

Unit 2

Numerical Representations

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 approximations to reality?

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
- ❑ Performance often remains excellent
 - With judicious choice of approximation techniques

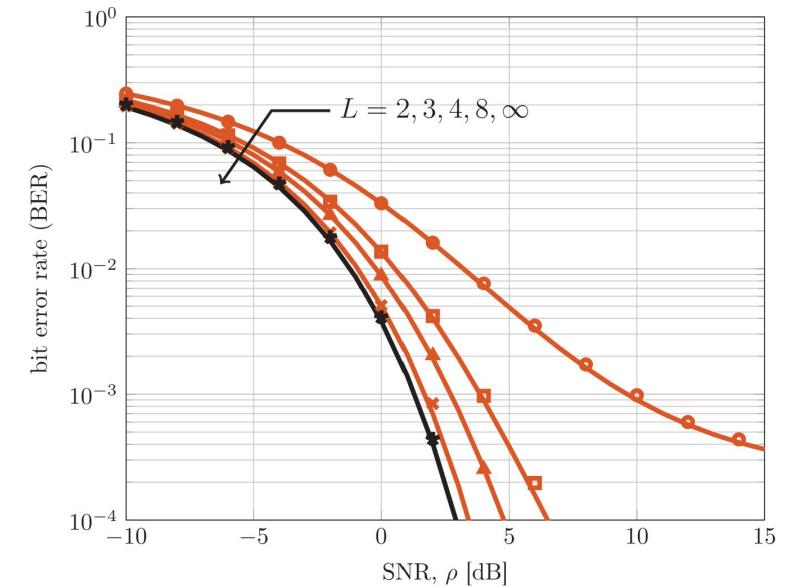
Model	Academic Benchmarks		
	Average Score	Leaderboard V1	Leaderboard V2
Llama-3.1-8B-Instruct	50.84	74.06	27.62
GPTQ (Frantar et al., 2022)	49.82	73.11	26.53
AWQ (Lin et al., 2024a)	50.05	72.69	27.40
Llama-3.1-70B-Instruct	62.93	84.20	41.66
GPTQ (Frantar et al., 2022)	62.18	83.77	40.58
AWQ (Lin et al.,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

💡 [hugging-quantz/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

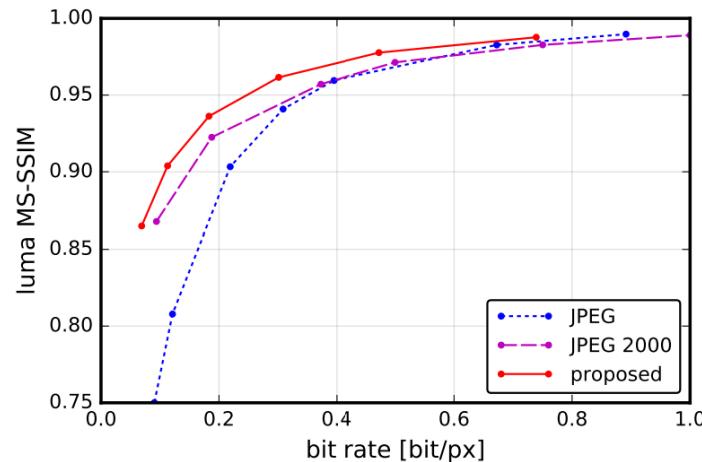
Jacobsson, S., Durisi, G., Coldrey, M., Gustavsson, U., & Studer, C. (2017). Throughput analysis of massive MIMO uplink with low-resolution ADCs. *IEEE Transactions on Wireless Communications*, 16(6), 4038-4051.

