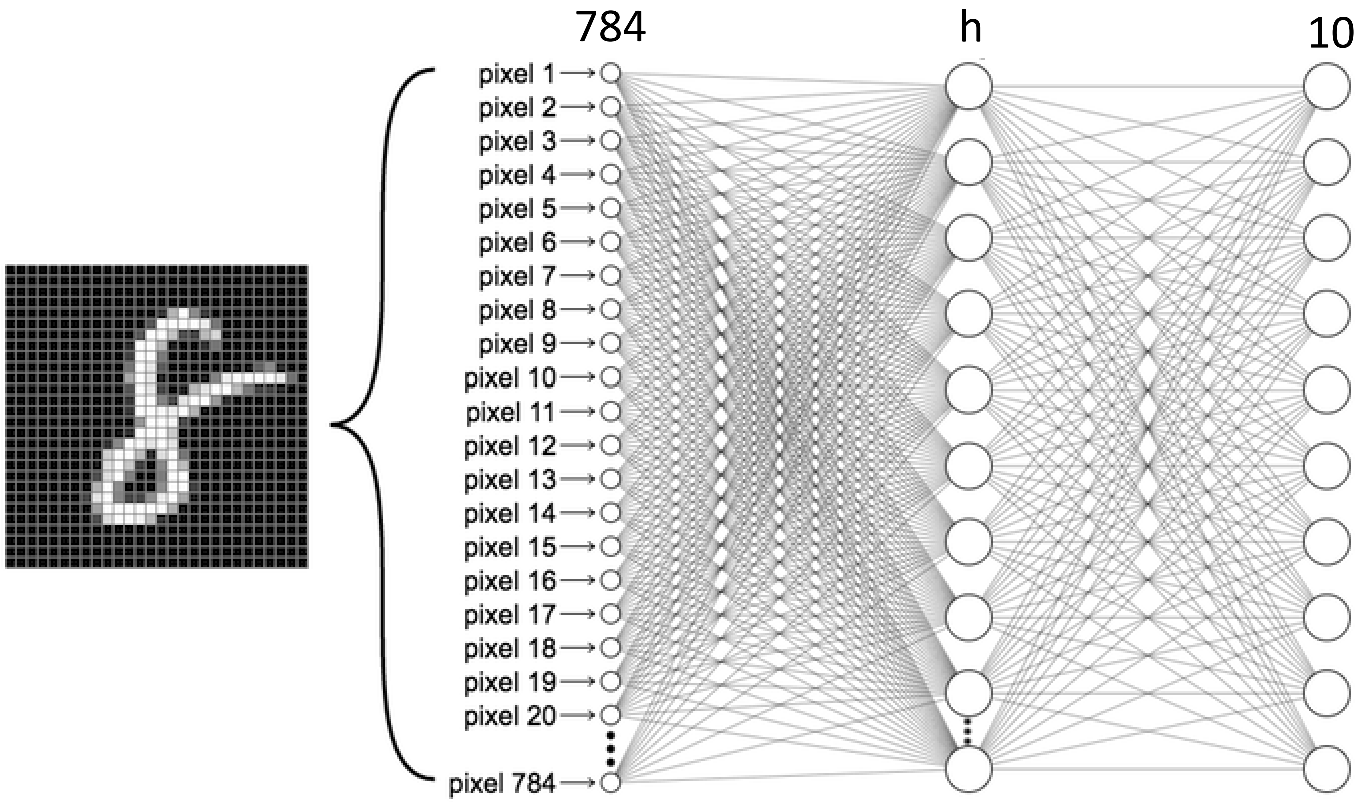
Floating Point for AI: An update from IEEE P3109

Andrew Fitzgibbon, Graphcore

Co-editor, IEEE P3109 Working Group

Understanding machine learning algorithms




```python
def ffn(W, x):
    ((W1,b1),(W2,b2)) = W
    t1 = W1 @ x + b1
    y1 = relu(t1)
    y2 = W2 @ y1 + b2
    return softmax(y2)
```

From https://github.com/aamini/introtodeeplearning/blob/master/lab2/Part1_MNIST.ipynb

Inside an AI model: Nomenclature

"**Weights**" W, a collection of arrays (tuple of tuples here)

Input x, e.g. an image flattened into an N-vector, or a "batch" of inputs, e.g. as an $N \times B$ matrix.

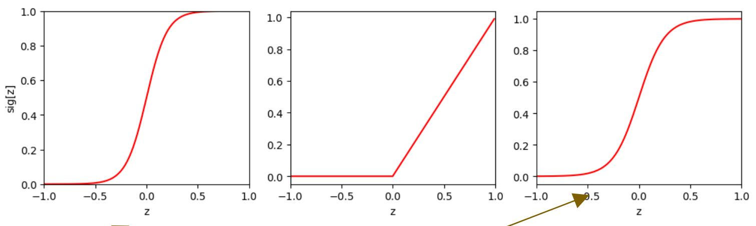
```python
def ffn(W, x):
    ((W1,b1),(W2,b2)) = W
    t1 = W1 @ x + b1
    y1 = relu(t1)
    y2 = W2 @ y1 + b2
    return softmax(y2)
```

Matrix multiply and add

[python uses "@" for m@tmul]

Intermediate results, a.k.a "activations"

Nonlinear "activation function"



Output:

vector of "probability",

e.g. size $10 \times B$ for batch of digits

$$s_i = \frac{e^{y_i}}{\sum_k e^{y_k}}$$

Transcendentals such as exp and tanh are common

More complex AI models

"Deep" MLP: W is a list of weights

```python
def ffn(W, x):
    (W0,b0) = W[0]
    x = W0 @ x + b0

    for (W1,b1) in W[1:]:
        y = relu(x)
        x = W1 @ y + b1

    return softmax(x)
```

for W_l in "layers":
$$x = f(W_l, x)$$

More complex AI models

Transformer: same "layers" loop, bigger numbers
$L = 32K, D_m = 16K, L \times D_m = 1GB$ Float16

Use of ∞

```
def transformer(W, input):
    L = input.shape[0]

    # Create mask: 0 to attend, -Inf to ignore
    mask = jnp.log(jnp.tril(jnp.ones((L, L))))

    # Start with token embeddings + positional encodings
    x = W.embeddings[input, :]      # L x Dm

    # Apply the transformer layers
    for Wl in W.layers:
        x = transformer_layer(Wl, x, mask)

    # And linearly project to output dimension
    return W.out_A @ x + W.out_b
```

"Deep" MLP: W is a list of weights

```
def ffn(W, x):
    (W0,b0) = W[0]
    x = W0 @ x + b0

    for (Wl,bl) in W[1:]:
        y = relu(x)
        x = Wl @ y + bl

    return softmax(x)
```

"layers"
$x = f(x)$

More complex AI models

Transformer: same "layers" loop, bigger numbers

$L = 32K, D_m = 16K, L \times D_m = 1GB$ Float16

```
def transformer(W, input):
    L = input.shape[0]

    # Create mask: 0 to attend, -Inf to ignore
    mask = jnp.log(jnp.tril(jnp.ones((L, L))))

    # Start with token embeddings + positional encodings
    x = W.embeddings[input, :]     # L x Dm

    # Apply the transformer layers
    for Wl in W.layers:
        x = transformer_layer(Wl, x, mask)

    # And linearly project to output dimension
    return W.out_A @ x + W.out_b
```

```
def transformer_layer(W, x, mask):
    # Layer-normalize embeddings
    t1 = standardize(x)
    t1 = W.p1A @ t1 + W.p1b

    # Multi-head self-attention
    for head in W.heads:
        # Project into this head's query/key space
        query = head.query @ t1 + head.qb        # L x Dk
        key = head.key @ t1 + head.kb            # L x Dk

        # Compute L x L attention matrix
        score = query @ key.T + mask             # L x L
        attn = softmax(tau * score)              # L x L

        value = head.value @ t1 + head.vb        # L x Dm
        self_attn = attn @ value                 # L x Dm

        x += self_attn                           # L x Dm

    # Layer-normalize embeddings
    t2 = standardize(x)
    t2 = W.p2A @ t2 + W.p2b   # L x Dm

    # Feedforward fully connected
    t2 = W.ffn1.A @ t2 + W.ffn1.b                # L x Dff
    t2 = relu(t2)
    t2 = W.ffn2.A @ t2 + W.ffn2.b                # L x Dm

    return x + t2
```

Divide by norm

Addition of $-\infty$

Still lots of m@muls

Inside an AI model.
That was “inference”.

Inside an AI model.

Model:

```python
def ffn(W, x):
    ((W1,b1),(W2,b2)) = W
    t1 = W1 @ x + b1
    y1 = relu(t1)
    y2 = W2 @ y1 + b2
    return softmax(y2)
```

Inference: (Using the model)

```python
def classify_digit(W, x) -> int:
    return argmax(ffn(W, x))
```

Training: (Building the model)

Given a set $\{x_i, l_i\}$ of pairs (image, label), we would like to find W which maximizes performance

$$W_{\text{trained}} = \underset{W}{\operatorname{argmax}} \sum_i \mathbb{I}[\text{classify}(W, x_i) = l_i]$$

But that is piecewise constant, so not amenable to gradient descent, so we maximize the output of the softmax

$$W_{\text{trained}} = \underset{W}{\operatorname{argmax}} \sum_i \text{ffn}(W, x_i)[l_i]$$

And in practice, minimize negative log:

$$W_{\text{trained}} = \underset{W}{\operatorname{argmin}} \sum_i -\log(\text{ffn}(W, x_i)[l_i])$$

Model:

```
def ffn(W, x):
    ((W1,b1),(W2,b2)) = W
    t1 = W1 @ x + b1
    y1 = relu(t1)
    y2 = W2 @ y1 + b2
    return softmax(y2)
```

Loss:

```
def loss(W, x, l):
    z = ffn(W,x)
    return -jnp.log(z[l])
```

Gradient:

Same mix of operations: matmul*, exp, nonlinearities

Primary concerns:

- Efficient use of FLOPs
- Minimize peak memory
- Minimize memory transfers

Real models typically run on multi-GPU clusters, but essentially the same concerns: FLOPs, Memory, Bandwidth

* Each matmul in “forward” computation generally yields two in “backward” pass

```
def loss_and_grads(W, x, l):
    ((W1, b1), (W2, b2)) = W
    t1 = W1 @ x + b1
    y1 = relu(t1)
    y2 = W2 @ y1 + b2
    z = softmax(y2)
    loss = -np.log(z[l])

    # Backward pass
    dz = z
    dz[l] -= 1 # Gradient of loss w.r.t z

    # Gradient of loss w.r.t W2 and b2
    dW2 = dz @ y1.T
    db2 = dz  # directly use dz as db2 to save memory

    # Gradient of loss w.r.t y1
    dy1 = W2.T @ dz

    # Gradient of loss w.r.t t1 (ReLU backprop)
    dt1 = dy1
    dt1[t1 <= 0] = 0  # applying ReLU's gradient

    # Gradient of loss w.r.t W1 and b1
    dW1 = dt1 @ x.T
    db1 = dt1  # directly use dt1 as db1 to save memory

    return loss,((dW1, db1), (dW2, db2))
```

