A Study of the `rsqrt` and `rcp` Instructions on Intel and AMD Platforms

J.M. Arnold

February 20, 2016

DRAFT

Background

Consider this short program (provided by V. Innocente):

```
#include <cmath>
#include <cstdio>
int main(int n, char* v[]) {
  float fn = n - 1.f;
  float k = 0.1f + fn;
  float q = 1.f/std::sqrt(k);
  prini("%a␣%a\n",k,q);
  return 0;
}
```

When run with no arguments, it should compute the IEEE 754 Binary32 (i.e., single precision or `float`) value of $1/\sqrt{0.1}$. When compiled with `gcc 5.2.0` and the `-O2 -mavx` options, the generated code uses a `sqrt_ss` instruction followed by a `div_ss` instruction to compute the result. However, if the `-Ofast` option is used instead of `-O2`, the generated code computes the result using an `rsqrt_ss` instruction followed by a single Newton-Raphson interation. In this latter case, the output of the *exact same* executable produces different results depending on whether it is executed on an Intel CPU or an AMD CPU:

- Intel: `q = 0x1.94c580p+1`
- AMD: `q = 0x1.94c582p+1`

Note that there is a 2 *ulp* difference in the results.

Another observation: when compiled with `icc 16.0` and the `-Ofast -mavx` options, the result is different from that obtained when compiling with `gcc 5.2.0` even when run on the same hardware platform.

Table 1 shows the program results obtained on Intel and AMD processors when compiled with `gcc` and `icc`, both using the `-Ofast -mavx` options.

gcc on Intel	3.16227722	0x1.94c580p+1	0x404a62c0
gcc on AMD	3.16227770	0x1.94c584p+1	0x404a62c2
icc on Intel	3.16227746	0x1.94c582p+1	0x404a62c1
icc on AMD	3.16227770	0x1.94c584p+1	0x404a62c2
Exact	3.16227770	0x1.94c584p+1	0x404a62c2

Table 1

The goal of this invesigation was to explain these various differences.

Reciprocal square root

Let x be a positive non-zero normal IEEE 754 Binary32 number with $x = 2^e m$ where $1 \leq m < 2$. Find the integer l such that $1 \leq x' = 4^l x < 4$. Then

$$\frac{1}{\sqrt{2}} < \frac{1}{\sqrt{x'}} \leq 1 \text{ if } x' \in [1, 2)$$

DRAFT

$$\frac{1}{2} < \frac{1}{\sqrt{x'}} \leq \frac{1}{\sqrt{2}} \text{ if } x' \in [2, 4)$$

Because scaling by a power of 2 is an exact operation in IEEE 754 binary arithmetic, we can, without loss of generality, restrict our analysis to the behavior of `rsqrt` in $[1, 4)$.

To compute $1/\sqrt{x'}$, we take an initial estimate $y_0 \approx 1/\sqrt{x'}$ and use a Newton-Raphson interation to improve that estimate. If we use Newton's method to find a root of the function $f(y) = 1/y^2 - x'$, we obtain the recurrence

$$y_{n+1} \leftarrow (3y_n - x'y_n^3)/2$$

It can be shown that each such interation doubles the accuracy of y_n.

The basic use of the `rsqrt` instruction then is to provide the initial estimate y_0. The code generated by the compiler consists of the `rsqrt` instruction followed by instructions which implement a single Newton-Raphson interation to refine that result.

DRAFT

Behavior of the `rsqrt` instruction

Intel

A study was made of the results computed by the `rsqrt` instruction for all positive non-zero normal Binary32 arguments. It was first verified that the results for arguments in the range $[1, 4)$, when suitably scaled by a power of 2 as outlined above, agreed with the results returned by `rsqrt` for every possible positive normal non-zero Binary32 argument. This justifies the assumption that only the behavior of `rsqrt` in $[1, 4)$ needs to be studied. (One of the advantages of working with the Binary32 datatype in such a restricted argument range is that exhaustive testing is quite feasible: no test reported here took longer than several minutes to run.)

