不同架构下浮点数表示的一致性问题探究

一、结论

在不同平台下，wasm中**NaN**的符号位表示存在**区别**，Rust编译的wasm能保证在不同架构上（x86&ARM）的不同平台（Wasmer、Wasmtime&Chrome v8）的结果一致性，而C++和手写的Wasm不能保证。

二、相关信息

2.1 浮点数运算基本定义

计算机组成原理说明：浮点数运算对于不同的芯片上的计算结果存在不确定性。

运算说明：与IEEE754标准对齐的架构都能实现IEEE754定义的基本运算。

2.2 WebAssembly中NaN相关问题

（1）ARM中NaN表示存在不确定性

来源：[ARM Developer]

结论：对于VFPv2芯片，NaN的表示方法中符号位的数值**未知**。

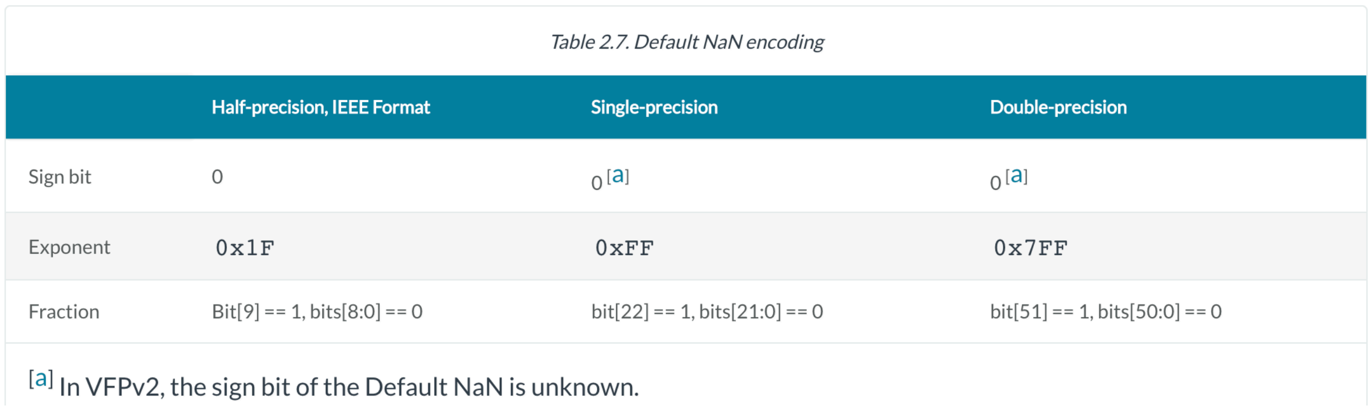


Table 2.7. Default NaN encoding

	Half-precision, IEEE Format	Single-precision	Double-precision
Sign bit	0	0 [a]	0 [a]
Exponent	`0x1F`	`0xFF`	`0x7FF`
Fraction	Bit[9] == 1, bits[8:0] == 0	bit[22] == 1, bits[21:0] == 0	bit[51] == 1, bits[50:0] == 0

