# Fixed Point Arithmetic for Embedded Systems

Systèmes Embarqués Microprogrammés

Prof. David Atienza Alonso, SEL-STI

# Content of Session

1. What is and why do we need fixed-point arithmetic?
2. Introduction to numeric representations
3. Arithmetic in fixed-point

# What is fixed-point (FxP)?

- Given a fixed number of digits in base 10, we can represent integer numbers by:

| $5$ $10^3$ | $4$ $10^2$ | $2$ $10^1$ | $9$ $10^0$ | 5429 |
|---|---|---|---|---|

**Positional notation**

- We can represent decimal numbers if we multiply implicitly by a negative power of 10:

| $5$ $10^1$ | $4$ $10^0$ | $2$ $10^{-1}$ | $9$ $10^{-2}$ | 54.29 |
|---|---|---|---|---|

- We can do the same in binary:

| $1$ $2^1$ | $0$ $2^0$ | $1$ $2^{-1}$ | $1$ $2^{-2}$ | 2.75 |
|---|---|---|---|---|

- The position of the decimal point is implicit (and fixed).
  - But we can modify it by explicitly shifting.

# What is floating-point (FP)?

- Floating point numbers are divided into two parts:
  - Significant digits (a.k.a. *mantissa*)
  - Exponent (a power of 2)
- A sequence of binary digits multiplied by [2 to the power of] the exponent:

| | | | |
|---|---|---|---|
| 0<br>Sign +/- | 1001<br>Mantissa = 9 | 100<br>Exponent = 4 | 144<br>($9 \times 2^4$) |
| 0<br>Sign +/- | 1001<br>Mantissa = 9 | 110<br>Exponent = 6 | 576<br>($9 \times 2^6$) |

**Floating point is a non-positional notation!**

- The position of the decimal point is determined by the exponent value

# Comparison between FxP and FP

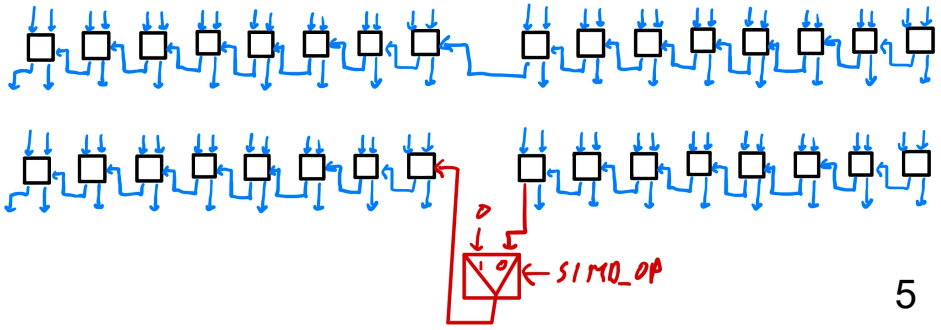
## Fixed Point

- Reuse integer arithmetic units
  - Both are positional
- Reuse integer registers
- SW can easily define its own representation
  - Because the position of the decimal point is implicit (i.e., hidden from the HW)
  - SW decides how to interpret the numbers
    - Always unsigned or 2's complement
- Easy to implement SIMD model for smaller bitwidths

## Floating Point

- Requires specific HW units
  - And the logic is more complex
- Requires new set of registers
  - (Just common practice)
- Difficult to define non-standard configurations that differ from what is supported by the HW

**How can we convert an integer adder into a SIMD adder?**



# Comparison between FxP and FP

- Floating point requires different HW units
  - →High cost in terms of area and power
    - Higher leakage due to larger area
    - Higher dynamic power due to more complex operation
- But it can also be executed in “SW-emulation” mode
  - Bit-manipulation operations to separate and operate on the mantissa and exponent of the operands independently
  - →High cost in terms of execution cycles and energy
    - Many more operations than for direct integer operations
    - Longer execution time → more energy ($E = P \cdot t$)
- Floating point has a <u>much</u> larger dynamic range

| 0 | 1111 | 111 |
|---|---|---|
| Sign +/- | Mantissa = 15 | Exponent = 7 |

**1920 ($15 \times 2^7$)**

**Max. value with 8 bits in integer positional notation is 255!**

| 0 | 11 | 11111 |
|---|---|---|
| Sign +/- | Mantissa = 3 | Exponent = 31 |

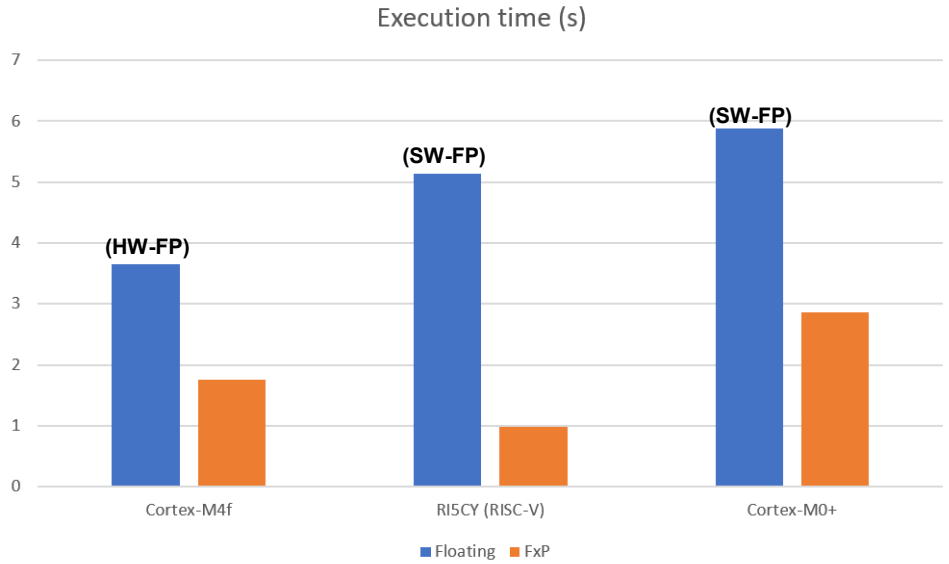
**6 442 450 944 ($3 \times 2^{31}$)**

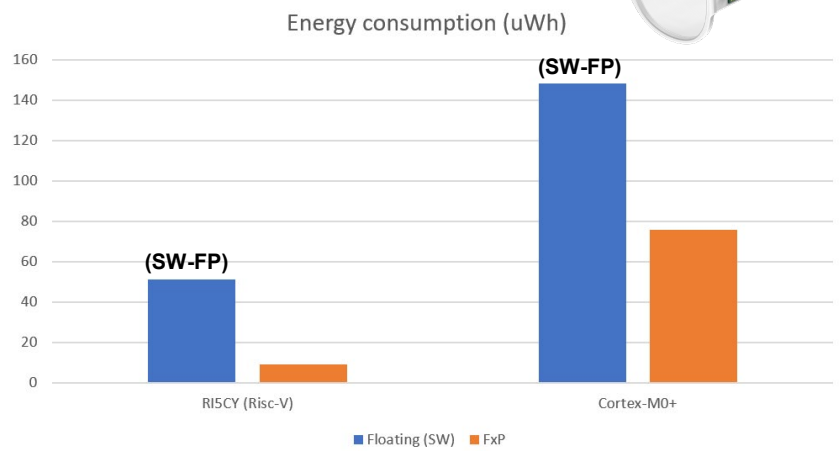
**For exp=$2^{31}$, it’s not possible to represent any numbers between 0, $2^{31}$, $2 \times 2^{31}$ and $3 \times 2^{31}$ !!**

**We can play with the dynamic range of our representation, but we can represent only 256 *distinct* values with 8 bits!**

# Efficiency of Fixed-point vs. floating-point

- Advantage: (Much) faster execution time in the NDS.
- Problem: The reduced dynamic range can introduce errors.

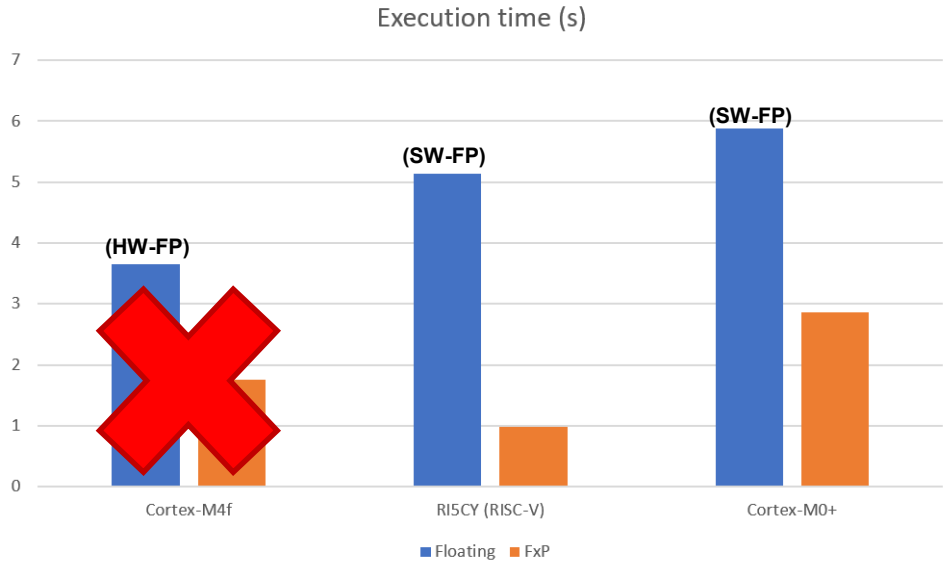


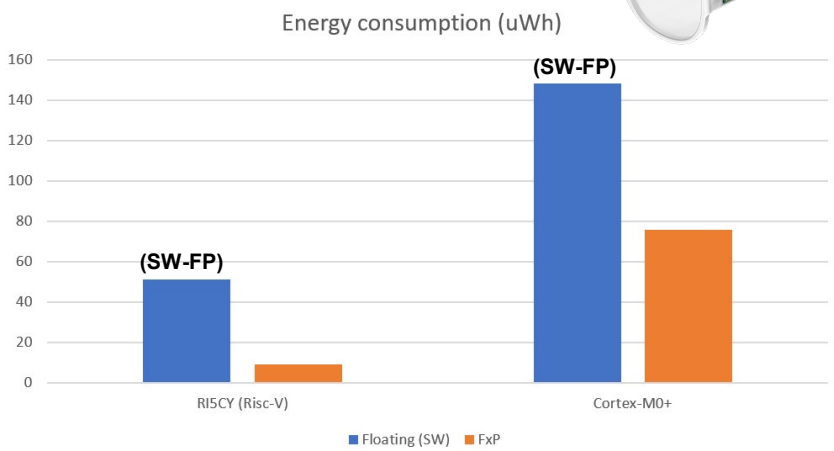




Example: Cognitive workload monitoring with the e-Glass wearable device. The Cortex-M4f has HW support for floating-point (FPU), while the RI5CY and Cortex-M0+ execute floating-point operations using SW emulation.

