



NYU

TANDON SCHOOL
OF ENGINEERING

Floating Point Arithmetic-

**IEEE 754 representation, Hardware
Implementation & RISC V Instructions**

Azeez Bhavnagarwala

ECE 6913 Computer Systems Architecture

Fall 2024

**NYU Tandon School of
Engineering**

Arithmetic for Computers

- How are *negative*, *fractions* and *real* numbers represented in a computer ?
- How *large or small can a number be and be operated on by a computer*
- How could a programmer *control the precision of arithmetic operations* executed by his/her program on any computer
- what happens *if an operation creates a number bigger than can be represented in the computer* ?
- How does RISC-V implement arithmetic operations ?
- How does computer hardware *multiply and divide numbers* ? How does a computer speed up these?

Unsigned Binary Numbers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$

- Example

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 32 bits

- 0 to +4,294,967,295

2s Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$

- Example

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Using 32 bits

- $-2,147,483,648$ to $+2,147,483,647$

2s Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers *have the same unsigned and 2s-complement representation*
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111 \dots 111_2 = -1$$
$$\bar{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000 \ 0000 \dots 0010_2$
 - $-2 = 1111 \ 1111 \dots 1101_2 + 1$
 $= 1111 \ 1111 \dots 1110_2$
 - reversing the bits in $-2 + 1$ yields
 - $1111 \ 1111 \dots 1110_2 + 1$
 - $= 0000 \ 0000 \dots 0010_2 = +2$

Overflow

- Occurs when *result of operation cannot be represented*
- Can occur when:
 - *Adding 2 operands with same sign*
 - *Subtracting an operand from another with different signs*
- Detect overflow when adding 2 operands, A, B:
 - A, B are positive, result is negative (most significant bit in 2's complement representation is 1) => overflow has occurred
 - A, B are negative, result is positive (most significant bit in 2's complement representation is 0) => overflow has occurred
- Detect overflow when subtracting A-B:
 - $A > 0$, $B < 0$, result is negative (most significant bit in 2's complement representation is 1) => overflow has occurred
 - $A < 0$, $B > 0$, result is positive (most significant bit in 2's complement representation is 0) => overflow has occurred
- With unsigned integers (memory addresses), overflow ignored
 - `add`, `addi`, `sub` cause exceptions on overflow
 - `addu`, `addiu`, `subu` do not cause exceptions on overflow

Overflow Exceptions/Interrupts

- What happens when overflow occurs ?
- C & Java ignore integer overflows
- MIPS detects overflow with *exceptions* or *interrupts*
 - Exception is unscheduled procedure call
 - address of *instruction that overflowed* saved in register
 - execution *jumps* to *predefined address* to invoke a routine for that exception
 - *interrupted address* saved so program can resume after corrective code is executed

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 above cannot fit in a 32-bit integer
- We use scientific notation to represent
 - very small numbers (e.g., 1.0×10^{-9})
 - very large numbers (e.g., 8.64×10^{13})
 - Scientific notation: $\pm d.f_1f_2f_3f_4\dots \times 10^{\pm e_1e_2e_3\dots}$

Fractions & Real # representation

- base r numbers:
- digits used: $\{0, 1, 2, \dots, r-1\}$

$$\underbrace{(d_n d_{n-1} \dots d_0)}_{\text{Integer part}} \cdot \underbrace{d_{-1} d_{-2} \dots d_{-m}}_{\text{Fractional part}})_r =$$

$$d_n \times r^n + d_{n-1} \times r^{n-1} + \dots + d_1 \times r^1 + d_0 \times r^0 + d_{-1} \times r^{-1} + d_{-2} \times r^{-2} + \dots + d_{-m} \times r^{-m}$$

Example

1. $r=10$

$$523.61 = 5 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 6 \times 10^{-1} + 1 \times 10^{-2}$$

2. $r=2$

$$1011.11 = 2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-2} = (11.75)_{10}$$

Convert from Decimal to base r

- $(531.375)_{10} = (x)_8$?
- Treat the integer part and the fraction part separately.

531		3
66		2
8		0
1		1
0		↑

remainder of
division by 8

.375		
3.000		3

integer part of
multiplying by 8

$x=1023.3$

More Examples

■ $r=3$

■ $(48.5)_{10} = (x)_3 ?$

$$\begin{array}{r|l} 48 & 0 \\ 16 & 1 \\ 5 & 2 \\ 1 & 1 \\ 0 & \end{array} \uparrow$$

$$\begin{array}{r|l} .5 & \\ \textcircled{1.5} & 1 \\ \textcircled{1.5} & 1 \\ \textcircled{1.5} & 1 \\ \vdots & \end{array} \downarrow$$

$$x = 1210.1111\dots = 1210.\overline{1}$$

■ $r=5$

■ $(76.85)_{10} = (x)_5 ?$

$$\begin{array}{r|l} 76 & 1 \\ 15 & 0 \\ 3 & 3 \\ 0 & \end{array} \uparrow$$

$$\begin{array}{r|l} .85 & \\ \textcircled{4.25} & 4 \\ \textcircled{1.25} & 1 \\ \textcircled{1.25} & 1 \\ \vdots & \end{array} \downarrow$$

$$x = 301.411\dots = 301.4\overline{1}$$

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}
- Examples of floating-point numbers in base 2...
 - 1.00101×2^{23} , 0.0100101×2^{25} , -1.101101×2^{-3} , -1101.101×2^{-6}
 - **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 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, 1 is negative)
 - Representation is called sign and magnitude
 - **E** is the Exponent field (signed)
 - Very large numbers have 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 the binary point)
 - More bits in fraction field improves the precision of FP numbers

S	Exponent	Fraction
----------	-----------------	-----------------

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

IEEE 754 Floating-Point Standard

- Found virtually in 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 bits	Fraction 23 bits
----------	------------------------	-------------------------

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

S	Exponent 11 bits	Fraction 52 bits
----------	-------------------------	-------------------------

Normalized Floating Point Numbers

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



- Significand** is equal to $(1.F)_2 = (1.f_1f_2f_3\dots)_2$
 - 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

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

$(-1)^S$ 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 = $\text{val}(E) = E - \text{Bias}$ (**Bias** is a constant)
- Recall that exponent field is **8 bits** for **single precision**
 - **E** can be in the range of **0 to 255**
 - **E=0** and **E=255** are **reserved for special use** (discussed later)
 - **E = 1** and **254** are used for **normalized** floating point numbers
 - **Bias = 127** (half of **254**), $\text{val}(E) = E - 127$
 - $\text{val}(E=1) = -126$, $\text{val}(E=127) = 0$, $\text{val}(E=254) = 127$

Biased Exponent – cont'd

- For **double precision**, exponent field is **11 bits**
- **E** can be in the range of **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**), $\text{val}(E) = E - 1023$
- $\text{val}(E=1) = -1022$, $\text{val}(E=1023)=0$, $\text{val}(E=2046)=1023$
- Value of a Normalized Floating Point Number is
 - $(-1)^S \times (1.F)_2 \times 2^{(E-Bias)}$
 - $(-1)^S \times (1.f_1f_2f_3f_4 \dots)_2 \times 2^{(E-Bias)}$
 - $(-1)^S \times (1 + f_1 \times 2^{-1} + f_2 \times 2^{-2} + f_3 \times 2^{-3} + f_4 \times 2^{-4} \dots)_2 \times 2^{(E-Bias)}$

Examples: Single Precision Float

1. 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 = $(1.0100...0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)
- Value in decimal = $-1.25 \times 2^{-3} = -0.15625$

2. What is the decimal value of?

01000001001001100000000000000000

Solution:

- Exponent = $(10000010)_2 = 130$
- Value in decimal = $+(1.01001100...0)_2 \times 2^{130-127}$
- $= (1.01001100)_2 \times 2^3 = (1010.01100...0)_2 = 10.375$

