Arithmetic (Ch: 3)

Dr. Rajib Ranjan Maiti

CSIS, BITS-Pilani, Hyderabad

Floating Point

Floating Point

- IEEE 754 floating-point standard

Floating Point

- IEEE 754 floating-point standard

31	30	29	28	27	26	25	24	23	
s	Exponent								
1	8								

Floating Point

- IEEE 754 floating-point standard

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
s	Exponent								Fraction / Significand / Mantissa																							
1	8								23																						

- Floating-point numbers are of the form
  - $(-1)^s \times F \times 2^E$
- IEEE 754 uses a bias of +127 for single precision,
  - A true exponent of -1 is represented by $-1 + 127_{ten}$, $= 126_{ten} = 0111\ 1110_{two}$
  - A true exponent of +1 is represented by $+1 + 127_{ten} = 128_{ten} = 1000\ 0000_{two}$

Floating Point

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
s	Exponent								Fraction / Significand / Mantissa																							
1	8								23																					

- Biased exponent to true exponent
  - $(1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$
- Smallest value
  - $\pm 1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{two} \times 2^{-126}$
- Largest value
  - $\pm 1.1111\ 1111\ 1111\ 1111\ 1111\ 111_{two} \times 2^{127}$

Floating point: Special numbers

- All 0 bits in the exponent E ➔ reserved and indicates floating-point representation of **zero**
- **All 1's in E ➔ either NaN or Infinity**

Example 1

- Represent -0.75_{ten} in IEEE 754 single and double precision format
- **Decimal to binary**
  - $-0.75_{ten} = -0.11_{two}$
- In **scientific notation**,
  - $-0.75_{ten} = -0.11_{two} \times 2^{0}$
- In **normalized scientific** notation
  - $-0.75_{ten} = -1.1_{two} \times 2^{-1}$
- In **IEEE 754** single precision format
  - $(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$
  - The value in exponent field: true exponent + 127 = -1 +127 = 126
  - So, $(-1)^{1} \times (1 + 0.1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{126}$
  - **➔actual exponent = The value in Exponent field - 127**
  - **➔ the value in Exponent field = actual exponent + 127**

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
S	Exponent								fraction																						
1	0 1 1 1 1 1 1 0								1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																							

Example 1

- The double precision representation is
  - $(1)^{1}$ x (1 + $0.1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$) x 2^{1022}

Floating-Point Addition

- $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$
- Assume
  - only 4 decimal digits of the significand and 2 decimal digits of the exponent
- **Step 1.**
  - **Align the decimal point of the number that has the smaller exponent to the higher exponent.**
  - $1.610_{ten} \times 10^{-1} \rightarrow 0.1610_{ten} \times 10^{0} \rightarrow 0.01610_{ten} \times 10^{1}$
  - But, only 4 decimal digits in significand
    - so, after shifting, the number becomes $\mathbf{0.016_{ten} \times 10^{1}}$

Floating-Point Addition

- **Step 2.**
  - Add the significands:
  - 9.999_{ten}
  - $+ \underline{0.016_{ten}}$
  - 10.015_{ten}
  - So, sum = $10.015_{ten} \times 10^{1}$
- **Step 3**
  - This sum is not in normalized scientific notation
  - So adjust it:
    - $10.015_{ten} \times 10^{1} \rightarrow 1.0015_{ten} \times 10^{2}$

Floating-Point Addition

- **Step 2.**
  - Add the significands:
  - 9.999_{ten}
  - $+ \underline{0.016_{ten}}$
  - 10.015_{ten}
  - So, sum = $10.015_{ten} \times 10^{1}$
- **Step 3**
  - This sum is not in normalized scientific notation
  - So adjust it:
    - $10.015_{ten} \times 10^{1} \rightarrow 1.0015_{ten} \times 10^{2}$
- **Step 4.**
  - The significand can be only 4 digits long (excluding the sign)
    - So, round the number
  - $1.0015_{ten} \times 10^{2}$ is rounded to $1.002_{ten} \times 10^{2}$
  - Notice that the sum may no longer be normalized (e.g., all 9s) and we would need to perform step 3 again.

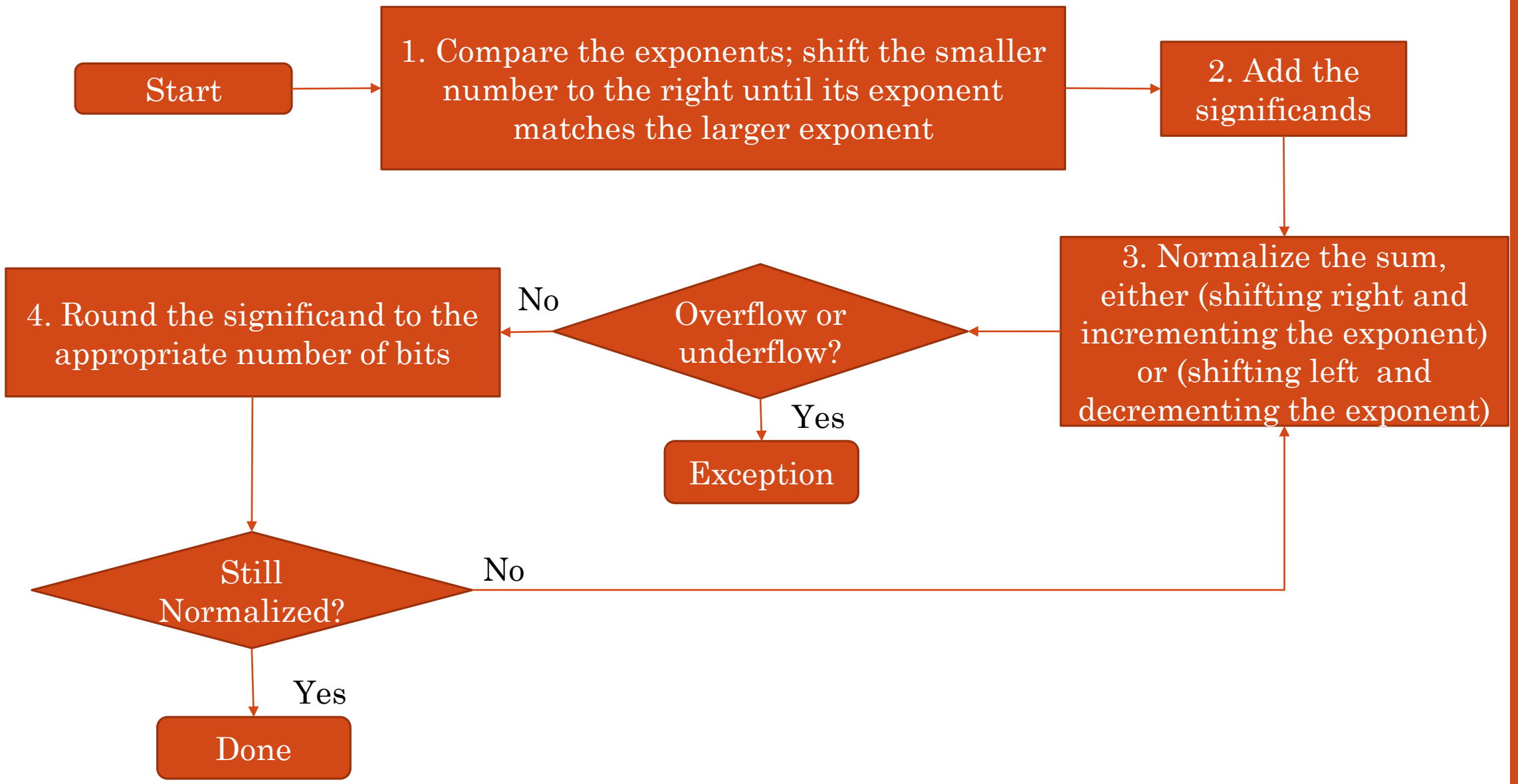
Binary Floating-Point Addition

- Add 0.5_{ten} and 0.4375_{ten} in binary
- Two numbers in normalized scientific notation (assume 4 bits of precision)
- $0.5_{ten} = 0.1_{two} = 0.1_{two} \times 2^{0} = 1.000_{two} \times 2^{-1}$
- $-0.4375_{ten} = -0.0111_{two} = -0.0111_{two} \times 2^{0} = -1.110_{two} \times 2^{-2}$
- **Step 1.**
  - the number with the lesser exponent = $-1.11_{two} \times 2^{-2}$
  - So, shift right until its exponent matches the larger number:
    - $1.110_{two} \times 2^{-2} \rightarrow 0.111_{two} \times 2^{-1}$
- **Step 2.**
  - Add the significands:
  - $1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$

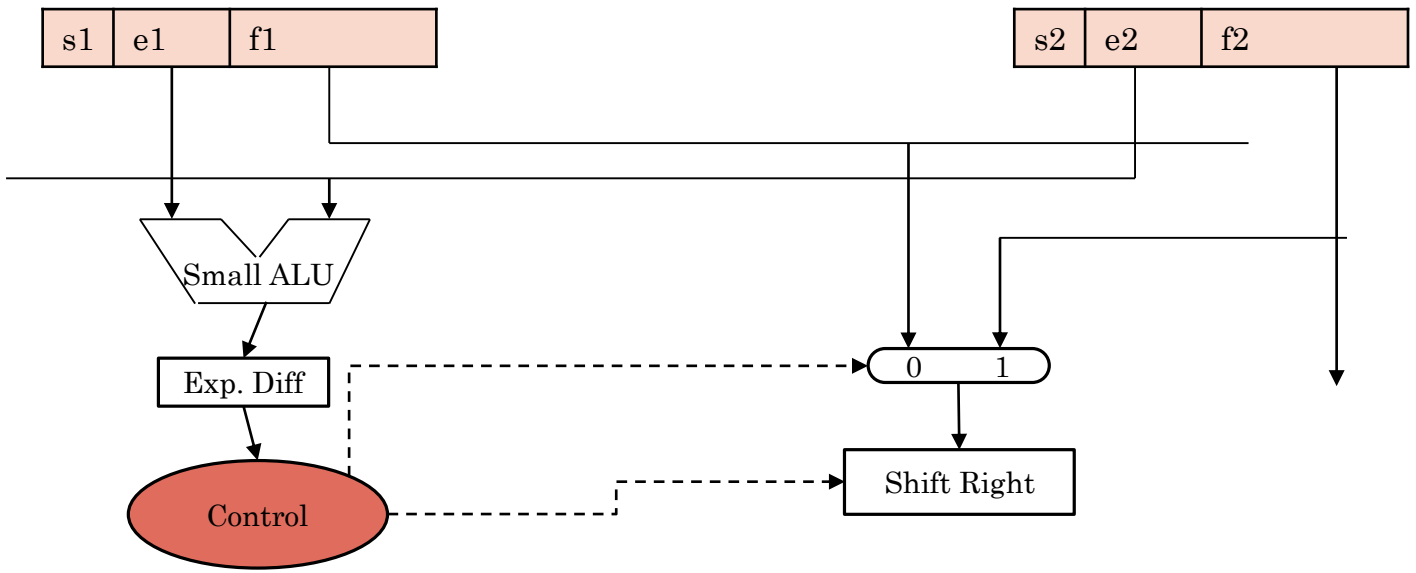
Binary Floating-Point Addition

- **Step 3.**
  - Normalize the sum, checking for overflow or underflow:
  - $0.001_{two} \times 2^{-1} \rightarrow 0.010_{two} \times 2^{-2} \rightarrow 0.100_{two} \times 2^{-3} \rightarrow 1.000_{two} \times 2^{-4}$
  - Since $+127 \geq -4 \geq -126$, there is no overflow or underflow.
  - The biased exponent = -4 + 127 = 123
- **Step 4.**
  - Round the sum:
  - $1.000_{two} \times 2^{-4}$
  - The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

