Smith College
Computer Science

Fixed-Point & **Floating-Point** Number Formats

CSC231—Assembly Language
Week #13

Dominique Thiébaut
dthiebaut@smith.edu

Reference

http://cs.smith.edu/dftwiki/index.php/CSC231_An_Introduction_to_Fixed-_and_Floating-Point_Numbers

```
public static void main(String[] args) {

   int n = 10;
   int k = -20;

   float x = 1.50;
   double y = 6.02e23;

}
```

```java
public static void main(String[] args) {

   int n = 10;
   int k = -20;

   float x = 1.50;
   double y = 6.02e23;

}
```

```java
public static void main(String[] args) {

   int n = 10;
   int k = -20;

   float x = 1.50;
   double y = 6.02e23;

}
```

Nasm knows
what **1.5** is!

```
        section    .data

x       dd         1.5
```

Nasm knows
what 1.5 is!

in memory, x is represented by

00111111 11000000 00000000 00000000
or 0x3FC00000

- **Fixed-Point Format**
- **Floating-Point Format**

Fixed-Point Format

- Used in very few applications, but **programmers know about it**.
- Some micro controllers (e.g. Arduino Uno) do not have Floating Point Units (**FPU**), and must rely on libraries to perform Floating Point operations (VERY SLOW)
- Can be used when storage is at a premium (can use small quantity of bits to represent a real number)

Review Decimal System

$$123.45 = 1\times10^2 + 2\times10^1 + 3\times10^0 + 4\times10^{-1} + 5\times10^{-2}$$

Decimal Point

Can we do the same in binary?

- Let's do it with **unsigned numbers** first:

$$1101.11 = 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2}$$

Binary Point

Can we do the same in binary?

- Let's do it with **unsigned numbers** first:

$1101.11 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$

$= 8 + 4 + 1 + 0.5 + 0.25$

$= 13.75$

Observations

- If we know where the binary point is, we do not need to "store" it anywhere. (Remember we used a bit to represent the +/- sign in 2's complement.)

- A format where the binary/decimal point is fixed between 2 groups of bits is called a **fixed-point format**.

Definition

- A number format where the numbers are **unsigned** and where we have *a* integer bits (on the left of the decimal point) and *b* fractional bits (on the right of the decimal point) is referred to as a ***U(a,b)*** *fixed-point format.*

- Value of an *N*-bit binary number in U(a,b):

$$x = (1/2^b) \sum_{n=0}^{N-1} 2^n x_n$$

Exercise 1

typical final exam question!

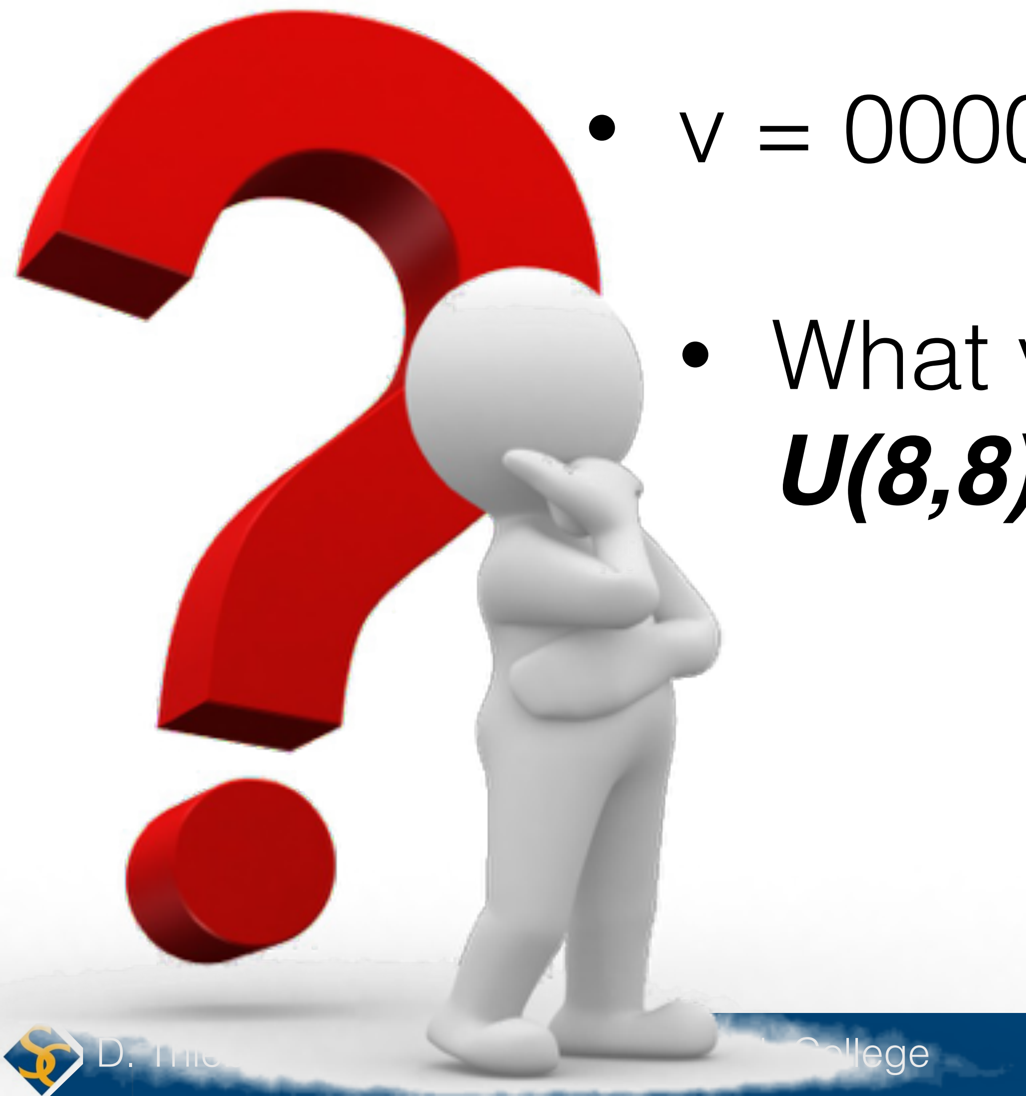
x = 1011 1111 = 0xBF

- What is the value represented by x in ***U(4,4)***
- What is the value represented by x in ***U(7,3)***

Exercise 2

typical final exam question!

- z = 00000001 00000000
- y = 00000010 00000000
  - v = 00000010 10000000
    - What values do z, y, and v represent in a ***U(8,8)*** format?



Exercise 3

typical final exam question!

- What is 12.25 in ***U(4,4)***? In ***U(8,8)***?

What about **Signed** Numbers?

Observation #1

- In an N-bit, **unsigned** integer format, the weight of the MSB is 2^{N-1}

nybble	Unsigned
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	+8
1001	+9
1010	+10
1011	+11
1100	+12
1101	+13
1110	+14
1111	+15

$N = 4$

$2^{N-1} = 2^3 = 8$

Observation #2

- In an N-bit **signed** 2's complement, integer format, the weight of the MSB is $\mathbf{-2^{N-1}}$

nybble	2's complement
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

$N=4$

$-2^{N-1} = -2^3 = -8$

Fixed-Point Signed Format

- **Fixed-Point signed** format = sign bit + *a* integer bits + *b* fractional bits = *N* bits = **A(*a*, *b*)**

- *N* = number of bits = 1 + a + b

- Format of an *N*-bit A(a, b) number:

$$x = (1/2^b)\left[-2^{N-1}x_{N-1} + \sum_{0}^{N-2} 2^n x_n\right],$$

Examples in A(7,8)

- 000000001 00000000 = 00000001 . 00000000 = ?
- 100000001 00000000 = 10000001 . 00000000 = ?
- 00000010 00000000 = 0000010 . 00000000 = ?
- 10000010 00000000 = 1000010 . 00000000 = ?
- 00000010 10000000 = 00000010 . 10000000 = ?
- 10000010 10000000 = 10000010 . 10000000 = ?

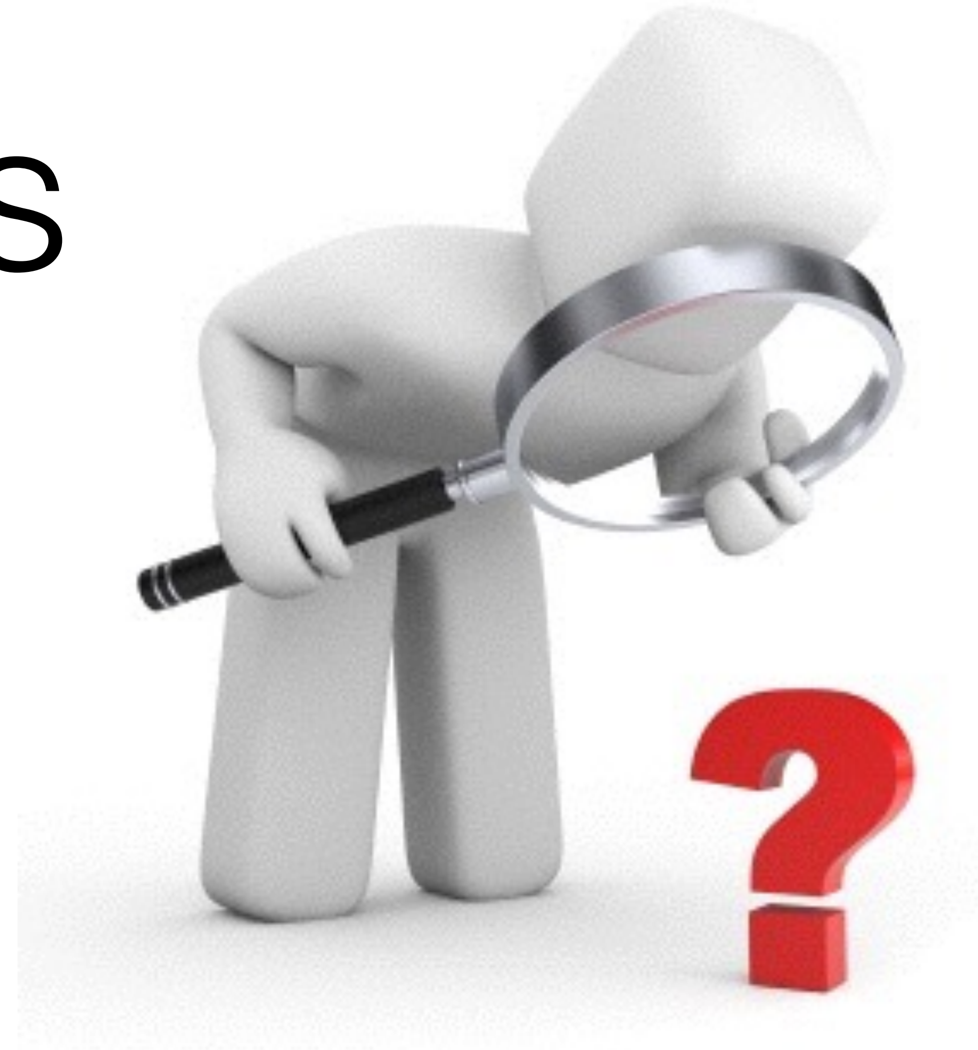
Examples in A(7,8)

- 000000001 00000000 = 00000001 . 00000000 = 1d
- 100000001 00000000 = 10000001 . 00000000 = -128 + 1 = -127d
- 00000010 00000000 = 0000010 . 00000000 = 2d
- 10000010 00000000 = 1000010 . 00000000 = -128 + 2 = -126d
- 00000010 10000000 = 00000010 . 10000000 = 2.5d
- 10000010 10000000 = 10000010 . 10000000 = -128 + 2.5 = -125.5d

Exercises

- What is -1 in ***A(7,8)***?
- What is -1 in ***A(3,4)***?
- What is 0 in ***A(7,8)***?
- What is the smallest number one can represent in ***A(7,8)***?
- The largest in ***A(7,8)***?

Exercises



- What is the largest number representable in ***U(a, b)***?
- What is the smallest number representable in ***U(a, b)***?
- What is the largest positive number representable in ***A(a, b)***?
- What is the smallest negative number representable in ***A(a, b)***?

We Stopped Here Last Time…

http://i.imgur.com/doh3mlZ.jpg

- **Fixed-Point Format**
  - **Definitions**
    - Range
    - Precision
    - Accuracy
    - Resolution
- Floating-Point Format

Range

- Range = difference between most positive and most negative numbers.

- **Unsigned Range**:
  The range of **U(*a*, *b*)** is $0 \leq x \leq 2^a - 2^{-b}$

- **Signed Range**:
  The range of **A(*a*, *b*)** is $-2^a \leq x \leq 2^a - 2^{-b}$

Precision

2 different definitions

- **Precision** = *b*, the number of fractional bits

  https://en.wikibooks.org/wiki/Floating_Point/Fixed-Point_Numbers

- **Precision** = *N*, the total number of bits

  Randy Yates, Fixed Point Arithmetic: An Introduction, Digital Signal Labs, July 2009.
  http://www.digitalsignallabs.com/fp.pdf

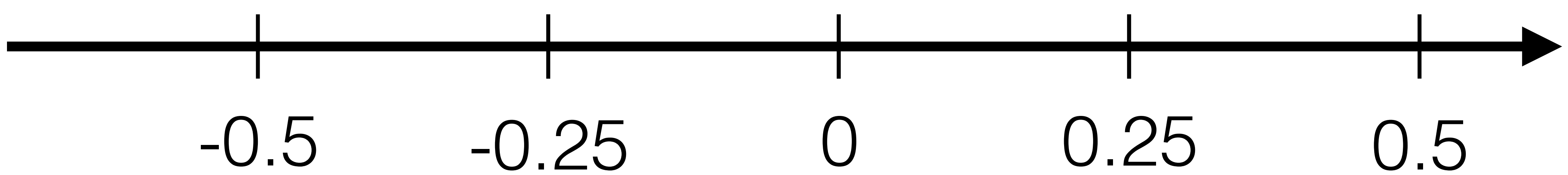
Resolution

- The **resolution** is the smallest non-zero magnitude representable.
- The **resolution** is the size of the intervals between numbers represented by the format
- Example: ***A(13, 2)*** has a resolution of 0.25.

