

Computer Architecture

Integer Multiplication
Integer Division
Floating Point Numbers

Overview

Multiplying Hardware & Software

Dividing Hardware & Software

Introduction to Floating Point

Doing Floating Point Arithmetic

MIPS Floating Point Instructions

The Dangers of Floating Point

MULTIPLY

- Paper and pencil example (unsigned):

$$\begin{array}{r} 1000 \text{ Multiplicand } U \\ \underline{1001 \text{ Multiplier } M} \\ 1000 \\ 0000 \\ 0000 \\ \times 1000 \\ \hline 01001000 \text{ Product } P \end{array}$$

- Binary multiplication is easy:
 - $P_i == 0 \Rightarrow$ place all 0's (0 \times multiplicand)
 - $P_i == 1 \Rightarrow$ place a copy of U (1 \times multiplicand)
 - Shift the multiplicand left before adding to product
 - *Could we multiply via add, shl?*

Multiply by Power of 2 via Shift Left

- Number representation: $B = b_{31}b_{30} \dots b_2b_1b_0$

$$B = b_{31} \times 2^{31} + b_{30} \times 2^{30} + \dots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

- What if multiply B by 2?

$$\begin{aligned} B \times 2 &= b_{31} \times 2^{31+1} + b_{30} \times 2^{30+1} + \dots + b_2 \times 2^{2+1} + b_1 \times 2^{1+1} + b_0 \times 2 \\ &= b_{31} \times 2^{32} + b_{30} \times 2^{31} + \dots + b_2 \times 2^3 + b_1 \times 2^2 + b_0 \times 2^1 \end{aligned}$$

- What if shift B left by 1?

$$B \ll 1 = b_{30} \times 2^{31} + b_{29} \times 2^{30} + \dots + b_2 \times 2^3 + b_1 \times 2^2 + b_0 \times 2^1$$

- Multiply by 2^i often replaced by **shift left i**

Multiply in MIPS

- Can multiply variable by any **constant** using MIPS `sll` and `add` instructions:

```
i' = i * 10; /* assume i: $s0 */  
  
sll $t0, $s0, 3           # i * 23  
add $t1, $zero, $t0  
sll $t0, $s0, 1           # i * 21  
add $s0, $t1, $t0
```

- MIPS multiply instructions: `mult`, `multu`

- `mult $t0, $t1`

- puts 64-bit product in pair of new registers `hi`, `lo`; copy to `$n` by `mfhi`, `mflo`
- 32-bit integer result in register `lo`

Is Shift Right Arith. \equiv Divide by 2?

- Shifting right by n bits would seem to be the same as dividing by 2^n
- Problem is signed integers
 - Zero fill (`srl`) is wrong for negative numbers

- Shift Right Arithmetic (`sra`); sign extends (replicates sign bit); but does it work?

- Divide -5 by 4 via `sra 2`; result should be -1

```
1111 1111 1111 1111 1111 1111 1111 1011
1111 1111 1111 1111 1111 1111 1111 1110
```

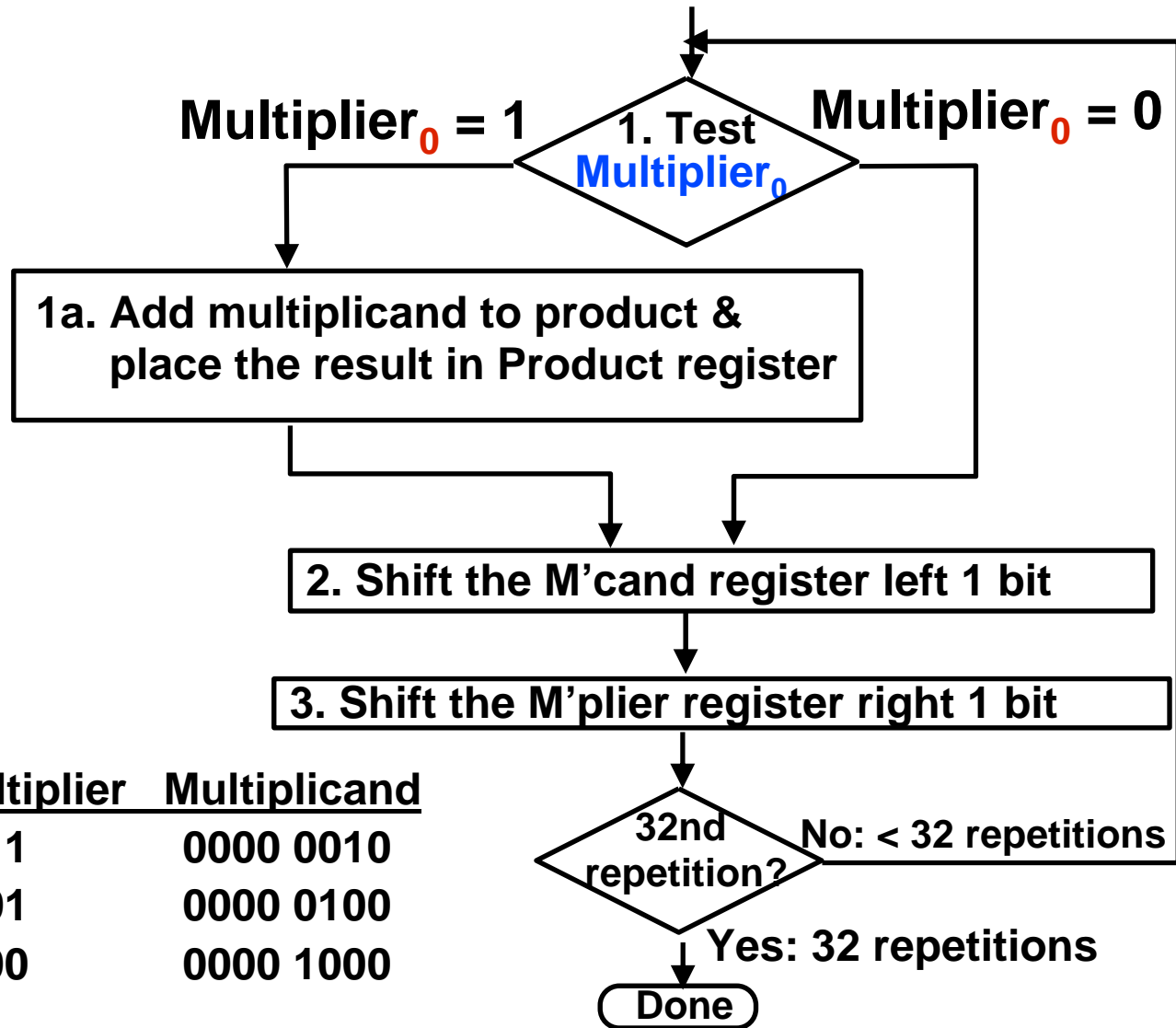
- = -2, not -1; Off by 1, so **doesn't work**

- **Is it always off by 1??**

Multiply Algorithm *Version 1*

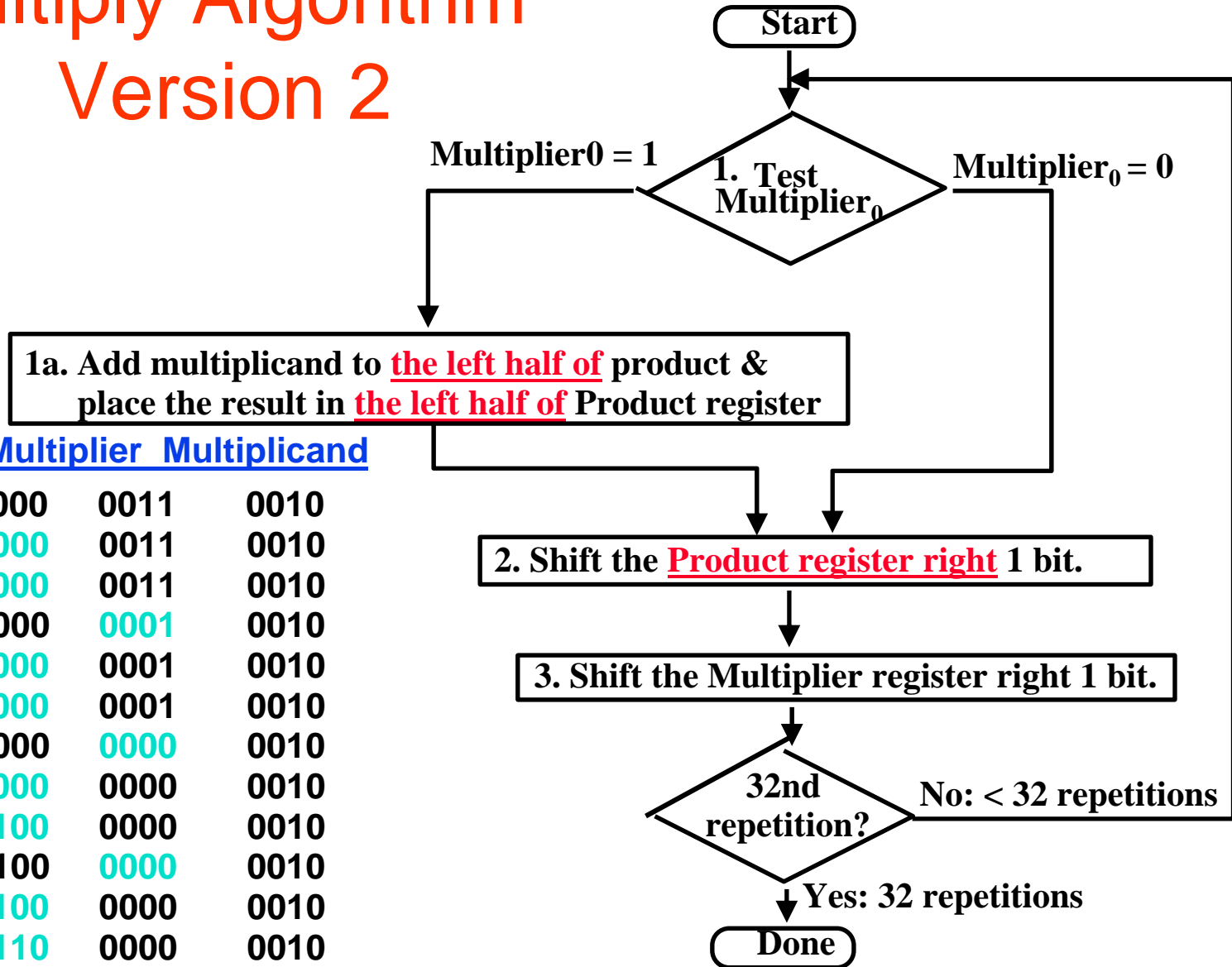
```

    0010
  x 0011
  -----
 00000110
  