DRAFT

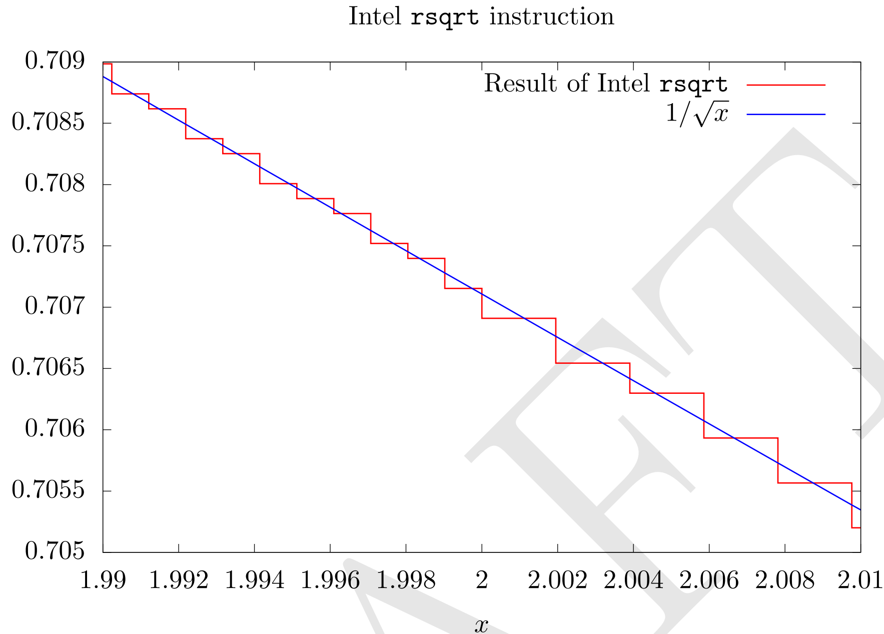

Figure 1: Value of the `rsqrt` instruction on an Intel platform

DRAFT

Examination of the results returned by `rsqrt` shows that they depend only on the sign, exponent and high-order 10 bits of the characteristic of the argument, excluding the so-called *hidden bit*. (The hidden bit is excluded since it is always 1 for normal values.) Thus the low-order 13 bits of the characteristic of an argument do not affect the result: as the arguments span a binade, the results of the instruction are a series of steps. We can think of the output of the instruction in terms of bins: the results consists of 1024 bins for each binade, and each bin corresponds to the result for 8192 contiguous arguments.

Figure 1 shows this behavior. Notice that the width of the bins is constant within a binade but changes by a factor of 2 when moving into a different binade.

Relative error plots were created as follows:

- Let each binade be divided into 1024 equal bins b_i following the pattern of the `rsqrt` instruction as implemented by Intel.
- For each bin b_i, let

$$\epsilon_i = \max_{x \in b_i} \left| \frac{F(x) - f(x)}{F(x)} \right|$$

  where $f(x)$ is the function being evaluated and $F(x)$ is the reference function. In the current study, $f(x)$ is the result of the `rsqrt` instruction for the argument x and $F(x)$ is the result of `1.0/std::sqrt((double)x)`.

- plot $log_2(\epsilon_i)$ versus $\min_{x \in b_i} x$ for all bins.

The low-order 11 bits of the characteristic of all results returned by the `rsqrt` instuction on Intel CPUs are zero, giving approximately 13 bits of accuracy as shown in Figure 2. For reference, the maximum relative error of 1.5×2^{-12} documented by Intel is shown as well.

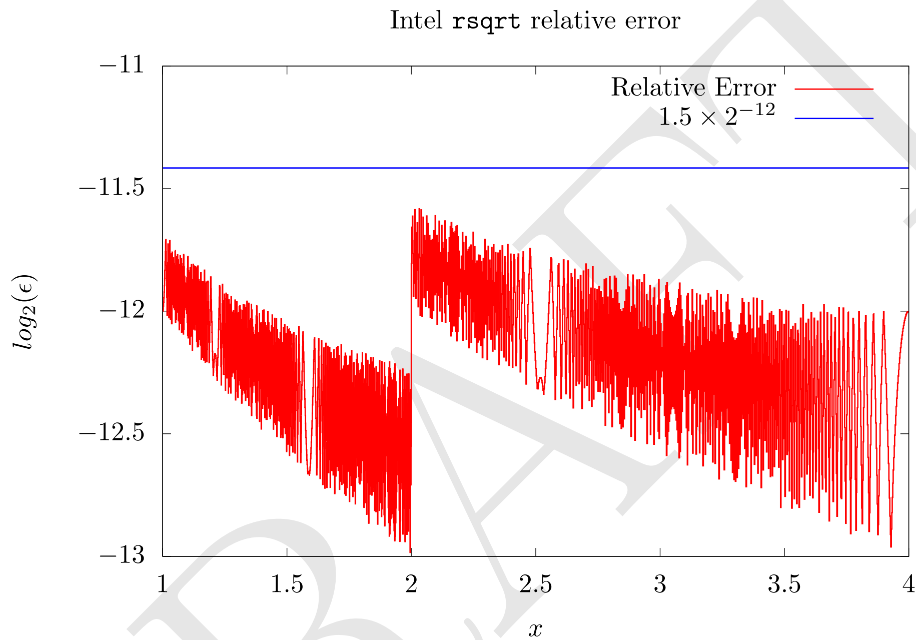

Figure 2: Relative error the `rsqrt` instruction on an Intel platform

AMD

Although the characteristics of the results returned by the `rsqrt` instruction on an AMD processor are somewhat similar to that on Intel CPUs, they differ in two very important ways.

