Division Exceptions B

B.1 DIVISION FUNDAMENTALS

The ADSP-2100 family processors' instruction set contains two instructions for implementing a non-restoring divide algorithm. These instructions take as their operands twos-complement or unsigned numbers, and in sixteen cycles produce a truncated quotient of sixteen bits. For most numbers and applications, these primitives produce the correct results. However, there are certain situations where results produced will be off by one LSB. This appendix documents these situations, and presents alternatives for producing the correct results.

Computing a 16-bit fixed point quotient from two numbers is accomplished by 16 executions of the DIVQ instruction for unsigned numbers. Signed division uses the DIVS instruction first, followed by fifteen DIVQs. Regardless of which division you perform, both input operands must be of the same type (signed or unsigned) and produce a result of the same type.

These two instructions are used to implement a conditional add/subtract, non-restoring division algorithm. As its name implies, the algorithm functions by adding or subtracting the divisor to/from the dividend. The decision as to which operation is perform is based on the previously generated quotient bit. Each add/subtract operation produces a new partial remainder, which will be used in the next step.

The phrase non-restoring refers to the fact that the final remainder is not correct. With a restoring algorithm, it is possible, at any step, to take the partial quotient, multiply it by the divisor, and add the partial remainder to recreate the dividend. With this non-restoring algorithm, it is necessary to add two times the divisor to the partial remainder if the previously determined quotient bit is zero. It is easier to compute the remainder using the multiplier than in the ALU.

B.1.1 Signed Division

Signed division is accomplished by first storing the 16-bit divisor in an X register (AX0, AX1, AR, MR2, MR1, MR0, SR1, or SR0). The 32-bit dividend must be stored in two separate 16-bit registers. The lower 16-bits must be stored in AY0, while the upper 16-bits can be in either AY1, or AF.

B Division Exceptions

The DIVS primitive is executed once, with the proper operands (ex. DIVS AY1, AX0) to compute the sign of the quotient. The sign bit of the quotient is determined by XORing (exclusive-or) the sign bits of each operand. The entire 32-bit dividend is shifted left one bit. The lower fifteen bits of the dividend with the recently determined sign bit appended are stored in AY0, while the lower fifteen bits of the upper word, with the MSB of the lower word appended is stored in AF.

To complete the division, 15 DIVQ instructions are executed. Operation of the DIVQ primitive is described below.

B.1.2 Unsigned Division

Computing an unsigned division is done like signed division, except the first instruction is not a DIVS, but another DIVQ. The upper word of the dividend must be stored in AF, and the AQ bit of the ASTAT register must be set to zero before the divide begins.

The DIVQ instruction uses the AQ bit of the ASTAT register to determine if the dividend should be added to, or subtracted from the partial remainder stored in AF and AY0. If AQ is zero, a subtract occurs. A new value for AQ is determined by XORing the MSB of the divisor with the MSB of the dividend. The 32-bit dividend is shifted left one bit, and the inverted value of AQ is moved into the LSB.

B.1.3 Output Formats

As in multiplication, the format of a division result is based on the format of the input operands. The division logic has been designed to work most efficiently with fully fractional numbers, those most commonly used in fixed-point DSP applications. A signed, fully fractional number uses one bit before the binary point as the sign, with fifteen (or thirty-one in double precision) bits to the right, for magnitude.

If the dividend is in M.N format (M bits before the binary point, N bits after), and the divisor is O.P format, the quotient's format will be (M-O+1).(N-P-1). As you can see, dividing a 1.31 number by a 1.15 number will produce a quotient whose format is (1-1+1).(31-15-1) or 1.15.

Before dividing two numbers, you must ensure that the format of the quotient will be valid. For example, if you attempted to divide a 32.0 number by a 1.15 number the result would attempt to be in (32-1+1).(0-15-1) or 32.-16 format. This cannot be represented in a 16-bit register!

Division Exceptions B

In addition to proper output format, you must insure that a divide overflow does not occur. Even if a division of two numbers produces a legal output format, it is possible that the number will overflow, and be unable to fit within the constraints of the output. For example, if you wished to divide a 16.16 number by a 1.15 number, the output format would be (16-1+1).(16-15-1) or 16.0 which is legal. Now assume you happened to have 16384 (0x4000) as the dividend and .25 (0x2000) as the divisor, the quotient would be 65536, which does not fit in 16.0 format. This operation would overflow, producing an erroneous results.

Input operands can be checked before division to ensure that an overflow will not result. If the magnitude of the upper 16 bits of the dividend is larger than the magnitude of the divisor, an overflow will result.

B.1.4 Integer Division

One special case of division that deserves special mention is integer division. There may be some cases where you wish to divide two integers, and produce an integer result. It can be seen that an integer-integer division will produce an invalid output format of (32-16+1).(0-0-1), or 17.-1.

