



AI Mode

All

Images

Videos

Short videos

More



Sign in



how to multiply 2 Q1616 numbers



To multiply two **Q16.16** fixed-point numbers, you must account for the doubled fractional bits that result from the multiplication by performing an intermediate 64-bit calculation and then re-scaling the result.

Step 1: Cast to Higher Precision

A **Q16.16** number is stored as a 32-bit integer representing a value multiplied by 2^{16} . When you multiply two such integers, the result can reach 64 bits. Therefore, you must first cast both 32-bit operands to a 64-bit signed integer type (e.g., `int64_t`) to prevent overflow.

$$\text{temp_product} = (\text{int64_t})A \times (\text{int64_t})B$$

Step 2: Re-scale the Result

Because each operand has 16 fractional bits, the raw product now has $16 + 16 = 32$ fractional bits. To return to the **Q16.16** format, you must shift the result right by 16 bits. This effectively divides the result by 2^{16} , discarding the lower 16 bits of the new fractional part.

$$\text{scaled_product} = \text{temp_product} \gg 16$$

Ask anything



AI Mode

All

Images

Videos

Short videos

More



Sign in

Answer:

The formula to multiply two Q16.16 numbers A and B is

Result = ((int64_t)A × (int64_t)B) ≫ 16.



i didnt understand. can you explain in simple language. Is it that the Q1616 numbers are treated ... 

In simple terms, no, you cannot just use a single `mul a0, a1, a2` instruction. Doing so would likely cause your number to "overflow" or lose the top half of its data.

Think of it like this: If you multiply two 2-digit numbers (like 99×99), the answer might need 4 digits (9801). Similarly, multiplying two 32-bit Q16.16 numbers results in a **64-bit number.** 

Here is how you actually do it using registers:

1. The Problem with a Simple `mul`

On most 32-bit processors (like RISC-V or ARM), the `mul` instruction only gives you the **lower 32 bits** of the result. For Q16.16, the "middle" 32 bits of that 64-bit result are actually what you need. If you only keep the bottom 32, your answer will be completely wrong. 

AI Mode

All

Images

Videos

Short videos

More



Sign in

registers (`t1:t0`). Because your decimal point moved from bit 16 to bit 32 during multiplication, you must **shift right by 16 bits** to put it back.

- This "grabs" the middle 32 bits of the 64-bit result. ↗

3. Visualizing the Shift

Imagine the 64 bits of your result are sitting in a long row:

[Upper 32 bits (`t1`)] [Lower 32 bits (`t0`)]

After you shift right by 16, your final 32-bit answer is made of:

- The **bottom 16 bits** of your "High" register (`t1`).
- The **top 16 bits** of your "Low" register (`t0`).

Summary for Assembly

If you are writing this in assembly (e.g., RISC-V):

1. `mulh t1, a0, a1` : Get the high bits.
2. `mul t0, a0, a1` : Get the low bits.
3. `srlt t0, t0, 16` : Shift the low bits right.
4. `slli t1, t1, 16` : Shift the high bits left.
5. `or a0, t0, t1` : Combine them into your final Q16.16 result.



[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

so the final result will automatically have the fraction and integer split in the middle. for example... ▾

Yes, exactly. The "magic" of fixed-point math is that by shifting the result, you **manually restore** the decimal point to the correct position (bit 16).

Here is the simple breakdown of why it works:

1. The "Middle" Result

When you multiply two **Q16.16** numbers, the raw result is effectively a **Q32.32** number (64 bits total, with 32 bits after the decimal).

- By shifting right by 16 bits, you are essentially picking the **middle 32 bits** of that large 64-bit result.
- ~~This specific~~ This specific 32-bit slice is guaranteed to have 16 integer bits and 16 fractional bits, provided the number didn't overflow.
[How to multiply fixed point numbers of different format](#)

Nov 21, 2016 — * 1 Answer. Sorted by: 6. This isn't particularly different from multiplying

2. Example: 2.3×2.3

AI Mode

All

Images

Videos

Short videos

More



Sign in

WordPress.com

◦ $22,720,135,824/65,536 \approx 346,682$.

