Table-Driven Floating-Point Exponential Function

Hung Tien Bui, Bashar Khalaf, and Sofiène Tahar

Technical Report

Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3H 1M8 Canada
E-mail: {ht_bui,b_khalaf,tahar}@ece.concordia.ca

October 1998

Abstract

This work aims at designing a floating-point exponential function using the table-driven method. The algorithm was first implemented using sequential VHDL and later translated to concurrent Verilog. The main part of the project consisted of creating modules that would handle basic IEEE-754 single precision number manipulation routines such as addition, multiplication, and rounding to nearest integer. Using these routines, a model was implemented using the described algorithm. The VHDL design, as well as the Verilog design, were simulated and the results proved to be satisfactory.

Contents

List of Figures

1 Introduction

The last two decades have brought extraordinary advances in numerical calculations. The improvements in hardware and in execution speed have contributed to the great progress in mathematical calculation speed and accuracy. Along with these advances came the development, in 1985, of a format that would revolutionize the world of science: the IEEE-754 single and double precision formats. The numbers represented under these formats, which are respectively 32 and 64 bits in length, have a greater range than their 2's complement counterparts. In the early days, there was no hardware available to perform floating-point arithmetics. The only way to perform these operations would be to write software routines. Unfortunately, the creation of such programs is rather complex and is not a trivial task for most people. Furthermore, the execution speed would be very slow when compared to a hardware implementation.

The interest then shifted to hardware design of such mathematical modules. The objective is to create an integrated circuit that would handle the transcendental mathematical functions in the IEEE-754 single and double precision formats. The project described below outlines the work that was put into creating the hardware implementation of an exponential function. The design of the circuit was done using two different hardware description languages, namely Verilog and VHDL. Although the programming was made to follow the algorithm outlined in [2] and [1], several changes had to be made to accommodate our single-precision implementation, in contrast to the reference, which used double-precision operations.

2 Floating-Point Exponential Function

2.1 Introduction

The table-driven implementation of the exponential function used by Ping Tak Peter Tang [2] consists of three main parts. The input value is first reduced to a certain working range. A shifted exponential function is then estimated using known polynomial approximations. Finally, the exponential function of the original input is reconstructed using a certain formula.

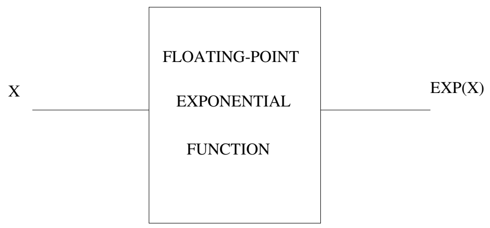

Figure 1: "Black-Box" Representation

The input X can be seen as being composed of many parts [2]:

$$x = (32 * m + j) * (log2)/32 + (r1 + r2) \tag{1}$$

where $|r1 + r2| \leq (log2)/64$, m and j are integers, and $r1$ and $r2$ are real numbers. Note that all logarithmic functions are, in reality, natural logarithmic functions (base e)

The polynomial approximation required is that of $exp(r) - 1$ which can be expressed as a Taylor series.

$$p(r) = r + a1 * r2 + a2 * r3 + ... \tag{2}$$

where $a1$ and $a2$ are the coefficients and r is the variable of the Taylor series.

The exponential function can then be reconstructed in the following manner starting from equation (1) and equating $r = r1 + r2$:

$$x = (32 * m + j) * (log2)/32 + r \tag{3}$$

$$x = m * log2 + (j * log2)/32 + r \tag{4}$$

$$exp(x) = exp((m * log2) + (j/32) * log2 + r) \tag{5}$$

$$exp(x) = exp(log2^m + log2^{j/32} + r) \tag{6}$$

$$exp(x) = exp(log2^m) * exp(log2^{j/32}) * exp(r) \tag{7}$$

$$exp(x) = 2^m * 2^{j/32} * (p(r) + 1) \tag{8}$$

$$exp(x) = 2^m * (2^{j/32} + 2^{j/32} * p(r)) \tag{9}$$

The objective of the algorithm would then be to isolate m and j and find the coefficients for the polynomial ($a1$ and $a2$).

2.2 Description of the Algorithm

The algorithm begins by checking for exceptional inputs. These are inputs for which the algorithm would yield an incorrect or undetermined answer. Examples of these type of exceptions are NaN (not a number), both infinities, an upper limit for which the output is positive infinity and a lower limit for which the output can be approximated by simple arithmetic operation. If the input does not fall into one of the previously mentioned categories, the program continues.

The next step is to calculate the values for m and j. The required values can be obtained by first multiplying equation (1) by a value of $32/(log2)$, known as INV_L. Performing these operations on equation (1), we get

$$X * 32/(log2) = (32 * m + j) + (r1 + r2) * 32/(log2) \tag{10}$$

Knowing that the value of $(r1 + r2) * 32/(log2)$ cannot exceed 0.5 (because $r1 + r2 \leq (log2)/64$), the previous equation can be approximated quite accurately by

$$INTEGER(X * 32/(log2)) = (32 * m + j) \tag{11}$$

The left side of the equation can easily be solved and will be assigned the letter N. Using the modulo-32 function, the values for $(32 * m)$ and j, named $N1$ and $N2$, respectively, can be calculated with little or no error.

Equation (11) can be rewritten as

$$N = N1 + N2 \tag{12}$$

where $N = INTEGER(X * 32/(log2))$, $N1 = 32 * m$, and $N2 = j$

The variables m and j can be derived from the previous results as follows:

$$m = N1/32 \tag{13}$$

$$j = N2 \tag{14}$$

With the value of N in hand, the value of $r1$ can be calculated as follows [2]:

If the absolute value of $N < 2^9$

then

$$r1 = (X - N * L1) \tag{15}$$

else

$$r1 = (X - N * L1) - N2 * L1 \tag{16}$$

The value of $r2$ is obtained by

$$r2 = -N * L2 \tag{17}$$

$L1$ and $L2$ are constants that can be added together in order to approximate 32/(log 2). The reason for separating the value into two constants is to increase the accuracy to one that is higher than that of single precision.

