Disclaimer

The slides presented here are a combination of the CS251 course notes from previous terms, the work of Xiao-Bo Li, and material from the required textbook “Computer Organization and Design, ARM Edition,” by David A. Patterson and John L. Hennessy. It is being used here with explicit permission from the authors.

CS251 course policy requires students to delete all course files after the term. Therefore, please do not post these slides to any website or share them.

CS251 - Computer Organization and Design

Binary Multiplication and Floating Point Numbers

Instructor: Zille Huma Kamal

University of Waterloo

Spring 2023

Objective:

- Binary Multiplication
- Simplified hardware to implement unsigned binary multiplication
- IEEE Floating Point Number Representation
- Floating point arithmetic

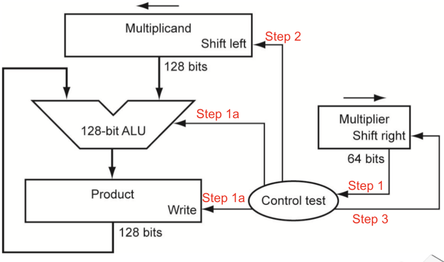
Unsigned Binary Multiplication

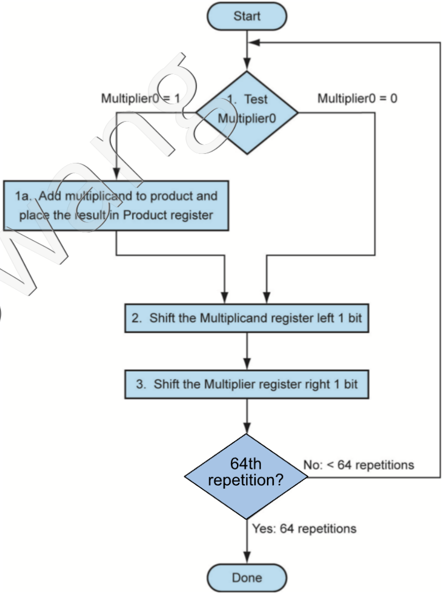
- Example:

13					1	1	0	1	Multiplicand
11					1	0	1	1	Multiplier
					1	1	0	1	
				1	1	0	1		
			0	0	0	0			
		1	1	0	1				
143	1	0	0	0	1	1	1	1	Product

- Note: for n bit numbers, result may be $2n$ bits

Simple Multiplication Hardware and Flowchart





- Initialization and termination not shown
- At start, Product is zero, 64-bit Multiplicand in right half of its register
- Note control inputs and outputs

4-Bit Multiplication Example, Simple HW Version

Multiplier = 1011, Multiplicand = 1101

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	1011	0000 1101	0000 0000
1	Add mpcd to prod			0000 1101
	Shift left mpcd		0001 1010	
	Shift right mplr	0101		
2	Add mpcd to prod			0010 0111
	Shift left mpcd		0011 0100	
	Shift right mplr	0010		
3	No operation			
	Shift left mpcd		0110 1000	
	Shift right mplr	0001		
4	Add mpcd to prod			1000 1111
	Shift left mpcd		1101 0000	
	Shift right mplr	0000		

Multiplier = 11, Multiplicand = 13 and Product = 143 = 1000 1111

Example: Unsigned Binary Multiplication

Try this

Compute 6×12 using the simple multiplication hardware discussed in class.

Hints for solution:

1. What is 6_{10} as a 4-bit unsigned binary?
2. What is 12_{10} as a 4-bit unsigned binary?
3. Say, multiplicand register is 6 in unsigned binary and multiplier register is unsigned binary of 12.
4. Follow the flowchart with number of repetitions set to 4 (since 4-bit numbers)

Solution: 4-Bit Unsigned Binary Multiplication

Multiplier $= 6_{10} = 0110_2$, Multiplicand $= 12_{10} = 1100_2$

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	0110	0000 1100	0000 0000
1	No operation Shift left mpcd Shift right mplr	 0011	 0001 1000 	
2	Add mpcd to prod Shift left mpcd Shift right mplr	 0001	 0011 0000 	0001 1000
3	Add mpcd to prod Shift left mpcd Shift right mplr	 0000	 0110 0000 	0100 1000
4	No operation Shift left mpcd Shift right mplr	 0000	 1100 0000 	

