EXTENDED NUMERIC REPRESENTATIONS IN WEB ASSEMBLY

by

Scott Butler

A Senior Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the Degree

Bachelor of Computer Science

School of Computing
The University of Utah
December 2019

Approved:

/	/
Matthew Flatt, PhD Thesis Faculty Supervisor	H. James de St. Germain Director of Undergraduate Studies School of Computing
/	
Ross Whittaker, PhD Director, School of Computing	

Abstract

WebAssembly (Wasm) is an emerging compilation target for programming languages, executed by web browsers and sandboxed environments like the Web Assembly System Interface (WASI)[1]. It offers portability, security and near-native execution speed, making it an attractive compilation target.

However, the Minimum Viable Product (MVP) release of Wasm lacks built-in support for many useful numeric representations such as arbitrary precision integers, exact rational numbers, and exact/inexact complex numbers. Source languages like CommonLisp, Haskell, Python and Racket require these representations at run-time to back their built-in numeric types and support their standard libraries.

This lack of representations, and operations that can be used on mixtures of these representations, makes compiling these source languages to Wasm modules unnecessarily difficult. To tackle this issue I have developed a library in C++ for representing these numeric abstractions and cross-compiled it to Wasm, the source code for which is available here:

https://github.com/ScottButler87/ExtendedNumerics

This thesis details background information about this issue and also serves as an exposition of the performance characteristics of the ExtendedNumerics library. The library's implementation is outlined and comparisons are drawn between its performance in native and Wasm compilations. Methods used for verifying correctness and benchmarking performance are also explained.

Despite Wasm's unusual choice of LEB128 as a core number encoding, the library performs well when cross-compiled for browsers/Nodejs. While there still exist other barriers to compiling Haskell, Racket, etc. to Wasm, this library serves as an effective bridge for crossing the existing numeric representation gap.

Contents

1 Introduction **5**

2 Background **7**
- 2.1 Numeric Representations and Dynamic Dispatch 7
- 2.2 Compilers and Assembly Languages 8
  - 2.2.1 WebAssembly: An Emerging Compilation Target 8
- 2.3 WebAssembly Numeric Representations and their Limitations: the Compilation Gap 10
  - 2.3.1 Numeric Representations 10
  - 2.3.2 Numeric Operations 10
- 2.4 Representations and Operations Comprising the Compilation Gap 10
- 2.5 Further Reading 12

3 Related Work **13**
- 3.1 Schism (Scheme to Wasm Compiler) 13
- 3.2 Asterius (Haskell to Wasm Compiler) 13

4 Methods **14**
- 4.1 Approaches Ruled Out 14
  - 4.1.1 Calling into a JavaScript Library 14
  - 4.1.2 LEB128 encoding for Bignums 14
  - 4.1.3 Initial Implementation 14
- 4.2 Library Implementation 15
  - 4.2.1 Implementation Details 15
  - 4.2.2 Cross-compilation to WebAssembly 16
  - 4.2.3 Notable Aside: GNU Multiple Precision Arithmetic Library 16
- 4.3 ExtendedNumerics Python Extension 17
- 4.4 Correctness/Testing 17
- 4.5 Benchmarking 17
  - 4.5.1 Failed Initial Course: Google benchmark and gtest with WebAssembly 17
  - 4.5.2 Performance Benchmarking 18
  - 4.5.3 Sanity Test Benchmarks 19
  - 4.5.4 Note: Benchmark Execution Environment 19

5 Results **20**
- 5.1 Summary 20
- 5.2 Operation Asymptotic Behavior 20
  - 5.2.1 WebAssembly Performance 20
  - 5.2.2 A Target of Opportunity: Benchmarking the Python Extension vs Python Builtins 21

6 Conclusions **23**

Appendices **25**

A WebAssembly Numeric Details **25**
A.1 Representations 25
A.2 Integer Operations 26
A.3 Floating-Point Operations 28

B Operation Doubling Behavior Comparison, Native vs Webassembly **29**
B.1 Bignums 29
B.2 Ratnums 30
B.3 Exact Complexnums 31

C Performance Per Operand Comparisons, Native Vs WebAssembly **33**
C.1 Fixnums 33
C.2 Inexact Complexnums 34
C.3 Bignums 35
C.4 Ratnums 36
C.5 Exact Complexnums 37

D Python Extension Performance Comparison Charts **38**
D.1 Net Benefit, Use Case Potential 38
D.2 Neutral 41
D.3 Net Loss 42

E Benchmarking Environment – CADE Lab Specifications **46**

List of Tables

1 Required Numeric Representations by Language 12
2 WebAssembly Numeric Representations 26
3 WebAssembly Integer Arithmetic 26
4 WebAssembly Integer Bit Operations 27
5 WebAssembly Integer Comparison Operations 27
6 WebAssembly Floating-Point Arithmetic Operations 28
7 WebAssembly Floating-Point Utility Operations 28
8 WebAssembly Floating-Point Comparison Operations 29

1 Introduction

The source code backing everyday programs and applications often involves calculations that employ approximate computer representations of a wide spectrum of different types of mathematical numbers, both discrete and continuous: integers, reals, complex numbers, and all manner of combinations thereof.

Precision requirements for these numeric representations can vary from one program to the next. An application that is calculating frames to be displayed one after another to a user playing a video game might be satisfied with representations that are rough approximations, resulting in some user-imperceptible level of imprecision. On the other hand, a scientist trying to predict the outcome of a chemical reaction by modeling atomic interactions might require computations with exact representations which provide sufficient precision for the calculation.

Among its other responsibilities, a compiler for a source language must choose appropriate representations for the mathematical numbers specified in, or resulting from the execution of, the source code of a program. These representations must be capable of modeling the operations the programmer expects, with the required precision.

To fully understand the crux of the issue being brought to discussion, it is necessary to distinguish between mathematical numbers and the numeric types and associated representations provided by programming languages to model these numbers. While mathematical types and their expected behavior during computations may be easily described, implementing this behavior at the level of computer hardware requires binary-compatible representations whose form can be radically different from the mathematical numbers being modeled.

Yet programmers expect to be able to rely on these representations to accurately emulate the behaviors expected from mathematical numbers. They rely on the ready availability of the necessary representations and associated operations to implement the numeric calculations required by their programs at run-time.

Although `Wasm` natively supports all of the numeric representations employed by `C`-like languages, it lacks any native support whatsoever for the numerical representations needed by compilers for `CommonLisp`, `Haskell`, `Python`, `Racket`, `Scheme` and other languages that require representations outside this limited subset. Representations for rationals, arbitrary precision integers and complex numbers are sorely lacking, and this leaves almost half the numeric stack of

these languages lacking a suitable representation in `Wasm`.

Thus although `Wasm`'s minimum viable product release came prepared to be a compilation target for `C`-like languages, it requires an extension or a library module to support compiled programs from languages with native support for a deeper numeric stack. In addition to providing appropriate representations for each type in the stack, such a library would also necessarily need to support operations on and *between* these types.

In order to efficiently meet the needs of a compiler, this library would need to provide the same abstraction to the compiler as it does to users: once a numeric value is encoded as some concrete representation, it should be employable as an operand for any appropriate numeric operation; treated as a generic number. Operations performed on these numbers must be dispatched to the correct operation for the given operands depending on their underlying representations.

Developing and compiling a library to `Wasm` to support these numeric types at run-time and testing the library's viability as an extension to `Wasm` is the primary goal of my research. The ExtendedNumerics library, written in C++, is the culmination of my work. Details of its implementation are laid out in the methods section of the paper. While using the numeric types provided by this library necessarily result in slower performance than possible through direct hardware or virtual machine support, it is a first step toward making `Wasm` a fully compatible compilation target for high-level languages that supply a deeper numeric stack to programmers than `C`-like languages do.

2 Background

High-level programming languages (such as `C/C++`, `Java`, `Python`, `Racket` and others) offer programmers convenient access to a wide variety of numeric representations that abstract away the specific complexities of representing a mathematical number in computer hardware and performing calculations with it. These implementation details are resolved by code imported as a library or by an interpreter or compiler for the language.

2.1 Numeric Representations and Dynamic Dispatch

One core difference between `C`-like languages and languages like `Racket`, `Scheme`, `Haskell` (and to some degree `Python`[1]) is that in `C`-like languages, the user must explicitly specify or provide the representation for a numeric value at the time the source code is written. If built into the language, the representation chosen is directly related to the implementation used by the compiler for that number under the hood.

In contrast, programmers writing in languages like `CommonLisp`, `Haskell`, `Scheme`, `Racket` and others must specify only a numeric literal and operations to be performed on it. The compiler chooses a representation when the program is translated into a target assembly language, inferring the appropriate representation from the shape of the source numeric literal or from the representations used for other operands in the same numeric operation. The representation employed by the compiler can be one directly supported by the target instruction set such as 64-bit integers or `IEEE-754` double precision floats, or a more complicated representation like an arbitrary-precision integer or rational number.

The delay in requiring specific representation choice until compile time helps make using these languages easier; programmers do not have to specify that some multiplication is between a floating-point number and an integer. A programmer simply defines two operands, without knowing or caring how the operands will be represented in hardware, and the appropriate representations and operations should be performed at run-time.

However, this convenience comes with a cost. Compiling languages that treat numerics and arithmetic in this manner requires support at the assembly

[1] While `Python` requires some degree of representation selection for a numeric value, it supports integers of arbitrary magnitude and values such as complex numbers. These qualities make it a challenging language for compilation to `Wasm`, similar to the other languages mentioned.

language level for both:

1. representing the full numeric tower of the source language, and
2. dynamically dispatching arithmetic operations based on the run-time representation of their operands.

For assembly targets that do not natively support these requirements, it is necessary to provide a supplemental library written in the target assembly language to fill in the gaps.

2.2 Compilers and Assembly Languages

Compilers are the category of programs generally tasked with translating programs written in a high-level programming language into assembly instructions for a target instruction set architecture. This compilation prepares the program to be assembled into machine code appropriate for the executing platform and loaded into memory for execution at run-time. Examples of assembly languages designed for hardware implementations include `MIPS`, `x86_64`, `RISC-V`.

Each of these assembly languages is the programming interface for an instruction set implemented by a physical device. Any given high-level language may be supported by a whole family of compilers (or compiler back-ends), each designed to convert source code written in that language into a different target assembly language.

As early as the IBM System/360 with the advent of CP/CMS, the concept of emulating the hardware that programs are executed on was conceived[2, 14]. This ushered in the era of virtual machines, and not long after came the idea of a virtual run-time: a virtual machine that acts as a translator between an intermediate language generated during initial compilation and the machine code for the physical machine it executes on.

In the modern era, this idea has matured into notable intermediate language and virtual machine pairs like Java Byte-code/Java Virtual Machine for the `Java` language and Common Intermediate Language/Common Language Runtime for `C#` and other .NET languages.

2.2.1 WebAssembly: An Emerging Compilation Target

A more recent addition to the growing list of intermediate languages and their companion virtual machines is `WebAssembly`. First announced in 2015, the

minimum viable product was finalized by the World Wide Web Consortium (W3C) in March, 2017.

Since its release, Wasm (Wasm) has become a new and interesting compilation target for compilers translating high-level languages. Its binary format is compact and highly portable, and was designed to be easily translated into an S-expression-based text format (file extension .wat) for readability. Once a module is downloaded over the internet, it can be executed directly by supporting browsers including Apple Safari, Google Chrome, Mozilla Firefox and Microsoft Edge. As an assembly language written for a well-defined conceptual machine, the set of platforms Wasm can execute on is extensible. Recently plans have been laid to develop a system interface[1] which would provide a Wasm platform for modules to be executed directly without the need of a browser.

Wasm has many features that make it an attractive compilation target. It is run in a sandboxed environment on the executing machine, protecting the operating system and hardware from malicious programs. It was designed with security, speed and portability in mind; downloaded Wasm code is validated before execution, and the linear memory used by executing modules for temporary storage is separated from the call stack and other vital data structures so that these structures are not vulnerable to mistaken or malicious access.

Because it is an assembly language that is converted to native code by an assembler before execution, it is capable of matching speed with any equivalent desktop application. Innovations by Google in their Chrome V8 engine show off some of the potential still waiting to be explored. The V8 engine translates downloaded Wasm modules into native code on the fly, function by function as they are received, using an optimizing compilation pipeline they call "TurboFan."