DRAFT

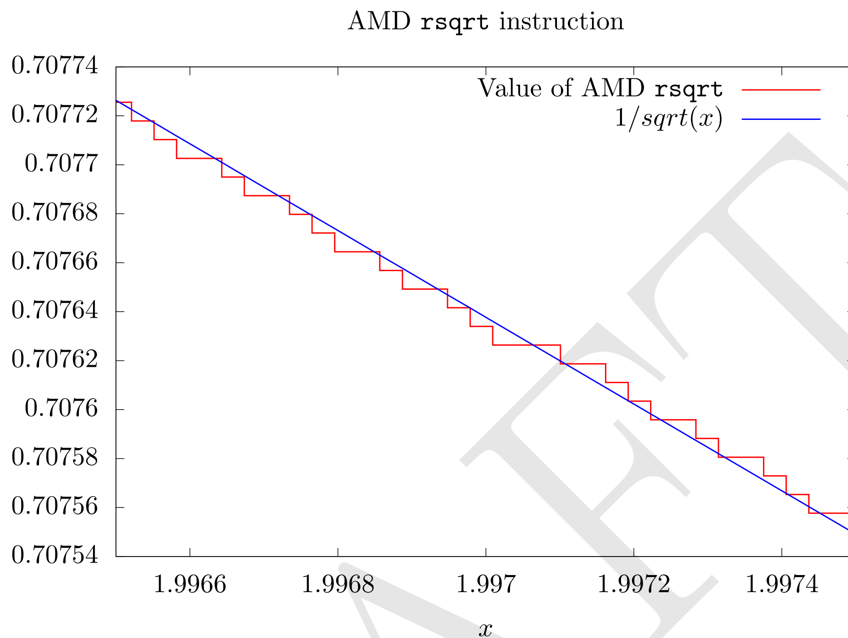

Figure 3: Relative error of the `rsqrt` instruction on an AMD platform

DRAFT

First, the steps are not all of the same width; although most are 256 arguments wide, others are either 512 or 768 arguments in width.

Number of arguments in bin	Frequency
256	46562
512	9421
768	44

Another way to view this though is to assume that all bins cover 256 arguments but that there are cases where the value computed for arguments in adjacent bins are identical. See Figure 3 for the behavior for `rsqrt` on AMD for $x \approx 1.997$.

Second, only the low-order 7 bits of each result are zero, implying an accuracy of approximately 17 bits. See Figure 4.

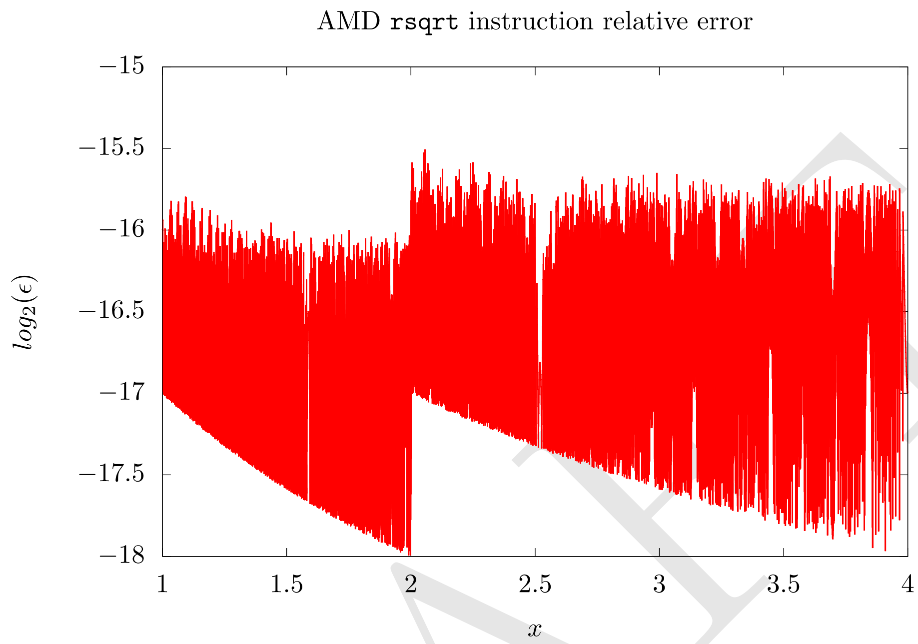

Figure 4: Relative error the `rsqrt` instruction on an AMD platform

I cannot explain why AMD chose this particular implementation for the `rsqrt` instruction.