Multiplier = 6, Multiplicand = 12 and Product = 72 = 0100 1000

How to Represent Real Numbers

- Real numbers: $\pi, \sqrt{2}, \dots$. In contrast to natural numbers $(0, 1, 2, \dots)$ and integers $(\dots, -2, -1, 0, 1, 2, \dots)$
- Use scientific notation to represent real numbers
- Recall, scientific notation
  - Normalized scientific notation: single non-zero digit to the left of the decimal point $-3.45 \times 10^3 \equiv -3450$
  - Numbers represented this way do not have a fixed decimal point position, they are called **floating point (FP)**.
- FP numbers on a computer are approximations of real numbers
- For computers, natural to use base 2
- Binary fractions work just like decimal
- Example 1: $1.01_2 \times 2^4 \equiv 10100_2 = 1 \times 2^4 + 1 \times 2^2 = 20_{10}$
- Example 2:

$$\begin{aligned} 1.01_2 &= 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} \\ &= 1 + 1 \times \frac{1}{4} \\ &= \frac{5}{4} \\ &= 1.25_{10} \end{aligned}$$

Example: Binary to Decimal FPs

Think About It

Binary fraction conversion is the same as for integers:

$$(1.01)_2 = 1(2^0) + 0(2^{-1}) + 1 \times 2^{-2} = 1 + \frac{1}{4} = \left(\frac{5}{4}\right)_{10}.$$

Therefore, the binary floating point number can be converted to decimal:

$$(1.01)_2 \times 2^4 = \left(\frac{5}{4}\right)_{10} \left(2^4\right) = 1.25 \times 16 = 20.$$

Terminology

- In scientific notation $-\mathbf{1}.010 \times 2^{3}$
- Sign, fraction, exponent
- Significand (mantissa) $= \mathbf{1} + 0.$fraction
- ARM uses the IEEE 754 floating-point standard format

31	30 – 23	22 – 0
s	exponent	fraction
1 bit	8 bits	23 bits

Single Precision FP

A FP representation must compromise between the fraction and exponent:

- more bits for the fraction increases **precision**,
- more bits for the exponent increases **range**.

- Single precision approximately allows numbers from 2.0×10^{-38} to 2.0×10^{38}
- Double precision: uses two 32-bit words, 11 bits for exponent, 52 bits for significand

Think About It

Think About It

Why `E` before `F`?

Think About It

What does the IEEE 754 FP format resemble?

Think About It

In computations, what binary number representation should we use to interpret `E` and `F`, unsigned, sign and magnitude or 2's complement?

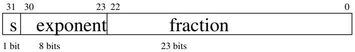
Floating Point Representation

- We want to compare FPs using integer comparisons.
- The sign bit is the most significand bit.
  - We can quickly test $FP < 0$, $FP > 0$, $FP == 0$.
- Place the exponent before the the fraction. This makes numbers with bigger exponents look larger than numbers with smaller exponents.
- If two's complement is used for exponent, then negative exponents look like big numbers. So instead, **bias notation** is used.
  - The most negative exponent is all 0's.
  - The most positive exponent is all 1's.
  - *Bias* for single precision is $(0111\ 1111)_2 = 2^7 - 1 = 127$.
  - *Bias* for double precision is $(011\ 1111\ 1111)_2 = 2^{10} - 1 = 1023$.

Biased Notation

- Exponent is stored in “biased” notation
- Exponent is interpreted as an unsigned binary integer
- The value is represented as
  $(-1)^{S} \times (1 + \text{fraction}) \times 2^{(\text{exponent}-\text{bias})}$,
  where bias $= 127$ for single precision
- So, in -1.010×2^{-1} the exponent of -1 is represented with a bit pattern of $-1 + 127 = 126$. That is, in the representation above exponent refers to the binary representation in the FP number. And $\texttt{exponent} - 127 = -1 = -1 + 127$
- Special cases:

E	F	FP value
0	0	0
all 1's	0	$\pm\infty$
all 1's	nonzero	Not a Number



- Exponent $= 0000\,0000$ reserved for 0
- Invalid operations such as $0/0$ or $\infty - \infty$ will produce result “Not a Number (NaN)”.

