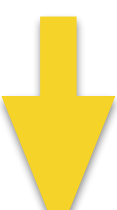
Fixed & Floating Point Formats

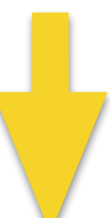
CSC231—Fall 2017
Week #15

Dominique Thiébaut
dthiebaut@smith.edu

Trick Question

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



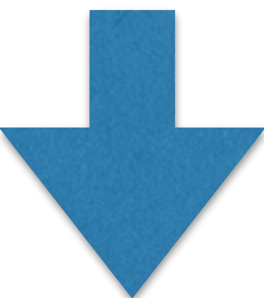


We stopped here last time…

Normalization (in decimal)

(normal = standard form)

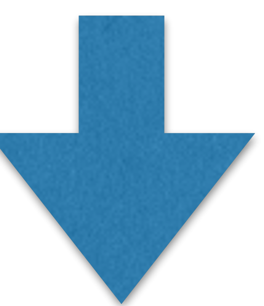
y = 123.456



$y = 1.23456 \times 10^2$

Normalization (in binary)

y = 1000.100111   (8.609375d)



$y = 1.000100111 \times 2^3$

Normalization (in binary)

$y = 1000.100111$

decimal

$y = 1.000100111 \times 2^3$

Normalization (in binary)

$y = 1000.100111$

$y = 1.000100111 \times 2^3$ (decimal)

$y = 1.000100111 \times 10^{11}$ (binary)

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

0	1000100111	11
sign	mantissa	exponent

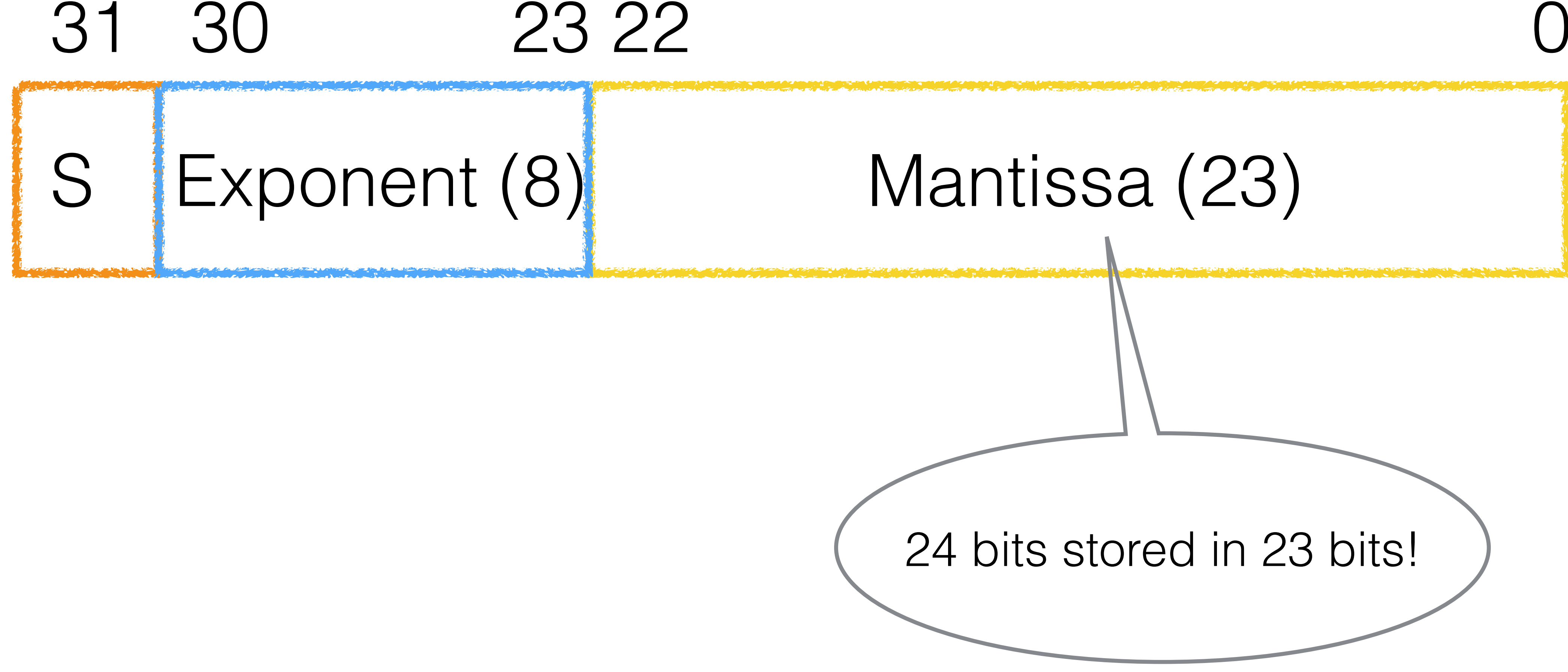
**But, remember,
all* numbers have
a leading 1, so, we can pack
the bits even more
efficiently!**

*really?

implied bit!
+1.0001001111 x 10^11

0
0001001110
11
sign
mantissa
exponent

IEEE Format



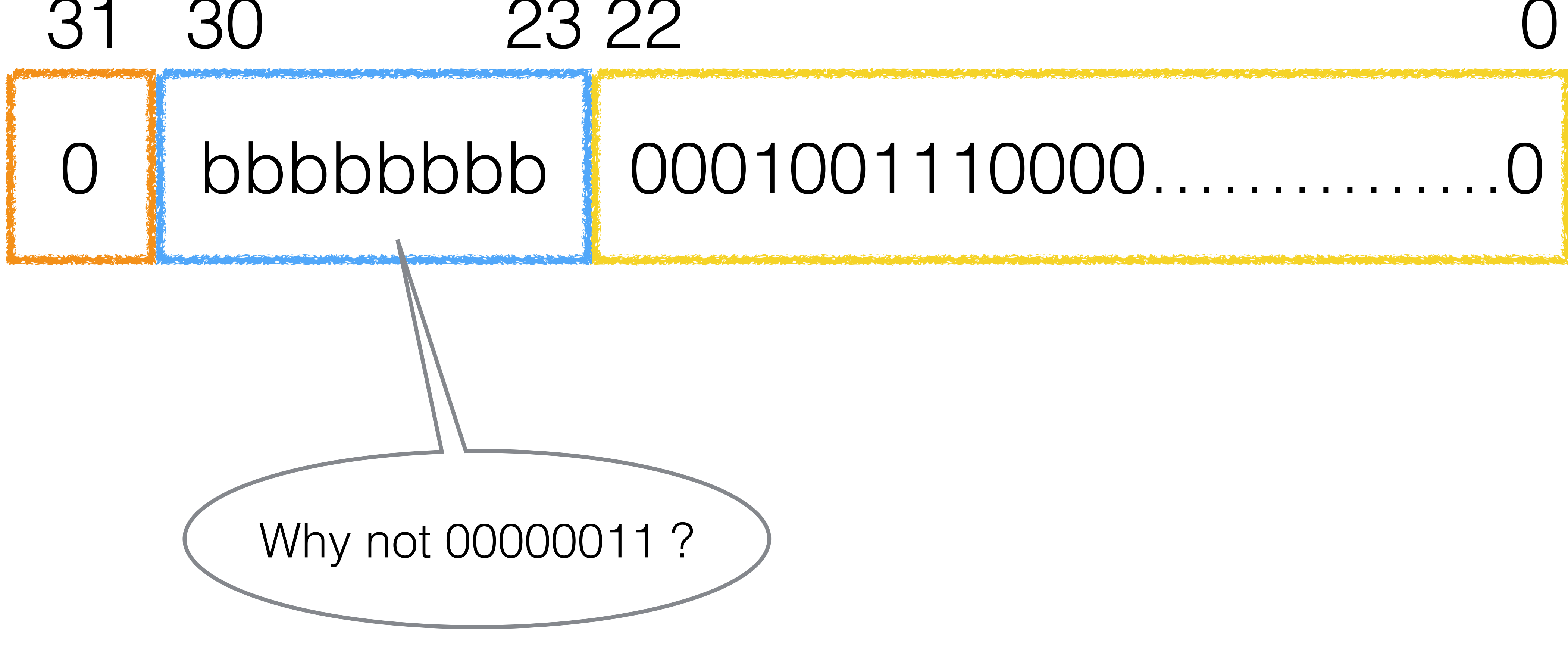
$y = 1000.100111$

$y = 1.000100111 \; 2^{3}$

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

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

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



How is the exponent coded?