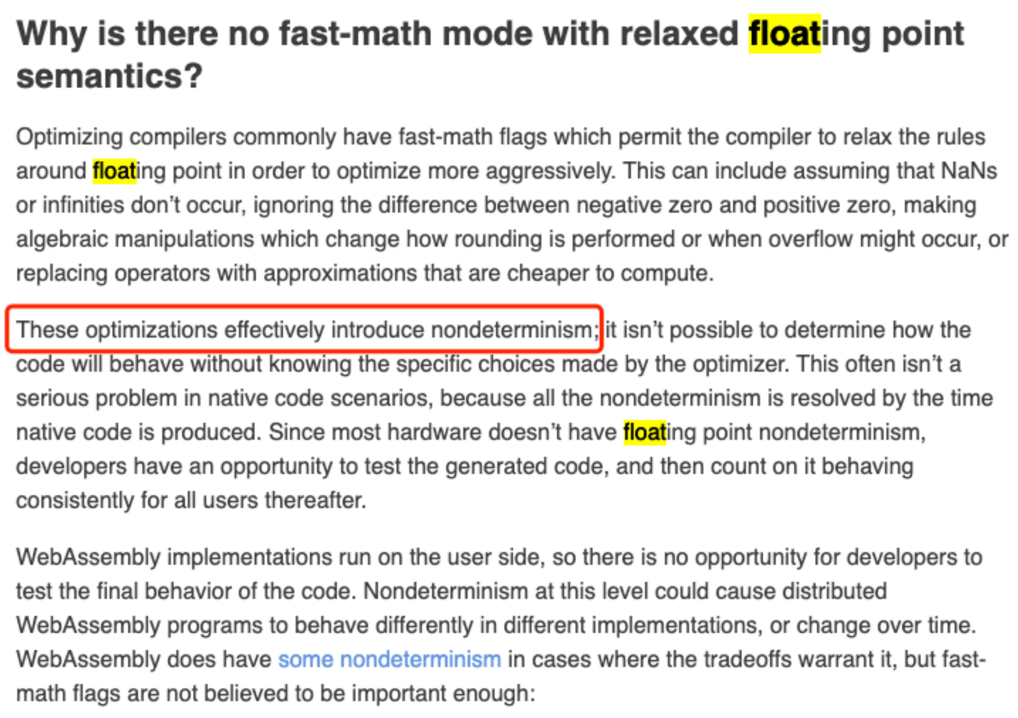
[a] In VFPv2, the sign bit of the Default NaN is unknown.

（2）Wasm中浮点数表示存在不确定性

来源：[WebAssembly官网]

结论：由于Wasm的优化，Wasm假设不存在**NaNs**和**infinities**，忽略了**-0**和**+0**的区别，数字的溢出和进位会产生不同（不确定）的处理

，导致了Wasm中浮点表示的**不确定性（nondeterminism）**。



Why is there no fast-math mode with relaxed floating point semantics?

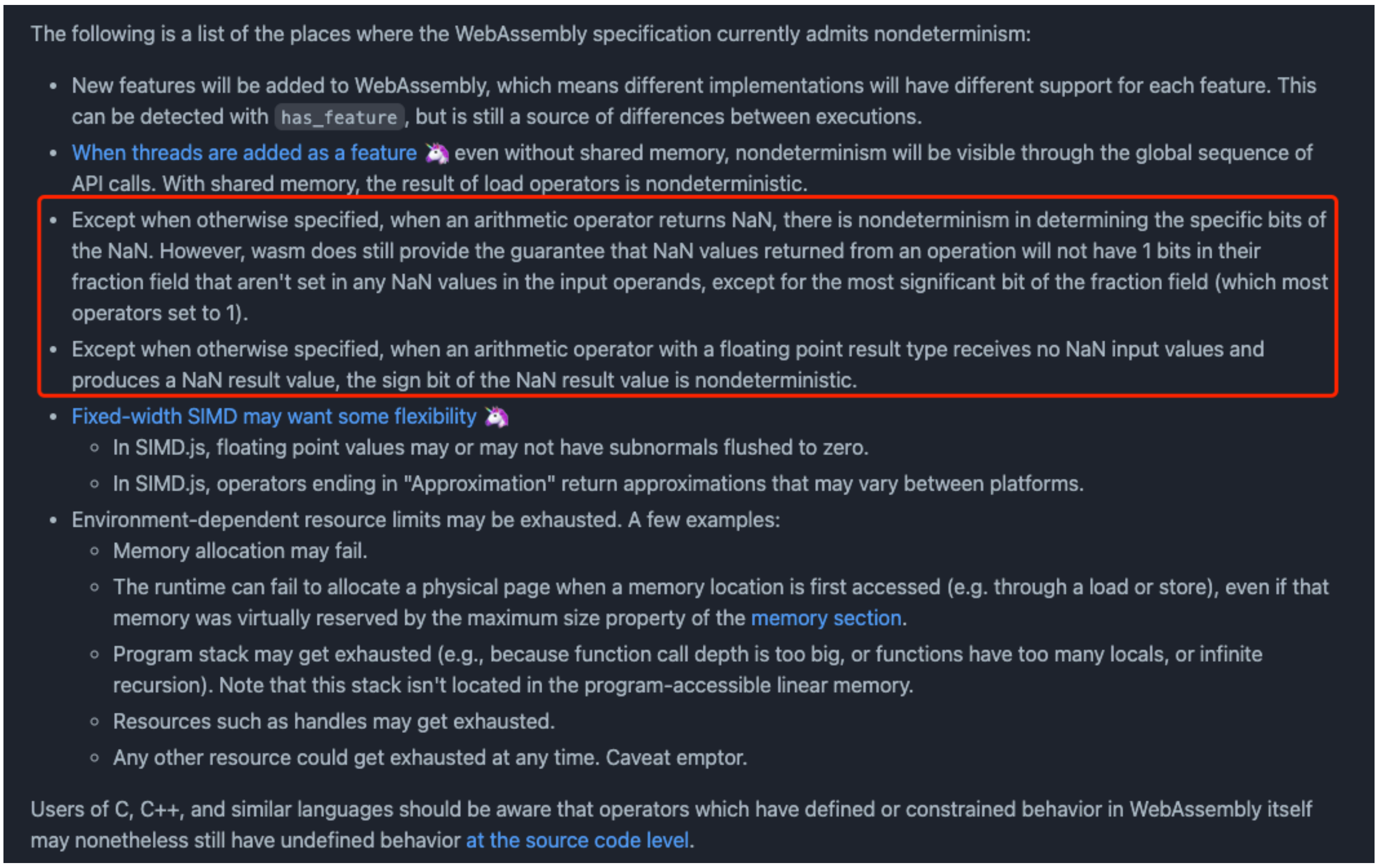
Optimizing compilers commonly have fast-math flags which permit the compiler to relax the rules around floating point in order to optimize more aggressively. This can include assuming that NaNs or infinities don't occur, ignoring the difference between negative zero and positive zero, making algebraic manipulations which change how rounding is performed or when overflow might occur, or replacing operators with approximations that are cheaper to compute.

