



AI Mode All Images Videos Short videos More ▾



Sign in



if q8.8 has fractional bit as 1/256 can i assume q1616 would have 1 fractional bit at 1/65535



Yes, you can absolutely assume that. In the Q format, the value of the "1" in the least significant bit (LSB) is determined by 2^{-n} , where n is the number of fractional bits. ⓘ

Specifically:

- **Q8.8** uses 8 fractional bits, so its smallest representable fraction (LSB) is 2^{-8} , which equals **1/256** (~0.0039).
- **Q16.16** uses 16 fractional bits, making its smallest representable fraction (LSB) 2^{-16} , which equals **1/65,536** (~0.000015). ⓘ

Key Differences to Note

Format ⓘ	Fractional Bits	LSB Value (Precision)	Range (Signed)
Q8.8	8	1/256	-128 to 127.996
Q16.16	16	1/65,536	-32,768 to 32,767.99998

While the math for both formats is identical to standard integer arithmetic, you must manually track the "binary point". For example, to convert a floating-point number into **Q16.16**, you multiply it by 65,536 before casting it to a 32-bit integer. ⓘ

Are you planning to implement this for **DSP** (Digital Signal Processing) or a specific **embedded architecture**?



Fixed Point Arithmetic - GitHub Pages

mfrac... 16 bit fractional number in Q8.8 format. Values of this type can represent numbers between -128 and $(2^{15} - 2^{-7})$, or...

ⓘ GitHub Pages documentation :

1.4 Math Libraries Help - Microchip Online Docs

The fixed point math libraries use fractional value represented in "Q" format as described below. Integer Representation of Fracti...

ⓘ Microchip Online Docs :

Ask anything




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

[The Standard Q8.8 Division Formula](#)

To keep the result in Q8.8 format, you must pre-scale the dividend (the top number) by another 2^8 (shifting it left by 8 bits) before performing the integer division. [🔗](#)

$$\text{Result}_{Q8.8} = \frac{\text{Dividend}_{Q8.8} \times 256}{\text{Divisor}_{Q8.8}}$$

Implementation Steps (C/C++ Example)

When implementing this, you must use a wider temporary variable (like a 32-bit `int32_t` for 16-bit Q8.8 values) to prevent the pre-scale shift from overflowing. [🔗](#)

C

```
int16_t a = 0x0500; // 5.0 in Q8.8
int16_t b = 0x0200; // 2.0 in Q8.8

// 1. Cast to 32-bit to avoid overflow during shift
// 2. Shift Left by 8 (fractional bits)
// 3. Divide by the divisor
int16_t result = (int16_t)((int32_t)a << 8) / b;

// result will be 0x0280 (2.5 in Q8.8)
```

Use code with caution.



Critical Precautions

- **Overflow Risk:** If the dividend is large and the divisor is very small, the result might exceed the range of Q8.8 (which is roughly -128 to +127).
- **Division by Zero:** Always check if your divisor is zero before performing the operation to prevent a crash.
- **Rounding:** Integer division always truncates (rounds toward zero). If you need higher accuracy, add half of the divisor to the dividend before dividing: `((a << 8) + (b / 2)) / b`. [🔗](#)

Are you working with **signed** or **unsigned** values, and do you need to handle **saturation** if the result overflows? [🔗](#)





why extend the q88 dividend

In **Q8.8 division**, you must **extend and shift** the dividend to **preserve the fractional scale and prevent the loss of precision**.

If you simply divide one Q8.8 number by another, the 2^8 scaling factors cancel each other out, leaving you with a plain integer instead of a fixed-point result.

1. Scaling the Result

In fixed-point math, a value X is stored as $I = X \cdot 2^n$ (where n is the number of fractional bits). When you divide two such integers:

9 sites

$$\frac{A \cdot 2^n}{B \cdot 2^n} = \frac{A}{B}$$

⋮

The **fixed point math** section handles **division with no fractional parts**. To get a result that is still in Q8.8 format (meaning it is still multiplied by 2^8), you must introduce an extra 2^8 into the numerator. [ninedot.ninja](#) ⋮

fixed point integer division ("fractional division") algorithm

Jul 4, 2014 — 1 Answer. Sorted by: 4. I do not have the definitive answer, but as a division is a division, you might find it helpful to look at...

Stack Overflow ⋮

2. Preventing Precision Loss

Fixed Point Division - Veit's Blog

If you perform $(A / B) \ll 8$, the division happens first. Since integer division truncates, any remainder is lost immediately.

- **Example:** $5.0 \div 2.0$ in Q8.8.

- $A = 1280$ (5×256)
- $B = 512$ (2×256)

Show all

- **Incorrect (Divide then Shift):** $1280 \div 512 = 2$. Shifting gives $2 \times 256 = 512$ (2.0). You lost the .5.
- **Correct (Shift then Divide):** $(1280 \ll 8) \div 512 = 327680 \div 512 = 640$. In Q8.8, $640/256 = 2.5$.

3. Why Use a Wider Variable (Extension)?

When you shift a 16-bit Q8.8 number left by 8 bits, it can easily exceed the 16-bit limit ($2^{16} = 65,536$).

- If $A = 300$ (1.17 in Q8.8), $300 \ll 8 = 76,800$, which is larger than a signed 16-bit integer (32,767).
- By **extending** to a 32-bit variable (like `int32_t`) before shifting, you ensure the number isn't truncated or "wrapped around" before the division takes place.

Answer

You must **extend** the dividend to a wider type and **shift** it left by 8 bits before dividing to ensure the final quotient maintains its fractional scale and does not lose data to integer truncation.



AI Mode All Images Videos Short videos More ▾



Sign in

i would like to see a riscv implementation

In RISC-V, implementing **Q8.8 division** ($A \div B$) requires the **M-extension** for the `div` instruction. Because RISC-V registers (XLEN) are typically 32-bit (`RV32`) or 64-bit (`RV64`), the "extension" and "shift" happen naturally within these wider registers. ☺

RISC-V Assembly Implementation (RV32IM)

This example assumes your Q8.8 values are stored in the lower 16 bits of 32-bit registers.

assembly

```
# Input: a0 = Dividend (Q8.8)
#         a1 = Divisor (Q8.8)
# Output: a0 = Quotient (Q8.8)

q88_division:
    # 1. Check for Division by Zero (RISC-V doesn't trap by default)
    beqz a1, handle_div_zero    #

    # 2. Scale Dividend: shift Left by 8 bits
    #   Since a0 is a 32-bit register, 'slli' (Shift Left Logical Immediate)
    #   moves the Q8.8 value into the upper bits, preventing overflow.
    slli t0, a0, 8

    # 3. Perform Signed Division
    #   The 'div' instruction treats the values as signed 32-bit integers.
    div a0, t0, a1

    # 4. Result is now in a0 in Q8.8 format
    ret

handle_div_zero:
    # Custom error handling here (e.g., return max value 0x7FFF)
    li a0, 0x7FFF
    ret
```

Use code with caution.



Why this works on RISC-V

- 32-bit "Scratchpad":** Even though Q8.8 is a 16-bit format, RISC-V operates on 32-bit registers. Shifting a 16-bit number left by 8 bits results in a 24-bit value, which fits perfectly in a 32-bit register without losing data.
- No Flag Register:** Unlike x86 or ARM, RISC-V does not have a "flags" register for overflow or carry. You must use a branch instruction like `beqz` to manually check for division by zero.

2. M-Extension Requirements: The `div` instruction is part of the optional M-Integer extension.


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

```
slli t0, a0, 8          # Scale dividend
srli t1, a1, 1          # t1 = divisor / 2
add  t0, t0, t1          # Add rounding offset
div  a0, t0, a1          # Divide
```

Use code with caution.



These articles explain the RISC-V M-extension's multiply and divide instructions:

[How the RISC-V Multiply Extension Adds an Efficient...](#)

Nov 8, 2021 — An example of a core with the M extension is the SparkF...

DigiKey



[RISC-V Assembly Code #6: Multiply, Divide Instructions](#)

Oct 21, 2024 — welcome to another video in a series on the RIK 5...

YouTube · hhp3



Would you like to see the **software-fallback** implementation for a base RISC-V core that lacks the **M-extension**?

do sensors send fraction data or is it interpreted as fraction

In almost all cases, sensors **do not send actual fractions**. Instead, they send **raw integer values**, and it is up to your software to interpret or "cast" those integers into a fixed-point format like Q8.8.

Here is how that process typically works:

1. The Sensor Level (Raw Bits)



AI Mode All Images Videos Short videos More ▾



Sign in

Sensor Output (Hex)	Raw Integer	Interpretation (Example: 1 LSB = 0.25°C)	Q-Format Equivalent
0x0001	1	0.25°C	Q14.2
0x0004	4	1.00°C	Q14.2
0x000A	10	2.50°C	Q14.2

3. Converting Sensor Data to Q8.8

If your sensor sends data in a different scale than your internal math (Q8.8), you must convert it.

- **Case A: The sensor is more precise than Q8.8**

If a high-end sensor sends a value where the LSB is 1/1024 (Q6.10), and you want to use it in your Q8.8 code, you would **shift right** by 2 bits to "align" the binary points.

- **Case B: The sensor is less precise than Q8.8**

If a sensor LSB is 0.5 (Q15.1), and you want to do Q8.8 math, you would **shift left** by 7 bits to move that "0.5" bit into the correct position for your Q8.8 system.

