Posit Tutorial

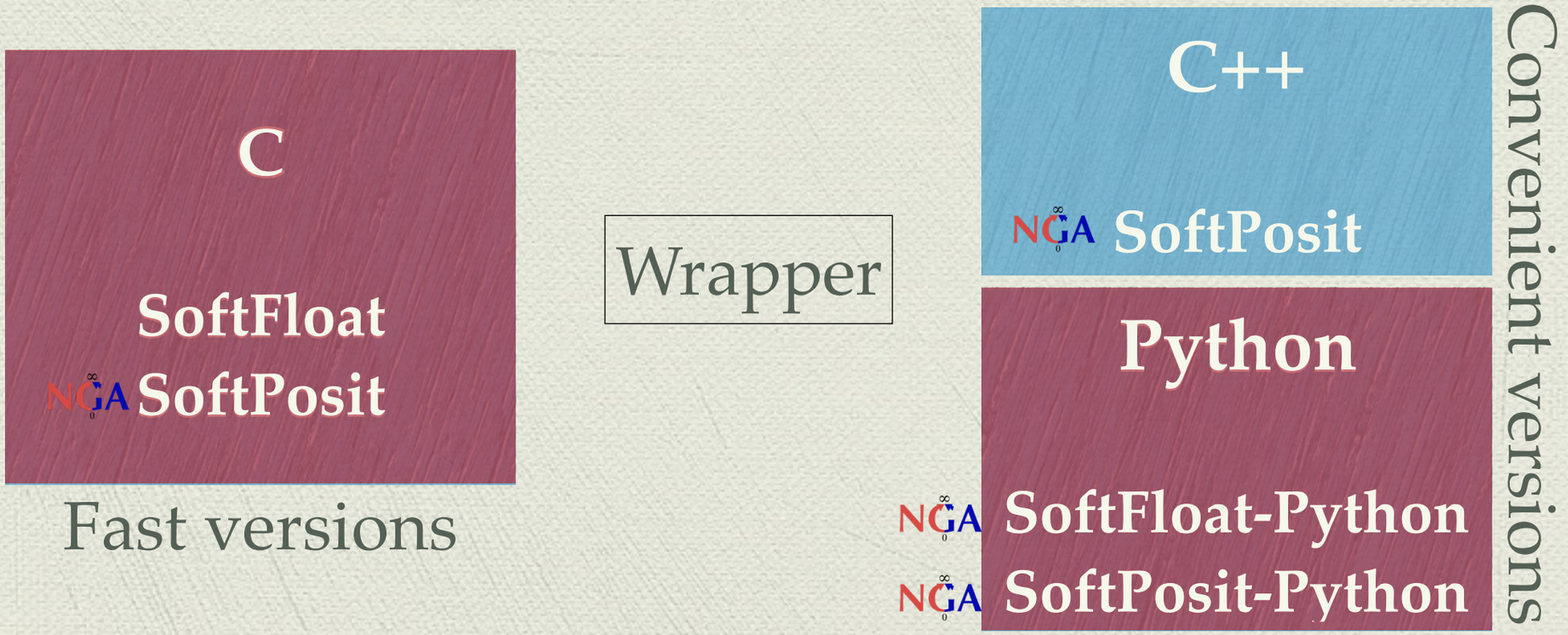
LEONG Siew Hoon (Cerlane)

14 March 2019

Sponsors

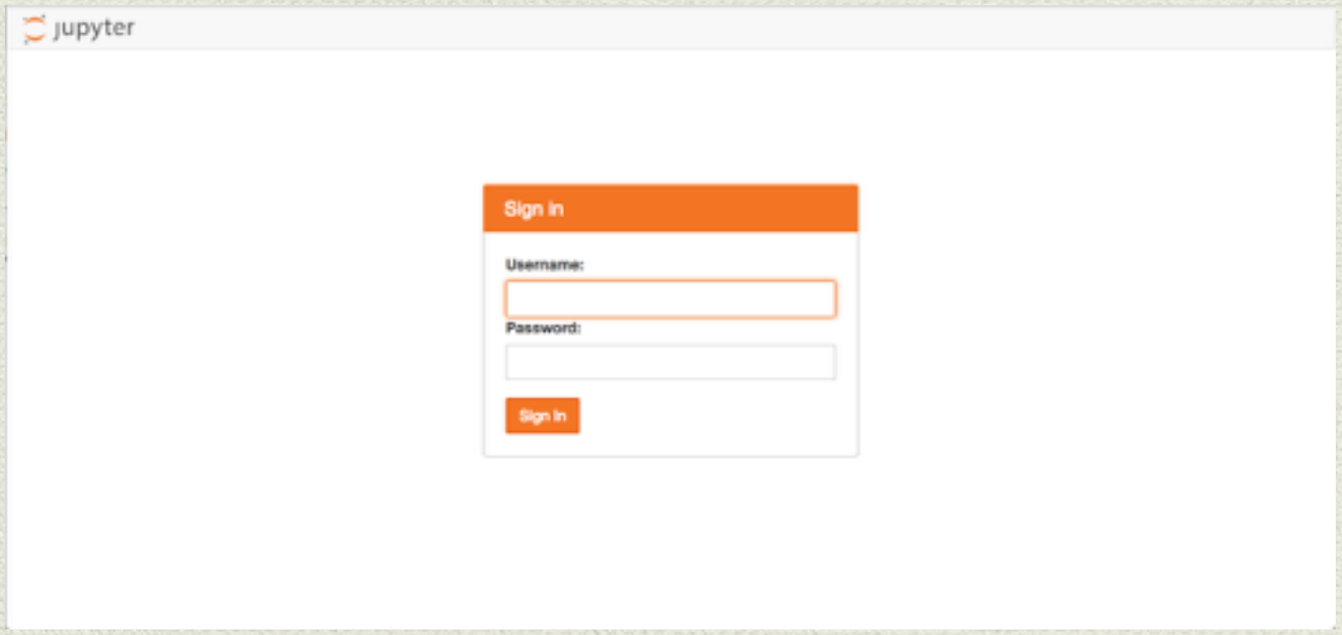
Background Info

- Two main software used:
  - Berkeley SoftFloat 3d by John Hauser (22 yrs old code)
  - NGA SoftPosit (latest from GitLab)



Setup

- Visit
  - https://posit.comp.nus.edu.sg/
- Username
  - [redacted]
- Password
  - [redacted]



Flexible size posits

posit5_2 

posit8_2 

If *es* stays the same, increasing *ps* is just padding zeros

SoftPosit:

Python:

$$pA = sp.\textbf{posit\_2}(value, \textbf{\textit{ps}})$$

C++:

$$\textbf{posit\_2}\ pA = \textbf{posit\_2}(value, \textbf{\textit{ps}})$$

C:

$$\textbf{posit\_2\_t}\ pA = \textbf{convertDoubleToPX2}(value, \textbf{\textit{ps}});$$

Flexible Posit Type

SoftPosit: Converting types

Python:

```
pB = pA.toPosit8()
pC = pA.toPosit_2(ps)
```

C++:

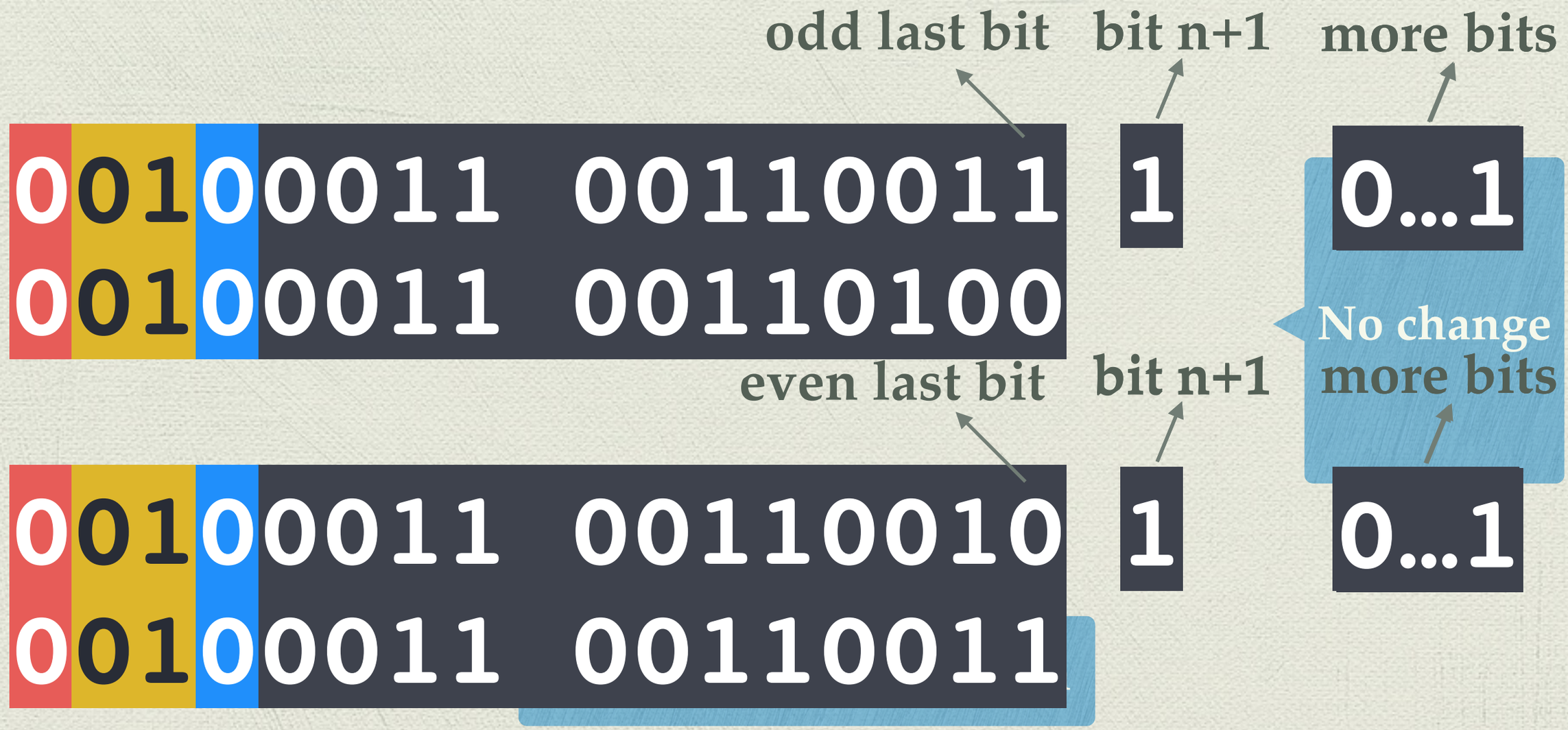
```
posit16 pB = p16(pA);
posit_2 pC = pX2(pA, ps);
```

C:

```
posit32_t pB = pX2_to_p32(pA);
posit_2 pC = pX2_to_pX2(pA, ps);
```

Banker's rounding

- Independent of bit type — Same rule applies



SoftPosit Functions

Conversion to Integers

Cast to integer → Truncation of fraction part →

1.6 → 1
2.9999 → 2

Round to "nearest even" integer → Banker's rounding →

1.1 → 1
2.9999 → 3

1.5 → 2
2.5 → 2

1.5 in binary is 1.1 → 10.0
2.5 in binary is 10.1 → 10.0

SoftPosit functions

Conversion to Integers

Cast to Integer

Return type is integer.

Python:

$iA = pA$.**toInt()**

C++:

long long int $iA = pA$.**toInt();**

C:

int64_t $iA =$ **p16_int**(pA);

SoftPosit functions

Conversion to Integers

Round to "nearest even" Integer

Return type is posit.

Python:

$pB = pA.$**rint()**

C++:

posit8 pB **= rint(**pA**);**

C:

posit16_t pB $=$ **p16_roundToInt(**pA**);**

SoftPosit functions

Conversion to Integers

Round to "nearest even" Integer

Return type is int.

Python:

iA = *pA*.**toRint()**

C++:

long long int i*A* = *pA*.**toRint();**

C:

int64_t *iA* = **p16_to_i64(p16_roundToInt**(*pA*));

SoftPosit functions

Square root

Python:

$pA = pA.\mathbf{sqrt()}$

C++:

$pA = \mathbf{sqrt}(pA);$

C:

$pA = \mathbf{p16\_sqrt}(pA);$

SoftPosit functions

Fused multiply add

Python:

pA= *pA*.**fma**(*value1*, *value2*)

C++:

pA= **fma**(*value1*, *value2*, *pA*);

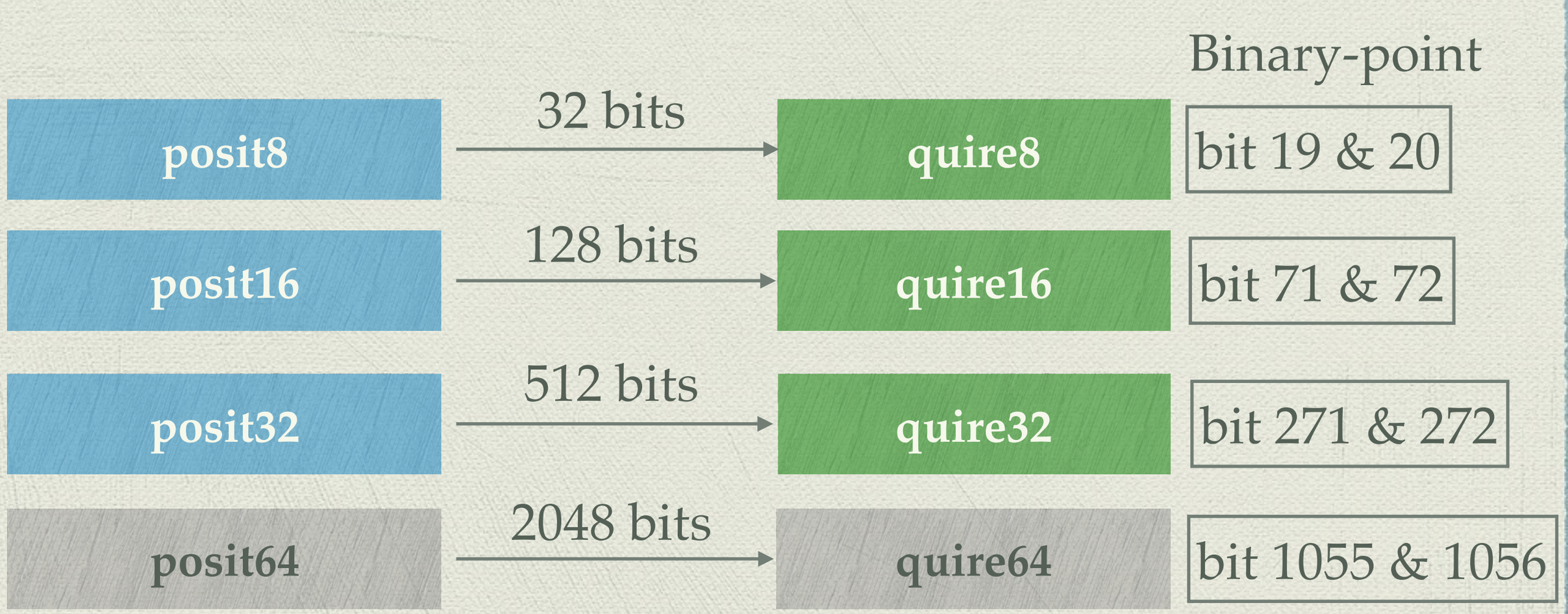
C:

pA = **p16_mulAdd**(*value1*, *value2*, *pA*);

