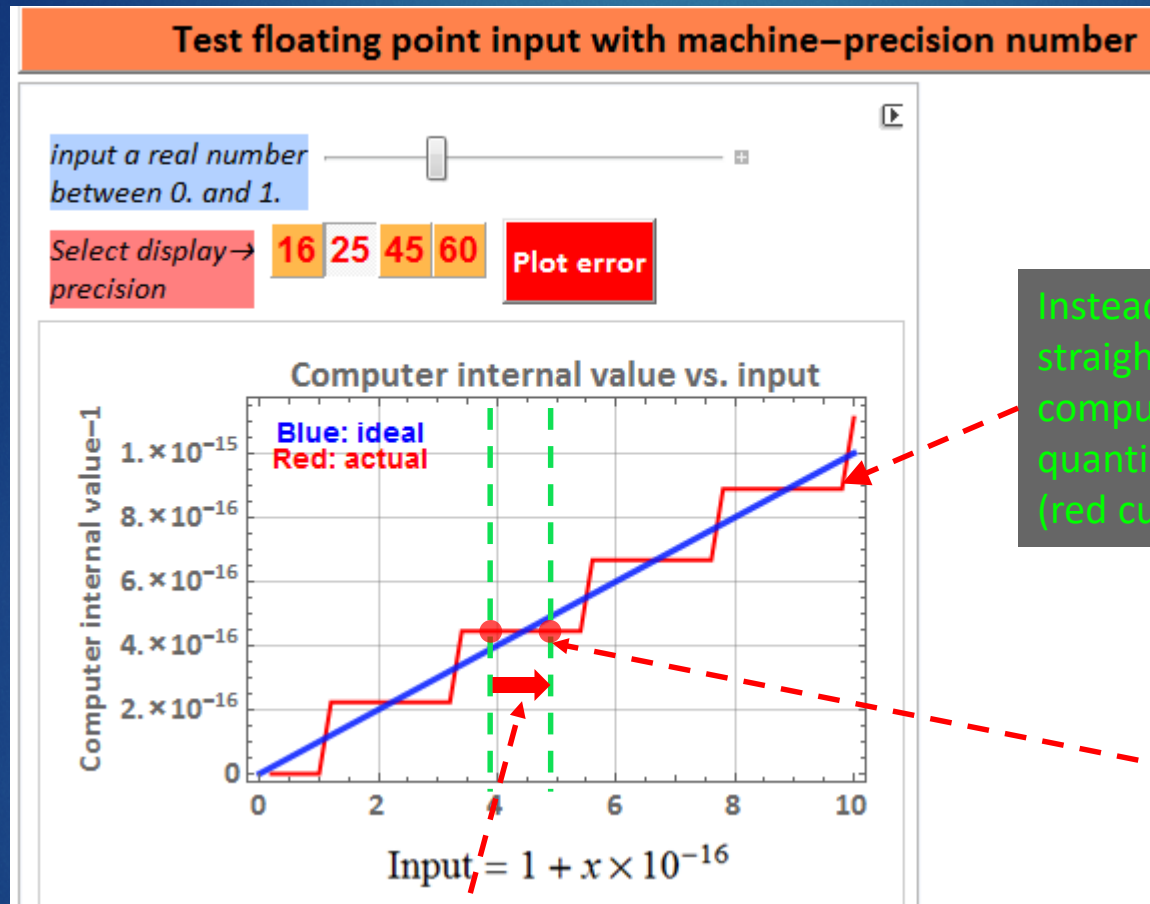


ECE3340

Binary representation of numbers

PROF. HAN Q. LE

# Remember this? now let's talk about it.




Instead of giving us a linear straight line like the blue line, the computer actual values are quantized into discrete step values (red curve). Why?

The computer can't tell the difference and hence, it returns to you zero.

you deposited  
\$1 mil




**machine precision** problem

A black t-shirt is centered against a blue background. The t-shirt has a white graphic on the chest and a small teal tag at the collar. The graphic consists of a white square containing a black '0' followed by the text 'There are only 10 types of people in the world, those who understand binary and those who don't'.

