heap. In practice, the encoding used by emscripten or Scala.js generally outperforms the use of `bigint`s.

It is therefore still important to find good algorithms for lowered 64-bit integers on the JavaScript platform.

B. Outline

Sections II–IV introduce notations and summarize the state of the art on which we build. Sections V–VIII present our contributions: the algorithms for division, constant divisors, and conversion to string, along with their proofs. We evaluate the performance of our division algorithm in Section IX, and conclude in Section X.

II. Notations

Let $\mathbb{F}_{64}$ be the set of binary64 values, and $\mathrm{RN} : \mathbb{R} \to \mathbb{F}_{64}$ be a function that rounds a real number to the nearest member of $\mathbb{F}_{64}$, with a tie-breaking rule. In JavaScript, all floating point operations break ties "to even", although our algorithms do not require that specific rule.

Throughout this paper, we canonically represent fixed-size integers as values of $\mathbb{N}$ corresponding to their *unsigned* interpretation.

We define the following functions for $a, b \in \mathbb{R}$, $b > 0$:

$$\mathrm{div}(a, b) = \lfloor a/b \rfloor ,$$
$$\mathrm{rem}(a, b) = a - b \cdot \mathrm{div}(a, b) .$$

We always have $0 \leq \mathrm{rem}(a, b) < b$. When a and b are non-negative integers that fit in a particular bit width, those operations correspond to unsigned division and remainder.

For $a \in \mathbb{Z}$ and $b \in \mathbb{N}, b > 0$, we have

$$a \bmod b = a - b \cdot \lfloor a/b \rfloor = \mathrm{rem}(a, b)\,.$$

We write $+_n$, $-_n$ and $\times_n$ for the n-bit modular integer operations:

$$a +_n b = a + b \bmod 2^n,$$
$$a -_n b = a - b \bmod 2^n,$$
$$a \times_n b = a \cdot b \bmod 2^n.$$

We use &, |, ^ and ~ for bitwise operations. We write $\lll$, $\ggg$ and $\gg$ for shift left, zero-extending shift right and sign-extending shift right, respectively. The bit width applied for these operations is inferred from the context. It is usually 32.

When we need to interpret an n-bit integer as a signed 2's complement number, we explicitly use the following function:

$$\mathrm{S}_n(a) = \begin{cases} a & \text{if } a < 2^{n-1} \\ a - 2^n & \text{otherwise.} \end{cases}$$

We use the truncating semantics for signed division of n-bit integers. That is consistent with what we naturally get in JavaScript for 32-bit integers, as well as with Scala semantics.

$$\mathrm{truncate}(x) = \begin{cases} \lfloor x \rfloor & \text{if } x \geq 0 \\ \lceil x \rceil & \text{if } x < 0\,, \end{cases}$$
$$\mathrm{sdiv}_n(a, b) = \mathrm{truncate}(\mathrm{S}_n(a) / \mathrm{S}_n(b)) \bmod 2^n,$$
$$\mathrm{srem}_n(a, b) = a -_n (b \times_n \mathrm{sdiv}_n(a, b))\,.$$

As defined, and as is the case in JavaScript and Scala, dividing -2^{n-1} by -1 is well defined and overflows back to -2^{n-1}.

III. 32-bit arithmetic in JavaScript

Throughout the paper, we assume that we have efficient 32-bit integer arithmetic at our disposal in JavaScript. From the specification of ECMAScript alone [3], it is not clear that this is the case. In this section, we briefly summarize the encoding of 32-bit integers used by compilers targeting JavaScript.

Implementation of 32-bit arithmetic in JavaScript has been well-known since its "discovery" in `asm.js` [7]. While `asm.js` itself was not widely adopted, its idioms to implement 32-bit arithmetic are recognized by modern JavaScript JITs (Just-In-Time compilers), and optimized as such.[1]

The `asm.js`-style encoding we use represents 32-bit integers as binary64 values equal to their *signed* interpretation (i.e., $\mathrm{S}_{32}(\cdot)$). Given that uniform representation, Table I summarizes relevant operations. For clarity, we explicitly write $\mathrm{FP}(a)$ and $\mathrm{FP}(\mathrm{S}_{32}(a))$ as abstract operations converting 32-bit integers to binary64 values. Those conversions are always exact, and therefore identities in $\mathbb{R}$.

`x>>>0` and `x|0` respectively compute $\mathrm{ToUint32}(x)$ and $\mathrm{ToInt32}(x)$, defined in [3] as

$$\mathrm{ToUint32}(x) = \mathrm{truncate}(x) \bmod 2^{32},$$
$$\mathrm{ToInt32}(x) = \mathrm{S}_{32}(\mathrm{ToUint32}(x))\,.$$

[1] It is difficult to find an authoritative source for this claim. However, in practice, (incorrectly) compiling 32-bit operations as the supposedly more primitive `a+b`, `a-b` and `a*b` makes run-time performance *worse*.

TABLE I
32-bit Integer Arithmetic in JavaScript

Abstract operation	JavaScript encoding
$a +_{32} b$	`(a+b)|0`
$a -_{32} b$	`(a-b)|0`
$a \times_{32} b$	`Math.imul(a, b)`
$\mathrm{div}(a, b)$	`((a>>>0) / (b>>>0))|0`
$\mathrm{rem}(a, b)$	`((a>>>0) % (b>>>0))|0`
$\mathrm{sdiv}_{32}(a, b)$	`(a / b)|0`
$\mathrm{srem}_{32}(a, b)$	`(a % b)|0`
$\mathrm{FP}(a)$	`a>>>0`
$\mathrm{FP}(\mathrm{S}_{32}(a))$	`a`
$\mathrm{truncate}(x) \bmod 2^{32}$	`x|0`

a and b are 32-bit integers. They are unsigned on the left, and signed on the right. x is a binary64 value.

They truncate and wrap their argument back into the unsigned (resp. signed) 32-bit integer range. In particular, for $x \in \mathbb{F}_{64}, 0 \leq x < 2^{32}$, the operation `x|0` computes the encoded ($\mathrm{S}_{32}(\cdot)$) value of

$$\mathrm{ToUint32}(x) = \mathrm{truncate}(x) \bmod 2^{32} = \lfloor x \rfloor \bmod 2^{32} = \lfloor x \rfloor$$

and for $m \in \mathbb{Z}$, `m|0` computes the encoded value of

$$\mathrm{ToUint32}(m) = \mathrm{truncate}(m) \bmod 2^{32} = m \bmod 2^{32}.$$

For integer multiplication, using `(a*b)|0` would be incorrect, as the intermediate `a*b` looses precision in the low-order bits. ECMAScript 2015 added the builtin function `Math.imul` to fill that gap.