bbbbbbbb

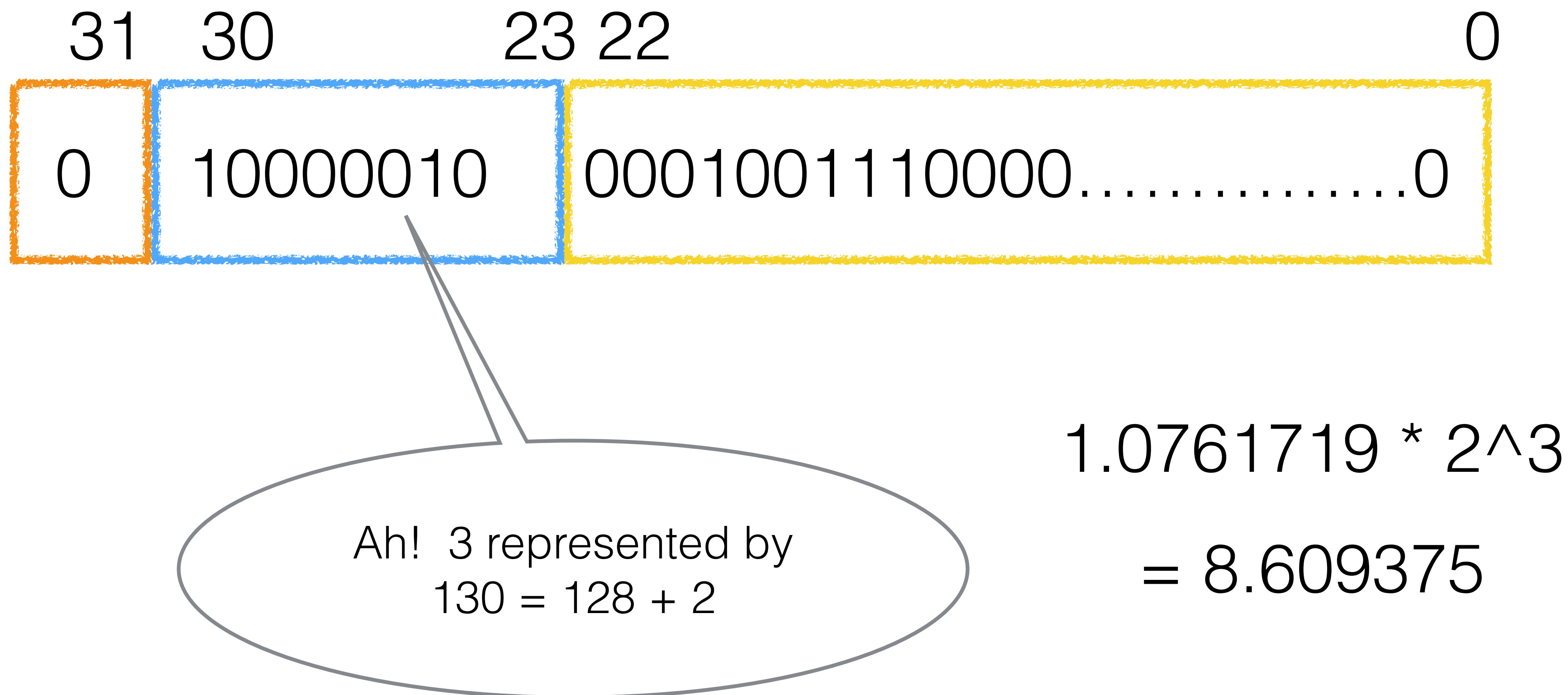
bias of 127

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

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



1.0761719 * 2^3

= 8.609375

Verification
8.6093752 in IEEE FP

Tools & Thoughts

IEEE-754 Floating Point Converter

Translations: de

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).

IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	+1	2^3	1.076171875
Encoded as:	0	130	638976
Binary:	0	10000010	00010011100000000000000

Decimal representation	8.609375
Value actually stored in float:	8.609375
Error due to conversion:	
Binary Representation	01000001000010011100000000000000
Hexadecimal Representation	0x4109c000

+1

-1

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

- 1.0 = 0 | 01111111 | 000…0
- 0.5 = 0 | 01111110 | 000…0
- 1.5 = 1.0 + 0.5 = 0 | 01111111 | 100…0
- -1.5 = 1 | 01111111 | 100…0
- 1 | 1000 0011 | 111 1000 0000 0000 0000 0000

  = - (stored exp = 131 —> real exp = 4)
  + 1.9375
  = - 1.9375 x 2^4 = -1.9375 x 16 = -31

what about 0.1?

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

0   01111011   10011001100110011001101

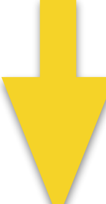
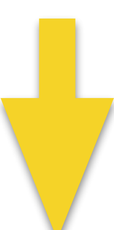
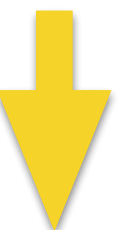
0.1 decimal, in 32-bit precision, IEEE Format:

0   01111011   10011001100110011001101

Value in double-precision: **0.10000000149011612**

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

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

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

0 | 00000000 | 001000…000

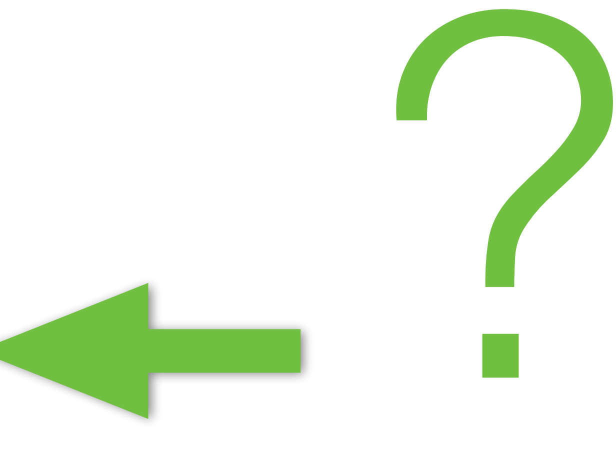
+ (2^{-126}) x (0.001)

binary

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

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 exponents

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

Very large exponents

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

Very large exponents

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

Very large exponents

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

Very large exponents

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

- 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 $+1$

```
// 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
```

NaN is *sticky*!

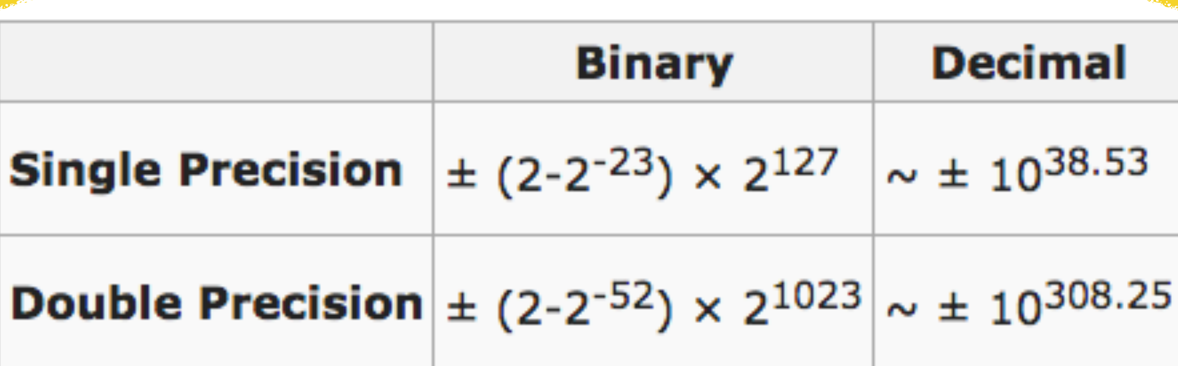
Range of Floating-Point Numbers

	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!

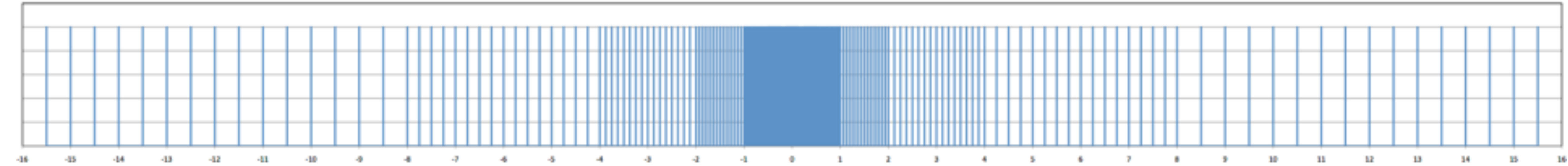
	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm\ 2^{-149}$ to $(1-2^{-23})\times2^{-126}$	$\pm\ 2^{-126}$ to $(2-2^{-23})\times2^{127}$	$\pm\ \sim10^{-44.85}$ to $\sim10^{38.53}$
Double Precision	$\pm\ 2^{-1074}$ to $(1-2^{-52})\times2^{-1022}$	$\pm\ 2^{-1022}$ to $(2-2^{-52})\times2^{1023}$	$\pm\ \sim10^{-323.3}$ to $\sim10^{308.3}$

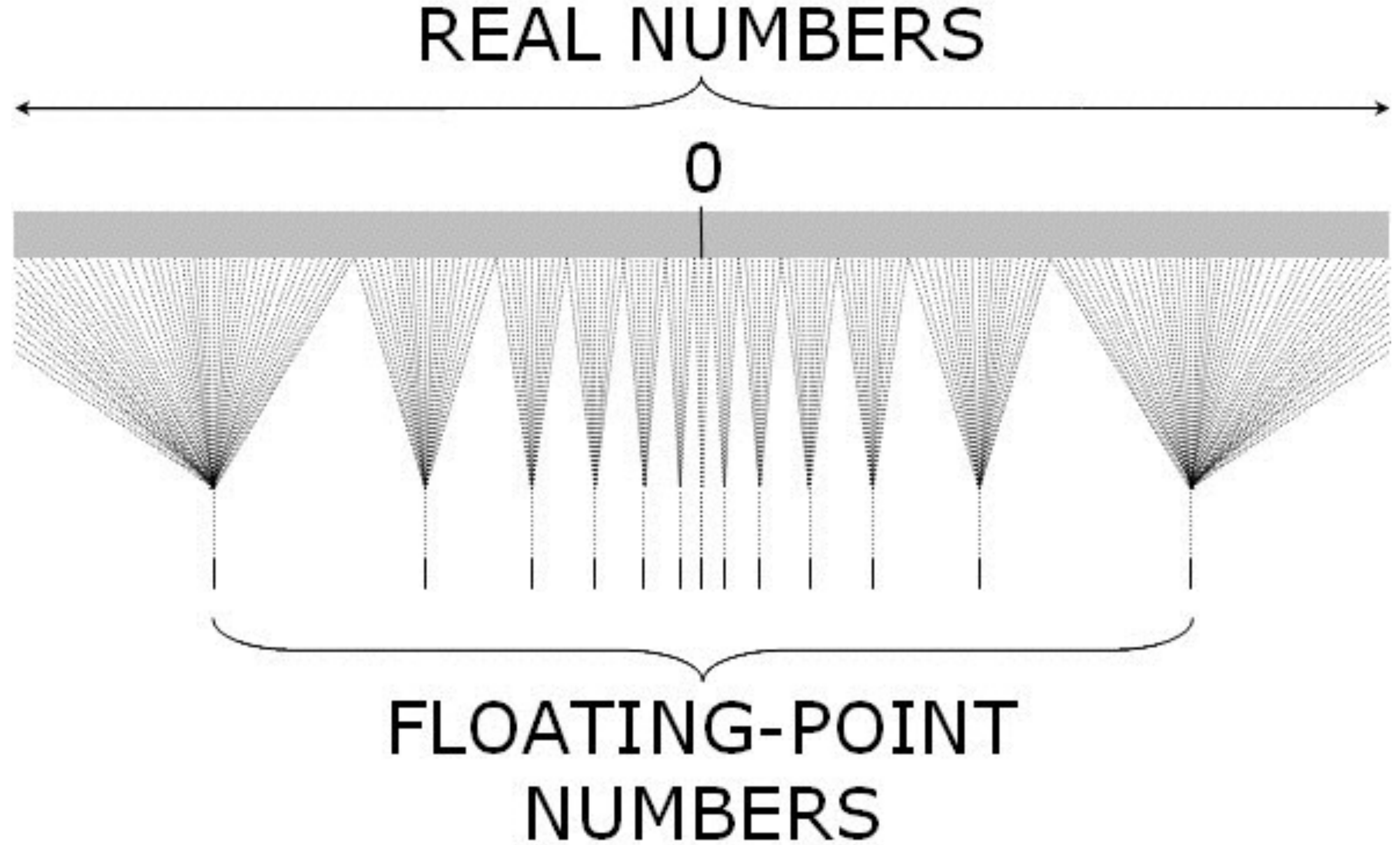


	Binary	Decimal
Single Precision	$\pm\ (2-2^{-23})\times2^{127}$	$\sim\pm\ 10^{38.53}$
Double Precision	$\pm\ (2-2^{-52})\times2^{1023}$	$\sim\pm\ 10^{308.25}$