A(13, 2) —> sbbb bbbb bbbb bb . bb

$2^{-2} = 0.25$

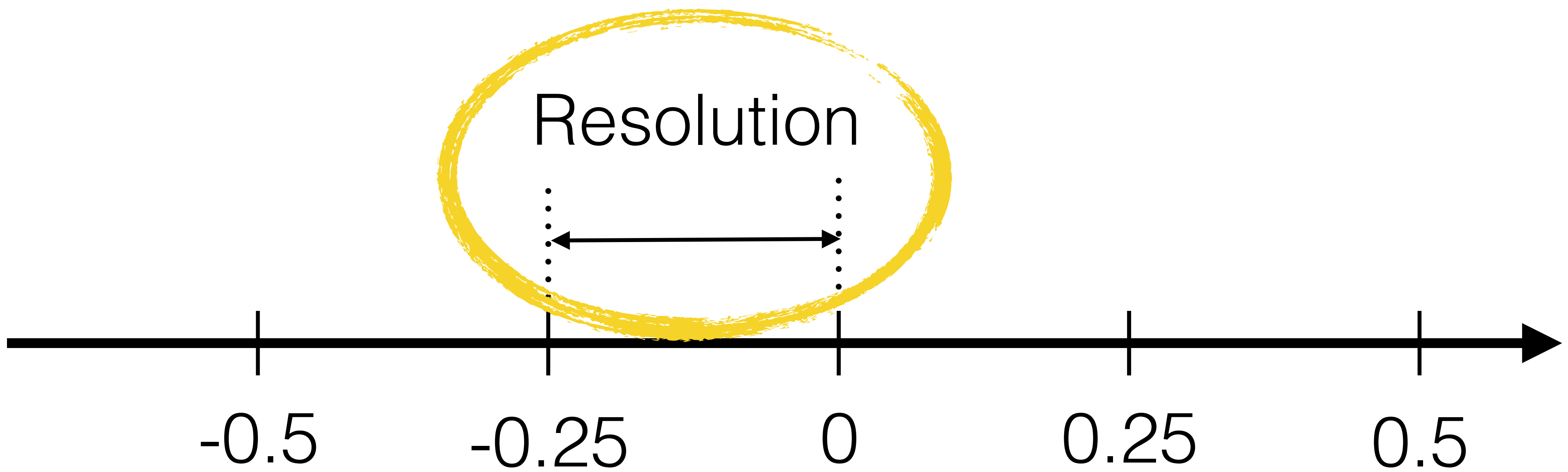
$2^{-1} = 0.5$



A(13, 2) —> sbbb bbbb bbbb bb . bb

$2^{-2} = 0.25$

$2^{-1} = 0.5$



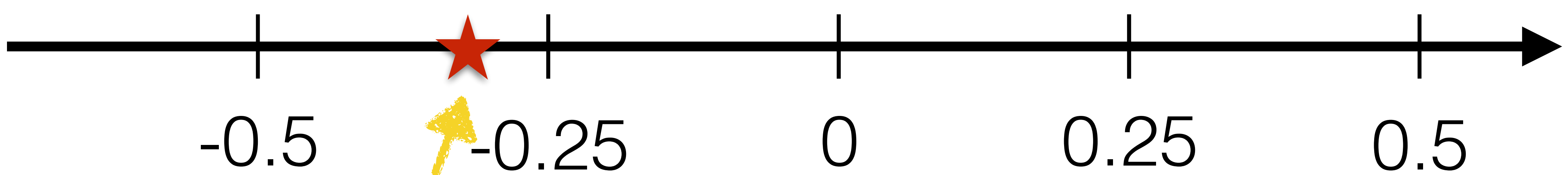
Accuracy

- The **accuracy** is the largest magnitude of the difference between a number and its representation.
- **Accuracy** = 1/2 **Resolution**

A(13, 2) —> sbbb bbbb bbbb bb . bb

$2^{-2}=0.25$

$2^{-1}=0.5$

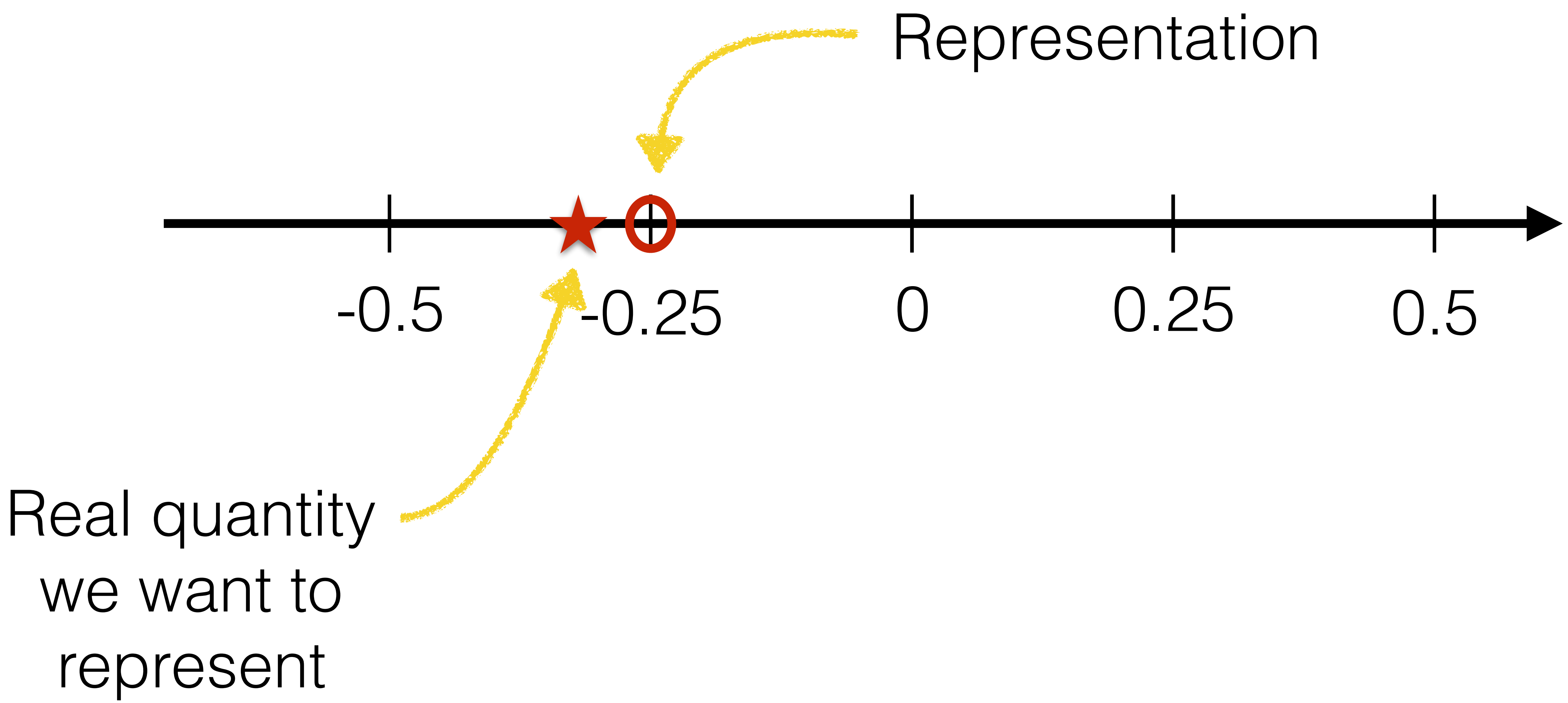


Real quantity we want to represent

A(13, 2) —> sbbb bbbb bbbb bb . bb

$2^{-2} = 0.25$

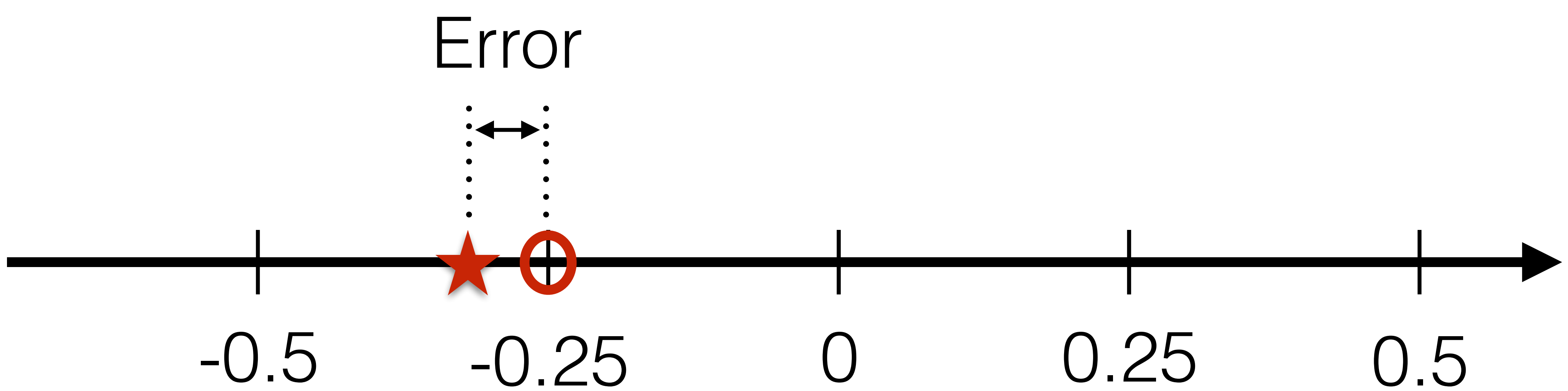
$2^{-1} = 0.5$



A(13, 2) —> sbbb bbbb bbbb bb . bb

$2^{-2} = 0.25$

$2^{-1} = 0.5$

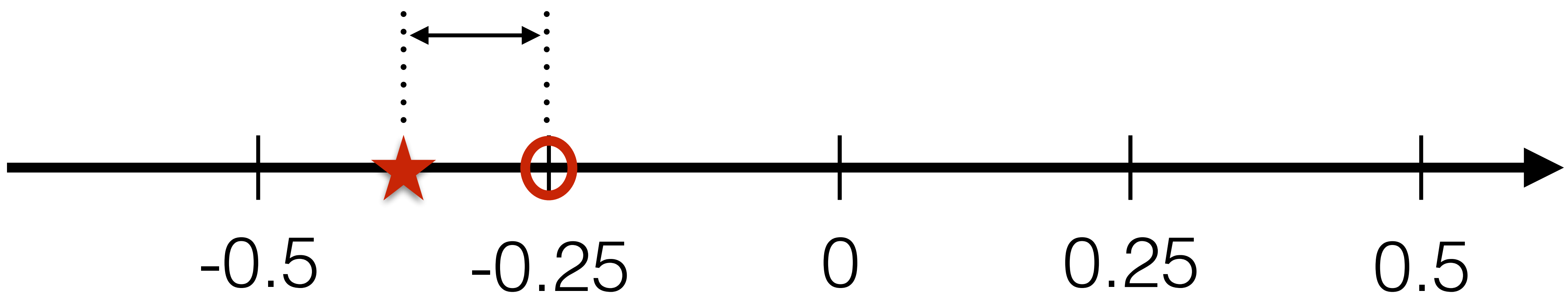


A(13, 2) —> sbbb bbbb bbbb bb . bb

$2^{-2} = 0.25$

$2^{-1} = 0.5$

Largest Error = **Accuracy**



Questions in search of answers…

- What is the accuracy of an U(7,8) number format?
- How good is U(7,8) at representing small numbers versus representing larger numbers? In other words, **is the format treating small numbers *better* than large numbers, or the opposite?**

- **Fixed-Point Format**
- **Floating-Point Format**

IEEE
Floating-Point
Number Format

A bit of history…

http://datacenterpost.com/wp-content/uploads/2014/09/Data-Center-History.png

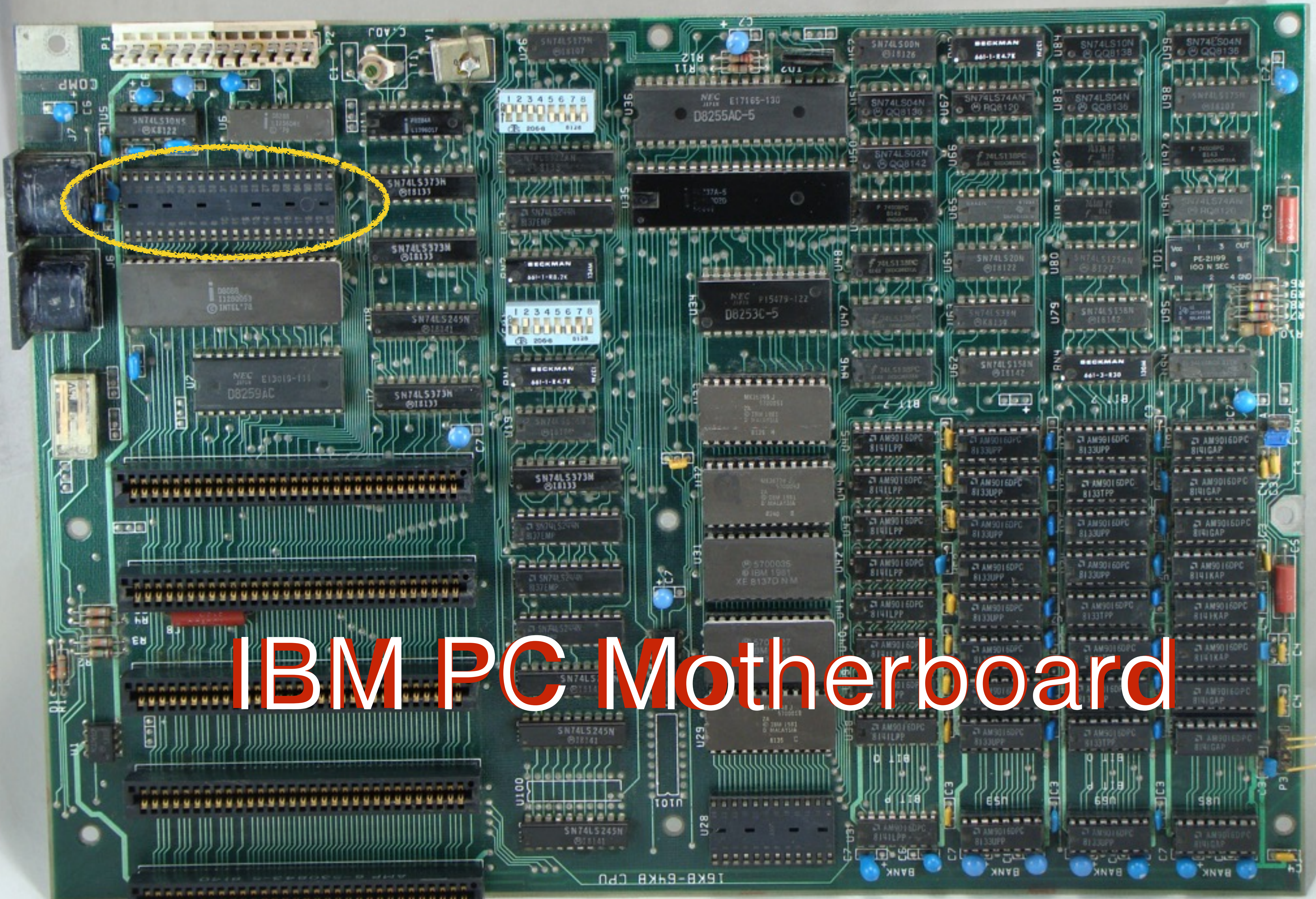
- 1960s, 1970s: many different ways for computers to **represent** and **process** real numbers. Large variation in way real numbers were operated on

- 1976: **Intel** starts design of first hardware floating-point **co-processo**r for 8086. Wants to define a **standard**

- 1977: Second meeting under umbrella of **Institute for Electrical and Electronics Engineers** (IEEE). Mostly microprocessor makers (IBM is observer)

