18-447 Lecture 4: Floating Point

James C. Hoe
Dept of ECE, CMU
January 28, 2009

Announcements: Read P&H Ch 2 (ISA) for Wednesday
Read ARCHITECTURE OF THE IBM SYSTEM/360, Amdahl, Blaauw, Brooks (on Blackboard)
Lab 1 due this week, both partners need to be present for check off

Handouts:

Limitations of Binary Numbers

- N-bit binary numbers
  - range: $0 \sim 2^n-1$
  - accuracy: 1 i.e., smallest resolvable difference
  - precision: good for large values, bad for small values

    precision: total number of digits to express a value

    range: the difference between the largest and the smallest representable values

    dynamic range: ratio of largest and smallest representable (non-zero) values
- Numerical applications often need
  - larger range and/or larger dynamic range
  - maintain precision, independent of magnitude
  - fractions and real numbers

How do you satisfy some or all with the same N bits?

Fixed-Point Representation

- Let $b_{n-1}b_{n-2}...b_2b_1b_0$ represent an n-bit unsigned fixed-point number
  - must also specify the weight of b_0

    (suppose it's 2^x, x could be negative)
  - its value is

$$2^x \cdot \sum_{i=0}^{n-1} 2^i b_i$$

why not $2^x \cdot \sum_{i=0}^{n-1} 4^i b_i$

- a finite representation with range between 0 and $2^{n+x}-2^x$ and with an accuracy of 2^x
  - if 0>x>-n+1 then the representation includes a whole-number portion and a fractional portion, i.e., $b_{n-1}b_{n-2}...b_{-x}.b_{-x-1}...b_2b_1b_0$
  - 2's complement fixed-point representation can be derived similarly
  - arithmetic between identically formatted numbers is unchanged except for multiplication which requires re-scaling to position the binary point (just like in long-hand)

Choosing a Fixed Point Format

- Num. of bits determines dynamic range and precision
- Weight of b_0 determines accuracy
- Determine the largest magnitude (say ~2^Y) and the smallest non-zero magnitude (say ~2^X) you want to represent accurately during a computation
  - ratio $2^Y/2^X$ = dynamic range
  - $(Y-X+1)$-bit fixed-point with b_0 weighing 2^X

    notice: the largest magnitude by itself is unimportant
- Precision and dynamic range matters the most if
  - a small value is multiplied by a large value, e.g. 3.14x10000
  - comparable large values are subtracted, e.g. $1.0\text{x}10^4$-9999
  - a small value is used to divide a large value, e.g. 10000/3.14

Shortcomings of Fixed Point

- Example: let $A=2^{63}$, $B=2^{32}$ and $C=2^{32}-1$

  a program with {A, B} requires a 32-bit format for dynamic range

  a program with {B, C} requires a 33-bit format for accuracy

  a program with {A, B, C} requires a 64-bit format even if A and C are never operated together

  Dynamic range and accuracy are inflexibly coupled

- Cannot represent large and small values to the same precision
  - e.g. 8'b11111111 / 2 vs. 8'b00000011 / 2

- Compactness of dynamic range encoding
  - for N-bit unsigned, dynamic range $\approx 2^N$

  Compact?? 32-bit$\Rightarrow 4\times10^9$ 64-bit$\Rightarrow 16\times10^{18}$

Runtime Re-Scaling

- Range, dynamic range and accuracy requirements of a computation are not static throughout
- One could design fixed-point algorithms where the weight of b_0 are scaled as necessary
- A very simple example,
  - suppose we are summing K same-format N-bit fixed-point numbers in a binary reduction tree
  - final sum could require $(\log_2 K+N)$ bits to not overflow
  - Alternatively, suppose N bits of precision is sufficient, we can retain a N-bit format throughout and only adjust the "binary point" 1 position to the right after each level of reduction
- Applicable in many cases but impossible without deep knowledge of both the algorithm and the expected input values

Floating Point Format

- Scientific notation (e.g., 6.022×10^{23}) in binary

$$(-1)^{s} \times (\text{significand}_{N}) \times 2^{(\text{exponent}_{M})}$$

s	exponent	significand

note** sign-magnitude based scheme

- A number of variants have been implemented
  - encoding size ≈ 1 + N + M
  - bit allocation for exponent vs. significand

    range = 0 ~ $2^{2^{M-1}}$

    dynamic range ≈ $2^{2^{M}}$

    N significant (binary) digits (precise to 1 part in 2^{N})

    Note "accuracy" changes with magnitude but precision is fixed
  - special encodings

IEEE 754-1985

- Virtually universally adopted (especially on anything mass marketed)
- Standard specifies: Only 20 pages!?
  - number representations
  - precisely defined operations and behaviors
  - exceptional conditions and trapping behaviors

    seemingly every detail is deliberately chosen
- Goals
  - portability of numerical code
  - maximize numerical stability and accuracy

    for us mortals, use "double" and forget about it
  - get it right once and for all
  - doesn't always make it easy for HW implementation

Formats

- 4 formats: single, extended-single, double, extend double

	single	single extended	double	double extended
encoding	32-bit	≥ 43-bit	64-bit	≥ 79bit
significand	24-bit	≥ 32-bit	53-bit	≥ 64
exponent	8-bit -126~127	≥ 11-bit	11-bit -1022~1023	≥ 15-bit

Note** in normal form (i.e., 1.xxxxx), leading bit is always 1 and hence 1 bit of the significand is implicit

- single precision
  - dynamic range 10^{-38}~10^{38}
  - precision ≈ 1 in 10^{7}
- double precision
  - dynamic range ≈10^{-307}~10^{307}
  - precision ≈ 1 in 10^{16}

Values

- In normal form, value = $(-1)^{s} \times 1.f \times 2^{e\text{-bias}}$
  - implicit leading 1 (how do you represent 0?)
  - biased exponent format (single::127, double::1023)
    enable positive values to be compared (>,<) as integers
  - note max and min values of **e** fall outside of allowable range!!
- if **e**==0 then
  - if **f**==0 then value = ± 0 (depend on sign)
  - else value = $(-1)^{s} \times 0.f \times 2^{1\text{-bias}}$ (denormal values)
- if **e**==111…111 then
  - if **f**==0 then value = $\pm \infty$ (infinity)
  - else NaN (Not a Number)

Denormal Number

- The smallest "normal" number is $1.0 \times 2^{1\text{-bias}}$
- Denormal allows a smooth approach toward 0 as precision gradually underflows

  $1.0 \times 2^{1\text{-bias}} / 2 = 0.1 \times 2^{1\text{-bias}}$
- Alternative is to flush denormal directly to 0
  - The difference is minute ($\sim 10^{-307}$ for double)
  - Why does it matter?
- suppose $x = 1.1 \times 2^{1\text{-bias}}$ and $y = 1.0 \times 2^{1\text{-bias}}$
  - $x \neq y$ (clearly distinguishable using integer comparison)
  - with flush-to-0, $x - y = 0$ (contradicts $x \neq y$)
  - with denormal, $x - y$ behaves more predictably near the edge of the representable space

Special Values and Rules

- NaN
  - generated by invalid operations, such SQRT(-1)
  - computation continues even after NaN is encountered
  - rule: any operation on NaN operands in turn outputs NaN

  $\Rightarrow$ enable more streamlined coding where exceptions are checked once at the end
- $\pm \infty$
  - generated by overflow or 1/0
  - certain operations can continue on $\pm \infty$ values

    i.e., $1/\infty \Rightarrow 0$, $1+\infty \Rightarrow \infty$ but $\infty-\infty \Rightarrow$ NaN, $\infty/\infty \Rightarrow$ NaN
- negative 0??
  - $1/-\infty \Rightarrow -0$, a negative value underflows to -0
  - if x==y & x!=0 then x - y $\Rightarrow$ +0
  - but (+0) + (+0) $\Rightarrow$ +0, (-0)+(-0) $\Rightarrow$ -0

