ARITHMETIC FOR COMPUTERS

Jo, Heeseung

Arithmetic for Computers

Operations on integers

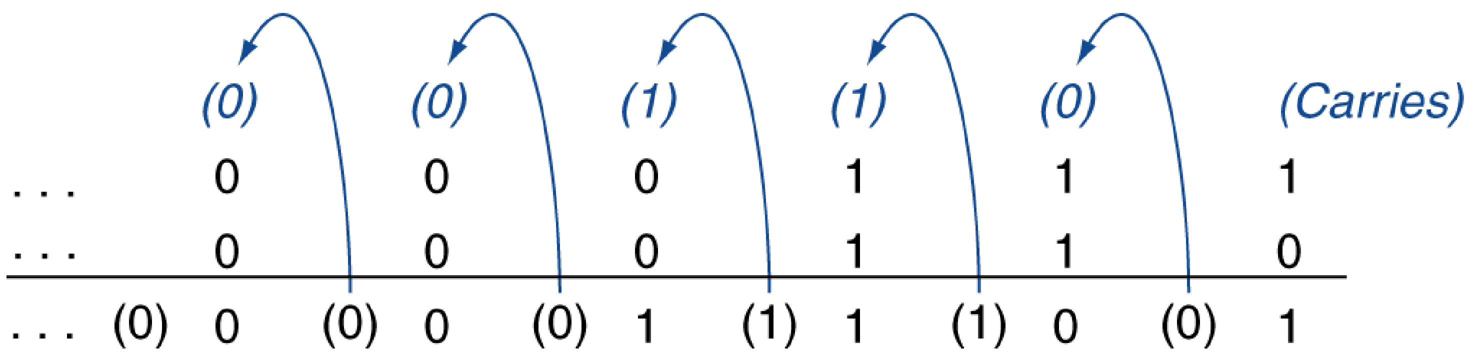
- Addition and subtraction
- Multiplication and division
- Dealing with overflow

Floating-point real numbers

- Representation and operations

Integer Addition

Example: 7 + 6



Overflow if result out of range

- Adding +ve and –ve operands, no overflow
- Adding two +ve operands
  - Overflow if result sign is 1
- Adding two –ve operands
  - Overflow if result sign is 0

Integer Subtraction

Add negation of second operand

Example: 7 - 6 = 7 + (-6)

```
+7: 0000 0000 … 0000 0111
-6: 1111 1111 … 1111 1010
+1: 0000 0000 … 0000 0001
```

Overflow if result out of range

- Subtracting two +ve or two -ve operands, no overflow
- Subtracting +ve from -ve operand
  - Overflow if result sign is 0
- Subtracting -ve from +ve operand
  - Overflow if result sign is 1

Dealing with Overflow

Some languages (e.g., C) ignore overflow

- Use MIPS addu, addui, subu instructions

Other languages (e.g., Ada, Fortran) require raising an exception

- Use MIPS add, addi, sub instructions
- On overflow, invoke exception handler
  - Save PC in exception program counter (EPC) register
  - Jump to predefined handler address
  - mfc0 (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

