Some of Java’s Problematic Floating-point Requirements

• Java requires orthogonal IEEE default behavior
  – No unmasked exceptions
  – No IEEE flags, (but this should and could be easily corrected)
  – Bit for bit exact results, i.e. no double rounding errors from extended register’s extra range and/or precision

• First, this ignores the installed base of PC using extended IEEE floating-point.

• Second, I know of no conforming multiply/divide implementations on extended architectures.
Sun is currently trying to address this Floating-point issue

• Sun is using its “open” PAS process
• Draft proposal published on the Internet
• Public comments were request by Sept 15, 1998
• One organized public response was from the JavaGrande group
• Sun’s response to the public comments has not been made public yet.
The proposal will likely contain two floating-point modes

- The default mode will be tolerant of different IEEE/ANSI Std. 754-1985 conforming hardware implementations

- The second mode will a “strict” orthogonal mode
  - a new keyword will be introduced “strict”
  - it will conform to Java’s current bit-for-bit exact FP results
Sun’s draft proposal’s algorithm to get strict multiply results on extended architectures

```
fld qword ptr [dx]  /* dz = dx * dy */
fclext               /* clear flags */
fmul qword ptr [dy]  /* 53-bits of sign., 15-bits of exp. */
fstsw word ptr [sw]  /* rounded-up in C1 and sticky in Precision(Inexact) */

fst qword ptr [dtmp] /* 53-bits of significand, */
fstsw ax              /* and 11-bits of exponent */
and ax,0x30           /* Precision/Inexact AND Underflow */
xor ax,0x30           /* set after fmul and store? */
jne skip              /* if not then okay, continue */
jsr fix_up            /* fix-up will use [sw] and top of x87 to round and clamp as required by STD Java */
    skip:
fstp qword ptr [dz]
```
New algorithm to perform a strict multiply on an extended IEEE architectures

precision control set to 53-bits

\[ x_{de} = x_d \]
\[ x_{de} *= 2.0^{(E_{max_d}-E_{max_{de}})} \]
\[ y_{de} = y_d \]
\[ x_{de} = x_{de} * y_{de} \]

\[ x_{de} *= 2.0^{(E_{max_{de}}-E_{max_d})} \]
\[ z_d = x_{de} \]

\[ E_{max_{de}}=0x7FFE-0x3FFF=0x3FFF \]
\[ E_{max_d} =0x7FE-0x3FF=0x3FF \]

Assume IA-32™ style architecture
exact, promotion
exact will scale down
exact, promotion
will underflow correctly (denormalize) if tiny
exact will scale up
will overflow correctly if huge

\[ E_{max_{de}}-E_{max_d} = 0x3C00 \]
\[ E_{max_d}-E_{max_{de}} =-0x3C00 \]
• Advantages of the new algorithm

• No expensive serializing operations on the control and status words.

• It allows for optimizations which hide the latency of floating-point operations.

• Its cost is only two more multiples from what current JVM’s are probably doing.
Conclusion and ways to follow up

• Questions on the new algorithm can be directed to me at roger.a.golliver@intel.com.

• JavaGrande is group interested in other issues related to using Java for Scientific and Engineering applications.
  – you can visit them at: http://www.javagrande.org
  – JavaGrande will have a “Birds-of-a-Feather” session at the SC98 (Supercomputing98) in November.