Summary: Numerics of AI

- Questions of dynamic range arise even in F32.

```python
def ffn(W, x):
    ((W1,b1),(W2,b2)) = W
    t1 = W1 @ x + b1
    y1 = relu(t1)
    y2 = W2 @ y1 + b2
    return softmax(y2)
```

```python
def loss(W, x, l):
    z = ffn(W,x)
    return -jnp.log(z[l])
```

- Pesky exponentials!

```python
# Problem dimensions:
#  ni: input image size
#  nh: hidden layer size
#  no: number of output classes
ni,nh,no = 28*28, 512, 10
# Make some random weights
W1,b1 = np.random.randn(nh,ni), np.zeros((nh,1))
W2,b2 = np.random.randn(no,nh), np.zeros((no,1))
W = (W1,b1),(W2,b2)
# Make a random input and label
x = np.random.rand(ni, 1)
l = 2
# What's the loss?
loss(W, x, l)
```

✓ 0.0s

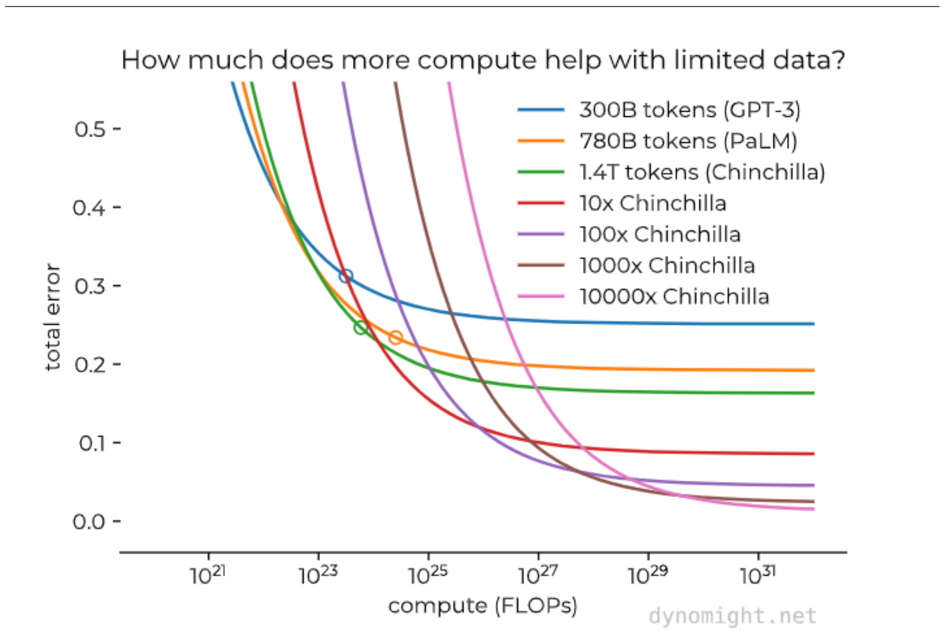
```
Array([inf], dtype=float32)
```

Summary: Numerics of AI

- Primary operations:
  - Large matrix-matrix multiplies
  - Comparisons (max, relu)
  - Transcendentals (exp, tanh)
- All need floating point for gradients

- Concerns:
  - FLOPs
  - Memory usage
  - Memory bandwidth

- And, for AI research:
  - “Hackability”, ease of debugging.



Summary: Numerics of AI

- Primary operations:
  - Large matrix-matrix multiplies
  - Comparisons (max, relu)
  - Transcendentals (exp, tanh)
- All need floating point for gradients

- Concerns:
  - FLOPs
  - Memory usage
  - Memory bandwidth

- And, for AI research:
  - "Hackability", ease of debugging.

FP4 Tensor Core Dense/Sparse	20 / 40 petaFLOPS
FP8/FP6 Tensor Core Dense/Sparse	10 / 20 petaFLOPS
INT8 Tensor Core Dense/Sparse	10 / 20 petaOPS
FP16/BF16 Tensor Core Dense/Sparse	5 / 10 petaFLOPS
TF32 Tensor Core Dense/Sparse	2.5 / 5 petaFLOPS
FP32	180 teraFLOPS
FP64 Tensor Core Dense	90 teraFLOPS
FP64	90 teraFLOPS
HBM Memory Architecture	HBM3e 8x2-sites
HBM Memory Size	Up to 384 GB
HBM Memory Bandwidth	Up to 16 TB/s

Formats in use today ("OCP")

Format		OCP Notes
E4M3		Signed, 2 NaNs, 0 Infs
E5M2		Signed, 6 NaNs, 2 Infs
E3M2		Signed, 0 NaNs, 0 Infs
E2M1		Signed, 0 NaNs, 0 Infs
E8M0		Unsigned, 1 NaN

Comparison table: Existing FP8 implementations

Existing implementations
nVidia, Intel, ARM: “E5M2, E4M3”
AMD, Qualcomm, Graphcore: “e5m2_fnuz”, “e4m3_fn”
Tesla: CFloat

	WG Proposals		'idia, Intel, AR		ɔre, Qualcom		Tesla	
Decision	**WG E4**	**WG E5**	**NIA E4**	**NIA E5**	**GQA E**	**GQA E**	**Tesla E**	**Tesla E5**
The set of nonreal values (e.g. {Inf, -Inf, -0, NaN}) shall be the same for each defined fo	yes		no		yes		yes	
FP8 value sets shall include exactly one NaN	yes	yes	no	no	yes	yes	no	no
FP8 value sets shall include subnormal values	yes	yes	yes	yes	yes	yes	yes	yes
FP8 formats shall include encodings for positive and negative infinity	yes	yes	no	yes	no	no	no	no
FP8 value sets shall not include negative zero			no	no	yes	yes	no	no
*The default bias for exponent width w shall be 2**w-1*			no	no	yes	yes	n/a	n/a

IEEE P3109 “Project Authorization Request”

Scope of proposed standard: This standard defines a binary arithmetic and data format for machine learning-optimized domains. It also specifies the default handling of exceptions that occur in this arithmetic. This standard provides a consistent and flexible arithmetic framework optimized for Machine Learning Systems (MLS) in hardware and/or software implementations to minimize the work required to make MLS interoperable with each other, as well as other dependent systems. This standard is aligned with IEEE Std 754-2019 for Floating-Point Arithmetic.

Need for this Work: Machine Learning Systems have different arithmetic requirements from most other domains. Precisions tend to be lower, and accuracy is measured in dimensions other than just numerical (e.g. inference accuracy). Furthermore, machine learning systems are often integrated into mission-critical and safety-critical systems. With no standards specifically addressing these needs, Machine Learning Systems are built with inconsistent expectations and assumptions that hinder the compatibility and reuse of machine learning hardware, software, and training data.

Stakeholders for the Standard: System developers, vendors, and users of machine learning applications across many industries and interests including but not limited to computation, storage, medical, telecommunications, e-commerce, fleet management, automotive, robotics, and security.

IEEE P3109 “Project Authorization Request”

Scope of proposed standard: This standard defines a binary arithmetic and data format for machine learning-optimized domains. It also specifies the default handling of exceptions that occur in this arithmetic. This standard provides a consistent and flexible arithmetic framework optimized for Machine Learning Systems (MLS) in hardware and/or software implementations to minimize the work required to make MLS interoperable with each other, as well as other dependent systems. This standard is aligned with IEEE Std 754-2019 for Floating-Point Arithmetic.

Need for this Work: Machine Learning Systems have different arithmetic requirements from most other domains. Precisions tend to be lower, and accuracy is measured in dimensions other than just numerical (e.g. inference accuracy). Furthermore, machine learning systems are often integrated into mission-critical and safety-critical systems. With no standards specifically addressing these needs, Machine Learning Systems are built with inconsistent expectations and assumptions that hinder the compatibility and reuse of machine learning hardware, software, and training data.

Stakeholders for the Standard: System developers, vendors, and users of machine learning applications across many industries and interests including but not limited to computation, storage, medical, telecommunications, e-commerce, fleet management, automotive, robotics, and security.

How to achieve the goals of the PAR?

- Possible solution: simply codify majority existing practice (e.g. OCP) in one document
  - Does not satisfy “consistent”
    - E5m2 has 6 nans, 2 infs
    - E4m3 has 2 nans, no infs
    - E3m2 has no nans, no infs
  - Hence not “flexible” - with no consistency between formats, how do we define new variants?
- Better solution:
  - Define a consistent and flexible arithmetic framework aligned with 754 that meets the requirements of ML systems

So… Fast small floats..

- Need for fast, small floats
- Different design space than IEEE-754
- Here's a hypothetical 6-bit float following 754:
  - 3 Subnormals
  - 64 code points
  - 6 NaNs (9.4%)
  - One negative zero (1.6%)
  - +/- Infinity (3.1%)