The sum of $r1$ and $r2$ will represent the scaling of the input X to a value r in the interval [-(log 2)/64, (log 2)/64]. Using the value of r in the Taylor series (2), the function of $exp(r) - 1$ can be approximated. For convenience purposes, only the first three elements will be considered.

The following step consists of forming the Taylor series which can be done in many ways. The implemented algorithm first calculated the second and third order elements. The coefficients $a1$ and $a2$ are constants that were calculated by Ping Tak Peter Tang using a Remez algorithm [2].

Examination of equation (9) reveals that only two more values need to be calculated in order to obtain the final result: 2^m and $2^j/32$. The table-driven implementation was named so because of the fact that the values for $2^j/32$, j ranging from 0 to 31, were calculated beforehand and stored in a table. These numbers are broken down into two parts, namely s_lead and s_trail, to increase the precision roughly by an order of two. With these values known, the final result can be determined, without difficulty, using equation (9).

3 Hardware Implementation of the Algorithm

The implementation of the algorithm, was done using two different hardware description languages, namely, Verilog and VHDL. The part constructed using VHDL made use of the sequential mode in contrast to the Verilog implementation that used combinational logic. Both essentially implement the same algorithm outlined in the previous section.

The VHDL design is composed of numerous procedures that perform IEEE 754 operations. These operations include the addition, multiplication, division by 32, rounding to the nearest integer, modulo 32, comparison and powers of 2. To ensure that the code is synthesizable, the program was made primitive and the length was much greater than it needed to be. A detailed description of each procedure follows.

Figure 2 contains the algorithm written by John Harrison [1] used in this implementation of the table-driven method for the floating-point exponential function.

```
Int_32 = Int(32)
Int_2e9 = Int(2 EXP 9)
Plus_one = float(0, 127, 0)
THRESHOLD_1 = float(0, 134, 6066890)
THRESHOLD_2 = float(0, 102, 0)
Inv_L = float(0, 132, 3713595)
L1 = float(0, 121, 3240448)
L2 = float(0, 102, 4177550)
A1 = float(0, 126, 68)
A2 = float(0, 124, 2796268)

var x:float, E:float, R1:float, R2:float, R:float, P:float, Q:float,
S:float, E1:float, N:Int, N1:Int, N2:Int, M:Int, J:Int;

if Isnan(X) then E:=X
else if X == Plus_infinity then e:= Plus_infinity
else if X == Minus_infinity then e:= Plus_zero
else if (abs(X) > THRESHOLD_1 then
  if X > Plus_zero then E := Plus_infinity
  else E := Plus_zero
else if abs(X) < THRESHOLD_2 then E := Plus_one + X
else
(N := INTRND(X * Inv_L);
 N2 := N % Int_32;
 N1 := N - N2;
 if abs(N) ≥ Int_2e9 then
  R1 := (X - Tofloat(N1) * L1) - Tofloat(N2) * L1
 else
  R1 := X - Tofloat(N) * L1;
 R2 := Tofloat(N) * L2;
 M := N1 / Int_32;
 J := N2;
 R := R1 + R2;
 Q := R * R (A1 + R * A2);
 P := R1 + (R2 + Q);
 S := S_Lead(J) + S_Trail(J);
 E1 := S_Lead(J) + (S_Trail(J) + S * P);
 E := Scalb(E1, M)
)
```

Figure 2: Exponential Function Algorithm

3.1 Addition

The addition procedure covers both the addition and the subtraction operations. The idea is mainly the same for both but handling both cases together is an added degree of complexity. The algorithm puts both numbers to the same exponent, adds or subtracts the numbers and then normalizes. Figure 3 shows the block diagram of the function.