- Intel first to put whole **math library** in a processor

IBM PC Motherboard

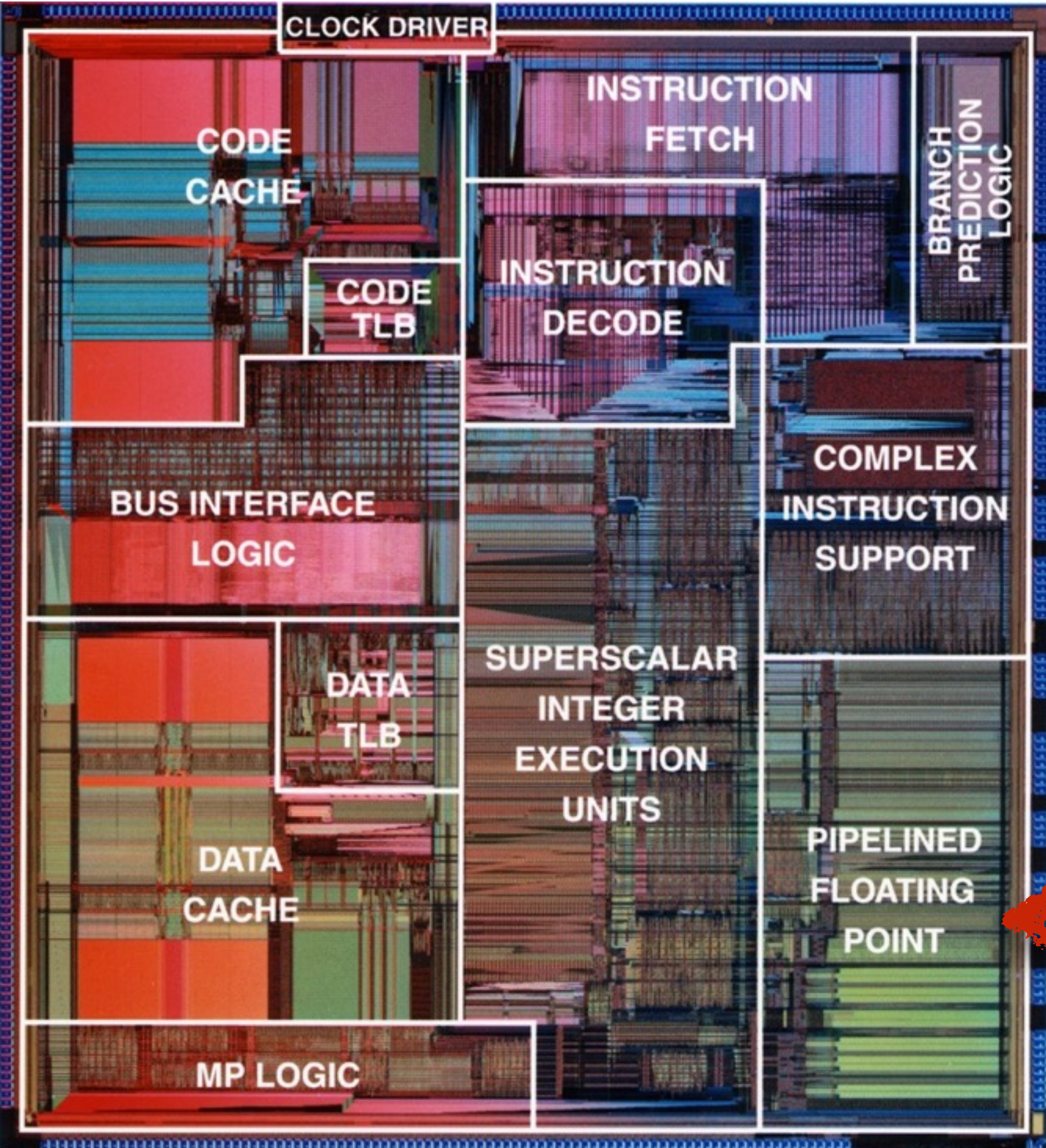


http://commons.wikimedia.org/wiki/File:IBM_PC_Motherboard_(1981).jpg

Intel Coprocessors

Intel		
Processor	Year	Description
8087	1980	Numeric coprocessor for 8086 and 8088 processors.
80C187	19??	Math coprocessor for 80C186 embedded processors.
80287	1982	Math coprocessor for 80286 processors.
80387	1987	Math co-processor for 80386 processors.
80487	1991	Math co-processor for SX versions of 80486 processors.
Xeon Phi	2012	Multi-core co-processor for Xeon CPUs.

(Early) Intel Pentium



http://semiaccurate.com/assets/uploads/2012/06/1993_intel_pentium_large.jpg

Integrated Coprocessor

Some Processors that do not contain FPUs

- Some ARM processors
- Arduino Uno
- Others

Some Processors that do not contain FPUs

*Few people have heard of ARM Holdings, even though sales of devices containing its flavor of chips are projected to be 25 times that of Intel. The chips found in **99 percent** of the world's smartphones and tablets are ARM designs. **About 4.3 billion people, 60 percent of the world's population, touch a device carrying an ARM chip each day**.*

Ashlee Vance, Bloomberg, Feb 2014

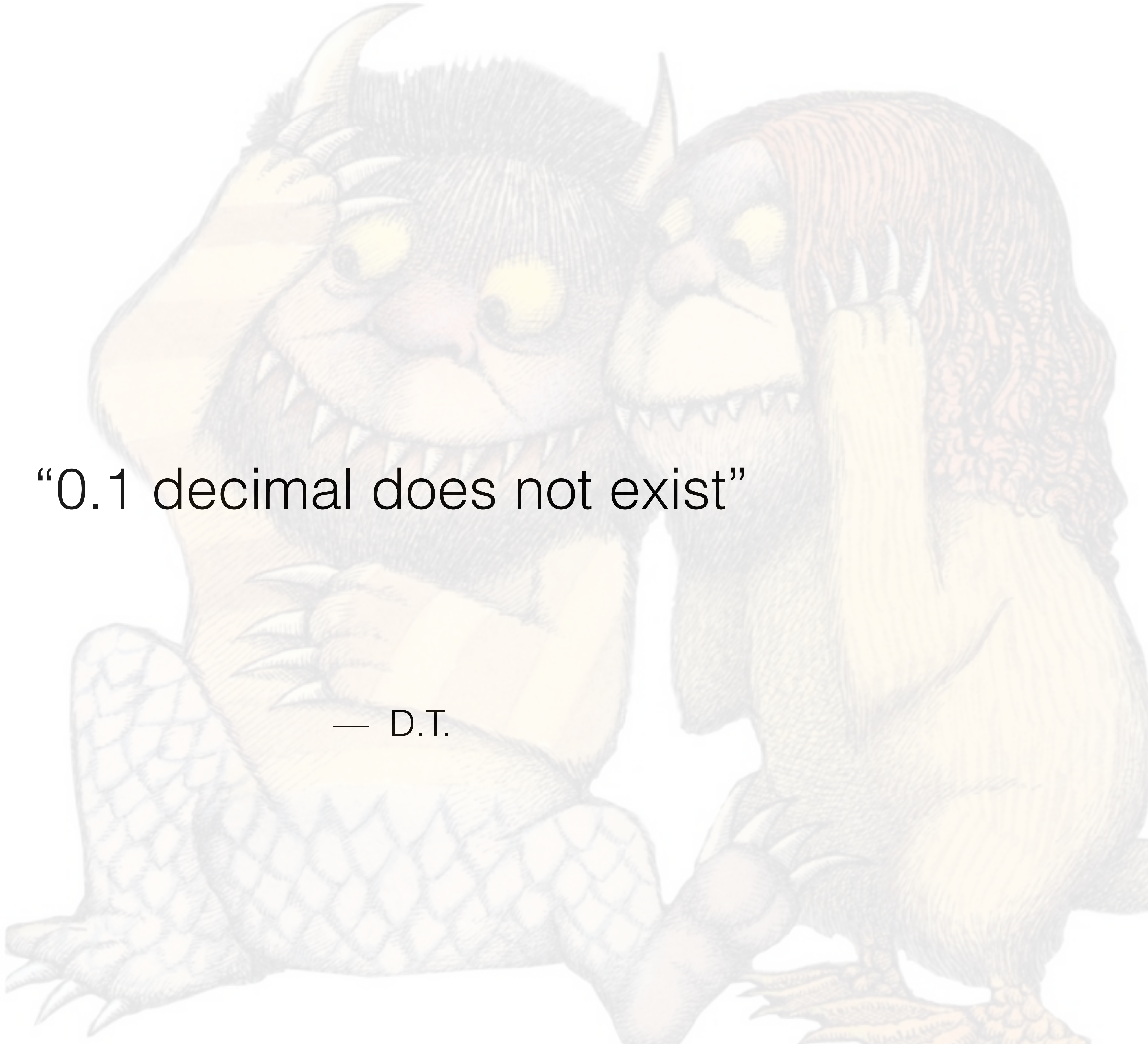
How Much Slower is Library vs FPU operations?

- Cristina Iordache and Ping Tak Peter Tang, "An Overview of Floating-Point Support and Math Library on the Intel XScale Architecture", In *Proceedings IEEE Symposium on Computer Arithmetic*, pages 122-128, **2003**

- http://stackoverflow.com/questions/15174105/performance-comparison-of-fpu-with-software-emulation

Library-emulated FP operations = **10 to 100 times slower** than hardware FP operations executed by FPU

Floating Point Numbers Are Weird…



“0.1 decimal does not exist”

— D.T.

```java
import java.util.*;

public class SomeFloats {
        public static void main(String args[]) {
            float x = 6.02E23f,
                y = -0.000001f,
                z = 1.23456789E-19f,
                t = -1.0f,
                u = 8000000000f;

            System.out.println(   "\nx = " + x
                                + "\ny = " + y
                                + "\nz = " + z
                                + "\nt = " + t
                                + "\nu = " + u );

        }
}
```

```java
import java.util.*;

public class SomeFloats {
        public static void main(String args[]) {
            float x = 6.02E23f,
                y = -0.000001f,
                z = 1.23456789E-19f,
                t = -1.0f,
                u = 8000000000f;

            System.out.println(    "\nx = " + x
                                 + "\ny = " + y
                                 + "\nz = " + z
                                 + "\nt = " + t
                                 + "\nu = " + u );

        }
}
```

```
231b@aurora ~/handout $ java SomeFloats

x = 6.02E23
y = -1.0E-6
z = 1.2345678E-19
t = -1.0
u = 8.0E9
```

$$1.230$$

$$= 12.30\ 10^{-1}$$

$$= 123.0\ 10^{-2}$$

$$= 0.123\ 10^{1}$$

IEEE Format

- 32 bits, single precision (floats in Java)
- 64 bits, double precision (doubles in Java)
- 80 bits*, extended precision (C, C++)

$$x = +/-\ 1.bbbbbb....bbb \times 2^{bbb...bb}$$

* 80 bits in assembly = 1 Tenbyte

10110.01

10110.01

1.011001×2^4

10110.01

1.011001×2^4

1.011001×2^{100}

$$10110.01$$

$$1.011001 \times 2^{4}$$

$$+\ 1.011001 \times 2^{100}$$

10110.01

1.011001×2^4

$\boxed{+}\ 1.\boxed{011001} \times 2^{\boxed{100}}$

$\boxed{0}\ \boxed{011001}\ \boxed{100}$

0 011001 100

10110.01

$\boxed{0}$ $\boxed{011001}$ $\boxed{100}$

Observations

$$x = +/- 1.bbbbbb....bbb \times 2^{bbb...bb}$$

- +/- is the sign. It is represented by a bit, equal to **0** if the number is **positive**, **1** if **negative**.
- the part 1.bbbbbb....bbb is called the **mantissa**
- the part bbb...bb is called the **exponent**
- **2** is the **base** for the exponent (could be different!)
- the number is **normalized** so that its binary point is moved to the right of the leading 1.
- because the leading bit will always be 1, we don't need to store it. This bit will be an **implied bit**.

IEEE 754 Converter

This page allows you to convert between the decimal representation of numbers (like "1.02") and the binary format used by all modern CPUs (IEEE 754 floating point). The conversion is limited to single precision numbers (32 Bit). The purpose of this webpage is to help you understand floating point numbers.

IEEE 754 Converter (JavaScript), V0.12

Note: This JavaScript-based version is still under development, please report errors here.

	Sign	Exponent	Mantissa
Value:	+1	2^{-4}	1.600000023841858
Encoded as:	0	123	5033165
Binary:			

Decimal Representation	0.1
Binary Representation	00111101110011001100110011001101
Hexadecimal Representation	0x3dcccccd
After casting to double precision	0.10000000149011612

http://www.h-schmidt.net/FloatConverter/IEEE754.html

Interlude…

```
for ( double d = 0; d != 0.3; d += 0.1 )
      System.out.println( d );
```

Normalization (in decimal)

(normal = standard form)

y = 123.456

$y = 1.23456 \times 10^2$

Normalization (in binary)

$y = 1000.100111$ (8.609375d)

$y = 1.000100111 \times 2^3$

Normalization (in binary)

$y = 1000.100111$

$y = 1.000100111 \times 2^3$

decimal

Normalization (in binary)

$$y = 1000.100111$$

decimal

$$y = 1.000100111 \times 2^3$$

binary

$$y = 1.000100111 \times 10^{11}$$

$$+1.000100111 \times 10^{11}$$

0	1000100111	11
sign	mantissa	exponent

But, remember,
all* numbers have
a leading 1, so, we can pack
the bits even more
efficiently!

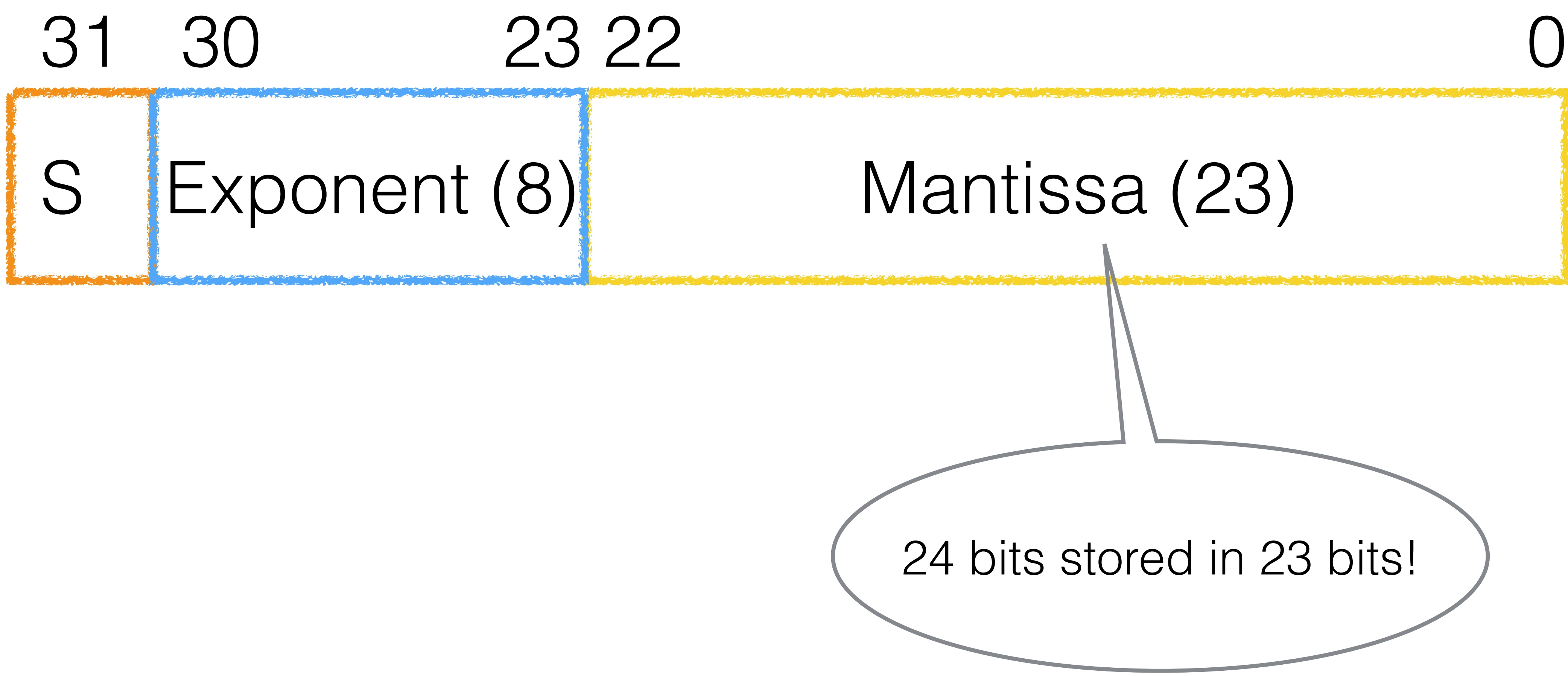
*really?

implied bit!

