

Computer Architecture

# Arithmetic for Computers




## Part 2

**Floating-Point Real Numbers**

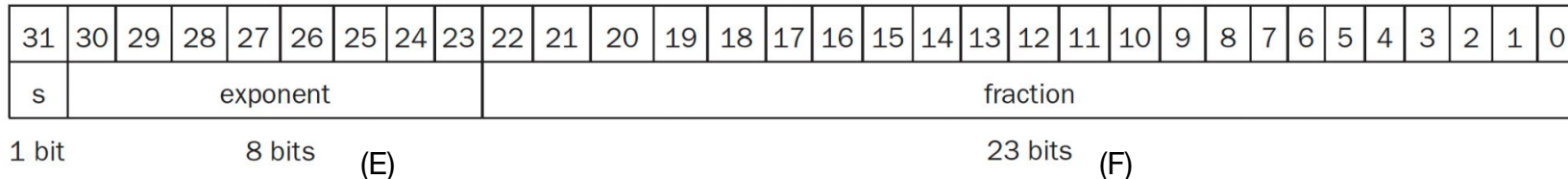
# Agenda

- Representation of real numbers
  - The format of the floating-point numbers
  - Excess-M representation
  - Special cases, overflow, and underflow
  - The precisions
- Arithmetic of floating-point numbers
- Accuracy

# Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - $-2.34 \times 10^{56}$   Normalized; i.e., no leading 0s
  - $+0.002 \times 10^{-4}$  
  - $+987.02 \times 10^9$   Not normalized
- In binary, the normalized notation is
  - $\pm 1.xxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

# Floating-Point Format



- It has three components,
  - The sign (1 bit)
  - The exponent (8 bits)
  - The fraction (23 bits), also called matissa
- A number =  $(-1)^s \times F \times 2^E$
- The format is a **compromise** between *the size of fraction* and *the size of exponent*.
  - Also, many different ways to represent the exponent and the fraction.

# Floating Point Standard

- Defined by IEEE Std 754-1985
- 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)
- Normalized significand:  $1.0 \leq |\text{significand}| < 2.0$ 
  - 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: using excess representation
  - Exponent is represented by unsigned binary numbers = actual exponent + Bias
  - Single: Bias = 127; Double: Bias = 1023

# Excess representation

It is also known as **biased representation**. It is a way for unsigned binary values to represent signed integers.

- In excess representation, an unsigned binary of integer  **$M$**  (called the bias) represents the value **0**, whereas **all zeros** in the bit pattern represents the integer  **$-M$**
- For a  $n$ -bit binary number system, a usual  $M$  is set to be  $2^{n-1}$ .

# Excess representation

Let  $Bias = 2^{(n-1)}$ ,  $n$  is the length of the binary.

So, if  $n=3$ ,  $2^2 \Rightarrow$  unsigned binary 100 is set to be 0. All 0s is set to be -4.

- Binary to decimal:
  - Treat the binary as unsigned binary
  - Convert it into decimal
  - Subtract the bias
  - e.g.,  $n=3$ , we have  $111 \Rightarrow 7 \Rightarrow 7-4=3$
- Decimal to binary:
  - Add the bias to the decimal
  - Convert it to binary as unsigned binary.

Using pattern of length three

Bit pattern	Value represented
111	3
110	2
101	1
100	0
011	-1
010	-2
001	-3
000	-4

Using pattern of length four

Bit pattern	Value represented
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	-1
0110	-2
0101	-3
0100	-4
0011	-5
0010	-6
0001	-7
0000	-8



# Excess representation in IEEE FP format

single: 8 bits  
double: 11 bits

single: 23 bits  
double: 52 bits

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

- Single precision:  $\text{Bias} = 127 = 2^7 - 1$
- Double precision:  $\text{Bias} = 1023 = 2^{10} - 1$
- Exponents 00000000 and 11111111 reversed for special cases

# Special cases


- Exponent = 000...0  $\Rightarrow$  hidden bit is 0, not 1

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Exponent = 000...00, Fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations  
of 0.0!