DRAFT

Code Generated for Newton-Raphson Iteration

`gcc 5.2.0` with the options `-Ofast -mavx` generates the following code for `1.0f/std::sqrt(x)`:

```
vrsqrtss  %xmm0, %xmm1, %xmm1
vmulss    %xmm0, %xmm1, %xmm0
vmulss    %xmm1, %xmm0, %xmm0
vaddss    LC0(%rip), %xmm0, %xmm0
vmulss    LC1(%rip), %xmm1, %xmm1
vmulss    %xmm1, %xmm0, %xmm0
```

This code following the `vrsqrtss` instruction implements the Newton-Raphson iteration to improve the accuracy of the result returned by the `rsqrt` instruction:

$$
\begin{aligned}
a &\leftarrow \texttt{rsqrt_ss}(\texttt{x}) \\
b &\leftarrow a * x \\
c &\leftarrow a * b = (a * (a * x)) \\
d &\leftarrow -3.0 + c = -3.0 + (a * (a * x)) \\
e &\leftarrow -0.5 * a \\
f &\leftarrow e * d = (-0.5 * a) * (-3.0 + (a * (a * x))) \\
\mathit{result} &\leftarrow 0.5 * a * (3.0 - (a * (a * x)))
\end{aligned}
$$

Equivalent code is generated by `icc 16.0` with the same options.

Note that the algebraically equivalent computation

$$\mathit{result} \leftarrow a + 0.5 * a * (1 - x * (a * a))$$

should produce a more accurate result on Intel CPUs:

- the product $a * a$ is exact since the low-order 13 bits of the characteristic of a are zero
- the round-off error in the final addition is smaller because, since $x * (a * a)$ is close to 1.0, there is a significant alignment shift between a and $0.5 * a * (1 - x * (a * a))$

DRAFT

However, this comes at the cost of one additional operation.

If source code for the improved Newton-Raphson computation is compiled with `gcc -O2 -mavx -mfma`, the following instructions are generated:

```
vrsqrtps       %ymm0, %ymm2
vmulps         %ymm2, %ymm2, %ymm1
vfnmadd213ps   .LC0(%rip), %ymm1, %ymm0
vmulps         %ymm2, %ymm0, %ymm0
vfmadd132ps    .LC1(%rip), %ymm2, %ymm0
```

Not only are fewer instructions required but the use of FMA instructions produces more accurate results because there are fewer roundings.

A comparison of the relative errors of these three methods of calculating the Newton-Raphson iteration is shown in Figure 5. The improved calculation results in a decrease of approximately 0.5 *ulp* in the relative error, and the use of FMA provides a slight additional decrease in the relative error.

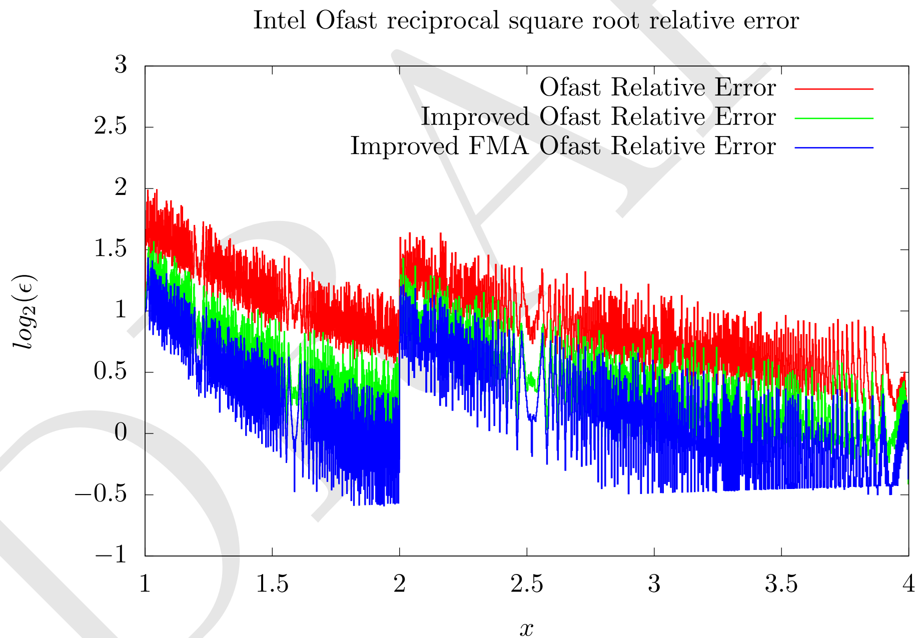

Figure 5: Relative error of Ofast reciprocal square root on an Intel platform

DRAFT

Differences Between gcc and icc

Consider these lines in the original sample program:

```
float fn = n - 1.f;
float k = 0.1f + fn;
```

When using the option `-Ofast`, `gcc` does more aggressive constant-folding than does `icc`: `gcc` computes `k = n - 0.9f` whereas `icc` computes `k = 0.1f + (n - 1.0f)`. This leads to a difference of 3 *ulps* in the argument passed to `std::sqrt()` which accounts for the 1 *ulp* difference in the results. This is unusual in that `gcc` is usually much more conservative than `icc` is its optionizations which may affect floating-point results.