Examples: Double Precision Float

1. What is the decimal value of this Double Precision float?

01000000010100101010000000000000
00000000000000000000000000000000

Solution:

- Sign = 0 is positive
- Value of exponent = $(10000000101)_2 = 1029$, E-bias = $1029 - 1023 = 6$
- Value of double float = $(1.0010101...0)_2 \times 2^6 = (1001010.10...0)_2 = 74.5$

2. What is the decimal value of?

Solution:

- ECE 6913 - Floating Point Arithmetic— 21

Converting FP Decimal to Binary

3. Convert -0.8125 to binary in single and double precision

■ Solution:

- Fraction bits can be obtained using multiplication by 2

- $0.8125 \times 2 = 1.625$

- $0.625 \times 2 = 1.25$

- $0.25 \times 2 = 0.5$

- $0.5 \times 2 = 1.0$

- Stop when fractional part is 0

- Fraction = $(0.1101)_2 = (1.101)_2 \times 2^{-1}$ (Normalized)

- Exponent = $-1 + \text{Bias} = 126$ (single precision) and 1022 (double)

- Single:

10111111010100000000000000000000

- Double:

10111111111101010000000000000000
00000000000000000000000000000000

Largest Normalized Float

4. What is the Largest normalized float?

Solution (Single Precision):

01111111011111111111111111111111

- Exponent – bias = $254 - 127 = 127$ (largest exponent for SP)
- Significand = $1.111...1_2 = \text{almost } 2$
- Value in decimal $\sim 2 \times 2^{127} \sim 2^{128} = 3.4028 \dots \times 10^{38}$

Solution (Double Precision):

01111111111101111111111111111111

11111111111111111111111111111111

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

Smallest Normalized Float

5. What is the Smallest (in absolute value) normalized float?

Solution (Single Precision):

00000000100000000000000000000000

- Exponent – bias = $1 - 127 = -126$ (smallest exponent for SP)
- Significand = $1.000...0)_2 = 1$
- Value in decimal $\sim 1 \times 2^{-126} = 1.17549 \dots \times 10^{-38}$

Solution (Double Precision):

000000000000010000000000000000000000

000000000000000000000000000000000000

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

More Examples

6. Represent -0.75

$$-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$$

$$S = 1$$

$$\text{Fraction} = 1000\dots00_2$$

$$\text{Exponent} = -1 + \text{Bias}$$

$$\text{Single: } -1 + 127 = 126 = 01111110_2$$

$$\text{Double: } -1 + 1023 = 1022 = 011111111110_2$$

■ Single: $10111111101000\dots00$

■ Double: $101111111111101000\dots00$

Floating-Point Example 2

7. What number is represented by the single-precision float

`11000000101000...00`

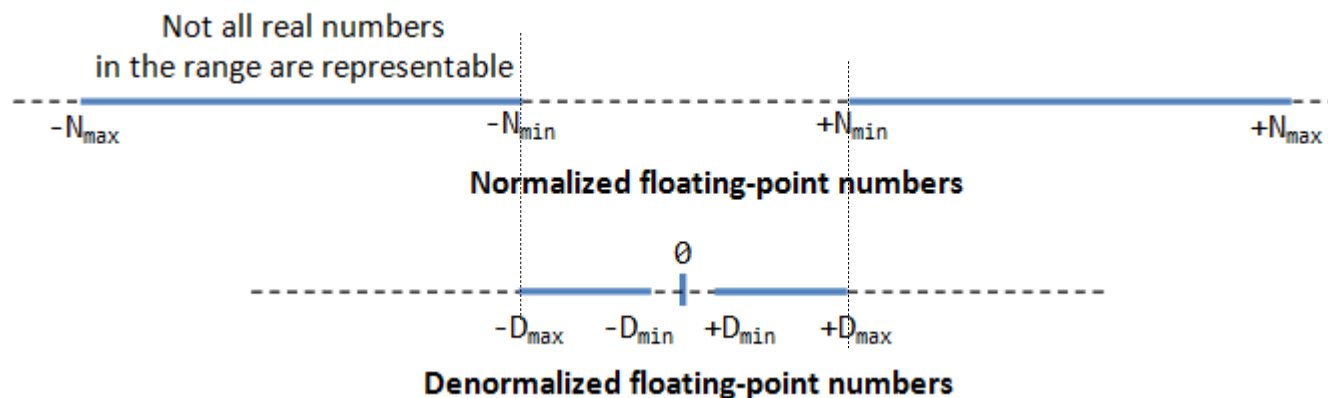
- $S = 1$
- Fraction = $01000...00_2$
- Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$
 $= (-1) \times 1.25 \times 2^2$
 $= -5.0$

Zero, Infinity and NaN

- Zero
 - Exponent field $E = 0$ and Fraction $F = 0$
 - $+0$ and -0 are possible according to the sign bit S
- Infinity
 - Infinity is a special value represented with maximum E and $F = 0$
 - For SP with 8 bit exponent: maximum $E = 255$
 - For DP with 11 bit exponent: maximum $E = 2047$
 - Infinity can result from overflow or division by 0
 - $+\text{inf}$ and $-\text{inf}$ are possible according to sign bit S
- NaN (Not a Number)
 - NaN is a special value represented with maximum E and $F \text{ not } = 0$
 - Result from exceptional situations, such as $0/0$ or $\text{sqrt}(\text{negative})$
 - Operation on a NaN results is NaN: $\text{Op}(X, \text{NaN}) = \text{NaN}$

Denormalized Numbers

- IEEE standard uses denormalized numbers to
 - Fill the gap between 0 and the smallest normalized float
 - Provide a gradual underflow to zero
- Denormalized: exponent field E is 0 and fraction F not = 0
 - Implicit 1. before - the fraction now becomes 0. (not normalized)
- Value of denormalized number (S, 0, F)
Single Precision: $(-1)^S \times (0.F)_2 \times 2^{-126}$
Double Precision: $(-1)^S \times (0.F)_2 \times 2^{-1022}$



Floating-Point Comparison

- IEEE 754 floating point numbers are ordered
 - Because exponent uses a biased representation...
 - Exponent value and its binary representation have the same ordering
 - Placing exponent before the fraction field **orders the magnitude**
 - *Larger exponent \Rightarrow larger magnitude*
 - *For equal exponents, larger fraction \Rightarrow larger magnitude*
 - $0 < (0.F)_2 \times 2^{E_{min}} < (1.F)_2 \times 2^{E-Bias} < inf \ (E_{min} = 1-Bias)$

Floating Point Addition Example

8. Consider Adding (Single-Precision Floating-Point)

$$\begin{array}{r} 1.1110010000000000000010_2 \times 2^4 \\ + 1.100001000000000110000101_2 \times 2^2 \end{array}$$

- Cannot add significands... why?
 - Because **exponents are not equal**
- How do we make exponents equal?
 - **Shift the significand of the lesser exponent right**
 - Difference between the 2 exponents = $4 - 2 = 2$
 - So, **shift right** second number by **2** bits and increment exponent

$$\begin{array}{r} + 1.100001000000000110000101_2 \times 2^2 \\ = 0.0110000100000000110000101_2 \times 2^4 \end{array}$$

Floating Point Addition – cont'd

- Now, ADD the Significands:

$$\begin{array}{r} 1.111001000000000000000010_2 \times 2^4 \\ + 1.1000000000000000110000101_2 \times 2^2 \\ \hline \end{array}$$

$$\begin{array}{r} 1.111001000000000000000010_2 \times 2^4 \\ + 0.011000000000000001100001_2 \times 2^4 \text{ (Shift right)} \\ \hline \end{array}$$

$$\begin{array}{r} + 10.010001000000000001100011_2 \times 2^4 \\ \hline \end{array}$$

- Addition produces a carry bit – result is NOT normalized
- Normalize* Result (shift right and increment exponent):

$$\begin{array}{r} 10.010001000000000001100011_2 \times 2^4 \\ = 1.001000100000000000110001_2 \times 2^5 \end{array}$$