$+1.000100111 \times 10^{11}$

0	0001001110	11
sign	mantissa	exponent

IEEE Format



$$y = 1000.100111$$

$$y = 1.000100111 \ 2^3$$

$$y = 1.000100111 \ 10^{11}$$

31	30 23	22 0
0	bbbbbbbb	0001001110000…………….0

$$y = 1.000100111 \times 10^{11}$$

31	30 … 23	22 … 0
0	bbbbbbbb	0001001110000…………….0

Why not 00000011 ?

How is the exponent coded?

bbbbbbbb

bias of 127

real exponent	stored exponent	Comments
-126	0	Special Case #1
-126 -125 -124 -123 . . . -1	1 2 3 4 . . . 126	
0	127	
1 2 3 . . . 127	128 129 130 . . . 254	
128	255	Special Case #2

bbbbbbbb

real exponent	stored exponent	Comments
-126	0	Special Case #1
-126 -125 -124 -123 . . . -1	1 2 3 4 . . . 126	
0	127	
1 2 3 . . . 127	128 129 130 . . . 254	
128	255	Special Case #2

$$y = 1.000100111 \times 10^{11}$$

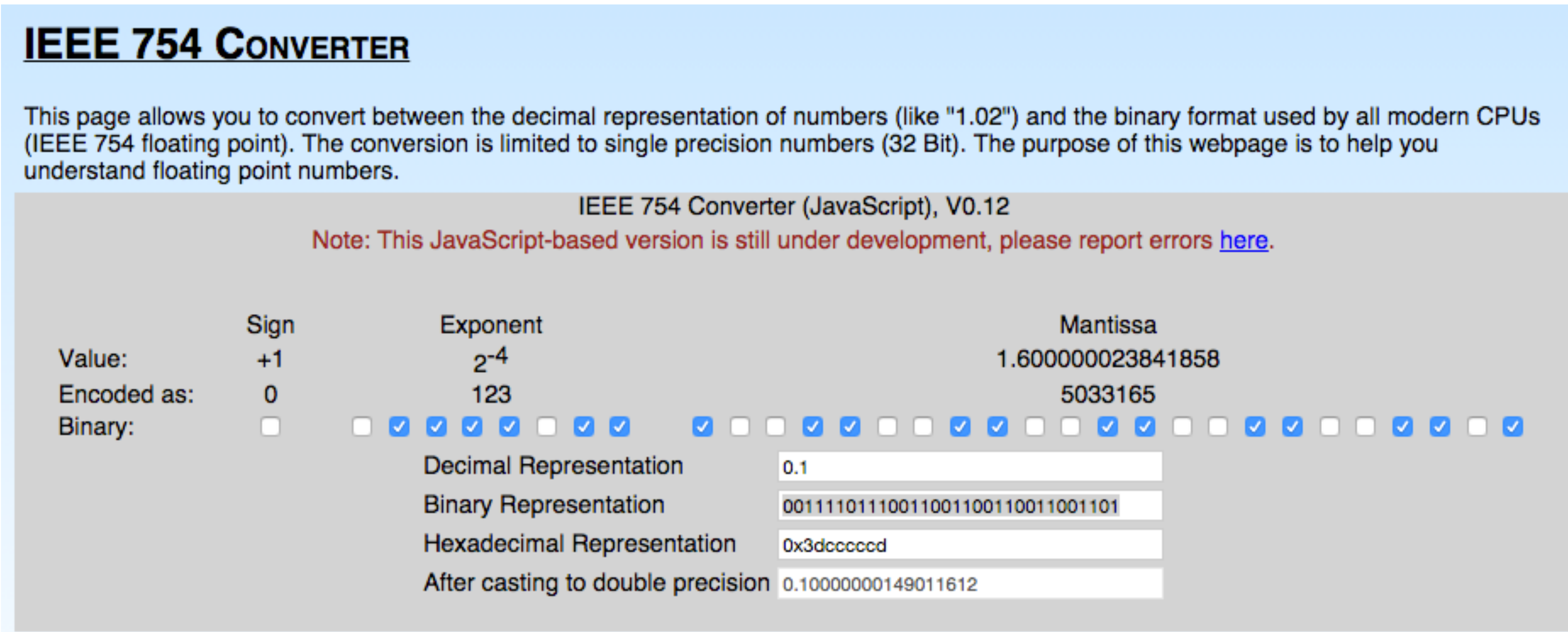
31	30 23	22 0
0	10000010	0001001110000…………….0

Ah! 3 represented by
130 = 128 + 2

1.0761719 * 2^3

= 8.6093752

Verification
8.6093752 in IEEE FP?



http://www.h-schmidt.net/FloatConverter/IEEE754.html

Exercises

- How is 1.0 coded as a 32-bit floating point number?
- What about 0.5?
- 1.5?
- -1.5?
- what floating-point value is stored in the 32-bit number below?

1 | 1000 0011 | 111 1000 0000 0000 0000 0000

what about 0.1?

0.1 decimal, in 32-bit precision, IEEE Format:

0   01111011   10011001100110011001101

0.1 decimal, in 32-bit precision, IEEE Format:

0 01111011 10011001100110011001101

Value in double-precision: **0.10000000149011612**

NEVER
NEVER
COMPARE FLOATS
OR DOUBLES FOR
EQUALITY!
N-E-V-E-R !

```
for ( double d = 0; d != 0.3; d += 0.1 )
      System.out.println( d );
```

Special Cases

bbbbbbbb

real exponent	stored exponent	Comments
-126	0	Special Case #1
-126 -125 -124 -123 . . . -1	1 2 3 4 . . . 126	
0	127	
1 2 3 . . . 127	128 129 130 . . . 254	
128	255	Special Case #2

Zero

- Why is it special?
- 0.0 = 0 00000000 00000000000000000000000

bbbbbbbb

real exponent	stored exponent	Comments
-126	0	Special Case #1
-126 -125 -124 -123 . . . -1	1 2 3 4 . . . 126	
0	127	
1 2 3 . . . 127	128 129 130 . . . 254	
128	255	Special Case #2

if mantissa is 0:
number = **0.0**

Very Small Numbers

- Smallest numbers have stored exponent of 0.
- In this case, the implied 1 is omitted, and the exponent is -126 (not -127!)

bbbbbbbb

real exponent	stored exponent	Comments
-126	0	Special Case #1
-126 -125 -124 -123 . . . -1	1 2 3 4 . . . 126	
0	127	
1 2 3 . . . 127	128 129 130 . . . 254	
128	255	Special Case #2

if mantissa is 0:
number = **0.0**
if mantissa is !0:
no hidden 1

Very Small Numbers

- Example: 0 00000000 00100000000000000000000

$\boxed{0}$ | $\boxed{00000000}$ | $\boxed{001000...000}$

$\boxed{+}$ $\boxed{(2^{-126})}$ x $\boxed{(\ 0.001\ )}$ binary

$+\ (2^{-126})\ \times\ (\ 0.125\ )\qquad = 1.469\ 10^{-39}$

Special Cases

bbbbbbbb

real exponent	stored exponent	Comments
-126	0	Special Case #1
-126 -125 -124 -123 . . . -1	1 2 3 4 . . . 126	
0	127	
1 2 3 . . . 127	128 129 130 . . . 254	
128	255	Special Case #2

✓

?

Very large numbers

- stored exponent = 1111 1111
- if the mantissa is = 0 → **+/- ∞**

Very large numbers

- stored exponent = 1111 1111
- if the mantissa is = 0 → +/- ∞

Very large numbers

- stored exponent = 1111 1111
- if the mantissa is = 0 ➡ **+/- ∞**
- if the mantissa is != 0 ➡ **NaN**

Very large numbers

- stored exponent = 1111 1111
- if the mantissa is = 0 ➡ **+/- ∞**
- if the mantissa is != 0 ➡ **NaN = Not-a-Number**

Very large numbers

- stored exponent = 1111 1111
- if the mantissa is = 0 ==> **+/- ∞**
- if the mantissa is != 0 ==> **NaN**

NaN is *sticky*!

- 0 11111111 00000000000000000000000 = + ∞
- 1 11111111 00000000000000000000000 = - ∞
- 0 11111111 10000010000000000000000 = NaN

Operations that create NaNs (http://en.wikipedia.org/wiki/NaN):

- The **divisions** 0/0 and $\pm\infty/\pm\infty$
- The **multiplications** $0\times\pm\infty$ and $\pm\infty\times 0$
- The **additions** $\infty + (-\infty)$, $(-\infty) + \infty$ and equivalent subtractions
- The **square root** of a negative number.
- The **logarithm** of a negative number
- The **inverse sine or cosine** of a number that is less than -1 or greater than $+1$

Generating NaNs

```java
// http://stackoverflow.com/questions/2887131/when-can-java-produce-a-nan
import java.util.*;
import static java.lang.Double.NaN;
import static java.lang.Double.POSITIVE_INFINITY;
import static java.lang.Double.NEGATIVE_INFINITY;

public class GenerateNaN {
    public static void main(String args[]) {
        double[] allNaNs = { 0D / 0D,
                POSITIVE_INFINITY / POSITIVE_INFINITY,
                POSITIVE_INFINITY / NEGATIVE_INFINITY,
                NEGATIVE_INFINITY / POSITIVE_INFINITY,
                NEGATIVE_INFINITY / NEGATIVE_INFINITY,
                0 * POSITIVE_INFINITY,
                0 * NEGATIVE_INFINITY,
                Math.pow(1, POSITIVE_INFINITY),
                POSITIVE_INFINITY + NEGATIVE_INFINITY,
                NEGATIVE_INFINITY + POSITIVE_INFINITY,
                POSITIVE_INFINITY - POSITIVE_INFINITY,
                NEGATIVE_INFINITY - NEGATIVE_INFINITY,
                Math.sqrt(-1),
                Math.log(-1),
                Math.asin(-2),
                Math.acos(+2), };
        System.out.println(Arrays.toString(allNaNs));
        System.out.println(NaN == NaN);
        System.out.println(Double.isNaN(NaN));
    }
}
```

```
[NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN]
false
true
```

Range of Floating-Point Numbers

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm\ 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$	$\pm\ 2^{-126}$ to $(2-2^{-23})\times 2^{127}$	$\pm\ \sim 10^{-44.85}$ to $\sim 10^{38.53}$
Double Precision	$\pm\ 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$	$\pm\ 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$	$\pm\ \sim 10^{-323.3}$ to $\sim 10^{308.3}$

Range of Floating-Point Numbers

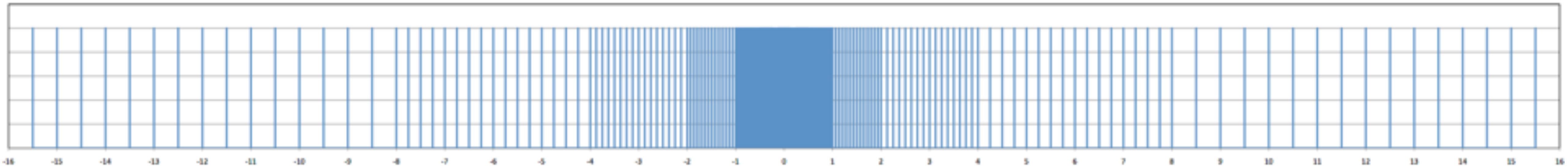
Remember that!

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm\ 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$	$\pm\ 2^{-126}$ to $(2-2^{-23})\times 2^{127}$	$\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$
Double Precision	$\pm\ 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$	$\pm\ 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$	$\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$

	Binary	Decimal
Single Precision	$\pm\ (2-2^{-23}) \times 2^{127}$	$\sim \pm\ 10^{38.53}$
Double Precision	$\pm\ (2-2^{-52}) \times 2^{1023}$	$\sim \pm\ 10^{308.25}$

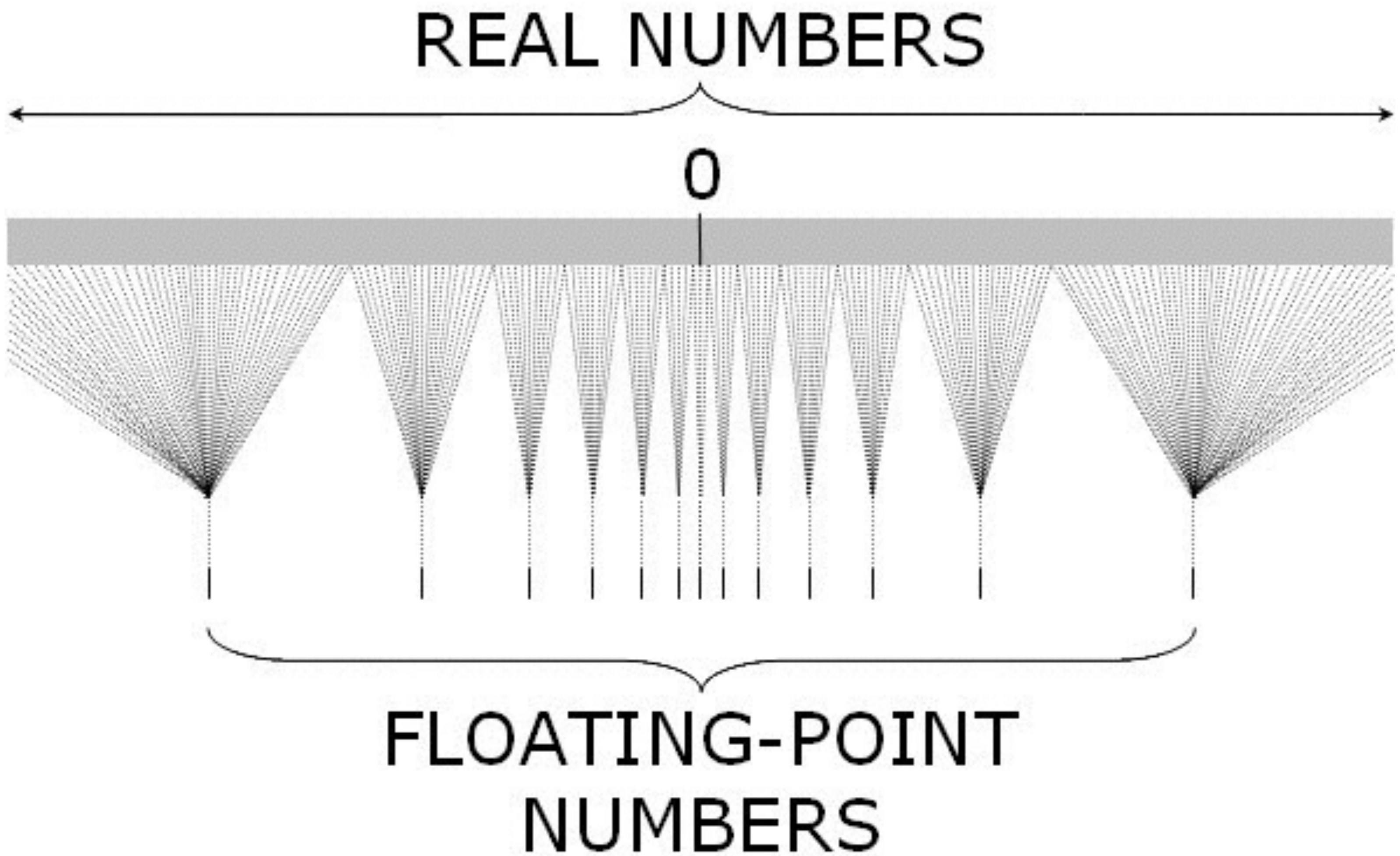
Resolution
of a Floating-Point Format

Check out table here: http://tinyurl.com/FPResol

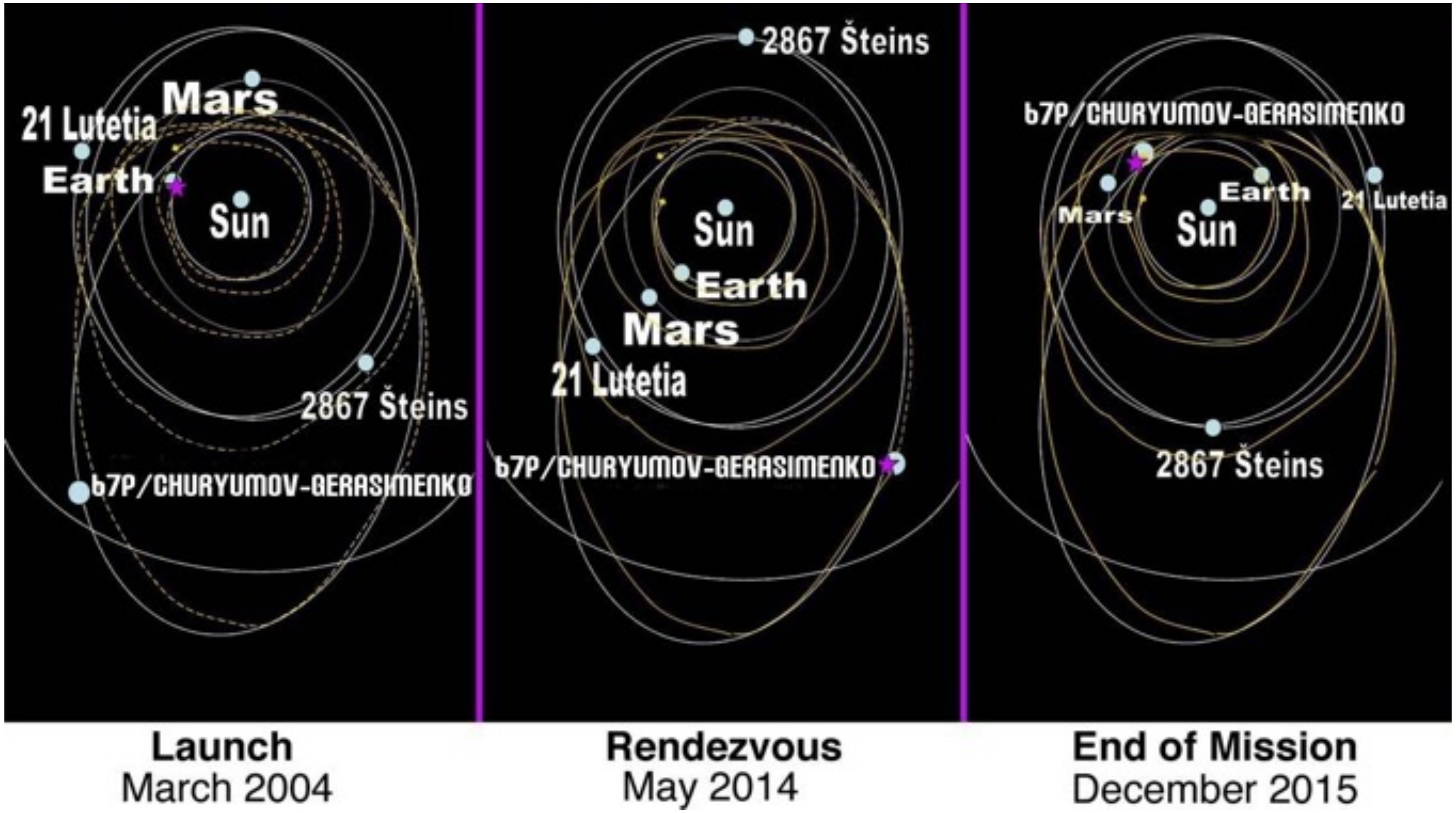


Resolution

Another way to look at it



http://jasss.soc.surrey.ac.uk/9/4/4.html



- Rosetta Landing on Comet
- 10-year trajectory

Why not using *2's Complement* for the Exponent?

0.00000005 = 0 **01100110** 10101101011111110010101

1 = 0 **01111111** 00000000000000000000000

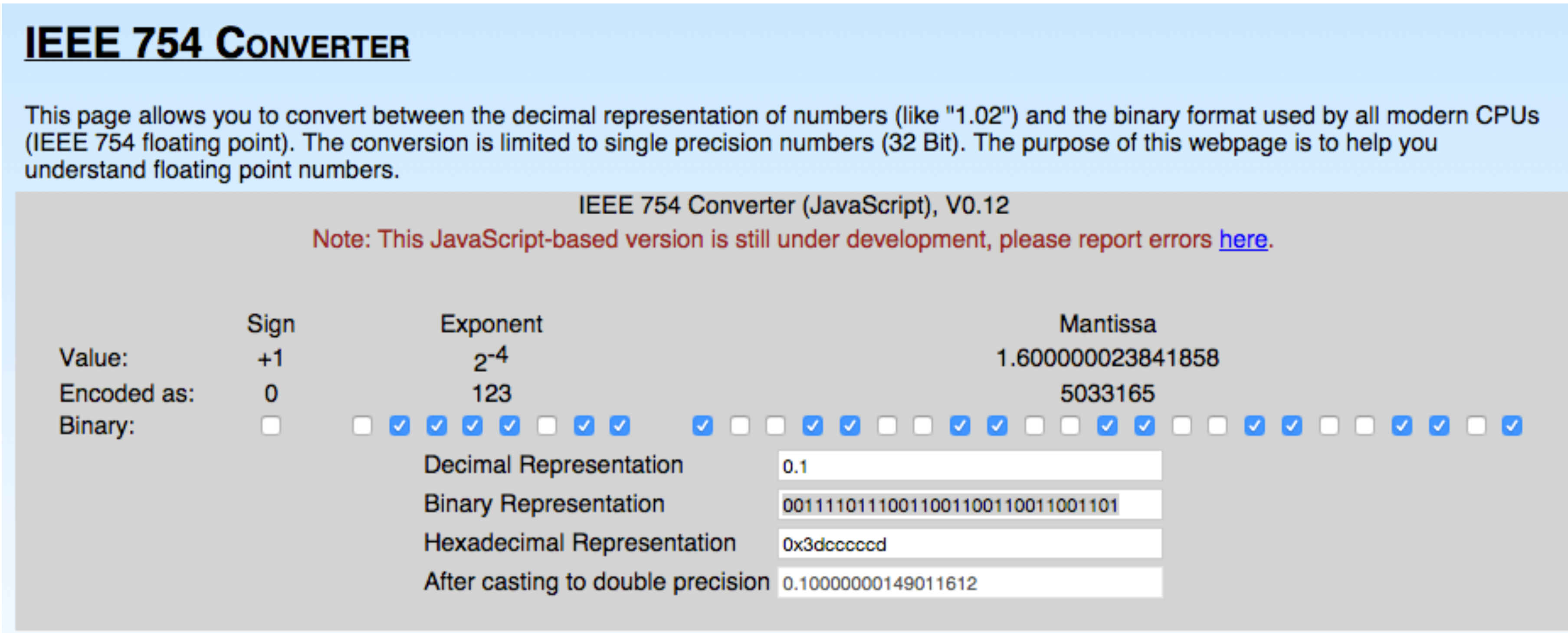
65536.5 = 0 **10001111** 00000000000000001000000

65536.25 = 0 **10001111** 00000000000000000100000

END OF THE SEMESTER!

Exercises

http://www.h-schmidt.net/FloatConverter/IEEE754.html



- Does this converter support NaN, and ∞?
- Are there several different representations of +∞?
- What is the largest float representable with the 32-bit format?
- What is the smallest normalized float (i.e. a float which has an implied leading 1. bit)?

How do we

add 2 FP numbers?

- fp1 = s1 m1 e1
  fp2 = s2 m2 e2
  fp1 + fp2 = ?

- **denormalize** both numbers (restore hidden 1)

- assume fp1 has largest exponent e1: make e2 **equal** to e1 and **shift decimal point** in m2 —> m2'

- compute **sum** m1 + m2'

- **truncate** & **round** result

- **renormalize** result (after checking for special cases)

$1.111 \times 2^5 + 1.110 \times 2^8$

$1.111 \times 2^5 + 1.110 \times 2^8$ after expansion

$$\begin{array}{rl} & 1.11000000 \times 2^8 \\ + & 0.00111100 \times 2^8 \\ \hline & 1.11111100 \times 2^8 \end{array}$$

locate largest number
shift mantissa of smaller

compute sum

$1.111|11100 \times 2^8$ round & truncate

$= 10.000 \times 2^8$ normalize

$= 1.000 \times 2^9$

How do we

multiply 2 FP numbers?

- fp1 = s1 m1 e1
  fp2 = s2 m2 e2
  fp1 x fp2 = ?

- Test for multiplication by special numbers (0, NaN, ∞)

- **denormalize** both numbers (restore hidden 1)

- compute product of m1 x m2

- compute **sum** e1 + e2

- **truncate** & **round** m1 x m2

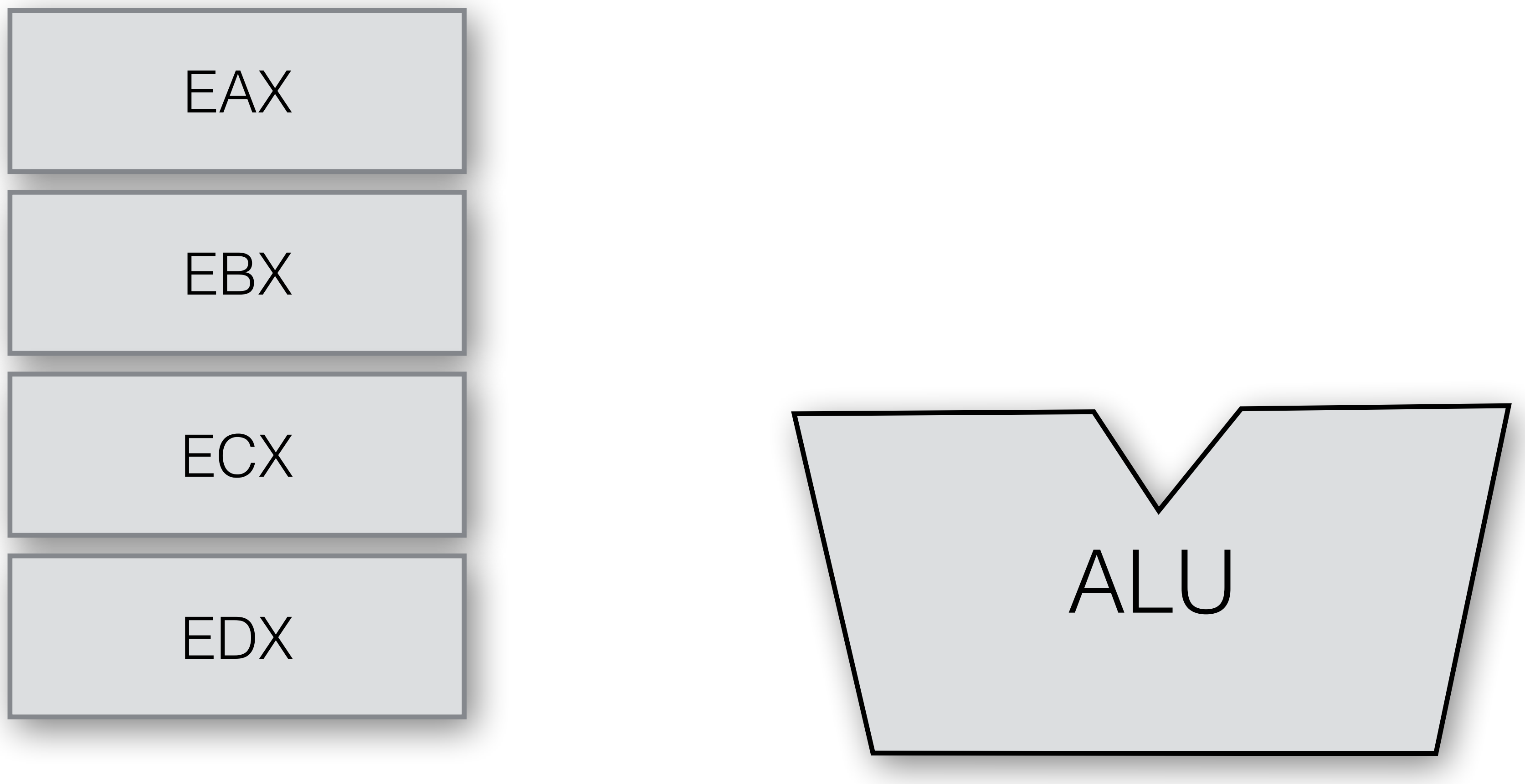
- **adjust** e1+e2 and **normalize**.

How do we **compare** two FP numbers?

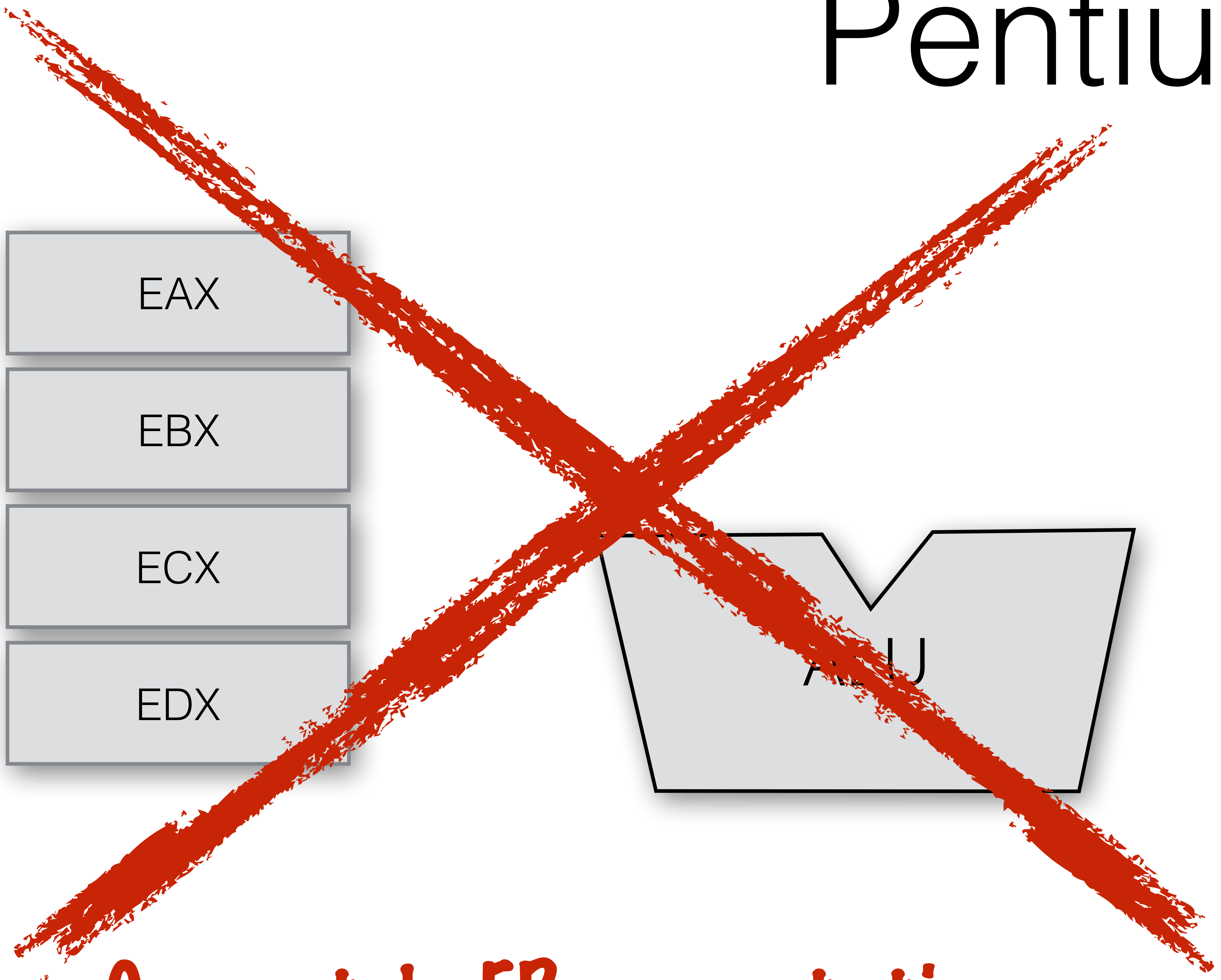
As unsigned integers!
No unpacking necessary!