DRAFT

Possible next steps

Although the latency and throughput of the `rsqrt` instruction relative to the `sqrt` and `div` instructions may change with each new processor, it may be worthwhile for `gcc` to improve the code generated for the Newton-Raphson iteration using with the `rsqrt` instruction, especially when FMA instructions are available. This would result in not only increased accuracy but potentially better performance.

For `clang`, it woud be worthwhile to actually implement any of these optimizations for x86. On Mac OS X, as of `Xcode 7.1`, `clang -Ofast` does not make use of the `rsqrt` instruction.

DRAFT

Behavior of the `rcp` instruction

Given this behavior of the `rsqrt` instruction, an investigation of the `rcp` instruction seemed a reasonable follow-on exercise because it too is not subject to any standardization. Thus, it may well behave differently on AMD and Intel platforms. The investigation followed a similar path as that of the `rsqrt` instruction.

Background

Consider this program:

```
#include <cmath>
#include <cstdio>
int main(int n, char* argv[]) {
    float fn = n - 1.f;
    float k = 0.5f + fn;
    float q = 1.f/k;
    printf("%6.6a %6.6a\n",k,q);
    return 0;
}
```

When run with no arguments, it computes the reciprocal of /0.5 in using IEEE 754 Binary32 arithmetic. When compiled with `gcc 5.2.0` and the `-O2 -mavx` options, the generated code uses a `div_ss` instruction to compute the result. However, if the options `-Ofast -mavx -mrecip` are used instead, the generated code computes the result using an `rcp_ss` instruction followed by a single Newton-Raphson interation. In this latter case, the output of the *exact same* executable produces different output results depending on whether it is executed on an Intel CPU or an AMD CPU:

- Intel: `q = 0x1.fffffep+0`
- AMD: `q = 0x1.000000p+1`

DRAFT

Reciprocal

Let x be a positive non-zero normal IEEE 754 Binary32 number with $x = 2^e m$ where $1 \le m < 2$. Let $x' = 2^{-e}x = m$. Then $1 \le x' < 2$ and $1/x = 2^{-e}/x'$.

To compute $1/x'$, we take an initial estimate $y_0 \approx 1/x'$ and use a Newton-Raphson interation to improve that estimate. If we use Newton's method to find a root of the function $f(y) = 1/y - x'$, we obtain the recurrence

$$y_{n+1} \leftarrow y_n(2 - x'y_n)$$

It can be shown that each such interation doubles the accuracy of y_n.

The basic use of the `rcp` instruction then is to provide the initial estimate y_0. The code generated by the compiler consists of the `rcp` instruction followed by instructions which implement a single Newton-Raphson interation to refine that result.

Intel

A study was made of the results computed by the `rcp` instruction for positive non-zero normal Binary32 arguments. First, it was verified that the results for arguments in the range $[1, 2)$, when suitably scaled by a power of 2, matched the results returned by `rcp` for every possible positive normal non-zero Binary32 argument. This justifies the assumption that only the behavior of `rcp` in $[1, 2)$ needs to be studied.