```



<u>Product</u>	<u>Multiplier</u>	<u>Multiplicand</u>
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110		

Multiply Algorithm Version 2



Product Multiplier Multiplicand

	0000 0000	0011	0010
1:	0010 0000	0011	0010
2:	0001 0000	0011	0010
3:	0001 0000	0001	0010
1:	0011 0000	0001	0010
2:	0001 1000	0001	0010
3:	0001 1000	0000	0010
1:	0001 1000	0000	0010
2:	0000 1100	0000	0010
3:	0000 1100	0000	0010
1:	0000 1100	0000	0010
2:	0000 0110	0000	0010
3:	0000 0110	0000	0010
	0000 0110	0000	0010

Divide: Paper & Pencil

```

                1001 Quotient
Divisor 1000  $\overline{)1001010}$  Dividend
                -1000
                   10
                   101
                   1010
                   -1000
                      10 Remainder (or Modulo result)

```

See how big a number can be subtracted, creating quotient bit on each step

Binary => 1 * divisor or 0 * divisor

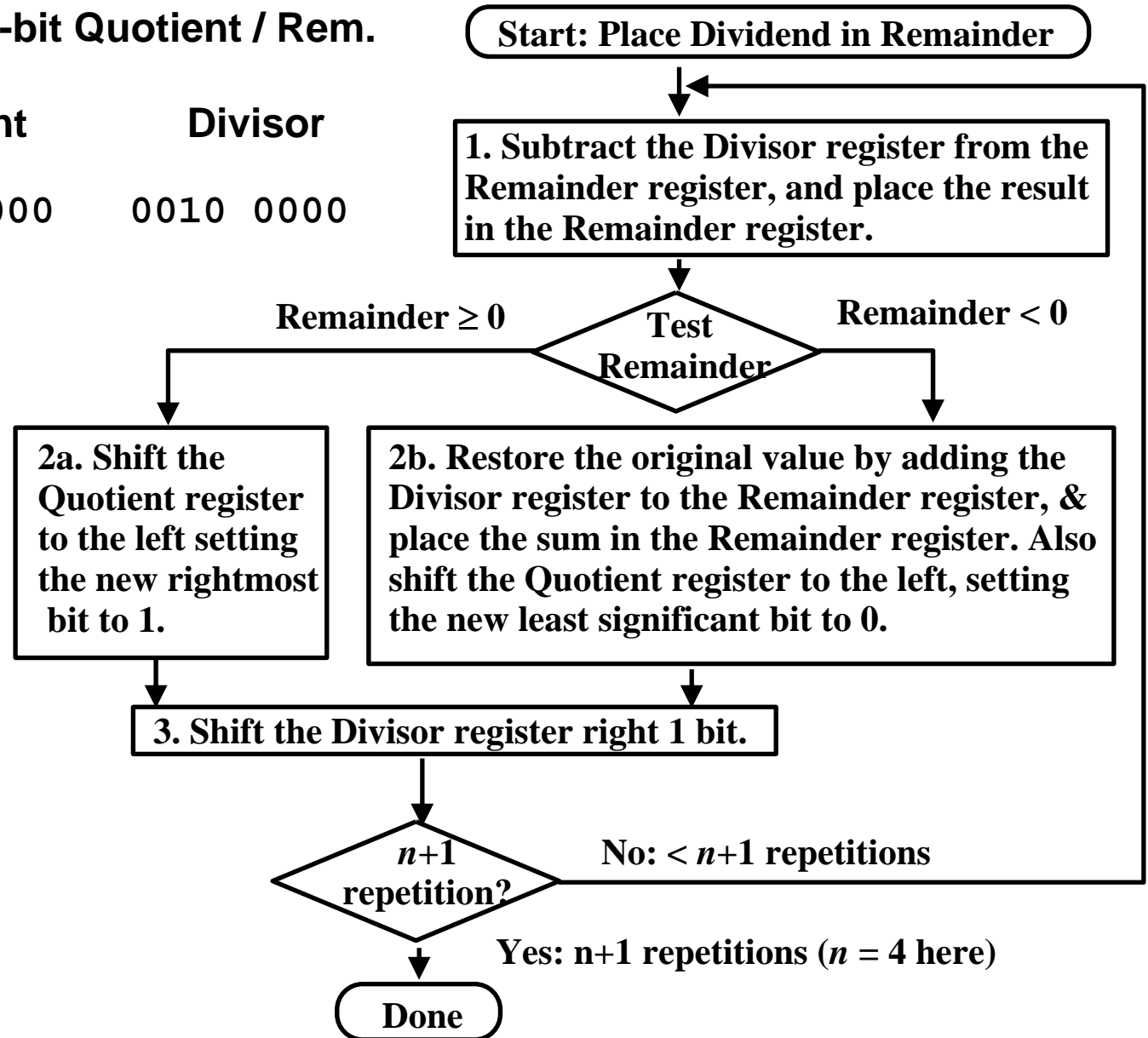
Dividend = Quotient \times Divisor + Remainder \Rightarrow

| Dividend | = | Quotient | + | Divisor |

Divide Algorithm

• Takes $n+1$ steps for n -bit Quotient / Rem.

Remainder	Quotient	Divisor
0000 0111	00000	0010 0000



Divide Algorithm

Let \$s0 = Dividend,
 \$s1 = Divisor,
 \$s2 = Remainder,
 \$s3 = Quotient,
 \$s4 = Repetitions

Start:

```
move    $s2, $s0
```

Loop:

```
sub     $s2, $s2, $s1    # Step 1
bltz   $s2, Label2b
sll    $s3, $s3, 1      # Step 2a
ori    $s3, $s3, 1
j      Label3
```

Label2b:

```
add    $s2, $s2, $s1    # Step 2b
```

```
sll    $s3, $s3, 1
```

Label3:

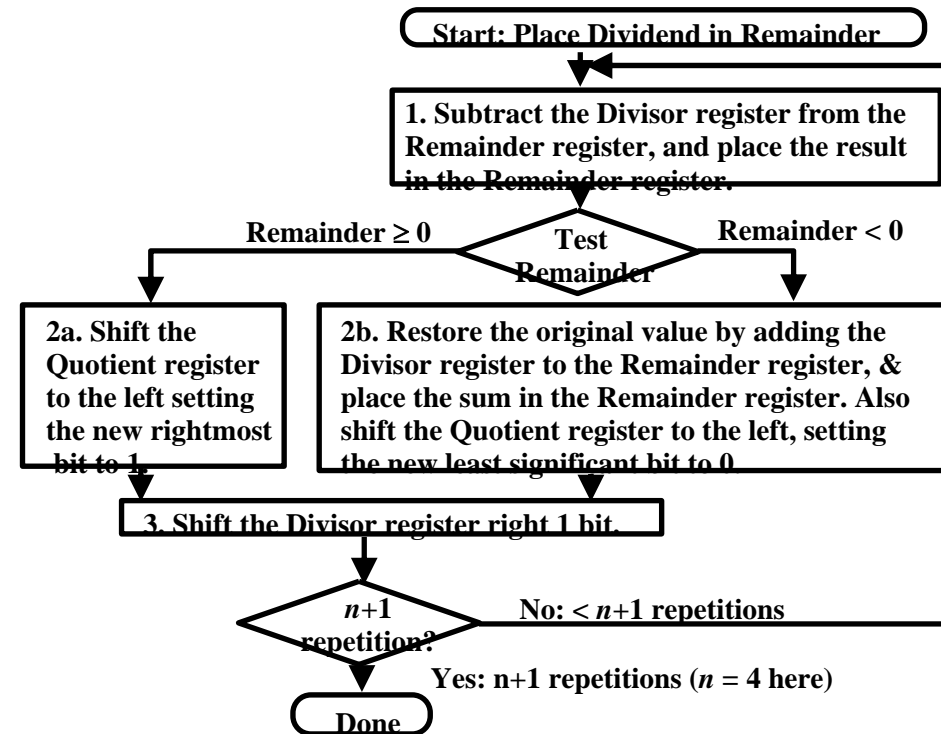
```
slr    $s1, $s1, 1      # Step 3
addi   $s4, $s4, -1     # Reprs.
bgtz   $s4, Loop
```

Quotient = 0; 32 bit *divisor* starts in left 1/2 of Divisor reg. and it is shifted right 1 on each step; Remainder = *dividend*;

If Remainder < 0, we need to add Divisor back to *dividend*; else 1 is generated for Quotient;

Shift Divisor right 1 bit;

