Floating Point

COE 301

Computer Organization

Prof. Muhamed Mudawar

College of Computer Sciences and Engineering

King Fahd University of Petroleum and Minerals

Presentation Outline

- ❖ Floating-Point Numbers
- ❖ IEEE 754 Floating-Point Standard
- ❖ Floating-Point Addition and Subtraction
- ❖ Floating-Point Multiplication
- ❖ MIPS Floating-Point Instructions

The World is Not Just Integers

- Programming languages support numbers with fraction
  - Called floating-point numbers
  - Examples:

    3.14159265… (π)

    2.71828… (e)

    0.000000001 or 1.0×10^{-9} (seconds in a nanosecond)

    86,400,000,000,000 or 8.64×10^{13} (nanoseconds in a day)

    last number is a large integer that cannot fit in a 32-bit integer
- We use a scientific notation to represent
  - Very small numbers (e.g. 1.0×10^{-9})
  - Very large numbers (e.g. 8.64×10^{13})
  - Scientific notation: $\pm\, d\,.\,f_1 f_2 f_3 f_4 \ldots \times 10^{\pm e_1 e_2 e_3}$

Floating-Point Numbers

- Examples of floating-point numbers in base 10 …
  - 5.341×10^{3} , 0.05341×10^{5} , -2.013×10^{-1} , -201.3×10^{-3} ← decimal point
- Examples of floating-point numbers in base 2 …
  - 1.00101×2^{23} , 0.0100101×2^{25} , -1.101101×2^{-3} , -1101.101×2^{-6} ← binary point
  - Exponents are kept in decimal for clarity
  - The binary number $(1101.101)_2 = 2^3+2^2+2^0+2^{-1}+2^{-3} = 13.625$
- Floating-point numbers should be normalized
  - Exactly one non-zero digit should appear before the point
    - In a decimal number, this digit can be from 1 to 9
    - In a binary number, this digit should be 1
  - Normalized FP Numbers: 5.341×10^{3} and -1.101101×2^{-3}
  - NOT Normalized: 0.05341×10^{5} and -1101.101×2^{-6}

Floating-Point 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

S	Exponent	Fraction

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

Next . . .

- Floating-Point Numbers
- IEEE 754 Floating-Point Standard
- Floating-Point Addition and Subtraction
- Floating-Point Multiplication
- MIPS Floating-Point Instructions

IEEE 754 Floating-Point Standard

- Found in virtually every computer invented since 1980
  - Simplified porting of floating-point numbers
  - Unified the development of floating-point algorithms
  - Increased the accuracy of floating-point numbers
- Single Precision Floating Point Numbers (32 bits)
  - 1-bit sign + 8-bit exponent + 23-bit fraction

S	Exponent^8	Fraction^{23}

- Double Precision Floating Point Numbers (64 bits)
  - 1-bit sign + 11-bit exponent + 52-bit fraction

S	Exponent^{11}	Fraction^{52}
(continued)		

Normalized Floating Point Numbers

- For a normalized floating point number (S, E, F)

S	E	F = $f_1\ f_2\ f_3\ f_4 \ldots$

- Significand is equal to $(1.F)_2 = (1.f_1f_2f_3f_4\ldots)_2$
  - IEEE 754 assumes hidden 1. (not stored) for normalized numbers
  - Significand is 1 bit longer than fraction
- Value of a Normalized Floating Point Number is

$$(-1)^S \times (1.F)_2 \times 2^{\text{val}(E)}$$
$$(-1)^S \times (1.f_1f_2f_3f_4 \ldots)_2 \times 2^{\text{val}(E)}$$
$$(-1)^S \times (1 + f_1\times2^{-1} + f_2\times2^{-2} + f_3\times2^{-3} + f_4\times2^{-4} \ldots)_2 \times 2^{\text{val}(E)}$$

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

Biased Exponent Representation

- How to represent a signed exponent? Choices are …
  - Sign + magnitude representation for the exponent
  - Two's complement representation
  - Biased representation
- IEEE 754 uses biased representation for the exponent
  - Value of exponent = val(E) = E – Bias (Bias is a constant)
- Recall that exponent field is 8 bits for single precision
  - E can be in the range 0 to 255
  - E = 0 and E = 255 are reserved for special use (discussed later)
  - E = 1 to 254 are used for normalized floating point numbers
  - Bias = 127 (half of 254), val(E) = E – 127
  - val(E=1) = –126, val(E=127) = 0, val(E=254) = 127

Biased Exponent – Cont'd

- For double precision, exponent field is 11 bits
  - E can be in the range 0 to 2047
  - E = 0 and E = 2047 are reserved for special use
  - E = 1 to 2046 are used for normalized floating point numbers
  - Bias = 1023 (half of 2046), val(E) = E – 1023
  - val(E=1) = –1022, val(E=1023) = 0, val(E=2046) = 1023
- Value of a Normalized Floating Point Number is

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

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

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

Examples of Single Precision Float

❖ What is the decimal value of this Single Precision float?

