The CORE-MATH cosh is correctly rounded

Paul Zimmermann
April 16, 2026

***Abstract*—We prove that the CORE-MATH binary64 hyperbolic cosine function is correctly rounded.**

***Index Terms*—IEEE 754, binary64 format, correct rounding.**

We consider the CORE-MATH `cosh.c` file from commit `911d9b6`. Since `cosh` is an even function, and the approximations used are even too, it suffices to prove the code is correctly rounded for $x \geq 0$.

I. Branch $0 \leq x < 2^{-26}$

For $0 \leq x < 2^{-26}$, the code simply returns $\mathrm{fma}(x, 2^{-55}, 1)$.

Lemma 1: For $0 \leq x < 2^{-26}$, $\mathrm{fma}(x, 2^{-55}, 1)$ yields the correct rounding of $\cosh x$, whatever the rounding mode.

Proof: First, for $x = \pm 0$, it returns 1 which is the correct rounding. Assume now $0 < x < 2^{-26}$. For $0 < x \leq 1$, we have $1 < \cosh x < f(x) := 1 + x^2/2 + x^4/2$. Since $x < 2^{-26}$, it follows $x \leq x_0 := 2^{-26} - 2^{-79}$, and $f(x_0) < 1 + 2^{-53} = 1 + \frac{1}{2}\mathrm{ulp}(1)$. Hence $1 < \cosh x < 1 + \frac{1}{2}\mathrm{ulp}(1)$ for $0 < x < 2^{-26}$, thus $\cosh x$ rounds to 1 to nearest, toward zero or toward $-\infty$, and to $1 + \mathrm{ulp}(1)$ toward $+\infty$. The same holds for $\mathrm{fma}(x, 2^{-55}, 1)$, which is thus the correct rounding. ■

Note that the bound $x < 2^{-26}$ is optimal, since for $x = 2^{-26}$, $\mathrm{fma}(x, 2^{-55}, 1)$ does not yield the correct rounding for rounding to nearest: it yields 1 instead of $1 + \mathrm{ulp}(1)$. We could replace the magic constant 2^{-55} by any constant $\leq 2^{-27}$. However, there is no binary64 constant m such that $\mathrm{fma}(x, m, 1)$ works in a larger range. Indeed, let $x_1 = \mathrm{nextup}(x_0) = 2^{-26}$. For $m = 2^{-27}$, $\mathrm{fma}(x_1, m, 1)$ rounds to 1 to nearest, thus m is too small. For $m = \mathrm{nextup}(2^{-27})$, $\mathrm{fma}(x_0, m, 1)$ rounds to $1 + \mathrm{ulp}(1)$ to nearest, thus m is too large.

II. Branch $2^{-26} \leq x < 2^{-3}$

In this branch, an even degree-10 polynomial $q(x) = 1 + c_0 x^2 + \cdots + c_4 x^{10}$ is used, with coefficients:

$$\begin{aligned} c_0 &= \texttt{0x1p-1} \\ c_1 &= \texttt{0x1.5555555555554p-5} \\ c_2 &= \texttt{0x1.6c16c16c1d0cp-10} \\ c_3 &= \texttt{0x1.a01a0075066b4p-16} \\ c_4 &= \texttt{0x1.27faff8dcc1c8p-22} \end{aligned}$$

In the error analysis below, we will need a bound on the error $|q(x) - \cosh x|/x^2$. Such a bound is given in Table I on binades covering $[2^{-26}, 2^{-3})$.

