Unit 3
Floating and Fixed Point Numbers

EL-GY 9463: INTRODUCTION TO HARDWARE DESIGN

PROFS. SUNDEEP RANGAN, SIDDHARTH GARG

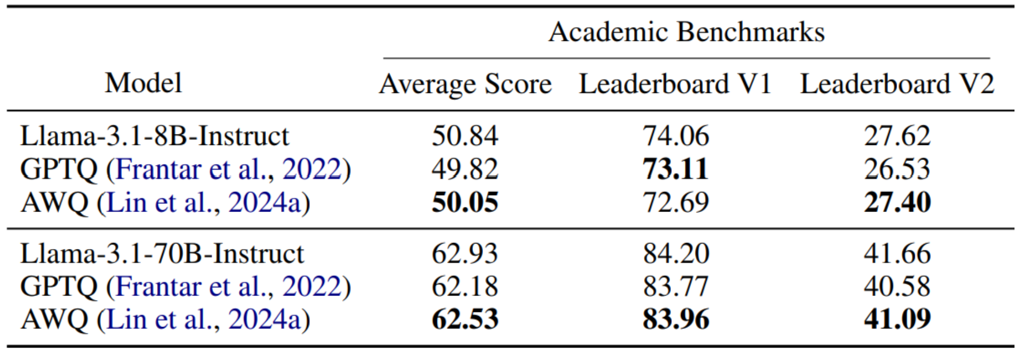
Today’s Lecture

The world is continuous and infinite, but computers are finite

- ❑ How do we represent numeric values in a finite numbers of bits?
- ❑ More generally, how can values be compressed?
- ❑ What representations are computationally simple?
- ❑ What representations result in good computational approximations?

Example 1: Embedded LLMs

- Full LLMs require massive amounts of memory
  - Cannot work on embedded devices
- Quantization
  - Aggressive reduction in number of bits / parameter
  - Ex: INT4 vs. FP32
- New techniques have enabled very low footprint
  - Often 1 to 1.5 GB
  - Fits on commercial smartphones
- Performance often remains excellent
  - With judicious choice of approximation techniques



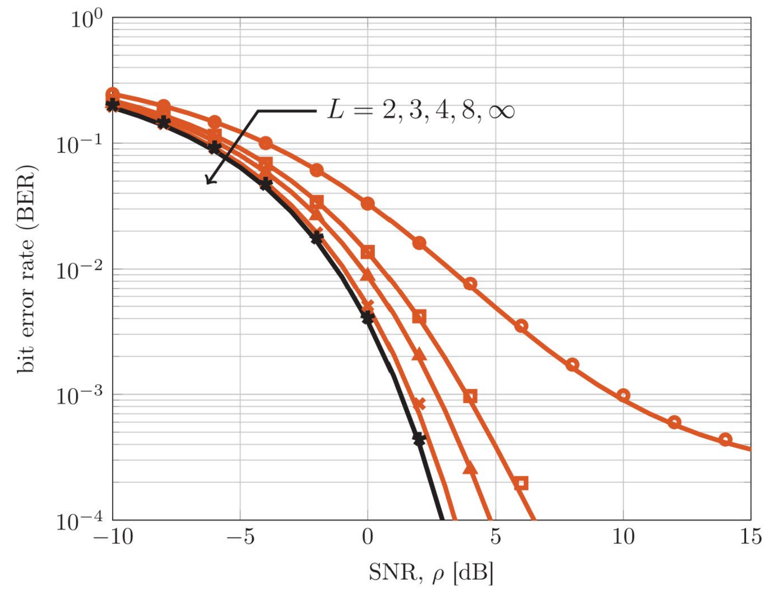
Model	Academic Benchmarks		
	Average Score	Leaderboard V1	Leaderboard V2
Llama-3.1-8B-Instruct	50.84	74.06	27.62
GPTQ (Frantar et al., 2022)	49.82	**73.11**	26.53
AWQ (Lin et al., 2024a)	**50.05**	72.69	**27.40**
Llama-3.1-70B-Instruct	62.93	84.20	41.66
GPTQ (Frantar et al., 2022)	62.18	83.77	40.58
AWQ (Lin et al., 2024a)	**62.53**	**83.96**	**41.09**

hugging-quants/**Meta-Llama-3.1-8B-Instruct-GPTQ-INT4**

Kurtic, Eldar, et al. "“Give Me BF16 or Give Me Death”? Accuracy-Performance Trade-Offs in LLM Quantization." *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2025.

Example 2: Next-Generation Wireless

- Massive MIMO systems in cellular base stations
  - 100s – 1000s of antennas
  - Each antenna gets data at ~1 Gsamp/s
- Data rate and processing is overwhelming
- Very low-rate quantization has been key
  - 2- 4 bits per antenna



Uplink performance vs number of quantization levels L

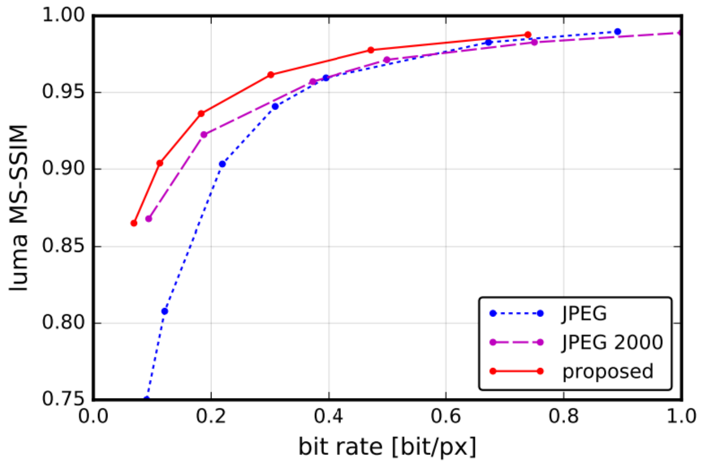
Jacobsson, S., Durisi, G., Coldrey, M., Gustavsson, U., & Studer, C. (2017). Throughput analysis of massive MIMO uplink with low-resolution ADCs. *IEEE Transactions on Wireless Communications*, *16*(6), 4038-4051.

Compression Research at NYU

- ML techniques at the forefront for learning new compression
- NYU has several leading researchers in the area

Ballé, J., Laparra, V., & Simoncelli, E. P. (2016). End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704*.

Jona Balle, NYU



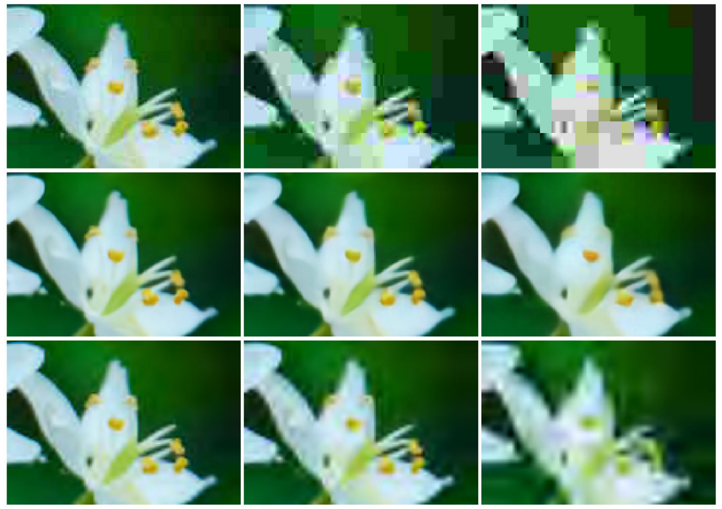


Figure 6: Cropped portion of an image compressed at three different bit rates. Middle row: the proposed method, at three different settings of λ. Top row: JPEG, with three different quality settings. Bottom row: JPEG 2000, with three different rate settings. Bit rates within each column are matched.