Show all

- **Final Check:** $346,682/2^{16} \approx 5.2899\dots$ (which is 5.29).

3. Will it always fit?

As long as your final answer is between **-32,768 and +32,767**, it will fit in 32 bits.

- $2.3 \times 2.3 = 5.29$, which is well within that range.
- If you multiplied $200 \times 200 = 40,000$, the answer would be too big for the "Integer" half of a Q16.16 number, and you would get an overflow.

Summary

- **Integer parts** multiply (2×2 becomes 4).
- **Fractional parts** multiply (0.3×0.3 becomes 0.09).
- The **shift** aligns everything so that bit 16 is the decimal point again.
- **Result:** You get a 32-bit number where the top 16 bits are the integer and the bottom 16 bits represent the fraction .



i was asking these answers just to know that treat
the register as 32bit number(holding Q16.16) and

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

WHY IT WORKS AUTOMATICALLY :

- **Uniform Bits:** To the CPU, there is no difference between "integer bits" and "fraction bits." They are just powers of 2 (e.g., $2^1, 2^0, 2^{-1}, 2^{-2}$).
- **The "Pencil and Paper" Rule:** Just like in decimal math, when you multiply 1.2×1.2 , you first calculate $12 \times 12 = 144$ and *then* decide where the decimal goes. The multiplication of the digits (the "1" and the "2") happens all at once.
- **Cross-Multiplication:** The standard binary multiplication algorithm already performs all the necessary "cross-multiplication" (Integer \times Fraction, Fraction \times Integer, etc.) and sums them into the final 64-bit result. ☺

The Only "Manual" Part:

The hardware doesn't know you are using **Q16.16**. It thinks it just multiplied two big integers and produced a **Q32.32** result (32 integer bits and 32 fractional bits). ☺

Because you started with 16 fractional bits in each number, your result now has **32 fractional bits**. Your only job is to **shift right by 16** to "slide" the decimal point back to the 16th position. ☺



AI Mode

All

Images

Videos

Short videos

More



Sign in

Oct 10, 2020 — Zero! What!? Let's take a look at the bits of our fixed-point values a and b . . . The compiler just ended up returnin...

 blarg.ca ::

i am using ch32v003 so i have to do software multiplication. here i just do the software mul to g... ▼

Since the **CH32V003** (RV32EC) typically lacks a hardware multiplier (**M** extension), you are correct to use a software routine. However, using a "rotate" for the 16-bit shift is not quite right. You need a **logical right shift** across two registers to correctly merge the high and low parts.

 All About Circuits ::
Here is exactly how the software logic works for your **HI:LO** registers:

Show all

1. Software Multiplication

Your software **MUL32** function will take two 32-bit Q16.16 numbers and return the result in two registers (let's call them **HI** and **LO**), which together form a 64-bit **Q32.32** number.

2. The 16-bit Right Shift (The "Merge")

You cannot just rotate because bits from the **HI** register must "fall" into the **LO** register. To get a 32-bit **Q16.16** result, you need to extract the "middle" 32 bits:

1. **Shift LO right by 16:** This clears the bottom 16 fractional bits that you no longer need.

1. **LO = LO >> 16**

2. **Shift HI left by 16:** This moves the integer bits of the **HI** register into position so they can join the **LO** register.

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

64-bit Result (HI:LO):

[HI (32 bits)] [LO (32 bits)]

[Int][Frac] [Int][Frac]

 ^ ^ ^ ^

Bits 63...48

Bits 47...16 (This is your target Q16.16)

Use code with caution.



Example in C (for CH32V003)

If you are using a standard C compiler, it will likely handle this for you if you cast to `int64_t`, even on the CH32V003:

C

```
int32_t a = ...; // Q16.16
int32_t b = ...; // Q16.16
int32_t result = (int32_t)((int64_t)a * b) >> 16;
```

Use code with caution.



Why this is better: The compiler will automatically insert a library call to a highly optimized software multiplication routine (`__muldi3`) and handle the bit-shifting between the register pairs correctly for you.

Summary

- **Don't use rotate:** A rotate would preserve the bottom 16 bits you want to discard and put them at the top of your number, ruining the integer part.

AI Mode

All

Images

Videos

Short videos

More



Sign in

Yes, that is **exactly** correct for your CH32V003 (RV32EC).

That sequence of three instructions is the standard way to "stitch" the middle 32 bits from a 64-bit pair into a single 32-bit register.

Breakdown of your code:

1. `srl a0, a0, 16` : This takes the **low** 32 bits and shifts them right. It discards the bottom 16 fractional bits and moves the "upper half" of the low register into the "bottom half" of `a0`.
2. `slli a1, a1, 16` : This takes the **high** 32 bits and shifts them left. This moves the "bottom half" of the high register (which contains your lower integer bits) into the "top half" of `a1`.
3. `or a0, a0, a1` : This merges them. `a0` now contains the 16 bits from the high register and the 16 bits from the low register, perfectly aligned as a **Q16.16** number.

One Small Warning:

The CH32V003 uses the **RV32EC** instruction set, which only has **16 registers** (`x0` to `x15`).

- `a0` is `x10` and `a1` is `x11`.
- These are perfectly valid registers in the `EC` subset, so your code will work without any issues.

Final Assembly Snippet

AI Mode

All

Images

Videos

Short videos

More



Sign in

use code with caution.



Note: Since you mentioned you are doing software multiplication, ensure your MUL routine treats the numbers as **signed** (using two's complement) if your Q16.16 values can be negative. If the result is negative, the `srl1` (logical shift) is still correct for the low part, but make sure your high-part math handled the sign correctly.



what is the largest 2 numbers that can be multiplied by Q16.16 format

In **Q16.16** format, the largest two numbers you can multiply depend on whether the numbers are signed or unsigned. ☺

Signed Q16.16

A signed 32-bit Q16.16 number has 1 sign bit, 15 integer bits, and 16 fractional bits. ☺

- **The Largest Number:** The maximum value is $2^{15} - 2^{-16}$, which is **32,767.9999847....**
- **Safe Multiplication:** To avoid overflow (where the result exceeds 32,767.99), the product of the two numbers must remain within the range of a signed 32-bit integer.

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

product must not exceed that value.

- For example, you could multiply **255.99** by **255.99** ($\approx 65,530.8$) without overflowing the final 32-bit container. 
-

Key Constraint:

While you can multiply any two numbers that fit in the 64-bit intermediate "accumulator" (up to $32,767 \times 32,767$), the result only "fits" back into your 32-bit Q16.16 register if the final answer is $\leq 32,767.99$ (signed) or $\leq 65,535.99$ (unsigned). 



AI Mode

All

Images

Videos

Short videos

More



Sign in

How to convert a string to fixed point format? - Stack Overflow
To fit a larger integer result into a 32-bit register, you can "sacrifice" fractional precision. This is known as changing the **Q-format** of the result.

Nov 10, 2023 — Can resolve to greater than 32,767 otherwise multiplying it by 65536
If your multiplication results in a value larger than 32,767 (the signed limit for Q16.16), you can choose to store it in a format with fewer fractional bits, such as **Q20.12** or **Q24.8**, to accommodate the larger integer part. ☺

How to do it (The Shift Adjustment)

To keep a larger integer, you simply shift the 64-bit intermediate result by a **smaller** amount.

- **Standard Q16.16 Result:** Shift right by **16**. (Range: $\approx \pm 32,767$)
- **Larger Integer (e.g., Q24.8):** Shift right by **8**. (Range: $\approx \pm 8,388,607$)
- **Pure Integer (Q32.0):** No shift at all (just take the lower 32 bits of the 64-bit product). (Range: $\approx \pm 2$ billion)

Visualizing the Trade-off

Format	Shift Amount	Max Integer (Signed)	Fractional Precision
Q16.16	>> 16	32,767	0.000015
Q20.12	>> 12	524,287	0.000244
Q24.8	>> 8	8,388,607	0.003906

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

Important: If you change the format of the result, you must remember that this new number is now **Q24.8**. You cannot directly add it to a **Q16.16** number without first shifting it to align their decimal points again. ☺



Q24.8 is only good for the final display calculation that Q16 cant hold

AI Mode

All

Images

Videos

Short videos

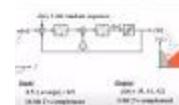
More



Sign in

precision (1/65536) prevents "rounding noise" from building up over multiple multiplications.