These optimizations effectively introduce nondeterminism; it isn't possible to determine how the code will behave without knowing the specific choices made by the optimizer. This often isn't a serious problem in native code scenarios, because all the nondeterminism is resolved by the time native code is produced. Since most hardware doesn't have floating point nondeterminism, developers have an opportunity to test the generated code, and then count on it behaving consistently for all users thereafter.

WebAssembly implementations run on the user side, so there is no opportunity for developers to test the final behavior of the code. Nondeterminism at this level could cause distributed WebAssembly programs to behave differently in different implementations, or change over time. WebAssembly does have some nondeterminism in cases where the tradeoffs warrant it, but fast-math flags are not believed to be important enough:

来源：[WebAssembly github]

结论：当算术运算返回NaN时，对应在Wasm中的表示是不确定的（nondeterminism）。



The following is a list of the places where the WebAssembly specification currently admits nondeterminism:

- New features will be added to WebAssembly, which means different implementations will have different support for each feature. This can be detected with `has_feature`, but is still a source of differences between executions.
- When threads are added as a feature 🦄 even without shared memory, nondeterminism will be visible through the global sequence of API calls. With shared memory, the result of load operators is nondeterministic.
- Except when otherwise specified, when an arithmetic operator returns NaN, there is nondeterminism in determining the specific bits of the NaN. However, wasm does still provide the guarantee that NaN values returned from an operation will not have 1 bits in their fraction field that aren't set in any NaN values in the input operands, except for the most significant bit of the fraction field (which most operators set to 1).
- Except when otherwise specified, when an arithmetic operator with a floating point result type receives no NaN input values and produces a NaN result value, the sign bit of the NaN result value is nondeterministic.
- Fixed-width SIMD may want some flexibility 🦄
  - In SIMD.js, floating point values may or may not have subnormals flushed to zero.
  - In SIMD.js, operators ending in "Approximation" return approximations that may vary between platforms.
- Environment-dependent resource limits may be exhausted. A few examples:
  - Memory allocation may fail.
  - The runtime can fail to allocate a physical page when a memory location is first accessed (e.g. through a load or store), even if that memory was virtually reserved by the maximum size property of the memory section.
  - Program stack may get exhausted (e.g., because function call depth is too big, or functions have too many locals, or infinite recursion). Note that this stack isn't located in the program-accessible linear memory.
  - Resources such as handles may get exhausted.
  - Any other resource could get exhausted at any time. Caveat emptor.

