TEXAS INSTRUMENTS

*Application Report*
*SPRA707 - October 2000*

# *TMS320C6000 Integer Division*

*Yao-Ting Cheng* *C6000 Applications*

**ABSTRACT**

This application report gives an explanation on the implementation of division in the TMS320C6x™ DSP. The scheme is slightly different from the TMS320C5x™ and TMS320C54x™ DSPs. That is because C6x™ provides some unique instructions that the user can take advantage of to implement division more efficiently. This application report also lists the fully optimized assembly subroutine that can be found in TI's public web site.

**Contents**

## 1 Design Problem

How to implement division in C6x?

## 2 Solution

This application report gives an explanation on the implementation of division in C6x. The scheme is slightly different from the C5x™ and C54x™. That is because C6x provides some unique instructions that the user can take advantage of to implement division more efficiently. This application report also lists the fully optimized assembly subroutine that can be found in TI's public web site.

Since division is a rare function in DSP, the C6x does not provide a single cycle divide instruction. In fact, the hardware to implement division is expensive. Similar to the other Texas Instruments DSP family, C6x does have a single cycle 1-bit divide instruction: conditional subtraction or SUBC. Let's review how it works in C5x/C54x first. The syntax is "`SUBC src`" and the action taken by SUBC in C5x/C54x is

```
IF ((ACC–(src<<15))>= 0)
      ((ACC–(src<<15)))<<1+1 → ACC
ELSE  ACC<<1 → ACC
```

The 16-bit numerator is stored in the accumulator low byte (ACCL) and the accumulator high byte (ACCH) is zero-filled. The denominator (src) is in data memory. The SUBC instruction is executed 16 times for 16-bit division. After completion of the last SUBC instruction, the quotient of the division is in the ACCL and the remainder is in the ACCH. The SUBC instruction assumes that the denominator and the numerator are both positive. The denominator is not sign extended. The numerator, in the ACCL, must initially be positive and must remain positive following the ACC shift, which occur in the first portion of the SUBC execution.

TMS320C6x, TMS320C5x, TMS320C54x, C6x, C5x, and C54x are trademarks of Texas Instruments.

As an example, assume a 4-bits machine is used. The numerator is decimal 11 (1011b), which is in the 8-bit ACC. The denominator is 3 (0011b) in the memory. Apparently, four SUBC need to be calculated since it shift denominator left 3 bits at the very beginning.

1. 1st SUBC:

```
                      0
0001 1000 | 0000 1011
            0000 0000
            ---------
            0000 1011

src<<3 | 0001 1000 |
Since   ACC < (src<<3)
So      ACC<<1 → ACC | 0001 0110 |
```

2. 2nd SUBC:

```
                     00
0001 1000 | 0001 0110
            0000 0000
            ---------
            0001 0110

src<<3 | 0001 1000 |
Since   ACC < (src<<3)
So      ACC<<1 → ACC | 0010 1100 |
```

3. 3rd SUBC:

```
                    001
0001 1000 | 0010 1100
            0001 1000
            ---------
            0001 0100

src<<3 | 0001 1000 |
Since   ACC > (src<<3)
So      (ACC–(src1<<3))<<1+1 → ACC | 0010 1001 |
```

4. 4th SUBC:

```
                   0011
0001 1000 | 0010 1001
            0001 1000
            ---------
            0001 0001

src<<3 | 0001 1000 |
Since   ACC > (src<<3)
So      (ACC–(src1<<3))<<1+1 → ACC | 0010 0011 |
```

For each left shift by SUBC, the quotient is recorded in the LSB of ACC. The remainder is calculated at the last subtraction. The number illustrated in red color above shows the numerator is conditional subtracted by the pre-shifted denominator and then the result is shifted left. Therefore the quotient is 3 (0011b) and the remainder is 2 (0010b) after the final shift is performed.