Rounding

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

1.00100010000000000000110001 | **1** **01** $\times 2^5$
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

1.001000100000000000001100**10** $\times 2^5$

Floating-Point Addition

9. Consider a 4-digit decimal example

$$9.999 \times 10^1 + 1.610 \times 10^{-1}$$

- 1. *Align decimal points*

Shift number with smaller exponent

$$9.999 \times 10^1 + 0.016 \times 10^1$$

- 2. *Add significands*

$$9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$$

- 3. *Normalize result & check for over/underflow*

$$1.0015 \times 10^2$$

- 4. *Round and renormalize if necessary*

$$1.002 \times 10^2$$

Floating-Point Addition

10. Now consider a 4-digit binary example

$$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2} \quad (0.5 + -0.4375)$$

- 1. *Align binary points*

Shift number with smaller exponent

$$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$$

- 2. *Add significands*

$$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$$

- 3. *Normalize result & check for over/underflow*

$$1.000_2 \times 2^{-4}, \text{ with no over/underflow}$$

- 4. *Round and renormalize if necessary*

$$1.000_2 \times 2^{-4} \text{ (no change) } = 0.0625$$

Floating-Point Subtraction

11. Consider subtracting:

$$+ 1.00000000101100010001101 \times 2^{-6}$$

$$- 1.000000000000000010011010 \times 2^{-1}$$

$$+ 0.00001000000001011000100 \ 01101 \times 2^{-1} \quad (\text{shift right 5 bits})$$

$$- 1.000000000000000010011010 \times 2^{-1}$$

$$0.00001000000001011000100 \ 01101 \times 2^{-1}$$

$$0.111111111111111101100110 \times 2^{-1} \quad (2\text{s complement})$$

$$1.00001000000001000101010 \ 01101 \times 2^{-1} \quad (\text{ADD})$$

$$- 0.111101111111110111010101 \ 10011 \times 2^{-1} \quad (2\text{s complement})$$

Floating-Point Subtraction – cont'd

- So, we now have:

$$+ 1.00000000101100010001101 \times 2^{-6}$$

$$- 1.000000000000000010011010 \times 2^{-1}$$

$$-----$$

$$- 0.11110111111110111010101 \text{ } \textcircled{1}0011 \times 2^{-1}$$

Guard bit

- To normalize this result, shift left 1 bit, decrement exponent:

$$- 1.11101111111110111010101\textcolor{red}{1}0011 \times 2^{-2} \text{ (normalized)}$$

Guard bit: guards against loss of a fraction bit. Needed when subtracting since result may have a leading zero and should be normalized

Floating-Point Subtraction – cont'd

- Next, normalized result should be rounded

$$- 1.1110111111111011101010101\mathbf{1} \mathbf{0} \mathbf{011} \times 2^{-2}$$

Round bit: **R=0**

Sticky bit: **S=1**

- Since $R=0$, it is more accurate to truncate the result even if $S=1$. So, we simply discard the extra bits

$$- 1.1110111111111011101010101\mathbf{1} \mathbf{0}_R \mathbf{1}_S \times 2^{-2}$$

$$- 1.111011111111101110101011 \times 2^{-2}$$

IEEE 754 representation:

1011110111101111111111011101011

Rounding to Nearest Even

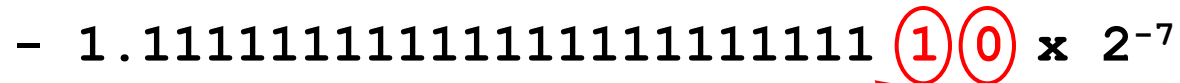
- Normalized result has the form:

1 . $f_1 f_2 f_3 \dots f_{23}$ RS

- The **Round bit R** appears after the last fraction bit **f_{23}**
- 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 last fraction bit
- **RS = 10** → Tie case – either truncate or increment:
 - Check last fraction bit (bit **f_{23}**)
 - if (bit **f_{23}**) is 0 the truncate result to keep fraction even
 - if (bit **f_{23}**) is 1 then increment result to make fraction even

Example on Rounding

12. Round the following using Rounding to Nearest Even

$$- 1.11111111111111111111111111111111 \text{ (1) (0) } \times 2^{-7}$$


Round bit: R=0

Sticky bit: S=1

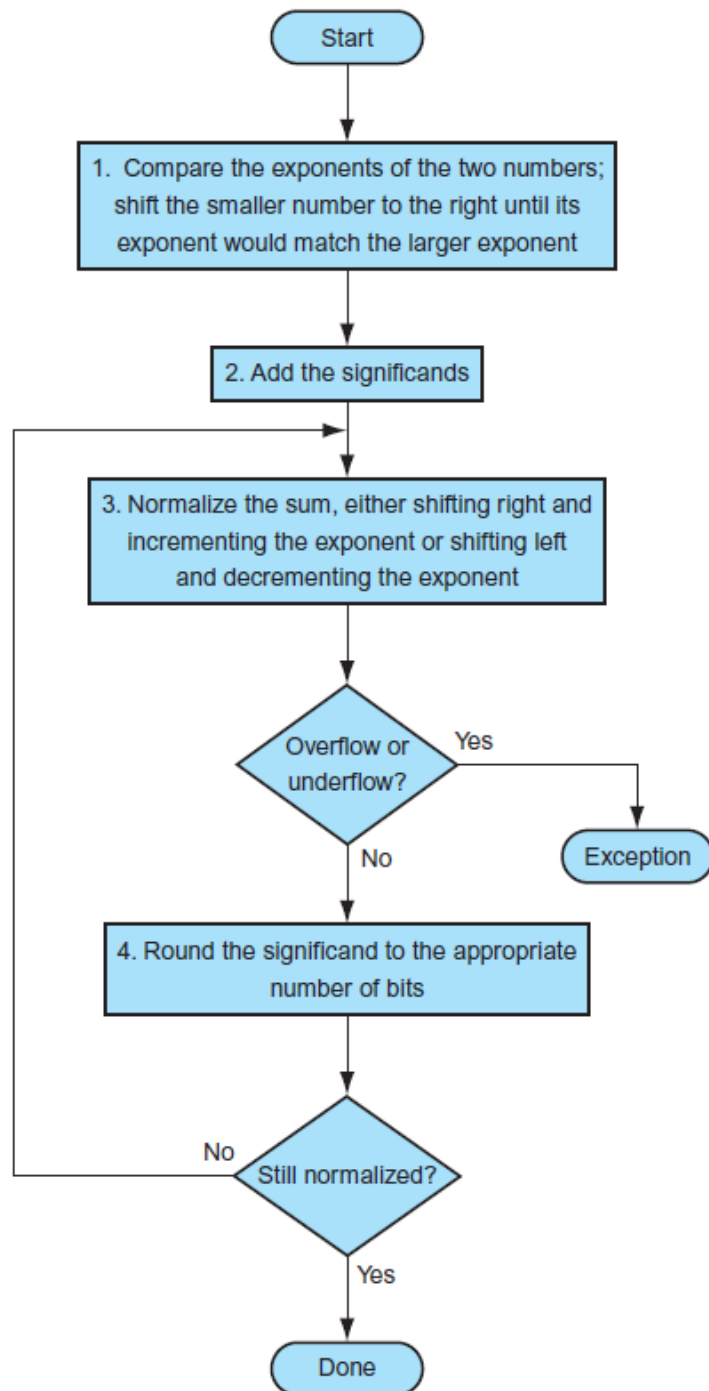
Increment result (add 1) since $RS = 10$ and $f_{23} = 1$

