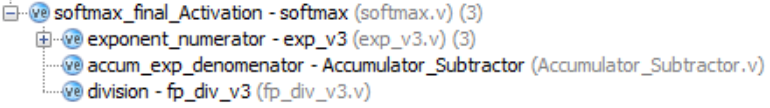
**Softmax**

Introduction:

Softmax is an activation function which is a normalized exponential function where it takes as input the vector from Fully Connected Layer and normalizes this vector into a probability distribution of probabilities proportional to the exponents of the input numbers.

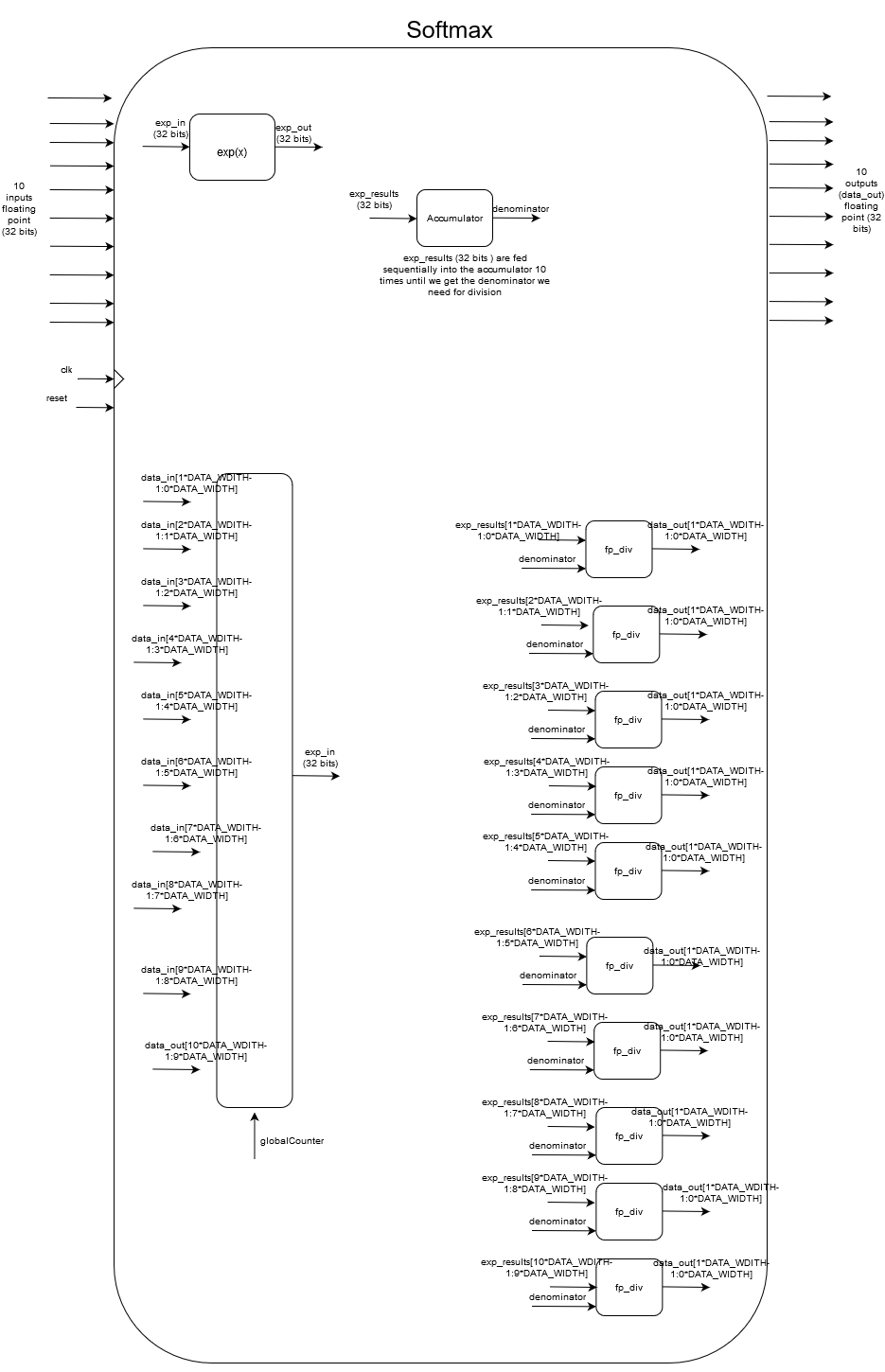
It is composed of three modules: (fp\_div, exp, accumulator)

Modules used:

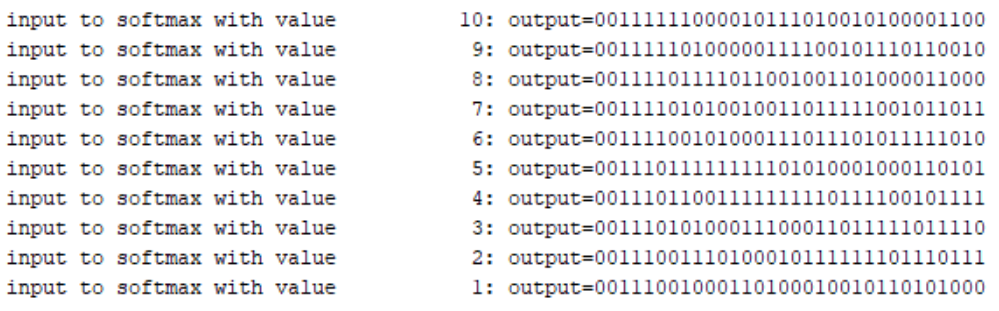
exp\_v3: calculates the exponent of one floating point input, it calculate the **first 10 terms** of the taylor series

Accumulator\_Subtractor: accumulates the what is input to it, which is used here to calculate the denominator of the equation of softmax which is the summation of exponentials of the inputs.

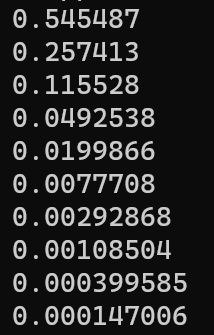
fp\_div\_v3: divides each output from the exp\_v3 by the divisor which comes out from the accumulator.

1)Block diagram:

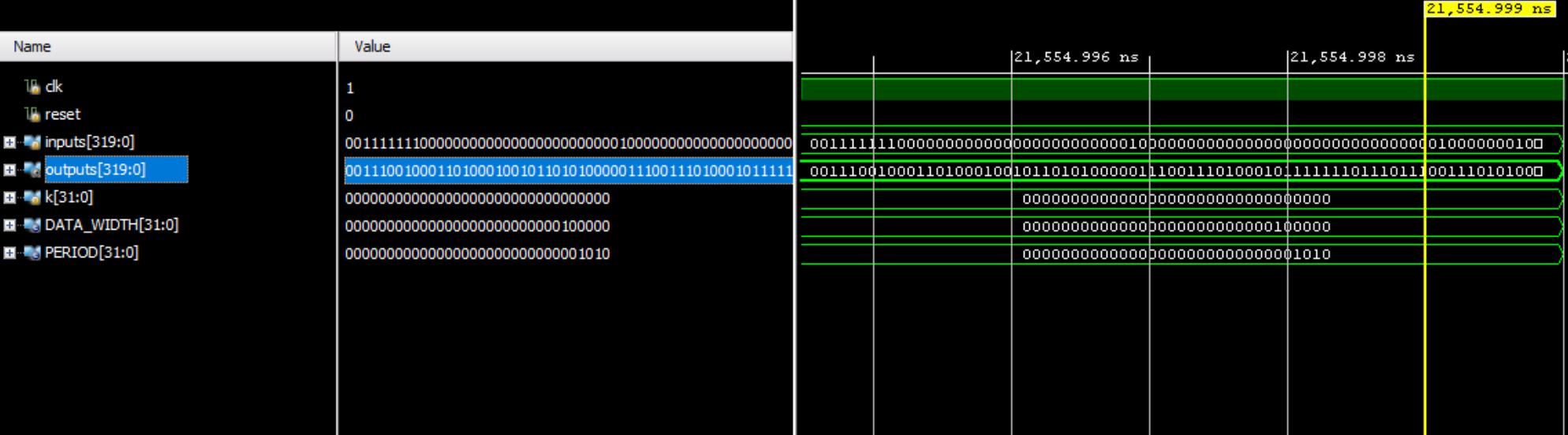
2) Simulation results: (softmax) module

The input vector to softmax tested is {10,9,8,7,6,5,4,3,2,1} and then for example the first line in the image below says “input to softmax with value 10:” so for this particular input which is 10, the output is “00111111000010111010010100001100” which is 0.54548 which is the same as we got from “**softmax.cpp**” to check the answers.

Result from TCL console from testbench (“softmax\_tb.v”)

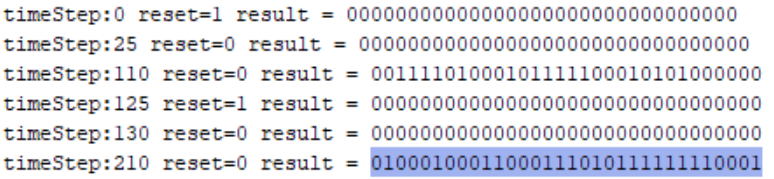


Results obtained from c++ program attached “softmax.cpp” which do the same as the Verilog module “softmax.v” do and the results match(correct). Results got from c++ program are in the image in the left.

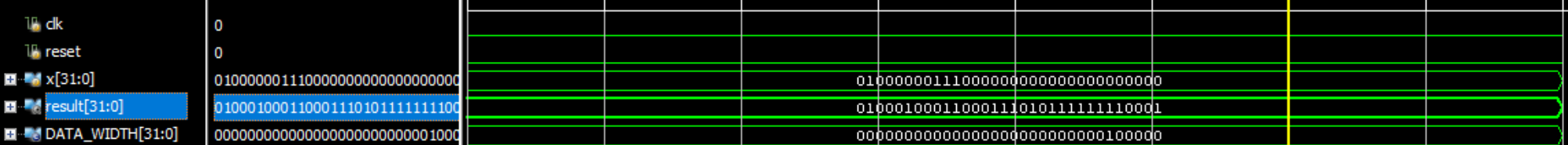
Waveform, results are in outputs, but it could be better seen or understood from the TCL console picture above.

Simulation results for exponent module (exp\_v3):

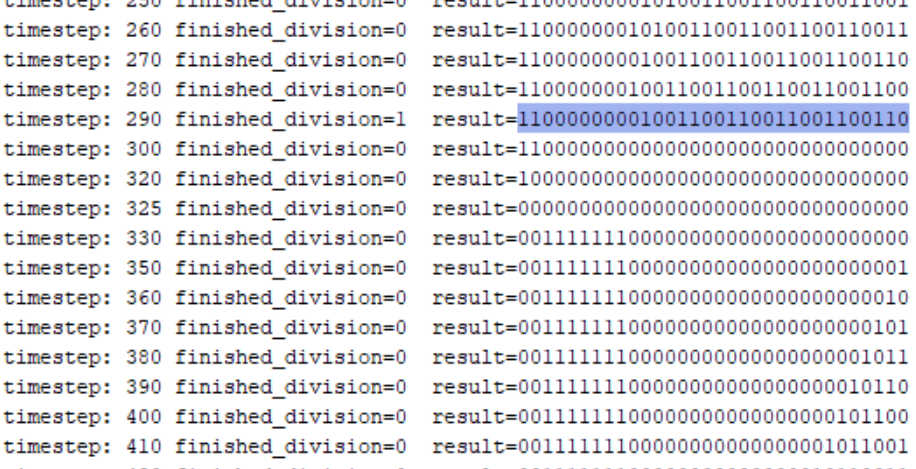
This module calculates the exponent of a number (e^x). It takes 10 cycles to finish as in each cycle, it computes one of the 10 terms.

