FPGA IMPLEMENTATION OF BASE-N LOGARITHM

Salvador E. Tropea

Electrónica e Informática
Instituto Nacional de Tecnología Industrial
Buenos Aires, Argentina
email: salvador@inti.gov.ar

ABSTRACT

In this work, we present an area optimized FPGA implementation of an IP core to compute the base-N logarithm. Nevertheless, we also discuss the area, speed and precision trade-offs. We selected an algorithm that could be implemented on any FPGA avoiding vendor specific features like block RAMs, embedded multipliers, *etc*. We report the implementation results of a fixed point version of the algorithm using various common configurations on Xilinx and Actel devices. This implementation achieved the required area goals providing a very good speed-area ratio.

Keywords: logarithm, FPGA, area optimized, multiplicative normalization

1. INTRODUCTION

The computation of the logarithm is widely used for many applications like digital filters, power computations (dB) and logarithmic number systems (LNS), to name a few. This is also a key component of LNS, which offer good advantages over traditional floating point systems [1].

This work is organized as follows: section two deals with about the algorithm selection and how it works, section three describes its implementation, section four briefly discusses the error introduced by the approximation, section five shows the results of the implementation and section six provides the conclusions.

2. ALGORITHM SELECTION AND DESCRIPTION

The available methods to compute the logarithm of a number using digital circuits can be divided in two main groups. On the one hand, we have the look-up table based algorithms and, on the other, iterative methods. The first approach is faster and straightforward, but only useful for low precision. This is due to the size of the look-up table. The second group is slower, but suitable for high precision. Many studies explore hybrid implementations that take advantages from both groups [2] and [3], [4], [5] cited by Kostopoulos.

Our project required an algorithm that could be implemented on FPGAs from any vendor, including the Actel's RTSX72SU FPGA. This is a radiation tolerant device and lacks the BRAMs needed to implement big look-up tables. Therefore, we only evaluated iterative algorithms that need small look-up tables. Moreover, we coded the algorithm using standard VHDL to ensure its portability.

Taylor's series expansion is among the most popular methods to manually compute logarithms, but it has a slow convergence and requires slow operations like the division [6]. Other algorithms like the one proposed by Kostopoulos [7] use a multiplier inside the iteration; therefore, they are slow when no embedded multipliers are available. A very well studied alternative is the multiplicative normalization. It just needs adders and shifters and provides a fast convergence [6], [8], [9], consequently it was suitable for our project.

2.1. Multiplicative Normalization

Based on the following logarithms property

$$\log(a \cdot b) = \log(a) + \log(b) \tag{1}$$

Assuming that x is our variable, the multiplicative normalization uses the following values for a and b

$$\log\left(x \cdot \prod_{i=m}^{n} b_i\right) = \log(x) + \sum_{i=m}^{n} \log(b_i) \tag{2}$$

Selecting the b_i values to get

$$x \cdot \prod_{i=m}^{n} b_i = 1 \tag{3}$$

we obtain

$$\log(x) = -\sum_{i=m}^{n} \log(b_i) \tag{4}$$

Storing the $\log(b_i)$ values in a small look-up table the logarithm computation is reduced to sums.

$$y_{i+1} = y_i - \log(b_i) \tag{5}$$

The only requirement is (3). To reduce the multiplications used in (3) to simple shifts we could use

$$b_i = 1 + d_i \cdot 2^{-i} \tag{6}$$

In this way the product computed in each iteration of (3) is

$$x_{i+1} = x_i + x_i \cdot d_i \cdot 2^{-i} \tag{7}$$

and the logarithm iteration

$$y_{i+1} = y_i - \log(1 + d_i \cdot 2^{-i}) \tag{8}$$

Now we must select the d_i values to achieve (3). The function to determine d_i is called decision function.

2.2. A Simple Decision Function

The selection of the decision function affects the convergence speed and the algorithm precision. Although high-radix systems consume big amounts of area, they provide high speed. However, selecting a simple decision function we can achieve a small area footprint.

We selected 1 and 0 as the only possible values for d_i and applied the following criterion: if using 1 makes x_{i+1} bigger than or equal to 1, then, we simply use 0. As the b_i values are smaller and smaller this method asymptotically approaches to 1.

The hardware implementation was very simple because to determine whether the resulting value is bigger than or equal to 1 we just need to check one bit in the result. When this bit becomes 1 the iteration is simply skipped because (7) and (8) are:

$$x_{i+1} = x_i + 0 \wedge y_{i+1} = y_i - \log(1) \tag{9}$$

This function uses only two values and is simpler than the one described by Koren [6]. Unlike the functions explained in [6] and [9], this function provides a good convergence of the approximation for the whole range of values.