This is how to do it by hand:

```
      011   ← Quotient
 11 ┌1011
     00     ← 1st SUBC
    ────
     101
      11    ← 2nd SUBC
    ────
      101
       11   ← 3rd SUBC
    ────
       10
```

Denominator 11 is not to 10 for the first SUBC, so a conditional subtraction is performed. That is to shift denominator right one bit and do the SUBC again. This time 11 is to 101 so that a subtraction is taken and the result becomes the new nominator. Then shift the denominator right one bit again to do the third SUBC to get the final result. Can you tell the difference between the two divisions? In the case of SUBC in C5x/C54x, you need to execute SUBC exactly 4 times. In the case of hand division, you need only SUBC 3 times. You also need to shift denominator right totally two times during the process to complete the division. But, why? That is because denominator is shifted left two bits instead of three bits in C5x/C54x and therefore aligned to numerator. SUB needs to be done “shift” plus one times. How does SUBC work in C6x? Similar to C5x/C54x, instead of right shift denominator for subtraction during the division, numerator is shifted left. The syntax of SUBC in C6x is “SUBC  (.unit) src1, src2, dst” and its action is

```
IF (cond) {
     IF (src1 >= src2)
          ((src1−src2)<<1)+1 → dst
     ELSE src1<<1 → dst
}
ELSE nop
```

Look at the previous example again to see how SUBC works in C6x. Assume numerator is in src1 and denominator is aligned to numerator and it is put in src2. Also let dst be src1.

1. 1st SUBC

```
           0
   1100 ┌1011
         0000
        ─────
         1011
```

Since src1<src2

So    src1<<1 → dst

src1 = (1011)<<1 = $\boxed{10110}$

2. 2nd SUBC

```
            01
      ---------
1100 | 10110
        1100
     ---------
        1010
```

```
Since src1 >= src2
So    (src1–src2)<<1)+1 → dst
src1 = (1010)<<1+1 = 10101
```

3. 3rd SUBC

```
           011
      ---------
1100 | 10101
        1100
     ---------
        1001
```

```
Since src1 >= src2
So    (src1–src2)<<1)+1 → dst
src1 = (1001)<<1+1 = 10011
```

Notice that the quotient is preserved in the last three bits of dst. The remainder is calculated at the last subtraction in red number, which is the same as hand division. The above simplified derivation shows SUBC in C6x works more efficiently based the assumption of denominator aligned. Then, how many bits of left shift is needed in order to align denominator? Actually, C6x provides a very useful instruction "LMBD" to test it. Simply use C intrinsic _LMBD as

`Shift = _LMBD(1,denominator) – _LMBD(1,numerator).`

The syntax of assembly LMBD is "`LMBD  (.unit) src1,src2,dst`". The LSB of the src1 operand determines whether to search for a leftmost 1 or 0 in src2. The number of bits to the left of the first 1 or 0 when searching for a 1 or 0, respectively, is placed in dst. Finally, with testing some special cases such as divide by zero, here is the unsigned division algorithm in C.

```
unsigned int udiv(unsigned int num, unsigned int den)
{
    int i, shift;

    if (den > num) return (0);
    if (num == 0)  return (0);
    if (den == 0)  return (–1);

    shift = _lmbd(1, den) – _lmbd(1, num);
    den <<= shift;                   /* align denominator */

    for (i=0; i<=shift; i++)
        num = _subc(num, den);

    return (num << (32–(shift+1))) >> (32–(shift+1));
                                     /* extract quotient */
}
```

If a remainder is required, simply shift “num” right (shift+1) bits to get the result. Now, how to implement a signed division. Actually, there are lots of ways to do it. Compare the signs of the input operands. If they are alike, plan a positive quotient, otherwise plan to negate the quotient. We can strip the signs of the numerator and denominator just by shifting their MSB bit right to the position of LSB. Then XOR them to get the sign for quotient. Perform the unsigned division and attach the proper sign based on the comparison of the inputs to the quotient. A signed division C subroutine for C6x is listed as below.

```
int sdiv(int num, int den)
{
    int i, shift, sign;
    sign = (num>>31) ^ (den>>31);  /* test the sign of inputs */

    num = _abs(num);
    den = _abs(den);

    if (den > num) return (0);
    if (num == 0)  return (0);
    if (den == 0)  return (–1);

    shift = _lmbd(1, den) – _lmbd(1, num);
    den <<= shift;                 /* align denominator */

    for (i=0; i<=shift; i++)
        num = _subc(num, den);

    num = _extu(num, (32–(shift+1)), ((32–(shift+1)));
                                   /* unsigned extraction */
    if (sign) return (–num);   /* attach sign back to quotient */
    else return (num);
}
```

Finally, the fully hand-optimized codes for both signed and unsigned divisions are listed below. They also can be found in the web site from Texas Instruments. For a 32-bit unsigned division, the cycle time C6x takes is around 18~42 depending on how many bits the denominator needs to be aligned. For a 32-bit signed division, C6x takes 16~41 cycles that is less than unsigned division. It is because the sign bit is exclusive in the process of the bit alignment for sign division. Actually, the division function _divi or _divu and remainder function _remi are automatically called from C6x C library when the user uses the operator “/” and “%” respectively. The cycles will be a little bit more than that of the following hand-optimized subroutines. Also notice that the division and remainder are two separate operations for C6x C compiler. The subroutine listed below returns the quotient and remainder in a structure and can be called by a C main program.

### Example 1. Unsigned Division Subroutine

```
*===============================================================================
*
*       TEXAS INSTRUMENTS, INC.
*       DIVMODU32 (32 bits unsigned division and modulo)
*       Revision Date:  07/15/97
*
*       USAGE
*               This routine is C Callable and can be called as:
*
*               struct  divmodu divmodu32(unsigned int a, unsigned int b);
*
*               a --- unsigned numerator
*               b --- unsigned denominator
*
*               If routine is not to be used as a C callable function then
*               you need to initialize values for all of the values passed
*               as these are assumed to be in registers as defined by the
*               calling convention of the compiler, (refer to the C compiler
*               reference guide).
*
*       C CODE
*               This is the C equivalent of the assembly code.  Note that
*               the assembly code is hand optimized and restrictions may
*               apply.
*
*
*               struct  divmodu {
*                       unsigned int div;
*                       unsigned int mod;
*               };
*
*
*               struct  divmodu divmodu32(unsigned int a, unsigned int b)
*               {
*                       struct divmodu tmp;
*
*                       tmp.div = a / b;
*                       tmp.mod = a % b;
*
*                       return tmp;
*               }
*
*       DESCRIPTION
*               This routine divides two unsigned 32 bit values and returns
*               their quotient and remainder.  The inputs are unsigned 32-bit
*               numbers, and the result is a unsigned 32-bit number.
*
*       TECHNIQUE
*               The loop is executed at least 6 times.  In the loop, the
*               conditional subtract divide step (SUBC) is block from doing
*               extraneous executions.  In short, the SUBC instruction
*               is conditional and will not necessarily be executed.
*
*       MEMORY NOTE
*               No memory bank hits under any conditions.
*
*       CYCLES
*               Minimum execution time -> 18 cycles
*               Maximum execution time -> 42 cycles
*
*===============================================================================
        .global _divmodu32
        .text

_divmodu32:

*** BEGIN Benchmark Timing ***
B_START:
        LMBD    .L2X    1,      A4,     B1      ; mag_num = lmbd(1, num)
||      LMBD    .L1X    1,      B4,     A1      ; mag_den = lmbd(1, den)
||      MVK     .S1     32,     A0              ; const 32
||      ZERO    .D1     A8                      ; first_div = 1
        CMPGTU  .L1X    B4,     A4,     A1      ; zero = (den > num)
||      SUB     .L2X    A1,     B1,     B0      ; i = mag_den – mag_num
||      MV      .D1     A4,     A5              ; save num
||[!B1] MVK     .S1     1,      A8              ; if (num32) first_div = 1
        SHL     .S2     B4,     B0,     B4      ; den <<= i
||[B1]  ADD     .D2     B0,     1,      B0      ; if (!num32) i++
||      MV              B0,     A6
        CMPGTU  .L2X    B4,     A4,     B2      ; gt = den > num
||      SUB     .L1X    A0,     B0,     A0      ; qs = 32 – i
||      SHL     .S1     A8,     A6,     A8      ; first_div <<= i
||      B       .S2     LOOP                    ;
||[B1]  MPY     .M2     B2,     0,      B2      ; num32 && gt
        ADD     .L1X    0,      B0,     A2
||[B2]  MV      .D2     B2,     B1              ; !(num32 && !gt)
||[B2]  SHRU    .S1     A8,     1,      A8      ; first_div >>= 1
||      B       .S2     LOOP                    ;
  [B2]  SHRU    .S2     B4,     1,      B4      ; if (num32 && gt) den >> 1
||[!B1] SUB     .L1X    A4,     B4,     A4      ; if (num32 && !gt) num –= den
||[B0]  SUB     .D2     B0,     1,      B0      ; i––
||      B       .S1     LOOP                    ;
  [!B1] SHRU    .S2     B4,     1,      B4      ; if (num32 && !gt) den >> 1
||[B2]  SUB     .L1X    A4,     B4,     A4      ; if (num32 && gt) num –= den
||      CMPLT   .L2     B0,     6,      B2      ; check for neg. loop counter
||      SUB     .D2     B0,     6,      B1      ; generate loop counter
||      B       .S1     LOOP                    ;
  [B2]  ZERO    .L2     B1                      ; zero negative loop counter
||[A2]  SUBC    .L1X    A4,     B4,     A4      ; num = subc(num, den)
||      B       .S2     LOOP                    ;
LOOP:
  [B0]  SUBC    .L1X    A4,     B4,     A4      ; num = subc(num, den)
||[B0]  SUB     .L2     B0,     1,      B0      ; i––
||[B1]  SUB     .D2     B1,     1,      B1      ; i––
||[B1]  B       .S1     LOOP                    ; for
;end of LOOP
        ADD     .L2     A3,     4,      B7      ; address for mod result
||[!A1] SHL     .S1     A4,     A0,     A6      ; q = num << qs
||[A1]  MPY     .M1     0,      A6,     A6      ; if (zero) q = zero
||      B       .S2     B3
  [!A1] SHRU    .S1     A6,     A0,     A6      ; q = num >> qs
||[A1]  MV      .L1     A5,     A2              ; if (zero) mod = num
||      MV              A8,     B5              ;
        ADD     .L2X    A6,     B5,     B8      ;
||[!A1] SHRU    .S1     A4,     A2,     A2      ; mod = n >> ms
        STW     .D1     B8,     *A3++           ; c[2 * i] = q
||      STW     .D2     A2,     *B7++           ; c[2 * i + 1] = mod
B_END:
*** END Benchmark Timing ***

        NOP     2
```

**Example 2. Signed Division Subroutine**

```
*==============================================================================
*
*       TEXAS INSTRUMENTS, INC.
*       DIVMOD32 (signed division)
*       Revision Date:  07/09/97
*
*       USAGE
*               This routine is C Callable and can be called as:
*
*               struct  divmod divmod32(int a, int b);
*
*               a ––– numerator
*               b ––– denominator
*
*               If routine is not to be used as a C callable function then
*               you need to initialize values for all of the values passed
*               as these are assumed to be in registers as defined by the
*               calling convention of the compiler, (refer to the C compiler
*               reference guide).
*
*       C CODE
*               This is the C equivalent of the assembly code.  Note that
*               the assembly code is hand optimized and restrictions may
*               apply.
*
*               struct  divmod {
*                       int div;
*                       int mod;
*               };
*
*               struct  divmod divmod32(int a, int b)
*               {
*                       struct divmod tmp;
*
*                       tmp.div = a / b;
*                       tmp.mod = a % b;
*
*                       return tmp;
*               }
*
*       DESCRIPTION
*               This routine divides two 32 bit values and returns their
*               quotient and remainder.  The inputs are 32–bit numbers, and
*               the result is a 32–bit number.
*
*       TECHNIQUE
*               The loop is executed at least 6 times.  In the loop, the
*               conditional subtract divide step (SUBC) is block from doing
*               extraneous executions.  In short, the SUBC instruction
*               is conditional and will not necessarily be executed.
*
*
*       MEMORY NOTE
*               No memory bank hits under any conditions.
*
*       CYCLES
*               Minimum execution time –> 16 cycles
*               Maximum execution time –> 41 cycles
*
*==============================================================================
        .global _divmod32
        .text
_divmod32:
*** BEGIN Benchmark Timing ***
B_START:
        SHRU    .S1     A4,     31,     A1      ; neg_num = num < 0
||      CMPLT   .L2     B4,     0,      B1      ; neg_den = den < 0
||      MV      .D1     A4,     A5              ; copy num
  [A1]  NEG     .L1     A4,     A4              ; abs_num = abs(num)
||[B1]  NEG     .S2     B4,     B4              ; abs_den = abs(den)
||      MPY     .M1     –1,     A1,     A6      ; copy neg_num
||      MPY     .M2     –1,     B1,     B9      ; copy neg_den
||      B       .S1     LOOP                    ;

        NORM    .L1     A4,     A2              ; mag_num = norm(abs_num)
||      NORM    .L2     B4,     B2              ; mag_den = norm(abs_den)
||      B       .S1     LOOP                    ;
||      ADD     .S2X    A3,     4,      B8      ; address for mod result
        CMPGTU  .L1X    B4,     A4,     A1      ; zero = (abs_den > abs_num)
||      SUB     .L2X    B2,     A2,     B0      ; i = mag_den – mag_num
||      MVK     .S1     31,     A0              ;
||      B       .S2     LOOP                    ;
        SHL     .S2     B4,     B0,     B4      ; abs_den <<= i
||      CMPLT   .L2     B0,     6,      B2      ; check for neg. loop counter
||      SUB     .D2     B0,     6,      B1      ; generate loop counter
||      SUB     .L1X    A0,     B0,     A0      ; qs = 31 – i
||      B       .S1     LOOP                    ;
  [B2]  ZERO    .L2     B1                      ; zero negative loop counter
||      SUBC    .L1X    A4,     B4,     A4      ; abs_num=subc(abs_num, abs_den)
||      ADD     .D2     1,      B0,     B2      ; ms = i + 1
||      B       .S2     LOOP                    ;
LOOP:
  [B0]  SUBC    .L1X    A4,     B4,     A4      ; abs_num=subc(abs_num, abs_den)
||[B0]  SUB     .L2     B0,     1,      B0      ; i––
||[B1]  SUB     .D2     B1,     1,      B1      ; i––
||[B1]  B       .S1     LOOP                    ; for
;end of LOOP
  [!A1] SHRU    .S2X    A4,     B2,     B1      ; mod = n >> ms
||[!A1] SHL     .S1     A4,     A0,     A4      ; q = abs_num << qs
||[A1]  MPY     .M1     0,      A4,     A4      ; if (zero) q = zero
||      XOR     .L1X    A6,     B9,     A2      ; neg_q = neg_num ^ neg_den
  [!A1] SHRU    .S1     A4,     A0,     A4      ; q = abs_num >> qs
||[A1]  MV      .L2X    A5,     B1              ; if (zero) mod = num
||[!A1] MV      .L1     A6,     A1              ; \ neg_mod = !zero && neg_num
||[A1]  ZERO    .D1     A1                      ; /
  [A2]  NEG     .L1     A4,     A4              ; if (neg_q) q = –q
||[A1]  NEG     .L2     B1,     B1              ; if (neg_mod) mod = –mod
||      B       .S2     B3                      ; return
        STW     .D1     A4,     *A3++           ; c[2 * i] = c_tmp.div
||      STW     .D2     B1,     *B8++           ; c[2 * i + 1] = c_tmp.mod
B_END:
*** END Benchmark Timing ***
        NOP     4
```