UNIVERSITY OF MASSACHUSETTS
Dept. of Electrical & Computer Engineering

Digital Computer Arithmetic
ECE 666

Part 4-C
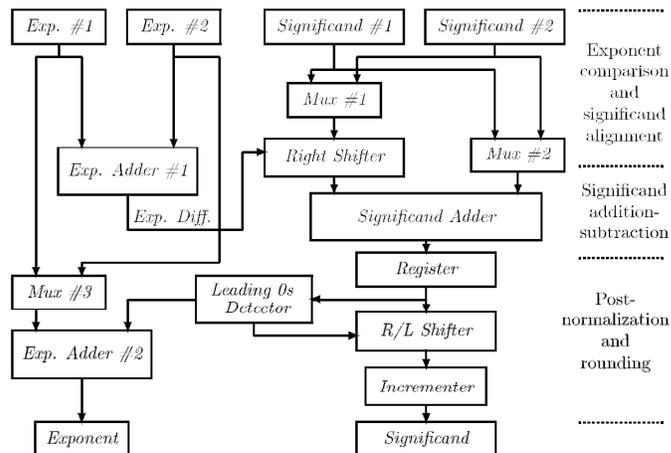### Floating-Point Arithmetic - III

Israel Koren

Floating-Point Adders

◆ **Addition - large number of steps executed sequentially - some can be executed in parallel**

Exponent comparison and significand alignment

Significand addition-subtraction

Post-normalization and rounding

Page 1

Effective Addition/Subtraction

♦ **Distinguish between effective addition and effective subtraction**
  * Depends on sign bits of operands and instruction executed

♦ **Effective addition:**
  * (1) Calculate exponent difference to determine alignment shift
  * (2) Shift significand of smaller operand, add aligned significands

♦ **The result can overflow by at most one bit position**
  * Long postnormalization shift not needed
  * Single bit overflow can be detected and, if found, a 1-bit normalization is performed using a multiplexor

Eliminate Increment in Rounding

♦ **Significand adder designed to produce two simultaneous results - sum and sum+1**
  * Called compound adder; can be implemented in various ways (e.g., carry-look-ahead or conditional sum)

♦ **Round-to-nearest-even - use rounding bits to determine which of the two should be selected**

♦ **These two are sufficient even if a single bit overflow occurs**
  * In case of overflow, 1 is added in R position (instead of LSB position), and since R=1 if rounding needed, a carry will propagate to LSB to generate correct sum+1

♦ **Directed roundings - R not necessarily 1 - sum+2 may be needed**

Effective Subtraction

◆ **Massive cancellation of most significant bits may occur - resulting in lengthy postnormalization**

◆ **Happens only when exponents of operands are close (difference ≤ 1) - pre-alignment can be eliminated**

◆ **Two separate procedures -**

  ∗ **(1) exponents are close (difference ≤ 1) – only a postnormalization shift may be needed**

  ∗ **(2) exponents are far (difference>1) - only a pre-alignment shift may be needed**

Step	CLOSE	FAR
1	Predict exponent	Subtract exponents
2	Subtract significands Predict number of leading zeroes	Align significands
3	Postnormalization	Subtract significands
4	Select properly rounded result or negate result	Select properly rounded result

---

CLOSE Case

◆ **Exponent difference predicted based on two least significant bits of operands - allows subtraction of significands to start as soon as possible**

  ∗ **If 0 - subtract executed with no alignment**

  ∗ **If ±1 - significand of smaller operand is shifted once to the right (using a multiplexor) and then subtracted from other significand**

◆ **In parallel - true exponent difference calculated**

  ∗ **If > 1 - procedure aborted and FAR procedure followed**

  ∗ **If ≤ 1 - CLOSE procedure continued**

◆ **In parallel with subtraction - number of leading zeros predicted to determine number of shift positions in postnormalization**

CLOSE Case - Normalization and Rounding

♦ **Next - normalization of significand and corresponding exponent adjustment**

♦ **Last - rounding - precomputing sum, sum+1 - selecting the one which is properly rounded - negation of result may be necessary**

♦ **Result of subtraction usually positive - negation not required**

♦ **Only when exponents equal - result of significand subtraction may be negative (in two's complement) - requiring a negation step**

♦ **No pre-alignment - no guard bits - no rounding (exact result)**

♦ **Negation and rounding steps - mutually exclusive**

FAR Case

♦ **First - exponent difference calculated**

♦ **Next - significand of smaller operand shifted to right for alignment**

♦ **Shifted-out bits used to set sticky bit**

♦ **Smaller significand subtracted from larger - result either normalized or requiring a single-bit-position left-shift (using a multiplexor)**

♦ **Last step - rounding**

Leading Zeros Prediction Circuit

♦ Predict position of leading non-zero bit in result of subtract before subtraction is completed

♦ Allowing to execute postnormalization shift immediately following subtraction

♦ Examine bits of operands (of subtract) in a serial fashion, starting with most significant bits to determine position of first 1

♦ This serial operation can be accelerated using a parallel scheme similar to carry-look-ahead

Alternative Prediction of Leading 1

♦ Generate in parallel intermediate bits e_i - $e_i=1$ if

* (1) $a_i = b_i$ and

* (2) a_{i-1} and b_{i-1} allow propagation of expected carry (at least one is 1)

* Subtract executed by forming one's complement of subtrahend and forcing carry into least significant position - carry expected

♦ $e_i = \overline{(a_i \oplus b_i)}\ (a_{i-1} + b_{i-1})$ -

$e_i=1$ if carry allowed to propagate to position i

* If forced carry propagates to position i - i-th bit of correct result will also be 1

* If not - correct result will have a 1 in position $i-1$ instead

* Position of leading 1 - either same as e_i or one to the right

♦ Count number of leading zeros in e_i - provide count to barrel shifter for postnormalization - at most one bit correction shift (left) needed

Exceptions in IEEE Standard

♦ **Five types** : overflow, underflow, division-by-zero, invalid operation, inexact result

♦ First three - found in almost all floating-point systems ; last two - peculiar to IEEE standard

♦ When an exception occurs - status flag set (remains set until cleared) - specified result generated

   ∗ **Example** - a correctly signed ∞ for division-by-zero

♦ Separate trap-enable bit for each exception

♦ If bit is on when corresponding exception occurs - user trap handler is called

♦ Sufficient information must be provided by floating-point unit to trap handler to allow taking action

   ∗ **Example** - exact identification of exception causing operation

---

Overflow - Trap Disabled

♦ **Overflow** exception flag set whenever exponent of result exceeds largest value allowed

♦ **Example** - single-precision - overflow occurs if E>254

♦ **Final result** determined by sign of intermediate (overflowed) result and rounding mode:

   ∗ Round-to-nearest-even - ∞ with sign of intermediate result

   ∗ Round toward 0 - largest representable number with sign of intermediate result

   ∗ Round toward -∞ - largest representable number with a plus sign if intermediate result positive; otherwise -∞

   ∗ Round toward ∞ - largest representable number with a minus sign if intermediate result negative; otherwise +∞

Overflow - Trap Enabled

◆ Trap handler receives intermediate result divided by 2^a and rounded

◆ $a = 192 / 1536$ for single / double-precision format

◆ Chosen in order to translate the overflowed result as nearly as possible to middle of exponent range so that it can be used in subsequent operations with less risk of causing further exceptions

---

Example

◆ Multiplying 2^{127} (with $E=254$ in single-precision) by 2^{127} - overflowed product has $E=254+254-127=381$ after being adjusted by 127

◆ Result overflows - $E>254$

◆ If product scaled (multiplied) by 2^{-192} - $E=381-192=189$ - "true" value of $189-127=62$

◆ Smaller risk of causing further exceptions

◆ Relatively small operands can result in overflow

◆ Multiply 2^{64} ($E=191$ in single-precision) by 2^{65} ($E=192$)

◆ Overflowed product - $E=191+192-127=256$

◆ Exponent adjusted by 192 - $E=256-192=64$ - "true" value of $64-127=-63$

Underflow - Trap Enabled

♦ **Underflow** exception flag is set whenever the result is a nonzero number between -2^{Emin} and 2^{Emin}

♦ $Emin=-126$ in single-precision format; 1022 in double-precision format

♦ Intermediate result delivered to underflow trap handler is the infinitely precise result multiplied by 2^a and rounded

♦ $a=192$ in single precision format; 1536 in double-precision format

---

Underflow - Trap Disabled

♦ Denormalized numbers allowed

♦ **Underflow** exception flag set only when an extraordinary loss of accuracy occurs while representing intermediate result (with a nonzero value between $\pm 2^{Emin}$) as a denormalized number

♦ Such a loss of accuracy occurs when either guard bit or sticky bit is nonzero- indicating an inexact result

♦ In an arithmetic unit where denormalized numbers are not implemented - delivered result is either $zero$ or $\pm 2^{Emin}$

Underflow - Trap Disabled - Example

♦ Denormalized numbers implemented

♦ Multiply 2^{-65} by 2^{-65}

  ∗ Result - E=(127-65)+(127-65)-127=-3 < 1

  ∗ Cannot be represent as a normalized number

  ∗ Result 2^{-130} represented as the denormalized number $0.0001 \cdot 2^{-126}$ - f=.0001 ; E=0

♦ No underflow exception flag is set

♦ If second operand is $(1+ulp)\, 2^{-65}$

  ∗ Correct product is $(1+ulp)\, 2^{-130}$

  ∗ Converted to a denormalized number - f=.0001 ; E=0

  ∗ Now sticky bit = 1

♦ Inexact result - underflow exception flag is set

Invalid Operation

♦ Flag is set if an operand is invalid for operation to be performed

♦ Result - when invalid operation trap is disabled - quiet NaN

♦ Examples of invalid operations :

  ∗ Multiplying 0 by ∞

  ∗ Dividing 0 by 0 or ∞ by ∞

  ∗ Adding +∞ and -∞

  ∗ Finding the square root of a negative operand

  ∗ Calculating the remainder x REM y where y=0 or x=∞

  ∗ Any operation on a signaling NaN

Division by Zero & Inexact Result

♦ **Divide-by-zero** exception flag is set whenever divisor is zero and dividend is a finite nonzero number

♦ When corresponding trap is disabled - result is a correctly signed ∞

♦ **Inexact Result** flag is set if rounded result is not exact or if it overflows without an overflow trap

♦ A rounded result is exact only when both guard bit and sticky bit are **zero** - no precision was lost when rounding

♦ Allows performing integer calculations in a floating-point unit

---

Accumulation of Round-off Errors

♦ Rounding in floating-point operations - even with best rounding schemes - results in errors that tend to accumulate as number of operations increases

* ∈ - relative round-off error in a floating-point operation

* ∗ - any floating-point arithmetic operation **+,-,×,÷**

* **Fl(x ∗ y)** - rounded or truncated result of **x ∗ y**

$$\epsilon = \frac{Fl(x \star y) - (x \star y)}{(x \star y)}$$

$$Fl(x \star y) = (x \star y) \cdot (1 + \epsilon)$$

Upper Bounds of Relative Errors

♦ **Truncation -**

♦ **Absolute error - maximum is least-significant digit of significand.**

♦ **Relative error - worst case when normalized result is smallest**

$$|\epsilon_{trunc}| \leq \frac{2^{-m}}{1/2} = 2^{-m+1}.$$

♦ **Round-to-nearest -**

♦ **Absolute error - maximum is half of ulp**

♦ **Relative error -**

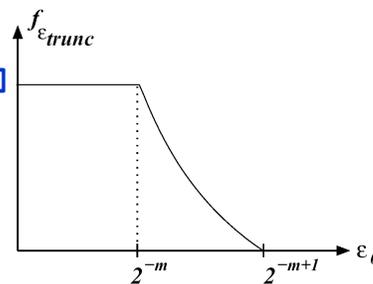$$|\epsilon_{round}| \leq \frac{1}{2}2^{-m+1} = 2^{-m}$$