10111110001000000000000000000000

❖ Solution:

- ✧ Sign = 1 is negative
- ✧ Exponent = $(01111100)_2$ = 124, *E* – bias = 124 – 127 = –3
- ✧ Significand = $(1.0100 \ldots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)
- ✧ Value in decimal = $-1.25 \times 2^{-3} = -0.15625$

❖ What is the decimal value of?

01000001001001100000000000000000

❖ Solution:

implicit

- ✧ Value in decimal = $+(1.01001100 \ldots 0)_2 \times 2^{130-127} =$ $(1.01001100 \ldots 0)_2 \times 2^3 = (1010.01100 \ldots 0)_2 =$ 10.375

Examples of Double Precision Float

❖ What is the decimal value of this Double Precision float ?

01000000010100101010000000000000
00000000000000000000000000000000

❖ Solution:

- ✧ Value of exponent = $(10000000101)_2$ – Bias = 1029 – 1023 = 6
- ✧ Value of double float = $(1.00101010 \ldots 0)_2 \times 2^6$ (1. is implicit) = $(1001010.10 \ldots 0)_2$ = 74.5

❖ What is the decimal value of ?

10111111100010000000000000000000
00000000000000000000000000000000

❖ Do it yourself! (answer should be $-1.5 \times 2^{-7} = -0.01171875$)

Converting FP Decimal to Binary

- Convert –0.8125 to binary in single and double precision
- Solution:
  - Fraction bits can be obtained using multiplication by 2
    - 0.8125 × 2 = 1.625
    - 0.625 × 2 = 1.25
    - 0.25 × 2 = 0.5
    - 0.5 × 2 = 1.0
    - Stop when fractional part is 0

  $\boxed{0.8125 = (0.1101)_2 = ½ + ¼ + 1/16 = 13/16}$

  - Fraction = $(0.1101)_2$ = $(1.101)_2 \times 2^{-1}$ (Normalized)
  - Exponent = –1 + Bias = 126 (single precision) and 1022 (double)

1	01111110	10100000000000000000000	Single Precision
1	01111111110	10100000000000000000	Double Precision
		00000000000000000000000000000000	

Largest Normalized Float

- What is the Largest normalized float?
- Solution for Single Precision:

0	11111110	11111111111111111111111

  - Exponent – bias = 254 – 127 = 127 (largest exponent for SP)
  - Significand = $(1.111 \ldots 1)_2$ = almost 2
  - Value in decimal ≈ $2 \times 2^{127} \approx 2^{128} \approx 3.4028 \ldots \times 10^{38}$
- Solution for Double Precision:

0	11111111110	11111111111111111111
		11111111111111111111111111111111

  - Value in decimal ≈ $2 \times 2^{1023} \approx 2^{1024} \approx 1.79769 \ldots \times 10^{308}$
- Overflow: exponent is too large to fit in the exponent field

Smallest Normalized Float

- What is the smallest (in absolute value) normalized float?
- Solution for Single Precision:

  `0 00000001 00000000000000000000000`

  - Exponent – bias = 1 – 127 = –126 (smallest exponent for SP)
  - Significand = $(1.000 \ldots 0)_2 = 1$
  - Value in decimal = $1 \times 2^{-126} = 1.17549 \ldots \times 10^{-38}$
- Solution for Double Precision:

  `0 00000000001 00000000000000000000`
  `00000000000000000000000000000000`

  - Value in decimal = $1 \times 2^{-1022} = 2.22507 \ldots \times 10^{-308}$
- Underflow: exponent is too small to fit in exponent field

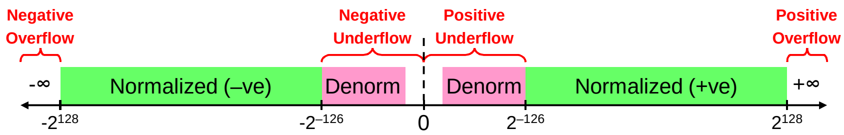
Zero, Infinity, and NaN

- Zero
  - Exponent field $E = 0$ and fraction $F = 0$
  - +0 and –0 are possible according to sign bit S
- Infinity
  - Infinity is a special value represented with maximum E and $F = 0$
    - For single precision with 8-bit exponent: maximum $E = 255$
    - For double precision with 11-bit exponent: maximum $E = 2047$
  - Infinity can result from overflow or division by zero
  - $+\infty$ and $-\infty$ are possible according to sign bit S
- NaN (Not a Number)
  - NaN is a special value represented with maximum E and $F \neq 0$
  - Result from exceptional situations, such as 0/0 or sqrt(negative)
  - Operation on a NaN results is NaN: Op(X, NaN) = NaN

