부록 D

모든 컴퓨터 과학자가 부동 소수점 산술에 대해 알아야 할 사항

주 – 이 부록은 Computing Surveys 1991 년 3 월호에 출판 된 David Goldberg가 발행한 *What Every Computer Scientist Should Know About Floating-Point Arithmetic* 논문의 편집 판입니다 .

요약

부동 소수점 산술은 많은 사람들에게 난해한 주제로 간주됩니다. 부동 소수점은 컴퓨터 시스템에서 어디에나 있기 때문에 다소 놀랍습니다. 거의 모든 언어에는 부동 소수점 데이터 유형이 있습니다. PC에서 슈퍼 컴퓨터에 이르는 컴퓨터에는 부동 소수점 가속기가 있습니다. 대부분의 컴파일러는 때때로 부동 소수점 알고리즘을 컴파일하기 위해 호출됩니다. 사실상 모든 운영 체제는 오버플로와 같은 부동 소수점 예외에 응답해야합니다. 이 백서에서는 컴퓨터 시스템 설계자에게 직접적인 영향을 미치는 부동 소수점의 측면에 대한 자습서를 제공합니다. 부동 소수점 표현 및 반올림 오류에 대한 배경 지식으로 시작하여 IEEE 부동 소수점 표준에 대한 논의를 계속하며 컴퓨터 빌더가 부동 소수점을 더 잘 지원할 수있는 방법에 대한 수많은 예제로 마무리합니다.

범주 및 주제 설명자 : (1 차) C.0 [컴퓨터 시스템 조직] : 일반- 명령 세트 설계 ; D.3.4 [프로그래밍 언어] : 프로세서- 컴파일러, 최적화 ; G.1.0 [Numerical Analysis] : 일반- 컴퓨터 산술, 오류 분석, 수치 알고리즘 (2 차)

D.2.1 [소프트웨어 엔지니어링] : 요구 사항 / 사양- 언어 ; D.3.4 프로그래밍 언어] : 형식적 정의 및 이론- 의미론 ; D.4.1 운영 체제] : 프로세스 관리- 동기화 .

일반 용어 : 알고리즘, 디자인, 언어

추가 키워드 및 구문 : 비정규 화 된 숫자, 예외, 부동 소수점, 부동 소수점 표준, 점진적 언더 플로, 보호 숫자, NaN, 오버플로, 상대 오류, 반올림 오류, 반올림 모드, ulp, 언더 플로.

소개

컴퓨터 시스템 빌더는 종종 부동 소수점 산술에 대한 정보를 필요로합니다. 그러나 이에 대한 자세한 정보의 출처는 매우 적습니다. 이 주제에 관한 몇 안되는 책 중 하나 인 Pat Sterbenz의 부동 소수점 계산 은 오래 절판되었습니다. 이 문서는 시스템 구축에 직접 연결되는 부동 소수점 산술 (이하 부동 소수점)의 측면에 대한 자습서입니다 . 느슨하게 연결된 세 부분으로 구성됩니다. 첫 번째 섹션 인 반올림 오류 에서는 더하기, 빼기, 곱하기 및 나누기의 기본 연산에 대해 다른 반올림 전략을 사용하는 것의 의미에 대해 설명합니다.

또한 반올림 오차를 측정하는 두 가지 방법 인 ulps 및relative error. 두 번째 부분에서는 상용 하드웨어 제조업체에서 빠르게 수용되고있는 IEEE 부동 소수점 표준에 대해 설명합니다. IEEE 표준에는 기본 작업을위한 반올림 방법이 포함되어 있습니다. 표준에 대한 논의는 반올림 오류 섹션의 자료를 기반으로합니다 . 세 번째 부분에서는 부동 소수점과 컴퓨터 시스템의 다양한 측면 설계 간의 연결에 대해 설명합니다. 주제에는 명령어 세트 설계, 컴파일러 최적화 및 예외 처리가 포함됩니다.

나는 특히 정당화가 기초 미적분보다 더 복잡한 것이 없기 때문에 진술이 사실 인 이유를 제시하지 않고 부동 소수점에 대한 진술을 피하려고 노력했습니다. 주된 주장의 중심이 아닌 설명은 "세부 사항"이라는 섹션으로 그룹화되어 원하는 경우 건너 뛸 수 있습니다. 특히 이 섹션에는 많은 정리의 증명이 나와 있습니다. 각 증명의 끝은 z 기호 로 표시됩니다 . 증명이 포함되어 있지 않으면 정리 진술 바로 뒤에 z 가 나타납니다.

반올림 오류

무한히 많은 실수를 유한 한 수의 비트로 압축하려면 대략적인 표현이 필요합니다. 무한히 많은 정수가 있지만 대부분의 프로그램에서 정수 계산 결과는 32 비트로 저장 될 수 있습니다. 반대로 고정 된 수의 비트가 주어지면 실수를 사용하는 대부분의 계산은 많은 비트를 사용하여 정확하게 표현할 수없는 양을 생성합니다. 따라서 부동 소수점 계산의 결과는 유한 표현에 다시 맞추기 위해 종종 반올림되어야합니다. 이 반올림 오류는 부동 소수점 계산의 특징입니다. 상대 오류 및 Ulps 섹션에서는 측정 방법을 설명합니다.

어쨌든 대부분의 부동 소수점 계산에는 반올림 오류가 있기 때문에 기본 산술 연산이 필요한 것보다 약간 더 많은 반올림 오류를 발생시키는 것이 중요합니까? 이 질문은이 섹션 전체에서 주요 주제입니다. 섹션 가드 자리는 설명 가드 자리,이 개 근처의 숫자를 뺀 때 오류를 줄일 수있는 방법을. 가드 디지트는 1968 년 System / 360 아키텍처의 배정 밀도 형식에 가드 디지트를 추가하고 (단일 정밀도에는 이미 가드 디지트가 있음) IBM에 의해 충분히 중요하다고 간주되었으며 현장의 모든 기존 기계를 개조했습니다. 가드 디지트의 유용성을 설명하기 위해 두 가지 예가 제공됩니다.

IEEE 표준은 가드 디지트 사용을 요구하는 것 이상을 제공합니다. 더하기, 빼기, 곱하기, 나누기 및 제곱근에 대한 알고리즘을 제공하고 구현시 해당 알고리즘과 동일한 결과를 생성해야합니다. 따라서 프로그램이 한 컴퓨터에서 다른 컴퓨터로 이동할 때 두 컴퓨터가 IEEE 표준을 지원하는 경우 기본 작업의 결과는 모든 비트에서 동일합니다. 이것은 프로그램의 이식을 크게 단순화합니다. 이 정확한 사양의 다른 용도는 정확히 반올림 작업에 나와 있습니다.

부동 소수점 형식

실수에 대한 몇 가지 다른 표현이 제안되었지만 지금까지 가장 널리 사용되는 것은 부동 소수점 표현입니다. [1] 부동 소수점 표현에는 밑 β(항상 짝수라고 가정)과 정밀도 *p*가 있습니다. 경우 β= 10, *P* = 3, 그 수는 1.00, 0.1로 표현 × 10^{-1} . 경우 β= 2, *P* = 24, 다음 진수 0.1 정확히 표현하지만 약 1.10011001100110011001101 인 할 수 없다 × 2^{-4} .

일반적으로, 부동 소수점 숫자로 표현한다 ± *d.dd* ... *D* × β^{E} , *d.dd* ... *D*가 호출되고 , 유효 숫자 [2] 및 갖는 *P*의 숫자. 더 정확하게는 ± $d_0 . d_1 d_2 \ldots d_{p-1} \times e$ 는 숫자를 나타냅니다. β

(1) $\pm(d_0 + d_1\beta^{-1} + \ldots + d_{p-1}\beta^{-(p-1)})\beta^{e}, (0 \le d_i < \beta)$.

용어 부동 소수점 숫자 는 논의중인 형식으로 정확하게 표현 될 수있는 실수를 의미하는 데 사용됩니다. 부동 소수점 표현과 관련된 두 개의 다른 매개 변수는 허용 가능한 최대 및 최소 지수 인 e_{max} 및 e_{min} 입니다. 있기 때문에 ^{P} 가능한 significands 및 전자 $_{최대}$ - 전자 $_{분}$ + 1 개 상정 지수는, 부동 소수점 수는 인코딩 될 수있다β

$$[\log_2(e_{max} - e_{min} + 1)] + [\log_2(\beta^p)] + 1$$

마지막 +1은 부호 비트에 대한 것입니다. 현재로서는 정확한 인코딩이 중요하지 않습니다.

실수가 부동 소수점 숫자로 정확하게 표현되지 않는 데에는 두 가지 이유가 있습니다. 가장 일반적인 상황은 십진수 0.1로 설명됩니다. 유한 십진 표현이 있지만 이진법에서는 무한 반복 표현이 있습니다. 따라서 β= 2 일 때 숫자 0.1은 두 부동 소수점 숫자 사이에 엄격하게 놓여 있으며 둘 중 어느 것도 정확하게 표현할 수 없습니다. 덜 일반적인 상황은 실수가 범위를 벗어났다는 것입니다. 즉, 절대 값이 × 보다 크거나 1.0 × 보다 작습니다 . 이 문서의 대부분은 첫 번째 이유로 인한 문제에 대해 설명합니다. 그러나 범위를 벗어난 숫자는 Infinity 및 Denormalized Numbers 섹션에서 설명합니다 .β $\beta^{e_{max}}$ $\beta^{e_{min}}$

부동 소수점 표현이 반드시 고유 한 것은 아닙니다. 예를 들어 0.01 × 10^{1} 과 1.00 × 10^{-1} 은 모두 0.1을 나타냅니다. 선행 숫자가 0이 아닌 경우 (위의 방정식 (1) 에서 d_0 ≠ 0), 표현은 정규화 되었다고합니다 . 부동 소수점 숫자 1.00 × 10^{-1} 은 정규화되지만 0.01 × 10^{1} 은 정규화 되지 않습니다. = 2, *P* = 3, 전자 $_{분}$ = -1 즉 $_{최대}$ = 2에 도시 된 바와 같이 16 개의 정규화 부동 소수점 수는있다β그림 D-1 . 굵은 해시 표시는 유효 숫자가 1.00 인 숫자에 해당합니다. 부동 소수점 표현이 정규화되도록 요구하면 표현이 고유합니다. 불행히도이 제한은 0을 표현하는 것을 불가능하게합니다! 0을 나타내는 자연스러운 방법은 1.0 ×입니다 $\beta^{e_{min}-1}$. 이것은 음이 아닌 실수의 숫자 순서가 부동 소수점 표현의 사전 순서와 일치한다는 사실을 보존하기 때문입니다. [3] 지수가 *k* 비트 필드에 저장되면 0을 나타내도록 예약해야하므로 지수로 사용할 수있는 값은 2^k -1 뿐입니다 .

부동 소수점 숫자 의 × 는 표기법의 일부이며 부동 소수점 곱하기 연산과 다릅니다. × 기호 의 의미 는 문맥에서 명확해야합니다. 예를 들어 (2.5 × 10^{-3}) × (4.0 × 10^{2}) 식은 단일 부동 소수점 곱셈 만 포함합니다.

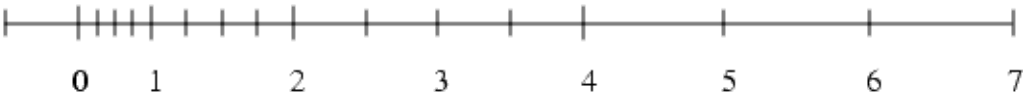


그림 D-1 β= 2, *p* = 3, e_{min} = -1, e_{max} = 2 인 경우 정규화 된 숫자

상대 오차 및 Ulps

반올림 오류는 부동 소수점 계산에 내재되어 있으므로이 오류를 측정하는 방법을 갖는 것이 중요합니다. 이 섹션 전체에서 사용되는 β = 10 및 *p* = 3 의 부동 소수점 형식을 고려하십시오. 부동 소수점 계산의 결과가 3.12 × 10^{-2} 이고 무한 정밀도로 계산했을 때의 답이 .0314이면 마지막 위치에서 2 단위 오류가 있음이 분명합니다. 마찬가지로 실수 .0314159가 3.14 × 10^{-2} 로 표시되면 마지막 자리에서 .159 단위로 오류가 발생합니다. 일반적으로 부동 소수점 숫자 *dd* ... *d* × ^{e} 가 *z* 를 나타내는 데 사용되는 경우β다음으

로는 오류가 *DD* ... *D* - (*Z* / ^{E}) $^{P-1}$ 의 마지막 위치에있는 단위. [4] , [5] *ulps* 라는 용어 는 "마지막 자리에있는 단위"의 약어로 사용됩니다. 계산 결과가 올바른 결과에 가장 가까운 부동 소수점 숫자 인 경우 여전히 .5 ulp만큼 오류가있을 수 있습니다. 부동 소수점 숫자와 근사하는 실수의 차이를 측정하는 또 다른 방법은 상대 오차입니다 . 이는 단순히 두 숫자를 실수로 나눈 차이입니다. 3.14 × 10 3.14159에 근접 할 때 예를 들어, 상대 오차는 최선을 다하고 ^{0}|β|β .00159 / 3.14159 ≈.0005입니다.

0.5 ULP에 대응이 실수가 가장 근접한 부동 소수점 숫자에 의해 근사 될 때 관찰되는 상대 오차 계산하기 *d.dd* ... *DD* × ^{E} , 오류가 될 수 큰 0.00 ... 00 ' × ^{E} 여기서 '가 숫자 인 / 2가있는 *P*의 부동 소수점 수의 유효 숫자의 유닛들, 및 *P*는 에러의 유효 숫자 0의 단위. 이 오류는 ((/ 2) $^{-p}$) × ^{e} 입니다. *d.dd* ... *dd* × ^{e} 형식의 숫자 이후 β ββββ ββ β β모두 동일한 절대 오차를 갖지만 ^{e} 와 × ^{e} 사이의 값을 가지며 상대 오차 범위는 ((/ 2) $^{-p}$) × ^{e} / ^{e} 와 ((/ 2) $^{-p}$) × ^{e} / $^{e + 1}$ 사이입니다. . 그건,ββ βββββββββ

(2) $\frac{1}{2}\beta^{-p} \le \frac{1}{2}\mathrm{ulp} \le \frac{\beta}{2}\beta^{-p}$

특히 .5 ulp에 해당하는 상대 오차는 β. 이 요인을 흔들림 이라고합니다 . 설정 ε= (β/ 2) $^{-p}$ 의 경계의 최대에 (2) 위의, 우리는 실수를 가장 가까운 부동 소수점 숫자로 반올림 됩니다 때 상대 오차는 항상 의해 제한되는 것을 말할 수있는 전자 ,라고하는 등을 기계 엡실론 .β

위의 예에서 상대 오류는 .00159 / 3.14159 ≈.0005입니다. 이러한 작은 숫자를 피하기 위해 상대 오류는 일반적으로 factor times으로 작성되며, ε이 경우 ε= (β/ 2) $^{-p}$ = 5 (10) $^{-3}$ = .005입니다. 따라서 상대 오차는 (.00159 / 3.14159) /. 005) 0.1 로 표현됩니다 .βε ≈ε

ulps와 상대 오차의 차이를 설명하기 위해 실수 *x* = 12.35를 고려하십시오 . 근사값은 $\tilde{x}$ = 1.24 × 10 ^{1} 입니다. 오차는 0.5ulps이고 상대 오차는 0 입니다. *8*ε . 다음으로 계산 8 을 고려하십시오 $\tilde{x}$. 정확한 값은 8 *x* = 98.8이고 계산 된 값은 8 $\tilde{x}$ = 9.92 × 10 ^{1} 입니다. 오류는 이제 4.0ulps이지만 상대 오류는 여전히 0 입니다. *8*ε . 상대 오차는 동일하더라도 ulps 단위로 측정 된 오차는 8 배 더 큽니다. 일반적으로 밑이 β이면 ulps로 표현되는 고정 상대 오차는 최대 계수까지 흔들릴 수 있습니다.β. 반대로, 위의 방정식 (2) 에서 알 수 있듯이 .5 ulps의 고정 오차는 β.

반올림 오차를 측정하는 가장 자연스러운 방법은 ulps입니다. 예를 들어 가장 가까운 부동 소수점 숫자로 반올림하면 .5 ulp 이하의 오류에 해당합니다. 그러나 다양한 공식으로 인한 반올림 오차를 분석 할 때 상대 오차가 더 나은 측정입니다. 이것에 대한 좋은 예시는 Theorem 9 섹션의 분석입니다 . ε의 흔들림 인자에 의해 가장 가까운 부동 소수점 수로 반올림 한 효과를 과대 평가할 수 있으므로 β수식의 오류 추정은 β.

반올림 오차의 크기 만 관심이있는 경우, ulps 및는 ε최대 β. 부동 소수점 숫자에 의해 오류가 예를 들어, *N* 개의 ulps는 기초적, 즉 오염 자릿수 로그 있음 없음 . 계산의 상대 오차가 *n* 이면βε

(3) 오염 된 숫자 ≈log *n* .β

가드 숫자

두 부동 소수점 숫자의 차이를 계산하는 한 가지 방법은 차이를 정확하게 계산 한 다음 가장 가까운 부동 소수점 숫자로 반올림하는 것입니다. 피연산자의 크기가 크게 다른 경우 이

는 매우 비쌉니다. 가정하면 , $P = 3$, $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ 로 계산 될

$$x = 2.15 \times 10^{12}$$
$$y = .000000000000000125 \times 10^{12}$$
$$x - y = 2.1499999999999999875 \times 10^{12}$$

2.15×10^{12}로 반올림됩니다 . 이러한 모든 숫자를 사용하는 대신 부동 소수점 하드웨어는 일반적으로 고정 된 숫자에서 작동합니다. 유지되는 자릿수가 p 이고 더 작은 피연산자가 오른쪽으로 이동하면 자릿수는 단순히 버려집니다 (반올림이 아님). 그리고 $2.15 \times 10^{(12)} - 1.25 \times 10^{-5}$ 가된다

$$x = 2.15 \times 10^{12}$$
$$y = 0.00 \times 10^{12}$$
$$x - y = 2.15 \times 10^{12}$$

대답은 차이가 정확히 계산 된 다음 반올림 된 것과 똑같습니다. 다른 예를 들어 보자 : 10.1-9.93. 이것은

$$x = 1.01 \times 10^{1}$$
$$y = 0.99 \times 10^{1}$$
$$x - y = .02 \times 10^{1}$$

정답은 .17이므로 계산 된 차이는 30ulps만큼 떨어져 있으며 모든 숫자에서 잘못되었습니다! 오류가 얼마나 심할 수 있습니까?

정리 1

파라미터를 가지는 부동 소수점 형식으로 사용 및 P를 , 그리고 사용의 차이 계산 P의 숫자 결과의 상대 오차가 큰 같을 수 - *1* .β β

증명

상대 오차 β식 1 - X - Y는 경우 발생 $X = 1.00 \ldots 0$, Y =를 . ρρ...ρ , where ρ= β -1. 여기서 y 는 p 자리 (모두)를 갖습니다 ρ. 정확한 차이는 $X - Y = {}^{-P}$. 그러나만을 사용하여 대답을 계산할 때, P의 숫자의 자릿수 우측 Y는 이동 된 도착하고, 계산 된 차이가 너무 ${}^{-(P)+1}$. 따라서 오류입니다 ${}^{-P}$ -ββββ ${}^{-P+1} = {}^{-P}(-1)$ 및 상대 오차가 ${}^{-P}(-1) / {}^{-(P)} = -1$ zββββββ

경우 β(2) = 상대 오차는 결과적으로 큰, 등 일 수 β= (10)는, 그것이 일 수 9 배. 즉, β= 2 일 때 방정식 (3) 은 오염 된 자릿수가 $\log_2(1/\varepsilon) = \log_2(2^p) = p$ 라는 것을 보여줍니다 . 즉, 결과의 모든 p 자리가 잘못되었습니다! 이 상황을 방지하기 위해 하나의 추가 숫자 (가드 숫자) 가 추가되었다고 가정하십시오 . 즉, 작은 숫자는 $p + 1$ 자리로 잘린 다음 빼기 결과는 p 자리 로 반올림됩니다 . 가드 디지트를 사용하면 이전 예는

$x = 1.01\ 0 \times 10^{1}$

$y = 0.993 \times 10^{1}$

$x - y = .017 \times 10^{1}$

답은 정확합니다. 가드 숫자가 하나 인 경우 결과의 상대 오차는 ε110-8.59에서와 같이 보다 클 수 있습니다 .

$x = 1.1\ 0 \times 10^{2}$

$y = .085 \times 10^{2}$

$x - y = 1.015 \times 10^{2}$

이 값은 101.41의 정답과 비교하여 .006의 상대 오차에 대해 102로 반올림되며 ε= .005 보다 큽니다 . 일반적으로 결과의 상대 오차는보다 약간만 클 수 있습니다 ε. 더 정확하게,

정리 2

경우 *X*는 및 *y*는 변수와 형식 번호 소수점 부동 와 *P* , 및 감산으로 수행하는 경우 , *P* + *1* 자리수 *(*즉 하나의 가드 자리*),* 그 결과에 대하여 라운딩 에러가 *2* 개 미만이다 .β ε

이 정리는 반올림 오류 에서 입증됩니다 . *x* 와 *y* 는 양수 또는 음수가 될 수 있으므로 덧셈 이 위의 정리에 포함됩니다 .

해제

마지막 섹션은 가드 디지트가 없으면 두 개의 인접 수량을 뺄 때 발생하는 상대적 오류가 매우 클 수 있다고 말하여 요약 할 수 있습니다. 즉, 빼기가 포함 된 표현식 (또는 반대 부호가있는 수량 추가)을 평가하면 모든 숫자가 의미 가 없을 정도로 큰 상대 오차가 발생할 수 있습니다 (정리 1). 가까운 수량을 뺄 때 피연산자의 최상위 숫자가 서로 일치하고 취소됩니다. 취소에는 두 가지 종류가 있습니다 : 재앙 적 및 정상적입니다.

피연산자에 반올림 오류가 발생하면 치명적인 취소 가 발생합니다. 이차 식 예를 들어, 식 $(B)^{2} - 4\ AC$가 발생한다. 수량 b^{2} 및 4 *ac* 는 부동 소수점 곱셈의 결과이므로 반올림 오류가 발생합니다. 가장 가까운 부동 소수점 숫자로 반올림되어 *.5* ulp 내에서 정확하다고 가정합니다 . 값을 빼면 취소로 인해 정확한 숫자가 많이 사라지고 반올림 오류로 오염 된 숫자가 주로 남습니다. 따라서 차이는 많은 ulps의 오류를 가질 수 있습니다. 예를 들어, *b* = 3.34, *a*를 고려 하십시오. = 1.22 및 *c* = 2.28. 정확한 값 $B^{2} - 4\ AC$는 0.0292이다. 그러나 b^{2} 는 11.2로 반올림되고 4 *ac* 는 11.1로 반올림되므로 11.2-11.1이 정확히 .1[6] 과 동일하더라도 최종 답은 .1입니다. 이는 70 ulps 오류 입니다. 뺄셈은 오류를 유발하지 않았지만 이전 곱셈에서 도입 된 오류를 노출했습니다.

양성 취소 는 정확히 알려진 수량을 뺄 때 발생합니다. 경우 *X* 및 *Y*는 없는 오류가 라운딩 한 후 정리 2에 의해 감산이 가드 숫자로 이루어진다면, 차이 *X* -y 매우 작은 상대 오차 (2 미만을 갖는다 ε).

치명적인 취소를 나타내는 공식은 때때로 문제를 제거하기 위해 재 배열 될 수 있습니다. 다시 2 차 공식을 고려하십시오.

(4) $r_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}, r_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$

때 $b^2 \gg ac$, 다음 b^2-4ac 취소를 포함하지 않으며,

$\sqrt{b^2-4ac} \approx |b|$.

그러나 공식 중 하나의 다른 더하기 (빼기)는 치명적인 취소를 갖습니다. , 다중 분자와 분모이를 방지하려면 R_1 에 의해

$-b-\sqrt{b^2-4ac}$

(그리고 비슷하게 r_2)

(5) $r_1 = \frac{2c}{-b-\sqrt{b^2-4ac}}, r_2 = \frac{2c}{-b+\sqrt{b^2-4ac}}$

경우 $b^2 \gg ac$ 와 $b>0$ 다음 컴퓨팅 R_1 화학식 이용 (4)의 취소를 포함 할 것이다. 따라서, 식 사용 (5)를 계산하기위한 R_1 및 (4) 에 대한 $(R)_2$. 한편, B <0, 사용 (4) 연산 R_1 및 (5) 에 대한 $(R)_2$.

식 $x^2 - y^2$ 는 치명적인 취소를 나타내는 또 다른 공식입니다. ($x - y$) ($x + y$) 로 평가하는 것이 더 정확합니다 . [7] 이차 공식과 달리이 개선 된 형태는 여전히 빨셈이 있지만 반올림 오류가없는 양의 양의 취소이며 치명적인 것이 아닙니다. 정리 2에 따르면 $x - y$ 의 상대 오차 는 최대 2 ε입니다. $x + y$도 마찬가지입니다 . 작은 상대 오차로 두 수량을 곱하면 상대 오차가 작은 제품이됩니다 (섹션 참조).반올림 오류).

정확한 값과 계산 된 값 사이의 혼동을 피하기 위해 다음 표기법이 사용됩니다. $x - y$ 가 x 와 y 의 정확한 차이를 나타내는 반면 , $x \ominus y$ 는 계산 된 차이를 나타냅니다 (즉, 반올림 오류 포함). 유사하게 ⊕, ⊗그리고 ⊘ 참조 부호를 각각 첨가, 곱셈, 나눗셈 연산. 모든 대문자는 LN(x)또는 에서와 같이 함수의 계산 된 값을 나타냅니다 SQRT(x). 소문자 함수와 전통적인 수학적 표기법은 ln (x) 및에서와 같이 정확한 값을 나타냅니다 $\sqrt{x}$.

($x \ominus y$) ⊗($x y$)는 x^2 -y ^{2}에 대한 훌륭한 근사치 이지만 부동 소수점 숫자 x 및 y 자체는 일부 실제 수량 및 에 대한 근사치 일 수 있습니다 . 예를 들어, 그리고 정확히 이진수로 정확하게 표현할 수없는 진수를 알 수 있습니다. 이 경우, 비록 X , Y가 좋은 근사이다 X - Y , 그것은 참된 식에 비해 큰 상대 오차를 가질 수 및 (이점 때문에 $X + Y$⊕ $\hat{x}\,\hat{y}\;\hat{x}\,\hat{y}\;\ominus\;\hat{x}-\hat{y}$) ($x - y$) over x^2 -y ^{2} 는 그다지 극적이지 않습니다. ($x + y$) ($x - y$) 계산은 x^2 -y ^{2} 계산과 거의 같은 양이므로,이 경우 분명히 선호되는 형식입니다. 그러나 일반적으로 입력이 (항상은 아니지만) 근사치이기 때문에 비용이 큰 경우 치명적인 취소를 무해한 취소로 대체하는 것은 가치가 없습니다. 그러나 데이터가 정확하지 않더라도 취소를 완전히 제거하는 것은 (2 차 공식에서와 같이) 가치가 있습니다. 이 문서에서는 알고리즘에 대한 부동 소수점 입력이 정확하고 결과가 가능한 한 정확하게 계산된다고 가정합니다.

$x^2 - y^2$ 표현식 은 ($x - y$) ($x + y$) 로 재 작성할 때 더 정확합니다. 이는 치명적인 취소가 무해한 취소로 대체되기 때문입니다. 다음에 우리는 양성 취소만을 나타내도록 재 작성할 수있는 치명적인 취소를 나타내는 수식의 더 흥미로운 예를 제시합니다.

삼각형의 면적은 그 측면의 길이의 관점에서 직접 표현할 수 , B , 및 C 로서

(6) $A = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$

(삼각형이 매우 평평하다고 가정합니다. 즉, a b + c . 그런 다음 s a 이고 식 (6) 의 항 (s - a)는 두 개의 가까운 숫자를 뺍니다. 그 중 하나는 반올림 오류가있을 수 있습니다. a = 9.0, b = c = 4.53, s 의 정확한 값 은 9.03이고 A 는 2.342입니다 .... s (9.05) 의 계산 된 값이 2ulps 에 의해서만 오류가 있지만 A 의 계산 된 값 은 3.04입니다. , 70 ulps의 오류. ≈ ≈

평평한 삼각형에 대해서도 정확한 결과를 반환하도록 공식 (6) 을 다시 작성하는 방법이 있습니다 [Kahan 1986]. 그것은

(7) $A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}, a \geq b \geq c$

경우 , B, 및 C가 만족하지 않는 B에 C를 도포하기 전에 이름을 변경 (7) . (6) 과 (7) 의 오른쪽 이 대수적으로 동일한 지 확인하는 것은 간단 합니다. 위 의 a , b , c 값을 사용하면 계산 된 면적이 2.35로, 오류가 1ulp이고 첫 번째 공식보다 훨씬 더 정확합니다. ≥ ≥

공식하지만 (7) 훨씬 더 정확보다 (6) 이 예를 들어, 얼마나 잘 알고 좋을 것이다 (7) 일반적으로 수행합니다.

정리 3

삼각형의 면적을 계산하기 위해 (7) 을 사용할 때 발생하는 반올림 오류 는 최대 *11*입니다 ε. 단, 가드 숫자 e .005로 빼기가 수행되고 제곱근이 1/2 ulp 이내로 계산됩니다 . ≤

e <.005가 사실상 모든 실제 부동 소수점 시스템에서 충족 되는 조건입니다 . 예를 들어 β = 2 일 때 p ≥ 8은 e <.005이고 β= 10 일 때 p ≥ 3이면 충분합니다.

표현식의 상대 오차를 설명하는 정리 3과 같은 명령문에서 표현식은 부동 소수점 산술을 사용하여 계산되는 것으로 이해됩니다. 특히 상대 오차는 실제로

(8) SQRT((a⊕ (b c)) (c (a b)) (c (a b)) (a (b c))) 4⊕ ⊗ ⊖ ⊖ ⊗⊕ ⊖ ⊗⊕ ⊖ ∅

(8) 의 성가신 성격 때문에 정리 진술에서 우리는 일반적 으로 원 표기법으로 E 를 쓰는 대신 E 의 계산 된 값을 말할 것 입니다.

오류 경계는 일반적으로 너무 비관적입니다. 위에 주어진 수치 예제에서 (7) 의 계산 된 값은 2.35이고, 상대 오차 0.7에 대한 실제 값 2.34216 (ε11보다 훨씬 작음)에 비해 비교 ε됩니다. 오류 경계를 계산하는 주된 이유는 정확한 경계를 얻기위한 것이 아니라 수식에 숫자 문제가 포함되어 있지 않은지 확인하기위한 것입니다.

무해한 취소를 사용하도록 다시 작성할 수있는 식의 마지막 예는 (1 + x) ^{n} $x \ll 1$ 입니다. 여기서 . 이 표현은 재무 계산에서 발생합니다. 매일 복리로 계산되는 연간 이자율이 6 %인 은행 계좌에 매일 $ 100를 입금하는 것을 고려하십시오. 경우 N = 365 그리고 난 0.06 =를 하나 연말 누적 금액은

100 $\frac{(1+i/n)^n - 1}{i/n}$

불화. 이 값이 β= 2 및 p = 24를 사용하여 계산 되면 결과는 $ 37615.45이며, $ 1.40의 불일치 인 $ 37614.05의 정확한 답과 비교됩니다. 문제의 원인은 쉽게 알 수 있습니다. 표현식 1 + i / n 은 .0001643836에 1을 더하는 것을 포함하므로 i / n 의 하위 비트 가 손실됩니다. 이 반올림 오차는 1 + i / n 이 n 번째 거듭 제곱 일 때 증폭됩니다 .

성가신 식 $(1 + i/n)^n$ 은 $e^{n\ln(1+i/n)}$ 으로 다시 쓸 수 있습니다 . 여기서 문제는 작은 x에 대해 $\ln(1+x)$ 를 계산하는 것 입니다. 한 가지 접근 방식은 근사값 $\ln(1+x)$ x를 사용하는 것 입니다.이 경우 지불액은 $ 37617.26이되고, 이는 $ 3.21만큼 떨어져 있고 명백한 공식보다 훨씬 덜 정확합니다. 그러나 정리 4가 보여주는 것처럼 $\ln(1+x)$를 매우 정확하게 계산하는 방법이 있습니다 [Hewlett-Packard 1982]. 이 공식은 2 센트 이내로 정확한 $ 37614.07를 산출합니다!≈

정리 4는 LN(x)$\ln(x)$이 1/2 ulp 이내로 근사 한다고 가정합니다 . 이 문제는 해결 될 때이다 X는 작고, LN(1 x는) 닫지 LN에 $(1+X)$ 때문에 1 X 의 하위 비트의 정보를 잃어버린 X를 . 즉, $\ln(1+x)$ 의 계산 된 값은 .⊕⊕ $x \ll 1$

정리 4

*ln (1 + x)*이 공식을 사용하여 계산되는 경우

$$\ln(1+x) = \begin{cases} x & \text{for } 1 \oplus x = 1 \\ \dfrac{x\ln(1+x)}{(1+x)-1} & \text{for } 1 \oplus x \neq 1 \end{cases}$$

0 x <3/4 일 때 상대 오차는 최대 *5* 이며, 가드 디지트 e <0.1로 빼기가 수행되고 ln이 1/2 ulp 이내로 계산됩니다.ε ≤

이 공식은 x의 모든 값에 대해 작동 하지만 $x \ll 1$ 순진한 공식 $\ln(1+x)$ 에서 치명적인 취소가 발생하는 에서만 흥미 롭습니다 . 수식이 신비스러워 보일 수 있지만 작동 이유에 대한 간단한 설명이 있습니다. $\ln(1+x)$를 다음과 같이 씁니다.

$$x\left(\frac{\ln(1+x)}{x}\right) = x\mu(x)$$

.

왼손 인자는 정확하게 계산할 수 있지만 오른손 인자 $\mu(x) = \ln(1+x)/x$ 는 x에 1을 더할 때 큰 반올림 오류가 발생 합니다. 그러나 $\ln(1+x)$ x 이므로 μ 는 거의 일정 합니다. 따라서 x를 약간 변경해도 많은 오류가 발생하지 않습니다. 즉, 만약 이면 계산 은 x $\mu(x) = \ln(1+x)$에 대한 좋은 근사치가 될 것 입니다. 값 있는가 하는 과 정확하게 계산 될 수있다? 있다; 즉 = (1 x≈ $\tilde{x} = x\,x\mu(\tilde{x})\,\tilde{x}\,\tilde{x}\,\tilde{x}+1\,\tilde{x}$⊕) ⊖ 1, 왜냐하면 $1+\tilde{x}$ 는 정확히 1 x와 같기 때문 입니다.⊕

이 섹션의 결과는 가드 디지트가 정확히 알려진 근처의 양을 빼면 정확도를 보장한다고 말함으로써 요약 할 수 있습니다 (양성 취소). 때때로 부정확 한 결과를 제공하는 공식은 무해한 취소를 사용하여 훨씬 더 높은 수치 정확도를 갖도록 다시 작성할 수 있습니다. 그러나이 절차는 가드 디지트를 사용하여 빼기가 수행되는 경우에만 작동합니다. 가드 디지트의 가격은 높지 않습니다. 덧셈기를 1 비트 더 넓게 만들기 만하면되기 때문입니다. 54 비트 배정 밀도 가산기의 경우 추가 비용은 2 % 미만입니다. 이 가격 으로 삼각형의 면적을 계산하기위한 공식 (6) 과 표현식 $\ln(1+x)$ 과 같은 많은 알고리즘을 실행할 수 있습니다.). 대부분의 최신 컴퓨터에는 가드 디지트가 있지만 그렇지 않은 경우 (예 : Cray 시스템)가 있습니다.

정확히 반올림 된 작업

부동 소수점 연산이 가드 디지트로 수행되면 정확히 계산 된 다음 가장 가까운 부동 소수점 숫자로 반올림 된 것처럼 정확하지 않습니다. 이러한 방식으로 수행되는 작업은 정확히 반올림 이라고 합니다 . [8] 정리 2 바로 앞의 예는 단일 가드 숫자가 항상 정확하게 반올림 된 결과를 제공하지 않음을 보여줍니다. 이전 섹션에서는 제대로 작동하기 위해 보호 숫자가

필요한 알고리즘의 몇 가지 예를 제공했습니다. 이 섹션에서는 정확한 반올림이 필요한 알고리즘의 예를 제공합니다.

지금까지 반올림의 정의는 제공되지 않았습니다. 반올림은 반올림하는 방법을 제외하고는 간단합니다. 예를 들어 12.5를 12 또는 13으로 반올림해야합니까? 한 학교에서는 10 자리 숫자를 반으로 나누고 {0, 1, 2, 3, 4}는 내림하고 {5, 6, 7, 8, 9}는 올림합니다. 따라서 12.5는 13으로 반올림됩니다. 이것이 Digital Equipment Corporation의 VAX 컴퓨터에서 반올림이 작동하는 방식입니다. 또 다른 사고 방식은 5로 끝나는 숫자가 가능한 두 반올림의 중간이므로 시간의 절반을 내림하고 나머지 절반을 반올림해야한다고 말합니다. 반올림 된 결과의 최하위 숫자가 짝수 여야하기 위해이 50 % 동작을 얻는 한 가지 방법입니다. 따라서 2는 짝수이기 때문에 12.5는 13이 아닌 12로 반올림됩니다. 다음 중 가장 좋은 방법은 무엇입니까? 반올림 또는 반올림? Reiser와 Knuth [1975]는 짝수보다 라운드를 선호하는 다음과 같은 이유를 제공합니다.

정리 5

*x*와 *y*를 부동 소수점 숫자라고하고 $x_0 = x$, $x_1 = (x_0 \; y) \; y$, ... , $x_n = (x_{n-1} \; y) \; y$를 정의합니다. 경우 와 정확히에도 라운드에 사용 반올림 다음 중 X_N = 모든 *N* 개의 *X* 또는 *X* 대 $_N = X_{(1)}$ 모든 *N* 대 1 Z ⊖ ⊕ ⊖ ⊕⊕⊖ ≥

이 결과를 명확히하기 위해 β= 10, *p* = 3을 고려 하고 *x* = 1.00, *y* = -.555로 둡니다. 반올림 할 때 시퀀스는

x_0 ⊖ y = 1.56, x_1 = 1.56 ⊖ .555 = 1.01, x_1 ⊖ y = 1.01 ⊕.555 = 1.57,

그리고 각 연속 값 X_N 까지 0.01 씩 증가, X_N = 9.45 (N ≤845) [9] . 짝수로 반올림에서 x_n 은 항상 1.00입니다. 이 예는 반올림 규칙을 사용할 때 계산이 점차 위로 드리프트 될 수있는 반면, 반올림을 사용하여 정리를 사용하면 이것이 일어날 수 없다고 말합니다. 이 문서의 나머지 부분에서는 짝수에서 짝수로 사용됩니다.

정확한 반올림의 한 적용은 다중 정밀도 산술에서 발생합니다. 더 높은 정밀도에 대한 두 가지 기본 접근 방식이 있습니다. 한 가지 접근 방식은 단어 배열에 저장되는 매우 큰 부호를 사용하여 부동 소수점 숫자를 나타내고 어셈블리 언어로 이러한 숫자를 조작하기위한 루틴을 코딩합니다. 두 번째 접근 방식은 고정밀 부동 소수점 숫자를 일반 부동 소수점 숫자의 배열로 나타내며, 배열의 요소를 무한 정밀도로 추가하면 고정밀 부동 소수점 숫자가 복구됩니다. 여기서 논의 할 두 번째 접근 방식입니다. 부동 소수점 숫자 배열을 사용하는 장점은 상위 수준 언어로 이식 가능하게 코딩 할 수 있지만 정확히 반올림 된 산술이 필요하다는 것입니다.

이 시스템에서 곱셈의 핵심은 곱 *x* y를 합계로 나타내는 것입니다. 여기서 각 합계는 *x* 및 *y* 와 동일한 정밀도를 갖습니다 . 이것은 *x* 와 *y* 를 분할하여 수행 할 수 있습니다 . $x = x_h + x_l$ 및 $y = y_h + y_l$을 쓰면 정확한 곱은 다음과 같습니다.

$$x\,y = x_h\,y_h + x_h\,y_l + x_l\,y_h + x_l\,y_l .$$

경우 *X* 및 *Y*가 있는 *P*의 비트 significands를 상기 피가수는 것 *(P)*의 비트 significands가 제공 *X*의 $_L$, *X*의 $_H$, Y_H , Y_L을 이용하여 표현 될 수있다 [*P*가 / 2] 비트. 때 *p*가 짝수, 분할을 쉽게 찾을 수 있습니다. 숫자 $x_0 . x_1 \ldots x_{p-1}$ 은 x_0 의 합으로 쓸 수 있습니다 . $x_1 \ldots x_{p/2-1}$그리고 $0.0 \ldots 0\, x_{p/2} \ldots x_{p-1}$. 때 *p*가 홀수,이 간단한 분할 방법은 작

동하지 않습니다. 그러나 추가 비트는 음수를 사용하여 얻을 수 있습니다. 예를 들어, β= 2, *p* = 5, *x* = .10111이면 *x* 는 x_h = .11 및 x_l = -.00001 로 분할 될 수 있습니다 . 숫자를 분할하는 방법은 여러 가지가 있습니다. 계산하기 쉬운 분할 방법은 Dekker [1971]에 의한 것이지만 하나 이상의 가드 숫자가 필요합니다.

정리 6

*p*가β 2 보다 크더라도 *p*가 짝수라는 제한과 함께 부동 소수점 정밀도로 간주하고 부동 소수점 연산이 정확히 반올림된다고 가정합니다. 그런 다음 *k* = [p / 2] 가 정밀도의 절반 *(*반올림*)*이고 *m* = ^{k} + 1 이면 *x*는 *x* = x_h + x_l 로 분할 될 수 있습니다 . 여기서β

$x_h = (m \ x) \ (m \ x \ x), \ x_l = x \ x_h$,⊗⊖ ⊗⊖ ⊖

각 x_i 는 [p / 2] 비트의 정밀도를 사용하여 표현할 수 있습니다.

이 정리가 예제에서 어떻게 작동하는지 보려면 let β= 10, *p* = 4, *b* = 3.476, *a* = 3.463, *c* = 3.479입니다. 그러면 가장 가까운 부동 소수점 수로 반올림 된 b^2 - *ac* 는 .03480이고, *b b* = 12.08, *a c* = 12.05이므로 b^2 - *ac* 의 계산 된 값 은 .03입니다. 이것은 480 ulps의 오류입니다. 정리 6을 사용하여 작성 *B* 0.024 - 3.5 = = 3.5 - .037 및 *C* , 0.021 - 3.5 = *b*를 ^{2} 3.5하게 ^{2} - 2⊗ ⊗ × 3.5 × .024 + .024 ^{2} . 각 합계 는 정확하므로 b^2 = 12.25-.168 + .000576이며, 여기서 합계는 평가되지 않은 상태로 남아 있습니다. 마찬가지로 *ac* = 3.5 $^{2-}$ (3.5 × .037 + 3.5 × .021) + .037 × .021 = 12.25-.2030 +.000777입니다. 마지막으로,이 두 계열 항을 항으로 빼면 b^2 - *ac* 에 대한 추정치 가 0 ⊕ .0350 ⊖ .000201 = .03480이며 이는 정확히 반올림 된 결과와 동일합니다. 정리 6이 실제로 정확한 반올림을 필요로한다는 것을 보여주기 위해 *p* = 3, β= 2 및 *x*를 고려하십시오.= 7. 그런 다음 *m* = 5, *mx* = 35, *m x* = 32. 단일 가드 숫자로 뺄셈을 수행하면 (*m x*) *x* = 28입니다. 따라서 x_h = 4 및 x_l = 3이므로 x_l 은 [*p / 2*] = 1 비트로 표현할 수 없습니다 .⊗ ⊗ ⊖

정확한 반올림의 마지막 예로서 *m* 을 10으로 나누는 것을 고려하십시오 . 결과는 일반적으로 *m* / 10 과 같지 않은 부동 소수점 숫자입니다 . 경우 β승산 = 2, *m* 복원 10 / 10 *m*를 , 실제 사용되는 라운딩 제공한다. 사실 더 일반적인 사실 (Kahan 덕분에)이 사실입니다. 그 증거는 독창적이지만 그러한 세부 사항에 관심이없는 독자 는 IEEE 표준 섹션으로 건너 뛸 수 있습니다 .

정리 7

= 2 일 때 , *m* 과 *n* 이 | *m* |이있는 정수이면 < 2^{p-1} 및 *n* 은 부동 소수점 연산이 정확히 반올림되는 경우 특수한 형식 *n* = 2^i + 2^j 이고 *(m n) n* = *m*입니다.β⊘ ⊗

증명

2의 거듭 제곱으로 스케일링하는 것은 중요도가 아닌 지수 만 변경하므로 무해합니다. 경우 *Q* = *m* / *n*은 다음 확장 *N* 이산화황 것을 $^{P - 1}$ *N* <2^P 스케일 *m* 되도록 1/2 < *Q* <1 따라서, (2) $^{P - 2}$ < *m* <2^P . 이후 *m*을 갖는 *P* 하위 비트를, 상기 이진 소수점의 오른쪽에

있는 많아야 하나의 비트에 갖는다. *m* 의 부호를 변경하는 것은 무해하므로 *q* > 0 이라고 가정합니다 .≤
경우 $\tilde{q}$ =는 *m* ∅ *N* , 정리를 증명하는 것을 나타내는 요구

(9) $|n\tilde{q} - m| \le \frac{1}{4}$

때문이다 *m*은 이진 소수점 많아야 1 비트 권리가 있으므로, $N^{\tilde{q}}$ 윌 라운드 *m* . 중간 사건을 처리하려면 | $n^{\tilde{q}}$ - *m* | = 1/4, 초기 스케일되지 않은 *m* 은 | *m* | <2 $^{p-1}$, 하위 비트는 0 이므로 스케일링 된 *m* 의 하위 비트 도 0입니다. 따라서 중간 케이스는 *m*으로 반올림됩니다 .
q = 라고 가정합니다 . $q_1 q_2 \ldots$, let $\hat{q}$ =. $q_1 q_2 \ldots q_p$ 1. 추정하려면 | $n^{\tilde{q}}$ - *m* |, 첫 번째 계산

| $\hat{q}$ - Q | = | *N* / 2 $^{p+1}$ - *m* / *n* |,

여기서 *N* 은 홀수 정수입니다. 이후 *N* = 2 ^{I} + 2 ^{J} 2 $^{P-}$ ^{1} *N* <2 ^{P} , 그 있어야 *N* = 2 인 $^{(P)-1}$ + 2 ^{K} 일부 유전율을위한 *P* - 2, 이에≤ ≤

$$|\tilde{q} - q| = \left|\frac{nN - 2^{p+1}m}{n2^{p+1}}\right| = \left|\frac{(2^{p-1-k}+1)N - 2^{p+1-k}m}{n2^{p+1-k}}\right|.$$

분자는 정수이고 *N* 은 홀수이므로 실제로 홀수 정수입니다. 그러므로,

| $\hat{q}$ - Q | ≥1 / (*n* 2 $^{p+1-k}$).

가정 *Q*를 < $\hat{q}$ (케이스 *Q*는 > $\hat{q}$ 와 유사하다). [10] 그런 다음 $n^{\tilde{q}}$ < *m* ,

$$|mn^{\tilde{q}}| = mn^{\tilde{q}} = n(q - \tilde{q}) = n(q - (\hat{q} - 2^{-p-1})) \le n\left(2^{-p-1} - \frac{1}{n2^{p+1-k}}\right)$$

$$= (2^{p-1} + 2^{k})\,2^{-p-1} - 2^{-p-1} + k = \frac{1}{4}$$

이것은 (9) 정리를 확립 하고 증명합니다. [11] z

정리는 β2 ^{i} + 2 ^{j} 가 ^{i} + ^{j} 로 바뀌는 한 모든 밑에 대해 참입니다 . 으로 커질수록, 그러나, 형태의 분모 ^{내가} + ^{j는} 멀리 멀리 떨어져 있습니다.ββββββ

우리는 이제 기본적인 산술 연산이 필요한 것보다 약간 더 많은 반올림 오류를 발생시키는 것이 중요합니까?라는 질문에 답할 수있는 위치에 있습니다. 그 대답은 그것이 중요하다는 것입니다. 정확한 기본 연산을 통해 공식이 작은 상대 오류가 있다는 의미에서 "정확"하다는 것을 증명할 수 있기 때문입니다. 취소 섹션 에서는 이러한 의미에서 올바른 결과를 생성하기 위해 보호 숫자가 필요한 여러 알고리즘에 대해 설명했습니다. 그러나 이러한 공식에 대한 입력이 부정확 한 측정을 나타내는 숫자이면 정리 3과 4의 경계가 덜 흥미로워집니다. 그 이유는 *x* 와 *y*의 경우 양성 취소 *x* - *y* 가 치명적일 수 있기 때문입니다.일부 측정 수량에 대한 근사치입니다. 그러나 정확한 연산은 데이터가 정확하지 않은 경우에도 유용합니다. 정리 6과 7에서 논의 된 것과 같은 정확한 관계를 설정할 수 있기 때문입니다. 이는 모든 부동 소수점 변수가 실제 값에 대한 근사치 일지라도 유용합니다.

IEEE 표준

부동 소수점 계산을위한 두 가지 IEEE 표준이 있습니다. IEEE 754는 단 정밀도의 경우 β = 2, p = 24 , 배정 밀도의 경우 p = 53을 요구하는 이진 표준입니다 [IEEE 1987]. 또한 단 정밀도 및 배정 밀도로 비트의 정확한 레이아웃을 지정합니다. IEEE 854는 β= 2 또는 β= 10을 허용하고 754와 달리 부동 소수점 숫자를 비트로 인코딩하는 방법을 지정하지 않습니다 [Cody et al. 1984]. p 에 대한 특정 값이 필요하지 않지만 대신 단 정밀도 및 배정 밀도 에 대해 허용 가능한 p 값에 대한 제약 조건을 지정합니다 . *IEEE* 표준 이라는 용어 는 두 표준에 공통된 속성을 설명 할 때 사용됩니다.

이 섹션에서는 IEEE 표준에 대해 살펴 봅니다. 각 하위 섹션에서는 표준의 한 측면과 포함된 이유에 대해 설명합니다. IEEE 표준이 최상의 부동 소수점 표준이라고 주장하는 것이이 백서의 목적이 아니라 주어진 표준을 수용하고 그 사용에 대한 소개를 제공하는 것입니다. 자세한 내용은 표준 자체를 참조하십시오 [IEEE 1987; Cody et al. 1984].

형식 및 운영

베이스

IEEE 854가 β= 10을 허용하는 이유는 분명합니다 . 10 진수는 인간이 숫자를 교환하고 생각하는 방법입니다. β= 10을 사용 하는 것은 각 연산의 결과가 계산기에 10 진수로 표시되는 계산기에 특히 적합합니다.

IEEE 854에서 밑 수가 10이 아닌 경우 2 여야하는 이유는 여러 가지가 있습니다. 상대 오류 및 Ulps 섹션에서는 한 가지 이유를 언급 β했습니다. 반올림 오류가 .5 ulp이기 때문에 오류 분석 결과 가 2 일 때 훨씬 더 엄격합니다. β상대 오차로 계산할 때 의 요인에 의해 흔들리고 , 상대 오차를 기반으로하면 오류 분석이 거의 항상 더 간단합니다. 관련된 이유는 큰 기지에 대한 효과적인 정밀도와 관련이 있습니다. = 2, p = 4 와 비교하여 β= 16, p = 1을 고려하십시오 . 두 시스템 모두 4 비트의 유효 값을가집니다. 15/8의 계산을 고려하십시오. 경우 (2) = 15 1.111로 표현 × $2^{(3)}$ 과 같은 1.111 15/8 ×ββ2^{0} . 따라서 15/8은 정확합니다. 그러나 β= 16 일 때 15는 $F \times 16^{0}$ 으로 표시되며 , 여기서 F 는 15 의 16 진수입니다. 그러나 15/8은 1×16^{0} 으로 표시되며 1 비트 만 정확합니다. 일반적으로 16 진수 는 최대 3 비트를 잃을 수 있으므로 p 16 진수 의 정밀도는 $4p$ 이진 비트가 아닌 $4p$ -3 만큼 낮은 유효 정밀도를 가질 수 있습니다 . 의 큰 값 에는 이러한 문제가 있으므로 IBM이 선택한 이유ββ= 시스템 / 370의 경우 16? 확실히 IBM만이 알고 있지만 두 가지 가능한 이유가 있습니다. 첫 번째는 지수 범위 증가입니다. system / 370의 단 정밀도는 β= 16, p = 6입니다. 따라서 significand에는 24 비트가 필요합니다. 32 비트에 맞아야하므로 지수에 대해 7 비트, 부호 비트에 대해 1 비트가 남습니다. 따라서 표현 가능한 숫자의 크기는 about 16^{-2^6} ~ about 16^{2^6} = 2^{2^8} 입니다. β= 2 일 때 유사한 지수 범위를 얻으려면 지수 9 비트가 필요하고 유효 숫자에 22 비트 만 남습니다. 그러나 β= 16 일 때 유효 정밀도는 $4p$ -3 = 21 비트 만큼 낮을 수 있다는 점이 지적되었습니다 . 더 나쁜 경우β = 2 (이 섹션의 뒷부분에서 설명하는대로) 추가 비트의 정밀도를 얻을 수 있으므로 β= 2 머신은 β= 16 머신 의 21-24 비트 범위와 비교할 23 비트의 정밀도를 갖습니다 .

β= 16 을 선택하는 또 다른 가능한 설명은 이동과 관련이 있습니다. 두 개의 부동 소수점 숫자를 더할 때 지수가 다르면 기수 포인트를 정렬하기 위해 유효 숫자 중 하나를 이동해야하므로 연산 속도가 느려집니다. 에서 β= 16, P = 1에있어서, 모든 숫자 1 ~ 15은 같은 지수를 가지며, 임의의 추가시 시프 팅되도록 (요구 $\frac{15}{2}$) 세트로부터 고유 번호 105 가능 쌍 =. 그러나 β= 2, p = 4 시스템에서이 숫자는 0에서 3까지의 지수를 가지며 105 쌍 중 70 개에 대해 이동이 필요합니다.

대부분의 최신 하드웨어에서 피연산자 하위 집합에 대한 이동을 피하여 얻은 성능은 무시할 수 있으므로 β= 2 의 작은 흔들림 이 선호되는 기준이됩니다. β= 2 를 사용하는 또 다른 이점은 추가로 의미를 얻을 수있는 방법이 있다는 것입니다. [12] 부동 소수점 숫자는 항상 정규화되기 때문에 Significand의 최상위 비트는 항상 1이며이를 나타내는 저장 공간을 낭비 할 이유가 없습니다. 이 트릭을 사용하는 형식에는 숨겨진 비트 가 있다고합니다 . 이것은 0에 대한 특별한 규약이 필요하다는 것이 부동 소수점 형식 에서 이미 지적되었습니다. 주어진 방법은 e_{min} -1 의 지수 와 모든 0의 유효 값은 $1.0 \times 2^{e_{min}-1}$, 오히려 0.

IEEE 754 단 정밀도는 부호에 1 비트, 지수에 8 비트, 유효 숫자에 23 비트를 사용하여 32 비트로 인코딩됩니다. 그러나 숨겨진 비트를 사용하므로 23 비트 만 사용하여 인코딩 된 경우에도 유효 숫자는 24 비트 (p = 24)입니다.

정도

IEEE 표준은 단일, 이중, 단일 확장 및 이중 확장의 네 가지 정밀도를 정의합니다. IEEE 754에서 단 정밀도 및 배정 밀도는 대부분의 부동 소수점 하드웨어가 제공하는 것과 대략 일치합니다. 단 정밀도는 단일 32 비트 워드, 배정 밀도 2 개의 연속 32 비트 워드를 차지합니다. 확장 정밀도는 최소한 약간의 추가 정밀도와 지수 범위를 제공하는 형식입니다 (표 D-1).

표 D-1 IEEE 754 형식 매개 변수

매개 변수	체재			
	단일	단일 확장	더블	이중 확장
피	24	≥ 32	53	≥ 64
e_{max}	+127	≥ 1023	+1023	> 16383
$e_{분}$	-126	≤ -1022	-1022	≤ -16382
지수 너비 (비트)	8	≤ 11	11	≥ 15
형식 너비 (비트)	32	≥ 43	64	≥ 79

IEEE 표준은 확장 정밀도가 제공하는 추가 비트 수에 대한 하한 만 지정합니다. 최소 허용 이중 확장 형식은 표에 79 비트를 사용하여 표시되어 있지만 *80* 비트 형식 이라고도합니다 . 그 이유는 확장 정밀도의 하드웨어 구현은 일반적으로 숨겨진 비트를 사용하지 않으므로 79 비트가 아닌 80 비트를 사용하기 때문입니다. [13]

이 표준은 확장 정밀도에 가장 중점을 두어 배정 밀도에 대한 권장 사항은 없지만 구현시 지원되는 가장 광범위한 기본 형식에 해당하는 확장 형식을 지원해야한다고 강력히 권장합니다 .

확장 정밀도에 대한 한 가지 동기는 계산기에서 비롯됩니다. 계산기는 종종 10 자리를 표시하지만 내부적으로는 13 자리를 사용합니다. 13 자리 중 10 자리 만 표시하면 계산기는 지수, 코사인 등을 10 자리의 정확도로 계산하는 "블랙 박스"로 사용자에게 표시됩니다. 계산기가 exp, log 및 cos와 같은 함수를 합리적인 효율성으로 10 자리 이내로 계산하려면 몇 개의 추가 숫자가 필요합니다. 마지막 자리에서 500 단위의 오차로 로그를 근사하는 단순한 합리적 표현을 찾는 것은 어렵지 않습니다. 따라서 13 자리로 계산하면 10 자리로

정답이됩니다. 이 여분의 3 자리 숫자를 숨기면 계산기는 운영자에게 간단한 모델을 제공합니다.

IEEE 표준의 확장 된 정밀도는 유사한 기능을 제공합니다. 이를 통해 라이브러리는 단일 (또는 이중) 정밀도로 약 .5 ulp 내에서 수량을 효율적으로 계산할 수 있으며, 해당 라이브러리 사용자에게 간단한 모델, 즉 간단한 곱하기 또는 로그 호출과 같은 각 기본 작업이 다음을 반환합니다. 약 .5 ulp 내에서 정확한 값. 그러나 확장 된 정밀도를 사용하는 경우 사용이 사용자에게 투명하게 표시되는지 확인하는 것이 중요합니다. 예를 들어 계산기에서 표시된 값의 내부 표현이 디스플레이와 동일한 정밀도로 반올림되지 않은 경우 추가 작업의 결과는 숨겨진 숫자에 따라 달라지며 사용자에게 예측할 수없는 것처럼 보입니다.

확장 된 정밀도를 더 자세히 설명하려면 IEEE 754 단 정밀도와 10 진수 간의 변환 문제를 고려하십시오. 이상적으로, 단 정밀도 숫자는 십진수를 다시 읽을 때 단 정밀도 숫자를 복구 할 수 있도록 충분한 자릿수로 인쇄됩니다. 단 정밀도 이진수를 복구하는 데 십진수 9 자리이면 충분합니다 (2 진수에서 십진수로 변환 섹션 참조).). 10 진수를 고유 한 이진 표현으로 다시 변환 할 때 1ulp만큼 작은 반올림 오류는 잘못된 답을 제공하기 때문에 치명적입니다. 다음은 효율적인 알고리즘을 위해 확장 된 정밀도가 필수적인 상황입니다. 단일 확장을 사용할 수있는 경우 십진수를 단 정밀도 이진수로 변환하는 매우 간단한 방법이 있습니다. 먼저 소수점을 무시하고 정수 N 으로 10 진수 9 자리를 읽습니다 . 에서 TABLE D-1 , $P \geq 32$, 10 이후 $2^{9} < 10^{32} \approx 4.3 \times 10^{9}$, N은 정확히 단일 확장으로 표현 될 수 있다. 다음으로 적절한 전원 (10) 찾을 P를 N 을 스케일링하는 데 필요합니다 . 이것은 (지금까지) 무시 된 소수점의 위치와 함께 십진수의 지수의 조합이 될 것입니다. 컴퓨팅 $10^{|P|}$. 만약 $|P| \leq 13$이면 $10^{13} = 2^{13} 5^{13}$, $5^{13} < 2^{32}$ 이므로 정확하게 표현됩니다 . 마지막으로 곱하기 (또는 $p < 0$이면 나누기) N 및 $10^{|P|}$. 이 마지막 작업이 정확히 수행되면 가장 가까운 이진수가 복구됩니다. 섹션 2 진수에서 10 진수로 변환마지막 곱하기 (또는 나누기)를 정확히 수행하는 방법을 보여줍니다. 따라서 $|P| \leq 13$도 13에서 단일 확장 형식을 사용하면 9 자리 십진수를 가장 가까운 이진수로 변환 할 수 있습니다 (즉, 정확히 반올림). 만약 $|P| > 13$ 일 경우, 단일 확장은 위의 알고리즘이 항상 정확히 반올림 된 이진 등가물을 계산하기에 충분하지 않지만 Coonen [1984]은 이진수를 십진수로 변환하고 역으로 변환하면 원래 이진수를 복구하는 것으로 충분하다는 것을 보여줍니다. .

배정 밀도가 지원되는 경우 위의 알고리즘은 단일 확장이 아닌 배정 밀도로 실행되지만 배정 밀도를 17 자리 십진수로 변환하고 역방향으로 되돌리려면 이중 확장 형식이 필요합니다.

멱지수

지수는 양수 또는 음수 일 수 있으므로 부호를 나타내는 방법을 선택해야합니다. 부호있는 숫자를 나타내는 두 가지 일반적인 방법은 부호 / 크기와 2의 보수입니다. 부호 / 크기는 IEEE 형식에서 부호의 부호에 사용되는 시스템입니다. 1 비트는 부호를 유지하는 데 사용되고 나머지 비트는 숫자의 크기를 나타냅니다. 2의 보수 표현은 정수 산술에서 자주 사용됩니다. 이 방식에서 $[-2^{p-1}, 2^{p-1}-1]$ 범위의 숫자는 모듈로 2^{p}에 합동하는 가장 작은 음이 아닌 숫자로 표시됩니다 .

IEEE 이진 표준은 지수를 나타내는 데 이러한 방법 중 하나를 사용하지 않고 대신 편향된 표현을 사용합니다 . 지수가 8 비트로 저장되는 단 정밀도의 경우 바이어스는 127입니다 (배정 밀도의 경우 1023). 이것이 의미하는 바는 만약 k is가 부호없는 정수로 해석되는 지수 비트의 값이면 부동 소수점 숫자의 지수 k 는 *-127* 이라는 것입니다. 이것은 편향 지수와 구별하기 위해 종종 편향 지수 라고합니다 k .

표 D-1을 참조하면 단 정밀도는 $e_{max} = 127$이고 $e_{min} = -126$입니다. 갖는 이유 | 전자 $_{분}$ | < e_{max} 는 가장 작은 숫자의 역수가 $(1/2^{e_{min}})$ 오버플로되지 않도록합니다. 가장 큰 숫자의 역수가 언더 플로되는 것은 사실이지만 일반적으로 언더 플로는 오버플로보다 덜 심각합니다. 단면 베이스 라고 설명 전자 $_{분}$ - 1 0을 나타내는 사용되며, 특별 수량 에 대한 사용 소개 전자 $_{최대}$ + 1에서 IEEE 단일 정밀도,이 수단은 편향 지수 사이의 범위 일 것으로e_{min} -1 = -127 및 e_{max} + 1 = 128 인 반면, 편향되지 않은 지수의 범위는 0에서 255 사이이며, 이는 정확히 8 비트를 사용하여 표현할 수있는 음이 아닌 숫자입니다.

운영

IEEE 표준에서는 더하기, 빼기, 곱하기 및 나누기의 결과가 정확히 반올림되어야합니다. 즉, 결과를 정확하게 계산 한 다음 가장 가까운 부동 소수점 수로 반올림해야합니다 (반올림에서 짝수로 사용). Guard Digits 섹션에서는 두 부동 소수점 숫자의 정확한 차이 또는 합을 계산하는 것이 지수가 실질적으로 다를 때 매우 비쌀 수 있다고 지적했습니다. 이 섹션에서는 상대적 오차가 작다는 것을 보장하면서 차이를 계산하는 실용적인 방법을 제공하는 가드 디지트를 소개했습니다. 그러나 단일 가드 숫자로 계산한다고해서 정확한 결과를 계산 한 다음 반올림하는 것과 항상 동일한 답이 제공되는 것은 아닙니다. 두 번째 가드 숫자와 세 번째 스티커 를 도입하여비트, 차이는 단일 가드 디지트보다 약간 더 많은 비용으로 계산할 수 있지만 그 결과는 차이가 정확히 계산 된 다음 반올림 된 것과 같습니다 [Goldberg 1990]. 따라서 표준을 효율적으로 구현할 수 있습니다.

산술 연산의 결과를 완전히 지정하는 한 가지 이유는 소프트웨어의 이식성을 향상시키기 위해서입니다. 프로그램이 두 시스템간에 이동하고 둘 다 IEEE 산술을 지원할 때 중간 결과가 다른 경우 산술의 차이가 아니라 소프트웨어 버그 때문이어야합니다. 정확한 사양의 또 다른 장점은 부동 소수점에 대해 쉽게 추론 할 수 있다는 것입니다. 부동 소수점에 대한 증명은 여러 종류의 산술에서 발생하는 여러 경우를 처리 할 필요없이 충분히 어렵습니다. 정수 프로그램이 올바른 것으로 입증 될 수있는 것처럼 부동 소수점 프로그램도 마찬가지입니다.이 경우 결과의 반올림 오류가 특정 범위를 충족한다는 것이 입증되었습니다. 정리 4는 그러한 증명의 예입니다. 추론되는 작업이 정확하게 지정되면 이러한 증명이 훨씬 쉬워집니다. 알고리즘이 IEEE 산술에 대해 올바른 것으로 입증되면 IEEE 표준을 지원하는 모든 시스템에서 올바르게 작동합니다.