# Special cases

- Exponent = 111...11, Fraction = 000...00
  - $\pm$ infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...11, Fraction  $\neq$  000...00
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0/0.0
  - Can be used in subsequent calculations

# Floating-Point Example

- Represent -0.75
  - $-0.75 = (-1)^1 \times 0.11_2 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - $S = 1$
  - Fraction =  $1000\dots00_2$
  - Exponent =  $-1 + \text{Bias}$ 
    - Single:  $-1 + 127 = 126 = 01111110_2$
    - Double:  $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000...00
- Double: 1011111111101000...00

# Floating-Point Example

- What number is represented by the single-precision float

11000000101000...00

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

# Overflow and Underflow

In the floating point format,

- **Overflow:** the exponent is too large to be represented in the exponent field;
- **Underflow:** when the nonzero fraction becomes too small that it cannot be represented;
- Single precision: exponent should be in  $(-126, 127)$
- Double precision: exponent should be in  $(-1022, 1023)$

# Single-Precision Range

Note: Exponents 00000000 and 11111111 reversed for special cases

- Smallest value

- Exponent: 00000001  $\Rightarrow$  actual exponent =  $1 - 127 = -126$
- Fraction: 000...00  $\Rightarrow$  significand = 1.0
- $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

- Largest value

- Exponent: 11111110  $\Rightarrow$  actual exponent =  $254 - 127 = +127$
- Fraction: 111...11  $\Rightarrow$  significand  $\approx 2.0$
- $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

# Double-Precision Range

Note: Exponents 0000...00 and 1111...11 reversed for special cases

- 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: 11111110  $\Rightarrow$  actual exponent = 2046 - 1023 = +1223
- Fraction: 111...11  $\Rightarrow$  significand  $\approx$  2.0
- $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$



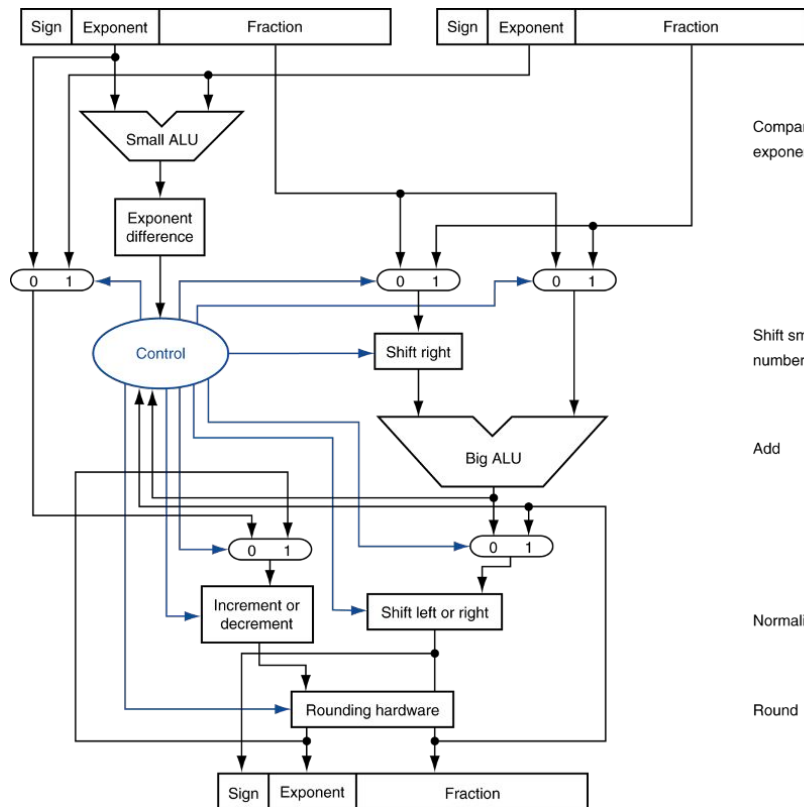
# Floating-Point Addition

- Consider a 4-digit decimal example:  $9.999 \times 10^1 + 1.610 \times 10^{-1}$ 
  - Align decimal points
    - Shift number with smaller exponent
    - $9.999 \times 10^1 + 0.016 \times 10^1$
  - Add significands
    - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
  - Normalize result & check for over/underflow
    - $1.0015 \times 10^2$
  - Round and renormalize if necessary (4 digits in this case)
    - $1.002 \times 10^2$