DRAFT

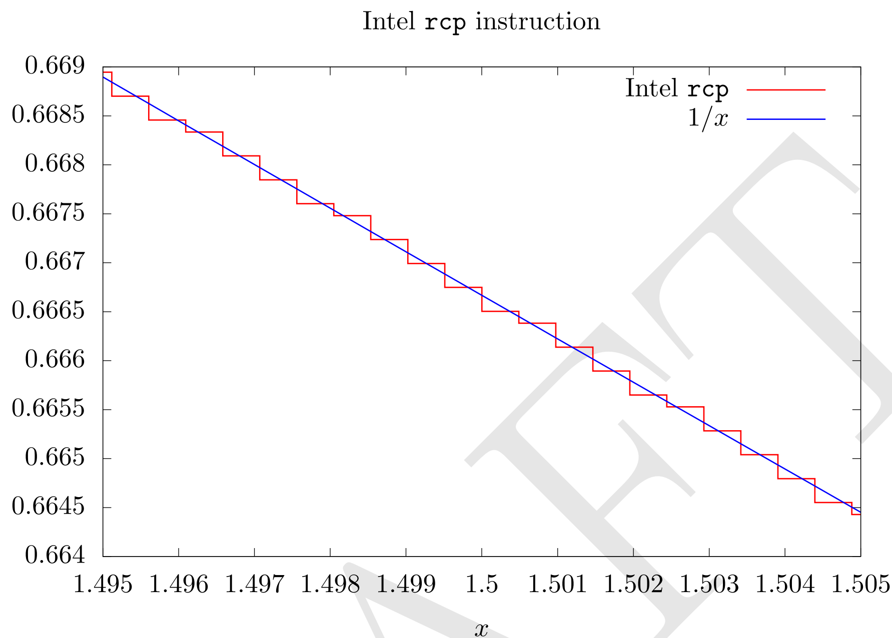

Figure 6: Value of the `rcp` instruction on an Intel platform

Just as with the **`rsqrt`** instruction, examination of the results shows that they depend only on the sign, exponent and high-order 11 bits of the characteristic of the argument. The low-order 13 bits of the characteristic of an argument are ignored. Thus the result of the instruction consists of 2048 bins in each binade, and each bin corresponds to the result for 4096 continguous arguments. Figure 6 shows this behavior.

In addition, the low-order 11 bits of the characteristic of all results returned by the instuction are zero giving approximately 13 bits of accuracy as shown in Figure 7. For reference, the maximum relative error of 1.5×2^{-12} documented by Intel is shown as well.

The relative error plots were created in a similar way as was done for the study of the **`rsqrt`** instruction except that there are 2048 bins each covering 4096 contiguous arguments.

DRAFT

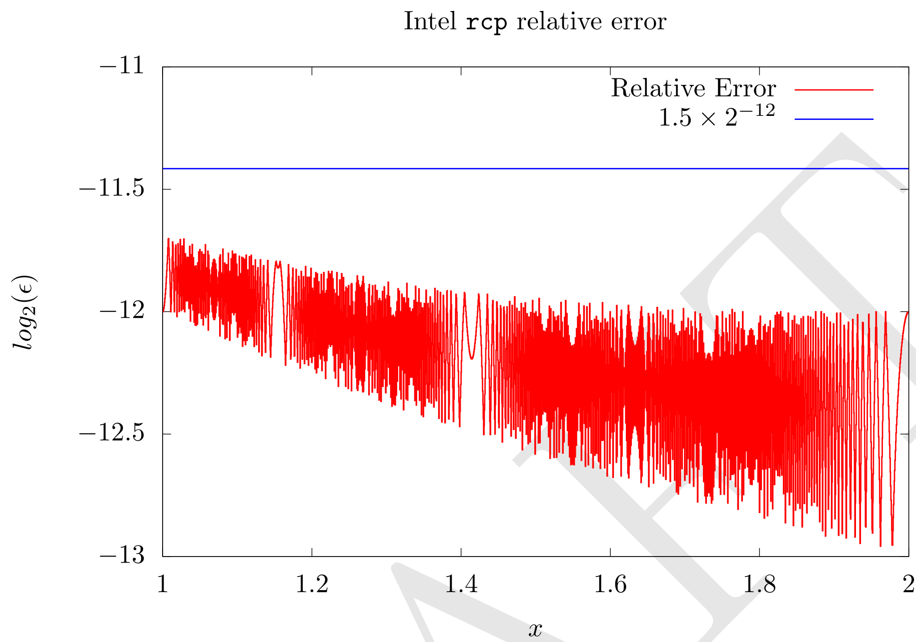

Figure 7: Relative error the `rcp` instruction on an Intel platform

AMD

Although the characteristics of the results returned by the `rcp` instruction on an AMD processor are somewhat similar to that found on Intel CPUs, they differ in two very important ways.