Learning Objectives

- ❑ Represent numbers in floating point and fixed point
- ❑ Implement arithmetic operations efficiently in these formats
- ❑ Identify cases for overflow
- ❑ Evaluate approximation errors via simulation
- ❑ Optimally select scaling factors and parameters for the formats
- ❑ Convert floating point algorithms to fixed point

Outline

- ❑ Floating Point Numbers
- ❑ Fixed Point Numbers
- ❑ Optimal Scaling of Fix Point Numbers
- ❑ Converting Floating Point Algorithms to Fixed Point

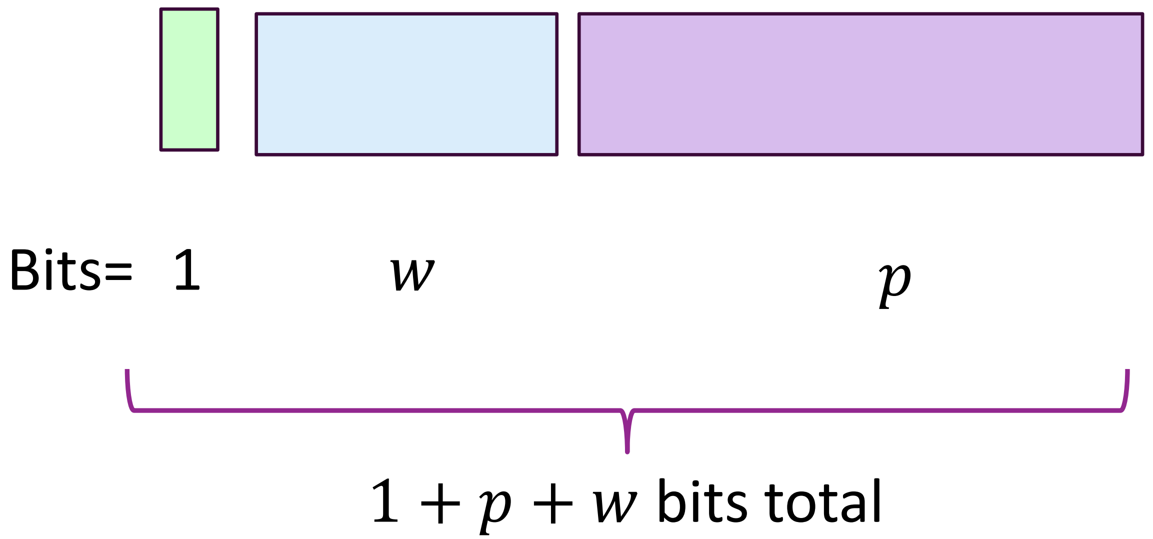
Limitations of Integers

- Unit 1 introduced two numeric types:
  - $I(b)$ = signed b bit integers
  - $U(b)$ = unsigned b bit integers
- Problem 1. Cannot handle fractional components
  - Ex: 7.9, -12.2
  - Rounding may cause excessive approximation error
- Problem 2. Limited range
  - Values overflow $\geq 2^b$ for $U(b)$ and $\geq 2^{b-1}$ for $I(b)$

Simple Floating Point

- Fixed point $FP(p,w)$
- Signed bit:
  - $s = 0,1$, One bit
- Exponent, w bits
  - $e \in U(w)$
- Mantissa, significand, p bits
  - $c \in U(p)$
  - Unsigned integer. Not two complement
  - p is the precision
- Fixed scale factor A
  - Constant, no bits
  - Used to set range of x
  - More later

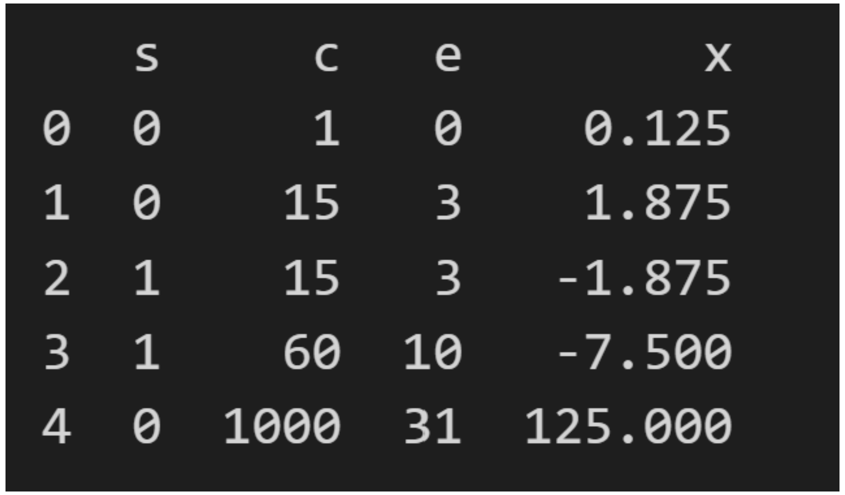
$$x = A(-1)^s\, 2^e\, c$$



Example

- Consider some values with:
  - Precision $p = 10$, Exponent width = 5
  - Scale $A = 2^{-8}$
  - These values are never used. Just for demo
- Total number of bits = $1 + p + w = 16$ bits
- Mantissa range is $c \in [0,1023]$, Exponent range $e \in [0,31]$
- Table to the right shows values with

$$x = A(-1)^s c 2^e$$



```
   s     c   e        x
0  0     1   0    0.125
1  0    15   3    1.875
2  1    15   3   -1.875
3  1    60  10   -7.500
4  0  1000  31  125.000
```

Example Problem

Question 5. Floating point representation

Consider a simple floating-point representation $x = A\,(-1)^s\,c\,2^e$, where the scale factor is $A = 2^{-3}$. The fields are:

- sign bit s
- 3-bit unsigned exponent e
- 5-bit mantissa c, interpreted as an *unsigned integer* (no implicit leading 1)

The bits are packed with the sign bit first, followed by the exponent bits, then the mantissa bits. What decimal values are represented by the following bit patterns?

a. `0 000 10000`
b. `1 011 11000`
c. `0 101 00101`

Solution

Solution:

a. Sign bit $s = 0 \to$ positive. Exponent bits $000 \to e = 0$. Mantissa bits $10000_2 = 16$. Value:

$$x = 2^{-3} \cdot (+1) \cdot 16 \cdot 2^0 = 2^{-3} \cdot 16 = 2$$

b. Sign bit $s = 1 \to$ negative. Exponent bits $011 \to e = 3$. Mantissa bits $11000_2 = 24$. Value:

$$x = 2^{-3} \cdot (-1) \cdot 24 \cdot 2^3 = (-1) \cdot 24 = -24$$

c. Sign bit $s = 0 \to$ positive. Exponent bits $101 \to e = 5$. Mantissa bits $00101_2 = 5$. Value:

$$x = 2^{-3} \cdot (+1) \cdot 5 \cdot 2^5 = 2^{-3} \cdot 160 = 20$$

Terminology

- To compare ways of representing numbers, we introduce some terminology
- Class or Format: A finite set of real-valued numbers
  - Example: $I(b), U(b), FP(p, w)$
- Bit representation: The bit encoding used to store each element in the class:
  - For integer: Representation is the two-complement or unsigned integer
  - For FP: representation is the triple (s, c, e)
- Decoded or numeric value: The real-value for a given bit representation
  - Example: $x = A(-1)^s c\, 2^e$

Dynamic Range

- A benefit of Floating point is its dynamic range
- Dynamic range: Given a number class:

$$D = \frac{\max\limits_{x>0} |x|}{\min\limits_{x\neq 0} |x|}$$

  - Represents span between the max before overflow and underflow
- Often quoted in dB: $D_{dB} = 20 \log_{10}(\ D\ )$
- Or bits: $D_{bits} = \log_2(\ D\ )$