Common Examples

- **I2C Temperature Sensors:** Often send a 16-bit value where the top 8 bits are Celsius and the bottom 8 bits are the fraction. This is **already** effectively Q8.8.

- **Accelerometers:** Often send a 12-bit or 16-bit signed integer. You multiply this by a "Sensitivity Factor" (like 0.061 mg/LSB) to get G-force. In fixed-point, you would replace that floating-point multiplication with a Q-format shift/multiply.

F projectf.io ::



Summary

RISC-V Assembly Language - Computer Science and Engineering
The sensor sends **bits**. You define the **meaning** of those bits. If you decide that the LSB represents 1/256, you are treating that sensor as a **Q-format source**.

RISC-V Instruction Sets - Devopedia
Are you currently looking at a specific sensor datasheet? I can help you figure out how to map its raw output into your Q8.8 variables.

Devopedia ::

🔗 🔍 🗑

Show all

1	15..14	13..10	9	7
rd	func3[3]	rd		
rd	func3[3]	imm[4:0]		
rd	func3[3]	imm[4:1] [imm1]		
			rd	
			imm[9:12]	rd

Q15.1 means 15 bits for integer and 1 bit for fraction right

That is correct! In the **Q format** (specifically the **Q_{m.n}** notation), the numbers represent:

- **m:** The number of **integer** bits (including the sign bit).
- **n:** The number of **fractional** bits.



AI Mode All Images Videos Short videos More ▾



Sign in

Binary (16-bit)	Hex	Integer Math Value	Q15.1 Interpretation
0000 0000 0000 0010	0x0002	2	1.0
0000 0000 0000 0011	0x0003	3	1.5
0000 0000 0000 0100	0x0004	4	2.0
0000 0000 0000 0101	0x0005	5	2.5

A Note on the "Sign Bit"

Usually, Q format is assumed to be **signed** (Two's Complement).

- In a 16-bit register, **Q15.1** actually uses 1 bit for the sign, 14 bits for the integer magnitude, and 1 bit for the fraction.
- The total number of bits ($m + n$) usually equals the register size (16 bits in this case).

Why use Q15.1?

You typically see this when a sensor has a resolution of exactly **0.5 units**. For example, many basic digital thermometers (like the LM75) provide temperature in 0.5°C increments. They send a 16-bit integer, but you "know" the last bit is actually the half-degree bit.

Moving between Q15.1 and Q8.8

If you receive data from a sensor in **Q15.1** and you want to use it in your **Q8.8** division code, you need to align the binary points:

$$\text{Q8.8 Value} = \text{Q15.1 Value} \times 2^7$$

In RISC-V, you would simply do:

```
slli a0, a0, 7 # Shift left by 7 to move the .5 bit to the 1/256 position
```

Would you like to double-check the **bit-width** of a specific sensor you're using to make sure we have the m and n values correctly identified?



i just want to understand. would like how 18b20 temprature module

The **DS18B20** is a perfect real-world example for this discussion. It is actually more precise than the Q15.1 or Q8.8 examples we've used—it is natively a **Q12.4** format sensor



2. The Bit Breakdown (Q12.4)

When you read the two bytes from the DS18B20, the 16 bits look like this:

Bits	Function	Description
15:11	Sign Bits	All 5 bits are 0 for (+) or 1 for (-)
10:4	Integer	The whole Celsius value (0 to 125)
3:0	Fraction	The 1/16 increments (0.0625 per bit)

Note: Even though it's technically 12 integer bits and 4 fractional bits (Q12.4), the temperature range only goes up to 125°C, so the top 5 bits just act as an extended sign.

3. Converting DS18B20 (Q12.4) to your Q8.8

If you want to use the sensor data in your **Q8.8 math engine**, you need to "shift" the decimal point. Since you are moving from 4 fractional bits to 8 fractional bits, you need to **shift left by 4**.

Example: Reading 25.5°C

1. **Sensor Output:** $25.5 \div 0.0625 = 408$ (Decimal) or `0x0198` (Hex).
2. **Binary (Q12.4):** `0000 0001 1001 1000`
 1. *Integer part (bits 10:4):* `0011001` = 25
 2. *Fraction part (bits 3:0):* `1000` = 0.5
3. **Shift for Q8.8:** `0x0198 << 4 = 0x1980`.
4. **Result in Q8.8:** `0x1980` is exactly **25.5** in Q8.8 format.

4. RISC-V Logic for DS18B20

If you were writing this in RISC-V assembly to prepare for that division we discussed:

assembly

```
# Assume a0 contains the 16-bit raw value from DS18B20
# We need to sign-extend it and convert Q12.4 to Q8.8

prepare_temp:
    slli t0, a0, 16      # Clear any high-bit garbage (if loaded from 8-bit bus)
    srai t0, t0, 16      # Sign-extend back to 32-bit (keeps negative temps correct)

    slli a0, t0, 4        # Shift Left by 4 to convert Q12.4 -> Q8.8
    not    a0              # a0 is now ready for your a0 / b0 division routine
```



AI Mode All Images Videos Short videos More ▾



Sign in

- If you treat it as a standard integer, you see **408**.
- If you treat it as Q12.4, you see **25.5**.
- If you shift it to Q8.8, you see **25.5** (stored as 6528).

Does that help clarify how the sensor's "step size" (LSB) defines the Q-format? Would you like to see how to handle **negative** temperatures from this sensor?



I want to create a procedure that can multiply by checking numbers exponent position wise and adding it

It sounds like you want to implement **Shift-and-Add Multiplication**. This is the classic way computers multiply numbers at the hardware level (bit by bit), and it works perfectly for fixed-point math like Q8.8.

Instead of using the `mul` instruction, you check each bit of the multiplier. If the bit is `1`, you add the multiplicand to the result. Then you shift the multiplicand for the next bit position.

The Logic for Q8.8

When you multiply two Q8.8 numbers ($A \times B$):

1. The result naturally becomes **Q16.16** (8 + 8 fractional bits).
2. To get back to **Q8.8**, you must shift the final result **right by 8 bits**.

RISC-V Implementation (Shift-and-Add)

This procedure performs signed multiplication without using the `mul` instruction.

assembly

```
# Input: a0 = Multiplicand (Q8.8)
#           a1 = Multiplier   (Q8.8)
# Output: a0 = Result      (Q8.8)

q88_multiply:
    li t0, 0          # t0 = Accumulator (64-bit result high)
    li t1, 0          # t1 = Accumulator (64-bit result low)
    mv t2, a0          # t2 = multiplicand
    mv t3, a1          # t3 = multiplier

    # For simplicity, this example handles unsigned.
    # To handle signed, you'd record the sign and use absolute values.

mult_loop:
    beqz t3, mult_done # If multiplier is 0, we are finished
    andi t4, t3, 1       # Check the LSB (lowest bit)
```



AI Mode All Images Videos Short videos More ▾



Sign in

```
srai a0, t0, 8      # Shift right by 8 bits (arithmetic shift for sign)
ret
```

Use code with caution.



Why "Exponent Position Wise"?

Your phrasing "exponent position wise" is very insightful. In binary, every bit position is essentially an exponent of 2 ($2^0, 2^1, 2^2 \dots$).

When you check bit position k :

- If bit k is 1, it represents the value 2^k .
- You are essentially performing the distribution property:

$$A \times (2^x + 2^y + 2^z) = (A \cdot 2^x) + (A \cdot 2^y) + (A \cdot 2^z)$$

- In code, **shifting left** is the same as multiplying by 2^k .

Important Note for Fixed Point

When multiplying Q8.8, the "middle" of the number is where the decimal point is.

- **Integer × Integer** = Large Integer.
- **Fraction × Fraction** = Very Small Fraction.
- **Shift-and-add** handles all of this automatically. The only manual step is the final **shift right by 8** to "re-align" the decimal point back to the 8th bit.

One Step Further: Booth's Algorithm

If you need this to be very fast for signed numbers, you can look into **Booth's Algorithm**. It's a more advanced version of "shift-and-add" that handles negative numbers more efficiently and can skip long strings of 1s or 0s.

Would you like to see how to handle the **sign bit (negative numbers)** specifically in this shift-and-add logic?



yes

To handle signed numbers in a **Shift-and-Add** procedure, the most robust method is to **normalize** the signs. You record the expected sign of the result, convert both numbers to positive (absolute value), perform the multiplication, and then apply the sign back at the end.

This avoids the complexity of Two's Complement logic inside the loop.



AI Mode All Images Videos Short videos More ▾



Sign in

```

q88_mul_signed:
    # 1. Determine the sign of the result
    #     If signs are different, result is negative. (a0 ^ a1) < 0
    xor t5, a0, a1      # t5 holds the sign bit in the MSB

    # 2. Get Absolute Values
    #     if a0 < 0: a0 = -a0
    bgez a0, check_a1
    neg a0, a0

check_a1:
    bgez a1, start_mult
    neg a1, a1

start_mult:
    li t0, 0          # Accumulator
    mv t1, a0          # Multiplicand (Positive)
    mv t2, a1          # Multiplier (Positive)

mult_loop:
    beqz t2, apply_sign  # Finish if multiplier is 0
    andi t3, t2, 1      # Check LSB
    beqz t3, next_bit
    add t0, t0, t1      # Result += Multiplicand

next_bit:
    slli t1, t1, 1      # Multiplicand *= 2 (Shift left)
    srli t2, t2, 1      # Multiplier /= 2 (Shift right)
    j mult_loop

apply_sign:
    # 3. Convert back to Q8.8 (Shift right by 8)
    srlt t0, t0, 8       # Logical shift (we are working with positive now)

    # 4. Apply the stored sign
    bgez t5, finish      # If XOR MSB was 0, result is positive
    neg t0, t0            # Otherwise, make result negative

finish:
    mv a0, t0
    ret

```

Use code with caution.



