TMS320 DSP
Number xx.0
DESIGNER'S NOTEBOOK
TEXAS INSTRUMENTS

ETSI Math Operations in C for the 'C62xx

Contributed by Richard Scales

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 Communictions (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.

Solution

The TMS320C62xx 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 in-lined 'C62xx 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 C62xx, 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 C62xx SADD instruction and will not result in a function call.

In order to aide the engineer in writing C62xx 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 C62xx instructions.

Example 3. ETSI Functions Mapped To C62xx Intrinsics

```
/* Constants and Globals  */
extern Flag Overflow;
extern Flag Carry;
#define MAX_32 (Word32)0x7fffffffL
#define MIN_32 (Word32)0x80000000L
#define MAX_16 (Word16)0x7fff
#define MIN_16 (Word16)0x8000

/* Functions defined with Intrinsics */
#define L_add(a,b) (_sadd(a,b))
#define L_sub(a,b) (_ssub(a,b))
#define L_mult(a,b) (_smpy(a,b))
#define L_mac(a,b,c) (_sadd(a,_smpy(b,c)))
#define L_msu(a,b,c) (_ssub(a,_smpy(b,c)))
#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 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 abs_s(a) (abs(a<<16)>>16)
#define add(a,b) (_sadd(a<<16, b<<16)>>16)
#define sub(a,b) (_ssub(a<<16, b<<16)>>16)
#define mult(a,b) (_smpy(a,b)>>16)
#define .....
#define mult_r(a,b) (_sadd(_smpy(a,b),0x8000L)>>16)
#define mac_r(a,b,c) (_sadd(_sadd(a,_smpy(b,c)),0x8000L)>>16)
#define msu_r(a,b,c) (_sadd(_ssub(a,_smpy(b,c)),0x8000L)>>16)
#define extract_h(a) ((unsigned)(a)>>16)
#define extract_l(a) (a&0xffff)
#define L_deposit_h(a) (a<<16)
#define L_deposit_l(a) (a)
#define L_abs(a) abs(a)
#define norm_s(a) (_norm(a)-16)
```

```
#define norm_l(a) (_norm(a))
#define L_negate(a) _ssub(0,(a))
#define L_sat(a) a

static inline Word16 negate (Word16 var1)
{
   Word16 var_out;

   var_out = (var1 == MIN_16) ? MAX_16 : -var1;
   return (var_out);
}

#define L_add_c(a,b) (a+b)
#define L_sub_c(a,b) (a-b)
#define L_msuNs(a,b,c) ((a)-_smpy((b),(c)))

static inline Word32 L_add_c (Word32 L_var1, Word32 L_var2)
{
   Word32 L_var_out;
   Word32 L_test;
   Flag carry_int = 0;

   L_var_out = L_var1 + L_var2 + Carry;
   L_test = L_var1 + L_var2;

   if ((L_var1 > 0) && (L_var2 > 0) && (L_test < 0)) {
      Overflow = 1;
      carry_int = 0;
      }
   else {
      if ((L_var1 < 0) && (L_var2 < 0)){
         if (L_test >= 0){
            Overflow = 1;
            carry_int = 1;
            }
         else{
            Overflow = 0;
            carry_int = 1;
            }
         }
      else{
         if (((L_var1 ^ L_var2) < 0) && (L_test >= 0)){
            Overflow = 0;
            carry_int = 1;
            }
         else{
            Overflow = 0; .....
            carry_int = 0;
           }
         }
      }
   if (Carry){
      if (L_test == MAX_32){
         Overflow = 1;
         Carry = carry_int;
         }
      else{
         if (L_test == (Word32) 0xFFFFFFFFL){
            Carry = 1;
            }
```

```
            else{
                Carry = carry_int;
                }
            }
        }
    else{
        Carry = carry_int;
        }
     return (L_var_out);
   }

#define L_macNs(a,b,c) L_add_c(a,_smpy(b,c))

static inline Word16 div_s (Word16 var1, Word16 var2)
{
    Word16 iteration;
    unsigned int var1int;
    int var2int;

    if(var1 == var2) return(MAX_16);

    var1int = var1 << 16;
    var2int = var2 << 16;

    if (var1 == 0){
        printf("var1 = 0");
        return(0);
        }
    else{
        for (iteration = 0; iteration < 16; iteration++){
            var1int = _subc(var1int,var2int); }
        }
 return (var1int & 0xffff);
}


/* Double precision operations */
static inline void L_Extract (Word32 L_32, Word16 *hi, Word16 *lo)
{
    *hi = extract_h (L_32);
    *lo = extract_l (L_msu (L_shr (L_32, 1), *hi, 16384));
    return;
}
static inline Word32 L_Comp (Word16 hi, Word16 lo)
{
    Word32 L_32;

    L_32 = L_deposit_h (hi);
    return (L_mac (L_32, lo, 1));       /* = hi<<16 + lo<<1 */
}

static inline Word32 Mpy_32 (Word16 hi1, Word16 lo1, Word16 hi2,
Word16 lo2)
{
    Word32 L_32;

    L_32 = L_mult (hi1, hi2);
    L_32 = L_mac (L_32, mult (hi1, lo2), 1);
    L_32 = L_mac (L_32, mult (lo1, hi2), 1);
```

```
    return (L_32);
}

static inline Word32 Mpy_32_16 (Word16 hi, Word16 lo, Word16 n)
{
    Word32 L_32;

    L_32 = L_mult (hi, n);
    L_32 = L_mac (L_32, mult (lo, n), 1);

    return (L_32);
}

static inline Word32 Div_32 (Word32 L_num, Word16 denom_hi,
Word16 denom_lo)
{
    Word16 approx, hi, lo, n_hi, n_lo;
    Word32 L_32;

    /* First approximation: 1 / L_denom = 1/denom_hi */

    approx = div_s ((Word16) 0x3fff, denom_hi);

    /* 1/L_denom = approx * (2.0 - L_denom * approx) */

    L_32 = Mpy_32_16 (denom_hi, denom_lo, approx);

    L_32 = L_sub ((Word32) 0x7fffffffL, L_32);

    L_Extract (L_32, &hi, &lo);

    L_32 = Mpy_32_16 (hi, lo, approx);

    /* L_num * (1/L_denom) */

    L_Extract (L_32, &hi, &lo);
    L_Extract (L_num, &n_hi, &n_lo);
    L_32 = Mpy_32 (n_hi, n_lo, hi, lo);
    L_32 = L_shl (L_32, 2);

    return (L_32);
}
```

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 C62xx compiler will map all of the typical C type operations to the correct C62xx 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 C62xx C compiler is greatly improved.

For more information on C intrinsics refer to the TMS320C6x Optimizing C Compiler User's Guide. For more information on further C code optimizations refer to the TMS320C6x Programmer's Guide.