Quire

Fixed point representation	Scratch-pad	By Kulisch and Miranker

Accumulate products → Fused dot product



Quire

$$k = \begin{cases} -m & \text{if } r = 0 \\ m - 1 & \text{if } r = 1 \end{cases}$$

Conversion of posit to quire

Step 1: Align fraction to binary point in quire

Step 2: Shift fraction based on scale computed from regime and exponent

positive: move left
negative: move right

$$scale = (2^{es} \times k) + e$$

magnitude: bits to move

posit8 `00110000` = 0.75

scale = 2^0 (-1)+0 = -1

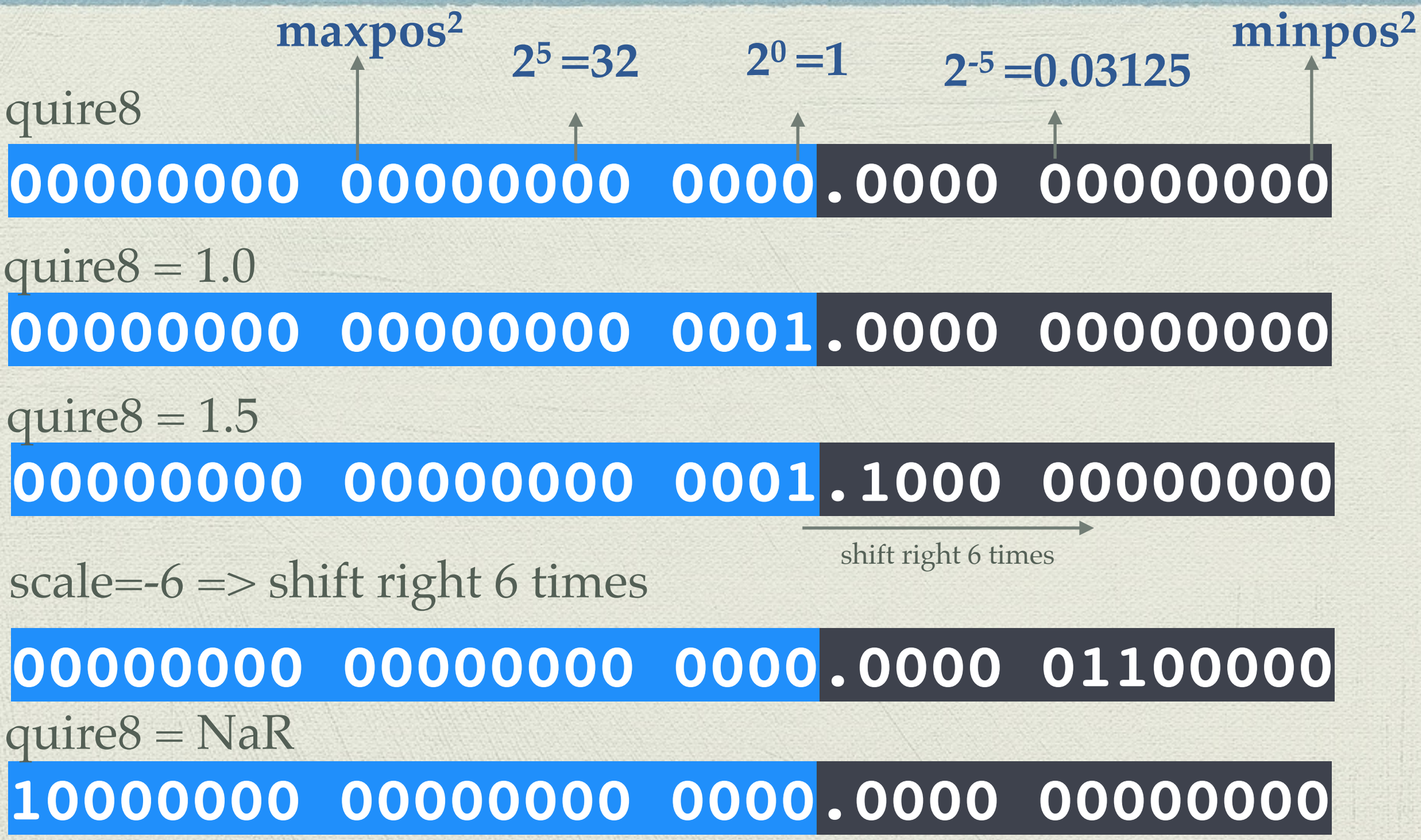
Step 1:

```
00000000 00000000 0001.1000 00000000
```

