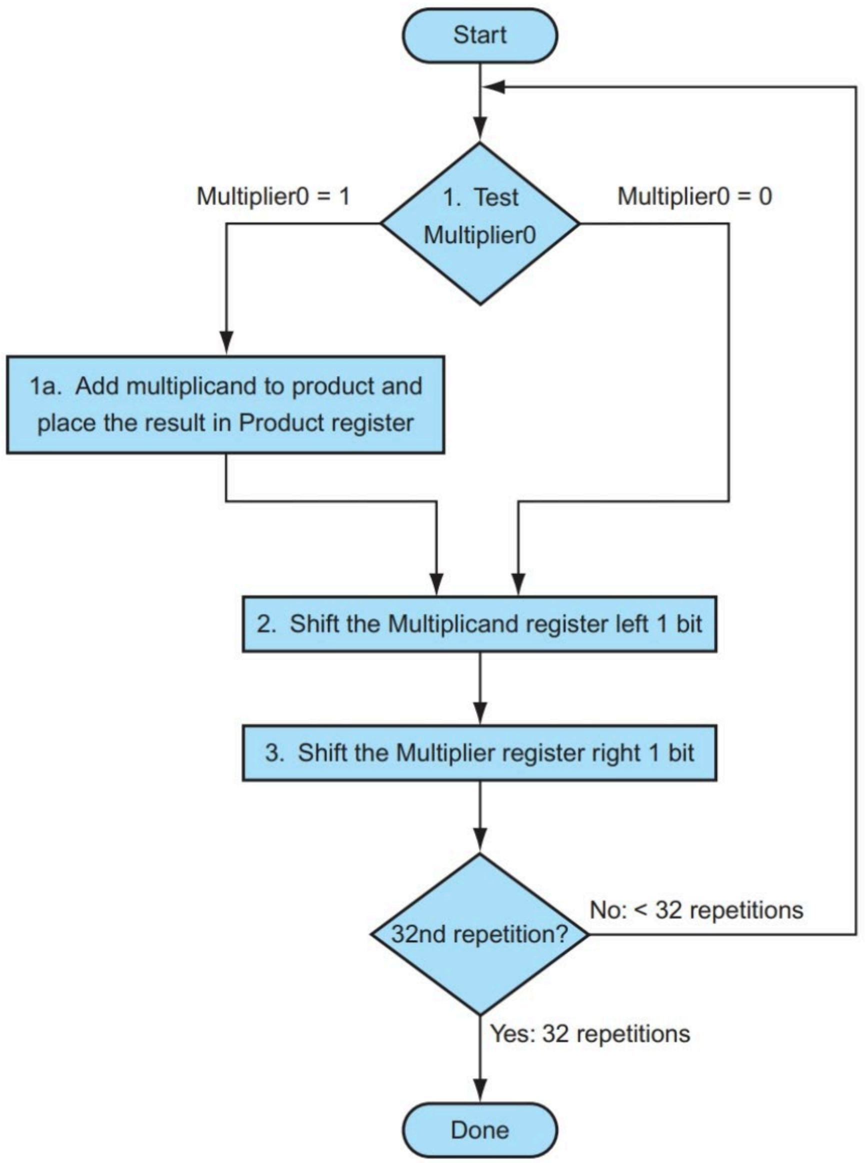
CPT_S 260 Intro to Computer Architecture
Lecture 9

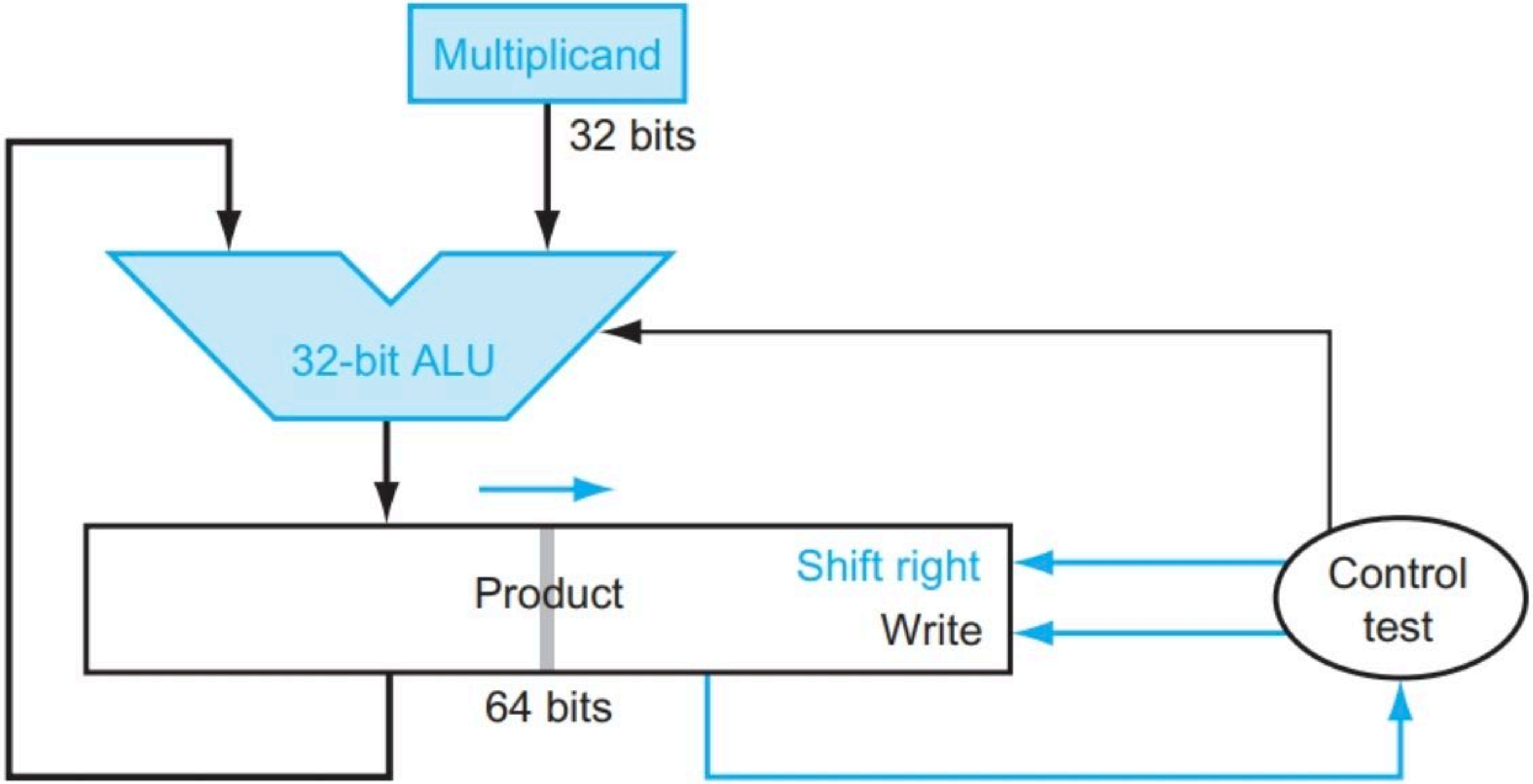
Floating Point Representation
January 31, 2022

Ganapati Bhat
School of Electrical Engineering and Computer Science
Washington State University

Multiplication Algorithm



Recap: Parallel Architecture



Conversion for Numbers with Fractions

- **In real mathematical operation, we have numbers with fractions**
  - Float and Double numbers in programing languages
- **We should take three steps:**
  - Convert the Integer Part (The same as integer numbers)
  - Convert the Fraction Part
  - Join the two results with a radix point

Fractional Part in Binary Format

- Repeatedly multiply the fraction by 2 and save the resulting integer digits. The digits for the binary number are the 0,1 in order of their computation.

- Convert 46.6875 to binary!