In August of 2018, an additional straight-line compilation pipeline called "Liftoff" was added to the V8 engine. The focus of this secondary pipeline is to allow the chrome browser and V8 engine to begin execution of the Wasm modules as they are received. In combination, these two pipelines allow the V8 engine to begin execution of Wasm programs as they are downloading and simultaneously perform optimizations on hot code paths (control flows that are often executed) in the background.

2.3 WebAssembly Numeric Representations and their Limitations: the Compilation Gap

While `Wasm` provides a panoply of operations for the numeric representations it supports, this set of representations is intentionally limited. The provided representations and operations are insufficient to support the full numeric stack for some of the high-level languages previously mentioned (`Haskell` etc.) on their own.

For languages that require these more complex representations, this represents a barrier to `Wasm` serving as a target for compilation. The lack of any `Wasm` library written to resolve this issue is what I call the compilation gap; a library would serve as the bridge to compilation.

The limited representations provided by `Wasm` form a sufficient platform for implementing such a library to provide the representations and dynamically dispatching numeric operations required. The source for this library could be written in `Wasm` text format or could be compiled to `Wasm` from a similar library written in a `C`-like high-level language.

2.3.1 Numeric Representations

`Wasm` provides 32 and 64 bit integer representations, which are LEB128[3] encoded two's complement[4] binary representations. They have no explicit signedness and this simplifies many of the operations performed on them. Further details can be found in Appendix A.

2.3.2 Numeric Operations

`Wasm` supports the full gamut of numeric operations for the representations it provides. These include conversions between representations, comparisons, arithmetic, common functions such as min, max, absolute value, negation and several types of rounding. Further details can be found in Appendix A.

2.4 Representations and Operations Comprising the Compilation Gap

To bridge the numeric compilation gap `CommonLisp`, `Haskell`, `Python`, `Racket` or `Scheme`, a library would have to be compiled to `Wasm` that:

1. provides the numeric representations required by these languages that `Wasm` is lacking, and

2. supports generic numeric operations for operating on the provided numeric representations as well as WebAssembly's built-in representations.

By generic numeric operation, it is meant that for any numeric operation provided by the library (for example, addition), it must be possible to apply that operation to any combination of numeric operands, even if their underlying representations do not match. At a minimum, the following subset of numeric operations would need to be implemented:

1. Negation
2. Addition
3. Subtraction
4. Multiplication
5. Division
6. Quotient
7. Remainder
8. Comparisons
    (a) Equality
    (b) Inequality
    (c) Less Than

Any other operation required could be constructed through the use of these operations. The table below illustrates the representations required by each specific language which are not provided by WebAssembly. Any library aiming to bridge the compilation gap must provide these necessary representations in addition to the minimum subset of numeric operations above.

Table 1: Required Numeric Representations by Language

Numeric Representation	CommonLisp	Haskell	Python	Racket	Scheme
Arbitrary Precision Integers	X	X	X	X	X
Arb. Prec. Rationals	X	X	X	X	X
Fixed Prec. Complex Integers	-	X	X	X	X
Arb. Prec. Complex Integers	X	X	-	X	X
Arb. Prec. Complex Rationals	X	X	-	X	X
Complex IEEE 754-2008 FPs	X	X	X	X	X

2.5 Further Reading

For the interested reader, some further background information can be found in these reports, by topic: Wasm[8], cross-compilation to Wasm[17][10], Wasm vs native code performance[13], and assessing the performance delta between Wasm and JavaScript for Numerical programs[9].

3 Related Work

While `Scheme`, `Lisp` and `Haskell`, and other language families' compilers have been providing the necessary numeric representations for decades in compilation to assembly for `x86/x86_64` instruction sets, at the time of writing I could find no ongoing projects that attempt to bring this capability to `Wasm`.

There are several active open source projects working towards compiling subsets of `Scheme`, `Haskell` and other languages that provide extended representations, though none of them have as of yet tackled the fundamental issue addressed in this thesis.

The following are known ongoing compiler projects that provide only the numeric representations natively supplied by `Wasm`. These projects would benefit from the creation of a library that bridges the aforementioned numeric compilation gap for their languages. I am sure these are not the only ongoing projects, and more likely crop up every day.

3.1 Schism (Scheme to Wasm Compiler)

"Schism is an experimental self-hosting compiler from a subset of R6RS `Scheme` to WebAssembly"[7] according to its project description. The project successfully self-hosts using snapshots checked of previous iterations checked into the repository. This compiler explicitly states that its support for the full numeric stack of Scheme is restricted to integers within the int32 range $(-2^1,\ 2^1 - 1)$.

3.2 Asterius (Haskell to Wasm Compiler)

This project's description reads: "A `Haskell` to WebAssembly compiler. Project status: alpha, in active development, some simple examples already work".[12] The project supports Javascript/Haskell interoperation via defined first-class Haskell types and additionally provides infinite precision integers through external calls to a JavaScript framework, taking advantage of the JavaScript built-in BigInt type added in late 2018. One drawback to this approach is the large amount of overhead when making calls to JavaScript from Wasm and vice versa.

4 Methods

This section outlines considerations made and methods used for implementing, porting, testing and benchmarking the proposed library, referred to henceforth by the name ExtendedNumerics.

4.1 Approaches Ruled Out

4.1.1 Calling into a JavaScript Library

Calling into a JavaScript library for specific numeric representations such as BigInts ultimately turned out to be unlikely to be a good approach. There is a lot of overhead associated with calls to and from Wasm/JavaScript. That overhead would make the implementation less performant than one where all the module's operations are self-contained.

4.1.2 LEB128 encoding for Bignums

Internally, integers in Wasm are encoded in LEB128, however this is likely due to space concerns or data streaming. It is possible that a library implementation that encoded its numeric representations in LEB128 could be more compatible with Wasm, resulting in better performance. However this is unlikely to be the case as it would be difficult to communicate the fact that representations are compatible to the intermediate compiler from C to Wasm.

4.1.3 Initial Implementation

I wrote the first iteration of the ExtendedNumerics library in C with no external dependencies. My hypothesis was that the reduction in overhead versus C++, combined with the similarity between C code and Wasm instructions, would result in better performance after cross-compilation than a C++ implementation.

However, before benchmarking began in earnest for this initial C implementation, I made a comparison with a C++ prototype for a small subset of the required functionality. This revealed that my hypothesis was wrong. After comparing operation performance, it was clear that fixnum operations were comparable in C and C++; furthermore the cpp_int arbitrary precision integer representation provided by the C++ Boost Multiprecision library easily outperformed my own implementation asymptotically.

In light of this revelation, and spurred on by my own desire to refactor the existing implementation, I made the decision to reimplement the library in C++14.

4.2 Library Implementation

4.2.1 Implementation Details

ExtendedNumerics is written in C++14 and takes a dependency on the Boost Multiprecision library header. It is built using CMake, a platform independent build system provided by Kitware, Inc.[11] The main class, Numeric, encompasses and abstracts away the complexity of the underlying internal classes. Behind the scenes one of several classes stores the necessary information and implements the functionality that enables the extended numeric stack. Each of these classes inherits from ExtendedNumerics, a base class that defines the shape and interface that must be implemented by backing types in order to further extend the numeric stack. At initialization time, an appropriate numeric back end is selected from one of these derived classes: BignumInternal, RatnumInternal, ExactComplexnumInternal, InexactComplexnumInternal.

Numerics can be instantiated with standard integer values, c-style strings, std::strings or doubles depending on the type literal to be represented. The Numeric class employs the Resource Acquisition is Initialization (RAII) technique to allow users to handle Numerics just as they would regular built-in C++ value types. Any dynamic memory needs are serviced under the hood by the backend classes and Numerics clean up after themselves when they go out of lexical scope, ensuring they do not cause memory leaks. Although the number they are representing can be of arbitrary precision and size, the Numeric itself always takes up only 64 bits on the stack.

Taking inspiration from the Chez Scheme[5] implementation, Fixnum values are stored directly in the Numeric on the stack in order to reduce the performance penalty of using Numerics for small values. Bignums, Ratnums and Exact/Inexact Complexnums are referenced via a pointer stored in the Numeric. Following the guidance of the principle of least surprise, like other value types Numerics are immutable; performing operations on them produces new Numeric values rather than mutating the originals.

In order to further support the outward appearance that Numerics are a value type, they had to be made to behave as other built-in value types, including interoperability with other built-ins and each other. Given a binary

operation on two Numerics or on a Numeric and a built-in, the correct action to take depends on the types of each. For instance, division should be handled very differently for integer versus complex values. Dispatching to the correct function presents a challenge in C++14, which does not support multimethods or multiple dispatch for user-defined classes.

To overcome this, operations are initially dispatched to the ExtendedNumerics base class based on the base type of the right hand operand. From there, derived classes are dispatched twice more based on explicit type tags carried at run-time, once for each operand. Built-in values are treated in the same fashion, first dispatching to ExtendedNumerics and then directly on to a friend function of the appropriate back end. This approach is necessitated by the lack of virtual friend functions in C++14.

4.2.2 Cross-compilation to WebAssembly

The library is compiled to Wasm via the Emscripten SDK, maintained, created and provided by the authors listed at emscripten.org[16]. This tool is as close to a drop-in replacement for gcc as is available. I used the emconfigure utility in combination with CMake and unix make to compile the Numerics library directly to Wasm from source. The boost dependency is serviced in Wasm by a port of the boost headers provided by Emscripten: github.com/emscripten-ports. This cross-compiled version of the ExtendedNumerics library is compatible with Nodejs as well as any modern browser that supports Wasm.

4.2.3 Notable Aside: GNU Multiple Precision Arithmetic Library

GNU Multiple Precision Arithmetic Library (GMP) is a reliable and widely used C/C++ multiprecision library maintained by the Free Software Foundation at gmplib.org. During implementation I experimented with using GMP's arbitrary precision integer back end instead of Boost's cpp_ints. However, I found that native speeds were relatively similar for numbers up to 10,000 bits long. Additionally, after struggling with Emscripten SDK to compile GMP to WebAssembly, I found that it was significantly slower than Boost when cross-compiled. For that reason, ExtendedNumerics employs Boost's headers instead. It is worth noting that it might be worth trying the conversion again in future, in case some silent build error occurring during cross-compilation was the source of the unexpected poor performance.

4.3 ExtendedNumerics Python Extension

In the interest of demonstrating a second use-case for this codebase in addition to numeric representations for Wasm, ExtendedNumerics was also compiled and exercised as a Python extension. I accomplished this using the Python distutils package in tandem with the Simplified Wrapper and Interface Generator (SWIG) open source software tool available from swig.org. An additional benefit of having this extension is that this enables integration testing of the library in situ. It also makes using the library and adding new tests tractable for a larger population of developers than would C++/Wasm alone.

4.4 Correctness/Testing

The test suite for ExtendedNumerics is written in Python and exercises the library compiled as a Python extension. In addition to a set of hand-made edge cases and regression tests, the suite includes a probabilistic test module that tests all operations and operand pairs. The suite generates random Numerics and their equivalents as Python values and tests the results of performing Numeric operations against the results generated by Python arithmetic operations. This leverages the reliability of the Python built-in arithmetic types to fuzz the ExtendedNumerics library and verify its results, searching for aberrant behavior. The number of iterations and floating point relative equality tolerance is adjustable and the fuzzing suite is readily extensible for new Numeric representations as necessary. The current ExtendedNumerics library implementation has withstood over a hundred million random samples without exception.

4.5 Benchmarking

4.5.1 Failed Initial Course: Google benchmark and gtest with WebAssembly

Tempted by the ease of use and reliability of the Google open source projects gtest and Google benchmark, initially I tried to take a dependency on these C++ libraries. Google benchmark's built-in timing, complexity measurements and anti-optimization facilities were of particular interest. What began as a modest 5-8 hour investment in learning to use this new library and writing benchmarks culminated in over 30 hours of build failures, intense frustration and ultimate failure, as I tried to coerce Emscripten, CMake, ExtendedNumerics,

gtest and Google benchmark to work together. I will briefly summarize this struggle below.

At the time and unbeknownst to me, the experimental Emscripten upstream llvm branch had a bug where vtables were not being properly populated and this caused the benchmark build to fail at link time. Switching to the more reliable Emscripten fastcomp backend seemed to relieve this issue.

The use of advanced compilation flags/options with em++ are not well documented and also not covered in great detail on Q/A sites like StackOverflow. This difficulty was aggravated by my lack of understanding of the CMake build system. At build time Google benchmark requires features such as pooled threading, expandable memory, link time optimization and more. It turned out that the reason these issues are not well-documented is that the Emscripten SDK provides the utilities emconfigure and emmake which automatically detect and resolve these issues.