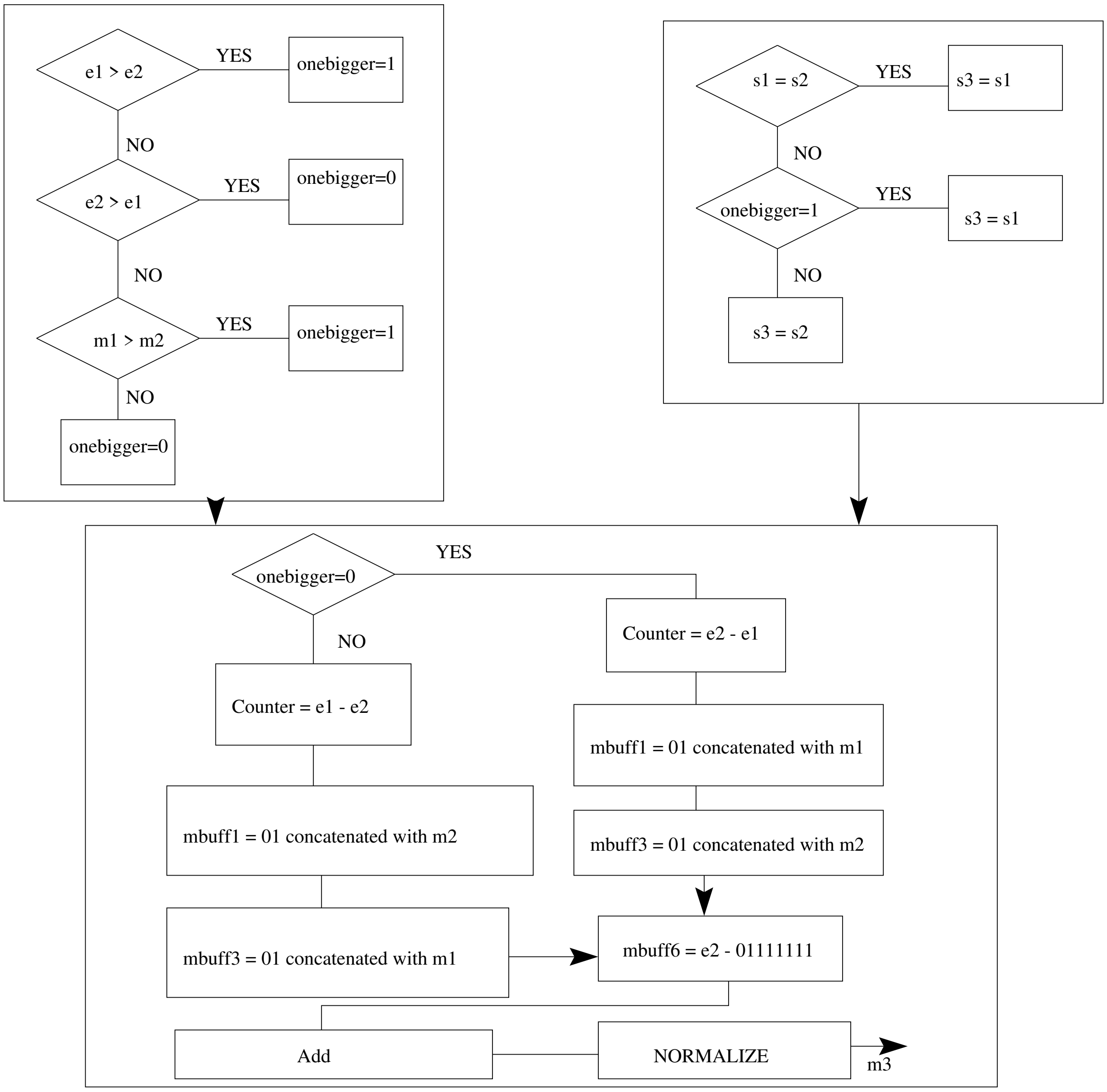

Figure 3: Addition Block Diagram

The first part of the addition procedure checks which input is greater. This is especially important in cases where the inputs are of opposite signs. If the inputs carry the same sign, the output sign will then be the same. When the signs are different, the input with the greater magnitude will impose its sign. The next step is to denormalize both inputs and perform the addition. However, before going on to that step, "01" has to be concatenated to both numbers. The reason for this is that the 1 is the implied 1 contained in the IEEE 754 format. The 0 is there to make sure that the carry bit is not lost. Denormalizing is done by right-shifting the smaller input by an amount determined by the difference in exponents. The exponent is unbiased by removing 127 ("01111111") from its biased value. Addition is then performed normally and the last part is normalizing. It would have been more convenient to use FOR loops for denormalizing purposes but the code would have been more dense and significantly more complex.

3.2 Multiplication

Multiplication is an operation that is quite straight-forward. Its algorithm is divided into three main parts corresponding to the three parts of the single precision format. The first part, the sign, is determined by an exclusive-OR function of the two input signs. The exponent of the output, the second part, is calculated by adding the two input exponents. And finally, the significand is determined by multiplying the two input significands each with a "1" concatenated to it. The result obtained will have about twice as many bits as the significand should normally have and so, the result will be truncated, normalized and the implied "1" will be removed (see Figure 4 for the block diagram). The normalization process will be fairly simple knowing that the multiplication of two 24 bit numbers with a one at the most significant bit position will yield a result with a one at the most significant bit (bit 47) or at bit 46. Depending on the situation, the result will either be shifted once or twice.

At the beginning of the algorithm, there is an IF statement that checks for exceptional cases where there is a zero in at least one of the inputs. Since inputs such as "zero", "NaN" (Not a Number) and both infinities are determined by a specific bit pattern, they have to be treated separately as it will be described shortly.