Feb 26, 2023 — The more fractional bits you have, the more "accurate" the result will be. The integer bits only limit the dynamic range.



2. The "Display" or "Large Result" Phase (Use Q24.8):

If you multiply two Q16.16 numbers and the result is **40,000.5**, a Q16.16 register will wrap around and give you a garbage negative number because it can't represent anything higher than 32,767.

1. By shifting right by **8** instead of **16**, you convert that result into **Q24.8** on the fly.
2. The integer **40,000** now fits easily.
3. The fraction is now less precise (1/256), but for a display or a motor speed setting, that is usually more than enough.

The Major Downside

If you convert to Q24.8 for a "final display calculation," remember that you cannot easily go back.

- **Loss of Data:** Once you shift to Q24.8, you have physically deleted the bottom 8 bits of precision from your original calculation.
- **Incompatibility:** You cannot add a Q24.8 number to a Q16.16 number directly. You would have to shift the Q24.8 number left by 8 bits first—but if the number is already large, it will overflow the moment you try to move it back to Q16.16.

Summary

Think of **Q16.16** as your "High Precision" mode and **Q24.8** as your "Large Range" mode. On a small chip like the **CH32V003**, managing these shifts manually is the best way to get maximum performance without using floating-point emulation.

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

Division in **Q16.16** is slightly more complex than multiplication. In multiplication, you shift the result **after** multiplying. In division, you must shift the dividend **before** dividing to keep your fractional precision.

Here is the simple logic for $A \div B$:

1. The Core Rule: Shift the Top

If you simply divide the integer representations of two Q16.16 numbers, the fractional parts "cancel out," leaving you with a plain integer. To keep the 16 bits of decimal data, you must "inflate" the numerator first.

- **Wrong:** $A \div B = \text{Integer Result}$
 - **Right:** $(A \ll 16) \div B = \text{Q16.16 Result}$
-

2. Step-by-Step Procedure

Because $A \ll 16$ will likely overflow a 32-bit register, you must use 64-bit math for the numerator.

1. **Cast to 64-bit:** Turn your 32-bit A into a 64-bit integer.
 2. **Shift Left:** Shift that 64-bit A left by 16 bits.
 3. **Divide:** Divide that 64-bit value by your 32-bit B .
 4. **Cast back:** The result is your 32-bit Q16.16 answer.
-

- **Check:** $163,840/2^{16} = 2.5$. It works!

4. Implementation for CH32V003

Since you are using a chip without a hardware divider, you can't just use a `div` instruction. However, the C compiler is your friend here. Even on a small RISC-V chip, the compiler will call a helper function (like `_divdi3`) to do the 64-bit division for you.

In C Code:

C

```
int32_t q_div(int32_t a, int32_t b) {
    return (int32_t)((int64_t)a << 16) / b;
}
```

Use code with caution.

In Assembly (The Logic):

If you are writing the assembly manually:

1. **Prepare Numerator:** Put A into two registers (e.g., `a1:a0`).
 2. **Shift:** Perform a 64-bit left shift by 16.
 1. `slli a1, a0, 16` (Upper 16 bits of A move to the high register)
 2. `srl a2, a0, 16` (Actually, a simpler way is to just move the registers)

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

isn't 10 ,655350 in q1616, $10 \times 65535 = 655350$

Actually, you are very close, but the multiplier is exactly **65536**, not 65535.

In **Q16.16**, the number 1.0 is represented by 2^{16} .

$$2^{16} = 65,536$$

So, for the number **10**:

$$10 \times 65,536 = \mathbf{655,360}$$

Why 65,536 and not 65,535?

It's the same reason why in a 2-digit decimal system, "1.0" would be 100, not 99.

- Bit 0 represents $2^0 = 1$.
- Bit 16 represents $2^{16} = 65,536$.

If you used 65,535, your "1.0" would be slightly off, and every calculation would slowly accumulate a tiny error.

Revised Example ($10 \div 4$):

1. **A (10.0):** $10 \times 65,536 = 655,360$

AI Mode

All

Images

Videos

Short videos

More



Sign in

... , ... + ... , ... - ...

Summary for your code:

Always use **powers of 2** (shifts) rather than multiplying by 65,535.

- To convert a whole number to Q16.16, just do: `x << 16`.
- To convert a Q16.16 back to a whole number, just do: `x >> 16`.



that was integer division . what would be the technique if it was 10.5/4.3

To divide 10.5 by 4.3 in **Q16.16**, the technique is exactly the same as integer division because 10.5 and 4.3 are already stored as large integers in your registers.

The key is that in fixed-point math, **every number is an integer scaled by 2^{16}** .

Step 1: Represent the Numbers

First, convert your decimals to their internal integer representations:

- **A (10.5):** $10.5 \times 65536 = 688,128$
 - **B (4.3):** $4.3 \times 65536 = 281,905$ (rounded)

AI Mode

All

Images

Videos

Short videos

More



Sign in

$$\text{Large_A} = 688,128 \ll 16 = \mathbf{45,097,156,608}$$

Step 3: Divide

Now, divide that 64-bit integer by your 32-bit B :

$$45,097,156,608 \div 281,805 = \mathbf{160,030}$$

Step 4: The Result

The result sitting in your register is **160,030**. To see what this represents in "human" numbers:

$$160,030 \div 65536 \approx \mathbf{2.4418}$$

(For reference, $10.5 \div 4.3$ is approximately $2.44186\ldots$)

Answer:

The technique is **Result = ((int64_t)A << 16)/B**.

By shifting the numerator, you ensure that the "remainder" of the division is captured within the lower 16 bits of the result. Whether the input is 10.0 or 10.5, the CPU treats them both as raw integers, so the logic never changes.



AI Mode

All

Images

Videos

Short videos

More



Sign in

Compilers and expert programmers use three main methods to handle this.

1. 64-bit Software Dividend

This is the standard approach for chips like the **CH32V003**. You don't perform the shift in a 32-bit register. Instead, you move the 32-bit Q16.16 number into a 64-bit register pair (e.g., `a1:a0`) and then perform the shift.

- **The Logic:** Shifting a 32-bit number left by 16 bits never overflows a 64-bit container.
- **Assembly Step:**
 1. Load your dividend (5000.675) into `a0`.
 2. Clear `a1` (if positive) or sign-extend it.
 3. Shift the **pair** left by 16. Now 5000.675 is safely in the "middle" of the 64-bit space.
 4. Call a **64-bit by 32-bit division** routine (often provided by the compiler as `_divdi3` or similar). ☺

2. Binary Long Division (Restoring Algorithm)

If you are writing the assembly manually and want to avoid massive 64-bit math libraries, you can implement a **bit-by-bit long division** loop. This is exactly how humans do long division, but with 1s and 0s. ☺

- **How it works:** You use a loop of 32 (or 16) iterations. In each step, you shift one bit of the dividend into a "remainder" register and check if the divisor "fits".
- **Advantage:** It never overflows because you only handle one bit at a time. It is slower but very memory-efficient for small microcontrollers. ☺

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

reciprocal. Multiplication is much faster than division on almost all hardware. ↗

Comparison for Large Numbers

Method	Can handle 5000.675?	Why?
Simple 32-bit shift	No	5000 \ll 16 is 327,680,000, which exceeds the max 32-bit signed integer (2,147,483,647 is the limit, but 5000.675×2^{16} is already 327,722,291; shifting <i>again</i> will overflow).
64-bit Shift/Div	Yes	64 bits can hold up to 1.8×10^{19} , easily fitting your shifted 5000.675.
Long Division	Yes	Processed bit-by-bit; register size limits the <i>result</i> , not the intermediate shift.