Dynamic Range of Integers

- ❑ Consider signed b bit integers: $x \in I(b)$
- ❑ Max value: $\max_{x>0} |x| = 2^{b-1} - 1 \approx 2^{b-1}$
- ❑ Min non-zero value: $\min |x| = 1$
- ❑ Hence dynamic range of integers: $D = 2^{b-1}$
- ❑ For integers, dynamic range is:

$$\boxed{D_{bits} = \log_2(D) = b - 1}$$

  - equal to number of non-sign bits

Floating-Point Dynamic Range

- Recall: $|x| = Ac2^e$
- Largest value for $|x|$:
  - $c = 2^p - 1 \approx 2^p,\ e = 2^w - 1 \Rightarrow \max|x| \approx A2^p 2^{2^w - 1} = A2^{p-1}2^{2^w}$
- Smallest non-zero value for $|x|$:
  - $c = 1, e = 0 \Rightarrow |x|_{min} = A(1)2^0 = A$
- Dynamic range: $D = 2^{p-1}2^{2^w}$
- For floating point:

$$\boxed{D_{bits} = p - 1 + 2^w}$$

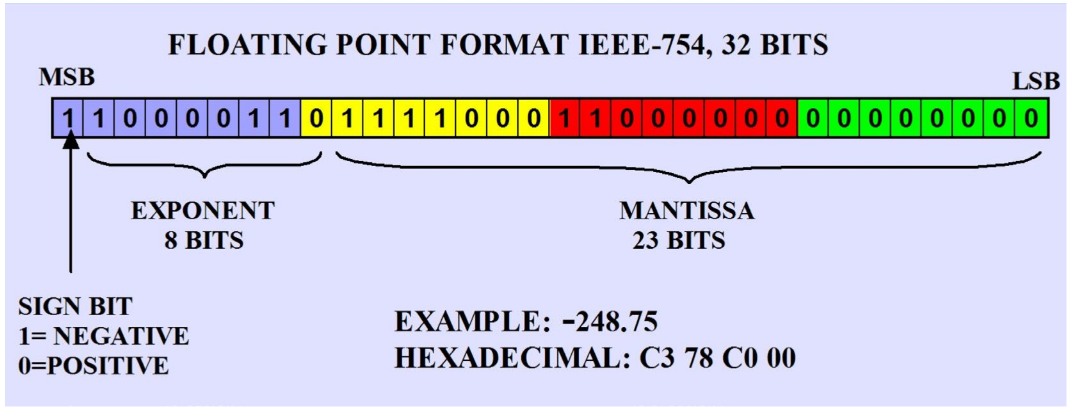
  - Grows exponentially with w

Example Comparison

- 32-bit integer:
  - Range goes from 1 to 2^{31}
  - Dynamic range in linear scale $\approx 2.14(10)^9$
  - $b = 32 \Rightarrow D_{bits} = b - 1 = 31$ bits
- 32-bit floating point:
  - $p = 23,\ w = 8$
  - Dynamic range in bits: $D_{bits} = p - 1 + 2^8 = 22 + 255 = 277$ bits
  - Dynamic range in linear scale $\approx 10^{83}$
- FP has > 70 orders of magnitude more range

IEEE-754 Floating Point Standard

Format	Mantissa bits (stored)	Significand precision (with implicit 1)	Exponent bits
FP16 (half)	10	11	5
FP32 (single)	23	24	8
FP64 (double)	52	53	11



IEEE 754 Normal and Sub-Normal

- For FP32: precision $p = 23$ and exponent width $w = 8$
- Uses a two regions
- Normal region:
  - Exponent in range $e \in [1,2^8 - 1)$
  - In this region: $x = (-1)^s m 2^{e-127}, \ m = 1 + \frac{c}{2^p}$
  - The 1 is as if the m has an implicit leading bit of 1
  - Note that $m \in [1,2)$
- Sub-normal region:
  - Exponent $e = 0$
  - In this region: $x = (-1)^s \frac{c}{2^p} e^{-126}$
  - Used for small numbers

IEEE 754 Properties

- Single precision (FP32):
  - Largest normal $\approx 3.4(10)^{38}$
  - Smallest normal $\approx 1.18(10)^{-38}$
  - Smallest sub-normal $\approx (10)^{-45}$
  - Vastly greater range than 32-bit integers
- Double precision (FP64):
  - Largest normal $\approx 1.80(10)^{308}$
  - Smallest normal $\approx 2.23(10)^{-308}$
  - Smallest sub-normal $\approx 5(10)^{-324}$

Floating Point Addition

- Consider two numbers in normal region (sub-normal performed similarly)
  - $x_i = (-1)^{s_i}\left(1+\frac{c_i}{2^p}\right)2^{e_i-127}$
- We need the representation for the sum: $x = x_1 + x_2 = (-1)^s\left(1+\frac{c}{2^p}\right)2^{e-127}$
- WLOG: Assume $e_1 \geq e_2$. For simplicity assume: $s = s_1 = s_2$ (other cases are similar)
- Then: $x = (-1)^s\, 2^p m\, 2^{e_1-127}$, $m = 2^p + c_1 + (2^p + c_2) \gg (e_1 - e_2)$
- Value $m \in 2^p[1,4)$
- Since we need $m \in 2^p[1,2)$ for normal numbers, we shift
- If $m \geq 2$ set $m \leftarrow \frac{m}{2}$, $e \leftarrow e_1 + 1$ otherwise $e \leftarrow e_1$
- Take $c \leftarrow m - 2^p$

Resulting Algorithm

- Given two numbers: $x_i = (-1)^{s_i}\left(1 + \frac{c_i}{2^p}\right) 2^{e_i - 127}$ with $s_1 = s_2 = s,\ e_1 \geq e_2$
- Compute $m = 2^p + c_1 + (2^p + c_2) \gg (e_1 - e_2)$
- If $m \geq 2^{p+1}$: set $m \leftarrow m \gg 1,\ e \leftarrow e_1 + 1$
- Else $e \leftarrow e_1$
- Set $c \leftarrow m - 2^p$
- Handle overflow and underflow
- Then: $x = x_1 + x_2 = (-1)^s \left(1 + \frac{c}{2^p}\right) 2^{e-127}$
- Conclusion: We can perform addition of two FP numbers
- But requires considerable extra logic for multiple cases and overflow / underflow

Floating Point Multiplication

- Consider two numbers in normal region (sub-normal performed similarly)
  - $x_i = (-1)^{s_i}\left(1 + \frac{c_i}{2^p}\right) 2^{e_i - 127}$
- We need the representation for the product: $x = x_1 x_2 = (-1)^s \left(1 + \frac{c}{2^p}\right) 2^{e-127}$
- WLOG: Assume $e_1 \geq e_2$.
- Then: $x = (-1)^s\, 2^p m\, 2^e$, where $s = s_1 \oplus s_2$ (XOR) and $e = e_1 + e_2 - 254$
- $m = \left(1 + \frac{c_1}{2^p}\right)\left(1 + \frac{c_2}{2^p}\right) = 1 + \frac{c}{2^p},\;\; c = c_1 + c_2 + ((c_1 c_2) \gg p)$
- We need $c \in [0, 2^p)$ If $c \geq 2^p$:
  - $c \leftarrow ((2^p + c) \gg 1) - 2^p$ and $e \leftarrow e + 1$
- Handle overflow and underflow
- Can perform multiplication, but requires considerable extra control logic

Comparison

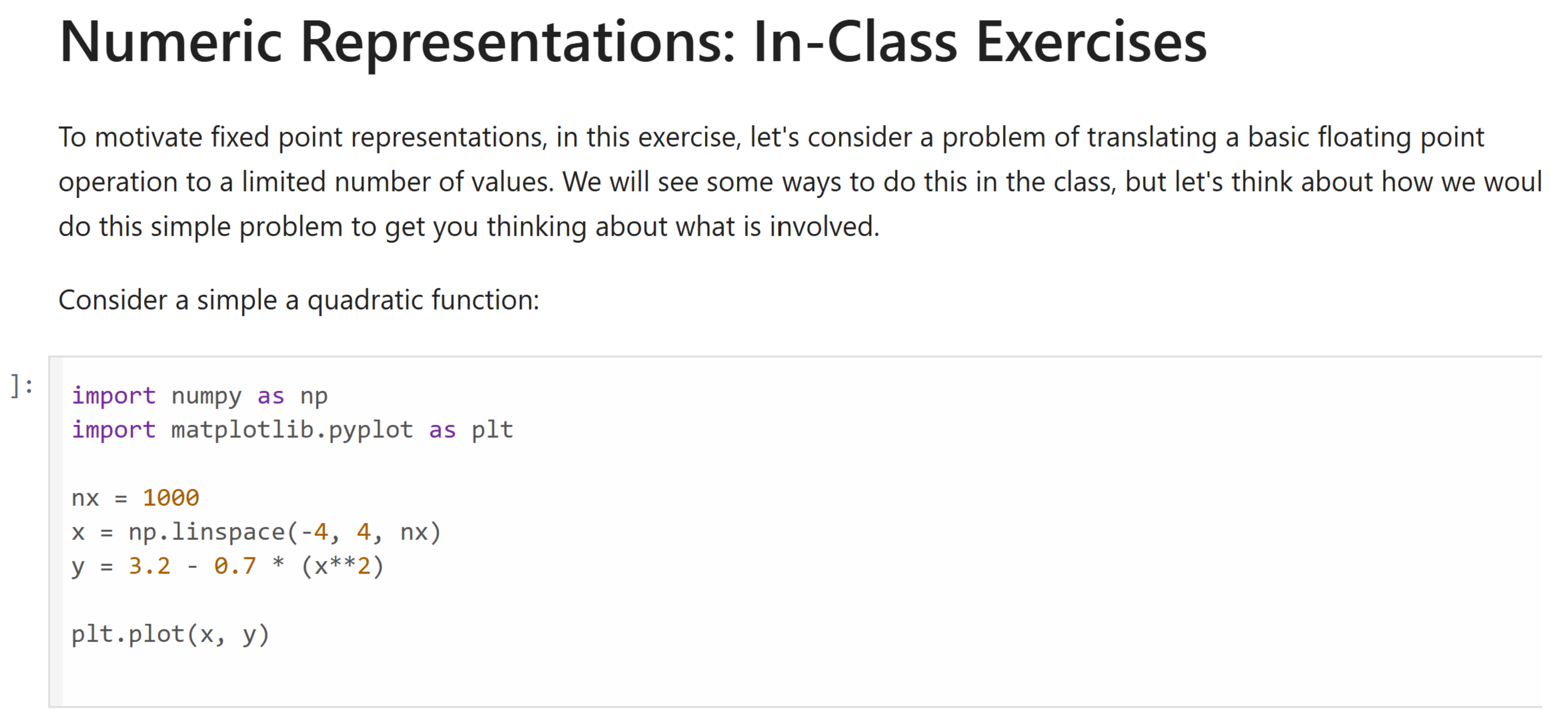
- ❑ Integer 32 × 32-bit multiplier
  - Implementation requires just the product: 64-bit result
- ❑ FP32 × FP32 multiplier
  - Significand multiply: 24×24 → 48-bit product (including hidden bit).
  - Exponent path: add exponents, subtract bias, detect overflow/underflow.
  - Normalization: possible shift + exponent adjust.
  - Rounding: guard/round/sticky bits, increment logic, possible renormalization.
  - Special cases: zeros, subnormals, infinities, NaNs, sign rules.

FP multiplication typically takes 2 to 4 × complexity

Outline

Motivating In-Class Exercise

❑ Try the following jupyter notebook



Numeric Representations: In-Class Exercises

To motivate fixed point representations, in this exercise, let's consider a problem of translating a basic floating point operation to a limited number of values. We will see some ways to do this in the class, but let's think about how we woul do this simple problem to get you thinking about what is involved.

Consider a simple a quadratic function:

```
]:  import numpy as np
    import matplotlib.pyplot as plt

    nx = 1000
    x = np.linspace(-4, 4, nx)
    y = 3.2 - 0.7 * (x**2)

    plt.plot(x, y)
```

Integers vs. Floating Point

- Integers:
  - Pro: Easy arithmetic. No extra bits for exponent
  - Cons: Do not natively handle fractional components.
  - Cons: Limited dynamic range
- Floating point:
  - Pro: Large dynamic range, handles fractional components
  - Cons: Arithmetic is harder, some loss of precision to store exponent
- Is there a way to handle fractions without increased computation overhead?

Fixed Point

- Floating point: use a mantissa and exponent:

$$x = (-1)^s\, m\, 2^e$$

  - Both mantissa and exponent are variables
  - Gives large dynamic range, but makes arithmetic more complex

- Fixed point: Key idea: Fix the exponent

$$x = q\, 2^{-n}$$

  - q is a standard integer with b bits
  - Scale factor 2^{-n} is fixed, not variable. Does not need to be stored or manipulated
  - Allows fractional values.

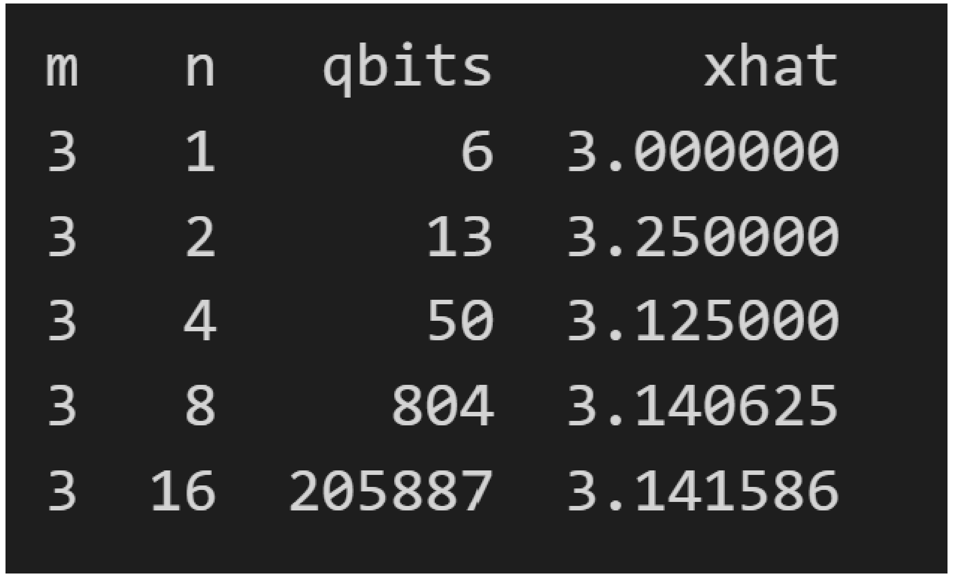
Fixed Point Q Notation

- ❑ $Qw.n$ and $UQw.n$: Standard fixed-point representation
  - ◦ w: total number of bits
  - ◦ n : number of fractional bits
  - ◦ $Qw.n$ = signed values, $UQw.n$: unsigned values
- ❑ Each element $Qw.n$ is simply a w bit signed or unsigned integer, q
  - ◦ q is called the integer representation
- ❑ Corresponds to a real-valued number:

$$x = float(q) = q2^{-n}$$

Examples

- Take $x_0 = \pi$
- Find closest approximation with:
  - $m = 3$ and n varies
- Table shows:
  - q value (qbits): integer representation
  - Floating point value: $\hat{x} = q2^{-n}$
- We see as n increases:
  - More accurate representation



```
m    n    qbits       xhat
3    1        6   3.000000
3    2       13   3.250000
3    4       50   3.125000
3    8      804   3.140625
3   16   205887   3.141586
```

Properties

- ❑ Signed values: If $q \in Qw.n$
  - There are $m = w - n$ integer bits (including one sign bit)
  - Range is $x \in \left[-2^{m-1}, 2^{m-1}\right),\ m = w - n$
- ❑ Unsigned values: If $q \in Qw.n$
  - There are $m = w - n$ integer bits
  - Range is $x \in [0, 2^m),\ m = w - n$
- ❑ Resolution: Minimum positive number: 2^{-n}
- ❑ Dynamic range:
  - $Qw.n\ :\ \mathrm{D} = \log_2\left(\frac{2^{m-1}}{2^{-n}}\right) = m + n - 1 = w - 1$ bits
  - $Uw.n : \mathrm{D} = \log_2\left(\frac{2^{m}}{2^{-n}}\right) = m + n = w$ bits

Addition and Type Casting Operations

- Addition:
  - Suppose that q_1 and q_2 are $Qw.n$ bit representations of $x_1 = q_1 2^{-n}$ and $x_2 = q_2 2^{-n}$
  - Sum is: $x = x_1 + x_2 = 2^n(q_1 + q_2) = 2^n$ with $q = q_1 + q_2$
  - Addition of numeric values = Addition of the integer representations

- Changing format:
  - Let q_1 is the $Qw_1.n_1$ representation of $x = q_1 2^{-n_1}$
  - Suppose we wish to find the representation q_2 in $Qw_2.n_2$
  - If $n_1 \geq n_2$ then: $q_2 = Truncate(\, q_1 \gg (n_1 - n_2), w_2)$ Right shift and take w_2 bits
  - If $n_1 \leq n_2$ then: $q_2 = Truncate(\, q_2 \ll (n_2 - n_1), w_2)$ Right shift and take w_2 bits

Multiplication

- Multiplication with same format
  - Suppose that q_1 and q_2 are $Qw.n$ bit representations of $x_1 = q_1 2^{-n}$ and $x_2 = q_2 2^{-n}$
  - $x = x_1 x_2 = 2^{2n} q_1 q_2 = 2^n q$ with $q = Truncate(\, q_1 q_2 \gg n, \;\; w\,)$
  - Multiplication of the numerical values = multiplication of integer values with $\gg n$
- Multiplication without overflow
  - To avoid overflow, make product width $w' = 2w - n$
- Multiplication with different formats
  - If q_1 and q_2 are $Qw_1.n_1$ and $Qw_2.n_2$ representations of x_1 and x_2
  - Product is $q = q_1 q_2$ is a $Qw.n$ representation of $x_1 x_2$ with:

    $$w = w_1 + w_2, \qquad n = n_1 + n_2$$
  - Can downshift this result

Summary

	Numeric operation	Input formats	Equivalent fixed point operation	Output format to avoid overflow
Addition with same format	$x = x_1 + x_2$	$q_1, q_2 \in Qw_0.n_0$	$q = q_1 + q_2$	$w = w_0 + 1$, $n = n_0$
Multiplication with same output precision	$x = x_1 x_2$	$q_1, q_2 \in Qw_0.n_0$	$q = (q_1 q_2) \gg n_0$	$w = 2w_0 - n_0$ $n = n_0$
Multiplication with increased precision	$x = x_1 x_2$	$q_1 \in Qw_1.n_1$, $q_2 \in Qw_2.n_2$	$q = q_1 q_2$	$w = w_1 + w_2$, $n = n_1 + n_2$
Reduce precision	$x_2 = x_1$ with lower precision	$q \in Qw_1.n_1$	$q_2 = q_1 \gg (n_2 - n_1)$	$w_2 = w_1 - n_2 + n_1$
Increase precision	$x_2 = x_1$ with higher precision	$q \in Qw_1.n_1$	$q_2 = q_1 \ll (n_1 - n_2)$	$w_2 = w_1 - n_2 + n_1$

Example Problem

Question 7. Fixed-point linear approximation

You wish to approximate the floating-point equation

$$y = ax + b$$

using Qm.n fixed-point arithmetic.

Let $a = 0.3125$, $b = -1.75$, and choose the format Q3.4 (3 integer bits including sign, 4 fractional bits).

a. Convert a and b into their Q3.4 integer representations.
b. Write SystemVerilog code that computes an approximation of y using only Q3.4 arithmetic.

Solution

Solution: Part a: Convert coefficients to Q3.4. In Q3.4, a real number x is represented as

$$x_{\text{int}} = \text{round}(x \cdot 2^4).$$

- $a = 0.3125$: $a_{\text{int}} = 0.3125 \cdot 16 = 5$
- $b = -1.75$: $b_{\text{int}} = -1.75 \cdot 16 = -28$

Part b: SystemVerilog implementation

```
logic signed [7:0] aint = 5;      // Q3.4 representation of a
logic signed [7:0] bint = -28;    // Q3.4 representation of b
logic signed [7:0] xint;          // Q3.4 input
logic signed [7:0] yint;          // Q3.4 output

yint = ((aint * xint) >>> 4) + bint;
```

The multiplication `aint * xint` produces a Q6.8 intermediate. Shifting right by 4 bits returns the result to Q3.4 format.

Comparison

Property	32-bit integer $I(32)$	32-bit floating point FP32	32–bit fixed point $Qm.n$, $m+n=32$
Complexity	Easy	Moderately hard	Easy
Can handle fractional values	No	Yes	Yes
Dynamic range	Low	High	Low

Outline

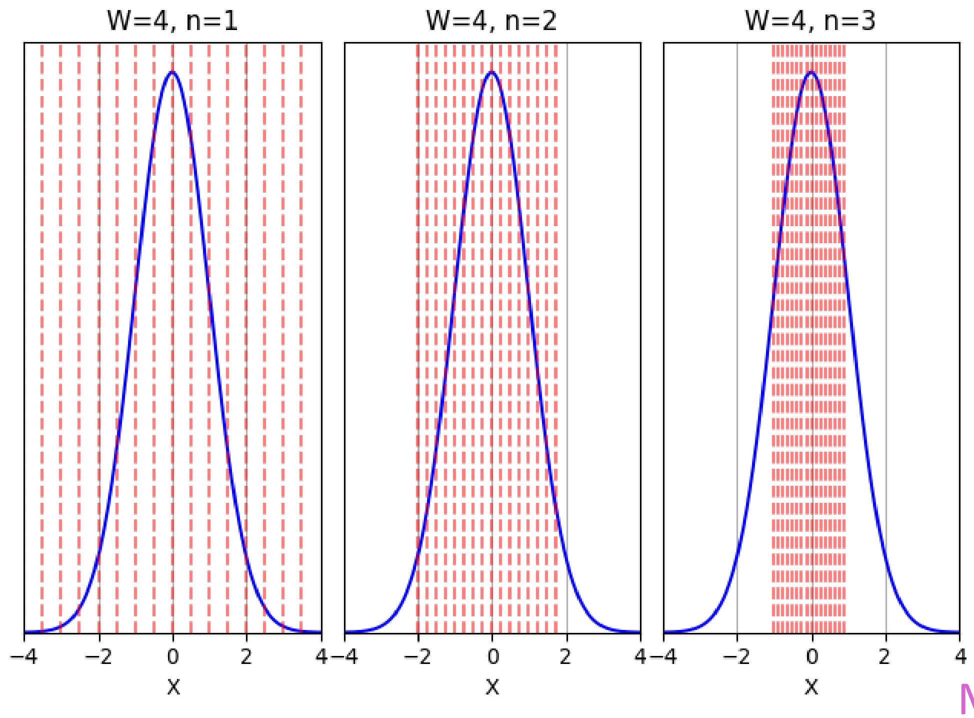
- ❑ Floating Point Numbers
- ❑ Fixed Point Numbers
- ❑ Optimal Scaling of FixP Numbers
- ❑ Converting Floating Point Algorithms to Fixed Point

Selecting the Precision

- ❑Suppose we have a real-valued number x
- ❑Want to represent as q in $Qw.n$
- ❑How do we select w and n?
- ❑Total width w determined by storage and complexity
- ❑How do we select n ?

