William Kahan: 1989 Turing Award Winner

Dan W. Randolph
Department of Computer Science
University of Wyoming, Laramie, WY
for COSC 5000

May 16, 2003

1 Introduction

William Kahan won the 1989 Turing Award for his lifelong efforts in improving floating- point computations. Kahan has dedicated himself to making the world safe for numerical computations [1].

Kahan has published many articles and algorithms on numerical methods. The order of operations can be modified to improve the accuracy of floating-point computations. This can help overcome the problems associated with underflow, round off, and truncation. Kahan was a major contributor in the design of the IEEE 754 Floating Point Standard. This is his most important contribution to the advancement of Computer Science.

2 Biographic Information

Kahan was born in 1933 in Canada. He received his Ph.D. in Mathematics from the University of Toronto in 1958. He has worked as a professor of Mathematics and Computer Science throughout his career. He began his academic career at the University of Toronto and has worked the majority of his career at the University of California at Berkley. Kahan's current "capacity" at Berkley is "Computer Science Ombudsman" [2].

William Kahan lists the following awards and lectureships on his web page: [2]

- IEEE Emanuel R. Piore Award, 2000
- Soc. Indust. and Appl. Math., 1997 John von Neumann Memorial Lecture
- ACM Fellow, 1994
- ACM Turing Award, 1989
- SIAM Activity Group on Linear Algebra, Prize for Outstanding Paper in 1988-1990 (with J. Demmel), 1991
- ACM 1st G. E. Forsythe Memorial Award, 1972

I found that there was very little biographical information available on Kahan so I sent him an e-mail with a request for some background information. The e-mail with Kahan's response can be found in the Appendix of this report.

3 Floating-Point Arithmetic and the IEEE 754 Standard

In 1976, John Palmer began work at Intel on what became the 8087 math coprocessor chip [3, 4]. Kahan was hired as a consultant to help with the design. Palmer wanted better floating point than what was currently available in the DEC VAX or the IBM 370 implementations. It needed to be reliable and safe for "a mass market" [3]. This work became the bases for the IEEE 754 standard.

People who do applications involving financial calculations, statistics, graphic displays, and calculations involving geometric accuracy are more susceptible to floating-point errors than those who do pure number crunching. The IEEE 754 standard has features to help prevent and diagnose common problems associated with earlier floating-point implementations.

Prior to the IEEE 754 standard, each computer manufacturer had its own way of doing floating-point arithmetic. For example, rounding and exception handling were differed between machines. This made it hard to write software that was portable and reliable across different platforms. The IEEE Floating Point standard has now been widely implemented in general purpose computers and solves these portability problems.

The basic formats of the standard include a 32 bit single precision format, and a 64 bit double precision format. The standard also allows for extended formats. Intel implements the optional double-extended format that uses 80 bits.

Finite real numbers are represented in the form:

$(-1)^s 2^{e+1-bias} \times$ 1.f

The first component in the binary block representation is the sign bit s. If s is 0, the sign is positive. If s is 1, the sign is negative. The next bits are made up of the exponent e. The exponent has a bias added to it so the range of values stored is nonnegative. The last part of the number (1.f) is the significand with f being the fractional part. The finite, real number approximation of the actual value is constructed by applying the appropriate bits in source memory to the above formula. See figure 1 for a schematic diagram of the binary representation for single and double precision values.

The bits in the fractional part are normalized by shifting f left until the leading significand bit becomes one. The binary exponent is decremented for each left shift. If the exponent is zero (smallest possible value), then the significand bit remains zero and the value is said to be in Normal Zero format.

Examples of some values:

Packed $32-Bit\ Hex$	*Unpacked decimal*
3F80 0000	$2^0 \times 1.00\ldots00 = 1$
4000 0000	$2^1 \times 1.00\ldots00 = 2$
3DCC CCCD	$2^{-4} \times 1.60000002384\ldots \approx 0.1$
3F8C CCCD	$2^0 \times 1.10000002384\ldots \approx 1.1$

Special values v are represented as follows: [5]

$Single-Precision$	
$if\ e = 255\ and\ f \neq 0,\ then\ v = NaN$	$(Not\ a\ Number)$
$if\ e = 255\ and\ f = 0,\ then\ v = (-1)^s \infty$	$(Signed\ infinity)$
$if\ 0 < e < 255,\ then\ v = (-1)^s 2^{e-127} \times 1.f$	$(Real\ number)$
$if\ e = 0\ and\ f \neq 0,\ then\ v = (-1)^s 2^{e-126} \times 0.f$	$(Normal\ Zero\ format)$
$if\ e = 0\ and\ f = 0,\ then\ v = (-1)^s 0$	$(Signed\ zero)$

$\underline{Double - Precision}$

$if\ e = 2047\ and\ f \neq 0,\ then\ v = NaN$	$(Not\ a\ Number)$
$if\ e = 2047\ and\ f = 0,\ then\ v = (-1)^s \infty$	$(Signed\ infinity)$
$if\ 0 < e < 2047,\ then\ v = (-1)^s 2^{e-1023} \times 1.f$	$(Real\ number)$
$if\ e = 0\ and\ f \neq 0,\ then\ v = (-1)^s 2^{e-1022} \times 0.f$	$(Normal\ Zero\ format)$
$if\ e = 0\ and\ f = 0,\ then\ v = (-1)^s 0$	$(Signed\ zero)$

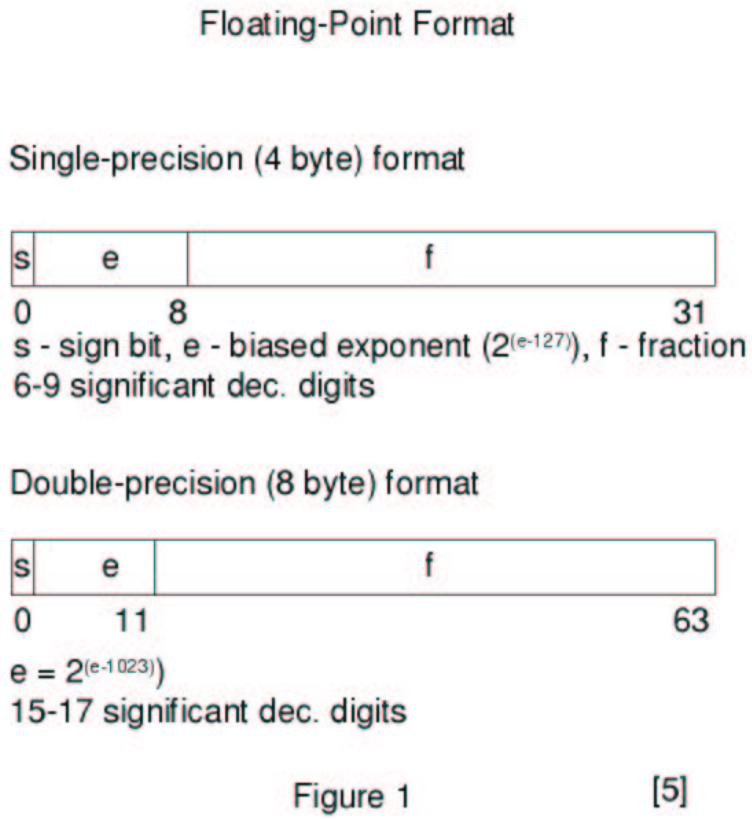

Figure 1 [5]

NaN is produced by invalid operation exceptions. The standard calls for exceptions to be trapped at the user's discretion. If the numerator is nonzero and the denominator is zero, then and exception flag is raised and the result is ∞. If both values are zero, the result is NaN. The set of operations that deliver a NaN result are:

$$real\sqrt{(Negative)},\ 0 * \infty,\ 0.0/0.0,\ \infty/\infty,\ REMAINDER(Anything, 0.0),$$
$$REMAINDER(\infty, Anything),\ \infty - \infty\ (when\ signs\ agree)$$

The Normal Zero format allows for gradual underflow. Before IEEE 754, existing floating-point implementations normalized all numbers as a rule. This left a gap between zero and the smallest floating-point number that was much larger than the gap between the rest of the set of normalized values. Gradual underflow eliminates this problem by preserving the remaining fraction bits in a subnormal format.

Gradual underflow was initially controversial [4]. There was a push to keep it out of the standard by Digital Equipment Corporation (DEC). They made the VAX and VMS operating system which was popular at the time. Gradual underflow was the main distinguishing factor between DEC's existing floating-point format and IEEE 754. Some committee members thought that gradual underflow would

slow performance. Kahan knew how to implement it without slowing down floating-point operations, but he could not talk about it until after his confidentiality agreement with Intel expired.

DEC commissioned G.W. Stewart to evaluate the error analysis aspects of gradual underflow. DEC thought that if it was proved to be harmful, then the idea would be killed and DEC would get its floating-point format passed as the new standard. DEC would only need to do minor changes to their existing hardware to handle the other details of IEEE 754 such as exception handling. Stewart reported that gradual underflow was the correct thing to do and DEC lost the battle for control of the IEEE 754 standard.

IEEE 754 uses some additional bits in internal arithmetic operations to improve accuracy and control roundoff. A guard bit, a round bit, and a sticky bit are utilized as part of the arithmetic logic [6]. These extra bits that trail the low order bits of the actual stored data help insure that rounding is done correctly.

If P and Q are floating-point numbers in the same format, and if $1/2 \leq \frac{P}{Q} \leq 2$, then $P - Q$ is computable exactly [7]. But without guard bits, $1.0 - 0.9999$ could result in 0.001 instead of 0.0001.

IEEE 754 does not solve all the problems with floating point computation, but at least it requires manufactures to conform to a standard that is consistent and produces the best overall accuracy.

The following C program demonstrates some limitations of binary floating-point computation:

```
main() {
  int i,k;
  double j = 1.0;
  double sum = 0;

  //sum from j=1.0,1.1,1.2,...,2.0 1000000 times
  for (i = 0; i < 1000000; i++) {
     sum += j;
     j += .1;
     if (j > 2.0) {
       j = 1.0;
     }
  }
  j = 1.0 + .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1;
  printf("j > 2.0, j = %18.16f\n",j);
  printf("sum = %f\n",sum);
}
```

Produces the output:
$j > 2.0,\ j = 2.0000000000000009$
$sum = 1450000.000000$

Now this result is clearly wrong. You can see from the output that adding a sequence of 1.0 plus ten 0.1's does not produce exactly 2.0. A slightly larger value is produced from some rounding in the binary representation of the decimal fractions. So the result is that the logic fails because the number 2.0 was never added to the sum as intended. The correct result should have been 1500000.0, but the range of j=1.0 to 1.9 was used instead of 1.0 to 2.0. This was due to j being reset to 1.0 before j was added to sum when j was equal to 2.0.

Another thing that can be confusing is the C formatting in printf functions. Printed decimal digits are automatically rounded so the format %17.15f would produce the output 2.000000000000001 instead of 2.0000000000000009, and the default (%f) output format is just 2.000000.

To fix the summing bug, the inequality test could be changed to:

```
if (j >= 2.1)
```

This will produce the output:
$sum = 1499999.5000000000000000$

I also tried changing the summing loop to:

```
int k = 0;
for (i = 0; i < 1000000; i++) {
   sum += j + (k * .1);
   ++k;
   if (k > 10)
     k = 0;
}
```

This produced the same result as the previous workaround.

I find it interesting that the summation of 1.0 to 2.0 by increments of 0.1 produces a roundup error, but the summation of the same sequence 1000000 times produces a rounding error that is 0.5 less than what the actual value should be. The lesson to be learned here is that for best results, programs should use multiplication instead of summation when possible to get more accurate results.

4 Conclusion

William Kahan, along with his associates in the IEEE 754 working group were able to design a floating-point standard that has made computers more accessible, easier to program, and more accurate than was previously possible. You no longer have to be a numerical expert or buy expensive math libraries to get good results. But working with floating-point number still requires some common sense.

In an area that is not well understood by many, Kahan has continued to champion the benefits of the standard. Exception handling is one area where IEEE 754 has not always been fully implemented or made accessible to programmers. Kahan has continued to work towards educating users about these lesser know features of IEEE 754 and proposing ways to solve this problem.

For example, Kahan has worked to get standard library functions added to the C language so that the status flags in IEEE 754 can be checked and changed as required by the programmer. The ISO C99 language standard defines functions to query and manipulate the floating-point status word [8]. There are also functions to control the rounding mode. Kahan is also pushing to get Java changed to comply with the IEEE 754 standard [9].

One particular quote from Kahan best summarizes his mission: [3]

> This type of analysis is what should convey a certain sense of satisfaction that people often attempt to get prematurely from aesthetic criteria. They believe that if a program or algorithm looks pretty, why then, it must be OK. If you think that beauty is the sole criterion, remember that beauty is in the eye of the beholder, and in the eyes of a bug, a rose is just fodder!

References

[1] ACM Turing Award Citation. http://www.acm.org/awards/turing_citations/kahan.html

[2] William Kahan's home page. http://www.cs.berkeley.edu/~wkahan/

[3] Jack Woehr. A Conversation with William Kahan. Dr. Dobb's Journal, Nov. 1997. http://www.ddj.com/documents/s=935/ddj9711a/9711a.htm

[4] Charles Severance. IEEE 754: An Iinterview with William Kahan. IEEE Computer, Vol 31, n3:114,115, March 1998.

[5] Draft 8.0 of IEEE Task P754. IEEE Computer Society, Vol 14, n3: 51-62, March 1981.

[6] Joseph Cavanagh. Digital Computer Arithmetic. McGraw-Hill, 1984.

[7] William Kahan. What can you learn about Floating-Point Arithmetic in One Hour? http://cch.loria.fr/documentation/IEEE754/

[8] GNU C Library Arithmetic Functions. http://www.gnu.org/manual/glibc-2.2.3/html_chapter/libc_20.html#SEC405

[9] William Kahan. How Java's Floating-Point Hurts Everyone Everywhere. http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf

A Appendix

Correspondence between Dan Randolph and William Kahan:

e-mail sent to Kahan:

Dr. Kahan,

I am a computer science graduate student at the University of Wyoming. I have previously worked in the seismic data processing business to aid in oil and gas exploration so I can appreciate the importance of reliable floating point processing.

I am in a seminar class where we are each doing a report and talk on our choice of a Turing Award winner. I chose to do my report on your efforts in the area of numerical computing. My experience in numerical processing was for clients who placed a great deal of investment in the accuracy of the processing results. 300000 Dolars was the typical cost of drilling a well, for example.

I am supposed to supply some background (biographical) information about you in my report and would be grateful if you could supply me with whatever information you feel is appropriate to present in front of a seminar of graduate students. Also, I would like to know what winning the Turing Award has meant to you.

Here is a link to our course web site: http://www.cs.uwyo.edu/~jlc/courses/5000/

Here is a link to my personal home page: http://www.mapsharecorp.com/danrandolph.html

Thank you for your time.

Sincerely,
Dan W. Randolph
Graduate Student in Computer Science
University of Wyoming

Kahan's reply:

Dan: I have just returned to my computers to find a monstrous heap of e-mail some of which must be answered very soon, so let this response be a start towards what you seem to need.

My web page will give you some idea of the range of my activities. In particular,
`http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`
`http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`
`http://www.cs.berkeley.edu/~wkahan/MathSand.pdf`
may answer some of your specific questions. See also my paper
"A Survey of Error Analysis" in "Info. Processing 71" (1972) pp.1214-1239; North-Holland Publ. Co., Amsterdam,
particularly the story about the graduate student. And the story about U-Boats in World War II may entertain you a bit.

As for 'what winning the Turing Award has meant', my habit every morning when I look in the mirror to shave is to ask "... and what have you done RECENTLY?"

With best wishes,
Prof. W. Kahan