DRAFT

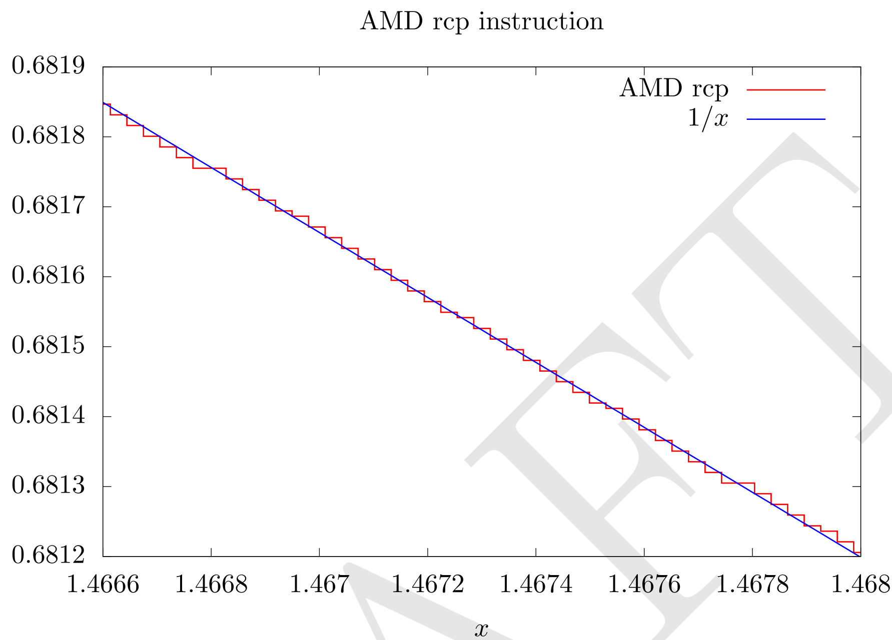

Figure 8: Value of the `rcp` instruction on an AMD platform

First, the steps (viz., bins) are not all of the same width; although most are 256 arguments wide.

Number of args in bin	Frequency
256	32636
512	66

See Figure 8 for the behavior for `rcp` on AMD for $x \approx 1.467$.

Second, only the low-order 7 bits of each result are zero, implying an accuracy of approximately 17 bits. See Figure 9.

DRAFT

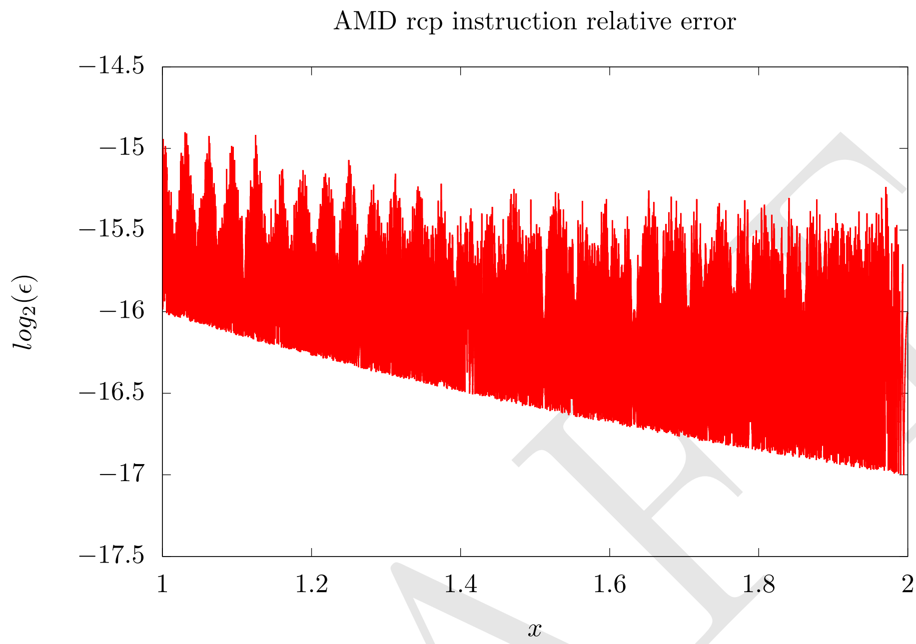

Figure 9: Relative error the `rcp` instruction on an AMD platform

Again, I cannot explain why AMD has chosen this particular implementation.

DRAFT

Code Generated for Newton-Raphson Iteration Following `rcp`

`gcc 5.2.0` with the options `-Ofast -mavx -mrecip` generates code similar to the following for `1.0f/x`:

```
vrcpps   %ymm0, %ymm1
vmulps   %ymm0, %ymm1, %ymm0
vmulps   %ymm0, %ymm1, %ymm0
vaddps   %ymm1, %ymm1, %ymm1
vsubps   %ymm0, %ymm1, %ymm1
```

The code following the `vrcpss` instruction implements a Newton-Raphson iteration to improve the accuracy of the result returned by the `rcp` instruction:

$$\begin{aligned} a &\leftarrow \texttt{rcp_ss}(\texttt{x}) \\ b &\leftarrow a * x \\ c &\leftarrow a * b = a * (a * x) \\ d &\leftarrow a + a = 2 * a \\ result &\leftarrow d - c = (2 * a) - (a * (a * x)) \end{aligned}$$