Step 2: Shift right 1 time

```
00000000 00000000 0000.1100 00000000
```

Quire


maxpos²
2⁵ =32
2⁰ =1
2⁻⁵ =0.03125
minpos²
quire8
00000000 00000000 0000.0000 00000000
quire8 = 1.0
00000000 00000000 0001.0000 00000000
quire8 = 1.5
00000000 00000000 0001.1000 00000000
shift right 6 times
scale=-6 => shift right 6 times
00000000 00000000 0000.0000 01100000
quire8 = NaR
10000000 00000000 0000.0000 00000000

Tutorial Part 3

Associative Law

$$A + B + C$$

Mathematically $(A + B) + C = A + (B + C)$

Computer reality $(A + B) + C \neq A + (B + C)$

A result of the accumulated rounding errors

⇒ A + B (round)

+ C (round)

⇒ B + C (round)

+ A (round)

Distributive Law

Mathematically $A \times (B + C) = AB + AC$

Computer reality $A \times (B + C) \neq AB + AC$

A result of the accumulated rounding errors

⇒ B + C (round)

× A (round)

⇒ A × B (round)

A × C (round)

sum (round)

Violation & Solution

- While posits can offer more precision in the most commonly used number range, it will violates associative and distributive laws

Solution:

Quire

Accumulate without rounding until the "end"

Matrix Multiplication

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$c_{11} = (a_{11} \times b_{11}) + (a_{12} \times b_{21})$$

Normal : $c_{11} = (a_{11} \times b_{11}) \overset{(3)}{+} (a_{12} \times b_{21})$ (1) (2)

FMA : $c_{11} = (a_{11} \times b_{11}) + (a_{12} \times b_{21})$ (1) (2)

Quire : $c_{11} = (a_{11} \times b_{11}) + (a_{12} \times b_{21})$ (1)

Matrix Multiplication

FMA is better than Normal?

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$$c_{11} = (a_{11} \times b_{11}) + (a_{12} \times b_{21}) + (a_{13} \times b_{31})$$

Normal :

$$c_{11} = \underset{(1)}{(a_{11} \times b_{11})} \quad + \quad \underset{(3)}{(a_{12} \times b_{21})} \quad + \quad \underset{(2)}{(a_{13} \times b_{31})} \qquad (5)\ (4)$$

FMA :

$$c_{11} = \underset{(1)}{(a_{11} \times b_{11})} \quad + \quad \underset{(2)}{(a_{12} \times b_{21})} \quad + \quad \underset{(3)}{(a_{13} \times b_{31})}$$

Quire :

$$c_{11} = \underset{(1)}{(a_{11} \times b_{11})} \quad + \quad (a_{12} \times b_{21}) \quad + \quad (a_{13} \times b_{31})$$

LU Decomposition

$$A = LU$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

- Introduced by Tadeusz Banachiewicz in 1983
- Viewed as the matrix form of Gaussian Elimination to solve square systems of linear equations

Special thanks to NUS for giving me
a VM to run the tutorial!

Questions?

Feedback form

https://bit.ly/2tDV6Oi

Please kindly fill it!