The inputs tested in the testbench are -3 and 7. The outputs we got from the testbench are “00111101000101111100010101000000” and “01000100011000111010111111110001” for the inputs numbers respectively, which are equivalent to 0. 0370535 and 910.749 which are same as the outputs we got from c++ program “**exp.cpp**” I made to check the outputs.

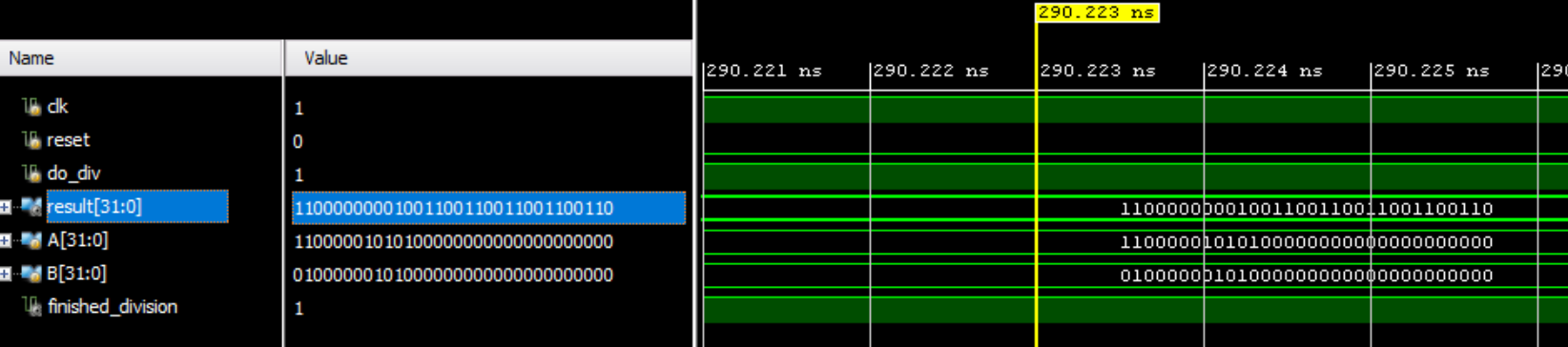
TCL console for testbench “exp\_v3\_tb.v”

Waveform from exp\_v3\_tb.v

Simulation results from division module (fp\_div\_v3):

The testbench “**fp\_div\_tb.v**” has many testcases some of the discussed here is -13/5 which should equal -2.6. There result got from testbench is -2.5999999 which is approximately -2.6. The is highlighted below in the screenshot along with a done flag.

Part of TCL console for test bench “fp\_div\_tb.v”

Waveform for the above testcase:

The rest of modules used in softmax were taken from phase1 so no change in them

3) Clock Cycles need for each module:

fp\_div\_v3.v(division): 28 clock cycles ( dividing the mantissas(24 cycles) + moving from s\_idle to s\_convert(2 cycles) + normalization(1 cycle) + resetting(1 cycle)).

**Notes: s\_idle and s\_convert are described in detail in comments at the top of the module “fp\_div\_v3.v” along with the explanation for how this module works. Also the same is done for the rest of the modules discussed in this part.**

Exp\_v3.v(exponential): 10 clock cycles as I am here calculating 10 terms described in the following equation: 1 + (x/1)( .....(1+ (x/6)(1 + (x/7)(1 + (x/8)(1 + x/9))))

Softmax.v: 140 clock cycles (10(for exp of each input) \* num of inputs(10)) + 10(accumulation) + 25(division) + 5(reset and counters)

4) Synthesis utilization reports: on pynq board

Synthesis reports can be view by CTRL+Click on the hyperlink, it is also present in a folder path to folder /softmax/synthesis\_reports/

Softmax: [..\synthesis\_reports\softmax\_synth\_report.txt](../synthesis_reports/softmax_synth_report.txt)

Exponential: [synthesis\_reports\exp\_v3\_synth\_report.txt](synthesis_reports/exp_v3_synth_report.txt)

Division: [synthesis\_reports\fp\_div\_v3\_synth\_report.txt](synthesis_reports/fp_div_v3_synth_report.txt)

Synthesis on KCU board:

Softmax: [synthesis\_reports\softmax\_synth\_report\_KCU.txt](synthesis_reports/softmax_synth_report_KCU.txt)

Exponential: [synthesis\_reports\exp\_v3\_synth\_report\_KCU.txt](synthesis_reports/exp_v3_synth_report_KCU.txt)

Division: [synthesis\_reports\fp\_div\_v3\_synth\_report\_KCU.txt](synthesis_reports/fp_div_v3_synth_report_KCU.txt)

5) Synthesis schematic:

Softmax: [schematics\softmax\_schematic\_rtl.pdf](schematics/softmax_schematic_rtl.pdf)

Exponential: [schematics\exp\_schemtaic.pdf](schematics/exp_schemtaic.pdf)

Division: [schematics\div\_schematic.pdf](schematics/div_schematic.pdf)

How softmax works:

Softmax module(softmax.v) uses three modules mainly exponential(exp\_v3.v), accumulator(Accumulator\_Subtractor.v), and floating point division(fp\_div\_v3.v). I will explain each of the modules used to make softmax till we reach the softmax itself and explain it.

*Eponential:*

The first is the exponential module (exp\_v3.v), which calculates the exponential of one 32 bits floating point input by computing the first ten terms of the taylor expansion. This calculation is done sequentially, as in each clock cycle it calculates one term, so the module will finish calculating the exponential after 10 clock cycles. The calculation of the ten terms was done as follows: we will rewrite the taylor expansion (1+x+x^2/2!+x^3/3!+….) by taking common factors so that it could be expressed as (1 + (x/1)( .....(1+ (x/6)(1 + (x/7)(1 + (x/8)(1 + x/9))))). So here we first start by calculating the inner most bracket which is 1+ x/9, where we need here an adder and two multipliers (x\*(1/9)\*1). Then the next term is calculated as follows 1+ x\*(1/8)\*(result of previous) and so on till finishing all terms. One of the multipliers takes as input a constant as (1/9,1/8,1/7,..) instead of using division here. The test bench (exp\_v3\_tb.v) has several numbers that were tested and checked using a c++ program (exp.cpp) attached to calculate the exponent by taylor expansion of first 10 terms.

*Accumulator:*

The second module used is the accumulator (Accumulator\_Subtractor.v). The use of this module in softmax is to calculate the denominator of softmax function which is the summation of the exponential of each element in the input vector. It takes ten clock cycles as in each cycle we accumulate one term (to save for utilization). The output from this module is called the divisor which will be used by the division module.

*Floating point division:*

The third module used the floating point division module (fp\_div\_v3.v). This module takes as input the one exponential output and the divisor and computes only one output 32 bits. It takes this module 28 clock cycles to divide two floating point 32 bits numbers ( dividing the mantissas(24 cycles) + moving from s\_idle to s\_convert(2 cycles) + normalization(1 cycle) + resetting(1 cycle)), detailed explanation is found at the top of the module itself presented as a comment . The calculation of division operation is done as follows: the sign of result is XORed, the exponents part of floating point are subtracted (exponent of dividend – exponent of divisor), then the final part is dividing the two mantissas (mantissa dividend / mantissa divisor). The division is done the same way as dividing two binary numbers by hand, where you compare part of the dividend to the divisor and if it is greater then put in the quotient 1 and subtract the dividend minus the divisor (and of course here we are talking about dividing the mantissas only of the floating point numbers), else if dividend is less than the divisor then a 0 will be put in quotient and we will shift the dividend to left and append a zero at the end and carry out same procedure of comparing and deciding whether to put a 0 or a 1 in the quotient at each clock cycle. At the end, there is the normalization part and see if there wasn’t a borrow in the last subtraction then will subtract one from the exponent. The test bench (fp\_div\_tb.v) has several test cases to test the division module.