Example: Biased Notation

Try this

Complete the following table, showing the sign and magnitude and the corresponding biased representations with a bias of 127.

Sign and Magnitude	Biased Notation
-100	
-77	
-106	
27	
-50	

Solution: Biased Notation

Sign Magnitude		Biased Notation
-100	$-\mathbf{100} + 127$	27
-77	$-\mathbf{77} + 127$	50
-106	$-\mathbf{106} + 127$	21
27	$\mathbf{27} + 127$	154
-50	$-\mathbf{50} + 127$	77

Floating Point Overflow

- Overflow: Exponent is too large to be represented in the exponent field.
- Underflow: Negative exponent is too large to be represented in the exponent field

ARM will throw an exception.

Double precision reduces overflow and underflow compared to single precision.

Fractional Numbers

- How to represent numbers less than 1?
- Digits to right of decimal point represent negative powers of two

0.	1	0	1	1
1	1/2	1/4	1/8	1/16

  0*1 + 1*1/2 + 0*1/4 + 1*1/8 + 1*1/16 = 11/16
- Simple examples
  1/2 = 0.1
  3/4 = 0.11
- May have to approximate
  Example: 1/3 as decimal is...
  0.1 in binary is...
  sqrt(2) in binary is...

Basic Conversion Examples

Since $(1)_2 = (1)_{10}$, we may sometimes omit showing the step that changes the base for the number 1.

$$(0.0011)_2 = 1(2^{-3}) + 1(2^{-4}) = \frac{1}{8} + \frac{1}{16} = \left(\frac{3}{16}\right)_{10}$$

Basic Conversion Examples

Binary floating point single precision to decimal:

$$
\begin{aligned}
S &= 1, \\
E &= 10000001 = 2^7 + 1 = 129, \\
F &= 01000000000000000000000 = 2^{-2} = 0.25 \\
FP &= (-1)^1 \times (1 + F) \times 2^{E-127} \\
&= (-1)(1 + 0.25) \times 2^{129-127} \\
&= -(1.25) \times 2^2 \\
&= -1.25 \times 4 \\
&= (-5)_{10}
\end{aligned}
$$

An Algorithm for Converting From a Decimal to a Binary

Multiple the decimal by 2 to get a result d, take the value before the decimal in d as our binary digit.

Repeat with the remaing decimal digits.

Stop when the digit to the right of the decimal point is 0. That is, $d = 1.0$.

Decimal to a Binary: Example 1

Consider $(0.625)_{10}$:

$$0.625 \times 2 = \mathbf{1}.25 \text{ answer so far: } 0.1$$
$$0.25 \times 2 = \mathbf{0}.5 \text{ answer so far: } 0.10$$
$$0.5 \times 2 = \mathbf{1}.0 \text{ answer so far: } 0.101$$

Therefore, $(0.625)_{10} = (0.101)_2 = (1.01)_2 \times 2^{-1}$.
In IEEE 754 FP: $1.01 \times 2^{exp-127=-1} = 1.01 \times 2^{126}$
This number is stored as a 32 bit word.

```
S: 0                                // positive
E: 0111 1110                        // exponent stored is 126
F: 0100 0000 0000 0000 0000 000 // fraction
----------------------------------------
S:E:F
0 0111 1110 0100 0000 0000 0000 0000 000
```

Decimal to a Binary: Example 2

Consider $(0.1)_{10}$:

$0.1 \times 2 = \mathbf{0}.2$ answer so far: 0.0

$0.2 \times 2 = \mathbf{0}.4$ answer so far: 0.00

$0.4 \times 2 = \mathbf{0}.8$ answer so far: 0.000

$0.8 \times 2 = \mathbf{1}.6$ answer so far: 0.0001

$0.6 \times 2 = \mathbf{1}.2$ answer so far: 0.00011

$0.2 \times 2 = \mathbf{0}.4$ answer so far: 0.000110 //above pattern repeats

$0.4 \times 2 = \mathbf{0}.8$ answer so far: 0.0001100

$0.8 \times 2 = \mathbf{1}.6$ answer so far: 0.00011001

$0.6 \times 2 = \mathbf{1}.2$ answer so far: 0.000110011

Therefore, $(0.1)_{10} = (0.0\overline{0011}) = 1.1\overline{0011} \times 2^{-4}$. Since this number is an infinite decimal, the computer has to approximate it. For a large number of precision (fraction) bits, we get very very close to 0.1.