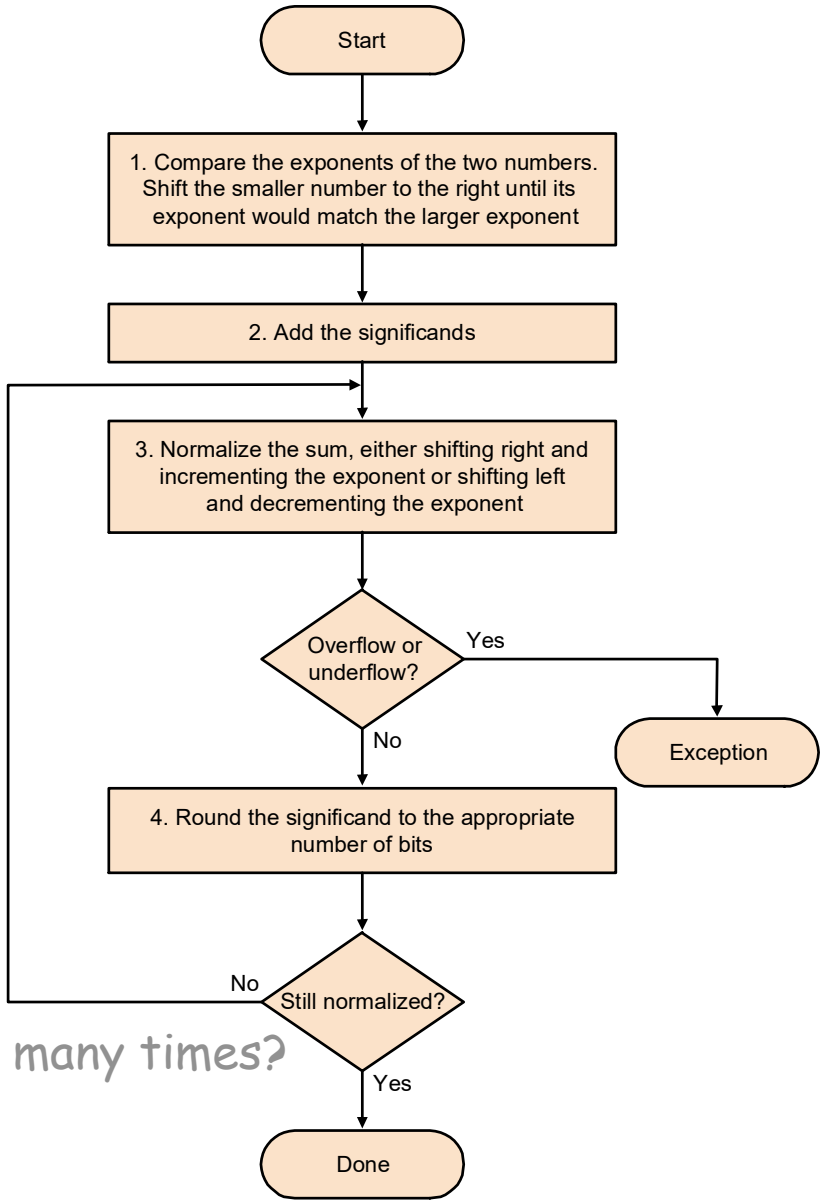
FP Exceptions

- Types
  - Invalid: when an operation receives unacceptable operands
  - Divide by Zero: 1 / 0
  - Overflow and Underflow : exponent too large or too small after operation
  - Inexact: when results are rounded

    Very subtle details in specification
- Effects
  - Exceptions should not stop computation by default!!

    output NAN and keep going
  - Exceptions set flags that must be explicitly cleared by user
  - Trapping is an implementation choice
  - Certain information must be preserved for the trap handler if invoked ⇒ precise interrupt

FP Add (think scientifc notation)



how many times?

[Figure 3.16 from P&H, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

$$\begin{array}{lll} & 9.999 & \times 10^{1} \\ + & 1.610 & \times 10^{-1} \end{array}$$

$$\begin{array}{lll} & 9.999 & \times 10^{1} \\ + & 0.01610 & \times 10^{1} \\ & 10.01510 & \times 10^{1} \end{array}$$

step 1

step 2

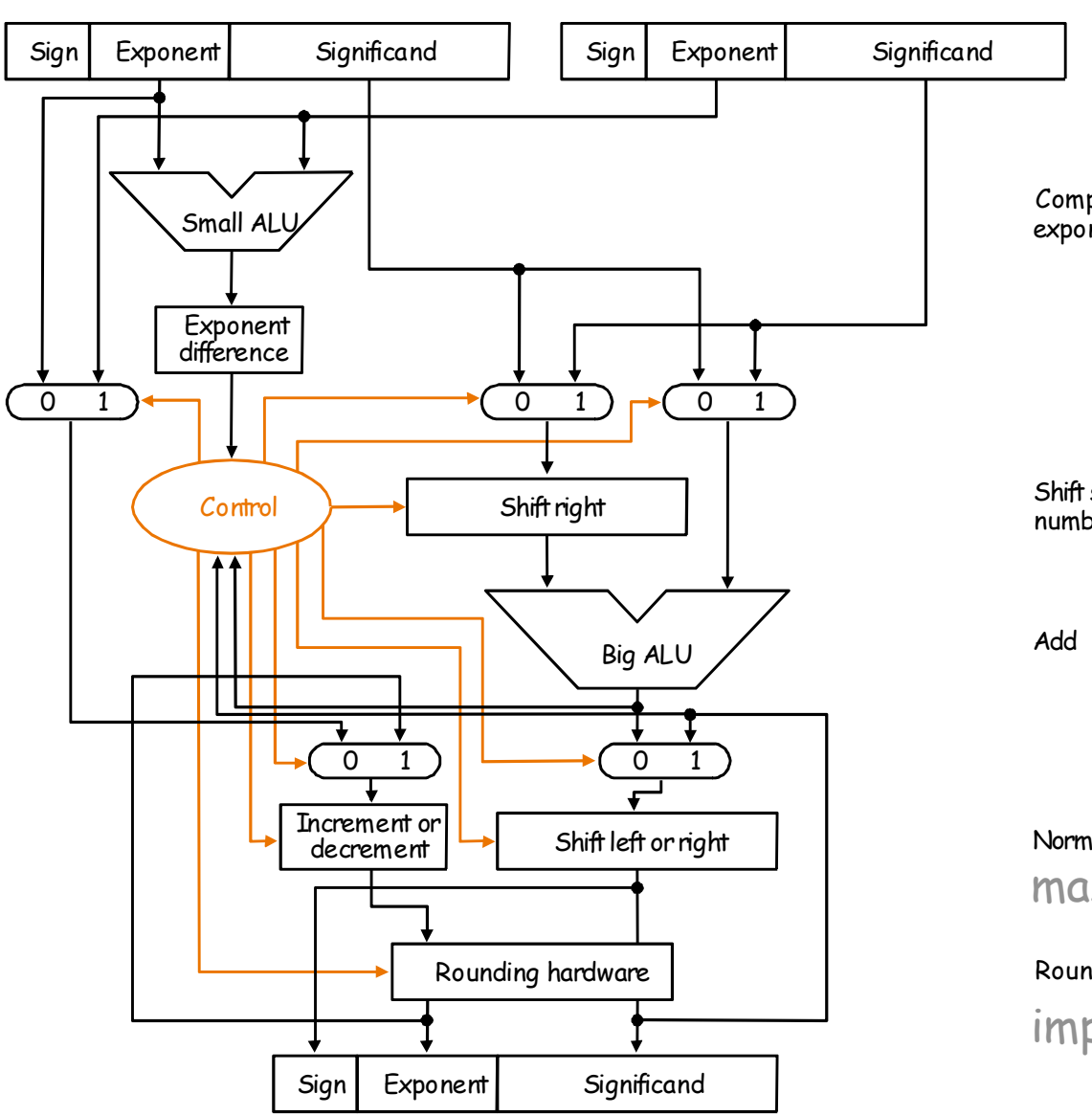
1.001510×10^{2} step 3

Is $(e > E_{max})$ or $(e < E_{min})$?

1.002×10^{2} step 4

Okay in this case, but 9.9999, for example, would require re-normalization after rounding

Datapath



Compare exponents

Shift smaller number right

step 1

Add

step 2

Normalize

step 3

max shift distance?

Round

step 4

implies an addition?

[Figure 3.17 from P&H, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Rounding

- "Nearest" rounding
  - the default rounding mode
  - as if arithmetic is performed with infinite precision and the infinitely precise result is rounded to the closet representable value
  - if the infinitely precise result is exactly mid-way between 2 representable values then choose the one that has an "even" significand

- Also, directed rounding modes
  - toward 0
  - toward ∞
  - toward $-\infty$

Round and Sticky

- When two significands are aligned for addition, some bits of the smaller value appears to not matter

  Can we throw them away? Not quite!

$$\begin{array}{rl} 9.999 & \times 10^1 \\ +\ 0.016\,10 & \times 10^1 \\ \hline 10.015\,10 & \times 10^1 \end{array}$$

- Given the number $b_{n-1}....b_2b_1b_0\ b\ b\ b\ b\ b\ ....$

$b_{n-1}....b_2b_1b_0$ — precision

r — round bit

$s_1s_2s_3.....$ — or'ed into a "sticky" bit

if r==0 then $b_{n-1}....b_2b_1b_0$

if r==1 && sticky==1 then $b_{n-1}....b_2b_1b_0+1$

if r==1 && sticky==0 then round to even

Guard

- When subtracting 2 significands
  - may lose the MSB
  - requires a left-shift to normalize
  - must keep an extra "guard" bit just in case

$$1.000 \times 2^0$$
$$- \quad 1.111 \times 2^{-3}$$

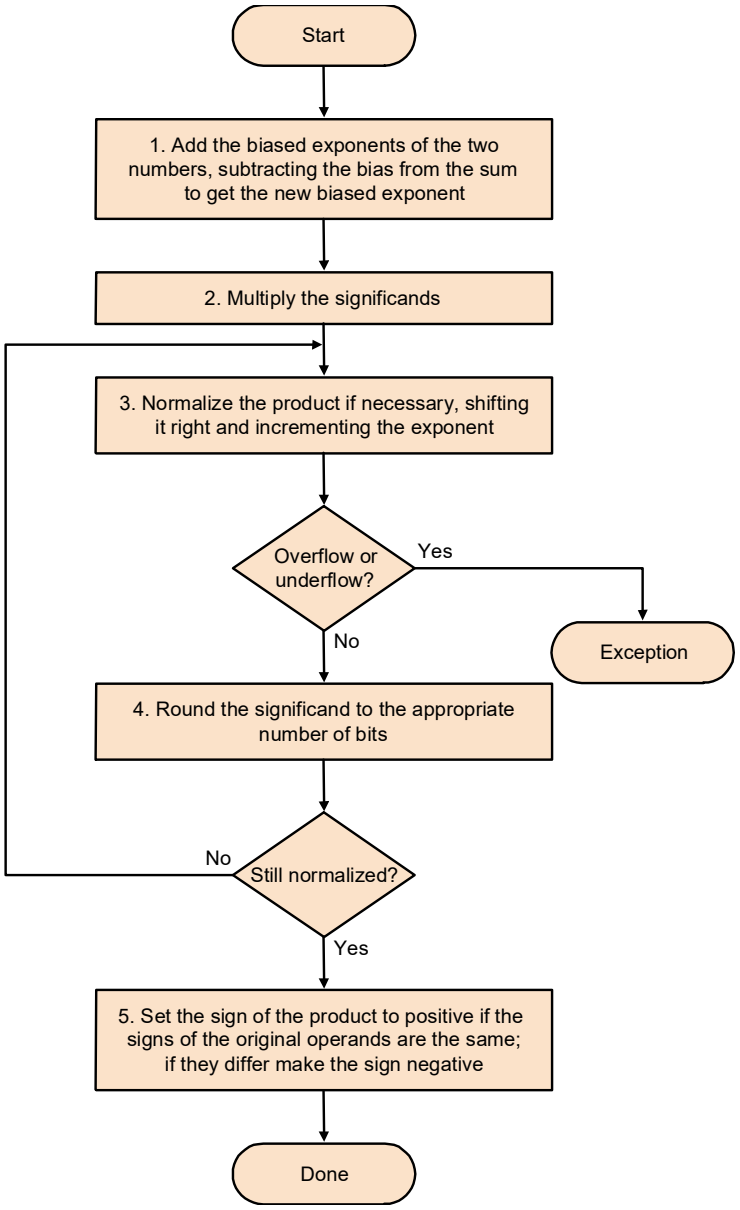
$$1.000 \times 2^0$$
$$- \quad 0.0011(s=1) \times 2^0$$
$$0.1100(s=1) \times 2^0$$

$$1.100(s=1) \times 2^{-1}$$

- What happens if you lose more than 1 MSB in subtraction?
  - only possible if two subtracted values are similar in magnitude
  - at most 1 initial right-shift in the first place

1 guard, 1 round and 1 sticky are sufficient always!

FP Mult



$(1.110 \times 10^{10}) \times (9.200 \times 10^{-5})$

$(1.110 \times 9.200) \times (10^{10} \times 10^{-5})$

$(1.110 \times 9.200) \times 10^{5}$ step 1

don't forget exponent is biased in IEEE float

10.212000×10^{5} step 2

1.0212000×10^{6} step 3

Is $(e > E_{max})$ or $(e < E_{min})$?

1.021×10^{6} step 4

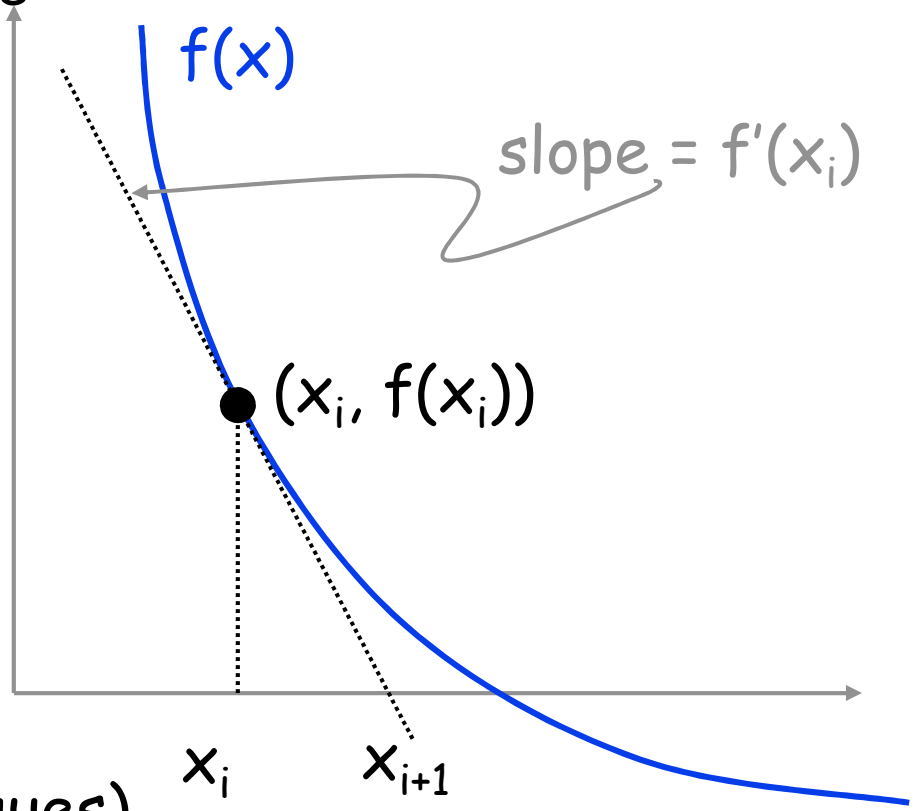
no carry, looks good

$sign_A \oplus sign_B$ step 5

FP Division by Newton Iteration

- Compute 1/b by finding the root of f(x)=1/x - b using Newton-Raphason
  - guess initial x_0
  - for x_i find root of a straight line going through $(x_i, f(x_i))$ with slope $f'(x_i)$
  - iterate using solved root as new guess $x_{i+1}=x_i-f(x_i)/f'(x_i)$ $=x_i(2-x_ib)$

  Precision double each iteration



- Fast division and sqrt if fast addition and multiplication are available (either as HW or SW techniques)

computing 1/3 using "double" and x_0=0.3 converges to 16 decimal places in 4 iter

Typical FP Latencies

- FAdd: 1 ~ 2 cycles, fully pipelined
- FMult: ~4 cycles, fully pipelined
- FDiv: 15~20 cycles, unpipelined

Relative to 1-cycle integer add

von Neumann on Floating Point

5.3. Several of the digital computers being built or planned in this country and England are to contain a so-called "floating decimal point". This is a mechanism for expressing each word as a characteristic and a mantissa—e.g. 123.45 would be carried in the machine as (0.12345,03), where the 3 is the exponent of 10 associated with the number. There appear to be two major purposes in a "floating" decimal point system both of which arise from the fact that the number of digits in a word is a constant, fixed by design considerations for each particular machine. The first of these purposes is to retain in a sum or product as many significant digits as possible and the second of these is to free the human operator from the burden of estimating and inserting into a problem "scale factors"—multiplicative constants which serve to keep numbers within the limits of the machine.

There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time so consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point one must waste memory capacity which could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.

There are certainly some problems within the scope of our device which really require more than 2^{-40} precision. To handle such problems we wish to plan in terms of words whose lengths are some fixed integral multiple of 40, and program the machine in such a manner as to give the corresponding aggregates of 40 digit words the proper treatment. We must then consider an addition or multiplication as a complex operation programmed from a number of primitive additions or multiplications (cf. §9, Part II). There would seem to be considerable extra difficulties in the way of such a procedure in an instrument with a floating binary point.

The reader may remark upon our alternate spells of radicalism and conservatism in deciding upon various possible features for our mechanism. We hope, however, that he will agree, on closer inspection, that we are guided by a consistent and sound principle in judging the merits of any idea. We wish to incorporate into the machine—in the form of circuits—only such logical concepts as are either necessary to have a complete system or highly convenient because of the frequency with which they occur and influence they exert in the relevant mathematical situations.

Burks, Goldstein, von Neumann, Preliminary discussion of the logical design of an electronic computing instrument,1946.

Intel Pentium FDIV Bug

If you believe "bug-free" processor designs exist in this world, try reading the Intel Core 2 Duo errata sheet (aka "Specification Updates" in polite circles) ftp://download.intel.com/design/mobile/SPECUPDT/31407906.pdf

Further Reading

- If you are interested about computer arithmetic, a great place to start is Appendix H of *Computer Architecture: A quantitative approach* by Hennessy and Patterson

- go to www.mkp.com/CA3 and follow link to "companion site"