*Softmax:*

Finally The softmax module (softmax.v) combines all three modules described above and takes the final output out from the division module to have the final calculated probability distribution. This module takes 1clock cycles to finish (10(for exp of each input) \* num of inputs(10)) + 10(accumulation) + 25(division) + 5(reset and synchronizing the inputs and outputs between modules). There is test bench (softmax.v) which tests this module on an input vector which contains {10,9,8,7,6,5,4,3,2,1} and gets the output correctly, and the output was checked using a c++ program (softmax.cpp) used to check the output from the test case in the test bench.

Research paper

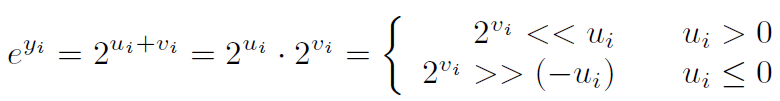
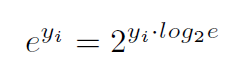
Title: A High-Speed and Low-Complexity Architecture for Softmax Function in Deep Learning (2018) ieee

Citation: M. Wang, S. Lu, D. Zhu, J. Lin and Z. Wang, "A High-Speed and Low-Complexity Architecture for Softmax Function in Deep Learning," 2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), Chengdu, 2018, pp. 223-226, doi: 10.1109/APCCAS.2018.8605654.

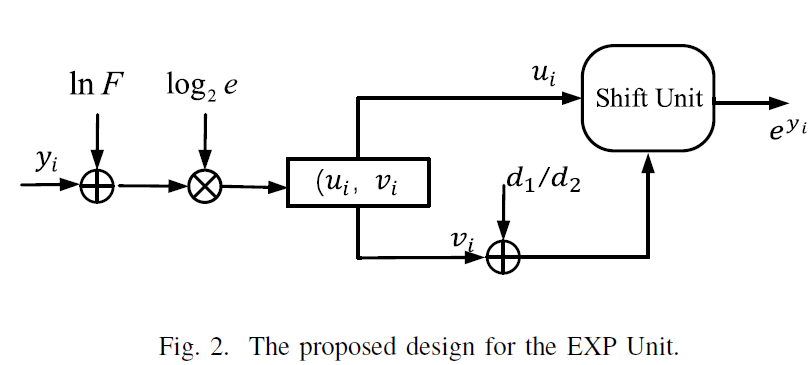
The main idea they discussed in this paper, it that they use mathematical transformations on the exponential and logarithmic functions into exp-log operations on a limited input range and other operations as constant multiplication(same was did in this project) shift and addition. Therefore, in calculating the exponential, they only used two adders and a constant multiplier. For them to implement the constant multiplication, they used carry save adders to optimize their use of adders.

Note: before they begin experimenting, they preprocess the input data and transform the floating point numbers into fixed point numbers

As the exponential function requires a lot of multiplications which will eventually cause a huge delay, mathematical transformations must be made on the exponential function. The exp function is rewritten as e^yi = 2^(yi.log2e), where yi.log2e is split into an integer number and decimal number, furthermore, the calculation is simplified more as in the picture on the right.

The benefit of this is that they transformed the range of inputs from (-∞,+∞) to be (0,1] which can be implemented using a LUT which will reduce the hardware complexity. The design is shown below.



As it appears from the block diagram, there are terms at which constant multiplication is needed as log2e. So, its calculation is using a number of carry save adders (the csa we studied in this course).

The Ln (natural logarithmic) was also discussed in this paper with its architecture, but our focus here was on mainly how they got to make the exponential function along with the mathematical transformations needed in order to achieve optimized design and reduced latency.