

Performances d'implantations de l'addition en précision quad-double sur différentes machines*

Sylvie Boldo, Marc Daumas

Sylvie.Boldo@ENS-Lyon.Fr et Marc.Daumas@ENS-Lyon.Fr
Laboratoire de l'Informatique du Parallélisme
UMR 5668 CNRS - INRIA - ENS de Lyon

Résumé

Les bancs d'essais (*benchmarks*) ont été créés pour comparer les nombreux processeurs existants. Ils ont eu des effets pervers : d'une part, les machines excellent toutes aux benchmarks, ce qui les rend de plus en plus semblables ; d'autre part, les applications d'un genre n'appartenant pas aux benchmarks risquent d'être négligées et d'avoir des performances amoindries. L'application étudiée ici est originale car extrêmement régulière : beaucoup d'additions flottantes, du calcul entier et des branchements conditionnels, rien d'autre. Classer les implantations est toutefois loin d'être évident.

Mots-clés : processeurs superscalaires, addition en précision quad-double.

1. Introduction

Fabriquer un microprocesseur est une œuvre de pragmatisme. Les livres de HENNESSY et PATTERSON [14, 15] illustrent parfaitement cet état de fait. Les concepteurs privilégient avec des importances relatives la vitesse d'horloge, le temps d'exécution des benchmarks et le temps d'exécution des tâches usuelles. Le calcul scientifique, bien que fortement minoritaire en investissement et en temps de cycle, reste bien représenté dans l'éventail des tâches usuelles. La nouvelle norme SPEC 2000 réunit 14 programmes flottants au lieu des 8 présents dans la norme 1995 [16, 23]. Ces programmes sont variés et couvrent des domaines très différents de l'expérimentation et du calcul numérique mais aucun d'entre eux ne présente la régularité des programmes que nous allons étudier ici. Cela explique probablement les réactions très différentes de processeurs qui semblent très comparables à première approche.

En faisant les choix qui sont les leurs, les concepteurs de circuits essaient de favoriser le plus grand nombre d'utilisateurs. Il est important de vérifier qu'ils ne le font pas au détriment trop grave des utilisateurs hors-normes.

Nous travaillons au sein du projet Arénaire en collaboration avec l'*University of California at Berkeley* et le *Lawrence Berkeley Laboratory* à la mise en œuvre d'une bibliothèque pour le calcul en précision *quad-double* [17]. Il s'agit d'un code de calcul où un nombre très précis est représenté à l'aide de quatre composantes qui sont des nombres normalisés IEEE en double précision [30]. Cette structure de données est une instantiation d'un type plus vaste, les *expansions*, défini et utilisé entre autres dans [26, 29, 7]. La bibliothèque *quad-double* fait suite aux travaux de BAILEY et de BRIGGS sur la précision *double-double*².

Au niveau algorithmique deux choix sont ouverts aboutissant à quatre implantations distinctes. Le premier choix est de savoir si l'on trie les données en fusionnant les listes de composantes des deux nombres *quad-double* avant tout traitement numérique. On obtient alors la meilleure complexité asymptotique, mais un tri nécessite de nombreux branchements et chaque erreur de prédiction ajoute un coût considérable au temps d'exécution³. Le second choix est de savoir si l'on adapte l'algorithme

* Ce travail a été en partie financé par une bourse du Fonds France-Berkeley.

2. Ces deux auteurs distribuent indépendamment leur bibliothèque par le réseau INTERNET aux adresses <http://www.nersc.gov/~dhh/mpdist/mpdist.html> et <http://www-epidem.plantsci.cam.ac.uk/~kbriggs/doubledouble.html>.

3. Nous invitons le lecteur qui n'est pas familier avec la prédiction de branchement et les architectures superscalaires à se reporter aux transparents du cours de Pascal Sainrat présenté à l'École sur la Conception d'Architecture de Systèmes Informatiques dédiés à des applications spécifiques de type 'enfoui'. On peut les trouver sur le réseau INTERNET à l'adresse http://www.ens-lyon.fr/LIP/Ecoles_Archi/sainrat.pdf.

exact pour ne générer qu'un nombre limité d'éléments du résultat. En toute généralité, la somme de deux nombres stockés sur quatre mots chacun peut occuper jusqu'à huit mots, mais dans la pratique, certaines de ces composantes sont nulles. En précision *quad-double*, nous ne conservons pas les quatre mots de poids inférieur. Là encore, la meilleure solution asymptotique consiste à générer uniquement les quatre premières composantes non nulles.

En précision *double-double*, l'écart entre la solution asymptotiquement optimale et la solution sans branchement est trop faible pour équilibrer le coût des mauvaises prédictions de branchement. Le coût de la génération des deux derniers mots de poids faible est lui aussi trop faible pour valoriser un test d'arrêt précoce. La précision *quad-double* est intéressante parce qu'elle favorise davantage les solutions asymptotiquement efficaces et un arrêt précoce du calcul. Nous présentons notre programme et son environnement dans la section 2. La section 3 présente les tests et leurs conditions. Nous terminons ce papier par quelques premières conclusions.

2. Expansions

Les processeurs actuels disposent à la fois d'une ou plusieurs unités de calcul entier souvent appelées Unités Arithmétique et Logique (en anglais *ALU*) et d'une ou plusieurs unités de calcul flottant appelées Unités Flottantes (en anglais *FPU*). Pour effectuer un calcul en précision multiple, on utilise naturellement la numération de position sur la base 2^{16} ou 2^{32} et l'*ALU* [8]. Ces bibliothèques sont bien adaptées aux cas où l'on désire travailler sur de gros codes qui manipulent des nombres extrêmement précis. La bibliothèque GMP [12] est un exemple de développement de ce type.

Dans le cas où on désire juste doubler ou quadrupler la précision d'un code numérique⁴, il est plus avantageux d'utiliser la *FPU*. Il n'est alors plus utile de convertir les nombres flottants en nombres entiers et les raffinements algorithmiques mis en œuvre dans les bibliothèques de calcul en précision arbitraire tels que la moyenne arithmético-géométrique [4] ne sont pas efficaces pour de si petites précisions.

Par ailleurs, on constate que les constructeurs investissent beaucoup dans le développement de la *FPU* ce qui en fait une unité plus puissante que l'*ALU*. Sur les photographies de l'Alpha 21264 publiées dans [20], les *FPU*s occupent sensiblement la même place que les cluster 0 et 1 d'*ALU*. Digital puis Compaq ont imposé et maintenu une avance pour le processeur Alpha qui ne s'est jamais démentie dans le domaine du calcul flottant [13].

Il existe deux techniques de calcul en précision multiple sur *FPU*. La plus simple consiste à utiliser la *FPU* comme une *ALU*. On traite des entiers suffisamment petits⁵ pour qu'il n'y ait jamais d'erreur d'arrondi [1]. Une solution plus récente est appelée développement en virgule flottante, ou un peu improprement *expansion*.

Une expansion x de taille n est simplement une liste ordonnée de n composantes qui sont des nombres flottants usuels. La valeur représentée par la structure de données x est la somme exacte sans arrondi de ses composantes x_0, x_1, \dots, x_{n-1} . Chaque x_i est un nombre machine, ici un flottant IEEE-754 double précision [10, 11]. On peut ainsi représenter un nombre avec plus de précision tout en utilisant les structures de données et les propriétés sur l'arrondi définies par la norme IEEE-754 ainsi que la rapidité de la *FPU*.

Pour décrire certaines de nos hypothèses et nos jeux de test, nous représentons les nombres flottants par des rectangles où seule apparaît la mantisse. Les positions relatives des rectangles donnent les différences entre les exposants. Par exemple, le schéma suivant représente cinq nombres flottants. Leur mantisse est stockée sur treize bits. Les exposants qui ne sont pas donnés dans le schéma sont tels que le premier nombre est beaucoup plus grand que le second, qui est beaucoup plus grand que le troisième, qui est lui-même beaucoup plus grand que le quatrième, et ainsi de suite.

1.000100000001
 1.11111111001
 1.00000001111
 1.101110000000
 1.100000000000

Si on note e_1, e_2, e_3, e_4 et e_5 les exposants des cinq nombres représentés, on lit que $e_1 \approx e_2 + 20$, $e_2 \approx e_3 + 15$, $e_3 \approx e_4 + 15$ et enfin $e_4 \approx e_5 + 18$.

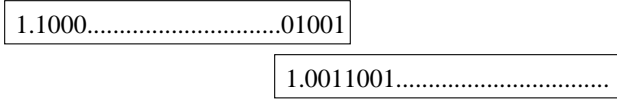
Si on omet les cas de dépassement de capacité vers l'infiniment petit ou l'infiniment grand, les algorithmes d'addition que nous allons étudier sont indépendants de l'échelle. On peut multiplier par une puissance de 2 tous les éléments et retrouver le même problème. Cela explique pourquoi il n'est

4. On trouvera dans [3] une expérience de doublement de la précision pour le calcul scientifique dans le cas où on cherche à étudier deux phénomènes numériquement stables mais d'échelles très différentes.

5. Une autre bibliothèque de Bailey appelée MPFUN utilise cette technique avec 2^{22} comme valeur maximale des entiers utilisés dans la représentation interne.

pas nécessaire de connaître précisément la valeur de l'exposant. Seules les valeurs des différences entre les exposants sont importantes. Comme dans le cas des produits partiels d'une multiplication, on aligne verticalement les chiffres de même poids. On retrouve cette notation par exemple dans [19, 24, 31].

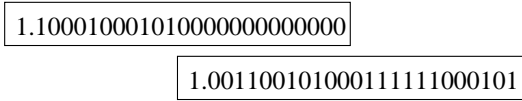
Le format *quad-double* ne stocke que quatre composantes au format double. Pour obtenir le maximum de précision possible, soit environ 200 bits, on évite que les composantes se chevauchent comme dans l'exemple qui suit.



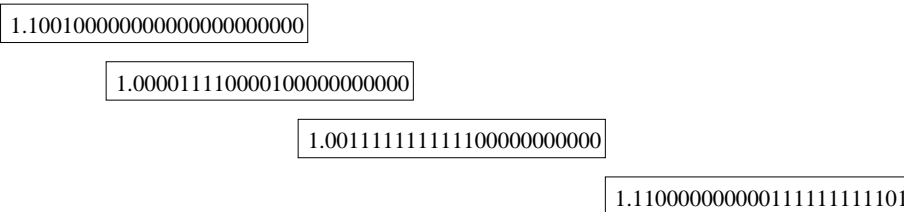
Plus précisément, on dit que deux flottants x et y ne se chevauchent pas s'il existe deux entiers r et s tels que $x = r \times 2^s$ et $|y| < 2^s$.

Une condition plus forte exige que les composantes ne soient pas adjacentes ce qui signifie qu'il y a un espace représentant un bit à zéro entre deux composantes dans nos schémas. Cette condition est toujours possible: avec une mantisse de 4 bits, on a ainsi $11101001 = 1110 \times 2^4 + 1001$ mais aussi $11101001 = 1111 \times 2^4 - 111$ où les composantes sont non adjacentes. Si on veut définir cette condition comme l'absence de chevauchement, cela signifie que l'on connaît deux entiers r et s tels que $x = r \times 2^s$ et $|y| < 2^s/2$

Avec la définition que nous venons de donner, les deux nombres suivants ne sont pas adjacents, mais on sent que l'on perd beaucoup de précision dans le chevauchement des bits non significatifs à 0 de la première composante.



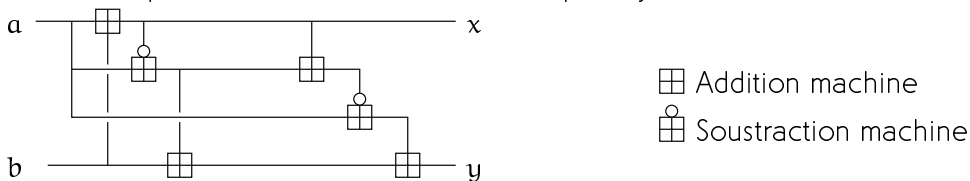
Dans les faits, le résultat d'un calcul ressemble souvent à l'expansion suivante.



On voit que les notions de non adjacence et de non chevauchement ne sont pas très adaptées au cas d'une expansion finie. On essaiera également de forcer une composante du résultat à être de l'ordre de l'ulp de la composante précédente. Le terme *ulp* est un acronyme signifiant *unit in the last place*. Dans les faits, $ulp(x) = 2^{-52} \times 2^{\text{exposant}(x)}$.

Les composantes de l'expansion précédente ne se chevauchent pas, mais le résultat pourrait être représenté avec moins de composantes pour gagner en place ou en précision. Les techniques de compression usuelles telles que celle mise en œuvre par SHEWCHUK [29] ou PRIEST [26] marchent bien mais travaillent sur la totalité du résultat produit, soit huit composantes dans notre cas. Là encore, on voudrait s'arrêter avant de traiter des composantes de poids trop faible qui n'auront peut-être que peu d'influence sur les quatre premières composantes.

Certaines routines simples sont à la base des algorithmes qui suivent. Elles sont dues à DEKKER [9], KNUTH [21], PRIEST [27] et SHEWCHUK [29]. On peut calculer par exemple la somme exacte de deux flottants x et y tels que $a + b = x + y$ et x et y ne sont pas adjacents.



On en déduit des algorithmes pour calculer la somme de deux expansions. Nous invitons le lecteur désireux de parcourir ces algorithmes à télécharger le rapport très pédagogique de SHEWCHUK sur ce sujet⁶.

Ces algorithmes ne fonctionnent que dans le cas d'une machine conforme à la norme IEEE-754 et dans quelques autres cas bien isolés. Sur les Pentiums, il faut désactiver l'usage des 64 bits de mantisse des registres de *double-extended* de 80 bits. Si tous les calculs se font à cette précision, l'algorithme qui additionne exactement deux réels en renvoyant le résultat et l'erreur commise ne fonctionne plus. On oblige l'ordinateur à arrondir la mantisse en précision usuelle IEEE à chaque opération de façon à être certain de la justesse du résultat.

3. Comparatif

Nous présentons à la Figure 1 les tests des algorithmes suivants qui réalisent tous une addition en *quad-double*:

Le programme `a11` fait simplement l'addition des deux expansions de taille 4 pour obtenir une expansion non adjacente de taille 8. On garde à la fin les 4 composantes les plus significatives. Cet algorithme est un peu brutal car il ne tient pas compte du fait que l'on ne veut que 4 composantes à la fin mais on est sûr d'être très proche du résultat exact.

Le programme `a112` utilise le même algorithme. Toutes les boucles sont déroulées statiquement à la compilation en utilisant le fait que les données ont exactement quatre composantes.

L'algorithme `branchless` (nous utiliserons `br1` dans le tableau) essaie d'éviter autant que possible les sauts conditionnels qui nuisent au bon fonctionnement du pipeline. Cet algorithme ne trie pas les données. Il n'utilise des sauts que dans la partie qui renormalise le résultat. Si l'étape de renormalisation fonctionne, le résultat obtenu est presque toujours proche du résultat exact.

L'addition `1s` trie les composantes des deux *quad-doubles* en une seule expansion triée mais avec des chevauchements puis les ajoute de la plus grande à la plus petite jusqu'à en avoir 4.

L'addition `s1` trie puis ajoute de la plus petite à la plus grande les 5 composantes les plus significatives. Si ce résultat intermédiaire peut-être stocké sur moins de 4 composantes, on rajoute les composantes des entrées une à une.

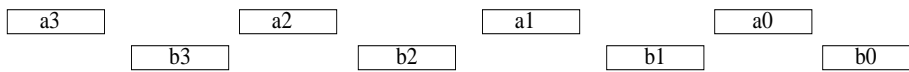
L'addition `d1s` trie les composantes des deux *quad-doubles*. On ajoute alors les composantes de la plus grande à la plus petite en utilisant un accumulateur de taille 2. On s'arrête dès qu'on a obtenu quatre composantes. Le résultat est proche du résultat exact.

L'addition `comp` consiste à utiliser l'algorithme de compression de SHEWCHUK [29]. On trie les composantes, on les compresse et on retourne les quatre composantes les plus significatives.

⁶. L'adresse est <http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-arithmetic.ps>.

Nous faisons les tests sur des ensemble de données *quad-doubles* différents comme on peut le voir dans les schémas qui suivent.

Set1 :



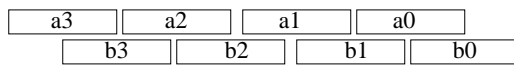
Set2 :



Set3 :



Set4 :



Le Set1 privilégie les cas où on teste souvent de façon à limiter les calculs, tandis que le Set3 nécessite l'ajout de toutes les composantes. Les tests d'arrêt du calcul sont alors inutiles.

Les codes de la Figure 1 correspondent aux machines suivantes provenant d'un spectre assez large [28]:

- PS Pentium II à 400MHz sous Solaris [18]
- P3 Pentium III à 550 MHz sous Linux
- K6 AMD-K6 à 300MHz sous Linux
- SS Sparc V9 à 360MHz sous Solaris [32]
- M Motorola Power PC 604e à 190MHz sous Linux [25]
- I IBM SP, POWER3 200 MHz
- C CRAY T3E, DEC Alpha 450 MHz [6]

Nous en déduisons le classement récapitulatif suivant des implantations selon leur rapidité au Set3, qui devrait être le plus fréquent. Il correspond également aux cas possibles de *cancellation*. Le système d'exploitation ne semble pas jouer un rôle prépondérant sur les vitesses réciproques des implantations.

PS	P3	K6	SS	M	I	C
comp	comp	comp	brl	brl	brl	all2
brl	brl	brl	comp	all2	ls	brl
ls	ls	ls	all2	comp	all2	comp
all2	all2	all2	ls	s1	comp	dls
s1	s1	dls	s1	ls	s1	all
dls	dls	s1	all	all	dls	s1
all	all	all	dls	dls	all	ls

Set1

Machine	PS	P3	K6	SS	M	I	C
comp	0.58	0.77	3.23	1.21	1.78	1.92	0.98
branchless	1.04	0.84	3.17	1.00	1.14	1.37	0.88
all2	0.77	1.34	5.28	1.52	2.21	0.98	0.86
dls	0.81	0.65	2.71	0.89	1.73	1.53	0.98
ls	1.05	0.76	2.88	1.12	1.86	0.98	0.84
sl	0.69	0.85	2.87	1.20	1.73	1.20	1.35
all	1.82	1.52	6.05	2.28	2.92	2.68	1.75

Set2

Machine	PS	P3	K6	SS	M	I	C
comp	0.87	0.83	2.68	1.20	1.89	1.79	1.05
branchless	1.11	0.82	3.07	0.82	1.12	1.37	0.86
all2	1.18	1.15	4.21	1.12	1.84	1.43	0.65
dls	1.56	1.29	4.23	1.30	2.62	2.09	1.46
ls	1.09	1.10	3.48	1.44	2.58	1.42	FPE
sl	1.33	1.25	3.81	1.39	2.28	1.71	1.77
all	1.67	1.30	4.55	1.54	2.47	2.30	1.45

Set3

Machine	PS	P3	K6	SS	M	I	C
comp	0.80	0.79	2.50	1.00	1.72	1.70	0.91
branchless	1.12	0.85	3.02	0.61	1.16	1.36	0.86
all2	1.14	1.16	3.97	1.21	1.85	1.45	0.63
dls	1.43	1.25	4.05	1.71	2.67	2.01	1.43
ls	1.02	1.09	3.32	1.22	2.50	1.44	FPE
sl	1.25	1.22	4.23	1.48	2.34	1.71	1.72
all	1.59	1.33	4.77	1.52	2.52	2.01	1.44

Set4

Machine	PS	P3	K6	SS	M	I	C
comp	0.78	0.75	2.50	1.16	1.79	1.63	0.97
branchless	0.96	0.85	3.01	0.61	1.16	1.36	0.88
all2	0.98	1.21	3.87	1.94	2.09	1.47	0.67
dls	1.36	1.16	3.98	1.37	2.71	2.03	1.57
ls	1.08	0.95	3.60	1.04	2.44	1.50	FPE
sl	1.23	1.33	4.73	1.53	2.72	1.52	1.82
all	1.77	1.38	5.07	1.71	2.78	2.32	1.57

FIG. 1 – Résultats sur les différentes machines

4. Conclusion

Le problème à traiter était extrêmement simple mais il a donné lieu à de multiples implantations. Ce problème régulier qui n'utilise que des additions s'est révélé avoir un comportement complexe extrêmement dépendant de l'architecture.

Les programmes sont uniquement composés d'additions flottantes, de manipulation d'entiers et de branchements. La gestion du pipeline et la prédiction des sauts est donc exploitée au maximum et on peut en voir les limites. Certaines architectures privilégient la rapidité des calculs au détriment du pipeline ou *vice versa*.

D'autres tests ont été effectués sur un Pentium MMX à 200MHz sous Linux, un Pentium Pro 200MHz sous Linux et un Pentium III à 500 MHz sous Linux. Les résultats étaient sensiblement équivalents si ce n'est que l'addition `a112` est plus rapide que `1s` sur ces trois machines. Les calculs par rapport à P3 sont environ 3.5 fois plus lents sur le Pentium MMX, 4 fois sur le Pentium Pro et 1.1 fois sur le Pentium III à 500 MHz.

Dans presque tous les cas, les additions les plus rapides sont `a112`, `branchless` et `comp`, qui considèrent toutes les composantes. Il semble qu'écarter les composantes les moins significatives ne soit pas intéressant du point de vue du temps de calcul. Il est souvent nécessaire de calculer au moins 5 composantes alors que le résultat exact en comporte au plus 8 et même 5 dans le cas du Set3.

La question du tri reste entière. En effet, `comp` nécessite un tri au contraire des deux autres et il est nettement plus rapide sur les Pentiums et sur le K6, mais pas sur les autres processeurs.

Éviter les sauts conditionnels semble porter ses fruits spécialement sur Sparc, Power PC et IBM. Cela signifie que beaucoup de temps est perdu lorsque l'on casse le pipeline.

L'algorithme `1s`, qui est probablement le plus simple, obtient d'assez bons résultats sur la plupart des architectures. Sur le CRAY, il crée une FPE (Floating Point Exception) pour une raison inconnue. Cela est peut-être dû au fait que le CRAY n'est pas compatible avec la norme IEEE alors que les implantations le supposent.

L'implantation `a112` est très bien placée sur toutes les architectures. C'est un bon compromis, car elle présente l'avantage d'une garantie d'exactitude au contraire de `branchless` et `comp`.

Il est heureux que les priorités des concepteurs soient si différentes. Sans modèle clair, il est toutefois difficile au programmeur d'écrire un bon algorithme et une bonne implantation pour résoudre un problème quand on voit les disparités des processeurs actuels. Vu la simplicité du problème étudié et les disparités rencontrées, on peut être sûr qu'un programme un peu gros très bon sur une machine puisse se révéler catastrophique sur une autre.

Bibliographie

1. BAILEY, D. H. Algorithm 719, multiprecision translation and execution of fortran programs. *ACM Transactions on Mathematical Software* 19, 3 (1993), 288–319.
2. BAILEY, D. H. A Portable High Performance Multiprecision Package. Tech. Rep. RNR-90-022, NASA Ames Research Center, Moffett Field, California, May 1993.
3. BAILEY, D. H., KRASNY, R., AND PELZ, R. Multiple precision, multiple processor vortex sheet roll-up computation. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing* (Philadelphia, Pennsylvania, 1993), pp. 52–56.
4. BOLDO, S. Calcul rapide et exact de fonctions élémentaires en précision arbitraire par la moyenne arithmético-géométrique. Training period, LORIA, 1999.
5. BOLDO, S. Quad double precision specifications and proofs about the addition. Training period, University of California at Berkeley, 2000.
6. COMPAQ. *Alpha 21164 Microprocessor Hardware Reference Manual*, 1998.
7. DAUMAS, M., AND FINOT, C. Division of floating point expansions with an application to the computation of a determinant. *Journal of Universal Computer Science* 5, 6 (1999), 323–338.
8. DAUMAS, M., AND MULLER, J.-M., Eds. *Qualité des calculs sur ordinateur: vers des arithmétiques plus fiables*. Masson, 1997.
9. DEKKER, T. J. A Floating-Point Technique for Extending the Available Precision. *Numerische Mathematik* 18 (1971), 224–242.
10. GOLDBERG, D. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys* 23, 1 (1991), 5–47.

11. GOLDBERG, D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996, ch. Computer Arithmetic, pp. A1–A77.
12. GRANLUND, T. *The GNU multiple precision arithmetic library*, 2000. Version 3.1.
13. GWENNAP, L. X86 outdoes RISC performance. *Microprocessor Report* 13, 17 (1999), 1, 6–9.
14. HENNESSY, J. L., AND PATTERSON, D. A. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 1990.
15. HENNESSY, J. L., AND PATTERSON, D. A. *Computer architecture: A quantitative approach*. Morgan Kaufmann, 1996.
16. HENNING, J. L. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer* 33, 7 (2000), 28–35.
17. HIDA, Y., LI, X. S., AND BAILEY, D. H. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th Symposium on Computer Arithmetic* (Vail, Colorado, 2001), N. Burgess and L. Ciminiera, Eds.
18. INTEL. *Pentium Family: Developer's Manual*, 1995. Architecture and Programming Manual.
19. KAHAN, W. Implementation of algorithms. Part 1. Lecture Notes AD-769 124, National Technical Information Service, 1973.
20. KESSLER, R. E. The Alpha 21264 microprocessor. *IEEE Micro* 19, 2 (1999), 24–36.
21. KNUTH, D. E. *The Art of Computer Programming: Seminumerical Algorithms*, second ed., vol. 2. Addison Wesley, Reading, Massachusetts, 1981.
22. KOREN, I. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.
23. KREWELL, K. SPEC CPU2000 released. *Microprocessor Report* 14, 4 (2000), 21–25.
24. MILLER, W. *The engineering of numerical software*. Prentice Hall, 1984.
25. MOTOROLA. *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, 1997.
26. PRIEST, D. M. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic* (Grenoble, France, 1991), P. Kornerup and D. Matula, Eds., IEEE Computer Society Press, pp. 132–144.
27. PRIEST, D. M. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, Department of Mathematics, University of California at Berkeley, Berkeley, California, Nov. 1992. Available by anonymous FTP to [ftp.icisi.berkeley.edu](ftp://ftp.icisi.berkeley.edu) as `pub/theory/priest-thesis.ps.Z`.
28. SEZNEC, A., AND MÉVEL, Y. évolution des gammes de processeurs MIPS, DEC Alpha, PowerPC, SPARC et xxx86. Publication interne 957, Institut de Recherche en Informatique et Systèmes Aléatoires, 1995.
29. SHEWCHUK, J. R. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry* 18, 3 (Oct. 1997), 305–363.
30. STEVENSON, D., ET AL. An american national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices* 22, 2 (1987), 9–25.
31. SUN MICROSYSTEMS. *Numerical Computation Guide*, 1996.
32. WEAVER, D. L., AND GERMOND, T., Eds. *The SPARC architecture manual*. Prentice Hall, 1994. Version 9.