Probabilistic Analysis

- ❑Let X = variable to quantize
- ❑Model X as a random variable
  - The HW will see different values over time
- ❑Example to right: $X \sim N(0,1)$
  - Blue = PDF of X
  - Fixed W = total bitwidth
  - Red lines = Qunatization bins for $Qw.n$
- ❑Selecting n
  - Tradeoff of overflow and precision



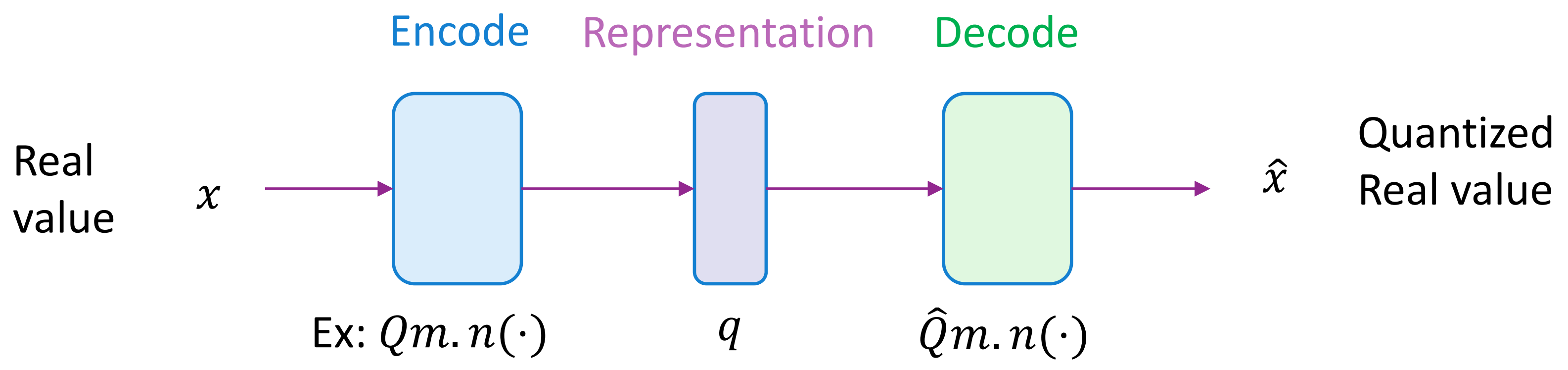
More integer bits

More fractional bits

Minimal overflow
Low precision

High overflow
High precision

Encoding, Representation, and Decoding



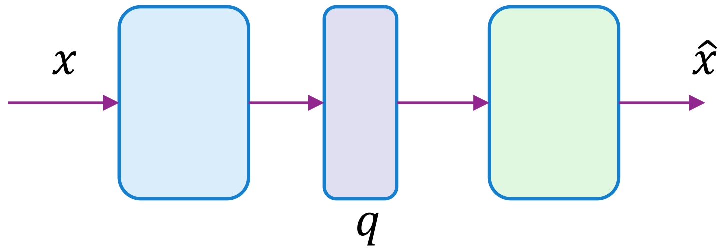
- Analyze any number class with three components: Encode, representation, decoding
- For fixed-point, we use the notation:
  - $q = Qm.n(x)$: Encodes a real number x to an integer representation q
  - $\hat{x} = \hat{Q}m.n(q)$: Decodes an integer representation back to a real number

Mean Squared Error

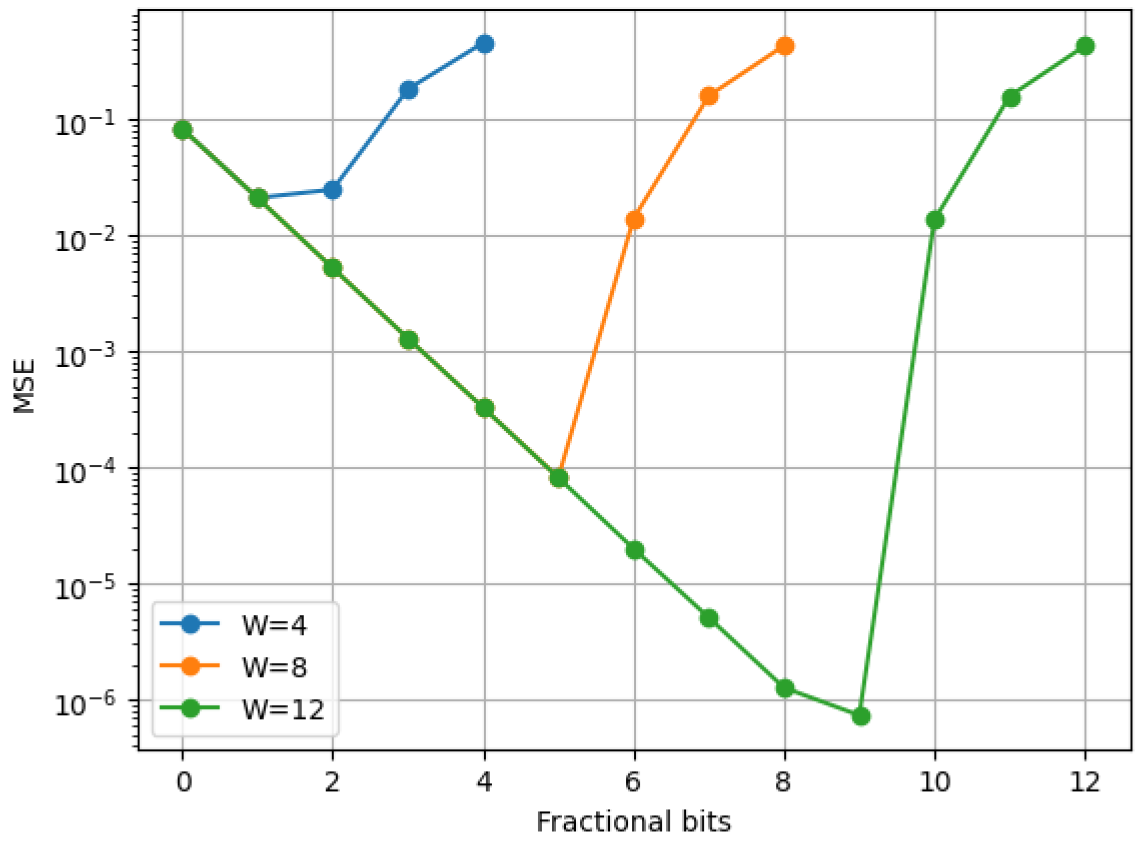
- Consider some encoding and decoding:
  - $q = Qm.n(x)$ and $\hat{x} = \hat{Q}m.n(q)$
- Suppose that real value can be modeled probabilistically
  - Has a probability density function $p(x)$
- Then we can measure the mean squared error:

$$MSE = \mathbb{E}\left((x - \hat{x})^2 \right) = \int p(x)(x - \hat{x})^2 \, dx$$

  - Measures the average error
  - Includes error to both precision and overflow
- We can then select n to minimize MSE
- Assumes we have a PDF of X
  - That is, we know what values we are expecting



Optimal Selection of n



- Example to the left: $X \sim N(0,1)$
- For each W we try different N
- Initially MSE decreases with N
  - Higher precision
- But then MSE increases
  - Overflow
- Optimal value is a few bits below W

Outline

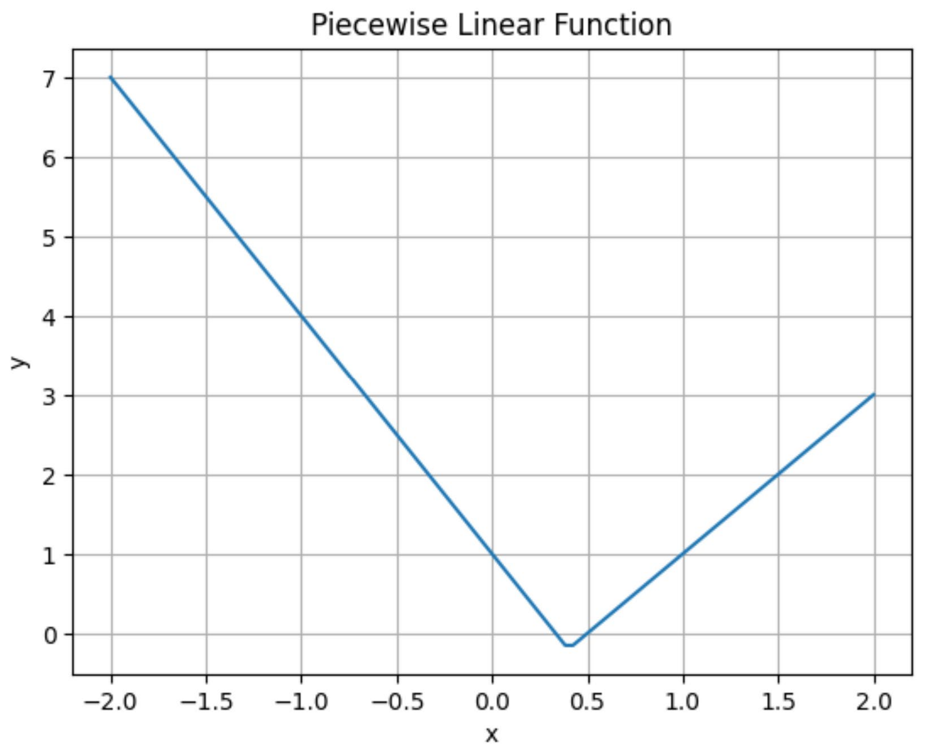
- ❑ Floating Point Numbers
- ❑ Fixed Point Numbers
- ❑ Optimal Scaling of FixP Numbers
- ❑ Converting Floating Point Algorithms to Fixed Point

Example: Piecewise Function

- See Jupyter notebook
- Consider piecewise linear function:

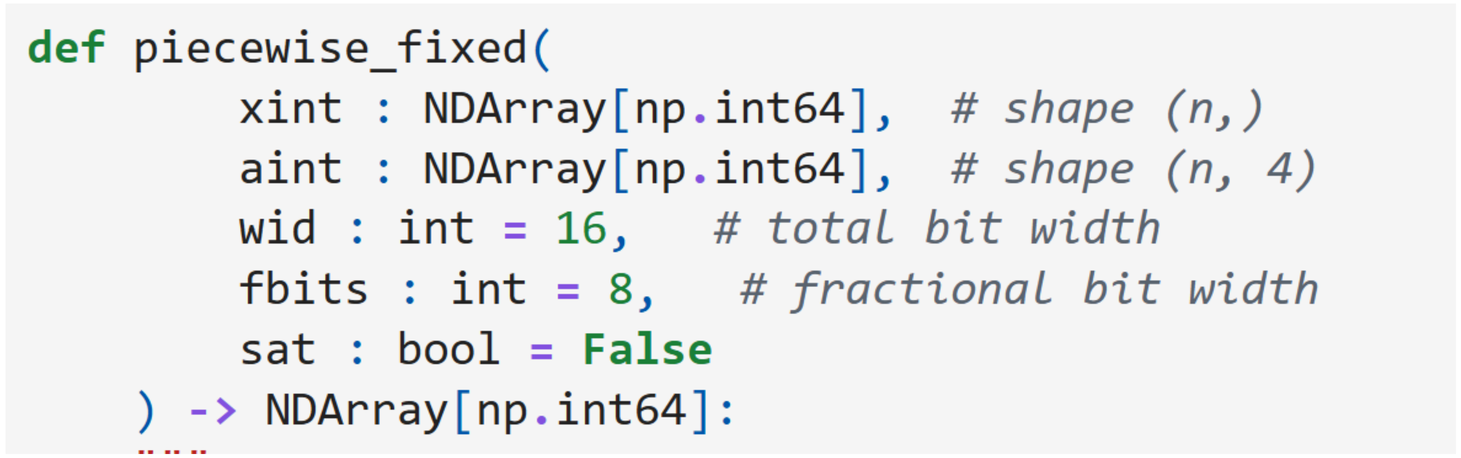
$$y = \max\{\, a_0 x + a_1, a_2 x + a_3\}$$

- How do we convert this to floating point



Python Model

- ❑Create a python model
- ❑Function has
  - Width and fractional bits
- ❑Inputs are integer representations

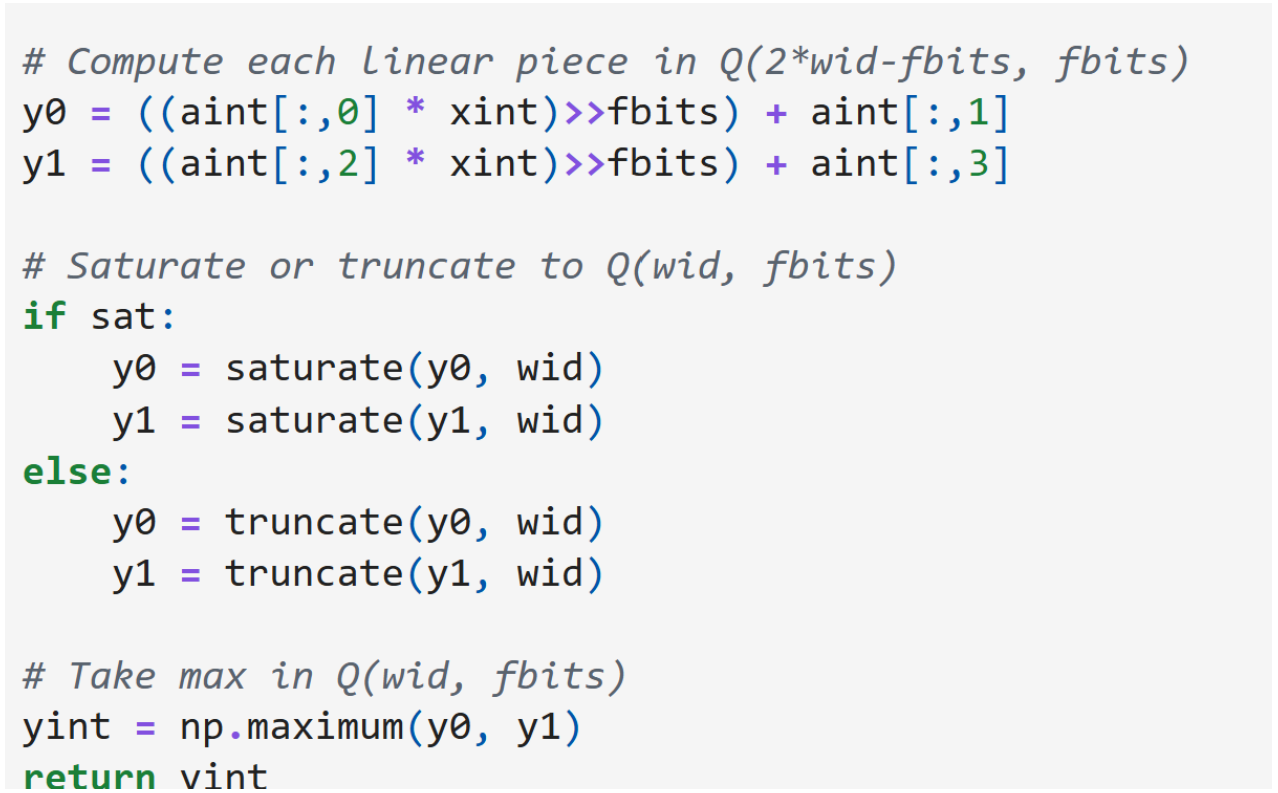


```
def piecewise_fixed(
        xint : NDArray[np.int64],  # shape (n,)
        aint : NDArray[np.int64],  # shape (n, 4)
        wid : int = 16,    # total bit width
        fbits : int = 8,    # fractional bit width
        sat : bool = False
    ) -> NDArray[np.int64]:
```

- ❑Variable stored as np.int64
  - Enables simply python processing
  - But actual numbers will only use smaller number of bits

Python Implementation

- ❑Direct operations using FP rules
- ❑Can truncate or saturate



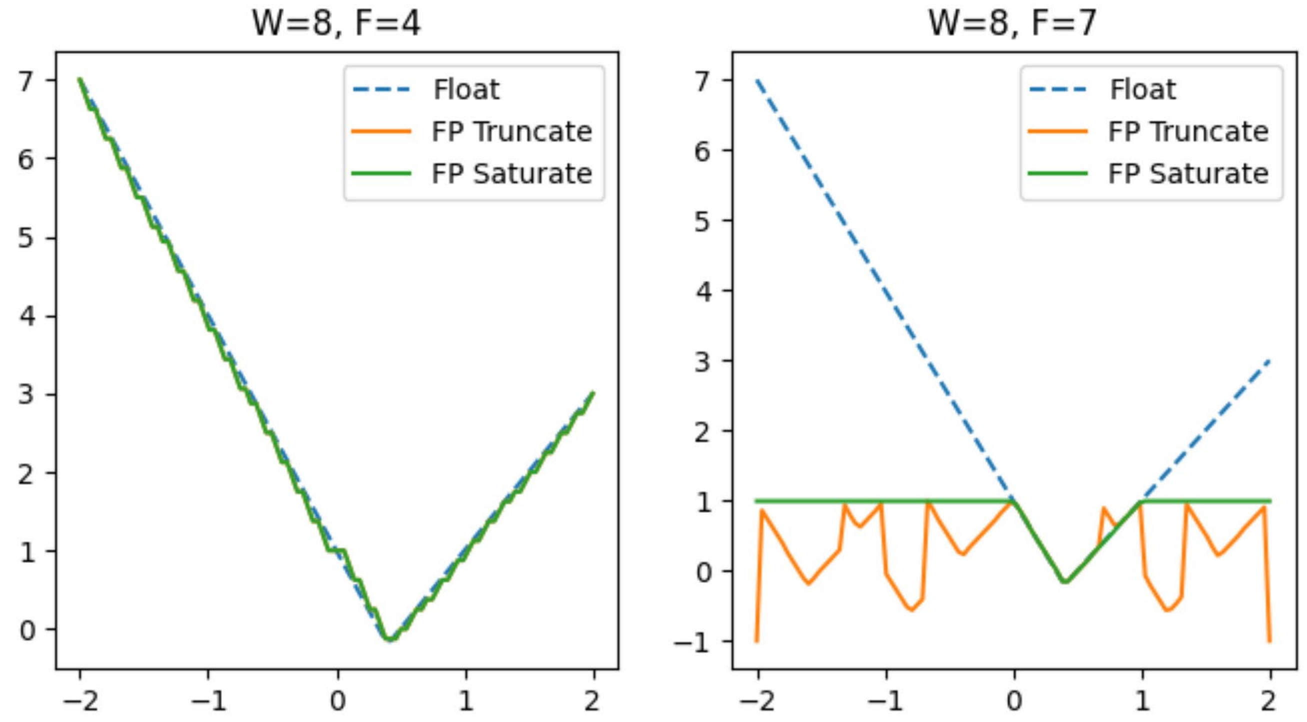
```
# Compute each linear piece in Q(2*wid-fbits, fbits)
y0 = ((aint[:,0] * xint)>>fbits) + aint[:,1]
y1 = ((aint[:,2] * xint)>>fbits) + aint[:,3]

# Saturate or truncate to Q(wid, fbits)
if sat:
    y0 = saturate(y0, wid)
    y1 = saturate(y1, wid)
else:
    y0 = truncate(y0, wid)
    y1 = truncate(y1, wid)

# Take max in Q(wid, fbits)
yint = np.maximum(y0, y1)
return yint
```

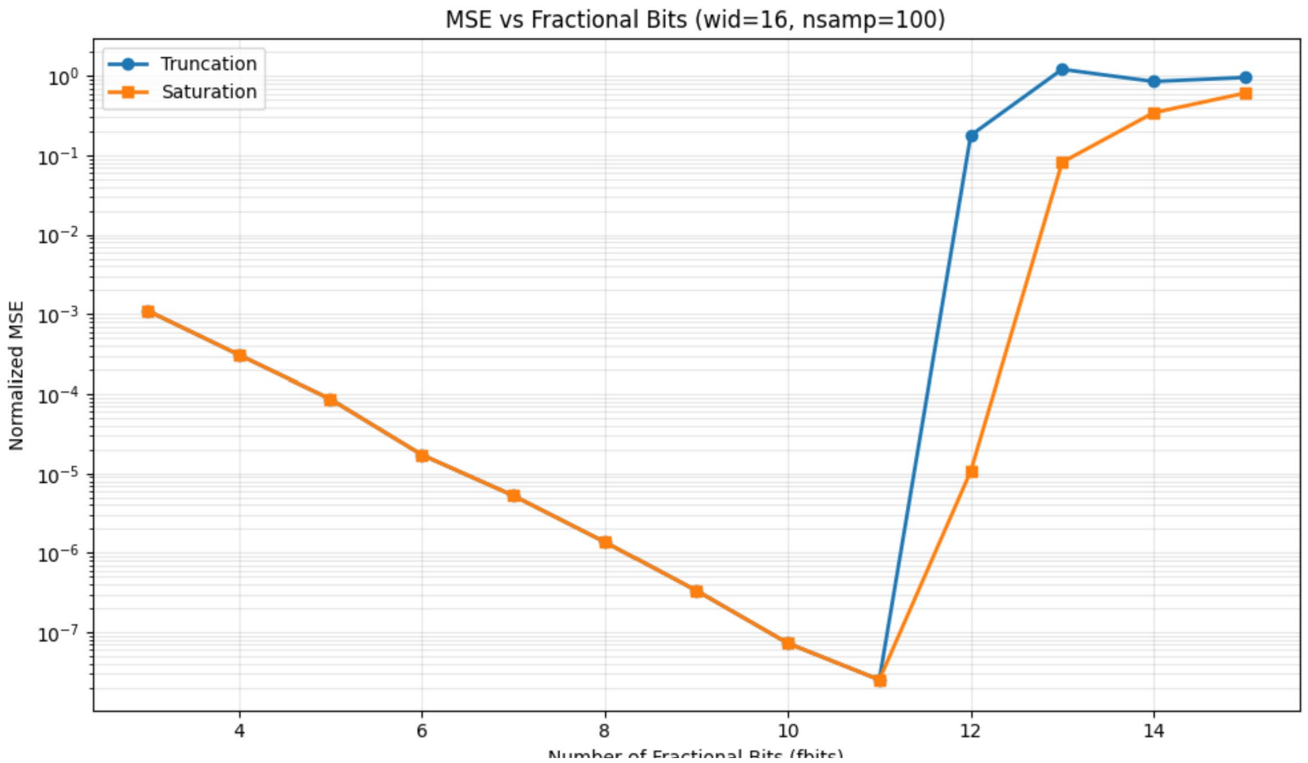
Quantization Examples

- For $F = 4$ fractional bits
  - Model roughly matches
  - But can see quantization error
- With $F = 7$ bits
  - Values begin to overflow
  - Saturation implementation clips
  - Truncate gives wrap-around values



Optimizing the MSE

- Initially increase F:
  - Increases precision
  - Lower MSE
- But eventually
  - Reduces number of integer bits
  - Begins to overflow
  - Fast increase in error



System Verilog Module Declaration

- ❑Define width parameters
- ❑Integer inputs
- ❑Optional saturation logic

```
module piecewise_fixed #(
    parameter int WID      = 16,    // total bit width
    parameter int FBITS    = 8,     // fractional bits
    parameter bit SATURATE = 0      // compile-time saturation flag
)(
    input  logic                      clk,

    // Inputs in Q(WID, FBITS)
    input  logic signed [WID-1:0]      x,
    input  logic signed [WID-1:0]      a0,
    input  logic signed [WID-1:0]      a1,
    input  logic signed [WID-1:0]      a2,
    input  logic signed [WID-1:0]      a3,

    // Output in Q(WID, FBITS)
    output logic signed [WID-1:0]      y
);
```

Implementation

- Register inputs (not shown)
- Compute y_0 and y_1 in high precision
- Optionally saturate
- Final max

```
// Compute full precision results
always_comb begin
    y0_full = ((a0_r * x_r) >>> FBITS) + a1_r;
    y1_full = ((a2_r * x_r) >>> FBITS) + a3_r;
end
```

```
generate
    if (SATURATE) begin : gen_sat
        always_comb begin
            if (y0_full > MAXV)      y0 = MAXV;
            else if (y0_full < MINV) y0 = MINV;
            else                     y0 = y0_full;

            if (y1_full > MAXV)      y1 = MAXV;
            else if (y1_full < MINV) y1 = MINV;
            else                     y1 = y1_full;
        end
    end else begin : gen_wrap
        always_comb begin
            y0 = y0_full;
            y1 = y1_full;
        end
    end
endgenerate
```

```
// Final max
always_comb begin
    y = (y0 > y1) ? y0 : y1;
end
```