Compression Research at NYU

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



Jona Ballé, NYU



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

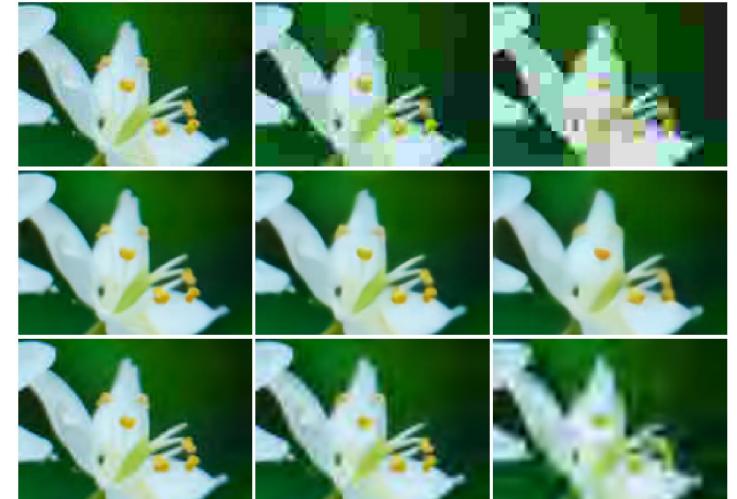
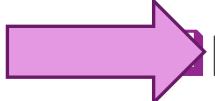


Figure 6: Cropped portion of an image compressed at three different bit rates. Middle row: the proposed method, at three different settings of λ . Top row: JPEG, with three different quality settings. Bottom row: JPEG 2000, with three different rate settings. Bit rates within each column are matched.

Learning Objectives

- ❑ Represent numbers in various formats: **Integers, floating point, fixed point**
- ❑ Implement arithmetic operations efficiently in these format
- ❑ 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

-  Integers, Truncation, and Overflows
- Floating Point Numbers
- Fixed Point Numbers
- Optimal Scaling of FixP Numbers
- Converting Floating Point Algorithms to Fixed Point

Integers

□ We will use notation

- $I(b)$: Signed integer with b bits in two complement
- $U(b)$: Unsigned integer with b bits

□ Range:

- $x \in I(b) \Rightarrow x \in [-2^{b-1}, 2^{b-1})$
- $x \in U(b) \Rightarrow x \in [0, 2^b - 1)$

□ Example: With $b = 8$ bits

- Range of $I(b) = [-128, 127)$
- Range of $U(b) = [0, 255)$

Truncation

```
logic signed [W-1:0] a = A;
```

- ❑ Occurs when a variable is assigned a value out of range
- ❑ In above code we need $A \in [-2^{W-1}, 2^{W-1})$
- ❑ When A is outside this range, SystemVerilog will **truncate**:
 - Take the W LSBs of A
 - Sign extend if necessary
- ❑ Equivalent to $A \bmod 2^W$

Sample Problem

Question 1. Truncation

Truncate the following numbers to 4 bits in both unsigned and signed representations. Provide the decimal value of the truncated numbers in each case.

- a. 9
- b. 15
- c. 23
- d. -1
- e. -12

Can see problem on LLM auto grader

Solution using Binary Representations

The first method is to convert the decimal number to binary, trucate to 4 bits, and then convert back to decimal.

- | | |
|---------|---|
| (a) 9 | (a) Unsigned: $(1001)_2 = 9$. Signed: $(1001)_2 = -7$. |
| (b) 15 | (b) Unsigned: $(1111)_2 = 15$. Signed: $(1111)_2 = -1$. |
| (c) 23 | (c) Unsigned: $(0111)_2 = 7$. Signed: $(0111)_2 = 7$. |
| (d) -1 | (d) Unsigned: $(1111)_2 = 15$. Signed: $(1111)_2 = -1$. |
| (e) -12 | (e) Unsigned: $(0100)_2 = 4$. Signed: $(0100)_2 = 4$. |

Solution using Modulo Arithmetic

You can also use modular arithmetic to compute the truncated values. For unsigned numbers, we compute the value modulo $2^4 = 16$. So, we find the remainder when dividing by 16. For signed numbers, we compute the value modulo $2^4 = 16$ and then adjust to the signed range $[-8, 7]$.

