**A REPORT**

**ON**

**Implementation of Tanh function on FPGA**

**Area and Speed Optimization**

**By**

|  |  |
| --- | --- |
| **Name of the student** | **ID No** |
| **Shantanu Nigam** | **2017A8PS0399P** |



**Under the mentorship of**

**Dr Karri Babu Ravi Teja**

**Assistant Professor, EEE Department**

**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI**

**(May, 2021)**

Table of Contents

1. **Introduction3**
2. **Representation Strategy3**
3. **Interpolation Strategy5**
4. **Improving the Design8**
5. **Conclusion11**
6. **References12**

Table of Tables

1. **Accepted error vs Required bits5**
2. **Comparison of various interpolation schemes6**
3. **Interleaving vs Memory required for storing with 14 bits for fractional part 6**
4. **Interleaving vs Memory required for storing with 15 bits for fractional part7**
5. **Best Individual Parameters 9**
6. **Summary of resource utilization by various models made10**

Table of Figures

1. **Graph of tanh(x)3**
2. **Splitting tanh in working ranges4**
3. **Redundancy in stored memory values8**
4. **Splitting ranges in Model B9**
5. **Abstract Black Box representation of the final design11**
6. **Timing diagram of Model-B12**
7. **Introduction**

Deep Neural Networks (DNN) have been widely adopted in various applications such as object classification, pattern recognition and regression problems{fix:: put references journal paper which 2-3}. It is quite recently that FPGAs are becoming a tool for running pre-trained networks for faster processing{references, journal paper 2-3}. {fix:: explain dnn 2-3 sentences me explain, use hidden layed word in these sentences}. In the hidden layer of DNN, we have to perform weighted sum of the inputs from previous layer and pass the output through an activation function. These activation functions are nonlinear functions. Most linear operations like weighted sum are easily performed on digital devices like FPGA. However, nonlinear activation function poses a difficulty. Efficient implementation of non-linear activation functions is essential to the implementation of deep learning models on FPGAs. Most commonly used activation functions are sigmoid and tanh {list more activation functions}. {fix:: continue explaining tanh}

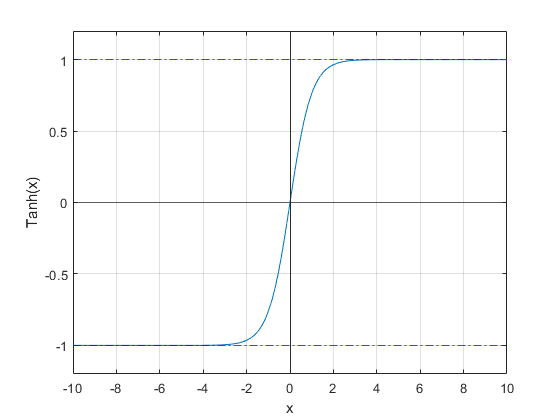
****

Figure 1: Graph of tanh(x)

In this paper we will present an accurate pipelined model of tanh activation function.

1. **Representation Strategy**

When designing the system, it is necessary to identify the type of input and output. The two options for representing fractional numbers are IEEE 754 floating-point representation and the fixed-point representation. While the floating-point can represent more range of values and is generally more flexible, however using it lowers performance and requires more hardware like floating-point unit (FPU). Small and simple operations like addition and subtraction become multicycle. The added large range does not help because the most dynamic characteristic of Tanh is visible is a very small range (0-6). Using fixed-point representation allows us to limit the number of bits we want to use, ease out the computational complexity of even multiplication which is generally multicycle. Thus, for a medium or large size FPGA, it is only better to use fixed-point representation.

Tanh is antisymmetric about the origin. Using this property, we will only perform computation on positive values. Negative values will be represented in 2’s complement format. If the MSB of any input is set, implying it is negative, we will work on its 2’s complement, perform the computations and get the result. The output will be 2’s complement of the computed result. To represent negative numbers, we will add one more bit at MSB and use the 2’s complement representation at both input and output sides.

Tanh can be divided into three ranges; linear, dynamic, and saturation.

In linear range: y = tanh(x) = x

In dynamic range: y = tanh(x)

In saturation range: y = tanh(x) = 1

The figure below illustrates the ranges.

