

This PDF represents the changes performed while forking mathematical libraries from **uniswap** library to make it compatible with **Fountain** project's `Oracle`` contract uniswap npm paths

- "@uniswap/lib/contracts/libraries/FullMath.sol"
- "@uniswap/lib/contracts/libraries/BitMath.sol"
- "@uniswap/lib/contracts/libraries/FixedPoint.sol"

Project's github repo

<https://github.com/puls369ar/fountain-solc-override/Oracle>

Changes Description

In Solidity, `type(uint224).max` returns the **maximum value** that a `uint224` (an unsigned 224-bit integer) can hold. The same we've got in older version inside `FullMath``, `BitMath`` and `FixedPoint`` libraries, by using `uint224(-1)`

Also arithmetic operations changed with binary ones to prevent `uint256` issues

When calculating `pow2``

below find code's **diff** between forked and original versions



Comparing sensitive data, confidential files or internal emails?

Most legal and privacy policies prohibit uploading sensitive data online. Diffchecker Desktop ensures your confidential information never leaves your computer. Work offline and compare documents securely.

[Download Diffchecker Desktop](#)
[Learn more](#)

Untitled diff

– 8 removals

86 lines

```

1 // SPDX-License-Identifier: GPL-3.0-or-later
2 pragma solidity >=0.5.0;
3
4 library BitMath {
5     // Returns the 0 indexed position of
6     // the most significant bit of the input x
7     // s.t. x >= 2**msb and x < 2**
8     // (msb+1)
9     function mostSignificantBit(uint256
10     x) internal pure returns (uint8 r) {
11         require(x > 0,
12         'BitMath::mostSignificantBit: zero');
13
14         if (x >=
15         0x100000000000000000000000000000000) {
16             x >>= 128;
17             r += 128;
18         }
19         if (x >= 0x100000000000000000000000000000000) {
20             x >>= 64;
21             r += 64;
22         }
23         if (x >= 0x100000000000000000000000000000000) {
24             x >>= 32;
25             r += 32;
26         }
27         if (x >= 0x100000000000000000000000000000000) {
28             x >>= 16;
29             r += 16;
30         }
31         if (x >= 0x100000000000000000000000000000000) {
32             x >>= 8;
33             r += 8;
34         }
35         if (x >= 0x100000000000000000000000000000000) {
36             x >>= 4;
37             r += 4;
38         }
39         if (x >= 0x100000000000000000000000000000000) {
40             x >>= 2;
41             r += 2;
42         }
43         if (x >= 0x100000000000000000000000000000000) {
44             x >>= 1;
45             r += 1;
46         }
47     }
48 }
```

+ 8 additions

86 lines

```

1 // SPDX-License-Identifier: GPL-3.0-or-later
2 pragma solidity >=0.8.20;
3
4 library BitMath {
5     // returns the 0 indexed position of
6     // the most significant bit of the input x
7     // s.t. x >= 2**msb and x < 2**
8     // (msb+1)
9     function mostSignificantBit(uint256
10     x) internal pure returns (uint8 r) {
11         require(x > 0,
12         'BitMath::mostSignificantBit: zero');
13
14         if (x >=
15         0x100000000000000000000000000000000) {
16             x >>= 128;
17             r += 128;
18         }
19         if (x >= 0x100000000000000000000000000000000) {
20             x >>= 64;
21             r += 64;
22         }
23         if (x >= 0x100000000000000000000000000000000) {
24             x >>= 32;
25             r += 32;
26         }
27         if (x >= 0x100000000000000000000000000000000) {
28             x >>= 16;
29             r += 16;
30         }
31         if (x >= 0x100000000000000000000000000000000) {
32             x >>= 8;
33             r += 8;
34         }
35         if (x >= 0x100000000000000000000000000000000) {
36             x >>= 4;
37             r += 4;
38         }
39         if (x >= 0x100000000000000000000000000000000) {
40             x >>= 2;
41             r += 2;
42         }
43         if (x >= 0x100000000000000000000000000000000) {
44             x >>= 1;
45             r += 1;
46         }
47     }
48 }
```

```

24         r += 16;
25     }
26     if (x >= 0x100) {
27         x >>= 8;
28         r += 8;
29     }
30     if (x >= 0x10) {
31         x >>= 4;
32         r += 4;
33     }
34     if (x >= 0x4) {
35         x >>= 2;
36         r += 2;
37     }
38     if (x >= 0x2) r += 1;
39 }
40
41 // Returns the 0 indexed position of
the least significant bit of the input x
42 // s.t. (x & 2**lsb) != 0 and (x &
(2**(lsb) - 1)) == 0)
43 // i.e. the bit at the index is set
and the mask of all lower bits is 0
44 function leastSignificantBit(uint256
x) internal pure returns (uint8 r) {
45     require(x > 0,
'BitMath::leastSignificantBit: zero');
46
47     r = 255;
48     if (x & type(uint128).max > 0) {
49         r -= 128;
50     } else {
51         x >>= 128;
52     }
53     if (x & type(uint64).max > 0) {
54         r -= 64;
55     } else {
56         x >>= 64;
57     }
58     if (x & type(uint32).max > 0) {
59         r -= 32;
60     } else {
61         x >>= 32;
62     }
63     if (x & type(uint16).max > 0) {

```

```

24         r += 16;
25     }
26     if (x >= 0x100) {
27         x >>= 8;
28         r += 8;
29     }
30     if (x >= 0x10) {
31         x >>= 4;
32         r += 4;
33     }
34     if (x >= 0x4) {
35         x >>= 2;
36         r += 2;
37     }
38     if (x >= 0x2) r += 1;
39 }
40
41 // returns the 0 indexed position of
the least significant bit of the input x
42 // s.t. (x & 2**lsb) != 0 and (x &
(2**(lsb) - 1)) == 0)
43 // i.e. the bit at the index is set
and the mask of all lower bits is 0
44 function leastSignificantBit(uint256
x) internal pure returns (uint8 r) {
45     require(x > 0,
'BitMath::leastSignificantBit: zero');
46
47     r = 255;
48     if (x & uint128(-1) > 0) {
49         r -= 128;
50     } else {
51         x >>= 128;
52     }
53     if (x & uint64(-1) > 0) {
54         r -= 64;
55     } else {
56         x >>= 64;
57     }
58     if (x & uint32(-1) > 0) {
59         r -= 32;
60     } else {
61         x >>= 32;
62     }
63     if (x & uint16(-1) > 0) {

```

```
64         r -= 16;
65     } else {
66         x >>= 16;
67     }
68     if (x & type(uint8).max > 0) {
69         r -= 8;
70     } else {
71         x >>= 8;
72     }
73     if (x & 0xf > 0) {
74         r -= 4;
75     } else {
76         x >>= 4;
77     }
78     if (x & 0x3 > 0) {
79         r -= 2;
80     } else {
81         x >>= 2;
82     }
83     if (x & 0x1 > 0) r -= 1;
84 }
85 }
86
```

```
64         r -= 16;
65     } else {
66         x >>= 16;
67     }
68     if (x & uint8(-1) > 0) {
69         r -= 8;
70     } else {
71         x >>= 8;
72     }
73     if (x & 0xf > 0) {
74         r -= 4;
75     } else {
76         x >>= 4;
77     }
78     if (x & 0x3 > 0) {
79         r -= 2;
80     } else {
81         x >>= 2;
82     }
83     if (x & 0x1 > 0) r -= 1;
84 }
85 }
86
```



Comparing sensitive data, confidential files or internal emails?

Most legal and privacy policies prohibit uploading sensitive data online. Diffchecker Desktop ensures your confidential information never leaves your computer. Work offline and compare documents securely.

[Download Diffchecker Desktop](#)[Learn more](#)

Untitled diff

- 41 removals

147 lines

```
1 // SPDX-License-Identifier: GPL-3.0-or-later
2 pragma solidity >=0.8.20;
3
4 import
  '@uniswap/lib/contracts/libraries/FullMath.sol';
5 import
  '@uniswap/lib/contracts/libraries/Babyonion.sol';
6 import './BitMath.sol';
7
8 // A library for handling binary fixed-point numbers
  (https://en.wikipedia.org/wiki/Q_(number_format))
9 library FixedPoint {
10     // Range: [0, 2**112 - 1]
11     // Resolution: 1 / 2**112
12     struct uq112x112 {
13         uint224 _x;
14     }
15
16     // Range: [0, 2**144 - 1]
17     // Resolution: 1 / 2**112
18     struct uq144x112 {
19         uint256 _x;
20     }
```

+ 38 additions

147 lines

```
1 // SPDX-License-Identifier: GPL-3.0-or-later
2 pragma solidity >=0.8.20;
3
4 import
  '@uniswap/lib/contracts/libraries/FullMath.sol';
5 import
  '@uniswap/lib/contracts/libraries/Babyonion.sol';
6 import './BitMath.sol';
7
8 // a library for handling binary fixed-point numbers
  (https://en.wikipedia.org/wiki/Q_(number_format))
9 library FixedPoint {
10     // range: [0, 2**112 - 1]
11     // resolution: 1 / 2**112
12     struct uq112x112 {
13         uint224 _x;
14     }
15
16     // range: [0, 2**144 - 1]
17     // resolution: 1 / 2**112
18     struct uq144x112 {
19         uint256 _x;
20     }
```

```

21
22     uint8 public constant RESOLUTION =
    112;
23     uint256 public constant Q112 =
    0x10000000000000000000000000000000; //
    2**112
24     uint256 private constant Q224 =
    0x1000000000000000000000000000000000000000000000000000000000000000; // 2**224
25     uint256 private constant LOWER_MASK
    = 0xffffffffffffffffffffffffffffffff; //
    Decimal of UQ*x112 (lower 112 bits)
26
27     // Encode a uint112 as a UQ112x112
28     function encode(uint112 x) internal
    pure returns (uq112x112 memory) {
29         return uq112x112(uint224(x) <<
    RESOLUTION);
30     }
31
32     // Encodes a uint144 as a UQ144x112
33     function encode144(uint144 x)
    internal pure returns (uq144x112
    memory) {
34         return uq144x112(uint256(x) <<
    RESOLUTION);
35     }
36
37     // Decode a UQ112x112 into a
    uint112 by truncating after the radix
    point
38     function decode(uq112x112 memory
    self) internal pure returns (uint112) {
39         return uint112(self._x >>
    RESOLUTION);
40     }
41
42     // Decode a UQ144x112 into a
    uint144 by truncating after the radix
    point
43     function decode144(uq144x112 memory
    self) internal pure returns (uint144) {
44         return uint144(self._x >>
    RESOLUTION);
45     }

```

```

21
22     uint8 public constant RESOLUTION =
    112;
23     uint256 public constant Q112 =
    0x10000000000000000000000000000000; //
    2**112
24     uint256 private constant Q224 =
    0x1000000000000000000000000000000000000000000000000000000000000000; // 2**224
25     uint256 private constant LOWER_MASK
    = 0xffffffffffffffffffffffffffffffff; //
    decimal of UQ*x112 (lower 112 bits)
26
27     // encode a uint112 as a UQ112x112
28     function encode(uint112 x) internal
    pure returns (uq112x112 memory) {
29         return uq112x112(uint224(x) <<
    RESOLUTION);
30     }
31
32     // encodes a uint144 as a UQ144x112
33     function encode144(uint144 x)
    internal pure returns (uq144x112
    memory) {
34         return uq144x112(uint256(x) <<
    RESOLUTION);
35     }
36
37     // decode a UQ112x112 into a
    uint112 by truncating after the radix
    point
38     function decode(uq112x112 memory
    self) internal pure returns (uint112) {
39         return uint112(self._x >>
    RESOLUTION);
40     }
41
42     // decode a UQ144x112 into a
    uint144 by truncating after the radix
    point
43     function decode144(uq144x112 memory
    self) internal pure returns (uint144) {
44         return uint144(self._x >>
    RESOLUTION);
45     }

```

```

46
47 // Multiply a UQ112x112 by a uint,
   returning a UQ144x112
48 // Reverts on overflow
49 function mul(uq112x112 memory self,
   uint256 y) internal pure returns
   (uq144x112 memory) {
50     uint256 z = 0;
51     require(y == 0 || (z = self._x
   * y) / y == self._x, 'FixedPoint::mul:
   overflow');
52     return uq144x112(z);
53 }
54
55 // Multiply a UQ112x112 by an int
   and decode, returning an int
56 // Reverts on overflow
57 function muli(uq112x112 memory
   self, int256 y) internal pure returns
   (int256) {
58     uint256 z =
   FullMath.mulDiv(self._x, uint256(y < 0
   ? -y : y), Q112);
59     require(z < 2**255,
   'FixedPoint::muli: overflow');
60     return y < 0 ? -int256(z) :
   int256(z);
61 }
62
63 // Multiply a UQ112x112 by a
   UQ112x112, returning a UQ112x112
64 // Lossy
65 function muluq(uq112x112 memory
   self, uq112x112 memory other) internal
   pure returns (uq112x112 memory) {
66     if (self._x == 0 || other._x ==
   0) {
67         return uq112x112(0);
68     }
69     uint112 upper_self =
   uint112(self._x >> RESOLUTION); // *
   2^0
70     uint112 lower_self =
   uint112(self._x & LOWER_MASK); // *
   2^-112

```

```

46
47 // multiply a UQ112x112 by a uint,
   returning a UQ144x112
48 // reverts on overflow
49 function mul(uq112x112 memory self,
   uint256 y) internal pure returns
   (uq144x112 memory) {
50     uint256 z = 0;
51     require(y == 0 || (z = self._x
   * y) / y == self._x, 'FixedPoint::mul:
   overflow');
52     return uq144x112(z);
53 }
54
55 // multiply a UQ112x112 by an int
   and decode, returning an int
56 // reverts on overflow
57 function muli(uq112x112 memory
   self, int256 y) internal pure returns
   (int256) {
58     uint256 z =
   FullMath.mulDiv(self._x, uint256(y < 0
   ? -y : y), Q112);
59     require(z < 2**255,
   'FixedPoint::muli: overflow');
60     return y < 0 ? -int256(z) :
   int256(z);
61 }
62
63 // multiply a UQ112x112 by a
   UQ112x112, returning a UQ112x112
64 // lossy
65 function muluq(uq112x112 memory
   self, uq112x112 memory other) internal
   pure returns (uq112x112 memory) {
66     if (self._x == 0 || other._x ==
   0) {
67         return uq112x112(0);
68     }
69     uint112 upper_self =
   uint112(self._x >> RESOLUTION); // *
   2^0
70     uint112 lower_self =
   uint112(self._x & LOWER_MASK); // *
   2^-112

