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 \((\text{fld } x_d)\)
  • \( x_{de} * 2.0^{(E_{max_d} - E_{max_{de}})} \) -- exact scale down\((\text{fmul } \text{const1}_{de})\)
  • \( y_{de} = y_d \) -- exact \((\text{fld } y_d)\)
  • \( x_{de} = x_{de} * y_{de} \) -- will denormalize correctly if tiny \((\text{fmul})\)
  • \( x_{de} *= 2.0^{(E_{max_{de}} - E_{max_d})} \) -- exact scale up \((\text{fmul } \text{const2}_{de})\)
  • \( z_d = x_{de} \) -- will overflow correctly if huge \((\text{fstp } z_d)\)

• \( E_{max_{de}} = 0x7FFE - 0x3FFF(bias_{de}) = 0x3FFF \)
• \( E_{max_d} = 0x7FE - 0x3FF(bias_d) = 0x3FF \)
• \( E_{max_{de}} - E_{max_d} = 0x3FFF - 0x3FF = 0x3C00 \)
Java algorithm for divide:

- with precision control set to 53-bits
- \( x_{de} = x_d \) -- exact \( \text{fld } x_d \)
- \( x_{de} *= 2.0^{(E_{max_d}-E_{max_de})} \) -- exact scale down\( \text{fmul } \text{const1}_{de} \)
- \( y_{de} = y_d \) -- exact \( \text{fld } y_d \)
- \( x_{de} = x_{de} / y_{de} \) -- will denormalize correctly if tiny \( \text{fdivp} \)
- \( x_{de} *= 2.0^{(E_{max_{de}}-E_{max_d})} \) -- exact scale up \( \text{fmul } \text{const2}_{de} \)
- \( z_d = x_{de} \) -- will overflow correctly if huge \( \text{fstp } z_d \)
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:
  - \( y_{de} = y_{de} \mod x_{de} \) -- exact (fprem)
  - \( ax = \text{flt-pt\_status\_word} \) -- 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:
  • \( y_{de} = y_{de} \ \text{REM} \ x_{de} \) -- exact \( (fprem1) \)
  • \( ax = \text{flt-pt}_{-}\text{status}_{-}\text{word} \) -- 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 $x\_d$)
- $x\_de = \text{sqrt}(x\_de)$ -- 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
- \[ x_{\text{de}} = x_{\text{d}} \] -- exact \hspace{1cm} (fld \ x_{\text{d}})
- \[ y_{\text{s}} = x_{\text{de}} \] -- single rounding error \hspace{1cm} (fstp \ y_{\text{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
  - $x_{\text{de}} = x_{\text{d}}$ -- exact \hspace{1cm} (fld qword ptr x_d)
  - $y_{\text{de}} = y_{\text{d}}$ -- exact \hspace{1cm} (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
  - $z_{de} = x_{de} <\text{fop}> y_{de}$ -- round  
  - 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:  
    - 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) (fmul tbyte ptr c2)
    
    z_de *= c2              -- exact

  – 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 $c_1$ and $c_2$:

- exponent all ones
  or exponent all zero’s
  - $c_1 = 1.0$    $c_2 = 1.0$
- $E_{min_d} \leq$ exponent
  or exponent $\leq E_{max_d}$
  - $c_1 = 1.0$    $c_2 = 1.0$
- exponent $> E_{max_d}$
  and Overflow masked
  - $c_1 = 2.0^{(E_{max_d} - E_{max_d})}$
  - $c_2 = 0.5^{(E_{max_d} - E_{max_d})}$
- exponent $> E_{max_d}$
  and Overflow unmasked
  - $c_1 = 1.0$
  - $c_2 = 2.0^{((E_{max_d}+1)*3/2)}$

- exponent $< E_{min_d}$
  and Underflow masked
  - $c_1 = 0.5^{(E_{max_d} - E_{max_d})}$
  - $c_2 = 2.0^{(E_{max_d} - E_{max_d})}$
- exponent $< E_{min_d}$
  and Underflow Unmasked
  - $c_1 = 1.0$
  - $c_2 = 2.0^{((E_{max_d}+1)*3/2)}$