2.3. Convergence Radius

Replacing (6) in (3) we obtain

$$x = \frac{1}{\prod_{i=m}^{n} (1 + d_i \cdot 2^{-i})} \tag{10}$$

The smaller value we can obtain is when the decision function is always 1 and the bigger when the function always selects 0.

$$\frac{1}{\prod_{i=m}^{n} (1 + 2^{-i})} < x < 1 \tag{11}$$

2.4. Range Extension

As (11) shows the valid range for the variable is small. This is not a problem for floating point arithmetic where the variable is normalized. When the values are not normalized, as in the fixed point case, we must reduce them and, then, compensate the reduction at the output of the approximation. A simple method to achieve this is to find a number that multiplied by the variable results in a number that satisfy (11). If we call z to this value and use (1) we can write

$$\log(x) = \log(z \cdot x) - \log(z) \tag{12}$$

The product computation can be simplified by choosing

$$z = 2^{-j} \tag{13}$$

where j is selected, as mentioned above, and the only thing that we must solve is

$$\log(z) = \log(2^{-j}) \tag{14}$$

These values can be stored in a look-up table. When the size of the look-up table becomes an issue, we can solve this as follows: (14) is very simple to compute when the base of the logarithm is 2

$$\log_2(z) = \log_2(2^{-j}) = -j \tag{15}$$

replacing (15) in (12)

$$\log_2(x) = \log_2(2^{-j} \cdot x) + j \tag{16}$$

When we need to compute another base, we could use the following property

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)} \tag{17}$$

applied to (16)

$$\log_a(x) = \frac{1}{\log_2(a)} \cdot \log_2(2^{-j} \cdot x) + j \tag{18}$$

This method replaces the above mentioned look-up table by a multiplication using a constant.

3. ALGORITHM IMPLEMENTATION

Our project used unsigned fixed point input values, whereby, we split (18) in

$$N = 2^{-j} \cdot x \tag{19}$$

$$A = \log_2(N) + j \tag{20}$$

$$M = \frac{1}{\log_2(a)} \cdot A \tag{21}$$

where we call (19) normalization, (20) approximation and (21) base selection.

Note that when using floating point values the mantissa is already normalized and we can start computing (20) using the exponent as *j* and the next step (21) is the same. The main difference is that the result of (21) must be normalized like in (19) achieving a comparable complexity. Moreover, additional circuitry could be needed to detect values out of the logarithm domain.

3.1. Normalization

This step must find the value of j that satisfies (11). When $m = 1$ and $n \geq 3$ we can satisfy (11) using

$$0.5 \leq x < 1 \tag{22}$$

or in binary notation

$$0.1 \leq x < 1.0 \tag{23}$$

For an unsigned fixed point value with *E* integer bits and *D* decimal bits we could find the value of j as follows: we start with *j=E,* then we shift the value to the left until the most significant bit is 1, decreasing *j* in each iteration. The result is a normalized fixed point value with zero integer bits and *E+D* decimal bits. Example: *E=5, D=3, x=00100.101*

x=00100.101 j=5

x=01001.010 j=4

x=10010.100 j=3

The result is *j=3* and the normalized value is *0.100101*. This algorithm consumes up to $E+D-1$ steps to achieve the normalization. Nonetheless, when the input values are chosen at random, half of the values are completed in zero steps and only one value in $2^{E+D}-1$ needs $E+D-1$ steps.

This process could be implemented using a combinational circuit, but it could consume a much bigger area.

3.2. Approximation

High-radix versions of the multiplicative normalization could be used for speed optimization, but for area optimization the proposed decision function is more suitable.

The *m* value in (4) does not need to be 0, thanks to the normalization step, and we can start from 1. The *n* value is determined by the number of bits used in the approximation. As the $\log(b_i)$ values quickly decrease, increasing their truncation error, we observed that for *N* bits the value of *n* should be $n \leq N-2$, where *N* is selected to achieve the desired precision.

We implemented the decision function (7) using *N+1* bits, thus the MSB of x_{i+1} is used to determine whether the step is skipped. A barrel shifter was used to implement the product, while a serial shifter did not show an important area saving and slowed down the computation. Fig. 1 (a) shows the flow chart, where *Table* contains the $\log(b_i)$ values and the *y* variable is the approximation result. Fig. 2 shows a simplified block diagram.

We also incorporated the *j* addition to this block and an error compensation coefficient explained in section four (25).

3.3. Base Selection

This step is a multiplication by a constant and provides a base change. The more common bases are 2, *e* (natural logarithms) and 10. In the first case the constant is 1 and this step is skipped, the approximated constants for the other two cases are 0.6931 and 0.3010.