- | | |
|---------|---|
| (a) 9 | (a) Unsigned: $9 \bmod 16 = 9$. Signed: $9 \bmod 16 = 9 - 16 = -7$. |
| (b) 15 | (b) Unsigned: $15 \bmod 16 = 15$. Signed: $15 \bmod 16 = 15 - 16 = -1$. |
| (c) 23 | (c) Unsigned: $23 \bmod 16 = 7$. Signed: 7 is in range, so 7. |
| (d) -1 | (d) Unsigned: $-1 \bmod 16 = 15$. Signed: $15 - 16 = -1$. |
| (e) -12 | (e) Unsigned: $-12 \bmod 16 = 4$. Signed: 4 is in range, so 4. |

Simulating Truncation in Python

```
def truncate(x, n, signed=True):
    mask = (1 << n) - 1
    x = x & mask
    if signed:
        I = (x >= (1 << (n - 1)))
        x -= (1 << n)*I
    return x
```

- ❑ This code works for numpy arrays
- ❑ Mask the n LSB
- ❑ Then sign extend if necessary

Saturation

❑ Truncation results in values far different from original

- Ex: Truncate 9 to 4 signed bits $\Rightarrow 9 - 16 = -7$

❑ Alternatively, can **saturate**

- Clip to max or min values before assignment
- Ex: Truncate 9 to 7
- Provides less approximation error
- But adds a bit of extra logic

❑ Example to right: $W_2 < W_1$

- Want to assign $b = a$

```
logic signed [W1-1:0] a;  
logic signed [W2-1:0] b;  
// Assign with truncation  
b = a; // lower W2 bits of a assigned to b  
  
// Assign with saturation  
if (a > (1 << (W2 - 1)) - 1) begin  
    b = (1 << (W2 - 1)) - 1;  
end else if (a < -(1 << (W2 - 1))) begin  
    b = -(1 << (W2 - 1));  
end else begin  
    b = a;  
end
```

Addition Overflow Rules

- ❑ Consider adding $c = a + b$
- ❑ If a and b are both signed or both unsigned:
 - No overflow requires $W(c) \geq \max(W(a), W(b)) + 1$
 - $W(x)$ = bit width of x
- ❑ If a is signed and b is unsigned,
 - No overflow requires $W(c) \geq \max(W(a), W(b)) + 2$
 - Example if $a = 7$ (4 bits signed), $b = 15$ (4 bits unsigned)
 - Sum is $c = 22$ which needs 6 bits signed.

```
logic signed [15:0] a;  
logic signed [15:0] b;  
logic signed [16:0] c1;  
logic signed [15:0] c2;  
c1 = a + b; // no overflow  
c2 = a + b; // potential overflow
```

Sample Problem

2. *Addition of multiple numbers:* Consider the addition

```
logic signed [7:0] a, b, c, d;  
logic signed [W-1:0] sum = a + b + c + d;
```

What value of W is required to ensure that no overflow occurs in the sum?

See solution in the autograder

Addition with Truncation

- ❑ Given a potential overflow, SV will:
 - Create a temporary variable with no overflow
 - Temporary variable has enough bits
 - Then it will truncate to lower bits
- ❑ Saturation must be implemented manually

```
logic signed [7:0] a;  
logic signed [10:0] b;  
logic signed [8:0] c;
```

```
c = a + b; // potential overflow
```

```
// SV implements with signed extension  
// then truncation  
logic signed [10:0] cfull;  
cfull = $signed(a) + $signed(b);  
c = cfull[8:0]; // Truncation
```

Multiplication Overflow

- Consider multiplying $c = a \cdot b$
 - a and b may be signed or unsigned

- To prevent overflow, we need:

$$W(c) \geq W(a)W(b)$$

```
logic signed [8:0] a;    // 9-bit signed
logic signed [12:0] b;   // 13-bit signed
logic signed [21:0] c1;  // 22-bit signed
logic signed [20:0] c2;  // 21-bit signed
```

- With overflow:
 - SV implements truncation

```
c1 = a * b; // No overflow
c2 = a * b; // Potential overflow
```

Summary

- ❑ Two integer types: $I(b)$ = signed integers $U(b)$ = unsigned
- ❑ Each has finite range
- ❑ Assignments outside that range result in **overflow**
- ❑ In case of overflow, system Verilog performs **truncation**
- ❑ Optionally, you can add code to explicitly **saturate**
 - Offers better performance but adds additional logic
- ❑ Arithmetic operations result in **bitwidth growth**
 - Addition of 2 numbers requires 1 extra bit
 - Multiplication of 2 numbers requires width = product of the two numbers

