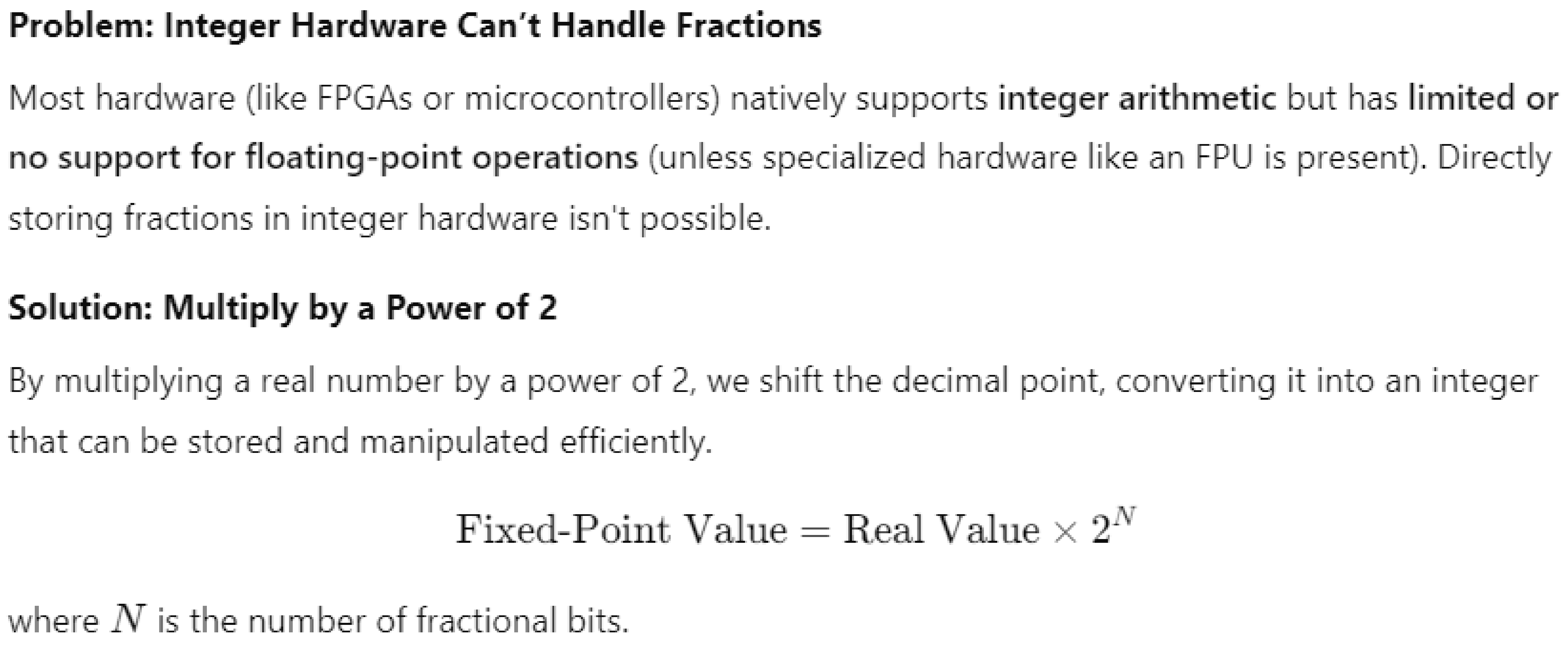
Fixed-Point Representation

Fixed-point representation allows us to use fractional numbers on low-cost integer hardware.



Problem: Integer Hardware Can't Handle Fractions

Most hardware (like FPGAs or microcontrollers) natively supports **integer arithmetic** but has **limited or no support for floating-point operations** (unless specialized hardware like an FPU is present). Directly storing fractions in integer hardware isn't possible.

Solution: Multiply by a Power of 2

By multiplying a real number by a power of 2, we shift the decimal point, converting it into an integer that can be stored and manipulated efficiently.

$$\text{Fixed-Point Value} = \text{Real Value} \times 2^N$$

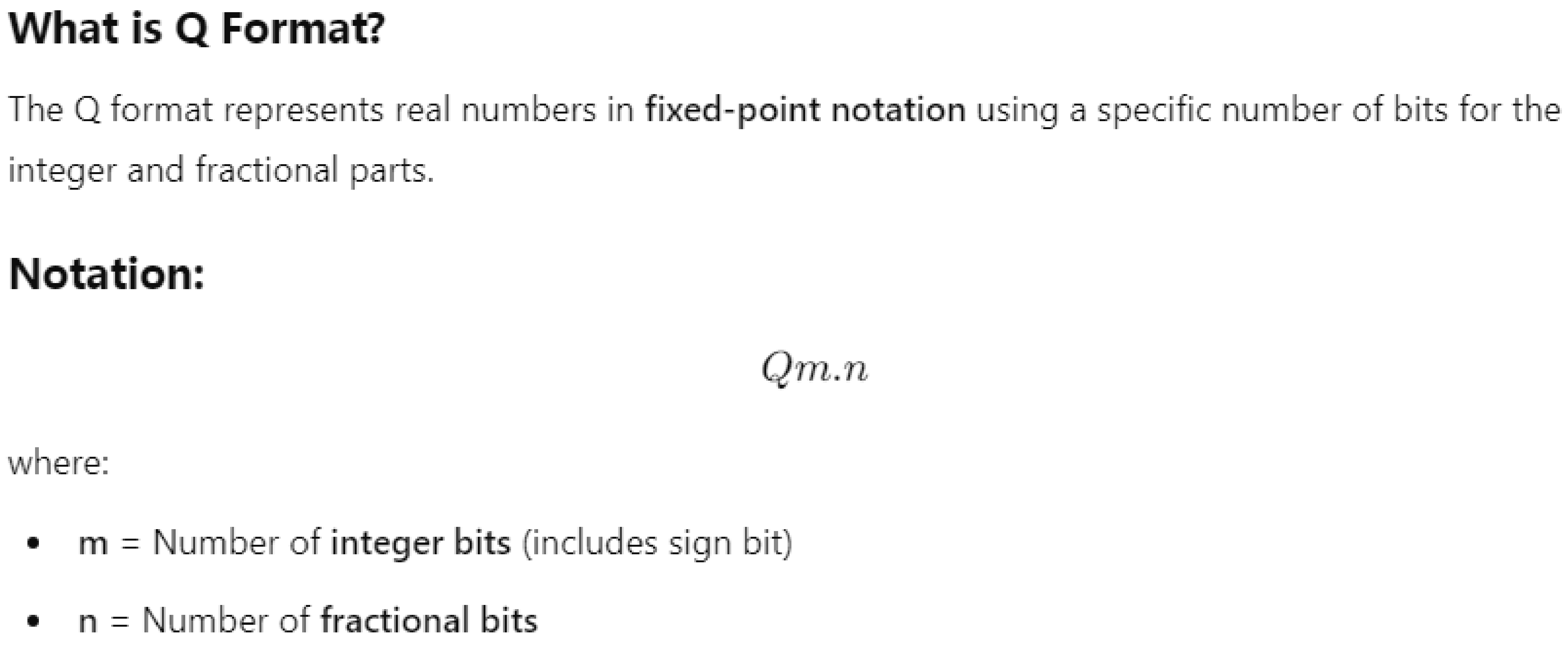
where N is the number of fractional bits.

Why Not Use Floating-Point?

•Floating-point requires specialized hardware (FPUs)

•It consumes more power and FPGA resources

•Fixed-point (Q format) is much faster and efficient



What is Q Format?

The Q format represents real numbers in **fixed-point notation** using a specific number of bits for the integer and fractional parts.

Notation:

$$Qm.n$$

where:

- **m** = Number of **integer bits** (includes sign bit)
- **n** = Number of **fractional bits**

Texas Instruments version

The Q notation, as defined by Texas Instruments,[1] consists of the letter `Q` followed by a pair of numbers *m*`.`*n*, where *m* is the number of bits used for the integer part of the value, and *n* is the number of fraction bits.

By default, the notation describes *signed* binary fixed point format, with the unscaled integer being stored in two's complement format, used in most binary processors. The first bit always gives the sign of the value(1 = negative, 0 = non-negative), and it is *not* counted in the *m* parameter. Thus, the total number *w* of bits used is $1 + m + n$.

Example 1:

Assume that an algorithm tested using floating-point arithmetic involves operations on $a = 9.216957436091198_{10}$*. Now that we are satisfied with the performance of the algorithm in floating-point representation, we have decided to implement it on a low-cost fixed-point processor which has a wordlength of 16 bits. What would be the appropriate Q format to represent* a *on this processor?*

min(m+1) =5

m=4, n=11

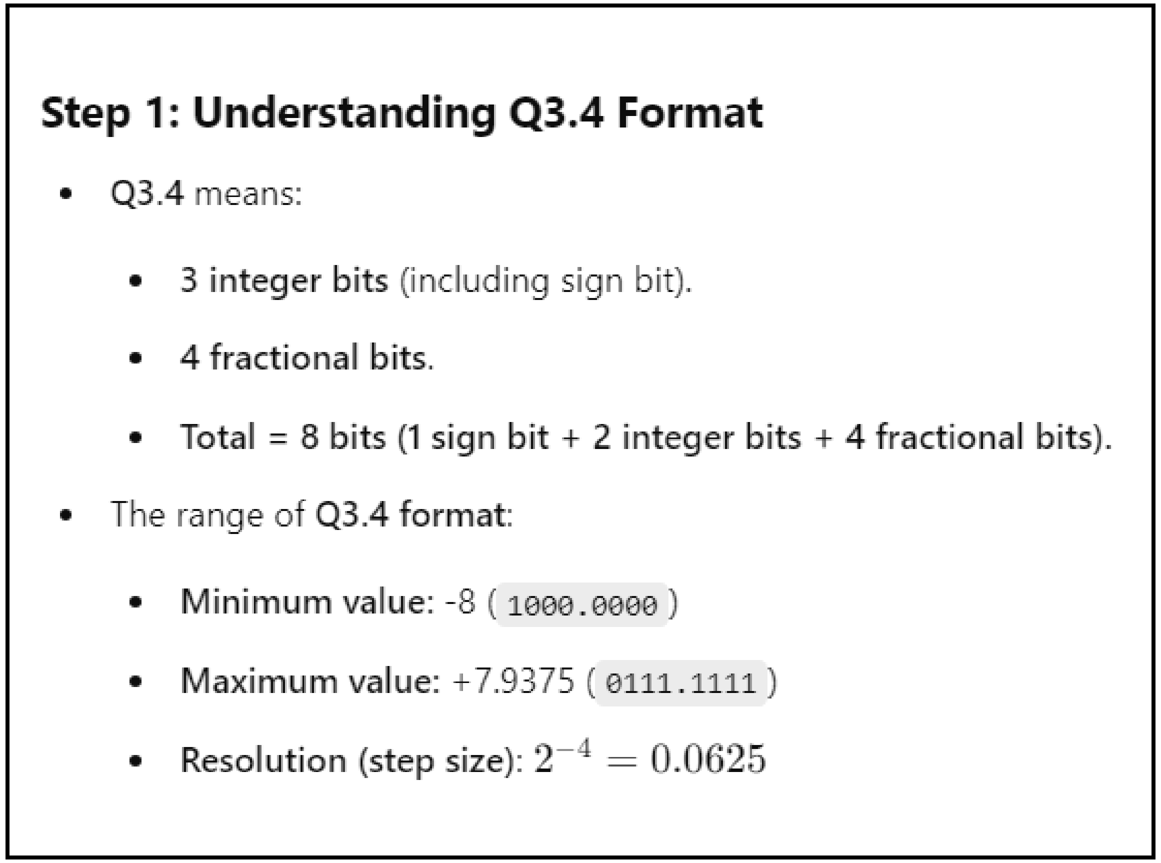
Q4.11 format

the Q4.11 representation of a without the implied binary point is equal to a multiplied by 2^11. Hence, to represent a in the Q4.11 format, we multiply it by 2^11, round it to the nearest integer, and convert the rounded result into the binary form.

$$a \times 2^{11} = 18876.3288 \approx 18876 = 100\ 1001\ 1011\ 1100_{(2)}$$

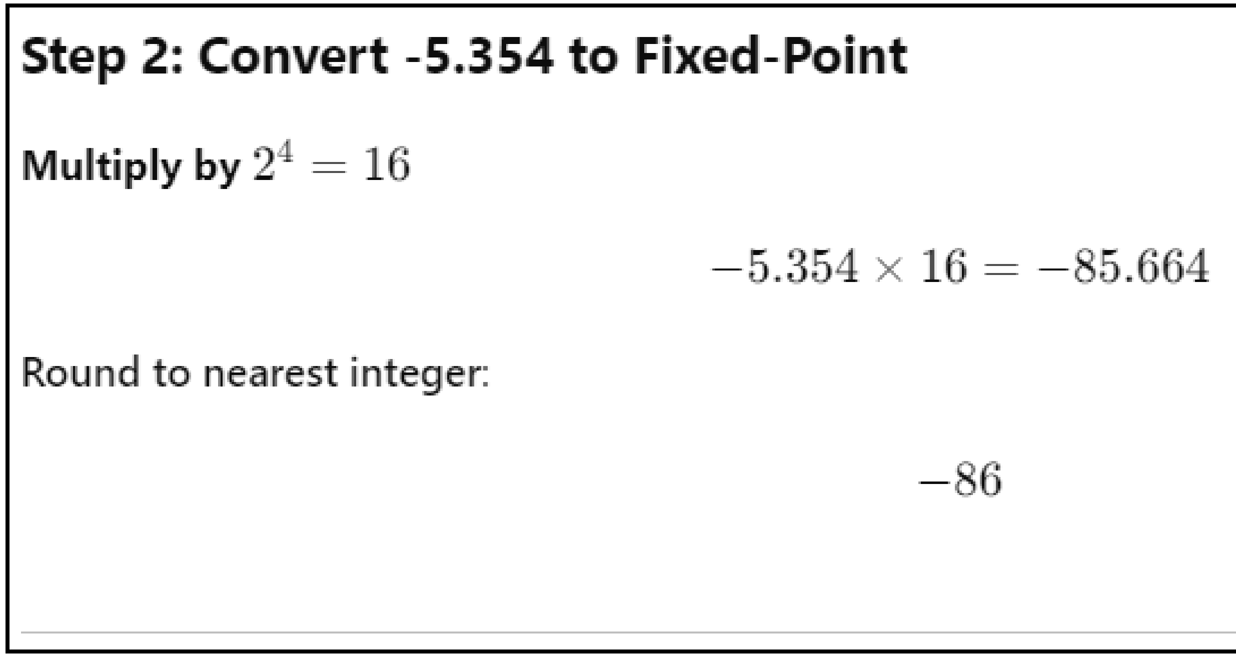
Since a is positive, we only need to consider a sign bit of zero. Therefore, the Q4.11 format of the number will be 01001.00110111100. For a negative number, we would have to first find the Q format of the absolute value and, then, convert it to the two's complement representation to take negative sign into account.

To represent **-5.354** in **Q3.4 format**



Step 1: Understanding Q3.4 Format

- Q3.4 means:
  - **3 integer bits** (including sign bit).
  - **4 fractional bits.**
  - **Total = 8 bits (1 sign bit + 2 integer bits + 4 fractional bits).**
- The range of **Q3.4 format**:
  - **Minimum value:** -8 (`1000.0000`)
  - **Maximum value:** +7.9375 (`0111.1111`)
  - **Resolution (step size):** $2^{-4} = 0.0625$



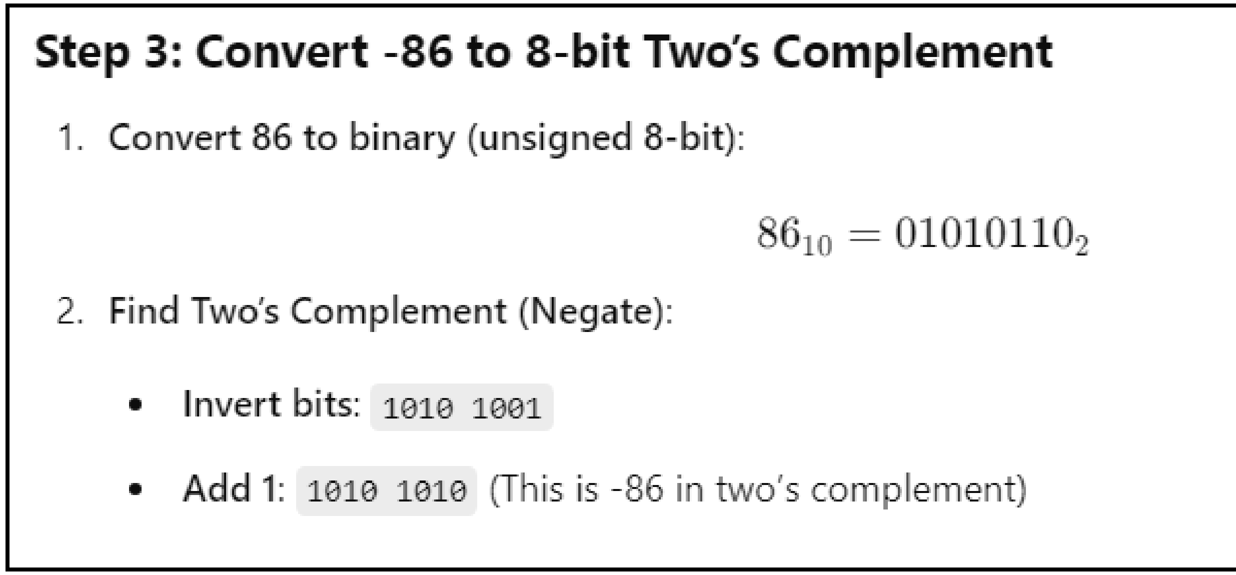
Step 2: Convert -5.354 to Fixed-Point

Multiply by $2^4 = 16$

$$-5.354 \times 16 = -85.664$$

Round to nearest integer:

$$-86$$

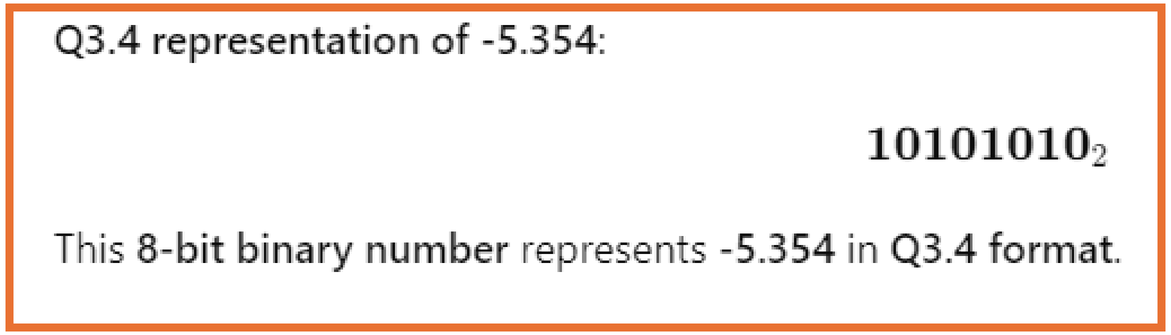


Step 3: Convert -86 to 8-bit Two's Complement

1. Convert 86 to binary (unsigned 8-bit):

$$86_{10} = 01010110_2$$

2. Find Two's Complement (Negate):
   - Invert bits: `1010 1001`
   - Add 1: `1010 1010` (This is -86 in two's complement)



Q3.4 representation of -5.354:

$$\mathbf{10101010_2}$$

This **8-bit binary number** represents **-5.354** in **Q3.4 format.**

Verification

- Convert `1010 1010` back to decimal:

  1. **Two's complement** of `1010 1010`:
     - Invert bits: `0101 0101`
     - Add 1: `0101 0110` = 86
  2. **Interpret as Q3.4**:

$$86/16 = 5.375$$

  3. **Small rounding error**: Original value was **-5.354**, stored as **-5.375** due to rounding.

Now, convert some real numbers from the range **-5 to 5** into **Q3.4 format**:

Real Value	Multiply by 16	Binary (8-bit)	Hex
-5.0	-80	**1011 0000**	B0
-3.5	-56	**1100 1000**	C8
-2.0	-32	**1110 0000**	E0
-1.25	-20	**1110 1100**	EC
0.0	0	**0000 0000**	00
1.5	24	**0001 1000**	18
3.75	60	**0011 1100**	3C
5.0	80	**0101 0000**	50

Choosing a Scaling Factor (Q-Format)

- The scaling factor is determined by how much **fractional precision** is needed.
- **Higher scaling factor** (e.g., 2^{14} for Q2.14) means **more precision** but **smaller range.**
- **Lower scaling factor** (e.g., 2^{4} for Q3.4) means **less precision** but **larger range.**

Example Comparisons:

Q-Format	Scaling Factor	Smallest Representable Step
Q1.15	$2^{15} = 32768$	$\frac{1}{32768} = 0.00003$
Q3.12	$2^{12} = 4096$	$\frac{1}{4096} = 0.00024$
Q5.10	$2^{10} = 1024$	$\frac{1}{1024} = 0.00098$

To represent -0.0265 in Q1.15 format, we follow these steps:

1. **Convert the positive value of 0.0265** into Q1.15 format (as done previously):
   - $0.0265 \times 32768 = 866.56$
   - Rounding gives 867, and in binary: $0000\,0011\,1000\,0011_2$.
2. **Represent the negative value** by taking the 2's complement of the positive value's binary form:
   - The binary of 867 is $0000\,0011\,1000\,0011_2$.
   - To find the 2's complement:
     - Invert the bits: $1111\,1100\,0111\,1100_2$
     - Add 1: $1111\,1100\,0111\,1101_2$

So, the 2's complement of 867 (for -0.0265) is $1111\,1100\,0111\,1101_2$.

3. **Convert the binary form to hexadecimal**:
   - $1111\,1100\,0111\,1101_2$ is equivalent to **0xFC7D** in hexadecimal.

Thus, **-0.0265** in Q1.15 format in hexadecimal is **0xFC7D**.