TMS320 DSP
Number 22
DESIGNER'S NOTEBOOK
TEXAS INSTRUMENTS

Fast Logarithms on a Floating-Point Device

Contributed by Keith Larson

Design Problem What is the fastest way to calculate logarithms (base 2) on a TMS320C30 or TMS320C40?

Solution The following TMS320C30/'C40 function calculates the log base two of a number in about half the time of conventional algorithms. Furthermore, the method can easily be scaled for faster execution if less accuracy is desired. The method is efficient because the algorithm uses the floating-point multipliers' exponent/normalization hardware in a unique way. The following is a proof of the algorithm.

The value of a floating point number X is given by

```
X = 2^EXP_old * mant_old
```

If you then consider that the bit fields used to store the exponent and mantissa are actually integer, you will notice that the exponent is already in log2 (log base 2) form. In fact, the exponent is nothing more than a normalizing shift value.

By converting both sides of the first equation to a logarithm, we find that the logarithm of the value becomes the sum of the exponent and mantissa in log form.

```
log2(X) = EXP_old + log2(mant_old)    (Log base two)
```

Since EXP is in the exponent register, no calculation is needed and the value can be used directly as an integer. To extract the value of the exponent, PUSH, POP, and masking operations are used.

The remaining mantissa conversion is done by first forcing the exponent bits to zero using an LDE 1.0 instruction. This causes the exponent term 2^EXP to equal 1.0, leaving $1.0 \leq \text{Value} < 2.0$. Then, by using the following identity, the logarithm of the mantissa can be extracted from the final results exponent.

If the value (mant_old) is repeatedly squared, the sequence becomes

```
X_new = mant_old^N          Where: 1.0 ≤ X_new < 2^N
                                   N = 1,2,4,8,16...
```

Since the hardware multiplier will restructure the new value (X_new) during each squaring operation, we see that X_new will be represented by a new exponent (EXP_new) and mantissa (mant_new).

```
X_new = 2^EXP_New * mant_new
```

By then applying familiar logarithm rules, we find that EXP_new holds the logarithm of Old_mant. This is best shown by setting the previous two equations equal to each other and taking the logarithm of both sides.

```
mant_old^N = 2^EXP_new * mant_new          N=1,2,4,8,16...

N * log2(mant_old) = EXP_new + log2(mant_new)

log2(mant_old) = EXP_new/N + log2(mant_new)/N
```

This last equation shows that the logarithm of mant_old is indeed related to EXP_new. And as shown earlier, EXP_new can be separated from the new mantissa and used as the logarithm of the original mantissa.

We also need to consider the divisor N, which is defined to be the series 1, 2, 4, 8, 16... , and EXP_new is an integer. The division by N becomes a shift for each squaring operation. What remains is to concatenate the bits of EXP_new to EXP_old and then repeat the process until the desired accuracy is achieved.

Example

Consider a mantissa value of 1.5 and an exponent value of 0 (giving an exponent multiplier 2^0, or 1.0). The The TMS320C30/'C40 extended register bit pattern for the algorithm sequence is shown below.

Squaring Operation of F0 = 1.5

Exp	S	Mantissa			
00000000	0	10000000000000000000000000000000	X	=1.5	Exp=0
00000001	0	00100000000000000000000000000000	X^2	=2.25	Exp=1
00000010	0	01000100000000000000000000000000	X^4	=5.0625	Exp=2
00000100	0	10011010000100000000000000000000	X^8	=25.628906	Exp=4
00001001	0	01001000011010111010000010000000	X^16	=656.84083	Exp=9
00010010	0	10100101010100111111101110011111	X^32	=431.43988-E3	Exp=18
00100101	0	01011010101101101010000101010001	X^64	=186.14037-E9	Exp=37
01001010	0	11010101100100100010101011000011	X^128	=34.648238-E21	Exp=74
XXXXXXXX	S	MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM			
← Exp →	S	← Mantissa →			

Hand-calculated value of log2(1.5)

```
log2(1.5) = 0.58496250 = 1001010 111000000
                    ⟶ xxxxxxx⟵  first 7 bits (exponent)
                          ⟶ mmm ⟵ quick 3 bits (mantissa)
```

If you compare the hand-calculated value and the binary representation of log2(1.5) you will find that the sequence of bits in the exponent (seven bits worth) are equivalent to the seven MSBs of the logarithm. If the exponent could hold all the bits needed for full accuracy, then it would be possible to continue the operation for all 24 bits of the mantissa. Since there are only eight bits in the exponent and the MSBs is used for negative values, only seven iterations are possible before the exponent must be off-loaded and reinitialized to zero.

By concatenating EXP_new to the previous exponent, longer strings of bits can be built for greater accuracy. The process is then repeated until the desired accuracy is achieved. Also remember that the original numbers exponent, which represents the whole number part of the result, becomes the eight MSBs of the final result.

Another trick is to look at the three MSBs of the mantissa, and apply a roundup from the fourth bit, those same MSBs can be used as a quick extension of the exponent (logarithm). To visualize this, consider the following tabulated values and graph.

Mant	log2(Mant)
1.000	0.000
1.250	0.322
1.500	0.585
1.750	0.807
1.999	0.999

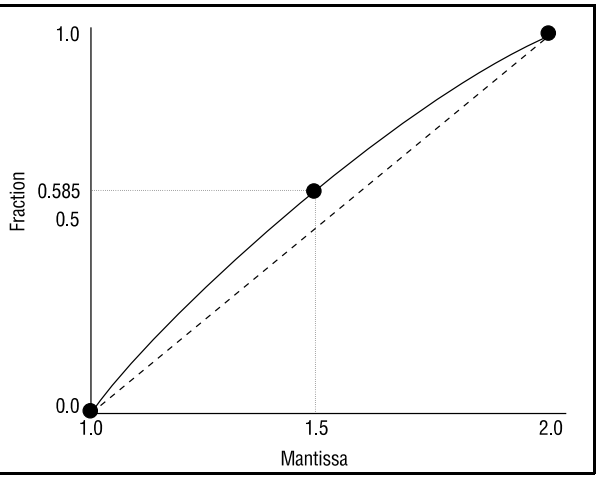

Figure 1.

Note: Notice how the fractional part is the same at the endpoints.

In the middle, only a slight bowing exists which can either be ignored or optionally rounded for better accuracy. The maximum actually occurs at a mantissa value of $\frac{1}{\ln(2.0)}$ or 1.442695. The value of log2(mant) at that point is 0.52876637, giving a maximum error of 0.086071.

When finished, the bits representing the finished logarithm are in a fixed-point notation and will need to be scaled. This is done by using the FLOAT instruction followed by a multiplication by a constant scaling factor. If the final result needs to be in any other base, the scaling factor is simply adjusted for that base.

Here are a few more helpful points.

The round-off accuracy of the first three squaring operations will affect the final result if >21 mantissa bits are desired. A RND instruction placed after the first three MPYF R0,R0 instructions will remedy this, but adds to the cycle count.

When the input value approaches 1.0, the result will be driven close to zero and accuracy will suffer. In this case, an input range comparison and a branch to a McLauren series expansion is used as a solution with minimal degradation in speed. This is because the power series converges quickly for input values close to 1.0.

If you only need to calculate a visual quality logarithm, such as in spectrum analysis, the logarithm can often be calculated in one cycle. In this case the mantissa is substituted directly into the fractional bits of the logarithm giving a maximum error of 0.086 (about 3.5 bits). The one cycle arises from the need to remove the 2's compliment sign bit in the TMS320C30/'C40's mantissa. As far as your eye is concerned, it will never notice the difference!

```
*********************************************************
*              FAST logarithm for FFT displays          *
*  >>>> NEED ONLY ADD ONE INSTRUCTION IN MANY CASES  <<<<  *
*********************************************************
          ||     ||          ;
          MPYF   REAL,REAL,R0  ; calculate the magnitude
          MPYF   IMAG,IMAG,R1  ; Note: sign bit is zero
          ADDF   R1,R0         ;
          ASH    -1,R0         ;<- One instruction logarithm!
          STF    R0,OUT        ;   scaled externally in DAC
          ||     ||          ;
*********************************************************
* _log_E.asm                  DEVICE: TMS320C30         *
*********************************************************
        .global _log_E
_log_E: POP     AR1          ; return address -> AR1
        POPF    R0           ; X -> R0
        LDF     R0,R1        ; use R1 to accumulate answer
        LDI     2,RC         ; repeat 3x
        RPTB    loop         ;                 8 + 13*3 + 9
        ASH     7,R1         ;
        LDE     1.0,R0       ; EXP = 0
        MPYF    R0,R0        ; mant^2
        MPYF    R0,R0        ; mant^4
        MPYF    R0,R0        ; mant^8
        MPYF    R0,R0        ; mant^16
        MPYF    R0,R0        ; mant^32
        MPYF    R0,R0        ; mant^64
        MPYF    R0,R0        ; mant^128
        PUSHF   R0           ; offload 7 bits of exponent
        POP     R3           ;
        ASH     -24,R3       ; remove mantissa
loop:   OR      R3,R1        ; R2 accumulates EXP <log2(man)>
        ASH     11,R1        ; Jam mant_R1 to top (concat. EXP_old)
        ASH     -20,R0       ; align and append the MSBs of mant_R0
        OR      R0,R1        ; (accurate to 3 bits)
        PUSHF   R1           ; PUSH EXP and Mantissa (sign is now data!)
        POP     R0           ; POP as integer (EXP+FRACTION)
        BD      AR1          ;
        FLOAT   R0           ; convert EXP+FRACTION to float
        MPYF    @CONST,R0    ; scale the result by 2^-24 and change base
        ADDI    1,SP         ; restore stack pointer
        .data
CONST_ADR:      .word CONST
CONST           .long    0e7317219h        ;;Base e hand calc w/1 lsb round
                .end
```

Figure 2.