# Efficiency of Fixed-point vs. floating-point

- Advantage: (Much) faster execution time in the NDS.
- Problem: The reduced dynamic range can introduce errors.




**With careful optimization, HW-FP on the Cortex-M4f is faster than FxP**

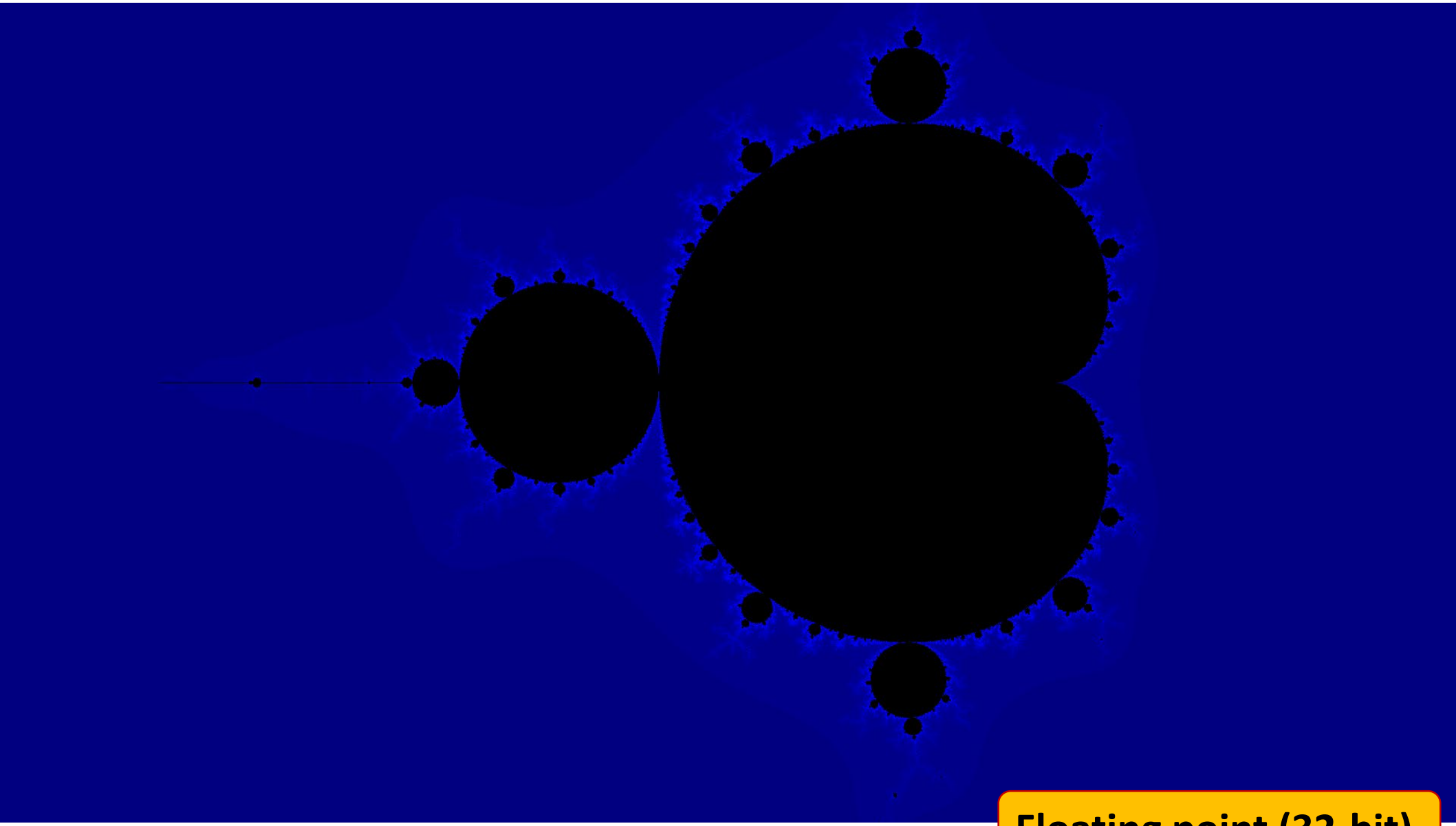
**If the CPU contains HW support for floating point, then it *may* be faster than FxP**

**The NDS CPU does not have HW support for floating point → FxP is faster**



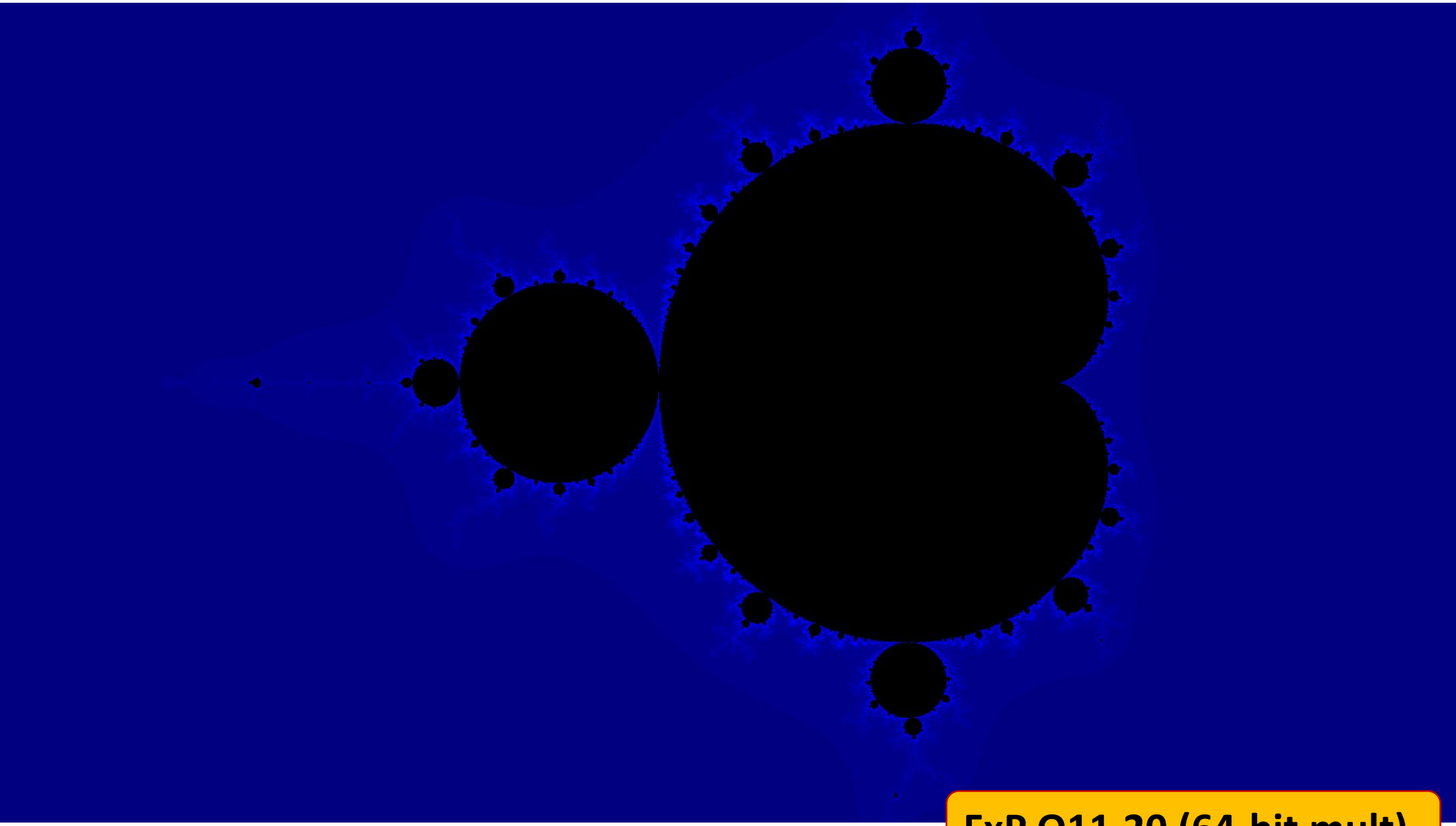
Data produced by Stefano Albini and Dimitrios Samakovlis, ESL/EPFL.

