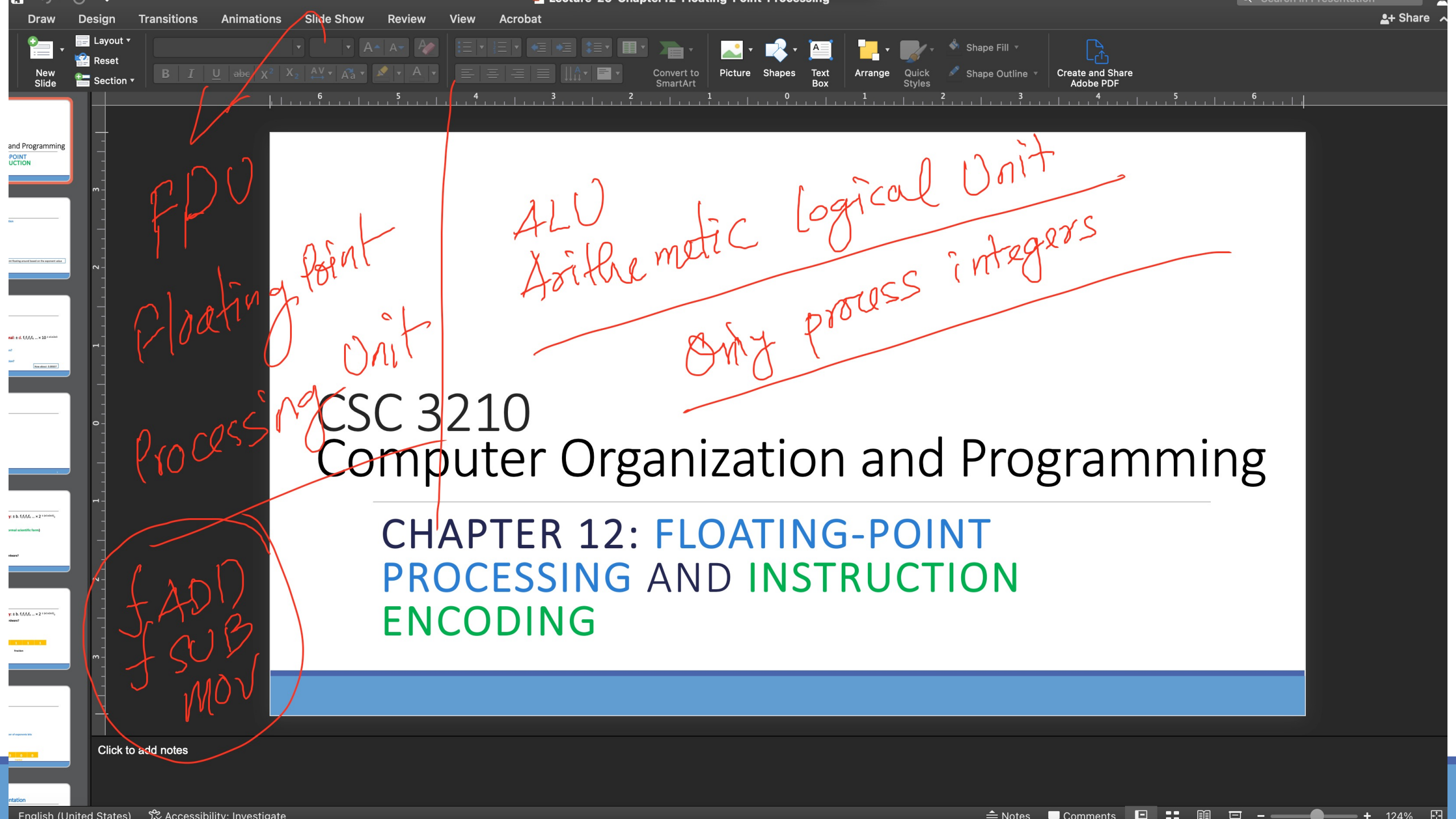

Draw
Design
Transitions
Animations
Slide Show
Review
View
Acrobat
Share
New Slide
Layout
Reset
Section
Convert to SmartArt
Picture
Shapes
Text Box
Arrange
Quick Styles
Shape Fill
Shape Outline
Create and Share Adobe PDF
FPU
Floating point
Processing Unit
ALU
Arithmetic logical Unit
Only process integers
fADD
fSUB
MOV
CSC 3210
Computer Organization and Programming
CHAPTER 12: FLOATING-POINT PROCESSING AND INSTRUCTION ENCODING
Click to add notes
English (United States)
Accessibility: Investigate
Notes
Comments
124%

16 ⟶ 10000 ⟹ Memory Reg.

$\pi = 3.1416 = \pm 31.416 \times 10^{-1} = 314.16 \times 10^{-2} =$

Floating-Point Numbers

- Programming languages support **numbers with fraction**
  - **Called floating-point numbers**
- **Examples**:
  - 3.14159265… (π)
  - 2.71828… (e)
- Scientific form: **26.6**

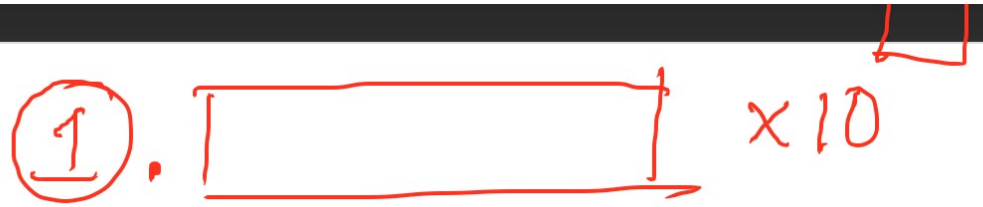
26.6×10^{4} = 266000

26.6×10^{-1} = 2.66

26.6×10^{-2} = 0.267

  - **We have to choose one form.**

The point floating around based on the exponent value



Floating-Point Numbers

403.16 → 4.0316 x 10^2

- We use a **Normal scientific notation** to represent
  - Very small numbers (e.g. 1.0×10^{-9})
  - Very large numbers (e.g. 8.64×10^{13})
- **Normal Scientific notation for Decimal**: $\pm$ d. $f_1f_2f_3f_4 \ldots \times 10^{\pm e1e2e3}$

$$3.14 = 3.14 \text{ X } 10^0$$

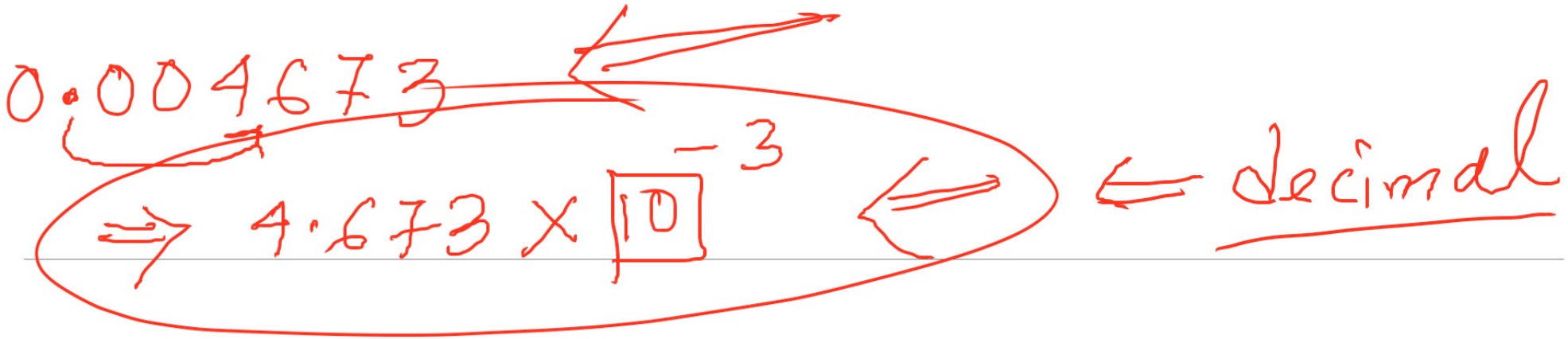
  - **366.99** is not in a **Normal Scientific notation?**

