고성능 부동 소수점 연산기에 대한 연구

박 우 찬[†] · 한 탁 돈[††]

요 약

부동 소수점 연산기는 고성능 컴퓨터에서 필수적이며, 최근 대부분의 고성능의 컴퓨터에서는 고성능의 부동 소수점 연산기가 내장되고 있는 추세이다. 부동 소수점 연산이 고속화 되면서 부동 소수점 연산기에서 한개의 단계를 차지하는 반올림 단계가 전체 부동 소수점 연산에 큰 영향을 미친다. 반올림 단계에서는 별도의 고속 가산기를 필요로하여 많은 처리 시간과 칩 면적을 차지하기 때문이다. 본 연구는 고성능 부동 소수점 연산기의 근간을 이루는 부동 소수점 덧셈/뺄셈기, 곱셈기, 나눗셈기의 처리 알고리즘을 살펴보고, 이를 분석하여 새로운 반올림 처리 알고리즘을 갖는 연산기를 제안하였다. 제안된 부동 소수점 연산기들은 반올림 처리를 위한 별도의 시간을 요하지 않고, 반올림 단계를 위한 가산기나 증가기를 필요로 하지 않는다. 따라서, 제안하는 부동 소수점 연산기들은 성능면이나 차지 면적 면에서 모두 효율적이다.

A Study on High Performance Floating Point Unit

Woo-Chan Park[†] · Tack-Don Han[††]

ABSTRACT

An FPU(Floating Point Unit) is the principle component in high performance computer and is placed on a chip together with main processing unit recently. As a processing speed of the FPU is accelerated, the rounding stage, which occupies one of the floating point processing steps for floating point operations, has a considerable effect on overall floating point operations. In this paper, by studying and analyzing the processing flows of the conventional floating point adder/subtractor, multipler and divider, which are main component of the FPU, efficient rounding mechanisms are presented. Proposed mechanisms do not require any additional execution time and any high speed adder for rounding operation. Thus, performance improvement and cost-effective design can be achieved by this approach.

1. 서 론

현재 정보처리시 데이타가 표현해야 하는 수의 영역이 확대되고 반도체 기술이 발전함에 따라 부동 소수점 연산기(Floating Point Unit)는 고성능을 요구하는 컴퓨터에 필수적으로 사용되고 있다. 특히, 최근에는 부동 소수점 연산기가 중앙처리 장치와 함께 한 칩에 내장되고 있다[1, 2, 3, 15]. 부동 소수점 연산을 하드웨어로 지원하는 경우 차지하는 면적으로 인하여, 부동 소수점 연산의 근간을 이루는 부동 소수점 덧셈/뺄셈, 곱셈, 나눗셈 연산에 대한 전용 하드웨어를 제공하고 나머지 연산은 약간의 하드웨어를 첨가하거나 소프트웨어로 처리해주는 것이 주를 이룬다.

부동 소수점 연산기가 고성능화 되어가면서 부동 소수점 연산기의 한 단계인 반올림 단계가 전체 부동

※이 논문은 1995년 학술진흥재단의 해외파견연구교수 연구비에 의하여 연구되었음.

† 준 회 원:연세대학교 컴퓨터과학과
†† 종신회원:연세대학교 컴퓨터과학과

논문접수:1997년 8월 29일, 심사완료:1997년 10월 6일

소수점에 매우 큰 영향을 미친다. 반올림을 지원해주기 위해서는 별도의 고속 덧셈기를 필요로 하여 많은 소요 시간과 칩 면적을 필요로 하기 때문이다. 부동 소수점 덧셈과 곱셈 연산기는 일반적으로 3-4 단계의 파이프라인으로 이루어져 있는데, 이중 한개가 반올림을 위한 파이프라인이다. 그런데, 부동 소수점 연산이 대부분이 덧셈/뺄셈과 곱셈 연산임으로, 고성능의 부동 소수점 연산기를 구현하는데 있어서 반올림 처리부는 대단히 큰 비중을 차지한다.

본 연구는 고성능 부동 소수점 연산기의 근간을 이루는 부동 소수점 덧셈/뺄셈기, 곱셈기, 나눗셈기의 처리 알고리즘을 살펴보고, 이를 분석하여 새로운 반올림 처리 알고리즘을 갖는 연산기를 제안하였다. 제안된 부동 소수점 연산기들은 반올림 처리를 위한 별도의 시간을 요하지 않고, 반올림 단계를 위한 가산기나 증가기를 필요로 하지 않는다. IEEE 표준안[13]이 발표된 이후의 거의 모든 부동 소수점 연산기는 IEEE 표준안을 따르기 때문에, 본 논문에서 반올림으로는 IEEE 표준안인 4가지 모드(mode)를 모두 지원하도록 하였다.

이번장에서는 IEEE 반올림과 앞으로 전개할 수식들의 정의들을 살펴보고, 2 장에서는 부동 소수점 덧셈/뺄셈 연산기의 기존의 처리 알고리즘을 분석하고 새로운 처리 알고리즘을 제시한다. 3 장에서는 부동 소수점 곱셈 및 나눗셈 연산기의 반올림 처리부에 대해서 살펴본다.

1.1 IEEE 반올림

IEEE 표준안에서는 부동 소수점 수를 표현하는데 있어서 정규화된 피연산자 A는 $A=(-1)^s\times 1.f\times 2^{e-bias}$ 과 같이 표현된다. s는 분수부에 대한 부호 bit이며, f는 절대치 형태의 분수부이며, e는 바이어스(bias)형태의 지수부이다. 분수부의 정규화된 형태는 MSB(Most Significant bit)가 1인 상태이며 부동 소수점 표현에서는 MSB가 생략된다. 이 MSB를 히든비트(hidden bit)라고 한다. 본 논문에서의 분수부는 히든비트를 포함한 분수부를 말한다.

부동 소수점 연산시 분수부의 비트수가 정해져 있기 때문에 연산 도중에 정보의 손실이 발생하게 된다. 예를 들면, 부동 소수점 덧셈/뺄셈의 경우 지수 정렬 단계와 부동 소수점 곱셈인 경우 n×n 곱셈 연산시 2n 비트가 발생하는 경우이다. 이때, 올바른 반올림을 위하여 손실되는 정보를 가지고 있어야 한다. 이를 위하여 3개의 비트인 가드 비트(G:Guard bit), 라운드 비트(R:Round bit), 스티키 비트(Sy:Sticky bit)가 정의되어 있다[5]. G는 LSB(least significant bit)보다 작은 가중치를 가져서 손실되는 분수부중에 MSB가 되며, R은 그 다음 비트가 되며, Sy는 손실되는 부분중 G와 R을 제외한 모든 비트를 논리합 연산한 값이다. 그런데 G는 부동 소수점 덧셈/뺄셈시만 필요로 하고 부동 소수점 곱셈 및 나눗셈 연산에는 필요가 없다.

IEEE 표준안에는 round-to-nearest mode, round-to-zero mode, round-to-positive-infinity mode, round-to-negative-infinity mode인 4가지 반올림 방식을 규정하고 있다. Round-to-nearest mode는 반올림으로 인한 오차가 최소가 되도록 가까운 쪽으로 반올림 하는 방법이며, 반올림으로 인한 오차가 동일할 경우에는 LSB가 0이 되도록 반올림 하는 것이다. Round-to-zero mode는 반올림의 결과가 0에 가깝게 하도록 반올림을 하는 것이다. Round-to-positive-infinity mode는 양의 무한대로 반올림을 하는 것이다. Round-to-negative-infinity mode는 음의 무한데로 반올림을 하는 것이다. 위의 4 가지 반올림 방법은 부호의 값에 따라 round-to-nearest mode, round-to-zero mode, round-to-infinity mode로 나눌 수 있다. 본 논문에서는 전개의 편의성을 위해서 위의 3가지 방법에 대해서 논하겠다.

다음은 분수부의 LSB, G, R, Sy bit에 대한 round-to-nearest mode, round-to-zero mode, round-to-infinity mode의 결과이다. 여기서 return 0는 반올림의 결과 버림을 뜻하며, return 1는 반올림의 결과 올림을 뜻한다. 그런데, G는 부동 소수점 덧셈/뺄셈 연산에만 필요한 비트 임으로, 부동 소수점 곱셈 및 나눗셈 연산에서는 G를 무시한다.

Algorithm 1: $Rounding_{Nearest}(LSB, G, R, Sy)$

```
if (G = 0) return 0
else if ((R = 1) or (Sy = 1)) return 1
    else if (LSB = 0) return 0
        else return 1
```

Algorithm 2: $Rounding_{Zero}(LSB, G, R, Sy)$
return 0

Algorithm 3: $Rounding_{Infinity}(LSB, G, R, Sy)$
if ((G = 1) or (R = 1) or (Sy = 1)) return 1
else return 0

1.2 수식 및 정의

A와 B를 길이가 n인 부동 소수점 수의 분수부 라고 하자. 이때, 앞으로 나올 수식을 간단하게 하기 위하여, 소수점을 LSB와 G비트 사이에 놓도록 하겠다. 그러면 G, R, Sy 비트 위치는 소수점 이하의 수가 되고, LSB를 포함한 상위 비트들은 소수점 이상의 수가 된다. 이때, 소수점 이상의 수에는 아래첨자로 I를 붙치고, 소수점 이하의 수에는 아래첨자로 T를 붙치겠다. (그림 1)은 F의 상위 (n+1)가 소수점 이상의 수이고 G, R, Sy 비트 위치는 소수점 이하의 수를 나타내는 예이다.

$$F = \overbrace{f_{2n-1}f_{2n-2}\cdots f_{n-1}}^{F_I} \cdot \overbrace{f_{n-2}\cdots f_1 f_0}^{F_T}$$

(그림 1) F_I과 F_T의 정의
(Fig. 1) The definitions of F_I and F_T

앞으로 전개되는 수식을 간단히 하기 위해서, 논리곱 연산을 '∧'으로, 논리합 연산을 '∨'으로, 배타적 논리합(exclusive-or)를 '⊕'으로 표시하겠다. X는 don't care condition을 나타낸다. 만약 Z 연산의 결과 오버플로우(overflow)가 발생하면 overflow(Z)는 1을 돌려주고, 오버플로우(overflow)가 발생하지 않으면 overflow(Z)는 0을 돌려준다. C_k^{in}는 (k−1) 번째의 비트 위치에서 k 번째 비트 위치로의 캐리(carry) 값을 나타낸다.

2. 부동 소수점 덧셈/뺄셈 연산기

이번 장에서는 기존의 부동 소수점 덧셈/뺄셈 연산기의 처리 알고리즘에 대해서 분석하고 이를 토대로 새로운 처리 알고리즘을 갖는 부동 소수점 덧셈/뺄셈 연산기를 제시하겠다.

2.1 관련 연구

일반적인 부동 소수점 덧셈/뺄셈 연산은 지수 정렬(alignment), 분수부 덧셈/뺄셈 연산(addition/subtraction), 정규화(normalization), 반올림(rounding)의 4 단계로 이루어진다[1, 3, 5]. 이런 구조의 연산부에는 반올림 처리를 위해 별도의 덧셈기를 필요로 한다. 또한, 반올림 단계에서 반올림시 오버플로우(overflow)로 인한 재정규화(renormalization)가 발생할 수 있다. 이런한 단점을 극복하기 위해서 여러가지 처리 방법이 제안되어졌다.

첫번째로, 지수부의 차이에 따라서 지수 정렬 단계 혹은 정규화 단계가 간소화 되는 특징을 이용한 방법이다. [6]에서는 두 지수부의 차의 절대값에 따라서 부동 소수점 덧셈/뺄셈 연산의 처리 알고리즘이 두개의 경로를 통해 수행될수 있다는 것을 발표하였다. 두 지수부의 차이가 1 보다 작거나 같을 때, 지수 정렬부는 분수부 전체에 대한 고성능의 쉬프트(shift) 대신 단순한 멀티플렉싱(multiplexing)으로 바꿀 수가 있다. 반면에 두 지수부의 차이가 1 보다 클때, 정규화 과정에서 필요한 분수부 전체에 대한 고성능의 쉬프트는 단순한 멀티플렉싱(multiplexing)으로 바꿀 수가 있다. 따라서, 부동 소수점 덧셈/뺄셈 연산의 최장 경로(critical path)는 지수정렬, 덧셈/뺄셈, 반올림 혹은 덧셈/뺄셈, 정규화, 반올림 중에 한개가 된다. 이러한 처리 알고리즘을 갖는 부동 소수점 덧셈/뺄셈 연산기는 [7, 8]에서 사용되었다. 위의 알고리즘은 분수부 전체에 대한 쉬프트를 1 개로 줄였다. 그러나, 반올림 단계가 여전히 존재하여 이에 대한 하드웨어와 처리 시간을 필요로 하며, 두가지 경로가 존재함으로써 부가적인 하드웨어가 필요하고, 전형적인 부동 소수점 덧셈/뺄셈 연산기에 비해 구현시 복잡하고 어렵다.

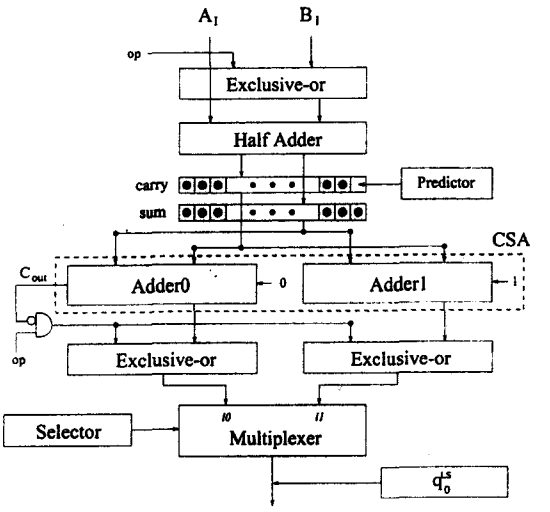
두번째로, 반올림과 덧셈/뺄셈 연산을 동시에 수행하는 방법이 제안되었다[9]. 덧셈/뺄셈 연산과 반올림의 특징을 분석하여서, 성능에 거이 영향을 미치지 않고 간단한 하드웨어를 첨가하여 덧셈/뺄셈 연산을 동시에 수행 가능하도록 하였다. 따라서, 반올림을 위한 고속 덧셈기를 제거하였고, 반올림이 정규화 전에 수행되기 때문에 재정규화도 제거 하였다. 따라서, 전체 하드웨어를 줄이고 부동 소수점 덧셈/뺄셈 연산을

3단계로 처리할 수 있어서 칩 차지 면적과 성능 면에서 매우 우수하다. 그런데, [9]에서는 덧셈인 경우 round-to-infinity mode시 지수 정렬 단계에서 발생되는 Sy 비트가 최장경로에 포함되기 때문에 Sy 비트가 성능에 영향을 미친다. Sy 비트는 지수 정렬부에서 발생하며 지수부의 차이에 의해서 버려지는 분수부중 R 비트 보다 낮은 비트를 전부 논리합을 수행한 결과값이다. 이번 장에서는 Sy 비트가 성능에 영향을 주지 않는 방법을 제안하겠다.

2.2 하드웨어 모델

[9]에서 제시된 분수부 덧셈/뺄셈과 반올림을 동시에 수행할 수 있는 하드웨어 모델이 (그림 2)와 같다. (그림 2)는 부동 소수점 덧셈/뺄셈 연산부중 지수정렬부와 정규화부분은 기존의 부동 소수점 덧셈/뺄셈 연산기와 동일하기 때문에 생략하였다. 제시된 하드웨어 모델은 분수부 덧셈/뺄셈과 반올림을 동시에 수행할 때의 특성에 대한 수식적인 분석을 통하여 얻어졌다. [9]에서는 제시된 하드웨어 모델을 각각의 반올림 모드에 대한 구현에 관하여 논하였다.

(그림 2)는 n 비트의 반가산기와 predictor와 분수부의 덧셈을 위한 고성능 덧셈기와 n 비트의 멀티플렉서와 selector와 q_0^{LS}와 n 비트의 배타적논리합 연산기로 이루어졌다. 부동 소수점 덧셈/뺄셈 연산의 특성으로 인하여, 분수부 덧셈을 위해서 고성능 덧셈기로는 거의 대부분이 올림수선택덧셈기(carry select adder)를 사용한다. 본 논문에서 제시한 부동 소수점 덧셈/뺄셈 연산기에서도 올림수선택덧셈기를 사용하였으며, (그림 2)에서 adder0와 adder1으로 나타낸 것이다. adder0와 adder1의 역할은 한개의 올림수선택덧셈기로 구현되어진다. 분수부 덧셈/뺄셈과 반올림을 동시에 수행하기 위하여 기존의 부동 소수점 덧셈/뺄셈 연산기에 추가된 하드웨어는 n 비트의 반가산기와 predictor와 q_0^{LS}이고 selector는 기존의 부동 소수점 덧셈/뺄셈 연산기에 비하여 약간 복잡하나 전체 성능에는 영향을 미치지 않는다. 반면에, 반올림을 덧셈/뺄셈과 동시에 수행할수 있으므로, 반올림을 위한 고성능의 덧셈기와 재정규화를 위한 하드웨어도 필요가 없다. 따라서, 성능 면에서나 차지면적에서 기존의 부동 소수점 덧셈/뺄셈 연산기에 비하여 매우 효율적이다. 그런데, 덧셈인 경우 round-to-infinity mode시 Sy 비트가 predictor의 입력으로 되기 때문에 Sy 비트의 발생이 최장경로에 포함된다.



(그림 2) IEEE 반올림과 덧셈/뺄셈을 병렬로 수행하는 하드웨어 모델

(Fig. 2) Hardware model performing IEEE rounding and addition/subtraction in parallel

2.3 제안하는 부동 소수점 덧셈/뺄셈 연산기

이번 절에서는 덧셈인 경우 round-to-infinity mode시 지수 정렬 단계에서 발생되는 Sy 비트가 최장경로에 포함되지 않는 방법에 대하여 논한다. 제안하는 부동 소수점 덧셈/뺄셈 연산기는 (그림 2)의 하드웨어 모델을 그대로 이용한다.

부동 소수점 덧셈/뺄셈 연산기에서 분수부의 덧셈/뺄셈을 수행하기 위해 먼저 두 지수부를 같은 값을 갖도록 맞추어야 한다. 이때, 올바른 반올림을 위해서 분수부의 데이타 손실에 대한 정보인 G, R, Sy 비트가 발생한다. 이때 두 피연산자중 지수부가 크거나 같은 쪽의 분수부를 B라 하고, 지수부가 작거나 같은 쪽의 분수부를 A라 한다. 분수부 A는 지수정렬시 G, R, Sy가 생성됨으로 $A = a_{n-1}a_{n-2}\cdots a_0.GRSy$이라 하겠고, $B = b_{n-1}b_{n-2}\cdots b_0$이라 하겠다. 분수부 덧셈의 결과를 $F = A + B = C_F f_{n-1}f_{n-2}\cdots f_0.GRSy$이라 하겠다. 여기서, C_F는 A + B 결과로 인한 오버플로우 비트(overflow bit)이다. 이때, 분수부 덧셈의 결과는 다

음 두가지 경우가 발생한다. 즉, 덧셈의 결과 오버플로우가 발생하거나, 발생하지 않을 경우이다. 따라서, 오버플로우 여부에 따라서 정규화시 오른쪽으로 1 비트의 쉬프트이거나 쉬프트를 필요로 하지 않는다.

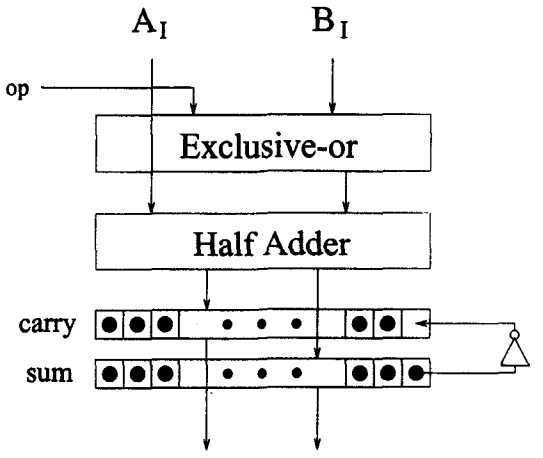
덧셈 연산후 정규화시 오른쪽으로 1 비트의 쉬프트가 필요한 경우를 RS(Right Shift)라고 하고 쉬프트가 필요하지 않은경우를 NS(No Shift)라 하자. 정규화를 위한 쉬프트를 거치지 않고 반올림을 수행한 후의 결과를 $Q=q_{n-2}\cdots q_0$라고 하면 NS시의 Q의 값은 다음과 같다.

$$Q^{NS}=f_{n-1}f_{n-2}\cdots f_0)+Rounding(f_0, G, R, Sy)$$
$$=F_I+Rounding(f_0, G, R, Sy) \quad (1)$$

한편, RS시의 Q의 값은 다음과 같다.

$$Q^{RS}=((C_Ff_{n-1}f_{n-2}\cdots f_1)+Rounding(f_1, f_0, G, R\vee Sy))\times 2$$
$$=((C_Ff_{n-1}f_{n-2}\cdots f_1, f_0)+2\times Rounding(f_1, f_0, G, R\vee Sy))\times 2$$
$$=F_I+2\times Rounding(f_1, f_0, G, R\vee Sy) \quad (2)$$

위의 식 (1)과 (2)에 의해 덧셈과 반올림을 동시에 수행하기 위하여, F_I, F_I+1, F_I+2를 해당 조건에 따라서 생성해야 됨을 알수 있다. 이를 생성하기 위하여 (그림 2)에서의 predictor와 selector를 적절히 선정하여야 한다.



(그림 3) predictor의 로직
(Fig. 3) The logic for predictor

그런데, [9]에서는 덧셈인 경우 round-to-infinity mode시 Sy 비트가 predictor의 입력으로 되기 때문에 Sy 비트의 발생이 최장 경로에 포함된다. 이를 해결하기 위해서, predictor를 (그림 3)에서 보는바와 같이 반가산기를 거쳐서 생성된 합(sum)의 LSB에 반전을 수행한 값으로 선정하였다. 반가산기를 거쳐서 생성된 합의 LSB를 sum_{LSB}라 하자. 이때, NS와 RS의 여부와 sum_{LSB} 값에 따라 4가지의 경우가 발생한다.

먼저 RS이고 $sum_{LSB}=0$일 경우에는 predictor=1이 되고, (그림 2)의 올림수선택덧셈기에서 덧셈을 수행한후 멀티플렉서의 i0와 i1에 입력되는 값은 F_I+predictor와 F_I+predictor+1임으로 F_I+1과 F_I+2에 해당한다. 이는, F_I의 LSB인 sum_{LSB}가 0 임으로, $C_Ff_{n-1}f_{n-2}\cdots f_1+0$와 $C_Ff_{n-1}f_{n-2}\cdots f_1+1$에 해당한다. 따라서, 이경우에는 반올림의 결과 올림이면 i1를 선택하고 반올림의 결과가 버림이면 i0를 선택하면 된다. 그리고, 임으로 유효숫자가 상위 n 비트임으로, RS시에는 덧셈 결과의 LSB는 반올림에만 영향을 미치며 버려지는 값이다. RS이고 $sum_{LSB}=1$일 경우에는 predictor=0가 되고, 올림수선택덧셈기에서 덧셈을 수행한 후 멀티플렉서에 입력되는 값은 F_I+0과 F_I+1에 해당한다. 그런데, $sum_{LSB}=1$임으로, F_I+1은 $C_Ff_{n-1}f_{n-2}\cdots f_1+1$에 해당한다. 따라서, 이런 경우에도 역시 반올림의 결과 올림이면 i1를 선택하고 반올림의 결과가 버림이면 i0를 선택하면 된다. 따라서, RS의 경우를 종합하면 selector는 반올림의 결과인 $Rounding(f_1, f_0, G, R\vee Sy)$가 된다.

NS일때는 반올림의 결과 버림이면 F_I+0을 선택하면 되고, 반올림의 결과가 올림이면 F_I+1를 선택하면 된다. 이때, $sum_{LSB}=0$일 경우 i0와 i1에 입력되는 값은 F_I+1과 F_I+2에 해당한다. 따라서, 반올림의 결과가 올림이면 i0를 선택하면 된다. 그런데, $sum_{LSB}=0$임으로 F_I+0과 F_I+1의 결과값의 상위 n−1 비트는 동일하다. 따라서, 반올림의 결과가 버림일 경우에도 i0를 선택하고 덧셈 결과의 LSB를 0으로 만들어 주면 된다. 따라서, NS시 $sum_{LSB}=0$일 경우 selector는 반올림의 결과와 상관없이 i0이고, 덧셈결과의 LSB는 반올림의 결과가 버림일 경우 0으로 되게 하면 된다. $sum_{LSB}=1$일 경우에는 올림수선택덧셈기에서 덧셈을 수행한후 i0와 i1에 입력되는 값은 F_I+0과 F_I+1에 해당한다. 따라서, 이경우에는 반올림의 결과 올림

이면 i1를 선택하고 반올림의 결과가 버림이면 i0를 선택하면 된다.

2.4 성능 비교

일반적인 처리과정을 수행하는 부동 소수점 덧셈/뺄셈 연산기와 [9]와 제안하는 처리과정을 비교하기 위하여 전세계적으로 널리 쓰이는 디자인 오토메이션 툴인 COMPASS를 이용하여 0.8 μ 공정에서 합성 및 시뮬레이션을 수행 하였다. 그 결과 일반적인 처리과정을 수행하는 부동 소수점 덧셈/뺄셈 연산기에 비해 [9]은 약 27%와 제안하는 처리과정은 약 31% 정도의 성능이 좋아졌다. 차지면적 또한 일반적인 처리과정을 수행하는 부동 소수점 덧셈/뺄셈 연산기에 비해 21% 정도 덜 차지하는 것으로 나타났다. 따라서, 제안하는 부동 소수점 덧셈/뺄셈 연산기는 성능과 차지 면적 면에서 효율적이다.

3. 부동 소수점 곱셈및 나눗셈 연산기의 반올림부

일반적인 부동 소수점 곱셈 연산기의 분수부 처리 단계는 곱셈, 덧셈, 반올림, 정규화로 구성되며, 부동 소수점 나눗셈 연산기의 분수부 처리 단계는 정수형 나눗셈 연산, 몫 변환, 반올림, 정규화의 단계로 이루어 진다. 정수형 나눗셈 연산은 대부분 SRT 반복을 사용하며, 여러 사이클(cycle) 후에 결과로 몫과 나머지가 중복(redundant)적으로 발생한다. 즉, 몫은 음수 부분과 양수 부분이 생성되고, 나머지는 캐리 부분과 합 부분이 발생한다. 그후, 몫변환 단계에서는 몫의 음수부분, 양수부분, 나머지의 부호를 가지고 이진 표현으로 몫을 생성해 낸다. 이 결과를 가지고 반올림과 정규화를 수행한다.

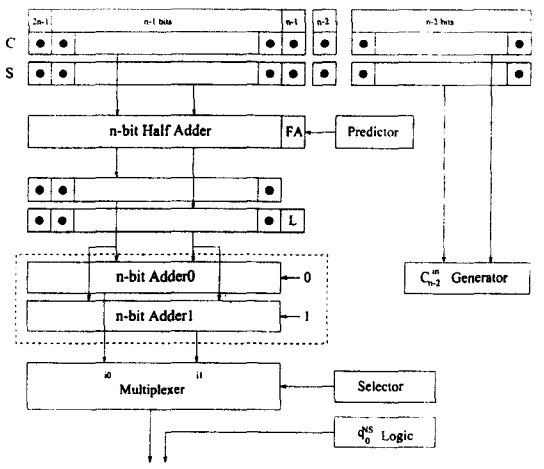
이와 같은 부동 소수점 나눗셈 연산기의 몫변환, 반올림, 정규화 부분은 설계의 효율을 위하여 부동 소수점 곱셈 연산기의 하드웨어를 공유하게 된다. 그 이유는 첫째, 부동 소수점 나눗셈 연산기는 많은 사이클이 필요로 하여 파이프라인을 못한다. 또한, 여러 응용 프로그램에서 사용되는 빈도가 다른 부동 소수점 연산에 비해 상대적으로 적어서, 몫변환, 반올림, 정규화를 위한 별도의 하드웨어를 만들기에는 가격대 성능비가 떨어진다. 둘째, 몫변환, 반올림, 정규화를 수행하는 부동 소수점 나눗셈기의 처리 과정과 덧셈, 반올림, 정규화를 수행하는 부동 소수점 곱셈 연산기의 처리 과정이 서로 유사하다. 그런데, 하드웨어를 공유할 경우 부동 소수점 나눗셈 연산으로 인하여 부동 소수점 곱셈 연산기의 하드웨어를 몇 사이클 동안 사용하게 된다. 따라서, 부동 소수점 곱셈 연산과 나눗셈 연산에 대한 콘드롤 로직이 복잡해질수 있고, 부동 소수점 곱셈 연산기의 최장경로에 영향을 줄 수 있다. 또한, 부동 소수점 곱셈 연산기가 부동 소수점 나눗셈 연산기에 비해 쓰이는 빈도수가 많고 가격이 높기 때문에, 하드웨어의 효율적인 공유가 요구된다.

본 논문에서는 몫변환과 반올림을 동시에 수행하고 기존에 부동 소수점 곱셈 연산기의 하드웨어와 효율적으로 공유하는 부동 소수점 곱셈 및 나눗셈 연산기의 반올림 처리부를 제안하였다. 제안하는 반올림 처리부는 부동 소수점 곱셈 연산시 덧셈, 반올림, 정규화를 부동 소수점 나눗셈 연산시 몫 변환, 반올림, 정규화를 1 사이클에 처리할수 있다. [12]에서는 덧셈과 반올림을 동시에 수행하는 부동 소수점 곱셈 연산기를 제안하였다. 제안하는 부동 소수점 나눗셈 연산기는 [12]의 하드웨어를 공유하도록 하였다.

이번 장에서는 본 논문에서 채택한 부동 소수점 곱셈 연산기의 하드웨어 모델을 살펴보고, 분수부 나눗셈 처리 알고리즘인 SRT 방식을 분석하고 부동 소수점 나눗셈 연산기의 새로운 반올림 방식을 제시하겠다.

3.1 부동 소수점 곱셈 연산기의 반올림부

부동 소수점 곱셈기의 성능을 높이기 위해서 여러 가지 기법이 제안되었는데, 그중 덧셈과 반올림을 동시에 할수 있는 기법이 제안 되었다[11, 12]. 이와 같은 기법은, 반올림시 소요되는 덧셈기를 없애주고, 수행 속도가 빨라진다. 그러나, [11]는 처리 알고리즘이 복잡하여 덧셈과 반올림시 2 개의 파이프라인 단계를 차지하며, 전체 하드웨어가 복잡하다. [12]에서 제시한 부동 소수점 곱셈 연산기는 하드웨어 적으로 간단하고 덧셈과 반올림을 1 개의 파이프라인 단계로 처리 가능하다. [12]의 하드웨어 모델은 (그림 4)에 나타나 있다. 제안하는 부동 소수점 나눗셈 연산기는 [12]에서 제안한 부동 소수점 곱셈 연산기의 하드웨어를 공유한다.



(그림 4) 부동 소수점 곱셈연산기의 덧셈과 반올림 단계
(Fig. 4) The addition and rounding stage for floating point multiplier

3.2 SRT 반복법[10]

정수형 나눗셈 연산은 다음 수식으로 나타낼수 있다.

$$x = q \times d + rem,\ |rem| < |d| \times ulp$$

여기서, dividend x와 divisor d는 입력되는 피연산자들이며, 최종 몫인 q와 최종 나머지 rem은 나눗셈 연산의 최종 결과 값이다. ulp(unit in the last position)는 몫의 precision을 말하는 것으로써, n-digit에 대한 radix-r의 분수부 결과의 ulp는 r^{-n}이다.

나눗셈을 수행하기 위해서는 다음의 반복을 수행하여야 한다.

$$rP_0 = x,\ P_{j+1} = rP_j - dq_{j+1}$$

여기서, q_{j+1}는 j +1번째 부분 몫 부분이고 P_j는 j 번째 반복후에 부분 나머지의 값이다.

현재 반복에 대해서 다음 반복후 부분 나머지가 bound 될려면 부분 몫은 $|P_{j+1}| \le d$의 조건을 만족하도록 선택해야 한다. 나눗셈 연산의 종료후 최종 몫은 각각의 반복마다 구해진 부분 몫을 모두 더함으로써 얻어진다.

$$Q_{final} = \sum_{j=1}^{n} q_j \times r^{-j}$$

일반적으로 반복의 성능을 높이기 위해서 부분 나머지와 부분 몫은 중복적인 형태로 표현된다. 즉, 부분 나머지는 올림수 부분과 합 부분으로 구성이 되며, 몫 부분은 양수 부분과 음수 부분으로 구성된다.

Radix-r시 SRT 방식으로 l번 반복 후 $(l \cdot \log_2 r)$비트의 몫이 구해진다. 그리고 반올림과 정규화를 위해 총 (n +2)비트의 몫이 필요하므로 SRT 반복이 끝나기 위해서 식 (3)을 만족해야한다.

$$l \cdot \log_2 r \ge n + 2,\ l \cdot \log_2 r = (n+2) + t \qquad (3)$$

여기서, $0 \le t < \log_2 r$이다. 이때 l번 반복후 (n +2)비트의 몫과 여분의 하위 t 비트의 몫을 더 얻게 된다. 여기서 l번째 반복에서 얻어지는 부분몫을 q_l, 부분나머지를 P_l이라 하자. 올바른 몫을 얻기위해서는 P_l의 부호가 양수여야 하므로 만약 P_l의 부호가 음수이면 나머지 P_l에 대한 복구(restoration)가 필요하다. 따라서, 복구후의 q_l과 P_l인 $\widetilde{q_l}$와 $\widetilde{P_l}$는 식(4)로 표현될수 있다.

$$\widetilde{q_l} = q_l - restore_l,\quad \widetilde{P_l} = P_l + restore_l \cdot div \qquad (4)$$

$$restore_l = \begin{cases} 1 & \text{if } P_l < 0 \\ 0 & \text{if } P_l \ge 0 \end{cases}$$

3.3 제안하는 부동 소수점 나눗셈 연산기의 반올림 처리부

이번 절에서는 반올림과 몫변환을 동시에 수행할 때의 특성을 분석하고, 이를 토대로 하드웨어 모델과 구현 방안을 제시하고, 제안하는 부동 소수점 나눗셈 연산기의 반올림부를 채택할 경우 기존의 부동 소수점 나눗셈 연산기와 수행 사이클을 비교하였다.

3.3.1 IEEE-754 표준에 따른 몫의 반올림에 대한 Analysis

SRT 반복에 의해 얻어진 몫과 나머지는 중복이진 표현된 값이다. 즉, 최종 몫은 두개의 값 Q_P와 Q_N의 합으로 얻어진다. 식(3)에 의해 몫 Q_P와 Q_N의 크기는 (n +2 +t) 비트이고 Q_N은 1의 보수(1's complement)의 음수이다. Q_P와 Q_N는 다음 식(5)와 같이 정의 한다.

$$Q_P = p_n \cdots p_0 . p_{-1} \cdots p_{-(t+1)},\quad Q_N = n_n \cdots n_0 . n_{-1} \cdots n_{-(t+1)} \qquad (5)$$

나머지는 몫의 $(-(t+1))$번째 위치의 나머지로 sum과 carry의 형태로 표현된 중복이진표현이며 나머지의 sum을 $Rem^{S}_{-(t+1)}$이라 하고 carry를 $Rem^{C}_{-(t+1)}$이라한다. $Rem^{S}_{-(t+1)}$과 $Rem^{C}_{-(t+1)}$는 2의 보수(2's complement)이며 크기는 $(n+1)$비트이다. 따라서, $Rem^{S}_{-(t+1)}$과 $Rem^{C}_{-(t+1)}$는 다음 식(6)과 같이 정의 한다.

$$Rem^{S}_{-(t+1)} = s_n \cdots s_1 s_0, \quad Rem^{C}_{-(t+1)} = c_n \cdots c_1 c_0 \tag{6}$$

Q_P의 정수부분을 P_I, Q_N의 정수부분을 N_I라 하고 몫의 $-(t+1)$번째 위치의 나머지 $Rem_{-(t+1)}$이라 하면 다음과 같이 정의한다.

$$P_I = p_n \cdots p_0, \quad N_I = n_n \cdots n_0$$
$$Rem_{-(t+1)} = Rem^{S}_{-(t+1)} + Rem^{C}_{-(t+1)} \tag{7}$$

그러면 Q_N이 1의 보수의 음수이므로 2의 보수의 몫을 구하기 위해서는 Q_N의 R-bit 위치인 n_{-1}의 위치에 1을 더해주어야만 한다. 그리고 나머지의 부호가 음수인경우에 복구를 하여야 하므로 Sy 비트를 포함한 상위 $(n+2)$비트의 몫 H를 식(8)로 쓸 수 있다.

$$\begin{aligned} H &= h_n h_{n-1} h_0 . h_{-1} Sy \\ &= (p_n \cdots p_0) + (n_n \cdots n_0) + 0.p_{-1} + 0.n_{-1} \\ &\quad + 0.1 - restore_{-1} + 0.0Sy \\ &= P_I + N_I + 0.p_{-1} + 0.n_{-1} + 0.\overline{restore_{-1}} + 0.0Sy \end{aligned}$$

$$\begin{aligned} restore_{-1} = \overline{overflow}((p_{-2} \cdots p_{-(t+1)}) + \\ (n_{-2} \cdots n_{-(t+1)}) + 1 - restore_{-(t+1)}) \end{aligned}$$

$$\begin{aligned} Sy &= 0 \quad \text{if} \quad Rem_{-(t+1)} = 0 \quad \text{and} \quad (h_{-2} \cdots h_{-(t+1)}) = 0 \\ &= 1 \quad \text{if} \quad Rem_{-(t+1)} = 1 \quad \text{or} \quad (h_{-2} \cdots h_{-(t+1)}) = 1 \end{aligned} \tag{8}$$

$restore_{-1}$은 몫의 (-1)번째 비트 위치에 대한 복구이다. 몫의 하위 t비트는 $restore_{-(t+1)}$과 함께 (-1)번째 위치에 대한 복구인 $restore_{-1}$을 생성하기 위해 사용된다. $restore_{-(t+1)}$는 몫의 $-(t+1)$번째에 대한 복구며 나머지의 부호비트의 값과 동일하다.

$D_I = d_n \cdots d_0$를 $P_I + N_I$로 정의하면, 식(8)를 다음 식(9)로 쓸 수 있다.

$$\begin{aligned} H &= p_{-1} \oplus n_{-1} \oplus \overline{restore_{-1}} \\ C_0^{in} &= overflow(p_{-1} + n_{-1} + \overline{restore_{-1}}) \end{aligned} \tag{9}$$

여기서 R은 라운드 비트이고 C_0^{in}는 0번째 비트위치로의 올림수이다. 따라서, H의 정수부분인 H_I는 다음과 같이 쓸 수 있다.

$$H_I = P_I + N_I + C_0^{in} = D_I + C_0^{in} \tag{10}$$

정규화는 H의 MSB의 값에 따라 다음의 두가지 경우로 나눌 수 있다. $h_n = 1$인 경우 정규화시 쉬프트(shift)가 필요하지 않으므로 NS(No Shift)라고 한다. $h_n = 0$인 경우 정규화시 1비트 왼쪽으로 쉬프트(left shift)를 필요하고 이것을 LS(Left Shift)라 한다. 정규화 과정은 NS와 LS로 나눌수 있다. LS의 경우 H에 대한 정규화 후에 결과 값을 O^{LS}라 하고, NS의 경우 H에 대한 정규화 후에 결과 값을 O^{NS}라 하자. O_I^{LS}와 O_T^{LS}는 다음 식 (11)과 같다.

$$O_I^{LS} = h_{n-1} h_{n-2} \cdots h_0, \quad O_T^{LS} = RSy \tag{11}$$

O_I^{NS}와 O_I^{NS}는 다음 식 (12)과 같다.

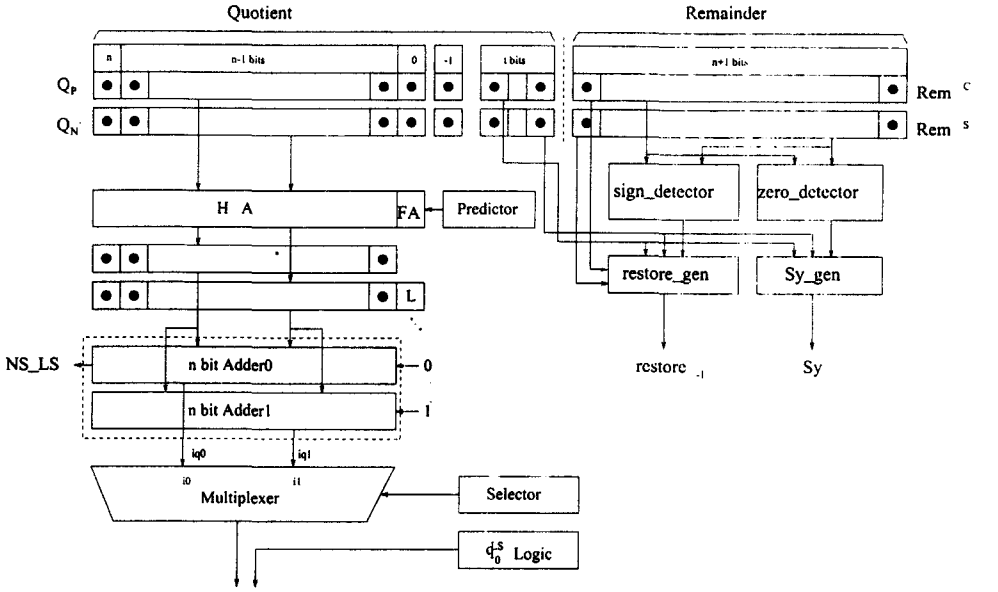
$$O_I^{NS} = h_n h_{n-1} \cdots h_1, \quad O_T^{NS} = h_0(R \vee Sy) \tag{12}$$

Q를 F에 대해서 정규화 전에 반올림을 수행한 값이라 하자. 그리고, NS시 Q의 값은 Q^{NS}이고 LS시 Q의 값은 Q^{LS}이다. 또한, $Rounding_{mode}(f_{n-1}, R, Sy)$는 해당 반올림 모드에서 반올림의 결과를 나타낸 것이다. 이때, NS시 Q^{NS}는 식 (10)과 (11)를 적용하면 다음과 같다.

$$\begin{aligned} Q^{LS} &= O_I^{NS} + Round_{mode}(h_0, R, Sy) \\ &= D_I + C_0^{in} + Round_{mode}(h_0, R, Sy) \end{aligned} \tag{13}$$

또한, NS시 Q^{NS}는 식 (10)과 (12)를 적용하면 다음과 같다.

$$\begin{aligned} Q^{NS} &= (O_I^{NS} + Round_{mode}(h_1, h_0, (R \vee Sy))) \times 2 \\ &= (h_n \cdots h_0) + 2 \times Round_{mode}(h_1, h_0, (R \vee Sy)) \\ &= D_I + C_0^{in} + 2 \times Round_{mode}(h_1, h_0, (R \vee Sy)) \end{aligned}$$



(그림 5) 몫변환과 반올림을 동시에 수행하는 하드웨어 모델
(Fig. 5) Hardware model performing quotient conversion and rounding in parallel

3.3.2 몫변환과 반올림을 동시에 수행하는 하드웨어 모델

몫변환과 반올림을 동시에 수행하는 반올림 처리 하드웨어 모델은 (그림 5)에 제시되어 있다. 이는 n비트의 반가산기(half adder)와 1비트의 전가산기(full adder), n 비트의 올림수선택덧셈기(carry select adder), predictor, selector, $restore_{-1}$ generater, q_0^{LS} generater, Sy bit generater로 구성되어 있다. (그림 5)는 (그림 4)에 몇개의 하드웨어를 추가하여 구현할 수 있음을 알 수 있다.

sign_detector는 식 (8)의 $Rem_{-(t+1)}$을 생성하는 logic이고, restore_gen은 식 (8)의 $restore_{-1}$을 생성하는 logic이다. 이들은 (그림 4)의 C_0^{in} genterator에 약간의 logic을 추가하여 구현할 수 있다. zero_detector는 나머지가 0임을 결정하는 logic이고, Sy_gen은 식 (8)에서 보는 바와 같이 zero_detector의 신호와 하위 몫으로 부터 Sy를 구한다.

n비트의 반가산기와 1비트의 전가산기는 Q_P와 Q_N의 상위 (n+1)비트를 더하여 (n+1)비트 크기의 sum과 n비트 크기의 carry를 생성한다. 이때, predictor는 전가산기의 입력이 되어서 더해진다. 따라서, 반가산기와 1비트의 전가산기에 의한 덧셈의 결과는 D_I + predictor가 된다. 이때 sum의 LSB는 L이라 한다. 상위 n비트의 sum과 carry는 다시 올림수선택덧셈기에서 더하여져 iq0와 iq1를 생성한다. 올림수선택덧셈기는 그림에서 점선으로 된 박스이고 Adder0는 LSB로의 올림수가 0인 덧셈기이며 덧셈결과는 iq0이고 그 값은 D_I +predictor이다. Adder1은 LSB로의 올림수가 1인 덧셈기이며 덧셈 결과는 iq1이고 그 값은 D_I +predictor +2이다. selector는 iq0와 iq1중 몫변환과 반올림 처리된 결과인 몫 Q를 선택한다.

선택된 결과 몫 Q는 D_I +predictor나 D_I +predictor +2중 한 값을 가지게 된다. predictor의 선택에 따라 결국 결과는 D_I +0, D_I +1, D_I +2, D_I +3의 값을 가질 수 있다. 따라서 식(13)와 식(14)에서 필요한 D_I +0, D_I +1, D_I +2, D_I +3를 모두 생성할 수 있다. 이때 선택된 결과는 n비트이므로 정규화시 LS의 경우 Q^{LS}의 LSB가 필요하다. q_0^{LS} generator logic에서 Q^{LS}의 LSB를 생성한다.

predictor는 반가산을 하기전에 이미 정하여진 입력

변수를 사용하여야 성능에 영향을 미치지 않는다. 여기서는 n_{-1} bit와 p_{-1} bit를 사용하였다. Adder0로부터 나온 결과인 iq0를 $E=e_n e_{n-1}\cdots e_1$이라고 하자. E의 LSB가 H의 1번째 비트위치에 해당하므로 실제 정수값 E_I은 $E\times 2$가 된다. 그러므로 P_I+N_I+predictor를 E_I^*라고 하면 $E_I^*=E_I+L$이다. 즉,

$$E_I^*=E_I+L=P_I+N_I+predictor=e_n\cdots e_1 L \qquad (15)$$

다음 절부터는 NS와 LS시의 H에 대한 반올림위치를 IEEE 754 표준의 각각의 반올림 모드에 대하여 수학적으로 분석을 하고 predictor와 selector를 결정하도록 한다.

3.3.3 Round-to-nearest Mode

Round-to-nearest시 LS의 경우 식 (13)에 의해 Q^{LS}는 D_I, D_I+1, D_I+2중 한개의 값을 가질수 있다. NS의 경우는 $Round_{Nearest}(h_1, h_0, (R\vee Sy))$가 올림이 되기 위해서는 Algorithm 3에 의해 $h_0=1$이 되어야 한다. 그런데, NS시 반올림시 올림을 위해 h_1 자리에 1을 더해주는 것은, 반올림의 결과가 올림시 $h_0=1$임으로 h_0 자리에 1을 더해주는 것과 결과가 같다. 따라서, NS의 경우도 Q^{NS}는 D_I, D_I+1, D_I+2중 한개의 값을 가질수 있다. 여기서는, selector를 간단하게 하기위해서 predictor는 다음과 같이 주어진다.

$$predictor=p_{-1}\wedge n_{-1}$$

위의 predictor에 의해 식(9)는 식(16)가 된다.

$$\begin{aligned}H&=P_I+N_I+0.p_{-1}+0.n_{-1}+0.\overline{restore_{-1}}+0.0Sy\\&=P_I+N_I+overflow(p_{-1}+n_{-1})+0.(p_{-1}\oplus n_{-1})+\\&\quad 0.\overline{restore_{-1}}+0.0Sy\\&=P_I+N_I+predictor+0.(p_{-1}\oplus n_{-1})+\\&\quad 0.\overline{restore_{-1}}+0.0Sy\\&=E_I+L+C_0^{in}+0.R+0.0Sy\\&=E_I+C_0^{in}\times 2+h_0+0.R+0.0Sy\end{aligned}$$

$$R=(p_{-1}\oplus n_{-1})\oplus\overline{restore_{-1}},\quad C_0^{in}=(p_{-1}\oplus n_{-1})\wedge\overline{restore_{-1}}$$

$$h_0=L\oplus C_0^{in},\quad C_1^{in}=L\wedge C_0^{in} \qquad (16)$$

LS의 경우 몫 Q^{LS}는 식(13)과 식(16)에 의해 다음 식(17)과 같이 주어진다.

$$Q^{LS}=E_I+C_1^{in}\times 2+h_0+Round_{nearest}(h_0, R, Sy)$$

이때, selector $=C_1^{in}\vee(h_0\wedge Round_{nearest}(h_0, R, Sy))$이고 $q_0^{LS}=h_0\oplus Round_{nearest}(h_0, R, Sy)$가 된다. NS의 경우 몫 Q^{NS}는 식(14)와 식(16)에 의해 다음 식(18)과 같이 쓸 수 있다.

$$Q^{NS}=E_I+2\times C_1^{in}+h_0+2\times Round_{nearest}(h_1, h_0, R\vee Sy)$$

따라서 selector $=C_1^{in}\vee Round_{nearest}(h_1, h_0, R\vee Sy)$이 된다.

3.3.4 Round-to-zero Mode

Round-to-zero의 predictor는 round-to-nearest의 predictor와 동일하게 놓는다. $Rounding_{Zero}(X, X, X)=0$임으로 NS시와 RS시 selector와 q_0^{LS}는 식 (17)과 (18)의 $Rounding_{nearest}$를 0으로 대치하여 적용하면 selector $=C_1^{in}$이고 $q_0^{LS}=h_0$이 된다.

3.3.5 Round-to-infinity mode

$L=p_{-1}\oplus n_{-1}=C_0^{in}=Sy=1$의 상황에서 round-to-nearest와 같은 predictor를 쓰는 경우, $C_0^{in}=1$, $h_0=0$, $R=0$, $Sy=1$이 된다. 따라서, Algorithm 1에 의하여 round-to-nearest의 경우 반올림의 결과가 버림이다. 그런데, round-to-infinity시에는 $Sy=1$ 임으로 Algorithm 3에 의하여 반올림의 결과가 올림이 된다. 따라서, round-to-infinity시에는 round-to-nearest의 predictor를 쓸 수가 없다. round-to-infinity일때 predictor를 다음 식으로 결정하였다.

$$predictor=(p_{-1}\vee n_{-1}) \qquad (19)$$

〈표 1〉은 p_{-1}, n_{-1}, $\overline{restore_{-1}}$에 따른 C_0^{in}, R, predictor의 결과를 나타낸 것이다. 〈표 1〉를 살펴보면 다음의 두가지 경우로 나누어 볼수 있다. 첫번째로, $\overline{restore_{-1}}=0$이고 $p_{-1}\oplus n_{-1}=1$인 경우로 h_0로의 올림수인 C_0^{in}이 0이고 R과 predictor가 1이 된다. 두번째는 그외의 경우로 C_0^{in}과 predictor의 값이 동일하다.

〈표 1〉 p_{-1}, n_{-1}, $\overline{restore^{-1}}$의 값에 따른 C_0^{in}, R, predictor의 결과

〈Table 1〉 The result values of C_0^{in}, R, predictor according to p_{-1}, n_{-1}, $\overline{restore^{-1}}$

p_{-1}	n_{-1}	$\overline{restore^{-1}}$	$p_{-1} \oplus n_{-1}$	C_0^{in}	R	Predictor
0	0	0	0	0	0	0
0	0	1	1	0	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	1
1	0	0	0	0	1	0
1	0	1	1	1	0	1
1	1	0	1	1	0	1
1	1	1	0	1	1	1

첫번째의 경우, R이 1임으로 Algorithm 3에 의해서 반올림시 올림이고 predictor가 1이다. 또한 〈표 1〉에 의해 C_0^{in}이 0이다. Q^{LS}는 식 (13)과 (15)에 의해 다음 식과 같다.

$$Q^{LS} = P_I + N_I + C_0^{in} + Round_{Infinity}(h_0, R, Sy)$$
$$= P_I + N_I + 1 = P_I + N_I + predictor = E_I + L$$

따라서, selector = 0이고 q_0^{LS} = L이다. Q^{NS}는 식 (13)과 (15)에 의해 다음 식과 같다.

$$Q^{NS} = P_I + N_I + C_0^{in} + 2 \times Round_{Infinity}(h_0, R, Sy)$$
$$= P_I + N_I + 2 = E_I + L + 1$$

따라서, 이때 selector = L이다.

두 번째의 경우는 C_0^{in}과 predictor의 값이 동일하다. 따라서, predictor는 C_0^{in}로 고쳐쓸 수 있다. LS의 경우 Q^{LS}은 다음과 같이 정리할 수 있다.

$$Q^{LS} = P_I + N_I + C_0^{in} + Round_{infinity}(h_0, R, Sy)$$
$$= E_I + L + Round_{infinity}(h_0, R, Sy)$$

이때 selector = $L \wedge Round_{infinity}(h_0, R, Sy)$이고 $q_0^{LS} = L \oplus Round_{infinity}(h_0, R, Sy)$이다.

NS의 경우 Q^{NS}은 다음과 같이 쓸 수 있다.

$$Q^{NS} = P_I + N_I + C_0^{in} + Round_{infinity}(h_1, h_0, R \vee Sy) \times 2$$
$$= E_I + L + Round_{infinity}(h_1, h_0, R \vee Sy) \times 2$$

이때 selector = $Round_{\infty}(h_1, h_0, R \vee Sy)$이 된다.

3.3.6 기존 부동 소수점 나눗셈 연산기와의 비교

〈표 2〉 배정밀도 부동 소수점 나눗셈 연산의 radix와 수행시간

〈Table 2〉 Radix and cycle time for double precision floating point divider

Processor	사이클수	radix
PowerPC604e[4]	31	4
PA-RISC[14]	31	4
Pentium[15]	33	4
Ultra Sparc[2]	22	8
R10000[1]	19	16
제안하는반올림처리모델	30	4
제안하는반올림처리모델	21	8
제안하는반올림처리모델	16	16

배정밀도 부동 소수점 나눗셈 연산의 경우 radix-4 SRT방법의 경우 정수나눗셈을 위한 SRT반복에 30사이클이 필요하다. 그리고 radix-8의 경우 21사이클, radix-16의 경우 16사이클이 각각 필요하다. 제안한 부동 소수점 나눗셈 연산기의 반올림 처리 하드웨어 모델은 몫 변환과 반올림을 동시에 수행하므로 이를 위해 1사이클의 수행시간이 필요하다. 그러므로 radix-4의 경우 제안한 반올림 처리 회로를 구현한 부동 소수점 나눗셈 연산기는 〈표 2〉와 같이 30사이클의 수행시간이 걸린다. 반면에 PowerPC604e[4]는 31사이클, PA-RISC[14]는 31사이클, Pentium[15]은 35사이클 시간이 필요하다. 그리고, radix-8의 경우 제안한 반올림 처리 회로를 갖는 부동 소수점 나눗셈 연산기는 수행시간이 21사이클이지만 UltraSparc[2]는 22사이클의 수행시간을 갖는다. 또한 radix-16의 경우 제안한 부동 소수점 나눗셈 연산기의 수행시간은 16사이클이지만 R10000[1]은 19사이클의 수행시간을 가진다. 결국 〈표 2〉에 제시된 기존 마이크로프로세서에 비해 수행시간 면에서 1-3사이클 빠른 장점이 있다. 최

근에 나오는 마이크로프로세서의 부동 소수점 나눗셈기의 사이클수가 짧아지는 추세로 볼 때 제안하는 부동 소수점 나눗셈 연산기의 반올림 로직을 채택하는 경우 상대적인 성능의 잇점이 계속 커지게 된다.

4. 결 론

일반적인 부동 소수점 연산기의 반올림 처리부는 반올림을 위하여 덧셈기가 필요하며 이를 수행하기 위한 별도의 수행시간이 요구된다. 본 논문에서는 부동 소수점 연산의 일부분인 덧셈과 반올림을 동시에 수행하는 부동 소수점 덧셈/뺄셈, 곱셈, 나눗셈 연산기에 대하여 논하였다. 제안하는 부동 소수점 덧셈/뺄셈, 곱셈, 나눗셈 연산기는 덧셈과 반올림을 하므로 반올림을 위한 덧셈기를 따로 필요로 않는다. 따라서, 처리 시간면과 차지 면적면에서 큰 잇점을 가지고 있다.

참 고 문 헌

[1] Kenneth C, Yeager, "The Mips R10000 Superscalar Microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28-40, Apr. 1996.

[2] Marc Tremblay and J. Michael O'Connor, "Ultra Sparc I @:A Four-Issue Processor Supporting Multimedia," *IEEE Micro*, vol. 16, no. 2, pp. 42-50, Apr. 1996.

[3] M.C. Becker et al, "The PowerPC 601 Microprocessor", *IEEE Micro*, vol. 13, no. 5, pp. 54-67, Oct. 1993.

[4] S. Peter Song, Marvin Denman and Joe Chang, "The PowerPC604 RISC Microprocessor," *IEEE Micro*, vol. 14, no. 5, pp. 8-17, Oct. 1994.

[5] J.L. Hennessy and D.A. Patterson, "Computer Architecture A Quantitative Approach," Morgan Kaufmann Publishers Inc, 1990.

[6] M. P. Farmwald, *On the Design of High Performance Digital Arithmetic Units*, PhD thesis, Stanford University, Aug. 1981.

[7] M. Birman, A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, and J. Barnes, "Developing the WTL3170/3171 sparc floating-point coprocessors," *IEEE Micro*, vol. 10, no. 1, pp. 55-64, Feb. 1990.

[8] B. J. Benschneider et al., "A Pipelined 50-Mhz CMOS 64-bit Floating-Point Arithmetic Processor," *IEEE Jounal of Solid-state Circuit*, vol. 24, no. 5, pp. 1317-1323, Oct. 1989.

[9] W. C. Park, S. W. Lee, O. Y. Kown, T. D. Han, and S. D. Kim, "Floating point adder/subtractor performing IEEE rounding and addition/subraction in parallel," *IEICE Trans. Information and Systems*, vol. e79-d, no. 4, pp. 297-305, April 1996.

[10] M. D. Ercegovac and T. Lang, "Division and Square Root:Digit recurrence Algorithms and Implementations," Kluwer Academic Publishers, 1994.

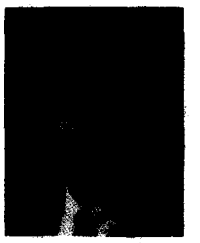
[11] J. Arjun Prabhu and Gregory B. Zyner, "167 MHz Radix-4 Floating Point Multiplier," *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pp. 149-154, July 1995.

[12] 박우찬, 정철호, 양진기, 한탁돈, "IEEE 반올림과 덧셈을 동시에 수행하는 부동 소수점 곱셈 연산기의 설계," 전자공학회 게재확정.

[13] IEEE Std 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic," *IEEE*, 1985.

[14] Craig Heikes and Glenn Colon-Bonet, "A Dual Floating Point Coprocessor with an FMAC Architecture," *ISSCC Digest of Technical Papers*, 1996.

[15] Intel, "Pentium Processor User's Manual," *Intel*, 1993.



박 우 찬

1993년 연세대학교 이과대학 전산과학과 졸업(학사)

1995년 연세대학교 대학원 전산과학과(이학석사)

1995년~현재 연세대학교 공과대학 컴퓨터과학과 박사 재학중

관심분야:ASIC 설계, 3차원 그래픽 가속기, Computer arithmetic, 고성능 컴퓨터 구조.

한 탁 돈

1978년 연세대학교 공과대학 전자공학과 졸업(학사)
1983년 Wayne State University 컴퓨터공학(공학석사)
1987년 University of Massachusetts 컴퓨터공학(공학박사)
1981년 금성전기 연구원
1987년~1989년 Cleveland 주립대학 조교수
1989년~현재 연세대학교 공과대학 컴퓨터과학과 부교수
관심분야:병렬처리 컴퓨터구조, ASIC 설계, 고성능 컴