Brown [1981]은 대부분의 기존 부동 소수점 하드웨어를 포함하는 부동 소수점에 대한 공리를 제안했습니다. 그러나이 시스템의 증명은 모든 하드웨어에없는 기능이 필요한 취소 및 정확히 반올림 된 작업 섹션의 알고리즘을 확인할 수 없습니다 . 더욱이 Brown의 공리는 단순히 정확하게 수행 한 다음 반올림 할 작업을 정의하는 것보다 더 복잡합니다. 따라서 Brown의 공리에서 정리를 증명하는 것은 일반적으로 연산이 정확히 반올림되었다고 가정하는 것보다 더 어렵습니다.

부동 소수점 표준이 다루어야하는 연산에 대한 완전한 합의가 없습니다. 기본 연산 +,-, × 및 / 외에도 IEEE 표준은 제곱근, 나머지 및 정수와 부동 소수점 간의 변환이 올바르게 반올림되도록 지정합니다. 또한 내부 형식과 십진수 간의 변환이 올바로 반올림되어야합니다 (매우 큰 숫자 제외). Kulisch와 Miranker [1986]는 정확하게 지정된 작업 목록에 내부 제품을 추가 할 것을 제안했습니다. 그들은 내부 제품이 IEEE 산술로 계산 될 때 최종 답이 매우 잘못 될 수 있음을 지적합니다. 예를 들어 합계는 내적의 특별한 경우이고 합계는 $((2 \times 10^{-30} + 10^{30}) - 10^{30}) - 10^{-30}$ 은 정확히 10^{-30} 과 같지만 IEEE 산술을 사용하는 컴퓨터에서는 계산 된 결과가 -10^{-30} 입니다. 빠른 승수를 구현하는 것보다 적은 하드웨어로 내적을 1ulp 이내로 계산할 수 있습니다 [Kirchner and Kulish 1987].[14] [15]

표준에 언급 된 모든 연산은 10 진수와 2 진수 간의 변환을 제외하고 정확하게 반올림되어야합니다. 그 이유는 변환을 제외한 모든 연산을 정확하게 반올림하는 효율적인 알고리즘이 알려져 있기 때문입니다. 변환의 경우 가장 잘 알려진 효율적인 알고리즘은 정확히 반올림 된 알고리즘보다 약간 더 나쁜 결과를 생성합니다 [Coonen 1984].

IEEE 표준은 테이블 제작자의 딜레마 때문에 초월 함수를 정확하게 반올림 할 필요가 없습니다 . 설명을 위해 지수 함수의 표를 4 자리로 만든다고 가정합니다. 그러면 exp (1.626) = 5.0835입니다. 5.083 또는 5.084로 반올림해야합니까? exp (1.626)을보다 신중하게 계산하면 5.08350이됩니다. 그리고 5.083500. 그리고 5.0835000. exp는 초월 적이므로 exp (1.626)가 5.083500 ... 0 *ddd* 인지 5.0834999 ... 9 *ddd* 인지 구별하기 훨씬 전에 임의로 진행될 수 있습니다 .. 따라서 초월 함수의 정밀도가 마치 무한 정밀도로 계산된 다음 반올림 된 것과 동일하도록 지정하는 것은 실용적이지 않습니다. 또 다른 접근 방식은 초월 함수를 알고리즘 적으로 지정하는 것입니다. 그러나 모든 하드웨어 아키텍처에서 잘 작동하는 단일 알고리즘은없는 것 같습니다. 합리적 근사, CORDIC, [16] 및 대형 테이블은 현대 기계에서 초월을 계산하는 데 사용되는 세 가지 다른 기술입니다. 각각은 서로 다른 종류의 하드웨어에 적합하며 현재 단일 알고리즘은 현재의 광범위한 하드웨어에서 허용되는 방식으로 작동하지 않습니다.

특별 수량

일부 부동 소수점 하드웨어에서 모든 비트 패턴은 유효한 부동 소수점 숫자를 나타냅니다. IBM System / 370이 그 예입니다. 반면에 VAX™ 는 예약 된 피연산자 라고하는 특수한 숫자를 나타 내기 위해 일부 비트 패턴을 예약합니다 . 이 아이디어는 CDC 6600으로 거슬러 올라갑니다. CDC 6600은 특수 수량 `INDEFINITE`및 `INFINITY`.

IEEE 표준은 이러한 전통을 이어 가며 NaN (*Not a Number*)과 무한대를 가지고 있습니다. 특별한 양이 없으면 계산을 중단하는 것 외에 음수의 제곱근을 취하는 것과 같은 예외적 인 상황을 처리 할 좋은 방법이 없습니다. IBM System / 370 FORTRAN에서 -4와 같은 음수의 제곱근 계산에 대한 기본 조치는 오류 메시지를 인쇄합니다. 모든 비트 패턴은 유효한 숫자를 나타내므로 제곱근의 반환 값은 부동 소수점 숫자 여야합니다. System / 370 FORTRAN의 경우 $\sqrt{-4} = 2$ 반환됩니다. IEEE 산술에서는이 상황에서 NaN이 반환됩니다.

IEEE 표준은 다음과 같은 특수 값을 지정합니다 (표 D-2 참조) : ± 0, 비정규 화 된 숫자, $\pm\infty$ 및 NaN (다음 섹션에서 설명하는 바와 같이 둘 이상의 NaN이 있음). 이러한 특수 값은 모두 $e_{max} + 1$ 또는 $e_{min} -1$의 지수로 인코딩됩니다 (0이 $e_{min} -1$ 의 지수를 갖는다는 것은 이미 지적되었습니다).

표 D-2 IEEE 754 특수 값

멱지수	분수	나타냅니다
$e = e_{분} -1$	$f = 0$	± 0
$e = e_{분} -1$	$f \neq 0$	$0.f \times 2^{e_{min}}$
$e_{min} \le e \le e_{max}$	-	$1.f \times 2^{e}$
$e = e_{최대} + 1$	$f = 0$	$\pm\infty$
$e = e_{최대} + 1$	$f \neq 0$	NaN

NaN

전통적으로 0/0 또는 $\sqrt{-1}$ 계산을 중단시키는 복구 불가능한 오류로 처리되었습니다. 그러나 이러한 상황에서 계산을 계속하는 것이 합당한 예가 있습니다. 함수의 제로 발견 서브 루틴 고려 *F*를 말한다 `zero(f)`. 전통적으로 제로 파인더 는 함수가 정의되고 제로 파인더가 검색 할 간격 [*a* , *b*]을 사용자가 입력해야합니다 . 서브 루틴이 호출되는 즉 `zero(f, a,b)`. 더 유용한 제로 파인더는 사용자가이 추가 정보를 입력 할 필요가 없습니다. 이보다 일반적인 제로 파인더는 함수를 입력하는 것이 당연하고 도메인을 지정해야하는 것이 어색한 계산기에 특히 적합합니다. 그러나 대부분의 제로 파인더에 도메인이 필요한 이유는 쉽게 알 수 있습니다. 제로 파인더는 `f`다양한 값에서 함수 를 조사하여 작업을 수행 합니다. 도메인 외부의 값을 검색 `f`하면에 대한 코드 `f`가 0/0 또는을 계산할 수 $\sqrt{-1}$ 있으며 계산이 중단되어 불필요하게 제로 찾기 프로세스를 중단합니다.

이 문제는 NaN이라는 특수 값을 도입하고 0/0과 같은 식의 계산 $\sqrt{-1}$ 이 중단되지 않고 NaN을 생성하도록 지정하여 방지 할 수 있습니다 . NaN을 유발할 수있는 몇 가지 상황 목록은 표 D-3에 나와 있습니다. 그런 다음 `zero(f)`의 도메인 외부에서 검색하면 `f`에 대한 코드 `f`가 NaN을 반환하고 제로 파인더를 계속할 수 있습니다. 즉, `zero(f)`잘못된 추측을했다고 "처벌"되지 않습니다. 이 예제를 염두에두면 NaN을 일반 부동 소수점 숫자와 결합한 결과가 무엇인지 쉽게 알 수 있습니다. 의 마지막 문장이 있다고 가정 `f`입니다 `return(-b + sqrt(d))/(2*a)`. 경우 *D* <0이면 `f`NaN인지를 반환한다. 이후 *D* <0, `sqrt(d)`NaN인지하고,`-b + sqrt(d)`NaN과 다른 숫자의 합이 NaN이면 NaN이됩니다. 마찬가지로 나누기 연산의 피연산자가 NaN이면 몫은 NaN이어야합니다. 일반적으로 NaN이 부동 소수점 연산에 참여할 때마다 결과는 또 다른 NaN입니다.

표 D-3 NaN을 생성하는 작업

조작	NaN 생산
+	$\infty + (-\infty)$
×	$0 \times \infty$
/	0/0, ∞ / ∞
REM	*x* REM 0, *y* ∞ REM
$\sqrt{}$	$\sqrt{x}$ (*x* < 0 일 때)

사용자가 도메인을 입력 할 필요가없는 제로 솔버를 작성하는 또 다른 방법은 신호를 사용하는 것입니다. 제로 파인더는 부동 소수점 예외에 대한 신호 처리기를 설치할 수 있습니다. 그런 다음 `f`도메인 외부에서 평가되고 예외가 발생하면 제어가 제로 솔버로 반환됩니다. 이 접근 방식의 문제점은 모든 언어가 신호 처리 방법이 다르기 때문에 (만약 방법이있는 경우) 이식성에 대한 희망이 없다는 것입니다.

IEEE 754에서 NaN은 종종 지수 $e_{max} + 1$ 및 0이 아닌 유효 숫자가있는 부동 소수점 숫자로 표시됩니다 . 구현은 시스템에 따른 정보를 중요 항목에 자유롭게 넣을 수 있습니다. 따라서 고유 한 NaN이 아니라 전체 NaN 제품군이 있습니다. NaN과 일반 부동 소수점 숫자가 결합 된 경우 결과는 NaN 피연산자와 동일해야합니다. 따라서 긴 계산의 결과가 NaN이면 시스템 종속 정보는 계산의 첫 번째 NaN이 생성 될 때 생성 된 정보가됩니다. 사

실, 마지막 진술에 대한주의 사항이 있습니다. 두 피연산자가 모두 NaN이면 결과는 해당 NaN 중 하나가되지만 처음 생성 된 NaN이 아닐 수도 있습니다.

무한대

NaN이 0/0과 같은식이 $\sqrt{-1}$ 발생 하거나 발생할 때 계산을 계속하는 방법을 제공하는 것처럼 무한대는 오버플로가 발생할 때 계속하는 방법을 제공합니다. 이것은 단순히 표현 가능한 가장 큰 숫자를 반환하는 것보다 훨씬 안전합니다. 예를 들어, = 10, p = 3, e_{max} = 98 $\sqrt{x^2+y^2}$ 일 때 계산을 고려해보십시오 . x = 3 × 10^{70} 및 y = 4 × 10^{70} 이면 x^2 가 오버플로되고 9.99 × 10^{98} 로 대체됩니다. . 마찬가지로 y^2 및 x^2β+ y^2 는 차례로 오버플로되며 9.99 × 10^{98} 로 대체됩니다 . 따라서 최종 결과는 $\sqrt{9.99\times10^{98}} = 3.16\times10^{49}$ 이며 이는 매우 잘못되었습니다. 정답은 5 × 10^{70} 입니다. IEEE 산술 연산의 결과 , X^2 되어 ∞, 그대로 Y^2 , X^2 + $Y^{(2)}$ 와 $\sqrt{x^2+y^2}$. 따라서 최종 결과는 ∞이며 정답 근처에없는 일반적인 부동 소수점 숫자를 반환하는 것보다 안전합니다. [17]

0을 0으로 나누면 NaN이됩니다. 그러나 0이 아닌 숫자를 0으로 나눈 값은 무한대를 반환합니다. 1/0 = ∞, -1/0 =- ∞. 구별의 이유는 이것이다 : 만약 x 가 어떤 한계에 가까워질 때 $f(x)$ →0과 $g(x)$ →0 이면 $f(x)/g(x)$는 어떤 값을 가질 수 있습니다. 예를 들어, $f(x)$ = sin x 및 $g(x)$ = x 이면 $f(x)/g(x)$ →1 as x → 0. 그러나 $f(x)$ = 1-cos x , $f(x)/g(x)$ →0. 0/0을 두 개의 매우 작은 수의 몫의 제한 상황으로 생각할 때, 0 / 0은 무엇이든 나타낼 수 있습니다. 따라서 IEEE 표준에서 0/0은 NaN이됩니다. 그러나 c > 0, $f(x)$ c , $g(x)$ 0이면 모든 분석 함수 f 및 g에 대해 $f(x)/g(x)$ ± 입니다. 만약→ →→ ∞$g(x)$ < 0은 small x 이고 $f(x)/g(x)$ →-∞이고 그렇지 않으면 한계가 +∞입니다. 는 IEEE 표준을 정의 그래서 C / 0 = ±∞ ,만큼 C ≠ 의 0으로 기호∞의 표시에 따라 C 때문에, 일반적인 방법으로 0이 -10/0 = -∞, -10 / -0 = +∞ . 당신은 점점 구별 할 수∞있기 때문에 오버 플로우로와지고∞섹션에서 자세히 설명 될 것이다 (상태 플래그를 확인하여 0으로 인해 분열의 깃발). 첫 번째 경우 오버플로 플래그가 설정되고 두 번째 경우 0으로 나누기 플래그가 설정됩니다.

피연산자로 무한대를 갖는 연산의 결과를 결정하는 규칙은 간단합니다. 무한대를 유한 숫자 x로 바꾸고 한계를 x 로 취하십시오 → ∞. 따라서 3 / ∞ = 0,

$$\lim_{x\to\infty} 3/x = 0$$

.

마찬가지로 4- ∞=- ∞및 $\sqrt{\infty}$ = ∞입니다. 한계가 존재하지 않으면 결과는 NaN이므로 ∞/ ∞는 NaN이됩니다 (표 D-3 에는 추가 예제가 있습니다). 이것은 0/0이 NaN이어야한다는 결론에 사용 된 추론과 일치합니다.

하위 표현식이 NaN으로 평가되면 전체 표현식의 값도 NaN입니다. 그러나 ±∞ 의 경우 식의 값은 1 / ∞= 0과 같은 규칙으로 인해 일반 부동 소수점 숫자가 될 수 있습니다. 다음은 무한 산술 규칙을 사용하는 실제 예제입니다. 함수 x / (x^2 + 1) 계산을 고려하십시오 . $x\sqrt{\beta}\beta^{e_{max}/2}$ 가보다 클 때 오버플로 될 뿐만 아니라 무한대 산술은 1 / x 근처의 숫자가 아닌 0을 산출하기 때문에 잘못된 답을 제공 하기 때문에 이것은 잘못된 공식 입니다. 그러나 x / (x^2 + 1)은 1 / (x +x^{-1}). 이 향상된 표현식은 너무 일찍 오버플로되지 않으며 무한대 산술로 인해 x = 0 일때 올바른 값을 갖게됩니다: 1 / (0 + 0^{-1}) = 1 / (0 +∞) = 1 /∞= 0. 무한대 산술이 없으면 식 1 / (x + x^{-1})에는 x = 0에대한 테스트가 필요합니

다.이는 추가 명령을 추가 할뿐만 아니라 파이프 라인을 방해 할 수도 있습니다. 이 예는 일반적인 사실을 보여줍니다. 즉, 무한대 산술은 종종 특별한 케이스 검사의 필요성을 피합니다. 그러나 수식은 무한대 ($x / (x^2 + 1)$처럼)에서 가짜 동작이 없는지 확인하기 위해 신중하게 검사해야합니다.

부호있는 0

0은 지수 e_{min} -1과 0으로 표시됩니다. 부호 비트는 두 개의 다른 값을 가질 수 있으므로 두 개의 0, +0 및 -0이 있습니다. +0과 -0을 비교할 때 구별이 이루어지면 `if (x = 0)`의 부호에 따라 같은 간단한 테스트 는 예측할 수없는 동작을 x합니다. 따라서 IEEE 표준은 -0 <+0이 아닌 +0 = -0이되도록 비교를 정의합니다. 항상 0의 부호를 무시할 수 있지만 IEEE 표준은 무시하지 않습니다. 곱셈 또는 나눗셈에 부호있는 0이 포함 된 경우 일반적인 부호 규칙이 답의 부호를 계산하는 데 적용됩니다. 따라서 3 · (+0) = +0, + 0 / -3 = -0. 0에 부호가 없으면 관계식 1 / (1 / x) = xx = ± 일∞ 때 유지하지 못할 것 입니다. 그 이유는 1 /- ∞및 1 / + ∞모두 0이되고 1/0 ∞은 부호 정보가 손실 된 +가되기 때문입니다. ID 1 / (1 / x) = x 를 복원하는 한 가지 방법 은 한 종류의 무한대 만 갖는 것입니다. 그러나 이는 넘친 양의 부호를 잃어 버리는 비참한 결과를 초래합니다.

부호있는 0 사용의 또 다른 예는 로그와 같이 0에서 불연속성이있는 함수 및 언더 플로와 관련이 있습니다. IEEE 산술에서 log 0 =-을 정의 ∞하고 x <0 일 때 log x 를 NaN 으로 정의하는 것은 당연 합니다. x 가 0으로 언더 플로 된 작은 음수를 나타낸다고 가정합니다 . 부호있는 0 덕분에 x 는 음수이므로 log는 NaN을 반환 할 수 있습니다. 그러나 부호있는 0이없는 경우 로그 함수는 언더 플로 된 음수를 0과 구별 할 수 없으므로-를 반환해야합니다 . 불연속성이 0 인 함수의 또 다른 예는 숫자의 부호를 반환하는 signum 함수입니다.∞

아마도 부호있는 0의 가장 흥미로운 사용은 복잡한 산술에서 발생합니다. 간단한 예를 들어, 방정식을 고려하십시오 $\sqrt{1/z} = 1/(\sqrt{z})$. 이것은 $z \geq 0$ 일 때 확실히 사실입니다. z = -1 이면 명백한 계산은 $\sqrt{1/(-1)} = \sqrt{-1} = i$ and를 제공합니다 $1/(\sqrt{-1}) = 1/i = -i$. 따라서 $\sqrt{1/z} \neq 1/(\sqrt{z})$! 문제는 제곱근이 다중 값이고 전체 복잡한 평면에서 연속되도록 값을 선택할 수있는 방법이 없다는 사실에서 추적 할 수 있습니다. 그러나 모든 음의 실수로 구성된 분기 절단 이 고려에서 제외 되면 제곱근은 연속적 입니다. 이것은 -x + i 0, 여기서 x> 0. 부호있는 0은이 문제를 해결하는 완벽한 방법을 제공합니다. 형태의 숫자는 X + I (+0)은 하나 개의 기호가 $(i\sqrt{x})$ 형태의 숫자 X + I (-0) 분기 컷의 반대편에 다른 기호를 가지고가 $(-i\sqrt{x})$. 사실, 컴퓨팅 $\sqrt{}$ 을 위한 자연적인 공식 은 이러한 결과를 제공합니다.

로 돌아 가기 $\sqrt{1/z} = 1/(\sqrt{z})$. 만약 Z = 1 = -1 + 난 후, 0

$$1 / z = 1 / (-1 + i\,0) = [(-1 - i\,0)] / [(-1 + i\,0)\,(-1 - i\,0)] = (-1 - i\,0) / ((-1)^2 - 0^2) = -1 + I\,(-0)$$

그래서 $\sqrt{1/z} = \sqrt{-1 + i(-0)} = -i$, 동안 $1/(\sqrt{z}) = 1/i = -i$. 따라서 IEEE 산술은 모든 z에 대해이 ID를 유지합니다 . Kahan [1987]은 좀 더 정교한 예를 제공합니다. +0과 -0을 구별하는 것이 장점이 있지만 때때로 혼동 될 수 있습니다. 예를 들어 부호있는 0은 $x = y \Leftrightarrow 1 / x = 1 / y$ 관계를 파괴합니다. x = +0이고 y = -0. 일 때 거짓 입니다. 그러나 IEEE위원회는 제로 부호를 사용하는 이점이 단점보다 더 크다고 결정했습니다.

비정규 화 된 숫자

β= 10, p = 3, e_{min} = -98 인 정규화 된 부동 소수점 숫자를 고려하십시오 . 숫자 x = 6.87×10^{-97} 및 y = 6.81×10^{-97} 은 가장 작은 부동 소수점 숫자 1.00×10^{-98} 보다 10 배 이상 큰 완벽하게 일반적인 부동 소수점 숫자로 보입니다 . 그러나 그들은 이상한

속성이 있습니다 : X의 $\ominus$ $Y = 0$ 비록 X의 Y ! 그 이유는 $x - y = .06 \times 10^{-97} \neq = 6.0 \times 10^{-99}$ 는 정규화 된 숫자로 표현하기에는 너무 작으므로 0으로 플러시해야합니다. 재산을 보존하는 것이 얼마나 중요한가

(10) $x = y$ $x-y = 0$? $\Leftrightarrow$

코드 조각을 작성하는 것을 상상하기가 매우 쉽습니다 . 그리고 훨씬 나중에 0으로 스퓨리어스 나누기로 인해 프로그램이 실패하게됩니다. 이와 같은 버그를 추적하는 것은 실망스럽고 시간이 많이 걸립니다. 더 철학적 인 수준에서 컴퓨터 과학 교과서는 큰 프로그램이 올바른지 증명하는 것이 현재 비실용적 임에도 불구하고이를 증명하는 아이디어로 프로그램을 설계하면 종종 더 나은 코드가된다고 지적합니다. 예를 들어, 불변성을 도입하는 것은 증명의 일부로 사용되지 않더라도 매우 유용합니다. 부동 소수점 코드는 다른 코드와 마찬가지로 신뢰할 수있는 사실을 입증하는 데 도움이됩니다. 화학식 분석 할 때 예를 들어, (6) , 이는 알고 매우 도움이되었다 $X / 2 < Y < 2X$ `if (x ≠ y) then z = 1/(x-y)` $\Rightarrow$ $\ominus$ $y = x - y$. 마찬가지로 (10) 이 참임을 알면신뢰할 수있는 부동 소수점 코드를 더 쉽게 작성할 수 있습니다. 대부분의 숫자에 대해서만 사실이라면 아무것도 증명하는 데 사용할 수 없습니다.

IEEE 표준은 (10) 및 기타 유용한 관계 를 보장 하는 비정규 화 된 18 개 숫자를 사용 합니다. 그들은 표준에서 가장 논란이 많은 부분이며 아마도 754 승인을 받기까지 오랜 지연을 설명했습니다. IEEE와 호환된다고 주장하는 대부분의 고성능 하드웨어는 비정규 화 된 숫자를 직접 지원하지 않고 비정규를 소비하거나 생성 할 때 트랩하고 IEEE 표준을 시뮬레이션하기 위해 소프트웨어에 맡깁니다. 19 비정규 화 된 숫자의 개념은 Goldberg [1967]로 거슬러 올라가며 매우 간단합니다. 지수가 e_{min} 이면 significand를 정규화 할 필요가 없으므로 $\beta = 10$, $p = 3$ 및 e_{min} 일 때 $= -98$, 1.00×10^{-98} 은 0.98×10^{-98} 도 부동 소수점 숫자 이기 때문에 더 이상 가장 작은 부동 소수점 숫자가 아닙니다 .

작은 걸리고있다 β의 지수와 숫자 때문에 숨겨진 비트가 사용되고 = 2 전자 분 항상 유효 수를 더 이상 없기 때문에 또는 암시 선두 비트의 1.0와 동일한 뜻. 솔루션은 0을 나타내는 데 사용 된 것과 유사하며 표 D-2에 요약되어 있습니다. 지수 e_{min} 은 비정규를 나타내는 데 사용됩니다. 보다 공식적으로, significand 필드의 비트가 $b_1, b_2, \ldots, b_{p-1}$ 이고 지수 값이 e 이면 $e > e_{min}-1$, 표시되는 숫자는 1입니다. $b_1 b_2 \ldots b_{p-1} \times 2^e$ 반면 $e = e_{min} -1$ 일 때 표시되는 숫자는 0입니다. $b_1 b_2 \ldots b_{p-1} \times 2^{e+1}$. 이 비정규의 지수 때문에 지수의 +1 필요 전자 분 아닌 $E_{분}$ 1 -.

이 섹션의 시작 부분에 제시된 $\beta = 10$, $p = 3$, $e_{min} = -98$, $x = 6.87 \times 10^{-97}$ 및 $y = 6.81 \times 10^{-97}$ 의 예를 생각해보십시오 . 비정규 화를 사용하면 $x - y$ 는 0으로 플러시되지 않고 대신 비정규 화 된 숫자 $.6 \times 10^{-98}$로 표시 됩니다. 이 동작을 점진적 언더 플로 라고 합니다. 점진적 언더 플로를 사용할 때 (10)이 항상 유지 되는지 확인하는 것은 쉽습니다 .

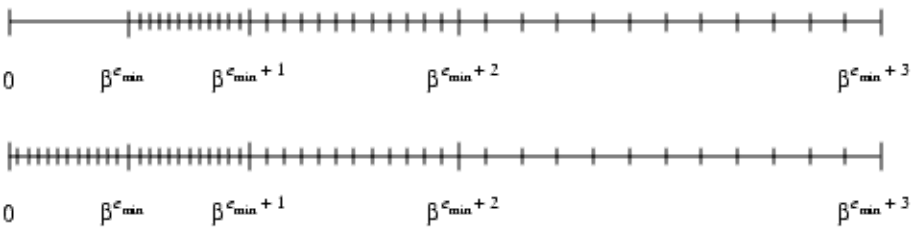


그림 D-2 점진적 언더 플로우와 비교하여 0으로 플러시

그림 D-2 는 비정규 화 된 숫자를 보여줍니다. 그림의 맨 위 숫자 라인은 정규화 된 부동 소수점 숫자를 보여줍니다. 0과 가장 작은 정규화 된 숫자 사이의 간격을 확인하십시오 $1.0 \times \beta^{e_{min}}$. 부동 소수점 계산의 결과가이 걸프에 속하면 0으로 플러시됩니다. 맨 아래 숫자 라인은 비정규가 부동 소수점 숫자 세트에 추가 될 때 발생하는 일을 보여줍니다. "걸프"가 채워지고 계산 결과가보다 작 으면 $1.0 \times \beta^{e_{min}}$ 가장 가까운 비정규로 표시됩니다. 비정규 화 된 숫자가 수직선에 추가되면 인접한 부동 소수점 숫자 사이의 간격이 규칙적으로 달라집 니다. 인접한 간격은 길이가 같거나 인수만큼 다릅니다 β. 비정규가 없으면 간격이에서 $\beta^{-p+1}\beta^{e_{min}}$ 로 갑자기 변경됩니다 . $\beta^{e_{min}}$, 이는의 요인에 β^{p-1} 의해 질서 정연한 변경이 아니라의 요인입니다 β. 이 때문에 언더 플로 임계 값에 가까운 정규화 된 숫자에 대해 큰 상대 오차를 가질 수있는 많은 알고리즘은 점진적 언더 플로가 사용될 때이 범위에서 잘 작동합니다.

점진적인 언더 플로가 없으면 $x = 6.87 \times 10^{-97}$ 및 $y = 6.81 \times 10^{-97}$에 대해 위에서 본 것처럼 간단한 표현식 $x - y$ 는 정규화 된 입력에 대해 매우 큰 상대 오차를 가질 수 있습니다 . 다음 예제가 [Demmel 1984]에 표시된 것처럼 취소 없이도 큰 상대 오류가 발생할 수 있습니다. 두 개의 복소수 $a + ib$ 및 $c + id$를 나누는 것을 고려하십시오 . 명백한 공식

$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}$. 나

분모 $c + id$ 의 한 구성 요소 가보다 크면 $\sqrt{\beta}\beta^{e_{max}/2}$ 최종 결과가 범위 내에 있어도 수식이 오버플로되는 문제가 있습니다. 몫을 계산하는 더 좋은 방법은 Smith의 공식을 사용하는 것입니다.

$$\frac{a+ib}{c+id} = \begin{cases} \frac{a+b(d/c)}{c+d(d/c)} + i\frac{b-a(d/c)}{c+d(d/c)} & \text{if}(|d| < |c|) \\ \frac{b+a(c/d)}{d+c(c/d)} + i\frac{-a+b(c/d)}{d+c(c/d)} & \text{if}(|d| \geq |c|) \end{cases}$$

(11)

Smith의 공식을 $(2 \cdot 10^{-98} + i\, 10^{-98}) / (4 \cdot 10^{-98} + i\,(2 \cdot 10^{-98}))$에 적용하면 점진적인 언더 플로가있는 0.5의 정답이됩니다. 0으로 플러시 한 상태에서 0.4를 산출하고 오류는 100ulps입니다. 비정규 화 된 숫자가 1.0 x까지 인수에 대한 오류 경계를 보장하는 것이 일반적입니다 $\beta^{e_{min}}$.

예외, 플래그 및 트랩 처리기

IEEE 산술에서 0으로 나누기 또는 오버플로와 같은 예외적 인 조건이 발생하면 기본값은 결과를 전달하고 계속하는 것입니다. 일반적인 기본 결과는 0/0 및 $\sqrt{-1}$, ∞1/0 및 오버플로에 대한 NaN입니다 . 이전 섹션에서는 이러한 기본값을 사용하여 예외에서 진행하는 것이 합당한 일인 예를 제공했습니다. 예외가 발생하면 상태 플래그도 설정됩니다. 사용자에게 상태 플래그를 읽고 쓰는 방법을 제공하려면 IEEE 표준을 구현해야합니다. 플래그는 일단 설정되면 명시 적으로 지워질 때까지 설정된 상태로 유지된다는 점에서 "고정"됩니다.

플래그를 테스트하는 것은 1/0을 구별하는 유일한 방법이며, 이는 오버플로와 진정한 무한대입니다.

때때로 예외 상황에서 계속 실행하는 것은 적절하지 않습니다. 섹션 Infinity 는 $x / (x^2 + 1)$의 예를 제공했습니다. 경우 X는 > $\sqrt{\beta}\beta^{e_{max}/2}$, 분모가 무한대 완전히 잘못된 0의 최종 해답 결과. 이 공식의 경우 문제는 $1 / (x + x^{-1}$ 로 다시 작성하여 해결할 수 있습니다.), 다시 쓰기가 항상 문제를 해결하는 것은 아닙니다. IEEE 표준에서는 구현시 트랩 처리기가 설치되도록 할 것을 강력히 권장합니다. 그런 다음 예외가 발생하면 플래그를 설정하는 대신 트랩 처리기가 호출됩니다. 트랩 처리기에서 반환 된 값은 작업의 결과로 사용됩니다. 상태 플래그를 지우거나 설정하는 것은 트랩 처리기의 책임입니다. 그렇지 않으면 플래그의 값이 정의되지 않을 수 있습니다.

IEEE 표준은 예외를 오버플로, 언더 플로, 0으로 나누기, 잘못된 연산 및 부정확 함의 5 가지 클래스로 나눕니다. 각 예외 클래스에 대해 별도의 상태 플래그가 있습니다. 처음 세 가지 예외의 의미는 자명합니다. 잘못된 작업은 표 D-3에 나열된 상황 과 NaN을 포함하는 모든 비교를 다룹니다 . 잘못된 예외를 발생시키는 작업의 기본 결과는 NaN을 반환하는 것이지만 그 반대는 참이 아닙니다. 연산의 피연산자 중 하나가 NaN이면 결과는 NaN이지만 연산이 표 D-3 의 조건 중 하나를 충족하지 않는 한 잘못된 예외가 발생하지 않습니다 .[20]

표 D-4 IEEE 754 *의 예외

예외	트랩이 비활성화되었을 때의 결과	트랩 처리기에 대한 인수
과다	$\pm\infty$ 또는 $\pm$ x 최대	라운드 ($X 2^{-\alpha}$)
언더 플로	0 $\pm 2^{e_{min}}$ 또는 비정규	원형 ($x 2^{\alpha}$)
0으로 나누다	$\pm\infty$	피연산자
유효하지 않음	NaN	피연산자
정확하지 않은	원형 (x)	원형 (x)

* x 는 연산의 정확한 결과이며, α단 정밀도의 경우 = 192, 배정 밀도의 경우 1536, x_{max} = 1.11 ... 11 × $2^{e_{max}}$ 입니다.

부동 소수점 연산의 결과가 정확하지 않으면 정확하지 않은 예외가 발생합니다. 에서 β= 10, P = 3에있어서, 3.5 ⊗4.2 = 14.7 정확한이지만, 3.5 ⊗ (3.5 이후 4.3 = 15.0 정확한 아니다 * 4.3 = 15.05), 및 부정확 한 예외를 제기한다. Binary to Decimal Conversion 은 정확하지 않은 예외를 사용하는 알고리즘에 대해 설명합니다. 다섯 가지 예외의 동작에 대한 요약은 표 D-4에 나와 있습니다.

부정확 한 예외가 너무 자주 발생한다는 사실과 관련된 구현 문제가 있습니다. 부동 소수점 하드웨어에 자체 플래그가 없지만 대신 부동 소수점 예외를 알리기 위해 운영 체제를 인터럽트하는 경우 부정확 한 예외 비용이 엄청날 수 있습니다. 이 비용은 소프트웨어가 상태 플래그를 유지함으로써 피할 수 있습니다. 처음으로 예외가 발생하면 적절한 클래스에 대한 소프트웨어 플래그를 설정하고 부동 소수점 하드웨어에 해당 예외 클래스를 마스크하도록 지시합니다. 그러면 운영 체제를 중단하지 않고 모든 추가 예외가 실행됩니다. 사용자가 해당 상태 플래그를 재설정하면 하드웨어 마스크가 다시 활성화됩니다.

트랩 핸들러

트랩 처리기의 한 가지 분명한 용도는 이전 버전과의 호환성입니다. 예외 발생시 중단 될 것으로 예상되는 이전 코드는 프로세스를 중단하는 트랩 처리기를 설치할 수 있습니다. 이는 .NET과 같은 루프가있는 코드에 특히 유용합니다 `do S until (x >= 100)`. NaN을 <,, ≤ >, 또는 = (그러나 아님) 가있는 숫자와 비교하면 항상 false가 반환되므로이 코드는 NaN이되면 무한 루프 가됩니다. $\geq \neq_x$

잠재적으로 오버플 로 될 수있는 제품을 계산할 때 나타나는 트랩 처리기에 대한 더 흥미로운 용도가 있습니다 . 한 가지 해결책은 로그를 사용하고 대신 exp 를 계산하는 것입니다. 이 접근 방식의 문제점은 정확도가 떨어지고 오버플로가 없더라도 간단한 표현식보다 비용이 많이 든다는 것 입니다. 이 두 가지 문제를 모두 방지하는 오버 / 언더 플로우 카운팅 이라는 트랩 핸들러를 사용하는 또 다른 솔루션 이 있습니다 [Sterbenz 1974]. $\Pi_{i=1}^{n} x_i$ $(\Sigma \log x_i)\ \Pi x_i$

아이디어는 다음과 같습니다. 0으로 초기화 된 글로벌 카운터가 있습니다. $p_k = \Pi_{i=1}^{k} x_i$ 일부 *k*에 대해 곱이 오버플로 될 때마다 트랩 처리기는 카운터를 1 씩 증가시키고 지수를 감싼 오버플로 수량을 반환합니다. IEEE 754 단일 정밀도 전자 $_{\text{최대}}$ = 127 그렇다면, *P*는 $_{K}$ $1.45 \times 2^{(130)}$,이 오버 플로우 및 변경 범위로 지수를 다시 감싸 호출되는 트랩 핸들러 발생할 *P*는 $_{K\text{가}}$ 1.45×2^{-62} (아래 참조). 마찬가지로 p_k 언더 플로가 발생하면 카운터는 감소하고 음의 지수는 양의 지수로 래핑됩니다. 모든 곱셈이 완료되었을 때 카운터가 0이면 최종 결과는 p_n 입니다. 카운터가 양수이면 제품이 넘치고 카운터가 음수이면 제품이 넘친 것입니다. 부분 제품이 범위를 벗어나지 않으면 트랩 처리기가 호출되지 않고 계산에 추가 비용이 발생하지 않습니다. 오버플로 / 언더 플로가 있더라도 각 p_k 는 p_{k-1} 에서 계산 되었기 때문에 대수로 계산했을 때보 다 계산이 더 정확합니다.완전 정밀도 곱하기를 사용합니다. Barnett [1987]은 오버 / 언더 플로우 계산의 전체 정확도가 해당 공식의 이전 표에서 오류를 발견 한 공식에 대해 설명합니다.

IEEE 754는 오버플로 또는 언더 플로 트랩 처리기가 호출 될 때 래핑 된 결과를 인수로 전달하도록 지정합니다. 오버플로에 대한 둘러싸 기의 정의는 결과가 무한 정밀도로 계산 된 다음 2로 나눈 α다음 관련 정밀도로 반올림되는 것입니다. 언더 플로의 경우 결과에 2를 곱합니다 α. 지수 α는 단 정밀도의 경우 192이고 배정 밀도의 경우 1536입니다. 1.45×2^{130} 이유는 1.45×2^{-62} 으로 변화시켰다 위의 예.

반올림 모드

IEEE 표준에서는 각 연산이 정확하게 계산 된 다음 반올림되기 때문에 연산 결과가 정확하지 않을 때마다 반올림이 발생합니다. 기본적으로 반올림은 가장 가까운쪽으로 반올림하는 것을 의미합니다. 표준은 세 가지 다른 반올림 모드, 즉 0쪽으로 반올림, + ∞방향으로 반올림 및-쪽으로 반올림하도록 요구합니다 ∞. 정수로 변환 연산과 함께 사용하는 경우-쪽으로 반올림 ∞하면 변환이 플로어 함수가되고 +쪽으로 반올림 ∞하면 천장이됩니다. 반올림 모드는 오버플로에 영향을줍니다. 왜냐하면 0쪽으로 반올림하거나-쪽으로 반올림 ∞하는 경우 양수 크기의 오버플로는 기본 결과가 +가 아닌 표현 가능한 가장 큰 숫자가되게합니다.∞. 마찬가지로, 음수 크기의 오버플로는 + ∞쪽으로 반올림하거나 0쪽으로 반올림할 때 가장 큰 음수를 생성합니다 .

반올림 모드의 한 응용 프로그램은 간격 산술에서 발생합니다 (다른 방법은 Binary to Decimal Conversion 에서 언급 됨). 구간 산술을 사용할 때 두 숫자 *x* 와 *y* 의 합은 구간

입니다 $[z,Z]$. 여기서 x y^{z} 는 -쪽으로 반올림 되고 x y 는 +쪽으로 반올림 됩니다. 추가의 정확한 결과는 간격 내에 포함됩니다 . 반올림 모드가 없으면, 간격 산술 연산에 의해 일반적으로 구현 하고 , 여기서 계산기 엡실론이다. [21] $\oplus\ \infty^{z} \oplus \infty^{[z,Z]} z = (x \oplus y)(1-\varepsilon)$ $z = (x \oplus y)(1+\varepsilon)$ ε 이로 인해 간격 크기가 과대 평가됩니다. 간격 산술 연산의 결과는 간격이므로 일반적으로 연산에 대한 입력도 간격이됩니다. 두 경우 간격 $[\underline{x},\bar{x}]$ 및이 $[\underline{y},\bar{y}]$ 첨가되고, 그 결과는 $[z,Z]$, 여기서 z 이다 $x \oplus y$ 향하여 원형의 반올림 모드 세트 - ∞ 및 z 인 $z \oplus y$ +쪽으로 라운드 라운딩 모드 세트 ∞.

간격 산술을 사용하여 부동 소수점 계산을 수행 할 때 최종 답은 계산의 정확한 결과를 포함하는 간격입니다. 정답이 해당 간격의 어느 위치 에나있을 수 있기 때문에 간격이 큰 경우 (자주 그렇듯이)별로 도움이되지 않습니다. 간격 산술은 다중 정밀도 부동 소수점 패키지와 함께 사용할 때 더 의미가 있습니다. 계산은 먼저 정밀도 *p*로 수행됩니다 . 구간 산술에서 최종 답이 부정확 할 수 있음을 시사하는 경우 최종 구간이 합리적인 크기가 될 때까지 더 높은 정밀도로 계산이 다시 수행됩니다.

플래그

IEEE 표준에는 여러 플래그와 모드가 있습니다. 위에서 설명한 것처럼 언더 플로, 오버플로, 0으로 나누기, 유효하지 않은 연산 및 정확하지 않은 5 가지 예외 각각에 대해 하나의 상태 플래그가 있습니다. 반올림 모드에는 가장 가까운쪽으로 반올림, +쪽으로 반올림, ∞ 0 쪽으로 반올림,-쪽으로 반올림의 네 가지 모드가 있습니다 ∞. 다섯 가지 예외 각각에 대해 활성화 모드 비트가있는 것이 좋습니다. 이 섹션에서는 이러한 모드와 플래그를 어떻게 유용하게 사용할 수 있는지에 대한 몇 가지 간단한 예를 제공합니다. 더 복잡한 예제는 2 진에서 10 진으로 변환 섹션에서 설명합니다 .

x^{n} 을 계산하는 서브 루틴을 작성하는 것을 고려하십시오 . 여기서 n 은 정수입니다. 때 $N > 0$, 같은 간단한 일상

```
PositivePower (x, n) {
 while (n is even) {
     x = x * x
     n = n / 2
 }
 u = x
 while (true) {
     n = n / 2
     (n == 0)이면 u를 반환
     x = x * x
     만약 (n이 홀수) u = u * x
 }
```

$n < 0$ 인 경우 x^{n} 을 계산하는 더 정확한 방법 은 호출하는 PositivePower(1/x, -n)것이 아니라 를 호출 1/PositivePower(x, -n)하는 것입니다. 첫 번째 표현식 은 나눗셈 (즉, 1 / *x*) 에서 반올림 오류가있는 *n* 개의 수량을 곱하기 때문 입니다. 두 번째 표현식에서 이들은 정확하고 (즉, *x*) 최종 나눗셈은 단 하나의 추가 반올림 오류 만 발생합니다. 불행히도 이것은

이 전략에서 약간의 문제입니다. 언더 플로가 발생 하면 언더 플로 트랩 처리기가 호출되거나 언더 플로 상태 플래그가 설정됩니다. x^{-n}이 언더 플로우되면 x^{n} 이 오버 플로우되거나 범위 내에 있기 때문에 이것은 올바르지 않습니다 . [22] `PositivePower(x, -n)`그러나 IEEE 표준은 사용자에게 모든 플래그에 대한 액세스를 제공하기 때문에 서브 루틴은이를 쉽게 수정할 수 있습니다. 단순히 오버플로 및 언더 플로 트랩 활성화 비트를 끄고 오버플로 및 언더 플로 상태 비트를 저장합니다. 그런 다음 `1/PositivePower(x, -n)`. 오버플로 또는 언더 플로 상태 비트가 설정되지 않은 경우 트랩 활성화 비트와 함께 복원됩니다. 상태 비트 중 하나가 설정되면 플래그를 복원하고를 사용하여 계산을 다시 실행 `PositivePower(1/x, -n)`하여 올바른 예외가 발생합니다.

플래그 사용의 또 다른 예는 공식을 통해 arccos를 계산할 때 발생합니다.

arccos x = 2 arctan $\sqrt{\frac{1-x}{1+x}}$.

arctan (∞)이 π/ 2로 평가 되면 arccos (-1)은 무한대 산술로 인해 2 · arctan (∞) = 로 올바르게 평가됩니다 π. 그러나 (1- x) / (1 + x) 계산은 arccos (-1)이 예외가 아니더라도 0으로 나누기 예외 플래그가 설정되도록 하기 때문에 작은 문제 가 있습니다. 이 문제에 대한 해결책은 간단합니다. arccos를 계산하기 전에 0으로 나누기 플래그의 값을 저장 한 다음 계산 후 이전 값을 복원하기 만하면됩니다.

시스템 측면

컴퓨터 시스템의 거의 모든 측면을 설계하려면 부동 소수점에 대한 지식이 필요합니다. 컴퓨터 아키텍처에는 일반적으로 부동 소수점 명령어가 있고 컴파일러는 이러한 부동 소수점 명령어를 생성해야하며 운영 체제는 이러한 부동 소수점 명령어에 대해 예외 조건이 발생할 때 수행 할 작업을 결정해야합니다. 컴퓨터 시스템 설계자는 일반적으로 컴퓨터 설계자가 아닌 소프트웨어 사용자와 작성자를 대상으로하는 수치 분석 텍스트에서 지침을 거의 얻지 못합니다. 그럴듯한 설계 결정이 예상치 못한 동작으로 이어질 수있는 방법에 대한 예로서 다음 BASIC 프로그램을 고려하십시오.

```
q = 3.0 / 7.0
q = 3.0 / 7.0이면 "Equal"을 인쇄합니다.
    그렇지 않으면 "같지 않음"을 인쇄합니다.
```

IBM PC에서 Borland의 Turbo Basic을 사용하여 컴파일하고 실행하면 프로그램은 `Not Equal`! 이 예제는 다음 섹션에서 분석됩니다.

덧붙여서, 어떤 사람들은 그러한 변칙에 대한 해결책이 부동 소수점 숫자를 동일성을 위해 비교하지 않고 대신 오류 경계 E 내에있는 경우 동등하다고 간주하는 것이라고 생각합니다 . 대답만큼 많은 질문을 제기하기 때문에 이것은 모든 치료법이 아닙니다. E 의 가치는 무엇입니까 ? 경우 X <0와 y는> 0 내에있다 E , 그들은 정말 서로 다른 징후에도 불구하고, 동일한 것으로 간주해야 하는가? 또한이 규칙에 의해 정의 된 관계 $a \sim b \Leftrightarrow |a-b| < E$ 는 $a \sim b$ 및 $b \sim c$ 이므로 등가 관계가 아닙니다.$a \sim c$를 의미하지는 않습니다 .

명령어 세트

알고리즘이 정확한 결과를 생성하기 위해 더 높은 정밀도의 짧은 버스트를 요구하는 것은 매우 일반적입니다. 한 가지 예가 2 차 공식 ($\frac{-b \pm \sqrt{b^2-4ac}}{}$) / 2 a 에서 발생합니다 . 섹션

에서 논의 된 바와 같이 정리 (4)의 증명 , b를 $^{2} \approx 4AC$, 라운딩 에러는 이차 식으로 계산 뿌리 절반 자리까지 오염시킬 수있다. subcalculation의 행함으로써 B^{2} - 4 AC 배정밀도를 루트의 절반 배정도 비트는 모든 단 정밀도 비트가 보존되도록하는 수단 손실된다.

의 계산 B^{2} - 4 AC 배정도의 수량을 각각 , B 및 C는 두 개의 단일 정밀 번호 취해 배정도 결과를 생성하는 승산 명령어가 있는지 단 정밀도에 용이하다. 두 개의 p- 자리 숫자 의 정확한 반올림 된 곱을 생성하려면 승수가 전체 $2p$ 를 생성해야합니다.진행함에 따라 조금씩 버릴 수 있지만 제품의 일부. 따라서 단 정밀도 피연산자에서 배정 밀도 제품을 계산하는 하드웨어는 일반적으로 단 정밀도 승수보다 약간 비싸고 배정 밀도 승수보다 훨씬 저렴합니다. 그럼에도 불구하고 최신 명령어 세트는 피연산자와 동일한 정밀도의 결과를 생성하는 명령어 만 제공하는 경향이 있습니다. [23]

두 개의 단 정밀도 피연산자를 결합하여 배정 밀도 제품을 생성하는 명령어가 2 차 공식에만 유용했다면 명령어 세트에 추가 할 가치가 없습니다. 그러나이 명령어는 다른 용도로 많이 사용됩니다. 선형 방정식 시스템을 푸는 문제를 고려하십시오.

$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$
$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \cdots a_{n1}x_1 + {}_{N2}X_2 + \cdots + {}_{NN}X_N = B$의 $_N$

행렬 형식으로 $Ax = b$ 로 쓸 수 있습니다 . 여기서

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2(1)n} \\ & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

해 $x^{(1)}$ 가 가우스 제거와 같은 방법으로 계산 된다고 가정합니다 . 반복적 개선 이라는 결과의 정확성을 개선하는 간단한 방법이 있습니다 . 첫 번째 컴퓨팅

(12) ξ = 액스 $^{(1)}$ - (B)를

그런 다음 시스템을 해결하십시오.

(13) $Ay = \xi$

경우 유의 $X^{(1)}$가 정확한 해결책이 후 ξ, 영 벡터이고,은이고 , Y . 일반적으로,의 계산 ξ 및 Y는 , 반올림 에러 발생 있도록 불안 액스 $^{(1)}$ - (B) = ($X^{(1)}$ - (X)) 여기서 X 제 (미지) 참 용액이다. 그러면 Y $(X)^{(1)}$ - (X) 의 용액에 대한 개선 된 추정치가되도록 $\approx \xi \approx \approx$

(14) $X^{(2)} = X^{(1)} - Y$

$x^{(1)}$ 을 $x^{(2)}$로 , $x^{(2)}$ 를 $x^{(3)}$로 바꾸어 세 단계 (12) , (13) , (14) 를 반복 할 수 있습니다 . $x^{(i+1)}$ 가 $x^{(i)}$ 보다 정확하다는 이 주장 은 비공식적 일뿐입니다. 자세한 내용은 [Golub and Van Loan 1989]를 참조하십시오.

반복 개선을 수행 할 때는 ξ요소가 근처에있는 부정확 한 부동 소수점 숫자의 차이 인 벡터이므로 치명적인 취소가 발생할 수 있습니다. 따라서 반복하지 않는 개선에 매우 유용하지 않다 ξ = 액스$^{(1)}$ - *(B)*가 배정 밀도 연산된다. 다시 한 번, 이것은 완전한 배정 밀도 결과가 필요한 두 개의 단 정밀도 숫자 (*A* 및 $x^{(1)}$) 의 곱을 계산하는 경우입니다 .

요약하면 두 개의 부동 소수점 숫자를 곱하고 피연산자의 정밀도가 두 배인 곱을 반환하는 명령어는 부동 소수점 명령어 세트에 유용한 추가 기능을 제공합니다. 이것이 컴파일러에 미치는 영향 중 일부는 다음 섹션에서 설명합니다.

언어 및 컴파일러

컴파일러와 부동 소수점의 상호 작용은 Farnum [1988]에서 논의되었으며,이 섹션의 많은 논의는 해당 백서에서 가져 왔습니다.

모호

이상적으로는 언어 정의는 프로그램에 대한 설명을 증명할 수있을만큼 정확하게 언어의 의미를 정의해야합니다. 이것은 일반적으로 언어의 정수 부분에 해당되지만 언어 정의는 부동 소수점과 관련하여 종종 큰 회색 영역을 갖습니다. 아마도 이것은 많은 언어 디자이너들이 반올림 오류를 수반하기 때문에 부동 소수점에 대해 아무것도 증명할 수 없다고 믿기 때문일 것입니다. 그렇다면 이전 섹션에서이 추론의 오류를 입증했습니다. 이 섹션에서는 처리 방법에 대한 제안을 포함하여 언어 정의의 몇 가지 일반적인 회색 영역에 대해 설명합니다.

놀랍게도, 일부 언어에서는 if `x`가 부동 소수점 변수 (값이 `3.0/10.0`)이면 (예)의 모든 항목 `10.0*x`이 동일한 값을 가져야 한다고 명확하게 지정하지 않습니다 . 예를 들어 Brown의 모델을 기반으로하는 Ada는 부동 소수점 산술이 Brown의 공리 만 충족하면되므로 표현식이 가능한 많은 값 중 하나를 가질 수 있음을 암시하는 것 같습니다. 이러한 퍼지 방식으로 부동 소수점에 대해 생각하는 것은 각 부동 소수점 연산의 결과가 정확하게 정의되는 IEEE 모델과 뚜렷한 대조를 이룹니다. IEEE 모델에서 우리는 그것이 (정리 7)로 `(3.0/10.0)*10.0` 평가 된다는 것을 증명할 수 있습니다 `3`. Brown의 모델에서는 할 수 없습니다.

대부분의 언어 정의에서 또 다른 모호성은 오버플로, 언더 플로 및 기타 예외에서 발생하는 일과 관련이 있습니다. IEEE 표준은 예외의 동작을 정확하게 지정하므로 표준을 모델로 사용하는 언어는이 점에서 모호함을 피할 수 있습니다.

또 다른 회색 영역은 괄호 해석과 관련이 있습니다. 반올림 오류로 인해 대수의 연관 법칙이 부동 소수점 숫자에 대해 반드시 유지되는 것은 아닙니다. 예를 들어, 표현은 `(x+y)+z`보다 완전히 다른 답 갖는 `x+(y+z)`경우 $X = 10^{(30)}$, Y는 -10^{30} 및 $Z = 1$ (이 후자 0 전자의 경우 1). 괄호 보존의 중요성은 아무리 강조해도 지나치지 않습니다. 정리 3, 4, 6에 제시된 알고리즘은 모두 그것에 의존합니다. 예를 들어 정리 6에서 공식 $x_h = mx - (mx - x)$는 $x_h =$로 축소됩니다.*x* 가 괄호가 아닌 경우 전체 알고리즘을 파괴합니다. 괄호를 사용할 필요가없는 언어 정의는 부동 소수점 계산에 쓸모가 없습니다.

하위 표현식 평가는 많은 언어에서 부정확하게 정의됩니다. 그 가정 `ds`배정 밀도이지만, `x` 하고 `y`단정하다. 그렇다면 식에서 `ds + x*y`제품이 단 정밀도 또는 배정 밀도로 수행됩니까?

또 다른 예 : in `x + m/n`where `m`and`n`정수, 나눗셈은 정수 연산입니까 아니면 부동 소수점입니까? 이 문제를 처리하는 방법에는 두 가지가 있으며 둘 다 완전히 만족스럽지 않습니다. 첫 번째는 표현식의 모든 변수가 동일한 유형을 가져야한다는 것입니다. 이것은 가장 간단한 해결책이지만 몇 가지 단점이 있습니다. 우선, 하위 범위 유형을 가진 Pascal과 같은 언어는 하위 범위 변수와 정수 변수를 혼합 할 수 있으므로 단 정밀도 변수와 배정 밀도 변수를 혼합하는 것을 금지하는 것이 다소 이상합니다. 또 다른 문제는 상수에 관한 것입니다. 표현에서`0.1*x`, 대부분의 언어는 0.1을 단 정밀도 상수로 해석합니다. 이제 프로그래머가 모든 부동 소수점 변수의 선언을 단 정밀도에서 배정 밀도로 변경하기로 결정했다고 가정합니다. 0.1이 여전히 단 정밀도 상수로 취급되면 컴파일 시간 오류가 발생합니다. 프로그래머는 모든 부동 소수점 상수를 찾아서 변경해야합니다.

두 번째 접근 방식은 혼합 식을 허용하는 것입니다.이 경우 하위 식 평가에 대한 규칙을 제공해야합니다. 여러 가지 안내 예제가 있습니다. C의 원래 정의에서는 모든 부동 소수점 표현식이 배정 밀도로 계산되어야했습니다 [Kernighan and Ritchie 1978]. 이로 인해이 섹션의 시작 부분에있는 예제와 같은 이상이 발생합니다. 표현식 `3.0/7.0`은 배정 밀도로 계산되지만가 `q`단 정밀도 변수 인 경우 몫은 저장을 위해 단 정밀도로 반올림됩니다. 3/7은 반복 이진 분수이므로 배정 밀도로 계산 된 값은 단 정밀도로 저장된 값과 다릅니다. 따라서 비교 *q* = 3/7은 실패합니다. 이것은 사용 가능한 가장 높은 정밀도로 모든 표현식을 계산하는 것이 좋은 규칙이 아님을 시사합니다.

또 다른 지침은 내부 제품입니다. 내적에 수천 개의 항이있는 경우 합계의 반올림 오류가 상당해질 수 있습니다. 이 반올림 오류를 줄이는 한 가지 방법은 합계를 배정 밀도로 누적하는 것입니다 (이는 옵티 마이저 섹션에서 자세히 설명합니다). 경우 `d`배정 밀도 변수이고, `x[]`그리고 `y[]`그 내적 루프 모양을 단 정밀도 배열입니다 `d = d + x[i]*y[i]`. 곱셈이 단 정밀도로 수행되는 경우 곱이 배정 밀도 변수에 추가되기 직전에 단 정밀도로 잘 리기 때문에 배정 밀도 누적의 장점이 많이 손실됩니다.

앞의 두 예제를 모두 다루는 규칙은 해당 표현식에서 발생하는 모든 변수의 가장 높은 정밀도로 표현식을 계산하는 것입니다. 그런 다음 `q = 3.0/7.0`전적으로 단 정밀도 [24] 로 계산되고 부울 값이 true 인 반면 `d = d + x[i]*y[i]`배정 밀도로 계산되어 배정 밀도 누적의 이점을 최대한 활용합니다. 그러나이 규칙은 너무 단순하여 모든 경우를 명확하게 포함 할 수 없습니다. 경우 `dx`와 `dy`배정도 변수 표현식은 `y = x + single(dx-dy)`두 피연산자가 단 정밀도 때문에 결과로서, 이중 정확도 변수를 포함하고 있지만, 배정 밀도 합을 행하는 것이 무의미하다.

보다 정교한 하위 표현식 평가 규칙은 다음과 같습니다. 먼저 각 연산에 해당 피연산자의 최대 정밀도 인 임시 정밀도를 할당합니다. 이 할당은 잎에서 표현식 트리의 루트까지 수행되어야합니다. 그런 다음 뿌리에서 잎으로 두 번째 패스를 수행하십시오. 이 단계에서 각 작업에 최대 임시 정밀도와 상위가 예상하는 정밀도를 할당합니다. 의 경우 `q = 3.0/7.0`모든 리프가 단 정밀도이므로 모든 작업이 단 정밀도로 수행됩니다. 의 경우 `d = d + x[i]*y[i]`곱하기 연산의 임시 정밀도는 단 정밀도이지만 두 번째 패스에서는 상위 연산이 배정 밀도 피연산자를 예상하기 때문에 배정 밀도로 승격됩니다. 그리고`y = x + single(dx-dy)`, 추가는 단 정밀도로 수행됩니다. Farnum [1988]은이 알고리즘을 구현하기 어렵지 않다는 증거를 제시합니다.

이 규칙의 단점은 하위 식의 평가가 포함 된 식에 따라 달라진다는 것입니다. 이것은 몇 가지 성가신 결과를 초래할 수 있습니다. 예를 들어, 프로그램을 디버깅 중이고 하위 표현식의 값을 알고 싶다고 가정합니다. 프로그램의 하위 식의 값은 포함 된 식에 따라 다르기 때문에 디버거에 하위 식을 입력하고 평가하도록 요청할 수 없습니다. 하위 식에 대한 마지막 설명 : 십진 상수를 이진으로 변환하는 것은 작업이므로, 평가 규칙은 소수 상수 해석에도 영향을줍니다. 이것은 `0.1`바이너리로 정확하게 표현할 수없는 것과 같은 상수에 특히 중요합니다 .

또 다른 잠재적 회색 영역은 언어에 내장 연산 중 하나로 지수가 포함될 때 발생합니다. 기본적인 산술 연산과는 달리 지수의 값이 항상 분명하지는 않습니다 [Kahan and Coonen 1982]. 경우 **누승 연산자 다음, (-3)**3확실히 값을 가지며 -27. 그러나 (-3.0)**3.0문제가 있습니다. 경우] **정수 능력에 대한 연산자를 검사, 그것은 계산 것이다 (-3.0)**3.0-3.0로 ^{3} = -27. 한편, 화학식 X의 ^{Y} = $E^{y\log X}$ 정의하는 데 사용된다 **(기록의 자연 정의를 사용하여 (즉, 결과는 NaN이 될 수 후 로그 함수에 따라 실제 인수 X) = NaN때 X를 <0). CLOG그러나 FORTRAN 함수를 사용하면 ANSI FORTRAN 표준 CLOG(-3.0)이 $i\pi$ + log 3으로 정의 되어 있으므로 대답은 -27이됩니다 [ANSI 1978]. 프로그래밍 언어 Ada는 정수 거듭 제곱에 대한 지수화 만 정의하여이 문제를 방지하는 반면, ANSI FORTRAN은 음수를 실제 거듭 제곱으로 올리는 것을 금지합니다.

실제로 FORTRAN 표준에 따르면

결과가 수학적으로 정의되지 않은 산술 연산은 금지됩니다.

불행히도 IEEE 표준에 의해 ±∞ 가 도입됨에 따라 수학적으로 정의되지 않은 의미 는 더 이상 완전히 명확하지 않습니다. 한 가지 정의는 Infinity 섹션에 표시된 방법을 사용하는 것 입니다. 예를 들어, a^{b} 의 값을 결정하려면 $f(x)$ a 및 $g(x)$ b 속성을 x 0 으로 하는 상수가 아닌 분석 함수 f 및 g 를 고려하십시오. If $f(x)^{g(x)} \to \to \to$항상 동일한 한계에 접근하면 a^{b} 값이어야합니다 . 이 정의는 2^{∞} = ∞매우 합리적으로 보입니다. 1.0^{∞} 의 경우 $f(x) = 1$ 및 $g(x) = 1 / x$ 일 때 한계가 1에 가까워 지지만 $f(x) = 1- x$ 및 $g(x) = 1 / x$ 일 때 한계는 e입니다. $^{-1}$. 따라서 1.0^{∞}은 NaN이어야합니다. 0^{0} , $f(x)^{g(x)} = e^{g(x)\log f(x)}$. 이후 F 및 g는 분석하고 0의 값을 0에 걸릴, $F(x$는 $) = {}_{1}X^{1} + {}_{2}X^{2}$ + ... 및 g을 $(X) = B_{(1\text{ 개})}, X^{(1)} + B_{2}X^{2}$ + ... 따라서 $\lim_{x\,0} g(x) \log f \to (x) = \lim_{x\,0} x \log(x(a_{1} + a_{2}x + \ldots)) = \lim_{x\,0} x \log(a_{1}x) =$ 0. 그래서 $f(x)^{g(x)} e^{0}$ = 모든 f 및 g에 대해 1입니다 . 이는 0^{0} = 1 을 의미합니다 . [25] [26] 이 정의를 사용하면 모든 인수에 대한 지수 함수가 모호하지 않게 정의되며 특히 다음과 같이 정의됩니다. → → → (-3.0)**3.0 -27이됩니다.

IEEE 표준

섹션 IEEE 표준IEEE 표준에는 네 가지 정밀도가 있습니다 (권장 구성은 단일 더하기 단일 확장 또는 단일, 이중 및 이중 확장). Infinity는 또 다른 예를 제공합니다. 나타낼 상수±∞ 서브 루틴에 의해 제공 될 수 있습니다. 그러나 이로 인해 상수 변수의 이니셜 라이저와 같이 상수 표현식이 필요한 곳에서는 사용할 수 없게 될 수 있습니다.

더 미묘한 상황은 계산과 관련된 상태를 조작하는 것입니다. 여기서 상태는 반올림 모드, 트랩 활성화 비트, 트랩 처리기 및 예외 플래그로 구성됩니다. 한 가지 접근 방식은 상태를 읽고 쓰기위한 서브 루틴을 제공하는 것입니다. 또한 새 값을 원자 적으로 설정하고 이전 값을 반환 할 수있는 단일 호출이 유용한 경우가 많습니다. 플래그 섹션의 예에서 볼 수 있듯이 IEEE 상태를 수정하는 매우 일반적인 패턴은 블록 또는 서브 루틴의 범위 내에서만 변경하는 것입니다. 따라서 프로그래머는 블록에서 각 출구를 찾고 상태가 복원되었는지 확인해야합니다. 블록 범위에서 상태를 정확하게 설정하기위한 언어 지원은 여기에서 매우 유용합니다. Modula-3은 트랩 핸들러를위한이 아이디어를 구현하는 하나의 언어입니다 [Nelson 1991].

IEEE 표준을 언어로 구현할 때 고려해야 할 몇 가지 사소한 사항이 있습니다. 이후 *X* - *X* = 모든 +0 *X* , [(27)] (0) - (0) = 0. 그러나-(+ 0) = -0이므로 -*x* 를 0-x로 정의해서는 안됩니다. NaN이 다른 수 (다른 NaN 포함)와 결코 같지 않기 때문에 NaN의 도입이 혼란 스러울 수 있으므로 *x* = *x* 가 더 이상 항상 참이 아닙니다. 사실, *x x* 표현식 은 IEEE 권장 함수 가 제공되지 않는 경우 NaN을 테스트하는 가장 간단한 방법 입니다. 또한 NaN은 다른 모든 숫자에 대해 순서가 지정되지 않으므로 *x y*≠ `isnan` ≤ 로 정의 할 수 없는 *X* > *Y* . NaN의 도입으로 부동 소수점 숫자가 부분적으로 정렬되기 때문에 `compare`<, =,> 또는 정렬되지 않은 함수 중 하나를 반환 하는 함수 를 사용하면 프로그래머가 비교를 더 쉽게 처리 할 수 있습니다.

IEEE 표준은 피연산자가 NaN 인 경우 NaN을 반환하는 기본 부동 소수점 연산을 정의하지만 이것이 복합 연산에 대한 최상의 정의는 아닐 수도 있습니다. 예를 들어 그래프를 그리는 데 사용할 적절한 배율을 계산할 때 값 집합의 최대 값을 계산해야합니다. 이 경우 max 연산이 NaN을 무시하는 것이 합리적입니다.

마지막으로 반올림이 문제가 될 수 있습니다. IEEE 표준은 반올림을 매우 정확하게 정의하며 반올림 모드의 현재 값에 따라 다릅니다. 이것은 때때로 유형 변환의 암시 적 반올림 정의 또는 `round`언어 의 명시 적 함수 와 충돌 합니다. 즉, IEEE 반올림을 사용하려는 프로그램은 자연어 프리미티브를 사용할 수 없으며 반대로 언어 프리미티브는 계속 증가하는 IEEE 머신에서 구현하는 데 비효율적입니다.

최적화 도구

컴파일러 텍스트는 부동 소수점 주제를 무시하는 경향이 있습니다. 예를 들어 Aho et al. [1986]은로 교체 `x/2.0`를 언급하여 `x*0.5`독자 `x/10.0`가로 교체해야 한다고 가정하게 합니다 `0.1*x`. 그러나이 두 표현식은 이진 시스템에서 동일한 의미를 갖지 않습니다. 0.1은 이진으로 정확하게 표현 될 수 없기 때문입니다. 이 교과서는 대체 제안 `x*y-x*z`에 의해 `x*(y-z)`우리가이 두 식을 때 매우 다른 값을 가질 수 있음을 알 경우에도, *y*로 ≈지. 옵티마이 저가 언어 정의를 위반해서는 안된다는 점에 주목하여 코드를 최적화 할 때 어떤 대수적 신원도 사용할 수 있다는 진술을 규정하지만 부동 소수점 의미가 그다지 중요하지 않다는 인상을 남깁니다. 언어 표준이 괄호를 준수해야한다고 지정하는지 여부에 관계없이 위에서 설명한 대로와 `(x+y)+z`완전히 다른 대답을 가질 수 있습니다 `x+(y+z)`. 다음 코드에서 설명하는 괄호 보존과 밀접하게 관련된 문제가 있습니다.

```
eps = 1;

do eps = 0.5 * eps; 동안 (eps + 1> 1);
```

:

이것은 기계 입실론에 대한 추정치를 제공하기 위해 설계되었습니다. 최적화 컴파일러가 *eps* + 1> 1 *eps* > 0을 발견하면 프로그램이 완전히 변경됩니다. 1 *x* 가 여전히 *x* (*x e*) 보다 크 도록 가장 작은 숫자 *x* 를 계산하는 대신 *x* / 2가 0 (*x*)으로 반올림 되는 가장 큰 숫자 *x* 를 계산합니다 . 이러한 종류의 "최적화"를 피하는 것은 매우 중요하므로 완전히 망가진 매우 유용한 알고리즘을 하나 더 제시 할 가치가 있습니다.⇔ ⊕ ≈ ≈ β^{-p} ≈ $\beta^{e_{min}}$

수치 적분 및 미분 방정식의 수치 솔루션과 같은 많은 문제는 많은 항으로 합계를 계산하는 것과 관련이 있습니다. 각 추가는 잠재적으로 .5 ulp만큼 큰 오류를 유발할 수 있으므로 수천 개의 항을 포함하는 합계는 상당한 반올림 오류를 가질 수 있습니다. 이를 수정하는 간단한 방법은 부분 합계를 배정 밀도 변수에 저장하고 배정 밀도를 사용하여 각 덧셈을 수행하는 것입니다. 계산이 단 정밀도로 수행되는 경우 대부분의 컴퓨터 시스템에서 배정 밀도로 합계를 수행하는 것이 쉽습니다. 그러나 계산이 이미 배정 밀도로 수행되고있는 경우 정

밀도를 두 배로 늘리는 것은 그렇게 간단하지 않습니다. 때때로 옹호되는 한 가지 방법은 숫자를 정렬하고 가장 작은 것에서 가장 큰 것까지 더하는 것입니다. 하나,

정리 8 (카한 합산 공식)

$\Sigma_{j=1}^{N} x_j$ 다음 알고리즘을 사용하여 계산 한다고 가정하십시오.

```
S = X [1];
C = 0;
j = 2에서 N {
    Y = X [j]-C;
    T = S + Y;
    C = (T-S)-Y;
    S = T;
}
```

그러면 계산 된 합계 *S* 는 $\Sigma x_j(1+\delta_j) + O(N\varepsilon^2)\Sigma|x_j|, \quad (|\delta_j| \le 2\varepsilon)$ *where*와 같습니다 .

순진한 공식을 사용하면 Σx_j 계산 된 합계는 $\Sigma x_j(1+\delta_j)$ where $|\delta_j| < (n-j)\varepsilon$. 이것을 Kahan 합계 공식의 오류와 비교하면 극적인 개선을 보여줍니다. 각 summand는 간단한 공식에서 *ne* 만큼 큰 섭동 대신 2 *e* 에 의해서만 섭동됩니다 . 자세한 내용은 Errors In Summation에 있습니다.δ

부동 소수점 산술이 대수의 법칙을 따랐다 고 믿었던 최적화 프로그램은 *C* = [*T* - *S*] -*Y* = [(*S* + *Y*) -*S*] -*Y* = 0으로 결론을 내릴 수 있으며, 이는 알고리즘을 완전히 쓸모 없게 만듭니다. 이러한 예는 수학적 실수를 유지하는 대수적 ID를 부동 소수점 변수를 포함하는 표현식에 적용 할 때 최적화 프로그램이 극도로주의해야한다고 요약 할 수 있습니다.

최적화 프로그램이 부동 소수점 코드의 의미를 변경할 수있는 또 다른 방법은 상수를 포함합니다. 표현식 `1.0E-40*x`에는 10 진수를 2 진 상수로 변환하는 암시 적 10 진수에서 2 진으로의 변환 연산이 있습니다. 이 상수는 바이너리로 정확하게 표현할 수 없기 때문에 정확하지 않은 예외가 발생해야합니다. 또한식이 단 정밀도로 평가되는 경우 언더 플로 플래그를 설정해야합니다. 상수가 정확하지 않기 때문에 이진수로의 정확한 변환은 IEEE 반올림 모드의 현재 값에 따라 다릅니다. 따라서 변환하는 최적화 프로그램`1.0E-40`컴파일 타임에 바이너리로 변경하면 프로그램의 의미가 변경됩니다. 그러나 사용 가능한 가장 작은 정밀도로 정확하게 표현할 수있는 27.5와 같은 상수는 항상 정확하고 예외를 발생시킬 수없고 반올림 모드의 영향을받지 않기 때문에 컴파일 타임에 안전하게 변환 할 수 있습니다. 컴파일 타임에 변환 할 상수는 `const pi = 3.14159265`.

공통 하위 표현식 제거는 다음 코드에 설명 된대로 부동 소수점 의미를 변경할 수있는 최적화의 또 다른 예입니다.

```
C = A * B;
RndMode = 위로
D = A * B;
```

A*B공통 하위 표현식으로 보일 수 있지만 두 평가 사이트에서 반올림 모드가 다르기 때문이 아닙니다. 마지막 세 가지 예 : $x = x$ 는 부울 상수로 대체 할 수 없습니다 . x 가 NaN true이면 실패하기 때문입니다 . - $X = 0 - X$ 실패에 대한 $X = 0$; 및 $X < Y$는 반대 아니다 의 X , Y NaN이 보통 부동 소수점 숫자보다 둘 이상도 이하이기 때문에. $\geq$

이러한 예에도 불구하고 부동 소수점 코드에서 수행 할 수있는 유용한 최적화가 있습니다. 우선, 부동 소수점 숫자에 유효한 대수적 정체성이 있습니다. IEEE 산술의 몇 가지 예는 $x + y = y + x$, $2 \times x = x + x$, $1 \times x = x$ 및 $0.5 \times x = x / 2$입니다. 그러나 이러한 단순한 ID조차도 CDC 및 Cray 슈퍼 컴퓨터와 같은 일부 컴퓨터에서는 실패 할 수 있습니다. 명령어 스케줄링 및 인라인 절차 대체는 잠재적으로 유용한 두 가지 다른 최적화입니다. [28]

최종 예를 들어, 표정 고려 dx = x*y, x및 y단 정밀도 변수, 그리고 dx배정도입니다. 두 개의 단 정밀도 숫자를 곱하여 배정 밀도 숫자를 생성하는 명령어가있는 기계 dx = x*y에서는 피연산자를 배정 밀도로 변환 한 다음 배정 밀도에서 배정 밀도 곱하기를 수행하는 일련의 명령어로 컴파일하는 대신 해당 명령어에 매핑 할 수 있습니다.

일부 컴파일러 작성자는 $(x + y) + z$ 를 $x + (y + z)$로 변환하는 것을 금지하는 제한을 무관 한 것으로 간주하고 이식 불가능한 트릭을 사용하는 프로그래머에게만 관심이 있습니다. 아마도 그들은 부동 소수점 숫자가 실수를 모델링하고 실수와 동일한 법칙을 따라야한다는 것을 염두에 두었을 것입니다. 실수 의미론의 문제는 구현 비용이 매우 비싸다는 것입니다. 두 n 비트 숫자가 곱해질 때마다 곱은 2 n 비트를 갖게됩니다 . 매번 두 n넓은 간격의 지수가있는 비트 번호가 추가되고 합계의 비트 수는 n + 지수 사이의 공간입니다. 합계는 최대 $(e^{max} - e^{min})$ + n 비트 또는 대략 $2 \cdot e^{max}$ + n 비트를 가질 수 있습니다 . 수천 개의 연산 (예 : 선형 시스템 해결)을 포함하는 알고리즘은 곧 중요한 비트가 많은 숫자에서 작동하며 절망적으로 느려질 것입니다. sin 및 cos와 같은 라이브러리 함수의 구현은 훨씬 더 어렵습니다. 이러한 초월 적 함수의 가치는 합리적이지 않기 때문입니다. 정확한 정수 산술은 종종 lisp 시스템에서 제공되며 일부 문제에 유용합니다. 그러나 정확한 부동 소수점 산술은 거의 유용하지 않습니다.

사실은 $(x + y) + z \neq x + (y + z)$ 사실을 활용 하고 경계가있을 때마다 작동 하는 유용한 알고리즘 (예 : Kahan 합계 공식)이 있다는 것입니다.

$$a \oplus b = (a + b) (1 + \delta)$$

홀드 (뿐만 아니라-, × 및 /에 대한 유사한 경계). 이러한 경계는 거의 모든 상용 하드웨어에 적용되기 때문에 수치 프로그래머가 그러한 알고리즘을 무시하는 것은 어리석은 일이며 컴파일러 작성자가 부동 소수점 변수가 실수 의미론을 갖는 것처럼 가장하여 이러한 알고리즘을 파괴하는 것은 무책임 할 것입니다.

예외 처리

지금까지 논의 된 주제는 주로 정확성과 정밀도의 시스템 영향에 관한 것입니다. 트랩 핸들러는 또한 몇 가지 흥미로운 시스템 문제를 제기합니다. IEEE 표준이 강력하게 사용자가 예외의 다섯 개 가지 클래스 각각에 대해 트랩 처리기 및 섹션 지정할 수 있습니다 것이 좋습니다 트랩 처리기 , 사용자 정의 트랩 처리기의 일부 응용 프로그램을 주었다. 유효하지 않은 연산 및 0으로 나누기 예외의 경우 처리기에 피연산자가 제공되어야하며 그렇지 않으면 정확히 반올림 된 결과가 제공됩니다. 사용중인 프로그래밍 언어에 따라 트랩 처리기는 프로그램의 다른 변수에도 액세스 할 수 있습니다. 모든 예외에 대해 트랩 처리기는 수행중인 작업과 대상의 정밀도를 식별 할 수 있어야합니다.

IEEE 표준은 연산이 개념적으로 직렬이고 인터럽트가 발생하면 연산과 피연산자를 식별할 수 있다고 가정합니다. 파이프 라이닝 또는 여러 산술 단위가있는 기계에서 예외가 발생하면 트랩 처리기가 프로그램 카운터를 검사하도록하는 것만으로는 충분하지 않을 수 있습니다. 트랩 된 작업을 정확히 식별하기위한 하드웨어 지원이 필요할 수 있습니다.

다음 프로그램 조각에서 또 다른 문제를 설명합니다.

```
x = y * z;
z = x * w;
a = b + c;
d = a / x;
```

두 번째 곱셈에서 예외가 발생하고 트랩 처리기가 값을 사용하려고한다고 가정합니다 `a`. 더하기와 곱하기를 병렬로 수행 할 수있는 하드웨어에서 옵티마이 저는 아마도 두 번째 곱하기 전에 더하기 연산을 이동하여 더하기가 첫 번째 곱셈과 병렬로 진행될 수 있습니다. 따라서 두 번째 곱하기 트랩 `a = b + c`이 이미 실행 된 경우 잠재적으로 결과가 변경됩니다.`a`. 컴파일러가 이러한 종류의 최적화를 피하는 것은 합리적이지 않습니다. 모든 부동 소수점 연산이 잠재적으로 트랩 할 수 있으므로 사실상 모든 명령 스케줄링 최적화가 제거되기 때문입니다. 이 문제는 트랩 처리기가 프로그램의 변수에 직접 액세스하지 못하도록 금지하여 방지 할 수 있습니다. 대신 핸들러에 피연산자 또는 결과를 인수로 제공 할 수 있습니다.

그러나 여전히 문제가 있습니다. 조각에서

```
x = y * z;
z = a + b;
```

두 명령은 병렬로 실행될 수 있습니다. 곱하기가 함정되면 `z`특히 더하기가 일반적으로 곱하기보다 빠르기 때문에 인수 가 이미 더하기로 덮어 쓰여졌을 수 있습니다. IEEE 표준을 지원하는 컴퓨터 시스템 `z`은 하드웨어에서 또는 컴파일러가 처음에 이러한 상황을 피하도록 하여의 값을 저장하는 방법을 제공해야합니다 .

W. Kahan은 이러한 문제를 피하기 위해 트랩 처리기 대신 전 치환 사용을 제안했습니다 . 이 방법에서 사용자는 예외가 발생했을 때 결과로 사용하고자하는 예외와 값을 지정합니다. 예를 들어, (죄 컴퓨팅을위한 코드에서 그 가정 *X*)를 / *X* , 사용자가 해당 결정 *X* = 0 이에 대한 테스트 방지하기 위해 성능을 향상시키는 것이 매우 드문 *X* 이 사건을 = 0, 대신 처리 할 때 0 / 0 트랩이 발생합니다. IEEE 트랩 처리기를 사용하여 사용자는 값 1을 반환하는 처리기를 작성하고 sin *x* / *x*를 계산하기 전에 설치합니다.. 사전 대체를 사용하면 사용자는 유효하지 않은 연산이 발생할 때 값 1을 사용해야한다고 지정합니다. Kahan은 예외가 발생하기 전에 사용할 값을 지정해야하기 때문에이 사전 대체를 호출합니다. 트랩 처리기를 사용할 때 반환되는 값은 트랩이 발생할 때 계산 될 수 있습니다.

사전 대체의 장점은 하드웨어 구현이 간단하다는 것입니다. [29] 예외 유형이 결정되는 즉시 원하는 작업 결과를 포함하는 테이블을 인덱싱하는 데 사용할 수 있습니다. 사전 대체에는 몇 가지 매력적인 특성이 있지만 IEEE 표준이 널리 채택되어 하드웨어 제조업체에서 널리 구현할 가능성이 낮습니다.

세부 사항

이 백서에서는 부동 소수점 산술의 속성과 관련하여 많은 주장을했습니다. 우리는 이제 부동 소수점이 흑 마법이 아니라 주장을 수학적으로 검증 할 수있는 간단한 주제임을 보여줍니다. 이 섹션은 세 부분으로 나뉩니다. 첫 번째 부분에서는 오류 분석을 소개하고 반올림 오류 섹션에 대한 세부 정보를 제공합니다 . 두 번째 부분에서는 이진에서 십진으로의 변환에 대해 살펴보고 IEEE 표준 섹션 의 일부 공백을 메 웁니다 . 세 번째 부분에서는 시스템 측면 섹션에서 예로 사용 된 Kahan 합계 공식에 대해 설명합니다 .

반올림 오류

반올림 오류에 대한 논의에서 단일 가드 숫자만으로도 덧셈과 뺄셈이 항상 정확하다는 것을 보장 할 수 있다고 언급했습니다 (정리 2). 이제이 사실을 확인합니다. 정리 2는 뺄셈과 덧셈의 두 부분으로 구성됩니다. 빼기 부분은

정리 9

*x*와 *y*가 매개 변수 와 *p* 가있는 형식의 양의 부동 소수점 숫자 이고 뺄셈이 *p + 1* 자리 *(*즉, 보호 숫자 *1* 개*)*로 수행되는 경우 결과의 상대 반올림 오류는 다음보다 작습니다. β

$$\left(\frac{\beta}{2}+1\right)\beta^{-p}=\left(1+\frac{2}{\beta}\right)\beta^{-p}\le 2\varepsilon$$

증명

필요한 경우 *x* 와 *y*를 교환 하여 *x* > y가 되도록합니다 . *x* 가 x_0으로 표시 되도록 *x* 와 *y* 를 스케일링하는 것도 무해합니다 . $x_1 \ldots x_{p-1} \times \beta^0$. *y* 가 y_0으로 표시되는 경우 . $y_1 \ldots y_{p-1}$ 이면 차이가 정확합니다. *y* 가 *0* 으로 표시되는 경우 . $y_1 \ldots y_p$ β , 보호 숫자는 계산 된 차이가 부동 소수점 숫자로 반올림 된 정확한 차이가되도록 보장하므로 반올림 오류는 최대 *e* 입니다. 일반적으로 *Y*는 0.0 = ... 0 *Y*의 $Y_{k+1} \ldots Y_{k+p}$를 하고 $\bar{y}$ 있을 *Y*가 잘린 *P* + 1 디지트. 그때

(15) $Y - \bar{y} < (\beta - 1)(\beta^{-(p)-1} + \beta^{-p-2} + \ldots + \beta^{-p-k})\beta\beta\beta$

가드 디지트의 정의에서의 계산 된 값 *X* - *Y*는 이고 , *X* - $\bar{y}$ 원형 인 부동 소수점 수 (것으로 *X* - $\bar{y}$) + δ, 라운딩 에러 δ를 만족

(16) $|\delta| \le (\beta/2)\beta^{-(p)}$

정확한 차이는 *X* - *Y* 오류 (그래서, *X* - *Y*) - (*X* - $\bar{y}$ + δ) = $\bar{y}$ - *Y* + δ. 세 가지 경우가 있습니다. x-y ≥1이면 상대 오차는 다음과 같이 제한됩니다.

(17) $\beta^{-(p)}[(\beta - 1)(\beta^{-1} + \ldots + \beta^{-k}) + \beta/2] < \beta^{-p}(\beta + 1/2) \cdot \frac{y-\bar{y}+\delta}{1} \le \beta\beta\beta\beta\beta\beta\beta\beta$

둘째, 만약 *X* - $\bar{y}$ <1 다음 δ= 0은 가장 작은 때문에 *X* - *Y*는 하다 할

$$1.0 - 0.\underbrace{0\ldots0}_{k}\underbrace{\rho\ldots\rho}_{p} > (\beta - 1)(\beta^{-1} + \ldots + \beta^{-k}) \text{ 여기서, } \beta = \beta - 1, \beta\beta\rho\beta$$

이 경우 상대 오차는

(18) $$\frac{y-\bar{y}+\delta}{(\beta-1)(\beta^{-1}+\ldots+\beta^{-k})} < \frac{(\beta-1)\beta^{-p}(\beta^{-1}+\ldots+\beta^{-k})}{(\beta-1)(\beta^{-1}+\ldots+\beta^{-k})} = \beta^{-p}$$.

마지막 경우는 언제 *X* Y <1이지만 - *X* - ȳ ≥1 유일한 방법이 일어날 수있는 경우 , *X* - ȳ = 1 인 경우에 δ= 0 그러나 만약 δ다음 = 0 (18)에 적용되도록 다시 상대 에러에 의해 제한된다 $^{-}P < {}^{-}P$ (+ 1 / 2). 지βββ

경우 β= 2, 바운드 정확히 2 즉 ,이 결합이 달성되는 $X = 1 + 2^{2-P}$ 와 $Y = 2^{(1)-(P)}$ - (2) $^{-1(2)}P$ 와 한계 *P* . 동일한 부호의 숫자를 추가 할 때 다음 결과에서 알 수 있듯이 정확도를 높이기 위해 보호 숫자가 필요하지 않습니다.→ ∞

정리 10

x 0 및 *y* 0 이면 보호 숫자가 사용되지 않더라도 *x* + *y* 계산시 상대 오차는 최대 2 입니다.≥ ≥ ε

증명

k 가드 숫자를 사용한 더하기 알고리즘은 빼기 알고리즘 과 유사합니다. 경우 의 *X* , *Y*는 시프트 의 *Y* 의 기수 점까지 오른쪽 *X* 및 *Y*는 정렬된다. *p* + *k* 위치를 지나서 이동 한 모든 숫자를 버립니다 . 이 두 *p* + *k* 숫자 의 합을 정확하게 계산합니다. 그런 다음 *p* 자리로 반올림하십시오 .≥
가드 숫자가 사용되지 않을 때 정리를 확인합니다. 일반적인 경우는 비슷합니다. *x y* 0이고 *x* 가 *d.dd ... d* × ^{0} 형식이되도록 스케일링 되었다고 가정 할 때 일반성이 손실되지 않습니다 . 첫째, 수행이 없다고 가정합니다. 그 다음의 숫자는 끝으로 이동 *(Y)*은 보다 작은 값을 갖는다 $^{-P+1}$ 및 상대 오차 미만 정도로 합은 적어도 $1^{-P+1} / 1 = 2E$. 수행이있는 경우 이동으로 인한 오류를 반올림 오류에 더해야합니다. ≥ ≥ βββ

$\frac{1}{2}\beta^{-p+2}$.

합계가 최소 β이므로 상대 오차는

$\left(\beta^{-p+1}+\frac{1}{2}\beta^{-p+2}\right)/\beta = (1+\beta/2)\beta^{-p} \le 2\varepsilon$. 지

이 두 정리를 결합하면 정리 2가됩니다. 정리 2는 하나의 작업을 수행하는 데 대한 상대 오차를 제공합니다. $x^2 - y^2$ 와 (*x* + *y*) (*x* - *y*) 의 반올림 오차를 비교하려면 여러 연산의 상대 오차를 알아야합니다. *x* ⊖ *y* 의 상대 오차 는 $_1$ = [(*x y*)-(*x* - *y*)] / (*x* - *y*) 이며, 이는 | $_1$ | 2 *e* . 또는 다른 방법으로 작성하려면δ ⊖ δ ≤

(19) *x* ⊖ *y* = (*x* - *y*) (1 + $_1$), | $_1$ | 2 개 *e*δ δ ≤

비슷하게

(20) *x y* = (*x* + *y*) (1 + $_2$), | $_2$ | 2 개 *e*⊕ δ δ ≤

정확한 곱을 계산 한 다음 반올림하여 곱셈을 수행한다고 가정하면 상대 오차는 최대 .5ulp 이므로

(21) *u v* = *uv* (1 + $_3$), | $_3$ | 이자형⊗ δ δ ≤

모든 부동 소수점 숫자 *u* 및 *v* . 이 세 방정식을 합치면 (*u* = *x* ⊖ *y* 및 *v* = *x y*)⊕

(22) $(x \ominus y) \otimes (x y) = (x - y) (1 + _1) (x + y) (1 + _2) (1 + _3) \oplus \delta\delta\delta$

따라서 (*x* - *y*) (*x* + *y*)를 계산할 때 발생하는 상대 오차 는 다음과 같습니다.

(23) $$\frac{(x-y)\ominus(x+y)-(x^2-y^2)}{(x^2-y^2)} = (1+\delta_1)(1+\delta_2)(1+\delta_3)-1$$

이 상대 오차는 $_1 + _2 + _3 + _1 _2 + _1 _3 + _2 _3 + _1 _2 _3$ 와 같 으며 $5 + 8^{2로}$ 제한됩니다 . 즉, 최대 상대 오차는 약 5 개의 반올림 오차입니다 (*e* 는 작은 수이므로 e^2 는 거의 무시할 수 있음).$\delta\delta\delta\delta\delta\delta\delta\delta\delta\delta\delta\varepsilon\varepsilon$

(*x x*) (*y y*) 의 유사한 분석은 상대 오차에 대해 작은 값을 생성 할 수 없습니다. *x* 와 *y* 의 두 개의 인접 값 을 x^2 -y ^{2에} 연결 하면 일반적으로 상대 오차가 상당히 크기 때문입니다. 이를 확인하는 또 다른 방법은 (*x y*) (*x y*)에서 작동했던 분석을 복제 하여 $\otimes \ominus \otimes \ominus \otimes \oplus$

$(X \otimes, X) \ominus (Y \otimes, Y) = X^{(2)} (1 + _1) - (Y)^2 (1 + _2) (1 + _3) = ((X^2 - 예^2) (1 + _1) + (_1 - _2) y^2) (1 + _3)\delta\delta\delta$
$\delta\delta\delta\delta$

경우 *X* 및 *Y*는 인근에, 에러 항 ($_1 - _2$), $Y^{(2)}$ 결과 한 크게 할 수있는 $X^2 - Y^2$. 이 계산은 공식적으로 (우리의 주장을 정당화 *X* - *Y* () *X* + *Y*가)보다 더 정확 $X^2 - y$를 ^{2} .$\delta\delta$

다음으로 삼각형 면적에 대한 공식 분석으로 넘어갑니다. (7)로 계산할 때 발생할 수있는 최대 오류를 추정 하려면 다음과 같은 사실이 필요합니다.$\leq \leq$

정리 11

가드 디지트와 *y* / *2 x 2y*를 사용하여 빼기를 수행하면 *x*-*y*가 정확하게 계산됩니다.$\leq \leq$

증명

경우 유의 *X* 및 *Y*는 동일 지수를 다음 확실히 $X \ominus$, *Y*는 정확하다. 그렇지 않으면 정리의 조건에서 지수는 최대 1만큼 다를 수 있습니다. 필요한 경우 *x* 와 *y*를 스케일링하고 교환하여 0 *y x* , *x* 가 $x_{0으로}$ 표시되도록 합니다. $x_1 \ldots x_{p-1}$ 이고 *y* 는 0입니다. $y_1 \ldots y_p$. 이어서 계산 알고리즘 *X*는 *Y*를 계산한다 *(X)* - *(Y)*를$\leq \leq \ominus$ 정확하고 부동 소수점 숫자로 반올림합니다. 차이가 0. $d_1 \ldots d_p$ 이면 차이는 이미 *p* 자리 길이이며 반올림 할 필요가 없습니다. 이후 , $X \leq 2$, *Y* , *X* - *Y (Y)*를 , 이후 , *Y*는 폼 0이다 $(D)_{(1)} \ldots D_P$, 그렇다 , *X* - *Y* . 지$\leq$

경우 $\beta > 2$, 정리 (11)의 가설에 의해 대체 될 수 없다 $Y / \beta \leq X$, *Y* ; 더 강한 조건 *y* / 2 *x* 2 *y* 가 여전히 필요합니다. 정리 10의 증명 직후 (*x* - *y*) (*x* + *y*) 의 오류 분석은 덧셈과 뺄셈의 기본 연산에서 상대 오차가 작다는 사실을 사용했습니다 (즉, 방정식 (19) 및 (20)). 이것은 가장 일반적인 종류의 오류 분석입니다. 그러나 공식 (7)을 분석 하려면 다음 증명에서 볼 수 있듯이 정리 11이라는 더 많은 것이 필요합니다.$\leq \beta \leq \leq$

정리 12

빼기가 가드 숫자를 사용하고 *a, b* 및 *c*가 삼각형의 변 *(a b c)*이면 계산의 상대 오차 *(a + (b + c)) (c-(a-b)) (c + (a-b)) (a + (b-c))*는 *e <.005* 인 경우 최대 16 입니다.$\geq \geq$

ε

증명

요인을 하나씩 살펴 보겠습니다. 정리 10에서 *b c* = (*b* + *c*) (1 + $_1$), 여기서 $_1$ 은 상대 오차이고, | $_1$ | 2 . 그런 다음 첫 번째 요소의 값은⊕ δδδ ≤ ε

(*a* ⊕ (*b c*)) = (*a* + (*b c*)) (1 + $_2$) = (*a* + (*b* + c) (1 + $_1$)) (1 + $_2$),⊕⊕ δδδ

따라서

(+ *B* + *C*) ((가) 1 - 2) ^{2} [+ (*B* + *C*) ((가) 1 - 2)] * (1-2) (*B* 형 *C*) [+ (*B* + *C*) (1 + 2)] (1 + 2) (*a* + *b* + *c*) (1 + 2) ^{2} ε ≤ εε
≤⊕⊕
≤ εε
≤ ε

이것은 $_1$ 이 있다는 것을 의미 합니다.η

(24) (a ⊕(b ⊕c)) = (a + b + c) (1 + $_1$) ^{2} , | $_1$ | 2 .ηη≤ ε

다음 용어의 잠재적으로 치명적인 빨셈 포함 *C* 와 을 ⊖ b 하기 때문에, *b*는 반올림 오류가 있을 수 있습니다. 때문에, B 및 C를 삼각형,의 양쪽이다 (B) + *(C)* , 및 주문에이 결합 *C* 는 *b*를 *A*가 제공 *B* + *C* 2 *B* 2 . 따라서 *a* - *b* 는 정리 11의 조건을 충족합니다. 이것은 *a* - *b* = *a b* 가 정확함을 의미하므로 *c* (*a* ⊖ ≤ ≤ ≤ ≤ ≤ ≤ ⊖ ⊖ -b) 정리 9에서 다음과 같이 추정 할 수있는 무해한 빨셈입니다.

(25) (*c* ⊖ (*a* ⊖ *b*)) = (*c*- (*a* - *b*)) (1 + $_2$), | $_2$ | 2ηη≤ ε

세 번째 항은 정확히 두 양의 양의 합입니다.

(26) (*c* ⊕ (*a* ⊖ *b*)) = (c + (*a* - *b*)) (1 + $_3$), | $_3$ | 2ηη≤ ε

마지막으로 마지막 학기는

(27) (*a* ⊕ (*b* ⊖ *c*)) = (*a* + (*b* -c)) (1 + $_4$) ^{2} , | $_4$ | 2 ,ηη≤ ε

정리 9와 정리 10을 모두 사용합니다. 곱셈이 정확히 반올림되었다고 가정하면 *x y* = *xy* (1 +)와 | | , (24) , (25) , (26) 및 (27)을 결합 하면⊗ ζζ≤ ε

(*a* ⊕ (b ⊕c)) (c ⊖ (a ⊖ b)) (c ⊕(a ⊖ b)) (*a*⊕ (*b* ⊖ *c*)) (*a* + (*b* + c)) (*c*- (*a* - *b*)) (*c* + (*a* - *b*)) (a + (b- *c*)) *E*
≤

어디

E = (1 + $_1$) ^{2} (1 + $_2$) (1 + $_3$) (1 + $_4$) ^{2} (1 + $_1$) (1 + $_2$) (1 + $_3$)ηηηη ζζζ

E 의 상한 은 (1 + 2 ε) ^{6} (1 + ε) ^{3} 이며, 1 + 15 ε + O (^{2})로 확장됩니다 . 일부 작가는 단순히 O (*e* ^{2}) 용어 를 무시 하지만 설명하기 쉽습니다. (1 + 2) ^{6} (1 +) ^{3} = 1 + 15 + *R* ( ), *R* ( )은 양의 계수를 갖는 *e* 의 다항식 이므로의 증가 함수입니다 . 이후 *R* (005) = 0.505, *R* ( ) <1 모두 따라서 <.005 및 *E* (1 + 2εεεεεεεεεε ≤ε) ^{6} (1 + ε) ^{3} <1 + 16 ε. *E* 에 대한 하한을 얻으려면 1-15 ε - εR (ε) <E이고 ε<.005,

1-16 ε<(1-2 ε) ^{6} (1- ε) ^{3} 인 경우에 유의하십시오 . 이 두 경계를 결합하면 1-16 ε< *E* <1 + 16이 ε됩니다. 따라서 상대 오차는 최대 16 ε입니다. 지

정리 12는 공식 (7) 에 치명적인 취소가 없음을 확실히 보여줍니다 . 따라서 공식 (7) 이 수치 적으로 안정 하다는 것을 보여줄 필요는 없지만 전체 공식에 대한 경계를 갖는 것이 만족 스럽습니다. 이것이 취소의 정리 3이 제공하는 것입니다.

정리 3 증명

허락하다

q = (*a* + (*b* + *c*)) (*c*- (*a* - *b*)) (*c* + (*a* - *b*)) (*a* + (*b* - *c*))

과

Q = (*a* ⊕ (*b c*))(c (*a b*))(*c* (*a b*))(*a* (*b c*)).⊕⊗⊖ ⊖ ⊗⊕ ⊖ ⊗⊕ ⊖

그러면 정리 12는 Q = *q* (1 + δ)이고 16 이라는 것을 보여줍니다 . 확인하기 쉽습니다.δ ≤ ε

(28) $1-0.52|\delta| \le \sqrt{1-|\delta|} \le \sqrt{1+|\delta|} \le 1+0.52|\delta|$

제공 / (52). 0.04 (2) 0.15, 이후 | | 16 16 (.005) = .08, 조건을 만족합니다. 그러므로 δ ≤ ≈δ≤ε ≤δ

$\sqrt{Q} = \sqrt{q(1+\delta)} = \sqrt{q}(1+\delta_1)$,

와 함께 | $_1$ | .52 | | 8.5 . 제곱근이 .5 ulp 이내로 계산되면 계산시 오류 는 (1 + $_1$) (1 + $_2$)이며 | $_2$ | . 경우 = 2, 더 이상의 오류가 발생하지 않으면 4로 나누어 때 최선을 다하고, 또 하나 개의 요인 1 + $_3$ 에 | $_3$ | 나눗셈에 필요하며 정리 12의 증명 방법을 사용하면 (1 + $_1$) (1 + $_2$) (1 + $_3$) 의 최종 오차 한계는 1 + $_4$ 가 지배 하며 | $_4$ | 11 . 지δ ≤δ≤ε$\sqrt{Q}$ δδδ≤ εβδδ≤ εδδδδδ≤ε

정리 4의 진술 바로 뒤의 휴리스틱 설명을 정확하게하기 위해 다음 정리는 μ (*x*)가 상수에 얼마나 근접한지 설명합니다 .

정리 13

만약 μ (*X*) = LN (1 + *X*) / *X* , 다음에 대한 0 , *X* , μ (*X*) 1 및 유도체 만족 | μ ′ (*x*) | . ≤ ≤ ≤ ≤ ≤

증명

참고 μ (*X*) = 1 - *X* / 2 + *X* ^{2} / 3 - ... 정도로 감소에 대한 용어와 번갈아 시리즈이며, *X* ≤ 1, μ (*X*) ≥1 - *X* / 2 ≥1/2. μ에 대한 계열 이 번갈아 가며 μ (*x*) ≤1. μ '(*x*) 의 Taylor 계열 도 번갈아 가며 *x* 에 감소하는 항이 있으면 -μ '(*x*) - + 2 배≤ ≤ ≤ / 3 또는 -μ '(*x*) 0, 따라서 | μ '(*x*) | . 지≤≤≤

정리 4의 증명

ln에 대한 Taylor 시리즈 이후

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$$

는 교대 계열, 0 < *x* -ln (1 + *x*) < x^2 / 2이며, ln (1 + *x*)를 *x* 로 근사 할 때 발생하는 상대 오차 는 *x* / 2로 제한됩니다 . 1 *x* = 1이면 | *x* | < 이므로 상대 오차는 / 2 로 제한됩니다 .⊕ εε

1 *x* 1이면 1 *x* = 1 + 를 통해 정의 합니다. 그러면 0 *x* <1이므로 (1 *x*) 1 = . 나눗셈과 로그가 ulp 내로 계산되는 경우 ln (1 + *x*) / ((1 + *x*)-1) 식의 계산 된 값은 다음과 같습니다.⊕ ≠ ˆ⊕ ˆ ≤ ⊕ ⊖ˆ 1 2

(29) $\frac{\ln(1\oplus x)}{(1\oplus x)\ominus 1}$ (1 + $_1$) (1 + $_2$) = (1 + $_1$) (1 + $_2$) = μ ( ) (1 + $_1$) (1 + $_2$)δδ $\frac{\ln(1+\hat{x})}{\hat{x}}$ δδ ˆδδ

어디 | $_1$ | 및 | $_2$ | . μ ( ) 를 추정하려면 평균값 정리를 사용하십시오.δ ≤ εδ ≤ ε ˆ

(30) μ (ˆ) - μ (*X*) = (ˆ - *X*) μ '를 (ξ)

일부 ξ사이의 *X* 및 ˆ . 의 정의에서 ˆ 다음과 같습니다. | ˆ - *X* | , 그리고 이것을 정리 13과 결합하면 | μ ( ) -μ (*x*) | / 2 또는 | μ ( ) / μ (*x*)-1 | / (2 | μ (*x*) |) 이는 μ ( ) = μ (*x*) (1 + $_3$)을 의미하며 | $_3$ | . 마지막으로 *x*를 곱 하면 최종 ≤ ε ˆ ≤ ε ˆ ≤ ε ≤ ε ˆ δδ ≤ εδ $_4$ 이므로 계산 된 값은

x · ln (1 *x*) / ((1 *x*) 1)⊕⊕ ⊖

이다

$$\frac{x\ln(1+x)}{(1+x)-1}(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\delta_4), \quad |\delta_i| \le \varepsilon$$

ε0.1 미만 이면 확인하기 쉽습니다.

(1 + $_1$) (1 + $_2$) (1 + $_3$) (1 + $_4$) = 1 + ,δδδδδ

와 함께 | δ| ≤5 ε. 지

공식 (19) , (20) , (21)을 사용한 오류 분석의 흥미로운 예가 2 차 공식에서 발생합니다 $(-b\pm\sqrt{b^2-4ac})/2a$. 취소 섹션에서는 방정식을 다시 작성하여 ± 연산으로 인한 잠재적 취소를 제거하는 방법을 설명 했습니다. 그러나 계산할 때 발생할 수있는 또 다른 소거 전위가 *D* = *B* $_{(2)}$ 4 - *AC*는 . 이것은 공식의 단순한 재 배열로 제거 될 수 없습니다. 대략적으로 말하면 b^2 ≈ 4 *ac*, 반올림 오차는 2 차 공식으로 계산 된 근에서 최대 절반 자릿수를 오염시킬 수 있습니다. 여기에 비공식적 증거가있다 (2 차 공식의 오차를 추정하는 또 다른 접근 방식은 Kahan [1972]에 나왔다).

b^2 4 *ac* 인 경우 반올림 오류는 *2* 차 공식으로 계산 된 근의 자릿수를 최대 절반까지 오염시킬 수 있습니다 . ≈ $(-b\pm\sqrt{b^2-4ac})/2a$

증명 : 쓰기 (*b* b) (4 *a c*) = (b^2 (1 + $_1$)-4 *ac* (1 + $_2$)) (1 + $_3$), 여기서 | 나는 | . 30은 사용 *D* = B^2 - 4 맥 이 같이 다시 쓸 수 (*D* (1 + $_1$) - (4) *AC* ($_2$ - $_1$)) (1 + $_3$). 이 오차의 크기에 대한 추정치를 얻으려면 *i*의 2 차 항을 무시하십시오 .이 경우 절대 오차는 *d*입니다. ⊗⊖ ⊗ δδδδ ≤ ε δδδδδ($_1$ + $_3$)-4ac $_4$, 여기서 | $_4$ | = | $_{(1)}$ - $_2$ | 2 .

이후 첫 번째 항 d ($_1$ + $_3$)은 무시할 수 있습니다. 두 번째 항을 추정하려면 $ax^2 + bx + c = a(x - r_1)(x - r_2)$이므로 $ar_1 r_2 = c$라는 사실을 사용하십시오 . b^2 4 이후δ δδδδδ≤ ε$^{d « 4ac}$ δδ ≈ac , $r_1 r_2$ 이므로 두 번째 오류 항은 . 따라서 계산 된 값 은≈ $4ac\delta_4 \simeq 4a^2 r_1 \delta_4^2 \sqrt{d}$

$\sqrt{d + 4a^2 r_1^2 \delta_4}$.

불평등

$$p - q \le \sqrt{p^2 - q^2} \le \sqrt{p^2 + q^2} \le p + q,\ p \ge q > 0$$

것을 보여줍니다

$$\sqrt{d + 4a^2 r_1^2 \delta_4} = \sqrt{d} + E$$,

어디

$$|E| \le \sqrt{4a^2 r_1^2 |\delta_4|}$$,

그래서 절대 오차 $\sqrt{d}/2$ A는 관한 것이다 $r_1\sqrt{\delta_4}$. 이후 $_4$ $^{-P}$, 중, 따라서 절대 오차 파괴한다 뿌리의 비트의 하반부는 r에 $_{1\ 개}$ R $_2$. 뿌리의 계산으로 산출 수반 때문에 즉, 및이 식의 저 차 반에 대응하는 위치에서 의미있는 비트가없는 $R_{I를}$, 다음의 하위 차수 비트 의 R은 $_{난}$ 의미가 없다. $_{지}\delta \approx \beta\sqrt{\delta_4} \simeq \beta^{-p/2} r_1 \sqrt{\delta_4} \approx (\sqrt{d})/(2a)$

마지막으로 정리 6의 증명을 살펴 보겠습니다. 이는 정리 14 및 정리 8 섹션에서 입증 된 다음 사실을 기반으로합니다 .

정리 14

하자 0 <K <p 및 집합 m = ^{K} + 1 , 및 부동 소수점 연산이 정확히 반올림 것으로 가정한다. 그러면 *(m x) (m x x)*는 *p-k* 유효 자릿수로 반올림 된 x와 정확히 같습니다. 보다 정확하게는 x의 유효 숫자를 취하고 k 개의 최하위 자릿수 바로 왼쪽에있는 기수 점을 상상하고 정수로 반올림하여 x를 반올림합니다.ββ⊗⊖ ⊗⊖

정리 6의 증명

정리 (14)에 의해, *(X)*의 $_{H는}$ 이고 , X는 반올림 P K = - 장소. 수행이 없으면 확실히 x_h 는 유효 숫자 로 나타낼 수 있습니다 . 수행이 있다고 가정하십시오. 만약 $X = X_0 . x_1 \ldots x_{p-1} \times e$ 이면 반올림하여 $x_{p-\ k\ -1}$에 1을 더하고 , 수행 할 수있는 유일한 방법은 $x_{p-\ k\ -1}$ = -1 인 경우입니다. x_h 의 하위 자릿수 ⌊$p/2$⌋⌊$p/2$⌋ ββ 는 1 + $x_{p-\ k\ -1}$ = 0이므로 다시 x_h 는 ⌊$p/2$⌋ 숫자 로 나타낼 수 있습니다.

다루는 X의 $_L$ 스케일은 X 충족시키는 정수로 $P^{1-} X^P$ - 1하자 는 IS P - K 의 높은 차수의 X를 , 그리고 는 IS K의 낮은 순서의 숫자. 고려해야 할 세 가지 경우가 있습니다. 이면 x 를 p - k 자리 로 반올림 하는 것은 자르기 및 , 및 . 최대 k 자릿수를 가지 므로 p가 짝수이면 최대 k = = 자릿수입니다. 그렇지 않으면 = 2 및β ≤ ≤ $\beta^{x = \bar{x}_h + \bar{x}_l \bar{x}_h \bar{x}_l \bar{x}_l < (\beta/2)\beta^{k-1}}$ $x_h = \bar{x}_h\ x_l = \bar{x}_l \bar{x}_l \bar{x}_l$ ⌈$p/2$⌉⌊$p/2$⌋ $\beta^{\bar{x}_1 < 2^{k-1}}$ k -1 개의 중요한 비트로 표현 가능 경우에 두 번째 경

우는 다음 컴퓨팅 및 엑스 H하는 반올림 포함하므로 $X_H = +^K$ 및 $X_L = X - X_H = X - -^K = -^K$. 다시 한 번, 최대 *k* 자리를 가지므로 p / 2 자리로 표현할 수 있습니다. 마지막으로 = (/ 2) $^{k-1}$ 이면 x_h = 또는≤ ⌊p/2⌋ x>(β/2)β^{k-1} $\bar{x}_h$β$\bar{x}_h$ β$\bar{x}_l$β$\bar{x}_l$⌊ ⌋ $\bar{x}_l$ββ$\bar{x}_h$$\bar{x}_h$ + ^{k} 는 반올림 여부에 따라 다릅니다. 따라서 x_l 은 (/ 2) $^{k-1}$ 또는 (/ 2) $^{k-1}$ - ^{k} = $^{-k}$ / 2 이며 둘 다 1 자리로 표시됩니다. 지ββββββββ

정리 6은 두 작업 정밀도 수의 곱을 합계로 정확하게 표현하는 방법을 제공합니다. 합계를 정확하게 표현하기위한 동반 공식이 있습니다. 만약 | *x* | ≥| *y* | 그런 다음 *x* + *y* = (*x y*) + (*x* (*x y*)) *y* [Dekker 1971; Knuth 1981, 섹션 4.2.2의 정리 C]. 그러나 정확하게 반올림 된 연산을 사용하는 경우이 수식은 *x* = .99998, *y* = .99997이 보여 주듯이 = 10에 대한 것이 아니라 = 2에 대해서만 참입니다 .⊕ ⊖ ⊕⊕ ββ

2 진수에서 10 진수로 변환

단 정밀도는 *p* = 24이고 2^{24} <10^8 이므로 이진수를 10 진수 8 자리로 변환하면 원래 이진수를 복구하는 데 충분할 것으로 예상 할 수 있습니다. 그러나 이것은 사실이 아닙니다.

정리 15

이진 *IEEE* 단 정밀도 숫자가 가장 가까운 *8* 자리 *10* 진수로 변환 될 때 항상 *10* 진수 *1*에서 이진 숫자를 고유하게 복구 할 수있는 것은 아닙니다. 그러나 *9* 개의 *10* 진수를 사용하는 경우 *10* 진수를 가장 가까운 *2* 진수로 변환하면 원래 부동 소수점 숫자가 복구됩니다.

증명

반 개방 구간 [10^3 , 2^{10}) = [1000, 1024)에있는 이진 단 정밀도 숫자 는 이진 점의 왼쪽에 10 비트, 이진 점의 오른쪽에 14 비트가 있습니다. 따라서, (2 거기에 ^{10} - $10^{(3)}$) $(2)^{(14)}$ 이 간격 = 393,216 다른 이진 숫자. 십진수는 8 자리로 표현되는 경우, (2 거기에 ^{10} - $10^{(3)}$) $(10)^{(4)}$ 와 동일한 간격 = 240,000 진수 번호. 240,000 개의 십진수가 393,216 개의 다른 이진수를 나타낼 수있는 방법은 없습니다. 따라서 10 진수 8 자리는 각 단 정밀도 이진수를 고유하게 표현하기에 충분하지 않습니다.
9 자리로 충분하다는 것을 보여주기 위해 이진수 사이의 간격이 항상 10 진수 사이의 간격보다 크다는 것을 보여 주면 충분합니다. 이렇게하면 각 십진수 *N* 에 대해 간격이

[*N* - $\frac{1}{2}$ ulp, *N* + $\frac{1}{2}$ ulp]

최대 하나의 이진수를 포함합니다. 따라서 각 이진수는 고유 한 10 진수로 반올림되고 차례로 고유 한 이진수로 반올림됩니다.
이진수 사이의 간격이 항상 10 진수 사이의 간격보다 크다는 것을 나타내려면 간격 [10^n , 10^{n+1}]을 고려하십시오. 이 간격에서 연속 십진수 사이의 간격은 $10^{(n+1)-9}$ 입니다. [10^n , 2^m]에서, 여기서 *m* 은 가장 작은 정수이므로 10^n <2^m , 이진수의 간격은 2^{m-24} 이고 간격은 간격에서 더 커집니다. 따라서 $10^{(n+1)-9}$ <2^{m-24} 를 확인하는

것으로 충분합니다 . 하지만 사실, 10 년 이후$n < 2m$, $10^{(n+1)-9} = 10n\ 10^{-8} < 2m\ 10^{-8} < 2m\ 2^{-24}$. 지

배정 밀도에 적용된 동일한 인수는 배정 밀도 숫자를 복구하는 데 십진수 17 자리가 필요함을 보여줍니다.

2 진 10 진수 변환은 플래그 사용의 또 다른 예를 제공합니다. Precision 섹션 에서 10 진수 확장에서 2 진수를 복구하려면 10 진수에서 2 진수로의 변환을 정확하게 계산해야한다는 사실을 상기하십시오. 즉 변환 수량 승산함으로써 행한다 N 과 $10^{|P|}$ (p <13 인 경우 모두 정확함) 단일 확장 정밀도로 반올림 한 다음이를 단 정밀도로 반올림합니다 (또는 p <0 인 경우 나누기 ; 두 경우 모두 유사 함). 물론 계산 $N \cdot 10^{|P|}$ 정확할 수 없습니다. 조합 된 연산 라운드 ($N \cdot 10^{|P|}$) 정확해야합니다. 반올림은 단 정밀도에서 단 정밀도로 반올림됩니다. 정확하지 않을 수있는 이유를 알아 보려면 간단한 경우 $\beta = 10$, 단일의 경우 $p = 2$, 단일 확장의 경우 $p = 3$을 사용하십시오. 곱이 12.51이면 단일 확장 곱하기 연산의 일부로 12.5로 반올림됩니다. 단 정밀도로 반올림하면 12가됩니다. 그러나 제품을 단 정밀도로 반올림하면 13이 제공되므로 그 대답은 정확하지 않습니다. 오류는 이중 반올림 때문입니다.

IEEE 플래그를 사용하면 다음과 같이 이중 반올림을 피할 수 있습니다. 정확하지 않은 플래그의 현재 값을 저장 한 다음 재설정합니다. 반올림 모드를 0으로 반올림으로 설정하십시오. 그런 다음 곱셈을 수행합니다. $N \cdot 10^{|P|}$. 정확하지 않은 플래그의 새 값을에 저장하고 ixflag반올림 모드와 부정확 한 플래그를 복원합니다. 경우 ixflag0 인 $N \cdot 10^{|P|}$ 정확하므로 round ($N \cdot 10^{|P|}$)는 마지막 비트까지 정확합니다. 경우 ixflag1 항상 자릅니다을 라운드로가 제로부터 다음 몇 가지 숫자, 절단되었다. 제품의 의미는 다음과 같습니다. $1.b_1 \ldots b_{22}\ b_{23} \ldots b_{31}$. 이중 라운딩 에러가 발생하는 경우 $(B)_{(23)} \ldots b$를 $_{(31)} = 10 \ldots$ 두 가지 경우를 고려하여 0 간단한 방법 논리 수행하는OR중을ixflag가진 $(B)_{31}$. 그러면 round ($N \cdot 10^{|P|}$)는 모든 경우에 올바르게 계산됩니다.

요약 오류

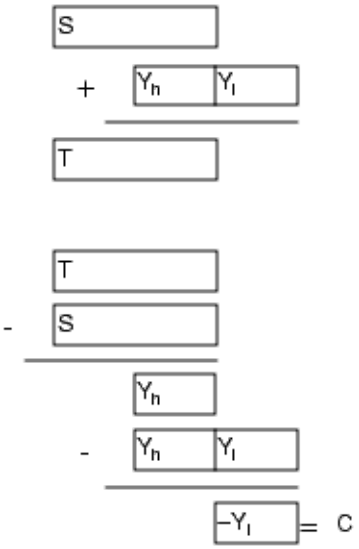
Optimizers 섹션에서는 매우 긴 합계를 정확하게 계산하는 문제를 언급했습니다. 정확도를 높이는 가장 간단한 방법은 정확도를 두 배로 높이는 것입니다. 정밀도를 두 배로 늘리면 합계의 정확도가 얼마나 향상되는지 대략적인 추정치를 얻으려면 $s_1 = x_1$, $s_2 = s_1\ x_2 \ldots$, $s_i = s_i$ -1 x_i라고 합니다. 그러면 $s_i = (1 + \ _i) (s_{i-1} + x_i)$, 여기서 $_i \oplus \oplus \delta|\delta| \le \varepsilon$, $_i$ 에서 2 차 항을 무시 하면δ

$$s_n = \sum_{j=1}^{n} x_j \left(1 + \sum_{k=j}^{n} \delta_k\right) = \sum_{j=1}^{n} x_j + \sum_{j=1}^{n} x_j \left(\sum_{k=j}^{n} \delta_k\right) \tag{31}$$

(31) 의 첫 번째 등식은 의 계산 된 Σx_j 값이 x_j의 교란 된 값에 대해 정확한 합이 수행 된 것과 동일 함을 보여줍니다 . 첫 번째 항 x_1 은 $n\varepsilon$ 에 의해 교란되고 마지막 항 x_n 은 ε. (31) 의 두 번째 등식은 오류 항이 $n\varepsilon\Sigma|x_j|$. 정밀도를 두 배로 늘리면 제곱 효과가 ε있습니다. 합계가 IEEE 배정 밀도 형식으로 수행되는 경우 $1 / 10^{16}$ 이므로 적절한 n 값이 됩니다. 따라서 정밀도를 두 배로 늘리면 n 의 최대 섭동이 필요합니다.$\varepsilon \approx n\varepsilon \ll 1\ \varepsilon$로 변경합니

다 $n\varepsilon^2 \ll \varepsilon$. 따라서 εKahan 합산 공식 (정리 8)에 대한 2 오류 한계는 단 정밀도보다 훨씬 낫지 만 배정 밀도를 사용하는 것만 큼 좋지 않습니다.

Kahan 합계 공식이 작동하는 이유에 대한 직관적 인 설명은 다음 절차 다이어그램을 고려하십시오.



summand가 추가 될 때마다 다음 루프에 적용될 수정 계수 *C* 가 있습니다. 따라서 먼저 이전 루프에서 계산 된 수정 *C* 를 X_j 에서 빼서 수정 된 합계 *Y*를 제공합니다 . 그런 다음 이 합계를 누적 합계 *S*에 추가합니다 . *Y* 의 하위 비트 (즉, Y_l)는 합계에서 손실됩니다. 다음의 상위 비트 계산 *Y*를 계산하여 *T*를 - *S*를 . *Y*는 이 감산되면, 하위 비트 *Y*복구됩니다. 이들은 다이어그램의 첫 번째 합계에서 손실 된 비트입니다. 이들은 다음 루프의 보정 요소가됩니다. Knuth [1981] 572 쪽에서 가져온 정리 8의 공식 증명은 정리 14 및 정리 8 섹션에 나와 있습니다. "

요약

컴퓨터 시스템 설계자가 부동 소수점과 관련된 시스템 부분을 무시하는 것은 드문 일이 아닙니다. 이것은 아마도 컴퓨터 과학 커리큘럼에서 부동 소수점이 거의 (있는 경우)주의를 기울이지 않기 때문일 것입니다. 이것은 부동 소수점이 정량화 할 수있는 주제가 아니라는 명백하게 널리 퍼진 믿음을 불러 일으켰으므로이를 처리하는 하드웨어 및 소프트웨어의 세부 사항에 대해 소란을 피울 필요가 거의 없습니다.

이 백서에서는 부동 소수점에 대해 엄격하게 추론 할 수 있음을 입증했습니다. 예를 들어, 기본 하드웨어에 보호 숫자가있는 경우 취소를 포함하는 부동 소수점 알고리즘은 상대적 오류가 작은 것으로 입증 될 수 있으며, 확장 정밀도가 다음과 같을 경우 반전 가능함이 입증 될 수있는 이진 10 진수 변환을위한 효율적인 알고리즘이 있습니다. 지원됩니다. 신뢰할 수있는 부동 소수점 소프트웨어를 구성하는 작업은 기본 컴퓨터 시스템이 부동 소수점을 지원할 때 훨씬 쉬워집니다. 방금 언급 한 두 가지 예 (가드 디지트 및 확장 된 정밀도) 외에도이 백서의 시스템 측면 섹션 에는 부동 소수점을 더 잘 지원하는 방법을 보여주는 명령어 세트 설계에서 컴파일러 최적화에 이르는 다양한 예가 있습니다.

IEEE 부동 소수점 표준에 대한 수용이 증가함에 따라 표준의 기능을 활용하는 코드가 더욱 이식 가능 해지고 있습니다. IEEE 표준 섹션에서는 IEEE 표준 의 기능을 실제 부동 소수점 코드 작성에 사용하는 방법을 보여주는 수많은 예를 제공했습니다.

감사의 말

이 기사는 1988 년 5 월부터 7 월까지 Sun Microsystems에서 W. Kahan이 제공 한 과정에서 영감을 얻었으며 Sun의 David Hough가 매우 잘 조직했습니다. 제 희망은 다른 사람들이 오전 8시 강의에 참석하기 위해 제 시간에 일어나지 않고도 부동 소수점과 컴퓨터 시스템의 상호 작용에 대해 배울 수 있도록하는 것입니다. Kahan과 Xerox PARC의 많은 동료 (특히 John Gilbert)가이 문서의 초안을 읽고 유용한 의견을 많이 제공해 주신 데 대해 감사드립니다. Paul Hilfinger와 익명의 심판의 리뷰도 프레젠테이션을 개선하는 데 도움이되었습니다.

참고 문헌

Aho, Alfred V., Sethi, R. 및 Ullman JD 1986. 컴파일러 *: Principles, Techniques and Tools* , Addison-Wesley, Reading, MA.

ANSI 1978. 미국 국가 표준 프로그래밍 언어 *FORTRAN* , ANSI 표준 X3.9-1978, 미국 국립 표준 협회, 뉴욕, 뉴욕.

Barnett, David 1987. *A Portable Floating-Point Environment* , 미발표 원고.

Brown, WS 1981. 부동 소수점 계산의 단순하지만 현실적인 모델 , ACM Trans. 수학. 소프트웨어 7 (4), 445-480 쪽.

Cody, W. J et. al. 1984. 부동 소수점 산술을위한 기수 및 단어 길이 독립적 인 표준 제안 , IEEE Micro 4 (4), pp. 86-100.

Cody, WJ 1988. 부동 소수점 표준-이론 및 실습 , "계산의 신뢰성 : 과학 컴퓨팅에서 간격 방법의 역할", ed. 저자 : Ramon E. Moore, pp. 99-107, Academic Press, Boston, MA.

Coonen, Jerome 1984. 이진 부동 소수점 산술을위한 제안 된 표준에 대한 공헌 , PhD 논문, Univ. 캘리포니아 버클리.

Dekker, TJ 1971. 사용 가능한 정밀도 확장을위한 부동 소수점 기법 , 숫자. 수학. 18 (3), pp. 224-242.

Demmel, James 1984. 수치 소프트웨어의 *Underflow* 및 신뢰성 , SIAM J. Sci. 합계. 계산. 5 (4), 887-919 쪽.

Farnum, Charles 1988. 부동 소수점 계산을위한 컴파일러 지원 , 소프트웨어 실습 및 경험, 18 (7), pp. 701-709.

Forsythe, GE and Moler, CB 1967. 선형 대수 시스템의 컴퓨터 솔루션 , Prentice-Hall, Englewood Cliffs, NJ.

Goldberg, I. Bennett 1967. *27* 비트는 *8* 자리 정확도에 충분하지 않음 , Comm. ACM 의. 10 (2), 105-106 쪽.

Goldberg, David 1990. *Computer Arithmetic* , in "Computer Architecture : A Quantitative Approach", by David Patterson 및 John L. Hennessy, Appendix A, Morgan Kaufmann, Los Altos, CA.

Golub, Gene H. and Van Loan, Charles F. 1989. *Matrix Computations* , 2nd edition, The Johns Hopkins University Press, Baltimore Maryland.

Graham, Ronald L., Knuth, Donald E. 및 Patashnik, Oren. 1989. *Concrete Mathematics,* Addison-Wesley, Reading, MA, p.162.

Hewlett Packard 1982. *HP-15C* 고급 기능 핸드북 .

IEEE 1987. 이진 부동 소수점 연산을위한 *IEEE* 표준 *754-1985* , IEEE, (1985). SIGPLAN 22 (2) pp. 9-25에 재 인쇄 됨.

Kahan, W. 1972. *A Survey Of Error Analysis* , in Information Processing 71, Vol 2, pp. 1214-1239 (Ljubljana, Yugoslavia), North Holland, Amsterdam.

Kahan, W. 1986. 바늘 모양의 삼각형의 면적과 각도 계산 , 미발표 원고.

Kahan, W. 1987. 복잡한 기본 기능에 대한 분기 컷 , "The State of the Art in Numerical Analysis", ed. MJD Powell 및 A. Iserles (영국 버밍엄 대학), 7 장, 뉴욕 옥스포드 대학 출판부.

Kahan, W. 1988. Sun Microsystems, Mountain View, CA에서 제공 한 미발표 강의.

Kahan, W. and Coonen, Jerome T. 1982. *The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments* , in "The Relationship Between Numerical Computation and Programming Languages", ed. 작성자 : JK Reid, pp. 103-115, North-Holland, Amsterdam.

Kahan, W. 및 LeBlanc, E. 1985. *IBM Acrith* 패키지의 *Anomalies* , Proc. 제 7 회 컴퓨터 산술에 관한 IEEE 심포지엄 (일리노이 주 어 바나), pp. 322-331.

Kernighan, Brian W. 및 Ritchie, Dennis M. 1978. *C* 프로그래밍 언어 , Prentice-Hall, Englewood Cliffs, NJ.

Kirchner, R. 및 Kulisch, U. 1987. 벡터 프로세서를위한 산술 , Proc. 제 8 회 IEEE 컴퓨터 산술 심포지엄 (이탈리아 코모), pp. 256-269.

Knuth, Donald E., 1981. *The Art of Computer Programming, Volume II* , Second Edition, Addison-Wesley, Reading, MA.

Kulisch, UW 및 Miranker, WL 1986. 디지털 컴퓨터의 산술 *:* 새로운 접근 방식 , SIAM Review 28 (1), pp 1-36.

Matula, DW 및 Kornerup, P. 1985. 유한 정밀도 유리 산술 *:* 슬래시 수 시스템 , IEEE Trans. Comput에서. C-34 (1), 3-18 쪽.

Nelson, G. 1991. *System Programming With Modula-3* , Prentice-Hall, Englewood Cliffs, NJ.

Reiser, John F. and Knuth, Donald E. 1975. 부동 소수점 덧셈의 드리프트 회피 , 정보 처리 편지 3 (3), pp 84-87.

Sterbenz, Pat H. 1974. 부동 소수점 계산 , Prentice-Hall, Englewood Cliffs, NJ.

Swartzlander, Earl E. 및 Alexopoulos, Aristides G. 1975. *The Sign / Logarithm Number System* , IEEE Trans. 계산. C-24 (12), 1238-1242 쪽.

Walther, JS, 1971. 기본 기능을위한 통합 알고리즘 , AFIP Spring Joint Computer Conf의 회보. 38, 379-385 쪽.

정리 14와 정리 8

이 섹션에는 텍스트에서 생략 된 두 가지 기술적 증명이 포함되어 있습니다.

정리 14

하자 0 <*K* <*p* 및 집합 *m* = ^{K} + 1 , 및 부동 소수점 연산이 정확히 반올림 것으로 가정한다. 그런 다음 (*m x*) (*m x x*) 는 *p-k* 유효 자릿수로 반올림 된 *x*와 정확히 같습니다. 보다 정확하게는 *x*의 유효 숫자를 취하고 *k* 개의 최하위 자릿수 바로 왼쪽에있는 기수 점을 상상하고 정수로 반올림하여 *x*를 반올림합니다.

증명

증명은 *mx* = ^{k} *x* + *x* 계산에 캐리 아웃이 있는지 여부에 따라 두 가지 경우로 나뉩니다 . 수행이 없다고 가정합니다. *x* 가 정수가되도록 스케일하는 것은 무해합니다 . 그러면 *mx* = *x* + ^{k} *x* 의 계산은 다음과 같습니다.

```
aa...aabb...bb
+ aa ... aabb ... bb
zz...zzbb...bb
```

여기서 *x* 는 두 부분으로 분할되었습니다. 하위 *k* 자리가 표시 b되고 상위 *p* - *k* 자리가 표시 a됩니다. *mx* 에서 *m x* 를 계산하려면 하위 *k* 자릿수 (으로 표시된 숫자)를 반올림 하여 b

(32) *m x* = *mx* - *x* mod (^{k}) + r ^{k}

r 의 값 .bb...b은보다 크면 1이고 그렇지 않으면 0입니다. 더 정확하게

(33) *R* = 1의 경우 a.bb...b에 발사 + 1, *R* = 0 그렇지에게.

다음으로 *m x* - *x* = *mx* - *x* mod (^{k}) + *r* ^{k} - *x* = ^{k} (*x* + *r*) -*x* mod (^{k})를 계산합니다. 아래 그림은 반올림 된 *m x* - *x* , 즉 (*m x*) *x* 의 계산을 보여줍니다 . 맨 윗줄은 ^{k} (*x* + *r*) 이며, 여기서은 최하위 숫자에 더한 결과 숫자 입니다. Brb

```
aa...aabb...bB00...00
- bb ... bb
zz... zzZ00...00
```

경우 .bb...b<는 다음 *r*에 = 0을 감산 표시된 숫자에서 차용 발생 B순 효과가 둥근 차이인 상단 라인 같다는 정도로하지만, 차이는 반올림하여, ^{K} *(X)* . 경우 > 다음 *r*에 = 1, 1부터 감산 결과가되므로 때문에 차용의 ^{K} *(X)* . 마지막으로 케이스 =를 고려하십시오 . 경우 *R* = 0 다음 짝수, 홀수 및 차이주는 반올림 ^{K} *X* . 마찬가지로 *r* = 1 이 홀수 일 때.bb...b B.bb...b BZBZ짝수이면 차이가 반올림되므로 다시 차이는 ^{k} *x* 입니다. 요약

(34) (*m x*) *x* = ^{k} *x*

방정식 (32) 와 (34)를 결합 하면 (*m x*)-(*m x x*) = *x* - *x* mod (^{k}) + · ^{k가} 됩니다. 이 계산을 수행 한 결과는 다음과 같습니다.

```
r00...00
+ aa...aabb...bb
- bb ... bb
aa...aA00...00
```

연산 규칙 *R* , 식 (33), 반올림 규칙과 동일하다 에 *P* - *K*의 장소. 따라서 컴퓨팅 *MX* - (*MX* - *X*) 부동 소수점 연산 정밀도로 라운딩 정확히 동일하다 *X*가 에 *P* - *K*의 경우, 위치 *(X)* + ^{K를} *X*는 수행하지 않는다.a... ab...b
시 *X* + ^{K} *X* 수행 않습니다, 다음 *MX* = ^{K} *X* + *X* 다음과 같다 :

```
  aa...aabb...bb
+ aa ... aabb ... bb
  zz...zZbb...bb
```

따라서, *m* *X* = *MX* - *X* 개조 (^{K}) + *w* $^{(K)}$ 여기서, *w* = - *Z* 경우 *Z* < / 2하지만 정확한 값 *w*는 중요하지 않다. 다음으로, *m* *x* - *x* = ^{k} *x* - *x* mod (^{k}) + *w* ^{k} . 사진에서⊗ ββ⊗ ββ

```
  aa...aabb...bb00...00
- bb... bb
+ w
  zz ... zZbb ...bb[31]
```

반올림은 (*m* *x*) *x* = ^{k} *x* + *w* ^{k} - *r* ^{k}를 제공합니다 . 여기서 *r* = 1 if > 또는 if = 및 *b* $_{0}$ = 1입니다. [32] 마지막으로,⊗ ⊖ ββ.bb...b$^{\frac{1}{2}}$.bb...b$^{\frac{1}{2}}$

(*m* *x*)-(*m* *x* x) = *mx* - *x* mod (^{k}) + *w* $^{k-}$ (^{k} *x* + *w* ^{k} - *r* ^{k}) = *x* - *x* mod (^{k}) + *r* ^{k} .⊗ ⊗ ⊖ββββββ ββ

그리고 다시 한 번, *p* - *k* 자리로 반올림 할 때 반올림이 포함 되는 경우 *r* = 1 입니다. 따라서 정리 14는 모든 경우에 입증되었습니다. 지a...ab...b

정리 8 (카한 합산 공식)

$\Sigma_{j=1}^{N} x_j$ 다음 알고리즘을 사용하여 계산 한다고 가정하십시오.

```
S = X [1];
C = 0;
j = 2에서 N {
Y = X [j]-C;
   T = S + Y;
   C = (T-S)-Y;
   S = T;
}
```

그러면 계산 된 합 *S* 는 *S* = *x* $_{j}$ (1 + $_{j}$) + *O* (*N* ^{2}) | *x* $_{j}$ |, 여기서 | $_{j}$ | 2 .Σ δεΣ δ≤ ε

증명

먼저 간단한 공식 *x* $_{i에}$ 대한 오류 추정치가 어떻게되었는지 기억하십시오 . 도입 *S* $_{1}$ = *(X)* $_{(1)}$, *s*의 $_{I는}$ (+ 1 = *I*) (*s*의 $_{전 - 1}$ + *X* $_{I}$). 그런 다음 계산 된 합은 *s* $_{n}$ 이며, 이는 용어의 합이며, 각각은 *x* $_{i에 j가}$ 포함 된 표현식을 곱한 것 입니다. *x* $_{1}$ 의 정확한 계수 는 (1 + $_{2}$) (1 + $_{3}$) ... (1 + $_{n}$)이므로 번호를 다시 매김으로써Σ δδ δδδ*x* $_{2}$ 는 (1 + $_{3}$) (1 + $_{4}$) ... (1 + $_{n}$) 등 이어야합니다 . 정리 8의 증명은 정확히 동일한 선을 따라 실행되며 *x* $_{1}$ 의 계수 만 더 복잡합니다. 자세히 *s* $_{0}$ = *c* $_{0}$ = 0 및δ δδ

y $_{k}$ = *x* $_{k}$ *c* $_{k-1}$ = (*x* $_{k}$ -*c* $_{k-1}$) (1 + $_{k}$) ⊖ η

$s_k = s_{k-1} y_k = (s_{k-1} + y_k) (1 + k) \oplus\approx \sigma$
$c_k = (s_k s_{k-1}) y_k = [(s_k - s_{k-1}) (1 + k) - y_k] (1 + k) \ominus \ominus \gamma\delta$

모든 그리스 문자는 ε. 계수 있지만 X_1 에서 의 k는 계수 계산하기 쉬운 것으로 판명 관심 궁극적 인 표현이며, X_1 에서 의 (K) - (C)의 K 및 C의 케이 .
$k = 1$ 일 때 ,

$c_1 = (s_1 (1 + 1) - y_1) (1 + d_1)\gamma$
$= y_1 ((1 + s_1) (1 + 1)-1) (1 + d_1)\gamma$
$= X_{(1)} (S_{(1)} + 1 + S_1 g_1) (1 + D_{(1)}) (1 + H_1)\gamma$
s의 $_1$ - $C_{(1)} = X_{(1)}$ [(1 + s의 $_1$) - ($S_{(1)} + g_{(1)}$ + (S) $_1 g_1$) (1 + $D_{(1)}$) (1 + H_1)
$= X_{(1)}$ [- 1 $g_{(1)}$ - (S) (1 개) (D) $_{(1)}$ - (S) $_1 g_{(1 개)}$ - (D) $_1 g_{(1)}$ - (S) $_1 g_{(1)}$ (D) $_{(1)}$ (1 + H_1)

계수 호출 X_1 이 식의 C의 K 및 S를 (k)를 다음 각각

$c_1 = 2\varepsilon + O (^2)\varepsilon$
$s_{(1)} = + _1 - _1 + 4 ^{(2)} + O (^3) \eta\gamma\varepsilon\varepsilon$

S_k 및 C_k에 대한 일반 공식을 얻으려면 s_k 및 c_k 의 정의를 확장하고 $i > 1$ 인 x_i 와 관련된 모든 항을 무시하고

$s_k = (s_{k-1} + y_k) (1 + k)\sigma$
$= [s_{k-1} + (x_k - c_{k-1}) (1 + k)] (1 + k)\eta\sigma$
$= [(s_{k-1} - c_{k-1}) -k c_{k-1}] (1+ k)\eta\sigma$
$c_k = [\{ s_k - s_{k-1} \} (1 + k) - y_k] (1 + k)\gamma\delta$
$= [\{((s_{k-1} - c_{k-1}) -k c_{k-1}) (1 + k) - s_{k-1} \} (1 + k) + c_{k-1} (1 + k)] (1 + k)\eta\sigma\gamma\eta\delta$
$= [\{(s_{k-1} - c_{k-1}) k - k c_{k-1} (1 + k) - c_{k-1} \} (1 + k) + c_{k-1} (1 + k)] (1 + k)\sigma\eta \sigma\gamma \eta\delta$
$= [(s_{k-1} - c_{k-1}) k (1 + k) - c_{k-1} (k + k (k + k + k k))] (1 + k),\sigma\gamma\gamma\eta\sigma\gamma\sigma\gamma\delta$
$s_k - c_k = ((s_{k-1} - c_{k-1}) -k c_{k-1}) (1 + k)\eta\sigma$
$-[(s_{k-1} - c_{k-1}) k (1 + k) - c_{k-1} (k + k (k + k + k k)] (1 + k)\sigma\gamma\gamma\eta\sigma\gamma\sigma\gamma\delta$
$= (s_{k-1} - c_{k-1}) ((1 + k) -k (1 + k) (1 + k))\sigma\sigma\gamma\delta$
$+ c_{k-1} (-k (1 + k) + (k + k (k + k + k k)) (1 + k))\eta\sigma\gamma\eta\sigma\gamma\sigma\gamma\delta$
$= (s_{-1} - c_{k-1}) (1- k (k + k + k k))\sigma\gamma\delta\gamma\delta$
+ C의 $_{K - 1}$ - [$\eta K + K + K (K + K K) + (K + K (K + K + K K)) K$]$\gamma\eta\gamma\sigma\gamma\gamma\eta\sigma\gamma\sigma\gamma\delta$

S_k 와 C_k 는 ^{2} 차까지만 계산 되기 때문에 이러한 공식은 다음과 같이 단순화 할 수 있습니다.ε

$C_k = (k + O(^2)) S_{k-1} + (-k + O(^2)) C_{k-1}$ σεγε

$S_k = ((1 + 2^2 + O(^3)) S_{k-1} + (2+(^2)) C_{k-1}$ εεεOε

이 공식을 사용하면

$C_2 = {}_2 + O(^2)$ σε

$S_{(2)} = 1 + {}_1 - {}_1 + 10^2 + O$를 (^{3}) ηγεε

일반적으로 귀납법으로 쉽게 확인할 수 있습니다.

C에서 $_K = K + O(^2)$ σε

$S_K = 1 + {}_1 - {}_1 + (4K+2)^2 + O(^3)$ ηγεε

마지막으로, 어떤 것은 수배의 계수 X_1 에서 의 $_k$는 . 이 값을 얻으려면,하자 $X_{N+1} = 0$ 의 첨자로 모든 그리스 문자를 보자 $N + 1$과 동일 0, 및 컴퓨팅 의 $_{N+1}$. 이어서 s의 $_{N+1} = (S)_N - C_{,N}$, 및 계수 $(X)_{(1)}$ 에서 의 $_N$ 의 계수보다 작은 S_{N+1} 이고, $S_{,N} = 1 + {}_1 - {}_1 + (4N + 2)$ $^{(2)}$ηγε= (1 + 2 ε+ O(n^2)). 지ε

IEEE 754 구현 간의 차이점

주 – 이 섹션은 출판 된 논문의 일부가 아닙니다. 독자가 논문에서 추론 할 수있는 IEEE 표준에 대한 특정 요점을 명확히하고 가능한 오해를 수정하기 위해 추가되었습니다. 이 자료는 David Goldberg가 작성한 것이 아니지만 그의 허락하에 여기에 나타납니다.

앞의 논문은 프로그래머가 프로그램의 정확성과 정확성을 위해 속성에 의존 할 수 있기 때문에 부동 소수점 산술은 신중하게 구현되어야 함을 보여주었습니다. 특히 IEEE 표준은 세심한 구현이 필요하며, 표준을 준수하는 시스템에서만 올바르게 작동하고 정확한 결과를 제공하는 유용한 프로그램을 작성할 수 있습니다. 독자는 그러한 프로그램이 모든 IEEE 시스템에 이식 가능해야한다고 결론 내릴 수 있습니다. 실제로 휴대용 소프트웨어는 "프로그램이 두 시스템간에 이동하고 둘 다 IEEE 산술을 지원할 때 중간 결과가 다를 경우 산술의 차이가 아니라 소프트웨어 버그 때문일 것입니다."라는 말이 있으면 작성하기가 더 쉬울 것입니다. 진실.

불행히도 IEEE 표준은 동일한 프로그램이 모든 준수 시스템에서 동일한 결과를 제공한다고 보장하지 않습니다. 대부분의 프로그램은 실제로 다양한 이유로 다른 시스템에서 다른 결과를 생성합니다. 첫째, 대부분의 프로그램은 십진수와 이진 형식 사이의 숫자 변환을 포함하며 IEEE 표준은 이러한 변환이 수행되어야하는 정확도를 완전히 지정하지 않습니다. 또 다른 경우에는 많은 프로그램이 시스템 라이브러리에서 제공하는 기본 함수를 사용하며 표준은 이러한 함수를 전혀 지정하지 않습니다. 물론 대부분의 프로그래머는 이러한 기능이 IEEE 표준의 범위를 벗어난다는 것을 알고 있습니다.

많은 프로그래머는 IEEE 표준에서 규정 한 숫자 형식과 연산 만 사용하는 프로그램도 다른 시스템에서 다른 결과를 계산할 수 있다는 것을 인식하지 못할 수 있습니다. 사실, 표준의 작성자는 다른 구현이 다른 결과를 얻을 수 있도록 허용하려고했습니다. 그들의 의도는 목적지 라는 용어의 정의에서 분명 합니다.IEEE 754 표준 : "대상은 사용자에 의해 명시 적으로 지정되거나 시스템에 의해 암시 적으로 제공 될 수 있습니다 (예 : 하위 표현식의 중간 결과 또는 프로 시저에 대한 인수). 일부 언어는 중간 계산 결과를 사용자의 대상을 넘어

선 대상에 배치합니다. 그럼에도 불구하고이 표준은 해당 대상의 형식과 피연산자 값의 관점에서 연산의 결과를 정의합니다. " (IEEE 754-1985, p. 7) 즉, IEEE 표준에서는 각 결과가 배치 될 대상의 정밀도로 올바르게 반올림되어야하지만 표준에서는 해당 대상의 정밀도가 사용자의 프로그램에 의해 결정됩니다. 따라서 서로 다른 시스템은 서로 다른 정밀도로 목적지에 결과를 전달할 수 있습니다.

이전 백서의 몇 가지 예는 부동 소수점 산술이 반올림되는 방식에 대한 지식에 따라 달라집니다. 이와 같은 예에 의존하기 위해 프로그래머는 프로그램이 해석되는 방식, 특히 IEEE 시스템에서 각 산술 연산의 대상 정밀도가 얼마인지 예측할 수 있어야합니다. 아아, IEEE 표준의 목적지 정의의 허점프로그램을 해석하는 방법을 아는 프로그래머의 능력을 약화시킵니다. 결과적으로 위에서 제시된 몇 가지 예는 고수준 언어로 명백하게 이식 가능한 프로그램으로 구현 될 때 프로그래머가 예상하는 것과 다른 정밀도로 대상에 결과를 일반적으로 전달하는 IEEE 시스템에서 제대로 작동하지 않을 수 있습니다. 다른 예제도 작동 할 수 있지만 작동한다는 것을 증명하는 것은 일반적인 프로그래머의 능력을 넘어 설 수 있습니다.

이 섹션에서는 일반적으로 사용하는 대상 형식의 정밀도에 따라 IEEE 754 산술의 기존 구현을 분류합니다. 그런 다음 논문의 몇 가지 예를 검토하여 프로그램이 예상하는 것보다 더 넓은 정밀도로 결과를 제공하면 예상 정밀도가 사용될 때 입증 가능하게 정확하더라도 잘못된 결과를 계산할 수 있음을 보여줍니다. 또한 우리 프로그램이 무효화되지 않는 경우에도 예상치 못한 정확성에 대처하는 데 필요한 지적 노력을 설명하기 위해 논문의 증명 중 하나를 다시 검토합니다. 이러한 예는 IEEE 표준이 규정하는 모든 것에도 불구하고 다른 구현간에 허용되는 차이로 인해 동작을 정확하게 예측할 수있는 이식 가능하고 효율적인 수치 소프트웨어를 작성하지 못할 수 있음을 보여줍니다. 그런 소프트웨어를 개발하려면

현재 IEEE 754 구현

IEEE 754 산술의 현재 구현은 하드웨어에서 서로 다른 부동 소수점 형식을 지원하는 정도에 따라 두 그룹으로 구분할 수 있습니다. 확장 기반Intel x86 프로세서 제품군으로 예시된 시스템은 확장 된 배정 밀도 형식에 대한 완전한 지원을 제공하지만 단 정밀도 및 배정밀도에 대한 부분적인 지원 만 제공합니다. 데이터를 단 정밀도 및 배정 밀도로로드하거나 저장하여 즉시 변환하는 명령을 제공합니다. 확장 double 형식으로 또는 확장 double 형식으로 이동하며, 산술 연산의 결과가 확장 double 형식으로 레지스터에 유지 되더라도 단 정밀도 또는 배정 밀도로 반올림되는 특수 모드 (기본값이 아님)를 제공합니다. (Motorola 68000 시리즈 프로세서는 이러한 모드에서 단일 또는 이중 형식의 정밀도와 범위 모두로 결과를 반올림합니다. Intel x86 및 호환 프로세서는 결과를 단일 또는 이중 형식의 정밀도로 반올림하지만 확장 이중 형식과 동일한 범위를 유지합니다.) 싱글 / 더블대부분의 RISC 프로세서를 포함한 시스템은 단 정밀도 및 배정 밀도 형식을 완벽하게 지원하지만 IEEE 호환 확장 배정 밀도 형식은 지원하지 않습니다. (IBM POWER 아키텍처는 단 정밀도에 대한 부분적인 지원 만 제공하지만이 섹션에서는 단일 / 이중 시스템으로 분류합니다.)

확장 기반 시스템에서 싱글 / 더블 시스템과 다르게 계산이 어떻게 다르게 작동하는지 확인하려면 시스템 측면 섹션의 C 버전 예제를 고려하십시오 .

```
int main () {
    이중 q;
    q = 3.0 / 7.0;
    if (q == 3.0 / 7.0) printf ( "같음 ₩ n");
    else printf ( "같지 않음 ₩ n");
    반환 0;
```

```
}
```

여기서 상수 3.0과 7.0은 배정 밀도 부동 소수점 숫자로 해석되며 3.0 / 7.0 표현식은 double데이터 유형을 상속합니다 . 단일 / 이중 시스템에서 표현식은 사용하기에 가장 효율적인 형식이므로 배정 밀도로 평가됩니다. 따라서 q배정 밀도로 올바르게 반올림 된 3.0 / 7.0 값이 할당됩니다. 다음 줄에서 3.0 / 7.0 표현식은 배정 밀도로 다시 평가되며 물론 결과는에 방금 할당 된 값과 같으 q므로 프로그램은 예상대로 "Equal"을 인쇄합니다.

확장 기반 시스템에서 3.0 / 7.0 표현식에 유형이 있더라도 double몫은 확장 배정 밀도 형식으로 레지스터에서 계산되므로 기본 모드에서는 확장 배정 밀도로 반올림됩니다. 그러나 결과 값이 변수에 할당되면 q메모리에 저장 될 수 있으며이 q선언 된 이후 double값은 배정 밀도로 반올림됩니다. 다음 줄에서 3.0 / 7.0 표현식은 확장 정밀도로 다시 평가되어에 저장된 배정 밀도 값과 다른 결과를 산출 q하여 프로그램이 "같지 않음"을 인쇄하게합니다. 물론 다른 결과도 가능합니다. 컴파일러는 3.0 / 7.0 표현식의 값을 두 번째 줄에 저장하고 반올림하여 비교하기 전에q, 또는 유지할 수 있습니다q저장하지 않고 확장 정밀도로 레지스터에. 최적화 컴파일러는 컴파일 시간에 아마도 배정 밀도 또는 확장 배정 밀도로 3.0 / 7.0 표현식을 평가할 수 있습니다. (하나의 x86 컴파일러를 사용하면 프로그램은 최적화로 컴파일 할 때 "Equal"을 인쇄하고 디버깅을 위해 컴파일 할 때 "Not Equal"을 인쇄합니다.) 마지막으로 확장 기반 시스템 용 일부 컴파일러는 레지스터에서 결과를 생성하는 연산을 수행하도록 자동으로 반올림 정밀도 모드를 변경합니다. 결과를 단 정밀도 또는 배정 밀도로 반올림하십시오. 따라서 이러한 시스템에서는 단순히 소스 코드를 읽고 IEEE 754 산술에 대한 기본적인 이해를 적용하는 것만으로는 프로그램의 동작을 예측할 수 없습니다. 하드웨어 나 컴파일러가 IEEE 754 호환 환경을 제공하지 못했다고 비난 할 수도 없습니다. 하드웨어는 필요에 따라 올바르게 반올림 된 결과를 각 대상에 전달했으며 컴파일러는 허용되는대로 사용자가 제어 할 수없는 대상에 일부 중간 결과를 할당했습니다.

확장 기반 시스템에서 계산의 함정

기존의 통념은 확장 기반 시스템이 싱글 / 더블 시스템에서 제공되는 것보다 정확하지는 않더라도 최소한 정확하지 않은 결과를 생성해야한다는 점을 유지합니다. 전자는 항상 최소한 그 이상의 정밀도를 제공하고 종종 후자보다 더 많은 것을 제공하기 때문입니다. 위의 C 프로그램과 같은 사소한 예와 아래에 설명 된 예를 기반으로 한 더 미묘한 프로그램은 이러한 지혜가 기껏해야 순진함을 보여줍니다. 실제로 단일 / 이중 시스템에서 이식 할 수있는 일부 이식 가능한 프로그램은 확장 프로그램에서 잘못된 결과를 제공합니다. 컴파일러와 하드웨어가 때때로 프로그램이 기대하는 것보다 더 많은 정밀도를 제공하기 위해 공모하기 때문에 기반 시스템.

현재 프로그래밍 언어는 프로그램이 예상하는 정밀도를 지정하기 어렵게 만듭니다. 언어 및 컴파일러 섹션에서 언급 했듯이 많은 프로그래밍 언어는 10.0*x동일한 컨텍스트에서와 같은 식의 각 발생 이 동일한 값으로 평가되어야한다고 지정하지 않습니다 . Ada와 같은 일부 언어는 IEEE 표준 이전에 서로 다른 산술 간의 차이에 의해 영향을 받았습니다. 최근에는 ANSI C와 같은 언어가 표준을 준수하는 확장 기반 시스템의 영향을 받았습니다. 실제로 ANSI C 표준은 컴파일러가 부동 소수점 식을 일반적으로 해당 형식과 관련된 것보다 더 넓은 정밀도로 평가할 수 있도록 명시 적으로 허용합니다. 결과적으로 표현식의 값은 10.0*x다양한 요인에 따라 달라질 수 있습니다.식이 변수에 즉시 할당되는지 또는 더 큰 식에서 하위 식으로 나타나는지 여부; 표현식이 비교에 참여하는지 여부; 표현식이 함수에 인수로 전달되는지 여부 및 인수가 값으로 전달되는지 참조로 전달되는지 여부; 현재 정밀도 모드; 프로그램이 컴파일 된 최적화 수준; 프로그램이 컴파일 될 때 컴파일러가 사용하는 정밀도 모드 및 표현식 평가 방법 등등.

언어 표준이 표현 평가의 모호함을 전적으로 비난하는 것은 아닙니다. 확장 기반 시스템은 가능할 때마다 확장 정밀도 레지스터에서 표현식을 평가할 때 가장 효율적으로 실행되지만 저장해야하는 값은 필요한 가장 좁은 정밀도로 저장됩니다. `10.0*x`모든 곳에서 동일한 값으로 평가되도록 언어를 제한하면 해당 시스템에 성능이 저하됩니다. 불행히도 이러한 시스템이 `10.0*x`구문 적으로 동등한 컨텍스트에서 다르게 평가되도록 허용하면 프로그래머가 의도 된 의미를 표현하기 위해 프로그램의 구문에 의존하는 것을 방지함으로써 프로그래머가 정확한 수치 소프트웨어를 사용할 수 없게됩니다.

실제 프로그램은 주어진 표현식이 항상 동일한 값으로 평가된다는 가정에 의존합니까? Fortran으로 작성된 ln (1 + *x*) 계산을 위해 정리 4에 제시된 알고리즘을 상기 하십시오.

```
실수 함수 log1p (x)
실제 x
(1.0 + x .eq. 1.0)이면
   log1p = x
그밖에
   log1p = log (1.0 + x) * x / ((1.0 + x)-1.0)
endif
반환
```

확장 기반 시스템에서 컴파일러는 `1.0 + x`확장 된 정밀도로 세 번째 줄 의 식 을 평가 하고 결과를 `1.0`. 그러나 동일한 표현식이 여섯 번째 줄의 로그 함수에 전달되면 컴파일러는 해당 값을 메모리에 저장하여 단 정밀도로 반올림 할 수 있습니다. 따라서, 경우는 `x`너무 작지 않다 `1.0 + x`순찰 `1.0`확장 된 정밀하지만 작은만큼에서 그 `1.0 + x`에게 라운드 `1.0`에서 단 정밀도에서 값을 반환 `log1p(x)`하는 대신 0이됩니다 `x`5보다는 더 큰 - 상대 오류가 하나가 될 것입니다 ε. 마찬가지로 하위 표현식의 재발을 포함하여 여섯 번째 행의 나머지 표현식 `1.0 + x`이 확장 정밀도로 평가 된다고 가정합니다 . 이 경우`x`은 작지만 단 정밀도로 `1.0 + x` 반올림 할만큼 충분히 작지 않은 `1.0`경우에서 반환 된 `log1p(x)`값이 올바른 값을 거의 초과할 수 `x`있으며 다시 상대 오류가 1에 접근 할 수 있습니다. 구체적인 예를 들어, `x`2^{-24} + 2^{-47} 이므로 `x`가장 작은 단 정밀도 숫자도 `1.0 + x`다음으로 큰 숫자 인 1 + 2^{-23}으로 반올림 됩니다 . 그러면 `log(1.0 + x)`대략 2^{-23} 입니다. 여섯 번째 줄에있는 표현식의 분모는 확장 정밀도로 평가되기 때문에 정확히 계산되어를 전달 `x`하므로 `log1p(x)`약 2^{-23}을 반환합니다., 이는 정확한 값의 거의 두 배입니다. (실제로 이는 하나 이상의 컴파일러에서 발생합니다. 이전 코드가 `-O`최적화 플래그를 사용하여 x86 시스템 용 Sun WorkShop Compilers 4.2.1 Fortran 77 컴파일러에 의해 컴파일 되면 생성 된 코드 `1.0 + x`가 설명 된대로 정확하게 계산 됩니다. 결과적으로 함수는 `log1p(1.0e-10)`및에 `1.19209E-07`대한 0. `log1p(5.97e-8)`)

Theorem 4의 알고리즘이 올바르게 작동하려면 표현식 `1.0 + x`이 나타날 때마다 동일한 방식으로 평가되어야합니다. 알고리즘은 `1.0 + x`한 인스턴스에서 확장 배정 밀도로 평가되고 다른 인스턴스에서는 단일 또는 배정 밀도로 평가 될 때만 확장 기반 시스템에서 실패 할 수 있습니다 . 물론 `log`은 Fortran의 일반 내장 함수이므로 컴파일러는 다음 표현식을 평가할 수 있습니다.`1.0 + x`전체적으로 확장 된 정밀도에서 동일한 정밀도로 로그를 계산하지만 컴파일러가 그렇게 할 것이라고 가정 할 수는 없습니다. (사용자 정의 함수를 포함하는 유사한 예제를 상상할 수도 있습니다.이 경우 컴파일러는 함수가 단 정밀도 결과를 반환하더라도 인수를 확장 정밀도로 유지할 수 있지만 기존 Fortran 컴파일러가이를 수행하는 경우는 거의 없습니다. .) 따라서 우리 `1.0 + x`는 변수에 할당하여 일관성있게 평가 되도록 시도할 수 있습니다 . 불행히도 해당 변수를 선언하면`real`, 변수의 한 모양에 대해 레지스터에

보관 된 값을 확장 정밀도로 대체하고 다른 변수에 대해 단일 정밀도로 메모리에 저장된 값을 대체하는 컴파일러에 의해 여전히 실패 할 수 있습니다. 대신 확장 정밀도 형식에 해당하는 유형으로 변수를 선언해야합니다. 표준 FORTRAN 77은이를 수행하는 방법을 제공하지 않으며 Fortran 95는 SELECTED_REAL_KIND다양한 형식을 설명 하는 메커니즘을 제공하지만 변수를 해당 정밀도로 선언 할 수 있도록 확장 된 정밀도로 표현식을 평가하는 구현을 명시 적으로 요구하지 않습니다. 요컨대, 1.0 + x증명을 무효화하는 방식으로 표현식 이 평가되는 것을 방지하는 표준 Fortran에서이 프로그램을 작성하는 이식 가능한 방법은 없습니다 .

각 하위 표현식이 저장되어 동일한 정밀도로 반올림되는 경우에도 확장 기반 시스템에서 오작동 할 수있는 다른 예가 있습니다. 원인은 이중 반올림입니다.. 기본 정밀도 모드에서 확장 기반 시스템은 처음에 각 결과를 확장 배정 밀도로 반올림합니다. 그 결과가 배정 밀도로 저장되면 다시 반올림됩니다. 이 두 반올림의 조합은 첫 번째 결과를 배정 밀도로 올바르게 반올림하여 얻은 값과 다른 값을 생성 할 수 있습니다. 이는 확장 배정 밀도로 반올림 된 결과가 "반의 경우"인 경우에 발생할 수 있습니다. 즉, 두 배정 밀도 숫자 사이의 정확히 중간에 있으므로 두 번째 반올림은 짝수 반올림 규칙에 의해 결정됩니다. 이 두 번째 반올림이 첫 번째와 같은 방향으로 반올림되면 순 반올림 오류는 마지막 자리에서 반 단위를 초과합니다. (그러나 이중 반올림은 배정 밀도 계산에만 영향을 미칩니다. 합, 차이,*p*는 숫자 비트 또는 제곱근 *(P)*의 비트 수는, 먼저 반올림 의 *Q* 비트 다음에 *P*의 결과가 단지 한번 둥근 것처럼 비트와 동일한 값을 제공 *P*의 비트는 제공된 $Q \geq 2P$ 따라서 + 2 확장 배정 밀도는 단 정밀도 계산이 이중 반올림을 겪지 않을만큼 충분히 넓습니다.)

올바른 반올림에 의존하는 일부 알고리즘은 이중 반올림으로 실패 할 수 있습니다. 실제로 올바른 반올림이 필요하지 않고 IEEE 754를 준수하지 않는 다양한 시스템에서 올바르게 작동하는 일부 알고리즘조차도 이중 반올림으로 실패 할 수 있습니다. 이들 중 가장 유용한 것은 Exactly Rounded Operations 섹션에서 언급 한 시뮬레이션 된 다중 정밀도 산술을 수행하기위한 이식 가능한 알고리즘입니다 . 예를 들어, 부동 소수점 숫자를 높은 부분과 낮은 부분으로 분할하는 정리 6에 설명 된 절차는 이중 반올림 산술에서 올바르게 작동하지 않습니다. 배정 밀도 숫자 $2^{52} + 3 \times 2^{26}$ 을 분할 해보십시오.-각각 최대 26 비트로 1을 두 부분으로 나눕니다. 각 연산이 배정 밀도로 올바르게 반올림되면 상위 부분은 $2^{52} + 2^{27}$ 이고 하위 부분은 2^{26-1} 이지만 각 연산을 먼저 확장 배정 밀도로 반올림 한 다음 배정 밀도로 반올림하면 절차가 진행됩니다. $2^{52} + 2^{28}$의 고차 부분과 -2^{26} 의 하위 부분을 생성합니다.-1. 후자의 숫자는 27 비트를 차지하므로 제곱을 배정 밀도로 정확하게 계산할 수 없습니다. 물론 확장 배정 밀도로이 숫자의 제곱을 계산할 수는 있지만 결과 알고리즘은 더 이상 단일 / 이중 시스템으로 이식 할 수 없습니다. 또한 다중 정밀도 곱셈 알고리즘의 이후 단계에서는 모든 부분 곱이 배정 밀도로 계산되었다고 가정합니다. 이중 및 확장 이중 변수의 혼합을 올바르게 처리하면 구현 비용이 훨씬 더 많이 듭니다.

마찬가지로 배정 밀도 숫자의 배열로 표현되는 다중 정밀도 숫자를 추가하는 이식 가능한 알고리즘은 이중 반올림 산술에서 실패 할 수 있습니다. 이러한 알고리즘은 일반적으로 Kahan의 합산 공식과 유사한 기술에 의존합니다. Errors In Summation 에 제공된 합산 공식에 대한 비공식적 인 설명에서 알 수 있듯이 if s및 is$y$ 부동 소수점 변수 $|s| \geq |y|$ 그리고 우리는 다음을 계산합니다.

```
t = s + y;
e = (s-t) + y;
```

그런 다음 대부분의 산술 e에서 계산에서 발생한 반올림 오류를 정확하게 복구합니다 t. 이 기술은 이중 반올림 산술에서는 작동하지 않습니다. if $s = 2^{52} + 1$ and $y = 1/2^{-2^{-54}}$ 이

면 s + y먼저 확장 배정 밀도에서 2^{52} + 3/2로 반올림하고이 값은 다음으로 반올림됩니다. 2^{52} + 2는 짝수 반올림 규칙에 따라 배정 밀도로 표시됩니다. 따라서 계산의 순 반올림 오류 t는 1/2 + 2^{-54}입니다. 이는 배정 밀도로 정확하게 표현할 수 없으므로 위에 표시된 표현식으로 정확하게 계산할 수 없습니다. 여기서 다시, 확장 된 배정 밀도로 합계를 계산하여 반올림 오류를 복구 할 수 있지만 최종 출력을 배정 밀도로 다시 줄이기 위해 프로그램이 추가 작업을 수행해야하며 이중 반올림이이 프로세스에 영향을 미칠 수 있습니다. 너무. 이러한 이유로 이러한 방법으로 다중 정밀도 산술을 시뮬레이션하기위한 이식 가능한 프로그램은 다양한 기계에서 정확하고 효율적으로 작동하지만 확장 기반 시스템에서 광고 된대로 작동하지 않습니다.

마지막으로, 첫눈에 올바른 반올림에 의존하는 것처럼 보이는 일부 알고리즘은 실제로 이중 반올림으로 올바르게 작동 할 수 있습니다. 이러한 경우 이중 반올림에 대처하는 비용은 구현에있는 것이 아니라 알고리즘이 광고 된대로 작동하는지 확인하는 데 있습니다. 설명하기 위해 정리 7의 다음 변형을 증명합니다.

정리 7 '

m 과 *n* 이 *IEEE 754* 배정 밀도로 표현 가능한 정수인 경우 | *m* | < 2^{52} 및 *n* 은 특수한 형태 *n* = 2^{i} + 2^{j} 이고 *(m ⊘ n) n* = *m*입니다. 정도.⊗

증명

손실없이 *m* > 0이라고 가정합니다. *q* = *m ⊘ n* 이라고 가정합니다 . 2의 거듭 제곱으로 스케일링하면 2^{52} *m* <2^{53} 이고 *q*에 대해서도 마찬가지로 동일한 설정을 고려할 수 있습니다 . 따라서 *m* 과 *q* 는 모두 최하위 비트가 단위 자리를 차지하는 정수입니다 (즉, ulp (*m*) = ulp (*q*) = 1). 스케일링 전에 *m* <2^{52}로 가정 했으므로 스케일링 후 *m* 은 짝수 정수입니다. 또한,의 스케일 값 때문에 *m* 및 *Q* 충족의 *m* / 2 <≤ *q* <2 *m* , *n* 의 해당 값은 *m* 또는 *q* 중 어느 것이 더 큰지 에 따라 두 가지 형태 중 하나를 가져야합니다 . *q* < *m* 이면 분명히 1 < *n* <2이고 *n* 은 2의 두 거듭 제곱의 합이므로 , *N* = 1 + 2^{-K} 일부 *K* ; 유사하게 *q* > *m* 이면 1/2 < *n* <1이므로 *n* = 1/2 + 2-$^{(k+1)}$ 입니다. (*n* 은 2의 두 거듭 제곱의 합이므로 *n*의 가능한 가장 가까운 값1은 *n* = 1 + 2^{-52} 입니다. 왜냐하면 *m* 은 / (1 + 2^{-52}가)보다 작고 다음 배정도 수보다 크지 않는다 *m* , 우리가 할 수 *Q* = *m*).

하자 *E* 나타내고 컴퓨팅 라운딩 에러 *q*는 , 그래서 *Q* = *m* / *N* + *E* 및 계산 값 *Q N* 의 (한두번) 둥근 값 것이다 *m* + *NE* . 먼저 각 부동 소수점 연산이 배정 밀도로 올바르게 반올림되는 경우를 고려하십시오. 이 경우 | *e* | <1/2. 경우 *n*이 형태를 갖는 2 + $2^{-(K+1)}$ 다음, *NE* = *NQ* - *m*은 2의 정수배 인 $^{-(K+1)}$⊗ 및 | *ne* | <1/4 + 2-$^{(k+2)}$. 이것은 | *ne* | ≤1/4. 리콜 차이 있음 *m* 및 다음으로 큰 표현할 수는 1이고, 차이 *m* , 다음 작은 표현할 수가 하나 인 경우 1 *m* > 2^{52} 1/2 경우 *m*은 = $2^{(52)}$. 따라서 | *ne* | ≤1/4, *m* + *ne* 는 *m*으로 반올림됩니다 . (*m* = 2^{52} 이고 *ne* = -1/4 인 경우에도 제품은 *m*으로 반올림됩니다.반올림-짝수 규칙에 의해.) 유사하게, *n* 이 1 + 2^{-k} 형식 이면 *ne* 는 2^{-k} 및 | *ne* | <1/2 + 2-$^{(k+1)}$; 이것은 의미 | *ne* | ≤1/2. *m* 이 *q* 보다 엄격하게 크기 때문

에 $m = 2^{52}$ 를 가질 수 없습니다 . 따라서 m 은 가장 가까운 표현 가능한 이웃과 ± 1 만큼 다릅니다 . 따라서, | ne | 1/2, 다시 $m + ne$ 는≤m . (해도 | 네브라스카 | = 1/2, 생성물 것이다 라운드 m 때문에 라운드 관계 투에도 규칙에 의하여 m이 짝수이다.)이 올바르게 반올림 연산을위한 증명을 완성한다.

이중 반올림 산술에서는 q 가 올바르게 반올림 된 몫 (실제로는 두 번 반올림 되었음에도 불구하고)이 발생할 수 있습니다 . 따라서 | e | 위와 같이 <1/2. 이 경우 q n 이 두 번 반올림 된다는 사실을 고려한다면 이전 단락의 주장에 호소 할 수 있습니다 . 이를 설명하기 위해 IEEE 표준에서는 확장 된 이중 형식이 최소 64 개의 유효 비트를 전달해야하므로 숫자 m ± 1/2 및 m ± 1/4을 확장 배정 밀도로 정확하게 표현할 수 있습니다. 따라서 n 이 $1/2 + 2^{-(k+1)}$ 형식 이면 | ne |⊗ ≤1/4 다음 라운딩 $m + NE$ 확장 배 정밀도가 결과를 생성해야 상이 m 최대 1/4, 그리고 전술 한 바와 같이,이 값 것이다 라운드 m 배정도이다. 마찬가지로 n 이 $1 + 2^{-k}$ 형식 이면 | ne | ≤1/2 다음 라운딩 $m + NE$ 확장 배 정밀도가 결과를 생성해야 상이 m 최대 1/2,이 값 것이다 라운드 m 배정도이다. (이 경우 $m > 2^{52}$ 를 기억하십시오.)

마지막으로, 우리는 q 가 이중 반올림으로 인해 올바르게 반올림 된 몫이 아닌 경우를 고려해야 합니다. 이 경우, 우리는 | e | $<1/2 + 2^{-(d+1)}$ 최악의 경우, 여기서 d 는 확장 double 형식의 추가 비트 수입니다. (기존의 모든 확장 기반 시스템은 정확히 64 개의 중요한 비트가있는 확장 된 double 형식을 지원합니다.이 형식의 경우 d = 64-53 = 11입니다.) 두 번째 반올림이 반올림으로 결정될 때만 반올림 된 결과가 잘못 반올림되기 때문입니다. -ties-to-even 규칙, q 는 짝수 정수 여야합니다. 따라서 n 의 형태가 $1/2 + 2^{-(k+1)}$ 이면다음 $NE = NQ - m$은 2의 정수배 인 2^{-K}는 한

$$| ne | <(1/2 + 2^{-(K+1)}) (2 + 2^{-(D+1)}) 1/4 + 2 = {}^{(-K+2)} + (2)^{-(D+2)} + 2^{-(k+d+2)}.$$

경우 케이 D |, 이것은 의미 ne | 1/4. 경우 $K > D$, 우리는이 | ne | $1/4 + 2^{-(d+2)}$. 두 경우 모두 제품의 첫 번째 반올림은 m 과 최대 1/4만큼 다른 결과를 제공하고 이전 인수에 의해 두 번째 반올림은 m로 반올림됩니다 . 마찬가지로 n 이 $1 + 2^{-k}$ 형식 이면 ne 는 $2^{-(k-1)}$ 의 정수 배수 이고≤ ≤≤

$$| ne | <1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}.$$

경우 케이 D |, 이것은 의미 ne | 1/2. 경우 $K > D$, 우리는이 | ne | $1/2 + 2^{-(d+1)}$. 두 경우 모두 제품의 첫 번째 반올림은 m 과 최대 1/2만큼 다른 결과를 제공 하고 다시 이전 인수에 의해 두 번째 반올림은 m로 반올림됩니다 . 지≤ ≤≤

앞의 증명은 몫이 발생하는 경우에만 곱이 이중 반올림을 발생할 수 있음을 보여줍니다. 그 후에도 올바른 결과로 반올림됩니다. 증명은 또한 두 개의 부동 소수점 연산 만있는 프로그램의 경우에도 이중 반올림 가능성을 포함하도록 추론을 확장하는 것이 어려울 수 있음을 보여줍니다. 더 복잡한 프로그램의 경우 이중 및 확장 배정 밀도 계산의 일반적인 조합은 말할 것도없고 이중 반올림의 효과를 체계적으로 설명하는 것이 불가능할 수 있습니다.

확장 정밀도를위한 프로그래밍 언어 지원

앞의 예는 확장 된 정밀도 자체 가 해롭다는 것을 암시하기 위해 사용되어서는 안됩니다 . 프로그래머가 선택적으로 사용할 수있는 경우 많은 프로그램이 확장 된 정밀도의 이점을

누릴 수 있습니다. 불행히도 현재의 프로그래밍 언어는 프로그래머가 확장 정밀도를 사용해야하는시기와 방법을 지정할 수있는 충분한 수단을 제공하지 않습니다. 필요한 지원을 나타 내기 위해 확장 된 정밀도 사용을 관리하는 방법을 고려합니다.

배정 밀도를 공칭 작업 정밀도로 사용하는 휴대용 프로그램에서 더 넓은 정밀도 사용을 제어 할 수있는 다섯 가지 방법이 있습니다.

1. 확장 기반 시스템에서 가능한 경우 확장 된 정밀도를 사용하여 가장 빠른 코드를 생성하도록 컴파일하십시오. 분명히 대부분의 수치 소프트웨어는 각 연산의 상대 오류가 "머신 엡실론"에 의해 제한되는 것보다 더 많은 산술을 요구하지 않습니다. 메모리의 데이터가 배정 밀도로 저장 될 때 입력 데이터가 입력 될 때 (올바르게 또는 잘못) 반올림 된 것으로 가정되고 결과가 결과가 될 것이라고 가정하기 때문에 기계 엡실론은 일반적으로 해당 정밀도에서 가장 큰 상대적 반올림 오류로 간주됩니다. 마찬가지로 저장 될 때 반올림됩니다. 따라서 중간 결과 중 일부를 확장 된 정밀도로 계산하면 더 정확한 결과를 얻을 수 있지만 확장 된 정밀도는 필수적이지 않습니다. 이 경우

2. 합리적으로 빠르고 충분히 넓은 경우 double보다 넓은 형식을 사용하고, 그렇지 않으면 다른 것에 의지하십시오. 일부 계산은 확장 된 정밀도를 사용할 수있을 때 더 쉽게 수행 할 수 있지만 약간의 노력만으로 배정 밀도로 수행 할 수도 있습니다. 배정 밀도 숫자로 구성된 벡터의 유클리드 노름 계산을 고려하십시오. 요소의 제곱을 계산하고 더 넓은 지수 범위를 가진 IEEE 754 확장 double 형식으로 그 합계를 누적함으로써 실제 길이의 벡터에 대한 조기 언더 플로 또는 오버플로를 사소하게 방지 할 수 있습니다. 확장 기반 시스템에서 이것은 표준을 계산하는 가장 빠른 방법입니다. 단일 / 이중 시스템에서 확장 된 이중 형식은 소프트웨어에서 에뮬레이션되어야하며 (지원되는 경우) 이러한 에뮬레이션은 단순히 배정 밀도를 사용하는 것보다 훨씬 느립니다. 예외 플래그를 테스트하여 언더 플로 또는 오버플로가 발생했는지 확인하고, 그렇다면 명시 적 스케일링으로 계산을 반복합니다. 확장 된 정밀도의 사용을 지원하려면 언어는 프로그램이 사용할 방법을 선택할 수 있도록 가장 광범위하고 사용 가능한 형식의 표시와 각 형식의 정밀도 및 범위를 나타내는 환경 매개 변수를 모두 제공해야합니다. , 프로그램이 가장 넓은 빠른 형식이 충분히 넓은 지 (예 : 두 배보다 더 넓은 범위를 가짐) 확인할 수 있습니다.

3. 소프트웨어에서 에뮬레이션해야하는 경우에도 두 배보다 넓은 형식을 사용하십시오. 유클리드 표준 예제보다 더 복잡한 프로그램의 경우 프로그래머는 프로그램의 두 가지 버전을 작성하지 않고 속도가 느리더라도 확장 된 정밀도에 의존하기를 원할 수 있습니다. 다시 말하지만, 프로그램이 사용 가능한 가장 광범위한 형식의 범위와 정밀도를 결정할 수 있도록 언어는 환경 매개 변수를 제공해야합니다.

4. 더 넓은 정밀도를 사용하지 마십시오. 확장 된 범위를 사용하더라도 결과를 double 형식의 정밀도로 올바르게 반올림합니다. 위에서 언급 한 일부 예제를 포함하여 올바르게 반올림 된 배정 밀도 산술에 의존하도록 가장 쉽게 작성되는 프로그램의 경우, 언어는 중간 결과를 계산할 수 있더라도 프로그래머가 확장 정밀도를 사용해서는 안된다는 것을 표시하는 방법을 제공해야합니다. double보다 더 넓은 지수 범위를 가진 레지스터에서. (이러한 방식으로 계산 된 중간 결과는 메모리에 저장 될 때 언더 플로되면 여전히 이중 반올림이 발생할 수 있습니다. 산술 연산의 결과가 먼저 53 개의 유효 비트로 반올림 된 다음 비정규 화되어야 할 때 더 적은 유효 비트로 다시 반올림됩니다. 최종 결과는 비정규 화 된 숫자로 한 번만 반올림하여 얻은 결과와 다를 수 있습니다. 물론 이러한 형태의 이중 반올림은 실제 프로그램에 악영향을 미칠 가능성이 거의 없습니다.)

5. 결과를 double 형식의 정밀도와 범위로 올바르게 반올림합니다. 이러한 배정 밀도의 엄격한 적용은 수치 소프트웨어 또는 이중 형식의 범위와 정밀도의 한계에 가까운 산술 자체를 테스트하는 프로그램에 가장 유용합니다. 이러한 세심한 테스트 프로그램

은 이식 가능한 방식으로 작성하기 어려운 경향이 있습니다. 결과를 특정 형식으로 반올림하기 위해 더미 서브 루틴 및 기타 트릭을 사용해야 할 때 훨씬 더 어려워지고 오류가 발생하기 쉽습니다. 따라서 확장 기반 시스템을 사용하여 모든 IEEE 754 구현에 이식 할 수 있어야하는 강력한 소프트웨어를 개발하는 프로그래머는 특별한 노력 없이 단일 / 이중 시스템의 산술을 에뮬레이션 할 수 있다는 것을 금방 알게 될 것입니다.

현재 언어는 이러한 5 가지 옵션을 모두 지원하지 않습니다. 사실 프로그래머에게 확장 된 정밀도의 사용을 제어 할 수있는 능력을 부여하려는 언어는 거의 없습니다. 한 가지 주목할 만한 예외는 ISO / IEC 9899 : 1999 프로그래밍 언어-C 표준으로, C 언어의 최신 개정판으로 현재 표준화의 마지막 단계에 있습니다.

C99 표준을 사용하면 구현에서 일반적으로 해당 유형과 관련된 형식보다 더 넓은 형식으로 식을 평가할 수 있지만 C99 표준에서는 세 가지 식 평가 방법 중 하나만 사용하도록 권장합니다. 권장되는 세 가지 방법은 표현식이 더 넓은 형식으로 "승격"되는 정도에 따라 특성화되며 구현시 전 처리기 매크로를 정의하여 사용하는 방법을 식별하는 것이 좋습니다 FLT_EVAL_METHOD. FLT_EVAL_METHOD0이면 각 표현식은 해당하는 형식으로 평가됩니다. 그 유형에; 경우 FLT_EVAL_METHOD1, float표현 형식으로 승격되는에 해당합니다 double; 및 if FLT_EVAL_METHOD가 2 float이고 double식은에 해당하는 형식으로 승격됩니다 long double. (구현은 설정할 수 있습니다FLT_EVAL_METHOD에가 -1 C99 표준은 또한 있어야). 식 평가 방법은 결정할 수없는 것을 나타 내기 위해 <math.h>헤더 파일 타입 정의 float_t와 double_t폭만큼 최소한이고, float그리고 double각각을하고 평가하기 위해 사용되는 유형과 일치하도록 의도 float및 double표현. 예를 들어,은 FLT_EVAL_METHOD2, 모두 float_t하고 double_t있습니다 long double. 마지막으로 C99 표준은 <float.h>헤더 파일이 각 부동 소수점 유형에 해당하는 형식의 범위와 정밀도를 지정하는 전 처리기 매크로를 정의 하도록 요구합니다 .

C99 표준에서 요구하거나 권장하는 기능 조합은 위에 나열된 5 가지 옵션 중 일부를 지원하지만 전부는 아닙니다. 예를 들어 구현에서 long double유형을 확장 된 double 형식으로 매핑하고 FLT_EVAL_METHOD2로 정의 하는 경우 프로그래머는 확장 된 정밀도가 상대적으로 빠르다고 합리적으로 가정 할 수 있으므로 Euclidean 표준 예제와 같은 프로그램은 단순히 유형 long double(또는 double_t) 의 중간 변수를 사용할 수 있습니다 . 반면에 동일한 구현은 익명 표현식이 메모리에 저장 될 때 (예 : 컴파일러가 부동 소수점 레지스터를 유출해야하는 경우) 확장 된 정밀도로 유지해야하며 선언 된 변수에 할당 된 표현식의 결과를 저장해야합니다.double레지스터에 보관할 수 있더라도 배정 밀도로 변환합니다. 따라서 현재 확장 기반 하드웨어에서 가장 빠른 코드를 생성하기 위해 double또는 double_t유형을 컴파일 할 수 없습니다.

마찬가지로 C99 표준은이 섹션의 예제에서 설명하는 일부 문제에 대한 솔루션을 제공하지만 전부는 아닙니다. log1p함수 의 C99 표준 버전은 식이 1.0 + x변수 (모든 유형)에 할당되고 해당 변수가 전체적으로 사용되는 경우 올바르게 작동하도록 보장 됩니다. 그러나 배정밀도 숫자를 높은 부분과 낮은 부분으로 분할하기위한 이식 가능하고 효율적인 C99 표준 프로그램은 더 어렵습니다. double표현식이 배정 밀도로 올바르게 반올림 된다는 것을 보장할 수없는 경우 올바른 위치에서 분할하고 이중 반올림을 피할 수있는 방법 ? 한 가지 해결책은double_t단일 / 이중 시스템에서는 배정 밀도로 분할을 수행하고 확장 기반 시스템에서는 확장 된 정밀도로 분할을 수행하므로 두 경우 모두 산술이 올바르게 반올림됩니다. 정리 14는 기본 산술의 정밀도를 알고있는 경우 모든 비트 위치에서 분할 할 수 있으며 FLT_EVAL_METHOD및 환경 매개 변수 매크로가이 정보를 제공해야 한다고 말합니다 .

다음 조각은 하나의 가능한 구현을 보여줍니다.

```
#include <math.h>

#include <float.h>
```

```
#if (FLT_EVAL_METHOD == 2)
#define PWR2 LDBL_MANT_DIG-(DBL_MANT_DIG / 2)
#elif ((FLT_EVAL_METHOD == 1) || (FLT_EVAL_METHOD == 0))
#define PWR2 DBL_MANT_DIG-(DBL_MANT_DIG / 2)
#그밖에
#error FLT_EVAL_METHOD 알 수 없음!
#endif
...
    더블 x, xh, xl;
    double_t m;
    m = scalbn (1.0, PWR2) + 1.0; // 2 ** PWR2 + 1
    xh = (m * x)-((m * x)-x);
    xl = x-xh;
```

물론,이 솔루션을 찾기 위해 프로그래머는 double표현식이 확장 된 정밀도로 평가 될 수 있고, 이어지는 이중 반올림 문제로 인해 알고리즘이 오작동 할 수 있으며, 정리 14에 따라 확장 된 정밀도가 대신 사용될 수 있음을 알아야합니다. 명백한 해결책은 각 표현식이 배정밀도로 올바르게 반올림되도록 지정하는 것입니다. 확장 기반 시스템에서는 반올림 정밀도 모드를 변경하기 만하면되지만 안타깝게도 C99 표준은이를 수행 할 수있는 이식 가능한 방법을 제공하지 않습니다. (부동 소수점 C 편집, 지원 부동 소수점으로 C90 표준을 만들 수있는 변화를 지정된 작업 문서의 초기 초안은, 정밀 반올림 모드와 시스템의 구현을 제공하는 것이 좋습니다 fegetprec및fesetprec반올림 방향을 가져오고 설정하는 fegetround및 fesetround함수 와 유사한 반올림 정밀도를 가져오고 설정하는 함수입니다. 이 권장 사항은 C99 표준이 변경되기 전에 제거되었습니다.)

공교롭게도, 다양한 정수 산술 기능을 가진 시스템 간의 이식성을 지원하는 C99 표준의 접근 방식은 다양한 부동 소수점 아키텍처를 지원하는 더 나은 방법을 제안합니다. 각 C99 표준 구현 <stdint.h>은 크기와 효율성에 따라 이름이 지정된 구현에서 지원하는 정수 유형을 정의 하는 헤더 파일을 제공합니다. 예를 들어 int32_t정확히 32 비트 너비 int_fast16_t의 정수 유형이고, 최소 16 비트 너비 이상의 구현에서 가장 빠른 정수 유형이며, intmax_t 지원되는 가장 넓은 정수 유형입니다. 부동 소수점 유형에 대한 유사한 체계를 상상할 수 있습니다. 예를 들어 float53_t정확히 53 비트 정밀도로 부동 소수점 유형의 이름을 지정할 수 있지만 범위가 더 넓을 수 있습니다.float_fast24_t최소 24 비트 정밀도로 구현의 가장 빠른 유형의 floatmax_t이름을 지정할 수 있으며 지원되는 가장 광범위하고 합리적으로 빠른 유형의 이름을 지정할 수 있습니다. 빠른 유형을 사용하면 확장 기반 시스템의 컴파일러에서 이름이 지정된 변수의 값이 레지스터 유출의 결과로 변경되지 않아야한다는 제약 조건에 따라 가능한 가장 빠른 코드를 생성 할 수 있습니다. 정확한 너비 유형은 확장 기반 시스템의 컴파일러가 지정된 정밀도로 반올림하도록 반올림 정밀도 모드를 설정하여 동일한 제약 조건에 따라 더 넓은 범위를 허용합니다. 드디어,double_t엄격한 이중 평가를 제공하는 IEEE 754 이중 형식의 정밀도와 범위를 모두 사용하여 유형의 이름을 지정할 수 있습니다. 그에 따라 명명 된 환경 매개 변수 매크로와 함께 이러한 체계는 위에서 설명한 다섯 가지 옵션을 모두 쉽게 지원하고 프로그래머가 프로그램에 필요한 부동 소수점 의미를 쉽고 명확하게 나타낼 수 있도록합니다.

확장 된 정밀도에 대한 언어 지원이 그렇게 복잡해야합니까? 싱글 / 더블 시스템에서는 위에 나열된 다섯 가지 옵션 중 네 가지가 일치하며 빠르고 정확한 너비 유형을 구별 할 필요가 없습니다. 그러나 확장 기반 시스템은 어려운 선택을 제시합니다. 두 가지를 혼합하는 것만 큼 효율적으로 순수 배정 밀도 또는 순수 확장 정밀도 계산을 지원하지 않으며 다른

프로그램은 서로 다른 혼합을 요구합니다. 더욱이 확장 된 정밀도를 사용할시기를 선택하는 것은 컴파일러 작성자에게 맡겨서는 안됩니다. 컴파일러 작성자는 종종 벤치 마크에 의해 (때로는 수치 분석가에 의해 노골적으로 말함) 부동 소수점 산술을 "본질적으로 부정확한"것으로 간주하므로 그럴 가치도없고 능력도 없습니다. 정수 산술의 예측 가능성. 대신 프로그래머에게 선택을 제시해야합니다.

결론

앞서 언급 한 내용은 확장 기반 시스템을 비난하기위한 것이 아니라 몇 가지 오류를 폭로하기위한 것입니다. 첫 번째는 모든 IEEE 754 시스템이 동일한 프로그램에 대해 동일한 결과를 제공해야한다는 것입니다. 우리는 확장 기반 시스템과 싱글 / 더블 시스템 간의 차이점에 중점을 두었지만 이러한 각 제품군 내의 시스템 간에는 더 많은 차이점이 있습니다. 예를 들어, 일부 단일 / 이중 시스템은 두 개의 숫자를 곱하고 한 번의 최종 반올림으로 세 번째를 더하는 단일 명령을 제공합니다. 이 작업을 융합 곱하기 더하기 라고 합니다., 동일한 프로그램이 서로 다른 단일 / 이중 시스템에서 다른 결과를 생성하도록 할 수 있으며 확장 된 정밀도와 마찬가지로 동일한 프로그램이 사용 여부와시기에 따라 동일한 시스템에서 다른 결과를 생성하도록 할 수도 있습니다. (융합 곱셈-더하기는 분할 할 필요없이 다중 정밀도 곱셈을 수행하기 위해 이식 할 수없는 방식으로 사용할 수 있지만 Theorem 6의 분할 프로세스를 방해 할 수 있습니다.) IEEE 표준은 이러한 분할을 예상하지 못했지만 그럼에도 불구하고 중간 제품은 사용자가 제어 할 수없는 "목적지"로 전달되며, 정확하게 보유 할 수있을만큼 넓고 최종 합계는 단 정밀도 또는 배정 밀도 대상에 맞게 올바르게 반올림됩니다.

IEEE 754가 특정 프로그램이 제공해야하는 결과를 정확하게 규정한다는 생각은 그럼에도 불구하고 매력적입니다. 많은 프로그래머는 프로그램의 동작을 이해할 수 있고 프로그램을 컴파일하는 컴파일러 나이를 실행하는 컴퓨터를 참조하지 않고도 올바르게 작동한다는 것을 증명할 수 있다고 믿고 싶어합니다. 여러면에서 이러한 믿음을 지원하는 것은 컴퓨터 시스템 및 프로그래밍 언어 설계자에게 가치있는 목표입니다. 불행히도 부동 소수점 산술의 경우 목표를 달성하기가 사실상 불가능합니다. IEEE 표준의 작성자는이를 알고 있었으며 이를 달성하려고 시도하지 않았습니다. 그 결과, 컴퓨터 산업 전반에 걸쳐 (대부분의) IEEE 754 표준을 거의 보편적으로 준수 함에도 불구하고, 휴대용 소프트웨어 프로그래머는 예측할 수없는 부동 소수점 연산에 계속해서 대처해야합니다.

프로그래머가 IEEE 754의 기능을 활용하려면 부동 소수점 산술을 예측할 수있는 프로그래밍 언어가 필요합니다. C99 표준은 프로그래머가 프로그램의 여러 버전을 각각 하나씩 작성하도록 요구하는 대신 예측 가능성을 어느 정도 향상시킵니다.`FLT_EVAL_METHOD`. 미래의 언어가 프로그래머가 IEEE 754 의미론에 의존하는 범위를 명확하게 표현하는 구문을 사용하여 단일 프로그램을 작성하도록 허용할지 여부는 아직 밝혀지지 않았습니다. 기존의 확장 기반 시스템은 컴파일러와 하드웨어가 주어진 시스템에서 계산을 수행하는 방법을 프로그래머보다 더 잘 알 수 있다고 가정함으로써 그러한 전망을 위협합니다. 그 가정은 두 번째 오류입니다. 계산 된 결과에 필요한 정확도는이를 생성하는 기계가 아니라 그로부터 도출 될 결론에만 의존하고 프로그래머, 컴파일러 및 하드웨어에 따라 달라집니다. 프로그래머는 이러한 결론이 무엇인지 알 수 있습니다.

[1] 다른 표현의 예는 부동 슬래시 와 부호있는 로그입니다 [Matula and Kornerup 1985; Swartzlander와 Alexopoulos 1975].

[2] 이 용어는 Forsythe and Moler [1967]에 의해 도입되었으며 일반적으로 이전 용어 인 가수를 대체했습니다.

[3] 이것은 지수가 유효 숫자의 왼쪽에 저장되는 일반적인 배열을 가정합니다.

4
숫자 *z* 가 +1보다 크거나 . 이러한 방식으로 범위를 벗어난 번호는 추후 공지가있을 때까지 고려되지 않습니다. $\beta^{e_{max}} \beta^{e_{min}}$

5
하자 *Z*는 '근사 부동 소수점 숫자 *Z*를 . 그러면 *dd* ... *d*- (*z* / ^{e}) $^{p-1}$ 은 *z* ' -*z* / ulp (*z* ')와 같습니다. 오차를 측정하는 더 정확한 공식은 *z* ' -*z* / ulp (*z*)입니다. -에드. $|\beta|\beta||||$

6
700 70 0.1되지 때문에 - 0.0708 = 0.0292, ULP 환산 오차 (0.0292)는 708 ulps는 기초적이다. -에드.

7
(*x* - *y*) (*x* + *y*) 표현식 이 치명적인 취소를 유발하지는 않지만 또는 이면 *x* ^{2} - *y* ^{2} 보다 약간 덜 정확 합니다. 이 경우 (*x* - *y*) (*x* + *y*)에는 세 개의 반올림 오류가 있지만 *x* ^{2} 및 *y* ^{2} 중 더 작은 값을 계산할 때 커밋 된 반올림 오류 가 최종 뺄셈에 영향을주지 않기 때문에 x ^{2} -y ^{2} 에는 ^{2} 개만 있습니다. $x \gg y\, x \ll y$

8
일반적으로 올바로 반올림 이라고도합니다 . -에드.

9 일
때, N = 845, X $_{N}$ = 9.45, X $_{N}$ + 0.555 = 10.0, 10.0 - 9.45 = 0.555. 따라서 *n* > 845 인 경우 $_{xn}$ = x $_{845}$ 입니다 .

10
이진수에서 *q* 는 같을 수 없습니다 . -에드. $\bar{q}$

11
독자를위한 연습 문제로 남았습니다. 증명을 2가 아닌 다른 기초로 확장하십시오.-Ed.

12
Knuth ([1981], 211 페이지)가이 아이디어를 Konrad Zuse에 귀속 시켰지만 이것은 Goldberg [1967]에 의해 처음 출판 된 것으로 보입니다. 13 Kahan에 따르면 확장 된 정밀도는 64 비트의 의미를 가지며, 이것이주기 시간을 늘리지 않고 Intel 8087에서 수행 할 수있는 가장 넓은 정밀도이기 때문입니다 [Kahan 1988]. 14 Kahan과 LeBlanc [1985]는 내적을 기본 작업 중 하나로 포함하는 것에 반대하는 몇 가지 주장을 제시했습니다.

15
Kirchner는 다음과 같이 씁니다. 클럭 사이클 당 하나의 부분 제품에서 하드웨어의 1ulp 이내로 내부 제품을 계산할 수 있습니다. 추가로 필요한 하드웨어는 해당 속도에 필요한 멀티 플라이어 어레이와 비교됩니다. 16 CORDIC은 Coordinate Rotation Digital Computer의 머리 글자이며 주로 시프트와 덧셈 (즉, 곱셈과 나눗셈이 거의 없음)을 사용하는 초월 함수를 계산하는 방법입니다 [Walther 1971]. 그 속도를 위해 어쨌든 필요한 멀티 플라이어 어레이에 비해 하드웨어가 추가로 필요한 방법입니다. d는 Intel 8087과 Motorola 68881 모두에서 사용됩니다. 17 요점 : IEEE 산술의 기본값은 오버플로 된 숫자를로 반올림하는 것이지만 기본값 을 변경할 수 있습니다 (반올림 모드 참조).

$^{\infty}$)

18
그들은이라고 비정규을 , 854에서 비정규 754에.

19
이것이 표준의 가장 까다로운 측면 중 하나의 원인입니다. 자주 언더 플로되는 프로그램은 소프트웨어 트랩을 사용하는 하드웨어에서 눈에 띄게 느리게 실행되는 경우가 많습니다. 20 "트래핑"NaN이 작업에 포함되지 않으면 잘못된 예외가 발생하지 않습니다. IEEE Std 754-1985의 섹션 6.2를 참조하십시오. -에드.

21
은 x와 y가 모두 음수 인 경우 보다 클 수 있습니다 . -에드. $\equiv \equiv$

22 는 범위에있을 수 있기 때문에, 만약 *X* <1, *N* <0, *X* - *N* 언더 임계치보다 단지 조금의 작은 다음 등 않을 수도 오버플 모든 IEEE 정밀도에서, 사람 - 전자 분 < 전자 최대 . $2^{e_{min}} \cdot r^{p} \simeq 2^{-e_{min}} < 2^{e_{max}}$

23 이는 설계자가 부동 소수점 명령어의 정밀도가 실제 연산과 독립적 인 "직교"명령어 세트를 좋아하기 때문일 것입니다. 곱셈에 대한 특별한 경우를 만들면이 직교성이 파괴됩니다.

24 이것은 단 정밀도 상수 인 반면는 배정 밀도 상수 인 일반적인 규칙을 가정합니다 . 3.03.0DO

25 0 0 = 1 이라는 결론은 *f* 가 일정하지 않다는 제한에 따라 달라집니다 . 이 제한이 제거되면 *f*를 동일하게 0 함수로 두면 lim x 0 *f* (*x*) *g* (*x*)에 대해 가능한 값으로 0 이 제공 되므로 0 0 은 NaN으로 정의되어야합니다. →

26 0 0 의 경우 타당성 주장을 할 수 있지만 Graham, Knuth 및 Patashnik의 "Concrete Mathematics"에서 설득력있는 주장을 찾을 수 있으며 이항 정리가 작동하려면 0 0 = 1 이라고 주장합니다 . -에드. 27 반올림 모드가-쪽으로 반올림되지 않는 한, 이 경우 *x* - *x* = -0.
∞

28 그들이 오히려보다 느린 서브 루틴 호출에 저렴한 점프를 사용한다는 점에서 VAX의 VMS 수학 라이브러리는 인라인 (in-line) 절차 대체의 약한 양식을 사용 하고 지침을 제공합니다. CALLSCALLG

29 사전 대체의 어려움은 직접 하드웨어 구현이 필요하거나 소프트웨어로 구현 된 경우 연속 부동 소수점 트랩이 필요하다는 것입니다. -에드.

30 이 비공식 증명에서 = 2 라고 가정 하여 4의 곱셈이 정확하고 i가 필요하지 않습니다 . βδ

31 이것은 w를 추가해도 수행되지 않는 경우의 합계입니다. w 추가가 수행되는 특수한 경우에는 추가 인수가 필요합니다. -에드.

32 반올림은 (*k* *x* + *w* *k*) 가 *k* *x* 의 형태를 유지하는 경우에만 *k* *x* + *w* *k* - *r* *k*를 제공 합니다 . - 에드. βββββ β

도서관 | 목차 | 이전 | 다음 | 인덱스