Programming FP Operations in Assembly…

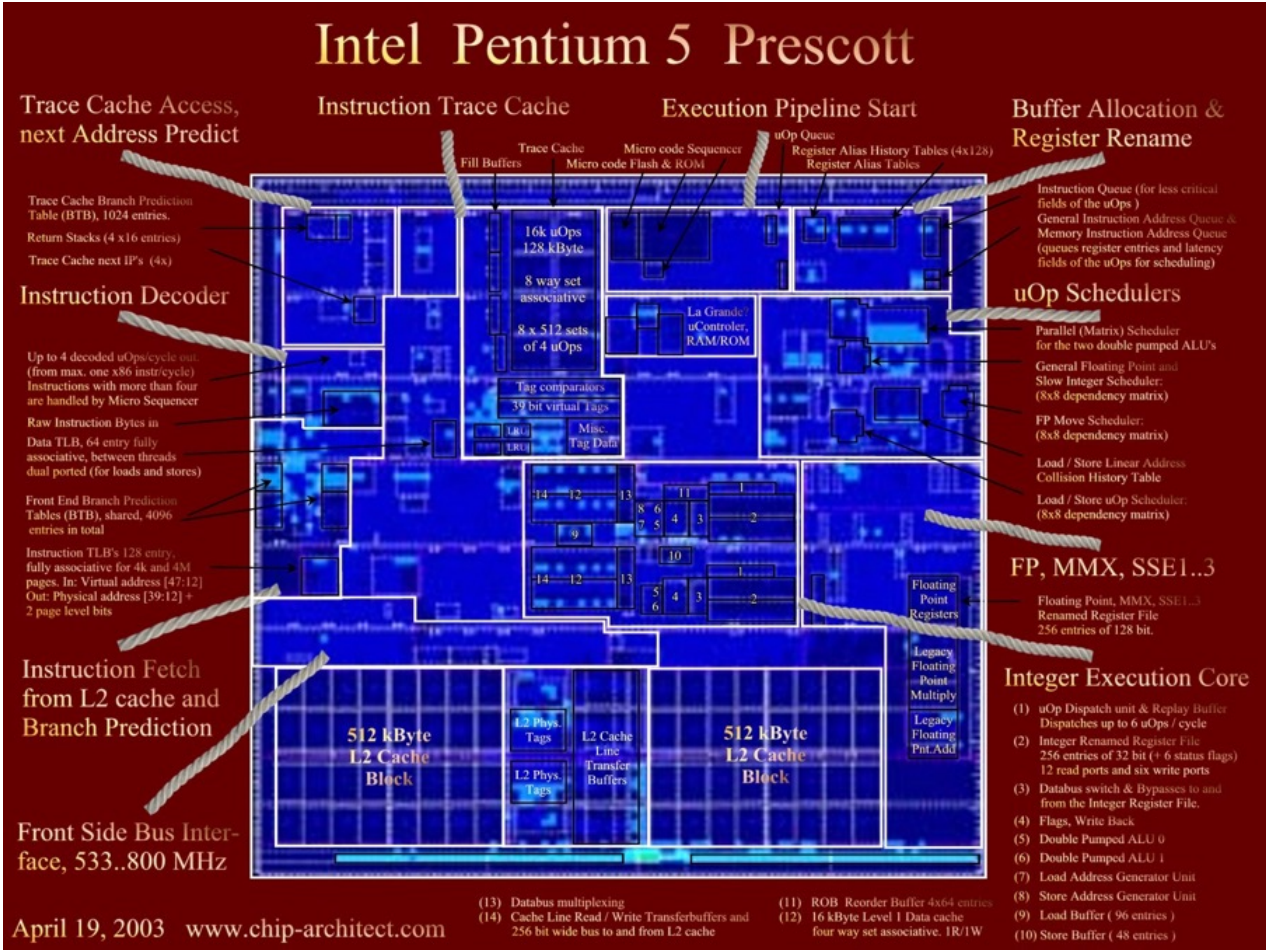
Pentium



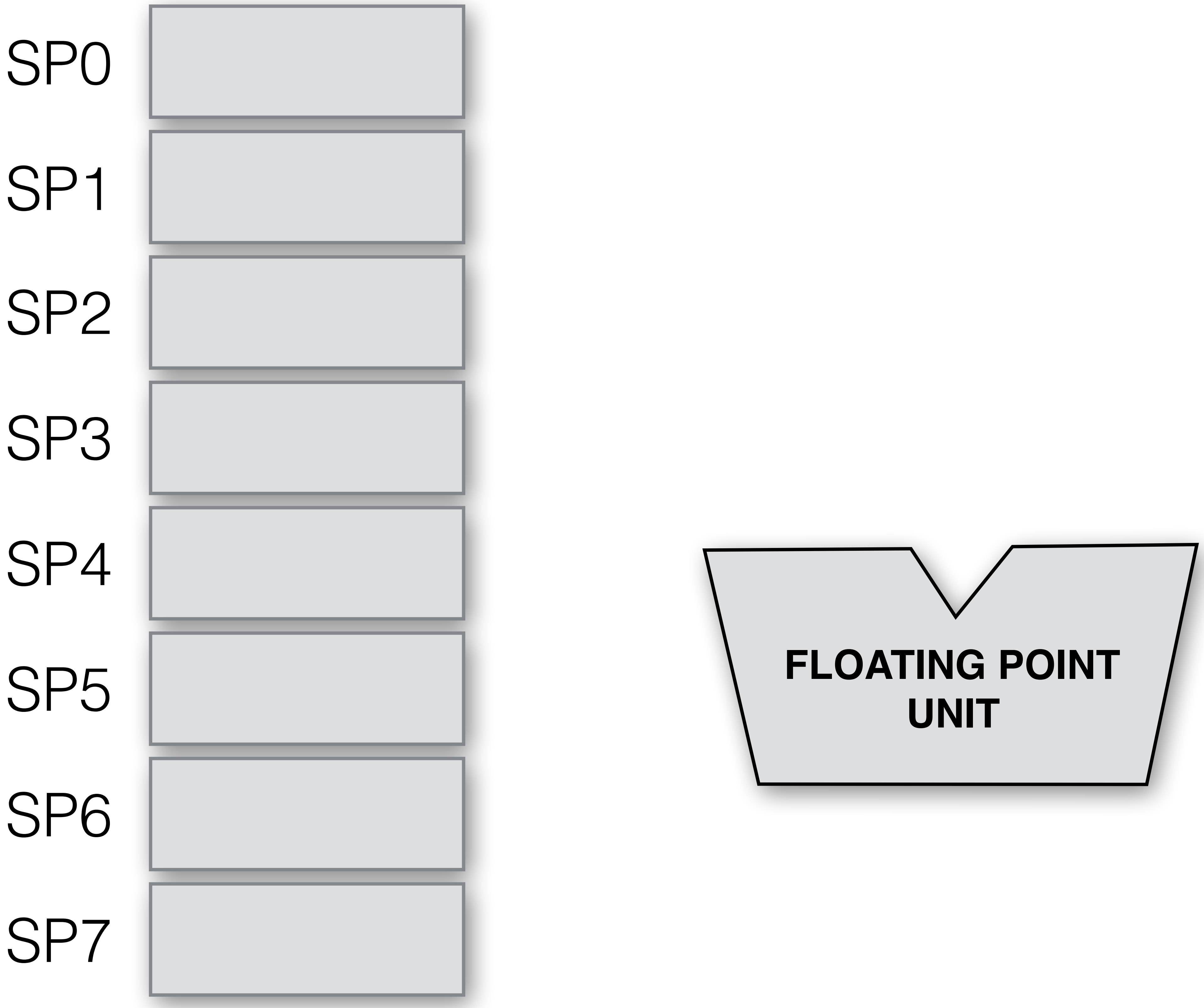
Pentium



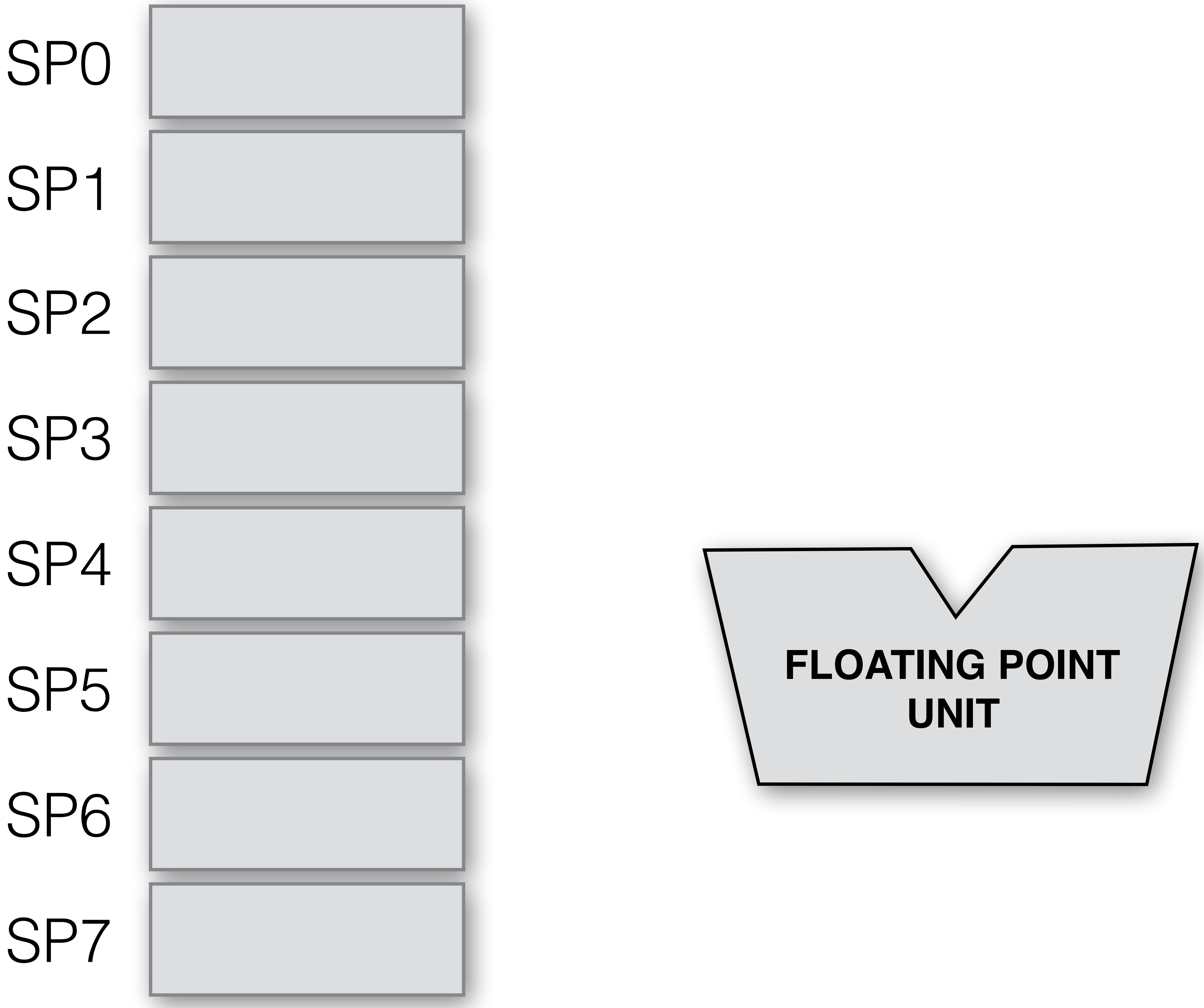
Cannot do FP computation



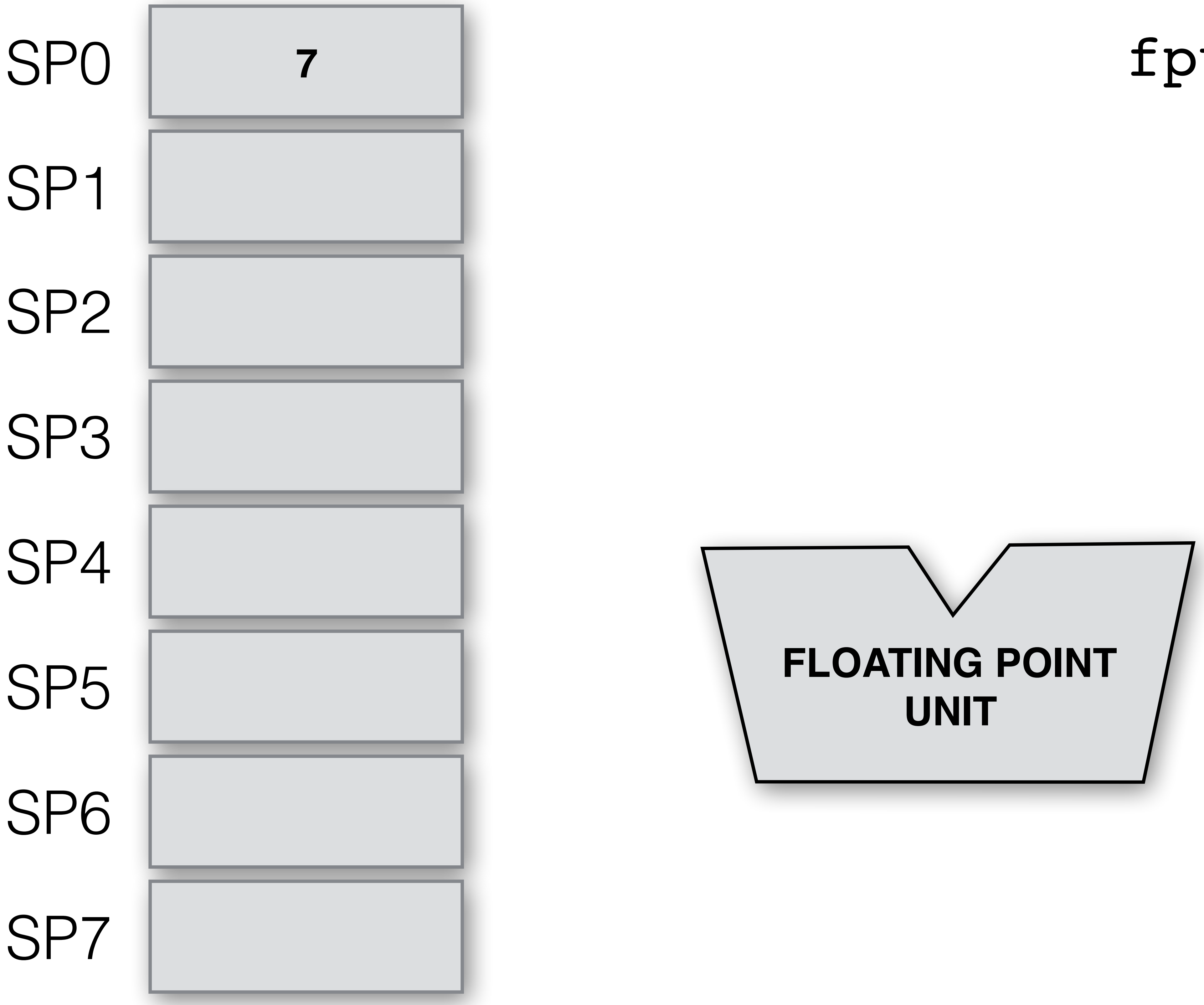
http://chip-architect.com/news/2003_04_20_looking_at_intels_prescott_part2.html


SP0
SP1
SP2
SP3
SP4
SP5
SP6
SP7
FLOATING POINT
UNIT

`Operation: (7+10)/9`

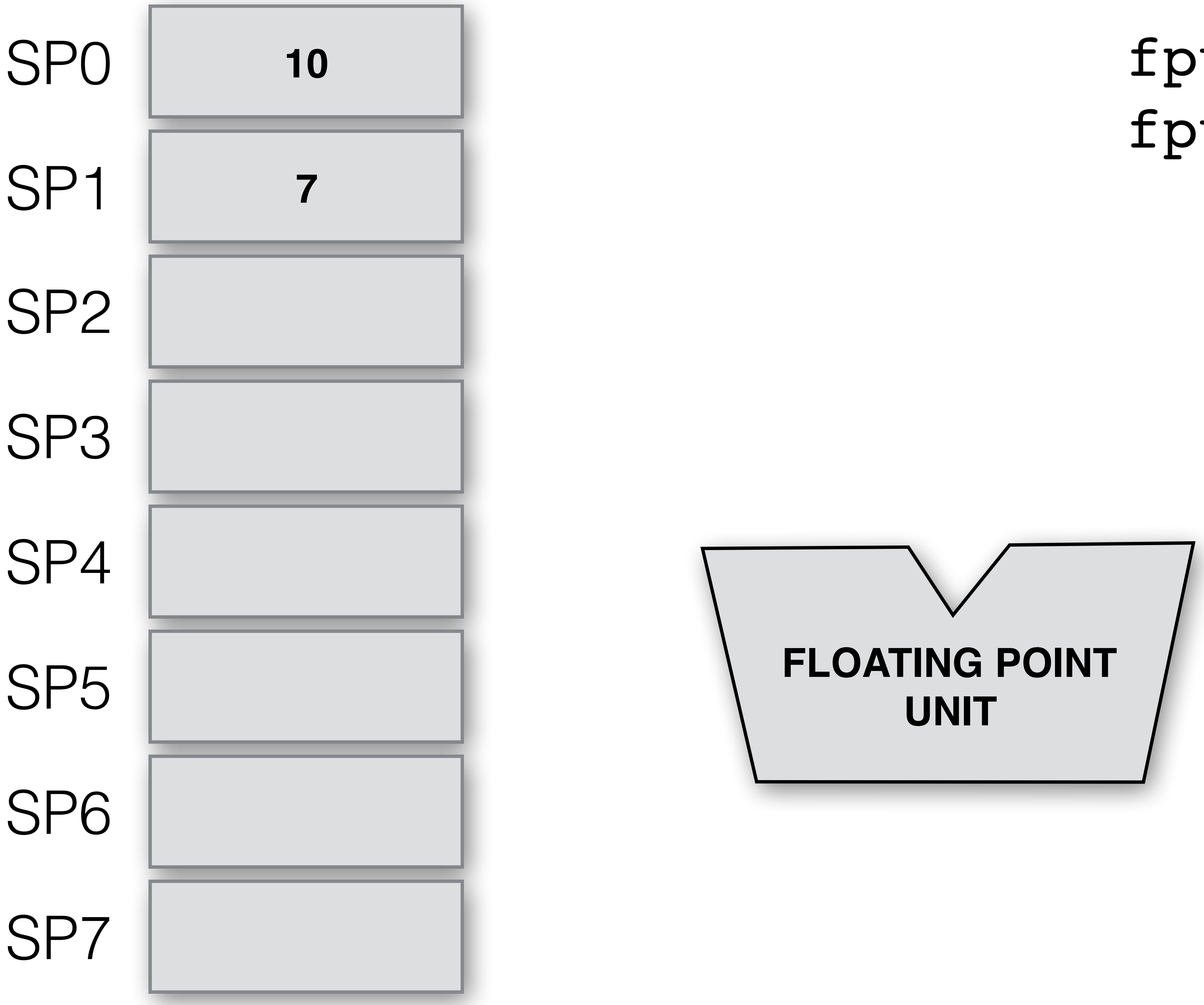


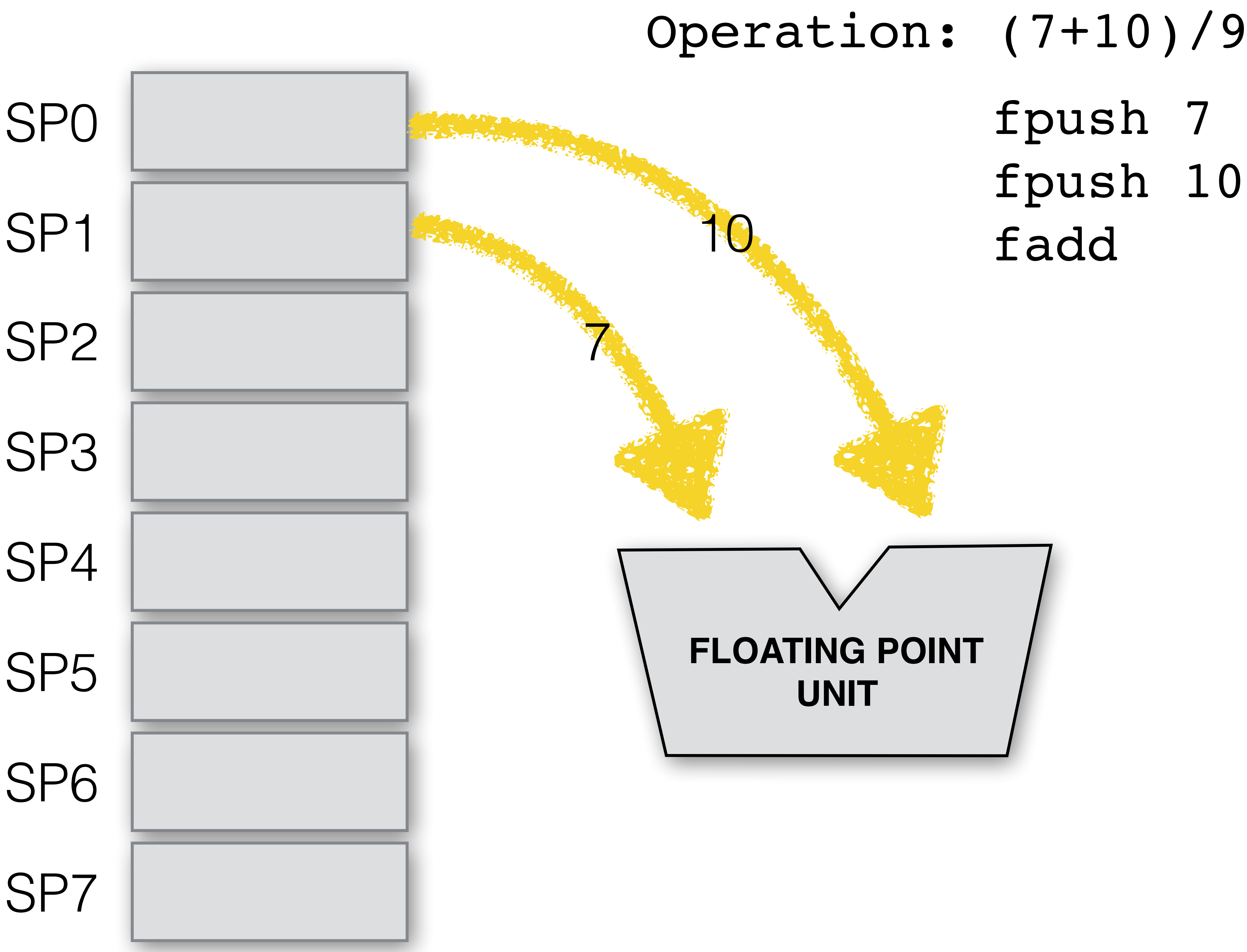
```
Operation: (7+10)/9

fpush 7
```



Operation: (7+10)/9

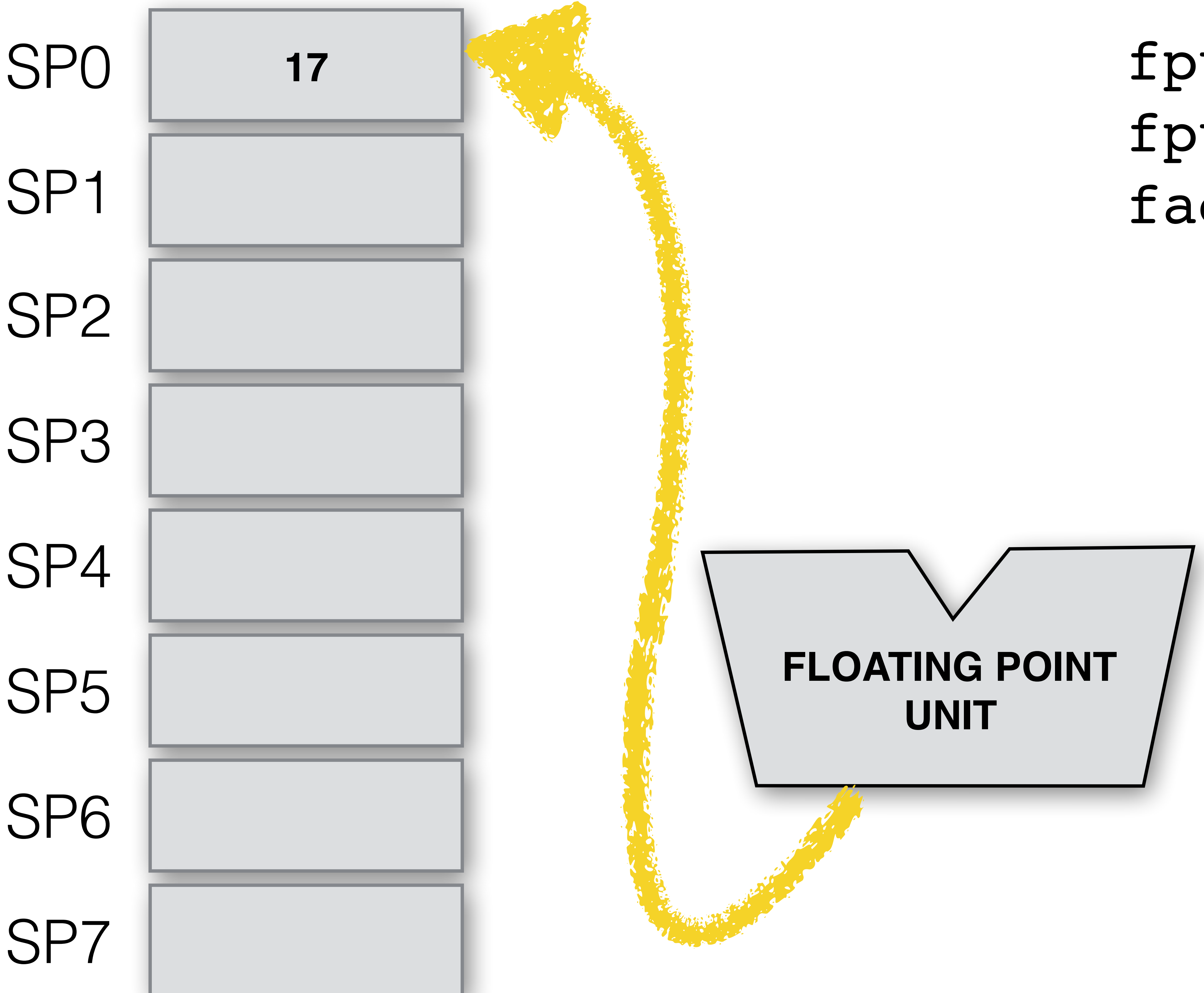
```
fpush 7
fpush 10
```




Operation: (7+10)/9
SP0
SP1
SP2
SP3
SP4
SP5
SP6
SP7
10
7
fpush 7
fpush 10
fadd
FLOATING POINT UNIT

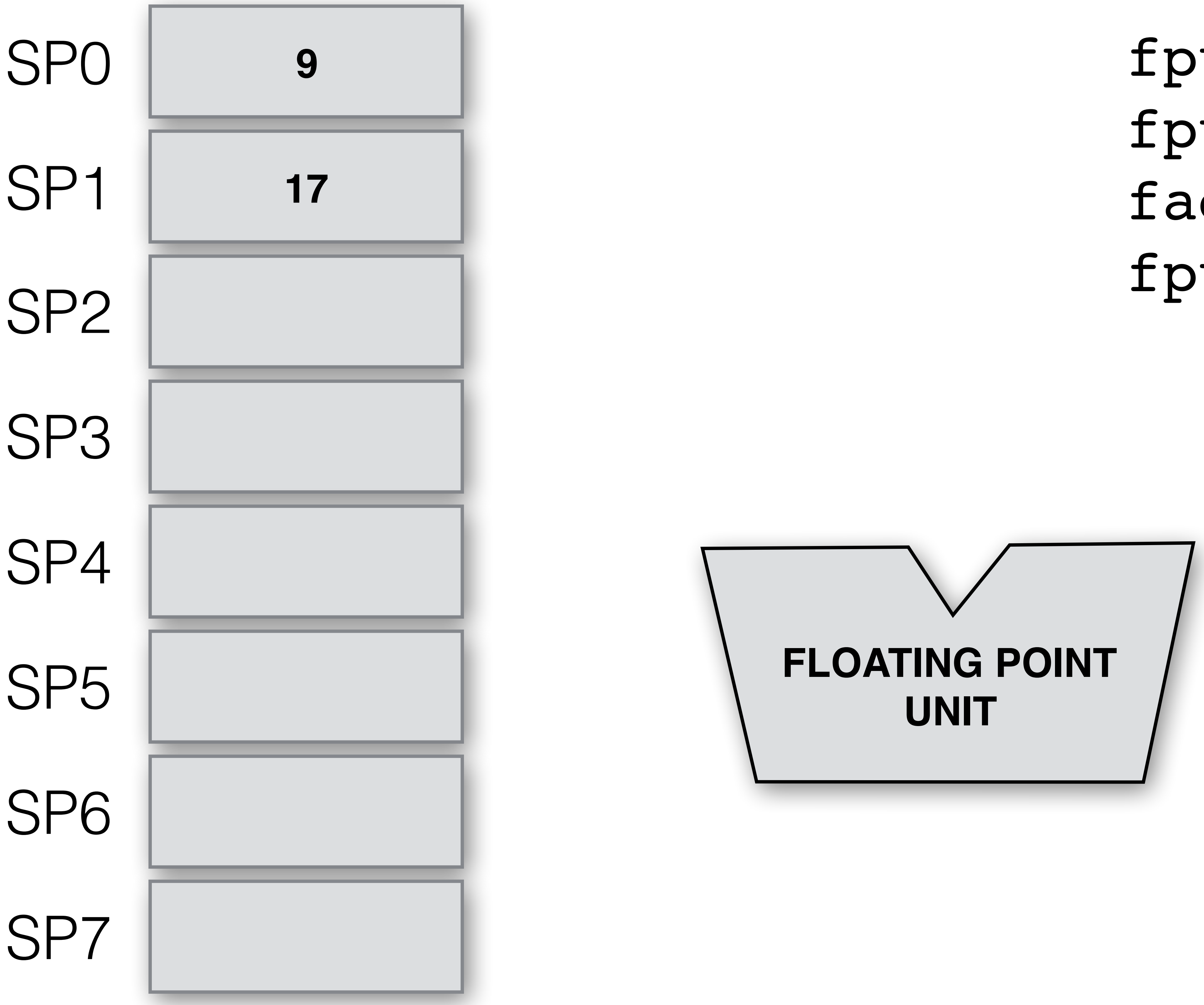
Operation: (7+10)/9

```
fpush 7
fpush 10
fadd
```