```

```

71      uint112 upper_other =
      uint112(other._x >> RESOLUTION); // *
      2^0
72      uint112 lower_other =
      uint112(other._x & LOWER_MASK); // *
      2^-112
73
74      // Partial products
75      uint224 upper =
      uint224(upper_self) * upper_other; // *
      2^0
76      uint224 lower =
      uint224(lower_self) * lower_other; // *
      2^-224
77      uint224 uppers_lowero =
      uint224(upper_self) * lower_other; // *
      2^-112
78      uint224 uppero_lowers =
      uint224(upper_other) * lower_self; // *
      2^-112
79
80      // So the bit shift does not
      overflow
81      require(upper <=
      type(uint112).max, 'FixedPoint::muluq:
      upper overflow');
82
83      // This cannot exceed 256 bits,
      all values are 224 bits
84      uint256 sum = uint256(upper <<
      RESOLUTION) + uppers_lowero +
      uppero_lowers + (lower >> RESOLUTION);
85
86      // So the cast does not
      overflow
87      require(sum <=
      type(uint224).max, 'FixedPoint::muluq:
      sum overflow');
88
89      return uq112x112(uint224(sum));
90  }
91
92      // Divide a UQ112x112 by a
      UQ112x112, returning a UQ112x112

```

```

71      uint112 upper_other =
      uint112(other._x >> RESOLUTION); // *
      2^0
72      uint112 lower_other =
      uint112(other._x & LOWER_MASK); // *
      2^-112
73
74      // partial products
75      uint224 upper =
      uint224(upper_self) * upper_other; // *
      2^0
76      uint224 lower =
      uint224(lower_self) * lower_other; // *
      2^-224
77      uint224 uppers_lowero =
      uint224(upper_self) * lower_other; // *
      2^-112
78      uint224 uppero_lowers =
      uint224(upper_other) * lower_self; // *
      2^-112
79
80      // so the bit shift does not
      overflow
81      require(upper <= uint112(-1),
      'FixedPoint::muluq: upper overflow');
82
83      // this cannot exceed 256 bits,
      all values are 224 bits
84      uint256 sum = uint256(upper <<
      RESOLUTION) + uppers_lowero +
      uppero_lowers + (lower >> RESOLUTION);
85
86      // so the cast does not
      overflow
87      require(sum <= uint224(-1),
      'FixedPoint::muluq: sum overflow');
88
89      return uq112x112(uint224(sum));
90  }
91
92      // divide a UQ112x112 by a
      UQ112x112, returning a UQ112x112

```



```

93     function divuq(uq112x112 memory
      self, uq112x112 memory other) internal
      pure returns (uq112x112 memory) {
94         require(other._x > 0,
          'FixedPoint::divuq: division by zero');
95         if (self._x == other._x) {
96             return
              uq112x112(uint224(Q112));
97         }
98         if (self._x <=
          type(uint144).max) { // Changed here
99             uint256 value =
              (uint256(self._x) << RESOLUTION) /
              other._x;
100         require(value <=
          type(uint224).max, 'FixedPoint::divuq:
          overflow');
101         return
            uq112x112(uint224(value));
102     }
103
104     uint256 result =
      FullMath.mulDiv(Q112, self._x,
      other._x);
105     require(result <=
      type(uint224).max, 'FixedPoint::divuq:
      overflow');
106     return
      uq112x112(uint224(result));
107 }
108
109 // Returns a UQ112x112 which
      represents the ratio of the numerator
      to the denominator
110 // Can be lossy
111
112 function fraction(uint256
      numerator, uint256 denominator)
      internal pure returns (uq112x112
      memory) {
113     require(denominator > 0,
      'FixedPoint::fraction: division by
      zero');
114     if (numerator == 0) return
      FixedPoint.uq112x112(0);
115 }

```

```

93     function divuq(uq112x112 memory
      self, uq112x112 memory other) internal
      pure returns (uq112x112 memory) {
94         require(other._x > 0,
          'FixedPoint::divuq: division by zero');
95         if (self._x == other._x) {
96             return
              uq112x112(uint224(Q112));
97         }
98         if (self._x <= uint144(-1)) {
99             uint256 value =
              (uint256(self._x) << RESOLUTION) /
              other._x;
100         require(value <=
          uint224(-1), 'FixedPoint::divuq:
          overflow');
101         return
            uq112x112(uint224(value));
102     }
103
104     uint256 result =
      FullMath.mulDiv(Q112, self._x,
      other._x);
105     require(result <= uint224(-1),
      'FixedPoint::divuq: overflow');
106     return
      uq112x112(uint224(result));
107 }
108
109 // returns a UQ112x112 which
      represents the ratio of the numerator
      to the denominator
110 // can be lossy
111
112 function fraction(uint256
      numerator, uint256 denominator)
      internal pure returns (uq112x112
      memory) {
113     require(denominator > 0,
      'FixedPoint::fraction: division by
      zero');
114     if (numerator == 0) return
      FixedPoint.uq112x112(0);
115 }