Decimal to a Binary: Example 3

Consider $(1/3)_{10} = (0.\overline{3})_{10}$:

$$0.\overline{3} \times 2 = \mathbf{0}.\overline{6} \text{ answer so far: } 0.0$$
$$0.\overline{6} \times 2 = \mathbf{1}.\overline{3} \text{ answer so far: } 0.01$$
$$0.\overline{3} \times 2 = \mathbf{0}.\overline{6} \text{ answer so far: } 0.010$$
$$0.\overline{6} \times 2 = \mathbf{1}.\overline{3} \text{ answer so far: } 0.0101$$

Therefore, $(0.\overline{3})_{10} = (0.\overline{01})_2 = (1.\overline{01})_2 \times 2^{-2}$.

For a number like $\sqrt{2}$ which has infinite decimals, regardless of the base, it has to be approximated.

Decimal to a Binary: Example 4

Consider $(2.625)_{10} = (2 + 0.625)_{10}$:

- The integer part is $(2)_{10} = (10)_2$.
- The fraction part is $(0.625)_{10} = (0.101)_2$, this was calculated previously.

Therefore,

$$\begin{aligned}(2 + 0.625)_{10} &= (10 + 0.101)_2 \\ &= (10.101)_2 \\ &= (1.0101)_2 \times 2^1 \\ &= (1.0101)_2 \times 2^{128-127}\end{aligned}$$

This number is stored as a 32 bit word.

```
S: 0                            // positive
E: 1000 0000                    // exponent stored is 128
F: 0101 0000 0000 0000 0000 000 // fraction
--------------------------------------
S:E:F
0 1000 0000 0101 0000 0000 0000 0000 000
0100 0000 0010 1000 0000 0000 0000 0000 // group by 4 bits
```

Decimal to a Binary: Example 5

Consider $(625.65625)_{10} = (625 + 0.65625)_{10}$:

- The integer part is

$$625_{10} = 2^9 + 2^6 + 2^5 + 2^4 + 2^0 = (10\ 0111\ 0001)_2$$

- The fraction part is $(0.65625)_{10}$:

$$\begin{aligned}
0.65625 \times 2 &= \mathbf{1}.3125 \text{ answer so far: } 0.1 \\
0.3125 \times 2 &= \mathbf{0}.625 \text{ answer so far: } 0.10 \\
0.625 \times 2 &= \mathbf{1}.25 \text{ answer so far: } 0.101 \\
0.25 \times 2 &= \mathbf{0}.5 \text{ answer so far: } 0.1010 \\
0.5 \times 2 &= \mathbf{1}.0 \text{ answer so far: } 0.10101
\end{aligned}$$

Therefore, $(0.65625)_{10} = (0.10101)_2$

Finally, $(10\ 0111\ 0001.10101)_2 = (1.00111000110101)_2 \times 2^9$.

Example 5 continued

Previously, we found

$$(625 + 0.65625)_{10} = 1.00111000110101 \times 2^9.$$

Now we will store this as a 32 bit word.

$$\begin{aligned} S &= 0 \\ E &= 9 + 127 = 136 = 2^7 + 2^3 = (10001000)_2 \\ F &= 00111000110101 \end{aligned}$$

Therefore, our word is:

```
0 10001000 00111000110101000000000
-----------------------------------------
0100 0100 0001 1100 0110 1010 0000 0000 //group by 4
```

Reading the Bit Pattern

Consider a 32 bit FP that is stored as the following bit pattern:

```
0011 1111 1101 0100 0000 0000 0000 0000
S=0
E=0111 1111
F=1010 1000 0000 0000 0000 000
```