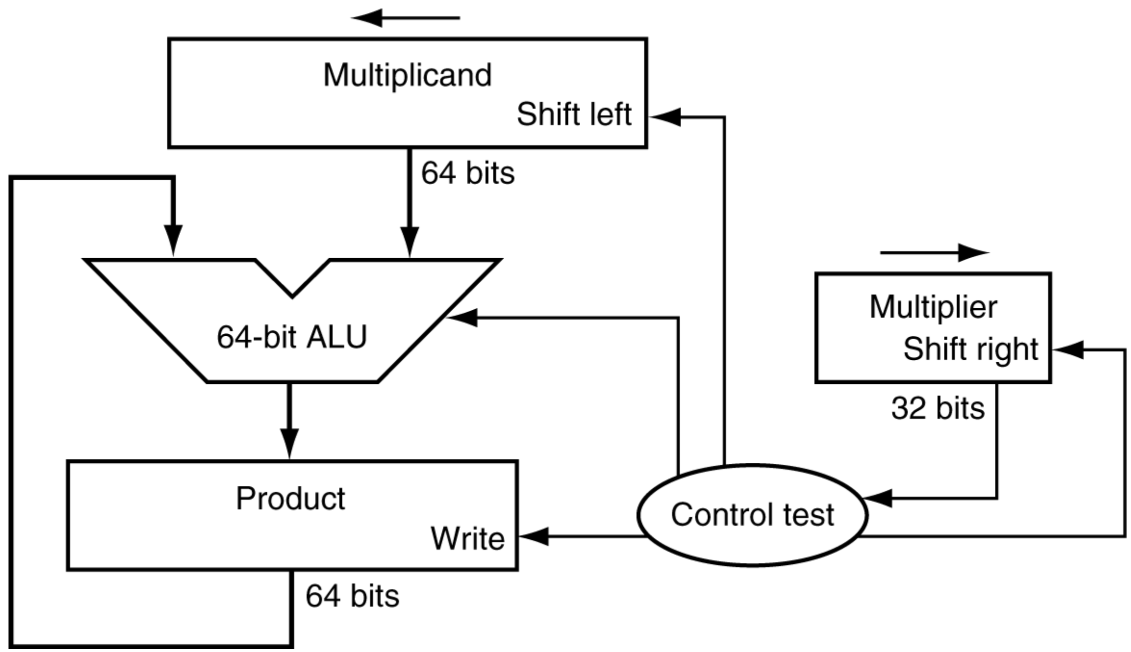
Multiplication

Start with long-multiplication approach

multiplicand → 1000
multiplier → × 1001

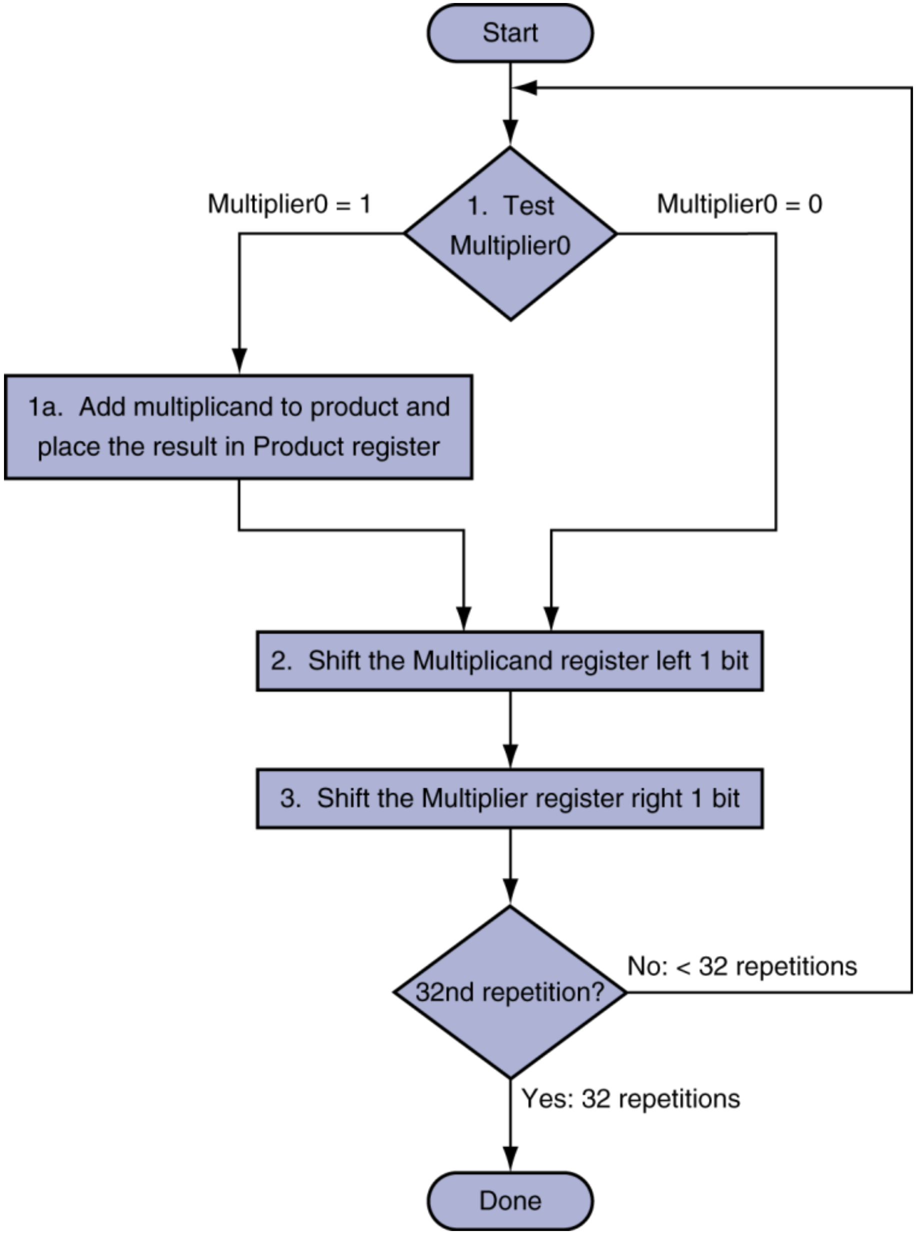
```
   1000
 × 1001
-------
   1000
  0000
 0000
1000
-------
1001000
```