After seemingly successful cross-compilation with the Emscripten fastcomp backend, the Google benchmark library's internal tests were failing. Specifically, when running the .js drivers produced by Emscripten, Nodejs complained that WebAssembly.Memory returned a memory buffer that waas not a SharedArrayBuffer. Node suggested that this was due to a lack of Wasm thread support.

Hours of background research later, I found that support for threads in Wasm has been disabled until further notice due to vulnerabilities to the Spectre attack vector first revealed in January, 2018. Details about the vulnerability can be found at https://meltdownattack.com.

While fixes for this issue involving authentication and authorization of browsing environment resource requests are in development, the timeline for resolution is still unclear. The best resource I could find for more information about these issues was at Mozzilla, here: Cross-Origin Resource Policy (CORP). Having finally met an insurmountable barrier, I resorted to writing the necessary benchmarks in C++14 myself.

4.5.2 Performance Benchmarking

To demonstrate the performance characteristics of the ExtendedNumerics library as both a native and Wasm compilation target, I created two primary benchmarks. The first of these measures the doubling behavior of all ExtendedNumerics operations, checking for differences in asymptotic behavior from one platform to the other. The other benchmarks the performance of each

Numeric subtype in all supported operations with every other Numeric subtype. This benchmark forms the baseline for objective ExtendedNumerics performance comparisons between the native and Wasm compilation targets.

4.5.3 Sanity Test Benchmarks

I wrote several other benchmarks for performing basic sanity checks on the library. As the results were more an aside, I am documenting them here rather than in the results section. Benchmarking operation performance symmetry (measuring the delta between NumericType1 BINOP NumericType2 vs NumericType2 BINOP NumericType1) unearthed several performance issues that were later fixed. It also gave me insights into some of the performance characteristics of the Boost multiprecision types I have employed as internal arbitrary precision representations.

A benchmark for measuring the overhead of using Numeric ratnums rather than the cpp_rational backend directly is also included. It reveals that the overhead associated with the extra level of indirection is about 20 nanoseconds per 1200 nanosecond operation. I also employed it to measure construction time overhead for Numerics, which clocks in at around 35 nanoseconds additional overhead.

I also wrote a benchmark to measure the difference in execution time between built-in 64 bit C integers and Numeric fixnums. The difference is not insignificant, with Numeric fixnum multiplication taking approximately 12 times as long as the built-in multiplications. This includes time spent by the Numeric determining its own internal type, as well as ensuring that no operations overflow the precision capacity of fixnums. Notably, operations that do not overflow are 3 times faster than equivalent operations that do require overflow from fixnum to bignum. All of these operations are orders of magnitude faster than calculations that must result to ratnum representation to prevent precision loss.

4.5.4 Note: Benchmark Execution Environment

To maximize consistency and reproducibility, all benchmarking results reported in the following section were collected by running the benchmarking suite on the University of Utah CADE lab desktop machines. Detailed specifications are included in appendix E.

5 Results

5.1 Summary

Benchmarking the ExtendedNumerics library compiled to both native x86_64 and Wasm revealed that while WebAssembly is undeniably faster than Javascript and its predecessor asm.js, there is still more ground to be covered. Regardless, the results were relatively positive.

I found that the natively compiled library executed benchmarks at around twice the speed observed from Wasm. This performance delta is consistent across all benchmarks, though the deficit may be exacerbated if test inputs grow large enough to merit taking advantage of multiprocessing in the ExtendedNumerics implementation. WebAssembly's multiprocessing support is still experimental.

Numeric operation asymptotic complexity remained stable, unchanged between native and Wasm benchmarking. Most assuring, the performance characteristics of Numeric operations are clearly preserved despite cross-compilation and a fundamental change in underlying binary representation. Figure 1 is a chart demonstrating the trends per operand for each operation in both native and Wasm. Charts with more fine-grained details can be found in appendices B, C and D. What follows is an overview of the most pertinent information acquired during benchmarking.

5.2 Operation Asymptotic Behavior

As mentioned above, no discrepancy was observed in the ExtendedNumerics library's asymptotic performance between native and Wasm. Perhaps unsurprisingly, multiplication and division were found to be not-quite-quadratic with respect to operand bit size, regardless of whether the operations were being performed on bignum, ratnum or exact complexnum back ends. Addition and subtraction both scaled linearly with operand bit size. Note: Results were not gathered for fixnums or for inexact complexnums, as these back ends are both fixed precision.

5.2.1 WebAssembly Performance

Although there was an across-the-board 50% reduction in performance, no performance degradations were observed for any of the permutations of operations

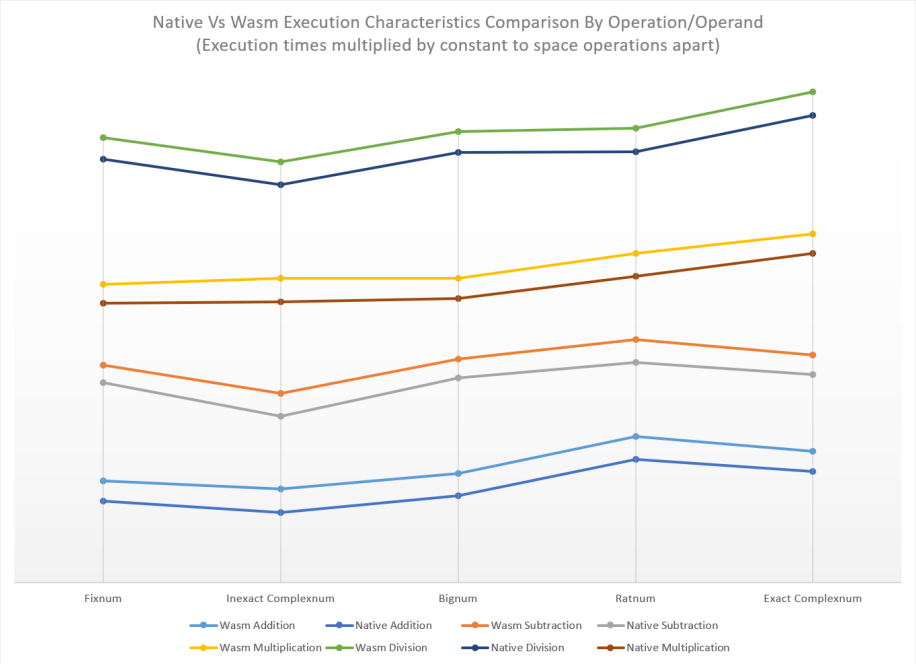


Figure 1: Performance characteristics of native vs Wasm operations across operands.

between different Numeric back ends. The breakdowns by operand and operation in appendix C are near mirror-images from native to Wasm compilation targets. Most importantly, fixnum and inexact complexnum operations are still significantly faster than their arbitrary-precision siblings, even in Wasm. This performance characteristic is an important factor in deciding whether or not Wasm is a desirable target for programs that perform large numbers of small calculations.

5.2.2 A Target of Opportunity: Benchmarking the Python Extension vs Python Builtins

The Python extension arose of necessity during development as a handy form of correctness verification for Numeric operations. This presented an opportunity to exercise the ExtendedNumerics library as an extension of Python functionality and measure its performance against Python builtin types and a few user-defined classes that emulate the functionality of the ExtendedNumerics

library.

Appendix D showcases the results of that benchmarking in detail. The results are broken down by subcategories – net benefit, neutral and net loss. Numerics employing each of the back ends were timed against Python built-in values (in the case of bignums and Python ints), library classes (ratnums and Fraction) and a few classes I defined myself for emulating the required equivalent ExtendedNumeric library functionality that were previously exclusively for correctness checking.

Python handily beats the Numerics extension in operations that can be accomplished using its own built in value types, presumably because these calculations are performed directly inside natively-compiled libraries rather than by interpreting Python byte code.

However, the Numerics library outperforms Python when it comes to operations between Python built-ins and user-defined/imported library classes. Perhaps the imported libraries are not as heavily optimized as the core numeric types. Numerics performed particularly well on operations between built-in inexact complex numbers and user-defined exact complex number classes, as well as on operations where one operand was small enough to fit within a fixnum. These benchmarking results highlight some enticing immediate use-cases for the ExtendedNumerics Python extension.

6 Conclusions

The development, testing and benchmarking of the ExtendedNumerics library represented a significant step towards proving viability of the WebAssembly platform as a compilation target for many functional languages, and providing a necessary stepping stone for compiling languages that require support for an extended numeric stack to Wasm.

My findings indicate that while Wasm is still somewhat lacking in performance for arithmetic-heavy applications in comparison to natively compiled code, it is more than ready to support general purpose computing that relies on the functionality ExtendedNumerics provides.

The Numeric class provides the necessary numeric representations, operations and interoperations necessary to readily support languages like Common-Lisp, Racket and the others mentioned at length in the background section. I am excited to be a continuing part of the effort to make general purpose computing on the Web a reality, and I look forward to seeing what more is accomplished using WebAssembly in the near future.

References

[1] Lin Clark. Standardizing WASI: A system interface to run WebAssembly outside the web – Mozilla Hacks - the Web developer blog, Mar 2019. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

[2] Wikipedia Contributors. Wikipedia - CP/CMS, Feb 2019. https://en.wikipedia.org/wiki/CP/CMS/.

[3] Wikipedia Contributors. Wikipedia - LEB128, Apr 2019. https://en.wikipedia.org/wiki/LEB128/.

[4] Wikipedia Contributors. Wikipedia - Two's Complement, Apr 2019. https://en.wikipedia.org/wiki/Two%27s_complement/.

[5] R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, New Mexico University, Chapel Hill, NC, USA, 1987. http://agl.cs.unm.edu/~williams/cs491/three-imp.pdf/ UMI Order No. GAX87-22287.

[6] Dan Gohman. sunfishcode/wasm-reference-manual, Jan 2019. https://github.com/sunfishcode/wasm-reference-manual/.

[7] Google. google/schism, Jan 2019. https://github.com/google/schism/.

[8] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Notices*, volume 52, number 6, pages 185–200. ACM, 2017.

[9] David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. WebAssembly and JavaScript Challenge: Numerical Program Performance Using Modern Browser Technologies and Devices. Technical report, McGill University School of Computer Science Sable Research Group, 2018.

[10] Eric Holk. Schism: A Self-Hosting Scheme to WebAssembly Compiler. In *Proceedings of the Scheme and Functional Programming Workshop 2018*. Scheme and Functional Programming Workshop, 2018.

[11] Kitware Inc. CMake, Dec 2019. https://cmake.org/.

[12] Tweag I/O. tweag/asterius, Jan 2019.
https://github.com/tweag/asterius/.

[13] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 107–120, 2019.

[14] Stuart E Madnick. Time-Sharing Systems: Virtual Machine Concept Vs. Conventional Approach. *Modern Data Systems*, 2(3):34–36, Mar 1969.
http://web.mit.edu/smadnick/www/papers/J004.pdf.

[15] IEEE Computer Society. *754-2008 IEEE Standard for Floating-Point Arithmetic*. IEEE Standards Association, 2008.
https://ieeexplore.ieee.org/servlet/opac?punumber=4610933/.

[16] Emscripten Contributors: https://emscripten.org/docs/contributing/AUTHORS.html. Emscripten SDK, Dec 2019.
https://github.com/emscripten-core/.

[17] Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

Appendices

A WebAssembly Numeric Details

A.1 Representations

Since integers use a 2's complement representation, most integer operations are not differentiated into signed and unsigned versions. For those operations where signedness does make a difference in the bit-level processes performed, signed and unsigned versions are provided.

Table 2: WebAssembly Numeric Representations

Name	Representation
i32	LEB128 encoded binary integer
i64	LEB128 encoded binary integer
f32	32 bit IEEE 754-2008 single-precision floating point
f64	64 bit IEEE 754-2008 double-precision floating point

A.2 Integer Operations

Table 3: WebAssembly Integer Arithmetic

Abbreviation	Description
`add`	Addition, carry bit discarded
`sub`	Subtraction, borrow bit discarded
`mul`	Multiplication, low bits of result only
`div_s`	Signed integer quotient, result rounded toward 0
`rem_s`	Remainder after signed division
`div_u`	Unsigned integer quotient, result rounded toward 0
`rem_u`	Remainder after unsigned division

Note: Both div and rem trap on division by zero. div_s traps on signed integer minimum divided by -1.