$E = 127 - 127 = 0$ and $F = 0.10101$. So our binary number is:

$$\mathbf{1}.10101 \times 2^0 = 1.10101 = (1.65625)_{10}$$

Review: IEEE 754 Single Precision Representation of Floating Point Numbers

Think About It

IEEE 754 FP Exponent[a]	Conversion	Exponent in Scientific Notation	Note
0000 0000	$0 - 127$	-127	reserved
0000 0001	$1 - 127$	-126	$S^b \times 2^{-126}$
0101 0101	$85 - 127$	-42	$S \times 2^{-42}$
1111 1111	$255 - 127$	128	reserved
0111 1111	$127 - 127$	0	$S \times 2^{0}$
1100 1010	$202 - 127$	75	$S \times 2^{75}$
1111 1110	$254 - 127$	127	$S \times 2^{127}$

[a] 8 bits: interpret as unsigned binary (never negative numbers)
[b] where S is a significant (1.Fraction)

Note: The exponent in scientific notation can be between -126 and $+127$.

Floating-Point Addition

- Decimal example: $9.54 \times 10^2 + 6.83 \times 10^1$
  (assume we can only store two digits to right of decimal point)
  1. Match exponents: $9.54 \times 10^2 + .683 \times 10^2$
  2. Add significands, with sign: 10.223×10^2
  3. Normalize: 1.0223×10^3
  4. Check for exponent overflow/underflow
  5. Round: 1.02×10^3
  6. May have to normalize again
- Same idea works for binary

Example 1: Binary Floating Point Addition

Try this

Compute $0.5_{10} + (-0.4375)_{10}$ using binary floating point numbers.

Convert to binary floating points:

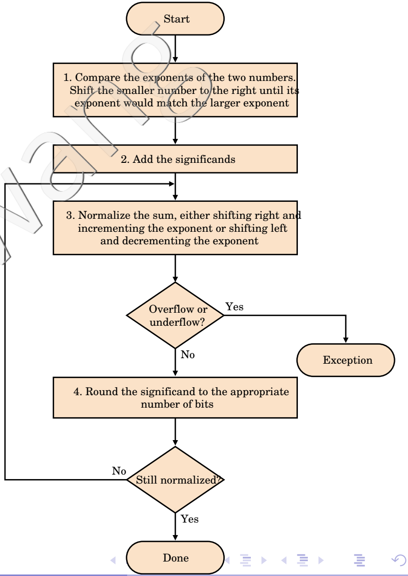
$$(0.5)_{10} = (1.000)_2 \times 2^{-1}$$
$$(-0.4375)_{10} = (-1.110)_2 \times 2^{-2}$$

Assume we only have 4 bits in hardware. Therefore we only keep four bits of the significand, this means 3 bits to the right of the binary point.

Solution: Binary Floating Point Addition

$(1.000)_2 \times 2^{-1} + [(-1.110)_2 \times 2^{-2}]$

1. $(-1.110)_2 \times 2^{-2} = -0.111 \times 2^{-1}$.
2. $1.000 - 0.111 = 0.001$.
3. $0.001 \times 2^{-1} = 1.0 \times 2^{-4}$.
4. No underflow, since $127 \geq -4 \geq -126$ Single precision biased exponents are between 1 and 254.
5. No rounding needed $(1.000)_2 \times 2^{-4}$ does not need rounding since (1.000) is exaclty 4 bits.
6. Sum is normalized.



Example 2: Binary Floating Point Addition

Try this

Compute $7.125 + 1.6015625$ using binary floating point numbers.

Convert to binary floating points:

$$\begin{aligned}(7.125)_{10} &= 7_{10} + 0.125_{10}\\ &= 111_2 + 0.001_2\\ &= 111.001_2\\ &= 1.11001 \times 2^2\end{aligned}$$

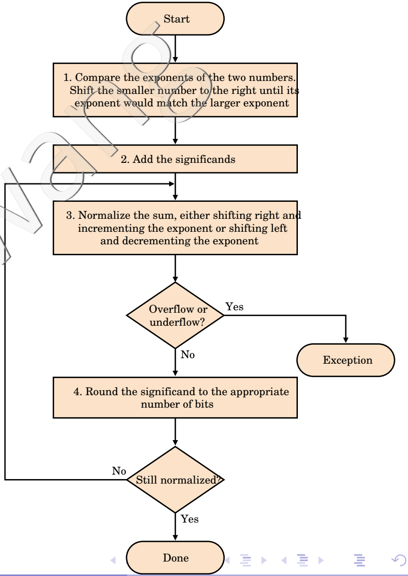
$$\begin{aligned}(1.6015625)_{10} &= 1_{10} + 0.6015625_{10}\\ &= 1_2 + 0.1001101_2\\ &= 1.1001101_2\\ &= 1.1001101 \times 2^0\end{aligned}$$

Assume IEEE 754 Floating Point representation, therefore, 22 bits for fraction.