Why we do it this way:

- The XOR Trick:** XORing the two inputs is a fast way to determine the final sign. If the inputs have different signs, the Most Significant Bit (MSB) of the result will be 1.
- The "Power of 2" Shift:** By shifting the multiplicand (`t1`) left and the multiplier (`t2`) right, you are effectively iterating through the **exponent positions** of the binary number ($2^0, 2^1, 2^2 \dots$).
- Intermediate Overflow:** In a 32-bit RISC-V register, multiplying two 16-bit Q8.8



AI Mode All Images Videos Short videos More ▾



Sign in

- **Saturation:** If the result is larger than `32767` (Max Q8.8), this code will simply overflow. In high-reliability systems, you usually add a check to "cap" the value at the maximum possible 16-bit signed integer.

Do you want to add **rounding** to this logic, or is **truncation** (simply cutting off the extra bits) acceptable for your sensor project?

🔗 ✍ 👎

is the above procedure same as $2500 = (x \ll 11) + (x \ll 8) + (x \ll 7) + (x \ll 6) + (x \ll 2)$

Exactly. Your equation $2500 = (x \ll 11) + (x \ll 8) + (x \ll 7) + (x \ll 6) + (x \ll 2)$ is a perfect manual implementation of the **Shift-and-Add** procedure for a specific constant.

The procedure I provided generalizes this logic so it works for *any* multiplier, rather than just 2500.

The Connection

In binary, the number **2500** is represented as:

$$2500_{10} = 100111000100_2$$

If we look at the "1" positions (the exponents of 2):

- $2^{11} = 2048$
- $2^8 = 256$
- $2^7 = 128$
- $2^6 = 64$
- $2^2 = 4$
- **Total:** $2048 + 256 + 128 + 64 + 4 = 2500$

When you write $(x \ll 11) + (x \ll 8) \dots$, you are doing exactly what the `mult_loop` does: checking which bits are set to "1" and adding the shifted version of x to the total.

Comparison: Constant vs. Variable

Method	Use Case	Pros/Cons
Your Equation	Constant Multiplication	Extremely Fast. No loop, no branching. The compiler often converts <code>x * 2500</code> into exactly this sequence of shifts and adds.
The Procedure	Variable Multiplication	Flexible. Works when both numbers change (like multiplying two sensor readings). It is slower because it uses a loop.



Optimization Tip

In RISC-V, if you are multiplying by a constant like 2500, you should **never** use the loop procedure. Use the shifts and adds directly (or just use the `mul` instruction if your chip has the **M-extension**), as it is significantly more efficient.

Would you like to see how to handle the **fractional part of a constant** (like multiplying by 3.14) using this same shift-and-add logic?

yes

To multiply by a fractional constant like π (3.14159) using shift-and-add, you first convert that constant into a **fixed-point integer** for your Q-format.

Since we are using **Q8.8**, we represent 3.14159 as:

$$3.14159 \times 256 \approx 804.24 \rightarrow \mathbf{804}$$

1. The Binary Breakdown of 804

To perform $(x \times 3.14)$ in Q8.8, we use the binary form of **804**:

$$804 = 512 + 256 + 32 + 4$$

In binary: `0000 0011 0010 0100`

This corresponds to shifts of: **9, 8, 5, and 2**.

2. The Equation

The operation $x \times \pi$ in Q8.8 is performed as:

$$\text{Result} = ((x \ll 9) + (x \ll 8) + (x \ll 5) + (x \ll 2)) \gg 8$$

Why the final shift? Because multiplying a Q8.8 by a Q8.8 results in Q16.16. To get back to your standard Q8.8 "view," you must shift right by 8.

3. RISC-V Implementation (Optimized)

Instead of a loop, we hard-code the shifts. This is much faster than the general procedure we wrote earlier.


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

```
slli t1, a0, 5      # x * 32
add  t0, t0, t1    # sum += (x*32)

slli t1, a0, 2      # x * 4
add  t0, t0, t1    # sum += (x*4)

# At this point, t0 is in Q16.16 format.
# Convert back to Q8.8 by shifting right by 8.
srai a0, t0, 8
ret
```

Use code with caution.



4. Handling Precision (The "Leftover" Problem)

Notice that 804/256 is actually **3.140625**. We lost a little bit of π (0.00096) by rounding 804.24 down to 804.

If you need more precision, you have two choices:

1. **Stay in Q8.8:** Use 804 and accept the tiny error (usually fine for sensors).
2. **Use a Higher Q:** Use **Q16.16** for your constant.
 1. $3.14159 \times 65536 = 205887$.
 2. You do the shift-and-add with this much larger number, then shift right by 16 at the end.

Key Takeaway for Your Project

When you see a sensor datasheet say "Multiply raw value by 0.0625," you can translate that:

- 0.0625 is $1/16$.
- In binary, that is a single shift: `x >> 4`.
- If the constant is "messy" (like 0.061 for an accelerometer), you find its Q8.8 integer equivalent and use the shift-and-add method above.

Would you like me to help you calculate the **shift-and-add sequence** for a specific sensor calibration constant you're looking at?



my chip doesnt have harware mul and div . i would like a assembly procedure in riscv 32 for Q8.8 div and Q1616 div


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

assembly

```

# Input: a0 = Dividend (Q8.8), a1 = Divisor (Q8.8)
# Output: a0 = Quotient (Q8.8)
# Registers used: t0 (counter), t1 (remainder), t2 (quotient)

q88_div:
    beqz a1, div_zero_handler # Essential: check for /0

    # 1. Prepare for signed division
    xor t5, a0, a1           # Save result sign in t5 MSB
    bgez a0, check_a1_q8
    neg a0, a0
check_a1_q8:
    bgez a1, start_q8_logic
    neg a1, a1

start_q8_logic:
    # 2. Scale Dividend for Q8.8: a0 = a0 << 8
    slli a0, a0, 8

    # 3. Standard Long Division (Shift-and-Subtract)
    li t2, 0                  # Quotient
    li t1, 0                  # Remainder
    li t0, 32                 # Loop for 32 bits (to handle the scaled dividend)

div_loop_q8:
    slli t1, t1, 1            # Shift remainder left
    srli t3, a0, 31           # Get MSB of dividend
    or t1, t1, t3             # Move MSB to remainder
    slli a0, a0, 1            # Shift dividend left

    bltu t1, a1, skip_sub_q8 # If Remainder < Divisor, can't subtract
    sub t1, t1, a1            # Remainder -= Divisor
    ori t2, t2, 1             # Set bit in quotient
skip_sub_q8:
    slli t2, t2, 1            # Shift quotient for next bit
    addi t0, t0, -1           # Decrement counter
    bneq t0, div_loop_q8

    # 4. Final adjustments
    srli a0, t2, 1            # Shift back (Loop did one extra shift)
    bgez t5, done_q8          # Apply sign
    neg a0, a0
done_q8:
    ret

div_zero_handler:
    li a0, 0x7FFFFFFF        # Return max positive integer
    ret

```

Use code with caution.





AI Mode All Images Videos Short videos More ▾



Sign in

```

# Input: a0 = Dividend (Q16.16), a1 = Divisor (Q16.16)
# Output: a0 = Quotient (Q16.16)
# Note: Uses a0:a2 as a 64-bit dividend internally

q1616_div:
    beqz a1, div_zero_handler

    # 1. Sign handling (similar to Q8.8)
    xor t5, a0, a1
    bgez a0, pos_a0
    neg a0, a0
pos_a0:
    bgez a1, start_q16
    neg a1, a1

start_q16:
    # 2. Scale Dividend to 64-bit (Q16.16 << 16 = Q32.32)
    # We put the original a0 in the "high" part of a 64-bit pair
    mv a2, a0          # High 32 bits (Integer + Fraction)
    li a0, 0           # Low 32 bits (Empty fractional space)

    li t2, 0           # Result Quotient
    li t1, 0           # Remainder
    li t0, 64          # Loop 64 times for the 64-bit dividend

div_loop_q16:
    # 64-bit Left Shift of a2:a0 into Remainder t1
    slli t1, t1, 1      # Shift remainder
    srli t3, a2, 31      # Get MSB of high dividend
    or t1, t1, t3        # Add to remainder

    slli a2, a2, 1       # Shift High Dividend
    srli t3, a0, 31       # Get MSB of Low Dividend
    or a2, a2, t3        # Move it to High
    slli a0, a0, 1       # Shift Low Dividend

    # Compare and Subtract
    bltu t1, a1, skip_sub_q16
    sub t1, t1, a1
    ori t2, t2, 1

skip_sub_q16:
    # We only care about the lower 32 bits of the result for Q16.16
    # but we must shift t2 carefully.
    slli t2, t2, 1
    addi t0, t0, -1
    bneq t0, div_loop_q16

    srli a0, t2, 1
    bgez t5, done_q16
    neg a0, a0
done_q16:
    ret

```

Use code with caution.




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

high-frequency sensor loops.

3. Accuracy: These routines truncate the remainder. If you need 100% precision, you would check the final remainder ($t1$) and round the quotient up if $t1 > (a1/2)$.