Resolution of a Floating-Point Format

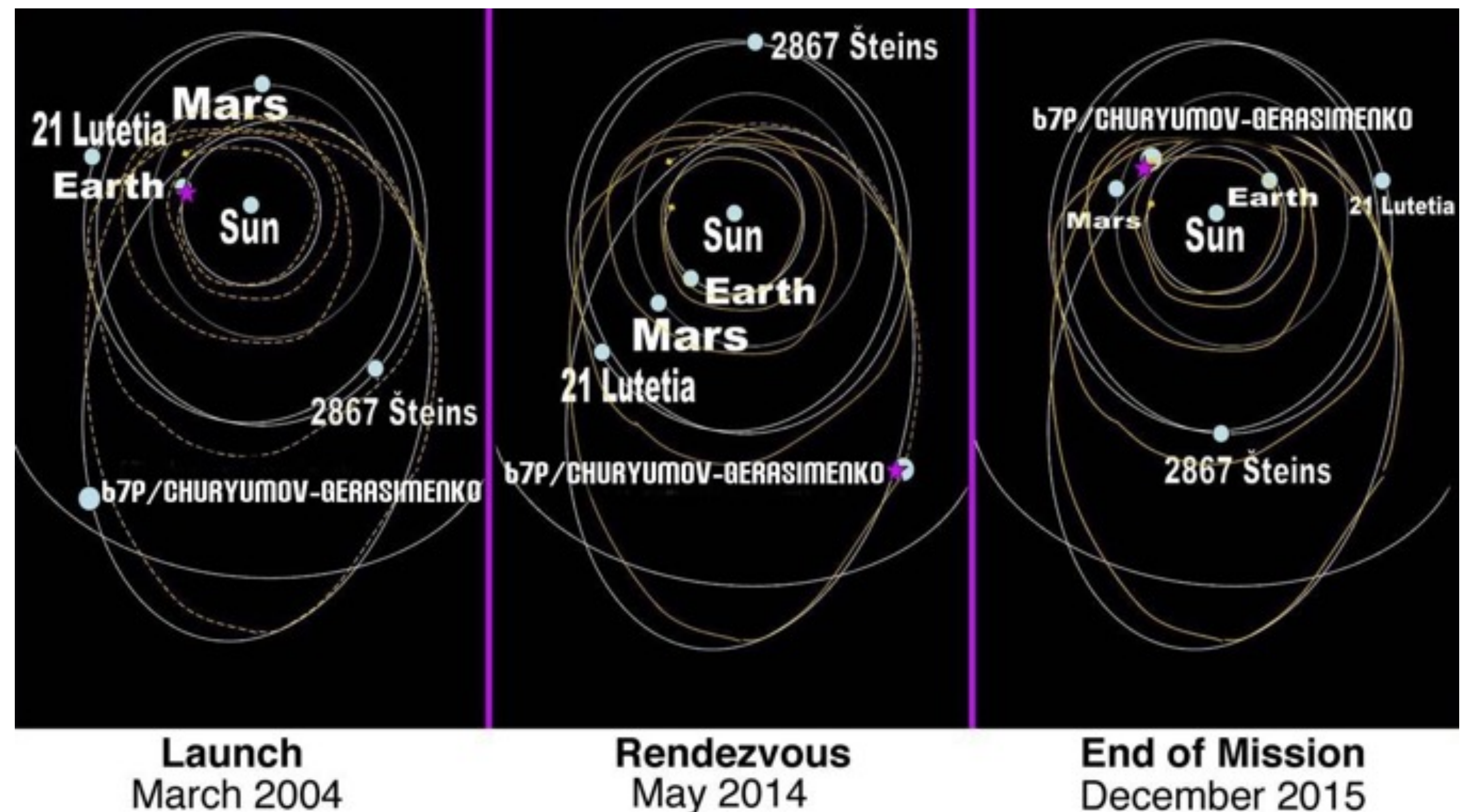
Check out table here: http://tinyurl.com/FPResol





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

What does it have to do with Art?



- Rosetta Landing on Comet
- 10-year trajectory

We stopped here
last time…

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

0.00000005 = 0 **01100110** 10101101011111110010101

1 = 0 **01111111** 00000000000000000000000

65536.25 = 0 **10001111** 00000000000000000100000

65536.5 = 0 **10001111** 00000000000000001000000

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

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

- 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?

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

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

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

locate largest number
shift mantissa of smaller

compute sum

$1.111 \vdots 11100 \times 2^8$ — round & truncate

$= 10.000 \times 2^8$ — normalize

$= 1.000 \times 2^9$

- 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)

What are the "dangers" of adding 2 FP numbers?

https://www.foodsafetymagazine.com/fsm/cache/file/3B40D087-FDF9-43B7-9353CE2E6C9CC945.jpg

Algorithm for adding a Series of FP numbers?

https://www.foodsafetymagazine.com/fsm/cache/file/3B40D087-FDF9-43B7-9353CE2E6C9CC945.jpg

How do we Multiply 2 FP Numbers?

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

$1.111 \times 2^5 \;\; \times \;\; 1.110 \times 2^8$ — after expansion

$$
\begin{array}{r}
1.110 \\
\times\ 1.111 \\
\hline
11.010010
\end{array}
$$

multiply mantissas

normalize

$= \; 1.110 \,\vdots\, 10010 \times 2^1$ — round & truncate

$= 1.111 \times 2^1$ — normalize

$1 + 5 + 8 = 14$ — add exponents

$= 1.111 \; 2^{14}$

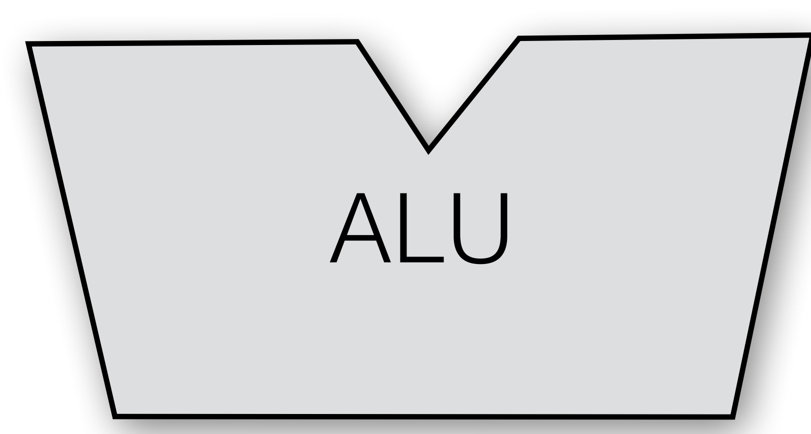
- fp1 = s1 m1 e1
  fp2 = s2 m2 e2
  fp1 x fp2 = ?
- Test for multiplication by special numbers (0, NaN, ∞)
- **denormalize** both numbers (restore hidden 1)
- compute product of m1 x m2
- compute **sum** e1 + e2
- **truncate** & **round** m1 x m2
- **adjust** e1+e2 and **normalize**.

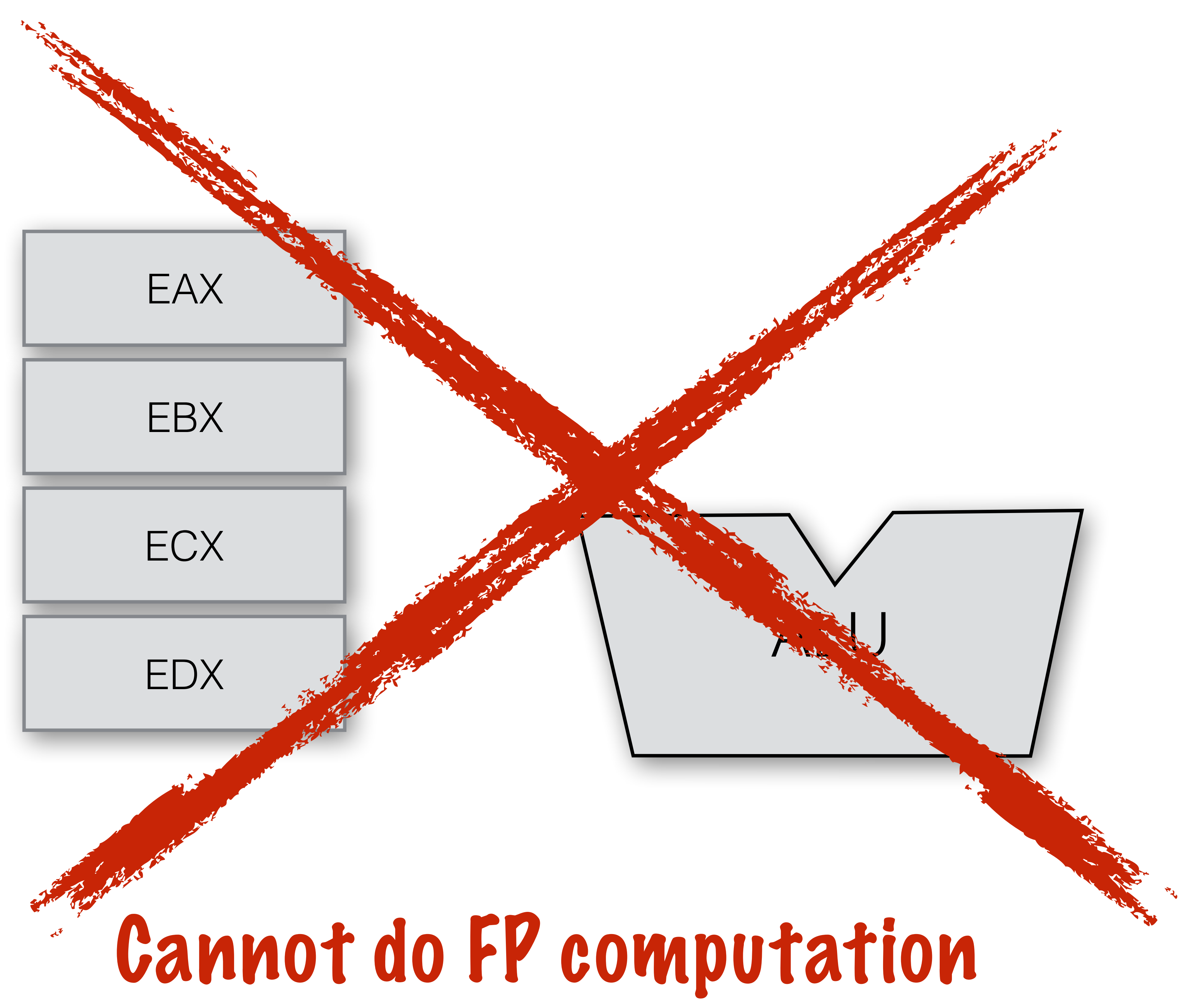
How do we compare two FP numbers?