Binary Floating-Point Addition

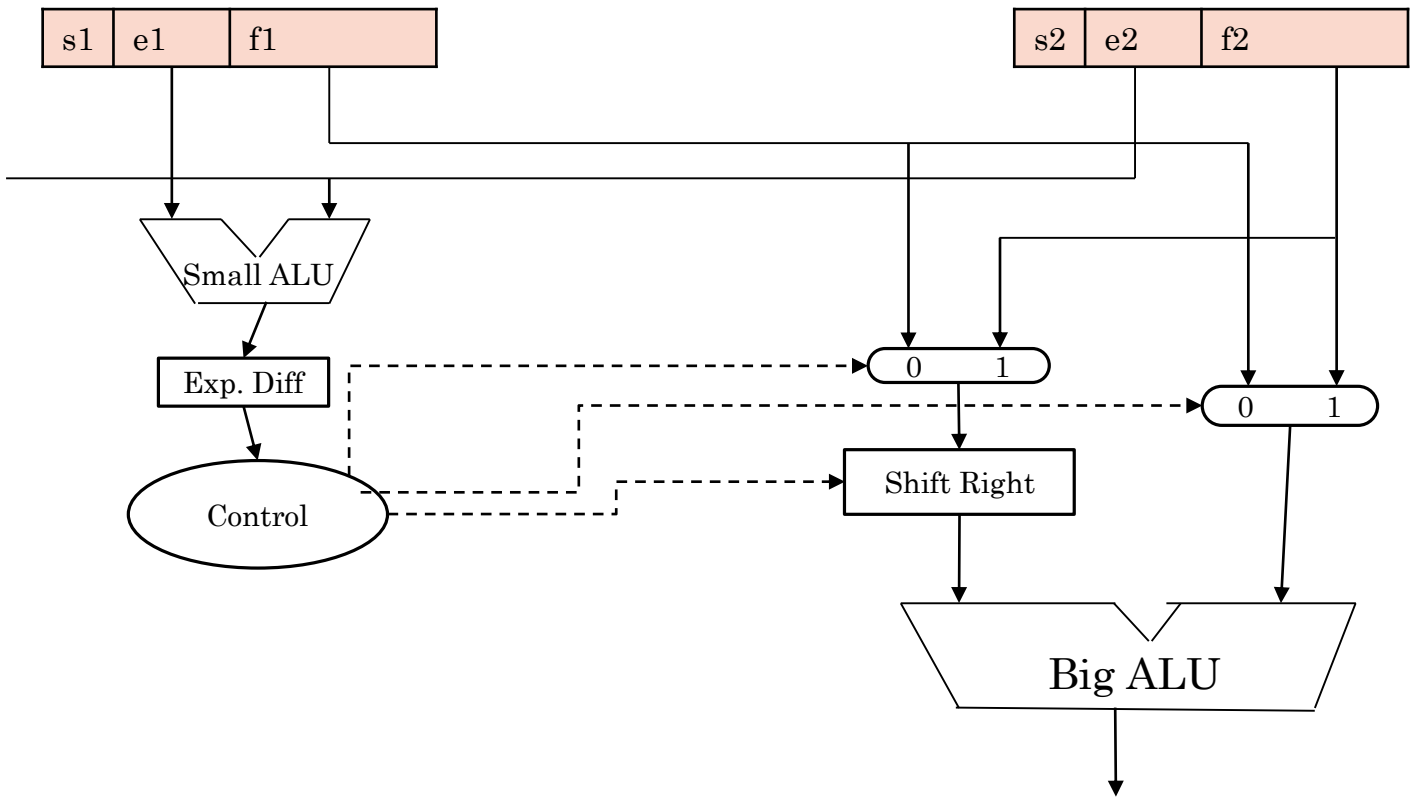


Compare the exponents;



Compare the exponents;

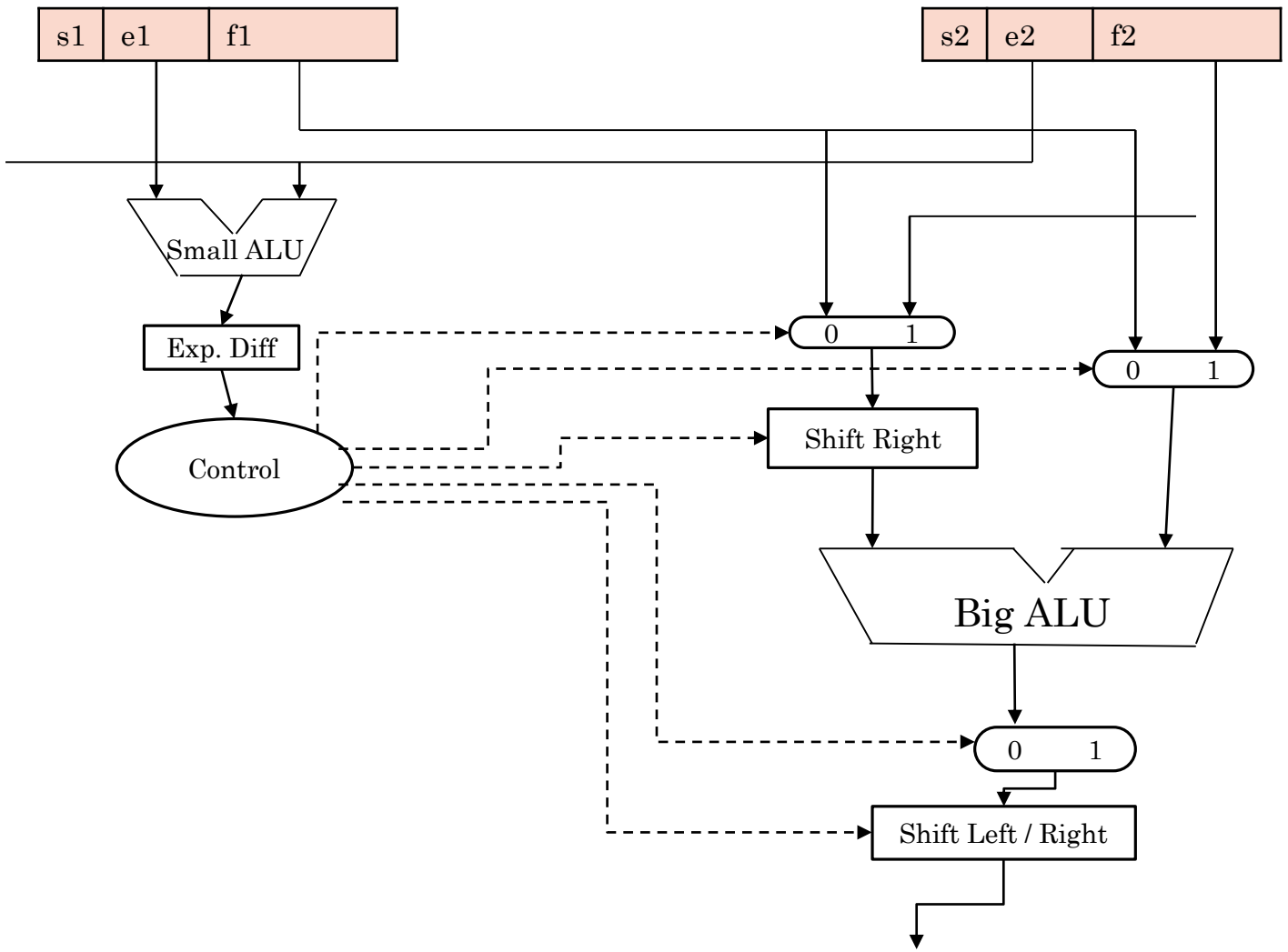
Shift if needed;