There are only 10 types  
of people in the world,  
those who understand binary  
and those who don't

We live in a digital world

A globe of Earth is shown in the upper right corner, appearing to float above a vast, glowing digital landscape. The landscape is composed of numerous parallel lines that recede into the distance, creating a sense of depth. The lines are illuminated with a mix of blue and orange-red lights, suggesting a complex network of data or digital infrastructure. The overall scene is set against a dark, gradient background that transitions from a light blue at the top to a dark purple at the bottom.

As we know, digital  
computer deals only  
with bits 1 and 0

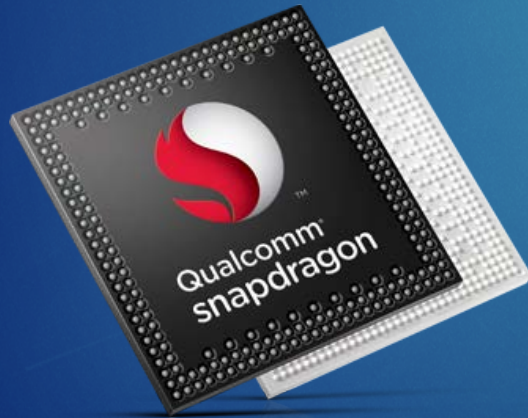
2005

Applications	32-bit	32-bit	32-bit x64	x64	Itanium
Windows Server	32-bit	32-bit	x64	x64	Itanium
Device Drivers	32-bit	32-bit	x64	x64	Itanium
Server Hardware	32-bit	x64	x64	x64	Itanium

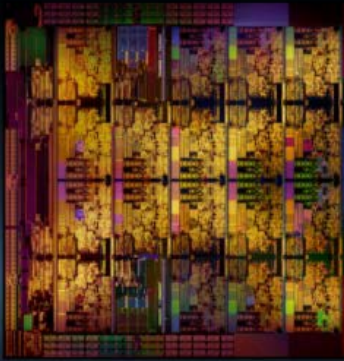
← Pure 32-bit Stack      Pure 64-bit Stack



Everything is 64 bit now



## NEW DIE MAP FOR INTEL® CORE™ X-SERIES PLATFORM



### INTEL® CORE™ i9-7980XE PROCESSOR DIE MAP

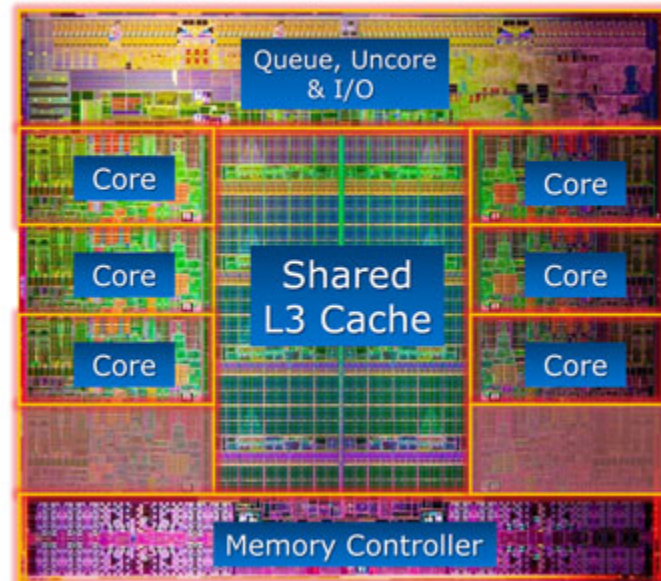
14 nm tri-gate 3D transistors



14

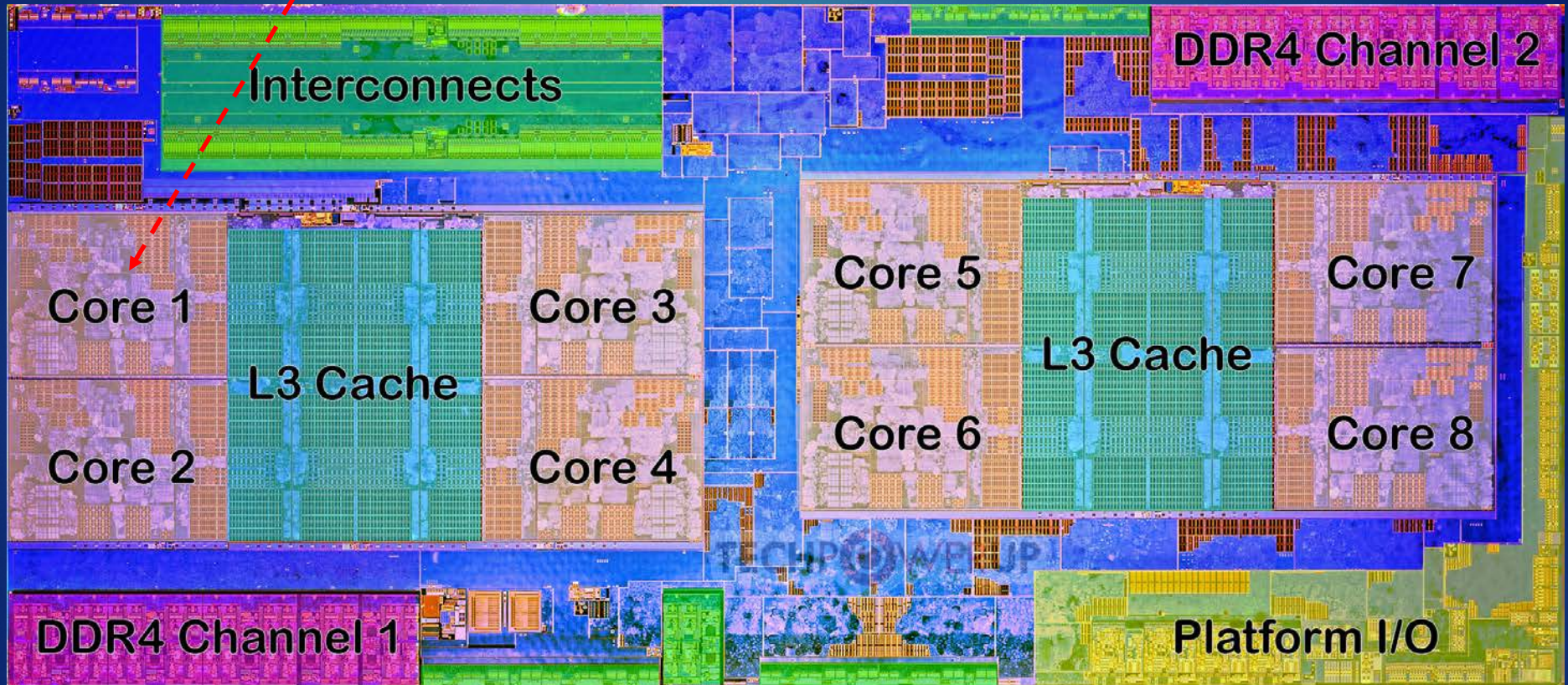
Floating point unit (FPU) inside each core

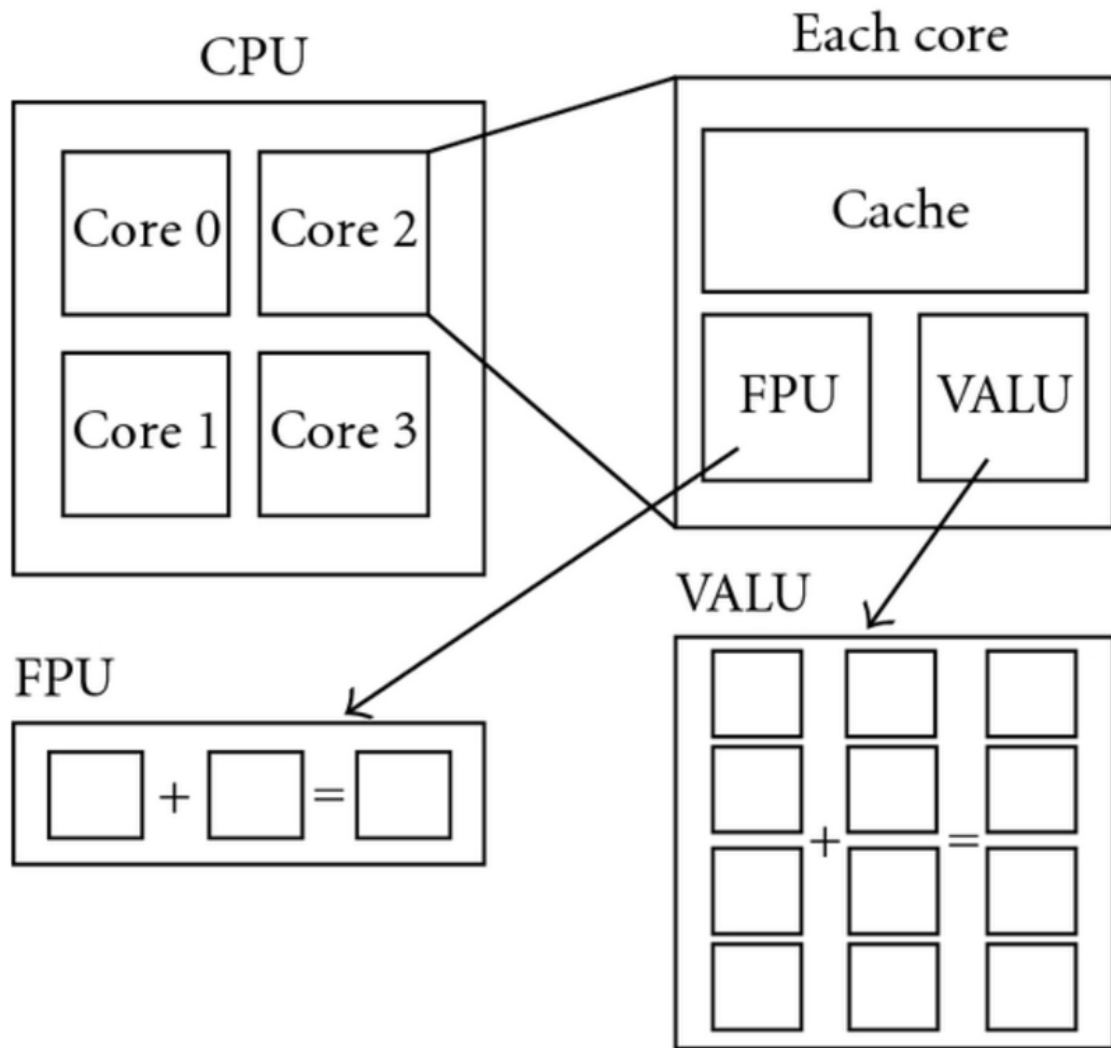
## Intel® Core™ i7-3960X Processor Die Detail



FPU in each core

# AMD Threadripper









APP ECE 3340-APP-  
1.0.1.1\_binary

# Assignment APP

BINARY REPRESENTATION OF NUMBERS

There are 3 choices to input a number

Choice 1: (run Mathematica) – click ENTER. A dialog box allows you to input a real (FP) or integer number of precision up to 100 decimal digits

Choice 2: Input a FP using mantissa and exponent sliders in this part

Choice 3: Input an integer up to 20 decimal digits using dials of this part

The APP will know whether you input an FP or an integer and states it here

It will tell the precision level of the input number

It will show what the actual value of the input is, based on internal binary representation

Click open

Click open

It will show value of the input in other bases from 2-16, selected by clicking here. For example, it shows the input in hex base here: 2.54b\*16^-15

# Problem 1

Enter the APP an integer that includes your birthdate in the following format: mmddyyyy. Example 02041998 for Feb 4 1994. (obviously in this case, you can drop the first digit 0). Obtain:

1. the number in hexadecimal, and in the binary representation (same as base 2)
2. write a code to verify you obtain the same results as the APP.
3. Do questions 1 and 2 again but this time with your student ID number in front of your birthdate number, for example: 12345678902041998.
4. Discuss any difference between the two numbers in terms of the number of bytes required.

You can use choice 1: (run Mathematica) – click ENTER. A dialog box allows you to input a real (FP) or integer number of precision up to 50 decimal digits

or directly dial in an integer up to 20 decimal digits using dials of this part

The image shows a Mathematica interface with several components:

- Top Left:** A button labeled "ENTER" with a red arrow pointing to it from the text above.
- Integer input panel:** A panel with a title "Integer input" containing two rows of dials. The top row has dials for  $10^9$  (0),  $10^6$  (1),  $10^3$  (331), and  $10^0$  (994). The bottom row has dials for  $10^{18}$  (0),  $10^{15}$  (0), and  $10^{12}$  (0). A red dashed oval highlights the top row dials, with a red arrow pointing to the text "or directly dial in an integer up to 20 decimal digits using dials of this part".
- Bottom Left:** A "precision level" dropdown set to  $\infty$ , a "CPU word length" field showing "64", and a "low byte" button.
- Bottom Center:** A 2x8 grid of binary digits:   
Row 1: 1 1 1 1 1 0 1 0   
Row 2: 0 0 0 1 0 1 0 0
- Bottom Left:** A 4x4 grid of red dots.
- Bottom Right:** A dialog box titled "Precision of previous entry: 7" with the prompt "Enter a new number below:". The input field contains "1311994" and an "OK" button. A red arrow points from the text "A dialog box allows you to input a real (FP) or integer number of precision up to 50 decimal digits" to this dialog box.

RUN STATUS/CONTROL →



### Common binary representation of numbers in most computers

Free form number entry if run APP in Mathematica → ENTER

#### Real input

sub-1-mantissa:

$10^N$ :

#### Integer input

$10^9$	$10^6$	$10^3$	$10^0$
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
$10^{18}$	$10^{15}$	$10^{12}$	
<input type="text"/>	<input type="text"/>	<input type="text"/>	

Your input

Your input in hexadecimal

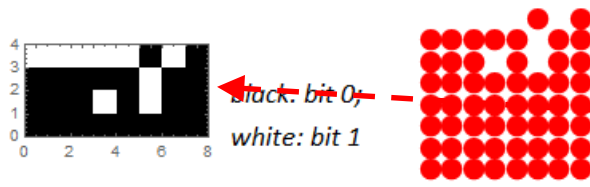
number input: integer → **1,311,994** precision level → ∞

in other base: 16 **1404fa<sub>16</sub>** CPU word length: 64

hi byte      low byte

2B-part 0	0 0 0 0 1 0 0	:	1 1 1 1 1 0 1 0
2B-part 1	0 0 0 0 0 0 0 0	:	0 0 0 1 0 1 0 0

Binary representation of your input. This shows it needs a minimum of 3 bytes. Shown here 2 full 16-bit units, even the last byte is just 0.



These help to visualize the bit pattern

# Problem 2

Enter the APP a floating point number that includes your birthdate in the following format:  $0.mmdyyy * 10^{(m1d1)}$ , where  $m1$  is the first non-zero digit of your month and  $d1$  is the first non-zero digit of your date. Example 02041999 for Feb 4 1999 will be entered as  $0.02041999 * 10^{24}$ . Obtain:

1. the number in hexadecimal and in base 2 (not the same as the binary representation)
2. The 11-bit exponent and the 52-bit mantissa of the binary representation. Then use Mathematica command `RealDigits[yournumber, 2]` to obtain the 53-bit mantissa in the first part of the output. Drop the 1<sup>st</sup> bit because it is always 1. Verify the remaining 52 bits match the APP output.
3. The last bit of the mantissa represent  $2^{-52}$ . This is the smallest difference between 2 numbers of the same exponent. Calculate it and verify that it is machine epsilon.
4. The bit pattern (copy, or print screen and paste)
5. Repeat questions 1-2 again but this time with your student ID number in front of your birthdate number in this way: `studentID.birthdate`. Example: `123456789.1311994` (use ENTER command, do not use sliders because it is pretty long). Obtain the bit pattern, which is your personal coded ID.

If the input is a FP number, it will show:  
 - 1 sign bit: 0 for  $\geq 0$  and 1 for  $\leq 0$

- 11 exponent bits

- 52 bits for the mantissa (which actually has 53 bits. The first bit is always 1 and needs not be stored here).

number input: real → **2.021 658 611 376 729 × 10<sup>-18</sup>** precision level → MachinePrecision

in other base: 16 **2.54b<sub>16</sub> × 16<sup>-15</sup>** CPU word length: 32

actual value: 131213,1457110159  
 649,0371073168,5345356631,2041152512

**2.0216586113 7672901505 3385403760 4932023117 723473358 × 10<sup>-18</sup>**

sign bit	exponent	mantissa
0	0 1 1 1 1 0 0 0 1 0 0	0 0 1 0 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0 0 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0 1 1 1 1 0 0 ◆ ◆ ◆ ◆

0 0 1 1 1 1 0 0  
 0 1 0 0 0 0 1 0  
 1 0 1 0 0 1 0 1  
 1 0 0 0 0 0 0 1  
 1 0 1 0 0 1 0 1  
 1 0 0 1 1 0 0 1  
 0 1 0 0 0 0 1 0  
 0 0 1 1 1 1 0 0

black: bit 0;  
 white: bit 1

total is a 64-bit word – shown here as 8 bytes

The 8 bytes can be arranged in 8x8 dot matrix display

# Problem for fun only

Find floating points or integers (easier) that have bits patterns representing the initials of your first and last name.

Example: 16,419,452,012,919,037,084 and 4,123,389,611,252,201,785 will give something. Check it out.

Below are illustrations of common patterns: each has a number. Find your "digital signature" with your name initials.

**Find floating points and/or integers for these patterns**

The image displays a grid of 21 patterns arranged in three rows. The first two rows consist of colorful dot patterns representing letters and numbers. The first row contains: a red 'E', an orange 'C', a yellow 'E', a light green '3', a dark green '3', a blue '4', and a purple 'O'. The second row contains: a green 'O', a red 'U', a red 'H', a teal 'W', a pink 'A', and a cyan 'B'. The third row contains four black and white pixelated patterns: a checkerboard, a square with a hole, a diagonal line, and a circle with a hole.



*Is there something fundamental and natural about how to represent a FP in binary? Is there something “sacred” about reserving 11 bits (instead of 10, 12,...) for exponent and 52 bits for mantissa?*

**No. None. Nada....**

- It is just a rule to map a FP number to a set of bits that the computer handles; based on practical considerations for applications.
- The rule can be thought of as a “binary-coded-decimal” convention, although IBM actually used that exact expression for their computers (aka BCD and EBCDIC in analogy to 8-bit ASCII) in the 60’s.
- The rules can be made up by different computer makers, different entities/organizations, and evolved over time.
- What we see here in this APP is IEEE-754 convention, which is useful for a wide range of applications and virtually the standard in all PC’s.

*So, do I need to know the history, the evolution, the rationale, the melodrama or tragicomedy behind the story how binary-FP has been done over the course of computing history?*

**No. Unless you are a computer science historian.**

# Loss of precision

OR *“HOW I LOST \$1 M TO A 16-BIT EXCEL MACHINE”*

Here is an example what happens in Excel:

	x array	y=1+x array	x0=y-1 array
1	small number	column A+B	Column C-A
1	5.000E-15	1.00000000000001000000E+00	5.10702591327572000000E-15
1	4.000E-15	1.00000000000000000000E+00	3.99680288865056000000E-15
1	3.000E-15	1.00000000000000000000E+00	3.10362412895044000000E-15
1	2.000E-15	1.00000000000000000000E+00	1.99345147432528000000E-15
1	1.900E-15	1.00000000000000000000E+00	1.99840144432528000000E-15
1	1.800E-15	1.00000000000000000000E+00	1.77635683940025000000E-15
1	1.700E-15	1.00000000000000000000E+00	1.77635683940025000000E-15
1	1.600E-15	1.00000000000000000000E+00	0.00000000000000000000E+00
1	1.500E-15	1.00000000000000000000E+00	0.00000000000000000000E+00

Remember this?

Notice that x and x0 are not equal as expected.

What we see here is the limit of **machine precision** which causes the **inaccuracy** observed.

what happens right here? why x0 cannot gradually go from  $1.776 \cdot 10^{-15}$  to  $\sim 1.6 \cdot 10^{-15}$  as expected, but jumps to zero?

imagine this: there is an investment fund containing gazillion dollars. Asset= 1 gazillion



You add to the account your saving of \$1 mil, because it promises 50% profit return in 1 year



but the investment company uses Excel on 16-bit CPU and your portion is below its precision, hence, it flushes to zero


1.77635683940025000000E-15  
0.00000000000000000000E+00

Hence, after one year, your account is deposited with this amount



0.000000000000000000000000000000E+00

At least, it has a lot of zeros!



Now, we look at this problem  
again but at the bit-level  
(binary of FPU process)

Click open this APP

UNIVERSITY of HOUSTON App by Han Q. Le ©

ECE 3340 – APP 1.0.2.1 – Loss of precision in adding/subtracting

RUN STATUS/CONTROL

Loss of precision illustration in binary

$\epsilon$  mantissa   $\times 10^{\wedge N}$  [0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15]

$\epsilon = 1$   $y = 1 + \epsilon$

1	10000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
$\epsilon$	10000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
y	10000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
y - 1	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

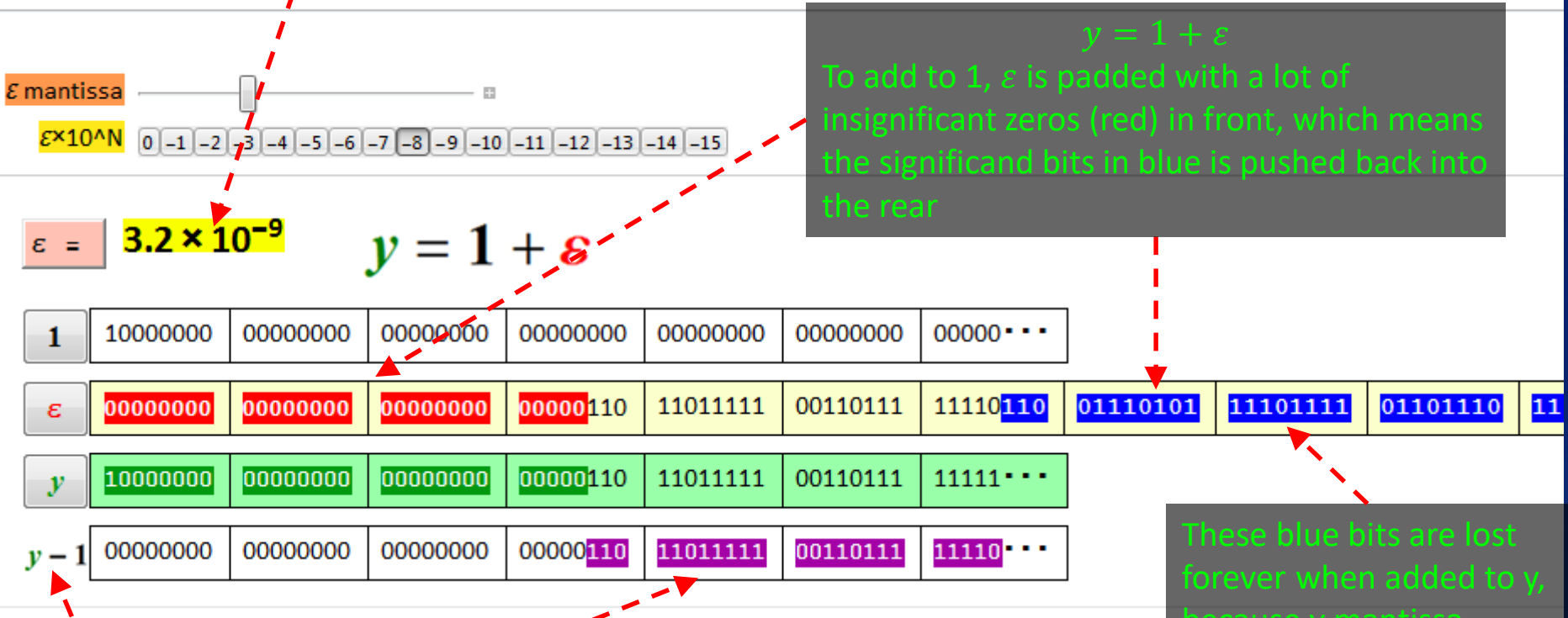
Control epsilon by selecting its mantissa and exponent. Shown here is when  $\epsilon = 1$