interval	s	ϵ
$[2^{-26}, 2^{-25}]$	2^{-80}	$79 \cdot 2^{-57}$
$[2^{-25}, 2^{-24}]$	2^{-80}	$79 \cdot 2^{-57}$
$[2^{-24}, 2^{-23}]$	2^{-80}	$79 \cdot 2^{-57}$
$[2^{-23}, 2^{-22}]$	2^{-80}	$80 \cdot 2^{-57}$
$[2^{-22}, 2^{-21}]$	2^{-80}	$81 \cdot 2^{-57}$
$[2^{-21}, 2^{-20}]$	2^{-80}	$81 \cdot 2^{-57}$
$[2^{-20}, 2^{-19}]$	2^{-80}	$89 \cdot 2^{-57}$
$[2^{-19}, 2^{-18}]$	2^{-80}	$96 \cdot 2^{-57}$
$[2^{-18}, 2^{-17}]$	2^{-80}	$96 \cdot 2^{-57}$
$[2^{-17}, 2^{-16}]$	2^{-80}	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-16}, 2^{-15}]$	2^{-80}	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-15}, 2^{-14}]$	2^{-80}	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-14}, 2^{-13}]$	2^{-80}	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-13}, 2^{-12}]$	2^{-80}	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-12}, 2^{-11}]$	$3 \cdot 2^{-80}$	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-11}, 2^{-10}]$	$11 \cdot 2^{-80}$	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-10}, 2^{-9}]$	$43 \cdot 2^{-80}$	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-9}, 2^{-8}]$	$170 \cdot 2^{-80}$	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-8}, 2^{-7}]$	$658 \cdot 2^{-80}$	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-7}, 2^{-6}]$	$2347 \cdot 2^{-80}$	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-6}, 2^{-5}]$	$5591 \cdot 2^{-80}$	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-5}, 2^{-4}]$	$\mathbf{5721 \cdot 2^{-80}}$	$\mathbf{97 \cdot 2^{-57}}$
$[2^{-4}, 2^{-3}]$	$5090 \cdot 2^{-80}$	$\mathbf{97 \cdot 2^{-57}}$

TABLE I
For several ranges covering $[2^{-26}, 2^{-3})$, Sollya bound s such that $|q(x) - \cosh x| < sx^2$, and corresponding value of ϵ such that $\ell_b \leq q(x) - sx^2$ and $q(x) + sx^2 \leq u_b$.

Algorithm 1 (FastPath)

Input: x a binary64 number, $2^{-26} \leq x < 2^{-3}$

1: let $c_0, ..., c_4$ the coefficients of $q(x)$
2: $x_2 \leftarrow \circ(x \cdot x), \quad x_4 \leftarrow \circ(x_2 \cdot x_2)$
3: $p_0 \leftarrow \circ(c_0 + \circ(x_2 c_1)), \quad p_2 \leftarrow \circ(c_2 + \circ(x_2 c_3))$
4: $p_2' \leftarrow \circ(p_2 + \circ(x_4 c_4)), \quad p_0' \leftarrow \circ(p_0 + \circ(x_4 p_2'))$
5: $p \leftarrow \circ(x_2 p_0')$
6: $e = \circ(\epsilon x_2)$ where $\epsilon = 97 \cdot 2^{-57}$
7: $\ell_b = 1 + \circ(p - e), \quad u_b = 1 + \circ(p + e)$
8: if $\circ(\ell_b) = \circ(u_b)$ return $\circ(\ell_b)$ else return FAIL

The fast path for this branch is detailed in Algorithm 1. For processors with hardware FMA (fused-multiply-add), the compiler might decide to use a FMA for instructions like $c_0 + x_2 c_1$; this is called *FMA contraction*. The rounding test is performed at line 8. (In the CORE-MATH code, ℓ_b and u_b design the rounded values.) If it fails, the accurate path is called.

Theorem 1: For $2^{-26} \leq x < 2^{-3}$, if Algorithm FastPath does not return FAIL, its return value $\circ(\ell_b)$ is the correct rounding of $\cosh x$.