```

```

115     if (numerator <=
type(uint144).max) { // Changed here
116         uint256 result = (numerator
<< RESOLUTION) / denominator;
117         require(result <=
type(uint224).max,
'FixedPoint::fraction: overflow');
118         return
uq112x112(uint224(result));
119     } else {
120         uint256 result =
FullMath.mulDiv(numerator, Q112,
denominator);
121         require(result <=
type(uint224).max,
'FixedPoint::fraction: overflow');
122         return
uq112x112(uint224(result));
123     }
124 }
125
126 // Take the reciprocal of a
UQ112x112
127 // Reverts on overflow
128 // Lossy
129 function reciprocal(uq112x112
memory self) internal pure returns
(uq112x112 memory) {
130     require(self._x != 0,
'FixedPoint::reciprocal: reciprocal of
zero');
131     require(self._x != 1,
'FixedPoint::reciprocal: overflow');
132     return uq112x112(uint224(Q224 /
self._x));
133 }
134
135 // Square root of a UQ112x112
136 // Lossy between 0/1 and 40 bits
137 function sqrt(uq112x112 memory
self) internal pure returns (uq112x112
memory) {
138     if (self._x <=
type(uint144).max) { // Changed here

```

```

115     if (numerator <= uint144(-1)) {
116         uint256 result = (numerator
<< RESOLUTION) / denominator;
117         require(result <=
uint224(-1), 'FixedPoint::fraction:
overflow');
118         return
uq112x112(uint224(result));
119     } else {
120         uint256 result =
FullMath.mulDiv(numerator, Q112,
denominator);
121         require(result <=
uint224(-1), 'FixedPoint::fraction:
overflow');
122         return
uq112x112(uint224(result));
123     }
124 }
125
126 // take the reciprocal of a
UQ112x112
127 // reverts on overflow
128 // lossy
129 function reciprocal(uq112x112
memory self) internal pure returns
(uq112x112 memory) {
130     require(self._x != 0,
'FixedPoint::reciprocal: reciprocal of
zero');
131     require(self._x != 1,
'FixedPoint::reciprocal: overflow');
132     return uq112x112(uint224(Q224 /
self._x));
133 }
134
135 // square root of a UQ112x112
136 // lossy between 0/1 and 40 bits
137 function sqrt(uq112x112 memory
self) internal pure returns (uq112x112
memory) {
138     if (self._x <= uint144(-1)) {

```

```
139         return
        uq112x112(uint224(Babylonian.sqrt(uint2
        56(self._x) << 112)));
140     }
141
142     uint8 safeShiftBits = 255 -
        BitMath.mostSignificantBit(self._x);
143     safeShiftBits -= safeShiftBits
        % 2;
144     return
        uq112x112(uint224(Babylonian.sqrt(uint2
        56(self._x) << safeShiftBits) << ((112
        - safeShiftBits) / 2)));
145 }
146 }
147
```

```
139         return
        uq112x112(uint224(Babylonian.sqrt(uint2
        56(self._x) << 112)));
140     }
141
142     uint8 safeShiftBits = 255 -
        BitMath.mostSignificantBit(self._x);
143     safeShiftBits -= safeShiftBits
        % 2;
144     return
        uq112x112(uint224(Babylonian.sqrt(uint2
        56(self._x) << safeShiftBits) << ((112
        - safeShiftBits) / 2)));
145 }
146 }
147
```



Comparing sensitive data, confidential files or internal emails?

Most legal and privacy policies prohibit uploading sensitive data online. Diffchecker Desktop ensures your confidential information never leaves your computer. Work offline and compare documents securely.

[Download Diffchecker Desktop](#)
[Learn more](#)

Untitled diff

– 8 removals

52 lines

```

1 // SPDX-License-Identifier: CC-BY-4.0
2 pragma solidity >=0.4.0;
3
4 // Taken from
   https://medium.com/coinmonks/math-in-
   solidity-part-3-percents-and-
   proportions-4db014e080b1
5 // License is CC-BY-4.0
6 library FullMath {
7     function fullMul(uint256 x, uint256
   y) internal pure returns (uint256 l,
   uint256 h) {
8         uint256 mm = mulmod(x, y,
   type(uint256).max);
9         l = x * y;
10        h = mm - l;
11        if (mm < l) h -= 1;
12    }
13
14    function fullDiv(
15        uint256 l,
16        uint256 h,
17        uint256 d
18    ) private pure returns (uint256) {
19        uint256 pow2 = d & (~d + 1); //
   Changed here
20        d /= pow2;
21        l /= pow2;
```

+ 6 additions

52 lines

```

1 // SPDX-License-Identifier: CC-BY-4.0
2 pragma solidity >=0.4.0;
3
4 // taken from
   https://medium.com/coinmonks/math-in-
   solidity-part-3-percents-and-
   proportions-4db014e080b1
5 // license is CC-BY-4.0
6 library FullMath {
7     function fullMul(uint256 x, uint256
   y) internal pure returns (uint256 l,
   uint256 h) {
8         uint256 mm = mulmod(x, y,
   uint256(-1));
9         l = x * y;
10        h = mm - l;
11        if (mm < l) h -= 1;
12    }
13
14    function fullDiv(
15        uint256 l,
16        uint256 h,
17        uint256 d
18    ) private pure returns (uint256) {
19        uint256 pow2 = d & -d;
20        d /= pow2;
21        l /= pow2;
```

```

22      // l += h * ((-pow2 / pow2) +
1); // Changed here, Just commenting it
is useless and out of our use

23      uint256 r = 1;
24      r *= 2 - d * r;
25      r *= 2 - d * r;
26      r *= 2 - d * r;
27      r *= 2 - d * r;
28      r *= 2 - d * r;
29      r *= 2 - d * r;
30      r *= 2 - d * r;
31      r *= 2 - d * r;
32      return l * r;
33  }
34
35  function mulDiv(
36      uint256 x,
37      uint256 y,
38      uint256 d
39  ) internal pure returns (uint256) {
40      (uint256 l, uint256 h) =
fullMul(x, y);
41
42      uint256 mm = mulmod(x, y, d);
43      if (mm > l) h -= 1;
44      l -= mm;
45
46      if (h == 0) return l / d;
47
48      require(h < d, 'FullMath:
FULLDIV_OVERFLOW');
49      return fullDiv(l, h, d);
50  }
51 }
52

```

```

22      l += h * ((-pow2) / pow2 + 1);

23      uint256 r = 1;
24      r *= 2 - d * r;
25      r *= 2 - d * r;
26      r *= 2 - d * r;
27      r *= 2 - d * r;
28      r *= 2 - d * r;
29      r *= 2 - d * r;
30      r *= 2 - d * r;
31      r *= 2 - d * r;
32      return l * r;
33  }
34
35  function mulDiv(
36      uint256 x,
37      uint256 y,
38      uint256 d
39  ) internal pure returns (uint256) {
40      (uint256 l, uint256 h) =
fullMul(x, y);
41
42      uint256 mm = mulmod(x, y, d);
43      if (mm > l) h -= 1;
44      l -= mm;
45
46      if (h == 0) return l / d;
47
48      require(h < d, 'FullMath:
FULLDIV_OVERFLOW');
49      return fullDiv(l, h, d);
50  }
51 }
52

```