Repeat **33** times



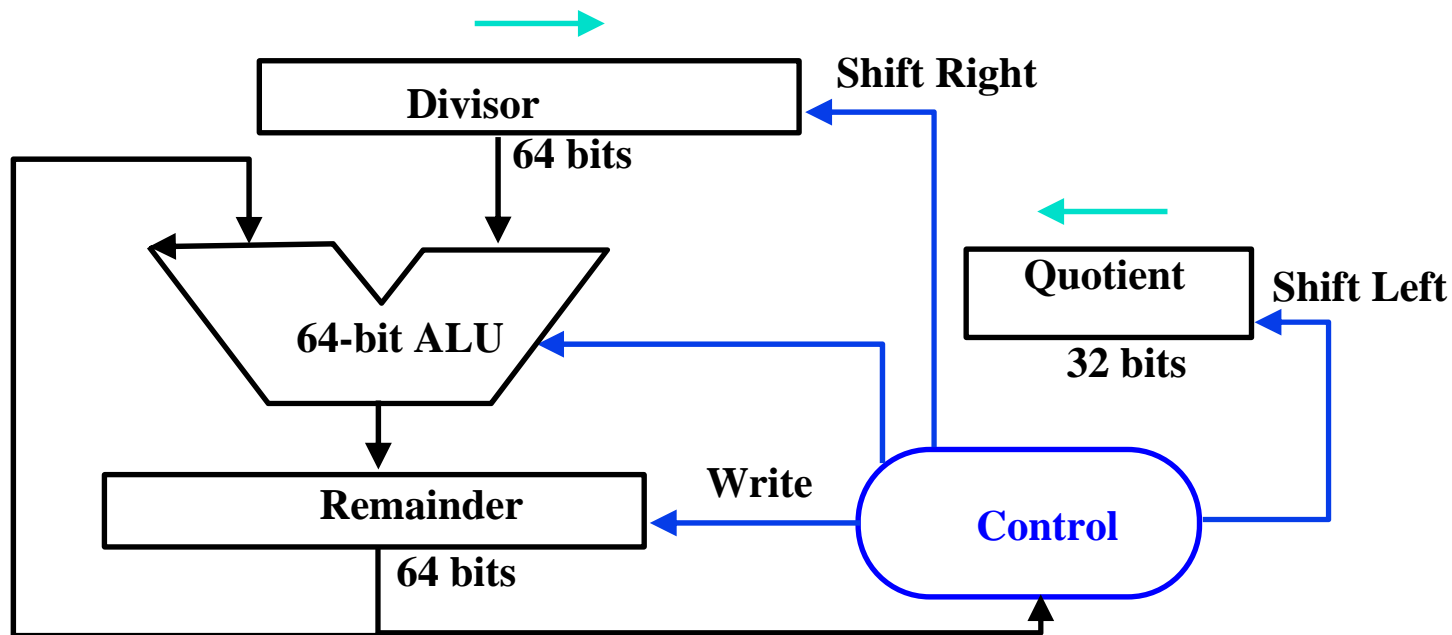
Divide Algorithm Example

	<u>Remainder</u>	<u>Quotient</u>	<u>Divisor</u>		
	0000	0111	00000	0010	0000
1:	1110	0111	00000	0010	0000
2:	0000	0111	00000	0010	0000
3:	0000	0111	00000	0001	0000
1:	1111	0111	00000	0001	0000
2:	0000	0111	00000	0001	0000
3:	0000	0111	00000	0000	1000
1:	1111	1111	00000	0000	1000
2:	0000	0111	00000	0000	1000
3:	0000	0111	00000	0000	0100
1:	0000	0011	00000	0000	0100
2:	0000	0011	00001	0000	0100
3:	0000	0011	00001	0000	0010
1:	0000	0001	00001	0000	0010
2:	0000	0001	00011	0000	0010
3:	0000	0001	00011	0000	0010

Answer:
Quotient = 3
Remainder = 1

Integer Division

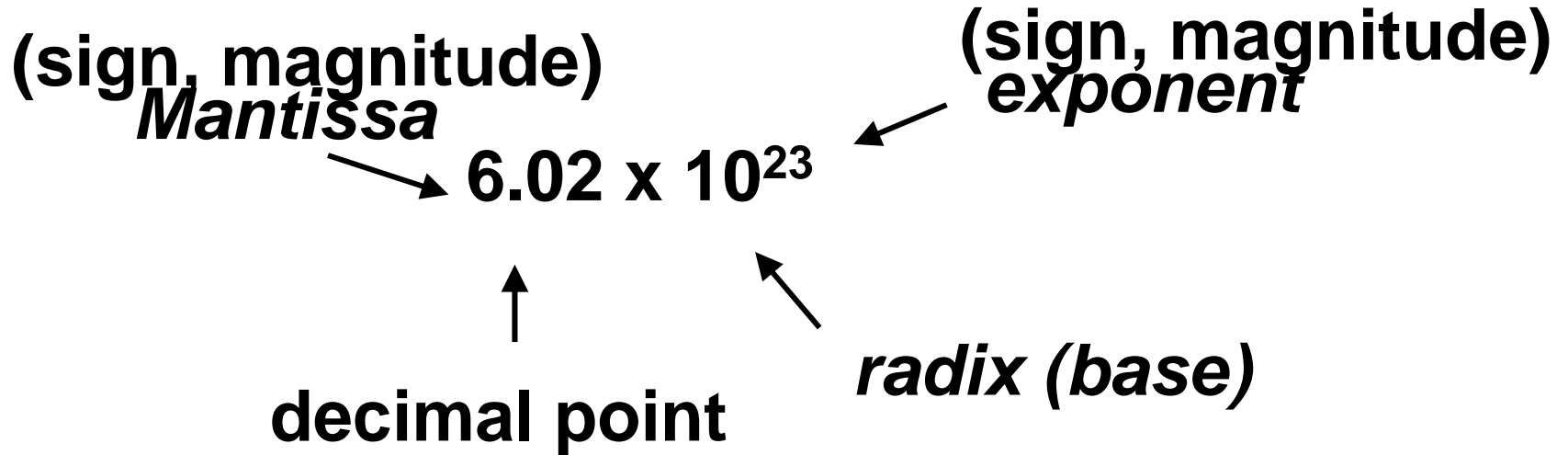
- ALU, Divisor, and Remainder registers: 64bit;
- Quotient register: 32 bits;
- 32 bit *divisor* starts in left 1/2 of Divisor reg. and it is shifted right 1 on each step
- Remainder register initialized with *dividend*



What is in a number?

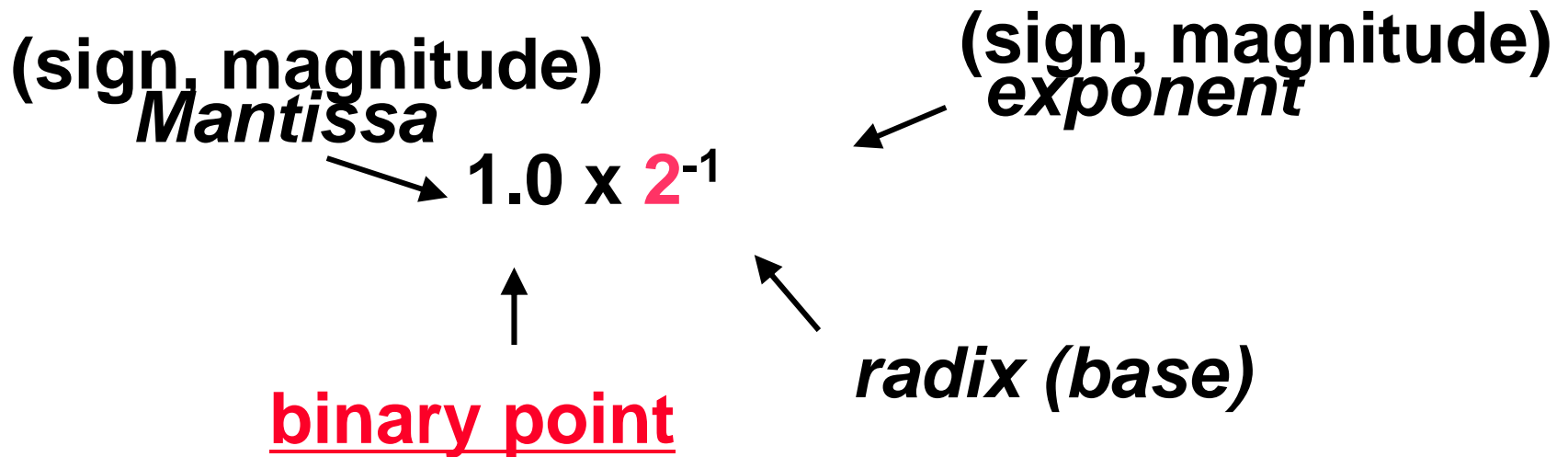
- What can be represented in N bits?
- Unsigned 0 to $2^N - 1$
- 2s Complement -2^{N-1} to $2^{N-1} - 1$
- 1s Complement $-2^{N-1}+1$ to $2^{N-1}-1$
- Excess M 2^{-M} to $2^{N-M}-1$
 - $(E = e + M)$
- BCD 0 to $10^{N/4} - 1$
- But, what about?
 - very large numbers?
9,349,398,989,787,762,244,859,087,678
 - very small number? 0.000000000000000000000000000045691
 - rationals $2/3$
 - irrationals $\sqrt{2}$
 - transcendentals e,

Recall Scientific Notation



- **Normal form:**
no leading 0s (digit **1** to left of decimal point)
- **Alternatives to representing 1/1,000,000,000**
Normalized: 1.0×10^{-9}
Not normalized: 0.1×10^{-8} , 10.0×10^{-10}