Compare the exponents;

Shift if needed;

Add Significands

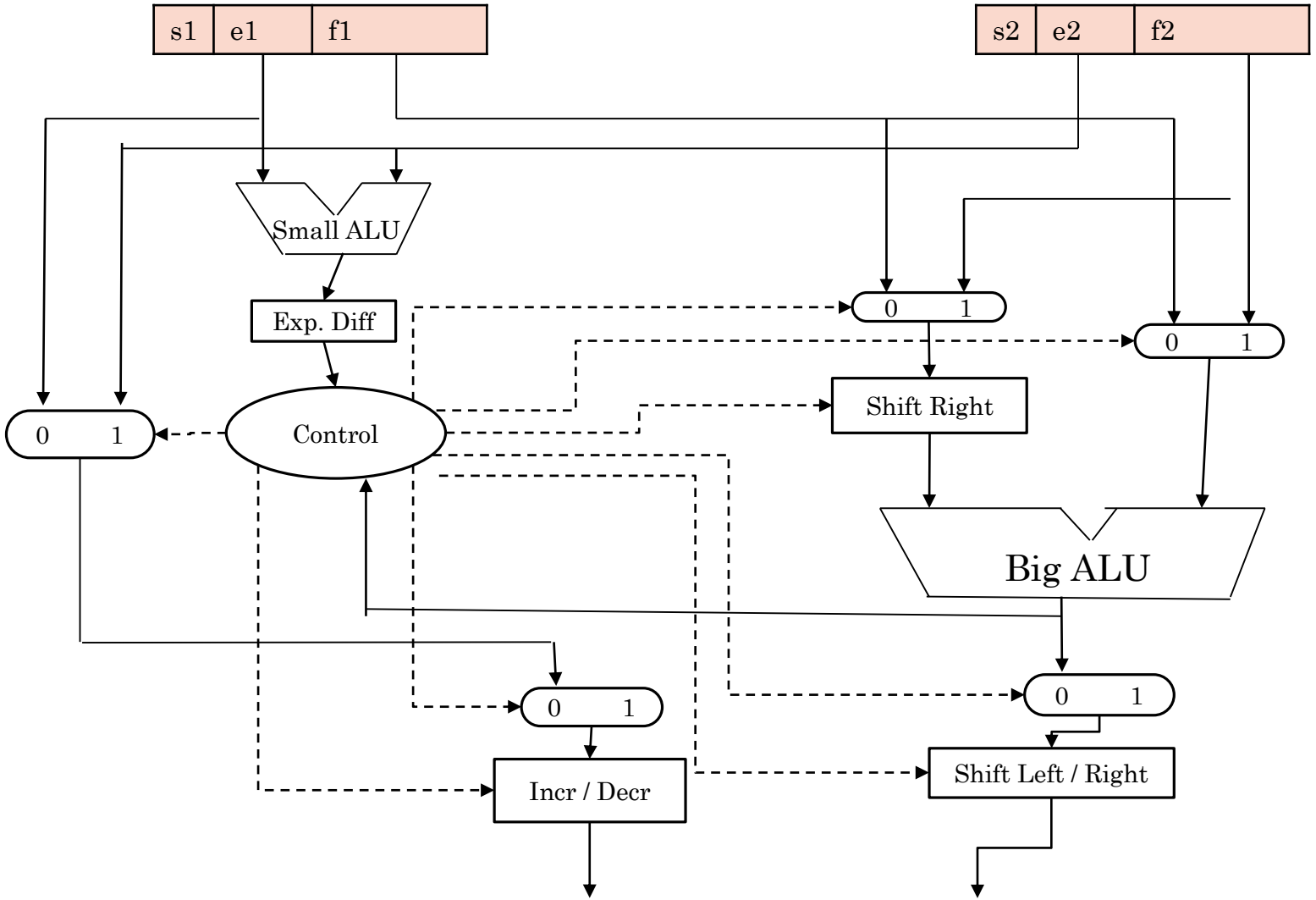


Compare the exponents;

Shift if needed;

