10.4 2의 보수 표기법2's Complementary 2008.08.09

■ 음수의 표기 및 저장

마이크로컴퓨터microcomputer나 DSP의 내부에서 처리할 수 있는 수의 체계는 이진수밖에 없으므로 우리가 음수라든지 소수점을 가진 실수 등을 처리하려면 특별한 수의 처리 및 저장 방법이 필요하다. 구체적인 예를 들어, DSP 외부에 장착된 램RAM에 음수나 소수점을 가진 실수를 저장하기 위해서는 특별한 변환 및 표기방법이 필요하다.

DSP 내부나 물리적인 메모리에서는 모든 숫자나 명령어가 '0'과 '1'의 2가지 상태로 표기되는데, 이러한 경우 음수를 표기하기 위한 방법을 생각해보자. 먼저 가장 간단하게 생각할 수 있는 아이디어는 각 비트bit의 가장 앞에 위치한 최상위비트MSB를 부호비트로 지정하여 사용하는 방법일 것이다. 예를 들면 3비트 이진체계에서 십진수 -3을 표기한다면 "011" 에 부호 비트를 추가하여 "111" 로 표기할 수 있을 것이다.

그런데 만약 상기와 같은 방법을 사용한다면 개념적으로는 간단하게 이해할 수 있지만 몇 가지 문제점이 발생한다. 3비트의 코드code로는 원래 나타낼 수 있는 수가 2^3 으로 8가지가 되는데, 상기와 같은 방법을 사용하면 "000"과 "100"이 모두 십진수 0을 나타내야 하므로 같은 비트로 표기할 수 있는 개수가 하나 감소하므로 수의 표기에 있어서 낭비가 발생한다. 또한 다른 중요한 문제는 상기와 같은 방법으로 음수를 표기하면 덧셈이나 곱셈 등의 수의 처리과정에서 경우를 따져서 일일이 분기문을 사용하여 처리해 주어야 한다는 것이다.

이진수에서 2의 보수법2's complementary을 사용하면 로우 레벨 언어 및 기계어의 내부에서 이진 수의 뺄셈 등의 연산을 매우 편리하게 할 수 있다. 2의 보수법을 사용한 음수 표기 방법을 알아보자.

■ 2의 보수법2's complementary

상기의 아이디어와 같이 음수의 2진 표기에 있어서 가장 앞의 최상위 비트를 부호비트로 정의 하는 방법도 있지만 다음과 같은 뺄셈을 이용한 방법으로 부호를 내재하여 표기하는 경우를 생각해보자. 상위비트에서 내림Borrow을 하나 제공하고 실제로 빼서 표기하는 것이다. 예를 들어 만약 십진수로 -3을 이진수로 표기한다면,

$$\begin{array}{r} 1\ 0\ 0\ 0 \\ -\quad 0\ 1\ 1 \\ \hline 1\ 0\ 1 \end{array} \qquad (10.4.1)$$

로 표기될 수 있다. 이러한 방법을 수식으로 표기한다면 어떠한 수 F의 2의 보수 표기법은,

$$F^* = 2^n - F \qquad (10.4.2)$$

과 같은 형태로 표기가 된다(n은 n비트의 수를 뜻함). 일반적으로 우리가 알고 있는 방법인 **"보수를 취한 후 1을 더하는 방법"**은 상기의 수식 (10.4.2)에 의하여 계산하는 방법과 동일하다(011의 보수는 100이고 여기에 001을 더하면 101이 됨). 상기와 같이 음수를 표현했을 때 3비트의 이진수가 나타낼 수 있는 바이폴라bipolar의 전체 범위는 다음과 같다.

표 10.4.1 2의 보수표기법으로 나타낸 음의 이진수

십진수	2의 보수 표기법
3	011
2	010
1	001
0	000
-1	111
-2	110
-3	101
-4	100

이러한 경우 단순하게 부호 비트를 사용할 때와는 달리 3비트로 8개의 수를 표현할 수 있으며, 연산에 있어서도 매우 편리하다. 유심히 보면 2의 보수 표기법에 처음 도입부분에 언급했던 부호비트 개념도 내재되어 있음을 알 수 있다(음수는 모두 최상위 비트가 1임). 음수의 표기시 이러한 2의 보수법을 사용했을 때의 가장 큰 장점은 덧셈 등의 연산시 기존의 프로세스를 이용하여 연산을 해도 자동적으로 결과의 부호가 동일한 표현방법에 의해 생성된다. 2의 보수법을 사용하여 식(10.4.3)과 같은 음수의 연산을 한번 고려해 보자.

$$2 - 3 = -1 \qquad (10.4.3)$$

3bit의 경우 이진수에서 -3을 음의 부호로 표기하면 다음과 같다.

표 10.4.2 2의 보수표기 예제

십진수	부호 뺀 이진수	2의 보수 표기법
-3	011	101

이를 기존의 이진 연산에서 사용하고 있는 일반적인 덧셈 연산을 사용하여 더하여 보자.

표 10.4.3 2의 보수법 사용시 음수의 연산

십진표기	2	+	(-3)	=	-1
이진표기	010		101		111

-3을 2의 보수법으로 표기하게 되면 101과 같이 표기되고, 이는 기존의 일반적인 비트 연산에 의해 010에 더해지면 111이 된다. 이는 2의 보수법으로 -1에 해당되는 수이다. 따라서 상기와 같이 음수를 2의 보수법으로 표기하게 되면 기존의 일반적인 덧셈 프로세스를 이용하여 음의 부호를 가진 수의 덧셈 처리가 가능하며, 기존의 양수에서 사용하는 연산기ALU 등을 그대로 사용할 수 있으므로 매우 빠르고 효율적인 처리가 가능하다. 만약 이러한 방법을 사용하지 않고, 단순히 한 비트를 부호용으로 사용한다던가 하면 단순 뺄셈 연산이 한 번의 사이클Cycle에 끝나지 못하고, 부호에 따른 조건 검색 및 분기 등의 추가 절차가 필요하게 된다.

정리하면, 양수는 기존의 이진수 표기 방법으로 표현하고, 음수의 경우 상기와 같은 2의 보수표기법2's complementary을 사용하면 음수의 뺄셈 연산시 간단하게 기존 덧셈 구조를 사용하여 행할 수 있다.

■ 이진수의 소수 표현

이진수에서도 십진수와 같이 소수점이 있는 수를 표현할 수 있다. 예컨대 십진수에서 10.1은 10과 10분의 1을 의미한다. 비슷한 방법으로 이진수에서는 10.1이 십진수로는 2와 2분의 1을 의미한다. 만약 11.011을 십진수로 풀어쓰면 다음과 같다(2진 소수 옆에 붙은 아래첨자는 2진수라는 표시임).

$$11.011_2 = 2 + 1 + \frac{1}{4} + \frac{1}{8} = \frac{27}{8} \qquad (10.4.4)$$

이러한 이진 소수를 제한된 메모리 안에 저장하는 방법을 생각해보자. 부동소수점floating point일 경우에는 다음과 같이 유효숫자 곱하기 지수 형태로 표현하는 것이 가장 간단하고 편리한 방법이다.

$$N = F \times 2^E \qquad (10.4.5)$$

여기서 F는 유효숫자(프랙션fraction)를 의미하며, E는 지수exponent를 의미한다. 여기서 프랙션과 지수는 둘 다 음수표현이 허용되며, 상기에서 설명한 2의 보수법을 사용한다. 만약 유효숫자가 4비트, 지수가 4비트의 크기를 가진다면, 만약

십진수로 2.5는 다음과 같이 표기될 수 있을 것이다.

$$F = 0.101, \quad E = 0010, \qquad N = \frac{5}{8} \times 2^2. \tag{10.4.6}$$

제한된 비트수에 상기와 같은 방법으로 어떠한 소수를 표현한다면, 유효숫자를 가장 크게 가져가는 것이 유리할 것이다. 따라서 일반적으로 프랙션Fraction이 가장 큰 숫자를 가지도록 표준화normalize하여 사용한다.

그렇다면 0은 어떻게 표현될까. 0은 가장 작은 프랙션에 가장 작은 지수를 곱한 방법으로 다음과 같이 표현된다.

$$F = 0.000, \quad E = 1000, \qquad N = 0 \times 2^{-8}. \tag{10.4.7}$$

■ 맨티사 Mantissa

TMS(Texas Instrument의 대표적인 DSP 시리즈)의 부동소수점floating point 연산가능 DSP 시리즈series에서 사용되는 부동소수점 표현 구조를 맨티사Mantissa라고 하는데, 이러한 맨티사Mantissa에도에도 이러한 2의 보수법을 사용한 방법이 사용되고 있다. 맨티사의 부동 소수점 표현구조는 다음과 같다.

$$X = s\bar{s}.f_2 \times 2^e \tag{10.4.8}$$

여기서 s는 사인sign 비트bit, f_2는 2진수로 표현되는 프랙션fraction 필드field, e는 지수Exponent 부분을 표시한다. 맨티사Mantissa는 유효숫자 부분이 두 개의 사인 비트bit와 나머지 소수점이하의 프랙션 비트로 구성되어 있는데, 만약 X가 양수이면 맨티사의 최초 2비트는 "01"이 되고, 음수면 "10"이 된다. 맨티사Mantissa 형식은 16비트 구조에서는 물리적으로 다음과 같이 정의된 형식으로 메모리에 저장된다.

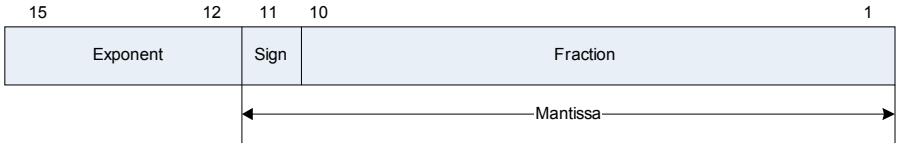

그림. 10.4.1 맨티사의 물리적 저장시 사용되는 구조

맨티사도 음수의 경우 2의 보수법을 사용하는데, 예를 들어 표현하려고 하는 수 X가 -3.0이라면 이는 2의 보수법으로 101이 되는데, 이는 맨티사로 다음과 같이 표현된다(여기서 10.1_2은 2진 소수이다.)

$$X = 10.1_2 \times 2 = 101_2 \text{(S = 1, F = 1, E = 1)} \quad (10.4.9)$$

만약 음의 정수 뒤에 프랙션fraction이 있더라도 상기와 같이 2의 보수법을 사용하여 편리하게 2진 소수로 표현할 수 있다. -3.0은 16비트의 경우 짧은부동소수점형식short floating-point format에서는 물리적으로 다음과 같이 메모리에 저장된다.

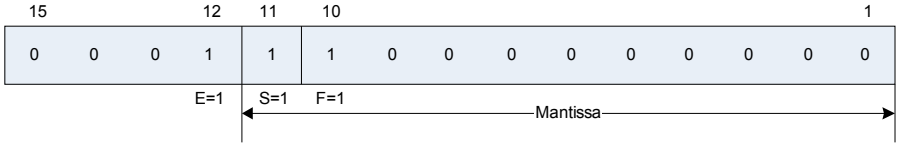

그림.10.4.2 -3.0의 메모리 저장 형식

부동소수점floating point은 단일 정밀도single precision 혹은 확장된 정밀도extended precision등과 같은 부동소수점의 종류와 사용하는 비트수(16/32)에 따라 할당된 비트의 수 등이 달라지지만 맨티사의 구조자체는 큰 차이가 없다.