The Newton-Raphson interation can also be written as

$$y_{n+1} \leftarrow y_n + y_n(1 - x' y_n)$$

and implemented by the instructions

```
vrcpps    %ymm0, %ymm2
vmulps    %ymm2, %ymm0, %ymm0
vmovaps   .LC0(%rip), %ymm1
vsubps    %ymm0, %ymm1, %ymm0
vmulps    %ymm2, %ymm0, %ymm0
vaddps    %ymm2, %ymm0, %ymm0
```


DRAFT

Although this "improved" computation does not seem to produce a more accurate result on Intel CPUs, it does have noticeably better performance. Also, this form of the calculation can be implemented very efficiently with FMA instructions. If the source code is compiled with `gcc -O2 -mavx -mfma`, the following instructions are generated:

```
vrcpps          %ymm0, %ymm1
vfnmadd213ps    .LC0(%rip), %ymm1, %ymm0
vfmadd132ps     %ymm0, %ymm1, %ymm1
```

Not only are fewer instructions required but the use of FMA instructions produces more accurate results because there are fewer roundings.

A comparison of the relative errors of the compiler-generated method of calculating the Newton-Raphson iteration with the improved FMA version is shown in Figure 10. The improved FMA calculation results in a decrease of approximately 0.5 *ulp* in the relative error. See Table A.1 for a comparison of the timings of these various methods.

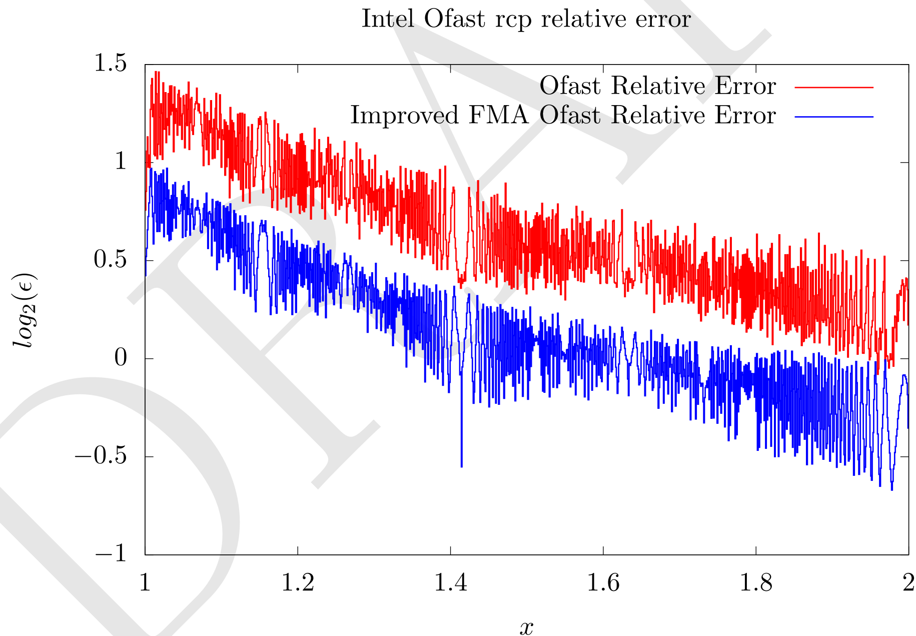

Figure 10: Relative error of Ofast reciprocal calculation on an Intel platform

DRAFT

Possible next steps

Although the latency and throughput of the `rcp` instruction relative to the `div` instruction may change with each new processor, it may be worthwhile for `gcc` to improve the code generated for the Newton-Raphson iteration using with the `rcp` instruction, especially when FMA instructions are available. This would result in not only increased performance but, in the case where FMA instructions are availble, improved accuracy.

DRAFT

Appendix A

Performance Information

Only relatively crude performance information has been gathered by timing various artifical single-threaded test programs using the `chrono` library. (Use of this library requires that the programs be compiled with the option `-std=c++11`.) Corrections for loop and call overheads were not made.

Results were measured for the following function variations:

- scalar Ofast
- packed Ofast with compiler-generated Newton-Raphson iteration code
- packed Ofast with improved Newton-Raphson interation code
- packed Ofast with improved Newton-Raphson interation code utilizing FMA instructions

The tests were run on `olhswep03.cern.ch`, a system with 4 Intel(R) Xeon(R) CPU E5-2698 V3 @ 2.30GHz processors (16 Haswell cores each, HT enabled), and 64 GB of memory.

The value reported is "mega-results per second": the number of results calculated per second scaled by 10^6.

Function	`rsqrt`	`rcp`
Scalar	303	257
Packed	1314	1308
Improved	2923	3047
FMA	3049	3048

Table A.1

I do not understand why the "packed" versions are so much slower than the "improved" versions. An explanation for this difference needs to be found.

DRAFT