Application Report

SPRA654 - March 2000

Autoscaling Radix-4 FFT for TMS320C6000™

Yao-Ting Cheng Taiwan Semiconductor Sales & Marketing

ABSTRACT

Fixed-point digital signal processors (DSPs) have limited dynamic range to deal with digital data. This application report proposes a scheme to test and scale the result output from each Fast Fourier Transform (FFT) stage in order to fix the accumulation overflow. The radix-4 FFT algorithm is selected since it provides fewer stages than radix-2 algorithm. Thus, the scaling operations are minimized. This application report is organized as follows:

- Basics of FFT
- Multiplication and addition overflow
- Algorithm to test bit growth and scaling the result
- Implementation by C and Linear Assembly on the C6000 DSP
- List of the codes

Contents

List of Figures

1 FFT (Fast Fourier Transform)

Many applications require the processing of signals in the digital world, digital signal processing. Because we may need to process a signal based on its frequency characteristics, there is a need to reformat the signal. The Discrete Fourier Transform (DFT) is one of the ways to convert the signal from time domain to frequency domain. DFT is a discrete version of Fourier Transform and is very computable by the modern microprocessor. The DFT equation is listed below:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \; k = 0 \text{ to } N-1 \text{ where } W_n = e^{-j2\pi/N}$$

Many calculations are needed. There are N^2 complex multiplications and N^2 complex additions for an N-point DFT. One of the algorithms that can reduce dramatically the number of computations is the radix-2 FFT, which takes advantage of the periodicity of the Twiddle Factor W_N^{nk}. For example, if n=N, then

$$W_N^{nk} = W_N^{Nk} = e^{-j\left(\frac{2\pi}{N}\right)NK} = e^{-j2\pi k} = -1.$$

The radix-2 FFT equation is listed below:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} = \sum_{n=0}^{\frac{N}{2}-1} \left[x(n) + (-1)^k x\left(n + \frac{N}{2}\right) \right] W_N^{nk}$$

The radix-2 FFT equation simply divides the DFT into two smaller DFTs. Each of the smaller DFTs is then further divided into smaller ones and so on (see Figure 1). It consists of $\log_2 N$ stages and each stage consists of N/2 butterflies. Each butterfly consists of two additions for the input data and one multiplication to the twiddle factor.

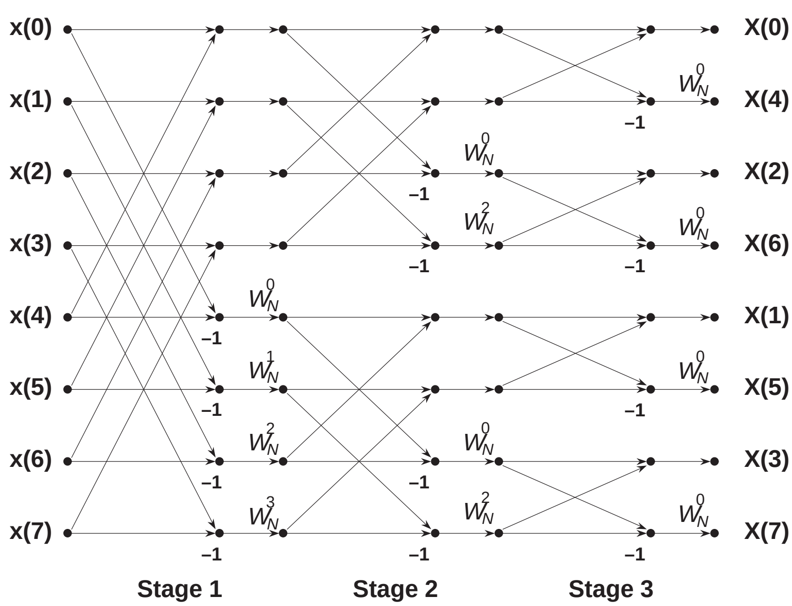

Figure 1. Radix-2 FFT for N=8

The other popular algorithm is the radix-4 FFT, which is even more efficient than the radix-2 FFT. The radix-4 FFT equation is listed below:

$$X(k) = \sum_{n=0}^{\frac{N}{4}-1} \left[x(n) + (-j)^k x\left(n + \frac{N}{4}\right) + (-1)^k x\left(n + \frac{N}{2}\right) + (j)^k x\left(n + \frac{3N}{4}\right) \right] W_N^{nk}$$