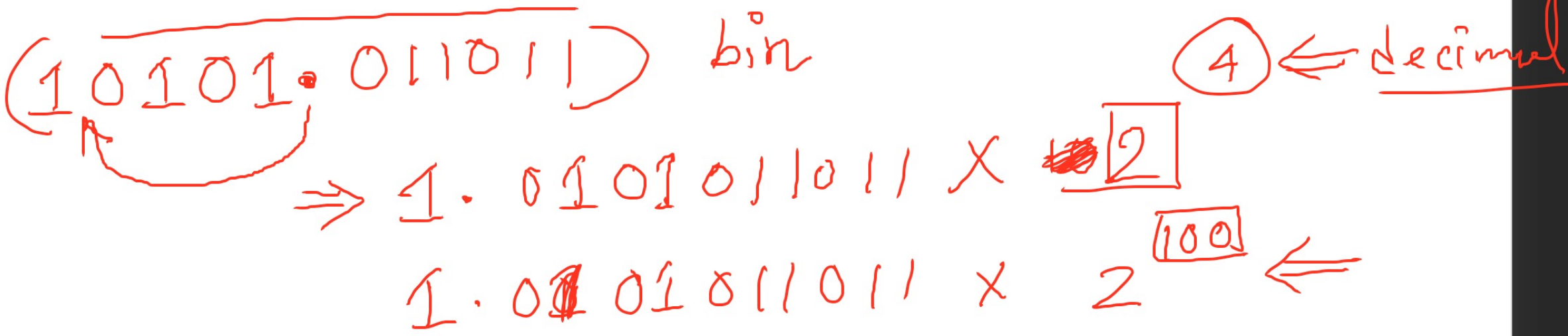
    $3.6699 \text{ X } 10^2$

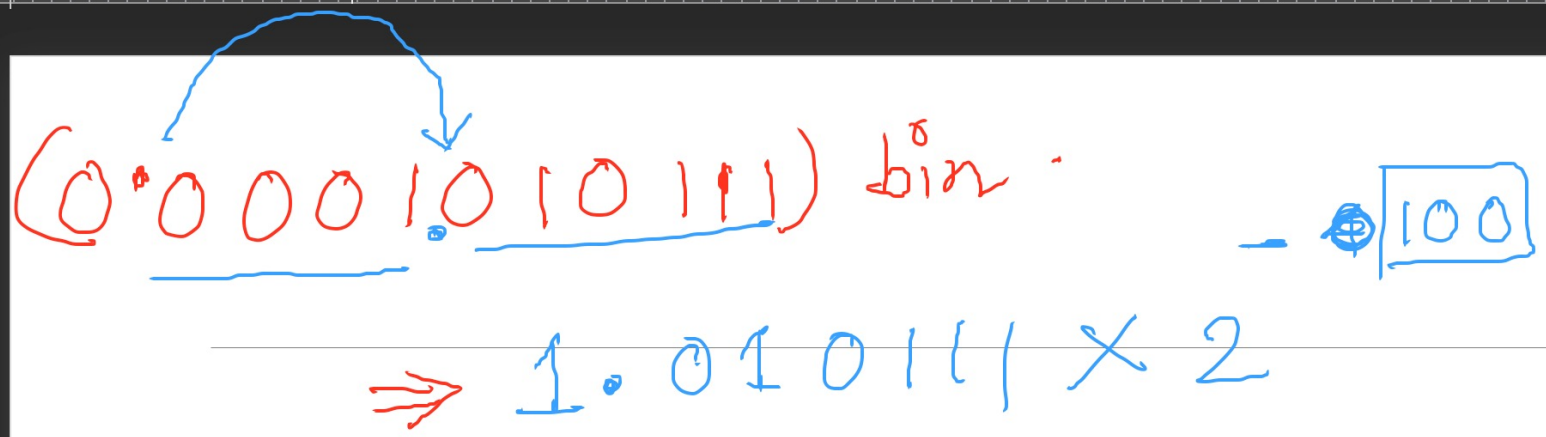
  - **0.36699** is not in a **Normal Scientific notation?**

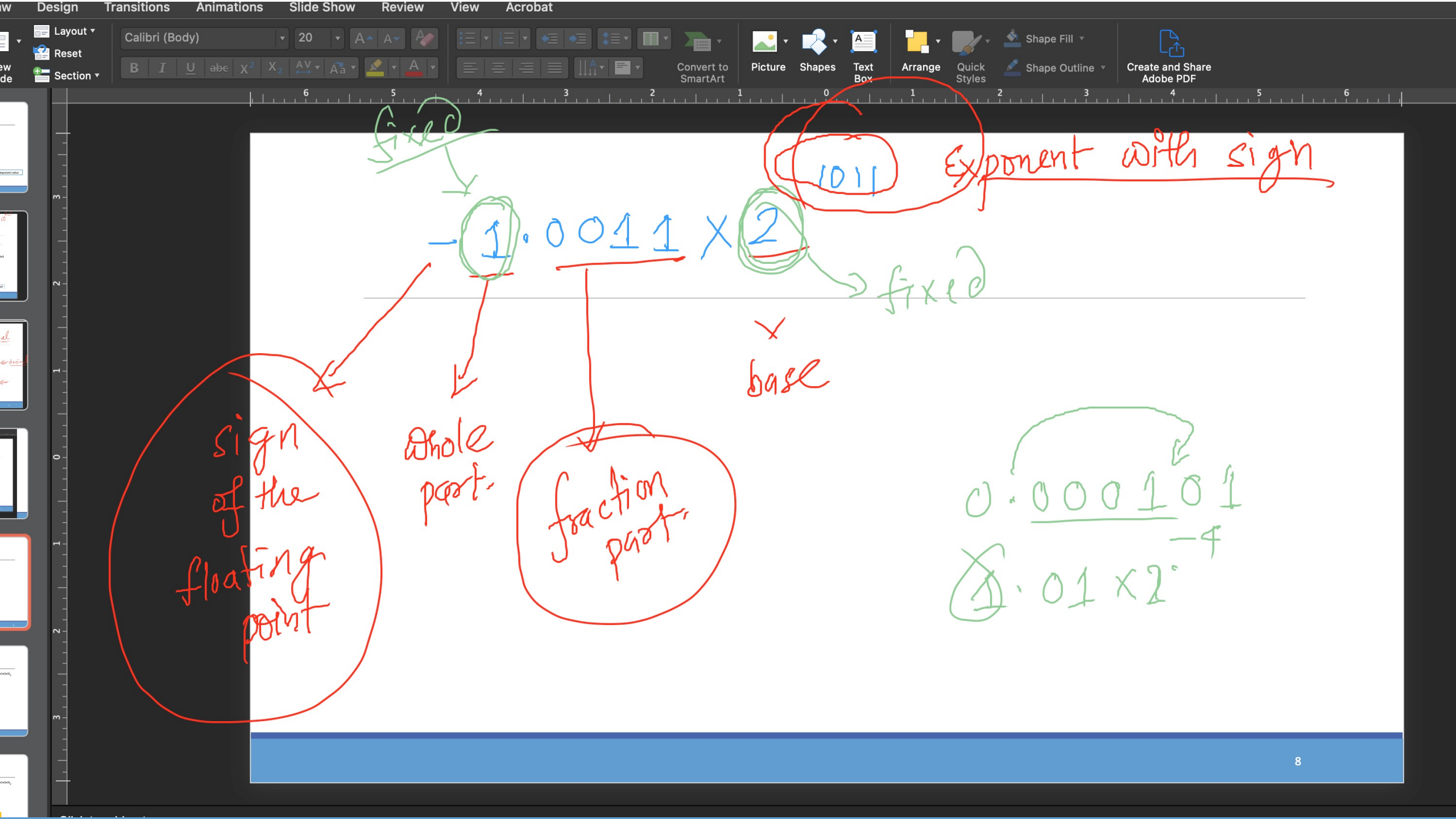
    $3.\ 6699 \text{ X } 10^{-1}$

How about 0.0063?