Incremented result: **-10.00000000000000000000000000000000** $\times 2^{-7}$

Renormalize and increment *exponent* (because of carry)

$$\boxed{-1.00000000000000000000000000000000 \times 2^{-6}}$$

Floating-Point Addition / Subtraction



1

Compare exponents & Shift significand by $d = |E_x - E_y|$

2

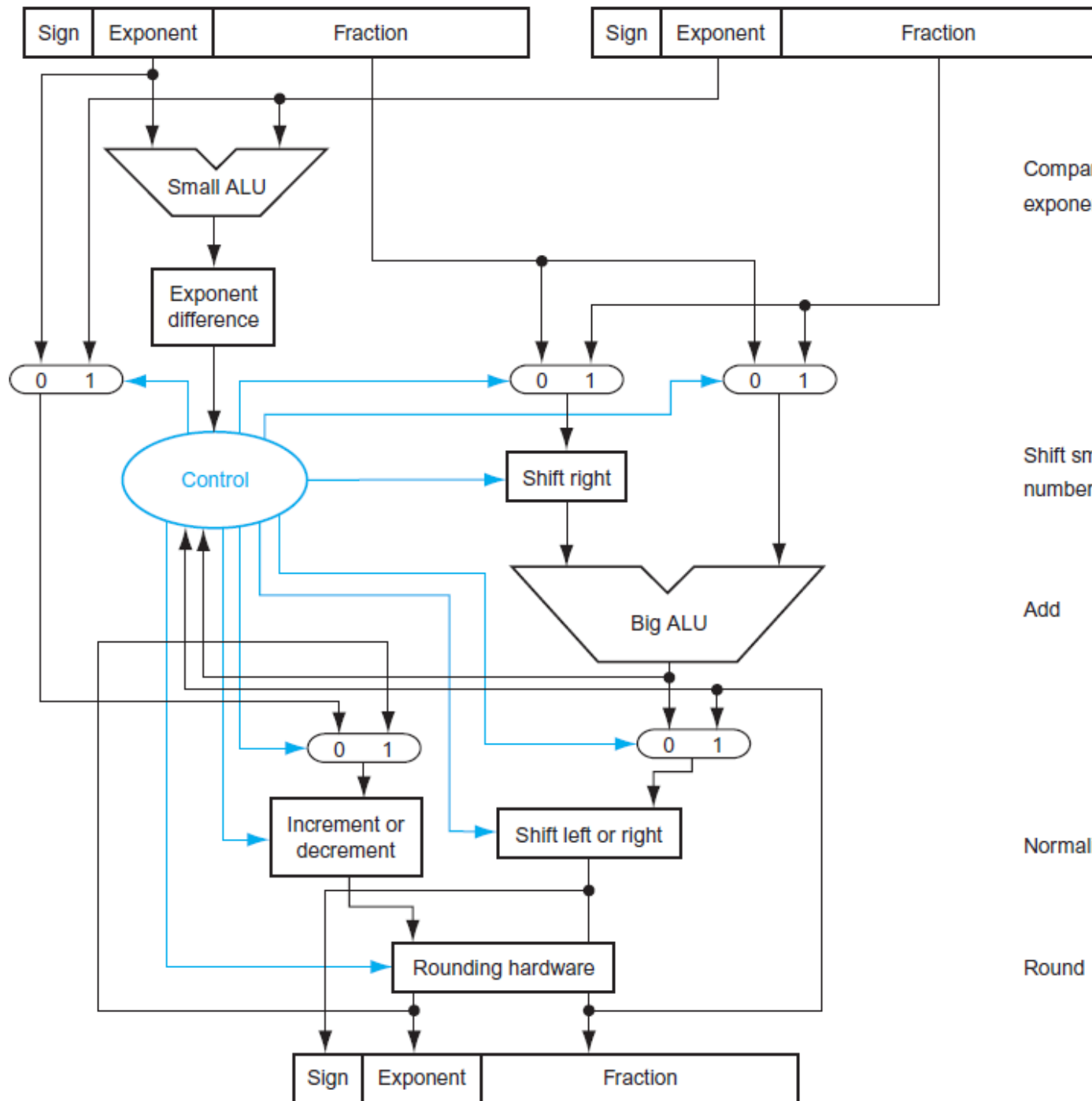
Add significands when signs are identical, subtract when different
 $X - Y$ becomes $X + (-Y)$

3

Normalization shifts right by 1 if there is a carry or shifts left by the number of leading zeros in the case of subtraction

4

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



Step 1

Step 2

Step 3

Step 4

Floating-Point Multiplication

- Consider multiplying:

$$\begin{array}{r} -1.110 \ 1000 \ 0100 \ 0000 \ 1010 \ 0001_2 \times 2^{-4} \\ 1.100 \ 0000 \ 0001 \ 0000 \ 0000 \ 0000_2 \times 2^{-2} \end{array}$$

- Unlike addition, we **add the exponents** of the operands
 - Result exponent value = $(-4) + (-2) = -6$
- Using the biased representation: **$E_z = E_x + E_y - \text{Bias}$**
 - $E_x = (-4) + 127 = 123$ (**Bias = 127 for SP**)
 - $E_y = (-2) + 127 = 125$
 - **$E_z = 123 + 125 - 127 = 121$ (value = -6)**
- Sign bit of product can be computed independently
- Sign bit of product = $\text{Sign}_x \text{ XOR } \text{Sign}_y = 1$ (**negative**)

Floating-Point Multiplication cont'd

- Now multiply the significands:

(Multiplicand)		1.11010000100000010100001
(Multiplier)	x	1.10000000001000000000000

		111010000100000010100001
		111010000100000010100001
		1.11010000100000010100001

		10.0100010010111101001101010010100001000000000000

- Multiplicand x 0 = 0 Zero rows are eliminated
- Multiplicand x 1 = Multiplicand (shifted left)

Floating-Point Multiplication, cont'd

- Normalize Product:

10.10111000111110111111001100101000010000000000000 $\times 2^{-6}$

- Shift right and increment exponent because of carry bit

1.010111000111110111111001100101000010000000000000 $\times 2^{-5}$

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

1.01011100011111011111100 1 100... $\times 2^{-5}$

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

- Final result

1.01011100011111011111101 $\times 2^{-5}$

IEEE 754 Representation

1 01111010 01011100011111011111101

Example 1

Consider a 4-digit decimal example

$$1.110 \times 10^{10} \times 9.200 \times 10^{-5}$$

- 1. Add exponents
 - For biased exponents, subtract bias from sum
 - New exponent = $10 + -5 = 5$
- 2. Multiply significands
 - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$
- 3. Normalize result & check for over/underflow
 - 1.0212×10^6
- 4. Round and renormalize if necessary
 - 1.021×10^6
- 5. Determine sign of result from signs of operands
 - $+1.021 \times 10^6$