IV. Basic 64-bit integer arithmetic

Now that we have established that we have efficient 32-bit integers, we can build 64-bit integers on top of them. All the puzzle pieces in this section have been well-known in the compiler literature. We recall them here because they are rarely used in compilers targeting JavaScript.

We represent a 64-bit integer x as a sum $2^{32}x_h + x_\ell$, where x_h and x_ℓ are 32-bit integers. We write (x_h, x_ℓ) as a short-hand. The canonical representation is unsigned in this presentation, as usual. The signed interpretation is

$$\mathrm{S}_{64}(x) = \mathrm{S}_{64}\big(2^{32}x_h + x_\ell\big) = 2^{32} \cdot \mathrm{S}_{32}(x_h) + x_\ell\,. \quad (1)$$

Indeed, $x < 2^{63}$ iff $x_h < 2^{31}$. If $x < 2^{63}$, we have

$$\mathrm{S}_{64}\big(2^{32}x_h + x_\ell\big) = 2^{32}x_h + x_\ell = 2^{32} \cdot \mathrm{S}_{32}(x_h) + x_\ell\,.$$

Otherwise, we have

$$\begin{aligned} \mathrm{S}_{64}\big(2^{32}x_h + x_\ell\big) &= 2^{32}x_h + x_\ell - 2^{64} \\ &= 2^{32} \cdot \big(x_h - 2^{32}\big) + x_\ell = 2^{32} \cdot \mathrm{S}_{32}(x_h) + x_\ell\,. \end{aligned}$$

A. Modular operations

Algorithms 1 and 2 show how to implement double-length addition and subtraction. The carry bit is computed with a comparison whose result is interpreted as a 32-bit integer : 1 for true and 0 for false. In JavaScript, we can get that interpretation with `((sl>>>0) < (al>>>0))|0`.

Algorithm 1 ADD64 – 64-bit addition

Input: $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$
Output: $s = (s_h, s_\ell) = a +_{64} b$
$s_\ell \leftarrow a_\ell +_{32} b_\ell$
$s_h \leftarrow a_h +_{32} b_h +_{32} (s_\ell < a_\ell)$

Algorithm 2 SUB64 – 64-bit subtraction

Input: $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$
Output: $d = (d_h, d_\ell) = a -_{64} b$
$d_\ell \leftarrow a_\ell -_{32} b_\ell$
$d_h \leftarrow a_h -_{32} b_h -_{32} (d_\ell > a_\ell)$

Modular multiplication of two 64-bit integers can be decomposed into their 32-bit components. Let $c = a_\ell b_\ell = 2^{32} c_h + c_\ell$ be the full product of the low words. $c_\ell = a_\ell \times_{32} b_\ell$. c_h is the high-order half of the product, for which Hacker's Delight [14] gives an algorithm in section 8-2. This leads to Algorithm 3.

Algorithm 3 MUL64 – 64-bit multiplication

Input: $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$
Output: $p = (p_h, p_\ell) = a \times_{64} b$
{Compute c_h with the algorithm from Hacker's Delight}
$a_0 \leftarrow a_\ell \,\&(2^{16} - 1)$; $a_1 \leftarrow a_\ell \ggg 16$
$b_0 \leftarrow b_\ell \,\&(2^{16} - 1)$; $b_1 \leftarrow b_\ell \ggg 16$
$c_{00} \leftarrow a_0 \times_{32} b_0$; $c_{10} \leftarrow a_1 \times_{32} b_0$
$c_{01} \leftarrow a_0 \times_{32} b_1$; $c_{11} \leftarrow a_1 \times_{32} b_1$
$t \leftarrow c_{10} +_{32} (c_{00} \ggg 16)$
$c_h \leftarrow c_{11} +_{32} (t \ggg 16) +_{32} ((c_{01} +_{32} (t \,\&(2^{16}-1))) \ggg 16)$
{Combine c_h with other subproducts to compute p}
$p_h \leftarrow (a_\ell \times_{32} b_h) +_{32} (a_h \times_{32} b_\ell) +_{32} c_h$
$p_\ell \leftarrow a_\ell \times_{32} b_\ell$

The last line can be replaced by

$$p_\ell \leftarrow c_{00} +_{32} ((c_{10} +_{32} c_{01}) \ll 16) .$$

It reuses intermediate products, trading one 32-bit multiplication for two additions and a shift, which may be beneficial.

B. Comparisons

Comparisons usually require two branches, whether signed or unsigned. However, some special cases can be tested more efficiently. We will use these special cases throughout the rest of the paper and assume the efficient replacements.

$$\begin{aligned} x \geq 2^{63} &\equiv \quad \mathrm{S}_{64}(x) < 0 \quad \equiv \mathrm{S}_{32}(x_h) < 0 \\ x < 2^{32} &\equiv \quad\quad x = x_\ell \quad\quad \equiv x_h = 0 \\ -2^{31} \leq \mathrm{S}_{64}(x) < 2^{31} &\equiv \mathrm{S}_{64}(x) = \mathrm{S}_{32}(x_\ell) \equiv x_h = x_\ell \gg 31 \end{aligned}$$

For an integer $0 \leq k < 32$, $m = {\sim}(2^k - 1)$:

$$x < 2^k \equiv x_h \,|\, (x_\ell \,\&\, m) = 0$$

For an integer $32 \leq k < 64$, $m = {\sim}(2^{k-32} - 1)$, $s = 63 - k$:

$$\begin{aligned} x < 2^k &\equiv x_h \,\&\, m = 0 \\ -2^k \leq \mathrm{S}_{64}(x) < 2^k &\equiv (x_h \,\hat{}\, (x_h \gg s)) \,\&\, m = 0 \end{aligned}$$

V. CONVERSIONS

This concludes our survey of well-known algorithms and how we can leverage them within the constraints of JavaScript. We now introduce algorithms that, to the best of our knowledge, have not been formally studied before.

Our division algorithms and conversions to strings will exploit the binary64 division, by converting 64-bit integers to binary64 values and back. It is therefore important to get good code for these conversions.

A. Conversions to binary64 values

Converting a 64-bit integer to a binary64 value using the unsigned interpretation is done as

$$\begin{aligned} \mathrm{RN}(x) &= \mathrm{RN}\big(2^{32} x_h + x_\ell\big) \\ &= \mathrm{RN}\big(\mathrm{RN}\big(2^{32} \cdot \mathrm{FP}(x_h)\big) + \mathrm{FP}(x_\ell)\big) , \end{aligned}$$

which gives the code 2^{32}`*(xh>>>0) + (xl>>>0)`. For the signed interpretation, (1) leads to

$$\begin{aligned} \mathrm{RN}(\mathrm{S}_{64}(x)) &= \mathrm{RN}\big(2^{32} \cdot \mathrm{S}_{32}(x_h) + x_\ell\big) \\ &= \mathrm{RN}\big(\mathrm{RN}\big(2^{32} \cdot \mathrm{FP}(\mathrm{S}_{32}(x_h))\big) + \mathrm{FP}(x_\ell)\big) , \end{aligned}$$

which gives the code 2^{32}`*xh + (xl>>>0)`.

B. Unsigned conversion from binary64 values

Given $x \in \mathbb{F}_{64}$, $0 \leq x < 2^{64}$, we are interested in finding (y_h, y_ℓ) such that $y = \lfloor x \rfloor$.

If we have access to the binary64 bit pattern of the input, we can use integer arithmetic operations as usual. In JavaScript, while we can extract the bits of an $\mathbb{F}_{64}$ using a scratch `DataView`, it is not always efficient, depending on the engine.

We use the JavaScript instructions `yh = (x*`2^{-32}`)|0` and `yl = x|0`, which compute

$$\begin{aligned} y_h &= \mathrm{truncate}\big(\mathrm{RN}\big(x \cdot 2^{-32}\big)\big) \bmod 2^{32} = \big\lfloor x/2^{32} \big\rfloor \bmod 2^{32}, \\ y_\ell &= \mathrm{truncate}(x) \bmod 2^{32} = \lfloor x \rfloor \bmod 2^{32} \end{aligned}$$

(even if x or $\mathrm{RN}\big(x \cdot 2^{-32}\big)$ are subnormal values).

Lemma 1. *Given* $x \in \mathbb{R}$, $0 \leq x < 2^{64}$, $y_h = \lfloor x/2^{32} \rfloor \bmod 2^{32}$ *and* $y_\ell = \lfloor x \rfloor \bmod 2^{32}$, *we have* $y = 2^{32} y_h + y_\ell = \lfloor x \rfloor$.

Proof. For all $a \in \mathbb{R}$, $b \in \mathbb{N}$, $b > 0$, we have

$$\lfloor a \rfloor \bmod b = \lfloor a \rfloor - b \cdot \lfloor \lfloor a \rfloor / b \rfloor = \lfloor a \rfloor - b \cdot \lfloor a/b \rfloor \quad (2)$$

where $\lfloor \lfloor a \rfloor / b \rfloor = \lfloor a/b \rfloor$ follows from Theorem 3.11 in [5]. Using (2) twice, and $x < 2^{64}$, we get

$$\begin{aligned} y_\ell &= \lfloor x \rfloor \bmod 2^{32} \\ &= \lfloor x \rfloor - 2^{32} \cdot \big\lfloor x/2^{32} \big\rfloor , \\ y_h &= \big\lfloor x/2^{32} \big\rfloor \bmod 2^{32} = \big\lfloor x/2^{32} \big\rfloor - 2^{32} \cdot \big\lfloor \big(x/2^{32}\big)/2^{32} \big\rfloor \\ &= \big\lfloor x/2^{32} \big\rfloor , \\ y &= 2^{32} y_h + y_\ell = 2^{32} \cdot \big\lfloor x/2^{32} \big\rfloor + \lfloor x \rfloor - 2^{32} \cdot \big\lfloor x/2^{32} \big\rfloor \\ &= \lfloor x \rfloor . \end{aligned}$$

□

VI. Division algorithm

We now present the main contribution of this paper: our division algorithm. We focus on unsigned division for most of this section, and briefly cover signed division in Section VI-C.

Given two 64-bit integers $a = (a_h, a_\ell)$ and $b = (b_h, b_\ell)$, we want to compute $q = \operatorname{div}(a, b)$ and $r = \operatorname{rem}(a, b)$.

We divide the algorithm into three cases, depending on the magnitude of b: $0 < b < 2^{21}$, $2^{21} \le b < 2^{63}$ and $2^{63} \le b < 2^{64}$. We present and prove the first two subalgorithms in separate sections. The third one is straightforward.

We will use the following properties.

Property 1. *For $x \in \mathbb{R}$, $y \in \mathbb{F}_{64}$, if $x \le y$ then $\mathrm{RN}(x) \le y$, and if $x \ge y$, then $\mathrm{RN}(x) \ge y$.*

Proof. Immediate from $\mathrm{RN}(y) = y$ and the weak monotonicity of RN. □

Property 2. *For $x, y \in \mathbb{N}$, $x, y \le 2^{53}$, $y \ne 0$, we have $\lfloor \mathrm{RN}(x/y) \rfloor = \lfloor x/y \rfloor$.*

Proof. Corollary of Theorem 1 in [11]. □

Property 3. *For $x, y \in \mathbb{R}$, $\hat{x} = \mathrm{RN}(x)$ and $\hat{y} = \mathrm{RN}(y)$, assuming no overflow, underflow or subnormal, we have*

$$\left| \frac{\mathrm{RN}(\hat{x} \cdot \hat{y}) - xy}{xy} \right| < 2^{-51}, \qquad \left| \frac{\mathrm{RN}(\hat{x}/\hat{y}) - x/y}{x/y} \right| < 2^{-51}.$$

Proof. Corollary of [1], section I.B, which shows the stricter bounds $< 3u$ for xy and $3u + u^2$ for x/y, with $u = 2^{-53}$. □

Property 4. *For $x, y \in \mathbb{R}$, $n \in \mathbb{N}$, if $0 \le x - y \le n$, then $0 \le \lfloor x \rfloor - \lfloor y \rfloor \le n$.*

Proof. Recall that the floor function is weakly monotonic. Since $x \ge y$, we have $\lfloor x \rfloor \ge \lfloor y \rfloor$ and $\lfloor x \rfloor - \lfloor y \rfloor \ge 0$. From $x \le n + y$, we get $\lfloor x \rfloor \le \lfloor n + y \rfloor = n + \lfloor y \rfloor$, hence $\lfloor x \rfloor - \lfloor y \rfloor \le n$. □

Property 5. *For $a, b, c, n \in \mathbb{Z}$, $x = a - b \cdot c$, if $0 \le x < 2^n$ (i.e., x fits in an unsigned n-bit integer), then $x = a -_n (b \times_n c)$. If $-2^{n-1} \le x < 2^{n-1}$ (i.e., x fits in a signed n-bit integer), then $x = \mathrm{S}_n(a -_n (b \times_n c))$.*

Proof. The unsigned case follows from $x = x \bmod 2^n$ and elementary modular arithmetics. The signed case follows from $x = \mathrm{S}_n(x \bmod 2^n)$. □

A. Case $0 < b < 2^{21}$

We use Algorithm 4. It essentially follows a 2-step short division algorithm in base 2^{32}, with some care when using floating point operations.

Algorithm 4 DIV64-CASE1 – division for $0 < b < 2^{21}$

Input: $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $0 < b < 2^{21}$
Output: $q = (q_h, q_\ell) = \operatorname{div}(a, b)$, $r = (r_h, r_\ell) = \operatorname{rem}(a, b)$
  Assert: $b_h = 0$.
  $q_h \leftarrow \operatorname{div}(a_h, b_\ell)$
  $k \leftarrow a_h -_{32} (b_\ell \times_{32} q_h)$
  $a' \leftarrow (k, a_\ell) = 2^{32}k + a_\ell$
  $q_0 \leftarrow \mathrm{RN}(\mathrm{RN}(a') / \mathrm{FP}(b_\ell))$
  Assert: $0 \le q_0 < 2^{32}$
  $q_\ell \leftarrow \lfloor q_0 \rfloor$
  $r_h \leftarrow 0$
  $r_\ell \leftarrow a_\ell -_{32} (b_\ell \times_{32} q_\ell)$

Theorem 1. *Given $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $0 < b < 2^{21}$, Algorithm 4 computes $q = (q_h, q_\ell) = \operatorname{div}(a, b)$ and $r = (r_h, r_\ell) = \operatorname{rem}(a, b)$.*

Proof. We have $b = b_\ell$ and $b_h = 0$.

Since $0 \le \operatorname{rem}(a_h, b) < b < 2^{32}$, by Property 5 we have

$$k = a_h -_{32} (b_\ell \times_{32} \operatorname{div}(a_h, b_\ell)) = \operatorname{rem}(a_h, b)\,. \tag{3}$$

When only the remainder is required, this property can be used to replace the first two lines of the algorithm by

$$k \leftarrow \operatorname{rem}(a_h, b_\ell)\,.$$

Since $k < b$, we have $a' = 2^{32}k + a_\ell < 2^{32} \cdot (k+1) \le 2^{32}b$. Hence, $a' < 2^{32+21} = 2^{53}$, and $a' \in \mathbb{F}_{64}$. $\mathrm{RN}(a') = a'$ and $\mathrm{FP}(b_\ell) = b_\ell = b$. This step of the proof is the reason we must have $b < 2^{21}$ in this subalgorithm, with $21 = 53 - 32$.

We now have $q_0 = \mathrm{RN}(\mathrm{RN}(a') / \mathrm{FP}(b_\ell)) = \mathrm{RN}(a'/b)$. Since $0 \le a', b < 2^{53}$, from Property 2, we get

$$q_\ell = \lfloor q_0 \rfloor = \lfloor \mathrm{RN}(a'/b) \rfloor = \lfloor a'/b \rfloor\,. \tag{4}$$

Using that equality and the fact that $0 \le a'/b < 2^{32}b/b = 2^{32}$, we know that $0 \le q_0 < 2^{32}$. Therefore, we know how to compute $\lfloor q_0 \rfloor$ using the 32-bit arithmetic from Section III.

Using (3) and (4), we can derive

$$a' = 2^{32}k + a_\ell = \left(2^{32}a_h + a_\ell\right) - 2^{32} \cdot b \cdot \operatorname{div}(a_h, b)$$
$$q_\ell = \lfloor a'/b \rfloor = \lfloor a/b \rfloor - 2^{32} \cdot \operatorname{div}(a_h, b)$$

and conclude that

$$q = 2^{32} \cdot \operatorname{div}(a_h, b) + \left(\lfloor a/b \rfloor - 2^{32} \cdot \operatorname{div}(a_h, b)\right) = \operatorname{div}(a, b)\,.$$

Finally, since $\operatorname{rem}(a, b) < b < 2^{32}$, by Property 5 we have

$$r = 2^{32} \cdot 0 + r_\ell = a_\ell -_{32} (b_\ell \times_{32} q_\ell) = \operatorname{rem}(a, b)\,.$$ □

B. Case $2^{21} \le b < 2^{63}$

In this case, we compute an approximation of the quotient using double division, where we approximate the dividend and divisor as binary64 values. We then compute the approximated remainder associated to that approximated quotient (using 64-bit integer operations for the definition of remainder), and use it to correct the result.

We use Algorithm 5. We prove it with the weaker requirement $b \le 2^{63}$, which is useful for signed division.

Theorem 2. *Given $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $2^{21} \le b \le 2^{63}$, Algorithm 5 computes $q = (q_h, q_\ell) = \operatorname{div}(a, b)$ and $r = (r_h, r_\ell) = \operatorname{rem}(a, b)$.*

Proof. If $a = 0$, it is straightforward to verify that $\hat{q} = 0$ and $\hat{r} = a = 0$. Therefore $\mathrm{S}_{64}(\hat{r}) < 0$ is false. The algorithm takes the **else** branch, where $q = \hat{q} = 0$ and $r = \hat{r} = 0$, as desired.

Algorithm 5 DIV64-CASE2 – division for $2^{21} \leq b \leq 2^{63}$

Input: $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $2^{21} \leq b \leq 2^{63}$
Output: $q = (q_h, q_\ell) = \mathrm{div}(a, b)$, $r = (r_h, r_\ell) = \mathrm{rem}(a, b)$

$\hat{a} \leftarrow \mathrm{RN}(a)$
$\hat{b} \leftarrow \mathrm{RN}(b)$
$\hat{q}_0 \leftarrow \mathrm{RN}(\hat{a}/\hat{b})$
$\hat{q}_1 \leftarrow \mathrm{RN}\big(\hat{q}_0 + 2^{-8}\big)$
Assert: $0 \leq \hat{q}_1 < 2^{64}$
$\hat{q} = (\hat{q}_h, \hat{q}_\ell) \leftarrow \lfloor \hat{q}_1 \rfloor$
$\hat{r} = (\hat{r}_h, \hat{r}_\ell) \leftarrow a -_{64} (b \times_{64} \hat{q})$
if $\mathrm{S}_{64}(\hat{r}) < 0$ **then**
  $q \leftarrow \hat{q} -_{64} 1$
  $r \leftarrow \hat{r} +_{64} b$
else
  $q \leftarrow \hat{q}$
  $r \leftarrow \hat{r}$
end if

For the remainder of the proof, we have $1 \leq a < 2^{64}$. Hence, $2^{-63} = 1/2^{63} \leq a/b < 2^{64}/2^{21} = 2^{43}$.

From Property 1, $1 \leq a < 2^{64}$ and $2^{21} \leq b \leq 2^{63}$, we have $1 \leq \hat{a} \leq 2^{64}$ and $2^{21} \leq \hat{b} \leq 2^{63}$. Therefore, $2^{-63} = 1/2^{63} \leq \hat{a}/\hat{b} \leq 2^{64}/2^{21} = 2^{43}$, and $2^{-63} \leq \hat{q}_0 = \mathrm{RN}(\hat{a}/\hat{b}) \leq 2^{43}$.

That means that no overflow nor underflow happens during the computation of $\hat{q}_0$, and that no subnormals are involved. Using Property 3, we have

$$\left| \frac{\mathrm{RN}(\hat{a}/\hat{b}) - a/b}{a/b} \right| < 2^{-51},$$

hence

$$|\hat{q}_0 - a/b| < 2^{-51} \cdot a/b < 2^{-51} \cdot 2^{43} = 2^{-8}.$$

If $\hat{q}_0 < 2^{-7}$, it follows from the above bound that $a/b < 1$ hence $\mathrm{div}(a, b) = \lfloor a/b \rfloor = 0$. Further, $\hat{q}_1 < 1$ and hence $\hat{q} = 0$.

Otherwise, $2^{-7} \leq \hat{q}_0 \leq 2^{43}$. In that case, the floating-point addition $\mathrm{RN}\big(\hat{q}_0 + 2^{-8}\big)$ is exact. Therefore, $\hat{q}_1 = \hat{q}_0 + 2^{-8}$. We bound $\hat{q}_0 - a/b$ on the negative and positive sides as

$$\begin{aligned} -2^{-8} < \hat{q}_0 - a/b &< 2^{-8} \\ 0 < \hat{q}_0 - a/b + 2^{-8} &< 2 \cdot 2^{-8} \\ 0 < \hat{q}_1 - a/b &< 2^{-7}. \end{aligned}$$

By Property 4, it follows that

$$0 \leq \lfloor \hat{q}_1 \rfloor - \lfloor a/b \rfloor = \hat{q} - \mathrm{div}(a, b) \leq 1.$$

We know how to compute $\mathrm{RN}(a)$ and $\mathrm{RN}(b)$ using the algorithm of Section V. Likewise, since $2^{-8} \leq \hat{q}_1 \leq 2^{43} + 2^{-8}$, we know how to compute a 64-bit integer $\hat{q} = (\hat{q}_h, \hat{q}_\ell) = \lfloor \hat{q}_1 \rfloor$.

Since $\hat{q}$ and $\mathrm{div}(a, b)$ are integers, $\hat{q} - \mathrm{div}(a, b) \in \{0, 1\}$. Moreover, $-2^{63} \leq -b \leq a - b \cdot \hat{q} < b \leq 2^{63}$. By Property 5,

$$a - b \cdot \hat{q} = \mathrm{S}_{64}(a -_{64} (b \times_{64} \hat{q})) = \mathrm{S}_{64}(\hat{r}).$$

We can therefore use the estimated remainder $\hat{r}$, interpreted as a signed integer, to determine whether $\hat{q}$ was off by one, and fix the result accordingly. □

Improvement: If $b \leq 2^{31}$, we get $-2^{31} \leq a - b \cdot \hat{q} < 2^{31}$. We can then compute $\hat{r}$ as a signed *32-bit* integer instead:

$$\hat{r} \leftarrow a_\ell -_{32} (b_\ell \times_{32} \hat{q}_\ell).$$

This is particularly useful when the divisor is constant (see Algorithm 9), in which case we do not need an additional branch to leverage this improvement.

We can extend the range of validity of the algorithm down to $2^{14} \leq b$, by increasing the constant 2^{-8} up to 2^{-1}. However, it is desirable for that constant to be as small as possible. That makes it more likely that we take the **else** branch. In addition to being less expensive, it makes it more predictable. Since the cases $b < 2^{21}$ are covered by Algorithm 4, we choose the smallest constant to cover $2^{21} \leq b$.

C. Putting it all together

Divisors greater than 2^{63} defeat the correction code of Algorithm 5. Handling them is however straightforward. We integrate them directly into Algorithm 6, which implements the full unsigned division. It dispatches on the various algorithms based on the scale of b.

Algorithm 6 DIV64 – full unsigned 64-bit division

Input: $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $b \neq 0$
Output: $q = (q_h, q_\ell) = \mathrm{div}(a, b)$, $r = (r_h, r_\ell) = \mathrm{rem}(a, b)$

if $b < 2^{21}$ **then**
  $(q, r) \leftarrow$ DIV64-CASE1(a, b)
else if $b < 2^{63}$ **then**
  $(q, r) \leftarrow$ DIV64-CASE2(a, b)
else if $a \geq b$ **then**
  $q \leftarrow 1$; $r \leftarrow a -_{64} b$
else
  $q \leftarrow 0$; $r \leftarrow a$
end if

Algorithm 7 implements signed division. It is straightforward: compute the absolute value of the operands, perform unsigned division, and fix the sign of the result. By construction, $|\mathrm{S}_{64}(b)| \leq 2^{63}$. Since Algorithm 5 accepts the case $b = 2^{63}$, we can avoid the second test and dispatch only between the first two cases. Moreover, we do not need a special case for dividing -2^{63} by -1. One can verify that it overflows back to -2^{63}, as specified.

Algorithm 7 SDIV64 – full signed 64-bit division

Input: $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $b \neq 0$
Output: $q = \mathrm{sdiv}_{64}(a, b)$, $r = \mathrm{srem}_{64}(a, b)$

$a' \leftarrow |\mathrm{S}_{64}(a)|$; $b' \leftarrow |\mathrm{S}_{64}(b)|$
if $b' < 2^{21}$ **then**
  $(q', r') \leftarrow$ DIV64-CASE1(a', b')
else
  $(q', r') \leftarrow$ DIV64-CASE2(a', b')
end if
$q \leftarrow$ **if** $\mathrm{S}_{64}(a \mathbin{\hat{}} b) < 0$ **then** $0 -_{64} q'$ **else** q'
$r \leftarrow$ **if** $\mathrm{S}_{64}(a) < 0$ **then** $0 -_{64} r'$ **else** r'

VII. Division by a constant

When the divisor is a constant, optimizing compilers typically rewrite divisions using shifts, additions and multiplications. For divisors that are powers of 2, straightforward application of the techniques found in [14] yields very efficient code with our 64-bit integers, including for signed division. For non-powers of 2, it is possible to use the general techniques described in [6]. However, they require computing the upper half of a $64 \times 64 \to 128$-bit multiplication, which is costly. We present variants of Algorithms 4 and 5 that are shorter and more efficient. We can select the appropriate one at compile time, based on the scale of b, and inline it.

A. Case $0 < b < 2^{18}$

When $0 < b < 2^{18}$, we can replace the floating-point division of Algorithm 4 by a multiplication, using the procedure in [6], Section 7. They show that if a and b are n-bit integers, and we have a floating-point system with $F \geq n + 3$ bits of mantissa ($F = 53$ for binary64), then

$$\lfloor a/b \rfloor = \lfloor \mathrm{RN}(a \cdot \hat{m}) \rfloor \text{ given } \hat{m} = \mathrm{RN}\left(\frac{1 + 2^{2-F}}{b}\right). \quad (5)$$

We precompute $\hat{m}$, then use Algorithm 8.

Algorithm 8 Div64-Case1-Const – division for a constant $0 < b < 2^{18}$

Input: $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $0 < b < 2^{18}$, $\hat{m} \in \mathbb{F}_{64}$
Output: $q = (q_h, q_\ell) = \mathrm{div}(a, b)$, $r = (r_h, r_\ell) = \mathrm{rem}(a, b)$
As Algorithm 4, except for the computation of q_0:
$q_0 \leftarrow \mathrm{RN}(\mathrm{RN}(a') \cdot \hat{m})$

Theorem 3. *Given $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $0 < b < 2^{18}$, and $\hat{m} = \mathrm{RN}((1+2^{-51})/\,\mathrm{FP}(b))$, Algorithm 8 computes $q = (q_h, q_\ell) = \mathrm{div}(a, b)$ and $r = (r_h, r_\ell) = \mathrm{rem}(a, b)$.*

Proof. Similar to Theorem 1. Here, $a' < 2^{32+18} = 2^{50}$. Therefore, a' and b are 50-bit integers. We apply (5) with $F = 53$ and $n = 50$ to derive

$$q_\ell = \lfloor q_0 \rfloor = \lfloor a' \cdot \hat{m} \rfloor = \lfloor a'/b \rfloor .$$

The rest of the proof is the same as in Theorem 1. □

Note that we leave $q_h \leftarrow \mathrm{div}(a_h, b_\ell)$ as is in the algorithm. JavaScript engines can recognize the integer division by the constant b_ℓ, and optimize it with the general technique of [6].[2]

B. Case $2^{18} \leq b < 2^{63}$

We use a variant of Algorithm 5. Instead of computing $\mathrm{RN}(\mathrm{RN}(\hat{a}/\hat{b}) + 2^{-8})$, we can precompute a well-chosen constant $\hat{m}$ and then perform a single multiplication $\mathrm{RN}(\hat{a} \cdot \hat{m})$ at run-time.

Choose $k \in \mathbb{N}, k \geq 14$ such that $m = 1/b + 2^{-51-k} \leq 2^{-k}$ (that requires $b > 2^{14}$). Precompute $\hat{m} = \mathrm{RN}(m)$, using extended precision in the intermediate computations. Then use Algorithm 9.

[2]For example: V8 (https://tinyurl.com/v8-div-const) and SpiderMonkey (https://tinyurl.com/sm-div-const) perform that optimization.

Algorithm 9 Div64-Case2-Const – division for a constant $2^{14} < b \leq 2^{63}$

Input: $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $2^{14} < b \leq 2^{63}$, $\hat{m} \in \mathbb{F}_{64}$
Output: $q = (q_h, q_\ell) = \mathrm{div}(a, b)$, $r = (r_h, r_\ell) = \mathrm{rem}(a, b)$
$\hat{a} \leftarrow \mathrm{RN}(a)$
$\hat{q}_1 \leftarrow \mathrm{RN}(\hat{a} \cdot \hat{m})$
Then as in Algorithm 5. We do not compute $\hat{b}$ nor $\hat{q}_0$.

Theorem 4. *Given $a = (a_h, a_\ell)$, $b = (b_h, b_\ell)$, $2^{14} < b \leq 2^{63}$, $k \in \mathbb{N}, k \geq 14$ such that $m = 1/b + 2^{-51-k} \leq 2^{-k}$, $\hat{m} = \mathrm{RN}(m)$, Algorithm 9 computes $q = (q_h, q_\ell) = \mathrm{div}(a, b)$ and $r = (r_h, r_\ell) = \mathrm{rem}(a, b)$.*

Proof. The case $a = 0$ is trivial. We assume $1 \leq a < 2^{64}$.

Since $1/b \geq 2^{-63}$, we have $m > 2^{-63}$. Together with $m \leq 2^{-k}$, we get $2^{-63} < am < 2^{64-k}$. We also have $2^{-63} \leq \hat{m} \leq 2^{-k}$, and $2^{-63} \leq \mathrm{RN}(\hat{a} \cdot \hat{m}) \leq 2^{64-k}$.

$\hat{q}_1 = \mathrm{RN}(\hat{a} \cdot \hat{m})$ is an approximation of am. No overflow, underflow or subnormal is involved in its computation, therefore Property 3 gives us

$$\left| \frac{\mathrm{RN}(\hat{a} \cdot \hat{m}) - am}{am} \right| < 2^{-51},$$

hence

$$|\hat{q}_1 - am| < 2^{-51} \cdot am \leq 2^{-51} a \cdot 2^{-k} = 2^{-51-k} a\,,$$

$$\begin{aligned} -2^{-51-k}a &< \hat{q}_1 - am && < 2^{-51-k}a \\ -2^{-51-k}a &< \hat{q}_1 - a/b - 2^{-51-k}a && < 2^{-51-k}a \\ 0 &< \hat{q}_1 - a/b && < 2 \cdot 2^{-51-k}a\,. \end{aligned}$$

Since $a < 2^{64}$ and $k \geq 14$, we get

$$0 < \hat{q}_1 - a/b < 2^{-50-k}a < 2^{-50-14} \cdot 2^{64} = 1$$

and from Property 4, we get $0 \leq \hat{q} - \mathrm{div}(a, b) \leq 1$ as before. The rest of the algorithm and the proof are as in Theorem 2. □

Under the conditions of the theorem, any choice of k is valid, though it is desirable to choose k as large as possible to minimize the offset 2^{-51-k} (minimizing the chance that a correction is necessary). From the constraint $1/b + 2^{-51-k} \leq 2^{-k}$, we can show that $k > \lfloor \log_2 b \rfloor$ is never valid, and that $14 \leq k \leq \lfloor \log_2 b \rfloor - 1$ is always valid. That makes the latter bound a near-optimal choice that is easy to compute.

VIII. Conversion to string

Converting an integer to string involves several divisions by the radix. Despite our improvements, our division algorithm still pales compared to a primitive 32-bit integer division. It is therefore useful to have dedicated algorithms.

JavaScript supports conversions using radices 2 to 36, using the method call `x.toString(Δ)`, where `x` is a binary64 number and Δ the radix. We will refer to that function as JSToString(x, Δ). Its specification guarantees a fixed representation without period when x is an integer and $|x| < 10^{21}$. This is useful to us. We can use it to accurately convert an

integer x to a string if $|x| \le 2^{53} < 10^{21}$, using the conversion algorithms of section V, since those conversions are exact. Algorithms 10 and 11 use this idea. They delegate to a third algorithm 12 when x is too large.

At least one of the two first tests is required, since ToStringLarge requires $x \ge 2^{30}$. They could be joined, or the second case could be joined with the third case. Experimentally, using the 3 cases gives the best performance.

Algorithm 10 ToString – unsigned conversion to string

Input: $x = (x_h, x_\ell)$, $\Delta \in \mathbb{N}$, $2 \le \Delta \le 36$
Output: the string representation of x in base Δ.
  if $x < 2^{32}$ **then** {i.e., $x = x_\ell$}
    return $\text{JSToString}(\text{FP}(x_\ell), \Delta)$
  else if $x < 2^{53}$ **then**
    return $\text{JSToString}(\text{RN}(x), \Delta)$
  else
    return $\text{ToStringLarge}(x, \Delta)$
  end if

Algorithm 11 SToString – signed conversion to string

Input: $x = (x_h, x_\ell)$, $\Delta \in \mathbb{N}$, $2 \le \Delta \le 36$
Output: the string representation of $\text{S}_{64}(x)$ in base Δ.
  if $\text{S}_{64}(x) = \text{S}_{32}(x_\ell)$ **then**
    return $\text{JSToString}(\text{FP}(\text{S}_{32}(x_\ell)), \Delta)$
  else if $-2^{53} \le \text{S}_{64}(x) < 2^{53}$ **then**
    return $\text{JSToString}(\text{RN}(\text{S}_{64}(x)), \Delta)$
  else if $\text{S}_{64}(x) < 0$ **then**
    return "-" $+ \text{ToStringLarge}(0 -_{64} x, \Delta)$
  else
    return $\text{ToStringLarge}(x, \Delta)$
  end if

To convert a large unsigned 64-bit integer $x \ge 2^{53}$, we use Algorithm 12. It is a variant of Algorithm 9. The divisor d and $\hat{m}$ come from a table `ToStringTable`. For a radix Δ, `ToStringTable`$[\Delta]$ contains a tuple $(d, w, \hat{m})$ such that:

- $d = \Delta^w$ for some integer w and $2^{30}/\Delta < d \le 2^{30}$, and
- $\hat{m} = \text{RN}(m)$ for $m = 1/d + 2^{-51-k}$ and $k = 24$.

That table can be precomputed ahead of time or computed on first use. It turns out that, for all the d's in that table,

$$\text{RN}\big(1/d + 2^{-75}\big) = \text{RN}\big(\text{RN}(1/\,\text{FP}(d)) + 2^{-75}\big)\,,$$

which provides a straightforward computation for $\hat{m}$.

In Algorithm 12, the quotients (approximated then fixed) are stored in an $\mathbb{F}_{64}$, which avoids a round-trip to 64-bit integers. The remainders are computed and stored as 32-bit integers, since we know that $d \le 2^{30}$.

Theorem 5. *Given $x = (x_h, x_\ell)$, $x \ge 2^{30}$, $2 \le \Delta \le 36$, Algorithm 12 returns the string representation of x in base Δ.*

Proof. Similar to Theorem 4, with the improvement of computing $\hat{r}$ as a 32-bit integer.

Since $\Delta \le 36$, the choice $k = 24$ guarantees

$$1/d + 2^{-51-k} \le 36/2^{30} + 2^{-75} < 2^{-24} = 2^{-k}.$$

Algorithm 12 ToStringLarge – large unsigned conversion to string

Input: $x = (x_h, x_\ell)$, $x \ge 2^{30}$, $\Delta \in \mathbb{N}$, $2 \le \Delta \le 36$
Output: the string representation of x in base Δ.
  $(d, w, \hat{m}) \leftarrow$ `ToStringTable`$[\Delta]$
  $\hat{x} \leftarrow \text{RN}(x)$
  $\hat{q}_1 \leftarrow \text{RN}(\hat{x} \cdot \hat{m})$
  $\hat{q} \leftarrow \lfloor \hat{q}_1 \rfloor$ {as an $\mathbb{F}_{64}$}
  $\hat{q}_\ell \leftarrow \text{truncate}(\hat{q}) \bmod 2^{32}$ {in JS: `q̂l = q̂|0`}
  $\hat{r} \leftarrow x_\ell -_{32} (d \times_{32} \hat{q}_\ell)$
  if $\text{S}_{32}(\hat{r}) < 0$ **then**
    $q \leftarrow \text{RN}(\hat{q} - 1)$
    $r \leftarrow \hat{r} +_{32} d$
  else
    $q \leftarrow \hat{q}$
    $r \leftarrow \hat{r}$
  end if
  $s_q \leftarrow$ `JSToString`(q, Δ)
  $s_r \leftarrow$ `JSToString`$(\text{FP}(r), \Delta)$`.padStart`$(w,$`"0"`$)$
  return $s_q + s_r$

We can use $\hat{q}_\ell$ instead of $\hat{q}$ in the computation of $\hat{r}$ because $\hat{q}_\ell = \text{truncate}(\hat{q}) \bmod 2^{32} = \hat{q} \bmod 2^{32}$. The floating point subtraction $\text{RN}(\hat{q} - 1)$ is exact because $\hat{q} \le 2^{40}$.

As in Theorem 4, we get $q = \text{div}(x, d)$ and $r = \text{rem}(x, d)$. The last two lines convert them to strings. Since $r < d$, the string length of r is bounded by w. The padding forces it to reach exactly w. $q = \lfloor x/d \rfloor \ge 1$, which means we always need the contribution of q to the resulting string. ☐

IX. Evaluation

We implemented all the algorithms mentioned in this paper in the Scala.js compiler [12]. Among other things, the test suite includes fuzzing of integer divisions and remainders.

We benchmarked the implementation of the various division algorithms. The magnitude of the dividends and divisors has a strong impact on most algorithms. Therefore, we measure the run-time cost of signed divisions of a by b, where $|a| < 2^m$ and $|b| < 2^n$, for several values of m and n. In each case, we use 100 random a's and b's, and we measure the 10,000 corresponding divisions. Figure 1 shows the results, labeled as m/n.

The algorithms we compare against are:

- shift-subtract: standard shift-and-subtract algorithm, used by emscripten, for example,
- shift-subtract with shortcut: a shift-and-subtract loop with a shortcut through a binary64 division when the remainder falls under 2^{53}; this was the existing algorithm used in Scala.js, studied in [2], Section 4.4.2,
- J2CL: an algorithm with repeated approximations using binary64 divisions and corrections, and
- bigints: conversion to a JavaScript `bigint` to perform the division and back into a 64-bit integer.

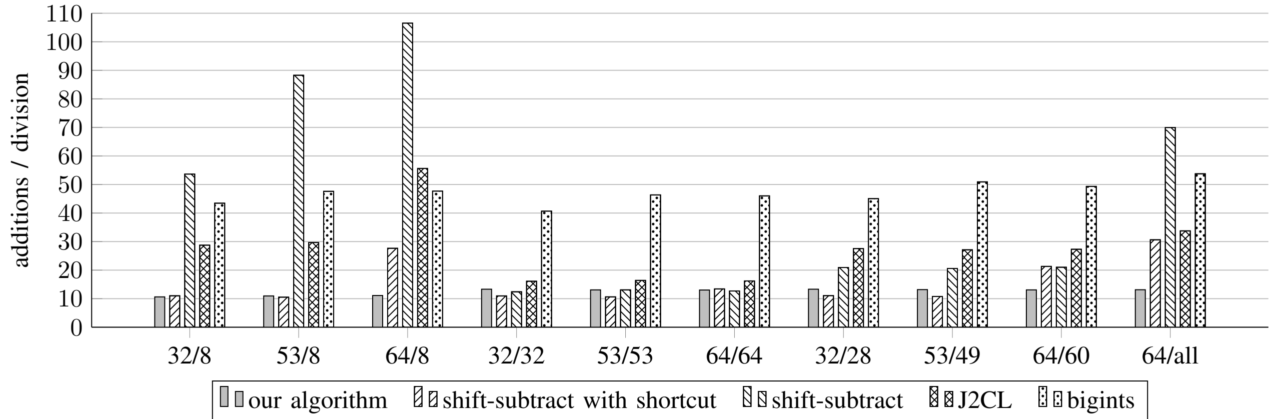

Fig. 1. Reciprocal throughput of divisions, normalized to the cost of one 64-bit addition (lower is better). m/n indicates that the absolute values of dividends (resp. divisors) fit in an m-bit (resp. n-bit) integer. 64/all uses divisors with bit-lengths uniformly distributed from 2 to 64.

In all cases, we use the flat representation with two 32-bit integers, and the underlying modular operations are implemented as described in Section IV.

When both $m \leq 53$ and $21 < n \leq 53$, shift-subtract with shortcut is slightly faster than our algorithm. However, in all other cases, our algorithm outperforms the alternatives. Moreover, it exhibits near-constant performance—it is slightly faster when $n \leq 21$ (i.e., when we use DIV64-CASE1)—whereas the other algorithms are subject to significant variation depending on the magnitude of the operands. This is not surprising, since they all have loops. The cases 64/8, 64/60 and 64/all are particularly striking.

Our algorithm is therefore more predictable, and generally faster, than previously-known algorithms for 64-bit integer divisions.

X. Conclusion

We have proposed a new 64-bit integer division algorithm for the JavaScript platform. It builds on the availability of efficient 32-bit integers and 64-bit floating point numbers, but no access to carry bits or the bits of floating point values. Unlike previously-known algorithms, it contains no loop. It performs at most one binary64 division and either a 32-bit integer division or a 64-bit integer multiplication. We have proven its correctness, including the correctness of conversions between 64-bit integers and binary64 values.

We have also proposed algorithms for constant divisors and for conversions of 64-bit integers to strings, using similar techniques, and proven the algorithms correct. These algorithms perform a single binary64 multiplication.

We implemented all the algorithms mentioned in this paper in the Scala.js compiler [12]. The same algorithms can be implemented by other compilers with 64-bit integers targeting JavaScript, such as emscripten [15], GHCJS [4], J2CL [8], Js_of_ocaml [9], or Kotlin/JS [10]. Our division algorithm is more predictable and generally faster than the alternatives. Currently, they use various loop-based algorithms for 64-bit integer divisions and conversions to strings.

Acknowlegments

The authors would like to thank the anonymous reviewers, whose insights and suggestions led to significantly improving the algorithms and results.

References

[1] N. Brisebarre, J.-M. Muller, and J. Picot, “Error in ulps of the multiplication or division by a correctly-rounded function or constant in binary floating-point arithmetic,” in IEEE Transactions on Emerging Topics in Computing, 2024, 12 (2), pp. 656–666.

[2] S. Doeraene. “Cross-Platform Language Design,” Ph.D. thesis, EPFL, 2018.

[3] ECMA-262, 16th ed., “ECMAScript® 2025 Language Specification,” June 2025.

[4] “GHCJS.” [Online]. https://github.com/ghcjs/ghcjs.

[5] R. L. Graham, D. E. Knuth, O. Patashnik, “Concrete Mathematics: A Foundation for Computer Science,” 2nd ed., Addison-Wesley Professional, February 1994.

[6] T. Granlund and P. L. Montgomery, “Division by invariant integers using multiplication,” in Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, 1994, pp. 61–72.

[7] D. Herman, L. Wagner, and A. Zakai, “asm.js,” August 2014. [Online]. http://asmjs.org/spec/latest/.

[8] “J2CL.” [Online]. https://github.com/google/j2cl.

[9] “Js_of_ocaml.” [Online]. https://github.com/ocsigen/js_of_ocaml.

[10] “Kotlin/JS.” [Online]. https://kotlinlang.org/docs/js-overview.html.

[11] V. Lefèvre, “The Euclidean Division Implemented with a Floating-Point Division and a Floor,” [Research Report] RR-5604, INRIA. 2005. inria-00070403.

[12] “Scala.js.”, [Online]. https://github.com/scala-js/scala-js.

[13] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien, and M. Holman. “Bringing the web up to speed with WebAssembly,” Commun. ACM 61, 12, December 2018, pp. 107–115.

[14] H. S. Warren, “Hacker’s Delight,” 2nd ed., Addison-Wesley Professional, October 2012.

[15] A. Zakai. “Emscripten: an LLVM-to-JavaScript compiler”. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (OOPSLA ’11). Association for Computing Machinery, 2011, pp. 301–312.