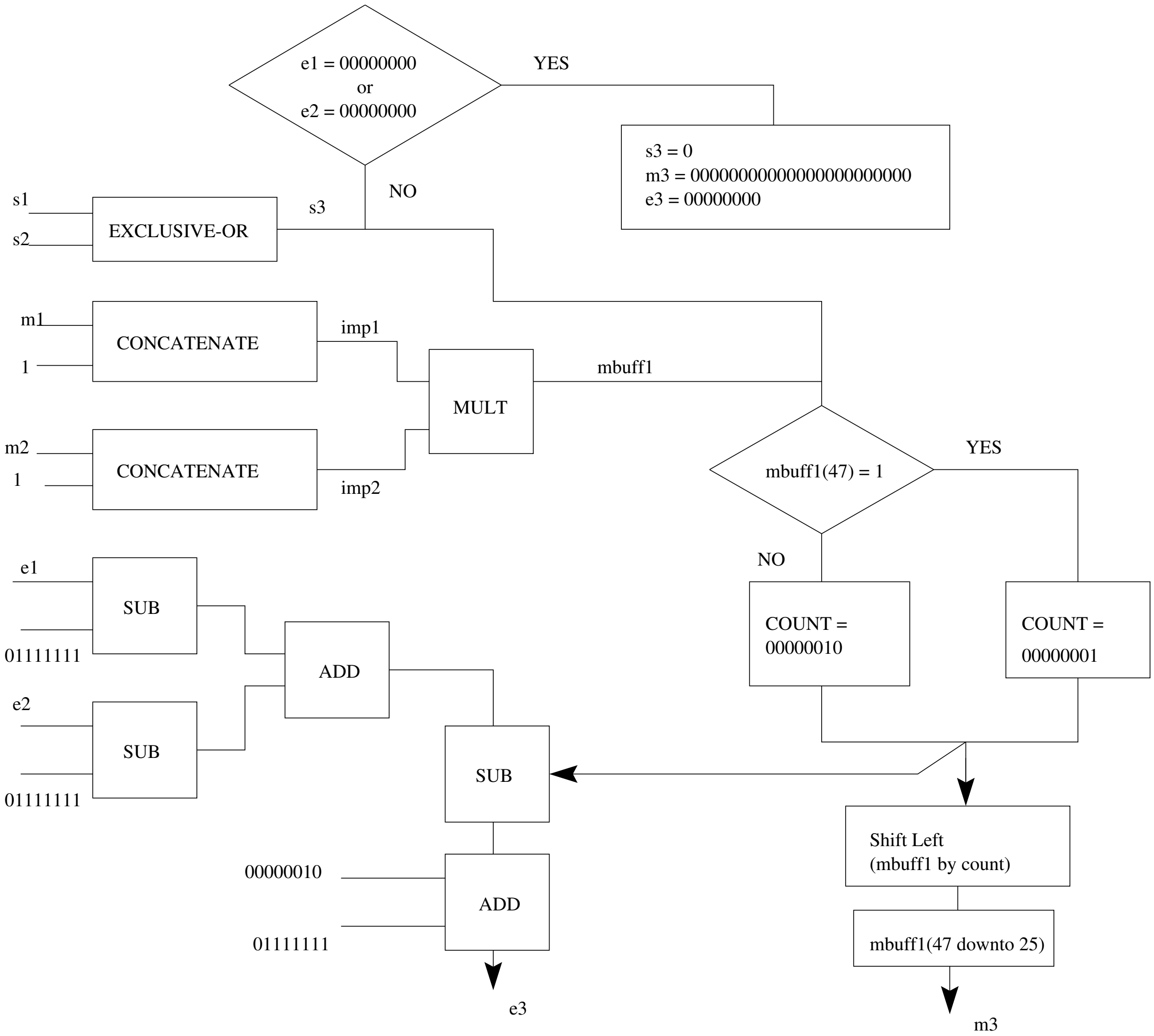

Figure 4: Multiplication Block Diagram

3.3 Division by 32

This function is only required to be used on a specific type of numbers: multiples of 32. Knowing this fact, the procedure does not need to support all possible ranges of inputs. The operations performed can be explained as follows: the algorithm will output zero if the input exponent is less than five and will simply subtract five from the exponent if it is not the case. This can be seen in Figure 5.

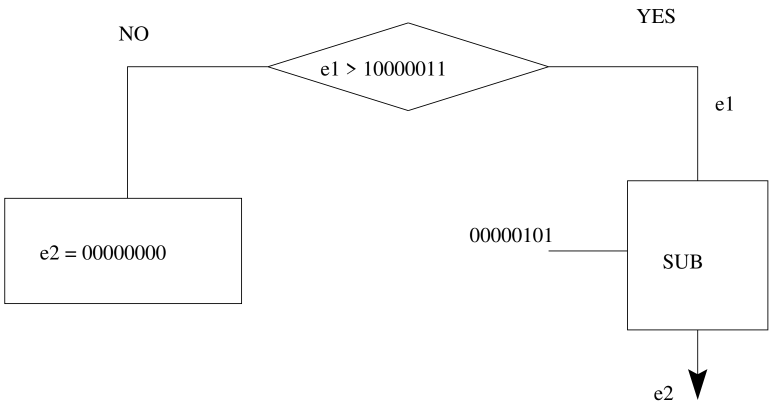

Figure 5: Division by 32 Block Diagram

3.4 Round to Nearest Integer

The algorithm starts by checking if the exponent is of the order of -2 or less. This would result in an output of zero. The second case is to check if the exponent is -1 in which case the output would be equal to 1. These two IF statements are for negative exponent handling since the main algorithm cannot deal with these cases.

The basic idea is to verify the bit at the 0.5 position. If the bit is set, the decimal positions are filled with zero and we add one to the resulting integer. If the bit is reset, the bits located to the right of the decimal point will be reset. To accomplish this, the input is first shifted right by a number of positions corresponding to the exponent (so that all the fraction bits are shifted out). The number obtained should be an integer. This number is then incremented by one if the bit at 0.5 is set else it should be left the same. Figure 6 illustrates this.

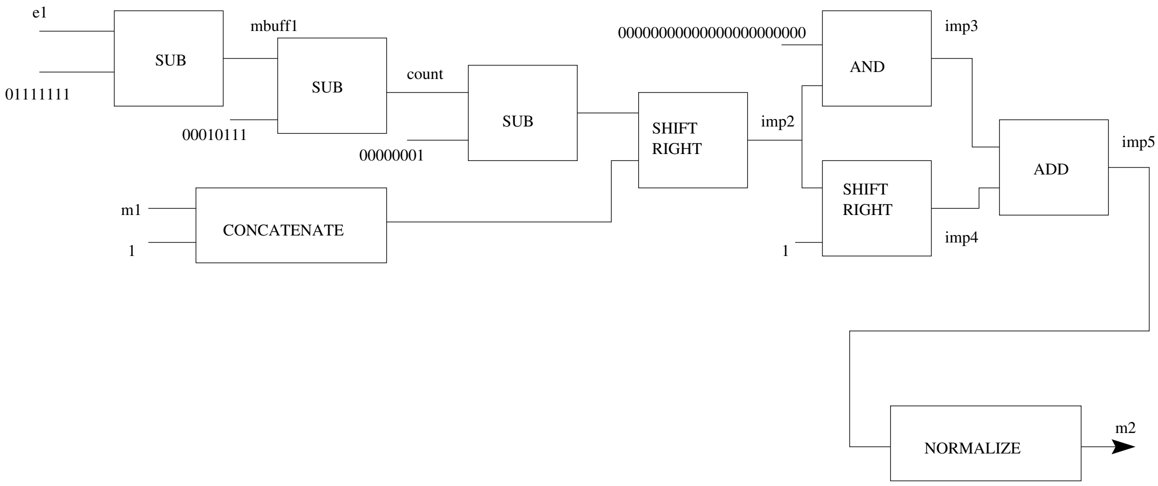

Figure 6: Round to Nearest Integer Block Diagram

3.5 Modulo 32

Modulo 32 is an operation that is done by simply taking the five first bits located to the left of the decimal point. The result will then be an unsigned 5-bit integer that will have to be converted to single precision format. For the block diagram, refer to Figure 7. The lower right portion of the figure shows a loop that was not actually implemented in the algorithm. It was drawn like that in order to reduce the complexity of the diagram. The variable "I" is used as a loop variable.

The procedure is somewhat similar to that of rounding to the nearest integer. The input is first shifted right by the number of bits corresponding to the exponent. The result is then ANDed with the "11111" bit pattern in order to isolate the five bits. The conversion process checks where the first 1 is located starting from the most significant position. An exponent is then assigned accordingly and the result is shifted left to comply with the rules of normalization.

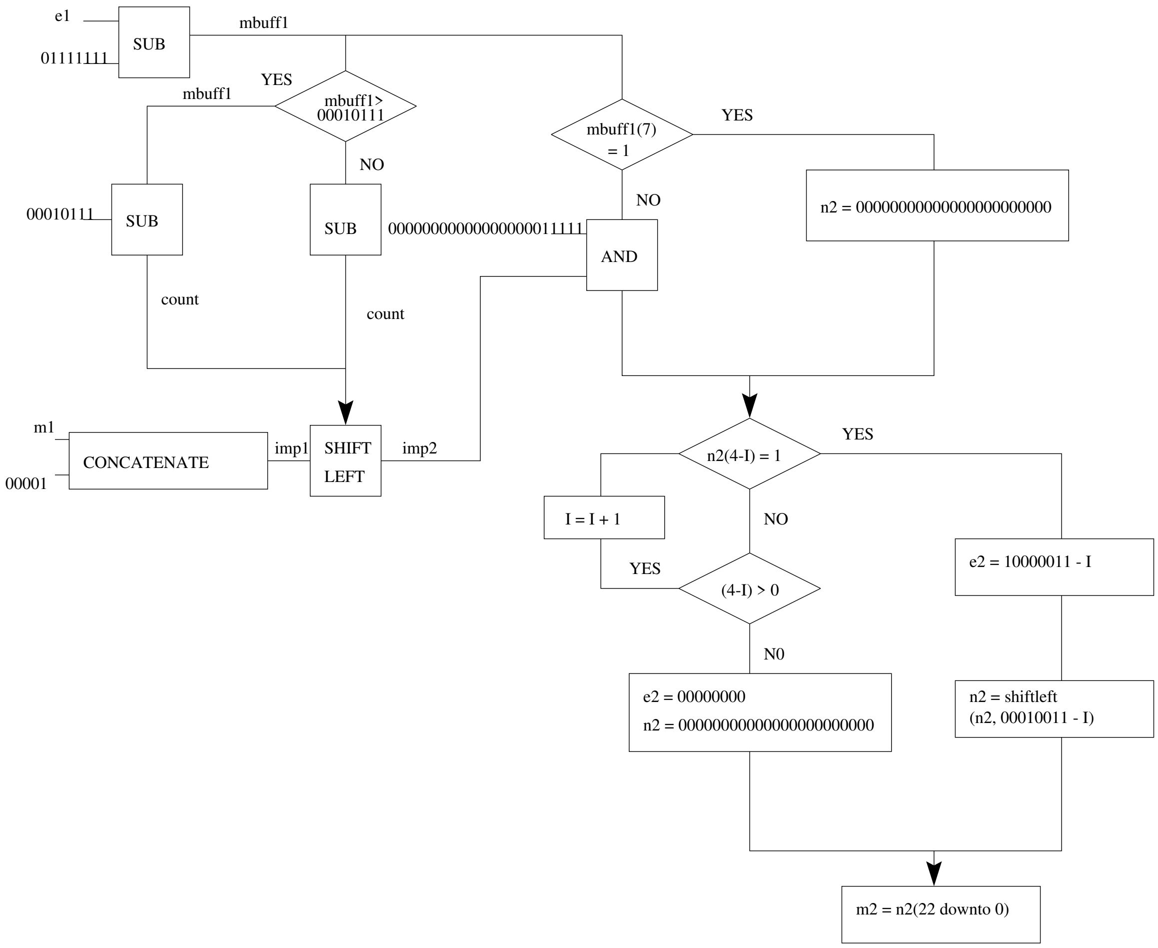

Figure 7: Modulo-32 Block Diagram

3.6 Comparison

Unlike the other procedures, the comparison does not output a number in the IEEE 754 format. Instead, it gives three bits that gives a comparative indication of the size of the first input with respect to the second input. If the first input is greater than the second one then the most significant bit is set. If the second input is greater than the first, then it is the least significand bit that is set. If the two inputs are equal then the middle bit is set. Only one bit can be set at any given time. Figure 8 shows the block diagram of this function.

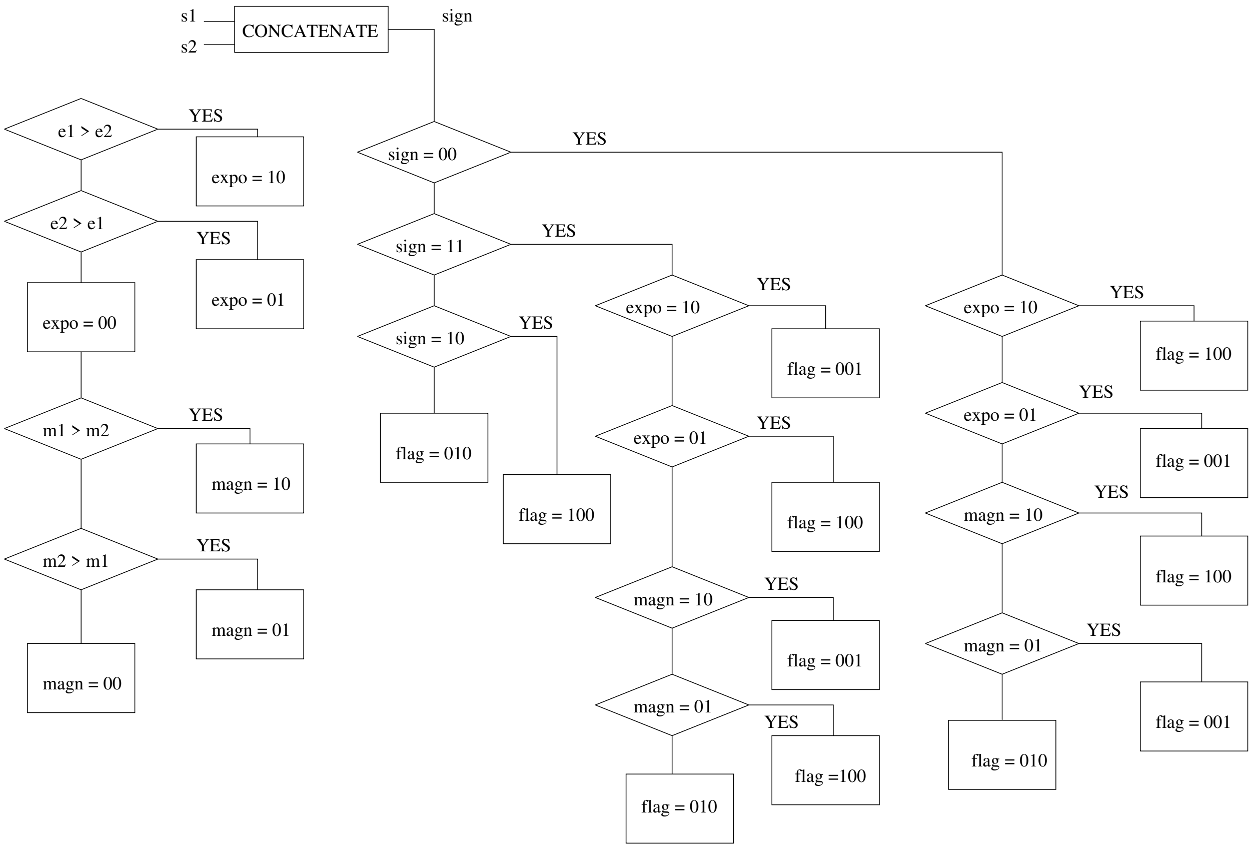

Figure 8: Comparison Block Diagram

3.7 Powers of Two

The powers of two function can be implemented by realizing that the value of the input is the value of the output exponent. For example, placing four as an input would result in two to the power of four, yielding four in the exponent field. The objective of the function would then be to convert the input, being an IEEE 754 number, to a 2's complement number. The bias of 127 would then be added to the result and the sum would be placed in the exponent field. The sign and significand fields will be filled with zeros because the result will always be positive and will always be an integer multiple of two. A detailed block digram of the Power of Two function is given in Figure 9. In this figure, there is a loop in the lower left section that is used to provide a concise description of the algorithm. It uses the variable "I" as a loop variable.