Example 2

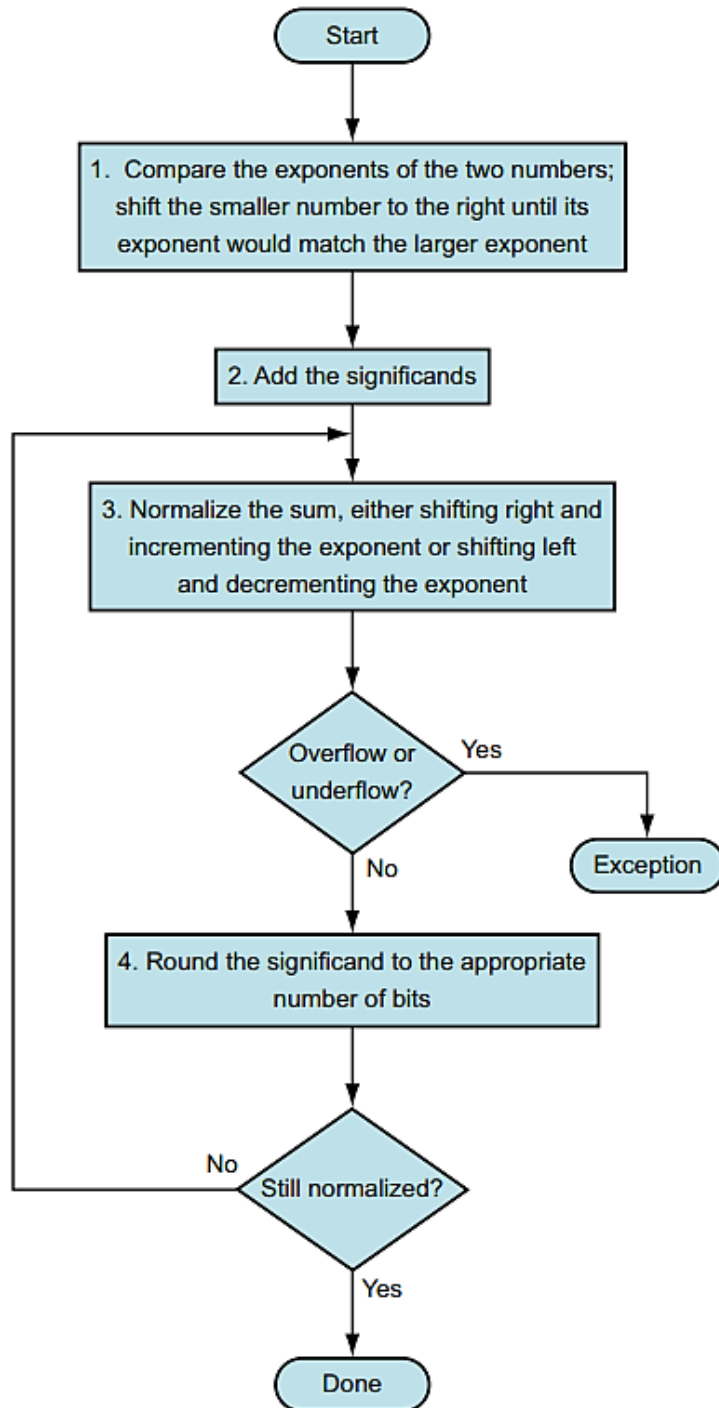
Now consider a 4-digit **binary** example

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

- 1. Add exponents
 - Unbiased: $-1 + -2 = -3$
 - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
 - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
 - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
 - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: $+ve \times -ve \Rightarrow -ve$
 - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
 - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
 - Addition, subtraction, multiplication, division, reciprocal, square-root
 - $\text{FP} \leftrightarrow \text{integer}$ conversion
- Operations usually takes several cycles
 - Can be pipelined



Biased Exponent Addition

$$E_z = E_x + E_y - \text{Bias}$$

Result sign

$$S_z = S_x \text{ XOR } S_y$$

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 by at most 1 bit and increment exponent

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

Integer Multiply Instructions

Multiply

General Form:

MUL RegD, Reg1, Reg2

Example:

MUL x4, x9, x13 # x4 = x9*x13

Description:

The contents of Reg1 is multiplied by the contents of Reg2 and the result is placed in RegD.

- Regardless of the size of the registers, the result of their multiplication will be twice as large, and therefore require 2 registers to contain.
- This instruction *captures the lower-order half of the result* and moves it into the destination register

Integer Multiply Instructions

Multiply – High Bits (Signed)

General Form:

MULH RegD, Reg1, Reg2

Example:

MULH x4, x9, x13 # x4 = HighBits(x9*x13)

Description:

The contents of Reg1 is multiplied by the contents of Reg2 and the most-significant half of the result is placed in RegD. Both operands and the result are interpreted as signed values.

Encoding:

This is a R-type instruction.

- This instruction *captures the upper half of the result* and moves it into the destination register

Integer Multiply Instructions

Recommended Usage: Typically, the programmer will want to obtain the full result of a multiplication, i.e., both upper half and lower halves. This requires two multiply instructions.

For example, the following sequence

```
MULH    x4 , x9 , x13    # compute upper half
MUL      x5 , x9 , x13    # compute lower half
```

will place the result in the register pair x4:x5.

Integer Divide Instructions

Consider dividing a by n (that is, a/n).

$q = a \text{ DIV } n$ # compute quotient
 $r = a \text{ REM } n$ # compute remainder

$$a = nq + r$$

$$|r| < |n|$$

Many languages (C, C++, Java) perform “truncated division”:

$$q = \text{trunc}(a/n)$$

$$r = a - n \text{ trunc}(a/n)$$

which produces these results:

$7 / 3 = 2$	$7 \% 3 = 1$
$-7 / 3 = -2$	$-7 \% 3 = -1$
$7 / -3 = -2$	$7 \% -3 = 1$
$-7 / -3 = 2$	$-7 \% -3 = -1$

Integer Divide Instructions

Divide (Signed)

General Form:

DIV RegD, Reg1, Reg2

Example:

DIV x4, x9, x13 # x4 = x9 DIV x13

Description:

The contents of Reg1 is divided by the contents of Reg2 and the quotient is placed in RegD. Both operands and the result are signed values.

Remainder (Signed)

General Form:

REM RegD, Reg1, Reg2

Example:

REM x4, x9, x13 # x4 = x9 REM x13

Description:

The contents of Reg1 is divided by the contents of Reg2 and the remainder is placed in RegD. Both operands and the result are signed values.

Integer Divide Instructions

- Often, both the quotient and remainder is of interest.
- It is recommended that the DIV be done first and the REM be done second

```
DIV x4, x9, x13 # x4 = x9 DIV x13
```

```
REM x5, x9, x13 # x5 = x9 REM x13
```

- In some implementations, the execution unit may recognize this common pattern and fuse these two instructions into a single division operation, thereby improving performance

FP Instructions in RISC-V

- Separate FP registers: f0, ..., f31
 - double-precision
 - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
 - Programs generally don't do integer ops on FP data, or vice versa
 - More registers with minimal code-size impact
- FP load and store instructions
 - flw, fld
 - fsw, fsd

Floating Point Extensions

- The RISC-V spec describes these extensions to support floating point arithmetic
 - F – Single precision floating point (32 bit values)
 - D – Double precision floating point (64 bit values)
 - Q – Quad precision floating point (128 bit values)
- The “D” extension is a superset of “F”; when double precision is implemented, all instructions operating on single precision values will also be included.
- Likewise, the “Q” is a superset of “D”; when the “Q” extension is implemented, all “F” and “D” instructions will also be implemented.

Floating Point Registers

- There are 32 floating point registers, named f0, f1, ... f31
- In the “F” extension, each register can hold one single precision floating point value - each register is 32 bits wide.
- All registers function identically - there is nothing special about f0, as there is with x0 of the integer registers
- In addition: *Floating Point Control and Status Register (FCSR)*
- Whose bits can be queried after a sequence of instructions to determine if any of several unusual conditions has occurred

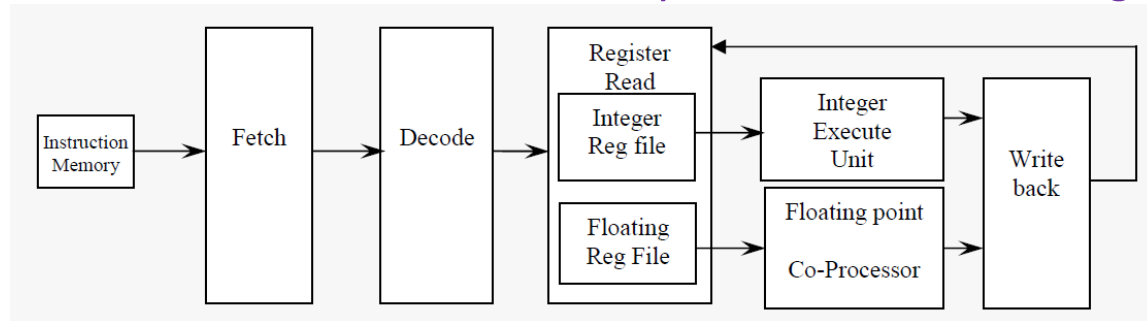
<u>Bits</u>	<u>Width in bits</u>	<u>Description</u>
0	1	NX: Inexact
1	1	UF: Underflow
2	1	OF: Overflow
3	1	DZ: Divide By Zero
4	1	NV: Invalid Operation
5:7	3	Floating Point Rounding Mode (FRM)
8:31	24	(unused)

FCSR

- **NX:** If the result of an operation had to be rounded, then the “NX: Inexact” bit will be set.
- **UF:** If the result is too small to fit in a normalized form and is also inexact
- **OF:** If the result is too large to be represented
- **DZ:** bit is set for operations like $1/0$ and $\log(0)$ and $+\infty$ or $-\infty$ will be returned as the result.
- **NV:** Invalid operations, such as “*square root of a negative number*” cause the NV bit to be set and NaN to be returned
- Floating Point instructions that have problems will set the FCSR bits but will *never cause a trap or exception*. Instead, instruction execution will continue uninterrupted.