0x00 0_000_00 = 0.0	0x20 1_000_00 = -0.0
0x01 0_000_01 = +0b0.01*2^-2 = 0.0625	0x21 1_000_01 = -0b0.01*2^-2 = -0.0625
0x02 0_000_10 = +0b0.10*2^-2 = 0.125	0x22 1_000_10 = -0b0.10*2^-2 = -0.125
0x03 0_000_11 = +0b0.11*2^-2 = 0.1875	0x23 1_000_11 = -0b0.11*2^-2 = -0.1875
0x04 0_001_00 = +0b1.00*2^-2 = 0.25	0x24 1_001_00 = -0b1.00*2^-2 = -0.25
0x05 0_001_01 = +0b1.01*2^-2 = 0.3125	0x25 1_001_01 = -0b1.01*2^-2 = -0.3125
0x06 0_001_10 = +0b1.10*2^-2 = 0.375	0x26 1_001_10 = -0b1.10*2^-2 = -0.375
0x07 0_001_11 = +0b1.11*2^-2 = 0.4375	0x27 1_001_11 = -0b1.11*2^-2 = -0.4375
0x08 0_010_00 = +0b1.00*2^-1 = 0.5	0x28 1_010_00 = -0b1.00*2^-1 = -0.5
0x09 0_010_01 = +0b1.01*2^-1 = 0.625	0x29 1_010_01 = -0b1.01*2^-1 = -0.625
0x0a 0_010_10 = +0b1.10*2^-1 = 0.75	0x2a 1_010_10 = -0b1.10*2^-1 = -0.75
0x0b 0_010_11 = +0b1.11*2^-1 = 0.875	0x2b 1_010_11 = -0b1.11*2^-1 = -0.875
0x0c 0_011_00 = +0b1.00*2^0 = 1.0	0x2c 1_011_00 = -0b1.00*2^0 = -1.0
0x0d 0_011_01 = +0b1.01*2^0 = 1.25	0x2d 1_011_01 = -0b1.01*2^0 = -1.25
0x0e 0_011_10 = +0b1.10*2^0 = 1.5	0x2e 1_011_10 = -0b1.10*2^0 = -1.5
0x0f 0_011_11 = +0b1.11*2^0 = 1.75	0x2f 1_011_11 = -0b1.11*2^0 = -1.75
0x10 0_100_00 = +0b1.00*2^1 = 2.0	0x30 1_100_00 = -0b1.00*2^1 = -2.0
0x11 0_100_01 = +0b1.01*2^1 = 2.5	0x31 1_100_01 = -0b1.01*2^1 = -2.5
0x12 0_100_10 = +0b1.10*2^1 = 3.0	0x32 1_100_10 = -0b1.10*2^1 = -3.0
0x13 0_100_11 = +0b1.11*2^1 = 3.5	0x33 1_100_11 = -0b1.11*2^1 = -3.5
0x14 0_101_00 = +0b1.00*2^2 = 4.0	0x34 1_101_00 = -0b1.00*2^2 = -4.0
0x15 0_101_01 = +0b1.01*2^2 = 5.0	0x35 1_101_01 = -0b1.01*2^2 = -5.0
0x16 0_101_10 = +0b1.10*2^2 = 6.0	0x36 1_101_10 = -0b1.10*2^2 = -6.0
0x17 0_101_11 = +0b1.11*2^2 = 7.0	0x37 1_101_11 = -0b1.11*2^2 = -7.0
0x18 0_110_00 = +0b1.00*2^3 = 8.0	0x38 1_110_00 = -0b1.00*2^3 = -8.0
0x19 0_110_01 = +0b1.01*2^3 = 10.0	0x39 1_110_01 = -0b1.01*2^3 = -10.0
0x1a 0_110_10 = +0b1.10*2^3 = 12.0	0x3a 1_110_10 = -0b1.10*2^3 = -12.0
0x1b 0_110_11 = +0b1.11*2^3 = 14.0	0x3b 1_110_11 = -0b1.11*2^3 = -14.0
0x1c 0_111_00 = inf	0x3c 1_111_00 = -inf
0x1d 0_111_01 = nan	0x3d 1_111_01 = nan
0x1e 0_111_10 = nan	0x3e 1_111_10 = nan
0x1f 0_111_11 = nan	0x3f 1_111_11 = nan

So… Fast small floats..

- Need for fast, small floats
- Different design space than IEEE-754
- Here's a P3109 6-bit float:
  - 3 Subnormals
  - *No negative zero*
  - *Just one NaN*
  - +/- Infinity
- And more questions:
  - What width, precision?
  - What exponent bias?
  - What operations?
  - To what accuracy?

0x00 0_000_00 = 0.0	0x20 1_000_00 = nan
0x01 0_000_01 = +0b0.01*2^-3 = 0.03125	0x21 1_000_01 = -0b0.01*2^-3 = -0.03125
0x02 0_000_10 = +0b0.10*2^-3 = 0.0625	0x22 1_000_10 = -0b0.10*2^-3 = -0.0625
0x03 0_000_11 = +0b0.11*2^-3 = 0.09375	0x23 1_000_11 = -0b0.11*2^-3 = -0.09375
0x04 0_001_00 = +0b1.00*2^-3 = 0.125	0x24 1_001_00 = -0b1.00*2^-3 = -0.125
0x05 0_001_01 = +0b1.01*2^-3 = 0.15625	0x25 1_001_01 = -0b1.01*2^-3 = -0.15625
0x06 0_001_10 = +0b1.10*2^-3 = 0.1875	0x26 1_001_10 = -0b1.10*2^-3 = -0.1875
0x07 0_001_11 = +0b1.11*2^-3 = 0.21875	0x27 1_001_11 = -0b1.11*2^-3 = -0.21875
0x08 0_010_00 = +0b1.00*2^-2 = 0.25	0x28 1_010_00 = -0b1.00*2^-2 = -0.25
0x09 0_010_01 = +0b1.01*2^-2 = 0.3125	0x29 1_010_01 = -0b1.01*2^-2 = -0.3125
0x0a 0_010_10 = +0b1.10*2^-2 = 0.375	0x2a 1_010_10 = -0b1.10*2^-2 = -0.375
0x0b 0_010_11 = +0b1.11*2^-2 = 0.4375	0x2b 1_010_11 = -0b1.11*2^-2 = -0.4375
0x0c 0_011_00 = +0b1.00*2^-1 = 0.5	0x2c 1_011_00 = -0b1.00*2^-1 = -0.5
0x0d 0_011_01 = +0b1.01*2^-1 = 0.625	0x2d 1_011_01 = -0b1.01*2^-1 = -0.625
0x0e 0_011_10 = +0b1.10*2^-1 = 0.75	0x2e 1_011_10 = -0b1.10*2^-1 = -0.75
0x0f 0_011_11 = +0b1.11*2^-1 = 0.875	0x2f 1_011_11 = -0b1.11*2^-1 = -0.875
0x10 0_100_00 = +0b1.00*2^0 = 1.0	0x30 1_100_00 = -0b1.00*2^0 = -1.0
0x11 0_100_01 = +0b1.01*2^0 = 1.25	0x31 1_100_01 = -0b1.01*2^0 = -1.25
0x12 0_100_10 = +0b1.10*2^0 = 1.5	0x32 1_100_10 = -0b1.10*2^0 = -1.5
0x13 0_100_11 = +0b1.11*2^0 = 1.75	0x33 1_100_11 = -0b1.11*2^0 = -1.75
0x14 0_101_00 = +0b1.00*2^1 = 2.0	0x34 1_101_00 = -0b1.00*2^1 = -2.0
0x15 0_101_01 = +0b1.01*2^1 = 2.5	0x35 1_101_01 = -0b1.01*2^1 = -2.5
0x16 0_101_10 = +0b1.10*2^1 = 3.0	0x36 1_101_10 = -0b1.10*2^1 = -3.0
0x17 0_101_11 = +0b1.11*2^1 = 3.5	0x37 1_101_11 = -0b1.11*2^1 = -3.5
0x18 0_110_00 = +0b1.00*2^2 = 4.0	0x38 1_110_00 = -0b1.00*2^2 = -4.0
0x19 0_110_01 = +0b1.01*2^2 = 5.0	0x39 1_110_01 = -0b1.01*2^2 = -5.0
0x1a 0_110_10 = +0b1.10*2^2 = 6.0	0x3a 1_110_10 = -0b1.10*2^2 = -6.0
0x1b 0_110_11 = +0b1.11*2^2 = 7.0	0x3b 1_110_11 = -0b1.11*2^2 = -7.0
0x1c 0_111_00 = +0b1.00*2^3 = 8.0	0x3c 1_111_00 = -0b1.00*2^3 = -8.0
0x1d 0_111_01 = +0b1.01*2^3 = 10.0	0x3d 1_111_01 = -0b1.01*2^3 = -10.0
0x1e 0_111_10 = +0b1.10*2^3 = 12.0	0x3e 1_111_10 = -0b1.10*2^3 = -12.0
0x1f 0_111_11 = inf	0x3f 1_111_11 = -inf

IEEE P3109 formats

binary$\langle K \rangle$p$\langle P \rangle \sigma\delta$

Width K: $1 \leq K < 16$
Precision P: $1 \leq P < K$
Signedness σ: “Signed”, “Unsigned”
Domain δ: “Finite”, “Extended”

OCP	P3109 analogue
E4M3	Binary8p4se
E5M2	Binary8p3se
E3M2	Binary6p3sf
E2M1	Binary4p2sf
E8M0	binary8p1uf

Subsetting

- Defining many formats covers many future use cases
  - But of course vendors cannot support all formats
  - But it is still useful to be able to describe precisely what one's system does support
  - Subsetting already exists: vendors today often don't support all of F16,F32,F64 (other than in software)

First *define*. *Then* restrict

- Total number of formats is rather large:
  - Signed: $K = 2 \ldots 15, 1 \leq P < K$, so 105 formats
  - Unsigned: $K = 2 \ldots 15, 1 \leq P \leq K$ so 119 formats
  - Total 224, which is MAX_FLOAT for binary8p4se….
  - x2 for Finite, not Extended, domain.

Operations

Scaled addition

Compute $X \times 2^{s_x} + Y \times 2^{s_y}$, and return a P3109 value. Scaling is applied in the extended reals, before projection to the target format.

Signature

$\mathrm{AddScaled}_{f_x,f_y,f_z,\pi}(x, s_x, y, s_y) \to z$

Parameters

f_x : format of x
f_y : format of y
f_z : format of z
π : projection specification

Operands

x : P3109 value, format f_x
s_x : integer log-scale factor for x
y : P3109 value, format f_y
s_y : integer log-scale factor for y

Output

z : P3109 value, format f_z

Behavior

AddScaled(NaN, $*$, $*$, $*$) $\to$ NaN
AddScaled($*$, $*$, NaN, $*$) $\to$ NaN
AddScaled(-Inf, $*$, Inf, $*$) $\to$ NaN
AddScaled(Inf, $*$, -Inf, $*$) $\to$ NaN
AddScaled(x, s_x, y, s_y) $\to$ $\mathrm{Project}_{f_z,\pi}(Z)$, where

$Z = X \times 2^{s_x} + Y \times 2^{s_y}$
$X = \mathrm{Decode}_{f_x}(x)$
$Y = \mathrm{Decode}_{f_y}(y)$

Figure 1. Textual specification of scaled addition

Scaled addition

Compute $X \times 2^{s_x} + Y \times 2^{s_y}$, and return a P3109 value. Scaling is applied in the extended reals, before projection to the target format.