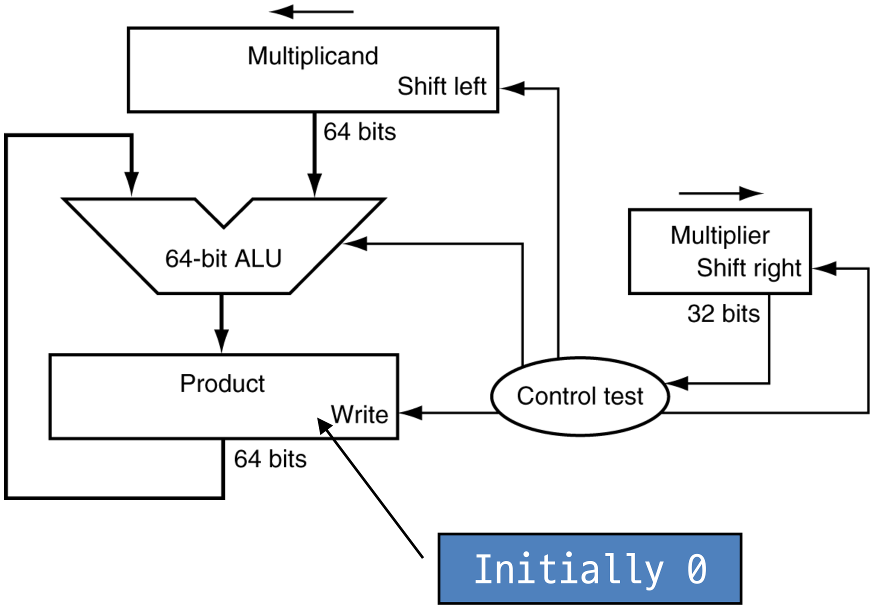
product → 1001000



Length of product is the sum of operand lengths

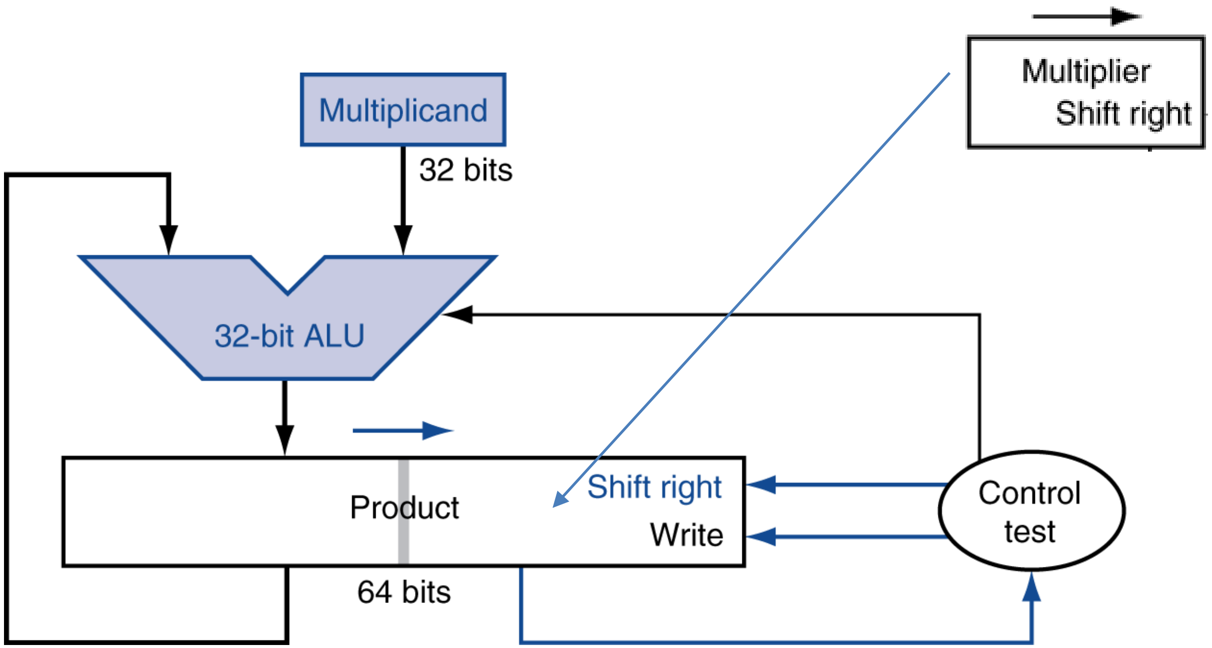
Multiplication Hardware



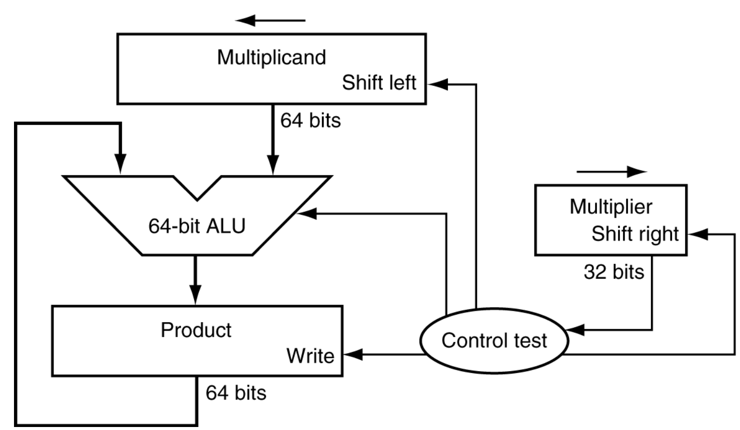


Optimized Multiplier

Perform steps in parallel: add/shift