Fractional Part in Binary Format

- **Note that in this conversion, the fractional part becomes 0 as a result of the repeated multiplications.**
- **In general, it may take many bits to get this to happen or it may never happen.**
- **Example: Convert 0.65 to binary!**
- **0.65 = 0.10100110011001 …**
- **The fractional part begins repeating every 4 steps yielding repeating 1001 forever!**
- **Solution: Specify number of bits to right of radix point and round or truncate to this number.**

Checking the Conversion

- **To convert back, sum the digits times their respective powers of r.**
- **From the prior conversion of 46.6875**

$$\begin{aligned}101110_2 &= 1\times 32 + 0\times 16 + 1\times 8 + 1\times 4 + 1\times 2 + 0\times 1\\ &= 32 + 8 + 4 + 2\\ &= 46\end{aligned}$$

$$\begin{aligned}0.1011_2 &= 1/2 + 1/8 + 1/16\\ &= 0.5000 + 0.1250 + 0.0625\\ &= 0.6875\end{aligned}$$

Normalized Numbers

- **A number in scientific notation that has no leading 0s is called a *normalized number***
- **Example:**
  - $1.0_{ten} \times 10^{-9}$ is in **normalized** scientific notation,
  - $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ **are not**
- **Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation**

$$\mathbf{1.0_2 \times 2^{-1}}$$

Binary point

Floating Point

- **Representation for non-integral numbers**
  - Including very small and very large numbers
- **Similar to scientific notation**
  - -2.34×10^{56} normalized
  - $+0.002 \times 10^{-4}$ not normalized
  - $+987.02 \times 10^{9}$
- **In binary**
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
  - Types float and double in C

Floating Point Standard

- **Defined by IEEE Std 754-1985**
- **Developed in response to divergence of representations**
  - Portability issues for scientific code
- **Now almost universally adopted**
- **Two representations**
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- **S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)**
- **Normalize significand: 1.0 ≤ |significand| < 2.0**
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- **Exponent: excess representation: actual exponent + Bias**
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Floating Point Representation – 2s Complement

- If we **use two's complement** or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field a negative exponent will look like a big number.
- For example, $1.0_{two} \times 2^{-1}$ would be represented as:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

- For example, $1.0_{two} \times 2^{+1}$ would be represented as:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Bias Exponent Representation

- 2's complement makes it difficult to compare exponents
- -1 is (111..111) where is 1 is (000….001). If we just look at 2's complement number, we cannot tell which has higher exponent
- IEEE 754 uses a bias of 127 for single precision, so an exponent of -1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$
- $+1 \rightarrow 1 + 127$, or $128_{ten} = 1000\ 0000_{two}$
- The exponent bias for *double* precision is 1023.

Single-Precision Range

- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
  - Exponent (also called biased exponent): 00000001 ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value**
  - Exponent (also called biased exponent): 11111110 ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- **Exponents 0000…00 and 1111…11 reserved (next slides show for what these exponents are used)**
  - Smallest value
  - Exponent: 00000000001 $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
  - Exponent: 11111111110 $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Special Cases

- **Zero**
  – Sign bit = 0; biased exponent = all 00 bits; and the fraction = all 00 bits;
- **Positive and Negative Infinity**
  – Sign bit = 00 for positive infinity, 11 for negative infinity; biased exponent = all 11 bits; and the fraction = all 00 bits;
- **NaN (Not-A-Number)**
  – Sign bit = 0 or 1; biased exponent = all 11 bits; and the fraction is anything but all 00 bits. NaN's occurs when one does an invalid operation on a floating point value, such as dividing by zero, or taking the square root of a negative number.

Therefore, for Infinities and NaNs, we have

- **Exponent = 111…1, Fraction = 000…0**
  – ±Infinity –sign bit =0 for (+); sign bit =1 for (-)
  – Can be used in subsequent calculations, avoiding need for overflow check
- **Exponent = 111…1, Fraction ≠ 000…0**
  – Not-a-Number (NaN)
  – Indicates illegal or undefined result
    » e.g., 0.0 / 0.0
  – Can be used in subsequent calculations

Floating-Point Precision

- **Relative precision**
  - All fraction bits are significant
- **Single: approx 2^{-23}**
  - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- **Double: approx 2^{-52}**
  - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example #1

- **Represent –0.75**
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = 1000…00
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110
    - Double: –1 + 1023 = 1022 = 01111111110
- **Single: 1011111101000…00**
- **Double: 1011111111101000…00**

Floating-Point Example #2

- **What number is represented by the single-precision float**
- **11000000101000…00**
  - S = 1
  - Fraction = 01000…00
  - Exponent = 10000001 = 129
- **$x = (-1) \times (1 + .01) \times 2^{(129-127)}$**
  - $= (-1) \times 1.25 \times 2^2$
  - $= -5.0$