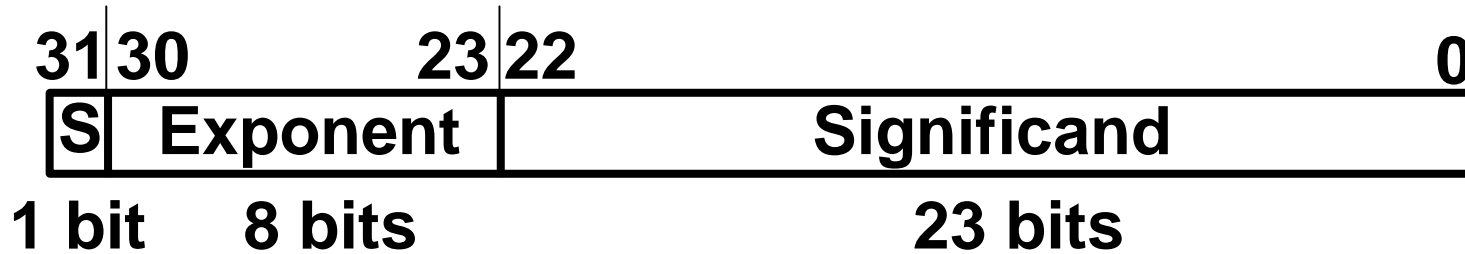
Scientific Notation for Binary Numbers



- Computer arithmetic that supports it called floating point, because it represents numbers where binary point is not fixed, as it is for integers
- Declare such a variable in C as `float` (`double`, `long double`)
- **Normalized form:** $1.\text{xxxxxxxxxx}_2 \times 2^{\text{yyyy}}_2$
Simplifies data exchange, increases accuracy
 $4_{10} == 1.0 \times 2^2$, $8_{10} == 1.0 \times 2^3$

Single Precision FP Representation

- Start with a single word (32-bits)



- Meaning: $(-1)^S \times \text{Mantissa} \times 2^E$
- Can now represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}
- Relationship between Mantissa and Significand bits?
Between E and Exponent?
- In C type `float`

Floating Point Number Representation

- What if result too large? ($> 2.0 \times 10^{38}$)

Overflow!

Overflow \Leftrightarrow Exponent **larger** than can be represented in 8-bit Exponent field

- What if result too small? ($>0, < 2.0 \times 10^{-38}$)

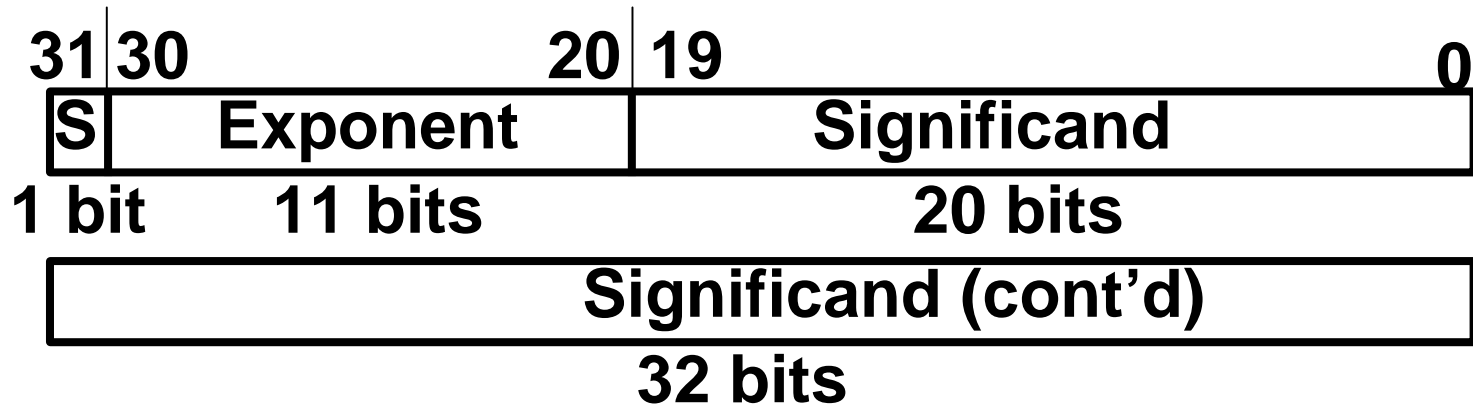
Underflow!

Underflow \Leftrightarrow Negative Exponent **too small**

- How to reduce chances of overflow or underflow?

Double Precision FP Representation

- Next Multiple of Word Size (64 bits)



- Double Precision (vs. Single Precision)
 1. C variable declared as `double`
 2. Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}
 3. But primary advantage greater accuracy due to larger significand
 4. There is also long `double` version (16 bytes)

MIPS follows IEEE 754 F.P. Standard

- To pack more bits, make leading 1 of mantissa **implicit** for normalized numbers

1 + 23 bits single, 1 + 52 bits double

0 has no leading 1, so reserve **exponent value 0** just for number 0.0

Meaning: (almost correct)

$$(-1)^S \times (1 + \text{Significand}) \times 2^{\text{Exponent}},$$

where **0 < Significand < 1**

- If label significand bits left-to-right as s_1, s_2, s_3, \dots then value is:

$$(-1)^S \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + \dots) \times 2^{\text{Exponent}}$$

Representing Exponent

- **Want to compare Fl. Pt. numbers as if they were integers, to help in sorting**

Sign **first** part of number

Exponent **next**, so bigger exponent \Rightarrow bigger number

$$1.1 \times 10^{20} > 1.9 \times 10^{10}$$

- **What About Negative Exponents?**

Use 2's comp? 1.0×2^{-1} vs. $1.0 \times 2^{+1}$ (1/2 v. 2)

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

This notation using integer compare of

1/2 vs. **2** makes $1/2 > 2!$

Doesn't work!

Representing Exponent

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

- Instead, pick notation **0000 0000** as most negative, and **1111 1111** as most positive
- 1.0×2^{-1} vs. $1.0 \times 2^{+1}$ (1/2 v. 2)
- Called **Biased Notation**, where bias is number subtracted to get real number
 - IEEE 754 uses bias of 127 for single precision
 - Representation (Finally, the truth!):
 $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - 127)}$
 - 1023 is bias for double precision

Example: Converting Decimal to FP

- Show MIPS representation of -0.75
(show exponent in decimal to simplify)

$$-0.75 = -3/4 = -3/2^2$$

$$-11_{\text{two}}/2^2 = -11_{\text{two}} \times 2^{-2} = -0.11_{\text{two}} \times 2^0$$

$$\text{Normalized to } -1.1_{\text{two}} \times 2^{-1}$$

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

$$(-1)^1 \times (1 + .100\ 0000 \dots 0000) \times 2^{(126-127)}$$

1	0111 1110	100 0000 0000 0000 0000 0000
---	-----------	------------------------------

S = 1; Exponent = 126; Significand = 100 ... 000₂

Example: Converting FP to Decimal

- Sign $S = 0 \Rightarrow$ positive

- Exponent E :

$$0110\ 1000_{\text{two}} = 104_{\text{ten}}$$

$$\text{Bias adjustment: } 104 - 127 = -13$$

- Mantissa:

$$1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$$

$$= 1 + (5,587,778 / 2^{23})$$

$$= 1 + (5,587,778 / 8,388,608) = 1.0 + 0.666115$$

- Represents: $1.666115_{\text{ten}} \times 2^{-13} \sim 2.034 \times 10^{-4}$

0	0110 1000	101 0101 0100 0011 0100 0010
---	-----------	------------------------------

Continuing Example: Binary to ???

- Convert 2's Complement to Integer:

$$2^{29} + 2^{28} + 2^{26} + 2^{22} + 2^{20} + 2^{18} + 2^{16} + 2^{14} + 2^9 + 2^8 + 2^6 + 2^1 = 878,003,010_{\text{ten}}$$

0011 0100 0101 0101 0100 0011 0100 0010

- Convert Binary to Instruction:

0011	0100	0101	0101	0100	0011	0100	0010
13	2	21	17218				

```
ori $s5, $v0, 17218
```

- Convert Binary to ASCII:

0011	0100	0101	0101	0100	0011	0100	0010
4	U	C	B				

Principle: Type not associated with Data

- What does this bit pattern mean:

```
0011 0100 0101 0101 0100 0011 0100 0010
```

2.034*10⁻⁴? 878,003,010? "4UCB"?
ori \$s5, \$v0, 17218?

- **Data can be anything; operation of instruction that accesses operand determines its type!**
Side-effect of stored program concept: instructions stored as numbers
- **Power/danger of unrestricted addresses/ pointers: use ASCII as Fl. Pt., instructions as data, integers as instructions, ...**

How to Order Floating Point Fields?

- **"Natural": Sign, Significand, Exponent?**

Problem: If want to sort using integer operations, won't work:

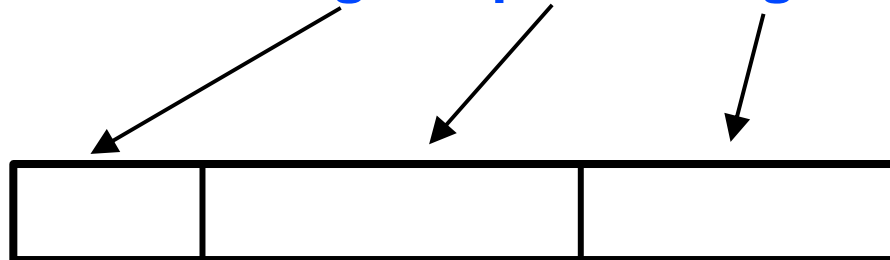
1.0×2^{20} vs. 1.1×2^{10} ; storing significant first makes FP comparisons impossible to carry out correctly via integer comparisons. Second FP looks bigger!