One cycle per partial-product addition



Division

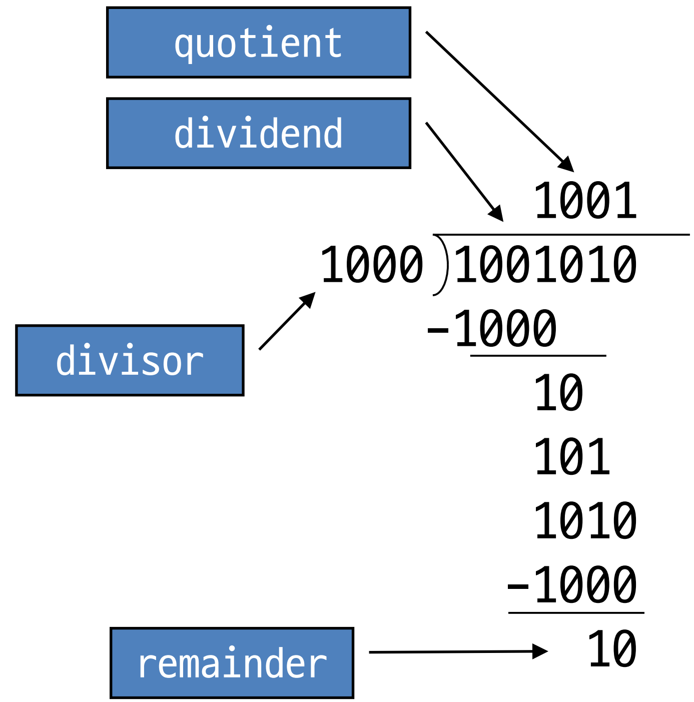
Check for 0 divisor

Long division approach

- If divisor ≤ dividend bits
  - 1 bit in quotient, subtract
- Otherwise
  - 0 bit in quotient, bring down next dividend bit