$y = 1 + \epsilon$   
Here,  $\epsilon = 1$  and  $y=2$  (its first digit is in front of both 1 (top) and  $\epsilon$  (row 2)).

We are interested in  $\delta = y - 1 = (1 + \epsilon) - 1$   
Is  $\delta = \epsilon$  as it should be? in this case, it is, as both = 1.

Here, we choose  $\epsilon$  to be a small number

### Loss of precision illustration in binary



$y = 1 + \epsilon$   
To add to 1,  $\epsilon$  is padded with a lot of insignificant zeros (red) in front, which means the significant bits in blue is pushed back into the rear

These blue bits are lost forever when added to  $y$ , because  $y$  mantissa doesn't have room to save them

We are interested in  $\delta = y - 1 = (1 + \epsilon) - 1$   
Is  $\delta = \epsilon$  as it should be? No, because it retains only a portion of  $\epsilon$  in purple. This is how precision is lost.  $y$  is so close to 1 and their subtraction retains only the purple portion. If  $\epsilon$  is less than **machine epsilon**, we have nothing but zero left.

# Problem 3

Use APP “1.0.2.1 loss of precision” to do the following: Chose  $\varepsilon$  mantissa to be “0.mmddyyyy” where mmddyyyy is your birthdate (do not include the front zero if you are born between Jan and Sept – for example, if you are born on March 21, 1999, just type in number 0.3211999. If your BD is 12/25/1998, then type in 0.12251998).

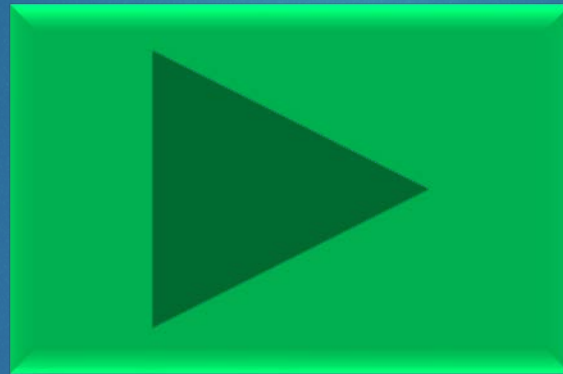
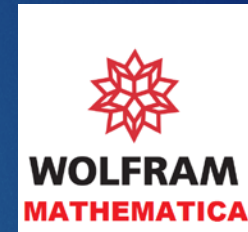
Follow instruction

$\varepsilon$ value (in decimal)	# $\varepsilon$ lost bits (blue)	# $\varepsilon$ bits added to y	# $\delta$ significant bits (purple)	$\delta$ value (decimal)	Your birthdate decoded from $\delta$ based on 0.mmddyyyy, and how “old” are you?

hint: For the next to last column, you can use Mathematica command **FromDigits** (demo in class).



# A note about Mathematica precision options



So, after all said and done, can I just use Mathematica arbitrary precision and skip all these hassles?

Yes and no. In some problems, yes.

Remember that for serious number crunching jobs that require huge # of FLOPs, **native machine processing is still the fastest.**

Write smart, robust, error-proof codes that can exploit machine FLOP with results well within our tolerances.