Signature

$\mathrm{AddScaled}_{f_x,f_y,f_z,\pi}(x, s_x, y, s_y) \rightarrow z$

Parameters

f_x : format of x
f_y : format of y
f_z : format of z
π : projection specification

Operands

x : P3109 value, format f_x
s_x : integer log-scale factor for x
y : P3109 value, format f_y
s_y : integer log-scale factor for y

Output

z : P3109 value, format f_z

Behavior

AddScaled(NaN, $*$, $*$, $*$) $\rightarrow$ NaN
AddScaled($*$, $*$, NaN, $*$) $\rightarrow$ NaN
AddScaled(-Inf, $*$, Inf, $*$) $\rightarrow$ NaN
AddScaled(Inf, $*$, -Inf, $*$) $\rightarrow$ NaN
AddScaled(x, s_x, y, s_y) $\rightarrow$ $\mathrm{Project}_{f_z,\pi}(Z)$, where

$$Z = X \times 2^{s_x} + Y \times 2^{s_y}$$
$$X = \mathrm{Decode}_{f_x}(x)$$
$$Y = \mathrm{Decode}_{f_y}(y)$$

Figure 1. Textual specification of scaled addition

Scaled addition

Compute $X \times 2$
Scaling is applie
to the target for

Signature

AddScaled$_{f_x, f_y, f}$

Parameters

f_x : format of x
f_y : format of y
f_z : format of z
π : projection sp

Operands

x : P3109 value
s_x : integer log-
y : P3109 value
s_y : integer log-

Output

z : P3109 value

Behavior

AddScaled(NaN
AddScaled($*$, $*$,
AddScaled(-Inf,
AddScaled(Inf,
AddScaled(x, s,

$Z = X \times 2^{s_x}$
X = Decode
Y = Decode$_{f_y}(y)$

Figure 1. Textual specification of scaled addition

4.3 Projection specifications

Operations on P3109 values are defined via conversion to extended real values, on which the mathematical operation is performed, before conversion back to the appropriate P3109 range. In general, operation results will not be exact P3109 values, and hence will be *projected* into the P3109 range via rounding and overflow handling. A *projection specification* is a pair (rounding mode, saturation mode). For a given projection specification π, these are written Rnd_π, Sat_π.

The defined rounding modes are as follows. The precise specifications of these modes are in the function RoundToPrecision (§4.6.2):

NearestTiesToEven	Round to nearest, ties to even
NearestTiesToAway	Round to nearest, ties away from zero
TowardPositive	Round toward positive
TowardNegative	Round toward negative
TowardZero	Round toward zero

Values are first rounded to the target precision, with exponent unbounded above. Those which are then outside the maximum value in the target format are then treated according to the saturation mode.

The defined saturation modes are as follows. The precise specifications of these modes are in the function Saturate (§4.6.4):

SatMax	All return values are clamped to the representable finite range.
SatFinite	Finite out-of-range values are clamped to the representable finite range, infinities are preserved.
OvfInf	Out-of-range values are replaced with extreme value, positive or negative infinity as indicated by the rounding mode.

Scaled addition

Compute $X \times 2^{s_x} + Y \times 2^{s_y}$, and return a P
Scaling is applied in the extended reals, befo
to the target format.

Signature

$\text{AddScaled}_{f_x,f_y,f_z,\pi}(x, s_x, y, s_y) \to z$

Parameters

f_x : format of x
f_y : format of y
f_z : format of z
π : projection specification

Operands

x : P3109 value, format f_x
s_x : integer log-scale factor for x
y : P3109 value, format f_y
s_y : integer log-scale factor for y

Output

z : P3109 value, format f_z

Behavior

AddScaled(NaN, $*$, $*$, $*$) $\to$ NaN
AddScaled($*$, $*$, NaN, $*$) $\to$ NaN
AddScaled(-Inf, $*$, Inf, $*$) $\to$ NaN
AddScaled(Inf, $*$, -Inf, $*$) $\to$ NaN
AddScaled(x, s_x, y, s_y) $\to$ $\text{Project}_{f_z,\pi}(Z)$, w

$Z = X \times 2^{s_x} + Y \times 2^{s_y}$
$X = \text{Decode}_{f_x}(x)$
$Y = \text{Decode}_{f_y}(y)$

Figure 1. Textual specification of scaled addition

4.6.2 Project

Project extended real value to P3109 format f, applying specified rounding and saturation.

Signature

$\text{Project}_{f,\pi}(X) \to x$

Parameters

f : target format, precision P_f, exponent bias b_f, maximum finite value M_f
π : projection specification: rounding mode Rnd_π, saturation Sat_π

Operands

X : extended real value

Output

x : P3109 value, format f

Behavior

$\text{Project}_{f,\pi}(X) \to x$

where

$R = \text{RoundToPrecision}_{\mathsf{P}_f, b_f, \text{Rnd}_\pi}(X)$ —Round to precision P_f with expone
$S = \text{Saturate}_{M_f}(\text{Sat}_\pi, \text{Rnd}_\pi, R)$
$x = \text{Encode}_f(S)$

Behavior

$$\mathsf{RoundToPrecision}(X \in \{0, -\infty, \infty\}) \to X$$
$$\mathsf{RoundToPrecision}(X) \to Z$$

where

$$E = \max(\lfloor \log_2(|X|) \rfloor, 1-b) - \mathsf{P} + 1 \qquad \text{—Subnormals handled by } \max(\cdot, 1-b)$$

$$S = |X| \times 2^{-E} \qquad \text{—Real-valued significand, to be rounded to integer}$$

$$\Delta = S - \lfloor S \rfloor$$

$$\mathsf{CodeIsEven} = \begin{cases} \mathsf{IsEven}(\lfloor S \rfloor) & \textbf{if } \mathsf{P} > 1 \\ (\lfloor S \rfloor = 0) \vee \mathsf{IsEven}(E + b) & \text{Otherwise} \end{cases}$$

$$I = \lfloor S \rfloor + \mathbb{1}[\mathsf{RoundAway}(\mathsf{Rnd})]$$

$$Z = \operatorname{sign}(X) \times I \times 2^{E}$$

and

$$\mathsf{RoundAway}(\mathsf{NearestTiesToEven}) = \Delta > 0.5 \vee (\Delta = 0.5 \wedge \neg\mathsf{CodeIsEven})$$

$$\mathsf{RoundAway}(\mathsf{NearestTiesToAway}) = \Delta \geq 0.5$$

$$\mathsf{RoundAway}(\mathsf{TowardPositive}) = \Delta > 0 \wedge X > 0$$

$$\mathsf{RoundAway}(\mathsf{TowardNegative}) = \Delta > 0 \wedge X < 0$$

$$\mathsf{RoundAway}(\mathsf{TowardZero}) = \mathsf{False}$$

Behavior

$$\mathsf{Saturate}(*, *, X \in [-M, M]) \to X$$

$$\mathsf{Saturate}(\mathsf{SatMax}, *, X \notin [-M, M]) \to \mathrm{sign}(X) \times M$$

$$\mathsf{Saturate}(\mathsf{SatFinite}, *, X \in \{\pm\infty\}) \to X$$
$$\mathsf{Saturate}(\mathsf{SatFinite}, *, |X| \in [M, \infty)) \to \mathrm{sign}(X) \times M$$

$$\mathsf{Saturate}(\mathsf{OvfInf}, *, X \in \{\pm\infty\}) \to X$$
$$\mathsf{Saturate}(\mathsf{OvfInf}, \mathsf{TowardZero}, |X| \in [M, \infty)) \to \mathrm{sign}(X) \times M$$
$$\mathsf{Saturate}(\mathsf{OvfInf}, \mathsf{TowardPositive}, X \in (-\infty, -M)) \to -M$$
$$\mathsf{Saturate}(\mathsf{OvfInf}, \mathsf{TowardNegative}, X \in (M, \infty)) \to M$$
$$\mathsf{Saturate}(\mathsf{OvfInf}, *, X) \to \mathrm{sign}(X) \times \infty$$

Scaled addition

Compute $X \times 2^{s_x} + Y \times 2^{s_y}$, and return a P3109 value. Scaling is applied in the extended reals, before projection to the target format.

Signature

$\mathrm{AddScaled}_{f_x,f_y,f_z,\pi}(x, s_x, y, s_y) \to z$

Parameters

f_x : format of x
f_y : format of y
f_z : format of z
π : projection specification

Operands

x : P3109 value, format f_x
s_x : integer log-scale factor for x
y : P3109 value, format f_y
s_y : integer log-scale factor for y

Output

z : P3109 value, format f_z

Behavior

AddScaled(NaN, $*$, $*$, $*$) $\to$ NaN
AddScaled($*$, $*$, NaN, $*$) $\to$ NaN
AddScaled(-Inf, $*$, Inf, $*$) $\to$ NaN
AddScaled(Inf, $*$, -Inf, $*$) $\to$ NaN
AddScaled(x, s_x, y, s_y) $\to$ $\mathrm{Project}_{f_z,\pi}(Z)$, where

$$Z = X \times 2^{s_x} + Y \times 2^{s_y}$$
$$X = \mathrm{Decode}_{f_x}(x)$$
$$Y = \mathrm{Decode}_{f_y}(y)$$

Figure 1. Textual specification of scaled addition

Scaled addition

Compute $X \times 2^{s_x} + Y \times 2^{s_y}$, and return a P3109 value.

Scaling is app

to the target f

Signature

AddScaled$_{f_x,}$

Parameters

f_x : format o

f_y : format o

f_z : format o

π : projectior

Operands

x : P3109 va

s_x : integer l

y : P3109 va

s_y : integer l

Output

z : P3109 val

Behavior

AddScaled(N

AddScaled(*,

AddScaled(-I

AddScaled(In

AddScaled(x,

$Z = X \times 2$

$X = \text{Decode}_{f_x}(x)$

$Y = \text{Decode}_{f_y}(y)$

Figure 1. Textual specification of scaled addition

Formal Verification of the IEEE P3109 Standard for Binary Floating-point Formats for Machine Learning

Christoph M. Wintersteiger
Imandra Inc
christoph@imandra.ai

***Abstract*—We present a formalization of an upcoming standard for floating-point formats for machine learning by the IEEE P3109 working group. This includes a definition of a number of small (< 16 bit) formats and a specification of arithmetic functions that operate on such numbers, as well as format conversion function, including conversions to and from IEEE 754 formats. We report on our experience with the use of an automated theorem prover for verification and analysis of our formalization of the specification, and on the utility of the formalization in future implementations of P3109-compliant hardware and software.**

1. Introduction

Progress in machine learning, and artificial intelligence that is close to IEEE 754, but that is parametrizable by bit-widths for exponents and significands.

2. Background

P3109 defines formats for 3 to 15-bit floating-point numbers, each with a significand precision of one up to $n-1$ bits (including a hidden significand bit). P3109 numbers include positive and negative infinities as well as a single NaN without payload and a single zero. It further specifies the usual arithmetic operations, square roots, and natural and binary logarithms and exponentials. Additionally it includes versions of addition and multiplication with (log-scale) scaling factors and a fused-multiply-add operation that takes a (scaled) IEEE 754 value as a summand alongside P3109 values and produces an IEEE 754 result. For more details

Scaled addition

Compute $X \times 2^{s_x} + Y \times 2^{s_y}$, and return a P3109 value. Scaling is applied in the extended reals, before projection to the target format.