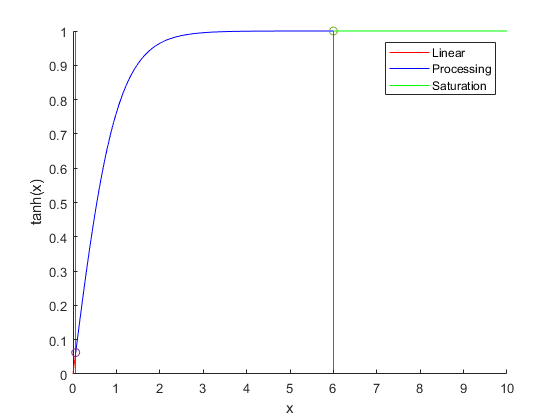


Figure 2: Splitting tanh in working ranges

There is no absolute limit to how many bits to be used to represent any value as it depends on the maximum tolerable error. The table below shows the linear, saturation, and working range for different tolerable error range.

Table 1: Acceptable Error vs Required bits

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OUTPUT | INPUT | | | | | OUTPUT | | |
| Maximum acceptable absolute error | Linear Range | Saturation Range | Bits required to represent Integer Part | Bits required to represent Fractional Part | Total bits  (1 extra sign bit is added) | Bits required to represent Integer Part | Bits required to represent Fractional Parted) | Total bits  (1 extra bit for sign) |
| 1.00E-02 | 0.25 | 3 | 2 | 7 | 10 | 1 | 7 | 9 |
| 1.00E-03 | 0.125 | 4 | 3 | 10 | 14 | 1 | 10 | 12 |
| 1.00E-04 | 0.0625 | 5 | 3 | 14 | 18 | 1 | 14 | 16 |
| 1.00E-05 | 0.03125 | 6 | 3 | 17 | 21 | 1 | 17 | 19 |
| 1.00E-06 | 0.015625 | 7 | 3 | 20 | 24 | 1 | 20 | 22 |
| 1.00E-07 | 0.0078125 | 8 | 4 | 24 | 29 | 1 | 24 | 26 |
| 1.00E-08 | 0.00390625 | 9 | 4 | 27 | 32 | 1 | 27 | 29 |
| 1.00E-09 | 0.00195313 | 10 | 4 | 30 | 35 | 1 | 30 | 32 |

Bits required = ceil(abs(log(range)/log(2)))

Bits required to represent Fractional Part for 10^-4 accuracy = ceil(abs(log(10^-4)/log(2))) = 14

1. **Interpolation Strategies**

We can simply store all the values in a memory location and read from it. However, storing 2^14 memory locations with 16-bit values will eat up all FPGA resources. Moreover, this does not serve any added advantage. What we propose to do is interpolate between 2 points and compute the intermediate values. There are different interpolation methods available like zero-order hold, first-order hold (piecewise linear), lagrangian interpolation (polynomial interpolation). We investigated all these on MATLAB to compare the quality of interpolation (maximum error and number of values it can interpolate) to the overhead computations involved. The main objective is to get an algorithm that is fast, efficient, and less resource expensive.

Table 2: Comparison of various interpolation schemes

|  |  |  |
| --- | --- | --- |
| Algorithm | Pros | Cons |
| Zero Order Hold | Very easy to implement | Does not offer good accuracy |
| First Order Hold | 1. Only one multiplication operation involved and some addition and comparison. 2. Relatively simple. 3. Offers better accuracy. 4. Can be pipelined | We are linearly interpolating between points. Accuracy will be higher than zero order hold but not very high. We will eventually have to store 100+ values to interpolate within the error limit. |
| Lagrangian Interpolation | Best Accuracy | Solving a 3rd or 5th order polynomial on FPGA will take up a lot of resources and the added accuracy is not very high as compared to First-order hold. |

As the first-order hold offers the best-optimized result and resource usage, we choose it for the design. We run multiple tests for a number of different values being interpolated. The number of values interpolated each time was taken as the power of 2. The results are shown below for 10^-4 accuracy taking 14 bits for the fractional part.

Table 3: Interleaving vs Memory required for storing with 14 bits for fractional part

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Interleave | Average error | Max error | LSB input bits to skip | Bytes to be stored |
| 1 | 0.00002999019538 | 0.00006103490486 | 0 | 143360 |
| 2 | 0.00003286090409 | 0.00009142969754 | 1 | 71680 |
| 4 | 0.00003635224616 | 0.0001061825791 | 2 | 35840 |
| 8 | 0.00003983166729 | 0.0001142323543 | 3 | 17920 |
| 16 | 0.00004327905094 | 0.0001176353302 | 4 | 8960 |
| 32 | 0.0000464276663 | 0.0001182602918 | 5 | 4480 |
| 64 | 0.00004946228777 | 0.0001201281114 | 6 | 2240 |
| 128 | 0.00005273905312 | 0.0001246543125 | 7 | 1120 |
| 256 | 0.00005834263479 | 0.0001338473667 | 8 | 560 |
| 512 | 0.00007526662861 | 0.0002114663001 | 9 | 280 |
| 1024 | 0.0001241714946 | 0.0004836462328 | 10 | 140 |

We can see that with the above-mentioned configuration we exceed 10^-4 max error limit at interleave 4. However, the average error is still within the acceptable range. This happens because the value of 14 bits is the minimum required value for 10^-4 acceptable error. To overcome this, we start by adding one more bit for the fractional part and then start coming down from it. The results are as follows:

Table 4:Interleaving vs Memory required for storing with 15 bits for fractional part

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Interleave | Average error | Max error | LSB input bits to skip | Bytes to be stored |
| 1 | 0.00001513785275 | 0.00003051732674 | 0 | 143360 |
| 2 | 0.00001656706037 | 0.00004569969386 | 1 | 71680 |
| 4 | 0.00001828143897 | 0.00005329966602 | 2 | 35840 |
| 8 | 0.00002004964801 | 0.00005707432296 | 3 | 17920 |
| 16 | 0.00002171578409 | 0.00005881592981 | 4 | 8960 |
| 32 | 0.0000231986359 | 0.00005993471475 | 5 | 4480 |
| 64 | 0.00002451962384 | 0.00006054450398 | 6 | 2240 |
| 128 | 0.00002606971713 | 0.00006091701885 | 7 | 1120 |
| 256 | 0.00002788188459 | 0.00006532446526 | 8 | 560 |
| 512 | 0.00003237681987 | 0.00008375917455 | 9 | 280 |
| 1024 | 0.00004668137582 | 0.0001513848248 | 10 | 140 |

Here just by adding one more bit for the representation we are able to bring up interleaving to 512 and bring down resource utilization exponentially. Interleaving 512 means there are 511 values to be interpolated between 2 samples stored in memory.

The following explains the memory requirement calculations with the above configuration and Interleave 512:

Saturation Address: 5x2^15 = 163840

19’b : 010,1000,0000,0000,0000

Linear Address: 0.0625x2^15 = 2048

19’b: 000,0000,1000,0000,0000

Since interleaving = 512, next address stored in memory is 2048+512 = 2560

19’b: 000,0000,1010,0000,0000

Hence we only use bit 18 to 10 of input as address space (considering LSB as bit number 1).

Address Span =

0,0000,0100 ----- 1,0100,0000

Values to store = Add\_max -Add\_min +1 = 317

Bits to store = 317x16 = 5072 bits

Or

Bytes to store = 634 bytes.

All the interleave values have been taken as multiples of 2 to substitute division by shifting operations at every place. A parameterized, pipeline model of the above configuration was made. First order interpolation was performed. The model took 4 cycles to perform processing. However, after the final model was ready, room for improvements in the design were realized.

## **Improving the Design**

It was realized that we have to store around 317 values in the memory according to the first design to get required accuracy. These values change very slowly after a point because the function starts to flat out after x=2. The figure below shows that most values are redundant towards the end.

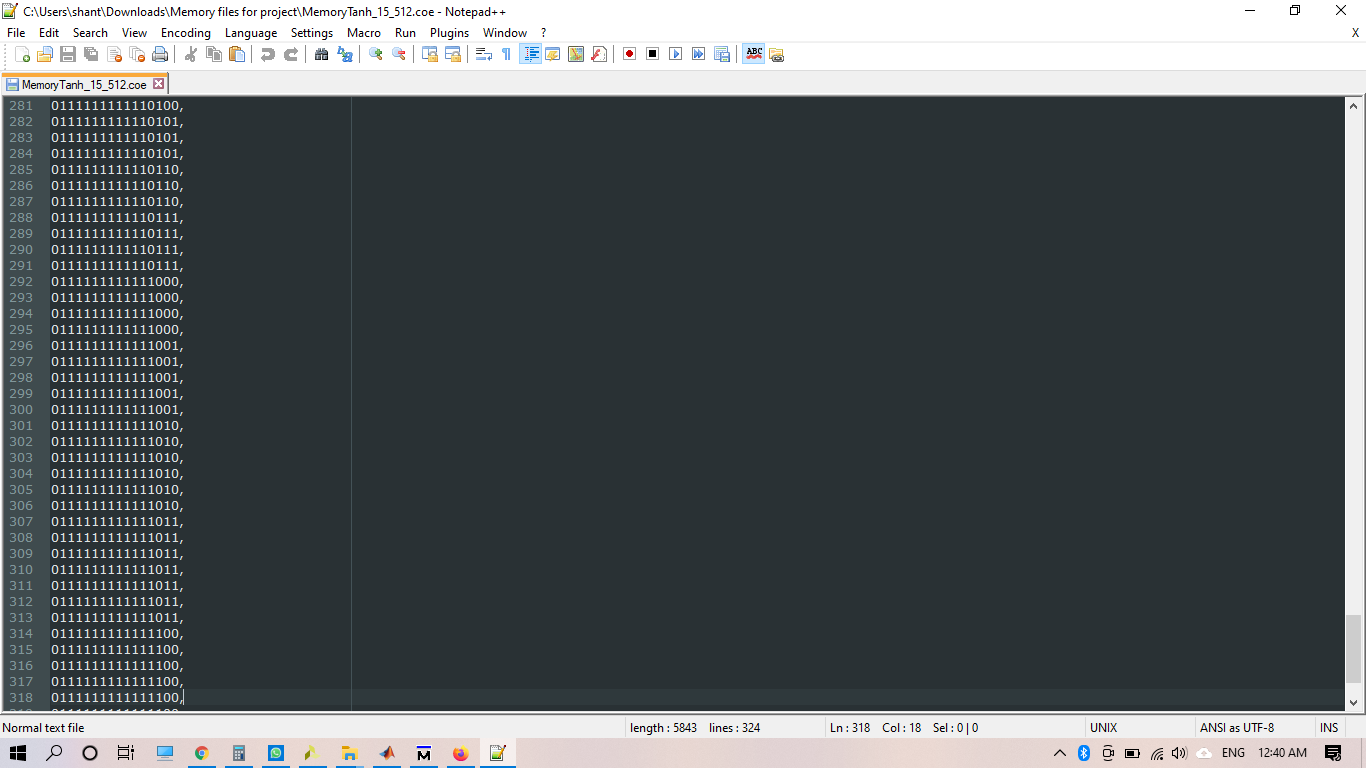


Figure 3: Redundancy in stored memory values

This led to the idea of using two interpolation values in two regions and remove the linear region. Hence, we use the same hardware multipliers by changing the inputs to interpolate between 2 regions. This should bring in less memory requirement at very less cost of added muxes and LUTs.

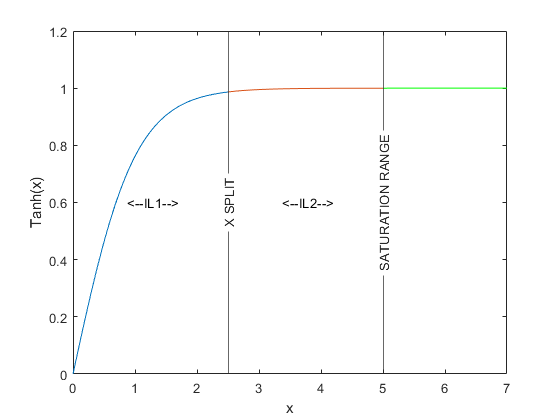


Figure 4: Splitting ranges for Model B

Finding the right size of memory required an optimization for first interleave value (IL1), second interleave value (IL2), the point to change from IL1 to IL2 and the number of bits taken to represent the fractional part (FB). We optimized it with the help of Genetic Algorithm Solver on MATLAB where fitness function was the Memory Requirement which had to be minimized. The constraint was that at any point error should be less than 10^-4. The point from which we change from IL1 to IL2 for interpolating is referred as X\_SPLIT.

After running this, we got the values as summarized in the table below.

Table 5: Best Individual Parameters

|  |  |
| --- | --- |
| Parameter | Value |
| IL1 | 14 |
| IL2 | 16 |
| FB | 19 |
| Total Memory required in bits | 1919 |

Best case memory requirement with previous model= 5072 bits.

With this implementation we get about 2.6 times less memory to store the values.

Moreover, if we increase required accuracy from 10e-4 to 10e-5 e, the additional memory required by the 2nd model will be much less than the 1st model. However, all current discussion is done with respect to 10e-4 target accuracy.

Moreover, earlier memory was read in 2 clock cycles for y1 and y2. But now we have split the memory in 2 banks of even and odd index. This will allow reading both y1 and y2 in same cycle and reduce pipeline stages.

Synthesis was done on Vivado for Kintex 7 xc7k70tfbg484-3. Various runs were performed and the table below will highlight the resource utilization.

Table 6: Summary of resource utilization by various models made

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Resource | With 1 interleave  (Model A) | With 2 interleave  (Model B-1) | With 2 interleave and memory IP  (Model B-2) |
| Without DSP Slice | Slice LUTs | 424 | 619 | 473 |
| Slice Registers | 146 | 107 | 119 |
| F7 Muxes | 36 | 0 | 0 |
| F8 Muxes | 4 | 0 | 0 |
| DSP Slice | 0 | 0 | 0 |
| With DSP Slice | Slice LUTs | 273 | 313 | 168 |
| Slice Registers | 146 | 107 | 119 |
| F7 Muxes | 36 | 0 | 0 |
| F8 Muxes | 4 | 0 | 0 |
| DSP Slice | 1 | 1 | 1 |

The IP used and referred in the table above the IP core for memory generated by Vivado for ROM memory synthesized out of LUT only. We can see that the IP is highly dense and optimized and reduces the usage of LUT drastically.

Using a DSP slice for multiplication reduces around 306 LUTs required in Model-B.

Using the IP for memory reduces around 145 LUTs required in Model B-2.

1. **Conclusion**

We were finally able to implement Tanh function on FPGA with minimum FPGA resources through proper optimization of parameters which delivered a maximum error less than 10^-4.

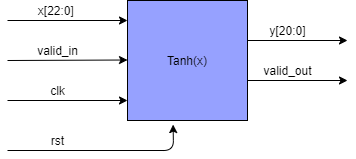


Figure 5: Abstract Black Box representation of the Final Design

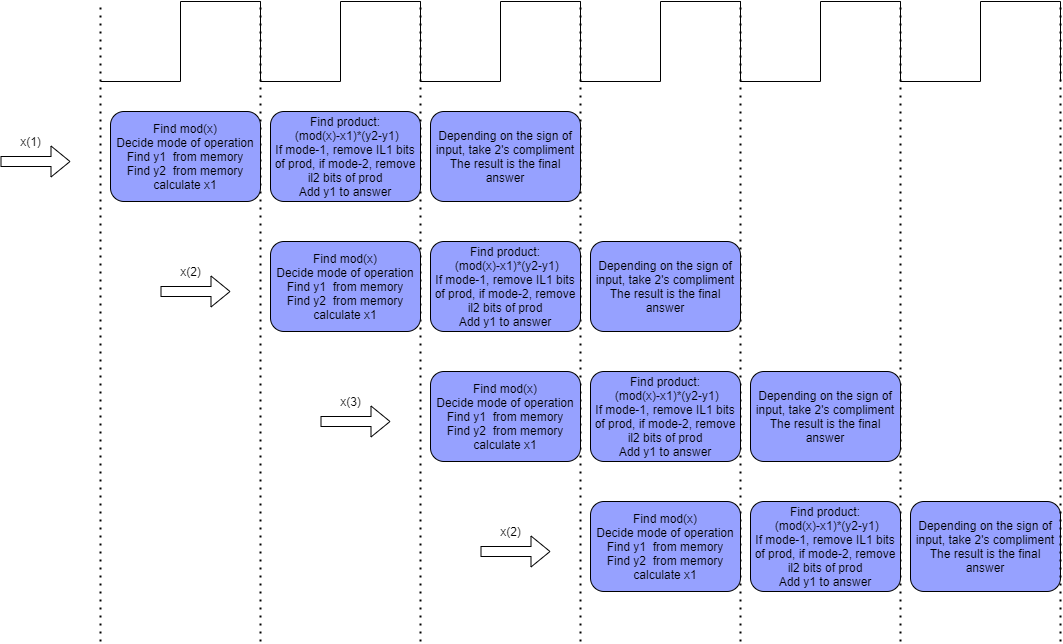


Figure 6: Timing Diagram of Model-B

1. **References**
   1. A. M. Abdelsalam, J. M. P. Langlois and F. Cheriet, "A Configurable FPGA Implementation of the Tanh Function Using DCT Interpolation," 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 168-171, doi: 10.1109/FCCM.2017.12.