이번에는 소수점이 있는 수의 표기를 한번 살펴보자. 예를 들어 만약 3/64을 맨티사 구조로 표현할 경우, 이는 이진수 3을 64(2^6)로 나누어 주면shift 될 것이다. 이를 맨티사 구조로 표현한다면,

$$X = 01.1_2 \times 2^{-5} = 0.000011_2 = \frac{3}{64} \text{(S = 0, F = 1, E = -5)} \quad (10.4.10)$$

과 같이 표현된다. 구조를 자세히 살펴보면 맨티사Mantissa의 맨 앞자리 2개가 $\bar{ss}$ 로 구성되어 있으므로 상기와 같이 지수exponent 부분을 보다 효율적으로 사용된다는 것을 알 수 있다. 상기의 경우 지수exponent부분 음수인데, 이 부분도 물리적으로 저장될 때 2의 보수법을 사용한다. 따라서 기본적으로 맨티사와 유사한 부동 소수점 연산을 사용하는 DSP는 소수를 표현 할 때 2의 n승으로 이루어진 분모와 이진수를 병행 사용하여 소수를 표기한다는 것을 알 수 있다(이진수의 소수이므로 분모는 2의 배수가 됨. 십진수의 경우에 10분의 1, 100분의 1로 표현되는 것과 마찬가지임).

■ 이종 비트수간의 변환

2의 보수방법으로 표기된 레지스터 등을 다룰 때, 가장 유의해야 할 점은 바로 자리수이다. 예를 들어 12비트bit의 정수의 경우 -3에 대한 표현식은 1111 1111 1101 이지만 이를 만약 16비트 구조로 읽으면 4093 의 양수로 매우 큰

차이를 가져오게 된다.

따라서 2의 보수법을 사용하는 경우 비트수가 다르면 오류가 발생하는데, 이는 DSP에서 레지스터의 크기와 다른 비트 수의 외부 기기 및 주변 칩과 통신할 때 주로 발생한다. 예를 들면, 32비트 DSP에 12 비트의 A/D변환기converter의가 장착된 경우 등이다. A/D변환기의 경우 가격 등의 이유로 거의 대부분 8 ~ 20 비트 내외를 사용하기 때문에, 이를 32비트의 DSP가 오류 없이 처리하려면 특별한 알고리듬이 필요하다.

외부에 장착된 A/D 변환기를 바이폴라bipolar 방식으로 DSP가 읽을 경우 2의 보수법을 사용한다면 반드시 그 A/D 변환기의 전달 규약을 참조하여 데이터를 DSP의 레지스터register 크기로 변환시켜 주어야 한다. 이는 주로 쉬프트shift 연산을 이용하여 레지스터에 저장하고 이를 다시 쉬프트 다운shift down 하는 방식을 이용하면 효율적으로 2의 보수 표현식을 유지할 수 있다.

다음의 함수 ADC_Read()는 16 비트의 A/D의 바이폴라bipolar 출력을 32 비트 레지스터에 저장하는 알고리듬이다. 2의 보수법으로 출력되는 A/D 변환기에서 사인sign 비트를 최상위 비트로 유지하면서 읽어 들이고, 이를 다시 쉬프트 다운shift down 하면 쉬프트 연산자가 자동으로 부호값을 반영하여 변환해 준다(컴파일러compiler 마다 쉬프트shift 연산에 차이가 있을 수 있음).

[프로그램 예제 10.4.1]

```
#define ADC_READ    *(long *)0x80C000
                            /* A/D의 출력 버퍼 주소 */
long ADC_Read(void)
{
    long lADValue;          /* 32 bit 의 long 변수 이용 */
    lADValue = (ADC_READ &0xffff) << 16;
                            /* 하위 16bit를 Shift 해서 읽는다 */
    lADValue = lADValue >> 16;
    return(lADValue);       /* 함수값 리턴 */
}
```