Add Significands

Shift

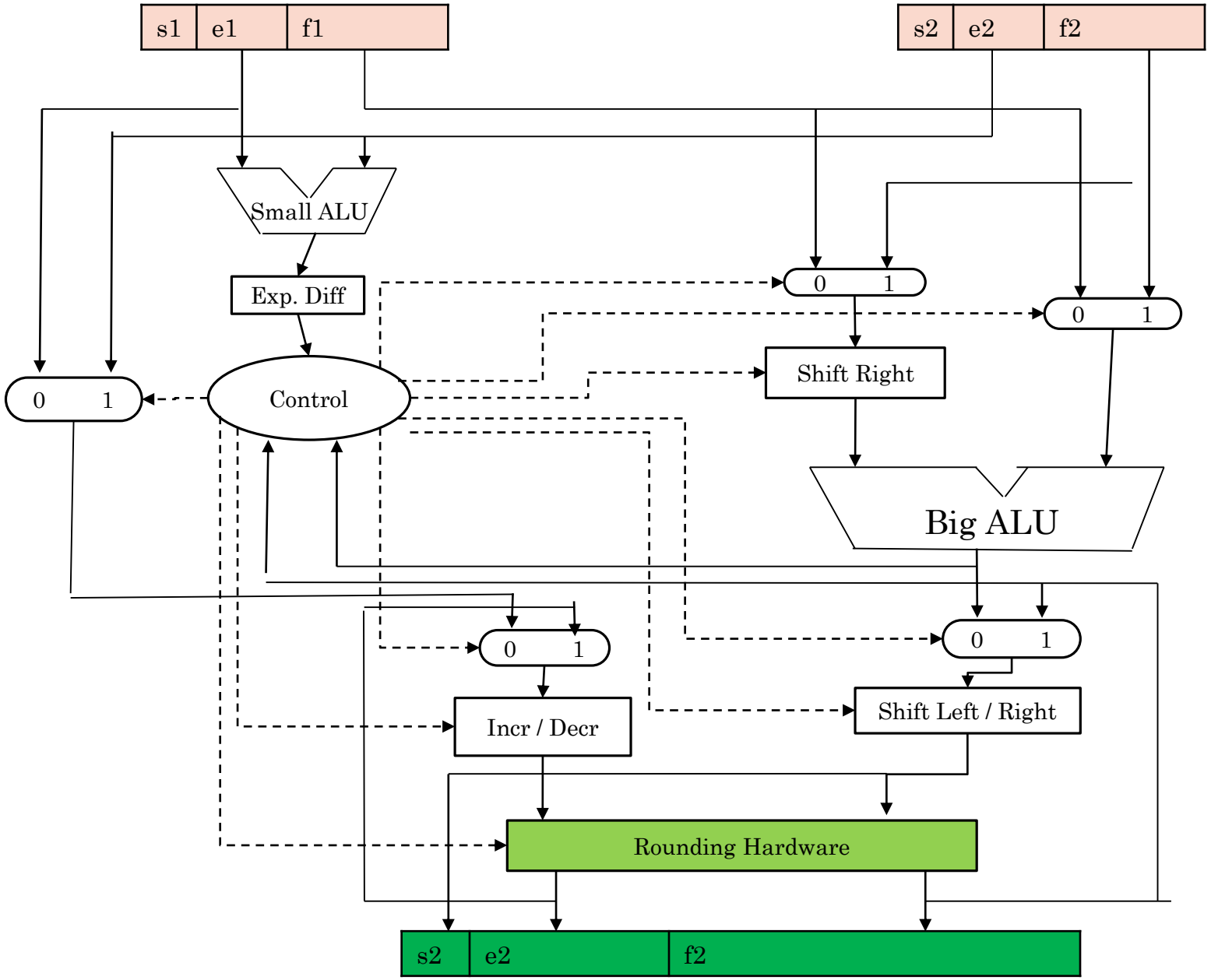


Compare the exponents;

Shift if needed;

Add Significands

Shift and Incr/Decr



Compare the exponents;

Shift if needed;