- **Exponent, Sign, Significand?**

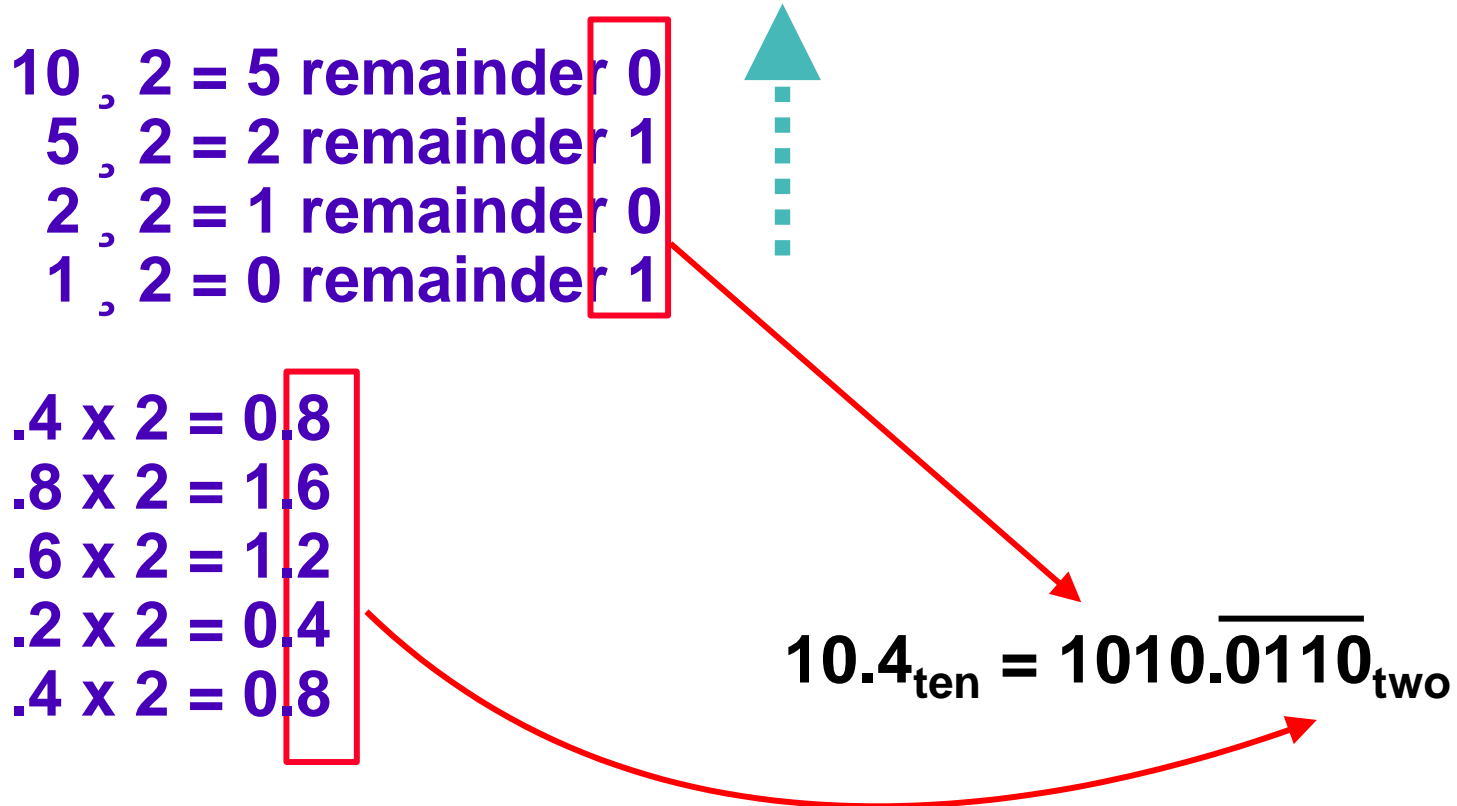
Need to get sign first, since negative < positive

- **Therefore order is Sign Exponent Significant**



How To Convert Decimal to Binary

- How convert 10.4_{ten} to binary?
- Deal with fraction & whole parts separately:



Do It Yourself

- Convert 10.4_{ten} to single precision floating point
- Recall that:

10.4_{ten} is 1010.0110_{two}

Do It Yourself

(1) Normalize

$$1010.0110_{\text{two}} \times 2^0 = 1.0100110 \times 2^3$$

(2) Determine Sign Bit

positive, so $S = 0$

(3) Determine Exponent:

$$2^3 \text{ so } 3 + \text{bias} (= 127) = 130 = 10000010_{\text{two}}$$

(4) Determine Significand

drop leading 1 of mantissa, expand to

23 bits = 01001100110011001100110



S

Exponent

Significand

Example: Converting FP to Decimal

1 **Sign: 0** \Rightarrow positive

2 **Exponent:**

$$0110\ 1000_2 = 104_{10}$$

$$\text{Bias adjustment: } 104 - 127 = -13$$

3 **Mantissa:**

$$\begin{aligned} &1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} \\ &= 1 + (5,587,778 / 2^{23}) \\ &= 1 + (5,587,778 / 8,388,608) = 1.0 + \\ &0.666115 \end{aligned}$$

4 **Represents:** $1.666115_{\text{ten}} * 2^{-13} \sim 2.034 * 10^{-4}$

0	0110 1000	101 0101 0100 0011 0100 0010
---	-----------	------------------------------

Example: Decimal F. P. Addition

- Assume 4 digit significand, 2 digit exponent
 - Let's add $9.999_{\text{ten}} \times 10^1 + 1.610_{\text{ten}} \times 10^{-1}$
 - Exponents must match, so adjust smaller number to match larger exponent
- $1.610 \times 10^{-1} = 0.1610 \times 10^0 = 0.01610 \times 10^1$
- Can represent only 4 digits, so must discard last two:

$$0.016 \times 10^1$$

Example: Decimal F. P. Addition

- Now, add significands:

$$\begin{array}{r} 9.999 \\ + 0.016 \\ \hline 10.015 \end{array}$$

- Thus, sum is 10.015×10^1
- Sum is not normalized, so correct it, checking for underflow/overflow:

$$10.015 \times 10^1 \Rightarrow 1.0015 \times 10^2$$

- Cannot store all digits, must round. Final result is:

$$1.002 \times 10^2$$

Basic Binary FP Addition Algorithm

For addition (or subtraction) of X to Y ($X < Y$):

1. Compute $D = \text{Exp}_Y - \text{Exp}_X$ (align binary points)
2. Right shift $(1+\text{Sig}_X)$ D bits $\Rightarrow (1+\text{Sig}_X) \cdot 2^{-D}$
3. Compute $(1+\text{Sig}_X) \cdot 2^{-D} + (1+\text{Sig}_Y)$; Normalize if necessary; continue until **MS** bit is **1**
4. Too small (e.g., **0.001xx...**) left shift result, decrement result exponent; check for **underflow**
- 4'. Too big (e.g., **10.1xx...**) right shift result, increment result exponent; check for **overflow**
5. If result significand is **0**, set exponent to **0**

FP Subtraction

- **Similar to addition**

- **How do we do it?**

 - De-normalize to match exponents

 - Subtract significands

 - Keep the same exponent

 - Normalize (possibly changing exponent)

- **Problems in implementing FP add/sub:**

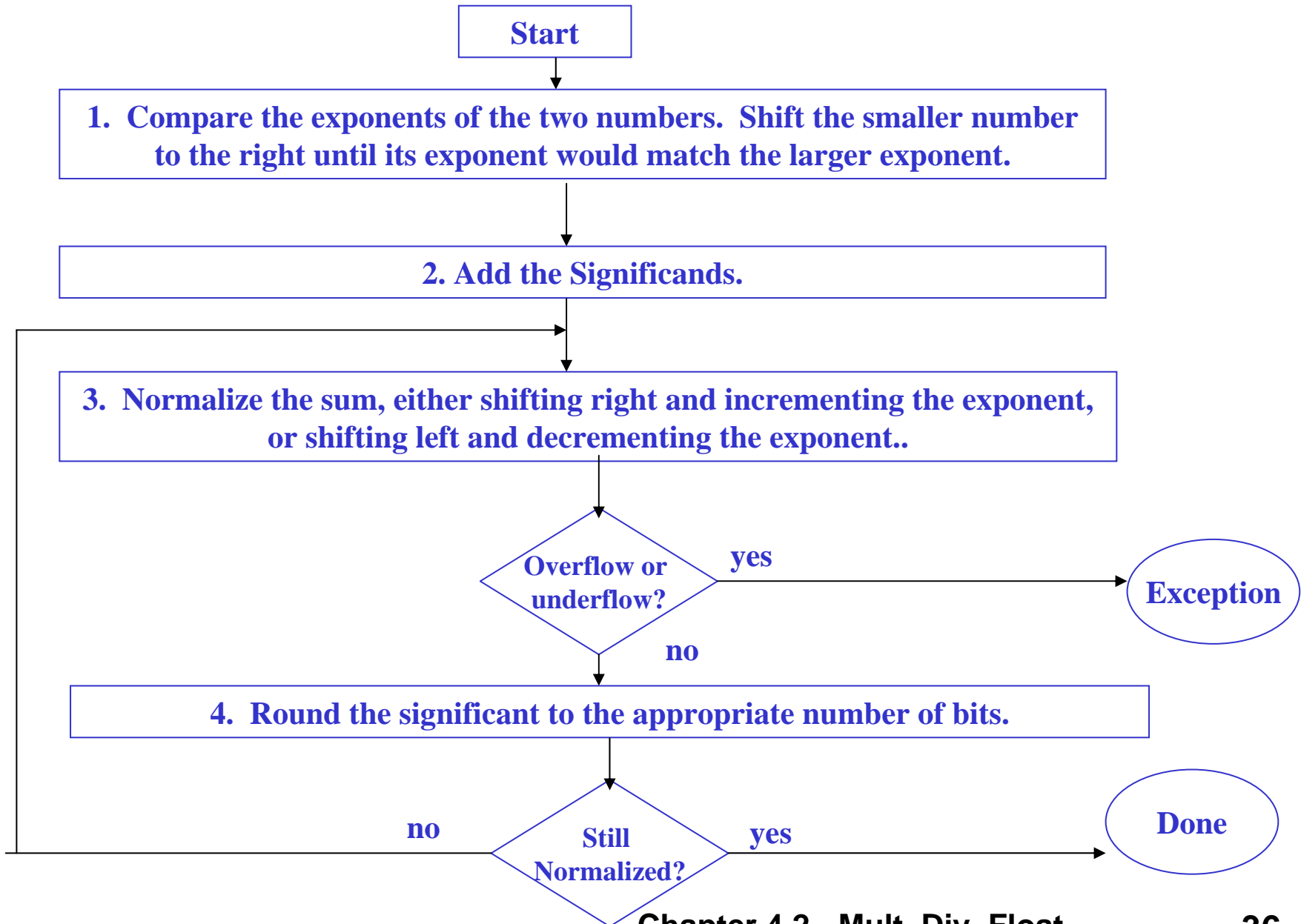
 - Managing the signs,

 - determining to add or sub,

 - swapping the operands.

- **Question: How do we integrate this into the integer arithmetic unit?**

Floating Point Addition



Example: Decimal F. P. Multiply

- Let's multiply:

$$1.110_{\text{ten}} \times 10^{10} \times 9.200_{\text{ten}} \times 10^{-5}$$

(Assume 4-digit significand, 2-digit exponent)

- First, add exponents:

$$\begin{array}{r} 10 \\ + -5 \\ \hline 5 \end{array}$$

- Next, multiply significands:

$$1.110 \times 9.200 = 10.212000$$

Example: Decimal F. P. Multiply

- Product is not normalized, so **correct** it, checking for underflow / overflow:

$$10.212000 \times 10^5 \Rightarrow 1.0212 \times 10^6$$

- Significand exceeds 4 digits, so **round**:

$$1.021 \times 10^6$$

- **Check signs** of original operands
same \Rightarrow positive
different \Rightarrow negative

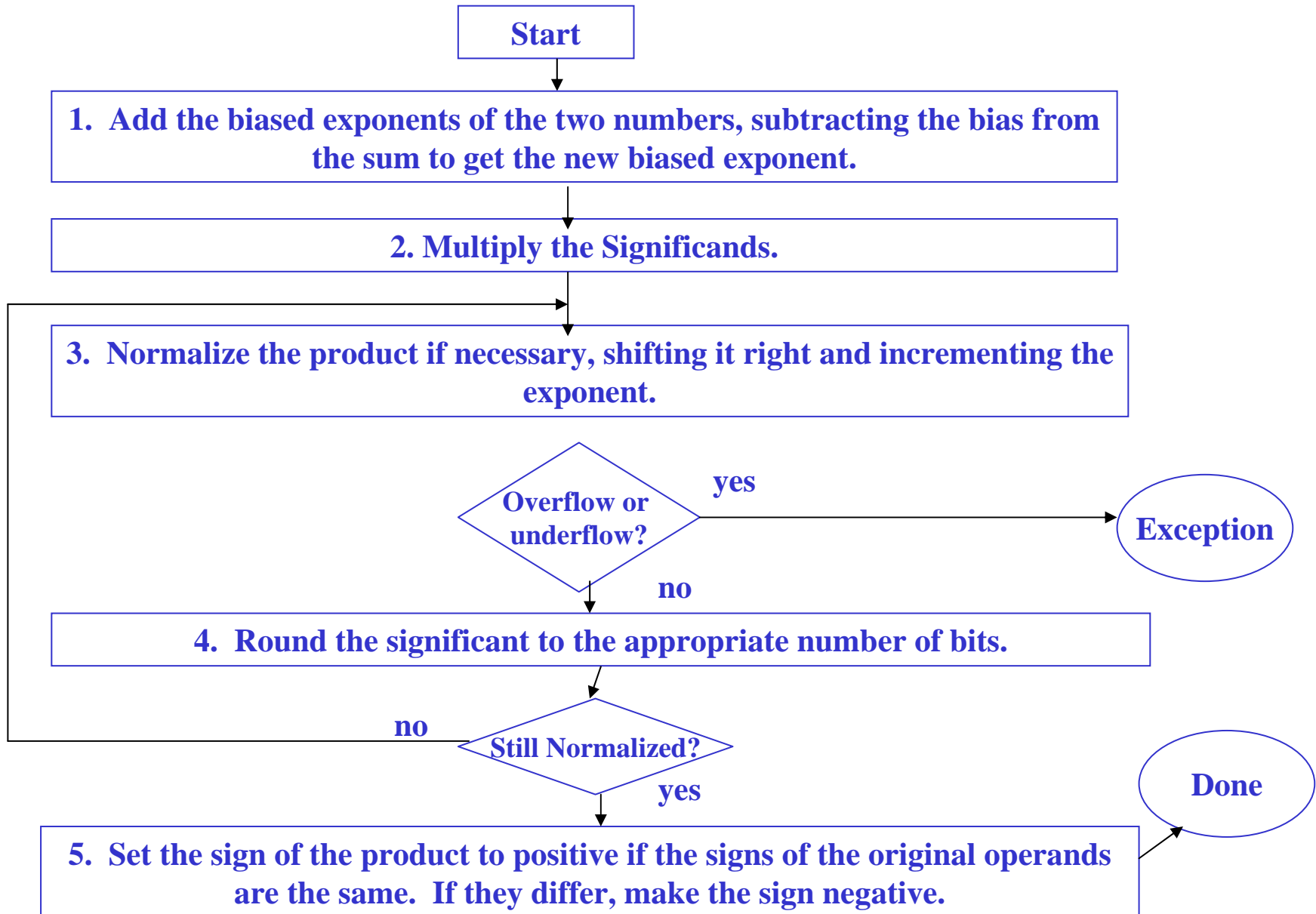
Final result is: $+1.021 \times 10^6$

Basic Binary FP Multiplication Algorithm

For multiplication of $P = X \times Y$:

1. **Compute** Exponent: $\text{Exp}_P = (\text{Exp}_Y + \text{Exp}_X) - \text{Bias}$
2. **Compute** Product: $(1 + \text{Sig}_X) \times (1 + \text{Sig}_Y)$
Normalize if necessary; continue until most significant bit is 1
4. **Too small** (e.g., **0.001xx...**) \rightarrow
left shift result, decrement result exponent
- 4'. **Too big** (e.g., **10.1xx...**) \rightarrow
right shift result, increment result exponent
5. If (**result significand is 0**) then set exponent to 0
6. if (**$\text{Sgn}_X == \text{Sgn}_Y$**) then
 $\text{Sgn}_P = \text{positive (0)}$
else
 $\text{Sgn}_P = \text{negative (1)}$

FP Multiplication Algorithm



Representation for Not a Number

- **What do I get if I calculate**

`sqrt(-4.0)` or

`0/0`?

- **If infinity is not an error, these shouldn't be either.**

Called Not a Number (NaN)

Exponent = 255, Significand nonzero

- **Why is this useful?**

Hope NaNs help with debugging?

They contaminate: $op(\text{NaN}, X) = \text{NaN}$

What else can I put in?

- What defined so far? (**Single Precision**)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. number
255	0	+/- infinity
255	<u>nonzero</u>	<u>???</u>

- Representing "**Not a Number**"; e.g., $\text{sqrt}(-4)$; called NaN

Exp == **255**, Significand **nonzero**

They contaminate FP ops: $(\text{NaN} \theta X) = \text{NaN}$

Hope NaNs help with debugging?

Only valid operations are == , !=

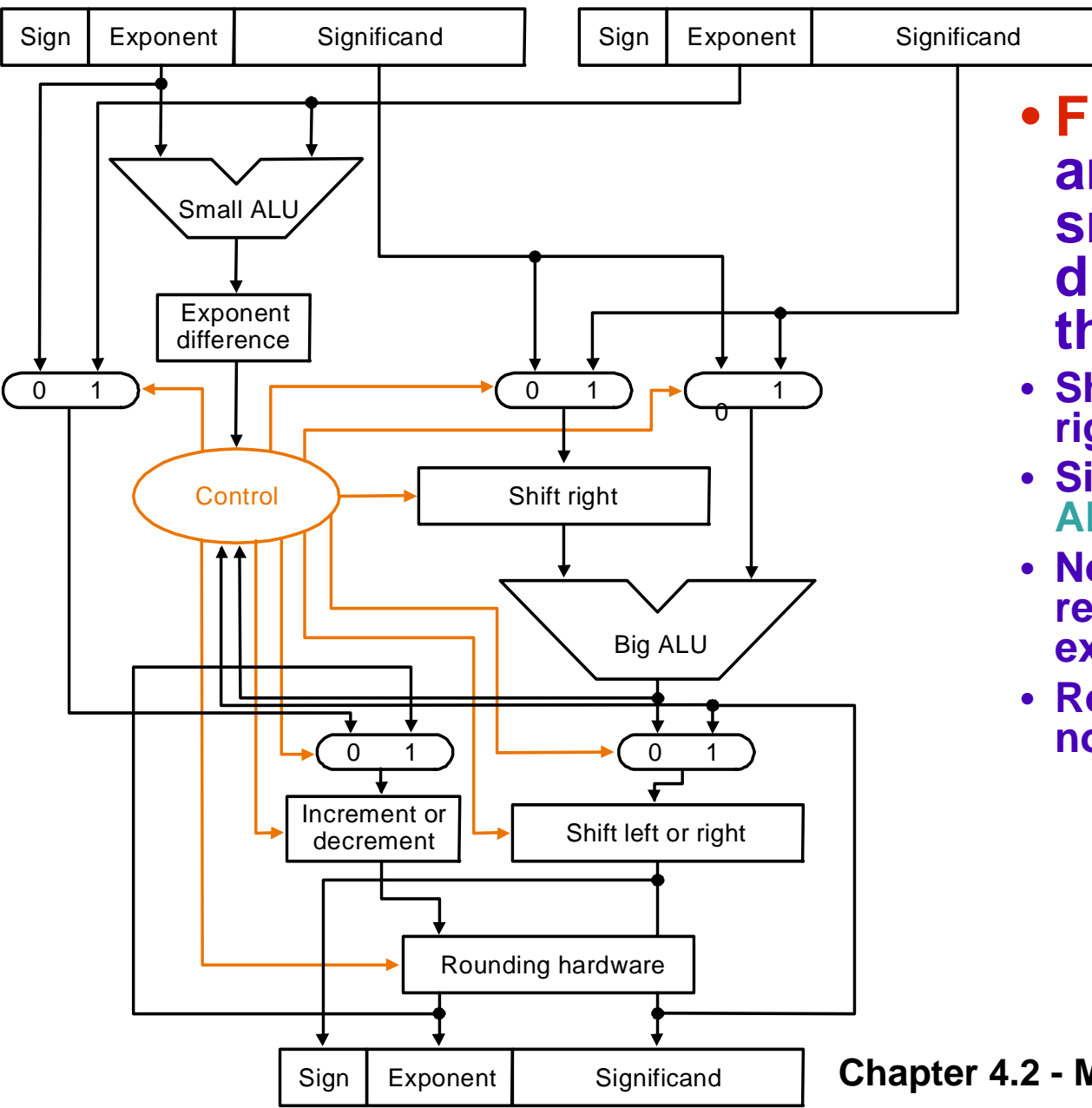
What else can I put in?