Signature

$\mathrm{AddScaled}_{f_x, f_y, f_z, \pi}(x, s_x, y, s_y) \to z$

Parameters

f_x : format of x
f_y : format of y
f_z : format of z
π : projection specification

Operands

x : P3109 value, format f_x
s_x : integer log-scale factor for x
y : P3109 value, format f_y
s_y : integer log-scale factor for y

Output

z : P3109 value, format f_z

Behavior

$\mathrm{AddScaled}(\mathrm{NaN}, *, *, *) \to \mathrm{NaN}$
$\mathrm{AddScaled}(*, *, \mathrm{NaN}, *) \to \mathrm{NaN}$
$\mathrm{AddScaled}(\text{-Inf}, *, \mathrm{Inf}, *) \to \mathrm{NaN}$
$\mathrm{AddScaled}(\mathrm{Inf}, *, \text{-Inf}, *) \to \mathrm{NaN}$
$\mathrm{AddScaled}(x, s_x, y, s_y) \to \mathrm{Project}_{f_z,\pi}(Z)$, where

$$Z = X \times 2^{s_x} + Y \times 2^{s_y}$$
$$X = \mathrm{Decode}_{f_x}(x)$$
$$Y = \mathrm{Decode}_{f_y}(y)$$

Figure 1. Textual specification of scaled addition

```
let internal_add_scaled
  (f_x : Format.t) (f_y : Format.t)
  (f_z : Format.t) (pi : Projection.t)
  (x : Float8.t) (s_x : int)
  (y : Float8.t) (s_y : int)
  : (t, string) Result.t =
  let open NaNOrExReal in
  match x, y with
  | _, y when y = nan -> Ok nan
  | x, _ when x = nan -> Ok nan
  | x, y when x = ninf && y = pinf -> Ok nan
  | x, y when x = pinf && y = ninf -> Ok nan
  | x, y ->
    let x = decode f_x x in
    let y = decode f_y y in
    (match x, y with
    | Ok NaN, _ | _, Ok NaN -> Ok nan
    | Ok (XR x) , Ok (XR y) ->
      let open ExReal.ResultInfix in
      (match ((Ok x) *. (2 ^. s_x)) +.
             ((Ok y) *. (2 ^. s_y)) with
      | Ok z -> project f_z pi z
      | Error e -> Error e)
    | _, Error e -> Error e
    | Error e, _ -> Error e)
```

```
theorem internal_add_scaled_ok
  (f_x : Format.t) (f_y : Format.t)
  (f_z : Format.t) (pi : Projection.t)
  (x : Float8.t) (s_x : int)
  (y : Float8.t) (s_y : int) =
  Result.is_ok (Float8.internal_add_scaled
      f_x f_y f_z pi x s_x y s_y)
```

Summary

$$\text{binary}\langle K \rangle \text{p} \langle P \rangle \sigma\delta$$

- A family of floating point formats, in a variety of bitwidths, precisions, signednesses, and with/without infinities.
- With precisely defined operation semantics under 5 rounding modes, 3 saturation modes
- Still to do: block formats, stochastic rounding, round to odd

Deltas from R2

- Operations and formal verification
- Signedness
- Domain

And switch from emax (a la 754) to bias, which yields better consistency with low P, and across signedness and domain.

github.com/graphcore-research/gfloat

README MIT license

gfloat: Generic floating-point types in Python

An implementation of generic floating point encode/decode logic, handling various current and proposed floating point types:

- IEEE 754: Binary16, Binary32
- OCP Float8: E5M2, E4M3
- IEEE WG P3109: P3109_{K}p{P} for K > 2, and 1 <= P < K.
- OCP MX Formats: E2M1, M2M3, E3M2, E8M0, INT8, and the MX block formats.

The library favours readability and extensibility over speed (although the *_ndarray functions are reasonably fast for large arrays, see the benchmarking notebook). For other implementations of these datatypes more focused on speed see, for example, ml_dtypes, bitstring, MX PyTorch Emulation Library.

See https://gfloat.readthedocs.io for documentation, or dive into the notebooks to explore the formats.

What width, precision?

What precision?

Reminder: precision is width of significand (including “hidden” bit)

Higher P => lower dynamic range

- Research found “4 is good for weights and activations, 3 for gradients”
- Other research has considered other values
- P=7 is a linear format
- P=1 (zero mantissa bits) is a pure-exponential format

Solution: define formats $Binary\{K\}p\{P\}$ for $2 \leq K < 16$ and $1 \leq P < K$

Implementations are not expected to support all, but to declare

“This system supports Binary8P3 and Binary8P4”

More on this later – which operations are supported for which format?

What exponent bias?

- In IEEE-754, the definitions are in terms of "emax"
- Consistently defined as $2^{k-p-1} - 1$

Table 3.5—Binary interchange format parameters

Parameter	binary16	binary32	binary64	binary128	binary{k} ($k \geq 128$)
k, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$k - \text{round}(4 \times \log_2(k)) + 13$
emax, maximum exponent e	15	127	1023	16383	$2^{(k-p-1)} - 1$
Encoding parameters					
bias, $E - e$	15	127	1023	16383	*emax*
sign bit	1	1	1	1	1
w, exponent field width in bits	5	8	11	15	$\text{round}(4 \times \log_2(k)) - 13$
t, trailing significand field width in bits	10	23	52	112	$k - w - 1$
k, storage width in bits	16	32	64	128	$1 + w + t$

What exponent bias?

- In IEEE-754, the definitions are in terms of "emax"
- Consistently defined as $2^{k-p-1} - 1$
- P3109 does the same.

Parameter	**binary8p{p}**	**binary8p5**	**binary8p4**	**binary8p3**	**binary16**	**binary32**	**binary64**
Storage width in bits k	**8**	**8**	**8**	**8**	**16**	**32**	**64**
Precision in bits p	**p**	**5**	**4**	**3**	**11**	**24**	**53**
Max exponent emax	$2^{k-p-1} - 1$	**3**	**7**	**15**	**15**	**127**	**1023**
Sign bit	1	1	1	1	1	1	1
Exponent field width w	$8 - p$	3	4	5	5	8	11
Exponent bias, bias	$\text{emax} + (p > 1)$	4	8	16	15	127	1023
Trailing significand field width in bits t	$p - 1$	4	3	2	10	23	52

But what's happening with the bias?

All-bits-one-exponents (ABOE)

OCP E4M3	OCP E5M2	WG P3	WG P2	WG P1
0_1110_001 = +0b1.001*2^7	0x71 0_11100_01 = +0b1.01*2^13	0x71 0_11100_01 = +0b1.01*2^12	0x71 0_111000_1 = +0b1.1*2^24	0x71 0_1110001_ = +0b1.0*2^50
0_1110_010 = +0b1.010*2^7	0x72 0_11100_10 = +0b1.10*2^13	0x72 0_11100_10 = +0b1.10*2^12	0x72 0_111001_0 = +0b1.0*2^25	0x72 0_1110010_ = +0b1.0*2^51
0_1110_011 = +0b1.011*2^7	0x73 0_11100_11 = +0b1.11*2^13	0x73 0_11100_11 = +0b1.11*2^12	0x73 0_111001_1 = +0b1.1*2^25	0x73 0_1110011_ = +0b1.0*2^52
0_1110_100 = +0b1.100*2^7	0x74 0_11101_00 = +0b1.00*2^14	0x74 0_11101_00 = +0b1.00*2^13	0x74 0_111010_0 = +0b1.0*2^26	0x74 0_1110100_ = +0b1.0*2^53
0_1110_101 = +0b1.101*2^7	0x75 0_11101_01 = +0b1.01*2^14	0x75 0_11101_01 = +0b1.01*2^13	0x75 0_111010_1 = +0b1.1*2^26	0x75 0_1110101_ = +0b1.0*2^54
0_1110_110 = +0b1.110*2^7	0x76 0_11101_10 = +0b1.10*2^14	0x76 0_11101_10 = +0b1.10*2^13	0x76 0_111011_0 = +0b1.0*2^27	0x76 0_1110110_ = +0b1.0*2^55
0_1110_111 = +0b1.111*2^7	0x77 0_11101_11 = +0b1.11*2^14	0x77 0_11101_11 = +0b1.11*2^13	0x77 0_111011_1 = +0b1.1*2^27	0x77 0_1110111_ = +0b1.0*2^56
0_1111_000 = +0b1.000*2^8	0x78 0_11110_00 = +0b1.00*2^15	0x78 0_11110_00 = +0b1.00*2^14	0x78 0_111100_0 = +0b1.0*2^28	0x78 0_1111000_ = +0b1.0*2^57
0_1111_001 = +0b1.001*2^8	0x79 0_11110_01 = +0b1.01*2^15	0x79 0_11110_01 = +0b1.01*2^14	0x79 0_111100_1 = +0b1.1*2^28	0x79 0_1111001_ = +0b1.0*2^58
0_1111_010 = +0b1.010*2^8	0x7a 0_11110_10 = +0b1.10*2^15	0x7a 0_11110_10 = +0b1.10*2^14	0x7a 0_111101_0 = +0b1.0*2^29	0x7a 0_1111010_ = +0b1.0*2^59
0_1111_011 = +0b1.011*2^8	0x7b 0_11110_11 = +0b1.11*2^15	0x7b 0_11110_11 = +0b1.11*2^14	0x7b 0_111101_1 = +0b1.1*2^29	0x7b 0_1111011_ = +0b1.0*2^60
0_1111_100 = +0b1.100*2^8	0x7c 0_11111_00 = inf	0x7c 0_11111_00 = +0b1.00*2^15	0x7c 0_111110_0 = +0b1.0*2^30	0x7c 0_1111100_ = +0b1.0*2^61
0_1111_101 = +0b1.101*2^8	0x7d 0_11111_01 = nan	0x7d 0_11111_01 = +0b1.01*2^15	0x7d 0_111110_1 = +0b1.1*2^30	0x7d 0_1111101_ = +0b1.0*2^62
0_1111_110 = +0b1.110*2^8	0x7e 0_11111_10 = nan	0x7e 0_11111_10 = +0b1.10*2^15	0x7e 0_111111_0 = +0b1.0*2^31	0x7e 0_1111110_ = +0b1.0*2^63
0_1111_111 = nan	0x7f 0_11111_11 = nan	0x7f 0_11111_11 = inf	0x7f 0_111111_1 = inf	0x7f 0_1111111_ = inf
ABOE has finites	ABOE all special	ABOE has finites	ABOE has finites	ABOE all special

maxFinite highlighted in yellow

Conclusion: When ABOE all special, emax occurs at $2^w - 2$, otherwise at $2^w - 1$, so bias is offset by 1

Choice of emax: Summary

- Near-symmetric distribution of values around 1
  - 63 encodings $0 < x < 1$, 63 encodings $1 \leq x < \infty$
- For P>2, all values are in FP16 dynamic range
- Most existing hardware implements fused “scale by power of two”, so choice of scale factor is less important
- E.g:

$$\mathrm{Multiply}(X, Y, L) \coloneqq X \times Y \times 2^{L}$$

On NaN

How many NaNs do we need

- Following IEEE-754 would introduce many NaNs
  - Initially used for hardware debugging, but modern chip design does not use them.
  - Various amusing “NaN-boxing” tricks have emerged over the years
    - In my experience, every time I commit code using NaN boxing, I commit code to remove it weeks, months, years later.

