Application Report
SPRA617A - November 2000

ETSI Math Operations in C for the TMS320C62x

Richard Scales *Digital Signal Processing Solutions*

ABSTRACT

Many standard vocoders follow the European Telecommunications Standards Institute (ETSI) for all math operations. One of the purposes of the ETSI math functions is to standardize all math operations into a set of function calls that can be reused by many different vocoders now and in the future.

The Global Systems for Mobile Communications (GSM) standard requires vocoders that follow the ETSI standard. C code available for GSM includes a function for each math operation and a function call to that function each time that math operation is performed. Each math function can be mapped to one or more DSP instructions. Obviously, an actual function all for every math operation is undesirable for performance reasons. The intent of providing these functions is to provide the engineer with a clear specification of all fixed-point math functionality.

When porting this C code to a particular DSP, an engineer will typically replace each math function with one or more DSP instructions. This is usually done when porting the vocoder to native assembly language of the DSP by hand. Although this is a precise method for implementing a vocoder in assembly, it can be very time consuming. The TMS320C62x™ compiler provides a way to avoid writing all of the code in hand coded assembly. By using C instrinsics, vocoder code is quickly and easily optimized for high performance. Intrinsics are special functions that map directly to inlined C62x™ instructions.

Contents

List of Examples

1 Design Problem

Many standard vocoders follow the European Telecommunications Standards Institute (ETSI) for all math operations. One of the purposes of the ETSI math functions is to standardize all math operations into a set of function calls that can be reused by many different vocoders now and in the future.

The Global Systems for Mobile Communications (GSM) standard requires vocoders that follows the ETSI standard. C code available for GSM includes a function for each math operation and a function call to that function each time that math operation is performed. Each math function can be mapped to one or more DSP instructions. Obviously, an actual function call for every math operation is undesirable for performance reasons. The intent of providing these functions is to provide the engineer with a clear spec of all fixed-point math functionality.

When porting this C code to a particular DSP, an engineer will typically replace each math function with one or more DSP instructions. This is usually done when porting the vocoder to native assembly language of the DSP by hand. Although this is a precise method for implementing a vocoder in assembly, it can be very time consuming.

2 Solution

The TMS320C62x compiler provides a way to avoid writing all of the code in hand coded assembly. By using C intrinsics, vocoder code is quickly and easily optimized for high performance. Intrinsics are special functions that map directly to inlined C62x instructions.

- Intrinsics are specified with a leading underscore and are accessed by calling them as you do a function.
- All ETSI specific math operations, as well as others, which are not easily expressible in C code are supported as intrinsics in the C compiler.

An example of a math operation not easily expressible in C is the saturate add.

Example 1. Saturated Add Without Intrinsics

```
int sadd(int a, int b)
{
    int result;

    result = a + b;
    if (((a ^ b) & 0x80000000) == 0){
        if ((result ^ a) & 0x80000000){
            result = (a < 0) ? 0x80000000 : 0x7fffffff;
        }
    }
    return (result);
}
```

This demonstrates how difficult simple DSP operations can be to represent in C. Not only that, but the resulting code generated by the compiler will most likely be very inefficient.

On the C62x, the same operation can be represented by single C intrinsic.

Example 2. Saturated Add With Instrinsics

```
result = _sadd(a, b);
```

The _sadd intrinsic looks like a function call in C but actually maps directly to the C62x SADD instruction and will not result in a function call.

In order to aide the engineer in writing C62x vocoder code, the following list of #define statements can be included as a header file in every ETSI standard vocoder file to efficiently replace all function calls with high performance, efficient C62x instructions.

Example 3. ETSI Functions Mapped To C62x Intrinsics

```
#include <stdlib.h>
#include <linkage.h>

extern int Overflow;
extern int Carry;

_IDECL int L_add_c (int, int);
_IDECL int L_sat   (int);
_IDECL short div_s (short, short);

/*****************************************************************************/
/* Macros for GSM operations                                                 */
/*****************************************************************************/
#define L_add(a,b)     (_sadd((a),(b)))             /* int sat addition       */
#define L_sub(a,b)     (_ssub((a),(b)))             /* int sat subtract       */
#define L_sub_c(a,b)   L_add_c((a),~(b))            /* integer subtraction    */
#define L_negate(a)    (_ssub(0,(a)))               /* integer negation       */
#define L_deposit_h(a) ((a)<<16)                    /* put short in upper 16  */
#define L_deposit_l(a) ((int)(a))                   /* put short in lower 16  */
#define L_abs(a)       (abs(a))                     /* int absolute value     */
#define L_mult(a,b)    (_smpy((a),(b)))             /* short sat mpy => 32    */
#define L_mac(a,b,c)   (_sadd((a),L_mult(b, c)))    /* saturated mpy & accum  */
#define L_macNs(a,b,c) L_add_c((a),L_mult(b,c))     /* mpy & accum w/o saturat*/
#define L_msu(a,b,c)   (_ssub((a),L_mult(b,c)))     /* saturated mpy & sub    */
#define L_msuNs(a,b,c) L_sub_c(a,L_mult(b,c))       /* mpy & sub w/o saturate */
#define L_shl(a,b)     ((b) < 0 ? (a) >> (–b) : _sshl((a),(b)))
#define L_shr(a,b)     ((b) < 0 ? _sshl((a),(–b)) : (a) >> (b))
#define L_shr_r(a,b)   (L_shr((a),(b)) + ((b)>0 && (((a) & (1<<((b)–1))) != 0)))
#define abs_s(a)       (abs((a)<<16)>>16)           /* short absolute value   */
#define add(a,b)       (_sadd((a)<<16,(b)<<16)>>16) /* short sat add          */
#define sub(a,b)       (_ssub((a)<<16,(b)<<16)>>16) /* short sat subtract     */
#define extract_h(a)   ((unsigned)(a)>>16)          /* extract upper 16 bits  */
#define extract_l(a)   ((a)&0xffff)                 /* extract lower 16 bits  */
#define round(a)       extract_h(_sadd((a),0x8000)) /* round                  */
#define mac_r(a,b,c)   (round(L_mac(a,b,c)))        /* mac w/ rounding        */
#define msu_r(a,b,c)   (round(L_msu(a,b,c)))        /* msu w/ rounding        */
#define mult_r(a,b)    (round(L_mult(a,b)))         /* sat mpy w/ round       */
#define mult(a,b)      (L_mult(a,b)>>16)            /* short sat mpy upper 16 */
#define norm_l(a)      (_norm(a))                   /* return NORM of int     */
#define norm_s(a)      (_norm(a)–16)                /* return NORM of short   */
#define negate(a)      (_ssub(0, ((a)<<16)) >> 16)  /* short sat negate       */
#define shl(a,b)       ((b) < 0 ? (a) >> (–b) : (_sshl((a),(b+16))>>16))
#define shr(a,b)       ((b) < 0 ? (_sshl((a),(–b+16))>>16) : ((a) >> (b)))
#define shr_r(a,b)     ((b) < 0 ? (_sshl((a),(–b+16))>>16) : (b)==0 ? (a) : \
                                    ((a)+(1<<((b)–1))) >> (b))
```

```
#ifdef _INLINE
/*****************************************************************************/
/* Integer (32–bit) add with carry and overflow testing.                     */
/*****************************************************************************/
static inline int L_add_c (int L_var1, int L_var2)
{
    unsigned L_test    = L_var1 + L_var2 + Carry;
    unsigned int uv1   = L_var1;
    unsigned int uv2   = L_var2;

    Overflow  = ((~(uv1 ^ uv2)) & (uv1 ^ L_test)) >> 31;
    Carry     = ((~L_test & (uv1 | uv2)) | (uv1 & uv2)) >> 31;

    return L_sat(L_test);
}
/*****************************************************************************/
/* Saturate any result after L_addc or L_sub_c if overflow is set.           */
/*****************************************************************************/
static inline int L_sat (int L_var1)
{
    int cin = Carry;
    return !Overflow ? (Carry = Overflow = 0, L_var1) :
                       (Carry = Overflow = 0, 0x7fffffff+cin);
}
/*****************************************************************************/
/* Short (16–bit) divide.                                                    */
/*****************************************************************************/
static inline short div_s (short var1, short var2)
{
    int          iteration;
    unsigned int var1int;
    int          var2int;
    if (var1 == 0)    return 0;
    if (var1 == var2) return 0x7fff;
    var1int = var1 << 16;
    var2int = var2 << 16;

    for (iteration = 0; iteration < 16; iteration++)
        var1int = _subc(var1int,var2int);
    return var1int & 0xffff;
}
#endif
```

Notice that not all math operations need to be represented as intrinsics. Simple operations like shifting and addition are easily represented in their native C form, (“>>” and “+”). The C62x compiler will map all of the typical C type operations to the correct C62x instructions automatically.

Example 3 also shows some double precision functions which were left in the form of actual function calls but will be statically inlined to avoid the call overhead. These were left as functions for the sake of clarity because they typically involve more ’C6x instructions.

By using the above #define statements and static inline functions vocoder performance of the C62x C compiler is greatly improved. For more information on C intrinsics refer to the *TMS320C6x Optimizing C Compiler User’s Guide* (SPRU187). For more information on further C code optimizations refer to the *TMS320C6x Programmer’s Guide* (SPRU198).