Solution: Binary Floating Point Addition

$1.11001 \times 2^2 + 1.1001101 \times 2^0$

1. $1.1001101 \times 2^0 = 0.011001101 \times 2^2$
2. $1.11001 + 0.011001101 = 10.001011101$
3. Normalize: 10.001011101. $10.001011101 = 1.0001011101 \times 2^1 \times 2^2 = 1.0001011101 \times 2^3$.
4. No overflow. Since $127 \geq 3 \geq -126$, there is no under or overflow. Single precision biased exponents are between 1 and 254.
5. No rounding needed.
6. Sum is still normalized.



Solution: Check Results of Binary Floating Point Addition

Convert binary result to decimal:

$$\begin{aligned}
1.11001 \times 2^2 + 1.1001101 \times 2^0 &= 1.0001011101 \times 2^3 \\
&= 1 + \frac{1}{16} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256} + \frac{1}{1024} \\
&= (1 + \frac{93}{1024}) \times 8 \\
&= 8.7265625
\end{aligned}$$

Check against decimal results:

$$\begin{aligned}
&7.125 + 1.6015625 \\
&\quad = 8.7265625_{10} \\
&\quad = 8_{10} + 0.7265625_{10} \\
&\quad = 1000_2 + 0.1011101_2 \\
&\quad = 1000.1011101 \\
&\quad = 1.0001011101 \times 2^3
\end{aligned}$$

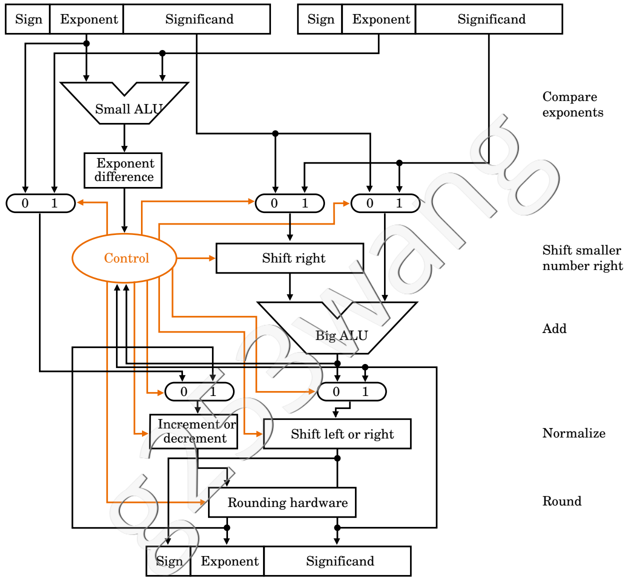


Figure 3.15, Hardware for Floating-Point Addition

Example: Use of hardware for FP Addition

a-b = a+!b+1

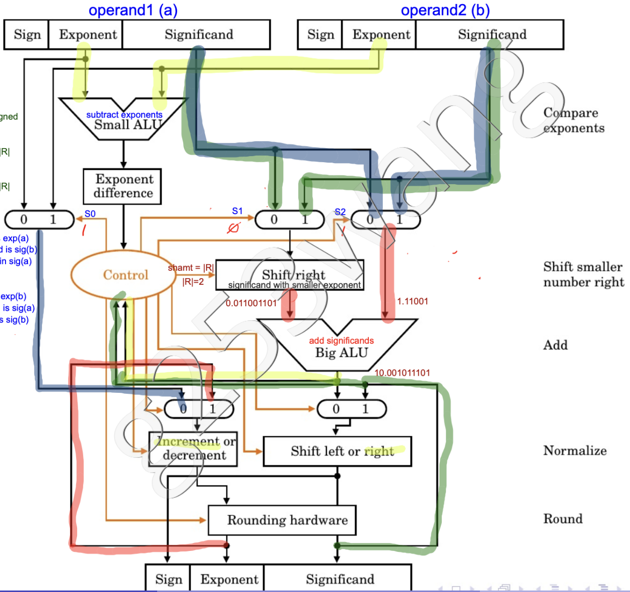
if result (R) = 0, exponents are aligned