- Uses of NaN in Machine Learning:
  - “Missing Value” indicator
  - “Something went wrong” indicator
    - Crucial for debugging
    - Important on accelerated hardware where exceptions cannot be synchronous

- WG decision: We shall encode a single NaN

On Negative Zero

Negative zero: pros and cons

Pros

- Consistent implementation of branch cuts
  - But atan2 not common in ML code
- Hardware simplifications
  - But existing implementations show only a small advantage

Cons

- An additional code point
  - But just 1 in 256…
- Implies $1/(1/-\infty) = \infty$ (or $\frac{1}{0}$ =NaN)
- A natural location for a single NaN
  - But what about sorting using integers?
    - Still requires an O(N) pre-pass
    - And anyway essentially “undefined behaviour”

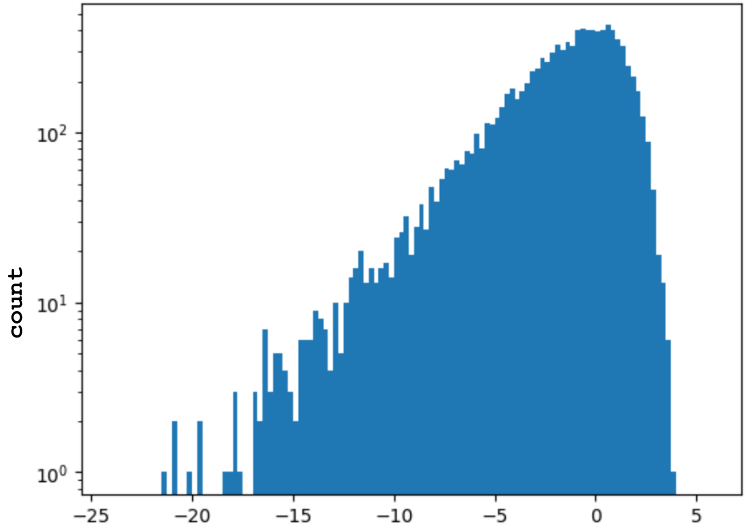
On Subnormals

On subnormals:

Whereas:
ML code is typically “well scaled” – values tend to be scaled so that

$$W + 10^{-3} * dW$$

is a value different from W, where W has some millions of entries distributed roughly according to…



If we consider the range of values of e.g. $p = 4$:

p3109_p4	p3109_p4 [without subnormals]
0x00 0_0000_000 = 0.0	0x00 0_0000_000 = 0.0
0x01 0_0000_001 = +0b0.001*2^-7 = ~0.001	0x01 0_0000_001 = +0b1.001*2^-8 = ~0.004
0x02 0_0000_010 = +0b0.010*2^-7 = ~0.002	0x02 0_0000_010 = +0b1.010*2^-8 = ~0.005
0x03 0_0000_011 = +0b0.011*2^-7 = ~0.003	0x03 0_0000_011 = +0b1.011*2^-8 = ~0.005
0x04 0_0000_100 = +0b0.100*2^-7 = ~0.004	0x04 0_0000_100 = +0b1.100*2^-8 = ~0.006
0x05 0_0000_101 = +0b0.101*2^-7 = ~0.005	0x05 0_0000_101 = +0b1.101*2^-8 = ~0.006
0x06 0_0000_110 = +0b0.110*2^-7 = ~0.006	0x06 0_0000_110 = +0b1.110*2^-8 = ~0.007
0x07 0_0000_111 = +0b0.111*2^-7 = ~0.007	0x07 0_0000_111 = +0b1.111*2^-8 = ~0.007
0x08 0_0001_000 = +0b1.000*2^-7 = ~0.008	0x08 0_0001_000 = +0b1.000*2^-7 = ~0.008
0x09 0_0001_001 = +0b1.001*2^-7 = ~0.009	0x09 0_0001_001 = +0b1.001*2^-7 = ~0.009
0x0a 0_0001_010 = +0b1.010*2^-7 = ~0.010	0x0a 0_0001_010 = +0b1.010*2^-7 = ~0.010
0x0b 0_0001_011 = +0b1.011*2^-7 = ~0.011	0x0b 0_0001_011 = +0b1.011*2^-7 = ~0.011
0x0c 0_0001_100 = +0b1.100*2^-7 = ~0.012	0x0c 0_0001_100 = +0b1.100*2^-7 = ~0.012
0x0d 0_0001_101 = +0b1.101*2^-7 = ~0.013	0x0d 0_0001_101 = +0b1.101*2^-7 = ~0.013
0x0e 0_0001_110 = +0b1.110*2^-7 = ~0.014	0x0e 0_0001_110 = +0b1.110*2^-7 = ~0.014
0x0f 0_0001_111 = +0b1.111*2^-7 = ~0.015	0x0f 0_0001_111 = +0b1.111*2^-7 = ~0.015

```
minFloat                   ~ 0.0010
minFloat_no_subnormals ~ 0.0044

So dynamic range increases by > 4x
```

- Pro: Subnormals increase dynamic range
  - BFloat16 initially had no subnormals,
    implementations increasingly moving to include them
    (See e.g. https://github.com/riscv/riscv-bfloat16/issues/51)

- Con: Subnormals impose additional hardware cost
  - But existing FP8 implementations have chosen to pay that cost

WG Decision: Formats shall include subnormals

On Infinities

Recall the Transformer

Transformer: same "layers" loop, bigger numbers

$L = 32K, D_m = 16K, L \times D_m = 1GB$ Float16

```
def transformer(W, input):
    L = input.shape[0]

    # Create mask: 0 to attend, -Inf to ignore
    mask = jnp.log(jnp.tril(jnp.ones((L, L))))

    # Start with token embeddings + positional encodings
    x = W.embeddings[input, :]      # L x Dm

    # Apply the transformer layers
    for Wl in W.layers:
        x = transformer_layer(Wl, x, mask)

    # And linearly project to output dimension
    return W.out_A @ x + W.out_b
```

```
def transformer_layer(W, x, mask):
    # Layer-normalize embeddings
    t1 = standardize(x)
    t1 = W.p1A @ t1 + W.p1b

    # Multi-head self-attention
    for head in W.heads:
        # Project into this head's query/key space
        query = head.query @ t1 + head.qb        # L x Dk
        key = head.key @ t1 + head.kb            # L x Dk

        # Compute L x L attention matrix
        score = query @ key.T + mask             # L x L
        attn = softmax(tau * score)              # L x L

        value = head.value @ t1 + head.vb        # L x Dm
        self_attn = attn @ value                 # L x Dm

        x += self_attn                           # L x Dm

    # Layer-normalize embeddings
    t2 = standardize(x)
    t2 = W.p2A @ t2 + W.p2b   # L x Dm

    # Feedforward fully connected
    t2 = W.ffn1.A @ t2 + W.ffn1.b                # L x Dff
    t2 = relu(t2)
    t2 = W.ffn2.A @ t2 + W.ffn2.b                # L x Dm

    return x + t2
```

Addition of $-\infty$

FLT_MAX not a substitute for ∞ in deep learning use cases

Example: Attention masking in transformers

$$M_i \in \{0, -\infty\} \quad \text{\# Define mask}$$

$$a = \log\left(\sum_j \exp(\tau \times (A_i + M_i))\right) \quad \text{\# Compute softmax}$$

$$\text{lse}(\boldsymbol{v}) \coloneqq \log(\textstyle\sum_i v_i) \quad \text{\# … uses the common “logsumexp” function}$$

$$\text{lse}(\boldsymbol{v}) = \text{lse}(\boldsymbol{v} - \max(\boldsymbol{v})) + \max(\boldsymbol{v}) \quad \text{\# … needs rewrite for stability}$$

Using infinity	$\text{lse}(0.1 * [-224, -\infty]) \to \text{lse}(0.1 * [0, -\infty])$
Using $\text{FLT}_{\text{MAX}} (= 240)$	$\text{lse}(0.1 * [-224, -240]) \to \text{lse}(0.1 * [0, -16])$

$\text{lse}([0, -1.6])$ rather different to $\text{lse}([0, -\infty])$

Saturation to FLT_MAX may disguise hard-to-find bugs

Example: batch/layer normalization

- Make random vector, using range well:

$$M = \text{rand}\big((1, N), \text{dtype} = \text{Float8E4}\big) * 128$$

- Compute norm, perhaps carelessly (e.g. not Kahan/Blue)

$$v = \sqrt{\sum_i m_i^2}$$

- If sum overflows silently to FLT_MAX, then $v \approx 16$, plausibly scaling M

BUT: we will want saturation in some situations – see later

Infinities: Summary

- Costs:
  - 2 codes out of 256
  - Extra (small) hardware complexity vs. saturate to FLT_MAX/NaN
- Benefits: Robustness in common deep learning use-cases

On Saturation

Saturation to FLOAT_MAX or Infinity

- ML includes many dot products
- Hardware needs to vary accumulation order for speed
- "Non-sticky" saturation to FLOAT_MAX can give arbitrarily wrong answers

Computation `f = 224`	OvSAT	OvINF	OvNAN
`[ f -f  f -f] . [f f f f]`	0	0	0
`[ -f -f f  f] . [f f f f]`	34848.0	Inf	NaN
`[ f  f -f -f] . [f f f f]`	-34848.0	-Inf	NaN
`[f f] . [f f] + [-f -f] . [f f]`	0	NaN	NaN

- `OvSAT`: Saturation: `return sign(v)*FLOAT_MAX`
- `OvNAN`: NaN on overflow: `return NaN`
- `OvINF`: Infinity: `return sign(v)*Inf`

https://github.com/awf/notes/blob/main/fp8/saturation.ipynb

```
check(OvSAT, BLK=1, [f, -f, f, -f])=0.0          <-- Correct
check(OvSAT, BLK=1, [f, f, -f, -f])=-34848.0     <-- Incorrect, silently
check(OvSAT, BLK=1, [-f, -f, f, f])=34848.0      <-- Incorrect, silently

check(OvINF, BLK=1, [f, -f, f, -f])=0.0          <-- Correct
check(OvINF, BLK=1, [f, f, -f, -f])=inf          <-- Incorrect, detectable
check(OvINF, BLK=1, [-f, -f, f, f])=-inf         <-- Incorrect, detectable

check(OvINF, BLK=2, [-f, -f, f, f])=nan          <-- NaN, rather than +/-Inf
```

The above discussion shows that OvSAT may be arbitrarily and silently incorrect, surely an alarming state of affairs.

