MICROPROCESSOR
PRODUCTS GROUP

Efficiently producing default orthogonal IEEE double results using extended IEEE hardware

Roger Golliver

Floating-point Center of Expertise

roger.a.golliver@intel.com

Java’s requirements

- Java requires SPARC’s IEEE default behavior
  - no unmasked exceptions
  - no “Denormal” flag
  - no double rounding errors!
- Notes for code fragments following:
  - “_de” is 80-bit value on x87 stack or in memory
  - “_d” is double precision value in memory
  - “_s” is single precision value in memory
- Note: Java should support IEEE flags, and these algorithms do get the IEEE flags correct.

Java algorithm for add:

- with precision control set to 53-bits
- x_de = x_d -- exact (fld x_d)
- y_de = y_d -- exact (fld y_d)
- x_de = x_de + y_de -- will denormalize correctly if tiny (fadd)
- z_d = x_de -- will overflow correctly if huge (fstp z_d)

Java algorithm for sub:

- with precision control set to 53-bits
- x_de = x_d -- exact (fld x_d)
- y_de = y_d -- exact (fld y_d)
- x_de = x_de - y_de -- will denormalize correctly if tiny (fsubr)
- z_d = x_de -- will overflow correctly if huge (fstp z_d)

Java algorithm for multiply:

- with precision control set to 53-bits
- x_de = x_d -- exact (fld x_d)
- x_de *= 2.0^(Emax_d-Emax_de) -- exact scale down(fmul const1_de)
- y_de = y_d -- exact (fld y_d)
- x_de = x_de * y_de -- will denormalize correctly if tiny (fmul)
- x_de *= 2.0^(Emax_de-Emax_d) -- exact scale up (fmul const2_de)
- z_d = x_de -- will overflow correctly if huge (fstp z_d)

- Emax_de = 0x7FFE-0x3FFF(bias_de) = 0x3FFF
- Emax_d = 0x7FE- 0x3FF(bias_d) = 0x3FF
- Emax_de - Emax_d = 0x3FFF - 0x3FF = 0x3C00

Java algorithm for divide:

- with precision control set to 53-bits
- x_de = x_d -- exact (fld x_d)
- x_de *= 2.0^(Emax_d-Emax_de) -- exact scale down(fmul const1_de)
- y_de = y_d -- exact (fld y_d)
- x_de = x_de / y_de -- will denormalize correctly if tiny (fdivp)
- x_de *= 2.0^(Emax_de-Emax_d) -- exact scale up (fmul const2_de)
- z_d = x_de -- will overflow correctly if huge (fstp z_d)

Java algorithm for remainder (%):

- with precision control set to 53-bits
- x_de = x_d -- exact (fld x_d)
- y_de = y_d -- exact (fld y_d)
- loop:
- y_de = y_de % x_de -- exact (fprem)
- ax = flt-pt_status_word -- read status word (fstsw ax)
- if (ax&0x0400) goto loop -- remainder not completed
- z_d = y_de -- exact (fstp z_d)
- x_d = x_de -- exact/clean up stack (fstp x_d)

Java algorithm for remainder (IEEE):

• set precision control to 53-bits		
• x_de = x_d	-- exact	(fld x_d)
• y_de = y_d	-- exact	(fld y_d)
• loop:		
• y_de = y_de REM x_de	-- exact	(fprem1)
• ax = flt-pt_status_word	-- read status word	(fstsw ax)
• if (ax&0x0400) goto loop	-- remainder not completed	
• z_d = y_de	-- exact	(fstp z_d)
• x_d = x_de	-- exact/clean up stack	(fstp x_d)

Java algorithm for sqrt:

- set precision control to 53-bits
- x_de = x_d -- exact (fld x_d)
- x_de = sqrt(x_de) -- single rounding error (fsqrt)
- z_d = x_de -- result can't be tiny or huge (fstp z_d)

Java algorithm for narrowing conversion:

- set precision control to 53-bits
- x_de = x_d -- exact (fld x_d)
- y_s = x_de -- single rounding error (fstp y_s)

Details of the general algorithm

- follows the IEEE definition closely
  - easily understandable
  - confidence in correctness, if x87 rounds correctly it does
- uses the x87 with all exceptions masked
- overhead of integer ops can be partially hid by the latency of the floating ops
- same method works for all IEEE operations that round
  - add, subtract, multiply, divide, remainder, square root, and conversions
- algorithm can be easily optimized for constrained environments, e.g. the Java algorithms above

The General Algorithm (double precision):

- Initialize the control and the status words
  - PC is set to 53-bits
  - RC is set to emulating RC
  - MASKs are all set
  - FLAGs are all cleared
- Convert the double operand(s) to double-extended
  - x_de = x_d -- exact (fld qword ptr x_d)
  - y_de = y_d -- exact (fld qword ptr y_d)
  - Note: Denormal flag may be set erroneously after these operations

The General Algorithm (first rounding):

- Calculate the double extended result
  - z_de = x_de <fop> y_de -- round (fop)
  - Invalid, Divide-by-Zero, or Precision may be set by fop
  - This is equivalent to the IEEE's first rounding operation, i.e. rounding the infinitely precise result with the exponent unbounded.
- Select two constants c1 and c2,
  - using exponent of z_de classify result: (fstp tbyte ptr z_de)
    - Zero, Infinity/NaN, Normal -- no extra work required
    - Tiny or Huge -- extra work required
  - and the state the control bits for
    - Overflow -- default or wrapped result
    - Underflow -- default or wrapped result

The General Algorithm (second rounding):

- recalculate the result
  - if(add,sub,mul, or div) (fld tbyte ptr x_de)
    x_de *= c1 -- exact (fmul tbyte ptr c1)
  - if(add or sub) (fld tbyte ptr y_de)
    y_de *= c1; -- exact (fmul tbyte ptr c1)
  - z_de = x_de <fop> y_de -- round and clamp exponent (fop)
  - if(add,sub,mul, or div)
    z_de *= c2 -- exact (fmul tbyte ptr c2)
  - z_d = z_de -- exact (fstp qword ptr z_d)
- Overflow, Underflow, and/or Precision may be set by fop
- z_de is equivalent to the IEEE's second rounding operation
- z_d is the IEEE standard's result

The General Algorithm (cont.)

- read the flags, and adjust if necessary
  - if Huge and Overflow is unmasked, set Overflow
  - if Tiny and Underflow is unmasked, set Underflow
  - if d_x or d_y is a NaN then clear Denormal
- report exceptional conditions
  - if d_x and d_y are NaNs then special NaN propagation needed
  - if flag is set for an unmasked exception, indicate “Exception”

How to choose the constants c1 and c2:

- exponent all ones
  or exponent all zero's
  - c1 = 1.0 c2 = 1.0
- Emin_d <= exponent
  or exponent <=Emax_d
  - c1 = 1.0 c2 = 1.0
- exponent > Emax_d
  and Overflow masked
  - c1 = 2.0^(Emax_de - Emax_d)
  - c2 = 0.5^(Emax_de - Emax_d)
- exponent > Emax_d
  and Overflow unmasked
  - c1 = 1.0
  - c2 = 0.5^((Emax_d+1)*3/2)
- exponent < Emin_d
  and Underflow masked
  - c1 = 0.5^(Emax_de - Emax_d)
  - c2 = 2.0^(Emax_de - Emax_d)
- exponent < Emin_d
  and Underflow Unmasked
  - c1 = 1.0
  - c2 = 2.0^((Emax_d+1)*3/2)