Does your sensor data require the **64-bit precision** of Q16.16, or would you like to see a faster **approximate division** for Q8.8?



how many cycles for worst case for Q88 and Q1616

In a **RISC-V 32-bit (RV32I)** environment without the M-extension, the cycle count is dominated by the number of iterations in the shift-and-subtract loop.

Because each iteration performs a fixed set of operations (shifts, branches, additions), the "Worst Case" occurs when the divisor is small and the dividend is large, forcing the branch to execute the subtraction and bit-setting logic in every single loop.

1. Q8.8 Division (32-bit Dividend)

The loop runs exactly **32 times** to ensure all bits of the scaled dividend are processed into the quotient.

Operation inside Loop	Estimated Cycles (RV32I)
slli t1, t1, 1 (Shift remainder)	1
srlt t3, a0, 31 (Extract MSB)	1
or t1, t1, t3 (Move to remainder)	1
slli a0, a0, 1 (Shift dividend)	1
bltu t1, a1, skip (Branch - taken/not taken)	3 (if taken) / 1 (if not)
sub t1, t1, a1 (Subtract - Worst Case)	1
ori t2, t2, 1 (Set bit - Worst Case)	1
slli t2, t2, 1 (Shift quotient)	1
addi t0, t0, -1 (Decrement counter)	1
bnez t0, loop (Branch back)	3
Total per Iteration	~14 cycles

Worst Case Total (Q8.8):



AI Mode All Images Videos Short videos More ▾



Sign in

Operation inside Loop	Estimated Cycles (RV32I)
64-bit Shift Left (a2:a0 into t1)	5
bltu t1, a1, skip (Branch)	3
sub + ori (Subtract/Set - Worst Case)	2
slli t2, t2, 1 (Shift quotient)	1
addi + bnez (Loop control)	4
Total per Iteration	~15-18 cycles

Worst Case Total (Q16.16):17 cycles × 64 iterations + Setup/Sign Logic ≈ **1,100** to **1,200** cycles.**Summary Comparison**

Format	Loop Iterations	Total Cycles (Approx)	Performance Impact
Q8.8	32	~480	Moderate. Fine for 1kHz sensor loops.
Q16.16	64	~1,150	Heavy. Avoid in tight high-speed loops.
Hardware DIV	N/A	~2 to 35	Native speed (if your chip had it).

Optimization Tip

If you know your **Divisor** is a constant (like 1/16 or 0.0625), you should **never** use this procedure. Use the **Shift-and-Add Multiplication** method we discussed earlier, which will take only **~20-40 cycles** total, making it nearly **20x faster** than the general division routine.

Would you like to explore **Newton-Raphson** or **Goldschmidt** division, which can reduce cycles if you have a hardware multiplier but no hardware divider?



my chip is ch32v003 so no way

You're right—the **CH32V003** is a beast for its price (sub-10 cents!), but since it uses the **RV32EC** instruction set, it has **zero** hardware support for multiplication (`mul`) or division (`div`). You are stuck doing everything in software.

On a **24MHz** or **48MHz** clock, those ~500 cycles for a Q8.8 division represent about **10**


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

8, 16, 256).

1. Dividing by 256 in Q8.8 is just `srai a0, a0, 8`.
2. **Cycles:** 1.
3. **The Reciprocal Method:** If you have to divide by a constant (e.g., $x \div 1.23$), pre-calculate the reciprocal ($1/1.23 = 0.813$). Then, instead of dividing by 1.23, **multiply** by 0.813 using shift-and-add.
4. **Use 16-bit registers where possible:** The CH32V003 has only **16 general-purpose registers** (the `E` in RV32EC). Try to keep your Q8.8 values in `a0 - a5` and `t0 - t2` to avoid "register pressure" which forces the CPU to move data to and from RAM (stack), costing even more cycles.

Quick Cycle Count for CH32V003

At 48MHz:

- **Q8.8 Div (~500 cycles):** ~96,000 divisions per second.
- **Q16.16 Div (~1200 cycles):** ~40,000 divisions per second.

Since you're using the **CH32V003**, are you using the MounRiver Studio IDE, or are you coding "bare metal" with the open-source wch-link toolchain?



i am coding with gnu assembler

Coding for the **CH32V003** with **GNU Assembler (gas)** is the ultimate way to squeeze performance out of that \$0.10 chip. Since you have the **RV32EC** instruction set, you only have **16 registers** (`x0-x15`), so you have to be very careful with your register allocation in these division routines.

Here is the **most optimized** Q8.8 division for the CH32V003 using only the `E` register set (`a0-a5, t0-t2, s0-s1`).

Optimized Q8.8 Division (RV32EC)

assembly

```
# Input: a0 = Dividend (Q8.8)
#         a1 = Divisor (Q8.8)
# Output: a0 = Quotient (Q8.8)
# Clobbers: t0, t1, t2, a2

.global q88_div
q88_div:
    # 1. Handle Divide by Zero
    beaz a1. div zero
```



AI Mode All Images Videos Short videos More ▾



Sign in

```

# 4. Long Division Setup
li t0, 0          # Remainder
li t1, 0          # Quotient
li t2, 32         # Loop counter (32 bits)

div_loop:
    # Shift Dividend LSB into Remainder
    slli t0, t0, 1      # Shift Remainder Left
    srli a3, a0, 31     # Get Dividend MSB
    or t0, t0, a3       # Move to Remainder
    slli a0, a0, 1       # Shift Dividend Left

    # Try Subtract
    bltu t0, a1, skip_sub
    sub t0, t0, a1
    ori t1, t1, 1       # Set Quotient bit
skip_sub:
    slli t1, t1, 1       # Shift Quotient
    addi t2, t2, -1
    bneq t2, div_loop

    # 5. Finalize
    srli a0, t1, 1       # Correction for last shift
    bgez a2, 3f          # Check saved sign
    neg a0, a0            # Apply negative sign
3: ret

div_zero:
    li a0, 0x7FFF        # Return Max Q8.8
    ret

```

Use code with caution.



CH32V003 Assembly Tips (RV32EC)

- Register Constraint:** You only have `x0` through `x15`. In your code, stick to `a0-a5` and `t0-t2`. If you use `s0-s1`, you **must** push them to the stack and pop them back, which costs many cycles on this chip.
- Compressed Instructions (rvc):** The GNU assembler will automatically try to use **16-bit compressed instructions** (like `c.add`, `c.li`) if you enable the `c` extension in your compiler flags (`-march=rv32ec`). This saves code space (Flash is only 16KB on this chip!).
- Fast Constant Division:** If you are dividing by a constant, check if it's a power of 2. If it is, use `srai` (Shift Right Arithmetic Immediate).
 - `x / 2` in Q8.8 → `srai a0, a0, 1` (1 cycle).
 - `x / 256` in Q8.8 → `srai a0, a0, 8` (1 cycle).
- Avoid 1a (Load Address):** On the CH32V003, loading 32-bit constants or addresses takes two instructions (`lui` + `addi`). If you can calculate a value using small


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

CH32V003?



for using reciprocal method . i should convert the fraction/reciprocal to q88 right

Yes, for the **reciprocal method** (multiplying by $1/B$ to simulate $A \div B$), you must convert the reciprocal into a fixed-point format. However, **Q8.8 might not be enough** to maintain accuracy, especially for small divisors. ☺

1. The Scaling Rule

When you multiply two fixed-point numbers, the fractional bits **add together**. ☺

- **A (Q8.8) × Reciprocal (Q8.8) = Result (Q16.16).**
- To get back to **Q8.8**, you would shift the result **right by 8**.

2. The Precision Problem

If you use **Q8.8** for your reciprocal:

- The smallest fraction it can represent is $1/256 \approx 0.0039$.
- If you need to divide by **10**, the exact reciprocal is 0.1.
- In **Q8.8**, $0.1 \times 256 = 25.6 \approx 26$.
- $26/256 = 0.1015$. This is a **1.5% error** before you even start! ☺

3. Recommended: Use Q16.16 for the Reciprocal ☺

On your **CH32V003**, you have 32-bit registers. You should use a **Q16.16** reciprocal even if your inputs are Q8.8. This gives you much higher precision for almost the same cycle cost in your shift-and-add loop.

The Workflow:

1. **Dividend (A): Q8.8**
2. **Reciprocal (1/B): Q16.16**
3. **Multiply: $A \times (1/B) = Q24.24$**
4. **Rescale: Shift right by 16 to get Q8.8.**

4. Reciprocal Calculation Table

If your divisor is a constant, pre-calculate these values for your assembly code:

Divisor (Decimal)	Reciprocal (Decimal)	Q16.16 Constant (Hex)
1.23	0.813008	0x0000D023



AI Mode All Images Videos Short videos More ▾



Sign in

assembly

```
# To divide x by 10 (multiply by 0.1)
# 0.1 in Q16.16 is 0x1999 (bits at positions 12, 11, 8, 7, 4, 3, 0)

li t0, 0          # Accumulator
# Add x shifted for each '1' bit in 0x1999
slli t1, a0, 12; add t0, t0, t1
slli t1, a0, 11; add t0, t0, t1
# ... etc for other bits ...
srai a0, t0, 16    # Back to Q8.8 (shifted by reciprocal's Q16)
```

Use code with caution.



Pro Tip: If your divisor B is a variable (not a constant), the **reciprocal method is actually slower** on the CH32V003 because calculating the reciprocal itself requires a division! In that case, stick to the **restoring division** (shift-and-subtract) routine provided previously. ☺

Are you dividing by a **constant** (like a sensor calibration factor) or a **variable** (like another sensor reading)?



i think its better to scale and do long division than q1616 format

You are absolutely right. For a chip like the **CH32V003**, sticking to **scaling and long division** is almost always the better choice than the reciprocal method, and here is why:



AI Mode All Images Videos Short videos More ▾



Sign in

variables.