- What defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	<u>???</u>
1-254	anything	+/- fl. pt. number
255	0	+/- infinity
255	nonzero	NaN

- Exp. = 0, Significand **nonzero**?
Can we get greater precision?
- Represent very, very small magnitude numbers
- $0 < x <$ smallest normalized number);
- Denormalized Numbers (text p. 300, and discussion later).

Floating Point ALU



- **FP ADD:** Exponents are subtracted by small ALU; the difference controls the 3 MUXes;
- Shift **smaller** exp. to the right until exponents match;
- Significands are added in **Big ALU**;
- Normalization step shifts result **left** or **right**, adjusts exponents;
- Rounding and possible normalization

MIPS Floating Point Architecture (1/4)

- **Separate floating point instructions:**
 - Single Precision:
`add.s, sub.s, mul.s, div.s`
 - Double Precision:
`add.d, sub.d, mul.d, div.d`
- **These instructions are far more complicated than their integer counterparts, so they can take much longer to execute.**

MIPS Floating Point Architecture (2/4)

- **Problems:**

It's inefficient to have different instructions take vastly differing amounts of time.

Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.

Some programs do no floating point calculations

It takes lots of hardware relative to integers to do Floating Point fast

MIPS Floating Point Architecture (3/4)

- **1990 Solution: Make a completely separate chip that handles only FP.**
- **Coprocessor 1: FP chip**
 1. contains 32 32-bit registers: $\$f0, \$f1, \dots$
 2. most of the registers specified in `.s` and `.d` instruction refer to this set
 3. separate load and store: `lwc1` and `swc1` (“load word coprocessor 1”, “store ...”)
 4. Double Precision: by convention, even/odd pair contain one DP FP number: $\$f0/\$f1, \$f2/\$f3, \dots, \$f30/\$f31$

MIPS Floating Point Architecture (4/4)

- **1990 Computer actually contains multiple separate chips:**

Processor: handles all the normal stuff

Coprocessor 1: handles FP and only FP;

more coprocessors?... Yes, later

Today, FP coprocessor integrated with CPU, or cheap chips may leave out FP
HW

- **Instructions to move data between main processor and coprocessors:**

`mfc1 rt, rd` Move floating point register `rd` to
CPU register `rt`.

`mtc1 rd, rt` Move CPU register `rt` to floating
point register `rd`.

`mfc1.d rdest, frsrcl` Move floating point registers
`frsrcl` & `frsrcl + 1` to CPU
registers `rdest` & `rdest + 1`.

- **Appendix pages A-70 to A-74 contain many, many more FP operations.**

Summary: MIPS F.P. Architecture

- **Single Precision, Double Precision versions of add, subtract, multiply, divide, compare**

Single `add.s, sub.s, mul.s, div.s, c.lt.s`

Double `add.d, sub.d, mul.d, div.d, c.lt.d`

See pages A-70 - A74

- **Registers?**

– Normally integer and Floating Point operations on different data, for performance should have separate registers.

– MIPS adds **32** 32-bit FP regs: `$f0, $f1, $f2 ...`,

– Thus need FP data transfers:

l.d **fdest, address** load the floating point double at address into register fdest.

mov.s **fd, fs** Move the floating point single from register fs to register fd.

– **Double Precision?** Even-odd pair of registers:

`$f0-$f1, $f2-$f3, etc.`, act as 64-bit register: `$f0, $f2, $f4,`

Example with F.P.: Matrix Multiply

```
void mm (double x[][], double y[][], double z[][] ){
    int i, j, k;

    for (i=0; i!=32; i=i+1)
        for (j=0; j!=32; j=j+1)
            for (k=0; k!=32; k=k+1)

                x[i][j] = x[i][j] + y[i][k] * z[k][j];
}
```

- Starting addresses are parameters in \$a0, \$a1, and \$a2. Integer variables are in \$t3, \$t4, \$t5. Arrays 32 by 32
- Use pseudoinstructions: `li` (load immediate), `l.d` / `s.d` (load / store 64 bits)

MIPS code 1st piece: initialize `x[][]`

- Initialize Loop Variables

```
mm:    ...
        li    $t1, 32        # $t1 = 32
        li    $t3, 0        # i = 0; 1st loop
L1:    li    $t4, 0        # j = 0; reset 2nd
L2:    li    $t5, 0        # k = 0; reset 3rd
```

- To fetch `x[i][j]`, skip `i` rows ($i*32$), add `j`

```
sll    $t2, $t3, 5        # $t2 = i * 25
addu   $t2, $t2, $t4     # $t2 = i*25 + j
```

- Get byte address (8 bytes), load `x[i][j]`

```
sll    $t2, $t2, 3        # i, j byte addr.
addu   $t2, $a0, $t2     # @ x[i][j]
ld     $f4, 0($t2)       # $f4 = x[i][j]
```

MIPS code 2nd piece: $z[k][j]$, $y[i][k]$

- Like before, but load $z[k][j]$ into $\$f16$

```
L3:  sll      $t0, $t5, 5           # $t0 = k * 25
      addu    $t0, $t0, $t4        # $t0 = k*25 + j
      sll     $t0, $t0, 3         # k, j byte addr.
      addu    $t0, $a2, $t0       # @ z[k][j]
      ld      $f16, 0($t0)       # $f16 = z[k][j]
```

- Like before, but load $y[i][k]$ into $\$f18$

```
      sll     $t0, $t3, 5         # $t0 = i * 25
      addu    $t0, $t0, $t5       # $t0 = i*25 + k
      sll     $t0, $t0, 3         # i, k byte addr.
      addu    $t0, $a1, $t0      # @ y[i][k]
      ld      $f18, 0($t0)      # $f18 = y[i][k]
```

- Summary: $\$f4$: $x[i][j]$, $\$f16$: $z[k][j]$, $\$f18$: $y[i][k]$

MIPS code for last piece: add/mul, loops

- Add $y*z$ to x

```
mul.d $f16,$f18,$f16      # y[][]*z[][]
add.d $f4, $f4, $f16      # x[][]+ y*z
```

- Increment k ; if end of inner loop, store x

```
addiu $t5, $t5,1          # k = k + 1
bne   $t5, $t1,L3         # if(k!=32) goto L3
s.d   $f4, 0($t2)        # x[i][j] = $f4
```

- Increment j ; middle loop if not end of j

```
addiu $t4, $t4,1          # j = j + 1
bne   $t4, $t1,L2         # if(j!=32) goto L2
```

- Increment i ; if end of outer loop, return

```
addiu $t3,$t3,1           # i = i + 1
bne   $t3,$t1,L2          # if(i!=32) goto L1
jr    $ra
```

Floating Point gottchas: Add Associativity?

- $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$

- $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$

$$= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = \underline{0.0}$$

- $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$

- $= (0.0) + 1.0 = 1.0$

- **Therefore, Floating Point addition not associative!**

1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ is
still 1.5×10^{38}

FP result approximation of real result!

- **What are the conditions that make smaller arguments “disappear” (rounded down to 0.0)?**

Basic Addition Algorithm/Multiply issues

Addition (or **subtraction**) includes the following steps:

- (1) compute $Y_e - X_e$ (*getting ready to align binary point*)
- (2) right shift X_m that many positions to form $X_m \times 2^{X_e - Y_e}$
- (3) compute $(X_m \times 2^{X_e - Y_e}) + Y_m$

**Good
Summary**

if representation demands normalization, then normalization step follows:

- (4) left shift result, decrement result exponent (e.g., 0.001xx...)
right shift result, increment result exponent (e.g., 101.1xx...)
continue until MSB of data is 1 (NOTE: **Hidden** bit in IEEE Standard)
- (5) for **Multiply**, doubly biased exponent must be corrected:

$$X_e = 7$$

$$Y_e = -3$$

Excess 8 extra subtraction step of the bias amount

- (6) if result is 0 mantissa, may need to zero exponent by special step

$$\begin{array}{r} X_e = 1111 \\ Y_e = 0101 \\ \hline 10100 \end{array} \qquad \begin{array}{r} = 15 \\ = 5 \\ \hline 20 \end{array} \qquad \begin{array}{r} = 7 + 8 \\ = -3 + 8 \\ \hline 4 + 8 + 8 \end{array}$$

Rounding and IEEE Rounding Modes

- When we perform math on “real” numbers, we have to worry about rounding to fit the result in the significant field.
- The FP hardware carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting a double to a single precision value, or converting a floating point number to an integer

Round towards $+\infty$

- ALWAYS round “up”: $2.001 \rightarrow 3$
- $-2.001 \rightarrow -2$

Round towards $-\infty$

- ALWAYS round “down”: $1.999 \rightarrow 1$,
- $-1.999 \rightarrow -2$

Truncate

- Just drop the last bits (round towards 0)

Round to (nearest) even

- Normal rounding, almost

Round to Even

- Round like you learned in grade school
- Except if the value is right on the borderline, in which case we round to the nearest **EVEN** number

2.5 -> 2

3.5 -> 4

- **Insures fairness on calculation**

This way, half the time we round up on tie, the other half time we round down

Ask statistics majors

- **This is the default rounding mode**

Summary: Extra Bits for Rounding

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt."

How many extra bits?

IEEE: As if computed the result exactly and rounded.

Addition:

$1.xxxxxx$	$1.xxxxxx$	$1.xxxxxx$
$+ 1.xxxxxx$	$0.001xxxxx$	$0.01xxxxxx$
<hr/>	<hr/>	<hr/>
$1x.xxxxxy$	$1.xxxxxxyyy$	$1x.xxxxxyyy$
post-normalization	pre-normalization	pre and post

- **Guard Digits:** digits to the right of the first p digits of significand to guard against loss of digits – can later be shifted left into first P places during normalization.
- Addition: carry-out shifted in
- Subtraction: borrow digit and guard
- Multiplication: carry and guard, Division requires guard

Summary: Rounding Digits

Normalized result, but some non-zero digits to the right of the significand --> the number should be rounded

E.g., $B = 10$, $p = 3$:

0	2	1.69	=	1.6900 * 10	2-bias
-	0	0 7.85	=	- .0785 * 10	2-bias
	0	2 1.61	=	1.6115 * 10	2-bias

one round digit must be carried to the right of the guard digit so that after a normalizing left shift, the result can be rounded, according to the value of the round digit

IEEE Standard: four rounding modes:

- round to nearest even (default)
- round towards plus infinity
- round towards minus infinity
- round towards 0

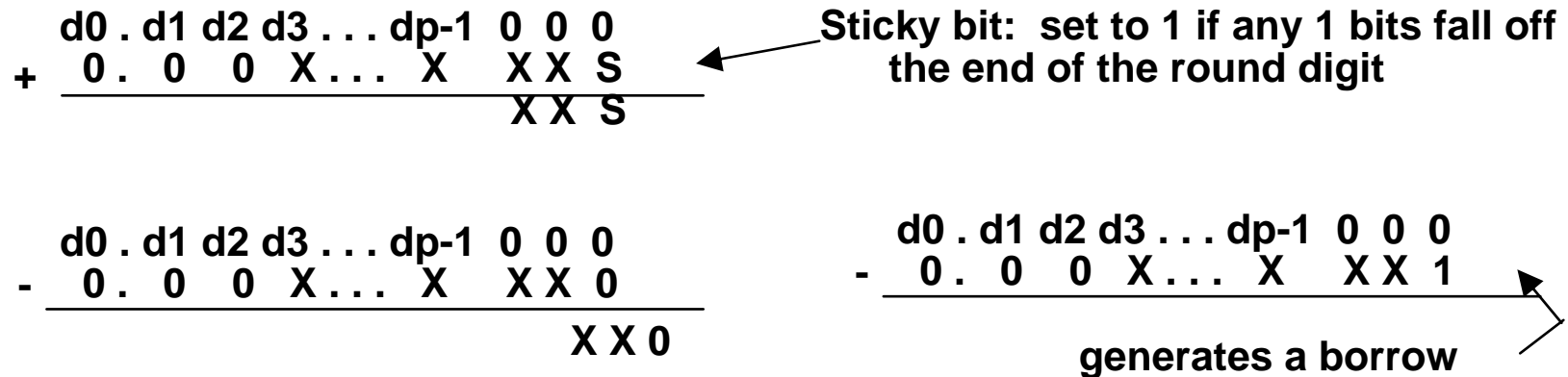
round to nearest:

- round digit $< B/2$ then truncate
- $> B/2$ then round up (add 1 to ULP: unit in last place)
- $= B/2$ then round to nearest even digit

it can be shown that this strategy minimizes the mean error introduced by rounding

Elaboration: Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding



Rounding Summary

Radix 2 minimizes wobble in precision

Normal operations in +,-,*,/ require one **carry/borrow** bit + one **guard** digit

One **round** digit needed for correct rounding

Sticky bit needed when round digit is B/2 for max accuracy

Rounding to nearest has mean error = 0, if *uniform distribution* of digits are assumed

C: Casting floats to ints and vice versa

- **(int) *floating point exp***

Coerces and converts it to the nearest integer (C uses truncation)

```
i = (int) (3.14159 * f);
```

- **(float) *exp***

converts integer to nearest floating point

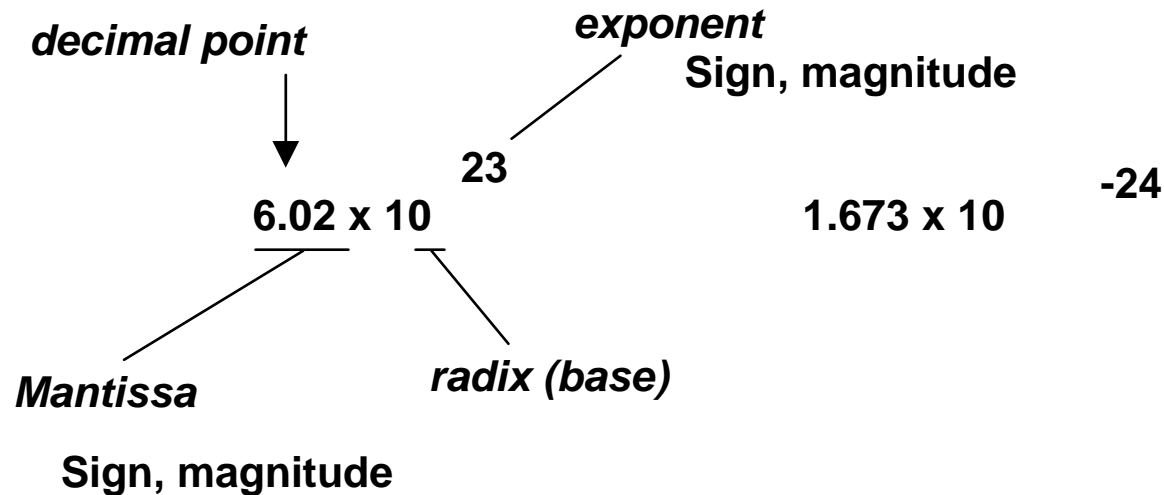
```
f = f + (float) i;
```

C: float -> int -> float

```
if (f == (float)((int) f)) {  
    printf("true");  
}
```

- Will not always print "true"
- Large values of integers don't have exact floating point representations
- What about double?
- Small floating point numbers (<1) don't have integer representations
- For other numbers, rounding errors

Summary: Scientific Notation



IEEE F.P. $\pm 1.M \times 2^{e - 127}$

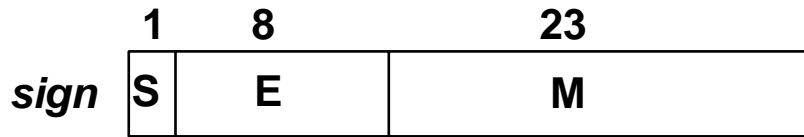
• Issues:

- Arithmetic (+, -, *, /)
- Representation, Normal form
- Range and Precision
- Rounding
- Exceptions (e.g., divide by zero, overflow, underflow)
- Errors
- Properties (negation, inversion, if $A \neq B$ then $A - B \neq 0$)

Summary : Floating-Point Arithmetic

Representation of floating point numbers in IEEE 754 standard:

single precision



exponent:
excess 127
binary integer

mantissa:
sign + magnitude, normalized
binary significand w/ hidden
integer bit: 1.M

actual exponent is
 $e = E - 127$

$$N = (-1)^S 2^{E-127} (1.M) \quad 0 < E < 255$$

$$0 = 0 \mathbf{00000000} 0 \dots 0 \quad -1.5 = 1 \mathbf{01111111} 10 \dots 0$$

Magnitude of numbers that can be represented is in the range:

$$2^{-126} (1.0) \quad \text{to} \quad 2^{127} (2 - 2^{-23})$$

which is approximately:

$$1.8 \times 10^{-38} \quad \text{to} \quad 3.40 \times 10^{38}$$

(integer comparison valid on IEEE Fl.Pt. numbers of same sign!)

Things to Remember

- **Floating Point numbers *approximate* values that we want to use.**
- **IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers**
- **New MIPS registers(\$f0-\$f31), instructions:**
 - Single Precision (32 bits, $2 \times 10^{-38} \dots 2 \times 10^{38}$): `add.s`,
`sub.s`, `mul.s`, `div.s`
 - Double Precision (64 bits, $2 \times 10^{-308} \dots 2 \times 10^{308}$): `add.d`,
`sub.d`, `mul.d`, `div.d`
- **Type is not associated with data, bits have no meaning unless given in context**