Proof: We use the following Gappa program, with version 1.7.0 of Gappa, where we substitute `RND` by all four rounding modes, `XMIN` and `XMAX` by interval bounds, and `BND` by the Sollya bound.

```
@rnd = float<ieee_64,RND>;
c0 = 0x1p-1;
c1 = 0x1.5555555555554p-5;
c2 = 0x1.6c16c16c1d0cp-10;
c3 = 0x1.a01a0075066b4p-16;
c4 = 0x1.27faff8dcc1c8p-22;
x2 = x*x;
x2r rnd= x*x;
x4 = x2*x2;
x4r rnd= x2r*x2r;
p0 = c0+x2*c1;
p0r rnd= c0 + x2r*c1;
p2 = c2+x2*c3;
p2r rnd= c2 + x2r*c3;
pp2 = p2 + x4*c4;
pp2r rnd= p2r + x4r*c4;
pp0 = p0 + x4*pp2;
pp0r rnd= p0r + x4r*pp2r;
p = x2*pp0;
q = 1 + p;
pr rnd= x2r*pp0r;
eps = EPS;
er rnd= eps*x2r;
lb = 1 + rnd(pr-er);
{ x in [XMIN,XMAX] -> (q-lb)/x2 >= BND }
```

For example, for the interval $[2^{-4}, 2^{-3}]$, this Gappa program proves with $\mathrm{EPS} = 97 \cdot 2^{-57}$:

$$q(x) - \ell_b \geq 5090 \cdot 2^{-80} x^3.$$

Since in this interval, the mathematical error $|q(x) - \cosh x|$ is less than $5090 \cdot 2^{-80} x^2$ (Table I), this proves $\ell_b \leq \cosh x$. A similar program checks the goal $u_b - q(x) \geq 5090 \cdot 2^{-80} x^3$, which proves $\cosh x \leq u_b$. For all intervals from Table I, we have proven both $q - \ell_b \geq sx^2$ and $u_b - q \geq sx^2$ where s is the corresponding mathematical error bound computed by Sollya. It follows $\ell_b \leq \cosh x \leq u_b$ and by monotonicity of the rounding, $\circ(\ell_b) \leq \circ(\cosh x) \leq \circ(u_b)$. ∎

A. A potential bug in a previous version

In commit `d495158` of CORE-MATH, a different polynomial was used in the fast path for $2^{-26} \leq x < 2^{-3}$, with error tolerance $\epsilon = 64 \cdot 2^{-57}$ in the rounding test. Consider $x =$ `0x1.6e8f8b1809a41p-5`. For that value and rounding downward, we got a value $u_b = x + \circ(p + e)$ which was smaller than $\cosh x$, which defeats the logic of the rounding test: we should always have $\ell_b \leq \cosh x \leq u_b$, so that $\circ(\ell_b) \leq \circ(\cosh x) \leq \circ(u_b)$, and if $\circ(\ell_b) = \circ(u_b)$, we can conclude this equals $\circ(\cosh x)$. This example requires $\epsilon \geq 79 \cdot 2^{-57}$ so that $\cosh x \leq u_b$.

However, it might be that such examples that defeat the logic of the rounding test are still correctly rounded, because $\cosh x$ is far from a rounding boundary. The largest value of ϵ in Algorithm FastPath for which we found a real failure is $\epsilon = 57 \cdot 2^{-57}$, for $x =$ `0x1.0f0a7d6ea89ep-14` (rounding up, without FMA contraction).

B. Accurate path for $x_0 \leq x < 1/4$

The accurate path in this branch corresponds to the `as_cosh_zero` routine. An odd degree-16 polynomial $r(x) = 1 + c_0 x^2 + \cdots + c_7 x^{16}$ is used, with double-double coefficients for degrees 2 to 8, and double coefficients for degrees 10 to 16:

$$\begin{aligned}
c_0 &= \texttt{0x1p-1} - \texttt{0x1.c7e8db669f624p-111}\\
c_1 &= \texttt{0x1.5555555555555p-5} + \texttt{0x1.5555555556135p-59}\\
c_2 &= \texttt{0x1.6c16c16c16c17p-10} - \texttt{0x1.f49f4a6e838f2p-65}\\
c_3 &= \texttt{0x1.a01a01a01a01ap-16} + \texttt{0x1.a4ffbe15316aap-76}\\
c_4 &= \texttt{0x1.27e4fb7789f5cp-22}\\
c_5 &= \texttt{0x1.1eed8eff9089cp-29}\\
c_6 &= \texttt{0x1.939749ce13dadp-37}\\
c_7 &= \texttt{0x1.ae9891efb6691p-45}
\end{aligned}$$