if R > 0 (a > b)
then shift significand of b right by |R|

if R < 0 (a < b)
then shift significand of b right by |R|

if R>0 (a>b)
S0 = 0 larger exponent is exp(a)
S1 = 1 smaller significand is sig(b)
S2 = 0 larger significand in sig(a)

if R<0 (a<b)
S0 =1 larger exponent is exp(b)
S1= 0 smaller significand is sig(a)
S2 = 1 larger significant is sig(b)



Let $a = 1.1001101 \times 2^0$ anc
$b = 1.11001 \times 2^2$. Then $a + b$

1. Small ALU $= 0 - 2 = -2$,
Then $R < 0$
2. Then, $S0 = 1, S1 = 0$ and
$S2 = 1$
3. This implies,
$1.11001 + 0.011001101 =$
10.001011101
4. Normalize Sum.
$10.001011101 =$
$1.0001011101 \times 2^1 \times 2^2 =$
1.0001011101×2^3.
5. No rounding needed. No
Overflow.

Floating-Point Multiplication

- Decimal example: $(9.54 \times 10^2) \times (6.83 \times 10^1)$
  (assume we can only store two digits to right of decimal point)
  1. Add exponents: $2 + 1 = 3$
     (Note: exponents stored in biased notation)
  2. Multiply significands: $9.54 \times 6.83 = 65.1582$
  3. Unnormalized result: 65.1582×10^3
  4. Normalize: 6.51582×10^4
  5. Check for overflow/underflow
  6. Round: 6.52×10^4
     (May need to renormalize)
  7. Set sign
- Same idea works for binary

FP Multiplication Example

We will multiply the following two numbers

$$(0.5)_{10} = (1.000)_2 \times 2^{-1}$$
$$(-0.4375)_{10} = (-1.110)_2 \times 2^{-2}$$

Assume we keep four bits of the significand, this means 3 bits to the right of the binary point.

FP Multiplication Example

$(1.000)_2 \times 2^{-1} \times (-1.110)_2 \times 2^{-2}$.

1. Add the exponents: $-1 + (-2) = -3$.
   If we want to use the bias notation, the sum is

$$(-1 + 127) + (-2 + 127) - 127 = -3 + 127 = 124.$$

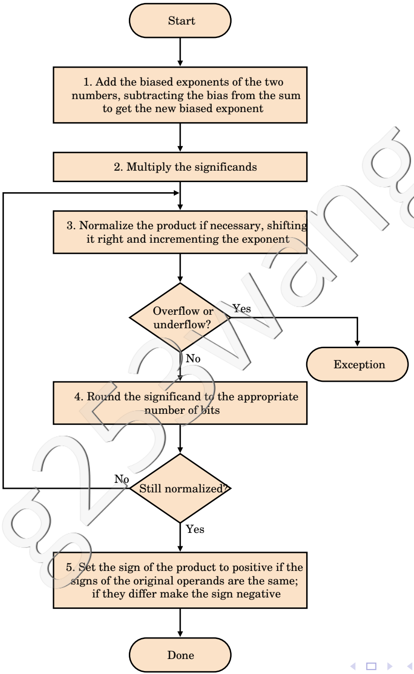
2. Multiply the significands.

$$\begin{array}{ccccccc}
 & & & 1. & 0 & 0 & 0 \\
 & & \times & 1. & 1 & 1 & 0 \\
\hline
 & & & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 0 & \\
 & 1 & 0 & 0 & 0 & & \\
1 & 0 & 0 & 0 & & & \\
\hline
1. & 1 & 1 & 0 & 0 & 0 & 0
\end{array}$$

The answer to four bits is $(1.110)_2 \times 2^{-3}$.

FP Multiplication Example

3. Make sure the product is normalized. $(1.110)_2 \times 2^{-3}$ is already normalized.
4. Check the exponent for overflow or underflow. Here $127 \geq -3 \geq -126$ so there is no overflow or underflow.
5. Round to four bits. The product $(1.110)_2 \times 2^{-3}$ does not require rounding.
6. Set the sign. This product has a negative sign, $-(1.110)_2 \times 2^{-3}$.


