컴퓨터 시스템 아키텍처

정수와 부동소수점

141019 김연우

1. 컴퓨터에서 정보의 표현과 저장 : 전기적 신호 -> 숫자의 표현 방법

A. 다양한 데이터 타입

i. 워드(WORD) (cf : bit 0 or 1, byte 8bit)

1. Bit와 byte는 고정된 형식 하지만 나머지 데이터 타입은 시스템의 정의에 따라 다루는 방식이 결정된다.
2. 컴퓨터 시스템의 기본적 단위의 크기
3. 일반적으로 정수데이터의 크기와 일치 (시스템 비트 수와 관련)

ii. 시스템 별로 데이터 타입이 크기가 다르기 때문에 호환성에 영향

iii. 실제 데이터들은 결과적으로 비트의 집합, 하지만 그것을 읽어내는 방법이 다름

B. 바이트의 저장 순서

i. Endian : 하나의 데이터 단위가 1바이트 이상일 때, 메모리에 어떻게 저장 해야 되는 지를 정의하는 것. (byte ordering)

1. 메모리의 저장되는 주소 값의 순서와 관련되므로, CPU의 연산과는 별개. 레지스터에 올라오면 결국 연산 순으로 저장된다.
2. Big Endian : 큰 놈이 끝(메모리 tail)에 있다.
3. Little Endian : 작은 놈이 끝(메모리 tail)에 있다.

ii. 컴퓨터는 결국 비트 집합들의 연산, 하지만 여러 데이터 타입들의 단위가 다르므로 시스템에서 데이터를 다루는 방식들이 다르게 정의되어 있고, 그것을 제대로 이해해야 적절하게 다룰 수 있다.

iii. 기계어 코드를 이해할 때 주소 데이터 단위로 작성될 수 있기 때문에, 각 Endian방식으로 값을 해석할 수 있어야 한다.

iv. 1 byte단위로 메모리를 읽어들이면 byte ordering에 무관한 방식으로 데이터를 출력할 수 있다. 이 방법으로 현 시스템의 Endian방식을 유추할 수 있다. (과제)

C. 문자열 표현

   i. Byte의 배열로 숫자 뿐 아니라 다양한 표현이 가능.
   ii. Encoding : 문법(syntax)이 정의되면 여러 형식의 의미로 변환가능
   iii. ASCII, Unicode, UTF-8 등 비트를 문자로 인코딩

D. 비트 수준의 연산

   i. AND, OR, NOT, XOR 연산
   ii. (1 / 0) >> encoding >> (TRUE / FALSE)
   iii. 산술 연산자 vs 논리 연산자
      1. 논리 연산자의 결과는 반드시 0 or 1
      2. 논리 연산자의 특징 Early Termination (ptr && ptr*)
   iv. Shift 연산
      1. X << Y (x 를 y만큼 왼쪽 시프트)
         A. domain에서 벗어난 자리수는 버림
         B. 새로 생긴 자리는 0으로 채움
      2. X >> Y
         A. 논리 시프트 : 왼쪽을 0으로 채움,
         B. 산술 시프트 : 원래 음수 부호가 있는 경우 다르다.
            i. 보수표현의 문법을 유지해야하기 때문
            ii. 오른쪽을 버리고 왼쪽을 0이아닌 MSB 비트로 채운다.
            iii. 왼쪽 시프트의 경우 기존 보수의 형식이 유지되므로 상관없음.
         C. Undefined : x << or >> y (for y <0 or >= word Size)
            i. 결국 CPU연산이므로 CPU연산 단위 이상으로 shift 연산 명령을 할 수 없다. (long long에 x << 50,

2. 정수의 표현과 산술 연산

A. Unsigned int

B. Signed Int

i. MBS : 추가 비를 붙이면 연산 불편, 데이터 다루기 어려움, 잘안씀

ii. 2의 보수

1. N의 보수

A. 더했을 때 n이 되는 수 (cf 보색 : 두 색을 더했을 때 흰색, 또는 검은색이 됨)

B. p진법의 k자릿수에서 x의 n의 보수 = (p^k – x) + n

2. n비트로 표현된 정수 x의 2의 보수 = 2^n – x

A. 2^n은 n개의 비트로 표현할 수 없음. OVER FLOW 이기 때문에 2^n 값은 무시됨.

B. 따라서 컴퓨터 내부에서 –x처럼 사용할 수 있다.

C. Cpu의 음수 계산방식이 간단해짐. (뺄셈을 덧셈으로)

3. 컴퓨터가 계산하는 것을 수학적으로 표현

A. Unsigned

i. B2U(X) = $\sum_{i=0}^{n-1} Xi * 2^{i}$

ii. Domain : 0 ~ 2^{n} - 1

B. Two's complement

i. B2T(X) = $-\mathrm{X}(\mathrm{n}-1) * 2^{n-1} + \sum_{i=0}^{n-2} Xi * 2^{i}$

ii. 결국 맨 위 비트가 부호비트(MBS)처럼 작동함

iii. Domain : $-2^{n-1}(Tmin)$~ $2^{n-1}-1(\mathrm{Tmax})$

C. 두 방식의 비교

i. 비트표현은 동일하지만 해석하는 방식에 따라 의미하는 숫자가 달라진다.

ii. 의미하는 숫자 값의 순서(ordering)가 같기 때문에 같은 연산방법으로 동일한 결과 비트를 얻을 수 있다. (over flow 포함)

D. 두 형식의 변환

i. 실제 주소에 저장된 비트 값은 변하지 않는다. 다만 유저에게 번역되는 방식이 변경될 뿐.

ii. (unsigned) 128 => (Two's complement) -128

n개의 비트로 표현된 숫자에서 Unsigned to Signed 변환방식

if(unsigned >= 2^(n-1)) signed = unsigned – 2^n

반대 경우는 쉽게 유추가능

C. 타입 변환의 특이성 고려

i. Signed 와 Unsigned가 혼합된 경우 Unsigned로 implicit casting된다.

ii. 예제 (숫자.U 는 unsigned)

1. 0.U > -1

2. 2^31 - 1> (int) 2^31.U

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

iii. 잠재적인 버그

입력 값 maxlen에 -1024를 넣었다면?

습관의 문제: 데이터 타입을 고려해서 사용해야 된다. 잠재적 형 변환 주의.

D. 연산

i. 확장

1. short => int 로 형 변환

2. 의미 값이 변하지 않은 상태로 데이터 형식을 변화시켜야 된다.

3. 부호가 없는 경우는 상위 비트에 0을 넣어주면 되는데, 부호가 –인 경우 상위 비트에 1을 넣어주어야 음수 값이 같아진다. (2의 보수 개념 상기)

ii. 절삭

1. int => short 로 형변환
2. 기본적인 모듈로 연산
3. – 부호가 있는 경우 의미 손실 가능성이 있다.

iii. 덧셈

1. 주어진 자릿수에서 Overflow된 경우 절삭.
2. (Unsigned) X + Y = (X + Y) % 2^n
3. 덧셈에 있어서는 Unsigned / Signed의 비트계산자체는 동일하므로 전달되는 값의 형식만 다를 뿐 실제로 다르지 않다. Unsigned로 계산한 뒤 Signed로 변환하는 게 마음 편함.(내 생각)
4. 2항 덧셈연산 overflow 발생 가능성
   A. 두 수가 모두 음수인데 합이 양수인 경우
   B. 두수가 모두 양수인데 합이 음수인 경우

iv. 곱셈

1. 마찬가지로 자릿수 넘어가면 절삭
2. (Unsigned) X * Y = (X * Y) % 2^n
3. 해석차이만 있는 Signed 곱셈 연산
4. 곱셈 연산을 코드에 넣는 경우 overflow 고려해야한다! (절삭 되서 의도한 값보다 훨씬 작을 수 있음)
5. 변수의 곱셈은 루프문 내부에 안 넣는게 좋다고 하심…
6. 상수의 곱셈은 많은 부분을 shift연산으로 대체할 수 있다. 컴파일러는 알아서 해준다… 고정된 값은 변수 쓰지말자. 하지만 shift연산을 하다가 본래 곱셈에서 발생하지 않는 OverFlow가 발생하므로 주의

v. 나눗셈 (11_04)

1. 오른쪽 시프트 연산으로 구현가능
2. 본래 수학적 나눗셈 연산은 0으로의 근사이다.
3. 하지만 시프트 연산으로 하면 영역 밖이 버려지기 때문에 0으로의 근사가 아니라 무조건 더 작은 수로 고정되기 때문에, 음수의 나눗셈 연산인 경우 시프트만으로는 부족하며 기존 나눗셈 연산으로 돌리기 위한 보정연산이 필요하다.
4. 음수 나눗셈의 경우 x/2^k = (x+2^k – 1) / 2^k 로 구현한다. 이 식은 음수인 피제수에 대하여 반드시 0으로의 근사를 적용할 수 있다.

```
/* x/8*/
if(x<0)
x += 7
return x >> 3
```

3. 실수 표현방식(부동 소수점)

A. 비율 이진수(Fractional Binary Numbers)

i. 기존 이진수의 실수표현방식

ii. x/2^k의 형태로 표시되는 숫자만 표시 가능

iii. 컴퓨터로 표현하는 것의 문제 : 소수점을 어떻게 표현할 것이냐. 정수부와 실수부의 구분은?

B. IEEE 부동 소수점

i. 소수를 컴퓨터로 표현하는 표준 방식.

ii. 부호 비트 s, 유효숫자 M, 지수 E 로 구성된다.

-1^0(s) * 11011(M)* 2^-2(E)

iii. 비트 인코딩

1. MSB : s부호 비트
2. Exp 필드: 지수 E 의 표현(항상 E와 같지는 않음)
3. frac필드 : 유효숫자 M의 표현 (항상 M과 같지는 않음)
4. 컴파일러는 타입에 의한 해석

iv. 정밀도 이슈

1. 몇 비트 컴퓨터이냐에 따라서 정밀도가 다름
2. 32비트의 경우 exp 8 bit, frac 23 bit
3. 64비트의 경우 exp 11 bit, frac 52 bit
4. 결국 나타낼 수 있는 표현의 경우는 2^비트갯수

v. 꼼수 : 0~1사이의 실수는 많이 쓰지만 1 이상의 실수가 적게 사용된다는 것을 사용. Exp의 값을 보고 결정한다.

1. 정규화된 값 : 1보다 큰 실수. Exp != 0 && Exp != 11111...
2. 비정규화된 값 : 0 ~ 1 사이에 있는 실수 Exp == 0
3. 특수 값 : 무한대와 NaN 표현 Exp == 11111...

vi. 정규화 값 (Exp != 0 && Exp != 1111...)

1. E = exp – Bias;
   A. 8비트 값인 exp가 음수도 표현 할 수 있어야 한다.
   B. Exp의 음수표현이 보수형태가 되면 비트만 보면 음수가 양수보다 큰 값처럼 보인다.
   C. 따라서 보수형태로 음수표현 하지 않고 Bias를 빼는 방식으로, 즉 비트값이 작으면 전체 값도 작도록, 실수의 크기 비교 연산이 비트 수준에서 해결 될 수 있도록 구현한다.
   D. 따라서 Bias = 2^(bit - 1) – 1
2. M = 1 + f = 1.xxxxxxx.... (implicit 한 정수부 1이 존재)
   A. 항상 선두에 선 값 1을 지정해줘야 한다. (0 제외)
   B. 항상 들어오므로 생략 가능하다. (비트 하나 생략 개이득)
   C. 따라서 M = 1 + f

vii. 비정규화 값 (Exp == 0)

1. E = 1 – Bias (exp는 무조건 0 이므로... 그렇다고 0 - Bias아님!)
   A. 결국 표현하는 영역 2^-126 이하 (정규화로 표현할 수 있는 최소값)
2. M = 0 + f (implicit 0, 0에 가까운 숫자를 쓰기 위해서)
   A. 이때 묵시적 선두 0을 사용해서 자연스럽게 연속된 실수를 표현 가능하다. (2^-126*0.111111111.... 2^-126*1.0)

viii. 특수 값 (Exp == 111111....)
1. Exp == 111..... && frac == 0
   A. 무한대(음수 / 양수)
   B. 연산의 오버 플로우를 표현
2. Exp == 111..... && frac != 0
   A. Not a Number (NaN)
   B. Undefined

ix. 부동소수점 값의 분포
1. Exp의 모든 값마다 같은 개수의 값을 갖는다. 따라서 값이 0 에 가까울 수록 조밀한 분포를 보인다.
2. 그리고 모든 부동 소수점 값은 비정규 값이나 정규값 모두 자신의 간격만큼 연속되는 속성을 갖는다.

x. 부동 소수점의 한계
1. 정밀도의 제약으로 표현하지 못하는 정수가 존재. Frac의 총 비트수가 n이라고 하면 2^n+1 + 1은 표현 불가능. frac으로 그 숫자의 자리 수 전체를 표현할 수 없기 때문이다. 32비트 컴퓨터에서는 2^24 + 1 부터 표현 불가.

xi. 부동 소수점 연산
1. 보수 개념이 없으므로 양수가 갑자기 음수가 되지 않는다
2. 오버플로우가 일어나는 경우 근사로 계산할 수 있다. (짝수 근사)
   A. 기본은 반올림
   B. 반올림은 쩜오에 대해 너무 편향적이니까 확률적으로 안정적인 방식 짝수 근사를 적용한다.

C. 쩜오를 정수부가 짝수가 되도록 올리거나 내린다.

D. 이진수의 경우 0.5 대신 1/4 가 적용된다.

3. 덧셈

A. 지수부가 있으므로 자릿수가 다를 수 있다.

B. 높은 자릿수에 맞춰서 덧셈 수행한뒤 결과값을 보정한다.

4. 곱셈

A. 곱셈이므로 자릿수 맞출 필요 없다.

B. 유효숫자를 그냥 곱하고 지수부는 더해준다. 그리고 결과값을 보정한다.

xii. 타입 변환

1. int -> float : 오버플로우 없다. 값 근사

2. int , float -> double : 값 유지되는 정확한 변환

3. float, double -> 오버플로우 있다.