The relative error is bounded by Sollya on $[2^{-26}, 2^{-3}]$:

$$|r(x)/\cosh x - 1| \leq 2^{-105.211}. \tag{1}$$

To avoid duplicating the coefficients, we denote them $c_i = c_{i,h}, c_{i,\ell}$ for double-double coefficients in Algorithm AccuratePath below. We denote roundings by $\circ(\cdot)$, where $\circ$ denotes the current rounding mode; and $\mathrm{fma}(a, b, c)$ denotes $\circ(ab + c)$. At line 2 of AccuratePath, y_2 is a double approximation of $c_4 x^2 + c_5 x^4 + c_6 x^6 + c_7 x^8$, then after the `polydd` call at line 3, $y_1 + y_2$ is a double-double approximation of $c_0 + c_1 x^2 + \cdots + c_7 x^{14}$, after line 4 it is a double-double approximation of $c_0 x^2 + \cdots + c_7 x^{16}$, and the last two lines add the constant term 1. Here fasttwosum denotes the classical FastTwoSum routine, and polydd, `muldd` are also described below.

Algorithm 2 (AccuratePath)

Input: x a binary64 number, $2^{-26} \leq x < 2^{-3}$
Output: $y_0 + y_1$ approximating $\cosh x$

1: $x_2 \leftarrow \circ(x \cdot x), \quad x_{2\ell} \leftarrow \mathrm{fma}(x, x, -x_2)$
2: $y_2 \leftarrow \circ(x_2 \cdot \circ(c_4 + \circ(x_2 \cdot \circ(c_5 + \circ(x_2 \cdot \circ(c_6 + \circ(x_2 \cdot c_7)))))))$
3: $y_1, y_2 \leftarrow \mathrm{polydd}(x_2, x_{2\ell}, 4, y_2)$
4: $y_1, y_2 \leftarrow \mathrm{muldd}(y_1, y_2, x_2, x_{2\ell})$
5: $y_0, y_1 \leftarrow \mathrm{fasttwosum}(1, y_1)$
6: $y_1 \leftarrow \circ(y_1 + y_2)$

Algorithm 3 (polydd)

Input: $x = x_h + x_\ell$ double-double, n integer, r binary64
Output: $h + \ell$ approximating $c_0 + c_1 \cdot x^2 + \cdots + (c_{n-1} + r) x^{2n-2}$

1: $h, t \leftarrow \mathrm{fasttwosum}(c_{n-1,h}, r)$
2: $\ell \leftarrow \circ(t + c_{n-1,\ell})$
3: for i from $n - 2$ downto 0
4: $\quad h, \ell \leftarrow \mathrm{muldd}(x_h, x_\ell, h, \ell)$
5: $\quad h, e \leftarrow \mathrm{fasttwosum}(c_{i,h}, h)$
6: $\quad \ell \leftarrow \circ(\circ(\ell + c_{i,\ell}) + e)$

Algorithm 4 (fasttwosum)

Input: a, b binary64 numbers
Output: h, ℓ such that $h + \ell$ approximates $a + b$
1: $h \leftarrow \circ(a + b)$
2: $t \leftarrow \circ(h - a)$
3: $\ell \leftarrow \circ(b - t)$

Algorithm 5 (muldd)

Input: x_h, x_ℓ, c_h, c_ℓ binary64 numbers
Output: h, ℓ approximating $(x_h + x_\ell)(c_h + c_\ell)$
1: $h \leftarrow \circ(x_h c_h)$
2: $\ell_1 \leftarrow \mathrm{fma}(x_h, c_h, -h)$
3: $\ell_2 \leftarrow \circ(x_h c_\ell)$
4: $\ell_3 \leftarrow \circ(x_\ell c_h)$
5: $\ell \leftarrow \circ(\circ(\ell_1 + \ell_2) + \ell_3)$

We wrote a Gappa program to analyze the rounding error of Algorithm AccuratePath. This program was generated automatically from a SageMath program. For example, we have a `gen_muldd` routine that generates the Gappa instructions for `muldd`, taking as argument the input and output variables, whose names are generated automatically to avoid name conflicts. The SageMath program also automatically generates Gappa hints, for example if `c=rnd(a*b)` is followed by `r = -(c-a*b)`, it generates the hint so that r is recognized by Gappa as the rounding error on c.

This Gappa program proves that the following holds, for $2^{-26} \le x < 2^{-3}$, and all rounding modes:

$$|y_0 + y_1 - r(x)|/r(x) \le 2^{-102.145}. \tag{2}$$

Since $|r(x)| < 1.0001|\cosh x|$ by Eq. (1), and $2^{-105.211} + 1.0001 \cdot 2^{-102.145} < 2^{-101.982}$, it follows:

$$|y_0 + y_1 - \cosh x| < 2^{-101.982} \cosh x. \tag{3}$$

Theorem 2: For any binary64 number x such that $\cosh x$ has less than 47 identical bits after the round bit, if $y_0 + y_1 + y_2$ is the double-word approximation of $\cosh x$ computed by Algorithm AccuratePath, then $\circ(y_0 + y_1)$ is the correct rounding of $\cosh x$.

Proof: Let $y = \cosh x$. If y has less than 47 identical bits after the round bit, then y is at distance at least $2^{-48}\mathrm{ulp}(y)$ from a rounding boundary. Thus for any rounding boundary z, $|y - z| \ge 2^{-48}\mathrm{ulp}(y)$. Since $\mathrm{ulp}(y) > 2^{-53}|y|$ (we are in the normal range), this yields $|y - z| \ge 2^{-101}|y|$. Thus Eq. (3) shows that $|y_0 + y_1 - \cosh x| < |\cosh x - z|$, i.e., $y_0 + y_1$ is closer from $\cosh x$ than any rounding boundary. Thus $\circ(y_0 + y_1) = \circ(\cosh x)$. ∎

Since CORE-MATH contains all hard-to-round cases of cosh with at least 43 identical bits after the round bit, it suffices to check correct rounding for these inputs. The CORE-MATH code checks whether y_1 has its low 52 significant bits zero (i.e., it is $\pm 2^k$), and then adjust the result according to the sign of the second order term y_2. This happens for 172 inputs, and it is enough to guarantee correct rounding for those.

III. Branch $x \ge 1/4$

This branch is proven correct by exhaustive testing, both with and without the use of FMA (fused-multiply-add). Indeed, in C code like `d = a*b+c`, the compiler might decide to emit a FMA if it is supported in hardware, or not to emit a FMA. Thus we have to check the result is correctly rounded in both cases.

Since $\cosh x$ overflows in binary64 for $|x| > 711$, we only have to check less than 12 binades, which is doable with academic resources, using the algorithm described in Section II.C from [1].

Acknowledgements. We thank Vincenzo Innocente who contributed to the exhaustive testing for $x \ge 1/8$ on a AMD EPYC 9754 128-Core Processor at CERN. The other exhaustive tests were performed on the EXPLOR center hosted by the University of Lorraine, and on the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see https://www.grid5000.fr/). Some failures were found by Adhemerval Zanella on an earlier version of the code, and the exhaustive tests revealed some further failures.

References

[1] Zimmermann, P. The GNU libc atanh is correctly rounded. In *ARITH 2026 - 33rd IEEE International Symposium on Computer Arithmetic* (Fulda, Germany, June 2026).

Appendix

A. Gappa program for the accurate path

The Gappa program we used is the following. In the goal, `y2_2r // c3h in [-1,1]` ensures the FastTwoSum condition is satisfied for `fasttwosum(c3h,y2_2r)`, and `@FIX(x2r*h,-1074)` ensures the rounding error in `x2r*h` is exactly representable, which is necessary for the error bound in `muldd(x2r,x2l,h,l)`.

```
@rnd = float<ieee_64,RND>;
x = rnd(x_); # x is a binary64 number
x2 = x*x;
x2r rnd= x*x;
x2l = -(x2r-x*x);
c0h = 0x1p-1;
c0l = -0x1.c7e8db669f624p-111;
c0 = c0h+c0l;
c1h = 0x1.5555555555555p-5;
c1l = 0x1.5555555556135p-59;
c1 = c1h+c1l;
c2h = 0x1.6c16c16c16c17p-10;
c2l = -0x1.f49f4a6e838f2p-65;
c2 = c2h+c2l;
c3h = 0x1.a01a01a01a01ap-16;
c3l = 0x1.a4ffbe15316aap-76;
c3 = c3h+c3l;
c4 = 0x1.27e4fb7789f5cp-22;
c5 = 0x1.1eed8eff9089cp-29;
c6 = 0x1.939749ce13dadp-37;
c7 = 0x1.ae9891efb6691p-45;
s6 = x2*c7;
s6r rnd= x2r*c7;
u6 = c6+s6;
u6r rnd= c6+s6r;
```

```
s5 = x2*u6;
s5r rnd= x2r*u6r;
u5 = c5+s5;
u5r rnd= c5+s5r;
s4 = x2*u5;
s4r rnd= x2r*u5r;
u4 = c4+s4;
u4r rnd= c4+s4r;
y2_2 = x2*u4;
y2_2r rnd= x2r*u4r;
C3 = c3+y2_2;
X2R = x2r+x2l;
# h,t = fasttwosum (c3h,y2_2r)
h rnd= c3h+y2_2r;
t = rnd(-(h-(c3h+y2_2r)));
l rnd= t+c3l;
C3r = h+l; # h+l approximates C3
# hh,ll = muldd (x2r,x2l,h,l)
hh rnd= x2r*h;
l1 = -(hh-x2r*h);
l2 rnd= x2r*l;
l3 rnd= x2l*h;
t1 rnd= l1+l2;
ll rnd= t1+l3;
D2 = x2*C3; # hh+ll approximates D2
# h1,e = fasttwosum (c2h,hh)
h1 rnd= c2h+hh;
e = rnd(-(h1-(c2h+hh)));
t2 rnd= ll+c2l;
l4 rnd= t2+e;
C2 = c2+D2;
C2r = h1+l4; # h1+l4 approximates C2
# hh1,ll1 = muldd (x2r,x2l,h1,l4)
hh1 rnd= x2r*h1;
l11 = -(hh1-x2r*h1);
l21 rnd= x2r*l4;
l31 rnd= x2l*h1;
t3 rnd= l11+l21;
ll1 rnd= t3+l31;
D1 = x2*C2; # hh1+ll1 approximates D1
# h2,e1 = fasttwosum (c1h,hh1)
h2 rnd= c1h+hh1;
e1 = rnd(-(h2-(c1h+hh1)));
t4 rnd= ll1+c1l;
l5 rnd= t4+e1;
C1 = c1+D1;
C1r = h2+l5; # h2+l5 approximates C1
# hh2,ll2 = muldd (x2r,x2l,h2,l5)
hh2 rnd= x2r*h2;
l12 = -(hh2-x2r*h2);
l22 rnd= x2r*l5;
l32 rnd= x2l*h2;
t5 rnd= l12+l22;
ll2 rnd= t5+l32;
D0 = x2*C1; # hh2+ll2 approximates D0
# y1_3,e2 = fasttwosum (c0h,hh2)
y1_3 rnd= c0h+hh2;
e2 = rnd(-(y1_3-(c0h+hh2)));
t6 rnd= ll2+c0l;
y2_3 rnd= t6+e2;
C0 = c0+D0;
C0r = y1_3+y2_3; # y1_3+y2_3 approximates C0
# y1_4,y2_4 = muldd (y1_3,y2_3,x2r,x2l)
y1_4 rnd= y1_3*x2r;
l13 = -(y1_4-y1_3*x2r);
l23 rnd= y1_3*x2l;
l33 rnd= y2_3*x2r;
t7 rnd= l13+l23;
y2_4 rnd= t7+l33;
Y4 = C0*x2;
Y4r = y1_4+y2_4;
# y0,y1_5 = fasttwosum (1,y1_4)
y0 rnd= 1+y1_4;
y1_5 = rnd(-(y0-(1+y1_4)));
# y1 = y1_5+y2_4
y1 rnd= y1_5+y2_4;
Y = 1+Y4;
Yr = y0+y1;

# goal
{x in [XMIN,XMAX] -> y2_2r // c3h in [-1,1]
 /\ hh // c2h in [-1,1] /\ hh1 // c1h in [-1,1]
 /\ hh2 // c0h in [-1,1] /\ y1_4 in [-1,1]
 /\ @FIX(x2r*h,-1074) /\ @FIX(x2r*h1,-1074)
 /\ @FIX(x2r*h2,-1074) /\ @FIX(y1_3*x2r,-1074)
 /\ |Yr - Y|/Y in ? }

# automatic hints
(x2r+x2l) - (x*x) -> x2l - (-(x2r-x*x));
(h+t) - (c3h+y2_2r) -> t - (-(h-(c3h+y2_2r)));
C3r-C3 -> ((h+t)-(c3h+y2_2r)) + (l-(t+c3l))
  + (y2_2r-y2_2);
(hh+l1) - (x2r*h) -> l1 - (-(hh-x2r*h));
(hh+ll) - (x2r+x2l)*(h+l) -> ((hh+l1)-(x2r*h))
  + (l2 - (x2r*l)) + (l3 - (x2l*h)) + (t1 - (l1+l2))
  + (ll - (t1+l3)) - x2l*l;
(h1+e) - (c2h+hh) -> e - (-(h1-(c2h+hh)));
C2r-C2 -> ((h1+e)-(c2h+hh)) + (t2-(ll+c2l))
  + (l4-(t2+e)) + ((hh+ll)-D2);
(hh1+l11) - (x2r*h1) -> l11 - (-(hh1-x2r*h1));
(hh1+ll1) - (x2r+x2l)*(h1+l4) -> ((hh1+l11)-(x2r*h1))
  + (l21 - (x2r*l4)) + (l31 - (x2l*h1))
  + (t3 - (l11+l21)) + (ll1 - (t3+l31)) - x2l*l4;
(h2+e1) - (c1h+hh1) -> e1 - (-(h2-(c1h+hh1)));
C1r-C1 -> ((h2+e1)-(c1h+hh1)) + (t4-(ll1+c1l))
  + (l5-(t4+e1)) + ((hh1+ll1)-D1);
(hh2+l12) - (x2r*h2) -> l12 - (-(hh2-x2r*h2));
(hh2+ll2) - (x2r+x2l)*(h2+l5) -> ((hh2+l12)-(x2r*h2))
  + (l22 - (x2r*l5)) + (l32 - (x2l*h2))
  + (t5 - (l12+l22)) + (ll2 - (t5+l32)) - x2l*l5;
(y1_3+e2) - (c0h+hh2) -> e2 - (-(y1_3-(c0h+hh2)));
C0r-C0 -> ((y1_3+e2)-(c0h+hh2)) + (t6-(ll2+c0l))
  + (y2_3-(t6+e2)) + ((hh2+l12)-D0);
(y1_4+l13) - (y1_3*x2r) -> l13 - (-(y1_4-y1_3*x2r));
(y1_4+y2_4) - (y1_3+y2_3)*(x2r+x2l) ->
  ((y1_4+l13)-(y1_3*x2r)) + (l23 - (y1_3*x2l))
  + (l33 - (y2_3*x2r)) + (t7 - (l13+l23))
  + (y2_4 - (t7+l33)) - y2_3*x2l;
(y0+y1_5) - (1+y1_4) -> y1_5 - (-(y0-(1+y1_4)));
(y1+y2) - (y1_5+y2_4) -> y2 - (-(y1-(y1_5+y2_4)));
Yr-Y -> ((y0+y1_5)-(1+y1_4)) + (y1 - (y1_5+y2_4))
  + (Y4r - Y4);
$ x in (1b-4);
```