0.004673
4.673 X 10^-3
decimal


(10101.01101) bin
4 decimal
1.0101011011 X 2
1.0101011011 X 2^100


(0.0001010111) bin.
100
⇒ 1.01011 | × 2


Design
Transitions
Animations
Slide Show
Review
View
Acrobat
Layout
Reset
Section
Calibri (Body)
20
Convert to SmartArt
Picture
Shapes
Text Box
Arrange
Quick Styles
Shape Fill
Shape Outline
Create and Share Adobe PDF
fixed
1011
Exponent with sign
-1.0011 X 2
fixed
base
sign of the floating point
whole part
fraction part
0.000101
-4
1.01 x 2
8

Floating-Point Numbers

- **Normal Scientific notation for binary**: $\pm$ b. $f_1f_2f_3f_4$ … $\times 2^{\pm (e1e2e3)_2}$
  - **Example: 101.11**

    101.11 **(convert into normal scientific form)**

    1.0111×2^{10}

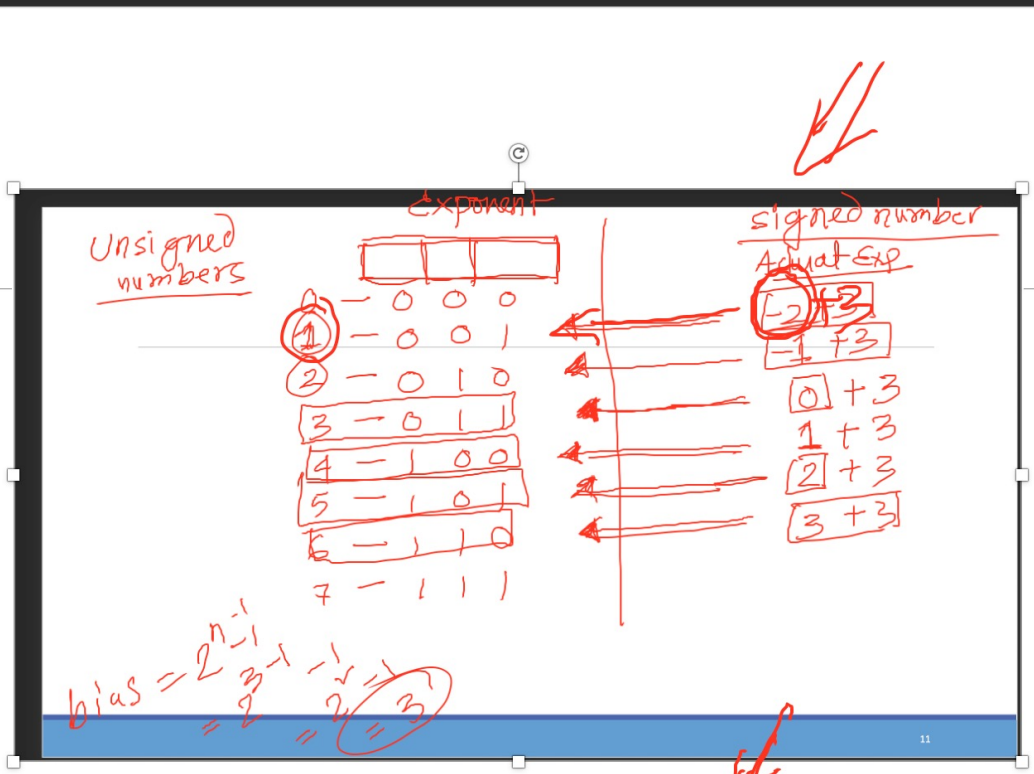
    - **How to convert 1.0111 into decimal?**

  - **How to store this number into a computer hardware?**

Floating-Point Numbers

- **Normal Scientific notation for binary**: $\pm$ b. $f_1f_2f_3f_4$ … $\times 2^{\pm (e1e2e3)_2}$
  - **How to store** 1.0111 X 2^{10} **into a computer hardware?**

$$\mathbf{1}.0111 \text{ X } 2^{10}$$



$-(1.110 \times 2^{-2})$

fraction

$exp = -2$

$bias = 2^{3-1}-1$

$= 3$

$exp = 1 \quad bias = 3 \quad = -2$

exp

1 | 0 0 1

sign

1 1 0 0

-2 fraction

$-(1.110 \times 2^{-2})$

11

Click to add notes

Floating-Point Numbers

- **How about storing 0.1 (**0.1×2^0) ?
  - convert into **normal scientific form** 1.0×2^{-1}
  - We **don't store the 1** before the . ? 0000
  - How to store the **negative exponent** (-1)?
    - No 2'com, then how?
    - **Use biasing method? $2^{n-1} - 1$, where n is the number of exponents bits**
    - **$2^{3-1} - 1 = 4 - 1 = 3$ (this is called biased/mantissa)**
    - **-1 + 3 = 2** 010
    - **Convert to binary and store the binary**

Floating-Point Binary Representation

- CONVERT A floating-point to a **binary**:

Value of a floating-point number = $(-1)^S \times \text{val}(F) \times 2^{\text{val}(E)}$

$(-1)^S \times (1.F)_2 \times 2^{\text{val}(E)}$

$(-1)^S \times (1.f_1f_2f_3f_4 \ldots)_2 \times 2^{\text{val}(E)}$

$(-1)^S \times (1 + f_1\times2^{-1} + f_2\times2^{-2} + f_3\times2^{-3} + f_4\times2^{-4} \ldots)_2 \times 2^{\text{val}(E)}$

$(-1)^S$ is 1 when S is 0 (positive), and –1 when S is 1 (negative)

Floating-Point Binary Representation

- A floating-point to a **binary**:

- **Sign** = 0 (+)
- **Exponent** = **saved Exponent** – **3**
- **Fraction** = **0010**
- $(-1)^{s}$ X $(1.\text{Signficant})_2$ X 2^{E-3}

- $(-1)^{0}$ X **1.001** X 2^{-1}
- **1 X 1.001 X 0.1 = .1001**

- **To decimal:**
  1 X 1.125 X 0.5 = 0.5625

$$(-1)^S \times (1.F)_2 \times 2^{\text{val}(E)}$$

$$(-1)^S \times (1.f_1f_2f_3f_4\ldots)_2 \times 2^{\text{val}(E)}$$

$$(-1)^S \times (1 + f_1\times 2^{-1} + f_2\times 2^{-2} + f_3\times 2^{-3} + f_4\times 2^{-4}\ldots)_2 \times 2^{\text{val}(E)}$$

$(-1)^S$ is 1 when S is 0 (positive), and –1 when S is 1 (negative)

Outline

- **Floating-Point Binary Representation**
- Floating-Point Unit
- x86 Instruction Encoding

Floating-Point Binary Representation

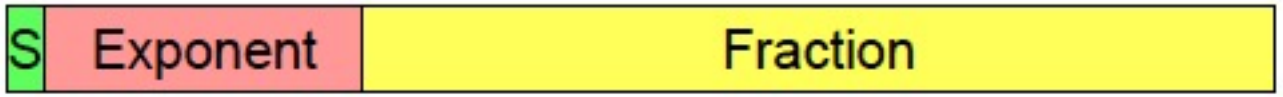
- There are **two fundamental types of microprocessor instructions**:
  - **integer** and **floating-point**.
- Accordingly, they are executed
  - On an Integer Processing Unit and
  - On a Floating-Point Processing Unit.
- But **what tells the processor to send an instruction** to IPU or FPU?
- How does it know **which instruction is which kind**?
  - Maybe an **instruction have a bit / flag / LUT** or something to differentiate one from another?

Floating-Point Binary Representation

- Each CPU instruction has an **opcode**.
- The CPU **looks at the opcode** to determine which **execution unit** the instruction should be dispatched to.
- For **floating point instructions** on x86 the **opcode** typically starts with:
  - **1101 1**.... i.e.
    - the first hex digit is **D**
    - and then the **MS bit** of the next digit is **set**.

Floating-Point Binary Representation

- A floating-point number is represented by the triple:



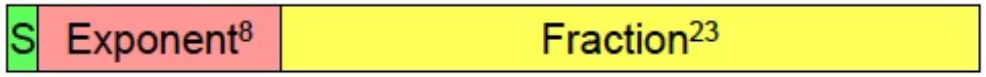
  - S is the Sign bit (0 is positive and 1 is negative)
    - Representation is called sign and magnitude
  - E is the Exponent field (signed)
    - Very **large numbers** **have large positive exponents**
    - Very **small close-to-zero numbers** **have negative exponents**
    - **More bits in exponent field** **increases** **range of values**
  - F is the Fraction field (fraction after binary point)
    - **More bits in fraction field** improves the **precision** of FP numbers

Floating-Point Binary Representation

- **IEEE Floating-Point Binary Reals**
- The Exponent
- Normalized Binary Floating-Point Numbers
- Creating the IEEE Representation
- Converting Decimal Fractions to Binary Reals

IEEE Floating-Point Binary Reals

- **Types**
  - **Single Precision**



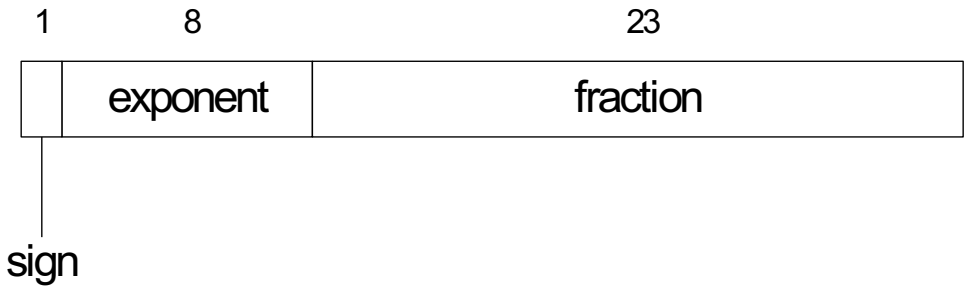
    - **32 bits**: **1 bit for the sign**, **8 bits for the exponent**, and **23 bits for the fractional** part of the significand.
  - **Double Precision**

    - **64 bits**: **1 bit for the sign**, **11 bits for the exponent**, and **52 bits for the fractional** part of the significand.
  - **Double Extended Precision**

    - **80 bits**: **1 bit for the sign**, **16 bits for the exponent**, and **63 bits for the fractional** part of the significand.