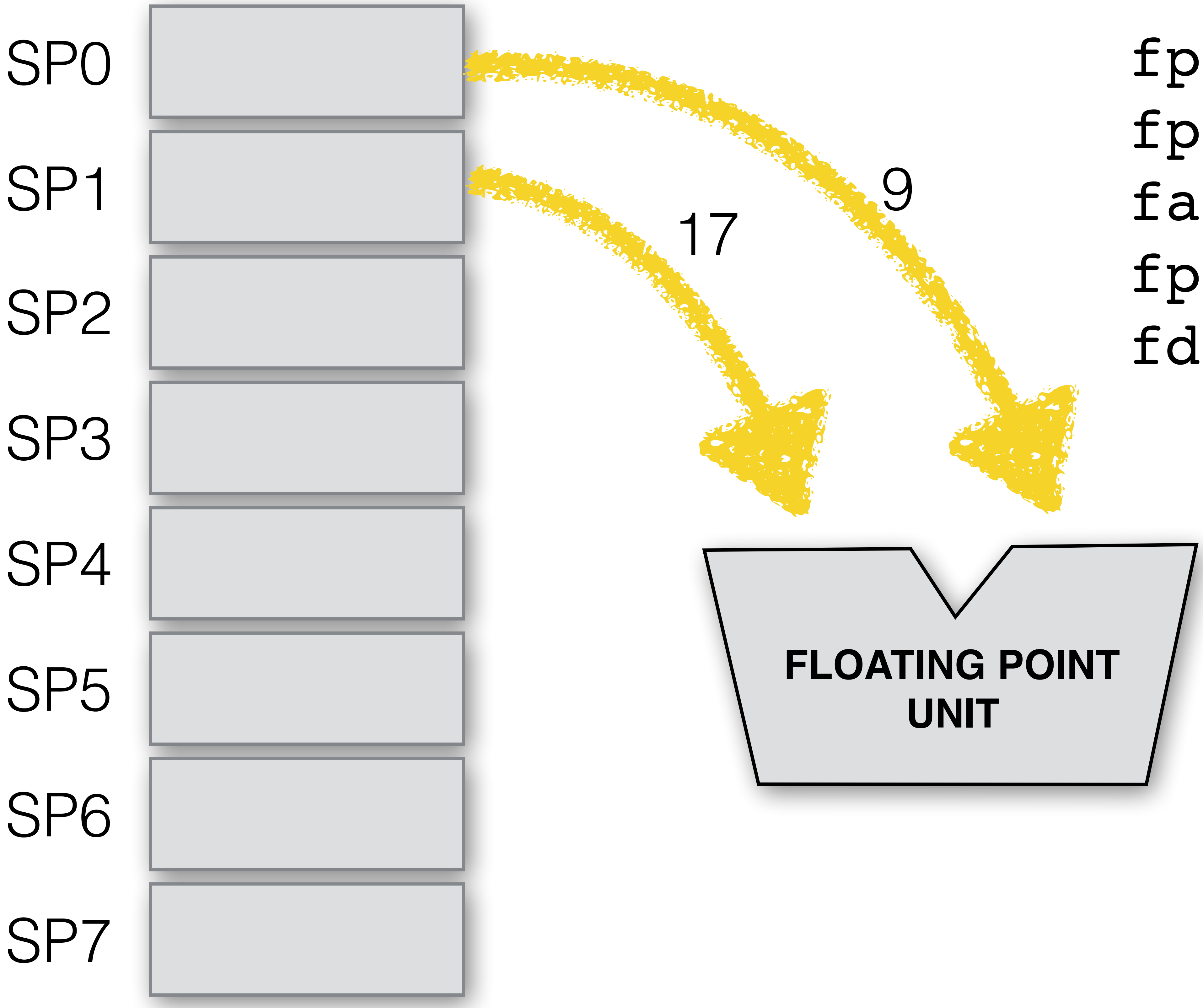


Operation: (7+10)/9

```
fpush 7
fpush 10
fadd
fpush 9
```

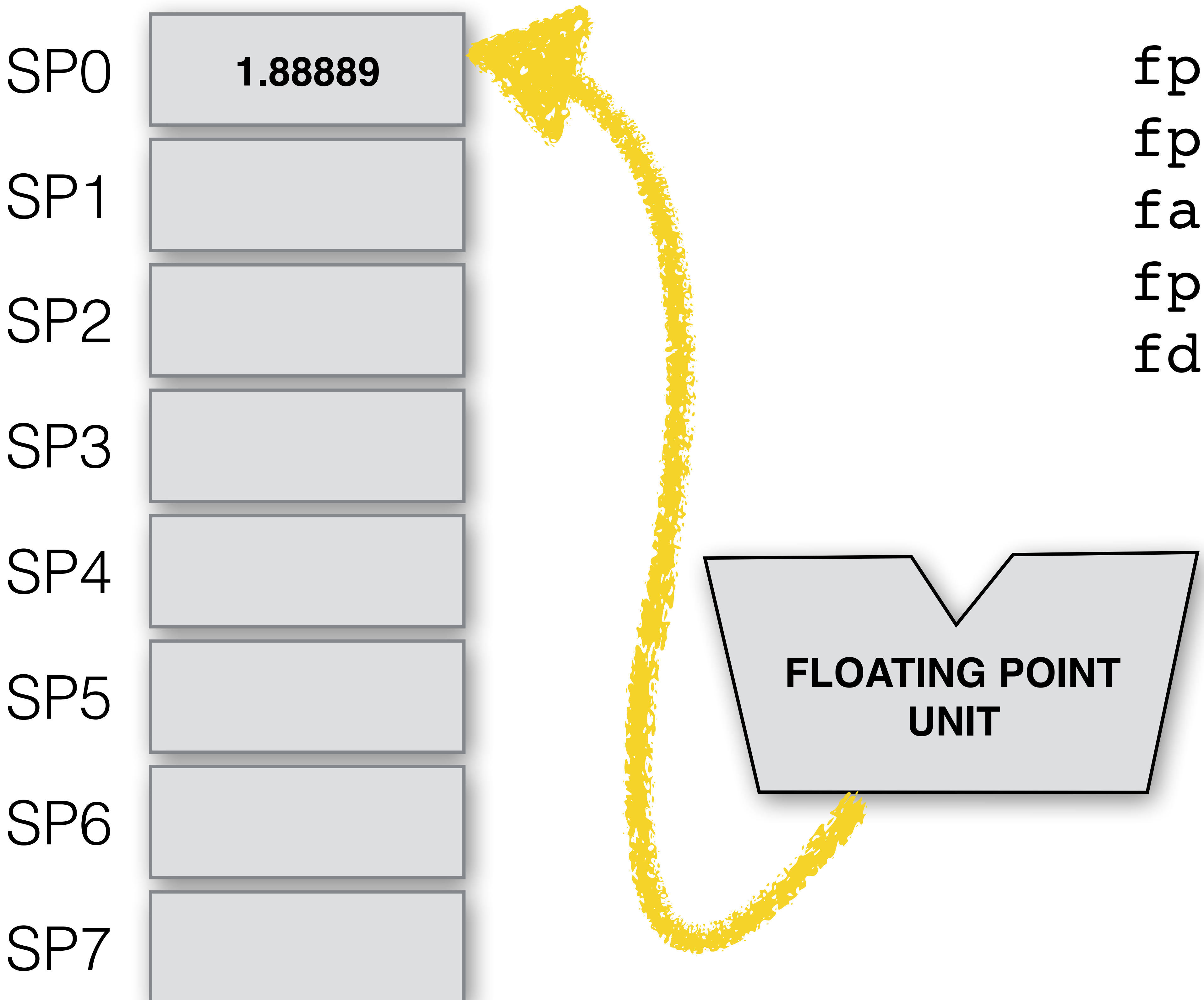


```
Operation: (7+10)/9

fpush 7
fpush 10
fadd
fpush 9
fdiv
```



Operation: (7+10)/9

```
fpush 7
fpush 10
fadd
fpush 9
fdiv
```



The Pentium computes
FP expressions
using RPN!

The Pentium computes FP expressions using RPN!

Reverse Polish Notation

Nasm Example: z = x+y

```
         SECTION .data

x        dd       1.5
y        dd       2.5
z        dd       0

; compute z = x + y
         SECTION .text


         fld      dword [x]
         fld      dword [y]
         fadd
         fstp     dword [z]
```

Printing floats in C

```
#include "stdio.h"

int main() {
   float z = 1.2345e10;
   printf( "z = %e\n\n", z );
   return 0;
}
```

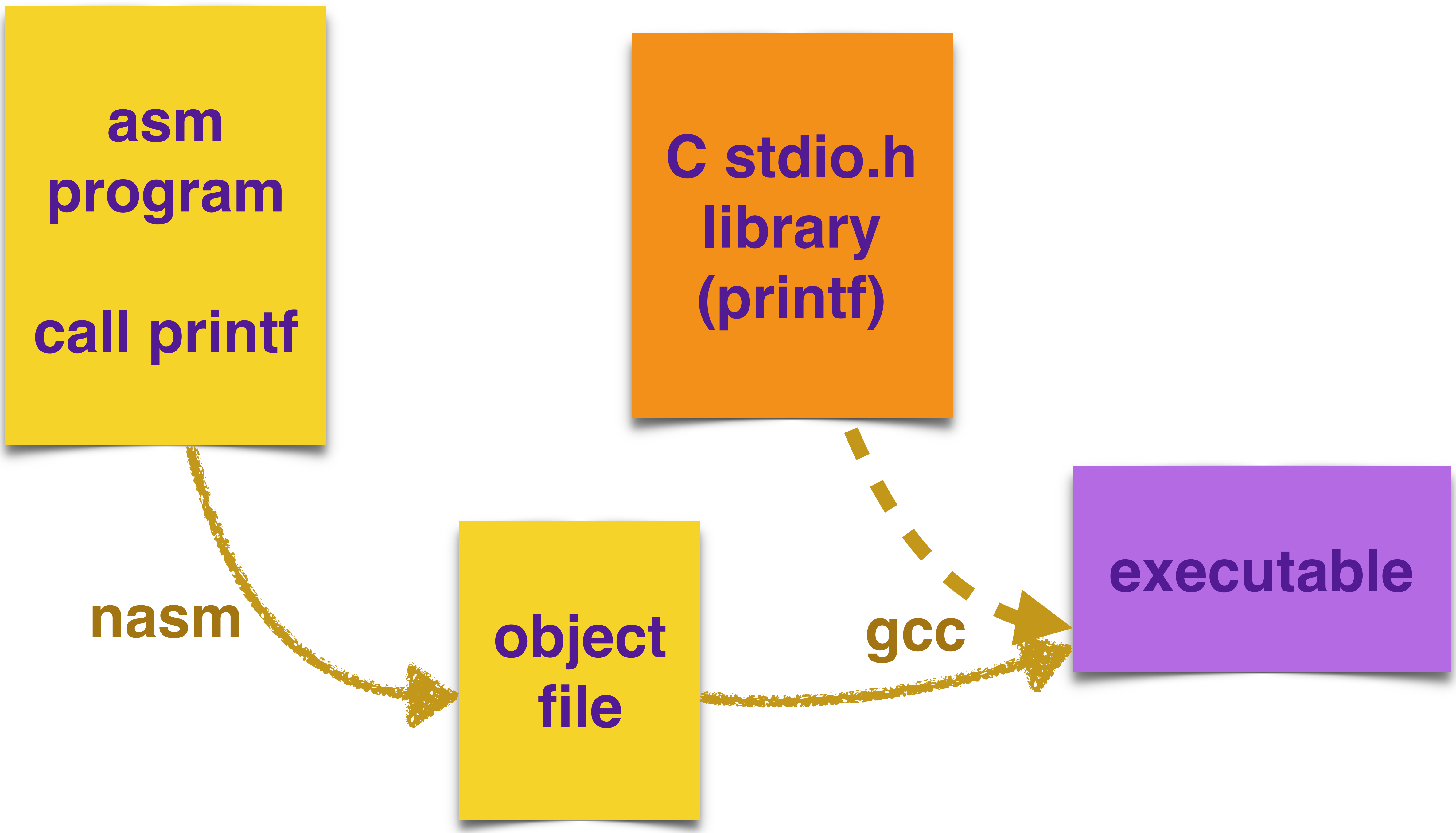
Printing floats in C

```
#include "stdio.h"

int main() {
   float z = 1.2345e10;
   printf( "z = %e\n\n", z );
   return 0;
}
```

works only on Linux with 32-bit Libraries

```
gcc -m32 -o printFloat printFloat.c
./printFloat
z = 1.234500e+10
```

Printing floats in Assembly?



```
        extern printf                           ; the C function to be called

        SECTION .data                           ; Data section

msg     db      "sum = %e",0x0a,0x00
x       dd      1.5
y       dd      2.5
z       dd      0
temp    dq      0

        SECTION .text                           ; Code section.
        global  main                            ; "C" main program
main:                                           ; label, start of main program
        fld     dword [x]                       ; need to convert 32-bit to 64-bit
        fld     dword [y]
        fadd
        fstp    dword [z]                       ; store sum in z

        fld     dword [z]                       ; transform z to 64-bit by pushing in stack
        fstp    qword [temp]                    ; and popping it back as 64-bit quadword

        push    dword [temp+4]                  ; push temp as 2 32-bit words
        push    dword [temp]
        push    dword msg                       ; address of format string
        call    printf                          ; Call C function
        add     esp, 12                         ; pop stack 3*4 bytes

        mov     eax, 1                          ; exit code, 0=normal
        mov     ebx, 0
        int     0x80                            ;
```

```
dthiebaut@hadoop:~/temp$ nasm -f elf addFloats.asm
dthiebaut@hadoop:~/temp$ gcc -m32 -o addFloats addFloats.o
dthiebaut@hadoop:~/temp$ ./addFloats
sum = 4.000000e+00
dthiebaut@hadoop:~/temp$
```

More code examples here:

http://cs.smith.edu/dftwiki/index.php/CSC231_An_Introduction_to_Fixed-_and_Floating-Point_Numbers#Assembly_Language_Programs