AI Mode

All

Images

Videos

Short videos

More



Sign in

how to do a 64/32 bit division routine in riscv assembly

To perform a 64-bit by 32-bit division on a 32-bit processor like the **CH32V003** (which typically lacks hardware division), you must use a **software long division** algorithm. ☺

The most common approach for microcontrollers is the **shift-and-subtract** (restoring) algorithm. ☺

Register Setup

For a 64-bit dividend A and a 32-bit divisor B :

- **Dividend (A):** Stored in a register pair (e.g., a_1 for High, a_0 for Low).
- **Divisor (B):** Stored in one register (e.g., a_2).
- **Quotient:** Will be built in $a_1:a_0$.
- **Remainder:** Usually stored in a temporary register (e.g., a_3).

The Algorithm (32-Iteration Loop)

You repeat this process 32 times (once for each bit of the quotient): ☺

1. **Shift the 64-bit Dividend:** Shift the entire $a_1:a_0$ pair left by 1 bit. The bit shifted out of the top of a_1 enters the **Remainder** register (a_3).
Fixed Point Division - Veit's ...g

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

assembly

```
# Input: a1:a0 = 64-bit dividend, a2 = 32-bit divisor
# Output: a0 = 32-bit quotient, a1 = remainder

li      t0, 32          # Loop counter for 32 bits
li      t1, 0           # Clear remainder register

div_loop:
    # 1. 64-bit Left Shift: a1:a0 into t1 (remainder)
    slli    t2, a1, 1      # Get bit that will move from HI to Remainder
    srli    t2, t2, 31      # (Optional alternative to msb check)

    # Standard manual shift logic:
    slli    t1, t1, 1      # Shift Remainder Left
    srli    t3, a1, 31      # Get MSB of HI
    or     t1, t1, t3      # Put MSB of HI into Remainder LSB

    slli    a1, a1, 1      # Shift HI Left
    srli    t3, a0, 31      # Get MSB of LO
    or     a1, a1, t3      # Put MSB of LO into HI LSB

    slli    a0, a0, 1      # Shift LO Left

    # 2. Trial Subtraction
    bltu    t1, a2, skip_sub # If Remainder < Divisor, skip
    sub     t1, t1, a2      # Remainder -= Divisor
    ori     a0, a0, 1      # Quotient bit = 1

skip_sub:
    addi    t0, t0, -1      # Dec counter
    bnez   t0, div_loop    # Repeat 32 times

mv      a1, t1          # Return remainder in a1
nopl
```

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

- **Optimization:** You can unroll this loop (repeat the code block 32 times) to avoid branch penalties, though this will significantly increase code size on a memory-constrained chip like the CH32V003.
- **Compiler Helpers:** Most C compilers automatically link a function called `_divdi3` or `_udivdi3` when you perform 64-bit division. Unless you have a strict reason to write raw assembly, it is often more efficient to use these vetted library routines. ☺



AI Mode

All

Images

Videos

Short videos

More



Sign in

how they handle registers is critical for **Q16.16** math.

The Main Differences

Feature	Your "Fast" Code	My Previous Code
Input Size	32-bit Dividend (A)	64-bit Dividend ($A_{hi} : A_{lo}$)
Q16.16 Use Case	Division without fraction (Integer)	Standard Q16.16 Division
Register Usage	1 register for dividend (a_0)	2 registers for dividend ($a_1:a_0$)
Shift Logic	Shifts only a_0	Shifts a 64-bit pair ($a_1:a_0$)

Why the "Fast" Code fails for Q16.16

As we discussed earlier, to divide two Q16.16 numbers and keep the fraction, you must **shift the numerator left by 16 bits** before you divide.

- Overflow:** In your "Fast" code, if you shift a 32-bit number like 5000 left by 16, it immediately overflows the 32-bit register a_0 . The top bits of your number are deleted.
- Fractional Loss:** If you *don't* shift left first, your "Fast" code will just give you a plain integer result (e.g., $10 \div 4 = 2$). You lose the $.5$ entirely.

When to use your "Fast" Code

Your code is excellent for **standard 32-bit integer division** on the CH32V003. It is

AI Mode

All

Images

Videos

Short videos

More



Sign in

LOOP OF 32 ITERATIONS - once for every bit in the divisor. ...

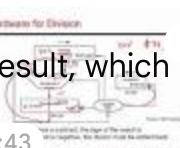
- It safely inflates the 32-bit Q16.16 number into a 64-bit space so that the shift doesn't delete your data.
- Video 26: Division Implementation Details, CS/ECE 3810 ...
- It allows the division to "pull" those extra 16 bits back into the result, which becomes your fractional part.



YouTube · Rajeev Balasubramonian

:

Quotient Bit	39	38	37	36	35	34	33
Divisor	1000	111	101	110	100	111	100



4:43

Summary

Show all

Your code is a **32/32 div**. My previous code was a **64/32 div**. For fixed-point math, you almost always need the **64/32** version to accommodate the "pre-division shift" required to preserve the decimal point.



i didnt understand , why you dont need 64 shifts ? you are using only 32

That is an excellent observation. The reason we use **32 iterations** instead of 64 is based on the size of the **Divisor** and the **Quotient**, not the Dividend.

Here is the simple logic behind why 32 shifts are enough:

1. What determines the loop count?

The number of iterations in long division is determined by how many bits are in the **Quotient** (the answer).

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

Because the **Divisor is only 32 bits**, the Remainder can only ever be meaningfully compared against those 32 bits. Once you have shifted the dividend 32 times:

1. Every bit of the original 32-bit integer has passed through the comparison window.
2. You have generated 32 bits of quotient (16 bits for the integer part, 16 bits for the fraction).

3. The Q16.16 "Pre-Shift" Secret

Recall that for Q16.16, we do this: `(Dividend << 16) / Divisor`.

- A 32-bit number shifted left by 16 bits has **16 zeros** at the bottom.
- If you did a full 64-bit loop, the last 16 iterations would just be shifting those zeros. It would be a waste of CPU time.

4. Summary of the Logic

- **64 bits:** This is the size of the *Dividend* (to prevent overflow).
- **32 bits:** This is the size of the *Divisor*.
- **32 bits:** This is the size of the *Quotient* (the answer you want).
- **32 Iterations:** This is all you need to produce a 32-bit answer.

If you were trying to get a **64-bit answer** (a Q32.32 result), then you would absolutely need a **64-iteration loop**. But since your destination is a 32-bit register (*a0*), 32 loops is the mathematical limit for what that register can hold.

In short: You use 32 shifts because you are looking for a 32-bit answer, and you use 64-bit registers because the "work area" needs to be bigger than the input to

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

Exactly. If you are performing a **64-bit by 64-bit division** to produce a **64-bit quotient**, you must use **64 iterations**.

The number of iterations in the shift-and-subtract algorithm must match the number of bits in the **quotient** (the result) you are trying to calculate.

Why 64-bit Division uses 64 Shifts

In a 64/64 division, you are essentially asking, "How many times does this 64-bit number fit into that 64-bit number?"

1. The result could potentially be a very large 64-bit number.
 2. To find every bit of that 64-bit result, you have to shift the dividend through the comparison "window" 64 times.
 3. If you only did 32 shifts, you would only discover the top 32 bits of the answer, and the rest would be lost.
-

When would you need this for Fixed Point?

You would only use a 64-bit loop in fixed-point math if you were moving to a **Q32.32** format (64-bit total size).

- **Dividend:** 64-bit (or even 128-bit intermediate).
- **Divisor:** 64-bit.
- **Loop:** 64 iterations.

[AI Mode](#)[All](#)[Images](#)[Videos](#)[Short videos](#)[More](#)[Sign in](#)

every shift and every subtraction has to be performed across multiple registers (e.g., `a3:a2:a1:a0`). This is why **Q16.16** is the "sweet spot" for 32-bit microcontrollers—it provides good precision while keeping the math loops short (32 bits).