Single-Precision Format

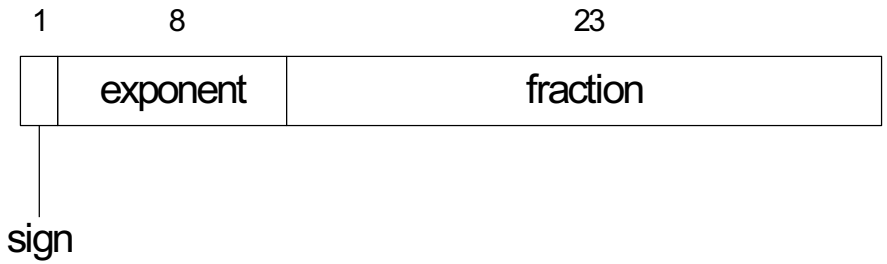
- Approximate normalized **range**: –126 to 127.
  - Also called a *short real*.



Components of a Single-Precision Real

- **Sign**
  - 1 = negative, 0 = positive
- **Significand**
  - decimal digits to the **left** & **right** of decimal point
  - **weighted** positional notation
  - **Example**:

    123.154 = $(1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) +$ **$(1 \times 10^{-1}) + (5 \times 10^{-2}) + (4 \times 10^{-3})$**
- **Exponent**
  - **unsigned** integer
  - integer **bias** (127 for single precision)



Components of a Single-Precision Real

- **Binary floating-point** numbers also **use weighted positional notation**.
- The **floating-point binary value 11.1011** is expressed as

$$11.1011 = (1 \times 2^{1}) + (1 \times 2^{0}) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4})$$

$$.1011 = 1/2 + 0/4 + 1/8 + 1/16 = 11/16 \qquad \textbf{0.6875}$$

Components of a Single-Precision Real

Table 12-2 Examples: Translating Binary Floating-Point to Fractions.

Binary Floating-Point	Base-10 Fraction
11.11	3 3/4
101.0011	5 3/16
1101.100101	13 37/64
0.00101	5/32
1.011	1 3/8
0.00000000000000000000001	1/8388608

Click to add notes

Components of a Single-Precision Real

Table 17-3 Binary and Decimal Fractions.

Binary	Decimal Fraction	Decimal Value
.1	1/2	.5
.01	1/4	.25
.001	1/8	.125
.0001	1/16	.0625
.00001	1/32	.03125

Converting Fractions to Binary Reals

- Express as a sum of **fractions having denominators that are powers of 2**
- **Examples:**

Decimal Fraction	Factored As...	Binary Real
1/2	1/2	.1
1/4	1/4	.01
3/4	1/2 + 1/4	.11
1/8	1/8	.001
7/8	1/2 + 1/4 + 1/8	.111
3/8	1/4 + 1/8	.011
1/16	1/16	.0001
3/16	1/8 + 1/16	.0011
5/16	1/4 + 1/16	.0101

The Exponent

- The exponent **range is from -126 to 127** (in **Single-Precision**)
- Sample Exponents represented in Binary
- **Add 127** to actual exponent to produce the **biased exponent**

Table 12-4 Sample Exponents Represented in Binary.

Exponent (E)	Biased (E + 127)	Binary
+5	132	10000100
0	127	01111111
−10	117	01110101
+127	254	11111110
−126	1	00000001
−1	126	01111110

Normalizing Binary Floating-Point Numbers

- **Mantissa** is normalized **when a single 1 appears to the left of the binary point**
- **Unnormalized**: shift binary point **until exponent is zero**
- **Examples**

Unnormalized	Normalized
1110.1	1.1101×2^{3}
.000101	1.01×2^{-4}
1010001.	1.010001×2^{6}

Real-Number Encodings

- **Normalized finite numbers**
  - all the nonzero finite values that can be encoded **in a normalized real number between zero and infinity**
- **Example**:

  binary **1.101 X 2^0** is represented as follows:

  - Sign bit: 0
  - Exponent: 01111111 **0 + 127**
  - Fraction: 10100000000000000000000 **23 bits: 10100000000000000000000**

0 01111111 10100000000000000000000

Real-Number Encodings

- **Example**:

Table 12-5 Examples of Single Precision Bit Encodings.

Binary Value	Biased Exponent	Sign, Exponent, Fraction
-1.11	127	1 01111111 11000000000000000000000
+1101.101	130	0 10000010 10110100000000000000000
-.00101	124	1 01111100 01000000000000000000000
+100111.0	132	0 10000100 00111000000000000000000
+.0000001101011	120	0 01111000 10101100000000000000000

Real-Number Encodings

- **Specific encodings** (single precision):

Value	Sign, Exponent, Significand
Positive zero	0 00000000 00000000000000000000000
Negative zero	1 00000000 00000000000000000000000
Positive infinity	0 11111111 00000000000000000000000
Negative infinity	1 11111111 00000000000000000000000
QNaN	x 11111111 1xxxxxxxxxxxxxxxxxxxxxx
SNaN	x 11111111 0xxxxxxxxxxxxxxxxxxxxxx[a]

[a] SNaN significand field begins with 0, but at least one of the remaining bits must be 1.

x can be either 1 or 0 (DO NOT CARE)

Converting Single-Precision to Decimal

1. If the **MSB** is 1, the **number is negative**; otherwise, it is **positive**.

2. The **next 8 bits** represent the **exponent**.

   - **Subtract** binary 01111111 (decimal 127), producing the unbiased exponent.
   - **Convert** the unbiased exponent to decimal.

Converting Single-Precision to Decimal

3. The **next 23 bits** represent the **significand**.
   - **Notate a "1.", followed by the significand bits**.
   - **Trailing zeros** can be ignored.
   - Create a floating-point binary number,
     - using the significand,
     - the sign determined in step 1,
     - and the exponent calculated in step 2.

Converting Single-Precision to Decimal

4. **Unnormalize** the binary number produced in step 3.
   - Shift the binary point the number of places equal to the value of the exponent.
   - Shift right if the exponent is positive, or left if the exponent is negative.

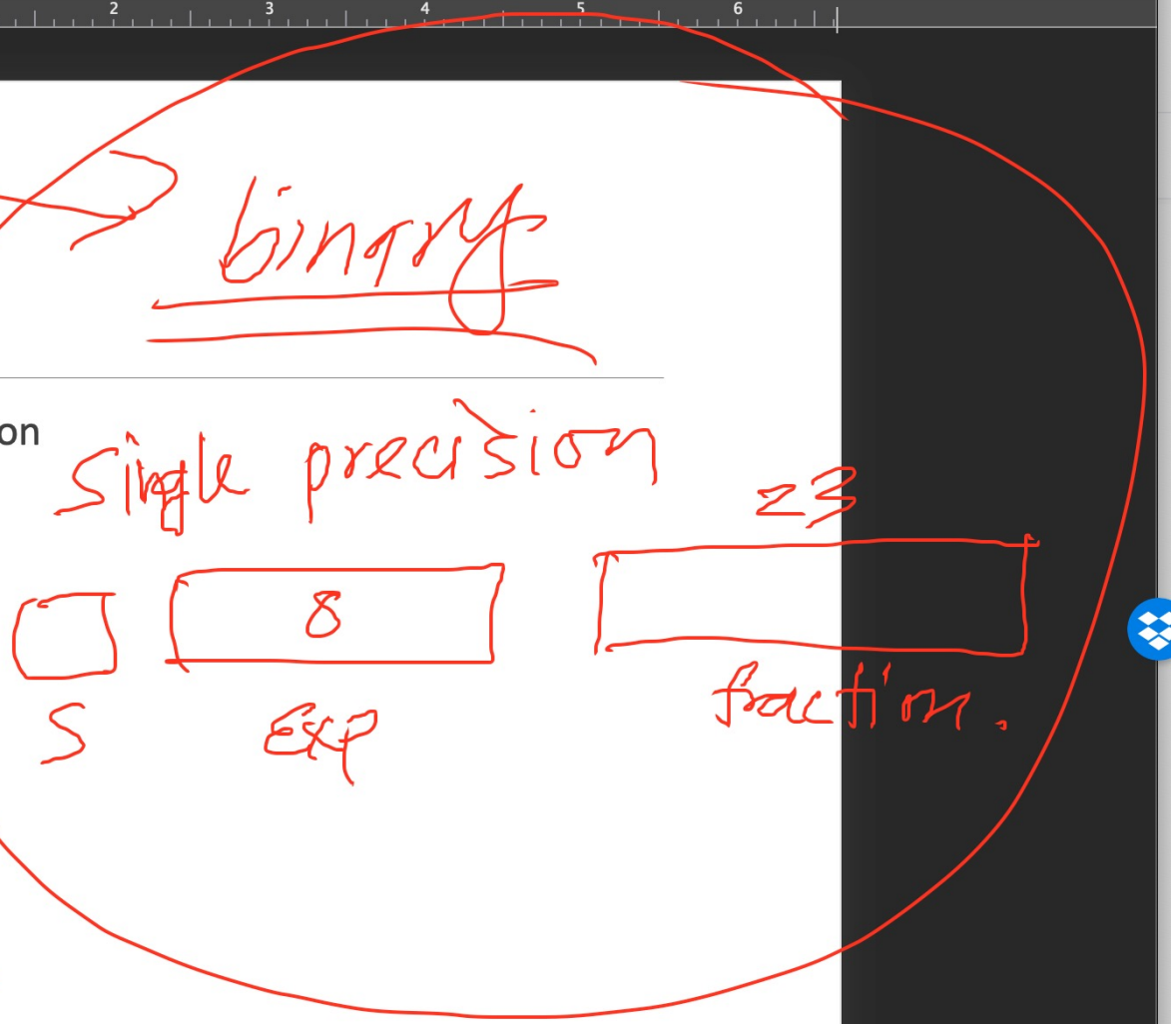
5. From **left to right**,
   - use weighted positional notation to **form the decimal sum of the powers of 2** represented by the floating-point binary number.