# Fixed-point arithmetic to compute fractals



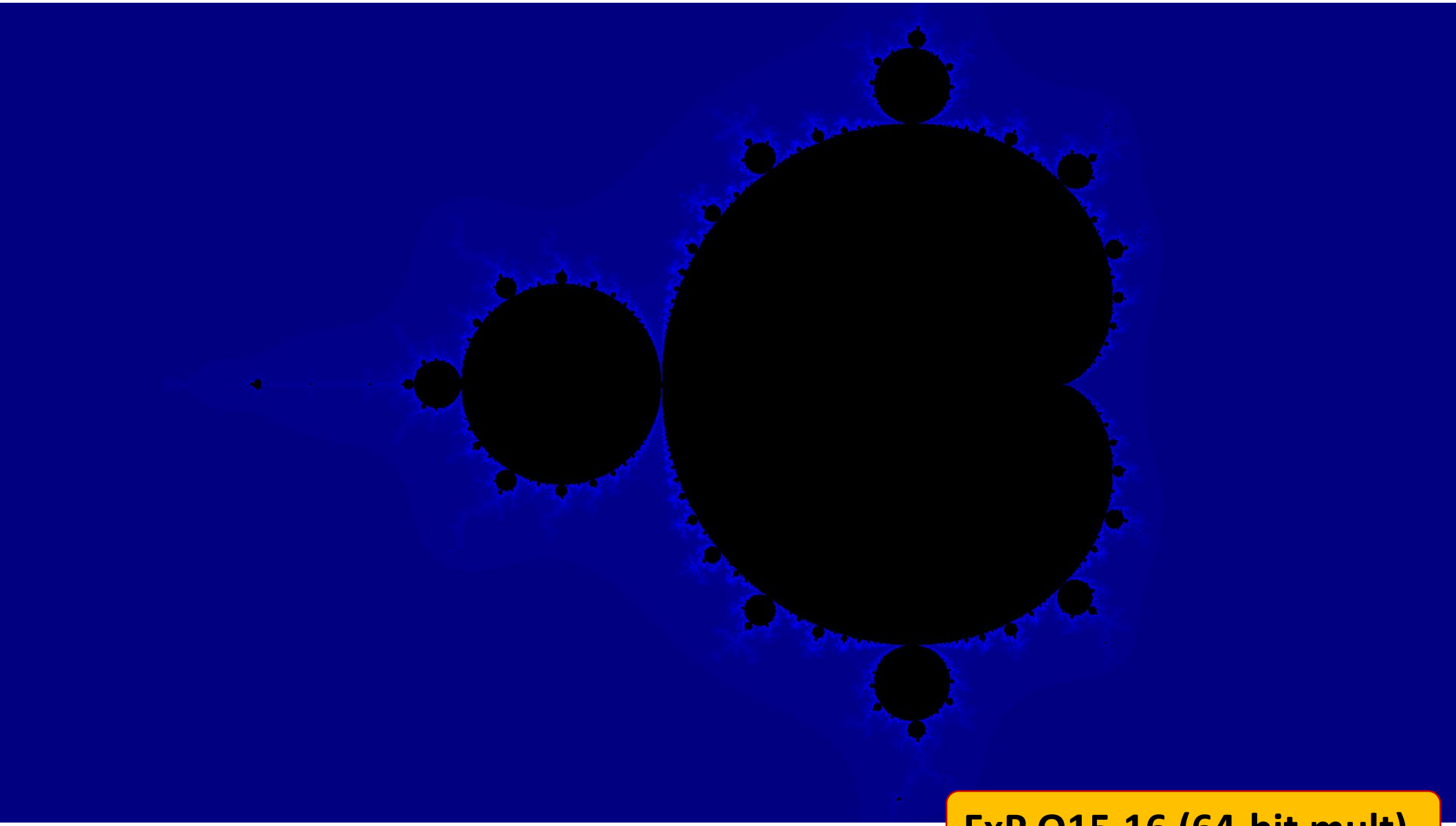
**Floating point (32-bit)**

# Fixed-point arithmetic to compute fractals



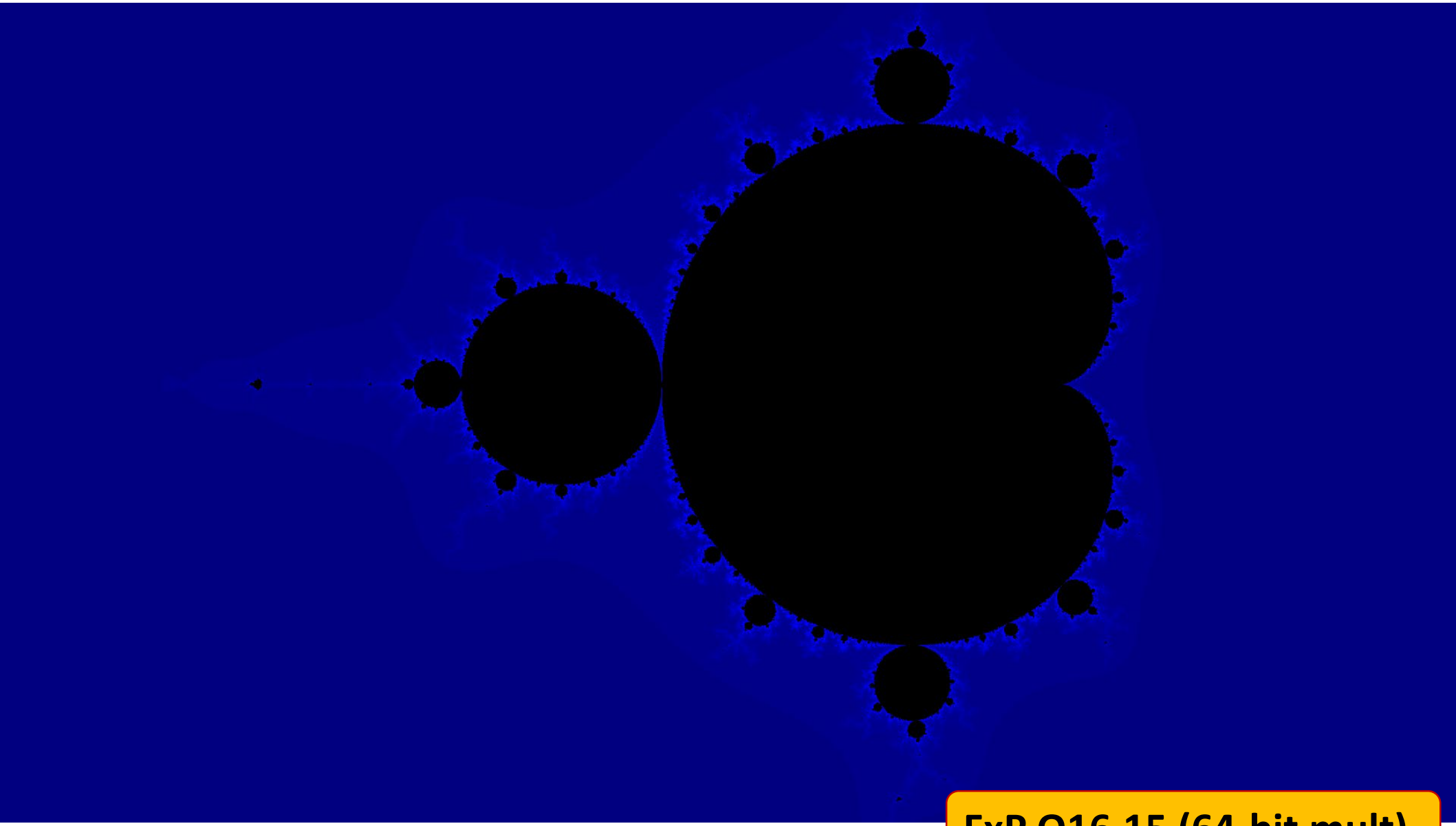
**FxP Q11.20 (64-bit mult)**

# Fixed-point arithmetic to compute fractals



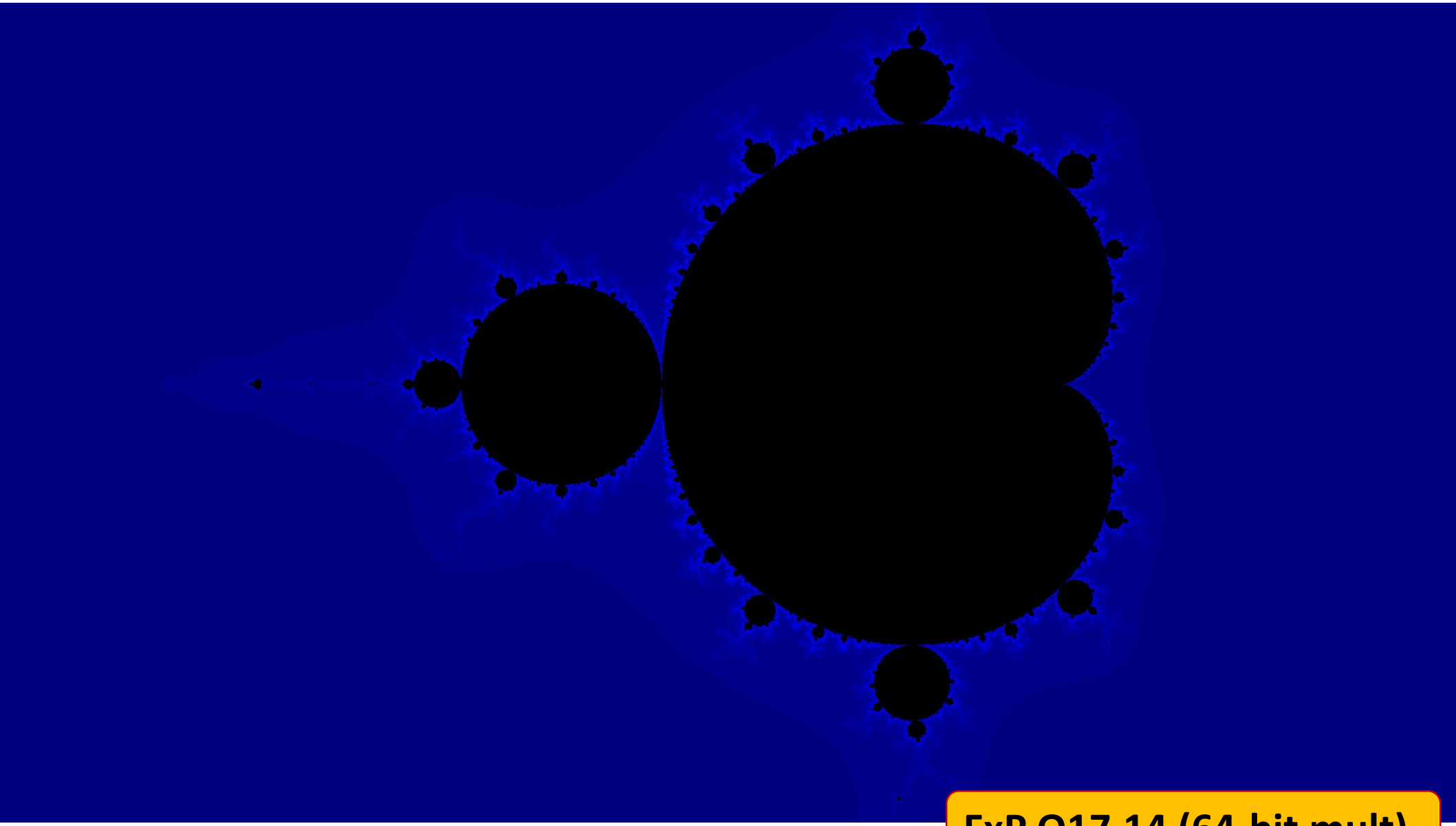
**FxP Q15.16 (64-bit mult)**

# Fixed-point arithmetic to compute fractals



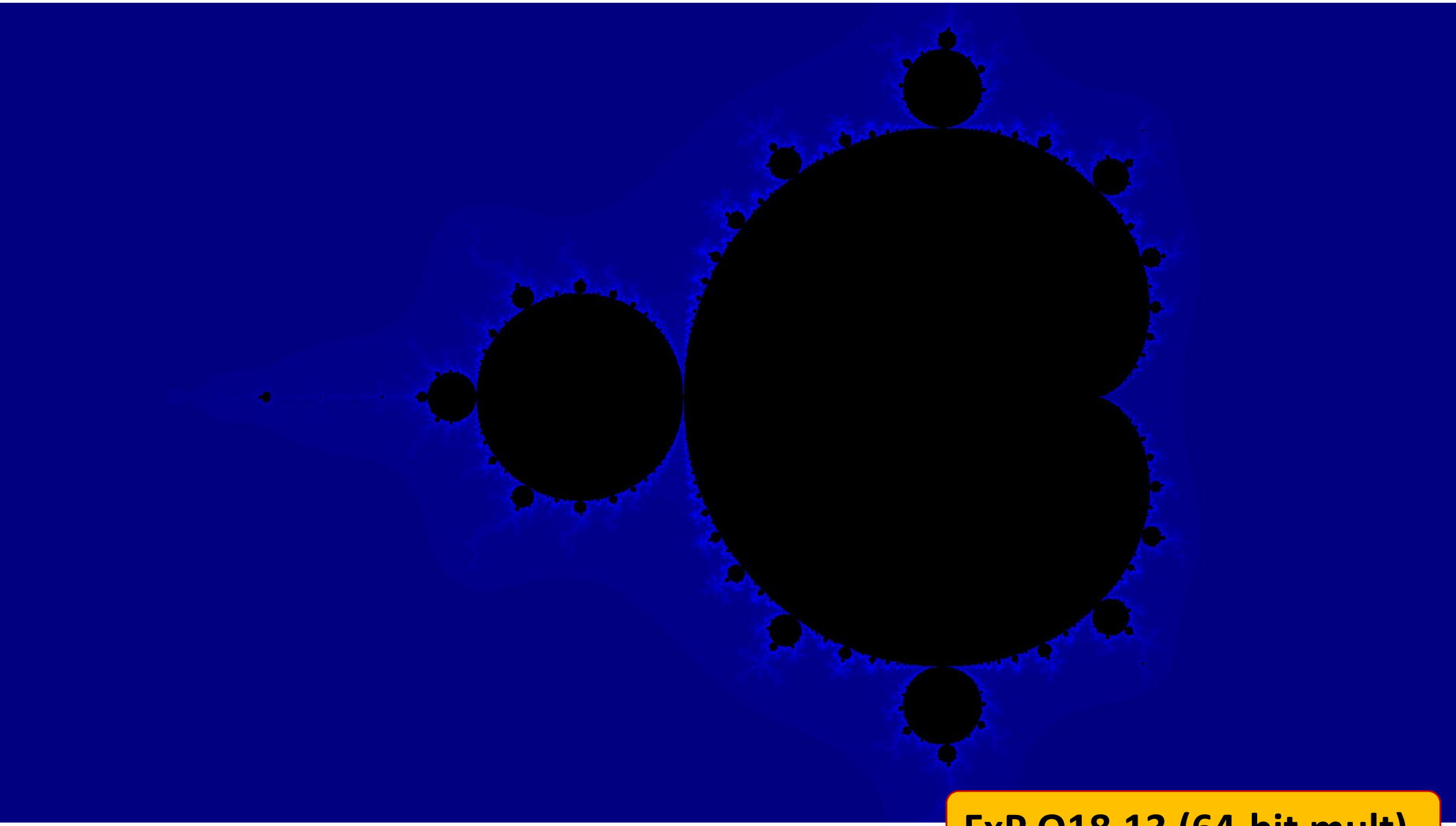
**FxP Q16.15 (64-bit mult)**

# Fixed-point arithmetic to compute fractals



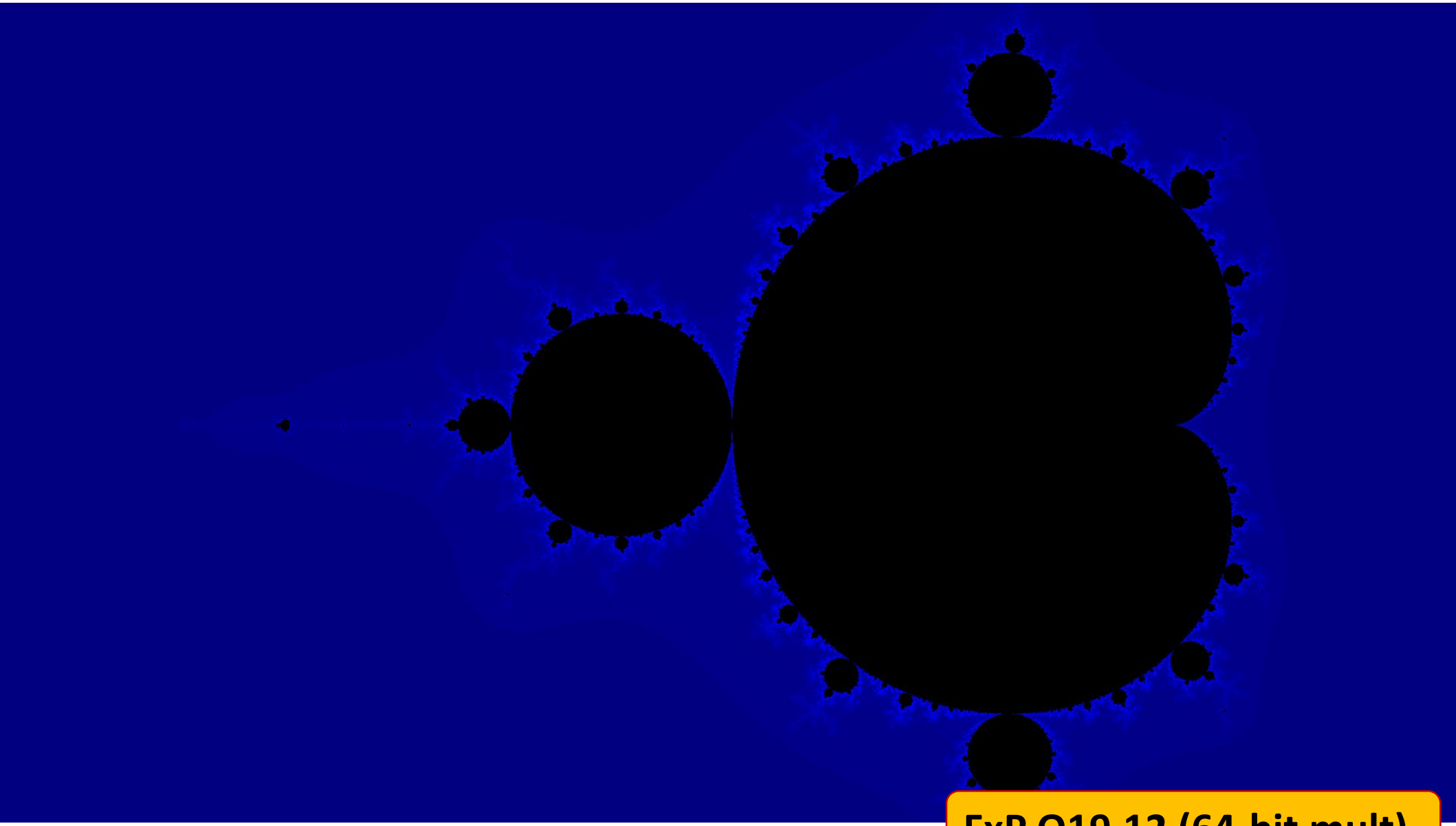
**FxP Q17.14 (64-bit mult)**

# Fixed-point arithmetic to compute fractals



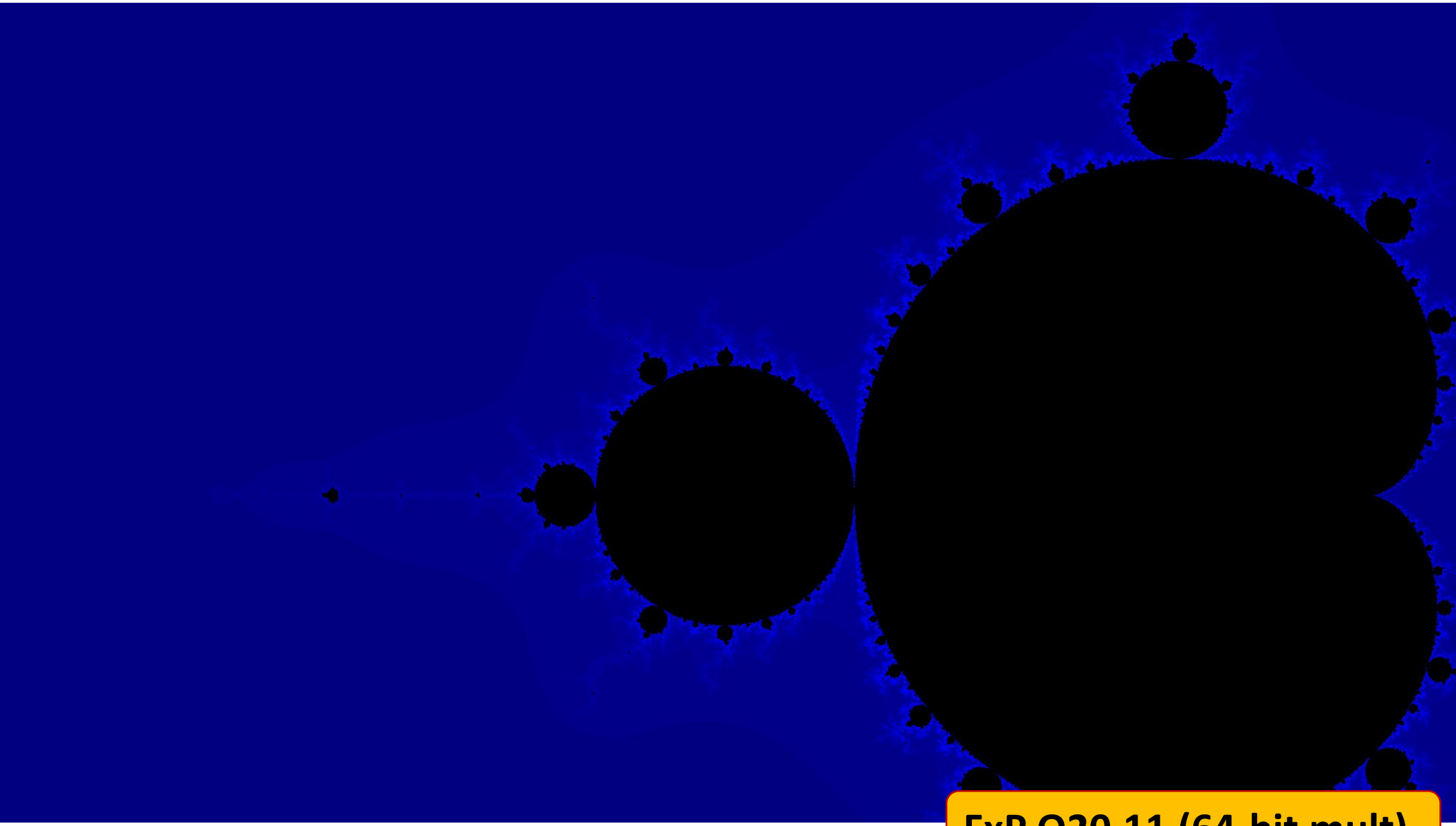
**FxP Q18.13 (64-bit mult)**

# Fixed-point arithmetic to compute fractals

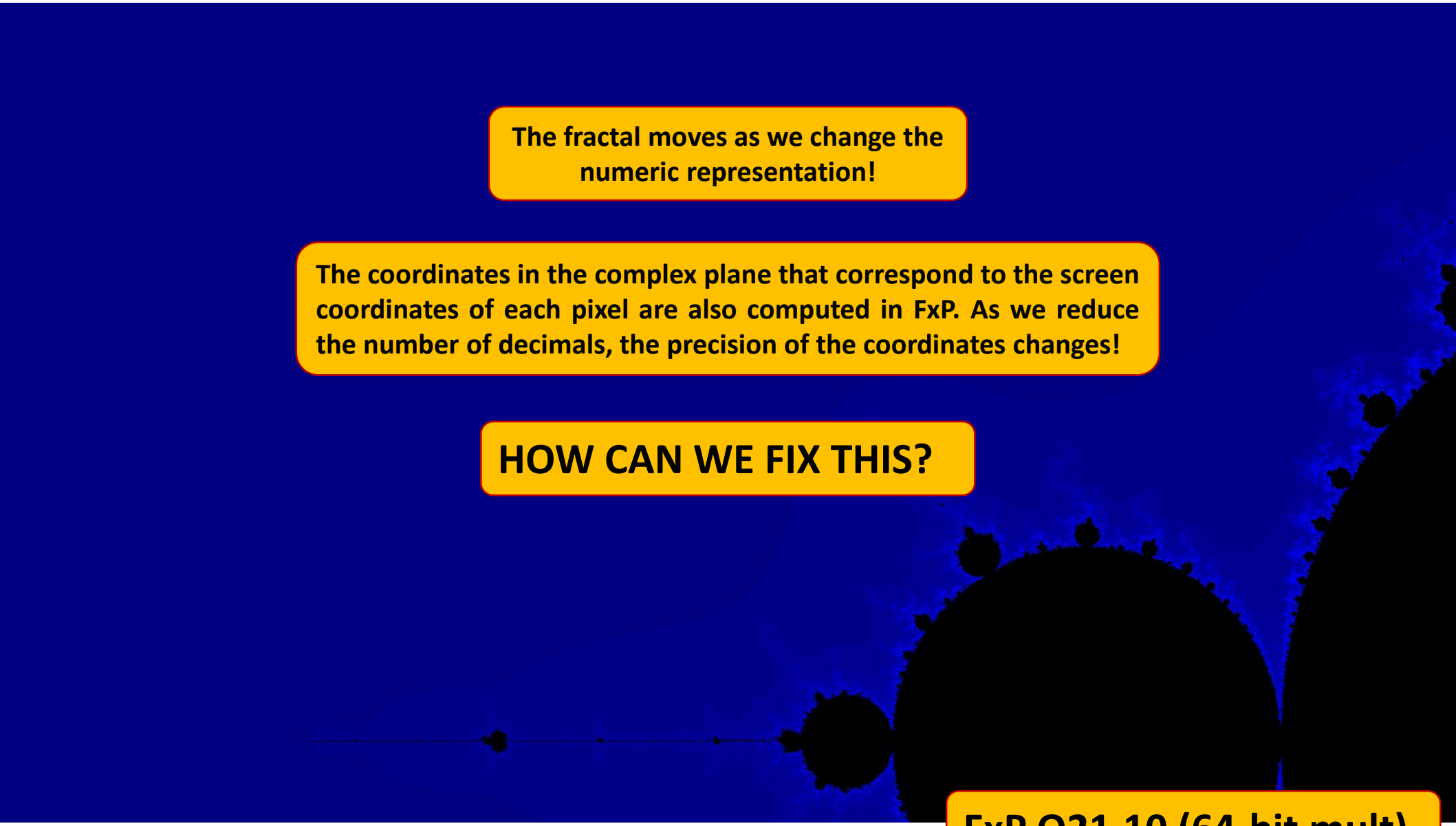


**FxP Q19.12 (64-bit mult)**

# Fixed-point arithmetic to compute fractals



# Fixed-point arithmetic to compute fractals



**The fractal moves as we change the numeric representation!**

**The coordinates in the complex plane that correspond to the screen coordinates of each pixel are also computed in FxP. As we reduce the number of decimals, the precision of the coordinates changes!**

**HOW CAN WE FIX THIS?**

**FxP Q21.10 (64-bit mult)**

# Fractal execution times (NDS)

| | NO THUMB | | THUMB | |
|---|---|---|---|---|
| Arithmetic | Time (s) | Improvement (over FP) | Time (s) | Improvement (over FP) |
| FP (32 bits) | 103 | 1 x | 105 | 1.0 x |
| FxP 16 bits: Q7.8 | 8 | 13 x | 30 | 3.5 x |
| FxP 16 bits: Q5.10 | 6 | 17 x | | |
| FxP 32 bits, mult-32: Q23.8 | 6 | 17 x | 29 | 3.6 x |
| FxP 32 bits, mult-32: Q21.10 | 5 | 21 x | 30 | 3.5 x |
| FxP 32 bits, mult-32: Q11.20 | 11 | 9 x | 83 | 1.3 x |
| FxP 32 bits, mult-64: Q21.10 | 10 | 10 x | 84 | 1.2 x |

# Why is FxP popular for CNNs?

- Reducing the model's memory footprint:
  - From 32-bit floating point
    - To FP-16: 2x reduction.
    - To FxP 16-bit: 2x reduction.
    - To FxP 8-bit: 4x reduction.
    - To FxP 4-bit: 8x reduction.
    - To binary (1-bit): 32x reduction.
  - Less memory in embedded devices.
  - Less memory bandwidth during inference, higher speed, less energy.

- If single instruction multiple data (SIMD) support exists, multiplied performance for the same energy!
  - FxP 8-bit with 32-bit SIMD registers: 4x faster than FP-32, (almost) the same energy.

# Why FxP instead of FP to reduce bit-width?

- Reduced bit-width is often not supported for FP.
  - Modern GPUs and processors increasingly support FP-16, FP-8 or even smaller sizes, particularly for DNNs.[1]
  - But not commonplace (yet).
  - And still requires dedicated HW.
- Embedded HW without floating point support.
  - Reuses integer functional units.
  - More energy and area efficient.
- Potential issues:
  - FxP has smaller dynamic range.
  - May produce under/overflows.
  - Or saturates too big or too small results.
  - Requires previous characterization of dynamic range.

**As in the previous "wandering" fractal example…**



[1] "FP8 Formats for Deep Learning," Paulius Micikevicius et al. arXiv, 2022

# Impact on DNN accuracy of FxP quantization

- Accuracy may be reduced.
  - Fixed point is not necessarily less accurate than floating point, but:
    - Smaller dynamic range.
    - Less precision as the bit-width is reduced.
- Experiments on a convolutional neural network (CNN) based industrial system:[1]
  - Weights can be stored with only decimals (±0.x).
  - Activations accumulate, hence they require integer bits.
  - Accuracy results:
    - Floating point 32-bit (baseline): 99.8 %
    - Fixed point 8-bit weights, 16-bit activations: 99.8 %
    - Fixed point 4-bit weights, 8-bit activations: 92.7 %
  - With no impact on accuracy, we can reduce the model size by a factor of 4, activation buffers by 2 and execute 2 times faster (with SIMD).

[1] "Impact of memory voltage scaling on accuracy and resilience of deep learning based edge devices." Benoît W. Denkinger, Flavio Ponzina, Soumya S. Basu, Andrea Bonetti, Szabolcs Balasi, Martino Ruggiero, Miguel Peón-Quirós, Davide Rossi, Andreas Burg, David Atienza. IEEE Design & Test, 2019. doi: 10.1109/MDAT.2019.2947282

# Content of Session

1. What is and why do we need fixed-point arithmetic?
2. Introduction to numeric representations
3. Arithmetic in fixed-point

# Numeric representations

- Integers
  - Positional notation.

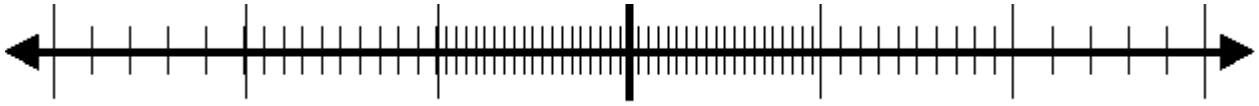
| 0 | 1 | 0 | 1 |
|---|---|---|---|
| $2^3$ | $2^2$ | $2^1$ | $2^0$ |

- Floating point (FP, 32-bit)
  - The fractional point can be moved *dynamically*.
  - Sign, exponent and fraction (significand) of fixed sizes.
  - Smallest positive normal number: $1.1754943508 \times 10^{-38}$
  - Range for subnormals: $\pm[1.175494210 \times 10^{-38}, 1.4012984643 \times 10^{-45}]$
- Fixed point (FxP): Positional notation
  - Fractional point divides integer and fractional parts with *fixed* bit-widths: It's a convention!
  - Range depends on number of bits for integer and fractional parts.

| 0 | 1 | . | 0 | 1 |
|---|---|---|---|---|
| $2^1$ | $2^0$ | | $2^{-1}$ | $2^{-2}$ |

# Characterization of numeric representations

- Word length
  - Number of bits in the representation: uint32_t, float, $Q_{15.16}$
- Range
  - Difference between most positive and most negative numbers.
- Resolution
  - Smallest non-zero magnitude representable.
  - FP32: $\pm 1.4012984643 \times 10^{-45}$
  - FxP (1+15+16 bits): $\pm 0.000015258789062 5$ ($2^{-16}$)
- Accuracy



  - Maximum difference between a real value and its representation.
  - For floating point, accuracy changes with the absolute value!
- Dynamic range
  - Ratio between maximum value and minimum positive value.

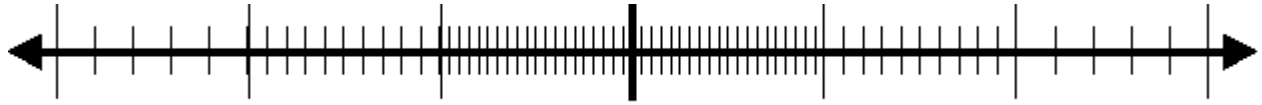
| 0 | 0 | 1 | . | 0 | 1 | ? |
|---|---|---|---|---|---|---|
| +/- | $2^1$ | $2^0$ | | $2^{-1}$ | $2^{-2}$ | |

# Characteristics of numeric representations: integers

- Unsigned, signed and magnitude, 2's complement.
- Range:
  - Unsigned: $[2^N-1, 0]$
  - 2's complement: $[2^{N-1}-1, -2^{N-1}]$
- Resolution: ±1
- Accuracy: 1.
- Dynamic range: $(2^{N-1}-1)/1$
  - For 32 bit, 2's complement: $2\,147\,483\,647 / 1 \sim 10^9$

# Characteristics of numeric representations: floating point

- IEEE 754 defines bit sizes: (16), **32**, 64, (80), 128, 256.
- For 32 bit (float):
  - 1 sign bit, 8 exponent bits, 23+1 significand bits (“normals”).
- Range: $[3.4028234664 \times 10^{38}, -3.4028234664 \times 10^{38}]$
- Resolution:
  - Smallest positive normal number: $1.1754943508 \times 10^{-38}$
  - Range for subnormals: $\pm[1.175494210 \times 10^{-38}, 1.4012984643 \times 10^{-45}]$
- Accuracy: Variable!



- Dynamic range:
  - $3.4028234664 \times 10^{38} / 1.1754943508 \times 10^{-38} \sim 10^{76}$

# Characteristics of numeric representations: fixed point

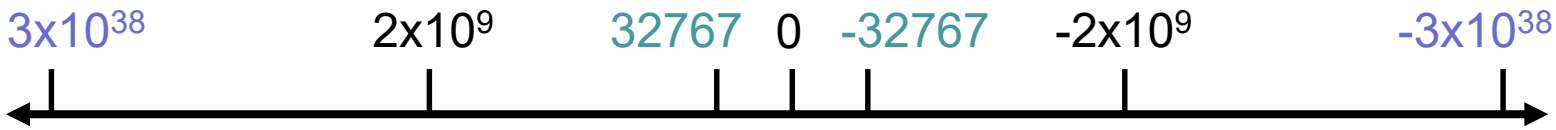
- Same size as native integers. Unsigned/signed (2's complement).
- Example: $Q_{15.16}$
  - 1 sign bit, 15 integer bits, 16 decimal bits.
- Range: [32767.999984741210938, -32768]
- Resolution:
  - Smallest positive normal number: 0.000015258789062 5 ($2^{-16}$)
- Accuracy: 0.000015258789062 5
- Dynamic range:
  - 32767.999984741210938 / 0.000015258789062 5 ~ $10^9$
  - → The same than the integer representation.

$Q_{14.17}$, $Q_{0.31}$ ???

**If we are using the same 32 bits, how is it possible to have a dynamic range of $10^{76}$ with FP32?**

# Representable range and density

- Representable ranges:




**How many *different* numbers can I represent in each case?**

- Density of representation:
  - uint32_t: One value every integer.
  - $Q_{15.16}$: One value every 0.0000152587890625
  - Floating point 32 bit:
    - ($2^{24}$, 0]: Every integer is representable.
    - ($2^{25}$, $2^{24}$]: **Only even integers are representable.**
    - ($2^{26}$, $2^{25}$]: **Only one out of four integers is representable.**

# Floating point representation

- Pros:
  - Large dynamic range.
  - Dynamic range adaptation.
  - Ideal when dealing with differing or unknown magnitudes.
  - Saturation to $\pm\infty$
- Cons:
  - Arithmetic is different than integers.
    - Requires specific HW.
  - Operations are more complex.
    - Larger area and higher energy consumption.
    - Dealing with infinites, NaNs, normal/subnormal numbers.
    - Addition requires aligning operands!
    - Re-normalization after every operation.
  - Not supported in many embedded platforms.
  - Complex SW emulation.

# Particularities of FP representations

- The varying density of the representation may produce unexpected results.

```
#include <stdio.h>

int main(int argc, char ** argv)
{
        float a = 0.0, b = 1.0, old = -1.0;

        while (old != a) {
                old = a;
                a += b;
        }

        printf("%0.9f\n", a);
        printf("0x%08X\n", *((unsigned int*) &a));

        return 0;
}
```

Will this program end?

Screen output

16777216.000000000

0x4B800000

**WHY?**

# Density of representation in FP

- 16 777 216 in FP32 is 0x4B800000:
  - 0 100 – 1011 – 1000 – 0000 – 0000 – 0000 – 0000 – 0000
  - Sign: 0, positive.
  - Exponent: 151-127=24
  - Significand: 1.0
    - Implicit “1.” in floating point representations.
  - Value: 1-0000-0000-0000-0000-0000-0000 = 16 777 216
    - (+1.00000000000000000000000*$2^{24}$ = 10000000000000000000000)
- Next possible binary value is 0x4B800001:
  - 0 100 – 1011 – 1000 – 0000 – 0000 – 0000 – 0000 – 0001
  - Sign: 0, positive.
  - Exponent: 151-127=24
  - Significand: 1.00000000000000000000001
  - Value: 1-0000-0000-0000-0000-0000-0010 = 16 777 218
    - (+1.00000000000000000000001*$2^{24}$ = 10000000000000000000010)

**Exponent in “excess-127” representation**

Add the implicit “1.” in front of the significant, then shift the fractional point “exponent” places.

**Not possible to represent the value 16 777 217 !**

# Special values in FP representations

- IEEE 754 defines several special values:
  - Infinites
    - Exp = 1…1, Significand = 0
  - NaNs (“Not-a-Number”)
    - Exp = 1…1, Significand ≠ 0
  - Positive and negative zeros
    - Exp = Minimum allowed exp – 1
    - Significand = 0
  - Subnormal numbers
    - To avoid jump from minimum normalized number ($1.1754943508 \times 10^{-38}$) to 0.
    - Exp = Minimum allowed exp – 1
    - Significand = Value represented with leading zeroes.
- Binary to text (round to even!):
  - 32 bits (float): Print with 9 decimal digits, round to even.
  - 64 bits (double): Print 17 decimal digits.

**This procedure keeps full precision between float/double and text.**

**Better: Dump the hex values rather than converting to base 10.**

# Fixed point representation

- Pros:
  - Positional notation, no special values.
  - (Almost) Same HW than integer operations.
  - Easy SW implementation.
  - Flexibility of representation.
    - Position of the fractional point by convention!
- Cons:
  - Requires that the dynamic range of the values is known.
  - Possible to change dynamic range
    - Change convention regarding position of fractional point.
    - But values can over/underflow.
  - May overflow (saturation may be introduced if required).
  - Multiplication and division require 2xN bits in intermediate operands.

# Is FP more precise than FxP?

- Not necessarily! With 32 bits:
  - IEEE 754 float uses 24 significand bits.
  - Fixed point $Q_{0.31}$ ($Q_{31}$) has 31 decimal bits.
  - For numbers in the range (1, -1):
    - $Q_{0.31}$ can represent accurately more numbers than float.
- The minimum representable values are:
  - Float: $1.4012984643 \times 10^{-45}$
  - $Q_{0.31}$ : 0.00000000046566128730773925781250 ($\sim 10^{-9}$)
- As seen before, the addition of two representable numbers with different magnitudes:
  - Can return one of the two terms in floating point.
  - Will always return a correct number in FxP
    - But it can over/underflow!!

# Other options: “Brain floating-point” format (bfloat16)

- Format proposed by Google specifically for DNNs.
- Based on IEEE-754 float (FP32).
  - Truncates the mantissa size.
  - Without changing the exponent size.
- Motivational insight:
  - For DNN applications, a reduced mantissa is enough as long as it is possible to still represent very small numbers without rounding to zero.
    - Avoid the “vanishing gradient” problem.
- Support: Google TPUs, Intel AVX-512 BF16, TensorFlow, ARM v8.6-A, …

# Bfloat16 bit format

- FP32: 1 + 8 + 23 (24)
  - 0 100 – 1011 – 1000 – 0000 – 0000 – 0000 – 0000 – 0000
  - Positive range: ~$3x10^{38}$ to ~$1x10^{-38}$
- FP16: 1 + 5 + 10 (11)
  - 0 100 – 1011 – 1000 – 0000
  - Positive range: 65 504 to ~$5.96x10^{-8}$
- Bfloat16 (bFP16): 1 + 8 + 7 (8)
  - 0 100 – 1011 – 1000 – 0000
  - Positive range: ~$3x10^{38}$ to ~$1x10^{-38}$

Sign / exponent / mantissa

# Bfloat16 pros and cons

- Pros:
  - Smaller mantissa requires lower area and power to implement arithmetic operations.
  - Similar dynamic range than FP32: ~$3x10^{38}$ to ~$1x10^{-38}$
    - Similar error behavior than FP32.
    - No need to adjust the loss function during training.
  - Fast conversion to/from FP32:
    - Keep exponent and truncate mantissa.
  - Support for infinites, NaNs and saturation, as in FP.
- Cons:
  - Worst representation for integer numbers (just 7 bits!).
  - Still requires dedicated HW, different than integer units.
  - More complex implementation than integer arithmetic (e.g., infinites, normalization, NaNs).

# Content of Session

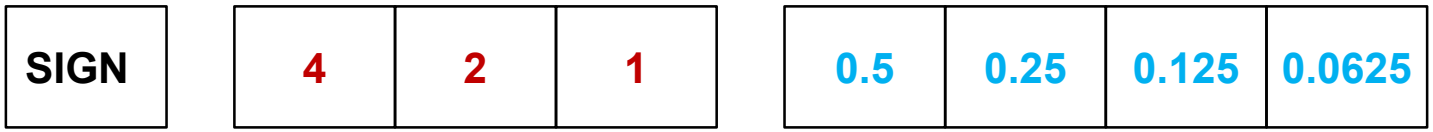
1. What is and why do we need fixed-point arithmetic?
2. Introduction to numeric representations
3. Arithmetic in fixed-point

# Nomenclature

- There are multiple ways to identify fixed-point numbers.
- In general, we need to identify the word length, the number of integer bits and the number of decimal bits.
- Common nomenclatures are:
  - $Q_{i.j}$: i integer bits and j decimal bits.
  - $Q_j$: j decimal bits (somewhat ambiguous).
- Also, indicate the presence of a sign bit.

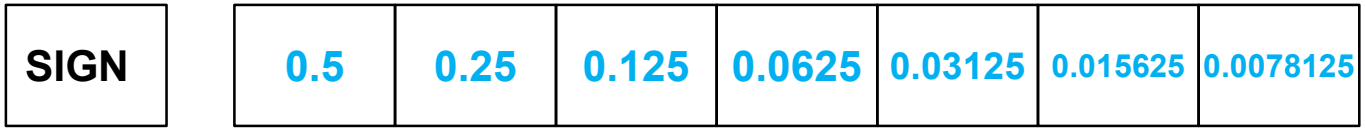
# Representation examples

- Q3.4




  - Range: [7.9375, -8]
  - Ex: 0, 0.0625, 0.125, 0.1875, 0.25, 0.3125, 0.375, 0.4375, 0.5, …

- Q0.7




  - Range: [0.9921875, -1]

- Q1.2 (4 bits)


  - Range: [1.75, -2]
  - Values: 1.75, 1.5, 1.25, 1, 0.75, 0.5, 0.25, 0, -0.25, -0.5, -0.75, -1, -1.25, -1.5, -1.75, -2.

# Choosing the decimal point position

- Requires the analysis of the dynamic range of numbers to represent.

- Trade-off between:
  - Dynamic range.
  - Number of distinct values that can be accurately represented.

# Implementation: Addition/Subtraction

- Addition in FxP → implemented with an addition in C.
  - Because the notation is positional.
  - No need to align the operands. No re-normalization.
  - The position of the decimal point is a convention.

**Reutilization of the integer ALUs!**

- In $Q_{2.3}$: (2's complement!)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0.875 | | 0 | 0 | 0 | 1 | 1 | 1 |
| 1.5 | + | 0 | 0 | 1 | 1 | 0 | 0 |
| 2.375 | | 0 | 1 | 0 | 0 | 1 | 1 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0.875 | | 0 | 0 | 0 | 1 | 1 | 1 |
| -0.25 | + | 1 | 1 | 1 | 1 | 1 | 0 |
| 0.625 | | 0 | 0 | 0 | 1 | 0 | 1 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0.875 | | 0 | 0 | 0 | 1 | 1 | 1 |
| 3.5 | + | 0 | 1 | 1 | 1 | 0 | 0 |
| -3.625 | | 1 | 0 | 0 | 0 | 1 | 1 |

**OVERFLOW!**

# Example: Addition

- Unsaturated addition and subtraction are exactly the same as their integer counterparts.
- No distinction between unsigned/signed.

```
////////////////////////////////////
//////////////////
// Fixed-point addition.
// NON-saturating (overflows)!
uint8_t FxpAdd(uint8_t a, uint8_t b)
{
  return a+b;
}
```

# Implementation: Addition/Subtraction with saturation

- FxP addition and subtraction can produce over/underflow.
  - In assembly, simply check the OV bit of the processor.
- Saturation can be achieved with a wider representation and introducing a check.
  - More expensive, but simulates the behavior of floating point arithmetic and avoids catastrophic errors.

```
int8_t addSatSigned(int8_t a, int8_t b)
{
    int16_t res;

    res = (int16_t)a + (int16_t)b;
    if (res > 0x7F)
        res = 0x7F;
    if (res < 0xFF80)
        res = 0xFF80;

    return (int8_t)res;
}
```

# Implementation: Multiplication

- Multiplication requires 2xN result bits.
- Use appropriate unsigned/signed integer variables!
- Multiply the two numbers normally.
- The result has double number of integer and fractional bits than the operands.
- To recover the size, shift right by the number of fractional bits.
  - This is equivalent to dividing by $2^N$, discarding the least significant bits.

- In $Q_{2.3}$:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.875 | | | | | 0 | 0 | 0 | 1 | 1 | 1 |
| 3.5 | | | | * | 0 | 1 | 1 | 1 | 0 | 0 |
| 3 | | | | | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | 0 | 0 | 0 | 0 | 0 | 0 | |
| | + | | 0 | 0 | 0 | 1 | 1 | 1 | | |
| | | 0 | 0 | 0 | 1 | 1 | 1 | | | |
| | 0 | 0 | 0 | 1 | 1 | 1 | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | |
| | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| | (Shift >> 3) | | | | | | | | | | |
| | | | | | 0 | 1 | 1 | 0 | 0 | 0 |

**3.0625 → 3**

# Example: Multiplication

- Multiplication can overflow. To address this, we can:
  - Sign-extend the operands to a larger size,
  - multiply,
  - shift in the result size and
  - then convert back to the operand size.
  - May be slower (specially on 32-bit processors).
- Use an arithmetic shift!
  - Automatic if signed/unsigned variables are used appropriately.

```
////////////////////////////////////////////////////////
// Fixed-point multiplications.

// 32-bit operands with 32-bit multiplication. Can overflow:
#define FixedMult32on32(i, j, shift) ((int32_t)(i)*(int32_t)(j) >> (shift))

// 32-bit operands with 64-bit multiplication. The shift is performed on the 64-bit temporary.
#define FixedMult32on64(i, j, shift) ((int32_t)((((int64_t)(i)*(int64_t)(j)) >> (shift))))

// 16-bit operands with 32-bit multiplication. The shift is performed on the 32-bit temporary.
#define FixedMult16on32(i, j, shift) ((int16_t)((((int32_t)(i)*(int32_t)(j)) >> (shift))))
```

# Example: Printing values

- We can convert FxP values to floating point, which are understood by the standard libraries.
  - SW emulation may be required!
- Simply separate the sign, integer and fractional parts.

```
////////////////////////////////////////////////////
//////
// Convert FxP (2's complement) to floating point.
double Fxp8Bit2Double(uint8_t value, uint8_t intBits)
{
  uint8_t decimalBits, mask, negative = 0;
  double res = 0.0, power;

  if (value & 0x80) {
    negative = 1;
    value = (~value) + 1;
  }

  // Extract integer part.
  decimalBits = 8 - intBits - 1;
  res += value >> decimalBits;

  // Extract fractional part.
  power = 0.5;
  mask = 1 << (decimalBits - 1);
  while (decimalBits > 0) {
    if (value & mask)
      res += power;
    power /= 2.0;
    value <<= 1;
    -- decimalBits;
  }

  return negative ? -res : res;
}
```

**Easier/faster implementation:**

```
#define FixedToFloat(i, shift) ((i) / (float)(1 << (shift)))
#define FloatToFixed(i, shift) ((i) * (float)(1 << (shift)))
```

**Why/when does this work?**

# Example Code

```
////////////////////////////////////////////////////////
/
int main(int argc, char **argv)
{
  int32_t value1, value2, res;

  ////////////////////////
  // Encoding examples.
  value1 = 0x0C; // 3.0
  printf("Binary: %hu - Decimal: %f\n", (uint16_t)value1,
    FixedToFloat(value1, 5));
  value1 = 0x0D; // 3.25
  printf("Binary: %hu - Decimal: %f\n", (uint16_t)value1,
    FixedToFloat(value1, 5));
  value1 = 0x65; // 12.625
  printf("Binary: %hu - Decimal: %f\n", (uint16_t)value1,
    FixedToFloat(value1, 4));
  value1 = 0xFFFFFF9B; // -12.625
  printf("Binary: %hu - Decimal: %f\n", (uint16_t)value1,
    FixedToFloat(value1, 4));
  value1 = 0xFFFFFF99; // -12.875
  printf("Binary: %hu - Decimal: %f\n", (uint16_t)value1,
    FixedToFloat(value1, 4));
  value1 = 0x67; // 12.875
  printf("Binary: %hu - Decimal: %f\n", (uint16_t)value1,
    FixedToFloat(value1, 4));
```

**Encodings**

```
  ////////////////////////
  // ADDITIONS.
  value1 = 0x65;
  value2 = 0xFFFFFFE7;
  res = value1 + value2;
  printf("\n----\nValue1: 0x%02X - Value2: 0x%02X\n",
    value1, value2);
  // Q28.3: 12.625 + (-3.125);
  printf("Q28.3: %f + %f = %f\n",
    FixedToFloat(value1, 3), FixedToFloat(value2, 3),
    FixedToFloat(res, 3));

  // Q29.2: 12.625 + (-3.125);
  printf("Q29.2: %f + %f = %f\n",
    FixedToFloat(value1, 2), FixedToFloat(value2, 2),
    FixedToFloat(res, 2));

  // Q30.1: 12.625 + (-3.125);
  printf("Q30.1: %f + %f = %f\n",
    FixedToFloat(value1, 1), FixedToFloat(value2, 1),
    FixedToFloat(res, 1));

  // Q24.7: 12.625 + (-3.125);
  printf("Q24.7: %f + %f = %f\n",
    FixedToFloat(value1, 7), FixedToFloat(value2, 7),
    FixedToFloat(res, 7));
```

**Additions**

# Example Code (Cont.)

```
//////////////////////
  // MULTIPLICATIONS.
  value1 = 0x02;
  value2 = 0x04;
  printf("\n----\nValue1: 0x%02X - Value2: 0x%02X\n",
value1, value2);

  // Q28.3: 0.25 * 0.5 = 0.125;
  res = FixedMult32on32(value1, value2, 3);
  printf("Q28.3: %f * %f = %f (0x%02X)\n",
    FixedToFloat(value1, 3), FixedToFloat(value2, 3),
    FixedToFloat(res, 3), (uint16_t)res);
  // Q30.1: 1.0 * 2.0 = 2.0;
  res = FixedMult32on32(value1, value2, 1);
  printf("Q30.1: %f * %f = %f (0x%02X)\n",
    FixedToFloat(value1, 1), FixedToFloat(value2, 1),
    FixedToFloat(res, 1), (uint16_t)res);
  // Q24.7: 0.015625 * 0.03125 = 0.00048828125 --> 0
  res = FixedMult32on32(value1, value2, 7);
  printf("Q24.7: %f * %f = %f (0x%02X)\n",
    FixedToFloat(value1, 7), FixedToFloat(value2, 7),
    FixedToFloat(res, 7), (uint16_t)res);

  value1 = 0x65;
  value2 = 0x02;
  printf("\nValue1: 0x%02X - Value2: 0x%02X\n", value1,
value2);
  // Q28.3: 12.625 * 0.25 = 3.15625 --> 3.125;
  res = FixedMult32on32(value1, value2, 3);
  printf("Q28.3: %f * %f = %f (0x%02X)\n",
    FixedToFloat(value1, 3), FixedToFloat(value2, 3),
    FixedToFloat(res, 3), (uint16_t)res);
  // Q24.7: 0.7890625 * 0.015625  = 0.0123291015625 -->
  // 0.0078125
  res = FixedMult32on32(value1, value2, 7);
  printf("Q24.7: %f * %f = %f (0x%02X)\n",
    FixedToFloat(value1, 7), FixedToFloat(value2, 7),
    FixedToFloat(res, 7), (uint16_t)res);

  value1 = 0x65;
  value2 = 0xFFFFFFFE;
  printf("\nValue1: 0x%02X - Value2: 0x%02X\n", value1,
value2);
  // Q28.3: 12.625 * -0.25 = -3.15625 --> -3.25;
  res = FixedMult32on32(value1, value2, 4);
  printf("Q28.3: %f * %f = %f (0x%02X)\n",
    FixedToFloat(value1, 3), FixedToFloat(value2, 3),
    FixedToFloat(res, 3), (uint16_t)res);
  // Q24.7: 0.7890625 * -0.015625  = -0.0123291015625 -->
  // -0.015625
  res = FixedMult32on32(value1, value2, 0);
  printf("Q24.7: %f * %f = %f (0x%02X)\n",
    FixedToFloat(value1, 7), FixedToFloat(value2, 7),
    FixedToFloat(res, 7), (uint16_t)res);

  return 0;
}
```

**Multiplications**

**The same binary value is interpreted differently according to the chosen representation.**

**The binary operations are the same!**

# Example: Results

```
Binary: 12 - Decimal: 3.000000
Binary: 13 - Decimal: 3.250000
Binary: 101 - Decimal: 12.625000
Binary: 155 - Decimal: -12.625000
Binary: 153 - Decimal: -12.875000
Binary: 103 - Decimal: 12.875000

----
Value1: 0x65 - Value2: 0xFFFFFFE7
Q28.3: 12.625000 + -3.125000 = 9.500000
Q29.2: 25.250000 + -6.250000 = 19.000000
Q30.1: 50.500000 + -12.500000 = 38.000000
Q24.7: 0.789062 + -0.195312 = 0.593750

----
Value1: 0x02 - Value2: 0x04
Q28.3: 0.250000 * 0.500000 = 0.125000 (0x01)
Q30.1: 1.000000 * 2.000000 = 2.000000 (0x04)
Q24.7: 0.015625 * 0.031250 = 0.000000 (0x00)

Value1: 0x65 - Value2: 0x02
Q28.3: 12.625000 * 0.250000 = 3.125000 (0x19)
Q24.7: 0.789062 * 0.015625 = 0.007812 (0x01)

Value1: 0x65 - Value2: 0xFFFFFFFE
Q28.3: 12.625000 * -0.250000 = -3.250000 (0xFFE6)
Q24.7: 0.789062 * -0.015625 = -0.015625 (0xFFFE)
```

**Exact results**

**Underflow**

**(3.15625)**
**(0.01232909375)**
**Effect of shifting is truncation.**

# Questions?

# Let's use fixed point arithmetic in the NDS!