Add Significands

Shift and Incr/Decr

Round if needed

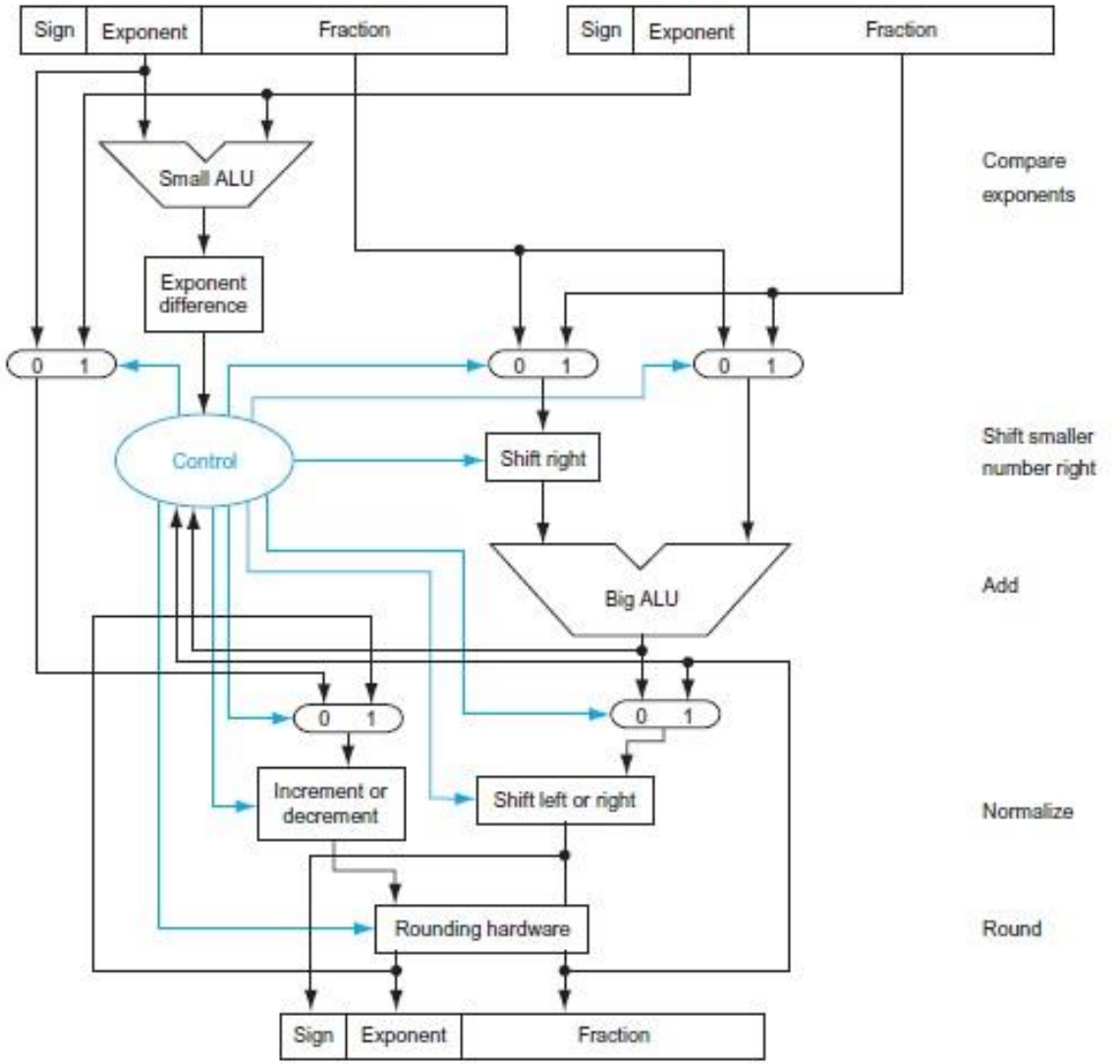


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the actual final result.

Floating-Point Multiplication

- We start by multiplying decimal numbers in scientific notation by hand:
  - $1.110_{ten} \times 10^{10}$
  - $\times 9.200_{ten} \times 10^{-5}$
  - ----------

- Assume only four digits of significand and two digits of exponent.

Floating-Point Multiplication

- We start by multiplying decimal numbers in scientific notation by hand:
  - $1.110_{ten} \times 10^{10}$
  - $\times 9.200_{ten} \times 10^{-5}$
  - ----------
- Assume only four digits of significand and two digits of exponent.
- **Step 1.**
  - The exponent of product = sum of the exponents of the operands = 10 + (-5) = 5
  - Cross-check with the biased exponents:
    - 10 ➔ 10 + 127 = 137, and 5 ➔ -5 + 127 = 122, so
    - New exponent 137 + 122 = 259 **(Wrong)**

Floating-Point Multiplication

- We start by multiplying decimal numbers in scientific notation by hand:
  - $1.110_{ten} \times 10^{10}$
  - $\times 9.200_{ten} \times 10^{-5}$
  - ----------

- Assume only four digits of significand and two digits of exponent.
- **Step 1.**
  - The exponent of product = sum of the exponents of the operands = 10 + (-5) = 5
  - Cross-check with the biased exponents:
    - **Remedy: New exponent = 137 + 122 - 127 = 259 - 127 = 132 (i.e, 5 + 127)**

Floating-Point Multiplication

- **Step 2.**
  - The multiplication of significands:
  - 1.110_{ten}
  - $\times 9.200_{ten}$
  - ----------
  - 0000
  - 0000
  - 2220
  - 9990
  - ----------
  - 10.212000_{ten}

Floating-Point Multiplication

- **Step 2.**
  - The multiplication of significands:
  - 1.110_{ten}
  - $\times 9.200_{ten}$
  - ----------
  - 0000
  - 0000
  - 2220
  - 9990
  - ----------
  - 10.212000_{ten}
- **Step 3.**
  - Normalize the product:
  - $10.212_{ten} \times 10^5 = 1.0212_{ten} \times 10^6$

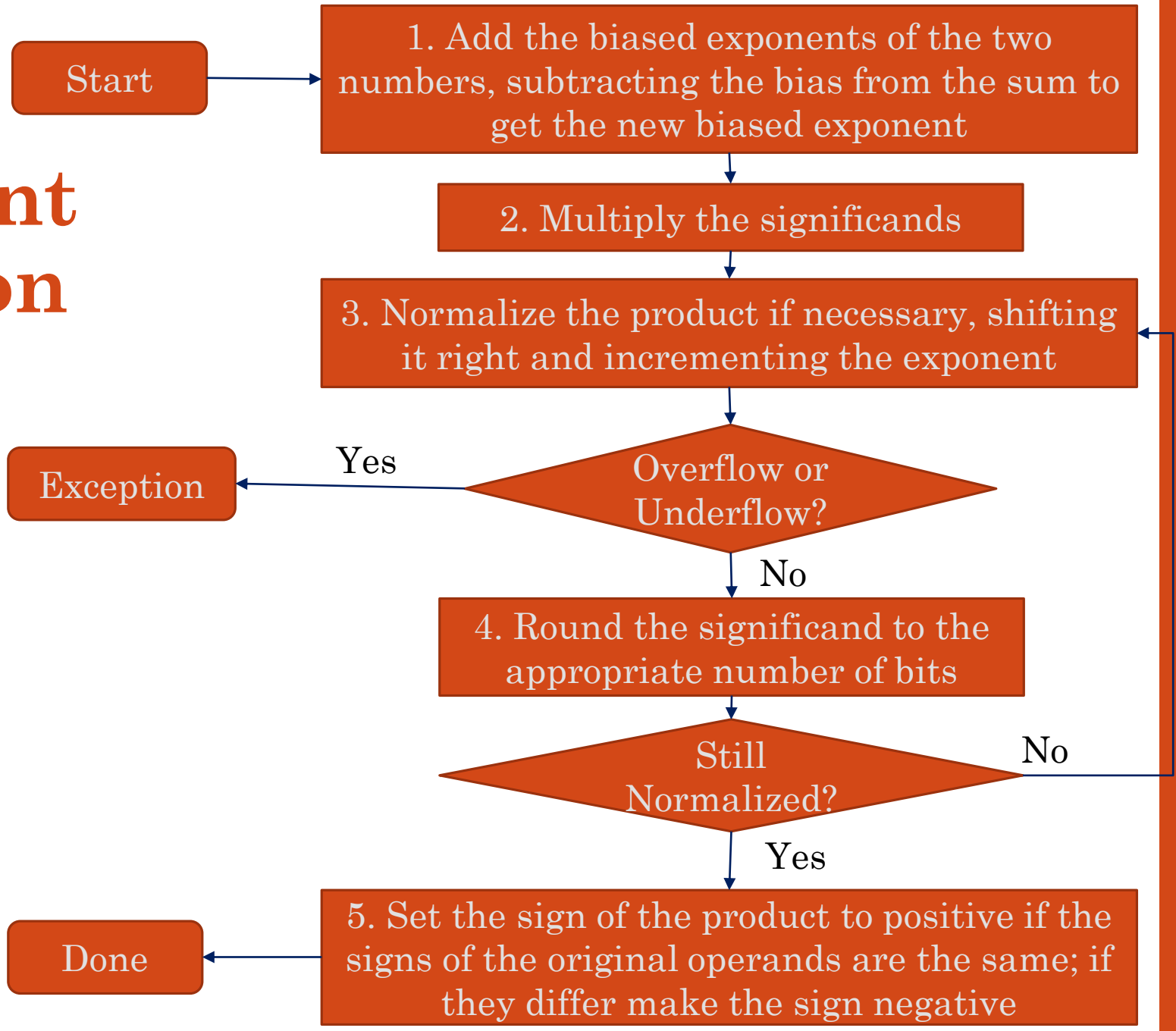
Floating-Point Multiplication

- **Step 2.**
  - The multiplication of significands:
  - 1.110_{ten}
  - $\times 9.200_{ten}$
  - ----------
  - 0000
  - 0000
  - 2220
  - 9990
  - ----------
  - 10.212000_{ten}
- **Step 3.**
  - Normalize the product:
  - $10.212_{ten} \times 10^5 = 1.0212_{ten} \times 10^6$
- **Step 4.**
  - The significand is only 4 digits long (excluding the sign), so round the number.
  - $1.0212_{ten} \times 10^6 \rightarrow 1.021_{ten} \times 10^6$

Floating-Point Multiplication

- **Step 2.**
  - The multiplication of significands:
  - 1.110_{ten}
  - $\times 9.200_{ten}$
  - ----------
  - 0000
  - 0000
  - 2220
  - 9990
  - ----------
  - 10.212000_{ten}
- **Step 3.**
  - Normalize the product:
  - $10.212_{ten} \times 10^5 = 1.0212_{ten} \times 10^6$
- **Step 4.**
  - The significand is only 4 digits long (excluding the sign), so round the number.
  - $1.0212_{ten} \times 10^6 \rightarrow 1.021_{ten} \times 10^6$
- **Step 5.**
  - The sign of the product:
    - based on the signs of operands.
  - If both are same, the sign is positive;
    - Otherwise, negative.
  - Hence,
    - Product = $+1.021_{ten} \times 10^6$

Floating-Point Multiplication



Floating-Point Instructions in MIPS

- MIPS supports the IEEE 754 single and double precision formats

Instruction	Meaning	Comment
add.s $f0, $f1, $f2	$f0 ← $f1 + $f2	Single precision
sub.s $f0, $f1, $f2	$f0 ← $f1 - $f2	Single precision
mul.s $f0, $f1, $f2	$f0 ← $f1 * $f2	Single precision
div.s $f0, $f1, $f2	$f0 ← $f1 / $f2	Single precision
c.x.s $f2, $f4	if $f2 <= $f4 then code = 1 else code = 0	x = le
add.d $f0, $f2, $f4		
sub.d $f0, $f2, $f4		
mul.d $f0, $f2, $f4		
div.d $f0, $f2, $f4		
c.x.d $f0, $f2		
bc1t label	Branch if FP condition flag 0 is true	Conditional Branch
bc1t 1, label	Branch if FP condition flag 1 is true	

Floating-Point Instructions in MIPS

- Floating-point comparison sets a bit to true or false (depending on the comparison condition), then
  - a floating-point branch decides whether or not to branch (depending on the condition)

Floating-Point Instructions in MIPS

- The MIPS code to load two single precision numbers from memory, add them, and then store the sum :
  - lwc1 $f4,c($sp) # Load 32-bit F.P. number into F4
  - lwc1 $f6,a($sp) # Load 32-bit F.P. number into F6
  - add.s $f2,$f4,$f6 # F2 = F4 + F6 single precision
  - swc1 $f2,b($sp) # Store 32-bit F.P. number from F2
- For double precision
  - use the even register number as its name
  - For example, $f2 and $f3 form the double precision register $f2

Compiling a Floating-Point C Program into MIPS

- C code
  - float f2c (float fahr){
    - return ((5.0/9.0) *(fahr – 32.0));
  - }
  - Assume that the fl oating-point argument fahr is passed in $f12 and the result should go in $f0. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

Compiling a Floating-Point C Program into MIPS

- C code
  - float f2c (float fahr){
    - return ((5.0/9.0) *(fahr – 32.0));
  - }
  - Assume that the fl oating-point argument fahr is passed in $f12 and the result should go in $f0. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?
- Ans
  - assume that the compiler places the three floating-point constants in memory within easy reach of the global pointer $gp.

Compiling a Floating-Point C Program into MIPS

- C code
  - float f2c (float fahr){
    - return ((5.0/9.0) *(fahr – 32.0));
  - }
  - Assume that the fl oating-point argument fahr is passed in $f12 and the result should go in $f0. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

- Ans
  - assume that the compiler places the three floating-point constants in memory within easy reach of the global pointer $gp.
  - The first two instructions load the constants 5.0 and 9.0 into floating-point registers:
    - f2c:
      - lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 in memory)
      - lwc1 $f18,const9($gp) # $f18 = 9.0 (9.0 in memory)

Compiling a Floating-Point C Program into MIPS

- C code
  - float f2c (float fahr){
    - return ((5.0/9.0) *(fahr – 32.0));
  - }
  - Assume that the fl oating-point argument fahr is passed in $f12 and the result should go in $f0. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

- Ans
  - assume that the compiler places the three floating-point constants in memory within easy reach of the global pointer $gp.
  - The first two instructions load the constants 5.0 and 9.0 into floating-point registers:
    - f2c:
      - lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 in memory)
      - lwc1 $f18,const9($gp) # $f18 = 9.0 (9.0 in memory)
      - div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0

Compiling a Floating-Point C Program into MIPS

- C code
  - float f2c (float fahr){
    - return ((5.0/9.0) *(fahr – 32.0));
  - }
  - Assume that the fl oating-point argument fahr is passed in $f12 and the result should go in $f0. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

- Ans
  - assume that the compiler places the three floating-point constants in memory within easy reach of the global pointer $gp.
  - The first two instructions load the constants 5.0 and 9.0 into floating-point registers:
    - f2c:
      - lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 in memory)
      - lwc1 $f18,const9($gp) # $f18 = 9.0 (9.0 in memory)
      - div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
      - lwc1 $f18, const32($gp) # $f18 = 32.0
      - sub.s $f18, $f12, $f18 # $f18 = fahr – 32.0

Compiling a Floating-Point C Program into MIPS

- C code
  - float f2c (float fahr){
    - return ((5.0/9.0) *(fahr – 32.0));
  - }
  - Assume that the fl oating-point argument fahr is passed in $f12 and the result should go in $f0. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

- Ans
  - assume that the compiler places the three floating-point constants in memory within easy reach of the global pointer $gp.
  - The first two instructions load the constants 5.0 and 9.0 into floating-point registers:
    - f2c:
      - lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 in memory)
      - lwc1 $f18,const9($gp) # $f18 = 9.0 (9.0 in memory)
      - div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
      - lwc1 $f18, const32($gp) # $f18 = 32.0
      - sub.s $f18, $f12, $f18 # $f18 = fahr – 32.0
      - mul.s $f0, $f16, $f18 # $f0 = (5/9)*(fahr – 32.0), placing the product in $f0 as the return result
      - jr $ra # return

Rounding with Guard Digits

- Compute $2.56_{ten} \times 10^0 + 2.34_{ten} \times 10^2$ first with guard and round digits, and then without them.
  - Assuming that we have three significant decimal digits.
  - Round to the nearest decimal number with three significant decimal digits
- Ans.
  - **First,** align the exponents, so $2.56_{ten} \times 10^0$ becomes $0.0256_{ten} \times 10^2$.
  - Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents.
    - The **guard digit holds 5 and the round digit holds 6**.
  - 2.3400_{ten}
  - $+ \underline{0.0256_{ten}}$
  - 2.3656_{ten}
  - Thus, the sum is $2.3656_{ten} \times 10^2$
  - Rounding the sum up with three significant digits yields $2.37_{ten} \times 10^2$

Rounding with Guard Digits

- Compute $2.56_{ten} \times 10^0 + 2.34_{ten} \times 10^2$ first with guard and round digits, and then without them.
  - Assuming that we have three significant decimal digits.
  - Round to the nearest decimal number with three significant decimal digits
- Ans.
  - **Second,** Doing this *without* guard and round digits drops two digits from the calculation.
  - The new sum is then
  - 2.34_{ten}
  - $+ \underline{0.02}_{ten}$
  - 2.36_{ten}
  - The answer is $2.36_{ten} \times 10^2$

Fallacies and Pitfalls

Fallacies

- *Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2.*
  - For example, suppose we want to divide -5_{ten} by 4_{ten}; the quotient should be -1_{ten}.
  - In two's complement, -5_{ten} = 1111 1111 1111 1111 1111 1111 1111 1011_{two}
  - Shifting right by two:
  - 0011 1111 1111 1111 1111 1111 1111 1110_{two} ; **Is this correct?**

Fallacies

- *Parallel execution strategies that work for integer data types also work for floating-point data types.*
  - Discussion start point:
  - Programs have typically been written first to run sequentially before being rewritten to run concurrently, so a natural question is, “Do the two versions get the same answer?” If the answer is no, you presume there is a bug in the parallel version that you need to track down.
  - *Hint:* since floating-point addition is not associative, the assumption does not hold.

Fallacies

- *Only theoretical mathematicians care about floating-point accuracy.*
  - A story involving a math professor at Lynchburg College in Virginia, Thomas Nicely, discovered a bug in floating point division algorithm discovered by Intel in September 1994.
  - Intel claimed that the bug will affect only the average spreadsheet user once in 27000 years.
  - But, soon IBM Research claimed that the bug will affect such a user every 24 days.
  - Intel was then forced to offer a free exchange of their existing processer with an updated processor that corrects the flaw

Pitfalls

- *Floating-point addition is not associative.*
  - Problems occur when adding two large numbers of opposite signs plus a small number.
  - For example, let's see if $c + (a + b) = (c + a) + b$.
  - Assume a, b and c are single precision numbers and c = $-1.5_{ten} \times 10^{38}$, a = $1.5_{ten} \times 10^{38}$, and b = 1.0
  - c + **(a + b)** = $-1.5_{ten} \times 10^{38}$ + **($1.5_{ten} \times 10^{38} + 1.0$)** = $-1.5_{ten} \times 10^{38} + 1.5_{ten} \times 10^{38} = 0$
  - **(c + a)** + b = **($-1.5_{ten} \times 10^{38} + 1.5_{ten} \times 10^{38}$)** + 1.0 = 0.0 + 1.0 = 1.0

Pitfalls

- *The MIPS instruction add immediate unsigned* (addiu) *sign-extends its 16-bit immediate field.*
  - MIPS has no subtract immediate instruction, and negative numbers need sign extension, so the MIPS architects decided to sign-extend the immediate field.


Thank You