Outline

- ❑ Integers, Truncation, and Overflows
-  ❑ Floating Point Numbers
- ❑ Fixed Point Numbers
- ❑ Mapping Floating to Fixed Point in Python
- ❑ System Verilog Implementation

Limitations of Integers

- ❑ Consider integers $I(b)$ or $U(b)$
- ❑ Cannot handle fractional components
 - Ex: 7.9, -12.2
 - Rounding may cause excessive approximation error
- ❑ Limited range
 - Values overflow $\geq 2^b$ for $U(b)$ and $\geq 2^{b-1}$ for $I(b)$

Simple Floating Point

- ❑ Fixed point $FP(p, w)$

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

- ❑ Signed bit:

- $s = 0, 1$, One bit

- ❑ Exponent, w bits

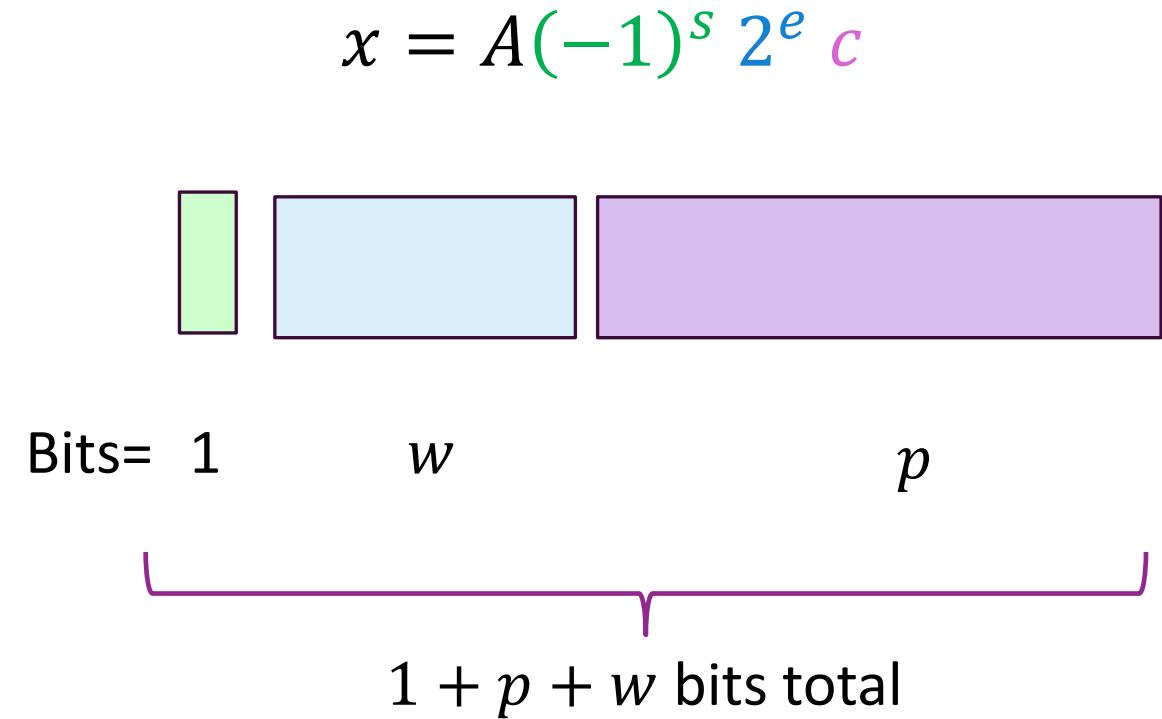
- $e \in U(w)$

- ❑ 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



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 $s = 0 \rightarrow$ positive. Exponent bits 000 $\rightarrow e = 0$. Mantissa bits $10000_2 = 16$. Value:

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

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

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

c. Sign bit $s = 0 \rightarrow$ positive. Exponent bits 101 $\rightarrow e = 5$. 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: $x = A(-1)^s c \cdot 2^e$