This block can take advantage of embedded multipliers when available. When optimizing for speed, high speed multipliers (i.e. adder tree) can be used. In our case we used a traditional shift-and-add algorithm.

We optimized the algorithm loading the multiplier with the result obtained after the first step and all the subsequent 0s. This can be illustrated with the following example: let us suppose that we are computing the base 10 logarithm and that we determined that the constant must be truncated at the 14th bit to achieve a desired precision

$$1/\log_2(10) \approx 0.01001101000100$$

The 0s at the right and at the left can be skipped and we get 10011010001. We modified the load of the multiplier to load the result obtained after the use of the four LSBs. As a result, the multiplier only needs to compute the remaining 1001101. In this example, a 14-bit multiplication is reduced to a 7-bit multiplication.

Fig. 1 (b) shows the flow chart, where *OPT* is the number of bits optimized during the load and *Kap(n)* represents the nth bit of the constant.

3.4. Interblock Communication

The above mentioned blocks needs different amounts of clock cycles to finish their tasks and, which is even worst, the first step needs a variable number of clock pulses. For this reason, we implemented a simple handshake mechanism to communicate the blocks. The protocol uses a request line to indicate that a new value is available for the

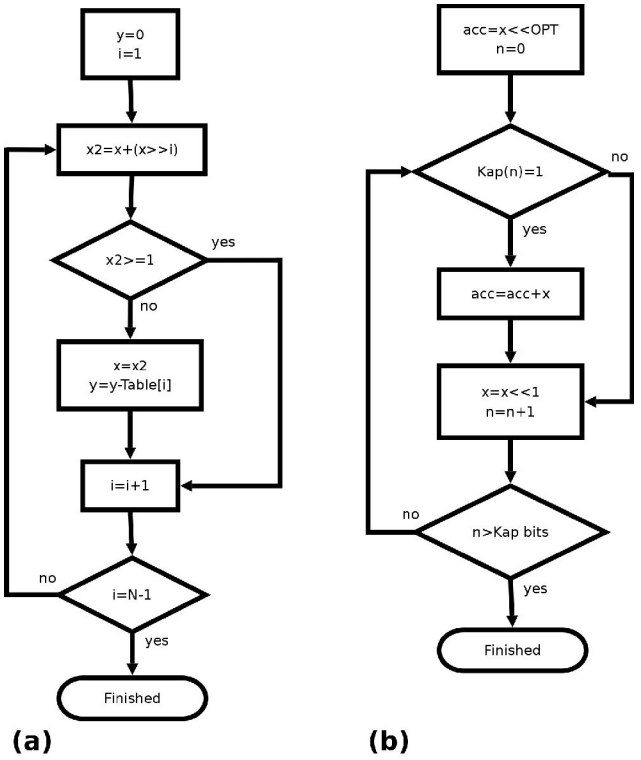

Fig. 1. Flow charts: approximation (a) and multiplier (b)

consumer and an acknowledge line to indicate the value was consumed.

The reported results include the area and clock pulses needed for the above mentioned handshake. The resulting core is composed by three independent blocks allowing the computation of up to three values on the fly.

4. ERROR ANALYSIS

In this section we briefly discuss the error introduced by this core. Unless otherwise specified all the errors are expressed in counts of the LSB, known as ulp (units in the last place). We only provide a simplified analysis to estimate the maximum error, the actual error should be less than or equal to the estimated one.

4.1. Normalization Error

When the total number of input bits is bigger than the number of bits used in the approximation, this block must truncate the value. In this case, the block introduces an error in the (-1;0] range. Adding a rounder, the introduced error is in the (-0.5;0.5] range. However, it does not reduce the total error, as we will see later.

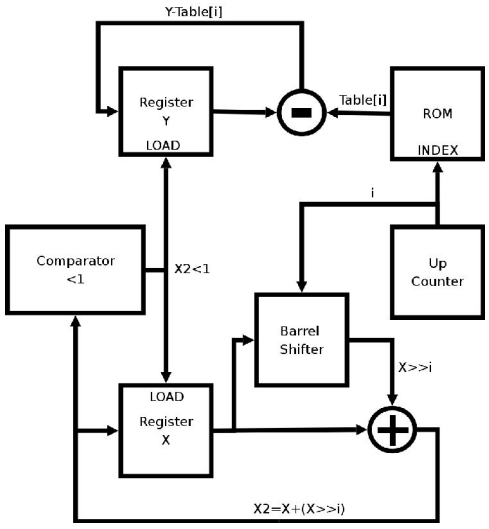

Fig. 2. Simplified block diagram for the approximation

4.2. Approximation Error

This error can be measured by simulation; we used a C implementation for that purpose. Table 1 shows the errors for various sizes.