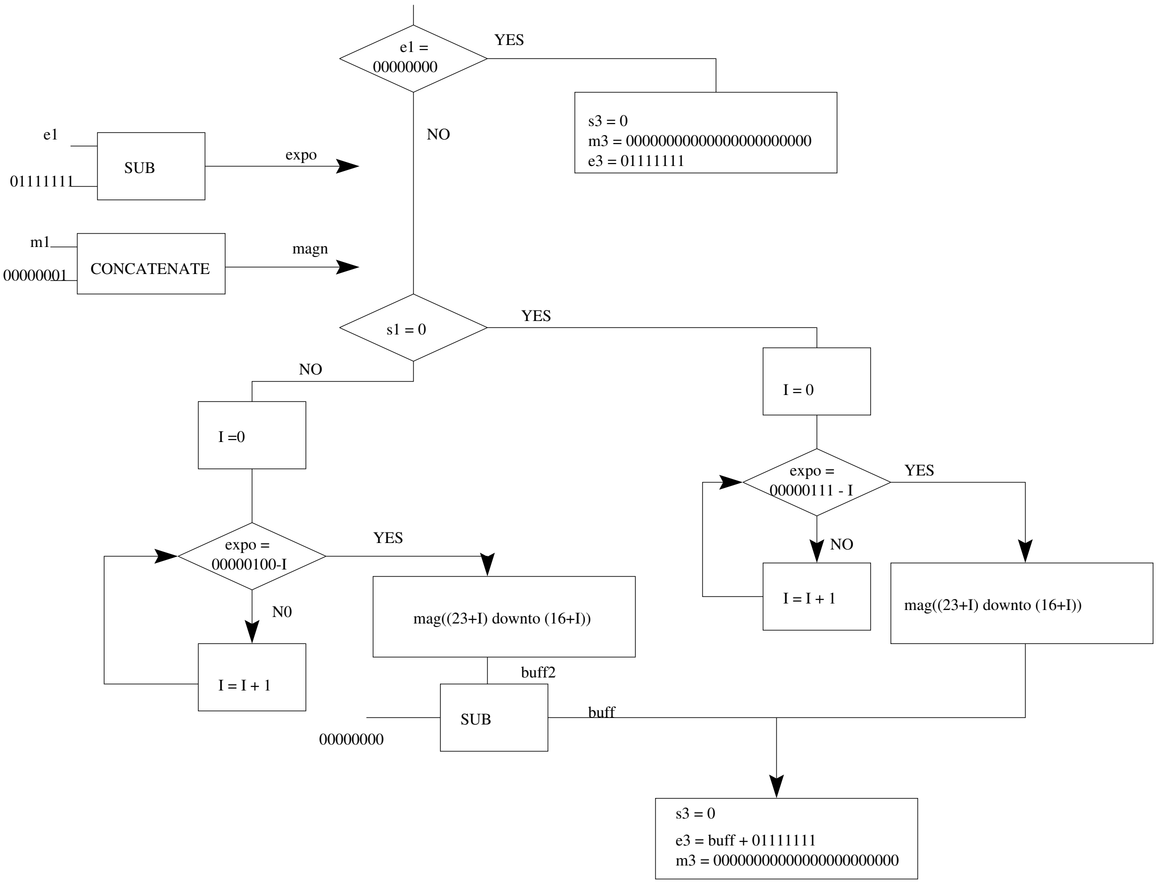

Figure 9: Powers of Two Block Diagram

3.8 Get J

The current implementation of the exponential circuit is the table- driven implementation. The table index should ideally be an unsigned integer to make the search easier. The "Get J" procedure takes care of this. It takes a number in the single-precision format and transforms it to an unsigned number. The procedure examines the exponent and extracts the corresponding bits from the significand. Even though the source code uses a series of IF statements, the block diagram in Figure 10 shows

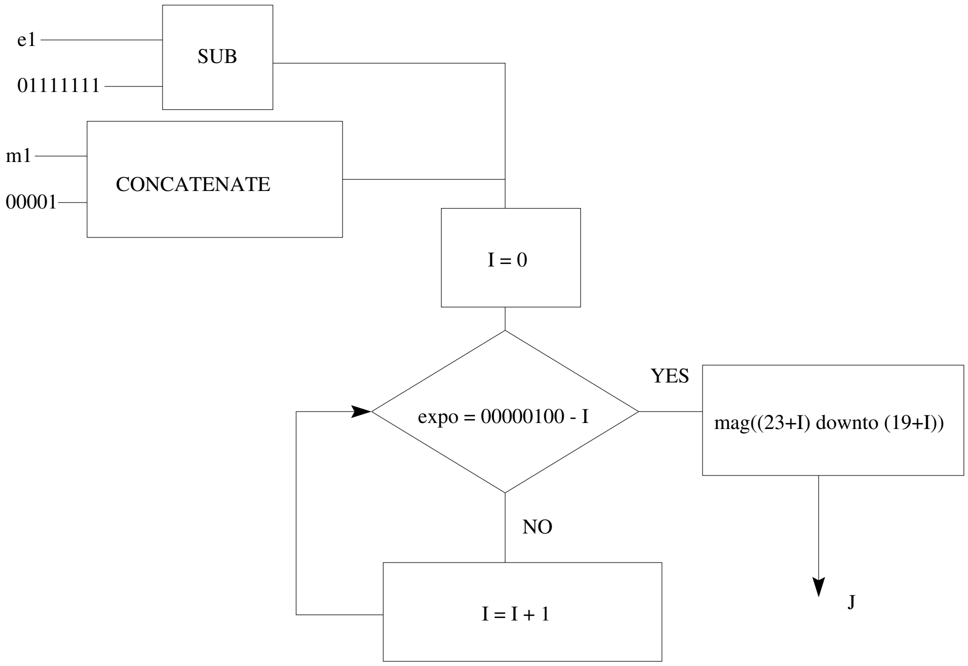

Figure 10: Get J Block Diagram

a loop that uses a variable "I" to perform the required task. Using an unsigned number for the search makes the task of finding a correct value for S easier (refer to the algorithm described in Figure 2).

3.9 Modifications and Remarks

The algorithm described by Ping Tak Peter Tang [2] used single precision in combination with double precision in order to achieve better accuracy. The work presented here does not cover double precision calculation and thus, several changes will have to be made.

The beginning of the source code contains a multitude of IF statements checking for special case considerations. One of those cases is an upper limit threshold beyond which the output goes to infinity. The second case is a lower limit threshold that checks to see if the input is low enough for the following approximation to hold: OUTPUT = 1 + INPUT. The lower limit can be left the same without any major consequences. However, the algorithm overflows for inputs far smaller than the upper limit and thus, the boundary had to be changed. The value for Threshold_1 was modified to 89 from its initial value of about 220.

In addition, modifications had to be made to the modulo-32 function which operates differently with negative numbers. The algorithm needs the output of this function to be positive and so, negative results will have 32 added to them.

The modules described above were used as building blocks to construct the floating-point exponential function. A block diagram of the whole system is shown in Figure 11. In this diagram, we use the same labels as in the algorithm in Figure 2.