Signed division

- Divide using absolute values
- Adjust sign of quotient and remainder as required



n-bit operands yield *n*-bit quotient and remainder

Floating Point

Representation for non-integral numbers

- Including very small and very large numbers

Scientific notation

- -2.34×10^{56} ← normalized
- $+0.002 \times 10^{-4}$ ← not normalized
- $+987.02 \times 10^{9}$ ← not normalized

In binary

- $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

Types float and double in C

Floating Point Standard

Defined by IEEE Std 754-1985

Developed in response to divergence of representations

- Portability issues for scientific code

Now almost universally adopted

Two representations

- Single precision (32-bit)
- Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)

Normalize significand: 1.0 ≤ |significand| < 2.0

- Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
- Significand is Fraction with the "1." restored

Exponent: excess representation: actual exponent + Bias

- Ensures exponent is unsigned
- Single: Bias = 127; Double: Bias = 1203

Floating-Point Example

Represent -0.75

- $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- S = 1
- Fraction = $1000\ldots00_2$
- Exponent = -1 + Bias
  - Single: -1 + 127 = 126 = 01111110_2
  - Double: -1 + 1023 = 1022 = 01111111110_2

S	Exponent	Fraction

Single: 1011111101000…00

Double: 101111111101000…00

Floating-Point Example

What number is represented by the single-precision float

11000000101000…00

S	Exponent	Fraction

- S = 1
- Fraction = $01000\ldots00_2$
- Exponent = 10000001_2 = 129

$$x = (-1)^1 \times (1 + 01_2) \times 2^{(129\ -\ 127)}$$

$$= (-1) \times 1.25 \times 2^2$$

$$= -5.0$$

Single-Precision Range

Exponents 00000000 and 11111111 reserved

S	Exponent	Fraction

Smallest value

- Exponent: 00000001
  ⇒ actual exponent = 1 – 127 = –126
- Fraction: 000…00 ⇒ significand = 1.0
- $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

Largest value

- exponent: 11111110
  ⇒ actual exponent = 254 – 127 = +127
- Fraction: 111…11 ⇒ significand ≈ 2.0
- $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

Exponents 0000…00 and 1111…11 reserved

Smallest value

S	Exponent	Fraction

- Exponent: 00000000001
  $\Rightarrow$ actual exponent = 1 – 1023 = –1022
- Fraction: 000…00 $\Rightarrow$ significand = 1.0
- $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$

Largest value

- Exponent: 11111111110
  $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
- Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
- $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

Relative precision

- all fraction bits are significant
- Single: approx 2^{-23}
  - Equivalent to $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- Double: approx 2^{-52}
  - Equivalent to $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Denormal Numbers

Condition: Exponent = 000…0

Cases

- **exponent = 000…0, frac = 000…0**
  - Represents value 0
  - Note that have distinct values +0 and −0

$$\boxed{x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0}$$

Two representations of 0.0!

- **exponent = 000…0, frac ≠ 000…0**
  - Numbers very close to 0.0

Infinities and NaNs

Exponent = 111...1, Fraction = 000...0

- ±Infinity
- Operation that overflows

Exponent = 111...1, Fraction ≠ 000...0

- Not-a-Number (NaN)
- Indicates illegal or undefined result
  - e.g., 0.0 / 0.0

FP Adder Hardware

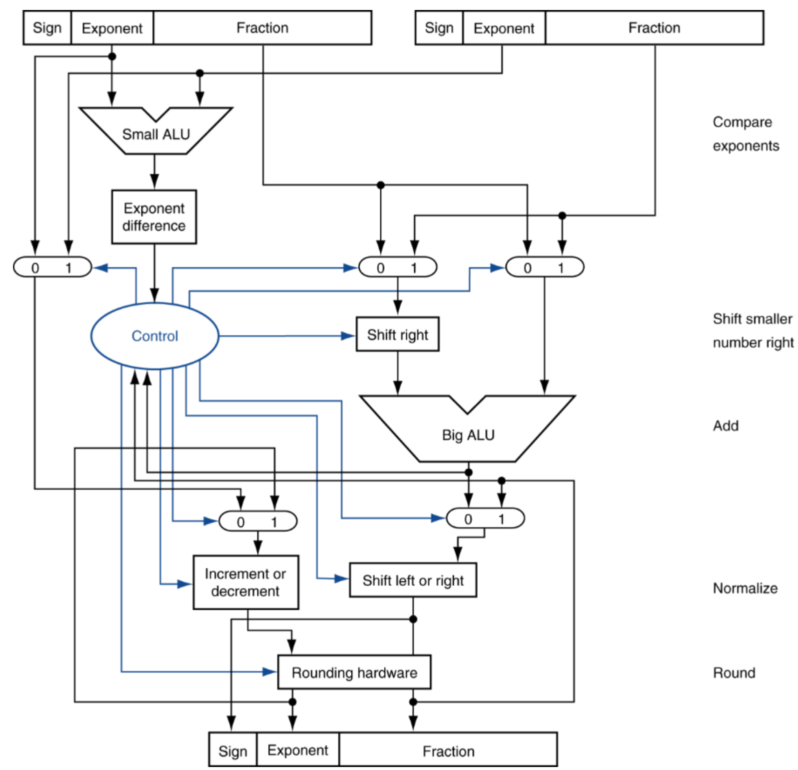
Much more complex than integer adder

Doing it in one clock cycle would take too long

- Much longer than integer operations
- Slower clock would penalize all instructions

FP adder usually takes several cycles

- Can be pipelined



FP Arithmetic Hardware

FP multiplier is of similar complexity to FP adder

FP arithmetic hardware usually does

- Addition, subtraction, multiplication, division, reciprocal, square-root
- FP ↔ integer conversion

Operations usually takes several cycles

- Can be pipelined

Interpretation of Data

Bits have no inherent meaning

- Interpretation depends on the instructions applied

Computer representations of numbers

- Finite range and precision
- Need to account for this in programs

Right Shift and Division

Left shift by i places multiplies an integer by 2^i

Right shift divides by 2^i?

- Only for unsigned integers

For signed integers

- Arithmetic right shift: replicate the sign bit
- e.g., -5 / 4
  - 11111011_2 >> 2 = 11111110_2 = -2
  - Rounds toward -infinity (We want to round to 0)
- Logical right shift: fill 0
- c.f. 11111011_2 >>> 2 = 00111110_2 = +62 (in Java)

Who Cares About FP Accuracy?

Important for scientific code

- But for everyday consumer use?
  - "My bank balance is out by 0.0002¢!" ☹

The Intel Pentium FDIV bug

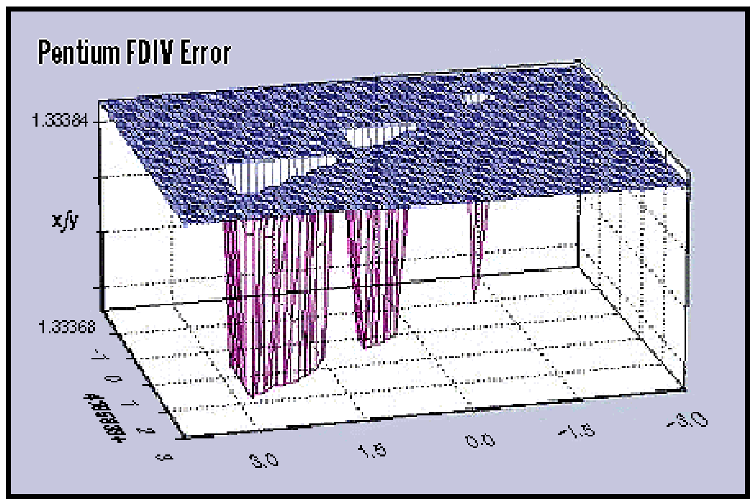
- The market expects accuracy
- See Colwell, *The Pentium Chronicles*

FIGURE 3.23 A sampling of newspaper and magazine articles from November 1994, including the *New York Times, San Jose Mercury News, San Francisco Chronicle,* **and** *Infoworld.* The Pentium floating-point divide bug even made the "Top 10 List" of the *David Letterman Late Show* on television. Intel eventually took a $300 million write-off to replace the buggy chips.

Floating Point Disasters

Intel Ships and Denies Bugs

- In 1994, Intel shipped its first Pentium processors with a floating-point divide bug
- The bug was due to bad look-up tables used in to speed up quotient calculations
- After months of denials, Intel adopted a no-questions replacement policy, costing $300M.
- (http://www.intel.com/support/processors/pentium/fdiv/)

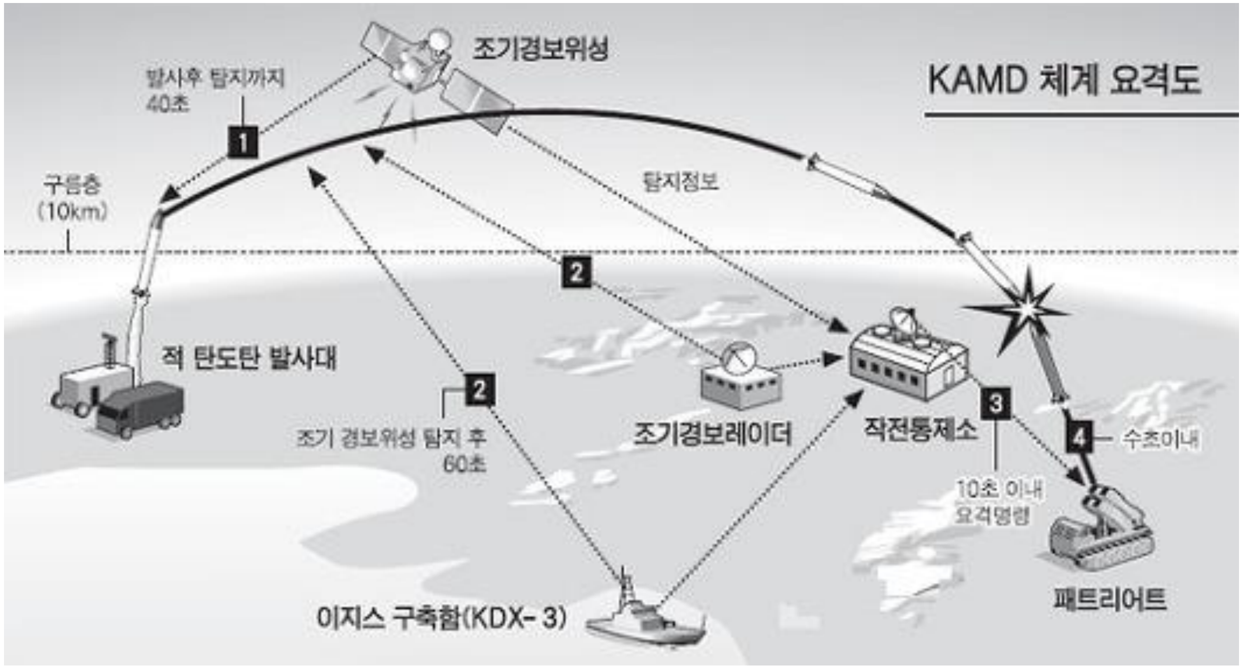


A 3-D plot of the ratio 4195835/3145727 calculated on a Pentium with FDIV bug. The depressed triangular areas indicate where incorrect values have been computed. The correct values all would round to 1.3338, but the returned values are 1.3337, an error in the fifth significant digit. [Byte Magazine](), March 1995.

Floating Point Disasters

Scud Missiles get through, 28 die

- In 1991, during the 1st Gulf War, a Patriot missile defense system let a Scud get through, hit a barracks, and kill 28 people
- The problem was due to a floating-point error when taking the difference of a converted & scaled integer
- (Source: Robert Skeel, "Round-off error cripples Patriot Missile", SIAM News, July 1992.)



Floating Point Disasters

$7B Rocket crashes (Ariane 5)

- When the first ESA Ariane 5 was launched on June 4, 1996, it lasted only 39 seconds, then the rocket veered off course and self-destructed
- An inertial system, produced a floating-point exception while trying to convert a 64-bit floating-point number to an integer
- Ironically, the same code was used in the Ariane 4, but the larger values were never generated
- (http://www.around.com/ariane.html).

Concluding Remarks

ISAs support arithmetic

- Signed and unsigned integers
- Floating-point approximation to reals

Bounded range and precision

- Operations can overflow and underflow