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Accuracy

- Only certain numbers can be represented accurately
- Typically the result of an operation cannot be represented precisely
- The result must be rounded. Multiple ways to round

For this class, we will round 1/2 up in magnitude

- Do we need to compute precisely and then round?
- Goal: save hardware by not keeping full precision internally during computation
- How few bits can be used to get correct n-bit result after rounding? Result should be the same as if we had kept full precision and rounded afterwards

Accuracy in Floating-Point Addition

- In adding two significands with n bits of precision, can use an n-bit adder (giving $n+1$-bit result)
- Is least significant bit of result enough to round correctly?
- Our addition examples will use $n = 4$

$$\begin{array}{rrrrrrrr} & & 1 & . & 0 & 1 & 0 & \times & 2^0 \\ + & & 1 & . & 1 & 1 & 1 & \times & 2^0 \\ \hline & 1 & 1 & . & 0 & 0 & 1 & \times & 2^0 \end{array}$$

  Hardware adder gives two bits to left of decimal point
- This is normalized to 1.1001×2^1 and then rounded to 1.101×2^1
- Problem may arise when one significand has to be shifted to match exponents

- Example: $1.010 \times 2^2 + 1.001 \times 2^1$
  After normalization, our input bits span range of $n + 1$ bits.
  How do we add this with an n-bit adder?
  Can we ignore low order bit?

$$
\begin{array}{cccccccccc}
 & & 1 & . & 0 & 1 & 0 & & \times & 2^2 \\
+ & & 0 & . & 1 & 0 & 0 & \boxed{1} & \times & 2^2 \\
\hline
 & 0 & 1 & . & 1 & 1 & 0 & & \times & 2^2
\end{array}
$$

  Note leftmost 0 is carry out of adder
- Here, boxed bit of second significand was not fed into adder
  But is boxed bit needed to round correctly?
- With it, normalized result is 1.1101×2^2, rounds to 1.111×2^2
- Without it, normalized result is 1.110×2^2
- Thus for n-bit accuracy, we need to keep $n + 2$ bits during the computation

Accuracy in Floating-Point Multiplication

- When multiplying two floating-point numbers, the significands are multiplied together
- If the significands have n bits of precision each, the result can have $2n$ bits of precision
- How many bits do we need to keep during the computation?
- Our multiplication examples will have $n = 3$

- Example:

$$\begin{array}{ccccccccc} & & 1 & . & 1 & 1 & & & \times\ 2^2 \\ \times & & 1 & . & 1 & 1 & & & \times\ 2^1 \\ \hline & 1 & 1 & . & 0 & 0 & 0 & 1 & \times\ 2^3 \end{array}$$

- In above example, only top 3 bits are needed for final result of 1.10×2^4
- Example: Do we need highlighted (fourth) bit?

$$\begin{array}{ccccccccc} & & 1 & . & 1 & 0 & & & \times\ 2^2 \\ \times & & 1 & . & 1 & 0 & & & \times\ 2^1 \\ \hline & 1 & 0 & . & 0 & \boxed{1} & 0 & 0 & \times\ 2^3 \end{array}$$

- With three bits, 10.0×2^3 is normalized to 1.00×2^4, which is incorrectly rounded
- With four bits, 10.01×2^3 is normalized to 1.001×2^4, and correctly rounded up to 1.01×2^4

Floating-Point Architectural Issues

- To maintain n bits of accuracy after an operation, preserve $n+2$ bits during the computation (the two extra bits are sometimes called *guard* and *round*)
- Separate floating-point registers?
- Separate floating-point coprocessors?
- Rounding or truncating?
- What to do about overflow (same issue as for integer arithmetic)?
- Less precision:

16-bit format	u	min	max
fp16	4.88×10^{-4}	6.10×10^{-5}	6.55×10^{4}
bfloat16	3.91×10^{-3}	1.18×10^{-38}	3.39×10^{38}

Further Study

Floating point numbers are an example of what you can study in the area of numerical computation, introduced in CS 370.

Textbook Readings

- Readings:
  - 3.3 (pages 191-194)
  - 3.5 (pages 205-220, 226-230)