Dynamic Range

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

$$D = \frac{\max_{x>0} |x|}{\min_{x \neq 0} |x|}$$

- 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:

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

- 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:

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

- 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

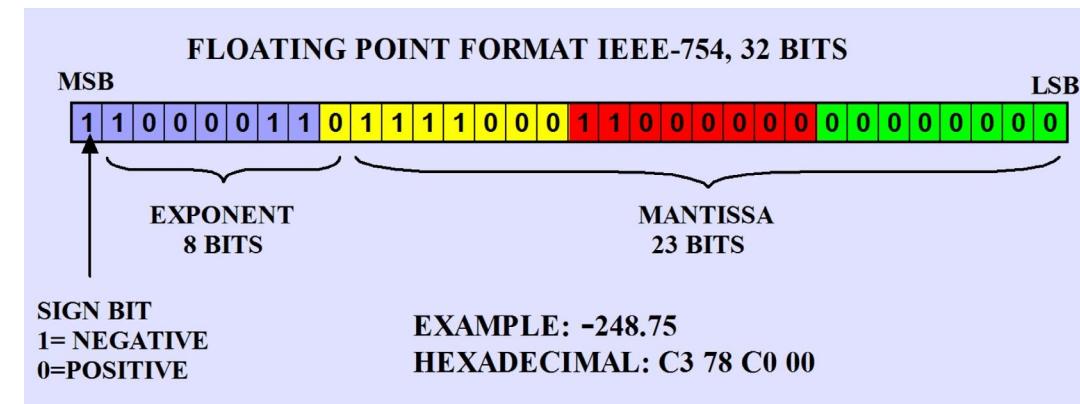
❑ 32-bit floating point:

- $p = 23, w = 8$
- 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

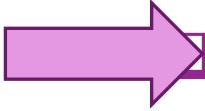
- Implementation requires just the product: 64-bit result

❑ FP32 \times FP32 multiplier

- Significand multiply: $24 \times 24 \rightarrow 48$ -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 \times complexity

Outline

- ❑ Integers, Truncation, and Overflows
- ❑ Floating Point Numbers
-  Fixed Point Numbers
- ❑ Mapping Floating to Fixed Point in Python
- ❑ System Verilog Implementation

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 would do this simple problem to get you thinking about what is involved.

Consider a simple 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**:

$$x = (-1)^s \text{ } m \text{ } 2^e$$

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

❑ **Fixed point:** Key idea: Fix the exponent

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

- 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

- $Qm.n$ and $UQm.n$: Standard fixed-point representation
 - m : number of integer bits
 - n : number of fractional bits
 - $Qm.n =$ signed values, $UQm.n$: unsigned values
- Each element $Qm.n$ is simply a $m + n$ integer, q
 - q is called the **integer representation**
- Corresponds to a real-valued number:

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

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

❑ Suppose that $q \in UQm.n$ or Type equation here.

❑ Total number of bits $b = m + n$

❑ Range:

- $q \in Qm.n \Rightarrow x \in [-2^{m-1}, 2^{m-1})$
- $q \in UQm.n \Rightarrow x \in [0, 2^m)$

❑ Resolution: Minimum positive number: 2^{-n}

❑ Dynamic range:

- $Qm.n : D = \log_2 \left(\frac{2^{m-1}}{2^{-n}} \right) = m + n - 1 = b - 1$ bits
- $Um.n : D = \log_2 \left(\frac{2^m}{2^{-n}} \right) = m + n = b$ bits

Arithmetic Operations

❑ Suppose that q_1 and q_2 are $Qm.n$ bit representations of $x_1 = q_1 2^{-n}$ and $x_2 = q_2 2^{-n}$

- Assume they are the same parameters m and n for now

❑ Addition:

- $x = x_1 + x_2 = 2^n(q_1 + q_2) = 2^n$ with $q = q_1 + q_2$
- Addition of numeric values = Addition of the integer representations

❑ Multiplication:

- $x = x_1 x_2 = 2^{2n} q_1 q_2 = 2^n q$ with $q = q_1 q_2 \gg n$
- Multiplication of the numerical values = multiplication of integer values with $\gg n$

❑ Same overflow rules as integers

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.