Distribution of Relative <u>Truncation</u> Error

♦ **Density function of relative truncation error -**

$$f_{\epsilon_{trunc}}(\epsilon_0) = \begin{cases} 2^{m-1}/\ln 2 & \text{if} \quad 0 \leq \epsilon_0 < 2^{-m} \\ (\frac{1}{\epsilon_0} - 2^{m-1})/\ln 2 & \text{if} \quad 2^{-m} \leq \epsilon_0 < 2^{-m+1} \end{cases}$$

♦ **Relative truncation errors -**
  * uniformly distributed in **[0, 2^-m]**
  * reciprocally in **[2 ^-m , 2^-m+1]**

♦ **Average relative error -**

$$\overline{\epsilon_{trunc}} = 2^{-m-1}/\ln 2 \approx 0.72 \cdot 2^{-m}$$
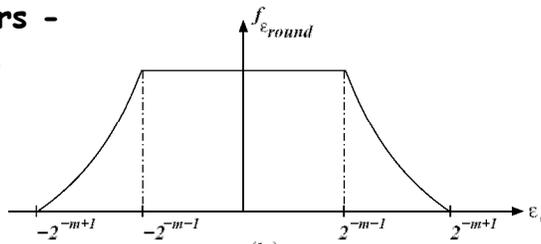
Page 11

Distribution of Relative <u>Rounding</u> Error

♦ **Density function** of relative rounding error -

$$f_{\epsilon_{round}}(\epsilon_0) = \begin{cases} 2^{m-1}/\ln 2 & \text{if} \quad -2^{-m-1} \le \epsilon_0 \le 2^{-m-1} \\ (\frac{1}{2|\epsilon_0|} - 2^{m-1})/\ln 2 & \text{if} \quad 2^{-m-1} < |\epsilon_0| < 2^{-m} \end{cases}$$

♦ **Relative rounding errors -**
  * uniformly distributed in **[-2^{-m-1}, 2^{-m-1}]**
  * reciprocally elsewhere
  * symmetric

♦ **Average relative error = 0**

♦ **Analytical expressions are in very good agreement with empirical results**

Accumulation of Errors - Addition

♦ **Adding two intermediate results A1, A2**
  * correct values - **A1^c, A2^c**
  * relative errors **∈1, ∈2**

$$A_1 = A_1^c(1 + \epsilon_1), \qquad A_2 = A_2^c(1 + \epsilon_2)$$

♦ **Assumption - no new error introduced in addition - relative error of sum**

$$\frac{A_1^c \cdot \epsilon_1 + A_2^c \cdot \epsilon_2}{A_1^c + A_2^c} = \frac{A_1^c}{A_1^c + A_2^c} \cdot \epsilon_1 + \frac{A_2^c}{A_1^c + A_2^c} \cdot \epsilon_2$$

♦ **Relative error of sum - weighted average of relative errors of operands**

♦ **If both operands positive - error in sum dominated by error in larger operand**

Accumulation of Errors - Subtraction

♦ **Subtracting two intermediate results A_1, A_2 - more severe error accumulation**

♦ **Relative error -**
$$\frac{A_1^c}{A_1^c - A_2^c} \cdot \epsilon_1 - \frac{A_2^c}{A_1^c - A_2^c} \cdot \epsilon_2$$

  ∗ **If A_1, A_2 are close positive numbers - accumulated relative error can increase significantly**

  ∗ **If ϵ_1, ϵ_2 have opposite signs - inaccuracy can be large**

♦ **Accumulation of errors for a long sequence of floating-point operations depends on the specific application - difficult to analyze - can be simulated**

♦ **In most cases - accumulated relative error in truncation is higher than in round-to-nearest**