To generate an integer quotient, you must shift the dividend to the left one bit, placing it in 31.1 format. The output format for this division will be (31-16+1).(1-0-1), or 16.0. You must ensure that no significant bits are lost during the left shift, or an invalid result will be generated.

B.2 ERROR CONDITIONS

Although the divide primitives for the ADSP-2100 family work correctly in most instances, there are two cases where an invalid or inaccurate result can be generated. The first case involves signed division by a negative number. If you attempt to use a negative number as the divisor, the quotient generated may be one LSB less than the correct result. The other case concerns unsigned division by a divisor greater than 0x7FFF. If the divisor in an unsigned division exceeds 0x7FFF, an invalid quotient will be generated.

B.2.1 Negative Divisor Error

The quotient produced by a divide with a negative divisor will generally be one LSB less than the correct result. The divide algorithm implemented on the ADSP-2100 family does not correctly compensate for the twos-complement format of a negative number, causing this inaccuracy.

B Division Exceptions

There is one case where this discrepancy does not occur. If the result of the division operation should equal 0x8000, then it will be correctly represented, and not be one LSB off.

There are several ways to correct for this error. Before changing any code, however, you should determine if a one-LSB error in your quotient is a significant problem. In some cases, the LSB is small enough to be insignificant. If you find it necessary have exact results, two solutions are possible.

One is to avoid division by negative numbers. If your divisor is negative, take its absolute value and invert the sign of the quotient after division. This will produce the correct result.

Another technique would be to check the result by multiplying the quotient by the divisor. Compare this value with the dividend, and if they are off by more than the value of the divisor, increase the quotient by one.

B.2.2 Unsigned Division Error

Unsigned divisions can produce erroneous results if the divisor is greater than 0x7FFF. You should not attempt to divide two unsigned numbers if the divisor has a one in the MSB. If it is necessary to perform a such a division, both operands should be shifted right one bit. This will maintain the correct orientation of operands.

Shifting both operands may result in a one LSB error in the quotient. This can be solved by multiplying the quotient by the original (not shifted) divisor. Subtract this value from the original dividend to calculate the error. If the error is greater than the divisor, add one to the quotient, if it is negative, subtract one from the quotient.

B.3 SOFTWARE SOLUTION

Each of the problems mentioned in this Appendix can be compensated for in software. Listing 1 shows the module *divide_solution*. This code can be used to divide two signed or unsigned numbers to produce the correct quotient, or an error condition.

In addition to correcting the problems mentioned, this module provides a check for division overflow and computes the remainder following the division.

Division Exceptions B

Since many applications do not require complete error checking, the code has been designed so you can remove tests that are not necessary for your project. This will decrease memory requirements, as well as increase execution speed.

The module *signed_div* expects the 32-bit dividend to be stored in AY1&AY0, and the divisor in AX0. Upon return either the AR register holds the quotient and MR0 holds the remainder, or the overflow flag is set. The entire routine takes at most twenty-seven cycles to execute. If an exception condition exists, it may return sooner. The first two instructions store the dividend in the MR registers, the absolute value of the dividend's MSW in AF, and the divisor's absolute value in AR.

The code block labeled *test_1* checks for division by 0x8000. Attempting to take the absolute value of 0x8000 produces an overflow. If the AV flag is set (from taking the absolute value of the divisor), then the quotient is – AY1. This can produce an error if AY1 is 0x8000, so after taking the negative of AY1, the overflow flag is checked again. If it is set control is returned to the calling routine, otherwise the remainder is computed. If it is not necessary to check for a divisor of 0x8000, this code block can be removed.

The code block labeled *test_2* checks for a division overflow condition. The absolute value of the divisor is subtracted from the absolute value of the dividend's MSW. If the divisor is less then the dividend, it is likely an overflow will occur. If the two are equal in magnitude, but different in sign, the result will be 0x8000, so this special case is checked. If your application does not require an overflow check, this code block can be removed. If you decide to remove *test_2* be sure to change the JUMP address in *test_1* to *do_divs*, instead of *test_2*.

After error checking, the actual division is performed. Since the absolute value of the divisor has been stored in AR, this is used as the X-operand for the DIVS instruction. 15 DIVQ instructions follow, computing the rest of the quotient. The correct sign for the quotient is determined, based on the AS flag of the ASTAT register. Since the MR register contains the original dividend, the remainder can be determine by a multiply subtract operation. The divisor times the quotient is subtracted from MR to produce the remainder in MR0.

The last step before returning is to clear the ASTAT register which may contain an overflow flag produced during the divide.

B Division Exceptions

The subroutine *unsigned_div* is very similar to *signed_div*. MR1 and AF are loaded with the MSW of the dividend, MR0 is loaded with the dividend LSW and the divisor is passed into AR. Since unsigned division with a large divisor (> 0x7FFF) is prohibited, the MSB of the divisor is checked. If it contains a one, the overflow flag is set, and the routine returns to the caller. Otherwise *test_11* checks for a standard divide overflow.