And yet, it has been used in practice to train large deep learning models, apparently without notable ill effects.

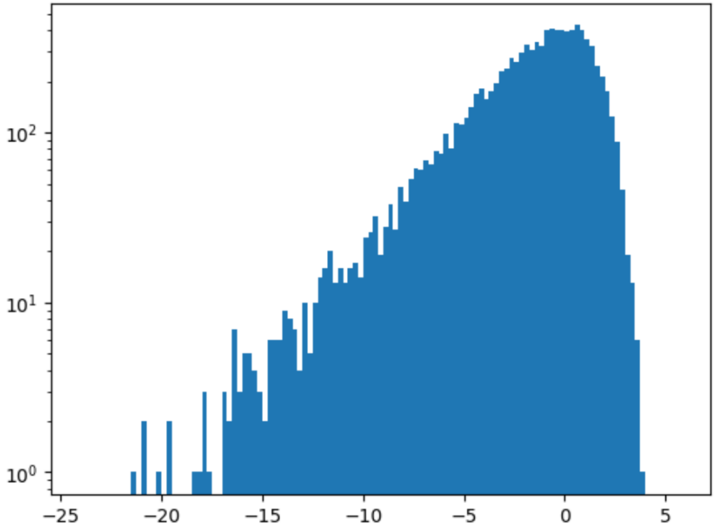
In order to explore this question, let us empirically ask some simple questions:

- Although in theory, errors can be large, how large are they in practice?
- Does the +/-Inf signal offer potentially lower errors on average (e.g. replace +/-Inf with +/-F/2)?
- How often does hierarchical reduction under OvINF in fact yield NaN?

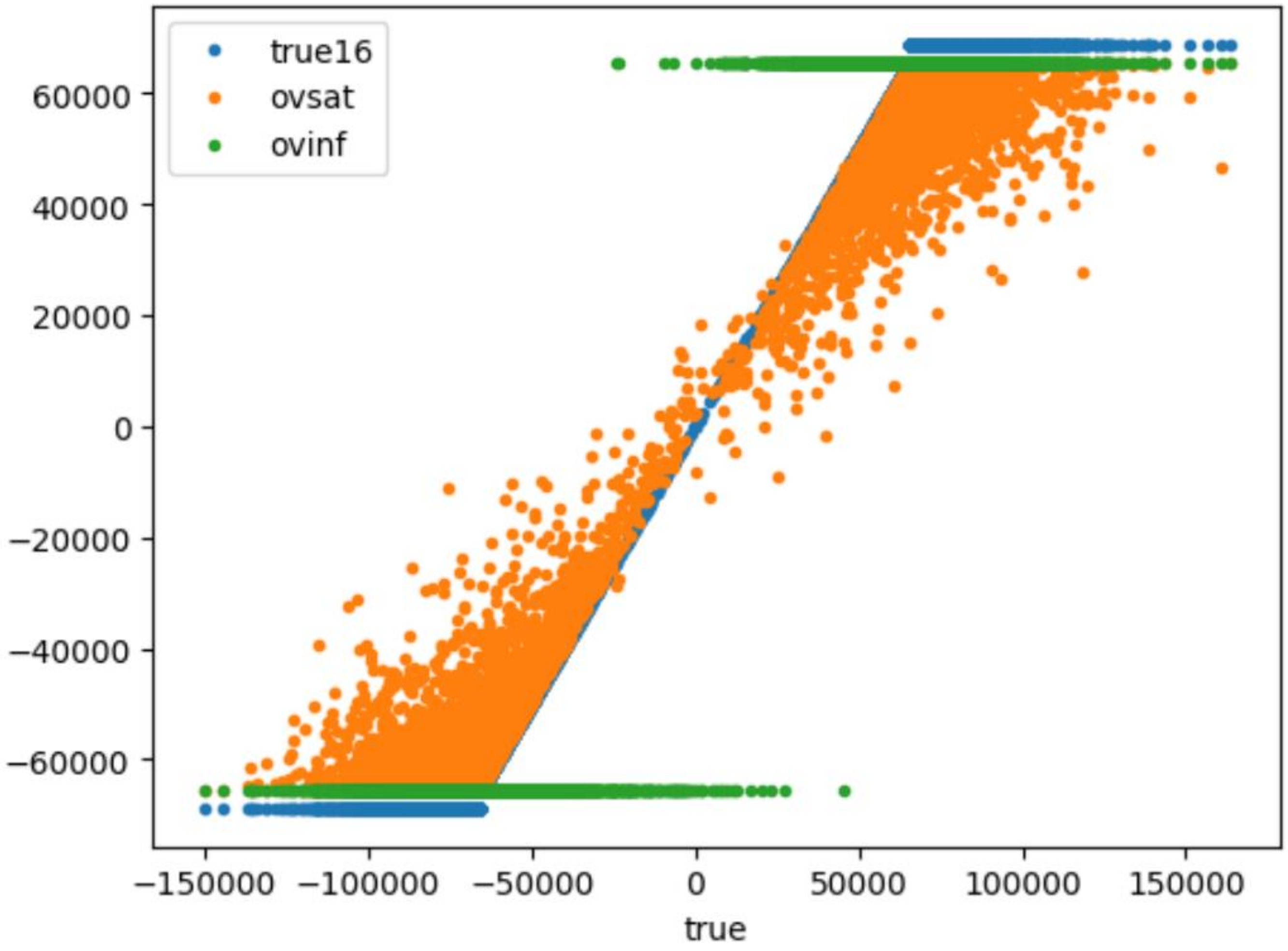
https://github.com/awf/notes/blob/main/fp8/saturation.ipynb

Saturation to FLOAT_MAX or Infinity

- `OvSAT`: Saturation: `return sign(v)*FLOAT_MAX`
- `OvNAN`: NaN on overflow: `return NaN`
- `OvINF`: Infinity: `return sign(v)*Inf`



Typical weight/activation histogram, transformer model [1]



rms_err_ovsat= 19705.81 max_err_ovsat= 111032.35
rms_err_ovinf= 19781.70 max_err_ovinf= 108611.63

blk=16, partials=float16

https://github.com/awf/notes/blob/main/fp8/saturation.ipynb

[1] 8-bit Numerical Formats for Deep Neural Networks, Badreddine Noune, Philip Jones, Daniel Justus, Dominic Masters, Carlo Luschi, 2022. https://arxiv.org/abs/2206.02915

On saturation

- Saturation can give arbitrarily wrong answers
- Overflow to infinity can give either +/-inf for a dot product
- And yet saturation works well in practice, so we need to define it
- And this discussion is completely orthogonal to the presence/absence of Inf

Operations

Operations: ToBinary{K}

Summary:

$$R \leftarrow \text{ToBinary\{K\}}\ X$$

Operands:

X: Binary8 Precision p

Result:

R: Binary$\{K\}$

X : Binary8{p}	ToBinary{K} **X**
NaN	Any [quiet?] NaN
$\pm\text{Inf}$	$\pm\text{Inf}$
X	X

All Binary8 values are exact Binary{K} values for all p for $K \geq 32$

For conversion to Binary16, some Binary8 values (e.g. for $p = 2$) are not exact Binary16 values. Therefore define

$$\text{ToBinary16}(X) \coloneqq Round\big(\text{ToBinary32(X)}\big)$$

Where $Round$ is an IEEE-754 rounding from 32->16

Operations: ConvertToBinary8

Summary:

$$R \leftarrow \mathrm{CONVERT\ SAT, RND}, X$$

Operands:

SAT: Boolean “Saturation”

RND: Enum rounding mode

X: Binary$\{k\}$ for $k \geq 16$

Result:

R: Binary8 Precision p

Note SAT and RND passed to operation “as if” in registers, but this standard says nothing about hardware implementation. They may be passed in opcodes, flags, registers, or be fixed in a given subset.

Example from CUDA: `__hadd2_sat(x,y)`
- Formats (h) and saturation in name, rounding mode in flags

Example from x86: `_mm512_add_round_ps(a, b, RND)`
- Formats in name, saturation not implemented

Again: we are specifying an input/output mapping, not an implementation

Operations: Convert

Summary:

$$R \leftarrow \text{CONVERT SAT, RND}, X$$

Operands:

SAT: Boolean "Saturation"

RND: Enum rounding mode

X: Binary$\{k\}$ for $k \geq 16$

Result:

R: Binary8 Precision p

X : Binary{K}	SAT : Bool	CONVERT SAT, X : Binary8{p}
Any NaN	*	NaN
$\pm$Inf	True	$\text{maxFloat} * \text{sign}(X)$
$\pm$Inf	False	X
-0	*	0 or NaN?
X	SAT	$\text{Round}(p, \text{SAT}, \text{RND}, X)$

CONVERT

X : Binary{K}	SAT	CONVERT SAT, X
Any NaN	*	NaN
$\pm$Inf	True	$\mathrm{maxFloat} * \mathrm{sign}(\mathrm{X})$
$\pm$Inf	False	X
-0	*	0
X	SAT	$\mathrm{Round}\langle p\rangle(\mathrm{SAT}, \mathrm{X})$

X : Binary{K}	SAT	CONVERT SAT, X
Any NaN	*	NaN
$\pm$Inf	True	$\mathrm{maxFloat} * \mathrm{sign}(\mathrm{X})$
$\pm$Inf	False	X
-0	*	NaN
X	SAT	$\mathrm{Round}\langle p\rangle(\mathrm{SAT}, \mathrm{X})$

- We have been unable to find an intentional use of negative zero in ML code.
- There is likely existing ML code in which -0 arises naturally but unintentionally (e.g. underflow), which will produce NaN under option (b) when ported to binary8.
  - These NaNs can be removed by first converting -0 to 0, but this may impose a large computational burden.
- Option B: Induces NaN results early in some troublesome computation sequences, e.g.

$$RECIP\left(SAT, RECIP(*, -\infty)\right)$$

- Implication that NEGATE(0) = NaN will likely impact ML code

Define operations in "extended reals" or "binary64"?

Add(X: $u256$, Y: $u256$, XFormat: $u7$, YFormat: $u7$, ResultFormat: $u7$, Ovf: $u2$, Rnd: $u5$) : $u256$ =
  # Add X (in XFormat) to Y (in YFormat), and return in ResultFormat
  if isNaN(X) **or** isNaN(Y) **or** (X $=$ $\pm$Inf **and** Y $= -$X) **then**
    return Encode(NaN, ResultFormat)
  end
  $\mathcal{X}$:= ToExtendedReal(X, XFormat) # $\mathcal{X} \in \mathbb{R}^\infty$
  $\mathcal{Y}$:= ToExtendedReal(Y, YFormat) # $\mathcal{Y} \in \mathbb{R}^\infty$
  $\mathcal{Z} := \mathcal{X} + \mathcal{Y}$ # "+" in extended reals, result $\mathcal{Z} \in \mathbb{R}^\infty$
  return ToBinary8($\mathcal{Z}$, ResultFormat, Ovf, Rnd)

This delegates all questions of format interpretation to a function "ToExtendedReal"

This delegates all questions of overflow and rounding to a function "ToBinary8"

All NaN handling is explicit: ToExtendedReal and ToBinary8 cannot receive/return a NaN.

