Computer Arithmetic
Floating Point

Chapter 3.6
EEC170 FQ 2005

About Floating Point Arithmetic

- Arithmetic basic operations on floating point numbers are:
  - Add, Subtract, Multiply, Divide
  - Transcendental operations (sine, cosine, log, exp, etc.) are synthesized from these

Floating Point Addition

- Like addition using base 10 scientific representation:
  - Align decimal points
  - Add
  - Normalize the result

Example:

$$\begin{array}{l} 9.998 \times 10^{2} \\ +\ 5.0 \times 10^{-1} \\ \hline \end{array}$$

Align (make smaller # same exp. as larger, why?):

$$\begin{array}{l} 9.998 \times 10^{2} \\ +\ .005 \times 10^{2} \\ \hline \end{array}$$

Floating Point Addition (cont.)

Add:

$$\begin{array}{l} 9.998 \times 10^{2} \\ +\ .005 \times 10^{2} \\ \hline 10.003 \times 10^{2} \end{array}$$

Normalize (integer part must be => 1, <= 9)

$$10.003 \times 10^{2} = 1.0003 \times 10^{3}$$

- Observation: By hand, the precision is unlimited. For computer hardware, precision is limited to a fixed number of bits.

IEEE Floating Point Standard (754)

- IEEE Single precision format:

- exponent is biased by 127, mantissa has a hidden 1.

Binary Floating Point Add

- Similar to add in base 10 scientific, but must operate on standard floating point representation, use basic computer operations

Example:

```
0.25 = 0 01111101   00000000000000000000000
100  = 0 10000101   10010000000000000000000
                (1.)
                (1.)
```

- Align: compare exponents by subtracting
  - Sign of result tells which is larger
  - Magnitude of result tells how many places smaller must be moved

```
B:  10000101
A: -01111101
------------
    00001000   # B is larger  by 1000two (8ten)
```

Binary Floating Point Add (cont.)

- Shift smaller number to the right, by magnitude of exponent subtraction, include hidden bit in shifting.
- Set smaller exponent equal to larger exponent

```
  0 10000101   00000001000000000000000
           (0.)

  0 10000101   10010000000000000000000
           (1.)
```

- Add mantissas (including hidden bits)

```
  0 10000101   00000001000000000000000
           (0.)

+ 0 10000101   10010000000000000000000
           (1.)
  ---------------------------------
  0 10000101   10010001000000000000000
           (1.)
```

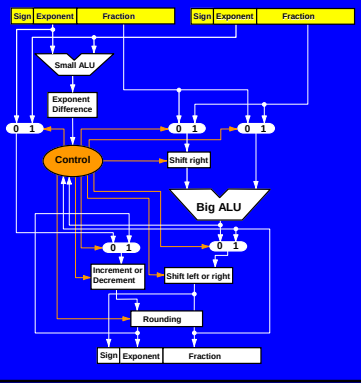

Binary Floating Point Add (cont.)

- Normalize the result (get "hidden bit" to be 1)
  - This example is already normalized, when would the result not be normalized?

Floating Point Subtract

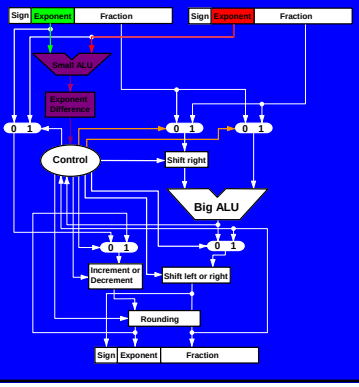
- Alignment binary point, like addition
- Then algorithm for sign magnitude numbers takes over
  - Negate the second operand, then add
- Must set sign bit of result to be consistent with outcome

Floating Point Add/Sub Hardware



Floating Point Add/Sub Hardware

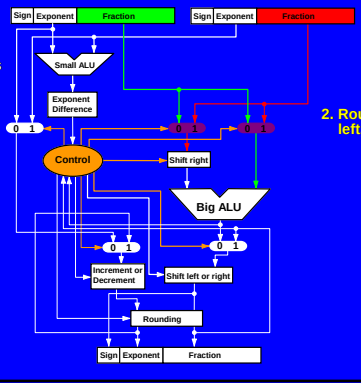
1. Compare Exponents



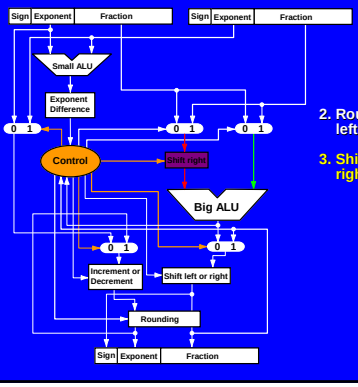
Floating Point Add/Sub Hardware

1. Compare Exponents

2. Route smaller # left, larger right

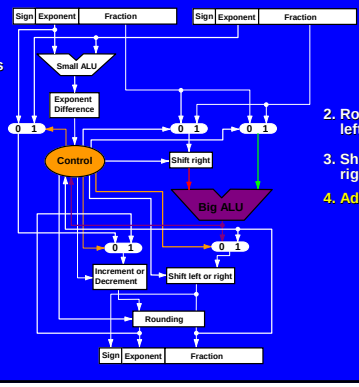


Floating Point Add/Sub Hardware



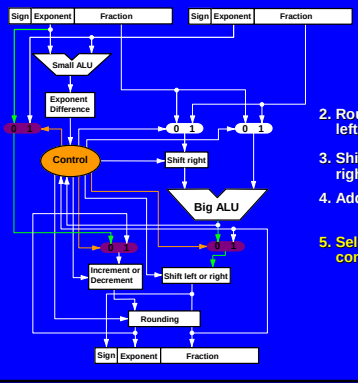
1. Compare Exponents
2. Route smaller # left, larger right
3. Shift smaller # right

Floating Point Add/Sub Hardware



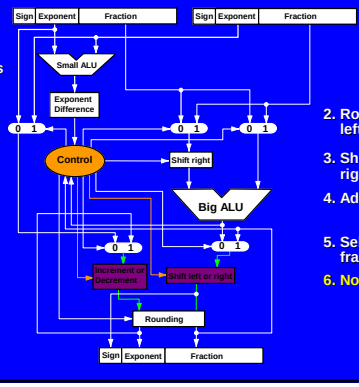
1. Compare Exponents
2. Route smaller # left, larger right
3. Shift smaller # right
4. Add

Floating Point Add/Sub Hardware



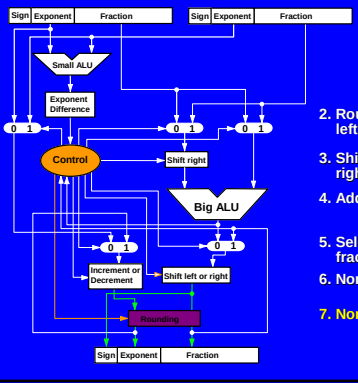
1. Compare Exponents
2. Route smaller # left, larger right
3. Shift smaller # right
4. Add
5. Select exponent, computed fraction

Floating Point Add/Sub Hardware



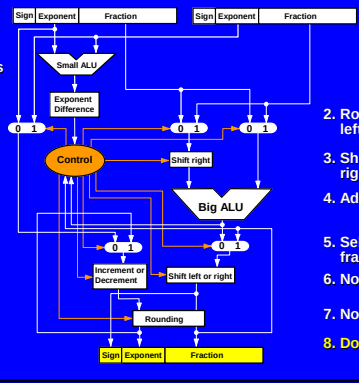
1. Compare Exponents
2. Route smaller # left, larger right
3. Shift smaller # right
4. Add
5. Select exponent, fraction
6. Normalize

Floating Point Add/Sub Hardware



1. Compare Exponents
2. Route smaller # left, larger right
3. Shift smaller # right
4. Add
5. Select exponent, fraction
6. Normalize
7. Normalize

Floating Point Add/Sub Hardware



1. Compare Exponents
2. Route smaller # left, larger right
3. Shift smaller # right
4. Add
5. Select exponent, fraction
6. Normalize
7. Normalize
8. Done

Floating Point Multiply

- Similar to multiply in base 10 scientific:
  - Multiply mantissas
  - Add exponents
  - Normalize

Example:

$$\begin{array}{l} 3.0 \times 10^{2} \\ +\ 5.0 \times 10^{-1} \\ \hline 15.0 \times 10^{1} \rightarrow 1.5 \times 10^{2} \end{array}$$

Binary Floating Point Multiply

- Similar to base 10, but must deal with standard floating point representation

Example (using only 4 mantissa bits):

```
    0 10000100  0100
x   1 00111100  1100
    ---------------
```

- Multiply the mantissas, don't forget the hidden 1s:

```
    1.0100
x   1.1100
---------
    00000
   00000
  10100
 10100
10100
---------
1000110000  -> 10.00110000
```


Binary Floating Point Multiply (cont.)

- Add exponents:

```
 10000100
 00111100
 --------
 11000000
```

- Exponent now has double bias (one for each term), so subtract 127:

```
 11000000
-01111111
 --------
 01000001
```

- Compute the result sign bit
  - XOR of operand sign bits
- Reconstruct the result

```
1 01000001 10.00110000
```


Binary Floating Point Multiply (cont.)

- Normalize the result
  - Shift binary point if integer part is >1, increment exponent:

```
1 01000010 1.001100000
```

- "Trim" excess bits low order bits and trim hidden bit

```
1 01000010 0011
```


Floating Point Division

- Dual of multiplication
  - Divide mantissas (include hidden hits)
  - Subtract exponents. Must add back in bias (127) because it has been eliminated
  - Normalize result
    - hidden bit must be 1
    - may require left shifts, exponent decrement
  - Trim excess bits

Arithmetic Instructions

- MIPS processor has separate instructions for integer and floating point arithmetic
- Floating point operations are relatively slow, want to use integer if appropriate:
  - Integer Add 1 time unit
  - Fl. Pt. Add 2 time units
  - Fl. Pt. Multiply 3 time units
  - Fl. Pt. Divide 20 time units

Floating Point Range: Maximum

- Floating point can represent very large numbers. The largest number is:

```
0 11111110 (1.)11111111111111111111111
```

  - Maximum exponent is: 254 - 127 = 127 => weight of 2^{127}.
  - Mantissa is: $1 + 1/2 + 1/4 + ... + 1/2^{23} = 2 - 1/2^{23} \cong 2$
  - Thus the largest number $\cong 2^{128} \cong 10^{38}$
  - Do we need larger numbers? Probably not:
    - Radius of the universe = age of the universe x speed of light = (10^{10} years) x ($10^{2.5}$ days/year) x (10^{5} seconds/day) x ($10^{8.5}$ meters/sec) = 10^{26} meters
    - Bill Gates' net worth is < 10^{11} dollars

Floating Point Range: Minimum

- Floating point can represent very small positive numbers. Normally, the smallest positive number is:

```
0 00000001 (1.)00000000000000000000000
```

  - Minimum exponent is: 1 - 127 = -126 => weight of 2^{-126}.
  - Mantissa is: 1
  - Thus the smallest number $\cong 2^{-126} \cong 10^{-38}$
  - Do we need smaller numbers? Probably not:
    - Mass of an electron: 10^{-27} grams

  But somebody was small minded:

Denormalized Numbers

- There is no hidden 1 when the exponent is 00000000, i.e. the number is no longer normalized
- 00000000 exponent has same meaning as 00000001, i.e., 2^{-126}
- Denormalized number can have a leading 1 in any bit position, allowing even smaller numbers
  - However smaller numbers have fewer bits of precision
- Smallest denormalized number is:

```
0 00000000 (0.)00000000000000000000001
```

  which represents $2^{-149} \cong 10^{-45}$
  - only has 1 bit of precision
- Denormalization complicates floating point hardware design, questionable usefulness

Overflow and Underflow

- When a result is larger than

```
0 11111110 (1.)11111111111111111111111
```

  overflow occurs, and the result becomes +infinity, which is represented as:

```
0 11111111 00000000000000000000000
```

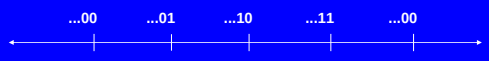
  - -infinity is the same, but with a 1 in the sign bit
- Operations with infinity will produce the expected result: # + infinity = infinity
- Underflow occurs when the result is smaller than the smallest denormalized number
  - result becomes 0

Precision

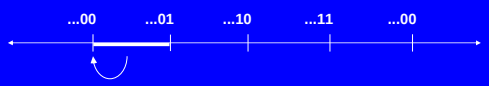
- The set of all real numbers is infinite.
- The set of floating point numbers is finite, 2^{32}
- We must map a range of real numbers onto a single floating point number
  - The mapping cannot be precise, some precision is lost. How much?

Integer Precision

- First, consider precision of mapping real numbers to binary integers:

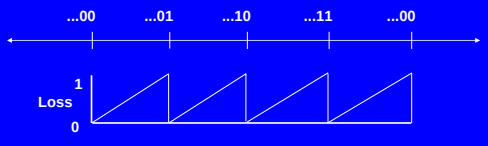


- The real numbers between two integers must be mapped to one of those integers.
  - Simplest method is truncation, discard the fractional part:



Truncation Precision

- With truncation, assuming real numbers are randomly distributed between integers, what is the expected loss in precision for a given number?

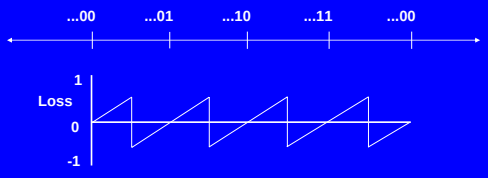


Cumulative Truncation Error

- Adding up a large list of numbers can quickly result in significant cumulative error.
- What is the expected roundoff error for adding a large list?
- What is the maximum roundoff error?
- The fastest computers can add more than 10^9 numbers/second, => cumulative error can reach 32-bits (4×10^9) in a few seconds

Rounding Precision

- With rounding to nearest integer, assuming real numbers are randomly distributed between integers, what is the magnitude of the expected loss in precision for a given number?



Cumulative Roundoff Error

- What is the expected cumulative roundoff error when adding up a large list of numbers?
- What is the maximum roundoff error?

Floating Point Precision

- Just as the precision of an integer is relative to the weight of the LSB (1), the precision of a floating point number is relative to the weight of the LSB
  - LSB weight is determined by the exponent
  - LSB weight is 2^{-23} times 2^{exp}
- precision ranges from -0.5 to 0.5 LSB for rounding, 0 to 1 LSB for truncation

Double Precision Floating Point

- If computers can accumulate errors so quickly, what to do?

  Larger representation, 2 words = 64 bits
- IEEE double precision format:

- exponent is biased by 1023, mantissa has a hidden 1.

Cumulative Double Precision Error

- Even the fastest computers will take a while to accumulate enough error to run out of precision using double precision FP

  (10^9 operations/sec) x (10^5 seconds/day) x ($10^{1.5}$ days/month) = $10^{15.5}$ operations/month = 2^{52} operations/month

- => a couple months before the max error magnitude approaches magnitude of mantissa

- Supercomputers have used quad precision floating point (128-bit) to avoid error accumulation for huge computations.

Double Precision Range

- Double precision allows much larger/smaller numbers due to larger exponent

  $2^{1023} \cong 10^{307}$

  - For comparison, there are an estimated 10^{70} atoms in the universe

- Increased precision is much more important than increased range.

Rounding Methods

- There are various methods for rounding, all of which are useful for certain situations
  - Truncation (rounding toward zero)
  - Rounding to nearest
  - Rounding toward +infinity:

    1.22 -> 1.3
    1.28 -> 1.3

    -2.81 -> -2.8
    -2.89 -> -2.8
  - Rounding toward -infinity:

    1.22 -> 1.2
    1.28 -> 1.2

    -2.81 -> -2.9
    -2.89 -> -2.9