Denormalized Numbers

- IEEE standard uses denormalized numbers to …
  - Fill the gap between 0 and the smallest normalized float
  - Provide gradual underflow to zero
- Denormalized: exponent field E is 0 and fraction $F \neq 0$
  - Implicit **1.** before the fraction now becomes **0.** (not normalized)
- Value of denormalized number (S, 0, F)

$$\boxed{\begin{array}{ll}\text{Single precision:} & (-1)^S \times (0.F)_2 \times 2^{-126} \\ \text{Double precision:} & (-1)^S \times (0.F)_2 \times 2^{-1022}\end{array}}$$



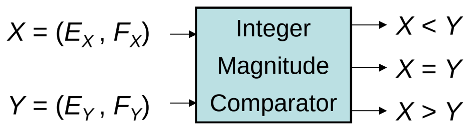
Summary of IEEE 754 Encoding

Single-Precision	Exponent = 8	Fraction = 23	Value
Normalized Number	1 to 254	Anything	$\pm (1.F)_2 \times 2^{E-127}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-126}$
Zero	0	0	± 0
Infinity	255	0	$\pm \infty$
NaN	255	nonzero	NaN

Double-Precision	Exponent = 11	Fraction = 52	Value
Normalized Number	1 to 2046	Anything	$\pm (1.F)_2 \times 2^{E-1023}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-1022}$
Zero	0	0	± 0
Infinity	2047	0	$\pm \infty$
NaN	2047	nonzero	NaN

Floating-Point Comparison

- ❖ IEEE 754 floating point numbers are ordered
  - ✧ Because exponent uses a biased representation …
    - Exponent value and its binary representation have same ordering
  - ✧ Placing exponent before the fraction field orders the magnitude
    - Larger exponent ⇒ larger magnitude
    - For equal exponents, Larger fraction ⇒ larger magnitude
    - $0 < (0.F)_2 \times 2^{E_{min}} < (1.F)_2 \times 2^{E-Bias} < \infty$ ($E_{\text{min}} = 1 - Bias$)
  - ✧ Because sign bit is most significant ⇒ quick test of signed <
- ❖ Integer comparator can compare magnitudes



Next . . .

- ❖ Floating-Point Numbers
- ❖ IEEE 754 Floating-Point Standard
- ❖ Floating-Point Addition and Subtraction
- ❖ Floating-Point Multiplication
- ❖ MIPS Floating-Point Instructions

Floating Point Addition Example

❖ Consider Adding (Single-Precision Floating-Point):

$+\ 1.11100100000000000000010_2 \times 2^{4}$

$+\ 1.10000000000000110000101_2 \times 2^{2}$

❖ Cannot add significands … Why?

- ✧ Because exponents are not equal

❖ How to make exponents equal?

- ✧ Shift the significand of the lesser exponent right
- ✧ Difference between the two exponents = 4 – 2 = 2
- ✧ So, shift right second number by 2 bits and increment exponent

$1.10000000000000110000101_2 \times 2^{2}$

$=\ 0.01100000000000001100001\ 01_2 \times 2^{4}$

Floating-Point Addition - cont'd

❖ Now, ADD the Significands:

```
+ 1.11100100000000000000010      × 2^4
+ 1.10000000000000110000101      × 2^2
-------------------------------------------
+ 1.11100100000000000000010      × 2^4
+ 0.01100000000000001100001 01   × 2^4  (shift right)
-------------------------------------------
+10.01000100000000001100011 01   × 2^4  (result)
```

❖ Addition produces a carry bit, result is NOT normalized

❖ Normalize Result (shift right and increment exponent):

$+\ 10.01000100000000001100011\ \ 01 \times 2^{4}$

$=\ +\ 1.00100010000000000110001\ \ 101 \times 2^{5}$

Rounding

- ❖ Single-precision requires only 23 fraction bits
- ❖ However, Normalized result can contain additional bits

```
1.00100010000000000110001 | 1 01 × 2^5
```

***Round Bit:* R = 1** · ***Sticky Bit:* S = 1**

- ❖ Two extra bits are needed for rounding
  - ✧ Round bit: appears just after the normalized result
  - ✧ Sticky bit: appears after the round bit (OR of all additional bits)
- ❖ Since **RS = 11**, increment fraction to round to nearest

```
1.00100010000000000110001 × 2^5
                       +1
---------------------------------
1.00100010000000000110010 × 2^5  (Rounded)
```


Floating-Point Subtraction Example

- ❖ Sometimes, addition is converted into subtraction
  - ✧ If the sign bits of the operands are different
- ❖ Consider Adding:

```
+ 1.00000000101100010001101 × 2^-6
– 1.00000000000000010011010 × 2^-1
------------------------------------------------
+ 0.00001000000001011000100 01101 × 2^-1 (shift right 5 bits)
– 1.00000000000000010011010       × 2^-1
------------------------------------------------
0 0.00001000000001011000100 01101 × 2^-1
1 0.11111111111111101100110       × 2^-1 (2's complement)
------------------------------------------------
1 1.00001000000001000101010 01101 × 2^-1 (ADD)
------------------------------------------------
- 0.11110111111110111010101 10011 × 2^-1 (2's complement)
```

- ❖ 2's complement of result is required if result is negative

Floating-Point Subtraction - cont'd

```
+ 1.00000000101100010001101 × 2^-6
- 1.00000000000000010011010 × 2^-1
-----------------------------------
- 0.11110111111110111010101 10011 × 2^-1
```
(result is negative)

❖ Result should be normalized

✧ For subtraction, we can have leading zeros. To normalize, count the number of leading zeros, then shift result left and decrement the exponent accordingly.

Guard bit

```
- 0.11110111111110111010101 1 0011 × 2^-1
------------------------------------------
- 1.11101111111101110101011   0011 × 2^-2
```
(Normalized)

❖ Guard bit: guards against loss of a fraction bit

✧ Needed for subtraction, when result has a leading zero and should be normalized.

Floating-Point Subtraction - cont'd

❖ Next, normalized result should be rounded

Guard bit

```
- 0.11110111111110111010101 1 0 011 × 2^-1
-------------------------------------------
- 1.11101111111101110101011   0 011 × 2^-2
```
(Normalized)

Round bit: R=0 *Sticky bit: S = 1*

❖ Since **R = 0**, it is more accurate to truncate the result even if **S = 1**. We simply discard the extra bits.

```
- 1.11101111111101110101011   0 011 × 2^-2
------------------------------------------
- 1.11101111111101110101011         × 2^-2
```
(Normalized)

(Rounded to nearest)

❖ IEEE 754 Representation of Result

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

Rounding to Nearest Even

- Normalized result has the form: 1. $f_1\ f_2\ \ldots\ f_l$ **R S**
  - The round bit **R** appears after the last fraction bit f_l
  - The sticky bit **S** is the OR of all remaining additional bits
- Round to Nearest Even: default rounding mode
- Four cases for **RS**:
  - **RS = 00** ➔ Result is Exact, no need for rounding
  - **RS = 01** ➔ **Truncate** result by discarding **RS**
  - **RS = 11** ➔ **Increment** result: ADD 1 to last fraction bit
  - **RS = 10** ➔ Tie Case (either truncate or increment result)
    - **Check Last fraction bit f_l** (f_{23} for single-precision or f_{52} for double)
    - **If f_l is 0 then truncate result to keep fraction even**
    - **If f_l is 1 then increment result to make fraction even**

Additional Rounding Modes

- IEEE 754 standard specifies four rounding modes:

1. Round to Nearest Even: described in previous slide
2. Round toward +Infinity: result is rounded up

   Increment result if sign is positive and **R** or **S** = **1**
3. Round toward -Infinity: result is rounded down

   Increment result if sign is negative and **R** or **S** = **1**
4. Round toward 0: always truncate result

- Rounding or Incrementing result might generate a carry
  - This occurs when all fraction bits are **1**
  - Re-Normalize after Rounding step is required only in this case

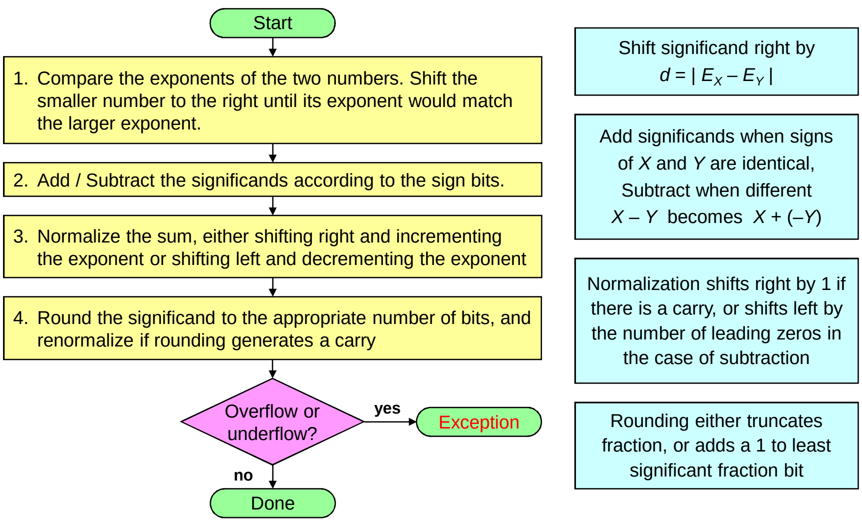
Example on Rounding

- Round following result using IEEE 754 rounding modes:

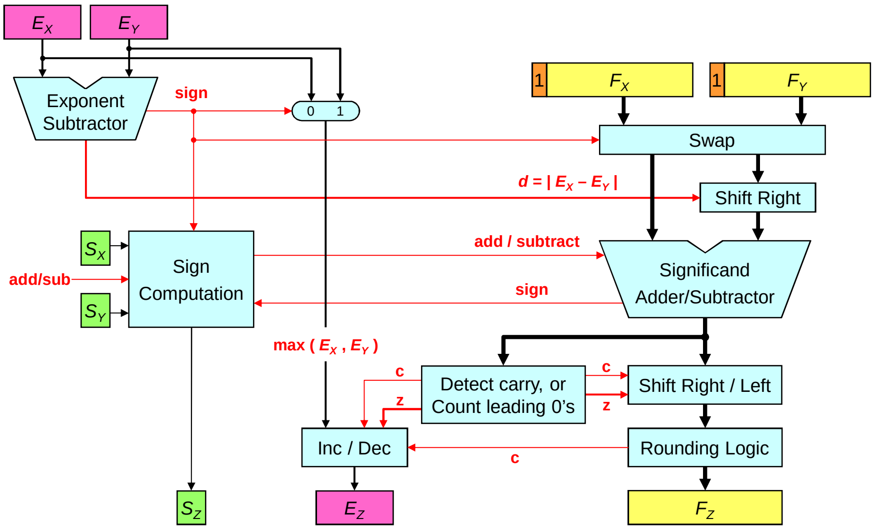
  –1.11111111111111111111111 1 0 × 2^-7^ — Round Bit: 1, Sticky Bit: 0

- Round to Nearest Even:
  - Increment result since **RS** = **10 and f_{23} = 1**
  - Incremented result: **–10.00000000000000000000000 × 2^-7^**
  - Renormalize and increment exponent (because of carry)
  - Final rounded result: **–1.00000000000000000000000 × 2^-6^**
- Round towards +∞: Truncate result since negative
  - Truncated Result: **–1.11111111111111111111111 × 2^-7^**
- Round towards –∞: Increment since negative and **R** = **1**
  - Final rounded result: **–1.00000000000000000000000 × 2^-6^**
- Round towards 0: Truncate always

Floating Point Addition / Subtraction



Floating Point Adder Block Diagram



Next . . .

- ❖ Floating-Point Numbers
- ❖ IEEE 754 Floating-Point Standard
- ❖ Floating-Point Addition and Subtraction
- ❖ Floating-Point Multiplication
- ❖ MIPS Floating-Point Instructions

Floating Point Multiplication Example

❖ Consider multiplying:

```
  -1.110 1000 0100 0000 1010 0001₂ × 2^-4
×  1.100 0000 0001 0000 0000 0000₂ × 2^-2
```

❖ Unlike addition, we add the exponents of the operands

✧ Result exponent value = (–4) + (–2) = –6

❖ Using the biased representation: $E_Z = E_X + E_Y - Bias$

✧ E_X = (–4) + 127 = 123 (*Bias* = 127 for single precision)

✧ E_Y = (–2) + 127 = 125

✧ E_Z = 123 + 125 – 127 = 121 (value = –6)

❖ Sign bit of product can be computed independently

❖ Sign bit of product = $Sign_X$ **XOR** $Sign_Y$ = **1** (negative)

Floating-Point Multiplication, cont'd

❖ Now multiply the significands:

```
        (Multiplicand)     1.11010000100000010100001
        (Multiplier)     × 1.10000000001000000000000
 ---------------------------------------------------
                  111010000100000010100001
       111010000100000010100001
     1.11010000100000010100001
 ---------------------------------------------------
    10.1011100011111011111100110010100001000000000000
```

❖ 24 bits × 24 bits ➔ 48 bits (double number of bits)

❖ Multiplicand × 0 = 0   Zero rows are eliminated

❖ Multiplicand × 1 = Multiplicand (shifted left)

Floating-Point Multiplication, cont'd

❖ Normalize Product:

-10.10111000111101111111001100... × 2^{-6}

Shift right and increment exponent because of carry bit

= -1.010111000111101111111001100... × 2^{-5}

❖ Round to Nearest Even: (keep only 23 fraction bits)

1.01011100011110111111100 | 1 100... × 2^{-5}

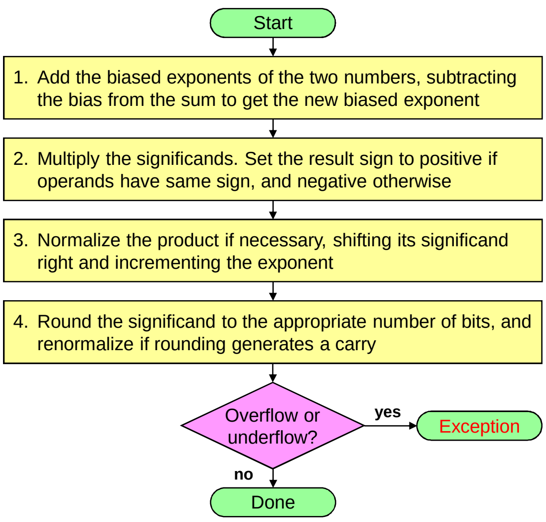
Round bit = **1**, Sticky bit = **1**, so increment fraction

Final result = **-1.01011100011110111111101 × 2^{-5}**

❖ IEEE 754 Representation

1	01111010	01011100011110111111101

Floating Point Multiplication



Biased Exponent Addition
$E_Z = E_X + E_Y - Bias$

Result sign $S_Z = S_X$ **xor** S_Y can be computed independently

Since the operand significands $1.F_X$ and $1.F_Y$ are ≥ 1 and < 2, their product is ≥ 1 and < 4.
To normalize product, we need to shift right at most by 1 bit and increment exponent

Rounding either truncates fraction, or adds a 1 to least significant fraction bit

Extra Bits to Maintain Precision

- Floating-point numbers are approximations for …
  - Real numbers that they cannot represent
- Infinite variety of real numbers exist between 1.0 and 2.0
  - However, exactly 2^{23} fractions represented in Single Precision
  - Exactly 2^{52} fractions can be represented in Double Precision
- Extra bits are generated in intermediate results when …
  - Shifting and adding/subtracting a *p*-bit significand
  - Multiplying two *p*-bit significands (product is 2*p* bits)
- But when packing result fraction, extra bits are discarded
- Few extra bits are needed: guard, round, and sticky bits
- Minimize hardware but without compromising accuracy

Advantages of IEEE 754 Standard

- Used predominantly by the industry
- Encoding of exponent and fraction simplifies comparison
  - Integer comparator used to compare magnitude of FP numbers
- Includes special exceptional values: NaN and ±∞
  - Special rules are used such as:
    - 0/0 is NaN, *sqrt*(–1) is NaN, 1/0 is ∞, and 1/∞ is 0
  - Computation may continue in the face of exceptional conditions
- Denormalized numbers to fill the gap
  - Between smallest normalized number $1.0 \times 2^{E_{min}}$ and zero
  - Denormalized numbers , values $0.F \times 2^{E_{min}}$, are closer to zero
  - Gradual underflow to zero

Floating Point Complexities

- Operations are somewhat more complicated
- In addition to overflow we can have underflow
- Accuracy can be a big problem
  - Extra bits to maintain precision: guard, round, and sticky
  - Four rounding modes
  - Division by zero yields Infinity
  - Zero divide by zero yields Not-a-Number
  - Other complexities
- Implementing the standard can be tricky
  - See text for description of 80x86 and Pentium bug!
- Not using the standard can be even worse

Accuracy can be a Big Problem

Value1	Value2	Value3	Value4	Sum
1.0E+30	-1.0E+30	9.5	-2.3	7.2
1.0E+30	9.5	-1.0E+30	-2.3	-2.3
1.0E+30	9.5	-2.3	-1.0E+30	0

- Adding double-precision floating-point numbers (Excel)
- Floating-Point addition is NOT associative
- Produces different sums for the same data values
- Rounding errors when the difference in exponent is large

Next . . .

- ❖ Floating-Point Numbers
- ❖ IEEE 754 Floating-Point Standard
- ❖ Floating-Point Addition and Subtraction
- ❖ Floating-Point Multiplication
- ❖ MIPS Floating-Point Instructions

MIPS Floating Point Coprocessor

- ❖ Called Coprocessor 1 or the Floating Point Unit (FPU)
- ❖ 32 separate floating point registers: \$f0, \$f1, …, \$f31
- ❖ FP registers are 32 bits for single precision numbers
- ❖ Even-odd register pair form a double precision register
- ❖ Use the even number for double precision registers
  - ✧ \$f0, \$f2, \$f4, …, \$f30 are used for double precision
- ❖ Separate FP instructions for single/double precision
  - ✧ Single precision: add.s, sub.s, mul.s, div.s (.s extension)
  - ✧ Double precision: add.d, sub.d, mul.d, div.d (.d extension)
- ❖ FP instructions are more complex than the integer ones
  - ✧ Take more cycles to execute

FP Arithmetic Instructions

Instruction	Meaning	Format					
add.s fd, fs, ft	(fd) = (fs) + (ft)	0x11	0	ft^5	fs^5	fd^5	0
add.d fd, fs, ft	(fd) = (fs) + (ft)	0x11	1	ft^5	fs^5	fd^5	0
sub.s fd, fs, ft	(fd) = (fs) – (ft)	0x11	0	ft^5	fs^5	fd^5	1
sub.d fd, fs, ft	(fd) = (fs) – (ft)	0x11	1	ft^5	fs^5	fd^5	1
mul.s fd, fs, ft	(fd) = (fs) × (ft)	0x11	0	ft^5	fs^5	fd^5	2
mul.d fd, fs, ft	(fd) = (fs) × (ft)	0x11	1	ft^5	fs^5	fd^5	2
div.s fd, fs, ft	(fd) = (fs) / (ft)	0x11	0	ft^5	fs^5	fd^5	3
div.d fd, fs, ft	(fd) = (fs) / (ft)	0x11	1	ft^5	fs^5	fd^5	3
sqrt.s fd, fs	(fd) = sqrt (fs)	0x11	0	0	fs^5	fd^5	4
sqrt.d fd, fs	(fd) = sqrt (fs)	0x11	1	0	fs^5	fd^5	4
abs.s fd, fs	(fd) = abs (fs)	0x11	0	0	fs^5	fd^5	5
abs.d fd, fs	(fd) = abs (fs)	0x11	1	0	fs^5	fd^5	5
neg.s fd, fs	(fd) = – (fs)	0x11	0	0	fs^5	fd^5	7
neg.d fd, fs	(fd) = – (fs)	0x11	1	0	fs^5	fd^5	7

FP Load/Store Instructions

- Separate floating point load/store instructions
  - lwc1: load word coprocessor 1
  - ldc1: load double coprocessor 1
  - swc1: store word coprocessor 1
  - sdc1: store double coprocessor 1

General purpose register is used as the base register

Instruction	Meaning	Format			
lwc1 \$f2, 40(\$t0)	(\$f2) = Mem[(\$t0)+40]	0x31	\$t0	\$f2	$im^{16} = 40$
ldc1 \$f2, 40(\$t0)	(\$f2) = Mem[(\$t0)+40]	0x35	\$t0	\$f2	$im^{16} = 40$
swc1 \$f2, 40(\$t0)	Mem[(\$t0)+40] = (\$f2)	0x39	\$t0	\$f2	$im^{16} = 40$
sdc1 \$f2, 40(\$t0)	Mem[(\$t0)+40] = (\$f2)	0x3d	\$t0	\$f2	$im^{16} = 40$

- Better names can be used for the above instructions
  - l.s = lwc1 (load FP single), l.d = ldc1 (load FP double)
  - s.s = swc1 (store FP single), s.d = sdc1 (store FP double)

FP Data Movement Instructions

❖ Moving data between general purpose and FP registers
  ✧ mfc1: move from coprocessor 1 (to general purpose register)
  ✧ mtc1: move to coprocessor 1 (from general purpose register)

❖ Moving data between FP registers
  ✧ mov.s: move single precision float
  ✧ mov.d: move double precision float = even/odd pair of registers

Instruction	Meaning	Format					
mfc1 $t0, $f2	($t0) = ($f2)	0x11	0	$t0	$f2	0	0
mtc1 $t0, $f2	($f2) = ($t0)	0x11	4	$t0	$f2	0	0
mov.s $f4, $f2	($f4) = ($f2)	0x11	0	0	$f2	$f4	6
mov.d $f4, $f2	($f4) = ($f2)	0x11	1	0	$f2	$f4	6

FP Convert Instructions

❖ Convert instruction: cvt.x.y
  ✧ Convert to destination format x from source format y

❖ Supported formats
  ✧ Single precision float = .s (single precision float in FP register)
  ✧ Double precision float = .d (double float in even-odd FP register)
  ✧ Signed integer word = .w (signed integer in FP register)

Instruction	Meaning	Format					
cvt.s.w fd, fs	to single from integer	0x11	0	0	fs^5	fd^5	0x20
cvt.s.d fd, fs	to single from double	0x11	1	0	fs^5	fd^5	0x20
cvt.d.w fd, fs	to double from integer	0x11	0	0	fs^5	fd^5	0x21
cvt.d.s fd, fs	to double from single	0x11	1	0	fs^5	fd^5	0x21
cvt.w.s fd, fs	to integer from single	0x11	0	0	fs^5	fd^5	0x24
cvt.w.d fd, fs	to integer from double	0x11	1	0	fs^5	fd^5	0x24

FP Compare and Branch Instructions

- FP unit (co-processor 1) has a condition flag
  - Set to 0 (false) or 1 (true) by any comparison instruction
- Three comparisons: equal, less than, less than or equal
- Two branch instructions based on the condition flag

Instruction	Meaning	Format					
c.eq.s fs, ft	cflag = ((fs) == (ft))	0x11	0	ft^5	fs^5	0	0x32
c.eq.d fs, ft	cflag = ((fs) == (ft))	0x11	1	ft^5	fs^5	0	0x32
c.lt.s fs, ft	cflag = ((fs) < (ft))	0x11	0	ft^5	fs^5	0	0x3c
c.lt.d fs, ft	cflag = ((fs) < (ft))	0x11	1	ft^5	fs^5	0	0x3c
c.le.s fs, ft	cflag = ((fs) <= (ft))	0x11	0	ft^5	fs^5	0	0x3e
c.le.d fs, ft	cflag = ((fs) <= (ft))	0x11	1	ft^5	fs^5	0	0x3e
bc1f Label	branch if (cflag == 0)	0x11	8	0	im^{16}		
bc1t Label	branch if (cflag == 1)	0x11	8	1	im^{16}		

Example 1: Area of a Circle

```
.data
  pi:     .double          3.1415926535897924
  msg:    .asciiz          "Circle Area = "
.text
main:
  ldc1    $f2, pi          # $f2,3 = pi
  li      $v0, 7           # read double (radius)
  syscall                  # $f0,1 = radius
  mul.d   $f12, $f0, $f0   # $f12,13 = radius*radius
  mul.d   $f12, $f2, $f12  # $f12,13 = area
  la      $a0, msg
  li      $v0, 4           # print string (msg)
  syscall
  li      $v0, 3           # print double (area)
  syscall                  # print $f12,13
```

Example 2: Matrix Multiplication

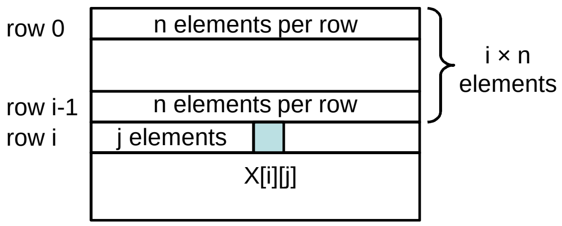
```
void mm (int n, double x[n][n], y[n][n], z[n][n]) {
  for (int i=0; i!=n; i=i+1)
    for (int j=0; j!=n; j=j+1) {
      double sum = 0.0;
      for (int k=0; k!=n; k=k+1)
        sum = sum + y[i][k] * z[k][j];
      x[i][j] = sum;
    }
}
```

- Matrices **x**, **y**, and **z** are **n×n** double precision float
- Matrix size is passed in **$a0 = n**
- Array addresses are passed in **$a1**, **$a2**, and **$a3**
- What is the MIPS assembly code for the procedure?

Address Calculation for 2D Arrays

- Row-Major Order: 2D arrays are stored as rows
- Calculate Address of: **X[i][j]**

  = Address of **X + (i×n+j)×8** (8 bytes per element)



- Address of **Y[i][k] =** Address of **Y + (i×n+k)×8**
- Address of **Z[k][j] =** Address of **Z + (k×n+j)×8**

Matrix Multiplication Procedure – 1/3

❖ Initialize Loop Variables

```
mm:  addu   $t1, $0, $0    # $t1 = i = 0; for 1st loop
L1:  addu   $t2, $0, $0    # $t2 = j = 0; for 2nd loop
L2:  addu   $t3, $0, $0    # $t3 = k = 0; for 3rd loop
     sub.d  $f0, $f0, $f0  # $f0 = sum = 0.0
```

❖ Calculate address of **y[i][k]** and load it into **$f2,$f3**

❖ Skip **i** rows (**i×n**) and add **k** elements

```
L3:  mul    $t4, $t1, $a0  # $t4 = i*size(row) = i*n
     addu   $t4, $t4, $t3  # $t4 = i*n + k
     sll    $t4, $t4, 3    # $t4 =(i*n + k)*8
     addu   $t4, $a2, $t4  # $t4 = address of y[i][k]
     l.d    $f2, 0($t4)    # $f2 = y[i][k]
```


Matrix Multiplication Procedure – 2/3

❖ Similarly, calculate address and load value of **z[k][j]**

❖ Skip **k** rows (**k×n**) and add **j** elements

```
     mul    $t5, $t3, $a0  # $t5 = k*size(row) = k*n
     addu   $t5, $t5, $t2  # $t5 = k*n + j
     sll    $t5, $t5, 3    # $t5 =(k*n + j)*8
     addu   $t5, $a3, $t5  # $t5 = address of z[k][j]
     l.d    $f4, 0($t5)    # $f4 = z[k][j]
```

❖ Now, multiply **y[i][k]** by **z[k][j]** and add it to **$f0**

```
     mul.d  $f6, $f2, $f4  # $f6 = y[i][k]*z[k][j]
     add.d  $f0, $f0, $f6  # $f0 = sum
     addiu  $t3, $t3, 1    # k = k + 1
     bne    $t3, $a0, L3   # loop back if (k != n)
```

Matrix Multiplication Procedure – 3/3

- Calculate address of **`x[i][j]`** and store **`sum`**

```
mul   $t6, $t1, $a0   # $t6 = i*size(row) = i*n
addu  $t6, $t6, $t2   # $t6 = i*n + j
sll   $t6, $t6, 3     # $t6 =(i*n + j)*8
addu  $t6, $a1, $t6   # $t6 = address of x[i][j]
s.d   $f0, 0($t6)     # x[i][j] = sum
```

- Repeat outer loops: L2 (for j = …) and L1 (for i = …)

```
addiu $t2, $t2, 1     # j = j + 1
bne   $t2, $a0, L2    # loop L2 if (j != n)
addiu $t1, $t1, 1     # i = i + 1
bne   $t1, $a0, L1    # loop L1 if (i != n)
```

- Return:

```
jr    $ra             # return
```