Check sign bits first, the compare as unsigned integers! No unpacking necessary!

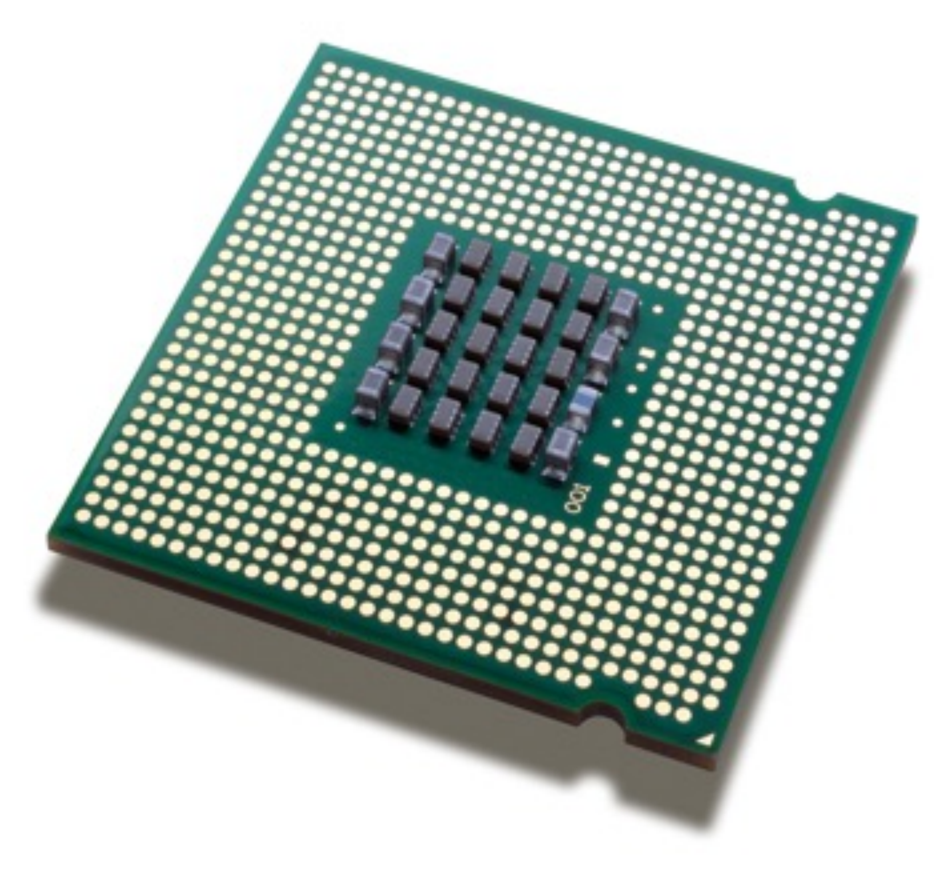
Programming FP Operations in Assembly…

Pentium





EAX
EBX
ECX
EDX
Cannot do FP computation



CLOCK DRIVER
CODE CACHE
CODE TLB
INSTRUCTION FETCH
INSTRUCTION DECODE
BRANCH PREDICTION LOGIC
BUS INTERFACE LOGIC
COMPLEX INSTRUCTION SUPPORT
DATA TLB
SUPERSCALER INTEGER EXECUTION UNITS
DATA CACHE
PIPELINED FLOATING POINT
MP LOGIC

http://download.intel.com/pressroom/kits/pentiumee/pentiumee_processor_back.jpg
https://people.cs.clemson.edu/~mark/330/colwell/pentium.gif

Intel Pentium 5 Prescott

Trace Cache Access, next Address Predict

Instruction Trace Cache

Execution Pipeline Start

Buffer Allocation & Register Rename

Trace Cache Branch Prediction Table (BTB), 1024 entries.

Return Stacks (4 x16 entries)

Trace Cache next IP's (4x)

Fill Buffers

Trace Cache

Micro code Sequencer

Micro code Flash & ROM

uOp Queue

Register Alias History Tables (4x128)

Register Alias Tables

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

16k uOps
128 kByte

8 way set associative

8 x 512 sets of 4 uOps

Tag comparators

39 bit virtual Tags

LRU

LRU

Misc. Tag Data

La Grande? uControler, RAM/ROM

Instruction Decoder

Up to 4 decoded uOps/cycle out (from max. one x86 instr/cycle) Instructions with more than four are handled by Micro Sequencer

Raw Instruction Bytes in

Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

Instruction TLB's 128 entry, fully associative for 4k and 4M pages. In: Virtual address [47:12] Out: Physical address [39:12] + 2 page level bits

uOp Schedulers

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)

FP Move Scheduler: (8x8 dependency matrix)

Load / Store Linear Address Collision History Table

Load / Store uOp Scheduler: (8x8 dependency matrix)

14 12 13 11 1 8 6 7 5 4 3 2 9 10

14 12 13 1 5 6 4 3 2

FP, MMX, SSE1..3

Floating Point, MMX, SSE1..3 Renamed Register File 256 entries of 128 bit.

Floating Point Registers

Legacy Floating Point Multiply

Legacy Floating Pnt.Add

Instruction Fetch from L2 cache and Branch Prediction

Front Side Bus Interface, 533..800 MHz

512 kByte L2 Cache Block

L2 Phys. Tags

L2 Phys. Tags

L2 Cache Line Transfer Buffers

512 kByte L2 Cache Block

Integer Execution Core

(1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle
(2) Integer Renamed Register File 256 entries of 32 bit (+ 6 status flags) 12 read ports and six write ports
(3) Databus switch & Bypasses to and from the Integer Register File.
(4) Flags, Write Back
(5) Double Pumped ALU 0
(6) Double Pumped ALU 1
(7) Load Address Generator Unit
(8) Store Address Generator Unit
(9) Load Buffer (96 entries)
(10) Store Buffer (48 entries)
(11) ROB Reorder Buffer 4x64 entries
(12) 16 kByte Level 1 Data cache four way set associative. 1R/1W
(13) Databus multiplexing
(14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

April 19, 2003 www.chip-architect.com

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

SP0

SP1

SP2

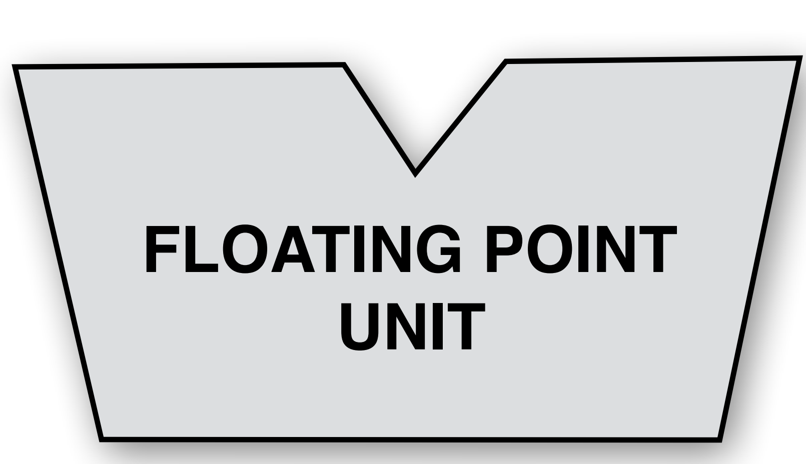
SP3

SP4

SP5

SP6

SP7



Operation: (7.0+10.0)/9.0

SP0

SP1

SP2

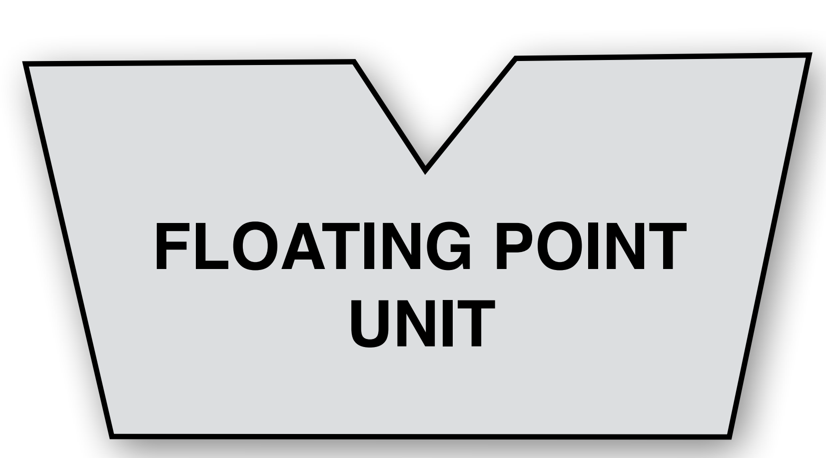
SP3

SP4

SP5

SP6

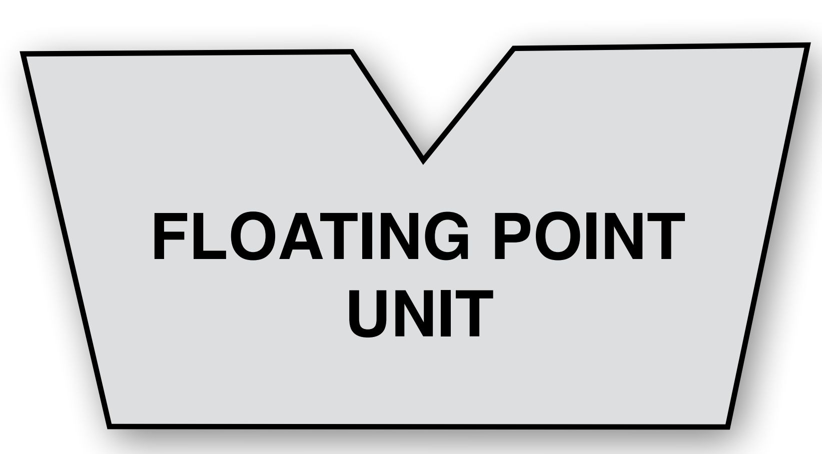
SP7



Operation: (7+10)/9

fpush 7

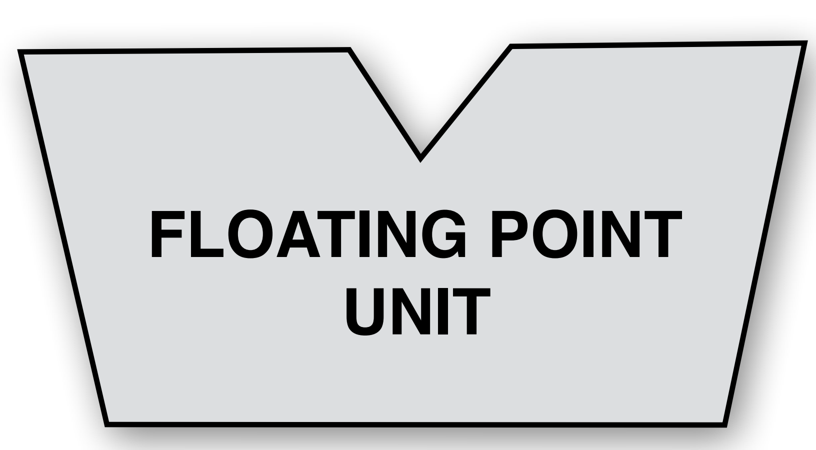
SP0	7
SP1	
SP2	
SP3	
SP4	
SP5	
SP6	
SP7	

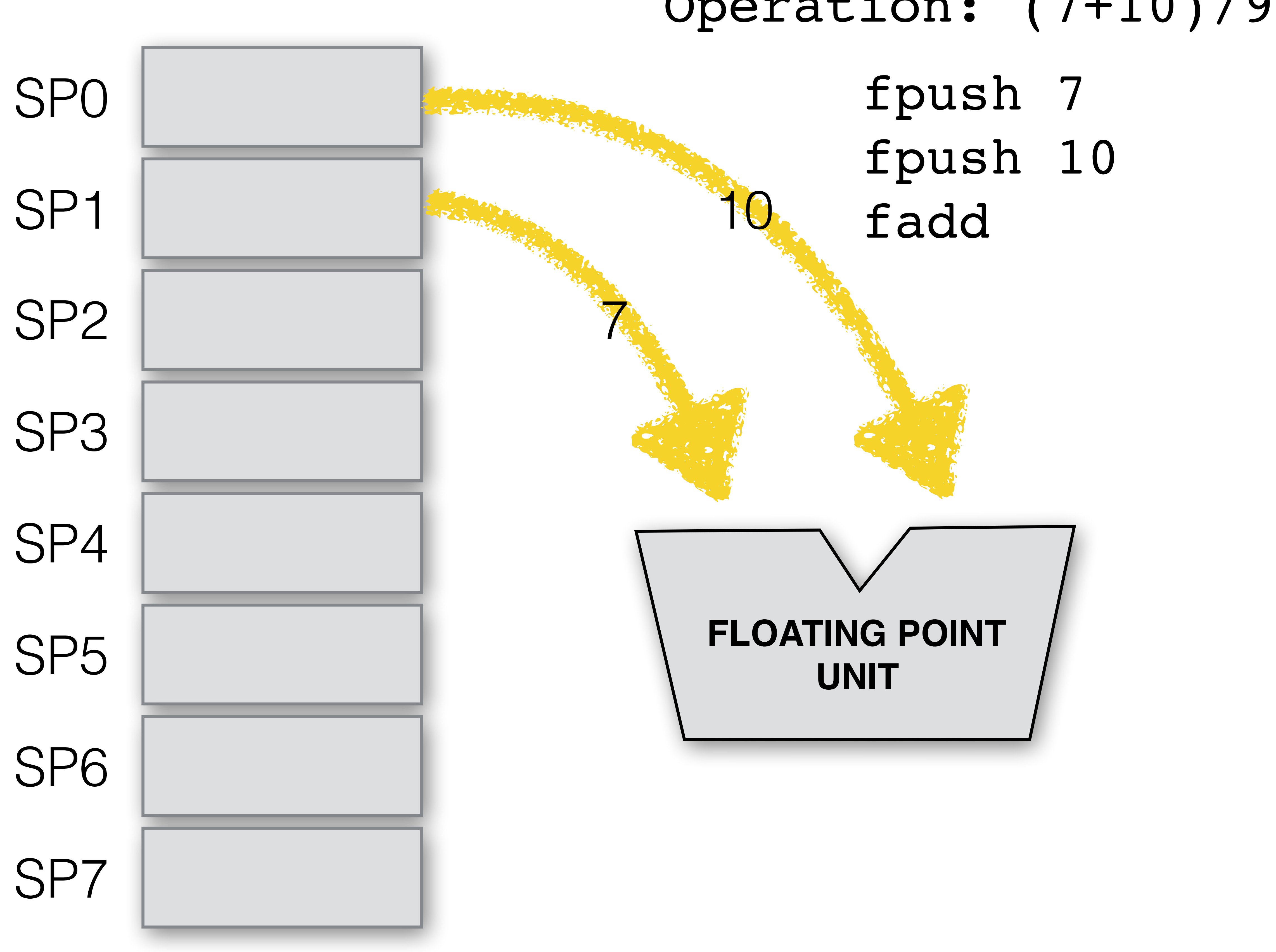


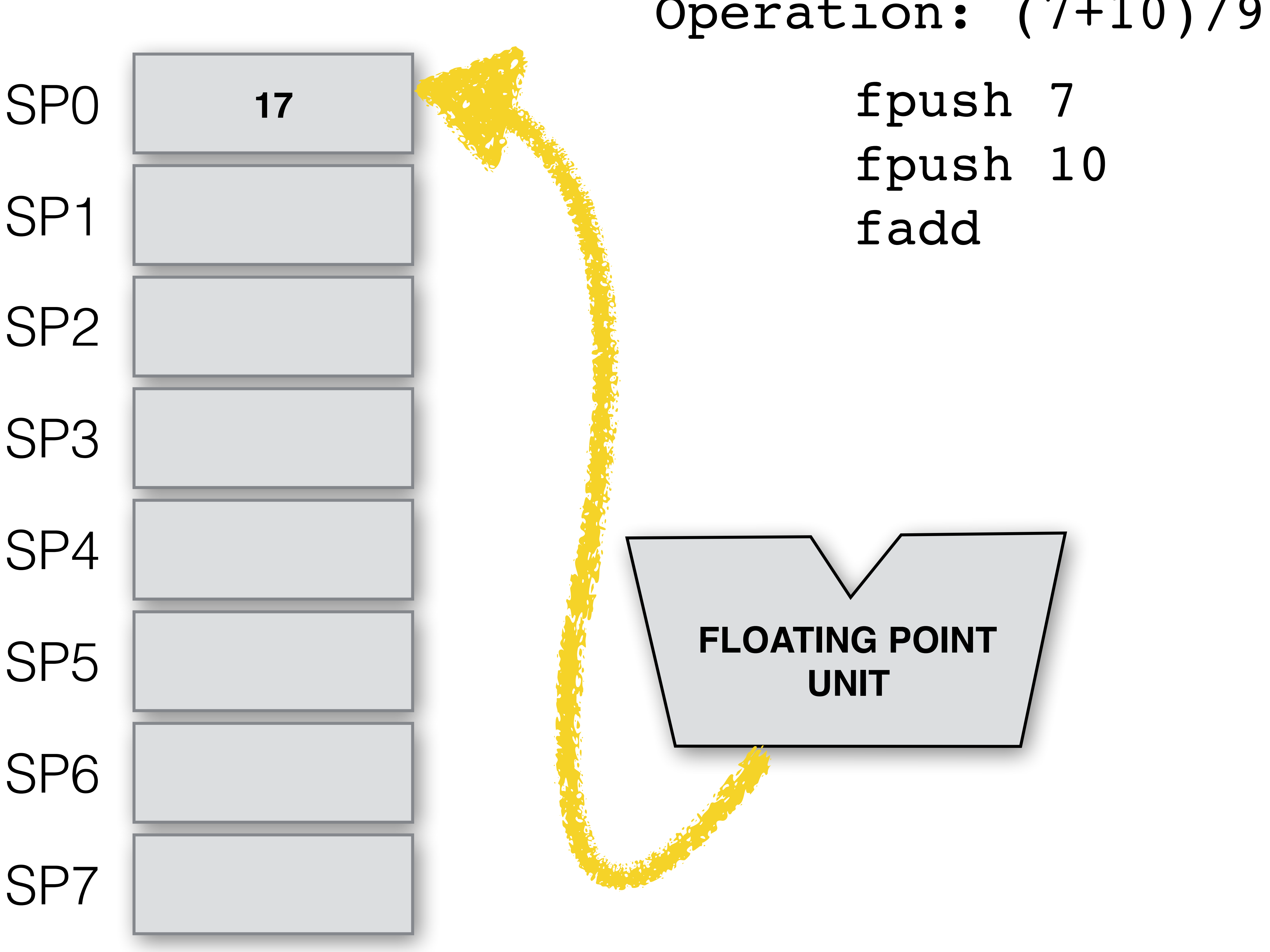
SP0	10
SP1	7
SP2	
SP3	
SP4	
SP5	
SP6	
SP7	

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

fpush 7
fpush 10
```

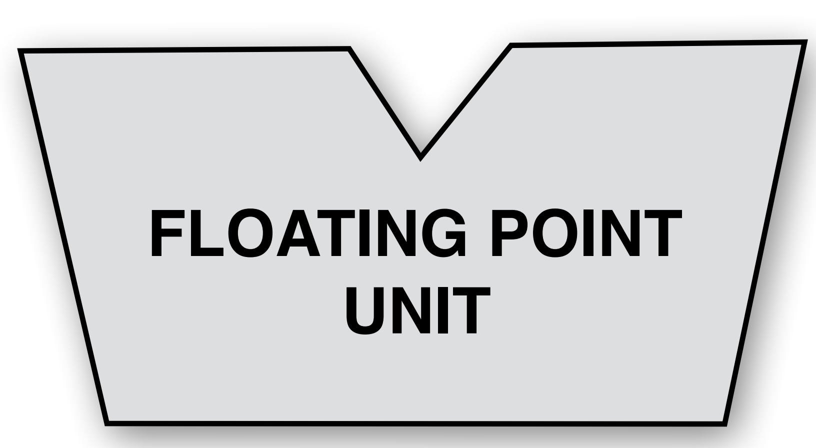


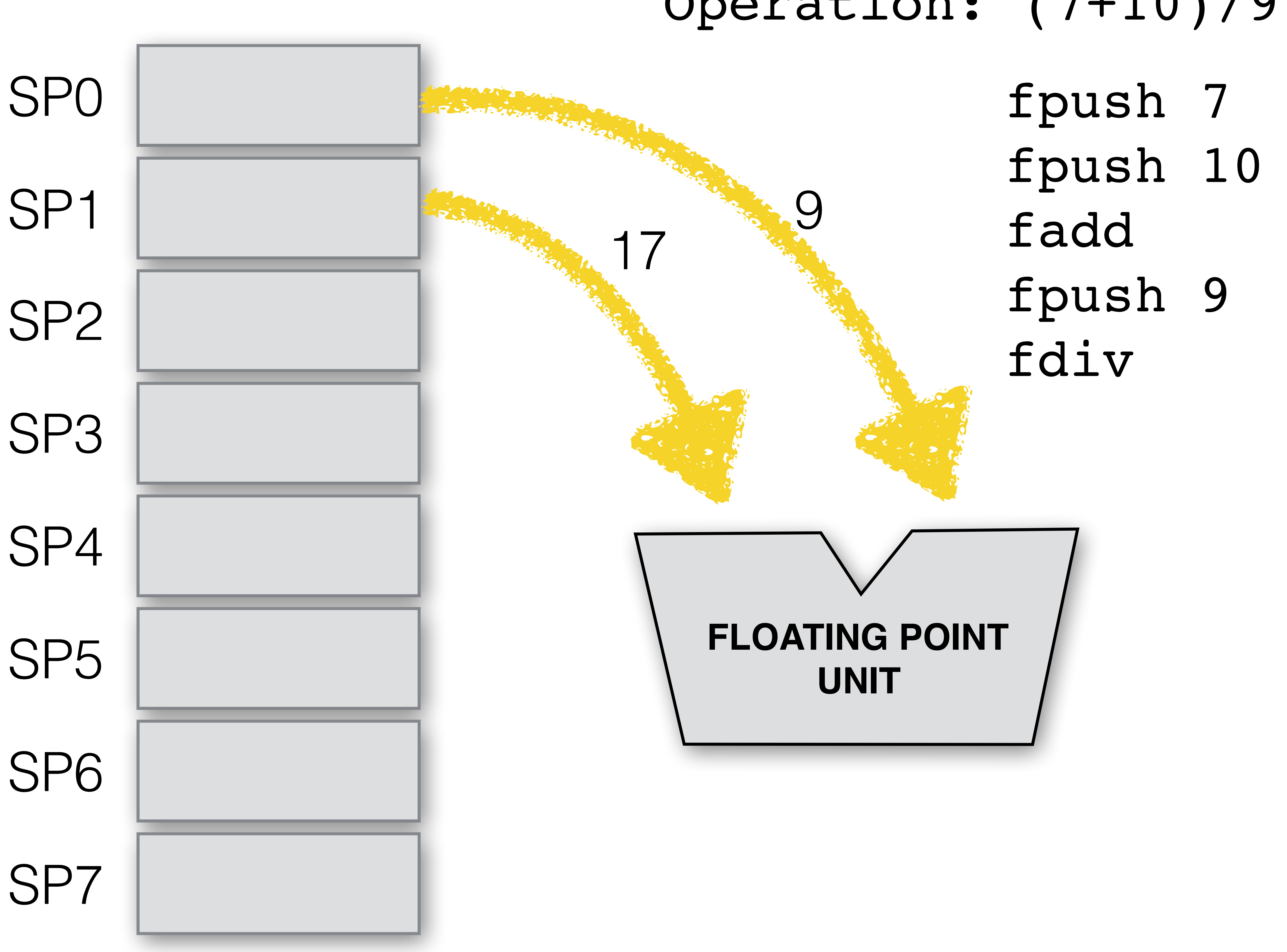

Operation: (7+10)/9
SP0
SP1
SP2
SP3
SP4
SP5
SP6
SP7
10
7
fpush 7
fpush 10
fadd
FLOATING POINT
UNIT

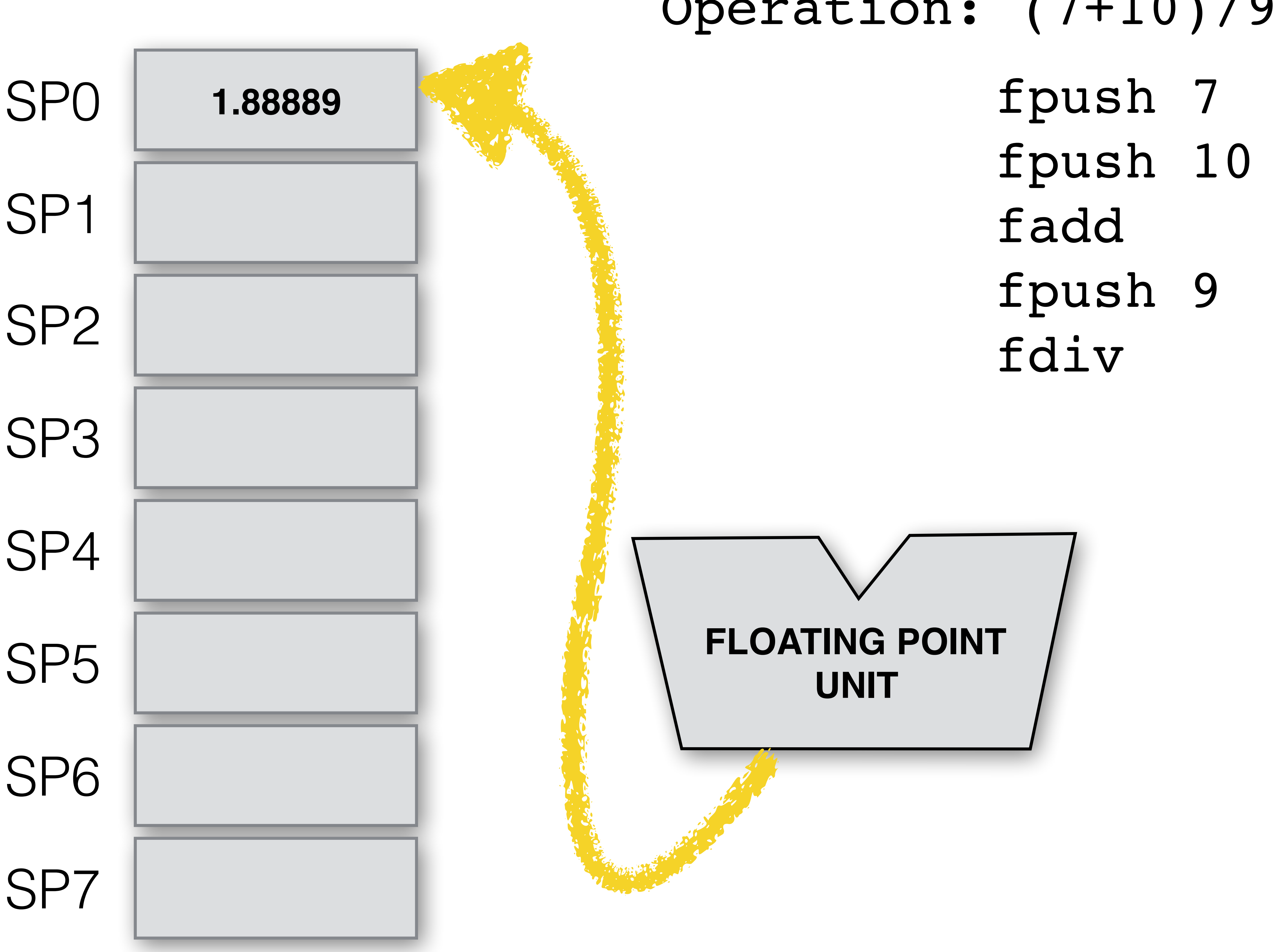

Operation: (7+10)/9
fpush 7
fpush 10
fadd
SP0
17
SP1
SP2
SP3
SP4
SP5
SP6
SP7
FLOATING POINT UNIT

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

fpush 7
fpush 10
fadd
fpush 9
```




Operation: (7+10)/9
SP0
SP1
SP2
SP3
SP4
SP5
SP6
SP7
17
9
fpush 7
fpush 10
fadd
fpush 9
fdiv
FLOATING POINT
UNIT


Operation: (7+10)/9
SP0
1.88889
SP1
SP2
SP3
SP4
SP5
SP6
SP7
fpush 7
fpush 10
fadd
fpush 9
fdiv
FLOATING POINT
UNIT

The Pentium computes FP expressions using RPN!

The Pentium computes FP expressions using RPN!

Reverse Polish Notation

https://en.wikipedia.org/wiki/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;
}
```

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

More code examples here:

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

THE END!