FP Hardware in RISC-V

- The coprocessor for floating point *integrates with integer pipeline*
- FPU acts a kind of accelerator & *works in parallel with the integer pipeline*



- The fetch unit fetches the instructions from the program memory based on the program counter value.
- Decoder decodes the instructions and *passes Register addresses to the register select unit*. If the current instruction is integer related instruction, then it passes to the integer pipeline and if the instruction is related the floating point, then passes to the floating point co-processor pipeline.
- For floating point instructions, integer decoder does only a partial decoding, full *decoding of FP instructions takes place inside the FPU coprocessor*

FP Hardware in RISC-V

- Floating point coprocessor (FPU) performs operations like *addition, subtraction, division, square root, multiplication, fused multiply and accumulate and compare*.
- Floating point operations are *part of ARM, MIPS, and RISC-V etc. instruction sets*
- RISC-V Floating point units designed for RISC-V floating point instructions is *fully compatible with IEEE 754-2008 standard*
- Capable of handling both *single and double precision floating point data operands*
- The front end of the floating point processor accepts *three data operands, rounding mode and associated Opcode fields* for decoding
- The FPU decodes instructions and executes them. The Final result is written back to the registers through write back unit

Floating Point L/S Instructions

Floating Load (Word)

General Form:

FLW FRegD, Immed-12 (Reg1)

Example:

FLW f4, 1234(x9) # f4 = Mem[x9+1234]

Description:

A 32-bit value is fetched from memory and moved into floating register FRegD.
The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

Floating Point L/S Instructions

Floating Load (Double)

General Form:

FLD FRegD, Immed-12 (Reg1)

Example:

FLD f4, 1234 (x9) # f4 = Mem[x9+1234]

Description:

A 64-bit value is fetched from memory and moved into floating register FRegD.
The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

Floating Point L/S Instructions

Floating Load (Double)

General Form:

FLD FRegD, Immed-12 (Reg1)

Example:

FLD f4, 1234 (x9) # f4 = Mem[x9+1234]

Description:

A 64-bit value is fetched from memory and moved into floating register FRegD.
The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

Floating Point L/S Instructions

Floating Store (Word)

General Form:

FSW FReg2, Immed-12 (Reg1)

Example:

FSW f4, 1234 (x9), f4 # Mem[x9+1234] = f4

Description:

A 32-bit value is copied from register FReg2 to memory. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

Floating Point Arithmetic Instructions

Floating Add

General Form:

FADD.S FRegD, FReg1, FReg2 (single precision)

FADD.D FRegD, FReg1, FReg2 (double precision)

FADD.Q FRegD, FReg1, FReg2 (quad precision)

Example:

FADD.S f4, f9, f13 # f4 = f9+f13 (32 bits)

FADD.D f4, f9, f13 # f4 = f9+f13 (64 bits)

FADD.Q f4, f9, f13 # f4 = f9+f13 (128 bits)

Description:

The value in FReg1 is added to the value in FReg2 and the result is placed in FRegD.

Floating Point Arithmetic Instructions

Floating Subtract

General Form:

FSUB.S	FRegD, FReg1, FReg2	(single precision)
FSUB.D	FRegD, FReg1, FReg2	(double precision)
FSUB.Q	FRegD, FReg1, FReg2	(quad precision)

Example:

FSUB.S	f4, f9, f13	# f4 = f9 - f13	(32 bits)
FSUB.D	f4, f9, f13	# f4 = f9 - f13	(64 bits)
FSUB.Q	f4, f9, f13	# f4 = f9 - f13	(128 bits)

Description:

The value in FReg1 is subtracted from the value in FReg2 and the result is placed in FRegD.

Floating Point Arithmetic Instructions

Floating Multiply

General Form:

FMUL.S FRegD, FReg1, FReg2 (single precision)

FMUL.D FRegD, FReg1, FReg2 (double precision)

FMUL.Q FRegD, FReg1, FReg2 (quad precision)

Example:

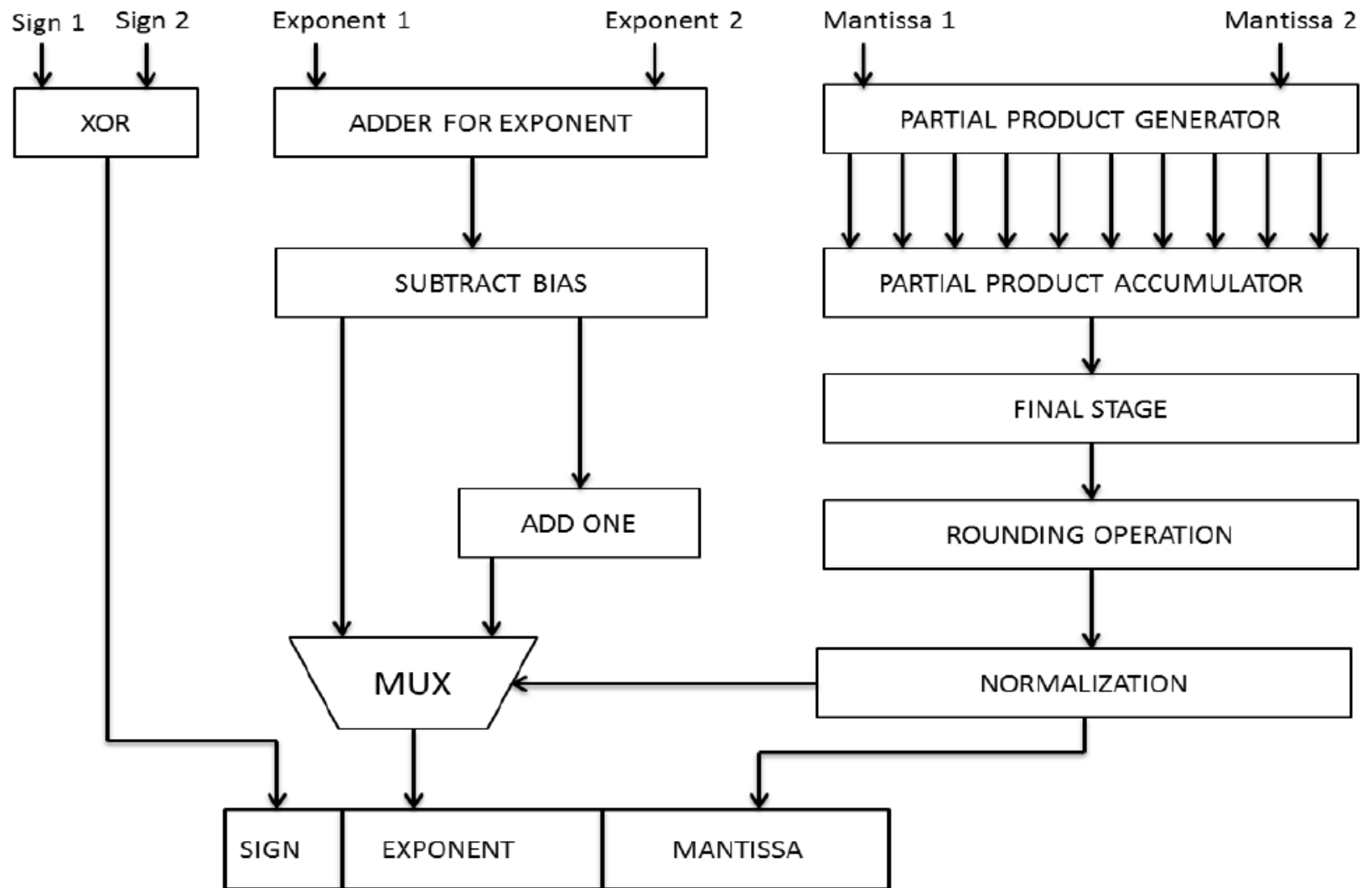
FMUL.S f4, f9, f13 # f4 = f9*f13 (32 bits)

FMUL.D f4, f9, f13 # f4 = f9*f13 (64 bits)

FMUL.Q f4, f9, f13 # f4 = f9*f13 (128 bits)

Description:

The value in FReg1 is multiplied by the value in FReg2 and the result is placed in FRegD.



FP Example: °F to °C

- C code:

```
float f2c (float fahr) {  
    return ((5.0/9.0)*(fahr - 32.0));  
}
```

- fahr in f10, result in f10, literals in global memory space

- Compiled RISC-V code:

f2c:

```
f1w    f0,const5(x3)    // f0 = 5.0f  
f1w    f1,const9(x3)    // f1 = 9.0f  
fdiv.s f0, f0, f1       // f0 = 5.0f / 9.0f  
f1w    f1,const32(x3)   // f1 = 32.0f  
fsub.s f10,f10,f1       // f10 = fahr - 32.0  
fmul.s f10,f0,f10       // f10 = (5.0f/9.0f) * (fahr-32.0f)  
jalr   x0,0(x1)        // return
```

Back Up

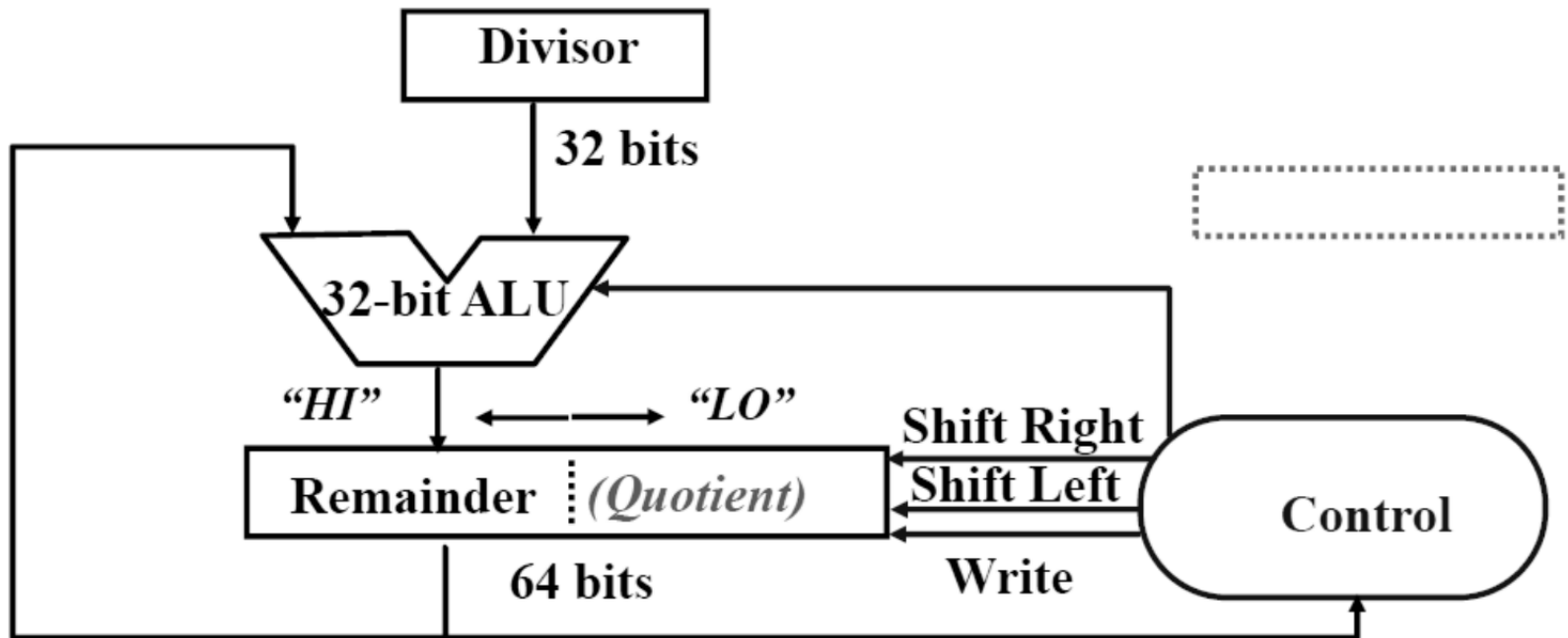


Divide - comments

- Eliminate Quotient register by combining with Remainder as shifted left
 - Start by shifting the Remainder left as before.
 - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
 - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will be shifted left one time too many.
 - Thus the final correction step must shift back only the remainder in the left half of the register

Divide hardware – improved Version

- 32-bit Divisor reg, 32-bit ALU, **64-bit Remainder reg**, (Quotient reg eliminated)