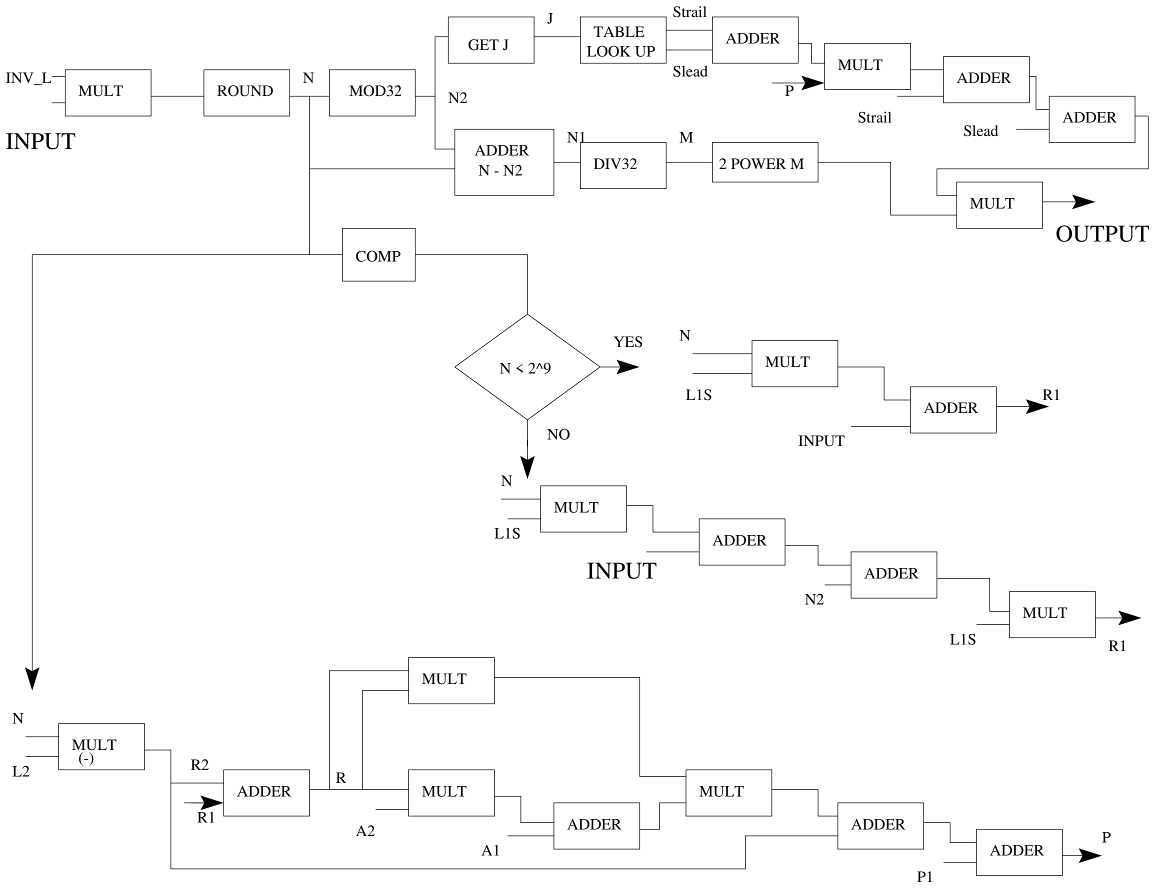

Figure 11: Main Block Diagram

4 Simulation Results

After understanding the algorithm and writing a program to implement it, the next step is to verify the accuracy of the outputs. The verification is done by comparing the expected results obtained using a normal calculator to the output generated by the algorithm.

The analysis was performed on 20 test vectors covering a widely-used range of inputs. The results obtained are tabulated in 1 below.

Input	Output	Expected
$+\infty$	$+\infty$	$+\infty$
$-\infty$	0	0
Nan	Nan	NaN
0	1.0	1.0
0.25	1.284025430	1.284025417
0.5	1.648721218	1.648721271
1	2.718281745	2.718281828
2	7.389055728	7.389056099
5	148.4131622	148.4131591
10	22026.46679	22026.46579
15	3269017.25	3269017.372
20	485165184.0	485165195.4
-0.25	0.778800726	0.778800783
-0.5	0.606530666	0.606530659
-1	0.36787945	0.367879441
-2	0.135335296	0.135335283
-5	6.737946531E-3	6.737946999E-3
-10	4.539992731E-5	4.539992976E-5
-15	3.059023469E-7	3.059023205E-7
-20	2.061153691E-9	2.061153622E-9

Table 1: Experimental Results

As it can be seen, the outputs differ from the expected results by only a small margin. These errors can be attributed to different factors that include the errors in reduction, errors in approximation and rounding errors. Errors in reduction occur because of the mapping of the input to a range of values r between $(log2)/64$. The approximation errors are present because of the use of only the first three elements in the Taylor series to calculate the $e^r - 1$ function. The rounding errors are due to the many cases where numbers had to be rounded because single precision did not provide enough accuracy. A more detailed analysis is described in the work of Ping Tak Peter Tang [2]. The numbers given there are however not applicable here because this project does not cover double precision arithmetics.

5 Conclusions

The completion of a project like this one opens the way for a multitude of similar works aimed at accelerating the execution of transcendental mathematical functions by designing an integrated circuit. Even though the contribution made by this project is substantial, there is still a lot of room left for improvement in terms of accuracy and compactness of the code. Most modifications will, however, not have a great impact on the performance of the design. As a finished product, this project seems promising in that it can be integrated along with other similar modules to form something of a transcendental mathematical unit.

The next logical step to take involves the synthesis and formal verification of the prototype. With the design written in two different hardware description languages, namely VHDL and Verilog, synthesis will yield a different circuit for each language. From this point, verification procedures can be made to formally verify the different levels of abstraction and eventually, check if the RTL implementation implies the high-level specification.

The process of formally verifying the algorithm described in [2] has previously been targeted by John Harrison [1]. He used a version of the HOL prover, namely HOL Light, to perform this verification at a high level. In contrast to this, our work is based at a lower level and out goal is to verify that the implementation implies the algorithm.

References

[1] John Harrison. Floating-Point Verification in HOL Light: The Exponential Function. Technical Report number 428, University of Cambridge Computer Laboratory, UK, June 1997.

[2] Ping Tak Peter Tang. Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic. *ACM Transactions on Mathematical Software*, vol. 15, no. 2, 1989.