Users of C, C++, and similar languages should be aware that operators which have defined or constrained behavior in WebAssembly itself may nonetheless still have undefined behavior at the source code level.

来源：Github issues

issue1

issue2

结论：Wasm中除了NaN，浮点表示都是确定的。

来源：[Wasm 官方doc]

> NaNs produced by floating-point instructions in WebAssembly have nondeterministic bit patterns in most circumstances. The bit pattern of a NaN is not usually significant, however there are a few ways that it can be observed:

结论：NaN 表示方式不确定。

三、实证研究

3.1 NaN是否存在不确定性

结论：

对于Rust编译而成的Wasm而言，运算产生`NaN`的情况只有`0.0/0.0`，其中`NaN`的`f32`表示在两种架构中表示均为`[127, 192, 0, 0]`，`f64`表示均为`[127, 248, 0, 0]`。

所有有关NaN的运算在ARM和x86结构的三种环境下运行结果和表示均相同。

而对于C++编译而成Wasm而言，在Chrome v8下运行产生的NaN的表示**有别**，即符号位**相反**。

对于手写的WAT而言，其在不同架构（Wasmer和Wasmtime）的相同运行环境下的结果**有别**，符号位**相反**。

总结**一致性**表格如下：

语言	Wasmer	Wasmtime	Chrome v8
Rust	√	√	√
C++			×
手写wat	×	×	

具体结果如下，以`f32`中的`NaN`表示为例：

语言	Wasmer	Wasmtime	Chrome v8
Rust	均为[127, 192, 0, 0]	均为[127, 192, 0, 0]	均为[127, 192, 0, 0]
C++			x86：[255, 192, 0, 0] ARM:[127, 192, 0, 0] 符号位有别
手写wat	x86：[255, 192, 0, 0] ARM:[127, 192, 0, 0] 符号位有别	x86：[255, 192, 0, 0] ARM:[127, 192, 0, 0] 符号位有别	

3.1.1 测试思路

1.测试环境

- ARM
- x86

2.测试RunTime

- wasmer
- wasmtime
- chrome v8

3.测试代码

- Rust编译后的Wasm
- C++编译后的Wasm
- 手写Wat

3.1.2 Rust编译的Wasm测试

测试代码

```
fn main() {
    let nan = 0.0_f32 / 0.0;
    println!("nan: {:?}", nan.to_be_bytes());
    // 相关运算测试
    // ...
}
```

测试结果：

1.Rust编译后的Wasm在同一个架构下不同运行环境中的结果相同

2.Rust编译后的Wasm在不同架构下的同一运行环境中结果相同

以f32::NAN为例，x86和ARM下不同运行环境结果均为：

$$[127, 192, 0, 0]$$

3.1.3 C++编译的Wasm测试

测试代码

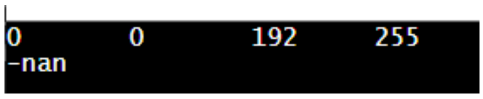
```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char ** argv) {
  float nan1 = 0.0;
  float nan2 = 0.0;
  float nan3 = (nan1 / nan2);
  unsigned char charArray[4];
  unsigned char *pdatas = (unsigned char *)&nan3;
  for (int i = 0; i < 4; i++){
    charArray[i] = *pdatas++;
    printf("%d\t", int(charArray[i]));
  }
  cout << endl;
  cout << to_string(nan3) << endl;
  return 0;
}
```

测试结果：

1.C++编译后的Wasm在同一个架构下不同运行环境中的结果相同

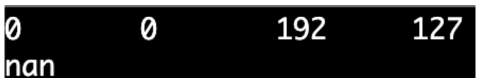
2.C++编译后的Wasm在不同架构下的同一运行环境中结果**不相同（chrome v8下运行结果不同）**

x86下chrome v8



```
0       0       192     255
-nan
```

ARM下chrome v8



```
0       0       192     127
nan
```

3.1.4 手写WAT测试

测试代码

```
(module
            (type $add_one_t (func (param f32) (result f32)))
            (func $add_one_f (type $add_one_t)  (param $value f32) (result f32)
              local.get $value
              f32.const 0.0
              f32.div)
            (export "add_one" (func $add_one_f))
          )

(module
            (func $add_one_f (result f32)
              f32.const 0.0
              f32.const 0.0
              f32.div)
            (export "add_one" (func $add_one_f))
          )
```