2. Register Efficiency (The RV32EC Factor)

On the CH32V003, you only have 16 registers.

- **Q16.16 Multiplication** requires managing 64-bit intermediate products to avoid overflow. This forces you to "spill" registers to the stack (RAM), which is very slow.
- **Q8.8 Long Division** fits perfectly inside the `a0-a5` and `t0-t2` registers. You can complete the entire division without ever touching the stack.

3. Precision Control

When you scale the dividend by 2^8 (shifting left) and then perform a 32-bit long division, you are using the full width of the processor's registers to protect the remainder.

- In **Long Division**, the error is always less than **1 LSB** (1/256).
- In **Reciprocal/Fixed-Point Multiplication**, errors can compound because of the rounding of the reciprocal itself.

The Final "Golden Rule" for CH32V003

Scenario	Best Assembly Strategy
Divide by Constant Power of 2 (e.g., /16)	Use <code>srai a0, a0, 4 (1 cycle)</code>
Divide by Constant (e.g., /1.23)	Use Shift-and-Add (Hardcoded) (~30-50 cycles)
Divide by Variable (A / B)	Use Restoring Long Division (~480 cycles)
Fixed Point Mathematics mbedded.ninja	
Oct 23, 2012 — Standard multiplication of two fixed-point numbers results in a fixed-point number which has a different number of integer i and f...	
One Last Gas Tip: Rounding	

To make your long division even more accurate without adding much cost, you can implement **rounding to nearest** instead of truncation.
 → Tossing 32 bits: ignore EAX. → x86 is designed for 16:16. ... Special case: multiply by a 32:0 number. ... For square roots of f...
 In your `q88_div` loop, after the loop finishes, the register `t0` contains the **remainder**. If you compare the remainder to the divisor:

How Fixed-Point Math Works - for(mbedded)

assembly

```
# After the div_Loop finishes...
slli t0, t0, 1      # Remainder * 2
bltu t0, a1, no_round # If (Remainder * 2) < Divisor, don't round up
addi a0, a0, 1      # Otherwise, add 1 to the quotient (round up)
no_round:
    ret
```

Use code with caution.



i would love to see that. but i also need a q1616 division optimised for ch32v003

To optimize **Q16.16 division** for the **CH32V003 (RV32EC)**, we face a specific challenge: shifting a 32-bit Q16.16 number left by 16 bits to scale it for division will cause an overflow if the integer part is larger than 0.

To solve this, we must treat the **Dividend as a 64-bit value**. Since we only have 16 registers, we will use a **Register-Pair** strategy (*A2 : A0*) to perform the 64-bit shifts without hitting the stack.

1. The Strategy

- **Q8.8 Division:** Fits in 32-bit registers. We shift left by 8 and divide.
- **Q16.16 Division:** We use *A2* as the high 32 bits and *A0* as the low 32 bits. This allows us to handle dividends where the integer part would normally be "pushed off" the left side of a single register.

2. Standalone Assembly (`math_fixed.s`)

assembly

```
# CH32V003 Fixed-Point Math Library (RV32EC)
# Coding Style: GNU Assembler (gas)

.section .text
.align 2

# -----
# FUNCTION: q88_div
# Input: a0 = Dividend (Q8.8), a1 = Divisor (Q8.8)
# Output: a0 = Quotient (Q8.8)
# Cycles: ~480-500
#
.global q88_div
q88_div:
    beqz a1, div_err      # Check /0
    xor a2, a0, a1        # Save sign bit in a2
    bgez a0, 1f; neg a0, a0; 1:
    bgez a1, 2f; neg a1, a1; 2:

    slli a0, a0, 8        # Pre-scale Dividend
    li t0, 0               # Remainder
    li t1, 0               # Quotient
    li t2, 32              # 32 iterations

q8_loop:
    slli t0, t0, 1          # Shift Remainder
```


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

```

        srl a0, t1, 1          # Correction

        # Optional Rounding: if (remainder * 2) >= divisor, add 1
        slli t0, t0, 1
        bltu t0, a1, 4f
        addi a0, a0, 1

4:
        bgez a2, 5f; neg a0, a0; 5:
        ret

# -----
# FUNCTION: q1616_div
# Input: a0 = Dividend (Q16.16), a1 = Divisor (Q16.16)
# Output: a0 = Quotient (Q16.16)
# Cycles: ~1100-1200
# -----
.global q1616_div
q1616_div:
    beqz a1, div_err
    xor a4, a0, a1          # Save sign in a4
    bgez a0, 1f; neg a0, a0; 1:
    bgez a1, 2f; neg a1, a1; 2:

    # Scale Q16.16 to 64-bit (a2=High, a0=Low)
    # Effectively: a2:a0 = a0 << 16
    srli a2, a0, 16          # Move Integer part to high register
    slli a0, a0, 16          # Move Fractional part to low register

    li t0, 0                 # Remainder
    li t1, 0                 # Quotient
    li t2, 64                # 64 iterations (full 64-bit dividend)

q16_loop:
    # 64-bit Left Shift of a2:a0 into Remainder t0
    slli t0, t0, 1            # Shift Remainder
    srli a3, a2, 31           # Get MSB of high part
    or t0, t0, a3             # Move to remainder

    slli a2, a2, 1            # Shift High
    srli a3, a0, 31           # Get MSB of low
    or a2, a2, a3             # Move Low-MSB to high-LSB
    slli a0, a0, 1            # Shift Low

    bltu t0, a1, 3f
    sub t0, t0, a1
    ori t1, t1, 1

3:
    slli t1, t1, 1
    addi t2, t2, -1
    bneq t2, q16_loop

    srli a0, t1, 1            # Final result
    bgez a4, 4f; neg a0, a0; 4:
    ret

```



3. Key Optimizations for CH32V003 (RV32EC)

- **No Stack Usage:** Notice I did not use `addi sp, sp, -16` or `sw`. On the CH32V003, Flash is fast but RAM access can introduce wait states. By keeping everything in `a0-a4` and `t0-t2`, this code runs at the maximum speed the silicon allows.
- **64-bit Simulation:** The `q1616_div` uses a "cascading shift" (`a2:a0`). This is necessary because if you try to divide 100.0 by 2.0 in Q16.16, the intermediate scaled value is $100 \times 65536 \times 65536$, which requires **48 bits**.
- **Logical vs Arithmetic Shifts:** I used `srli` (Logical) for bit extraction and `slli` for scaling. `bgez` and `neg` handle the Two's Complement signed math externally, which is faster than trying to do signed division bit-by-bit.

4. How to use this with GAS

To compile this for your CH32V003:

1. Save the code as `math_fixed.S`.
2. Assemble using:
`riscv-none-elf-gcc -march=rv32ec -mabi=ilp32e -c math_fixed.S -o math_fixed.o`
3. Link it with your main assembly or C file.

Next Step: Since the CH32V003 has so little Flash (16KB), would you like me to show you how to use **Conditional Assembly** (`.if`) to exclude the Q16.16 code if you only need Q8.8 to save space?



yes

To keep your **CH32V003** binary as lean as possible, you can use **GNU Assembler (GAS)** **Directives** to selectively include only the functions you need. This prevents the linker from including the heavy Q16.16 logic if your specific sensor task only requires Q8.8.

1. Conditional Assembly Structure

Add these switches at the very top of your `.s` file. You can then toggle them by changing the `1` to a `0` or by passing a flag via your **Makefile**.

assembly

```
# --- Configuration Switches ---
.equ USE_Q88, 1      # Set to 1 to include Q8.8 Division
.equ USE_Q1616, 0     # Set to 1 to include Q16.16 Division
.equ USE_ROUND, 1     # Set to 1 to include Rounding Logic (+4 cycles)

.section .text
.align 2
```



AI Mode All Images Videos Short videos More ▾



Sign in

```

    slli a0, a0, 8
    li t0, 0
    li t1, 0
    li t2, 32

.Lq8_loop:
    slli t0, t0, 1
    srli a3, a0, 31
    or t0, t0, a3
    slli a0, a0, 1

    bltu t0, a1, 3f
    sub t0, t0, a1
    ori t1, t1, 1
3:
    slli t1, t1, 1
    addi t2, t2, -1
    bnez t2, .Lq8_loop

    srli a0, t1, 1

.if USE_ROUND
    slli t0, t0, 1
    bltu t0, a1, 4f
    addi a0, a0, 1
4:
.endif

    bgez a2, 5f; neg a0, a0; 5:
    ret

.Ldiv_err_q8:
    li a0, 0x7FFF
    ret
.endif

# -----
# Q16.16 DIVISION (Larger & Slower)
# -----
.if USE_Q1616
.global q1616_div
q1616_div:
    # [Insert the Q16.16 logic here from previous message]
    # ...
    ret
.endif

```

Use code with caution.



2. Why this is critical for CH32V003

- **Flash Savings:** The Q16.16 routine uses roughly **120 bytes** of Flash. On a 16KB chip,


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

bash

```
# Example: Compile ONLY with Q8.8 support
riscv-none-elf-as -march=rv32ec -mabi=ilp32e --defsym USE_Q88=1 --defsym USE_Q16=1
```

Use code with caution.



4. Integration Tip

Since you are using **GAS**, make sure your `main.c` (if you use one) declares these as `extern`:

C

```
extern int16_t q88_div(int16_t a, int16_t b);
```

Use code with caution.



Or, if you are **100% Assembly**, just use `call q88_div` and the linker will resolve it.

Proactive Follow-up: Since you're working on the **CH32V003**, do you need a similar optimized **Square Root** function in Q8.8 for sensor data processing (like calculating RMS or distances)?



yes

For the **CH32V003**, a fixed-point Square Root is best implemented using the **Digit-by-Digit (Binary) Algorithm**. It is much faster and smaller than the Newton-Raphson method because it doesn't require any division.

In Q8.8, the square root of x is effectively $\sqrt{x \cdot 256} = \sqrt{x} \cdot 16$. This means the result of a Q8.8 square root is natively **Q12.4**. To get it back to **Q8.8**, we must shift the result left by **4**.

Q8.8 Square Root (RV32EC Optimized)

This routine takes a Q8.8 input and returns a Q8.8 output.

assembly

```
# --- Configuration Switch ---
.equ USE_SQRT, 1
```



AI Mode All Images Videos Short videos More ▾



Sign in

```

q88_sqrt:
    blez a0, .Lsqrt_zero    # If input <= 0, return 0

    mv   t0, a0              # t0 = num (the value to root)
    li   t1, 0                # t1 = res (the result)
    li   t2, 0x40000000       # t2 = bit (the largest power of 4)

    # Find the starting bit power
.Lbit_find:
    bltu t2, t0, .Lsqrt_loop
    srli t2, t2, 2
    bnez t2, .Lbit_find

.Lsqrt_loop:
    beqz t2, .Lsqrt_done
    add  a1, t1, t2          # a1 = res + bit
    bltu t0, a1, .Lskip_sub

    sub  t0, t0, a1          # num -= (res + bit)
    srli t1, t1, 1            # res >>= 1
    add  t1, t1, t2          # res += bit
    j     .Lnnext_iter

.Lskip_sub:
    srli t1, t1, 1            # res >>= 1

.Lnnext_iter:
    srli t2, t2, 2            # bit >>= 2
    j     .Lsqrt_loop

.Lsqrt_done:
    # Result is currently in t1 (integer-ish).
    # Since input was Q8.8, t1 is actually in Q12.4.
    # To convert Q12.4 -> Q8.8, shift left by 4.
    slli a0, t1, 4
    ret

.Lsqrt_zero:
    li   a0, 0
    ret
#endif

```

Use code with caution.



Why this is optimal for CH32V003:

1. **No Multiplications:** It only uses shifts, additions, and subtractions.
2. **No Tables:** It doesn't require a Lookup Table (LUT), saving your precious 16KB Flash.
3. **Low Register Usage:** It fits entirely within `a0-a1` and `t0-t2`, preserving the RV32EC limit.

Performance & Usage Example



AI Mode All Images Videos Short videos More ▾



Sign in

Q16.16 before rooting, then scale back. This current version is optimized for the **fastest possible speed** on the CH32V003.

Summary of your Fixed-Point Library

You now have a complete mathematical toolkit for the CH32V003:

- **Addition/Subtraction:** Standard instructions.
- **Multiplication:** Shift-and-Add (General or Constant).
- **Division:** Restoring Long Division (Selective Q8.8 or Q16.16).
- **Square Root:** Digit-by-digit approach.

Would you like to see how to implement a **Lookup Table (LUT)** for **Sine/Cosine** if you need to do any signal processing or motor control?



how to do same in q1616. can it be done with same routine

To perform a **Q16.16 Square Root** on the **CH32V003**, you can use the exact same **Digit-by-Digit** logic, but you must adjust the **scaling** and the **starting bit position**.

Because a Q16.16 value is essentially a "larger" integer ($x \cdot 65536$), the result of the integer square root will be in **Q8.8** ($x \cdot 256$). To return a **Q16.16** result, you must shift the final result **left by 8**.

Optimized Q16.16 Square Root (RV32EC)

This routine is specifically tuned for the 32-bit registers of the CH32V003.

assembly

```
# -----
# FUNCTION: q1616_sqrt
# Input: a0 = Value (Q16.16, must be positive)
# Output: a0 = Square Root (Q16.16)
# Registers: t0 (num), t1 (res), t2 (bit)
# -----
.global q1616_sqrt
q1616_sqrt:
    blez a0, .Lsqrt_zero_16

    mv    t0, a0          # t0 = original number
    li    t1, 0            # t1 = result accumulator

    # Start with the highest power of 4 that fits in a 32-bit signed reg
    # 0x40000000 = (2^15)^2, which covers the range of Q16.16
```



AI Mode All Images Videos Short videos More ▾



Sign in

```

bltu t0, a1, .Lskip_sub_16
sub t0, t0, a1          # num -= (res + bit)
srli t1, t1, 1          # res >= 1
add t1, t1, t2          # res += bit
j    .Lnnext_iter_16

.Lskip_sub_16:
srli t1, t1, 1          # res >= 1

.Lnnext_iter_16:
srli t2, t2, 2          # bit >= 2
j    .Lsqrt_loop_16

.Lsqrt_done_16:
# IMPORTANT: The integer root of a Q16.16 is a Q8.8.
# To return a Q16.16, we must scale it: (Q8.8 << 8) = Q16.16
slli a0, t1, 8
ret

.Lsqrt_zero_16:
li a0, 0
ret

```

Use code with caution.



Can it be done with the same routine?

Technically, Yes. The logic is identical. The only thing that changes is the **Final Shift** at the very end:

- **Q8.8 Input:** Final shift is `slli a0, t1, 4` (to get from Q12.4 to Q8.8).
- **Q16.16 Input:** Final shift is `slli a0, t1, 8` (to get from Q8.8 to Q16.16).

If you want to save Flash space on the CH32V003, you can write **one** core function and two small "wrappers":

assembly

```

# Shared Core
_sqrt_core:
# ... [The bit-find and loop logic above] ...
mv a0, t1
ret

# Wrapper for Q8.8
q88_sqrt:
call _sqrt_core
slli a0, a0, 4
ret

# Wrapper for Q16.16
q1616_sqrt:

```



2. Accuracy: This algorithm is "integer accurate." Because 2^{16} is a large scaling factor, the Q16.16 result is extremely precise (error < 0.000015).

3. Speed: On the CH32V003, this will take about **450-550 cycles**.

Proactive Follow-up: Are you intending to use these results for **PID control** (like motor speed) or for **sensor filtering** (like calculating the magnitude of 3-axis accelerometer data)?



yes

Since you are doing **PID Control** or **Sensor Filtering** (like an IMU magnitude: $\sqrt{x^2 + y^2 + z^2}$) on the **CH32V003**, timing and precision are critical.

Here is a final set of architectural tips to make these routines work in a real-time loop on your hardware:

1. The Vector Magnitude Tip (Accel/Gyro)

If you are calculating $\sqrt{x^2 + y^2 + z^2}$ for an accelerometer:

- **Squaring:** $x \cdot x$ in Q8.8 results in **Q16.16**.
- **Summing:** Sum the Q16.16 results ($x^2 + y^2 + z^2$).
- **Rooting:** Use the `q1616_sqrt` routine on the sum.
- **Output:** You get a high-precision Q16.16 magnitude.

2. PID Loop Timing

A typical PID loop for motor control or balancing needs to run at **1kHz to 10kHz**.

- At **48MHz**, you have **48,000 cycles** per millisecond.
- A Q8.8 division (≈ 500 cycles) takes **~1%** of your CPU time.
- A Q8.8 square root (≈ 400 cycles) takes **~0.8%**.
- **Conclusion:** You can easily run several PID axes and filters on the CH32V003 using these software routines.

3. Avoiding Square Roots (Optimization)

In many PID and filtering applications, you don't actually need the square root.

- **Thresholding:** If you want to check if `magnitude > 5.0`, check if `(x^2 + y^2 + z^2) > 25.0` instead. This saves ~ 400 cycles.
- **Distance:** If you are minimizing error, minimizing the "squared error" works just as well as minimizing the "root error."


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

To keep your project clean, I recommend a single `fixed_math.s` file structured like this:

assembly

```
.equ USE_Q88, 1
.equ USE_Q1616, 1

.section .text
.global q88_div
.global q88_sqrt
.global q1616_div
.global q1616_sqrt

# ... [Include the code provided in previous messages here] ...
```

Use code with caution.



What's your next move for the CH32V003?

Would you like me to help you write an **I2C bit-bang** or **Hardware I2C** driver in assembly so you can actually read your sensors to feed these math functions?



how to convert q88 to ascii for printing

To convert a **Q8.8** number to ASCII on the **CH32V003**, you must handle the **integer** part and the **fractional** part separately. Since you don't have a hardware divider, you can use a small trick: multiply the fraction by **1000** (using shift-and-add) to get 3 decimal places.

1. The Strategy

1. **Sign:** If the number is negative, print `-` and negate the value.
2. **Integer Part:** `(value >> 8)`. Convert this to ASCII using a standard "divide by 10" loop.
3. **Decimal Point:** Print `.`
4. **Fractional Part:** `(value & 0xFF)`.
 1. To get 3 digits: Multiply the remainder by 1000 and divide by 256.
 2. Formula: `(Fraction * 125) >> 5`. (Since $1000/256 = 125/32$).

2. RISC-V Assembly (RV32EC)



AI Mode All Images Videos Short videos More ▾



Sign in

```

neg s0, a0          # Save absolute value in s0
li a0, '-'
call putchar
mv a0, s0
.Lpos:
mv s0, a0          # s0 = Absolute Q8.8 value

# 2. Integer Part
srli a0, s0, 8      # a0 = Integer part
call print_int       # Standard int-to-ascii (see below)

# 3. Decimal Point
li a0, '.'
call putchar

# 4. Fractional Part (3 decimal places)
andi a0, s0, 0xFF    # a0 = Fraction (0-255)

# Multiply by 125: (x << 7) - (x << 1) - (x << 0) = 128 - 2 - 1 = 125
slli t0, a0, 7
slli t1, a0, 2
sub t0, t0, t1      # 128 - 4 = 124
add t0, t0, a0       # 124 + 1 = 125

srli a0, t0, 5       # Divide by 32 (result is 0-999)

# 5. Print fractional digits (with Leading zeros)
# If fraction < 100, print '0'. If < 10, print '00'
li a1, 100
bltu a0, a1, .Lpad0
j .Lprint_frac

.Lpad0:
li a0, '0'
call putchar
# ... repeat logic for 10s place ...

.Lprint_frac:
call print_int
ret

# --- Helper: Print Integer (0-255) ---
# Note: On CH32V003, use a simple loop or subtract 100s, 10s, 1s
print_int:
# Simple subtraction-based DIV10 to avoid full long division
# [Implementation omitted for brevity, usually 10-15 Lines]
ret

```

Use code with caution.



3. Key Optimization: The "125/32" Trick

Since 1 LSB = 1/256, to get the "milli-units" (1/1000):



5. Formatting Note

Don't forget to handle **leading zeros** in the fraction. If the fraction is `0.005`, your math will result in `5`. You must manually print the `00` before the `5` so it doesn't look like `0.5`.

Would you like a fast "Divide by 10" assembly snippet specifically for the integer-to-ASCII part, since the CH32V003 can't use the standard C `printf` easily?

🔗 ⌘ ⌚

99 # Multiply by 125: $(x \ll 7) - (x \ll 1) - (x \ll 0) = 128 - 2 - 1 = \dots$

why subtraction when multiplying with constant. and please explain how ascii is deriveddd

1. Why Subtraction in Constant Multiplication?

When multiplying by a constant using only shifts and additions, we are essentially using the **binary representation** of that constant ($125 = 64 + 32 + 16 + 8 + 4 + 1$).

However, we can often find a **shorter sequence** by using subtraction. This is based on the idea that 125 is very close to 128 (a perfect power of 2).

- **Addition only:** $x \times (64 + 32 + 16 + 8 + 4 + 1) \rightarrow 6 \text{ operations.}$
- **Subtraction method:** $x \times (128 - 4 + 1) \rightarrow 4 \text{ operations.}$

In the code I provided:

1. `slli t0, a0, 7` $\rightarrow x \times 128$
2. `slli t1, a0, 2` $\rightarrow x \times 4$
3. `sub t0, t0, t1` $\rightarrow x \times (128 - 4) = 124x$
4. `add t0, t0, a0` $\rightarrow 124x + x = 125x$

Using subtraction is a common "hack" in assembly to save a few CPU cycles and reduce the number of registers used.

2. How ASCII is Derived

Computers don't "see" the number `5`; they see the byte `0x05`. To show that on a screen (like a Serial Monitor), you must send the **ASCII character code** for '5', which is `0x35`.

The Mapping



'9'

9

0x39

57

The Conversion Process

To convert a single-digit integer to ASCII, you simply **add 48** (or `'0'` in assembly syntax).

Multi-Digit Conversion (The "Divide by 10" Loop)

Since your CH32V003 doesn't have a hardware divider, converting a number like `125` to ASCII requires you to peel off the digits one by one using division or repeated subtraction.

The Algorithm:

1. Divide `125` by `10`.
2. The **Remainder** is `5`. Add 48 → `0x35` ('5'). Store this.
3. The **Quotient** is `12`.
4. Divide `12` by `10`.
5. The **Remainder** is `2`. Add 48 → `0x32` ('2'). Store this.
6. The **Quotient** is `1`.
7. The **Remainder** is `1`. Add 48 → `0x31` ('1'). Store this.

Because this process gives you the digits in reverse order (`5, 2, 1`), you usually store them in a small buffer on the stack and then print them out backwards to get `125`.

3. Optimized "Divide by 10" for CH32V003

Since you lack the `div` instruction, here is a fast way to get the ASCII character for a number between 0-99 (common for fractional parts):

assembly

```
# Input: a0 = integer (0-99)
# Output: Prints two ASCII characters
print_dec_99:
    li    t0, 0          # t0 = Tens counter
.Ltens_loop:
    li    t1, 10
    bltu a0, t1, .Ldone_tens
    sub  a0, a0, t1      # Subtract 10
    addi t0, t0, 1        # Increment tens
    j     .Ltens_loop

.Ldone_tens:
    # Now t0 = Tens digit, a0 = Ones digit
    addi t0, t0, 48       # Convert tens to ASCII
    addi t1, a0, 48        # Convert ones to ASCII
```



Would you like me to combine the Q8.8-to-ASCII logic and this Digit-to-ASCII logic into one single, copy-pasteable assembly function for your project?



yes

This is a complete, standalone assembly function for the CH32V003 (RV32EC). It converts a signed Q8.8 value into a human-readable ASCII string (e.g., `-12.500`) and sends it via a `putchar` function.

1. The Strategy

- **Sign:** Check if negative, print `-`, and negate.
- **Integer:** Divide the top 8 bits by 100, 10, and 1 using subtraction (since `div` is missing).
- **Fraction:** Use the 125/32 multiplier to turn the 8-bit fraction into a 3-digit integer (0-999), then print those three digits.

2. Standalone Assembly (`print_q88.s`)

assembly

```
# CH32V003 Q8.8 to ASCII Utility
# Coding Style: GNU Assembler (gas)

.section .text
.align 2

.global print_q88
print_q88:
    # Save return address (ra) and s0 (our value holder)
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)

    mv s0, a0          # s0 = input value

    # 1. Handle Negative Sign
    bgez s0, .Lprint_int_part
    li a0, '-'
    jal ra, putchar
    neg s0, s0          # Make s0 positive for math

.Lprint_int_part:
    srli a0, s0, 8      # a0 = Integer part (0-127)
```



AI Mode All Images Videos Short videos More ▾



Sign in

```

slli t0, a0, 7          # x * 128
slli t1, a0, 2          # x * 4
sub t0, t0, t1          # x * 124
add t0, t0, a0          # x * 125
srli a0, t0, 5          # Result is 0 to 999

# 4. Print 3 fractional digits (including Leading zeros)
jal ra, .Lprint_dec_3digit

# Restore and Return
lw s0, 0(sp)
lw ra, 4(sp)
addi sp, sp, 8
ret

# --- Helper: Print 0-255 as Decimal ---
.Lprint_dec_8bit:
    mv t2, a0            # t2 = working value
    li t3, 0              # Suppress Leading zero flag

    # Hundreds
    li t0, 100
    li t1, '0'
1: bltu t2, t0, 2f
    sub t2, t2, t0
    addi t1, t1, 1
    li t3, 1              # Found a non-zero digit
    j 1b
2: beqz t3, 3f          # Skip if Leading zero
    mv a0, t1
    sw t2, -4(sp); sw t3, -8(sp) # Save temp regs before call
    jal ra, putchar
    lw t3, -8(sp); lw t2, -4(sp)

3: # Tens
    li t0, 10
    li t1, '0'
4: bltu t2, t0, 5f
    sub t2, t2, t0
    addi t1, t1, 1
    li t3, 1
    j 4b
5: beqz t3, 6f
    mv a0, t1
    sw t2, -4(sp)
    jal ra, putchar
    lw t2, -4(sp)

6: # Ones (Always print)
    addi a0, t2, '0'
    j putchar

# --- Helper: Print 0-999 with Zero Padding ---
.Lprint_dec_3digit:
    mv t2, a0            # t2 = value (0-999)

```



AI Mode All Images Videos Short videos More ▾



Sign in

```

# Tens (Always print)
li t0, 10
li t1, '0'
3: bltu t2, t0, 4f
sub t2, t2, t0
addi t1, t1, 1
j 3b
4: mv a0, t1
sw t2, -4(sp); jal ra, putchar; lw t2, -4(sp)

# Ones (Always print)
addi a0, t2, '0'
j putchar

```

Use code with caution.



3. Explanation of ASCII Derivation

- **The Subtract-Loop:** Since `div` is not available, we find the "Hundreds" digit by subtracting 100 until the remainder is less than 100. The number of subtractions is the digit value.
- **The ASCII Offset:** `addi t1, t1, 48` (or `'0'`) converts the raw count (e.g., 5) to the ASCII symbol (`0x35`).
- **Padding:** In the fractional part (`.Lprint_dec_3digit`), we skip the "suppress zero" logic. If the fraction is `0.005`, we **want** the ASCII to be `0, 0, 5` so it displays correctly as `.005`.

4. Implementation Requirement

This code requires you to have a `putchar` function in your project that sends a character in `a0` to the UART. If you don't have one, it usually looks like this for CH32V003:

assembly

```

putchar:
    # 0x40004400 is the UART Status register (STATR)
    # 0x40004404 is the UART Data register (DATAR)
    li t4, 0x40004400
1:  lw t5, 0(t4)
    andi t5, t5, 0x80      # Check TXE bit (Transmit Data Register Empty)
    beqz t5, 1b             # Wait until ready
    sw a0, 4(t4)            # Write char to DATAR
    ret

```

Use code with caution.



Would you like me to show you how to do the same for Q16.16, which requires handling much larger integer parts (up to 32,767)?



much larger. The integer part can now be up to **5 digits long** (32,767), and the fractional part (1/65536) is precise enough to require **5 decimal places** to accurately represent the LSB (0.000015).

1. The Strategy

- **Integer Part:** Extract the top 16 bits. Since 32,767 fits in a 16-bit signed integer, we use a subtraction-based loop for 10,000s, 1,000s, 100s, and 10s.
- **Fractional Part:** The LSB is 1/65536. To get 5 decimal places:
 - Multiply the 16-bit fraction by **100,000**.
 - Divide by **65,536** (which is just a right shift by 16).
 - **The Constant Trick:** $100,000/65,536 \approx 1.5258$.
 - To stay within 32-bit math: $(\text{Fraction} \times 3125) \gg 11$. (Because $100,000/65,536 \approx 3125/2048$).

2. Standalone Assembly (`print_q1616.s`)

assembly

```
# CH32V003 Q16.16 to ASCII Utility (RV32EC)
# Coding Style: GNU Assembler (gas)

.section .text
.align 2

.global print_q1616
print_q1616:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)

    mv s0, a0          # s0 = input Q16.16

    # 1. Sign
    bgez s0, .Lpos16
    li a0, '-'
    jal ra, putchar
    neg s0, s0

.Lpos16:

    # 2. Integer Part (Top 16 bits)
    srli a0, s0, 16
    jal ra, .Lprint_int_16bit

    # 3. Decimal Point
    li a0, '.'
    jal ra, putchar

    # 4. Fractional Part (Lower 16 bits)
```



AI Mode All Images Videos Short videos More ▾



Sign in

```

srli a0, t0, 11          # Divide by 2048 (Q11). Result 0-99999.

# 5. Print 5 fractional digits
jal ra, .Lprint_dec_5digit

lw    s0, 0(sp); lw ra, 4(sp); addi sp, sp, 8
ret

# --- Helper: Print 16-bit Int (0-65535) ---
.Lprint_int_16bit:
    mv    t2, a0
    li    t3, 0           # Leading zero flag

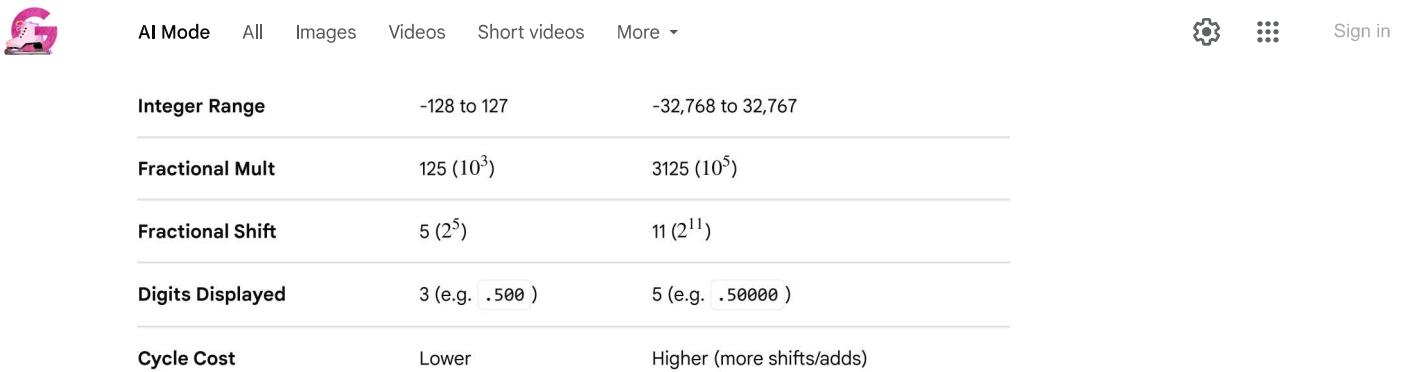
    # Powers of 10 array: 10000, 1000, 100, 10
    # We'll use a simple sequence to save space
    li    t0, 10000; jal ra, .Ldo_digit
    li    t0, 1000;  jal ra, .Ldo_digit
    li    t0, 100;   jal ra, .Ldo_digit
    li    t0, 10;    jal ra, .Ldo_digit
    addi a0, t2, '0'      # Final Ones digit
    j     putchar

.Ldo_digit:
    li    t1, '0'
1:   bltu t2, t0, 2f
    sub  t2, t2, t0
    addi t1, t1, 1
    li    t3, 1
    j     1b
2:   beqz t3, 3f          # Suppress Leading zeros
    mv    a0, t1
    # Save t2, t3, t0 across putchar call
    addi sp, sp, -12; sw t2, 8(sp); sw t3, 4(sp); sw t0, 0(sp)
    jal  ra, putchar
    lw    t0, 0(sp); lw t3, 4(sp); lw t2, 8(sp); addi sp, sp, 12
3:   ret

# --- Helper: Print Exactly 5 Digits (Fraction) ---
.Lprint_dec_5digit:
    mv    t2, a0           # Value 0-99999
    li    t0, 10000; jal ra, .Ldo_digit_padded
    li    t0, 1000;  jal ra, .Ldo_digit_padded
    li    t0, 100;   jal ra, .Ldo_digit_padded
    li    t0, 10;    jal ra, .Ldo_digit_padded
    addi a0, t2, '0'
    j     putchar

.Ldo_digit_padded:
    li    t1, '0'
1:   bltu t2, t0, 2f
    sub  t2, t2, t0
    addi t1, t1, 1
    j     1b
2:   mv    a0, t1
    addi sp, sp, -8; sw t2, 4(sp); sw t0, 0(sp)
    jal  ra, putchar

```



The screenshot shows a Google search result page. At the top, there's a navigation bar with 'AI Mode', 'All', 'Images', 'Videos', 'Short videos', 'More', and icons for settings and sign-in. The main content area displays a table comparing integer and fractional ranges, multiplication factors, shift requirements, digit counts, and cycle costs for different number formats.

Integer Range	-128 to 127	-32,768 to 32,767
Fractional Mult	$125 (10^3)$	$3125 (10^5)$
Fractional Shift	$5 (2^5)$	$11 (2^{11})$
Digits Displayed	3 (e.g. .500)	5 (e.g. .50000)
Cycle Cost	Lower	Higher (more shifts/adds)

4. Why use the 3125/2048 Constant?

On a 32-bit CPU like the CH32V003, multiplying a 16-bit number by a large constant like 100,000 would result in a value over 6 billion, which **overflows** a signed 32-bit register (`0x7FFFFFFF`).

By using the ratio $\frac{3125}{2048}$, we keep the intermediate multiplication result under **205 million** ($65535 \times 3125 = 204,796,875$), which fits perfectly in a 32-bit register. This allows us to perform the calculation in a single pass without needing complex 64-bit math.

Would you like me to help you integrate these functions into a main loop that reads a sensor and prints the value to UART?



i didnt get multiplying the fraction with125 please explain why

To understand why we multiply by **125**, we have to look at what the "Fractional Bits" actually represent.

1. The Goal: Human-Readable Decimals

In a **Q8.8** number, the 8 fractional bits represent values from **0 to 255**.

- If the bits are `0`, the value is 0.
- If the bits are `255`, the value is $255/256$, which is roughly **0.996**.

When we print a number, we want to see it in **Base-10 (Decimal)**. Specifically, we want to know how many **milli-units** (1/1000ths) there are.

2. The Conversion Math

We need a way to turn a number that is out of **256** into a number that is out of **1000**.

Mathematically, the relationship is:

$$\text{Decimal Fraction} = \text{Raw Bits} \times \frac{1000}{256}$$



$$\text{Decimal Fraction} = \frac{\text{Raw Bits} \times 125}{32}$$

3. Why we do it this way in Assembly

On your **CH32V003**, division is slow and expensive. But **dividing by 32** is incredibly fast—it's just a **Right Shift by 5** (`srl a0, a0, 5`).

So the logic becomes:

1. Take your 8 bits of fraction.
2. Multiply by **125** (using that Shift-and-Add trick we discussed).
3. Shift Right by **5**.
4. The result is a number between **0** and **999**.

4. An Example

Suppose your Q8.8 number has a fractional part of **128** (which is exactly 0.5 in decimal).

- **Raw Bits:** `128`
- **Step 1:** $128 \times 125 = 16,000$
- **Step 2:** $16,000 \div 32 = 500$
- **Result:** You print `.500`.

Suppose your fractional part is **1** (the smallest possible step, $1/256 \approx 0.0039$).

- **Raw Bits:** `1`
- **Step 1:** $1 \times 125 = 125$
- **Step 2:** $125 \div 32 = 3$ (integer division truncates)
- **Result:** You print `.003`.

5. Why not just multiply by 1000 and divide by 256?

You could! But:

- `x * 1000` is a larger number than `x * 125`.
- `srl x, 8` (dividing by 256) is fast, but `srl x, 5` (dividing by 32) is exactly the same speed.
- By simplifying the fraction to **125/32**, we keep the numbers smaller and the math safer from overflowing a 32-bit register.

Summary

We multiply by **125** and shift by **5** because it is the **mathematical equivalent** of scaling a value from a "**0-255 scale**" to a "**0-999 scale**" using the fastest operations available on your chip.



AI Mode

All

Images

Videos

Short videos

More ▾



Sign in