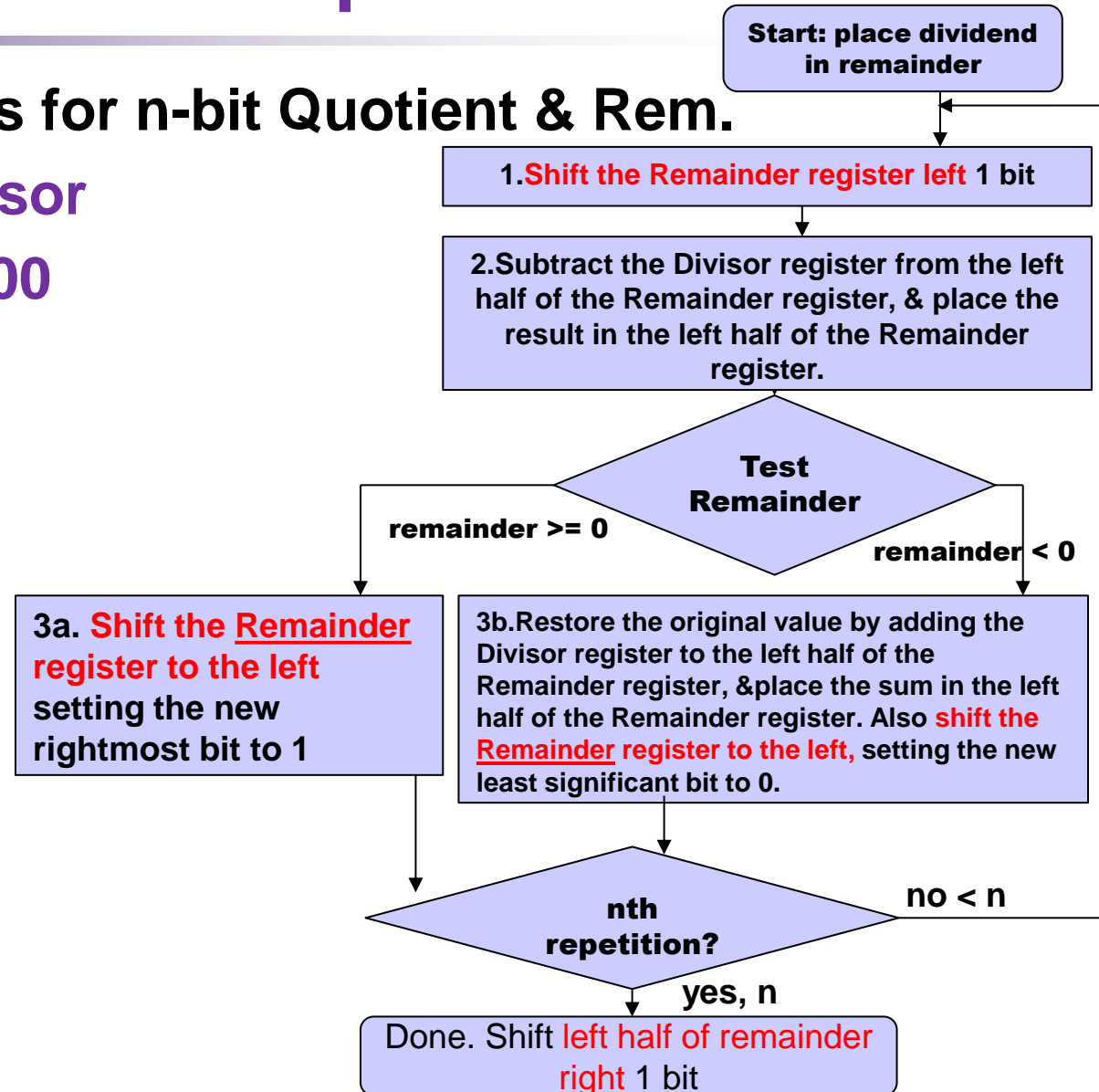


Divide algorithm – improved Version

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

Remainder Divisor
0100 1010 1000

$n=4$



Divide version - comments

- Same Hardware as Multiply: just need ALU to add or subtract, and 64-bit register to shift left or shift right
- Hi and Lo registers in MIPS and RISC-V combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
 - Note: Dividend and Remainder must have same sign
 - Note: Quotient negated if Divisor sign & Dividend sign disagree. e.g., $-7 \div 2 = -3$, remainder = -1

FP Example: Array Multiplication

- $C = C + A \times B$
 - All 32×32 matrices, 64-bit double-precision elements

- C code:

```
void mm (double c[][],  
         double a[][], double b[][]) {  
    size_t 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)  
                c[i][j] = c[i][j]  
                    + a[i][k] * b[k][j];  
}
```

- Addresses of c, a, b in x10, x11, x12, and
i, j, k in x5, x6, x7

FP Example: Array Multiplication

■ RISC-V code:

```
mm: ...  
  
    li    x28,32      // x28 = 32 (row size/loop end)  
    li    x5,0        // i = 0; initialize 1st for loop  
L1:  li    x6,0        // j = 0; initialize 2nd for loop  
L2:  li    x7,0        // k = 0; initialize 3rd for loop  
    slli  x30,x5,5     // x30 = i * 2**5 (size of row of c)  
    add   x30,x30,x6    // x30 = i * size(row) + j  
    slli  x30,x30,3     // x30 = byte offset of [i][j]  
    add   x30,x10,x30   // x30 = byte address of c[i][j]  
    fld   f0,0(x30)    // f0 = c[i][j]  
L3:  slli  x29,x7,5     // x29 = k * 2**5 (size of row of b)  
    add   x29,x29,x6    // x29 = k * size(row) + j  
    slli  x29,x29,3     // x29 = byte offset of [k][j]  
    add   x29,x12,x29   // x29 = byte address of b[k][j]  
    fld   f1,0(x29)    // f1 = b[k][j]
```

FP Example: Array Multiplication

```
slli    x29,x5,5      // x29 = i * 2**5 (size of row of a)
add     x29,x29,x7     // x29 = i * size(row) + k
slli    x29,x29,3      // x29 = byte offset of [i][k]
add     x29,x11,x29    // x29 = byte address of a[i][k]
fld     f2,0(x29)      // f2 = a[i][k]
fmul.d  f1, f2, f1     // f1 = a[i][k] * b[k][j]
fadd.d  f0, f0, f1     // f0 = c[i][j] + a[i][k] * b[k][j]
addi    x7,x7,1        // k = k + 1
bltu    x7,x28,L3      // if (k < 32) go to L3
fsd     f0,0(x30)      // c[i][j] = f0
addi    x6,x6,1        // j = j + 1
bltu    x6,x28,L2      // if (j < 32) go to L2
addi    x5,x5,1        // i = i + 1
bltu    x5,x28,L1      // if (i < 32) go to L1
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
 - Extra bits of precision (guard, round, sticky)
 - Choice of rounding modes
 - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
 - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Subword Parallelism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
 - Example: 128-bit adder:
 - Sixteen 8-bit adds
 - Eight 16-bit adds
 - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

x86 FP Architecture

- Originally based on 8087 FP coprocessor
 - 8 × 80-bit extended-precision registers
 - Used as a push-down stack
 - Registers indexed from TOS: ST(0), ST(1), ...
- FP values are 32-bit or 64 in memory
 - Converted on load/store of memory operand
 - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
 - Result: poor FP performance

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- Optional variations
 - I: integer operand
 - P: pop operand from stack
 - R: reverse operand order
 - But not all combinations allowed

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
 - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
 - 2 × 64-bit double precision
 - 4 × 32-bit double precision
 - Instructions operate on them simultaneously
 - Single-Instruction Multiple-Data

Matrix Multiply

■ Unoptimized code:

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.     for (int i = 0; i < n; ++i)
4.         for (int j = 0; j < n; ++j)
5.             {
6.                 double cij = C[i+j*n]; /* cij = C[i][j] */
7.                 for(int k = 0; k < n; k++ )
8.                     cij += A[i+k*n] * B[k+j*n]; /* cij +=
A[i][k]*B[k][j] */
9.                 C[i+j*n] = cij; /* C[i][j] = cij */
10.            }
11. }
```

Matrix Multiply

■ x86 assembly code:

```
1. vmovsd (%r10),%xmm0    # Load 1 element of C into
   %xmm0
2. mov %rsi,%rcx           # register %rcx = %rsi
3. xor %eax,%eax           # register %eax = 0
4. vmovsd (%rcx),%xmm1     # Load 1 element of B into
   %xmm1
5. add %r9,%rcx            # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
   element of A
7. add $0x1,%rax           # register %rax = %rax + 1
8. cmp %eax,%edi           # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>      # jump if %eax > %edi
11. add $0x1,%r11d         # register %r11 = %r11 + 1
12. vmovsd %xmm0, (%r10)   # Store %xmm0 into C element
```

Matrix Multiply

■ Optimized C code:

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.     for ( int i = 0; i < n; i+=4 )
5.         for ( int j = 0; j < n; j++ ) {
6.             __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 =
C[i][j] */
7.             for( int k = 0; k < n; k++ )
8.                 c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j]
*/
9.                                     _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                                     _mm256_broadcast_sd(B+k+j*n)));
11.             _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.         }
13. }
```

Matrix Multiply

■ Optimized x86 assembly code:

```
1. vmovapd (%r11),%ymm0      # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx             # register %rcx = %rbx
3. xor %eax,%eax             # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax             # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1,4 A elements
7. add %r9,%rcx              # register %rcx = %rcx + %r9
8. cmp %r10,%rax             # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0  # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>      # jump if not %r10 != %rax
11. add $0x1,%esi            # register % esi = % esi + 1
12. vmovapd %ymm0, (%r11)    # Store %ymm0 into 4 C elements
```

Right Shift and Division

- Left shift by i places multiplies an integer by 2^i
- Right shift divides by 2^i ?
 - Only for unsigned integers
- For signed integers
 - Arithmetic right shift: replicate the sign bit
 - e.g., $-5 / 4$
 - $11111011_2 \gg 2 = 11111110_2 = -2$
 - Rounds toward $-\infty$
 - c.f. $11111011_2 \ggg 2 = 00111110_2 = +62$

Associativity

- Parallel programs may interleave operations in unexpected orders
 - Assumptions of associativity may fail

		$(x+y)+z$	$x+(y+z)$
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism