Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers

    4,600,000,000 or 4.6×10^9

    0.0000000000000000000000000166 or 1.6×10^{-27}

Can't be represented by integer

- Like scientific notation
  - 2×10^{-7}
  - $+0.002 \times 10^{-4}$

Normalized (no leading zeros)

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types float and double in C

```
float   a;   // single precision
double b;  //double precision
```

Floating Point Standard- IEEE Std 754-1985

- Single precision - 32-bit

single: 8 bits | single: 23 bits

S	Exponent	Fraction

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

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

Significand= 1+fraction

- S: sign bit ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)
- Normalized number $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
  - Always has a leading 1, so no need to represent it explicitly (hidden bit)
- Exponent: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single precision: Bias = 127, Double precision: Bias = 1023

Floating-Point Example – single-precision

- What number is represented by the following single-precision float?

  $x = 1\,10000001\,01000\ldots00_2$ (32-bit)

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129-127)}$

  $= (-1) \times (1+1/4) \times 2^2$

  $= -5.0$

Floating-Point Example

- Represent –0.75 in single-precision floating point
  - $-0.75 = -(1/4+1/2)=(-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = 1000…00 Hidden 1 is not represented
  - Exponent = –1 + Bias=126=01111110_2

Answer: 1011111101000…00

Why uses bias in the exponents

- Easy to compare which number is larger
  - Just need to check the bit from left to right

8 bits		Bias=127		
127	01111111	254	11111110	
126	01111110	253	11111101	
		…..		
………..		……		
1	00000001	128	10000000	
0	00000000	127	01111111	
-1	111111111	….		
….				
-126	10000010	1	00000001	
-127	10000001	0	00000000	reserved (discussed later)
-128	10000000	255	11111111	reserved (discussed later)

  - Allow quick comparison of floating point numbers
    - from MSB to LSB (except the sign bit)

Floating Point Standard- IEEE Std 754-1985

- Double precision (64-bit)

double: 11 bits — double: 52 bits

S	Exponent	Fraction

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

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

- S: sign bit ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)
- Normalized number
  - Have hidden 1

$\pm 1.xxxxxxx_2 \times 2^{yyyy}$

- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Double: Bias = 1023

Floating-Point Example – double-precision

- What number is represented by the following double float?

  $x = 1011111111011000...00_2$ (64-bit)
  - S = 1
  - Fraction = $1000...00_2$
  - Exponent = 01111111101_2 = 1021

- $x = (-1)^1 \times (1 + .1_2) \times 2^{(1021 - 1023)}$

  $= (-1) \times (1+1/2) \times 2^{-2}$

  $= -3/8$

Floating-Point Example

- Represent –0.75 in double-precision floating point
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$ Hidden 1 is not represented
  - Exponent = –1 + Bias= -1+1023= $1022=01111111110_2$

Ans: 1011111111101000…00

IEEE 754 Encoding of FP number

- Encoding
  - Exp. 00…00 and 111…11 reserved
  - Exp.=00000000 and Fract.=00000…00 => 0
  - Exp.=0, and Fract. != 0 => denormalized number (discuss later)
  - Exp.=111..111 and Fract.= 000…000 => $\pm\infty$(discuss later)
  - Exp.=111…111 and Fract.!=0 => Non a Number (NaN) (discuss later)

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Single-Precision Range (for EXP 1 to 254)

- Smallest value
  - 000…01 00000000
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - Exponent = 1 – 127 = –126
  - Smallest value = $1.0 \times 2^{-126} \approx 1.2 \times 10^{-38}$
- Largest value
  - 111…110 1111111
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - Exponent = 254 – 127 = +127
  - Largest value ≈ $2.0 \times 2^{+127} \approx 3.4 \times 10^{+38}$

Exponents 00000000 and 11111111 reserved

S	Exponent	Fraction

Double-Precision Range (for EXP 1 to 2046)

Exponents 00000…00 and 11111…111 reserved

- Smallest value
  - 00000000001 0000….000
  - Exponent = 1 – 1023 = –1022
  - Fraction: 00000000000 ⇒ Significand = 1.0
  - 1.0×2^{-1022}
- Largest value
  - 11111111110 111….11111
  - Exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $2.0 \times 2^{+1023}$

S	Exponent	Fraction

Denormalized Numbers

- (Review) Smallest normalized value
  - 00000001 00000000……0000
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - Exponent = 1 – 127 = –126
  - Smallest value = 1.0×2^{-126}
- How to represent number smaller than $1.0\text{x}2^{-126}$?
- E.g. $0.5\text{x}2^{-126}$ =>Use denormalized number

S	Exponent	Fraction

Denormalized Numbers (32-bit)

- Exponent = **00000000**
- Fraction $\Rightarrow$ hidden bit is 0

$$x = (-1)^S \times (\text{Fraction}) \times 2^{-126}$$

$0.5\text{x}2^{-126}$ = 0 00000000 10000000000000...000

- Allow for gradual underflow
- Denormalized with fraction = 000...0 ➔ 0

$$x = (-1)^S \times (0 + 0) \times 2^{-126} = \pm 0.0$$

Two representations of 0.0! +0.0 and -0.0

Example

- Smallest positive single precision normalized number

$1.00000000\ldots00000_2 x 2^{-126}$

S	Exp	Fraction
0	0000 0001	0000 0000 0000 0000 0000 000
	2^{-126}	x1.0

- Smallest positive single precision denormalized no. (Hint: Fraction is 23-bit)

S	Exp	Fraction
0	0000 0000	0000 0000 0000 0000 0000 001
	2^{-126}	$x1/2^{23}$

Special number: Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - $\pm\infty$
  - Can be used in subsequent calculations, avoiding need for overflow check
  - E.g. $F+(+\infty)=+\infty$, or $F/\infty=0$
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., xx / 0.0
  - Can be used in subsequent calculations

Floating-Point Addition

- 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

    1.0015×10^{2}
- 4. Round and renormalize if necessary

    1.002×10^{2}
  - Renormalize may be required due to rounding
    - E.g. rounding 9.9999×10^{2}

Floating-Point Addition

Why do we shift the number with a smaller exponent?

- Example
  - $9.99999 \times 10^{40} + 1.610 \times 10^{-1}$
  - Shift 9.99999×10^{40} to 10^{-1} ➔ significand may be too large ➔ overflow
  - 1.610×10^{-1} to 10^{40} => significand may be very small, but still don't overflow, using round
  - Therefore, we shift the smaller number

Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (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

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

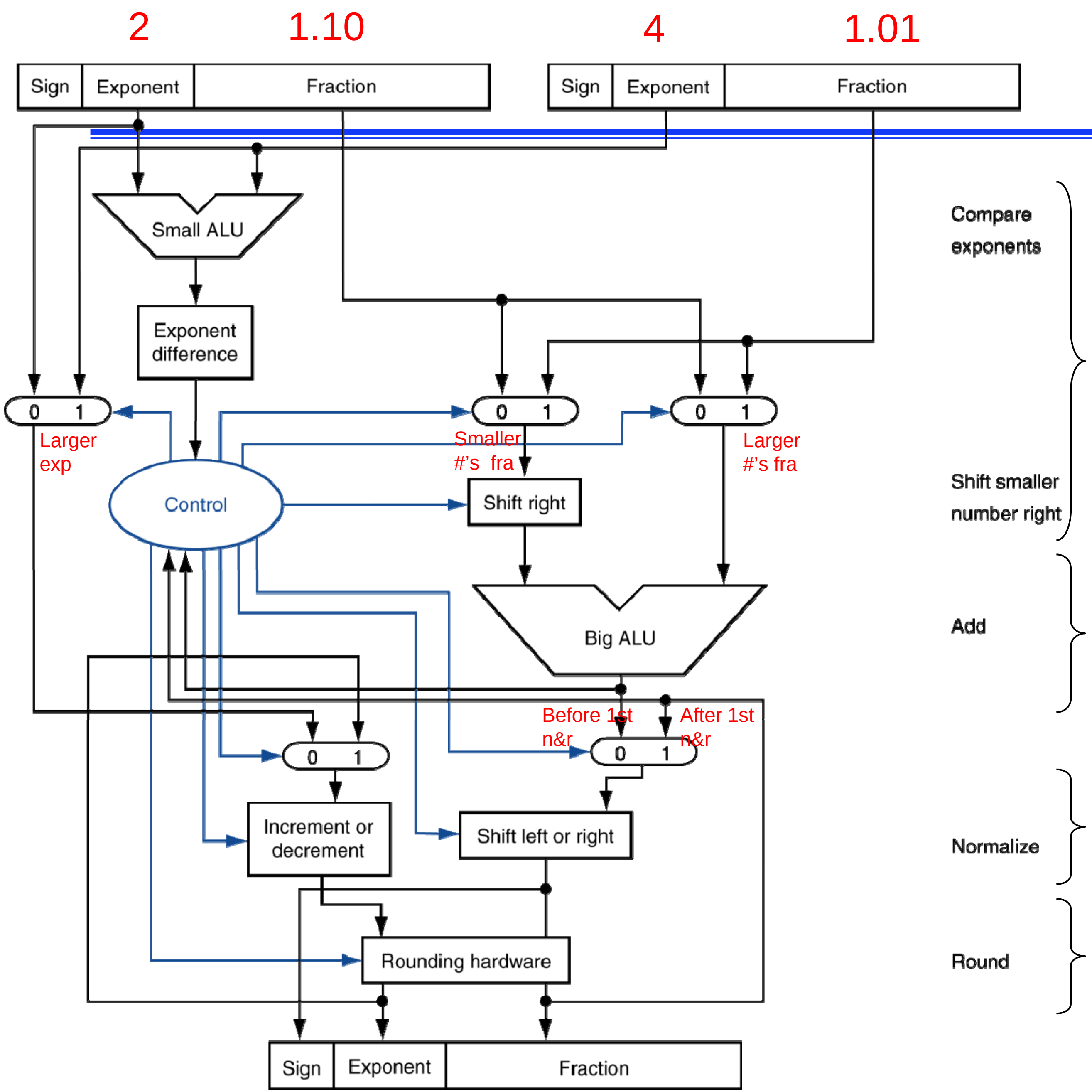
4. Round and renormalize if necessary

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

FP Adder Hardware

- Much more complex than integer adder
  - Steps includes shift exponents and fraction, add fraction, …, etc.
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined (see Chapter 4 about pipeline)

FP Adder Hardware



1. Align binary points
2. Add significands

Step 1

3. Normalize result & check for over/underflow
4. Round and renormalize if necessary

Step 2

Step 3

Step 4

Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum (described later)
  - 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
  - $+1.021 \times 10^6$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 1. Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) -127= –3 + 254 – 127

Remove one bias

- 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: $+ \times - \Rightarrow -$
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers for single precision
  - 32 single-precision: $f0, $f1, … $f31
  - These registers are also used for double-precision computation (described later)
- FP instructions operate only on FP registers
- Single-precision FP load and store instructions
  - lwc1, swc1 e.g., lwc1 $f8, 32($sp)
- Single-precision arithmetic e.g., add.s $f0, $f1, $f6
  - add.s, sub.s, mul.s, div.s
- Single-precision comparison e.g. c.lt.s $f3, $f4
  - c.*xx*.s (*xx* is eq, neq, lt, le, …)
  - Sets or clears FP condition-code bit

Branch on FP condition code true/false
bc1t, **bc1f** e.g., bc1t TargetLabel

FP Example: °F to °C

- C code:

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

  – Assume $gp can access three constant 5. 0, 9.0, and 32.0
  – Assume F in $f12, and put result in $f0

- Compiled MIPS code:

```
lwc1  $f16, const5($gp)
lwc1  $f18, const9($gp)
div.s  $f16, $f16, $f18          # f16 = 5.0/9.0
lwc1  $f18, const32($gp)
sub.s $f18, $f12, $f18           # f18 = F – 32.0
mul.s $f0,  $f16, $f18           # f0 = (5/9)x (F-32)
jr $ra
```

• $f0 and $f1 are used for FP results, similar to $v0 and $v1 for INT results

FP Instructions in MIPS for double-precision

- Use 32 FP registers $f0 - $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
- FP Double-precision load and store instructions
  - ldc1, sdc1
- Double-precision arithmetic
  - add.d, sub.d, mul.d, div.d

mul.d $f4, $f4, $f6

- Double-precision comparison
  - c.*xx*.d (*xx* is eq, lt, le, …)
  - Sets or clears FP condition-code bit

c.lt.d $f4, $f6

- Branch on FP condition code true or false
  - bc1t, bc1f
    - e.g., bc1t TargetLabel

FP Example: Array Multiplication

- $X = X + Y \times Z$
  - All 32×32 matrices, 64-bit double-precision elements
- C code:

```
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];
}
```

  - Addresses of x, y, z in \$a0, \$a1, \$a2, and i, j, k in \$s0, \$s1, \$s2

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]
    l.d  $f4, 0($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]
    l.d  $f16, 0($t0)   # $f16 = 8 bytes of z[k][j]
    …
```

the same as ldc1, but a pseudo instruction

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
```

Improve Accuracy: Guard and Round Bits

- 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
- Consider the addition $2.56\times10^0+2.34\times10^2=2.3656\times10^2$
- Without guard and round bits

$$0.02\times10^2+2.34\times10^2=2.36\times10^2$$

- With guard and round bits

$$0.0256\times10^2+2.3400\times10^2=2.3656\times10^2=2.37\times10^2$$

closer to accurate answer

Improve Accuracy: Rounding Modes

- 4 rounding modes
  - Round towards + ∞
  - Round towards - ∞
  - Round towards 0
    - truncate
  - Round to nearest (even)
    - Default
    - if the number falls midway, it is rounded to the nearest value with an even (zero) least significant bit
    - The sticky bit can improve the accuracy here (see next slide)

Improve Accuracy: Sticky Bit

- Sticky bit: one bit is set when there are nonzero bits to the right of the round bit.
  - Allow computer to see the difference between $0.50000..0_{10}$ and $0.50000..1_{10}$

- Without Sticky bit

  2.3450000000001 will be stored as 2.345

- With Sticky bit

  2.3450000000001 will be stored as 2.345 and sticky bit =1

- Used for rounding

  2.345 with sticky bit=1 is larger than 2.345

Interpretation of Data

The BIG Picture

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Associativity

- Is (x+y)+z equal to x+(y+z) ???

		(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

- Parallel Programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail
- Parallel execution strategies that work for integers may not work for floating-point numbers
  - Need to validate parallel programs under varying degrees of parallelism

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow