Digits	Input	Bits
1	$49517601571415211 \cdot 2^{-094}$	63
2	$49517601571415211 \cdot 2^{-095}$	60
3	$54390733528642804 \cdot 2^{-133}$	63
4	$71805402319113924 \cdot 2^{-157}$	62
5	$40435277969631694 \cdot 2^{-179}$	61
6	$57241991568619049 \cdot 2^{-165}$	61
7	$65224162876242886 \cdot 2^{+058}$	65
8	$70173376848895368 \cdot 2^{-138}$	61
9	$37072848117383207 \cdot 2^{-099}$	61
10	$56845051585389697 \cdot 2^{-176}$	64
11	$54791673366936431 \cdot 2^{-145}$	64
12	$66800318669106231 \cdot 2^{-169}$	64
13	$66800318669106231 \cdot 2^{-170}$	61
14	$66574323440112438 \cdot 2^{-119}$	65
15	$65645179969330963 \cdot 2^{-173}$	62
16	$61847254334681076 \cdot 2^{-109}$	63
17	$39990712921393606 \cdot 2^{-145}$	62
18	$59292318184400283 \cdot 2^{-149}$	62
19	$69116558615326153 \cdot 2^{-143}$	65
20	$69116558615326153 \cdot 2^{-144}$	62
21	$39462549494468513 \cdot 2^{-152}$	63
22	$39462549494468513 \cdot 2^{-153}$	61

Table 21: Stress Inputs for Converting 56-bit Binary to Decimal, > 1/2 ULP

Digits	Input	Bits
1	$50883641005312716 \cdot 2^{-172}$	65
2	$38162730753984537 \cdot 2^{-170}$	64
3	$50832789069151999 \cdot 2^{-101}$	64
4	$51822367833714164 \cdot 2^{-109}$	62
5	$66840152193508133 \cdot 2^{-172}$	64
6	$55111239245584393 \cdot 2^{-138}$	64
7	$71704866733321482 \cdot 2^{-112}$	62
8	$67160949328233173 \cdot 2^{-142}$	61
9	$53237141308040189 \cdot 2^{-152}$	63
10	$62785329394975786 \cdot 2^{-112}$	62
11	$48367680154689523 \cdot 2^{-077}$	61
12	$42552223180606797 \cdot 2^{-102}$	62
13	$63626356173011241 \cdot 2^{-112}$	62
14	$43566388595783643 \cdot 2^{-099}$	64
15	$54512669636675272 \cdot 2^{-159}$	61
16	$52306490527514614 \cdot 2^{-167}$	67
17	$52306490527514614 \cdot 2^{-168}$	65
18	$41024721590449423 \cdot 2^{-089}$	62
19	$37664020415894738 \cdot 2^{-132}$	60
20	$37549883692866294 \cdot 2^{-093}$	62
21	$69124110374399839 \cdot 2^{-104}$	65
22	$69124110374399839 \cdot 2^{-105}$	62

Table 20: Stress Inputs for Converting 56-bit Binary to Decimal, $< 1/2$ ULP

Digits	Input	Bits
1	$9 \cdot 10^{+26}$	9
2	$79 \cdot 10^{-8}$	10
3	$393 \cdot 10^{+26}$	14
4	$9171 \cdot 10^{-40}$	17
5	$56257 \cdot 10^{-16}$	26
6	$281285 \cdot 10^{-17}$	26
7	$4691113 \cdot 10^{-43}$	29
8	$29994057 \cdot 10^{-15}$	30
9	$834548641 \cdot 10^{-46}$	33
10	$1058695771 \cdot 10^{-47}$	37
11	$87365670181 \cdot 10^{-18}$	43
12	$872580695561 \cdot 10^{-36}$	43
13	$6638060417081 \cdot 10^{-51}$	47
14	$88473759402752 \cdot 10^{-52}$	50
15	$412413848938563 \cdot 10^{-27}$	54
16	$5592117679628511 \cdot 10^{-48}$	63
17	$83881765194427665 \cdot 10^{-50}$	63
18	$638632866154697279 \cdot 10^{-35}$	64
19	$3624461315401357483 \cdot 10^{-53}$	67
20	$75831386216699428651 \cdot 10^{-30}$	70
21	$356645068918103229683 \cdot 10^{-42}$	74
22	$7022835002724438581513 \cdot 10^{-33}$	77

Table 19: Stress Inputs for Conversion to 56-bit Binary, > 1/2 ULP

Digits	Input	Bits
1	$7 \cdot 10^{-27}$	9
2	$37 \cdot 10^{-29}$	10
3	$743 \cdot 10^{-18}$	16
4	$7861 \cdot 10^{-33}$	19
5	$46073 \cdot 10^{-30}$	20
6	$774497 \cdot 10^{-34}$	25
7	$8184513 \cdot 10^{-33}$	26
8	$89842219 \cdot 10^{-28}$	34
9	$449211095 \cdot 10^{-29}$	34
10	$8128913627 \cdot 10^{-40}$	38
11	$87365670181 \cdot 10^{-18}$	43
12	$436828350905 \cdot 10^{-19}$	44
13	$5569902441849 \cdot 10^{-49}$	46
14	$60101945175297 \cdot 10^{-32}$	52
15	$754205928904091 \cdot 10^{-51}$	54
16	$5930988018823113 \cdot 10^{-37}$	57
17	$51417459976130695 \cdot 10^{-27}$	62
18	$826224659167966417 \cdot 10^{-41}$	65
19	$9612793100620708287 \cdot 10^{-57}$	68
20	$93219542812847969081 \cdot 10^{-39}$	71
21	$544579064588249633923 \cdot 10^{-48}$	74
22	$4985301935905831716201 \cdot 10^{-48}$	80

Table 18: Stress Inputs for Conversion to 56-bit Binary, $< 1/2$ ULP

Digits	Input	Bits
1	$12676506 \cdot 2^{-102}$	32
2	$12676506 \cdot 2^{-103}$	29
3	$15445013 \cdot 2^{+086}$	34
4	$13734123 \cdot 2^{-138}$	32
5	$12428269 \cdot 2^{-130}$	30
6	$15334037 \cdot 2^{-146}$	31
7	$11518287 \cdot 2^{-041}$	30
8	$12584953 \cdot 2^{-145}$	31
9	$15961084 \cdot 2^{-125}$	32
10	$14915817 \cdot 2^{-146}$	31
11	$10845484 \cdot 2^{-102}$	30
12	$16431059 \cdot 2^{-061}$	29

Table 16: Stress Inputs for Converting 24-bit Binary to Decimal, $< 1/2$ ULP

Digits	Input	Bits
1	$16093626 \cdot 2^{+069}$	30
2	$9983778 \cdot 2^{+025}$	31
3	$12745034 \cdot 2^{+104}$	31
4	$12706553 \cdot 2^{+072}$	31
5	$11005028 \cdot 2^{+045}$	30
6	$15059547 \cdot 2^{+071}$	31
7	$16015691 \cdot 2^{-099}$	29
8	$8667859 \cdot 2^{+056}$	33
9	$14855922 \cdot 2^{-082}$	35
10	$14855922 \cdot 2^{-083}$	33
11	$10144164 \cdot 2^{-110}$	32
12	$13248074 \cdot 2^{+095}$	33

Table 17: Stress Inputs for Converting 24-bit Binary to Decimal, $> 1/2$ ULP

Digits	Input	Bits
1	$5 \cdot 10^{-20}$	7
2	$67 \cdot 10^{+14}$	13
3	$985 \cdot 10^{+15}$	15
4	$7693 \cdot 10^{-42}$	17
5	$55895 \cdot 10^{-16}$	25
6	$996622 \cdot 10^{-44}$	27
7	$7038531 \cdot 10^{-32}$	32
8	$60419369 \cdot 10^{-46}$	33
9	$702990899 \cdot 10^{-20}$	35
10	$6930161142 \cdot 10^{-48}$	41
11	$25933168707 \cdot 10^{+13}$	42
12	$596428896559 \cdot 10^{+20}$	45

Table 14: Stress Inputs for Conversion to 24-bit Binary, $< 1/2$ ULP

Digits	Input	Bits
1	$3 \cdot 10^{-23}$	10
2	$57 \cdot 10^{+18}$	12
3	$789 \cdot 10^{-35}$	16
4	$2539 \cdot 10^{-18}$	18
5	$76173 \cdot 10^{+28}$	22
6	$887745 \cdot 10^{-11}$	24
7	$5382571 \cdot 10^{-37}$	26
8	$82381273 \cdot 10^{-35}$	32
9	$750486563 \cdot 10^{-38}$	38
10	$3752432815 \cdot 10^{-39}$	38
11	$75224575729 \cdot 10^{-45}$	42
12	$459926601011 \cdot 10^{+15}$	46

Table 15: Stress Inputs for Conversion to 24-bit Binary, $> 1/2$ ULP

9 Appendix

Tables 14 through 21 give stress inputs for decimal-to-binary and binary-to-decimal conversion for 24 and 56 bit floating-point mantissas (IEEE single precision and VAX *D* format, respectively). For each type of conversion, the first table gives stress inputs with excesses lying just below a half ULP, and the second inputs with excesses lying just above a half ULP.

References

[C+84] W. J. Cody et al. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro*, pages 86–100, August 1984.

[Cli90] William D. Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, New York*, June 1990.

[Coo84] Jerome Toby Coonen. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*. PhD thesis, Department of Mathematics, University of California, Berkeley, June 1984.

[Gay90] David M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Technical report, AT&T Bell Laboratories, November 1990. Numerical Analysis Manuscript 90-10.

[Hou91] David Hough. Testbase. Private communication, March 1991.

[JW90] Guy L. Steele Jr. and Jon L. White. How to print floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, New York*, June 1990.

[Knu81] Donald E. Knuth. *Seminumerical Algorithms*. Addison–Wesley, second edition, 1981.

[NZ80] Ivan Niven and Herbert S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, fourth edition, 1980.

[Pet91] Tim Peters. Minimizing `b*s mod c` over a range of s; "the worst" case. Internet `validgh!numeric-interest@uunet.uu.net` electronic mailing list, April 7th 1991.

8 Acknowledgments

This work was supported by an NSF Graduate Fellowship and by the generosity of the Lawrence Berkeley Laboratory's Computer Systems Engineering Department, for which the author is grateful.

I am much indebted to David Hough of Sun Microsystems, Inc., for his aid in understanding and modifying *Testbase*, as well as for lucid explanations of the issues underlying the testing of decimal–binary conversion. I am equally indebted to Tim Peters of Kendall Square Research Corporation, for his invaluable help in understanding the workings of his modular minimization algorithms and the motivation behind them as developed in Section 3.

I wish to thank the many people who made their systems available for testing: Wes Bethel, Keith Bostic, Jane Colman, Nancy Johnston, Craig Leres, Chris Saltmarsh, Lindsay Schachinger, and Marc Teitelbaum. I also wish to thank Ed Theil for providing the basic computational support for developing the extended *Testbase*, without which this project would have been impossible.

Finally, I especially wish to thank my wife, Lindsay, for her continual patience and support during the pursuit of this project.

All of these efforts are very much appreciated.

7 Summary

Our measurements found that only Sun Microsystems' and David Gay's conversion routines are completely correct. In general, VAX/VMS conversion is very good. IBM RS/6000, HP-UX 9000, SGI Iris 4D/35, and DECsystem 5500 all provide conformant (but not fully correct) implementations. VAX 4.3 BSD Unix loses conformance with 15 significant decimal digits and suffers from monotonicity errors, a poor implementation. Stardent ST3000VX is not conformant regardless of decimal significance and also suffers from monotonicity errors. It is a very poor implementation.

Stress-testing proved invaluable for uncovering incorrect single precision conversions. It did not, however, reveal any faults in double precision conversion that *Testbase* did not uncover independently. As *stress-testing* for double precision values is computationally intensive, this suggests that double precision conversion can be adequately tested using *Testbase* alone.

System	Input	Output	Correct
dec5500	$5500000000000000 \cdot 2^{0}$	5e+15	6e+15
hpux	$6236527588955251 \cdot 2^{525}$	6.8e+173	6.9e+173
rs6000	$5587935447692871 \cdot 2^{27}$	8e+23	7e+23
sgi	$5500000000000000 \cdot 2^{0}$	5e+15	6e+15
stardent	$4503599627370726 \cdot 2^{54}$	8.112963841461085e+31	8.112963841461083e+31
vaxunix	$36028797018963978 \cdot 2^{38}$	9.903520314283046e+27	9.903520314283045e+27
vaxunix	$48828121778774530 \cdot 2^{11}$	9.9999993402930235e+19	9.9999993402930237e+19
vms-G	$9007195228209151 \cdot 2^{67}$	$1329 \cdots 128.2$	$1329 \cdots 128$

Table 11: Sample Errors in Binary-to-Decimal Double Precision Conversions

System	Input	Output	Correct
dec5500	7.038531e-26	$11420670 \cdot 2^{-107}$	$11420669 \cdot 2^{-107}$
hpux	7.038531e-26	$11420670 \cdot 2^{-107}$	$11420669 \cdot 2^{-107}$
rs6000	7.038531e-26	$11420670 \cdot 2^{-107}$	$11420669 \cdot 2^{-107}$
sgi	7.038531e-26	$11420670 \cdot 2^{-107}$	$11420669 \cdot 2^{-107}$
stardent	4.1358803e34	$16704688 \cdot 2^{91}$	$16704687 \cdot 2^{91}$
vaxunix	9.55610858e-6	$10507053 \cdot 2^{-40}$	$10507052 \cdot 2^{-40}$
vms-F	9.55610858e-6	$10507053 \cdot 2^{-40}$	$10507052 \cdot 2^{-40}$

Table 12: Sample Errors in Decimal-to-Binary Single Precision Conversions

System	Input	Output	Correct
dec5500	1e+126	$6653062250012736 \cdot 2^{366}$	$6653062250012735 \cdot 2^{366}$
hpux	1e+126	$6653062250012736 \cdot 2^{366}$	$6653062250012735 \cdot 2^{366}$
rs6000	9.51206426453718e-27	$6628941296109132 \cdot 2^{-139}$	$6628941296109133 \cdot 2^{-139}$
sgi	1e+126	$6653062250012736 \cdot 2^{366}$	$6653062250012735 \cdot 2^{366}$
stardent	3e+97	$7906648457422895 \cdot 2^{271}$	$7906648457422892 \cdot 2^{271}$
vaxunix	9e+26	$52386894822120666 \cdot 2^{34}$	$52386894822120667 \cdot 2^{34}$

Table 13: Sample Errors in Decimal-to-Binary Double Precision Conversions

System	Input	Output	Correct
dec5500	$15000000 \cdot 2^{0}$	1e+07	2e+07
hpux	$12680205 \cdot 2^{-107}$	7.8147796833e-26	7.8147796834e-26
rs6000	$8667859 \cdot 2^{56}$	6.2458506e+23	6.2458507e+23
sgi	$15000000 \cdot 2^{0}$	1e+07	2e+07
stardent	$8500000 \cdot 2^{0}$	9e+06	8e+06
vaxunix	$12584953 \cdot 2^{-145}$	2.8216441e-37	2.8216440e-37

Table 10: Sample Errors in Binary-to-Decimal Single Precision Conversions

Systems with a number and not "all" in the final column have a limit on of how many digits the decimal input can consist. Values beyond this limit result in *sscanf* failures.

stardent suffers from monotonicity errors. When the values

```
5.5225015152609010e+14
5.5225015152609011e+14
```

(which differ by one ULP) are converted to binary, the results are

$$8836002424417442 \cdot 2^{-4}$$
$$8836002424417441 \cdot 2^{-4}$$

respectively, differing by 1 ULP in the wrong direction.

Tables 10 through 13 show sample errors made by the various systems. We omit systems that only generated errors for greater than 17 significant digits, as this is the maximum significance required by the IEEE standard. With that constraint, only *stardent* and *vaxunix* were found to suffer grievous errors, 2 ULP's or larger. The other errors are all incorrect roundings close enough to be conformant. The *VMS-G* error of gratuitously adding ".2" to *%f* conversions is puzzling. We found numerous values for which the system made this error or similar ones (e.g., adding ".00002" for a *%.5f* format). In each case, the correctly converted decimal value had no fractional part.

System	Correct	Conformant	2 ULP's	Monotonicity	*%f* format
dec5500	0	17	17	22+	17
dgay	22+	22+	22+	22+	all
hpux	1	17	17	22+	17
rs6000	0	17	17	22+	17
sgi	0	17	17	22+	17
stardent	0	14	15	22+	17
sun3	22+	22+	22+	22+	all
sun3-68881	22+	22+	22+	22+	all
sun4	22+	22+	22+	22+	all
vaxunix	0	15	16	16	17
vms-D	22+	22+	22+	22+	all
vms-G	0/22+	0/22+	0/22+	22+	128

Table 8: Summary of Binary-to-Decimal Double Precision Conversion

System	Correct	Conformant	2 ULP's	Monotonicity	*%f* format
dec5500	0	17	17	22+	63
dgay	22+	22+	22+	22+	all
hpux	1	17	(9)	22+	all
rs6000	14	(7)	(7)	(17)	all
sgi	0	17	17	22+	60
stardent	0	0	0	16	62
sun3	22+	22+	22+	22+	all
sun3-68881	22+	22+	22+	22+	all
sun4	22+	22+	22+	22+	all
vaxunix	0	19	19	22+	all
vms-D	17	22+	22+	22+	all
vms-G	17	22+	22+	22+	all

Table 9: Summary of Decimal-to-Binary Double Precision Conversion

System	Correct	Conformant	2 ULP's	Monotonicity
dec5500	**7**	12+	12+	12+
hpux	**7**	12+	12+	12+
rs6000	**7**	12+	12+	12+
sgi	**7**	12+	12+	12+
stardent	**8**	12+	12+	12+
sun3	12+	12+	12+	12+
sun3-68881	12+	12+	12+	12+
sun4	12+	12+	12+	12+
vaxunix	**9**	12+	12+	12+
vms-F	**9**	12+	12+	12+

Table 7: Summary of Decimal-to-Binary Single Precision Conversion

rectly rounded results. For *dec5500, hpux, rs6000* and *sgi* incorrect rounding occurred at 32 stress bits; for *stardent*, 31 stress bits; and for *vaxunix* and *vms-F*, 34 stress bits.

Measurements for double precision binary-to-decimal conversion are summarized in Table 8. The "0/22+" entries for *vms-G* indicate the failure of that system to correctly perform *%f* format conversions. An example is given in Table 11 below.

vaxunix suffers from monotonicity errors. When the values:

$$65725925722776214 \cdot 2^{55}$$
$$65725925722776215 \cdot 2^{55}$$

(which differ by one ULP) are converted to decimal, the results are

```
2.36802603674940685e+33
2.36802603674940680e+33
```

respectively, differing by 5 ULP's in the wrong direction.

Finally, Table 9 summarizes tests of double precision decimal-to-binary conversion. Values listed in parentheses indicate a failing occurring only for subnormal values. For example, *rs6000* does not always convert 7 or more digit decimals to their subnormal binary equivalents within one ULP, but for non-subnormal values, conversion was correct for all tested significances.

System	Correct	Conformant	2 ULP's	Monotonicity	*%f* format
dec5500	0	12+	12+	12+	17
hpux	**10**	12+	12+	12+	17
rs6000	**7**	12+	12+	12+	17
sgi	0	12+	12+	12+	17
stardent	0	12+	12+	12+	17
sun3	12+	12+	12+	12+	all
sun3-68881	12+	12+	12+	12+	all
sun4	12+	12+	12+	12+	all
vaxunix	**7**	12+	12+	12+	17
vms-F	12+	12+	12+	12+	all

Table 6: Summary of Binary-to-Decimal Single Precision Conversion

- Non-extended *Testbase* testing to check for conformant rounding.

The results for tests of single-precision binary-to-decimal conversion are summarized in Table 6. The first column lists the name of the system. The second through fifth columns give the maximum decimal significance for which conversion was correct (within a half ULP), conformant (within one ULP), within two ULP's, and monotonic. A value of "12+" indicates that the system achieved the given level of performance for all significances from 1 through 12. A value of 0 indicates the system failed to achieve the given level for a decimal significance of 1. A boldface value indicates that only *stress-testing* revealed the failing. For *hpux* this occurred for a minimum of 32 stress bits; for *rs6000*, 29 stress bits; and for *vaxunix*, 31 stress bits.

The final column indicates how the system performs *%f* conversions. This conversion specifies no exponent to be used in the output, only decimal digits (perhaps with a decimal point). An entry of 17 indicates that at most 17 digits are given, followed by enough zeroes to reach the decimal point and pad it beyond to the requested significance. An entry of *all* indicates that the system generates all digits up to those requested beyond the decimal point (and does so correctly). Either behavior is allowed by the IEEE standard.

The results for tests of single-precision decimal-to-binary conversion are summarized in Table 7. Clearly *stress-testing* is invaluable for catching incor-

We ran the extended version of *Testbase* on the 13 systems listed in Table 5. *dgay* refers to freely available routines for decimal–binary conversion written by David M. Gay of AT&T Bell Laboratories, described in [Gay90]. It was run on a Silicon Graphics machine, though presumably that had no effect on its performance. These routines do not include single precision conversion. For all other systems, we tested both single and double precision conversion, and for VMS we tested the three provided single and double precisions[4].

The tests were conducted as follows. On each system we tested single precision conversion for 1–12 decimal digits; for double precision, 1-22 digits were tested[5]. For each decimal digit significance, we made four types of tests:

- *stress-testing* to check for correct rounding (within a half ULP). Algorithm 1 was run for each exponent in the floating-point range to test stress values with an excess just below and just above half an ULP (as well as testing the intermediate values produced during each iteration of the algorithm). In addition, for each final pair of stress values generated, we made one random test of each of the types listed in Section 2. These latter tests include monotonicity testing; tests of stress values and their intermediate values do not.

- Non-extended *Testbase* testing to check for correct rounding. Our intent was to discover whether *stress-testing* uncovered errors that would not otherwise be detected. 1,376 regions were tested for single precision and 10,336 for double precision. Each region involved a random test of one of the types listed in Section 2 plus associated monotonicity testing (four adjacent values also tested).

- *stress-testing* to check for conformant rounding (within one ULP). When run in this mode, the stress values generated have an excess just below or just above exactly representable.

[4]VMS *F* format is identical in range to IEEE single precision, though it does not include subnormals and extreme values such as *NAN*. *G* format is likewise similar to IEEE double precision. *D* format has a 56-bit mantissa and exponent range -127 to 127. It was the default floating-point format on the VAX we tested; it also apparently is the only floating-point format supported by 4.3 BSD Unix running on a VAX, as noted in the *vaxunix* entry.

[5]Not all degrees of significance were tested for VAX *G* format due to lack of available CPU time. The precisions tested were 1, 5, 8, 10, 12, 14, 16, 17, 18, and 22.

System	Description
dec5500	DECsystem 5500 running Ultrix V2.1 (Rev. 14)
dgay	Conversion routines by David M. Gay
hpux	HP 9000 model S300 running HP-UX release 7.0
rs6000	IBM RS/6000 running AIX 3.1
sgi	Silicon Graphics Personal Iris 4D/35 running IRIX 3.3.2
stardent	Stardent ST3000VX running 4.0 B II
sun3	Sun 3/160 running SunOS 4.1, software floating-point
sun3-68881	Sun 3/160 running SunOS 4.1, MC68881 floating-point
sun4	Sun 4/20 running SunOS 4.1
vaxunix	VAX 8600 running 4.3 BSD Unix (VAX D floating-point format)
vms-D	VAX 11/780 running VMS 5.4, D floating-point format
vms-F	VAX 11/780 running VMS 5.4, F floating-point format
vms-G	VAX 11/780 running VMS 5.4, G floating-point format

Table 5: Tested Conversion Systems

If the conversion routines correctly convert stress values such as these then we know they utilize enough internal precision to correctly convert any value. The conversion routines still might err, though, by not using the extra internal precision when needed. To catch this error, we not only test the input values requiring maximal internal precision, but also each intermediate step corresponding to the value of s at *Step 2* of Algorithm 1. Each step requires higher internal precision to correctly convert. Algorithm 1 typically generates intermediate steps requiring 2–3 extra bits of precision more than the previous step. In the course of running it for all possible output exponents, each threshold of an extra bit of precision is crossed numerous times. Thus if the conversion routines make an error in assessing when a switch to higher internal precision is necessary, our approach will find it.

The results of running the extended *Testbase* on a number of systems are presented in the next section.

6 Measurements

the input's exact representation lies very near but not quite at half an ULP between two representable binary numbers[2]. The "Bits" column lists the number of bits beyond the 53 of IEEE double precision necessary to distinguish whether the value is exactly half an ULP between two representable values (in which case round-to-even should be used) or indeed below half an ULP, in which case the value should be rounded down in magnitude. The "Bits" value is computed as $\log_2 q/d$, where q is as given in Equation (4) and d is the modular distance of the given value from exactly one-half ULP.

The "worst" 17-digit decimal input is $78,459,735,791,271,921 \cdot 10^{+049}$. The first 119 bits of this number's binary representation are:

```
11101110011010000000010001001110000010011000101001110
01111111111111111111111111111111111111111111111111111
1111111111110 ...
```

Each row is 53 bits wide. We see that only if we compute this number to at least $53+13 = 66$ bits precision beyond that available in double precision can we determine that it does indeed lie below half an ULP rather than exactly at half an ULP between two representable binary numbers, and therefore that its value should be rounded down[3] rather than rounded according to round-to-even. Note that even a quadruple precision format with 64 extra bits does not suffice to correctly discern which rounding rule should be applied.

Table 2 lists the same information for values whose excess is just above half an ULP. Tables 3 and 4 list analogous values for converting from 53-bit IEEE binary to decimal. The "Bits" values for conversion to decimal are all quite similar. This is because it is the number of input digits that determines how closely one can approach a half-ULP excess in the output, and the number of input digits is unchanged regardless of the decimal significance.

Similar tables for 24-bit IEEE single precision and VAX 56-bit D format are given in the Appendix. Of particular note is that there exist 9-decimal-digit values that require 35 or more bits beyond the 24 of a single precision value to correctly convert to binary. These stress values are therefore out of range of double precision calculation, too, and therefore an implementation cannot rely on double precision sufficing for all single precision conversions.

[2]There may be other inputs that are equally "most difficult" for a given significance, but none which are closer but still below half an ULP between.

[3]Actually, in this case even if the excess is exactly half an ULP it will be rounded down, due to round-to-even.

Digits	Input	Bits
1	$6567258882077402 \cdot 2^{+952}$	62
2	$6712731423444934 \cdot 2^{+535}$	65
3	$6712731423444934 \cdot 2^{+534}$	63
4	$5298405411573037 \cdot 2^{-957}$	62
5	$5137311167659507 \cdot 2^{-144}$	61
6	$6722280709661868 \cdot 2^{+363}$	64
7	$5344436398034927 \cdot 2^{-169}$	61
8	$8369123604277281 \cdot 2^{-853}$	65
9	$8995822108487663 \cdot 2^{-780}$	63
10	$8942832835564782 \cdot 2^{-383}$	66
11	$8942832835564782 \cdot 2^{-384}$	64
12	$8942832835564782 \cdot 2^{-385}$	61
13	$6965949469487146 \cdot 2^{-249}$	67
14	$6965949469487146 \cdot 2^{-250}$	65
15	$6965949469487146 \cdot 2^{-251}$	63
16	$7487252720986826 \cdot 2^{+548}$	63
17	$5592117679628511 \cdot 2^{+164}$	65
18	$8887055249355788 \cdot 2^{+665}$	67
19	$6994187472632449 \cdot 2^{+690}$	64
20	$8797576579012143 \cdot 2^{+588}$	62
21	$7363326733505337 \cdot 2^{+272}$	61
22	$8549497411294502 \cdot 2^{-448}$	66

Table 4: Stress Inputs for Converting 53-bit Binary to Decimal, > 1/2 ULP

Digits	Input	Bits
1	$8511030020275656 \cdot 2^{-0342}$	63
2	$5201988407066741 \cdot 2^{-0824}$	63
3	$6406892948269899 \cdot 2^{+0237}$	62
4	$8431154198732492 \cdot 2^{+0072}$	61
5	$6475049196144587 \cdot 2^{+0099}$	64
6	$8274307542972842 \cdot 2^{+0726}$	64
7	$5381065484265332 \cdot 2^{-0456}$	64
8	$6761728585499734 \cdot 2^{-1057}$	64
9	$7976538478610756 \cdot 2^{+0376}$	67
10	$5982403858958067 \cdot 2^{+0377}$	63
11	$5536995190630837 \cdot 2^{+0093}$	63
12	$7225450889282194 \cdot 2^{+0710}$	66
13	$7225450889282194 \cdot 2^{+0709}$	64
14	$8703372741147379 \cdot 2^{+0117}$	66
15	$8944262675275217 \cdot 2^{-1001}$	63
16	$7459803696087692 \cdot 2^{-0707}$	63
17	$6080469016670379 \cdot 2^{-0381}$	62
18	$8385515147034757 \cdot 2^{+0721}$	64
19	$7514216811389786 \cdot 2^{-0828}$	64
20	$8397297803260511 \cdot 2^{-0345}$	64
21	$6733459239310543 \cdot 2^{+0202}$	63
22	$8091450587292794 \cdot 2^{-0473}$	63

Table 3: Stress Inputs for Converting 53-bit Binary to Decimal, $< 1/2$ ULP

Digits	Input	Bits
1	$9 \cdot 10^{-265}$	13
2	$85 \cdot 10^{-037}$	16
3	$623 \cdot 10^{+100}$	20
4	$3571 \cdot 10^{+263}$	24
5	$81661 \cdot 10^{+153}$	26
6	$920657 \cdot 10^{-023}$	30
7	$4603285 \cdot 10^{-024}$	30
8	$87575437 \cdot 10^{-309}$	37
9	$245540327 \cdot 10^{+122}$	42
10	$6138508175 \cdot 10^{+120}$	42
11	$83356057653 \cdot 10^{+193}$	45
12	$619534293513 \cdot 10^{+124}$	49
13	$2335141086879 \cdot 10^{+218}$	53
14	$36167929443327 \cdot 10^{-159}$	57
15	$609610927149051 \cdot 10^{-255}$	57
16	$3743626360493413 \cdot 10^{-165}$	63
17	$94080055902682397 \cdot 10^{-242}$	64
18	$899810892172646163 \cdot 10^{+283}$	69
19	$7120190517612959703 \cdot 10^{+120}$	73
20	$25188282901709339043 \cdot 10^{-252}$	73
21	$308984926168550152811 \cdot 10^{-052}$	77
22	$6372891218502368041059 \cdot 10^{+064}$	81

Table 2: Stress Inputs for Conversion to 53-bit Binary, $> 1/2$ ULP

Digits	Input	Bits
1	$5 \cdot 10^{+125}$	13
2	$69 \cdot 10^{+267}$	17
3	$999 \cdot 10^{-026}$	20
4	$7861 \cdot 10^{-034}$	21
5	$75569 \cdot 10^{-254}$	28
6	$928609 \cdot 10^{-261}$	30
7	$9210917 \cdot 10^{+080}$	31
8	$84863171 \cdot 10^{+114}$	34
9	$653777767 \cdot 10^{+273}$	40
10	$5232604057 \cdot 10^{-298}$	41
11	$27235667517 \cdot 10^{-109}$	45
12	$653532977297 \cdot 10^{-123}$	47
13	$3142213164987 \cdot 10^{-294}$	51
14	$46202199371337 \cdot 10^{-072}$	58
15	$231010996856685 \cdot 10^{-073}$	58
16	$9324754620109615 \cdot 10^{+212}$	61
17	$78459735791271921 \cdot 10^{+049}$	66
18	$272104041512242479 \cdot 10^{+200}$	72
19	$680260103780606 1975 \cdot 10^{+198}$	72
20	$20505426358836677347 \cdot 10^{-221}$	74
21	$836168422905420598437 \cdot 10^{-234}$	76
22	$4891559871276714924261 \cdot 10^{+222}$	86

Table 1: Stress Inputs for Conversion to 53-bit Binary, $< 1/2$ ULP

Proof. Consider the i produced at *Step 4* of the algorithm during iteration j. It has the property that $\langle i \rangle \leq d_j$.

Suppose $d_j/2 < \langle i \rangle \leq d_j$. Then $k_j \leftarrow 1$ at *Step 5*, and we will have

$$\begin{aligned} d_{j+1} = d_j \bmod \langle i \rangle &= d_j - \langle i \rangle \\ &< d_j/2. \end{aligned}$$

If on the other hand we have $\langle i \rangle \leq d_j/2$ then we have

$$\begin{aligned} d_{j+1} = d_j \bmod \langle i \rangle &< \langle i \rangle \\ &< d_j/2. \end{aligned}$$

Thus for the number of calls to *FirstModBelow* we have

$$m \leq \log c < \log b$$

and hence the total running time is $O(\log b)$.

5 Extending Testbase

We extended *Testbase* with "stress-testing" as follows. We added implementations of Algorithms 1 and 2, along with the necessary *bigint* support (primarily *bigint* multiplication and division and fast algorithms for computing powers of 2 and 10), entailing about 1,600 lines of additional *C* code.

To the types of test inputs listed in Section 2, we added a sixth. When doing this sixth type of testing, we pick an exponent within the output exponent range (for example, if testing decimal-to-binary for IEEE double precision, we would pick an exponent in the range -1023 through 1023), taking care to choose exponent not previously tested. We next compute the corresponding $e_{\min}$ and $e_{\max}$, as given by Equations (2) and (3). Then for each e_1 within this range we run Algorithm 1 and its complement to find the inputs closest to just below and just above a half-ULP between representable numbers in the output base.

Such stress-testing generates inputs requiring maximal internal precision for correct conversion. Table 1 lists the most difficult decimal-to-binary conversion inputs for 1–22 significant decimal digits. "Most difficult" means

Step 2: If $\langle q_n \rangle > c$, set $n \leftarrow n + 2$ and repeat.
Step 3: If $n = 1$ then set $i \leftarrow q_n$ and terminate.
Step 4: Set $n \leftarrow n - 2$. We then have $q_n < i \leq q_{n+2}$. Lemma 8 assures us that i has the form $q_n + jq_{n+1}$, where $0 < j \leq k$ for $k = (q_{n+2} - q_n)/q_{n+1}$.
Step 5: Set $d \leftarrow \langle q_n \rangle - c$. d is now the distance that we need to cover using j steps of q_{n+1} beyond q_n.
Step 6: Compute the step size s each step of q_{n+1} gives us. Recall that by Lemma 6, $\langle q_{n+1} \rangle$ is quite large (i.e., near b), since $n + 1$ is even. Set $s \leftarrow b - \langle q_{n+1} \rangle$.
Step 7: Set $j \leftarrow \lceil d/s \rceil$.
Step 8: Set $i \leftarrow q_n + jq_{n+1}$. Terminate.

Again, there is an analogous algorithm (using Lemma 9) for producing a positive integer i such that $\langle i \rangle \geq c$ and for all positive $j < i$, $\langle j \rangle < c$, which is needed by the analog of Algorithm 1. We omit it here.

We now turn our attention to the running-time of Algorithm 1.

Note that the only looping in Algorithm 2 occurs at *Step 2*. If the algorithm is being used in the context of Algorithm 1 then we know that for each subsequent use of Algorithm 2, c will be smaller than on the previous call. Hence we can initialize n in *Step 1* to whatever its final value was on the previous call, rather than 1. Each execution of Algorithm 2 will therefore either take constant time (if $\langle q_{n+2} \rangle \leq c$ for the initial value of n) or will "consume" one or more convergents, which will not be used again during the execution of Algorithm 1. Hence the total time spent executing Algorithm 2 will be at most proportional to the number of convergents of a/b. Knuth [Knu81] proves that the maximum number of convergents of a/b is $\approx 2.078 \log b + 1.672$.

Thus, over the course of an execution of Algorithm 1, if m calls are made to Algorithm 2, no more than $O(m) + O(\log b)$ time will be spent executing Algorithm 2.

Finally, we need to place an upper bound on m, which is done with the aid of the following Lemma.

Lemma 10 *When executing Algorithm 1, each iteration reduces the distance to the goal c by more than a factor of 2. I.e., if d_j is the value of d at* Step 2 *on iteration j of the algorithm, then $d_{j+1} < d_j/2$.*

Lemma 8 *Given a, b, an odd positive integer i, and a positive integer q_i, with q_i the denominator of the ith convergent of a/b, let $k = (q_{i+2} - q_i)/q_{i+1}$. Then for any non-negative integer $j < k$, the least $l > q_i + jq_{i+1}$ such that $\langle l \rangle < \langle q_i + jq_{i+1} \rangle$ is*

$$l = q_i + (j+1)q_{i+1}.$$

Proof. Let m be the least positive integer such that

$$\langle m \rangle \geq b - \langle q_i + jq_{i+1} \rangle. \tag{14}$$

We must have $m \leq q_{i+1}$, since Lemmas 4 and 6 assure us that

$$\langle q_{i+1} \rangle \geq b - \langle n \rangle$$

for all $n < q_{i+2}$, so surely q_{i+1} satisfies Equation (14). Now from the chain of inequalities given to us by Lemma 7 we have

$$\langle q_i + jq_{i+1} \rangle < \langle q_i \rangle$$

and therefore from Lemma 4 we have $m \geq q_{i+1}$, and hence $m = q_{i+1}$. We then apply the result of Lemma 1 with $c = b$ and the proof is complete.

There is an analogous Lemma for even i, whose proof we omit here.

Lemma 9 *Given a, b, an even positive integer i, and a positive integer q_i, with q_i the denominator of the ith convergent of a/b, let $k = (q_{i+2} - q_i)/q_{i+1}$. Then for any non-negative integer $j < k$, the least $l > q_i + jq_{i+1}$ such that $\langle l \rangle > \langle q_i + jq_{i+1} \rangle$ is*

$$l = q_i + (j+1)q_{i+1}.$$

We now are prepared to construct the *FirstModBelow* function referred to in Algorithm 1. Again, the algorithm is due to Tim Peters [Pet91].

Algorithm 2 (FirstModBelow) *Given a, b, a non-negative integer c, and a positive integer s, with $c < b$, produces a positive integer i such that $\langle i \rangle \leq c$ and for all positive $j < i$, $\langle j \rangle > c$.*

Step 1: Set $n \leftarrow 1$.

Lemma 6 *Given a, b, and positive integers i and q_i, with q_i the denominator of the ith convergent of a/b, then $\langle q_i \rangle \leq b/2$ if i is odd and $\langle q_i \rangle > b/2$ if i is even.*

Proof. Let $\epsilon = a/b - p_i/q_i$. As before, we have

$$aq_i - bp_i = \epsilon b q_i.$$

Suppose i is odd. Then from Lemma 5 and Equation (9) we have

$$\langle q_i \rangle = \epsilon b q_i \leq b/2$$

since $\epsilon > 0$. If i is even then instead we have

$$\langle q_i \rangle = b + \epsilon b q_i > b/2.$$

Lemma 7 *Given a, b, an odd positive integer i, and a positive integer q_i, with q_i the denominator of the ith convergent of a/b, let $k = (q_{i+2} - q_i)/q_{i+1}$. Then for any positive $j \leq k$*

$$\langle q_i + j q_{i+1} \rangle < \langle q_i + (j-1) q_{i+1} \rangle.$$

Proof. We know that $q_i + (j-1)q_{i+1} < q_{i+2}$ since $j \leq k$. But from Lemmas 4 and 6, $\langle q_{i+1} \rangle < b - \langle l \rangle$ implies $l \geq q_{i+2}$. Therefore we must have

$$\langle q_{i+1} \rangle > b - \langle q_i + (j-1) q_{i+1} \rangle.$$

We now construct the inequality

$$b > \langle q_{i+1} \rangle > b - \langle q_i + (j-1) q_{i+1} \rangle$$

and then we immediately attain the desired result by adding $\langle q_i + (j-1)q_{i+1} \rangle$ to the inequality and taking the result (mod b).

Now define δ as follows:

$$\delta = \begin{cases} \langle r \rangle / br, & \text{if } \langle r \rangle \leq b/2, \text{ and} \\ (b - \langle r \rangle)/br & \text{otherwise.} \end{cases} \tag{10}$$

It follows that

$$|\langle r \rangle|_b = |\delta br|.$$

Then Equation (7) gives us

$$\begin{aligned} |\delta br| &< |\epsilon b q_i| \\ |\delta r| &< |\epsilon q_i|. \end{aligned} \tag{11}$$

Let $\varphi = a/b$. From Equation (10) we have

$$ar \equiv \delta br \pmod{b}$$

and thus there exists some integer k such that

$$\begin{aligned} ar - kb &= \delta br \\ \varphi r - k &= \delta r. \end{aligned} \tag{12}$$

Also, Equation (8) gives us

$$\varphi q_i - p = \epsilon q. \tag{13}$$

Combining Equations (11), (12), and (13) then gives us

$$|\varphi r - k| < |\varphi q_i - p|$$

and thus by Theorem 3, we have

$$r \geq q_{i+1}.$$

Lemma 5 *Given a rational number φ and p_n/q_n, the convergents of its continued fraction representation, then for any $n \geq 0$ but less than the total number of convergents of φ, $p_n/q_n < \varphi$ if n is even and $p_n/q_n > \varphi$ if n is odd.*

Proof. This result immediately follows from Theorem 7.6 of [NZ80].

Definition 3 *For positive integers a and b, define*

$$|a|_b \equiv \min(a, b-a).$$

We will need the following simple property of modular distance.

Property 2 *For any positive integer $c < b$, we have*

$$|c|_b = \begin{cases} c, & \text{if } c \leq b/2, \text{ and} \\ b-c & \text{if } c > b/2. \end{cases}$$

Lemma 4 *Given a, b, and positive integers i, p_i, q_i, and r, with p_i/q_i the ith convergent of a/b, and*

$$|\langle r \rangle|_b < |\langle q_i \rangle|_b \tag{7}$$

then

$$r \geq q_{i+1}.$$

Proof. Let $\epsilon = a/b - p_i/q_i$. We have

$$\begin{aligned} aq_i - bp_i &= \epsilon b q_i \\ \langle q_i \rangle &= (\epsilon b q_i) \bmod b. \end{aligned} \tag{8}$$

By Theorem 1 we have $|\epsilon| < 1/(q_i q_{i+1}) < 1/{q_i}^2$, so $-b < \epsilon b q_i < b$. If $\epsilon \geq 0$ then

$$(\epsilon b q_i) \bmod b = \epsilon b q_i$$

and if $\epsilon < 0$

$$(\epsilon b q_i) \bmod b = b - \epsilon b q_i.$$

Since for any $i > 1$ by Lemma 3 we have $q_i \geq 2$, then by Theorem 1 and Property 2 we have for $i > 0$

$$|\epsilon b q_i| < b/2. \tag{9}$$

Therefore regardless of the sign of ϵ, we have

$$|\epsilon b q_i|_b = |\epsilon b q_i|.$$

But since $a < b$, $a \bmod b = a$ and we have

$$q_1 \quad = \quad \left\lfloor \frac{a}{b \bmod a} \right\rfloor$$

Trivially $b \bmod a < a$ and hence $q_1 \geq 1$, and therefore by Property 1, $q_2 \geq 2$.

The following theorems are adopted from [NZ80]; the proofs are omitted here, as the only changes we require are that the proofs be for a rational number φ instead of an irrational number ξ. This difference does not materially alter the proofs given by those authors.

Theorem 1 (Niven-Zuckerman 7.11) *Given a rational number φ and p_n/q_n, the convergents of its continued fraction representation, then for any $n \geq 0$,*

$$\left| \varphi - \frac{p_n}{q_n} \right| < \frac{1}{q_n q_{n+1}}$$

Theorem 2 (Niven-Zuckerman 7.12) *Given a rational number φ and its convergents p_n/q_n, then for all $n > 0$,*

$$\left| \varphi - \frac{p_n}{q_n} \right| < \left| \varphi - \frac{p_{n-1}}{q_{n-1}} \right| .$$

Theorem 3 (Niven-Zuckerman 7.13) *Given a rational number φ, and its convergents p_n/q_n, i an integer and j a positive integer, and for some integer $n > 0$*

$$|\varphi j - i| < |\varphi q_n - p_n|$$

then

$$j \geq q_{n+1}.$$

We now set out to use these theorems to prove results concerning how close $\langle q_i \rangle$ is to 0 or b for q_i the denominator of a convergent of a/b.

It is convenient to introduce a notion of modular "distance", analogous to absolute value.

where $i \geq -1$. Then define

$$\begin{aligned} p_{-2} &= 0, p_{-1} = 1 \\ q_{-2} &= 1, q_{-1} = 0 \\ p_i &= \left\lfloor \frac{a_{i-1}}{b_{i-1}} \right\rfloor p_{i-1} + p_{i-2} \\ q_i &= \left\lfloor \frac{a_{i-1}}{b_{i-1}} \right\rfloor q_{i-1} + q_{i-2} \end{aligned}$$

where $i \geq 0$. Then the ith convergent *of the continued fraction representation of a/b is p_i/q_i; p_i is the numerator of the ith convergent and q_i is the denominator.*

We state without proof the following basic property of the convergents of a/b.

Property 1 *Given a and b (conforming to the conventions given in Notation 1), there exists a least positive integer i' such that $b_{i'} = 0$, and that $p_{i'} = a$, $q_{i'} = b$. Also, for $0 < i \leq i'$, $q_{i-1} < q_i$.*

Throughout the remainder of this section we adopt the convention that when any variable $\mathcal{V}$ is used in the context of indexing the numerator or denominator of the convergents of a/b, then $\mathcal{V}$ is an integer and $0 < \mathcal{V} \leq i'$.

Lemma 3 *Given a and b, we have $q_2 \geq 2$ (and hence for $2 \leq i \leq i'$, $q_i \geq 2$).*

Proof. By definition we have

$$\begin{aligned} q_0 &= \left\lfloor \frac{a_{-1}}{b_{-1}} \right\rfloor q_{-1} + q_{-2} \\ &= 1 \\ q_1 &= \left\lfloor \frac{a_0}{b_0} \right\rfloor q_0 + q_{-1} \\ &= \left\lfloor \frac{b_{-1}}{a_{-1} \bmod b_{-1}} \right\rfloor \\ &= \left\lfloor \frac{(a \bmod b)}{b \bmod (a \bmod b)} \right\rfloor. \end{aligned}$$

Algorithm 1 (Modmin) *Given a, b, a non-negative integer c and positive integers $s_{\min}$ and $s_{\max}$, with $c < b$, $s_{\min} \leq s_{\max}$, produces an s subject to $s_{\min} \leq s \leq s_{\max}$, such that $\langle s \rangle \leq c$ and for all other s' with $s_{\min} \leq s' \leq s_{\max}$, either $\langle s' \rangle > c$ or $\langle s' \rangle < \langle s \rangle$.*

Step 1: Set $s \leftarrow s_{\min}$.
Step 2: Set $d \leftarrow c - \langle s \rangle$.
Step 3: If d is negative, set $d \leftarrow d + b$. If d is zero then minimal value is s; terminate.
Step 4: Set $i \leftarrow \mathit{FirstModBelow}(a, b, d)$.
Step 5: By Lemma 1, $s + i$ is now the least value $s' > s$ such that $c - \langle s' \rangle > c - \langle s \rangle$. We have $\langle s' \rangle = \langle s \rangle + \langle i \rangle$. If $\langle i \rangle < c - \langle s' \rangle$ then we already know that $s + 2i$ is the least $s'' > s$ such that $c - \langle s'' \rangle < c - \langle s \rangle$, and we can then repeat the process again if $\langle i \rangle < c - \langle s'' \rangle$. In general, we can add up to k multiples of i to s. Set $k \leftarrow \lfloor d / \langle i \rangle \rfloor$.
Step 6: If $s + ki \leq s_{\max}$ then set $s \leftarrow s + ki$ and go to *Step 2*. Otherwise set $k \leftarrow \lfloor (s_{\max} - s) / \langle i \rangle \rfloor$; set $s \leftarrow s + ki$; minimal value is s; terminate.

There is an analogous algorithm (based on Lemma 2), which we omit here, for producing an s subject to $s_{\min} \leq s \leq s_{\max}$, such that $\langle s \rangle \geq c$ and for all other s' with $s_{\min} \leq s' \leq s_{\max}$, either $\langle s' \rangle < c$ or $\langle s' \rangle > \langle s \rangle$.

We now turn to the problem of an algorithm for computing the *FirstModBelow* function. To do so we need to develop a fair amount of theory, much of it based upon the convergents of the continued fraction representation of a/b, as the denominators q_i of these convergents also yield especially small and large values of $\langle q_i \rangle$.

The convergents of the continued fraction representation of a/b can be defined as follows.

Definition 2 *Given a and b, define*

$$\begin{aligned} a_{-2} &= a \\ b_{-2} &= b \\ a_i &= b_{i-1} \\ b_i &= a_{i-1} \bmod b_{i-1} \end{aligned}$$

Lemma 2 *Given a, b, and positive integers c and s, with $s < b$, and $\langle s \rangle > c \geq 0$, let i be the least positive number such that*

$$\langle i \rangle \geq (b + c) - \langle s \rangle.$$

Then for any j such that $0 < j < i$, either

$$\langle s + j \rangle \quad > \quad \langle s \rangle$$

or

$$\langle s + j \rangle \quad < \quad c.$$

Proof. Clearly we have

$$\begin{aligned} (b + c) - \langle s \rangle \quad &> \quad \langle j \rangle \quad > \quad 0 \\ b + c \quad &> \quad \langle j \rangle + \langle s \rangle \quad > \quad \langle s \rangle \end{aligned}$$

($\langle j \rangle \neq 0$ since $0 < j < i \leq b$ and $\gcd(a, b) = 1$.) If $\langle s \rangle + \langle j \rangle \geq b$ then

$$c > (\langle s \rangle + \langle j \rangle) \bmod b$$

and therefore by Equation (6)

$$\langle s + j \rangle < c.$$

If $\langle s \rangle + \langle j \rangle < b$ then again by Equation (6)

$$\langle s + j \rangle > \langle s \rangle$$

and the proof is complete.

Assuming we have a function

$$\mathit{FirstModBelow}(a, b, c) = \text{least } i \text{ such that } \langle i \rangle \leq c$$

we now can construct an algorithm for minimizing $\langle s \rangle$ over $s_{\min} \leq s \leq s_{\max}$. This algorithm is due to Tim Peters [Pet91].

Lemma 1 *Given a, b, and positive integers c and s, with $s < b$, and $\langle s \rangle < c \leq b$, let i be the least positive number such that*

$$\langle i \rangle \leq c - \langle s \rangle.$$

Then for any integer j such that $0 < j < i$, either

$$\langle s + j \rangle \;\; < \;\; \langle s \rangle$$

or

$$\langle s + j \rangle \;\; > \;\; c.$$

Proof. Clearly we have

$$b > \langle j \rangle > c - \langle s \rangle. \tag{5}$$

But

$$\begin{aligned} \langle s + j \rangle &= (a \cdot s + a \cdot j) \bmod b \\ &= (\langle s \rangle + \langle j \rangle) \bmod b. \end{aligned} \tag{6}$$

From Equation (5) we have

$$b + \langle s \rangle > \langle s \rangle + \langle j \rangle > c.$$

If $\langle s \rangle + \langle j \rangle \geq b$ then

$$\langle s \rangle > (\langle s \rangle + \langle j \rangle) \bmod b$$

and therefore

$$\langle s + j \rangle < \langle s \rangle.$$

If $c < \langle s \rangle + \langle j \rangle < b$ then

$$\langle s + j \rangle > c.$$

A similar proof then gives the following complementary Lemma.

where p and q are integers.

When converting a from base b_1 to base b_2 with d_2 significant digits, only the integer part of r' can be represented exactly. The fractional part is the fraction of one unit in the last place by which a is not exactly representable in base b_2; this fractional part is equal to $((j \cdot p) \bmod q)/q$.

Thus, to find a base b_1 number which is not exactly representable in base b_2 by a fraction k/q units in the last place, we must minimize in a modular sense $(j \cdot p - k) \bmod q$. In particular, if we want to find a base b_1 number with d_1 significant digits as close as possible to exactly half way between two representable b_2 numbers, we need to find j such that

$$(j \cdot p - \lfloor q/2 \rfloor) \bmod q \tag{4}$$

is minimal, subject to the constraint $e_{\min} \leq e_1 \leq e_{\max}$.

The corresponding base b_1 number will thus be exceptionally difficult to correct convert to base b_2, since high internal precision will be required to determine whether b_1 is just below, just above, or exactly at half an ULP between two representable base b_2 numbers.

We now turn to the problem of efficiently solving such a modular minimization problem.

4 Modular Minimization with Constraints

Many of the lemmas and algorithms in this section concern the value of $(a \cdot i) \bmod b$ for particular positive integers a, b with $\gcd(a, b) = 1$, and another positive integer i. We introduce the following notations:

Notation 1 *Throughout this section, the variables a and b will stand for positive integers with $a < b$ and $\gcd(a, b) = 1$.*

Definition 1 *Given a, b, and a positive integer i, define*

$$\langle i \rangle \equiv (a \cdot i) \bmod b.$$

$$\begin{aligned} e_2 \log b_2 &\leq (e_{\min} + 1) \log b_1 \\ \frac{e_2 \log b_2}{\log b_1 - 1} &\leq e_{\min} \end{aligned}$$

As $e_{\min}$ is an integer, we subsequently have:

$$emin = \left\lceil \frac{e_2 \log b_2}{\log b_1 - 1} \right\rceil \tag{2}$$

Similarly, we have:

$$emax = \left\lfloor \frac{(e_2 + 1) \log b_2}{\log b_1} \right\rfloor \tag{3}$$

Consider now the conversion of a number of the form $a = m{b_1}^{e_1}$, where $e_{\min} \leq e_1 \leq e_{\max}$. Since a has d_1 significant digits we have

$$a = j{b_1}^{-(d_1 - 1)}$$

for some integer j.

The exact representation of a in base b_2 can be expressed as a rational number r, with $1 \leq r < b_2$:

$$r = \frac{j{b_1}^{e_1 - d_1 + 1}}{{b_2}^{e_2}}$$

Let $r' = r{b_2}^{d_2 - 1}$. Then the constraint on the range of r then gives

$${b_2}^{(d_2 - 1)} \leq r' = \frac{j{b_1}^{(e_1 - d_1 + 1)}}{{b_2}^{(e_2 - d_2 + 1)}} < {b_2}^{d_2}$$

Let $f_1 = \max(-(e_1 - d_1 + 1), 0)$ and $f_2 = \max(-(e_2 - d_2 + 1), 0)$. Then we have

$$r' = \frac{{b_1}^{(e_1 - d_1 + 1 + f_1)} {b_2}^{f_2}}{{b_2}^{(e_2 - d_2 + 1 + f_2)} {b_1}^{f_1}} j$$

We now can write r' as the product of j and a fraction p/q:

$$\begin{aligned} r' &= j \cdot p/q \\ p &= {b_1}^{(e_1 - d_1 + 1 + f_1)} {b_2}^{f_2} \\ q &= {b_2}^{(e_2 - d_2 + 1 + f_2)} {b_1}^{f_1} \end{aligned}$$

When comparing u_1 with u_2 *Testbase* is capable of evaluating a wide variety of rounding-modes as well as IEEE halfway-rounded-to-even. Of relevance for this paper are its capabilities of also evaluating rounding-to-within-one-ULP and "biased" rounding, namely that used by VAX hardware.

A final level of testing done by *Testbase* is to check *monotonicity*. The IEEE standard requires that if an input value is increased, its converted representation must not decrease. *Testbase* tests for conformance by perturbing each of its input values by ± 1 and ± 2 ULP's (in the input base) and confirming that the converted values maintain monotonicity.

3 Difficult Conversions

While *Testbase* tests a rather large battery of different conversions, its reliance on random operands is a little unsatisfying, as perhaps there are quite rare operands that are exceptionally difficult to correctly convert. *Testbase* might never generate such an operand, and thus give an impression of correctness when such is not truly warranted. This section develops how for any particular output range (given exponent and number of significant digits) there exist "worst-case" input values that lie exceedingly close to exactly one half-ULP between two representable output values.

Consider the task of converting a positive base b_1 number with d_1 digits to the closest base b_2 number with d_2 digits and exponent e_2. The representable base b_2 numbers in this range are:

$$b_2{}^{e_2} \leq n b_2{}^{e_2} < b_2{}^{e_2+1} \tag{1}$$

where n has the form $n_0.n_1 n_2 \cdots n_{d_2-1}$, $0 \leq n_i < b_2$ for $0 < i < d_2$, and $0 < n_0 < b_2$. The base b_1 numbers must therefore lie in this range. In general, base b_1 numbers have the form $m b_1{}^{e_1}$, where m has the form $m_0.m_1 m_2 \cdots m_{d_1-1}$, $0 \leq m_i < b_1$ for $0 < i < d_1$, and $0 < m_0 < b_1$. For a base b_1 number to lie in the range given in equation 1 it must satisfy

$$b_2{}^{e_2} \leq m b_1{}^{e_1} < b_2{}^{e_2+1}.$$

From this equation we can derive $e_{\min}$ and $e_{\max}$, the minimum and maximum values for e_1:

$$b_2{}^{e_2} \quad \leq \quad m b_1{}^{e_{\min}} < b_1{}^{e_{\min}+1}$$

- For random targets $\pm$ half an ULP, a random floating-point number is constructed as above. It is then converted to *unpacked* format and its half-ULP value computed exactly. Both of these values are then converted to decimal strings, added exactly, and the corresponding *decsig* digits kept.

Once the input has been constructed, it is converted exactly to u_1, its corresponding *unpacked* form, and also converted to binary using the native conversion routine. The binary result is then unpacked exactly to u_2. u_1 and u_2 are now exactly converted to corresponding *bigint*'s b_1 and b_2 as follows. If u_1's exponent is negative then u_2's significand is scaled up by u_1's base raised to the negative of u_1's exponent. u_1's exponent is then set to zero. Similarly, u_1's significand is scaled up if u_2's exponent is negative and u_2's exponent set to zero. If either u_1 or u_2 at this point still have positive exponents, their significands are scaled up by their base raised to their exponent. At this point, b_1 and b_2 correspond to the scaled significands of u_1 and u_2. These values can then be compared exactly to determine by how many half-ULP's they differ. As u_1 was computed exactly to begin with, the half-ULP difference is the degree to which the binary representation resulting from the native conversion differs from the true, infinitely precise binary representation.

Testing binary-to-decimal conversion is similar. The input values are generated as follows:

- For powers of 2, an *unpacked* value is created to represent the power of 2. It is then converted exactly to the native representation.
- For powers of 10 and random targets, an E- or F-format *decsig*-digit decimal string is constructed and then converted to binary using the native conversion.
- For random sources, a random floating-point value is constructed as described above.
- For random targets $\pm$ half an ULP, an E- or F-format decimal string is constructed and half a decimal ULP added. This value is then converted to binary using the native conversion.

The input is then converted to an exact *unpacked* form u_1 and a converted *unpacked* form u_2 and the results compared in a manner analogous to that described above.

- random *sources*, where for decimal-to-binary conversion a random source is a random decimal string, and for binary-to-decimal conversion it is a random representable floating-point value;

- random *targets*, namely random representable floating-point values for decimal-to-binary conversion and random decimal strings for binary-to-decimal conversion; and

- random targets $\pm$ half an ULP in the target representation.

Testbase does all internal computations using its own library of routines for manipulating arbitrarily large integer values (*bigint*'s). A floating-point number is then exactly representable as a *bigint* "significand" times the quantity of a "base" (either 2 or 10) raised to some integer power. Such a representation is referred to as *unpacked*. Each system to be tested using *Testbase* must provide routines to correctly convert between its native floating-point representation and the *unpacked* representation. This conversion entails packing or unpacking the contents of a native floating-point value, and thus can be performed exactly[1].

In the discussion that follows, *decsig* refers to the number of significant decimal digits being tested.

When testing decimal-to-binary conversion, *Testbase* generates its input values as follows:

- For powers of 2, an *unpacked* value is created to represent the power of 2. It is then converted exactly to the native representation, and then converted (perhaps only approximately) to a *decsig*-digit decimal string using the native conversion routine.

- For powers of 10 and random sources, an E- or F-format *decsig*-digit decimal string is constructed directly.

- For random targets, a random floating-point number is constructed (using random mantissa and exponent), which is then converted using the native conversion routine to a *decsig*-digit decimal string.

[1]Note that arbitrary decimal digit strings can also be converted exactly to an *unpacked* representation.

be chosen. If the second, then the smaller. If the third, then the IEEE halfway-rounded-to-even rule should be applied.

The IEEE standard essentially only requires that the converted value be accurate to within an ULP [C+84], rather than a half ULP correctly rounded, which would be fully correct. However algorithms are publicly available for fully correct conversion [Gay90], and thus we are interested in developing a program for testing whether a given conversion library is fully correct, IEEE-conformant, or incorrect.

In the next section we outline David Hough's *Testbase* program for testing decimal–binary conversion [Hou91]. In the following section we derive a modular equation which if minimized produces especially difficult conversion inputs; those that lie as close as possible to exactly half way between two representable outputs. We then develop the theoretical framework for demonstrating the correctness of two algorithms developed by Tim Peters for solving such a modular minimization problem in $O(\log(N))$ time. We next discussed how we extended *Testbase* to use these algorithms for generating *stress-testing*, namely testing of especially difficult inputs. We also list a number of "worst-case" inputs for varying degrees of decimal digit significance. In the penultimate section we present measurements made on a number of different computer systems to evaluate the quality of their decimal–binary conversion; the final section summarizes the results.

2 Testbase

Testbase is a 4,800 line *C* program developed by David Hough of Sun Microsystems to test binary-to-decimal and decimal-to-binary conversion for any given number of significant decimal digits and for a wide variety of IEEE floating-point representations. *Testbase* tests both conversion of E-format decimals (those with an exponent field) and F-format (those represented solely by a string of digits and perhaps a decimal point). In what follows, the "native conversion" refers to the *C sscanf* and *sprintf* conversion routines provided by a system for decimal–binary conversion.

Testbase tests the conversion of five types of inputs:

- positive and negative powers of 2;
- positive and negative powers of 10;

A Program for Testing IEEE Decimal–Binary Conversion

Vern Paxson
CS 279
Prof. Kahan
May 22, 1991

1 Introduction

Regardless of how accurately a computer performs floating-point operations, if the data to operate on must be initially converted from the decimal-based representation used by humans into the internal representation used by the machine, then errors in that conversion will irrevocably pollute the results of subsequent computations. Similarly, if internal numbers are not correctly converted to their decimal equivalents for output display, again the computational results will be tainted with error, even if all internal operations are performed exactly.

In this paper we concern ourselves with the problem of correctly converting between decimal numbers and IEEE single and double precision values. At first blush the conversion problem might appear simple and therefore uninteresting, but neither is the case. Indeed, [Coo84] devotes 49 pages to a thorough treatment of the topic. See also [Cli90] and [JW90] for discussions of correct decimal-to-binary and binary-to-decimal conversion, respectively.

Part of the difficulty is that numbers in the input base (be it 2 or 10) can lie extremely close to exactly half way between adjacent representable numbers in the output base. In order to produce the closest possible representation of the input in the output base, a conversion algorithm must distinguish these cases as to whether they lie above, below, or exactly at half a unit in the last place (ULP) in their output base representation. If the first, then the larger of the adjacent output base representations should