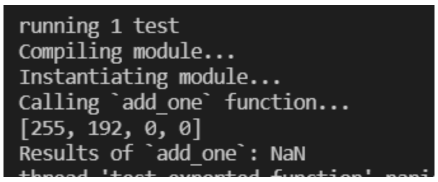
测试结果：

1.在同一个架构下不同运行环境中的结果相同

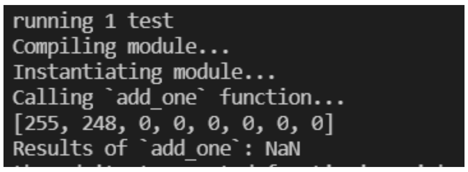
2.在不同架构下的同一运行环境中结果**不相同**

x86下

1.wasmer（f32和f64）

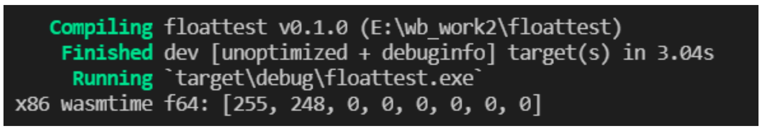


```
running 1 test
Compiling module...
Instantiating module...
Calling `add_one` function...
[255, 192, 0, 0]
Results of `add_one`: NaN
```

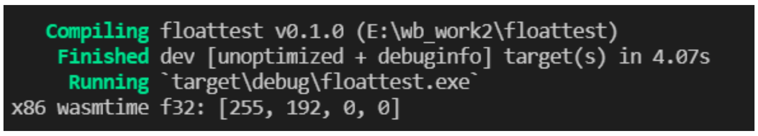


```
running 1 test
Compiling module...
Instantiating module...
Calling `add_one` function...
[255, 248, 0, 0, 0, 0, 0, 0]
Results of `add_one`: NaN
```

2.wasmtime（f32和f64）



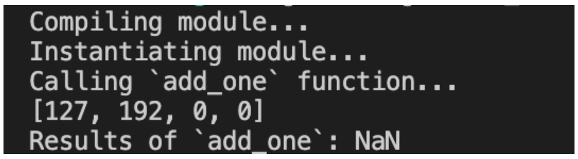
```
   Compiling floattest v0.1.0 (E:\wb_work2\floattest)
    Finished dev [unoptimized + debuginfo] target(s) in 3.04s
     Running `target\debug\floattest.exe`
x86 wasmtime f64: [255, 248, 0, 0, 0, 0, 0, 0]
```



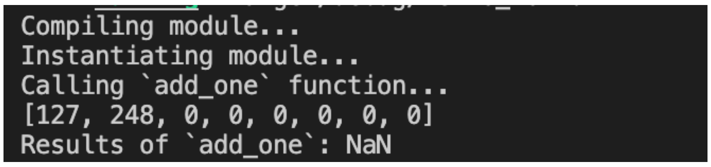
```
   Compiling floattest v0.1.0 (E:\wb_work2\floattest)
    Finished dev [unoptimized + debuginfo] target(s) in 4.07s
     Running `target\debug\floattest.exe`
x86 wasmtime f32: [255, 192, 0, 0]
```

ARM下

1.wasmer（f32和f64）

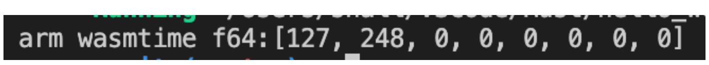


```
Compiling module...
Instantiating module...
Calling `add_one` function...
[127, 192, 0, 0]
Results of `add_one`: NaN
```

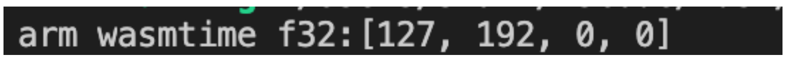


```
Compiling module...
Instantiating module...
Calling `add_one` function...
[127, 248, 0, 0, 0, 0, 0, 0]
Results of `add_one`: NaN
```

2.wasmtime（f32和f64）



```
arm wasmtime f64:[127, 248, 0, 0, 0, 0, 0, 0]
```



```
arm wasmtime f32:[127, 192, 0, 0]
```