Note: $\mathbb{R}^\infty = \mathbb{R} \cup \{-\infty, \infty\}$ is the usual mathematical extended reals.

Could we define ops via upconversion to IEEE 754?

Counterexample from Jeffrey Sarnoff via Nathalie Revol

Example using binary8p3:

$x = 3/1024$, $y = 49152/1$, $z = 1/131072 = 2^{-17}$

using `binary64s`, $\mathtt{fma}(x, y, z) = x \times y + z = 144.000007629453$

using `binary32s`, $\widetilde{\mathtt{fma}}(x, y, z) = 144.0$

down-converting, the best fit in `binary8p3`

converting from `binary64s`: 160.0

converting from `binary32s`: 128.0

144 is exactly halfway between 128 and 160

using `RoundNearestToEven` with 144.0 gives 128.0

using `RoundNearestToOdd` with 144.0 gives 160.0

the exact result is closer to 160.0.

In Summary…

Summary

- Defining many formats covers many future use cases
  - But of course vendors cannot support all formats
  - But it is still useful to be able to describe precisely what one's system does support
  - Subsetting already exists: vendors today often don't support all of F16,F32,F64 (other than in software)

- Total number of formats is rather large:
  - Signed: $K = 2 \ldots 15, 1 \leq P < K$, so 105 formats
  - Unsigned: $K = 2 \ldots 15, 1 \leq P \leq K$ so 119 formats
  - Total 224, which is MAX_FLOAT for $K8P4$....

- Not covered: block formats, accuracy specs

Resources

- P3109 public materials: https://github.com/P3109/Public
- My testing library (not P3109 official code): https://gfloat.readthedocs.io

Code to value mapping: binary8p4

```
0x00 00000000 0.0000.000 +0.000*2^-7 = 0.0
0x01 00000001 0.0000.001 +0.001*2^-7 = 0.0009765625
0x02 00000010 0.0000.010 +0.010*2^-7 = 0.001953125
0x03 00000011 0.0000.011 +0.011*2^-7 = 0.0029296875
0x04 00000100 0.0000.100 +0.100*2^-7 = 0.00390625
0x05 00000101 0.0000.101 +0.101*2^-7 = 0.0048828125
0x06 00000110 0.0000.110 +0.110*2^-7 = 0.005859375
0x07 00000111 0.0000.111 +0.111*2^-7 = 0.0068359375
0x08 00001000 0.0001.000 +1.000*2^-7 = 0.0078125
0x09 00001001 0.0001.001 +1.001*2^-7 = 0.0087890625
0x0a 00001010 0.0001.010 +1.010*2^-7 = 0.009765625
0x0b 00001011 0.0001.011 +1.011*2^-7 = 0.0107421875
0x0c 00001100 0.0001.100 +1.100*2^-7 = 0.01171875
...
0x73 01110011 0.1110.011 +1.011*2^6 = 88.0
0x74 01110100 0.1110.100 +1.100*2^6 = 96.0
0x75 01110101 0.1110.101 +1.101*2^6 = 104.0
0x76 01110110 0.1110.110 +1.110*2^6 = 112.0
0x77 01110111 0.1110.111 +1.111*2^6 = 120.0
0x78 01111000 0.1111.000 +1.000*2^7 = 128.0
0x79 01111001 0.1111.001 +1.001*2^7 = 144.0
0x7a 01111010 0.1111.010 +1.010*2^7 = 160.0
0x7b 01111011 0.1111.011 +1.011*2^7 = 176.0
0x7c 01111100 0.1111.100 +1.100*2^7 = 192.0
0x7d 01111101 0.1111.101 +1.101*2^7 = 208.0
0x7e 01111110 0.1111.110 +1.110*2^7 = 224.0
0x7f 01111111 0.1111.111 +1.111*2^7 = +Inf
```

```
0x80 10000000 1.0000.000 -0.000*2^-7 =  NaN
0x81 10000001 1.0000.001 -0.001*2^-7 = -0.0009765625
0x82 10000010 1.0000.010 -0.010*2^-7 = -0.001953125
0x83 10000011 1.0000.011 -0.011*2^-7 = -0.0029296875
0x84 10000100 1.0000.100 -0.100*2^-7 = -0.00390625
0x85 10000101 1.0000.101 -0.101*2^-7 = -0.0048828125
0x86 10000110 1.0000.110 -0.110*2^-7 = -0.005859375
0x87 10000111 1.0000.111 -0.111*2^-7 = -0.0068359375
0x88 10001000 1.0001.000 -1.000*2^-7 = -0.0078125
0x89 10001001 1.0001.001 -1.001*2^-7 = -0.0087890625
0x8a 10001010 1.0001.010 -1.010*2^-7 = -0.009765625
0x8b 10001011 1.0001.011 -1.011*2^-7 = -0.0107421875
0x8c 10001100 1.0001.100 -1.100*2^-7 = -0.01171875

0xf3 11110011 1.1110.011 -1.011*2^6 = -88.0
0xf4 11110100 1.1110.100 -1.100*2^6 = -96.0
0xf5 11110101 1.1110.101 -1.101*2^6 = -104.0
0xf6 11110110 1.1110.110 -1.110*2^6 = -112.0
0xf7 11110111 1.1110.111 -1.111*2^6 = -120.0
0xf8 11111000 1.1111.000 -1.000*2^7 = -128.0
0xf9 11111001 1.1111.001 -1.001*2^7 = -144.0
0xfa 11111010 1.1111.010 -1.010*2^7 = -160.0
0xfb 11111011 1.1111.011 -1.011*2^7 = -176.0
0xfc 11111100 1.1111.100 -1.100*2^7 = -192.0
0xfd 11111101 1.1111.101 -1.101*2^7 = -208.0
0xfe 11111110 1.1111.110 -1.110*2^7 = -224.0
0xff 11111111 1.1111.111 -1.111*2^7 = -Inf
```

Code to value mapping: binary8p3

```
0x00 00000000 0.00000.00 +0.00*2^-15 = 0.0
0x01 00000001 0.00000.01 +0.01*2^-15 = 7.62939453125e-06
0x02 00000010 0.00000.10 +0.10*2^-15 = 1.52587890625e-05
0x03 00000011 0.00000.11 +0.11*2^-15 = 2.288818359375e-05
0x04 00000100 0.00001.00 +1.00*2^-15 = 3.0517578125e-05
0x05 00000101 0.00001.01 +1.01*2^-15 = 3.814697265625e-05
0x06 00000110 0.00001.10 +1.10*2^-15 = 4.57763671875e-05
0x07 00000111 0.00001.11 +1.11*2^-15 = 5.340576171875e-05
0x08 00001000 0.00010.00 +1.00*2^-14 = 6.103515625e-05
0x09 00001001 0.00010.01 +1.01*2^-14 = 7.62939453125e-05
0x0a 00001010 0.00010.10 +1.10*2^-14 = 9.1552734375e-05
0x0b 00001011 0.00010.11 +1.11*2^-14 = 0.0001068115234375
0x0c 00001100 0.00011.00 +1.00*2^-13 = 0.0001220703125
...
0x73 01110011 0.11100.11 +1.11*2^12 = 7168.0
0x74 01110100 0.11101.00 +1.00*2^13 = 8192.0
0x75 01110101 0.11101.01 +1.01*2^13 = 10240.0
0x76 01110110 0.11101.10 +1.10*2^13 = 12288.0
0x77 01110111 0.11101.11 +1.11*2^13 = 14336.0
0x78 01111000 0.11110.00 +1.00*2^14 = 16384.0
0x79 01111001 0.11110.01 +1.01*2^14 = 20480.0
0x7a 01111010 0.11110.10 +1.10*2^14 = 24576.0
0x7b 01111011 0.11110.11 +1.11*2^14 = 28672.0
0x7c 01111100 0.11111.00 +1.00*2^15 = 32768.0
0x7d 01111101 0.11111.01 +1.01*2^15 = 40960.0
0x7e 01111110 0.11111.10 +1.10*2^15 = 49152.0
0x7f 01111111 0.11111.11 +1.11*2^15 = +Inf
```

```
0x80 10000000 1.00000.00 -0.00*2^-15 =  NaN
0x81 10000001 1.00000.01 -0.01*2^-15 = -7.62939453125e-06
0x82 10000010 1.00000.10 -0.10*2^-15 = -1.52587890625e-05
0x83 10000011 1.00000.11 -0.11*2^-15 = -2.288818359375e-05
0x84 10000100 1.00001.00 -1.00*2^-15 = -3.0517578125e-05
0x85 10000101 1.00001.01 -1.01*2^-15 = -3.814697265625e-05
0x86 10000110 1.00001.10 -1.10*2^-15 = -4.57763671875e-05
0x87 10000111 1.00001.11 -1.11*2^-15 = -5.340576171875e-05
0x88 10001000 1.00010.00 -1.00*2^-14 = -6.103515625e-05
0x89 10001001 1.00010.01 -1.01*2^-14 = -7.62939453125e-05
0x8a 10001010 1.00010.10 -1.10*2^-14 = -9.1552734375e-05
0x8b 10001011 1.00010.11 -1.11*2^-14 = -0.0001068115234375
0x8c 10001100 1.00011.00 -1.00*2^-13 = -0.0001220703125

0xf3 11110011 1.11100.11 -1.11*2^12 = -7168.0
0xf4 11110100 1.11101.00 -1.00*2^13 = -8192.0
0xf5 11110101 1.11101.01 -1.01*2^13 = -10240.0
0xf6 11110110 1.11101.10 -1.10*2^13 = -12288.0
0xf7 11110111 1.11101.11 -1.11*2^13 = -14336.0
0xf8 11111000 1.11110.00 -1.00*2^14 = -16384.0
0xf9 11111001 1.11110.01 -1.01*2^14 = -20480.0
0xfa 11111010 1.11110.10 -1.10*2^14 = -24576.0
0xfb 11111011 1.11110.11 -1.11*2^14 = -28672.0
0xfc 11111100 1.11111.00 -1.00*2^15 = -32768.0
0xfd 11111101 1.11111.01 -1.01*2^15 = -40960.0
0xfe 11111110 1.11111.10 -1.10*2^15 = -49152.0
0xff 11111111 1.11111.11 -1.11*2^15 = -Inf
```

From https://github.com/P3109/Public, [P3](#)