# Floating-Point Addition

- Now consider a 4-digit binary example:  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ , i.e., (0.5+-0.4375)
  - Align binary points
    - Shift number with smaller exponent
    - $1.000_2 \times 2^{-1} + -0.1110_2 \times 2^{-1}$
  - Add significands
    - $1.000_2 \times 2^{-1} + -0.1110_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
  - Normalize result & check for over/underflow
    - $1.000_2 \times 2^{-4}$ , with no over/underflow
  - Round and renormalized if necessary
    - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

# Floating-Point Addition



Compare exponents

Shift smaller number right

Add

Normalize

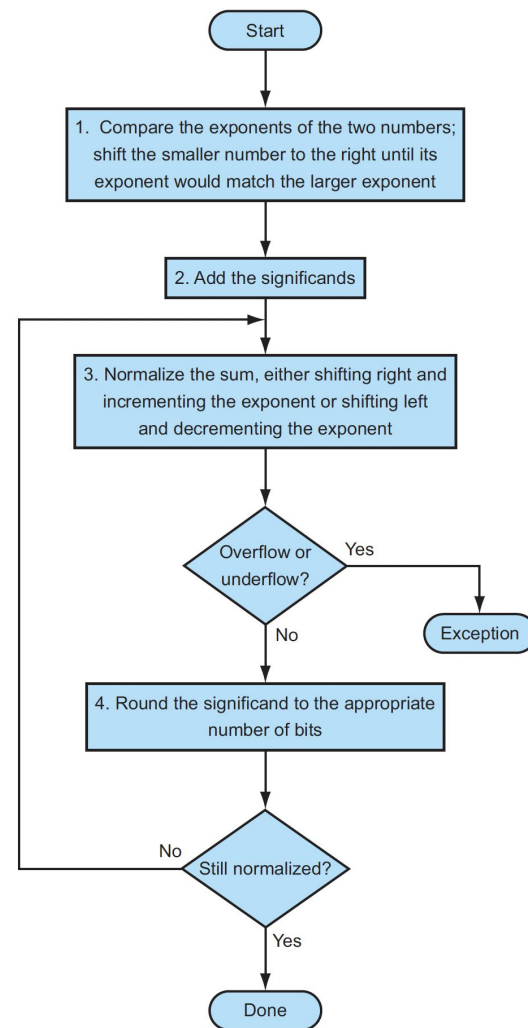
Round

Step 1

Step 2

Step 3

Step 4



# Floating-Point Multiplication

- Consider a 4-digit decimal example:  $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$ 
  - Add exponents
    - For biased exponents, subtract bias from sum
    - New exponent =  $10 + -5 = 5$
  - Multiply significand
    - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$
  - Normalize result & check for over/underflow
    - $1.0212 \times 10^6$
  - Round and renormalize if necessary
    - $1.021 \times 10^6$
  - Determine sign of result from signs of operands
    - $+1.021 \times 10^6$

# Floating-Point Multiplication

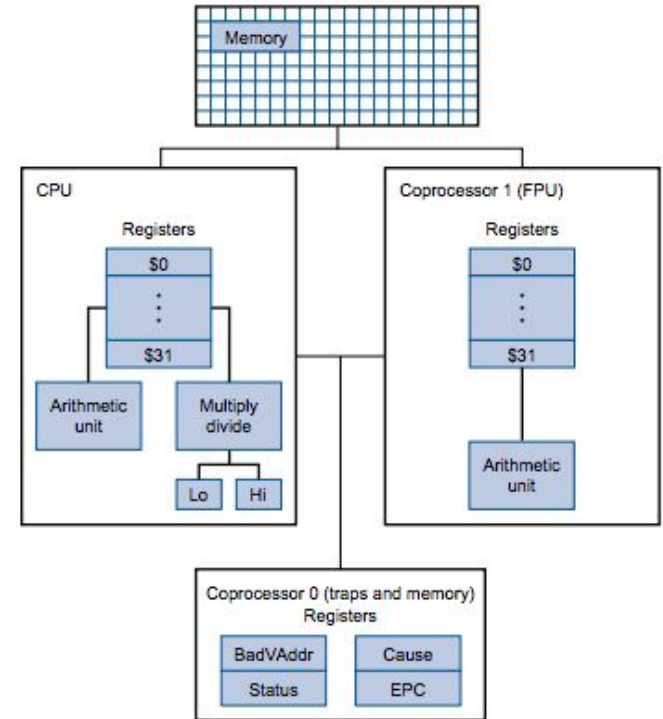
- Now consider a 4-digit binary example:  $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ , (i.e.,  $0.5 \times -0.4375$ )
  - Add exponents
    - unbiased:  $-1 + -2 = -3$
    - biased:  $(-1+127)+(-2+127) = -3+254-127 = -3 +127$
  - Multiply significands
    - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2} \Rightarrow 1.110_2 \times 2^{-3}$
  - Normalize result & check for over/underflow
    - $1.110_2 \times 2^{-3}$  (no change) with no over/underflow
  - Round and renormalize if necessary
    - $1.110_2 \times 2^{-3}$  (no change)
  - Determine the sign: **+ve** x **-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 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

# FP Instructions in MIPS

- MIPS has a specialized component which is designed to handle floating-point arithmetic operations, named as, MIPS Floating Point Unit (FPU), or Coprocessor 1.
- FPU is an adjunct processor that extends the ISA



# FP Instructions in MIPS

- FPU has separate register file:
  - 32 single-precision: `$f0, $f1, ..., $31`
  - Paired for double-precision: `$f0/$f1, $f2/$f3, ...`
    - Release 2 of MIPS ISA supports 32 x 64-bit registers
- FP instructions operate only on FP registers
  - Programs generally don't do integer operations on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1, ldc1, swc1, sdc1` #load/save single/double precision data to a register in Coprocessor 1
    - e.g., `ldc1 $f8, 32($sp)` #load register f8 from the memory address with offset 32 from `$sp`



# FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s, sub.s, mul.s, div.s`
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d, sub.d, mul.d, div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single-and double-precision comparison
  - `c.xx.s, c.xx.d` (xx is `eq, lt, le, ...` )
- Branch on FP condition code true and false
  - `bc1t, bc1f` #branch on Coprocessor 1 if true/false
    - e.g., `bc1t TargetLabel`

# FP Example: Fahrenheit to Celsius

- C code: fahr in \$f12, result in \$f0, literals in global memory space

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

- Compiled MIPS code:

```
f2c: lwc1    $f16, const5($gp)  
     lwc2    $f18, const9($gp)  
     div.s   $f16, $f16, $f18  
     lwc1    $f18, const32($gp)  
     sub.s   $f18, $f12, $f18  
     mul.s   $f0, $f16, $f18  
     jr      $ra
```

# FP Example: Array Multiplication

- $X = X + Y \times Z$ 
  - All 32 x 32 matrices, 64-bit double-precision elements
- C code:
  - address of  $x$ ,  $y$ ,  $z$  in  $\$a0$ ,  $\$a1$ ,  $\$a2$  and  $i$ ,  $j$ ,  $k$  in  $\$s0$ ,  $\$s1$ ,  $\$s2$

```
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];  
}
```

# FP Example: Array Multiplication

## ■ MIPS code:

```
li    $t1, 32      # $t1 = 32 (row size/loop end)
li    $s0, 0       # i = 0; initialize 1st for loop
L1:   li    $s1, 0   # j = 0; restart 2nd for loop
L2:   li    $s2, 0   # k = 0; restart 3rd for loop
      sll   $t2, $s0, 5 # $t2 = i * 32 (size of row of x)
      addu  $t2, $t2, $s1 # $t2 = i * size(row) + j
      sll   $t2, $t2, 3  # $t2 = byte offset of [i][j]
      addu  $t2, $a0, $t2 # $t2 = byte address of x[i][j]
      ld    $f4, 0($t2)  # $f4 = 8 bytes of x[i][j]
L3:   sll   $t0, $s2, 5 # $t0 = k * 32 (size of row of z)
      addu  $t0, $t0, $s1 # $t0 = k * size(row) + j
      sll   $t0, $t0, 3  # $t0 = byte offset of [k][j]
      addu  $t0, $a2, $t0 # $t0 = byte address of z[k][j]
      ld    $f16, 0($t0) # $f16 = 8 bytes of z[k][j]
```

...

# FP Example: Array Multiplication

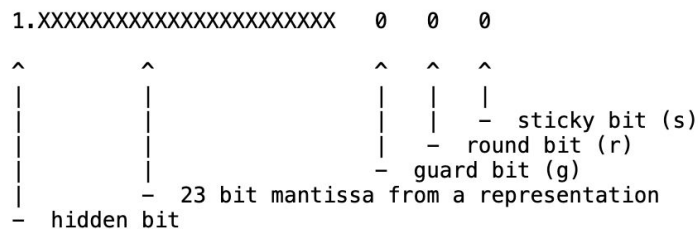
```
...
sll  $t0, $s0, 5      # $t0 = i*32 (size of row of y)
addu $t0, $t0, $s2     # $t0 = i*size(row) + k
sll  $t0, $t0, 3      # $t0 = byte offset of [i][k]
addu $t0, $a1, $t0     # $t0 = byte address of y[i][k]
l.d   $f18, 0($t0)     # $f18 = 8 bytes of y[i][k]
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16   # f4=x[i][j] + y[i][k]*z[k][j]
addiu $s2, $s2, 1      # $k k + 1
bne   $s2, $t1, L3     # if (k != 32) go to L3
s.d   $f4, 0($t2)      # x[i][j] = $f4
addiu $s1, $s1, 1      # $j = j + 1
bne   $s1, $t1, L2     # if (j != 32) go to L2
addiu $s0, $s0, 1      # $i = i + 1
bne   $s0, $t1, L1     # if (i != 32) go to L1
```

# Accurate Arithmetic

- Floating-point numbers are normally approximations for a number they cannot really represent.
  - There are cases that real numbers that being represented by an infinite variety between, 0 and 1.  $\Rightarrow$  But no more than  $2^{53}$  can be represented exactly in double precision floating point.
- IEEE 754 offers several modes of rounding to let the programmer pick the desired approximation.

# Accurate Arithmetic

- The way for rounding is straightforward  $\Rightarrow$  let the hardware to include **extra bits** in the calculation. (Actually, If every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round.)
- In IEEE 754, it keeps extra bits on the **right** during intermedia additions, called guard and round, respectively.
  - **guard**: the first of two extra bits kept on the right during intermediate calculations of floating-point number; used to improve rounding accuracy.
  - **round**: the second of two extra bits kept on the right during intermediate calculations of floating-point number
  - **sticky**: a bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit. (an extra bit (nonzero) on the right of round.)



The guard, round, and sticky are used internally. If the mantissa shifts to the right during calculation, the least significant bit will be kept in the extra bits, which will determine the round off.

# Accurate Arithmetic

- IEEE Standard 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.



# Summary

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
  - e.g., unsigned numbers, signed numbers, floating-point numbers
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs