C++ 템플릿 기반의 Fixed-Point 연산 라이브러리 [1]

황석중*, 김선욱*, 민병권**
*고려대학교 전기전자전파공학
**뮤텔테크놀러지(주)
e-mail : {nzthing,seon}@korea.ac.kr, bgmin@mewtel.com

C++ Template-based Fixed-Point Arithmetic Library

Seon Joong Hwang*, Seon Wook Kim*, Byung Gueon Min**
*School of Electrical Engineering, Korea University
**Mewtel Technology Inc.

요 약

디지털 신호처리 알고리즘들은 실제 시스템에 적용할 때 임베디드 시스템 등 하드웨어의 성능과 소비전력 및 비용에 제약이 있을 경우 연산 정밀도가 높은 floating-point 연산 대신 제한된 정밀도와 적은 연산 비용을 요구하는 fixed-point 연산을 사용하여 구현한다. 시스템의 개발단계에서는 적용할 알고리즘을 floating-point 연산을 이용한 코드를 먼저 작성한 후 이를 fixed-point 연산으로 대체하는 과정을 거치게 되는데, 이는 숙련된 개발자와 상당한 양의 개발기간을 요하는 까다로운 작업이다. 이에 본 연구에는 코드작성 편의를 높이고 개발기간을 단축하기 위해 C++ template 기반의 fixed-point 연산 라이브러리를 개발하였다. 이는 floating-point 연산 코드와 fixed-point 연산 코드를 별도로 개발할 필요 없이 하나의 코드를 이용하여 자유로이 연산 정밀도를 지정할 수 있으며 개발자는 기존의 floating-point 연산을 이용하는 코드를 작성하는 것처럼 쉽게 코드를 작성할 수 있도록 한다. 또한, template 기반으로 작성되어 기존의 연구들과 달리 추가적인 작업도구 없이도 범용 C++ 컴파일러가 최적화된 코드를 생성할 수 있도록 되어있는 것이 특징이다.

1. 서론

디지털 신호처리 알고리즘을 실제 시스템에 적용할 때 임베디드 시스템 등 하드웨어 성능과 소비전력 및 비용에 제약이 있는 경우 연산 정밀도가 높은 floating-point 연산 대신 제한된 정밀도와 적은 연산 비용을 요구하는 fixed-point 연산을 사용하여 구현한다. 시스템의 개발단계에서는 적용할 알고리즘을 floating-point 연산을 이용한 코드를 먼저 작성하여 검증 및 성능을 평가한 후 이를 fixed-point 연산하는 과정을 거치게 된다. 실수 타입의 데이터를 이용하는 floating-point 연산과 달리 fixed-point 연산은 정수 타입의 데이터를 이용하여 일정 정밀도를 갖는 실수 연산을 수행하는 것이다. 이는 숙련된 개발자를 요하는 까다로운 작업일 뿐만 아니라 최적을 정밀도를 얻기 위해 다양한 정밀도들에 대해 코드를 작성하고 시뮬레이션을 통해 평가해야 하므로 상당한 양의 개발기간을 요한다.

이러한 어려움을 극복하기 위해 본 연구에서는 코드작성의 편의를 높이고 개발기간을 단축시킬 수 있는 C++ template 기반의 fixed-point 연산 라이브러리를 개발하였다. 본 연구에서 개발한 라이브러리는 floating-point 연산 코드와 fixed-point 연산 코드를 별도로 개발할 필요 없이 하나의 코드를 이용하여 자유로이 연산 정밀도를 지정할 수 있도록 하였다. 개발자는 연산 정밀도 지정을 제외하고는 기존의 floating-point 연산을 이용하는 코드를 작성하는 것과 거의 동일하게 코드를 작성할 수 있도록 되어있어 개발 편의를 높였으며 기존 개발되었던 코드들도 본 연구에서 개발한 라이브러리를 이용하도록 쉽게 수정할 수 있다.

Fixed-point 연산에 대한 기존의 연구들은 크게 floating-point 연산 코드를 fixed-point 연산 코드로 변환을 자동화하는 연구[1], fixed-point 연산의 최적의 정밀도를 찾는 연구[1-4]와 최적화된 fixed-point 연산 코드를 생성하는 연구[4-5]로 나누어 볼 수 있다. 본 연구는 라이브러리를 통한 최적화된 fixed-point 연산 코드를 생성하는 연구의 범주에 속한다. 기존의 연구들이 이를 위해 컴파일러 수정이나 별도의 코드 변환 도구를 요구하는 반면, 본 연구는 C++의 template 의 장점을 활용하여 라이브러리 만으로 최적화된 코드를 생성하는 것이 특징이다. 이 방식은 기존의 모든 개발환경에 적용이 가능하며, 별도의 변환과정을 거치지 않으므로 개발자가 원시코드 수준에서 보다 정교한 최적화 과정을 수행할 수 있는 장점이 있다.

본 논문은 연구를 통해 개발한 C++ template 기반의 fixed-point 연산 라이브러리의 구조와 사용법, 통신

[1] 본 연구는 서울시 산학연 협력사업(10920)과 중소기업청 기업부설연구소 업그레이드 지원사업으로 지원된 것임.

알고리즘 중 패킷 검출기를 라이브러리 적용하여 구현하는 것을 예로 소개하는 것으로 구성하였다.

2. 라이브러리 특징

본 연구에서 개발한 라이브러리에서는 임의의 정밀도를 설정할 수 있는 새로운 데이터 타입을 도입하였고 데이터 타입은 클래스 텟플릿으로 정의하고 데이터에 대한 연산을 클래스의 연산자를 overloading 하여 구현하였다. <표 1>은 기본 데이터 타입인 임의의 정밀도 설정이 가능한 scalar 타입에 대해 기존의 연구[4](이하 gFix)에서 정의된 것과 본 연구에서 개발한 것(이하 vp_scalar)의 일부를 나타내고 있다.

<표 1> 임의의 정밀도 설정이 가능한 scalar 타입의 정의 비교.

기존의 연구에서 정의한 scalar type

```
class gFix
{
  Integer   m;
  short   iwl;
  short   wl;
  char   represent;
  char   saturation;
  char   round;
```

본 연구에서 정의한 scalar type

```
template<bool UNSIGNED, int REGSZ>
struct vp_register;

template<bool SIGNED>
struct vp_register<SIGNED, 0> { double value; };

template<> struct vp_register<false, 8> { int8     value; };
template<> struct vp_register<true , 8> { uint8    value; };
template<> struct vp_register<false,16> { int16    value; };
template<> struct vp_register<true ,16> { uint16   value; };
template<> struct vp_register<false,32> { int32    value; };
template<> struct vp_register<true ,32> { uint32   value; };
template<> struct vp_register<false,64> { int64    value; };
template<> struct vp_register<true ,64> { uint64   value; };

template <
  int INTG,
  int FRAC = 0,
  int MODE = (INTG+FRAC)==0?VPM_Float:VPM_Def>
class vp_scalar
{
  vp_register<
    !!(MODE & VPM_Uns),
     ((INTG+FRAC)<=32)
   ? ((INTG+FRAC)<=16)
   ? ((INTG+FRAC)<=8 )
  ? 8 : 16 : 32 : 64> reg;
```

gFix 의 경우 정밀도에 대한 정보를 클래스의 멤버변수로 저장하도록 되어있다. 이러한 방식은 컴파일러가 최적화를 수행하기 어려운 문제가 있으며, 클래스의 객체들의 크기가 실제 데이터를 저장하기 위해 필요한 공간보다 커져 메모리 사용의 효율성이 떨어지게 된다. 따라서 gFix 를 최적화할 수 있도록 컴파일러 수정하거나 별도의 코드 변환 도구를 통하여 최적화할 필요가 있는 것이다.

vp_scalar 는 이 문제를 클래스 템플릿을 이용하여 해결하였다. 클래스 템플릿은 파라미터로 데이터 타입을 입력 받는 것이 일반적지만 vp_scalar 는 C++가 데이터 타입 이외에 상수 값의 입력도 허용하는 점을 이용하여 정밀도 정보를 입력 받도록 하였다. 이는 상수 값을 취하기 때문에 컴파일러가 최적화를 잘 수행할 수 있으며, 추가적인 클래스 멤버변수로 인한 메모리 낭비를 막을 수 있다.

vp_math 는 하나의 멤버변수인 vp_register 타입의 reg 를 갖는다. 이는 임의의 정밀도를 갖는 데이터를 저장하기 위한 공간으로 템플릿 특수화 기법을 이용하여, 해당 정밀도에 필요한 최소한의 크기를 갖는다. 가령, floating-point 연산모드로 정밀도가 설정이 되면, reg 는 double 타입으로 선언되며, fixed-point 연산모드에서 정수부와 소수부의 해상도의 합이 8 이하이면 char 타입으로 선언되고 16 이하면 short 으로 선언되는 등의 방식이다.

```
vp_scalar<I,F,M> varname;
```

vp_scalar 를 이용한 변수 사용방법은 의와 같다. I 는 정수부의 해상도, F 는 실수부의 해상도 M 은 연산결과값의 대입의 모드이며 saturation, rounding, truncation 등을 지정할 수 있으며(<표 2> 참고), 미지정시 saturation+rounding 이 설정된다. I 와 F 값을 0 으로 지정할 경우, floating-point 연산을 사용하게 된다.

<표 2> 연산결과 대입 모드.

MODE	의미
VPM_Uns	변수가 unsigned type 임을 나타낸다.
VPM_Rnd	입력되는 값의 소수부가 더 클 경우 rounding 시킨다.
VPM_Sat	입력되는 값의 정수부가 더 클 경우 saturation 시킨다.
VPM_Arr	변수가 array 임을 나타낸다.
VPM_Float	변수가 floating-point type 임을 나타낸다.
VPM_RndSat VPM_Def	= VPM_Rnd | VPM_Sat (default)
VPM_URndSat	= VPM_Uns | VPM_Rnd | VPM_Sat
VPM_USat	= VPM_Uns | VPM_Sat
VPM_URnd	= VPM_Uns | VPM_Rnd

<그림 1>은 라이브러리의 사용 예제와 라이브러리의 내부 동작을 나타낸 것이다. <표 3>과 같이 나눗셈을 제외한 사칙연산과 비교연산이 연산자 overloading 을 통하여 floating-point 연산 코드와 동일하게 사용할 수 있게 되어있다. 연산자 overloading 함수는 피연산자들(x,y)의 연산결과가 가질 수 있는 최대의 정밀도를 갖는 임시 저장변수에 결과를 저장하여 반환하게 되며,

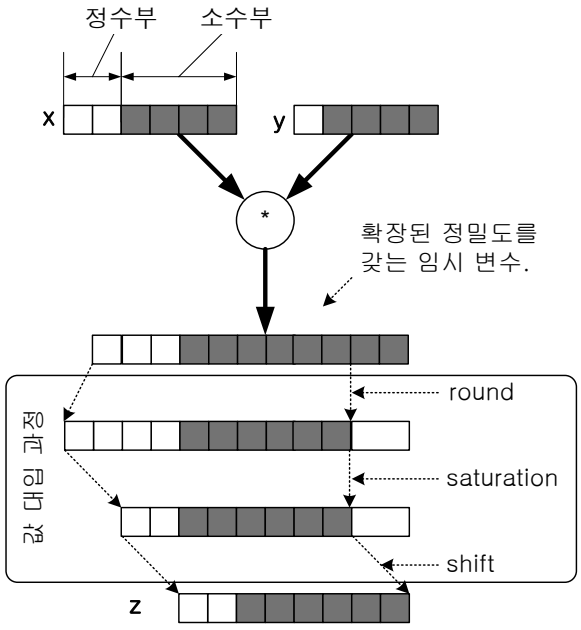
임시 저장변수가 최종 저장 변수(z)에 입력될 때 shift, saturation, rounding 등을 통하여 정밀도를 맞추게 된다. 단, 나눗셈에 대해 연산자 overloading 을 지원하지 않는 이유는, 나눗셈의 경우 연산결과가 가질 수 있는 정밀도가 무한대일 수 있기 때문이다. 대신에 나눗셈은 명시적인 함수 호출로 이루어질 수 있다.

<표 3> vp_scalar 에 지원되는 연산자 및 함수.

Operator	+,+= -,-= * >, <, >=, <= ==, !=,	
Function	div(x,y)	z = x/y 를 위해 다음과 같이 div 함수를 이용할 수 있다. vp_scalar<8,8> x,y,z; z.div(x,y);

예제 코드:

```
vp_scalar<2,4> x;
vp_scalar<1,4> y;
vp_scalar<2,6> z;
…
z=x*y;
```



(그림 1) 라이브러리 사용 예제와 내부 동작.

라이브러리는 floating-point 값을 vp_scalar 변수에 입력하거나, vp_scalar 를 floating-point 타입으로 변환을 자유롭게 수행 할 수 있도록 되어있다. <표 4>과 같이 vp_scalar 변수에는 floating-point 값을 아무 제한 없이 입력할 수 있으며, 이 때 라이브러리가 자동으로 정밀도를 변환하여 입력하게 된다. vp_scalar 변수를 floating-point 타입으로 변환하는 것은 단순히 casting 연산자를 사용하여 수행할 수 있다.

<표 4> 데이터 타입 변환.

vp_scalar 에 double 값의 입력은 다음과 같이 단순히 = operator 를 이용하면 된다.
vp_scalar<8,8> x; x = 3.141592
vp_scalar 값을 double 변환하는 경우도 단순히 double 혹은 float 으로 casting 하면 된다.
vp_scalar<8,8> x; x = ... printf("%llf\n", (double) x);

Scalar 타입인 vp_scalar 와 함께 복소수를 위한 vp_complex 를 지원한다. vp_complex 는 다음과 같이 실수부와 허수부를 위해 두개의 vp_scalar 변수를 갖고 <표 5>와 같이 연산자 overloading 을 지원할 뿐아니라, 복소수 연산의 편의를 위한 함수들(conjugate, magnitude)이 추가로 제공된다.

```
class vp_complex<>
{
    vp_scalar<> re; // real
    vp_scalar<> im; // image
};
```

<표 5> vp_scalar 에 지원되는 연산자 및 함수.

Operator	+,+= -,-= *	
Function	conj()	해당 변수의 conjugate 값을 반환한다. vp_complex<8,8> x,y; x = y.conj();
	norm()	= re*re + im*im
	div(x,y)	= x/y (y 는 vp_complex, vp_scalar 모두 가능)

3. 라이브러리 사용예

본 장에서는 연구에서 개발한 라이브러리를 통해 통신 알고리즘 중 패킷 검출기를 라이브러리 적용하여 구현하는 것을 예로 소개한다.

<표 6> 패킷 검출기 구현 예제.

Floating-point 연산 코드

```
int packet_detection(
   double *ADC,
   double *Ref)
{
   double sample;
   double cAn;
   double An = 0;
   double Bn = 0;
   double threshold = 10;
   int location=0;
```

```
    for(i=0; i<128; i++) {
       Bn += (sample=ADC[i]).norm();
    }
    while( location<250 && An<= Bn*threshold)    {
       cAn.re = cAn.im = 0;
       for(i=0; i<128; i++) {
          sample = ADC[location+i];
          cAn += sample * Ref[i];
       }
       An = cAn.norm();
       Bn += sample.norm();
       Bn -= (sample=ADC[location-128]).norm();
       location++;
    }
}
```

라이브러리 기반의 코드

```
#define Prec_ADC          6,10
#define Prec_PD_REF       6,10
#define Prec_PD_IN        6,10
#define Prec_PD_cAn       14,10,VPM_Rnd // No saturation
#define Prec_PD_An        14,10,VPM_URnd // Unsigned
#define Prec_PD_Bn        20,10,VPM_URnd // Unsigned

int packet_detection(
   vp_complex<Prec_ADC> *ADC,
   vp_scalar<Prec_PD_REF> *Ref)
{
   vp_complex<Prec_PD_IN> sample;
   vp_complex<Prec_PD_cAn> cAn;
   vp_scalar<Prec_PD_An> An = 0;
   vp_scalar<Prec_PD_Bn> Bn = 0;
   vp_scalar<5> threshold = 10;
   int location=1, i;

.. 이하 동일
```

<표 6>은 패킷 검출기에 대한 일반적인 floating-point 연산 코드와 라이브러리를 이용하여 fixed-point 연산 코드로 구현한 것을 나타내고 있다. 코드를 비교하여 보면, 데이터 타입 선언을 제외하면 floating-point 연산 코드와 동일한 방식으로 코드를 작성할 수 있는 것을 알 수 있다.

<표 7> 컴파일러의 어셈블리 코드 생성 결과.

```
.L5:
        ld.w $r15, ($r0)              ;;memory load
        ld.w $r8, ($r0 + -4)          ;;memory load
        add3 $r7, $r11, $r13          ;;add
        ld.w $r7, ($r7)               ;;memory load
        addi $r0, #8                  ;;constant add
        addi $r11, #4                 ;;constant add
        mac $r9, $r7, $r8, $r9        ;;mac
        mac $r12, $r7, $r15, $r12     ;;mac
        cmpi $r11, #512               ;;compare
        bne .L5                       ;;branch not equal
```

<표 7>는 라이브러리 기반의 코드를 컴파일 하였을 때 생성되는 어셈블리 코드에서 innermost loop 에 해당하는 부분이다(PICO 프로세서의 컴파일러를 사용하였다.[6]). 생성된 어셈블리 코드는 정수형 명령어만을 사용할 뿐만 아니라, MAC(Multiply & Accumulate) 명령어를 사용하는 최적화된 코드임을 알 수 있다.

4. 결론

본 연구에서는 fixed-point 연산 코드를 개발에 어려움을 극복하기 위해 본 연구에서는 코드작성의 편의를 높이고 개발기간을 단축시킬 수 있는 C++ template 기반의 fixed-point 연산 라이브러리를 개발하였다. 본 연구에서 개발한 라이브러리는 floating-point 연산 코드와 fixed-point 연산 코드를 별도로 개발할 필요 없이 하나의 코드를 이용하여 자유로이 연산 정밀도를 지정할 수 있도록 하였다. 개발자는 연산 정밀도 지정을 제외하고는 기존의 floating-point 연산을 이용하는 코드를 작성하는 것과 거의 동일하게 코드를 작성할 수 있도록 되어있어 개발 편의를 높였으며 기존 개발되었던 코드들도 본 연구에서 개발한 라이브러리를 이용하도록 쉽게 수정할 수 있다. 연구를 통해 개발한 C++ template 기반의 fixed-point 연산 라이브러리의 구조와 사용법과, 통신 알고리즘 중 패킷 검출기를 라이브러리 적용하여 구현하는 것을 예로 소개하여 라이브러리의 사용 편의성과 최적화의 이점을 보였다.

참고문헌

[1] H. Keding, M. Willems, M. Coors, and H. Meyr, “FRIDGE: a fixed-point design and simulation environment,” Design, Automation and Test in Europe, Proceedings, 1998, pp. 429-435.

[2] C. Shi and R.W. Brodersen, “Automated fixed-point data-type optimization tool for signal processing and communication systems,” Proceedings of the 41st annual Design Automation Conference, San Diego, CA, USA, 2004, pp. 478-483.

[3] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou, “Smart bit-width allocation for low power optimization in a SystemC based ASIC design environment,” Proceedings of the conference on Design, automation and test in Europe: Proceedings, Munich, Germany, 2006, pp. 618-623.

[4] S. Kim, K.I. Kum, and W. Sung, “Fixed-Point Optimization Utility for C and C Based Digital Signal Processing Programs,” IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II: ANALOG AND DIGITAL SIGNAL PROCESSING, vol. 45, 1998, p. 1455.

[5] L.D. Coster, M. Adé, R. Lauwereins, and J. Peperstaraete, “Code generation for compiled bit-true simulation for DSP application,” Proceedings of the 11th international symposium on System synthesis, Hsinchu, Taiwan, China, 1998, pp. 9-14.

[6] Jiho Chu, Yeoul Na, Jongwook Shoh, Moongi Seok, Jungwon Kang, and Seon Wook Kim, Implementation and Verification of PICO Processor, 제 16 회 반도체 학술대회, CDC-20, 2009 년 2 월 18 일-20 일.