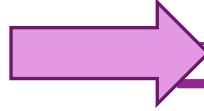
Type Casting

- ❑ Suppose that q_0 in the bit representation in $Qm_0.n_0$.
- ❑ We want to convert to $q_1 = Qm_1.n_1$
- ❑ Since $x = q_0 2^{-n_0} = (q_0 2^{n_1-n_0}) 2^{-n_1} = q_1 2^{-n_1}$ where $q_1 = q_0 2^{n_1-n_0}$
- ❑ Implementable with left shift by $n_1 - n_0$ or right shift by $n_0 - n_1$
- ❑ For $q_1 \geq 0$, will overflow when $q_1 \geq 2^{m_1+n_1-1}$
- ❑ Overflow can only occur when $m_1 < m_0$:
 - Why? Suppose $m_1 \geq m_0$
 - Since $q_0 \leq 2^{m_0+n_0-1}$, $q_1 = q_0 2^{n_1-n_0} \leq 2^{m_0+n_1-1} \leq 2^{m_1+n_1-1} \Rightarrow$ No overflow
- ❑ However changing n_1 or n_0 may change precision

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

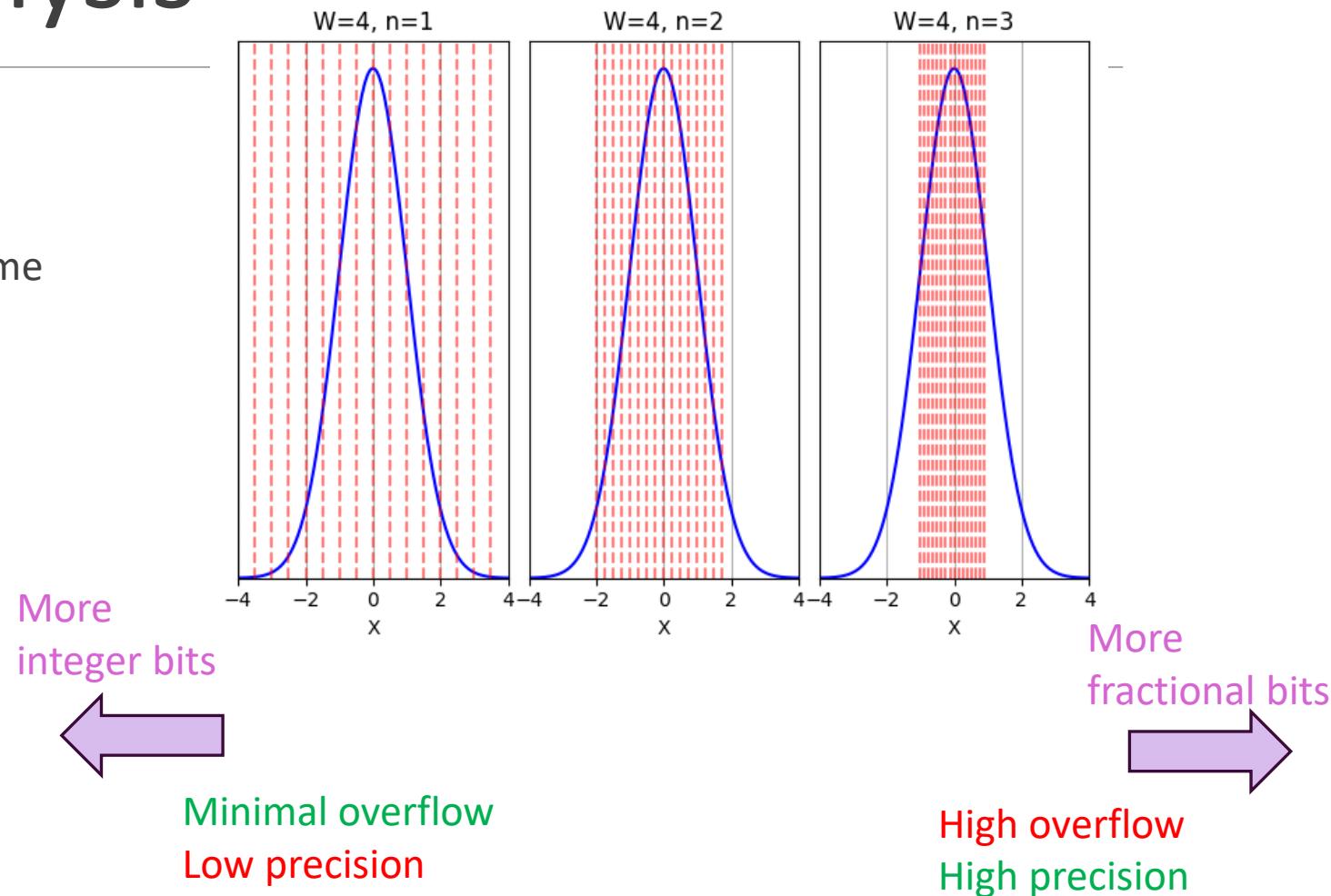
- ❑ Integers, Truncation, and Overflows
- ❑ Floating Point Numbers
- ❑ Fixed Point Numbers
-  ❑ Optimal Scaling of FixP Numbers
- ❑ Converting Floating Point Algorithms to Fixed Point

Selecting the Precision

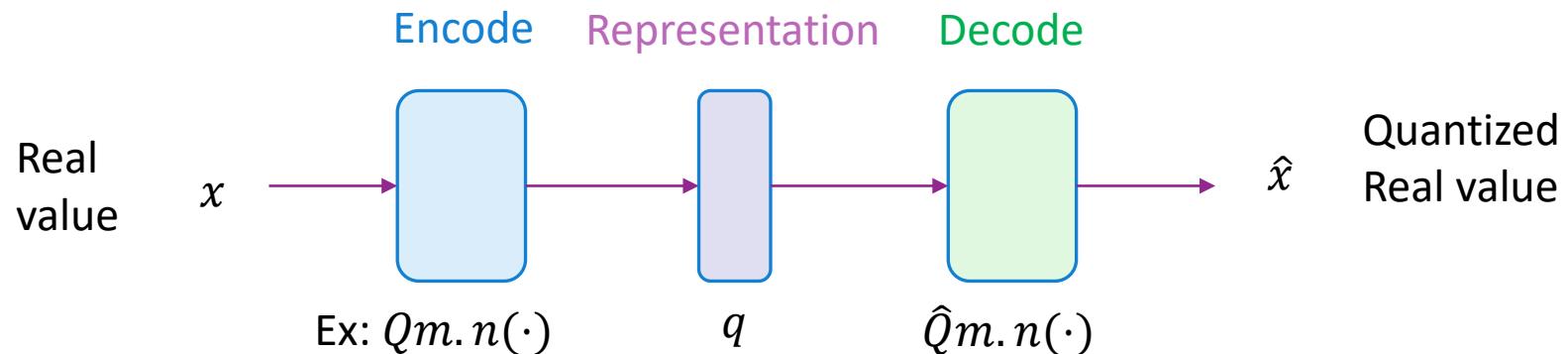
- ❑ Suppose we have a real-valued number x
- ❑ Want to represent as q in $Qm.n$
- ❑ How do we select m and n ?
- ❑ Total width $W = m + n$ determined by storage and complexity
- ❑ For a fixed width W we can vary n
- ❑ How do we select n ?

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 $W = m + n$ = total bitwidth
 - Red lines = Quantization bins for $Q_m.n$
- ❑ Selecting n
 - Tradeoff of overflow and 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:

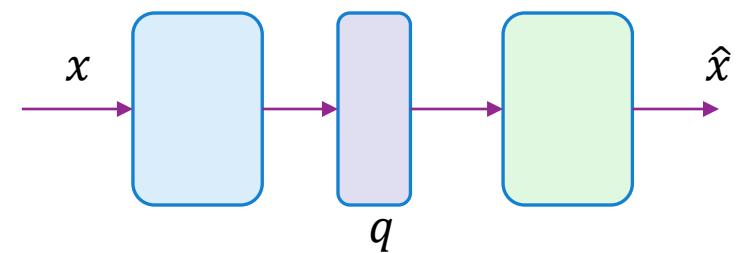
$$MSE = \mathbb{E}((x - \hat{x})^2) = \int p(x)(x - \hat{x})^2 dx$$

- Measures the average error
- Includes error to both precision and overflow

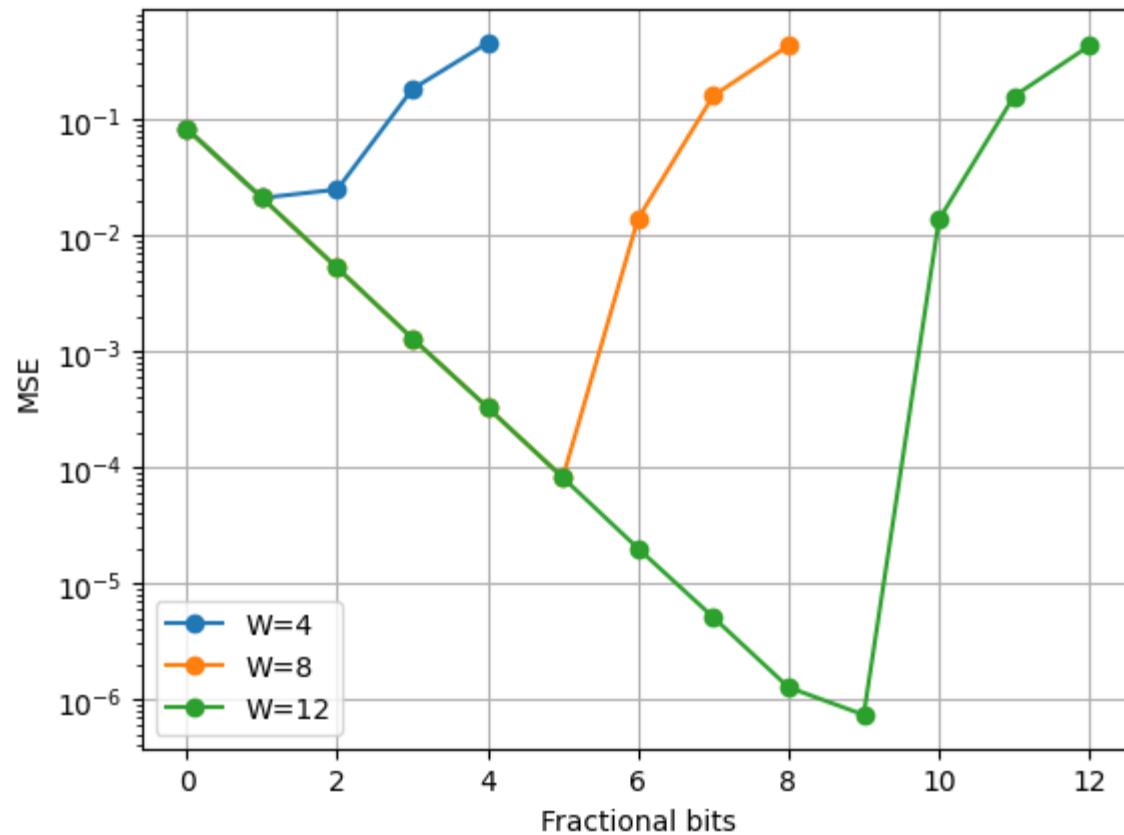
□ We can then select n to minimize MSE

□ Assumes we have a PDF of X

- That is, we know what values we are expecting

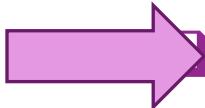


Optimal Selection of n



- ❑ 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

- ❑ Integers, Truncation, and Overflows
 - ❑ Floating Point Numbers
 - ❑ Fixed Point Numbers
 - ❑ Optimal Scaling of Fix Point Numbers
-  Converting Floating Point Algorithms to Fixed Point