The normalization error is processed by the approximation converting the (-1;0] range to an approximated (-3;0] range, and it should be added to the error introduced in this step, shown between parentheses in Table 1.

4.3. Multiplication Error

This block introduces two different errors. One is introduced when we truncate the constant. This error is multiplied by the maximum input value, where this is less than E, when the integer part of x has E bits. If we call K to the constant and Kap to the approximated value

$$Etv_{max} = E \cdot (Kap - K) \tag{24}$$

Note that this error is always negative unless we round the constant instead of truncate it. Additionally (24) is not valid when E is 0, in this case we should use -0.5 instead of E.
The other error is the result of the truncation of the output value and is in the (-1;0] range.

In addition, the errors from the previous blocks are affected by the multiplication; consequently, they are reduced.

4.4. Example

We will show an example of the error estimation to clarify the above mentioned components. Let us suppose the following setup: base 10, input values 34.7 (unsigned),

Table 1. Approximation Error

Bits	*Negative Error (ulp)*	*Positive Error (ulp)*
8	3 (6)	5
10	4 (7)	5
12	7 (10)	5
14	8 (11)	6
16	11 (14)	6
26	22 (25)	6

output decimals 10 and the constant is truncated at the 12th bit.

From Table 1 we know that the error at the output of the approximation step is in the [-7;5] range. The negative boundary of the error can be obtained multiplying this error by *Kap,* and then adding *Etv* together with the truncation error. The value of *Kap* is

$$0.010011010001 = 1233/4096$$

and using (24)

$$Etv_{max} = 34 \cdot \left(\frac{1233}{4096} - \frac{1}{\log_2(10)} \right) \approx -0.16\, ulp$$

The negative boundary of the error is

$$-7 \cdot 0.3010 - 0.16 - 1 = -3.27 \approx -3$$

The positive boundary is not affected by the truncation

$$5 \cdot 0.3010 - 0.16 = 1.35 \approx 1$$

resulting in an expected range of [-3;1].

4.5. Error Balancing

As the above analysis shows, the core has an unbalanced error range. This is because the core truncates various values, and we could balance this error adding a constant to (20)

$$A = \log_2(N) + j + Ecomp \qquad (25)$$

In the previous example we could add 3 ulp changing the error at the output of the approximation from [-7;5] to [-4;8] obtaining a final error in the [-2;2] range.

This simple compensation reduced the maximum absolute error from 3 to 2 for this example.

Table 2. Implementation Results

In	*Out*	*Est. Error*	*Meas. Error*	*Clocks*
8.16	4.16	-4;3	-4;3	33/56/18
16.16	4.16	-4;3	-4;3	33/64/18
0.15	4.15	-3;3	-3;3	28/42/17
34.7	5.10	-2;2	-2;2	23/63/12
0.24	0.26	-5;5	-3;4	50/50/28

5. IMPLEMENTATION RESULTS

In this work, we selected some representative unsigned fixed point values (8.16, 16.16, 0.15); the particular case of our project (34.7 using 10 bits for the approximation) and the equivalent to a single precision floating point mantissa (0.24). Furthermore, we selected the decimal logarithm. In the 0.24 case we assumed an already normalized input value.

In the Xilinx case we used ISE Webpack 7.1 (7.1.03i H.41) tool for synthesis, and the Virtex 4 LX (xc4vlx15-12-ff668) device to measure the speed. We disabled the use of BRAMs and DSP48 units, selected area optimization and set the optimization effort to high. We did not use any time constrain. Note that the consumed resources (FFs and LUTs) for the Virtex 4 are approximately the same for smaller devices like Spartan II.

Table 2 shows the error estimated using the method described in section four, the measured error and the number of clocks needed to compute the result. We measured the error using 100,000 random values. The first two values for the clocks are the minimum and maximum number of clocks needed to compute the result (latency), the third value is the average number of clock cycles between output results when the core is constantly fed with random values (throughput).

Table 3 shows area usage and maximum operating frequency reported by the tool after the place and route. The values for the area are flip-flops, 4 inputs LUTs, number of LUTs used as a route-thru and slices.

In the Actel case we used Libero 7.2 (Synplify Pro 8.5F, Build 001R and Actel Designer 7.2.0.31), and the RT54SX72S-1-208CQFP device. We selected a maximum fanout of 20 to optimize the used area. Table 4 shows area usage and maximum operating frequency reported by the tool after the place and route. The values for the area are flip-flops, combinational cells and the percentage of area used. Note that Actel devices use a simpler combinational unit, but they provide two of them for each flip-flop.

Fig. 3 shows the measured error distribution for the 0.24 case, as mentioned above, it was obtained using 100,000 random values. The bar for -4 ulp can not be observed because only four values presented this error.

Table 3. Implementation Results (Xilinx)

In	*Out*	*Area* *(FF/LUT/RLUT/Slices)*	*Max. freq.*
8.16	4.16	150/281/ 8/155	176 MHz
16.16	4.16	158/290/ 8/160	170 MHz
0.15	4.15	128/251/ 8/141	172 MHz
34.7	5.10	138/217/ 5/123	191 MHz
0.24	0.26	166/409/13/221	144 MHz

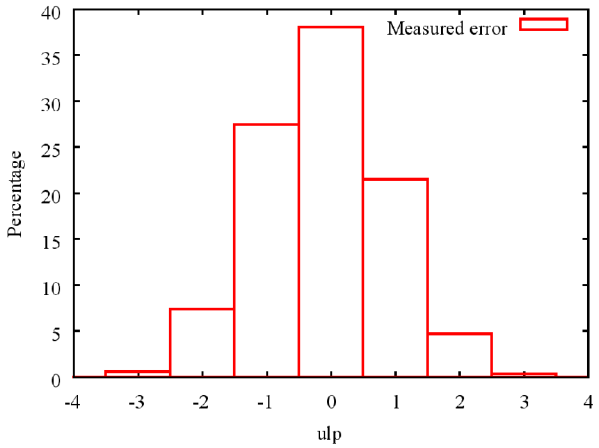

Fig. 3 Error distribution for the 0.24/0.26 case.

6. CONCLUSION

The proposed algorithm could be implemented on FPGAs from any vendor. As a result, it does not rely on specific resources like BRAMs, embedded multipliers, *etc*.

The used area was small. For example when the 8.16 configuration was used for a decimal logarithm, it was around 13% of a 100,000 gates equivalent device and less than 1/500 of a modern Virtex 4 LX 200.

When we used the selected Virtex 4, this core computed an average of 9.78 million values per second for the 8.16 configuration and over than 5 million values, when the output needed 23 exact bits. The last case needed three extra bits and provided the precision needed for a single precision floating point number.

The core showed a very good area-speed relationship, and for applications where the speed requirements are bigger more than one core could be used in parallel.

7. ACKNOWLEDGMENTS

We would like to thank to Gustavo Sutter (UAM, Spain), he contributed valuable ideas and references.

Table 4. Implementation Results (Actel)

In	*Out*	*Area* *(Reg./Comb./Area%)*	*Max. freq.*
8.16	4.16	154/500/10,8 %	33 MHz
16.16	4.16	162/505/11,1 %	32 MHz
0.15	4.15	129/446/9,5 %	32 MHz
34.7	5.10	139/372/8,5 %	39 MHz
0.24	0.26	174/747/15,3 %	23 MHz

8. REFERENCES

[1] M. Haselman, M. Beauchamp, A. Wood, et al, "A Comparison of Floating Point and Logarithmic Number Systems for FPGAs," *Proc. of the 13th Annual IEEE Symp. on Field-Prog. Custom Comp. (FCCM'05) 0-7695-2445*, Jan. 2005.

[2] Y. Wan and C. L. Wey, "Efficient algorithms for binary logarithmic conversion and addition," IEE Proc.-Comp. Digit. Tech., vol. 146, no. 3, May. 1999.

[3] T. C. Chen, "Automatic computation of exponential, logarithms rations and square roots," *IBM J. Res. Develop.*, pp. 380–388, Jul. 1972.

[4] H.-Y. Lo and Y. Aoki, "Generation of a precise binary logarithm with difference grouping programmable logic array," *IEEE Trans. Comput.*, vol. C-34, pp. 681–691, Aug. 1985.

[5] S.-Y. Shi, "Shortcut to logarithms combine table lookup and computation," Comput. Design., pp. 186–189, May. 1976.

[6] I. Koren, "Computer arithmetic algorithms, 2nd edition," ISBN 1-56881-160-8, pp. 225-232.

[7] D. K. Kostopoulos, "An algorithm for the computation of binary logarithms," *IEEE Trans. on Comp.,* vol. 40, no. 11, 0018-9340/91, Nov. 1991.

[8] B. Parhami, "Computer Arithmetic Algorithms and Hardware Designs," ISBN 0-19-512583-5, pp 378-381.

[9] M. Pascale, "Microcontrollers & CORDIC methods,", Dr. Dobb's Journal, http://www.ddj.com/184404244, Jul. 2001.