Table 4: WebAssembly Integer Bit Operations

Abbreviation	Arity	Description
`clz`	unary	Count leading zeroes
`ctz`	–	Count trailing zeroes
`popcnt`	–	Population count, number of bits set to one
`and`	binary	**L** bitwise AND **R**
`or`	–	**L** bitwise OR **R**
`xor`	–	**L** bitwise XOR **R** (exclusive or)
`shl`	–	Shift bits of **L** left by **R**'s value, filling with zeroes
`shr_s`	–	Shift bits of **L** right by **R**'s value, sign extending on the left
`shr_u`	–	Shift bits of **L** right by **R**'s value, filling with zeroes
`rotl`	–	Rotate bits of **L**, most significant to least significant, **R** times
`rotr`	–	Rotate bits of **L**, least significant to most significant, **R** times

Note: Bitwise negation can be performed using xor with -1 as the first operand.

Table 5: WebAssembly Integer Comparison Operations

Abbreviation	Signed	Operation		
`eqz`	N/A	**L**	==	0
`eq`	N/A	**L**	==	**R**
`ne`	N/A	**L**	!=	**R**
`lt_s`	Yes	**L**	$<$	**R**
`le_s`	Yes	**L**	$\leq$	**R**
`gt_s`	Yes	**L**	$>$	**R**
`ge_s`	Yes	**L**	$\geq$	**R**
`lt_u`	No	**L**	$<$	**R**
`le_u`	No	**L**	$\leq$	**R**
`gt_u`	No	**L**	$>$	**R**
`ge_u`	No	**L**	$\geq$	**R**

Note: These comparisons return a Boolean result, which is represented in WebAssembly as an i32. Any non-zero value is considered true, while 0 is interpreted as false.

A.3 Floating-Point Operations

For the most part, WebAssembly floating-point representations behave as specified by IEEE standard 754-2008[15] which describes a specification for hardware implementations of floating point. However, there are many caveats and intricacies involved in the behavior of the utility operations described below. For more information, consult the WebAssembly reference manual[6].

Table 6: WebAssembly Floating-Point Arithmetic Operations

Abbreviation	Description
`add`	Addition
`sub`	Subtraction
`mul`	Multiplication
`div`	Division interpreting each operand as unsigned

Note: Each operation conforms to the IEEE 754-2008 specification of its behavior.

Table 7: WebAssembly Floating-Point Utility Operations

Abbreviation	Arity	Description
`abs`	unary	IEEE 754-2008 compliant absolute value
`neg`	–	IEEE 754-2008 compliant negate value
`sqrt`	–	IEEE 754-2008 compliant square root
`ceil`	–	IEEE 754-2008 compliant round to integral, towards positive
`floor`	–	IEEE 754-2008 compliant round to integral, towards negative
`trunc`	–	IEEE 754-2008 compliant round to integral, towards zero
`nearest`	–	IEEE 754-2008 compliant round to integral, ties to even
`min`	binary	Minimum value considering **L** and **R**, -0 < 0
`max`	–	Maximum value considering **L** and **R**, -0 < 0
`copysign`	–	Result has magnitude of **L** and sign of **R**

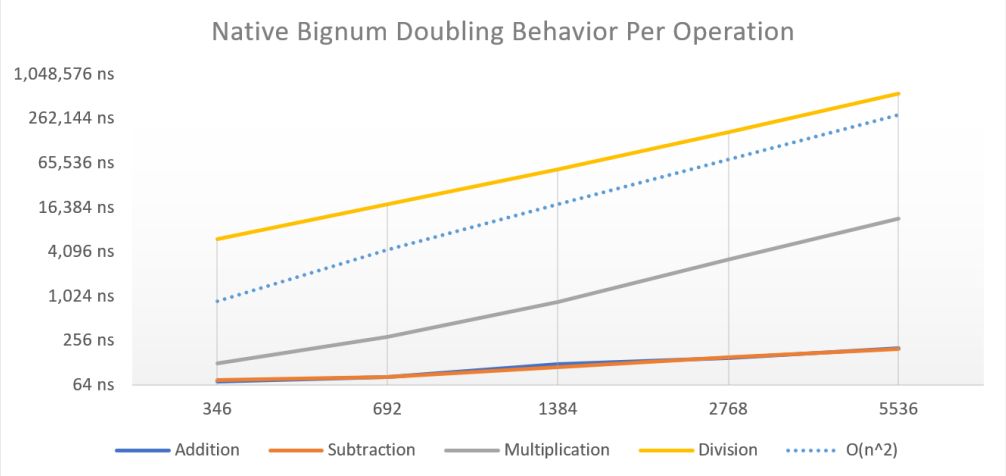
Table 8: WebAssembly Floating-Point Comparison Operations

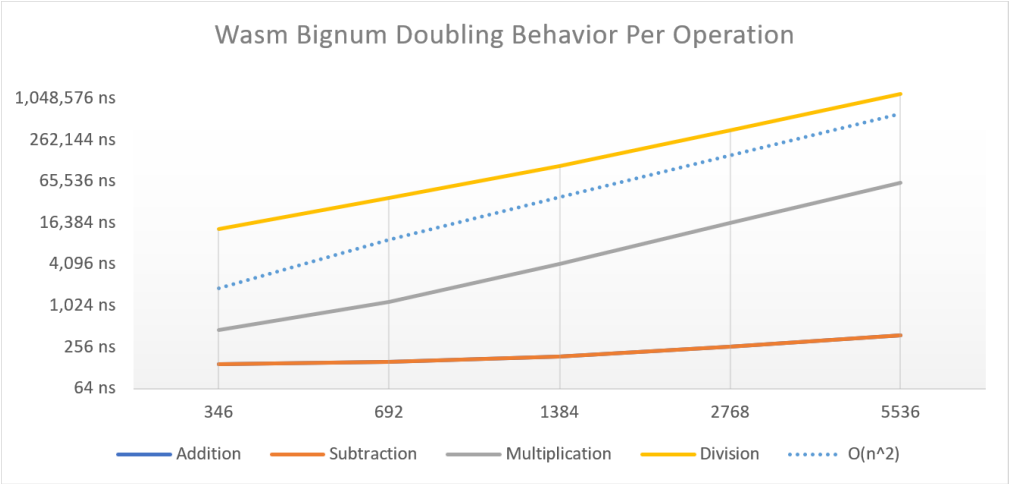
Abbreviation	Operation
`eq`	**L** == **R**
`ne`	**L** != **R**
`lt`	**L** $<$ **R**
`le`	**L** $\leq$ **R**
`gt`	**L** $>$ **R**
`ge`	**L** $\geq$ **R**

Note: All floating-point comparisons are IEEE 754-2008 compliant. These comparisons return a Boolean result, which is represented in WebAssembly as an i32. Any non-zero value is considered true, while 0 is interpreted as false.

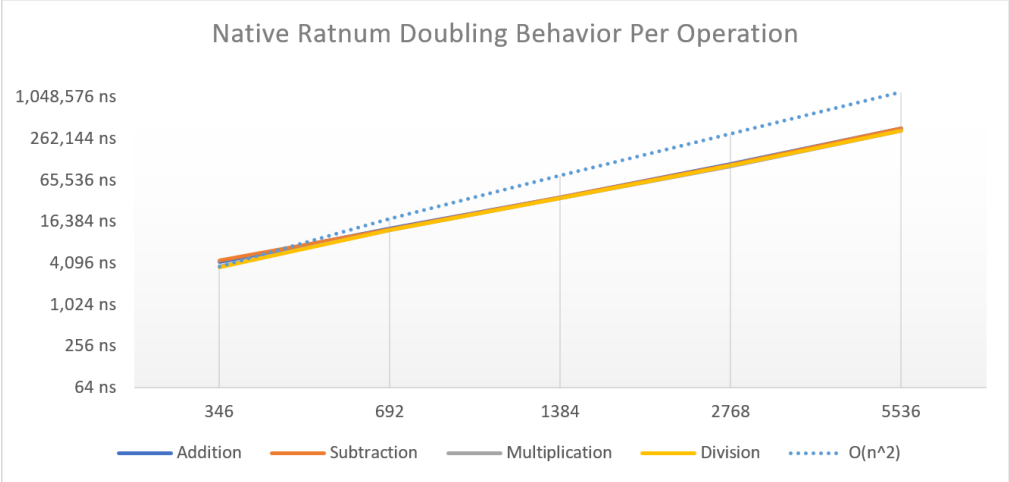
B Operation Doubling Behavior Comparison, Native vs Webassembly

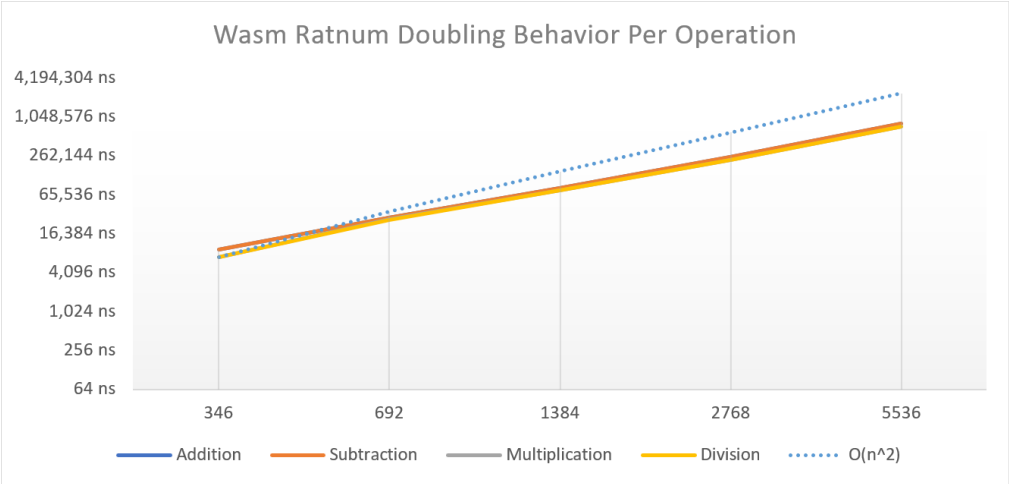
B.1 Bignums



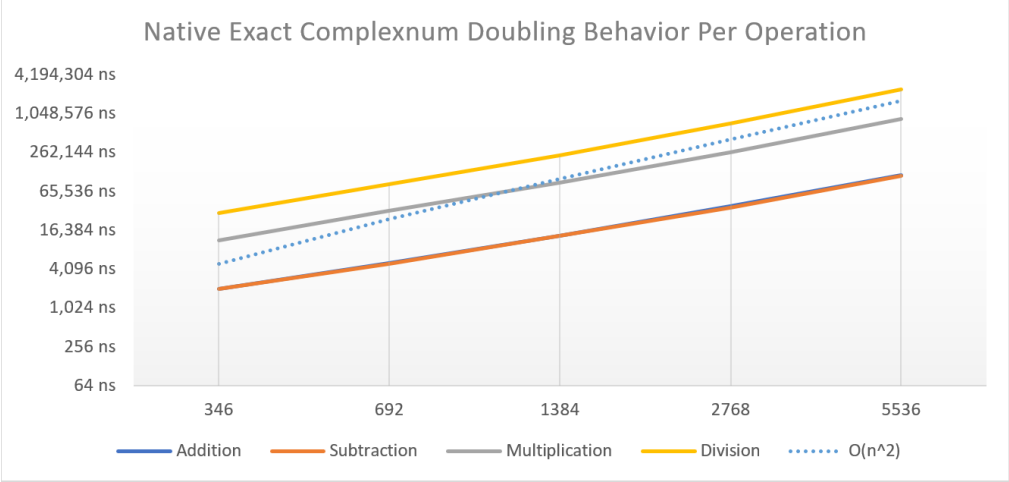

Wasm Bignum Doubling Behavior Per Operation
1,048,576 ns
262,144 ns
65,536 ns
16,384 ns
4,096 ns
1,024 ns
256 ns
64 ns
346
692
1384
2768
5536
Addition
Subtraction
Multiplication
Division
O(n^2)

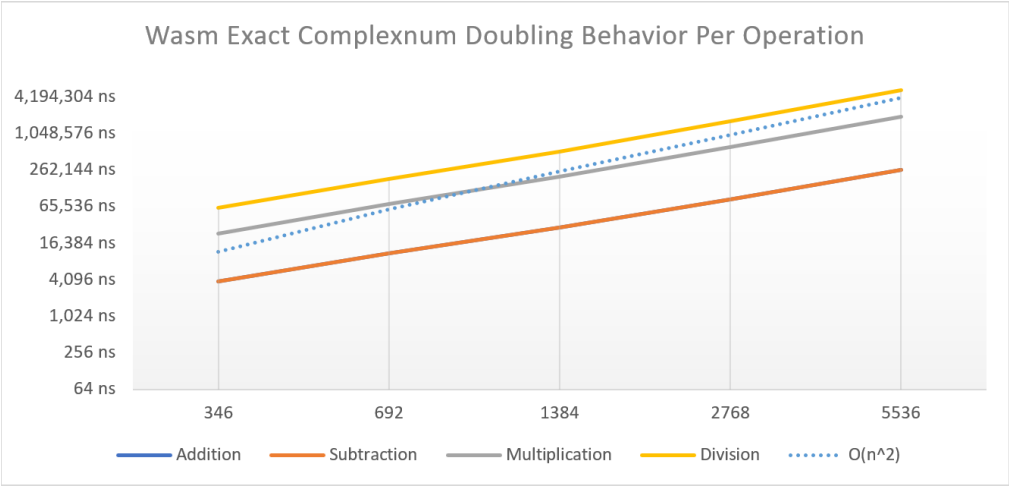
B.2 Ratnums


Native Ratnum Doubling Behavior Per Operation
1,048,576 ns
262,144 ns
65,536 ns
16,384 ns
4,096 ns
1,024 ns
256 ns
64 ns
346
692
1384
2768
5536
Addition
Subtraction
Multiplication
Division
O(n^2)



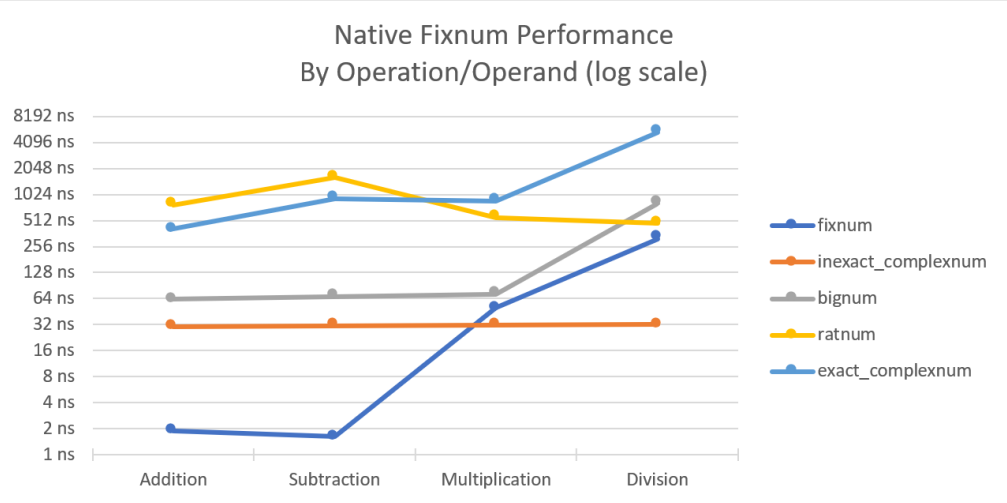
B.3 Exact Complexnums

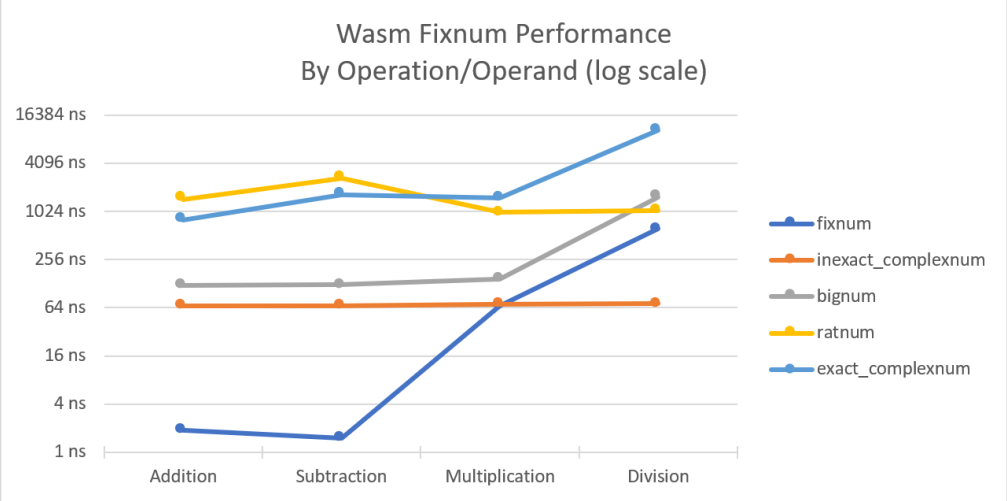



Wasm Exact Complexnum Doubling Behavior Per Operation
4,194,304 ns
1,048,576 ns
262,144 ns
65,536 ns
16,384 ns
4,096 ns
1,024 ns
256 ns
64 ns
346
692
1384
2768
5536
Addition
Subtraction
Multiplication
Division
O(n^2)

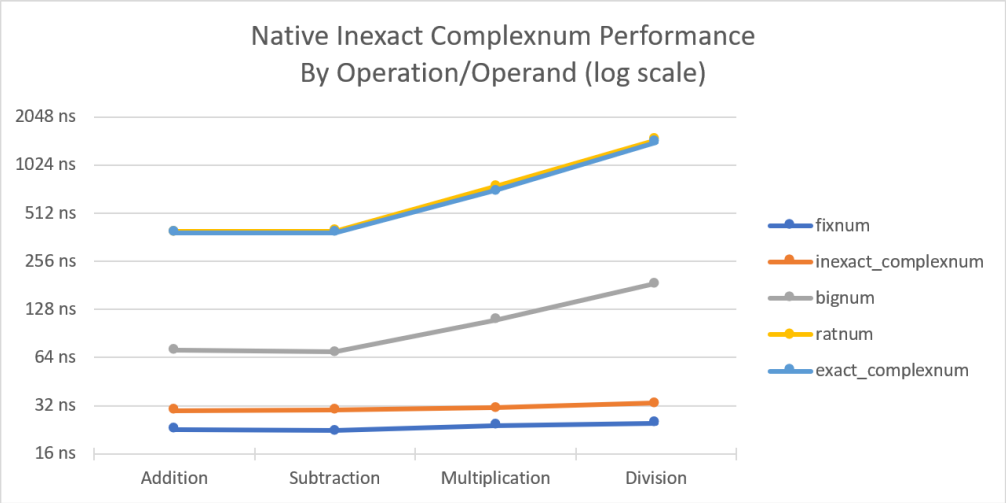
C Performance Per Operand Comparisons, Native Vs WebAssembly

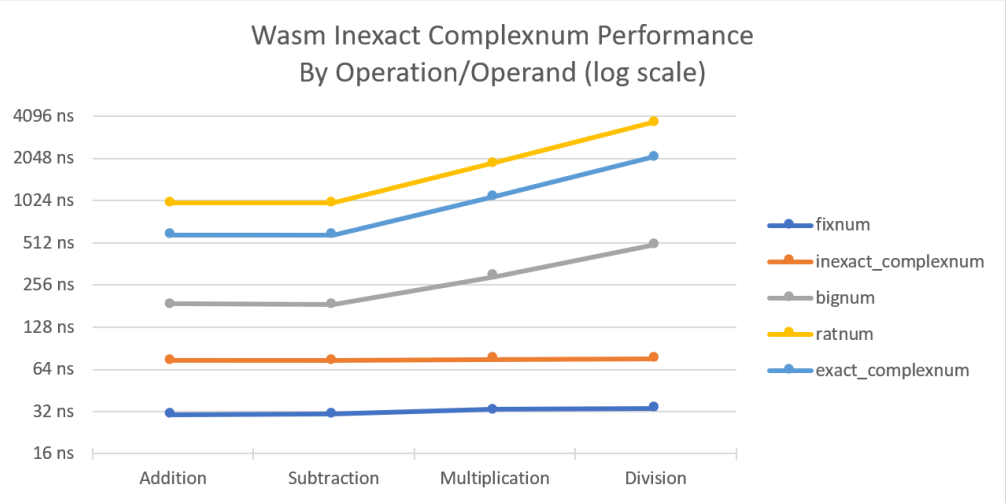
C.1 Fixnums



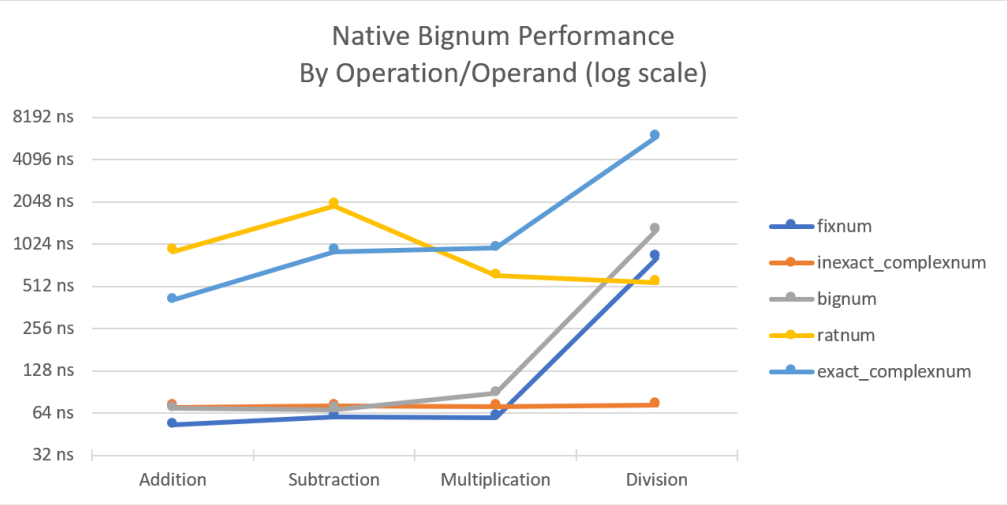


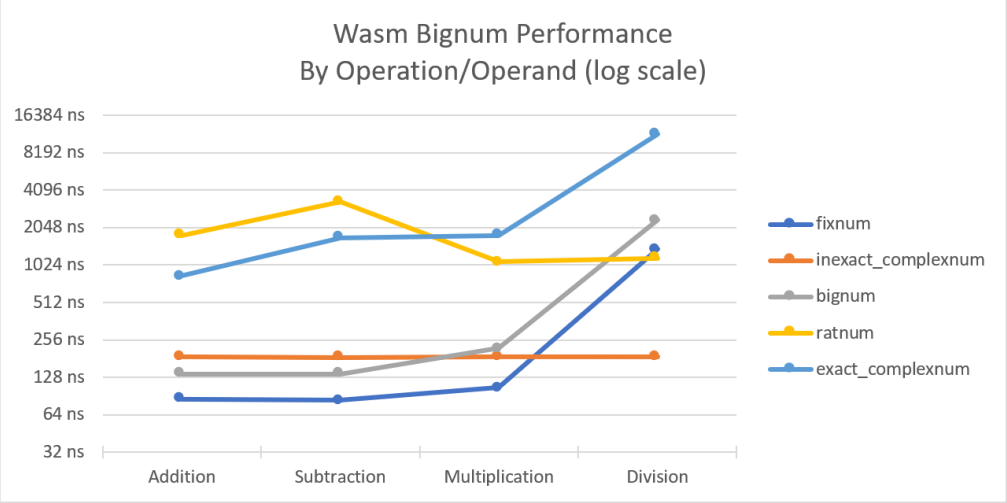
C.2 Inexact Complexnums



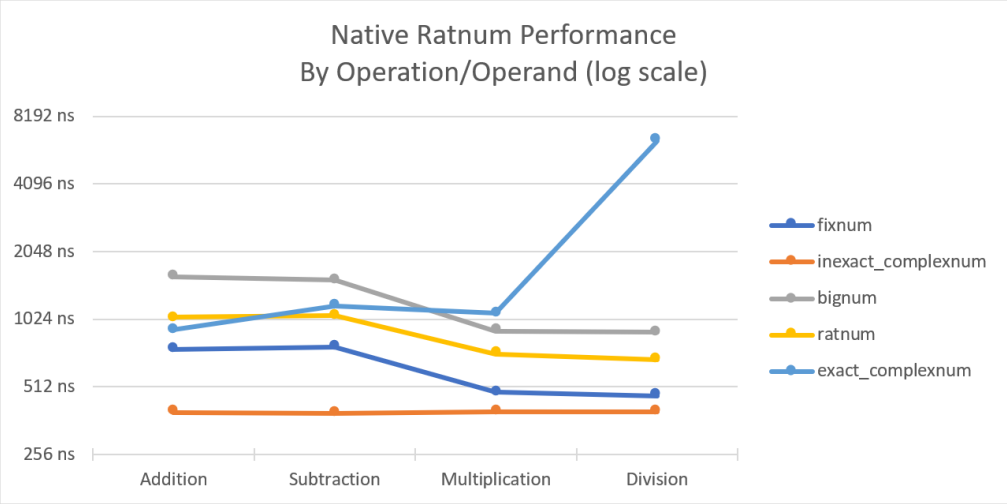


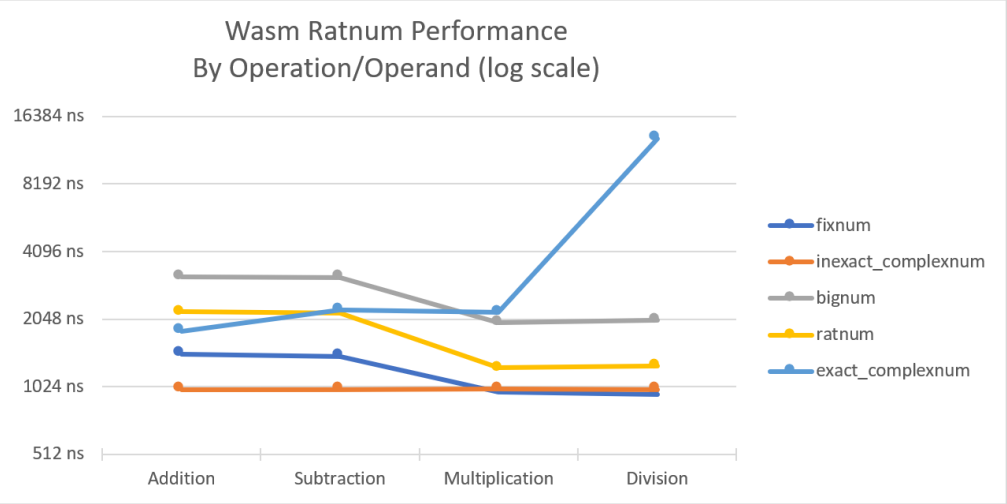
C.3 Bignums


Native Bignum Performance
By Operation/Operand (log scale)
8192 ns
4096 ns
2048 ns
1024 ns
512 ns
256 ns
128 ns
64 ns
32 ns
Addition
Subtraction
Multiplication
Division
fixnum
inexact_complexnum
bignum
ratnum
exact_complexnum

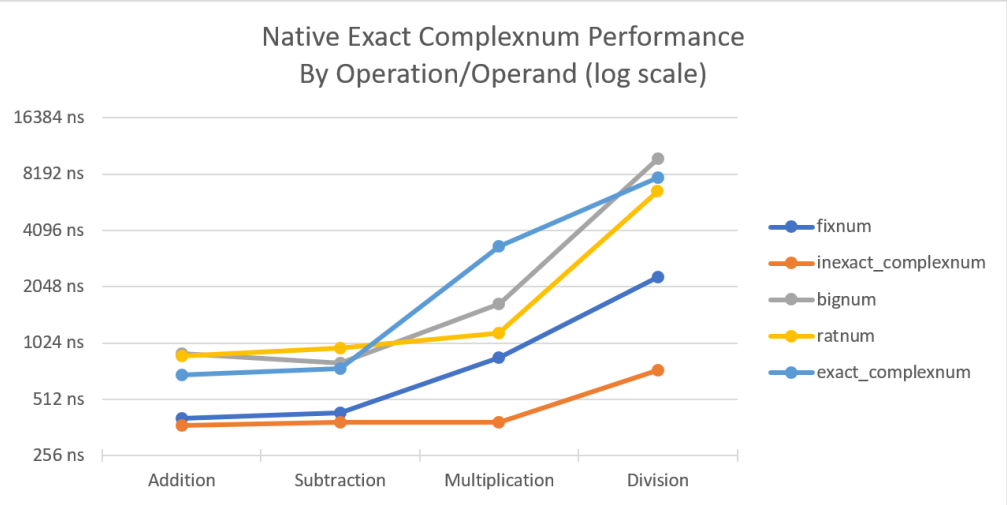

Wasm Bignum Performance
By Operation/Operand (log scale)
16384 ns
8192 ns
4096 ns
2048 ns
1024 ns
512 ns
256 ns
128 ns
64 ns
32 ns
Addition
Subtraction
Multiplication
Division
fixnum
inexact_complexnum
bignum
ratnum
exact_complexnum

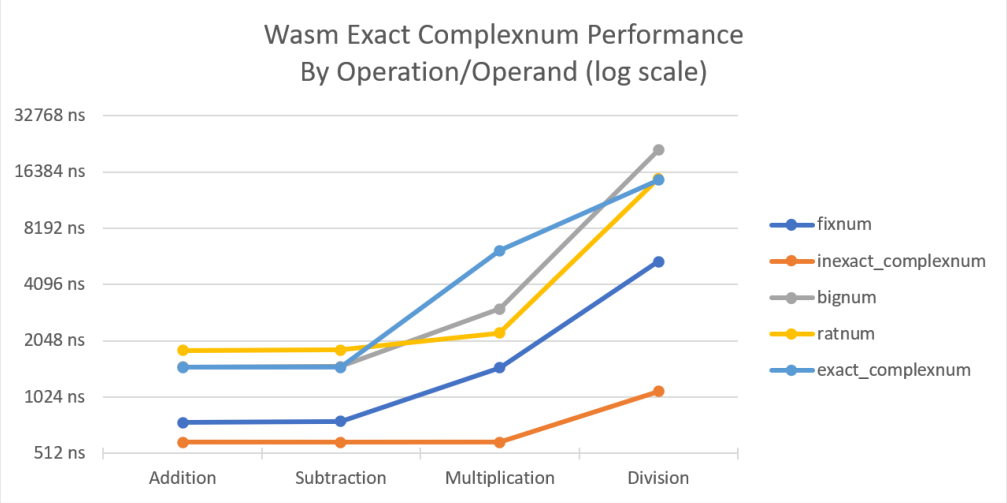
C.4 Ratnums


Native Ratnum Performance
By Operation/Operand (log scale)
8192 ns
4096 ns
2048 ns
1024 ns
512 ns
256 ns
Addition
Subtraction
Multiplication
Division
fixnum
inexact_complexnum
bignum
ratnum
exact_complexnum


Wasm Ratnum Performance
By Operation/Operand (log scale)
16384 ns
8192 ns
4096 ns
2048 ns
1024 ns
512 ns
Addition
Subtraction
Multiplication
Division
fixnum
inexact_complexnum
bignum
ratnum
exact_complexnum

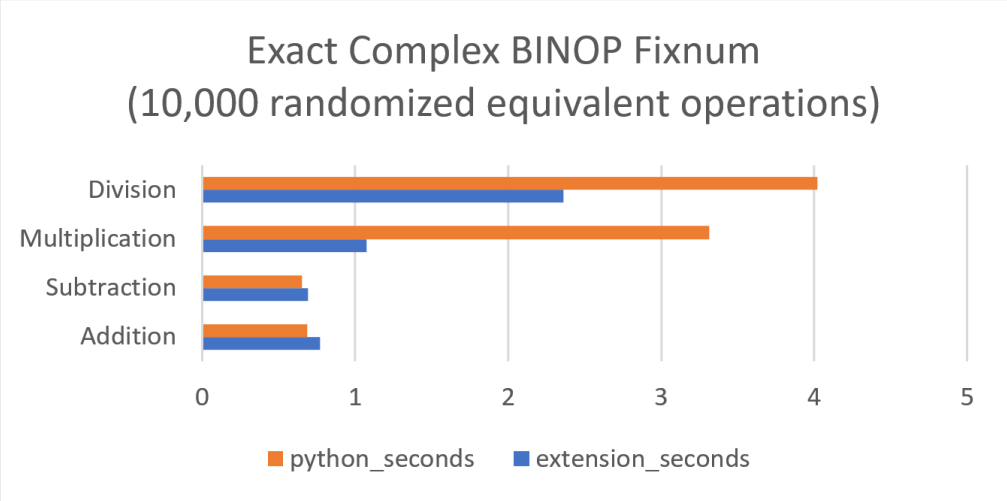
C.5 Exact Complexnums

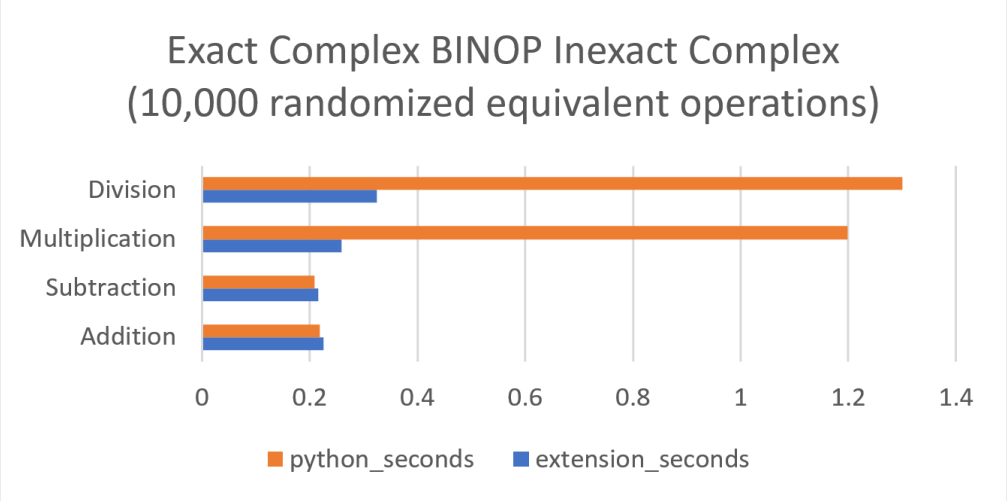


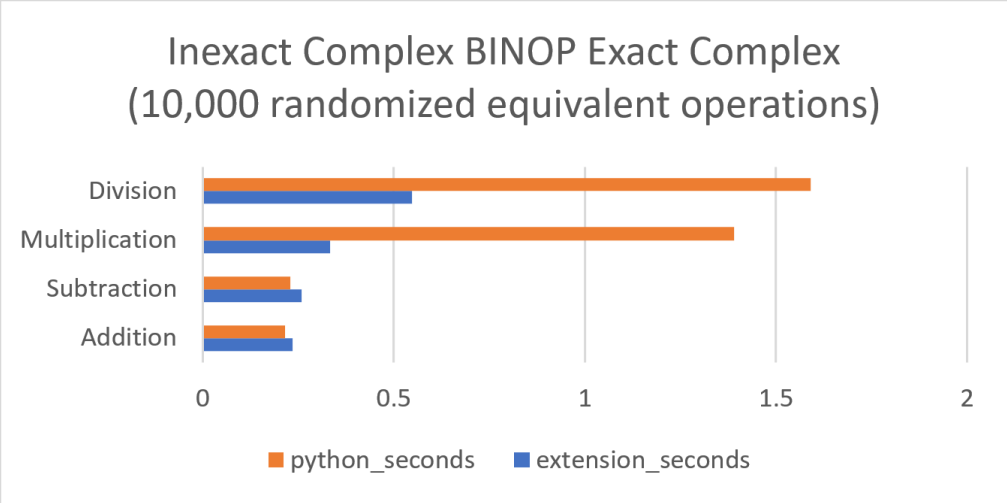


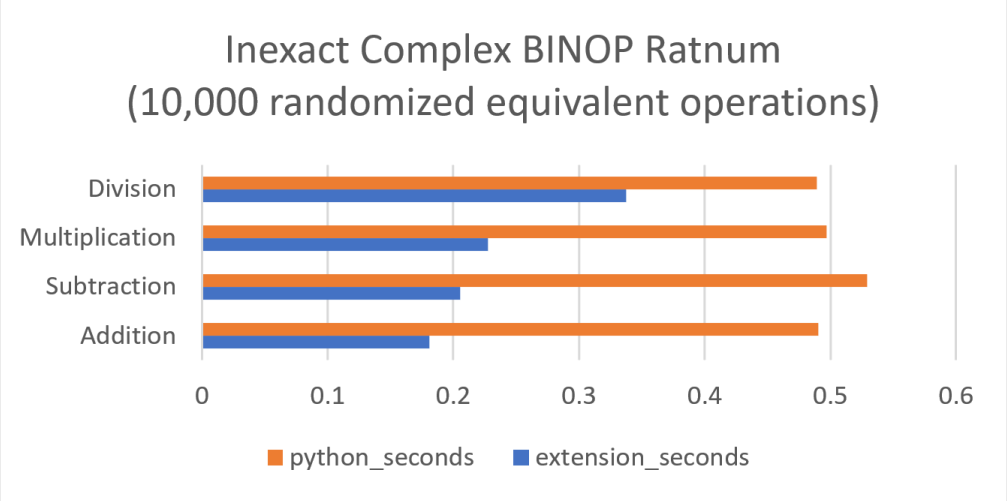
D Python Extension Performance Comparison Charts

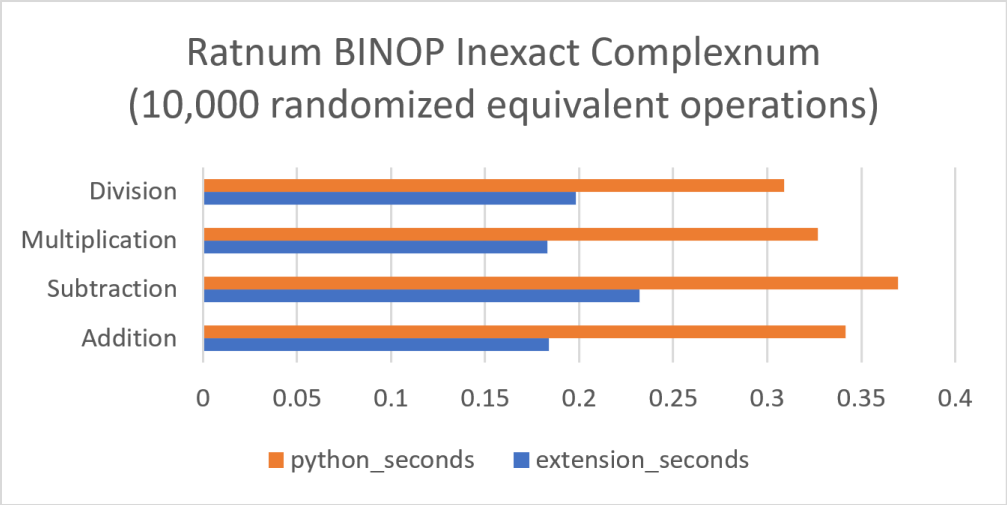
D.1 Net Benefit, Use Case Potential

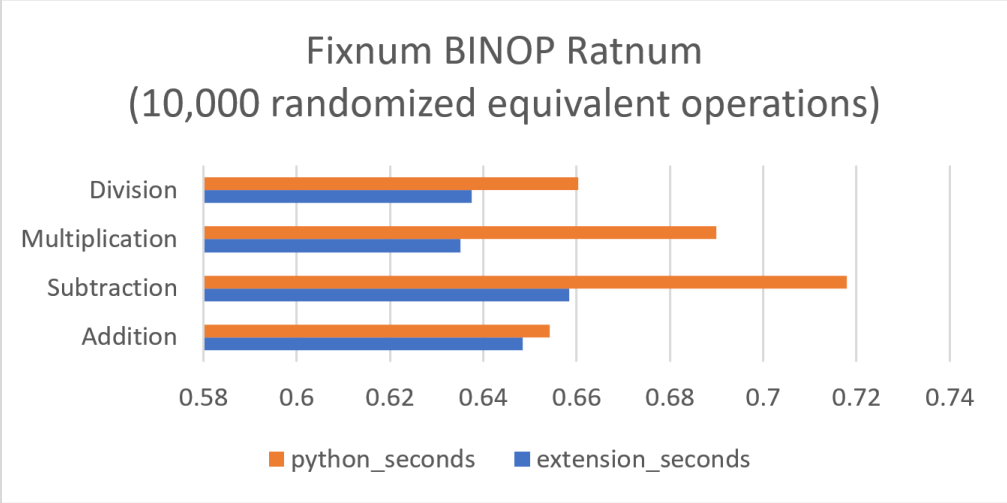




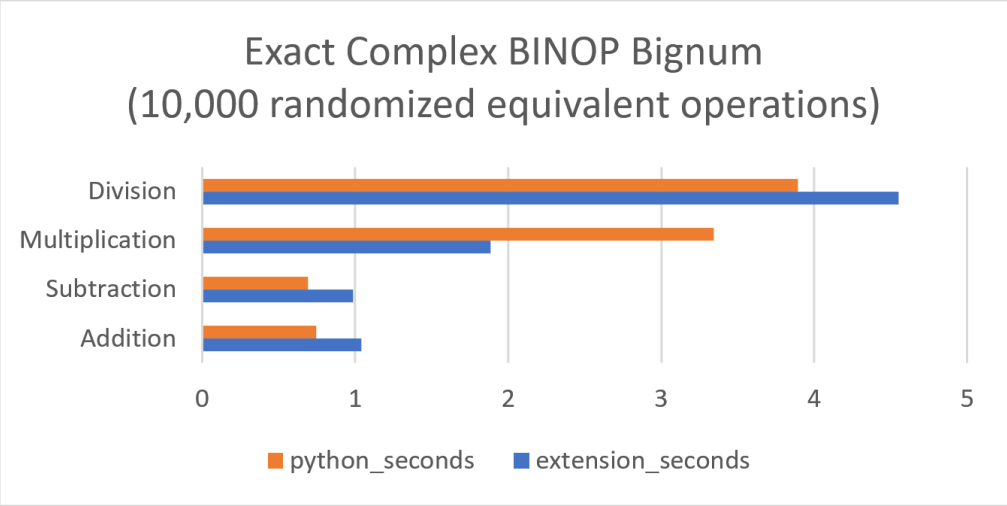

Inexact Complex BINOP Exact Complex
(10,000 randomized equivalent operations)
Division
Multiplication
Subtraction
Addition
0
0.5
1
1.5
2
python_seconds
extension_seconds

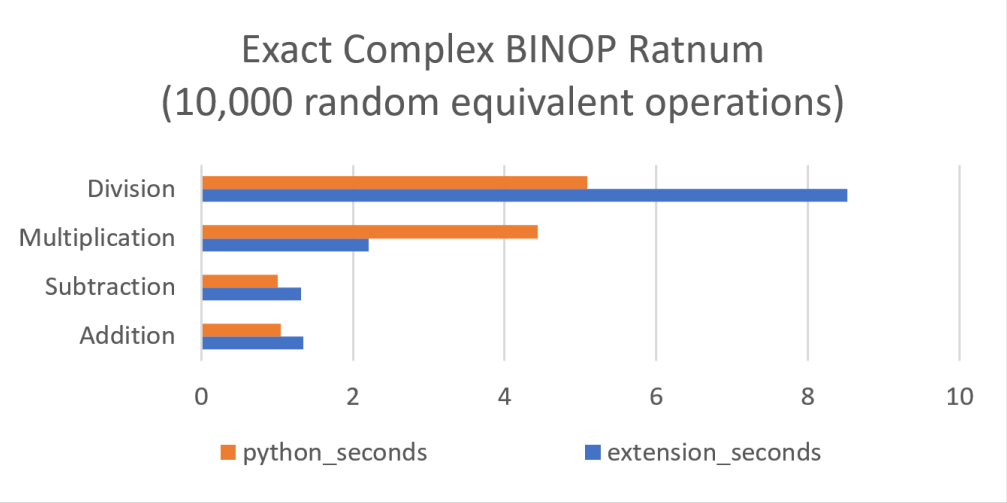

Inexact Complex BINOP Ratnum
(10,000 randomized equivalent operations)
Division
Multiplication
Subtraction
Addition
0
0.1
0.2
0.3
0.4
0.5
0.6
python_seconds
extension_seconds

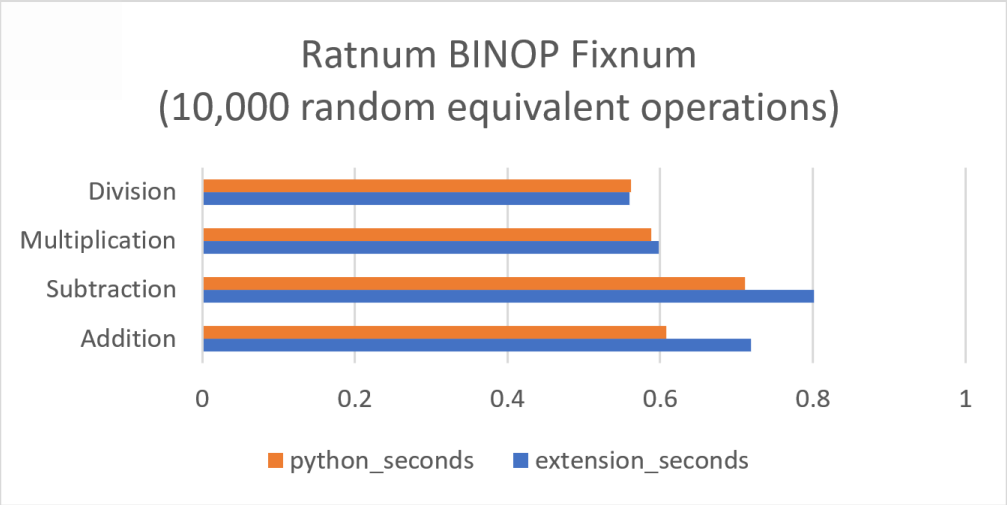

Ratnum BINOP Inexact Complexnum
(10,000 randomized equivalent operations)
Division
Multiplication
Subtraction
Addition
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
python_seconds
extension_seconds


Fixnum BINOP Ratnum
(10,000 randomized equivalent operations)
Division
Multiplication
Subtraction
Addition
0.58
0.6
0.62
0.64
0.66
0.68
0.7
0.72
0.74
python_seconds
extension_seconds

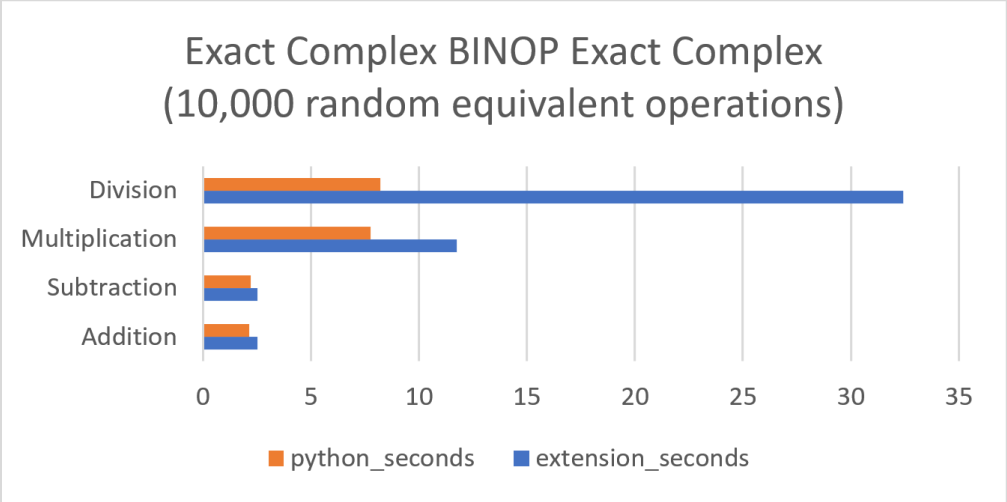
D.2 Neutral

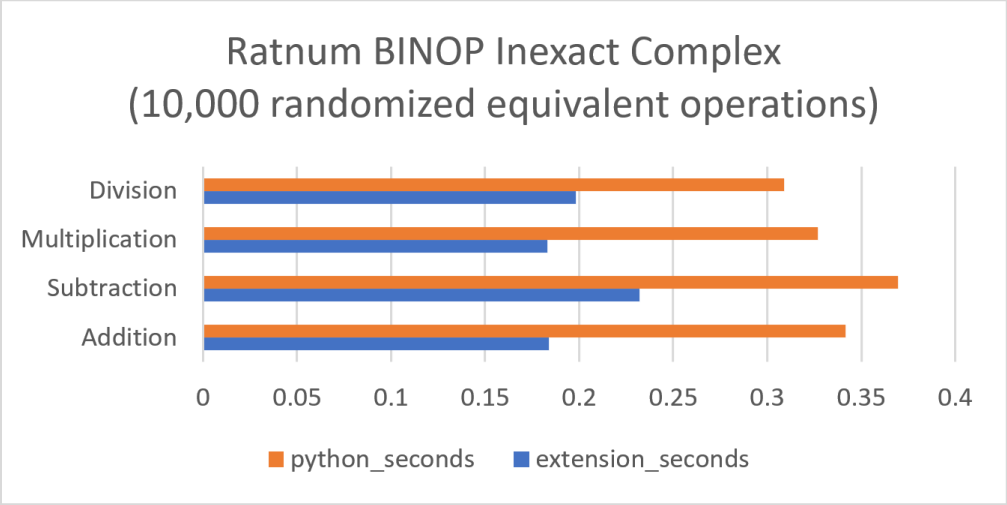

Exact Complex BINOP Bignum
(10,000 randomized equivalent operations)
Division
Multiplication
Subtraction
Addition
0
1
2
3
4
5
python_seconds
extension_seconds

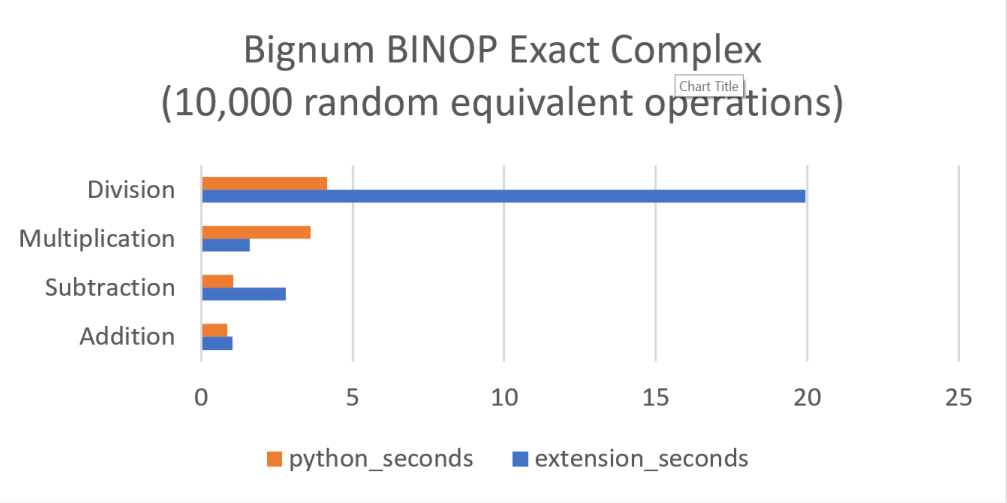

Exact Complex BINOP Ratnum
(10,000 random equivalent operations)
Division
Multiplication
Subtraction
Addition
0
2
4
6
8
10
python_seconds
extension_seconds

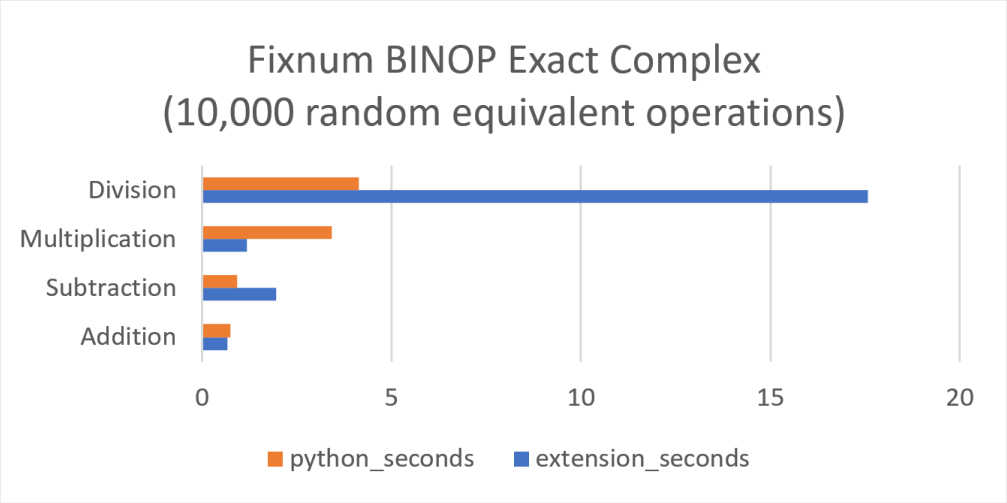

Ratnum BINOP Fixnum
(10,000 random equivalent operations)
Division
Multiplication
Subtraction
Addition
0
0.2
0.4
0.6
0.8
1
python_seconds
extension_seconds

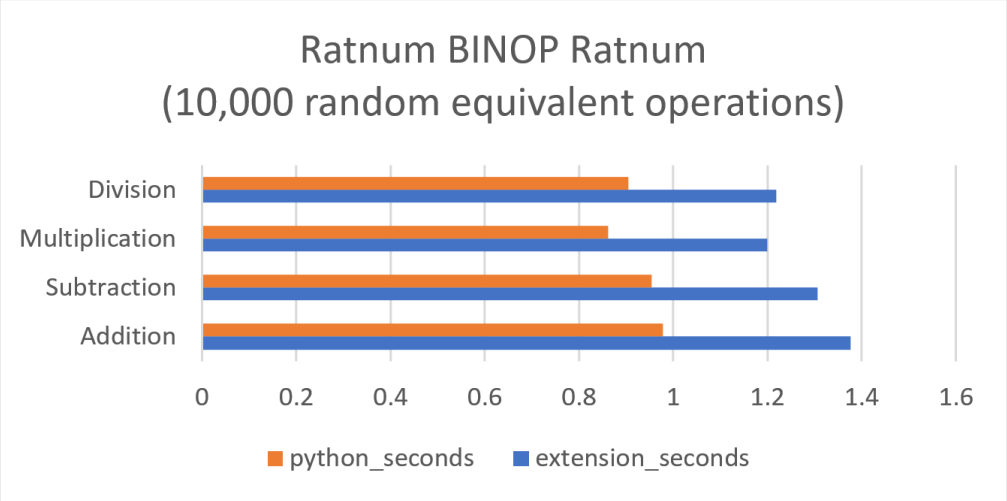
D.3 Net Loss

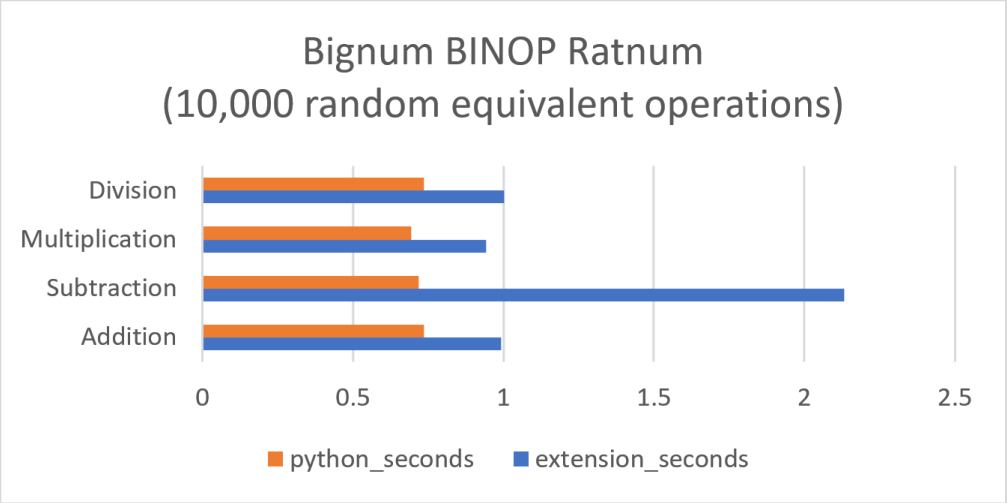

Exact Complex BINOP Exact Complex
(10,000 random equivalent operations)
Division
Multiplication
Subtraction
Addition
0
5
10
15
20
25
30
35
python_seconds
extension_seconds

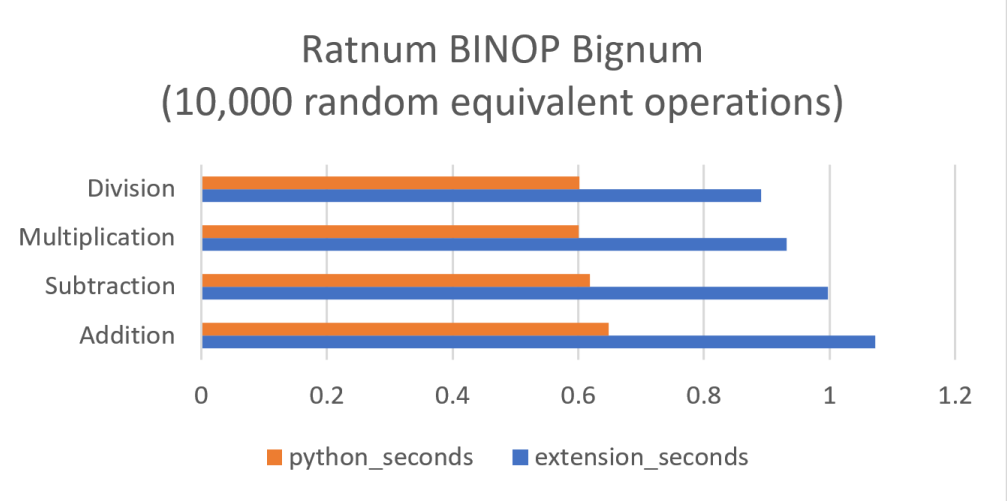

Ratnum BINOP Inexact Complex
(10,000 randomized equivalent operations)
Division
Multiplication
Subtraction
Addition
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
python_seconds
extension_seconds


Bignum BINOP Exact Complex
(10,000 random equivalent operations)
Chart Title
Division
Multiplication
Subtraction
Addition
0
5
10
15
20
25
python_seconds
extension_seconds


Fixnum BINOP Exact Complex
(10,000 random equivalent operations)
Division
Multiplication
Subtraction
Addition
0
5
10
15
20
python_seconds
extension_seconds


Ratnum BINOP Ratnum
(10,000 random equivalent operations)
Division
Multiplication
Subtraction
Addition
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
python_seconds
extension_seconds


Bignum BINOP Ratnum
(10,000 random equivalent operations)
Division
Multiplication
Subtraction
Addition
0
0.5
1
1.5
2
2.5
python_seconds
extension_seconds


Ratnum BINOP Bignum
(10,000 random equivalent operations)
Division
Multiplication
Subtraction
Addition
0
0.2
0.4
0.6
0.8
1
1.2
python_seconds
extension_seconds

E Benchmarking Environment – CADE Lab Specifications

All benchmarks reported in the results section were acquired by running benchmarking suites in the University of Utah CADE lab. The following specifications were acquired by running the hostnamectl and lscpu commands on CADE Lab1-1.

```
$ hostnamectl

Static hostname:  lab1-1.eng.utah.edu
Icon name:  computer-desktop
Chassis:  desktop
Machine ID: 642cbea72ee846839af0f1d891e6bbde
Boot ID: 595b0ed4115943128266e025a7cd009d
Operating System:  Red Hat Enterprise Linux
CPE OS Name:  cpe:/o:redhat:enterprise_linux:7.7:GA:server
Kernel:  Linux 3.10.0-1062.7.1.el7.x86_64
Architecture:  x86-64

$ lscpu

Architecture:  x86_64
CPU op-mode(s):  32-bit, 64-bit
Byte Order:  Little Endian
CPU(s):  8
On-line CPU(s) list:  0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):  1
NUMA node(s):  1
Vendor ID: GenuineIntel
CPU family:  6
Model:  60
Model name:  Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
Stepping:  3
```

```
CPU MHz:  3999.902
CPU max MHz:  4000.0000
CPU min MHz:  800.0000
BogoMIPS: 7183.85
Virtualization:  VT-x
L1d cache:  32K
L1i cache:  32K
L2 cache:  256K
L3 cache:  8192K
NUMA node0 CPU(s):  0-7

Flags:  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl
vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic
movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm
epb invpcid_single ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority
ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt
dtherm ida arat pln pts md_clear spec_ctrl intel_stibp flush_l1d
```