Example2

- Convert - **19.3** to IEEE Floating-Point Binary Representation

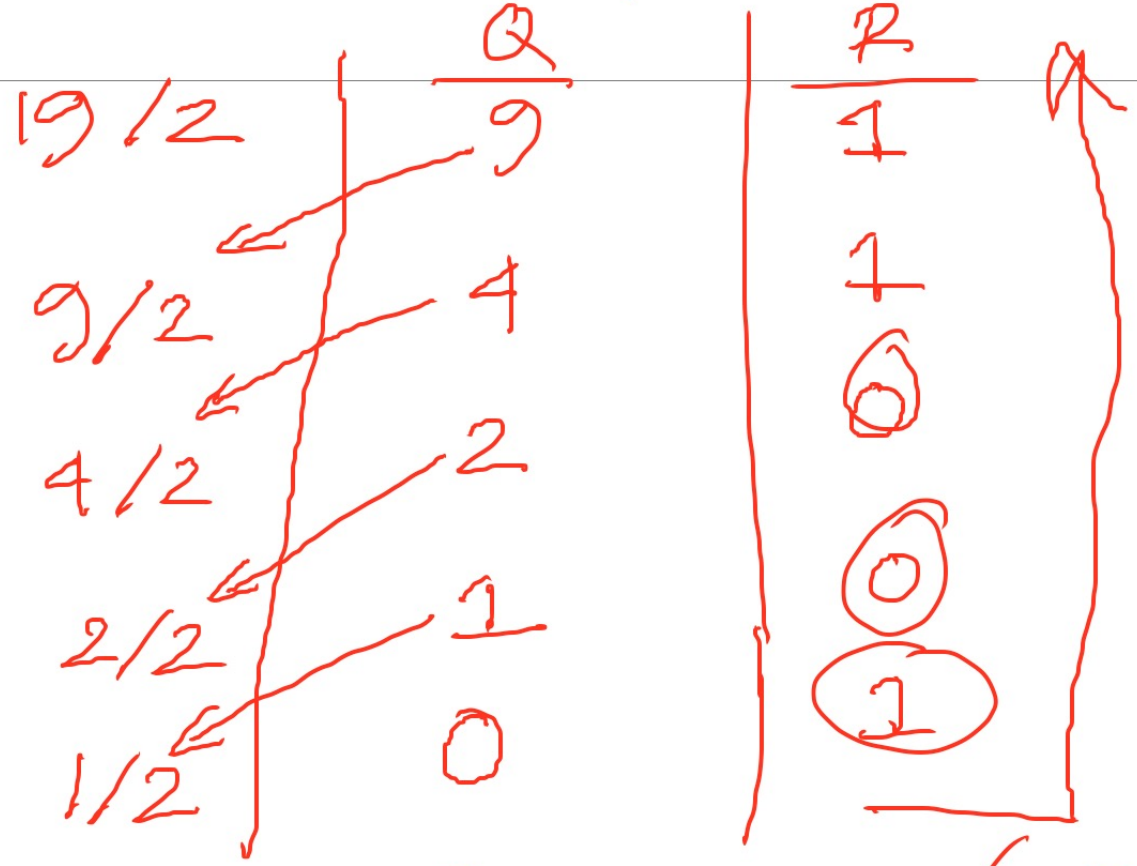
1. Convert - **19.3** into binary
2. Normalize the converted binary
3. Find the exponent and store it (add it to the Biased)
4. **Write the normalized number in IEEE format**

Sign	Exponent	Fraction
?	?	?

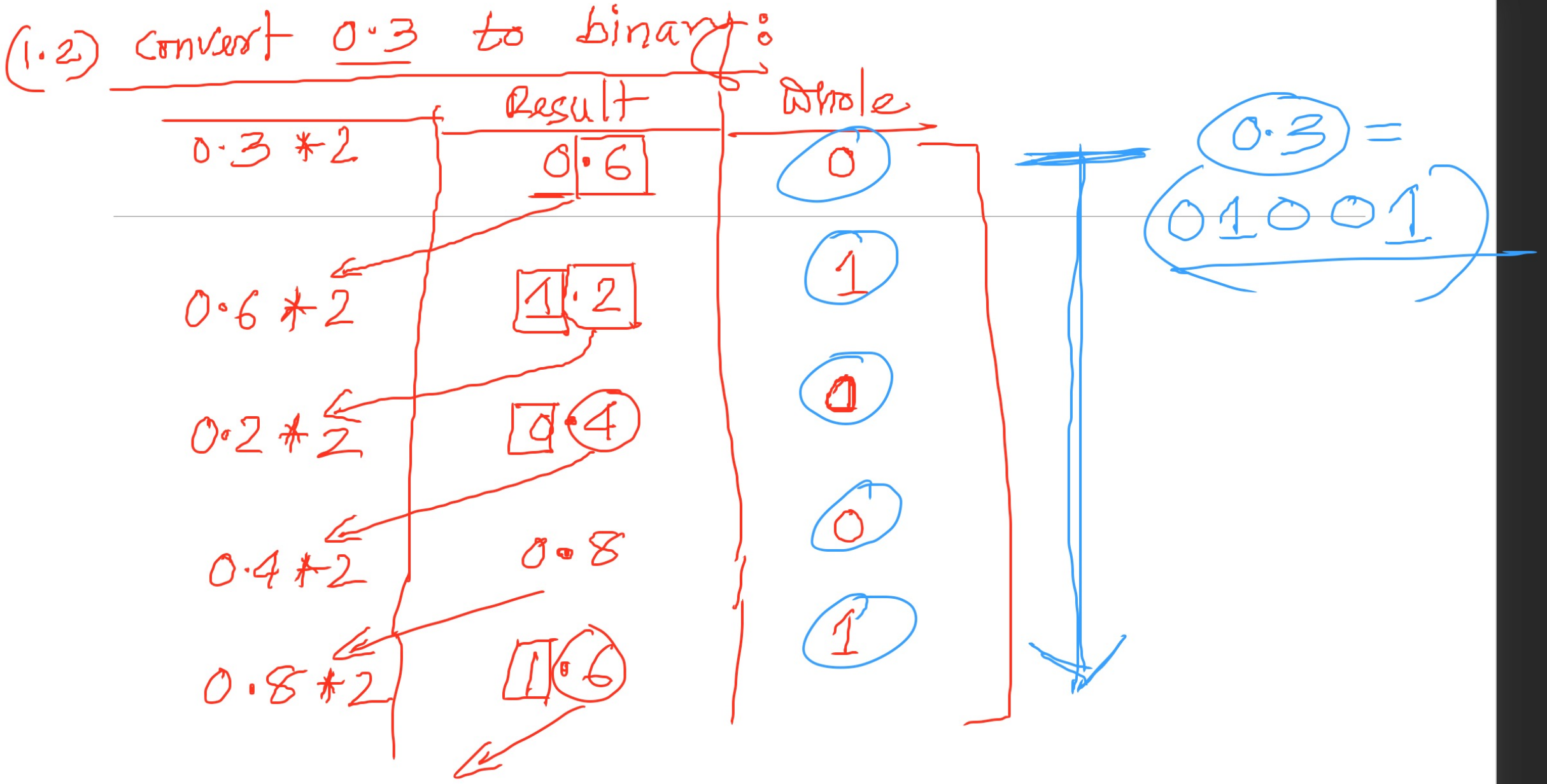


-19.3

(1) Convert the whole part into binary:



$(19)_{dec} = (10011)_{bin}$.


(1.2) convert 0.3 to binary:
Result
Whole
0.3 *2
0.6
0
0.6 *2
1.2
1
0.2 *2
0.4
0
0.4 *2
0.8
0
0.8 *2
1.6
1
0.3 =
0 1 0 0 1

sign exp

0 | 1000001 | 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

-ve

19.3 ⇒ (10011.01001) bin.

(2) Convert floating point to scientific notation.

$1.0011010001 \times 2^{4}$

(3)

Actual Exponent = 4

bias = $2^{8-1}-1 = 127$.

When you store exponent, add bias to it

exp + bias = 4 + 127 = (131) ⇒ bin = 1000011

Example1

- **Convert 01000001001011000000000000000000 to Decimal**

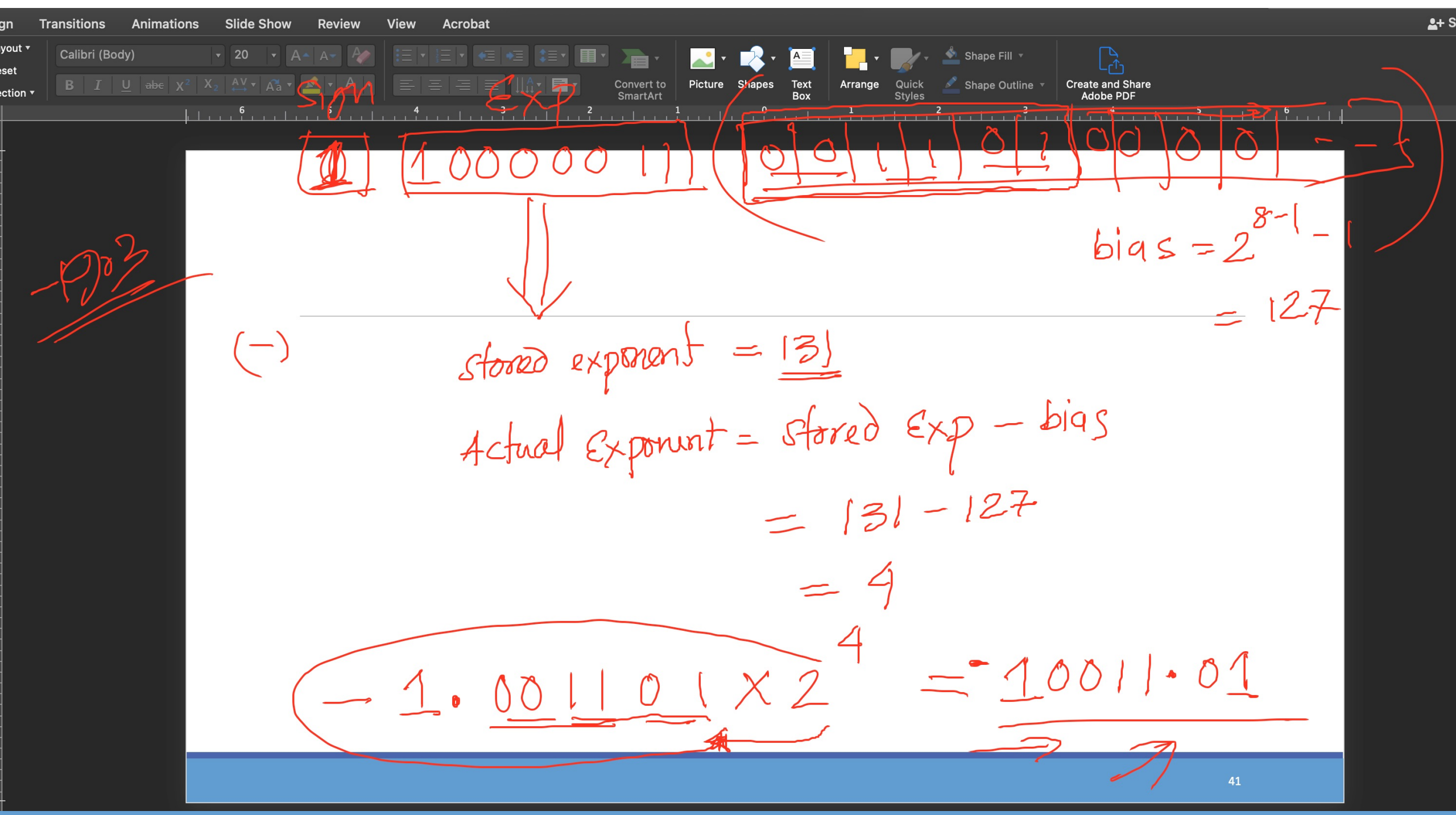
1. The number is **positive**.

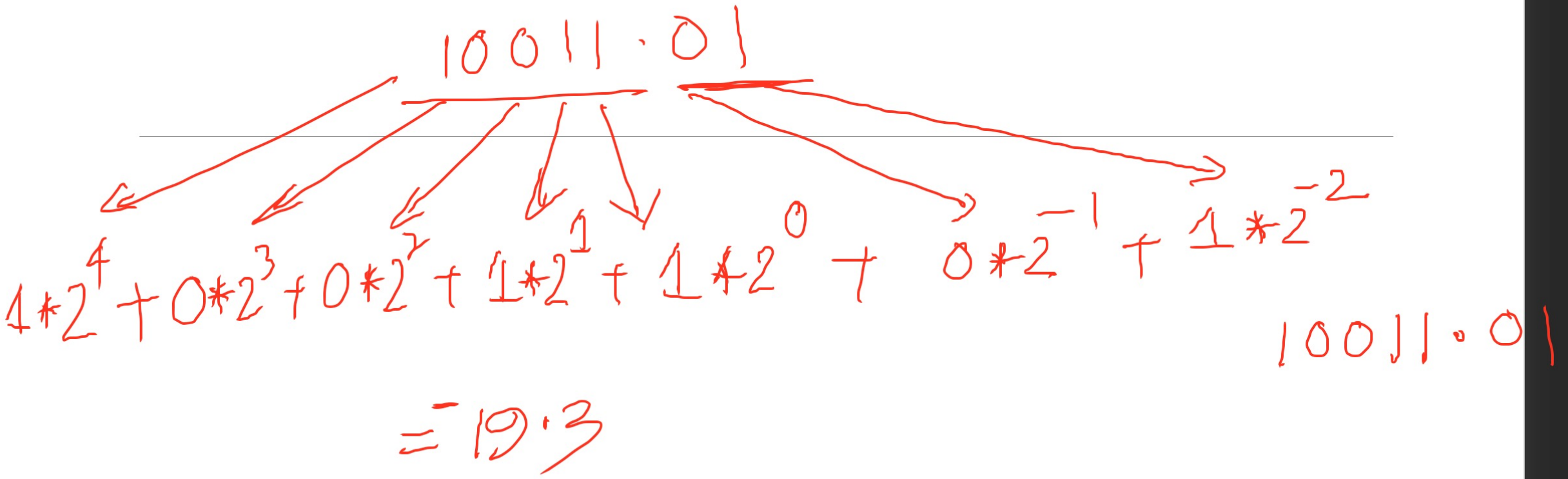
2. The **unbiased exponent** is binary **00000011**, or decimal 3 (130 (**10000010**) – 127 (**01111111**) = 3).

3. **Combining** the **sign**, **exponent**, and **significand**, the binary number is **+1.01011 X 2^3.**

4. The **unnormalized** binary number is **+1010.11**

5. The **decimal value** is **+10 3/4**, or +**10.75**.


Calibri (Body)
20
Convert to SmartArt
Picture
Shapes
Text Box
Arrange
Quick Styles
Shape Fill
Shape Outline
Create and Share Adobe PDF
sign
exp
1 10000 111
0 1 0 1 1 1 0 1 0 0 0 0 0 1 - -
bias = 2^(8-1) - 1
= 127
-ve
(-)
stored exponent = 131
Actual Exponent = stored exp - bias
= 131 - 127
= 4
- 1.00110 1 X 2^4
= -10011.01
41



=19.3

Outline

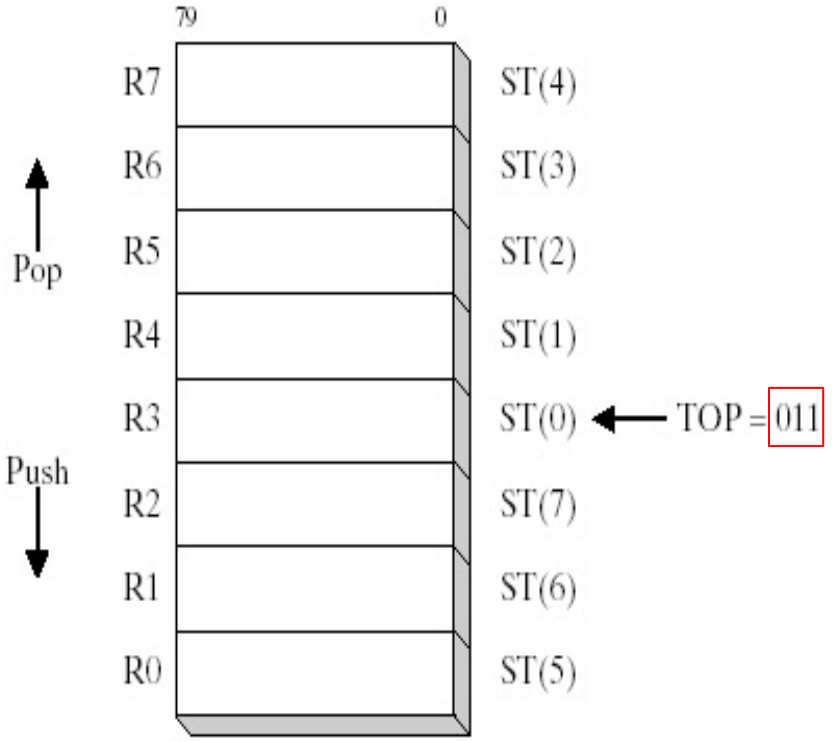
- Floating-Point Binary Representation
- **Floating-Point Unit**
- x86 Instruction Encoding

Floating Point Unit

- **FPU Register Stack**
- Rounding
- Floating-Point Exceptions
- Floating-Point Instruction Set
- Arithmetic Instructions
- Mixed-Mode Arithmetic
- Comparing Floating-Point Values
- Reading and Writing Floating-Point Values

FPU Register Stack

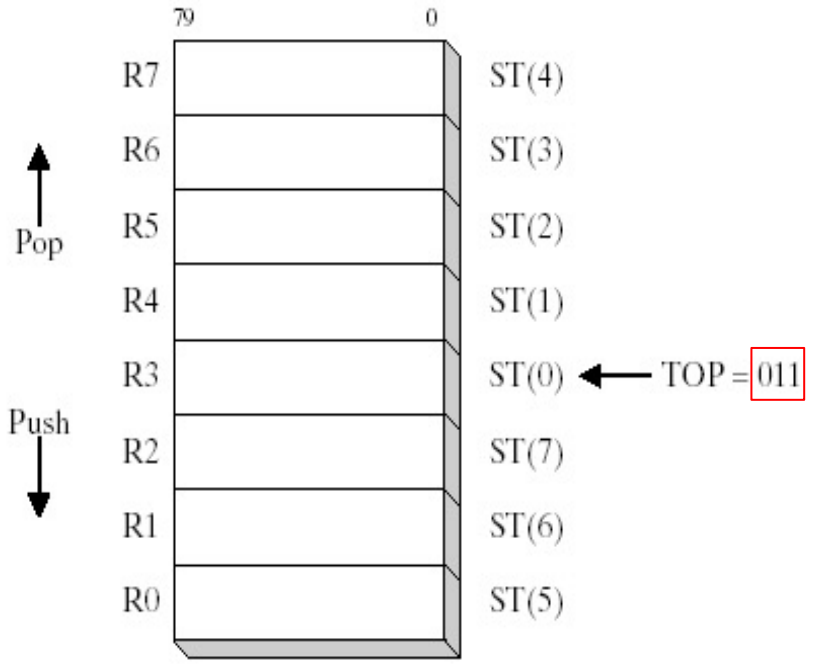
- The **FPU does not use the general-purpose registers** (EAX, EBX, etc.).
- Instead, it has **its own set of registers** called a **register stack**.
  - **Eight individually addressable 80-bit data registers**
    - named **R0 through R7**
  - **Three-bit field** named **TOP** in the **FPU status** word identifies
    - **The register number** that is **currently the top of stack**.



FPU Register Stack

- **FPU**:
  - **Loads** values **from memory** into **the register stack**,
  - **Performs calculations**, and
  - **Stores stack values** into **memory**
- **FPU instructions** evaluate mathematical **expressions** in **postfix format**

  - **infix** expression: **(5 * 6) - 4**
  - **postfix** equivalent is: **5 6 * 4 -**



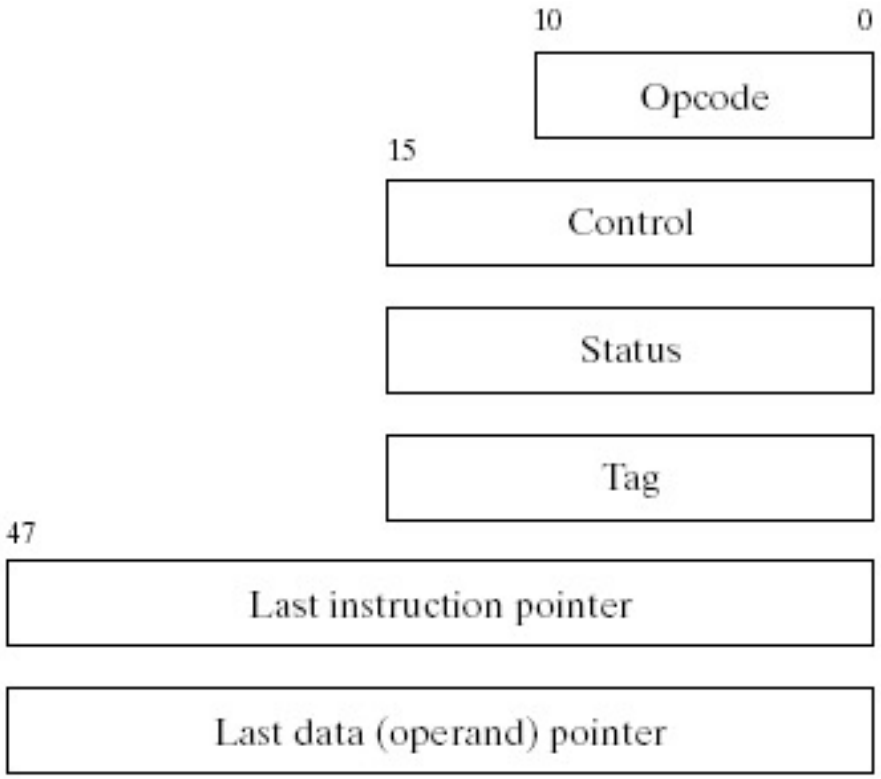
FPU Register Stack

- Translating **infix** expressions to **postfix**

Table 12-8 Infix to Postfix Examples.

Infix	Postfix
A + B	A B +
(A - B) / D	A B - D /
(A + B) * (C + D)	A B + C D + *
((A + B) / C) * (E - F)	A B + C / E F - *

Special-Purpose Registers



Floating Point Unit

- FPU Register Stack
- **Rounding**
- Floating-Point Exceptions
- Floating-Point Instruction Set
- Arithmetic Instructions
- Mixed-Mode Arithmetic
- Comparing Floating-Point Values
- Reading and Writing Floating-Point Values

Rounding

- **FPU** attempts to round an **infinitely** accurate result from a floating-point calculation
  - may be impossible because of **storage limitations**
- **Example**
  - suppose **3 fractional bits can be stored**, and a calculated value equals **+1.0111**.
    - **rounding up** by **adding** .0001 produces 1.100
    - **rounding down** by **subtracting** .0001 produces 1.011

Rounding

- **Examples**

TABLE 12-9 Example: Rounding +1.0111.

Method	Precise Result	Rounded
Round to nearest even	1.0111	1.100
Round down toward $-\infty$	1.0111	1.011
Round toward $+\infty$	1.0111	1.100
Round toward zero	1.0111	1.011

TABLE 12-10 Example: Rounding –1.0111.

Method	Precise Result	Rounded
Round to nearest (even)	-1.0111	-1.100
Round toward $-\infty$	-1.0111	-1.100
Round toward $+\infty$	-1.0111	-1.011
Round toward zero	-1.0111	-1.011

Floating Point Unit

- FPU Register Stack
- Rounding
- **Floating-Point Exceptions**
- Floating-Point Instruction Set
- Arithmetic Instructions
- Mixed-Mode Arithmetic
- Comparing Floating-Point Values
- Reading and Writing Floating-Point Values

Floating-Point Exceptions

- **Six types** of exception conditions
  - **Invalid operation**
  - **Divide by zero**
  - **Denormalized operand**
  - **Numeric overflow**
  - **Numeric underflow**
  - **Inexact precision**
- Each has a corresponding ***mask*** bit
  - if **set** when an exception occurs, **the exception is handled automatically by FPU**
  - if **clear** when an exception occurs, a **software exception handler is invoked**

Floating Point Unit

- FPU Register Stack
- Rounding
- Floating-Point Exceptions
- **Floating-Point Instruction Set**
- Arithmetic Instructions
- Mixed-Mode Arithmetic
- Comparing Floating-Point Values
- Reading and Writing Floating-Point Values

FPU Instruction Set

- Instruction mnemonics **begin with letter F**
- **Second letter identifies data type of memory operand**
  - **FLBD** load **binary coded decimal:** B = bcd
  - **FISTP** store **integer** and **pop stack:** I = integer
  - **FMUL** multiply **floating-point** operands **no letter**: floating point

FPU Instruction Set

- **Operands**
  - **zero**, **one**, or **two**
  - **no immediate operands**
  - **no general-purpose registers** (EAX, EBX, ...)
  - **integers** must be (**before being used in calculations**)
    - **loaded** from memory onto the stack and
    - **converted** to floating-point
  - if an instruction **has two operands**,
    - **one must be a FPU register**

FPU Instruction Set

- **Data Types**

Table 17-11 Intrinsic Data Types.

Type	Usage
QWORD	64-bit integer
TBYTE	80-bit (10-byte) integer
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

Load Floating-Point Value

- **FLD**
  - **copies** floating point **operand from memory** into **the top of the FPU stack, ST(0)**

```
FLD m32fp
FLD m64fp
FLD m80fp
FLD ST(i)
```

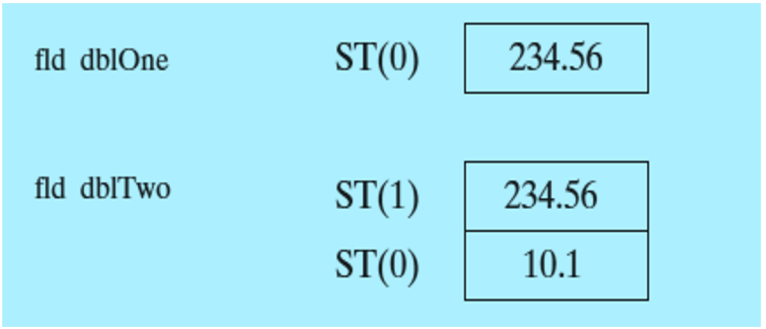
i is a register number

- **Example**

```
.data
dblOne   REAL8 234.56
dblTwo   REAL8 10.1
.code
fld  dblOne
fld  dblTwo
```

```
; ST(0) = dblOne
; ST(0) = dblTwo, ST(1) = dblOne
```

FPU stack



Store Floating-Point Value

- **FST**
  - **copies** floating point operand **from the top of the FPU stack into memory**
  - **does not pop the top of the stack**

```
FST m32fp
FST m64fp
FST ST(i)
```

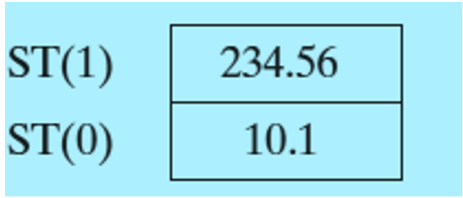
- **Example:**

Assume ST(0) equals **10.1** and ST(1) equals **234.56**:

```
fst   dblThree ; 10.1
fst   dblFour  ; 10.1
```

expected **dblFour** to equal **234.56**

FPU stack



Store Floating-Point Value

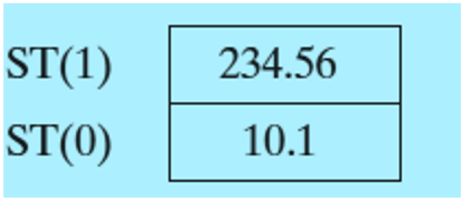
- **FSTP** (copies floating point operand **from the top of the FPU stack into memory**)
  - **pops** the stack after copying

```
FST  m32fp
FST  m64fp
FST  ST(i)
```

- **Example:**

  Assume ST(0) equals **10.1** and ST(1) equals **234.56**:

```
fstp   dblThree    ; 10.1
fstp   dblFour     ; 234.56
```



Floating Point Unit

- FPU Register Stack
- Rounding
- Floating-Point Exceptions
- Floating-Point Instruction Set
- **Arithmetic Instructions**
- Mixed-Mode Arithmetic
- Comparing Floating-Point Values
- Reading and Writing Floating-Point Values

Arithmetic Instructions

- Same **operand types** as **FLD** and **FST**

Table 12-12 Basic Floating-Point Arithmetic Instructions.

FCHS	Change sign
FADD	Add source to destination
FSUB	Subtract source from destination
FSUBR	Subtract destination from source
FMUL	Multiply source by destination
FDIV	Divide destination by source
FDIVR	Divide source by destination

Floating-Point Add

- **FADD**
  - adds **source** to **destination**
    1) **No-operand version pops the FPU stack after adding**

- **Example**:

```
FADD[4]
FADD  m32fp
FADD  m64fp
FADD  ST(0), ST(i)
FADD  ST(i), ST(0)
```

i **is a register number**

fadd	Before:	ST(1)	234.56
		ST(0)	10.1
	After:	ST(0)	244.66

Destination is always FPU Register

Floating-Point Add

- **FADD**
  - adds **source** to **destination**
    - **2) Register Operands**

```
FADD[4]
FADD  m32fp
FADD  m64fp
FADD  ST(0), ST(i)
FADD  ST(i), ST(0)
```

- **Example**:

```
fadd st(1), st(0)
```

Before:	ST(1)	234.56
	ST(0)	10.1
After:	ST(1)	244.66
	ST(0)	10.1

Destination is always FPU Register

Floating-Point Add

- **FADD**
  - adds **source** to **destination**
    3) **Memory Operands:** **FADD** adds the operand to ST(0).

```
FADD[4]
FADD  m32fp
FADD  m64fp
FADD  ST(0), ST(i)
FADD  ST(i), ST(0)
```

- **Example**:

```
fadd  mySingle              ; ST(0) += mySingle
fadd  REAL8 PTR[esi]        ; ST(0) += [esi]
```

Destination is always FPU Register

Floating-Point Subtract

- **FSUB**
  - subtracts **source** from **destination**.
  - **No-operand** version **pops the FPU stack after** subtracting

```
FSUB[5]
FSUB m32fp
FSUB m64fp
FSUB ST(0), ST(i)
FSUB ST(i), ST(0)
```

- **Example:**

```
fsub mySingle                ; ST(0) -= mySingle
fsub array[edi*8]            ; ST(0) -= array[edi*8]
```

Destination is always FPU Register

Floating-Point Multiply

- **FMUL**
  - **Multiplies source by destination**, stores product in destination

```
FMUL[6]
FMUL  m32fp
FMUL  m64fp
FMUL  ST(0), ST(i)
FMUL  ST(i), ST(0)
```

```
fmul  mySingle              ; ST(0) *= mySingle
```

The no-operand versions of FMUL and FDIV pop the stack after multiplying or dividing.

Destination is always FPU Register

Example for: Fld, Fstp Fadd, Fsub, fmul

- E = (A + B) * (C - D)

  A=1.9, B=2.1, C=5.3, D=0.3, E=?

Floating-Point Divide

- **FDIV**
  - **Divides destination** by **source**, then pops the stack

```
FDIV[7]
FDIV m32fp
FDIV m64fp
FDIV ST(0), ST(i)
FDIV ST(i), ST(0)
```

The no-operand versions of FMUL and FDIV pop the stack after multiplying or dividing.

```
.data
dblOne    REAL8   1234.56

  dblTwo    REAL8   10.0
  dblQuot   REAL8   ?
  .code
  fld   dblOne                  ; load into ST(0)
  fdiv  dblTwo                  ; divide ST(0) by dblTwo
  fstp  dblQuot                 ; store ST(0) to dblQuot
```

Destination is always FPU Register

FPU Code Example

```
.data
valA REAL8 1.5
valB REAL8 2.5
valC REAL8 3.0
valD REAL8 ?              ; will be +6.0

.code
fld valA                  ; ST(0) = valA
fchs                      ; change sign of ST(0)
fld  valB                 ; load valB into ST(0)
fmul valC                 ; ST(0) *= valC
fadd                      ; ST(0) += ST(1)
fstp valD                 ; store ST(0) to valD
```

valD = –valA + (valB * valC).