In *test_11* the divisor is subtracted from the MSW of the dividend. If the result is less then zero division can proceed, otherwise the overflow flag is set. If you wish to remove *test_11*, be sure to change the JUMP address in *test_10* to *do_divq*.

The actual unsigned division is performed by first clearing the AQ bit of the ASTAT register, then executing sixteen DIVQ instructions. The remainder is computed, after first setting MR2 to zero. This is necessary since MR1 automatically sign-extends into MR2. Also, the multiply must be executed with the unsigned switch. To ensure that the overflow flag is clear, ASTAT is set to zero before returning.

In both subroutines, the computation of the remainder requires only one extra cycle, so it is unlikely you would need to remove it for speed. If it is a problem to have the multiply registers altered, remove the multiply/subtract instruction just before the return, and remove the register transfers to MR0 and MR1 in the first two multifunction instructions. Be sure to remove the MR2=0; instruction in the *unsigned_div* subroutine also.

```
.MODULE/ROM   Divide_solution;

{
This module can be used to generate correct results when using the divide primitives
of the ADSP-2100 family. The code is organized in sections. This entire module can
be used to handle all error conditions, or individual sections can be removed to
increase execution speed.

   Entry Points
   signed_div Computes 16-bit signed quotient
   unsigned_div Computes 16-bit unsigned quotient

   Calling Parameters
   AX0 = 16-bit divisor
   AY0 = Lower 16 bits of dividend
   AY1 = Upper 16 bits of dividend
```

```
   Return Values
   AR = 16-bit quotient
   MR0 = 16-bit remainder
   AV flag set if divide would overflow

   Altered Registers
   AX0, AX1, AR, AF, AY0, AY1, MR, MY0

   Computation Time: 30 cycles
}

.ENTRY          signed_div, unsigned_div;

signed_div:     MR0=AY0,AF=AX0+AY1;         {Take divisor's absolute value}
                MR1=AY1, AR=ABS AX0;        {See if divisor, dividend have
                                             same magnitude}

test_1:         IF NE JUMP test_2;          {If divisor non-zero, do test 2}
                ASTAT=0x4;                  {Divide by zero, so overflow}
                RTS;                        {Return to calling program}

test_2:         IF NOT AV JUMP test_3;      {If divisor 0x8000, then the}
                AY0=AY1, AF=ABS AY1;        {quotient is simply -AY1}
                IF NOT AV JUMP recover_sign;
                ASTAT=0x4;                  {0x8000 divided by 0x8000,}
                RTS;                        {so overflow}

test_3:         AF=PASS AF;                 {Check for division overflow}
                IF NE JUMP test_4;          {Not equal, jump test 4}
                AY0=0x8000;                 {Quotient equals -1}
                ASTAT=0x0;                  {Clear AS bit of ASTAT}
                JUMP recover_sign;          {Compute remainder}

test_4:         AF=ABS MR1;                 {Get absolute of dividend}
                AR=ABS AX0;                 {Restore AS bit of ASTAT}
                AF=AF-AR;                   {Check for division overflow}
                IF LT JUMP do_divs;         {If Divisor>Dividend do divide}
                ASTAT=0x4;                  {Division overflow}
                RTS;
```

Listing B.1 Division Error Routine

(continues on next page)

B Division Exceptions

```
do_divs:        DIVS AY1, AR; DIVQ AR;      {Compute sign of quotient}
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;

recover_sign:   MY0=AX0,AR=PASS AY0;        {Put quotient into AR}
                IF NEG AR=-AY0;             {Restore sign if necessary}
                MR=MR-AR*MY0 (SS);          {compute remainder dividend neg}
                RTS;                        {Return to calling program}

unsigned_div:   MR0=AY0, AF=PASS AY1;       {Move dividend MSW to AF}
                MR1=AY1, AR=PASS AX0;       {Is MSB set?}

test_10:        IF GT JUMP test_11;         {No, so check overflow}
                ASTAT=0x4;                  {Yes, so set overflow flag}
                RTS;                        {Return to caller}

test_11:        AR=AY1-AX0;                 {Is divisor<dividend?}
                IF LT JUMP do_divq;         {No, so go do unsigned divide}
                ASTAT=0x4;                  {Set overflow flag}
                RTS;

do_divq:        ASTAT=0;                    {Clear AQ flag}
                DIVQ AX0; DIVQ AX0;         {Do the divide}
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;

uremainder:     MR2=0;                      {MR0 and MR1 previous set}
                MY0=AX0, AR=PASS AY0;       {Divisor in MYO, Quotient in AR}
                MR=MR-AR*MY0 (UU);          {Determine remainder}
                RTS;                        {Return to calling program}

.ENDMOD;
```

Listing B.1 Division Error Routine