The radix-4 FFT equation essentially combines two stages of a radix-2 FFT into one, so that half as many stages are required (see Figure 2). Since the radix-4 FFT requires fewer stages and butterflies than the radix 2 FFT, the computations of FFT can be further improved. For example, to calculate a 16-point FFT, the radix-2 takes $\log_2 16=4$ stages but the radix-4 takes only $\log_4 16=2$ stages. Next, we discuss the numerical issue that arises from a finite length problem. Most people use a fixed-point DSP to perform the calculation in their embedded system because the fixed-point DSP is highly programmable and is cost efficient. The drawback is that the fixed-point DSP has limited dynamic range, which is worsened by the summation overflow problem that occurs all the time in FFT. A scheme is needed to overcome this issue.

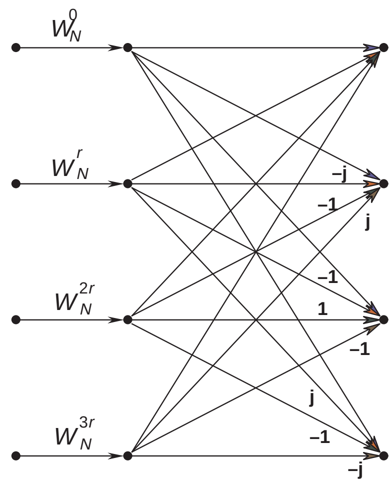

Figure 2. Radix-4 Butterfly

2 Multiplication and Additions Overflow

FFT is nothing but a bundle of multiplications and summations which may overflow during multiplication and addition. This application report adopts the radix-4 algorithm developed by C. S. Burrus and T. W. Parks to explain how to solve these two kinds of overflow on a C6000 DSP. The radix-4 FFT C equivalent program is listed below:

```
void radix4(int n, short x[], short w[])
{
   int   n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3, j, k;
   short t, r1, r2, s1, s2, co1, co2, co3, si1, si2, si3;

   n2 = n;
   ie = 1;
   for (k = n; k > 1; k >>= 2) {           // number of stages
       n1 = n2;
```

```
        n2 >>= 2;
        ia1 = 0;
        for (j = 0; j < n2; j++) {          // number of butterflies
            ia2 = ia1 + ia1;                // per stage
            ia3 = ia2 + ia1;
            co1 = w[ia1 * 2 + 1];
            si1 = w[ia1 * 2];
            co2 = w[ia2 * 2 + 1];
            si2 = w[ia2 * 2];
            co3 = w[ia3 * 2 + 1];
            si3 = w[ia3 * 2];
            ia1 = ia1 + ie;
           for (i0 = j; i0 < n; i0 += n1) {  // loop for butterfly
                i1 = i0 + n2;                // calculations
                i2 = i1 + n2;
                i3 = i2 + n2;
                r1 = x[2 * i0] + x[2 * i2];
                r2 = x[2 * i0] - x[2 * i2];
                t = x[2 * i1] + x[2 * i3];
                x[2 * i0] = r1 + t;
                r1 = r1 - t;
                s1 = x[2 * i0 + 1] + x[2 * i2 + 1];
                s2 = x[2 * i0 + 1] - x[2 * i2 + 1];
                t = x[2 * i1 + 1] + x[2 * i3 + 1];
                x[2 * i0 + 1] = s1 + t;
                s1 = s1 - t;
                x[2 * i2] = (r1 * co2 + s1 * si2)    >> 15;
                x[2 * i2 + 1] = (s1 * co2 - r1 * si2)>> 15;
                t = x[2 * i1 + 1] - x[2 * i3 + 1];
                r1 = r2 + t;
                r2 = r2 - t;
                t = x[2 * i1] - x[2 * i3];
                s1 = s2 - t;
                s2 = s2 + t;
                x[2 * i1] = (r1 * co1 + s1 * si1)    >> 15;
                x[2 * i1 + 1] = (s1 * co1 - r1 * si1)>> 15;
                x[2 * i3] = (r2 * co3 + s2 * si3)    >> 15;
                x[2 * i3 + 1] = (s2 * co3 - r2 * si3)>> 15;
           }
        }
       ie <<= 2;
}
```

To deal with the multiplication overflow, we need to interpret all input samples and twiddle factors, W_N^{nk}, as fractional numbers because a fractional number times a fractional number is always less than or equal to one. For the C6000 DSP, the largest 16-bit positive fractional binary number is <u>0.111 1111 1111 1111</u>, which is mapped as 32767 in integer domain (or 0x7FFF in hexadecimal). The smallest negative number is <u>1.000 0000 0000 0000</u>, which is noted as 32768 in integer (or 0x8000 in hexadecimal). The only exception that multiplication still occurs is –1 times –1; the result of which should be equal to positive 1. However, we have the largest positive number <u>0.111 1111 1111 1111</u>, which is very close to one but not precisely the perfect 1. The

C6000 DSP provides Saturation Multiplication instructions such as SMPY that can fix this problem.

The second overflow comes from additions. According to the algorithm listed above, up to five additions are needed to calculate the output. For example, one of the FFT output data is calculated as

```
x[2 * i1] = r1 * co1 + s1 * si1
         = (r2 + t) * co1 + (s2 - t) * si1
         = (r2 + (x[2*i1+1] – x[2*i3+1])) * co1 +
                    (s2 – (x[2*i1] – x[2*i3])) * s11.
```

It can contribute up to a 3-bit growth within the butterflies. The easiest way to fix it is to scale down the input samples 3 bits at each stage. Somehow, it costs a lot of dynamic range. The other way to fix it is to detect if the bit grows at the output of each stage. Then, scale down the result based on how many bits have grown before feeding the result into the next stage.

3 Bit-Growth Detection and Scaling Algorithm

The C6000 DSP provides the instruction NORM that can help detect how many bits grow after each addition. For example, assume the content of the 32-bit register A1 is 0000 0000 0000 0000 0010 0010 1100 1111. After performing the NORM operation such as

```
NORM.L1    A1, A2
```

The A1 will stay unchanged and A2 will be 17, which is simply the number of non-redundant sign bits as shown with double underscore. If the content of A1 grows 3 bits as 0000 0000 0000 0001 0010 0010 0011 0011, the result of NORM will be 14 because the non-redundant sign bit is decreased by 3 bits. Once we have the number of bit-growth, we can properly scale down the result by right-shifting the content of the register. One more issue to be considered is the input data format. Generally, the Q15 number is adopted for most of the system. It means that there is one sign bit in the most significant bit (MSB) for 16-bit data such as S.XXX XXXX XXXX XXXX, where S is the sign bit. To prevent addition overflow for radix-4 FFT, we need three guard bits; therefore, the data should be Q12, such as SSSS. XXXX XXXX XXXX. It is a reasonable approach since the resolution of most of the analog-to-digital converters is less than or equal to 12 bits. The result returned by the NORM instruction for Q12 data is therefore 19. The algorithm is summarized below:

Step 1: Input data should be in the format of Q12 to gain three guard bits. Set exp = 19, which is the number of non-redundant sign bits of Q12 data.

Step 2: At the end of each butterfly calculation, take the test of bit growth and record the maximum as follows:

```
if ((exp_temp = _norm(X[k])) < 19)
    if (exp_temp < exp) exp = exp_temp;
```

Step 3: At the end of each stage, test to see if FFT is not in the last stage. There is no need to scale the last output. Then, test if the bit grows. If it does, scale down the output back to Q12.

```
if (!last_stage) {
    if (exp < 19) {
        for (i=0; i<(2*N); i++) X[I]>>=(18-exp);
```

```
                scale += (19-exp);
                exp = 18;
            }
        }
```

Example programs are listed below. Example 1 is the main program that provides the input samples and the twiddle factors for 16-point FFT. Example 2 is the autoscaling radix-4 FFT implemented in C with C6000 intrinsics. Example 3 is the FFT subroutine implemented with C6000 linear assembly.

4 Example 1 – Main Program

```
#define Q12_SCALE 8
extern int r4_fft(short, short*, short*);
short x[32]={   0,                0,       // input samples
                4617/Q12_SCALE,   0,       // Scale the data from Q15 to Q12
                9118/Q12_SCALE,   0,
                13389/Q12_SCALE,  0,
                17324/Q12_SCALE,  0,
                20825/Q12_SCALE,  0,
                23804/Q12_SCALE,  0,
                26187/Q12_SCALE,  0,
                27914/Q12_SCALE,  0,
                28941/Q12_SCALE,  0,
                29242/Q12_SCALE,  0,
                28811/Q12_SCALE,  0,
                27658/Q12_SCALE,  0,
                25811/Q12_SCALE,  0,
                23318/Q12_SCALE,  0,
                20241/Q12_SCALE,  0   };
short w[32]={   0,      32767,  // Twiddle Factors
                12540,  30274,  // 32768*sin(2PI*n/N), 32768*cos(2PI*n/N)
                23170,  23170,
                30274,  12540,
                32767,  0,
                30274,  -12540,
                23170,  -23170,
                12540,  -30274,
                0,      -32767,
                -12540, -30274,
                -23170, -23170,
                -30274, -12540,
                -32767, 0,
                -30274, 12540,
                -23170, 23170,
                -12540, 30274   };
short index[16]={   0,  4,  8,  12,   // index for 16-points digit reverse
                    1,  5,  9,  13,
                    2,  6,  10, 14,
                    3,  7,  11, 15  };
short y[32];   // outputs
main()
```

```
{
    int n=16;
    int i;
    int scale;
    scale = r4_fft(n,x,w);
    for(i=0; i<n; i++) {
        y[2*i] = x[index[i]*2];
        y[2*i+1] = x[index[i]*2+1];
    }
}
```

5 Example 2 – Autoscaling Radix-4 FFT With C6000 C Intrinsics

```
int r4_fft(short n, int x[], const int w[])
{
    int n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3, j, k;
    int t0, t1, t2;
    int xtmph, xtmpl;
    int shift, exp=19, scale=0;
    n2 = n;
    ie = 1;
    for ( k=n; k>1; k>>=2 ) {
        n1 = n2;
        n2 >>= 2;
        ia1 = 0;
        for ( j=0; j<n2; j++ ) {
            ia2 = ia1 + ia1;
            ia3 = ia2 + ia1;
            for ( i0=j; i0<n; i0+=n1) {
                i1 = i0 + n2;
                i2 = i1 + n2;
                i3 = i2 + n2;
                t0 = _add2(x[i1],x[i3]);
                t1 = _add2(x[i0],x[i2]);
                t2 = _sub2(x[i0],x[i2]);
                x[i0] = _add2(t0,t1);
                t1 = _sub2(t1,t0);
                xtmph = (_smpyh(t1,w[ia2]) - _smpy(t1,w[ia2])) & 0xffff0000;
                xtmpl = ((_smpylh(t1,w[ia2]) + _smpyhl(t1,w[ia2])) >> 16) &
                        0x0000ffff;
                x[i2] = xtmph | xtmpl;
                t0 = _sub2(x[i1],x[i3]);
                t1 = -(t0 << 16);
                t0 = t1 | ((t0 >> 16) & 0x0000ffff);
                t1 = _add2(t2,t0);
                t2 = _sub2(t2,t0);
                xtmph = (_smpyh(t1,w[ia1]) - _smpy(t1,w[ia1])) & 0xffff0000;
                xtmpl = ((_smpylh(t1,w[ia1]) + _smpyhl(t1,w[ia1])) >> 16) &
                        0x0000ffff;
                x[i1] = xtmph | xtmpl;
                xtmph = (_smpyh(t2,w[ia3]) - _smpy(t2,w[ia3])) & 0xffff0000;
                xtmpl = ((_smpylh(t2,w[ia3]) + _smpyhl(t2,w[ia3])) >> 16) &
                        0x0000ffff;
```

```
                x[i3] = xtmph | xtmpl;
            }
            ia1 = ia1 + ie;
        }
        if ( k > 4 ) {
            ie <<= 2;
            j=0;
            while ( (exp > 16) && (j < n) ) {
                xtmph = _norm(x[j] >> 16);
                xtmpl = _norm(x[j] << 16 >> 16);
                if ( xtmph < exp ) exp=xtmph;
                if ( xtmpl < exp ) exp=xtmpl;
                j++;
            }
           if ( exp < 19 ) {
                shift = 19-exp;
                exp = 19;
                scale += shift;
                _nassert(j>15);
                for ( j=0; j<n; j++ ) {
                    xtmph = (x[j] >> shift) & 0xffff0000;
                    xtmpl = ((x[j] << 16) >> (16+shift)) & 0x0000ffff;
                    x[j] = xtmph | xtmpl;
                }
            }
        }
    }
    return scale;
}
```

6 Example 3 – Autoscaling Radix-4 FFT With C6000 Linear Assembly

```
        .title  ”r4_fft.sa”
        .def    _r4_fft
        .text
_r4_fft .cproc  n, p_x, p_w
        .reg    n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3, j, k;
        .reg    t0, t1, t2, w, x0, x1, x2, x3;
        .reg    tmp, mskh, xtmph, xtmpl;
        .reg    exp, scale;
        add     n, 0, n2
        mvk     1, ie
        zero    mskh
        mvkh    0xffff0000, mskh
        zero    scale
        add     n, 0, k
stage_loop:
        add     n2, 0, n1
        shr     n2, 2, n2
        zero    ia1
        zero    j
group_loop:
```

```
        add     ia1, ia1, ia2
        add     ia2, ia1, ia3
        add     j, 0, i0
butterfly_loop:
        add     i0, n2, i1
        add     i1, n2, i2
        add     i2, n2, i3
        ldw     *+p_x[i0], x0
        ldw     *+p_x[i1], x1
        ldw     *+p_x[i2], x2
        ldw     *+p_x[i3], x3
        add2    x1, x3, t0
        add2    x0, x2, t1
        sub2    x0, x2, t2
        add2    t0, t1, x0              ; x0
        sub2    t1, t0, t1
        ldw     *+p_w[ia2], w           ; load twiddle factor w2
        smpyh   t1, w, tmp
        smpy    t1, w, xtmph
        sub     tmp, xtmph, xtmph
        and     xtmph, mskh, xtmph
        smpylh  t1, w, tmp
        smpyhl  t1, w, xtmpl
        add     tmp, xtmpl, xtmpl
        shru    xtmpl, 16, xtmpl
        or      xtmph, xtmpl, x2        ; x2
        sub2    x1, x3, t0
        shl     t0, 16, t1
        neg     t1, t1
        extu    t0, 0 ,16, t0
        or      t1, t0, t0
        add2    t2, t0, t1
        sub2    t2, t0, t2
        ldw     *+p_w[ia1], w           ; load twiddle factor w1
        smpyh   t1, w, tmp
        smpy    t1, w, xtmph
        sub     tmp, xtmph, xtmph
        and     xtmph, mskh, xtmph
        smpylh  t1, w, tmp
        smpyhl  t1, w, xtmpl
        add     tmp, xtmpl, xtmpl
        shru    xtmpl, 16, xtmpl
        or      xtmph, xtmpl, x1        ; x1
        ldw     *+p_w[ia3], w           ; load twiddle factor w2
        smpyh   t2, w, tmp
        smpy    t2, w, xtmph
        sub     tmp, xtmph, xtmph
        and     xtmph, mskh, xtmph
        smpylh  t2, w, tmp
        smpyhl  t2, w, xtmpl
        add     tmp, xtmpl, xtmpl
```

```
        shru    xtmpl, 16, xtmpl
        or      xtmph, xtmpl, x3        ; x3
        stw     x0, *+p_x[i0]
        stw     x1, *+p_x[i1]
        stw     x2, *+p_x[i2]
        stw     x3, *+p_x[i3]
        add     i0, n1, i0
        cmplt   i0, n, tmp
   [tmp]b       butterfly_loop  ; branch to butterfly loop
        add     ia1, ie, ia1
        add     j, 1, j
        cmplt   j, n2, tmp
   [tmp]b       group_loop      ; branch to group loop
        cmpeq   k, 4, tmp       ; test if last stage
   [tmp]b       end             ; if true, branch to end
        mvk     2, exp          ; initialize exponent
        zero    j               ; initialize index
        mvkl    0x0000ffff, t2  ; mask for masking xtmpl
        mvkh    0x0000ffff, t2
test_bit_growth:   .trip 16
        ldw     *+p_x[j], tmp
        norm    tmp, xtmph      ; test for redundant sign bit of HI half
        shl     tmp, 16, xtmpl
        norm    xtmpl, xtmpl    ; test for redundant sign bit of LO half
        cmplt   xtmph, exp, tmp         ; test if bit grow
   [tmp]add     xtmph, 0, exp
        cmplt   xtmpl, exp, tmp         ; test if bit grow
   [tmp]add     xtmpl, 0, exp
                cmpgt   exp, 2, tmp     ; if exp>2 than no scaling
   [tmp]b       no_scale

        cmpeq   exp, 0, tmp             ; compare if bit grow 3 bits
   [tmp]sub     3, exp, t0              ; calculate shift
   [tmp]mvk     0x0213, t1              ; csta & cstb to ext xtmpl
   [tmp]add     scale, t0, scale        ; accumulate scale
   [tmp]b       scaling
        cmpeq   exp, 1, tmp             ; compare if bit grow 2 bit
   [tmp]sub     3, exp, t0
   [tmp]mvk     0x0212, t1              ; csta & cstb to ext xtmpl
   [tmp]add     scale, t0, scale        ; accumulate scale
   [tmp]b       scaling

        sub     3, exp, t0              ; grows 1 bit
        mvk     0x0211, t1              ; csta & cstb to ext xtmpl
        add     scale, t0, scale        ; accumulate scale
        b       scaling
no_scale:
        add     j, 1, j
        cmplt   j, n, tmp               ; compare if test all output
   [tmp]b       test_bit_growth         ; if not, test next output
        b       next_stage              ; else go to next stage
```

```
scaling:
        zero    j
scaling_loop:   .trip   16
        ldw     *+p_x[j], tmp
        shr     tmp, t0, xtmph          ; scaling HI half
        and     xtmph, mskh, xtmph      ; mask HI half
        ext     tmp, t1, xtmpl          ; scaling LO half
        and     xtmpl, t2, xtmpl        ; mask LO half by 0x0000ffff
        or      xtmph, xtmpl, tmp       ; x[j]=[xtmph | xtmpl]
        stw     tmp, *+p_x[j]
        add     j, 1, j
        cmplt   j, n, tmp
   [tmp]b       scaling_loop
next_stage:
        shl     ie, 2, ie
        shr     k, 2, k
        b       stage_loop      ; end of stage loop
end:
        .return scale
        .endproc
```

7 References

1. C.S. Burrus and T.W. Parks, *DFT/FFT and Convolution Algorithms and Implementation*, John Wiley & Sons, New York, 1985.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.